

THE RUBY WAY, SECOND EDITION

HAL FULTON

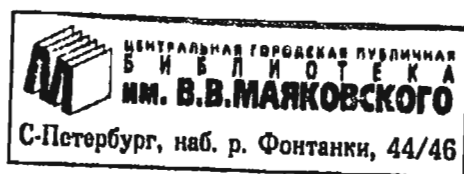
◆ Addison Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ RUBY

Хэл Фултон

854466



Москва, 2007

УДК 004.438
ББК 32.973.26-018.2

Фултон Х.

Ф94 Программирование на языке Ruby. – М.: ДМК Пресс, 2007. – 688 с.: ил.

ISBN 5-94074-357-9

Ruby – относительно новый объектно-ориентированный язык, разработанный Юкихио Мацумото в 1995 году и позаимствовавший некоторые особенности у языков LISP, Smalltalk, Perl, CLU и других. Язык активно развивается и применяется в самых разных областях: от системного администрирования до разработки сложных динамических сайтов.

Книга является полноценным руководством по Ruby – ее можно использовать и как учебник, и как справочник, и как сборник ответов на вопросы типа «как сделать то или иное в Ruby». В ней приведено свыше 400 примеров, разбитых по различным аспектам программирования, и к которым автор дает обстоятельные комментарии.

Издание предназначено для программистов самого широкого круга и самой разной квалификации, желающих научиться качественно и профессионально работать на Ruby.

УДК 004.438
ББК 32.973.26-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission from the publisher. For information on getting permission for reprints and excerpts, contact permission@peachpit.com. RUSSIAN language edition published by DMK PUBLISHERS, Copyright © 2007.

ISBN 0672328844
ISBN 5-94074-357-9

Copyright 2007 © Pearson Education, Inc.
© Оформление, ДМК Пресс, 2007

Содержание

Предисловие	12
Об авторе	17
Введение	18
О втором издании	18
Как организована эта книга	21
Об исходных текстах, приведенных в книге	23
«Путь Ruby»	24
Глава 1. Обзор Ruby	29
1.1. Введение в объектно-ориентированное программирование	30
1.2. Базовый синтаксис и семантика Ruby	35
1.3. ООП в Ruby	48
1.4. Динамические аспекты Ruby	57
1.5. Потренируйте свою интуицию: что следует запомнить	61
1.6. Жаргон Ruby	76
1.7. Заключение	79
Глава 2. Строки	80
2.1. Представление обычных строк	80
2.2. Альтернативная нотация для представления строк	81
2.3. Встроенные документы	83
2.4. Получение длины строки	83
2.5. Построчная обработка	84
2.6. Побайтовая обработка	84
2.7. Специализированное сравнение строк	84
2.8. Разбиение строки на лексемы	85
2.9. Форматирование строк	87
2.10. Строки в качестве объектов ввода/вывода	87
2.11. Управление регистром	88
2.12. Вычленение и замена подстрок	88
2.13. Подстановка в строках	90
2.14. Поиск в строке	91

2.15. Преобразование символов в коды ASCII и обратно	92
2.16. Явные и неявные преобразования	92
2.17. Дописывание в конец строки	94
2.18. Удаление хвостовых символов новой строки и прочих	94
2.19. Удаление лишних пропусков	95
2.20. Повтор строк	96
2.21. Включение выражений в строку	96
2.22. Отложенная интерполяция.....	96
2.23. Разбор данных, разделенных запятыми	97
2.24. Преобразование строки в число (десятичное или иное).....	98
2.25. Кодирование и декодирование строк в кодировке rot13.....	99
2.26. Шифрование строк	100
2.27. Сжатие строк	101
2.28. Подсчет числа символов в строке	101
2.29. Обращение строки.....	102
2.30. Удаление дубликатов	102
2.31. Удаление заданных символов	102
2.32. Печать специальных символов.....	102
2.33. Генерирование последовательности строк	103
2.34. Вычисление 32-разрядного CRC.....	103
2.35. Вычисление MD5-свертки строки	104
2.36. Вычисление расстояния Левенштейна между двумя строками	105
2.37. base64-кодирование и декодирование	106
2.38. Кодирование и декодирование строк (uuencode/uudecode).....	107
2.39. Замена символов табуляции пробелами и сворачивание пробелов в табуляторы	107
2.40. Цитирование текста.....	108
2.41. Заключение	109

Глава 3. Регулярные выражения 110

3.1. Синтаксис регулярных выражений.....	110
3.2. Компиляция регулярных выражений.....	112
3.3. Экранирование специальных символов	113
3.4. Якоря.....	113
3.5. Кванторы	114
3.6. Позитивное и негативное заглядывание вперед	116
3.7. Обратные ссылки.....	117
3.8. Классы символов.....	119
3.9. Обобщенные регулярные выражения	120

3.10. Сопоставление точки символу конца строки	121
3.11. Внутренние модификаторы	122
3.12. Внутренние подвыражения	122
3.13. Ruby и Oniguruma	123
3.14. Примеры регулярных выражений	129
3.15. Заключение	133
Глава 4. Интернационализация в Ruby	134
4.1. Исторические сведения и терминология	135
4.2. Кодировки в пост-ASCII мире	139
4.3. Справочники сообщений	150
4.4. Заключение	157
Глава 5. Численные методы	158
5.1. Представление чисел в языке Ruby	158
5.2. Основные операции над числами	159
5.3. Округление чисел с плавающей точкой	160
5.4. Сравнение чисел с плавающей точкой	162
5.5. Форматирование чисел для вывода	163
5.6. Вставка разделителей при форматировании чисел	163
5.7. Работа с очень большими числами	164
5.8. Использование класса BigDecimal	164
5.9. Работа с рациональными числами	166
5.10. Перемножение матриц	167
5.11. Комплексные числа	171
5.12. Библиотека <code>mathn</code>	172
5.13. Разложение на простые множители, вычисление НОД и НОК	172
5.14. Простые числа	173
5.15. Явные и неявные преобразования чисел	174
5.16. Приведение числовых значений	175
5.17. Поразрядные операции над числами	176
5.18. Преобразование системы счисления	177
5.19. Извлечение кубических корней, корней четвертой степени и т. д.	178
5.20. Определение порядка байтов	178
5.21. Численное вычисление определенного интеграла	179
5.22. Тригонометрия в градусах, радианах и градах	180
5.23. Неэлементарная тригонометрия	181
5.24. Вычисление логарифмов по произвольному основанию ...	182
5.25. Вычисление среднего, медианы и моды набора данных ...	182

5.26. Дисперсия и стандартное отклонение	183
5.27. Вычисление коэффициента корреляции	184
5.28. Генерирование случайных чисел	185
5.29. Кэширование функций с помощью метода memoize	186
5.30. Заключение	187
Глава 6. Символы и диапазоны	188
6.1. Символы.....	188
6.2. Диапазоны.....	192
6.3. Заключение	200
Глава 7. Дата и время	202
7.1. Определение текущего момента времени	203
7.2. Работа с конкретными датами (после точки отсчета)	203
7.3. Определение дня недели.....	204
7.4. Определение даты Пасхи.....	204
7.5. Вычисление n-ого дня недели в месяце	205
7.6. Преобразование из секунд в более крупные единицы.....	206
7.7. Вычисление промежутка времени, прошедшего от точки отсчета	207
7.8. Високосные секунды	207
7.9. Определение порядкового номера дня в году.....	208
7.10. Контроль даты и времени	208
7.11. Определение недели в году	209
7.12. Проверка года на високосность	210
7.13. Определение часового пояса.....	210
7.14. Манипулирование временем без даты.....	211
7.15. Сравнение моментов времени.....	211
7.16. Прибавление интервала к моменту времени.....	211
7.17. Вычисление разности между двумя моментами времени	212
7.18. Работа с конкретными датами (до точки отсчета).....	212
7.19. Взаимные преобразования объектов Date, Time и DateTime	213
7.20. Извлечение даты и времени из строки.....	214
7.21. Форматирование и печать даты и времени	215
7.22. Преобразование часовых поясов	216
7.23. Определение числа дней в месяце	216
7.24. Разбиение месяца на недели	216
7.25. Заключение	218

Глава 8. Массивы, хэши

и другие перечисляемые структуры.....	219
8.1. Массивы.....	219
8.2. Хэши.....	242
8.3. Перечисляемые структуры в общем	252
8.4. Заключение	259

Глава 9. Более сложные структуры данных

9.1. Множества	260
9.2. Стеки и очереди	263
9.3. Деревья.....	268
9.4. Графы.....	274
9.5. Заключение	280

Глава 10. Ввод/вывод и хранение данных

10.1. Файлы и каталоги	282
10.2. Доступ к данным более высокого уровня	306
10.3. Библиотека KirbyBase	314
10.4. Подключение к внешним базам данных.....	317
10.5. Заключение	329

Глава 11. ООП и динамические механизмы в Ruby

11.1. Рутинные объектно-ориентированные задачи	331
11.2. Более сложные механизмы.....	356
11.3. Динамические механизмы	375
11.4. Заключение	395

Глава 12. Графические интерфейсы для Ruby

12.1. Ruby/Tk.....	397
12.2. Ruby/GTK2	409
12.3. FXRuby (FOX).....	422
12.4. QtRuby	436
12.5. Другие библиотеки для создания графических интерфейсов	446
12.6. Заключение	447

Глава 13. Потoki в Ruby

13.1. Создание потоков и манипулирование ими.....	449
13.2. Синхронизация потоков.....	458
13.3. Заключение	473

Глава 14. Сценарии**и системное администрирование 474**

14.1. Запуск внешних программ	474
14.2. Флаги и аргументы в командной строке	479
14.3. Библиотека Shell.....	482
14.4. Переменные окружения	485
14.5. Сценарии на платформе Microsoft Windows.....	487
14.6. Моментальный инсталлятор для Windows	493
14.7. Библиотеки, о которых полезно знать	494
14.8. Работа с файлами, каталогами и деревьями	495
14.9. Различные сценарии.....	498
14.10. Заключение	502

Глава 15. Ruby и форматы данных 503

15.1. Разбор XML и REXML.....	503
15.2. RSS и Atom	508
15.3. Обработка изображений с помощью RMagick	512
15.4. Создание документов в формате PDF с помощью библиотеки PDF::Writer.....	521
15.5. Заключение	530

Глава 16. Тестирование и отладка 531

16.1. Библиотека Test::Unit	531
16.2. Комплект инструментов ZenTest	535
16.3. Работа с отладчиком Ruby	538
16.4. Использование irb в качестве отладчика	541
16.5. Измерение покрытия кода	542
16.6. Измерение производительности	543
16.7. Объекты печати	547
16.8. Заключение	548

Глава 17. Создание пакетов**и распространение программ 550**

17.1. Программа RDoc	550
17.2. Установка и подготовка пакета	555
17.3. RubyForge и RAA	559
17.4. Заключение	560

Глава 18. Сетевое программирование 561

18.1. Сетевые серверы.....	562
18.2. Сетевые клиенты	572
18.3. Заключение	591



Глава 19. Ruby и Web-приложения	592
19.1. Программирование CGI на Ruby	592
19.2. FastCGI	597
19.3. Ruby on Rails	599
19.4. Разработка Web-приложений с помощью Nitro.....	603
19.5. Введение в Wee	615
19.6. Разработка Web-приложений с помощью IOWA.....	617
19.7. Ruby и Web-сервер	622
19.8. Заключение	629
Глава 20. Распределенный Ruby.....	630
20.1. Обзор: библиотека drb.....	630
20.2. Пример: эмуляция биржевой ленты	633
20.3. Rinda: пространство кортежей в Ruby	636
20.4. Обнаружение сервисов в распределенном Ruby.....	640
20.5. Заключение	641
Глава 21. Инструменты разработки для Ruby	642
21.1. Система RubyGems.....	642
21.2. Программа Rake	644
21.3. Оболочка irb.....	647
21.4. Утилита ri	652
21.5. Поддержка со стороны редакторов.....	653
21.6. Интегрированные среды разработки	654
21.7. Заключение	656
Глава 22. Сообщество пользователей Ruby	657
22.1. Ресурсы в Web	657
22.2. Новостные группы и списки рассылки.....	658
22.3. Блоги и онлайн-журналы	658
22.4. Запросы на изменение Ruby	659
22.5. Каналы IRC	659
22.6. Конференции по Ruby.....	659
22.7. Локальные группы пользователей Ruby	660
22.8. Заключение	660
Алфавитный указатель	662

Предисловие

Предисловие ко второму изданию

В древнем Китае люди, в особенности философы, полагали, что под внешней оболочкой мира и любого существа скрыто нечто, чего нельзя ни объяснить, ни описать словами. Это нечто китайцы называли Дао, а японцы – До. На русский язык это слово можно перевести как Путь. Слово «до» входит в такие названия, как дзюдо, кендо, карате-до и айкидо. Это не просто боевые искусства, а целая философия и взгляд на жизнь.

Так и в языке программирования Ruby есть своя философия и способ мышления. Этот язык заставляет думать по-новому. Он помогает программистам получать удовольствие от своей работы. И не потому, что Ruby был создан в Японии, а потому, что программирование стало важной частью существования – по крайней мере, для некоторых людей, жизнь которых Ruby призван улучшить.

Как всегда, описать, что такое Дао, трудно. Я чувствую это, но не могу подобрать нужных слов даже на японском, моем родном языке. А вот один смельчак по имени Хэл Фултон попытался, и его первая попытка (первое издание этой книги) оказалась довольно удачной. Попытка номер два увенчалась еще лучшим результатом, чему немало способствовала помощь многих людей из сообщества пользователей Ruby. По мере того как Ruby набирает популярность (отчасти благодаря продукту Ruby on Rails), все важнее становится овладение секретами мастерства производительного программирования на этом языке. Надеюсь, что книга, которую вы держите в руках, поможет вам в решении этой задачи.

Удачной работы!
Юкихино «Мац» Мацумото
Август 2006 года, Япония

Предисловие к первому изданию

Вскоре после того как я впервые познакомился с компьютерами в начале 1980-х годов, меня заинтересовали языки программирования. С тех пор я буквально помешался на этой теме. Думаю, причина такого интереса в том, что языки программирования – это способ выражения мыслей. Они по сути своей предназначены для человека.

Но вопреки этому факту языки программирования почему-то всегда оказывались в большей степени машинно-ориентированными. Многие из них спроектированы с учетом удобства для компьютеров.

По мере того как компьютеры становятся мощнее и дешевле, ситуация постепенно меняется. Возьмем, к примеру, структурное программирование. Машины все равно, насколько хорошо структурирована программа: она просто исполняет ее команда за командой. Идеи структурного программирования обращены к людям, а не к машинам. То же относится и к объектно-ориентированному программированию.

Пришло время проектировать языки, удобные для людей!

В 1993 году я разговаривал со своим коллегой о сценарных языках, их выразительности и перспективах. Я считал, что программирование пойдет именно по этому пути и будет ориентироваться на человека. Но меня не вполне устраивали существующие языки, такие как Perl и Python. Я хотел видеть язык более мощный, чем Perl, и более объектно-ориентированный, чем Python. Найти идеальный вариант мне не удалось, поэтому остался один выход: изобрести свой собственный.

Ruby – не самый простой язык, но ведь и человеческая душа не проста. Ей равно нравятся простота и сложность. Она не приемлет ни слишком примитивных, ни чересчур заумных вещей. Она ищет равновесия.

Поэтому при проектировании ориентированного на человека языка – Ruby – я следовал принципу наименьшего удивления. Иными словами, работа шла под девизом: хорошо то, что не кажется мне странным. Вот почему я чувствую себя в своей родной стихии и испытываю радость, когда программирую на Ruby. А с момента выхода в свет первой версии в 1995 году многие программисты во всем мире разделили со мной эту радость.

Как всегда, хочу выразить величайшую благодарность всем членам сообщества, сложившегося вокруг Ruby. Они – причина его успеха.

Также я благодарен автору этой книги, Хэлу Фултону, за то, что он показал другим Путь Ruby. В книге объясняется философия, стоящая за языком Ruby. Это квинтэссенция моих мыслей и ощущений членов сообщества. Интересно, как Хэлу удалось прочитать мои мысли и раскрыть секрет Пути Ruby!.. Я никогда не встречался с ним лично, надеюсь, что скоро это все-таки произойдет.

В заключение хочу выразить надежду, что эта книга и сам язык Ruby помогут вам получить удовольствие и радость от программирования.

Юкиhiro «Мац» Мацумото
Сентябрь 2001, Япония

Благодарности

Благодарности за второе издание

Здравый смысл подсказывает, что второе издание требует в два раза меньше работы, чем первое. Но здравый смысл ошибается.

Хотя значительная часть текста книги перекочевала прямиком из первого издания, даже эту часть пришлось сильно править. К каждому предложению приходилось ставить вопрос: «Сохранилось ли в 2006 году то, что было верно в 2001-м?» И это только начало!

Короче говоря, я потратил много сотен часов на подготовку второго издания – примерно столько же, сколько ушло на первое. И тем не менее я «всего лишь автор».

Книга появляется на свет в результате усилий многих людей. Если говорить об издательстве, то я благодарен Дебре Вильямс Коули (Debra Williams Cauley), Сонглин Кю (Songlin Qiu) и Менди Фрэнк (Mandie Frank) за тяжелый труд и бесконечное терпение. Спасибо Джениль Бриз (Geneil Breeze) за неутомимое вычитывание и выпалывание сорняков из моего английского. Есть много других сотрудников, которых я не могу назвать, поскольку их работа проходила за кулисами и я никогда с ними не встречался.

За техническое редактирование отвечали главным образом Шашанк Дейт (Shashank Date) и Фрэнсис Хван (Francis Hwang). Они прекрасно справились со своей задачей – если остались какие-то ошибки, это всецело моя вина.

Большое спасибо людям, которые предлагали объяснения, писали код примеров и отвечали мне на многочисленные вопросы. Это сам Мац (Юкихио Мацумото), Дейв Томас (Dave Thomas), Кристиан Нойкирхен (Christian Neukirchen), Чед Фаулер (Chad Fowler), Дэниэл Береп (Daniel Berger), Армин Рерль (Armin Roehrl), Стефан Шмидль (Stefan Schmiedl), Джим Вайрих (Jim Weirich), Райан Дэвис (Ryan Davis), Дженни У. (Jenny W.), Джим Фриз (Jim Freeze), Лайл Джонсон (Lyle Johnson), Мартин Де Мелло (Martin DeMello), Март Лоуренс (Mart Lawrence), Рон Джеффрис (Ron Jeffries), Тим Хантер (Tim Hunter), Чет Хендриксон (Chet Hendrickson), Натаниэль Талбот (Nathaniel Talbott) и Бил Клеб (Bil Kleb).

Особая благодарность активным помощникам. Эндрю Джонсон (Andrew Johnson) здорово обогатил мои знания о регулярных выражениях. Пол Бэтли (Paul Battley) предоставил обширный материал для главы об интернационализации. Масао Муго (Masao Mutoh) помог в написании той же главы, а также снабдил меня материалами по GTK. Остин Зиглер (Austin Ziegler) научил меня секретам

подготовки PDF-файлов. Калев Теннис (Kaleb Tennis) дал материал по Qt. Эрик Ходел (Eric Hodel) помог в описании продуктов Rinda и Ring, а Джеймс Бритт (James Britt) внес большой вклад в главу о разработке приложений для Web.

И еще раз выражаю искреннюю благодарность и восхищение Мацу – не только за помощь, но и прежде всего – за создание Ruby. Domo arigato gozaimasu!

Еще раз хочу поблагодарить своих родителей. Они постоянно подбадривали меня и с нетерпением ждали выхода книги... Я еще сделаю из них программистов!

И опять – спасибо всем членам сообщества пользователей Ruby за неутомимую энергию и дух коллективизма. Особенно я благодарен читателям этой книги (обоих изданий). Я надеюсь, что вы найдете ее информативной, полезной и, быть может, даже увлекательной.

Благодарности за первое издание

Создание книги – плод усилий целого коллектива. Я не понимал этого в полной мере, пока не взялся за это дело сам. Рекомендую всем пройти подобное испытание, хотя оно и не из легких. Нет сомнений, что без помощи многих и многих людей книга не увидела бы свет.

Прежде всего, выражаю благодарность и восхищение Мацу (Юкихиро Мацумото), который создал язык Ruby. Domo arigato gozaimasu!

Спасибо Конраду Шнейкеру (Conrad Schneiker), который подал мне идею написать эту книгу и помог выработать ее общий план. Он же оказал мне неоценимую услугу, познакомив с языком Ruby в 1999 году.

Материалом для книги меня снабжали несколько человек. Первым хочу назвать Гая Хэрста (Guy Hurst), который написал значительную часть начальных глав, а также два приложения. Его помощь поистине бесценна.

Спасибо также другим помощникам, называя которых, я не придерживаюсь какого-то определенного порядка. Кэвин Смит (Kevin Smith) многое сделал для раздела главы 6, посвященного GTK, избавив меня от изучения сложной темы в условиях жесткого графика. Патрик Логан (Patrick Logan) пролил свет на графическую систему FOX GUI, описанную в той же главе. Чед Фаулер (Chad Fowler) в главе 9 углубился в тайны XML, а также помог при написании раздела о CGI.

Благодарю всех, кто правил корректуру, писал рецензии и помогал иными способами: Дона Мучоу (Don Muchow), Майка Стока (Mike Stok), Мико Огисима (Miho Ogishima) и уже упомянутых выше. Спасибо Дэвиду Эппштейну (David Eppstein), профессору математики, который ответил на вопросы по теории графов.

Одна из замечательных особенностей Ruby – поддержка со стороны сообщества пользователей. В списке рассылки и в конференции многие отвечали на мои вопросы, подавали идеи и всячески помогали. Опять же не придерживаясь определенного порядка, хочу упомянуть Дейва Томаса (Dave Thomas), Энди Ханта (Andy Hunt), Хи-Соб Парка (Hee-Sob Park), Майка Уилсона (Mike Wilson), Ави Брайанта (Avi Bryant), Ясуси Шоджи (Yasushi Shoji «Yashi»), Шюро Маэда (Shugo

* Огромное спасибо (яп.)

Maeda), Джима Вайриха (Jim Weirich) и Масаки Сукета (Masaki Suketa). Как это ни печально, но, скорее всего, кого-то я пропустил.

Очевидно, что книга никогда не вышла бы без помощи издателя. Многие работали над ней незримо для меня, но особенно я хотел бы поблагодарить Вильяма Брауна (William Brown), который тесно сотрудничал со мной и постоянно поощрял, и Скотта Мейера (Scott Meyer), который скрупулезно занимался объединением всех материалов. Других назвать не могу, потому что никогда о них не слышал. Но они знают себя сами.

Хочу поблагодарить своих родителей, которые следили за проектом издае-
ка, подбадривали меня и даже дали себе труд ради меня освоить азы программи-
рования.

Один мой друг как-то сказал: «Если ты написал книгу, которую никто не чита-
ет, значит, ты не написал ее вовсе». Поэтому напоследок я хочу поблагодарить чи-
тателей. Эта книга для вас. Надеюсь, что она окажется полезной.

Об авторе

Хэл Фултон – обладатель двух ученых степеней по информатике, полученных в Университете штата Миссисипи. Он четыре года преподавал информатику в колледже, пока не переехал в Остин, штат Техас, для работы по контрактам (в основном с отделением компании IBM в Остине). Более 15 лет он работал с различными версиями ОС UNIX, в том числе AIX, Solaris и Linux. С языком Ruby впервые познакомился в 1999 году, а в 2001 приступил к работе над первым изданием этой книги – второй книги на английском языке, посвященной Ruby. Фултон присутствовал на шести конференциях по Ruby и проводил презентации на четырех из них, в частности на первой европейской конференции по языку Ruby, состоявшейся в Карлсруэ (Германия). Сейчас Хэл работает в компании Broadwing Communications, располагающейся в Остине, и занимается вопросами, связанными с большим хранилищем данных и относящимися к нему телекоммуникационными приложениями. В работе он использует язык C++, СУБД Oracle и, конечно, Ruby.

Фултон по-прежнему постоянно присутствует в списке рассылки и в IRC-канале, посвященном Ruby, а также участвует в нескольких разрабатываемых проектах на Ruby. Он член Ассоциации по вычислительной технике (ACM – Association for Computing Machinery) и компьютерного общества IEEE (Институт инженеров по электротехнике и электронике). В свободное от работы время увлекается музыкой, чтением, искусством и фотографией. Кроме всего прочего, Хэл член общества по изучению Марса и энтузиаст космических полетов. Мечтал бы когда-нибудь совершить такой полет. Проживает в Остине, штат Техас.

Введение

.....
*Путь, о котором можно поведать, –
не постоянный Путь.
Лао Цзы, «Дао де цзин»**

Эта книга называется «Путь Ruby». Название нуждается в некотором пояснении.

Я ставил себе целью выразить в этой книге философию языка Ruby, насколько это в моих силах. Ту же цель преследовали мои добровольные помощники. Успех должно разделить между всеми, а ошибки остаются моей и только моей виной.

Конечно, я не могу абсолютно точно сказать, в чем же состоит истинный дух Ruby. Эту задачу я оставляю Мацу, но подозреваю, что даже ему трудно будет облечь свои ощущения в слова.

Короче говоря, «Путь Ruby» – всего лишь книга, а Путь Ruby – удел создателя языка и сообщества в целом. Втиснуть его в рамки книги довольно трудно. И все-таки в этом введении я попытаюсь хотя бы отчасти передать неуловимый дух Ruby. Мудрый ученик не воспримет эту попытку как окончательный вердикт!

Не забывайте, что это второе издание. Большая часть введения сохранена, но впереди раздел «О втором издании», в котором описываются изменения и вновь включенный материал.

О втором издании

Все меняется, и Ruby – не исключение. Я пишу это введение в 2006 году, когда с момента выхода первого издания прошло уже почти пять лет. Настало время для обновления.

В это издание внесено немало исправлений и добавлено много нового материала. Препятствия глава 4 («Простые задачи, связанные с обработкой данных») превратилась в шесть глав, две из которых («Диапазоны и символы» и «Интернационализация в Ruby») совсем новые; в четыре остальных добавлены новые примеры и комментарии к ним. Сильно расширен материал о регулярных выражениях: теперь рассматриваются не только «классические» выражения, но и новая библиотека для их поддержки, Oniguruma.

Главы 8 и 9 раньше составляли одну главу. Она была разбита на две, поскольку из-за вновь добавленного материала оказалась слишком большой.

* Пер. В. Малявина. – Прим. ред.

Аналогичным образом главы 18, 19 и 20 образовались в результате разделения главы 9. Приложения удалены, чтобы освободить место для основного материала.

Появились также следующие новые главы:

- глава 15, «Форматы данных в Ruby». Здесь рассматриваются форматы XML, RSS, графические файлы, создание PDF-файлов и другие вопросы;
- глава 16, «Тестирование и отладка». Речь идет о тестировании, профилировании, отладке, анализе кода и тому подобных вещах;
- в главе 17, «Создание пакетов и распространение программ», обсуждаются, в частности, инструмент `setup.rb` и создание пакетов в формате `RubyGem`;
- глава 21, «Инструменты разработки для Ruby», знакомит с поддержкой Ruby в редакторах и интегрированных средах разработки (IDE), утилитой `ri` и форматом `RubyGem` с точки зрения пользователя;
- в главе 22, «Сообщество Ruby», приводятся основные Web-сайты, списки рассылки, форумы, конференции, IRC-каналы по Ruby и прочие дополнительные сведения.

В широком смысле все главы в этой книге «новые». Каждую из них я подверг тщательной ревизии, внес сотни мелких и десятки крупных изменений. Был исключен материал, утративший актуальность и показавшийся мне не слишком важным. Я учел изменения в самом языке Ruby, а также добавил в каждую главу новые примеры и комментарии.

Возможно, вас интересует, что было добавлено в старые главы. Отвечаю: уже упомянутая библиотека регулярных выражений `Oniguruma`; математические библиотеки и классы, включая `BigDecimal`, `mathn` и `matrix`; такие новые классы, как `Set` и `DateTime`.

В главу 10, «Ввод/вывод и хранение данных», я добавил материал о методе `readpartial`, неблокирующем вводе/выводе и классе `StringIO`. Также рассмотрены форматы CSV, YAML и `KirbyBase`. В ту часть главы 10, которая посвящена базам данных, включены сведения о СУБД Oracle и SQLite, интерфейсе DBI, а также об объектно-реляционном отображении (Object-Relational Mappers – ORM).

Глава 11, «ООП и динамические механизмы в Ruby», пополнилась информацией о таких недавно добавленных в язык конструкциях, как `initialize_copy`, `const_get`, `const_missing` и `define_method`. Также я рассматриваю делегирование и перенаправление.

Глава 12, «Графические интерфейсы для Ruby», была переработана целиком (в особенности разделы, посвященные GTK и Fox). Раздел по QtRuby – новый от начала до конца.

В главе 14, «Сценарии и системное администрирование», теперь обсуждаются моментальный инсталлятор для Windows и ряд аналогичных пакетов. Кроме того, улучшен код примеров.

В главе 18, «Сетевое программирование», появились разделы о вложениях в электронные письма и о взаимодействии с IMAP-сервером. Также рассматривается библиотека `OpenURI`.

В главе 19, «Ruby и Web-приложения», теперь рассматриваются продукты Ruby on Rails, Nitro, Wee, IOWA и другие инструменты для Web. Также уделено внимание инструментам WEBrick и в какой-то мере Mongrel.

В главу 20, «Распределенный Ruby», добавлен материал о системе Rinda – реализации пространства кортежей, написанной на Ruby. Тут же приводятся сведения о родственной системе Ring.

Так ли необходимы все эти добавления? Да, уверяю вас.

Напомню, кстати, что «Путь Ruby» – это вторая вышедшая на английском языке книга по языку Ruby; первой была знаменитая «Мотыга», или «Programming Ruby», Дэйва Томаса (Dave Thomas) и Энди Ханта (Andy Hunt). Моя книга была составлена так, чтобы не перекрывать, а дополнять свою предшественницу; это и стало одной из основных причин ее популярности.

Когда я приступал к работе над первым изданием, еще не было международных конференций по Ruby. Не было сайтов RubyForge, ruby-doc.org, не было Wiki-страницы rubygarden.org. Вообще в Сети не было почти ничего, кроме официального сайта Ruby. В архиве приложений Ruby насчитывалось всего несколько сотен программ.

В то время лишь немногие периодические издания (как обычные, так и онлайн-новые) знали о существовании этого языка. Когда где-то публиковалась статья о Ruby, мы все брали ее на заметку; о ней сообщалось в списке рассылки и там же проводилось обсуждение.

Многих привычных сегодня инструментов и библиотек еще не существовало. Все пока было впереди: и система RDoc, и пакет REXML для анализа XML-документов. Математическая библиотека была куда беднее нынешней. Поддержка баз данных была фрагментарной, а драйверы ODBC и вовсе отсутствовали. Tk был чуть ли не единственным графическим интерфейсом. Приложения для Web разрабатывались в виде низкоуровневых CGI-сценариев.

Еще не появился «моментальный» инсталлятор для Windows. Пользователям Windows приходилось выполнять компиляцию исходных текстов в среде Cygwin или с помощью minigw.

Системы RubyGem не было даже в примитивной форме. Процедура поиска и установки приложений проводилась вручную; для решения этой задачи использовались инструменты типа tar и make.

Никто слыхом не слыхал о Ruby on Rails. Никто (насколько мне известно) не употреблял термина «утилитизация»^{*}. Не было ни YAML для Ruby, ни системы Rake.

В то время мы пользовались версией Ruby 1.6.4 и считали ее безмерно крутой. Но Ruby 1.8.5 (с которой я обычно работаю сейчас) еще круче!

Был незначительно изменен синтаксис, но не настолько серьезно, чтобы об этом писать. По большей части речь идет о «границных случаях», и теперь синтаксис в этих ситуациях выглядит более разумно. Ruby всегда отличали странности в отношении к необязательности скобок; в 98% случаев вы никакой раз-

^{*} Речь идет о переводе выражения «duck typing». Его смысл и происхождение объясняются в разделе 1.6. Приному названию ревизионизма чистоты русского языка за то, что не смог удержаться от игры слов и выдумал этот «термин» вместо «утиной типизации». – *Прим. перев.*

ницы не заметите, а если заметите, то, наверное, согласитесь, что язык стал более последовательным.

Изменилась семантика некоторых системных методов. Но несущественно. Например, раньше метод `Dir#chdir` не принимал в качестве параметра блок, но несколько лет назад это стало допустимо.

Некоторые системные методы объявлены устаревшими или переименованы. Метод `class` утратил свой псевдоним `type` (поскольку в Ruby мы обычно не говорим о типах объектов). Метод `intern` получил более понятное название `to_sym`, а метод `Array#indices` называется `Array#values_at`. Можно было бы продолжить, но думаю, что суть вы уловили.

Кроме того, было добавлено несколько новых системных методов, например `Enumerable#inject`, `Enumerable#zip` и `IO#readpartial`. Старая библиотека `futils` теперь называется `fileutils`, и у нее появилось собственное пространство имен `FileUtil`, тогда как раньше она добавляла методы в класс `File`.

Есть и еще много изменений. Но важно понимать, что все они вносились очень осторожно и аккуратно. Язык как был Ruby, так им и остался. Красота Ruby в немалой степени обязана тому факту, что он изменяется медленно и обдуманно, ведомый мудростью Маца и других разработчиков.

Сегодня нет недостатка в книгах по Ruby. Публикуется больше статей, чем мы в состоянии переварить. Множатся руководства и документация в сети Web.

Появились новые инструменты и библиотеки. По разным причинам большая их часть — это каркасы и инструменты для разработки Web-приложений, средства для создания сетевых дневников (блогов), разметки, а также для объектно-реляционного отображения (ORM). Но есть и инструментарий для работы с базами данных, организации графических интерфейсов, математических расчетов, Web-сервисов, обработки изображений, управления версиями и т.д.

Поддержка Ruby в редакторах широко распространена и достигла немалой изощренности. Существуют интегрированные среды разработки (IDE), весьма полезные и зрелые; частично они перекрываются с конструкторами графических интерфейсов.

Нет сомнений и в том, что сообщество пользователей разрослось и изменилось. Сегодня Ruby никак не назовешь нишевым языком: им пользуются в НАСА, Национальной администрации по океану и атмосфере (NOAA), компании Motorola и во многих других крупных организациях. Он применяется для работы с графикой, доступа к базам данных, численного анализа, Web-приложений и в других областях. Короче говоря, Ruby стал весьма популярным языком.

Я работал над новой редакцией этой книги с любовью. Надеюсь, что она окажется вам полезной.

Как организована эта книга

Вряд ли вы станете изучать Ruby по этой книге. В ней не так уж много вводного и учебного материала. Если вы еще ничего не знаете о Ruby, то лучше начать с какой-нибудь другой книги.

Но программисты – народ упорный, и я допускаю, что научиться Ruby *только* по этой книге возможно. В главе 1, «Обзор Ruby», приводится краткое введение в язык и очень скромное руководство.

Также в главе 1 есть довольно полный перечень «скользких мест» (который трудно поддерживать в актуальном состоянии). Для разных читателей этот перечень полезен в разной мере, поскольку что для одного интуитивно очевидно, для другого выглядит странно.

В основном эта книга призвана отвечать на вопросы типа «Как сделать?». И потому вы, вероятно, многое будете пропускать. Я почти за честь, если кто-то прочтет книгу от корки до корки, но не надеюсь на это. Скорее я ожидаю, что вы будете искать в оглавлении темы, которые вас интересуют в конкретный момент. Впрочем, с момента выхода первого издания мне приходилось беседовать с разными людьми, и оказалось, что многие прочли книгу целиком. Более того, несколько человек писали мне, что выучили по ней Ruby. Что ж, все возможно!..

Некоторые рассматриваемые в книге вопросы могут показаться элементарными. Но ведь у разных людей и опыт разный; то, что очевидно одному, будет откровением для другого. Я старался сделать изложение как можно более полным. С другой стороны, было стремление уложиться в разумный объем (ясно, что эти цели противоречивы).

Можно назвать эту книгу «справочником наоборот». Вы ищете то, что нужно, не по имени класса или метода, а по функции или назначению. Например, в классе `String` есть несколько методов для манипулирования регистром букв: `capitalize`, `upcase`, `casecmp`, `downcase` и `swapcase`. В настоящем справочнике они встречались бы в алфавитном порядке, а в этой книге собраны в одном месте.

Конечно, в борьбе за полноту охвата материала я иногда сворачивал на путь, которому следуют справочные руководства. Во многих случаях я старался компенсировать это, предлагая не совсем обычные примеры или разнообразя их по сравнению со справочниками.

Я старался не перегружать код комментариями. Если не считать первой главы, то думаю, что достиг этой цели. Писатель может стать не в меру болтливым, но программист-то хочет видеть код (а если не хочет, то должен хотеть).

Иногда примеры выглядят искусственными, за что я приношу свои извинения. Проиллюстрировать какой-то прием или принцип в отрыве от реальной задачи бывает сложно. Но чем сложнее задача, чем выше ее уровень, тем большие усилия я прилагал к подысканию реального примера. Так, если речь идет о конкатенации строк, то, наверное, вы увидите безыскусный фрагмент кода с упоминанием пресловутых “foo” и “bar”, но когда рассматривается тема разбора XML-документа, будет приведен куда более содержательный и реалистичный пример.

Есть в этой книге два-три каприза, в которых хочу заранее сознаться. Во-первых, я всеми силами старался избегать «уродливых» глобальных переменных типа `$_` и ей подобных, пришедших из языка Perl. Они есть в Ruby и прекрасно работают, даже применяются в повседневной работе всеми или большинством программистов. Но почти всегда от их использования можно уйти, что я и позволил себе чуть ли не во всех примерах.

Другой каприз состоит в том, что я избегаю пользоваться обособленными выражениями, если у них нет побочных эффектов. В Ruby выражения – одна из основ языка, и это прекрасно; я старался извлечь из этой особенности максимум пользы. Но во фрагментах кода предпочитаю не употреблять выражения, которые просто возвращают никак не используемое значение. Например, для иллюстрации конкатенации строк достаточно было бы написать "abc" + "def", но я в этом случае пишу что-то вроде `str = "abc" + "def"`. Кому-то это покажется излишеством, но выглядит естественным для программиста на языке C, привыкшего к тому, что бывают функции типа `void` и `non-void` (а также программисту на Pascal, мыслящему в терминах процедур и функций).

Третий каприз заключается в моем нежелании употреблять символ решетки для обозначения методов экземпляра. Многие поклонники Ruby считают, что я проявляю излишнюю болтливость, когда пишу «метод экземпляра `crypt` класса `String`», а не просто `String#crypt`, но я полагаю, что так никто не запутается. (На самом деле мне придется постепенно смириться с использованием такой нотации, так как ясно, что она уже никуда не исчезнет.)

Я старался давать ссылки на внешние ресурсы там, где это уместно. Ограничения по времени и объему не позволили мне включить в книгу все, что я хотел бы, но надеюсь, что это хотя бы отчасти компенсируется указаниями на то, где найти недостающую информацию. Из всех источников самым главным, наверное, следует считать архив приложений Ruby (Ruby Application Archive) в сети: вы не раз встретите ссылки на него.

В начале книги принято приводить соглашения об использовании шрифтов, применяемых для выделения кода, и о том, как отличить пример от обычного текста. Но я не стану оскорблять вас недоверием к вашим умственным способностям, – вы ведь и раньше читали техническую литературу.

Хочу подчеркнуть, что примерно 10% текста книги было написано другими людьми. И это не считая технического редактирования и корректуры!.. Вы просто обязаны прочитать благодарности, приведенные в этой (и любой другой) книге. Большинство читателей пропускают их. Прошу, прочтите прямо сейчас. Это будет так же полезно, как питание овощами.

Об исходных текстах, приведенных в книге

Все сколько-нибудь значительные фрагменты кода собраны в архив, который можно загрузить из сети. Этот архив есть на сайте www.awprofessional.com и на моем собственном сайте (www.rubyhacker.com).

Он предлагается в виде `tgz`-файла и в виде `zip`-файла. При именовании файлов в нем принято следующее соглашение: код, которому в тексте соответствует пронумерованный листинг, находится в файле с таким же именем (например, `listing14-1.rb`). Более короткие фрагменты именуются по номеру страницы, возможно, с добавленной буквой (например, `p260a.rb` и `p260b.rb`). Совсем короткие фрагменты, которые нельзя исполнить «вне контекста», в архиве обычно отсутствуют.

«Путь Ruby»

Что мы имеем в виду, говоря о Пути Ruby? Я полагаю, что тут есть два взаимосвязанных аспекта: философия проектирования Ruby и философия использования этого языка. Естественно, что дизайн и применение связаны друг с другом, будь то программное или аппаратное обеспечение. Иначе зачем бы существовала наука эргономика?.. Если я снабжаю устройство ручкой, то, наверное, предполагаю, что кто-то за эту ручку возьмется.

В языке Ruby имеется не выразимое словами качество, которое делает его тем, что он есть. Мы наблюдаем это качество в дизайне синтаксиса и семантики языка, присутствует оно и в написанных на нем программах. Все же стоит попытаться сформулировать, в чем состоит эта отличительная особенность.

Очевидно, Ruby – не просто инструмент для написания программ, но и сам по себе является программой. Почему работа программ, написанных на Ruby, должна следовать законам, отличным от тех, которым подчинена работа интерпретатора? В конце концов, Ruby – исключительно динамичный и расширяемый язык. Могут найтись причины, по которым эти два уровня где-то расходятся, вероятно, стараясь приспособиться к несовершенству реального мира. Но в общем случае мыслительные процессы могут и должны быть сходными. Интерпретатор Ruby можно было бы написать на самом Ruby, в полном соответствии с принципом Хофштадтера, хотя в настоящее время это еще не сделано.

Мы печасто задумываемся над этимологией слова «путь», но оно употребляется в двух разных смыслах. Во-первых, это метод или техника, а во-вторых – дорога. Ясно, что оба значения взаимосвязаны, и, говоря «путь Ruby», я имею в виду и то и другое.

Следовательно, мы говорим о мыслительном процессе, но вместе с тем и о дороге, по которой движемся. Даже величайшие гуру программирования не могут сказать о себе, что достигли совершенства: они лишь на пути к нему. Таких путей может быть несколько, но я здесь говорю только об одном.

Привычная мудрость гласит, что форма определяется функцией. Это верно, спору нет. Однако Фрэнк Ллойд Райт* (имея в виду свою собственную область интересов) как-то сказал: «Форма определяется функцией, которая была понята неправильно. Форма и функция должны быть едины, сливаться в духовном единении».

Что Райт имел в виду? Я бы сказал, что на этот вопрос вы найдете ответ не в книгах, а в собственном опыте.

Однако я думаю, что Райт выразил эту мысль где-то еще, разбив ее на части, которые проще переварить. Он был великим поборником простоты, который однажды заметил: «Самые полезные инструменты архитектора – это ластик рядом с чертежной доской и гвоздодер на строительной площадке».

Итак, одним из достоинств Ruby является простота. Надо ли цитировать других мыслителей, высказывавшихся на эту тему? Согласно Антуану де Сент-Экзюпери,

* Фрэнк Ллойд Райт (1867–1959) – знаменитый архитектор и дизайнер. Одна из самых известных работ – Музей Гугенхайма в Нью-Йорке. (Прим. перев.)

«совершенство достигнуто не тогда, когда нечего добавить, а тогда, когда нечего убрать».

Но Ruby – сложный язык. Почему же я называю его простым?

Если бы мы лучше понимали мироздание, то, наверное, открыли бы «закон сохранения сложности» – факт, который вменяется в нашу жизнь подобно энтропии, которую мы не можем преодолеть, а способны лишь рассевать.

И в этом ключ. Нельзя избежать сложности, но можно укрыться от нее. Мы можем убрать ее из виду! Это тот же старый добрый принцип черного ящика, внутри которого решается сложная задача, хотя на поверхности лишь голая простота.

Если вам еще не наскучили цитаты, то будет уместно привести слова Альберта Эйнштейна: «Все должно быть просто настолько, насколько возможно, но не проще».

Таким образом, на взгляд программиста, Ruby – это воплощенная простота (хотя у человека, отвечающего за сопровождение интерпретатора, взгляд может быть иной). Но вместе с тем имеется пространство для компромиссов. В реальном мире всем нам приходится немного «прогибаться». К примеру, все сущности в программе на Ruby должны были бы быть истинными объектами, однако некоторые, в том числе целые числа, хранятся как непосредственные значения. Это компромисс, знакомый всем студентам отделений информатики уже много десятилетий: элегантность дизайна приносится в жертву практической реализации. По существу, мы променяли одну простоту на другую.

То, что Ларри Уолл говорил о языке Perl, остается справедливым: «Когда вы хотите что-то выразить на маленьком языке, оно становится большим. А когда вы пытаетесь выразить то же самое на большом языке, оно становится маленьким». Это верно и в отношении английского языка. Если биолог Эрнст Хэккель смог всего тремя словами выразить глубокую мысль «онтогенез повторяет филогенез», то лишь потому, что эти слова с весьма специфическим смыслом были в его распоряжении. Мы соглашаемся на внутреннюю сложность языка, потому что она позволяет избежать сложности в отдельных высказываниях.

Переформулирую этот принцип по-другому: «не пишите 200 строк кода, когда достаточно 10».

Я считаю само собой разумеющимся, что краткость в общем случае хороша. Короткий фрагмент кода занимает меньше места в мозгу программиста, его проще воспринять как единое целое. Благоприятным побочным эффектом следует считать и то, что в короткой программе будет меньше ошибок.

Конечно, не нужно забывать предупреждение Эйнштейна о простоте. Если расположить краткость слишком высоко в списке приоритетов, то получится совершенно загадочный код. Согласно теории информации, сжатые данные статистически похожи на белый шум. Если вы видели код на C или APL либо регулярное выражение – особенно плохо написанные, то понимаете, что я имею в виду. «Просто, но не слишком просто» – это ключевая фраза. Стремитесь к краткости, но не жертвуйте понятностью.

Трюизмом следует считать мысль о том, что краткость в сочетании с понятностью – это хорошо. Но тому есть причина, причем настолько фундаментальная, что мы часто о ней забываем. А состоит она в том, что компьютер создан для человека, а не человек для компьютера.

В старые добрые дни все было почти наоборот. Компьютеры стоили миллионы долларов и пожирали многие киловатты электричества. Люди же вели себя так, будто компьютер – божество, а программисты – его скромные жрецы. Час машинного времени стоил дороже часа личного времени.

Когда компьютеры стали меньше и дешевле, приобрели популярность языки высокого уровня. Они неэффективны с точки зрения машины, зато эффективны с позиции человека. Ruby – всего лишь одно из последних достижений на этом пути. Некоторые даже называют его языком сверхвысокого уровня (VHLL – Very High-Level Language). Хотя этот термин еще не получил четкого определения, я думаю, что он оправдан.

Компьютер призван быть слугой, а не хозяином, а, как сказал Мац, толковый слуга должен выполнять сложное задание при минимуме указаний. Так было на протяжении всей истории информатики. Мы начали с машинного языка, перешли к языку ассемблера, а потом добрались и до языков высокого уровня.

Мы сейчас говорим о смещении парадигмы: от машиноцентрической к человекоцентрической. На мой взгляд, Ruby дает великолепный пример человекоцентрического программирования.

Теперь я хочу взглянуть на вопрос под несколько иным углом. В 1980 году вышла чудесная книжка Джеффри Джеймса «Дао программирования» (Geoffrey James, The Tao of Programming). Каждая строчка из нее достойна цитирования, но я ограничусь лишь одной выдержкой: «Программа должна следовать “закону наименьшего удивления”. Что это за закон? Все просто: программа должна отвечать пользователю так, чтобы вызывать у него как можно меньше удивления». (Конечно, если речь идет об интерпретаторе языка, то пользователем является программист.)

Не знаю, Джеймс ли придумал термин «закон наименьшего удивления», но я впервые узнал его из упомянутой книги. Этот закон хорошо известен и часто цитируется в сообществе пользователей Ruby. Правда, обычно его называют «принципом наименьшего удивления» (Principle of Least Surprise, POLS). Лично я упрямо придерживаюсь акронима LOLA – Law of Least Astonishment.

Но, как ни называй, правило остается справедливым и служит основополагающим принципом продолжающейся работы над языком Ruby. О нем полезно помнить и тем, кто разрабатывает библиотеки и пользовательские интерфейсы.

Конечно, одна проблема остается: разные люди удивляются разным вещам; не существует всеобщего согласия о том, как «должен» вести себя объект или метод. Но мы можем стремиться быть последовательными и пахотить веские обоснования принимаемым проектным решениям, а каждый человек должен тренировать собственную интуицию.

Кстати, Мац как-то заметил, что «принцип наименьшего удивления» должен относиться и к нему как к дизайнеру. Чем больше ваше мышление походит на его,

тем меньше удивления будет вызывать Ruby. И смею уверить, подражание Мацу большинству из нас пойдет только на пользу.

Какой бы ни была логическая конструкция системы, тренировать свою интуицию необходимо. Каждый язык программирования – это отдельный мир со своими допущениями, точно так же как и любой естественный язык. Когда я начал изучать немецкий, то обнаружил, что все существительные пишутся с прописной буквы... за исключением слова *deutsch* (немецкий язык). Я пожаловался профессору, подчеркивая, что ведь это же название самого языка. Он улыбнулся и ответил: «Не надо с этим бороться».

Профессор говорил, что надо позволить немцу оставаться немцем. Продолжая эту мысль, хочу дать совет всем, кто переходит на использование Ruby после освоения других языков. Пусть Ruby остается Ruby! Не ожидайте, что это будет Perl. Не требуйте от него поведения, характерного для языков LISP или Smalltalk. С другой стороны, у Ruby есть элементы, присущие любому из этих трех языков. Для начала действуйте в соответствии с априорными представлениями, но когда они оказываются неверны, не боритесь с установленными правилами (если только Мац не согласится с тем, что в них необходимо внести изменения).

Каждый программист сегодня знает о принципе ортогональности (хотя лучше было бы назвать его принципом ортогональной полноты). Вообразим пару осей, по одной из которых откладываются языковые сущности, а по другой – множество атрибутов и возможностей. Когда мы говорим об ортогональности, то обычно имеем в виду, что пространство, определяемое этими осями, настолько «полно», насколько это логически возможно.

Одна из составных частей Пути Ruby – стремление к ортогональности. Масив в некоторых отношениях подобен хэшу, а потому и операции над ними должны быть похожи. До тех пор пока мы не вступаем в область, где эти сущности начинают отличаться друг от друга.

Мац говорит, что «естественность» важнее ортогональности. Но чтобы понять, что естественно, а что нет, надо долго думать и писать программы.

Ruby стремится быть дружелюбным к программисту. Например, у многих методов есть синонимы; оба метода *size* и *length* возвращают число элементов в масиве. Два разных написания слова – *indexes* и *indices* – относятся к имени одного и того же метода. Некоторые называют это досадным недоразумением, но я склонен считать такую избыточность хорошим дизайном.

Ruby стремится к последовательности и единообразию. В этом нет ничего мистического: во всех жизненных ситуациях мы жаждем регулярности и размеренности. Сложнее научиться понимать, когда этот принцип следует нарушить.

Например, в Ruby принято добавлять вопросительный знак (?) в конец имени метода, ведущего себя как предикат. Это хорошо и удобно, программа становится яснее, а в пространстве имен легче ориентироваться. Но менее последовательным является аналогичное употребление восклицательного знака для обозначения потенциально «деструктивных» или «опасных» методов (в том смысле, что они модифицируют внутреннее состояние вызывающего объекта). Непоследовательность

состоит в том, что не все деструктивные методы помечаются таким образом. Нужно ли восстановить справедливость?

Нет, на самом деле не нужно. Некоторые методы по сути своей изменяют состояние (например, методы `replace` и `concat` класса `Array`). Одни являются «методами установки», которые допускают присваивание атрибуту класса; ясно, что не следует добавлять восклицательный знак к имени атрибута или к знаку равенства. Другие в каком-то смысле изменяют состояние объекта, например `read`, но это происходит так часто, что нет смысла особо отмечать данный факт. Если бы имя каждого деструктивного метода заканчивалось символом `!`, то программа превратилась бы в рекламную брошюру фирмы, занимающейся многоуровневым маркетингом.

Вы замечаете действие разнонаправленных сил, тенденцию нарушать все правила? Тогда позвольте мне сформулировать второй закон Фултона: «У каждого правила есть исключения, кроме второго закона Фултона». (Доля шутки тут есть, но небольшая.)

В Ruby мы видим не «педантичную непротиворечивость», а строгое следование набору простых правил. Может быть, отчасти Путь Ruby состоит в том, что его подход не является застывшим и неподвижным. Мац как-то сказал, что при проектировании языка нужно «следовать велениям своего сердца». И еще один аспект философии Ruby: «Не бойтесь изменений во время выполнения, не бойтесь быть динамичными». Мир динамичен, так почему язык программирования должен быть статичным? Ruby – один из самых динамичных среди существующих языков.

С некоторыми оговорками я бы выделил и такой аспект: «Не будьте рабом производительности». Если производительность оказывается недопустимо низкой, проблему придется решать, но не следует с самого начала выводить ее на первый план. Предпочитайте элегантность эффективности в тех случаях, когда эффективность не слишком критична. Впрочем, когда вы пишете библиотеку, которая будет использоваться непредвиденными способами, о производительности следует задуматься с самого начала.

Когда я смотрю на язык Ruby, то вижу равновесие между разными проектными целями, вижу сложное взаимодействие, напоминающее о задаче *n* тел в физике. Я могу представить себе, что он моделировался как мобил Александра Кальдера. Быть может, больше всего завораживает само взаимодействие, гармония, лежащая в основе философии Ruby, а не отдельные составные части. Программисты знают, что их ремесло – не просто сплав науки и технологии, но еще и искусство. Мне неловко говорить, что в компьютерных дисциплинах есть какой-то духовный аспект, но – строго между нами! – он безусловно присутствует. (Если вы не читали книгу Роберта Пирсига «Дзен и искусство ухода за мотоциклом» (Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*), горячо рекомендую.)

Источником Ruby стала человеческая потребность создавать полезные и красивые вещи. Программы, написанные на Ruby, должны проистекать из того же боговдохновенного источника. Это, на мой взгляд, и является квинтэссенцией Пути Ruby.

Глава 1. Обзор Ruby

.....
*Язык формирует способ нашего мышления
и определяет то, о чем мы можем размышлять.*
Бенджамин Ди Уорф

Стоит напомнить, что в новом языке программирования иногда видят напанаею, особенно его адепты. Но ни один язык не сможет заменить все остальные. Не существует инструмента, безусловно пригодного для решения любой мыслимой задачи. Есть много предметных областей и много ограничений, налагаемых решаемыми в них задачами.

А самое главное – есть разные пути обдумывания задач, и это следствие разного опыта и личных качеств самих программистов. Поэтому в обозримой перспективе будут появляться все новые и новые языки. А пока есть много языков, будет много людей, которые их критикуют и защищают. Короче говоря, «языковым войнам» конца не предвидится, но мы в этой книге не станем принимать в них участия.

И тем не менее в постоянном поиске новой, более удачной системы записи программ нас иногда озаряют идеи, переживающие контекст, в котором зародились. Как Pascal многое позаимствовал у Algol, как Java выросла из C, так и каждый язык что-то берет у своих предшественников.

Язык – это одновременно набор инструментов и площадка для игр. У него есть практическая сторона, но он же служит и полигоном для испытания новых идей, которые могут быть приняты или отвергнуты сообществом программистов.

Одна из наиболее далеко идущих идей – концепция объектно-ориентированного программирования (ООП). Многие скажут, что значимость ООП имеет скорее эволюционный, нежели революционный характер, но никто не возразит против того, что оно оказало огромное влияние на индустрию. Двадцать пять лет назад объектная ориентированность представляла в основном академический интерес; сегодня это универсально принятая парадигма.

Вездесущность ООП породила много «рекламной чепухи» в индустрии. В классической работе, написанной в конце 1980-х годов, Роджер Кинг отметил: «Если вы хотите продать кошку специалисту по компьютерам, скажите, что она объектно-ориентированная». Мнения по поводу того, что на самом деле представляет собой ООП, весьма неоднородны, и даже среди тех, кто разделяет общую точку зрения, имеются разногласия относительно терминологии.

Мы не ставим себе целью поучаствовать в спорах. Мы согласны, что ООП – полезный инструмент и ценная методология решения задач; мы не утверждаем, что она способна излечить рак.

Что касается истинной природы ООП, то у нас есть любимые определения и термины, но мы пользуемся ими лишь для эффективного общения, так что пререкаться по поводу смысла слов не станем.

Обо всем этом пришлось сказать лишь потому, что знакомство с основами ООП необходимо для чтения этой книги и понимания примеров и подходов. Что бы ни говорили о Ruby, он безусловно является объектно-ориентированным языком.

1.1. Введение в объектно-ориентированное программирование

Прежде чем начать разговор о самом языке Ruby, неплохо было бы потолковать об объектно-ориентированном программировании вообще. Поэтому сейчас мы вкратце рассмотрим общие идеи, лишь слегка касаясь Ruby.

1.1.1. Что такое объект

В объектно-ориентированном программировании объект – фундаментальное понятие. *Объект* – это сущность, служащая контейнером для данных и управляющая доступом к этим данным. С объектом связан набор *атрибутов*, которые в сущности представляют собой просто переменные, принадлежащие объекту. (В этой книге мы будем без стеснения употреблять привычный термин «переменная» в применении к атрибутам.) Кроме того, с объектом ассоциирован набор функций, предоставляющих интерфейс к функциональным возможностям объекта. Эти функции называются *методами*.

Важно отметить, что любой объектно-ориентированный язык предоставляет механизм инкапсуляции. В общепринятом смысле это означает, во-первых, что атрибуты и методы объекта ассоциированы именно с этим объектом, а во-вторых, что область видимости атрибутов и методов по умолчанию ограничена самим объектом (применение принципа сокрытия информации).

Объект считается экземпляром класса объекта (обычно он называется просто *классом*). Класс можно представлять себе как чертеж или образец, а объект – как вещь, изготовленную по этому чертежу. Также класс часто называют абстрактным типом, то есть типом более сложным, нежели целое или строка символов.

Создание объекта (экземпляра класса) называют *инстанцированием*. В некоторых языках имеются явные конструкторы и деструкторы – функции, выполняющие действия, необходимые соответственно для инициализации и уничтожения объекта. Отметим попутно, что в Ruby есть нечто, что можно назвать конструктором, но никакого аналога деструктора не существует (благодаря наличию механизма сборки мусора).

Иногда возникает ситуация, когда некоторые данные имеют широкую область видимости, не ограниченную одним объектом, и помещать копию такого атрибута в каждый экземпляр класса неправильно. Рассмотрим, к примеру, класс `MyDogs` и

три объекта этого класса: *fido*, *rover* и *spot*. У каждой собаки могут быть такие атрибуты, как возраст и дата вакцинации. Предположим, однако, что нужно сохранить еще и имя владельца всех собак. Можно, конечно, поместить его в каждый объект, но это пустая трата памяти, к тому же искажающая смысл дизайна. Ясно, что атрибут «имя владельца» принадлежит не отдельному объекту, а классу в целом. Такие атрибуты (синтаксис их определения в разных языках различен) называются *атрибутами класса* (или *переменными класса*).

Есть немало ситуаций, в которых может понадобиться переменная класса. Допустим, например, что нужно знать, сколько всего было создано объектов некоторого класса. Можно было бы завести переменную класса, инициализировать ее нулем и увеличивать на единицу при создании каждого объекта. Эта переменная ассоциирована именно с классом, а не с каким-то конкретным объектом. С точки зрения области видимости она не отличается от любого другого атрибута, но существует лишь одна ее копия для всего множества объектов данного класса.

Чтобы отличить атрибуты класса от обыкновенных атрибутов, последние часто называют *атрибутами объекта* (или *атрибутами экземпляра*). Условимся, что в этой книге под словом «атрибут» понимается атрибут экземпляра, если явно не оговорено, что это атрибут класса.

Точно так же методы объекта служат для управления доступом к его атрибутам и предоставляют четко определенный интерфейс для этой цели. Но иногда желательно или даже необходимо определить метод, ассоциированный с самим классом. Неудивительно, что метод класса управляет доступом к переменным класса, кроме того, выполняя действия, распространяющиеся на весь класс, а не на какой-то конкретный объект. Как и в случае с атрибутами, мы будем считать, что метод принадлежит объекту, если явно не оговорено противное.

Стоит отметить, что в некотором смысле все методы являются методами класса. Не нужно думать, что, создав сто объектов, мы породили сотню копий кода методов! Однако правила ограничения области видимости гласят, что метод каждого объекта оперирует данными только того объекта, от имени которого вызван. Тем самым у нас создается иллюзия, будто методы объекта ассоциированы с самими объектами.

1.1.2. Наследование

Мы подходим к одной из самых сильных сторон ООП – наследованию. *Наследование* – это механизм, позволяющий расширять ранее определенную сущность путем добавления новых возможностей. Короче говоря, наследование – это способ повторного использования кода. (Простой и эффективный механизм повторного использования долго был Святым Граалем в информатике. Много десятилетий назад его поиски привели к изобретению параметризованных процедур и библиотек. ООП – лишь одна из последних попыток реализации искомого.)

Обычно наследование рассматривается на уровне класса. Если нам необходим какой-то класс, а в наличии имеется более общий, то можно определить свой класс так, чтобы он наследовал поведение уже существующего. Предположим, например, что есть класс *Polygon*, описывающий выпуклые многоугольники. Тогда класс

прямоугольника `Rectangle` можно унаследовать от `Polygon`. При этом `Rectangle` будет иметь все атрибуты и методы класса `Polygon`. Так, может уже быть написан метод, вычисляющий периметр путем суммирования длин всех сторон. Если все было реализовано правильно, этот метод автоматически будет работать и для нового класса; переписывать код не придется.

Если класс `В` наследует классу `А`, мы говорим, что `В` является подклассом `А`, а `А` – суперкласс `В`. По-другому говорят, что `А` – *базовый* или *родительский класс*, а `В` – *производный* или *дочерний класс*.

Как мы видели, производный класс может трактовать методы своего базового класса как свои собственные. С другой стороны, он может переопределить метод базового класса, предоставив иную его реализацию. Кроме того, в большинстве языков есть возможность вызвать из переопределенного метода метод базового класса с тем же именем. Иными словами, метод `foo` класса `В` знает, как вызвать метод `foo` класса `А`. (Любой язык, не предоставляющий такого механизма, можно заподозрить в отсутствии истинной объектной ориентированности.) То же верно и в отношении атрибутов.

Отношение между классом и его суперклассом интересно и важно, обычно его называют отношением «является». Действительно, квадрат `Square` «является» прямоугольником `Rectangle`, а прямоугольник `Rectangle` «является» многоугольником `Polygon` и т.д. Поэтому, рассматривая иерархию наследования (а такие иерархии в том или ином виде присутствуют в любом объектно-ориентированном языке), мы видим, что в любой ее точке специализированные сущности «являются» подклассами более общих. Отметим, что это отношение транзитивно, – если обратиться к предыдущему примеру, то квадрат «является» многоугольником. Однако отношение «является» не коммутативно – каждый прямоугольник есть многоугольник, но не каждый многоугольник – прямоугольник.

Это подводит нас к теме множественного наследования. Можно представить себе класс, который наследует нескольким классам. Например, классы `Dog` (Собака) и `Cat` (Кошка) могут наследовать классу `Mammal` (Млекопитающее), а `Sparrow` (Воробей) и `Raven` (Ворон) – классу `WingedCreature` (Крылатое). Но как быть с классом `Bat` (ЛетучаяМышь)? Он с равным успехом может наследовать и `Mammal`, и `WingedCreature`! Это хорошо согласуется с нашим жизненным опытом, ведь многие вещи можно отнести не к одной категории, а сразу к нескольким, не вложенным друг в друга.

Множественное наследование, вероятно, наиболее противоречивая часть ООП. Некоторые указывают на потенциальные неоднозначности, требующие разрешения. Например, если в обоих классах `Mammal` и `WingedCreature` имеется атрибут `size` (размер) или метод `eat` (есть), то какой из них имеется в виду, когда мы обращаемся к нему из объекта класса `Bat`? С этой трудностью тесно связана проблема ромбовидного наследования; она называется так из-за формы диаграммы наследования, возникающей, когда оба суперкласса наследуют одному классу. Представьте себе, что классы `Mammal` и `WingedCreature` наследуют общему предку `organism` (Организм); тогда иерархия наследования от `Organism` к `Bat` будет иметь форму ромба. Но как быть с атрибутами, которые оба промежуточных класса наследуют от

своего родителя? Получает ли `bat` две копии? Или они должны быть объединены в один атрибут, поскольку все равно заимствованы у общего предка?

Это скорее проблемы проектировщика языка, а не программиста. В разных объектно-ориентированных языках они решаются по-разному. Иногда вводятся правила, согласно которым какое-то одно определение атрибута «выигрывает». Либо же предоставляется возможность различать одноименные атрибуты. Иногда даже язык позволяет вводить псевдонимы или переименовывать идентификаторы. Многими это рассматривается как аргумент против множественного наследования – о механизмах разрешения подобных конфликтов имен нет единого мнения, поэтому все они «языкозависимы». В языке `C++` предлагается минимальный набор средств для разрешения неоднозначностей; механизмы языка `Eiffel`, наверное, получше, а в `Perl` проблема решается совсем по-другому.

Есть и альтернатива – полностью запретить множественное наследование. Такой подход принят в языках `Java` и `Ruby`. На первый взгляд, это даже не назовешь компромиссным решением, но, вскоре мы убедимся, что все не так плохо, как кажется. Мы познакомимся с приемлемой альтернативой традиционному множественному наследованию, но сначала обсудим полиморфизм – еще одно понятие из арсенала ООП.

1.1.3. Полиморфизм

Термин «полиморфизм», наверное, вызывает самые жаркие семантические споры. Каждый знает, что это такое, но все понимают его по-разному. (Не так давно вопрос «Что такое полиморфизм?» стал популярным во время собеседования при поступлении на работу. Если его зададут вам, рекомендую процитировать какого-нибудь эксперта, например Бертрана Мейера или Бьерна Страуструпа; если собеседник не согласится, то пусть он спорит с классиком, а не с вами.)

Буквально слово «полиморфизм» означает «способность принимать разные формы или обличья». В самом широком смысле так называют ситуацию, когда различные объекты по-разному отвечают на одно и то же сообщение или вызов метода.

Дамиан Конвей (Damian Conway) в книге «Object-Oriented Perl» проводит смысловое различие между двумя видами полиморфизма. Первый, *наследственный полиморфизм*, – то, что имеет в виду большинство программистов, говорящих о полиморфизме.

Если некоторый класс наследует своему суперклассу, то по определению все методы суперкласса присутствуют также и в подклассе. Таким образом, цепочка наследования представляет собой линейную иерархию классов, отвечающих на одни и те же методы. Нужно, конечно, помнить, что в любом подклассе метод может быть переопределен; именно это и составляет сильную сторону наследования. При вызове метода объекта обычно отвечает либо метод, унаследованный от суперкласса, либо более специализированный вариант этого метода, созданный в интересах именно данного подкласса.

В языках со статической типизацией, например в `C++`, наследственный полиморфизм гарантирует совместимость типов вниз по цепочке наследования (но не

в обратном направлении). Скажем, если в наследует А, то указатель на объект класса А может указывать и на объект класса В; обратное же неверно. Совместимость типов – существенная черта ООП в подобных языках, можно даже сказать, что полиморфизм ей и исчерпывается. Но, конечно же, полиморфизм можно реализовать и в отсутствие статической типизации (как в Ruby).

Второй вид полиморфизма, упомянутый Конвеем, – это *интерфейсный полиморфизм*. Для него не требуется наличия отношения наследования между классами; нужно лишь, чтобы в интерфейсах объектов были методы с одним и тем же именем. Такие объекты можно трактовать как принадлежащие одному виду, и потому мы имеем некую разновидность полиморфизма (хотя в большинстве работ он так не называется).

Читатели, знакомые с языком Java, понимают, что в нем реализованы оба вида полиморфизма. Класс в Java может расширять другой класс, наследуя ему с помощью ключевого слова `extends`, а может с помощью ключевого слова `implements` реализовывать интерфейс, за счет чего приобретает заранее известный набор методов (которые необходимо переопределить). Такой синтаксис позволяет интерпретатору Java во время компиляции определить, можно ли вызывать данный метод для конкретного объекта.

Ruby поддерживает интерфейсный полиморфизм, но по-другому. Он позволяет определять модули, методы которых допускается «подмешивать» к существующим классам. Но обычно модули так не используются. Модуль состоит из методов и констант, которые можно использовать так, будто они являются частью класса или объекта. Когда модуль подмешивается с помощью предложения `include`, мы получаем ограниченную форму множественного наследования. (По словам проектировщика языка Юкихио Мацумото, это можно рассматривать как одиночное наследование с разделением реализации.) Таким образом удастся сохранить преимущества множественного наследования, не страдая от его недостатков.

1.1.4. Еще немного терминов

В языках, подобных C++, существует понятие *абстрактного класса*. Такому классу разрешается наследовать, но создать его экземпляр невозможно. В более динамичном языке Ruby такого понятия нет, но если программист пожелает, то может смоделировать его, потребовав, чтобы все методы были переопределены в производных классах. Полезно это или нет, оставляем на усмотрение читателя.

Создатель языка C++ Бьерн Страуструп определяет также понятие *конкретного типа*. Это класс, существующий только для удобства. Он спроектирован не для наследования; более того, ожидается, что ему никто никогда наследовать не будет. Другими словами, преимущества ООП в этом случае сводятся только к инкапсуляции. Ruby не поддерживает такой конструкции синтаксически (как и C++), но по природе своей прекрасно приспособлен для создания подобных классов.

Считается, что некоторые языки поддерживают более «чистую» модель ООП, чем другие. (К ним мы применяем термин «радикально объектно-ориентированный».) Это означает, что любая сущность в языке является объектом, даже примитивные типы представлены полноценными классами, а переменные и константы

рассматриваются как экземпляры. В таких языках, как Java, C++ и Eiffel, дело обстоит иначе. В них примитивные типы (особенно константы) не являются настоящими объектами, хотя иногда могут рассматриваться как таковые с помощью «классов-обертков». Вероятно, есть языки, которые более радикально объектно ориентированы, чем Ruby, но их немного.

Большинство объектно-ориентированных языков статично; методы и атрибуты, принадлежащие классу, глобальные переменные и иерархия наследования определяются во время компиляции. Быть может, самый сложный концептуальный переход заключается в том, что в Ruby все это происходит динамически. И определения, и даже порядок наследования можно задавать во время исполнения. Честно говоря, каждое объявление или определение выполняется во время работы программы. Помимо прочих достоинств, это позволяет избавиться от условной компиляции, и во многих случаях получается более эффективный код.

На этом мы завершаем беглую экскурсию в мир ООП. Мы старались последовательно применять введенные здесь термины на протяжении всей книги. Перейдем теперь к краткому обзору самого языка Ruby.

1.2. Базовый синтаксис и семантика Ruby

Выше мы отметили, что Ruby – настоящий динамический объектно-ориентированный язык.

Прежде чем переходить к обзору синтаксиса и семантики, упомянем некоторые другие его особенности.

Ruby – прагматичный (agile) язык. Он пластичен и поощряет частую переработку (рефакторинг), которая выполняется без особого труда.

Ruby – интерпретируемый язык. Разумеется, в будущем ради повышения производительности могут появиться и компиляторы Ruby, но мы считаем, что у интерпретатора много достоинств. Он не только позволяет быстро создавать прототипы, но и сокращает весь цикл разработки.

Ruby ориентирован на выражения. Зачем писать предложение, когда выражения достаточно? Это означает, в частности, что программа становится более компактной, поскольку общие части выносятся в отдельное выражение и повторения удастся избежать.

Ruby – язык сверхвысокого уровня (VHLL). Один из принципов, положенных в основу его проектирования, заключается в том, что компьютер должен работать для человека, а не наоборот. Под «плотностью» Ruby понимают тот факт, что сложные, запутанные операции можно записать гораздо проще, чем в языках более низкого уровня.

Начнем мы с рассмотрения общего духа языка и некоторых применяемых в нем терминов. Затем вкратце обсудим природу программ на Ruby, а потом уже перейдем к примерам.

Прежде всего отметим, что программа на Ruby состоит из отдельных строк, – как в C, но не как в «древних» языках наподобие Фортрана. В одной строке может быть сколько угодно лексем, лишь бы они правильно отделялись пропусками.

В одной строке может быть несколько предложений, разделенных точками с запятой; только в этом случае точка с запятой и необходима. Логическая строка может быть разбита на несколько физических при условии, что все, кроме последней, заканчиваются обратной косой чертой или лексическому анализатору дан знак, что предложение еще не закончено. Таким знаком может, например, быть запятая в конце строки.

Главной программы как таковой (функции `main`) не существует; исполнение происходит сверху вниз. В более сложных программах в начале текста могут располагаться многочисленные определения, за которыми следует (концептуально) главная программа. Но даже в этом случае программа исполняется сверху вниз, так как в Ruby все определения исполняются.

1.2.1. Ключевые слова и идентификаторы

Ключевые (или зарезервированные) слова в Ruby обычно не применяются ни для каких иных целей. Вот их полный перечень:

BEGIN	END	alias	and	begin
break	case	class	def	defined?
do	else	elsif	end	ensure
false	for	if	in	module
next	nil	not	or	redo
rescue	retry	return	self	super
then	true	undef	unless	until
when	while	yield		

Имена переменных и других идентификаторов обычно начинаются с буквы или специального модификатора. Основные правила таковы:

- имена локальных переменных (и таких псевдопеременных, как `self` и `nil`) начинаются со строчной буквы или знака подчеркивания `_`;
- имена глобальных переменных начинаются со знака доллара `$`;
- имена переменных экземпляра (принадлежащих объекту) начинаются со знака «собачки» `@`;
- имена переменных класса (принадлежащих классу) предваряются двумя знаками `@` (`@@`);
- имена констант начинаются с прописной буквы;
- в именах идентификаторов знак подчеркивания `_` можно использовать наравне со строчными буквами;
- имена специальных переменных, начинающиеся со знака доллара (например, `$1` и `$/`), здесь не рассматриваются.

Примеры:

- локальные переменные `alpha`, `_ident`, `some_var`;
- псевдопеременные `self`, `nil`, `__FILE__`;
- константы `K6chip`, `Length`, `LENGTH`;
- переменные экземпляра `@foobar`, `@thx1138`, `@NOT_CONST`;
- переменные класса `@@phydeaux`, `@my_var`, `@@NOT_CONST`;
- глобальные переменные `$beta`, `$B12vitamin`, `$NOT_CONST`.

1.2.2. Комментарии и встроенная документация

Комментарии в Ruby начинаются со знака решетки (#), находящегося вне строки или символьной константы, и продолжаются до конца строки:

```
x = y + 5    # Это комментарий.
# Это тоже комментарий.
print "# А это не комментарий."
```

Предполагается, что встроенная документация будет извлечена из программы каким-нибудь внешним инструментом. С точки зрения интерпретатора это обычный комментарий. Весь текст, расположенный между строками, которые начинаются с лексем `=begin` и `=end` (включительно), игнорируется интерпретатором (этим лексемам не должны предшествовать пробелы).

```
=begin
Назначение этой программы -
излечить рак
и установить мир во всем мире.
=end
```

1.2.3. Константы, переменные и типы

В Ruby переменные не имеют типа, однако объекты, на которые переменные ссылаются, тип имеют. Простейшие типы – это *символ*, *число* и *строка*.

Числовые константы интуитивно наиболее понятны, равно как и строки. В общем случае строка, заключенная в двойные кавычки, допускает интерполяцию выражений, а заключенная в одиночные кавычки интерпретируется почти буквально – в ней распознается только экранированная обратная косая черта.

Ниже показана «интерполяция» переменных и выражений в строку, заключенную в двойные кавычки:

```
a = 3
b = 79
puts "#{a} умноженное на #{b} = #{a*b}"    # 3 умноженное на 79 = 237
```

Более подробная информация о литералах (числах, строках, регулярных выражениях и т. п.) приведена в следующих главах.

Стоит упомянуть особую разновидность строк, которая полезна прежде всего в небольших сценариях, применяемых для объединения более крупных программ. Строка, выводимая программой, посылается операционной системе в качестве подлежащей исполнению команды, а затем результат выполненной команды подставляется обратно в строку. В простейшей форме для этого применяются строки, заключенные в обратные кавычки. В более сложном варианте используется синтаксическая конструкция `%x`:

```
'whoami'
'ls -l'
%x[grep -i meta *.html | wc -l]
```

Регулярные выражения в Ruby похожи на символьные строки, но используются по-другому. Обычно в качестве ограничителя выступает символ косой черты.

Синтаксис регулярных выражений в Ruby и Perl имеет много общего. Подробнее о регулярных выражениях см. главу 3.

Массивы в Ruby – очень мощная конструкция; они могут содержать данные любого типа. Более того, в одном массиве можно хранить данные разных типов. В главе 8 мы увидим, что все массивы – это экземпляры класса `Array`, а потому к ним применимы разнообразные методы. Массив-константа заключается в квадратных скобки. Примеры:

```
[1, 2, 3]
[1, 2, "застегни мне молнию на сапоге"]
[1, 2, {3,4}, 5]
["alpha", "beta", "gamma", "delta"]
```

Во втором примере показан массив, содержащий целые числа и строки. В третьем примере мы видим вложенный массив, а в четвертом – массив строк. Как и в большинстве других языков, нумерация элементов массива начинается с нуля. Так, в последнем из показанных выше примеров элемент `"gamma"` имеет индекс 2. Все массивы динамические, задавать размер при создании не нужно.

Поскольку массивы строк встречаются очень часто (а набирать их неудобно), для них предусмотрен специальный синтаксис:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

Здесь не нужны ни кавычки, ни запятые; элементы разделяются пробелами. Если встречаются элементы, содержащие внутренние пробелы, такой синтаксис, конечно, неприменим.

Для доступа к конкретному элементу массива по индексу применяются квадратные скобки. Результирующее выражение можно получить или выполнить для него присваивание:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Еще одна «могучая» конструкция в Ruby – это хэш. Его также называют ассоциативным массивом или словарем. Хэш – это множество пар данных; обыкновенно он применяется в качестве справочной таблицы или как обобщенный массив, в котором индекс не обязан быть целым числом. Все хэши являются экземплярами класса `Hash`.

Хэш-константа, как правило, заключается в фигурные скобки, а ключи отделяются от значений символом `=>`. Ключ можно считать индексом для доступа к ассоциированному с ним значению. На типы ключей и значений не налагается никаких ограничений. Примеры:

```
{1=>1, 2=>4, 3=>9, 4=>16, 5=>25, 6=>36}
{"cat"=>"cats", "ox"=>"oxen", "bacterium"=>"bacteria"}
{"водород"=>1, "гелий"=>2, "углерод"=>12}
{"нечетные"=>[1,3,5,7], "четные"=>[2,4,6,8]}
{"foo"=>123, {4,5,6}=>"my array", "867-5309"=>"Jenny"}
```

К содержимому хэша-переменной доступ осуществляется так же, как для массивов, — с помощью квадратных скобок:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
```

Однако следует подчеркнуть, что у массивов и хэшей много методов, именно они и делают эти контейнеры полезными. Ниже, в разделе «ООП в Ruby», мы рассмотрим эту тему более подробно.

1.2.4. Операторы и приоритеты

Познакомившись с основными типами данных, перейдем к операторам в языке Ruby. В приведенном ниже списке они представлены в порядке убывания приоритета:

::	Разрешение области видимости
[]	Взятие индекса
**	Возведение в степень
+ - ! ~	Унарный плюс/минус, НЕ ...
* / %	Умножение, деление ...
+ -	Сложение/вычитание
<< >>	Логические сдвиги ...
&	Поразрядное И
^	Поразрядное ИЛИ, исключающее ИЛИ
> >= < <=	Сравнение
== === <=> != =~ !~	Равенство, неравенство ...
&&	Логическое И
	Логическое ИЛИ
.. ...	Операторы диапазона
= (also +=, -=, ...)	Присваивание
?:	Тернарный выбор
not	Логическое отрицание
and or	Логическое И, ИЛИ

Некоторые из перечисленных символов служат сразу нескольким целям. Например, оператор << обозначает поразрядный сдвиг влево, но также применяется для добавления в конец (массива, строки и т. д.) и как маркер встроенного документа. Аналогично знак + означает сложение чисел и конкатенацию строк. Ниже мы увидим, что многие операторы — это просто сокращенная запись вызова методов.

Итак, мы определили большую часть типов данных и многие из возможных над ними операций. Прежде чем двигаться дальше, приведем пример программы.

1.2.5. Пример программы

В любом руководстве первой всегда приводят программу, печатающую строку `Hello, world!`, но мы рассмотрим что-нибудь более содержательное. Вот небольшая интерактивная консольная программа, позволяющая переводить температуру из шкалы Фаренгейта в шкалу Цельсия и наоборот.

```

print "Введите температуру и шкалу (C or F): "
str = gets
exit if str.nil? or str.empty?
str.chomp!
temp, scale = str.split(" ")

abort "#{temp} недопустимое число." if temp !~ /\-?\d+/

temp = temp.to_f
case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
  else
    abort "Необходимо задать C или F."
  end

  if f.nil?
    print "#{c} градусов C\n"
  else
    print "#{f} градусов F\n"
  end
end

```

Ниже приведены примеры прогона этой программы. Показано, как она переводит градусы Фаренгейта в градусы Цельсия и наоборот, а также как обрабатывает неправильно заданную шкалу или число:

```

Введите температуру и шкалу (C or F): 98.6 F
37.0 градусов C

```

```

Введите температуру и шкалу (C or F): 100 C
212.0 градусов F

```

```

Введите температуру и шкалу (C or F): 92 G
Необходимо задать C или F.

```

```

Введите температуру и шкалу (C or F): junk F
junk недопустимое число.

```

Теперь рассмотрим, как эта программа работает. Все начинается с предложения `print`, которое есть не что иное, как вызов метода `print` из модуля `Kernel`. Данный метод выполняет печать на стандартный вывод. Это самый простой способставить курсор в конце строки.

Далее мы вызываем метод `gets` (прочитать строку из стандартного ввода) и присваиваем полученное значение переменной `str`. Для удаления хвостового символа новой строки вызывается метод `chomp!`.

Обратите внимание, что `print` и `gets`, которые выглядят как «свободные» функции, на самом деле являются методами класса `Object` (который, вероятно, наследует `Kernel`). Точно так же `chomp!` — метод, вызываемый от имени объекта

`str`. При вызовах методов в Ruby обычно можно опускать скобки: `print "foo"` и `print ("foo")` – одно и то же.

В переменной `str` хранится символьная строка, но могли бы храниться данные любого другого типа. В Ruby данные имеют тип, а переменные – нет. Переменная начинает существовать, как только интерпретатор распознает присваивание ей; никаких предварительных объявлений не существует.

Метод `exit` завершает программу. В той же строке мы видим управляющую конструкцию, которая называется «модификатор `if`». Он аналогичен предложению `if`, существующему в большинстве языков, только располагается после действия. Для модификатора `if` нельзя задать ветвь `else`, и он не требует закрытия. Что касается условия, мы проверяем две вещи: имеет ли переменная `str` значение (то есть не равна `nil`) и не является ли она пустой строкой. Если встретится конец файла, то будет истинно первое условие; если же пользователь нажмет клавишу `Enter`, не введя никаких данных, – второе.

Это предложение можно было бы записать и по-другому:

```
exit if not str or not str[0]
```

Эти проверки работают потому, что переменная может иметь значение `nil`, а `nil` в Ruby в логическом контексте вычисляется как «ложно». На самом деле как «ложно» вычисляются `nil` и `false`, а все остальное – как «истинно». Это означает, кстати, что пустая строка `""` и число `0` – не «ложно».

В следующем предложении над строкой выполняется операция `chomp!` (для удаления хвостового символа новой строки). Восклицательный знак в конце предупреждает, что операция изменяет значение самой строки, а не возвращает новую. Восклицательный знак применяется во многих подобных ситуациях как напоминание программисту о том, что у метода есть побочное действие или что он более «опасен», чем аналогичный метод без восклицательного знака. Так, метод `chomp` возвращает такой же результат, но не модифицирует значение строки, для которой вызван.

В следующем предложении мы видим пример множественного присваивания. Метод `split` разбивает строку на куски по пробелам и возвращает массив. Двум переменным в левой части оператора присваиваются значения первых двух элементов массива в правой части.

В следующем предложении `if` с помощью простого регулярного выражения выясняется, введено ли допустимое число. Если строка не соответствует образцу, который состоит из необязательного знака «минус» и одной или более цифр, то число считается недопустимым и программа завершается. Отметим, что предложение `if` оканчивается ключевым словом `end`. Хотя в данном случае это не нужно, мы могли бы включить перед `end` ветвь `else`. Ключевое слово `then` необязательно; в этой книге мы стараемся не употреблять его.

Метод `to_f` преобразует строку в число с плавающей точкой. Это число записывается в ту же переменную `temp`, в которой раньше хранилась строка.

Предложение `case` выбирает одну из трех ветвей: пользователь указал `C`, `F` или какое-то другое значение в качестве шкалы. В первых двух случаях выполняется вычисление, в третьем мы печатаем сообщение об ошибке и выходим.

Кстати, предложение `case` в Ruby позволяет гораздо больше, чем показано в примере. Нет никаких ограничений на типы данных, а все выражения могут быть произвольно сложными, в том числе диапазонами или регулярными выражениями.

В самом вычислении нет ничего интересного. Но обратите внимание, что переменные `s` и `f` впервые встречаются внутри ветвей `case`. В Ruby нет никаких объявлений – переменная начинает существовать только в результате присваивания. А это означает, что после выхода из `case` лишь одна из переменных `s` и `f` будет иметь действительное значение.

Мы воспользовались этим фактом, чтобы понять, какая ветвь исполнялась, и в зависимости от этого вывести то или другое сообщение. Сравнение `f` с `nil` позволяет узнать, есть ли у переменной осмысленное значение. Этот прием применен только для демонстрации возможности: ясно, что при желании можно было бы поместить печать прямо внутрь предложения `case`.

Внимательный читатель заметит, что мы пользовались только «локальными» переменными. Это может показаться странным, так как, на первый взгляд, их областью видимости является вся программа. На самом деле они локальны относительно верхнего уровня программы. Глобальными они кажутся лишь потому, что в этой простой программе нет контекстов более низкого уровня. Но если бы мы объявили какие-нибудь классы или методы, то в них переменные верхнего уровня были бы не видны.

1.2.6. Циклы и ветвление

Потратим немного времени на изучение управляющих конструкций. Мы уже видели простое предложение `if` и модификатор `if`. Существуют также парные структуры, в которых используется ключевое слово `unless` (в них также может присутствовать необязательная ветвь `else`), а равно применяемые в выражениях формы `if` и `unless`. Все они сведены в таблицу 1.1.

Таблица 1.1. Условные предложения

Формы с <code>if</code>	Формы с <code>unless</code>
<code>if x < 5 then statement1 end</code>	<code>unless x >= 5 then statement1 end</code>
<code>if x < 5 then statement1 else statement2 end</code>	<code>unless x < 5 then statement2 else statement1 end</code>
<code>statement1 if y == 3</code>	<code>statement1 unless y != 3</code>
<code>x = if a>0 then b else c end</code>	<code>x = unless a<=0 then c else b end</code>

Здесь формы с ключевыми словами `if` и `unless`, расположенные в одной строке, выполняют в точности одинаковые функции. Обратите внимание, что слово

then можно опускать во всех случаях, кроме последнего (предназначенного для использования в выражениях). Также заметьте, что в модификаторах (третья строка) ветви else быть не может.

Предложение case в Ruby позволяет больше, чем в других языках. В его ветвях можно проверять различные условия, а не только сравнивать на равенство. Так, например, разрешено сопоставление с регулярным выражением. Проверки в предложении case эквивалентны оператору ветвящегося равенства (===), поведение которого зависит от объекта. Рассмотрим пример:

```
case "Это строка символов."
  when "одно значение"
    puts "Ветвь 1"
  when "другое значение"
    puts "Ветвь 2"
  when /симв/
    puts "Ветвь 3"
  else
    puts "Ветвь 4"
end
```

Этот код напечатает Ветвь 3. Почему? Сначала проверяемое выражение сравнивается на равенство с двумя строками: "одно значение" и "другое значение". Эта проверка завершается неудачно, поэтому мы переходим к третьей ветви. Там находится образец, с которым сопоставляется выражение. Поскольку оно соответствует образцу, то выполняется предложение print. В ветви else обрабатывается случай, когда ни одна из предшествующих проверок не прошла.

Если проверяемое выражение – целое число, то его можно сравнивать с целочисленным диапазоном (например, 3..8); тогда проверяется, что число попадает в диапазон. В любом случае выполняется код в первой подошедшей ветви.

В Ruby имеется богатый набор циклических конструкций. К примеру, while и until – циклы с предварительной проверкой условия, и оба работают привычным образом: в первом случае задается условие продолжения цикла, а во втором – условие завершения. Есть также их формы с модификатором, как для предложений if и unless. Кроме того, в модуле kernel есть метод loop (по умолчанию бесконечный цикл), а в некоторых классах реализованы итераторы.

В примерах из таблицы 1.2 предполагается, что где-то определен такой массив list:

```
list = %w[alpha bravo charlie delta echo];
```

В цикле этот массив обходится и печатается каждый его элемент.

Таблица 1.2. Циклы

<pre># Цикл 1 (while) i=0 while i < list.size do print "#{list[i]} " i += 1 end</pre>	<pre># Цикл 2 (until) i=0 until i == list.size do print "#{list[i]} " i += 1 end</pre>

Таблица 1.2. Циклы

<pre># Цикл 3 (for) for x in list do print "#{x} " end</pre>	<pre># Цикл 4 (итератор 'each') list.each do x print "#{x} " end</pre>
<pre># Цикл 5 (метод 'loop') i=0 n=list.size-1 loop do print "#{list[i]} " i += 1 break if i > n end</pre>	<pre># Цикл 6 (метод 'loop') i=0 n=list.size-1 loop do print "#{list[i]} " i += 1 break unless i <= n end</pre>
<pre># Цикл 7 (итератор 'times') n=list.size n.times do i print "#{list[i]} " end</pre>	<pre># Цикл 8 (итератор 'upto') n=list.size-1 0.upto(n) do i print "#{list[i]} " end</pre>
<pre># Цикл 9 (for) n=list.size-1 for i in 0..n do print "#{list[i]} " end</pre>	<pre># Цикл 10 ('each_index') list.each_index do xt print "#{list[x]} " end</pre>

Рассмотрим эти примеры более подробно. Циклы 1 и 2 – «стандартные» формы циклов `while` и `until`; ведут они себя практически одинаково, только условия противоположны. Циклы 3 и 4 – варианты предыдущих с проверкой условия в конце, а не в начале итерации. Отметим, что использование слов `begin` и `end` в этом контексте – просто грязный трюк; на самом деле это был бы блок `begin/end` (применяемый для обработки исключений), за которым следует модификатор `while` или `until`. Однако для тех, кто желает написать цикл с проверкой в конце, разницы нет.

На мой взгляд, конструкции 3 и 4 – самый «правильный» способ кодирования циклов. Они заметно проще всех остальных: нет ни явной инициализации, ни явной проверки или инкремента. Это возможно потому, что массив «знает» свой размер, а стандартный итератор `each` (цикл 6) обрабатывает такие детали автоматически. На самом деле в цикле 5 производится неявное обращение к этому итератору, поскольку цикл `for` работает с любым объектом, для которого определен итератор `each`. Цикл `for` – лишь сокращенная запись для вызова `each`; часто такие сокращения называют «синтаксической глазурью», имея в виду, что это не более чем удобная альтернативная форма другой синтаксической конструкции.

В циклах 5 и 6 используется конструкция `loop`. Выше мы уже отмечали, что хотя `loop` выглядит как ключевое слово, на самом деле это метод модуля `Kernel`, а вовсе не управляющая конструкция.

В циклах 7 и 8 используется тот факт, что у массива есть числовой индекс. Итератор `times` исполняется заданное число раз, а итератор `upto` увеличивает свой параметр до заданного значения. И тот, и другой для данной ситуации приспособлены плохо.

Цикл 9 – это вариант цикла `for`, предназначенный специально для работы со значениями индекса при помощи указания диапазона. В цикле 10 мы пробегаем весь диапазон индексов массива с помощью итератора `each_index`.

В предыдущих примерах мы уделили недостаточно внимания вариантам циклов `while` и `loop` с модификаторами. Они довольно часто используются из-за краткости. Вот еще два примера, в которых делается одно и то же:

```
perform_task() until finished
```

```
perform_task() while not finished
```

Также из таблицы 1.2 осталось неясным, что циклы не всегда выполняются от начала до конца. Число итераций не всегда предсказуемо. Нужны дополнительные средства управления циклами.

Первое из них – ключевое слово `break`, встречающееся в циклах 5 и 6. Оно позволяет досрочно выйти из цикла; в случае вложенных циклов происходит выход из самого внутреннего. Для программистов на С это интуитивно очевидно.

Ключевое слово `retry` применяется в двух случаях: в контексте итератора и в контексте блока `begin-end` (обработка исключений). В теле итератора (или цикла `for`) оно заставляет итератор заново выполнить инициализацию, то есть повторно вычислить переданные ему аргументы. Отметим, что к циклам общего вида это не относится.

Ключевое слово `redo` – обобщение `retry` на циклы общего вида. Оно работает в циклах `while` и `until`, как `retry` в итераторах.

Ключевое слово `next` осуществляет переход на конец самого внутреннего цикла и возобновляет исполнение с этой точки. Работает для любого цикла и итератора.

Как мы только что видели, итератор – важное понятие в Ruby. Но следует отметить, что язык позволяет определять и пользовательские итераторы, не ограничиваясь встроенными.

Стандартный итератор для любого объекта называется `each`. Это существенно отчасти из-за того, что позволяет использовать цикл `for`. Но итераторам можно давать и другие имена и применять для разных целей.

В качестве примера рассмотрим многоцелевой итератор, который имитирует цикл с проверкой условия в конце (как в конструкции `do-while` в С или `repeat-until` в Pascal):

```
def repeat(condition)
  yield
  retry if not condition
end
```

В этом примере ключевое слово `yield` служит для вызова блока, который задается при таком вызове итератора:


```
j=0
repeat (j >= 10) do
  j+=1
  puts j
end
```

С помощью `yield` можно также передать параметры, которые будут подставлены в список параметров блока (между вертикальными черточками). В следующем искусственном примере итератор всего лишь генерирует целые числа от 1 до 10, а вызов итератора порождает кубические степени этих чисел:

```
def my_sequence
  for i in 1..10 do
    yield i
  end
end

my_sequence { |x| puts x**3 }
```

Отметим, что вместо фигурных скобок, в которые заключен блок, можно написать ключевые слова `do` и `end`. Различия между этими формами есть, но довольно тонкие.

1.2.7. Исключения

Как и многие другие современные языки, Ruby поддерживает исключения.

Исключения – это механизм обработки ошибок, имеющий существенные преимущества по сравнению с прежними подходами. Нам удастся избежать возврата кодов ошибок и запутанной логики их анализа, а код, который обнаруживает ошибку, можно отделить от кода, который ее обрабатывает (чаще всего они так или иначе разделены).

Предложение `raise` возбуждает исключение. Отметим, что `raise` – не зарезервированное слово, а метод модуля `Kernel`. (У него есть синоним `fail`.)

```
raise # Пример 1
raise "Произошла ошибка" # Пример 2
raise ArgumentError # Пример 3
raise ArgumentError, "Неверные данные" # Пример 4
raise ArgumentError.new("Неверные данные ") # Пример 5
raise ArgumentError, " Неверные данные ", caller[0] # Пример 6
```

В первом примере повторно возбуждается последнее встретившееся исключение. В примере 2 создается исключение `RuntimeError` (подразумеваемый тип), которому передается сообщение "Произошла ошибка".

В примере 3 возбуждается исключение типа `ArgumentError`, а в примере 4 такое же исключение, но с сообщением "Неверные данные". Пример 5 – просто другая запись примера 4. Наконец, в примере 6 еще добавляется трассировочная информация вида `"filename:line"` или `"filename:line:in 'method'"` (хранящаяся в массиве `caller`).

А как обрабатываются исключения в Ruby? Для этой цели служит блок `begin...end`. В простейшей форме внутри него нет ничего, кроме кода:

```
begin
  # Ничего полезного.
  # ...
end
```

Просто перехватывать ошибки не очень осмысленно. Но у блока может быть один или несколько обработчиков `rescue`. Если произойдет ошибка в любой точке программы между `begin` и `rescue`, то управление сразу будет передано в подходящий обработчик `rescue`.

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue ArgumentError
  puts "Ошибка при извлечении квадратного корня."
rescue ZeroDivisionError
  puts "Попытка деления на нуль."
end
```

Того же эффекта можно достичь следующим образом:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue => err
  puts err
end
```

Здесь в переменной `err` хранится объект-исключение; при выводе ее на печать объект будет преобразован в осмысленную символьную строку. Отметим, что коль скоро тип ошибки не указан, то этот обработчик `rescue` будет перехватывать все исключения, производные от класса `StandardError`. В конструкции `rescue => variable` можно перед символом `=>` дополнительно указать тип ошибки.

Если типы ошибок указаны, то может случиться так, что тип реально возникшего исключения не совпадает ни с одним из них. На этот случай после всех обработчиков `rescue` разрешается поместить ветвь `else`.

```
begin
  # Код, в котором может возникнуть ошибка...
rescue Type1
  # ...
rescue Type2
  # ...
else
  # Прочие исключения...
end
```

Часто мы хотим каким-то образом восстановиться после ошибки. В этом поможет ключевое слово `retry` (внутри тела обработчика `rescue`). Оно позволяет повторно войти в блок `begin` и попытаться еще раз выполнить операцию:

```
begin
  # Код, в котором может возникнуть ошибка...
```

```
rescue
  # Пытаемся восстановиться...
  retry # Попробуем еще раз.
end
```

Наконец, иногда необходим код, который «подчищает» что-то после выполнения блока `begin-end`. В этом случае можно добавить часть `ensure`:

```
begin
  # Код, в котором может возникнуть ошибка...
rescue
  # Обработка исключений.
ensure
  # Этот код выполняется в любом случае.
end
```

Код, помещенный внутрь части `ensure`, выполняется при любом способе выхода из блока `begin-end` — вне зависимости от того, произошло исключение или нет.

Исключения можно перехватывать еще двумя способами. Во-первых, существует форма `rescue` в виде модификатора:

```
x = a/b rescue puts("Деление на ноль!")
```

Кроме того, тело определения метода представляет собой неявный блок `begin-end`; слово `begin` опущено, а все тело метода подготовлено к обработке исключения и завершается словом `end`:

```
def some_method
  # Код...
rescue
  # Восстановление после ошибки...
end
```

На этом мы завершаем как обсуждение обработки исключений, так и рассмотрение основ синтаксиса и семантики в целом.

У Ruby есть многочисленные аспекты, которых мы не коснулись. Оставшаяся часть главы посвящена более развитым возможностям языка, в том числе рассмотрению ряда практических приемов, которые помогут программисту среднего уровня научиться «думать на Ruby».

1.3. ООП в Ruby

В языке Ruby есть все элементы, которые принято ассоциировать с объектно-ориентированными языками: объекты с инкапсуляцией и сокрытием данных, методы с полиморфизмом и переопределением, классы с иерархией и наследованием. Но Ruby идет дальше, добавляя ограниченные возможности создания метаклассов, синглетные методы, модули и классы-примеси.

Похожие идеи, только под иными именами, встречаются и в других объектно-ориентированных языках, но одно и то же название может скрывать тонкие различия. В этом разделе мы уточним, что в Ruby понимается под каждым из элементов ООП.

1.3.1. Объекты

В Ruby все числа, строки, массивы, регулярные выражения и многие другие сущности фактически являются объектами. Работа программы состоит в вызове методов разных объектов:

```
3.succ           # 4
"abc".upcase     # "ABC"
[2,1,5,3,4].sort # [1,2,3,4,5]
someObject.someMethod # какой-то результат
```

В Ruby каждый объект представляет собой экземпляр какого-то класса. Класс содержит реализацию методов:

```
"abc".class      # String
"abc".class.class # Class
```

Помимо инкапсуляции собственных атрибутов и операций объект в Ruby имеет уникальный идентификатор:

```
"abc".object_id # 53744407
```

Этот идентификатор объекта обычно не представляет интереса для программиста.

1.3.2. Встроенные классы

Свыше 30 классов уже встроено в Ruby. Как и во многих других объектно-ориентированных языках, в нем не допускается множественное наследование, но это еще не означает, что язык стал менее выразительным. Современные языки часто построены согласно модели одиночного наследования. Ruby поддерживает модули и классы-примеси, которые мы обсудим в следующей главе. Также реализованы идентификаторы объектов, что позволяет строить устойчивые, распределенные и перемещаемые объекты.

Для создания объекта существующего класса обычно используется метод `new`:

```
myFile = File.new("textfile.txt", "w")
myString = String.new("Это строковый объект")
```

Однако не всегда его обязательно вызывать явно. В частности, при создании объекта `String` можно и не упоминать этот метод:

```
yourString = "Это тоже строковый объект"
aNumber = 5 # и здесь метод new не нужен
```

Ссылки на объекты хранятся в переменных. Выше уже отмечалось, что сами переменные не имеют типа и не являются объектами – они лишь ссылаются на объекты.

```
x = "abc"
```

Из этого правила есть исключение: небольшие неизменяемые объекты некоторых встроенных классов, например `Fixnum`, непосредственно копируются в переменные, которые на них ссылаются. (Размер этих объектов не превышает размера указателя, поэтому хранить их таким образом более эффективно.) В таком случае во время присваивания делается копия объекта, а куча не используется.

При присваивании переменных ссылки на объекты обобществляются.

```
y = "abc"
x = y
x                                     # "abc"
```

После выполнения присваивания `x = y` и `x`, и `y` ссылаются на один и тот же объект:

```
x.object_id      # 53732208
y.object_id      # 53732208
```

Если объект изменяемый, то модификация, примененная к одной переменной, отражается и на другой:

```
x.gsub!(/a/, "x")
y                                     # "xbc"
```

Однако новое присваивание любой из этих переменных не влияет на другую:

```
# Продолжение предыдущего примера
x = "abc"
```

```
y                                     # по-прежнему равно "xbc"
```

Изменяемый объект можно сделать неизменяемым, вызвав метод `freeze`:

```
x.freeze
x.gsub!(/b/, "y")      # Ошибка!
```

Символ в Ruby ссылается на переменную по имени, а не по ссылке. Во многих случаях он может вообще не ссылаться на идентификатор, а вести себя как некая разновидность неизменяемой строки. Символ можно преобразовать в строку с помощью метода `to_s`.

```
Hearts  = :Hearts  # Это один из способов присвоить
Clubs   = :Clubs   # уникальное значение константе,
Diamonds = :Diamonds # некий аналог перечисления
Spades  = :Spades  # в языках Pascal или C.
```

```
puts Hearts.to_s      # Печатается "Hearts"
```

Продemonстрированный выше фокус с «перечислением» был более осмыслен на ранних этапах развития Ruby, когда еще не было класса `Symbol`, а наличие двоеточия перед идентификатором превращало его в целое число. Если вы пользуетесь таким трюком, не предполагайте, что фактическое значение символа будет неизменным или предсказуемым – просто используйте его как константу, значение которой неважно.

1.3.3. Модули и классы-примеси

Многие встроенные методы наследуются от классов-предков. Особо стоит отметить методы модуля `Kernel`, подмешиваемые к суперклассу `Object`. Поскольку класс `Object` повсеместно доступен, то и добавленные в него из `Kernel` методы также доступны в любой точке программы. Эти методы играют важную роль в Ruby.

Термины «модуль» и «примесь» – почти синонимы. *Модуль* представляет собой набор методов и констант, внешних по отношению к программе на Ruby. Его можно использовать просто для управления пространством имен, но основное применение модулей связано с «подмешиванием» его возможностей в класс (с помощью директивы `include`). В таком случае он используется как *класс-примесь*.

Этот термин очевидно заимствован из языка Python. Стоит отметить, что в некоторых вариантах LISP такой механизм существует уже больше двадцати лет.

Не путайте описанное выше употребление термина «модуль» с другим значением, которое часто придается ему в информатике. Модуль в Ruby – это не внешний исходный текст и не двоичный файл (хотя может храниться и в том, и в другом виде). Это объектно-ориентированная абстракция, в чем-то похожая на класс.

Примером использования модуля для управления пространством имен служит модуль `Math`. Так, чтобы получить определение числа π , необязательно включать модуль `Math` с помощью предложения `include`; достаточно просто написать `Math::PI`.

Примесь дает способ получить преимущества множественного наследования, неотягощенные характерными для него проблемами. Можно считать, что это ограниченная форма множественного наследования, но создатель языка Мац называет его одиночным наследованием с разделением реализации.

Отметим, что предложение `include` включает имена из указанного пространства имен (модуля) в текущее. Метод `extend` добавляет объекту функции из модуля. В случае применения `include` методы модуля становятся доступны как методы экземпляра, а в случае `extend` – как методы класса.

Необходимо оговориться, что операции `load` и `require` не имеют ничего общего с модулями: они относятся к исходным и двоичным файлам (загружаемым динамически или статически). Операция `load` читает файл и вставляет его в текущую точку исходного текста, так что начиная с этой точки становятся видимы все определения, находящиеся во внешнем файле. Операция `require` аналогична `load`, но не загружает файл, если он уже был загружен ранее.

Программисты, только начинающие осваивать Ruby, особенно имеющие опыт работы с языком C, могут поначалу путать операции `require` и `include`, которые никак не связаны между собой. Вы еще поймете себя на том, что сначала вызываете `require`, а потом `include` для того, чтобы воспользоваться каким-то внешним модулем.

1.3.4. Создание классов

В Ruby есть множество встроенных классов, и вы сами можете определять новые. Для определения нового класса применяется такая конструкция:

```
class ClassName
# ...
end
```

Само имя класса – это глобальная константа, поэтому оно должно начинаться с прописной буквы. Определение класса может содержать константы, переменные класса, методы класса, переменные экземпляра и методы экземпляра. Данные

уровня класса доступны всем объектам этого класса, тогда как данные уровня экземпляра доступны только одному объекту.

Попутное замечание: строго говоря, классы в Ruby не имеют имен. «Имя» класса – это всего лишь константа, ссылающаяся на объект типа Class (поскольку в Ruby class – это класс). Ясно, что на один и тот же класс могут ссылаться несколько констант, и их можно присваивать переменным точно так же, как мы поступаем с любыми другими объектами (поскольку в Ruby class – это объект). Если вы немного запутались, не расстраивайтесь. Удобства ради новичок может считать, что в Ruby имя класса – то же самое, что в C++.

Вот как определяется простой класс:

```
class Friend
  @myname = "Эндрю" # переменная класса

  def initialize(name, sex, phone)
    @name, @sex, @phone = name, sex, phone
    # Это переменные экземпляра
  end

  def hello # метод экземпляра
    puts "Привет, я #{@name}."
  end

  def Friend.our_common_friend # метод класса
    puts "Все мы друзья #{@myname}."
  end
end

f1 = Friend.new("Сюзанна", "F", "555-0123")
f2 = Friend.new("Том", "M", "555-4567")

f1.hello # Привет, я Сюзанна.
f2.hello # Привет, я Том.
Friend.our_common_friend # Все мы друзья Эндрю.
```

Поскольку данные уровня класса доступны во всем классе, их можно инициализировать в момент определения класса. Если определен метод с именем `initialize`, то гарантируется, что он будет вызван сразу после выделения памяти для объекта. Этот метод похож на традиционный конструктор, но не выполняет выделения памяти. Память выделяется методом `new`, а освобождается неявно сборщиком мусора.

Теперь взгляните на следующий фрагмент, обращая особое внимание на методы `getmyvar`, `setmyvar` и `myvar=`:

```
class MyClass

  NAME = "Class Name" # константа класса
  @@count = 0 # инициализировать переменную класса
  def initialize # вызывается после выделения памяти для объекта
```

```

    @@count += 1
    @myvar = 10
end

def MyClass.getcount # метод класса
    @@count          # переменная класса
end

def getcount          # экземпляр возвращает переменную класса!
    @count            # переменная класса
end

def getmyvar          # метод экземпляра
    @myvar            # переменная экземпляра
end

def setmyvar(val)     # метод экземпляра устанавливает @myvar
    @myvar = val
end
def myvar=(val)       # другой способ установить @myvar
    @myvar = val
end
end

foo = MyClass.new      # @myvar равно 10
foo.setmyvar 20        # @myvar равно 20
foo.myvar = 30         # @myvar равно 30

```

Здесь мы видим, что `getmyvar` возвращает значение переменной `@myvar`, а `setmyvar` устанавливает его. (Многие программисты говорят о методах чтения и установки). Все это работает, но не является характерным способом действий в Ruby. Метод `myvar=` похож на перегруженный оператор присваивания (хотя, строго говоря, таковым не является); это более удачная альтернатива `setmyvar`, но есть способ еще лучше.

Класс `Module` содержит методы `attr`, `attr_accessor`, `attr_reader` и `attr_writer`. Ими можно пользоваться (передавая символы в качестве параметров) для автоматизации управления доступом к данным экземпляра. Например, все три метода `getmyvar`, `setmyvar` и `myvar=` можно заменить одной строкой в определении класса:

```
attr_accessor :myvar
```

При этом создается метод `myvar`, который возвращает значение `@myvar`, и метод `myvar=`, который позволяет изменить значение той же переменной. Методы `attr_reader` и `attr_writer` создают соответственно версии методов доступа к атрибуту для чтения и для изменения.

Внутри методов экземпляра, определенных в классе, можно при необходимости пользоваться переменной `self`. Это просто ссылка на объект, от имени которого вызван метод экземпляра.

Для управления видимостью методов класса можно пользоваться модификаторами `private`, `protected` и `public`. (Переменные экземпляра всегда закрыты, обращаться к ним извне класса можно только с помощью методов доступа.) Каждый модификатор принимает в качестве параметра символ, например `:foo`, а если он опущен, то действие модификатора распространяется на все последующие определения в классе. Пример:

```
class MyClass

  def method1
    # ...
  end
  def method2
    # ...
  end

  def method3
    # ...
  end

  private :method1
  public

  :method2
  protected :method3

  private

  def my_method
    # ...
  end

  def another_method
    # ...
  end

end
```

В этом классе метод `method1` закрытый, `method2` открытый, а `method3` защищенный. Поскольку далее вызывается метод `private` без параметров, то методы `my_method` и `another_method` будут закрытыми.

Уровень доступа `public` не нуждается в объяснениях, он не налагает никаких ограничений ни на доступ к методу, ни на его видимость. Уровень `private` означает, что метод доступен исключительно внутри класса или его подклассов и может вызываться только в «функциональной форме» от имени `self`, причем вызывающий объект может указываться явно или подразумеваться неявно. Уровень `protected` означает, что метод вызывается только внутри класса, но, в отличие от закрытого метода, не обязательно от имени `self`.

По умолчанию все определенные в классе методы открыты. Исключение составляет лишь `initialize`. Методы, определенные на верхнем уровне программы, тоже по умолчанию открыты. Если они объявлены закрытыми, то могут вызываться только в функциональной форме (как, например, методы, определенные в классе `Object`).

Классы в Ruby сами являются объектами – экземплярами метакласса `Class`. Классы в этом языке всегда конкретны, абстрактных классов не существует. Однако теоретически можно реализовать и абстрактные классы, если вам это для чего-то понадобится.

Класс `Object` является корнем иерархии. Он предоставляет все методы, определенные во встроенном модуле `Kernel`.

Чтобы создать класс, наследующий другому классу, нужно поступить следующим образом:

```
class MyClass < OtherClass
  # ...
end
```

Помимо использования встроенных методов, вполне естественно определить и собственные либо переопределить унаследованные. Если определяемый метод имеет то же имя, что и существующий, то старый метод замещается. Если новый метод должен обратиться к замещенному им «родительскому» методу (так бывает часто), можно воспользоваться ключевым словом `super`.

Перегрузка операторов, строго говоря, не является неотъемлемой особенностью ООП, но этот механизм знаком программистам на C++ и некоторых других языках. Поскольку большинство операторов в Ruby так или иначе являются методами, то не должен вызывать удивления тот факт, что их можно переопределять или определять в пользовательских классах. Переопределять семантику оператора в существующем классе редко имеет смысл, зато в новых классах определение операторов – обычное дело.

Можно создавать синонимы методов. Для этого внутри определения класса предоставляется такой синтаксис:

```
alias newname oldname
```

Число параметров будет таким же, как для старого имени, и вызываться метод-синоним будет точно так же. Обратите внимание на отсутствие запятой; `alias` – это не имя метода, а ключевое слово. Существует метод с именем `alias_method`, который ведет себя аналогично, но в случае его применения параметры должны разделяться запятыми, как и для любого другого метода.

1.3.5. Методы и атрибуты

Как мы уже видели, методы обычно используются в сочетании с простыми экземплярами классов и переменными, причем вызывающий объект отделяется от имени метода точкой (`receiver.method`). Если имя метода является знаком препинания, то точка опускается. У методов могут быть аргументы:

```
Time.mktime(2000, "Aug", 24, 16, 0)
```

Поскольку каждое выражение возвращает значение, то вызовы методов могут сцепляться:

```
3.succ.to_s
/(x.z).*(z.z).*/.match("x1z_1a3_x2z_1b3_").to_a[1..3]
3+2.succ
```

Отметим, что могут возникать проблемы, если выражение, являющееся результатом сцепления, имеет тип, который не поддерживает конкретный метод. Точнее, при определенных условиях некоторые методы возвращают `nil`, а вызов любого метода от имени такого объекта приведет к ошибке. (Конечно, `nil` — полноценный объект, но он не обладает теми же методами, что и, например, массив.)

Некоторым методам можно передавать блоки. Это верно для всех итераторов — как встроенных, так и определенных пользователем. Блок обычно заключается в операторные скобки `do-end` или в фигурные скобки. Но он не рассматривается так же, как предшествующие ему параметры, если таковые существуют. Вот пример вызова метода `File.open`:

```
my_array.each do |x|
  some_action
end
```

```
File.open(filename) { |f| some_action }
```

Именованные параметры будут поддерживаться в последующих версиях Ruby, но на момент работы над этой книгой еще не поддерживались. В языке Python они называются ключевыми аргументами, сама идея восходит еще к языку Ada.

Методы могут принимать переменное число аргументов:

```
receiver.method(arg1, *more_args)
```

В данном случае вызванный метод трактует `more_args` как массив и обращается с ним, как с любым другим массивом. На самом деле звездочка в списке формальных параметров (перед последним или единственным параметром) может «свернуть» последовательность фактических параметров в массив:

```
def mymethod(a, b, *c)
  print a, b
  c.each do |x| print x end
end
```

```
mymethod(1,2,3,4,5,6,7)
```

```
# a=1, b=2, c=[3,4,5,6,7]
```

В Ruby есть возможность определять методы на уровне объекта (а не класса). Такие методы называются синглетными; они принадлежат одному-единственному объекту и не оказывают влияния ни на класс, ни на его суперклассы. Такая возможность может быть полезна, например, при разработке графических интерфейсов пользователя: чтобы определить действие кнопки, вы задаете синглетный метод для данной и только данной кнопки.

Вот пример определения синглтного метода для строкового объекта:

```
str = "Hello, world!"
str2 = "Goodbye!"

def str.spell
  self.split(/./).join("-")
end

str.spell      # "H-e-l-l-o-, -w-o-r-l-d-!"
str2.spell     # Ошибка!
```

Имейте в виду, что метод определяется для объекта, а не для переменной.

Теоретически с помощью синглтных методов можно было бы создать систему объектов на базе прототипов. Это менее распространенная форма ООП без классов^{*}. Основной структурный механизм в ней состоит в конструировании нового объекта путем использования существующего в качестве образца; новый объект ведет себя как старый за исключением тех особенностей, которые были перепределены. Тем самым можно строить системы на основе прототипов, а не наследования. Хотя у нас нет опыта в этой области, мы полагаем, что создание такой системы позволило бы полнее раскрыть возможности Ruby.

1.4. Динамические аспекты Ruby

Ruby – динамический язык в том смысле, что объекты и классы можно изменять во время выполнения. Ruby позволяет конструировать и интерпретировать фрагменты кода в ходе выполнения статически написанной программы. В нем есть хитроумный API отражения, с помощью которого программа может получать информацию о себе самой. Это позволяет сравнительно легко создавать отладчики, профилировщики и другие подобные инструменты, а также применять нетривиальные способы кодирования.

Наверное, это самая трудная тема для программиста, приступающего к изучению Ruby. В данном разделе мы вкратце рассмотрим некоторые следствия, вытекающие из динамической природы языка.

1.4.1. Кодирование во время выполнения

Мы уже упоминали директивы `load` и `require`. Важно понимать, что это не встроенные предложения и не управляющие конструкции; на самом деле это методы. Поэтому их можно вызывать, передавая переменные или выражения как параметры, в том числе условно. Сравните с директивой `#include` в языках C и C++, которая обрабатывается во время компиляции.

Код можно строить и интерпретировать по частям. В качестве несколько искусственного примера рассмотрим приведенный ниже метод `calculate` и вызывающий его код:

```
def calculate(op1, operator, op2)
  string = op1.to_s + operator + op2.to_s
```

^{*} Типичный пример – язык JavaScript (*Прим. перев.*)

```
# Предполагается, что operator - строка; построим длинную
# строку, состоящую из оператора и операндов.
eval(string) # Вычисляем и возвращаем значение.
end
```

```
@alpha = 25
@beta = 12
```

```
puts calculate(2, "+", 2) # Печатается 4
puts calculate(5, "*", "@alpha") # Печатается 125
puts calculate("@beta", "**", 3) # Печатается 1728
```

Вот та же идея, доведенная чуть ли не до абсурда: программа запрашивает у пользователя имя метода и одну строку кода. Затем этот метод определяется и вызывается:

```
puts "Имя метода: "
meth_name = gets
puts "Строка кода: "
code = gets

string = %[def #{meth_name}\n #{code}\n end] # Строим строку.
eval(string) # Определяем метод.
eval(meth_name) # Вызываем метод.
```

Зачастую необходимо написать программу, которая могла бы работать на разных платформах или при разных условиях, но при этом сохранить общий набор исходных текстов. Для этого в языке С применяются директивы `#ifdef`, но в Ruby все определения исполняются. Не существует такого понятия, как «этап компиляции»; все конструкции динамические, а не статические. Поэтому для принятия решения такого рода мы можем просто вычислить условие во время выполнения:

```
if platform == Windows
  action1
elsif platform == Linux
  action2
else
  default_action
end
```

Конечно, за такое кодирование приходится расплачиваться некоторым снижением производительности, поскольку иногда условие вычисляется много раз. Но рассмотрим следующий пример, который делает практически то же самое, однако весь платформенно-зависимый код помещен в один метод, имя которого от платформы не зависит:

```
if platform == Windows
  def my_action
    action1
  end
elsif platform == Linux
  def my_action
```

```
    action2
  end
else
  def my_action
    default_action
  end
end
```

Таким способом мы достигаем желаемого результата, но условие вычисляется только один раз. Когда программа вызовет метод `my_action`, он уже будет правильно определен.

1.4.2. Отражение

В языках Smalltalk, LISP и Java реализована (с разной степенью полноты) идея рефлексивного программирования – активная среда может опрашивать структуру объектов и расширять либо модифицировать их во время выполнения.

В языке Ruby имеется развитая поддержка отражения, но все же он не заходит так далеко, как Smalltalk, где даже управляющие конструкции являются объектами. В Ruby управляющие конструкции и блоки не представляют собой объекты. (Объект `Proc` можно использовать для того, чтобы представить блок в виде объекта, но управляющие конструкции объектами не бывают никогда.)

Для определения того, используется ли идентификатор с данным именем, служит ключевое слово `defined?` (обратите внимание на вопросительный знак в конце слова):

```
if defined? some_var
  puts "some_var = #{some_var}"
else
  puts "Переменная some_var неизвестна."
end
```

Аналогично метод `respond_to?` выясняет, может ли объект отвечать на вызов указанного метода (то есть определен ли данный метод для данного объекта). Метод `respond_to?` определен в классе `Object`.

В Ruby запрос информации о типе во время выполнения поддерживается очень полно. Тип или класс объекта можно определить, воспользовавшись методом `type` (из класса `Object`). Метод `is_a?` сообщает, принадлежит ли объект некоторому классу (включая и его суперклассы); синонимом служит имя `kind_of?`. Например:

```
puts "abc".class "          # Печатается String
puts 345.class   # Печатается Fixnum
rover = Dog.new

print rover.class          # Печатается Dog

if rover.is_a? Dog
  puts "Конечно, является."
end

if rover.kind_of? Dog
  puts "Да, все еще собака."
```

```
end

if rover.is_a? Animal
  puts "Да, он к тому же и животное."
end
```

Можно получить полный список всех методов, которые можно вызывать для данного объекта. Для этого предназначен метод `methods` из класса `Object`. Имеются также его варианты `private_instance_methods`, `public_instance_methods` и т. д.

Аналогично можно узнать, какие переменные класса или экземпляра ассоциированы с данным объектом. По самой природе ООП в перечни методов и переменных включаются те, что определены как в классе самого объекта, так и во всех его суперклассах. В классе `Module` имеется метод `constants`, позволяющий получить список всех констант, определенных в модуле.

В классе `Module` есть метод `ancestors`, возвращающий список модулей, включенных в данный модуль. В этот список входит и сам данный модуль, то есть список, возвращаемый вызовом `mod.ancestors`, содержит по крайней мере элемент `mod`. В этот список входят не только родительские классы (отобранные в силу наследования), но и «родительские» модули (отобранные в силу включения).

В классе `Object` есть метод `superclass`, который возвращает суперкласс объекта или `nil`. Не имеет суперкласса лишь класс `Object`, и, значит, только для него может быть возвращен `nil`.

Модуль `ObjectSpace` применяется для получения доступа к любому «живому» объекту. Метод `_id2ref` преобразует идентификатор объекта в ссылку на него; можно считать, что это операция, обратная той, что выполняет двоеточие в начале имени. В модуле `ObjectSpace` есть также итератор `each_object`, который перебирает все существующие в данный момент объекты, включая и те, о которых иным образом узнать невозможно. (Напомним, что некоторые неизменяемые объекты небольшого размера, например принадлежащие классам `Fixnum`, `NilClass`, `TrueClass` и `FalseClass`, не хранятся в куче из соображений оптимизации.)

1.4.3. Отсутствующие методы

При вызове метода (`myobject.my_method`) Ruby ищет поименованный метод в следующем порядке:

1. Синглетные методы, определенные для объекта `myobject`.
2. Методы, определенные в классе объекта `myobject`.
3. Методы, определенные в предках класса объекта `myobject`.

Если найти метод `my_method` не удастся, Ruby ищет метод с именем `method_missing`. Если он определен, то ему передается имя отсутствующего метода (в виде символа) и все переданные ему параметры. Этот механизм можно применять для динамической обработки неизвестных сообщений, посланных во время выполнения.

1.4.4. Сборка мусора

Управлять памятью на низком уровне трудно и чревато ошибками, особенно в таком динамичном окружении, какое создает Ruby. Наличие механизма сборки мусора –

весомое преимущество. В таких языках, как C++, за выделение и освобождение памяти отвечает программист. В более поздних языках, например Java, память освобождается сборщиком мусора (когда объект покидает область видимости).

Явное управление памятью может приводить к двум видам ошибок. Если освобождается память, занятая объектом, на который еще есть ссылки, то при последующем доступе к нему объект может оказаться в противоречивом состоянии. Так называемые *висячие указатели* трудно отлаживать, поскольку вызванные ими ошибки часто проявляются далеко от места возникновения. Утечка памяти имеет место, когда не освобождается объект, на который больше никто не ссылается. В этом случае программа потребляет все больше и больше памяти и в конечном счете аварийно завершается; такие ошибки искать тоже трудно. В языке Ruby для отслеживания неиспользуемых объектов и освобождения занятой ими памяти применяется механизм сборки мусора. Для тех, кто в этом разбирается, отметим, что в Ruby используется алгоритм пометки и удаления, а не подсчета ссылок (у последнего возникают трудности при обработке рекурсивных структур).

Сборка мусора влечет за собой некоторое снижение производительности. Модуль GC предоставляет ограниченные средства управления, позволяющие программисту настроить его работу в соответствии с нуждами конкретной программы. Можно также определить чистильщика (finalizer) объекта, но это уже тема для «продвинутых» (см. раздел 11.3.14).

1.5. Потренируйте свою интуицию:

что следует запомнить

Надо честно признаться: «все становится интуитивно ясным после того, как поймешь». Эта истина и составляет суть данного раздела, поскольку в Ruby немало особенностей, отличающих его от всего, к чему привык программист на одном из традиционных языков.

Кто-то из читателей решит, что не нужно зря тратить время на повторение известного. Если вы из их числа, можете пропустить разделы, содержание которых кажется вам очевидным. Программисты имеют неодинаковый опыт; искушенные пользователи C и Smalltalk воспримут Ruby совсем по-разному. Впрочем, мы надеемся, что внимательное прочтение последующих разделов поможет многим читателям разобраться в том, что же такое Путь Ruby.

1.5.1. Синтаксис

Синтаксический анализатор Ruby сложен и склонен прощать многие огрехи. Он пытается понять, что хотел сказать программист, а не навязывать ему жесткие правила. Но к такому поведению надо еще привыкнуть. Вот перечень того, что следует знать о синтаксисе Ruby.

- Скобки при вызове методов, как правило, можно опускать. Все следующие вызовы допустимы:

```
foobar
foobar()
```



```
foobar(a,b,c)
foobar a, b, c
```

- Коль скоро скобки необязательны, что означает такая запись: `x y z`? Оказывается, вот что: «Вызвать метод `y`, передав ему параметр `z`, а результат передать в виде параметра методу `x`.» Иными словами, `x(y(z))`. Это поведение в будущем изменится. См. обсуждение поэтического режима в разделе 1.6 ниже.

- Попробуем передать методу хэш:

```
my_method {a=>1, b=>2}
```

Это приведет к синтаксической ошибке, поскольку левая фигурная скобка воспринимается как начало блока. В данном случае скобки необходимы:

```
my_method({a=>1, b=>2})
```

- Предположим теперь, что хэш – единственный (или последний) параметр метода. Ruby снисходительно разрешает опускать фигурные скобки:

```
my_method(a=>1, b=>2)
```

Кто-то увидит здесь вызов метода с именованными параметрами. Это обманчивое впечатление, хотя никто не запрещает применять подобную конструкцию и в таком смысле.

- Есть и другие случаи, когда пропуски имеют некоторое значение. Например, на первый взгляд все четыре выражения ниже означают одно и то же:

```
x = y + z
x = y+z
x = y+ z
x = y +z
```

Но фактически эквивалентны лишь первые три. В четвертом же случае анализатор считает, что вызван метод `y` с параметром `+z`! И выдаст сообщение об ошибке, так как метода с именем `y` не существует. Мораль: пользуйтесь пробелами разумно.

- Аналогично `x = y*z` – это умножение `y` на `z`, тогда как `x = y *z` – вызов метода `y`, которому в качестве параметра передается расширение массива `z`.
- В именах идентификаторов знак подчеркивания `_` считается строчной буквой. Следовательно, имя идентификатора может начинаться с этого знака, но такой идентификатор не будет считаться константой, даже если следующая буква прописная.
- В линейной последовательности вложенных предложений `if` применяется ключевое слово `elsif`, а не `else if` или `elif`, как в некоторых других языках.
- Ключевые слова в Ruby нельзя назвать по-настоящему зарезервированными. Если метод вызывается от имени некоторого объекта (и в других случаях, когда не возникает неоднозначности), имя метода может совпадать с ключевым словом. Но поступайте так с осторожностью, не забывая, что программу будут читать люди.

- Ключевое слово `then` (в предложениях `if` и `case`) необязательно. Если вам кажется, что с ним программа понятнее, включайте его в код. То же относится к слову `do` в циклах `while` и `until`.
- Вопросительный и восклицательный знаки не являются частью идентификатора, который модифицируют, – их следует рассматривать как суффиксы. Таким образом, хотя идентификаторы `chop` и `chop!` считаются различными, использовать восклицательный знак в любом другом месте имени не разрешается. Аналогично в Ruby есть конструкция `defined?`, но `defined` – ключевое слово.
- Внутри строки символ решетки `#` – признак начала выражения. Значит, в некоторых случаях его следует экранировать обратной косой чертой, но лишь тогда, когда сразу за ним идет символ `{`, `$` или `@`.
- Поскольку вопросительный знак можно добавлять в конец идентификатора, то следует аккуратно расставлять пробелы в тернарном операторе. Пусть, например, имеется переменная `my_flag`, которая может принимать значения `true` или `false`. Тогда первое из следующих предложений правильно, а второе содержит синтаксическую ошибку:

```
x = my_flag? 23 : 45          # Правильно.  
x = my_flag? 23 : 45          # Синтаксическая ошибка.
```

- Концевой маркер для встроенной документации не следует считать лексемой. Он помечает строку целиком, поэтому все находящиеся в той же строке символы не являются частью текста программы, а принадлежат встроенному документу.
- В Ruby нет произвольных блоков, то есть нельзя начать блок в любом месте, как в C. Блоки разрешены только там, где они нужны, – например, могут присоединяться к итератору. Исключение составляет блок `begin-end`, который можно употреблять практически везде.
- Не забывайте, что ключевые слова `BEGIN` и `END` не имеют ничего общего с `begin` и `end`.
- При статической конкатенации строк приоритет конкатенации ниже, чем у вызова метода. Например:

```
str = "Первая " + 'second'.center(20)    # Примеры 1 and 2  
str = "Вторая " + 'second'.center(20)    # дадут одно и то же.  
str = "Первая вторая".center(20)         # Примеры 3 and 4  
str = ("Первая " + 'вторая').center(20)  # дадут одно и то же.
```

- В Ruby есть несколько псевдопеременных, которые выглядят как локальные переменные, но применяются для особых целей. Это `self`, `nil`, `true`, `false`, `__FILE__` и `__LINE__`.

1.5.2. Перспективы программирования

Наверное, каждый, кто знает Ruby (сегодня), в прошлом изучал или пользовался другими языками. Это, с одной стороны, облегчает изучение Ruby, так как многие

средства похожи на аналогичные средства в других языках. С другой стороны, у программиста может возникнуть ложное чувство уверенности при взгляде на знакомые конструкции Ruby. Он может прийти к неверным выводам, основанным на прошлом опыте; можно назвать это явление «багажом эксперта».

Немало специалистов переходит на Ruby со Smalltalk, Perl, C/C++ и других языков. Ожидания этих людей сильно различаются, но так или иначе присутствуют. Поэтому рассмотрим некоторые вещи, на которых многие спотыкаются.

- Символ в Ruby представляется целым числом. Это не самостоятельный тип, как в Pascal, и не эквивалент строки длиной 1. В ближайшем будущем положение изменится и символьная константа станет строкой, но на момент написания данной книги этого еще не произошло. Рассмотрим следующий фрагмент:

```
x = "Hello"
y = ?A
puts "x[0] = #{x[0]}"    # Печатается x[0] = 72
puts "y = #{y}"          # Печатается y = 65
if y == "A"              # Печатается no
  puts "yes"
else
  puts "no"
end
```

- Не существует булевого типа. TrueClass и FalseClass – это два разных класса, а единственными их экземплярами являются объекты true и false.
- Многие операторы в Ruby напоминают операторы в языке C. Два заметных исключения – операторы инкремента и декремента (++ и --). Их в Ruby нет ни в «пост», ни в «пред» форме.
- Известно, что в разных языках оператор деления по модулю работает по-разному для отрицательных чисел. Не вдаваясь в споры о том, что правильно, проиллюстрируем поведение в Ruby:

```
puts (5 % 3)      # Печатается 2
puts (-5 % 3)     # Печатается 1
puts (5 % -3)     # Печатается -1
puts (-5 % -3)    # Печатается -2
```

- Некоторые привыкли думать, что «ложь» можно представлять нулем, пустой строкой, нулевым символом и т. п. Но в Ruby все это равно «истине». На самом деле истиной будет все кроме объектов false и nil.
- В Ruby переменные не принадлежат никакому классу: класс есть только у значений.
- Переменные в Ruby не объявляются, однако считается хорошим тоном присваивать переменной начальное значение nil. Разумеется, при этом с переменной не ассоциируется никакой тип и даже не происходит истинной инициализации, но анализатор знает, что данное имя принадлежит переменной, а не методу.

- ARGV[0] – первый аргумент в командной строке; они нумеруются начиная с нуля. Это не имя файла или сценария, предшествующего параметрам, как argv[0] в языке C.
- Большинство операторов в Ruby на самом деле является методами; их запись в виде «знаков препинания» – не более чем удобство. Первое исключение из этого правила – набор операторов составного присваивания (+=, -= и т.д.). Второе исключение – операторы =, .., ..., !, not, &&, and, ||, or, !=, !~.
- Как и в большинстве современных языков программирования (хотя и не во всех), булевские операции закорачиваются, то есть вычисление булевского выражения заканчивается, как только его значение истинности становится известным. В последовательности операций or вычисление заканчивается, когда получено первое значение true, а в последовательности операций and – когда получено первое значение false.
- Префикс @@ применяется для переменных класса (то есть ассоциированных с классом в целом, а не с отдельным экземпляром).
- loop – не ключевое слово. Это метод модуля Kernel, а не управляющая конструкция.
- Кому-то синтаксис unless-else может показаться интуитивно неочевидным. Поскольку unless – противоположность if, то ветвь else выполняется, когда условие истинно.
- Простой тип Fixnum передается как непосредственное значение и, стало быть, не может быть изменен внутри метода. То же относится к значениям true, false и nil.
- Не путайте операторы && и || с операторами & и |. Те и другие используются в языке C; первые два предназначены для логических операций, последние два – для поразрядных.
- Операторы and и or имеют более низкий приоритет, чем && и ||. Взгляните на следующий фрагмент:

```
a = true
b = false
c = true
d = true
a1 = a && b or c && d      # Операции && выполняются первыми.
a2 = a && (b or c) && d     # Операция or выполняется первой.
puts a1                   # Печатается false
puts a2                   # Печатается true
```

- Не забывайте, что «оператор» присваивания имеет более высокий приоритет, чем операторы and и or! (это относится и к составным операторам присваивания: +=, -= и пр.). Например, код x = y or z выглядит как обычное предложение присваивания, но на самом деле это обособленное выражение (эквивалент (x=y) or z). Вероятно, программист имел в виду следующее: x = (y or z).

```

y = false
z = true

x = y or z      # Оператор = выполняется РАНЬШЕ or!
puts x          # Печатается false

(x = y) or z    # Строка 5: то же, что и выше.
puts x          # Печатается false

x = (y or z)    # Оператор or вычисляется сначала.
puts x          # Печатается true

```

- Не путайте атрибуты объектов с локальными переменными. Если вы привыкли к C++ или Java, можете забыть об этом! Переменная `@my_var` в контексте класса – это переменная экземпляра (или атрибут), но `my_var` в том же контексте – локальная переменная.
- Во многих языках, и в Ruby в том числе, есть цикл `for`. Рано или поздно возникает вопрос, можно ли модифицировать индексную переменную. В некоторых языках эту управляющую переменную запрещено изменять вовсе (выводится предупреждение либо сообщение об ошибке на этапе компиляции или выполнения); в других это допустимо, хотя и приводит к изменению поведения цикла. В Ruby принят третий подход. Переменная, управляющая циклом `for`, считается обычной переменной, которую можно изменять в любой момент, но это изменение не оказывает влияния на поведение цикла! Цикл `for` присваивает этой переменной последовательные значения, что бы с ней ни происходило внутри тела цикла. Например, следующий цикл будет выполнен ровно 10 раз и напечатает значения от 1 до 10:

```

for var in 1..10
  puts "var = #{var}"
  if var > 5
    var = var + 2
  end
end

```

- Имена переменных не всегда легко «на глаз» отличить от имен методов. Как решает этот вопрос анализатор? Правило такое: если анализатор видит, что идентификатору присваивается значение до его использования, то он считается переменной; в противном случае это имя метода. (Отметим, что операция присваивания может и не выполняться: достаточно того, что интерпретатор ее видел.)

1.5.3. Предложение `case` в Ruby

Во всех современных языках есть та или иная форма многопутевого ветвления. В C/C++ и Java это предложение `switch`, а в Pascal – предложение `case`. Служат они одной и той же цели и функционируют примерно одинаково.

Предложение `case` в Ruby похоже, но при ближайшем рассмотрении оказывается настолько уникальным, что варианты ветвления, принятые в C и в Pascal.

кажутся близкими родственниками. Точного аналога предложению `case` в Ruby нет ни в каком другом знакомом мне языке, поэтому оно заслуживает особого внимания.

Выше мы уже рассматривали синтаксис этого предложения, а теперь сосредоточимся на его семантике.

- Для начала рассмотрим тривиальный пример. Выражение `expression` сравнивается со значением `value`, и, если они совпадают, выполняется некоторое действие. Ничего удивительного.

```
case expression
  when value
    некоторое действие
end
```

В Ruby для этой цели есть специальный оператор `===` (называется *оператором отношения*). Иногда его еще называют (не совсем правильно) оператором ветвящегося равенства. Неправильность в том, что он не всегда относится именно к проверке на равенство.

- Предыдущее предложение можно записать и так:

```
if value === expression
  некоторое действие
end
```

- Не путайте оператор отношения с оператором проверки на равенство (`==`). Они принципиально различны, хотя во многих случаях ведут себя одинаково. Оператор отношения определен по-разному в разных классах, а для данного класса его поведение может зависеть от типа переданного операнда.
- Не думайте, что проверяемое выражение – это объект, которому сравниваемое значение передается в качестве параметра. На самом деле как раз наоборот (мы это только что видели).
- Это подводит нас к наблюдению, что `x === y` означает вовсе не то же самое, что `y === x`! Иногда результат совпадает, но в общем случае оператор отношения не коммутативен. (Именно поэтому нам не нравится термин «оператор ветвящегося равенства» – ведь проверка на равенство всегда коммутативна.) Если перевернуть исходный пример, окажется, что следующий код ведет себя иначе:

```
case value
  when expression
    некоторое действие
end
```

- В качестве примера рассмотрим строку `str` и образец (регулярное выражение) `pat`, с которым эта строка успешно сопоставляется. Выражение `str =~ pat` истинно, как в языке Perl. Поскольку Ruby определяет противоположную семантику оператора `==` в классе `Regexp`, можно также сказать, что выражение `pat == str` истинно. Следуя той же логике,

мы обнаруживаем, что истинно и `pat === str` (исходя из того, как определен оператор `===` в классе `Regexp`). Однако выражение `str === pat` истинным не является. А значит, фрагмент

```
case "Hello"
  when /Hell/
    puts "Есть соответствие."
else
  puts "Нет соответствия."
end
```

делает не то же самое, что фрагмент

```
case /Hell/
  when "Hello"
    puts "Есть соответствие."
else
  puts "Нет соответствия."
end
```

Если это вас смущает, просто постарайтесь запомнить. А если не смущает, тем лучше!

- Программисты, привыкшие к C, могут быть озадачены отсутствием предложений `break` в ветвях `case`. Такое использование `break` в Ruby необязательно (и недопустимо). Связано это с тем, что «проваливание» редко бывает желательно при многопутевом ветвлении. В конце каждой ветви `when` имеется неявный переход на конец предложения `case`. В этом отношении Ruby напоминает Pascal.
- Значения в каждой ветви `case` могут быть произвольными. На типы никаких ограничений не налагается. Они не обязаны быть константами; допускаются и переменные, и сложные выражения. Кроме того, в ветви может проверяться попадание в диапазон.
- В ветвях `case` могут находиться пустые действия (пустые предложения). Значения в разных ветвях не обязательно должны быть уникальными — допускаются перекрытия, например:

```
case x
  when 0
  when 1..5
    puts "Вторая ветвь"
  when 5..10
    puts "Третья ветвь"
else
  puts "Четвертая ветвь"
end
```

Если `x` принимает значение 0, ничего не делается. Для значения 5 печатается строка «Вторая ветвь» — несмотря на то что 5 удовлетворяет и условию в третьей ветви.

- Перекрытие ветвей допускается потому, что они вычисляются в строгом порядке и выполняется закорачивание. Иными словами, если вычисление

выражения в какой-то ветви оказалось успешным, то следующие ветви не вычисляются. Поэтому не стоит помещать в ветви `case` выражения, в которых вызываются методы с побочными эффектами. (Впрочем, такие вызовы в любом случае сомнительны). Имейте также в виду, что такое поведение может замаскировать ошибки, которые произошли бы во время выполнения, если бы выражение вычислялось. Например:

```
case x
  when 1..10
    puts "Первая ветвь"
  when foobar()                # Возможен побочный эффект?
    puts "Вторая ветвь"
  when 5/0                      # Деление на ноль!
    puts "Третья ветвь"
else
  puts "Четвертая ветвь"
end
```

Если `x` находится в диапазоне от 1 до 10, то метод `foobar()` не вызывается, а выражение `5 / 0` (которое, естественно, привело бы к ошибке) не вычисляется.

1.5.4. Рубизмы и идиомы

Материал в этом разделе во многом пересекается с изложенным выше. Но не задумывайтесь особо, почему мы решили разбить его именно таким образом. Просто многие вещи трудно точно классифицировать и организовать единственно правильным образом. Мы ставили себе задачу представить информацию в удобном для усвоения виде.

Ruby проектировался как непротиворечивый и ортогональный язык. Но вместе с тем это сложный язык, в котором есть свои идиомы и странности. Некоторые из них мы обсудим ниже.

- С помощью ключевого слова `alias` можно давать глобальным переменным и методам альтернативные имена (синонимы).
- Пронумерованные глобальные переменные `$1`, `$2`, `$3` и т.д. не могут иметь синонимов.
- Мы не рекомендуем использовать «специальные переменные» `$=`, `$_`, `$/` и им подобные. Иногда они позволяют написать более компактный код, но при этом он не становится более понятным. Поэтому в данной книге мы прибегаем к ним очень редко, что и вам рекомендуем.
- Не путайте операторы диапазона `..` и `...` – первый включает верхнюю границу, второй исключает. Так, диапазон `5..10` включает число 10, а диапазон `5...10` – нет.
- С диапазонами связана одна мелкая деталь, которая может вызвать путаницу. Если дан диапазон `m..n`, то метод `end` вернет конечную его точку `n`, равно как и его синоним `last`. Но те же методы возвращают значение `n` и для диапазона `m...n`, хотя `n` не включается в него. Чтобы различить эти две ситуации, предоставляется метод `end_excluded?`.

- Не путайте диапазоны с массивами. Следующие два присваивания абсолютно различны:

```
x = 1..5
```

```
x = [1, 2, 3, 4, 5]
```

Однако есть удобный метод `to_a` для преобразования диапазона в массив. (Во многих других типах тоже есть такой метод.)

- Часто бывает необходимо присвоить переменной значение лишь в том случае, когда у нее еще нет никакого значения. Поскольку «неприсвоенная» переменная имеет значение `nil`, можно решить эту задачу так: `x = x || 5` или сокращенно `x ||= 5`. Имейте в виду, что значение `false`, а равно и `nil`, будет при этом перезаписано.
- В большинстве языков для обмена значений двух переменных нужна дополнительная временная переменная. В Ruby наличие механизма множественного присваивания делает ее излишней: выражение `x, y = y, x` обменивает значения `x` и `y`.
- Четко отличайте класс от экземпляра. Например, у переменной класса `@foobar` областью видимости является весь класс, а переменная экземпляра `@foobar` заново создается в каждом объекте класса.
- Аналогично метод класса ассоциирован с тем классом, в котором определен; он не принадлежит никакому конкретному объекту и не может вызываться от имени объекта. При вызове метода класса указывается имя класса, а при вызове метода экземпляра – имя объекта.
- В публикациях, посвященных Ruby, часто для обозначения метода экземпляра применяют решеточную нотацию. Например, мы пишем `File.chmod`, чтобы обозначить метод `chmod` класса `File`, и `File#chmod` для обозначения метода экземпляра с таким же именем. Эта нотация не является частью синтаксиса Ruby. Мы старались не пользоваться ей в этой книге.
- В Ruby константы не являются истинно неизменными. Их нельзя изменять в теле методов экземпляра, но из других мест это вполне возможно.
- Ключевое слово `yield` пришло из языка CLU и некоторым программистам может быть непонятно. Оно используется внутри итератора, чтобы передать управление блоку, с которым итератор был вызван. В данном случае `yield` не означает, что нужно получить результат или вернуть значение. Скорее, речь идет о том, чтобы уступить процессор для работы.
- Составные операторы присваивания `+=`, `--` и пр. – это не методы (собственно, это даже не операторы). Это всего лишь «синтаксическая глазурь» или сокращенная форма записи более длинной формы. Поэтому `x += y` значит в точности то же самое, что `x = x + y`. Если оператор `+` перегружен, то оператор `+=` «автоматически» учитывает новую семантику.
- Из-за того, как определены составные операторы присваивания, их нельзя использовать для инициализации переменных. Если первое обращение к переменной `x` выглядит как `x += 1`, возникнет ошибка. Это интуитивно

понятно для программистов, если только они не привыкли к языку, в котором переменные автоматически инициализируются нулем или пустым значением.

- Такое поведение можно в некотором смысле обойти. Можно определить операторы для объекта `nil`, так что в случае, когда начальное значение переменной равно `nil`, мы получим желаемый результат. Так, метод `nil.+`, приведенный ниже, позволит инициализировать объект типа `String` или `Fixnum`, для чего достаточно вернуть аргумент `other`. Таким образом, `nil + other` будет равно `other`.

```
def nil.+(other)
  other
end
```

Мы привели этот код для иллюстрации возможностей Ruby, но стоит ли поступать так на практике, оставляем на усмотрение читателя.

- Уместно будет напомнить, что `Class` – это объект, а `Object` – это класс. Мы попытаемся прояснить этот вопрос в следующей главе, а пока просто повторяйте это как мантру.
- Некоторые операторы нельзя перегружать, потому что они встроены в сам язык, а не реализованы в виде методов. К таковым относятся `=`, `..`, `...`, `and`, `or`, `not`, `&&`, `||`, `!`, `!=` и `!~`. Кроме того, нельзя перегружать составные операторы присваивания (`+=`, `-=` и т.д.). Это не методы и, пожалуй, даже не вполне операторы.
- Имейте в виду, что хотя оператор присваивания перегружать нельзя, тем не менее возможно написать метод экземпляра с именем `foo=` (тогда станет допустимым предложение `x.foo = 5`). Можете рассматривать знак равенства как суффикс.
- Напомним: «голый» оператор разрешения области видимости подразумевает наличие `Object` перед собой, то есть `::Foo` – то же самое, что `Object::Foo`.
- Как уже говорилось, `fail` – синоним `raise`.
- Напомним, что определения в Ruby исполняются. Вследствие динамической природы языка можно, например, определить два метода совершенно по-разному в зависимости от значения признака, проверяемого во время выполнения.
- Напомним, что конструкция `for` (`for x in a`) на самом деле вызывает итератор `each`. Любой класс, в котором такой итератор определен, можно обходить в цикле `for`.
- Не забывайте, что метод, определенный на верхнем уровне, добавляется в модуль `kernel` и, следовательно, становится членом класса `Object`.
- Методы установки (например, `foo=`) должны вызываться от имени объекта, иначе анализатор решит, что речь идет о присваивании переменной с таким именем.

- Напомним, что ключевое слово `retry` можно использовать в итераторах, но не в циклах общего вида. В контексте итератора оно заставляет заново инициализировать все параметры и возобновить текущую итерацию с начала.
- Ключевое слово `retry` применяется также при обработке исключений. Не путайте два этих вида использования.

- Метод объекта `initialize` всегда является закрытым.

- Когда итератор заканчивается левой фигурной скобкой (или словом `end`) и возвращает значение, это значение можно использовать для вызова последующих методов, например:

```
squares = [1,2,3,4,5].collect do |x| x**2 end.reverse
# squares теперь равно [25,16,9,4,1]
```

- В конце программы на Ruby часто можно встретить идиому

```
if $0 == __FILE__
```

Таким образом проверяется, выполняется ли файл как автономный кусок кода (`true`) или как дополнительный, например библиотека (`false`). Типичное применение – поместить некую «главную программу» (обычно с тестовым кодом) в конец библиотеки.

- Обычное наследование (порождение подкласса) обозначается символом `<`:

```
class Dog < Animal
  # ...
end
```

Однако для создания синглетного класса (анонимного класса, который расширяет единственный экземпляр) применяется символ `<<`:

```
class << platypus
  # ...
end
```

- При передаче блока итератору есть тонкое различие между фигурными скобками `{}` и операторными скобками `do-end`. Связано оно с приоритетом:

```
my_method param1, foobar do ... end
# Здесь do-end связано с my_method.
```

```
my_method param1, foobar { ... }
# А здесь {} связано с именем foobar, предполагается, что это метод.
```

- Традиционно в Ruby однострочные блоки заключают в фигурные скобки, а многострочные – в скобки `do-end`, например:

```
my_array.each { |x| puts x }
```

```
my_array.each do |x|
  print x
  if x % 2 == 0
    puts "четно."
  else
    puts "нечетно."
  end
end
```

```
end
end
```

Это необязательно и в некоторых случаях даже нежелательно.

- Помните, что строки (strings) в некотором смысле двулики: их можно рассматривать как последовательность символов или как последовательность строчек (lines). Кому-то покажется удивительным, что итератор `each` оперирует строками (здесь под «строкой» понимается группа символов, завершающаяся разделителем записей, который по умолчанию равен символу новой строки). У `each` есть синоним `each_line`. Если вы хотите перебирать символы, можете воспользоваться итератором `each_byte`. Итератор `sort` также оперирует строками. Для строк (strings) не существует итератора `each_index` из-за возникающей неоднозначности. Действительно, хотим ли мы обрабатывать строку посимвольно или построчно? Все это со временем войдет в привычку.

- Замыкание (closure) запоминает контекст, в котором было создано. Один из способов создать замыкание – использование объекта `Proc`. Например:

```
def power(exponent)
  proc {|base| base**exponent}
end
```

```
square = power(2)
cube = power(3)
```

```
a = square.call(11)    # Результат равен 121.
b = square.call(5)     # Результат равен 25.
c = cube.call(6)       # Результат равен 216.
d = cube.call(8)       # Результат равен 512.
```

Обратите внимание, что замыкание «знает» значение показателя степени, переданное ему в момент создания.

- Однако помните: в замыкании используется переменная, определенная во внешней области видимости (что вполне допустимо). Это свойство может оказаться полезным, но приведем пример неправильного использования:

```
$exponent = 0
```

```
def power
  proc {|base| base**$exponent}
end
```

```
$exponent = 2
square = power
```

```
$exponent = 3
cube = power
```

```
a = square.call(11)    # Неверно! Результат равен 1331.
```

```
b = square.call(5) # Неверно! Результат равен 125.
```

```
# Оба результата неверны, поскольку используется ТЕКУЩЕЕ
# значение $exponent. Так было бы даже в том случае, когда
# используется локальная переменная, покинувшая область
# видимости (например, с помощью define_method).
```

```
c = cube.call(6) # Результат равен 216.
```

```
d = cube.call(8) # Результат равен 512.
```

- Напоследок рассмотрим несколько искусственный пример. Внутри блока итератора `times` создается новый контекст, так что `x` — локальная переменная. Переменная `closure` уже определена на верхнем уровне, поэтому для блока она не будет локальной.

```
closure = nil # Определим замыкание, чтобы его имя было известно.
1.times do # Создаем новый контекст.
  x = 5 # Переменная x локальная в этом блоке.
  closure = Proc.new { puts "В замыкании, x = #{x}" }
end
```

```
x = 1
```

```
# Определяем x на верхнем уровне.
```

```
closure.call # Печатается: В замыкании, x = 5
```

Обратите внимание, что переменная `x`, которой присвоено значение 1, — это новая переменная, определенная на верхнем уровне. Она не совпадает с одноименной переменной, определенной внутри блока. Замыкание печатает 5, так как запоминает контекст своего создания, в котором была определена переменная `x` со значением 5.

- Переменные с именами, начинающимися с одного символа `@`, определенные внутри класса, — это, вообще говоря, переменные экземпляра. Однако если они определены вне любого метода, то становятся переменными экземпляра класса. (Это несколько противоречит общепринятой терминологии ООП, в которой «экземпляр класса» — то же самое, что и «экземпляр» или «объект».) Пример:

```
class MyClass
```

```
  @x = 1 # Переменная экземпляра класса.
```

```
  @y = 2 # Еще одна.
```

```
  def mymethod
```

```
    @x = 3 # Переменная экземпляра.
```

```
    # Заметим, что в этой точке @y недоступна.
```

```
  end
```

```
end
```

Переменная экземпляра класса @y в предыдущем примере – в действительности атрибут объекта класса MyClass, являющегося экземпляром класса Class. (Напомним, что Class – это объект, а Object – это класс.) На переменные экземпляра класса нельзя ссылаться из методов экземпляра и, вообще говоря, они не очень полезны.

- attr, attr_reader, attr_writer и attr_accessor – сокращенная запись для определения методов чтения и установки атрибутов. В качестве аргументов они принимают символы (экземпляры класса Symbol).
- Присваивание переменной, имя которой содержит оператор разрешения области видимости, недопустимо. Например, `math::PI = 3.2` – ошибка.

1.5.5. Ориентация на выражения и прочие вопросы

В Ruby выражения важны почти так же, как предложения. Для программиста на C это звучит знакомо, а для программиста на Pascal – откровенная нелепость. Но Ruby ориентирован на выражения даже в большей степени, чем C.

Заодно в этом разделе мы остановимся на паре мелких вопросов, касающихся регулярных выражений; считайте это небольшим бонусом.

- В Ruby любое присваивание возвращает то же значение, которое стоит в правой части. Поэтому иногда мы можем немного сократить код, как показано ниже, но будьте осторожны, имея дело с объектами! Не забывайте, что это почти всегда ссылки.

```
x = y = z = 0      # Все переменные сейчас равны 0.
```

```
a = b = c = []    # Опасно! a, b и c ссылаются
                  # на ОДИН И ТОТ ЖЕ пустой массив.
```

```
x = 5
y = x += 2        # Сейчас x и y равны 7.
```

Напомним однако, что значения типа Fixnum и им подобные хранятся непосредственно, а не как ссылки на объекты.

- Многие управляющие конструкции возвращают значения, в частности if, unless и case. Следующий код корректен; он показывает, что при принятии решения ветви могут быть выражениями, а не полноценными предложениями.

```
a = 5
x = if a < 8 then 6 else 7 end    # x равно 6.
```

```
y = if a < 8      # y тоже равно 6;
    6            # предложение if может располагаться
else            # на одной строке
    7            # или на нескольких.
end
```

```
# unless тоже работает; z присваивается значение 4.
z = unless x == y then 3 else 4 end
```

```

t = case a           # t получает
  when 0..3          # значение
    "low"            # medium.
  when 4..6
    "medium"
  else
    "high"
  end

```

Здесь мы сделали такие отступы, будто `case` является присваиванием. Мы воспринимаем такую запись спокойно, хотя вам она может не понравиться.

- Отметим, что циклы `while` и `until`, напротив, не возвращают никаких полезных значений; обычно их значением является `nil`:

```

i = 0
x = while (i < 5)    # x равно nil.
  puts i+=1
end

```

- Тернарный оператор можно использовать как в предложениях, так и в выражениях. В силу синтаксических причин (или ограничений анализатора) скобки здесь обязательны:

```

x = 6
y = x == 5 ? 0 : 1          # y равно 1.
x == 5 ? puts("Привет") : puts("Пока") # Печатается: "Пока"

```

- Предложение `return` в конце метода можно опускать. Метод всегда возвращает значение последнего вычисленного выражения, в каком бы месте это вычисление ни происходило.
- Когда итератор вызывается с блоком, последнее выражение, вычисленное в блоке, возвращается в качестве значения блока. Если при этом в теле итератора есть предложение `x = yield`, то `x` будет присвоено это значение.
- *Регулярные выражения.* Напомним, что после регулярного выражения можно написать модификатор многострочности `/m`, и в этом случае точка `(.)` будет сопоставляться с символом новой строки.
- *Регулярные выражения.* Опасайтесь соответствий нулевой длины. Если все элементы регулярного выражения необязательны, то такому образцу будет соответствовать «ничто», причем соответствие всегда будет найдено в начале строки. Это типичная ошибка, особенно часто ее допускают новички.

1.6. Жаргон Ruby

Заново начинать учить английский для освоения Ruby необязательно. Но нужно знать кое-какие жаргонные выражения, обычные в сообществе. Некоторые из них имеют другой смысл, чем принято в компьютерном мире. Им и посвящен настоящий раздел.

В Ruby термин «атрибут» носит неофициальный характер. Можно считать, что атрибут – это переменная экземпляра, которая раскрывается внешнему миру с помощью одного из методов семейства `attr`. Но тут нет полной определенности: могут существовать методы `foo` и `foo=`, не соответствующие переменной `@foo`, как можно было бы ожидать. И, конечно, не все переменные экземпляра считаются атрибутами. Как обычно, нужно придерживаться здравого смысла.

Атрибуты в Ruby можно подразделить на *методы чтения* (*reader*) и *установки* (*writer*). *Метод доступа*, или *акцессор* (*accessor*), является одновременно методом чтения и установки. Это согласуется с названием метода `attr_accessor`, но противоречит принятой в других сообществах семантике, согласно которой акцессор дает доступ только для чтения.

Оператор `===` имеется только в Ruby (насколько мне известно). Обыкновенно он называется оператором ветвящегося равенства (*case equality operator*), поскольку неявно используется в предложениях `case`. Но это название, как я уже говорил, не вполне точно, потому что речь идет не только о «равенстве». В данной книге я часто употребляю термин «*оператор отношения*» (*relationship operator*). Изобрел его не я, но проследить происхождение мне не удалось, к тому же он употребляется нечасто. Жаргонное название – «оператор тройного равенства» (*threeequal operator*) или просто «три равно».

Оператор `<=>`, наверное, лучше всего называть *оператором сравнения*. На жаргоне его называют космическим оператором (*spaceship operator*), поскольку он напоминает летающую тарелку – так ее изображали в старых видеоиграх.

Термин «*поэтический режим*» (*poetry mode*) подчеркивает, что можно опускать ненужные знаки препинания и лексемы (насмешливый намек на отношение поэтов к пунктуации на протяжении последних шестидесяти лет). Поэтический режим также часто означает «опускание скобок при вызове метода».

```
some_method(1, 2, 3)      # Избыточные скобки.  
some_method 1, 2, 3      # "Поэтический режим".
```

Но мне этот принцип представляется более общим. Например, когда хэш передается в качестве последнего или единственного параметра, можно опускать фигурные скобки. В конце строки можно не ставить точку с запятой (а потому никто этого и не делает). В большинстве случаев разрешается опускать ключевое слово `then` в предложениях `if` и `case`.

Некоторые программисты заходят еще дальше, опуская скобки даже в определении методов, но большинство так не поступает:

```
def my_method(a, b, c) # Можно и так: def my_method a, b, c  
  # ...  
end
```

Стоит отметить, что в некоторых случаях сложность грамматики Ruby приводит к сбоям анализатора. Во вложенных вызовах методов скобки для ясности лучше оставлять. Иногда в текущей версии Ruby выводятся предупреждения:

```
def alpha(x)  
  x*2  
end  
def beta(y)
```



```

y*3
end

gamma = 5
delta = alpha beta gamma # 30 -- то же, что alpha(beta(gamma))
# Выдается предупреждение:
# warning: parenthesize argument(s) for future version
# предупреждение: заключайте аргумент(ы) в скобки для совместимости с
# с будущими версиями

```

Термин *duck typing* («утинная типизация» или просто «утипизация»), насколько я знаю, принадлежит Дейву Томасу (Dave Thomas) и восходит к поговорке: «если кто-то выглядит как утка, ковыляет как утка и крякает как утка, то, наверное, это и есть утка». Точный смысл термина «утипизация» — тема для дискуссий, но мне кажется, что это намек на тенденцию Ruby заботиться не столько о точном классе объекта, сколько о том, какие методы для него можно вызывать и какие операции над ним можно выполнять. В Ruby мы редко пользуемся методом `is_a?` или `kind_of`, зато гораздо чаще прибегаем к методу `respond_to?`. Обычное дело — просто передать объект методу, зная, что при неправильном использовании будет возбуждено исключение. Так оно рано или поздно и случается.

Унарную звездочку, которая служит для расширения массива, можно было бы назвать оператором расширения массива, но не думаю, что кто-нибудь слышал такое выражение. В хакерских кругах ходят словечки «звездочка» (*star*) и «расплющивание» (*splat*), а также производные определения — «расплющенный» (*splatted*) и «сплющенный» (*unsplatted*). Дэвид Алан Блэк придумал остроумное название «унарный оператор линеаризации» (*unary unarray operator*)*.

Термин *singleton* (синглет) многие считают перегруженным. Это вполне обычное английское слово, означающее вещь, существующую в единственном экземпляре. Пока мы используем его в качестве модификатора, никакой путаницы не возникает.

Но *Singleton* (Одиночка) — это еще и хорошо известный паттерн проектирования, относящийся к классу, для которого может существовать лишь один объект. В Ruby для реализации этого паттерна имеется библиотека *singleton*.

Синглетный класс (*singleton class*) в Ruby — подобная классу сущность, методы которой хранятся на уровне объекта, а не класса. Пожалуй, это не «настоящий класс», потому что его нельзя инстанцировать. Ниже приведен пример открытия синглетного класса для строкового объекта с последующим добавлением метода:

```

str = "hello"
class << str                                     # Альтернатива:
  def hyphenated                               # def str.hyphenated
    self.split("").join("-")
  end
end

str.hyphenated                                # "h-e-l-l-o"

```

* Прошу читателя извинить меня за это нагромождение выдуманных слов, но жаргон вряд ли следует переводить академически правильным языком. Хочется надеяться, что это все же лучше, чем кальки английских слов. (Прим. перев.)

Кто-то предложил использовать термин `eigenclass` (класс в себе) – производное от немецкого слова `eigen` (свой собственный), коррелирующее с термином «собственное значение» (`eigenvalue`), применяемым в математике и физике. Остроумно, но в сообществе не прижилось и некоторым активно не нравится.

Вернемся к предыдущему примеру. Поскольку метод `hyphenate` не существует ни в каком-либо другом объекте, ни в классе, это *синглетный метод* данного объекта. Это не вызывает неоднозначности. Иногда сам объект называется синглетным, поскольку он единственный в своем роде – больше ни у кого такого метода нет.

Однако вспомним, что в Ruby сам класс является объектом. Поэтому мы можем добавить метод в синглетный класс класса, и этот метод будет уникален для объекта, который – по чистой случайности – оказался классом. Пример:

```
class MyClass
  class << self                # Альтернатива: def self.hello
    def hello                  # или: def MyClass.hello
      puts "Привет от #{self}!"
    end
  end
end
```

Поэтому необязательно создавать объект класса `MyClass` для вызова этого метода.

```
MyClass.hello                # Привет от MyClass!
```

Впрочем, вы, наверное, заметили, что это не что иное, как метод класса в Ruby. Иными словами, метод класса – синглетный метод объекта-класса. Можно также сказать, что это синглетный метод, определенный для объекта, который случайно оказался классом.

Осталась еще парочка терминов. Переменная класса – это, разумеется, то, имя чего начинается с двух символов `@`. Возможно, название неудачно из-за нетривиального поведения относительно наследования. Переменная экземпляра класса – нечто совсем иное. Это обычная переменная экземпляра; только объект, которому она принадлежит, является классом. Дополнительную информацию по этой теме вы найдете в главе 11.

1.7. Заключение

На этом завершается наш обзор объектно-ориентированного программирования и краткая экскурсия по языку Ruby. В последующих главах изложенный материал будет раскрыт более полно.

Хотя в мои намерения не входило «учить Ruby» новичков, не исключено, что даже начинающие программисты на Ruby почерпнут что-то полезное из этой главы. Как бы то ни было, последующие главы будут полезны «рубистам» начального и среднего уровня. Надеюсь, что даже опытные программисты на Ruby найдут для себя что-то новенькое.

Глава 2. Строки

.....
*Когда-то элементарными кирпичиками мироздания считались атомы,
потом протоны, потом кварки. Теперь таковыми считаются струны*.*

*Дэвид Гросс, профессор теоретической физики,
Принстонский университет*

В начале 1980-х годов один профессор информатики, начиная первую лекцию по структурам данных, не представился студентам, не сказал, как называется курс, не рассказал о его программе и не порекомендовал никаких учебников – а вместо этого спросил: «Какой тип данных самый важный?»

Было высказано несколько предположений. Когда профессор услышал слово «указатели», он выразил удовлетворение, но все-таки не согласился со студентом, а высказал свое мнение: «Самым важным является тип символ».

У него были на то основания. Компьютерам предназначено быть нашими слугами, а не хозяевами, а человеку понятны только символьные данные. (Есть, конечно, люди, которые без труда читают и двоичные данные, но о них мы сейчас говорить не будем.) Символы, а стало быть, и строки, позволяют человеку общаться с компьютером. Любую информацию, в том числе и текст на естественном языке, можно закодировать в виде строк.

Как и в других языках, строка в Ruby – просто последовательность символов. Подобно другим сущностям, строки являются полноценными объектами. В программах приходится выполнять разнообразные операции над строками: конкатенировать, выделять лексемы, анализировать, производить поиск и замену и т.д. Язык Ruby позволяет все это делать без труда.

Почти всюду в этой главе предполагается, что байт – это символ. Но при работе в многоязычной среде это не совсем так. Вопросы интернационализации обсуждаются в главе 4.

2.1. Представление обычных строк

Строка в Ruby – это последовательность 8-битовых байтов. Она не завершается нулевым символом, как в C, и, следовательно, может содержать такие символы. В строке могут быть символы с кодами больше 0xFF, но такие строки имеют смысл лишь если выбран некоторый набор символов (кодировка). Дополнительную информацию о кодировках вы найдете в главе 4.

* В английском языке словом «string» обозначается как «строка», так и «струна» (Прим. перев.)

Простейшая строка в Ruby заключается в одиночные кавычки. Такие строки воспринимаются буквально; в качестве управляющих символов в них распознаются только экранированная одиночная кавычка (\') и экранированный символ обратной косой черты (\\):

```
s1 = 'Это строка.'          # Это строка.
s2 = 'Г-жа О'Лири'         # Г-жа О'Лири.
s3 = 'Смотри в C:\\ТЕМР'    # Смотри в C:\ТЕМР.
```

Строки, заключенные в двойные кавычки, обладают большей гибкостью. В них допустимо много других управляющих последовательностей, в частности для представления символов забоя, табуляции, возврата каретки и перевода строки. Можно также включать произвольные символы, представленные восьмеричными цифрами:

```
s1 = "Это знак табуляции: (\t)"
s2 = "Несколько символов забоя: xyz\b\b\b"
s3 = "Это тоже знак табуляции: \011"
s4 = "А это символы подачи звукового сигнала: \a \007"
```

Внутри строки, заключенной в двойные кавычки, могут встречаться даже выражения (см. раздел 2.21).

2.2. Альтернативная нотация для представления строк

Иногда встречаются строки, в которых много метасимволов, например одиночных и двойных кавычек и т.д. В этом случае можно воспользоваться конструкциями %q и %Q. Вслед за ними должна идти строка, обрамленная с обеих сторон символами-ограничителями; лично я предпочитаю квадратные скобки ([]).

При этом %q ведет себя как одиночные кавычки, а %Q — как двойные.

```
s1 = %q[Как сказал Маргритт, "Ceci n'est pas une pipe."]
s2 = %q[Это не табуляция: (\t)] # равнозначно 'Это не табуляция: \t'
s3 = %Q[А это табуляция: (\t)] # равнозначно "А это табуляция: \t"
```

В обоих вариантах можно применять и другие ограничители, помимо квадратных скобок: круглые, фигурные, угловые скобки.

```
s1 = %q(Билл сказал: "Боб сказал: 'This is a string.'")
s2 = %q{Другая строка.}
s3 = %q<В этой строке есть специальные символы '"[](){}>."
```

Допустимы также непарные ограничители. В этом качестве может выступать любой символ, кроме букв, цифр и пропусков (пробелов и им подобных), который имеет визуальное представление и не относится к числу перечисленных выше парных ограничителей.

```
s1 = %q: "Я думаю, что это сделала корова г-жи О'Лири," сказал он.:
s2 = %q*\r - это control-M, а \n - это control-J.*
```

2.3. Встроенные документы

Для представления длинной строки, занимающей несколько строк в тексте, можно, конечно, воспользоваться обычными строками в кавычках:

```
str = "Три девицы под окном  
Пряли поздно вечером..."
```

Но тогда отступ окажется частью строки.

Можно вместо этого воспользоваться встроенным документом, изначально предназначенным для многострочных фрагментов. (Идея и сам термин заимствованы из более старых языков.) Синтаксически он начинается с двух знаков <<, за которыми следует концевой маркер, нуль или более строк текста и в завершение тот же самый концевой маркер в отдельной строке:

```
str = <<EOF  
Три девицы под окном  
Пряли поздно вечером...  
EOF
```

Но следите внимательно, чтобы после завершающего концевого маркера не было пробелов. В текущей версии Ruby маркер в такой ситуации не распознается.

Встроенные документы могут быть вложенными. В примере ниже показано, как передать методу три представленных таким образом строки:

```
some_method(<<str1, <<str2, <<str3)  
первый кусок  
текста...  
str1  
второй кусок...  
str2  
третий кусок  
текста.  
str3
```

По умолчанию встроенный документ ведет себя как строка в двойных кавычках, то есть внутри него интерпретируются управляющие последовательности и интерполируются выражения. Но если концевой маркер заключен в одиночные кавычки, то и весь документ ведет себя как строка в одиночных кавычках:

```
str = <<'EOF'  
Это не знак табуляции: \t  
а это не символ новой строки: \n  
EOF
```

Если концевому маркеру встроенного документа предшествует дефис, то маркер может начинаться с красной строки. При этом удаляются только пробелы из той строки, на которой расположен сам маркер, но не из предшествующих ей строк документа.

```
str = <<-EOF  
Каждая из этих строк  
начинается с пары  
пробелов.  
EOF
```

Опишу стиль, который нравится лично мне. Предположим, что определен такой метод margin:

```
class String  
  def margin
```

```

arr = self.split("\n")           # Разбить на строки.
arr.map! {|x| x.sub!(/\s*\/, "")} # Удалить начальные символы.
str = arr.join("\n")             # Объединить в одну строку.
self.replace(str)                 # Подменить исходную строку.
end
end

```

Для ясности я включил подробные комментарии. В этом коде применяются конструкции, которые будут рассмотрены ниже – как в этой, так и в последующих главах.

Используется этот метод так:

```

str = <<end.margin
|Этот встроенный документ имеет "левое поле"
|на уровне вертикальной черты в каждой строке.
|
| Можно включать цитаты,
| делать выступления и т. д.
end

```

В качестве концевого маркера естественно употребить слово `end`. (Впрочем, это дело вкуса. Выглядит такой маркер как зарезервированное слово `end`, но на самом деле этот выбор ничуть не хуже любого другого.) Каждая строка начинается с символа вертикальной черты, но эти символы потом отбрасываются вместе с начальными пробелами.

2.4. Получение длины строки

Для получения длины строки служит метод `length`. У него есть синоним `size`.

```

str1 = "Капл"
x = str1.length # 4
str2 = "Дойль"
x = str2.size # 5

```

2.5. Построчная обработка

Строка в Ruby может содержать символы новой строки. Например, можно прочитать в память файл и сохранить его в виде одной строки. Применяемый по умолчанию итератор `each` в этом случае перебирает отдельные строки:

```

str = "Когда-то\пдавным-давно... \пКонец\n"
num = 0
str.each do |line|
  num += 1
  print "Строка #{num}: #{line}"
end

```

Выполнение этого кода дает следующий результат:

```

Строка 1: Когда-то
Строка 2: `давным-давно...
Строка 3: Конец

```

Альтернативно можно было бы воспользоваться методом `each_with_index`.

2.6. Побайтовая обработка

Поскольку на момент написания этой книги язык Ruby еще не поддерживал интернационализацию в полной мере, то символ и байт – по существу одно и то же. Для последовательной обработки символов пользуйтесь итератором `each_byte`:

```
str = "ABC"
str.each_byte {|char| print char, " " }
# Результат: 65 66 67.
```

В текущей версии Ruby строку можно преобразовать в массив односимвольных строк с помощью метода `scan`, которому передается простое регулярное выражение, соответствующее одному символу:

```
str = "ABC"
chars = str.scan(/./)
chars.each {|char| print char, " " }
# Результат: A B C.
```

2.7. Специализированное сравнение строк

В язык Ruby уже встроен механизм сравнения строк: строки сравниваются в привычном лексикографическом порядке (то есть на основе упорядочения, присущего данному набору символов). Но при желании можно задать собственные правила сравнения любой сложности.

Предположим, например, что мы хотим игнорировать английские артикли *a*, *an* и *the*, если они встречаются в начале строки, а также не обращать внимания на большинство знаков препинания. Для этого следует переопределить встроенный метод `<=>` (он вызывается из методов `<`, `<=`, `>` и `>=`). В листинге 2.1 показано, как это сделать.

Листинг 2.1. Специализированное сравнение строк

```
class String
  alias old_compare <=>
  def <=>(other)
    a = self.dup
    b = other.dup
    # Удалить знаки препинания.
    a.gsub!(/[\\.\.\?\\!\:\;\]/, "")
    b.gsub!(/[\\.\.\?\\!\:\;\]/, "")
    # Удалить артикли из начала строки.
    a.gsub!(/^ (a |an |the )/i, "")
    b.gsub!(/^ (a |an |the )/i, "")
    # Удалить начальные и хвостовые пробелы.
    a.strip!
    b.strip!
    # Вызвать старый метод <=>.
    a.old_compare(b)
  end
end
```

```
end
title1 = "Calling All Cars"
title2 = "The Call of the Wild"
# При стандартном сравнении было бы напечатано "yes".
if title1 < title2
  puts "yes"
else
  puts "no"                # А теперь печатается "no".
end
```

Обратите внимание, что мы «сохранили» старый метод `<=>` с помощью ключевого слова `alias` и в конце вызвали его. Если бы мы вместо этого воспользовались методом `<`, то был бы вызван новый метод `<=>`, что привело бы к бесконечной рекурсии и в конечном счете к аварийному завершению программы.

Отметим также, что оператор `==` не вызывает метод `<=>` (принадлежащий классу-примеси `Comparable`). Это означает, что для специализированной проверки строк на равенство пришлось бы отдельно переопределить метод `==`. Но в рассмотренном случае в этом нет необходимости.

Допустим, что мы хотим сравнивать строки без учета регистра. Для этого есть встроенный метод `casecmp`; надо лишь добиться, чтобы он вызывался при сравнении. Вот как это можно сделать:

```
class String
  def <=>(other)
    casecmp(other)
  end
end
```

Есть и более простой способ:

```
class String
  alias <=> casecmp(other)
end
```

Но это не все. Надо еще переопределить оператор `==`, чтобы он вел себя точно так же:

```
class String
  def ==(other)
    casecmp(other) == 0
  end
end
```

Теперь все строки будут сравниваться без учета регистра. И при всех операциях сортировки, которые определены в терминах метода `<=>`, регистр тоже не будет учитываться.

2.8. Разбиение строки на лексемы

Метод `split` разбивает строку на части и возвращает массив лексем. Ему передаются два параметра: разделитель и максимальное число полей (целое).

По умолчанию разделителем является пробел, а точнее, значение специальной переменной `$;` или ее англоязычного эквивалента `$FIELD_SEPARATOR`. Если же первым параметром задана некоторая строка, то она и будет использоваться в качестве разделителя лексем.

```
s1 = "Была темная грозовая ночь."
words = s1.split          # ["Была", "темная", "грозовая", "ночь"]
s2 = "яблоки, груши, персики"
list = s2.split(", ")     # ["яблоки", "груши", "персики"]
s3 = "львы и тигры и медведи"
zoo = s3.split(/ и /)     # ["львы", "тигры", "медведи"]
```

Второй параметр ограничивает число возвращаемых полей, при этом действуют следующие правила:

1. Если параметр опущен, то пустые поля в конце отбрасываются.
2. Если параметр – положительное число, то будет возвращено не более указанного числа полей (если необходимо, весь «хвост» строки помещается в последнее поле). Пустые поля в конце сохраняются.
3. Если параметр – отрицательное число, то количество возвращаемых полей не ограничено, а пустые поля в конце сохраняются.

Ниже приведены примеры:

```
str = "alpha,beta,gamma,,"
list1 = str.split(",") # ["alpha","beta","gamma"]
list2 = str.split(", ", 2) # ["alpha", "beta,gamma,,"]
list3 = str.split(", ", 4) # ["alpha", "beta", "gamma", "", ""]
list4 = str.split(", ", 8) # ["alpha", "beta", "gamma", "", "", "", "", ""]
list5 = str.split(", ", -1) # ["alpha", "beta", "gamma", "", "", "", "", ""]
```

Для сопоставления строки с регулярным выражением или с другой строкой служит метод `scan`:

```
str = "I am a leaf on the wind..."
# Строка интерпретируется буквально, а не как регулярное выражение.
arr = str.scan("a") # ["a","a","a"]
# При сопоставлении с регулярным выражением возвращаются все соответствия.
arr = str.scan(/w+/) # ["I", "am", "a", "leaf", "on", "the", "wind"]
# Можно задать блок.
str.scan(/w+/) {|x| puts x }
```

Класс `StringScanner` из стандартной библиотеки отличается тем, что сохраняет состояние сканирования, а не выполняет все за один раз:

```
require 'strscan'
str = "Смотри, как я парю!"
ss = StringScanner.new(str)
loop do
  word = ss.scan(/w+/) # Получать по одному слову.
  break if word.nil?
  puts word
end
```

```
sep = ss.scan(/\W+/)      # Получить следующий фрагмент,
                           # не являющийся словом.

break if sep.nil?
end
```

2.9. Форматирование строк

В Ruby, как и в языке C, для этой цели предназначен метод `sprintf`. Он принимает строку и список выражений, а возвращает строку. Набор спецификаторов в форматной строке мало чем отличается от принятого в функции `sprintf` (или `printf`) из библиотеки C.

```
name = "Боб"
age = 28
str = sprintf("Привет, %s... Похоже, тебе %d лет.", name, age)
```

Спрашивается, зачем нужен этот метод, если можно просто интерполировать значения в строку с помощью конструкции `# {expr}`? А затем, что `sprintf` позволяет выполнить дополнительное форматирование – например, задать максимальную ширину поля или максимальное число цифр после запятой, добавить или подавить начальные нули, выравнивать строки текста по левой или правой границе и т.д.

```
str = sprintf("%-20s %3d", name, age)
```

В классе `String` есть еще метод `%`, который делает почти то же самое. Он принимает одно значение или массив значений любых типов:

```
str = "%-20s %3d" % [name, age] # То же, что и выше
```

Имеются также методы `ljust`, `rjust` и `center`; они принимают длину результирующей строки и дополняют ее до указанной длины пробелами, если это необходимо.

```
str = "Моби Дик"
s1 = str.ljust(12)      # "Моби Дик"
s2 = str.center(12)     # "  Моби Дик  "
s3 = str.rjust(12)      # "      Моби Дик"
```

Можно задать и второй параметр, который интерпретируется как строка заполнения (при необходимости она будет урезана):

```
str = "Капитан Ахав"
s1 = str.ljust(20, "+")  # "Капитан Ахав+++++++"
s2 = str.center(20, "-") # "—Капитан Ахав—"
s3 = str.rjust(20, "123") # "12312312Капитан Ахав"
```

2.10. Строки в качестве объектов ввода/вывода

Помимо методов `sprintf` и `scanf`, есть еще один способ имитировать ввод/вывод в строку: класс `StringIO`.

Из-за сходства с объектом `IO` мы рассмотрим его в главе, посвященной вводу/выводу (см. раздел 10.1.24).

2.11. Управление регистром

В классе `String` есть множество методов управления регистром. В этом разделе мы приведем их краткий обзор.

Метод `downcase` переводит символы всей строки в нижний регистр, а метод `upcase` – в верхний:

```
s1 = "Бостонское чаепитие"
s2 = s1.downcase           # "бостонское чаепитие"
s3 = s2.upcase             # "БОСТОНСКОЕ ЧАЕПИТИЕ"
```

Метод `capitalize` представляет первый символ строки в верхнем регистре, а все остальные – в нижнем:

```
s4 = s1.capitalize         # "Бостонское чаепитие"
s5 = s2.capitalize         # "Бостонское чаепитие"
s6 = s3.capitalize         # "Бостонское чаепитие"
```

Метод `swapcase` изменяет регистр каждой буквы на противоположный:

```
s7 = "ЭТО БЫВШИЙ ПОПУГАЙ."
s8 = s7.swapcase           # "это бывший ПОПУГАЙ."
```

Начиная с версии 1.8, в язык Ruby включен метод `casecmp`, который работает аналогично стандартному методу `<=>`, но игнорирует регистр:

```
n1 = "abc".casecmp("xyz") # -1
n2 = "abc".casecmp("XYZ") # -1
n3 = "ABC".casecmp("xyz") # -1
n4 = "ABC".casecmp("abc") # 0
n5 = "xyz".casecmp("abc") # 1
```

У каждого из перечисленных методов имеется аналог, осуществляющий модификацию «на месте» (`upcase!`, `downcase!`, `capitalize!`, `swapcase!`).

Не существует встроенных методов, позволяющих узнать регистр буквы, но это легко сделать с помощью регулярных выражений:

```
if string =~ /[a-z]/
  puts "строка содержит символы в нижнем регистре"
end
if string =~ /[A-Z]/
  puts "строка содержит символы в верхнем регистре"
end
if string =~ /[A-Z]/ and string =~ /a-z/
  puts "строка содержит символы в разных регистрах"
end
if string[0..0] =~ /[A-Z]/
  puts "строка начинается с прописной буквы"
end
```

Отметим, что все эти методы не учитывают местные особенности (`locale`).

2.12. Вычленение и замена подстрок

В Ruby к подстрокам можно обращаться разными способами. Обычно применяются квадратные скобки, как для массивов, но внутри скобок может находиться

пара объектов класса `Fixnum`, диапазон, регулярное выражение или строка. Ниже мы рассмотрим все варианты.

Если задана пара объектов класса `Fixnum`, то они трактуются как смещение от начала строки и длина, а возвращается соответствующая подстрока.

```
str = "Шалтай-Болтай"
sub1 = str[7,4]           # "Болт"
sub2 = str[7,99]          # "Болтай" (выход за границу строки допускается)
sub3 = str[10,-4]         # nil (отрицательная длина)
```

Важно помнить, что это именно смещение и длина (число символов), а не начальное и конечное смещение.

Если индекс отрицателен, то отсчет ведется от конца строки. В этом случае индекс начинается с единицы, а не с нуля. Но при нахождении подстроки указанной длины все равно берутся символы правее, а не левее начального:

```
str1 = "Алиса"
sub1 = str1[-3,3]         # "иса"
str2 = "В Зазеркалье"
sub3 = str2[-8,6]         # "зеркал"
```

Можно задавать диапазон. Он интерпретируется как диапазон позиций внутри строки. Диапазон может включать отрицательные числа, но в любом случае нижняя граница не должна быть больше верхней. Если диапазон «инвертированный» или нижняя граница оказывается вне строки, возвращается `nil`:

```
str = "Уинстон Черчилль"
sub1 = str[8..13]         # "Черчил"
sub2 = str[-4..-1]        # "илль"
sub3 = str[-1..-4]        # nil
sub4 = str[25..30]        # nil
```

Если задано регулярное выражение, то возвращается строка, соответствующая образцу. Если соответствия нет, возвращается `nil`:

```
str = "Alistair Cooke"
sub1 = str[/l..t/]        # "list"
sub2 = str[/s.*r/]        # "stair"
sub3 = str[/foo/]         # nil
```

Если задана строка, то она и возвращается, если встречается в качестве подстроки в исходной строке; в противном случае возвращается `nil`:

```
str = "theater"
sub1 = str["heat"]        # "heat"
sub2 = str["eat"]         # "eat"
sub3 = str["ate"]         # "ate"
sub4 = str["beat"]        # nil
sub5 = str["cheat"]       # nil
```

Наконец, в тривиальном случае, когда в качестве индекса задано одно число `Fixnum`, возвращается ASCII-код символа в соответствующей позиции (или `nil`, если индекс выходит за границы строки):

```
str = "Aaron Burr"
ch1 = str[0] # 65
```

```
ch1 = str[1]           # 97
ch3 = str[99]          # nil
```

Важно понимать, что все описанные выше способы могут использоваться не только для доступа к подстроке, но и для ее замены:

```
str1 = "Шалтай-Болтай"
str1[7,3] = "Хва"      # "Шалтай-Хватай"
str2 = "Алиса"
str2[-3,3] = "ександра" # "Александра"
str3 = "В Зазеркалье"
str3[-9,9] = "стеколье" # "В Застеколье"
str4 = "Уинстон Черчилль"
str4[8..11] = "Х"      # "Уинстон Хилль"
str5 = "Alistair Cooke"
str5[/e$/] = "ie Monster" # "Alistair Cookie Monster"
str6 = "theater"
str6["er"] = "re"      # "theatre"
str7 = "Aaron Burr"
str7[0] = 66 # "Baron Burr"
```

Присваивание выражения, равного nil, не оказывает никакого действия.

2.13. Подстановка в строках

Мы уже видели, как выполняются простые подстановки. Методы `sub` и `gsub` предоставляют более развитые средства, основанные на сопоставлении с образцом. Имеются также варианты `sub!` и `gsub!`, позволяющие выполнить подстановку «на месте».

Метод `sub` заменяет первое вхождение строки, соответствующей образцу, другой строкой или результатом вычисления блока:

```
s1 = "spam, spam, and eggs"
s2 = s1.sub(/spam/, "bacon") # "bacon, spam, and eggs"
s3 = s2.sub(/(\w+), (\w+)/, '\2, \1,') # "spam, bacon, and eggs"
s4 = "Don't forget the spam."
s5 = s4.sub(/spam/) { |m| m.reverse } # "Don't forget the maps."
s4.sub!(/spam/) { |m| m.reverse }
# s4 теперь равно "Don't forget the maps."
```

Как видите, в подставляемой строке могут встречаться специальные символы `\1`, `\2` и т.д. Но такие специальные переменные, как `$&` (или ее англоязычная версия `$MATCH`), не допускаются.

Если употребляется форма с блоком, то допустимы и специальные переменные. Если вам нужно лишь получить сопоставленную с образцом строку, то она будет передана в блок как параметр. Если эта строка вообще не нужна, то параметр, конечно, можно опустить.

Метод `gsub` (глобальная подстановка) отличается от `sub` лишь тем, что заменяются все вхождения, а не только первое:

```
s5 = "alfalfa abracadabra"
s6 = s5.gsub(/a[bl]/, "xx") # "xxfxxxfa xxracadxrra"
s5.gsub!(/[\lfdblr]/) { |m| m.upcase + "-" }
# s5 теперь равно "aL-F-aL-F-a aB-R-acad-aB-R-a"
```

Метод `Regexp.last_match` эквивалентен действию специальной переменной `$&` (она же `$MATCH`).

2.14. Поиск в строке

Помимо различных способов доступа к подстрокам, есть и другие методы поиска в строке. Метод `index` возвращает начальную позицию заданной подстроки, символа или регулярного выражения. Если подстрока не найдена, возвращается `nil`:

```
str = "Albert Einstein"
pos1 = str.index(?E)      # 7
pos2 = str.index("bert")  # 2
pos3 = str.index(/in/)    # 8
pos4 = str.index(?W)      # nil
pos5 = str.index("bart")  # nil
pos6 = str.index(/wein/)  # nil
```

Метод `rindex` начинает поиск с конца строки. Но номера позиций отсчитываются тем не менее от начала:

```
str = "Albert Einstein"
pos1 = str.rindex(?E)     # 7
pos2 = str.rindex("bert") # 2
pos3 = str.rindex(/in/)   # 13 (найдено самое правое соответствие)
pos4 = str.rindex(?W)     # nil
pos5 = str.rindex("bart") # nil
pos6 = str.rindex(/wein/) # nil
```

Метод `include?` сообщает, встречается ли в данной строке указанная подстрока или один символ:

```
str1 = "mathematics"
flag1 = str1.include? ?e      # true
flag2 = str1.include? "math"  # true
str2 = "Daylight Saving Time"
flag3 = str2.include? ?s      # false
flag4 = str2.include? "Savings" # false
```

Метод `scan` многократно просматривает строку в поисках указанного образца. Будучи вызван внутри блока, он возвращает массив. Если образец содержит несколько (заклученных в скобки) групп, то массив окажется вложенным:

```
str1 = "abracadabra"
sub1 = str1.scan(/a./)
# sub1 теперь равно ["ab", "ac", "ad", "ab"]
str2 = "Acapulco, Mexico"
sub2 = str2.scan(/(.) (c.)/)
# sub2 теперь равно [ ["A", "ca"], ["l", "co"], ["i", "co"] ]
```

Если при вызове задан блок, то метод поочередно передает этому блоку найденные значения:

```
str3 = "Kobayashi"
str3.scan(/[^aeiou]+[aeiou]/) do |x|
  print "Слог: #{x}\n"
end
```

Этот код выводит такой результат:

```
Слог: Ko
Слог: ba
Слог: ya
Слог: shi
```

2.15. Преобразование символов в коды ASCII и обратно

В Ruby символ представляется целым числом. Это поведение изменится в версии 2.0, а возможно и раньше. В будущем предполагается хранить символы в виде односимвольных строк.

```
str = "Martin"
print str[0] # 77
```

Если в конец строки дописывается объект типа `Fixnum`, то он предварительно преобразуется в символ:

```
str2 = str << 111 # "Martino"
```

2.16. Явные и неявные преобразования

На первый взгляд, методы `to_s` и `to_str` могут вызвать недоумение. Ведь оба преобразуют объект в строковое представление, так?

Но есть и различия. Во-первых, любой объект в принципе можно как-то преобразовать в строку, поэтому почти все системные классы обладают методом `to_s`. Однако метод `to_str` в системных классах не реализуется никогда.

Как правило, метод `to_str` применяется для объектов, очень похожих на строки, способных «замаскироваться» под строку. В общем, можете считать, что метод `to_s` — это явное преобразование, а метод `to_str` — неявное.

Я уже сказал, что ни в одном системном классе не определен метод `to_str` (по крайней мере, мне о таких классах неизвестно). Но иногда они вызывают `to_str` (если такой метод существует в соответствующем классе).

Первое, что приходит на ум, — подкласс класса `String`; но на самом деле объект любого класса, производного от `String`, уже является строкой, так что определять метод `to_str` излишне.

А вот пример из реальной жизни. Класс `Pathname` определен для удобства работы с путями в файловой системе (например, конкатенации). Но путь естественно отображается на строку (хотя и не наследует классу `String`).

```
require 'pathname'
path = Pathname.new("/tmp/myfile")
name = path.to_s # "/tmp/myfile"
name = path.to_str # "/tmp/myfile" (Ну и что?)
# Вот где это оказывается полезно...
heading = "Имя файла равно " + path
puts heading # "Имя файла равно /tmp/myfile"
```

В этом фрагменте мы просто дописали путь в конец обычной строки "Имя файла равно". Обычно такая операция приводит к ошибке во время выполнения,

поскольку оператор `+` ожидает, что второй операнд – тоже строка. Но так как в классе `Pathname` есть метод `to_str`, то он вызывается. Класс `Pathname` «маскируется» под строку, то есть может быть неявно преобразован в `String`.

На практике методы `to_s` и `to_str` обычно возвращают одно и то же значение, но это необязательно. Неявное преобразование должно давать «истинное строковое значение» объекта, а явное можно расценивать как «принудительное» преобразование.

Метод `puts` обращается к методу `to_s` объекта, чтобы получить его строковое представление. Можно считать, что это неявный вызов явного преобразования. То же самое справедливо в отношении интерполяции строк. Вот пример:

```
class Helium
  def to_s
    "He"
  end
  def to_str
    "гелий"
  end
end

e = Helium.new
print "Элемент "
puts e                # Элемент He.
puts "Элемент " + e   # Элемент гелий.
puts "Элемент #{e}"   # Элемент He.
```

Как видите, разумное определение этих методов в собственном классе может несколько повысить гибкость применения. Но что сказать об идентификации объектов, переданных методам вашего класса?

Предположим, например, что вы написали метод, который ожидает в качестве параметра объект `String`. Вопреки философии «утипизации», так делают часто, и это вполне оправдано. Например, предполагается, что первый параметр метода `File.new` – строка.

Решить эту проблему просто. Если вы ожидаете на входе строку, проверьте, имеет ли объект метод `to_str`, и при необходимости вызывайте его.

```
def set_title(title)
  if title.respond_to? :to_str
    title = title.to_str
  end
  # ...
end
```

Ну а если объект не отвечает на вызов метода `to_str`? Есть несколько вариантов действий. Можно принудительно вызвать метод `to_s`; можно проверить, принадлежит ли объект классу `String` или его подклассу; можно, наконец, продолжать работать, понимая, что при попытке выполнить операцию, которую объект не поддерживает, мы получим исключение `ArgumentError`.

Короткий путь к цели выглядит так:


```
title = title.to_str rescue title
```

Он опирается на тот факт, что при отсутствии реализации метода `to_str` возникнет исключение. Разумеется, модификаторы `rescue` могут быть вложенными:

```
title = title.to_str rescue title.to_s rescue title
```

Обрабатывается маловероятный случай, когда отсутствует даже метод `to_s`.

С помощью неявного преобразования можно было бы сделать строки и числа практически эквивалентными:

```
class Fixnum
  def to_str
    self.to_s
  end
end
```

```
str = "Число равно " + 345 # Число равно 345.
```

Но я не рекомендую так поступать: «много хорошо тоже нехорошо». В Ruby, как и в большинстве языков, строки и числа – разные сущности. Мне кажется, что ясности ради преобразования, как правило, должны быть явными.

И еще: в методе `to_str` нет ничего волшебного. Предполагается, что он возвращает строку, но если вы пишете такой метод сами, ответственность за то, что он действительно так и поступает, ложится на вас.

2.17. Дописывание в конец строки

Для конкатенации строк применяется оператор `<<`. Он «каскадный», то есть позволяет выполнять подряд несколько операций над одним и тем же операндом-приемником.

```
str = "A"
str << [1,2,3].to_s << " " << (3.14).to_s
# str теперь равно "A123 3.14".
```

Если число типа `Fixnum` принадлежит диапазону 0..255, то оно будет преобразовано в символ:

```
str = "Marlow"
str << 101 << ", Christopher"
# str теперь равно "Marlowe, Christopher".
```

2.18. Удаление хвостовых символов новой строки и прочих

Часто бывает необходимо удалить лишние символы в конце строки. Типичный пример – удаление символа новой строки после чтения строки из внешнего источника.

Метод `chop` удаляет последний символ строки (обычно это символ новой строки). Если перед символом новой строки находится символ перевода каретки (`\r`), он тоже удаляется. Причина такого поведения заключается в том, что

разные операционные системы неодинаково трактуют понятие «новой строки». В UNIX-подобных системах новая строка представляется символом `\n`. А в DOS и Windows для этой цели используется пара символов `\r\n`.

```
str = gets.chomp      # Прочитать, удалить символ новой строки.
s2 = "Some string\n"  # "Some string" (нет символа новой строки).
s3 = s2.chomp!        # s2 теперь тоже равно "Some string".
s4 = "Other string\r\n"
s4.chomp!             # "Other string" (нет символа новой строки).
```

Обратите внимание, что при вызове варианта `chomp!` операнд-источник модифицируется.

Важно еще отметить, что последний символ удаляется, даже если это не символ новой строки:

```
str = "abcxyz"
s1 = str.chomp # "abcxy"
```

Поскольку символ новой строки присутствует не всегда, иногда удобнее применять метод `chomp`:

```
str = "abcxyz"
str2 = "123\n"
str3 = "123\r"
str4 = "123\r\n"
s1 = str.chomp      # "abcxyz"
s2 = str2.chomp     # "123"
# Если установлен стандартный разделитель записей, то удаляется не только
# \n, но также \r и \r\n.
s3 = str3.chomp     # "123"
s4 = str4.chomp     # "123"
```

Как и следовало ожидать, имеется также метод `chomp!` для замены «на месте».

Если методу `chomp` передана строка-параметр, то удаляются перечисленные в ней символы, а не подразумеваемый по умолчанию разделитель записей. Кстати, если разделитель записей встречается в середине строки, то он не удаляется:

```
str1 = "abcxyz"
str2 = "abcxyz"
s1 = str1.chomp("yz") # "abcx"
s2 = str2.chomp("x")  # "abcxyz"
```

2.19. Удаление лишних пропусков

Метод `strip` удаляет пропуски в начале и в конце строки, а вариант `strip!` делает то же самое «на месте».

```
str1 = "\t \nabc \t\n"
str2 = str1.strip      # "abc"
str3 = str1.strip!     # "abc"
# str1 теперь тоже равно "abc".
```

Под пропусками, как обычно, понимаются пробелы, символы табуляции и перехода на новую строку.

Чтобы удалить пропуски только в начале или только в конце строки, примените методы `lstrip` и `rstrip`:

```
str = " abc "
s2 = str.lstrip # "abc "
s3 = str.rstrip # " abc"
```

Имеются также варианты `lstrip!` и `rstrip!` для удаления «на месте».

2.20. Повтор строк

В Ruby оператор (или метод) умножения перегружен так, что в применении к строкам выполняет операцию повторения. Если строку умножить на n , то получится строка, состоящая из n конкатенированных копий исходной:

```
etc = "Etc. " * 3 # "Etc. Etc. Etc. "
ruler = "+" + ("." * 4 + "5" * "+")." * 4 + "+" * 3
# "+....5....+....5....+....5....+"
```

2.21. Включение выражений в строку

Это легко позволяет сделать синтаксическая конструкция `#{}.` Нет нужды думать о преобразовании, добавлении и конкатенации; нужно лишь интерполировать переменную или другое выражение в любое место строки:

```
puts "#{temp_f} по Фаренгейту равно #{temp_c} по Цельсию"
puts "Значение определителя равно #{b*b - 4*a*c}."
puts "#{word} это #{word.reverse} наоборот."
```

Внутри фигурных скобок могут находиться даже полные предложения. При этом возвращается результат вычисления последнего выражения.

```
str = "Ответ равен #{ def factorial(n)
                        n=0 ? 1 : n*factorial(n-1)
                      end
      factorial(3) * 7 }, естественно."
# Ответ равен 42, естественно.
```

При интерполяции глобальных переменных, а также переменных класса и экземпляра фигурные скобки можно опускать:

```
print "$gvar = $gvar и ivar = #@ivar."
```

Интерполяция не производится внутри строк, заключенных в одиночные кавычки (поскольку их значение не интерпретируется), но применима к заключенным в двойные кавычки встроенным документам и к регулярным выражениям.

2.22. Отложенная интерполяция

Иногда желательно отложить интерполяцию значений в строку. Идеального способа решить эту задачу не существует, но можно воспользоваться блоком:

```
str = proc { |x,y,z| "Числа равны #{x}, #{y} и #{z}" }
s1 = str.call(3,4,5) # Числа равны 3, 4 и 5.
s2 = str.call(7,8,9) # Числа равны 7, 8 и 9.
```

Другое, более громоздкое решение состоит в том, чтобы сохранить строку, заключенную в одиночные кавычки, потом «обернуть» ее двойными кавычками и вычислить:

```
str = '#{name} - мое имя, а #{nation} - моя родина'
name, nation = "Стивен Дедал", "Ирландия"
s1 = eval('"' + str + '"')
# Стивен Дедал - мое имя, а Ирландия - моя родина.
```

Можно также передать eval другую функцию привязки:

```
bind = proc do
  name, nation = "Гулливер Фойл", "Земля"
  binding
end.call # Надуманный пример; возвращает привязанный контекст блока
s2 = eval('"' + str + '"', bind)
# Гулливер Фойл - мое имя, а Земля - моя родина.
```

У техники работы с eval есть свои «причуды». Например, будьте осторожны, вставляя управляющие последовательности, скажем \n.

2.23. Разбор данных, разделенных запятыми

Данные, разделенные запятыми, часто встречаются при программировании. Это в некотором роде «наибольший общий делитель» всех форматов обмена данными. Например, так передаются данные между несовместимыми СУБД или приложениями, которые не поддерживают никакого другого общего формата.

Будем предполагать, что данные представляют собой строки и числа, а все строки заключены в кавычки. Еще предположим, что все символы должным образом экранированы (например, запятые и кавычки внутри строки).

Задача оказывается простой, поскольку такой формат данных подозрительно напоминает встроенные в Ruby массивы данных разных типов. Достаточно заключить все выражение в квадратные скобки, чтобы получить массив.

```
string = gets.chomp!
# Предположим, что прочитана такая строка:
# "Doe, John", 35, 225, "5'10\"", "555-0123"
data = eval("[ " + string + "]") # Преобразовать в массив.
data.each {|x| puts "Значение = #{x}"}
```

Этот код выводит такой результат:

```
Значение = Doe, John
Значение = 35
Значение = 225
Значение = 5' 10"
Значение = 555-0123
```

Более общее решение дает стандартная библиотека CSV. Есть также усовершенствованный инструмент под названием FasterCSV. Поищите его в сети, он не входит в стандартный дистрибутив Ruby.

2.24. Преобразование строки в число (десятичное или иное)

Есть два основных способа преобразовать строку в число: методы `Integer` и `Float` модуля `Kernel` и методы `to_i` и `to_f` класса `String`. (Имена, начинающиеся с прописной буквы, например `Integer`, обычно резервируются для специальных функций преобразования.)

Простой случай тривиален, следующие два предложения эквивалентны:

```
x = "123".to_i      # 123
y = Integer("123")  # 123
```

Но если в строке хранится не число, то поведение этих методов различается:

```
x = "junk".to_i      # Молча возвращает 0.
y = Integer("junk")  # Ошибка.
```

Метод `to_i` прекращает преобразование, как только встречает первый символ, не являющийся цифрой, а метод `Integer` в этом случае возбуждает исключение:

```
x = "123junk".to_i  # 123
y = Integer("123junk") # Ошибка.
```

Оба метода допускают наличие пропусков в начале и в конце строки:

```
x = " 123 ".to_i    # 123
y = Integer(" 123 ") # 123
```

Преобразование строки в число с плавающей точкой работает аналогично:

```
x = "3.1416".to_f    # 3.1416
y = Float("2.718")   # 2.718
```

Оба метода понимают научную нотацию:

```
x = Float("6.02e23") # 6.02e23
y = "2.9979246e5".to_f # 299792.46
```

Методы `to_i` и `Integer` также по-разному относятся к системе счисления. По умолчанию, естественно, подразумевается система по основанию 10, но другие тоже допускаются (это справедливо и для чисел с плавающей точкой).

Говоря о преобразовании из одной системы счисления в другую, мы всегда имеем в виду строки. Ведь целое число неизменно хранится в двоичном виде.

Следовательно, преобразование системы счисления – это всегда преобразование одной строки в другую. Здесь мы рассмотрим преобразование из строки (обратное преобразование рассматривается в разделах 5.18 и 5.5).

Числу в тексте программы может предшествовать префикс, обозначающий основание системы счисления. Префикс `0b` обозначает двоичное число, `0` – восьмеричное, а `0x` – шестнадцатеричное.

Метод `Integer` такие префиксы понимает, а метод `to_i` – нет:

```
x = Integer("0b111") # Двоичное - возвращает 7.
y = Integer("0111")  # Восьмеричное - возвращает 73.
z = Integer("0x111") # Шестнадцатеричное - возвращает 291.
```

```
x = "0b111".to_i    # 0
```

```
y = "0111".to_i      # 0
z = "0x111".to_i     # 0
```

Однако у метода `to_i` есть необязательный второй параметр для указания основания. Обычно применяют только четыре основания: 2, 8, 10 (по умолчанию) и 16. Впрочем, префиксы не распознаются даже при определении основания.

```
x = "111".to_i(2)      # 7
y = "111".to_i(8)      # Восьмеричное - возвращает 73.
z = "111".to_i(16)     # Шестнадцатеричное - возвращает 291.
```

```
x = "0b111".to_i      # 0
y = "0111".to_i      # 0
z = "0x111".to_i      # 0
```

Из-за «стандартного» поведения этих методов цифры, недопустимые при данном основании, обрабатываются по-разному:

```
x = "12389".to_i(8)    # 123 (8 игнорируется).
y = Integer("012389")  # Ошибка (8 недопустима).
```

Хотя полезность этого и сомнительна, метод `to_i` понимает основания вплоть до 36, когда в представлении числа допустимы все буквы латинского алфавита. (Возможно, это напомнило вам о `base64`-кодировании; дополнительную информацию по этому поводу вы найдете в разделе 2.37.)

```
x = "123".to_i(5)      # 66
y = "ruby".to_i(36)    # 1299022
```

Для преобразования символьной строки в число можно также воспользоваться методом `scanf` из стандартной библиотеки, которая добавляет его в модуль `Kernel`, а также классы `IO` и `String`:

```
str = "234 234 234"
x, y, z = str.scnf("%d %o %x") # 234, 156, 564
```

Метод `scanf` реализует всю имеющую смысл функциональность стандартных функций `scanf`, `sscanf` и `fscanf` из библиотеки языка C. Но строки, представляющие двоичные числа, он не обрабатывает.

2.25. Кодирование и декодирование строк в кодировке rot13

Rot13 – наверное, самый слабый из известных человечеству шифров. Исторически он просто препятствовал «случайному» прочтению текста. Он часто встречается в конференциях Usenet; например, так можно закодировать потенциально обидную шутку или сценарий фильма «Звездные войны. Эпизод 13» накануне премьеры.

Принцип кодирования состоит в смещении символов относительно начала алфавита (латинского) на 13: А превращается в N, В – в О и т.д. Строчные буквы смещаются на ту же величину; цифры, знаки препинания и прочие символы игнорируются. Поскольку 13 – это ровно половина от 26 (число букв в латинском алфавите), то функция является обратной самой себе, то есть ее повторное применение восстанавливает исходный текст.

Ниже приведена реализация этого метода, добавленного в класс `String`, никаких особых комментариев она не требует:

```
class String

  def rot13
    self.tr("A-Ma-mN-Zn-z", "N-Zn-zA-Ma-m")
  end

end

joke = "Y2K bug"
joke13 = joke.rot13 # "L2X oht"

episode2 = "Fcbvyre: Naanxva qbrfa'g trg xvyyrq."
puts episode2.rot13
```

2.26. Шифрование строк

Иногда нежелательно, чтобы строки можно было легко распознать. Например, пароли не следует хранить в открытом виде, какими бы ограничительными ни были права доступа к файлу.

В стандартном методе `crypt` применяется стандартная функция с тем же именем для шифрования строки по алгоритму DES. Она принимает в качестве параметра «затравку» (ее назначение то же, что у затравки генератора случайных чисел). На платформах, отличных от UNIX, параметр может быть иным.

Ниже показано тривиальное приложение, которое запрашивает пароль, знакомый любителям Толкиена:

```
coded = "hfCghNIE5LAM."

puts "Говори, друг, и жми Enter!"

print "Пароль: "
password = gets.chomp

if password.crypt("hf") == coded
  puts "Добро пожаловать!"
else
  puts "Кто ты, орк?"
end
```

Стоит отметить, что на такое шифрование не стоит полагаться в серверных Web-приложениях, поскольку пароль, введенный в поле формы, все равно передается по сети в открытом виде. В таких случаях проще всего воспользоваться протоколом SSL (Secure Sockets Layer). Разумеется, никто не запрещает пользоваться шифрованием на сервере, но по другой причине – чтобы защитить пароль в хранилище, а не во время передачи по сети.

2.27. Сжатие строк

Для сжатия строк и файлов применяется библиотека `zlib`.

Зачем может понадобиться сжимать строки? Возможно, чтобы ускорить ввод/вывод из базы данных, оптимизировать использование сети или усложнить распознавание строк.

В классах `Deflate` и `Inflate` имеются методы класса `deflate` и `inflate` соответственно. У метода `deflate` (он выполняет сжатие) есть дополнительный параметр, задающий режим сжатия. Он определяет компромисс между качеством сжатия и скоростью. Если значение равно `BEST_COMPRESSION`, то строка сжимается максимально, но это занимает сравнительно много времени. Значение `BEST_SPEED` задает максимальную скорость, но при этом строка сжимается хуже. Подразумеваемое по умолчанию значение `DEFAULT_COMPRESSION` выбирает компромиссный режим.

```
require 'zlib'
include Zlib
```

```
long_string = ("abcde"*71 + "defghi"*79 + "ghijkl"*113)*371
# long_string состоит из 559097 символов.
```

```
s1 = Deflate.deflate(long_string, BEST_SPEED)      # 4188 символов.
s3 = Deflate.deflate(long_string)                  # 3568 символов
s2 = Deflate.deflate(long_string, BEST_COMPRESSION) # 2120 символов
```

Неформальные эксперименты показывают, что скорость отличается примерно в два раза, а плотность сжатия – в обратной пропорции на ту же величину. И скорость, и плотность сильно зависят от состава строки. Разумеется, на скорость влияет и имеющееся оборудование.

Имейте в виду, что существует пороговое значение длины строки. Если строка коротче, то сжимать ее практически бесполезно (если только вы не хотите сделать ее нечитаемой). В этом случае неизбежные накладные расходы могут даже привести к тому, что сжатая строка окажется длиннее исходной.

2.28. Подсчет числа символов в строке

Метод `count` подсчитывает число вхождений в строку символов из заданного набора:

```
s1 = "abracadabra"
a = s1.count("c")    # 1
b = s1.count("bdr")  # 5
```

Строковый параметр ведет себя как простое регулярное выражение. Если он начинается с символа `^`, то берется дополнение к списку:

```
c = s1.count("^a")    # 6
d = s1.count("^bdr")  # 6
```

Дефис обозначает диапазон символов:

```
e = s1.count("a-d")    # 9
f = s1.count("^a-d")    # 2
```


2.29. Обращение строки

Для обращения строки служит метод `reverse` (или его вариант для обращения «на месте» `reverse!`):

```
s1 = "Star Trek"
s2 = s1.reverse      # "kerT ratS"
s1.reverse!          # s1 теперь равно "kerT ratS"
```

Пусть требуется обратить порядок слов (а не символов). Тогда можно сначала воспользоваться методом `String#split`, который вернет массив слов. В классе `Array` тоже есть метод `reverse`, поэтому можно обратить массив, а затем с помощью метода `join` объединить слова в новую строку:

```
phrase = "Now here's a sentence"
phrase.split(" ").reverse.join(" ")
# "sentence a here's Now"
```

2.30. Удаление дубликатов

Цепочки повторяющихся символов можно сжать до одного методом `squeeze`:

```
s1 = "bookkeeper"
s2 = s1.squeeze      # "bokeper"
s3 = "Hello..."
s4 = s3.squeeze      # "Helo."
```

Если указан параметр, то будут удаляться только дубликаты заданных в нем символов:

```
s5 = s3.squeeze(".") # "Hello."
```

Этот параметр подчиняется тем же правилам, что и параметр метода `count` (см. раздел 2.28), то есть допускаются дефис и символ `^`.

Имеется также метод `squeeze!`.

2.31. Удаление заданных символов

Метод `delete` удаляет из строки те символы, которые включены в список, переданный в качестве параметра:

```
s1 = "To be, or not to be"
s2 = s1.delete("b") # "To e, or not to e"
s3 = "Veni, vidi, vici!"
s4 = s3.delete(",!") # "Veni vidi vici"
```

Этот параметр подчиняется тем же правилам, что и параметр метода `count` (см. раздел 2.28), то есть допускаются символы - (дефис) и `^` (каре).

Имеется также метод `delete!`.

2.32. Печать специальных символов

Метод `dump` позволяет получить графическое представление символов, которые обычно не печатаются вовсе или вызывают побочные эффекты:

```
s1 = "Внимание" << 7 << 7 << 7 # Добавлено три символа ASCII BEL.
puts s1.dump                       # Печатается: Внимание\007\007\007
s2 = "abc\t\tdef\tghi\n\n"
puts s2.dump                       # Печатается: abc\t\tdef\tghi\n\n
s3 = "Двойная кавычка: \""
puts s3.dump                       # Печатается: Двойная кавычка: \"
```

При стандартном значении переменной \$KCODE метод `dump` дает такой же эффект, как вызов метода `inspect` для строки. Переменная \$KCODE рассматривается в главе 4.

2.33. Генерирование последовательности строк

Иногда бывает необходимо получить «следующую» строку. Так, следующей для строки «aaa» будет строка «aab» (затем «aac», «aad» и так далее).

В Ruby для этой цели есть метод `succ`:

```
droid = "R2D2"
improved = droid.succ # "R2D3"
pill = "Vitamin B"
pill2 = pill.succ     # "Vitamin C"
```

Не рекомендуется применять этот метод, если точно не известно, что начальное значение предсказуемо и разумно. Если начать с какой-нибудь экзотической строки, то рано или поздно вы получите странный результат.

Существует также метод `upto`, который в цикле вызывает `succ`, пока не будет достигнуто конечное значение:

```
"Files, A".upto "Files, X" do |letter|
  puts "Opening: #{letter}"
end
```

Выводится 24 строки.

Еще раз подчеркнем, что эта возможность используется редко, да и то на ваш страх и риск. Кстати, метода, возвращающего «предшествующую» строку, не существует.

2.34. Вычисление 32-разрядного CRC

Контрольный код циклической избыточности (Cyclic Redundancy Checksum, CRC) – хорошо известный способ получить «сигнатуру» файла или произвольного массива байтов. CRC обладает тем свойством, что вероятность получения одинакового кода для разных входных данных равна $1 / 2^{**N}$, где N – число битов результата (чаще всего 32).

Вычислить его позволяет библиотека `zlib`, написанная Уэно Кацухиро (Ueno Katsuhiko). Метод `crc32` вычисляет CRC для строки, переданной в качестве параметра.

```
require 'zlib'
include Zlib
crc = crc32("Hello") # 4157704578
crc = crc32(" world!", crc) # 461707669
crc = crc32("Hello world!") # 461707669 (то же, что и выше)
```

В качестве необязательного второго параметра можно передать ранее вычисленный CRC. Результат получится такой, как если бы конкатенировать обе строки и вычислить CRC для объединения. Это полезно, например, когда нужно вычислить CRC файла настолько большого, что прочитать его можно только по частям.

2.35. Вычисление MD5-свертки строки

Алгоритм MD5 вырабатывает 128-разрядный цифровой отпечаток или дайджест сообщения произвольной длины. Это разновидность свертки, то есть функция шифрования односторонняя, так что восстановить исходное сообщение по дайджесту невозможно. Для Ruby имеется расширение, реализующее MD5; интересующиеся могут найти его в каталоге `ext/md5` стандартного дистрибутива.

Для создания нового объекта MD5 есть два эквивалентных метода класса: `new` и `md5`:

```
require 'md5'
hash = MD5.md5
hash = MD5.new
```

Есть также четыре метода экземпляра: `clone`, `digest`, `hexdigest` и `update`. Метод `clone` просто копирует существующий объект, а метод `update` добавляет новые данные к объекту:

```
hash.update("Дополнительная информация...")
```

Можно создать объект и передать ему данные за одну операцию:

```
secret = MD5.new("Секретные данные")
```

Если задан строковый аргумент, он добавляется к объекту путем обращения к методу `update`. Повторные обращения эквивалентны одному вызову с конкатенированными аргументами:

```
# Эти два предложения:
cryptic.update("Данные...")
cryptic.update(" еще данные.")
# ...эквивалентны одному такому:
cryptic.update("Данные... еще данные.")
```

Метод `digest` возвращает 16-байтовую двоичную строку, содержащую 128-разрядный дайджест.

Но наиболее полезен метод `hexdigest`, который возвращает дайджест в виде строки в коде ASCII, состоящей из 32 шестнадцатеричных символов, соответствующих 16 байтам. Он эквивалентен следующему коду:

```
def hexdigest
  ret = ''
  digest.each_byte {|i| ret << sprintf('%02x', i) }
  ret
end

secret.hexdigest # "b30e77a94604b78bd7a7e64ad500f3c2"
```

```
m = MD5.new("Секретные данные").hexdigest
```

```
def levenshtein(other, ins=2, del=2, sub=1)
  # ins, del, sub - взвешенные стоимости.

  return nil if self.nil?
  return nil if other.nil?
  dm = []      # Матрица расстояний.

  # Инициализировать первую строку.
  dm[0] = (0..self.length).collect { |i| i * ins }
  fill = [0] * (self.length - 1)

  # Инициализировать первую колонку.
  for i in 1..other.length
    dm[i] = [i * del, fill.flatten]
  end

  # Заполнить матрицу.
  for i in 1..other.length
    for j in 1..self.length
      # Главное сравнение.
      dm[i][j] = [
        dm[i-1][j-1] +
          (self[j-1] == other[i-1] ? 0 : sub),
        dm[i][j-1] + ins,

```

```

        dm[i-1][j] + del
    ].min
end
end

# Последнее значение в матрице и есть
# расстояние Левенштейна между строками.
dm[other.length][self.length]
end

end

```

```

s1 = "ACUGAUGUGA"
s2 = "AUGGAA"
d1 = s1.levenshtein(s2)    # 9
s3 = "pennsylvania"
s4 = "pencilvaneya"
d2 = s3.levenshtein(s4)    # 7
s5 = "abcd"
s6 = "abcd"
d3 = s5.levenshtein(s6)    # 0

```

Определив расстояние Левенштейна, мы можем написать метод `similar?`, вычисляющий меру схожести строк. Например:

```

class String

  def similar?(other, thresh=2)
    if self.levenshtein(other) < thresh
      true
    else
      false
    end
  end
end

```

```

end

```

```

if "polarity".similar?("hilarity")
  puts "Электричество - забавная штука!"
end

```

Разумеется, можно было бы передать методу `similar?` три взвешенные стоимости, которые он в свою очередь передал бы методу `levenshtein`. Но для простоты мы не стали этого делать.

2.37. base64-кодирование и декодирование

Алгоритм *base64* часто применяется для преобразования двончных данных в текстовую форму, не содержащую специальных символов. Например, в конференциях так обмениваются исполняемыми файлами.

Простейший способ осуществить base64-кодирование и декодирование – воспользоваться встроенными возможностями Ruby. В классе Array есть метод `pack`, который возвращает строку в кодировке base64 (если передать ему параметр "m"). А в классе String есть метод `unpack`, который декодирует такую строку:

```
str = "\007\007\002\abdce"
new_string = [str].pack("m") # "BwcCB2JkY2U="
original = new_string.unpack("m") # ["\a\a\002\abdce"]
```

Отметим, что метод `unpack` возвращает массив.

2.38. Кодирование и декодирование строк (uencode/udecode)

Префикс `uu` в этих именах означает UNIX-to-UNIX. Утилиты `uencode` и `udecode` – это проверенный временем способ обмена данными в текстовой форме (аналогичный base64).

```
str = "\007\007\002\abdce"

new_string = [str].pack("u") # '(!P<!!V)D8V4'
original = new_string.unpack("u") # ["\a\a\002\abdce"]
```

Отметим, что метод `unpack` возвращает массив.

2.39. Замена символов табуляции пробелами и сворачивание пробелов в табуляторы

Бывает, что имеется строка с символами табуляции, а мы хотели бы преобразовать их в пробелы (или наоборот). Ниже показаны два метода, реализующих эти операции:

```
class String

  def detab(ts=8)
    str = self.dup
    while (leftmost = str.index("\t")) != nil
      space = " " * (ts - (leftmost % ts))
      str[leftmost] = space
    end
    str
  end

  def entab(ts=8)
    str = self.detab
    areas = str.length / ts
    newstr = ""
    for a in 0..areas
      temp = str[a*ts..a*ts+ts-1]
      if temp.size == ts
```

```

        if temp =~ / +/
            match=Regexp.last_match[0]
            endmatch = Regexp.new(match+"$")
            if match.length>1
                temp.sub!(endmatch,"\t")
            end
        end
    end
    newstr += temp
end
newstr
end
,

end

foo = "Это      всего      лишь      тест.  "

puts foo
puts foo.entab(4)
puts foo.entab(4).dump

```

Отметим, что этот код не распознает символы забоя.

2.40. Цитирование текста

Иногда бывает необходимо напечатать длинные строки текста, задав ширину поля. Приведенный ниже код решает эту задачу, разбивая текст по границам слов и учитывая символы табуляции (но символы забоя не учитываются, а табуляция не сохраняется):

```

str = <<-EOF
When in the Course of human events it becomes necessary
for one people to dissolve the political bands which have
connected them with another, and to assume among the powers
of the earth the separate and equal station to which the Laws
of Nature and of Nature's God entitle them, a decent respect
for the opinions of mankind requires that they should declare
the causes which impel them to the separation.
EOF

max = 20

line = 0
out = [""]

input = str.gsub(/\n/, " ")
words = input.split(" ")
while input != ""

```

```

word = words.shift
break if not word
if out[line].length + word.length > max
  out[line].squeeze!(" ")
  line += 1
  out[line] = ""
end
out[line] << word + " "
end

```

`out.each {|line| puts line}` # Печатает 24 очень коротких строки.

Библиотека `Format` решает как эту, так и много других схожих задач. Поищите ее в сети.

2.41. Заключение

Мы обсудили основы представления строк (заклученных в одиночные или двойные кавычки). Поговорили о том, как интерполировать выражения в строку в двойных кавычках; узнали, что в таких строках допустимы некоторые специальные символы, представленные управляющими последовательностями. Кроме того, мы познакомились с конструкциями `%q` и `%Q`, которые позволяют нам по своему вкусу выбирать ограничители. Наконец, рассмотрели синтаксис встроенных документов, унаследованных из старых продуктов, в том числе командных интерпретаторов в UNIX.

В этой главе были продемонстрированы все наиболее важные операции, которые программисты обычно выполняют над строками: конкатенация, поиск, извлечение подстрок, разбиение на лексемы и т.д. Мы видели, как можно кодировать строки (например, по алгоритму `base64`) и сжимать их.

Пришло время перейти к тесно связанной со строками теме – регулярным выражениям. Регулярные выражения – это мощное средства сопоставления строк с образцами. Мы рассмотрим их в следующей главе.

Глава 3. Регулярные выражения

.....
Я провела бы его по лабиринту, где тропы орнаментом украшены...
Эми Лоуэлл

Мощь регулярных выражений как инструмента программирования часто недооценивается. Первые теоретические исследования на эту тему датируются сороковыми годами прошлого века, в вычислительные системы они проникли в 1960-х годах, а затем были включены в различные инструментальные средства операционной системы UNIX. В 1990-х годах популярность языка Perl привела к тому, что регулярные выражения вошли в обиход, перестав быть уделом бородатых гуру.

Красота регулярных выражений заключается в том, что почти весь наш опыт можно выразить в терминах образцов. А если имеется образец, то можно провести сопоставление с ним, можно найти то, что ему соответствует, и заменить найденное чем-то другим по своему выбору.

Во время работы над данной книгой язык Ruby находился в переходном состоянии. Старая библиотека регулярных выражений заменялась новой под названием Oniguruma. Этой библиотеке посвящен раздел 3.13 данной главы. Что касается интернационализации, то это тема главы 4.

3.1. Синтаксис регулярных выражений

Обычно регулярное выражение ограничено с двух сторон символами косой черты. Применяется также форма `%r`. В таблице 3.1 приведены примеры простых регулярных выражений:

Таблица 3.1. Простые регулярные выражения

Регулярное выражение	Пояснение
<code>/Ruby/</code>	Соответствует одному слову Ruby
<code>/[Rr]uby/</code>	Соответствует Ruby или ruby
<code>/^abc/</code>	Соответствует abc в начале строки
<code>%r(xyz\$)</code>	Соответствует xyz в конце строки
<code>%r [0-9]* </code>	Соответствует любой последовательности из нуля или более цифр

Сразу после регулярного выражения можно поместить однобуквенный модификатор. В таблице 3.2 приведены наиболее часто употребляемые модификаторы.

Таблица 3.2. Модификаторы регулярных выражений

Модификатор	Назначение
I	Игнорировать регистр
O	Выполнять подстановку выражения только один раз
M	Многострочный режим (точка сопоставляется с символом новой строки)
X	Обобщенное регулярное выражение (допускаются пробелы и комментарии)

Дополнительные примеры будут рассмотрены в главе 4.

Чтобы завершить введение в регулярные выражения, в таблице 3.3 мы приведем наиболее употребительные символы и обозначения.

Таблица 3.3. Общеупотребительные обозначения в регулярных выражениях

Обозначение	Пояснение
^	Начало строки текста (line) или строки символов (string)
\$	Конец строки текста или строки символов
.	Любой символ, кроме символа новой строки (если не установлен многострочный режим)
\w	Символ – часть слова (цифра, буква или знак подчеркивания)
\W	Символ, не являющийся частью слова
\s	Пропуск (пробел, знак табуляции, символ новой строки и т.д.)
\S	Символ, не являющийся пропуском
\d	Цифра (то же, что [0-9])
\D	Не цифра
\A	Начало строки символов (string)
\Z	Конец строки символов или позиция перед конечным символом новой строки
\z	Конец строки символов (string)
\b	Граница слова (только вне квадратных скобок [])
\B	Не граница слова
\b	Забой (только внутри квадратных скобок [])
[]	Произвольный набор символов
*	0 или более повторений предыдущего подвыражения
*?	0 или более повторений предыдущего подвыражения (нежадный алгоритм)
+	1 или более повторений предыдущего подвыражения
+?	1 или более повторений предыдущего подвыражения (нежадный алгоритм)
{m,n}	От m до n вхождений предыдущего подвыражения
{m,n}?	От m до n вхождений предыдущего подвыражения (нежадный алгоритм)
?	0 или 1 повторений предыдущего подвыражения

Таблица 3.3. Общеупотребительные обозначения в регулярных выражениях

Обозначение	Пояснение
	Альтернативы
(?=)	Позитивное заглядывание вперед
(?!)	Негативное заглядывание вперед
()	Группировка подвыражений
(>)	Вложенное подвыражение
(?:)	Несохраняющая группировка подвыражений
(?imx-imx)	Включить/выключить режимы, начиная с этого места
(?imx-imx: expr)	Включить/выключить режимы для этого выражения
(?#)	Комментарий

Умение работать с регулярными выражениями – большой плюс для современного программиста. Полное рассмотрение этой темы выходит далеко за рамки настоящей книги, но, если вам интересно, можете обратиться к книге Jeffrey Friedl, *Mastering Regular Expressions*^{*}.

Дополнительный материал вы также найдете в разделе 3.13.

3.2. Компиляция регулярных выражений

Для компиляции регулярных выражений предназначен метод `Regexp.compile` (синоним `Regexp.new`). Первый параметр обязателен, он может быть строкой или регулярным выражением. (Отметим, что если этот параметр является регулярным выражением с дополнительными флагами, то флаги не будут перенесены в новое откомпилированное выражение.)

```
pat1 = Regexp.compile(/^foo.*/) # /^foo.*/
pat2 = Regexp.compile(/bar$/i)  # /bar/ (i не переносится)
```

Если второй параметр задан, обычно это поразрядное объединение (ИЛИ) каких-либо из следующих констант: `Regexp::EXTENDED`, `Regexp::IGNORECASE`, `Regexp::MULTILINE`. При этом любое отличное от `nil` значение приведет к тому, что регулярное выражение не будет различать регистры; мы рекомендуем опускать второй параметр.

```
options = Regexp::MULTILINE || Regexp::IGNORECASE
pat3 = Regexp.compile(/^foo", options)
pat4 = Regexp.compile(/bar/, Regexp::IGNORECASE)
```

Третий параметр, если он задан, включает поддержку многобайтных символов. Он может принимать одно из четырех значений:

- "N" или "n" означает отсутствие поддержки
- "E" или "e" означает EUC

^{*} Дж. Фридл. Регулярные выражения. – Питер, 2003 (Прим. перев.)

"S" или "s" означает Shift-JIS

"U" или "u" означает UTF-8

Литеральное регулярное выражение можно задавать и не вызывая метод `new` или `compile`. Достаточно заключить его в ограничители (символы косой черты).

```
pat1 = /^foo.$/
```

```
pat2 = /bar$/i
```

Более подробная информация приводится в главе 4.

3.3. Экранирование специальных символов

Метод класса `Regexp.escape` экранирует все специальные символы, встречающиеся в регулярном выражении. К их числу относятся звездочка, вопросительный знак и квадратные скобки.

```
str1 = "[*?]"
```

```
str2 = Regexp.escape(str1) # "\[*\?\]"
```

Синонимом является метод `Regexp.quote`.

3.4. Якоря

Якорь – это специальное выражение, соответствующее позиции в строке, а не конкретному символу или последовательности символов. Позже мы увидим, что это простой частный случай утверждения нулевой длины, то есть соответствия, которое не продвигает просмотр исходной строки ни на одну позицию.

Наиболее употребительные якоря уже были представлены в начале главы. Простейшими из них являются `^` и `$`, которые соответствуют началу и концу строки символов.

```
string = "abcXdefXghi"
/def/   =~ string    # 4
/abc/   =~ string    # 0
/ghi/   =~ string    # 8
/^def/  =~ string    # nil
/def$/  =~ string    # nil
/^abc/  =~ string    # 0
/ghi$/  =~ string    # 8
```

Впрочем, я немного уклонился от истины. Эти якоря на самом деле соответствуют началу и концу не строки символов (`string`), а строки текста (`line`). Вот что произойдет, если те же самые образцы применить к строке, внутри которой есть символы новой строки:

```
string = "abc\ndef\nghi"
/def/   =~ string    # 4
/abc/   =~ string    # 0
/ghi/   =~ string    # 8
/^def/  =~ string    # 4
/def$/  =~ string    # 4
/^abc/  =~ string    # 0
/ghi$/  =~ string    # 8
```

Однако имеются якоря `\A` и `\Z`, которые соответствуют именно началу и концу самой строки символов.

```
string = "abc\ndef\nghi"
/\Adef/ =~ string      # nil
/def\Z/ =~ string      # nil
/\Aabc/ =~ string      # 0
/ghi\Z/ =~ string      # 8
```

Якорь `\z` отличается от `\Z` тем, что последний устанавливает соответствие перед конечным символом новой строки, а первый должен соответствовать явно.

```
string = "abc\ndef\nghi"
str2 << "\n"
/ghi\Z/ =~ string      # 8
/\Aabc/ =~ str2        # 8
/ghi\z/ =~ string      # 8
/ghi\z/ =~ str2        # nil
```

Можно также устанавливать соответствие на границе слова с помощью якоря `\b` или с позицией, которая не находится на границе слова (`\b`). Примеры использования метода `gsub` показывают, как эти якоря работают:

```
str = "this is a test"
str.gsub(/\b/, "|")    # "|this| |is| |a| |test|"
str.gsub(/\B/, "-")    # "t-h-i-s i-s a t-e-s-t"
```

Не существует способа отличить начало слова от конца.

3.5. Кванторы

Немалая часть аппарата регулярных выражений связана с обработкой необязательных элементов и повторений. Элемент, за которым следует вопросительный знак, необязателен; он может присутствовать или отсутствовать, а общее соответствие зависит от прочих частей регулярного выражения. (Этот квантор имеет смысл применять только к подвыражению ненулевой длины, но не к якорям.)

```
pattern = /ax?b/
pat2    = /a[xy]?b/
pattern =~ "ab"      # 0
pattern =~ "acb"     # nil
pattern =~ "axb"     # 0
pat2    =~ "ayb"     # 0
pat2    =~ "acb"     # nil
```

Элементы часто повторяются неопределенное число раз (для формулировки этого условия служит квантор `+`). Например, следующий образец соответствует любому положительному числу:

```
pattern = /[0-9]+/
pattern =~ "1"      # 0
pattern =~ "2345678" # 0
```

Еще один типичный случай – образец, повторяющийся нуль или более раз. Конечно, это условие можно выразить с помощью кванторов `+` и `?`. Вот, например, как сказать, что после строки `Huzzah` должно быть нуль или более восклицательных знаков:

```
pattern = /Huzzah(!+)?/ # Скобки здесь обязательны.
pattern =~ "Huzzah"      # 0
pattern =~ "Huzzah!!!!"  # 0
```

Но есть и способ лучше. Требуемое поведение описывается квантором `*`.

```
pattern = /Huzzah!*/      # * применяется только к символу !
pattern =~ "Huzzah"       # 0
pattern =~ "Huzzah!!!!"   # 0
```

Как распознать американский номер социального страхования? С помощью такого образа:

```
ssn = "987-65-4320"
pattern = /\d\d\d-\d\d-\d\d\d\d\d/
pattern =~ ssn           # 0
```

Но это не вполне понятно. Лучше явно сказать, сколько цифр должно быть в каждой группе. Это можно сделать, указав число повторений в фигурных скобках:

```
pattern = /\d{3}-\d{2}-\d{4}/
```

Необязательно, что такой образец будет короче, но он более понятен читателю программы.

Можно также использовать диапазоны, границы которых разделены запятой. Предположим, что номер телефона в Элбонии состоит из двух частей: в первой может быть от трех до пяти цифр, а во второй – от трех до семи. Вот как выглядит соответствующий образец:

```
elbonian_phone = /\d{3,5}-\d{3,7}/
```

Нижняя и верхняя границы диапазона необязательны (но хотя бы одна должна быть задана):

```
/x{5}/      # Соответствует 5 х.
/x{5,7}/    # Соответствует 5-7 х.
/x{,8}/     # Соответствует не более 8 х.
/x{3,}/     # Соответствует по меньшей мере 3 х.
```

Ясно, что кванторы `?`, `+` и `*` можно переписать и так:

```
/x?/        # То же, что /x{0,1}/
/x*/        # То же, что /x{0,}/
/x+/        # То же, что /x{1,}/
```

Фразеология, применяемая при описании регулярных выражений, изобилует яркими терминами: жадный (greedy), неохотный (reluctant), ленивый (lazy) и собственнический (possessive). Самым важным является различие между жадными и нежадными выражениями.

Рассмотрим следующий фрагмент кода. На первый взгляд, это регулярное выражение должно сопоставляться со строкой "where the", но на самом деле ему соответствует более длинная подстрока "where the sea meets the":

```
str = "where the sea meets the moon-blanch'd land,"
match = /.*/.match(str)
p match[0] # Вывести полученное соответствие:
           # "where the sea meets the"
```

Причина состоит в том, что оператор `*` выполняет жадное сопоставление, то есть продвигается так далеко по строке, как только можно, в поисках самого длинного соответствия. Чтобы излечить его от жадности, нужно добавить вопросительный знак:

```
str = "Where the sea meets the moon-blanch'd land,"
match = /.?*the/.match(str)
p match[0]                                # Вывести полученное соответствие:
                                           # "Where the" .
```

Итак, оператор `*` жадный, если за ним не стоит `?`. То же самое относится к кванторам `+` и `{m,n}` и даже к самому квантору `?`.

Я не сумел найти разумных примеров применения конструкций `{m,n}?` и `??`. Если вам о них известно, пожалуйста, поделитесь со мной своим опытом.

Дополнительная информация о кванторах содержится в разделе 3.13.

3.6. Позитивное и негативное заглядывание вперед

Понятно, что регулярное выражение сопоставляется со строкой линейно (осуществляя при необходимости возвраты). Поэтому существует понятие «текущего положения» в строке, это аналог указателя файла или курсора.

Термин «заглядывание» означает попытку сопоставить часть строки, находящуюся дальше текущего положения. Это утверждение нулевой длины, поскольку даже если соответствие будет найдено, никакого продвижения по строке не произойдет (то есть текущее положение не изменится).

В следующем примере строка "New World" будет сопоставлена, если за ней следует одна из строк "Symphony" или "Dictionary". Однако третье слово не будет частью соответствия.

```
s1 = "New World Dictionary"
s2 = "New World Symphony"
s3 = "New World Order"

reg = /New World(?: Dictionary| Symphony)/
m1 = reg.match(s1)
m.to_a[0]                                # "New World"
m2 = reg.match(s2)
m.to_a[0]                                # "New World"
m3 = reg.match(s3)                        # nil
```

Вот пример негативного заглядывания:

```
reg2 = /New World(?:! Symphony)/
m1 = reg2.match(s1)
m.to_a[0]                                # "New World"
m2 = reg2.match(s2)
m.to_a[0]                                # nil
m3 = reg2.match(s3)                       # "New World"
```

В данном случае строка "New World" подходит, только если за ней не следует строка "Symphony".

3.7. Обратные ссылки

Каждая заключенная в круглые скобки часть регулярного выражения является отдельным соответствием. Они нумеруются, и есть несколько способов сослаться на такие части по номерам. Сначала рассмотрим традиционный «некрасивый» способ.

Сослаться на группы можно с помощью глобальных переменных \$1, \$2 и т.д.:

```
str = "a123b45c678"
if /(a\d+)(b\d+)(c\d+)/ =~ str
  puts "Частичные соответствия: '#$1', '#$2', '#$3'"
  # Печатается: Частичные соответствия: 'a123', 'b45', 'c768'
end
```

Эти переменные нельзя использовать в подставляемой строке в методах sub и gsub:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=#$1, 2nd=#$2, 3rd=#$3")
# "1st=, 2nd=, 3rd="
```

Почему такая конструкция не работает? Потому что аргументы sub вычисляются перед вызовом sub. Вот эквивалентный код:

```
str = "a123b45c678"
s2 = "1st=#$1, 2nd=#$2, 3rd=#$3"
reg = /(a\d+)(b\d+)(c\d+)/
str.sub(reg, s2)
# "1st=, 2nd=, 3rd="
```

Отсюда совершенно понятно, что значения \$1, \$2, \$3 никак не связаны с сопоставлением, которое делается внутри вызова sub.

В такой ситуации на помощь приходят специальные коды \1, \2 и т.д.:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, '1st=\1, 2nd=\2, 3rd=\3')
# "1st=a123, 2nd=b45, 3rd=c768"
```

Обратите внимание на одиночные (твердые) кавычки в предыдущем примере. Если бы мы воспользовались двойными (мягкими) кавычками, не приняв никаких мер предосторожности, то элементы, которым предшествует обратная косая черта, были бы интерпретированы как восьмеричные числа:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\1, 2nd=\2, 3rd=\3")
# "1st=\001, 2nd=\002, 3rd=\003"
```

Обойти эту неприятность можно за счет двойного экранирования:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\\1, 2nd=\\2, 3rd=\\3")
# "1st=a123, 2nd=b45, 3rd=c678"
```

Допустима и блочная форма подстановки, в которой можно использовать глобальные переменные:

```
str = "a123b45c678"
str.sub{/(a\d+)(b\d+)(c\d+)/} { "1st=#$1, 2nd=#$2, 3rd=#$3" }
# "1st=a123, 2nd=b45, 3rd=c678"
```


При таком применении блока числа с обратной косой чертой нельзя использовать ни в двойных, ни в одиночных кавычках. Если вы немного поразмыслите, то поймете, что это разумно.

Упомяну попутно о том, что существуют незапоминаемые группы (noncapturing groups). Иногда при составлении регулярного выражения нужно сгруппировать символы, но чему будет соответствовать в конечном счете такая группа, несущественно. На этот случай и предусмотрены незапоминаемые группы, описываемые синтаксической конструкцией `(?:...)`:

```
str = "a123b45c678"
str.sub(/(a\d+)(?:b\d+)(c\d+)/, "1st=\1, 2nd=\2, 3rd=\3")
# "1st=a123, 2nd=c678, 3rd="
```

В предыдущем примере вторая группа не запоминается, поэтому та группа, которая должна была бы быть третьей, становится второй.

Лично мне не нравится ни одна из двух нотаций `(\1` и `$1)`. Иногда они удобны, но никогда не бывают необходимы. Все можно сделать «красивее», в объектно-ориентированной манере.

Метод класса `Regexp.last_match` возвращает объект класса `MatchData` (как и метод экземпляра `match`). У этого объекта есть методы экземпляра, с помощью которых программист может получить обратные ссылки.

Обращаться к объекту `MatchData` можно с помощью квадратных скобок, как если бы это был массив соответствий. Специальный элемент с индексом 0 содержит текст всей сопоставляемой строки, а элемент с индексом `n` ссылается на `n`-ую запомненную группу:

```
pat = /(.[aiu])(.[aiu])(.[aiu])(.[aiu])/i
# В этом образце есть четыре одинаковых группы.
refs = pat.match("Fujiyama")
# refs is now: ["Fujiyama", "Fu", "ji", "ya", "ma"]
x = refs[1]
y = refs[2..3]
refs.to_a.each {|x| print "#{x}\n"}
```

Отметим, что объект `refs` – не настоящий массив. Поэтому, если мы хотим обращаться с ним как с таковым, применяя итератор `each`, следует сначала преобразовать его в массив с помощью метода `to_a` (как показано в примере).

Есть и другие способы нахождения сопоставленной подстроки внутри исходной строки. Методы `begin` и `end` возвращают смещения начала и конца соответствия. (Важно понимать, что смещение конца – это индекс символа, следующего за найденным соответствием.)

```
str = "alpha beta gamma delta epsilon"
#      0....5....0....5....0....5....
#      (для удобства подсчета)
```

```
pat = /(b[^ ]+ )(g[^ ]+ )(d[^ ]+ )/
# Три слова, каждое из которых представляет собой отдельное соответствие.
refs = pat.match(str)
```

```
# "beta "
p1 = refs.begin(1)      # 6
p2 = refs.end(1)        # 11
# "gamma "
p3 = refs.begin(2)      # 11
p4 = refs.end(2)        # 17
# "delta "
p5 = refs.begin(3)      # 17
p6 = refs.end(3)        # 23
# "beta gamma delta"
p7 = refs.begin(0)      # 6
p8 = refs.end(0)        # 23
```

Аналогично метод `offset` возвращает массив из двух чисел: смещение начала и смещение конца соответствия. Продолжим предыдущий пример:

```
range0 = refs.offset(0) # [6,23]
range1 = refs.offset(1) # [6,11]
range2 = refs.offset(2) # [11,17]
range3 = refs.offset(3) # [17,23]
```

Части строки, которые находятся перед сопоставленной подстроки и после нее, можно получить методами `pre_match` и `post_match` соответственно. В том же коде:

```
before = refs.pre_match # "alpha "
after  = refs.post_match # "epsilon"
```

3.8. Классы символов

Классы символов – это просто форма перечисления (указание альтернатив), в котором каждая группа состоит из одного символа. В простейшем случае список возможных символов заключается в квадратные скобки:

```
/[aeiou]/ # Соответствует любой из букв a, e, i, o, u; эквивалентно
          # /(a|e|i|o|u)/, только группа не запоминается.
```

Внутри класса символов управляющие последовательности типа `\n` по-прежнему распознаются, но такие метасимволы, как `.` и `?`, не имеют специального смысла:

```
/[.\n?]/ # Сопоставляется с точкой, символом новой строки,
          # вопросительным знаком.
```

Символ каре (`^`) внутри класса символов имеет специальный смысл, если находится в начале; в этом случае он формирует дополнение к списку символов:

```
/^aeiou/ # Любой символ, КРОМЕ a, e, i, o, u.
```

Дефис внутри класса символов обозначает диапазон (в лексикографическом порядке):

```
/[a-mA-M]/ # Любой символ из первой половины алфавита.
/^a-mA-M]/ # Любой ДРУГОЙ символ, а также цифры и символы, отличные
          # от букв и цифр.
```

Дефис в начале или в конце класса символов, а также каре в середине теряют специальный смысл и интерпретируются буквально. То же относится

к левой квадратной скобке, но правая квадратная скобка, очевидно, должна экранироваться:

```
/[-^[\]]/ # Сопоставляется с дефисом, каре и правой квадратной скобкой.
```

Регулярные выражения в Ruby могут содержать ссылки на именованные классы символов вида `[[:name:]]`. Так, `[[:digit:]]` означает то же самое, что образец `[0-9]`. Во многих случаях такая запись оказывается короче или, по крайней мере, понятнее.

Есть еще такие именованные классы: `[[:print:]]` (символы, имеющие графическое начертание) и `[[:alpha:]]` (буквы):

```
s1 = "abc\007def"
/[[:print:]]*/.match(s1)
m1 = Regexp::last_match[0] # "abc"

s2 = "1234def"
/[[:digit:]]*/.match(s2)
m2 = Regexp::last_match[0] # "1234"

/[[:digit:]]+[[:alpha:]]/.match(s2)
m3 = Regexp::last_match[0] # "1234d"
```

Каре перед именем класса символов формирует его дополнение:

```
/[^[:alpha:]]/ # Все символы, кроме букв.
```

Для многих классов имеется также сокращенная нотация. Наиболее распространены сокращения `\d` (любая цифра), `\w` (любой символ, входящий в состав «слова») и `\s` (пропуски – пробел, знак табуляции или новой строки):

```
str1 = "Wolf 359"
/\w+/.match(str1) # Соответствует "Wolf" (то же, что /[a-zA-Z_0-9]+)/
/\w+ \d+/.match(str1) # Соответствует "Wolf 359"
/\w+ \w+/.match(str1) # Соответствует "Wolf 359"
/\s+/.match(str1) # Соответствует " "
```

«Дополнительные» формы обычно записываются в виде прописной буквы:

```
/\W/ # Любой символ, не входящий в состав слова.
/\D/ # Все кроме цифр.
/\S/ # Все кроме пропусков.
```

Дополнительная информация, относящаяся только к Oniguruma, приводится в разделе 3.13.

3.9. Обобщенные регулярные выражения

Регулярные выражения, особенно длинные, часто выглядят загадочно. Модификатор `x` позволяет записывать регулярное выражение на нескольких строках. При этом пробелы и символы новой строки игнорируются, так что можно делать для наглядности отступы. Заодно разрешается оставлять комментарии, хотя это возможно даже в простых регулярных выражениях.

Чтобы привести несколько искусственный пример умеренно сложного регулярного выражения, предположим, что имеется такой список адресов:

```
addresses =
[ "409 W Jackson Ave",      "No. 27 Grande Place",
  "16000 Pennsylvania Avenue", "2367 St. George St.",
  "22 Rue Morgue",          "33 Rue St. Denis",
  "44 Rue Zeeday",          "55 Santa Monica Blvd.",
  "123 Main St., Apt. 234",  "123 Main St., #234",
  "345 Euneva Avenue, Suite 23", "678 Euneva Ave, Suite A"]
```

Здесь каждый адрес состоит из трех частей: номер дома, название улицы и необязательный номер квартиры. Я предполагаю, что перед числом может быть необязательная строка No., а точку в ней можно опускать. Еще предположим, что название улицы может включать символы, обычно входящие в состав слова, а также апостроф, дефис и точку. Наконец, если адрес содержит необязательный номер квартиры, то ему должны предшествовать запятая и одна из строк Apt., Suite или # (знак номера).

Вот какое регулярное выражение я составил для разбора адреса. Обратите внимание, насколько подробно оно прокомментировано (может быть, даже излишне подробно):

```
regex = / ^                # Начало строки.
        ((No\.\?)\s+)?      # Необязательно: No[.]
        \d+ \s+             # Цифры и пробелы.
        ((\w|['-])+)        # Название улицы... может
        \s*                 # состоять из нескольких слов.
    )+
    (,\s*                   # Необязательно: запятая и т.д.
     (Apt\.\?|Suite|#)     # Apt[.], Suite, #
     \s+                   # Пробелы.
     (\d+|[A-Z]))          # Цифры или одна буква.
    )?
    $                       # Конец строки.
/x
```

Идея понятна. Когда сложность регулярного выражения достигает некоего порога (какого именно – дело вкуса), делайте его обобщенным, чтобы можно было добавить форматирование и комментарий.

Возможно, вы заметили, что я пользовался обычными комментариями Ruby (# ...), а не специальными, применяемыми в регулярных выражениях (?# ...). Почему? Просто потому, что это разрешено! Специальный комментарий необходим только тогда, когда его следует закончить раньше конца строки (например, если в той же строке за комментарием продолжается регулярное выражение).

3.10. Сопоставление точки символу конца строки

Обычно точка соответствует любому символу, кроме конца строки. Если задан модификатор многострочности `m`, точка будет сопоставляться и с этим символом. Другой способ – задать флаг `Regexp::MULTILINE` при создании регулярного выражения:

```
str = "Rubies are red\nAnd violets are blue.\n"
pat1 = /red./
pat2 = /red./m
```

```
str =~ pat1      # nil
str =~ pat2      # 11
```

Этот режим не оказывает влияния на то, где устанавливается соответствие якорям (^, \$, \A, \Z). Изменяется только способ сопоставления с точкой.

3.11. Внутренние модификаторы

Обычно модификаторы (например, `i` или `m`) задаются после регулярного выражения. Но что если мы хотим применить модификатор только к части выражения?

Существует специальная нотация для включения и исключения модификаторов. Заключенный в круглые скобки вопросительный знак, за которым следует один или несколько модификаторов, «включает» их до конца регулярного выражения. А если некоторым модификаторам предшествует минус, то соответствующие режимы «выключаются»:

```
/abc(?i)def/      # Соответствует abcdef, abcDEF, abcDef, ...,
                  # но не ABCdef.
/ab(?i)cd(?-i)ef/ # Соответствует abcdef, abCDef, abcDef, ...,
                  # но не ABCdef или abcdEF.
/(?imx).*/        # То же, что /.*/imx
/abc(?i-m).*/m     # Для последней части регулярного выражения включить
                  # распознавание регистра, выключить многострочный
                  # режим.
```

При желании можно поставить перед подвыражением двоеточие, и тогда заданные модификаторы будут действовать только для этого подвыражения:

```
/ab(?i:cd)ef/      # То же, что /ab(?i)cd(?-i)ef/
```

По техническим причинам использовать таким образом модификатор `o` нельзя. Модификатор `x` — можно, но я не знаю, кому бы это могло понадобиться.

3.12. Внутренние подвыражения

Для указания подвыражений применяется нотация `>>`:

```
re = /(??>abc)(??>def)/      # То же, что /abcdef/
re.match("abcdef").to_a      # ["abcdef"]
```

Отметим, что наличие подвыражения еще не означает группировки. С помощью дополнительных скобок их, конечно, можно превратить в запоминаемые группы.

Еще обратим внимание на то, что эта конструкция собственническая, то есть жадная и при этом не допускает возврата в подвыражение.

```
str = "abccccdef"
re1 = /(abc*)cdef/
re2 = /(??>abc*)cdef/
```

```
re1 =~ str          # 0
re2 =~ str          # nil
re1.match(str).to_a # ["abcccccdef", "abccc"]
re2.match(str).to_a # []
```

В предыдущем примере подвыражение `abc*` выражения `re2` поглощает все вхождения буквы `c` и (в соответствии с собственническим инстинктом) не отдает их назад, препятствуя возврату.

3.13. Ruby и Oniguruma

Новая библиотека регулярных выражений в Ruby называется *Oniguruma*. Это японское слово означает что-то вроде «колесо духов». (Те, кто не владеет японским, часто пишут его неправильно; имейте в виду, что тут не обойтись без «гугл»!)

Новая библиотека превосходит старую в нескольких отношениях. Прежде всего, она лучше работает с иноязычными строками, а также добавляет кое-какие интересные возможности к регулярным выражениям. Наконец, лицензия на ее использование мягче, чем на использование Ruby в целом. Когда писалась эта книга, *Oniguruma* еще не была полностью интегрирована в Ruby.

В следующем разделе мы расскажем, как определить, присутствует ли библиотека *Oniguruma*. А затем покажем, как можно ее собрать, если она не включена в дистрибутив.

3.13.1. Проверка наличия Oniguruma

Если вас интересует библиотека *Oniguruma*, то первым делом нужно выяснить, есть ли она в вашем экземпляре Ruby. В версиях 1.8.4 и младше ее, скорее всего, нет. Стандартно она включается в дистрибутив версии 1.9.

Вот как можно без труда выяснить, присутствует ли *Oniguruma*, проверив три условия. Во-первых, как я сказал, она стандартно поставляется в версии 1.9 и старше. В последних версиях обеих библиотек для работы с регулярными выражениями определена строковая константа `Regexp::ENGINE`. Если она содержит подстроку *Oniguruma*, то у вас новая библиотека. И последний шаг: если вы все еще не знаете, с какой библиотекой работаете, можно попытаться вычислить регулярное выражение, записанное в «новом» синтаксисе. Если при этом возникнет исключение `SyntaxError`, значит, у вас старая библиотека; в противном случае — новая.

```
def oniguruma?
  return true if RUBY_VERSION >= "1.9.0"

  if defined?(Regexp::ENGINE) # Константа ENGINE определена?
    if Regexp::ENGINE.include?('Oniguruma')
      return true           # Какая-то версия Oniguruma.
    else
      return false         # Старая библиотека.
    end
  end

  eval("/(?<!a)b/")          # Новый синтаксис.
```

```

return true
rescue SyntaxError
  return false
end

puts oniguruma?

```

Сработало: новая библиотека.
Не сработало: старая библиотека.

3.13.2. Сборка Oniguruma

Если в вашу версию библиотека Oniguruma не включена, можете самостоятельно откомпилировать Ruby и скомпоновать с недостающей библиотекой. Ниже приведены соответствующие инструкции. Эта процедура должна работать начиная с версии 1.6.8 (хотя она уже совсем старенькая).

Получить исходный текст Oniguruma можно из архива приложений Ruby RAA (<http://raa.ruby-lang.org/>) или найти в другом месте. Исходные тексты Ruby, естественно, находятся на официальном сайте.

Если вы работаете на платформе UNIX (в том числе в среде Cygwin в Windows или Mac OS/X), выполните следующие действия:

1. `gunzip oniguruma.tar.gz`
2. `tar xvf oniguruma.tar`
3. `cd oniguruma`
4. `./configure with-rubydir=<ruby-source-dir>`
5. Одно из следующих:
 - `make 16` # Для Ruby 1.6.8
 - `make 18` # Для Ruby 1.8.0/1.8.1
6. `cd ruby-source-dir`
7. `./configure`
8. `make clean`
9. `make`
10. `make test` # Простой тест интерпретатора Ruby.
11. `cd ../oniguruma` # Укажите путь к библиотеке.
12. `make rtest`

Или:

```
make rtest RUBYDIR=ruby-install-dir
```

Если же вы работаете на платформе Win32, скажем в Windows XP, то потребуются Visual C++ и исполняемый файл `patch.exe`. Выполните следующие действия:

1. Распакуйте архив любой имеющейся у вас программой.
2. `copy win32\Makefile Makefile`
2. Одно из следующих:
 - `nmake 16 RUBYDIR=ruby-source-dir` # для Ruby 1.6.8
 - `nmake 18 RUBYDIR=ruby-source-dir` # для Ruby 1.8.0/1.8.1
4. Следуйте инструкции в файле `ruby-source-dir\win32\README.win32`.

При возникновении ошибок обратитесь в список рассылки или конференцию.

3.13.3. Некоторые новые возможности Oniguruma

Oniguruma добавляет много новых возможностей к механизму работы с регулярными выражениями в Ruby. Из самых простых отметим дополнительную управляющую последовательность для указания класса символов. Если `\d` и `\D` соответствуют десятичным цифрам и не цифрам, то `\h` и `\H` являются аналогами для шестнадцатеричных цифр:

```
"abc" =~ /\h+/    # 0
"DEF" =~ /\h+/    # 0
"abc" =~ /\H+/    # nil
```

Добавилось возможностей у классов символов в квадратных скобках. Для организации вложенных классов можно применять оператор `&&`. Вот как можно записать регулярное выражение, соответствующее любой букве, кроме гласных а, е, i, o, u:

```
reg1 = /[a-z&&[^aeiou]]/    # Задаёт пересечение.
```

А следующее выражение соответствует всему алфавиту, кроме букв от m до p:

```
reg2 = /[a-z&&[^m-p]]/
```

Поскольку такие выражения выглядят не очень понятно, рекомендую пользоваться этим средством осмотрительно.

Другие возможности Oniguruma, например оглядывание назад и именованные соответствия, будут рассмотрены ниже. Все связанное с интернационализацией отложим до главы 4.

3.13.4. Позитивное и негативное оглядывание назад

Если заглядывания вперед вам недостаточно, то Oniguruma предлагает еще и *оглядывание назад*, позволяющее определить, предшествует ли текущему положению заданный образец.

Как и многое другое в регулярных выражениях, эту возможность довольно трудно понять и обосновать. Спасибо Эндрю Джексону за следующий пример.

Предположим, что вам нужно проанализировать некоторую генетическую последовательность (молекула ДНК состоит из четырех основных белков, которые обозначаются А, С, G и Т.) Допустим, что мы ищем все неперекрывающиеся цепочки нуклеотидов (длины 4), следующие за Т. Нельзя просто попытаться найти Т и взять следующие четыре символа, поскольку Т может быть последним символом в предыдущем соответствии.

```
gene = 'GATTACAAACTGCCTGACATACGAA'
seqs = gene.scan(/T(\w{4})/)
# seqs равно: [{"TACA"}, {"GCCT"}, {"ACGA"}]
```

Но в этом коде мы пропустили цепочку GACA, которая следует за GCCT. Позитивное оглядывание назад позволит найти все нужные цепочки:

```
gene = 'GATTACAAACTGCCTGACATACGAA'
seqs = gene.scan(/(?<=T)(\w{4})/)
# seqs равно: [{"TACA"}, {"GCCT"}, {"GACA"}, {"ACGA"}]
```


Следующий пример – небольшая модификация примера, предложенного К. Косако (K. Kosako). Предположим, что есть текст в формате XML (или HTML), и мы хотим перевести в верхний регистр весь текст вне тегов (то есть cdata). Вот как можно сделать это с помощью оглядывания назад:

```
text = <<-EOF
<body> <h1>This is a heading</h1>
<p> This is a paragraph with some
<i>italics</i> and some <b>boldface</b>
in it...</p>
</body>
EOF

pattern = /(?:^|      # Начало или...
              (?<=>)  # текст после '>'
            )
          ([^<]*)    # И все символы, кроме '<' (запомнены).
/x

puts text.gsub(pattern) {|s| s.upcase }

# Вывод:
# <body> <h1>THIS IS A HEADING</h1>
# <p>THIS IS A PARAGRAPH WITH SOME
# <i>ITALICS</i> AND SOME <b>BOLDFACE</b>
# IN IT...</p>
# </body>
```

3.13.5. Еще о кванторах

Мы уже встречались с атомарными подвыражениями в «классической» библиотеке регулярных выражений в Ruby. Они выделяются с помощью нотации (?:...) и являются «собственническими» в том смысле, что жадные и не допускают возврата внутрь подвыражения.

Oniguruma предлагает еще один способ выразить собственническую природу – с помощью квантора +. Он отличается от метасимвола + в смысле «один или более» и даже может использоваться с ним совместно. (На самом деле это «вторичный» квантор, как и ?, который можно употреблять в таких контекстах, как ??, +? и *?.)

Применение + к повторяющемуся образцу эквивалентно заключению его в скобки как независимого подвыражения, например:

```
r1 = /x*/      # То же, что /(?:x*)/
r2 = /x+*/     # То же, что /(?:x+)/
r3 = /x?+*/    # То же, что /(?:x?)/
```

По техническим причинам Ruby не считает конструкцию {n,m}+ собственнической.

Понятно, что новый квантор – не более чем удобное обозначение, никакой новой функциональности он не несет.

3.13.6. Именованные соответствия

Специальной формой подвыражения является именованное выражение, которое позволяет присвоить образцу имя (а не просто порядковый номер).

Синтаксически это выглядит так: `(?<name>expr)`, где `name` – имя, начинающееся с буквы (как идентификаторы в Ruby). Обратите внимание на сходство этой конструкции с неименованным атомарным подвыражением.

Для чего может понадобиться именованное выражение? Например, для того, чтобы сослаться на него внутри обратной ссылки. Ниже приведен пример простого регулярного выражения для сопоставления с повторяющимся словом (см. также раздел 3.14.6):

```
re1 = /\s+(\w+)\s+\1\s+/
str = "Now is the the time for all..."
re1.match(str).to_a      # ["the the", "the"]
```

Здесь мы запомнили слово, а затем сослались на него по номеру `\1`. Примерно так же можно пользоваться ссылками на именованные выражения. При первом обнаружении подвыражения ему присваивается имя, а в обратной ссылке употребляется символ `\k`, за которым следует это имя (всегда в угловых скобках):

```
re2 = /\s+(?<anyword>\w+)\s+\k<anyword>\s+/
```

Второй вариант длиннее, зато понятнее. (Имейте в виду, что в одном и том же регулярном выражении нельзя использовать и именованные, и нумерованные обратные ссылки.) Если нравится, пользуйтесь!

В Ruby уже давно можно включать обратные ссылки в строки, передаваемые методам `sub` и `gsub`. Раньше с этой целью допускалось лишь использование нумерованных ссылок, но в самых последних версиях именованные тоже разрешены:

```
str = "I breathe when I sleep"

# Нумерованные соответствия...
r1 = /I (\w+) when I (\w+)/
s1 = str.sub(r1, 'I \2 when I \1')

# Именованные соответствия...
r1 = /I (?<verb1>\w+) when I (?<verb2>\w+)/
s2 = str.sub(r2, 'I \k<verb2> when I \k<verb1>')

puts s1      # I sleep when I breathe
puts s2      # I sleep when I breathe
```

Еще одно возможное применение именованных выражений – повторное употребление выражения. В таком случае перед именем ставится символ `\g` (а не `\k`). Определим, например, образец `spaces` так, чтобы можно было использовать его многократно. Тогда последнее выражение примет вид:

```
re3 = /( (?<spaces>\s+) (?<anyword>\w+) \g<spaces> \k<anyword> \g<spaces> /
```

Обратите внимание, что этот образец многократно употребляется с помощью маркера `\g`. Особенно удобна такая возможность в рекурсивных регулярных выражениях, но это тема следующего раздела.

Нотацией `<1>` можно пользоваться и тогда, когда именованных подвыражений нет. Тогда запомненное ранее подвыражение вызывается по номеру, а не по имени.

И последнее замечание об именованных соответствиях. В самых последних версиях Ruby имя (в виде строки или символа) может передаваться методу `MatchData` в качестве индекса, например:

```
str = "My hovercraft is full of eels"
reg = /My (?<noun>\w+) is (?<predicate>.*)/
m = reg.match(str)
puts m[:noun]           # hovercraft
puts m["predicate"]     # full of eels
puts m[1]               # то же, что m[:noun] или m["noun"]
```

Как видите, обычные индексы тоже не запрещены. Обсуждается возможность добавить в объект `MatchData` и синглетные методы.

```
puts m.noun
puts m.predicate
```

Но во время работы над книгой это еще не было реализовано.

3.13.7. Рекурсия в регулярных выражениях

Возможность повторно обращаться к подвыражению позволяет создавать *рекурсивные регулярные выражения*. Например, данный код находит любое вложенное выражение с правильно расставленными скобками (спасибо Эндрю Джексону):

```
str = "a * ((b-c)/(d-e) - f) * g"

reg = /(?                # Начало именованного выражения.
  \(                    # Открывающая круглая скобка.
    (?                # Незапоминаемая группа.
      (?>              # Сопоставление с собственническим выражением:
        \{ \}          # экранированная скобка
        |              # ЛИБО
        [^()]          # вообще не скобка.
      )                # Конец собственнического выражения.
    )                  # ЛИБО
    \g                 # Вложенная группа в скобках (рекурсивный вызов).
  ) *                  # Незапоминаемая группа повторяется нуль или
                        # более раз.
  \)                  # Закрывающая круглая скобка.
)                    # Конец именованного выражения.
/x

m = reg.match(str).to_a # ["((b-c)/(d-e) - f)", "((b-c)/(d-e) - f)"]
```

Отметим, что левосторонняя рекурсия запрещена. Следующий пример допустим:

```
str = "bbbacc"
re1 = /(?!<foo>a\b\g<foo>c)/
re1.match(str).to_a # ["bbbacc", "bbbacc"]
```

А такой – нет:

```
re2 = /(?!<foo>a\b\g<foo>c)/ # Синтаксическая ошибка!
```

Ошибка объясняется наличием рекурсивного обращения в начале каждой альтернативы. Немного подумав, вы поймете, что это приведет к бесконечному возврату.

3.14. Примеры регулярных выражений

В этом разделе мы приведем краткий перечень регулярных выражений, которые могут оказаться полезны на практике или просто послужат учебными примерами. Для простоты примеров ни одно выражение не зависит от наличия Oniguruma.

3.14.1. Сопоставление с IP-адресом

Пусть мы хотим понять, содержит ли строка допустимый IPv4-адрес. Стандартно он записывается в точечно-десятичной нотации, то есть в виде четырех десятичных чисел, разделенных точками, причем каждое число должно находиться в диапазоне от 0 до 255.

Приведенный ниже образец решает эту задачу (за немногими исключениями типа «127.1»). Для удобства восприятия мы разобьем его на части. Отметим, что символ `\d` дважды экранирован, чтобы косая черта не передавалась из строки в регулярное выражение (чуть ниже мы решим и эту проблему).

```
num = "(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5])"
pat = "^({num}\\.){{3}}#{num}$"
ip_pat = Regexp.new(pat)

ip1 = "9.53.97.102"

if ip1 =~ ip_pat          # Печатается: "да"
  puts "да"
else
  puts "нет"
end
```

Надо признать, что в определении переменной `num` слишком много символов обратной косой черты. Определим ее в виде регулярного выражения, а не строки:

```
num = /(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5])/
```

Когда одно регулярное выражение интерполируется в другое, вызывается метод `to_s`, который сохраняет всю информацию из исходного регулярного выражения.

```
num.to_s      # "(?-mix:(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5]))"
```

Иногда для встраивания удобно использовать регулярное выражение, а не строку. Хорошее эвристическое правило: интерполируйте регулярные выражения, если нет веских причин интерполировать строки.

IPv6-адреса пока не очень широко распространены, но для полноты рассмотрим и их. Они записываются в виде восьми шестнадцатеричных чисел, разделенных двоеточиями, с подавлением начальных нулей.

```
num = /[0-9A-Fa-f]{0,4}/
pat = /^(#{num}:){7}#{num}$/
ipv6_pat = Regexp.new(pat)
```

```
v6ip = "abcd::1324:ea54::dead::beef"

if v6ip =~ ipv6_pat          # Печатается: "да"
  puts "да"
else
  puts "нет"
end
```

3.14.2. Сопоставление с парой «ключ-значение»

Иногда приходится работать со строками вида «ключ=значение» (например, при разборе конфигурационного файла приложения).

Следующий код извлекает ключ и значение. Предполагается, что ключ состоит из одного слова, значение продолжается до конца строки, а знак равенства может быть окружен пробелами:

```
pat = /(\w+)\s*=\s*(.*)$/
str = "color = blue"

matches = pat.match(str)

puts matches[1]      # "color"
puts matches[2]      # "blue"
```

3.14.3. Сопоставление с числами, записанными римскими цифрами

Следующее довольно сложное регулярное выражение сопоставляется с любым правильно записанным римскими цифрами числом (до 3999 включительно). Как и раньше, для удобства восприятия образец разбит на части:

```
rom1 = /m{0,3}/i
rom2 = /(d?c{0,3}|c[dm])/i
rom3 = /(l?x{0,3}|x[lc])/i
rom4 = /(v?i{0,3}|i[vx])/i
roman = /^#{rom1}#{rom2}#{rom3}#{rom4}$/

year1985 = "MCMLXXXV"

if year1985 =~ roman          # Печатается: "да"
  puts "да"
else
  puts "нет"
end
```

Возможно, у вас появилось искушение поставить в конец всего выражения модификатор *i*, чтобы сопоставлялись и строчные буквы:

```
# Это не работает!

rom1 = /m{0,3}/
rom2 = /(d?c{0,3}|c[dm])/
rom3 = /(l?x{0,3}|x[lc])/
rom4 = /(v?i{0,3}|i[vx])/
roman = /^#{rom1}#{rom2}#{rom3}#{rom4}$/i
```

```
rom1.to_s # "(?-mix:m{0,3})"
```

3.14.4. Сопоставление с числовыми константами

```
int_pat = /^[+-]?[1-9][\d_]*$/
```

```
hex_pat = /^[+-]?0x[\da-f_]+$/i
```

```
oct_pat = /^[+-]?0[0-7_]+$
```

```
bin_pat = /^[+-]?0b[01_]+$ /i
```

```
float_pat = /^(\d[\d_]*)*\.[\d_]*$/
```

Образец для чисел, записанных в научной нотации, основан на предыдущем:

```
sci_pat = /^(\\d[\\d_]*)?\\.([\\d_]*(e[+-])?)?(_*\\d[\\d_]*)$/i
```

3.14.5. Сопоставление с датой и временем

Пусть надо выделить дату и время, записанные в формате mm/dd/yy hh:mm:ss. Вот первая попытка: `datetime = /(\d\d)\ /(\d\d)\ /(\d\d) (\d\d): (\d\d): (\d\d)/`.

Но такой образец распознает некоторые некорректные даты и отвергает правильные. Следующий вариант более избирателен. Обратите внимание, как мы строим его путем интерполяции мелких регулярных выражений в более крупное:

```
mo = /(0?[1-9]|1[0-2])/          # От 01 до 09 или от 1 до 9 или 10-12.
dd = /([0-2]?[1-9]|1[3-9]|01)/    # 1-9 или 01-09 или 11-19 и т.д.
yy = /(\\d\\d)/                    # 00-99
hh = /([01]?[1-9]|1[2][0-4])/      # 1-9 или 00-09 или...
mi = /([0-5]\\d)/                  # 00-59, обе цифры должны присутствовать.
ss = /([0-6]\\d)?/                 # Разрешены еще и доли секунды ;-)
```

```
date = /#{mo}\/#{dd}\/#{yy})/  
time = /#{hh}:#{mi}:#{ss})/
```

```
datetime = /({date} {time})/
```

Вот как можно вызвать это регулярное выражение из метода `String#scan`, чтобы получить массив соответствий:

```
str="Recorded on 11/18/07 20:31:00"
str.scan(datetime)
# [{"11/18/07 20:31:00", "11/18/07", "11", "18", "00",
#   "20:31:00", "20", "31", ":00"}]
```

Разумеется, все это можно было сделать с помощью одного большого регулярного выражения:

```
datetime = %r((
  (0?[1-9]|1[0-2])/          # mo: от 01 до 09 или от 1 до 9 или 10-12.
  ([0-2]?[1-9]|1[1-3][01])/  # dd: 1-9 или 01-09 или 11-19 и т.д.
  (\d\d) [ ]                 # yy: 00-99
  ([01]?[1-9]|1[12][0-4]):   # hh: 1-9 или 00-09 или...
  ([0-5]\d):                 # mm: 00-59, обе цифры должны присутствовать.
  ([0-6]\d)?                 # ss: разрешены еще и доли секунды ;-))
))x
```

Обратите внимание на конструкцию `%r()`, позволяющую не экранировать символы обратной косой черты.

3.14.6. Обнаружение повторяющихся слов в тексте

В этом разделе мы реализуем детектор повторяющихся слов. Повторение одного и того же слова два раза подряд – типичная опечатка. Следующий код распознает такие ситуации:

```
double_re = /\b(['A-Z]+) +\1\b/i

str="There's there's the the pattern."
str.scan(double_re) # [{"There's"}, {"the"}]
```

Обратите внимание на модификатор `i` в конце выражения, он позволяет проводить сопоставление без учета регистра. Каждой группе соответствует массив, поэтому в результате получается массив массивов.

3.14.7. Поиск слов, целиком набранных прописными буквами

Мы упростили пример, предположив, что в тексте нет чисел, подчеркивов и т.д.

```
allcaps = /\b[A-Z]+\b/

string = "This is ALL CAPS"
string[allcaps] # "ALL"
Suppose you want to extract every word in all-caps:
string.scan(allcaps) # ["ALL", "CAPS"]
```

При желании можно было бы обобщить эту идею на идентификаторы Ruby и аналогичные вещи.

3.14.8. Сопоставление с номером версии

Принято присваивать библиотекам и приложениям номера версий, состоящие из трех чисел, разделенных точками. Следующее регулярное выражение сопоставляется с подобной строкой, выделяя попутно имя пакета и отдельные части номера версии:

```
package = "mylib-1.8.12"
matches = package.match(/(.*)-(\d+)\.(\d+)\.(\d+)/)
name, major, minor, tiny = matches[1..-1]
```

3.14.9. Еще несколько образцов

Завершим наш список несколькими выражениями из категории «разное». Как обычно, почти все эти задачи можно решить несколькими способами.

Пусть нужно распознать двузначный почтовый код американского штата. Проще всего, конечно, взять выражение `/[A-Z]{2}/`. Но оно сопоставляется с такими строками, как `xx` или `zz`, которые допустимы, но бессмысленны. Следующий образец распознает все стандартные аббревиатуры, общим числом 51 (50 штатов и DC — округ Колумбия):

```
state = /^A[LKZR] | C[AOT] | D[EC] | FL | GA | HI | I[DLNA] |
        K[SY] | LA | M[EDAINSOT] | N[EVHJMYCD] | O[HKR] |
        PA | RI | S[CD] | T[NX] | UT | V[TA] | W[AVIY]$/x
```

Для ясности я воспользовался обобщенным регулярным выражением (модификатор `x`). Пробелы и символы новой строки в нем игнорируются.

Продолжая эту тему, приведем регулярное выражение для распознавания почтового индекса США (он может состоять из пяти или девяти цифр):

```
zip = /^d{5}(-d{4})?$/
```

Якоря (в этом и других выражениях) призваны лишь гарантировать, что ни до, ни после сопоставленной строки никаких лишних символов нет. Отметим, что это выражение не отбрасывает несуществующие индексы, поэтому оно не так полезно, как предыдущее.

Следующее регулярное выражение распознает номер телефона в формате NANP (североамериканский план нумерации). Есть три способа записи такого номера:

```
phone = /^(\\(\\d{3}\\) |\\d{3}-)\\d{3}-\\d{4}|\\d{3}\\.|\\d{3}\\.|\\d{4})$/
"(512) 555-1234" =~ phone      # true
"512.555.1234"   =~ phone      # true
"512-555-1234"   =~ phone      # true
"(512)-555-1234" =~ phone      # false
"512-555.1234"   =~ phone      # false
```

Распознавание денежной суммы в долларах также не составит труда:

```
dollar = /^$d+(\\.d)d)?$/
```

Ясно, что слева от десятичной точки должна быть хотя бы одна цифра, а после знака доллара не должно быть пробелов. Отметим еще, что если вы хотите только выделить, а не проконтролировать суммы в долларах, то якоря следовало бы удалить, а центры сделать необязательными.

3.15. Заключение

На этом мы заканчиваем обсуждение регулярных выражений в Ruby. Ознакомившись со строками и регулярными выражениями, мы можем перейти к теме интернационализации, где пригодится рассмотренный выше материал.

Глава 4. Интернационализация в Ruby

.....
*Посему дано ему имя: Вавилон, ибо там смешал Господь
язык всей земли, и оттуда рассеял их Господь по всей земле.*
Бытие, 11:9

Мы уже говорили, что тип символа, наверное, самый важный из всех. Но что такое символные данные? Какие символы? Из какого алфавита? Какой язык? Какие культурные особенности?

В прошлом в вычислительной технике и информатике применялся исключительно английский язык. Традиция восходит, вероятно, еще к Чарльзу Беббиджу. Это не так уж и плохо, ведь надо же с чего-то начинать, а алфавит из 26 букв без диакритических знаков — неплохое начало.

Но теперь компьютеры распространились повсеместно. Наверное, в каждой стране есть хотя бы несколько компьютеров и тот или иной вид доступа в сеть. Естественно, любой человек предпочитает читать Web-страницы, электронную почту и прочие данные на своем родном языке, а не только на английском.

Человеческие языки поразительно разнообразны. Некоторые являются почти фонетическими, к другим это определение применимо с большой натяжкой. В одних есть настоящий алфавит, другие же предлагают набор из нескольких тысяч символов, ведущих происхождение от пиктограмм. В некоторых языках не один алфавит, а несколько. На каких-то языках пишут сверху вниз, на других — справа налево. Некоторые алфавиты очень просты, в других ряд букв украшен приводящими в трепет точечками, черточками, кружочками, галочками и штрихами... Есть языки, где при определенных условиях соседние буквы комбинируются; иногда это обязательно, а иногда и нет. В каких-то языках предусмотрено различение строчных и прописных букв, но большинство таких различий не уславливает.

За 25 лет мы прошли длинный путь. Мы худо-бедно научились приводить в порядок хаотическое нагромождение символов и языков.

Если вам часто приходится иметь дело с программами, спроектированными для работы в различной языковой среде, то вы знаете, что такое интернационализация. Это способность программы поддерживать более одного естественного языка.

С интернационализацией тесно связаны мультиязычность и локализация. Почему-то принято сокращать эти слова, удаляя средние буквы и подставляя вместо них число, равное количеству удаленных букв:

```
def shorten(str)
  (str[0..0] + str[1..-2].length.to_s + str[-1..-1]).upcase
end
```

```
shorten("internationalization")    # I18N
shorten("multilingualization")      # M17N
shorten("localization")              # L10N
```

Термины I18N и M17N – практически синонимы; еще говорят «глобализация», но это слово имеет и другой смысл. Термин L10N более широкий: он подразумевает полную поддержку местной культуры и соглашений (например, символов обозначения валюты, способов форматирования даты и времени, использования точек или запятой для отделения целой и дробной частей в десятичном числе и многое другое).

Начнем с терминологии, поскольку в этой области широко используется жаргон. Заодно совершим небольшой экскурс в историю, так как текущее состояние дел становится понятным, лишь если рассмотреть его в контексте медленной эволюции. Уроки истории будут сведены к минимуму.

4.1. Исторические сведения и терминология

В «недобрые старые дни» становления компьютерных технологий, примерно совпадающие по времени с периодом использования перфокарт, существовало множество наборов символов. К счастью, с появлением кода ASCII в 1970-х годах эти дни миновали безвозвратно.

Аббревиатура ASCII означает *American Standard Code for Information Interchange* (Американский стандартный код обмена информацией). Это был большой шаг вперед, однако ключевое слово здесь «американский». Код проектировался даже без учета европейских языков, не говоря уже об азиатских.

Но в нем были и огрехи. Набор символов ASCII состоит из 128 символов (он 7-разрядный). Но как можно так расточительно относиться к дополнительному биту? Возникла естественная идея расширить набор ASCII, воспользовавшись кодами от 128 до 255 для других целей. Беда в том, что эта идея была реализована многократно и по-разному компанией IBM и другими. Не было общепринятого соглашения о том, какому символу соответствует, например, код 221.

Недостатки такого подхода очевидны. Даже если отправитель и получатель договорятся об используемом наборе символов, все равно они не смогут общаться на нескольких языках: для всех сразу не хватит символов. Если вы хотите писать по-немецки, но вставить в текст несколько цитат на греческом или иврите, то, скорее всего, ничего не получится. И эта схема не позволила даже приблизиться к решению проблем, связанных с азиатскими языками, например китайским, японским и корейским.

Было два основных способа решить эту задачу. Первый – использовать гораздо более обширный набор символов, например представляя каждый символ 16 битами (так называемые *широкие символы*). Второй – обратиться к многобайтовым кодировкам переменной длины. При такой схеме одни символы представляются

единственным байтом, другие – двумя, а третьи – тремя или даже большим числом. При этом, очевидно, возникает масса вопросов. В частности, любая строка должна однозначно декодироваться. Первый байт многобайтового символа мог бы принадлежать специальному классу, а потому мы сумели бы понять, что следует ожидать дополнительный байт; но как быть со вторым и последующими? Разрешено ли им перекрываться с набором однобайтовых символов? Могут ли определенные символы выступать в роли второго и третьего байта или это следует запретить? Сможем ли мы перейти в середину строки и при этом не запутаться? Сможем ли просматривать строку в обратном направлении? Для разных кодировок были приняты различные проектные решения.

В конечном счете родилась идея *кодировки Unicode*. Считайте, что это «всемирный набор символов». Увы, на практике все не так просто.

Возможно, вы слышали, что Unicode был (или остается) ограничен 65536 символами (именно столько различных комбинаций можно представить 16 битами). Распространенное заблуждение!.. При проектировании Unicode такие ограничения не закладывались. С самого начала было ясно, что во многих случаях это будет многобайтовая схема. Количество представимых с помощью Unicode символов практически безгранично, и это хорошо, так как 65000 никогда не хватит для всех языков мира.

Говоря об интернационализации, нужно прежде всего понимать, что интерпретация строки не является внутренне присущей самой строке. Это заблуждение проистекает из уже неактуального представления, будто существует лишь один способ хранения строки.

Подчеркну, это исключительно важное положение. Внутренне строка – всего лишь последовательность байтов. Представьте себе, что в памяти машины хранится один байт в кодировке ASCII. Если это буква, которую мы называем «прописная латинская А», то реально хранится число 65.

Почему мы считаем, что 65 – это А? Потому что так мы договорились использовать (интерпретировать) это значение. Если мы складываем его с другим числом, то оно используется (интерпретируется) как число. А если отправляем его на терминал по последовательной линии связи – значит, интерпретируем как ASCII-символ.

Если можно по-разному интерпретировать одиночный байт, то почему же нельзя так сделать для последовательности байтов? На самом деле, чтобы получилась осмысленная строка, предполагаемая схема интерпретации (или кодировка) должна быть известна заранее. Кодировка – это просто соответствие между двойными числами и символами. И снова не все так просто.

Поскольку Ruby появился в Японии, он прекрасно справляется с двумя различными японскими кодировками (и ASCII). Не буду тратить время на рассказ о поддержке японского языка; если вы японец, то в вашем распоряжении сколько угодно книг по Ruby на этом языке. А для всех остальных наиболее распространенной кодировкой является Unicode. О ней мы и будем говорить в этой главе.

Но перед тем как перейти к деталям, познакомимся с некоторыми терминами. Называть вещи полезными именами – одна из основ мудрости!

- *Байт* – это просто восемь битов (хотя когда-то даже это было неверно). По традиции многие считают, что байт соответствует одному символу. Ясно, что в контексте I18N это не так.
- *Кодовая позиция* – один элемент воображаемой таблицы, с помощью которой представляется набор символов. Хотя это и не совсем верно, можете считать, что кодовые позиции взаимно однозначно отображаются на символы. Точнее будет сказать, что иногда для уникального указания символа требуется несколько кодовых позиций.
- *Глиф (печатный знак)* – визуальное представление кодовой позиции. Хотя интуитивно это и не совсем очевидно, символ и его визуальное представление – разные вещи. (Я могу открыть текстовый редактор и набрать прописную А десятком разных шрифтов, но все это будет один и тот же символ А.)
- Понятие *графемы* близко к глифу, но о графемах мы говорим в контексте языка, а не программного обеспечения. Графема может быть комбинацией (простой или не очень) двух и более глифов. Так пользователь воспринимает символ в контексте своего родного языка. Разница настолько тонкая, что большинство программистов могут о ней никогда не задумываться.

Что же тогда такое символ? Даже в мире Unicode нет четкого понимания этого предмета, поскольку языки ведут себя по-разному, а программисты мыслят иначе, чем прочие люди. Будем говорить, что *символ* – это абстракция написания знака, который визуальным образом может быть представлен одним или несколькими способами.

Перейдем к конкретике. Сначала я хочу познакомить вас с нотацией. Традиционно кодовые позиции Unicode записываются как U+, а затем четыре или более шестнадцатеричных цифр в верхнем регистре. То, что мы называем латинской буквой А, можно представить в виде U+0041.

Теперь возьмем букву é (строчная е с акутом). Ее можно представить в Unicode двумя способами. Во-первых, это одна кодовая позиция U+00E9 (СТРОЧНАЯ ЛАТИНСКАЯ Е С АКУТОМ). С другой стороны, это сочетание двух кодовых позиций: строчная е + диакритический знак акут – U+0065 и U+0301. Иными словами, СТРОЧНАЯ ЛАТИНСКАЯ Е, за которой следует АКУТ.

Обе формы одинаково правильны. Более короткая называется монолитной (precomposed) формой. Однако имейте в виду, что не для каждого языка имеются монолитные варианты, поэтому не всегда можно свести подобный символ к одной кодовой позиции.

Я назвал Unicode кодировкой, но это не вполне верно. Unicode отображает символы на кодовые позиции; существуют разные способы отобразить кодовые позиции на двоичное представление. По существу, Unicode – это семейство кодировок.

Возьмем, к примеру, строку "Matz". Она состоит из четырех кодовых позиций Unicode:

"Matz" # U+004d U+0061 U+0074 U+007a

Естественнее всего сохранить их в виде простой последовательности байтов.

00 4d 00 61 00 74 00 7a

Такая кодировка называется UCS-2 (два байта) или UTF-16 (16 битов). Отметим, что эта кодировка имеет две разновидности: тупоконечную (big-endian) и остроконечную (little-endian) – в зависимости от того, старший или младший байт хранится первым.

Заметим, однако, что каждый второй байт в этой последовательности нулевой. Это не просто совпадение, английский язык редко выходит за пределы кодовой позиции U+00FF. Так разбрасываться памятью расточительно.

И это наблюдение подводит нас к идее кодировки UTF-8. В ней «традиционные» символы представлены одним байтом, а остальные – несколькими. Вот как записывается та же строка в кодировке UTF-8:

```
4d 61 74 7a
```

Мы всего лишь избавились от нулей. Однако более важен тот факт, что мы получили обычную кодировку ASCII. Так и задумано: «простой ASCII» можно считать собственным подмножеством UTF-8.

Отсюда, в частности, следует, что при интерпретации текста в кодировке UTF-8 как ASCII-текста он выглядит «как обычно» (особенно если это преимущественно англоязычный текст). Иногда вы видите, что браузер или другое приложение отображает английский текст правильно, но местами появляются «крокозябры». Это, скорее всего, означает, что программа сделала неверные предположения об используемой кодировке.

Итак, можно сказать, что UTF-8 экономит память. Конечно, я снова становлюсь на англоцентрическую точку зрения (по крайней мере, ASCII-центрическую). Если текст в основном состоит из ASCII-символов, то да, память экономится, но для других языков, например греческого или русского, размер строк даже увеличится.

Еще одно очевидное достоинство UTF-8 – «обратная совместимость» с кодировкой ASCII, которая, по-видимому, все еще остается самой распространенной однобайтовой кодировкой в мире. Наконец, у UTF-8 есть некоторые особенности, делающие ее удобной для программистов.

Во-первых, байты, входящие в состав многобайтовых символов, тщательно подобраны. Нулевой байт (ASCII 0) никогда не встречается в качестве n -ого байта в последовательности (где $n > 1$); то же самое справедливо для таких распространенных символов, как косая черта (обычно используется для разделения компонентов пути к файлу). На самом деле никакой байт из диапазона 0x00-0x7F не может быть частью никакого другого символа.

Второй байт многобайтового символа однозначно определяет, сколько байтов за ним следует. Этот второй байт всегда выбирается из диапазона от 0xC0 до 0xFF, а следующие за ним – из диапазона от 0x80 до 0xBF. Таким образом, схема кодирования свободна от состояния и позволяет восстанавливать пропущенные или искаженные байты.

UTF-8 – одна из самых распространенных и гибких кодировок в мире. Она применяется с начала 1990-х годов и является кодировкой по умолчанию для XML-документов. В этой главе мы будем иметь дело главным образом именно с UTF-8.

```
eacute = ""
eacute << 0303 << 0251 # U+00E9
sword = eacute + "p" + eacute + "e"
p eacute # "\303\251"

p sword # "\303\251p\303\251e"

$KCODE = "u"
```

```
p eacute          # "é"
p sword           # "épée"
```

Регулярные выражения в режиме UTF-8 тоже становятся несколько «умнее».

```
$KCODE = "n"
letters = sword.scan(/./)
# [{"\303"}, {"\251"}, {"p"}, {"\303"}, {"\251"}, {"e"}]
puts letters.size # 6
```

```
$KCODE = "u"
```

```
letters = sword.scan(/./)
# [{"é"}, {"p"}, {"é"}, {"e"}]
puts letters.size # 4
```

Библиотека `jcode` предоставляет также несколько полезных методов, например `jlength` и `each_char`. Рекомендую включать эту библиотеку с помощью директивы `require` всякий раз, как вы работаете с кодировкой UTF-8.

В следующем разделе мы снова рассмотрим некоторые типичные операции со строками и регулярными выражениями. Заодно поближе познакомимся с `jcode`.

4.2.2. Возвращаясь к строкам и регулярным выражениям

При работе с UTF-8 некоторые операции ничем не отличаются. Например, конкатенация строк выполняется так же, как и раньше:

```
"ép" + "ée"      # "épée"
"ép" << "ée"     # "épée"
```

Поскольку UTF-8 не имеет состояния, то для проверки вхождения подстроки тоже ничего специально делать не нужно:

```
"épée".include?("é") # true
```

Однако при написании интернациональной программы некоторые типичные допущения все же придется переосмыслить. Ясно, что символ больше не эквивалентен байту. При подсчете символов или байтов надо думать о том, что именно мы хотим сосчитать и для чего. То же относится к числу итераций.

По общепринятому соглашению, кодовую позицию часто представляют себе как «программистский символ». Это еще одна полуправда, но иногда она оказывается полезной.

Метод `jlength` возвращает число кодовых позиций в строке, а не байтов. Если нужно получить число байтов, пользуйтесь методом `length`.

```
$KCODE = "u"
require 'jcode'
```

```
sword = "épée"
sword.jlength # 4
sword.length  # 6
```

Такие методы, как `upcase` и `capitalize`, обычно неправильно работают со специальными символами. Это ограничение текущей версии Ruby. (Не стоит считать

это ошибкой, поскольку получить представление слова с первой прописной буквой довольно трудно; такая задача просто не решается в схеме интернационализации Ruby. Считайте, что это нереализованное поведение.)

```
$KCODE = "u"
sword.upcase      # "ÉPÉE"
sword.capitalize  # "épée"
```

Если вы не пользуетесь монолитной формой, то в некоторых случаях метод может сработать, поскольку латинские буквы отделены от диакритических знаков. Но в общем случае работать не будет – в частности, для турецкого, немецкого, голландского и любого другого языка с нестандартными правилами преобразования регистра.

Возможно, вы думаете, что неакцентированные символы в некотором смысле эквивалентны своим акцентированным вариантам. Это почти всегда не так. Здесь мы имеем дело с разными символами. Убедимся в этом на примере метода count:

```
$KCODE = "u"
sword.count("e")  # 1 (не 3)
```

Но для составных (не монолитных) символов верно прямо противоположное. В этом случае латинская буква распознается.

Метод count возвращает сбивающий с толку результат, когда ему передается многобайтовый символ. Метод jcount ведет себя в этом случае правильно:

```
$KCODE = "u"
sword.count("eé") # 5 (не 3)
sword.jcount("eé") # 3
```

Существует вспомогательный метод mbchar?, который определяет, есть ли в строке многобайтовые символы.

```
$KCODE = "u"
sword.mbchar? # 0 (смещение первого многобайтового символа)
"foo".mbchar? # nil
```

В библиотеке jcode переопределены также методы chop, delete, squeeze, succ, tr и tr_s. Применяя их в режиме UTF-8, помните, что вы работаете с версиями, «знающими о многобайтовости». При попытке манипулировать многобайтовыми строками без библиотеки jcode вы можете получить странные или ошибочные результаты.

Можно побайтно просматривать строку, как обычно, с помощью итератора each_byte. А можно просматривать посимвольно с помощью итератора each_char. Второй способ имеет дело с односимвольными строками, первый (в текущей версии Ruby) – с однобайтными целыми.

Разумеется, мы в очередной раз приравниваем кодовую позицию к символу. Несмотря на название, метод each_char на самом деле перебирает кодовые позиции, а не символы.

```
$KCODE = "u"
sword.each_byte {|x| puts x } # Шесть строк с целыми числами.
sword.each_char {|x| puts x } # Четыре строки со строками.
```


Если вы запутались, не переживайте. Все мы через это проходили. Я попытаюсь свести все вышесказанное в таблицу 4.1.

Таблица 4.1. Составные и монолитные формы

Монолитная форма "é"				
Название символа	Глиф	Кодовая позиция	Байты UTF-8	Примечания
Строчная латинская е с акутом	é	U+00E9	0xC3 0xA9	Один символ, одна кодовая позиция, один байт
Составная форма "é"				
Название символа	Глиф	Кодовая позиция	Байты UTF-8	Примечания
Строчная латинская е	e	U+0065	0x65	Один символ, две кодовых позиции (два «программистских символа»), три байта UTF-8
Модифицирующий акут	'	U+0301	0xCC 0x81	

Что еще надо учитывать при работе с интернациональными строками? Квадратные скобки по-прежнему относятся к байтам, а не к символам. Но при желании это можно изменить. Ниже приведена одна из возможных реализаций (не особенно эффективная, зато понятная):

```
class String

  def [](index)
    self.scan(/./)[index]
  end

  def []=(index,value)
    arr = self.scan(/./)
    arr[index] = value
    self.replace(arr.join)
    value
  end

end
```

Конечно, здесь не реализована значительная часть функциональности настоящего метода `[]`, который понимает диапазоны, регулярные выражения и т.д. Если вам все это нужно, придется запрограммировать самостоятельно.

У метода `unpack` есть параметры, помогающие манипулировать Unicode-строками. Указав в форматной строке параметр `U*`, мы можем преобразовать строку в кодировке UTF-8 в массив кодовых позиций (`U` без звездочки преобразует только первую кодовую позицию):

```
codepoints = sword.unpack('U*') # [233, 112, 233, 101]
```

Вот несколько более полезный пример, в котором все кодовые позиции в строке, отличные от ASCII (то есть начиная с `U+0080`), преобразуются к виду `U+xxxx`, который мы обсуждали выше:

У метода `String#unpack` есть «близкий родственник» `Array#pack`, выполняющий обратную операцию:

Мы можем воспользоваться им, чтобы вставить Unicode-символы, которые трудно ввести с клавиатуры:

Регулярным выражениям тоже известно о многобайтовых символах, особенно если вы пользуетесь библиотекой Oniguruma (мы рассматривали ее в главе 3). Например, образец `/./` сопоставляется с одним многобайтовым символом.

Модификатор `u` извещает регулярное выражение о том, что мы работаем с кодировкой UTF-8. Если `$KCODE` равно `"u"`, то модификатор можно не задавать, однако это и не повредит. (К тому же такая избыточность может быть полезна, если код является частью большой программы, а какое значение переменной `$KCODE` в ней установлено, вам неизвестно.)

Даже без Oniguruma регулярные выражения распознают, относится ли данный **многобайтовый** символ к категории тех, что могут входить в состав слова:

При наличии Onigiguma последовательности, начинающиеся с символа обратной косой черты (\w, \s и т.п.) распознают и более широкие диапазоны кодовых точек: слова, пропуски и т.д.

Регулярные выражения позволяют безопасно выполнять простые манипуляции со строками. Мы и так можем без труда усекаать строки. Следующий код возвращает не более 20 символов из строки `ascii_string`:

```
ascii_string[0,20]
```

Однако, поскольку кодовая позиция Unicode может занимать более одного байта, такую технику нельзя безопасно применять к строке в кодировке UTF-8. Есть риск, что в конце строки окажется недопустимая последовательность байтов. Кроме того, это не слишком полезно, так как мы не можем заранее сказать, сколько в результате получится кодовых позиций. На помощь приходят регулярные выражения:

```
def truncate(str, max_length)
  str[/.{0,#{max_length}}/m]
end
```

4.2.3. Распознавание кодировки

Распознать, в какой кодировке записана данная строка, довольно сложно. Многобайтовые кодировки обладают отличительными признаками, по которым их можно опознать, но с однобайтовыми – а именно они применяются в западных языках – дело обстоит куда хуже. Для решения можно применить статистические методы, но эта тема выходит за рамки данной книги (к тому же результат в общем случае получается не слишком надежным).

К счастью, обычно перед нами стоит более простая задача – выяснить, записана ли строка в кодировке UTF-8. На этот вопрос можно дать достаточно надежный ответ. Приведем один способ (основанный на том, что метод `unpack` возбуждает исключение, если ему передана некорректная строка):

```
class String
  def utf8?
    unpack('U*') rescue return false
    true
  end
end
```

4.2.4. Нормализация Unicode-строк

До сих пор мы пользовались монолитными символами, в которых базовый символ и диакритический знак объединены в одну кодовую позицию. Но, вообще говоря, в Unicode символы и диакритические знаки представлены отдельно. Вместо того чтобы хранить букву *é* в кодовой позиции **СТРОЧНАЯ ЛАТИНСКАЯ БУКВА Е С АКУТОМ**, можно было бы представить ее в составной форме как **СТРОЧНУЮ ЛАТИНСКУЮ БУКВУ Е** и **МОДИФИЦИРУЮЩИЙ АКУТ**.

Для чего это может понадобиться? Для обеспечения дополнительной гибкости и возможности применять диакритические знаки к любому символу, а не ограничивать себя комбинациями, которые предусмотрел проектировщик кодировки. На самом деле в шрифты включены глифы для наиболее распространенных комбинаций символа и диакритического знака, но отображение символа и его кодирование – вещи разные.

При проектировании Unicode приходилось учитывать такие вещи, как эффективность и совместимость с существующими национальными кодировками. Иногда это приводит к избыточности; например, в Unicode имеются кодовые позиции как для составных форм, так и для многих уже применяющихся монолитных форм.

Рассмотрим, к примеру, немецкое слово «öffnen» (открывать). Даже если забыть о регистре, его можно закодировать четырьмя способами:

1. **о** + **МОДИФИЦИРУЮЩАЯ ТРЕМА** ($U+0308$) + $f + f + n + e + n$
2. **СТРОЧНАЯ ЛАТИНСКАЯ БУКВА О С ТРЕМОЙ** ($U+00F6$) + $f + f + n + e + n$
3. **о** + **МОДИФИЦИРУЮЩАЯ ТРЕМА** + **ЛИГАТУРА ДВОЙНОЕ F** ($U+FB00$) + $n + e + n$

4. СТРОЧНАЯ ЛАТИНСКАЯ БУКВА О С ТРЕМОЙ + ЛИГАТУРА ДВОЙНОЕ F + n + e + n

Трема – это две точки над буквой (в немецком языке называется «умляут»).

Нормализацией называется процедура приведения разных представлений символа к стандартной форме. Можно быть уверенным, что после нормализации данный символ закодирован вполне определенным образом. Каким именно, зависит от того, чего мы хотим достичь. В приложении 15 к стандарту Unicode перечислены четыре формы нормализации:

1. Форма D (каноническая декомпозиция).
2. Форма C (каноническая декомпозиция с последующей канонической композицией).
3. Форма KD (совместимая декомпозиция).
4. Форма KC (совместимая декомпозиция с последующей канонической композицией).

Иногда можно встретить аббревиатуры NKFC (Normalization Form KC) и т.д.

Точные правила, сформулированные в стандарте, довольно сложны; в них проведено различие между «канонической эквивалентностью» и «совместимой эквивалентностью». (Корейский и японский языки требуют особого рассмотрения, но мы не станем тратить на это время.) В таблице 4.2 показано, как форма нормализации влияет на приведенные выше строки.

Таблица 4.2. Нормализованные формы в Unicode

Исходная	NFD	NFC	NFKD	NFKC
o~+f+f+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n
ö+f+f+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n
o~+ff+n+e+n	o~+ff+n+e+n	ö+ff+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n
ö+ff+n+e+n	o~+ff+n+e+n	ö+ff+n+e+n	o~+f+f+n+e+n	ö+f+f+n+e+n

Формы C и D обратимы, KC и KD – нет. С другой стороны, потеря некоторых данных в формах KC и KD – свидетельство того, что все четыре строки двоично эквивалентны. Какая форма лучше всего подходит, зависит от приложения. Мы еще вернемся к этой теме в следующем разделе.

Для Ruby есть библиотека, позволяющая выполнить описанные нормализации, хотя в стандартный дистрибутив она не входит. Вы можете скачать ее со страницы <http://www.yoshidam.net/Ruby.html> и установить командой `gem install unicode`.

Если библиотека unicode установлена, то для выполнения любой нормализации достаточно вызвать один из методов `Unicode.normalize_x:`

```
require 'unicode'
sword_kd = Unicode.normalize_KD(sword)
sword_kd.scan(/./) # ["e", "'", "p", "e", "'", "e"]
sword_kc = Unicode.normalize_KC(sword)
sword_kc.scan(/./) # ["é", "p", "é", "e"]
```

4.2.5. Упорядочение строк

Обычно, хотя и не всегда, строки упорядочиваются по алфавиту или сходным образом. *Упорядочение* тесно связано с нормализацией: в обоих случаях применяются одни и те же идеи и библиотеки.

Предположим, например, что мы хотим отсортировать такой массив строк:

```
eacute = [0x00E9].pack('U')
acute  = [0x0301].pack('U')
array = ["epicurian", "#{eacute}p#{eacute}e", "e#{acute}lan"]
# ["epicurian", "épée", "élan"]
```

Что произойдет, если передать этот массив методу `Array#sort`?

```
array.sort # ["epicurian", "élan", "épée"]
```

Не годится!.. Попытаемся понять, почему так получилось. Сортируемые строки Ruby сравнивает побайтно. Чтобы убедиться в этом, достаточно взглянуть на первые несколько байтов каждой строки:

```
array.map{|item| "#{item}: #{item.unpack('C*')[0,3].join(',')}" }
# ["epicurian: 101,112,105", "épée: 195,169,112",
#  "e'lan: 101,204,129"]
```

Тут возникают две трудности. Во-первых, символы UTF-8, не имеющие аналога в кодировке ASCII, начинаются с байта, имеющего большое числовое значение, а стало быть, после сортировки неизбежно окажутся после ASCII-символов. Во-вторых, составные латинские символы оказываются раньше монолитных из-за первого ASCII-байта.

В системные библиотеки обычно включают функции сортировки, которые сравнивают строки в соответствии с правилами конкретного языка. В библиотеке, поставляемой вместе с компилятором языка C, для этого служат функции `strxfrm` и `strcoll`.

Имейте в виду, что проблема возникает даже в случае кодировки ASCII. При сортировке ASCII-строк в Ruby производится прямое лексикографическое сравнение, однако в реальной жизни (например, если мы хотим отсортировать по названиям книги из библиотеки Конгресса США) есть много правил, которые не учитываются при таком упрощенном подходе.

Для упорядочения строк можно создать промежуточные строки и отсортировать именно их. Как конкретно это сделать, зависит от предъявляемых требований и языка; универсального алгоритма не существует.

Предположим, что список обрабатывается согласно правилам английского языка, причем диакритические знаки игнорируются. Первым делом нужно определить методику трансформации. Мы приведем все символы к составному виду, а затем исключим диакритические знаки, оставив только базовые символы. Для модифицирующих диакритических знаков в Unicode выделен диапазон от U+0300 до U+036F:

```
def transform(str)
  Unicode.normalize_KD(str).unpack('U*').select{|cp|
    cp < 0x0300 || cp > 0x036F
  }.pack('U*')
end

array.map{|x| transform(x) } # ["epicurian", "epee", "elan"]
```

Затем создадим хэшированную таблицу, чтобы установить соответствие между исходными и трансформированными строками, и воспользуемся ей для сортировки исходных строк. Наличие такой таблицы позволяет провести трансформацию только один раз.

```
def collate(array)
  transformations = array.inject({}) do |hash, item|
    hash[item] = yield item
  end
  hash
  array.sort_by {|x| transformations[x] }
end

collate(array) {|a| transform(a) }    # ["élan", "épée", "epicurian"]
```

Уже лучше, но мы еще не учли прописные буквы и эквивалентность символов. Возьмем для примера немецкий язык.

На самом деле в немецком языке есть несколько способов упорядочения; мы остановимся на стандарте DIN-2 (как в телефонном справочнике). Согласно этому стандарту, символ ß (эсцет) эквивалентен ss, а умляут эквивалентен букве e (то есть ö – то же самое, что oe и т.д.).

Наш метод трансформации должен учитывать эти детали. Снова начнем с декомпозиции составных символов. Например, модифицирующая трема (умляут) представляется кодовой позицией U+0308. За основу мы возьмем метод преобразования регистра, имеющийся в Ruby, но несколько дополним его. Вот как выглядит теперь код трансформации:

```
def transform_de(str)
  decomposed = Unicode.normalize_KD(str).downcase
  decomposed.gsub!('ß', 'ss')
  decomposed.gsub!([0x0308].pack('U'), 'e')
end

array = ["Straße", "öffnen"]
array.map {|x| transform_de(x) }    # ["strasse", "oeffnen"]
```

Не для всех языков годится такой прямолинейный подход. Например, в испанском между буквами n и o есть еще буква ñ. Однако, если каким-то образом сдвинуть оставшиеся буквы, то мы справимся и с этой проблемой. В листинге 4.1 для упрощения обработки нормализация применена к монолитным символам. Кроме того, мы облегчили себе жизнь, игнорируя различия между буквами с диакритическими знаками и без них.

Листинг 4.1. Упорядочение строк в испанском языке

```
def map_table(list)
  table = {}
  list.each_with_index do |item, i|
    item.split(',').each do |subitem|
      table[Unicode.normalize_KC(subitem)] = (?a + i).chr
    end
  end
end
```

```

end
end
table
end

```

```

ES_SORT = map_table(%w(
  a,A,á,Á b,B c,C d,D e,E,é,É f,F g,G h,H i,I,í,Í j,J k,K l,L m,M
  n,N ñ,Ñ o,O,ó,Ó p,P q,Q r,R s,S t,T u,U,ú,Ú v,V w,W x,X y,Y z,Z
))

```

```

def transform_es(str)
  array = Unicode.normalize_KC(str).scan(/./u)
  array.map { |c| ES_SORT[c] || c }.join
end

```

```

array = %w[éste estoy año apogeo amor]
array.map { |a| transform_es(a) }
# ["etue", "etupz", "aop", "aqpgper", "amps"]

collate(array) { |a| transform_es(a) }
# ["amor", "año", "apogeo", "éste", "estoy"]

```

В реальности упорядочение немного сложнее, чем показано в примерах выше; обычно требуется до трех уровней обработки. На первом уровне сравниваются только базовые символы без учета диакритических знаков и регистра, на втором учитываются диакритические знаки, а на третьем – регистр. Второй и третий уровень необходимы лишь в том случае, когда на предыдущих уровнях строки совпали. Кроме того, в некоторых языках последовательности, состоящие из нескольких символов, сортируются как единая семантическая единица (например, в хорватском *lj* расположено между *l* и *m*). Поэтому разработка языковозависимого или обобщенного алгоритма сортировки – задача нетривиальная: необходимо хорошо разбираться в конкретном языке. Невозможно изобрести по-настоящему универсальный алгоритм сортировки, который давал бы правильные результаты для всех языков, хотя попытки в этом направлении производились.

4.2.6. Преобразование из одной кодировки в другую

В стандартной библиотеке Ruby имеется интерфейс к библиотеке *iconv* для преобразования из одной кодировки символов в другую. Она должна работать на всех платформах, в том числе и в Windows (если дистрибутив устанавливался моментальным инсталлятором).

Чтобы преобразовать строку из UTF-8 в ISO-8859-15, библиотека *iconv* используется следующим образом:

```

require 'iconv'
converter = Iconv.new('ISO-8859-15', 'UTF-8')
sword_iso = converter.iconv(sword)

```

Важно помнить, что сначала указывается целевая кодировка, а потом исходная (как при присваивании). Количество и названия поддерживаемых кодировок

зависят от платформы, но наиболее распространенные стандартизованы и имеются везде. Если установлена пакетная утилита `iconv`, то перечень распознаваемых кодировок можно получить с помощью команды `iconv -l`.

Помимо названия кодировки, `iconv` принимает еще флаги, управляющие ее поведением. Они указываются в конце строки, содержащей целевую кодировку.

Обычно `iconv` возбуждает исключение, если получает недопустимые входные данные или почему-либо не может представить их в целевой кодировке. Флаг `//IGNORE` подавляет исключение.

```
broken_utf8_string = "hello\xfe"
converter = Iconv.new('ISO-8859-15', 'UTF-8')
# будет возбуждено исключение Iconv::IllegalSequence
converter.iconv(broken_utf8_string)
```

```
converter = Iconv.new('ISO-8859-15//IGNORE', 'UTF-8')
converter.iconv(broken_utf8_string) # "hello"
```

Этот же флаг позволяет очистить строку от неверных данных:

```
broken_sword = "épée\xfe"
converter = Iconv.new('UTF-8//IGNORE', 'UTF-8')
converter.iconv(broken_sword) # "épée"
```

Иногда некоторые символы нельзя представить в целевой кодировке. Обычно в этом случае возбуждается исключение. Флаг `//TRANSLIT` говорит `iconv`, что нужно вместо этого попытаться подобрать приблизительные эквиваленты.

```
converter = Iconv.new('ASCII', 'UTF-8')
converter.iconv(sword) # Возбуждается Iconv::IllegalSequence.
converter = Iconv.new('ASCII//IGNORE', 'UTF-8')
converter.iconv(sword) # "pe"
converter = Iconv.new('ASCII//TRANSLIT', 'UTF-8')
converter.iconv(sword) # "ep'ee"
```

Этим свойством можно воспользоваться, чтобы получить URL, содержащий только ASCII-символы:

```
str = "Straße épée"
converter = Iconv.new('ASCII//TRANSLIT', 'UTF-8')
converter.iconv(sword).gsub(/ /, '-').gsub(/[^\a-z\-\_]/in).downcase
# "strasse-epée"
```

Однако работать это будет лишь в отношении латиницы.

В листинге 4.2 приведен реальный пример совместного применения библиотек `iconv` и `open-uri` для скачивания Web-страницы и перекодирования ее в UTF-8.

Листинг 4.2. Перекодирование Web-страницы в кодировку UTF-8

```
require 'open-uri'
require 'iconv'

def get_web_page_as_utf8(url)
  open(url) do |io|
    source = io.read
```

```

type, *parameters = io.content_type_parse
# Не перекодировать, если не (X)HTML
unless type =~ %r!^(?:text/html|application/xhtml+xml)$!
  return source
end
# Сначала проверяем заголовки, присланные сервером:
if pair = parameters.assoc('charset')
  encoding = pair.last
# Затем анализируем HTML:
elsif source =~ /\s*charset=(["']\s*)+/i
  encoding = $1
# Если не удалось определить, предполагаем кодировку по умолчанию,
# определенную в стандарте HTTP.
else
  encoding = 'ISO-8859-1'
end
converter = Iconv.new('UTF-8//IGNORE', encoding)
return converter.iconv(source)
end
end

```

Это еще не все системные вопросы, связанные с преобразованием кодировок. Предположим, что в операционной системе, где установлен Ruby, определена локаль, отличная от UTF-8, или Ruby общается с ОС не в UTF-8 (так, например, обстоит дело в дистрибутиве для Win32). Тогда возникают дополнительные сложности.

Например, Windows поддерживает Unicode в именах файлов и на системном уровне работает исключительно в Unicode. Но в настоящее время Ruby взаимодействует с Windows при помощи устаревших кодовых страниц. Для англоязычного и большинства других западных изданий это страница 1252 (или windows-1252).

Внутри программы можно пользоваться и кодировкой UTF-8, но все имена файлов придется перевести в кодировку, заданную кодовой страницей. Iconv поможет это сделать, но важно не забывать, что кодовая страница позволяет описать только малое подмножество всех символов, имеющихся в Unicode.

Кроме того, это означает, что пока Ruby для Windows не может открывать файлы, имена которых нельзя описать с помощью кодовой страницы. Это ограничение не относится к Mac OS X, Linux и другим системам с локалью UTF-8.

4.3. Справочники сообщений

.....

*Ложбан не зависит от национальных особенностей.
Его словарь был создан алгоритмически на основе шести
наиболее распространенных в мире разговорных языков:
китайского, хинди, английского, русского, испанского и арабского.
Nick Nicholas, John Cowan. What is Lojban?*

Справочник сообщений — это набор сообщений на одном языке. Данное понятие неотъемлемо от концепции локализации (L10N). Идея в том, чтобы отделить

языково-зависимые строки от остальной программы. Тогда для того, чтобы программа «заговорила» на другом языке, достаточно всего лишь подменить справочник.

«Наилучший» способ реализовать эту идею в Ruby – воспользоваться библиотекой `Ruby-GetText-Package`. Я буду называть ее просто `gettext`, поскольку именно так называется содержащий ее файл. (не путайте с утилитой `gettext`!). Эту великолепную библиотеку написал Масао Мута (Masao Mutoh), он же очень помог при написании данного раздела.

Библиотека представляет собой реализацию на Ruby (не обертку) набора утилит `gettext` из проекта GNU (самый известный продукт в этой области). Ее официальный сайт находится по адресу <http://gettext.rubyforge.org/>, а утилиты GNU можно найти на сайте <http://www.gnu.org/software/gettext/>.

4.3.1. Исторические сведения и терминология

Библиотека `gettext` на самом деле, как мы увидим, состоит из нескольких библиотек. Для доступа к основным функциям нужно включить предложение `require 'gettext'`, а для получения разного рода дополнительных средств (в частности, работы со справочниками сообщений) – предложение `require 'gettext/utils'`.

Главная причина, по которой мы используем справочники сообщений, – это, конечно, перевод сообщений на другие языки. С их помощью мы также обрабатываем случаи, когда формы единственного и множественного числа различаются (один файл, два файла). Кстати, эти правила очень сильно зависят от конкретного языка.

Обычно у каждой библиотеки и приложения имеется собственный справочник сообщений. Следовательно, в дистрибутив можно включать набор переведенных на разные языки справочников.

Учитываются переменные окружения `LANG` и `GETTEXT_RATN`. Их назначение мы рассмотрим ниже.

Для сопровождения справочника сообщений есть две основных операции (они выполняются вне вашей программы): извлечь сообщения из исходного текста Ruby-программы для формирования начального справочника и включить новые сообщения из исходного текста в существующий справочник (слияние). Операции извлечения и слияния мы рассмотрим в разделе 4.3.3.

4.3.2. Приступаем к работе со справочниками сообщений

Возможно, библиотека `gettext` на вашем компьютере уже установлена. Если нет, проще всего выполнить команду `gem install gettext`.

Для разработки вам понадобятся утилиты GNU. Если вы работаете в системе UNIX, то, скорее всего, они уже установлены. В случае платформы Win32 можно установить Glade/GTK+ для Windows; заодно вы получите и утилиты GNU. В любом случае необходимы они только на этапе разработки, а не во время выполнения.

Если у вас нет программы `rake`, установите ее из `gem`-пакета. Это дополнительное удобство.

Коль скоро среда настроена и все установлено, можно приступать к работе со справочниками. Но сначала познакомимся с терминологией.

- ❶ *РО-файл* – это переносимый объектный файл. Так называется текстовое (понятное человеку) представление справочника сообщений. У каждого такого файла есть вариант для различных поддерживаемых локалей. *РОТ-файл* – это шаблон.
- ❷ *МО-файл* – это переносимый двоичный файл справочника. Он создается из РО-файла. Библиотека для Ruby умеет читать только МО-файлы, но не РО-файлы.
- ❸ *Текстовый домен* – это, по существу, просто базовое имя МО-файла. Он ассоциирован с приложением (привязан к нему).

4.3.3. Локализация простого приложения

В следующем примере определяется класс `Person`, после чего с ним выполняются различные действия. Метод `show` выводит локализованные сообщения:

```
require 'gettext'

class Person
  include GetText

  def initialize(name, age, children_num)
    @name, @age, @children_num = name, age, children_num
    bindtextdomain("myapp")
  end

  def show
    puts _("Information")
    puts _("Name: %{name}, Age: %{age}") % {:name => @name, :age => @age}
    puts n_("%{name} has a child.", "%{name} has %{num} children.",
            @children_num) % {:name => @name, :num => @children_num}
  end
end

john = Person.new("John", 25, 1)
john.show
linda = Person.new("Linda", 30, 3)
linda.show
```

Предположим, что этот код сохранен в файле `myapp/person.rb`. Как вы скоро увидите, иерархия каталогов имеет значение. Вызов метода `bindtextdomain` связывает текстовый домен "myapp" с объектом `Person` во время выполнения.

В методе `show` есть три обращения к библиотеке `gettext`. Вызываемый метод называется `_` (одно подчеркивание), чтобы не отвлекать внимание.

Первое обращение просто выводит локализованное сообщение, соответствующее строке "Information". Второе демонстрирует локализованное сообщение с двумя параметрами. В хэше задается список значений, подставляемых в строку. Интерполировать их напрямую нельзя, потому что это вступало бы в противоречие с основной целью: хранить в справочнике небольшое число сообщений.

Отметим также, что параметры отделены от текста сообщения, поэтому при необходимости могут подставляться в другом порядке. Ведь иногда при переводе на другой язык приходится переставлять слова.

Тот же метод можно вызвать и короче:

```
puts _("Name: %s, Age: %d") % [@name, @age]
```

Однако мы рекомендуем более длинную запись. Она понятнее и дает больше информации переводчику.

Метод `n_` предназначен для обработки единственного и множественного числа. Значение параметра `@children_num` – индекс, говорящий о том, какую из заранее заданных строк использовать. (Правило `Plural-Forms`, о котором я скоро расскажу, определяет порядок вычисления индекса.)

Отметим, что сообщения по умолчанию обязаны быть англоязычными (даже если родной язык программиста не английский). Нравится вам это или нет, но английский ближе всего к универсальному языку с точки зрения большинства переводчиков.

Я сказал, что нам пригодится программа `rake`. Создадим файл `Rakefile` (в каталоге `myapp`) для сопровождения справочников сообщений. Он будет выполнять две основные операции: обновлять РО-файлы и создавать МО-файлы.

```
require 'gettext/utils'
```

```
desc "Update pot/po files."
```

```
task :updatepo do
```

```
  GetText.update_pofiles("myapp", ["person.rb"], "myapp 1.0.0")
```

```
end
```

```
desc "Create mo-files"
```

```
task :makemo do
```

```
  GetText.create_mofiles
```

```
end
```

Здесь мы воспользовались библиотекой `gettext/utils`, в которой имеются функции для работы со справочниками сообщения. Метод `update_pofiles` создает начальный файл `myapp/po/myapp.pot` на основе исходного текста `person.rb`. При втором (и всех последующих) вызовах эта функция выполнит обновление, или слияние файла `myapp/po/myapp.pot` и всех файлов вида `myapp/po/#{lang}/myapp.po`.

Второй параметр – массив целевых файлов. Обычно он задается примерно так:

```
GetText.update_pofiles("myapp",  
  Dir.glob("{lib,bin}/**/*.{rb,rhtml}"),  
  "myapp 1.0.0")
```

Вызов метода `GetText.create_mofiles` создает необходимые подкаталоги в каталоге `data/locale/` и генерирует МО-файлы из РО-файлов.

Итак, выполнив команду `rake updatepo`, мы создадим каталог `myapp/po`, а в нем файл `myapp.pot`.

Теперь отредактируем заголовок файла `po/myapp.pot`. Он содержит описание приложения (название, имя автора, адрес электронной почты, условия лицензирования и т.д.).

```
# Пример приложения.
# Copyright (C) 2006 Foo Bar
# Файл распространяется по лицензии XXX.
#
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: myapp 1.0.0\n"
#...
```

(Осмысленное название)

(Автор приложения)

(Лицензия)

(Информация о переводчике)

(ID и версия проекта)

Что такое маркер `fuzzy`? Так отмечается тот факт, что какая-то часть не переведена или перевод вызывает сомнения. Все автоматически сгенерированные сообщения помечаются таким образом, чтобы человек знал, что их нужно проверить и изменить.

Файл `myapp.pot` нужно разослать переводчикам. (Конечно, вы можете перевести его и самостоятельно.)

Предположим, что вы переводите на японский язык. На машине установлена локаль `ja_JP.UTF-8`, что означает «Япония (`ja`), японский язык (`JP`), кодировка UTF-8».

Для начала скопируем файл `myapp.pot` в `myapp.po`. При наличии набора GNU-утилит `gettext` лучше воспользоваться командой `msginit`, а не просто `cp`. Эта утилита учитывает переменные окружения и правильно устанавливает некоторые переменные в заголовке. В UNIX она вызывается следующим образом:

```
LANG=ja_JP.UTF-8 msginit -i myapp.pot -o myapp.po
```

Затем отредактируйте файл `myapp.po`, как показано в листинге 4.3. Редактировать необходимо в той кодировке, которая указана в строке `Content-Type`.

Листинг 4.3. Файл `myapp.po` после редактирования

```
# Пример приложения.
# Copyright (C) 2006 Foo Bar
# Файл распространяется по лицензии XXX.
#
# Ваше имя <yourname@foo.com>, 2006.
#
msgid ""
msgstr ""
"Project-Id-Version: myapp 1.0.0\n"
"POT-Creation-Date: 2006-05-22 23:27+0900\n"
"PO-Revision-Date: 2006-05-23 14:39+0900\n"
# Информация о текущем переводчике.
"Last-Translator: Your Name <foo@bar.com>\n"
"Language-Team: Japanese\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

(Вся информация о переводчике)

(Удалите строку 'fuzzy')

(Ваш язык)

(Кодировка файла)

```
# Форма множественного числа.
*Plural-Forms: nplurals=2; plural=(n != 1);\n"
```

```
#: person.rb:12
msgid "Information"
msgstr "Jouhou"
```

```
#: person.rb:13
msgid "Name: %{name}, Age: %{age}"
msgstr "Namae: %{name}, Nenrei: %{age}"
```

```
#: person.rb:14
msgid "%{name} has a child."
msgid_plural "%{name} has %{num} children."
msgstr[0] "%{name} ha hitori kodomo ga imasu."
msgstr[1] "%{name} ha %{num} nin no kodomo ga imasu."
```

Тегом `msgid` помечается исходное сообщение, а тегом `msgstr` – переведенное. При наличии строки `msgid_plural` необходимо включить отдельные строки `msgstr[i]` в соответствии с правилом `Plural-Forms`. Индекс `i` вычисляется на основе выражения `Plural-Forms`. В данном случае при `num != 1` используется `msgstr[1]` (сообщение с существительным во множественном числе).

Истоки синтаксиса правила `Plural-Forms` следует искать в языке C. Как видим, он опирается на тот факт, что булевские выражения в C возвращают 0 или 1.

Имейте в виду, что формы единственного и множественного числа в большой степени зависят от языка. Во многих языках есть несколько форм множественного числа. Например, в польском слово «файл» в единственном числе записывается как «plik». Если количество экземпляров заканчивается на 2, 3 и 4, то во множественном числе пишется «pliki», а во всех остальных случаях – «plików».

Поэтому для польского языка правило `Plural-Forms` выглядит так:

```
Plural-Forms: nplurals=3; \
    plural=n==1 ? 0 : \
    n%10>=2 && n%10<=4 && (n%100=20) ? 1 : 2;
```

Заголовок файла – не пустая формальность. Особенно важны разделы `Content-Type` и `Plural-Forms`. При использовании утилитой `msginit` они вставляются автоматически, в противном случае необходимо добавить их вручную.

Закончив работу, переводчик посылает файлы обратно разработчику (или вы сами возвращаетесь к роли разработчика).

Файлы `myapp.po`, полученные от переводчиков, помещаются в соответствующие каталоги (внутри каталога `myapp.po`). Например, французскую версию следовало бы поместить в каталог `myapp.po/fr/myapp.po`, немецкую – в каталог `myapp.po/de/myapp.po` и т.д.

Затем выполните команду `rake makemo`. Она преобразует `PO`-файлы в `MO`-файлы. Сгенерированные `MO`-файлы будут помещены в каталог `myapp/data/locale/` (в котором есть подкаталоги для каждого языка).

Таким образом, окончательная структура дерева каталогов выглядит так:

```
myapp/
  Rakefile
  person.rb
  po/
    myapp.pot
    de/myapp.po
    fr/myapp.po
    ja/myapp.po
  :
  data/
    locale/
      de/LC_MESSAGES/myapp.mo
      fr/LC_MESSAGES/myapp.mo
      ja/LC_MESSAGES/myapp.mo
    :
```

Перевод закончен, можно протестировать пример. Но предварительно следует указать, где искать МО-файлы и для какой локали проводится тестирование. Установим переменные окружения `GETTEXT_PATH` и `LANG`, запустим программу и посмотрим, что она выведет.

```
export GETTEXT_PATH="data/locale"
export LANG="ja_JP.UTF-8"
ruby person.rb
```

Программа выводит локализованные сообщения в соответствии со значением переменной `LANG`.

4.3.4. Прочие замечания

Если вы распространяете вместе со своей программой справочники сообщений, то лучше собрать пакет с помощью системы `RubyGems` или библиотеки `setup.rb`. Дополнительную информацию по этому поводу вы найдете в разделе 17.2.

При установке пакета, собранного `RubyGems`, справочники сообщений копируются в каталоги вида:

```
(gem-packages-installed-dir)/myapp-x.x.x/data/locale/
```

Такие каталоги уже включены в путь поиска для библиотеки `gettext`, поэтому ваша программа будет локализована даже без явной установки переменной окружения `GETTEXT_PATH`.

В случае сборки пакета с помощью библиотеки `setup.rb` справочники сообщений помещаются в каталог `(system-dir)/share/locale/`. И в этом случае локализация достигается без установки переменной `GETTEXT_PATH`.

Напомним, что описанная библиотека не является оберткой набора утилит `gettext` от GNU. Однако файлы сообщений совместимы, поэтому при желании вы можете пользоваться средствами сопровождения GNU. Понятно, что во время выполнения программы все эти инструменты не нужны, то есть пользователь не обязан их устанавливать на свой компьютер.

4.4. Заключение

В этой главе мы рассмотрели один из самых сложных аспектов программирования – проблему интернационализации кода. При этом нам понадобился материал из двух предыдущих глав, так как интернационализация тесно связана со строками и регулярными выражениями.

Мы видели, что в Ruby некоторые задачи решаются просто благодаря наличию библиотеки `jscode` и сопутствующих инструментов. Заодно мы познакомились с наборами символов вообще и с набором Unicode в частности.

Мы узнали, что регулярные выражения в общем случае лучше поддерживают Unicode, чем средства работы со строками, а также рассмотрели методы `pack` и `unpack` с точки зрения полезности для манипулирования Unicode-строками.

Наконец, мы довольно подробно остановились на справочниках сообщений. Мы поняли, для чего они нужны, как их создавать и поддерживать.

Детально рассмотрев вопрос о строках и регулярных выражениях, вернемся на главную дорогу. Глава 5 посвящена численному анализу в языке Ruby.

Глава 5. Численные методы

.....
Дважды [члены Парламента] задавали мне вопрос: «А скажите, мистер Бэббидж, если вы зложите в эту машину неверные числа, то получите правильный результат?» Не могу даже представить себе, насколько извращенно должен мыслить человек, задающий такие вопросы.

Чарльз Бэббидж

Числа – самый первичный тип данных, естественный для любого компьютера. Придется сильно постараться, чтобы найти такую область знания, в которой нет места числам. Будь вы бухгалтером или конструктором воздухоплавательных аппаратов, без чисел вам не обойтись. В этой главе мы обсудим различные способы обработки, преобразования и анализа числовых данных.

Как и всякий современный язык, Ruby прекрасно умеет работать с любыми числами – как целыми, так и с плавающей точкой. В нем есть полный набор ожидаемых математических операторов и функций, а вместе с тем и кое-какие приятные сюрпризы: классы `Bignum`, `BigDecimal` и `Rational`.

Помимо средств для манипуляции числами, имеющихся в системной и стандартной библиотеках, мы рассмотрим более специфические темы (тригонометрия, математический анализ и статистика). Примеры приведены не только для справки, но и как образцы кода на языке Ruby, иллюстрирующие принципы, изложенные в других частях книги.

5.1. Представление чисел в языке Ruby

Если вы знакомы с любым другим языком программирования, то представление чисел в Ruby не вызовет у вас никакого удивления. Объект класса `Fixnum` может представлять число со знаком или без знака:

```
237          # Число без знака (положительное).  
+237         # То же, что и выше.  
-237         # Отрицательное число.
```

Если число длинное, то между любыми цифрами можно вставлять знак подчеркивания. Это сделано исключительно для удобства, на значении константы `nil` как не сказывается. Обычно подчеркивки вставляются в те же места, где бухгалтеры вставляют пробелы:

```
1048576      # Число в обычной записи.  
1_048_576    # То же самое значение.
```

Целые числа можно представлять и в других системах счисления (по основанию 2, 8 и 16). Для этого в начале ставятся префиксы 0b, 0 и 0x соответственно.

```
0b10010110    # Двоичное.
0b1211         # Ошибка!
01234         # Восьмеричное (основание 8).
01823         # Ошибка!
0xdeadbeef     # Шестнадцатеричное (основание 16).
0xDEADBEEF     # То же самое.
0xdeadpork     # Ошибка!
```

В числах с плавающей точкой десятичная точка должна присутствовать, а показатель степени, возможно со знаком, необязателен:

```
3.14          # Число пи, округленное до сотых.
-0.628        # -2*pi, поделенное на 10, округленное до тысячных.
6.02e23       # Число Авогадро.
6.626068e-34  # Постоянная Планка.
```

В классе Float есть константы, определяющие минимальные и максимальные значения чисел с плавающей точкой. Они машиннозависимы. Вот некоторые наиболее важные:

```
Float::MIN     # 2.2250738585072e-308 (на конкретной машине)
Float::MAX     # 1.79769313486232e+308
Float::EPSILON # 2.22044604925031e-16
```

5.2. Основные операции над числами

Обычные операции сложения, вычитания, умножения и деления в Ruby, как и во всех распространенных языках программирования, обозначаются операторами +, -, *, /. Операторы в большинстве своем реализованы в виде методов (и потому могут быть переопределены).

Возведение в степень обозначается оператором **, как в языках BASIC и FORTRAN. Эта операция подчиняется обычным математическим правилам.

```
a = 64**2      # 4096
b = 64**0.5    # 8.0
c = 64**0      # 1
d = 64**-1     # 0.015625
```

При делении одного целого числа на другое дробная часть результата отбрасывается. Это не ошибка, так и задумано. Если вы хотите получить результат с плавающей точкой, позаботьтесь о том, чтобы хотя бы один из операндов был числом с плавающей точкой.

```
3 / 3          # 3
5 / 3          # 1
3 / 4          # 0
3.0 / 4        # 0.75
3 / 4.0        # 0.75
3.0 / 4.0      # 0.75
```

Если вы работаете с переменными и сомневаетесь относительно их типа, воспользуйтесь приведением типа к `Float` или методом `to_f`:

```
z = x.to_f / y
z = Float(x) / y
```

См. также раздел 5.17 «Поразрядные операции над числами».

5.3. Округление чисел с плавающей точкой

.....
Кирк: Какие, вы говорите, у нас шансы выбраться отсюда?

*Спок: Трудно сказать точно, капитан. Приблизительно 7824.7 к одному.
 Стар Трек, «Миссия милосердия»*

Метод `round` округляет число с плавающей точкой до целого:

```
pi = 3.14159
new_pi = pi.round           # 3
temp = -47.6
temp2 = temp.round         # -48
```

Иногда бывает нужно округлить не до целого, а до заданного числа знаков после запятой. В таком случае можно воспользоваться функциями `sprintf` (которая умеет округлять) и `eval`:

```
pi = 3.1415926535
pi6 = eval(sprintf("%8.6f",pi)) # 3.141593
pi5 = eval(sprintf("%8.5f",pi)) # 3.14159
pi4 = eval(sprintf("%8.4f",pi)) # 3.1416
```

Это не слишком красиво. Поэтому инкапсулируем оба вызова функций в метод, который добавим в класс `Float`:

```
class Float

  def roundf(places)
    temp = self.to_s.length
    sprintf("%#{temp}.#{places}f",self).to_f
  end

end
```

Иногда требуется округлять до целого по-другому. Традиционное округление $n+0.5$ с избытком со временем приводит к небольшим ошибкам; ведь $n+0.5$ все-таки ближе к $n+1$, чем к n . Есть другое соглашение: округлять до ближайшего четного числа, если дробная часть равна 0.5 . Для реализации такого правила можно было бы расширить класс `Float`, добавив в него метод `round2`:

```
class Float

  def round2
    whole = self.floor
```

```
fraction = self - whole
if fraction == 0.5
  if (whole % 2) == 0
    whole
  else
    whole+1
  end
else
  self.round
end
end

end
```

```
a = (33.4).round2 # 33
b = (33.5).round2 # 34
c = (33.6).round2 # 34
d = (34.4).round2 # 34
e = (34.5).round2 # 34
f = (34.6).round2 # 35
```

Видно, что `round2` отличается от `round` только в том случае, когда дробная часть в точности равна 0.5. Отметим, кстати, что число 0.5 можно точно представить в двоичном виде. Не так очевидно, что этот метод правильно работает и для отрицательных чисел (попробуйте!). Отметим еще, что скобки в данном случае необязательны и включены в запись только для удобства восприятия.

Ну а если мы хотим округлять до заданного числа знаков после запятой, но при этом использовать метод «округления до четного»? Тогда нужно добавить в класс `Float` также метод `roundf2`:

```
class Float

  # Определение round2 такое же, как и выше.
  def roundf2(places)
    shift = 10**places
    (self * shift).round2 / shift.to_f
  end

end
```

```
a = 6.125
b = 6.135
x = a.roundf2(a) # 6.12
y = b.roundf2(b) # 6.13
```

У методов `roundf` и `roundf2` есть ограничение: большое число с плавающей точкой может стать непредставимым при умножении на большую степень 10. На этот случай следовало бы предусмотреть проверку ошибок.

5.4. Сравнение чисел с плавающей точкой

Печально, но факт: в компьютере числа с плавающей точкой представляются не-точно. В идеальном мире следующий код напечатал бы «да», но на всех машинах, где мы его запускали, печатается «нет»:

```
x = 1000001.0/0.003
y = 0.003*x
if y == 1000001.0
  puts "да"
else
  puts "нет"
end
```

Объясняется это тем, что для хранения числа с плавающей точкой выделено конечное число битов, а с помощью любого, сколь угодно большого, но конечного числа битов нельзя представить периодическую десятичную дробь с бесконечным числом знаков после запятой.

Из-за этой неустранимой неточности при сравнении чисел с плавающей точкой мы можем оказаться в ситуации (продемонстрированной выше), когда с практической точки зрения два числа равны, но аппаратура упрямо считает их различными.

Ниже показан простой способ выполнения сравнения с «поправкой», когда числа считаются равными, если отличаются не более чем на величину, задаваемую программистом:

```
class Float

  EPSILON = 1e-6 # 0.000001

  def ==(x)
    (self-x).abs < EPSILON
  end

end

x = 1000001.0/0.003
y = 0.003*x
if y == 1.0 # Пользуемся новым оператором ==.
  puts "да" # Теперь печатается "да".
else
  puts "нет"
end
```

В зависимости от ситуации может понадобиться задавать разные погрешности. Для этого определим в классе Float новый метод `equals?`. (При таком выборе имени мы избежим конфликта со стандартными методами `equal?` и `eq?`; последний, кстати, вообще не следует переопределять).

```
class Float

  EPSILON = 1e-6
```

```
def equals?(x, tolerance=EPSILON)
  (self-x).abs < tolerance
end

flag1 = (3.1416).equals? Math::PI # false
flag2 = (3.1416).equals?(Math::PI, 0.001) # true
```

Можно также ввести совершенно новый оператор для приближенного сравнения, назвав его, например, `==~`.

Имейте в виду, что это нельзя назвать настоящим решением. При последовательных вычислениях погрешность накапливается. Если вам совершенно необходимы числа с плавающей точкой, смиритесь с неточностями (см. также разделы 5.8 и 5.9).

5.5. Форматирование чисел для вывода

Для вывода числа в заданном формате применяется метод `printf` из модуля `Kernel`. Он практически не отличается от одноименной функции в стандартной библиотеке C. Дополнительную информацию см. в документации по методу `printf`.

```
x = 345.6789
i = 123
printf("x = %6.2f\n", x) # x = 345.68
printf("x = %9.2e\n", x) # x = 3.457e+02
printf("i = %5d\n", i) # i = 123
printf("i = %05d\n", i) # i = 00123
printf("i = %-5d\n", i) # i = 123
```

Чтобы сохранить результат в строке, а не печатать его немедленно, воспользуйтесь методом `sprintf`. При следующем обращении возвращается строка:

```
str = sprintf("%5.1f", x) # "345.7"
```

Наконец, в классе `String` есть метод `%`, решающий ту же задачу. Слева от знака `%` должна стоять форматная строка, а справа – единственный аргумент (или массив значений), результатом является строка.

```
# Порядок вызова: 'формат % значение'
str = "%5.1f" % x # "345.7"
str = "%6.2f, %05d" % [x,i] # "345.68, 00123"
```

5.6. Вставка разделителей при форматировании чисел

Возможно, есть и более удачные способы достичь цели, но приведенный ниже код работает. Мы инвертируем строку, чтобы было удобнее выполнять глобальную замену, а в конце инвертируем ее еще раз:

```
def commas(x)
  str = x.to_s.reverse
  str.gsub!(/([0-9]{3})/, "\1,")
```

```

str.gsub(/,$/, "").reverse
end

puts commas(123)           # "123"
puts commas(1234)          # "1,234"
puts commas(12345)         # "12,435"
puts commas(123456)        # "123,456"
puts commas(1234567)       # "1,234,567"

```

5.7. Работа с очень большими числами

.....
*Управлять массами все равно что управлять немногими:
 дело в частях и в числе.
 Сунь-Цзы**

При необходимости Ruby позволяет работать с произвольно большими целыми числами. Переход от `Fixnum` к `Bignum` производится автоматически, прозрачно для программиста. В следующем разделе результат оказывается настолько большим, что преобразуется из объекта `Fixnum` в `Bignum`:

```

num1 = 1000000             # Один миллион (10**6)
num2 = num1*num1           # Один триллион (10**12)
puts num1                  # 1000000
puts num1.class            # Fixnum
puts num2                  # 100000000000000
puts num2.class            # Bignum

```

Размер `Fixnum` зависит от машинной архитектуры. Вычисления с объектами `Bignum` ограничены только объемом памяти и быстродействием процессора. Конечно, они потребляют больше памяти и выполняются несколько медленнее, тем не менее операции над очень большими целыми (сотни знаков) реальны.

5.8. Использование класса `BigDecimal`

Стандартная библиотека `bigdecimal` позволяет работать с дробями, имеющими много значащих цифр. Число хранится как массив цифр, а не преобразуется в двоичное представление. Тем самым достижима произвольная точность, естественно, ценой замедления работы.

Чтобы оценить преимущества, рассмотрим следующий простой фрагмент кода, в котором используются числа с плавающей точкой:

```

if (3.2 - 2.0) == 1.2
  puts "равны"
else
  puts "не равны"          # Печатается "не равны"!
end

```

* Тракта́т «Искусство войны».

В подобной ситуации на помощь приходит класс `BigDecimal`. Однако в случае бесконечных периодических дробей проблема остается. Другой подход обсуждается в разделе 5.9 «Работа с рациональными числами».

Объект `BigDecimal` инициализируется строкой. (Объекта типа `Float` было бы недостаточно, поскольку погрешность вкралась бы еще до начала конструирования `BigDecimal`.) Метод `BigDecimal` эквивалентен `BigDecimal.new`; это еще один особый случай, когда имя метода начинается с прописной буквы. Поддерживаются обычные математические операции, например `+` и `*`. Отметим, что метод `to_s` может принимать в качестве параметра форматную строку. Дополнительную информацию вы найдете на сайте ruby-doc.org.

```
require 'bigdecimal'

x = BigDecimal("3.2")
y = BigDecimal("2.0")
z = BigDecimal("1.2")

if (x - y) == z
  puts "равны"          # Печатается "равны"!
else
  puts "не равны"
end

a = x*y*z
a.to_s      # "0.768E1" (по умолчанию: научная нотация)
a.to_s("F") # "7.68"    (обычная запись)
```

Если необходимо, можно задать число значащих цифр. Метод `prec` возвращает эту информацию в виде массива, содержащего два числа: количество использованных байтов и максимальное число значащих цифр.

```
x = BigDecimal("1.234",10)
y = BigDecimal("1.234",15)
x.prec      # [8, 16]
y.prec      # [8, 20]
```

В каждый момент число использованных байтов может оказаться меньше максимального. Максимум может также оказаться больше запрошенного вами (поскольку `BigDecimal` пытается оптимизировать использование внутренней памяти).

У обычных операций (сложение, вычитание, умножение и деление) есть варианты, принимающие в качестве дополнительного параметра число значащих цифр. Если результат содержит больше значащих цифр, чем указано, производится округление до заданного числа знаков.

```
a = BigDecimal("1.23456")
b = BigDecimal("2.45678")

# В комментариях "BigDecimal:objectid" опущено.
c = a+b      # <'0.369134E1',12(20)>
c2 = a.add(b,4) # <'0.3691E1',8(20)>
```



```
d = a-b          # <' -0.122222E1',12(20)>
d2 = a.sub(b,4)   # <' -0.1222E1',8(20)>

e = a*b           # <'0.3033042316 8E1',16(36)>
e2 = a.mult(b,4)  # <'0.3033E1',8(36)>

f = a/b           # <'0.5025114173 8372992290 7221E0',24(32)>
f2 = a.div(b,4)   # <'0.5025E0',4(16)>
```

В классе `BigDecimal` определено и много других функций, например `floor`, `abs` и т.д. Как и следовало ожидать, имеются операторы `%` и `**`, а также операторы сравнения, к примеру `<`. Оператор `==` не умеет округлять свои операнды – эта обязанность возлагается на программиста.

В модуле `bigmath` определены константы `E` и `PI` с произвольной точностью. (На самом деле это методы, а не константы.) Там же определены функции `sin`, `cos`, `exp` и пр.; все они принимают число значащих цифр в качестве параметра.

Следующие подбиблиотеки являются дополнениями к `BigDecimal`.

<code>bigdecimal/math</code>	Модуль <code>BigMath</code>
<code>bigdecimal/jacobian</code>	Методы для вычисления матрицы Якоби
<code>bigdecimal/ludcmp</code>	Модуль <code>LUSolve</code> , разложение матрицы в произведение верхнетреугольной и нижнетреугольной
<code>bigdecimal/newton</code>	Методы <code>nlscolve</code> и <code>norm</code>

В настоящей главе эти подбиблиотеки не описываются. Для получения дополнительной информации обратитесь к сайту ruby-doc.org или любому подробному справочному руководству.

5.9. Работа с рациональными числами

Класс `Rational` позволяет (во многих случаях) производить операции с дробями с «бесконечной» точностью, но лишь если это настоящие рациональные числа (то есть частное от деления двух целых чисел). К иррациональным числам, например π или e , он неприменим.

Для создания рационального числа мы вызываем специальный метод `Rational` (еще один из немногих методов, имя которого начинается с прописной буквы; обычно такие методы служат для преобразования данных или инициализации).

```
r = Rational(1,2)  # 1/2 или 0.5
s = Rational(1,3)  # 1/3 или 0.3333...
t = Rational(1,7)  # 1/7 или 0.14...
u = Rational(6,2)  # "то же самое, что" 3.0
z = Rational(1,0)  # Ошибка!
```

Результатом операции над двумя рациональными числами, как правило, снова является рациональное число.

```
r+t      # Rational(9, 14)
r-t      # Rational(5, 14)
r*s      # Rational(1, 6)
r/s      # Rational(3, 2)
```

Вернемся к примеру, на котором мы демонстрировали неточность операций над числами с плавающей точкой (см. раздел 5.4). Ниже мы выполняем те же действия над рациональными, а не вещественными числами и получаем «математически ожидаемый» результат:

```
x = Rational(1000001,1)/Rational(3,1000)
y = Rational(3,1000)*x
if y == 1000001.0
  puts "да"           # Теперь получаем "да"!
else
  puts "нет"
end
```

Конечно, не любая операция дает рациональное же число в качестве результата:

```
x = Rational(9,16)      # Rational(9, 16)
Math.sqrt(x)            # 0.75
x**0.5                  # 0.75
x**Rational(1,2)        # 0.75
```

Однако библиотека `mathn` в какой-то мере изменяет это поведение (см. раздел 5.12).

5.10. Перемножение матриц

Стандартная библиотека `matrix` предназначена для выполнения операций над числовыми матрицами. В ней определено два класса: `Matrix` и `Vector`.

Следует также знать о прекрасной библиотеке `Array`, которую написал Масахиро Танака (Masahiro Tanaka) – ее можно найти на сайте www.rubyforge.org. Хотя эта библиотека не относится к числу стандартных, она широко известна и очень полезна. Если вы предъявляете повышенные требования к быстродействию, нуждаетесь в особом представлении данных или желаете выполнять быстрое преобразование Фурье, обязательно ознакомьтесь с этим пакетом. Впрочем, для типичных применений стандартной библиотеки `matrix` должно хватить, поэтому именно ее мы и рассмотрим.

Чтобы создать матрицу, мы, конечно же, обращаемся к методу класса. Сделать это можно несколькими способами. Самый простой – вызвать метод `Matrix.[]` и перечислить строки в виде массивов. Ниже мы записали вызов на нескольких строчках, но, разумеется, это необязательно:

```
m = Matrix[[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Вместо этого можно вызвать метод `rows`, передав ему массив массивов (в таком случае «дополнительные» скобки необходимы). Необязательный параметр `copy`, по умолчанию равный `true`, указывает, надо ли скопировать переданные массивы или просто сохранить на них ссылки. Оставляйте значение `true`, если нужно защитить исходные массивы от изменений, и задавайте `false`, если это несущественно.

```

row1 = [2,3]
row2 = [4,5]
m1 = Matrix.rows([row1,row2])      # copy=true
m2 = Matrix.rows([row1,row2],false) # Не копировать.
row1[1] = 99                        # Теперь изменим row1.
p m1                                # Matrix[[2, 3], [4, 5]]
p m2                                # Matrix[[2, 99], [4, 5]]

```

Можно задать матрицу и путем перечисления столбцов, если воспользоваться методом `columns`. Ему параметр `copy` не передается, потому что столбцы в любом случае расщепляются, так как во внутреннем представлении матрица хранится построчно:

```

m1 = Matrix.rows([[1,2],[3,4]])
m2 = Matrix.columns([[1,3],[2,4]])  # m1 == m2

```

Предполагается, что все матрицы прямоугольные, но это не проверяется. Если вы создадите матрицу, в которой отдельные строки или столбцы длиннее либо короче остальных, то можете получить неверные или неожиданные результаты.

Некоторые специальные матрицы, особенно квадратные, конструируются проще. Так, тождественную матрицу конструирует метод `identity` (или его синонимы `I` и `unit`):

```

im1 = Matrix.identity(3)      # Matrix[[1,0,0],[0,1,0],[0,0,1]]
im2 = Matrix.I(3)             # То же самое.
im3 = Matrix.unit(3)          # То же самое.

```

Более общий метод `scalar` строит диагональную матрицу, в которой все элементы на диагонали одинаковы, но не обязательно равны 1:

```

sm = Matrix.scalar(3,8)       # Matrix[[8,0,0],[0,8,0],[0,0,8]]

```

Еще более общим является метод `diagonal`, который формирует диагональную матрицу с произвольными элементами (ясно, что параметр, задающий размерность, в этом случае не нужен).

```

dm = Matrix.diagonal(2,3,7)   # Matrix[[2,0,0],[0,3,0],[0,0,7]]

```

Метод `zero` создает нулевую матрицу заданной размерности (все элементы равны 0):

```

zm = Matrix.zero(3)           # Matrix[[0,0,0],[0,0,0],[0,0,0]]

```

Понятно, что методы `identity`, `scalar`, `diagonal` и `zero` создают квадратные матрицы.

Чтобы создать матрицу размерности $1 \times N$ или $N \times 1$, воспользуйтесь методом `row_vector` или `column_vector` соответственно.

```

a = Matrix.row_vector(2,4,6,8)   # Matrix[[2,4,6,8]]
b = Matrix.column_vector(6,7,8,9) # Matrix[[6],[7],[8],[9]]

```

К отдельным элементам матрицы можно обращаться, указывая индексы в квадратных скобках (оба индекса заключаются в одну пару скобок). Отметим, что не существует метода `[]`. По той же причине, по которой его нет в классе `fixnum`: матрицы – неизменяемые объекты (такое решение было принято автором библиотеки).

```

m = Matrix[[1,2,3],[4,5,6]]
puts m[1,2]    # 6

```

Индексация начинается с 0, как и для массивов в Ruby. Возможно, это противоречит вашему опыту работы с матрицами, но индексация с 1 в качестве альтернативы не предусмотрена. Можно реализовать эту возможность самостоятельно:

Наивный подход... не поступайте так!

```
class Matrix
  alias bracket []

  def [](i,j)
    bracket(i-1,j-1)
  end
end

m = Matrix[[1,2,3],[4,5,6],[7,8,9]]
p m[2,2] # 5
```

На первый взгляд, этот код должен работать. Большинство операций над матрицами даже будет давать правильный результат при такой индексации. Так в чем же проблема? В том, что мы не знаем деталей внутренней реализации класса `Matrix`. Если в нем для доступа к элементам матрицы всегда используется собственный метод `[]`, то все будет хорошо. Но если где-нибудь имеются прямые обращения к внутреннему массиву или применяются иные оптимизированные решения, то возникнет ошибка. Поэтому, решившись на такой трюк, вы должны тщательно протестировать новое поведение.

К тому же необходимо изменить методы `row` и `vector`. В них индексы тоже начинаются с 0, но метод `[]` не вызывается. Я не проверял, что еще придется модифицировать.

Иногда необходимо узнать размерность или форму матрицы. Для этого есть разные методы, например `row_size` и `column_size`.

Метод `row_size` возвращает число строк в матрице. Что касается метода `column_size`, тут есть одна тонкость: он проверяет лишь размер первой строки. Если по каким-либо причинам матрица не прямоугольная, то полученное значение бессмысленно. Кроме того, поскольку метод `square?` (проверяющий, является ли матрица квадратной) обращается к `row_size` и `column_size`, его результат тоже нельзя считать стопроцентно надежным.

```
m1 = Matrix[[1,2,3],[4,5,6],[7,8,9]]
m2 = Matrix[[1,2,3],[4,5,6],[7,8]]
m1.row_size # 3
m1.column_size # 3
m2.row_size # 3
m2.column_size # 3 (неправильно)
m1.square? # true
m2.square? # true (неправильно)
```

Решить эту мелкую проблему можно, например, определив метод `rectangular?`.

```
class Matrix
  def rectangular?
    arr = to_a
```

```

first = arr[0].size
arr[1..-1].all? {|x| x.size == first }
end
end

```

Можно, конечно, модифицировать метод `square?`, так чтобы сначала он проверял, является ли матрица прямоугольной. В таком случае нужно будет изменить метод `column_size`, чтобы он возвращал `nil` для непрямоугольной матрицы.

Для вырезания части матрицы имеется несколько методов. Метод `row_vectors` возвращает массив объектов класса `Vector`, представляющих строки (см. обсуждение класса `Vector` ниже.) Метод `column_vectors` работает аналогично, но для столбцов. Наконец, метод `minor` возвращает матрицу меньшего размера; его параметрами являются либо четыре числа (нижняя и верхняя границы номеров строк и столбцов), либо два диапазона.

```

m = Matrix[[1,2,3,4],[5,6,7,8],[6,7,8,9]]

rows = m.row_vectors           # Три объекта Vector.
cols = m.column_vectors        # Четыре объекта Vector.
m2 = m.minor(1,2,1,2)          # Matrix[[6,7],[7,8]]
m3 = m.minor(0..1,1..3)        # Matrix[[2,3,4],[6,7,8]]

```

К матрицам применимы обычные операции: сложение, вычитание, умножение и деление. Для выполнения некоторых из них должны соблюдаться ограничения на размеры матриц-операндов; в противном случае будет возбуждено исключение (например, при попытке перемножить матрицы размерностей 3×3 и 4×4).

Поддерживаются стандартные преобразования: `inverse` (обращение), `transpose` (транспонирование) и `determinant` (вычисление определителя). Для целочисленных матриц определитель лучше вычислять с помощью библиотеки `mathn` (раздел 5.12).

Класс `Vector` – это, по существу, частный случай одномерной матрицы. Его объект можно создать с помощью методов `[]` или `elements`; в первом случае параметром является развернутый массив, а во втором – обычный массив и необязательный параметр `copy` (по умолчанию равный `true`).

```

arr = [2,3,4,5]
v1 = Vector[*arr]                # Vector[2,3,4,5]
v2 = Vector.elements(arr)        # Vector[2,3,4,5]
v3 = Vector.elements(arr, false) # Vector[2,3,4,5]
arr[2] = 7                       # Теперь v3 – Vector[2,3,7,5].

```

Метод `covector` преобразует вектор длины N в матрицу размерности $N \times 1$ (выполняя попутно транспонирование).

```

v = Vector[2,3,4]
m = v.covector                    # Matrix[[2,3,4]]

```

Поддерживается сложение и вычитание векторов одинаковой длины. Вектор можно умножать на матрицу и на скаляр. Все эти операции подчиняются обычным математическим правилам.

```

v1 = Vector[2,3,4]
v2 = Vector[4,5,6]

```

```
v3 = v1 + v2           # Vector[6,8,10]
v4 = v1*v2.covector    # Matrix[[8,10,12],[12,15,18],[16,20,24]]
v5 = v1*5              # Vector[10,15,20]
```

Имеется метод `inner_product` (скалярное произведение):

```
v1 = Vector[2,3,4]
v2 = Vector[4,5,6]
x  = v1.inner_product(v2)  # 47
```

Дополнительную информацию о классах `Matrix` и `Vector` можно найти в любом справочном руководстве, например воспользовавшись командной утилитой `ri`, или на сайте ruby-doc.org.

5.11. Комплексные числа

Стандартная библиотека `complex` предназначена для работы с комплексными числами в Ruby. Большая ее часть не требует пояснений.

Для создания комплексного числа применяется следующая несколько необычная нотация:

```
z = Complex(3,5)      # 3+5i
```

Необычно в ней то, что имя метода совпадает с именем класса. В данном случае наличие скобок указывает на то, что это вызов метода, а не ссылка на константу. Вообще говоря, имена методов не похожи на константы, и я не рекомендую начинать имена методов с прописной буквы, разве что в подобных специальных случаях. (Отметим, что имеются также методы `Integer` и `Float`; вообще, имена, начинающиеся с прописной буквы, зарезервированы для методов, которые выполняют преобразование данных и аналогичные действия.)

Метод `im` преобразует вещественное число в мнимое (по существу, умножая его на i). Поэтому представлять комплексные числа можно и с помощью более привычной нотации:

```
a = 3.im              # 3i
b = 5 - 2.im          # 5-2i
```

Если вас больше интересуют полярные координаты, то можно обратиться к методу `polar`:

```
z = Complex.polar(5,Math::PI/2.0)  # Радиус, угол.
```

В классе `Complex` имеется также константа `I`, которая представляет число i — квадратный корень из минус единицы:

```
z1 = Complex(3,5)
z2 = 3 + 5*Complex::I      # z2 == z1
```

После загрузки библиотеки `complex` некоторые стандартные математические функции изменяют свое поведение. Тригонометрические функции — `sin`, `sinh`, `tanh` (а также некоторые другие, например, `exp` и `log`) начинают принимать еще и комплексные аргументы. Некоторые функции, например `sqrt`, даже возвращают комплексные числа в качестве результата.

```
x = Math.sqrt(Complex(3,5))  # Приблизленно Complex(2.1013, 1.1897)
y = Math.sqrt(-1)            # Complex(0,1)
```

Дополнительную информацию ищите в любой полной документации, в частности на сайте ruby-doc.org.

5.12. Библиотека `mathn`

В программах, выполняющих большой объем математических вычислений, очень пригодится замечательная библиотека `mathn`, которую написал Кейдзу Исидзука (Keiju Ishitsuka). В ней есть целый ряд удобных методов и классов; кроме того, она унифицирует все классы Ruby для работы с числами так, что они начинают хорошо работать совместно.

Простейший способ воспользоваться этой библиотекой – включить ее с помощью директивы `require` и забыть. Поскольку она сама включает библиотеки `complex`, `rational` и `matrix` (в таком порядке), то вы можете этого не делать.

В общем случае библиотека `mathn` пытается вернуть «разумные» результаты вычислений. Например, при извлечении квадратного корня из `Rational` будет возвращен новый объект `Rational`, если это возможно; в противном случае `Float`. В таблице 5.1 приведены некоторые последствия загрузки этой библиотеки.

Таблица 5.1. Результаты вычислений в случае отсутствия и наличия библиотеки `mathn`

Выражение	Без <code>mathn</code>	С <code>mathn</code>
<code>Math.sqrt(Rational(9,16))</code>	0.75	<code>Rational(3,4)</code>
<code>1/2</code>	0	<code>Rational(1,2)</code>
<code>Matrix.identity(3)/3</code>	<code>Matrix[[[0,0,0], [0,0,0],[0,0,0]]]</code>	<code>Matrix[[[1/3,0,0], [0,1/3,0],[0,0,1/3]]]</code>
<code>Math.sqrt(64/25)</code>	1.4142...	<code>Rational(8,5)</code>
<code>Rational(1,10).inspect</code>	<code>Rational(1,10)</code>	<code>1/10</code>

Библиотека `mathn` добавляет методы `**` и `power2` в класс `Rational`. Она изменяет поведение метода `Math.sqrt` и добавляет метод `Math.rsqrt`, умеющий работать с рациональными числами.

Дополнительная информация приводится в разделах 5.13 и 5.14.

5.13. Разложение на простые множители, вычисление НОД и НОК

В библиотеке `mathn` определены также некоторые новые методы в классе `Integer`. Так, метод `gcd2` служит для нахождения наибольшего общего делителя (НОД) объекта, от имени которого он вызван, и другого числа.

```
n = 36.gcd2(120) # 12
k = 237.gcd2(79) # 79
```

Метод `prime_division` выполняет разложение на простые множители. Результат возвращается в виде массива массивов, в котором каждый вложенный массив содержит простое число и показатель степени, с которым оно входит в произведение:

```
factors = 126.prime_division # [[2,1], [3,2], [7,1]]
# То есть 2**1 * 3**2 * 7**1
```

Имеется также метод класса `Integer.from_prime_division`, который восстанавливает исходное число из его сомножителей. Это именно метод класса, потому что выступает в роли «конструктора» целого числа.

```
factors = [[2,1],[3,1],[7,1]]
num = Integer.from_prime_division(factors) # 42
```

Ниже показано, как разложение на простые множители можно использовать для отыскания наименьшего общего кратного (НОК) двух чисел:

```
require 'mathn'

class Integer
  def lcm(other)
    pf1 = self.prime_division.flatten
    pf2 = other.prime_division.flatten
    h1 = Hash[*pf1]
    h2 = Hash[*pf2]
    hash = h2.merge(h1) {|key,old,new| [old,new].max }
    Integer.from_prime_division(hash.to_a)
  end
end

p 15.lcm(150) # 150
p 2.lcm(3)    # 6
p 4.lcm(12)   # 12
p 200.lcm(30) # 600
```

5.14. Простые числа

В библиотеке `mathn` есть класс для порождения простых чисел. Итератор `each` возвращает последовательные простые числа в бесконечном цикле. Метод `succ` порождает следующее простое число. Вот, например, два способа получить первые 100 простых чисел:

```
require 'mathn'

list = []
gen = Prime.new
gen.each do |prime|
  list << prime
  break if list.size == 100
end

# или:

list = []
gen = Prime.new
100.times { list << gen.succ }
```


В следующем фрагменте проверяется, является ли данное число простым. Отметим, что если число велико, а машина медленная, то на выполнение может уйти заметное время:

```
require 'mathn'

class Integer
  def prime?
    max = Math.sqrt(self).ceil
    max -= 1 if max % 2 == 0
    pgen = Prime.new
    pgen.each do |factor|
      return false if self % factor == 0
      return true if factor > max
    end
  end
end

31.prime?           # true
237.prime?          # false
1500450271.prime?  # true
```

5.15. Явные и неявные преобразования чисел

Программисты, только начинающие изучать Ruby, часто удивляются, зачем нужны два метода `to_i` и `to_int` (и аналогичные им `to_f` и `to_float`). В общем случае метод с коротким именем применяется для явных преобразований, а метод с длинным именем — для неявных.

Что это означает? Во-первых, в большинстве классов определены явные конверторы, но нет неявных. Насколько мне известно, методы `to_int` и `to_float` не определены ни в одном из системных классов.

Во-вторых, в своих собственных классах вы, скорее всего, будете определять неявные конверторы, но не станете вызывать их вручную (если только не заняты написанием «клиентского» кода или библиотеки, которая пытается не конфликтовать с внешним миром).

Следующий пример, конечно, надуманный. В нем определен класс `MyClass`, который возвращает константы из методов `to_i` и `to_int`. Такое поведение лишено смысла, зато иллюстрирует идею:

```
class MyClass

  def to_i
    3
  end

  def to_int
    5
  end

end
```

Желая явно преобразовать объект класса `MyClass` в целое число, мы вызовем метод `to_i`:

```
m = MyClass.new
x = m.to_i      # 3
```

Но при передаче объекта `myClass` какой-нибудь функции, ожидающей целое число, будет неявно вызван метод `to_int`. Предположим, к примеру, что мы хотим создать массив с известным начальным числом элементов. Метод `Array.new` может принять целое, но что если вместо этого ему будет передан объект `myClass`?

```
m = MyClass.new
a = Array.new(m)  # [nil, nil, nil, nil, nil]
```

Как видите, метод `new` оказался достаточно «умным», чтобы вызвать `to_int` и затем создать массив из пяти элементов.

Дополнительную информацию о поведении в другом контексте (строковом) вы найдете в разделе 2.16. См. также раздел 5.16.

5.16. Приведение числовых значений

Приведение можно считать еще одним видом неявного преобразования. Если некоторому методу (например, `+`) передается аргумент, которого он не понимает, он пытается привести объект, от имени которого вызван, и аргумент к совместимым типам, а затем сложить их. Принцип использования метода `coerce` в вашем собственном классе понятен из следующего примера:

```
class MyNumberSystem

  def +(other)
    if other.kind_of?(MyNumberSystem)
      result = some_calculation_between_self_and_other
      MyNumberSystem.new(result)
    else
      n1, n2 = other.coerce(self)
      n1 + n2
    end
  end
end
```

Метод `coerce` возвращает массив из двух элементов, содержащий аргумент и вызывающий объект, приведенные к совместимым типам.

В данном примере мы полагаемся на то, что приведение выполнит тип аргумента. Но если мы хотим быть законопослушными гражданами, то должны реализовать приведение в своем классе, сделав его пригодным для работы с другими типами чисел. Для этого нужно знать, с какими типами мы в состоянии работать непосредственно, и при необходимости выполнять приведение к одному из этих типов. Если мы не можем сделать это самостоятельно, то должны обратиться за помощью к родительскому классу.

```
def coerce(other)
  if other.kind_of?(Float)
```

```

    return other, self.to_f
  elsif other.kind_of?(Integer)
    return other, self.to_i
  else
    super
  end
end

```

Разумеется, это будет работать только, если наш объект реализует методы `to_i` и `to_f`.

Метод `coerce` можно применить для реализации автоматического преобразования строк в числа, как в языке Perl:

```

class String

  def coerce(n)
    if self['.']
      [n, Float(self)]
    else
      [n, Integer(self)]
    end
  end
end

```

```

x = 1 + "23"      # 24
y = 23 * "1.23"   # 28.29

```

Мы не настаиваем на таком решении. Но рекомендуем реализовывать `coerce` при создании любого класса для работы с числовыми данными.

5.17. Поразрядные операции над числами

Иногда требуется работать с двоичным представлением объекта `Fixnum`. На прикладном уровне такая необходимость возникает нечасто, но все-таки возникает.

Ruby обладает всеми средствами для таких операций. Для удобства числовые константы можно записывать в двоичном, восьмеричном или шестнадцатеричном виде. Поразрядным операциям И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ соответствуют операторы `&`, `|`, `^` и `~`.

```

x = 0377      # Восьмеричное      (десятичное 255)
y = 0b00100110 # Двоичное          (десятичное 38)
z = 0xBEEF    # Шестнадцатеричное (десятичное 48879)

a = x | z      # 48895 (поразрядное ИЛИ)
b = x & z      # 239 (поразрядное И)
c = x ^ z      # 48656 (поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ)
d = ~ y        # -39 (отрицание или дополнение до 1)

```

Метод экземпляра `size` позволяет узнать размер слова для той машины, на которой выполняется программа.

```

size      # Для конкретной машины возвращает 4.

```

Имеются операторы сдвига влево и вправо (<< и >> соответственно). Это логические операторы сдвига, они не затрагивают знаковый бит (хотя оператор >> расширяет его).

```
x = 8
y = -8
```

```
a = x >> 2      # 2
b = y >> 2      # -2
c = x << 2      # 32
d = y << 2      # -32
```

Конечно, если сдвиг настолько велик, что дает нулевое значение, то знаковый бит теряется, поскольку -0 и 0 – одно и то же.

Квадратные скобки позволяют трактовать числа как битовые массивы. Бит с номером 0 всегда является младшим, вне зависимости от порядка битов в конкретной машинной архитектуре.

```
x = 5           # То же, что 0b0101
a = x[0]        # 1
b = x[1]        # 0
c = x[2]        # 1
d = x[3]        # 0
# И так далее  # 0
```

Присваивать новые значения отдельным битам с помощью такой нотации невозможно (поскольку Fixnum хранится как непосредственное значение, а не как ссылка на объект). Но можно имитировать это действие путем сдвига 1 влево на нужное число позиций с последующим выполнением операции ИЛИ или И.

```
# Выполнить присваивание x[3] = 1 нельзя,
# но можно поступить так:
x |= (1<<3)
# Выполнить присваивание x[4] = 0 нельзя,
# но можно поступить так:
x &= ~(1<<4)
```

5.18. Преобразование системы счисления

Ясно, что любое целое число можно представить в любой системе счисления, поскольку хранятся эти числа в двоичном виде. Мы знаем, что Ruby умеет работать с целыми константами, записанными в любой из четырех наиболее популярных систем. Следовательно, разговор о преобразовании системы счисления может вестись только применительно к числам, записанным в виде строк.

Вопрос о преобразовании строки в целое рассмотрен в разделе 2.24.

Для преобразования числа в строку проще всего воспользоваться методом `to_s`, которому можно еще передать основание системы счисления. По умолчанию оно равно 10, но в принципе может быть любым вплоть до 36 (когда задействованы все буквы латинского алфавита).

```
237.to_s(2)      # "11101101"
237.to_s(5)      # "1422"
```

```
237.to_s(8)      # "355"
237.to_s         # "237"
237.to_s(16)     # "ed"
237.to_s(30)     # "7r"
```

Другой способ – обратиться к методу % класса String:

```
hex = "%x" % 1234 # "4d2"
oct = "%o" % 1234 # "2322"
bin = "%b" % 1234 # "10011010010"
```

Метод sprintf тоже годится:

```
str = sprintf(str, "Nietzsche is %x\n", 57005)
# str теперь равно: "Nietzsche is dead\n"
```

Если нужно сразу же вывести преобразованное в строку значение, то подойдет и метод printf.

5.19. Извлечение кубических корней, корней четвертой степени и т. д.

В Ruby встроена функция извлечения квадратного корня (`Math.sqrt`), поскольку именно она применяется чаще всего. А если надо извлечь корень более высокой степени? Если вы еще не забыли математику, то эта задача не вызовет затруднений.

Можно, например, воспользоваться логарифмами. Напомним, что e в степени x – обратная функция к натуральному логарифму x и что умножение чисел эквивалентно сложению их логарифмов.

```
x = 531441
cuberoot = Math.exp(Math.log(x)/3.0) # 81.0
fourthroot = Math.exp(Math.log(x)/4.0) # 27.0
```

Но можно просто использовать дробные показатели степени (оператор возведения в степень принимает в качестве аргумента произвольное целое число или число с плавающей точкой).

```
include Math
y = 4096
cuberoot = y**(1.0/3.0) # 16.0
fourthroot = y**(1.0/4.0) # 8.0
fourthroot = sqrt(sqrt(y)) # 8.0 (то же самое)
twelfthroot = y**(1.0/12.0) # 2.0
```

Отметим, что во всех примерах мы пользовались при делении числами с плавающей точкой (чтобы избежать отбрасывания дробной части).

5.20. Определение порядка байтов

Интересно, что производители компьютеров никак не могут договориться, в каком порядке лучше хранить двоичные байты. Следует ли размещать старший бит по большему или по меньшему адресу? При передаче сообщения по проводам нужно сначала посылать старший или младший бит?

Хотите верить, хотите нет, но решение не произвольно. Существуют убедительные аргументы в пользу обеих точек зрения (обсуждать их здесь мы не будем).

Вот уже больше двадцати лет, как для описания противоположных позиций применяются термины «остроконечный» (little-endian) и «тупоконечный» (big-endian). Кажется, впервые их употребил Дэнни Коэн (Danny Cohen); см. его классическую статью "On Holy Wars and a Plea for Peace" (IEEE Computer, October 1981). Взяты они из романа Джонатана Свифта «Путешествия Гулливера».

Обычно нам безразличен порядок байтов в конкретной машинной архитектуре. Но как быть, если все-таки его нужно знать?

Можно воспользоваться показанным ниже методом. Он возвращает одну из строк LITTLE, BIG или OTHER. Решение основано на том факте, что директива l выполняет упаковку в машинном формате, а директива n распаковывает в сетевом порядке байтов (по определению тупоконечном).

```
def endianness
  num=0x12345678
  little = "78563412"
  big    = "12345678"
  native = [num].pack('l')
  netunpack = native.unpack('N')[0]
  str = "%8x" % netunpack
  case str
    when little
      "LITTLE"
    when big
      "BIG"
    else
      "OTHER"
  end
end

puts endianness      # В данном случае печатается "LITTLE"
```

Этот прием может оказаться удобным, если, например, вы работаете с двоичными данными (скажем, отсканированным изображением), импортированными из другой системы.

5.21. Численное вычисление определенного интеграла

.....
Я очень хорошо владею дифференциальным и интегральным исчислением...
У.С.Джильберт, «Пираты Пензанса», акт 1

Для приближенного вычисления определенного интеграла имеется проверенная временем техника. Любой студент, изучавший математический анализ, вспомнит, что она называется суммой Римана.

Приведенный ниже метод `integrate` принимает начальное и конечное значения зависимой переменной, а также приращение. Четвертый параметр (который на самом деле параметром не является) – это блок. В блоке должно вычисляться значение функции от переданной в него зависимой переменной (здесь слово «переменная» употребляется в математическом, а не программистском смысле). Необязательно отдельно определять функцию, которая вызывается в блоке, но для ясности мы это сделаем.

```
def integrate(x0, x1, dx=(x1-x0)/1000.0)
```

```
    x = x0
```

```
    sum = 0
```

```
    loop do
```

```
        y = yield(x)
```

```
        sum += dx * y
```

```
        x += dx
```

```
        break if x > x1
```

```
    end
```

```
    sum
```

```
end
```

```
def f(x)
```

```
    x**2
```

```
end
```

```
z = integrate(0.0,5.0) {|x| f(x) }
```

```
puts z, "\n"           # 41.7291875
```

Здесь мы опираемся на тот факт, что блок возвращает значение, которое может быть получено с помощью `yield`. Кроме того, сделаны некоторые допущения. Во-первых, мы предполагаем, что x_0 меньше x_1 (в противном случае получится бесконечный цикл). Читатель сам легко устранил подобные огрехи. Во-вторых, мы считаем, что функцию можно вычислить в любой точке заданной области. Если это не так, мы получим хаотическое поведение. (Впрочем, подобные функции все равно, как правило, не интегрируемы – по крайней мере, на указанном интервале. В качестве примера возьмите функцию $f(x) = x/(x-3)$ в точке $x=3$.)

Призвав на помощь полузабытые знания об интегральном исчислении, мы могли бы вычислить, что в данном случае результат равен примерно 41.666 (5 в кубе, поделенное на 3). Почему же ответ не так точен, как хотелось бы? Из-за выбранного размера приращения; чем меньше величина dx , тем точнее результат (ценой увеличения времени вычисления).

Напоследок отметим, что подобная методика более полезна для действительно сложных функций, а не таких простых, как $f(x) = x**2$.

5.22. Тригонометрия в градусах, радианах и градах

При измерении дуг математической, а заодно и «естественной» единицей измерения является *радиан*. По определению, угол в один радиан соответствует длине дуги, равной радиусу окружности. Немного поразмыслив, легко понять, что угол 2π радиан соответствует всей окружности.

Дуговой градус, которым мы пользуемся в повседневной жизни, – пережиток древневавилонской системы счисления по основанию 60: в ней окружность делится на 360 градусов. Менее известна псевдометрическая единица измерения *град*, определенная так, что прямой угол составляет 100 град (а вся окружность – 400 град).

При вычислении тригонометрических функций в языках программирования по умолчанию чаще всего используются радианы, и Ruby в этом отношении не исключение. Но мы покажем, как производить вычисления и в градусах, и в градах для тех читателей, которые по образованию не инженеры, а по происхождению не древние вавилоняне.

Поскольку число любых угловых единиц в окружности – константа, можно легко переходить от одних единиц к другим. Мы определим соответствующие константы и будем пользоваться ими в коде. Для удобства поместим их в модуль `math`.

```
module Math
```

```
RAD2DEG = 360.0/(2.0*PI)    # Радианы в градусы.
RAD2GRAD = 400.0/(2.0*PI)    # Радианы в грады.
```

end

Теперь можно определить и новые тригонометрические функции. Поскольку мы всегда преобразуем в радианы, то будем делить на определенные выше коэффициенты. Можно было бы поместить определения функций в тот же модуль `Math`, но мы этого делать не стали.

```
def sin_d(theta)
  Math.sin (theta/Math::RAD2DEG)
end
```

```
def sin_g(theta)
  Math.sin (theta/Math::RAD2GRAD)
end
```

Функции $\cos$ и $\tan$ можно было бы определить аналогично.

С функцией `atan2` дело обстоит несколько сложнее. Она принимает два аргумента (длины противолежащей и прилежащей сторон прямоугольного треугольника). Поэтому мы преобразуем результат, а не аргумент:

```
def atan2_d(y,x)
  Math.atan2(y,x)/Math::RAD2DEG
end
```

```
def atan2_g(y,x)
    Math.atan2(y,x)/Math::RAD2GRAD
end
```

5.23. Неэлементарная тригонометрия

В ранних версиях Ruby не было функций `arcsin` и `arccos`. Равно как и гиперболических функций `sinh`, `cosh` и `tanh`. Их определения были приведены в первом издании этой книги, но сейчас они являются стандартной частью модуля `Math`.

5.24. Вычисление логарифмов по произвольному основанию

Чаще всего мы пользуемся натуральными логарифмами (по основанию e , часто натуральный логарифм обозначается как $\ln$), иногда также десятичными (по основанию 10). Эти функции реализованы в методах `Math.log` и `Math.log10` соответственно.

В информатике, а в особенности в таких ее областях, как кодирование и теория информации, обычно применяются логарифмы по основанию 2. Например, так вычисляется минимальное число битов, необходимых для представления числа. Определим функцию с именем `log2`:

```
def log2(x)
  Math.log(x)/Math.log(2)
end
```

Ясно, что обратной к ней является функция 2^{**x} (как обращением $\ln x$ служит `Math::E**x` или `Math.exp(x)`).

Эта идея обобщается на любое основание. В том маловероятном случае, если вам понадобится логарифм по основанию 7, можно поступить так:

```
def log7(x)
  Math.log(x)/Math.log(7)
end
```

На практике знаменатель нужно вычислить один раз и сохранить в виде константы.

5.25. Вычисление среднего, медианы и моды набора данных

Пусть дан массив x , вычислим среднее значение по всем элементам массива. На самом деле есть три общеупотребительные разновидности среднего значения. Среднее арифметическое – это то, что мы называем средним в обыденной жизни. Среднее гармоническое – это число элементов, поделенное на сумму обратных к ним. И, наконец, среднее геометрическое – это корень n -ой степени из произведения n значений. Вот эти определения, воплощенные в коде:

```
def mean(x)
  sum=0
  x.each {|v| sum += v}
  sum/x.size
end

def hmean(x)
  sum=0
  x.each {|v| sum += (1.0/v)}
  x.size/sum
end
```

```
def gmean(x)
  prod=1.0
  x.each {|v| prod *= v}
  prod**(1.0/x.size)
end

data = [1.1, 2.3, 3.3, 1.2, 4.5, 2.1, 6.6]
```

```
am = mean(data)      # 3.014285714
hm = hmean(data)     # 2.101997946
gm = gmean(data)     # 2.508411474
```

Медианой набора данных называется значение, которое оказывается приблизительно в середине отсортированного набора (ниже приведен код для вычисления медианы). Примерно половина элементов набора меньше медианы, а другая половина – больше. Ясно, что такая статистика показательна не для всякого набора.

```
def median(x)
  sorted = x.sort
  mid = x.size/2
  sorted[mid]
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts median(data)    # 4
```

Мода набора данных – это наиболее часто встречающееся в нем значение. Если такое значение единственно, набор называется *унимодальным*, в противном случае – *мультимодальным*. Мультимодальные наборы более сложны, здесь мы их рассматривать не будем. Интересующийся читатель может обобщить и улучшить приведенный ниже код:

```
def mode(x)
  f = {}           # Таблица частот.
  fmax = 0         # Максимальная частота.
  m = nil          # Мода.
  x.each do |v|
    f[v] ||= 0
    f[v] += 1
    fmax, m = f[v], v if f[v] > fmax
  end
  return m
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts mode(data)    # 7
```

5.26. Дисперсия и стандартное отклонение

Дисперсия – это мера «разброса» значений из набора. (Здесь мы не различаем смещенные и несмещенные оценки.) Стандартное отклонение, которое обычно обозначается буквой σ , равно квадратному корню из дисперсии.

```
data = [2, 3, 2, 2, 3, 4, 5, 5, 4, 3, 4, 1, 2]
```

```
def variance(x)
  m = mean(x)
  sum = 0.0
  x.each {|v| sum += (v-m)**2 }
  sum/x.size
end
```

```
def sigma(x)
  Math.sqrt(variance(x))
end
```

```
puts variance(data)    # 1.461538462
puts sigma(data)       # 1.20894105
```

Отметим, что функция `variance` вызывает определенную выше функцию `mean`.

5.27. Вычисление коэффициента корреляции

Коэффициент корреляции – одна из самых простых и полезных статистических мер. Он измеряет «линейность» набора, состоящего из пар (x, y) , и изменяется от -1.0 (полная отрицательная корреляция) до $+1.0$ (полная положительная корреляция).

Для вычисления воспользуемся функциями `mean` и `sigma` (стандартное отклонение), которые были определены в разделах 5.25 и 5.26. О смысле этого показателя можно прочитать в любом учебнике по математической статистике.

В следующем коде предполагается, что есть два массива чисел одинакового размера:

```
def correlate(x,y)
  sum = 0.0
  x.each_index do |i|
    sum += x[i]*y[i]
  end
  xymean = sum/x.size.to_f
  xmean = mean(x)
  ymean = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean - (xmean*ymean)) / (sx*sy)
end
```

```
a = [3, 6, 9, 12, 15, 18, 21]
b = [1.1, 2.1, 3.4, 4.8, 5.6]
c = [1.9, 1.0, 3.9, 3.1, 6.9]
```

```
c1 = correlate(a,a)          # 1.0
c2 = correlate(a,a.reverse)  # -1.0
c3 = correlate(b,c)          # 0.8221970228
```

Приведенная ниже версия отличается лишь тем, что работает с одним массивом, каждый элемент которого – массив, содержащий пару (x, y):

```
def correlate2(v)
  sum = 0.0
  v.each do |a|
    sum += a[0]*a[1]
  end
  xymean = sum/v.size.to_f
  x = v.collect {|a| a[0]}
  y = v.collect {|a| a[1]}
  xmean = mean(x)
  ymean = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean - (xmean*ymean)) / (sx*sy)
end

d = [[1,6.1], [2.1,3.1], [3.9,5.0], [4.8,6.2]]

c4 = correlate2(d)    # 0.2277822492
```

И, наконец, в последнем варианте предполагается, что пары (x, y) хранятся в хэше. Код основан на предыдущем примере:

```
def correlate_h(h)
  correlate2(h.to_a)
end

e = { 1 => 6.1, 2.1 => 3.1, 3.9 => 5.0, 4.8 => 6.2}

c5 = correlate_h(e)   # 0.2277822492
```

5.28. Генерирование случайных чисел

Если вас устраивают псевдослучайные числа, вам повезло. Именно они предоставляются в большинстве языков, включая и Ruby.

Метод `rand` из модуля `kernel` возвращает псевдослучайное число x с плавающей точкой, отвечающее условиям $x \geq 0.0$ и $x < 1.0$. Например (вы можете получить совсем другое число):

```
a = rand          # 0.6279091137
```

Если при вызове задается целочисленный параметр `max`, то возвращается целое число из диапазона $0 \dots \text{max}$ (верхняя граница не включена). Например:

```
n = rand(10)      # 7
```

Чтобы «затравить» генератор случайных чисел (задать начальное значение – `seed`), применяется метод `srand` из модуля `kernel`, который принимает один числовой параметр. Если не передавать никакого значения, то метод `srand` самостоятельно изготовит затравку, учитывая (среди прочего) текущее время. Если же параметр передан, то именно он и становится затравкой. Это бывает полезно при

тестировании, когда для воспроизводимости результатов многократно вызываемая программа должна получать одну и ту же последовательность псевдослучайных чисел.

```
srand(5)
i, j, k = rand(100), rand(100), rand(100)
# 26, 45, 56
```

```
srand(5)
l, m, n = rand(100), rand(100), rand(100)
# 26, 45, 56
```

5.29. Кэширование функций с помощью метода memoize

Пусть имеется вычислительно сложная математическая функция, которую нужно многократно вызывать по ходу работы программы. Если быстроедействие критично и при этом можно пожертвовать небольшим количеством памяти, то имеет смысл сохранить результаты вычисления функции в таблице и обращаться к ней во время выполнения. (Тут неявно предполагается, что функция будет часто вызываться с одними и теми же параметрами, то есть получается, что мы «выбрасываем» результат дорогостоящего вычисления и снова повторяем его позже.) Такая техника иногда называется запоминанием (memoizing), отсюда и название библиотеки memoize.

Эта библиотека не входит в стандартный дистрибутив, поэтому придется установить ее вручную.

В следующем примере демонстрируется сложная функция zeta. Она применяется при решении одной задачи из области популяционной генетики, но вдаваться в объяснения мы не станем.

```
require 'memoize'
include Memoize

def zeta(x,y,z)
  lim = 0.0001
  gen = 0
  loop do
    gen += 1
    p,q = x + y/2.0, z + y/2.0
    x1, y1, z1 = p*p*1.0, 2*p*q*1.0, q*q*0.9
    sum = x1 + y1 + z1
    x1 /= sum
    y1 /= sum
    z1 /= sum
    delta = [[x1,x],[y1,y],[z1,z]]
    break if delta.all? {|a,b| (a-b).abs < lim}
    x,y,z = x1,y1,z1
  end
  gen
end
```

```
g1 = zeta(0.8,0.1,0.1)
```

```
memoize(:zeta) # Сохранить таблицу в памяти.
```

```
g2 = zeta(0.8,0.1,0.1)
```

```
memoize(:zeta,"z.cache") # Сохранить таблицу на диске.
```

```
g3 = zeta(0.8,0.1,0.1)
```

Обратите внимание, что можно задать имя файла. Это может несколько замедлить работу, зато экономится память, и таким образом мы можем сохранить запомненные результаты и воспользоваться ими при следующих вызовах программы.

В ходе неформального тестирования мы вызывали функцию 50000 раз в цикле. Оказалось, что `g2` вычисляется примерно в 1100 раз быстрее, чем `g1`, а `g3` — примерно в 700 раз. На вашей машине может получиться иной результат.

Отметим еще, что библиотека `memoize` предназначена не только для математических функций. Ее можно использовать для запоминания результатов работы любого вычислительно сложного метода.

5.30. Заключение

В этой главе были рассмотрены различные представления чисел, в том числе целых (в разных системах счисления) и с плавающей точкой. Мы видели, какие трудности возникают при работе с числами с плавающей точкой и как можно частично обойти эти трудности, применяя рациональные числа. Мы познакомились с явными и неявными преобразованиями, а также с приведениями типов.

Также мы изучили разнообразные способы манипулирования числами, векторами и матрицами. Был приведен обзор стандартных библиотек, полезных для численного анализа, в частности библиотеки `mathn`.

Пойдем дальше. В следующей главе мы обсудим два очень характерных для Ruby типа данных: символы и диапазоны.

Глава 6. Символы и диапазоны

.....
Я слышу и забываю. Я вижу и запоминаю. Я делаю и понимаю.
Конфуций

Символы и диапазоны – объекты, весьма характерные для языка Ruby. Они рассматриваются в одной главе не потому, что тесно связаны между собой, а потому, что сказать о них можно не так уж много.

Концепцию символа в Ruby понять непросто. Они напоминают «атомы» в языке Lisp. Вместо того чтобы давать длинное и сложное определение, я расскажу о том, что можно делать с символами и как они применяются. В конце концов, на вопрос «что такое число» можно дать очень глубокомысленный ответ, но нам нужно всего лишь знать, как манипулировать числами.

Диапазоны проще. Это всего лишь представление множества, заданного конечными точками. Аналогичные конструкции есть в языках Pascal, PHP и даже SQL.

Познакомимся с символами и диапазонами поближе, чтобы понять, как они практически используются в программах на Ruby.

6.1. Символы

Символ в Ruby – это экземпляр класса `Symbol`. Синтаксически он обычно обозначается двоеточием (:), за которым следует идентификатор.

Символ похож на строку, он тоже соответствует последовательности символов. Отличие от строки состоит в том, что у каждого символа есть только один экземпляр (как и в случае с объектами `Fixnum`). Следовательно, имеет место проблема потребления памяти или производительности, о которой нужно помнить. Например, в нижеприведенном коде строка `"foo"` представлена в памяти тремя различными объектами, а символ `:foo` – одним, на который есть несколько ссылок:

```
array = ["foo", "foo", "foo", :foo, :foo, :foo]
```

Некоторых смущает двоеточие перед именем символа. Не волнуйтесь, это всего лишь синтаксическое соглашение. У строк, массивов и хэшей есть начальный и конечный ограничители, а у символов – только начальный. Считайте, что это унарный, а не бинарный ограничитель. На первый взгляд синтаксис кажется странным, но ничего таинственного в нем нет.

Стоит отметить, что в старых версиях Ruby (до 1.6) символьные константы не были полноценными объектами, поскольку преобразовывались в `Fixnum` и в таком виде хранились. Внутреннее представление осталось таким же; символу ставится в соответствие число, и хранится он как непосредственное значение. Само число можно получить, вызвав метод `to_i`, но в этом редко возникает необходимость.

По словам Джима Вайриха, символ – это «объект, у которого есть имя». Остин Зиглер предпочитает говорить об «объекте, который сам является именем». Как бы то ни было, существует взаимно однозначное соответствие между символами и именами. К чему можно применить имена? Например, к переменным, методам и произвольным константам.

Типичное применение символов – для представления имени переменной или метода. Например, чтобы добавить в класс атрибут, допускающий чтение и изменение, можно поступить следующим образом:

```
class SomeClass
  attr_accessor :whatever
end
```

То же самое можно выразить иначе:

```
class SomeClass
  def whatever
    @whatever
  end
  def whatever=(val)
    @whatever = val
  end
end
```

Другими словами, символ `:whatever` говорит методу `attr_accessor`, что методам чтения и установки (а равно и самой переменной экземпляра) следует присвоить имена, определяемые указанным символом.

Но почему не воспользоваться просто строкой? Вообще-то можно. Многие, даже большинство системных методов, ожидающих символ в качестве параметра, соглашаются и на строку.

```
attr_reader :alpha
attr_reader "beta" # Так тоже можно.
```

На самом деле символ «похож» на строку в том смысле, что ему соответствует последовательность символов. Поэтому некоторые говорят, что «символ – это просто неизменяемая строка». Но класс `Symbol` не наследует классу `String`, а типичные операции над строками необязательно применимы к символам.

Также неправильно думать, что символы напрямую соответствуют идентификаторам. Из-за этого непонимания некоторые говорят о «таблице символов» (как если бы речь шла об ассемблированном объектном коде). В действительности это представление бессмысленно; хотя символы и хранятся в какой-то внутренней таблице (а как же иначе?), Ruby не дает к ней доступа, поэтому программистам все равно, существует она или нет.

Более того, символы даже не всегда выглядят как идентификаторы. Обычно это так, что бы под этим ни понимать, но символ может содержать и знаки препинания, если заключен в кавычки. Все показанные ниже символы допустимы:

```
sym1 = : "This is a symbol"
sym2 = : "This is, too!"
sym3 = : ")(*&^%" # И даже такой.
```


Можно даже использовать символы для определения переменных и методов экземпляра, но тогда для ссылки на них пришлось бы применять такие методы, как `send` и `instance_variable_get`. Вообще говоря, такая практика не рекомендуется.

6.1.1. Символы как перечисления

В языке Pascal и в поздних версиях C есть понятие перечисляемого типа. В Ruby ничего подобного быть не может, ведь никакого контроля типов не производится. Но символы часто используются как мнемонические имена; стороны света можно было бы представить как `:north`, `:south`, `:east` и `:west`.

Быть может, немного понятнее хранить их в виде констант:

```
North, South, East, West = :north, :south, :east, :west
```

Если бы это были строки, а не символы, то определение их в виде констант могло бы сэкономить память, но каждый символ все равно существует в объектном пространстве в единственном экземпляре. (Символы, подобно объектам `Fixnum`, хранятся как непосредственные значения.)

6.1.2. Символы как метазначения

Мы нередко пользуемся исключениями, чтобы уйти от кодов возврата. Но никто не мешает возвращать коды ошибки, если вам так хочется. К тому же в Ruby метод может возвращать более одного значения.

В таком механизме часто возникает необходимость. Когда-то символ NUL кода ASCII вообще не считался символом. В языке C есть понятие нулевого указателя (`NULL`), в Pascal есть указатель `nil`, в SQL `NULL` означает отсутствие какого бы то ни было значения. В Ruby, конечно, тоже есть свой `nil`.

Проблема в том, что такие метазначения часто путают с действительными значениями. В наши дни все считают NUL настоящим символом кода ASCII. И в Ruby нельзя сказать, что `nil` не является объектом; его можно хранить, над ним можно выполнять какие-то операции. Поэтому не вполне понятно, как интерпретировать ситуацию, когда `hash[key]` возвращает `nil`: то ли указанный ключ вообще не найден, то ли с ним ассоциировано значение `nil`.

Идея в том, что иногда символы могут выступать в роли подходящих метазначений. Представьте метод, который получает строку из сети (возможно, по протоколу HTTP или иным способом). При желании можно было бы вернуть нестроковое значение как индикатор исключительной ситуации.

```
str = get_string
case str
  when String
    # Нормальная обработка.
  when :eof
    # Конец файла, закрытие сокета и т. п.
  when :error
    # Ошибка сети или ввода/вывода.
  when :timeout
    # Ответ не получен вовремя.
end
```

Можно ли сказать, что это «лучше», чем механизм исключений? Необязательно. Но такую методику стоит иметь в виду, особенно когда приходится обрабатывать «границные случаи», которые не считаются ошибками.

6.1.3. Символы, переменные и методы

Наверное, чаще всего символы применяются для определения атрибутов класса:

```
class MyClass
  attr_reader :alpha, :beta
  attr_writer :gamma, :delta
  attr_accessor :epsilon
  # ...
end
```

Имейте в виду, что в этом фрагменте на самом деле исполняется некий код. Например, `attr_accessor` использует имя символа для определения имени переменной экземпляра, а также методов для ее чтения и изменения. Это не означает, что всегда имеется точное соответствие между символом и именем переменной экземпляра. Например, обращаясь к методу `instance_variable_set`, мы должны задать точное имя переменной, включая и знак `@`:

```
sym1 = :@foo
sym2 = :foo
instance_variable_set(sym1, "str") # Правильно.
instance_variable_set(sym2, "str") # Ошибка.
```

Короче говоря, символ, передаваемый методам из семейства `attr`, — всего лишь аргумент, а сами эти методы создают требуемые переменные и методы экземпляра, основываясь на значении символа. (В конец имени метода изменения добавляется знак равенства, а в начало имени переменной экземпляра — знак `@`.) Бывают также случаи, когда символ должен точно соответствовать идентификатору, на который ссылается.

В большинстве случаев (если не во всех!) методы, ожидающие на входе символ, принимают также строку. Обратное не всегда верно.

6.1.4. Преобразование строки в символ и обратно

Строки и символы можно преобразовывать друг в друга с помощью методов `to_str` и `to_sym`:

```
a = "foobar"
b = :foobar
a == b.to_str # true
b == a.to_sym # true
```

Для метапрограммирования иногда бывает полезен такой метод:

```
class Symbol
  def +(other)
    (self.to_s + other.to_s).to_sym
  end
end
```

Он позволяет конкатенировать символы (или дописывать строку в конец символа). Ниже приведен пример использования; мы принимаем на входе символ и пытаемся определить, представляет ли он какой-нибудь метод доступа (то есть существует ли метод чтения или установки атрибута с таким именем):

```
class Object
  def accessor?(sym)
    return (self.respond_to?(sym) and self.respond_to?(sym+"="))
  end
end
```

Упомяну также о более изощренном способе применения символов. Иногда при выполнении операции `map` нужно указать сложный блок. Однако во многих случаях мы просто вызываем некоторый метод для каждого элемента массива или набора:

```
list = words.map {|x| x.capitalize }
```

Не кажется ли вам, что для такой простой задачи слишком много знаков препинания? Давайте вместо этого определим метод `to_proc` в классе `Symbol`. Он будет приводить любой символ к типу объекта `proc`. Но какой именно объект `proc` следует вернуть? Очевидно, соответствующий самому символу в контексте объекта; иными словами, такой, который пошлет сам символ в виде сообщения объекту.

```
def to_proc
  proc {|obj, *args| obj.send(self, *args) }
end
```

Кстати, этот код заимствован из проекта Гэвина Синклера (Gavin Sinclair) «Расширения Ruby». Имея такой метод, мы можем следующим образом переписать первоначальный код:

```
list = words.map(&:capitalize)
```

Стоит потратить немного времени и разобраться, как это работает. Метод `map` обычно принимает только блок (никаких других параметров). Наличие знака `&` (амперсанд) позволяет передать объект `proc` вместо явно указанного блока. Поскольку мы применяем амперсанд к объекту, не являющемуся `proc`, то интерпретатор пытается вызвать метод `to_proc` этого объекта. Получающийся в результате объект `proc` подставляется вместо явного блока, чтобы метод `map` вызывал его для каждого элемента массива. А зачем передавать `self` в виде сообщения элементу массива? Затем, что объект `proc` является замыканием и, следовательно, помнит контекст, в котором был создан. А в момент создания `self` был ссылкой на символ, для которого вызывался метод `to_proc`.

6.2. Диапазоны

Понятие диапазона интуитивно понятно, но и у него имеются некоторые неочевидные особенности и способы применения. Одним из самых простых является числовой диапазон:

```
digits = 0..9
scale1 = 0..10
scale2 = 0...10
```

Оператор `..` включает конечную точку, а оператор `...` не включает. (Если это для вас неочевидно, просто запомните.) Таким образом, диапазоны `digits` и `scale2` из предыдущего примера одинаковы.

Но диапазоны могут состоять не только из целых чисел – более того, не только из чисел. Началом и концом диапазона в Ruby может быть любой объект. Однако, как мы вскоре увидим, не все диапазоны осмыслены или полезны.

Основные операции над диапазоном – обход, преобразование в массив, а также выяснение, попадает ли некоторый объект в данный диапазон. Рассмотрим разнообразные варианты этих и других операций.

6.2.1. Открытые и замкнутые диапазоны

Диапазон называется *замкнутым*, если включает конечную точку, и *открытым* – в противном случае:

```
r1 = 3..6           # Замкнутый.
r2 = 3...6          # Открытый.
a1 = r1.to_a        # [3,4,5,6]
a2 = r2.to_a        # [3,4,5]
```

Нельзя сконструировать диапазон, который не включал бы начальную точку. Можно считать это ограничением языка.

6.2.2. Нахождение границ диапазона

Методы `first` и `last` возвращают соответственно левую и правую границу диапазона. У них есть синонимы `begin` и `end` (это еще и ключевые слова, но интерпретируются как вызов метода, если явно указан вызывающий объект).

```
r1 = 3..6
r2 = 3...6
r1a, r1b = r1.first, r1.last  # 3, 6
r1c, r1d = r1.begin, r1.end   # 3, 6
r2a, r2b = r2.begin, r2.end   # 3, 6
```

Метод `exclude_end?` сообщает, включена ли в диапазон конечная точка:

```
r1.exclude_end?  # false
r2.exclude_end?  # true
```

6.2.3. Обход диапазона

Обычно диапазон можно обойти. Для этого класс, которому принадлежат границы диапазона, должен предоставлять осмысленный метод `succ` (следующий).

```
(3..6).each {|x| puts x }      # Печатаются четыре строки
                               # (скобки обязательны).
```

Пока все хорошо. И тем не менее будьте очень осторожны при работе со строковыми диапазонами! В классе `String` имеется метод `succ`, но он не слишком полезен. Пользоваться этой возможностью следует только при строго контролируемых условиях, поскольку метод `succ` определен не вполне корректно. (В определении используется, скорее, «интуитивно очевидный», нежели лексикографический порядок, поэтому существуют строки, для которых «следующая» не имеет смысла.)

```

r1 = "7".."9"
r2 = "7".."10"
r1.each {|x| puts x } # Печатаются три строки.
r2.each {|x| puts x } # Ничего не печатается!

```

Предыдущие примеры похожи, но ведут себя по-разному. Отчасти причина в том, что границы второго диапазона – строки разной длины. Мы ожидаем, что в диапазон входят строки "7", "8", "9" и "10", но что происходит на самом деле?

При обходе диапазона `r2` мы начинаем со значения "7" и входим в цикл, который завершается, когда текущее значение окажется больше правой границы. Но ведь "7" и "10" – не числа, а строки, и сравниваются они как строки, то есть лексикографически. Поэтому левая граница оказывается больше правой, и цикл не выполняется ни разу.

А что сказать по поводу диапазонов чисел с плавающей точкой? Такой диапазон можно сконструировать и, конечно, проверить, попадает ли в него конкретное число. Это полезно. Но обойти такой диапазон нельзя, так как метод `succ` отсутствует.

```

fr = 2.0..2.2
fr.each {|x| puts x } # Ошибка!

```

Почему для чисел с плавающей точкой нет метода `succ`? Теоретически можно было бы увеличивать число на некоторое приращение. Но величина такого приращения сильно зависела бы от конкретной машины, при этом даже для обхода «небольшого» диапазона понадобилось бы гигантское число итераций, а полезность такой операции весьма сомнительна.

6.2.4. Проверка принадлежности диапазону

Зачем нужен диапазон, если нельзя проверить, принадлежит ли ему конкретный объект? Эта задача легко решается с помощью метода `include?`:

```

r1 = 23456..34567
x = 14142
y = 31416
r1.include?(x)      # false
r1.include?(y)      # true

```

У этого метода есть также синоним `member?`.

А как он работает? Как интерпретатор определяет, принадлежит ли объект диапазону? Просто путем сравнения с границами (поэтому проверка принадлежности диапазону возможна лишь, если определен осмысленный оператор `<=>`).

Следовательно, запись `(a..b).include?(x)` эквивалентна `x >= a and x <= b`.

Еще раз предупреждаем: будьте осторожны со строковыми диапазонами!

```

s1 = "2".."5"
str = "28"
s1.include?(str) # true (неправильно!)

```

6.2.5. Преобразование в массив

Когда диапазон преобразуется в массив, интерпретатор последовательно вызывает метод `succ`, пока не будет достигнута правая граница, и помещает каждый элемент диапазона в возвращаемый массив:

```
r = 3..12
arr = r.to_a      # [3,4,5,6,7,8,9,10,11,12]
```

Ясно, что для диапазонов чисел типа Float такой подход не работает. Со строковыми диапазонами иногда будет работать, но лучше этого не делать, поскольку результат не всегда очевиден или осмыслен.

6.2.6. Обратные диапазоны

Имеет ли смысл говорить об обратном диапазоне? И да, и нет. Следующий диапазон допустим:

```
r = 6..3
x = r.begin      # 6
y = r.end        # 3
flag = r.end_excluded? # false
```

Как видите, мы можем определить обе границы и узнать, что правая граница включена. Но этим перечень возможных операций практически исчерпывается.

```
arr = r.to_a      # []
r.each {|x| p x } # Ни одной итерации.
y = 5
r.include?(y)     # false (для любого значения y)
```

Означает ли это, что обратные диапазоны всегда бесполезны? Вовсе нет. В некоторых случаях разумно инкапсулировать границы в один объект.

На самом деле массивы и строки часто принимают обратные диапазоны в качестве индексов, поскольку индексация для них начинается с 0, если отсчитывать от левой границы, и с -1 – если от правой. Поэтому допустимы такие выражения:

```
string = "flowery"
str1   = string[0..-2] # "flower"
str2   = string[1..-2] # "lower"
str3   = string[-5..-3] # "owe" (по существу, прямой диапазон)
```

6.2.7. Оператор переключения

Диапазон в составе условия обрабатывается особым образом. В этом случае .. называется *оператором переключения* (flip-flop operator), поскольку это, по существу, переключатель, который сохраняет свое состояние.

Такой прием, позаимствованный из языка Perl, бывает полезен. Но понять, как он работает, довольно трудно.

Представьте себе исходный текст программы на Ruby, в который встроена документация, ограниченная маркерами =begin и =end. Как бы вы подошли к задаче отыскания и вывода этих и только этих фрагментов? (Состояние переключается между «внутри раздела» и «вне раздела», отсюда и понятие переключения.) Решение, хотя интуитивно и не очевидное, дает следующий код:

```
loop do
  break if eof?
  line = gets
  puts line if (line =~ /begin/) .. (line =~ /end/)
end
```

«Волшебство» объясняется принципом работы оператора переключения.

Во-первых, надо осознать, что «диапазон» сохраняет свое состояние, хотя оно и скрыто. Когда становится истинным условие, заданное в качестве левой границы, сам диапазон принимает значение `true`. Он сохраняет это состояние до тех пор, пока не станет истинным условие на правой границе, и в этот момент состояние переключается в `false`.

Такое поведение полезно во многих случаях, в частности для разбора HTML-документов или конфигурационных файлов, разбитых на разделы, выбора диапазонов элементов из списков и т. д.

Но лично мне такой синтаксис не нравится. Недовольны им и многие другие, включая и самого Маца. Возможно, в будущем эта возможность будет исключена из Ruby. Однако я покажу удобный способ реализовать ту же функциональность по-другому.

Что меня не устраивает в операторе переключения? В контексте предыдущего примера рассмотрим строку, начинающуюся с маркера `=begin`. Напомним, что оператор `==` не возвращает `true` или `false`, как можно было бы ожидать; он возвращает начальную позицию найденного соответствия (`Fixnum`) или `nil`, если соответствие не найдено. Следовательно, при вычислении выражений для строк, попадающих и не попадающих в диапазон, мы получаем 0 и `nil` соответственно.

Однако при попытке сконструировать диапазон от 0 до `nil` возникает ошибка, поскольку такой диапазон не имеет смысла:

```
range = 0..nil      # Ошибка!
```

Далее, напомним, что в Ruby только `false` и `nil` дают значение «ложь» — все остальные объекты в логическом контексте вычисляются как «истина». А значит, следуя общей идеологии диапазон не должен вычисляться как «ложь».

```
puts "hello" if x..y
# Печатается "hello" для любого допустимого диапазона x..y.
```

Но предположим, что мы сохранили эти значения в переменных, а потом из них сконструировали диапазон. Все перестанет работать, так как проверка всегда дает `true`.

```
loop do
  break if eof?
  line = gets
  start = line =~ /begin/
  stop = line =~ /end/
  puts line if start..stop
end
```

А что если сам диапазон поместить в переменную? Тоже не получится — проверка снова дает `true`.

```
loop do
  break if eof?
  line = gets
  range = (line =~ /begin/) .. (line =~ /end/)
  puts line if range
end
```

Чтобы понять, в чем дело, нужно осознать, что весь диапазон (включая обе границы) вычисляется на каждой итерации цикла, но с учетом внутреннего состояния. Поэтому оператор переключения – вообще не настоящий диапазон. Тот факт, что он выглядит похожим на диапазон, хотя по сути таковым не является, многие считают «злом».

И наконец, задумаемся о границах в операторе переключения. Они вычисляются каждый раз, но результат вычисления нельзя сохранить в переменной и затем просто подставить ее. В некотором смысле граничные точки оказываются похожи на объекты `proc`. Это не значения, а исполняемый код. Тот факт, что нечто, выглядящее как обычное выражение, на самом деле представляет собой `proc`, тоже не вызывает восторга.

И несмотря на все вышесказанное, функциональность-то полезная!.. Можно ли написать класс, который инкапсулирует ее, но при этом не будет таким «магическим»? Можно и даже не очень трудно. В листинге 6.1 приведен простой класс `Transition`, имитирующий поведение оператора переключения.

Листинг 6.1. Класс `Transition`

```
class Transition
  A, B = :A, :B
  T, F = true, false

  # state,p1,p2 => newstate, result
  Table = {[A,F,F]=>[A,F], [B,F,F]=>[B,T],
           [A,T,F]=>[B,T], [B,T,F]=>[B,T],
           [A,F,T]=>[A,F], [B,F,T]=>[A,T],
           [A,T,T]=>[A,T], [B,T,T]=>[A,T]}

  def initialize(proc1, proc2)
    @state = A
    @proc1, @proc2 = proc1, proc2
    check?
  end

  def check?
    p1 = @proc1.call ? T : F
    p2 = @proc2.call ? T : F
    @state, result = *Table[@state,p1,p2]
    return result
  end
end
```

В классе `Transition` для управления переходами применяется простой конечный автомат. Он инициализируется парой объектов `proc` (теми же, что для оператора переключения). Мы утратили небольшое удобство: все переменные (например, `line`), которые используются внутри этих объектов, должны уже находиться в области видимости. Зато теперь у нас есть решение, свободное от «магии», и все выражения ведут себя так, как в любом другом контексте Ruby.

Вот слегка измененный вариант того же подхода. Здесь метод `initialize` принимает `proc` и два произвольных выражения:

```
def initialize(var, flag1, flag2)
  @state = A
  @proc1 = proc { flag1 == var.call }
  @proc2 = proc { flag2 == var.call }
  check?
end
```

Оператор ветвящегося равенства проверяет соотношение между границами и переменной. Переменная обернута в объект `proc`, потому что мы передаем это значение только один раз, но хотим иметь возможность вычислять его повторно. Поскольку `proc` – замыкание, это не составляет проблемы.

Вот как используется новая версия:

```
line = nil
trans = Transition.new(proc {line}, /=begin/, /=end/)
loop do
  break if eof?
  line = gets
  puts line if trans.check?
end
```

Я рекомендую именно такой подход, поскольку в нем все делается открыто, без привлечения «волшебства». Особую актуальность это приобретет, когда оператор переключения будет исключен из языка.

6.2.8. Нестандартные диапазоны

Рассмотрим пример диапазона, состоящего из произвольных объектов. В листинге 6.2 приведен класс для работы с римскими числами.

Листинг 6.2. Класс для работы с римскими числами

```
class Roman
  include Comparable

  I, IV, V, IX, X, XL, L, XC, C, CD, D, CM, M =
    1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000

  Values = %w[M CM D CD C XC L XL X IX V IV I]

  def Roman.encode(value)
    return "" if self == 0
    str = ""
    Values.each do |letters|
      rnum = const_get(letters)
      if value >= rnum
        return(letters + str.encode(value-rnum))
      end
    end
    str
  end

  def Roman.decode(rvalue)
    sum = 0
```

```
letters = rvalue.split('')
letters.each_with_index do |letter,i|
  this = const_get(letter)
  that = const_get(letters[i+1]) rescue 0
  op = that > this ? :- : :+
  sum = sum.send(op,this)
end
sum
end

def initialize(value)
  case value
  when String
    @roman = value
    @decimal = Roman.decode(@roman)
  when Symbol
    @roman = value.to_s
    @decimal = Roman.decode(@roman)
  when Numeric
    @decimal = value
    @roman = Roman.encode(@decimal)
  end
end

def to_i
  @decimal
end

def to_s
  @roman
end

def succ
  Roman.new(@decimal+1)
end

def <=>(other)
  self.to_i <=> other.to_i
end

def Roman(val)
  Roman.new(val)
end
```

Сначала несколько слов о самом классе. Его конструктору можно передать строку, символ (представляющий число, записанное римскими цифрами) или Fixnum (число, записанное обычными арабскими цифрами). Внутри выполняется преобразование и сохраняются обе формы. Имеется вспомогательный метод

Roman, это просто сокращенная запись вызова `Roman.new`. Методы класса `encode` и `decode` занимаются преобразованием из арабской формы в римскую и наоборот.

Для простоты я опустил контроль данных. Кроме того, предполагается, что римские цифры представлены прописными буквами.

Метод `to_i`, конечно же, возвращает десятичное значение, а `to_s` – число, записанное римскими цифрами. Метод `succ` возвращает следующее римское число: например, `Roman(:IV).succ` вернет `Roman(:V)`.

Оператор сравнения сравнивает десятичные эквиваленты. Мы включили с помощью директивы `include` модуль `Comparable`, чтобы получить доступ к операторам «меньше» и «больше» (реализация которых опирается на наличие метода сравнения `<=>`).

Обратите внимание на использование символов в следующем фрагменте:

```
op = that > this ? :- : :+
sum = sum.send(op,this)
```

Здесь мы решаем, какую будем выполнять операцию (она обозначается символом): сложение или вычитание. Это не более чем краткий способ выразить следующую идею:

```
if that > this
  sum -= this
else
  sum += this
end
```

Второй вариант длиннее, зато более понятен.

Поскольку в этом классе есть метод `succ` и полный набор операторов сравнения, его можно использовать для конструирования диапазонов. Пример:

```
require 'roman'

y1 = Roman(:MCMLXVI)
y2 = Roman(:MMIX)
range = y1..y2                # 1966..2009
range.each {|x| puts x}        # Выводятся 44 строки.

epoch = Roman(:MCMLXX)
range.include?(epoch)          # true

doomsday = Roman(2038)
range.include?(doomsday)       # false

Roman(:V) == Roman(:IV).succ   # true
Roman(:MCM) < Roman(:MM)      # true
```

6.3. Заключение

В этой главе мы познакомились с тем, что такое символы в Ruby и как они применяются. Мы продемонстрировали как стандартные, так и определенные пользователем способы употребления символов.

Также мы подробно остановились на диапазонах: поговорили о том, как преобразовать диапазон в массив, как применить его в качестве индекса для массива или строки, как обойти диапазон и т. д. Рассмотрели оператор переключения (и альтернативу старому синтаксису). Наконец, создали класс, который корректно работает в сочетании с операторами диапазона.

На этом обсуждение символов и диапазонов заканчивается. Но, поскольку они используются в Ruby очень часто (и, несомненно, полезны), то мы еще не раз встретимся в них в примерах кода.

Глава 7. Дата и время

.....
Знает ли кто-нибудь, что такое время на самом деле?
Чикаго, Чикаго IV

Один из самых сложных и противоречивых аспектов человеческой жизни – измерение времени. Чтобы приблизиться к истинному пониманию предмета, необходимо хорошо знать физику, астрономию, историю, юриспруденцию, бизнес и религию. Астрономам известно (в отличие от большинства из нас!), что солнечное и звездное время – не совсем одно и то же. Ведомо им и то, почему иногда к году добавляется «високосная секунда». Историки знают, что в октябре 1582 года, когда Италия переходила с григорианского календаря на юлианский, из календаря было изъято несколько дней. Немногим известна разница между астрономической и церковной Пасхой (почти всегда они совпадают). Многие не в курсе, что год, который не делится на 400 (например, 1900), високосным не является.

Вычисления, в которых участвуют дата и время, выполняются компьютерами уже давно, но в большинстве языков программирования это весьма утомительное занятие. Это относится и к Ruby в силу самой природы данных. Но в Ruby было последовательно предпринято несколько шагов с целью упрощения этих операций.

Для удобства читателя мы определим некоторые термины, которые, возможно, не всем известны. Они пришли как из естественного языка, так и из других языков программирования.

Среднее время по Гринвичу (Greenwich Mean Time, GMT) – устаревший термин, который теперь официально не употребляется. Новый глобальный стандарт называется «*всеобщее скоординированное время*» (Coordinated Universal Time, или UTC от французской аббревиатуры). GMT и UTC – по существу, одно и то же. По прошествии ряда лет разница между ними составит несколько секунд. В большинстве промышленных программ (в том числе в Ruby) эти системы измерения времени не различаются.

На *летнее время* переходят раз в полгода, сдвигая официальное время на один час. Поэтому обозначения часовых поясов в США обычно заканчиваются на ST (Standard Time – стандартное время) или DT (Daylight Time – летнее время). Это происходит в большинстве штатов США (если не во всех), да и во многих других странах.

Точка отсчета (epoch) – термин, пришедший из мира UNIX. В этой системе время обычно хранится как число секунд, прошедших с определенного момента (называемого точкой отсчета), а именно с полуночи 1 января 1970 года по Гринвичу.

(Отметим, что во временных поясах США точкой отсчета оказывается 31 декабря предыдущего года). Тем же словом обозначается не только начальный момент, но и время, прошедшее с этого момента.

Для выполнения большинства операций используется класс `Time`. Классы `Date` и `DateTime` обеспечивают дополнительную гибкость.

7.1. Определение текущего момента времени

Самый главный вопрос при манипуляциях с датами и временем: какой сегодня день и сколько сейчас времени? В Ruby при создании объекта класса `Time` без параметров устанавливаются текущие дата и время.

```
t0 = Time.new
```

Синонимом служит `Time.now`:

```
t0 = Time.now
```

Отметим, что разрешающая способность системного таймера на разных машинах различна. Иногда это микросекунды; в таком случае два объекта `Time`, созданных подряд, могут фиксировать разное время.

7.2. Работа с конкретными датами (после точки отсчета)

Большинству программ нужно работать только с датами, относящимися к будущему или недавнему прошлому. Для таких целей класса `Time` достаточно. Наиболее интересны методы `mktime`, `local`, `gm` и `utc`.

Метод `mktime` создает новый объект `Time` на основе переданных параметров. Параметры задаются по убыванию длительности промежутка: год, месяц, день, часы, минуты, секунды, микросекунды. Все параметры, кроме года, необязательны; по умолчанию предполагается минимально возможное значение. В некоторых машинных архитектурах микросекунды игнорируются. Час выражается числом от 0 до 23.

```
t1 = Time.mktime(2001)                # 1 января 2001 года, 0:00:00
t2 = Time.mktime(2001,3)
t3 = Time.mktime(2001,3,15)
t4 = Time.mktime(2001,3,15,21)
t5 = Time.mktime(2001,3,15,21,30)
t6 = Time.mktime(2001,3,15,21,30,15)  # 15 марта 2001 года, 21:30:15
```

Отметим, что в методе `mktime` используется местное поясное время. Поэтому у него есть синоним `Time.local`.

```
t7 = Time.local(2001,3,15,21,30,15)   # 15 марта 2001 года, 21:30:15
```

Метод `Time.gm`, по сути, делает то же самое, но в нем предполагается время GMT (или UTC). Поскольку автор книги проживает в центральном часовом поясе США, то разница составляет 8 часов:

```
t8 = Time.gm(2001,3,15,21,30,15)      # March 15, 2001 21:30:15 pm
# Это 13:30:15 по центральному времени!
```

У этого метода есть синоним `Time.utc`:

```
t9 = Time.utc(2001,3,15,21,30,15)      # March 15, 2001 21:30:15 pm
# Снова 13:30:15 по центральному времени.
```

Отметим одну важную вещь. Все эти методы могут принимать и альтернативный набор параметров. Метод экземпляра `to_a` (который преобразует время в массив отдельных компонентов) возвращает набор значений в следующем порядке: секунды, минуты, часы, день, месяц, год, день недели (0..6), порядковый номер дня в году (1..366), летнее время (`true` или `false`), часовой пояс (строка).

Поэтому такие вызовы тоже допустимы:

```
t0 = Time.local(0,15,3,20,11,1979,2,324,false,"GMT-8:00")
t1 = Time.gm(*Time.now.to_a)
```

Однако, глядя на первый пример, не думайте, что вы сможете изменить вычисляемые параметры, например день недели (в данном случае 2 означает вторник). Такое действие противоречило бы принципам организации календаря, поэтому на созданном объекте `Time` оно никак не отражается. 20 ноября 1979 года был вторник, и никакой код не сможет этого изменить.

И наконец, отметим, что есть много способов задать время некорректно, например указав тринадцатый месяц или 35-й день месяца. При любой подобной попытке возникнет исключение `ArgumentError`.

7.3. Определение дня недели

Есть несколько способов определить день недели. Во-первых, метод экземпляра `to_a` возвращает массив, содержащий всю информацию о моменте времени. Можно обратиться к его седьмому элементу; это число от 0 до 6, причем 0 соответствует воскресенью, а 6 – субботе.

```
time = Time.now
day = time.to_a[6]          # 2 (вторник)
```

Еще лучше воспользоваться методом экземпляра `yday`:

```
day = time.wday             # 2 (вторник)
```

Но и тот, и другой способ не очень удобны. Иногда нужно получить день недели в виде числа, но чаще нас интересует его название в виде строки. Для этого можно обратиться к методу `strftime`. Его название знакомо программистам на C. Он распознает около двадцати спецификаторов, позволяя по-разному форматировать дату и время (см. раздел 7.21).

```
day = time.strftime("%a")    # "Tue"
```

Можно получить и полное название:

```
long = time.strftime("%A")   # "Tuesday"
```

7.4. Определение даты Пасхи

Дату этого праздника всегда было сложно вычислить, так как она привязана к лунному календарю. Солнечный год не делится нацело на лунные месяцы, поэтому даты, основанные на таком исчислении времени, будут из года в год меняться.

Представленный ниже алгоритм хорошо известен с давних времен. Мы видели его реализацию на языках BASIC, Pascal и C. А теперь перевели и на Ruby:

```
def easter(year)
  c = year/100
  n = year - 19*(year/19)
  k = (c-17)/25
  i = c - c/4 - (c-k)/3 + 19*n + 15
  i = i - 30*(i/30)
  i = i - (i/28)*(1 - (i/28)*(29/(i+1))*((21-n)/11))
  j = year + year/4 + i + 2 - c + c/4
  j = j - 7*(j/7)
  l = i - j
  month = 3 + (l+40)/44
  day = 1 + 28 - 31*(month/4)
  [month, day]
end

date = easter 2001      # Найти месяц и день для 2001 года.
date = [2001] + date    # Добавить в начало год.
t = Time.local *date    # Передать параметры Time.local.
puts t                 # Sun Apr 15 01:00:00 GMT-8:00 2001
```

Кто-то, прочитав этот раздел о Пасхе, непременно спросит: «Церковная или астрономическая?» Честно говоря, не знаю. Если вам удастся выяснить, сообщите всем нам.

Я бы с удовольствием объяснил вам этот алгоритм, только вот сам его не понимаю... Что-то надо принимать на веру, а в данном случае это особенно уместно!

7.5. Вычисление n-ого дня недели в месяце

Иногда, зная год и месяц, хочется вычислить дату, скажем, третьего понедельника или второго вторника в этом месяце. Такую задачу решает код в листинге 7.1.

Чтобы найти n-ое вхождение данного дня недели, мы передаем n в качестве первого параметра. Второй параметр – номер дня недели (0 – воскресенье, 1 – понедельник и т.д.). Третий и четвертый параметры – месяц и год соответственно.

Листинг 7.1. Вычисление n-ого дня недели в месяце

```
def nth_wday(n, wday, month, year)
  if (!n.between? 1,5) or
    (!wday.between? 0,6) or
    (!month.between? 1,12)
    raise ArgumentError
  end
  t = Time.local year, month, 1
  first = t.wday
  if first == wday
    fwd = 1
  elsif first < wday
```



```

    fwd = wday - first + 1
  elsif first > wday
    fwd = (wday+7) - first + 1
  end
  target = fwd + (n-1)*7
  begin
    t2 = Time.local year, month, target
  rescue ArgumentError
    return nil
  end
  if t2.mday == target
    t2
  else
    nil
  end
end
end

```

Странный код в конце текста метода призван скорректировать давнюю традицию, принятую в функциях работы с датами. Если вы думаете, что попытка создать объект для представления 31 ноября приведет к ошибке, то разочарую вас. Почти все системы молчаливо преобразуют эту дату в 1 декабря. Если вы давным-давно программируете в UNIX, то, наверное, полагаете, что так и должно быть. Другие сочтут это ошибкой.

Не станем спорить о том, что должна делать системная библиотека и должен ли Ruby изменить это поведение. Но мы не хотим, чтобы наша процедура продолжала эту традицию. Если вы ищете, к примеру, пятую пятницу в ноябре 2000 года, то она вернет nil (а не 1 декабря 2000 года).

7.6. Преобразование из секунд в более крупные единицы

Иногда нужно преобразовать заданное число секунд в дни, часы, минуты и секунды. Это можно сделать следующим образом:

```

def sec2dhms(secs)
  time = secs.round           # Отбрасываем микросекунды.
  sec = time % 60             # Извлекаем секунды.
  time /= 60                  # Отбрасываем секунды.
  mins = time % 60            # Извлекаем минуты.
  time /= 60                  # Отбрасываем минуты.
  hrs = time % 24              # Извлекаем часы.
  time /= 24                  # Отбрасываем часы.
  days = time                  # Дни (последний остаток).
  [days, hrs, mins, sec]     # Возвращаем массив [d,h,m,s].
end

t = sec2dhms(1000000)         # Миллион секунд равно...

puts "#{t[0]} days,"         # 11 дней,
puts "#{t[1]} hours,"        # 13 часов,

```

```
puts "#{t[2]} minutes,"      # 46 минут
puts " and #{t[3]} seconds." # и 40 секунд.
```

Можно было пойти и дальше. Но неделю вряд ли назовешь полезной единицей, месяц не слишком точно определен, а год не всегда содержит одно и то же число дней.

Ниже приведена также обратная функция:

```
def dhms2sec(days,hrs=0,min=0,sec=0)
  days*86400 + hrs*3600 + min*60 + sec
end
```

7.7. Вычисление промежутка времени, прошедшего от точки отсчета

По разным причинам может понадобиться перейти от внутреннего (традиционного) представления времени к стандартному. В системе время хранится как число секунд, прошедших с точки отсчета.

Метод класса `Time.at` создает новый объект `Time`, зная, сколько секунд прошло с точки отсчета:

```
epoch = Time.at(0)          # Найти точку отсчета (1 января 1970 GMT)
newmil = Time.at(978307200) # Счастливого миллениума! (1 января 2001)
```

Обратная функция – это метод экземпляра `to_i`, который преобразует дату в целое число.

```
now = Time.now              # 16 Nov 2000 17:24:28
sec = now.to_i              # 974424268
```

Если нужны микросекунды, и система поддерживает такую точность, то можно воспользоваться методом `to_f` для преобразования в число с плавающей точкой.

7.8. Високосные секунды

.....

*Вот снова день исчез, как ветра легкий стон,
Из нашей жизни, друг, навеки выпал он.
Но я, куда живу, тревожиться не стану
О дне, что отошел, и дне, что не рожден.
Омар Хайям, «Рубаи»*

Хотите иметь дело с «високосными» секундами? Мой совет: не делайте этого.

Високосные секунды – это не миф. Одна была добавлена в 2005 году; его последняя минута состояла из 61 секунды, а не из 60. Библиотечные функции уже много лет учитывают возможность появления минут, состоящих из 61 секунды, но наш опыт показывает, что большинство операционных систем високосные се-

кунды игнорирует. Говоря «большинство», мы имеем в виду все, с которыми когда-либо сталкивались.

Известно, например, что високосная секунда была добавлена в конец последнего дня 1998 года. Вслед за моментом 23:59:59 наступил редкий момент 23:59:60. Но стандартная библиотека языка С, которой пользуется Ruby, этого в расчет не принимает.

```
t0 = Time.gm(1998, 12, 31, 23, 59, 59)
t1 = t0 + 1
puts t1                # Fri Jan 01 00:00:00 GMT 1999
```

Быть может (хотя и маловероятно), Ruby скорректирует эту ситуацию. Но во время работы над данной книгой таких планов не было.

7.9. Определение порядкового номера дня в году

Порядковый номер дня в году иногда еще называют юлианской датой, хотя это не имеет прямого отношения к юлианскому календарю, давно вышедшему из употребления. Многие считают, что такое название неправильно, поэтому мы им больше пользоваться не будем.

Но как ни называй порядковый номер дня, иногда хочется его узнать, то есть получить число от 1 до 366. В Ruby это просто – достаточно вызвать метод `yday`:

```
t = Time.now
day = t.yday          # 315
```

7.10. Контроль даты и времени

В разделе 7.5 было показано, что стандартные функции не проверяют корректность даты, а «переносят» ее вперед, если необходимо. Например, 31 ноября становится 1 декабря.

Иногда такое поведение даже желательно. А если нет, то спешу обрадовать: стандартная библиотека `Date` не считает такие даты правильными. Мы можем воспользоваться этим фактом для контроля переданной даты.

```
class Time

  def Time.validate(year, month=1, day=1,
                    hour=0, min=0, sec=0, usec=0)
    require "date"

    begin
      d = Date.new(year, month, day)
    rescue
      return nil
    end
    Time.local(year, month, day, hour, min, sec, usec)
  end

end
```

```
t1 = Time.validate(2000,11,30) # Создается корректный объект.
t2 = Time.validate(2000,11,31) # Возвращается nil.
```

Здесь не мудрствуя лукаво мы просто возвращаем `nil`, если переданные параметры не соответствуют правильной дате (полагаясь на вердикт. вынесенный классом `Date`). Мы оформили этот метод как метод класса `Time` по аналогии с другими методами создания объектов.

Отметим, что класс `Date` может работать и с датами, предшествующими точке отсчета, то есть дата 31 мая 1961 года с точки зрения этого класса вполне допустима. Но при попытке передать такие значения классу `Time` возникнет исключение `ArgumentError`. Мы не пытаемся его перехватить, полагая, что это лучше делать на том же уровне пользовательского кода, где обрабатывались бы исключения, скажем, от метода `Time.local`.

Раз уж зашла речь о `Time.local`, то отметим, что мы воспользовались именно этим методом. Захоти мы работать со временем по Гринвичу, нужно было бы вызывать метод `gmt`. Лучше реализовать оба варианта.

7.11. Определение недели в году

Что такое «порядковый номер недели», не вполне ясно. Разные компании, коалиции, правительственные учреждения и органы стандартизации по-разному определяют это понятие. Путаница связана с тем, что год может начинаться с любого дня недели. Все зависит от того, хотим ли мы учитывать неполные недели. К тому же в одних странах неделя начинается с воскресенья, в других – с понедельника.

В этом разделе мы предложим три варианта. Первые два основаны на методе `strftime` класса `Time`. Спецификатор `%U` отсчитывает недели, начинающиеся с воскресенья, а спецификатор `%W` – начинающиеся с понедельника.

Третью возможность предоставляет класс `Date`. В нем имеется метод `cweek`, который возвращает порядковый номер недели, следуя определению из стандарта ISO 8601 (согласно которому первой считается неделя, содержащая первый вторник года).

Если все это вам не подходит, можете придумать собственный алгоритм. Все три решения включены в один фрагмент кода:

```
require "date"

# Посмотрим, в какую неделю попадает 1 мая в 2002 и 2005 годах.

t1 = Time.local(2002,5,1)
d1 = Date.new(2002,5,1)

week1a = t1.strftime("%U").to_i # 17
week1b = t1.strftime("%W").to_i # 17
week1c = d1.cweek                # 18

t2 = Time.local(2005,5,1)
d2 = Date.new(2005,5,1)
```

```

week2a = t2.strftime("%U").to_i    # 18
week2b = t2.strftime("%W").to_i    # 18
week2c = d2.cweek                  # 17

```

7.12. Проверка года на високосность

В классе `Date` есть два метода класса `julian_leap?` и `gregorian_leap?`, но только последний применим к относительно недавнему времени. Есть также метод `leap?`, но это просто синоним `gregorian_leap?`.

```

require "date"
flag1 = Date.julian_leap? 1700    # true
flag2 = Date.gregorian_leap? 1700 # false
flag3 = Date.leap? 1700          # false

```

Любой ребенок знает первое правило проверки на високосность: год должен делиться на 4. Меньшее число людей знают второе правило: год не должен делиться на 100. И уж совсем немногие знают про исключение из второго правила: если год делится на 400, то он високосный. Таким образом, последний год тысячелетия является високосным, только если делится на 400; так, 1900 год не был високосным, а 2000 был. (Эта поправка необходима, потому что в году не ровно 365.25 дней, а приблизительно 365.2422.)

В классе `Time` нет аналогичного метода, но при желании его легко можно добавить.

```

class Time

  def Time.leap? year
    if year % 400 == 0
      true
    elsif year % 100 == 0
      false
    elsif year % 4 == 0
      true
    else
      false
    end
  end

end

```

Я привел этот код только для того, чтобы прояснить алгоритм; конечно, гораздо проще вызвать метод `Date.leap?`. В моей реализации это метод класса по аналогии с классом `Date`, но можно было бы сделать его и методом экземпляра.

7.13. Определение часового пояса

Метод `zone` класса `Time` возвращает название часового пояса в виде строки:

```

z1 = Time.gm(2000,11,10,22,5,0).zone    # "GMT-6:00"
z2 = Time.local(2000,11,10,22,5,0).zone  # "GMT-6:00"

```

7.14. Манипулирование временем без даты

Есть и другие возможности – поэкспериментируйте!

7.15. Сравнение моментов времени

Все четыре предложения if возвращают true.

7.16. Прибавление интервала к моменту времени

```
t0 = Time.now
t1 = t0 + 60          # Ровно одна минута с момента t0.
```

```
t2 = t0 + 3600      # Ровно один час с момента t0.
t3 = t0 + 86400     # Ровно один день с момента t0.
```

Может пригодиться функция `dhms2sec` (определена в разделе 7.6). Напомним, что по умолчанию параметры, соответствующие часам, минутам и секундам, равны 0.

```
t4 = t0 + dhms2sec(5,10)      # 5 дней, 10 часов в будущем.
t5 = t0 + dhms2sec(22,18,15)  # 22 дня, 18 часов, 15 минут в будущем.
t6 = t0 - dhms2sec(7)         # Ровно неделю назад.
```

Не забывайте, что для получения момента времени в прошлом нужно вычитать, как при вычислении `t6` в примере выше.

7.17. Вычисление разности между двумя моментами времени

Можно вычислить интервал между двумя моментами времени. В результате вычитания одного объекта `Time` из другого получаем число секунд:

```
today = Time.local(2000,11,10)
yesterday = Time.local(2000,11,9)
diff = today - yesterday      # 86400 секунд.
```

И снова оказывается полезной функция `sec2dhms`, которая определена в разделе 7.6.

```
past = Time.local(1998,9,13,4,15)
now = Time.local(2000,11,10,22,42)
diff = now - past
unit = sec2dhms(diff)
puts "#{unit[0]} дней,"      # 789 дней,
puts "#{unit[1]} часов,"     # 18 часов,
puts "#{unit[2]} минут"      # 27 минут
puts "и #{unit[3]} секунд."  # и 0 секунд.
```

7.18. Работа с конкретными датами (до точки отсчета)

В стандартной библиотеке `Date` есть одноименный класс для работы с датами, предшествующими полудню 1 января 1970 года.

Несмотря на некоторое перекрытие с классом `Time`, между ними есть существенные различия. Самое главное состоит в том, что класс `Date` вообще игнорирует время, то есть работает с точностью до одного дня. Кроме того, класс `Date` строже контролирует ошибки, чем класс `Time`: попытка обратиться к 31 июня (или к 29 февраля невисокосного года) приведет к исключению. Код даже «знает» о различных датах перехода на григорианский календарь в Италии и Англии (в 1582 и 1752 году соответственно) и может обнаружить «несуществующие» даты, появившиеся в результате такого перехода. Эта стандартная библиотека – паноптикум интересного и местами загадочного кода. К сожалению, у нас нет места для более подробного разговора о ней.

7.19. Взаимные преобразования объектов Date, Time и DateTime

В Ruby есть три основных класса для работы с датами и временем: `Time`, `Date` и `DateTime`. Опишем их особенности:

- Класс `Time` преимущественно обертывает соответствующие функции из стандартной библиотеки языка C. Они, как правило, опираются на точку отсчета в UNIX и потому не способны представлять моменты времени раньше 1970 года.
- Класс `Date` создан для преодоления недостатков класса `Time`. Он без труда справляется с датами в более отдаленном прошлом – например, позволяет представить день рождения Леонардо да Винчи (15 апреля 1452 года), и, кроме того, знает о реформе календаря. Но у него есть свои слабые места: он работает только с датами, игнорируя время.
- Класс `DateTime` наследует `Date` и пытается компенсировать отсутствующие в нем возможности. Он может представлять даты не хуже `Date` и время не хуже `Time`. Часто его способ представления даты и времени оказывается наилучшим.

Однако не думайте, что объект `DateTime` – это просто объект `Date`, к которому механически присоединен объект `Time`. На самом деле в классе `DateTime` отсутствуют такие методы, как `usec`, `dst`? и некоторые другие.

Итак, у нас есть три класса. К сожалению, не существует стандартного способа преобразовать один из них в любой другой. По мере развития Ruby подобные шероховатости будут устраняться. А пока обойдемся методами, приведенными в листинге 7.2. Спасибо Кирку Хейнсу (Kirk Haines).

Листинг 7.2. Преобразования между классами, представляющими даты и время

```
class Time
  def to_date
    Date.new(year, month, day)
  rescue NameError
    nil
  end

  def to_datetime
    DateTime.new(year, month, day, hour, min, sec)
  rescue NameError
    nil
  end
end

class DateTime
  def to_time
    Time.local(year, month, day, hour, min, sec)
  end
end
```



```
end

class Date
  def to_time
    Time.local(year, month, day)
  end
end
```

Эти методы пропускают наверх все исключения, кроме `NameError`. Зачем нужно его перехватывать? Потому что могло случиться так, что программа не затребовала (с помощью директивы `require`) библиотеку `date` (напомним, что классы `Date` и `DateTime` входят в эту стандартную библиотеку, а не являются системными). В таком случае методы `to_datetime` и `to_date` возвращают `nil`.

7.20. Извлечение даты и времени из строки

Дата и время могут быть представлены в виде строки самыми разными способами: в полной или сокращенной форме, с разной пунктуацией, различным порядком компонентов и т.д. Из-за такого разнообразия очень сложно написать код, интерпретирующий символьную строку как дату. Рассмотрим несколько примеров:

```
s1 = "9/13/98 2:15am"
s2 = "1961-05-31"
s3 = "11 July 1924"
s4 = "April 17, 1929"
s5 = "20 July 1969 16:17 EDT"
s6 = "Mon Nov 13 2000"
s7 = "August 24, 79"           # День разрушения Помпеи.
s8 = "8/24/79"
```

К счастью, большую часть работы за нас уже проделали. В модуле `ParseDate` есть единственный класс с таким же именем, а в нем — единственный метод `parsedate`. Он возвращает массив компонентов даты в следующем порядке: год, месяц, день, час, минута, секунда, часовой пояс, день недели. Вместо полей, которые не удалось распознать, возвращается `nil`.

```
require "parsedate.rb"
include ParseDate

p parsedate(s1)           # [98, 9, 13, 2, 15, nil, nil, nil]
p parsedate(s2)           # [1961, 5, 31, nil, nil, nil, nil, nil]
p parsedate(s3)           # [1924, 7, 11, nil, nil, nil, nil, nil]
p parsedate(s4)           # [1929, 4, 17, nil, nil, nil, nil, nil]
p parsedate(s5)           # [1969, 7, 20, 16, 17, nil, "EDT", nil]
p parsedate(s6)           # [2000, 11, 13, nil, nil, nil, nil, 1]
p parsedate(s7)           # [79, 8, 24, nil, nil, nil, nil, nil]
p parsedate(s8, true)     # [1979, 8, 24, nil, nil, nil, nil, nil]
```

Последние две строки иллюстрируют назначение второго параметра `parsedate`, который называется `guess_year`. Из-за привычки записывать год двумя цифрами может возникнуть неоднозначность. Последние две строки интерпретируются

по-разному; при разборе `s8` мы установили значение `guess_year` равным `true`, вследствие чего программа сочла, что имеется в виду четырехзначный год. С другой стороны, `s7` – это дата извержения Везувия в 79 году, так что двузначный год был употреблен сознательно.

Правило применения параметра `guess_year` таково: если год меньше 100 и `guess_year` равно `true`, преобразовать в четырехзначный год. Преобразование выполняется так: если год больше либо равен 70, прибавить к нему 1900, в противном случае прибавить 2000. Таким образом, 75 преобразуется в 1975, а 65 – в 2065. Такое правило применяется программистами повсеместно.

А что сказать о строке `s1`, в которой, вероятно, имелся в виду 1998 год? Не все потеряно, если полученное число передается другому фрагменту программы, который интерпретирует его как 1998.

Учтите, что `parsedate` практически не контролирует ошибки. Например, если подать ему на вход дату, в которой день недели установлен некорректно, то он несоответствия не обнаружит. Это всего лишь анализатор – со своей работой он справляется неплохо, а требовать от него большего было бы неправильно.

Следует особо отметить склонность этого кода к «американизмам». Когда американец пишет 3/4/2001, он обычно имеет в виду 4 марта 2001 года. В Европе и большинстве других мест это означает 3 апреля. Но если при записи всех дат применяется одно и то же соглашение, ничего страшного не произойдет. Ведь возвращается просто массив, и ничто не мешает вам мысленно переставить первый и второй элементы. Кстати, имейте в виду, что вышеописанным образом интерпретируется даже такая дата, как 15/3/2000, хотя нам совершенно очевидно, что 15 – это день, а не месяц. Метод же `parsedate` «на голубом глазу» сообщит, что 15 – номер месяца!..

7.21. Форматирование и печать даты и времени

Для получения канонического представления даты и времени служит метод `asctime`; у него есть синоним `ctime`.

Аналогичный результат дает метод `to_s`. Точно такая же строка будет напечатана, если просто передать объект, представляющий дату и время, методу `puts`.

С помощью метода `strftime` класса `Time` можно отформатировать дату и время почти произвольным образом. В этой главе мы уже встречали спецификаторы `%a`, `%A`, `%u`, `%w`, `%n`, `%m`, `%S`, `%I` и `%p`, а ниже приведены оставшиеся:

- `%b` Сокращенное название месяца ("Jan")
- `%B` Полное название месяца ("January")
- `%c` Предпочтительное представление локальной даты и времени
- `%d` День месяца (1..31)
- `%j` Порядковый номер дня в году (1..366); так называемая «юлианская дата»
- `%m` Номер месяца (1..12)
- `%w` Номер дня недели (0..6)
- `%x` Предпочтительное представление даты без времени
- `%y` Год в двузначном формате (без указания века)

- %Y Год в четырехзначном формате
- %Z Название часового пояса
- %% Знак % (процент)

Дополнительную информацию вы найдете в справочном руководстве по языку Ruby.

7.22. Преобразование часовых поясов

Обычно приходится работать только с двумя часовыми поясами: GMT (или UTC) и тем, в котором вы находитесь.

Метод `gmtime` преобразует время к поясу GMT (модифицируя сам вызывающий объект). У него есть синоним `utc`.

Может быть, вы думаете, что можно просто преобразовать момент времени в массив, подменить часовой пояс и выполнить обратное преобразование? Проблема в том, что все методы класса, к примеру `local` и `gm` (а также их синонимы `mktime` и `utc`), готовы создавать объект `Time` только в предположении, что указано либо местное время, либо время по Гринвичу.

Есть обходной путь для преобразования часового пояса. Но предполагается, что вы заранее знаете разницу во времени. Взгляните на следующий фрагмент:

```
mississippi = Time.local(2000,11,13,9,35)      # 9:35 am CST
california  = mississippi - 2*3600             # Минус два часа.
```

```
time1 = mississippi.strftime("%X CST")         # 09:35:00 CST
time2 = california.strftime("%X PST")          # 07:35:00 PST
```

Спецификатор `%X` в методе `strftime` просто выводит время в формате `hh:mm:ss`.

7.23. Определение числа дней в месяце

В текущей версии Ruby еще нет встроенной функции для этой цели. Но ее можно без труда написать самостоятельно:

```
require 'date'
def month_days(month,year=Date.today.year)
  mdays = [nil,31,28,31,30,31,30,31,31,30,31,30,31]
  mdays[2] = 29 if Date.leap?(year)
  mdays[month]
end

days = month_days(5)           # 31 (May)
days = month_days(2,2000)     # 29 (February 2000)
days = month_days(2,2100)     # 28 (February 2000)
```

7.24. Разбиение месяца на недели

Представьте, что нужно разбить месяц на недели, например чтобы напечатать календарь. Эту задачу решает приведенный ниже код. Возвращаемый массив состоит из подмассивов, по семь элементов в каждом. При этом первому элементу каждого

внутреннего массива соответствует воскресенье. Начальные элементы для первой недели и конечные для второй могут быть равны nil.

```
def calendar(month,year)
  days = month_days(month,year)
  t = Time.mktime(year,month,1)
  first = t.wday
  list = *1..days
  weeks = [[]]
  week1 = 7 - first
  week1.times { weeks[0] << list.shift }
  nweeks = list.size/7 + 1
  nweeks.times do |i|
    weeks[i+1] ||= []
    7.times do
      break if list.empty?
      weeks[i+1] << list.shift
    end
  end
  pad_first = 7-weeks[0].size
  pad_first.times { weeks[0].unshift(nil) }
  pad_last = 7-weeks[-1].size
  pad_last.times { weeks[-1].unshift(nil) }
  weeks
end

arr = calendar(12,2008) # [[nil, 1, 2, 3, 4, 5, 6],
  # [7, 8, 9, 10, 11, 12, 13],
  # [14, 15, 16, 17, 18, 19, 20],
  # [21, 22, 23, 24, 25, 26, 27],
  # [28, 29, 30, 31, nil, nil, nil]]
```

Чтобы было понятнее, распечатаем этот массив массивов:

```
def print_calendar(month,year)
  weeks = calendar(month,year)
  weeks.each do |wk|
    wk.each do |d|
      item = d.nil? ? " *4 : " %2d " % d
      print item
    end
    puts
  end
  puts
end
```

Выводится:

```
#      1  2  3  4  5  6
#  7  8  9 10 11 12 13
# 14 15 16 17 18 19 20
# 21 22 23 24 25 26 27
# 28 29 30 31
```

7.25. Заключение

В этой главе мы рассмотрели класс `Time`, который является оберткой для функций из стандартной библиотеки языка C. Были показаны его возможности и ограничения.

Мы также узнали, зачем существуют классы `Date` и `DateTime` и какую функциональность они предоставляют. Мы научились выполнять преобразования между этими классами и добавили несколько собственных полезных методов.

На этом обсуждение даты и времени завершается. Переходим к массивам, хэшам и другим перечисляемым структурам в Ruby.

Глава 8. Массивы, хэши и другие перечисляемые структуры

.....
*Все детали должны соединяться без усилий. Помните, что механизм,
который вы пытаетесь собрать, вами же был и разобран.
Если не удастся соединить детали, на то должна быть причина.
Ни в коем случае не пользуйтесь молотком.
Руководство по техническому обслуживанию компании IBM (1925)*

Простых переменных для практического программирования недостаточно. В любом современном языке поддерживаются более сложные виды структурированных данных и предоставляются механизмы для создания новых абстрактных типов данных.

Исторически самой первой и широко распространеннейшей составной структурой данных был *массив*. Давным-давно, еще в языке ФОРТРАН, массивы назывались индексированными переменными; сегодня они несколько видоизменились, но основная идея во всех языках одна и та же.

Относительно недавно очень популярной структурой стали хэши. Как и массив, *хэш* представляет собой индексированный набор данных. Но, в отличие от массива, в качестве индекса может выступать любой объект. (В Ruby, как и в большинстве других языков, элементы массива индексируются числами.)

Наконец, мы рассмотрим сам модуль Enumerable и разберемся, как он работает. И массивы, и хэши подмешивают этот модуль. То же самое может сделать и любой другой класс, которому необходима аналогичная функциональность.

Но не будем забегать вперед. Начнем с массивов.

8.1. Массивы

В Ruby массивы индексируются целыми числами; индексация начинается с нуля, как в языке C. На этом, впрочем, сходство и заканчивается.

Массивы в Ruby динамические. Можно (хотя это и не обязательно) задать размер массива при создании. Но после создания он может расти без вмешательства со стороны программиста.

Массивы в Ruby неоднородны, то есть в них могут храниться данные разных типов. В самом деле в массиве хранятся только ссылки на объекты, а не объекты как таковые. Исключение составляют только непосредственные значения, например объекта класса Fixnum.

Вместе с массивом хранится и его длина, поэтому нам не нужно тратить время на ее вычисление или сохранение во внешней переменной, обновляемой синхронно с массивом. К тому же итераторы определены таким образом, что на практике нам вообще редко приходится задумываться о длине массива.

Наконец, класс `Array` в Ruby предоставляет немало полезных функций для работы с массивами: доступ, поиск, конкатенирование и т. п. В этом разделе мы изучим встроенную функциональность и расширим ее.

8.1.1. Создание и инициализация массива

Для создания массива применяется специальный метод класса `[]`; перечисленные внутри скобок данные помещаются во вновь созданный массив. Ниже показаны три способа вызвать этот метод. (Массивы `a`, `b` и `c` инициализируются одинаково.)

```
a = Array.[](1,2,3,4)
b = Array[1,2,3,4]
c = [1,2,3,4]
```

Имеется также метод класса `new`, который принимает 0, 1 или 2 параметра. Первый параметр задает начальный размер массива (число элементов в нем). Второй определяет начальное значение каждого элемента:

```
d = Array.new                # Создать пустой массив.
e = Array.new(3)             # [nil, nil, nil]
f = Array.new(3, "blah")     # ["blah", "blah", "blah"]
```

Обратите особое внимание на последний пример. Типичная «ошибка начинающего» — думать, что все объекты в этом массиве различны. На самом деле это три ссылки на один и тот же объект. Поэтому, если вы его измените (а не замените другим), то изменятся все элементы массива. Чтобы не попасть в эту ловушку, воспользуйтесь блоком. Блок будет вычисляться по одному разу для каждого элемента, поэтому все элементы окажутся различными объектами:

```
f[0].capitalize!            # f равно: ["Blah", "Blah", "Blah"]
g = Array.new(3) { "blah" } # ["blah", "blah", "blah"]
g[0].capitalize!           # g равно: ["Blah", "blah", "blah"]
```

8.1.2. Доступ к элементам массива и присваивание им значений

Получить ссылку на элемент и присвоить ему значение можно с помощью методов класса `[]` и `[]=` соответственно. Каждый из них принимает один целочисленный параметр — либо пару целых чисел (начало и конец), либо диапазон. Отрицательные индексы отсчитываются от конца массива, начиная с -1.

Специальный метод экземпляра `at` реализует простейший случай получения ссылки на элемент. Поскольку он может принимать только один целочисленный параметр, то работает чуть быстрее.

```
a = [1, 2, 3, 4, 5, 6]
b = a[0]                # 1
c = a.at(0)              # 1
d = a[-2]               # 5
```

```

e = a.at(-2)           # 5
f = a[9]               # nil
g = a.at(9)            # nil
h = a[3,3]             # [4, 5, 6]
i = a[2..4]            # [3, 4, 5]
j = a[2...4]           # [3, 4]

a[1] = 8               # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30]  # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8] # [2, 4, 6, 8, 5, 6]
a[-1] = 12             # [2, 4, 6, 8, 5, 12]

```

В следующем примере ссылка на элемент, расположенный за концом массива, приводит к росту массива. Отметим, что подмассив можно заменить другим массивом, содержащим больше элементов, чем было. В этом случае массив также автоматически вырастет.

```

k = [2, 4, 6, 8, 10]
k[1..2] = [3, 3, 3]      # [2, 3, 3, 3, 8, 10]
k[7] = 99                # [2, 3, 3, 3, 8, 10, nil, 99]

```

Наконец, если одному элементу присвоить в качестве значения массив, то на место этого элемента будет вставлен вложенный массив (в отличие от присваивания диапазону):

```

m = [1, 3, 5, 7, 9]
m[2] = [20, 30]          # [1, 3, [20, 30], 7, 9]

```

С другой стороны...

```

m = [1, 3, 5, 7, 9]
m[2..2] = [20, 30]       # [1, 3, 20, 30, 7, 9]

```

Метод `slice` – синоним метода `[]`:

```

x = [0, 2, 4, 6, 8, 10, 12]
a = x.slice(2)           # 4
b = x.slice(2,4)         # [4, 6, 8, 10]
c = x.slice(2..4)        # [4, 6, 8]

```

Специальные методы `first` и `last` возвращают первый и последний элемент массива соответственно. Если массив пуст, они возвращают `nil`:

```

x = %w[alpha beta gamma delta epsilon]
a = x.first              # "alpha"
b = x.last               # "epsilon"

```

Мы уже видели ранее, что иногда ссылка на элементы может возвращать целый подмассив. Но существуют и другие способы обратиться к нескольким элементам.

Метод `values_at` принимает список индексов и возвращает массив, содержащий только указанные элементы. Его можно использовать в тех случаях, когда диапазон не подходит (так как нужные элементы находятся не в соседних позициях).

В более ранних версиях Ruby метод `values_at` назывался `indices` (синоним `indexes`). Теперь эти названия не используются.


```
x = {10, 20, 30, 40, 50, 60}
y = x.values_at(0, 1, 4)      # {10, 20, 50}
z = x.values_at(0..2,5)       # {10, 20, 30, 60}
```

8.1.3. Определение размера массива

Метод `length` и его синоним `size` возвращают число элементов в массиве. (Как всегда, эта величина на единицу больше индекса последнего элемента.)

```
x = ["a", "b", "c", "d"]
a = x.length                # 4
b = x.size                   # 4
```

Метод `nitems` отличается от предыдущих тем, что не учитывает элементы равные `nil`:

```
y = [1, 2, nil, nil, 3, 4]
c = y.size                   # 6
d = y.length                 # 6
e = y.nitems                  # 4
```

8.1.4. Сравнение массивов

При сравнении массивов возможны неожиданности – будьте осторожны!

Для сравнения массивов служит метод экземпляра `<=>`. Он работает так же, как в других контекстах, то есть возвращает -1 (меньше), 0 (равно) или 1 (больше). Методы `==` и `!=` опираются на реализацию метода `<=>`.

Массивы сравниваются поэлементно; первая же пара несовпадающих элементов определяет результат всего сравнения. (Предпочтение отдается левее расположенным элементам, как при сравнении двух длинных целых чисел «на глазок», когда мы сравниваем по одной цифре за раз.)

```
a = [1, 2, 3, 9, 9]
b = [1, 2, 4, 1, 1]
c = a <=> b                  # -1 (то есть a < b)
```

Если все элементы равны, то массивы считаются равными. Если один массив длиннее другого и все элементы вплоть до длины более короткого массива равны, то более длинный массив считается большим.

```
d = [1, 2, 3]
e = [1, 2, 3, 4]
f = [1, 2, 3]
if d < e                      # false
  puts "d меньше e"
end
if d == f
  puts "d равно f"           # Печатается "d равно f"
end
```

Поскольку класс `Array` не подмешивает модуль `Comparable`, то обычные операторы сравнения `<`, `>`, `<=` и `>=` для массивов не определены. Но при желании их легко определить самостоятельно:

```
class Array

  def <(other)
    (self <=> other) == -1
  end

  def <=(other)
    (self < other) or (self == other)
  end

  def >(other)
    (self <=> other) == 1
  end

  def >=(other)
    (self > other) or (self == other)
  end

end
```

Впрочем, было бы проще включить модуль Comparable:

```
class Array
  include Comparable
end
```

Определив эти операторы, можно пользоваться ими как обычно:

```
if a < b
  print "a < b"      # Печатается "a < b"
else
  print "a >= b"
end

if d < e
  puts "d < e"       # Печатается "d < e"
end
```

Может статься, что при сравнении массивов мы столкнемся с необходимостью сравнивать два элемента, для которых оператор `<=>` не определен или не имеет смысла. Следующий код приводит к возбуждению исключения (TypeError) во время выполнения, так как сравнение `3 <=> "x"` лишено смысла:

```
g = [1, 2, 3]
h = [1, 2, "x"]
if g < h
  puts "g < h"      # Ошибка!
                  # Ничего не выводится.
end
```

Если и это вас не смущает, то добавим, что сравнение на равенство и неравенство в этом случае работает. Объясняется это тем, что объекты разных типов считаются неравными, хотя мы и не можем сказать, какой из них больше.

```
if g != h
  puts "g != h"     # Здесь ошибка не возникает.
                  # Печатается "g != h"
end
```

Наконец, не исключено, что два массива, содержащих несравнимые типы данных, все равно можно сравнить с помощью операторов `<` и `>`. В примере ниже мы получаем определенный результат еще до того, как натолкнемся на несравнимые элементы:

```
i = [1, 2, 3]
j = [1, 2, 3, "x"]
if i < j          # Здесь ошибка не возникает.
  puts "i < j"    # Печатается "i < j"
end
```

8.1.5. Сортировка массива

Самый простой способ отсортировать массив – воспользоваться встроенным методом `sort`:

```
words = %w(the quick brown fox)
list = words.sort      # ["brown", "fox", "quick", "the"]
# Или отсортировать на месте:
words.sort!            # ["brown", "fox", "quick", "the"]
```

Здесь предполагается, что все элементы массива сравнимы между собой. При сортировке неоднородного массива, например `[1, 2, "three", 4]`, обычно возникает ошибка.

В подобных случаях можно воспользоваться также блочной формой того же метода. Ниже предполагается, что у каждого элемента есть хотя бы метод `to_s` (преобразующий его в строку):

```
a = [1, 2, "three", "four", 5, 6]
b = a.sort {|x,y| x.to_s <=> y.to_s}
# b равно [1, 2, 5, 6, "four", "three"]
```

Конечно, подобное упорядочение (в данном случае основанное на кодировке ASCII) может оказаться бессмысленным. При работе с неоднородным массивом нужно прежде всего задать себе вопрос, зачем вообще его сортировать. И почему приходится хранить в массиве объекты разных типов?

Описанная методика работает, потому что блок возвращает целое число (`-1` или `1`) при каждом вызове. Если возвращена `-1`, то есть `x` меньше `y`, то два элемента меняются местами. Чтобы отсортировать массив по убыванию, достаточно всего лишь изменить порядок сравнения:

```
x = [1, 4, 3, 5, 2]
y = x.sort {|a,b| b <=> a}    # [5, 4, 3, 2, 1]
```

Блоки можно применять и для более сложных сортировок. Предположим, что нужно отсортировать названия книг и фильмов следующим способом: регистрируется, полностью игнорируются пробелы, а также ряд знаков препинания и артикли. Ниже приведен простой пример (и преподаватели английского языка, и программисты будут удивлены таким способом упорядочения по алфавиту).

```
titles = ["Starship Troopers",
          "A Star is Born",
          "Star Wars",
```

```

"Star 69",
"The Starr Report"]
sorted = titles.sort do |x,y|
  # Удалить артикли
  a = x.sub(/^(a |an |the )/i, "")
  b = y.sub(/^(a |an |the )/i, "")
  # Удалить пробелы и знаки препинания
  a.delete!(" .,-?!")
  b.delete!(" .,-?!")
  # Преобразовать в верхний регистр
  a.upcase!
  b.upcase!
  # Сравнить a и b
  a <=> b
end

# Теперь sorted равно:
# [ "Star 69", "A Star is Born", "The Starr Report"
#   "Starship Troopers", "Star Wars"]

```

Данный пример не слишком полезен и, конечно, его можно было бы записать более компактно. Но идея в том, что для сравнения двух операндов в определенном порядке над ними можно выполнять произвольно сложный набор операций. (Отметим, однако, что мы не изменили исходные операнды, так как работали с их копиями.) Эта общая техника полезна во многих ситуациях, например для сортировки по нескольким ключам или по ключам, вычисляемым во время выполнения.

В последних версиях Ruby в модуль Enumerable добавлен метод `sort_by` (который, конечно, подмешивается к классу `Array`). Важно понимать, что он делает.

В методе `sort_by` применяется то, что программисты на Perl называют преобразованием Шварца – в честь Рэндала Шварца (Randal Schwartz), внесшего немалый вклад в развитие этого языка. Вместо того чтобы сортировать сами элементы массива, мы применяем к ним некоторую функцию и сортируем возвращаемые ей результаты.

В качестве искусственного примера рассмотрим список файлов, который необходимо отсортировать по размеру. Прямолинейный способ выглядит так:

```
files = files.sort {|x,y| File.size(x) <=> File.size(y) }
```

Однако тут есть две проблемы. Во-первых, слишком многословно. Надо бы сделать покомпактнее.

Во-вторых, при такой сортировке приходится многократно обращаться к диску, а это довольно дорогая операция (по сравнению с операциями в оперативной памяти). Хуже того, одна и та же операция может выполняться несколько раз.

Метод `sort_by` решает обе проблемы. Вот «правильный» способ:

```
files = files.sort_by {|x| File.size(x) }
```

Здесь каждый ключ вычисляется ровно один раз, а затем сохраняется в виде пары ключ-данные. Для небольших массивов производительность при таком подходе может даже снизиться, зато код получается более понятным

Не существует метода `sort_by!`. Но при желании вы можете написать его самостоятельно.

А как обстоит дело с сортировкой по нескольким ключам? Предположим, что имеется массив объектов, который нужно отсортировать по трем атрибутам: имени, возрасту и росту. Из того, что массивы можно сравнивать, следует, что такое решение будет работать:

```
list = list.sort_by {|x| [x.name, x.age, x.height] }
```

Конечно, элементы массива могут быть и не такими простыми. Допустимы произвольно сложные выражения.

8.1.6. Выборка из массива по заданному критерию

Иногда нужно найти в массиве один или несколько элементов так, как будто мы опрашиваем таблицу в базе данных. Для этого есть несколько способов; рассмотренные ниже реализованы в подмешанном модуле `Enumerable`.

Метод `detect` находит не больше одного элемента. Он принимает блок (которому элементы передаются последовательно) и возвращает первый элемент, для которого значение блока оказывается равным `true`.

```
x = [5, 8, 12, 9, 4, 30]
# Найти первый элемент, кратный 6.
x.detect {|e| e % 6 == 0 }          # 12
# Найти первый элемент, кратный 7.
x.detect {|e| e % 7 == 0 }          # nil
```

Разумеется, хранящиеся в массиве объекты могут быть произвольно сложными, равно как и условие, проверяемое в блоке.

Метод `find` – синоним `detect`. Метод `find_all` возвращает несколько элементов, а не один-единственный; `select` – синоним `find_all`.

```
# Продолжение предыдущего примера...
x.find {|e| e % 2 == 0 }            # 8
x.find_all {|e| e % 2 == 0 }        # [8, 12, 4, 30]
x.select {|e| e % 2 == 0 }          # [8, 12, 4, 30]
```

Метод `grep` вызывает оператор сравнения (то есть оператор ветвящегося равенства) для сопоставления каждого элемента с заданным образцом. В простейшей форме он возвращает массив, состоящий из элементов, соответствующих образцу. Так как используется оператор `==`, то образец не обязан быть регулярным выражением. (Имя `grep` пришло из UNIX и связано с командой старого редактора `g/re/p`.)

```
a = %w[January February March April May]
a.grep(/ary/)                      # ["January", "February"]
b = [1, 20, 5, 7, 13, 33, 15, 28]
b.grep(12..24)                     # [20, 13, 15]
```

Существует также блочная форма, которая позволяет преобразовать каждый результат перед записью в массив. Получающийся в результате массив содержит значения, возвращенные блоком, а не те, что были в блок первоначально переданы:

```
# Продолжение предыдущего примера...
# Будем сохранять длины строк.
a.grep(/ary/) { |m| m.length}      # [7, 8]
# Будем сохранять квадраты исходных элементов.
b.grep(12..24) { |n| n*n}          # {400, 169, 225}
```

Метод `reject` — полная противоположность `select`. Он исключает из массива элементы, для которых блок возвращает значение `true`. Имеется также вариант `reject!` для модификации массива «на месте»:

```
c = [5, 8, 12, 9, 4, 30]
d = c.reject { |e| e % 2 == 0}      # [5, 9]
c.reject! { |e| e % 3 == 0}
# c равно [5, 8, 4]
```

Методы `min` и `max` ищут минимальное и максимальное значение в массиве. У каждого метода есть две формы. В первой используется сравнение «по умолчанию», что бы это ни означало в конкретной ситуации (на базе оператора `<=>`). Во второй форме применяется блок для выполнения нестандартного сравнения.

```
a = %w[Elrond Galadriel Aragorn Saruman Legolas]
b = a.min                               # "Aragorn"
c = a.max                               # "Saruman"
d = a.min { |x,y| x.reverse <=> y.reverse } # "Elrond"
e = a.max { |x,y| x.reverse <=> y.reverse } # "Legolas"
```

Чтобы найти индекс минимального или максимального элемента (в предположении, что такой элемент один), применяется метод `index`:

```
# Продолжение предыдущего примера...
i = a.index a.min                      # 2
j = a.index a.max                      # 3
```

Такую же технику можно использовать и в других похожих ситуациях. Однако, если элемент не единственный, то будет найден только первый.

8.1.7. Специализированные функции индексирования

Для отображения индексов на элементы массива интерпретатор языка пользуется функцией индексирования. Поскольку методы доступа к элементам массива можно переопределять, мы можем реализовать любой способ индексирования.

Например, ниже реализован массив, в котором индексы начинаются с 1, а не с нуля:

```
class Array2 < Array

  def [](index)
    if index > 0
      super(index-1)
    else
      raise IndexError
    end
  end
end
```

```
def []=(index,obj)
  if index>0
    super(index-1,obj)
  else
    raise IndexError
  end
end

end
```

```
x = Array2.new
```

```
x[1]=5
x[2]=3
x[0]=1      # Ошибка.
x[-1]=1     # Ошибка.
```

Отметим, что отрицательные индексы (от конца массива) здесь запрещены. Имейте в виду, что в реальной задаче придется внести и другие изменения, например переопределить метод `slice` и пр. Но общую идею вы поняли.

Аналогичный подход можно применить для реализации многомерных массивов (мы еще вернемся к ним в разделе 8.1.11).

Можно также реализовать нечто вроде треугольной матрицы, как показано ниже. Это частный случай двумерного массива, в котором элемент в позиции x, y совпадает с элементом в позиции y, x (поэтому хранить можно только один). Иногда это бывает полезно, например для хранения неориентированного графа (как мы покажем ближе к концу главы).

```
class TriMatrix

  def initialize
    @store = []
  end

  def [](x,y)
    if x > y
      index = (x*x+x)/2 + y
      @store[index]
    else
      raise IndexError
    end
  end

  def []=(x,y,v)
    if x > y
      index = (x*x+x)/2 + y
      @store[index] = v
    else

```

```

        raise IndexError
      end
    end

    end

    t = TriMatrix.new

    t[3,2] = 1
    puts t[3,2] # 1

    puts t[2,3] # IndexError
  
```

В этом примере мы реализовали матрицу так, что номер строки должен быть больше или равен номеру столбца. Но можно было бы просто отобразить симметричные пары индексов на один и тот же элемент. Проектное решение зависит от предполагаемого способа использования матрицы.

Можно было унаследовать классу `Array`, но нам кажется, что наше решение понять легче. Формула индексирования довольно сложна, но десяти минут с карандашом и бумагой хватит, чтобы убедить любого в ее правильности. Чтобы сделать данный класс по-настоящему полезным, надо бы немного усовершенствовать его; оставляем вам это в качестве упражнения.

Кроме того, треугольную матрицу можно реализовать в виде массива, содержащего массивы, размер которых увеличивается по мере увеличения номера строки. Примерно так мы и поступили в разделе 8.1.11. Нетривиальная задача – гарантировать, что строка случайно не окажется больше, чем положено.

8.1.8. Реализация разреженной матрицы

Иногда бывает нужен массив, в котором определена лишь небольшая часть элементов, а остальные не определены вовсе или (даже чаще) равны 0. Подобная разреженная матрица потребляет так много памяти зря, что были найдены способы более изощренной ее реализации.

Конечно, в большинстве случаев обычного массива Ruby вполне достаточно, так как в современных компьютерах недостатка памяти не ощущается. Элемент, которому не присвоено значение, будет равен `nil`, так что на его хранение расходуется всего несколько байтов.

С другой стороны, присваивание значения элементу массива, лежащему за текущей правой границей, приводит к созданию всех промежуточных элементов, причем они получают значение `nil`. Например, если определены элементы от 0 до 9 и затем производится присваивание элементу 1000, то создаются также элементы с индексами от 10 до 999, равные `nil`. Если это неприемлемо, надо поискать альтернативу.

В предлагаемом нами варианте массивы вообще не используются. Для реализации разреженной матрицы лучше подойдет хэш (за дополнительной информацией обратитесь к разделу 8.2.14).

8.1.9. Массивы как математические множества

В большинстве языков множества напрямую не реализованы (Pascal составляет исключение). Но массивы в Ruby обладают некоторыми свойствами, которые позволяют использовать их как множества. В данном разделе мы рассмотрим эти свойства и добавим свои собственные.

В последних версиях Ruby стандартная библиотека содержит класс Set. Если вам приходится часто иметь дело с множествами, подумайте об использовании объектов Set вместо массивов. Этот класс рассмотрен в главе 9.

Массив нельзя назвать идеальным средством для представления множества, поскольку он может содержать дубликаты. Если вы хотите трактовать массив как множество, то дубликаты можно удалить (с помощью метода `uniq` или `uniq!`).

Над множествами производятся две основные операции: объединение и пересечение. Для этого применяются операторы `|` (или) и `&` (и) соответственно. Поскольку множество по определению не содержит дубликатов, то повторяющиеся элементы удаляются (вопреки ожиданиям тех, кому доводилось работать с объединением и пересечением массивов в других языках).

```
a = [1, 2, 3, 4, 5]
b = [3, 4, 5, 6, 7]
c = a | b           # [1, 2, 3, 4, 5, 6, 7]
d = a & b           # [3, 4, 5]
```

```
# Дубликаты удаляются...
```

```
e = [1, 2, 2, 3, 4]
f = [2, 2, 3, 4, 5]
g = e & f           # [2, 3, 4]
```

Для объединения множеств можно использовать и оператор конкатенации (`+`), но он не удаляет дубликаты.

Метод `-` соответствует операции «разность множеств»; результатом является множество, куда входят те элементы первого множества, которые не являются элементами второго (см. раздел 8.1.12).

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 6, 7]
c = a - b           # [1, 2, 3]
# Отметим, что наличие элементов 6 and 7 не отражается на результате.
```

Для «аккумуляции» множеств можно применять оператор `|=`; как и следовало ожидать, `a |= b` — то же самое, что `a = a | b`. Аналогичным образом оператор `&=` последовательно «сужает» множество.

Для массивов не определена операция ИСКЛЮЧАЮЩЕЕ ИЛИ, но мы можем без труда реализовать ее. В терминах теории множеств она соответствует выборке тех элементов, которые входят в объединение двух множеств, но не входят в их пересечение.

```
class Array

  def ^ (other)
```

```
(self | other) - (self & other)
end

end
```

```
x = {1, 2, 3, 4, 5}
y = {3, 4, 5, 6, 7}
z = x ^ y           # [1, 2, 6, 7]
```

Чтобы проверить, входит ли некий элемент в множество, пользуйтесь методом `include?` или `member?` (синоним, подмешанный из модуля `Comparable`):

```
x = {1, 2, 3}
if x.include? 2
  puts "yes"          # Печатается "yes"
else
  puts "no"
end
```

Конечно, это некоторое отступление от канонов математики, где для обозначения принадлежности множеству применяется символ, похожий на греческую букву эпсилон. Отступление в том смысле, что множество находится слева, а не справа от оператора, то есть мы спрашиваем не «принадлежит ли данный элемент множеству», а «содержит ли множество данный элемент».

Многим это безразлично. Но привыкшие к языку Pascal или Python (или впитавшие математический формализм с молоком матери) хотели бы, чтобы было по-другому. Такую возможность мы реализуем в следующем фрагменте:

```
class Object
  def in(other)
    other.include? self
  end

end

x = {1, 2, 3}
if 2.in x
  puts "yes"          # Печатается "yes"
else
  puts "no"
end
```

Лично я отправил запрос на изменение Ruby (RCR 241) с предложением ввести в язык оператор `in`. Он должен походить на одноименный оператор в языках Pascal, Python и даже SQL.

У этой идеи есть свои достоинства (к тому же `in` – уже зарезервированное слово), но единодушного одобрения она не получила. Может быть, оператор `in` появится в Ruby, а может, и нет.

Теперь обратимся к подмножествам и надмножествам. Как определить, является ли данное множество подмножеством или надмножеством другого? Встроенных методов для этого нет, но мы можем поступить следующим образом:

```
class Array

  def subset?(other)
    self.each do |x|
      if !(other.include? x)
        return false
      end
    end
    true
  end

  def superset?(other)
    other.subset?(self)
  end

end

a = [1, 2, 3, 4]
b = [2, 3]
c = [2, 3, 4, 5]

flag1 = c.subset? a      # false
flag2 = b.subset? a      # true
flag3 = c.superset? b    # true
```

Обратите внимание: мы выбрали «естественный» порядок, то есть задаем вопрос `x.subset? y` — «является ли `x` подмножеством `y`?», а не наоборот.

Для распознавания пустого множества достаточно проверить, пуст ли массив. Это делает метод `empty?`.

Операция дополнения опирается на идею универсального множества. Однако «универсальное множество» в каждой конкретной ситуации определяется по-разному, поэтому лучшим решением будет самое простое: сначала определим, что такое универсальное множество, а потом вычислим разность.

```
universe = [1, 2, 3, 4, 5, 6]
a = [2, 3]
b = universe - a      # Дополнение a = [1, 4, 5, 6]
```

Если считаете необходимым, можете определить и унарный оператор (например, `-` или `~`) для выполнения этой операции.

Элементы множества можно перебирать, обходя массив. Единственная разница заключается в том, что элементы будут появляться в определенном порядке, а это может оказаться нежелательным. О том, как перебирать массив в случайном порядке, будет рассказано в разделе 8.1.18.

Наконец, иногда возникает необходимость вычислить степень множества. Это не что иное, как множество всех подмножеств данного множества (включая его

само и пустое множество). Читатели, знакомые с дискретной математикой, в особенности с комбинаторикой, понимают, что число таких подмножеств равно 2^n . Сгенерировать степень множества можно следующим образом:

```
class Array

  def powerset
    num = 2**size
    ps = Array.new(num, [])
    self.each_index do |i|
      a = 2**i
      b = 2**(i+1) - 1
      j = 0
      while j < num-1
        for j in j+a..j+b
          ps[j] += [self[i]]
        end
        j += 1
      end
    end
    ps
  end

end

x = [1, 2, 3]
y = x.powerset
# y равно:
# [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

8.1.10. Рандомизация массива

Иногда нужно переставить элементы массива в случайном порядке. Первое, что приходит на ум, — тасование карточной колоды, но есть и другие применения — например, случайная сортировка списка вопросов.

Для решения этой задачи пригодится метод `rand` из модуля `Kernel`. Ниже показан один из возможных способов:

```
class Array

  def randomize
    self.sort_by { rand } # Сортировать по ключу, являющемуся
  end                    # случайным числом.

  def randomize!
    self.replace(self.randomize)
  end

end

x = [1, 2, 3, 4, 5]
```

```
y = x.randomize # {3, 2, 4, 1, 5}
x.randomize!    # x равно {3, 5, 4, 1, 2}
```

Из-за самой природы сортировки, вероятно, вносится некоторое статистическое смещение. Но обычно это не играет роли.

Выбрать случайный элемент массива (не запрещая дубликатов) можно так:

```
class Array

  def pick_random
    self[rand(self.length)]
  end

end
```

Наконец, не стоит забывать, что метод `rand` позволяет сгенерировать предсказуемую последовательность (например, для тестирования), если затравить алгоритм известным значением с помощью метода `srand` (см. раздел 5.28).

8.1.11. Многомерные массивы

Если для численного анализа вам нужны многомерные массивы, то в архиве приложений Ruby есть прекрасная библиотека `NArray`, которую написал Масахиро Танака (Masahiro Tanaka). Если необходим аппарат для работы с матрицами, обратитесь к стандартной библиотеке `matrix.rb`, которая была упомянута в разделе 5.10.

В следующем примере показан способ работы с многомерными массивами за счет перегрузки методов `[]` и `[]=` для отображения элементов на вложенный массив. Представленный класс `Array3` обеспечивает рудиментарные операции с трехмерными массивами, но он далеко не полон:

```
class Array3

  def initialize
    @store = [{}]
  end

  def [](a,b,c)
    if @store[a]==nil ||
       @store[a][b]==nil ||
       @store[a][b][c]==nil
      return nil
    else
      return @store[a][b][c]
    end
  end

  def []=(a,b,c,x)
    @store[a] = [{}] if @store[a]==nil
    @store[a][b] = [ ] if @store[a][b]==nil
    @store[a][b][c] = x
  end

end
```

```
end
```

```
x = Array3.new
x[0,0,0] = 5
x[0,0,1] = 6
x[1,2,3] = 99
```

```
puts x[1,2,3]
```

Единственное, чего мы реально добились, – так это удобного использования запятой в обозначении $[x,y,z]$ вместо употребляемой в языке С нотации $[x][y][z]$. Если С-подобная нотация вас устраивает, можете просто воспользоваться вложенными массивами Ruby. Еще одно мелкое достоинство – предотвращение ситуации, когда объектом, от имени которого вызывается оператор `[]`, оказывается `nil`.

8.1.12. Нахождение элементов, принадлежащих одному массиву и не принадлежащих другому

В Ruby эта задача решается проще, чем во многих других языках. Нужно просто вычислить «разность множеств»:

```
text = %w[the magic words are squeamish ossifrage]
dictionary = %w[an are magic the them these words]
# Найти неправильно написанные слова
unknown = text - dictionary # ["squeamish", "ossifrage"]
```

8.1.13. Преобразование или отображение массивов

Метод `collect` из модуля `Enumerable` часто позволяет сэкономить время и силы. Тем, кто привык к языку `Smalltalk`, он покажется интуитивно очевидным в большей степени, чем программистам на С.

Этот метод просто воздействует неким произвольным образом на каждый элемент массива, порождая в результате новый массив. Иными словами, он «отображает» один массив на другой (отсюда и синоним `map`).

```
x = %w[alpha bravo charlie delta echo foxtrot]
# Получить начальные буквы.
a = x.collect {|w| w[0..0]} # %w[a b c d e f]
# Получить длины строк.
b = x.collect {|w| w.length} # [5, 5, 7, 5, 4, 7]
# map – просто синоним.
c = x.map {|w| w.length} # [5, 5, 7, 5, 4, 7]
```

Имеется также вариант `collect!` (или `map!`) для модификации на месте.

```
x.collect! {|w| w.upcase}
# x равно %w[ALPHA BRAVO CHARLIE DELTA ECHO FOXTROT]
x.map! {|w| w.reverse}
# x равно %w[AHPLA OVARB EILRAHC ATLED OHCE TORTXOF]
```

8.1.14. Удаление из массива элементов равных nil

Метод `compact` (и его вариант `compact!` для модификации на месте) удаляет из массива элементы равные `nil`, оставляя все остальные без изменения:

```
a = [1, 2, nil, 3, nil, 4, 5]
b = a.compact      # [1, 2, 3, 4, 5]
a.compact!         # a равно [1, 2, 3, 4, 5]
```

8.1.15. Удаление заданных элементов из массива

В Ruby легко удалить элементы из массива – для этого даже существует много способов. Чтобы удалить элемент с известным индексом, достаточно вызвать метод `delete_at`:

```
a = [10, 12, 14, 16, 18]
a.delete_at(3)          # Возвращает 16.
# a равно [10, 12, 14, 18]
a.delete_at(9)          # Возвращает nil (вне диапазона).
```

Все элементы с заданным значением поможет удалить метод `delete`. Он возвращает значения удаленных элементов или `nil`, если искомый элемент не найден:

```
b = %w(spam spam bacon spam eggs ham spam)
b.delete("spam")        # Возвращает "spam"
# b равно ["bacon", "eggs", "ham"]
b.delete("caviar")      # Возвращает nil
```

Метод `delete` принимает также блок. Это не вполне согласуется с интуицией; если объект не найден, происходит вычисление блока (при этом могут выполняться разнообразные операции) и возвращается вычисленное значение.

```
c = ["alpha", "beta", "gamma", "delta"]
c.delete("delta") { "Nonexistent" }
# Возвращается "delta" (блок не вычисляется).
c.delete("omega") { "Nonexistent" }
# Возвращается "Nonexistent".
```

Метод `delete_if` передает каждый элемент массива в блок и удаляет те элементы, для которых вычисление блока дает `true`. Примерно так же ведет себя метод `reject!` с тем отличием, что последний может возвращать `nil`, когда массив не изменяется.

```
email = ["job offers", "greetings", "spam", "news items"]
# Удалить слова из четырех букв
email.delete_if { |x| x.length==4 }
# email равно ["job offers", "greetings", "news items"]
```

Метод `slice!` получает доступ к тем же элементам, что и `slice`, но, помимо возврата их значений, еще и удаляет из массива:

```
x = [0, 2, 4, 6, 8, 10, 12, 14, 16]
a = x.slice!(2)          # 4
# x is now [0, 2, 6, 8, 10, 12, 14, 16]
b = x.slice!(2,3)        # [6, 8, 10]
# x is now [0, 2, 12, 14, 16]
c = x.slice!(2..3)       # [12, 14]
# x is now [0, 2, 16]
```

Для удаления элементов из массива можно также пользоваться методами `shift` и `pop` (дополнительную информацию об их исходном предназначении вы найдете в разделе 9.2).

```
x = [1, 2, 3, 4, 5]
x.pop           # Удалить последний элемент.
# x is now [1, 2, 3, 4]
x.shift         # Удалить первый элемент.
# x is now [2, 3, 4]
```

Метод `reject` принимает блок и формирует новый массив без тех элементов, для которых блок возвращает `true`:

```
arr = [1,2,3,4,5,6,7,8]
odd = arr.reject {|x| x % 2 == 0 }    # [1,3,5,7]
```

Наконец, метод `clear` удаляет из массива все элементы. Это эквивалентно присваиванию переменной пустого массива, но чуть-чуть эффективнее:

```
x = [1, 2, 3]
x.clear
# x равно []
```

8.1.16. Конкатенирование массивов и добавление в конец массива

Часто нужно добавить в конец существующего массива отдельный элемент или целый массив. В Ruby это можно сделать разными способами.

Оператор `<<` добавляет объект в конец массива; в качестве значения он возвращает сам массив, поэтому можно объединять несколько таких операций в цепочку.

```
x = [1, 5, 9]
x << 13      # x равно [1, 5, 9, 13]
x << 17 << 21 # x равно [1, 5, 9, 13, 17, 21]
```

Аналогичную операцию выполняют методы `unshift` и `push`, которые добавляют элемент в начало и в конец массива соответственно (см. также следующий раздел данной главы).

Массивы можно конкатенировать методом `concat` или с помощью операторов `+` и `+=`:

```
x = [1,2]
y = [3,4]
z = [5,6]
b = y + z      # [3,4,5,6]
b += x         # [3,4,5,6,1,2]
z.concat y     # z равно [5,6,3,4]
```

Имейте в виду, что оператор `+=` всегда создает новый объект. Также не забывайте, что оператор `<<` добавляет в конец новый элемент, который сам может быть массивом.

```
a = [1,2]
b = [3,4]
a += b         # [1,2,3,4]
```

```
a = [1,2]
b = [3,4]
a << b         # [1,2,[3,4]]
```



```
a = [1,2]
b = [3,4]
a = a.concat(b) # [1,2,3,4]
```

8.1.17. Использование массива в качестве стека или очереди

Базовые операции со стеком называются `push` и `pop`, они добавляют и удаляют элементы в конец массива. Базовые операции с очередью – это `shift` (удаляет элемент из начала массива) и `unshift` (добавляет элемент в начало массива). Для добавления в конец массива можно также пользоваться оператором `<<` (по существу, синоним `push`).

Постарайтесь не запутаться. Методы `shift` и `unshift` модифицируют массив в начале, а `push`, `pop` и `<<` – в конце.

Эта тема будет продолжена в разделе 9.2.

8.1.18. Обход массива

Как и следовало ожидать, в классе `Array` есть стандартный итератор `each`. Но имеются и другие полезные итераторы.

Метод `reverse_each` обходит массив в обратном порядке. Результат такой же, как если бы мы вызвали сначала метод `reverse`, а потом `each`, но работает быстрее.

```
words = %w(Son I am able she said)
str = ""
words.reverse_each { |w| str += "#{w} " }
# str равно "said she able am I Son "
```

Если нужно только перебрать все индексы, можно воспользоваться итератором `each_index`. Конструкция `x.each_index` эквивалентна `(0..(x.size-1)).each` (то есть обходу всего диапазона индексов).

Итератор `each_with_index` (подмешанный из модуля `Comparable`) передает в блок как сам элемент, так и его индекс.

```
x = ["alpha", "beta", "gamma"]
x.each_with_index do |x,i|
  puts "Элемент #{i} равен #{x}"
end
# Выводятся три строки.
```

Предположим, что нужно обойти массив в случайном порядке. Ниже представлен итератор `random_each` (который просто вызывает метод `randomize`, описанный в разделе 8.1.10).

```
class Array

  # Предполагается, что метод randomize определен.

  def random_each
    temp = self.randomize
    temp.each {|x| yield x}
  end
end
```

```
dwarves = %w(Sleepy Doepy Happy Sneezzy Grumpy Bashful Doc)
list = ""
dwarves.random_each {|x| list += "#{x} "}
# list равен:
# "Bashful Doepy Sleepy Happy Grumpy Doc Sneezzy "
# (На вашей машине порядок может быть другим.)
```

8.1.19. Преобразование массива в строку с разделителями

Часто требуется вставить разделители между элементами массива, но не перед первым и не после последнего. Для этого предназначены метод `join` и оператор `*`.

```
been_there = ["Veni", "vidi", "vici."]
journal = been_there.join(", ") # "Veni, vidi, vici."
```

```
letters = ["Phi", "Mu", "Alpha"]
musicians = letters.join(" ") # "Phi Mu Alpha"
```

```
people = ["Bob", "Carol", "Ted", "Alice"]
movie = people * " and "
# movie равно "Bob and Carol and Ted and Alice"
```

Если необходимо обрабатывать последний элемент особым образом, например вставить перед ним слово «and», это можно сделать вручную:

```
list = %w[A B C D E F]
with_commas = list[0..-2]**", " + ", and " + list[-1]
# with_commas равно "A, B, C, D, E, and F"
```

8.1.20. Обращение массива

Чтобы переставить элементы массива в обратном порядке, воспользуйтесь методами `reverse` или `reverse!`:

```
inputs = ["red", "green", "blue"]
outputs = inputs.reverse # ["green", "blue", "red"]
priorities = %w(eat sleep code)
priorities.reverse! # ["code", "sleep", "eat"]
```

8.1.21. Удаление дубликатов из массива

Чтобы удалить из массива повторяющиеся экземпляры, воспользуйтесь методом `uniq` (или его вариантом для модификации на месте `uniq!`):

```
breakfast = %w(spam spam eggs ham eggs spam)
lunch = breakfast.uniq # ["spam", "eggs", "ham"]
breakfast.uniq! # Массив breakfast изменился.
```

8.1.22. Чередувание массивов

Предположим, что есть два массива и надо построить из них третий, который содер­жит массивы из двух элементов, взятых из соответственных позиций исходных массивов. В последних версиях Ruby модуль `Enumerable` содержит метод `zip`:

```
a = [1, 2, 3, 4]
b = ["a", "b", "c", "d"]
c = a.zip(b)
```

```
# c равно [{1,"a"}, {2,"b"}, {3,"c"}, {4,"d"}]
# Чтобы устранить вложенность, воспользуйтесь методом flatten
d = c.flatten
# d равно [1, "a", 2, "b", 3, "c", 4, "d"]
```

8.1.23. Вычисление частоты различных значений в массиве

Для массивов нет метода count, как для строк (чтобы подсчитать число вхождений каждого элемента). Поэтому создадим свой собственный:

```
class Array

  def count
    k=Hash.new(0)
    self.each{|x| k[x]+=1 }
    k
  end

end

meal = %w[spam spam eggs ham eggs spam]
items = meal.count
# items равно {"ham" => 1, "spam" => 3, "eggs" => 2}
spams = items["spam"] # 3
```

Обратите внимание, что метод возвращает хэш.

8.1.24. Инвертирование массива для получения хэша

Массив нужен для того, чтобы ассоциировать целое число (индекс) с данными. А если нужно инвертировать это отношение, то есть ассоциировать данные с индексом? Иными словами, породить хэш? Это можно сделать так:

```
class Array

  def invert
    h={}
    self.each_with_index{|x,i| h[x]=i}
    h
  end

end

a = ["red","yellow","orange"]
h = a.invert # {"orange"=>2, "yellow"=>1, "red"=>0}
```

8.1.25. Синхронная сортировка нескольких массивов

Предположим, что необходимо отсортировать массив, которому соответствуют «параллельные» массивы, то есть в соответственных позициях находятся логически связанные данные. Не хотелось бы, чтобы в результате сортировки это соответствие нарушилось.

В представленном ниже решении мы сортируем массив и сохраняем получившийся набор индексов. Затем список индексов (который сам является массивом) можно применить к любому другому массиву, чтобы расставить его элементы в том же порядке.

```
class Array

  def sort_index
    d=[]
    self.each_with_index{|x,i| d[i]=[x,i]}
    if block_given?
      d.sort {|x,y| yield x[0],y[0]}.collect{|x| x[1]}
    else
      d.sort.collect{|x| x[1]}
    end
  end

  def sort_with(ord=[])
    return nil if self.length!=ord.length
    self.values_at(*ord)
  end

end

a = [21, 33, 11, 34, 36, 24, 14]
b = a.sort_index
a2 = a.sort_with(b)
c = a.sort_index {|x,y| x%2 <=> y%2 }
a3 = a.sort_with(c)

p a # [21, 33, 11, 34, 36, 24, 14]
p b # [2, 6, 0, 5, 1, 3, 4]
p a2 # [11, 14, 21, 24, 33, 34, 36]
p c # [6, 5, 4, 3, 2, 1, 0]
p a3 # [14, 24, 36, 34, 11, 33, 21]
```

8.1.26. Указание значения по умолчанию

Для новых элементов массива

Когда массив растет и в нем создаются новые элементы, по умолчанию им присваивается значение nil:

```
a = Array.new
a[0]="x"
a[3]="y"
# a равно ["x", nil, nil, "y"]
```

Но, допустим, нам требуется, чтобы новые элементы получали другое значение. Тогда в качестве конкретного применения общего принципа предлагаем класс ZArray, описывающий массив, в котором вновь созданные элементы будут равны 0:

```

class ZArray < Array

  def [](x)
    if x > size
      for i in size+1..x
        self[i]=0
      end
    end
    v = super(x)
  end

  def []=(x,v)
    max = size
    super(x,v)
    if size - max > 1
      (max..size-2).each do |i|
        self[i] = 0
      end
    end
  end

end

num = ZArray.new
num[1] = 1
num[2] = 4
num[5] = 25
# num равно [0, 1, 4, 0, 0, 25]

```

8.2. Хэши

Хэши еще называют ассоциативными массивами, словарями и т.д. Особенно хорошо эта структура данных знакома программистам на языках Perl и Java.

Массив можно представить как структуру, которая создает ассоциацию между индексом x и элементом данных y . Хэш тоже создает подобную ассоциацию, но с двумя отличиями. Во-первых, в случае с массивом x – целое число, а для хэша это не обязательно. Во-вторых, массив – упорядоченная структура, тогда как элементы хэша обычно располагаются в непредсказуемом порядке.

Ключ хэша может иметь произвольный тип. Как следствие, хэш является не последовательной структурой данных. Мы знаем, что в массиве четвертый элемент следует за третьим. А в хэше тип ключа может быть таким, что понятия следующего и предыдущего значения не определены. По этой (и по другим) причинам в Ruby нет обозначений, наводящих на мысль о том, что пары в хэше следуют в каком-то определенном порядке.

Можно считать, что хэш – это массив со специальным индексом или некий аналог «таблицы синонимов» в базе данных. только оба поля хранятся в памяти.

Как бы вы ни представляли себе хэш, это полезный и мощный инструмент программирования.

8.2.1. Создание нового хэша

Как и в случае с классом `Array`, для создания хэша служит специальный метод класса `{}`. Данные, перечисленные в квадратных скобках, образуют ассоциированные пары. Ниже показаны шесть способов вызвать этот метод (все хэши с `a1` до `c2` содержат одни и те же данные).

```
a1 = Hash.new {"flat", 3, "curved", 2}
a2 = Hash.new {"flat"=>3, "curved"=>2}
b1 = Hash["flat", 3, "curved", 2]
b2 = Hash["flat"=>3, "curved"=>2]
c1 = {"flat", 3, "curved", 2}
c2 = {"flat"=>3, "curved"=>2}
# Для a1, b1 и c1 число элементов должно быть четным.
```

Есть также метод `new`, который может принимать параметр, задающий значение по умолчанию. Отметим, что это значение не является частью хэша – оно просто используется вместо `nil`.

```
d = Hash.new           # Создать пустой хэш.
e = Hash.new(99)       # Создать пустой хэш.
f = Hash.new("a"=>3)   # Создать пустой хэш.
e["angled"]            # 99
e.inspect              # {}
f["b"]                 # {"a"=>3} (значением по умолчанию
                        # является тоже хэш).
f.inspect              # {}
```

8.2.2. Указание значения по умолчанию для хэша

Значением по умолчанию для хэша является объект, возвращаемый вместо `nil` в случае, когда указанный ключ не найден. Это полезно, если вы планируете вызывать для возвращенного значения методы, которые для `nil` не определены. Задать значение по умолчанию можно в момент создания хэша или позже с помощью метода `default=`.

Все отсутствующие ключи указывают на один и тот же объект по умолчанию, поэтому изменение данного объекта имеет побочный эффект.

```
a = Hash.new("missing") # Объект по умолчанию – строка "missing".
a["hello"]               # "missing"
a.default="nothing"
a["hello"]               # "nothing"
a["good"] << "bye"       # "nothingbye"
a.default                 # "nothingbye"
```

Имеется также специальный метод экземпляра `fetch`, который возбуждает исключение `IndexError`, если в объекте типа `hash` нет указанного ключа. Он принимает также второй параметр, играющий роль значения по умолчанию. Кроме того, методу `fetch` можно передать необязательный блок, который выработает значение

по умолчанию, если ключ не будет найден. Таким образом, каждому отсутствующему ключу можно сопоставить свое «значение по умолчанию».

```
a = {"flat", 3, "curved", 2, "angled", 5}
a.fetch("pointed")           # IndexError
a.fetch("curved", "na")      # 2
a.fetch("x", "na")           # "na"
a.fetch("flat") {|x| x.upcase} # 3
a.fetch("pointed") {|x| x.upcase} # "POINTED"
```

8.2.3. Доступ к парам ключ-значение и добавление новых пар

В классе `Hash` есть методы класса `[]` и `[]=`. Используются они почти так же, как одноименные методы в классе `Array`, но принимают лишь один параметр. В качестве параметра может выступать любой объект, а не только строка (хотя строки используются чаще всего).

```
a = {}
a["flat"] = 3           # {"flat"=>3}
a.[]=("curved", 2)      # {"flat"=>3, "curved"=>2}
a.store("angled", 5)    # {"flat"=>3, "curved"=>2, "angled"=>5}
```

Метод `store` – просто синоним `[]=`, оба могут принимать два аргумента, как показано в примере выше.

Метод `fetch` аналогичен методу `[]`, но возбуждает исключение `IndexError`, когда ключ отсутствует. Есть у него и необязательный второй аргумент (или блок) для указания значения по умолчанию (см. раздел 8.2.2).

```
a["flat"]              # 3
a.[]("flat")           # 3
a.fetch("flat")         # 3
a["bent"]               # nil
```

Предположим, что мы не уверены, существует ли объект `hash`, но хотели бы избежать очистки имеющегося хэша. Очевидное решение – проверить, определен ли интересующий нас объект:

```
unless defined? a
  a={}
end
a["flat"] = 3
```

Но есть и другой способ:

```
a ||= {}
a["flat"] = 3
# Или даже так:
(a ||= {})["flat"] = 3
```

Тот же вопрос можно поставить для отдельных ключей, когда новое значение следует присваивать, лишь если такого ключа еще нет:

```
a=Hash.new(99)
a[2]           # 99
a              # {}
a[2] ||= 5     # 99
```

```
a # {}
b=Hash.new
b # {}
b[2] # nil
b[2] ||= 5 # 5
b # {2=>5}
```

Отметим, что `nil` может выступать и в качестве ключа, и в качестве значения:

```
b={}
b[2] # nil
b[3]=nil
b # {3=>nil}
b[2].nil? # true
b[3].nil? # true
b[nil]=5
b # {3=>nil, nil=>5}
b[nil] # 5
b[b[3]] # 5
```

8.2.4. Удаление пар ключ–значение

Удалить пары ключ–значение из хэша можно с помощью методов `clear`, `delete`, `delete_if`, `reject`, `reject!` и `shift`.

Метод `clear` удаляет из хэша все пары. Эффект такой же, как от присваивания переменной нового пустого хэша, но работает чуть быстрее.

Метод `shift` удаляет незаданную пару ключ–значение и возвращает ее в виде массива из двух элементов или `nil`, если никаких ключей не осталось.

```
a = {1=>2, 3=>4}
b = a.shift # [1,2]
# a равно {3=>4}
```

Метод `delete` удаляет конкретную пару ключ–значение. Он принимает в качестве параметра ключ и возвращает ассоциированное с ним значение, если такой ключ существовал (и был удален). В противном случае возвращается значение по умолчанию. Метод также принимает блок, который вырабатывает уникальное значение по умолчанию вместо того, чтобы возвращать ссылку на общий объект.

```
a = {1=>1, 2=>4, 3=>9, 4=>16}
a.delete(3) # 9
# a is now {1=>1, 2=>4, 4=>16}
a.delete(5) # В этом случае nil.
a.delete(6) { "не найдено" } # "не найдено".
```

Пользуйтесь методами `delete_if`, `reject` или `reject!` в сочетании с обязательным блоком, чтобы удалить все ключи, для которых блок возвращает значение `true`. Метод `reject` работает с копией хэша, а метод `reject!` возвращает `nil`, если не произведено никаких изменений.

8.2.5. Обход хэша

В классе `Hash` имеется стандартный итератор `each`, а кроме него итераторы `each_key`, `each_pair` и `each_value` (`each_pair` – синоним `each`).

Можно также получить массив, содержащий только ключи или только значения:

```
h.keys      # ["a", "b"]
h.values    # [1, 2]
```

Наконец, можно поместить в массив только значения, соответствующие заданному списку ключей. Этот метод работает для хэшей примерно так же, как одноименный метод для массивов. (Кроме того, как и в случае массивов, метод `values_at` заменяет устаревшие методы `indices` и `indexes`.)

```
h = {1=>"one", 2=>"two", 3=>"three", 4=>"four", "cinco"=>"five"}
h.values_at(3, "cinco", 4)      # ["three", "five", "four"]
h.values_at(1, 3)               # ["one", "three"]
```

8.2.9. Выборка пар ключ–значение по заданному критерию

К классу `Hash` подмешан модуль `Enumerable`, поэтому можно обращаться к методам `detect` (`find`), `select` (`find_all`), `grep`, `min`, `max` и `reject` (как и для массивов).

Метод `detect` (синоним `find`) находит одну пару ключ–значение. Он принимает блок (которому передается по одной паре за раз) и возвращает первую пару, для которой вычисление блока дает `true`.

```
names = {"fred"=>"jones", "jane"=>"tucker",
         "joe"=>"tucker", "mary"=>"SMITH"}
# Найти tucker.
names.detect {|k,v| v=="tucker" } # ["joe", "tucker"]
# Найти имена, записанные прописными буквами.
names.find {|k,v| v==v.upcase }   # ["mary", "SMITH"]
```

Разумеется, объекты в хэше могут быть сколь угодно сложными, как и условие, проверяемое в блоке, но сравнение объектов разных типов может оказаться проблематичным.

Метод `select` (синоним `find_all`) возвращает все пары, удовлетворяющие условию, а не только первую:

```
names.select {|k,v| v=="tucker" }
# [{"joe", "tucker"}, {"jane", "tucker"}]
names.find_all {|k,v| k.count("r")>0}
# [{"mary", "SMITH"}, {"fred", "jones"}]
```

8.2.10. Сортировка хэша

Хэши по природе своей не упорядочены ни по ключам, ни по значениям. Чтобы отсортировать хэш, Ruby преобразует его в массив, который затем сортирует. Понятно, что и результатом является массив.

```
names = {"Jack"=>"Ruby", "Monty"=>"Python",
         "Blaise"=>"Pascal", "Minnie"=>"Perl"}
list = names.sort
# list павно:
# [{"Blaise", "Pascal"}, {"Jack", "Ruby"},
#  {"Minnie", "Perl"}, {"Monty", "Python"}]
```

8.2.11. Объединение двух хэшей

Иногда бывает нужно объединить хэши. Метод `merge` получает два хэша и формирует из них третий, перезаписывая обнаружившиеся дубликаты:

```
dict = {"base"=>"foundation", "pedestal"=>"base"}
added = {"base"=>"non-acid", "salt"=>"NaCl"}
new_dict = dict.merge(added)
# {"base"=>"non-acid", "pedestal"=>"base", "salt"=>"NaCl"}
```

У метода `merge` есть синоним `update`.

Если задан блок, то он может содержать алгоритм устранения коллизий. В нижеприведенном примере, если два ключа совпадают, в объединенном хэше остается меньшее значение (по алфавиту, по числовому значению или в каком-то ином смысле):

```
dict = {"base"=>"foundation", "pedestal"=>"base"}
added = {"base"=>"non-acid", "salt"=>"NaCl"}
new_dict = dict.merge(added) { |key, old, new| old < new ? old : new }
# {"salt"=>"NaCl", "pedestal"=>"base", "base"=>"foundation"}
```

Таким образом, при использовании блока результат может получиться не такой, как в случае, когда блок не задан. Имеются также методы `merge!` и `update!`, которые изменяют вызывающий объект «на месте».

8.2.12. Создание хэша из массива

Простейший способ сделать это – прибегнуть к способу создания хэшей с помощью квадратных скобок. Следующий способ годится, если массив состоит из четного числа элементов.

```
array = [2, 3, 4, 5, 6, 7]
hash = Hash[*array]
# hash равно: {2=>3, 4=>5, 6=>7}
```

8.2.13. Вычисление разности и пересечения хэшей

Ключи хэша можно скопировать в отдельный массив, а к получившимся из разных хэшей массивам применить методы `&` и `-` класса `Array`. Результатом являются пересечение и разность множеств ключей. Соответствующие им значения можно получить с помощью метода `each`, примененного к хэшу, содержащему все образованные таким способом ключи.

```
a = {"a"=>1, "b"=>2, "z"=>3}
b = {"x"=>99, "y"=>88, "z"=>77}
intersection = a.keys & b.keys
difference = a.keys - b.keys
c = a.dup.update(b)
inter = {}
intersection.each { |k| inter[k]=c[k] }
# inter равно {"z"=>77}
diff={}
difference.each { |k| diff[k]=c[k] }
# diff равно {"a"=>1, "b"=>2}
```

8.2.14. Хэш как разреженная матрица

Часто в массиве или матрице заполнена лишь небольшая часть элементов. Можно хранить их как обычно, но такое расходование памяти неэкономно. Хэши позволяет хранить только реально существующие значения.

В следующем примере предполагается, что несуществующие значения по умолчанию равны нулю:

```
values = Hash.new(0)
values[1001] = 5
values[2010] = 7
values[9237] = 9
x = values[9237]      # 9
y = values[5005]      # 0
```

Ясно, что обычный массив в таком случае содержал бы более 9000 неиспользуемых элементов, что не всегда приемлемо.

А если нужно реализовать разреженную матрицу размерности два или более? В этом случае можно было бы использовать массивы в качестве ключей:

```
cube = Hash.new(0)
cube[[2000,2000,2000]] = 2
z = cube[[36,24,36]]      # 0
```

Здесь обычная матрица содержала бы миллиарды элементов.

8.2.15. Реализация хэша с повторяющимися ключами

Приверженцы математической строгости скажут, что хэш с повторяющимися ключами – вообще не хэш. Не станем спорить. Называйте как хотите, но на практике бывают случаи, когда нужна структура данных, обладающая гибкостью и удобством хэша и в то же время содержащая ключи-дубликаты.

В листинге 8.1 предложено частичное решение. Оно неполно по двум причинам. Во-первых, мы не стали реализовывать всю желательную функциональность, ограничившись лишь некоторым достаточно представительным подмножеством. Во-вторых, внутреннее устройство Ruby таково, что литеральный хэш всегда является экземпляром класса `Hash`, и, хотя мы наследуем классу `Hash`, литерал все равно не сможет содержать повторяющихся ключей (мы подумаем об этом позже).

Листинг 8.1. Хэш с повторяющимися ключами

```
class HashDup

  def initialize(*all)
    raise IndexError if all.size % 2 != 0
    @store = {}
    if all[0] # не nil
      keyval = all.dup
      while !keyval.empty?
        key = keyval.shift
        if @store.has_key?(key)
          @store[key] += [keyval.shift]
        end
      end
    end
  end
end
```

```
        else
          @store[key] = [keyval.shift]
        end
      end
    end
  end
end

def store(k,v)
  if @store.has_key?(k)
    @store[k] += [v]
  else
    @store[k] = [v]
  end
end

def [](key)
  @store[key]
end

def []=(key,value)
  self.store(key,value)
end

def to_s
  @store.to_s
end

def to_a
  @store.to_a
end

def inspect
  @store.inspect
end

def keys
  result=[]
  @store.each do |k,v|
    result += ([k]*v.size)
  end
  result
end

def values
  @store.values.flatten
end

def each
  @store.each {|k,v| v.each {|y| yield k,y}}
end
```

```
alias each_pair each

def each_key
  self.keys.each {|k| yield k}
end

def each_value
  self.values.each {|v| yield v}
end

def has_key? k
  self.keys.include? k
end

def has_value? v
  self.values.include? v
end

def length
  self.values.size
end

alias size length

def delete k
  val = @store[k]
  @store.delete k
  val
end

def delete k,v
  @store[k] -= [v] if @store[k]
  v
end

# Остальные методы опущены...

end

# Не будет работать... для повторяющихся ключей
# актуально только последнее значение.
h = {1=>1, 2=>4, 3=>9, 4=>16, 2=>0}

# А так будет...
h = HashDup.new(1,1, 2,4, 3,9, 4,16, 2,0)
k = h.keys      # [4, 1, 2, 2, 3]
v = h.values    # [16, 1, 4, 0, 9]
```

```
n = h.size           # 5

h.each {|k,v| puts "#{k} => #{v}" }
# Печатается:
# 4 => 16
# 1 => 1
# 2 => 4
# 2 => 0
# 3 => 9
```

Но если не пользоваться литеральными хэшами, то задача решаема. В листинге 8.1 реализован класс, содержащий атрибут `@store`, который является обычным хэшем; каждое значение этого хэша представляет собой массив. Доступ к хэшу организован так, что при необходимости добавить ключ, который уже существует, мы на самом деле добавляем новое значение в массив, ассоциированный с этим ключом.

Что должен возвращать метод `size`? Очевидно, «истинное» число пар ключ–значение, включая и дубликаты. Аналогично метод `keys` возвращает массив, который может содержать дубликаты. Итераторы ведут себя естественно; как и в случае обычного хэша, порядок обхода непредсказуем.

Помимо стандартного метода `delete` мы реализовали метод `delete_pair`. Первый удаляет все значения, ассоциированные с данным ключом, второй – только конкретную пару ключ–значение. (Отметим, что было бы затруднительно реализовать единственный метод вида `delete(k,v=nil)`, так как `nil` – допустимое значение в любом хэше.)

Для краткости мы не стали реализовывать весь класс целиком и, честно говоря, для некоторых методов, например `invert`, пришлось бы принимать небанальные решения по поводу желательного поведения. Интересующийся читатель может восполнить пробелы.

8.3. Перечисляемые структуры в общем

Что делает набор перечисляемым? Вообще-то сам тот факт, что это набор. Модуль `Enumerable` требует, чтобы был определен стандартный итератор `each`. Последовательность обхода не имеет значения, так как даже неупорядоченные наборы, например хэш, могут обладать итераторами.

Кроме того, если предполагается пользоваться методами `min`, `max` и `sort`, то для набора должен быть определен метод сравнения (`<=>`). Все это достаточно очевидно.

Итак, *перечисляемая структура* представляет собой набор, в котором можно производить поиск, который можно обойти и, быть может, отсортировать. В любой определенный пользователем набор, не являющийся подклассом существующего системного класса, имеет смысл подмешивать модуль `Enumerable`.

Имейте в виду – все сказанное о какой-то одной перечисляемой структуре относится ко всем. В качестве примеров таких структур можно назвать массив, хэш, дерево и т.д.

Конечно, у каждой структуры есть свои нюансы. Массив – это упорядоченный набор отдельных элементов, а хэш – неупорядоченный набор пар ключ–значение. Понятно, что в каких-то отношениях они будут вести себя по-разному.

Многие методы, с которыми мы познакомились при изучении массивов и хэшей (например, `map` и `find`), на самом деле определены в модуле `Enumerable`. Часто было трудно решить, как подать материал. Любая путаница или неточность – моя вина!..

Массив – наиболее часто употребляемый набор, подмешивающий этот модуль. Поэтому по умолчанию я буду пользоваться в примерах именно массивами.

8.3.1. Метод `inject`

Метод `inject` пришел в Ruby из языка Smalltalk (впервые он появился в версии Ruby 1.8). Его поведение интересно, хотя с первого раза понять его нелегко.

Он отражает тот факт, что мы часто хотим обойти список и по ходу «аккумулировать» некоторый результат. Конечно, самый естественный пример – суммирование чисел в списке. Но и для других операций обычно есть некий «аккумулятор» (которому присваивается начальное значение) и применяемая функция (в Ruby она представлена блоком).

В качестве тривиального примера рассмотрим массив чисел, которые нужно просуммировать:

```
nums = [3,5,7,9,11,13]
sum = nums.inject(0) {|x,n| x+n }
```

Обратите внимание, что начальное значение аккумулятора равно 0 («нейтральный элемент» для операции сложения). Затем блок получает текущее значение аккумулятора и значение текущего элемента списка. Действие блока заключается в прибавлении нового значения к текущей сумме.

Ясно, что этот код эквивалентен следующему:

```
sum = 0
nums.each {|n| sum += n }
```

В данном случае уровень абстракции лишь немногим выше. Если идея метода `inject` не укладывается у вас в голове, не пользуйтесь им. Но если удалось преодолеть первоначальное непонимание, то вы сможете найти ему новые элегантные применения.

Начальное значение аккумулятора задавать необязательно. Если оно опущено, то в качестве такового используется значение первого элемента, который при последующих итерациях пропускается.

```
sum = nums.inject {|x,n| x+n }
```

То же самое, что:

```
sum = nums[0]
nums[1..-1].each {|n| sum += n }
```

Другой похожий пример – вычисление произведения чисел. В данном случае аккумулятору следует присвоить начальное значение 1 (нейтральный элемент для операции умножения).


```
prod = nums.inject(1) {|x,n| x*n }
```

```
# или
```

```
prod = nums.inject {|x,n| x*n }
```

В следующем немного более сложном примере мы находим самое длинное слово в списке:

```
words = %w[ alpha beta gamma delta epsilon eta theta ]
longest_word = words.inject do |best,w|
  w.length > best.length ? w : best
end
# Возвращается значение "epsilon".
```

8.3.2. Кванторы

Кванторы `any?` и `all?` появились в версии Ruby 1.8, чтобы было проще проверять некоторые свойства набора. Оба квантора принимают в качестве параметр блок (который должен возвращать значение `true` или `false`).

```
nums = [1,3,5,8,9]
```

```
# Есть ли среди чисел четные?
```

```
flag1 = nums.any? {|x| x % 2 == 0 }    # true
```

```
# Все ли числа четные?
```

```
flag2 = nums.all? {|x| x % 2 == 0 }    # false
```

Если блок не задан, то просто проверяется значение истинности каждого элемента. Иными словами, неявно добавляется блок `{|x| x}`.

```
flag1 = list.all?      # list не содержит ни одного false или nil.
```

```
flag1 = list.any?      # list содержит хотя бы одно истинное значение
                        #   не nil и не false).
```

8.3.3. Метод `partition`

Как говорится, «в мире есть два сорта людей: те, что делят людей по сортам, и те, что не делят». Метод `partition` относится не к людям (хотя мы можем представить их в Ruby как объекты), но тоже делит набор на две части.

Если при вызове `partition` задан блок, то он вычисляется для каждого элемента набора. В результате создаются два массива: в первый попадают элементы, для которых блок вернул значение `true`, во второй – все остальные. Метод возвращает массив, двумя элементами которого являются эти массивы.

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
odd_even = nums.partition {|x| x % 2 == 1 }
# [[1,3,5,7,9],[2,3,4,6,8]]
```

```
under5 = nums.partition {|x| x < 5 }
# [[1,2,3,4],[5,6,7,8,9]]
```

```
squares = nums.partition {|x| Math.sqrt(x).to_i**2 == x }
# [[1,4,9],[2,3,5,6,7,8]]
```

Если нужно разбить набор больше чем на две группы, придется написать собственный метод. Я назвал его `classify` по аналогии с методом из класса `Set`.

```
module Enumerable
  def classify(&block)
    hash = {}
    self.each do |x|
      result = block.call(x)
      (hash[result] ||= []) << x
    end
    hash
  end
end

nums = [1,2,3,4,5,6,7,8,9]
mod3 = nums.classify {|x| x % 3 }
# { 0=>[3,6,9], 1=>[1,4,7], 2=>[2,5,8] }

words = %w[ area arboreal brick estrous clear donor ether filial
patina ]
vowels = words.classify {|x| x.count("aeiou") }
# {1=>["brick"], 2=>["clear", "donor", "ether"],
# 3=>["area", "estrous", "filial", "patina"], 4=>["arboreal"]}

initials = words.classify {|x| x[0..0] }
# {"a"=>["area", "arboreal"], "b"=>["brick"], "c"=>["clear"],
#  "d"=>["donor"], "p"=>["patina"], "e"=>["estrous", "ether"],
#  "f"=>["filial"]}
```

8.3.4. Обход с группировкой

До сих пор мы обходили список по одному элементу за раз. Но иногда желательно на каждой итерации анализировать по два, три или более элементов.

Итератор `each_slice` принимает в качестве параметра число `n`, равное числу просматриваемых на каждой итерации элементов. (Для работы с ним нужна библиотека `enumerator`.) Если не осталось достаточного количества элементов, размер последнего фрагмента будет меньше.

```
require 'enumerator'

arr = [1,2,3,4,5,6,7,8,9,10]
arr.each_slice(3) do |triple|
  puts triple.join(",")
end

# Выводится:
# 1,2,3
# 4,5,6
# 7,8,9
# 10
```

Имеется также итератор `each_cons`, который позволяет обходить набор методом «скользящего окна» заданного размера. (Если название кажется вам странным, знайте, что это наследие языка Lisp.) В таком случае фрагменты всегда будут иметь одинаковый размер.

```
require 'enumerator'

arr = [1,2,3,4,5,6,7,8,9,10]
arr.each_cons(3) do |triple|
  puts triple.join(",")
end
```

```
# Выводится:
# 1,2,3
# 2,3,4
# 3,4,5
# 4,5,6
# 5,6,7
# 6,7,8
# 7,8,9
# 8,9,10
```

8.3.5. Преобразование в массив или множество

Каждая перечисляемая структура теоретически может быть тривиально преобразована в массив (методом `to_a`). Например, такое преобразование для хэша дает вложенный массив пар:

```
hash = {1=>2, 3=>4, 5=>6}
arr = hash.to_a          # [[5, 6], [1, 2], [3, 4]]
```

Синонимом `to_a` является метод `entries`.

Если была затребована библиотека `set`, становится доступен также метод `to_set`. Дополнительная информация о множествах приведена в разделе 9.1.

```
require 'set'
hash = {1=>2, 3=>4, 5=>6}
set = hash.to_set        # #<Set: {[1, 2], [3, 4], [5, 6]}>
```

8.3.6. Эnumераторы

Объект класса `Enumerator` — по существу, обертка, превращающая итераторный метод в полноценный объект `Enumerable`. Оборнутый таким способом итератор приобретает все методы и свойства, присущие перечисляемым структурам.

В следующем искусственном примере в классе `Foo` есть итератор и больше ничего. Да и сам-то итератор не делает ничего полезного, только четыре раза вызывает `yield`. Чтобы подчеркнуть особенность его работы, итератор назван `every`, а не `each`.

```
require 'enumerator'
```

```
class Foo
  def every
```

```

yield 3
yield 2
yield 1
yield 4
end
end

foo = Foo.new

```

```

# Передается объект и имя итератора...
enum = Enumerable::Enumerator.new(foo, :every)

```

```

enum.each {|x| p x }      # Печатаются элементы
array = enum.to_a        # [3,2,1,4]
sorted = enum.sort       # [1,2,3,4]

```

Преобразование выглядит загадочно, но, по сути, это не что иное как:

```

enum = []
foo.every {|x| enum << x }

```

В примере выше `enum` – настоящий массив, а не просто объект `Enumerator`. Как следствие, несмотря на некоторые тонкие различия, это еще один способ преобразовать объект в перечисляемую структуру `Enumerable`.

Если затребована библиотека `enumerator`, то в классе `Object` появляется метод `enum_for`. Поэтому создание объекта в первом примере можно записать компактнее:

```

enum = foo.enum_for(:every)

```

Мы уже видели, как итераторы `each_slice` и `each_cons` позволяют осуществлять обход с группировкой. Оказывается, что есть специальные методы `enum_slice` и `enum_cons`, которые создают из таких итераторов объекты-эnumераторы (по существу, трансформируя имя итератора в `each`). Имейте в виду, что методы `Enumerable::Enumerator.new` и `enum_for` могут принимать необязательный список аргументов в качестве последнего параметра. Ниже мы воспользовались этим для передачи итератору «размера окна»:

```

array = [5,3,1,2]

discrete = array.enum_slice(2)
# То же, что Enumerable::Enumerator.new(array, :each_slice, 2)

overlap = array.enum_cons(2)
# То же, что Enumerable::Enumerator.new(array, :each_cons, 2)

discrete.each {|x| puts x.join(",") }
# Выводится:
# 5,3
# 1,2

overlap.each {|x| puts x.join(",") }

```

```
# Выводится:
# 5,3
# 3,1
# 1,2
```

8.3.7. Объекты-генераторы

Идея генератора довольно интересна. Обычный итератор в Ruby является внутренним, он запускает некоторый алгоритм, повторно вызывая блок кода.

Но бывают также и внешние итераторы. В этом случае алгоритм запускается самой программой, а итератор поставляет данные «по запросу», а не в соответствии с собственным «графиком».

В качестве аналогии можно рассмотреть метод `getline`, который выступает в роли внешнего итератора для объекта класса `IO`. Вы сами вызываете его в удобные моменты времени, а он возвращает прочитанные данные. Сравните это с поведением итератора `each_line`, который последовательно передает программе прочитанные строки.

Иногда внутренние итераторы не вполне подходят. Они позволяют решить задачу, но не лучшим способом. Внешний итератор был бы удобнее.

Библиотека `generator` позволяет преобразовать внутренний итератор во внешний. Она предоставляет такие же методы `next`, `rewind` и `end?`, как в классе `IO`. Вот пример:

```
require 'generator'

array = [7,8,9,10,11,12]

gen = Generator.new(array)

what = gen.current      # 7
where = gen.index       # 0 (то же, что pos)

while gen.end? and gen.current < 11
  gen.next
end

puts gen.current        # 11
puts gen.next           # 11
puts gen.index          # 4   (index - то же, что pos)
puts gen.next?          # true (next? - то же, что end?)
puts gen.next           # 12
puts gen.next?          # false
```

Обратите внимание, как мы «читаем» набор по одному элементу в одном или нескольких циклах. Метод `end?` обнаруживает конец набора; если вы проигнорируете его «совет», генератор возбудит исключение `EOFError`. Синонимом `end?` служит `next?`.

Метод `index` (синоним `pos`) сообщает индекс или позицию в наборе. Естественно, индексация начинается с нуля, как в случае с массивом или смещением от начала файла.

Методы `current` и `next`, возможно, интуитивно неочевидны. Представьте себе, что в начале выполняется операция «получить»; тогда текущий (`current`) элемент оказывается таким же, как следующий (`next`). Ясно, что метод `next` продвигает указатель на следующую позицию, а `current` – нет.

Поскольку для многих наборов возможно только продвижение в прямом направлении, то и генератор ведет себя так же. Не существует метода `prev` (предыдущий); теоретически его можно было бы добавить, но не всегда он был бы применим. Метод `rewind` устанавливает указатель в начало набора.

Недостаток библиотеки `generator` заключается в том, что она реализована с помощью продолжений (`continuation`). Во всех имеющихся на сегодняшний день версиях Ruby это требует большого объема вычислений, поэтому, если итераций много, работа заметно замедляется.

8.4. Заключение

Мы подробно рассмотрели массивы, хэши и перечисляемые структуры в общем. Мы установили определенное сходство между массивами и хэшами, объясняемое тем, что в оба класса подмешан модуль `Enumerable`. Но есть и различия. Мы показали, как преобразовать массив в хэш и наоборот, а также узнали несколько интересных способов расширить стандартное поведение.

Мы изучили различные методы обхода структур, например `each_slice` и `each_cons`, а также выяснили, как работают эnumераторы и генераторы.

В главе 9 мы продолжим изучение высокоуровневых структур данных. Не все они входят в ядро Ruby или в стандартные библиотеки. Речь пойдет о множествах, стеках, очередях, деревьях и графах.

Глава 9. Более сложные структуры данных

.....
*Графическое представление данных абстрагирует банки памяти
любого компьютера. Невообразимая сложность. Лучи света,
протянувшиеся в не-пространстве разума, скопления
и созвездия данных. Как гаснущие огни большого города.*
Вильям Гибсон

Есть, конечно, более сложные и интересные структуры данных, чем массивы и хэши. Некоторые из тех, с которыми мы познакомимся в этой главе, имеют прямую или косвенную поддержку в Ruby, другие приходится программировать самостоятельно. К счастью, Ruby упрощает создание нестандартных структур данных.

Математические множества можно, как мы видели, моделировать с помощью массивов. Но в последних версиях Ruby есть также класс `Set`, который хорошо поддерживает эту структуру.

Стеки и очереди – две весьма распространенные в информатике структуры данных. В первом издании этой книги им было уделено чрезмерно много внимания. Для тех, кого интересуют общие вопросы, я оставил кое-какой материал; для остальных есть немало великолепных книг по структурам данных и алгоритмам.

Деревья полезны для сортировки, поиска и просто представления иерархических данных. Мы рассмотрим двоичные деревья и сделаем несколько замечаний о деревьях более высокой степени.

Граф – это обобщение понятия дерева. Граф представляет собой множество вершин, соединенных ребрами, причем с каждым ребром может быть связан вес или направление. Они полезны для решения многих задач, в том числе при анализе сетей и организации знаний.

Но самыми простыми структурами являются множества. С них мы и начнем.

9.1. Множества

Мы уже видели, что некоторые методы класса `Array` позволяют использовать массивы для представления математических множеств. Однако для написания более строгого и компактного кода в Ruby есть также класс `Set`, который скрывает от программиста большую часть деталей реализации.

Чтобы получить в свое распоряжение класс `Set`, достаточно написать:

```
require 'set'
```

При этом также добавляется метод `to_set` в модуль `Enumerable`, так что любой перечисляемый объект становится возможно преобразовать в множество.

Создать новое множество нетрудно. Метод `{ }` работает почти так же, как для хэшей. Метод `new` принимает в качестве необязательных параметров перечисляемый объект и блок. Если блок задан, то он выступает в роли «препроцессора» для списка (подобно операции `map`).

```
s1 = Set[3,4,5]           # В математике обозначается {3,4,5}.
arr = [3,4,5]
s2 = Set.new(arr)         # То же самое.
s3 = Set.new(arr) { |x| x.to_s } # Множество строк, а не чисел.
```

9.1.1. Простые операции над множествами

Для объединения множеств служит метод `union` (синонимы `|` и `+`):

```
x = Set[1,2,3]
y = Set[3,4,5]

a = x.union(y)           # Set[1,2,3,4,5]
b = x | y                # То же самое.
c = x + y                # То же самое.
```

Пересечение множеств вычисляется методом `intersection` (синоним `&`):

```
x = Set[1,2,3]
y = Set[3,4,5]

a = x.intersection(y)    # Set[3]
b = x & y                 # То же самое.
```

Унарный минус обозначает разность множеств; мы обсуждали эту операцию в разделе 8.1.9.

```
diff = Set[1,2,3] - Set[3,4,5]    # Set[1,2]
```

Принадлежность элемента множеству проверяют методы `member?` или `include?`, как для массивов. Напомним, что порядок операндов противоположен принятому в математике.

```
Set[1,2,3].include?(2)    # true
Set[1,2,3].include?(4)    # false
Set[1,2,3].member?(4)     # false
```

Чтобы проверить, является ли множество пустым, мы вызываем метод `empty?`, как и в случае с массивами. Метод `clear` очищает множество, то есть удаляет из него все элементы.

```
s = Set[1,2,3,4,5,6]
s.empty?                  # false
s.clear
s.empty?                  # true
```

Можно проверить, является ли одно множество подмножеством, собственным подмножеством или надмножеством другого.


```

x = Set[3,4,5]
y = Set[3,4]

x.subset?(y)      # false
y.subset?(x)      # true
y.proper_subset?(x) # true
x.subset?(x)      # true
x.proper_subset?(x) # false
x.superset?(y)    # true

```

Метод `add` (синоним `<<`) добавляет в множество один элемент и обычно возвращает его в качестве значения. Метод `add?` возвращает `nil`, если такой элемент уже присутствовал в множестве. Метод `merge` полезен, если надо добавить сразу несколько элементов. Все они, конечно, могут изменить состояние вызывающего объекта. Метод `replace` работает так же, как в случае со строкой или массивом.

Наконец, два множества можно сравнить на равенство интуитивно очевидным способом:

```
Set[3,4,5] == Set[5,4,3]    # true
```

9.1.2. Более сложные операции над множествами

Разумеется, можно обойти множество, но (как и для хэшей) не ожидайте какого-то определенного порядка появления элементов, потому что множества по сути своей неупорядочены, и Ruby не гарантирует никакой последовательности. (Временами можно получить повторяющиеся, ожидаемые результаты, но полагаться на это неразумно.)

```

s = Set[1,2,3,4,5]
s.each {|x| puts x; break }    # Выводится: 5

```

Метод `classify` подобен методу `partition`, но с разбиением на несколько частей; он послужил источником идеи для реализации нашей версии метода `classify` в разделе 8.3.3.

```

files = Set.new(Dir["*"])
hash = files.classify do |f|
  if File.size(f) <= 10_000
    :small
  elsif File.size(f) <= 10_000_000
    :medium
  else
    :large
  end
end

big_files = hash[:large]    # big_files - это Set.

```

Метод `divide` аналогичен, но вызывает блок, чтобы выяснить «степень общности» элементов, и возвращает множество, состоящее из множеств.

Если «арность» (число аргументов) блока равна 1, то метод выполняет вызовы вида `block.call(a) == block.call(b)`, чтобы определить, принадлежат ли `a` и `b`

одному подмножеству. Если «арность» равна 2, для той же цели выполняются вызовы вида `block.call(a,b)`.

Например, следующий блок (с «арностью» 1) разбивает множество на два подмножества, одно из которых содержит четные числа, а другое – нечетные:

```
require 'set'
numbers = Set{1,2,3,4,5,6,7,8,9,0}
set = numbers.divide{|i| i % 2}
p set # => #<Set: {#<Set: {5, 1, 7, 3, 9}>, #<Set: {0, 6, 2, 8, 4}>}>
```

Вот еще один, несколько искусственный пример. Простыми числами-близнецами называются простые числа, отличающиеся на 2 (например, 11 и 13); все прочие называются одиночными (например, 23). Следующий код разбивает множество на группы, помещая числа-близнецы в одно и то же подмножество. В данном случае применяется блок с «арностью» 2:

```
primes = Set{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
set = primes.divide{|i,j| (i-j).abs == 2}
# set is: #<Set: {#<Set: {23}>, #<Set: {11, 13}>,
#          #<Set: {17, 19}>, #<Set: {5, 7, 3}>,
#          #<Set: {2}>, #<Set: {29, 31}>}>
# Более компактно: {{23},{11,13},{17,19},{5,7,3},{2},{29,31}}
```

На мой взгляд, этот метод труден для понимания; я рекомендую пользоваться методом `classify`, более простым и интуитивно очевидным.

Важно понимать, что класс `Set` не всегда требует, чтобы параметр или операнд также был множеством (если вам это кажется странным, вспомните обсуждение «утилизации» в главе 1). На самом деле большая часть методов данного класса принимает в качестве параметра любой перечисляемый объект. Считайте, что так и задумано.

Есть и другие методы, которые применяются в частности к множествам (в том числе все методы из модуля `Enumerable`). Я не стану рассматривать здесь такие методы, как `flatten`. Дополнительную информацию можно найти на сайте <http://ruby-doc.org/> или в любом другом справочном руководстве.

9.2. Стеки и очереди

Стеки и очереди – это первые из встретившихся нам структур, которые, строго говоря, не встроены в Ruby. Иными словами, в Ruby нет классов `Stack` и `Queue`, в отличие от `Array` и `Hash` (впрочем, класс `Queue` есть в библиотеке `thread.rb`, которую мы еще будем рассматривать).

И все же в некотором смысле они встроены в Ruby. Ведь класс `Array` реализует все, что нужно для того, чтобы рассматривать его как стек или очередь.

Стек организован по принципу «последним пришел, первым обслужен» (LIFO – last-in first-out). Традиционный пример – стопка подносов на подпружиненной подставке в кафетерии: подносы кладутся сверху и сверху же снимаются.

Над стеком можно выполнять ограниченный набор операций. Как минимум это операции заталкивания (`push`) и выталкивания (`pop`), то есть помещения в стек и извлечения из него. Обычно также предоставляется способ проверить, пуст ли

стек, и исследовать верхний элемент, не извлекая его из стека. Но никогда реализация не позволяет получить доступ к элементу в середине стека.

Как же реализовать стек на базе массива, если к элементам массива можно обращаться в произвольном порядке, а стек таким свойством не обладает? Ответ прост. Стек – более абстрактная структура, чем массив. Он является стеком лишь до тех пор, пока мы обращаемся с ним как с таковым. В тот момент, когда вы пытаетесь обратиться к элементу недопустимым образом, стек перестает быть стеком.

Но можно без труда определить класс `Stack` так, что к элементам можно будет обращаться только законно. И мы покажем, как это сделать.

Стоит отметить, что во многих алгоритмах стек применяется как основа элегантного рекурсивного решения. Причина станет ясна, если чуточку подумать. При вызове функции или метода параметры заталкиваются в системный стек и вытаскиваются из него при возврате. Таким образом, рекурсивный алгоритм просто подменяет явно определенный пользователем стек системным. Что лучше? Зависит от того, какое значение вы придадите понятности программы, ее эффективности и другим аспектам.

Очередь организована по принципу «первым пришел, первым обслужен» (FIFO – first-in first out). Аналогом может служить очередь за билетами в театр: вновь подходящие становятся в конец очереди, а те, кто пришел раньше, обслуживаются первыми. В программировании очереди используются реже, чем стеки.

Очереди полезны в системах реального времени, когда события нужно обрабатывать в порядке возникновения. Находят они применение и в ситуации «производитель–потребитель» (особенно в многопоточных программах и многозадачных средах). Неплохой пример – очередь к принтеру: задания на печать помещаются в один конец и ожидают, пока не будут извлечены с другого конца.

Две основные операции над очередью называются «поместить» (`enqueue`) и «извлечь» (`dequeue`). Им соответствуют методы `unpush` и `shift` в классе `Array`.

Отметим, что метод `unshift` может использоваться в сочетании с `shift` при реализации массива, но никак не очереди, поскольку `unshift` добавляет элемент в тот же конец массива, из которого `shift` его удаляет. С помощью различных комбинаций этих методов можно реализовать и стек, и очередь, но рассматривать все возможные сочетания мы не будем.

На этом мы закончим введение в стеки и очереди. Самое время рассмотреть некоторые примеры.

9.2.1. Более строгая реализация стека

Мы обещали показать, как можно сделать стек защищенным от некорректного доступа. Выполняем обещание! Вот пример простого класса, который хранит внутри себя массив и управляет доступом к этому массиву. (Есть и другие способы, например делегирование, но описанная реализация проста и прекрасно работает.)

```
class Stack
```

```
  def initialize
    @store = []
  end
```

```
def push(x)
  @store.push x
end

def pop
  @store.pop
end

def peek
  @store.last
end

def empty?
  @store.empty?
end

end
```

Мы добавили одну операцию, которая для массивов не определена; метод `peek` возвращает элемент, находящийся на вершине стека, не выталкивая его.

Нижеследующие примеры подтверждают адекватность такого определения класса.

9.2.2. Обнаружение несбалансированных скобок

В силу самой природы употребления различного вида скобок в выражениях проверить корректность написания можно с помощью стека. При открытии каждого следующего уровня вложенности скобок стек растёт. Как только встречается закрывающая скобка, соответствующий элемент выталкивается из стека. Если при обнаружении закрывающей скобки в стеке ничего не оказалось или, наоборот, выражение уже закончилось, а в стеке что-то осталось, значит, выражение записано неверно.

```
def paren_match(str)
  stack = Stack.new
  lsym = "{[(<"
  rsym = "})]>"
  str.each_byte do |byte|
    sym = byte.chr
    if lsym.include? sym
      stack.push(sym)
    elsif rsym.include? sym
      top = stack.peak
      if lsym.index(top) != rsym.index(sym)
        return false
      else
        stack.pop
      end
    end
    # Игнорируем символы, отличные от скобок...
  end
end
```

```

end
# Убедимся, что стек пуст...
return stack.empty?
end

str1 = "(((a+b))*((c-d)-(e*f)))"
str2 = "[[(a-(b-c))], [[x,y]]]"

paren_match str1      # false
paren_match str2      # true

```

Наличие вложенности естественным образом наводит на мысль о применении стека. Чуть сложнее распознать несбалансированные теги в HTML- или XML-документе. Лексемы состоят из нескольких символов, но логическая структура задачи остается той же самой. Вот еще типичные примеры задач, требующих стека: преобразование выражений из инфиксной формы в постфиксную (и наоборот), вычисление постфиксного выражения (как делается в виртуальной машине Java и многих других интерпретаторах) и вообще любая задача, имеющая рекурсивное решение. В следующем разделе мы немного поговорим о связи между стеком и рекурсией.

9.2.3. Стек и рекурсия

В качестве примера изоморфизма, существующего между стеком и рекурсией, рассмотрим классическую задачу о Ханойской башне.

По легенде где-то далеко на востоке существует старинный храм. Обитающие в нем монахи заняты решением единственной задачи: перемещением дисков с одного шеста на другой с соблюдением определенных правил. Первоначально на первом шесте было 64 диска. Когда все диски будут перемещены, настанет конец света.

Попутно разоблачим миф. Похоже, что на самом деле эту задачу впервые сформулировал французский математик Эдуард Люка в 1883 году, и никаких истоков в восточной культуре она не имеет. Сам Люка называл ее «Ханойской башней».

Так что если вас пугает конец света, можете успокоиться. Да и в любом случае для перемещения 64 дисков потребуется $2^{64}-1$ ходов. Небольшой расчет на калькуляторе покажет, что монахи будут заняты своим делом несколько миллионов лет.

Однако вернемся к правилам игры. (Сформулируем их, хотя эту загадку знал уже самый первый студент самого первого факультета информатики.) Имеется шест, на который надето несколько дисков; назовем его исходным. Мы хотим переместить все диски на целевой шест, используя еще один вспомогательный шест как место промежуточного хранения. Проблема в том, что за один ход можно перемещать только один диск; при этом нельзя класть больший диск на меньший.

В следующем примере приведено решение этой задачи с использованием стека. Мы ограничились тремя дисками, потому что для перемещения 64 компьютеру потребовались бы века.

```

def towers(list)
  while !list.empty?
    n, src, dst, aux = list.pop
    if n == 1
      puts "Перемещаем диск с #{src} на #{dst}"
    end
  end
end

```

```

else
    list.push [n-1, aux, dst, src]
    list.push [1, src, dst, aux]
    list.push [n-1, src, aux, dst]
end
end
end

```

```

list = []
list.push([3, "a", "c", "b"])

```

```
towers(list)
```

Вот что напечатает эта программа:

```

Перемещаем диск с a на c
Перемещаем диск с a на b
Перемещаем диск с c на b
Перемещаем диск с a на c
Перемещаем диск с b на a
Перемещаем диск с b на c
Перемещаем диск с a на c

```

Конечно, классическое решение этой задачи рекурсивно. Но, как мы отмечали, тесная связь между обоими алгоритмами не должна вызывать удивления, так как для рекурсии применяется невидимый системный стек.

```

def towers(n, src, dst, aux)
  if n==1
    puts "Перемещаем диск с #{src} на #{dst}"
  else
    towers(n-1, src, aux, dst)
    towers(1, src, dst, aux)
    towers(n-1, aux, dst, src)
  end
end

```

```
towers(3, "a", "c", "b")
```

Печатается точно такой же результат. Возможно, вам будет интересно знать, что мы «закомментировали» предложения, осуществляющие вывод, и сравнили время работы. Никому не говорите, но рекурсивное решение оказалось в два раза быстрее!

9.2.4. Более строгая реализация очереди

Мы определим очередь примерно так же, как стек. Если вы хотите защититься от некорректного доступа к структуре данных, рекомендуем поступать аналогично.

```

class Queue

  def initialize
    @store = []
  end

```

```
def enqueue(x)
  @store << x
end

def dequeue
  @store.shift
end

def peek
  @store.first
end

def length
  @store.length
end

def empty?
  @store.empty?
end

end
```

Отметим, что класс `Queue` имеется в библиотеке `thread` для поддержки многопоточных программ. Имеется даже вариант `SizedQueue` для организации очереди ограниченного размера.

В упомянутых классах методы имеют короткие имена: `enq` и `deq`. У них есть также синонимы `push` и `pop`, что лично мне кажется неоправданным. Это структура данных FIFO, а не LIFO, то есть именно очередь, а не стек.

Разумеется, класс `Queue` в библиотеке `thread.rb` безопасен относительно потоков. Если вы хотите реализовать такой же класс `Stack`, рекомендую взять `Queue` в качестве отправной точки. Потребуется внести не так много изменений.

В первом издании книги был длинный пример, демонстрирующий работу с очередями. Но, как и некоторые примеры, касающиеся стеков, он был исключен ради экономии места.

9.3. Деревья

.....
*Я не увижу никогда, наверное,
Поэму столь прекрасную как дерево.
Джойс Килмер, «Деревья»*

В информатике идея дерева считается интуитивно очевидной (правда, изображаются они обычно с корнем наверху, а листьями снизу). И немудрено, ведь в повседневной жизни мы постоянно сталкиваемся с иерархическими данными:

генеалогическое дерево, организационная схема компании, структура каталогов на диске.

Терминология, описывающая деревья, богата, но понять ее легко. Элементы дерева называются *узлами*; верхний или самый первый узел называется *корневым* или *корнем*. У узла могут быть *потомки*, расположенные ниже него, а непосредственные потомки называются *детьми* или *дочерними узлами*. Узел, не имеющий потомков, называется *листовым* или просто *листом*. Поддерево состоит из некоторого узла и всех его потомков. Посещение всех узлов дерева (например, с целью распечатки) называется *обходом дерева*.

Нас будут интересовать в основном двоичные деревья, хотя в принципе узел может иметь произвольное число детей. Мы покажем, как создавать дерево, добавлять в него узлы и выполнять обход. Рассмотрим также некоторые реальные задачи, при решении которых используются деревья.

Отметим, что во многих языках, например в С или Pascal, деревья реализуются с помощью адресных указателей. Но в Ruby (как и в Java) указателей нет, вместо них используются ссылки на объекты, что ничуть не хуже, а иногда даже лучше.

9.3.1. Реализация двоичного дерева

Ruby позволяет реализовать двоичное дерево разными способами. Например, хранить значения узлов можно в массиве. Но мы применим более традиционный подход, характерный для кодирования на С, только указатели заменим ссылками на объекты.

Что нужно для описания двоичного дерева? Понятно, что в каждом узле должен быть атрибут для хранения данных. Кроме того, в каждом узле должны быть атрибуты для ссылки на левое и правое поддерево. Еще необходим способ вставить новый узел в дерево и получить хранящуюся в дереве информацию. Для этого нам потребуются два метода.

В первом дереве, которое мы рассмотрим, эти методы будут реализованы неортодоксальным способом. Позже мы расширим класс `Tree`.

В некотором смысле дерево определяется алгоритмом вставки и способом обхода. В нашем первом примере (листинг 9.1) метод `insert` будет осуществлять поиск в дереве «в ширину», то есть сверху вниз и слева направо. При этом глубина дерева растет относительно медленно, и оно всегда оказывается сбалансированным. Методу вставки соответствует итератор `traverse`, который обходит дерево в том же порядке.

Листинг 9.1. Вставка и обход дерева в ширину

```
class Tree

  attr_accessor :left
  attr_accessor :right
  attr_accessor :data

  def initialize(x=nil)
    @left = nil
```

```
@right = nil
@data = x
end

def insert(x)
  list = []
  if @data == nil
    @data = x
  elsif @left == nil
    @left = Tree.new(x)
  elsif @right == nil
    @right = Tree.new(x)
  else
    list << @left
    list << @right
    loop do
      node = list.shift
      if node.left == nil
        node.insert(x)
        break
      else
        list << node.left
      end
      if node.right == nil
        node.insert(x)
        break
      else
        list << node.right
      end
    end
  end
end

def traverse()
  list = []
  yield @data
  list << @left if @left != nil
  list << @right if @right != nil
  loop do
    break if list.empty?
    node = list.shift
    yield node.data
    list << node.left if node.left != nil
    list << node.right if node.right != nil
  end
end

items = [1, 2, 3, 4, 5, 6, 7]
```

```
tree = Tree.new
items.each {|x| tree.insert(x)}

tree.traverse {|x| print "#{x} "}
print "\n"
```

Печатается "1 2 3 4 5 6 7 "

Такое дерево не слишком интересно. Но оно годится в качестве введения и фундамента, на котором можно возводить здание.

9.3.2. Сортировка с помощью двоичного дерева

Двоичное дерево позволяет эффективно реализовать сортировку произвольных данных. (Правда, если данные уже отсортированы, оно вырождается в обычный связанный список.) Причина ясна: при каждом сравнении мы исключаем половину мест, в которые можно поместить новый узел.

Хотя в настоящее время такой способ сортировки применяется редко, знать о нем не повредит. Код в листинге 9.2 основан на предыдущем примере.

Листинг 9.2. Сортировка с помощью двоичного дерева

```
class Tree

  # Предполагается, что определения взяты из предыдущего примера...

  def insert(x)
    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new x
      else
        @left.insert x
      end
    else
      if @right == nil
        @right = Tree.new x
      else
        @right.insert x
      end
    end
  end

  def inorder()
    @left.inorder {|y| yield y} if @left != nil
    yield @data
    @right.inorder {|y| yield y} if @right != nil
  end
end
```

```

def preorder()
  yield @data
  @left.preorder {|y| yield y} if @left != nil
  @right.preorder {|y| yield y} if @right != nil
end

def postorder()
  @left.postorder {|y| yield y} if @left != nil
  @right.postorder {|y| yield y} if @right != nil
  yield @data
end
end

```

```

items = [50, 20, 80, 10, 30, 70, 90, 5, 14,
         28, 41, 66, 75, 88, 96]

```

```
tree = Tree.new
```

```
items.each {|x| tree.insert(x)}
```

```

tree.inorder {|x| print x, " "}
print "\n"
tree.preorder {|x| print x, " "}
print "\n"
tree.postorder {|x| print x, " "}
print "\n"

```

```
# Печатается:
```

```
# 5 10 14 20 28 30 41 50 66 70 75 80 88 90 96
```

```
# 50 20 10 5 14 30 28 41 80 70 66 75 90 88 96
```

```
# 5 14 10 28 41 30 20 66 75 70 88 96 90 80 50
```

9.3.3. Использование двоичного дерева как справочной таблицы

Пусть дерево уже отсортировано. Тогда оно может служить прекрасной справочной таблицей; например, для поиска в сбалансированном дереве, содержащем миллион узлов, понадобится не более 20 сравнений (глубина дерева равна логарифму числа узлов по основанию 2). Чтобы поиск был осмысленным, предположим, что в каждом узле хранится не какое-то одно значение, а ключ и ассоциированные с ним данные.

Почти всегда лучше использовать в качестве справочной таблицы хэш или даже таблицу во внешней базе данных. Но все равно приведем код:

```
class Tree
```

```
  # Предполагается, что определения взяты из предыдущего примера...
```

```

def search(x)
  if self.data == x
    return self
  end
end

```

```

    elsif x < self.data
      return left ? left.search(x) : nil
    else
      return right ? right.search(x) : nil
    end
  end
end

keys = [50, 20, 80, 10, 30, 70, 90, 5, 14,
        28, 41, 66, 75, 88, 96]

tree = Tree.new

keys.each {|x| tree.insert(x)}

s1 = tree.search(75)  # Возвращает ссылку на узел, содержащий 75...

s2 = tree.search(100) # Возвращает nil (не найдено).
```

9.3.4. Преобразование дерева в строку или массив

С помощью тех же приемов, которые применяются для обхода дерева, мы можем преобразовать его в строку или в массив. Ниже мы выполняем обход во внутреннем порядке, хотя подошел бы и любой другой способ:

```

class Tree

  # Предполагается, что определения взяты из предыдущего примера...

  def to_s
    "[" +
      if left then left.to_s + "," else "" end +
      data.inspect +
      if right then "," + right.to_s else "" end + "]"
  end

  def to_a
    temp = []
    temp += left.to_a if left
    temp << data
    temp += right.to_a if right
    temp
  end

end

items = %w[bongo grimace monoid jewel plover nexus synergy]
tree = Tree.new
items.each {|x| tree.insert x}
```

```
str = tree.to_a * ","
# str is now "bongo,grimace,jewel,monoid,nexus,plover,synergy"
arr = tree.to_a
# arr равно:
# ["bongo",["grimace",["jewel"],"monoid",["nexus"],"plover",
# ["synergy"]]]]
```

Отметим, что глубина вложенности получающегося массива равна глубине дерева с корнем в том узле, с которого мы начали обход. Чтобы получить плоский массив, можете воспользоваться методом `flatten`.

9.4. Графы

Графом называется множество вершин, произвольным образом соединенных друг с другом. (Дерево – частный случай графа.) Не будем слишком углубляться в эту тему, поскольку теория и терминология весьма сложны. Очень скоро мы перешли бы от информатики в область чистой математики.

И все же у графов есть немало практических приложений. Возьмите обычную дорожную карту, на которой города соединены скоростными магистралями, или печатную плату. То и другое удобно представлять в виде графов. Компьютерную сеть тоже можно описать в терминах теории графов, будь то локальная сеть из нескольких десятков машин или весь Интернет, насчитывающий миллионы узлов.

Под графом мы обычно понимаем *неориентированный граф*. Попросту говоря, в нем не проставлены стрелки на соединительных линиях; две вершины либо соединены, либо нет. Между тем в *ориентированном графе (орграфе)* могут быть «улицы с односторонним движением»; из того, что вершина *x* соединена с вершиной *y*, не следует, что верно и обратное. Наконец, во *взвешенном графе* ребрам можно назначать веса. Например, вес может выражать «расстояние» между вершинами. Мы ограничимся только этими основными видами графов; интересующегося читателя отсылаем к многочисленным учебникам информатики и математики.

В Ruby, как и во многих других языках, граф можно представить разными способами, например в виде настоящей сети взаимосвязанных объектов или в виде матрицы, в которой хранятся ребра графа. Мы рассмотрим оба способа и на примерах покажем, как можно манипулировать графами.

9.4.1. Реализация графа в виде матрицы смежности

Нижеприведенный пример основан на двух предыдущих. В листинге 9.3 неориентированный граф реализован в виде матрицы смежности с помощью класса `Array` (см. раздел 8.1.26). Это нужно для того, чтобы новые элементы по умолчанию получали значение 0. Также мы унаследовали классу `TriMatrix` (см. раздел 8.1.7), чтобы получить нижнетреугольную матрицу.

Листинг 9.3. Матрица смежности

```
class LowerMatrix < TriMatrix

  def initialize
```

```
@store = ZArray.new
end
```

```
end
```

```
class Graph
```

```
  def initialize(*edges)
    @store = LowerMatrix.new
    @max = 0
    for e in edges
      e[0], e[1] = e[1], e[0] if e[1] > e[0]
      @store[e[0],e[1]] = 1
      @max = [@max, e[0], e[1]].max
    end
  end
```

```
  def [](x,y)
    if x > y
      @store[x,y]
    elsif x < y
      @store[y,x]
    else
      0
    end
  end
```

```
  def []=(x,y,v)
    if x > y
      @store[x,y]=v
    elsif x < y
      @store[y,x]=v
    else
      0
    end
  end
```

```
  def edge? x,y
    x,y = y,x if x < y
    @store[x,y]==1
  end
```

```
  def add x,y
    @store[x,y] = 1
  end
```

```
  def remove x,y
    x,y = y,x if x < y
```

```

@store[x,y] = 0
if (degree @max) == 0
  @max -= 1
end
end

def vmax
  @max
end

def degree x
  sum = 0
  0.upto @max do |i|
    sum += self[x,i]
  end
  sum
end

def each_vertex
  (0..@max).each {|v| yield v}
end

def each_edge
  for v0 in 0..@max
    for v1 in 0..v0-1
      yield v0,v1 if self[v0,v1]==1
    end
  end
end

end

mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

# Напечатать степени всех вершин: 2 3 2 3.
mygraph.each_vertex {|v| puts mygraph.degree(v)}

# Напечатать список ребер.
mygraph.each_edge do |a,b|
  puts "#{a},#{b}"
end

# Удалить одно ребро.
mygraph.remove 1,3

# Напечатать степени всех вершин: 2 2 2 2.
mygraph.each_vertex {|v| p mygraph.degree v}

```

Отметим, что приведенная выше реализация не допускает ребер, ведущих из некоторого узла в него же. Кроме того, два узла могут быть соединены только одним ребром.

Мы позволяем задать начальный состав ребер, передавая пары в конструктор. Кроме того, можно добавлять и удалять ребра, а также проверять наличие ребра между двумя вершинами. Метод `vmax` возвращает вершину с наибольшим номером. Метод `degree` вычисляет степень указанной вершины, то есть количество исходящих из нее ребер.

Наконец, имеются два итератора `each_vertex` и `each_edge`, которые позволяют перебрать все вершины и все ребра соответственно.

9.4.2. Является ли граф связным?

Не все графы связные. Иногда нет способа «добраться из одной точки в другую», то есть между двумя вершинами нет никакого пути, составленного из ребер. *Связность* – это важное свойство графа, его надо уметь вычислять. В связном графе любая вершина достижима из любой другой.

Не будем объяснять принцип работы алгоритма, интересующийся читатель может найти описание в любой книге по дискретной математике. Но в листинге 9.4 приведена его реализация на Ruby.

Листинг 9.4. Выяснение того, является ли граф связным

```
class Graph

  def connected?
    x = vmax
    k = [x]
    l = [x]
    for i in 0..@max
      l << i if self[x,i]==1
    end
    while !k.empty?
      y = k.shift
      # Теперь ищем все ребра (y,z).
      self.each_edge do |a,b|
        if a==y || b==y
          z = a==y ? b : a
          if !l.include? z
            l << z
            k << z
          end
        end
      end
    end
    if l.size < @max
      false
    else
      true
    end
  end
end
```



```

end
end
end

```

```
mygraph = Graph.new([0,1], [1,2], [2,3], [3,0], [1,3])
```

```
puts mygraph.connected?      # true
```

```
puts mygraph.euler_path?     # true
```

```
mygraph.remove 1,2
```

```
mygraph.remove 0,3
```

```
mygraph.remove 1,3
```

```
puts mygraph.connected?      # false
```

```
puts mygraph.euler_path?     # false
```

В примере упомянут метод `euler_path?`, с которым мы еще не встречались. Он определен в разделе 9.4.4.

Можно было бы усовершенствовать этот алгоритм, так чтобы он находил все связные компоненты несвязного графа. Но мы не станем этого делать.

9.4.3. Есть ли в графе эйлеров цикл?

.....
*Нет такой отрасли математики, сколь угодно абстрактной,
 которая со временем не нашла бы применения в реальной жизни.*
Николай Лобачевский

Иногда нужно знать, есть ли в графе *эйлеров цикл*. Термин связан с математиком Леонардом Эйлером, который основал область топологии, занимающуюся этим вопросом. (Графы, обладающие таким свойством, называют иногда *уникурсивными*, поскольку их можно нарисовать не отрывая карандаша от бумаги и не проходя дважды по одному и тому же ребру.)

В немецком городе Кенигсберг был остров посередине реки. С двумя берегами остров связывало семь мостов. Горожане хотели знать, можно ли обойти город так, чтобы побывать на каждом мосту ровно один раз и вернуться в исходную точку. В 1735 году Эйлер доказал, что это невозможно. Эта классическая задача стала первой проблемой теории графов.

Как часто бывает в жизни, решение кажется простым, когда оно найдено. Оказалось, что для существования в графе эйлерова цикла необходимо и достаточно, чтобы все вершины имели четную степень. Вот короткий код, проверяющий выполнение этого свойства:

```
class Graph
```

```
  def euler_circuit?
```

```

return false if !connected?
for i in 0..@max
  return false if degree(i) % 2 != 0
end
true
end

end

mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

flag1 = mygraph.euler_circuit?    # false

mygraph.remove 1,3

flag2 = mygraph.euler_circuit?    # true

```

9.4.4. Есть ли в графе эйлеров путь?

Эйлеров путь и эйлеров цикл – разные вещи. Слово «цикл» подразумевает, что нужно вернуться в исходную точку. А наличие пути предполагает, что нужно лишь посетить каждую вершину ровно один раз. В следующем фрагменте демонстрируется это различие:

```

class Graph

  def euler_path?
    return false if !connected?
    odd=0
    each_vertex do |x|
      if degree(x) % 2 == 1
        odd += 1
      end
    end
    odd <= 2
  end

end

mygraph = Graph.new([0,1],[1,2],[1,3],[2,3],[3,0])

flag1 = mygraph.euler_circuit?    # false
flag2 = mygraph.euler_path?      # true

```

9.4.5. Инструменты для работы с графами в Ruby

В сообществе пользователей Ruby известно несколько таких инструментов. Они в большинстве своем имеют ограниченную функциональность и предназначены для работы с ориентированными или неориентированными графами. Поищите

эти инструменты в архиве RAA (<http://raa.ruby-lang.org>) или на сайте Rubyforge (<http://rubyforge.org>). Называются они как-то вроде RubyGraph, RGraph, GraphR и по большей части еще не достигли зрелости.

Если вас интересует великолепный пакет GraphViz, который умеет представлять сложные графы в виде изображений или программ на языке Postscript, то к нему есть по меньшей мере два работоспособных интерфейса. Есть даже элемент управления GnomeGraphwidget, который, если верить документации, «можно использовать в приложениях Ruby Gnome для генерирования, визуализации и манипулирования графами». Мы его, впрочем, не изучали; пока еще не вышла даже официальная альфа-версия.

Короче говоря, потребность в подобных инструментах может возникнуть. В таком случае я призываю вас написать собственный инструмент или присоединиться к какому-нибудь существующему проекту. Если работать с графами станет достаточно просто, то мы еще будем недоумевать, как раньше могли без них обходиться!..

9.5. Заключение

Мы познакомились с классом Set в Ruby, а также с несколькими примерами «доморощенных» структур данных. Мы видели, как можно создавать сложные структуры данных путем наследования существующему классу или ограниченного делегирования, когда экземпляр существующего класса инкапсулируется в новой структуре. Также были рассмотрены изобретательные способы хранения данных, применения различных структур данных и создания итераторов для обхода таких структур.

Мы уделили внимание стекам и очередям и способам их использования для решения задач. Кроме того, окинули беглым взглядом деревья и графы.

В следующей главе мы снова займемся манипулированием данными. Но если до сих пор нас интересовало хранение объектов в основной памяти, то теперь мы обратимся к вспомогательной памяти, то есть файлам (и вводу/выводу в общем), базам данных и устойчивым объектам.

Глава 10. Ввод/вывод и хранение данных

.....
На чистом диске можно искать бесконечно.
Томас Б. Стил младший

Вычислительные машины хороши для вычислений. В этой тавтологии больше смысла, чем кажется на первый взгляд. Если бы программа только потребляла процессорное время да изредка обращалась к оперативной памяти, жизнь была бы куда проще.

Но от компьютера, занятого исключительно собой, мало толку. Рано или поздно придется получать информацию извне и отправлять ее во внешний мир, и вот тут-то жизнь перестает казаться медом.

Есть несколько факторов, затрудняющих ввод/вывод. Во-первых, ввод и вывод – совершенно разные вещи, но обычно мы мысленно объединяем их. Во-вторых, операции ввода/вывода столь же разнообразны, как и мир насекомых.

История знает такие устройства, как магнитные барабаны, перфоленты, магнитные ленты, перфокарты и телетайпы. Некоторые имели механические детали, другие были электромагнитными от начала и до конца. Одни позволяли только считывать информацию, другие – только записывать, а третьи умели делать и то и другое. Часть записывающих устройств позволяла стирать данные, другая – нет. Одни были принципиально последовательными, другие допускали произвольный доступ. На иных устройствах информация хранилась постоянно, другие были энергозависимыми. Некоторые требовали человеческого вмешательства, другие – нет. Есть устройства символьного и блочного ввода/вывода. На некоторых блочных устройствах можно хранить только блоки постоянной длины, другие допускают и переменную длину блока. Одни устройства надо периодически опрашивать, другие управляются прерываниями. Прерывания можно реализовать аппаратно, программно или смешанным образом. Есть буферизованный и небуферизованный ввод/вывод. Бывает ввод/вывод с отображением на память и канальный, а с появлением таких операционных систем, как UNIX, мы узнали об устройствах ввода/вывода, отображаемых на элементы файловой системы. Программировать ввод/вывод доводилось на машинном языке, на языке ассемблера и на языках высокого уровня. В некоторые языки механизм ввода/вывода жестко встроен, другие вообще не включают ввод/вывод в спецификацию языка. Приходилось выполнять ввод/вывод с помощью подходящего драйвера или уровня абстракции и без него.

Возможно, все это показалось вам хаотичным нагромождением разнородных фактов; если так, вы абсолютно правы!.. Отчасти сложность проистекает из самой

природы ввода/вывода, отчасти это результат компромиссов, принятых при проектировании, а отчасти следствие наследия прошлых лет, устоявшихся традиций и особенностей различных языков и операционных систем.

Ввод/вывод в Ruby сложен, потому что он сложен в принципе. Но мы старались описать его как можно понятнее и показать, где и когда стоит применять различные приемы.

В основе системы ввода/вывода в Ruby лежит класс `IO`, который определяет поведение всех операций ввода/вывода. С ним тесно связан (и наследует ему) класс `File`. В класс `File` вложен класс `Stat`, инкапсулирующий различные сведения о файле (например, разрешения и временные штампы). Методы `stat` и `lstat` возвращают объекты типа `File::Stat`.

В модуле `FileTest` также есть методы, позволяющие опрашивать практически те же свойства. Он подмешивается к классу `File`, но может использоваться и самостоятельно.

Наконец, методы ввода/вывода есть и в модуле `Kernel`, который подмешивается к классу `Object` (предку всех объектов, включая и классы). Это простые процедуры, которыми мы пользовались на протяжении всей книги, не думая о том, от имени какого объекта они вызываются. По умолчанию они настроены на стандартный ввод и стандартный вывод.

Поначалу может показаться, что это хаотическое хитросплетение перекрывающейся функциональности. Но в каждый момент времени вам необходима лишь небольшая часть всего каркаса.

На более высоком уровне Ruby предлагает механизмы, позволяющие сделать объекты устойчивыми. Метод `marshal` реализует простую сериализацию объектов; он лежит в основе более изощренной библиотеки `PStore`. Мы включили в эту главу и библиотеку `DBM`, хотя она умеет работать только со строками.

На самом высоком уровне возможен интерфейс с системами управления базами данных, например `MySQL` или `Oracle`. Эта тема настолько сложна, что ей можно было бы посвятить одну или даже несколько книг. Мы ограничимся лишь кратким введением. В некоторых случаях будут даны ссылки на архивы в сети.

10.1. Файлы и каталоги

Под *файлом* мы обычно, хотя и не всегда, понимаем файл на диске. Концепция файла в Ruby, как и в других языках, – это полезная абстракция. Говоря «*каталог*», мы подразумеваем каталог или папку в смысле, принятом в UNIX и Windows.

Класс `File` тесно связан с классом `IO`, которому наследует. Класс `Dir` связан с ним не так тесно, но мы решили рассмотреть файлы и каталоги вместе, поскольку между ними имеется концептуальная связь.

10.1.1. Открытие и закрытие файлов

Метод класса `File.new`, создающий новый объект `File`, также открывает файл. Первым параметром, естественно, является имя файла.

Необязательный второй параметр называется *строкой указания режима*; он говорит, как нужно открывать файл – для чтения, для записи и т.д. (Строка указаний

режима не имеет ничего общего с разрешениями.) По умолчанию предполагается режим "r", то есть только чтение. Ниже показано, как открывать файлы для чтения и записи.

```
file1 = File.new("one")      # Открыть для чтения.
file2 = File.new("two", "w") # Открыть для записи.
```

Есть также разновидность метода new, принимающая три параметра. В этом случае второй параметр задает начальные разрешения для файла (обычно записывается в виде восьмеричной константы), а третий представляет собой набор флагов, объединенных союзом ИЛИ. Флаги обозначаются константами, например: File::CREAT (создать файл, если он еще не существует) и File::RDONLY (открыть только для чтения). Такая форма используется редко.

```
file = File.new("three", 0755, File::CREAT|File::WRONLY)
```

В виде любезности по отношению к операционной системе и среде исполнения всегда закрывайте открытые вами файлы. Если файл был открыт для записи, то это не просто вежливость, а способ предотвратить потерю данных. Для закрытия файла предназначен метод close:

```
out = File.new("captains.log", "w")
# Обработка файла...
out.close
```

Имеется также метод open. В простейшей форме это синоним new:

```
trans = File.open("transactions", "w")
```

Но методу open можно также передать блок, и это более интересно. Если блок задан, то ему в качестве параметра передается открытый файл. Файл остается открытым на протяжении всего времени нахождения в блоке и автоматически закрывается при выходе из него. Пример:

```
File.open("somefile", "w") do |file|
  file.puts "Строка 1"
  file.puts "Строка 2"
  file.puts "Третья и последняя строка"
end
# Теперь файл закрыт.
```

Это изящный способ обеспечить закрытие файла по завершении работы с ним. К тому же при такой записи весь код обработки файла сосредоточен в одном месте.

10.1.2. Обновление файла

Чтобы открыть файл для чтения и записи, достаточно добавить знак плюс (+) в строку указания режима (см. раздел 10.1.1):

```
f1 = File.new("file1", "r+")
# Чтение/запись, от начала файла.

f2 = File.new("file2", "w+")
# Чтение/запись; усесть существующий файл или создать новый.

f3 = File.new("file3", "a+")
# Чтение/запись; перейти в конец существующего файла или создать новый.
```

10.1.3. Дописывание в конец файла

Чтобы дописать данные в конец существующего файла, нужно задать строку указания режима "a" (см. раздел 10.1.1):

```
logfile = File.open("captains_log", "a")
# Добавить строку в конец и закрыть файл.
logfile.puts "Stardate 47824.1: Our show has been canceled."
logfile.close
```

10.1.4. Прямой доступ к файлу

Для чтения из файла в произвольном порядке, а не последовательно, можно воспользоваться методом `seek`, который класс `File` наследует от `IO`. Проще всего перейти к байт в указанной позиции. Номер позиции отсчитывается от начала файла, причем самый первый байт находится в позиции 0.

```
# myfile содержит строку: abcdefghi
file = File.new("myfile")
file.seek(5)
str = file.gets                # "fghi"
```

Если все строки в файле имеют одинаковую длину, то можно перейти сразу в начало нужной строки:

```
# Предполагается, что все строки имеют длину 20.
# Строка N начинается с байта (N-1)*20
file = File.new("fixedlines")
file.seek(5*20)                # Шестая строка!
```

Для выполнения относительного поиска воспользуйтесь вторым параметром. Константа `IO::SEEK_CUR` означает, что смещение задано относительно текущей позиции (и может быть отрицательным):

```
file = File.new("somefile")
file.seek(55)                  # Позиция 55.
file.seek(-22, IO::SEEK_CUR)   # Позиция 33.
file.seek(47, IO::SEEK_CUR)    # Позиция 80.
```

Можно также искать относительно конца файла, в таком случае смещение может быть только отрицательным:

```
file.seek(-20, IO::SEEK_END)   # Двадцать байтов от конца файла.
```

Есть еще и третья константа `IO::SEEK_SET`, но это значение по умолчанию (поиск относительно начала файла).

Метод `tell` возвращает текущее значение позиции в файле, у него есть синоним `pos`:

```
file.seek(20)
pos1 = file.tell               # 20
file.seek(50, IO::SEEK_CUR)
pos2 = file.pos                # 70
```

Метод `rewind` устанавливает указатель файла в начало. Его название («обратная перемотка») восходит ко временам использования магнитных лент.

Для выполнения прямого доступа файл часто открывается в режиме обновления (для чтения и записи). Этот режим обозначается знаком + в начале строки указания режима (см. раздел 10.1.2).

10.1.5. Работа с двоичными файлами

Когда-то давно программисты на языке C включали в строку указания режима символ "b" для открытия файла как двоичного. (Вопреки распространенному заблуждению, это относилось и к ранним версиям UNIX.) Как правило, эту возможность все еще поддерживают ради совместимости, но сегодня с двоичными файлами работать не так сложно, как раньше. Строка в Ruby может содержать двоичные данные, а для чтения двоичного файла не нужно никаких специальных действий.

Исключение составляет семейство операционных систем Windows, в которых различие все еще имеет место. Основное отличие двоичных файлов от текстовых на этой платформе состоит в том, что в двоичном режиме конец строки не преобразуется в один символ перевода строки, а представляется в виде пары «возврат каретки – перевод строки». Еще одно важно отличие – интерпретация символа control-Z как конца файла в текстовом режиме:

```
# Создать файл (в двоичном режиме).
File.open("myfile", "wb") { |f| f.syswrite("12345\0326789\r") }
# Обратите внимание на восьмеричное 032 (^Z).
```

```
# Читать как двоичный файл.
str = nil
File.open("myfile", "rb") { |f| str = f.sysread(15) }
puts str.size           # 11
```

```
# Читать как текстовый файл.
str = nil
File.open("myfile", "r") { |f| str = f.sysread(15) }
puts str.size           # 5
```

В следующем фрагменте показано, что на платформе Windows символ возврата каретки не преобразуется в двоичном режиме:

```
# Входной файл содержит всего одну строку: Строка 1.
file = File.open("data")
line = file.readline # "Строка 1.\n"
puts "#{line.size} символов."           # 10 символов.
file.close
```

```
file = File.open("data", "rb")
line = file.readline # "Строка 1.\r\n"
puts "#{line.size} символов."           # 11 символов.
file.close
```

Отметим, что упомянутый в коде метод binmode переключает поток в двоичный режим. После переключения вернуться в текстовый режим невозможно.

```
file = File.open("data")
file.binmode
```



```
line = file.readline           # "Строка 1.\r\n"
puts "#{line.size} символов." # 11 символов.
file.close
```

При необходимости выполнить низкоуровневый ввод/вывод можете воспользоваться методами `sysread` и `syswrite`. Первый принимает в качестве параметра число подлежащих чтению байтов, второй принимает строку и возвращает число записанных байтов. (Если вы начали читать из потока методом `sysread`, то никакие другие методы использовать не следует. Результаты могут быть непредсказуемы.)

```
input = File.new("infile")
output = File.new("outfile")
instr = input.sysread(10);
bytes = output.syswrite("Это тест.")
```

Отметим, что метод `sysread` возбуждает исключение `EOFError` при попытке вызвать его, когда достигнут конец файла (но не в том случае, когда конец файла встретился в ходе успешной операции чтения). Оба метода возбуждают исключение `SystemCallError` при возникновении ошибки ввода/вывода.

При работе с двоичными данными могут оказаться полезны метод `pack` из класса `Array` и метод `unpack` из класса `String`.

10.1.6. Блокировка файлов

В тех операционных системах, которые поддерживают такую возможность, метод `flock` класса `File` блокирует или разблокирует файл. Вторым параметром может быть одна из констант `File::LOCK_EX`, `File::LOCK_NB`, `File::LOCK_SH`, `File::LOCK_UN` или их объединение с помощью оператора ИЛИ. Понятно, что многие комбинации не имеют смысла; чаще всего употребляется флаг, задающий неблокирующий режим.

```
file = File.new("somefile")

file.flock(File::LOCK_EX) # Исключительная блокировка; никакой другой
                          # процесс не может обратиться к файлу.
file.flock(File::LOCK_UN) # Разблокировать.

file.flock(File::LOCK_SH) # Разделяемая блокировка (другие
                          # процессы могут сделать то же самое).
file.flock(File::LOCK_UN) # Разблокировать.
```

```
locked = file.flock(File::LOCK_EX | File::LOCK_NB)
# Пытаемся заблокировать файл, но не приостанавливаем программу, если
# не получилось; в таком случае переменная locked будет равна false.
```

Для семейства операционных систем Windows эта функция не реализована.

10.1.7. Простой ввод/вывод

Вы уже знакомы с некоторыми методами ввода/вывода из модуля `Kernel`; мы вызывали их без указания вызывающего объекта. К ним относятся функции `gets` и `puts`, а также `print`, `printf` и `p` (последний вызывает метод объекта `inspect`, чтобы распечатать его в понятном для нас виде).

Но есть и другие методы, которые следует упомянуть для полноты. Метод `putc` выводит один символ. (Парный метод `getc` не реализован в модуле `kernel` по техническим причинам, однако он есть у любого объекта класса `IO`). Если параметром является объект `String`, то печатается первый символ строки.

```
putc(?\\n)    # Вывести символ новой строки.
putc("X")    # Вывести букву X.
```

Интересный вопрос: куда направляется вывод, если эти методы вызываются без указания объекта? Начнем с того, что в среде исполнения Ruby определены три глобальные константы, соответствующие трем стандартным потокам ввода/вывода, к которым мы привыкли в UNIX. Это `STDIN`, `STDOUT` и `STDERR`. Все они имеют тип `IO`.

Имеется также глобальная переменная `$stdout`, именно в нее направляется весь вывод, формируемый методами из `kernel`. Она инициализирована значением `STDOUT`, так что данные отправляются на стандартный вывод, как и следовало ожидать. В любой момент переменной `$stdout` можно присвоить другое значение, являющееся объектом `IO`.

```
diskfile = File.new("foofile", "w")
puts "Привет..."      # Выводится на stdout.
$stdout = diskfile
puts "Пока!"            # Выводится в файл "foofile".
diskfile.close
$stdout = STDOUT        # Восстановление исходного значения.
puts "Это все."         # Выводится на stdout.
```

Помимо метода `gets` в модуле `kernel` есть методы ввода `readline` и `readlines`. Первый аналогичен `gets` в том смысле, что возбуждает исключение `EOFError` при попытке читать за концом файла, а не просто возвращает `nil`. Последний эквивалентен методу `IO.readlines` (то есть считывает весь файл в память).

Откуда мы получаем ввод? Есть переменная `$stdin`, которая по умолчанию равна `STDIN`. Точно так же существует поток стандартного вывода для ошибок (`$stderr`, по умолчанию равен `STDERR`).

Еще имеется интересный глобальный объект `ARGV`, представляющий конкатенацию всех файлов, указанных в командной строке. Это не объект класса `File`, хотя и напоминает таковой. По умолчанию ввод связан именно с этим объектом, если в командной строке задан хотя бы один файл.

```
# Прочитать все файлы, а затем вывести их.
puts ARGV.read
# А при таком способе более экономно расходуется память:
while ! ARGV.eof?
  puts ARGV.readline
end
# Пример: ruby cat.rb file1 file2 file3
```

При чтении из стандартного ввода (`STDIN`) методы `kernel` не вызываются. Поэтому можно обойти (или не обходить) `ARGV`, как показано ниже:

```
# Прочитать строку из стандартного ввода.
str1 = STDIN.gets
# Прочитать строку из ARGV.
```

```
str2 = ARGF.gets
# А теперь снова из стандартного ввода.
str3 = STDIN.gets
```

10.1.8. Буферизованный и небуферизованный ввод/вывод

В некоторых случаях Ruby осуществляет буферизацию самостоятельно. Рассмотрим следующий фрагмент:

```
print "Привет... "
sleep 10
print "Пока!\n"
```

Если запустить эту программу, то вы увидите, что сообщения «Привет» и «Пока» появляются одновременно, после завершения `sleep`. При этом первое сообщение не завершается символом новой строки.

Это можно исправить, вызвав метод `flush` для опустошения буфера вывода. В данном случае вывод идет в поток `$stdout` (подразумеваемый по умолчанию для всех методов `Kernel`, которые занимаются выводом). И поведение оказывается ожидаемым, то есть первое сообщение появляется раньше второго.

```
print "Привет... "
STDOUT.flush
sleep 10
print "Пока!\n"
```

Буферизацию можно отключить (или включить) методом `sync=`, а метод `sync` позволяет узнать текущее состояние.

```
buf_flag = $stdout.sync # true
STDOUT.sync = false
buf_flag = STDOUT.sync # false
```

Есть еще по крайней мере один низкий уровень буферизации, который не виден. Если метод `getc` возвращает символ и продвигает вперед указатель файла или потока, то метод `ungetc` возвращает символ назад в поток.

```
ch = mystream.getc # ?A
mystream.ungetc(?C)
ch = mystream.getc # ?C
```

Тут следует иметь в виду три вещи. Во-первых, только что упомянутая буферизация не имеет отношения к механизму буферизации, о котором мы говорили выше в этом разделе. Иными словами, предложение `sync=false` не отключает ее. Во-вторых, вернуть в поток можно только один символ; при попытке вызвать метод `ungetc` несколько раз будет возвращен только символ, прочитанный последним. И, в-третьих, метод `ungetc` не работает для принципиально небуферизуемых операций (например, `sysread`).

10.1.9. Манипулирование правами владения и разрешениями на доступ к файлу

Вопрос о владении файлами и разрешениях сильно зависит от платформы. Как правило, в системе UNIX функций больше, чем предоставляет Ruby, а на других платформах многие возможности не реализованы.

Для определения владельца и группы файла (это целые числа) класс `File::Stat` предоставляет методы экземпляра `uid` и `gid`:

```
data = File.stat("somefile")
owner_id = data.uid
group_id = data.gid
```

В классе `File::Stat` есть также метод экземпляра `mode`, который возвращает текущий набор разрешений для файла.

```
perms = File.stat("somefile").mode
```

В классе `File` имеется метод класса и экземпляра `chown`, позволяющий изменить идентификаторы владельца и группы. Метод класса принимает произвольное число файлов. Если идентификатор не нужно изменять, можно передать `nil` или `-1`.

```
uid = 201
gid = 10
File.chown(uid, gid, "alpha", "beta")
f1 = File.new("delta")
f1.chown(uid, gid)
f2 = File.new("gamma")
f2.chown(nil, gid)      # Оставить идентификатор владельца без изменения.
```

Разрешения можно изменить с помощью метода `chmod` (у него также есть два варианта: метод класса и метод экземпляра). Традиционно разрешения представляют восьмеричным числом, хотя это и не обязательно.

```
File.chmod(0644, "epsilon", "theta")
f = File.new("eta")
f.chmod(0444)
```

Процесс всегда работает от имени какого-то пользователя (возможно, `root`), поэтому с ним связан *идентификатор пользователя* (мы сейчас говорим о действующем идентификаторе). Часто нужно знать, имеет ли данный пользователь право читать, писать или исполнять данный файл. В классе `File::Stat` есть методы экземпляра для получения такой информации.

```
info = File.stat("/tmp/secrets")
rflag = info.readable?
wflag = info.writable?
xflag = info.executable?
```

Иногда нужно отличить действующий идентификатор пользователя от реального. На этот случай предлагаются методы экземпляра `readable_real?`, `writable_real?` и `executable_real?`.

```
info = File.stat("/tmp/secrets")
rflag2 = info.readable_real?
wflag2 = info.writable_real?
xflag2 = info.executable_real?
```

Можно сравнить владельца файла с действующим идентификатором пользователя (и идентификатором группы) текущего процесса. В классе `File::Stat` для этого есть методы `owned?` и `growned?`.

Отметим, что многие из этих методов можно найти также в модуле `FileTest`:

```
rflag = FileTest::readable?("pentagon_files")
# Прочие методы: writable? executable? readable_real?
# writable_real? executable_real? owned? grpowned?
# Отсутствуют здесь: uid gid mode.
```

Маска `umask`, ассоциированная с процессом, определяет начальные разрешения для всех созданных им файлов. Стандартные разрешения `0777` логически пересекаются (AND) с отрицанием `umask`, то есть биты, поднятые в маске, «маскируются» или сбрасываются. Если вам удобнее, можете представлять себе эту операцию как вычитание (без заимания). Следовательно, если задана маска `022`, то все файлы создаются с разрешениями `0755`.

Получить или установить маску можно с помощью метода `umask` класса `File`. Если ему передан параметр, то он становится новым значением маски (при этом метод возвращает старое значение).

```
File.umask(0237)          # Установить umask.
current_umask = File.umask # 0237
```

Некоторые биты режима файла (например, бит фиксации – sticky bit) не имеют прямого отношения к разрешениям. Эта тема обсуждается в разделе 10.1.12.

10.1.10. Получение и установка временных штампов

С каждым файлом на диске связано несколько *временных штампов* (в разных операционных системах они различны). Ruby понимает три таких штампа: время модификации (когда в последний раз было изменено содержимое файла), время доступа (когда в последний раз файл читался) и время изменения (когда в последний раз была изменена информация о файле, хранящаяся в каталоге).

Получить эту информацию можно тремя разными способами, хотя все они дают один и тот же результат.

Методы `mtime`, `atime` и `ctime` класса `File` возвращают временные штампы, не требуя предварительного открытия файла или даже создания объекта `File`.

```
t1 = File.mtime("somefile")
# Thu Jan 04 09:03:10 GMT-6:00 2001
t2 = File.atime("somefile")
# Tue Jan 09 10:03:34 GMT-6:00 2001
t3 = File.ctime("somefile")
# Sun Nov 26 23:48:32 GMT-6:00 2000
```

Если файл, представленный экземпляром `File`, уже открыт, то можно воспользоваться методами этого экземпляра.

```
myfile = File.new("somefile")
t1 = myfile.mtime
t2 = myfile.atime
t3 = myfile.ctime
```

А если имеется экземпляр класса `File::Stat`, то и у него есть методы, позволяющие получить ту же информацию:

```
myfile = File.new("somefile")
info = myfile.stat
```

```
t1 = info.mtime
t2 = info.atime
t3 = info.ctime
```

Отметим, что объект `File::Stat` возвращается методом класса (или экземпляра) `stat` из класса `File`. Метод класса `lstat` (или одноименный метод экземпляра) делает то же самое, но возвращает информацию о состоянии самой ссылки, а не файла, на который она ведет. Если имеется цепочка из нескольких ссылок, то метод следует по ней и возвращает информацию о предпоследней (которая уже указывает на настоящий файл).

Для изменения времени доступа и модификации применяется метод `utime`, которому можно передать несколько файлов. Время можно создать в виде объекта `Time` или числа секунд, прошедших с точки отсчета.

```
today = Time.now
yesterday = today - 86400
File.utime(today, today, "alpha")
File.utime(today, yesterday, "beta", "gamma")
```

Поскольку оба временных штампа изменяются одновременно, то при желании оставить один без изменения его сначала следует получить и сохранить.

```
mtime = File.mtime("delta")
File.utime(Time.now, mtime, "delta")
```

10.1.11. Проверка существования и получение размера файла

Часто необходимо знать, существует ли файл с данным именем. Это позволяет объяснить метод `exist?` из модуля `FileTest`:

```
flag = FileTest::exist?("LochNessMonster")
flag = FileTest::exists?("UFO")
# exists? является синонимом exist?
```

Понятно, что такой метод не может быть методом экземпляра `File`, поскольку после создания объекта файл уже открыт. В классе `File` мог бы быть метод класса с именем `exist?`, но его там нет.

С вопросом о том, существует ли файл, связан другой вопрос: а есть ли в нем какие-нибудь данные? Ведь файл может существовать, но иметь нулевую длину — а это практически равносильно тому, что он отсутствует.

Если нас интересует только, пуст ли файл, то в классе `File::Stat` есть два метода экземпляра, отвечающих на этот вопрос. Метод `zero?` возвращает `true`, если длина файла равна нулю, и `false` в противном случае.

```
flag = File.new("somefile").stat.zero?
```

Метод `size?` возвращает либо размер файла в байтах, если он больше нуля, либо `nil` для файла нулевой длины. Не сразу понятно, почему `nil`, а не `0`. Дело в том, что метод предполагалось использовать в качестве предиката, а значение истинности нуля в Ruby — `true`, тогда для `nil` оно равно `false`.

```
if File.new("myfile").stat.size?
  puts "В файле есть данные."
else
```

```
puts "Файл пуст."
end
```

Методы `zero?` и `size?` включены также в модуль `FileTest`:

```
flag1 = FileTest::zero?("file1")
flag2 = FileTest::size?("file2")
```

Далее возникает следующий вопрос: «Каков размер файла?» Мы уже видели, что для непустого файла метод `size?` возвращает длину. Но если мы применяем его не в качестве предиката, то значение `nil` только путает.

В классе `File` есть метод класса (но не метод экземпляра) для ответа на этот вопрос. Метод экземпляра с таким же именем имеется в классе `File::Stat`.

```
size1 = File.size("file1")
size2 = File.stat("file2").size
```

Чтобы получить размер файла в блоках, а не в байтах, можно обратиться к методу `blocks` из класса `File::Stat`. Результат, конечно, зависит от операционной системы. (Метод `blksize` сообщает размер блока операционной системы.)

```
info = File.stat("somefile")
total_bytes = info.blocks * info.blksize
```

10.1.12. Опрос специальных свойств файла

У файла есть много свойств, которые можно опросить. Мы перечислим в этом разделе те встроенные методы, для которых не нашлось другого места. Почти все они являются предикатами.

Читая этот раздел (да и большую часть этой главы), помните о двух вещах. Во-первых, так как класс `File` подмешивает модуль `FileTest`, то любую проверку, для которой требуется вызывать метод, квалифицированный именем модуля, можно также выполнить, обратившись к методу экземпляра любого файлового объекта. Во-вторых, функциональность модуля `FileTest` и объекта `File::Stat` (возвращаемого методом `stat` или `lstat`) сильно перекрывается. В некоторых случаях есть целых три разных способа вызвать по сути один и тот же метод. Мы не будем каждый раз приводить все варианты.

В некоторых операционных системах устройства подразделяются на блочные и символьные. Файл может ссылаться как на то, так и на другое, но не на оба сразу. Методы `blockdev?` и `chardev?` из модуля `FileTest` проверяют тип устройства:

```
flag1 = FileTest::chardev?("/dev/hdisk0") # false
flag2 = FileTest::blockdev?("/dev/hdisk0") # true
```

Иногда нужно знать, ассоциирован ли данный поток с терминалом. Метод `tty?` класса `IO` (синоним `isatty`) дает ответ на этот вопрос:

```
flag1 = STDIN.tty? # true
flag2 = File.new("diskfile").isatty # false
```

Поток может быть связан с каналом (`pipe`) или сокетом. В модуле `FileTest` есть методы для опроса этих условий:

```
flag1 = FileTest::pipe?(myfile)
flag2 = FileTest::socket?(myfile)
```

Напомним, что каталог – это разновидность файла. Поэтому нужно уметь отличать каталоги от обычных файлов, для чего предназначены два метода из модуля `FileTest`:

```
file1 = File.new("/tmp")
file2 = File.new("/tmp/myfile")
test1 = file1.directory?      # true
test2 = file1.file?           # false
test3 = file2.directory?      # false
test4 = file2.file?           # true
```

В классе `File` есть также метод класса `ftype`, который сообщает вид потока; одноименный метод экземпляра находится в классе `File::Stat`. Этот метод возвращает одну из следующих строк: `file`, `directory`, `blockSpecial`, `characterSpecial`, `fifo`, `link` или `socket` (строка `fifo` относится к каналу).

```
this_kind = File.ftype("/dev/hdisk0")      # "blockSpecial"
that_kind = File.new("/tmp").stat.ftype     # "directory"
```

В маске, описывающей режим файла, можно устанавливать или сбрасывать некоторые биты. Они не имеют прямого отношения к битам, обсуждавшимся в разделе 10.1.9. Речь идет о битах `set-group-id`, `set-user-id` и бите фиксации (`sticky bit`). Для каждого из них есть метод в модуле `FileTest`.

```
file = File.new("somefile")
info = file.stat
sticky_flag = info.sticky?
setgid_flag = info.setgid?
setuid_flag = info.setuid?
```

К дисковому файлу могут вести *символические* или *физические ссылки* (в тех операционных системах, где такой механизм поддерживается). Чтобы проверить, является ли файл символической ссылкой на другой файл, обратитесь к методу `symlink?` из модуля `FileTest`. Для подсчета числа физических ссылок на файл служит метод `nlink` (он есть только в классе `File::Stat`). Физическая ссылка неотличима от обычного файла – это просто файл, для которого есть несколько имен и записей в каталоге.

```
File.symlink("yourfile", "myfile")          # Создать ссылку
is_sym = FileTest::symlink?("myfile")      # true
hard_count = File.new("myfile").stat.nlink  # 0
```

Отметим попутно, что в предыдущем примере мы воспользовались методом класса `symlink` из класса `File` для создания символической ссылки.

В редких случаях может понадобиться информация о файле еще более низкого уровня. В классе `File::Stat` есть еще три метода экземпляра, предоставляющих такую информацию. Метод `dev` возвращает целое число, идентифицирующее устройство, на котором расположен файл. Метод `rdev` возвращает целое число, описывающее тип устройства, а для дисковых файлов метод `ino` возвращает номер первого индексного узла, занятого файлом.

```
file = File.new("diskfile")
info = file.stat
```



```
device = info.dev
devtype = info.rdev
inode = info.ino
```

10.1.13. Каналы

Ruby поддерживает разные способы читать из канала и писать в него. Метод класса `IO.popen` открывает канал и связывает с возвращенным объектом стандартные ввод и вывод процесса. Часто с разными концами канала работают разные потоки, но в примере ниже запись и чтение осуществляет один и тот же поток:

```
check = IO.popen("spell","r+")
check.puts("'T was brillig, and the slithy toves")
check.puts("Did gyre and gimble in the wabe.")
check.close_write
list = check.readlines
list.collect! { |x| x.chomp }
# list равно %w[brillig gimble gyre slithy toves wabe]
```

Отметим, что вызов `close_write` обязателен, иначе мы никогда не достигнем конца файла при чтении из канала.

Существует также блочная форма:

```
File.popen("/usr/games/fortune") do |pipe|
  quote = pipe.gets
  puts quote
  # На чистом диске можно искать бесконечно. - Том Стил.
end
```

Если задана строка `--`, то запускается новый экземпляр Ruby. Если при этом задан еще и блок, то он работает в двух разных процессах, как в результате разветвления (`fork`); блоку в процессе-потомке передается `nil`, а в процессе-родителе – объект `IO`, с которым связан стандартный ввод или стандартный вывод.

```
IO.popen("--") do |mypipe|
  if mypipe
    puts "Я родитель: pid = #{Process.pid}"
    listen = mypipe.gets
    puts listen
  else
    puts "Я потомок: pid = #{Process.pid}"
  end
end
```

```
# Печатается:
#   Я родитель: pid = 10580
#   Я потомок: pid = 10582
```

Метод `pipe` возвращает также два конца канала, связанных между собой. В следующем примере мы создаем два потока, один из которых передает сообщение другому (то самое сообщение, которое Сэмюэль Морзе впервые послал по телеграфу). Если вы не знаете, что такое потоки, обратитесь к главе 3.

```

pipe = IO.pipe
reader = pipe[0]
writer = pipe[1]

str = nil
thread1 = Thread.new(reader,writer) do |reader,writer|
  # writer.close_write
  str = reader.gets
  reader.close
end

thread2 = Thread.new(reader,writer) do |reader,writer|
  # reader.close_read
  writer.puts("What hath God wrought?")
  writer.close
end

thread1.join
thread2.join

puts str          # What hath God wrought?

```

10.1.14. Специальные операции ввода/вывода

В Ruby можно выполнять низкоуровневые операции ввода/вывода. Мы только упомянем о существовании таких методов; если вы собираетесь ими пользоваться, имейте в виду, что некоторые машиннозависимы (различаются даже в разных версиях UNIX).

Метод `ioctl` принимает два аргумента: целое число, определяющее операцию, и целое число либо строку, представляющую параметр этой операции.

Метод `fcntl` также предназначен для низкоуровневого управления файловыми потоками системно зависимым образом. Он принимает такие же параметры, как `ioctl`.

Метод `select` (в модуле `kernel`) принимает до четырех параметров. Первый из них — массив дескрипторов для чтения, а остальные три необязательны (массив дескрипторов для записи, дескрипторов для ошибок и величина тайм-аута). Если на каком-то из устройств, дескрипторы которых заданы в первом массиве, оказываются новые данные для чтения или какое-то из устройств, дескрипторы которых перечислены во втором массиве, готово к выполнению записи, метод возвращает массив из трех элементов, каждый из которых в свою очередь является массивом, где указаны дескрипторы устройств, готовых к выполнению ввода/вывода.

Метод `syscall` из модуля `kernel` принимает по меньшей мере один целочисленный параметр (а всего до девяти целочисленных или строковых параметров). Первый параметр определяет выполняемую операцию ввода/вывода.

Метод `fileno` возвращает обычный файловый дескриптор, ассоциированный с потоком ввода/вывода. Это наименее системно зависимый из всех перечисленных выше методов.

```
desc = $stderr.fileno    # 2
```

10.1.15. Неблокирующий ввод/вывод

«За кулисами» Ruby предпринимает согласованные меры, чтобы операции ввода/вывода не блокировали выполнение программы. В большинстве случаев для управления вводом/выводом можно пользоваться потоками — один поток может выполнить блокирующую операцию, а второй будет продолжать работу.

Это немного противоречит интуиции. Потоки Ruby работают в том же процессе, они не являются платформенными потоками. Быть может, вам кажется, что блокирующая операция ввода/вывода должна приостанавливать весь процесс, а значит, и все его потоки. Это не так — Ruby аккуратно управляет вводом/выводом прозрачно для программиста.

Но если вы все же хотите включить неблокирующий режим ввода/вывода, такая возможность есть. Небольшая библиотека `io/nonblock` предоставляет методы чтения и установки для объекта IO, представляющего блочное устройство:

```
require 'io/nonblock'

# ...

test = mysock.nonblock?           # false

mysock.nonblock = true            # Отключить блокирующий режим.
# ...
mysock.nonblock = false           # Снова включить его.

mysock.nonblock { some_operation(mysock) }
# Выполнить some_operation в неблокирующем режиме.

mysock.nonblock(false) { other_operation(mysock) }
# Выполнить other_operation в блокирующем режиме.
```

10.1.16. Применение метода `readpartial`

Метод `readpartial` появился сравнительно недавно с целью упростить ввод/вывод при определенных условиях. Он может использоваться с любыми потоками, например с сокетами.

Параметр «максимальная длина» (`max length`) обязателен. Если задан параметр `buffer`, то он должен ссылаться на строку, в которой будут храниться данные.

```
data = sock.readpartial(128)      # Читать не более 128 байтов.
```

Метод `readpartial` игнорирует установленный режим блокировки ввода/вывода. Он может блокировать программу, но лишь при выполнении следующих условий: буфер объекта IO пуст, в потоке ничего нет и поток еще не достиг конца файла.

Таким образом, если в потоке есть данные, то `readpartial` не будет блокировать программу. Он читает не более указанного числа байтов, а если байтов оказалось меньше, то прочитает их и продолжит выполнение.

Если в потоке нет данных, но при этом достигнут конец файла, то `readpartial` немедленно возбуждает исключение `EOFError`.

Если вызов блокирующий, то он ожидает, пока не произойдет одно из двух событий: придут новые данные или обнаружится конец файла. Если поступают данные, метод возвращает их вызывающей программе, а в случае обнаружения конца файла возбуждает исключение `EOFError`.

При вызове метода `sysread` в блокирующем режиме он ведет себя похоже на `readpartial`. Если буфер пуст, их поведение вообще идентично.

10.1.17. Манипулирование путевыми именами

Основными методами для работы с путевыми именами являются методы класса `File`: `dirname` и `basename`; они работают, как одноименные команды UNIX, то есть возвращают имя каталога и имя файла соответственно. Если вторым параметром методу `basename` передана строка с расширением имени файла, то это расширение исключается.

```
str = "/home/dave/podbay.rb"
dir = File.dirname(str)           # "/home/dave"
file1 = File.basename(str)       # "podbay.rb"
file2 = File.basename(str, ".rb") # "podbay"
```

Хотя это методы класса `File`, на самом деле они просто манипулируют строками.

Упомянем также метод `File.split`, который возвращает обе компоненты (имя каталога и имя файла) в массиве из двух элементов:

```
info = File.split(str)           # ["/home/dave", "podbay.rb"]
```

Метод класса `expand_path` преобразует путевое имя в абсолютный путь. Если операционная система понимает сокращения `~` и `~user`, то они тоже учитываются.

```
Dir.chdir("/home/poole/personal/docs")
abs = File.expand_path("../misc") # "/home/poole/misc"
```

Если передать методу `path` открытый файл, то он вернет путевое имя, по которому файл был открыт.

```
file = File.new("../foo/bar")
name = file.path                # "../foo/bar"
```

Константа `File::Separator` равна символу, применяемому для разделения компонентов путевого имени (в Windows это обратная косая черта, а в UNIX – прямая косая черта). Имеется также синоним `File::SEPARATOR`.

Метод класса `join` использует этот разделитель для составления полного путевого имени из переданного списка компонентов:

```
path = File.join("usr", "local", "bin", "someprog")
# path равно "usr/local/bin/someprog".
# Обратите внимание, что в начало имени разделитель не добавляется!
```

Не думайте, что методы `File.join` и `File.split` взаимно обратны, – это не так.

10.1.18. Класс `Pathname`

Следует знать о существовании стандартной библиотеки `pathname`, которая предоставляет класс `Pathname`. В сущности, это обертка вокруг классов `Dir`, `File`, `FileTest`

и `FileUtils`, поэтому он комбинирует многие их функции логичным и интуитивно понятным способом.

```
path = Pathname.new("/home/hal")
file = Pathname.new("file.txt")
p2 = path + file
path.directory?      # true
path.file?           # false
p2.directory?        # false
p2.file?             # true

parts = path2.split   # [Путевое имя:/home/hal, Путевое имя:file.txt]
ext = path2.extname   # .txt
```

Как и следовало ожидать, имеется ряд вспомогательных методов. Метод `root?` пытается выяснить, относится ли данный путь к корневому каталогу, но его можно «обмануть», так как он просто анализирует строку, не обращаясь к файловой системе. Метод `parent?` возвращает путевое имя родительского каталога данного пути. Метод `children` возвращает непосредственных потомков каталога, заданного своим путевым именем; в их число включаются как файлы, так и каталоги, но рекурсивного спуска не производится.

```
p1 = Pathname.new("/")      # Странно, но допустимо.
p1.root?                    # true
p2 = Pathname.new("/home/poole")
p3 = p2.parent              # Путевое имя:/home
items = p2.children         # Массив объектов Pathname
                                # (все файлы и каталоги, являющиеся
                                # непосредственными потомками p2).
```

Как и следовало ожидать, методы `relative` и `absolute` пытаются определить, является ли путь относительным или абсолютным (проверяя, есть ли в начале имени косая черта):

```
p1 = Pathname.new("/home/dave")
p1.absolute?               # true
p1.relative?               # false
```

Многие методы, например `size`, `unlink` и пр., просто делегируют работу классам `File`, `FileTest` и `FileUtils`; повторно функциональность не реализуется.

Дополнительную информацию о классе `Pathname` вы найдете на сайте [ruby-doc.org](#) или в любом другом справочном руководстве.

10.1.19. Манипулирование файлами на уровне команд

Часто приходится манипулировать файлами так, как это делается с помощью командной строки: копировать, удалять, переименовывать и т.д.

Многие из этих операций реализованы встроенными методами, некоторые находятся в модуле `FileUtils` из библиотеки `fileutils`. Имейте в виду, что раньше функциональность модуля `FileUtils` подмешивалась прямо в класс `File`; теперь эти методы помещены в отдельный модуль.

Для удаления файла служит метод `File.delete` или его синоним `File.unlink`:

```
File.delete("history")
File.unlink("toast")
```

Переименовать файл позволяет метод `File.rename`:

```
File.rename("Ceylon", "SriLanka")
```

Создать ссылку на файл (физическую или символическую) позволяют методы `File.link` и `File.symlink` соответственно:

```
File.link("/etc/hosts", "/etc/hostfile") # Физическая ссылка.
File.symlink("/etc/hosts", "/tmp/hosts") # Символическая ссылка.
```

Файл можно усечь до нулевой длины (или до любой другой), воспользовавшись методом экземпляра `truncate`:

```
File.truncate("myfile", 1000) # Теперь не более 1000 байтов.
```

Два файла можно сравнить с помощью метода `compare_file`. У него есть синонимы `cmp` и `compare_stream`:

```
require "fileutils"

same = FileUtils.compare_file("alpha", "beta") # true
```

Метод `copy` копирует файл в другое место, возможно, с переименованием. У него есть необязательный флаг, говорящий, что сообщения об ошибках нужно направлять на стандартный вывод для ошибок. Синоним — привычное для программистов UNIX имя `cp`.

```
require "fileutils"

# Скопировать файл epsilon в theta с протоколированием ошибок.
FileUtils.copy("epsilon", "theta", true)
```

Файл можно перемещать методом `move` (синоним `mv`). Как и `copy`, этот метод имеет необязательный параметр, включающий вывод сообщений об ошибках.

```
require "fileutils"

FileUtils.move("/tmp/names", "/etc") # Переместить в другой каталог.
FileUtils.move("colours", "colors") # Просто переименовать.
```

Метод `safe_unlink` удаляет один или несколько файлов, предварительно пытаясь сделать их доступными для записи, чтобы избежать ошибок. Если последний параметр равен `true` или `false`, он интерпретируется как флаг, задающий режим вывода сообщений об ошибках.

```
require "fileutils"

FileUtils.safe_unlink("alpha", "beta", "gamma")
# Протоколировать ошибки при удалении следующих двух файлов:
FileUtils.safe_unlink("delta", "epsilon", true)
```

Наконец, метод `install` делает практически то же, что и `syscopy`, но сначала проверяет, что целевой файл либо не существует, либо содержит такие же данные.

```
require "fileutils"
```

```
FileUtils.install("foo.so", "/usr/lib")
# Существующий файл foo.so не будет переписан,
# если он не отличается от нового.
```

Дополнительную информацию о модуле FileUtils см. на сайте ruby-doc.org или в любом другом справочном руководстве.

10.1.20. Ввод символов с клавиатуры

В данном случае мы имеем в виду небуферизованный ввод, когда символ обрабатывается сразу после нажатия клавиши, не дожидаясь, пока будет введена вся строка.

Это можно сделать и в UNIX, и в Windows, но, к сожалению, совершенно по-разному.

Версия для UNIX прямолинейна. Мы переводим терминал в режим прямого ввода (raw mode) и обычно одновременно отключаем эхо-контроль.

```
def getchar
  system("stty raw -echo") # Прямой ввод без эхо-контроля.
  char = STDIN.getc
  system("stty -raw echo") # Восстановить режим терминала.
  char
end
```

На платформе Windows придется написать расширение на C. Пока что альтернативой является использование одной из функций в библиотеке win32API.

```
require 'Win32API'

def getchar
  char = Win32API.new("crt.dll", "_getch", [], 'L').Call
end
```

Поведение в обоих случаях идентично.

10.1.21. Чтение всего файла в память

Чтобы прочитать весь файл в массив, не нужно даже его предварительно открывать. Все сделает метод IO.readlines: откроет файл, прочитает и закроет.

```
arr = IO.readlines("myfile")
lines = arr.size
puts "myfile содержит #{lines} строк."
```

```
longest = arr.collect {|x| x.length}.max
puts "Самая длинная строка содержит #{longest} символов."
```

Можно также воспользоваться методом IO.read (который возвращает одну большую строку, а не массив строк).

```
str = IO.read("myfile")
bytes = str.size
puts "myfile содержит #{bytes} байтов."
```

```
longest=str.collect {|x| x.length}.max # строки - перечисляемые объекты!
puts "Самая длинная строка содержит #{longest} символов."
```

Поскольку класс `IO` является предком `File`, то можно вместо этого писать `File.readlines` и `File.read`.

10.1.22. Построчное чтение из файла

Чтобы читать по одной строке из файла, можно обратиться к методу класса `IO.foreach` или к методу экземпляра `each`. В первом случае файл не нужно явно открывать.

```
# Напечатать все строки, содержащие слово "target".
IO.foreach("somefile") do |line|
  puts line if line =~ /target/
end
```

```
# Другой способ...
file = File.new("somefile")
file.each do |line|
  puts line if line =~ /target/
end
```

Отметим, что `each_line` – синоним `each`.

10.1.23. Побайтное чтение из файла

Для чтения из файла по одному байту служит метод экземпляра `each_byte`. Напомним, что он передает в блок символ (то есть целое число); воспользуйтесь методом `chr`, если хотите преобразовать его в «настоящий» символ.

```
file = File.new("myfile")
e_count = 0
file.each_byte do |byte|
  e_count += 1 if byte == ?e
end
```

10.1.24. Работа со строкой как с файлом

Иногда возникает необходимость рассматривать строку как файл. Что под этим понимается, зависит от конкретной задачи.

Объект определяется прежде всего своими методами. В следующем фрагменте показано, как к объекту `source` применяется итератор; на каждой итерации выводится одна строка. Можете ли вы что-нибудь сказать о типе объекта `source`, глядя на этот код?

```
source.each do |line|
  puts line
end
```

Это могли бы быть как файл, так и строка, содержащая внутри символы новой строки. В таких случаях строку можно трактовать как файл без всякого труда.

В последних версиях Ruby имеется также библиотека `stringio`.

Интерфейс класса `StringIO` практически такой же, как в первом издании этой книги. В нем есть метод доступа `string`, ссылающийся на содержимое самой строки.

```
require 'stringio'
```

```
ios = StringIO.new("abcdefghijkl\nABC\n123")
```



```

ios.seek(5)
ios.puts("xyz")

puts ios.tell           # 8

puts ios.string.dump    # "abcdexyzijkl\nABC\n123"

c = ios.getc
puts "c = #{c}"         # c = 105

ios.ungetc(?w)

puts ios.string.dump    # "abcdexyzwjkl\nABC\n123"

puts "Ptr = #{ios.tell}"

s1 = ios.gets           # "wjkl"
s2 = ios.gets           # "ABC"

```

10.1.25. Чтение данных, встроенных в текст программы

Когда подростком вы учили язык BASIC, копируя программы из журналов, то, наверное, для удобства часто пользовались предложением DATA. Оно позволяло включать информацию прямо в текст программы, но читать ее так, будто она поступает из внешнего источника.

При желании то же самое можно сделать и в Ruby. Директива `__END__` в конце программы говорит, что дальше идут встроенные данные. Их можно читать из глобальной константы `DATA`, которая представляет собой обычный объект IO. (Отметим, что маркер `__END__` должен располагаться с начала строки.)

```

# Распечатать все строки "задом наперед"...
DATA.each_line do |line|
  puts line.reverse
end
__END__
A man, a plan, a canal... Panama!
Madam, I'm Adam.
,siht daer nac uoy fI
.drah oot gnikrow neeb ev'uoy

```

10.1.26. Чтение исходного текста программы

Если вы хотите получить доступ к исходному тексту собственной программы, то можете воспользоваться уже описанным выше трюком (см. раздел 10.1.25).

Глобальная константа `DATA` – это объект класса IO, ссылающийся на данные, которые расположены после директивы `__END__`. Но если выполнить метод `rewind`, то указатель файла будет переустановлен на начало текста программы.

Следующая программа выводит собственный текст, снабжая его номерами строк. Это не очень полезно, но, быть может, вы найдете и другие применения такой техники.

```
DATA.rewind
num = 1
DATA.each_line do |line|
  puts "#{'%03d' % num} #{line}"
  num += 1
end
__END__
```

Отметим, что наличие директивы `__END__` обязательно – без нее к константе `DATA` вообще нельзя обратиться.

10.1.27. Работа с временными файлами

Во многих случаях необходимо работать с файлами, которые по сути своей анонимны. Мы не хотим возиться с присваиванием им имен и проверять, что при этом не возникает конфликтов с существующими файлами. И помнить о том, что такие файлы нужно удалять, тоже не хочется.

Все эти проблемы решает библиотека `Tempfile`. Метод `new` (синоним `open`) принимает базовое имя в качестве строки-затравки и конкатенирует его с идентификатором процесса и уникальным порядковым номером. Необязательный второй параметр – имя каталога, в котором создается временный файл; по умолчанию оно равно значению первой из существующих переменных окружения `TMPDIR`, `TMP` или `TEMP`, а если ни одна из них не задана, то `"/tmp"`.

Возвращаемый объект `io` можно многократно открывать и закрывать на протяжении всей работы программы, а по ее завершении временный файл будет автоматически удален.

У метода `close` есть необязательный флаг; если он равен `true`, то файл удаляется сразу после закрытия (не дожидаясь завершения программы). Метод `path` возвращает полное имя файла, если оно вам по какой-то причине понадобится.

```
require "tempfile"
```

```
temp = Tempfile.new("stuff")
name = temp.path                # "/tmp/stuff17060.0"
temp.puts "Здесь был Вася"
temp.close

# Позже...
temp.open
str = temp.gets                 # "Здесь был Вася"
temp.close(true)               # Удалить СЕЙЧАС.
```

10.1.28. Получение и изменение текущего каталога

Получить имя текущего каталога можно с помощью метода `Dir.pwd` (синоним `Dir.getwd`). Эти имена уже давно употребляются как сокращения от «print working directory» (печатать рабочий каталог) и «get working directory» (получить рабочий каталог). На платформе Windows символы обратной косой черты преобразуются в символы прямой косой черты.

Для изменения текущего каталога служит метод `Dir.chdir`. В Windows в начале строки можно указывать букву диска.

```
Dir.chdir("/var/tmp")
puts Dir.pwd          # "/var/tmp"
puts Dir.getwd        # "/var/tmp"
```

Этот метод также принимает блок в качестве параметра. Если блок задан, то текущий каталог изменяется только на время выполнения блока, а потом восстанавливается первоначальное значение:

```
Dir.chdir("/home")
Dir.chdir("/tmp") do
  puts Dir.pwd          # /tmp
  # Какой-то код...
end
puts Dir.pwd # /home
```

10.1.29. Изменение текущего корня

В большинстве систем UNIX можно изменить «представление» процесса о том, что такое корневой каталог /. Обычно это делается из соображений безопасности перед запуском небезопасной или непротестированной программы. Метод `chroot` делает указанный каталог новым корнем:

```
Dir.chdir("/home/guy/sandbox/tmp")
Dir.chroot("/home/guy/sandbox")
puts Dir.pwd # "/tmp"
```

10.1.30. Обход каталога

Метод класса `foreach` – это итератор, который последовательно передает в блок каждый элемент каталога. Точно так же ведет себя метод экземпляра `each`.

```
Dir.foreach("/tmp") { |entry| puts entry }
```

```
dir = Dir.new("/tmp")
dir.each { |entry| puts entry }
```

Оба фрагмента печатают одно и то же (имена всех файлов и подкаталогов в каталоге /tmp).

10.1.31. Получение содержимого каталога

Метод класса `Dir.entries` возвращает массив, содержащий все элементы указанного каталога:

```
list = Dir.entries("/tmp")      # %w[. .. alpha.txt beta.doc]
```

Как видите, включаются и элементы, соответствующие текущему и родительскому каталогу. Если они вам не нужны, придется отфильтровать их вручную.

10.1.32. Создание цепочки каталогов

Иногда необходимо создать глубоко вложенный каталог, причем промежуточные каталоги могут и не существовать. В UNIX мы воспользовались бы для этого командой `mkdir -p`.

В программе на Ruby такую операцию выполняет метод `FileUtils.makedirs` (из библиотеки `fileutils`):

```
require "fileutils"
```

```
FileUtils.makedirs("/tmp/these/dirs/need/not/exist")
```

10.1.33. Рекурсивное удаление каталога

В UNIX команда `rm -rf dir` удаляет все поддерево начиная с каталога `dir`. Понятно, что применять ее надо с осторожностью.

В последних версиях Ruby в класс `Pathname` добавлен метод `rmtree`, решающий ту же задачу. В модуле `FileUtils` есть аналогичный метод `rm_r`.

```
require 'pathname'
dir = Pathname.new("/home/poole/")
dir.rmtree
```

или:

```
require 'fileutils'
FileUtils.rm_r("/home/poole")
```

10.1.34. Поиск файлов и каталогов

Ниже мы воспользовались стандартной библиотекой `find.rb` для написания метода, который находит один или более файлов и возвращает их список в виде массива. Первый параметр — это начальный каталог, второй — либо имя файла (строка), либо регулярное выражение.

```
require "find"

def findfiles(dir, name)
  list = []
  Find.find(dir) do |path|
    Find.prune if [".", ".."].include? path
    case name
    when String
      list << path if File.basename(path) == name
    when Regexp
      list << path if File.basename(path) =~ name
    else
      raise ArgumentError
    end
  end
  list
end

findfiles "/home/hal", "toc.txt"
# ["/home/hal/docs/toc.txt", "/home/hal/misc/toc.txt"]

findfiles "/home", /^([a-z]+).doc/
# ["/home/hal/docs/alpha.doc", "/home/guy/guide.doc",
#  "/home/bill/help/readme.doc"]
```

10.2. Доступ к данным более высокого уровня

Часто возникает необходимость хранить и извлекать данные более прозрачным способом. Модуль `Marshal` предоставляет простые средства сохранения объектов, а на его основе построена библиотека `RStore`. Наконец, библиотека `dbm` позволяет организовать нечто вроде хэша на диске. Строго говоря, она не относится к теме данного раздела, но уж слишком проста, чтобы рассказывать о ней в разделе, посвященном базам данных.

10.2.1. Простой маршалинг

Часто бывает необходимо создать объект и сохранить его для последующего использования. В Ruby есть рудиментарная поддержка для обеспечения устойчивости объекта или маршалинга. Модуль `Marshal` позволяет сериализовать и десериализовать объекты.

```
# Массив элементов [composer, work, minutes]
works = [{"Leonard Bernstein", "Overture to Candide", 11},
         {"Aaron Copland", "Symphony No. 3", 45},
         {"Jean Sibelius", "Finlandia", 20}]

# Мы хотим сохранить его для последующего использования...
File.open("store", "w") do |file|
  Marshal.dump(works, file)
end

# Намного позже...
File.open("store") do |file|
  works = Marshal.load(file)
end
```

Недостаток такого подхода заключается в том, что не все объекты можно сохранить. Для объектов, включающих другие объекты низкого уровня, маршалинг невозможен. К числу таких низкоуровневых объектов относятся, в частности, `IO`, `Proc` и `Binding`. Нельзя также сериализовать синглетные объекты, анонимные классы и модули.

Метод `Marshal.dump` можно вызывать еще двумя способами. Если он вызывается с одним параметром, то возвращает данные в виде строки, в которой первые два байта – это номер старшей и младшей версии.

```
s = Marshal.dump(works)
p s[0] # 4
p s[1] # 8
```

Обычно попытка загрузить такие данные оказывается успешной только в случае, если номера старших версий совпадают и номер младшей версии данных не больше младшей версии метода. Но если при вызове интерпретатора Ruby задан флаг «болтливости» (`verbose` или `v`), то версии должны совпадать точно. Эти номера версий не связаны с номерами версий Ruby.

Третий параметр `limit` (целое число) имеет смысл, только если сериализуемый объект содержит вложенные объекты. Если он задан, то интерпретируется методом `Marshal.dump` как максимальная глубина обхода объекта. Если уровень вложенности

меньше указанного порога, то объект сериализуется без ошибок; в противном случае возбуждается исключение `ArgumentError`. Проще пояснить это на примере:

```
File.open("store","w") do |file|
  arr = [ ]
  Marshal.dump(arr,file,0)    # Внутри 'dump': превышена пороговая глубина.
                              # (ArgumentError)

  Marshal.dump(arr,file,1)

  arr = [1, 2, 3]
  Marshal.dump(arr,file,1)    # Внутри 'dump': превышена пороговая глубина.
                              # (ArgumentError)

  Marshal.dump(arr,file,2)

  arr = [1, [2], 3]
  Marshal.dump(arr,file,2)    # Внутри 'dump': превышена пороговая глубина.
                              # (ArgumentError)

  Marshal.dump(arr,file,3)
end

File.open("store") do |file|
  p Marshal.load(file)        # [ ]
  p Marshal.load(file)        # [1, 2, 3]
  p Marshal.load(file)        # arr = [1, [2], 3]
end
```

По умолчанию третий параметр равен 1. Отрицательное значение означает, что глубина вложенности не проверяется.

10.2.2. Более сложный маршалинг

Иногда мы хотим настроить маршалинг под свои нужды. Такую возможность дают методы `_load` и `_dump`. Они вызываются во время выполнения маршалинга, чтобы вы могли самостоятельно реализовать преобразование данных в строку и обратно.

В следующем примере человек получает 5-процентный доход на начальный капитал с момента рождения. Мы не храним ни возраст, ни текущий баланс, поскольку они являются функциями времени.

```
class Person

  attr_reader :name
  attr_reader :age
  attr_reader :balance

  def initialize(name,birthdate,beginning)
    @name = name
    @birthdate = birthdate
    @beginning = beginning
    @age = (Time.now - @birthdate)/(365*86400)
    @balance = @beginning*(1.05**@age)
  end
end
```

```

def marshal_dump
  Struct.new("Human", :name, :birthdate, :beginning)
  str = Struct::Human.new(@name, @birthdate, @beginning)
  str
end

def marshal_load(str)
  self.instance_eval do
    initialize(str.name, str.birthdate, str.beginning)
  end
end

# Прочие методы...

end

p1 = Person.new("Rudy", Time.now - (14 * 365 * 86400), 100)
p [p1.name, p1.age, p1.balance] # ["Rudy", 14.0, 197.99315994394]

str = Marshal.dump(p1)
p2 = Marshal.load(str)

p [p2.name, p2.age, p2.balance] # ["Rudy", 14.0, 197.99315994394]

```

При сохранении объекта этого типа атрибуты `age` и `balance` не сохраняются. А когда объект восстанавливается, они вычисляются заново. Заметьте: метод `marshal_load` предполагает, что объект существует; это один из немногих случаев, когда метод `initialize` приходится вызывать явно (обычно это делает метод `new`).

10.2.3. Ограниченное «глубокое копирование» в ходе маршалинга

В Ruby нет операции «глубокого копирования». Методы `dup` и `clone` не всегда работают, как ожидается. Объект может содержать ссылки на вложенные объекты, а это превращает операцию копирования в игру «собери палочки».

Ниже предлагается способ реализовать глубокое копирование с некоторыми ограничениями, обусловленными тем, что наш подход основан на использовании класса `Marshal` со всеми присущими ему недостатками:

```

def deep_copy(obj)
  Marshal.load(Marshal.dump(obj))
end

```

```
a = deep_copy(b)
```

10.2.4. Обеспечение устойчивости объектов с помощью библиотеки PStore

Библиотека `PStore` реализует хранение объектов Ruby в файле. Объект класса `PStore` может содержать несколько иерархий объектов Ruby. У каждой иерархии есть корень, идентифицируемый ключом. Иерархии считываются с диска в начале транзакции и записываются обратно на диск в конце.

```
require "pstore"
```

```
# Сохранить.
```

```
db = PStore.new("employee.dat")
```

```
db.transaction do
```

```
  db["params"] = {"name" => "Fred", "age" => 32,  
                  "salary" => 48000 }
```

```
end
```

```
# Восстановить.
```

```
require "pstore"
```

```
db = PStore.new("employee.dat")
```

```
emp = nil
```

```
db.transaction { emp = db["params"] }
```

Обычно внутри блока транзакции используется переданный ему объект `pstore`. Но можно получить и сам вызывающий объект, как показано в примере выше.

Эта техника ориентирована на транзакции; в начале блока обрабатываемые данные читаются с диска. А в конце прозрачно для программиста записываются на диск.

Мы можем завершить транзакцию досрочно, вызвав метод `commit` или `abort`. В первом случае все изменения сохраняются, во втором отбрасываются. Рассмотрим более длинный пример:

```
require "pstore"
```

```
# Предполагается, что существует файл с двумя объектами.
```

```
store = PStore.new("objects")
```

```
store.transaction do |s|
```

```
  a = s["my_array"]
```

```
  h = s["my_hash"]
```

```
# Опушен воображаемый код, манипулирующий объектами
```

```
# a, h и т.д.
```

```
# Предполагается, что переменная "condition" может
```

```
# принимать значения 1, 2, 3...
```

```
case condition
```

```
  when 1
```

```
    puts "Отмена."
```

```
    s.abort          # Изменения будут потеряны.
```

```
  when 2
```

```
    puts "Фиксируем и выходим."
```

```
    s.commit         # Изменения будут сохранены.
```

```
  when 3
```

```
    # Ничего не делаем...
```

```
end
```



```
puts "Транзакция дошла до конца."
# Изменения будут сохранены.
```

```
end
```

Внутри транзакции можно вызвать метод `roots`, который вернет массив корней (или метод `root?`, чтобы проверить принадлежность). Есть также метод `delete`, удаляющий корень.

```
store.transaction do |s|
  list = s.roots          # [{"my_array", "my_hash"}]
  if s.root?("my_tree")
    puts "Найдено my_tree."
  else
    puts "Не найдено # my_tree."
  end
  s.delete("my_hash")
  list2 = s.roots         # [{"my_array"}]
end
```

10.2.5. Работа с данными в формате CSV

CSV (comma-separated values – значения, разделенные запятыми) – это формат, с которым вам доводилось сталкиваться, если вы работали с электронными таблицами или базами данных. К счастью, Хироси Накамура (Hiroshi Nakamura) написал для Ruby соответствующий модуль и поместил его в архив приложений Ruby.

Имеется также библиотека `FasterCSV`, которую создал Джеймс Эдвард Грей III (James Edward Gray III). Как явствует из названия, она работает быстрее, к тому же имеет несколько видоизмененный и улучшенный интерфейс (хотя для пользователей старой библиотеки есть «режим совместимости»). Во время работы над книгой велись дискуссии о том, следует ли сделать библиотеку `FasterCSV` стандартной, заменив старую библиотеку (при этом ей, вероятно, будет присвоено старое имя).

Ясно, что это не настоящая база данных. Но более подходящего места, чем эта глава, для нее не нашлось.

Модуль `CSV` (`csv.rb`) разбирает или генерирует данные в формате CSV. О том, что представляет собой последний, нет общепринятого соглашения. Автор библиотеки определяет формат следующим образом:

- разделитель записей: CR + LF;
- разделитель полей: запятая (,);
- данные, содержащие символы CR, LF или запятую, заключаются в двойные кавычки;
- двойной кавычке внутри двойных кавычек должен предшествовать еще один символ двойной кавычки (" → "");
- пустое поле в кавычках обозначает пустую строку (данные,"","данные);
- пустое поле без кавычек означает NULL (данные,,данные).

В настоящем разделе мы рассмотрим лишь часть функциональных возможностей библиотеки. Этого достаточно для введения в предмет, а самую актуальную документацию, как всегда, можно найти в сети (начните с сайта ruby-doc.org).

Начнем с создания файла. Чтобы вывести данные, разделенные запятыми, мы просто открываем файл для записи; метод `open` передает объект-писатель в блок. Затем с помощью оператора добавления мы добавляем массивы данных (при записи они преобразуются в формат CSV). Первая строка является заголовком.

```
require 'csv'

CSV.open("data.csv", "w") do |w|
  w << ["name", "age", "salary"]
  w << ["mark", "29", "34500"]
  w << ["joe", "42", "32000"]
  w << ["fred", "22", "22000"]
  w << ["jake", "25", "24000"]
  w << ["don", "32", "52000"]
end
```

В результате исполнения этого кода мы получаем такой файл `data.csv`:

```
"name","age","salary"
"mark",29,34500
"joe",42,32000
"fred",22,22000
"jake",25,24000
"don",32,52000
```

Другая программа может прочитать этот файл:

```
require 'csv'

CSV.open('data.csv', 'r') do |row|
  p row
end

# Выводится:
# ["name", "age", "salary"]
# ["mark", "29", "34500"]
# ["joe", "42", "32000"]
# ["fred", "22", "22000"]
# ["jake", "25", "24000"]
# ["don", "32", "52000"]
```

Этот фрагмент можно было бы записать и без блока, тогда метод `open` просто вернул бы объект-читатель. Затем можно было бы вызвать метод `shift` читателя (как если бы это был массив) для получения очередной строки. Но блочная форма мне представляется более естественной.

В библиотеке есть и более развитые средства, а также вспомогательные методы. Для получения дополнительной информации обратитесь к сайту ruby-doc.org или архиву приложений Ruby.

10.2.6. Маршалинг в формате YAML

Аббревиатура *YAML* означает «YAML Ain't Markup Language» (YAML – не язык разметки). Это не что иное, как гибкий, понятный человеку формат хранения данных. Он напоминает XML, но «красивее».

Затребовав директивой `require` библиотеку `yaml`, мы добавляем в каждый объект метод `to_yaml`. Поучительно будет посмотреть на результат вывода в этом формате нескольких простых и более сложных объектов.

```
require 'yaml'

str = "Hello, world"
num = 237
arr = %w[ Jan Feb Mar Apr ]
hsh = {"This" => "is", "just a"=>"hash."}
```

```
puts str.to_yaml
puts num.to_yaml
puts arr.to_yaml
puts hsh.to_yaml
```

```
# Выводится:
# --- "Hello, world"
# --- 237
# ---
# - Jan
# - Feb
# - Mar
# - Apr
# ---
# just a: hash.
# This: is
```

Обратным по отношению к `to_yaml` является метод `YAML.load`, который принимает в качестве параметра строку или поток.

Предположим, что имеется такой файл `data.yaml`:

```
---
- "Hello, world"
- 237
-
  - Jan
  - Feb
  - Mar
  - Apr
-
  just a: hash.
  This: is
```

Это те же четыре элемента данных, которые мы видели раньше, только они сгруппированы в единый массив. Если загрузить этот поток, то получим массив:

```
require 'yaml'
file = File.new("data.yaml")
array = YAML.load(file)
file.close
p array
# Выводится:
# ["Hello, world", 237, ["Jan", "Feb", "Mar", "Apr"],
# {"just a"=>"hash.", "This"=>"is"}]
```

В общем и целом YAML – еще один способ выполнить маршалинг объектов. На верхнем уровне его можно использовать для самых разных целей. Например, человек может не только читать данные в этом формате, но и редактировать их, поэтому его естественно применять для записи конфигурационных файлов и т. п.

YAML позволяет и многое другое, о чем мы не можем здесь рассказать. Дополнительную информацию вы найдете на сайте ruby-doc.org или в справочном руководстве.

10.2.7. Преобладающие объекты и библиотека Madeleine

В некоторых кругах популярна идея *преобладающих объектов* (object prevalence). Смысл ее в том, что память дешева и продолжает дешеветь, а базы данных в большинстве своем невелики, поэтому о них можно вообще забыть и хранить все объекты в памяти.

Классической реализацией является пакет Prevalyer, написанный на языке Java. Версия для Ruby называется Madeleine.

Madeleine годится не для всех приложений. У методики преобладающих объектов есть собственные правила и ограничения. Все объекты должны, во-первых, помещаться в память; во-вторых, быть сериализуемы.

Объекты должны быть детерминированы, то есть вести себя одним и тем же образом при получении одних и тех же данных. (Следовательно, применение системного таймера или случайных чисел оказывается под вопросом.)

Объекты должны быть по возможности изолированы от ввода/вывода (файлов и сети). Обычно весь ввод/вывод выполняется вне системы преобладающих объектов.

Наконец, любая команда, которая изменяет состояние системы преобладающих объектов, должна иметь вид объекта-команды (то есть для таких объектов тоже должна иметься возможность сериализации и сохранения).

Madeleine предлагает два основных метода доступа к системе объектов. Метод `execute_query` позволяет выполнить запрос или получить доступ для чтения. Метод `execute_command` инкапсулирует любую операцию, которая изменяет состояние объектов в системе.

Оба метода принимают в качестве параметра объект `Command`. По определению такой объект должен иметь метод `execute`.

Работа системы состоит в том, что во время исполнения приложения она периодически делает моментальные снимки всей системы объектов. Команды сериализуются наравне с другими объектами. В настоящее время не существует способа «откатить» набор транзакций.

Трудно привести содержательный пример использования этой библиотеки. Если вы знакомы с Java-версией, рекомендую изучить API для Ruby и освоить ее таким образом. Хороших руководств нет – может быть, вы напишете первое.

10.2.8. Библиотека DBM

DBM – платформенно-независимый механизм для хранения строк в файле, как в хэше. И ключ, и ассоциированные с ним данные должны быть строками. Интерфейс dbm включен в стандартный дистрибутив Ruby.

Для использования этого класса нужно создать объект DBM, указав для него имя файла, а дальше работать с ним, как с обычным хэшем. По завершении работы файл следует закрыть.

```
require 'dbm'

d = DBM.new("data")
d["123"] = "toodle-oo!"
puts d["123"]          # "toodle-oo!"
d.close

puts d["123"]          # RuntimeError: закрытый DBM-файл.

e = DBM.open("data")
e["123"]               # "toodle-oo!"
w=e.to_hash            # {"123"=>"toodle-oo!"}
e.close

e["123"]               # RuntimeError: закрытый DBM-файл.
w["123"]               # "toodle-oo!"
```

Интерфейс к DBM реализован в виде одного класса, к которому подмешан модуль Enumerable. Два метода класса (синонимы) new и open являются синглетами, то есть в любой момент времени можно иметь только один объект DBM, связанный с данным файлом.

```
q=DBM.new("data.dbm")  #
f=DBM.open("data.dbm") # Errno::EWOULDBLOCK:
                       # Try again - "data.dbm"
```

Всего есть 34 метода экземпляра, многие из которых являются синонимами или аналогичны методам хэша. Почти все операции с настоящим хэшем применимы и к объекту dbm.

Метод to_hash создает представление файла в виде хэша в памяти, а метод close закрывает связь с файлом. Остальные методы по большей части аналогичны методам хэшам, однако дополнительно есть методы rehash, sort, default, default=. Метод to_s возвращает строковое представление идентификатора объекта.

10.3. Библиотека KirbyBase

KirbyBase – небольшая библиотека, с которой должен освоиться каждый программист на Ruby. В настоящее время она не входит в стандартный дистрибутив, а если бы входила, то была бы еще полезнее.

KirbyBase – плод трудов Джейми Криббса (Jamey Cribbs), названный, к слову, в честь его собаки. Во многих отношениях это полноценная база данных, но есть причины, по которым мы рассматриваем ее здесь, а не вместе с MySQL и Oracle.

Во-первых, это не автономное приложение. Это библиотека для Ruby, и без Ruby ее использовать нельзя. Во-вторых, она вообще не знает, что такое язык SQL. Если вам без SQL не обойтись, то эта библиотека не для вас. В-третьих, если приложение достаточно сложное, то функциональных возможностей и быстродействия KirbyBase может не хватить.

Но несмотря на все это, есть немало причин любить KirbyBase. Это написанная целиком на Ruby библиотека, состоящая из единственного файла, которую не нужно ни устанавливать, ни конфигурировать. Она работает на всех платформах, и созданные с ее помощью файлы можно переносить с одной платформы на другую. Это «настоящая» база данных в том смысле, что данные не загружаются целиком в память.

Библиотекой легко пользоваться, а ее интерфейс выдержан в духе Ruby с легким налетом DBI. В общем, база данных соответствует каталогу, а каждая таблица – одному файлу. Формат данных в таблицах таков, что человек может их читать (и редактировать). Дополнительно таблицы можно зашифровать – но только для того, чтобы затруднить редактирование. База знает об объектах Ruby; допускается их хранение и извлечение без потери информации.

Наконец, благодаря интерфейсу dRuby библиотека может работать в распределенном режиме. К данным, хранящимся в KirbyBase, можно с одинаковым успехом обращаться как с локальной, так и с удаленной машины.

Чтобы открыть базу данных, нужно сначала указать, является ли она локальной. Следующие два параметра обычно равны nil, а четвертый указывает каталог, в котором будут храниться файлы с данными (по умолчанию это текущий каталог).

Чтобы создать таблицу, вызывается метод `create_table` объекта, представляющего базу данных; ему передается имя таблицы (объект `Symbol`); имя файла на диске образуется из этого имени. Затем передается последовательность пар символов, описывающих имена и типы полей.

```
require 'kirbybase'
```

```
db = KirbyBase.new(:local, nil, nil, "mydata")
```

```
books = db.create_table(:books,           # Имя таблицы.  
                        :title, :String,   # Поле, тип, ...  
                        :author, :String)
```

В текущей версии KirbyBase распознает следующие типы полей: `String`, `Integer`, `Float`, `Boolean`, `Time`, `Date`, `DateTime`, `Memo`, `Blob` и `YAML`. К тому моменту, когда вы будете читать эту главу, возможно, появятся и новые типы.

Для вставки записи в таблицу применяется метод `insert`. Ему можно передать список значений, хэш или любой объект, отвечающий на заданные имена полей.

```
books.insert("The Case for Mars", "Robert Zubrin")  
books.insert(:title => "Democracy in America",
```

```

      :author => "Alexis de Tocqueville")
Book = Struct.new(:title, :author)
book = Book.new("The Ruby Way", "Hal Fulton")
books.insert(book)

```

В любом случае метод `insert` возвращает идентификатор строки, соответствующей новой записи (вы можете использовать его или игнорировать). Это «скрытое» автоинкрементное поле, присутствующее в каждой записи любой таблицы.

Для выборки записей служит метод `select`. Без параметров он выбирает все поля всех записей таблицы. Набор полей можно ограничить, передав в качестве параметров символы. Если задан блок, то он определяет, какие записи отбирать (примерно так же, как работает метод `find_all` для массивов).

```

list1 = people.select           # Все люди, все поля.
list2 = people.select(:name, :age) # Все люди, только имя и возраст.

list3 = people.select(:name) {|x| x.age >= 18 && x.age < 30 }
# Имена всех людей от 18 до 30 лет.

```

В блоке допустимы любые операции. Это означает, например, что можно формулировать запрос с помощью регулярных выражений (в отличие от типичной SQL-базы).

Результирующий набор, возвращаемый `KirbyBase`, можно сортировать по нескольким ключам в порядке возрастания или убывания. Для сортировки по убыванию перед именем ключа ставится минус. (Это работает, потому что в класс `Symbol` добавлен метод, соответствующий унарному минусу.)

```

sorted = people.select.sort(:name, -:age)
# Отсортировать в порядке возрастания name и в порядке убывания age.

```

У результирующего набора есть одно интересное свойство: он может представлять массивы, «срезающие» результат. С первого раза это довольно трудно понять.

Предположим, что есть результирующий набор записей, представляющих людей, и в каждой записи хранятся имя, возраст, рост и вес. Понятно, что этот результирующий набор можно индексировать как массив, но одновременно он имеет методы, названные так же, как поля. Каждый такой метод возвращает массив значений только соответствующего ему поля. Например:

```

list = people.select(:name, :age, :height, :weight)
p list[0]           # Вся информация о человеке 0.
p list[1].age       # Только возраст человека 1.
p list[2].height     # Рост человека 2.

ages = list.age      # Массив: возрасты всех людей.
names = list.name     # Массив: имена всех людей.

```

В `KirbyBase` есть ограниченные средства печати отчетов; достаточно вызвать метод `to_report` для любого результирующего набора. Пример:

```

rpt = books.select.sort(:title).to_report
puts rpt

```

```
# Выводится:
# resno | title | author
# -----
# 2 | Democracy in America | Alexis de Tocqueville
# 1 | The Case for Mars | Robert Zubrin
# 3 | The Ruby Way | Hal Fulton
```

Атрибут таблицы `encrypt` можно установить в `true` – тогда данные нельзя будет читать и редактировать, как обычный текст. Но имейте в виду, что для этого применяется шифр Вигенера – не «игрушечный», но и не являющийся криптографически безопасным. Так что пользоваться шифрованием имеет смысл только для того, чтобы помешать редактированию, но никак не для сокрытия секретных данных. Обычно режим шифрования устанавливается в блоке при создании таблицы:

```
db.create_table(:mytable, f1, :String, f2, :Date) { |t| t.encrypt = true }
```

Поскольку удаленный доступ – интересное средство, уделим ему немного внимания. Вот пример сервера:

```
require 'kirbybase'
require 'drb'
host = 'localhost'
port = 44444
db = KirbyBase.new(:server) # Создать экземпляр базы данных.
```

```
DRb.start_service("druby://#{host}:#{port}", db)
DRb.thread.join
```

Это прямое применение интерфейса `dRuby` (см. главу 20). На стороне клиента следует при подключении к базе данных задать символ `:client` вместо обычно `:local`.

```
db = KirbyBase.new(:client, 'localhost', 44444)
# Весь остальной код не изменяется.
```

Можно также выполнять обычные операции: обновлять и удалять записи, удалять таблицы и т.д. Есть и более сложные механизмы, о которых я не буду рассказывать подробно: связи один-ко-многим, вычисляемые поля и нестандартные классы записей. Подробнее см. документацию по `KirbyBase` на сайте `RubyForge`.

10.4. Подключение к внешним базам данных

Благодаря усилиям многих людей `Ruby` может взаимодействовать с разными базами данных, от монолитных систем типа `Oracle` до более скромного `MySQL`. Для полноты описания мы включили в него также текстовые файлы в формате `CSV`.

Уровень функциональности, реализованный в этих пакетах, постоянно изменяется. Обязательно познакомьтесь с последней версией документации в сети. Неплохой отправной точкой станет архив приложений `Ruby`.

10.4.1. Интерфейс с `SQLite`

`SQLite` – популярная база данных для тех, кто ценит программное обеспечение, которое не нужно конфигурировать. Это небольшая автономная исполняемая

программа, написанная на языке С, которая хранит всю базу данных в одном файле. Хотя обычно она используется для небольших баз, но теоретически способна управиться с терабайтными объемами.

Привязка Ruby к SQLite довольно прямолинейна. API, написанный на С, обернут в класс `SQLite::API`. Поскольку при этом методы отображаются один в один и интерфейс не назовешь образцом объектной ориентированности, пользоваться этим API стоит только в случае острой необходимости.

В большинстве ситуаций вам будет достаточно класса `SQLite::Database`. Вот пример кода:

```
require 'sqlite'

db = SQLite::Database.new("library.db")

db.execute("select title,author from books") do |row|
  p row
end

db.close

# Выводится:
# ["The Case for Mars", "Robert Zubrin"]
# ["Democracy in America", "Alexis de Tocqueville"]
# ...
```

Если блок не задан, то метод `execute` возвращает объект `ResultSet` (по сути, курсор, который можно перемещать по набору записей).

```
rs = db.execute("select title,author from books")
rs.each {|row| p row }      # Тот же результат, что и выше.
rs.close
```

Если получен объект `ResultSet`, то программа должна будет рано или поздно закрыть его (как показано в примере выше). Если нужно обойти список записей несколько раз, то с помощью метода `reset` можно вернуться в начало. (Это экспериментальное средство, которое в будущем может измениться.) Кроме того, можно производить обход в духе генератора с помощью методов `next` и `eof?`.

```
rs = db.execute("select title,author from books")
while ! rs.eof?
  rec = rs.next
  p rec      # Тот же результат, что и выше.
end
rs.close
```

Методы библиотеки могут возбуждать различные исключения. Все они являются подклассами класса `SQLite::Exception`, так что легко перехватываются поодиночке или целой группой.

Отметим еще, что библиотека написана так, что может работать совместно с библиотекой `ArrayFields` Ары Ховарда (Ara Howard). Она позволяет получить доступ к элементам массива по индексу или по имени. Если перед `sqlite`

затребована библиотека `arrayfields`, то объект `ResultSet` можно индексировать как числами, так и именами полей. (Но можно задать и такую конфигурацию, что вместо этого будет возвращаться объект `hash`.)

Хотя библиотека `sqlite` вполне развита, она не покрывает всех мыслимых потребностей просто потому, что сама база данных `SQLite` не полностью реализует стандарт `SQL92`. Дополнительную информацию об `SQLite` и привязке к `Ruby` ищите в сети.

10.4.2. Интерфейс с MySQL

Интерфейс `Ruby` с `MySQL` – один из самых стабильных и полнофункциональных среди всех интерфейсов с базами данных. Это расширение, которое должно устанавливаться после инсталляции `Ruby` и `MySQL`.

Для использования модуля нужно выполнить три шага: прежде всего, загрузить модуль в свой сценарий, затем установить соединение с базой данных и, наконец, начать работать с таблицами. Для установления соединения следует задать обычные параметры: имя хоста, имя пользователя, пароль, имя базы данных и т.д.

```
require 'mysql'
```

```
m = Mysql.new("localhost","ruby","secret","maillist")
r = m.query("SELECT * FROM people ORDER BY name")
r.each_hash do |f|
  print "#{f['name']} - #{f['email']}"
end
```

Выводится что-то вроде:

```
# John Doe - jdoe@rubynewbie.com
# Fred Smith - smithf@rubyexpert.com
```

Особенно полезны методы класса `Mysql.new` и `MysqlRes.each_hash`, а также метод экземпляра `query`.

Модуль состоит из четырех классов (`Mysql`, `MysqlRes`, `MysqlField` и `MysqlError`), описанных в файле `README`. Мы приведем сводку некоторых наиболее употребительных методов, а дополнительную информацию вы сможете найти сами в официальной документации.

Метод класса `Mysql.new` принимает несколько строковых параметров, которые по умолчанию равны `nil`, и возвращает объект, представляющий соединение. Параметры называются `host`, `user`, `passwd`, `db`, `port`, `sock` и `flag`. У метода `new` есть синонимы `real_connect` и `connect`.

Методы `create_db`, `select_db` и `drop_db` принимают в качестве параметров имя базы данных и используются, как показано ниже. Метод `close` закрывает соединение с сервером.

```
m=Mysql.new("localhost","ruby","secret")
m.create_db("rtest")      # Создать новую базу данных.
m.select_db("rtest2")    # Выбрать другую базу данных.
m.drop_db("rtest")       # Удалить базу данных.
m.close                  # Закрывает соединение.
```

В последних версиях методы `create_db` и `drop_db` объявлены устаревшими. Но можно «воскресить» их, определив следующим образом:

```
class Mysql
  def create_db(db)
    query("CREATE DATABASE #{db}")
  end

  def drop_db(db)
    query("DROP DATABASE #{db}")
  end
end
```

Метод `list_dbs` возвращает список имен доступных баз данных в виде массива.

```
dbs = m.list_dbs      # ["people", "places", "things"]
```

Метод `query` принимает строковый параметр и по умолчанию возвращает объект `MysqlRes`. В зависимости от заданного значения свойства `query_with_result` может также возвращаться объект `Mysql`.

Если произошла ошибка, то ее номер можно получить, обратившись к методу `errno`. Метод `error` возвращает текст сообщения об ошибке.

```
begin
  r=m.query("create table rtable
  (
    id int not null auto_increment,
    name varchar(35) not null,
    desc varchar(128) not null,
    unique id(id)
  )")

# Произошло исключение...

rescue
  puts m.error
  # Печатается: You have an error in your SQL syntax
  # near 'desc varchar(128) not null ,
  #   unique id(id)
  # )' at line 5"

  puts m.errno
  # Печатается 1064
  # ('desc' is reserved for descending order)
end
```

Ниже перечислено несколько полезных методов экземпляра, определенных в классе `MysqlRes`:

- `fetch_fields` возвращает массив объектов `MysqlField`, соответствующих полям в следующей строке;
- `fetch_row` возвращает массив значений полей в следующей строке;

- `fetch_hash(with_table=false)` возвращает хэш, содержащий имена и значения полей в следующей строке;
- `num_rows` возвращает число строк в результирующем наборе;
- `each` – итератор, последовательно возвращающий массив значений полей;
- `each_hash(with_table=false)` – итератор, последовательно возвращающий хэш вида {имя_поля => значение_поля} (пользуйтесь нотацией `x['имя_поля']` для получения значения поля).

Вот некоторые методы экземпляра, определенные в классе `MysqlField`:

- `name` возвращает имя поля;
- `table` возвращает имя таблицы, которой принадлежит поле;
- `length` возвращает длину поля, заданную при определении таблицы;
- `max_length` возвращает длину самого длинного поля в результирующем наборе;
- `hash` возвращает хэш с именами и значениями следующих элементов описания: `name`, `table`, `def`, `type`, `length`, `max_length`, `flags`, `decimals`.

Если изложенный здесь материал противоречит онлайн-официальной документации, предпочтение следует отдать документации. Более подробную информацию вы найдете на официальном сайте MySQL (<http://www.mysql.com>) и в архиве приложений Ruby.

10.4.3. Интерфейс с PostgreSQL

В архиве RAA есть также расширение, реализующее доступ к СУБД PostgreSQL (работает с версиями PostgreSQL 6.5/7.0).

В предположении, что PostgreSQL уже установлена и сконфигурирована (и в базе данных есть таблица `testdb`), нужно лишь выполнить те же шаги, что и для всех остальных интерфейсов Ruby с базами данных: загрузить модуль, установить соединение с базой данных и начать работу. Надо полагать, вам понадобится способ послать запрос, получить результаты и работать с транзакциями.

```
require 'postgres'
conn = PGconn.connect("", 5432, "", "", "testdb")

conn.exec("create table rtest ( number integer default 0 );")
conn.exec("insert into rtest values ( 99 )")
res = conn.query("select * from rtest")
# res id [{"99"}]
```

В классе `PGconn` есть метод `connect`, который принимает обычные параметры для установления соединения: имя хоста, номер порта, имя базы данных, имя и пароль пользователя. Кроме того, третий и четвертый параметры – соответственно, флаги и параметры терминала. В приведенном примере мы установили соединение через сокет UNIX от имени привилегированного пользователя, поэтому не указывали ни имя пользователя, ни пароль, а имя хоста, флаги и параметры терминала оставили пустыми. Номер порта должен быть целым числом, а остальные параметры – строками. У метода `connect` есть синоним `new`.

Для работы с таблицами нужно уметь выполнять запросы. Для этого служат методы `PGconn#exec` и `PGconn#query`.

Метод `exec` посылает переданную ему строку – SQL-запрос – серверу PostgreSQL и получает ответ в виде объекта `PGresult`, если выполнение завершилось успешно. В противном случае он возбуждает исключение `PGError`.

Метод `query` также посылает свой строковый параметр в виде SQL-запроса. Но в случае успеха получает массив кортежей. В случае ошибки возвращается `nil`, а подробности можно получить, вызвав метод `error`.

Имеется специальный метод `insert_table` для вставки записи в указанную таблицу. Вопреки названию он не создает новую таблицу, а добавляет данные в существующую. Этот метод возвращает объект `PGconn`.

```
conn.insert_table("rtest", [[34]])
res = conn.query("select * from rtest")
# res равно [{"99"}, {"34"}]
```

В этом примере в таблицу `rtest` вставляется одна строка. Для простоты мы указали только одну колонку. Отметим, что объект `res` класса `PGresult` после обновления возвращает массив из двух кортежей. Чуть ниже мы рассмотрим методы, определенные в классе `PGresult`.

В классе `PGconn` определены также следующие полезные методы:

- `db` возвращает имя базы, с которой установлено соединение;
- `host` возвращает имя сервера, с которым установлено соединение;
- `user` возвращает имя аутентифицированного пользователя;
- `error` возвращает сообщение об ошибке;
- `finish`, `close` закрывают соединение;
- `loimport(file)` импортирует файл в большой двоичный объект (BLOB), в случае успеха возвращает объект `PGlarge`, иначе возбуждает исключение `PGError`;
- `loexport(oid, file)` выгружает BLOB с идентификатор `oid` в указанный файл;
- `locreate([mode])` возвращает объект `PGlarge` в случае успеха, иначе возбуждает исключение `PGError`;
- `loopen(oid, [mode])` открывает BLOB с идентификатором `oid`. Возвращает объект `PGlarge` в случае успеха. Аргумент `mode` задает режим работы с открытым объектом: `"INV_READ"` или `"INV_WRITE"` (если этот аргумент опущен, по умолчанию предполагается `"INV_READ"`);
- `lounlink(oid)` удаляет BLOB с идентификатором `oid`.

Отметим, что пять последних методов (`loimport`, `loexport`, `locreate`, `loopen` и `lounlink`) работают с объектами класса `PGlarge`. У этого класса есть собственные методы для доступа к объекту и его изменения. (BLOB'ы создаются в результате выполнения методов `loimport`, `locreate`, `loopen` экземпляра.)

Ниже перечислены методы, определенные в классе `PGlarge`:

- `open([mode])` открывает BLOB. Аргумент `mode` задает режим работы с объектом, как и в случае с методом `PGconn#loopen`;
- `close` закрывает BLOB (BLOB'ы также закрываются автоматически, когда их обнаруживает сборщик мусора);
- `read([length])` пытается прочитать `length` байтов из BLOB'a. Если параметр `length` не задан, читаются все данные;
- `write(str)` записывает строку в BLOB и возвращает число записанных байтов;
- `tell` возвращает текущую позицию указателя;
- `seek(offset, whence)` перемещает указатель в позицию `offset`. Параметр `whence` может принимать значения `SEEK_SET`, `SEEK_CUR` и `SEEK_END` (равные соответственно 0,1,2);
- `unlink` удаляет BLOB;
- `oid` возвращает идентификатор BLOB'a;
- `size` возвращает размер BLOB'a;
- `export(file)` сохраняет BLOB в файле с указанным именем.

Более интересны методы экземпляра, определенные в классе `PGresult` (перечислены ниже). Объект такого класса возвращается в результате успешного выполнения запроса. (Для экономии памяти вызывайте метод `PGresult#clear` по завершении работы с таким объектом.)

- `result` возвращает массив кортежей, описывающих результат запроса;
- `each` – итератор;
- `[]` – метод доступа;
- `fields` возвращает массив описаний полей результата запроса;
- `num_tuples` возвращает число кортежей в результате запроса;
- `fieldname(name)` возвращает индекс поля с указанным именем;
- `type(index)` возвращает целое число, соответствующее типу поля;
- `size(index)` возвращает размер поля в байтах. 1 означает, что поле имеет переменную длину;
- `getvalue(tup_num, field_num)` возвращает значение поля с указанным порядковым номером; `tup_num` – номер строки;
- `getlength(tup_num, field_num)` возвращает длину поля в байтах;
- `cmdstatus` возвращает строку состояния для последнего запроса;
- `clear` очищает объект `PGresult`.

10.4.4. Интерфейс с LDAP

Для Ruby есть по меньшей мере три разных библиотеки, позволяющих работать с протоколом LDAP. Ruby/LDAP, написанная Такааки Татеиси (Takaaki Tateishi), – это довольно «тонкая» обертка. Если вы хорошо знакомы с LDAP, то ее может

оказаться достаточно; в противном случае вы, наверное, сочтете ее слишком сложной. Пример:

```
conn = LDAP::Conn.new("rsads02.foo.com")

conn.bind("CN=username,CN=Users,DC=foo,DC=com", "password") do |bound|
  bound.search("DC=foo,DC=com", LDAP::LDAP_SCOPE_SUBTREE,
    "(&(name=*) (objectCategory=person))", ['name', 'ipPhone'])
do |user|
  puts "#{user['name']} #{user['ipPhone']}"
end
end
```

Библиотека ActiveLDAP организована по образцу ActiveRecord. Вот пример ее использования, взятый с домашней страницы:

```
require 'activeldap'
require 'examples/objects/user'
require 'password'

# Установить соединение Ruby/ActiveLDAP и т.д.
ActiveLDAP::Base.connect(:password_block
  => Proc.new { Password.get('Password: ') },
  :allow_anonymous => false)

# Загрузить запись с данными о пользователе
# (ее класс определен в примерах).
wad = User.new('wad')
```

```
# Напечатать общее имя.
p wad.cn
```

```
# Изменить общее имя.
wad.cn = "Will"
```

```
# Сохранить в LDAP.
wad.write
```

Есть также сравнительно недавняя библиотека, написанная Фрэнсисом Чиа-фрокка (Francis Cianfrocca), многие предпочитают именно ее:

```
require 'net/ldap'

ldap = Net::LDAP.new :host => server_ip_address,
  :port => 389,
  :auth => {
    :method => :simple,
    :username => "cn=manager,dc=example,dc=com",
    :password => "opensesame"
  }

filter = Net::LDAP::Filter.eq( "cn", "George*" )
```

```
treebase = "dc=example,dc=com"

ldap.search( :base => treebase, :filter => filter ) do |entry|
  puts "DN: #{entry.dn}"
  entry.each do |attribute, values|
    puts "    #{attribute}:"
    values.each do |value|
      puts "        --->#{value}"
    end
  end
end

p ldap.get_operation_result
```

Какая из этих библиотек лучше – дело вкуса. Я рекомендую познакомиться со всеми и сформировать собственное мнение.

10.4.5. Интерфейс с Oracle

Oracle – одна из наиболее мощных и популярных СУБД в мире. Понятно, что было много попыток реализовать интерфейс с этой базой данных из Ruby. На сегодняшний день лучшей считается библиотека OCI8, которую написал Кубо Такехи-ро (Kubo Takehiro).

Вопреки названию, библиотека OCI8 работает и с версиями Oracle младше 8. Но она еще не вполне зрелая, поэтому не позволяет воспользоваться некоторыми средствами, появившимися в последних версиях.

API состоит из двух уровней: тонкая обертка (низкоуровневый API, довольно точно повторяющий интерфейс вызовов Oracle – Call Level Interface). Но в большинстве случаев вы будете работать с высокоуровневым API. Не исключено, что в будущем низкоуровневый API станет недокументированным.

Модуль OCI8 включает классы Cursor и Blob. Класс OCIException служит предком всех классов исключений, которые могут возникнуть при работе с базой данных: OCIError, OCIBreak и OCIInvalidHandle.

Чтобы установить соединение с сервером, вызывается метод OCI8.new, которому нужно передать как минимум имя и пароль пользователя. В ответ возвращается описатель, который можно использовать для выполнения запросов. Пример:

```
require 'oci8'

session = OCI8.new('user', 'password')

query = "SELECT TO_CHAR(SYSDATE, 'YYYY/MM/DD') FROM DUAL"
cursor = session.exec(query)
result = cursor.fetch          # В данном случае всего одна итерация.
cursor.close
session.logoff
```

В примере выше показано, как манипулировать курсором, хотя в данном случае перед закрытием выполняется всего одна операция fetch. Конечно, можно брать и несколько строк:


```

query = 'select * from some_table'
cursor = session.exec(query)
while row = cursor.fetch
  puts row.join(",")
end
cursor.close
# Или с помощью блока:

```

```

nrows = session.exec(query) do |row|
  puts row.join(",")
end

```

Связанные переменные в запросе напоминают символы. Есть несколько способов связать переменные со значениями:

```

session = OCI8.new("user", "password")
query = "select * from people where name = :name"

```

```

# Первый способ...
session.exec(query, 'John Smith')

```

```

# Второй...
cursor = session.parse(query)
cursor.exec('John Smith')

```

```

# Третий...
cursor = session.parse(query)
cursor.bind_param(':name', 'John Smith') # Связывание по имени.
cursor.exec

```

```

# И четвертый.
cursor = session.parse(query)
cursor.bind_param(1, 'John Smith')      # Связывание по номеру.
cursor.exec

```

Для тех, кто предпочитает интерфейс DBI, имеется соответствующий адаптер. Дополнительную информацию можно найти в документации по OCI8.

10.4.6. Обертка вокруг DBI

Теоретически интерфейс DBI обеспечивает доступ к любым базам данных. Иными словами, один и тот же код должен работать и с Oracle, и с MySQL, и с PostgreSQL, и с любой другой СУБД, стоит лишь изменить одну строку, в которой указан нужный адаптер. Иногда эта идеология не срабатывает для сложных операций, специфичных для конкретной СУБД, но для рутинных задач она вполне годится.

Пусть имеется база данных под управлением Oracle и используется драйвер (он же адаптер), поставляемый вместе с библиотекой OCI8. Методу connect следует передать достаточно информации для успешного соединения с базой данных. Все более или менее интуитивно очевидно.

```

require "dbi"

```

```

db = DBI.connect("dbi:OCI8:mydb", "user", "password")

```

```
query = "select * from people"
```

```
stmt = db.prepare(query)
stmt.execute
```

```
while row = stmt.fetch do
  puts row.join(",")
end
```

```
stmt.finish
db.disconnect
```

Здесь метод `prepare` – это некий вариант компиляции или синтаксического анализа запроса, который позже выполняется. Метод `fetch` извлекает одну строку из результирующего набора и возвращает `nil`, если строк не осталось (поэтому мы и воспользовались циклом `while`). Метод `finish` можно считать вариантом закрытия или освобождения ресурсов.

Полную информацию обо всех возможностях DBI можно найти в любом справочном руководстве. Список имеющихся драйверов приведен на сайте [RubyForge](http://www.rubyforge.org) и в архиве приложений Ruby.

10.4.7. Объектно-реляционные отображения (ORM)

Традиционная реляционная база данных прекрасно справляется со своими задачами. Она эффективно выполняет произвольные запросы, о которых заранее ничего не знает. Но эта модель плохо уживается с объектной ориентированностью.

Повсеместная распространенность обеих моделей (РСУБД и ООП) и «несогласованный импеданс» между ними побудил многих людей попытаться перебраться мост. Этот программный мост получил название «объектно-реляционное отображение» (Object-Relational Mapper – ORM).

К этой задаче существуют разные подходы. У каждого есть свои достоинства и недостатки. Ниже мы рассмотрим два популярных ORM: `ActiveRecord` и `Og` (последняя аббревиатура обозначает «object graph» – граф объектов).

Библиотека `ActiveRecord` для Ruby названа в честь предложенного Мартином Фаулером (Martin Fowler) паттерна проектирования «Active Record» (активная запись). Смысл его в том, что таблицам базы данных сопоставляются классы, в результате чего данными становится возможно манипулировать без привлечения SQL. Точнее говоря, «она (активная запись) обертывает строку таблицы или представления, инкапсулирует доступ к базе данных и наделяет данные логикой, присущей предметной области» (см. книгу Martin Fowler «Patterns of Enterprise Application Architecture», Addison Wesley, 2003 [ISBN: 0-321-12742-0e]).

Каждая таблица описывается классом, производным от `ActiveRecord::Base`. Как и в случае с DBI, для установления соединения нужно предоставить достаточной информации для идентификации пользователя и базы данных. Вот небольшой пример, демонстрирующий весь механизм в действии:

```
require 'active_record'
```

```
ActiveRecord::Base.establish_connection(:adapter => "oci8",
```

```

:username => "username",
:password => "password",
:database => "mydb",
:host => "myhost")

```

```

class SomeTable < ActiveRecord::Base
  set_table_name "test_table"
  set_primary_key "some_id"
end

```

```

SomeTable.find(:all).each do |rec|
  # Обработать запись...
end

```

```

item = SomeTable.new
item.id = 1001
item.some_column = "test"
item.save

```

Библиотека предлагает богатый и сложный API. Я рекомендую ознакомиться со всеми руководствами, которые вы сможете найти в сети или в книгах. Поскольку эта библиотека составляет неотъемлемую часть системы «Ruby on Rails», то мы еще вернемся к ней в главе, посвященной этой теме.

Og отличается от ActiveRecord тем, что в центре внимания последней находится база данных, а первая делает упор на объекты. Og может сгенерировать схему базы данных, имея определения классов на языке Ruby (но не наоборот).

При работе с Og нужен совсем другой стиль мышления; она не так распространена, как ActiveRecord. Но мне кажется, что у этой библиотеки есть свои «изюминки», и ее следует рассматривать как мощный и удобный механизм ORM, особенно если вы проектируете базу данных исходя из структуры имеющихся объектов.

Определяя подлежащий хранению класс, мы пользуемся методом `property`, который похож на метод `attr_accessor`, только с ними ассоциирован тип (класс).

```

class SomeClass
  property :alpha, String
  property :beta, String
  property :gamma, String
end

```

Поддерживаются также типы данных Integer, Float, Time, Date и пр. Потенциально возможно связать со свойством произвольный объект Ruby.

Соединение с базой данных устанавливается так же, как в случае ActiveRecord или DBI.

```

db = Og::Database.new(:destroy => false,
                      :name => 'mydb',
                      :store => :mysql,
                      :user => 'ha19000',
                      :password => 'chandra')

```

У каждого объекта есть метод `save`, который и вставляет соответствующую ему запись в базу данных:

```
obj = SomeClass.new
obj.alpha = "Poole"
obj.beta  = "Whitehead"
obj.gamma = "Kaminski"
obj.save
```

Имеются также методы для описания связей объекта в терминах классической теории баз данных:

```
class Dog
  has_one :house
  belongs_to :owner
  has_many :fleas
end
```

Эти, а также другие методы, например `many_to_many` и `refers_to`, помогают создавать сложные связи между объектами и таблицами.

Библиотека `OG` слишком велика, чтобы ее документировать на страницах этой книги. Дополнительную информацию вы можете найти в онлайн-источниках (например, на сайте <http://oxyliquit.de>).

10.5. Заключение

В данной главе был представлен обзор ввода/вывода в Ruby. Мы рассмотрели сам класс `IO` и его подкласс `File`, а также связанные с ними классы, в частности `Dir` и `Pathname`. Мы познакомились с некоторыми полезными приемами манипулирования объектами `IO` и файлами.

Также было уделено внимание вопросам хранения данных на более высоком уровне, точнее, на внешних носителях в виде сериализованных объектов. Наконец, мы дали краткий обзор решений, которые Ruby предлагает для интерфейса с настоящими базами данных, а кроме того, познакомились с некоторыми объектно-ориентированными подходами к взаимодействию с реляционными СУБД.

Ниже мы еще вернемся к вводу/выводу в контексте сокетов и сетевого программирования. Но предварительно рассмотрим некоторые другие темы.

Глава 11. ООП и динамические механизмы в Ruby

.....
Как введение иррациональных чисел... стало удобным мифом, упростившим законы арифметики... так физические объекты и постулированные сущности делают наше восприятие реальности более простым и завершенным... Концептуальная схема физических объектов напоминает удобный миф, который проще, чем истина, но при этом содержит тут и там частички истины.
Уиллард Ван Орман Квин

Это необычная глава. В большинстве других глав рассматривается какой-то конкретный аспект, например строки или файлы, но в этой все иначе. Если расположить «пространство задачи» по одной оси системы координат, то данная глава окажется на другой оси, поскольку содержит по кусочку из всех других областей. Связано это с тем, что объектно-ориентированное программирование и динамичность сами по себе являются не задачами, а парадигмами, которые могут быть применены к решению любой задачи, будь то системное администрирование, низкоуровневое сетевое программирование или разработка приложений для Web.

Вот почему значительная часть материала данной главы должна быть уже знакома любому программисту, знающему Ruby. На самом деле все остальное в этой книге не имеет смысла без понимания изложенных здесь основ. Например, любой программист на Ruby знает, как создать подкласс. Возникает вопрос: что включить, а без чего можно обойтись? Знает ли любой программист о методе `extend`? А о методе `instance_eval`? То, что одному представляется очевидным, может оказаться откровением для другого.

Мы решили отдать предпочтение полноте. В главу включены некоторые экзотические вещи, которые можно делать с помощью динамического ООП в Ruby, но не забыты и рутинные задачи – на случай, если кто-то не знаком с ними. Мы спустились до самого простого уровня, поскольку многие по-разному представляют себе, где кончается «средний» уровень. И попытались дать кое-какую дополнительную информацию даже при изложении самых базовых вопросов, чтобы оправдать включение их в эту главу. С другой стороны, тем, которые раскрываются в других частях книги, мы здесь не касались.

Еще два замечания. Во-первых, ничего магического в динамическом ООП нет. Объектная ориентированность языка Ruby и его динамическая природа прекрасно уживаются между собой, но неотъемлемой связи между ними нет. Мы рассказываем о том и другом в одной главе только для удобства. Во-вторых, мы затрагиваем кое-какие особенности языка, которые, строго говоря, не относятся ни к одной из

двух заявленных тем. Если хотите, считайте это мелким обманом. Но надо же было поместить их куда-то.

11.1. Рутинные объектно-ориентированные задачи

.....
*Of his quick objects hath the mind no part,
 Nor his own vision holds what it doth catch...*
 Вильям Шекспир, Сонет 113*

Если вы вообще не знакомы с ООП, то эта глава вас ничему не научит. А если вы понимаете, что такое ООП в языке Ruby, то, наверное, ее и читать не стоит. Если понятия ООП не слишком свежи в памяти, просмотрите главу 1, где мы приводим их краткий обзор (или обратитесь к другой книге).

С другой стороны, большая часть материала в этой главе изложена в виде руководства и довольно элементарна. Поэтому она будет полезна начинающему и менее интересна для программиста на Ruby среднего уровня. Эта книга организована как устройство ввода/вывода с произвольной выборкой, так что можете свободно пропускать те части, которые вас не интересуют.

11.1.1. Применение нескольких конструкторов

В Ruby нет «настоящих» конструкторов, как в C++ или в Java. Сама идея, конечно, никуда не делась, поскольку объекты необходимо создавать и инициализировать, но реализация выглядит иначе.

В Ruby каждый класс имеет метод класса `new`, который вызывается для создания новых объектов. Метод `new` вызывает специальный определяемый пользователем метод `initialize`, который инициализирует атрибуты объекта, после чего `new` возвращает ссылку на новый объект.

А если мы хотим иметь несколько конструкторов? Как быть в этом случае?

Ничто не мешает завести дополнительные методы класса, которые возвращают новые объекты. В листинге 11.1 приведен искусственный пример класса для представления прямоугольника, у которого есть две длины сторон и три значения цвета. Мы создали дополнительные методы класса, предполагающие определенные умолчания для каждого параметра. (Например, квадрат – это прямоугольник, у которого все стороны равны.)

Листинг 11.1. Несколько конструкторов

```
class ColoredRectangle

  def initialize(r, g, b, s1, s2)
    @r, @g, @b, @s1, @s2 = r, g, b, s1, s2
  end
```

* Ни сердцу, ни сознанию беглый взгляд
 Не хочет дать о виденном отчет. (Пер. С. Маршак)

```

def ColoredRectangle.white_rect(s1, s2)
  new(0xff, 0xff, 0xff, s1, s2)
end

def ColoredRectangle.gray_rect(s1, s2)
  new(0x88, 0x88, 0x88, s1, s2)
end

def ColoredRectangle.colored_square(r, g, b, s)
  new(r, g, b, s, s)
end

def ColoredRectangle.red_square(s)
  new(0xff, 0, 0, s, s)
end

def inspect
  "#@r #@g #@b #@s1 #@s2"
end

a = ColoredRectangle.new(0x88, 0xaa, 0xff, 20, 30)
b = ColoredRectangle.white_rect(15,25)
c = ColoredRectangle.red_square(40)

```

Таким образом, можно определить любое число методов, создающих объекты по различным спецификациям. Вопрос о том, уместен ли здесь термин «конструктор», мы оставим «языковым адвокатам».

11.1.2. Создание атрибутов экземпляра

Имени атрибута экземпляра в Ruby всегда предшествует знак @. Это обычная переменная в том смысле, что она начинает существовать после первого присваивания.

В ОО-языках часто создаются методы для доступа к атрибутам, чтобы обеспечить сокрытие данных. Мы хотим контролировать доступ к «внутренностям» объекта извне. Обычно для данной цели применяются методы чтения и установки (getter и setter), хотя в Ruby эта терминология не используется. Они просто читают (get) или устанавливают (set) значение атрибута.

Можно, конечно, запрограммировать такие функции «вручную», как показано ниже:

```

class Person

  def name
    @name
  end

  def name=(x)
    @name = x
  end

  def age

```

```
@age
end

# ...
```

```
end
```

Но Ruby предоставляет более короткий способ. Метод `attr` принимает в качестве параметра символ и создает соответствующий атрибут. Кроме того, он создает одноименный метод чтения, а если необязательный второй параметр равен `true`, то и метод установки.

```
class Person
  attr :name, true      # Создаются @name, name, name=
  attr :age             # Создаются @age, age
end
```

Методы `attr_reader`, `attr_writer` и `attr_accessor` принимают в качестве параметров произвольное число символов. Первый создает только «методы чтения» (для получения значения атрибута); второй – только «методы установки», а третий – то и другое. Пример:

```
class SomeClass
  attr_reader :a1, :a2   # Создаются @a1, a1, @a2, a2
  attr_writer :b1, :b2   # Создаются @b1, b1=, @b2, b2=
  attr_accessor :c1, :c2 # Создаются @c1, c1, c1=, @c2, c2, c2=
  # ...
end
```

Напомним, что для выполнения присваивания атрибуту необходимо указывать вызывающий объект, а внутри метода нужно в качестве такого объекта указывать `self`.

11.1.3. Более сложные конструкторы

По мере усложнения объектов у них появляется все больше атрибутов, которые необходимо инициализировать в момент создания. Соответствующий конструктор может оказаться длинным и запутанным, его параметры даже не будут помещаться на одной строке.

Чтобы справиться со сложностью, можно передать методу `initialize` блок (листинг 11.2). Тогда инициализация объекта выполняется в процессе вычисления этого блока. Хитрость в том, что вместо обычного `eval` для вычисления блока в контексте объекта, а не вызывающей программы, следует использовать метод `instance_eval`.

Листинг 11.2. «Хитрый» конструктор

```
class PersonalComputer
  attr_accessor :manufacturer,
                :model, :processor, :clock,
                :ram, :disk, :monitor,
                :colors, :vres, :hres, :net
```



```

def initialize(&block)
  instance_eval &block
end

# Прочие методы...
end

desktop = PersonalComputer.new do
  self.manufacturer = "Acme"
  self.model = "TNX-1138"
  self.processor = "986"
  self.clock = 9.6      # ГГц
  self.ram = 16         # Гб
  self.disk = 20        # Тб
  self.monitor = 25     # дюймы
  self.colors = 16777216
  self.vres = 1280
  self.hres = 1600
  self.net = "T3"
end

p desktop

```

Отметим несколько нюансов. Во-первых, мы пользуемся методами доступа к атрибутам, поэтому присваивание им значений интуитивно понятно. Во-вторых, ссылка на `self` необходима, поскольку метод установки требует явного указания вызывающего объекта, чтобы можно было отличить вызов метода от обычного присваивания локальной переменной. Конечно, можно было не определять методы доступа, а воспользоваться функциями установки.

Ясно, что в теле блока можно делать все, что угодно. Например, можно было бы вычислить некоторые поля на основе других.

А если вам не нужны методы доступа для всех атрибутов? Если хотите, можете избавиться от лишних, вызвав для них метод `undef` в конце конструирующего блока. Как минимум, это предотвратит «случайное» присваивание значения атрибуту извне объекта.

11.1.4. Создание атрибутов и методов уровня класса

Метод или атрибут не всегда ассоциируются с конкретным экземпляром класса, они могут принадлежать самому классу. Типичным примером метода класса может служить `new`, он вызывается для создания новых экземпляров, а потому не может принадлежать никакому конкретному экземпляру.

Мы можем определять собственные методы класса, как показано в разделе 11.1.1. Конечно, их функциональность не ограничивается конструированием — они могут выполнять любые операции, имеющие смысл именно на уровне класса.

В следующем далеко не полном фрагменте предполагается, что мы создаем класс для проигрывания звуковых файлов. Метод `play` естественно реализовать как метод экземпляра, ведь можно создать много объектов, каждый из которых

будет проигрывать свой файл. Но у метода `detect_hardware` контекст более широкий; в зависимости от реализации может оказаться, что создавать какие-либо объекты вообще не имеет смысла, если этот метод возвращает ошибку. Следовательно, его контекст – вся среда воспроизведения звука, а не конкретный звуковой файл.

```
class SoundPlayer

  MAX_SAMPLE = 192

  def SoundPlayer.detect_hardware
    # ...
  end

  def play
    # ...
  end

end
```

Есть еще один способ объявить этот метод класса. В следующем фрагменте делается практически то же самое:

```
class SoundPlayer

  MAX_SAMPLE = 192

  def play
    # ...
  end

end

def SoundPlayer.detect_hardware
  # ...
end
```

Единственная разница касается использования объявленных в классе констант. Если метод класса объявлен вне объявления самого класса, то эти константы оказываются вне области видимости. Например, в первом фрагменте метод `detect_hardware` может напрямую обращаться к константе `MAX_SAMPLE`, а во втором придется пользоваться нотацией `SoundPlayer::MAX_SAMPLE`.

Не удивительно, что помимо методов класса есть еще и переменные класса. Их имена начинаются с двух знаков `@`, а областью видимости является весь класс, а не конкретный его экземпляр.

Традиционный пример использования переменных класса – подсчет числа его экземпляров. Но они могут применяться всегда, когда информации имеет смысл в контексте класса в целом, а не отдельного объекта. Другой пример приведен в листинге 11.3.

Листинг 11.3. Переменные и методы класса

```
class Metal

  @@current_temp = 70

  attr_accessor :atomic_number

  def Metal.current_temp=(x)
    @@current_temp = x
  end

  def Metal.current_temp
    @@current_temp
  end

  def liquid?
    @@current_temp >= @melting
  end

  def initialize(atnum, melt)
    @atomic_number = atnum
    @melting = melt
  end

end

aluminum = Metal.new(13, 1236)
copper = Metal.new(29, 1982)
gold = Metal.new(79, 1948)

Metal.current_temp = 1600

puts aluminum.liquid?      # true
puts copper.liquid?        # false
puts gold.liquid?          # false

Metal.current_temp = 2100

puts aluminum.liquid?      # true
puts copper.liquid?        # true
puts gold.liquid?          # true
```

Здесь переменная класса инициализируется до того, как впервые используется в методе класса. Отметим также, что мы можем обратиться к переменной класса из метода экземпляра, но обратиться к переменной экземпляра из метода класса нельзя. Немного подумав, вы поймете, что так и должно быть.

А если попытаться, что произойдет? Что если мы попробуем напечатать атрибут `@atomic_number` из метода `Metal.current_temp`? Обнаружится, что переменная вроде бы существует – никакой ошибки не возникает, – но имеет значение `nil`. В чем дело?

В том, что на самом деле мы обращаемся вовсе не к переменной экземпляра класса `Meta1`, а к переменной экземпляра класса `Class`. (Напомним, что в Ruby `class` – это класс!)

Мы столкнулись с переменной экземпляра класса (термин заимствован из языка Smalltalk). Дополнительные замечания на эту тему приводятся в разделе 11.2.4.

В листинге 11.4 иллюстрируются все аспекты этой ситуации.

Листинг 11.4. Данные класса и экземпляра

```
class MyClass

  SOME_CONST = "alpha"          # Константа уровня класса.

  @@var = "beta"                # Переменная класса.
  @var = "gamma"                # Переменная экземпляра класса.

  def initialize
    @var = "delta"              # Переменная экземпляра.
  end

  def mymethod
    puts SOME_CONST             # (Константа класса.)
    puts @@var                  # (Переменная класса.)
    puts @var                   # (Переменная экземпляра.)
  end

  def MyClass.classmeth1
    puts SOME_CONST             # (Константа класса.)
    puts @@var                  # (Переменная класса.)
    puts @var                   # (Переменная экземпляра класса.)
  end

end

def MyClass.classmeth2
  puts MyClass::SOME_CONST      # (Константа класса.)
  # puts @@var                  # Ошибка: вне области видимости.
  puts @var                     # (Переменная экземпляра класса.)
end

myobj = MyClass.new
MyClass.classmeth1              # alpha, beta, gamma
MyClass.classmeth2              # alpha, gamma
myobj.mymethod                  # alpha, beta, delta
```

Следует еще сказать, что метод класса можно сделать *закрытым*, воспользовавшись методом `private_class_method`. Это аналог метода `private` на уровне экземпляра. См. также раздел 11.2.10.

11.1.5. Наследование суперклассу

Можно унаследовать класс, воспользовавшись символом <:

```
class Boojum < Snark
  # ...
end
```

Это объявление говорит, что класс Boojum является подклассом класса Snark или – что то же самое – класс Snark является суперклассом класса Boojum. Всем известно, что каждый буюм является снарком, но не каждый снарк – буюм.

Ясно, что цель наследования – расширить или специализировать функциональность. Мы хотим получить из общего нечто более специфическое.

Попутно отметим, что во многих языках, например в C++, допускается множественное наследование (МН). В Ruby, как и в Java, и в некоторых других языках, множественного наследования нет, но наличие классов-примесей компенсирует его отсутствие (см. раздел 11.1.12).

Рассмотрим несколько более реалистичный пример. У нас есть класс Person (человек), а мы хотим создать производный от него класс Student (студент).

Определим класс Person следующим образом:

```
class Person

  attr_accessor :name, :age, :sex

  def initialize(name, age, sex)
    @name, @age, @sex = name, age, sex
  end

  # ...

end
```

А класс Student – так:

```
class Student < Person

  attr_accessor :idnum, :hours

  def initialize(name, age, sex, idnum, hours)
    super(name, age, sex)
    @idnum = idnum
    @hours = hours
  end

  # ...

end

# Создать два объекта.
a = Person.new("Dave Bowman", 37, "m")
b = Student.new("Franklin Poole", 36, "m", "000-13-5031", 24)
```

Посмотрим внимательно, что здесь сделано. Что за `super`, вызываемый из метода `initialize` класса `Student`? Это просто вызов соответствующего метода родительского класса. А раз так, то ему передается три параметра (хотя наш собственный метод `initialize` принимает пять).

Не всегда необходимо использовать слово `super` подобным образом, но часто это удобно. В конце концов, атрибуты любого класса образуют надмножество множества атрибутов его родительского класса, так почему не воспользоваться для инициализации конструктором родительского класса?

Если говорить об истинном смысле наследования, то оно, безусловно, описывает отношение «является». Студент является человеком, как и следовало ожидать. Сделаем еще три замечания:

- Каждый атрибут (и метод) родительского класса отражается в его потомках. Если в классе `Person` есть атрибут `height`, то класс `Student` унаследует его, а если родитель имеет метод `say_hello`, такой метод будет и у потомка.
- Потомок может иметь дополнительные атрибуты и методы, мы это только что видели. Поэтому создание подкласса часто еще называют *расширением суперкласса*.
- Потомок может переопределять любые атрибуты и методы своего родителя.

Последнее замечание подводит нас к вопросу о том, как разрешается вызов метода. Откуда я знаю, вызывается ли метод конкретного класса или его суперкласса?

Краткий ответ таков: не знаю и не интересуюсь. Если вызывается некий метод от имени объекта класса `Student`, то будет вызван метод, определенный в этом классе, если он существует. А если нет, вызывается метод суперкласса и так далее вверх по иерархии наследования. Мы говорим «и так далее», потому что у каждого класса (кроме `Object`) есть суперкласс.

А что если мы хотим вызвать метод суперкласса, но не из соответствующего метода подкласса? Можно сначала создать в подклассе синоним:

```
class Student      # Повторное открытие класса.

  # Предполагается, что в классе Person есть метод say_hello...

  alias :say_hi :say_hello

  def say_hello
    puts "Привет."
  end
  def formal_greeting
    # Поприветствовать так, как принято в суперклассе.
    say_hi
  end
end
```

У наследования есть разные тонкости, которых мы здесь касаться не будем. Общий принцип мы изложили, но не пропустите следующий раздел.

11.1.6. Опрос класса объекта

Часто возникает вопрос: «Что это за объект? Как он соотносится с данным классом?» Есть много способов получить тот или иной ответ.

Во-первых, метод экземпляра `class` всегда возвращает класс объекта. Применявшийся ранее синоним `type` объявлен устаревшим.

```
s = "Hello"
n = 237
sc = s.class      # String
nc = n.class      # Fixnum
```

Не думайте, будто методы `class` или `type` возвращают строку, представляющую имя класса. На самом деле возвращается экземпляр класса `Class`! При желании мы могли бы вызвать метод класса, определенный в этом типе, как если бы это был метод экземпляра класса `Class` (каковым он в действительности и является).

```
s2 = "some string"
var = s2.class      # String
my_str = var.new("Hi...") # Новая строка.
```

Можно сравнить такую переменную с константным именем класса и выяснить, равны ли они; можно даже использовать переменную в роли суперкласса и определить на ее основе подкласс! Запутались? Просто помните, что в Ruby `Class` – это объект, а `Object` – это класс.

Иногда нужно сравнить объект с классом, чтобы понять, принадлежит ли данный объект указанному классу. Для этого служит метод `instance_of?`, например:

```
puts (5.instance_of? Fixnum)      # true
puts ("XYZZY".instance_of? Fixnum) # false
puts ("PLUGH".instance_of? String) # true
```

А если нужно принять во внимание еще и отношение наследования? К вашим услугам метод `kind_of?` (похожий на `instance_of?`). У него есть синоним `is_a?`, что вполне естественно, ибо мы описываем классическое отношение «является».

```
n = 9876543210
flag1 = n.instance_of? Bignum      # true
flag2 = n.kind_of? Bignum          # true
flag3 = n.is_a? Bignum             # true
flag3 = n.is_a? Integer            # true
flag4 = n.is_a? Numeric            # true
flag5 = n.is_a? Object             # true
flag6 = n.is_a? String             # false
flag7 = n.is_a? Array              # false
```

Ясно, что метод `kind_of` или `is_a?` более общий, чем `instance_of?`. Например, всякая собака – млекопитающее, но не всякое млекопитающее – собака.

Для новичков в Ruby приготовлен один сюрприз. Любой модуль, подмешиваемый в класс, становится субъектом отношения «является» для экземпляров этого

класса. Например, в класс `Array` подмешан модуль `Enumerable`; это означает, что всякий массив является перечисляемым объектом.

```
x = [1, 2, 3]
flag8 = x.kind_of? Enumerable      # true
flag9 = x.is_a? Enumerable         # true
```

Для сравнения двух классов можно пользоваться также операторами сравнения. Интуитивно очевидно, что оператор «меньше» обозначает наследование суперклассу.

```
flag1 = Integer < Numeric          # true
flag2 = Integer < Object           # true
flag3 = Object == Array            # false
flag4 = IO >= File                 # true
flag5 = Float < Integer            # nil
```

В любом классе обычно определен оператор «тройного равенства» `===`. Выражение `class === instance` истинно, если экземпляр `instance` принадлежит классу `class`. Этот оператор еще называют оператором ветвящегося равенства, потому что он неявно используется в предложении `case`. Дополнительную информацию о нем вы найдете в разделе 11.1.7.

Упомянем еще метод `respond_to`. Он используется, когда нам безразлично, какому классу принадлежит объект, но мы хотим знать, реализует ли он конкретный метод. Это рудиментарный вид получения информации о типе. (Вообще-то можно сказать, что это самая важная информация о типе.) Методу `respond_to` передается символ и необязательный флаг, который говорит, нужно ли включать в рассмотрение также и закрытые методы.

```
# Искать открытые методы.
if wumpus.respond_to?(:bite)
  puts "У него есть зубы!"
else
  puts "Давай-ка подразним его."
end

# Необязательный второй параметр позволяет
# просматривать и закрытые методы.

if woozle.respond_to?(:bite,true)
  puts "Вузлы кусаются!"
else
  puts "Ага, это не кусающийся вузл."
end
```

Иногда нужно знать, является ли данный класс непосредственным родителем объекта или класса. Ответ на этот вопрос дает метод `superclass` класса `Class`.

```
array_parent = Array.superclass    # Object
fn_parent    = 237.class.superclass # Integer
obj_parent   = Object.superclass   # nil
```

У любого класса, кроме `Object`, есть суперкласс.

11.1.7. Проверка объектов на равенство

.....
Все животные равны, но некоторые равнее других.
Джордж Оруэлл, «Скотный двор»

При написании своих классов желательно, чтобы семантика типичных операций была такой же, как у встроенных в Ruby классов. Например, если объекты класса можно упорядочивать, то имеет смысл реализовать метод `<=>` и подмешать модуль `Comparable`. Тогда к объектам вашего класса будут применимы все обычные операции сравнения.

Однако картина перестает быть такой однозначной, когда дело доходит до проверки объектов на равенство. В Ruby объекты реализуют пять разных методов для этой операции. И в ваших классах придется реализовать хотя бы некоторые из них, поэтому рассмотрим этот вопрос подробнее.

Самым главным является метод `equal?` (унаследованный от класса `Object`); он возвращает `true`, если вызывающий объект и параметр имеют один и тот же идентификатор объекта. Это фундаментальный аспект семантики объектов, поэтому переопределять его не следует.

Самым распространенным способом проверки на равенство является старый добрый оператор `==`, который сравнивает значения вызывающего объекта и аргумента. Наверно, интуитивно это наиболее очевидный способ.

Следующим в шкале абстракции стоит метод `eql?` – тоже часть класса `Object`. (На самом деле метод `eql?` реализован в модуле `Kernel`, который подмешивается в `Object`.) Как и оператор `==`, этот метод сравнивает значения вызывающего объекта и аргумента, но несколько более строго. Например, разные числовые объекты при сравнении с помощью `==` приводятся к общему типу, но метод `eql?` никогда не считает объекты разных типов равными.

```
flag1 = (1 == 1.0)      # true
flag2 = (1.eql?(1.0))   # false
```

Метод `eql?` существует по одной-единственной причине: для сравнения значений ключей хэширования. Если вы хотите переопределить стандартное поведение Ruby при использовании объектов в качестве ключей хэша, то переопределите методы `eql?` и `hash`.

Любой объект реализует еще два метода сравнения. Метод `===` применяется для сравнения проверяемого значения в предложении `case` с каждым селектором: `selector===target`. Хотя правило на первый взгляд кажется сложным, на практике оно делает предложения `case` в Ruby интуитивно очевидными. Например, можно выполнить ветвление по классу объекта:

```
case an_object
when String
  puts "Это строка."
when Numeric
  puts "Это число."
```

```
else
  puts "Это что-то совсем другое."
end
```

Эта конструкция работает, потому что в классе `Module` реализован метод `===`, проверяющий, принадлежит ли параметр тому же классу, что вызывающий объект (или одному из его предков). Поэтому, если `an_object` – это строка «cat», выражение `String === an_object` окажется истинным и будет выбрана первая ветвь в предложении `case`.

Наконец, в Ruby реализован оператор сопоставления с образцом `=~`. Традиционно он применяется для сопоставления строки с регулярным выражением. Но если вы найдете ему применение в других классах, то можете переопределить.

У операторов `==` и `=~` есть противоположные формы: `!=` и `!~` соответственно. Внутри они реализованы путем обращения значения основной формы. Это означает, что если, например, вы реализовали метод `==`, то метод `!=` получаете задаром.

11.1.8. Управление доступом к методам

В Ruby объект определяется, прежде всего, своим интерфейсом: теми методами, которые он раскрывает внешнему миру. Но при написании класса часто возникает необходимость во вспомогательных методах, вызывать которые извне класса опасно. Тут-то и приходит на помощь метод `private` класса `Module`.

Использовать его можно двумя способами. Если в теле класса или модуля вы вызовете `private` без параметров, то все последующие методы будут закрытыми в данном классе или модуле. Если же вы передадите ему список имен методов (в виде символов), то эти и только эти методы станут закрытыми. В листинге 11.5 показаны оба варианта.

Листинг 11.5. Закрытые методы

```
class Bank
  def open_safe
    # ...
  end

  def close_safe
    # ...
  end

  private :open_safe, :close_safe

  def make_withdrawal(amount)
    if access_allowed
      open_safe
      get_cash(amount)
      close_safe
    end
  end

  # Остальные методы закрытые.
```

```
private

def get_cash
  # ...
end

def access_allowed
  # ...
end
```

Поскольку методы из семейства `attr` просто определяют методы, метод `private` определяет и видимость атрибутов.

Реализация метода `private` может показаться странной, но на самом деле она весьма хитроумна. К закрытым методам нельзя обратиться, указав вызывающий объект; они вызываются только от имени неявно подразумеваемого объекта `self`. То есть вызвать закрытый метод из другого объекта не удастся: просто не существует способа указать объект, от имени которого данный метод вызывается. Заодно это означает, что закрытые методы доступны подклассам того класса, в котором определены, но опять же в рамках одного объекта.

Модификатор доступа `protected` налагает меньше ограничений. Защищенные методы доступны только экземплярам того класса, в котором определены, и его подклассов. Для защищенного метода разрешается указывать вызывающий объект, так что к ним можно обращаться из других объектов (при условии, что вызывающий и вызываемый объекты принадлежат одному классу). Обычно защищенные методы применяются для определения методов доступа, чтобы два объекта одного типа могли взаимодействовать. В следующем примере объекты класса `Person` можно сравнивать по возрасту, но сам возраст недоступен вне класса `Person`:

```
class Person
  def initialize(name, age)
    @name, @age = name, age
  end

  def <=>(other)
    age <=> other.age
  end

  attr_reader :name, :age
  protected :age
end
```

```
p1 = Person.new("fred", 31)
p2 = Person.new("agnes", 43)
compare = (p1 <=> p2)      # -1
x = p1.age                 # Ошибка!
```

Чтобы завершить картину, модификатор `public` делает метод открытым. Неудивительно!..

И последнее: методы, определенные вне любого класса и модуля (то есть на верхнем уровне программы), по умолчанию закрыты. Поскольку они определены в классе `Object`, то видимы глобально, но обращаться к ним с указанием вызывающего объекта нельзя.

11.1.9. Копирование объектов

Встроенные методы `Object#clone` и `#dup` порождают копию вызывающего объекта. Различаются они объемом копируемого контекста. Метод `#dup` копирует только само содержимое объекта, тогда как `clone` сохраняет и такие вещи, как синглетные классы, ассоциированные с объектом.

```
s1 = "cat"
```

```
def s1.upcase
  "CaT"
end
```

```
s1_dup   = s1.dup
s1_clone = s1.clone
s1                #=> "cat"
s1_dup.upcase     #=> "CaT"  (синглетный метод не копируется)
s1_clone.upcase   #=> "CaT"  (используется синглетный метод)
```

И `dup`, и `clone` выполняют поверхностное копирование, то есть копируют лишь содержимое самого вызывающего объекта. Если вызывающий объект содержит ссылки на другие объекты, то последние не копируются – копия будет ссылаться на те же самые объекты. Проиллюстрируем это на примере. Объект `arr2` – копия `arr1`, поэтому изменение элемента целиком, например `arr2[2]`, не оказывает влияния на `arr1`. Но исходный массив и его копия содержат ссылку на один и тот же объект `String`, поэтому изменение строки через `arr2` приведет к такому же изменению значения, на которое ссылается `arr1`.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = arr1.dup
```

```
arr2[2] = 99
arr2[1][2] = 'a'
```

```
arr1      # [1, "flapper", 3]
arr2      # [1, "flapper", 99]
```

Иногда необходимо глубокое копирование, при котором копируется все дерево объектов с корнем в исходном объекте. В этом случае между оригиналом и копией гарантированно не будет никакой интерференции. Ruby не предоставляет встроенного метода для глубокого копирования, но есть приемы, позволяющие достичь желаемого результата.

Самый «чистый» способ – потребовать, чтобы классы реализовывали метод `deep_copy`. Он мог бы рекурсивно обходить все объекты, на которые ссылается исходный объект, и вызывать для них метод `deep_copy`. Необходимо было бы еще

добавить метод `deep_copy` во все встроенные классы Ruby, которыми вы пользуетесь.

Но есть и более быстрый способ с использованием модуля `marshal`. Если вы сериализуете исходный объект, представив его в виде строки, а затем загрузите в новый объект, то этот новый объект будет копией исходного.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = Marshal.load(Marshal.dump(arr1))
```

```
arr2[2] = 99
arr2[1][2] = 'a'
```

```
arr1          # [1, "flipper", 3]
arr2          # [1, "flapper", 99]
```

Обратите внимание, что изменение строки через `arr2` не отразилось на строке, на которую ссылается `arr1`.

11.1.10. Метод `initialize_copy`

При копировании объекта методом `dup` или `clone` конструктор не вызывается. Копируется вся информация о состоянии.

Но что делать, если вам такое поведение не нужно? Рассмотрим пример:

```
class Document
  attr_accessor :title, :text
  attr_reader   :timestamp

  def initialize(title, text)
    @title, @text = title, text
    @timestamp = Time.now
  end
end

doc1 = Document.new("Random Stuff", File.read("somefile"))
sleep 300                                # Немного подождем...
doc2 = doc1.clone

doc1.timestamp == doc2.timestamp # true
# Оп... временные штампы одинаковы!
```

При создании объекта `Document` с ним ассоциируется временной штамп. При копировании объекта копируется и его временной штамп. А как быть, если мы хотим запомнить время, когда было выполнено копирование?

Для этого нужно определить метод `initialize_copy`. Он вызывается как раз при копировании объекта. Этот метод аналогичен `initialize` и позволяет полностью контролировать состояние объекта.

```
class Document                                # Определяем новый метод в классе.
  def initialize_copy(other)
    @timestamp = Time.now
  end
```

```
end

doc3 = Document.new("More Stuff",File.read("otherfile"))
sleep 300                                # Немного подождем...
doc4 = doc3.clone
```

```
doc3.timestamp == doc4.timestamp  # false
# Теперь временные штампы правильны.
```

Отметим, что метод `initialize_copy` вызывается после того, как вся информация скопирована. Поэтому мы и опустили строку:

```
@title, @text = other.title, other.text
```

Кстати, если метод `initialize_copy` пуст, то поведение будет такое же, как если бы он не был определен вовсе.

11.1.11. Метод `allocate`

Редко, но бывает, что нужно создать объект, не вызывая его конструктор (в обход метода `initialize`). Например, может случиться, что состояние объекта полностью определяется методами доступа к нему; тогда не нужно вызывать метод `new` (который вызовет `initialize`), разве что вы сами захотите это сделать. Представьте, что для инициализации состояния объекта вы собираете данные по частям: начать следует с «пустого» объекта, а не получить все данные заранее, а потом вызывать конструктор.

Метод `allocate` появился в версии Ruby 1.8, чтобы упростить решение этой задачи. Он возвращает «чистый», еще не инициализированный объект класса.

```
class Person
  attr_accessor :name, :age, :phone

  def initialize(n,a,p)
    @name, @age, @phone = n, a, p
  end
end

p1 = Person.new("John Smith",29,"555-1234")

p2 = Person.allocate

p p1.age    # 29
p p2.age    # nil
```

11.1.12. Модули

Для использования модулей в Ruby есть две основных причины. Первая — облегчить управление пространством имен; если поместить константы и методы в модуль, то будет меньше конфликтов имен. Хранящийся таким образом метод (метод модуля) вызывается с указанием имени модуля, то есть без вызывающего объекта. Точно так же вызывается и метод класса. Увидев вызовы вида `File.ctime` или `FileTest.exist?`, мы не можем определить по контексту, что `File` — это класс, а `FileTest` — модуль.

Вторая причина более интересна: мы можем использовать модуль как примесь. *Примеси* – это способ реализации множественного наследования, при котором наследуется только интерфейс.

Мы уже говорили о методах модуля, а как насчет методов экземпляра? Модуль – это не класс, у него не может быть экземпляров, а к методу экземпляра нельзя обратиться, не указав вызывающий объект.

Но оказывается, модуль может иметь методы экземпляра. Они становятся частью класса, который включил модуль директивой `include`.

```
module MyMod

  def meth1
    puts "Это метод 1."
  end

end

class MyClass

  include MyMod

  # ...

end

x = MyClass.new
x.meth1           # Это метод 1.
```

Здесь модуль `MyMod` подмешан к классу `MyClass`, а метод экземпляра `meth1` унаследован. Вы видели также, как директива `include` употребляется на верхнем уровне программы; в таком случае модуль подмешивается к классу `Object`.

А что происходит с методами модуля, если таковые определены? Если вы думаете, что они становятся методами класса, то ошибаетесь. Методы модуля не подмешиваются.

Но если такое поведение желательно, то его можно реализовать с помощью нехитрого трюка. Существует метод `append_features`, который можно переопределить. Он вызывается с параметром – «целевым» классом или модулем (в который включается данный модуль). Пример приведен в листинге 11.6.

Листинг 11.6. Включение модуля с переопределенным методом `append_features`

```
module MyMod

  def MyMod.append_features(someClass)
    def someClass.modmeth
      puts "Метод модуля (класса) "
    end
    super    # Этот вызов обязателен!
```

```
end

def meth1
  puts "Метод 1"
end

end

class MyClass

  include MyMod

  def MyClass.classmeth
    puts "Метод класса"
  end

  def meth2
    puts "Метод 2"
  end

end

x = MyClass.new

# Выводится:
MyClass.classmeth # Метод класса
x.meth1           # Метод 1
MyClass.modmeth   # Метод модуля (класса)
x.meth2           # Метод 2
```

Этот пример заслуживает детального рассмотрения. Во-первых, надо понимать, что метод `append_features` не просто вызывается в ходе выполнения `include`; на самом деле именно он и несет ответственность за включение. Поэтому-то вызов `super` необходим, без него оставшаяся часть модуля (в данном случае метод `meth1`) вообще не была бы включена.

Отметим также, что внутри тела `append_features` имеется определение метода. Выглядит это необычно, но работает, поскольку вложенное определение порождает синглетный метод (уровня класса или модуля). Попытка определить таким образом метод экземпляра привела бы к ошибке `Nested method error` (Ошибка при определении вложенного метода).

Модуль мог бы захотеть узнать, кто был инициатором примеси. Для этого тоже можно воспользоваться методом `append_features`, потому что класс-инициатор передается ему в качестве параметра.

Можно также подмешивать методы экземпляра модуля как методы класса. В листинге 11.7 приведен соответствующий пример.

Листинг 11.7. Методы экземпляра модуля становятся методами класса

```
module MyMod
```

```
  def meth3
    puts "Метод экземпляра модуля meth3"
    puts "может стать методом класса."
  end
```

```
end
```

```
class MyClass
```

```
  class << self      # Здесь self - это MyClass.
    include MyMod
  end
```

```
end
```

```
MyClass.meth3
```

```
# Выводится:
#   Метод экземпляра модуля meth3
#   может стать методом класса.
```

Здесь полезен метод `extend`. Тогда пример можно записать так:

```
class MyClass
  extend MyMod
end
```

Мы все время говорим о методах. А как насчет переменных экземпляра? Конечно, модуль может иметь собственные данные экземпляра, но обычно так не делают. Впрочем, если вы решили, что без этого никак не обойтись, ничто вас не остановит.

Можно подмешивать модуль к объекту, а не классу (например, методом `extend`), см. по этому поводу раздел 11.2.2.

Важно понимать еще одну вещь. В вашем классе можно определить методы, которые будут вызываться из примеси. Это удивительно мощный прием, знакомый тем, кто пользовался интерфейсами в Java.

Классический пример, с которым мы уже сталкивались ранее, – подмешивание модуля `Comparable` и определение метода `<=>`. Поскольку подмешанные методы могут вызывать метод сравнения, то мы получаем операторы `<`, `>`, `<=` и т.д.

Другой пример – подмешивание модуля `Enumerable` и определение метода `<=>` и итератора `each`. Тем самым мы получаем целый ряд полезных методов: `collect`, `sort`, `min`, `max` и `select`.

Можно также определять и собственные модули, ведущие себя подобным образом. Возможности ограничены только вашим воображением.

11.1.13. Трансформация или преобразование объектов

Иногда объект имеет нужный вид в нужное время, а иногда хочется преобразовать его во что-то другое или сделать вид, что он является чем-то, чем на самом деле не является. Всем известный пример – метод `to_s`.

Каждый объект можно тем или иным способом представить в виде строки. Но не каждый объект может успешно «прикинуться» строкой. Именно в этом и состоит различие между методами `to_s` и `to_str`. Рассмотрим этот вопрос подробнее.

При использовании метода `puts` или интерполяции в строку (в контексте `#{...}`) ожидается, что в качестве параметра будет передан объект `String`. Если это не так, объект просят преобразовать себя в `String`, посылая ему сообщение `to_s`. Именно здесь вы можете определить, как объект следует отобразить; просто реализуйте метод `to_s` в своем классе так, чтобы он возвращал подходящую строку.

```
class Pet

  def initialize(name)
    @name = name
  end

  # ...

  def to_s
    "Pet: #{@name}"
  end

end
```

Другие методы (например, оператор конкатенации строк `+`) не так требовательны, они ожидают получить нечто достаточно близкое к объекту `String`. В этом случае Мац решил, что интерпретатор не будет вызывать метод `to_s` для преобразования нестроковых аргументов, поскольку это могло бы привести к большому числу ошибок. Вместо этого вызывается более строгий метод `to_str`. Из всех встроенных классов только `String` и `Exception` реализуют `to_str`, и лишь `String`, `Regexp` и `Marshal` вызывают его. Увидев сообщение `TypeError: Failed to convert xyz into String`, можно смело сказать, что интерпретатор пытался вызвать `to_str` и потерпел неудачу.

Вы можете реализовать метод `to_str` и самостоятельно, например, если хотите, чтобы строку можно было конкатенировать с числом:

```
class Numeric

  def to_str
    to_s
  end

end

label = "число " + 9      # "число 9"
```

Аналогично обстоит дело и в отношении массивов. Для преобразования объекта в массив служит метод `to_a`, а метод `to_ary` вызывается, когда ожидается массив и ничего другого, например в случае множественного присваивания. Допустим, есть предложение такого вида:

```
a, b, c = x
```

Если `x` – массив из трех элементов, оно будет работать ожидаемым образом. Но если это не массив, интерпретатор попытается вызвать метод `to_ary` для преобразования в массив. В принципе это может быть даже синглетный метод (принадлежащий конкретному объекту). На само преобразование не налагается никаких ограничений, ниже приведен пример (нереалистичный), когда строка преобразуется в массив строк:

```
class String

  def to_ary
    return self.split("")
  end

end

str = "UFO"
a, b, c = str      # ["U", "F", "O"]
```

Метод `inspect` реализует другое соглашение. Отладчики, утилиты типа `irb` и метод отладочной печати `p` вызывают `inspect`, чтобы преобразовать объект к виду, пригодному для вывода на печать. Если вы хотите, чтобы во время отладки объект раскрывал свое внутреннее устройство, переопределите `inspect`.

Есть и еще одна ситуация, когда желательно выполнять такие преобразования «за кулисами». Пользователь языка ожидает, что `Fixnum` можно прибавить к `Float`, а комплексное число `Complex` разделить на рациональное. Но для проектировщика языка это проблема. Если метод `+` класса `Fixnum` получает аргумент типа `Float`, то что он должен с ним делать? Он знает лишь, как складывать значения типа `Fixnum`. Для решения проблемы в Ruby реализован механизм приведения типов `coerce`.

Когда оператор `+` (к примеру) получает аргумент, которого не понимает, он пытается привести вызывающий объект и аргумент к совместимым типам, а затем значения этих типов сложить. Общий принцип использования метода `coerce` прямолинеен:

```
class MyNumberSystem

  def +(other)
    if other.kind_of?(MyNumberSystem)
      result = some_calculation_between_self_and_other
      MyNumberSystem.new(result)
    else
      n1, n2 = other.coerce(self)
      n1 + n2
    end
  end

end
```

Метод `coerce` возвращает массив из двух элементов: аргумент и вызывающий объект, приведенные к совместимым типам.

В примере выше мы полагались на то, что класс аргумента умеет как-то выполнять приведение. Будь мы законопослушными гражданами, реализовали бы приведение и в собственном классе, чтобы он мог работать с числами других видов. Для этого нужно знать, с какими типами мы можем работать напрямую, и приводить объект к одному из этих типов, когда возникает необходимость. Если мы сами не знаем, как это сделать, следует спросить у родителя:

```
def coerce(other)
  if other.kind_of?(Float)
    return other, self.to_f
  elsif other.kind_of?(Integer)
    return other, self.to_i
  else
    super
  end
end
```

Конечно, чтобы этот пример работал, наш объект должен реализовывать методы `to_i` и `to_f`.

Метод `coerce` можно использовать для реализации автоматического преобразования строк в числа, как это делается в языке Perl:

```
class String

  def coerce(n)
    if self['.']
      [n, Float(self)]
    else
      [n, Integer(self)]
    end
  end
end
```

```
x = 1 + "23"      # 24
y = 23 * "1.23"   # 29.29
```

Впрочем, поступать так необязательно. Однако мы настоятельно рекомендуем реализовывать метод `coerce` при разработке разного рода числовых классов.

11.1.14. Классы, содержащие только данные (Struct)

Иногда нужно просто сгруппировать взаимосвязанные данные, не определяя никакие специфические методы обработки. Можно для этого создать класс:

```
class Address

  attr_accessor :street, :city, :state

  def initialize(street1, city, state)
    @street, @city, @state = street, city, state
  end
end
```

```
end
```

```
books = Address.new("411 Elm St", "Dallas", "TX")
```

Такое решение годится, но каждый раз прибегать к нему утомительно; к тому же здесь слишком много повторов. Тут-то и приходит на помощь встроенный класс `Struct`. Если вспомогательные методы типа `attr_accessor` определяют методы доступа к атрибутам, то `Struct` определяет целый класс, который может содержать только атрибуты. Такие классы называются *структурными шаблонами*.

```
Address = Struct.new("Address", :street, :city, :state)
```

```
books = Address.new("411 Elm St", "Dallas", "TX")
```

Зачем передавать первым параметром конструктора имя создаваемой структуры и присваивать результат константе (в данном случае `Address`)?

При вызове `Struct.new` для создания нового структурного шаблона на самом деле создается новый класс внутри самого класса `Struct`. Этому классу присваивается имя, переданное первым параметром, а остальные параметры становятся именами его атрибутов. При желании к вновь созданному классу можно было бы получить доступ, указав пространство имен `Struct`:

```
Struct.new("Address", :street, :city, :state)
```

```
books = Struct::Address.new("411 Elm St", "Dallas", "TX")
```

Создав структурный шаблон, вы вызываете его метод `new` для создания новых экземпляров данной конкретной структуры. Необязательно присваивать значения всем атрибутам в конструкторе. Опущенные атрибуты получают значение `nil`. После того как структура создана, к ее атрибутам можно обращаться с помощью обычного синтаксиса или указывая их имена в скобках в качестве индекса, как будто структура – это объект класса `hash`. Более подробную информацию о классе `Struct` можно найти в любом справочном руководстве (например, на сайте ruby-doc.org).

Кстати, не рекомендуем создавать структуру с именем `Tms`, так как уже есть предопределенный класс `Struct::Tms`.

11.1.15. Замораживание объектов

Иногда необходимо воспрепятствовать изменению объекта. Это позволяет сделать метод `freeze` (определенный в классе `Object`). По существу, он превращает объект в константу.

Попытка модифицировать замороженный объект приводит к исключению `TypeError`. В листинге 11.8 приведено два примера.

Листинг 11.8. Замораживание объекта

```
str = "Это тест. "  
str.freeze
```

```
begin
```

```
  str << " Не волнуйтесь."      # Попытка модифицировать.
```

```
rescue => err
```

```
  puts "#{err.class}#{err}"
```

```

end

arr = [1, 2, 3]
arr.freeze

begin
  arr << 4          # Попытка модифицировать.
rescue => err
  puts "#{err.class} #{err}"
end

# Выводится:
#   TypeError: can't modify frozen string
#   TypeError: can't modify frozen array

```

Однако имейте в виду, что метод `freeze` применяется к ссылке на объект, а не к переменной! Это означает, что любая операция, приводящая к созданию нового объекта, завершится успешно. Иногда это противоречит интуиции. В примере ниже мы ожидаем, что операция `+=` не выполнится, но все работает нормально. Дело в том, что присваивание – не вызов метода. Эта операция воздействует на переменные, а не на объекты, поэтому новый объект создается беспрепятственно. Старый объект по-прежнему заморожен, но переменная ссылается уже не на него.

```

str = "counter-"
str.freeze
str += "intuitive"      # "counter-intuitive"

arr = [8, 6, 7]
arr.freeze
arr += [5, 3, 0, 9]     # [8, 6, 7, 5, 3, 0, 9]

```

Почему так происходит? Предложение `a += x` семантически эквивалентно `a = a + x`. При вычислении выражения `a + x` создается новый объект, который затем присваивается переменной `a`! Все составные операторы присваивания работают подобным образом, равно как и другие методы. Всегда задавайте себе вопрос: «Что я делаю – создаю новый объект или модифицирую существующий?» И тогда поведение `freeze` не станет для вас сюрпризом.

Существует метод `frozen?`, который сообщает, заморожен ли данный объект.

```

hash = { 1 => 1, 2 => 4, 3 => 9 }
hash.freeze
arr = hash.to_a
puts hash.frozen?      # true
puts arr.frozen?       # false
hash2 = hash
puts hash2.frozen?     # true

```

Как видите (на примере `hash2`), замораживается именно объект, а не переменная.

11.2. Более сложные механизмы

Не все в модели ООП, реализованной в Ruby, одинаково очевидно. Что-то сложнее, что-то применяется реже. Линия раздела для каждого программиста проходит в разных местах. В этой части главы мы попытались собрать те средства, которые не так просты или не так часто встречаются в программах.

Иногда вы задаетесь вопросом, можно ли решить на Ruby ту или иную задачу. Краткий ответ таков: Ruby – богатый, динамический, объектно-ориентированный язык с широким набором разумно ортогональных средств; если нечто можно сделать на каком-то другом языке, то, скорее всего, можно и на Ruby.

Теоретически все полные по Тьюрингу языки более или менее одинаковы. Весь смысл проектирования языков в поиске осмысленной, удобной нотации. Читателю, сомневающемуся в важности нотации, стоит попробовать написать интерпретатор LISP на языке COBOL или выполнить деление чисел, записанных римскими цифрами.

Конечно, мы не хотим сказать, что любая задача на Ruby решается элегантно или естественно. Попытайся мы высказать такое утверждение, кто-нибудь очень быстро докажет, что мы не правы.

В этом разделе мы поговорим также о реализации на Ruby различных стилей программирования, например функционального и аспектно-ориентированного. Мы не претендуем на роль экспертов в этих областях, просто приводим мнения других. Относитесь к этому с долей скепсиса.

11.2.1. Отправка объекту явного сообщения

В статическом языке вы считаете очевидным, что имя вызываемой функции «зашито» в программу, это часть исходного текста. Динамический язык обладает в данном отношении большей гибкостью.

При любом вызове метода вы посылаете объекту сообщение. Обычно эти сообщения так же жестко «зашиты» в код, как и в статическом языке, но это необязательно. Можно написать программу, которая во время выполнения решает, какой метод вызывать. Метод `send` позволяет использовать `Symbol` для представления имени метода.

Пусть, например, имеется массив объектов, который нужно отсортировать, причем в качестве ключей сортировки хотелось бы использовать разные поля. Не проблема – можно просто написать специализированные блоки для сортировки. Не хотелось бы найти более элегантное решение, позволяющее обойтись одной процедурой, способной выполнить сортировку по любому указанному ключу. В листинге 11.9 такое решение приведено.

Этот пример был написан для первого издания книги. Теперь метод `sort_by` стал стандартным и даже более эффективным, поскольку реализует преобразование Шварца (по имени известного гуру в языке Perl Рэндала Шварца) и сохраняет преобразованные значения вместо многократного их вычисления. Впрочем, листинг 11.9 по-прежнему дает пример использования метода `send`.

Листинг 11.9. Сортировка по любому ключу

```
class Person
  attr_reader :name, :age, :height

  def initialize(name, age, height)
    @name, @age, @height = name, age, height
  end

  def inspect
    "#@name #@age #@height"
  end
end

class Array
  def sort_by(sym) # Наш вариант метода sort_by.
    self.sort {|x,y| x.send(sym) <=> y.send(sym) }
  end
end

people = []
people << Person.new("Hansel", 35, 69)
people << Person.new("Gretel", 32, 64)
people << Person.new("Ted", 36, 68)
people << Person.new("Alice", 33, 63)

p1 = people.sort_by(:name)
p2 = people.sort_by(:age)
p3 = people.sort_by(:height)

p p1 # [Alice 33 63, Gretel 32 64, Hansel 35 69, Ted 36 68]
p p2 # [Gretel 32 64, Alice 33 63, Hansel 35 69, Ted 36 68]
p p3 # [Alice 33 63, Gretel 32 64, Ted 36 68, Hansel 35 69]
```

Отметим еще, что синоним `__send__` делает в точности то же самое. Такое странное имя объясняется, вероятно, опасением, что имя `send` уже может быть задействовано (случайно или намеренно) для определенного пользователем метода.

11.2.2. Специализация отдельного объекта

.....

*Я солипсист и, признаться, удивлен,
что большинство из нас таковыми не являются.
Из письма, полученного Бертраном Расселом*

В большинстве объектно-ориентированных языков все объекты одного класса ведут себя одинаково. Класс – это шаблон, порождающий объекты с одним и тем же интерфейсом при каждом вызове конструктора.

Ruby ведет себя так же, но это не конец истории. Получив объект, вы можете изменить его поведение на лету. По сути дела, вы ассоциируете с объектом частный, анонимный подкласс, все методы исходного подкласса остаются доступными, но добавляется еще и поведение, уникальное для данного объекта. Поскольку это поведение присуще лишь данному объекту, оно встречается только один раз. Нечто, встречающееся только один раз, называется *синглетом* (singleton). Так, мы имеем синглетные методы и синглетные классы.

Слово «синглет» может стать источником путаницы, потому что оно употребляется и в другом смысле – как название хорошо известного паттерна проектирования, описывающего класс, для которого может существовать лишь один объект. Если вас интересует такое использование, обратитесь к библиотеке `singleton.rb`.

В следующем примере мы видим два объекта, оба строки. Для второго мы добавляем метод `upcase`, который переопределяет существующий метод с таким же именем.

```
a = "hello"
b = "goodbye"

def b.upcase      # Создать синглетный метод.
  gsub(/(.)(.)/) { $1.upcase + $2 }
end

puts a.upcase     # HELLO
puts b.upcase     # GoOdBye
```

Добавление синглетного метода к объекту порождает синглетный класс для данного объекта, если он еще не был создан ранее. Родителем синглетного класса является исходный класс объекта. (Можно считать, что это анонимный подкласс исходного класса.) Если вы хотите добавить к объекту несколько методов, то можете создать синглетный класс явно:

```
b = "goodbye"

class << b

  def upcase      # Создать синглетный метод.
    gsub(/(.)(.)/) { $1.upcase + $2 }
  end

  def upcase!
    gsub!(/(.)(.)/) { $1.upcase + $2 }
  end

end

puts b.upcase     # GoOdBye
puts b            # goodbye
b.upcase!
puts b            # GoOdBye
```

Отметим попутно, что у более «примитивных» объектов (например, `Fixnum`) не может быть добавленных синглетных методов. Связано это с тем, что такие объекты хранятся как непосредственные значения, а не как ссылки. Впрочем, реализация подобной функциональности планируется в будущих версиях Ruby (хотя непосредственные значения сохраняются).

Если вам приходилось разбираться в коде библиотек, то наверняка вы сталкивались с идиоматическим использованием синглетных классов. В определении класса иногда встречается такой код:

```
class SomeClass

  # Stuff...

  class << self
    # Какой-то код...
  end

  # ...продолжение.

end
```

В теле определения класса слово `self` обозначает сам определяемый класс, поэтому создание наследующего ему синглета модифицирует класс этого класса. Можно сказать, что методы экземпляра синглетного класса извне выглядят как методы самого класса.

```
class TheClass
  class << self
    def hello
      puts "hi"
    end
  end
end

# вызвать метод класса
TheClass.hello # hi
```

Еще одно распространенное применение такой техники – определение на уровне класса вспомогательных функций, к которым можно обращаться из других мест внутри определения класса. Например, мы хотим определить несколько функций доступа, которые преобразуют результат своей работы в строку. Можно, конечно, написать отдельно код каждой такой функции. Но есть и более элегантное решение – определить функцию уровня класса `accessor_string`, которая сгенерирует необходимые нам функции (как показано в листинге 11.10).

Листинг 11.10. Метод уровня класса `accessor_string`

```
class MyClass

  class << self

    def accessor_string(*names)
```

```

    names.each do |name|
      class_eval <<-EOF
        def #{name}
          @#{name}.to_s
        end
      EOF
    end
  end

  def initialize
    @a = [ 1, 2, 3 ]
    @b = Time.now
  end

  accessor_string :a, :b
end

```

```

o = MyClass.new
puts o.a          # 123
puts o.b          # Mon Apr 30 23:12:15 CDT 2001

```

Вы наверняка сможете придумать и другие, более изобретательные применения.

Метод `extend` подмешивает к объекту модуль. Методы экземпляра, определенные в модуле, становятся методами экземпляра объекта. Взгляните на листинг 11.11.

Листинг 11.11. Использование метода `extend`

```

module Quantifier

  def any?
    self.each { |x| return true if yield x }
    false
  end

  def all?
    self.each { |x| return false if not yield x }
    true
  end
end

list = [1, 2, 3, 4, 5]

list.extend(Quantifier)

flag1 = list.any? { |x| x > 5 }      # false

```

```
flag2 = list.any? {|x| x >= 5 }      # true
flag3 = list.all? {|x| x <= 10 }    # true
flag4 = list.all? {|x| x % 2 == 0 } # false
```

В этом примере к массиву `list` подмешаны методы `any?` и `all?`.

11.2.3. Вложенные классы и модули

Классы и модули можно вкладывать друг в друга произвольным образом. Программисты, приступающие к изучению Ruby, могут этого и не знать.

Основная цель данного механизма — упростить управление пространствами имен. Скажем, в класс `File` вложен класс `Stat`. Это помогает «инкапсулировать» класс `Stat` внутри тесно связанного с ним класса, а заодно оставляет возможность в будущем определить класс `Stat`, не конфликтуя с существующим (скажем, для сбора статистики).

Другой пример дает класс `Struct::Tms`. Любая новая структура `Struct` помещается в это пространство имен, не «загрязняя» расположенные выше, а `Tms` — в действительности тоже `Struct`.

Кроме того, вложенный класс можно создавать просто потому, что внешний мир не должен знать о нем или обращаться к нему. Иными словами, можно создавать целые классы, подчиняющиеся тому же принципу «сокрытия данных», которому переменные и методы экземпляра следуют на более низком уровне.

```
class BugTrackingSystem
```

```
  class Bug
    #...
  end

  #...
end
```

```
# Никто снаружи не знает о классе Bug.
```

Можно вкладывать класс в модуль, модуль в класс и т.д. Если вы придумаете интересные и изобретательные способы применения этой техники, дайте нам знать.

11.2.4. Создание параметрических классов

.....
Изучи правила, потом нарушай их.
Басё

Предположим, что нужно создать несколько классов, отличающихся только начальными значениями переменных уровня класса. Напомним, что переменная класса обычно инициализируется в самом определении класса.

```
class Terran
  @@home_planet = "Earth"
```

```

def Terran.home_planet
  @@home_planet
end

def Terran.home_planet=(x)
  @@home_planet = x
end

#...

end

```

Все замечательно, но что если нам нужно определить несколько подобных классов? Новичок подумает: «Ну так я просто определяю суперкласс!» (листинг 11.12).

Листинг 11.12. Параметрические классы: неправильное решение

```

class IntelligentLife  # Неправильный способ решения задачи!

  @@home_planet = nil

  def IntelligentLife.home_planet
    @@home_planet
  end

  def IntelligentLife.home_planet=(x)
    @@home_planet = x
  end

  #...
end

class Terran < IntelligentLife
  @@home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  @@home_planet = "Mars"
  #...
end

```

Но это работать не будет. Вызов `terrان.home_planet` напечатает не "Earth", а "Mars"! Почему так? Дело в том, что переменные класса – на практике не вполне переменные класса; они принадлежат не одному классу, а всей иерархии наследования. Переменная класса не копируется из родительского класса, а разделяется с родителем (и, стало быть, со всеми братьями).

Можно было бы вынести определение переменной класса в базовый класс, но тогда перестали бы работать определенные нами методы класса! Можно было бы исправить и это, перенеся определения в дочерние классы, однако тем самым

губится первоначальная идея, ведь таким образом объявляются отдельные классы без какой бы то ни было «параметризации».

Мы предлагаем другое решение. Отложим вычисление переменной класса до момента выполнения, воспользовавшись методом `class_eval`. Полное решение приведено в листинге 11.13.

Листинг 11.13. Параметрические классы: улучшенное решение

```
class IntelligentLife

  def IntelligentLife.home_planet
    class_eval("@@home_planet")
  end

  def IntelligentLife.home_planet=(x)
    class_eval("@@home_planet = #{x}")
  end

  #...
end

class Terran < IntelligentLife
  @@home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  @@home_planet = "Mars"
  #...
end

puts Terran.home_planet      # Earth
puts Martian.home_planet    # Mars
```

Не стоит и говорить, что механизм наследования здесь по-прежнему работает. Все методы и переменные экземпляра, определенные в классе `IntelligentLife`, наследуются классами `Terran` и `Martian`.

В листинге 11.14 предложено, наверное, наилучшее решение. В нем используются только переменные экземпляра, а от переменных класса мы вообще отказались.

Листинг 11.14. Параметрические классы: самое лучшее решение

```
class IntelligentLife
  class << self
    attr_accessor :home_planet
  end

  #...
end

class Terran < IntelligentLife
```

```

self.home_planet = "Earth"
#...
end

```

```

class Martian < IntelligentLife
  self.home_planet = "Mars"
  #...
end

```

```

puts Terran.home_planet      # Earth
puts Martian.home_planet    # Mars

```

Здесь мы открываем синглетный класс и определяем метод доступа `home_planet`. В двух подклассах определяются собственные методы доступа и устанавливается переменная. Теперь методы доступа работают строго в своих классах.

В качестве небольшого усовершенствования добавим еще вызов метода `private` в синглетный класс:

```
private :home_planet=
```

Сделав метод установки закрытым, мы запретили изменять значение вне иерархии данного класса. Как всегда, `private` реализует «рекомендательную» защиту, которая легко обходится. Но объявление метода закрытым по крайней мере говорит, что мы не хотели, чтобы метод вызывался вне определенного контекста.

Есть и другие способы решения этой задачи. Проявите воображение.

11.2.5. Использование продолжений для реализации генератора

Одно из самых трудных для понимания средств Ruby – *продолжение* (continuation). Это структурированный способ выполнить нелокальный переход и возврат. В объекте продолжения хранятся адрес возврата и контекст выполнения. В каком-то смысле это аналог функций `setjmp/longjmp` в языке C, но объем сохраняемого контекста больше.

Метод `callcc` из модуля `kernel` принимает блок и возвращает объект класса `Continuation`. Возвращаемый объект передается в блок как параметр, что еще больше все запутывает.

В классе `Continuation` есть всего один метод `call`, который обеспечивает нелокальный возврат в конец блока `callcc`. Выйти из метода `callcc` можно либо достигнув конца блока, либо вызвав метод `call`.

Считайте, что продолжение – что-то вроде операции «сохранить игру» в классических «бродилках». Вы сохраняете игру в точке, где все спокойно, а потом пробуете выполнить нечто потенциально опасное. Если эксперимент заканчивается гибелью, то вы восстанавливаете сохраненное состояние игры и пробуете пойти другим путем.

Самый лучший способ разобраться в продолжениях – посмотреть фильм «Бегн, Лола, беги».

Есть несколько хороших примеров того, как пользоваться продолжениями. Самые лучшие предложил Джим Вайрих (Jim Weirich). Ниже показано, как Джим

реализовал «генератор» после дискуссии еще с одним программистом на Ruby, Хью Сассе (Hugh Sasse).

Идея генератора навеяна методом `suspend` из языка Icon (он есть также в Prolog), который позволяет возобновить выполнение функции с места, следующего за тем, где она в последний раз вернула значение. Хью называет это «`yield` наоборот».

Библиотека `generator` теперь входит в дистрибутив Ruby. Дополнительную информацию по этому вопросу вы найдете в разделе 8.3.7.

В листинге 11.15 представлена предложенная Джимом реализация генератора чисел Фибоначчи. Продолжения применяются для того, чтобы сохранить состояние между вызовами.

Листинг 11.15. Генератор чисел Фибоначчи

```
class Generator

  def initialize
    do_generation
  end

  def next
    callcc do |here|
      @main_context = here;
      @generator_context.call
    end
  end

  private

  def do_generation
    callcc do |context|
      @generator_context = context;
      return
    end
    generating_loop
  end

  def generate(value)
    callcc do |context|
      @generator_context = context;
      @main_context.call(value)
    end
  end
end

# Порождаем подкласс и определяем метод generating_loop.

class FibGenerator < Generator
  def generating_loop
    generate(1)
    a, b = 1, 1
  end
end
```



```

loop do
  generate(b)
  a, b = b, a+b
end
end
end

# Создаем объект этого класса...

fib = FibGenerator.new

puts fib.next      # 1
puts fib.next      # 1
puts fib.next      # 2
puts fib.next      # 3
puts fib.next      # 5
puts fib.next      # 8
puts fib.next      # 13

# И так далее...

```

Есть, конечно, и более практичные применения продолжений. Один из примеров – каркас *Borges* для разработки Web-приложений (названный в честь Хорхе Луиса Борхеса), который построен по образу *Seaside*. В этой парадигме традиционный поток управления в Web-приложении «вывернут с изнанки на лицо», так что логика представляется «нормальной». Например, вы отображаете страницу, получаете результат из формы, отображаете следующую страницу и так далее, ни в чем не противореча интуитивным ожиданиям.

Проблема в том, что продолжение – «дорогая» операция. Необходимо сохранить состояние и потратить заметное время на переключение контекста. Если производительность для вас критична, прибегайте к продолжениям с осторожностью.

11.2.6. Хранение кода в виде объекта

Неудивительно, что Ruby предлагает несколько вариантов хранения фрагмента кода в виде объекта. В этом разделе мы рассмотрим объекты *Proc*, *Method* и *UnboundMethod*.

Встроенный класс *Proc* позволяет обернуть блок в объект. Объекты *Proc*, как и блоки, являются *замыканиями*, то есть запоминают контекст, в котором были определены.

```

myproc = Proc.new { |a| puts "Параметр равен #{a}" }

myproc.call(99)      # Параметр равен 99

```

Кроме того, Ruby автоматически создает объект *Proc*, когда метод, последний параметр которого помечен амперсандом, вызывается с блоком в качестве параметра:

```

def take_block(x, &block)
  puts block.class

```

```
x.times { |i| block[i, i*i] }
end
```

```
take_block(3) { |n,s| puts "#{n} в квадрате равно #{s}" }
```

В этом примере демонстрируется также применение квадратных скобок как синонима метода `call`. Вот что выводится в результате исполнения:

```
Proc
0 в квадрате 0
1 в квадрате 1
2 в квадрате 4
```

Объект `Proc` можно передавать методу, который ожидает блок, предварив имя знаком `&`:

```
myproc = proc { |n| print n, "... " }
(1..3).each(&myproc) # 1... 2... 3...
```

Ruby позволяет также превратить метод в объект. Исторически для этого применяется метод `Object#method`, который создает объект класса `Method` как замыкание в конкретном объекте.

```
str = "cat"
meth = str.method(:length)
```

```
a = meth.call # 3 (длина "cat")
```

```
str << "erpillar"
```

```
b = meth.call # 11 (длина "caterpillar")
```

```
str = "dog"
```

Обратите внимание на следующий вызов! Переменная `str` теперь ссылается на новый объект ("dog"), но `meth` по-прежнему связан со старым объектом.

```
c = meth.call # 11 (длина "caterpillar")
```

Начиная с версии Ruby 1.6.2, можно также применять метод `Module#instance_method` для создания объектов `UnboundMethod`. С их помощью представляется метод, ассоциированный с классом, а не с конкретным объектом. Прежде чем вызывать объект `UnboundMethod`, нужно связать его с каким-то объектом. Результатом операции связывания является объект `Method`, который можно вызывать как обычно:

```
umeth = String.instance_method(:length)
```

```
m1 = umeth.bind("cat")
m1.call # 3
```

```
m2 = umeth.bind("caterpillar")
m2.call # 11
```

Явное связывание делает объект `UnboundMethod` интуитивно более понятным, чем `Method`.

11.2.7. Как работает включение модулей?

Когда модуль включается в класс, Ruby на самом деле создает прокси-класс, являющийся непосредственным родителем данного класса. Возможно, вам это покажется интуитивно очевидным, возможно, нет. Все методы включаемого модуля «маскируются» методами, определенными в классе.

```
module MyMod
  def meth
    "из модуля"
  end
end

class ParentClass
  def meth
    "из родителя"
  end
end

class ChildClass < ParentClass
  include MyMod
  def meth
    "из потомка"
  end
end

x = ChildClass.new
p x.meth           # Из потомка.
```

Выглядит это как настоящее наследование: все, что потомок переопределил, становится действующим определением вне зависимости от того, вызывается ли `include` до или после переопределения.

Вот похожий пример, в котором метод потомка вызывает `super`, а не просто возвращает строку. Как вы думаете, что будет возвращено?

Модуль `MyMod` и класс `ParentClass` не изменились.

```
class ChildClass < ParentClass
  include MyMod
  def meth
    "Из потомка: super = " + super
  end
end

x = ChildClass.new
p x.meth           # Из потомка: super = из модуля
```

Отсюда видно, что модуль действительно является новым родителем класса. А что если мы точно также вызовем `super` из модуля?

```
module MyMod
  def meth
```

```
"Из модуля: super = " + super
end
end

# ParentClass не изменился.

class ChildClass < ParentClass
  include MyMod
  def meth
    "Из потомка: super = " + super
  end
end

x = ChildClass.new
p x.meth      # Из потомка: super = из модуля: super = из родителя.
```

Метод `meth`, определенный в модуле `MyMod`, может вызвать `super` только потому, что в суперклассе (точнее, хотя бы в одном из его предков) действительно есть метод `meth`. А что произошло бы, вызови мы этот метод при других обстоятельствах?

```
module MyMod
  def meth
    "Из модуля: super = " + super
  end
end

class Foo      include MyMod
end

x = Foo.new
x.meth
```

При выполнении этого кода мы получили бы ошибку `NoMethodError` (или обращение к методу `method_missing`, если бы таковой существовал).

11.2.8. Оpoznание параметров, заданных по умолчанию

В 2004 году Ян Макдональд (Ian Macdonald) задал в списке рассылки вопрос: «Можно ли узнать, был ли параметр задан вызывающей программой или взято значение по умолчанию?» Вопрос интересный. Не каждый день он возникает, но от того не менее интересен.

Было предложено по меньшей мере три решения. Самое удачное и простое нашел Нобу Накада (Nobu Nakada). Оно приведено ниже:

```
def meth(a, b=(flag=true; 345))
  puts "b равно #{b}, a flag равно #{flag.inspect}"
end

meth(123)          # b равно 345, a flag равно true
meth(123,345)      # b равно 345, a flag равно nil
meth(123,456)      # b равно 456, a flag равно nil
```

Как видим, этот подход работает даже, если вызывающая программа явно указала значение параметра, совпадающее с подразумеваемым по умолчанию. Трюк становится очевидным, едва вы его увидите: выражение в скобках устанавливает локальную переменную `flag` в `true`, а затем возвращает значение по умолчанию 345. Это дань могуществу Ruby.

11.2.9. Делегирование или перенаправление

В Ruby есть две библиотеки, которые предлагают решение задачи о делегировании или перенаправлении вызовов методов другому объекту. Они называются `delegate` и `forwardable`; мы рассмотрим обе.

Библиотека `delegate` предлагает три способа решения задачи.

Класс `SimpleDelegator` полезен, когда объект, которому делегируется управление (делегат), может изменяться на протяжении времени жизни делегирующего объекта. Чтобы выбрать объект-делегат, используется метод `__setobj__`.

Однако мне этот способ представляется слишком примитивным. Поскольку я не думаю, что это существенно лучше, чем то же самое, сделанное вручную, задерживаться на классе `SimpleDelegator` не стану.

Метод верхнего уровня `DelegateClass` принимает в качестве параметра класс, которому делегируется управление. Затем он создает новый класс, которому мы можем унаследовать. Вот пример создания класса `Queue`, который делегирует объекту `Array`:

```
require 'delegate'

class MyQueue < DelegateClass(Array)

  def initialize(arg=[])
    super(arg)
  end

  alias_method :enqueue, :push
  alias_method :dequeue, :shift
end

mq = MyQueue.new

mq.enqueue(123)
mq.enqueue(234)

p mq.dequeue      # 123
p mq.dequeue      # 234
```

Можно также унаследовать класс `Delegator` и реализовать метод `__getobj__`; именно таким образом реализован класс `SimpleDelegator`. При этом мы получаем больший контроль над делегированием.

Но если вам необходим больший контроль, то, вероятно, вы все равно осуществляете делегирование на уровне отдельных методов, а не класса в целом. Тогда лучше воспользоваться библиотекой `forwardable`. Вернемся к примеру очереди:

```
require 'forwardable'

class MyQueue
  extend Forwardable

  def initialize(obj={})
    @queue = obj    # Делегировать этому объекту.
  end

  def_delegator :@queue, :push, :enqueue

  def_delegator :@queue, :shift, :dequeue

  def_delegators :@queue, :clear, :empty?, :length, :size, :<<

  # Прочий код...
end
```

Как видно из этого примера, метод `def_delegator` ассоциирует вызов метода (скажем, `enqueue`) с объектом-делегатом `@queue` и одним из методов этого объекта (`push`). Иными словами, когда мы вызываем метод `enqueue` для объекта `MyQueue`, производится делегирование методу `push` объекта `@queue` (который обычно является массивом).

Обратите внимание, мы пишем `:@queue`, а не `:queue` или `@queue`. Объясняется это тем, как написан класс `Forwardable`; можно было бы сделать и по-другому.

Иногда нужно делегировать методы одного объекта одноименным методам другого объекта. Метод `def_delegators` позволяет задать произвольное число таких методов. Например, в примере выше показано, что вызов метода `length` объекта `MyQueue` приводит к вызову метода `length` объекта `@queue`.

В отличие от первого примера, остальные методы делегирующим объектом просто не поддерживаются. Иногда это хорошо, ведь не хотите же вы вызывать метод `[ ]` или `[ ]=` для очереди; если вы так поступаете, то очередь перестает быть очередью.

Отметим еще, что показанный выше код позволяет вызывающей программе передавать объект конструктору (для использования в качестве объекта-делегата). В полном соответствии с духом «утилизации» это означает, что я могу выбирать вид объекта, которому делегируется управление, коль скоро он поддерживает те методы, которые вызываются в программе.

Например, все приведенные ниже вызовы допустимы. (В последних двух предполагается, что предварительно было выполнено предложение `require 'thread'`.)

```
q1 = MyQueue.new           # Используется любой массив.
q2 = MyQueue.new(my_array) # Используется конкретный массив.
q3 = MyQueue.new(Queue.new) # Используется Queue (thread.rb).
q4 = MyQueue.new(SizedQueue.new) # Используется SizedQueue (thread.rb).
```

Так, объекты `q3` и `q4` волшебным образом становятся безопасными относительно потоков, поскольку делегируют управление безопасному в этом отношении объекту (если, конечно, какой-нибудь не показанный здесь код не нарушит эту гарантию).

Существует также класс `SingleForwardable`, который воздействует на один экземпляр, а не на класс в целом. Это полезно, если вы хотите, чтобы какой-то конкретный объект делегировал управление другому объекту, а все остальные объекты того же класса так не поступали.

Быть может, вы задумались о том, что лучше – делегирование или наследование. Но это неправильный вопрос. В некоторых ситуациях делегирование оказывается более подходящим решением. Предположим, к примеру, что имеется класс, у которого уже есть родитель. Унаследовать еще от одного родителя мы не можем (в Ruby множественное наследование запрещено), но делегирование в той или иной форме вполне допустимо.

11.2.10. Автоматическое определение методов чтения и установки на уровне класса

Мы уже рассматривали методы `attr_reader`, `attr_writer` и `attr_accessor`, которые немного упрощают определение методов чтения и установки атрибутов экземпляра. А как быть с атрибутами уровня класса?

В Ruby нет аналогичных средств для их автоматического создания. Но можно написать нечто подобное самостоятельно.

В первом издании этой книги была показана хитроумная схема на основе метода `class_eval`. С ее помощью мы создали такие методы, как `cattr_reader` и `cattr_writer`.

Но есть более простой путь. Откроем синглетный класс и воспользуемся в нем семейством методов `attr`. Получающиеся переменные экземпляра для синглетного класса станут переменными экземпляра класса. Часто это оказывается лучше, чем переменные класса, поскольку они принадлежат данному и только данному классу, не распространяясь вверх и вниз по иерархии наследования.

```
class MyClass

  @alpha = 123                                # Инициализировать @alpha.

  class << self
    attr_reader :alpha                        # MyClass.alpha()
    attr_writer :beta                        # MyClass.beta=()
    attr_accessor :gamma                    # MyClass.gamma() и
  end                                       #   MyClass.gamma=()

  def MyClass.look
    puts "#@alpha, #@beta, #@gamma"
  end

  #...
end

puts MyClass.alpha                            # 123
MyClass.beta = 456
MyClass.gamma = 789
```

puts MyClass.gamma

789

MyClass.look

123, 456, 789

Как правило, класс без переменных экземпляра бесполезен. Но здесь мы их для краткости опустили.

11.2.11. Поддержка различных стилей программирования

.....

Brother, can you paradigm?

Граффити на здании IBM в Остине, 1989

В различных кругах популярны разные философии программирования. Часто их трудно охарактеризовать с точки зрения объектной ориентированности или динамичности, а некоторые вообще не зависят от того, является ли язык динамическим или объектно-ориентированным.

Поскольку мы отнюдь не эксперты в этих вопросах, будем полагаться в основном на чужие слова. Так что воспринимайте то, что написано ниже, с некоторой долей скепсиса.

Некоторые программисты предпочитают стиль ООП на основе прототипов (или ООП без классов). В этом мире объект не описывается как экземпляр класса, а строится с нуля. На базе такого прототипа могут создаваться другие объекты. В Ruby есть рудиментарная поддержка такого стиля программирования, поскольку допускаются синглетные методы, имеющиеся только у отдельных объектов, а метод `clone` клонирует синглеты. Интересующийся читатель может также обратить внимание на простой класс `OpenStruct` для построения объектов в духе языка Python; не забывайте также о том, как работает метод `method_missing`.

Парочка ограничений в Ruby препятствует реализации ООП без классов. Некоторые объекты, например `Fixnum`, хранятся как непосредственные значения, а не ссылки, поэтому не могут иметь синглетных методов. В будущем ситуация, вероятно, изменится, но пока невозможно предсказать, когда это произойдет.

В функциональном программировании (ФП) упор делается на вычисление выражений, а не на исполнение команд. *Функциональным* называется язык, поддерживающий ФП, но на этом всякая определенность заканчивается. Почти все согласятся, что Haskell – настоящий функциональный язык, а Ruby таковым, безусловно, не является.

Но в Ruby есть минимальная поддержка ФП, он располагает богатым набором методов для манипулирования массивами (списками) и поддерживает объекты `Proc`, позволяющие инкапсулировать и многократно вызывать код. Ruby также допускает сцепление методов, весьма распространенное в ФП. Правда, дело портят «восклицательные» методы (например, `sort!` или `gsub!`), которые возвращают `nil`, если вызываемый объект не изменился в результате выполнения.

Предпринимались попытки создать библиотеку, которая стала бы «уровнем совместимости» с ФП, заимствуя некоторые идеи из языка Haskell. Пока эти попытки ни к чему законченному не привели.

Интересна идея аспектно-ориентированного программирования (АОП). Это попытка расчесть модульную структуру программы. Иными словами, некоторые задачи и механизмы системы разбросаны по разным участкам кода, а не собраны в одном месте. То есть мы пытаемся придать модульность вещам, которым в традиционном объектно-ориентированном или процедурном программировании с трудом поддаются «модуляризации». Взгляд на программу оказывается перпендикулярен обычному.

Разумеется, Ruby создавался без учета АОП. Но это гибкий и динамический язык, поэтому не исключено, что такой подход может быть реализован в виде библиотеки. Уже сейчас существует библиотека `AspectR`, представляющая собой первую попытку внести аспектно-ориентированные черты в Ruby; последнюю ее версию можно найти в архиве приложений Ruby.

Идея «проектирования по контракту» (`Design by Contract – DBC`) хороша знакома поклонникам языка `Eiffel`, хотя и вне этого круга она тоже известна. Смысл состоит в том, что некоторый кусок кода (метод или класс) реализует контракт; чтобы код правильно работал, должны выполняться определенные предусловия, и тогда гарантируется, что по завершении работы будут выполнены некоторые постусловия. Надежность системы можно существенно повысить, введя возможность формулировать контракт явно и автоматически проверять его во время выполнения. Полезность такого подхода подкрепляется наследованием информации о контракте при расширении классов.

В язык `Eiffel` методология `DBC` встроена явно, в Ruby – нет. Однако имеется по крайней мере две работающие библиотеки, реализующие `DBC`, и мы рекомендуем вам выбрать одну из них и изучить внимательнее.

Паттерны проектирования стали темой оживленных дискуссий на протяжении последних нескольких лет. Конечно, они мало зависят от конкретного языка и могут быть реализованы на самых разных языках. Но необычайная гибкость Ruby, возможно, делает их практически более полезными, чем в других средах. Хорошо известные примеры приведены в других местах; паттерн `Visitor` (Посетитель) реализуется стандартным итератором `each`, а многие другие паттерны входят в стандартный дистрибутив Ruby (библиотеки `delegator.rb` и `singleton.rb`).

С каждым днем все больше приверженцев завоевывает методология *экстремального программирования* (`Extreme Programming – XP`), поощряющая, среди прочего, раннее тестирование и постоянную переработку (рефакторинг).

`XP` – технология, не зависящая от языка, хотя к некоторым языкам она, возможно, более приспособлена. Разумеется, на наш взгляд, в Ruby рефакторинг реализуется проще, чем во многих языках, но это субъективное мнение. Однако, наличие библиотеки `Test::Unit` (и других) позволяет «поженить» Ruby и `XP`. Эта библиотека облегчает автономное тестирование компонентов, она функциональна богата, проста в использовании и доказала свою полезность в ходе разработки эксплуатируемых в настоящее время программ на Ruby. Мы горячо поддерживаем рекомендуемое `XP` раннее и частое тестирование, а тем, кто желает воплотить этот совет в Ruby, предлагаем ознакомиться с `Test::Unit`. (`ZenTest` – еще один отличный пакет, включающий некоторые возможности, которые в `Test::Unit` отсутствуют.)

Когда вы будете читать этот раздел, многие обсуждаемые в нем технологии совершенствуются. Как обычно, самую актуальную информацию можно найти на следующих ресурсах:

Конференция comp.lang.ruby

Архив приложений Ruby

rubyforge.org

ruby-doc.org

Есть и другие полезные ресурсы, особенно для тех, кто говорит по-японски. Трудно перечислять онлайн-ресурсы в печатном издании, поскольку они постоянно изменяются. Поисковая машина – ваш лучший друг.

11.3. Динамические механизмы

.....
*Скайнет осознал себя в 2:14 утра по восточному времени
29 августа 1997 года.
Терминатор 2, Судный День*

Многие читатели имеют опыт работы со статическими языками, например С. Им я адресую риторический вопрос: «Можете ли вы представить себе написанную на С функцию, которая принимает строку, рассматривает ее как имя переменной и возвращает значение этой переменной?»

Нет? А как насчет того, чтобы удалить или заменить определение функции? А перехватить обращения к несуществующим функциям? Или узнать имя вызывающей функции? Или автоматически получить список определенных пользователем элементов программы (например, перечень всех написанных вами функций)?

В Ruby все это возможно. Такая гибкость во время выполнения, способность опрашивать и изменять программные элементы во время выполнения намного упрощают решение задач. Утилиту трассировки выполнения, отладчик, профилировщик – все это легко написать на Ruby и для Ruby. Хорошо известные программы `irb` и `xmp`, используя динамические возможности Ruby, творят это волшебство.

К подобным возможностям нужно привыкнуть, их легко употребить во вред. Все эти идеи появились отнюдь не вчера (они стары по крайней мере так же, как язык LISP) и считаются «проверенными и доказанными» в сообществах пользователей Scheme и Smalltalk. Даже в языке Java, который так многим обязан С и C++, есть некоторые динамические средства, поэтому мы ожидаем, что со временем их популярность будет только расти.

11.3.1. Динамическая интерпретация кода

Глобальная функция `eval` компилирует и исполняет строку, содержащую код на Ruby. Это очень мощный (и вместе с тем опасный) механизм, поскольку позволяет строить подлежащий исполнению код во время работы программы. Например, в следующем фрагменте считываются строки вида «имя = выражение», затем каждое выражение вычисляется, а результат сохраняется в хэше, индексированном именем переменной.

```
parameters = {}
```

```
ARGF.each do |line|
  name, expr = line.split(/\s*=\s*/, 2)
  parameters[name] = eval expr
end
```

Пусть на вход подаются следующие строки:

```
a = 1
b = 2 + 3
c = 'date'
```

Тогда в результате мы получим такой хэш: {"a"=>1, "b"=>5, "c"=>"Mon Apr 30 21:17:47 CDT 2001\n"}. На этом примере демонстрируется также опасность вычисления с помощью `eval` строк, содержимое которых вы не контролируете; злонамеренный пользователь может подsunуть строку `d='rm *'` и стереть всю вашу дневную работу.

В Ruby есть еще три метода, которые интерпретируют код «на лету»: `class_eval`, `module_eval` и `instance_eval`. Первые два – синонимы, и все они выполняют одно и то же: интерпретируют строку или блок, но при этом изменяют значение псевдопеременной `self` так, что она указывает на объект, от имени которого эти методы вызваны. Наверное, чаще всего метод `class_eval` применяется для добавления методов в класс, на который у вас имеется только ссылка. Мы продемонстрируем это в коде метода `hook_method` в примере утилиты `Trace` в разделе 11.3.13. Другие примеры вы найдете в динамических библиотечных модулях, например `delegate.rb`.

Метод `eval` позволяет также вычислять локальные переменные в контексте, не принадлежащем их области видимости. Мы не рекомендуем легкомысленно относиться к этой возможности, но знать, что она существует, полезно.

Ruby ассоциирует локальные переменные с блоками, с определениями высокоуровневых конструкций (класса, модуля и метода) и с верхним уровнем программы (кодом, расположенным вне любых определений). С каждой из этих областей видимости ассоциируются привязки переменных и другие внутренние детали. Наверное, самым главным потребителем информации о привязках является программа `irb` – интерактивная оболочка для Ruby, которая пользуется привязками, чтобы отделить собственные переменные от тех, которые принадлежат вводимой программе.

Можно инкапсулировать текущую привязку в объект с помощью метода `Kernel#binding`. Тогда вы сможете передать привязку в виде второго параметра методу `eval`, установив контекст исполнения для интерпретируемого кода.

```
def some_method
  a = "local variable"
  return binding
end
```

```
the_binding = some_method
eval "a", the_binding # "local variable"
```

Интересно, что информация о наличии блока, ассоциированного с методом, сохраняется как часть привязки, поэтому возможны такие трюки:

```
def some_method
  return binding
end

the_binding = some_method { puts "hello" }
eval "yield", the_binding           # hello
```

11.3.2. Метод const_get

Метод const_get получает значение константы с заданным именем из модуля или класса, которому она принадлежит.

```
str = "PI"
Math.const_get(str)      # Значение равно Math::PI.
```

Это способ избежать обращения к методу eval, которое иногда считается неэлегантным. Такой подход дешевле с точки зрения потребления ресурсов и безопаснее. Есть и другие аналогичные методы: instance_variable_set, instance_variable_get и define_method.

Метод const_get действительно работает быстрее, чем eval. В неформальных тестах – на 350% быстрее, хотя у вас может получиться другой результат. Но так ли это важно? Ведь в тестовой программе на 10 миллионов итераций цикла все равно ушло менее 30 секунд.

Истинная полезность метода const_get в том, что его проще читать, он более специфичен и лучше самодокументирован. Даже если бы он был всего лишь синонимом eval, все равно это стало бы большим шагом вперед.

11.3.3. Динамическое создание экземпляра класса, заданного своим именем

Такой вопрос мы видели многократно. Пусть дана строка, содержащая имя класса; как можно создать экземпляр этого класса?

Правильный способ – воспользоваться методом const_get, который мы только что рассмотрели. Имена всех классов в Ruby – константы в «глобальном» пространстве имен, то есть члены класса Object.

```
classname = "Array"

klass = Object.const_get(classname)
x = klass.new(4, 1)      # {1, 1, 1, 1}
```

А если имена вложены? Как выясняется, следующий код не работает:

```
class Alpha
  class Beta
    class Gamma
      FOOBAR = 237
    end
  end
end
```

```
str = "Alpha::Beta::Gamma::FOOBAR"
val = Object.const_get(str)           # Ошибка!
```

Дело в том, что метод `const_get` недостаточно «умен», чтобы распознать такие вложенные имена. Впрочем, в следующем примере приведена работающая идиома:

```
# Структура класса та же
```

```
str = "Alpha::Beta::Gamma::FOOBAR"
val = str.split("::").inject(Object) {|x,y| x.const_get(y)} # 237
```

Такой код встречается часто (и демонстрирует интересное применение `inject`).

11.3.4. Получение и установка переменных экземпляра

Отвечая на пожелание употреблять `eval` как можно реже, в Ruby теперь включены методы, которые могут получить или присвоить новое значение переменной экземпляра, имя которой задано в виде строки:

```
class MyClass
  attr_reader :alpha, :beta

  def initialize(a,b,g)
    @alpha, @beta, @gamma = a, b, g
  end
end

x = MyClass.new(10,11,12)

x.instance_variable_set("@alpha",234)
p x.alpha                               # 234

x.instance_variable_set("@gamma",345)   # 345
v = x.instance_variable_get("@gamma")   # 345
```

Прежде всего, отметим, что имя переменной должно начинаться со знака `@`, иначе произойдет ошибка. Если это кажется вам неочевидным, вспомните, что метод `attr_accessor` (и ему подобные) принимает для формирования имени метода символ, поэтому-то знак `@` и опускается.

Не нарушает ли существование таких методов принцип инкапсуляции? Нет. Конечно, эти методы потенциально опасны. Пользоваться ими следует с осторожностью, а не при всяком удобном случае. Но нельзя говорить, что инкапсуляция нарушена, не видя, как эти инструменты применяются в конкретном случае. Если это делается обдуманно, ради ясно осознанной цели, то все хорошо. Если же цель состоит в том, чтобы нарушить проект или обойти неудачное проектное решение, это печально. Ruby намеренно предоставляет доступ к внутренним деталям объектов тем, кому это действительно нужно; ответственный программист не станет пользоваться свободой во вред.

11.3.5. Метод `define_method`

Помимо ключевого слова `def`, единственный нормальный способ добавить метод в класс или объект – воспользоваться методом `define_method`, причем он позволяет сделать это во время выполнения.

Конечно, в Ruby практически все происходит во время выполнения. Если окружить определение метода обращениями к `puts`, как в примере ниже, вы это сами увидите.

```
class MyClass
  puts "до"

  def meth
    #...
  end

  puts "после"
end
```

Но внутри тела метода или в другом аналогичном месте нельзя заново открыть класс (если только это не синглетный класс). В таком случае в прежних версиях Ruby приходилось прибегать к помощи `eval`, теперь же у нас есть метод `define_method`. Он принимает символ (имя метода) и блок (тело метода).

Первая (ошибочная) попытка воспользоваться этим методом могла бы выглядеть так:

```
# Не работает, так как метод define_method закрытый.

if today =~ /Saturday|Sunday/
  define_method(:activity) { puts "Отдыхаем!" }
else
  define_method(:activity) { puts "Работаем!" }
end
```

activity

Поскольку `define_method` — закрытый метод, приходится поступать так:

```
# Работает (Object — это контекст верхнего уровня).
```

```
if today =~ /Saturday|Sunday/
  Object.class_eval { define_method(:activity) { puts "Отдыхаем!" } }
else
  Object.class_eval { define_method(:activity) { puts "Работаем!" } }
end
```

activity

Можно было бы поступить так же внутри определения класса (в применении к классу `Object` или любому другому). Такое редко бывает оправданно, но если вы можете сделать это внутри определения класса, вопрос о закрытости не встает.

```
class MyClass
  define_method(:mymeth) { puts "Это мой метод." }
end
```

Есть еще один трюк: включить в класс метод, который сам вызывает `define_method`, избавляя от этого программиста:

```
class MyClass
  def self.new_method(name, &block)
    define_method(name, &block)
  end
end

MyClass.new_method(:mymeth) { puts "Это мой метод." }
x = MyClass.new
x.mymeth           # Печатается "Это мой метод."
```

То же самое можно сделать и на уровне экземпляра, а не класса:

```
class MyClass
  def new_method(name, &block)
    self.class.send(:define_method,name, &block)
  end
end

x = MyClass.new
x.new_method(:mymeth) { puts "Это мой метод." }
x.mymeth           # Печатается "Это мой метод."
```

Здесь метод экземпляра тоже определен динамически. Изменился только способ реализации метода `new_method`. Обратите внимание на трюк с `send`, позволивший нам обойти закрытость метода `define_method`. Он работает, потому что в текущей версии Ruby метод `send` позволяет вызывать закрытые методы. (Некоторые сочтут это «дыркой»; как бы то ни было, пользоваться этим механизмом следует с осторожностью.)

По поводу метода `define_method` нужно сделать еще одно замечание. Он принимает блок, а в Ruby блок – замыкание. Это означает, что в отличие от обычного определения метода, мы запоминаем контекст, в котором метод был определен. Следующий пример практически бесполезен, но этот момент иллюстрирует:

```
class MyClass
  def self.new_method(name, &block)
    define_method(name, &block)
  end
end

a,b = 3,79

MyClass.new_method(:compute) { a*b }
x = MyClass.new
puts x.compute           # 237

a,b = 23,24
puts x.compute           # 552
```

Смысл здесь в том, что новый метод может обращаться к переменным в исходной области видимости блока, хотя сама эта область более не существует и никаким другим способом не доступна. Иногда это бывает полезно, особенно в случае

метапрограммирования или при разработке графических интерфейсов, когда нужно определить методы обратного вызова, реагирующие на события.

Отметим, что замыкание оказывается таковым только тогда, когда имя переменной то же самое. Изредка из-за этого могут возникать сложности. Ниже мы воспользовались методом `define_method`, чтобы предоставить доступ к переменной класса (вообще-то это следует делать не так, но для иллюстрации подойдет):

```
class SomeClass
  @@var = 999

  define_method(:peek) { @@var }
end
```

```
x = SomeClass.new
p x.peek          # 999
```

А теперь попробуем сделать с переменной экземпляра класса такой трюк:

```
class SomeClass
  @var = 999

  define_method(:peek) { @var }
end
```

```
x = SomeClass.new
p x.peek          # Печатается nil
```

Мы ожидали, что будет напечатано 999, а получили `nil`. Почему? Объясню чуть позже.

С другой стороны, такой код работает правильно:

```
class SomeClass
  @var = 999
  x = @var

  define_method(:peek) { x }
end
```

```
x = SomeClass.new
p x.peek          # 999
```

Так что же происходит? Да, замыкание действительно запоминает переменные в текущем контексте. Но ведь контекст нового метода — это контекст экземпляра объекта, а не самого класса.

Поскольку имя `@var` в этом контексте относится к переменной экземпляра объекта, а не класса, то переменная экземпляра класса оказывается скрыта переменной экземпляра объекта, хотя последняя никогда не использовалась и технически не существует.

В предыдущих версиях Ruby мы часто определяли методы во время выполнения с помощью `eval`. В принципе во всех таких случаях может и должен использоваться метод `define_method`. Некоторые тонкости вроде рассмотренной выше не должны вас останавливать.

11.3.6. Метод `const_missing`

Метод `const_missing` аналогичен методу `method_missing`. При попытке обратиться к неизвестной константе вызывается этот метод — если он, конечно, определен. В качестве параметра ему передается символ, ссылающийся на константу.

Чтобы перехватывать обращения к отсутствующим константам глобально, определите следующий метод в самом классе `Module` (это родитель класса `Class`).

```
class Module
  def const_missing(x)
    "Из Module"
  end
end

class X
end

p X::BAR      # "Из Module"
p BAR         # "Из Module"
p Array::BAR  # "Из Module"
```

Можно выполнить в нем любые действия: вернуть фиктивное значение константы, вычислить его и т.д. Помните класс `Roman` из главы 6? Воспользуемся им, чтобы трактовать любые последовательности римских цифр как числовые константы:

```
class Module
  def const_missing(name)
    Roman.decode(name)
  end
end

year1 = MCMLCCIV      # 1974
year2 = MMVIII        # 2008
```

Если такая глобальность вам не нужна, определите этот метод на уровне конкретного класса. Тогда он будет вызываться из этого класса и его потомков.

```
class Alpha
  def self.const_missing(sym)
    "В Alpha нет #{sym}"
  end
end

class Beta
  def self.const_missing(sym)
    "В Beta нет #{sym}."
  end
end

class A < Alpha
end
```

```
class B < Beta
end

p Alpha::FOO      # "B Alpha нет FOO"
p Beta::FOO       # "B Beta нет FOO"
p A::FOO          # "B Alpha нет FOO"
p B::FOO          # "B Beta нет FOO"
```

11.3.7. Удаление определений

Вследствие динамичности Ruby практически все, что можно определить, можно и уничтожить. Это может пригодиться, например, для того, чтобы «развязать» два куска кода в одной и той же области действия, избавляясь от переменных после того, как они были использованы. Другой повод – запретить вызовы некоторых потенциально опасных методов. Но по какой бы причине вы ни удаляли определение, делать это нужно крайне осторожно, чтобы не создать себе проблемы во время отладки.

Радикальный способ уничтожить определение – воспользоваться ключевым словом `undef` (неудивительно, что его действие противоположно действию `def`). Уничтожать можно определения методов, локальных переменных и констант на верхнем уровне. Хотя имя класса – тоже константа, удалить определение класса таким способом невозможно.

```
def asbestos
  puts "Теперь не огнеопасно"
end
```

```
tax = 0.08
```

```
PI = 3
```

```
asbestos
puts "PI=#{PI}, tax=#{tax}"
```

```
undef asbestos
undef tax
undef PI
```

Любое обращение к этим трем именам теперь приведет к ошибке.

Внутри определения класса можно уничтожать определения методов и констант в том же контексте, в котором они были определены. Нельзя применять `undef` внутри определения метода, а также к переменной экземпляра.

Существуют (определены в классе `Module`) также методы `remove_method` и `undef_method`. Разница между ними тонкая: `remove_method` удаляет текущее (или ближайшее) определение метода, а `undef_method` ко всему прочему удаляет его и из суперклассов, не оставляя от метода даже следа. Это различие иллюстрирует листинг 11.16.

Листинг 11.16. Методы `remove_method` и `undef_method`

```
class Parent

  def alpha
    puts "alpha: родитель"
  end

  def beta
    puts "beta: родитель"
  end
end

class Child < Parent

  def alpha
    puts "alpha: потомок"
  end

  def beta
    puts "beta: потомок"
  end

  remove_method :alpha    # Удалить "этот" alpha.
  undef_method :beta      # Удалить все beta.

end

x = Child.new

x.alpha      # alpha: родитель
x.beta      # Ошибка!
```

Метод `remove_const` удаляет константу.

```
module Math

  remove_const :PI

end

# PI больше нет!
```

Отметим, что таким способом можно удалить и определение класса (потому что идентификатор класса – это просто константа):

```
class BriefCandle
  #...
end

out_out = BriefCandle.new
class Object
```

```
remove_const :BriefCandle
end
```

```
# Создать еще один экземпляр класса BriefCandle не получится!
# (Хотя out_out все еще существует...)
```

Такие методы, как `remove_const` и `remove_method`, являются закрытыми (что и понятно). Поэтому во всех примерах они вызываются изнутри определения класса или модуля, а не снаружи.

11.3.8. Получение списка определенных сущностей

API отражения в Ruby позволяет опрашивать классы и объекты во время выполнения. Рассмотрим методы, имеющиеся для этой цели в `Module`, `Class` и `Object`.

В модуле `Module` есть метод `constants`, который возвращает массив всех констант, определенных в системе (включая имена классов и модулей). Метод `nesting` возвращает массив всех вложенных модулей, видимых в данной точке программы.

Метод экземпляра `Module#ancestors` возвращает массив всех предков указанного класса или модуля.

```
list = Array.ancestors
# [Array, Enumerable, Object, Kernel]
```

Метод `constants` возвращает список всех констант, доступных в данном модуле. Включаются также его предки.

```
list = Math.constants # ["E", "PI"]
```

Метод `class_variables` возвращает список всех переменных класса в данном классе и его суперклассах. Метод `included_modules` возвращает список модулей, включенных в класс.

```
class Parent
  @@var1 = nil
end
```

```
class Child < Parent
  @@var2 = nil
end
```

```
list1 = Parent.class_variables # ["@@var1"]
list2 = Array.included_modules # [Enumerable, Kernel]
```

Методы `instance_methods` и `public_instance_methods` класса `Class` – синонимы; они возвращают список открытых методов экземпляра, определенных в классе. Методы `private_instance_methods` и `protected_instance_methods` ведут себя аналогично. Любой из них принимает необязательный булевский параметр, по умолчанию равный `true`; если его значение равно `false`, то суперклассы не учитываются, так что список получается меньше.

```
n1 = Array.instance_methods.size # 121
n2 = Array.public_instance_methods.size # 121
n3 = Array.private_instance_methods.size # 71
```

```
n4 = Array.protected_instance_methods.size # 0
n5 = Array.public_instance_methods(false).size # 71
```

В классе Object есть аналогичные методы, применяющиеся к экземплярам (листинг 11.17). Метод `methods` возвращает список всех методов, которые можно вызывать для данного объекта. Метод `public_methods` возвращает список открытых методов и принимает параметр, равный по умолчанию `true`, который говорит, нужно ли включать также методы суперклассов. Методы `private_methods`, `protected_methods` и `singleton_methods` тоже принимают такой параметр.

Листинг 11.17. Отражение и переменные экземпляра

```
class SomeClass

  def initialize
    @a = 1
    @b = 2
  end

  def mymeth
    #...
  end

  protected :mymeth

end

x = SomeClass.new

def x.newmeth
  # ...
end

iv = x.instance_variables      # ["@b", "@a"]

p x.methods.size               # 42

p x.public_methods.size       # 41
p x.public_methods(false).size # 1

p x.private_methods.size      # 71
p x.private_methods(false).size # 1

p x.protected_methods.size    # 1
p x.singleton_methods.size    # 1
```

Если вы работаете с Ruby уже несколько лет, то заметите, что эти методы *немного* изменились. Теперь параметры по умолчанию равны `true`, а не `false`.

11.3.9. Просмотр стека вызовов

.....

And you may ask yourself:

*Well, how did I get here?**

Talking Heads, «Once in a Lifetime»

Иногда необходимо знать, кто вызвал метод. Эта информация полезна, если, например, произошло неисправимое исключение. Метод `caller`, определенный в модуле `kernel`, дает ответ на этот вопрос. Он возвращает массив строк, в котором первый элемент соответствует вызвавшему методу, следующий – методу, вызвавшему этот метод, и т.д.

```
def func1
  puts caller[0]
end
```

```
def func2
  func1
end
```

```
func2          # Печатается: somefile.rb:6:in 'func2'
```

Строка имеет формат «файл;строка» или «файл;строка в методе».

11.3.10. Мониторинг выполнения программы

Программа на Ruby может следить за собственным выполнением. У этой возможности есть много применений; интересующийся читатель может заглянуть в исходные тексты программ `debug.rb`, `profile.rb` и `tracer.rb`. С ее помощью можно даже создать библиотеку для «проектирования по контракту» (design-by-contract, DBC), хотя наиболее популярная в данный момент библиотека такого рода этим средством не пользуется.

Интересно, что этот фокус реализован целиком на Ruby. Мы пользуемся методом `set_trace_func`, который позволяет вызывать указанный блок при возникновении значимых событий в ходе исполнения программы. В справочном руководстве описывается последовательность вызова `set_trace_func`, поэтому здесь мы ограничимся простым примером:

```
def meth(n)
  sum = 0
  for i in 1..n
    sum += i
  end
  sum
end
```

** И задаешь себе вопрос:
Как же я оказался здесь?*

```

set_trace_func(proc do |event, file, line,
                      id, binding, klass, *rest|
  printf "%8s %s:%d %s/%s\n", event, file, line,
                      klass, id
end)

meth(2)

```

Отметим, что здесь соблюдается стандартное соглашение о заключении многострочного блока в операторные скобки `do-end`. Круглые скобки обязательны из-за особенностей синтаксического анализатора Ruby. Можно было бы, конечно, вместо этого поставить фигурные скобки.

Вот что будет напечатано в результате выполнения этого кода:

```

line prog.rb:13 false/
  call prog.rb:1  Object/meth
  line prog.rb:2  Object/meth
  line prog.rb:3  Object/meth
  c-call prog.rb:3 Range/each
  line prog.rb:4  Object/meth
  c-call prog.rb:4 Fixnum/+
c-return prog.rb:4 Fixnum/+
  line prog.rb:4  Object/meth
  c-call prog.rb:4 Fixnum/+
c-return prog.rb:4 Fixnum/+
c-return prog.rb:4 Range/each
  line prog.rb:6  Object/meth
  return prog.rb:6 Object/meth

```

С этим методом тесно связан метод `Kernel#trace_var`, который вызывает указанный блок при каждом присваивании значения глобальной переменной.

Предположим, что вам нужно извне протрассировать выполнение программы в целях отладки. Проще всего воспользоваться для этого библиотекой `tracer`. Пусть имеется следующая программа `prog.rb`:

```

def meth(n)
  (1..n).each {|i| puts i}
end

meth(3)

```

Можно запустить `tracer` из командной строки:

```

% ruby -r tracer prog.rb
#0:prog.rb:1::-:      def meth(n)
#0:prog.rb:1:Module:>:  def meth(n)
#0:prog.rb:1:Module:<:  def meth(n)
#0:prog.rb:8::-:      meth(2)
#0:prog.rb:1:Object:>:  def meth(n)
#0:prog.rb:2:Object:-:  sum = 0
#0:prog.rb:3:Object:-:  for i in 1..n
#0:prog.rb:3:Range:>:  for i in 1..n

```

```
#0:prog.rb:4:Object:-:      sum += i
#0:prog.rb:4:Fixnum:>:      sum += i
#0:prog.rb:4:Fixnum:<:      sum += i
#0:prog.rb:4:Object:-:      sum += i
#0:prog.rb:4:Fixnum:>:      sum += i
#0:prog.rb:4:Fixnum:<:      sum += i
#0:prog.rb:4:Range:<:       sum += i
#0:prog.rb:6:Object:-:      sum
#0:prog.rb:6:Object:<:      sum
```

Программа `tracer` выводит номер потока, имя файла и номер строки, имя класса, тип события и исполняемую строку исходного текста трассируемой программы. Бывают следующие типы событий: '-' — выполняется строка исходного текста, '>' — вызов, '<' — возврат, 'C' — класс, 'E' — конец. (Если вы автоматически включите эту библиотеку с помощью переменной окружения `RUBYOPT` или каким-то иным способом, то может быть напечатано много тысяч строк.)

11.3.11. Обход пространства объектов

Система исполнения Ruby должна отслеживать все известные объекты (хотя бы для того, чтобы убрать мусор, когда на объект больше нет ссылок). Информацию о них можно получить с помощью метода `ObjectSpace.each_object`.

```
ObjectSpace.each_object do |obj|
  printf "%20s: %s\n", obj.class, obj.inspect
end
```

Если задать класс или модуль в качестве параметра `each_object`, то будут возвращены лишь объекты указанного типа.

Модуль `ObjectSpace` полезен также для определения чистильщиков объектов (см. раздел 11.3.14).

11.3.12. Обработка вызовов несуществующих методов

Иногда бывают полезны классы, отвечающие на вызовы произвольных методов. Например, для того чтобы обернуть обращения к внешним программам в класс, который представляет каждое такое обращение как вызов метода. Заранее имена всех программ вы не знаете, поэтому написать определения всех методов при создании класса не получится. На помощь приходит метод `Object#method_missing`. Если объект Ruby получает сообщение для метода, который в нем не реализован, то вызывается метод `method_missing`. Этим можно воспользоваться для превращения ошибки в обычный вызов метода. Реализуем класс, обертывающий команды операционной системы:

```
class CommandWrapper

  def method_missing(method, *args)
    system(method.to_s, *args)
  end

end
```



```

cw = CommandWrapper.new
cw.date           # Sat Apr 28 22:50:11 CDT 2001
cw.dup '-s', '/tmp' # 166749 /tmp

```

Первый параметр метода `method_missing` – имя вызванного метода (которое не удалось найти). Остальные параметры – все то, что было передано при вызове этого метода.

Если написанная вами реализация `method_missing` не хочет обрабатывать конкретный вызов, она должна вызвать `super`, а не возбуждать исключение. Тогда методы `method_missing` в суперклассах получат возможность разобраться с ситуацией. В конечном счете будет вызван `method_missing`, определенный в классе `Object`, который и возбудит исключение.

11.3.13. Отслеживание изменений в определении класса или объекта

А зачем, собственно? Кому интересны изменения, которым подвергался класс?

Одна возможная причина – желание следить за состоянием выполняемой программы на Ruby. Быть может, мы реализуем графический отладчик, который должен обновлять список методов, добавляемых «на лету».

Другая причина: мы хотим вносить соответствующие изменения в другие классы. Например, мы разрабатываем модуль, который можно включить в определение любого класса. С момента включения будут трассироваться любые обращения к методам этого класса. Что-то в этом роде:

```

class MyClass
  include Tracing

  def one
  end

  def two(x, y)
  end
end

m = MyClass.new
m.one           # Вызван метод one. Параметры =
m.two(1, 'cat') # Вызван метод two. Параметры = 1, cat

Он должен работать также для всех подклассов трассируемого класса:
class Fred < MyClass

  def meth(*a)
  end

end

Fred.new.meth(2,3,4,5) # вызван метод meth. Параметры = 2, 3, 4, 5

```

Возможная реализация такого модуля показана в листинге 11.18.

Листинг 11.18. Трассирующий модуль

```
module Tracing
  def Tracing.included(into)
    into.instance_methods(false).each { |m|
Tracing.hook_method(into, m) }
    def into.method_added(meth)
      unless @adding
        @adding = true
        Tracing.hook_method(self, meth)
        @adding = false
      end
    end
  end

  def Tracing.hook_method(klass, meth)
    klass.class_eval do
      alias_method "old_#{meth}", "#{meth}"
      define_method(meth) do |*args|
        puts "Вызван метод #{meth}. Параметры = #{args.join(', ')}"
        self.send("old_#{meth}", *args)
      end
    end
  end

  class MyClass
    include Tracing

    def first_meth
    end

    def second_meth(x, y)
    end
  end

  m = MyClass.new
  m.first_meth           # Вызван метод first_meth. Параметры =
  m.second_meth(1, 'cat') # Вызван метод second_meth. Параметры = 1, cat
```

В этом коде два основных метода. Первый, `included`, вызывается при каждой вставке модуля в класс. Наша версия делает две вещи: вызывает метод `hook_method` для каждого метода, уже определенного в целевом классе, и вставляет определенные метода `method_added` в этот класс. В результате любой добавленный позже метод тоже будет обнаружен и для него вызван `hook_method`.

Сам метод `hook_method` работает прямолинейно. При добавлении метода ему сразу назначается синоним `old_name`. Исходный метод заменяется кодом трассировки, который выводит имя и параметры метода, а затем вызывает метод, к которому было обращение.

Обратите внимание на использование конструкции `alias_method`. Работает она почти так же, как `alias`, но только для методов (да и сама является методом, а не ключевым словом). Можно было бы записать эту строку иначе:

```
# Еще два способа записать эту строку...
```

```
# Символы с интерполяцией:
```

```
alias_method :old_#{meth}", :_#{meth}"
```

```
# Преобразование строк с помощью to_sym:
```

```
alias_method "old_#{meth}".to_sym, meth.to_sym
```

Чтобы обнаружить добавление нового метода класса в класс или модуль, можно определить метод класса `singleton_method_added` внутри данного класса. (Напомним, что синглтонный метод в этом смысле – то, что мы обычно называем методом класса, поскольку `Class` – это объект.) Этот метод определен в модуле `Kernel` и по умолчанию ничего не делает, но мы можем переопределить его, как сочтем нужным.

```
class MyClass
```

```
  def MyClass.singleton_method_added(sym)
    puts "Добавлен метод #{sym.to_s} в класс MyClass."
  end
```

```
  def MyClass.meth1
    puts "Я meth1."
  end
```

```
end
```

```
def MyClass.meth2
  puts "А я meth2."
end
```

В результате выводится следующая информация:

```
Добавлен метод singleton_method_added в класс MyClass.
```

```
Добавлен метод meth1 в класс MyClass.
```

```
Добавлен метод meth2 в класс MyClass.
```

Отметим, что фактически добавлено три метода. Возможно, это противоречит вашим ожиданиям, но метод `singleton_method_added` может отследить и добавление самого себя.

Метод `inherited` (из `Class`) используется примерно так же. Он вызывается в момент создания подкласса.

```
class MyClass
```

```
  def MyClass.inherited(subclass)
    puts "#{subclass} наследует MyClass."
  end
```

```
  # ...
```

```
end

class OtherClass < MyClass
  # ...

end

# Выводится: OtherClass наследует MyClass.
```

Можно также следить за добавлением методов экземпляра модуля к объекту (с помощью метода `extend`). При каждом выполнении `extend` вызывается метод `extend_object`.

```
module MyMod

  def MyMod.extend_object(obj)
    puts "Расширяется объект id #{obj.object_id}, класс #{obj.class}"
    super
  end

  # ...

end

x = [1, 2, 3]
x.extend(MyMod)

# Выводится:
# Расширяется объект id 36491192, класс Array
```

Обращение к `super` необходимо для того, чтобы мог отработать исходный метод `extend_object`. Это напоминает поведение метода `append_features` (см. раздел 11.1.12); данный метод годится также для отслеживания использования модулей.

11.3.14. Определение чистильщиков для объектов

У классов в Ruby есть конструкторы (методы `new` и `initialize`), но нет деструкторов (методов, которые уничтожают объекты). Объясняется это тем, что в Ruby применяется алгоритм пометки и удаления объектов, на которые не осталось ссылок (сборка мусора); вот почему деструктор просто не имеет смысла.

Однако тем, кто переходит на Ruby с таких языков, как C++, этот механизм представляется необходимым – часто задается вопрос, как написать код очистки уничтожаемых объектов. Простой ответ звучит так: невозможно сделать это надежно. Но можно написать код, который будет вызываться, когда сборщик мусора уничтожает объект.

```
a = "hello"
puts "Для строки 'hello' ИД объекта равен #{a.id}."
ObjectSpace.define_finalizer(a) { |id| puts "Уничтожается #{id}." }
puts "Нечего убирать."
```

```
GC.start
a = nil
puts "Исходная строка - кандидат на роль мусора."
GC.start
```

Этот код выводит следующее:

```
Для строки 'hello' ИД объекта равен 537684890.
Нечего убирать.
Исходная строка - кандидат на роль мусора.
Уничтожается 537684890.
```

Подчеркнем, что к моменту вызова чистильщика объект уже фактически уничтожен. Попытка преобразовать идентификатор в ссылку на объект с помощью метода `ObjectSpace._id2ref` приведет к исключению `RangeError` с сообщением о том, что вы пытаетесь воспользоваться уничтоженным объектом.

Имейте в виду, что в Ruby применяется консервативный вариант сборки мусора по алгоритму пометки и удаления. Нет гарантии, что любой объект будет убран до завершения программы.

Однако все это может оказаться и ненужным. В Ruby существует стиль программирования, в котором для инкапсуляции работы с ресурсами служат блоки. В конце блока ресурс освобождается, и жизнь продолжается без помощи чистильщиков. Рассмотрим, например, блочную форму метода `File.open`:

```
File.open("myfile.txt") do |file|
  line1 = file.read
  # ...
end
```

Здесь в блок передается объект `File`, а по выходе из блока файл закрывается, причем все это делается под контролем метода `open`. Функциональное подмножество метода `File.open` на чистом Ruby (сейчас этот метод ради эффективности написан на C) могло бы выглядеть так:

```
def File.open(name, mode = "r")
  f = os_file_open(name, mode)
  if block_given?
    begin
      yield f
    ensure
      f.close
    end
    return nil
  else
    return f
  end
end
```

Мы проверяем наличие блока. Если блок был передан, то мы вызываем его, передавая открытый файл. Делается это в контексте блока `begin-end`, который гарантирует, что файл будет закрыт по выходе из блока, даже если произойдет исключение.

11.4. Заключение

В этой главе были приведены примеры использования более сложных и даже экзотических механизмов ООП, а также решения некоторых рутинных задач. Мы видели, как реализуются некоторые паттерны проектирования. Познакомились мы и с API отражения в Ruby, продемонстрировали ряд интересных следствий динамической природы Ruby и некоторые трюки, возможные в динамическом языке.

Пришло время вернуться в реальный мир. Ведь ООП – не самоцель, а всего лишь средство достижения цели. Последняя же заключается в написании эффективных, безошибочных и удобных для сопровождения приложений.

В современном окружении таким приложениям часто необходим графический интерфейс. В главе 12 мы рассмотрим создание графических интерфейсов на языке Ruby.

Глава 12. Графические интерфейсы для Ruby

.....
Нет ничего хуже четкого образа нечеткой идеи.
Ансель Адамс

Нет смысла отрицать, что мы живем в век графических интерфейсов (ГИ). В обозримом будущем тот или иной вид графического интерфейса станет основным способом взаимодействия с компьютерами.

Я не думаю, что командная строка не переживет следующее десятилетие – у нее есть свое место в мире. Но даже закоренелые хакеры прежних лет (предпочитающие команду `ср -R` перетаскиванию файлов мышкой) все-таки не прочь воспользоваться ГИ, когда это оправданно.

Однако у графического программирования есть свои сложности. Главная проблема, конечно, состоит в том, чтобы определить, как должен выглядеть удобный интерфейс к программе; при проектировании интерфейсов картинка не всегда заменяет тысячу слов. В этой книге мы не можем уделить внимание данному аспекту, все-таки наша тема – не эргономика, не эстетика и не психология.

Вторая очевидная проблема в том, что графическое программирование сложнее. Надо учитывать форму, размер, положение и поведение всех находящихся на экране элементов управления, которыми можно манипулировать с помощью мыши и клавиатуры.

Третья трудность заключается в следующем: на различных платформах представления о том, что такое оконная система и как она должна быть реализована, существенно разнятся. Чтобы в полной мере оценить это расхождение, нужно иметь опыт работы в разных системах. Многие программисты пытались создать кросс-платформенные инструменты и обнаружили, что именно с нестыковкой графических интерфейсов справиться труднее всего.

Эта глава не поможет вам в разрешении вышеназванных проблем. Максимум, что я могу сделать, – предложить очень краткое введение в несколько популярных графических систем (в той мере, в какой они относятся к Ruby), а также несколько советов и наблюдений.

Большая часть главы посвящена библиотекам Tk, GTK+, FOX и Qt. Велики шансы на то, что у вас возникнет вопрос: «А почему здесь нет (подставьте имя своей любимой библиотеки)?»

Причин несколько. Прежде всего это ограниченность места: все же книга посвящена не только графическим интерфейсам. Другая причина заключается в том,

что для вашей системы могут быть еще не написаны привязки к Ruby (и в таком случае мы призываем вас этим заняться). И наконец, не все графические интерфейсы «равны». Мы попытались рассказать о самых важных и зрелых, а остальные только упомянули.

12.1. Ruby/Tk

Своими корнями Tk уходит в далекий 1988 год, если считать и предварительные версии. Долгое время эта система считалась спутником языка Tcl, но вот уже несколько лет как она используется и с другими языками, в том числе Perl и Ruby.

Если бы у Ruby был «родной» графический интерфейс, наверное, им стал бы Tk. В настоящее время он все еще широко распространен, а в некоторые дистрибутивы Ruby входит в более или менее готовом виде.

Я упомянул о Perl не зря. Привязки Tk к Ruby и Perl похожи настолько, что любая информация о Perl/Tk применима и к Ruby/Tk. В этой связи стоит упомянуть книгу Нэнси Уолш (Nancy Walsh) «Learning Perl/Tk».

12.1.1. Обзор

В 2001 году Tk был, наверное, самым популярным графическим интерфейсом для Ruby. Он был первым и долгое время входил в состав стандартного дистрибутива Ruby. Сейчас он, пожалуй, не так распространен, но все еще широко применяется.

Кто-то скажет, что Tk уже устарел. Те, кому нравятся объектно-ориентированные интерфейсы, будут им немного разочарованы. Но есть и достоинства: широкая известность, переносимость и стабильность.

Любое приложение Ruby/Tk должно загрузить расширение `tk`, выполнив директиву `load tk`. Далее интерфейс приложения строится поэтапно, начиная с того или иного контейнера, в который помещаются отдельные элементы управления. В конце выполняется вызов метода `Tk.mainloop`, в котором обрабатываются события: перемещения мыши, нажатия кнопок и т. д.

```
require "tk"  
# Подготовка интерфейса приложения...  
Tk.mainloop
```

Как и в большинстве оконных систем, графические элементы управления в Tk называются *виджетами*. Виджеты группируются с помощью контейнеров. Контейнер верхнего уровня называется *корневым*. Явно задавать корневой контейнер не всегда обязательно, но лучше это сделать.

Классы виджетов именуются так, как принято в мире Tk (в начале идет префикс `tk`). Например, виджету `Frame` соответствует класс `TkFrame`.

Понятно, что виджеты создаются методом `new`. Первый параметр определяет контейнер, в который помещается виджет; если он опущен, подразумевается корневой контейнер.

Дополнительные параметры виджета можно задавать двумя способами. Первый (в духе Perl) – передать хэш, содержащий названия и значения атрибутов. (Напомним, что в Ruby при передаче хэша последним или единственным параметром фигурные скобки можно опускать).


```
my_widget = TkSomewidget.new( "borderwidth" => 2, "height" => 40 ,
                               "justify" => "center" )
```

Другой способ – передать конструктору блок, который будет вычислен методом `instance_eval`. Внутри блока можно вызывать методы для установки атрибутов виджета (такие методы называются так же, как сами атрибуты). Имейте в виду, что блок вычисляется в контексте вызываемого объекта, а не вызывающей программы. Это означает, например, что к переменным экземпляра вызывающего объекта в блоке обращаться нельзя.

```
my_widget = TkSomewidget.new do
  borderwidth 2
  height 40
  justify "center"
end
```

Tk предоставляет три геометрических менеджера для управления относительным размером и расположением виджетов на экране. Наиболее распространенный из них – `pack`, остальные два – `grid` и `place`. Менеджер `grid` обладает богатыми возможностями, но не свободен от ошибок; `place` – самый простой из трех, он требует, чтобы были заданы абсолютные координаты всех расположенных внутри него виджетов. В примерах ниже мы будем пользоваться только менеджером `pack`.

12.1.2. Простое оконное приложение

Продemonстрируем очень простое приложение – окно, в котором выводится текущая дата. Начнем с явного создания корневого контейнера `root` и поместим в него виджет `Label`.

```
require "tk"

root = TkRoot.new() { title "Today's Date" }
str = Time.now.strftime("Today is \n%B %d, %Y")
lab = TkLabel.new(root) do
  text str
  pack("padx" => 15, "pady" => 10,
       "side" => "top")
end
Tk.mainloop
```

Здесь мы создали корневой контейнер, сформировали строку даты и создали метку. В качестве текста, изображаемого на метке, мы задали строку `str`, а чтобы все выглядело аккуратно, вызвали метод `pack`, которому сказали, что отступ по горизонтали должен составлять 15 пикселей, а по вертикали – 10 пикселей. Текст мы попросили отцентрировать в границах метки.

На рис. 12.1 показано, как выглядит окно приложения.

Как было сказано выше, создать метку можно было бы и так:

```
lab = TkLabel.new(root) do
  text str
  pack("padx" => 15, "pady" => 10,
       "side" => "top")
end
```



Рис. 12.1. Простое приложение Tk

Экранные единицы измерения (в примере выше мы их использовали для указания `padx` и `pady`) – по умолчанию пиксели. Можно применять и другие единицы, если добавить к числу суффикс. Тогда значение становится строковым, но поскольку Ruby/Tk на это наплевать, то и мы не станем беспокоиться. Допустимы следующие единицы измерения: сантиметры (с), миллиметры (м), дюймы (i) и пункты (p). Все показанные ниже способы указания `padx` правильны:

```
pack("padx" => "80m")
pack("padx" => "8с")
pack("padx" => "3i")
pack("padx" => "12p")
```

Атрибут `side` в данном случае не дает ничего, поскольку мы установили его значение по умолчанию. Если вы измените размер окна, то текст останется «приклеенным» к верхней части той области, которой принадлежит. Другие возможные значения `side`: `right`, `left` и `bottom`.

У метода `pack` есть и другие атрибуты, управляющие размещением виджетов на экране. Мы рассмотрим не все.

Атрибут `fill` указывает, должен ли виджет заполнять весь выделенный для него прямоугольник (по горизонтали и/или по вертикали). Допустимые значения: `x`, `y`, `both` и `none` (по умолчанию `none`).

Атрибут `anchor` «скрепляет» виджет с теми или иными сторонами его прямоугольника; при этом применяется нотация «сторон света». По умолчанию подразумевается значение `center`, другие допустимые значения: `n`, `s`, `e`, `w`, `ne`, `nw`, `se`, `sw`.

Атрибут `in` упаковывает виджет относительно контейнера, отличного от его родительского. По умолчанию, конечно, принимается родительский контейнер.

Атрибуты `before` и `after` позволяют произвольно задавать порядок упаковки виджетов. Это полезно, поскольку виджеты могут создаваться не в том порядке, в котором расположены на экране.

В общем, Tk обеспечивает достаточную гибкость при размещении виджетов в окне. Читайте документацию и экспериментируйте.

12.1.3. Кнопки

В любом графическом интерфейсе кнопка – один из наиболее употребительных виджетов. Как и следовало ожидать, в Ruby/Tk кнопка представляется классом `TkButton`.

В нетривиальных приложениях обычно создаются фреймы, содержащие разные виджеты. Кнопка может располагаться внутри такого фрейма.

Обычно для кнопки задаются по меньшей мере три атрибута:

- текст кнопки;
- ассоциированная с кнопкой команда (исполняемая в результате нажатия);
- способ упаковки кнопки в объемлющем контейнере.

Вот простенький пример:

```
btn_OK = TkButton.new do
  text "OK"
  command { puts "Пользователь говорит ОК." }
  pack("side" => "left")
end
```

Здесь мы создаем новую кнопку и присваиваем объект переменной `btn_OK`. Конструктору передается блок, хотя при желании можно было бы воспользоваться и хэшем. В данном случае мы записали блок на нескольких строчках (нам так больше нравится), хотя на практике в одну строку можно «напихать» сколько угодно кода. Напомним, кстати, что блок вычисляется методом `instance_eval`, то есть в контексте объекта (в данном случае – вновь созданного объекта `TkButton`).

Текст, заданный в качестве значения атрибута `text`, рисуется на кнопке. Он может состоять из нескольких слов и даже строк.

Как работает метод `pack`, мы уже видели, ничего нового здесь нет. Стоит лишь отметить, что без вызова `pack` виджет не будет виден.

Интересная часть – метод `command`, который принимает объект `Proc` и ассоциирует его с кнопкой. Часто для этой цели – и в данном случае тоже – применяется метод `lambda_proc` из модуля `Kernel`, который преобразует блок в объект `Proc`.

Выполняемое действие не очень осмысленно. Когда пользователь нажимает кнопку, вызывается неграфический метод `puts`, выводящий строку в окно команд, из которого была запущена программа, или, быть может, в окно дополнительной консоли.

Следующий пример более содержателен. В листинге 12.1 приведено приложение, имитирующее термостат. В нем отображается то увеличивающаяся, то уменьшающаяся температура (создавая иллюзию, будто мы включаем обогрев или охлаждение). Код сопровождается комментариями.

Листинг 12.1. Имитация термостата

```
require 'tk'

# Типичные параметры упаковки...
Top = { 'side' => 'top', 'padx'=>5, 'pady'=>5 }
Left = { 'side' => 'left', 'padx'=>5, 'pady'=>5 }
Bottom = { 'side' => 'bottom', 'padx'=>5, 'pady'=>5 }

temp = 74 # Начальная температура...

root = TkRoot.new { title "Thermostat" }

top = TkFrame.new(root) { background "#606060" }
bottom = TkFrame.new(root)

tlabel = TkLabel.new(top) do
  text temp.to_s
  font "Arial" 54 {bold}"
```

```

foreground "green"
background "#606060"
pack Left
end

TkLabel.new(top) do                # Символ градуса
  text "o"
  font "{Arial} 14 {bold}"
  foreground "green"
  background "#606060"
  # Включить в хэш прикрепление к северу (символ градуса отображается
  # в виде верхнего индекса).
  pack Left.update({ 'anchor' => 'n' })
end

TkButton.new(bottom) do
  text " Up "
  command proc { tlab.configure("text"=>(temp+=1).to_s) }
  pack Left
end

TkButton.new(bottom) do
  text "Down"
  command proc { tlab.configure("text"=>(temp-=1).to_s) }
  pack Left
end

top.pack Top
bottom.pack Bottom

Tk.mainloop

```

Здесь мы создали два фрейма. Верхний служит только для отображения температуры. Она измеряется по шкале Фаренгейта и для улучшения дизайна выводится крупным шрифтом (а символ градуса отображается маленькой буквой «o», расположенной справа сверху). Нижний фрейм содержит кнопки «вверх» и «вниз».

Обратите внимание на не встречавшиеся еще атрибуты объекта TkLabel. Метод `font` задает гарнитуру и размер шрифта, которым выводится текст метки. Строковое значение платформенно-зависимо; то, что приведено в примере, предназначено для ОС Windows. В системах UNIX обычно указывается длинное и малопонятное имя шрифта, принятое в X Window, например: `-Adobe-Helvetica-Bold-R-Normal*-120-*-*-*-*-*`.

Метод `foreground` задает цвет текста. Здесь мы передаем строку "green" (которая в Tk имеет предопределенный смысл). Если вы хотите знать, предопределены ли тот иной цвет в Tk, то самое простое – попробовать.

Аналогично метод `background` задает цвет фона, на котором выводится текст. В данном случае мы передаем строку в другом формате, а именно указываем красную, зеленую и синюю компоненты в шестнадцатеричном виде, как принято в языке HTML и других случаях. (Строка "#606060" соответствует приятному серому цвету.)

Мы не включили никакой кнопки «выхода» (чтобы не загромождать окно). Как обычно, для того чтобы закрыть приложение, достаточно щелкнуть по иконке Close в правом верхнем углу рамки окна.

Отметим использование метода `configure` в описании команд для кнопок; он изменяет текст метки по мере того, как текущая температура уменьшается или увеличивается. Мы уже говорили, что таким образом почти все атрибуты можно изменять во время выполнения, причем изменение отображается на экране незамедлительно.

Упомянем еще две операции над текстовыми кнопками. Метод `justify` принимает один параметр ("`left`", "`right`" или "`center`"), определяющий выравнивание текста внутри кнопки (по умолчанию подразумевается "`center`"). Мы говорили, что можно отображать многострочный текст; метод `wrplength` задает номер колонки, в которой происходит перенос слова.

Стиль кнопки можно изменить методом `relief`, придав ей трехмерный вид. В качестве параметра этому методу можно передать одну из строк: "`flat`", "`groove`", "`raised`", "`ridge`" (по умолчанию), "`sunken`" или "`solid`". Методы `width` и `height` явно задают размеры кнопки. Имеется также метод `borderwidth` и аналогичные. О других атрибутах (которых немало) вы можете прочесть в руководстве.

Рассмотрим еще один пример использования кнопки. На этой кнопке будет изображение, а не просто текст.

Я создал GIF-файлы с изображениями стрелок, указывающих вверх и вниз (`up.gif` и `down.gif`). Для получения ссылок на них можно воспользоваться классом `TkPhotoImage`, а затем передать эти ссылки в качестве параметров при создании кнопок.

```
up_img = TkPhotoImage.new("file"=>"up.gif")
down_img = TkPhotoImage.new("file"=>"down.gif")

TkButton.new(bottom) do
  image up_img
  command proc { tlab.configure("text"=>(temp+=1).to_s) }
  pack Left
end

TkButton.new(bottom) do
  image down_img
  command proc { tlab.configure("text"=>(temp-=1).to_s) }
  pack Left
end
```

Здесь просто заменены некоторые строки в первом примере. Если не считать внешнего вида кнопок, то поведение не изменилось. На рис. 12.2 показано окно приложения.

12.1.4. Текстовые поля

Чтобы отобразить поле для ввода текста и манипулировать им, применяется виджет `TkEntry`. Как и следовало ожидать, для указания размера, цвета и поведения



Рис. 12.2. Имитация термостата
(с графическими кнопками)

предусмотрены многочисленные атрибуты. Мы приведем довольно объемный пример, иллюстрирующий применение некоторых из них.

Поле ввода полезно лишь, если существует способ получить введенное в него значение. Обычно поле связывается с переменной (если быть точным, с объектом `TkVariable`), хотя можно воспользоваться и методом `get`.

Предположим, что мы разрабатываем telnet-клиент, который принимает четыре параметра: адрес хоста, номер порта (по умолчанию 23), имя пользователя и его пароль. Для красоты добавим еще две кнопки для операций «войти» и «отменить».

В представленном фрагменте используются фреймы, чтобы форма выглядела аккуратнее. Правда, написанный код не переносим, и настоящий знаток Tk с презрением отверг бы его. Но просто для сведения мы все-таки документировали этот «небрежный» подход к организации информации на экране.



Рис. 12.3. Имитация telnet-клиента

Вид окна показан на рис. 12.3, а код – в листинге 12.2.

Листинг 12.2. Имитация telnet-клиента

```
require "tk"

def packing(padx, pady, side=:left, anchor=:n)
  { "padx" => padx, "pady" => pady,
    "side" => side.to_s, "anchor" => anchor.to_s }
end

root = TkRoot.new() { title "Telnet session" }
```

```
top = TkFrame.new(root)
fr1  = TkFrame.new(top)
fr1a = TkFrame.new(fr1)
fr1b = TkFrame.new(fr1)
fr2  = TkFrame.new(top)
fr3  = TkFrame.new(top)
fr4  = TkFrame.new(top)

LabelPack = packing(5, 5, :top, :w)
EntryPack = packing(5, 2, :top)
ButtonPack = packing(15, 5, :left, :center)
FramePack = packing(2, 2, :top)
Frame1Pack = packing(2, 2, :left)

var_host = TkVariable.new
var_port = TkVariable.new
var_user = TkVariable.new
var_pass = TkVariable.new

lab_host = TkLabel.new(fr1a) do
  text "Host name"
  pack LabelPack
end
ent_host = TkEntry.new(fr1a) do
  textvariable var_host
  font "{Arial} 10"
  pack EntryPack
end

lab_port = TkLabel.new(fr1b) do
  text "Port"
  pack LabelPack
end

ent_port = TkEntry.new(fr1b) do
  width 4
  textvariable var_port
  font "{Arial} 10"
  pack EntryPack
end

lab_user = TkLabel.new(fr2) do
  text "User name"
  pack LabelPack
end

ent_user = TkEntry.new(fr2) do
  width 21
  font "{Arial} 12"
```

```
    textvariable var_user
    pack EntryPack
end

lab_pass = TkLabel.new(fr3) do
  text "Password"
  pack LabelPack
end

ent_pass = TkEntry.new(fr3) do
  width 21
  show "*"
  textvariable var_pass
  font "{Arial} 12"
  pack EntryPack
end

btn_signon = TkButton.new(fr4) do
  text "Sign on"
  command proc {} # Ничего не делает!
  pack ButtonPack
end

btn_cancel = TkButton.new(fr4) do
  text "Cancel"
  command proc { exit } # Просто выход.
  pack ButtonPack
end

top.pack FramePack
fr1.pack FramePack
fr2.pack FramePack
fr3.pack FramePack
fr4.pack FramePack
fr1a.pack Frame1Pack
fr1b.pack Frame1Pack

var_host.value = "addison-wesley.com"
var_user.value = "debra"
var_port.value = 23

ent_pass.focus
foo = ent_user.font

Tk.mainloop
```

Прежде всего разберемся с размещением виджетов. Сначала мы создали несколько фреймов, расположенных друг под другом. В самом верхнем фрейме есть два фрейма поменьше, расположенных по горизонтали.

В листинге 12.2 встречается также метод `packing`, единственная цель которого – сделать код чуточку чище. Он возвращает хэш, содержащий значения атрибутов `padx`, `pady`, `side` и `anchor`.

Объекты `TkVariable` предназначены для ассоциирования полей ввода с переменными. В классе `TkVariable` определен метод доступа `value`, который позволяет получать и устанавливать значение, хранящееся в объекте.

При создании объекта `TkEntry`, например `ent_host`, задаем атрибут `textvariable`, который связывает его с соответствующим объектом `TkVariable`. Иногда мы явно указываем ширину поля методом `width`; если это не сделано, то будет автоматически выбрана разумная ширина, обычно определяемая значением, которое в данный момент хранится в поле. Часто ширину подбирают методом проб и ошибок.

Шрифты задаются для полей ввода так же, как для меток. Аналогично обстоит дело и с цветами, которые в этом примере не задаются. Если шрифт пропорциональный, то два поля одинаковой ширины на экране могут оказаться различными.

Как обычно, необходимо вызвать метод `pack`. Мы немного упростили вызовы за счет использования констант.

Для поля, содержащего пароль, вызывается метод `show`, поскольку вводимое в него значение не должен видеть человек, заглядывающий через плечо. Вместо набираемых пользователем символов в таком поле будет отображаться символ, переданный методу `show` в качестве параметра (в данном случае звездочка).

Я уже сказал, что кнопки тут нужны только для красоты. Кнопка `Sign on` вообще ничего не делает, а `Cancel` завершает программу.

У полей ввода есть и другие атрибуты. Можно изменять значение из программы, не заставляя пользователя вводить его; можно задать шрифт и цвета; можно изменить характеристики курсора, указывающего место вставки, и перемещать его в нужное место. И многое, многое другое. Полное описание вы найдете в руководстве.

Раз уж мы заговорили о вводе текста, будет уместно упомянуть виджет `Text`. По сравнению с полем ввода это примерно то же, что двухместный планер по сравнению с космическим челноком. Виджет `Text` спроектирован специально для ввода больших фрагментов текста, насчитывающих много строк, и по существу является основой для создания полноценного редактора. Он довольно сложен, поэтому мы не будем его здесь рассматривать.

12.1.5. Прочие виджеты

`Tk` содержит еще много виджетов. Упомянем некоторые из них.

Флажок обычно используется для представления полей, которые могут принимать одно из двух значений: да/нет или вкл/выкл. В `Tk` он называется «`check button`», а соответствующий ему класс – `TkCheckButton`.

Пример в листинге 12.3 – это лишь скелет, в нем даже ни одной кнопки нет. Выводятся три флажка, соответствующие курсам, на которые можно записаться (информатика, музыка и литература). На консоль подается сообщение при каждом изменении состояния флажка.

Листинг 12.3. Флажки в Tk

```
require 'tk'

root = TkRoot.new { title "Checkbox demo" }
top = TkFrame.new(root)

PackOpts = { "side" => "top", "anchor" => "w" }

cb1var = TkVariable.new
cb2var = TkVariable.new
cb3var = TkVariable.new

cb1 = TkCheckButton.new(top) do
  variable cb1var
  text "Информатика"
  command { puts "Button 1 = #{cb1var.value}" }
  pack PackOpts
end

cb2 = TkCheckButton.new(top) do
  variable cb2var
  text "Музыка"
  command { puts "Button 2 = #{cb2var.value}" }
  pack PackOpts
end

cb3 = TkCheckButton.new(top) do
  variable cb3var
  text "Литература"
  command { puts "Button 3 = #{cb3var.value}" }
  pack PackOpts
end

top.pack PackOpts

Tk.mainloop
```

Отметим, что переменная, ассоциированная с флажком, принимает значение 1, когда флажок отмечен, и 0 – когда он сброшен. Эти значения можно изменить с помощью методов `onvalue` и `offvalue`. Кроме того, еще до создания флажка можно установить значение переменной и тем самым задать начальное состояние флажка.

Если по какой-то причине мы хотим закрасить флажок серым, то можем с помощью метода `state` установить состояние `disabled`. Остальные состояния – `active` (отмечен) и `normal` (сброшен), причем последнее принято по умолчанию.

Изменим пример в листинге 12.3. Пусть речь идет не о дополнительных, а о профилирующих университетских курсах. Если не считать двойных курсов, то в каждый момент времени можно выбрать не более одного предмета. В таком случае нам понадобятся переключатели (их реализует класс `TkRadioButton`).

Пример в листинге 12.4 мало чем отличается от листинга 12.3. Конечно, имя класса теперь другое. Еще одно важное отличие состоит в том, что всем переключателям соответствует одна и та же переменная. Именно поэтому Tk знает, что переключатели принадлежат одной группе. В форме может быть и несколько групп переключателей, но для каждой группы выделяется по одной переменной.

Листинг 12.4. Переключатели в Tk

```
require 'tk'

root = TkRoot.new() { title "Radiobutton demo" }
top = TkFrame.new(root)

PackOpts = { "side" => "top", "anchor" => "w" }

major = TkVariable.new

b1 = TkRadioButton.new(top) do
  variable major
  text "Информатика"
  value 1
  command { puts "Major = #{major.value}" }
  pack PackOpts
end

b2 = TkRadioButton.new(top) do
  variable major
  text "Музыка"
  value 2
  command { puts "Major = #{major.value}" }
  pack PackOpts
end

b3 = TkRadioButton.new(top) do
  variable major
  text "Литература"
  value 3
  command { puts "Major = #{major.value}" }
  pack PackOpts
end

top.pack PackOpts

Tk.mainloop
```

Здесь метод `value` ассоциирует с каждым переключателем конкретное значение. Значения могут быть произвольны (например, строки). Мы не использовали строки, так как хотели подчеркнуть, что не существует прямой связи между текстом виджета и возвращаемым им значением.

Для настройки внешнего вида и поведения флажков и переключателей есть множество атрибутов. К примеру, метод `image` позволяет выводить не текстовую строку, а изображение. Применимы также обычные атрибуты форматирования и отображения виджетов; подробности вы найдете в руководстве.

Если бы данная книга (или хотя бы эта глава) была целиком посвящена Tk, мы бы рассказали много чего еще. Но невозможно уделить внимание всем деталям – упомянем их лишь для того, чтобы вы знали, что имеется в вашем распоряжении.

Виджет `Listbox` (`TkListbox`) позволяет вывести список, из которого пользователь выбирает элементы. Режим выбора (метод `selectmode`) принимает следующие значения: `single`, `extended`, `browse`. Первые два режима определяют, можно ли выбрать только один или сразу несколько элементов. Режим `browse` аналогичен режиму `single` с тем отличием, что выбранный элемент можно перемещать в списке мышью. Список можно прокручивать, так что число элементов в нем не ограничено.

Tk располагает развитыми средствами для работы с меню: выпадающие меню, уединенные (`tear-off`) меню, каскадные подменю, клавиши быстрого выбора, переключатели в меню и многое другое. Ознакомьтесь с классами `TkMenu`, `TkMenubar` и `TkMenuItem`.

Пожалуй, самый «творческий» виджет – это `TkCanvas`, который позволяет программисту манипулировать изображением на уровне пикселей. У него есть методы для рисования линий и фигур, манипулирования цветами и загрузки изображений в различных форматах. Если вашему приложению необходима развитая графика или рисование под контролем пользователя, то этот виджет вас заинтересует.

Полоса прокрутки позволяет реализовать нестандартную прокрутку по горизонтали и по вертикали (например, синхронную прокрутку двух окон). Виджет `Scale` представляет собой бегунок для представления числовых значений; его можно ориентировать по горизонтали или по вертикали и использовать как для ввода, так и для вывода. О других виджетах вы можете прочесть в документации.

12.1.6. Дополнительные замечания

Перспективы Tk туманны (то же можно сказать и о большинстве программных систем), но в ближайшем будущем он никуда не денется. Текущая версия Ruby/Tk основана на Tk 8.4, но, вероятно, со временем будет обновлена.

Нужно еще сказать несколько слов об операционных системах. Теоретически Tk – платформенно-независимая система, и практика не далека от теории. Однако есть сообщения о том, что версия для Windows не так стабильна, как для UNIX. На всякий случай отмечу, что все приведенные в этой главе примеры были протестированы в Windows и работают как задумано.

12.2. Ruby/GTK2

Библиотека GTK+ представляет собой побочный продукт развития графического редактора GIMP (the GNU Image Manipulation Program); аббревиатура расшифровывается как GIMP Toolkit. Как UNIX и BSD, GTK+ разработан в Калифорнийском университете в Беркли.

Если вы знакомы с системой X/Motif, скажем, что GTK+ внешне похожа на нее, но не так громоздка. Библиотека GTK+ зародилась в мире UNIX и лежит в основе графического менеджера GNOME (набирающего популярность у пользователей Linux), но при этом является более или менее кросс-платформенной. Начиная с версии GTK+ 2.0, поддерживаются не только различные варианты UNIX, но и семейство операционных систем MS Windows, а также Mac OS X с X Window System. Идет перенос на «родную» платформу Mac OS X, хотя пока эта версия еще не стабильна.

Расширение Ruby/GTK2 основано на GTK+ 2.0. Не путайте с Ruby/GTK (основанном на GTK+ 1.2), это расширение не совместимо и вообще считается устаревшим. В этом разделе мы будем говорить только о Ruby/GTK2.

12.2.1. Обзор

Ruby/GTK2 – это библиотека, позволяющая приложениям, написанным на языке Ruby, обращаться к средствам библиотеки GTK+ 2.x. GTK+ распространяется в исходных текстах на условиях лицензии GNU LGPL, поэтому может бесплатно использоваться в коммерческих приложениях.

Как и в большинстве библиотек для построения графических интерфейсов, в GTK+ есть концепции фреймов, окон, диалогов и менеджеров размещения. Она располагает богатым набором виджетов, включающим большинство стандартных, например метки, кнопки и текстовые поля, а также ряд более сложных: деревья и многоколонные списки.

Хотя GTK+ написана на C, спроектирована она в объектно-ориентированной манере. В связи с этим Ruby/GTK2 предоставляет объектно-ориентированный API, не слишком отдаляясь от исходной реализации на C. Кроме того, Ruby/GTK2 написана вручную, а не с помощью таких генераторов кода, как SWIG. Поэтому API выдержан в духе Ruby, с использованием блоков, необязательных аргументов и т.д. Справочное руководство можно найти на сайте <http://ruby-gnome2.sourceforge.jp/>.

Библиотека GTK+ создана на базе других библиотек: GLib, Pango, ATK, Cairo и GDK. Она поддерживает неграфические функции (GLib), отображение многоязычных текстов в кодировке UTF-8 (Pango), средства облегчения работы (Atk), графический рендеринг (Cairo), низкоуровневые графические объекты (Gdk), а также множество виджетов и высокоуровневых графических объектов (Gtk).

Во время работы над книгой текущей была версия Ruby/GTK2 0.14.1, совместимая с текущими стабильными версиями Ruby и GTK+ (2.0). Помимо Linux, она поддерживает семейство OC Windows и Mac OS X (с X Window System). Идет работа по переносу на «родную» платформу Mac OS X, хотя пока эта версия еще не стабильна.

GTK+ – объектно-ориентированная библиотека, поддерживающая логически стройную иерархию виджетов. Классы `Gtk::Bin` и `Gtk::Container` весьма развиты, а комбинация менеджеров размещения `Gtk::Box` и `Gtk::Table` обеспечивает простоту, но в то же время гибкие средства управления геометрией. В Ruby/GTK2 есть удобный механизм установки обработчиков сигналов.

Среди виджетов GTK+ вы найдете меню, панели инструментов, всплывающие подсказки, деревья, индикаторы хода процесса, бегунки и календари. К слабым

местам текущей версии GTK+ можно отнести недостаточно богатый набор стандартных диалогов, с которыми к тому же трудно работать в модальном режиме. Есть недостатки и у стандартного многострочного текстового редактора.

Все строки, передаваемые методам Ruby/GTK2, должны быть представлены в кодировке UTF-8. Нельзя употреблять не ASCII-символы из некоторых одно- или многобайтовых кодовых страниц Windows. Поэтому не забывайте при редактировании Ruby-сценариев переключать редактор в режим UTF-8 и помещайте предложение `$KCODE="U"` в начале сценария.

12.2.2. Простое оконное приложение

Любая программа, в которой используется Ruby/GTK2, должна содержать директиву `require "gtk2"`. Функциональность Ruby/GTK2 предоставляется посредством модулей `Gtk` и `Gdk`, поэтому имена классов GTK+ обычно начинаются с префикса `Gtk::` или `Gdk::`.

Как правило, для инициализации Ruby/GTK2 мы вызываем метод `Gtk.init`, а затем создаем окно верхнего уровня и обработчик сигнала `destroy` (который поступает, когда пользователь закрывает окно). Метод `show_all` делает окно видимым, а обращение к `Gtk.main` запускает цикл обработки событий.

Мы еще вернемся к этой теме, но сначала рассмотрим пример. Следующий код, как и рассмотренная выше программа для Tk, отображает текущую дату:

```
$KCODE = "U"
require "gtk2"
Gtk.init

window = Gtk::Window.new("Today's Date")
window.signal_connect("destroy") { Gtk.main_quit }
str = Time.now.strftime("Today is %B %d, %Y")
window.add(Gtk::Label.new(str))
window.set_default_size(200, 100)
window.show_all
Gtk.main
```

О переменной `$KCODE` речь шла в главе 4. Метод `Gtk.init` инициализирует Ruby/GTK2.

Главное окно (типа `Gtk::Window`) создается как окно «верхнего уровня», а указанный текст отображается в полосе заголовка.

Далее создается обработчик сигнала `destroy`, который посылается при закрытии главного окна. Этот обработчик (в данном случае один блок) просто завершает главный цикл обработки событий. В документации по Ruby/GTK2 перечислены все сигналы, которые могут поступать каждому виджету (не забудьте и о супер-классах). Обычно они генерируются в результате манипуляций с мышью и клавиатурой, срабатывания таймеров, изменений состояния окна и т. д.

В следующей строке мы добавляем метку прямо в главное окно. Размер метки вычисляется автоматически на основе длины текста.

По умолчанию размеры родительских виджетов в GTK+ устанавливают автоматически, исходя из размеров потомков. В данном случае длина строки,

отображаемой выбранным по умолчанию шрифтом, определяет размер метки, а размер главного окна выбирается так, чтобы в нем поместилась метка. Такое окно получилось бы слишком маленьким, поэтому с помощью метода `set_default_size` мы говорим, что начальный размер главного окна должен составлять 200×100 пикселей.

Затем мы вызываем метод `show_all`, чтобы сделать главное окно и всех его потомков видимыми. По умолчанию главное окно скрыто, поэтому в большинстве приложений такой вызов необходим.

Метод `Gtk.main` запускает цикл обработки событий в GTK+. Он не возвращает управления, пока приложение не завершится. В данном случае обработчик события `destroy` приводит к выходу из `Gtk.main`, после чего завершается и все приложение.

12.2.3. Кнопки

Для создания кнопки в Ruby/GTK2 предназначен класс `Gtk::Button`. В простейшем случае мы задаем обработчик события `clicked`, которое возникает, когда пользователь щелкает по кнопке.

Программа в листинге 12.5 позволяет ввести одну строку в текстовое поле и после нажатия кнопки `All Caps!` преобразует ее в верхний регистр. На рис. 12.4 показано текстовое поле до нажатия кнопки.

Листинг 12.5. Кнопки в GTK

```
$KCODE = "U"
require "gtk2"

class SampleWindow < Gtk::Window

  def initialize
    super("Ruby/GTK2 Sample")
    signal_connect("destroy") { Gtk.main_quit }

    entry = Gtk::Entry.new

    button = Gtk::Button.new("All Caps!")
    button.signal_connect("clicked") {
      entry.text = entry.text.upcase
    }

    box = Gtk::HBox.new
    box.add(Gtk::Label.new("Text:"))
    box.add(entry)
    box.add(button)

    add(box)
    show_all
  end
end
```

```
Gtk.init
SampleWindow.new
Gtk.main
```



Рис. 12.4. Пример простой кнопки в GTK

В листинге 12.5 определен класс `SampleWindow`; при таком подходе класс может управлять собственным отображением и поведением (не заставляя вызывающую программу конфигурировать окно). Класс главного окна наследует `Gtk::Window`.

Как и в примере «Текущая дата», обработчик сигнала `destroy` завершает цикл обработки событий после закрытия главного окна.

Этот класс создает однострочное поле ввода (класс `Gtk::Entry`) и кнопку `Gtk::Button` с текстом `All Caps!`. С кнопкой связан обработчик события `clicked`, которое генерируется, когда пользователь нажимает и отпускает кнопку мыши, в то время как ее указатель находится над кнопкой.

Класс `Gtk::Window` — производный от `Gtk::Bin`, поэтому может содержать только один дочерний виджет. Чтобы добавить в окно два виджета, мы сначала помещаем их в контейнер `Gtk::VBox`, который, в свою очередь, делаем потомком главного окна. Виджеты, добавляемые в контейнер `Gtk::VBox`, по умолчанию размещаются начиная с его правой границы. Есть также контейнер `Gtk::VBox`, который упаковывает своих потомков по вертикали.

Как и раньше, чтобы главное окно (и все его потомки) стало видимым, необходимо вызвать метод `show_all`.

Обработчик события `clicked` вызывается при нажатии кнопки. Он получает текст, находящийся в поле ввода, преобразует его в верхний регистр и записывает обратно в поле ввода.

Собственно код приложения находится после определения класса `SampleWindow`. В нем всего лишь создается главное окно и запускается цикл обработки событий.

12.2.4. Текстовые поля

В библиотеке GTK+ есть класс `Gtk::Entry` для ввода одной строки текста — мы видели его в предыдущем примере. Существует также класс `Gtk::TextView`, представляющий собой мощный многострочный редактор; его мы и опишем.

Программа в листинге 12.6 создает многострочное текстовое поле и помещает в него текст. По мере изменения содержимого поля текущая длина текста отображается с помощью метки, расположенной в нижней части окна (рис. 12.5).

Листинг 12.6. Текстовый редактор в GTK

```
$KCODE = "U"
require "gtk2"

class TextWindow < Gtk::Window

  def initialize
```



```

super("Ruby/GTK2 Text Sample")
signal_connect("destroy") { Gtk.main_quit }
set_default_size(200, 100)
@text = Gtk::TextView.new
@text.wrap_mode = Gtk::TextTag::WRAP_WORD

@buffer = @text.buffer
@buffer.signal_connect("changed") {
  @status.text = "Length: " + @buffer.char_count.to_s
}

@buffer.create_tag('notice',
  'font' => "Times Bold Italic 18",
  'foreground' => "red")

@status = Gtk::Label.new

scroller = Gtk::ScrolledWindow.new
scroller.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_NEVER)
scroller.add(@text)

box = Gtk::VBox.new
box.add(scroller)
box.add(@status)
add(box)

iter = @buffer.start_iter
@buffer.insert(iter, "This is an editor")
iter.offset = 5
@buffer.insert(iter, "really ", "notice")

show_all
end
end

Gtk.init
TextWindow.new
Gtk.main

```

Структура программы такая же, как в примере с кнопкой: инициализировать Ruby/GTK2, определить класс главного окна, задать обработчик события, корректно завершающий приложение, и установить начальный размер окна. После



Рис. 12.5. Небольшой текстовый редактор в GTK

`initialize` вызывается метод `show_all`, который делает окно видимым. В последних двух строчках создается окно и запускается цикл обработки событий.

Мы создали виджет редактора с именем `@text`. Включен режим переноса строк, по умолчанию строки разбиваются без учета границы слов.

Переменная `@buffer` – это текстовый буфер для виджета `@text`. Мы установили обработчик события `changed`; он будет вызываться при вставке, удалении и изменении текста. Обработчик пользуется методом `char_count`, чтобы узнать текущую длину текста в редакторе и преобразовать ее в строку сообщения. Предложение `@status.text = text` отображает это сообщение в окне.

Далее мы конфигурируем виджет `@text` так, чтобы он показывал текст другим стилем. Для этого с помощью метода `create_tag` создается тег «notice», с которым связан шрифт «Times Bold Italic 18» и красный цвет. Класс `Gtk::TextTag` позволяет задавать и другие свойства тегов.

В данном случае мы хотим воспользоваться шрифтом из семейства Times; на платформе Windows мы, скорее всего, получим какой-то вариант шрифта Times Roman. В ОС Linux/UNIX параметром должна быть стандартная для X Window System строка указания шрифта. Система вернет шрифт, наиболее близкий к заданному.

Метка `@status` первоначально пуста. Ее текст будет изменен позже.

GTK+ предлагает два способа добавить полосы прокрутки. Можно напрямую создать объект `Gtk::Scrollbar` и с помощью сигналов синхронизировать его с ассоциированным виджетом. Но в большинстве случаев проще воспользоваться виджетом `Gtk::ScrolledWindow`.

Виджет `Gtk::ScrolledWindow` наследует `Gtk::Bin`, поэтому может содержать только один дочерний виджет. Но этот виджет может принадлежать классу `Gtk::Box` или любому другому контейнеру, допускающему несколько потомков. Ряд виджетов GTK+, в том числе и `Gtk::TextView`, автоматически взаимодействуют с `Gtk::ScrolledWindow`, не требуя почти никакого дополнительного кода.

В данном примере мы создали виджет `Gtk::ScrolledWindow` с именем `scroller` и сконфигурировали его методом `set_policy`. Мы решили не отображать горизонтальную полосу прокрутки вовсе, а вертикальную – только тогда, когда в редакторе больше строк, чем видно в окне. Сам текстовый редактор сделан непосредственным потомком `scroller`.

Теперь надо настроить контейнер `Gtk::VBox`, который расположит наши виджеты по вертикали. Сначала добавляется прокручиваемое окно, содержащее поле ввода, поэтому оно окажется самым верхним. Метка `@status` располагается под ним. Напоследок сам контейнер добавляется в главное окно.

В следующих четырех строчках в поле ввода добавляется текст. В первой строчке мы получаем объект `Gtk::TextIter`, соответствующий началу текста (`offset = 0`), и вставляем в это место строку. Поскольку до этого момента никакого текста в поле еще не было, только сюда и можно его вставить. Затем вставляется другой кусок текста со смещением 5. В результате редактор будет содержать строку *This really is an editor.*

Поскольку мы предварительно установили обработчик события `changed`, он будет вызываться после каждого обращения к `insert`. Следовательно, статус будет

отображаться правильно, несмотря на то что пользователь еще не вносил никаких изменений в текст.

12.2.5. Прочие виджеты

Даже для организации сравнительно простого графического интерфейса текстовых полей и кнопок может оказаться недостаточно. Нужны переключатели, флажки и другие виджеты. В следующем примере демонстрируются некоторые из них.

В листинге 12.7 предполагается, что пользователь хочет заказать билет на самолет. Для выбора города назначения используются классы `Gtk::TreeView`, `Gtk::ListStore` и `Gtk::TreeViewColumn` (многоколонный список). Флажок (класс `Gtk::CheckButton`) определяет, нужен ли обратный билет, а переключатель (класс `Gtk::RadioButton`) позволяет указать класс салона. Завершает интерфейс кнопка `Purchase` (Заказать) – рис. 12.6.

Листинг 12.7. Заказа билета на самолет

```
$KCODE = "U"
require "gtk2"

class TicketWindow < Gtk::Window

  def initialize
    super("Purchase Ticket")
    signal_connect("destroy") { Gtk.main_quit }

    dest_model = Gtk::ListStore.new(String, String)
    dest_view = Gtk::TreeView.new(dest_model)
    dest_column = Gtk::TreeViewColumn.new("Destination",
      Gtk::CellRendererText.new,
      :text => 0)
    dest_view.append_column(dest_column)
    country_column = Gtk::TreeViewColumn.new("Country",
      Gtk::CellRendererText.new,
      :text => 1)
    dest_view.append_column(country_column)
    dest_view.selection.set_mode(Gtk::SELECTION_SINGLE)

    [ ["Cairo", "Egypt"], ["New York", "USA"],
      ["Tokyo", "Japan"] ].each do |destination, country|
      iter = dest_model.append
      iter[0] = destination
      iter[1] = country
    end
    dest_view.selection.signal_connect("changed") do
      @city = dest_view.selection.selected[0]
    end

    @round_trip = Gtk::CheckButton.new("Round Trip")

    purchase = Gtk::Button.new("Purchase")
    purchase.signal_connect("clicked") { cmd_purchase }
```

```

@result = Gtk::Label.new

@coach = Gtk::RadioButton.new("Coach class")
@business = Gtk::RadioButton.new(@coach, "Business class")
@first = Gtk::RadioButton.new(@coach, "First class")

flight_box = Gtk::VBox.new
flight_box.add(dest_view).add(@round_trip)

seat_box = Gtk::VBox.new
seat_box.add(@coach).add(@business).add(@first)

top_box = Gtk::HBox.new
top_box.add(flight_box).add(seat_box)

main_box = Gtk::VBox.new
main_box.add(top_box).add(purchase).add(@result)

add(main_box)
show_all
end

def cmd_purchase
  text = @city
  if @first.active?
    text += ": first class"
  elsif @business.active?
    text += ": business class"
  elsif @coach.active?
    text += ": coach"
  end
  text += ", round trip " if @round_trip.active?
  @result.text = text
end

end

Gtk.init
TicketWindow.new
Gtk.main

```



Рис. 12.6. Различные виджеты GTK

В этом приложении, как и в предыдущих примерах, создается главное окно с обработчиком события. Затем формируется список с двумя колонками, дизайн которого следует паттерну Модель–Вид–Контроллер (Model–View–Controller – MVC); класс `Gtk::ListStore` (модель) имеет две колонки типа `String`.

Далее создается виджет `Gtk::TreeView`. Класс `Gtk::TreeViewColumn` конфигурирует эту колонку. Первая колонка называется «Destination», а для отображения клеток применяется класс рисовальщика `Gtk::CellRendererText`. Первая колонка модели (с номером 0) – `Gtk::ListStore` – служит значением текстового свойства. Итак, рисовальщики клеток наполняют древесную модель данными. В GTK+ 2.x есть несколько готовых рисовальщиков клеток, в том числе `Gtk::CellRendererText`, `Gtk::CellRendererPixbuf` и `Gtk::CellRendererToggle`. Далее в список добавляются три строки данных и устанавливается обработчик события "changed", который будет вызываться, когда пользователь выберет другую строку. Этот обработчик изменит значение переменной `@city`, записав в нее текст из первой колонки только что выбранной строки.

Затем создается простой флажок (`Gtk::CheckButton`) и кнопка (`Gtk::Button`). Обработчик события нажатия кнопки вызовет метод `cmd_purchase`. Метка `@result` первоначально пуста, но позже в нее будет записана строка, определяющая вид заказанного билета.

Три переключателя создаются как члены одной группы, то есть в любой момент может быть выбран лишь один из них. Когда пользователь щелкает по любому переключателю, равнее выбранный сбрасывается. Первым параметром конструктору переключателя передается первый переключатель из той же группы. Поэтому у конструктора первого переключателя в группе этого параметра нет, а остальным передается ссылка на первый переключатель.

Виджеты нужно организовать на экране так, чтобы пользователю было удобно. Мы воспользовались комбинацией контейнеров `Gtk::HBox` и `Gtk::VBox`. Список расположен над флажком. Все три переключателя расположены вертикально справа от списка. А кнопка помещена под всеми остальными виджетами.

Метод `cmd_purchase` очень прост: он строит строку, отражающую состояние всех виджетов в момент нажатия кнопки. У переключателей и флажков есть метод `active?`, который возвращает `true`, если виджет отмечен. Построенная строка записывается в метку `@result` и потому появляется на экране.

Во многих приложениях интерфейс содержит меню. В следующем примере показано, как можно организовать меню в Ruby/GTK2. Заодно демонстрируется применение всплывающих подсказок – мелкая деталь, способная украсить любую программу.

В листинге 12.8 создается главное окно с меню, содержащим пункт `File` и еще два фиктивных пункта. В меню `File` есть команда `Exit`, которая завершает приложение. Оба пункта `File` и `Exit` снабжены всплывающими подсказками.

Листинг 12.8. Пример меню в GTK

```
$KCODE = "U"
require "gtk2"
class MenuWindow < Gtk::Window
```

```
def initialize
  super("Ruby/GTK2 Menu Sample")
  signal_connect("destroy") { Gtk.main_quit }

  file_exit_item = Gtk::MenuItem.new("_Exit")
  file_exit_item.signal_connect("activate") { Gtk.main_quit }

  file_menu = Gtk::Menu.new
  file_menu.add(file_exit_item)

  file_menu_item = Gtk::MenuItem.new("_File")
  file_menu_item.submenu = file_menu

  menubar = Gtk::MenuBar.new
  menubar.append(file_menu_item)
  menubar.append(Gtk::MenuItem.new("_Nothing"))
  menubar.append(Gtk::MenuItem.new("_Useless"))

  tooltips = Gtk::Tooltips.new
  tooltips.set_tip(file_exit_item, "Exit the app", "")

  box = Gtk::VBox.new
  box.pack_start(menubar, false, false, 0)
  box.add(Gtk::Label.new("Try the menu and tooltips!"))
  add(box)
  set_default_size(300, 100)
  show_all
end
end

Gtk.init
MenuWindow.new
Gtk.main
```

И здесь базовая структура программы такая же, как в предыдущих примерах. В данном случае мы создаем пункт меню `Gtk::MenuItem` с именем `Exit` и задаем для него обработчик события, который завершает программу. Событие называется `activate` и генерируется, когда пользователь выбирает пункт меню.

Далее создается меню `File` и в него добавляется пункт `Exit`. Это все, что требуется для создания выпадающего меню. В конце создается пункт меню `File`; именно он и появится в полосе меню. Чтобы присоединить пункт `File` к меню `File`, мы вызываем метод `submenu=`.

Затем создается полоса меню `Gtk::MenuBar`, в которую добавляются три меню: `File`, `Nothing` и `Useless`. Что-то делает лишь первое меню, остальные приведены только для демонстрации.

Всплывающими подсказками управляет единственный объект `Gtk::Tooltips`. Чтобы создать подсказку для любого виджета, например для пункта меню, нужно вызвать метод `set_tip`, которому передаются сам виджет, текст подсказки и строка,

содержащая дополнительный скрытый текст. Скрытый текст не показывается в составе подсказки, но может, например, использоваться для организации оперативной справки.

Чтобы разместить полосу меню в верхней части главного окна, мы взяли `Gtk::Vbox` в качестве самого внешнего контейнера. В данном случае мы добавляем в него полосу меню не методом `add`, а методом `pack_start`, чтобы точнее контролировать внешний вид и положение виджета.

Первым параметром методу `pack_start` передается размещаемый виджет. Вторым параметр – булевский признак, показывающий, должен ли виджет занимать все доступное пространство. Отметим, что виджет при этом не растет, а обычно просто размещается в центре контейнера. Мы хотим, чтобы полоса меню располагалась сверху, поэтому передаем `false`.

Третий параметр – тоже булевская величина, говорящая о том, должны ли размеры виджет быть изменены так, чтобы он занял все отведенное пространство. Нам нужна лишь узкая полоса меню, поэтому мы и тут передаем `false`. Последний параметр метода `pack_start` задает отступы, то есть пустое место вокруг виджета. Нам это ни к чему, поэтому мы передаем нуль.

Большую часть главного окна занимает метка. Напоследок мы принудительно устанавливаем размер окна 300×100 пикселей.

12.2.6. Дополнительные замечания

Ruby/GTK2 – это часть проекта Ruby-GNOME2. *GNOME* – пакет более высокого уровня, основанный на библиотеке *GTK+*, а *Ruby-GNOME2* – набор привязок для библиотек, входящих в состав *GNOME*.

Ruby-GNOME2 включает следующие библиотеки:

- *Базовые библиотеки.* Они включены в пакеты `ruby-gtk2`. Иногда термином «Ruby/GTK2» обозначают всю совокупность этих библиотек. Они работают на платформах UNIX, MS Windows, Mac OS X (с X11) и Cygwin (с X11). Все они необходимы для других библиотек, входящих в состав Ruby-GNOME2.
- *Ruby/GLib2.* GLib – низкоуровневая инфраструктурная библиотека. Она предоставляет структуры данных на языке C, слой, обеспечивающий переносимость, поддержку Unicode и интерфейсы для поддержки цикла обработки событий, потоков, динамической загрузки и системы объектов. Ruby/GLib2 – обертка библиотеки GLib. Поскольку в Ruby уже есть хорошие классы для работы со строками и списками, некоторые функции GLib не реализованы. С другой стороны, Ruby/GLib2 содержит ряд важных функций для преобразования между объектами на C и на Ruby. Эта библиотека необходима для всех остальных библиотек, входящих в состав Ruby/GTK2.
- *Ruby/ATK.* Эта библиотека предоставляет набор интерфейсов для облегчения работы. Приложение или набор средств разработки, поддерживающие интерфейсы ATK, могут применяться с такими инструментами, как считыватели с экрана, лупы и альтернативные устройства ввода.

- *Ruby/Pango*. Библиотека для отображения текста с упором на интернационализацию с использованием кодировки UTF-8. Образует основу для работы с текстами и шрифтами в GTK+ (2.0).
- *Ruby/GdkPixbuf2*. Библиотека для загрузки и манипулирования изображениями. Поддерживает многочисленные графические форматы, включая JPEG, PNG, GIF и другие.
- *Ruby/GDK2*. Промежуточный слой, изолирующий GTK+ от деталей оконной системы.
- *Ruby/GTK2*. Основные виджеты для построения графических интерфейсов.
- *Дополнительные библиотеки* включены в пакеты *ruby-gnome2* наряду с базовыми. Все они работают в UNIX, а некоторые (*Ruby/GtkGLExt*, *Ruby/Libglade2*) также в MS Windows и Mac OS X. Некоторые библиотеки теоретически должны работать в Mac OS X (с X11) и Cygwin (с X11), но недостаточно хорошо протестированы.
- *Ruby/GNOME2*. Содержит дополнительные виджеты для проекта GNOME.
- *Ruby/GnomeCanvas2*. Виджет для интерактивного создания структурной графики.
- *Ruby/GConf2*. Прозрачная для процесса конфигурационная база данных (аналог реестра в Windows).
- *Ruby/GnomeVFS*. Позволяет приложениям одинаково обращаться к локальным и удаленным файлам.
- *Ruby/Gstreamer*. Мультимедийный каркас для обработки аудио и видеoinформации.
- *Ruby/GtkHtml2*. Виджет для представления HTML-документов.
- *Ruby/GtkGLExt*. Предлагает трехмерный рендеринг с использованием технологии OpenGL.
- *Ruby/GtkSourceView*. Виджет Text с поддержкой синтаксической подсветки и других возможностей, ожидаемых от редактора исходных текстов.
- *Ruby/GtkMozEmbed*. Виджет, включающий механизм рендеринга Mozilla Gecko.
- *Ruby/Libart2*. Поддержка базовых средств рисования.
- *Ruby/Libgda*. Интерфейс к архитектуре GDA (GNU Data Access), обеспечивающий доступ к источникам данных, например СУБД и LDAP.
- *Ruby/Libglade2*. Позволяет приложению загружать описание пользовательского интерфейса из XML-файлов во время выполнения. XML-файлы создаются мощным редактором интерфейсов GLADE, который упрощает издание интернационализированных графических интерфейсов пользователя.
- *Ruby/PanelApplet*. Библиотека для создания апплетов, размещаемых на панели GNOME.

- *Ruby/GnomePrint* и *Ruby/GnomePrintUI*. Виджеты для печати.
- *Ruby/RSVG*. Поддержка векторной графики в формате SVG.
- *Внешние библиотеки* загружаются библиотеками, входящими в состав Ruby-GNOME2.
- *Ruby/Cairo*. Библиотека двумерной графики с поддержкой разнообразных устройств вывода. В текущей версии поддерживаются X Window System, win32 и буферы изображения. На стадии эксперимента находятся поддержка OpenGL (с помощью библиотеки *glitz*), Quartz, XCB, PostScript и PDF. Эта библиотека загружается базовыми библиотеками. Для Ruby/Cairo требуется также Ruby/GLib2. Официальный сайт проекта – <http://cairographics.org/>.
- *Ruby/OpenGL*. Интерфейс к библиотеке трехмерной графики OpenGL. Требуется библиотеке Ruby/GtkGLExt2. Работает на многих платформах. Официальный сайт проекта – <http://www2.giganet.net/~yoshi/>.
- *Ruby-GetText-Package*. Предоставляет средства для управления справочниками переведенных сообщений для локализации (см. главу 4). С помощью этого пакета локализована библиотека Ruby/Libglade2, то же самое можно сделать и для других библиотек. Официальный сайт проекта <http://gettext.rubyforge.org/>.

Официальная домашняя страница проекта Ruby-GNOME2 – <http://ruby-gnome2.sourceforge.jp/>. Там вы найдете выпущенные версии всех библиотек, руководство по установке, справочные руководства по API, учебные пособия и примеры программ. Официальный сайт проекта GNOME – <http://www.gnome.org/>, а проекта GTK+ – <http://www.gtk.org/>.

12.3. FXRuby (FOX)

FOX – относительно новая технология, упор в ней сделан на быстроедействие и межплатформенную совместимость. В значительной степени совместимость обусловлена самодостаточностью – в отличие от многих систем, это не обертка платформенного API.

Сама система написана на языке C++, хотя привязки можно создать практически для любого языка (для Ruby они уже имеются). Поскольку система изначально объектно-ориентированная, она хорошо сопрягается с Ruby и довольно естественно расширяется.

Технология FOX не так широко распространена, как Tk или GTK+, но популярна в среде программистов на Ruby. Отчасти это обусловлено наличием великолепной привязки FXRuby (см. сайт <http://fxruby.org>). FXRuby – плод трудов Лайла Джонсона (Lyle Johnson), который немало сделал для поддержки и документирования библиотеки. Он же в течение многих лет предоставляет техническую поддержку и оказал неоценимую помощь при написании этого раздела.

12.3.1. Обзор

FXRuby – это привязка к Ruby библиотеки FOX, написанной на C++. В нее входит много классов для разработки полноценных графических приложений. Хотя аббревиатура FOX означает Free Objects for X (Бесплатные объекты для X), она была успешно перенесена и на другие платформы, включая MS Windows. Лайл Джонсон написал привязку FOX к Ruby, а также перенес саму библиотеку на платформу Windows. Исходную версию библиотеки FOX разработал Джероен ван дер Зийп (Jeroen van der Zijp) при поддержке компании CFD Research Corporation.

Виджеты FOX обладают современным внешним обликом. По полноте они могут соперничать с платформенными интерфейсами, в том числе и с MS Windows, при этом располагая возможностями, сильно превосходящими многие другие библиотеки виджетов.

Библиотеку классов FOX легко освоит программист, знакомый с другими средствами разработки графических интерфейсов. API не содержит зависимостей от платформы. Поскольку FOX написана на C++, некоторые аспекты API *FxRuby* сохраняют влияние статической природы и соглашений, принятых в C++ (например, перечисления и поразрядные операции).

Центральным механизмом, упрощающим работу с FOX, является парадигма сообщение/получатель. Любой объект в FOX – это экземпляр класса *fxObject* или одного из его подклассов. Определяемые пользователем объекты также должны наследовать одному из этих классов. Любой экземпляр *fxObject* может посылать и получать сообщения. Сообщение связывается к конкретным получателем во время выполнения в момент отправки.

Внутри FOX сообщение представляется типом, идентификатором и данными. Классы FOX пользуются общим набором определений сообщений, что позволяет виджетам взаимодействовать.

Обработчик сообщения должен вернуть 1, если сообщение обработано, и 0 в противном случае. FOX не перенаправляет необработанные сообщения другим виджетам неявно. Возвращаемое значение используется для того, чтобы понять, нужно ли обновлять интерфейс. Приложение *FXRuby* могло бы воспользоваться возвращаемым значением, чтобы самостоятельно перенаправить необработанные сообщения и тем самым реализовать паттерн Chain of Responsibility (цепочка обязанностей), описанный в книге E. Gamma, R. Helm, R. Johnson, J. Vlissides «Design Patterns»^{*}.

Еще один механизм FOX – парадигма автоматического обновления. Неявный цикл обработки событий в FOX включает фазу обновления, в которой объекты FOX могут обработать сообщения об обновлении. Обычно обработчик такого сообщения изменяет внешний вид того или иного виджета, основываясь на текущем состоянии данных приложения. Например, программа, показанная в листинге 12.9 (см. раздел 12.3.3), имеет кнопку, которая обновляет собственное состояние «активна / не активна» в зависимости от значения некоторой переменной.

^{*} Русский перевод: Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влассидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – М.: ДМК, Питер, 2001.

12.3.2. Простое оконное приложение

Вот пример минимального приложения FXRuby, которое делает то же самое, что рассмотренные выше приложения Tk и GTK+:

```
require 'fox16'      # Используются привязки к FOX 1.6.

include Fox
application = FXApp.new
main = FXMainWindow.new(application, "Today's Date")
str = Time.now.strftime("&Today is %B %d, %Y")
button = FXButton.new(main, str)
button.connect(SEL_COMMAND) { application.exit }
application.create
main.show(PLACEMENT_SCREEN)
application.run
```

Этого примера достаточно для демонстрации двух важнейших классов FXRuby: FXApp и FXMainWindow. Приложение должно в самом начале создать и инициализировать объект FXApp. FXMainWindow – подкласс FXTopWindow; каждый виджет в FOX – некая разновидность «окна». Класс FXTopWindow представляет окно верхнего уровня, которое появляется непосредственно на экране. Более сложное приложение FXRuby обычно создает подкласс FXMainWindow и размещает в нем виджеты на этапе инициализации.

Конструктору FXMainWindow необходимо передать первым параметром объект FXApp. Второй параметр – заголовок окна. По умолчанию экземпляр FXMainWindow размещается в центре экрана и снабжается всеми стандартными элементами, присущими FXTopWindow. Таким образом, для окна отображается полоса заголовка с кнопками свертывания, разворачивания и закрытия. Его размеры можно изменять.

Атрибут decorations главного окна позволяет явно указать необходимые элементы оформления. Например, можно запретить изменение размеров:

```
main = FXMainWindow.new(application, "Today's Date")
main.decorations = DECOR_TITLE | DECOR_CLOSE
```

Значение decorations образуется комбинированием битовых флагов, как это принято в C++. В примере выше окно имеет только заголовок и кнопку закрытия.

В этом простом примере главное окно содержит всего один виджет – экземпляр класса FXButton, в котором отображается текущая дата.

```
str = Time.now.strftime("&Today is %B %d, %Y")
button = FXButton.new(main, str)
```

Первый аргумент конструктора FXButton – родительское окно, содержащее данный виджет. В нашем примере это главное окно. Второй аргумент – текст, рисуемый на кнопке.

В следующей строчке показано, как с помощью метода connect ассоциировать с кнопкой блок:

```
button.connect(SEL_COMMAND) { application.exit }
```

Здесь говорится, что когда кнопка отправляет командное сообщение (то есть сообщение типа SEL_COMMAND), следует вызвать метод exit.

В оставшихся строчках мы наблюдаем «ритуал обручения» объектов FXApp и FXMainWindow:

```
application.create
main.show(PLACEMENT_SCREEN)
application.run
```

Любое приложение FXRuby должно включать подобные строки, чтобы создать экземпляр приложения, показать окно FXMainWindow и запустить цикл обработки событий. Аргумент PLACEMENT_SCREEN метода show определяет, в каком месте экрана должно появиться окно. Из других возможных значений упомянем PLACEMENT_CURSOR (поместить окно там, где находится курсор), PLACEMENT_OWNER (в центре окна-владельца) и PLACEMENT_MAXIMIZED (раскрыть окно на весь экран).

12.3.3. Кнопки

Вы уже видели, как организуется работа с кнопками в FXRuby. Заглянем немного глубже.

На кнопке может размещаться не только короткая строка. Допустимы и несколько строк, разделенных символом новой строки:

```
text = "&Hello, World!\n" +
      "Do you see multiple lines of text?"
FXButton.new(self, text)
```

Обратите внимание на амперсанд перед буквой Н в строке "hello, world!". Он задает «горячую клавишу», нажатие которой эквивалентно щелчку по кнопке.

На кнопке может быть также нарисовано изображение, заданное в разных форматах. Например:

```
text = "&Hello, World!\n" +
      "Do you see the icon?\n" +
      "Do you see multiple lines of text?"
icon = File.open("some_icon.gif", "rb") do |file|
  FXGIFIcon.new(app, file.read)
end
FXButton.new(self, text, icon)
```

В листинге 12.9 иллюстрируется механизм обновления состояния интерфейса, реализованный в FOX:

Листинг 12.9. Обновление состояния интерфейса в FOX

```
require 'fox16'

include Fox

class TwoButtonUpdateWindow < FXMainWindow

  def initialize(app)
    # Сначала инициализируем базовый класс.
    super(app, "Update Example", nil, nil,
          DECOR_TITLE | DECOR_CLOSE)
```

```
# Первая кнопка:
@button_one = FXButton.new(self, "Enable Button 2")
@button_one_enabled = true

# Вторая кнопка:
@button_two = FXButton.new(self, "Enable Button 1")
@button_two.disable
@button_two_enabled = false

# Устанавливаем обработчики сообщений.
@button_one.connect(SEL_COMMAND, method(:onCommand))
@button_two.connect(SEL_COMMAND, method(:onCommand))
@button_one.connect(SEL_UPDATE, method(:onUpdate))
@button_two.connect(SEL_UPDATE, method(:onUpdate))
end

def onCommand(sender, sel, ptr)
  # Обновить состояние приложения.
  @button_one_enabled = !@button_one_enabled
  @button_two_enabled = !@button_two_enabled
end

def onUpdate(sender, sel, ptr)
  # Обновить кнопки в зависимости от состояния приложения.
  @button_one_enabled ?
    @button_one.enable : @button_one.disable
  @button_two_enabled ?
    @button_two.enable : @button_two.disable
end

end

application = FXApp.new
main = TwoButtonUpdateWindow.new(application)
application.create
main.show(PLACEMENT_SCREEN)
application.run
```

Здесь в главное окно добавлено две кнопки. Мы снова воспользовались методом `connect`, чтобы связать сообщение `SEL_COMMAND` от кнопок с кодом, но на этот раз код представляет собой метод, а не блок:

```
@button_one.connect(SEL_COMMAND, method(:onCommand))
```

В этом примере мы встречаем еще один тип сообщения – `SEL_UPDATE`. Такое сообщение позволяет сделать виджеты независимыми друг от друга и от кода приложения. Как видим, ни одна кнопка не подозревает о существовании другой. Первая кнопка обновляет состояние второй, посылая сообщение обработчикам, которые отвечают за изменение состояния.

12.3.4. Текстовые поля

FOX располагает полезными средствами для ввода текста. В следующем примере демонстрируется применение класса `FXTextField` для редактирования одной строки. Параметры определяют формат текста. Значение `TEXTFIELD_PASSWD` скрывает текст, являющийся паролем, `TEXTFIELD_REAL` позволяет вводить только действительные числа в научной нотации, а `TEXTFIELD_INTEGER` – только целые числа.

```
simple = FXTextField.new(main, 20, nil, 0,
                        JUSTIFY_RIGHT|FRAME_SUNKEN|
                        FRAME_THICK|LAYOUT_SIDE_TOP)
simple.text = "Simple Text Field"
passwd = FXTextField.new(main, 20, nil, 0,
                        JUSTIFY_RIGHT|TEXTFIELD_PASSWD|
                        FRAME_SUNKEN|FRAME_THICK|
                        LAYOUT_SIDE_TOP)
passwd.text = "Password"
real = FXTextField.new(main, 20, nil, 0,
                      TEXTFIELD_REAL|FRAME_SUNKEN|
                      FRAME_THICK|LAYOUT_SIDE_TOP|
                      LAYOUT_FIX_HEIGHT, 0, 0, 0, 30)
real.text = "1.0E+3"
int = FXTextField.new(main, 20, nil, 0, TEXTFIELD_INTEGER|
                      FRAME_SUNKEN|FRAME_THICK|
                      LAYOUT_SIDE_TOP|LAYOUT_FIX_HEIGHT,
                      0, 0, 0, 30)
int.text = "1000"
```

Ниже показан простой способ ввода текста с помощью диалогового окна. В зависимости от выбранного метода можно будет вводить любой текст, только действительные или только целые числа.

```
puts FXInputDialog.getString("initial text",
                             self, "Диалог для ввода текст",
                             "Введите текст:", nil)
puts FXInputDialog.getInteger(1200, self,
                              "Диалог для ввода целого числа",
                              "Введите целое число:", nil)
puts FXInputDialog.getReal(1.03e7, self,
                           "Диалог для ввода числа в научной нотации",
                           "Введите действительное число:", nil)
```

Для экономии места мы не станем приводить полный текст приложения. Но, конечно, перед выводом диалогового окна необходимо выполнить обычную инициализацию.

12.3.5. Прочие виджеты

В следующем примере демонстрируется использование меню и полос меню в приложениях. Отметим, что объекты `FXMenuCommand` следуют общей для FOX парадигме сообщение/получатель, с которой мы уже сталкивались при работе с кнопками:

```
require 'fox16'

include Fox
```

```

application = FXApp.new
main = FXMainWindow.new(application, "Simple Menu")
menubar = FXMenuBar.new(main, LAYOUT_SIDE_TOP |
    LAYOUT_FILL_X)
filemenu = FXMenuPane.new(main)
quit_cmd = FXMenuCommand.new(filemenu, "&Quit\tCtl-Q")
quit_cmd.connect(SEL_COMMAND) { application.exit }
FXMenuTitle.new(menubar, "&File", nil, filemenu)
application.create
main.show(PLACEMENT_SCREEN)
application.run

```

Здесь и `FXMenuBar`, и `FXMenuPane` добавляются непосредственно в главное окно `FXMainWindow`. Благодаря параметрам `LAYOUT_SIDE_TOP` и `LAYOUT_FILL_X` полоса меню размещается в верхней части родительского окна и простирается от левой до правой границы. Текст команды меню `"&Quit\tCtl-Q"` подразумевает, что комбинация клавиш `Alt+Q` играет роль «горячей клавиши», а `Ctrl+Q` – клавиши быстрого выбора пункта меню. Последовательное нажатие `Alt+F` и `Alt+Q` эквивалентно щелчку по меню `File` с последующим выбором пункта `Quit`. Нажатие `Ctrl+Q` заменяет всю последовательность.

В классе `FXTopWindow` есть метод для свертывания главного окна. Следующие три строчки добавляют в меню `File` команду, которая свернет окно:

```

FXMenuCommand.new(filemenu, "&Icon\tCtl-I") do |cmd|
  cmd.connect(SEL_COMMAND) { main.minimize }
end

```

На этом примере мы видим еще один прием, полезный при конструировании команды меню. Если вам не нужна ссылка на виджет, представляющий команду меню, то можно просто присоединить блок к вызову `FXMenuCommand.new` и выполнить всю инициализацию виджета внутри блока. Разумеется, этот прием применим к любому встроенному в `FOX` классу.

В листинге 12.10 демонстрируются переключатели.

Листинг 12.10. Переключатели в `FOX`

```

require 'fox16'

include Fox

class RadioButtonHandlerWindow < FXMainWindow
  def initialize(app)
    # Invoke base class initialize first
    super(app, "Radio Button Handler", nil, nil,
          DECOR_TITLE | DECOR_CLOSE)

    choices = [ "Good", "Better", "Best" ]

    group = FXGroupBox.new(self, "Radio Test Group",
                          LAYOUT_SIDE_TOP |

```

```
FRAME_GROOVE |
LAYOUT_FILL_X)
```

```
choices.each do |choice|
  FXRadioButton.new(group, choice,
    nil, 0,
    ICON_BEFORE_TEXT |
    LAYOUT_SIDE_TOP)
end
end
end

application = FXApp.new
main = RadioButtonHandlerWindow.new(application)
application.create
main.show(PLACEMENT_SCREEN)
application.run
```

Группы переключателей – стандартное средство в графических приложениях, предназначенное для выбора одного из взаимно исключающих вариантов. В данном примере варианты представлены массивом из трех строк:

```
choices = [ "Good", "Better", "Best" ]
```

В главное окно добавляется объект `FXGroupBox`, который визуально указывает, что три переключателя взаимосвязаны, а затем в этот контейнер добавляются сами переключатели (по одному на каждый вариант). Но сам контейнер `FXGroupBox` ничего не делает для того, чтобы обеспечить взаимное исключение. Если запустить пример в таком виде, то вы сможете выбрать более одного переключателя.

Есть несколько способов обеспечить ожидаемое поведение переключателей, но в приложениях FOX чаще всего для этой цели используют получатель данных – класс `FXDataTarget`. Это специальный объект, играющий роль хранителя какого-то значения. Как и любой другой объект в FOX, `FXDataTarget` может посылать и получать сообщения.

Программа в листинге 12.11 – модифицированный вариант предыдущей, в ней демонстрируется применение получателей данных.

Листинг 12.11. Переключатели в FOX и получатели данных

```
require 'fox16'

include Fox

class RadioButtonHandlerWindow < FXMainWindow

  def initialize(app)
    # Сначала вызвать инициализатор базового класса.
    super(app, "Radio Button Handler", nil, nil,
      DECOR_TITLE | DECOR_CLOSE)

    choices = [ "Good", "Better", "Best" ]
```

```

default_choice = 0
@choice = FXDataTarget.new(default_choice)

group = FXGroupBox.new(self, "Radio Test Group",
    LAYOUT_SIDE_TOP |
    FRAME_GROOVE |
    LAYOUT_FILL_X)

choices.each_with_index do |choice, index|
    FXRadioButton.new(group, choice,
        @choice, FXDataTarget::ID_OPTION+index,
        ICON_BEFORE_TEXT |
        LAYOUT_SIDE_TOP)
end
end
end

application = FXApp.new
main = RadioButtonHandlerWindow.new(application)
application.create
main.show(PLACEMENT_SCREEN)
application.run

```

В этом примере `@choice` – экземпляр `FXDataTarget`, значением которого является целочисленный индекс выбранного в данный момент положения переключателя. Получатель данных инициализирован нулем, что соответствует элементу «Good» массива `choices`.

При конструировании каждого переключателя задается получатель данных, а идентификатор сообщения от переключателя делается равным `FXDataTarget::ID_OPTION` плюс желаемое значение. Если теперь запустить пример, то вы увидите, что переключатель стал вести себя как положено.

Для добавления в окно списка `FXList` и его инициализации тоже достаточно нескольких строк. Значение `LIST_BROWSESELECT` позволяет выбирать из списка ровно один элемент. В начальный момент выбран самый первый из них. Значение `LIST_SINGLESELECT` допускает выбор не более одного элемента; в этом случае в начальный момент ни один элемент не выбран:

```

@list = FXList.new(self, nil, 0,
    LIST_BROWSESELECT |
    LAYOUT_FILL_X)

@names = ["Chuck", "Sally", "Franklin", "Schroeder",
    "Woodstock", "Matz", "Lucy"]
@names.each { |name| @list.appendItem(name) }

```

Отметим, что вместо метода `appendItem` можно использовать оператор вставки в массив, то есть последнюю строку можно было бы записать и так:

```
@names.each { |name| @list << name }
```

Весь пример целиком приведен в листинге 12.12. Сообщение обрабатывается в главном окне, в результате выводится выбранный элемент. Если был задан режим `LIST_SINGLESELECT`, то важно отличать щелчок, при котором элемент был выбран, от щелчка, который отменил выбор.

Листинг 12.12. Виджет FXList

```
require 'fox16'

include Fox

class ListHandlerWindow < FXMainWindow

  def initialize(app)
    # Сначала вызвать инициализатор базового класса.
    super(app, "List Handler", nil, nil,
          DECOR_TITLE | DECOR_CLOSE)

    @list = FXList.new(self, nil, 0,
                       LIST_BROWSESELECT |
                       LAYOUT_FILL_X)
    @list.connect(SEL_COMMAND) do |sender, sel, pos|
      puts pos.to_s + " => " + @names[pos]
    end

    @names = ["Chuck", "Sally", "Franklin",
              "Schroeder", "Woodstock",
              "Matz", "Lucy"]
    @names.each { |name| @list << name }
  end
end

application = FXApp.new
main = ListHandlerWindow.new(application)
application.create
main.show(PLACEMENT_SCREEN)
application.run
```

Если вместо LIST_BROWSESELECT поставить LIST_EXTENDEDSELECT, то в списке можно будет выбирать несколько элементов:

```
@list = FXList.new(self, nil, 0, LIST_EXTENDEDSELECT | LAYOUT_FILL_X)
```

Обработчик сообщений можно изменить так, чтобы он отображал все выбранные элементы. Чтобы понять, какие элементы списка выбраны, придется перебрать все:

```
@list.connect(SEL_COMMAND) do |sender, sel, pos|
  puts "Был щелчок по " + pos.to_s + " => " +
    @names[pos]
  puts "Выбраны следующие элементы:"
  @list.each do |item|
    if item.selected?
      puts "    " + item.text
    end
  end
end
```

Атрибут `numVisible` объекта `FXList` позволяет указать, сколько элементов списка видно одновременно. Существует также виджет `FXListBox`, который отображает только выбранное значение. Его интерфейс похож на интерфейс `FXList` с несколькими отличиями. Аргументы конструктора точно такие же, как видно из следующего примера. Отметим, что `FXListBox` позволяет выбирать только один элемент, поэтому значение `LIST_EXTENDEDSELECT` игнорируется:

```
@list_box = FXListBox.new(self, nil, 0, LIST_BROWSESELECT | LAYOUT_FILL_X)
@names = ["Chuck", "Sally", "Franklin", "Schroeder",
          "Woodstock", "Matz", "Lucy"]
@names.each { |name| @list_box << name }
```

Диалоговое окно можно определить один раз как подкласс класса `FXDialogBox`, а затем использовать для создания модальных или немодальных диалогов. Однако способы взаимодействия модальных и немодальных диалогов со своим владельцем различны.

Под модальным мы понимаем окно или диалог, который препятствует доступу к другим частям приложения, пока не будет закрыт. Немодальный диалог позволяет передавать фокус другим окнам приложения.

В следующем примере определяется класс модального и немодального диалогов. Для модального класса используются предопределенные сообщения `ID_CANCEL` и `ID_ACCEPT`. Немодальный класс пользуется только предопределенным сообщением `ID_HIDE`.

Для отображения немодального диалогов применяется уже знакомый метод `FXTopWindow.show`. Модальный диалог имеет собственный цикл обработки событий, отличный от цикла всего приложения. Для его отображения служит метод `FXDialogBox.execute`. Как видно из полного листинга программы, значение, возвращаемое методом `execute`, зависит от того, какое значение было передано методу приложения `stopModal` для завершения цикла обработки событий модального диалогов. В этом примере значение 1 говорит о том, что пользователь нажал кнопку `Accept`.

```
modal_btn.connect do
  dialog = ModalDialogBox.new(self)
  if dialog.execute(PLACEMENT_OWNER) == 1
    puts dialog.text
  end
end
```

Немодальный диалог работает параллельно с другими окнами приложения. Приложение должно запрашивать интересующие его данные у диалогов по мере необходимости. Один из способов известить о появлении новых данных – включить в диалог кнопку `Apply` (Применить), которая будет посылать зависящее от приложения сообщение главному окну. В примере ниже используется также таймер – еще одна интересная особенность `FxRuby`. Когда таймер срабатывает, главному окну посылается сообщение. Обработчик этого сообщения (показан ниже) запрашивает у диалогов новое значение и взводит таймер еще на одну секунду:

```
def onTimer(sender, sel, ptr)
  text = @non_modal_dialog.text
```

```

unless text == @previous
  @previous = text
  puts @previous
end
getApp().addTimeout(1000, method(:onTimer))
end

```

В листинге 12.13 приведен полный текст примера использования модальных и немодальных диалогов.

Листинг 12.13. Модальные и немодальные диалоги

```

require 'foxl6'

include Fox

class NonModalDialogBox < FXDialogBox

  def initialize(owner)
    # Сначала вызвать инициализатор базового класса.
    super(owner, "Test of Dialog Box",
          DECOR_TITLE|DECOR_BORDER)

    text_options = JUSTIFY_RIGHT | FRAME_SUNKEN |
                   FRAME_THICK | LAYOUT_SIDE_TOP
    @text_field = FXTextField.new(self, 20, nil, 0,
                                  text_options)
    @text_field.text = ""

    layout_options = LAYOUT_SIDE_TOP | FRAME_NONE |
                    LAYOUT_FILL_X | LAYOUT_FILL_Y |
                    PACK_UNIFORM_WIDTH
    layout = FXHorizontalFrame.new(self, layout_options)

    options = FRAME_RAISED | FRAME_THICK |
             LAYOUT_RIGHT | LAYOUT_CENTER_Y
    hide_btn = FXButton.new(layout, "&Hide", nil, nil, 0,
                           options)
    hide_btn.connect(SEL_COMMAND) { hide }
  end

  def text
    @text_field.text
  end
end

class ModalDialogBox < FXDialogBox

  def initialize(owner)
    # Сначала вызвать инициализатор базового класса.

```

```

    super(owner, "Test of Dialog Box",
          DECOR_TITLE|DECOR_BORDER)

    text_options = JUSTIFY_RIGHT | FRAME_SUNKEN |
                  FRAME_THICK | LAYOUT_SIDE_TOP
    @text_field = FXTextField.new(self, 20, nil, 0,
                                 text_options)
    @text_field.text = ""

    layout_options = LAYOUT_SIDE_TOP | FRAME_NONE |
                   LAYOUT_FILL_X | LAYOUT_FILL_Y |
                   PACK_UNIFORM_WIDTH
    layout = FXHorizontalFrame.new(self, layout_options)

    options = FRAME_RAISED | FRAME_THICK |
             LAYOUT_RIGHT | LAYOUT_CENTER_Y

    cancel_btn = FXButton.new(layout, "&Cancel", nil,
                              self, 0, options)
    cancel_btn.connect(SEL_COMMAND) do
      app.stopModal(self, 0)
      hide
    end

    accept_btn = FXButton.new(layout, "&Accept", nil,
                              self, 0, options)
    accept_btn.connect(SEL_COMMAND) do
      app.stopModal(self, 1)
      hide
    end

    def text
      @text_field.text
    end

    class DialogTestWindow < FXMainWindow

      def initialize(app)
        # Сначала инициализировать базовый класс.
        super(app, "Dialog Test", nil, nil,
              DECOR_ALL, 0, 0, 400, 200)

        layout_options = LAYOUT_SIDE_TOP | FRAME_NONE |
                       LAYOUT_FILL_X | LAYOUT_FILL_Y |
                       PACK_UNIFORM_WIDTH
        layout = FXHorizontalFrame.new(self, layout_options)

```

```

    button_options = FRAME_RAISED | FRAME_THICK |
                      LAYOUT_CENTER_X | LAYOUT_CENTER_Y
    nonmodal_btn = FXButton.new(layout, "&Non-Modal Dialog...", nil,
                                nil, 0, button_options)
    nonmodal_btn.connect(SEL_COMMAND) do
      @non_modal_dialog.show(PLACEMENT_OWNER)
    end

    modal_btn = FXButton.new(layout, "&Modal Dialog...", nil,
                              nil, 0, button_options)
    modal_btn.connect(SEL_COMMAND) do
      dialog = ModalDialogBox.new(self)
      if dialog.execute(PLACEMENT_OWNER) == 1
        puts dialog.text
      end
    end

    getApp.addTimeout(1000, method(:onTimer))
    @non_modal_dialog = NonModalDialogBox.new(self)
  end

  def onTimer(sender, sel, ptr)
    text = @non_modal_dialog.text
    unless text == @previous
      @previous = text
      puts @previous
    end
    getApp.addTimeout(1000, method(:onTimer))
  end

  def create
    super
    show(PLACEMENT_SCREEN)
  end

  application = FXApp.new
  DialogTestWindow.new(application)
  application.create
  application.run

  Перед началом длинного вычисления в FXRuby следует заменить текущий курсор курсором ожидания, а по завершении восстановить исходный. В классе FXApp есть два удобных метода, позволяющих изменить курсор без явного запоминания предыдущего: beginWaitCursor и endWaitCursor. Если метод beginWaitCursor вызывается в блоке, то по выходе из блока будет автоматически вызван метод endWaitCursor:
  getApp.beginWaitCursor do
    # Выполнить длительную операцию...
  end

```

12.3.6. Дополнительные замечания

Библиотека FOX располагает еще многими виджетами и возможностями, например: деревья, стыкуемые панели инструментов, всплывающие подсказки, строки состояния и страницы с вкладками. К числу более сложных средств следует отнести перетаскивание между приложениями и получатели данных, упрощающие связывание данных с виджетами. В библиотеке имеются также неграфические средства для поддержки кросс-платформенного программирования, в том числе класс `FXRegistry`.

Для связи между приложением и его окружением можно использовать сигналы, а также различные каналы ввода и вывода, которые транслируются в сообщения, посылаемые объектам FOX.

Имеются виджеты, поддерживающие наиболее распространенные графические форматы, а также API для работы с библиотекой OpenGL. Это не просто дань вежливости трехмерной графике: на базе библиотеки FOX C++ было реализовано немало инженерных приложений.

Учитывая все вышесказанное, библиотеку `FXRuby` можно считать мощным и гибким инструментом. В последние несколько лет она приобрела популярность в сообществе пользователей Ruby; ожидается, что число поклонников будет расти и дальше. Возможности библиотеки быстро изменяются и расширяются, самую актуальную информацию о привязках к Ruby можно найти на сайте <http://fxruby.org>.

12.4. QtRuby

Qt – это библиотека и комплект средств разработки, созданные и распространяемые компанией Trolltech. Основной упор в *Qt* сделан на кросс-платформенности, единый программный интерфейс предоставляется для операционных систем Windows, Mac, и UNIX. Разработчику нужно написать код только один раз, он будет оттранслирован на всех трех платформах без модификации.

Qt распространяется на условиях одной из двух лицензий: GPL или коммерческая лицензия для разработки продуктов без раскрытия исходных текстов. Такой же политики двойного лицензирования придерживаются и другие компании, например MySQL. Она позволяет использовать библиотеку в проектах с открытыми исходными текстами, в которых предлагаемые средства находят полезное применение. Но при этом Trolltech может получать доход от продажи коммерческих лицензий клиентам, которых не устраивают ограничения GPL.

12.4.1. Обзор

Привязки *QtRuby* – результат работы многих людей, прежде всего Ричарда Дейла (Richard Dale). Эшли Уинтерс (Ashley Winters), Жермен Гаран (Germain Garand) и Давид Форе (David Faure) написали большую часть инструмента генерации кода привязки (он называется SMOKE). Другие отправляли отчеты о найденных ошибках и вносили исправления.

Расширение *QtRuby* содержит не только обширный набор относящихся к графическим интерфейсам классов, но и целый комплект дополнительных средств.

часто необходимых программистам (например, библиотеки для работы с XML и SQL).

В последние несколько лет привязки QtRuby основывались на версии Qt 3.x. В конце 2005 года вышла версия 4. Сейчас есть варианты QtRuby и для Qt3, и для Qt4, но это разные пакеты. Поскольку Qt3 никогда не поставлялась в исходных текстах для Windows, то в этой книге мы рассматриваем только привязки к Qt4. Однако приведенные в этом разделе примеры будут работать и для Qt3. Весь код был проверен на платформах Windows, Linux и Mac с версией QtRuby для Qt4.

Ключевой аспект Qt, а значит и QtRuby, – концепция сигналов и слотов. *Сигналы* представляют собой асинхронные события, возникающие, когда в приложении происходит какое-то событие (например, щелчок кнопкой мыши или ввод текста в поле). *Слот* – это просто метод, вызываемый в ответ на возникновение сигнала. Для связывания сигналов со слотами мы будем использовать метод connect.

Чтобы иметь возможность пользоваться сигналами и слотами, а также многими другими возможностями QtRuby, все наши классы будут наследовать классу `Qt::Object`. Более того, классы, используемые в графических интерфейсах, будут наследовать классу `Qt::Widget`, который, в свою очередь, является производным от `Qt::Object`.

12.4.2. Простое оконное приложение

Приложение QtRuby должно в самом начале загрузить библиотеку Qt. QtRuby раскрывает свою функциональность посредством модуля `qt` (следовательно, имена всех классов начинаются с префикса `Qt::`). Имена всех классов в исходной библиотеке Qt начинаются с буквы Q, но при переходе к QtRuby эта буква опускается. Так, например, класс, основанный на `QWidget`, в QtRuby будет называться `Qt::Widget`.

```
require 'Qt'

app = Qt::Application.new(ARGV)
str = Time.now.strftime("Today is %B %d, %Y")
label = Qt::Label.new(str)
label.show
app.exec
```

Рассмотрим этот код подробнее. Вызов `Qt::Application.new` запускает приложение Qt; он инициализирует оконную систему и выполняет подготовительные действия для создания виджетов.

Затем создается объект `Qt::Label` – простейший способ показать текст пользователю. В данном случае текст инициализируется в предыдущей строчке. Следующая строчка говорит метке, что она должна отобразить себя на экране.

На последнем шаге вызов `app.exec` запускает цикл обработки событий. Он не возвращает управления, пока приложение не завершится. Обычно это происходит, когда пользователь нажимает кнопку закрытия окна.

12.4.3. Кнопки

Создание кнопки в QtRuby сводится к созданию экземпляра класса `Qt::PushButton` (см. листинг 12.14 и рис. 12.7). Обычно при нажатии кнопки нужно выполнить некоторое действие. Для этого применяется механизм событий и слотов QtRuby.

Листинг 12.14. Кнопки в QtRuby

```
require 'Qt'

class MyWidget < Qt::Widget
  slots 'buttonClickedSlot()'
  def initialize(parent = nil)
    super(parent)

    setWindowTitle("QtRuby example");
    @lineedit = Qt::LineEdit.new(self)
    @button = Qt::PushButton.new("All Caps!",self)

    connect(@button, SIGNAL('clicked()'),
            self, SLOT('buttonClickedSlot()'))

    box = Qt::HBoxLayout.new
    box.addWidget(Qt::Label.new("Text:"))
    box.addWidget(@lineedit)
    box.addWidget(@button)

    setLayout(box)
  end

  def buttonClickedSlot
    @lineedit.setText(@lineedit.text.upcase)
  end
end

app = Qt::Application.new(ARGV)
widget = MyWidget.new
widget.show
app.exec
```

В этом примере мы создали собственный класс виджета с именем `mywidget`; он наследует классу `Qt::Widget`, являющемуся предком любого нестандартного виджета.



Рис. 12.7. Кнопки в Qt

Перед инициализацией мы подготовили список слотов, которые будут определены в нашем классе. Слоты – это обычные методы класса, но необходимо указать их имена, чтобы во время выполнения QtRuby знала, что мы собираемся использовать их именно в качестве слотов. Метод класса `slots` принимает список строк:

```
slots 'slot1()', 'slot2()'
```

Инициализатор класса принимает аргумент `parent`, он есть почти у всех виджетов в Qt и определяет, какой виджет будет владельцем вновь создаваемого. Значение `nil` означает, что это «виджет верхнего уровня», у которого нет владельца. Концепция «владения», наверное, имеет более понятный смысл в C++; родители владеют своими детьми, то есть при уничтожении или удалении родителя удаляются и все его потомки.

Наш класс создает объект `Qt::LineEdit` для ввода текста и кнопку `Qt::PushButton` с надписью `All Caps!`. В качестве родителя каждому виджету передается `self`. Это означает, что создаваемый экземпляр `MyWidget` «усыновляет» эти виджеты.

Далее мы обращаемся к ключевой части библиотеки Qt – механизму соединения сигналов со слотами. В классе `Qt::PushButton` определен сигнал `clicked`, который испускается при нажатии кнопки. Этот сигнал можно соединить со слотом, в данном случае с методом `buttonClickedSlot`. Имя слота может быть любым, суффикс `slot` мы употребили просто для наглядности.

В самом конце мы создаем экземпляр класса `Qt::VBoxLayout`. При добавлении виджетов в этот контейнер он автоматически изменяет их размеры, так что нам больше не о чем беспокоиться.

12.4.4. Текстовые поля

Как видно из листинга 12.14, в QtRuby есть класс `Qt::LineEdit` для ввода одной строки текста. Для ввода нескольких строк предназначен класс `Qt::TextEdit`.

В листинге 12.15 демонстрируется многострочное текстовое поле. Под ним расположена метка, в которой отображается текущая длина текста (рис. 12.8).

Листинг 12.15. Простой редактор в Qt

```
require 'Qt'

class MyTextWindow < Qt::Widget
  slots 'theTextChanged()'

  def initialize(parent = nil)
    super(parent)

    @textedit = Qt::TextEdit.new(self)
    @textedit.setWordWrapMode(Qt::TextOption::WordWrap)
    @textedit.setFont( Qt::Font.new("Times", 24) )

    @status = Qt::Label.new(self)

    box = Qt::VBoxLayout.new
    box.addWidget(@textedit)
```

```

box.addWidget(@status)
setLayout(box)

@textedit.insertPlainText("This really is an editor")

connect(@textedit, SIGNAL('textChanged()'),
        self, SLOT('theTextChanged()'))
end

def theTextChanged
  text = "Length: " + @textedit.toPlainText.length.to_s
  @status.setText(text)
end
end

app = Qt::Application.new(ARGV)
widget = MyTextWindow.new
widget.setWindowTitle("QtRuby Text Editor")
widget.show
app.exec

```



Рис. 12.8. Простой редактор в Qt

Виджет конструируется примерно так же, как в предыдущем примере. Но теперь мы создаем объект `Qt::TextEdit`, а также метку `Qt::Label` для показа текущего состояния.

Стоит отметить, что для объекта `@textedit` мы указали шрифт Times высотой 24 пункта. У каждого класса, наследующего `Qt::Widget` (в том числе и у `Qt::TextEdit`) есть свойство `font`, которое можно опросить или установить.

Затем мы создаем менеджер вертикального размещения (`Qt::VBoxLayout`), который будет контейнером для всех своих потомков, добавляем в него виджет `@textedit` и связываем сигнал `textChanged` с определенным нами слотом `theTextChanged`.

В методе `theTextChanged` мы запрашиваем у редактора текст и получаем его длину, а затем записываем возвращенное значение в метку `@status`.

Отметим, что весь механизм сигналов и слотов работает асинхронно. После того как приложение входит в цикл обработки событий (`app.exec`), оно уже не

получает управления явно. Вот почему сигналы и слоты так важны. Мы определяем события, которые нас интересуют (сигналы), и действия, которые нужно выполнить при возникновении таких событий (слоты).

12.4.5. Прочие виджеты

В библиотеке Qt есть еще много встроенных виджетов, например переключатели, флажки и т. п. В листинге 12.16 продемонстрированы некоторые из них, а на рис. 12.9 показано, как выглядит окно приложения.

Листинг 12.16. Прочие виджеты в Qt

```
require 'Qt'

class MyWindow < Qt::Widget
  slots 'somethingClicked(QAbstractButton *)'
  def initialize(parent = nil)
    super(parent)

    groupbox = Qt::GroupBox.new("Some Radio Button",self)

    radio1 = Qt::RadioButton.new("Radio Button 1", groupbox)
    radio2 = Qt::RadioButton.new("Radio Button 2", groupbox)
    check1 = Qt::CheckBox.new("Check Box 1", groupbox)

    vbox = Qt::VBoxLayout.new
    vbox.addWidget(radio1)
    vbox.addWidget(radio2)
    vbox.addWidget(check1)
    groupbox.setLayout(vbox)

    bg = Qt::ButtonGroup.new(self)
    bg.addButton(radio1)
    bg.addButton(radio2)
    bg.addButton(check1)

    connect(bg, SIGNAL('buttonClicked(QAbstractButton *)'),
            self, SLOT('somethingClicked(QAbstractButton *)') )

    @label = Qt::Label.new(self)
```

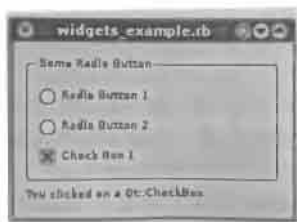


Рис. 12.9. Простое приложение Tk

```

vbox = Qt::VBoxLayout.new
vbox.addWidget(groupbox)
vbox.addWidget(@label)
setLayout(vbox)
end

def somethingClicked(who)
  @label.setText("You clicked on a " + who.className)
end

end

app = Qt::Application.new(ARGV)
widget = MyWindow.new
widget.show
app.exec

```

В этом классе мы сначала создаем объект `Qt::GroupBox` – контейнер с рамкой и необязательным заголовком, в который можно помещать другие виджеты. Далее создаются два переключателя `Qt::RadioButtons` и флажок `Qt::CheckBox`, а в качестве их родителя указывается ранее созданный контейнер.

Затем создается менеджер размещения `Qt::VBoxLayout`, в который помещаются переключатели и флажок, после чего этот менеджер связывается с групповым контейнером и начинает управлять его размещением на экране.

Следующий важный шаг – создание объекта `Qt::ButtonGroup`, в который помещаются флажок и переключатели. `Qt::ButtonGroup` предназначен для логической группировки кнопок, флажков и переключателей. На их визуальное расположение он никак не влияет, зато обеспечивает, к примеру, взаимное исключение (гарантирует, что только один из группы виджетов может быть отмечен). В данном случае этот объект будет источником сигнала `buttonClicked`, который испускается при нажатии любой кнопки в группе.

Этот сигнал отличается от виденных ранее тем, что ему сопутствует аргумент, а именно объект, по которому щелкнули мышкой. Обратите внимание на то, как синтаксис – `QAbstractButton *` – напоминает о C++-ных корнях Qt. В некоторых случаях употребления принятой в C++ нотации для обозначения типов параметров не избежать (хотя в будущих версиях это, возможно, и исправят).

В результате такого вызова метода `connect` при щелчке по любому виджету, принадлежащему группе, этот виджет будет передан слоту `somethingClicked`. Наконец, мы создаем метку `Qt::Label`, контейнер `Qt::VBoxLayout` и увязываем все вместе.

Внутри слота `somethingClicked` мы модифицируем текст метки при щелчке по любому переключателю или флажку. В данном случае выводится имя класса объекта, который испустил сигнал, приведший к вызову слота.

Если встроенных виджетов недостаточно, то Qt предоставляет мощную систему рисования для создания собственных. В листинге 12.17 приведен небольшой пример, иллюстрирующий малую часть возможностей.

Листинг 12.17. Нестандартный виджет TimerClock

```
require 'Qt'

class TimerClock < Qt::Widget

  def initialize(parent = nil)
    super(parent)

    @timer = Qt::Timer.new(self)
    connect(@timer, SIGNAL('timeout()'), self, SLOT('update()'))
    @timer.start(25)

    setWindowTitle('Stop Watch')
    resize(200, 200)
  end

  def paintEvent(e)
    fastHand = Qt::Polygon.new([Qt::Point.new(7, 8),
                                Qt::Point.new(-7, 8),
                                Qt::Point.new(0, -80)])

    secondHand = Qt::Polygon.new([Qt::Point.new(7, 8),
                                   Qt::Point.new(-7, 8),
                                   Qt::Point.new(0, -65)])

    secondColor = Qt::Color.new(100, 0, 100)
    fastColor = Qt::Color.new(0, 150, 150, 150)

    side = [width, height].min
    time = Qt::Time.currentTime

    painter = Qt::Painter.new(self)
    painter.renderHint = Qt::Painter::Antialiasing
    painter.translate(width() / 2, height() / 2)
    painter.scale(side / 200.0, side / 200.0)

    painter.pen = Qt::NoPen
    painter.brush = Qt::Brush.new(secondColor)

    painter.save
    painter.rotate(6.0 * time.second)
    painter.drawConvexPolygon(secondHand)
    painter.restore

    painter.pen = secondColor
    (0...12).each do |i|
      painter.drawLine(88, 0, 96, 0)
      painter.rotate(30.0)
    end
  end
end
```

```

painter.pen = Qt::NoPen
painter.brush = Qt::Brush.new(fastColor)

painter.save
painter.rotate(36.0 * (time.msec / 100.0) )
painter.drawConvexPolygon(fastHand)
painter.restore

painter.pen = fastColor
(0...60).each do |j|
  if (j % 5) != 0
    painter.drawLine(92, 0, 96, 0)
  end
  painter.rotate(6.0)
end

painter.end
end
end
app = Qt::Application.new(ARGV)
wid = TimerClock.new
wid.show
app.exec

```

Созданный в этом примере виджет называется `timerClock`. В инициализаторе мы создаем объект `Qt::Timer`, который конфигурируется для периодического испускания сигнала. Его сигнал `timeout` мы соединяем со слотом `update` нашего виджета. Это встроенный слот, он заставляет виджет перерисовать себя.

Таймер запускается методом `start`. Переданный ему аргумент говорит, что таймер должен срабатывать (и испускать сигнал `timeout`) каждые 25 миллисекунд. Следовательно, слот `update` будет вызываться каждые 25 миллисекунд.

Далее определяется метод `paintEvent`. Мы переопределяем одноименный метод класса `Qt::Widget`. Когда виджет собирается перерисовать себя (то есть при срабатывании таймера), он вызывает этот метод. Переопределяя его, мы решаем, как виджет должен отображаться на экране. Код этого метода вызывает различные графические примитивы рисования.

Начиная с этого места идет сплошная геометрия. Мы создаем несколько многоугольников `Qt::Polygon`, представляющих стрелки часов. Ориентация многоугольников не имеет значения, потому что манипулировать ими мы будем позже.

Задаются значения нескольких свойств. Устанавливаются цвета `Qt::Color` обеих стрелок. Аргументами инициализатора `Qt::Color` являются значения в формате RGB с необязательной альфа-прозрачностью.

Часы должны быть квадратными, поэтому в переменную `side` (длина стороны) записывается минимум из ширины и высоты виджета. Кроме того, мы запоминаем текущее время, обращаясь к методу `Qt::Time.currentTime`.

Далее создается объект `Qt::Painter`, и с его помощью мы начинаем рисовать. Задается режим сглаживания (`antialiasing`), чтобы на стрелках часов не было

«лесенки». Начало координат помещается в центр области рисования (`painter.translate(width/2, height/2)`). Для объекта `Painter` устанавливается масштаб в предположении, что сторона квадрата составляет 200 единиц. Если размер окна изменится, то масштабирование будет произведено автоматически.

Затем выполняется последовательность операций рисования. Различные геометрические преобразования (например, поворот), сопровождаются парой вызовов `painter.save` и `painter.restore`. Метод `save` сохраняет текущие свойства объекта `Painter` в стеке, чтобы их можно было позднее восстановить.

Программа рисует обе стрелки, предварительно повернув их на нужный угол в соответствии с текущим временем. Кроме того, мы наносим риски вдоль границы циферблата.

И напоследок мы сообщаем объекту `Painter`, что рисование закончилось (вызывая метод `painter.end`). Довершают картину четыре строчки, в которых создаются объект приложения `Qt::Application` и наш виджет, а затем запускается цикл обработки событий. На рис. 12.10 показан конечный результат.



Рис. 12.10. Виджет `TimerClock`

12.4.6. Дополнительные замечания

Поскольку библиотека `Qt` написана на `C++`, неизбежны некоторые идиомы, отражающие ограничения этого языка. Иногда перевод на `Ruby` не выглядит на 100% естественным, поскольку в `Ruby` аналогичные вещи делаются несколько иначе. Поэтому в ряде случаев вводится избыточность, позволяющая выражать свои намерения «по-рублистски».

Например, имена методов, в которых сохранена «верблужья Нотация», свойственная `C++`, можно записывать и с подчеркиками (`_`). Так, следующие два вызова эквивалентны:

```
Qt::Widget::minimumSizeHint
Qt::Widget::minimum_size_hint
```

Все методы установки свойств в `Qt` начинаются со слова `set`, например, `Qt::Widget::setMinimumSize`. В `Ruby` можно это слово опускать и пользоваться присваиванием, например:

```
widget.setMinimumSize(50)
widget.minimumSize = 50      # То же самое.
widget.minimum_size = 50     # То же самое.
```


Аналогично в Qt имена методов, возвращающих булевское значение, часто начинаются с `is` или `has`, например, `Qt::Widget::isVisible`. QtRuby позволяет именовать их в духе Ruby:

```
a.isVisible
a.visible?      # То же самое.
```

12.5. Другие библиотеки для создания графических интерфейсов

Мы предупреждали, что вы можете и не найти своего любимого графического интерфейса. Но напоследок кратко упомянем имеющиеся альтернативы.

Некоторые из упоминаемых продуктов еще не вполне зрелые, могут быть неполными и содержать ошибки. Но мы полагаем, что этот список будет расти, а подерживаемые привязки со временем обретут стабильность.

12.5.1. Ruby и X

Систему X Window System в разговорной речи называют (не совсем корректно) просто X Windows. Вероятно, она является прародителем если не всех, то абсолютного большинства графических интерфейсов пользователя.

Пользователи всех вариантов UNIX давно уже знакомы с X (как пользователи, а то и как разработчики). Часто поверх X запускается оконный менеджер Motif.

К достоинствам X следует отнести широкую известность, переносимость и богатый набор возможностей. К недостаткам – сложность работы.

Неудивительно, что существуют библиотеки для работы с X из Ruby. Из-за их сложности мы не приводим документацию, а отсылаем вас к архиву приложений Ruby RAA, где вы найдете библиотеку Xlib, написанную Кадзухиро Иосида (Kazuhiro Yoshida, известный также как moriq), и Ruby/X11 Мэтью Бушара (Mathieu Bouchard, он же matju). Обе годятся для создания X-клиентов.

12.5.2. Ruby и wxWidgets

Система wxWidgets (прежнее название wxWindows) функционально богата и стабильна. Они широко применяется в мире Python и по существу является «родным» графическим интерфейсом для этого языка. Философия библиотеки – пользоваться платформенными виджетами, когда это возможно. Версия для UNIX более зрелая, чем для Windows, но это положение, конечно, меняется.

В данный момент существует достаточно зрелая библиотека wxRuby. Если вам нравится именно эта система, то можете найти ее вместе с документацией на сайте <http://wxruby.rubyforge.org/>.

12.5.3. Apollo (Ruby и Delphi)

Настоящий хакер знает, что для серьезного программирования чистый Pascal бесполезен. Но на протяжении многих лет предпринималось немало попыток сделать этот язык пригодным для практического применения. Одна из самых успешных – Object Pascal компании Borland, ставший основой среды быстрой разработки Delphi.

Своей популярностью Delphi обязана не расширению языка Pascal, хотя это тоже играет свою роль, но самой среде и богатству графического интерфейса. Delphi предлагает множество виджетов для создания стабильных, привлекательных графических приложений на платформе MS Windows.

Библиотека Apollo – попытка «поженить» Ruby и Delphi. Это детище Кадзухи-ро Иосида, хотя свой вклад внесли и многие другие. Основное достоинство Apollo – гигантский набор стабильных, удобных виджетов, а основной недостаток заключается в том, что на сегодняшний день она требует слегка «подправленной» версии Ruby. Она должна работать и с «классическим» продуктом Borland Kylix, который, по существу, является версией Delphi для Linux. Дополнительную информацию ищите в архиве RAA.

12.5.4. Ruby и Windows API

В главе 8 мы рассматривали вариант «графического интерфейса для бедных», когда для доступа к возможностям браузера Internet Explorer и другим подобным вещам используется библиотека win32ole. С деталями можно ознакомиться по приведенным там примерам. Если вам нужно что-то сделать быстро, не заботясь об эlegantности, то такое решение может оказаться приемлемым.

Если у вас есть склонность к мазохизму, то можете работать с Windows API напрямую. В этом вам поможет библиотека win32api (она обсуждается также в главе 14). Лично я не рекомендую такой подход, но о его существовании вы должны знать.

12.6. Заключение

В этой главе был представлен обзор различных средств разработки графических интерфейсов пользователя для Ruby. Мы познакомились с общими концепциями: циклами обработки событий, сообщениями или сигналами и т. п. Была продемонстрирована работа с различными виджетами: кнопками, флажками, переключателями, текстовыми полями и т. д.

Мы рассмотрели реализацию общих концепций на примере библиотек Tk, GTK, FOX и Qt. Выяснили, что в каждом случае применяется своя терминология и слегка отличающиеся варианты основной парадигмы. Отметим также специфические средства и достоинства, присущие каждой библиотеке.

А теперь перейдем к совсем другой теме. В главе 13 будет рассмотрена работа с потоками в Ruby.

Глава 13. Потоки в Ruby

.....
*Он тянет нить своего красноречия искуснее, чем развивает свои доводы.
Шекспир, «Бесплодные усилия любви», акт V, сцена 1**

Потоки еще иногда называют *облегченными процессами*. Это просто способ обеспечить параллельное выполнение без накладных расходов, связанных с контекстным переключением между процессами. (Впрочем, общего согласия по поводу того, что такое поток, нет, поэтому мы не будем углубляться в данный вопрос.)

В Ruby потоки определены на пользовательском уровне и не зависят от операционной системы. Они работают в DOS так же, как и в UNIX. Но, конечно, это снижает производительность, а на сколько именно, зависит от операционной системы.

Потоки полезны, например, тогда, когда некоторые части программы могут работать независимо друг от друга. Применяются они и в тех случаях, когда приложение тратит много времени на ожидание события. Часто, пока один поток ждет, другой может выполнять полезную работу.

С другой стороны, у потоков есть и недостатки. Всегда надо взвешивать, оправдано ли их применение в конкретном случае. К тому же, иногда доступ к ресурсу принципиально должен осуществляться строго последовательно, поэтому потоки не дадут никакого выигрыша. И, наконец, бывает так, что накладные расходы на синхронизацию доступа к глобальным ресурсам превышают экономию, достигаемую за счет использования нескольких потоков.

По этой и ряду других причин некоторые авторитеты вообще рекомендуют держаться подальше от многопоточного программирования. Действительно, такие программы сложны и подвержены ошибкам, которые трудно отлаживать. Но мы оставим читателю самому решать, когда стоит применять эту технику.

Проблемы, связанные с несинхронизированными потоками, хорошо известны. При одновременном доступе к глобальным данным со стороны нескольких потоков данные могут быть запорчены. Если один поток делает какое-то допущение о том, что успел выполнить другой поток, возможна гонка (*race condition*); обычно это приводит к «недетерминированному» коду, который дает разные результаты при каждом запуске. Наконец, существует опасность тупиковой ситуации, когда ни один поток не может продолжить выполнение, поскольку ожидает ресурс, занятый другим потоком. Код, написанный так, что ни одна из этих проблем не возникает, называется безопасным относительно потоков.

Не все в Ruby безопасно относительно потоков, но имеются методы синхронизации, которые позволяют контролировать доступ к переменным и ресурсам.

* Пер. М. Кузмина. – Прим. ред.

защищать критические секции программы и избегать тупиковых ситуаций. Мы рассмотрим их в этой главе и проиллюстрируем на примерах.

13.1. Создание потоков и манипулирование ими

К числу основных операций над потоками относятся создание потока, передача ему входной информации и получение результатов, останов потока и т. д. Можно получить список запущенных потоков, опросить состояние потока и выполнить ряд других проверок.

Ниже представлен обзор основных операций.

13.1.1. Создание потоков

Создать поток просто: достаточно вызвать метод `new` и присоединить блок, который будет исполняться в потоке.

```
thread = Thread.new do
  # Предложения, исполняемые в потоке...
end
```

Возвращаемое значение – объект типа `Thread`. Главный поток программы может использовать его для управления вновь созданным потоком.

А если нужно передать потоку параметры? Достаточно передать их методу `Thread.new`, который, в свою очередь, передаст их блоку.

```
a = 4
b = 5
c = 6
thread2 = Thread.new(a,b,c) do |a, x, y|
  # Манипуляции с a, x и y.
end
```

```
# Если переменная a будет изменена новым потоком,
# то главный поток не получит об этом никакого уведомления.
```

Параметры блока, являющиеся ссылками на существующие переменные, практически неотличимы от самих переменных. Поэтому, например, переменная `a` в каком-то смысле «опасна», что и отражено в комментарии.

Поток может также обращаться к переменным из той же области видимости, в которой был создан. Ясно, что без синхронизации это может стать источником проблем. Главный и любой другой поток могут изменять такую переменную независимо друг от друга, и результаты подобных действий непредсказуемы.

```
x = 1
y = 2
thread3 = Thread.new do
  # Этот поток может манипулировать переменными x and y
  # из внешней области видимости, но это не всегда безопасно.
  sleep(rand(0)) # Спать в течение случайно выбранного времени
                  # (меньше секунды).

  x = 3
end
```

```
sleep(rand(0))
puts x
# Если запустить эту программу несколько раз подряд, то может быть
# напечатано как 1, так и 3!
```

У метода `new` есть синоним `fork` — это имя выбрано по аналогии с хорошо известным системным вызовом в UNIX.

13.1.2. Доступ к локальным переменным потока

Мы знаем об опасности доступа из потока к переменным, определенным вне его области видимости, но мы также знаем, что у потока могут быть локальные данные. А что делать, если поток хочет «обнародовать» часть принадлежащих ему данных?

Для этой цели предусмотрен специальный механизм. Если объект `Thread` рассматривать как хэш, то к локальным данным потока можно обратиться из любого места в области видимости этого объекта. Мы не хотим сказать, что так можно обратиться к настоящим локальным переменным; это допустимо лишь для доступа к именованным данным, своим для каждого потока.

Существует также метод `key?`, который сообщает, используется ли указанное имя в данном потоке.

Внутри потока к таким данным тоже следует обращаться, как к хэшу. Метод `Thread.current` позволяет сделать запись чуть менее громоздкой.

```
thread = Thread.new do
  t = Thread.current
  t[:var1] = "Это строка"
  t[:var2] = 365
end

# Доступ к локальным данным потока извне...

x = thread[:var1]          # "Это строка"
y = thread[:var2]          # 365

has_var2 = thread.key?("var2") # true
has_var3 = thread.key?("var3") # false
```

Отметим, что эти данные доступны другим потокам даже после того, их владелец завершил работу (как в данном случае).

Помимо символа (см. выше), для идентификации локальной переменной потока можно употреблять и строки.

```
thread = Thread.new do
  t = Thread.current
  t["var3"] = 25
  t[:var4] = "foobar"
end

a = thread[:var3] = 25
b = thread["var4"] = "foobar"
```

Не путайте эти специальные имена с настоящими локальными переменными. В следующем фрагменте разница видна более отчетливо:

```
thread = Thread.new do
  t = Thread.current
  t["var3"] = 25
  t[:var4] = "foobar"
  var3 = 99          # Настоящие локальные переменные
  var4 = "zorch"     # (извне недоступны)
end
```

```
a = thread[:var3]    # 25
b = thread["var4"]   # "foobar"
```

И еще отметим, что ссылку на объект (на настоящую локальную переменную) внутри потока можно использовать для сокращенной записи. Это справедливо, если вы сохраняете одну и ту же ссылку, а не создаете новую.

```
thread = Thread.new do
  t = Thread.current
  x = "nXxeQPdMdxIBAxh"
  t[:my_message] = x
  x.reverse!
  x.delete! "x"
  x.gsub!(/[A-Z]/, "")
  # С другой стороны, присваивание создает новый объект,
  # поэтому "сокращение" становится бесполезным...
end
a = thread[:my_message] # "hidden"
```

Ясно, что сокращение не будет работать и в том случае, когда вы имеете дело с объектами наподобие `Fixnum`, которые хранятся как непосредственные значения, а не ссылки.

13.1.3. Опрос и изменение состояния потока

В классе `Thread` есть несколько полезных методов класса. Метод `list` возвращает массив «живых» потоков, метод `main` возвращает ссылку на главный поток программы, который породил все остальные, а метод `current` позволяет потоку идентифицировать самого себя.

```
t1 = Thread.new { sleep 100 }
t2 = Thread.new do
  if Thread.current == Thread.main
    puts "Это главный поток."      # НЕ печатается.
  end
  1.upto(1000) { sleep 0.1 }
end

count = Thread.list.size          # 3
if Thread.list.include?(Thread.main)
  puts "Главный поток жив."      # Печатается всегда!
end
if Thread.current == Thread.main
  puts "Я главный поток."        # Здесь печатается...
end
```

Методы `exit`, `pass`, `start`, `stop` и `kill` служат для управления выполнением потоков (как изнутри, так и извне):

```
# В главном потоке...
Thread.kill(t1)           # Завершить этот поток.
Thread.pass               # Передать управление t2.
t3 = Thread.new do
  sleep 20
  Thread.exit              # Выйти из потока.
  puts "Так не бывает!"   # Никогда не выполняется.
end
Thread.kill(t2)           # Завершить t2.
# Выйти из главного потока (все остальные тоже завершаются).
Thread.exit
```

Отметим, что не существует метода экземпляра `stop`, поэтому поток может приостановить собственное выполнение, но не выполнение другого потока.

Существуют различные методы для опроса состояния потока. Метод экземпляра `alive?` сообщает, является ли данный поток «живым» (не завершил выполнение), а метод `stop?` – находится ли он в состоянии «приостановлен».

```
count = 0
t1 = Thread.new { loop { count += 1 } }
t2 = Thread.new { Thread.stop }
sleep 1
flags = [t1.alive?,      # true
         t1.stop?,       # false
         t2.alive?,      # true
         t2.stop?]       # true
```

Получить состояние потока позволяет метод `status`. Он возвращает значение `"run"`, если поток выполняется; `"sleep"` – если он приостановлен, спит или ожидает результата ввода/вывода; `false` – если поток нормально завершился, и `nil` – если поток завершился в результате исключения.

```
t1 = Thread.new { loop {} }
t2 = Thread.new { sleep 5 }
t3 = Thread.new { Thread.stop }
t4 = Thread.new { Thread.exit }
t5 = Thread.new { raise "exception" }
s1 = t1.status   # "run"
s2 = t2.status   # "sleep"
s3 = t3.status   # "sleep"
s4 = t4.status   # false
s5 = t5.status   # nil
```

Глобальную переменную `$SAFE` можно установить по-разному в разных потоках. Стало быть, она вовсе не является глобальной, но стоит ли жаловаться на это, если она позволяет разным потокам работать с разным уровнем безопасности? Метод `safe_level` возвращает текущий уровень безопасности потока.

```
t1 = Thread.new { $SAFE = 1; sleep 5 }
t2 = Thread.new { $SAFE = 3; sleep 5 }
```

```
sleep 1
lev0 = Thread.main.safe_level # 0
lev1 = t1.safe_level          # 1
lev2 = t2.safe_level          # 3
```

Метод доступа `priority` позволяет узнать и изменить приоритет потока:

```
t1 = Thread.new { loop { sleep 1 } }
t2 = Thread.new { loop { sleep 1 } }
t2.priority = 3 # Установить для потока t2 приоритет 3
p1 = t1.priority # 0
p2 = t2.priority # 3
```

Поток с большим приоритетом будет чаще получать процессорное время.

Специальный метод `pass` позволяет передать управление планировщику. Иными словами, поток просто уступает свой временной квант, но не приостанавливается и не засыпает.

```
t1 = Thread.new do
  puts "alpha"
  Thread.pass
  puts "beta"
end

t2 = Thread.new do
  puts "gamma"
  puts "delta"
end

t1.join
t2.join
```

В этом искусственном примере вызов `Thread.pass` приводит к печати строк в следующем порядке: alpha gamma delta beta. Без него было бы напечатано alpha beta gamma delta. Конечно, этот механизм следует использовать не для синхронизации, а только для экономного расходования процессорного времени.

Выполнение приостановленного потока можно возобновить методами методами `run` или `wakeup`:

```
t1 = Thread.new do
  Thread.stop
  puts "Здесь есть изумруд."
end

t2 = Thread.new do
  Thread.stop
  puts "Вы находитесь в точке Y2."
end

sleep 1
t1.wakeup
t2.run
```

Между этими методами есть тонкое различие. Метод `wakeup` изменяет состояние потока, так что он становится готовым к выполнению, но не запускает его немедленно. Метод же `run` пробуждает поток и сразу же планирует его выполнение.

В данном случае `t1` просыпается раньше `t2`, но `t2` планируется первым, что приводит к следующему результату:

Вы находитесь в точке Y2.

Здесь есть изумруд.

Конечно, было бы неосмотрительно реализовывать синхронизацию на основе этого механизма.

Метод экземпляра `raise` возбуждает исключение в потоке, от имени которого вызван. (Этот метод необязательно вызывать в том потоке, которому адресовано исключение.)

```
factorial1000 = Thread.new do
  begin
    prod = 1
    1.upto(1000) { |n| prod *= n }
    puts "1000! = #{prod}"
  rescue
    # Ничего не делать...
  end
end

sleep 0.01 # На вашей машине значение может быть иным.
if factorial1000.alive?
  factorial1000.raise("Стоп!")
  puts "Вычисление было прервано!"
else
  puts "Вычисление успешно завершено."
end
```

Поток, запущенный в предыдущем примере, пытался вычислить факториал 1000. Если для этого не хватило одной сотой секунды, то главный поток завершит его. Как следствие, на относительно медленной машине будет напечатано сообщение «Вычисление было прервано!» Что касается части `rescue` внутри потока, то в ней мог бы находиться любой код, как, впрочем, и всегда.

13.1.4. Назначение randеву (и получение возвращенного значения)

Иногда главный поток хочет дождаться завершения другого потока. Для этой цели предназначен метод `join`:

```
t1 = Thread.new { do_something_long() }

do_something_brief()
t1.join # Ждать завершения t1.
```

Отметим, что вызывать метод `join` необходимо, если нужно дождаться завершения другого потока. В противном случае главный поток завершится, а вместе с ним и все остальные. Например, следующий код никогда не напечатал бы окончательный ответ, не будь в конце вызова `join`:

```
meaning_of_life = Thread.new do
  puts "Смысл жизни заключается в..."
  sleep 10
```

```
puts 42
end

sleep 9
meaning_of_life.join
```

Существует полезная идиома, позволяющая вызвать метод `join` для всех «живых» потоков, кроме главного (ни один поток, даже главный, не может вызывать `join` для самого себя).

```
Thread.list.each { |t| t.join if t != Thread.main }
```

Конечно, любой поток, а не только главный, может вызвать `join` для любого другого потока. Если главный поток и какой-то другой попытаются вызвать `join` друг для друга, возникнет тупиковая ситуация. Интерпретатор обнаружит это и завершит программу.

```
thr = Thread.new { sleep 1; Thread.main.join }
```

```
thr.join          # Тупиковая ситуация!
```

С потоком связан блок, который может возвращать значение. Следовательно, и сам поток может возвращать значение. Метод `value` неявно вызывает `join` и ждет, пока указанный поток завершится, а потом возвращает значение последнего вычисленного в потоке выражения.

```
max = 10000
thr = Thread.new do
  sum = 0
  1.upto(max) { |i| sum += i }
  sum
end
```

```
guess = (max*(max+1))/2
print "Формула "
if guess == thr.value
  puts "правильна."
else
  puts "неправильна."
end
```

13.1.5. Обработка исключений

Что произойдет, если в потоке возникнет исключение? Как выясняется, поведение можно сконфигурировать заранее.

Существует флаг `abort_on_exception`, который работает как на уровне класса, так и на уровне экземпляра. Он реализован в виде метода доступа (то есть позволяет читать и устанавливать атрибут) на обоих уровнях. Если `abort_on_exception` для некоторого потока равен `true`, то при возникновении в этом потоке исключения будут завершены и все остальные потоки.

```
Thread.abort_on_exception = true
```

```
t1 = Thread.new do
```

```
puts "Привет!"
sleep 2
raise "some exception"
puts "Пока!"
end

t2 = Thread.new { sleep 100 }

sleep 2
puts "Конец"
```

В этом примере флаг `abort_on_exception` установлен в `true` на уровне системы в целом (отменяя подразумеваемое по умолчанию значение). Следовательно, когда в потоке `t1` возникает исключение, завершаются и `t1`, и главный поток. Печатаются только слово «Привет!».

В следующем примере эффект такой же:

```
t1 = Thread.new do
  puts "Привет!"
  sleep 2
  raise "some exception"
  puts "Пока!"
end

t1.abort_on_exception = true

t2 = Thread.new { sleep 100 }

sleep 2
puts "Конец"
```

А вот в следующем оставлено принимаемое по умолчанию значение `false`, и мы наконец-то видим слово «Конец», печатаемое главным потоком (слова «Пока!» мы не увидим никогда, поскольку поток `t1` при возникновении исключения завершается безусловно).

```
t1 = Thread.new do
  puts "Привет!"
  sleep 2
  raise "some exception"
  puts "Пока!"
end

t2 = Thread.new { sleep 100 }

sleep 2
puts "Конец"

# Выводится:
Привет!
Конец
```

13.1.6. Группы потоков

Группа потоков – это механизм управления логически связанными потоками. По умолчанию все потоки принадлежат группе `Default` (это константа класса). Но если создать новую группу, то в нее можно будет помещать потоки.

В любой момент времени поток может принадлежать только одной группе. Если поток помещается в группу, то он автоматически удаляется из той группы, которой принадлежал ранее.

Метод класса `ThreadGroup.new` создает новую группу потоков, а метод экземпляра `add` помещает поток в группу.

```
f1 = Thread.new("file1") { |file| waitfor(file) }
f2 = Thread.new("file2") { |file| waitfor(file) }
```

```
file_threads = ThreadGroup.new
file_threads.add f1
file_threads.add f2
```

Метод экземпляра `list` возвращает массив всех потоков, принадлежащих данной группе.

```
# Подсчитать все "живые" потоки в группе this_group.
count = 0
this_group.list.each { |x| count += 1 if x.alive? }
if count < this_group.list.size
  puts "Некоторые потоки в группе this_group уже скончались."
else
  puts "Все потоки в группе this_group живы."
end
```

В класс `ThreadGroup` можно добавить немало полезных методов. В примере ниже показаны методы для возобновления всех потоков, принадлежащих группе, для группового ожидания потоков (с помощью `join`) и для группового завершения потоков:

```
class ThreadGroup

  def wakeup
    list.each { |t| t.wakeup }
  end

  def join
    list.each { |t| t.join if t != Thread.current }
  end

  def kill
    list.each { |t| t.kill }
  end

end
```

13.2. Синхронизация потоков

Почему необходима синхронизация? Потому что из-за «чередования» операций доступ к переменным и другим сущностям может осуществляться в порядке, который не удастся установить путем чтения исходного текста отдельных потоков. Два и более потоков, обращающихся к одной и той же переменной, могут взаимодействовать между собой непредвиденными способами, и отлаживать такую программу очень трудно.

Рассмотрим простой пример:

```
x = 0
t1 = Thread.new do
  1.upto(1000) do
    x = x + 1
  end
end

t2 = Thread.new do
  1.upto(1000) do
    x = x + 1
  end
end

t1.join
t2.join
puts x
```

Сначала переменная `x` равна 0. Каждый поток увеличивает ее значение на тысячу раз. Логика подсказывает, что в конце должно быть напечатано 2000.

Но фактический результат противоречит логике. На конкретной машине было напечатано значение 1044. В чем дело?

Мы предполагали, что инкремент целого числа – атомарная (неделимая) операция. Но это не так. Рассмотрим последовательность выполнения приведенной выше программы. Поместим поток `t1` слева, а поток `t2` справа. Каждый квант времени занимает одну строчку и предполагается, что к моменту, когда был сделан этот мгновенный снимок, переменная `x` имела значение 123.

t1	t2
Прочитать значение <code>x</code> (123)	
Увеличить значение на 1 (124)	Прочитать значение <code>x</code> (123)
Записать результат в <code>x</code>	Увеличить значение на 1 (124)
	Записать результат в <code>x</code>

Ясно, что каждый поток увеличивает на 1 то значение, которое видит. Но не менее ясно и то, что после увеличения на 1 обоими потоками `x` оказалось равно всего 124.

И это лишь самая простая из проблем, возникающих в связи с синхронизацией. Для решения более сложных приходится прилагать серьезные усилия – это предмет изучения специалистами в области теоретической информатики и математики.

13.2.1. Синхронизация с помощью критических секций

Простейший способ синхронизации дают критические секции. Когда поток входит в критическую секцию программы, гарантируется, что никакой другой поток не войдет в нее, пока первый не выйдет.

Если аксессуару `Thread.critical` присвоить значение `true`, то выполнение других потоков не будет планироваться. В следующем примере мы переработали код предыдущего, воспользовавшись аксессуаром `critical` для определения критической области, которая защищает уязвимые участки программы.

```
x = 0
t1 = Thread.new do
  1.upto(1000) do
    Thread.critical = true
    x = x + 1
    Thread.critical = false
  end
end

t2 = Thread.new do
  1.upto(1000) do
    Thread.critical = true
    x = x + 1
    Thread.critical = false
  end
end

t1.join
t2.join
puts x
```

Теперь последовательность выполнения изменилась; взгляните, в каком порядке работают потоки `t1` и `t2`. (Конечно, вне того участка, где происходит увеличение переменной, потоки могут чередоваться более-менее случайным образом.)

t1	t2
Прочитать значение x (123)	
Увеличить значение на 1 (124)	
Записать результат в x	
	Прочитать значение x (124)
	Увеличить значение на 1 (125)
	Записать результат в x

Возможны такие комбинации операций с потоками, при которых поток планируется даже тогда, когда какой-то другой поток находится в критической секции.

Простейший случай – вновь созданный поток начинает исполнение немедленно вне зависимости от того, занимает какой-то другой поток критическую секцию или нет. Поэтому описанную технику лучше применять только в самых простых ситуациях.

13.2.2. Синхронизация доступа к ресурсам (mutex.rb)

В качестве примера рассмотрим задачу индексирования Web-сайтов. Мы извлекаем слова из многочисленных страниц в Сети и сохраняем их в хэше. Ключом является само слово, а значением – строка, идентифицирующая документ и номер строки в этом документе.

Постановка задачи и так достаточно груба. Но мы огрубим ее еще больше, введя следующие упрощающие допущения:

- будем представлять удаленные документы в виде строк;
- ограничимся всего тремя строками (они будут «зашиты» в код);
- сетевые задержки будем моделировать «засыпанием» на случайный промежуток времени.

Взгляните на программу в листинге 13.1. Она даже не печатает получаемые данные целиком, а выводит лишь счетчик слов (не уникальный). Каждый раз при чтении или обновлении хэша мы вызываем метод `hesitate`, который приостанавливает поток на случайное время. Тем самым поведение программы становится недетерминированным и приближенным к реальности.

Листинг 13.1. Программа индексирования с ошибками (гонка)

```
@list = []
@list[0]="shoes ships\nsealing-wax"
@list[1]="cabbages kings"
@list[2]="quarks\nships\ncabbages"

def hesitate
  sleep rand(0)
end

@hash = {}

def process_list(listnum)
  lnum = 0
  @list[listnum].each do |line|
    words = line.chomp.split
    words.each do |w|
      hesitate
      if @hash[w]
        hesitate
        @hash[w] += ["#{listnum}:#{lnum}"]
      else
        hesitate
```

```

    @hash[w] = ["#{listnum}:#{lnum}"]
  end
end
lnum += 1
end
end

t1 = Thread.new(0) { |num| process_list(num) }
t2 = Thread.new(1) { |num| process_list(num) }
t3 = Thread.new(2) { |num| process_list(num) }

t1.join
t2.join
t3.join

count = 0
@hash.values.each { |v| count += v.size }

```

```
puts "Всего слов: #{@count} "      # Может быть напечатано 7 или 8!
```

Здесь имеется проблема. Если ваша система ведет себя примерно так же, как наша, то программа может напечатать одно из двух значений! В наших тестах с одинаковой вероятностью печаталось 7 или 8. Если слов и списков больше, то и разброс окажется более широким.

Попробуем исправить положение с помощью *мьютекса*, который будет контролировать доступ к разделяемому ресурсу. (Слово «mutex» – это сокращение от *mutual exclusion*, «взаимная блокировка».)

Обратимся к листингу 13.2. Библиотека *Mutex* позволяет создавать мьютексы и манипулировать ими. Мы можем захватить (*lock*) мьютекс перед доступом к хэшу и освободить (*unlock*) его по завершении операции.

Листинг 13.2. Программа индексирования с мьютексом

```

require 'thread.rb'

@list = []
@list[0]="shoes ships\nsealing-wax"
@list[1]="cabbages kings"
@list[2]="quarks\nships\ncabbages"

def hesitate
  sleep rand(0)
end

@hash = {}

@mutex = Mutex.new

def process_list(listnum)

```

```

lnum = 0
@list[listnum].each do |line|
  words = line.chomp.split
  words.each do |w|
    hesitate
    @mutex.lock
    if @hash[w]
      hesitate
      @hash[w] += ["#{listnum}:#{lnum}"]
    else
      hesitate
      @hash[w] = ["#{listnum}:#{lnum}"]
    end
    @mutex.unlock
  end
  lnum += 1
end
end

```

```

t1 = Thread.new(0) { |num| process_list(num) }
t2 = Thread.new(1) { |num| process_list(num) }
t3 = Thread.new(2) { |num| process_list(num) }

```

```

t1.join
t2.join
t3.join

```

```

count = 0
@hash.values.each { |v| count += v.size }

```

```

puts "Всего слов: #{count} "      # Всегда печатается 8!

```

Отметим, что помимо метода `lock` в классе `Mutex` есть также метод `try_lock`. Он отличается от `lock` тем, что если мьютекс уже захвачен другим потоком, то он не дожидается освобождения, а сразу возвращает `false`.

```

require 'thread'

```

```

mutex = Mutex.new
t1 = Thread.new { mutex.lock; sleep 30 }

```

```

sleep 1

```

```

t2 = Thread.new do
  if mutex.try_lock
    puts "Захватил"
  else
    puts "Не сумел захватить"      # Печатается немедленно.
  end
end
end

```

```

sleep 2

```

Эта возможность полезна, если поток не хочет приостанавливать выполнение

Есть также метод `synchronize`, который захватывает мьютекс, а потом автоматически освобождает его.

```
mutex = Mutex.new
```

```
mutex.synchronize do
```

```
  # Любой код, нуждающийся в защите...
```

```
end
```

Существует еще библиотека `mutex_m`, где определен модуль `Mutex_m`, который можно подмешивать к классу (или использовать для расширения объекта). У того расширенного объекта будут все методы мьютекса, так что он сам может выступать в роли мьютекса.

```
require 'mutex_m'
```

```
class MyClass
```

```
  include Mutex_m
```

```
  # Теперь любой объект класса MyClass может вызывать
```

```
  # методы lock, unlock, synchronize...
```

```
  # Внешние объекты также могут вызывать эти
```

```
  # методы для объекта MyClass.
```

```
end
```

13.2.3. Предопределенные классы синхронизированных очередей

В библиотеке `thread.rb` есть пара классов, которые иногда бывают полезны. Класс `Queue` реализует безопасную относительно потоков очередь, доступ к обоим концам которой синхронизирован. Это означает, что разные потоки могут, ничего не опасаясь, работать с такой очередью. Класс `SizedQueue` отличается от предыдущего тем, что позволяет ограничить размер очереди (число элементов в ней).

Оба класса имеют практически один и тот же набор методов, поскольку `SizedQueue` наследует `Queue`. Правда, в подклассе определен еще акцессор `max`, позволяющий получить и установить максимальный размер очереди.

```
buff = SizedQueue.new(25)
```

```
upper1 = buff.max           # 25
```

```
# Увеличить размер очереди...
```

```
buff.max = 50
```

```
upper2 = buff.max           # 50
```

В листинге 13.3 приведено решение задачи о производителе и потребителе. Для производителя задержка (аргумент `sleep`) чуть больше, чем для потребителя, чтобы единицы продукции «накапливались».

Листинг 13.3. Задача о производителе и потребителе

```
require 'thread'
```

```
buffer = SizedQueue.new(2)
```

```
producer = Thread.new do
  item = 0
  loop do
    sleep rand 0
    puts "Производитель произвел #{item}"
    buffer.enq item
    item += 1
  end
end

consumer = Thread.new do
  loop do
    sleep (rand 0)+0.9
    item = buffer.deq
    puts "Потребитель потребил #{item}"
    puts "  ожидает = #{buffer.num_waiting}"
  end
end
```

sleep 60 # Работать одну минуту, потом завершить оба потока.

Чтобы поместить элемент в очередь и извлечь из нее, рекомендуется применять соответственно методы `enq` и `deq`. Можно было бы для помещения в очередь пользоваться также методом `push`, а для извлечения — методами `pop` и `shift`, но их названия не так mnemonicны в применении к очередям.

Метод `empty?` проверяет, пуста ли очередь, а метод `clear` опустошает ее. Метод `size` (и его синоним `length`) возвращает число элементов в очереди.

Предполагается, что другие потоки не мешают...

```
buff = Queue.new
buff.enq "one"
buff.enq "two"
buff.enq "three"
n1 = buff.size      # 3
flag1 = buff.empty? # false
buff.clear
n2 = buff.size      # 0
flag2 = buff.empty? # true
```

Метод `num_waiting` возвращает число потоков, ожидающих доступа к очереди. Если размер очереди не ограничен, то это потоки, ожидающие возможности удалить элементы; для ограниченной очереди включаются также потоки, пытающиеся добавить элементы.

Необязательный параметр `non_block` метода `deq` в классе `Queue` по умолчанию равен `false`. Если же он равен `true`, то при попытке извлечь элемент из пустой очереди он не блокирует поток, а возбуждает исключение `ThreadError`.

13.2.4. Условные переменные

.....
*Да зовите моих скрипачей, трубачей...
«Веселый король» (детский стишок)**

Условная переменная – это, по существу, очередь потоков. Они используются в сочетании с мьютексами для лучшего управления синхронизацией потоков.

Условная переменная всегда ассоциируется с каким-то мьютексом. Ее назначение – освободить мьютекс до тех пор, пока не начнет выполняться определенное условие. Представьте себе ситуацию, когда поток захватил мьютекс, но не готов продолжать выполнение. Тогда он может заснуть под контролем условной переменной, ожидая, что будет разбужен, когда условие станет истинным.

Важно понимать, что пока поток ждет условную переменную, мьютекс свободен, поэтому другие потоки могут получить доступ к защищенному им ресурсу. А как только другой поток сигнализирует этой переменной, ожидающий поток пробуждается и пытается вновь захватить мьютекс.

Рассмотрим несколько искусственный пример в духе задачи об обедающих философах. Представьте себе, что вокруг стола сидят три скрипача, ожидающих своей очереди поиграть. Но у них есть всего две скрипки и один смычок. Понятно, что скрипач сможет играть, только если одновременно завладеет одной из скрипок и смычком.

Мы поддерживаем счетчики свободных скрипок и смычков. Когда скрипач хочет получить скрипку и смычок, он должен ждать их освобождения. В программе ниже мы защитили проверку условия мьютексом и под его защитой ждем скрипку и смычок порознь. Если скрипка или смычок заняты, поток засыпает. Он не владеет мьютексом до тех пор, пока другой поток не просигнализирует о том, что ресурс свободен. В этот момент первый поток просыпается и снова захватывает мьютекс.

Код представлен в листинге 13.4.

Листинг 13.4. Три скрипача

```
require 'thread'

@music = Mutex.new
@violin = ConditionVariable.new
@bow = ConditionVariable.new

@violins_free = 2
@bows_free = 1

def musician(n)
  loop do
    sleep rand(0)
```

* Пер. С. Маршака. – Прим. ред.

```

@music.synchronize do
  @violin.wait(@music) while @violins_free == 0
  @violins_free -= 1
  puts "#{n} владеет скрипкой"
  puts "скрипок #{@violins_free}, смычков #{@bows_free}"

  @bow.wait(@music) while @bows_free == 0
  @bows_free -= 1
  puts "#{n} владеет смычком"
  puts "скрипок #{@violins_free}, смычков #{@bows_free}"
end

sleep rand(0)
puts "#{n}: {...играет...}"
sleep rand(0)
puts "#{n}: Я закончил."

@music.synchronize do
  @violins_free += 1
  @violin.signal if @violins_free == 1
  @bows_free += 1
  @bow.signal if @bows_free == 1
end
end
end

threads = []
3.times { |i| threads << Thread.new { musician(i) } }

threads.each { |t| t.join }

```

Мы полагаем, что это решение никогда не приводит к тупиковой ситуации, хотя доказать этого не сумели. Но интересно отметить, что описанный алгоритм не справедливый. В наших тестах оказалось, что первый скрипач играет чаще двух остальных, а второй чаще третьего. Выяснение причин такого поведения и его исправление мы оставляем читателю в качестве интересного упражнения.

13.2.5. Другие способы синхронизации

Еще один механизм синхронизации – это монитор, который в Ruby реализован в библиотеке `monitor.rb`. Это более развитый по сравнению с мьютексом механизм, основное отличие состоит в том, что захваты одного и того же мьютекса не могут быть вложенными, а монитора – могут.

Тривиальный случай возникновения такой ситуации вряд ли возможен. В самом деле, кто станет писать такой код:

```

@mutex = Mutex.new

@mutex.synchronize do
  @mutex.synchronize do
    #...

```

```
end  
end
```

Но нечто подобное может произойти в сложной программе (или при рекурсивном вызове метода). Какова бы ни была причина, последствием будет тупиковая ситуация. Уход от нее – одно из достоинств модуля-примеси Monitor.

```
@mutex = Mutex.new  
  
def some_method  
  @mutex.synchronize do  
    #...  
    some_other_method    # Тупиковая ситуация!  
  end  
end  
  
def some_other_method  
  @mutex.synchronize do  
    #...  
  end  
end
```

Модуль-примесь monitor обычно применяется для расширения объекта. Для создания условной переменной предназначен метод new_cond.

Класс ConditionVariable в библиотеке monitor.rb дополнен по сравнению с определением в библиотеке thread. У него есть методы wait_until и wait_while, которые блокируют поток в ожидании выполнения условия. Кроме того, возможен тайм-аут при ожидании, поскольку у метода wait имеется параметр timeout, равный количеству секунд (по умолчанию nil).

Поскольку примеры работы с потоками у нас кончаются, то в листинге 13.5 мы предлагаем реализацию классов Queue и SizedQueue с помощью монитора. Код приводится с разрешения автора, Шюго Маэда (Shugo Maeda).

Листинг 13.5. Реализация класса Queue с помощью монитора

Автор: Shugo Maeda

```
require 'monitor'  
  
class Queue  
  def initialize  
    @que = []  
    @monitor = Monitor.new  
    @empty_cond = @monitor.new_cond  
  end  
  
  def enq(obj)  
    @monitor.synchronize do  
      @que.push(obj)  
      @empty_cond.signal  
    end  
  end  
end
```

```
def deq
  @monitor.synchronize do
    while @que.empty?
      @empty_cond.wait
    end
    return @que.shift
  end
end

class SizedQueue < Queue
  attr :max

  def initialize(max)
    super()
    @max = max
    @full_cond = @monitor.new_cond
  end

  def enq(obj)
    @monitor.synchronize do
      while @que.length >= @max
        @full_cond.wait
      end
      super(obj)
    end
  end

  def deq
    @monitor.synchronize do
      obj = super
      if @que.length < @max
        @full_cond.signal
      end
      return obj
    end
  end

  def max=(max)
    @monitor.synchronize do
      @max = max
      @full_cond.broadcast
    end
  end
end
```

Еще один вариант синхронизации (двухзначную блокировку со счетчиком) предлагает библиотека `sync.rb`. В ней определен модуль `Sync_m`, который можно применять вместе с ключевыми словами `include` и `extend` (как и `Mutex_m`). Этот модуль содержит методы `locked?`, `shared?`, `exclusive?`, `lock`, `unlock` и `try_lock`.

13.2.6. Тайм-аут при выполнении операций

Часто встречается ситуация, когда на выполнение операции отводится определенное максимальное время. Это позволяет избежать бесконечных циклов и более строго контролировать порядок работы. Подобная возможность очень полезна, в частности, в сетевых приложениях, где ответ от сервера может и не прийти.

Библиотека `timeout.rb` предлагает решение этой проблемы на основе потоков (см. листинг 13.6). С методом `timeout` ассоциирован выполняемый блок. Если истечет заданное число секунд, метод возбуждает исключение `TimeoutError`, которое можно перехватить с помощью `rescue`.

Листинг 13.6. Пример тайм-аута

```
require 'timeout.rb'

flag = false
answer = nil

begin
  timeout(5) do
    puts "Хочу печенье!"
    answer = gets.chomp
    flag = true
  end
rescue TimeoutError
  flag = false
end

if flag
  if answer == "cookie"
    puts "Спасибо! Хрум, хрум..."
  else
    puts "Это же не печенье!"
    exit
  end
else
  puts "Эй, слишком медленно!"
  exit
end

puts "До встречи..."
```

13.2.7. Ожидание события

Часто один или несколько потоков следят за «внешним миром», а остальные выполняют полезную работу. Все примеры в этом разделе надуманные, но общий принцип они все же иллюстрируют.

В следующем примере прикладную задачу решают три потока. Четвертый поток каждые пять секунд просыпается, проверяет глобальную переменную `$flag` и, когда видит, что флаг поднят, пробуждает еще два потока. Это освобождает три

рабочих потока от необходимости напрямую общаться с двумя другими и, возможно, от многочисленных попыток разбудить их.

```
$flag = false
work1 = Thread.new { job1() }
work2 = Thread.new { job2() }
work3 = Thread.new { job3() }

thread4 = Thread.new { Thread.stop; job4() }
thread5 = Thread.new { Thread.stop; job5() }

watcher = Thread.new do
  loop do
    sleep 5
    if $flag
      thread4.wakeup
      thread5.wakeup
      Thread.exit
    end
  end
end
```

Если в какой-то момент выполнения метода `job`, переменная `$flag` станет равной `true`, то в течение пяти секунд после этого потоки `thread4` и `thread5` гарантированно запустятся. После этого поток `watcher` завершается.

В следующем примере мы ждем создания файла. Каждые 30 секунд проверяется его существование, и как только файл появится, мы запускаем новый поток. Тем временем остальные потоки занимаются своим делом. На самом деле ниже мы наблюдаем за тремя разными файлами.

```
def waitfor(filename)
  loop do
    if File.exist? filename
      file_processor = Thread.new { process_file(filename) }
      Thread.exit
    else
      sleep 30
    end
  end
end

waiter1 = Thread.new { waitfor("Godot") }
sleep 10
waiter2 = Thread.new { waitfor("Guffman") }
sleep 10
headwaiter = Thread.new { waitfor("head") }
```

Основной поток занимается другими делами...

Есть много ситуаций, когда поток должен ожидать внешнего события (например, в сетевых приложениях так бывает, когда сервер на другом конце соединения работает медленно или ненадежно).

13.2.8. Продолжение обработки во время ввода/вывода

Часто приложению приходится выполнять одну или более длительных операций ввода/вывода. Прежде всего, речь идет о вводе данных с клавиатуры, поскольку человек печатает куда медленнее, чем вращается диск. Это время можно употребить на пользу с помощью потоков.

Возьмем, к примеру, шахматную программу, которая должна ждать, пока человек сделает ход. Конечно, мы можем изложить только сам принцип, не вдаваясь в технические детали.

Предположим, что итератор `predict_move` генерирует вероятные ходы человека (и ответные ходы программы). Тогда в момент, когда человек сделает ход, не исключено, что у компьютера уже будет готов ответ.

```
scenario = {}           # Хэш ход-ответ.
humans_turn = true
thinking_ahead = Thread.new(board) do
  predict_move do |m|
    scenario[m] = my_response(board,m)
    Thread.exit if humans_turn == false
  end
end

human_move = get_human_move(board)
humans_turn = false    # Остановить поток.

# Теперь можно посмотреть, нет ли в хэше scenario хода,
# сделанного пользователем...
```

Конечно, настоящие шахматные программы работают не так.

13.2.9. Реализация параллельных итераторов

Предположим, что нужно параллельно обходить несколько объектов, то есть для каждого объекта найти первый элемент, потом второй, потом третий и т.д.

Рассмотрим следующий пример. Пусть `compose` — имя магического метода, который выполняет композицию итераторов. Допустим еще, что у каждого объекта есть стандартный итератор `each` и что каждый объект возвращает по одному элементу на каждой итерации.

```
arr1 = [1, 2, 3, 4]
arr2 = [5, 10, 15, 20]
compose(arr1, arr2) { |a,b| puts "#{a} и #{b}" }

# Должно быть напечатано:
# 1 и 5
# 2 и 10
# 3 и 15
# 4 и 20
```

Можно было бы, конечно, использовать для этой цели `zip`. Но если нужно более элегантное решение, при котором все элементы не будут храниться одновременно, то без потоков не обойтись. Такое решение представлено в листинге 13.7.

Листинг 13.7. Параллельные итераторы

```

def compose(*objects)
  threads = []
  for obj in objects do
    threads << Thread.new(obj) do |myobj|
      me = Thread.current
      me[:queue] = []
      myobj.each {|x| me[:queue].push(x) }
    end
  end

  list = [0] # Фиктивное значение, отличное от nil.
  while list.nitems > 0 do # Еще есть не nil.
    list = []
    for thr in threads
      list << thr[:queue].shift # Удалить по одному из каждого.
    end
    yield list if list.nitems > 0 # Не вызывать yield, если все равны nil.
  end
end

x = [1, 2, 3, 4, 5, 6, 7, 8]
y = "    первый\n    второй\n    третий\n    четвертый\n    пятый\n"
z = %w[a b c d e f]

compose(x, y, z) {|a,b,c| p [a, b, c] }

# Выводится:
#
# [1, "    первый\n", "a"]
# [2, "    второй\n", "b"]
# [3, "    третий\n", "c"]
# [4, "    четвертый\n", "d"]
# [5, "    пятый\n", "e"]
# [6, nil, "f"]
# [7, nil, nil]
# [8, nil, nil]

```

Обратите внимание: мы не предполагаем, что все объекты имеют одно и то же число элементов. Если один итератор доходит до конца раньше остальных, то он будет генерировать значения `nil` до тех пор, пока не закончит работу «самый длинный» итератор.

Конечно, можно написать и более общий метод, который на каждой итерации будет обрабатывать более одного элемента. (В конце концов, не все итераторы возвращают по одному значению за раз.) Можно было бы в первом параметре передавать число значений для каждого итератора.

Можно также пользоваться произвольными итераторами (а не только стандартным `each`). Их имена можно было бы передавать в виде строк, а вызывать с помощью метода `send`. Много чего еще можно придумать.

Впрочем, мы полагаем, что приведенного кода достаточно для большинства целей. Вариации на эту тему оставляем читателю в качестве упражнения.

13.2.10. Параллельное рекурсивное удаление

Забавы ради напишем код, который будет удалять дерево каталогов. Процедура рекурсивного удаления использует потоки. Как только обнаруживается очередной подкаталог, мы запускаем новый поток, который будет обходить его и удалять содержимое.

Созданные в ходе работы программы потоки хранятся в массиве `threads`. Поскольку это локальная переменная, у каждого потока будет собственная копия массива. Раз к ней может обращаться всего один поток, синхронизировать доступ не надо.

Отметим также, что в блок потока передается полное имя файла `fullname`, чтобы не нужно было беспокоиться по поводу того, что поток обращается к переменной, которую кто-то еще изменяет. Поток делает для себя локальную копию `fn` этой переменной.

Прежде чем удалять очередной каталог, мы должны дождаться завершения всех созданных в процессе его обхода потоков.

```
def delete_all(dir)
  threads = []
  Dir.foreach(dir) do |e|
    next if [".", ".."].include? e          # Пропустить . и ..
    fullname = dir + "/" + e
    if FileTest.directory?(fullname)
      threads << Thread.new(fullname) { |fn| delete_all(fn) }
    else
      File.delete(fullname)
    end
  end
  threads.each { |t| t.join }
  Dir.delete(dir)
end
delete_all("/tmp/stuff")
```

Будет ли работать такая программа быстрее, чем ее вариант без потоков? В наших тестах получалось по-разному. Возможно, это зависит от операционной системы и структуры конкретного каталога – глубины, количества файлов и т.д.

13.3. Заключение

Как было сказано, в Ruby не используются платформенные потоки. Программа не станет работать быстрее при наличии нескольких процессоров, но некоторого параллеливания работы достичь все же можно. Потоки полезны во многих случаях, но писать и отлаживать многопоточную программу довольно трудно, особенно если для получения правильного результата приходится применять изощренные способы синхронизации.

Для синхронизации Ruby предоставляет такие классы, как `Mutex`, `Monitor` и `ConditionVariable`. Имеются также безопасные относительно потоков классы очередей `Queue` и `SizedQueue`.

В главе 14 мы перейдем от обсуждения техники программирования к решению конкретных задач, а именно сценариев системного администрирования.

Глава 14. Сценарии и системное администрирование

.....
И сказал главный программист: «Даже если программа состоит всего из трех строчек, когда-то ее придется сопровождать».
Джеффри Джеймс, «Дао программирования»

Программистам часто приходится писать небольшие сценарии для запуска внешних программ и работы с операционной системой на достаточно высоком уровне. Особенно это относится к ОС UNIX, где для повседневной работы составляются многочисленные сценарии на языке интерпретатора команд (shell).

Ruby не всегда удобно использовать в качестве такого «склеивающего» языка, поскольку он задуман как более универсальный инструмент. Но в принципе все, что можно сделать на языке `bash` (и ему подобных), можно реализовать и на Ruby.

Нередко для этой цели можно воспользоваться каким-то более традиционным языком. Преимущества Ruby в его универсальности, богатстве функций и объектной ориентированности. Предполагая, что найдутся люди, желающие использовать Ruby для взаимодействия с операционной системой на таком уровне, мы продемонстрируем несколько полезных приемов.

Выстроить эту главу было трудно, поскольку есть много способов логически сгруппировать рассматриваемый материал. Если вы не смогли найти нужную тему там, где ожидали, попробуйте просмотреть другие разделы.

Кроме того, многие вопросы, которые можно было бы включить сюда, вошли в другие главы. Обратите внимание, в частности, на главу 10, где рассматриваются ввод/вывод и атрибуты файлов; эта информация часто бывает полезна при написании сценариев.

14.1. Запуск внешних программ

Никакой язык не может использоваться в качестве «клея», если он не позволяет запускать внешние программы. В Ruby для этого есть несколько способов.

Не могу не обмолвиться о том, что перед запуском внешней программы неплохо бы понять, что она делает. Я имею в виду вирусы и другие потенциально разрушительные программы. Не запускайте произвольную командную строку, особенно поступившую из внешнего источника. Это касается не только приложений, ориентированных на Web.

14.1.1. Методы system и exec

Метод system (из модуля kernel) эквивалентен одноименной функции из библиотеки языка C. Он выполняет указанную команду в отдельной оболочке.

```
system("/usr/games/fortune")
# Вывод направляется, как обычно, на stdout...
```

Второй параметр, если он задан, должен содержать список аргументов; как правило, аргументы можно задавать и в командной строке – эффект будет тот же. Разница лишь в том, что алгоритм расширения имени файла применяется только к первой из переданных строк.

```
system("rm", "/tmp/file1")
system("rm /tmp/file2")
# Оба варианта годятся.
```

```
# А тут есть различие...
system("echo **")      # Печатается список всех файлов.
system("echo", "**")    # Печатается звездочка (расширение
                        # имени файла не производится).
```

```
# Более сложные командные строки тоже работают.
system("ls -l | head -n 1")
```

Посмотрим, как это будет работать в семействе операционных систем Windows. В случае с простой исполняемой программой поведение должно быть таким же, как в UNIX. В зависимости от варианта Ruby для вызова встроенных в оболочку команд может потребоваться запуск cmd.exe – интерпретатора команд в Windows (в некоторых версиях ОС он называется command.com). Ниже приведены примеры запуска внешней и встроенной команды:

```
system("notepad.exe", "myfile.txt") # Никаких проблем...
system("cmd /c dir", "somefile")    # 'dir' - встроенная команда!
```

Другое решение – воспользоваться библиотекой win32API и определить собственный вариант метода system.

```
require "Win32API"
```

```
def system(cmd)
  sys = Win32API.new("crtdll", "system", ['P'], 'L')
  sys.Call(cmd)
end
```

```
system("dir") # cmd /c необязательно!
```

Таким образом, можно добиться более-менее системно-независимого поведения system. Но если вы хотите запомнить выведенную программой информацию (например, в переменной), то system – не лучший способ (см. следующий раздел).

Упомяну еще метод exec. Он ведет себя аналогично system с тем отличием, что новый процесс замещает текущий. Поэтому код, следующий за exec, исполняться не будет.

```
puts "Содержимое каталога:"
exec("ls", "-l")
```

```
puts "Эта строка никогда не исполняется!"
```

14.1.2. Перехват вывода программы

Простейший способ перехватить информацию, выведенную программой, – заключить команду в обратные кавычки, например:

```
listing = 'ls -l' # Одна строка будет содержать несколько строчек (lines).
now = 'date'      # "Mon Mar 12 16:50:11 CST 2001"
```

Обобщенный ограничитель `%x` вызывает оператор обратных кавычек (который в действительности является методом модуля `kernel`). Работает он точно так же:

```
listing = %x(ls -l)
now = %x(date)
```

Применение `%x` бывает полезно, когда подлежащая исполнению строка содержит такие символы, как одиночные и двойные кавычки.

Поскольку обратные кавычки – это на самом деле метод (в некотором смысле), то его можно переопределить. Изменим его так, чтобы он возвращал не одну строку, а массив строк. Конечно, при этом мы создадим синоним старого метода, чтобы его можно было вызвать.

```
alias old_execute '

def '(cmd)
  out = old_execute(cmd) # Вызвать исходный метод обратной кавычки.
  out.split("\n")       # Вернуть массив строк!
end
```

```
entries = 'ls -l /tmp'
num = entries.size # 95
```

```
first3lines = %x(ls -l | head -n 3)
how_many = first3lines.size # 3
```

Как видите, при таком определении изменяется также поведение ограничителя `%x`.

В следующем примере мы добавили в конец команды конструкцию интерпретатора команд, которая перенаправляет стандартный вывод для ошибок в стандартный вывод:

```
alias old_execute '

def '(cmd)
  old_execute(cmd + " 2>&1")
end

entries = 'ls -l /tmp/foobar'
# "/tmp/foobar: No such file or directory\n"
```

Есть, конечно, и много других способов изменить стандартное поведение обратных кавычек.

14.1.3. Манипулирование процессами

В этом разделе мы обсудим манипулирование процессами, хотя создание нового процесса необязательно связано с запуском внешней программы. Основной способ создания нового процесса – это метод `fork`, название которого в соответствии с традицией UNIX подразумевает разветвление пути исполнения, напоминая развилку на дороге. (Отметим, что в базовом дистрибутиве Ruby метод `fork` на платформе Windows не поддерживается.)

Метод `fork`, находящийся в модуле `Kernel` (а также в модуле `Process`), не следует путать с одноименным методом экземпляра в классе `Thread`.

Существуют два способа вызвать метод `fork`. Первый похож на то, как это обычно делается в UNIX, – вызвать и проверить возвращенное значение. Если оно равно `nil`, мы находимся в дочернем процессе, в противном случае – в родительском. Родительскому процессу возвращается идентификатор дочернего процесса (`pid`).

```
pid = fork
if (pid == nil)
  puts "Ага, я, должно быть, потомок."
  puts "Так и буду себя вести."
else
  puts "Я родитель."
  puts "Пора отказаться от детских шуточек."
end
```

В этом не слишком реалистичном примере выводимые строки могут чередоваться, а может случиться и так, что строки, выведенные родителем, появятся раньше. Но сейчас это несущественно.

Следует также отметить, что процесс-потомок может пережить своего родителя. Для потоков в Ruby это не так, но системные процессы – совсем другое дело.

Во втором варианте вызова метод `fork` принимает блок. Заключенный в блок код выполняется в контексте дочернего процесса. Так, предыдущий вариант можно было бы переписать следующим образом:

```
fork do
  puts "Ага, я, должно быть, потомок."
  puts "Так и буду себя вести."
end

puts "Я родитель."
puts "Пора отказаться от детских шуточек."
```

Конечно, `pid` по-прежнему возвращается, мы просто не показали его.

Чтобы дождаться завершения процесса, мы можем вызвать метод `wait` из модуля `Process`. Он ждет завершения любого потомка и возвращает его идентификатор. Метод `wait2` ведет себя аналогично, только возвращает массив, содержащий `pid`, и сдвинутый влево код завершения.

```
pid1 = fork { sleep 5; exit 3 }
pid2 = fork { sleep 2; exit 3 }
```

```
Process.wait      # Возвращает pid2
Process.wait2    # Возвращает [pid1, 768]
```


Чтобы дождаться завершения конкретного потомка, применяются методы `waitpid` и `waitpid2`.

```
pid3 = fork { sleep 5; exit 3 }
pid4 = fork { sleep 2; exit 3 }
```

```
Process.waitpid(pid4, Process::WNOHANG) # Возвращает pid4
Process.waitpid2(pid3, Process::WNOHANG) # Возвращает [pid3, 768]
```

Если второй параметр не задан, то вызов может блокировать программу (если такого потомка не существует). Второй параметр можно с помощью ИЛИ объединить с флагом `Process::WUNTRACED`, чтобы перехватывать остановленные процессы. Этот параметр системно зависим, поэкспериментируйте.

Метод `exit!` немедленно завершает процесс (не вызывая зарегистрированных обработчиков). Если задан целочисленный аргумент, то он возвращается в качестве кода завершения; по умолчанию подразумевается значение 1 (не 0).

```
pid1 = fork { exit! } # Вернуть код завершения -1.
pid2 = fork { exit! 0 } # Вернуть код завершения 0.
```

Методы `pid` и `ppid` возвращают соответственно идентификатор текущего и родительского процессов.

```
procl = Process.pid
fork do
  if Process.ppid == procl
    puts "procl - мой родитель" # Печатается это сообщение.
  else
    puts "Что происходит?"
  end
end
```

Метод `kill` служит для отправки процессу сигнала, как это понимается в UNIX. Первый параметр может быть целым числом, именем POSIX-сигнала с префиксом `SIG` или именем сигнала без префикса. Второй параметр – идентификатор процесса-получателя; если он равен нулю, подразумевается текущий процесс.

```
Process.kill(1, pid1) # Послать сигнал 1 процессу pid1.
Process.kill("HUP", pid2) # Послать SIGHUP процессу pid2.
Process.kill("SIGHUP", pid2) # Послать SIGHUP процессу pid3.
Process.kill("SIGHUP", 0) # Послать SIGHUP самому себе.
```

Для обработки сигналов применяется метод `Kernel.trap`. Обычно он принимает номер или имя сигнала и подлежащий выполнению блок.

```
trap(1) { puts "Перехвачен сигнал 1" }
sleep 2
Process.kill(1, 0) # Послать самому себе.
```

О применении метода `trap` в более сложных ситуациях читайте в документации по Ruby и UNIX.

В модуле `Process` есть также методы для опроса и установки таких атрибутов процесса, как идентификатор пользователя, действующий идентификатор пользователя, приоритет и т.д. Дополнительную информацию вы отыщите в справочном руководстве по Ruby.

14.1.4. Стандартный ввод и вывод

В главе 10 мы видели, как работают методы `IO.open` и `IO.pipe`, но существует еще небольшая библиотека, которая иногда бывает удобна.

В библиотеке `Open3.rb` есть метод `popen3`, который возвращает массив из трех объектов `IO`. Они соответствуют стандартному вводу, стандартному выводу и стандартному выводу для ошибок того процесса, который был запущен методом `popen3`.

Вот пример:

```
require "open3"

filenames = %w{ file1 file2 this that another one_more }

inp, out, err = Open3.popen3("xargs", "ls", "-l")

filenames.each { |f| inp.puts f } # Писать в stdin процесса.
inp.close                       # Закрывать обязательно!

output = out.readlines           # Читать из stdout.
errout = err.readlines          # Читать также из stderr.

puts "Послано #{filenames.size} строк входных данных."
puts "Получено #{output.size} строк из stdout"
puts "и #{errout.size} строк из stderr."
```

В этом искусственном примере мы выполняем команду `ls -l` для каждого из заданных имен файлов и по отдельности перехватываем стандартный вывод и стандартный вывод для ошибок. Отметим, что вызов `close` необходим, чтобы порожденный процесс увидел конец файла. Также отметим, что в библиотеке `Open3` используется метод `fork`, не реализованный на платформе Windows; для этой платформы придется пользоваться библиотекой `win32-open3` (ее написали и поддерживают Дэниэль Бергер (Daniel Berger) и Парк Хисоб (Park Heesob)).

См. также раздел 14.3.

14.2. Флаги и аргументы в командной строке

Слухи о кончине командной строки сильно преувеличены. Хотя мы и живем в век графических интерфейсов, ежедневно тысячи и тысячи программистов по тем или иным причинам обращаются к командным утилитам.

Мы уже говорили, что корнями своими Ruby уходит в UNIX. Но даже в Windows существует понятие командной строки, и, честно говоря, мы не думаем, что в обозримом будущем она исчезнет.

На этом уровне для управления работой программы применяются аргументы и флаги. О них мы и поговорим ниже.

14.2.1. Разбор флагов в командной строке

Для разбора командной строки чаще всего применяется библиотека `getoptlong` (библиотека `getopts.rb`, обладающая менее развитой функциональностью, считается устаревшей). Она понимает однобуквенные и длинные флаги и распознает

двойной дефис (--) как признак конца флагов. В целом библиотека ведет себя так же, как соответствующие функции GNU.

Необходимо создать объект класса `GetoptLong`, который и будет выполнять функции анализатора. Затем ему передаются допустимые значения флагов, а он извлекает их по одному.

У объекта-анализатора есть метод `set_options`, который принимает список массивов. Каждый массив содержит один или несколько флагов (в виде строк) и один «признак наличия аргумента», который говорит, должны ли эти флаги сопровождаться аргументами. Все флаги в одном массиве считаются синонимами; первый из них является «каноническим именем», которое и возвращает операция `get`.

Предположим, что имеется программа, понимающая следующие флаги: `-h` или `-help` (печать справки), `-f` или `--file` (указание имени файла), `-l` или `-lines` (вывод не более указанного числа строк, по умолчанию 100).

Такая программа могла бы начинаться следующим образом:

```
require "getoptlong"
```

```
parser = GetoptLong.new
parser.set_options(
  ["-h", "--help", GetoptLong::NO_ARGUMENT],
  ["-f", "--file", GetoptLong::REQUIRED_ARGUMENT],
  ["-l", "--lines", GetoptLong::OPTIONAL_ARGUMENT])
```

Теперь можно в цикле вызвать метод `get` (см. листинг 14.1). Наличие операторных скобок `begin-end` имитирует цикл с проверкой условия в конце. У метода `get` есть синоним `get_option`, существуют также итераторы `each` и `each_option`, которые в точности идентичны.

Листинг 14.1. Получение флагов из командной строки

```
filename = nil
lines = 0                                # По умолчанию вывод не усекается.

loop do
  begin
    opt, arg = parser.get
    break if not opt
    # Только для отладки...
    puts (opt + " => " + arg)

    case opt
      when "-h"
        puts "Usage: ..."
        break                                # Прекратить обработку, если задан флаг -h.
      when "-f"
        filename = arg # Запомнить аргумент - имя файла.
      when "-l"
        if arg != ""
          lines = arg # Запомнить аргумент - число строк (если задан).
```

```

    else
      lines = 100 # Оставляем по умолчанию число строк.
    end
  end

  rescue => err
    puts err
    break
  end
end

puts "имя файла   = #{filename}"
puts "число строк = #{lines}"

```

Метод `get` возвращает `nil`, если флаг отсутствует, но пустую строку, если для флага не задан аргумент. Возможно, это ошибка.

В этом примере мы перехватываем исключения. Всего их может быть четыре:

- `AmbiguousOption` – указано сокращенное длинное имя флага, но сокращение не уникально;
- `InvalidOption` – неизвестный флаг;
- `MissingArgument` – для флага не задан аргумент;
- `NeedlessArgument` – указан аргумент для флага, который не должен сопровождаться аргументом.

Сообщения об ошибках обычно выводятся на `stderr`, но вывод можно подавить, присвоив аксессуару `quiet` значение `true`.

Библиотека `getoptlong` располагает и другими возможностями, которых мы здесь не обсуждали. Подробности вы найдете в документации.

Существуют другие библиотеки, например `OptionParser`, предлагающие несколько иную функциональность. Дополнительная информация приведена в архиве приложений Ruby.

14.2.2. Константа ARGF

Глобальная константа `ARGF` представляет псевдофайл, получающийся в результате конкатенации всех имен файлов, заданных в командной строке. Во многих отношениях она ведет себя так же, как объект `IO`.

Когда в программе встречается «голый» метод ввода (без указания вызывающего объекта), обычно имеется в виду метод, подмешанный из модуля `kernel` (например, `gets` и `readlines`). Если в командной строке не задано ни одного файла, то по умолчанию источником ввода является объект `STDIN`. Но если файлы заданы, то данные читаются из них. Понятно, что конец файла достигается в конце последнего из указанных файлов.

Если хотите, можете обращаться к `ARGF` явно:

```

# Скопировать все файлы на stdout.
puts ARGF.readlines

```

Быть может, вопреки ожиданиям, признак конца файла устанавливается после каждого файла. Так, предыдущий код выведет все файлы, а следующий – только первый файл:

```
until ARGF.eof?
  puts ARGF.gets
end
```

Является ли это ошибкой, предоставим судить вам. Впрочем, сюрпризы могут быть и приятными. Входные данные – не просто поток байтов; мы можем применять к ARGF операции `seek` и `rewind`, как если бы это был «настоящий файл».

С константой ARGF ассоциирован метод `file` – он возвращает объект IO, соответствующий файлу, обрабатываемому в данный момент. Естественно, возвращаемое значение изменяется по мере перехода от одного файла к другому.

А если мы не хотим интерпретировать имена аргументов в командной строке как имена файлов? Тогда не надо обращаться к методам ввода без указания вызывающего объекта. Если вы хотите читать из стандартного ввода, укажите в качестве такого объекта `STDIN`, и все будет работать правильно.

14.2.3. Константа ARGV

Глобальная константа ARGV представляет список аргументов, переданных в командной строке. По сути дела, это массив.

```
n = ARGV.size
argstr = '' + ARGV*"," + ''
puts "Мне было передано аргументов: #{n}..."
puts "Вот они: #{argstr}"
puts "Заметьте, что ARGV[0] = #{ARGV[0]}"
```

Если запустить эту программу с аргументами `red green blue`, то она напечатает:

```
Мне было передано аргументов: 3...
Вот они: "red,green,blue"
Заметьте, что ARGV[0] = red
```

Ясно, что отдельно передавать число аргументов, как в былые времена, не нужно; эта информация – часть массива.

Привычных к старым соглашениям программистов может смутить также тот факт, что нулевой элемент массива – настоящий аргумент (а не, скажем, имя сценария). Нумерация аргументов начинается с нуля, а не с единицы, как в языке Си в различных интерпретаторах команд.

14.3. Библиотека Shell

Не всегда Ruby удобен в качестве языка сценариев. Например, в языке `bash` для запуска внешней программы достаточно просто указать ее имя безо всякого дополнительного синтаксиса.

Оборотной стороной мощи и гибкости Ruby является более сложный синтаксис. Кроме того, функциональность разнесена по различным классам, модулям и библиотекам.

Это послужило основанием для создания библиотеки Shell, которая упрощает, к примеру, организацию конвейеров команд и перенаправление вывода в файл. Кроме того, она сводит воедино функциональность из разных источников, скрывая ее за интерфейсом объекта Shell. (На платформе Windows эта библиотека работает не во всех случаях.)

14.3.1. Использование библиотеки Shell для перенаправления ввода/вывода

В классе Shell для создания объектов есть два метода: `new` и `cd`. Первый создает объект, ассоциированный с текущим каталогом, второй – объект, для которого рабочим будет указанный каталог.

```
require "shell"
```

```
sh1 = Shell.new          # Работать в текущем каталоге.
sh2 = Shell.cd("/tmp/hal") # Работать в каталоге /tmp/hal.
```

Библиотека Shell определяет несколько встроенных команд (например, `echo`, `cat` и `tee`) в виде методов. Они всегда возвращают объекты класса `Filter` (как и определяемые пользователем команды, с которыми мы вскоре познакомимся).

Класс `Filter` понимает, что такое перенаправление ввода/вывода. В нем определены методы (или операторы) `<`, `>` и `|`, которые ведут себя примерно так, как мы ожидаем по многолетнему опыту написания shell-сценариев.

Если методу перенаправления передать в качестве параметра строку, то она будет считаться именем файла. Если же параметром является объект `IO`, он используется для операций ввода/вывода. Примеры:

```
sh = Shell.new
# Вывести файл motd на stdout.
sh.cat("/etc/motd") > STDOUT

# Напечатать его еще раз.
(sh.cat < "/etc/motd") > STDOUT
(sh.echo "Это тест") > "myfile.txt"

# Добавить строку в конец файла /etc/motd.
sh.echo("Hello, world!") >> "/etc/motd"

# Вывести два файла на stdout и продублировать (tee) вывод в третий файл.
(sh.cat "file1" "file2") | (tee "file3") > STDOUT
```

Отметим, что у оператора `>` высокий приоритет. Скобки, которые вы видите в данном примере, в большинстве случаев обязательны. Вот два примера правильного использования и один – неправильного:

```
# Интерпретатор Ruby понимает такую конструкцию...
sh.cat("myfile.txt") > STDOUT

# ...и такую тоже.
```

```
(sh.cat "myfile.txt") > STDOUT
```

```
# TypeError! (ошибка связана с приоритетами).
```

```
sh.cat "myfile.txt" > STDOUT
```

Отметим еще, что можно «инсталлировать» системные команды по своему выбору. Для этого служит метод `def_system_command`. Ниже определяются два метода: `ls` и `ll`, которые выводят список файлов в текущем каталоге (в коротком и длинном формате).

```
# Имя метода совпадает с именем команды...
```

```
# Необходим только один параметр:
```

```
Shell.def_system_command "ls"
```

```
# А здесь должно быть два параметра:
```

```
Shell.def_system_command "ll", "ls -l"
```

```
sh = Shell.new
```

```
sh.ls > STDOUT # Короткий формат.
```

```
sh.ll > STDOUT # Длинный формат.
```

Вы, наверное, обратили внимание на то, что в большинстве случаев мы явно отправляем вывод объекту `STDOUT`. Связано это с тем, что объект `Shell` автоматически вывод команд никуда не направляет. Он просто ассоциирует его с объектом `Filter`, который уже может быть связан с файлом или с объектом `IO`.

14.3.2. Дополнительные замечания по поводу библиотеки `shell.rb`

Метод `transact` исполняет блок в контексте вызывающего объекта. Таким образом, допустима следующая сокращенная запись:

```
sh = Shell.new
```

```
sh.transact do
```

```
  echo("Строка данных") > "somefile.txt"
```

```
  cat("somefile.txt", "otherfile.txt") > "thirdfile"
```

```
  cat("thirdfile") | tee("file4") > STDOUT
```

```
end
```

Итератор `foreach` принимает в качестве параметра файл или каталог. Если это файл, он перебирает все его строки, а если каталог — все имена файлов в нем.

```
sh = Shell.new
```

```
# Напечатать все строки файла /tmp/foo.
```

```
sh.foreach("/tmp/foo") {|l| puts l }
```

```
# Вывести список файлов в каталоге /tmp.
```

```
sh.foreach("/tmp") {|f| puts f }
```

Метод `pushdir` запоминает текущий каталог, а метод `popdir` делает последний запомненный каталог текущим. У них есть синонимы `pushd` и `popd`. Метод `pwd` возвращает текущий рабочий каталог, его синонимы — `getwd`, `cwd` и `dir`.

```
sh = Shell.cd "/home"

puts sh.pwd      # /home
sh.pushd "/tmp"
puts sh.pwd      # /tmp

sh.popd
puts sh.pwd      # /home
```

Для удобства в класс Shell импортируются методы из различных источников, в том числе из класса File, модуля FileTest и библиотеки ftools.rb. Это избавляет от необходимости выполнять require, include, создавать объекты, квалифицировать вызовы методов и т.д.

```
sh = Shell.new
flag1 = sh.exist? "myfile"      # Проверить существование файла.
sh.delete "somefile"           # Удалить файл.
sh.move "/tmp/foo", "/tmp/bar"  # Переместить файл.
```

У библиотеки Shell есть и другие возможности, которые мы здесь не рассматриваем. Дополнительную информацию ищите в документации.

14.4. Переменные окружения

Иногда необходимо обращаться к переменным окружения, которые являются связующим звеном между программой и внешним миром. Переменные окружения — это просто метки, связанные с некоторым текстом (обычно небольшим); в них хранятся, например, пути к файлам, имена пользователей и т.п.

Переменные окружения широко применяются в ОС UNIX. Система Windows (а еще раньше MS-DOS) позаимствовала эту идею у UNIX, поэтому приведенные ниже коды будут работать на обеих платформах.

14.4.1. Чтение и установка переменных окружения

Глобальная константа ENV — это хэш, с помощью которого можно читать и изменять переменные окружения. В примере ниже мы читаем значение переменной PATH (в Windows вместо двоеточия нужно употреблять точку с запятой):

```
mypath = ENV["PATH"]
# А теперь получим массив...
dirs = mypath.split(":")
```

А вот пример установки переменной. Новый процесс мы создали, чтобы проиллюстрировать две вещи. Во-первых, дочерний процесс наследует переменные окружения от своего родителя. Во-вторых, значение переменной окружения, установленное в дочернем процессе, родителю не видно.

```
ENV["alpha"] = "123"
ENV["beta"] = "456"
puts "Родитель: alpha = #{ENV['alpha']}"
puts "Родитель: beta = #{ENV['beta']}"
fork do # Код потомка...
```



```

x = ENV["alpha"]
ENV["beta"] = "789"
y = ENV["beta"]
puts "Потомок: alpha = #{x}"
puts "Потомок: beta  = #{y}"
end
Process.wait
a = ENV["alpha"]
b = ENV["beta"]
puts "Родитель: alpha = #{a}"
puts "Родитель: beta  = #{b}"

```

Программа выводит следующие строки:

```

Родитель: alpha = 123
Родитель: beta  = 456
Потомок: alpha = 123
Потомок: beta  = 789
Родитель: alpha = 123
Родитель: beta  = 456

```

Это следствие того факта, что родитель ничего не знает о переменных окружения своих потомков. Поскольку программа на Ruby обычно выполняется в оболочке, то после ее завершения все сделанные изменения переменных окружения не будут видны в текущей оболочке.

14.4.2. Хранение переменных окружения в виде массива или хэша

Важно понимать, что объект `ENV` — не настоящий хэш, а лишь выглядит как таковой. Например, мы не можем вызвать для него метод `invert`; будет возбуждено исключение `NameError`, поскольку такого метода не существует. Причина такой реализации в том, что существует тесная связь между объектом `ENV` и операционной системой; любое изменение хранящихся в нем значений отражается на состоянии ОС, а такое поведение с помощью простого хэша не смоделируешь.

Однако имеется метод `to_hash`, который вернет настоящий хэш, отражающий текущее состояние:

```

envhash = ENV.to_hash
val2var = envhash.invert

```

Получив такой хэш, мы можем преобразовать его к любому другому виду (например, в массив):

```

envarr = ENV.to_hash.to_a

```

Обратное присваивание объекту `ENV` недопустимо, но при необходимости можно пойти обходным путем:

```

envhash = ENV.to_hash
# Выполняем произвольные операции... и записываем обратно в ENV.
envhash.each {|k,v| ENV[k] = v }

```

14.4.3. Импорт переменных окружения как глобальных переменных

Существует библиотечка `importenv.rb`, которая импортирует все переменные окружения, сопоставляя им глобальные переменные программы:

```
require "importenv"
```

```
# Теперь переменные окружения стали глобальными переменными...
# Например, $PWD и $LOGNAME
```

```
where = $PWD
who = $LOGNAME
puts "В каталоге #{where}, вошел как #{who}"
```

Поскольку библиотека `importenv` пользуется библиотекой `trace_var`, отражение на самом деле двустороннее: если присвоить глобальной переменной новое значение, реальная переменная окружения получит то же значение.

```
require "importenv"
```

```
puts "Мой путь #{ $PATH }"
# Печатается: /usr/local/bin:/usr/bin:/usr/ucb:/etc:.
$PATH = "/ruby-1.8.0:" + $PATH
```

```
puts "Моя переменная $PATH теперь равна #{ENV['PATH']}"
# Печатается: /ruby-1.8.0:/usr/local/bin:/usr/bin:/usr/ucb:/etc:.
```

Еще раз подчеркнем, что любые изменения переменных окружения, выполненные внутри программы на Ruby, не отражаются на их значениях, видимых вне этой программы.

14.5. Сценарии на платформе Microsoft Windows

.....
*Как и на лыжном курорте, где девушки ищут мужей,
 а мужья — девушек, ситуация не так симметрична,
 как может показаться на первый взгляд.
 Алан Линдсей Маккей*

Уже отмечалось, что Ruby больше любит ОС UNIX. В каком-то смысле это правда: язык разрабатывался в среде UNIX, в ней лучше всего и работает. Сейчас он, впрочем, перенесен на другие платформы, в том числе на Macintosh; ведется даже работа по переносу на Palm OS. Но если UNIX — основная платформа, то следующая по значимости — Windows.

Пользователи Windows не брошены на произвол судьбы. Существует немало инструментов и библиотек для этой платформы, а разрабатывается еще больше. Многие аспекты Ruby, даже механизм потоков, изначально не зависят от платформы. Наибольшие трудности возникают при управлении процессами, выполнении ввода/вывода и других операций низкого уровня.

В прошлом существовало несколько вариантов Ruby для Windows. Интерпретатор мог быть собран компилятором gcc или Visual C, его работа могла зависеть от наличия библиотеки Cygwin DLL и т.д. Но в последние годы появился «моментальный» инсталлятор для Windows (см. раздел 14.6).

Среда изменяется слишком быстро, чтобы можно было ее сейчас документировать, однако в этом разделе мы все же рассмотрим некоторые вопросы написания сценариев и автоматизации на платформе Windows. Описанные приемы и утилиты должны работать в любой ОС. Если возникнут проблемы, сообщество придет на помощь.

14.5.1. Расширение Win32API

Расширение win32API – исключительно мощный инструмент, если вы собираетесь программировать на относительно низком уровне. Оно позволяет вызывать из Ruby функции Windows API, находящиеся в любой DLL.

Указанная функция становится объектом, а методу new передаются параметры, точно описывающие функцию. Первый параметр – строка, идентифицирующая DLL, в которой находится функция (например, `crt.dll`). Второй параметр – имя самой функции, третий – массив строк, описывающих типы параметров функции (массив импорта), а четвертый – строка, описывающая тип возвращаемого значения (строка экспорта).

Массив импорта может содержать следующие значения (регистр не играет роли):

I	целое
L	число
N	число
P	указатель на строку

Строка экспорта также может содержать любое из этих значений, а также значение «V», означающее «void».

После того как объект создан, можно обратиться к его методу `call` для вызова функции Windows. Синоним – `call`.

В примере ниже мы вызываем функцию `GetCursorPos`, которая возвращает указатель на структуру `POINT`. Эта структура состоит из двух полей типа `long`. Чтобы получить их значения, мы можем воспользоваться методом `unpack`:

```
require 'Win32API'
result = "0"*8 # Восемь байтов (достаточно для двух long).
getCursorXY = Win32API.new("user32", "GetCursorPos", ["P"], "V")
getCursorXY.call(result)
x, y = result.unpack("LL") # Два long.
```

В данном случае функция вернула составные двоичные данные, а иногда такие данные нужно подать на вход функции. Понятно, что для этого нужно воспользоваться методом `pack`, который упакует данные в строку.

У описанной техники может быть много применений. Еще два примера приведены в разделах 10.1.20 и 14.1.1.

14.5.2. Расширение Win32OLE

Расширение win32OLE (правильно писать его имя строчными буквами: `win32ole`) реализует интерфейс к OLE-автоматизации в Windows. Программа на Ruby может выступать в роли клиента любого сервера автоматизации, к числу которых относятся, например, Microsoft Word, Outlook, Internet Explorer, а также многие продукты третьих фирм.

Для того чтобы начать взаимодействие с внешним приложением, мы создаем объект класса WIN32OLE. С его помощью мы получим доступ ко всем свойствам и методам, которые раскрывает данное приложение. В примере ниже объект ассоциируется с редактором Microsoft Word. Атрибуту visible мы присвоим значение true, а в конце вызовем метод quit, чтобы завершить внешнюю программу.

```
require "win32ole"

word = WIN32OLE.new "Word.Application"

word.visible = true

# ...

word.quit
```

Свойства сервера автоматизации выглядят как атрибуты объекта. Их можно читать и устанавливать.

Имеется и альтернативная нотация, в которой для доступа к свойствам используется конструкция, напоминающая хэш.

```
player["FileName"] = "file.wav"
name = player["FileName"]
# Эквивалентно следующим предложениям:
# player.FileName = "file.wav"
# name = player.FileName
```

У этой нотации есть то преимущество, что она позволяет проще осуществлять динамический доступ к свойствам, как показано в искусственном примере ниже:

```
puts "Введите имя свойства"
prop = gets
puts "Введите новое значение"
val = gets
old = obj[prop]
obj[prop] = val
puts "#{prop} было #{old}... стало #{obj[prop]}"
```

Но обратимся к более жизненным примерам. Следующий код получает от пользователя имя файла, передает его Microsoft Word и распечатывает файл:

```
require "win32ole"

print "Введите имя файла для распечатки: "
docfile = gets

word = WIN32OLE.new "Word.Application"
word.visible = true
word.documents.open docfile
word.options.printBackground = false
# Можно было бы также установить свойство printBackground в true,
# но тогда пришлось бы дожидаться, пока весь файл будет
# скопирован в буфер принтера, и только потом вызывать quit...
```

```
word.activeDocument.printOut  
word.quit
```

В следующем примере проигрывается WAV-файл. Недостаток заключается в том, что в конце программы мы поставили `sleep` на случайно выбранное время, а не просто ждем, когда воспроизведение закончится. Предлагаем читателю устранить этот недочет в качестве упражнения.

```
require "win32ole"  
  
sound = WIN32OLE.new("MCI.MMcontrol")  
  
wav = "c:\\windows\\media\\tada.wav"  
sound.fileName = wav  
  
sound.autoEnable = true  
  
sound.command = "Open"  
sound.command = "Play"  
  
sleep 7
```

В листинге 14.2 мы просим Internet Explorer открыть диалог для ввода текста.

Листинг 14.2. Открытие диалога для ввода текста в браузере

```
require "win32ole"  
  
def ieInputBox( msg, default )  
  ie = WIN32OLE.new("InternetExplorer.Application");  
  ie.visible = false  
  ie.navigate "about:blank"  
  sleep 0.01 while (ie.busy)  
  
  script = ie.Document.Script;  
  result = script.prompt(msg,default);  
  ie.quit  
  
  result  
end  
  
# Главная программа...  
  
result = ieInputBox( "Введите свое имя",  
                    "Дэйв Боумэн")  
  
if result  
  puts result  
else  
  puts "Пользователь нажал Cancel"  
end
```

В листинге 14.3 мы открываем IE в небольшом окне и выводим в него HTML-документ.

Листинг 14.3. Для вывода в окно браузера требуется win32ole

```
html = <<EOF
<html>
<body>
<h3>А теперь что-нибудь</h3>
<h2>совсем</h2>
<h1>другое...</h1>
</body>
</html>
EOF

ie = WIN32OLE.new("InternetExplorer.Application");

ie.left      = 150
ie.top       = 150
ie.height    = 200
ie.width     = 300
ie.menubar   = 0
ie.toolbar   = 0
ie.navigate "about:blank"
ie.visible=TRUE;

ie.document.open
ie.document.write html
ie.document.close
sleep 5
ie.quit
```

В следующем примере открывается диалоговое окно, где пользователь может выбрать файл из списка:

```
require "win32ole"

cd = WIN32OLE.new("MSComDlg.CommonDialog")

# Задать фильтр файлов
cd.filter = "All Files (*.*)|*.*" +
            "|Ruby Files (*.rb)|*.rb"
cd.filterIndex = 2

cd.maxFileSize = 128      # Установить MaxFileSize.

cd.showOpen()
file = cd.fileName        # Получить путь к файлу.
if not file or file==" "
  puts "Файл не выбран."
else
```

```
puts "Пользователь выбрал: #{file}\n"
end
```

И, наконец, определим IP-адрес своего компьютера:

```
require "win32ole"
```

```
ws = WIN32OLE.new "MSWinsock.Winsock"
```

```
# Получить свойство LocalIP
```

```
ipAddress = ws.localIP
```

```
puts "Локальный IP-адрес равен : #{ipAddress}"
```

Как видите, возможности не ограничены. Развлекайтесь и не забывайте делиться своими программами с другими!

14.5.3. Использование ActiveScriptRuby

Наверняка вам приходилось открывать в браузере Internet Explorer страницы, содержащие код на языке JavaScript или VBScript. (Мы не будем здесь касаться различий между JScript и JavaScript.)

Но сценарий можно написать и на языке ActiveScriptRuby, представляющем собой мост между COM и Ruby. Вот как можно включить код на Ruby в HTML-страницу (листинг 14.4).

Листинг 14.4. Код на Ruby, встроенный в HTML-страницу

```
<html>

<script language="RubyScript">
  # Это код на Ruby...
  def helloMethod
    @window.alert "Работает Ruby!"
  end
</script>

<body>

  Это кнопка...
  <input id=hello type=button onclick="helloMethod"
    language="RubyScript">

</body>
</html>
```

С помощью той же техники можно вызывать написанный на Ruby код из любого Windows-приложения, поддерживающего интерфейс IActiveScript, например из Explorer или WScript (исполняемый файл называется WSH). Дополнительную информацию вы можете найти на странице arton (<http://www.geocities.co.jp/SiliconValley-PaloAlto/9251/rubymain.html>).

14.6. Моментальный инсталлятор для Windows

С точки зрения пользователей Microsoft Windows одним из самых значительных шагов в развитии Ruby за последние годы стал так называемый «моментальный инсталлятор» (one-click installer). Главным разработчиком этого проекта (официально он называется Ruby Installer) является Курт Гиббс (Curt Hibbs). Процедура инсталляции выполнена в «родном» для Windows стиле.

Инсталлятор особенно ценен тем, что работает в полном соответствии с ожиданиями пользователей Windows. Он имеет графический интерфейс и выполняет шаги установки в строго определенном порядке. Разумеется, инсталлируется двойная версия, так что компилятор не нужен. Но это не единственные его привлекательные черты.

Устанавливаемый дистрибутив весьма полон («батарейки в комплекте»). Он включает не только интерпретатор Ruby со всеми системными классами и стандартными библиотеками, но и целый ряд дополнительных библиотек и приложений. Многие из них предназначены исключительно для платформы win32.

Устанавливаются следующие компоненты (некоторые из них необязательны):

- сам интерпретатор Ruby (пакет `ruby-mswin32` и пакет `RubySrc` для тех, кто хочет познакомиться с исходными текстами на языке C);
- два часто используемых приложения: `RubyGems` и `rake`;
- бесплатная копия книги Дейва Томаса (Dave Thomas) и Энди Ханта (Andy Hunt) «Programming Ruby» – первое издание в формате Windows Help;
- библиотека `fxruby` (обычно несколько версий), реализующая привязки к комплекту инструментов для создания графических интерфейсов FOX;
- инструменты для разработки приложений трехмерной графики OpenGL и GLUT;
- утилиты `fxirb` и `fxri` – графические версии программ `irb` и `ri`, написанные с применением библиотеки `FXRuby`;
- `FreeRIDE` – интегрированная среда разработки для Ruby с встроенным редактором, обозревателем исходных текстов и отладчиком (работа над совершенствованием этой программы ведется постоянно);
- `SciTE` – текстовый редактор на базе `Scintilla`;
- `SWin` и `VRuby` – инструменты для обработки сообщений Windows и разработки графических интерфейсов (обе являются частью проекта `VisualuRuby`, во главе которого стоит Ясухира Насикава);
- два анализатора XML (`XMLParser` и `Expat`), а также `HTMLParser`;
- библиотеки для работы с базами данных `RubyDBI` и `DBD/ODBC`;
- прочие библиотеки и инструменты, в том числе `log4r`, `zlib`, `OpenSSL`, `Iconv`, `readline` и другие.

Планируются, но еще не готовы варианты этого инсталлятора и для других платформ.

14.7. Библиотеки, о которых полезно знать

Если вы программируете на Ruby в Windows, вам абсолютно необходим пакет, созданный Дэниэлем Бергером (Daniel Berger), одним из самых известных специалистов по Ruby на этой платформе. Библиотека `win32-utils` – в действительности целый набор мелких библиотек. Мы не можем рассмотреть их все подробно, но хотя бы перечислим.

- `win32-changenotify` – для мониторинга событий файловой системы;
- `win32-clipboard` – для взаимодействия с буфером обмена Windows;
- `win32-etc` – предоставляет аналоги таких UNIX-функций, как `getpwnam` и `getpwuid`;
- `win32-event` – интерфейс с событиями Windows (объектами Event);
- `win32-eventlog` – интерфейс с журналом событий;
- `win32-ipc` – базовый класс для всех объектов синхронизации в Windows (используется в библиотеке `win32-event` и др.);
- `win32-mmap` – интерфейс к файлам, проецируемым на память, в Windows;
- `win32-open3` – библиотека `open3` для Windows (запустить команды и получить три описателя файлов);
- `win32-pipe` – именованные каналы в Windows;
- `win32-process` – реализация для Windows методов `fork`, `wait` и `kill`, имеющих в UNIX;
- `win32-sapi` – интерфейс к Microsoft Speech API;
- `win32-service` – интерфейс к службам Windows;
- `win32-shortcut` – интерфейс для создания и модификации ярлыков в Windows;
- `win32-sound` – интерфейс для воспроизведения звуковых файлов в Windows;

Вот еще несколько библиотек, которые полезно иметь под рукой:

- `Win32::Console` – это перенос пакетов `Win32::Console` и `Win32::Console::ANSI`, первоначально написанных на языке Perl. Эта библиотека значительно упрощает работу с консолью в Windows (изменение цветов, позиционирование курсора, запрос информации и эмуляцию управляющих символов ANSI);
- `ActiveDirectory` позволяет легко взаимодействовать с экземплярами Active Directory, работающими на серверах под управлением Microsoft Windows;
- `ruby-inifile` позволяет работать с ini-файлами (читать, разбирать и обновлять их).

В сети есть еще много библиотек, которые могут вам пригодиться. Ищите их на сайтах <http://raa-ruby-lang.org> и <http://rubyforge.org>.

14.8. Работа с файлами, каталогами и деревьями

При выполнении рутинных задач приходится много работать с файлами и каталогами, в том числе с целыми иерархиями каталогов. Немало материала на эту тему вошло в главу 4, но кое-какие важные моменты мы хотим осветить здесь.

Поскольку ввод/вывод – вещь системно-зависимая, то для различных систем приходится применять разные приемы. Если сомневаетесь, экспериментируйте!..

14.8.1. Несколько слов о текстовых фильтрах

Многие инструменты, которыми мы постоянно пользуемся (как поставляемые производителем, так и разрабатываемые собственными силами), – просто текстовые фильтры. Иными словами, они принимают на входе текст, каким-то образом преобразуют его и выводят. Классическими примерами текстовых фильтров в UNIX служат, в частности, программы `sed` и `tr`.

Иногда файл настолько мал, что целиком помещается в памяти. В этом случае возможны такие виды обработки, которые по-другому было бы сложно реализовать.

```
file = File.open(filename)
lines = file.readlines
# Какие-то операции...
lines.each { |x| puts x }
```

Бывает, что нужно обрабатывать файл построчно.

```
IO.foreach(filename) do |line|
  # Какие-то операции...
  puts line
end
```

Наконец, не забывайте, что все имена файлов, указанные в командной строке, автоматически собираются в объект `ARGV`, представляющий конкатенацию всех выходных данных (см. раздел 14.2.2). Мы можем вызывать, к примеру, метод `ARGV.readlines`, как если бы `ARGV` был объектом класса `IO`. Вся выходная информация будет, как обычно, направлена на стандартный вывод.

14.8.2. Копирование дерева каталогов (с символическими ссылками)

Пусть нужно скопировать целое дерево каталогов в новое место. Сделать это можно по-разному, но если в дереве есть символические ссылки, задача усложняется.

В листинге 14.5 приведено рекурсивное решение. Оно достаточно дружелюбно – контролирует входные данные и выводит информацию о порядке запуска.

Листинг 14.5. Копирование дерева каталогов

```
require "fileutils"

def recurse(src, dst)
  Dir.mkdir(dst)
  Dir.foreach(src) do |e|
    # Пропустить . и ..
```

```

next if [".",".."].include? e
fullname = src + "/" + e
newname = fullname.sub(Regexp.new(Regexp.escape(src)),dst)
if FileTest::directory?(fullname)
  recurse(fullname,newname)
elsif FileTest::symlink?(fullname)
  linkname = 'ls -l #{fullname}'.sub(/.* -> /,"").chomp
  newlink = linkname.dup
  n = newlink.index($oldname)
  next if n == nil
  n2 = n + $oldname.length - 1
  newlink[n..n2] = $newname
  newlink.sub!(/\\\/\\\/,"/")
  # newlink = linkname.sub(Regexp.new(Regexp.escape(src)),dst)
  File.symlink(newlink, newname)
elsif FileTest::file?(fullname)
  FileUtils.copy(fullname, newname)
else
  puts "??? : #{fullname}"
end
end
end

# "Главная программа"

if ARGV.size != 2
  puts "Usage: copytree oldname newname"
  exit
end

oldname = ARGV[0]
newname = ARGV[1]

if ! FileTest::directory?(oldname)
  puts "Ошибка: первый параметр должен быть именем существующего каталога."
  exit
end

if FileTest::exist?(newname)
  puts "Ошибка: #{newname} уже существует."
  exit
end

oldname = File.expand_path(oldname)
newname = File.expand_path(newname)

$oldname=oldname
$newname=newname

recurse(oldname, newname)

```

Возможно, и существуют варианты UNIX, в которых команда `cp -R` сохраняет символические ссылки, но нам о них ничего не известно. Программа, показанная в листинге 14.5, была написана для решения этой практической задачи.

14.8.3. Удаление файлов по времени модификации и другим критериям

Предположим, вы хотите удалить самые старые файлы из какого-то каталога. В нем могут, к примеру, храниться временные файлы, протоколы, кэш браузера и т. п.

Ниже представлена небольшая программа, удаляющая файлы, которые в последний раз модифицировались раньше указанного момента (заданного в виде объекта `Time`):

```
def delete_older(dir, time)
  Dir.chdir(dir) do
    Dir.foreach(".") do |entry|
      # Каталоги не обрабатываются.
      next if File.stat(entry).directory?
      # Используем время модификации.
      if File.mtime(entry) < time
        File.unlink(entry)
      end
    end
  end
end

delete_older("/tmp", Time.local(2001, 3, 29, 18, 38, 0))
```

Неплохо, но можно обобщить. Создадим метод `delete_if`, который принимает блок, возвращающий значение `true` или `false`. И будем удалять те и только те файлы, которые удовлетворяют заданному критерию.

```
def delete_if(dir)
  Dir.chdir(dir) do
    Dir.foreach(".") do |entry|
      # Каталоги не обрабатываются.
      next if File.stat(entry).directory?
      if yield entry
        File.unlink(entry)
      end
    end
  end
end

# Удалить файлы длиннее 3000 байтов.
delete_if("/tmp") { |f| File.size(f) > 3000 }

# Удалить файлы с расширениями LOG и BAK.
delete_if("/tmp") { |f| f =~ /(log|bak)$/i }
```

14.8.4. Вычисление свободного места на диске

Пусть нужно узнать, сколько байтов свободно на некотором устройстве. В следующем примере это делается по-простому, путем запуска системной утилиты:

```
def freespace(device=".")
  lines = %x(df -k #{device}).split("\n")
  n = lines.last.split[1].to_i * 1024
end
```

```
puts freespace("/tmp")      # 16772204544
```

Эту задачу лучше решать, обернув метод `statfs` в расширение Ruby. Такие попытки в прошлом предпринимались, но, похоже, проект умер.

Для Windows имеется несколько более элегантное решение (предложено Дэниэлем Бергером):

```
require 'Win32API'
```

```
GetDiskFreeSpaceEx = Win32API.new('kernel32', 'GetDiskFreeSpaceEx',
                                   'PPPP', 'I')
```

```
def freespace(dir=".")
  total_bytes = [0].pack('Q')
  total_free  = [0].pack('Q')
  GetDiskFreeSpaceEx.call(dir, 0, total_bytes, total_free)

  total_bytes = total_bytes.unpack('Q').first
  total_free  = total_free.unpack('Q').first
end
```

```
puts freespace("C:")      # 5340389376
```

Этот код должен работать во всех вариантах Windows.

14.9. Различные сценарии

Приведем еще несколько примеров. Не претендуя на оригинальность, мы отнесли их к категории «разное».

14.9.1. Ruby в виде одного файла

Иногда нужно быстро или временно установить Ruby. Или даже включить Ruby в состав собственной программы, поставляемой в виде одного исполняемого файла.

Мы уже познакомились с «моментальным инсталлятором» Ruby для Windows. Существуют планы (пока еще не оформившиеся) создать подобный инсталлятор для Linux и Mac OS X.

Эрик Веенстра (Erik Veenstra) недавно добился значительных успехов в создании пакетов, включающих как Ruby, так и написанные на нем приложения. Он автор пакетов AllInOneRuby, Tar2RubyScript и RubyScript2Exe (все они есть на его сайте <http://www.erikveen.dds.nl>).

AllInOneRuby – это дистрибутив Ruby в одном файле. В пакет входят интерпретатор Ruby, системные классы и стандартные библиотеки, упакованные в единый архив, который легко перемещать или копировать. Например, его можно записать на USB-диск, носить в кармане и «установить» на любую машину за считанные секунды. Работает AllInOneRuby на платформах Windows и Linux; имеется также экспериментальная поддержка для Mac OS X.

Что такое Tar2RubyScript, следует из самого названия. Программа получает на входе дерево каталогов и создает самораспаковывающийся архив, включающий написанную на Ruby программу и архив в формате tar. Идея та же, что у JAR-файлов в языке Java. Запускаемый сценарий должен называться `init.rb`; если сохраняется библиотека, а не автономное приложение, этот файл можно опустить.

Название RubyScript2Exe, наверное, не вполне удачно. Программа действительно преобразует написанное на Ruby приложение в один двоичный файл, однако работает она не только в Windows, но и в Linux и Mac OS X. Можете называть ее компилятором, хотя в действительности она им, конечно, не является. Она собирает файлы, являющиеся частью установленного дистрибутива Ruby на вашей машине, поэтому не нуждается в кросс-компиляции (даже если бы такая возможность имелась). Имейте в виду, что исполняемый файл «усечен» в том смысле, что неиспользуемые библиотеки Ruby в него не включаются.

Архив, созданный программой Tar2RubyScript, можно запустить на любой машине, где установлен Ruby (и программы, которые необходимы самому приложению). RubyScript2Exe не имеет такого ограничения, поскольку включает (наряду с вашим приложением) интерпретатор Ruby, всю среду исполнения и все необходимые внешние программы. Можете использовать эти инструменты вместе или порознь.

14.9.2. Подача входных данных Ruby по конвейеру

Поскольку интерпретатор Ruby – это однопроходный транслятор, можно подать ему на вход некий код и выполнить его. Это может оказаться полезным, когда обстоятельства вынуждают вас работать на традиционном языке сценариев, но для каких-то сложных задач вы хотите применить Ruby.

В листинге 14.6 представлен `bash`-сценарий, который вызывает Ruby (посредством вложенного документа) для вычисления интервала в секундах между двумя моментами времени. Ruby-программа печатает на стандартный вывод одно значение, которое перехватывается вызывающим сценарием.

Листинг 14.6. `bash`-сценарий, вызывающий Ruby

```
#!/usr/bin/bash

# Для вычисления разницы в секундах между двумя моментами времени
# bash вызывает Ruby...

export time1="2007-04-02 15:56:12"
export time2="2007-12-08 12:03:19"

cat <<EOF | ruby | read elapsed
```

```
require "parsedate"

time1 = ENV["time1"]
time2 = ENV["time2"]

args1 = ParseDate.parsedate(time1)
args2 = ParseDate.parsedate(time2)

args1 = args1[0..5]
args2 = args2[0..5]

t1 = Time.local(*args1)
t2 = Time.local(*args2)

diff = t2 - t1
puts diff
EOF
```

```
echo "Прошло секунд = " $elapsed
```

В данном случае оба исходных значения передаются в виде переменных окружения (которые необходимо экспортировать). Строки, читающие эти значения, можно было бы записать так:

```
time1="$time1" # Включить переменные оболочки непосредственно
time2="$time2" # в строку...
```

Но возникающие при этом проблемы очевидны. Очень трудно понять, имеется ли в виду переменная `bash` или глобальная переменная `Ruby`. Возможна также путаница при экранировании и расстановке кавычек.

Флаг `-e` позволяет создавать однострочные `Ruby`-сценарии. Вот пример обращения строки:

```
#!/usr/bin/bash

string="Francis Bacon"

ruby -e "puts '$string'.reverse" | read reversed

# $reversed теперь равно "nocat sircarF"
```

Знаатоки `UNIX` заметят, что `awk` использовался подобным образом с незапамятных времен.

14.9.3. Получение и установка кодов завершения

Метод `exit` возбуждает исключение `SystemExit` и в конечном счете возвращает указанный код завершения операционной системе (или тому, кто его вызвал). Этот метод определен в модуле `Kernel`. Метод `exit!` отличается от него в двух отношениях: он не выполняет зарегистрированные обработчики завершения и по умолчанию возвращает `-1`.

```
# ...
if (all_OK)
```

```

exit      # Нормально (0).
else
  exit!    # В спешке (-1).
end

```

Когда операционная система печатает возвращенный Ruby код (например, выполнив команду `echo $?`), мы видим то же самое число, что было указано в программе. Если завершается дочерний процесс, то код его завершения, полученный с помощью метода `wait2` (или `waitpid2`), будет сдвинут влево на восемь битов. Это причуда стандарта POSIX, которую Ruby унаследовал.

```
child = fork { sleep 1; exit 3 }
```

```

pid, code = Process.wait2      # [12554,768]
status = code << 8             # 3

```

14.9.4. Работает ли Ruby в интерактивном режиме?

Чтобы узнать, работает ли программа в интерактивном режиме, нужно проверить стандартный ввод. Метод `isatty?` возвращает `true`, если устройство интерактивное, а не диск или сокет. (Для Windows этот метод не реализован.)

```

if STDIN.isatty?
  puts "Привет! Я вижу, вы печатаете"
  puts "на клавиатуре."
else
  puts "Входные данные поступают не с клавиатуры."
end

```

14.9.5. Определение текущей платформы или операционной системы

Если программа хочет знать, в какой операционной системе выполняется, то может опросить глобальную константу `RUBY_PLATFORM`. В ответ будет возвращена загадочная строка (что-то вроде `i386-cygwin` или `sparc-solaris2.7`), содержащая информацию о платформе, для которой был собран интерпретатор Ruby.

Поскольку мы в основном работаем с вариантами UNIX (Solaris, AIX, Linux) и Windows (98, NT, 2000, XP), то считаем полезным следующий очень грубый код. Он отличает UNIX от Windows (бесцеремонно отправляя всех остальных в категорию «прочие»).

```

def os_family
  case RUBY_PLATFORM
  when /ix/i, /ux/i, /gnu/i,
       /sysv/i, /solaris/i,
       /sunos/i, /bsd/i
    "unix"
  when /win/i, /ming/i
    "windows"
  else
    "other"
  end
end

```


Этот небольшой набор регулярных выражений корректно распознает абсолютное большинство платформ. Конечно, это весьма неуклюжий способ обработки системных зависимостей. Даже если вы правильно определите семейство ОС, откуда еще не следует, что нужная вам функциональность имеется (или отсутствует).

14.9.6. Модуль Etc

Модуль `Etc` получает различную информацию из файлов `/etc/passwd` и `/etc/group`. Понятно, что полезен он только на платформе UNIX.

Метод `getlogin` возвращает имя пользователя, от имени которого запущена программа. Если он завершается неудачно, может помочь метод `getpwnid` (принимаящий в качестве необязательного параметра идентификатор пользователя `uid`).

```
myself = getlogin # hal9000
myname = getpwnid(2001).name # hal9000
```

```
# Если параметр не задан, getpwnid вызывает getuid...
me2 = getpwnid.name # hal9000
```

Метод `getpwnam` возвращает структуру `passwd`, которая содержит поля `name`, `dir` (начальный каталог), `shell` (начальный интерпретатор команд) и др.

```
rootshell = getpwnam("root").shell # /sbin/sh
```

Методы `getgrgid` и `getgrnam` ведут себя аналогично, но по отношению к группам. Они возвращают структуру `group`, содержащую имя группы и т.д.

Итератор `passwd` обходит все записи в файле `/etc/passwd`. Запись передается в блок в виде структуры `passwd`.

```
all_users = []
passwd { |entry| all_users << entry.name }
```

Имеется также итератор `group` для обхода записей в файле `/etc/group`.

14.10. Заключение

На этом мы завершаем обсуждение применения Ruby для решения рутинных задач автоматизации. Мы видели, как передавать в программу и получать от нее информацию в виде переменных окружения и с помощью стандартного ввода/вывода. Мы познакомились с типичными операциями «склеивания», позволяющими разным программам взаимодействовать. Рассмотрели мы и различные уровни взаимодействия с операционной системой.

Поскольку значительная часть изложенного материала системно зависима, я призываю вас экспериментировать. Между платформами Windows и UNIX имеются серьезные отличия. Есть они и между различными операционными системами, относящимися к одному семейству.

Следующая тема, которую мы рассмотрим, тоже весьма широка. Речь пойдет о работе с данными в разных форматах, от графических до XML.

Глава 15. Ruby и форматы данных

.....
– Ваша информация, сэр, – говорит Библиотекарь.
– Сможешь увязать эту информацию
с утилитой «ВЫ ЗДЕСЬ»? – говорит Хиро.
– Я посмотрю, что можно сделать, сэр.
Форматы представляются совместимыми.
Нил Стивенсон, «Лавина»

В этой главе мы расскажем обо всем понемножку. Значительную часть материала можно было бы поместить и в другое место. Не все рассмотренные вопросы одинаковы по важности и сложности, но каждому из них стоит уделить внимание.

В программировании так уж получается, что по мере усложнения какой-то информации для ее описания вырабатывается отдельный «мини-язык», а чаще даже несколько таких языков. Мы называем их форматами файлов или данных.

Любой из вас припомнит сотни примеров форматов файлов. Это и графические форматы типа JPG, GIF и PNG, и форматы документов (RTF и PDF), и «универсальные» форматы (CSV, XML или YAML) и бесчисленные форматы, разработанные отдельными компаниями, многие из которых являются просто вариациями на тему хранения данных в виде таблицы с фиксированной шириной колонок, столь популярного в древние времена (я имею в виду 1960-е годы).

Один из самых простых и наиболее употребительных форматов данных – обычный текст. Но даже на такой формат можно наложить ту или иную структуру (отсюда и популярность XML). Бывают также чисто двоичные и двоично-текстовые форматы. В принципе можно было бы разработать «иерархию» форматов, подобную сетевой модели ISO, в которой информация представляется по-разному на разных уровнях протоколов.

Но в каком бы формате данные ни хранились, рано или поздно их придется читать, разбирать и снова сохранять. В этой главе мы рассмотрим лишь несколько самых распространенных форматов файлов; в одной книге невозможно охватить все существующие. Если вы хотите разбирать файлы в таких форматах, как vCard, iCal и пр., то придется поискать соответствующие библиотеки или, быть может, написать свою собственную.

15.1. Разбор XML и REXML

Язык XML (который внешне «похож» на HTML или SGML) стал популярен в 1990-х годах. Благодаря некоторым свойствам он действительно лучше таблицы

с фиксированной шириной колонки. Например, он позволяет задавать имена полей, представлять иерархически организованные данные и, самое главное, хранить данные переменной длины.

Конечно, сорок лет назад XML был бы невозможен из-за ограничений на объем памяти. Но представим себе, что он появился бы тогда. Знаменитая проблема 2000 года, которой пресса уделяла так много внимания в 1999 году (хотя проблема-то и яйца выеденного не стоила!) при наличии XML вообще не возникла бы. Ведь причина была в том, что в унаследованных системах данные хранились в формате с фиксированной длиной. Так что, несмотря на некоторые недостатки, у XML есть сферы применения. В Ruby для работы с XML чаще всего применяется библиотека *REXML*, написанная Шоном Расселом (Sean Russell). Начиная с 2002 года REXML (произносится «рекс-эм-эль») входит в стандартный дистрибутив Ruby.

Сразу отмечу, что REXML работает довольно медленно. Достаточно ли ее быстроедействие для вашего конкретного приложения, решать вам. Не исключено, что со временем вам придется перейти на библиотеку *libxml2* (которую мы здесь не рассматриваем). Она, конечно, работает очень быстро (поскольку написана на C), но, пожалуй, не так близка по духу к Ruby.

REXML – это процессор XML, написанный целиком на Ruby в полном соответствии со стандартом XML 1.0. Он не проверяет достоверность документа (соответствие схеме) и удовлетворяет всем тестам OASIS (Organization for the Advancement of Structured Information Standards – организация по внедрению стандартов структурирования информации) для таких процессоров.

Библиотека REXML предлагает несколько API. Сделано это, конечно, для того, чтобы обеспечить большую гибкость, а не внести путаницу. Два классических API – интерфейсы на базе DOM (объектной модели документа) и SAX (поточный интерфейс). В первом случае весь документ считывается в память и хранится в древовидной форме. Во втором разбор осуществляется по мере чтения документа. Этот способ не требует загрузки документа в память и потому применяется, когда документ слишком велик, а память ограничена.

Во всех примерах мы будем использовать один и тот же XML-файл (см. листинг 15.1), представляющий часть описания личной библиотеки.

Листинг 15.1. Файл *books.xml*

```
<library shelf="Recent Acquisitions">
  <section name="Ruby">
    <book isbn="0672328844">
      <title>The Ruby Way</title>
      <author>Hal Fulton</author>
      <description>Second edition. The book you are now reading.
        Ain't recursion grand?
      </description>
    </book>
  </section>
  <section name="Space">
    <book isbn="0684835509">
```

```

<title>The Case for Mars</title>
<author>Robert Zubrin</author>
<description>Pushing toward a second home for the human
               race.
</description>
</book>
<book isbn="074325631X">
  <title>First Man: The Life of Neil A. Armstrong</title>
  <author>James R. Hansen</author>
  <description>Definitive biography of the first man on
               the moon.
  </description>
</book>
</section>
</library>

```

15.1.1. Дрезовидное представление

Сначала покажем, как работать с XML-документом, представленным в виде дерева. Для начала затребуем библиотеку `rexml/document`; обычно для удобства мы включаем также директиву `include REXML`, чтобы импортировать все необходимое в пространство имен верхнего уровня. В листинге 15.2 продемонстрировано несколько полезных приемов.

Листинг 15.2. Разбор документа с применением DOM

```

require 'rexml/document'
include REXML

input = File.new("books.xml")
doc = Document.new(input)

root = doc.root
puts root.attributes["shelf"]      # Недавние приобретения

doc.elements.each("library/section") { |e| puts e.attributes["name"] }
# Выводится:
#   Ruby
#   Space

doc.elements.each("*/section/book") { |e| puts e.attributes["isbn"] }
# Выводится:
#   0672328844
#   0321445619
#   0684835509
#   074325631X

sec2 = root.elements[2]
author = sec2.elements[1].elements["author"].text      # Robert Zubrin

```

Обратите внимание: атрибуты представляются в виде хэша. Обращаться к элементам можно либо по пути, либо по номеру. В последнем случае учтите, что согласно спецификации XML индексация элементов начинается с 1, а не с 0, как в Ruby.

15.1.2. Поточковый разбор

А теперь попробуем разобрать тот же самый файл в потоковом стиле (на практике это вряд ли понадобилось бы, потому что размер файла невелик). У этого подхода несколько вариантов, в листинге 15.3 показан один из них. Идея в том, чтобы определить класс слушателя, методы которого анализатор будет вызывать для обработки событий.

Листинг 15.3. SAX-разбор

```
require 'rexml/document'
require 'rexml/streamlistener'
include REXML

class MyListener
  include REXML::StreamListener
  def tag_start(*args)
    puts "tag_start: #{args.map {|x| x.inspect}.join(', ')}"
  end

  def text(data)
    return if data =~ /^~w*$/      # Ничего, кроме пропусков.
    abbrev = data[0..40] + (data.length > 40 ? "..." : "")
    puts "  text    :   #{abbrev.inspect}"
  end
end

list = MyListener.new
source = File.new "books.xml"
Document.parse_stream(source, list)
```

В этом нам поможет класс `StreamListener`; сам по себе он содержит только заглушки, то есть пустые методы обратного вызова. Вы должны переопределить их в своем подклассе. Когда анализатор встречает открывающий тег, он вызывает метод `tag_open`. Можете считать это чем-то вроде метода `method_missing`, которому в качестве параметра передается имя тега (и все его атрибуты в форме хэша). Аналогично работает метод `text`; о других методах вы можете прочитать в документации на сайте <http://ruby-doc.org> или в каком-нибудь другом месте.

Программа в листинге 15.3 протоколирует обнаружение каждого открывающего и каждого закрывающего тега. Результат работы показан в листинге 15.4 (для краткости текст приведен не полностью).

Листинг 15.4. Результат работы программы потокового разбора

```
tag_start: "library", {"shelf"=>"Recent Acquisitions"}
tag_start: "section", {"name"=>"Ruby"}
```

```

tag_start: "book", {"isbn"=>"0672328844"}
tag_start: "title", {}
  text    :   "The Ruby Way"
tag_start: "author", {}
  text    :   "Hal Fulton"
tag_start: "description", {}
  text    :   "Second edition. The book you are now read..."
tag_start: "section", {"name"=>"Space"}
tag_start: "book", {"isbn"=>"0684835509"}
tag_start: "title", {}
  text    :   "The Case for Mars"
tag_start: "author", {}
  text    :   "Robert Zubrin"
tag_start: "description", {}
  text    :   "Pushing toward a second home for the huma..."
tag_start: "book", {"isbn"=>"074325631X"}
tag_start: "title", {}
  text    :   "First Man: The Life of Neil A. Armstrong"
tag_start: "author", {}
  text    :   "James R. Hansen"
tag_start: "description", {}
  text    :   "Definitive biography of the first man on ..."

```

15.1.3. XPath и другие интерфейсы

Альтернативным способом работы с XML-документом является язык XPath, с помощью которого описывается, как обратиться к конкретным элементам и атрибутам XML-документа.

Библиотека REXML поддерживает XPath с помощью класса `xPath`. Предполагается, что документ представлен в виде DOM (см. выше листинг 15.2). Рассмотрим следующий код:

```

# (Этап подготовки опущен.)
book1 = XPath.first(doc, "//book")    # Найдена информация о первой книге.
p book1

# Распечатать названия всех книг.
XPath.each(doc, "//title") { |e| puts e.text }

# Получить массив всех элементов "author".
names = XPath.match(doc, "//author").map { |x| x.text }
p names

```

Вот что он напечатает:

```

<book isbn='0672328844'> ... </>
The Ruby Way
The Case for Mars
First Man: The Life of Neil A. Armstrong
["Hal Fulton", "Robert Zubrin", "James R. Hansen"]

```

REXML поддерживает также API на основе стандарта SAX2 (с некоторыми добавлениями в духе Ruby) и экспериментальный анализатор на основе технологии «вытягивания». Они в этой книге не рассматриваются – можете обратиться к сайту <http://ruby-doc.org> или аналогичному ресурсу.

15.2. RSS и Atom

Часто изменяющийся контент распространяется в Интернете с помощью *синдицированных каналов*, или просто *каналов*. Обычно данные описываются на некотором диалекте языка XML.

Наверное, из всех форматов подобного рода наиболее распространен формат RSS. Эта аббревиатура означает Rich Site Summary (обогащенная сводка сайта), хотя некоторые расшифровывают ее как RDF Site Summary, понимая под RDF Resource Description Format (формат описания ресурса).

В сети Web очень много временной или часто изменяемой информации: записи в блогах, статьи в онлайн-журналах и т. д. Канал представляет собой естественный способ распространения и синдицирования такого контента.

Еще одним популярным форматом является *Atom*; некоторые даже считают, что он превосходит RSS. Но вообще-то сейчас предпочитают говорить не «RSS-канал» или «Atom-канал», а просто «канал».

Мы вкратце рассмотрим обработку форматов RSS и Atom. В первом случае применяется стандартная библиотека Ruby, во втором – библиотека, еще не вошедшая в стандартный дистрибутив.

15.2.1. Стандартная библиотека rss

Формат RSS основан на XML, поэтому разбирать его можно как обычный XML-документ. Но, поскольку это все-таки специализированный вариант, для него имеет смысл разработать специальный анализатор. Кроме того, запутанность стандарта RSS уже стала притчей во языцех – некорректно написанные программы могут генерировать такие RSS-документы, которые будет очень трудно разобрать.

Ситуация осложняется еще и тем, что существуют несовместимые версии стандарта; чаще всего используются 0.9, 1.0 и 2.0. В общем, подобно производству колбасы, RSS – такая вещь, в детали которой лучше не вникать.

В дистрибутив Ruby входит стандартная библиотека, понимающая версии стандарта 0.9, 1.0 и 2.0. Даже если вы не укажете версию входного документа явно, библиотека попытается определить ее самостоятельно.

Рассмотрим пример. Мы загрузили канал с сайта <http://marsdrive.com> и распечатали заголовки нескольких статей из него:

```
require 'rss'
require 'open-uri'

URL = "http://www.marstoday.com/rss/mars.xml"
open(URL) do |h|
  resp = h.read
  result = RSS::Parser.parse(resp, false)
```

```
puts "Канал: #{result.channel.title}"
result.items.each_with_index do |item,i|
  i += 1
  puts "#{i} #{item.title}"
end
end
```

Прежде чем двигаться дальше, я хотел бы оказать любезность поставщикам каналов. Программами, подобными приведенной выше, следует пользоваться с осторожностью, так как они потребляют ресурсы сервера поставщика. В любом реальном приложении, например в агрегаторе каналов, следует прибегать к кэшированию. Но это уже выходит за рамки простого примера.

В этой программе мы для удобства воспользовались библиотекой `open-uri`. Подробно мы рассмотрим ее в главе 18, а пока достаточно знать, что она позволяет вызывать метод `open` для URI, как для обычного файла.

Отметим, что канал извлекает из документа анализатор RSS, а наша программа печатает название канала. Кроме того, метод доступа `items` формирует список элементов канала, то есть статей, а мы распечатываем их заголовки.

Понятно, что результат меняется со временем; когда я запускал эту программу, она напечатала вот что:

```
Title: Mars Today Top Stories
1  NASA Mars Picture of the Day: Lava Levees
2  NASA Mars Global Surveyor TES Dust And Temperature Maps 25 June -
2  2 July 2006
3  Mars Institute Core Team Arrives at the HMP Research Station on
Devon Island
4  Assessment of NASA's Mars Architecture 2007-2016
5  NASA Mars Picture of the Day: Rush Hour
```

Есть также возможность генерировать документы в формате RSS (листинг 15.5). Для этого нужно инвертировать показанную выше процедуру.

Листинг 15.5. Создание RSS-канала

```
require 'rss'

feed = RSS::Rss.new("2.0")

chan = RSS::Rss::Channel.new
chan.description = "Feed Your Head"
chan.link = "http://nosuchplace.org/home/"

img = RSS::Rss::Channel::Image.new
img.url = "http://nosuchplace.org/images/headshot.jpg"
img.title = "Y.T."
img.link = chan.link

chan.image = img
feed.channel = chan
```



```
i1 = RSS::Rss::Channel::Item.new
i1.title = "Once again, here we are"
i1.link = "http://nosuchplace.org/articles/once_again/"
i1.description = "Don't you feel more like you do now than usual?"

i2 = RSS::Rss::Channel::Item.new
i2.title = "So long, and thanks for all the fiche"
i2.link = "http://nosuchplace.org/articles/so_long_and_thanks/"
i2.description = "I really miss the days of microfilm..."

i3 = RSS::Rss::Channel::Item.new
i3.title = "One hand clapping"
i3.link = "http://nosuchplace.org/articles/one_hand_clapping/"
i3.description = "Yesterday I went to an amputee convention..."

feed.channel.items << i1 << i2 << i3

puts feed
```

Большая часть этой программы понятна без слов. Мы создаем канал в формате RSS 2.0 (с пустыми элементами `channel` и `image`), а потом с помощью методов доступа добавляем данные. Элемент `image` ассоциируется с элементом `channel`, а последний – с самим RSS-каналом.

Наконец, мы создаем последовательность статей и помещаем их в канал. Отметим, что необходимо добавлять статьи именно по отдельности. Возникает искушение пойти по более простому пути:

```
feed.channel.items = [i1,i2,i3]
```

но такое решение работать не будет. Почему-то в классе `Channel` нет аксессуара `items=`. Можно было бы написать `items[0] = i1` и т.д., или то же самое в цикле. Наверное, есть и другие способы добиться нужного результата, но представленное выше решение вполне годится.

У библиотеки `rss` есть еще много возможностей, но не все в данный момент хорошо документированы. Если вы не сможете найти то, что вам нужно, в качестве последнего средства обратитесь к исходным текстам.

Многие предпочитают не RSS, а Atom. Библиотека `rss` не умеет работать с этим форматом, но есть прекрасная (хотя и не стандартная) библиотека `feedtools`. Мы рассмотрим ее в следующем разделе.

15.2.2. Библиотека `feedtools`

Библиотека `feedtools` (распространяемая в виде gem-пакета) – плод работы Боба Амана (Bob Aman). Она более или менее единообразно работает с обоими форматами RSS и Atom и сохраняет все данные в общем внутреннем формате (основанном преимущественно на Atom). В нее встроены собственные средства для работы с IRI, так что явно включать библиотеки `net/http` или `open-uri` не требуется.

Вот простой пример, эквивалентный первому примеру из предыдущего раздела:

```
require 'feed_tools'
```

```
URL = "http://www.marstoday.com/rss/mars.xml"  
feed = FeedTools::Feed.open(URL)  
puts "Description: #{feed.title}\n"
```

```
feed.entries.each_with_index {|x,i| puts "#{i+1} #{x.title}" }
```

Этот вариант короче и яснее предыдущего. Некоторые вещи не так очевидны, например у объекта `feed` нет явного метода `channel`. Однако такие методы, как `title` и `description` можно вызывать непосредственно для объекта `feed`, поскольку канал может быть только один.

Ниже показано, как читать новости из канала в формате Atom:

```
require 'feedtools'
```

```
URL = "http://www.atomenabled.org/atom.xml"  
feed = FeedTools::Feed.open(URL)  
puts "Description: #{feed.title}\n"
```

```
feed.entries.each_with_index {|x,i| puts "#{i+1} #{x.title}" }
```

Обратите внимание – изменился только сам URL! Это замечательно, поскольку мы можем обрабатывать каналы независимо от формата. Результат, естественно, похож на то, что мы видели раньше:

```
Description: AtomEnabled.org  
1 AtomEnabled's Atom Feed  
2 Introduction to Atom  
3 Moving from Atom 0.3 to 1.0  
4 Atom 1.0 is Almost Final  
5 Socialtext Supports Atom
```

Хочу еще раз предостеречь вас: не тратьте впустую ресурсы сервера, принадлежащего поставщику канала. Реальное приложение должно кэшировать содержимое канала, а если вы занимаетесь просто тестированием, лучше создайте собственный канал. Библиотека `feedtools` поддерживает довольно развитый механизм кэширования в базе данных, которого должно хватить для большинства применений.

А теперь добавим к предыдущему примеру еще две строки:

```
str = feed.build_xml("rss",2.0)  
puts str
```

Мы только что преобразовали канал Atom в канал RSS 2.0. А можно было бы вместо этого указать RSS 0.9 или RSS 1.0. Возможно и преобразование в обратном направлении: прочитать новости из RSS-канала и записать их в Atom-канал. Это одна из сильных сторон библиотеки.

Во время работы над книгой текущей версией библиотеки `feedtools` была 0.2.25. Вероятно, со временем изменится и набор возможностей, и API.

15.3. Обработка изображений с помощью RMagick

Последние пятнадцать лет на нас обрушивается все больше и больше графической информации. В качестве основного поставщика «услуги для глаз» во всех формах компьютеры уже обогнали телевизоры. А значит, программистам приходится манипулировать графическими данными, представленными в различных форматах. На языке Ruby это лучше всего делать с помощью библиотеки RMagick, которую написал Тим Хантер (Tim Hunter).

RMagick – это привязка к Ruby библиотеки ImageMagick (или ее ветви, GraphicsMagick). Устанавливается она как gem-пакет, но для работы нужно еще установить одну из базовых библиотек (IM или GM). Если вы работаете в Linux, то, вероятно, та или другая библиотека уже имеется, а, если нет, можете загрузить ее с сайта <http://imagemagick.org> (или <http://graphicsmagick.org>).

Поскольку RMagick – лишь привязка, то спрашивать, какие графические форматы она поддерживает, – все равно что спрашивать, какие форматы поддерживает базовая библиотека. Все наиболее распространенные, в частности JPG, GIF, PNG, TIFF наряду с десятками других.

То же относится и к операциям, поддерживаемым RMagick. Они ограничены лишь возможностями базовой библиотеки, поскольку RMagick дублирует весь ее API. Кстати говоря, API не только функционально богат, но и является прекрасным примером API «в духе Ruby»: в нем привычно используются символы, блоки и префиксы методов, так что большинству программистов Ruby он покажется интуитивно очевидным.

Заметим попутно, что API очень объемный. Ни этой главы, ни даже всей книги целиком не хватило бы для рассмотрения всех его деталей. В следующих разделах мы дадим лишь общее представление об RMagick, а полную информацию вы можете найти на сайте проекта (<http://rmagick.rubyforge.org>).

15.3.1. Типичные графические задачи

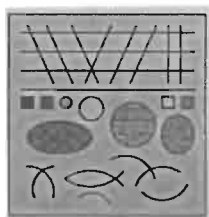
Одна из самых простых и распространенных задач, связанных с графическим файлом, – получение характеристик изображения (ширина и высота в пикселях и т.д.). Посмотрим, как можно извлечь эти метаданные.

На рис. 15.1 приведены два простых изображения, на которые мы будем ссылаться в этом и последующих примерах. Первое (smallpic.jpg) – просто абстрактная картинка, созданная в графическом редакторе; в ней присутствуют несколько оттенков серого цвета, а также прямые и кривые линии. Второе – фотография старенького автомобиля, которую я сделал в 2002 году в сельском районе Мексики. Для книги оба изображения переведены в черно-белый формат. В листинге 15.6 показано, как извлечь из соответствующих файлов необходимую информацию.

Листинг 15.6. Получение информации об изображении

```
require 'RMagick'

def show_info(fname)
  img = Magick::Image::read(fname).first
```



smallpic.jpg



vw.jpg

Рис. 15.1.
Два примера
изображений

```

fmt = img.format
w,h = img.columns, img.rows
dep = img.depth
nc = img.number_colors
nb = img.filesize
xr = img.x_resolution
yr = img.y_resolution
res = Magick::PixelsPerInchResolution ? "дюйм" : "см"
puts <<-EOF
Файл:      #{fname}
Формат:    #{fmt}
Размеры:   #{w}x#{h} пикселей
Цветов:    #{nc}
Длина файла:  #{nb} байтов
Разрешение:  #{xr}/#{yr} пикселей на #{res}
EOF
puts
end

show_info("smallpic.jpg")
show_info("vw.jpg")

```

Вот результат работы этой программы:

Файл: smallpic.jpg
Формат: JPEG
Размеры: 257x264 пикселей
Цвета: 248
Длина файла: 19116 байтов
Разрешение: 72.0/72.0 пикселей на дюйм

Файл: vw.jpg
Формат: JPEG
Размеры: 640x480 пикселей
Цвета: 256
Длина файла: 55892 байтов
Разрешение: 72.0/72.0 пикселей на дюйм
2.0 pixels per inch

Посмотрим, как именно работает эта программа. Для чтения файла мы вызываем метод `Magick::Image::read`. Поскольку один файл (например, анимированный GIF) может содержать несколько изображений, эта операция возвращает массив изображений (мы получаем лишь первое, вызывая метод `first`). Для чтения файла можно также воспользоваться методом `Magick::ImageList.new`.

У объекта, представляющего изображение, есть ряд методов чтения: `format` (название формата изображения), `filesize`, `depth` и другие. Не так очевидно, что для получения ширины и высоты изображения служат методы `columns` и `rows` соответственно (поскольку изображение представляется в виде прямоугольной таблицы пикселей). Разрешение представляется двумя числами, так как может быть разным по вертикали и горизонтали.

Можно получить и другие метаданные об изображении. Подробнее об этом вы можете прочитать в онлайн-документации по RMagick.

Часто возникает необходимость перевести изображение из одного формата в другой. В RMagick это проще всего сделать, прочитав изображение из файла в одном из поддерживаемых форматов и записав его в другой файл. Новый формат определяется расширением имени файла. Понятно, что «за кулисами» при этом происходит преобразование данных. Пример:

```
img = Magick::Image.read("smallpic.jpg")  
img.write("smallpic.gif") # Преобразовать в формат GIF.
```

Иногда нужно изменить размер изображения (сделать его больше или меньше). Для этого обычно применяется один из четырех методов: `thumbnail`, `resize`, `sample` и `scale`. Все они принимают либо число с плавающей точкой (коэффициент масштабирования), либо два числа (новые размеры в пикселях). Различия между этими методами продемонстрированы в листинге 15.7. Если вас волнует быстродействие, рекомендую провести тесты на своем компьютере, используя собственные данные.

Листинг 15.7. Четыре способа масштабирования изображения

```
require 'RMagick'  
  
img = Magick::ImageList.new("vw.jpg")
```

```
# Все эти методы могут принимать либо один параметр - коэффициент
# масштабирования, либо два - ширину и высоту.

# Метод thumbnail самый быстрый, особенно если нужно получить очень
# маленькое изображение.

pic1 = img.thumbnail(0.2)      # Уменьшить до 20%.
pic2 = img.thumbnail(64,48)    # Новый размер - 64x48 пикселей.

# resize работает со средней скоростью. Если заданы третий и четвертый
# параметры, то они интерпретируются как фильтр и размывание
# соответственно. По умолчанию подразумевается фильтр LanczosFilter
# и коэффициент размывания 1.0.

pic3 = img.resize(0.40)        # Уменьшить до 40%.
pic4 = img.resize(320,240)     # Новый размер - 320x240.
pic5 = img.resize(300,200,Magick::LanczosFilter,0.92)

# Метод sample также имеет среднее быстродействие (и не выполняет
# интерполяцию цветов).

pic6 = img.sample(0.35)        # Уменьшить до 35%.
pic7 = img.sample(320,240)     # Новый размер - 320x240.

# Метод scale в моих тестах оказался самым медленным.

pic8 = img.scale(0.60)         # Уменьшить до 60%.
pic9 = img.scale(400,300)      # Новый размер - 400x300.
```

Изображения можно подвергать и многим другим трансформациям. Некоторые просты и понятны, другие гораздо сложнее. В следующем разделе мы рассмотрим несколько интересных трансформаций и специальных эффектов.

15.3.2. Специальные эффекты и трансформации

Над изображением можно выполнять следующие операции: зеркально отражать, инвертировать цвета, поворачивать, растягивать, перекрашивать и т.д. В RMagick есть десятки методов для выполнения подобных операций, причем многие из них еще и настраиваются путем указания параметров.

В листинге 15.8 показано 12 различных эффектов. Метод `example` принимает имя файла, символ, соответствующий методу, и имя нового файла; он читает файл, вызывает метод и записывает результат в новый файл. Сами методы (скажем, `do_rotate`) по большей части просты – они получают изображение и вызывают его метод экземпляра (а возвращают результат трансформации).

Листинг 15.8. Двенадцать специальных эффектов и трансформаций

```
require 'RMagick'

def do_flip(img)
```

```
img.flip
end

def do_rotate(img)
  img.rotate(45)
end

def do_implode(img)
  img = img.implode(0.65)
end

def do_resize(img)
  img.resize(120,240)
end

def do_text(img)
  text = Magick::Draw.new
  text.annotate(img, 0, 0, 0, 100, "HELLO") do
    self.gravity = Magick::SouthGravity
    self.pointsize = 72
    self.stroke = 'black'
    self.fill = '#FAFAFA'
    self.font_weight = Magick::BoldWeight
    self.font_stretch = Magick::UltraCondensedStretch
  end
  img
end

def do_emboss(img)
  img.emboss
end

def do_spread(img)
  img.spread(10)
end

def do_motion(img)
  img.motion_blur(0,30,170)
end

def do_oil(img)
  img.oil_paint(10)
end

def do_charcoal(img)
  img.charcoal
end
```

```
def do_vignette(img)
  img.vignette
end

def do_affine(img)
  spin_xform = Magick::AffineMatrix.new(1, Math::PI/6, Math::PI/6, 1, 0,
0)
  img.affine_transform(spin_xform)      # Применить преобразование.
end

###

def example(old_file, meth, new_file)
  img = Magick::ImageList.new(old_file)
  new_img = send(meth, img)
  new_img.write(new_file)
end

example("smallpic.jpg", :do_flip,      "flipped.jpg")
example("smallpic.jpg", :do_rotate,    "rotated.jpg")
example("smallpic.jpg", :do_resize,    "resized.jpg")
example("smallpic.jpg", :do_implode,   "imploded.jpg")
example("smallpic.jpg", :do_text,      "withtext.jpg")
example("smallpic.jpg", :do_emboss,    "embossed.jpg")

example("vw.jpg", :do_spread,          "vw_spread.jpg")
example("vw.jpg", :do_motion,          "vw_motion.jpg")
example("vw.jpg", :do_oil,             "vw_oil.jpg")
example("vw.jpg", :do_charcoal,        "vw_char.jpg")
example("vw.jpg", :do_vignette,        "vw_vig.jpg")
example("vw.jpg", :do_affine,          "vw_spin.jpg")
```

Мы продемонстрировали методы flip, rotate, implode, resize, annotate и др. Результаты представлены на рис. 15.2.

О том, какие еще существуют трансформации изображений, читайте в онлайн-новой документации.

15.3.3. API рисования

В RMagick имеется развитый API для рисования линий, многоугольников и различных кривых. Он поддерживает заливку, полупрозрачность, выбор цвета, шрифтов, вращение, растяжение и другие операции.

Чтобы получить представление об имеющихся возможностях, рассмотрим простой пример.

В листинге 15.9 приведена программа, которая рисует на заданном фоне сетку, а поверх нее несколько закрашенных геометрических фигур. Черно-белое изображение, получившееся в результате, показано на рис. 15.3.

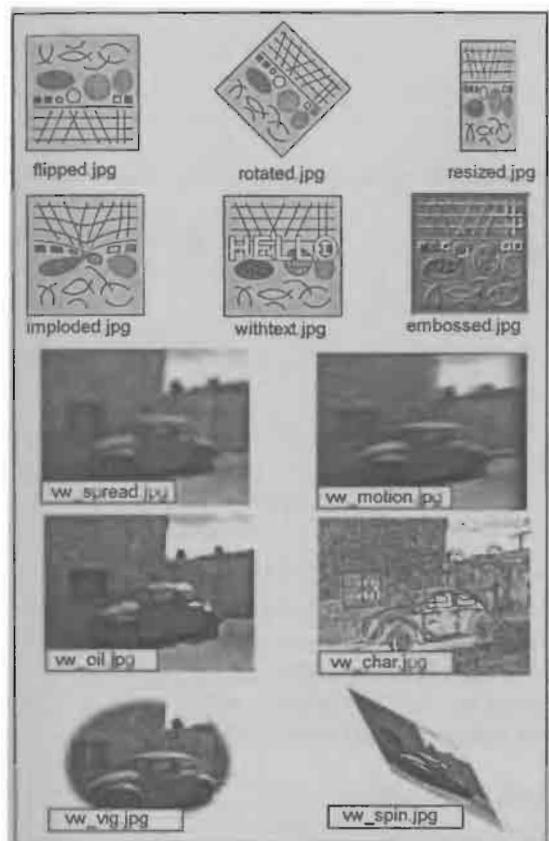


Рис. 15.2. Двенадцать специальных эффектов и трансформаций

Листинг 15.9. Простая программа рисования

```
require 'RMagick'

img = Magick::ImageList.new
img.new_image(500, 500)

purplish = "#ff55ff"
yuck = "#5fff62"
bleah = "#3333ff"

line = Magick::Draw.new
50.step(450,50) do |n|
  line.line(n,50, n,450) # Вертикальная прямая.
  line.draw(img)
  line.line(50,n, 450,n) # Горизонтальная прямая.
  line.draw(img)
end
```

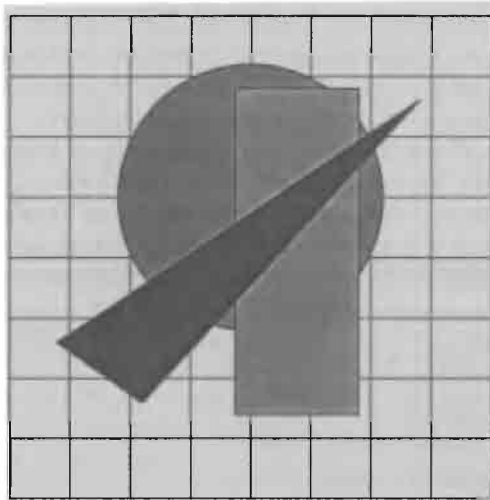


Рис. 15.3. Простая программа рисования

```
# Нарисовать круг.
cir = Magick::Draw.new
cir.fill('purplish')
cir.stroke('black').stroke_width(1)
cir.circle(250,200, 250,310)
cir.draw(img)

rect = Magick::Draw.new
rect.stroke('black').stroke_width(1)
rect.fill('yuck')
rect.rectangle(340,380,237,110)
rect.draw(img)

tri = Magick::Draw.new
tri.stroke('black').stroke_width(1)
tri.fill('bleah')
tri.polygon(90,320,160,370,390,120)
tri.draw(img)

img = img.quantize(256,Magick::GRAYColorspace)

img.write("drawing.gif")
```

Рассмотрим эту программу подробнее. Сначала мы создаем «пустое» изображение методом `ImageList.new`, а потом вызываем для возвращенного объекта метод `new_image`. Можно считать, что мы получили «чистый холст» заданного размера (500 × 500 пикселей).

Для удобства определим несколько цветов с понятными именами, например `purplish` и `yuck`. Цвета определяются так же, как в HTML. Базовая библиотека `xMagick` сама распознает много названий цветов, например, `red` и `black`; если сомневаетесь, пробуйте или задавайте цвета в шестнадцатеричном виде.

Затем мы создаем объект рисования `line`; это объект Ruby, соответствующий графическому объекту, который мы видим на экране. Переменную иногда называют `gc` или как-то похоже (от «graphics context» – графический контекст), но нам кажется естественным употребить имя, отражающее природу объекта.

Далее вызывается метод `line` объекта рисования, по два раза на каждой итерации цикла. Взглянув на то, как изменяются координаты, вы поймете, что на каждой итерации рисуется одна горизонтальная и одна вертикальная прямая.

После каждого обращения к `line` мы вызываем метод `draw` того же объекта и передаем ему ссылку на изображение. Именно на этом шаге графический объект помещается на холст.

Лично меня обращения вида `shape.draw(image)` немного путают. В общем случае вызов любого метода выглядит так:

```
big_thing.operation(little_thing)
# Например: dog.wag(tail) (собака.вилять(хвост))
```

Но методы RMagick записываются, скорее, в виде:

```
little_thing.operation(big_thing)
# Продолжая аналогию: tail.wag(dog) (хвост.вилять(собака))
```

Впрочем, эта идиома достаточно распространена, особенно в программах рисования и графических интерфейсах. И в классической объектно-ориентированной идеологии это вполне оправданно: фигура знает, как нарисовать себя, а стало быть, должна иметь метод `draw`. Он же, в свою очередь, должен знать, где рисовать, поэтому ему нужно передать ссылку на холст (или что-то подобное).

Но, возможно, вас не мучает вопрос, от имени какого объекта следует вызывать метод. Тем лучше!..

Покончив с сеткой, мы переходим к рисованию фигур. Метод `circle` принимает в качестве параметров центр окружности и какую-нибудь точку на ней (радиус не передается!). Метод `rectangle` еще проще; для рисования прямоугольника нужно задать координаты левого верхнего угла (первые два параметра) и координаты правого нижнего угла (последние два параметра). Треугольник же является частным случаем многоугольника; мы задаем координаты всех его вершин, а замыкающий отрезок (из конечной точки в начальную) рисуется автоматически.

У каждого графического объекта есть еще несколько методов. Взгляните на этот «сцепленный» вызов:

```
shape.stroke('black').stroke_width(1)
```

Это что-то вроде пера, которое рисует черными чернилами линию толщиной в один пиксель. Цвет штриха часто имеет значение, особенно если мы хотим закрасить фигуру.

Конечно, у каждой из трех этих фигур есть еще метод `fill`, при вызове которого указывается цвет заливки. (Имеются также более сложные способы заливки, например, штриховкой, с наложением тени и т.д.) Метод `fill` заменяет цвет внутренних пикселей фигуры указанным, ориентируясь на цвет границы, чтобы отличить внутреннюю часть от внешней.

API рисования содержит также методы для настройки полупрозрачности, пространственных преобразований и многого другого. Есть методы для анализа, рисования

и манипулирования текстовыми строками. Существует даже специальный RVG API (Ruby Vector Graphics – векторная графика в Ruby), совместимый с рекомендацией консорциума W3C по масштабируемой векторной графике (SVG).

Мы не можем привести здесь документацию по всем этим бесчисленным возможностям. Дополнительную информацию вы можете найти на сайте <http://rmagick.rubyforge.org>.

15.4. Создание документов в формате PDF с помощью библиотеки PDF::Writer

Библиотека PDF::Writer предназначена для создания PDF-документов из программы на языке Ruby. Ее можно установить из gem-пакета или скачать с сайта RubyForge. Последовательность создания документа проста:

```
require 'rubygems'
require 'pdf/writer'

pdf = PDF::Writer.new
```

15.4.1. Основные концепции и приемы

Одна из серьезных проблем, встающих перед любым дизайнером документов, – текстовые шрифты. Библиотека PDF::Writer поддерживает пять основных шрифтов, причем первые три допускают полужирное и курсивное начертание:

- Times-Roman
- Helvetica
- Courier
- ZapfDingbats
- Symbol

Если шрифт не указан, по умолчанию предполагается Helvetica. При выборе шрифта можно создать таблицу замены символов, которая позволяет имитировать символы, не имеющие графического начертания или отсутствующие в кодовой странице. В шрифтах Times-Roman, Helvetica и Courier по 315 печатаемых символов (из них у 149 есть предопределенные байтовые коды); в шрифте Symbol – 190 символов (у 189 есть предопределенные коды), а в шрифте ZapfDingbats – 202 символа (всем соответствуют коды). Шрифты представлены в кодировке Adobe, но в момент выбора шрифта отдельные символы можно переопределить.

Текущая версия не позволяет напечатать все 315 символов, определенных в шрифтовом файле, поскольку после того как шрифт выбран, изменить таблицу замены символов уже невозможно. В последующих версиях PDF::Writer эта проблема будет решена.

В следующем примере мы задали для PDF-документа шрифт Times-Roman. Программа чтения PDF-файлов будет считать, что текст представлен в кодировке WinAnsiEncoding, но вместо символа с кодом 0x01 подставит глиф «lozenge» (ромб). Мы еще увидим его ниже (листинг 15.11).

```
pdf.select_font "Times-Roman",
  { :encoding => "WinAnsiEncoding",
    :differences => {0x01 => "lozenge"}
  }
```

Библиотека `PDF::Writer` располагает средствами для форматирования текста и создания таблиц, которые хорошо документированы. Не так очевидно, что пока не срабатывает автоматическая разбивка на страницы, можно форматировать страницу вручную весьма любопытными способами. С помощью переноса осей и масштабирования мы можем нарисовать четыре страницы на одной.

В текущей версии `PDF::Writer` (1.1.3) каждая такая «страница» должна полностью уместиться на одной физической странице. Если в дело вмешивается механизм автоматического разбиения на страницы, то будет создана новая физическая страница. В следующих версиях усовершенствованный вариант этой техники будет работать и для многоколонных страниц.

Для демонстрации создадим метод `quadrant` (листинг 15.10). Он войдет также составной частью в длинный пример из следующего раздела, который преследует две цели: показать, как создается документ из четырех страниц и как можно разместить четыре страницы PDF-документа на одной странице книги, сэкономив тем самым место.

Листинг 15.10. Метод `quadrant`

```
def quadrant(pdf, quad)
  raise unless block_given?

  mx = pdf.absolute_x_middle
  my = pdf.absolute_y_middle

  pdf.save_state

  case quad
  when :ul
    pdf.translate_axis(0, my)
  when :ur
    pdf.translate_axis(mx, my)
  when :ll
    nil # pdf.translate_axis(0, 0)
  when :lr
    pdf.translate_axis(mx, 0)
  end

  pdf.scale_axis(0.5, 0.5)
  pdf.y = pdf.page_height
  yield
  pdf.restore_state
end
```

Здесь каждая страница целиком строится в отдельном блоке. Таким образом, мы можем изменять масштаб и положение осей, никак не затрагивая код построения

страницы. Первым делом мы, конечно, сохраняем текущее состояние. Это позволит нам не восстанавливать вручную масштаб и начало системы координат по завершении работы. Перед тем как приступить к конструированию, мы помещаем начало координат квадранта в нужное место страницы (`pdf.translate_axis x, y`).

Предположим, что начало координат находится не в точке (0, 0), а в точке (50, 50). Тогда отрезок из точки (15, 20) в точку (35, 40) на самом деле будет соединять точки с координатами (65, 70) и (85, 90). Но код рисования отрезка об этом ничего не знает.

После переноса оси (то есть сдвига начала координат) мы можем изменить масштаб вдоль оси. Чтобы получить четыре квадранта, следует уменьшить вдвое масштаб по осям X и Y (`pdf.scale_axis 0.5, 0.5`). Иными словами, если бы сейчас я провел отрезок между точками (0, 0) и (90, 90), то без переноса осей он соединял бы точки с физическими координатами (0, 0) и (45, 45), а с переносом – точки с координатами (90, 90) и (135, 135). В любом случае будет проведена линия вдоль диагонали длиной 90 единиц измерения. Просто из-за масштабирования сами единицы стали в два раза меньше.

Затем мы отдаем управление блоку, а когда он закончит работу, восстанавливаем состояние, вызывая предоставленный библиотекой метод `restore_state`. Иначе пришлось бы вручную увеличивать масштаб вдвое и переносить ось в обратном направлении.

15.4.2. Пример документа

Для демонстрации рассмотренной выше техники мы создадим четыре страницы в четырех разных квадрантах. Три из них – слегка измененные варианты демонстрационных программ, включенных в дистрибутив `PDF::Writer`:

- `demo.rb`, квадрант 1
- `individual-i.rb`, квадрант 3
- `gettysburg.rb`, квадрант 4

Четвертая страница (в квадранте 2) не имеет прямого аналога среди демонстрационных программ, она ближе всего к программе `chunkybacon.rb`.

Весь код приведен в листинге 15.11, а результат показан на рис. 15.4. Пример довольно длинный, мы подробно обсудим его ниже.

Листинг 15.11. Создание демонстрационного документа

```
require 'rubygems'
require 'pdf/writer'

def quadrant(pdf, quad)
  raise unless block_given?

  mx = pdf.absolute_x_middle
  my = pdf.absolute_y_middle
  pdf.save_state
```

```

case quad
when :ul
  pdf.translate_axis 0, my
when :ur
  pdf.translate_axis mx, my
when :ll
  nil # no translation needed
when :lr
  pdf.translate_axis mx, 0
end

pdf.scale_axis(0.5, 0.5)
pdf.y = pdf.page_height
yield
pdf.restore_state
end

pdf = PDF::Writer.new
pdf.select_font("Times-Roman",
               :encoding => "WinAnsiEncoding",
               :differences => { 0x01 => "lozenge" })

mx = pdf.absolute_x_middle
my = pdf.absolute_y_middle

pdf.line(0, my, pdf.page_width, my).stroke
pdf.line(mx, 0, mx, pdf.page_height).stroke

# Левый верхний: Demo (UL).

quadrant(pdf, :ul) do
  x = pdf.absolute_right_margin
  rl = 25

  40.step(1, -3) do |xw|
    tone = 1.0 - (xw / 40.0) * 0.2
    pdf.stroke_style(PDF::Writer::StrokeStyle.new(xw))
    pdf.stroke_color(Color::RGB.from_fraction(1, tone, tone))
    pdf.line(x, pdf.bottom_margin, x,
             pdf.absolute_top_margin).stroke
    x -= xw+2
  end

  40.step(1, -3) do |xw|
    tone = 1.0 - (xw / 40.0) * 0.2
    pdf.stroke_style(PDF::Writer::StrokeStyle.new(xw))
    pdf.stroke_color(Color::RGB.from_fraction(1, tone, tone))
    pdf.circle_at(pdf.left_margin + 10, pdf.margin_height - 15,
                  rl).stroke
    rl += xw
  end
end

```

```

end
pdf.stroke_color(Color::RGB::Black)

x = pdf.absolute_left_margin
y = pdf.absolute_bottom_margin
w = pdf.margin_width
h = pdf.margin_height
pdf.rectangle(x, y, w, h).stroke

text = "The Ruby Way"

y = pdf.absolute_top_margin
50.step(5, -5) do |size|
  height = pdf.font_height(size)
  y -= height
  pdf.add_text(pdf.left_margin + 10, y, text, size)
end

(0...360).step(20) do |angle|
  pdf.fill_color(Color::RGB.from_fraction(rand, rand, rand))

  pdf.add_text(300 + Math.cos(PDF::Math.deg2rad(angle)) * 40,
               300 + Math.sin(PDF::Math.deg2rad(angle)) * 40,
               text, 20, angle)
end
end

pdf.fill_color Color::RGB::Black

# Правый верхний: Grampian Highlands (UR).
quadrant(pdf, :ur) do
  pdf.image("grampian-highlands.jpg",
            :height => pdf.margin_height,
            :resize => :width)
  pdf.text("The Grampian Highlands, Scotland",
            :justification => :center,
            :font_size => 36)
  pdf.text("\001August 2001\001", :justification => :center,
            :font_size => 24)

  pdf.move_pointer(24)
  info = <<-'EOS'.split($/).join(" ").squeeze(" ")
  This picture was taken during a driving vacation through the
  Scottish highlands in August 2001 by Austin Ziegler.
  EOS
  pdf.text(info, :justification => :full, :font_size => 16,
            :left => 100, :right => 100)
end

```



```
pdf.fill_color Color::RGB::Black
# Левый нижний: Individual-I (LL).

quadrant(pdf, :ll) do
  require 'color/palette/monocontrast'

  class IndividualI
    def initialize(size = 100)
      @size = size
    end

    # Размер буквы "i" в пунктах.
    attr_accessor :size

    def half_i(pdf)
      pdf.move_to(0, 82)
      pdf.line_to(0, 78)
      pdf.line_to(9, 78)
      pdf.line_to(9, 28)
      pdf.line_to(0, 28)
      pdf.line_to(0, 23)
      pdf.line_to(18, 23)
      pdf.line_to(18, 82)
      pdf.fill
    end
    private :half_i

    def draw(pdf, x, y)
      pdf.save_state
      pdf.translate_axis(x, y)
      pdf.scale_axis(1 * (@size / 100.0), -1 * (@size / 100.0))

      pdf.circle_at(20, 10, 7.5)
      pdf.fill

      half_i(pdf)

      pdf.translate_axis(40, 0)
      pdf.scale_axis(-1, 1)

      half_i(pdf)
      pdf.restore_state
    end
  end

  ii = IndividualI.new(24)

  x = pdf.absolute_left_margin
  y = pdf.absolute_top_margin
  bg = Color::RGB.from_fraction(rand, rand, rand)
```

```
fg = Color::RGB.from_fraction(rand, rand, rand)
pal = Color::Palette::MonoContrast.new(bg, fg)
```

```
sz = 24
```

```
(-5..5).each do |col|
  pdf.fill_color pal.background[col]
  ii.draw(pdf, x, y)
  ii.size += sz
  x += sz / 2.0
  y -= sz / 2.0
  pdf.fill_color pal.foreground[col]
  ii.draw(pdf, x, y)
  x += sz / 2.0
  y += sz / 2.0
  ii.size += sz
end
end
```

```
pdf.fill_color Color::RGB::Black
```

```
# Правый нижний: Gettysburg Address (LR).
# Это текст Геттисбергского обращения Авраама Линкольна.
```

```
quadrant(pdf, :lr) do
  pdf.text("The Gettysburg Address\n\n",
           :font_size => 36, :justification => :center)
  y0 = pdf.y + 18
```

```
speech = <<-'EOS'.split($/).join(" ").squeeze(" ")
Four score and seven years ago our fathers brought forth on
this continent a new nation, conceived in liberty and
dedicated to the proposition that all men are created equal.
Now we are engaged in a great civil war, testing whether
that nation or any nation so conceived and so dedicated can
long endure. We are met on a great battlefield of that war.
We have come to dedicate a portion of that field as a final
resting-place for those who here gave their lives that that
nation might live. It is altogether fitting and proper that
we should do this. But in a larger sense, we cannot
dedicate, we cannot consecrate, we cannot hallow this
ground. The brave men, living and dead who struggled here
have consecrated it far above our poor power to add or
detract. The world will little note nor long remember what
we say here, but it can never forget what they did here. It
is for us the living rather to be dedicated here to the
unfinished work which they who fought here have thus far so
nobly advanced. It is rather for us to be here dedicated to
the great task remaining before us that from these honored
dead we take increased devotion to that cause for which they
```

gave the last full measure of devotion that we here highly
resolve that these dead shall not have died in vain, that
this nation under God shall have a new birth of freedom, and
that government of the people, by the people, for the people
shall not perish from the earth.

EOS

```
pdf.text(speech, :justification => :full, :font_size => 14,  
         :left => 50, :right => 50)  
pdf.move_pointer(36)  
pdf.text("U.S. President Abraham Lincoln, 19 November 1863",  
         :justification => :right, :right => 100)  
pdf.text("Gettysburg, Pennsylvania", :justification => :right,  
         :right => 100)  
pdf.rounded_rectangle(pdf.left_margin + 25, y0, pdf.margin_width - 50,  
                       y0 - pdf.y + 18, 10).stroke  
end  
  
pdf.save_as("4page.pdf")
```

Итак, в четырех квадрантах расположены следующие страницы:

- левый верхний: demo.rb;
- правый верхний: фотография Грампианских холмов, Шотландия;
- левый нижний: individual-i.rb;
- правый нижний: Геттисбергское обращение.

Для краткости будем называть эти квадранты UL, UR, LL и LR. В тексте программы используются соответствующие символы (:ul и т.д.).

Первый квадрант (UL) заполнен вертикальными линиями, толщина которых постепенно уменьшается, начиная с 40 единиц, с одновременным осветлением. Затем рисуются круги увеличивающегося радиуса, при этом толщина линий уменьшается, а цвет становится светлее. И наконец, выводятся два набора текстов: один – сверху вниз с постепенным уменьшением размера шрифта, а другой – с поворотом вокруг центральной оси как раз там, где кончаются вертикальные линии.

Страница во втором квадранте (UR) содержит картинку и ее описание. Особый интерес представляет строка с датой. Мы вставляем в поток байт с кодом 0x01; при отображении вместо него будет поставлен символ ромба в соответствии с таблицей замены, заданной при выборе шрифта.

В третьем квадранте (UR) с помощью программы Individual-I мы снова демонстрируем технику переноса осей и масштабирования. Самое интересное здесь – инверсия осей. Если по оси выбирается отрицательный масштаб, то команды вывода текста и рисования меняют направление. Следовательно, при рисовании буквы I достаточно задать лишь правила формирования половины рисунка, а потом инвертировать ось X, вызвав метод pdf.scale_axis(-1, 1), и повторить ту же последовательность операций.

The Ruby Way

The Ruby Way

The Ruby Way

The Ruby Way

The Ruby Way

The Ruby Way

The Ruby Way

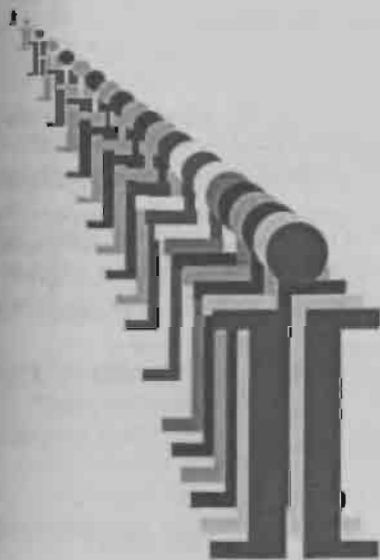
The Ruby Way

The Ruby Way



The Grampian Highlands, Scotland
6 August 20010

This picture was taken during a driving vacation
through the Scottish highlands in August 2001 by
Austin Ziegler



The Gettysburg Address

Four scores and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this. But in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it far above our poor power to add or detract. The world will little note nor long remember what we say here, but it can never forget what they did here. It is for us the living rather to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us—that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion—that we here highly resolve that these dead shall not have died in vain, that this nation under God shall have a new birth of freedom, and that government of the people, by the people, for the people shall not perish from the earth.

U. S. President Abraham Lincoln, 19 November 1863
Gettysburg, Pennsylvania

Рис. 15.4. Пример документа, состоящего из четырех страниц в разных квадрантах

Последний квадрант (LR) заполняется сравнительно легко. Мы форматируем и заключаем в прямоугольник со скругленными углами текст речи, которую президент Линкольн произнес в Геттисберге.

Сохранение PDF-документа – воплощенная простота. Если нужно записать его на диск, мы вызываем метод `save_as` объекта PDF:

```
pdf.save_as("4page.pdf")
```

Нетрудно также отправить PDF-документ браузеру из CGI-программы:

```
require 'cgi'
```

```
cgi = CGI.new
```

```
out = pdf.render
```

```
puts <<-EOS
```

```
Content-Type: application/pdf
```

```
Content-Disposition: inline; filename="4page.pdf"
```

```
Size: #{out.size}
```

```
EOS
```

Конечно, в этом разделе мы сумели затронуть лишь малую толику библиотеки `PDF::Writer`. Дополнительную информацию ищите в онлайн-официальной документации. Если вы знакомы с форматом PDF, имейте в виду, что библиотека еще развивается и пока не поддерживает спецификацию в полном объеме.

15.5. Заключение

В этой главе мы показали, как с помощью библиотеки `REXML` можно разбирать XML-документы, представленные в виде дерева DOM или потока. Познакомились мы и с интерфейсом `REXML` к языку `XPath`.

Был продемонстрирован разбор информации из новостных каналов, представленных в формате на базе XML. Библиотека `rss` умеет работать только с форматом RSS, а библиотека `feedtools` понимает форматы RSS и Atom (и умеет преобразовывать из одного в другой).

Мы также видели, как можно читать и манипулировать графическими изображениями разного формата с помощью библиотеки `RMagick`. Рассмотрели мы и API рисования, позволяющий включать в изображение произвольный текст и геометрические фигуры. Наконец, мы показали, как с помощью библиотеки `PDF::Writer` можно создавать из программы сложные PDF-документы высокого качества.

Следующая глава посвящена совсем другой теме. Речь пойдет об эффективном тестировании и отладке написанных на Ruby программ.

Глава 16. Тестирование и отладка

.....
*Неполадки в блоке АЕ-35. В ближайшие семьдесят два часа
блок может отказать.*

Артур Кларк, «Космическая Одиссея 2001 года»

Тестирование – вещь важная. Все компетентные программисты об этом знают, хотя не всегда этот вопрос стоит для них на первом месте.

Конечно, исчерпывающее тестирование, как правило, невозможно. Программа сколько-нибудь заметного размера на протяжении своего жизненного цикла обязательно преподнесет сюрпризы. Максимум, что мы можем сделать, – тестировать тщательно и избирательно, стараясь проверить как можно больше.

Исторически сложилось так, что программисты не всегда тестируют как положено. Объясняют это обычно тем, что тесты трудно готовить и прогонять, что вся процедура требует ручного вмешательства или отнимает слишком много времени.

В 1990 году в сообществе программистов стала распространяться «культура тестирования». Идеи экстремального программирования и управляемой тестами разработки начали овладевать умами разработчиков по всему миру.

Являетесь ли вы твердокаменным приверженцем идеологии «тестируй с самого начала», не так существенно. Важно, что любой человек может воспользоваться инструментами, которые позволяют автоматизировать тестирование, упростив написание и прогон тестов.

Такие инструменты, как `Test::Unit` и `ZenTest`, написать на Ruby было проще в силу динамичности и гибкости языка. Не менее легко и (посмею ли сказать?) приятно ими пользоваться. Внес изменение в программу, а потом смотришь, как все тесты успешно доходят до конца, – положительно в этом что-то есть!

Помимо этих инструментов в Ruby есть еще немало программ и библиотек для отладки, профилирования и испытания различных путей исполнения. Эта глава посвящена обзору имеющихся средств.

16.1. Библиотека `Test::Unit`

«Стандартный» способ автономного тестирования компонентов в Ruby – библиотека `Test::Unit` Натаниэля Тэлбота (Nathaniel Talbott). Она была включена в дистрибутив Ruby еще в 2001 году.

В этой библиотеке для анализа тестового кода применяется отражение. Когда вы создаете подкласс класса `Test::Unit::TestCase`, все методы, имена которых начинаются с `test`, считаются тестовыми.

```
require 'test/unit'

class TC_MyTest < Test::Unit::TestCase
  def test_001
    # ...
  end
  def test_002
    # ...
  end
  # ...
end
```

Методы необязательно нумеровать, как показано в этом примере. Это мое личное соглашение, но, конечно, есть и другие.

Нежелательно и, пожалуй, даже неправильно составлять тесты так, чтобы их поведение зависело от порядка запуска. Однако `Test::Unit` прогоняет их в алфавитном (лексикографическом) порядке, поэтому, нумеруя свои методы, я вижу, как они выполняются в определенной последовательности.

Я также предпочитаю включать некий «заголовок» в имя метода (описывающий его область действия или назначение):

```
def test_053_default_to_current_directory
  # ...
end
def test_054_use_specified_directory
  # ...
end
```

Кроме прочего, неплохо оставлять хотя бы однострочный комментарий, касающийся цели и смысла теста. Вообще говоря, у каждого теста должна быть только одна цель.

А если нужно организовать некую среду выполнения, для чего требуется время? Неразумно делать это для каждого теста, и мы не вправе заводить для данной цели отдельный метод (поскольку поведение не должно зависеть от порядка прогона).

Если всем тестам нужна особая среда, можно воспользоваться методами класса `setup` и `teardown`. Возможно, вам это покажется странным, но вызываются они для каждого теста. Если вы хотите выполнить настройку один раз, перед прогоном одного конкретного или всех тестов, то можете поместить соответствующий код в тело класса раньше всех тестовых методов (или даже до самого класса).

А если после выполнения всех тестов нужно разрушить созданную среду? По техническим причинам (так уж работает библиотека `Test::Unit`) сделать это трудно. «Самый лучший» способ – переопределить метод `run` всего комплекта тестов (но не метод класса `run`), обернув его функциональность. Рассмотрим пример в листинге 16.1.

Листинг 16.1. Подготовка и разрушение среды исполнения

```
require 'test/unit'

class MyTest < Test::Unit::TestCase
```

```
def self.major_setup
  # ...
end

def self.major_teardown
  # ...
end

def self.suite
  mysuite = super          # Вызвать метод suite родителя.

  def mysuite.run(*args)   # Добавить синглетный метод
    MyTest.major_setup
    super
    MyTest.major_teardown
  end

  mysuite                  # и вернуть новое значение.
end

def setup
  # ...
end

def teardown
  # ...
end

def test_001
  # ...
end

def test_002
  # ...
end

# ...
end
```

Вряд ли вы будете поступать так часто. О методе `suite` мы поговорим чуть позже, а пока продолжим рассмотрение отдельных тестов.

Что должно входить в тест? Нужно как-то решить, прошел он или нет. Для этой цели применяются *утверждения*.

Простейшее утверждение – это метод `assert`. Он принимает проверяемый параметр и еще один необязательный параметр (сообщение). Если значение параметра истинно (то есть все, кроме `false` и `nil`), тест прошел. В противном случае тест не прошел – тогда печатается сообщение, если оно было задано.

Есть и другие методы для формулирования утверждений. Обратите внимание, что «ожидаемое» значение всегда предшествует «фактическому».


```

assert_equal(expected, actual)      # assert(expected==actual)
assert_not_equal(expected, actual)  # assert(expected!=actual)
assert_match(regex, string)         # assert(regex =~ string)
assert_no_match(regex, string)      # assert(regex !~ string)
assert_nil(object)                   # assert(object.nil?)
assert_not_nil(object)              # assert(!object.nil?)

```

Некоторые утверждения носят более объектно-ориентированный характер:

```

assert_instance_of(klass, obj)      # assert(obj.instance_of? klass)
assert_kind_of(klass, obj)          # assert(obj.kind_of? klass)
assert_respond_to(obj, meth)        # assert(obj.respond_to? meth)

```

Другие относятся к исключениям и символам, которые генерируются методом `throw`. Понятно, что такие методы принимают блок.

```

assert_nothing_thrown { ... }      # Не было throw.
assert_nothing_raised { ... }      # Не было raise.
assert_throws(symbol) { ... }      # Символ в результате throw.
assert_raises(exception) { ... }    # Исключение в результате raise.

```

Есть еще несколько утверждений, но эти применяются чаще всего и отвечают почти всем потребностям. Дополнительную информацию можно найти в онлайн-новой документации на сайте <http://ruby-doc.org>.

Имеется еще метод `flunk`, который всегда завершается неудачно. Можно считать, что это некий вид заглушки.

Если при запуске тестового файла вы ничего специально не указываете, то по умолчанию вызывается консольный исполнитель тестов. Это возвращает нас к старой доброй технологии 1970-х годов. Имеются и другие исполнители, например графический `Test::Unit::UI::GTK::TestRunner`. Любой исполнитель тестов можно вызвать, обратившись к его методу `run`, которому передается специальный параметр, описывающий набор тестов:

```

class MyTests < Test::Unit::TestCase
  # ...
end

# Явное указание исполнителя тестов...
runner = Test::Unit::UI::Console::TestRunner
runner.run(MyTests)

```

Параметром может быть любой объект, обладающий методом `suite`, который возвращает объект, представляющий комплект тестов. Что все это означает?

Познакомимся к понятием *комплекта тестов* ближе. Оказывается, комплект тестов может состоять из набора тестов или набора подкомплектов. Следовательно, можно сгруппировать тесты так, что будет прогоняться либо только один набор, либо сразу все.

Пусть, например, есть три набора тестов, и вы хотите прогнать их как единый комплект. Можно было бы поступить так:

```

require 'test/unit/testsuite'

require 'tc_set1'

```

```
require 'tc_set2'
require 'ts_set3'

class TS_MyTests
  def self.suite
    mysuite = Test::Unit::TestSuite.new
    mysuite << TC_Set1.suite
    mysuite << TC_Set2.suite
    mysuite << TS_Set3.suite
    return mysuite
  end
end

Test::Unit::UI::Console::TestRunner.run(TS_MyTests)
```

Но такая сложность ни к чему. Имея отдельные наборы тестов, библиотека `Test::Unit` в состоянии просмотреть пространство объектов и объединить их все в один комплект. Поэтому следующий код тоже будет работать (и даже вызывать подразумеваемый по умолчанию исполнитель тестов):

```
require 'test/unit'

require 'tc_set1'
require 'tc_set2'
require 'ts_set3'
```

Библиотека `Test::Unit` располагает и другими возможностями, а в дальнейшем, вероятно, будет усовершенствована. Самую свежую информацию ищите в сети.

16.2. Комплект инструментов ZenTest

Этот великолепный инструментарий написал Райан Дэвис (Ryan Davis). Основной инструмент (`zestest`) – это исполняемая программа, которая генерирует файл с тестами на основе анализа вашего кода.

Тестируемый класс (*class under test* – *CUT*) служит основой тестового класса (*test class* – *TC*). На каждом уровне области видимости в начало имени класса добавляется строка `Test`, а в начало имени метода – строка `test_`. Иногда имена методов приходится «подправлять», например в случае с методом `==` (к имени которого нельзя добавлять никакой префикс) или если имя метода оканчивается на `?`, `!` или `=`.

В листинге 16.2 приведен пример подлежащего тестированию кода:

Листинг 16.2. Класс, подлежащий тестированию

```
class Alpha

  class Beta

    attr_accessor :foo, :bar

    def initialize
    end
```

```
def foo?  
  @foo  
end  
  
end  
  
def initialize  
end  
  
def process  
end  
  
def process!  
end  
  
def ==(other)  
end  
  
def ===(other)  
end  
  
end
```

После запуска команды `zentest file.rb >tfile.rb` получится файл, показанный в листинге 16.3.

Листинг 16.3. Результат работы ZenTest

```
# Code Generated by ZenTest v. 3.2.0  
#  
#      . classname: asrt / meth = ratio%  
#      Alpha::Beta:    0 /    7 =  0.00%  
  
require 'test/unit' unless defined? $ZENTEST and $ZENTEST  
  
class TestAlpha < Test::Unit::TestCase  
  def test_process  
    raise NotImplementedError, 'Need to write test_process'  
  end  
  
  def test_process_bang  
    raise NotImplementedError, 'Need to write test_process_bang'  
  end  
end  
  
module TestAlpha  
  class TestBeta < Test::Unit::TestCase  
    def test_bar  
      raise NotImplementedError, 'Need to write test_bar'  
    end  
  
    def test_bar_equals
```

```

    raise NotImplementedError, 'Need to write test_bar_equals'
  end
  def test_foo
    raise NotImplementedError, 'Need to write test_foo'
  end

  def test_foo_eh
    raise NotImplementedError, 'Need to write test_foo_eh'
  end

  def test_foo_equals
    raise NotImplementedError, 'Need to write test_foo_equals'
  end
end
end

# Number of errors detected: 9

```

Обратите внимание, что каждый тестовый метод возбуждает исключение (предложение `raise`). Идея в том, что все тесты завершаются неудачно, пока вы явно не напишете код.

Исходный файл почему-то не включается в тестовый. Можно поместить в начало тестового файла директиву `require 'file'` или эквивалентную ей (предварительно затребовав `test/unit`). Тогда тестовый код увидит определения ваших классов.

В командной строке можно указать и второй параметр. Если добавить код в тестируемый класс, то тестовые классы окажутся устаревшими. Чем обновлять их вручную, можно сгенерировать только «обновления»:

```
zentest file.rb tfile.rb >tfile2.rb
```

В комплект входит еще одна полезная программа: `unit_diff`. Рассмотрим простое утверждение `assert_equal("foo", "bar")`. Оно приводит к печати следующего сообщения:

```

1) Failure:
testme(Foo) [(irb):7]:
<"foo"> expected but was
<"bar">.

```

Тут все просто и понятно. Но предположим, что каждая из переданных строк (`string`) состояла из нескольких строчек (`line`), а различие начиналось только в седьмой строчке. Программа `unit_diff` как раз призвана навести порядок в таком плохо читаемом тексте. Она работает аналогично утилите `diff`, имеющейся в UNIX; вызывать ее следует как фильтр после обычной тестовой программы.

```
ruby testfile.rb | unit_diff
```

Программа понимает следующие флаги:

- h Справка о порядке запуска
- v Номер версии
- b Не считать различными фрагменты, различающиеся только по количеству пробелов

- с Выполнять контекстное сравнение
- k Не удалять временные файлы
- l Включать в дельту номера строк
- u Выполнять унифицированное сравнение

Программа `autotest` наблюдает за всеми вашими комплектами тестов и запускает те, которые были недавно изменены. Она предназначена для тех лентяев, которым трудно даже ввести имя файла для прогона тестов.

Для пользования этим инструментом нужно соблюдать некоторые соглашения об именах. Правила простые:

- все тесты должны находиться в каталоге `test`;
- имена всех файлов должны начинаться с `test_`;
- имена классов должны начинаться с `Test`;
- подлежащий тестированию код должен находиться в каталоге `lib`;
- файлам в каталоге `lib` должны соответствовать файлы в каталоге `test` (их имена, конечно, должны начинаться с `test_`).

Будучи запущена, программа `autotest` прогоняет тесты по мере их обновления. Если какой-то тест завершается неудачно, она будет прогонять его снова и снова, пока вы не исправите ошибку. Она начнет «сверху», если нажать комбинацию клавиш `Ctrl+C`, и завершится, если нажать `Ctrl+C` во второй раз.

Программа `multiruby` позволяет тестировать код относительно разных версий Ruby. Она входит в комплект `ZenTest`, но пока еще работает не очень стабильно и плохо документирована.

16.3. Работа с отладчиком Ruby

Честно говоря, отладчик Ruby не особенно популярен. Лично я им не пользуюсь и встречал не так уж много людей, которые с ним работали. Но сознавать, что он есть, приятно. Ниже приводится краткое описание работы с ним.

Для вызова отладчика нужно просто затребовать библиотеку `debug` – например, в командной строке:

```
ruby -rdebug myfile.rb
```

В ответ на приглашение вида `(rdb:1)` вы можете вводить различные команды, например `list` для получения текста всей программы или ее части, `step` для пошагового исполнения метода и т.д. Некоторые команды перечислены в таблице 16.1 (полужирным шрифтом набраны сокращения).

В листинге 16.4 приведен код простой программы (даже слишком простой, чтобы ее отлаживать).

Листинг 16.4. Простая программа для демонстрации работы отладчика

```
STDOUT.sync = true
```

```
def palindrome?(word)
  word == word.reverse
end
```

```

def signature(w)
  w.split("").sort.join
end

def anagrams?(w1,w2)
  signature(w1) == signature(w2)
end

print "Give me a word: "
w1 = gets.chomp

print "Give me another word: "
w2 = gets.chomp

verb = palindrome?(w1) ? "is" : "is not"
puts "'#{w1}' #{verb} a palindrome."

verb = palindrome?(w2) ? "is" : "is not"
puts "'#{w2}' #{verb} a palindrome."

verb = anagrams?(w1,w2) ? "are" : "are not"
puts "'#{w1}' and '#{w2}' #{verb} anagrams."

```

Таблица 16.1. Наиболее употребительные команды отладчика

Команда	Описание
break	Установить точку прерывания или получить их список.
delete	Удалить все или некоторые точки прерывания.
catch	Установить точку перехвата или получить их список.
step	Вход внутрь метода.
next	Перейти к следующей строке (без захода внутрь метода).
help	Справка (вывести список всех команд).
quit	Выйти из отладчика.

В листинге 16.5 показан полный сеанс отладки. Некоторые места не слишком понятны, потому что на консоль выводят данные и сама программа, и отладчик.

Листинг 16.5. Сеанс отладки простой программы

```

$ ruby -rdebug db.rb
Debug.rb
Emacs support available.

db.rb:1:STDOUT.sync = true
(rdb:1) b palindrome?
Set breakpoint 1 at db.rb:palindrome?
(rdb:1) b anagrams?

```

Set breakpoint 2 at db.rb:anagrams?

(rdb:1) b

Breakpoints:

1 db.rb:palindrome?

2 db.rb:anagrams?

(rdb:1) n

db.rb:3: def palindrome?(word)

(rdb:1) n

db.rb:7: def signature(w)

(rdb:1) n

db.rb:11: def anagrams?(w1,w2)

(rdb:1) n

db.rb:15: print "Give me a word: "

(rdb:1) n

Give me a word: db.rb:16:w1 = gets.chomp

(rdb:1) live

db.rb:16: undefined local variable or method 'live' for main:Object

(rdb:1) n

live

db.rb:18: print "Give me another word: "

(rdb:1) n

Give me another word: db.rb:19:w2 = gets.chomp

(rdb:1) n

evil

db.rb:21: verb = palindrome?(w1) ? "is" : "is not"

(rdb:1) c

Breakpoint 1, palindrome? at db.rb:palindrome?

db.rb:3: def palindrome?(word)

(rdb:1) n

db.rb:4: word == word.reverse

(rdb:1) word

"live"

(rdb:1) n

db.rb:22: puts "'#{w1}' #{verb} a palindrome."

(rdb:1) verb

"is not"

(rdb:1) n

'live' is not a palindrome.

db.rb:24: verb = palindrome?(w2) ? "is" : "is not"

(rdb:1) n

db.rb:24: verb = palindrome?(w2) ? "is" : "is not"

(rdb:1) n

Breakpoint 1, palindrome? at db.rb:palindrome?

db.rb:3: def palindrome?(word)

(rdb:1) n

db.rb:4: word == word.reverse

(rdb:1) c

'evil' is not a palindrome.

Breakpoint 2, anagrams? at db.rb:anagrams?

```
db.rb:11: def anagrams?(w1, w2)
  (rdb:1) n
db.rb:12: signature(w1) == signature(w2)
  (rdb:1) n
db.rb:28: puts "'#{w1}' and '#{w2}' #{verb} anagrams."
  (rdb:1) verb
"are"
  (rdb:1) c
'live' and 'evil' are anagrams.
```

Если вы затребуете другие библиотеки, то, возможно, придется в самом начале «перешагнуть» через несколько методов. Я рекомендую перед началом отладки установить где-нибудь в собственном коде точку прерывания, а потом выполнить команду `continue` для останова в этой точке.

Отладчик понимает и много других команд. Можно просматривать стек вызовов и перемещаться по нему. Можно «наблюдать» за выражениями и автоматически останавливать выполнение, как только выражение изменится. Можно добавлять выражения в «отображаемый список». Поддерживается работа с несколькими потоками и переключение между ними.

Вероятно, полной и качественной документации вы нигде не найдете. Если возникнет такая необходимость, рекомендуется обратиться к оперативной справке и действовать методом проб и ошибок.

Современные отладчики имеют графический интерфейс. Если вам необходим такой инструмент, загляните в главу 21, где обсуждаются интегрированные среды разработки для Ruby.

16.4. Использование `irb` в качестве отладчика

Библиотеку `ruby-breakpoint` написал Флориан Гросс (Florian Gross). Этот великолепный, несмотря на малый объем, инструмент позволяет расставлять в программе точки прерывания методом `breakpoint`. Когда в процессе исполнения встречается точка прерывания, запускается сеанс `irb` (программа интерактивной работы с Ruby `irb` подробно рассматривается в главе 21).

Эта библиотека не входит в стандартный дистрибутив. Установить ее можно, например, выполнив команду `gem install ruby-breakpoint`.

Внесем несколько изменений в программу из листинга 16.4. Поместим в начало директиву `require 'breakpoint'` и добавим вызов метода `breakpoint` после обоих обращений к `gets`:

```
require 'breakpoint'

# ...
w2 = gets.chomp

breakpoint

# ...
```


Теперь запустим ее. В следующем протоколе сеанса показано, как мы входим в `irb`, после чего можем делать все что угодно – в частности, вызывать ранее определенные методы и изменять значения переменных.

```
$ ruby myprog.rb
Give me a word: parental
Give me another word: prenatal
Executing break point at myprog.rb:23
irb(main):001:0> w1
=> "parental"
irb(main):002:0> w2
=> "prenatal"
irb(main):003:0> palindrome?(w1)
=> false
irb(main):004:0> palindrome?("detartrated")
=> true
irb(main):005:0> signature(w1)
=> "aaelnprt"
irb(main):006:0> quit
'parental' is not a palindrome.
'prenatal' is not a palindrome.
'parental' and 'prenatal' are anagrams.
```

Особенно подкупает, что отлаживаемая программа может быть не только командной или текстовой. Существует клиент `drb` (распределенный Ruby), который позволяет удаленно отлаживать программу Ruby, работающую в другом процессе.

Чтобы воспользоваться этой возможностью, нужно включить вызов следующего метода в отлаживаемую программу (естественно, до первого обращения к методу `breakpoint`):

```
Breakpoint.activate_drb("druby://127.0.0.1:2001", "localhost")
# Запустить сервер на порту 2001 машины localhost.
```

Запустите клиент командой `breakpoint_client`. Каждые три секунды он будет пытаться установить соединение с сервером, пока это не получится или вы не завершите его принудительно.

```
$ breakpoint_client druby://localhost:2001
No connection to breakpoint service at druby://localhost:2001
(DRb::DRbConnError)
Tries to connect will be made every 3 seconds...
```

После установления соединения вы можете и не получить приглашение `irb`. Программа будет выполняться до точки прерывания – вот тогда-то вы и увидите приглашение.

Дополнительную информацию об этой библиотеке поищите в документации, которая входит в комплект поставки.

16.5. Измерение покрытия кода

Очень полезно знать, какие части программы не были протестированы, а следовательно, нуждаются в автономных тестах. Иногда и сам инструмент для замера по-

крытия может обнаружить ошибки. Допустим, в программе есть предложение `if`, которое «должно» исполняться примерно в половине всех случаев. Если выясняется, что оно не исполняется никогда, значит, имеет место ошибка.

Командную утилиту `rcov` (и соответствующую библиотеку) написал Маурицио Фернандес (Mauricio Fernandez). Устанавливается она в виде gem-пакета.

В простейшем случае для ее запуска достаточно указать имя вашей программы в качестве параметра:

```
rcov myfile.rb
```

Одновременно с исполнением вашей программы `rcov` будет собирать статистику. По умолчанию она создает каталог `coverage`, в котором вы найдете HTML-файлы. В файле `index.html` представлены сводные результаты и ссылки на исходные тексты, где строки, которые хотя бы раз исполнялись, подсвечены.

Из-за цветового кодирования трудно привести черно-белый снимок с экрана. Но сам инструмент настолько прост, что, потратив пару минут, вы сможете увидеть все сами.

Хотя программа `rcov` полезна даже в стандартном режиме, она понимает порядка 30 различных параметров. Можно указать каталог для выходных файлов, образцы имен файлов, для которых собирать и не собирать статистику, задать режим сортировки по именам файлов и многое другое. Можно выводить результаты в текстовом виде и даже запросить цветную диаграмму покрытия. Рекомендую прочитать поставляемую документацию, запросить справку командой `rcov -h` и... получать удовольствие.

Можно использовать `rcov` и в качестве библиотеки для написания аналогичных инструментов анализа. Ее API состоит из трех основных классов:

- `Rcov::FileStatistics` позволяет отличить исполняемые предложения от комментариев (и тем самым уточнить статистику покрытия);
- `Rcov::CodeCoverageAnalyzer` применяется для трассировки выполнения, возвращает информацию о покрытии и счетчики выполненных предложений;
- `Rcov::CallSiteAnalyzer` нужен для того, чтобы понять, где определены методы и откуда они вызываются.

Обсуждение API далеко выходит за рамки этого раздела. Почитайте документацию и начинайте экспериментировать.

16.6. Измерение производительности

Я не люблю уделять слишком много внимания оптимизации скорости. В общем случае нужно правильно выбрать алгоритм и придерживаться здравого смысла.

Конечно, быстрое действие имеет значение. Иногда даже очень большое. Однако начинать думать об этом на раннем этапе цикла разработки – ошибка. Как говорится, «преждевременная оптимизация – источник всех зол»; эту мысль впервые высказал Хоар (Hoare), а потом подтвердил Кнут (Knuth). Или, перефразируя, «сначала пусть работает правильно, а уж потом быстро». На уровне отдельного

приложения эта рекомендация обычно оказывается хорошим эвристическим правилом, хотя для больших систем она, быть может, и не так актуальна.

Я бы еще добавил: «Не оптимизируйте, пока не измерите».

Это не такое уж серьезное ограничение. Просто не приступайте к переработке ради скорости, пока не ответите на два вопроса: «Действительно ли программа работает медленно? Какие именно ее части снижают производительность?»

Второй вопрос важнее, чем кажется на первый взгляд. Программисты часто уверены, что и так знают, на что программа тратит большую часть времени, но специальные исследования убедительно свидетельствуют о том, что в среднем эти догадки имеют очень мало общего с действительностью. «Теоретическая» оптимизация для большинства из нас – плохая идея.

Нам нужны объективные измерения. Профилировщик нужен.

В комплект поставки Ruby входит профилировщик `profile`. Для его вызова достаточно включить библиотеку:

```
ruby -rprofile myprog.rb
```

Рассмотрим листинг 16.6. Эта программа открывает файл `/usr/share/dict/words` и ищет в нем анаграммы. Затем смотрит, у каких слов оказалось больше всего анаграмм, и распечатывает их.

Листинг 16.6. Поиск анаграмм в словаре

```
words = File.readlines("/usr/share/dict/words")
words.map! {|x| x.chomp }

hash = {}
words.each do |word|
  key = word.split("").sort.join
  hash[key] ||= []
  hash[key] << word
end

sizes = hash.values.map {|v| v.size }
most = sizes.max
list = hash.find_all {|k,v| v.size == most }

puts "Ни у одного слова нет более #{most-1} анаграмм."
list.each do |key,val|
  anagrams = val.sort
  first = anagrams.shift
  puts "Слово #{first} имеет #{most-1} анаграмм:"
  anagrams.each {|a| puts "    #{a}" }
end

num = 0
hash.keys.each do |key|
  n = hash[key].size
  num += n if n > 1
end
```

```
puts
puts "Всего слов в словаре: #{words.size},"
puts "из них имеют анаграммы: #{num}."
```

Наверняка вам интересно, какие получились результаты. Вот какие:

Ни у одного слова нет более 14 анаграмм.

Слово alerts имеет 14 анаграмм:

```
alters
artels
estral
laster
lastre
rastle
ratels
relast
resalt
salter
slater
staler
stelar
talers
```

Всего слов в словаре: 483523,
из них имеют анаграммы: 79537.

На моем компьютере этот файл содержит более 483 000 слов, и программа работала чуть меньше 18 секунд. Как вы думаете, на что ушло это время? Попробуем выяснить. Профилировщик выдал более 100 строк, отсортированных в порядке убывания времени. Мы покажем только первые 20:

%	cumulative	self	self	total		
time	seconds	seconds	calls	ms/call	ms/call	name
42.78	190.93	190.93	15	12728.67	23647.33	Array#each
10.78	239.04	48.11	1404333	0.03	0.04	Hash#[]
7.04	270.48	31.44	2	15720.00	25575.00	Hash#each
5.66	295.73	25.25	483523	0.05	0.05	String#split
5.55	320.51	24.78	1311730	0.02	0.02	Array#size
3.64	336.76	16.25	1	16250.00	25710.00	Array#map
3.24	351.23	14.47	483524	0.03	0.03	Array#sort
3.12	365.14	13.91	437243	0.03	0.03	Fixnum#==
3.04	378.72	13.58	483526	0.03	0.03	Array#join
2.97	391.98	13.26	437244	0.03	0.03	Hash#default
2.59	403.53	11.55	437626	0.03	0.03	Hash#[]=
2.43	414.38	10.85	483568	0.02	0.02	Array#<<
2.29	424.59	10.21	1	10210.00	13430.00	Array#map!
1.94	433.23	8.64	437242	0.02	0.02	Fixnum#<=>
1.86	441.54	8.31	437244	0.02	0.02	Fixnum#>
0.72	444.76	3.22	483524	0.01	0.01	String#chomp!
0.11	445.26	0.50	4	125.00	125.00	Hash#keys
0.11	445.73	0.47	1	470.00	470.00	Hash#values
0.06	446.00	0.27	1	270.00	270.00	IO#readlines
0.05	446.22	0.22	33257	0.01	0.01	Fixnum#+

Видно, что больше всего времени программа тратит в методе `Array#each`. Это понятно: ведь цикл выполняется для каждого слова и на каждой итерации делает довольно много. Среднее значение в данном случае сбивает с толку, поскольку почти все время уходит на первый вызов `each`, а остальные 14 (см. `anagrams.each`) выполняются очень быстро.

Мы также видим, что `Hash#[]` – дорогая операция (главным образом потому, что часто выполняется); на 1.4 миллиона вызовов было потрачено почти 11 секунд.

Обратите внимание, что метод `readlines` оказался чуть ли не в самом конце списка. Эта программа тратит время не на ввод/вывод, а на вычисления. На чтение всего файла ушло всего-то четверть секунды.

Но этот пример не показывает, в чем истинная ценность профилирования. В программе нет ни методов, ни классов. На практике вы, скорее всего, увидите свои методы среди системных. И тогда будете точно знать, какие из ваших методов входят в число первых 20 «пожирателей времени».

Надо ясно понимать, что профилировщик Ruby (видно, по иронии судьбы) работает медленно. Он подключается к программе во многих местах и следит за ее выполнением на низком уровне (причем сам написан на чистом Ruby). Так что не удивляйтесь, если ваша программа в ходе сеанса профилирования будет работать на несколько порядков медленнее. В нашем примере она работала 7 минут 40 секунд (460 секунд), то есть в 25 раз медленнее обычного.

Помимо профилировщика, есть еще один низкоуровневый инструмент – стандартная библиотека `benchmark`, которая тоже полезна для измерения производительности.

Один из способов ее применения – вызвать метод `Benchmark.measure` и передать ему блок.

```
require 'benchmark'
```

```
file = "/usr/share/dict/words"
```

```
result = Benchmark.measure { File.readlines(file) }
```

```
puts result
```

```
# Выводится:      0.350000   0.070000   0.420000 ( 0.418825)
```

Этот метод выводит следующую информацию:

- время, затраченное процессором в режиме пользователя (в секундах);
- время, затраченное процессором в режиме ядра (в секундах);
- полное затраченное время – сумму вышеупомянутых величин;
- время работы программы (по часам).

Для сравнения производительности отдельных участков удобен метод `Benchmark.bm`. Передайте ему блок, а он сам передаст блоку объект формирования отчета. Если вызвать этот объект, передав ему метку и блок, то он выведет метку, а за ней временные характеристики блока. Пример:

```
require 'benchmark'
```

```
n = 200_000
```

```
s1 = ""
s2 = ""
s3 = ""
Benchmark.bm do |rep|
  rep.report("str <<      ") { n.times { s1 << "x" } }
  rep.report("str.insert ") { n.times { s3.insert(-1,"x") } }
  rep.report("str +=      ") { n.times { s2 += "x" } }
end
```

Здесь мы сравниваем три способа добавить символ в конец строки, дающие один и тот же результат. Чтобы можно было получить более точные цифры, каждая операция выполняется 200 000 раз. Вот что вышло:

user	system	total	real
str <<	0.180000	0.000000	0.180000 (0.174697)
str.insert	0.200000	0.000000	0.200000 (0.200479)
str +=	15.250000	13.120000	28.370000 (28.375998)

Обратите внимание, что последний вариант на два порядка медленнее остальных. Почему? Какой урок можно извлечь отсюда?

Вы можете предположить, что оператор + почему-то работает медленно, но дело в другом. Это единственный из трех способов, который не работает с одним и тем же объектом, а каждый раз создает новый.

Стало быть, вывод такой: создание объекта – дорогая операция. Библиотека Benchmark может преподать много подобных уроков, но я все же рекомендую сначала заняться высокоуровневым профилированием.

16.7. Объекты печати

Метод inspect (и вызывающий его метод p) предназначен для вывода объектов в виде, понятном человеку. В этом смысле он является связующим звеном между тестированием и отладкой, поэтому рассмотрение его в этой главе оправданно.

Проблема в том, что результат, формируемый методом p, бывает трудно читать. Из-за этого и появилась библиотека pp, добавляющая одноименный метод.

Рассмотрим следующий искусственный пример объекта my_obj:

```
class MyClass

  attr_accessor :alpha, :beta, :gamma

  def initialize(a,b,c)
    @alpha, @beta, @gamma = a, b, c
  end

end

x = MyClass.new(2, 3, 4)
y = MyClass.new(5, 6, 7)
z = MyClass.new(7, 8, 9)
```

```
my_obj = { x => y, z => [:p, :q] }
```

```
p my_obj
```

Вызов метода `p` печатает следующее:

```
{#<MyClass:0xb7eed86c @beta=3, @alpha=2,
 @gamma=4>=>#<MyClass:0xb7eed72c @beta=6, @alpha=5, @gamma=7>,
 #<MyClass:0xb7eed704 @beta=8, @alpha=7, @gamma=9>=>[:p, :q]}
```

Все правильно и в общем-то даже читаемо. Но... некрасиво. А давайте затребуем библиотеку `pp` и воспользуемся предоставляемым ей методом `pp`:

```
require 'pp'
```

```
# ...
```

```
pp my_obj
```

Теперь вывод приобретает такой вид:

```
{#<MyClass:0xb7f7a050 @alpha=7, @beta=8, @gamma=9>=>[:p, :q],
 #<MyClass:0xb7f7a1b8 @alpha=2, @beta=3, @gamma=4>=>
  #<MyClass:0xb7f7a078 @alpha=5, @beta=6, @gamma=7>}
```

Мы получили хотя бы пробелы и разбиение на строки. Уже лучше. Но можно пойти еще дальше. Предположим, что в классе `MyClass` определен специальный метод `pretty_print`:

```
class MyClass

  def pretty_print(printer)
    printer.text "MyClass(#{@alpha}, #{@beta}, #{@gamma})"
  end

end
```

Аргумент `printer` передается вызывающей программой (или методом `pp`). Это аккумулятор текста, являющийся экземпляром класса `PP`; мы вызываем его метод `text` и передаем ему текстовое представление `self`. Вот что получается в результате:

```
(MyClass(7, 8, 9)=>[:p, :q], MyClass(2, 3, 4)=>MyClass(5, 6, 7))
```

Разумеется, можно настроить поведение по своему вкусу. Можно, например, печатать переменные экземпляра на разных строчках с отступами.

На самом деле в библиотеке `pp` есть много средств для подготовки ваших классов к совместной работе с методом `pp`. Методы `object_group`, `seplist`, `breakable` и прочие позволяют управлять расстановкой запятых, разбиением на строки и другими способами форматирования. Дополнительную информацию можно найти в документации на сайте <http://ruby-doc.org>.

16.8. Заключение

В этой главе мы рассмотрели некоторые подходы к тестированию (преимущественно, к автономному тестированию компонентов). Мы познакомились с библиотекой `Test::Unit` и комплектом инструментов `ZenTest`.

Мы бросили беглый взгляд на отладчик Ruby, а также показали, как с помощью библиотеки `ruby-breakpoint` можно переходить в интерактивную оболочку `irb` для проведения сеанса отладки.

Мы рассмотрели инструмент для анализа покрытия кода `rcov` и обсудили, для чего могут понадобиться такие измерения. Наконец, остановились на профилировании и замере временных характеристик программы на Ruby.

Предположим, что вы работаете над программой, которую собираетесь предложить обществу. Что вы делаете по завершении тестирования? Пришло время задуматься над оформлением дистрибутивного пакета и способами его распространения. Этим мы и займемся в следующей главе.

Глава 17. Создание пакетов и распространение программ

.....
*Все больше и больше продуктов – и в первую очередь астирин –
выпускается в упаковке, защищенной до такой степени,
что потребитель уже и воспользоваться ими не может.*
Дэйв Бэрри

Эта глава посвящена вопросу о том, как проявлять уважение к конечному пользователю своей программы. Если у вас конечного пользователя нет или вы не хотите казаться вежливым, можете пропустить ее.

Есть две вещи, о которых типичный программист не желает задумываться: документация и процедура установки. Ниже я попытаюсь убедить вас не пугаться ни того, ни другого. Нашлись люди, приложившие максимум усилий к написанию библиотек и инструментов, упрощающих решение этих задач.

Однако данная глава оказалась короткой. Дополнительную информацию о затронутых в ней темах можете почерпнуть, изучая сделанное другими.

Один из самых замечательных имеющихся инструментов – программа RDoc, написанная Дэйвом Томасом. С нее и начнем.

17.1. Программа RDoc

RDoc – не единственный инструмент документирования для Ruby: его предшественником является программа RDTOOL. Но во многих отношениях RDoc превосходит своих конкурентов; к тому же он наиболее распространенный, по крайней мере, в США.

Замечательной особенностью RDoc является то, что он пытается вывести нечто полезное, даже если в исходном тексте вообще нет комментариев. Для этого он анализирует текст программы и собирает информацию обо всех классах, модулях, константах, методах и т. д.

Тем самым вы можете получить более или менее полезный HTML-файл из исходного текста, не содержащего никакой внутренней документации. Если раньше не пробовали, попробуйте сейчас.

Но это еще не все. RDoc также пытается ассоциировать найденные комментарии с конкретными частями программы. Общее правило таково: блочный комментарий, предшествующий определению (скажем, класса или метода), считается описанием этого определения.

Если просто вызвать RDoc для какого-нибудь исходного текста на Ruby, будет создан каталог doc, в который помещаются все выходные файлы (этот стандартный шаблон уже неплох, но есть и другие). Откройте в браузере файл index.html и изучите его.

В листинге 17.1 приведен простой (почти ничего не содержащий) исходный файл. Все определенные в нем методы пусты. Но RDoc даже в таком случае формирует симпатичную страницу документации (рис. 17.1).

Листинг 17.1. Простой исходный файл

```
require 'foo'

# Внешний класс MyClass.

class MyClass
  CONST = 237

  # Внутренний класс MyClass::Alpha...

  class Alpha

    # Класс The MyClass::Alpha::Beta...

    class Beta
      # Метод класса Beta mymeth1.
      def mymeth1
        end
      end

      # Метод класса Alpha mymeth2.
      def mymeth2
        end
    end

    # Инициализировать объект.

    def initialize(a,b,c)
      end

    # Создать объект со значениями по умолчанию

    def self.create
      end

    # и метод экземпляра.

    def do_something
      end

  end
end
```

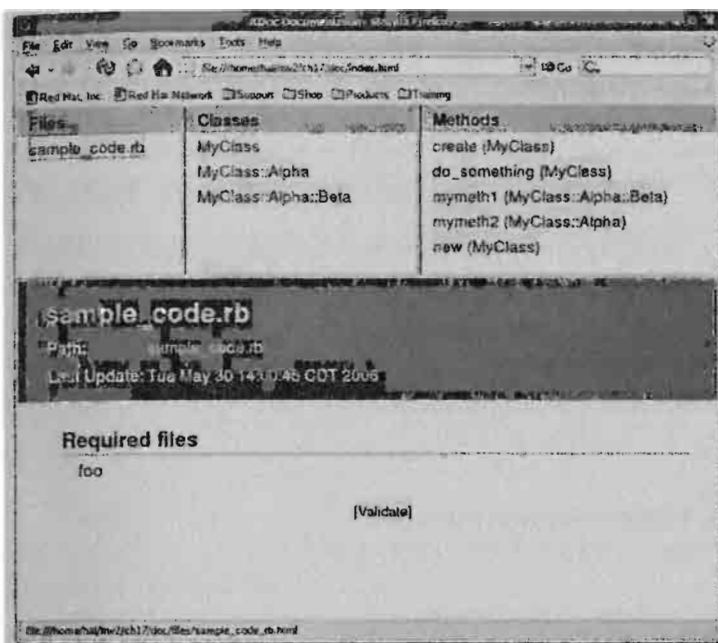


Рис. 17.1. Выходной файл, формируемый программой RDoc по исходному тексту в листинге 17.1

В этом разделе мы обсудим еще две полезные функции. Имя каждого метода является ссылкой, при щелчке по которой открывается исходный текст метода. При изучении библиотеки это оказывается исключительно полезно – документация API ссылается на сам код.

Кроме того, когда программа RDoc распознает URL, она помещает в выходной файл гиперссылку. По умолчанию текст гиперссылки совпадает с самим URL, но это можно изменить. Если перед URL поместить в фигурных скобках какой-нибудь описательный текст, то он и станет содержимым ссылки. Если текст состоит из одного слова, фигурные скобки можно опустить.

17.1.1. Простая разметка

Если вы хотите «разукрасить» вывод, то редактировать HTML-файлы вручную необязательно. На самом деле даже нежелательно, так как при повторной генерации документации ваши изменения будут затерты.

RDoc располагает собственным механизмом разметки, поэтому можно включать в исходный текст информацию о форматировании. Правила языка разметки выбраны так, что текст в редакторе выглядит «естественно», но вместе с тем может быть легко преобразован в HTML.

В листинге 17.2 приведено несколько примеров разметки; дополнительную информацию ищите в книге «Programming Ruby» или в документации по RDoc. На рис. 17.2 показано, во что преобразуется текст в листинге 17.2 (нижний фрейм).

Листинг 17.2. Пример разметки для RDoc

```
# This block comment will be detected and
# included in the rdoc output.
#
=begin rdoc
So will this one. Note the presence of the "rdoc"
tag on the begin-line. This is to distinguish the
block comment as belonging to rdoc as opposed to
being read by some other tool.
=end
=begin rdoc
Here are some formatting tricks.

Boldface, italics, and "code" (without spaces):
This is *bold*, this is _italic_, and this is +code+.
```

With spaces:

```
This is a bold phrase. Have you read Intruder
in the Dust? Don't forget to require thread
at the top.
```

```
= First level heading
== Second level heading
=== Third level heading
```

Here's a horizontal rule:

```
---
```

Here's a list:

```
- item one
- item two
- item three
```

```
=end
```

```
=begin
```

```
This block comment is untagged and will not show up in
rdoc output. Also, I'm not putting blank lines between
the comments, as this will terminate the comments until
some real program source is seen. If this comment had
been before the previous one, processing would have
stopped here until program text appeared.
```

```
=end
```

В листинге 17.2 приведено несколько правил разбора документов, принятых в RDoc. Не все они интуитивно очевидны. Считается, что пустые строки завершают блок комментариев, даже если вслед за пустой строкой сразу идет еще один такой блок.

Внутри блока комментариев, начинающегося со знака #, можно отключить копирование текста в выходной файл, вставив строку `--` (а следующая такая строка

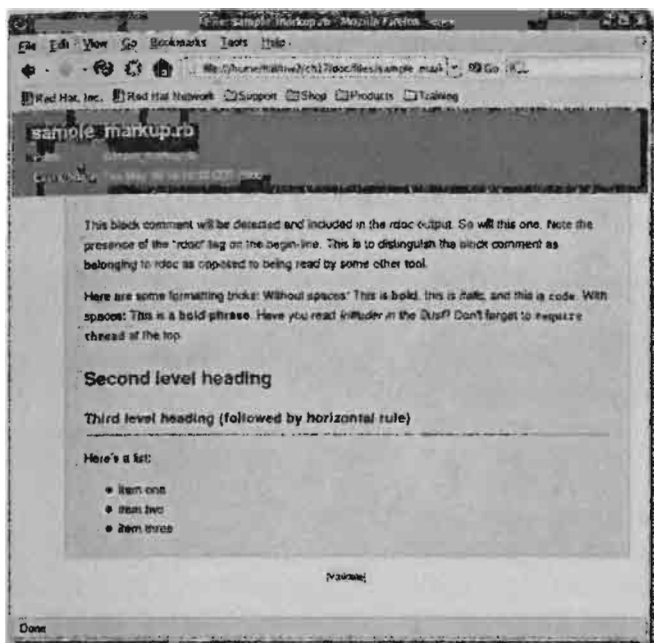


Рис. 17.2. Результат работы RDoc для примера из листинга 17.2

вновь включает копирование). Ведь не все комментарии должны быть частью пользовательской документации.

Отметим еще, что если используются маркеры `=begin` и `=end`, то после `=begin` должен находиться тег `rdoc`, иначе RDoc проигнорирует весь блок целиком. Это сделано во избежание конфликтов с более старыми инструментами, в которых такие блоки активно использовались.

17.1.2. Более сложное форматирование

RDoc позволяет довольно точно управлять тем, какие части исходного текста документируются и как к ним следует относиться. Для этого служат специальные теги в комментариях (модификаторы документации).

Одним из самых важных является тег `:nodoc:`, отключающий вывод документации для определенного фрагмента. Обычно он ставится в той же строке, где начинается определение класса или метода.

```
class Alpha      # :nodoc:
  .class Beta
    # ...
  end
  # ...
end
```

Здесь класс Alpha не будет документироваться. Однако тег `:nodoc:` не является рекурсивным – класс Beta документировается. Если желательно рекурсивное

поведение, укажите `:nodoc: all`. В следующем примере игнорируются оба класса `Gamma` и `Delta`:

```
class Alpha      # :nodoc: all
  class Beta
    # ...
  end
  # ...
end
```

Имеется также модификатор `:doc:` с прямо противоположным смыслом. Он включает документацию для фрагментов, которые иначе не были бы документированы.

Модификатор `:notnew:` специальный; он предотвращает документирование метода `new` (на основе существующего метода `initialize`).

Если вы хотите дать осмысленные имена параметрам `yield`, воспользуйтесь темом `:yields:..`. Например, если в самом тексте употребляются ничего не значащие имена `x` и `y`, то в документации их можно заменить.

```
def iterate      # :yields: element, index
  # ...
  yield x, i
end
```

Некоторые теги используются только внутри блока комментариев, например:

- `:include:` – включить содержимое указанного файла в документацию. При этом будут сформированы подходящие отступы;
- `:title:` – задать заголовок документа;
- `:main:` – задать начальную страницу документации.

Дополнительную информацию вы найдете в книге «Programming Ruby» или в любом онлайн-справочном руководстве.

17.2. Установка и подготовка пакета

У пользователя должно быть ощущение «коробочного продукта». Как пользователи мы готовы подписаться под этим тезисом обеими руками, но как разработчики не любим заниматься вопросами создания пакетов и установки.

К счастью, в Ruby все это не так болезненно, как в некоторых других языках и средах. Вы обязательно должны знать о библиотеке `setup` и системе `RubyGems` – «родных» для Ruby инструментах создания пакетов и развертывания.

17.2.1. Библиотека `setup.rb`

Автором библиотеки `setup.rb` является Минеро Аоки (Minero Aoki). Он же разработал библиотеку `install.rb`, которая сейчас используется реже.

Кто-то скажет, что по мере развития системы `RubyGems` все это становится не актуальным. А кто-то возразит, что у `gem`-пакетов есть свои проблемы (технические, политические и пр.). А кто-то считает, что «добропорядочный гражданин» должен включать `setup.rb` даже в `gem`-пакет (упрощая задачу перепакетирования, например для создания Linux-дистрибутива). Решать вам.

Половина дела будет сделана, если вы скопируете все файлы в нужные места. Вы должны организовать свой архив простым и разумным образом (создав каталоги с предопределенными именами).

Предположим, что дистрибутив содержит единственный пакет в архиве (наиболее распространенный случай). Тогда дерево каталогов организуется примерно так (причем файл `setup.rb` помещается на верхний уровень).

```
top_level/  
  setup.rb  
  metaconfig (необязательно)  
  lib/  
  ext/  
    myext/  
  bin/  
  data/  
  conf/  
  man/  
  test/
```

Пустые каталоги можно опускать. Ниже описано назначение каждого каталога:

- `lib` – программы на Ruby;
- `ext` – расширения Ruby (написанные на C);
- `myext` – имя расширения (на том же уровне могут располагаться и другие расширения); в каталоге каждого расширения должен находиться либо файл `extconf.rb`, либо `MANIFEST`;
- `bin` – команды;
- `data` – файлы данных;
- `conf` – конфигурационные файлы;
- `man` – страницы руководства;
- `test` – автономные тесты и другие тестовые программы.

В общем случае эти файлы никак не обрабатываются, а просто копируются в нужное место. Но для специализированной настройки предусмотрены места подключения к каждому этапу процесса.

Три основных этапа – это `config`, `setup` и `install`, вызываемые пользователем именно в таком порядке (на последнем шаге могут потребоваться полномочия `root` или, по крайней мере, выполнение `sudo`).

Для подключения к этапу вы просто помещаете в нужный каталог написанную на Ruby программу с известным именем. Например, если необходимо перед обработкой сделать что-то нестандартное с файлом `lib/foobar`, следует создать файл `lib/foobar/pre-setup.rb` и поместить в него произвольный код.

Имя файла формируется следующим образом: префикс `pre` или `post`, дефис, имя задачи. Определены следующие имена задач: `config`, `setup`, `install`, `test`, `clean` и `dist-clean`.

В библиотеке `setup.rb` есть понятия *каталога исходных файлов*, или *исходного каталога* (`source directory`) и *каталога объектных файлов*, или *объектного каталога*

(object directory). Как правило, вы должны читать из исходного каталога и записывать в текущий каталог.

Существует «API для подключения» (hook API), упрощающий решение ряда задач. Приведем некоторые определенные в нем методы:

- `get_config_key(key)` – принимает в качестве параметра ключ и возвращает ассоциированное с ним значение (например, `get_config('prefix')` возвращает путь, определенный с помощью конфигурационного параметра `--prefix`);
- `set_config_key(key, val)` – устанавливает значение конфигурационного параметра;
- `config_key(key)` – то же, что `get_config_key`;
- `curr_srcdir` – текущий исходный каталог;
- `curr_objdir` – текущий объектный каталог;
- `srcfiles(rel_path=".")` – список всех файлов в каталоге с путем `rel_path` (относительно текущего исходного каталога).

На верхнем уровне может находиться файл `metaconfig`. Если он есть, то в нем задаются некоторые глобальные конфигурационные параметры. Для этой цели имеется специальный «`metaconfig API`» – небольшой набор вспомогательных методов, в частности:

- `add_path_config(confname, default, description)` – определяет конфигурационный параметр, являющийся путем; задаются имя и значение по умолчанию. При вызове с флагом `--help` эта информация печатается;
- `add_bool_config(confname, default, description)` – аналог `add_path_config`, но описывается булевский параметр.

Дополнительную информацию по этим API можно найти в актуальной онлайн-новой документации.

17.2.2. Система RubyGems

Идея и название системы RubyGems принадлежат Райану Ливенгуду (Ryan Leavengood), но текущая реализация зародилась на ночной вечеринке, состоявшейся после Международной конференции по Ruby 2003 года в Остине, штат Техас. Первый вариант кода написали Чэд Фаулер (Chad Fowler), Джим Вайрих (Jim Weirich), Дэвид Алан Блэк (David Alan Black), Рич Килмер (Rich Kilmer) и Пол Брэннен (Paul Brannan). С тех пор к ним присоединились и другие; особо стоит отметить Эрика Ходеля (Eric Hodel) и Райана Дэвиса (Ryan Davis).

В настоящее время RubyGems, наверное, самая распространенная система создания пакетов, хотя до сих пор не включена в дистрибутив. Я полагаю, что после устранения нескольких мелких огрехов она станет настоящим стандартом для Ruby.

Как и повсюду в этой главе, мы рассматриваем вопрос с точки зрения разработчика. Вы узнаете, как представлять плоды своего труда в виде gem-пакета, но о манипулировании пакетами извне мы говорить не будем. Это тема другого раздела.

Возникает естественный вопрос: «Зачем нужно использовать gem-пакеты?» Вот перечень лишь некоторых их достоинств:

- простота установки и удаления;
- поддержка нескольких версий;
- управление зависимостями;
- механизм запроса и поиска пакетов.

Имя gem-пакета обычно состоит из короткого описательного слова, за которым следует дефис и стандартный номер версии в формате «основной.дополнительный.рабочий», который ныне принят почти повсеместно (конечно, каждая часть номера может состоять из нескольких цифр). Мы настоятельно рекомендуем пользоваться механизмом рациональной нумерации версии; если вы с ним не знакомы, поищите описание в сети.

Для построения gem-пакета нужно начать с создания предопределенного дерева каталогов (примерно такого же, как для `setup`). На верхнем уровне неплохо поместить файл `README`, в который включаются информация об авторе и способе связи с ним, авторские права, лицензионное соглашение, перечень известных ошибок и т. д. Если вы напишете этот файл в формате `RDoc`, его можно будет включить в состав `HTML`-документации проекта.

Для построения gem-пакета необходимо создать его спецификацию (`gemspec`). Это один из тех случаев, когда стирается грань между кодом и данными. Спецификация – это просто исполняемый файл на языке `Ruby`:

```
require 'rubygems'
SPEC = Gem::Specification.new do |s|
  s.name      = "Drummer"
  s.version   = "1.0.2"
  s.author    = "H. Thoreau"
  s.email     = "cabin@waldenpond.com"
  s.homepage  = "http://waldenpond.com/Drummer"
  s.platform  = Gem::Platform::RUBY
  s.summary   = "A Ruby app for those who march to a different drummer"
  s.files     = Dir["./**"] + Dir["**/*"]
  s.test_file = "test/ts_drum.rb"
  s.has_rdoc  = true
  s.require_path = "lib"
  s.extra_rdoc_files = ["README", "ChangeLog"]
  s.add_dependency "KirbyBase", ">=2.5.0"
end
```

Назначение многих атрибутов понятно из названия. Но есть и другие, хотя обычно они не так важны и встречаются реже. Обратитесь к онлайн-официальной документации по `RubyGems`.

Имя файла спецификации, можно создать gem-пакет двумя способами. Во-первых, можно просто выполнить файл (в конце концов, это ведь обычная программа на `Ruby`). Тогда в текущем каталоге будет создан пакет с указанным именем. Альтернативно можно воспользоваться командой `gem build`, которой передается имя файла спецификации. Результат будет один и тот же.

Созданный gem-пакет можно распространять через Web или любым другим способом. Я настоятельно рекомендую для управления проектом пользоваться сайтом RubyForge. Если загруженный на него архив содержит файл спецификации, то пакет будет создан автоматически. В следующем разделе мы поговорим о сайте RubyForge и архиве приложений Ruby (RAA).

17.3. RubyForge и RAA

Есть два основных источника библиотек и приложений на языке Ruby (помимо стандартного дистрибутива). Первый и самый старый – это архив приложений Ruby (Ruby Application Archive или RAA), расположенный на официальном сайте Ruby по адресу <http://raa.ruby-lang.org>.

Название не вполне корректно, потому что в архиве хранятся главным образом библиотеки, а не приложения. Кроме того, надо понимать, что это прежде всего репозиторий метаданных, а не самих файлов. Поэтому постоянной головной болью пользователей RAA являются «битые» ссылки. Но все равно это очень важный ресурс, поскольку многие «рубисты» сначала ищут решение в нем, а потом уже прибегают к услугам поисковой машины.

В состав метаданных входят название и описание проекта, имя и электронный адрес автора, URL домашней страницы проекта или адрес, по которому его можно скачать. (Понятно, что в интересах самого автора предоставить как можно больше информации и поддерживать ее в актуальном состоянии.)

Когда писалась эта книга, в архиве RAA было свыше 1400 проектов, распределенных по четырем основным рубрикам и 243 подрубрикам. Основными являются рубрики «Приложения», «Документация», «Библиотеки» и «Перенос на другие платформы». Из них последняя самая маленькая, так как задача переноса Ruby далеко не тривиальна. Рубрика «Документация» немногим больше. Более 90% содержимого RAA попадает в рубрики «Библиотеки» и «Приложения». Скорее всего, все созданное вами окажется там же.

Рубрика «Приложения» состоит из 94 подкатегорий, включающих 449 проектов. Некоторые подрубрики достаточно обширны, например: «Языки», «Базы данных», «Редакторы», «Криптография», «Wiki», «Графические интерфейсы», «HTML». В рубрике «Библиотеки» 952 проекта, отнесенных к 127 подрубрикам, например: «Аудио», «Календарь», «Химия», «Файловые системы», «Численные методы» и «Телефония». Разумная классификация помогает быстрее найти интересующий вас проект. Конечно, на сайте есть и система поиска, но все равно иметь возможность просмотра иерархии полезно.

Впрочем, следует отметить, что классификация не всегда полна и в ряде случаев не ортогональна. Если точной подрубрики для вашего проекта еще нет, пойдите на компромисс или создайте новую подрубрику (но осторожно).

Само собой разумеется, что не следует усугублять проблему «битых» ссылок. Попробуйте разместить свой проект на сайте, который никуда не денется. Время от времени просматривайте информацию о своих проектах в RAA и проверяйте, не изменился ли URL проекта или ваш электронный адрес.

Второй лучший друг «рубиста» – сайт RubyForge (<http://rubyforge.org>). Это огромное хранилище проектов, построенное по образцу SourceForge. Конечно, по своей философии RubyForge сильно отличается от RAA. На этом сайте хранятся не только метаданные, но и сами файлы программ, документация и т. д. В настоящее время там находится свыше 1600 проектов (больше чем в RAA) и зарегистрировано более 6800 пользователей.

RubyForge поддерживает списки рассылки и другие подобные сервисы. Если в вашем проекте заинтересовано много людей, будь то пользователи или разработчики, то такие механизмы помогут им контактировать друг с другом.

У каждого проекта есть страница скачивания, которая поддерживается более или менее автоматически. Когда вы загружаете новые файлы и создаете новую версию, она сразу же становится доступна всем, кто заходит на сайт.

У сайта RubyForge есть приятная особенность: если загруженный пакет содержит спецификацию gem-пакета, этот пакет будет создан и предложен для скачивания автоматически. Вот так и должно работать все в этом мире!

Помимо страницы скачивания у каждого проекта может быть собственная страница. Многие, однако, обходятся без нее; заходя на страницу такого проекта, вы нередко видите всего лишь подразумеваемое по умолчанию сообщение «Coming soon!» (Скоро). Не думайте, что проект закрыт или заморожен – просто его владелец не счел нужным заполнять эту страницу. Если вы еще не поняли, к чему я веду, то скажу ясно – создавайте начальные страницы для своих проектов, даже если они размещены где-то в другом месте. Не так уж сложно поддерживать синхронизацию между ними.

Обычно URL проекта начинается с короткого имени, например <http://rvmagick.rubyforge.org> (это библиотека RMagick Тима Хантера).

Для доступа к своему проекту на RubyForge, как правило, достаточно воспользоваться программой `scp` и скопировать файлы в точно определенные места. Для многих операций, скажем управления списками рассылки, имеется Web-интерфейс.

Для организации более выразительного интерфейса с проектами на сайте RubyForge существуют также командные (и, быть может, даже графические) утилиты. Но они не являются «официальными» и широкого распространения не получили. Чтобы узнать, поддерживаются ли такие инструменты в настоящее время, займитесь поиском в сети.

17.4. Заключение

В этой главе мы изучили основы документирования проектов с помощью инструмента `RDoc`, а также познакомились с программой `setup.rb` и системой `RubyGems` – двумя наиболее популярными способами создания дистрибутивных пакетов.

Наконец, мы бегло рассмотрели сайт RubyForge и архив приложений Ruby (RAA), которые позволяют оповещать о созданном программном обеспечении и распространять его. В следующей главе мы снова переключим передачу и поговорим об интересной и сложной предметной области: программировании для сетей и, в частности, Web.

Глава 18. Сетевое программирование

Если торговец в разговоре с вами произносит слово «сеть», скорее всего, он желает всучить свою визитную карточку. Но в устах программиста это слово обозначает электронное взаимодействие физически удаленных машин – неважно, находятся они в разных углах комнаты, в разных районах города или в разных частях света.

Для программистов сеть чаще всего ассоциируется с набором протоколов TCP/IP – тем языком, на котором неслышно беседуют миллионы машин, подключенных к сети Интернет. Несколько слов об этом наборе, перед тем как мы перейдем к конкретным примерам.

Концептуально сетевое взаимодействие принято представлять в виде различных уровней (или слоев) абстракции. Самый нижний – *канальный уровень*, на котором происходит аппаратное взаимодействие; о нем мы говорить не будем. Сразу над ним расположен *сетевой уровень*, который отвечает за перемещение пакетов в сети – это епархия протокола IP (Internet Protocol). Еще выше находится *транспортный уровень*, на котором расположились протоколы TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). Далее мы видим *прикладной уровень* – это мир telnet, FTP, протоколов электронной почты и т. д.

Можно обмениваться данными непосредственно по протоколу IP, но обычно так не поступают. Чаще нас интересуют протоколы TCP и UDP.

Протокол TCP обеспечивает надежную связь между двумя компьютерами (хостами). Он упаковывает данные в пакеты и распаковывает их, подтверждает получение пакетов, управляет тайм-аутами и т. д. Поскольку протокол надежный, приложению нет нужды беспокоиться о том, получил ли удаленный хост посланные ему данные.

Протокол UDP гораздо проще: он отправляет пакеты (датаграммы) удаленному хосту, как будто это двоичные почтовые открытки. Нет никакой гарантии, что данные будут получены, поэтому протокол называется ненадежным (а, следовательно, приложению придется озаботиться дополнительными деталями).

Ruby поддерживает сетевое программирование на низком уровне (главным образом по протоколам TCP и UDP), а также и на более высоких, в том числе по протоколам telnet, FTP, SMTP и т. д.

На рис. 18.1 представлена иерархия классов, из которой видно, как организована поддержка сетевого программирования в Ruby. Показаны классы HTTP и некоторые другие столь же высокого уровня; кое-что для краткости опущено.

Отметим, что большая часть этих классов прямо или косвенно наследует классу IO. Следовательно, мы можем пользоваться уже знакомыми методами данного класса.

Попытка документировать все функции всех показанных классов завела бы нас далеко за рамки этой книги. Я лишь покажу, как можно применять их к решению

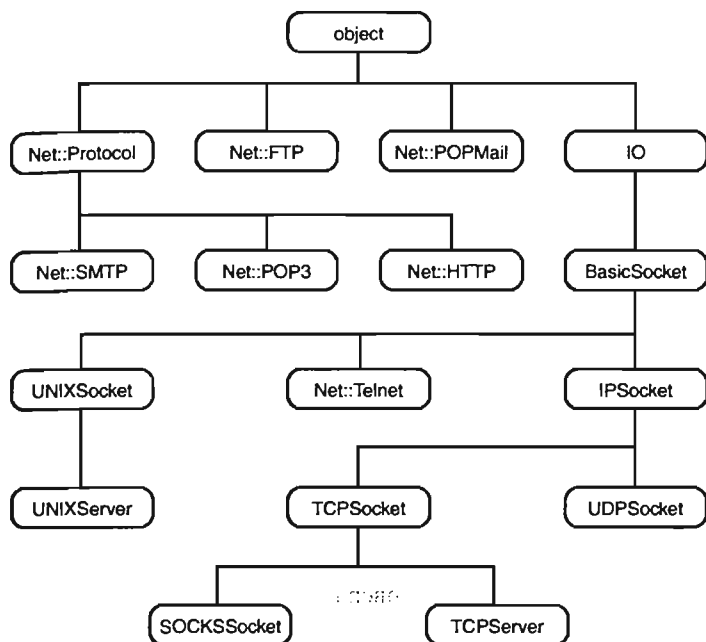


Рис. 18.1. Часть иерархии наследования для поддержки сетевого программирования в Ruby

конкретных задач, сопровождая примеры краткими пояснениями. Полный перечень всех методов вы можете найти в справочном руководстве на сайте ruby-doc.org.

Ряд важных областей применения в данной главе вообще не рассматривается, поэтому сразу упомянем о них. Класс `Net::Telnet` упоминается только в связи с NTP-серверами в разделе 18.2.2; этот класс может быть полезен не только для реализации собственного telnet-клиента, но и для автоматизации всех задач, поддерживающих интерфейс по протоколу telnet.

Библиотека `Net::FTP` также не рассматривается. В общем случае автоматизировать обмен по протоколу FTP несложно и с помощью уже имеющихся клиентов, так что необходимость в этом классе возникает реже, чем в прочих.

Класс `Net::Protocol`, являющийся родительским для классов `HTTP`, `POP3` и `SMTP` полезен скорее для разработки новых сетевых протоколов, но эта тема в данной книге не обсуждается.

На этом завершим краткий обзор и приступим к рассмотрению низкоуровневого сетевого программирования.

18.1. Сетевые серверы

Жизнь сервера проходит в ожидании входных сообщений и ответах на них. Не исключено, что для формирования ответа требуется серьезная обработка, например

обращение к базе данных, но с точки зрения сетевого взаимодействия сервер просто принимает запросы и отправляет ответы.

Но даже это можно организовать разными способами. Сервер может в каждый момент времени обслуживать только один запрос или иметь несколько потоков. Первый подход проще реализовать, зато у второго есть преимущества, когда много клиентов одновременно обращается с запросами.

Можно представить себе сервер, единственное назначение которого состоит в том, чтобы облегчить общение между клиентами. Классические примеры – чат-серверы, игровые серверы и файлообменные сети.

18.1.1. Простой сервер: время дня

Рассмотрим самый простой сервер, который вы только способны представить. Пусть некоторая машина располагает такими точными часами, что ее можно использовать в качестве стандарта времени. Такие серверы, конечно, существуют, но взаимодействуют не по тому тривиальному протоколу, который мы обсудим ниже. (В разделе 18.2.2 приведен пример обращения к подобному серверу по протоколу telnet.)

В нашем примере все запросы обслуживаются в порядке поступления однопоточным сервером. Когда приходит запрос от клиента, мы возвращаем строку, содержащую текущее время. Ниже приведен код сервера:

```
require "socket"

PORT = 12321
HOST = ARGV[0] || 'localhost'

server = UDPSocket.open      # Применяется протокол UDP...
server.bind nil, PORT

loop do
  text, sender = server.recvfrom(1)
  server.send(Time.new.to_s + "\n", 0, sender[3], sender[1])
end
```

А это код клиента:

```
require "socket"
require "timeout"

PORT = 12321
HOST = ARGV[0] || 'localhost'

socket = UDPSocket.new
socket.connect(HOST, PORT)

socket.send("", 0)
timeout(10) do
  time = socket.gets
  puts time
end
```

Чтобы сделать запрос, клиент посылает пустой пакет. Поскольку протокол UDP ненадежен, то, не получив ответа в течение некоторого времени, мы завершаем работу по тайм-ауту.

В следующем примере такой же сервер реализован на базе протокола TCP. Он прослушивает порт 12321; запросы к этому порту можно посылать с помощью программы telnet (или клиента, код которого приведен ниже).

```
require "socket"

PORT = 12321

server = TCPServer.new(PORT)

while (session = server.accept)
  session.puts Time.new
  session.close
end
```

Обратите внимание, как просто использовать класс TCPServer. Вот TCP-версия клиента:

```
require "socket"

PORT = 12321
HOST = ARGV[0] || "localhost"

session = TCPSocket.new(HOST, PORT)
time = session.gets
session.close
puts time
```

18.1.2. Реализация многопоточного сервера

Некоторые серверы должны обслуживать очень интенсивный поток запросов. В таком случае эффективнее обрабатывать каждый запрос в отдельном потоке.

Ниже показана реализация сервера текущего времени, с которым мы познакомились в предыдущем разделе. Он работает по протоколу TCP и создает новый поток для каждого запроса.

```
require "socket"

PORT = 12321

server = TCPServer.new(PORT)

while (session = server.accept)
  Thread.new(session) do |my_session|
    my_session.puts Time.new
    my_session.close
  end
end
```

Многопоточность позволяет достичь высокого параллелизма. Вызывать метод `join` не нужно, поскольку сервер исполняет бесконечный цикл, пока его не остановят вручную.

Код клиента, конечно, остался тем же самым. С точки зрения клиента, поведение сервера не изменилось (разве что он стал более надежным).

18.1.3. Пример: сервер для игры в шахматы по сети

Не всегда нашей конечной целью является взаимодействие с самим сервером. Иногда сервер – всего лишь средство для соединения клиентов друг с другом. В качестве примера можно привести файлообменные сети, столь популярные в 2001 году. Другой пример – серверы для мгновенной передачи сообщений, например ICQ, и разного рода игровые серверы.

Давайте напишем скелет шахматного сервера. Мы не имеем в виду программу, которая будет играть в шахматы с клиентом. Нет, наша задача – связать клиентов так, чтобы они могли затем играть без вмешательства сервера.

Предупреждаю, что ради простоты показанная ниже программа ничего не знает о шахматах. Логика игры просто заглушена, чтобы можно было сосредоточиться на сетевых аспектах.

Для установления соединения между клиентом и сервером будем использовать протокол TCP. Можно было бы остановиться и на UDP, но этот протокол ненадежен, и нам пришлось бы использовать тайм-ауты, как в одном из примеров выше.

Клиент может передать два поля: свое имя и имя желательного противника. Для идентификации противника условимся записывать его имя в виде `user:hostname`; мы употребили двоеточие вместо напрашивающегося знака `@`, чтобы не вызывать ассоциаций с электронным адресом, каковым эта строка не является.

Когда от клиента приходит запрос, сервер сохраняет сведения о клиенте у себя в списке. Если поступили запросы от обоих клиентов, сервер посылает каждому из них сообщение; теперь у каждого клиента достаточно информации для установления связи с противником.

Есть еще вопрос о выборе цвета фигур. Оба партнера должны как-то договориться о том, кто каким цветом будет играть. Для простоты предположим, что цвет назначает сервер. Первый обратившийся клиент будет играть белыми (и, стало быть, ходить первым), второй – черными.

Уточним: компьютеры, которые первоначально были клиентами, начиная с этого момента общаются друг с другом напрямую; следовательно, один из них становится сервером. Но на эту семантическую тонкость я не буду обращать внимания.

Поскольку клиенты посылают запросы и ответы попеременно, причем сеанс связи включает много таких обменов, будем пользоваться протоколом TCP. Следовательно, клиент, который на самом деле играет роль «сервера», создает объект `TCPServer`, а клиент на другом конце – объект `TCPSocket`. Будем предполагать, что номер порта для обмена данными заранее известен обоим партнерам (разумеется, у каждого из них свой номер порта).

Мы только что описали простой протокол прикладного уровня. Его можно было бы сделать и более хитроумным.

Сначала рассмотрим код сервера (листинг 18.1). Чтобы его было проще запускать из командной строки, создадим поток, который завершит сервер при нажатии клавиши Enter. Сервер многопоточный – он может одновременно обслуживать нескольких клиентов. Данные о пользователях защищены мьютексом, ведь теоретически несколько потоков могут одновременно попытаться добавить новую запись в список.

Листинг 18.1. Шахматный сервер

```
require "thread"
require "socket"

PORT = 12000
HOST = "96.97.98.99" # Заменить этот IP-адрес.

# Выход при нажатии клавиши Enter.
waiter = Thread.new do
  puts "Нажмите Enter для завершения сервера."
  gets
  exit
end

$mutex = Mutex.new
$list = {}

def match?(p1, p2)
  return false if !$list[p1] or !$list[p2]

  if ($list[p1][0] == p2 and $list[p2][0] == p1)
    true
  else
    false
  end
end

def handle_client(sess, msg, addr, port, ipname)
  $mutex.synchronize do
    cmd, player1, player2 = msg.split
    # Примечание: от клиента мы получаем данные в виде user:hostname,
    # но храним их в виде user:address.
    p1short = player1.dup # Короткие имена
    p2short = player2.split(":")[0] # (то есть не ":address").
    player1 << ":{addr}" # Добавить IP-адрес клиента.

    user2, host2 = player2.split(":")
    host2 = ipname if host2 == nil
    player2 = user2 + ":" + IPSocket.getaddress(host2)

    if cmd != "login"
      puts "Ошибка протокола: клиент послал сообщение #{msg}."
    end
  end
end
```

```

$list[player1] = {player2, addr, port, ipname, sess}

if match?(player1, player2)
  # Имена теперь переставлены: если мы попали сюда, значит
  # player2 зарегистрировался первым.
  p1 = $list[player1]
  p2 = $list[player2]
  # ID игрока = name:ipname:color
  # Цвет: 0=белый, 1=черный
  plid = "#{p1short}:#{p1[3]}:1"
  p2id = "#{p2short}:#{p2[3]}:0"
  sess1 = p1[4]
  sess2 = p2[4]
  sess1.puts "#{p2id}"
  sess2.puts "#{plid}"
  sess1.close
  sess2.close
end
end
end

text = nil

$server = TCPServer.new(HOST, PORT)
while session = $server.accept do
  Thread.new(session) do |sess|
    text = sess.gets
    puts "Получено: #{text}" # Чтобы знать, что сервер получил.
    domain, port, ipname, ipaddr = sess.peeraddr
    handle_client sess, text, ipaddr, port, ipname
    sleep 1
  end
end

waiter.join # Выходим, когда была нажата клавиша Enter.

```

Метод `handle_client` сохраняет информацию о клиенте. Если запись о таком клиенте уже существует, то каждому клиенту посылается сообщение о том, где находится другой партнер. Этим обязанности сервера исчерпываются.

Клиент (листинг 18.2) оформлен в виде единственной программы. При первом запуске она становится ТСП-сервером, а при втором – ТСП-клиентом. Честно говоря, решение о том, что сервер будет играть белыми, совершенно произвольно. Вполне можно было бы реализовать приложение так, чтобы цвет не зависел от подобных деталей.

Листинг 18.2. Шахматный клиент

```

require "socket"
require "timeout"

ChessServer = '96.97.98.99' # Заменить этот IP-адрес.

```

```
ChessServerPort = 12000
PeerPort        = 12001

WHITE, BLACK = 0, 1
Colors = %w[White Black]

def draw_board(board)
  puts <<-EOF
+-----+
| Заглушка! Шахматная доска... |
+-----+
  EOF
end

def analyze_move(who, move, num, board)
  # Заглушка - черные всегда выигрывают на четвертом ходу.
  if who == BLACK and num == 4
    move << " Мат!"
  end
  true # Еще одна заглушка - любой ход считается допустимым.
end

def my_move(who, lastmove, num, board, sock)
  ok = false
  until ok do
    print "\nВаш ход: "
    move = STDIN.gets.chomp
    ok = analyze_move(who, move, num, board)
    puts "Недопустимый ход" if not ok
  end
  sock.puts move
  move
end

def other_move(who, move, num, board, sock)
  move = sock.gets.chomp
  puts "\nПротивник: #{move}"
  move
end

if ARGV[0]
  myself = ARGV[0]
else
  print "Ваше имя? "
  myself = STDIN.gets.chomp
end

if ARGV[1]
  opponent_id = ARGV[1]
else
  print "Ваш противник? "
```

```
opponent_id = STDIN.gets.chomp
end

opponent = opponent_id.split(":")[0]    # Удалить имя хоста.
# Обратиться к серверу

socket = TCPSocket.new(ChessServer, ChessServerPort)

response = nil

socket.puts "login #{myself} #{opponent_id}"
socket.flush
response = socket.gets.chomp

name, ipname, color = response.split ":"
color = color.to_i

if color == BLACK                                # Цвет фигур другого игрока.
  puts "\nУстанавливается соединение..."

  server = TCPServer.new(PeerPort)
  session = server.accept

  str = nil
  begin
    timeout(30) do
      str = session.gets.chomp
      if str != "ready"
        raise "Ошибка протокола: получено сообщение о готовности #{str}."
      end
    end
  rescue TimeoutError
    raise "Не получено сообщение о готовности от противника."
  end

  puts "Ваш противник #{opponent}... у вас белые.\n"

  who = WHITE
  move = nil
  board = nil                                # В этом примере не используется.
  num = 0
  draw_board(board)                          # Нарисовать начальное положение для белых.
  loop do
    num += 1
    move = my_move(who, move, num, board, session)
    draw_board(board)
    case move
    when "resign"
      puts "\nВы сдались. #{opponent} выиграл."
      break
    when /Checkmate/
```

```

    puts "\nВы поставили мат #{opponent}!"
    draw_board(board)
    break
end
move = other_move(who, move, num, board, session)
draw_board(board)
case move
  when "resign"
    puts "\n#{opponent} сдался... вы выиграли!"
    break
  when /Checkmate/
    puts "\n#{opponent} поставил вам мат."
    break
end
end
else
    # Мы играем черными.
    puts "\nУстанавливается соединение..."

    socket = TCPSocket.new(ipname, PeerPort)
    socket.puts "ready"

    puts "Ваш противник #{opponent}... у вас черные.\n"

    who = BLACK
    move = nil
    board = nil
    num = 0
    draw_board(board)
    # В этом примере не используется.
    # Нарисовать начальное положение.

    loop do
        num += 1
        move = other_move(who, move, num, board, socket)
        draw_board(board)
        # Нарисовать доску после хода белых.
        case move
          when "resign"
            puts "\n#{opponent} сдался... вы выиграли!"
            break
          when /Checkmate/
            puts "\n#{opponent} поставил вам мат."
            break
          end
        move = my_move(who, move, num, board, socket)
        draw_board(board)
        case move
          when "resign"
            puts "\nВы сдались. #{opponent} выиграл."
            break
          when /Checkmate/
            puts "\n#{opponent} поставил вам мат."
            break
          end
        end
    end
end

```

```

end
end
socket.close
end

```

Я определил этот протокол так, что черные посылают белым сообщение «ready», чтобы партнер знал о готовности начать игру. Затем белые делают первый ход. Ход посылается черным, чтобы клиент мог нарисовать такую же позицию на доске, как у другого игрока.

Повторю, приложение ничего не знает о шахматах. Вместо проверки допустимости хода вставлена заглушка; проверка выполняется локально, то есть на той стороне, где делается ход. Никакой реальной проверки нет – заглушка всегда говорит, что ход допустим. Кроме того, мы хотим, чтобы имитация игры завершалась после нескольких ходов, поэтому мы написали программу так, что черные всегда выигрывают на четвертом ходу. Победа обозначается строкой «Checkmate!» в конце хода. Эта строка печатается на экране соперника и служит признаком выхода из цикла.

Помимо «традиционной» шахматной нотации (например, «Р-К4») существует еще «алгебраическая», которую многие предпочитают. Но написанный код вообще не имеет представления о том, какой нотацией мы пользуемся.

Поскольку это было несложно сделать, мы позволяем игроку в любой момент сдаться. Рисование доски тоже заглушено. Желающие могут реализовать грубый рисунок, выполненный ASCII-символами.

Метод `my_move` всегда относится к локальному концу, метод `other_move` – к удаленному.

В листинге 18.3 приведен протокол сеанса. Действия клиентов нарисованы друг против друга.

Листинг 18.3. Протокол сеанса шахматной игры

```

% ruby chess.rb Hal
Carablanca;deephought.org

```

```

Устанавливается соединение...
Ваш противник Carablanca... у вас белые.
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

```

Ваш ход: N-QB3
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

```

Противник: Р-К4
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

```

% ruby chess.rb
Hal:deepdoodoo.org

```

```

Устанавливается соединение...
Ваш противник Hal... у вас черные.
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

```

Противник: N-QB3
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

```

Ваш ход: Р-К4
+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: P-K4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Противник: B-QB4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: B-QB4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Противник: Q-KR5

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: N-KB3

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Противник: QxP Checkmate!

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Carablanca поставил вам мат.

Противник: P-K4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: B-QB4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Противник: B-QB4

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: Q-KR5

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Противник: N-KB3

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Ваш ход: QxP

```

+-----+
| Заглушка! Шахматная доска... |
+-----+

```

Вы поставили мат Hal!

18.2. Сетевые клиенты

Иногда сервер пользуется хорошо известным протоколом — тогда нам надо лишь спроектировать клиента, который общается с сервером на понятном тому языке.

В разделе 18.1 мы видели, что это можно сделать с помощью протоколов TCP или UDP. Но чаще применяются протоколы более высокого уровня, например HTTP или SNMP. Рассмотрим несколько примеров.

18.2.1. Получение истинно случайных чисел из Web

.....
*Всякий, кто пытается сгенерировать случайное число, пользуясь
 детерминированными средствами, безусловно, живет во грехе.
 Джон фон Нейман*

В модуле kernel есть функция rand, которая возвращает случайное число, но вот беда — число-то не является истинно случайным. Если вы математик, криптограф

или еще какой-нибудь педант, то назовете эту функцию генератором псевдослучайных чисел, поскольку она пользуется алгебраическими методами для детерминированного порождения последовательности чисел. Стороннему наблюдателю эти числа представляются случайными и даже обладают необходимыми статистическими свойствами, но рано или поздно последовательность начнет повторяться. Мы можем даже намеренно (или случайно) повторить ее, задав ту же самую загрузку.

Но природные процессы считаются истинно случайными. Поэтому при розыгрыше призов в лотерее счастливицы определяются лототроном, который хаотично выбрасывает шары. Другие источники случайности – радиоактивный распад или атмосферный шум.

Есть источники случайных чисел и в Web. Один из них – сайт www.random.org, который мы задействуем в следующем примере.

Программа в листинге 18.4 имитирует подбрасывание пяти обычных (шестигранных) костей. Конечно, игровые фанаты могли бы увеличить число граней до 10 или 20, но тогда стало бы сложно рисовать ASCII-картинки.

Листинг 18.4. Случайное бросание костей

```
require 'net/http'

HOST = "www.random.org"
RAND_URL = "/cgi-bin/randnum?col=5&"

def get_random_numbers(count=1, min=0, max=99)
  path = RAND_URL + "num=#{count}&min=#{min}&max=#{max}"
  connection = Net::HTTP.new(HOST)
  response, data = connection.get(path)
  if response.code == "200"
    data.split.collect { |num| num.to_i }
  else
    []
  end
end

DICE_LINES = [
  "+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ ",
  "|       | | *   | | *   | | * * | | * * | | * * | | * * |",
  "|  *   | |   | |   | |   | |   | |   | |   | |   | |   |",
  "|       | | *   | | *   | | * * | | * * | | * * | | * * |",
  "+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ "
]

DIE_WIDTH = DICE_LINES[0].length/6

def draw_dice(values)
  DICE_LINES.each do |line|
    for v in values
      print line[(v-1)*DIE_WIDTH, DIE_WIDTH]
    end
  end
end
```



```

        print " "
    end
    puts
end
end

draw_dice(get_random_numbers(5, 1, 6))

```

Здесь мы воспользовались классом `Net::HTTP` для прямого взаимодействия с Web-сервером. Считайте, что эта программа – узкоспециализированный браузер. Мы формируем URL и пытаемся установить соединение; когда оно будет установлено, мы получаем ответ, возможно, содержащий некие данные. Если код ответа показывает, что ошибок не было, то можно разобрать полученные данные. Предполагается, что исключения будут обработаны вызывающей программой.

Посмотрим на вариацию этой идеи. Что если вы захотели бы применить случайные числа в каком-нибудь приложении? Поскольку обслуживающая программа на стороне сервера позволяет указать количество возвращаемых чисел, то было бы логично сохранить их в буфере. Учитывая, что при обращении к удаленному серверу задержки неизбежны, следует сразу заполнить буфер во избежание лишних запросов по сети.

В листинге 18.5 эта мысль реализована. Буфер заполняется отдельным потоком и совместно используется всеми экземплярами класса. Размер буфера и «нижняя отметка» (`@slack`) настраиваются; какие значения задать в реальной программе, зависит от величины задержки при обращении к серверу и от того, как часто приложение выбирает случайное число из буфера.

Листинг 18.5. Генератор случайных чисел с буферизацией

```

require "net/http"
require "thread"

class TrueRandom

  def initialize(min=nil,max=nil,buff=nil,slack=nil)
    @buffer = []
    @site = "www.random.org"
    if ! defined? @init_flag
      # Принять умолчания, если они не были заданы явно И
      # это первый созданный экземпляр класса...
      @min = min || 0
      @max = max || 1
      @bufsize = buff || 1000
      @slacksize = slack || 300
      @mutex = Mutex.new
      @thread = Thread.new { fillbuffer }
      @init_flag = TRUE      # Значение может быть любым.
    else
      @min = min || @min
      @max = max || @max
      @bufsize = buff || @bufsize
    end
  end

  def fillbuffer
    # ...
  end
end

```

```

    @slacksize = slack || @slacksize
  end
  @url = "/cgi-bin/randnum" +
    "?num=#@bufsize&min=#@min&max=#@max&col=1"
end

def fillbuffer
  h = Net::HTTP.new(@site, 80)
  resp, data = h.get(@url, nil)
  @buffer += data.split
end

def rand
  num = nil
  @mutex.synchronize { num = @buffer.shift }
  if @buffer.size < @slacksize
    if ! @thread.alive?
      @thread = Thread.new { fillbuffer }
    end
  end
  if num == nil
    if @thread.alive?
      @thread.join
    else
      @thread = Thread.new { fillbuffer }
      @thread.join
    end
    @mutex.synchronize { num = @buffer.shift }
  end
  num.to_i
end

end

t = TrueRandom.new(1,6,1000,300)

count = {1=>0, 2=>0, 3=>0, 4=>0, 5=>0, 6=>0}

10000.times do |n|
  x = t.rand
  count[x] += 1
end

p count

# При одном прогоне:
# {4=>1692, 5=>1677, 1=>1678, 6=>1635, 2=>1626, 3=>1692}

```

18.2.2. Запрос к официальному серверу времени

Как мы и обещали, приведем программу для обращения к NTP-серверу в сети (NTP – Network Time Protocol (синхронизирующий сетевой протокол). Показанный ниже код заимствован с небольшой переработкой у Дэйва Томаса.

```
require "net/telnet"
timeserver = "www.fakedomain.org"

local = Time.now.strftime("%H:%M:%S")
tn = Net::Telnet.new("Host"      => timeserver,
                    "Port"      => "time",
                    "Timeout"   => 60,
                    "Telnetmode" => false)

msg = tn.recv(4).unpack('N')[0]
# Преобразовать смещение от точки отсчета
remote = Time.at(msg - 2208988800).strftime("%H:%M:%S")

puts "Местное      : #{local}"
puts "Удаленное   : #{remote}"
```

Мы устанавливаем соединение и получаем четыре байта. Они представляют 32-разрядное число в сетевом (тупоконечном) порядке байтов. Это число преобразуется в понятную форму, а затем – из смещения от точки отсчета в объект Time.

Мы не указали имя реального сервера. Дело в том, что его полезность часто зависит от того, где вы находитесь. Кроме того, многие серверы ограничивают доступ, так что для запроса вы должны получить разрешение или хотя бы уведомить владельца. Поисковая машина поможет найти открытый NTP-сервер в радиусе 1000 км от вас.

18.2.3. Взаимодействие с POP-сервером

Многие серверы электронной почты пользуются почтовым протоколом (Post Office Protocol – POP). Имеющийся в Ruby класс POP3 позволяет просматривать заголовки и тела всех сообщений, хранящихся для вас на сервере, и обрабатывать их как вы сочтете нужным. После обработки сообщения можно удалить.

Для создания объекта класса Net::POP3 нужно указать доменное имя или IP-адрес сервера; номер порта по умолчанию равен 110. Соединение устанавливается только после вызова метода start (которому передается имя и пароль пользователя).

Вызов метода mails созданного объекта возвращает массив объектов класса POPMail. (Имеется также итератор each для перебора этих объектов.)

Объект POPMail соответствует одному почтовому сообщению. Метод header получает заголовки сообщения, а метод all – заголовки и тело (у метода all, как мы вскоре увидим, есть и другие применения).

Фрагмент кода стоит тысячи слов. Вот пример обращения к серверу с последующей распечаткой темы каждого сообщения:

```
require "net/pop"

pop = Net::POP3.new("pop.fakedomain.org")
pop.start("gandalf", "mellon")      # Имя и пароль пользователя.
pop.mails.each do |msg|
```

```
puts msg.header.grep /^Subject: /
end
```

Метод `delete` удаляет сообщение с сервера. (Некоторые серверы требуют, чтобы POP-соединение было закрыто методом `finish`, только тогда результат удаления становится необратимым.) Вот простейший пример фильтра спама:

```
require "net/pop"

pop = Net::POP3.new("pop.fakedomain.org")
pop.start("gandalf", "mellon") # Имя и пароль пользователя.
pop.messages.each do |msg|
  if msg.all =~ /.make money fast.*/
    msg.delete
  end
end
pop.finish
```

Отметим, что при вызове метода `start` можно также задавать блок. По аналогии с методом `File.open` в этом случае открывается соединение, выполняется блок, а затем соединение закрывается.

Метод `all` также можно вызывать с блоком. В блоке просто перебираются все строки сообщения, как если бы мы вызвали итератор `each` для строки, возвращенной методом `all`.

```
# Напечатать все строки в обратном порядке... полезная штука!
msg.all { |line| print line.reverse }
# То же самое...
msg.all.each { |line| print line.reverse }
```

Методу `all` можно также передать объект. В таком случае для каждой строки (`line`) в полученной строке (`string`) будет вызван оператор конкатенации (`<<`). Поскольку в различных объектах он может быть определен по-разному, в результате такого обращения возможны самые разные действия:

```
arr = [] # Пустой массив.
str = "Mail: " # String.
out = $stdout # Объект IO.

msg.all(arr) # Построить массив строчек.
msg.all(str) # Конкатенировать с str.
msg.all(out) # Вывести на stdout.
```

Наконец, покажем еще, как вернуть только тело сообщения, игнорируя все заголовки.

```
module Net

  class POPMail

    def body
      # Пропустить байты заголовка
      self.all[self.header.size..-1]
    end
  end
end
```

end

end

end

Если вы предпочитаете протокол IMAP, а не POP3, обратитесь к разделу 18.2.5.

18.2.4. Отправка почты по протоколу SMTP

.....

Это понял бы и пятилетний ребенок.

Дайте мне пятилетнего ребенка.

Гроучо Маркс

Название «простой протокол электронной почты» (Simple Mail Transfer Protocol – SMTP) не вполне правильно. Если он и «простой», то только по сравнению с более сложными протоколами.

Конечно, библиотека `smtp.rb` скрывает от программиста большую часть деталей протокола. Но, на наш взгляд, эта библиотека интуитивно не вполне очевидна и, пожалуй, слишком сложна (надеемся, что в будущем это изменится). В этом разделе мы приведем несколько примеров, чтобы помочь вам освоиться.

В классе `Net::SMTP` есть два метода класса: `new` и `start`. Метод `new` принимает два параметра: имя сервера (по умолчанию `localhost`) и номер порта (по умолчанию 25).

Метод `start` принимает следующие параметры:

- `server` – доменное имя или IP-адрес SMTP-сервера; по умолчанию это `"localhost"`;
- `port` – номер порта, по умолчанию 25;
- `domain` – доменное имя отправителя, по умолчанию `ENV["HOSTNAME"]`;
- `account` – имя пользователя, по умолчанию `nil`;
- `password` – пароль, по умолчанию `nil`;
- `authtype` – тип авторизации, по умолчанию `:cram_md5`.

Обычно большую часть этих параметров можно не задавать.

Если метод `start` вызывается «нормально» (без блока), то он возвращает объект класса `SMTP`. Если же блок задан, то этот объект передается прямо в блок.

У объекта `SMTP` есть метод экземпляра `sendmail`, который обычно и занимается всеми деталями отправки сообщения. Он принимает три параметра:

- `source` – строка или массив (или любой объект, у которого есть итератор `each`, возвращающий на каждой итерации одну строку);
- `sender` – строка, записываемая в поле «from» сообщения;
- `recipients` – строка или массив строк, описывающие одного или нескольких получателей.

Вот пример отправки сообщения с помощью методов класса:

```
require 'net/smtp'
```

```
msg = <<EOF
Subject: Разное
... пришла пора
Подумать о делах:
О башмаках, о сургуче,
Капусте, королях.
И почему, как суп в котле,
Кипит вода в морях.
EOF
```

```
Net::SMTP.start("smtp-server.fake.com") do |smtp|
  smtp.sendmail msg, 'walrus@fake1.com', 'alice@fake2.com'
end
```

Поскольку в начале строки находится слово `Subject:`, то получатель сообщения увидит тему Разное.

Имеется также метод экземпляра `start`, который ведет себя практически так же, как метод класса. Поскольку почтовый сервер определен в методе `new`, то задавать его еще и в методе `start` не нужно. Поэтому этот параметр пропускается, а остальные не отличаются от параметров, передаваемых методу класса. Следовательно, сообщение можно послать и с помощью объекта `SMTP`:

```
require 'net/smtp'
```

```
msg = <<EOF
Subject: Ясно и логично
"С другой стороны, - добавил Тарарам, -
если все так и было, то все именно так и было.
Если же все было бы так, то все не могло бы быть
не так. Но поскольку все было не совсем так, все
было совершенно не так. Ясно и логично!"
EOF
```

```
smtp = Net::SMTP.new("smtp-server.fake.com")
smtp.start
smtp.sendmail msg, 'tweedledee@fake1.com', 'alice@fake2.com'
```

Если вы еще не запутались, добавим, что метод экземпляра может принимать еще и блок:

```
require 'net/smtp'
```

```
msg = <<EOF
Subject: Моби Дик
Зовите меня Измаил.
EOF
```

```
addressees = ['reader1@fake2.com', 'reader2@fake3.com']
```

```
smtp = Net::SMTP.new("smtp-server.fake.com")
smtp.start do |obj|
  obj.sendmail msg, 'narrator@fake1.com', addressees
end
```

Как видно из примера, объект, переданный в блок (`obj`), не обязан называться так же, как объект, от имени которого вызывается метод (`smtp`). Кроме того, хочу подчеркнуть: несколько получателей можно представить в виде массива строк.

Существует еще метод экземпляра со странным названием `ready`. Он похож на `sendmail`, но есть и важные различия. Задаются только отправитель и получатели, тело же сообщения конструируется с помощью объекта `adapter` класса `Net::NetPrivate::WriteAdapter`, у которого есть методы `write` и `append`. Адаптер передается в блок, где может использоваться произвольным образом*:

```
require "net/smtp"

smtp = Net::SMTP.new("smtp-server.fake1.com")

smtp.start

smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj.write "Пошли вдвоем, пожалуй.\r\n"
  obj.write "Уж вечер небо навзничью распяло,\r\n"
  obj.write "Как пациента под ножом наркоз...\r\n"
end
```

Отметим, что пары символов «возврат каретки», «перевод строки» обязательны (если вы хотите разбить сообщение на строчки). Читатели, знакомые с деталями протокола, обратят внимание на то, что сообщение «завершается» (добавляется точка и слово «QUIT») без нашего участия.

Можно вместо метода `write` воспользоваться оператором конкатенации:

```
smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj << "В гостиной разговаривают тети\r\n"
  obj << "О Микеланджело Буонаротти.\r\n"
end
```

И еще одно небольшое усовершенствование: мы добавим метод `puts`, который вставит в сообщение символы перехода на новую строку:

```
class Net::NetPrivate::WriteAdapter
  def puts(args)
    args << "\r\n"
    self.write(*args)
  end
end
```

Новый метод позволяет формировать сообщение и так:

```
smtp.ready("t.s.eliot@fake1.com", "reader@fake2.com") do |obj|
  obj.puts "Мы были призваны в глухую глубину,"
end
```

* В тексте приводится отрывок из стихотворения Т.Элиота «Песнь любви Дж. Альфреда Пруффра» (пер. В. Топорова). – *Прим. перев.*

```
obj.puts "В мир дев морских, в волшебную страну,"
obj.puts "Но нас окликнули - и мы пошли ко дну."
end
```

Если всего изложенного вам не хватает, поэкспериментируйте самостоятельно. А если соберетесь написать новый интерфейс к протоколу SMTP, не стесняйтесь.

18.2.5. Взаимодействие с IMAP-сервером

Протокол IMAP нельзя назвать вершиной совершенства, но во многих отношениях он превосходит POP3. Сообщения могут храниться на сервере сколь угодно долго (с индивидуальными пометками «прочитано» и «не прочитано»). Для хранения сообщений можно организовать иерархию папок. Этих возможностей уже достаточно для того, чтобы считать протокол IMAP более развитым, чем POP3.

Для взаимодействия с IMAP-сервером предназначена стандартная библиотека `net/imap`. Естественно, вы должны сначала установить соединение с сервером, а затем идентифицировать себя с помощью имени и пароля:

```
require 'net/imap'

host = "imap.hogwarts.edu"
user, pass = "lupin", "riddikulus"

imap = Net::IMAP.new(host)
begin
  imap.login(user, pass)
  # Или иначе:
  # imap.authenticate("LOGIN", user, pass)
rescue Net::IMAP::NoResponseError
  abort "Не удалось аутентифицировать пользователя #{user}"
end

# Продолжаем работу...

imap.logout # Разорвать соединение.
```

Установив соединение, можно проверить почтовый ящик методом `examine`; по умолчанию почтовый ящик в IMAP называется INBOX. Метод `responses` возвращает информацию из почтового ящика в виде хэша массивов (наиболее интересные данные находятся в последнем элементе массива). Показанный ниже код показывает общее число сообщений в почтовом ящике ("EXISTS") и число непрочитанных сообщений ("RECENT"):

```
imap.examine("INBOX")
total = imap.responses["EXISTS"].last # Всего сообщений.
recent = imap.responses["RECENT"].last # Непрочитанных сообщений.
imap.close # Закрыть почтовый ящик.
```

Отметим, что метод `examine` позволяет только читать содержимое почтового ящика. Если нужно удалить сообщения или произвести какие-то другие изменения, пользуйтесь методом `select`.

Почтовые ящики в протоколе IMAP организованы иерархически, как имена путей в UNIX. Для манипулирования почтовыми ящиками предусмотрены методы `create`, `delete` и `rename`:

```
imap.create("lists")
imap.create("lists/ruby")
imap.create("lists/rails")
imap.create("lists/foobar")

# Уничтожить последний созданный ящик:
imap.delete("lists/foobar")
```

Имеются также методы `list` (получить список всех почтовых ящиков) и `lsub` (получить список «активных» ящиков, на которые вы «подписались»). Метод `status` возвращает информацию о состоянии ящика.

Метод `search` находит сообщения, удовлетворяющие заданному критерию, а метод `fetch` возвращает запрошенное сообщение:

```
msgs = imap.search("TO", "lupin")
msgs.each do |mid|
  env = imap.fetch(mid, "ENVELOPE")[0].attr("ENVELOPE")
  puts "От #{env.from[0].name}      #{env.subject}"
end
```

Команда `fetch` в предыдущем примере выглядит так сложно, потому что возвращает массив хэшей. Сам конверт тоже представляет собой сложную структуру; некоторые методы доступа к нему возвращают составные объекты, другие – просто строки.

В протоколе IMAP есть понятия UID (уникального идентификатора) и порядкового номера сообщения. Обычно методы типа `fetch` обращаются к сообщениям по номерам, но есть и варианты (например, `uid_fetch`) для обращения по UID. У нас нет места объяснять, почему нужны обе системы идентификации, но если вы собираетесь серьезно работать с IMAP, то должны понимать различие между ними (и никогда не путать одну с другой).

Библиотека `net/imap` располагает разнообразными средствами для работы с почтовыми ящиками, сообщениями, вложениями и т. д. Дополнительную информацию поищите в онлайн-овой документации на сайте ruby-doc.org.

18.2.6. Кодирование и декодирование вложений

Для вложения в почтовое сообщение или в сообщение, отправляемое в конференцию, файл обычно кодируется. Как правило, применяется кодировка `base64`, для работы с которой служит метод `pack` с аргументом `m`:

```
bin = File.read("new.gif")
str = [bin].pack("m")      # str закодирована.

orig = str.unpack("m")[0]  # orig == bin
```

Старые почтовые клиенты работали с кодировкой `uuencode` / `uudecode`. В этом случае вложение просто добавляется в конец текста сообщения и ограничивается

строками `begin` и `end`, причем в строке `begin` указываются также разрешения на доступ к файлу (которые можно и проигнорировать) и имя файла. Аргумент `u` метода `pack` позволяет представить строку в кодировке `uuencode`. Пример:

```
# Предположим, что mailtext содержит текст сообщения.
```

```
filename = "new.gif"
bin = File.read(filename)
encoded = [bin].pack("u")

mailtext << "begin 644 #{filename}"
mailtext << encoded
mailtext << "end"
# ...
```

На принимающей стороне мы должны извлечь закодированную информацию и декодировать ее методом `unpack`:

```
# ...
# Предположим, что 'attached' содержит закодированные данные
# (включая строки begin и end).

lines = attached.split("\n")
filename = /begin \d\d\d (.*).scan(lines[0]).first.first
encoded = lines[1..-2].join("\n")
decoded = encoded.unpack("u")      # Все готово к записи в файл.
```

Современные почтовые клиенты работают с почтой в формате MIME; даже текстовая часть сообщения обернута в конверт (хотя клиент удаляет все заголовки, прежде чем показать сообщение пользователю).

Подробное рассмотрение формата MIME заняло бы слишком много места, да и не относится к рассматриваемой теме. Но в следующем простом примере показано, как можно закодировать и отправить сообщение, содержащее текстовую часть и двоичное вложение. Двоичные куски обычно представлены в кодировке `base64`:

```
require 'net/smtp'

def text_plus_attachment(subject,body,filename)
  marker = "MIME_boundary"
  middle = "--#{marker}\n"
  ending = "--#{middle}--\n"
  content = "Content-Type: Multipart/Related; " +
    "boundary=#{marker}; " +
    "typw=text/plain"

  head1 = <<-EOF
MIME-Version: 1.0
#{content}
Subject: #{subject}
EOF
  binary = File.read(filename)
  encoded = [binary].pack("m")      # base64
```

```

head2 = <<EOF
Content-Description: "#{filename}"
Content-Type: image/gif; name="#{filename}"
Content-Transfer-Encoding: Base64
Content-Disposition: attachment; filename="#{filename}"
EOF

# Возвращаем...
head1 + middle + body + middle + head2 + encoded + ending
end

domain = "someserver.com"
smtp    = "smtp.#{domain}"
user, pass = "elgar","enigma"

body = <<EOF
Это мое сообщение. Особо
говорить не о чем. Я вложил
небольшой GIF-файл.

-- Боб
EOF
mailtext = text_plus_attachment("Привет...",body,"new.gif")

Net::SMTP.start(smtp, 25, domain, user, pass, :plain) do |mailer|
  mailer.sendmail(mailtext, 'fromthisguy@wherever.com',
    ['destination@elsewhere.com'])
end

```

18.2.7. Пример: шлюз между почтой и конференциями

В онлайн-новых сообществах общение происходит разными способами. К наиболее распространенным относятся списки рассылки и конференции (новостные группы).

Но не каждый хочет подписываться на список рассылки и ежедневно получать десятки сообщений; кто-то предпочитает время от времени заходить в конференцию и просматривать новые сообщения. С другой стороны, есть люди, которым система Usenet кажется слишком медлительной – они хотели бы видеть сообщение, пока еще электроны не успели остыть.

Таким образом, мы имеем ситуацию, когда в сравнительно небольшом закрытом списке рассылки рассматриваются те же темы, что в немодерируемой конференции, открытой всему миру. В конце концов кому-то пришла в голову мысль организовать зеркало – шлюз между обеими системами.

Подобный шлюз подходит не к любой ситуации, но в случае списка рассылки Ruby он вполне годится. Сообщения из конференции нужно скопировать в список, а сообщения, отправляемые в список рассылки, направить также и в конференцию.

Эта задача была решена Дэйвом Томасом (Dave Thomas) – конечно, на Ruby, – и с его любезного разрешения мы приводим код в листингах 18.6 и 18.7.

Но сначала небольшое вступление. Мы уже немного познакомились с тем, как отправлять и получать электронную почту, но как быть с конференциями Usenet? Доступ к конференциям обеспечивает протокол NNTP (Network News Transfer Protocol – сетевой протокол передачи новостей). Кстати, создал его Ларри Уолл (Larry Wall), который позже подарил нам язык Perl.

В Ruby нет «стандартной» библиотеки для работы с NNTP. Однако один японский программист (известный нам только по псевдониму greentea) написал прекрасную библиотеку для этой цели.

В библиотеке `nntp.rb` определен модуль `NNTP`, содержащий класс `NNTP::IO`. В этом классе имеются, в частности, методы экземпляра `connect`, `get_head`, `get_body` и `post`. Чтобы получить сообщения, необходимо установить соединение с сервером и в цикле вызывать методы `get_head` и `get_body` (мы, правда, немного упрощаем). Чтобы отправить сообщение, нужно сконструировать его заголовки, соединиться с сервером и вызвать метод `post`.

В приведенных ниже программах используется библиотека `smtp`, с которой мы уже познакомились. В оригинальной версии кода производится также протоколирование хода процесса и ошибок, но для простоты мы эти фрагменты опустили.

Файл `params.rb` нужен обоим программам. В нем описаны параметры, управляющие всем процессом зеркалирования: имена серверов, имена пользователей и т.д. Ниже приведен пример, который вы можете изменить самостоятельно. (Все доменные имена, содержащие слово «fake», очевидно, фиктивные.)

Различные параметры, необходимые шлюзу между почтой и конференциями.

```
module Params
  NEWS_SERVER = "usenet.fakel.org"      # Имя новостного сервера.
  NEWSGROUP   = "comp.lang.ruby"       # Зеркалируемая конференция.
  LOOP_FLAG   = "X-rubymirror: yes"    # Чтобы избежать циклов.
  LAST_NEWS_FILE = "/tmp/m2n/last_news" # Номер последнего прочитанного
                                         # сообщения.
  SMTP_SERVER = "localhost"            # Имя хоста для исходящей почты.

  MAIL_SENDER = "myself@fake2.org"     # От чьего имени посылать почту.
  # (Для списков, на которые подписываются, это должно быть имя
  # зарегистрированного участника списка.)

  MAILING_LIST = "list@fake3.org"      # Адрес списка рассылки.
end
```

Модуль `Params` содержит лишь константы, нужные обоим программам. Большая их часть не нуждается в объяснениях, упомянем лишь парочку. Во-первых, константа `LAST_NEWS_FILE` содержит путь к файлу, в котором хранится идентификатор последнего прочитанного из конференции сообщения; эта «информация о состоянии» позволяет избежать дублирования или пропуска сообщений.

Константа `LOOP_FLAG` определяет строку, которой помечаются сообщения, уже прошедшие через шлюз. Тем самым мы препятствуем возникновению бесконечной

рекурсии, а заодно негодование возмущенных обитателей сети, получивших тысячи копий одного и того же сообщения.

Возникает вопрос: «А как вообще почта поступает в программу mail2news?» Ведь она, похоже, читает из стандартного ввода. Автор рекомендует следующую настройку: сначала в файле `.forward` программы `sendmail` вся входящая почта перенаправляется на программу `procmail`. Файл `.procmail` конфигурируется так, чтобы извлекать сообщения, приходящие из списка рассылки, и по конвейеру направлять их программе `mail2news`. Уточнить детали можно в документации, сопровождающей приложение `RubyMirror` (в архиве `RAA`). Если вы работаете не в `UNIX`, то придется изобрести собственную схему конфигурирования.

Ну а все остальное расскажет сам код, приведенный в листингах 18.6 и 18.7.

Листинг 18.6. Перенаправление почты в конференцию

```
# mail2news: Принимает почтовое сообщение и отправляет
# его в конференцию.

require "nntp"
include NNTP

require "params"

# Прочитать сообщение, выделив из него заголовок и тело.
# Пропускаются только определенные заголовки.

HEADERS = %w{From Subject References Message-ID
              Content-Type Content-Transfer-Encoding Date}

allowed_headers = Regexp.new(%{^#{HEADERS.join("|")}}:})

# Прочитать заголовок. Допускаются только некоторые заголовки.
# Добавить строки Newsgroups и X-rubymirror.

head = "Newsgroups: #{Params::NEWSGROUP}\n"
subject = "unknown"
while line = gets
  exit if line =~ /^#{Params::LOOP_FLAG}/o # Такого не должно быть!
  break if line =~ /^s*$/
  next if line =~ /^s/
  next unless line =~ allowed_headers

  # Вырезать префикс [ruby-talk:nnnn] из темы, прежде чем
  # отправлять в конференцию.
  if line =~ /^Subject:s*(.*)/
    subject = $1

    # Следующий код вырезает специальный номер ruby-talk
    # из начала сообщения в списке рассылки, перед тем
    # как отправлять его новостному серверу.
```

```

line.sub!(/\{ruby-talk:(\d+)\}\s*/, '')
subject = "#{\$1} #{(line)}"
head << "X-ruby-talk: \$1\n"
end
head << line
end

head << "#{(Params::LOOP_FLAG)}\n"

body = ""
while line = gets
  body << line
end

msg = head + "\n" + body
msg.gsub!(/\r?\n/, "\r\n")

nntp = NNTPIO.new(Params::NEWS_SERVER)
raise "Failed to connect" unless nntp.connect
nntp.post(msg)

```

Листинг 18.7. Перенаправление конференции в почту

```

##
# Простой сценарий для зеркалирования трафика
# из конференции comp.lang.ruby в список рассылки ruby-talk.
#
# Вызывается периодически (скажем, каждые 20 минут).
# Запрашивает у новостного сервера все сообщения с номером,
# большим номера последнего сообщения, полученного
# в прошлый раз. Если таковые есть, то читает сообщения,
# отправляет их в список рассылки и запоминает номер последнего.

require 'nntp'
require 'net/smtp'
require 'params'

include NNTP

##
# Отправить сообщения в список рассылки. Сообщение должно
# быть отправлено участником списка, хотя в строке From:
# может стоять любой допустимый адрес.
#

def send_mail(head, body)
  smtp = Net::SMTP.new
  smtp.start(Params::SMTP_SERVER)
  smtp.ready(Params::MAIL_SENDER, Params::MAILING_LIST) do |a|

```

```
a.write head
a.write "#{Params::LOOP_FLAG}\r\n"
a.write "\r\n"
a.write body
end
end

##
# Запоминаем идентификатор последнего прочитанного из конференции
# сообщения.

begin
  last_news = File.open(Params::LAST_NEWS_FILE) { |f| f.read } .to_i
rescue
  last_news = nil
end

##
# Соединяемся с новостным сервером и получаем номера сообщений
# из конференции comp.lang.ruby.
#
nntp = NNTP::IO.new(Params::NEWS_SERVER)
raise "Failed to connect" unless nntp.connect
count, first, last = nntp.set_group(Params::NEWSGROUP)

##
# Если номер последнего сообщения не был запомнен раньше,
# сделаем это сейчас.

if not last_news
  last_news = last
end

##
# Перейти к последнему прочитанному ранее сообщению
# и попытаться получить следующие за ним. Это может привести
# к исключению, если сообщения с указанным номером
# не существует, но мы не обращаем на это внимания.

begin
  nntp.set_stat(last_news)
rescue
end

##
# Читаем все имеющиеся сообщения и отправляем каждое
# в список рассылки.

new_last = last_news
```

```
begin
  loop do
    nntp.set_next
    head = ""
    body = ""
    new_last, = nntp.get_head do |line|
      head << line
    end

    # Не посылать сообщения, которые программа mail2news
    # уже отправляла в конференцию ранее (иначе заиклимся).
    next if head =~ %r{^X-rubymirror:}

    nntp.get_body do |line|
      body << line
    end

    send_mail(head, body)
  end
rescue
end

##
# И записать в файл новую отметку.

File.open(Params::LAST_NEWS_FILE, "w") do |f|
  f.puts new_last
end unless new_last == last_news
```

18.2.8. Получение Web-страницы с известным URL

Пусть нам нужно получить HTML-документ из Web. Возможно, вы хотите проверить контрольную сумму и узнать, не изменился ли документ, чтобы послать автоматическое уведомление. А быть может, вы пишете собственный браузер – тогда это первый шаг на пути длиной в тысячу километров.

```
require "net/http"

begin
  h = Net::HTTP.new("www.marsdrive.com", 80) # MarsDrive Consortium
  resp, data = h.get("/index.html", nil)
rescue => err
  puts "Ошибка: #{err}"
  exit
end

puts "Получено #{data.split.size} строк, #{data.size} байтов"
# Обработать...
```

Сначала мы создаем объект класса HTTP, указывая доменное имя и номер порта сервера (обычно используется порт 80). Затем выполняется операция get,

которая возвращает ответ по протоколу HTTP и вместе с ним строку данных. В примере выше мы не проверяем ответ, но если возникла ошибка, то перехватываем ее и выходим.

Если мы благополучно миновали предложение `rescue`, то можем ожидать, что содержимое страницы находится в строке `data`. Мы можем обработать ее как сочтем нужным.

Что может пойти не так, какие ошибки мы перехватываем? Несколько. Может не существовать или быть недоступным сервер с указанным именем; указанный адрес может быть перенаправлен на другую страницу (эту ситуацию мы не обрабатываем); может быть возвращена пресловутая ошибка 404 (указанный документ не найден). Обработку подобных ошибок мы оставляем вам.

Следующий раздел окажется в этом смысле полезным. В нем мы представим несколько более простой способ решения данной задачи.

18.2.9. Библиотека Open-URI

Библиотеку Open-URI написал Танака Акира (Tanaka Akira). Ее цель – унифицировать работу с сетевыми ресурсами из программы, предоставив интуитивно очевидный и простой интерфейс.

По существу она является оберткой вокруг библиотек `net/http`, `net/https` и `net/ftp` и предоставляет метод `open`, которому можно передать произвольный URI. Пример из предыдущего раздела можно было бы переписать следующим образом:

```
require 'open-uri'

data = nil
open("http://www.marsdrive.com/") {|f| data = f.read }

puts "Получено #{data.split.size} строк, #{data.size} байтов"
```

Объект, возвращаемый методом `open` (`f` в примере выше), – не просто файл. У него есть также методы из модуля `OpenURI::Meta`, поэтому мы можем получить метаданные:

```
uri = f.base_uri          # Объект URI с собственными методами доступа.
ct  = f.content_type      # "text/html"
cs  = f.charset           # "utf-8"
ce  = f.content_encoding  # []
```

Библиотека позволяет задать и дополнительные заголовочные поля, передавая методу `open` хэш. Она также способна работать через прокси-серверы и обладает рядом других полезных функций. В некоторых случаях этой библиотеки недостаточно (например, если необходимо разбирать заголовки HTTP, буферизовать очень большой скачиваемый файл, отправлять куки и т.д.). Дополнительную информацию можно найти в онлайн-овой документации на сайте <http://ruby-doc.org>.

18.3. Заключение

Эта глава представляет собой введение в сетевое программирование на низком уровне. В частности, приведены простые примеры серверов и клиентов. Мы видели, как написать клиент для существующего сервера, созданного не нами.

Мы рассмотрели также протоколы более высокого уровня, например POP и IMAP для получения почты. Аналогично мы говорили о протоколе отправки почты SMTP. Попутно был продемонстрирован способ кодирования и декодирования вложений в почтовые сообщения. В контексте разработки шлюза между списком рассылки и конференциями мы упомянули о протоколе NNTP.

Настала пора тщательно изучить более узкий вопрос, относящийся к данной теме. В настоящее время один из самых важных видов сетевого программирования – это разработка для Web, которой и посвящена следующая глава.

Глава 19. Ruby и Web-приложения

.....
Как ловко мы сплетаем сеть...
Сэр Вальтер Скотт, «Мармион»

Ruby – универсальный язык, его ни в коей мере нельзя считать исключительно «языком Web». Но, несмотря на это, одно из наиболее типичных его применений – создание приложений (да и вообще инструментов в широком смысле) для Web.

Существует множество способов разрабатывать приложения для Web на Ruby – от сравнительно небольших и низкоуровневых библиотек до каркасов, которые диктуют стиль кодирования и мышления.

Начнем с низкого уровня и рассмотрим библиотеку `cgi.rb`, входящую в стандартный дистрибутив Ruby.

19.1. Программирование CGI на Ruby

Всякий, кто знаком с программированием для Web, хотя бы раз встречал аббревиатуру *CGI* (Common Gateway Interface – общий шлюзовой интерфейс). Спецификация CGI появилась на заре развития Web с целью обогатить взаимодействие между пользователем и Web-сервером. С тех пор были изобретены бесчисленные альтернативные технологии, но CGI все еще живет и прекрасно себя чувствует. Своим успехом и долговечностью технология CGI обязана простоте, благодаря которой программы, удовлетворяющие этой спецификации, можно без труда писать на любом языке. Спецификация определяет, как процесс Web-сервера должен передавать данные своим потомкам. По большей части взаимодействие сводится к стандартным переменным окружения и потокам ввода/вывода.

Программирование с применением CGI, да и вообще для протокола HTTP, должно учитывать отсутствие «состояния» в механизме запрос-ответ. В общем случае клиент (обычно браузер) для каждого запроса создает новое соединение и посылает по нему единственную команду HTTP. Чаще всего используются команды GET и POST (к смыслу этих слов мы вернемся чуть позже). Сервер посылает ответ, после чего закрывает свой конец соединения.

В следующем примере, который лишь немногим сложнее пресловутой программы «Hello, world», показано, как выполняются ввод и вывод по спецификации CGI.

```
def parse_query_string
  inputs = Hash.new
  raw = ENV['QUERY_STRING']
```

```
raw.split("&").each do |pair|
  name,value = pair.split("=")
  inputs[name] = value
end
inputs
end

inputs = parse_query_string
print "Content-type: text/html\n\n"
print "<HTML><BODY>"
print "<B><I>Hello</I>, #{inputs['name']}!</B>"
print "</BODY></HTML>"
```

Так, обращение с помощью этой программы к URL <http://mywebserver/cgi-bin/hello.cgi?name=Dali> приведет к отправке браузеру сообщения «Hello, Dali!».

Мы уже упомянули, что есть два основных способа обратиться к UTL: методы GET и POST, определенные в протоколе HTTP. Для краткости мы предложим простые объяснения, а не строгие определения. Метод GET обычно вызывается, когда вы щелкаете по ссылке или указываете URL непосредственно (как в предыдущем примере). Параметры передаются в строке запроса, которую CGI-программы видят как значение переменной окружения QUERY_STRING. Метод POST обычно применяется для отправки HTML-форм. Параметры включаются в тело сообщения и в URL не видны. CGI-программе они доставляются через стандартный поток ввода.

Предыдущий пример был предельно простым, но в менее тривиальных случаях программа быстро становится запутанной. Приходится иметь дело с различными методами HTTP, загрузкой файлов, куками, сессиями «с состоянием» и другими сложностями, которые лучше оставить библиотеке общего назначения, поддерживающей спецификацию CGI. К счастью, в Ruby имеется обширный набор классов для автоматизации рутинной работы.

Есть еще немало библиотек и инструментов, стремящихся упростить разработку в рамках CGI. К числу лучших следует отнести библиотеку *ruby-web* (прежнее название *Narf*) Патрика Мэя (*Patrick May*). Если вам нужно работать на низком уровне, но стандартная библиотека почему-либо не устраивает, попробуйте эту (<http://ruby-web.org>).

Если необходимо решение на основе шаблонов, возможно, подойдет библиотека *Amrita* (<http://amrita.sourceforge.jp>). Обратите также внимание на *Cerise* – сервер приложений на базе *Amrita* (<http://cerise.rubyforge.org>).

Наверное, существуют и другие библиотеки. Если вы не нашли того, что искали, среди упомянутых здесь продуктов, обратитесь к поисковой машине или задайте вопрос в конференции.

19.1.1. Введение в библиотеку *cgi.rb*

Библиотека *cgi.rb* включена в стандартный дистрибутив Ruby. Ее функциональность в основном реализована в центральном классе *CGI*. Первое, что нужно сделать при работе с данной библиотекой, – создать экземпляр этого класса.

```
require "cgi"
cgi = CGI.new("html4")
```

Инициализатор класса CGI принимает единственный параметр – уровень спецификации языка HTML, который должен поддерживаться методами генерации разметки, входящими в пакет CGI. Эти методы избавляют программиста от ручного написания длинных фрагментов экранированной разметки в достаточно простом коде на Ruby:

```
cgi.out do
  cgi.html do
    cgi.body do
      cgi.h1 { "Hello Again, " } +
      cgi.b { cgi['name']}
    end
  end
end
```

Здесь мы почти точно воспроизвели функциональность предыдущей программы, воспользовавшись библиотекой CGI. Как видите, класс CGI берет на себя заботу о разборе и сохранении параметров в структуре, напоминающей хэш. Поэтому, если указать URL `some_program.cgi?age=4`, значение параметра `age` можно получить как `cgi['age']`.

Отметим, что в данном примере нам понадобилось только значение, возвращаемое блоком; HTML-разметка строится постепенно и сохраняется, а не выводится сразу же. Это означает, что конкатенировать строки абсолютно необходимо, иначе мы увидели бы только последнюю вычисленную строку.

В классе CGI есть также удобные механизмы для кодирования строк URL и экранирования специальных символов в HTML и XML-коде. *URL-кодирование* – это представление «небезопасных» символов с помощью других символов, допустимых в URL. В результате получаются странные строки, изобилующие знаками %, которые вы часто встречали в Web. На самом деле это просто шестнадцатеричные ASCII-коды символов с предшествующим знаком %.

```
require "cgi"
s = "This| is^(aT$test"
s2 = CGI.escape(s)           # "This%7C+is%5E%28aT%24test"
puts CGI.unescape(s2)       # Печатается "This| is^(aT$test"
```

Аналогично класс CGI позволяет экранировать части HTML или XML-текста, которые должны отображаться в браузере буквально. Например, без специальных мер строка `<some_stuff>` не будет показана в браузере именно в таком виде. Если необходимо показать саму разметку, например, в качестве примера в руководстве по HTML, то можно воспользоваться классом CGI для преобразования специальных символов:

```
require "cgi"
some_text = "<B>This is how you make text bold</B>"
translated = CGI.escapeHTML(some_text)
# "<B>This is how you make text bold</B>"
```

```
puts CGI.unescapeHTML(translated)
# Печатается "<B>This is how you make text bold</B>"
```

19.1.2. Вывод и обработка форм

Наиболее распространенный способ взаимодействия с CGI-программой – форма. *HTML-формы* создаются с помощью специальных тегов, которые браузер представляет в виде элементов ввода данных. Подробное их обсуждение выходит за рамки данной книги, но существует немало книг и онлайн-руководств по этому предмету.

В классе CGI имеются методы для генерирования всех элементов HTML-форм. В примере ниже показано, как можно вывести и обработать форму.

```
require "cgi"

def reverse_ramblings(ramblings)
  if ramblings[0] == nil then return " " end
  chunks = ramblings[0].split(/\s+/)
  chunks.reverse.join(" ")
end

cgi = CGI.new("html4")
cgi.out do
  cgi.html do
    cgi.body do
      cgi.h1 { "sdrawkcaB txeT" } +
      cgi.b { reverse_ramblings(cgi['ramblings']) } +
      cgi.form("action" => "/cgi-bin/rb/form.cgi") do
        cgi.textarea("ramblings") { cgi['ramblings'] } + cgi.submit
      end
    end
  end
end
```

Здесь отображается многострочное поле ввода, текст в котором разбивается на слова и выводится в обратном порядке. Так, если набрать фразу «This is a test», то после обработки вы увидите «test a is This». Метод `form` класса CGI принимает параметр `method` – один из методов отправки формы, определенных в протоколе HTTP (GET, POST и т.д.). По умолчанию предполагается значение POST.

В этом примере демонстрируется лишь малая толика элементов форм, которые могут встречаться на HTML-странице. Полный перечень вы найдете в любом руководстве по HTML.

19.1.3. Куки

Мы уже упоминали, что HTTP – протокол без состояния. Это означает, что после того как сервер закончил обрабатывать запрос, он не может сказать, пришел ли следующий запрос от того же или какого-либо другого браузера. Тут-то и приходят на помощь *куки* (cookies) – способ, быть может, несколько грубоватый, сохранить состояние между последовательными запросами от одного и того же браузера.

Механизм куков работает следующим образом. Сервер посылает браузеру команду (в виде HTTP-заголовка) с просьбой сохранить пару имя–значение. Данные могут храниться в памяти или на диске. При каждом последующем запросе к любому серверу из домена, указанного в куке, браузер пошлет сохраненные данные в HTTP-заголовке.

Можно, конечно, читать и формировать куки вручную, но, как вы, наверное, догадались, в этом нет необходимости, поскольку библиотека CGI предоставляет класс `Cookie`, который инкапсулирует все технические детали.

```
require "cgi"
lastacc = CGI::Cookie.new("kabhi",
                           "lastaccess=#{Time.now.to_s}")

cgi = CGI.new("html3")
if cgi.cookies.size < 1
  cgi.out("cookie" => lastacc) do
    "Hit refresh for a lovely cookie"
  end
else
  cgi.out("cookie" => lastacc) do
    cgi.html do
      "Hi, you were last here at: "+
      "#{cgi.cookies['kabhi'].join.split('=')[1]}"
    end
  end
end
```

Здесь создается кук "kabhi", ключ которого "lastaccess" содержит текущее время. Если у браузера уже был такой кук, то выводится его значение. Куки хранятся в хэше, который является переменной экземпляра в классе `CGI`. Каждый кук может содержать несколько пар ключ–значение, поэтому при доступе к куку по имени вы получаете массив.

19.1.4. Сеансы пользователей

Куки – это хорошо, если вам нужно сохранить простые данные и вы не прочь возложить на браузер заботу об их хранении. Но часто предъявляются более жесткие требования. Что если нужно сохранить много данных и вы не хотите гонять их «взад-вперед» при каждом запросе? К тому же данные могут быть секретными, так что доверять их хранение браузеру нежелательно.

В таких случаях можно воспользоваться классом `CGI::Session`. Он аналогичен классу `CGI::Cookie` в том смысле, что значения хранятся в структуре, напоминающей хэш.

```
require "cgi"
require "cgi/session"

cgi = CGI.new("html4")

sess = CGI::Session.new( cgi, "session_key" => "a_test",
                          "prefix" => "rubysess.")
```

```

lastaccess = sess["lastaccess"].to_s
sess["lastaccess"] = Time.now
if cgi['bgcolor'][0] =~ /[a-z]/
  sess["bgcolor"] = cgi['bgcolor']
end

cgi.out do
  cgi.html do
    cgi.body ("bgcolor" => sess["bgcolor"]) do
      "Фон этой страницы" +
      "изменяется в зависимости от значения 'bgcolor'," +
      "хранящегося в сеансе каждого пользователя." +
      "Время последнего доступа: #{lastaccess}"
    end
  end
end
end

```

Если обратиться к URL `/thatscript.cgi?bgcolor=red`, то фоновый цвет страницы у данного пользователя станет красным и останется таким до тех пор, пока он не обратится к такому же URL, но с другим значением параметра "bgcolor". При создании объекта `CGI::Session` указываются объект `CGI` и набор параметров в хэше. Необязательный параметр `session_key` определяет ключ, с помощью которого браузер будет идентифицировать себя при каждом запросе. Сеансовые данные хранятся во временном файле, своем для каждого сеанса, а параметр `prefix` задает строку, с которой должно начинаться имя файла, чтобы проще было опознать все такие файлы в файловой системе сервера.

Классу `CGI::Session` пока недостает многих возможностей, в частности умения хранить объекты, отличные от `String`, организации общего хранилища сеансовых данных для нескольких серверов и пр. К счастью, уже готов подключаемый механизм `database_manager`, так что некоторые из этих функций нетрудно добавить. Если вы придумаете что-нибудь интересное в отношении класса `CGI::Session`, не забудьте поделиться с сообществом.

19.2. FastCGI

Чаще всего CGI критикуют за то, что при каждом запросе создается новый процесс, и это заметно снижает производительность. Невозможность сохранять в памяти объекты между последовательными запросами также не украшает дизайн системы. Для разрешения этих проблем была создана технология FastCGI.

По сути дела, *FastCGI* – это определение и программная реализация протокола. Обычно она реализуется в надстройке над Web-сервером, например модуля в случае сервера Apache. FastCGI позволяет работающему внутри процесса компоненту перехватывать HTTP-запросы и направлять их через сокет другому процессу, работающему в течение длительного времени. По сравнению с традиционным порождением новых процессов это существенно ускоряет работу. Кроме того, программист получает возможность оставить данные в памяти и найти их там при обработке следующего запроса.

Серверы, адаптированные для работы с FastCGI, реализованы на многих языках, в том числе на Ruby. Эли Грин (Eli Green) написал целиком на Ruby модуль (он есть в архиве RAA), который реализует протокол FastCGI и упрощает разработку FastCGI-программ.

Не вдаваясь в детали реализации, мы представили в листинге 19.1 пример приложения. Как видите, он повторяет функциональность предыдущего примера.

Листинг 19.1. Пример FastCGI

```
require "fastcgi"
require "cgi"

last_time = ""

def get_ramblings(instream)
  # Не слишком красивый способ извлечь значение из первой пары
  # имя-значение. CGI сделал бы это за нас.
  data = ""
  if instream != nil
    data = instream.split("&")[0].split("=")[1] || ""
  end
  return CGI.unescape(data)
end

def reverse_ramblings(ramblings)
  if ramblings == nil then return "" end
  chunks = ramblings.split(/\s+/)
  chunks.reverse.join(" ")
end

server = FastCGI::TCP.new('localhost', 9000)
begin
  server.each_request do |request|
    stuff = request.in.read
    out = request.out

    out << "Content-type: text/html\r\n\r\n"
    out << <<-EOF
    <html>
    <head><title>Отражатель текста</title></head>
    <h1>sdrawkcaB txeT</h1>
    <i>Вы перед этим сказали: #{last_time}</i><BR>
    <b>#{reverse_ramblings(get_ramblings(stuff))}</b>
    <form method="POST" action="/fast/serv.rb">
    <textarea name="ramblings">
    </textarea>
    <input type="submit" name="submit">
    </form>
    </body></html>
    EOF
```

```
last_time = get_ramblings(stuff)
request.finish
end
ensure
  server.close
end
```

Сразу же бросается в глаза (если вы читали предыдущий раздел) то, что в FastCGI приходится вручную делать кое-какие вещи, которые были не нужны при работе с библиотекой CGI. Во-первых, «зашивание» в код экранированного HTML-кода. Во-вторых, метод `get_ramblings`, который сам разбирает входные данные и возвращает нужное значение. Кстати, этот код будет работать только для POST-запросов – еще одно удобство, утраченное с отказом от библиотеки CGI.

Но вместе с тем у FastCGI есть и достоинства. Мы не проводили замеров, но – как следует из самого названия – FastCGI быстрее CGI. Вместо накладных расходов на создание нового процесса мы просто открываем соединение с портом 9000 на локальной машине (`FastCGI::TCP.new('localhost', 9000)`). Кроме того, в переменной `last_time` хранится часть состояния сеанса – вещь, невозможная в традиционной технологии CGI.

Отметим также, что эти библиотеки можно до определенного предела «сшивать». Вспомогательные функции из `cgi.rb`, например метод `CGI.escapeHTML`, можно использовать автономно (не делая библиотеку основой приложения). Тогда предыдущий пример оказался бы несколько проще для восприятия.

19.3. Ruby on Rails

В сообществе пользователей Ruby одним из самых широко известных каркасов для Web является *Ruby on Rails* (или просто *Rails*). Его автор Дэвид Хайнемайер Хансон (David Heinemeier Hansson).

В каркасе Rails широко используются динамические свойства Ruby. У него также есть собственная философия, направленная на быструю разработку Web-приложений.

Rails хорошо известен и прекрасно документирован. В этой книге мы рассмотрим его лишь поверхностно.

19.3.1. Принципы и техника

Каркас Rails построен на основе паттерна Модель–Вид–Контроллер (Model–View–Controller – MVC). Каждое приложение естественно разбивается на модели (моделирующие предметную область), виды (с помощью которых информация представляется пользователю и организуется возможность взаимодействия) и контроллеры (играющие роль арбитров между моделями и видами).

В основу поведения Rails как каркаса положены определенные принципы. Один из них – «принцип минимизации кода»: не пишите код для связывания одного с другим, если такое связывание можно организовать автоматически.

С ним также связан принцип «примата соглашений над конфигурацией». Придерживаясь ряда заранее оговоренных стилей кодирования и именования, можно

обойтись почти без конфигурирования (и приблизиться к идеальной среде с «нулевым конфигурированием»).

Rails прекрасно справляется с автоматизацией не слишком сложных задач. Если это имеет смысл, он сам генерирует код, избавляя программиста от необходимости писать его вручную.

Web-приложения часто хранят данные в базе, и Rails обеспечивает бесшовную интеграцию с базой данных. У Web-каркасов наблюдается тенденция проявлять «склонность» к какому-то конкретному объектно-реляционному отображению (object-relational mapper, ORM), и Rails – не исключение. Стандартным для Rails является отображение ActiveRecord, которое мы рассматривали в главе 10.

Базы данных описываются в файле `config/database.yml` – одном из немногих необходимых конфигурационных файлов (конечно же, в формате YAML). В нем перечислены три разных базы данных: для разработки, для тестирования и для промышленной эксплуатации. На первый взгляд, это перебор, но в действительности такая схема оказывается очень удобной.

Rails генерирует для вас пустые модели и контроллеры. В ходе редактирования моделей вы определяете связи между таблицами базы данных с помощью таких методов, как `has_many` и `belongs_to` (на самом деле их гораздо больше). Поскольку между моделями и таблицами есть соответствие, то написанный вами код заодно определяет и связи между самими моделями. Для контроля данных служат такие методы, как `validates_presence_of` (проверяет, что данные присутствуют) и `validates_uniqueness_of` (проверяет, что данные уникальны).

В результате создания приложения Rails командой вида `rails appname` вы получаете каталог `appname` с такой структурой:

```
app
  controllers
  helpers
  models
  views
config
db
doc
lib
log
public
script
test
vendor
```

Большая часть кода находится в каталоге `app`. Как видите, сама его структура следует паттерну MVC.

Схемы баз данных находятся в каталоге `db`. Инкрементные файлы миграции тоже попадут сюда.

В Rails есть концепция «обстраивания» (scaffolding), которая очень упрощает жизнь. Если ввести команду `script/generate scaffold Product` (`Product` – имя мо-

дели), то для таблицы `Products` (обратите внимание на множественное число) будет сгенерирована функциональность «создать–обновить–удалить».

Можно обстроиться и не генерируя никакой код, достаточно вызвать внутри контроллера `Product` метод `scaffold`:

```
class ProductController < ActiveRecord::Base
  scaffold :product
end
```

Здесь мы достигаем той же цели, но не записываем никакой код на диск. Оба способа допустимы. Конечно, в результате обстраивания создаются страницы ввода/обновления, которые вполне функциональны, но не слишком красивы; почти всегда вы захотите заменить их чем-то более симпатичным. Тем не менее такая техника взаимодействия с базой данных полезна, особенно на этапе разработки.

В старых версиях Rails расхождение между `ActiveRecord` и базой данных было более существенным. Недавно появившаяся концепция миграции делает управление базой данных проще. То же касается и уже существующих в базе данных таблиц, работать с которыми было трудно; сейчас можно создать файл `schema.rb`, в котором будет перечислены все существующие таблицы (см. также `rake tasks db:schema:load` и `db:schema:dump`).

19.3.2. Тестирование и отладка приложений Rails

В Rails встроена серьезная поддержка тестирования. Обратите внимание на каталог `test`, который создается для каждого нового приложения. Он заполняется по мере развития приложения; вы можете (и должны) добавлять тесты по ходу создания новой функциональности.

По принятому в Rails соглашению, автономные тесты служат для тестирования моделей, а функциональные – для тестирования контроллеров. Этим объясняется наличие подкаталогов `unit` и `functional` в каталоге `test`. (Вне контекста Ruby on Rails эти термины используются несколько иначе.)

Для тестирования в Rails большое значение имеет понятие стенда. *Стенд* (*fixture*) – это моментальный снимок начального содержимого модели, иными словами, набор фиктивных данных для тестирования. Все данные такого рода хранятся в формате YAML в файлах, расположенных в каталоге `test/fixtures`.

Каталог `test/mocks` предназначен для хранения кода, играющего роль объекта-заглушки. *Заглушки* служат для имитации еще не реализованного сервиса или класса. Считайте это подобием голливудской декорации, которая только внешне похожа на реальность. Классический пример – шлюз в систему обработки кредитных карт; с помощью заглушек мы можем протестировать взаимодействие с ним, не имея настоящего шлюза.

Следует также знать о понятии консоли в Rails. Запуск сценария `script/console` открывает сеанс, похожий на `irb`, в котором доступен код модели. В нем вы можете выполнять запросы с помощью `ActiveRecord` и другие подобные операции.

Еще полезнее прерыватель (*breakpointer*). Расположите в любом месте своей программы вызов метода `breakpoint` и запустите утилиту `script/breakpointer`. Вы

окажетесь в сеансе `irb` в контексте установленной точки прерывания и сможете просматривать и изменять значения переменных экземпляра и т. п.

В последних версиях Rails появилась также поддержка для тестирования со-пряжений. Для этого применяется специализированный язык DSL, описывающий поток управления в Web-приложении на верхнем уровне. Он ориентирован на заказчиков, не имеющих технической подготовки, но пригодится и тем пользователям, которые хорошо знакомы с внутренним устройством приложения.

19.3.3. Базовые расширения

Одно из достоинств Rails – наличие широкого набора вспомогательных методов, определенных в модуле `ActiveSupport::CoreExtensions`. Они «безвредны» и после добавления в основные классы становятся доступны всему приложению.

Некоторые из этих методов имеют отношение к работе с датами и временем. Поскольку внутри системы время хранится в секундах, то имеют смысл методы `minutes` и `hours`:

```
elapsed = 3.days + 4.hours + 17.minutes
later = Time.now + elapsed
```

Можно делать и такие вещи:

```
time = 3.minutes.from_now      # То же, что Time.now + 3.minutes.
t2 = 5.days.from_now
```

Операции со временем представлены особенно полно. С помощью методов `midnight`, `next_month` и `beginning_of_week` можно точно и кратко задавать моменты времени.

Один из самых известных трюков в этом модуле – метод `Symbol#to_proc`. Он позволяет передавать символ, представляющий имя метода, вместо блока. Например, следующие два предложения эквивалентны:

```
arr = array.map { |x| x.upcase }
arr = array.map(&:upcase)
```

Помимо упомянутых здесь методов, есть еще десятки других. Одни занимаются преобразованием различных единиц измерения времени, другие – переводом из YAML или XML, третьи – проверкой правописания и расстановкой знаков препинания в строках и т. д.

19.3.4. Дополнительные инструменты и библиотеки

Неизбежным было появление внешних инструментов для работы с Rails. Например, в редактор `TextMate` встроена неплохая поддержка (синтаксическая подсветка, автоматическое дописывание кода и т. д.). Этот редактор, наверное, самый популярный среди программистов на Ruby на платформе OS X.

Многообещающим выглядит проект `InstantRails` (<http://instantrails.rubyforge.org>). Это единый пакет, содержащий Ruby, Rails, MySQL и Apache, причем все продукты уже сконфигурированы и готовы к работе. Первая версия работает только в Windows, но планируется перенос на другие платформы.

На платформе OS/X есть эквивалентный проект `Locomotive`. Это достаточно зрелая и хорошо работающая среда для развертывания Rails «одним щелчком».

Если вы поклонник проекта Eclipse, то должны знать о RadRails – интегрированной среде разработки для Rails, надстроенной над Eclipse. На сайте проекта (<http://radrails.org>) написано, что «продукт поддерживает управление версиями, отладку, серверы WEBrick, мастер генерации кода, синтаксическую подсветку, инструменты для работы с данными и многое другое». Он должен работать на всех платформах, где работает сам Eclipse.

Важно также понимать, что такое подключаемые к Rails модули (plugins). Это небольшие автономные программы, которые модифицируют поведение ActiveRecord или Rails. Их несложно писать и развертывать.

Для установки подключаемого модуля достаточно раскрыть архив и скопировать его в каталог `vendor/plugins`. Примером может служить хорошо известный модуль аннотирования (Annotate models), написанный Дэйвом Томасом. Он добавляет в начало каждого исходного файла модели ActiveRecord комментарии, содержащие краткую аннотацию текущей схемы (эта функциональность становится доступной с помощью задачи rake). Существуют сотни небольших подключаемых модулей, занимающихся аутентификацией, генерацией GUID, интернационализацией, поддержкой CSS и т. д.

Невозможно описать Rails на нескольких страницах – этой теме посвящены целые книги, причем самые первые к лету 2006 года уже устарели. Если вы хотите углубить свои познания в этой области, заходите на сайт <http://rubyonrails.org> – основное место встреч сообщества пользователей Rails.

19.4. Разработка Web-приложений с помощью Nitro

Nitro – еще один комплект инструментов для разработки Web-приложений. Хотя в сочетании с библиотекой Og для объектно-реляционного отображения Nitro хорошо приспособлен к созданию традиционных приложений в духе MVC, задуман он для поддержки различных архитектур.

Установить Nitro проще всего с помощью системы RubyGems. Gem-пакет зависит от нескольких внешних библиотек, которые тоже придется установить (ог, redcloth и ряд других).

```
gem install nitro --include-dependencies
```

Во время работы над книгой последней была версия Nitro 0.31.0. Но, конечно, API и библиотеки постоянно изменяются. Кроме того, имейте в виду, что приведенный ниже обзор Nitro далеко не полон.

19.4.1. Создание простого приложения Nitro

Nitro часто используется в сочетании с Og – библиотекой ORM, обеспечивающей устойчивость объектов Ruby. Но наличие Og необязательно; Nitro, в частности, хорош тем, что не нужно заранее решать, понадобится ли приложению база данных или иная форма обеспечения устойчивости. Если со временем проект изменится и такая нужда возникнет, то для превращения объектов Ruby в устойчивые достаточно будет добавить в модели несколько строк кода. Есть планы модифицировать семейство методов `attr`, так что в будущем задача может стать еще проще.

Библиотека Og рассматривалась в разделе 10.4.7. В примерах работы с Nitro мы почти не будем пользоваться ей.

Хотя у Nitro очень много возможностей, для создания простого приложения нужно немногим больше, чем для создания обычного статического сайта.

Сначала создадим каталог приложения:

```
/home/jbritt/demo
```

Затем добавим в него папку public, содержащую файл index.html:

```
/home/jbritt/demo/public/index.html
```

Для начала сделаем этот файл совсем простым:

```
<html>
  <head>
    <title>Nitro!</title>
  </head>
  <body>
    <h1>The Ruby Way</h1>
    <h2>Hal Fulton</h2>
  </body>
</html>
```

Теперь в корневом каталоге приложения создадим файл run.rb:

```
require 'nitro'
Nitro.run
```

Чтобы увидеть новое приложение в действии, запустите файл run.rb (из каталога demo/). Потом откройте браузер и введите URL <http://127.0.0.1:9999> (здесь 9999 – порт Nitro по умолчанию).

Если все пройдет удачно, то появится приведенная выше простая страница. Принимайте поздравления – вы создали первое приложения для Nitro! Разумеется, Nitro позволяет делать гораздо больше, поэтому посмотрим, как это приложение можно расширить.

Прежде всего, интерес представляет файл run.rb. В зависимости от того, как вы развернули свою программу, его запуском может заниматься диспетчерский сценарий в каталоге public. Но для демонстрации и тестирования вы можете запустить его вручную и воспользоваться встроенным диспетчером WEBrick. Впрочем, для промышленной эксплуатации имеет смысл изучить предоставляемую Nitro поддержку Mongrel, CGI или FastCGI.

Nitro поддерживает много разных архитектур и паттернов, а приложение обычно строится на базе паттерна Модель–Вид–Контроллер (model–view–controller, MVC). Окончательный выбор всегда остается за вами, а Nitro облегчает переход от простых сайтов, предназначенных только для просмотра, к полноценным приложениям с поддержкой базы данных.

По умолчанию при обработке запроса Nitro сначала ищет файл в папке public. Предполагается, что если имя страницы явно не указано, речь идет о файле index.html. В этом отношении Nitro ведет себя как любая статическая система. При желании мы можем поместить в папку public дополнительные статические HTML-страницы, а в ее подпапках хранить изображения и CSS-файлы.

Но интереснее то, что происходит, когда Nitro не может найти прямого соответствия запрошенной странице. Давайте изменим расширение файла `index.html` на `.xhtml`:

```
public/index.xhtmll
```

Перезапустите сценарий `run.rb`. Снова введите URL `http://127.0.0.1:9999`, вы должны увидеть ту же самую страницу. Не найдя файла `index.html`, Nitro ищет файл `index.xhtmll` и загружает его. По умолчанию `xhtml` – расширение, применяемое в Nitro для динамического контента. В общем случае, получив запрос, Nitro сначала ищет файл с расширением `html`, а потом – `xhtml`.

В XHTML-файлах могут быть подставляемые переменные и программная логика. Измените файл `index.xhtmll` следующим образом:

```
<html>
  <head>
    <title>Nitro!</title>
  </head>
  <body>
    <h1>The Ruby Way</h1>
    <h2>Hal Fulton</h2>
    <p>Page last updated:  #{Time.now}</p>
  </body>
</html>
```

После перезагрузки страницы вы увидите текущие дату и время. Nitro также поддерживает синтаксис команд обработки, принятый в XML:

```
<?r curr_date = Time.new.strftime( "%a, %b %d, %Y" ) ?>
<html>
  <head>
    <title>Nitro!</title>
  </head>
  <body>
    <h1>The Ruby Way</h1>
    <h2>Hal Fulton</h2>
    <p>Page last updated:  #{curr_date}</p>
  </body>
</html>
```

Отметим, что из наличия синтаксиса `<?r ... ?>` не следует, что весь шаблон должен подчиняться правилам XML. Nitro располагает средствами для обработки шаблонов как XML-документов, а такой синтаксис позволяет иметь шаблоны, которые одновременно являются корректными XML-документами.

19.4.2. Nitro и паттерн MVC

Вставка кода непосредственно в шаблон удобна для экспериментов и перехода со статических страниц на динамические. Но со временем обнаруживается, что тестировать и сопровождать такое приложение становится все сложнее. Паттерн Модель–Вид–Контроллер позволяет упростить жизнь за счет переноса кода в классы Ruby.

Начнем с создания класса контроллера `main.rb`. Если в каталоге приложения есть папка `src`, Nitro добавит ее в список путей, по которым ищутся страницы. Следуя принятым в Nitro соглашениям, создадим файл `<app_root>/src/controller/book.rb`:

```
class BookController
  def index
    @author = "Hal Fulton"
    @title = "The Ruby Way"
    @last_update = Time.new
  end
end
```

Изменим файл `index.html`, включив в него ссылки на эти переменные:

```
<html>
  <head>
    <title>Nitro!</title>
  </head>
  <body>
    <h1>#{@title}</h1>
    <h2>#{@author}</h2>
    <p>Page last updated:  #{@last_update}</p>
  </body>
</html>
```

Придется также немного подправить файл `run.rb`:

```
require 'nitro'

require 'controller/book'
Nitro.run(BookController)
```

Перезапустите сервер WEBrick и перезагрузите страницу, чтобы посмотреть на результат.

Отметим несколько вещей. Файлы шаблонов могут остаться в папке `public`; класс контроллера не обязан расширять какой-то специальный базовый класс; класс, переданный методу `Nitro.run`, автоматически сопоставляется с корневым URL приложения. Все эти аспекты конфигурируемы.

По умолчанию Nitro ищет шаблоны в каталогах `template` и `public`. Если вы не хотите помещать шаблоны в каталог `public` (скажем, потому, что предпочитаете зарезервировать его только для статических HTML-файлов), то можете создать каталог `template` и хранить их там. Предполагается, что пути к шаблонам повторяют пути, заданные в URL, относительно корневой папки шаблонов. Наш файл `index.html` мог бы храниться как `public/index.html` или `template/index.html`. Использование каталога `public` упрощает переход от статического сайта к динамическому, но лучше организовать раздельное хранение статических файлов и шаблонов.

Классы контроллеров могут отображаться на пути URL с помощью файла `Server.map`. Например, наше демонстрационное приложение может содержать статическую главную страницу со списком литературы (в котором на данный момент

присутствует всего одна книга), находящуюся совсем по другому пути. Для этого нужно следующим образом изменить файл `run.rb`:

```
require 'nitro'
require 'controller/book'
```

```
Nitro::Server.map = { '/books' => BookController }
Nitro.run()
```

Соответствующий шаблон нужно будет переместить в другое место, которое соответствовало бы новому пути (`template/books/index.html`)

Перезапустите сервер и укажите новый URL:

```
http://127.0.0.1:9999/books
```

Сейчас самое время создать настоящую страницу сайта, но ради экономии места подавим это желание. Интереснее добавить механизм для запроса данных о нескольких книгах (сколь бы замечательна ни была та, что вы держите в руках). Добавим файл `src/model/book.rb`, содержащий описание модели `Book`:

```
class Book
  @@items = {}
  attr_accessor :title
  attr_accessor :author
  attr_accessor :update_time

  def initialize( values = {} )
    @title, @author = values[:title], values[:author]
    @update_time = Time.now
  end

  def save
    @@items[author] = self
  end

  def self.find(author)
    @@items[author]
  end
end
```

Теперь контроллер может запрашивать данные у класса `Book`. Изменим метод `index`:

```
def index
  book = Book.find_by_author("Mark Twain")
  @author = book.author
  @title = book.title
  @last_update = book.update_time
end
```

А в файл `run.rb` вставим ссылку на модель `Book` и загрузим какие-нибудь данные:

```
require 'model/book'
Book.new(:title => "Life on the Mississippi",
        :author => "Mark Twain" ).save
```

После перезапуска приложения и перезагрузки страницы мы увидим сведения о книге. А если книг больше? Вместо того чтобы «зашивать» в код данные о них, мы можем попросить контроллер выполнить поиск по имени автора. Добавим в него новый метод `find`:

```
def find(author)
  book = Book.find_by_author(author)
  @author = book.author
  @title = book.title
  @last_update = book.update_time
end
```

Это тот же код, что в методе `index` (и для создания соответствующего ему шаблона достаточно было бы переименовать `index.html` в `find.html`), но он принимает один аргумент – имя автора. Хотя мы разрабатываем Web-приложение, класс контроллера мало чем отличается от класса в любом другом приложении, написанном на Ruby. Самое заметное отличие заключается в том, что методы не возвращают значений, но никаких зависимостей от специальных переменных окружения или каких-то особых объектов нет. (Впрочем, имейте в виду, что Nitro все же добавляет разные «приятности» для программирования Web-приложений; для доступа к ним достаточно унаследовать свой класс от `Nitro::Controller`.) Например, Nitro обеспечивает «обстраивание», быструю передачу сообщений от одного запроса другому и хитроумный конвейер конструирования страницы, но все это только если вы сами захотите.

Сделаем еще одно замечание по поводу методов контроллера. Методы, предназначенные для обработки запросов на получение страницы, обычно идут в паре с шаблонами, имеющими схожее имя. Для получения конечного результата Nitro объединяет метод и шаблон. Мы видели, что Nitro может работать без контроллеров, ограничиваясь только видами. Но верно и обратное. Метод контроллера может генерировать всю страницу вообще без шаблона. Запросы на получение страниц обрабатываются в Nitro как действия (actions). *Действие* – это комбинация вида и методов контроллера. Внутри себя Nitro динамически создает методы действия, объединяющие то и другое. Но если какой-то компоненты недостает, ничего страшного не случится. Если у действия нет шаблона, результатом становится значение, возвращаемое методом контроллера.

Например, некоторый URL может отвечать на запрос, возвращая лишь название книги. Для этого достаточно такого метода контроллера `BookController`:

```
def sparse(author)
  @context.content_type = 'text/plain'
  book = Book.find_by_author(author)
  book.title
end
```

Если метод контроллера возвращает не HTML-документ, то следует изменить заголовок `content-type`, отправляемый вместе с ответом в результате присваивания `@context.content_type=`. (Кстати, даже если парный шаблон существует, его можно не использовать, переопределив метод `render_text`.)

Но откуда метод `find` или `sparse` получает аргумент? По умолчанию Nitro следует привычному для Web-приложений паттерну, согласно которому сегменты пути URL отображаются на контроллеры, методы и аргументы. Перезапустите приложение и введите такой URL: <http://127.0.0.1:9999/books/find/Hal%20Fulton>.

Обратите внимание, что `%20` – это кодированное представление пробела в имени автора. Nitro производит декодирование еще до вызова метода `find`.

В общем случае отображение URL выполняется так: `/controller/method/arg1`. Дополнительные аргументы можно передать в виде последующих сегментов пути. Впрочем, в Nitro есть механизм настройки маршрутизации, так что вы не обязаны отражать в URL детали реализации.

19.4.3. Nitro и Og

Хотя библиотека объектно-реляционного отображения Og, используемая в паре с Nitro, рассматривалась в другом месте, наше изложение было бы неполным без демонстрации того, как просто добавить к приложению интерфейс с базой данных.

Сначала настроим Og в файле `run.rb`:

```
# Прямо перед вызовом Book.new :
require 'og'
```

```
Og.setup(:store => 'mysql',
        :name => 'demo',
        :user => 'root',
        :destroy => true,
        :port => 3316)
```

Затем изменим модель `Book`:

```
require 'glue/timestamped'
```

```
class Book
  is Timestamped

  property :title, String
  property :author, String

  def initialize( values = {} )
    @title, @author = values[:title], values[:author]
  end
end
```

От хранения экземпляров `Book` в переменных класса мы отказались. Вызовы `attr_accessor` заменены обращением к методу `property`, который служит несколькими целям. Он тоже создает методы доступа к переменным, но, кроме того, сообщает Og, что этот класс должен обладать устойчивостью. Метод `initialize` почти не изменяется, но затребовав файл `timestamped` и поместив в класс маркер `is Timestamped`, мы автоматически получаем атрибут `update_time`.

Остальные методы можно удалить, теперь они реализуются библиотекой Og. При перезапуске приложения Nitro создаст для него базу данных MySQL, а в ней –

таблицу для хранения объектов book. Так разрабатываются приложения «на чистом Ruby без SQL».

19.4.4. Решение типичных для Web-приложений задач в Nitro

Nitro обладает поразительно богатой системой составления страниц. Ее полное описание выходит за рамки данной книги. Но любому Web-разработчику приходится постоянно решать некоторые типичные задачи, поэтому посмотрим, как это можно сделать в Nitro.

Если вы создаете сайт с одной-двумя страницами, не так уж важно, повторяются ли многократно одна и та же разметка и текст. Но когда страниц больше, обновлять все одинаковые фрагменты вручную становится утомительно и чревато ошибками. Nitro помогает следовать принципу DRY (Don't Repeat Yourself – «Не повторяйся»), предлагая целый ряд способов повторного использования.

Простейший из них – включение файла. Пусть, например, все страницы должны иметь общий хвостовик. Можно было бы поместить его в отдельный файл шаблона и включить во все страницы, как показано ниже:

```
<?include href='/footer' ?>
```

Сам файл `footer.xinc` мог бы выглядеть, к примеру, так:

```
<div id='footer'>Read More Ruby Books</div>
```

Если в качестве значения атрибута `href` указан относительный путь, Nitro будет просматривать папки `template`, определенные для текущего контроллера. Если же путь абсолютный, то просматривается только папка `template` в корневом каталоге приложения.

Показанный выше способ лучше всего подходит для статического контента. Существует и иной синтаксис включения, позволяющий вставить указанный файл в вызывающий шаблон непосредственно перед компиляцией шаблона:

```
<include href='/footer' />
```

Результат получается таким, как если бы включаемый текст был частью вызывающего шаблона.

Более сложная форма включения контента связана с элементом `render`:

```
<render href='/controller/action' />
```

где `href` – некий путь в каталоге приложения.

Процедура компиляции при включении частичных видов с помощью `<render />` практически не отличается от случая полного вида. В контроллере могут быть методы, соответствующие включаемому файлу; их можно использовать для установки значений переменных экземпляра во включаемом шаблоне.

В Nitro граница между кодом шаблона и кодом на Ruby размыта. Один из примеров мы уже приводили: действие контроллера может быть «размазано» между методом и файлом шаблона, а может целиком входить туда или сюда. Другой пример – элементы Nitro (Nitro Elements), способ инкапсуляции кода и разметки в нестандартном теге, который можно использовать в видах.

Вместо того чтобы определять в каждом виде HTML-разметку всей страницы, мы можем определить общий HTML-код и повторно использовать его в различных действиях. Создадим файл `element/layout.xhtml`:

```
<html>
  <head>
    <title>#{@title}</title>
    <style>
      body {
        background-color: white;
        font-family: sans-serif;
      }
    </style>
  </head>
  #{@content}
</html>
```

Теперь воспользуемся новым элементом в файле `template/books/find.xhtml`:

```
<Layout title='Details for #{@title}'>
  <h1>#{@title}</h1>
  <h2>#{@author}</h2>
  <p>Page last updated: #{@last_update}</p>
</Layout>
```

Все содержимое элемента `Layout` вставляется в переменную `content` в файле `layout.xhtml`. Элементы могут принимать параметры; атрибут `title` в открывающем теге `Layout` становится значением переменной экземпляра `@title` в файле `layout.xhtml`.

Вам это напоминает вызов метода с передачей ему аргументов? Так оно и есть. Мы можем определить разметку в виде класса Ruby (`src/element/layout2.rb`):

```
require 'nitro/element'

class Layout2 < Nitro::Element
  def render
    %^<html>
    <head>
      <title>#{@title}</title>
      <style>
        body {
          background-color: white;
          font-family: sans-serif;
        }
      </style>
    </head>
    #{@content}
    </html>^
  end
end
```

А затем изменим файл `find.xhtml`, воспользовавшись элементом `Layout2` (нужно еще будет затребовать класс нового элемента в файле `run.rb`). Элементы могут содержать другие элементы, так что виды можно собирать из повторно используемых компонентов.

Часто встречаются большие фрагменты кода, содержащие логику, общую для нескольких приложений. Например, во многих Web-приложениях есть понятие учетной записи пользователя и авторизации. Чем заново писать код для каждой такой программы, можно включить уже готовый: это сэкономит время и упростит сопровождение.

Такой вид повторного использования называется частью. *Часть* (part) – это, по существу, мини-сайт, реализующий одну конкретную функцию. (В дистрибутив Nitro входит одна такая часть под названием Admin.) Код в таком подсайте не нуждается в отдельном файле `run.rb`, хотя включать его полезно, если вы хотите, чтобы часть могла выполняться автономно в демонстрационных целях.

Естественно, части хранятся в папке `part`. Пусть имеется некий код аутентификации пользователя, который можно использовать повторно. Тогда дерево частей приложения могло бы выглядеть так:

```
<app_root>/part/users
<app_root>/part/users.rb
<app_root>/part/users/public/
<app_root>/part/users/controller.rb
<app_root>/part/users/model/user.rb
<app_root>/part/users/model/acl.rb
<app_root>/part/users/template/login.xhtml
<app_root>/part/users/template/form.xinc
<app_root>/part/users/run.rb
```

Главный файл `run.rb` мог бы включить такую часть с помощью одной директивы `require`:

```
require 'part/users'
```

Теперь Nitro будет рассматривать весь код в каталоге `part/users`, как если бы он находился в дереве исходных текстов главного приложения. Поиск шаблонов начинается в папке `template` приложения и продолжается в подкаталогах каталога `part`. Если вы хотите переопределить шаблон, поставляемый вместе с частью, то нужно просто поместить замещающий вариант в соответствующий подкаталог папки `template`.

Часто приложение должно отображать повторяющиеся данные; обычно они оформлены в виде HTML-таблицы в шаблоне. Если число строк заранее не известно, то придется обойти какой-то набор.

Можно встроить код Ruby непосредственно в шаблон, но Nitro предлагает специальный механизм конвейерной компиляции, чтобы упростить программирование типичных логических конструкций.

Конвейерная компиляция – это последовательность преобразований, которым подвергаются шаблоны по мере объединения в действия. Существуют классы преобразования для различных задач, в частности статического включения файлов,

XSLT-преобразований и локализации. Класс `Morphing` исследует разметку в шаблоне и ищет специальные атрибуты, которые обозначают различные преобразования.

В нашем демонстрационном приложении данных мало, но если представить себе много авторов, каждый из которых написал несколько книг, то можно было бы показать список книг данного автора. Метод контроллера поместил бы в переменную `@books` список книг, а часть шаблона для обхода этого цикла могла бы выглядеть так:

```
<h4>Books by #{@author}</h4>
<ul>
<li each="book in @books" > #{book.title}</li>
</ul>
```

Класс `Morphing` находит атрибут `each` элемента `li` и преобразует его в следующий код:

```
<?r for book in @books ?>
<li>#{book.title} </li>
<?r end ?>
```

Порожденная таким образом разметка передается следующему этапу конвейера.

Точно так же для повтора элемента можно использовать атрибут `times`. Например, фрагмент

```
<img src='/img/ruby.png' alt='' times='@book.rating' />
```

будет преобразован в следующий код:

```
<?r 3.times do ?>
<img src='/img/ruby.png' alt='' />
<?r end ?>
```

19.4.5. Прочие детали

В Nitro слишком много различных механизмов, чтобы все их можно было рассмотреть здесь. Но в этом разделе мы упомянем некоторые особенно приятные «вкусности».

В состав Nitro входит вспомогательный код, позволяющий использовать многочисленные библиотеки JavaScript, которые поддерживают различные формы DHTML и Ajax. Для облегчения интеграции в Nitro применяется высокоуровневый синтаксис. Например, в дистрибутиве Nitro есть пример поиска по сайту Flickr и вывода уменьшенных изображений. Текстовое поле для ввода тегов поиска поддерживает технологию Ajax, что достигается следующей разметкой:

```
<input type="text" id="tags" name="tags" auto_complete="true" />
```

Контроллер реализует метод `tags_auto_complete`, который возвращает строку в формате XML, зависящую от содержимого поля.

Nitro позволяет кэшировать действия (то есть отрисованные страницы целиком), результаты вызова методов и сгенерированные фрагменты текста. Например, чтобы поместить в кэш результат отрисовки страницы `index`, контроллер должен вызвать следующий метод:

```
cache_output :index ,
```


Кэширование можно добавить и в сегменты кода:

```
<?r cache(:book_list_cache_key) do ?>
  <ul>
    <li each="book in Books.all">#{book.title}</li>
  </ul>
<?r end ?>
```

В Nitro встроен класс преобразования для локализации, позволяющий автоматически подставлять в шаблон то или иное содержимое. Он не является стандартным этапом конвейерной компиляции, вы сами должны добавить его в файл `run.rb`:

```
require 'nitro/compiler/localization'
include Nitro
```

```
Compiler.transformation_pipeline = [
  StaticInclude,
  Elements,
  Morphing,
  Markup,
  Localization,
  Cleanup
]
```

Отметим, что конвейер можно переконфигурировать, выбросив стандартные преобразования и добавив собственные. Далее определяются локали:

```
Localization.locales = {
  :en => 'conf/locales/en.yml',
  :de => 'conf/locales/de.yml'
}
```

Локаль – это просто YAML-файл, сопоставляющий одним строкам другие:

```
---
:author: Autor
:language: Sprache
:book_rank: Buchrank
```

В шаблонах применяются специальные конструкции для обозначения подставляемого текста:

```
<div class='detail'>[:author]: #{@book.author}</div>
<div class='detail'>[:language]: #{@book.language}</div>
<div class='detail'>[:book_rank]: #{@book.rank}</div>
```

Выбор файла локали определяется значением `session[:LOCALE]`. В методах контроллера можно получить текущую локаль с помощью специальной переменной `@lc`.

```
@language = @lc[:language]
```

Если зависимости от языка более обширны, то можно организовать несколько каталогов шаблонов, по одному для каждого языка. Какой из них использовать, определяется значением установленной локали.

Дополнительную информацию о Nitro можно найти на следующих ресурсах:

- <http://www.nitroproject.org/> (домашняя страница Nitro)
- http://rubyforge.org/forum/forum.php?forum_id=5921 (страница проекта Nitro на сайте RubyForge);
- <http://oxyliquit.de/> (справочное и учебное руководство по Nitro).

19.5. Введение в Wee

Согласно заявлению автора Майкла Ноймана (Michael Neumann), *Wee* – это «каркас для создания очень динамичных, компонентных Web-приложений, на дизайн которого оказал большое влияние продукт Seaside». Название расшифровывается как «Web Engineering Easy» (сделаем конструирование Web проще).

Установить Wee проще всего из gem-пакета (`gem install wee`). Во время работы над книгой текущей была версия 0.10.0. В документации по Wee говорится, что, хотя код достаточно устойчив, могут возникать некоторые проблемы из-за продолжений, поэтому пока не стоит использовать этот каркас для критически важных приложений.

Но даже и с такими оговорками Wee заслуживает изучения ради своей компонентной модели, а также потому, что продолжения – интересная, но недостаточно исследованная область, лежащая в стороне от главного направления разработки для Web. Автор говорит, что на него повлияли идеи Seaside, а основанный на продолжениях каркас для Web-приложений на языке Smalltalk написал Ави Брайант (Avi Bryant).

В gem-пакет для инсталляции Wee входит большое число разнообразных примеров. Один из них – Web-интерфейс к обозревателю объектного пространства, другой – применение Ajax на основе библиотеки JavaScript-функций Prototype. Есть также пример, демонстрирующий совместную работу Wee и Nitro.

В основе Wee лежит идея компонентов. Они похожи на виджеты в графических интерфейсах. Компоненты легко поддаются повторному использованию, инкапсулируют состояние, представление и поведение, хотя все это можно делегировать внешним шаблонам или моделям.

19.5.1. Простой пример

Во время установки Wee создается генератор простых приложений, который, естественно, называется `wee`. Команда `wee create my-demo` создает подкаталог `my-demo` в текущем каталоге и записывает в него простое приложение на базе WEBrick.

Созданное приложение всего лишь подсчитывает, сколько раз пользователь щелкнул по ссылке. Файл `run.rb` на стороне сервера подготавливает компоненты приложения и главный класс, после чего запускает приложение под управлением сервера WEBrick.

```
require 'wee'
require 'wee/utils'
require 'wee/adaptors/webbrick'
```

```
# Ваши компоненты.
```

```
require 'components/main'
```

```
app = Wee::Utils.app_for do
  Main.new.add_decoration(Wee::PageDecoration.new('Wee'))
end
Wee::Utils::autoreload_glob('components/**/*.rb')
Wee::WEBrickAdaptor.register('/app' => app).start
```

Класс `Main` вызывается как главный компонент приложения. Каждый компонент должен реализовать метод `render`, порождающий разметку. Вызов метода `add_decoration(Wee::PageDecoration.new('Wee'))` изменяет конвейер построения страницы так, что результаты обращения к `Main#render` дополняются HTML-кодом заголовка и хвостовика.

Далее конфигурируется автоматическая перезагрузка файлов, чтобы можно было изменить код и проверить, как работает новая версия приложения, не перезапуская `WEBrick`. И наконец, запускается экземпляр сервера `WEBrick`, который обслуживает запросы к URL, начинающемуся с пути `'/app'`. По умолчанию подразумевается порт 2000, но можно при запуске указать любой другой номер в качестве параметра:

```
Wee::WEBrickAdaptor.register('/app' => app).start(:Port => 8787 )
```

Компонент `Main` определяет метод `render`, который порождает разметку.

```
class Main < Wee::Component

  def initialize
    super()
    # Здесь должен быть ваш код инициализации...
  end

  def render
    r.anchor.callback(:click).with { r.h1("Welcome to Wee!") }
    r.text "#{ @clicks || 'No' } clicks"
  end

  def click
    @clicks = (@clicks || 0) + 1
  end

end
```

`Wee` позволяет пользоваться синтаксисом Ruby для генерации HTML-кода примерно так же, как библиотека `XML Builder` Джима Вайриха и генератор XML в `Nitro`. Однако в `Wee` можно еще связать ссылку с действием (в данном случае с методом `click`). Когда пользователь щелкает по ссылке, сгенерированной `Wee`, приложение понимает, что нужно вызвать метод `click`.

19.5.2. Ассоциирование состояния с URL

В примере выше отслеживается текущее значение переменной `@click`, но она не связывается с URL. Если вы запустите эту программу, что увидите, что `Wee`

генерирует довольно длинный URL, который по сути является GUID'ом (globally unique identifier, глобально уникальным идентификатором). URL остается таким же, если не считать завершающего символа косой черты и целого числа. При каждом щелчке по ссылке Welcome to Wee число увеличивается на единицу.

Если вручную изменить URL в браузере, то вы получите ту же самую страницу, показываемый счетчик щелчков не изменится. Нет никакой связи между URL и состоянием сервера. (Когда будете ставить эксперимент, не забудьте отключить кэширование в браузере).

Впрочем, это положение можно изменить, слегка модифицировав файл `main.rb`. Добавьте в метод `Main` такой код:

```
def backtrack_state(snap)
  super
  snap.add(self)
end
```

Теперь перезапустите приложение. Щелкнув несколько раз по ссылке, вручную измените URL в браузере так, чтобы повторно загрузилась предыдущая страница. Теперь счетчик должен показать то значение переменной `@click`, которое соответствует моменту генерации данного URL.

Чтобы проверить то же самое с использованием механизма продолжений Wee, добавьте после директив `require` в файл `run.rb` следующую строку:

```
require 'wee/continuation'
```

В Wee слишком много возможностей, чтобы рассмотреть их здесь подробно. Дополнительную информацию можно найти на следующих ресурсах:

- страница проекта Wee (<http://rubyforge.org/projects/wee/>);
- страница проекта Nemo (<http://rubyforge.org/projects/nemo/>);
- проект Seaside (<http://seaside.st/>)

Одна из интересных особенностей – возможность иметь вложенные компоненты и организовывать цепочки обязанностей, что позволяет собирать сайты из повторно используемых компонентов пользовательского интерфейса. Стоит также познакомиться с проектом Nemo – реализацией Mewa (Meta-level Architecture for Web Applications, метауровневая архитектура Web-приложений) на Wee.

19.6. Разработка Web-приложений с помощью IOWA

IOWA (Interpreted Objects for Web Applications – интерпретируемые объекты для Web-приложений) – это каркас, написанный Кирком Хейнсом (Kirk Haines). Он позволяет создавать повторно используемые, инкапсулированные Web-компоненты для генерации сайта.

19.6.1. Основные идеи IOWA

Приложение IOWA работает как фоновый процесс, который прослушивает сокет в ожидании запросов. IOWA включает различные адаптеры, так что источником запроса может быть CGI, Mongrel, WEBrick и т.д.

На домашней странице IOWA представлены хорошее объяснение архитектуры и учебное руководство, поэтому здесь мы приведем лишь краткий обзор основных особенностей.

Загрузить IOWA можно с сайта rubyforge.org. Gem-пакет будет подготовлен вместе с версией 1.0. Примеры, представленные в этом разделе, составлены на базе предварительной версии, доступной в виде zip- или tgz-файла.

В состав дистрибутива входит несколько примеров и тестов, демонстрирующих работу каркаса. Мы рассмотрим простое приложение, разработанное на основе этих примеров.

Для приложения IOWA нужен код, который запустит фоновый процесс, а в качестве фронтального процесса мы будем пользоваться встроенным сервером WEBrick. Сценарий `app.rb` решает обе задачи:

```
require 'iowa_webrick'

class HwApplication < Iowa::Application
  self.daemonize = true
  attr_accessor :dbpool

  def initialize(*args)
    super
    Iowa.config[Iowa::Cappplication][Iowa::Croot_url] = 'http://127.0.0.1:2000'
  end
end

Iowa.run
```

По умолчанию конфигурационные данные читаются из файла `app.cnf` в текущем каталоге. Вот как выглядит наш файл:

```
socket:
  hostname: localhost
  path: ..
logging:
  basedir: ../log
  minlevel: 0
  maxsize: 10000000
  maxage: 86400
application:
  daemonize: false
  sessioncache:
    class: LRUCache
    maxsize: 20
    ttl: 3600
  dispatcher:
    class: StandardDispatcher
  policy:
    class: iowa/Policy
```

Этот файл, записанный в формате YAML, содержит различную информацию о желаемом поведении IOWA. Необходим и еще один конфигурационный файл

mapfile.cnf, в котором описывается, как запросы отображаются на компоненты. Вот пример такого файла, содержащий всего одну строку:

```
/main.html: Main
```

Запросы к IOWA обычно обслуживаются комбинацией HTML-шаблонов и компонентов IOWA. Парные файлы имеют одинаковое базовое имя, но разные расширения. Принимаемый по умолчанию шаблон/объект называется main, так что приложение состоит из файлов main.html и main.iwa.

Файлы с расширением .iwa представляют собой программы на Ruby; такое расширение в IOWA используется для того, чтобы отличить их от прочего кода, который может входить в состав приложения. Эти файлы играют ту же роль, что классы контроллеров в Nitro и Rails. Определенные в классе компонента методы доступны из соответствующего HTML-файла.

Демонстрационный файл main.html выглядит так:

```
<html>
  <head><title>Текущее время...</title></head>
  <body>
    <p>Текущее время @now.</p>
    <p>Счетчик равен @count.</p>
    <a oid="reload">RELOAD</a>
  </body>
</html>
```

В шаблонах IOWA можно смешивать обычный HTML-код и переменные экземпляра компонента. Отметим, что переменные необязательно «интерполировать» обычным способом, достаточно просто включить их в разметку.

Имеется также специальная переменная oid, IOWA использует ее для динамического изменения шаблона в процессе прорисовки. В нашем примере с ее помощью создается ссылка на метод reload из класса компонента, определенного в файле main.iwa. Если задержать мышь над этой ссылкой в полученной странице (или посмотреть ее исходный текст), то вы увидите URL следующего вида:

```
http://127.0.0.1:2000/main.html/6b38f6fb-4f087af7-ab6JaQUM9KyWE.a.1.7
```

Такие URL позволяют IOWA отслеживать состояние сеанса. Если несколько раз щелкнуть по ссылке, то вы увидите, что URL изменяется. Если вручную восстановить предыдущее значение, вы получите состояние сеанса, соответствующее URL.

В данном случае состояние – это значение переменной экземпляра @count. Вот как выглядит файл main.iwa:

```
class Main < Iowa::Component
  attr_accessor :count
  def awake
    @count = 0
  end

  def setup
    @count += 1
  end
end
```

```

def now
  Time.now.asctime
end
end

```

19.6.2. Шаблоны в IOWA

Большинство Web-приложений только выигрывает от разделения кода и шаблонов представления, но IOWA, как и Nitro, позволяет вообще обходиться без компонентов и помещать весь код на Ruby в представление. Ниже приведен файл PureView.html, содержащий как код класса, так и HTML-разметку:

```

<%
  class PureView < Iowa::Component
    def right_now
      Time.now
    end
  end
%>
<html>
  <head><title>Автономный вид</title></head>
  <body>
    <p>Текущее время @right_now.</p>
  </body>
</html>

```

Но, в отличие от Nitro, это работает только в том случае, когда для представления нет соответствующего компонента. Если имеются оба файла, то IOWA не станет анализировать код, встроенный в HTML-файл.

Шаблон может содержать циклы и условные предложения. Добавим такой метод в файл main.iwa:

```

def after_dinner?
  Time.now.hour > 19
end

```

Тогда можно реализовать в Main.html условную прорисовку с помощью элемента if:

```

<if oid='after_dinner?'>
  <p>Обед закончен. Что на десерт?</p>
</if>

```

Хороший вопрос! Что на десерт? Пусть IOWA и ответит. Мы поручим Main.iwa подготовить меню десертов в виде массива:

```

def desserts
  %w{ Пирожные
      Печенье
      Фрукты
      Мороженое
    }
end

```

А в файле `Main.html` выведем его. Изменим содержимое элемента `if`, включив список десертов:

```
<p>Обед закончен. Вот что мы можем предложить на десерт:</p>
<ul oid="dessert_list">
  <li>@dessert_item</li>
</ul>
</if>
<p>
```

Мы еще должны сообщить IOWA, что выводить на каждой итерации, поэтому в конце файла `Main.iwa` после определения класса добавим раздел определения связей:

```
<?
dessert_list {
  item = dessert_item
  list = desserts
}
?>
```

Тем самым производится связывание списка `dessert_list` в шаблоне. На каждой итерации элемент списка заполняется из переменной `dessert_item`, а данные в целом поступают от метода компонента `desserts`.

19.6.3. Передача управления компоненту

Бывает полезно разнести логику приложения по нескольким классам компонентов. Мы видели, как можно отобразить URL на компоненты. Имеется также способ передать управление, не изменяя базового пути в URL.

Добавим в файл `Main.iwa` метод для обработки щелчка по ссылке в меню десертов:

```
def dessert_choice
  new_page = page_named('DessertChoice')
  new_page.choice = @dessert_item
  yield new_page
end
```

Также изменим цикл формирования списка десертов в `Main.html`:

```
<ul oid="dessert_list">
  <li><a oid='dessert_choice'>@dessert_item</a></li>
</ul>
```

Тут происходит немало интересного; атрибут `oid` элемента `ul` управляет формированием цикла, а такой же атрибут элемента `a` создает специальную ссылку на то, что добавленный метод `dessert_choice`. Для довершения дела странице передается еще и текст ссылки (хотя и несколько загадочным способом). Метод `dessert_choice` сам по себе короткий, в нем вызывается метод `page_named` для создания экземпляра еще одного класса компонента `DessertChoice`. Для передачи выбранного десерта вызывается метод `choice=`. Затем `yield` передает управление новому компоненту.

Новый компонент также определяется с помощью пары файлов с расширениями `.iwa` и `.html`. Вот код класса:

```
class DessertChoice < Iowa::Component
  attr_accessor :choice
  def details
    "Детали #{@choice} нужно было брать из базы данных."
  end
end
```

А в файле `DessertChoice.html` хранится разметка:

```
<html>
<head><title>Выбранный вами десерт</title></head>
<body>
<h1>Десерт!</h1>
<p>@details</p>
</body>
</html>
```

Об IOWA можно было бы рассказывать еще долго. Для получения дополнительной информации зайдите на домашнюю страницу IOWA (<http://enigo.com/projects/iowa/>) или на страницу проекта IOWA на сайте RubyForge (<http://rubyforge.org/projects/iowa>).

19.7. Ruby и Web-сервер

На сегодняшний день одним из самых популярных Web-серверов является Apache. Если вы работаете с ним, то должны знать о модуле `mod_ruby`, который описывается в разделе 19.7.1.

Еще одна полезная возможность на стороне сервера – встроенный Ruby; эту технологию поддерживают инструменты `erb` (рассматриваемый ниже) и `eruby`. Они позволяют встраивать код на Ruby в текст страницы (обычно HTML или XML), вследствие чего данные можно вставлять динамически. Данный подход описывается в разделе 19.7.2.

Некоторые разработчики реализовали Web-серверы, написанные целиком на Ruby. Естественно возникает вопрос: зачем писать новый Web-сервер, когда их уже и так существует немало – взять хотя бы тот же Apache?

Во-первых, есть ситуации, когда желательно иметь специализированный Web-сервер, например ради нестандартного способа обработки страниц, когда можно пожертвовать функциональностью ради скорости, или для автоматической трансляции специальной разметки в HTML.

Во-вторых, может возникнуть желание поэкспериментировать с поведением сервера и его взаимодействием с внешним кодом, например с CGI-программами. Возможно, у вас есть какие-то идеи относительно создания сервера приложений и среды разработки на стороне сервера. А все мы знаем, что Ruby прекрасно подходит для экспериментов.

В-третьих, иногда бывает разумно встроить Web-сервер в другое приложение. К этой возможности прибегают разработчики, желающие предоставить

функциональность программной системы внешнему миру; протокол HTTP прост и четко определен, а Web-браузеры в качестве клиентов есть повсюду. Этот прием можно даже использовать для удаленной отладки, если система часто обновляет свое внутреннее состояние и делает его доступным встроенному серверу.

И последняя причина заключается в том, что небольшой автономный Web-сервер может упростить развертывание и конфигурирование. Например, перезапустить сервер для приложения Rails гораздо проще, если в этом качестве выступает WEBrick, а не Apache.

Имея все это в виду, посмотрим, что Ruby предлагает в плане Web-серверов. В прошлом было по крайней мере четыре таких сервера, но летом 2006 года остались два наиболее значимых: WEBrick и Mongrel. Они описаны в разделах 19.7.3 и 19.7.4 соответственно.

19.7.1. Модуль `mod_ruby`

Обычно, если CGI-сценарий пишется на интерпретируемом языке, то при каждом запросе загружается новый экземпляр интерпретатора. Это дорого обходится с точки зрения потребления ресурсов сервера и времени выполнения.

Сервер Apache решает эту проблему путем создания загружаемых модулей, которые, по существу, становятся частью сервера. Они загружаются динамически по мере необходимости и становятся общими для всех зависящих от них сценариев. Одним из таких модулей является `mod_ruby` (имеется в архиве RAA).

Модуль `mod_ruby` реализует несколько директив Apache, в частности:

- `RubyRequire` определяет одну или несколько потребных библиотек;
- `RubyHandler` определяет обработчик для объекта Ruby;
- `RubyPassEnv` определяет имена переменных окружения, передаваемых сценариям;
- `RubySetEnv` устанавливает переменные окружения;
- `RubyTimeout` задает величину тайм-аута для Ruby-сценариев;
- `RubySafeLevel` задает уровень безопасности `SSAFE`;
- `RubyKanjiCode` устанавливает кодировку символов для Ruby.

В состав пакета входят также классы и модули Ruby для взаимодействия с Apache. Модуль Apache (здесь слово «модуль» употребляется в смысле, принятом в Ruby) включает несколько функций, например `server_version` и `unescape_url`; там же определены классы `Request` и `Table`.

`Apache::Request` – это обертка для типа данных `request_rec`, определяющая такие методы, как `request_method`, `content_type`, `readlines` и т.д. Класс `Apache::Table` – обертка для типа данных `table`; он определяет, среди прочих, методы `get`, `add` и `each`.

Имеются подробные инструкции по компиляции и установке пакета `mod_ruby`. Обратитесь к поставляемой в комплекте с ним документации (или эквивалентной информации в Сети).

19.7.2. Использование erb

Для начала договоримся о терминологии. Речь не идет о встраивании интерпретатора Ruby в электронные устройства, скажем, в телевизор или тостер. Мы говорим о встраивании кода на Ruby в текст.

Далее отметим, что встраивать Ruby-код в текстовые файлы можно несколькими способами. В этом разделе мы рассмотрим лишь самый общеупотребительный инструмент, а именно программу `erb` (автор Шуго Маэда).

Почему мы упоминаем подобный инструмент в связи с Web? Очевидно, потому, что чаще всего Ruby-код встраивается в HTML или XML-тексты.

Но можно придумать и другие применения. Например, в старомодных текстовых играх, или в каких-нибудь утилитах слияния почты, или как часть задания `cron` для динамического создания файла с «сообщением дня» (`/etc/motd`) в полночь. Не ограничивайте свое воображение. Ищите новые интересные применения `erb` и делитесь своими находками с сообществом. Большая часть примеров в этом разделе носит общий (и потому искусственный) характер, конкретно с HTML они почти не связаны.

Утилита `erb` — это просто фильтр или препроцессор. Для выделения кода, выражений и комментариев, написанных на Ruby, применяется специальная нотация, весь остальной текст передается без изменений.

Текст, который нужно специально обрабатывать, заключается в скобки `<% и %>`. Есть три вида такой нотации, отличающиеся первым символом внутри «тега».

Если первый символ — знак равенства (=), то содержимое рассматривается как выражение Ruby; результат его вычисления подставляется в текущее место файла, например:

```
This is <%= "ylno".reverse %> a test.  
Do <%= "NOT".downcase %> be alarmed.
```

Если назвать этот файл `myfile.txt`, то для его фильтрации надо будет выполнить команду:

```
erb myfile.txt
```

Результат направляется на стандартный вывод:

```
This is only a test.  
Do not be alarmed.
```

Комментарий вводится символом `#`:

```
Life <## so we've heard %> is but a dream.
```

Как и следовало ожидать, комментарии игнорируются, то есть мы увидим такой результат:

```
Life is but a dream.
```

Все остальные символы после знака процента рассматриваются как часть кода. Все, что этот код выводит (не результат вычисления значения), помещается в текстовый поток. Для удобства восприятия рекомендую оставлять после процента пробел, хотя `erb` этого и не требует.

В следующем примере тег в первой строчке ничего не вставляет в текст, так как ничего не выводит. Во вторую же строку вставляется 42, как и следовало ожидать.

Ответ равен `<% "42" %>`.

А точнее, ответ равен `<% puts "42" %>`.

Результат получается такой:

Ответ равен .

А точнее, ответ равен 42.

Фрагменты кода на Ruby не являются независимыми. Так, переменную, определенную в одном теге, можно использовать в следующем за ним.

```
<% x=3; y=4; z=5 %>
```

Если стороны треугольника равны `<%=x%>`, `<%=y%>` и `<%=z%>`,

мы знаем, что он прямоугольный, поскольку

```
<%= x*x %> + <%= y*y %> = <%= z*z %>.
```

Пробелы внутри тегов в последней строке необязательны, но с ними текст становится понятнее. Выводится следующий текст:

Если стороны треугольника равны 3, 4 и 5,

мы знаем, что он прямоугольный, поскольку

```
9 + 16 = 25.
```

Попробуйте сделать внутри тега синтаксическую ошибку. Вы обнаружите, что `erb` выдает очень подробное сообщение, в котором печатается сгенерированный код и максимально точно указывается место ошибки.

Что если включить «магическую строку» в сам текст в виде литерала? Экранирование с помощью обратной косой черты не работает. Мы рекомендуем такой способ:

В этой строке есть цепочка меньше-процент `<%= "<%" %>`,

а в этой больше-процент `<%= "%"+ ">" %>`.

Здесь мы видим `<%= "<%" %>` и `<%= "%"+ ">" %>`.

Результат таков:

В этой строке есть цепочка меньше-процент `<%`,

а в этой больше-процент `>%`.

Здесь мы видим `<%` и `<#%`.

Включать открывающий тег немного проще, чем закрывающий. Объясняется это тем, что теги не могут быть вложенными, а программа `erb` недостаточно «умная», чтобы игнорировать закрывающий символ внутри строки.

Конечно, в `erb` есть кое-какие функции, «заточенные» под HTML. Для установки режима работы предназначен флаг `-m`, он может принимать значения `f`, `s` и `n`.

Режим `f` (фильтр) подразумевается по умолчанию, поэтому во всех предыдущих примерах мы не задавали флаг `-mf` явно. В режиме `-ms` (CGI) все ошибки печатаются в виде HTML. В режиме `-mn` (NPH-CGI – без разбора заголовков) автоматически выводятся дополнительные HTTP-заголовки. В двух последних режимах для безопасности переменная `$SAFE` устанавливается в 1 (исходя из допущения, что это CGI-приложение, которое может вызвать враждебный пользователь). Флаг `-n` (и эквивалентный ему `--noheader`) подавляет вывод CGI-заголовка.

Можно настроить сервер Apache так, что он будет распознавать страницы с встроеным кодом на Ruby. Для этого нужно ассоциировать тип `application/x-httpd-erb` с каким-нибудь расширением (было бы логично использовать `.rhtml`) и определить

действие, которое ассоциирует этот тип с исполняемым файлом `eruby`. Более подробные сведения вы найдете в документации по Apache.

19.7.3. Сервер WEBrick

Авторами WEBrick являются Масаёси Такахашаи (Masayoshi Takahashi) и Юзоу Готоу (Yuuzou Gotou) при участии многих других разработчиков. Это библиотека для создания полноценного HTTP-сервера; она входит в стандартный дистрибутив Ruby. Название происходит от слова «brick» (кирпич) – то есть подразумевается нечто небольшое, компактное и автономное.

WEBrick почти ничего не знает о деталях Web-приложений. Он не понимает, что такое сеанс пользователя и прочие тонкости. Он оперирует лишь серверами, работающими независимо друг от друга. Если вам необходима функциональность более высокого уровня, поищите другую библиотеку (возможно, надстройку над WEBrick наподобие IOWA или Tofu) или напишите свою собственную.

Работа с WEBrick сводится к такой последовательности действий: создаете экземпляр сервера; определяются обработчики монтирования и обработчики сигналов; запускается сервер. Вот небольшой пример:

```
require 'webrick'

server = WEBrick::HTTPServer.new(:DocumentRoot => '.')
# (В этом простом примере нет обработчиков монтирования)
trap('INT') { server.shutdown}
trap('TERM') { server.shutdown}
server.start
```

Запустив эту программу, вы получите Web-сервер, работающий на стандартном порте 80. Он раздает файлы из текущего каталога.

Для создания сервлета требуется создать класс, производный от `WEBrick::HTTPServlet::AbstractServlet`. При обработке указанного в запросе URL сервер ищет самый длинный префикс (наилучшее соответствие). Ниже приведен «пустой» пример (в нем обработчики не делают ничего полезного):

```
class EventHandler < HTTPServlet::AbstractServlet
  # ...
end
class RecentHandler < HTTPServlet::AbstractServlet
  # ...
end
class AlphaHandler < HTTPServlet::AbstractServlet
  # ...
end

# ...

server.mount('/events', EventHandler)
server.mount('/events/recent', RecentHandler)
server.mount('/events/alpha', AlphaHandler)
```

Как работает сервлет? Идея в том, чтобы определить метод для каждой поддерживаемой HTTP-операции, например `do_GET` для запросов типа GET. Если вы привыкли писать программы, обращающиеся к серверу, то теперь придется встать на противоположную точку зрения, ведь ваш код становится частью Web-сервера. Вы не получаете ошибку с кодом 404, а сами посылаете этот код. Вот простой пример:

```
class TinyHandler < WEBrick::HTTPServlet::AbstractServlet
```

```
  def do_GET(request, response)
    # Обработать запрос, вернуть ответ.
    status, ctype, body = process_request(request)
    response.status = status
    response['Content-type'] = ctype
    response.body = body
  end
```

```
  def process_request(request)
    text = "Очень короткая Web-страница..."
    return 200, "text/html", text
  end
```

```
end
```

В более сложном сервлете, вероятно, использовался бы метод `initialize`. Тогда передаваемые ему параметры были бы последними при вызове метода `server.mount`.

К счастью, не нужно писать сервлеты для каждой крохотной задачи, которую должен выполнять WEBrick. У него есть несколько своих предопределенных сервлетов (все в пространстве имен `WEBrick::HTTPServlet`):

- `FileHandler`
- `ProcHandler`
- `CGIHandler`
- `ERBHandler`

Поскольку сервлет `ProcHandler` особенно интересен, скажем о нем несколько слов. Он позволяет «лениться», то есть не создавать подкласса `AbstractServlet`. Вместо этого мы передаем `proc`:

```
# Непосредственное монтирование блока...
```

```
server.mount_proc('/here') do |req, resp|
  resp.body = "Это то, что выводит мой блок."
end
```

```
# Создать объект Proc и смонтировать его...
```

```
some_proc = Proc.new do |req, resp|
  resp.body = ' Это то, что выводит мой Proc.'
end
```

```
server.mount_proc('/there', some_proc)
# Другой способ смонтировать Proc...
```

```
my_handler = HTTPServlet::ProcHandler.new(some_proc)
server.mount('/another', my_handler)
```

WEBrick располагает и многими другими возможностями, например точками подключения для выполнения дополнительных задач (скажем, запуска какой-нибудь программы при старте). Имеются также развитые средства протоколирования, аутентификации по протоколу HTTP и т.д. Дополнительную информацию ищите в онлайн-официальной документации на сайте <http://ruby-doc.org> или где-либо еще.

19.7.4. Сервер Mongrel

Автором Mongrel является Зед Шоу (Zed Shaw) при участии других. Основная цель этого сервера – повысить производительность по сравнению с WEBrick. В данном отношении он добился значительных успехов и работает во много раз быстрее (хотя провести точные измерения сложно: результат зависит от многих факторов).

Mongrel часто используют в сочетании с Rails, а документация местами ориентирована на Rails. Но жесткой привязки к Rails нет – сервер может работать и в других контекстах.

Mongrel – скорее, приложение, тогда как WEBrick больше напоминает библиотеку. Во многом они схожи, но порядок запуска и API различаются.

Во многих случаях Mongrel можно запустить как приложение без написания какого-либо кода. Он понимает три основные команды: `start`, `stop` и `restart`. У команды `start` много параметров, модифицирующих поведение сервера, например: `--port portnum`, `--log filename`, `--daemonize` и т.д. Чтобы получить полный список, введите такую команду:

```
mongrel_rails start -h
```

Запуск в стандартном режиме – это хорошо, но рано или поздно возникнет нужда в чем-то необычном. На этот случай предусмотрены конфигурационные файлы.

Простой способ подготовить конфигурационный файл для Mongrel – воспользоваться флагом `-G`. Например, можно ввести такую командную строку:

```
mongrel_rails start -G myconfig.yml -p 3000 -r /home/hal/docs -l my.log
```

Заданные параметры будут сохранены (в формате YAML) в файле `myconfig.yml` (при наличии флага `-G` сервер завершается сразу после создания конфигурационного файла).

Для чтения параметров из конфигурационного файла задайте флаг `-C`:

```
mongrel_rails start -C myconfig.yml
```

Не употребляйте флаг `-s` вместе с другими. Он предполагает, что все параметры находятся в указанном файле.

Mongrel предлагает API для тонкой настройки поведения сервера. Флаг `-s` позволяет задать имя сценария, написанного с использованием этого API, который представляет собой небольшой язык предметной области (DSL – Domain-Specific Language). В документации приведен пример такого сценария (который добавляет обработчик для каталога, отличного от текущего):

```
# Файл: config/mongrel.conf
uri "/newstuff", :handler => DirHandler.new("/var/www/newstuff")

# Вызывается такой командой:
# mongrel_rails start -S config/mongrel.conf
```

Mongrel можно использовать и так же, как WEBrick. Следующая интуитивно понятная программа прекрасно работает:

```
require 'mongrel'

class TinyHandler < Mongrel::HttpHandler
  def process(request, response)
    response.start(200) do |head,out|
      head["Content-Type"] = "text/html"
      out.write <<-EOF
      Это всего лишь тест...

      EOF
    end
  end
end
```

```
server = Mongrel::HttpServer.new("0.0.0.0", "3000")
server.register("/stuff", TinyHandler.new)
server.register("/other", Mongrel::DirHandler.new("./other"))
server.run.join # Ждать в потоке сервера
```

Если вы активно работаете с Mongrel, то вас может заинтересовать система GemPlugin. Это, по сути, автозагружаемые gem-пакеты, которые становятся «частью» Mongrel. Например, модуль «Mongrel cluster» позволяет легко управлять кластером серверов Mongrel.

О Mongrel можно рассказывать еще долго. Дополнительная информация о протоколировании, отладке, деталях системы подключения модулей и пр. содержится в онлайн-овой документации на сайте <http://mongrel.rubyforge.org>.

19.8. Заключение

В этой главе мы рассмотрели низкоуровневые детали программирования CGI-приложений для Web. Мы видели, как в этом помогают такие инструменты, как модуль mod-ruby. Существует еще немало инструментов, помимо упомянутых в этой книге, например ruby-web, Amrita, Tofu и Cerise. Как обычно, перед тем как сделать выбор, поищите альтернативы.

Мы также познакомились с высокоуровневыми библиотеками и каркасами: Rails, Nitro, IOWA и Wee. Было уделено внимание и автономному серверу WEBrick (и гораздо более быстрому Mongrel).

Теперь обратимся к иному, значительно более простому и абстрактному способу сетевого программирования. Следующая глава посвящена распределенной обработке с применением Ruby.

Глава 20. Распределенный Ruby

.....
Меньше – это больше.

Робер Браунинг, «Андреа дель Карто»

В настоящее время имеется немало технологий, поддерживающих распределенную обработку: различные варианты RPC, а также COM, CORBA, DCE и Java RMI.

Одни проще, другие сложнее, но в принципе все делают одно и то же – предоставляют относительно прозрачный способ связи между находящимися в сети объектами так, чтобы с удаленными объектами можно было работать, как с локальными.

Зачем это вообще может понадобиться? Причин много. Например, чтобы распределить некоторую вычислительную задачу между многими процессорами. Примером может послужить программа SETI@home, которая использует ваш ПК для обработки небольших объемов данных в поисках внеземного разума (кстати, эта программа не является проектом института SETI). Другой пример – привлечение широких масс к взлому шифра RSA129 (эта попытка увенчалась успехом несколько лет назад). Существует очень много задач, которые можно разбить на небольшие части, пригодные для распределенного решения.

Можно также представить себе, что вы хотите предоставить интерфейс к некоему сервису, не раскрывая исходных текстов. Часто это делается с помощью Web-приложений, но из-за отсутствия состояния в протоколе HTTP это не всегда удобно (есть и другие недостатки). Механизм распределенного программирования позволяет решать подобные задачи более естественно.

В мире Ruby ответом на этот вызов стала программа `drb`, написанная Масатоси Секи (Masatoshi Seki); еще ее название записывают так: `DRb`. Существуют и другие способы распределенной обработки на Ruby, но `drb`, пожалуй, самый легкий. Здесь нет сложных служб разрешения имен, как в CORBA. Это всего лишь простая и удобная библиотека, предоставляющая всю необходимую функциональность. В данной главе мы рассмотрим основы работы как с ней самой, так и с надстроенной над ней системой Rinda.

20.1. Обзор: библиотека `drb`

Библиотека `drb` состоит из двух основных частей: серверной и клиентской. Грубую границу между ними можно провести следующим образом:

Сервер:

- запускает `TCPServer` и начинает прослушивать порт;
- привязывает объект к экземпляру сервера `drb`;

- принимает запросы на соединение от клиентов и отвечает на их сообщения;
- дополнительно может предоставлять контроль доступа (безопасность).

Клиент:

- устанавливает соединение с сервером;
- привязывает локальный объект к удаленному экземпляру сервера;
- посылает сообщения серверу и получает ответы.

Метод класса `start_service` отвечает за запуск TCP-сервера, прослушивающего указанный порт. Он принимает два параметра: URI (универсальный идентификатор ресурса), задающий порт (если он равен `nil`, то порт выбирается динамически), и объект, к которому мы хотим привязаться. Этот объект будет доступен удаленному клиенту, который сможет вызывать его методы, как если бы объект был локальным.

```
require "drb"
```

```
myobj = MyServer.new
```

```
DRb.start_service("druby://:1234", myobj) # Порт 1234.
```

```
# ...
```

Если порт выбирается динамически, то для получения полного URI, включающего и номер порта, можно воспользоваться методом класса `uri`.

```
DRb.start_service(nil, myobj)
```

```
myURI = DRb.uri # "druby://hal9000:2001"
```

Поскольку `drb` – многопоточная программа, любое серверное приложение должно выполнять `join` в потоке сервера (чтобы не дать приложению завершиться преждевременно и тем самым уничтожить выполняющийся поток).

```
# Предотвратить преждевременный выход.
```

```
DRb.thread.join
```

На стороне клиента мы вызываем метод `start_service` без параметров и с помощью класса `DRbObject` создаем локальный объект, соответствующий удаленному. Обычно первым параметром методу `DRbObject.new` передается `nil`.

```
require "drb"
```

```
DRb.start_service
```

```
obj = DRbObject.new(nil, "druby://hal9000:2001")
```

```
# Сообщения, передаваемые obj, перенаправляются
```

```
# удаленному объекту на стороне сервера...
```

Следует подчеркнуть, что на стороне сервера привязка осуществляется к единственному объекту, который должен отвечать на все получаемые запросы. Если клиентов несколько, то объект должен быть безопасным относительно потоков, чтобы не оказаться в некорректном состоянии. (Для совсем простых или узкоспециализированных приложений это может быть и необязательно.)

Мы не можем вдаваться в технические детали. Но имейте в виду, что если клиент читает или изменяет внутреннее состояние удаленного объекта, то при наличии нескольких клиентов возможна интерференция. Во избежание таких неприятностей мы рекомендуем применять механизмы синхронизации, например класс `Mutex`. (Подробнее о потоках и синхронизации рассказывается в главе 13.)

Скажем хотя бы несколько слов о безопасности. Ведь не всегда желательно, чтобы с вашим сервером мог соединяться кто угодно. Помешать им пытаться вы не можете, зато можете сделать такие попытки безуспешными.

В программе `drb` есть понятие списка контроля доступа (ACL). Это не что иное, как списки клиентов (или категорий клиентов), которым явно разрешен (или запрещен) доступ.

Приведем пример. Для создания нового списка ACL мы воспользуемся классом `ACL`, которому передадим один или два параметра.

Второй (необязательный) параметр метода `ACL.new` служит для ответа на вопрос: «Мы запрещаем доступ всем клиентам, кроме некоторых, или, наоборот, разрешаем доступ всем клиентам, кроме некоторых?» По умолчанию принимается первый вариант, который обозначается константой `DENY_ALLOW` равной 0. Второй режим обозначается `ALLOW_DENY` равной 1.

Первый параметр `ACL.new` представляет собой обычный массив строк, которые идут парами. Первая строка в паре должна быть равна `"deny"` или `"allow"`, вторая описывает одного клиента или группу клиентов (по имени или по адресу):

```
require "drb/acl"
acl = ACL.new( %w[ deny all
                  allow 192.168.0.*
                  allow 210.251.121.214
                  allow localhost] )
```

Первая пара в этом примере, строго говоря, излишня, но проясняет смысл всей конструкции.

А как используются ACL? Метод `install_acl` приводит ACL в действие. Его необходимо вызывать перед обращением к `start_service`, иначе он не возымеет эффекта.

```
# Продолжение примера...
```

```
DRb.install_acl(acl)
DRb.start_service(nil, some_object)
# ...
```

Теперь, после запуска сервиса любой неавторизованный запрос на соединение приведет к исключению `RuntimeError`.

Это, конечно, не все, что можно сказать о библиотеке `drb`. Но для обзора вполне достаточно. В следующем разделе мы рассмотрим простой `drb`-сервер и `drb`-клиент, близкие к реальным программам. А затем поговорим о программах `Rinda` и `Ring`.

20.2. Пример: эмуляция биржевой ленты

В этом примере сервер публикует в сети биржевые котировки акций. К серверу может обратиться любой клиент, желающий узнать, сколько сейчас стоит его пакет.

Но мы добавили одну тонкость. Не желая следить за малейшими колебаниями цен, мы реализовали модуль `Observer`, который позволяет подписаться на информационный канал. Клиент следит за поступающими сведениями и предупреждает нас, когда изменение цены превысит заданный порог.

Сначала рассмотрим модуль `DrbObservable`. Это прямолинейная реализация паттерна `Observer` (Наблюдатель), описанного в замечательной книге Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влссидеса «Паттерны проектирования» (см. сноску в разделе 12.3.1). Еще этот паттерн называют «Издатель–Подписчик».

В листинге 20.1 наблюдатель определен как объект, отвечающий на вызов метода `update`. Сервер добавляет наблюдателей по их просьбе и посылает им уведомления, обращаясь к методу `notify_observers`.

Листинг 20.1. Модуль `DrbObservable`

```
module DRbObservable

  def add_observer(observer)
    @observer_peers ||= []
    unless observer.respond_to? :update
      raise NameError, "наблюдатель должен отвечать на вызов 'update'"
    end
    @observer_peers.push observer
  end

  def delete_observer(observer)
    @observer_peers.delete observer if defined? @observer_peers
  end

  def notify_observers(*arg)
    return unless defined? @observer_peers
    for i in @observer_peers.dup
      begin
        i.update(*arg)
      rescue
        delete_observer(i)
      end
    end
  end

end
```

Сервер (он же канал) в листинге 20.2 эмулирует биржевые котировки с помощью последовательности псевдослучайных чисел (простите мою иронию, но это очень точно соответствует характеру рынка). Символ, идентифицирующий компанию, – всего лишь косметическое украшение, никакого реального смысла

в этой программе он не имеет. При каждом изменении цены посылается уведомление всем наблюдателям.

Листинг 20.2. Канал биржевых котировок (drb-сервер)

```
require "drb"
require "drb_observer"

# Генерировать случайные котировки.
class MockPrice

  MIN = 75
  RANGE = 50

  def initialize(symbol)
    @price = RANGE / 2
  end

  def price
    @price += (rand() - 0.5)*RANGE
    if @price < 0
      @price = -@price
    elsif @price >= RANGE
      @price = 2*RANGE - @price
    end
    MIN + @price
  end
end

class Ticker # Периодически получать котировку акций.
  include DRBObservable

  def initialize(price_feed)
    @feed = price_feed
    Thread.new { run }
  end

  def run
    lastPrice = nil
    loop do
      price = @feed.price
      print "Текущая котировка: #{price}\n"
      if price != lastPrice
        lastPrice = price
        notify_observers(Time.now, price)
      end
      sleep 1
    end
  end
end
```

```

ticker = Ticker.new(MockPrice.new("MSFT"))

DRb.start_service('druby://localhost:9001', ticker)
puts 'Нажмите [return] для завершения.'
gets

```

На платформе Windows примененный способ завершения программы вызывает сложности. Функция `gets` в этом случае может блокировать главный поток. Если вы это видите, попробуйте вместо обращения к `gets` поставить `DRb.thread.join` (а завершайте программу нажатием `Ctrl+C`).

Неудивительно, что клиент (листинг 20.3) начинает с установления соединения с сервером. Он получает ссылку на объект показа котировок и устанавливает верхний и нижний пороги изменения цены. Затем клиент выводит сообщение пользователю всякий раз, как цена выходит за пределы указанного диапазона.

Листинг 20.3. Наблюдатель биржевых котировок (drb-клиент)

```

require "drb"

class Warner
  include DRbUndumped

  def initialize(ticker, limit)
    @limit = limit
    ticker.add_observer(self) # Любой объект Warner
                              # является наблюдателем.
  end
end

class WarnLow < Warner
  def update(time, price)      # Обратный вызов наблюдателя.
    if price < @limit
      print "--- #{time.to_s}: Цена ниже #{@limit}: #{price}\n"
    end
  end
end

class WarnHigh < Warner
  def update(time, price)      # Обратный вызов наблюдателя.
    if price > @limit
      print "+++ #{time.to_s}: Цена выше #{@limit}: #{price}\n"
    end
  end
end

DRb.start_service
ticker = DRbObject.new(nil, "druby://localhost:9001")

WarnLow.new(ticker, 90)
WarnHigh.new(ticker, 110)

```

```
puts 'Нажмите [return] для завершения.'
gets
```

Модуль `DRbUndumped` (см. листинге 20.3) следует включать в любой объект, который не нужно подвергать маршалингу. Самого присутствия этого модуля в числе предков объекта достаточно, чтобы `drb` не пытался применять к нему маршалинг. Вот исходный текст этого модуля целиком:

```
module DRbUndumped
  def _dump(dummy)
    raise TypeError, "can't dump"
  end
end
```

Приложение из этого раздела достаточно содержательно, и в то же время в нем легко разобраться. Есть и другие подходы к решению подобных задач. Но способ, показанный нами, демонстрирует простоту и элегантность распределенного Ruby.

20.3. Rinda: пространство кортежей в Ruby

Термин «пространство кортежей» появился в 1985 году, а сама идея еще старше. *Кортежем* называется массив или вектор, состоящий из элементов данных (как строка в таблице базы данных). *Пространство кортежей* – это большое объектное пространство, наполненное кортежами, нечто вроде «информационного супа».

Пока реализация пространства кортежей кажется ничем не примечательной. Но все становится гораздо интереснее, стоит лишь осознать, что к нему могут обращаться многие клиенты и доступ должен синхронизироваться. Короче говоря, это распределенная сущность; любой клиент может читать из пространства кортежей или писать в него, то есть его можно рассматривать как большое распределенное хранилище или даже способ коммуникации.

Первой реализацией пространства кортежей был проект Linda – исследование в области параллельного программирования, выполненное в Йельском университете в 1980-х годах. Реализация на языке Ruby (конечно, на основе библиотеки `drb`), естественно, называется Rinda.

Кортеж в Rinda может быть массивом или хэшем. На хэш налагается дополнительное ограничение: все ключи должны быть строками. Вот несколько примеров простых кортежей:

```
t1 = [:add, 5, 9]
t2 = [:name, :add_service, Adder.new, nil]
t3 = { 'type' => 'add', 'value_1' => 5, 'value_2' => 9 }
```

Элемент кортежа может быть произвольным объектом; это работает, потому что `drb` умеет выполнять маршалинг и демаршалинг объектов Ruby. (Конечно, необходимо либо включить модуль `DRbUndumped`, либо сделать определения объектов доступными серверу.)

Пространство объектов создается методом `new`:

```
require 'rinda/tuplespace'
```

```
ts = Rinda::TupleSpace.new  
# ...
```

Поэтому сервер выглядит так:

```
require 'rinda/tuplespace'
```

```
ts = Rinda::TupleSpace.new  
DRb.start_service("druby://somehost:9000", ts)  
gets # Нажать Enter для завершения сервера.
```

А клиент – так:

```
require 'rinda/tuplespace'
```

```
DRb.start_service  
ts = DRbObject.new(nil, "druby://somehost:9000")  
# ...
```

К пространству кортежей в Rinda применимы пять операций: `read`, `read_all`, `write`, `take` и `notify`.

Операция чтения `read` позволяет получить один кортеж. Но способ идентификации кортежа не вполне очевиден: необходимо задать кортеж, соответствующий искомому; при этом `nil` соответствует любому значению.

```
t1 = ts.read [:Sum,nil] # Может извлечь, например, [:Sum, 14].
```

Обычно операция `read` блокирует выполнение программы (для синхронизации). Чтобы быстро проверить существование кортежа, можно выполнить неблокирующее чтение, задав нулевой тайм-аут:

```
t2 = ts.read [:Result,nil],0 # Возбуждает исключение, если кортеж  
# не существует.
```

Если мы точно знаем или предполагаем, что образцу будет соответствовать не один, а несколько кортежей, можно воспользоваться методом `read_all`, который возвращает массив:

```
tuples = ts.read_all [:Foo, nil, nil]  
tuples.each do |t|  
  # ...  
end
```

Метод `read_all` не принимает второго параметра. Он всегда блокирует программу, если не найдено ни одного подходящего кортежа.

Операция `take` – это чтение, за которым следует удаление. Иными словами, метод `take` удаляет кортеж из пространства кортежей и возвращает его вызывающей программе:

```
t = ts.take [:Sum, nil] # Кортежа больше нет в пространстве кортежей.
```

Может возникнуть вопрос, почему не существует явного способа удаления. Надо полагать, что этой цели служит метод `take`.

Метод `write` помещает кортеж в пространство кортежей. Второй параметр показывает, сколько секунд кортеж может существовать, прежде чем система сочтет,

что срок его хранения истек. (По умолчанию его значение равно nil, то есть срок хранения не ограничен.)

```
ts.write [:Add, 5, 9]           # Хранить "вечно".
ts.write [:Foo, "Bar"], 10     # Хранить 10 секунд.
```

Здесь уместно будет сказать несколько слов о синхронизации. Предположим, что два клиента пытаются одновременно забрать (take) один и тот же кортеж. Одному это удастся, а другой будет заблокирован. Если первый клиент затем изменит кортеж и запишет (write) его обратно в хранилище, то второй получит модифицированную версию. Можно считать, что операция «обновления» – это последовательность take и write, которая не приводит к потере данных. Конечно, как и при любом варианте многопоточного программирования, нужно позаботиться о том, чтобы не возникали тупиковые ситуации.

Метод notify позволяет следить за пространством кортежей и получать уведомления, когда над интересующим вас кортежем была выполнена какая-то операция. Этот метод возвращает объект NotifyTemplateEntry и может наблюдать на операциями четырех видов:

- write;
- take;
- удаление (когда истекает срок хранения кортежа);
- закрытие (когда истекает срок хранения объекта NotifyTemplateEntry).

Поскольку операция чтения ничего не изменяет, то система не поддерживает уведомлений о чтениях. В листинге 20.4 приведен пример использования notify.

Листинг 20.4. Уведомление в системе Rinda

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new

alberts = ts.notify "write", ["Albert", nil]
martins = ts.notify "take", ["Martin", nil]

thr1 = Thread.new do
  alberts.each {|op,t| puts "#{op}: #{t.join(' ')}" }
end

thr2 = Thread.new do
  martins.each {|op,t| puts "#{op}: #{t.join(' ')}" }
end

sleep 1

ts.write ["Martin", "Luther"]
ts.write ["Albert", "Einstein"]
ts.write ["Martin", "Fowler"]
```

```
ts.write ["Albert", "Schweitzer"]
ts.write ["Martin", "Scorsese"]
ts.take ["Martin", "Luther"]
```

```
# Выводится:
# write: Albert Einstein
# write: Albert Schweitzer
# take: Martin Luther
```

Мы видели, что `read` и другие операции пользуются шаблонами для сопоставления с образцами (и этим напоминают регулярные выражения). Мы уже знаем, что `nil` выступает в роли метасимвола, но можно указать и класс; ему будет соответствовать любой экземпляр этого класса.

```
tem1 = ["X", Integer]      # Соответствует ["X",5], но не ["X","Files"].
tem2 = ["X", NilClass]     # Соответствует литералу nil в кортеже.
```

Кроме того, разрешается определять собственный оператор ветвящегося равенства (`==`), если вы хотите проводить сопоставление особым способом. В противном случае для сравнения будет использован стандартный оператор `==`.

Время жизни кортежа можно задать в момент записи. В сочетании с величинами тайм-аутов для различных операций над кортежами это позволяет ограничить время выполнения простых и более сложных манипуляций.

Тот факт, что у кортежа может быть конечный срок хранения, заодно означает, что по истечении этого срока кортеж можно обновить с помощью специально написанного объекта. В библиотеке имеется готовый класс `SimpleRenewer`, который каждые 180 секунд обращается к `drb`-серверу, создавшему кортеж. Если сервер не отвечает, то кортеж удаляется. Но не пытайтесь программировать обновление, пока не освоитесь с парадигмой пространства кортежей.

В листинге 20.5 приведен еще один пример работы с пространством кортежей. Он решает ту же задачу о производителе и потребителе, которая была рассмотрена в главе 13.

Листинг 20.5. Задача о производителе и потребителе

```
require 'rinda/tuplespace'

ts = Rinda::TupleSpace.new

producer = Thread.new do
  item = 0
  loop do
    sleep rand(0)
    puts "Производитель произвел #{item}"
    ts.write ["Item", item]
    item += 1
  end
end

consumer = Thread.new do
```

```

loop do
  sleep rand(0)
  tuple = ts.take ["Item", nil]
word, item = tuple
  puts "Потребитель потребил #{item}"
end
end

sleep 60    # Работать одну минуту, потом завершиться и завершить потоки.

```

20.4. Обнаружение сервисов в распределенном Ruby

Методика обнаружения сервисов может оказаться полезной, когда имеется много локально работающих сервисов, поскольку дает возможность находить сервис по имени. Если же число сервисов невелико и их местонахождение точно известно, особого смысла в автоматическом обнаружении нет.

Раз уж вы продолжили чтение, то, наверное, хотите знать, как работает механизм обнаружения сервисов. Такую возможность предоставляет библиотека `Rinda::Ring` (естественно, основанная на системе `Rinda`). В чем-то она похожа на службу `DNS`; это центральная служба регистрации, где хранится информация (в виде пространства кортежей) о `drb`-процессах. Сервисы `drb` могут по протоколу `UDP` найти ближайший сервер регистрации, объявить о своем присутствии или найти другие работающие поблизости сервисы.

Класс `Rinda::RingServer` реализует сервер регистрации. Он управляет пространством имен, в котором хранится информация о местонахождении других `drb`-сервисов. `RingServer` прослушивает сеть в ожидании широкоэвещательных `UDP`-пакетов с запросами о местонахождении сервера. В ответ на такой запрос он устанавливает соединение (посредством `drb`) с отправившим его сервисом. Пример:

```

require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service
Rinda::RingServer.new(Rinda::TupleSpace.new)

DRb.thread.join

```

Класс `Rinda::RingProvider` регистрирует сервис, объявляя о его присутствии серверу `RingServer`. При этом сообщается о типе сервиса и о фронтальном объекте, предоставляющем этот сервис, а также передается описательная информация. В следующем примере мы создаем простой сервис `Adder`, который складывает два числа, а потом объявляем о нем всему миру:

```

require 'rinda/ring'

class Adder
  include DRbUndumped

```

```

def add(val1, val2)
  return val1 + val2
end

add = Adder.new
DRb.start_service(nil, add)
Rinda::RingProvider.new(:add, add, 'Simple Adder')

DRb.thread.join

Класс Rinda::RingFinger (названный так по аналогии с имеющейся в UNIX
командой finger) применяется для обнаружения сервера RingServer. Он посылает
широковещательный UDP-пакет и ждет ответа от сервера. Затем RingFinger мож-
но использовать для поиска объявленных сервисов в пространстве кортежей.

require 'rinda/ring'

DRb.start_service
rs = Rinda::RingFinger.primary
list = [rs] + Rinda::Ringfinger.to_a
svc = list.find_all [:name, :add, nil, nil]

```

20.5. Заключение

Эта глава содержит введение в распределенный Ruby. Мы познакомились с тем, как сервис запускается и взаимодействует с клиентами, а также рассмотрели вопросы безопасности.

Мы выяснили, что система Rinda может выступать в роли простого распределенного хранилища объектов, обеспечивающего синхронизацию доступа. Наконец, было показано, как можно использовать библиотеку Rinda::Ring для обнаружения drb-сервисов.

На этом рассмотрение распределенного Ruby заканчивается. Переходим к следующей теме – инструментам разработки на языке Ruby, в частности программе Rake, оболочке irb, интегрированным средам разработки (IDE) и др.

Глава 21. Инструменты разработки для Ruby

.....
Человек – это животное, создающее инструменты.
Бенджамин Франклин

Среда разработки включает не только интерпретатор. У каждого хорошего разработчика под рукой всегда есть целый набор средств, облегчающих жизнь. Некоторые из них зависят от языка, другие – нет.

Самый важный инструмент – редактор. Основная работа программиста – манипулирование текстом, поэтому выбор редактора (и ваше умение им пользоваться) оказывает огромное влияние на производительность труда. Немаловажно также наличие в нем поддержки конкретного языка или средств настройки. Мы дадим очень краткий обзор распространенных редакторов.

Другие инструменты помогают в составлении документации, установке библиотек, отладке и т.д. Мы уже познакомились с библиотекой отладки (не являющейся автономным приложением) в главе 16, а в главе 17 рассказали о системе RDoc. Поэтому здесь мы их касаться не будем. Также в главе 17 шла речь о системе RubyGems с точки зрения разработчика, создающего пакет; теперь же мы станем на место программиста, который пользуется пакетами, созданными другими людьми.

Также в этой главе мы рассмотрим программы `irb` (интерактивный Ruby) и `ri` (инструмент для чтения документации). И завершим ее кратким обсуждением интегрированных сред разработки (IDE), которые хорошо работают с Ruby.

21.1. Система RubyGems

RubyGems – это не «официальная» система управления пакетами для Ruby, но одна из тех, что пользуются обширной поддержкой со стороны сообщества. В июле 2006 года она еще не входила в стандартный дистрибутив, но может стать его частью в будущем. Мы довольно подробно рассматривали процедуру создания gem-пакета в главе 17. А в этой главе расскажем, как пользоваться пакетами, созданными другими людьми.

RubyGems пытается решить несколько основных проблем, как то: простота установки, центральный репозиторий, управление версиями библиотек, управление зависимостями и т.д. Gem-пакеты предоставляют доступ к документации и позволяют легко контролировать, какие библиотеки установлены.

Если система RubyGems на вашем компьютере не установлена, зайдите на страницу <http://rubyforge.org/projects/rubygems> и выполните простые инструкции по загрузке и установке. В дальнейшем RubyGems будет обновляться автоматически.

Сам исполняемый файл называется `gem`. Структура команд похожа на применяемую в `cvs` и других аналогичных системах, то есть у каждой команды есть подкоманды, а у каждой подкоманды – специфичные для нее параметры. Ниже приведена информация о порядке запуска*:

RubyGems – развитый менеджер пакетов для Ruby. Ниже приведена краткая справка с указанием на источники дополнительной информации.

Порядок запуска:

```
gem -h/--help
gem -v/--version
gem command [аргументы...] [флаги...]
```

Примеры:

```
gem install rake
gem list --local
gem build package.gemspec
gem help install
```

Более подробные справки:

```
gem help commands    показать все команды 'gem'
gem help examples    показать примеры использования
gem help COMMAND     вывести справку о команде COMMAND
(например, 'gem help install')
```

Дополнительная информация:

<http://rubygems.rubyforge.org>

Самые важные команды не вызывают затруднений. Например, чтобы установить библиотеку `feedtools`, нужно выполнить команду:

```
gem install feedtools
```

Сначала она будет искать `gem`-пакет локально, а если не найдет, то обратится к серверу RubyForge. По умолчанию устанавливается последняя версия (хотя в некоторых случаях вам задают вопрос, какой именно пакет вы хотите установить). Можно и явно указать версию с помощью флага `-v` или `--version`. Есть еще несколько флагов, управляющих установкой; подробную информацию дает команда `gem help install`.

Иногда `gem`-пакет зависит от других пакетов. В таком случае вас спросят, надо ли их устанавливать. Понятно, что, отказавшись от требуемых пакетов, вы не сможете установить и тот, что хотели изначально.

Как узнать имя нужного вам пакета? Если он находится на сайте RubyForge, то обычно называется так же, как проект. Например, программа `rake` находится на

* Программа печатает эту информацию на английском языке. Для удобства читателей приводим русский перевод. – *Прим. перев.*

странице <http://rubyforge.org/projects/rake> или просто <http://rake.rubyforge.org>, поэтому для установки нужно выполнить команду `gem install rake`.

Если вы не располагаете даже такой информацией, попробуйте воспользоваться командой `gem search`. Если указать часть предполагаемого имени `gem`-пакета, то будут найдены все пакеты с похожими именами. По умолчанию поиск производится на локальном компьютере; чтобы искать в центральном репозитории, задайте флаг `--remote`. Например, команда `gem search xml --remote` нашла 12 пакетов.

Чтобы получить информацию об установленном пакете, введите команду `gem specification` – она напечатает все содержащиеся в пакете метаданные (в основном содержимое самого файла `gemspec`). Флаг `--remote` существует, но пока не реализован.

Кроме того, имеется команда деинсталляции `uninstall`, а также команды `query` и `list`, отчасти перекрывающие друг друга и команду `search`. Есть и еще много команд (дополнительную информацию вы найдете во встроенной справке или в онлайн-овой документации).

Полезно установить `gem`-сервер и на своей системе – не в качестве репозитория, из которого другие смогут дистанционно устанавливать пакеты, а как централизованный механизм, позволяющий просматривать в браузере все локально установленные пакеты вместе с документацией.

Для запуска `gem`-сервера просто наберите команду `gem_server` (обычно сервер работает в фоновом режиме). Указав в адресной строке браузера URL `localhost:8808`, вы увидите страницу «RubyGems Documentation Index», где в алфавитном порядке перечислены все установленные `gem`-пакеты со ссылками на документацию в формате `rdoc` и на домашнюю страницу проекта в Web.

О системе RubyGems можно еще многое сказать; к тому же она постоянно развивается. Самая актуальная информация представлена на сайте <http://rubygems.rubyforge.org>.

21.2. Программа Rake

Утилита `rake` – это вариация на тему давно известной в UNIX программы `make`. Но вместо «странного» синтаксиса `make`, который все мы знаем и терпеть не можем, в `rake` используется код на самом Ruby. Программу написал Джим Вайрих (Jim Weirich); это первый, насколько мне известно, пример формальной реализации языка DSL (domain-specific language – язык предметной области) на Ruby.

Есть два варианта написания названия: `Rake` и `rake`. Первый – это название инструмента, второй – имя самого исполняемого файла. На мой взгляд, различие несущественное.

Нет сомнения, что на дизайн `Rake` оказала большое влияние программа `make`, так что терминология одна и та же. Как и раньше, мы говорим о целях, действиях, зависимостях и правилах.

Применениям `Rake` нет числа. Вы можете пользоваться ею для сборки проектов, написанных на C, C++ или Java (или на любом другом языке). Годится она и

для генерирования документации с помощью RDoc, развертывания программ, обновления проекта на сайте RubyForge и множества других задач.

Неудивительно, что на вход Rake подается командный файл (rake-файл), который по умолчанию называется `rakefile` или `Rakefile`. Если вы хотите назвать его иначе, укажите имя с помощью флага `-f` или `--rakefile`:

```
$ rake           # Искать сначала 'rakefile', потом 'Rakefile'.
$ rake -f myfile  # Использовать файл с именем 'myfile'.
```

Основная «единица работы» в Rake – *задание*; именуется задание посредством символов Ruby. Предполагается, что в каждом rake-файле есть задание по умолчанию `:default` – оно будет выполняться, если явно не указано другое имя.

```
$ rake           # Выполнить задание по умолчанию.
$ rake mytask    # Выполнить задание с именем 'mytask'.
```

Внутри rake-файла мы указываем задания с помощью метода `task`, передавая символ и блок:

```
task :mytask do
  # ...
end
```

Содержимое блока в примере выше опущено. То, что в нем находится, называется *действием*.

В действии можно выполнять произвольный код на Ruby. Для типичных операций предусмотрены готовые методы. Так, метод `sh` (название которого напоминает об интерпретаторе команд в UNIX) запускает системную команду.

Методы `cp`, `mv` и `rm` предназначены соответственно для копирования, перемещения и удаления файлов. (Как и `make`, Rake беззастенчиво выдает свое происхождение от UNIX.) Есть и другие подобные команды; дополнительную информацию вы найдете в онлайн-овой документации на сайте <http://docs.rubyrake.org>.

При желании можете заключать блок в фигурные скобки, но обычно в этом случае интерпретатор Ruby ожидает, что параметры будут заключены в круглые скобки.

```
task(:mytask) { do_something }
```

Рассмотрим более конкретный пример. Предположим, что имеется программа `myprog.c`, написанная на C, и еще два связанных с ней исходных файла (каждый со своим заголовочным файлом). Иными словами, у нас есть пять исходных файлов:

```
myprog.c
sub1.c
sub1.h
sub2.c
sub2.h
```

Мы хотим собрать из них исполняемый файл `myprog`. Эта процедура состоит из нескольких шагов: откомпилировать все файлы с расширением `.c`, а затем скомпоновать получившиеся в результате файлы с расширением `.o`.

Начнем с метода `file`, который определяет зависимости файлов:

```
file "myprog.o" => ["myprog.c"]
file "sub1.o" => ["sub1.c", "sub1.h"]
file "sub2.o" => ["sub2.c", "sub2.h"]
file "myprog" => ["sub1.o", "sub2.o"]
```

Отметим, что метод `file` принимает хэш, который ассоциирует имя файла с массивом имен файлов, от которых он зависит.

Теперь перейдем к построению двоичных файлов. Расширим написанный выше код. Если после вызова метода `file` указать блок, то мы сможем ассоциировать с файлом набор действий, которые необходимо выполнить для создания этого файла:

```
file "myprog.o" => ["myprog.c"] do
  sh "cc -c -o myprog.o myprog.c"
end

file "sub1.o" => ["sub1.c", "sub1.h"] do
  sh "cc -c -o sub1.o sub1.c"
end

file "sub2.o" => ["sub2.c", "sub2.h"] do
  sh "cc -c -o sub2.o sub2.c"
end

file "myprog" => ["sub1.o", "sub2.o"] do
  sh "cc -o myprog myprog.o sub1.o sub2.o"
end
```

Здесь имеется некоторое дублирование, но от него можно избавиться. В Rake есть специальный механизм, который называется `FileList`; он понимает метасимволы (шаблоны) и позволяет работать сразу с несколькими файлами. В данном случае поместим все файлы с расширением `.c` в список `SRC`. Константа типа `FileList` ведет себя как массив:

```
SRC = FileList["*.c"]
```

Теперь можно определить действия в цикле, как показано ниже. И обратите внимание, что зависимости здесь не упоминаются – Rake самостоятельно учтет эту информацию, если она была задана в другом месте.

```
SRC.each do |src|
  obj = src.sub(/.c$/, ".o")
  file(obj) { sh "cc -c -o #{obj} #{src}" }
end
```

Однако проще пользоваться правилами. Это еще один механизм Rake, естественно, позаимствованный у `make`:

```
rule '.o' => '.c' do |target|
  sh "cc -c -o #{target.name} #{target.source}"
end
```

Тут мы наблюдаем небольшое волшебство. Rake устанавливает атрибут `source`, подставляя расширение имени файла из хэша (в данном случае `.o` заменяется на `.c`).

Продолжим сеанс магии. Если затребовать библиотеку `rake/clean`, то появятся константы `CLEAN` и `CLOBBER` (первоначально пустые) и задания `:clean` и `:clobber`. По сложившейся традиции `clean` удаляет временные файлы, а `clobber` — еще и собранный исполняемый файл.

Для этих напоминающих массив констант определен метод `include`, принимающий маску имени файлов; это неявное использование механизма `FileList`.

Теперь наш `rake`-файл принимает такой вид:

```
require 'rake/clean'

CLEAN.include("**.o")
CLOBBER.include("myprog")

SRC = FileList['*.c']
OBJ = SRC.ext('o')

rule '.o' => '.c' do |t|
  sh "cc -c -o #{t.name} #{t.source}"
end

file "hello" => OBJ do
  sh "cc -o hello #{OBJ}"
end

file "myprog.o" => ["myprog.c"]
file "sub1.o" => ["sub1.c", "sub1.h"]
file "sub2.o" => ["sub2.c", "sub2.h"]

task :default => ["myprog"]
```

Обратите внимание, что мы не задавали задания «`clean`» и «`clobber`» явно. Кроме того, отметим что «`clobber`» неявно включает операцию «`clean`». И наконец, мы определили задание `default` для удобства запуска `rake`-файла; теперь можно не указывать явно имя задания, осуществляющего компиляцию и сборку.

У программы `rake` есть несколько параметров командной строки. Иногда желательнее протестировать `rake`-файл, не выполняя никаких (потенциально опасных) операций; для этого служит флаг `-n` или `--dry-run`. Флаг `-T` выводит список всех целей в `rake`-файле. Имеются также флаги, управляющие поиском библиотек, трассировкой, протоколированием и т. д.

Программа `Rake` сложнее, чем я описал в этом разделе (особенно это касается правил). И она продолжает развиваться. Как обычно, самую свежую информацию ищите в онлайн-овой документации (<http://docs.rubyrake.org>).

21.3. Оболочка irb

Утилита `irb` (интерактивный Ruby) уже много лет как включена в дистрибутив Ruby. Можно считать, что это «испытательный стенд» или «площадка для игр», на которой вы опробуете новые приемы и идеи.

Работать с `irb` просто. После запуска вы получаете приглашение и можете вводить произвольные выражения Ruby. Выражение вычисляется, и печатается результат. Вот пример короткого сеанса:

```
$ irb
irb(main):001:0> "cell" + "o"
=> "cello"
irb(main):002:0> 3*79
=> 237
irb(main):003:0> Dir.entries(".").size
=> 17
irb(main):004:0> rand
=> 0.850757389880155
irb(main):005:0> rand
=> 0.679879756672551
irb(main):006:0> defined? foo
=> nil
irb(main):007:0> defined? Object
=> "constant"
irb(main):008:0> quit
$
```

Конечно, это больше, чем калькулятор. При желании можно ввести произвольный код на Ruby:

```
[hal@localhost ch21]$ irb
irb(main):001:0> require 'mathn'
=> true
irb(main):002:0> gen = Prime.new
=> #
```

При установке флага `-r` выполняется `require`, поэтому можно включать код из внешнего файла. Предположим, что вы написали такой файл:

```
# File: foo.rb
class MyClass

  attr_accessor :alpha, :beta

  def initialize(a, b)
    @alpha, @beta = a, b
  end

end

obj1 = MyClass.new(23,34)
obj2 = MyClass.new("abc","xyz")
```

Тогда можно сделать следующее:

```
$ irb -rfoo
irb(main):001:0> obj = MyClass.new(88,99)
=> #
```

Отметим, что хотя обращаться к сущностям, определенным в файле (например, к константе `myClass`) мы можем, это не относится к локальным переменным. Локальные переменные доступны только в самом файле, `require` (выполненное хоть внутри, хоть вне `irb`) доступа к ним не дает.

Новичков часто смущает информация, выводимая `irb`:

```
$ irb -rfoo
irb(main):001:0> puts "hello"
hello
=> nil
```

Позвольте, что тут делает `nil`? Разумеется, это значение, возвращенное методом `puts`.

Еще один источник недоразумений – метод `eval`. Взгляните на следующий сеанс:

```
$ irb
irb(main):001:0> eval("var = 567")
=> 567
irb(main):002:0> eval("var")
=> 567
irb(main):003:0> var
=> 567
```

Вас ничего не удивляет? Но давайте запустим следующий сценарий и посмотрим, что произойдет:

```
p eval("var = 567")
p eval("var")
p var

# Results:
# 567
# 567
# temp.rb:3: undefined local variable or method 'var' for main:Object
# (NameError)
```

У Ruby есть такая особенность: когда вы выполняете `eval`, а вслед за ним еще один, они в некотором смысле разделяют «общую область видимости». Поэтому к переменной, определенной в первой строке, можно обратиться во второй (вне или внутри `irb`). Но различие проявляется, когда мы попытаемся обратиться к той же переменной без использования `eval`. В `irb` это работает, а в сценарии мы получаем ошибку. Что происходит?

Поведение сценария следует считать более правильным. Не забывайте, что сама программа `irb` написана на Ruby; здравый смысл подсказывает, что она, скорее всего, внутри вызывает `eval`. Но мы только что убедились, что результат применения `eval` может отличаться от того, что мы видим на верхнем уровне, поэтому исполнение кода внутри `irb` не всегда идентично исполнению такого же кода в сценарии. Об этом не следует забывать, особенно если вы ставите какие-нибудь экзотические эксперименты.

Имейте в виду, что `irb` настраивается в широких пределах. При запуске она читает все инициализационные файлы, которые может найти, в следующем порядке:

- файл `~/.irbrc`;
- файл `.irbrc`;
- файл `irb.rc`;
- файл `_irbrc`;
- путь, указанный в переменной окружения `$irbrc`.

Инициализационные файлы пишутся на Ruby. В них можно настраивать текст приглашения и многое другое. Подробно эти файлы обсуждаются в книге «Programming Ruby». Ниже мы сделаем лишь несколько замечаний.

Если ваша версия Ruby собрана с поддержкой библиотеки GNU readline (обычно так и есть), то вы можете перемещаться по истории команд с помощью клавиш со стрелками вверх и вниз. Еще важнее, что работает механизм завершения по клавише Tab: когда вы набираете начало идентификатора, а затем нажимаете Tab, `irb` пытается дописать имя идентификатора за вас.

Чтобы включить механизм завершения, добавьте в файл `.irbrc` следующий фрагмент:

```
IRB.conf[:AUTO_INDENT] = true
IRB.conf[:USE_READLINE] = true
IRB.conf[:LOAD_MODULES] ||= {}
IRB.conf[:LOAD_MODULES] |= ['irb/completion']
```

В файле `.irbrc` может содержаться произвольный код. Например, я часто пользуюсь описанным ниже методом. Для краткости он назван `sm` (сокращение от «show methods»), а цель – вывести (в алфавитном порядке) список всех методов, которые можно вызывать для данного объекта, за исключением тех, которые он унаследовал от своих предков:

```
def sm(obj)
  list = obj.methods
  anc = obj.class.ancestors - [obj.class]
  anc.each {|a| list -= a.instance_methods }
  list.sort
end
```

Вот пример его использования:

```
irb(main):001:0> str = "hello"
=> "hello"
irb(main):002:0> sm str
=> ["%", "!", "+", "<<", "<=>", "[]", "[]=", "capitalize",
"capitalize!", "casecmp", "center", "chomp", "chomp!", "chop", "chop!",
"concat", "count", "crypt", "delete", "delete!", "downcase", "downcase!",
"dump", "each", "each_byte", "each_line", "empty?", "gsub", "gsub!", "hex",
"index", "insert", "intern", "length", "ljust", "lstrip", "lstrip!", "match",
"next", "next!", "oct", "replace", "reverse", "reverse!", "rindex", "rjust",
"rstrip", "rstrip!", "scan", "size", "slice", "slice!", "split", "squeeze"
```

```
"squeeze!", "strip", "strip!", "sub", "sub!", "succ", "succ!", "sum",
"swapcase", "swapcase!", "to_f", "to_i", "to_str", "to_sym", "tr", "tr!",
"tr_s", "tr_s!", "unpack", "upcase", "upcase!", "upto"]
irb(main):003:0> sm String
=> ["allocate", "new", "superclass"]
irb(main):004:0> sm 123
=> ["%", "&", "*", "**", "+", "-", "/", "<<", ">>", "[]", "^",
"id2name", "power!", "rdiv", "rpower", "size", "to_f", "to_sym", "|", "-"]
```

Программа irb позволяет запускать подсеансы внутри сеанса, хотя это используется и нечасто. Можно запустить несколько сеансов и переключаться между ними, у каждого сеанса собственный набор привязок.

Может быть, вам это и не кажется полезным, но можно, например, задать объект в сочетании с подкомандой irb. Тогда контекстом подсеанса станет этот объект, псевдопеременная self будет ссылаться на него, он же станет областью видимости и т.д.:

```
$ irb
irb(main):001:0> t0 = Time.now
=> Mon Jul 31 04:51:50 CDT 2006
irb(main):002:0> irb t0
irb#1(Mon Jul 31 04:51:50 CDT 2006):001:0> strftime("%a %b %c")
=> "Mon Jul Mon Jul 31 04:51:50 2006"
irb#1(Mon Jul 31 04:51:50 CDT 2006):002:0> to_i
=> 1154339510
irb#1(Mon Jul 31 04:51:50 CDT 2006):003:0> self + 1000
=> Mon Jul 31 05:08:30 CDT 2006
irb#1(Mon Jul 31 04:51:50 CDT 2006):004:0> wday
=> 1
irb#1(Mon Jul 31 04:51:50 CDT 2006):005:0> class
SyntaxError: compile error
(irb#1):5: syntax error, unexpected $end
      from (irb#1):5
irb#1(Mon Jul 31 04:51:50 CDT 2006):006:0> self.class
=> Time
irb#1(Mon Jul 31 04:51:50 CDT 2006):007:0> quit
=> #<IRB::Irb: @scanner=#<RubyLex:0xb7ee8394>,
@signal_status=:IN_EVAL, @context=#<IRB::Context:0xb7ee86f0>>
irb(main):003:0> quit
$
```

Мы уже убедились в полезности библиотеки ruby-breakpoint (см. главу 16). В сочетании с ней irb становится мощным средством отладки, поскольку вы можете установить точку прерывания и «оказаться» в сеансе irb. Конечно, это не настоящий отладчик, потому что не позволяет исполнять код в пошаговом режиме.

Иногда бывает полезна библиотека xmp. Она принимает предложения на Ruby, исполняет их и помещает возвращаемое значение в комментарий. В книге «Programming Ruby» рассматривается xmp, а также библиотека rtags (которая генерирует файл TAGS для редакторов emacs или vi).

У `irb` есть еще одна приятная особенность, о которой стоит знать. Понятно, что `irb` умеет анализировать написанный на Ruby код, но лексический анализатор не сложно использовать и в других приложениях. Вот простой пример программы, которая открывает саму себя и анализирует собственный код, выводя отсортированный список всех идентификаторов и констант:

```
require 'irb/ruby-lex'

file = File.new(__FILE__)

parse = RubyLex.new # (file)
parse.set_input(file)

idents = []

loop do
  token = parse.token
  break if token.nil?
  if token.is_a? RubyToken::TkIDENTIFIER or
     token.is_a? RubyToken::TkCONSTANT
    idents << token.name
  end
end

p idents.uniq.sort

# Выводится:
# ["File", "RubyLex", "RubyToken", "TkCONSTANT", "TkIDENTIFIER", "file",
#  "idents", "loop", "name", "new", "p", "parse", "require", "set_input",
#  "sort", "token", "uniq"]
```

Насколько мне известно, этот механизм не документирован на английском языке. Но если вам необходим лексический анализатор Ruby, то можете изучить исходный текст и адаптировать его под свои нужды.

21.4. Утилита `ri`

Сокращение `ri`, вероятно, означает «Ruby index» или нечто подобное. Это командная утилита для чтения документации, предоставляющая информацию о классах, методах, модулях и т.д. Пример:

```
$ ri each_with_index
-----
enumObj.each_with_index {|obj, i| block } -> nil
-----
```

Calls block with two arguments, the item and its index, for each item in `enumObj`.

```
hash = Hash.new
```

```
%w(cat dog wombat).each_with_index {|item, index|
  hash[item] = index
}
hash    #=> {"dog"=>1, "wombat"=>2, "cat"=>0}
```

Отметим, что в ней есть ошибки и странности. Можете сообщать о них (а равно об опечатках и других неточностях), если найдете человека, который сознается в авторстве.

Мартин Анкерл (Martin Ankerl) написал графическую утилиту `fxri`, которая работает устойчиво. Она получает данные от RDoc, как и `ri`. Кроме того, в ней есть панель, где запущена `irb`.

21.5. Поддержка со стороны редакторов

Любой современный редактор должен проявлять дружелюбие по отношению к программистам. Память о двадцатом веке уходит в прошлое, и мы принимаем как должное, что редакторы изменяют свое поведение в зависимости от типа редактируемого файла. Мы ожидаем синтаксической подсветки, автоматического формирования отступов и прочего в том же духе.

Программист, пишущий на Ruby, не будет разочарован имеющимся инструментарием. Многие возможности стандартно включены в редактор, другие приходится настраивать.

Два прекрасных редактора – это SciTe (на базе виджета редактирования Scintilla) и TextMate (имеется только для Mac OS X). Оба хорошо поддерживают Ruby, но больше я о них ничего сказать не могу.

Третий редактор – jEdit, написанный на Java редактор для программистов (www.jedit.com). Роб МакКиннон дополнил его модулем для Ruby (<http://rubyjedit.org/>). Но в этой книге я не могу задерживаться на его описании.

Среди программистов наиболее распространены редакторы `vi` (или `vim`) и `emacs`. Их мы кратко рассмотрим.

Исторически для пользователей `vim` было создано три разных пакета. К счастью, теперь они объединены в пакет `vim-ruby`. Это набор конфигурационных файлов, поддерживающий синтаксическую подсветку, автоматическое формирование отступов и завершение кода. Кроме того, он позволяет вызывать интерпретатор Ruby прямо из редактора.

Если вы работаете с `vim` версии 7 (как и положено), то этот пакет, вероятно, уже установлен. (Если же по какой-то неуважительной причине вы все еще используете версию 6.x, то самое время провести обновление.)

Но может оказаться, что пакет не активирован (возможно, в целях обратной совместимости). Тогда нужно его активировать, добавив в файл `.vimrc` следующие строки:

```
set nocompatible
syntax on
filetype on
filetype indent on
filetype plugin on
```


Есть также по меньшей мере две реализации сворачивания кода для Ruby. На мой взгляд, любая методика сворачивания, которая требует вставлять в код специальные комментарии или символы, не оптимальна. Хороший механизм сворачивания должен анализировать код и распознавать в нем классы, модули, методы и т.д. Такую реализацию для vim написал Маурицио Фернандес ([http://eigenclass.org/hiki.rb? Usable+Ruby+folding+for+Vim](http://eigenclass.org/hiki.rb?Usable+Ruby+folding+for+Vim)).

В принципе есть много причин не любить редактор vim; одна из самых основных – язык vimscript. Хорошая новость – теперь есть возможность программировать поведение vim на Ruby! И плохая... она неважно документирована. Если хотите узнать подробнее, рекомендую для начала набрать команду `:help ruby` в vim. Дополнительная информация имеется на странице <http://wiki.rubygarden.org/Ruby/page/show/VimRubyInterface>. Наконец, загляните на сайт <http://vim.org>; может быть, там появилось что-то новенькое.

Еще один популярнейший в мире редактор – emacs. Вообще-то называть его редактором не вполне правильно. Это, скорее, миниатюрная операционная система, которая заодно позволяет и редактировать тексты. Одно из главных достоинств emacs – поразительная расширяемость; пользователь программирует его поведение на диалекте языка Lisp (elisp). Язык elisp мощнее, чем vimscript, но читать написанные на нем программы не менее сложно (правда, по другой причине).

Я сам не работаю с emacs, но очень уважаю тех, кто им пользуется. Не могу почти ничего сказать о поддержке Ruby в emacs; отсылаю вас на wiki-страницу RubyGarden (<http://wiki.rubygarden.org/Ruby/page/show/EmacsExtensions>).

21.6. Интегрированные среды разработки

Можно спорить с тем, что компания Borland со своим языком Turbo Pascal (в середине 1980-х годов) стала праотцом всех современных интегрированных сред разработки (IDE), но без сомнения Borland сделала идею популярной. Кто бы ни стоял у ее истоков, идея IDE в обозримом будущем никуда не денется.

В центре IDE обычно стоит мощный редактор, знающий обо всех особенностях синтаксиса языка. Такие средства, как синтаксическая подсветка и автоматическое формирование отступов, стали стандартными. Обычно включают также отладчик и средства управления проектом. Все чаще в среду интегрируются механизмы тестирования и сопряжение с системой управления версиями.

Трудно проводить осмысленное сравнение IDE. Все они похожи и все в чем-то различаются. Выбор такой же личный и субъективный, как при покупке автомобиля. Я попытался провести небольшое исследование, чтобы помочь вам принять решение:

- Одна из самых важных для Ruby сред – FreeRIDE. Она «заточена» специально под Ruby и всецело контролируется разработчиками, входящими в сообщество пользователей этого языка. В число основных авторов входят Рич Килмер (Rich Kilmer), Курт Гиббс (Curt Hibbs) и Лоран Жуллиар (Laurent Julliard), но есть и много других. Как и в случае любого крупного проекта с открытыми исходными текстами, разработка ведется медленно. Получить

самую свежую информацию и загрузить программу можно на сайте <http://freeride.rubyforge.org>.

- Одно из самых последних решений – программа Komodo производства компании ActiveState. Это мощная, полнофункциональная среда, но имейте в виду, что продукт коммерческий. Полную информацию можно найти на странице <http://www.activestate.com/Products/Komodo/>.
- Если вы поклонник Eclipse, то должны знать о существовании Ruby Development Tool (RDT). Это набор подключаемых модулей для поддержки Ruby на платформе Eclipse. Детали изложены на странице <http://sourceforge.net/projects/rubyecclipse>.
- ArachnoRuby – еще один коммерческий продукт; автор Лотар Шольц (Lothar Scholz). Набор функций достаточно полон, но в июле 2006 года он работал еще не так устойчиво, как другие IDE. Дополнительную информацию ищите на странице http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php.
- Наконец, есть еще RDE, Ruby Development Environment. Работает неплохо, достаточно мощная и бесплатная. Но только на платформе Windows.

В таблице 21.1 перечислены характеристики различных IDE.

Таблица 21.1. Сравнение пяти IDE для Ruby

Характеристика	Free-RIDE	RDE	RDT	Arachno-Ruby	Komodo
Коммерческий	нет	нет	нет	да	да
Платформы	все	Win32	все	Linux, Win32	все
Синтаксическая подсветка	да	да	да	да	да
Автоматическое формирование отступов	да	да	да	да	да
Сворачивание кода	да	нет	нет	да	да
Редактирование нескольких документов	да	да	да	да	да
Поддержка других языков (Perl и т.д.)	ограниченная	да	да	да	да
Настраиваемые привязки клавиш	да	нет	нет	да	да
Макросы редактора	да	нет	нет	да	да
Обзор кода	да	да	да	да	да
Интеграция с системой управления версиями	нет	нет	да	нет	да
Построитель графических интерфейсов	нет	нет	нет	нет	да
Управление проектом	да	нет	да	да	да
Интегрированное тестирование	да	нет	да	нет	нет
Прочие замечания	чистый Ruby	–	–	включены инструменты для Web	построен на базе Mozilla

Естественно, это лишь отправная точка для ваших собственных изысканий. Для настоящего сравнения потребовалась бы сотня страниц (а к моменту завершения работы данные уже устарели бы).

В таблице рассматриваются только три платформы: семейство Linux/UNIX, семейство Win32 и Mac OS X. Когда мы пишем «все», имеются в виду только эти платформы.

21.7. Заключение

В этой главе мы рассмотрели ряд инструментов, облегчающих жизнь программисту на Ruby. К числу общеупотребительных относятся интерактивная оболочка для Ruby `irb`, инсталлятор `RubyGems` и утилита чтения документации `ri`. Мы также дали обзор модулей для поддержки Ruby, которые подключаются к редакторам. Наконец, мы вкратце упомянули некоторые IDE с поддержкой Ruby. Но самый главный инструмент — это не программы, а люди, то есть все сообщество пользователей Ruby. Это и есть тема следующей (и последней) главы.

Глава 22. Сообщество пользователей Ruby

.....
*...Тот, кто не способен вступить в общение или, считая себя
существом самодовлеющим, не чувствует потребности ни в чем,
уже не составляет элемента государства, становясь
либо животным, либо божеством.
Аристотель, «Политика»**

Уже было сказано, что одна из лучших черт Ruby – сообщество его пользователей. Конечно, это дело вкуса, вы составите собственное мнение. Моя цель в настоящей главе – представить вашему вниманию некоторые «посиделки»: источники новостей и новых знаний, форумы как в сети, так и вне нее, а также места, где «рубисты» встречаются в киберпространстве и в реальной жизни.

Эта глава намеренно сделана очень краткой. Значительная часть приведенной в ней информации стабильна, но ведь в жизни все меняется. Если сомневаетесь, воспользуйтесь поисковой машиной.

22.1. Ресурсы в Web

Основной сайт Ruby – www.ruby-lang.org; все остальные находятся от него в нескольких щелчках мышью. Отсюда всегда можно скачать самую свежую версию Ruby.

Еще один важный сайт – rubygarden.org, который раньше содержал только wiki-страницу. Wiki как была, так и осталась, но теперь имеется еще модерируемый раздел, в котором много полезных статей и учебных руководств.

Если вас интересует документация, не пропустите сайт ruby-doc.org, который поддерживает Джеймс Бритт. На нем собраны документы, сгенерированные программой `gdoc` для системных и стандартных библиотек, а также много других полезных материалов.

RubyCentral – некоммерческая организация, занимающаяся пропагандой Ruby (rubycentral.org). Среди прочего, эта группа ежегодно проводит международную конференцию по Ruby. Граждане США могут вносить на ее счет пожертвования, не облагаемые налогом.

Сайт rubycentral.com не имеет к предыдущему никакого отношения. Это еще один прекрасный ресурс, один из первых в Англии, созданный группой «Pragmatic Programmers».

* Пер. С. Жебелева. – Прим. ред.

22.2. Новостные группы и списки рассылки

Список рассылки `ruby-talk` – наверное, старейший англоязычный форум для программистов на Ruby. Новостная группа (конференция) `Usenet comp.lang.ruby` появилась немного позже, в мае 2000 года.

Почти сразу после организации новостной группы Дэйвом Томасом (см. ниже) было создано ее «зеркало» для пересылки сообщений между самой группой и списком рассылки `ruby-talk`. Если не считать редких технических проблем, сообщения в них должны быть идентичны.

В списке часто задаваемых вопросов (FAQ) для группы `comp.lang.ruby` обсуждаются вопросы сетевого этикета и излагаются цели создания группы. Он обновляется ежемесячно и хранится также на сайте rubyhacker.com.

Для списка `ruby-talk` (а также для связанных с ним списков `ruby-core` и `ruby-math`) на сайте ruby-talk.org можно производить поиск.

Есть также множество списков рассылки, относящихся к конкретным проектам, – слишком много, чтобы перечислять их здесь. Почти для каждого крупного проекта, относящегося к Ruby, в архиве RAA или на сайте rubyforge.org имеется свой список. Кстати, эти два сайта – лучшие отправные точки для поиска информации о списке рассылки или о домашней странице проекта.

22.3. Блоги и онлайн-журналы

Я полагаю, что тенденция создания онлайн-дневников (блогов) только будет нарастать, и не вижу признаков того, что они выйдут из моды в ближайшее время.

Существует много блогов, имеющих отношение к Ruby. Поисковая машина отыщет их все, а я приведу ссылки на блоги нескольких ведущих «рубистов».

- Дэйв Томас (Dave Thomas): <http://blogs.pragprog.com/cgi-bin/pragdave.cgi>;
- Чэд Фаулер (Chad Fowler): <http://chadfowler.com/>;
- Джим Вайрих (Jim Weirich): <http://onestepback.org/>;
- Джеймис Бак (Jamis Buck): <http://jamis.jamisbuck.org/>;
- Натаниэль Талботт (Nathaniel Talbott): <http://blog.talbott.ws/>;
- *why the lucky stiff*: <http://redhanded.hobix.com>.

Несколько централизованных сайтов содержат ссылки на отдельные блоги или агрегируют их контент. Они появились сравнительно недавно, поэтому я не буду давать на них ссылки; не исключено, что, когда вы будете читать эту главу, перечень уже изменится.

Языку Ruby посвящены по меньшей мере два онлайн-журнала. Самый старый – The Rubyist (<http://jp.rubyist.net>) – выходит на японском языке и основан на публикациях в wiki-страницах. Неплохой стимул к изучению японского!..

На сайте Artima (artima.com) размещается отличный журнал Ruby Code & Style (<http://www.artima.com/ruby/cs>); он подает большие надежды. Ходят слухи о создании новых онлайн-журналов, но публиковать информацию о них пока преждевременно.

22.4. Запросы на изменение Ruby

Язык Ruby продолжает развиваться. К достоинствам языка следует отнести и то, что изменяется он медленно и обдуманно.

Но, конечно, Ruby не достиг совершенства. Поэтому имеется механизм подачи предложений об изменении синтаксиса, системных классов и стандартных библиотек. Процедура называется «запрос на изменение Ruby» (Ruby Change Request – RCR).

На сайте rchrchive.net («архив RCR») четко прописано, как следует подавать запросы, посылать критические замечания и отзывы и даже голосовать за или против предложенных изменений.

Прежде чем подавать новый запрос, убедитесь, что никто не предлагал такого же изменения ранее. Возможно, предложение, идентичное вашему, ожидает своей участи или уже отвергнуто.

Во-вторых, проверьте, что «расставили все точки над i». Если вы предлагаете изменить поведение метода, рассмотрите вопрос о работоспособности уже написанных программ. Если предлагаете ввести новый оператор, подумайте, является ли он в действительности методом, каков должен быть его приоритет, какое влияние он окажет на синтаксический анализатор и т. д.

Никто не гарантирует, что ваш запрос будет одобрен, – все в умелых руках Маца. Но чем тщательнее вы обдумаете свое предложение заранее, тем больше шансов, что оно будет принято всерьез. Список рассылки ruby-core – тоже подходящее место для обсуждения таких вопросов (но не тратьте зря время разработчиков ядра).

22.5. Каналы IRC

Можно побиться об заклад, что в любое время суток десятки «рубистов» болтают в IRC-чатах. Такие серверы принадлежат freenode.net; зайдите на этот сайт и поищите ближайший к вам. Поиск в Web поможет вам найти подходящий IRC-клиент для любой платформы.

В канале [#ruby-lang](http://irc.freenode.net/#ruby-lang) довольно интенсивный трафик. Это международный форум, так что кого-нибудь вы там встретите в любое время суток. Языком международного общения де-факто является английский, но кто-нибудь наверняка подскажет вам каналы, где говорят на вашем языке.

Соблюдайте общепринятые в IRC правила общения. Главное, не «флудите», посылая фрагменты кода: одна-две строки на сообщение – это максимум. При необходимости пользуйтесь службой отправки фрагментов rafb.net или аналогичной.

Есть и другие каналы, скажем, посвященный Rails. Как обычно, любая информация такого рода может быстро устареть.

22.6. Конференции по Ruby

Первая международная конференция по Ruby состоялась во Флориде в 2001 году, затем города проведения менялись. Обычно на конференции приезжает и сам

Мац, и еще несколько японских рубистов; вообще, участники прибывают со всех континентов. (Если вы работаете на антарктической научной станции, не забудьте взять отпуск и посетить очередную конференцию!)

Проведение ежегодных конференций курируется организацией RubyCentral, на сайте rubycentral.com вы можете найти актуальную информацию. А можете зайти непосредственно на сайт <http://rubyconf.org>, подать заявку на выступление или зарегистрироваться в качестве участника.

Есть и другие конференции. Европейская конференция по Ruby (EuRuKo) впервые была проведена в Карлсруэ (Германия) в 2003 году. Там обычно бывает меньше народу, но если вы живете в Европе, то добираться удобнее. Я ничего не знаю об официальном сайте этой конференции; задайте поисковой машине слово «EuRuKo» для получения дополнительной информации.

Популярность Ruby возросла настолько, что ему уже были посвящены выступления на конференциях OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications – объектно-ориентированное программирование, системы, языки и приложения), OSCON (Open Source Convention – конвенция о программном обеспечении с открытыми исходными текстами) и др. То ли еще будет!

Популярность каркаса Ruby on Rails для разработки Web-приложений привела к организации посвященных только ему конференций. Первая международная конференция по Rails состоялась в Чикаго в июне 2006 года, другие уже на подходе. Свежую информацию ищите на сайте railsconf.org.

Не так давно я принимал участие в конференции Силиконовой долины по Ruby, которая состоялась в г. Санта-Клара в апреле 2006 года. Это первая посвященная Ruby конференция в США, если не считать международных конференций, спонсируемых RubyCentral. Надеюсь, что не последняя.

22.7. Локальные группы пользователей Ruby

В США и по всему миру возникают многочисленные локальные группы пользователей. Обычно они называются как-то вроде Город.rb, что напоминает имя файла, содержащего Ruby-программу (кстати, .rb означает еще и Ruby Brigade). Некоторые группы неформальны, другие – более структурированы. Крупные, активные группы есть в Сиэттле (штат Вашингтон), Остине (штат Техас), Портланде (штат Орегон) и Нью-Йорке. Информацию о ближайшей к вам группе можно найти на сайте rubygarden.org.

22.8. Заключение

Вы дошли до конца этого объемистого тома. В противовес моим ожиданиям, читатели говорили, что прочли первое издание от корки до корки, а некоторые даже учили Ruby по этой книге (хотя я уделил очень мало времени изложению основ).

Мне не важно, читали вы книгу последовательно или случайно наткнулись на этот абзац. Как бы то ни было, вы дошли до конца, с чем вас и поздравляю.

Но программирование осваивают не по книгам, а применяя полученные из книг знания на практике. Поэтому я призываю вас заняться тем, чем и положено

заниматься программисту: идите и пишите программы. Только так вы приобретете настоящий опыт. А если что-то не получается и в книгах нет ответа, обратитесь к сообществу. Наверняка найдутся люди, которые сумеют вам помочь (и люди, которым поможете вы).

Кто входит в «сообщество пользователей Ruby»? Ну, раз вы читаете этот текст, то и вы в том числе. От имени прочих членов приветствую вас и призываю принять активное участие в жизни сообщества.

Алфавитный указатель

Символы

- ! (восклицательный знак),
 - в идентификаторах 63
- " (двойные кавычки), строки 81
- : (двоеточие) См. Двоеточие
- # (решетка)
 - # {}, синтаксическая конструкция, строки 96
 - комментарии 37
 - методы экземпляра 70
 - строки 63
- \$& переменная, подстановка строк 90
- \$default, поток ввода/вывода 288
- \$KCODE, глобальная переменная 139
- \$MATCH, переменная, подстановка строк 90
- \$\$SAFE, переменная, потоки 452
- \$stdout, переменная 287
- %Q, конструкция, строки 81
- %U, спецификатор, определения дня недели 209
- %W, спецификатор, определения дня недели 209
- % (процент), метод
 - класс String 87
 - преобразование системы счисления 178
 - форматирование чисел 163
- &, оператор
 - математические множества 261
 - поразрядные операции над числами 176
- &&, оператор 65
- () (круглые скобки)
 - методы 77
 - строки 81
- **, оператор возведения в степень 159
- *, квантор 115
- +, оператор над множествами 261
- (дефис), классы символов 119
- /m (модификатор многострочности), регулярные выражения 76
- <<, оператор дописывания в конец строки 94
- < (меньше)
 - << оператор сдвига влево 177
 - <=> оператор сравнения 77, 85
- <>, угловые скобки 81
- = (знак равенства)
 - == (оператор отношения) 67, 77, 341
 - == (оператор проверки на равенство) 67, 85, 343
 - =end, маркер, встроенная документация 37
- > (больше)
 - >> оператор сдвига вправо 177
- ? (вопросительный знак)
 - ?, конструкция, регулярные выражения 122
 - в идентификаторах 63
- @@, префикс (переменные) 65
- [], квадратные скобки
 - математические множества, создание 261
 - строки 81
- ^, каре
 - классы символов 119
 - оператор исключающего или, поразрядные операции над числами 176
 - строковые параметры 101
- __END__, директива 302
- _ (знак подчеркивания)
 - идентификаторы 36, 62
 - числа 158
- ` (обратная кавычка) 37, 476
- { } (фигурные скобки)
 - блоки 72
 - строки 81
- | (вертикальная черта)
 - ||, оператор 65
 - конвейер 499

A

- Абстрактные классы 34, 55
- Аксессуары 77
 - cweek, определение недели в году 209
 - zone, определение часового пояса 210
- Аргументы
 - ARGF 481
 - ARGV 482
 - именованные 56
 - метода 56
- Аспектно-ориентированное программирование (AOP) 374
- Атрибуты
 - аксессуары 77
 - и локальные переменные 66
 - класса 31
 - методы установки 77
 - методы чтения 77
 - определение 30
 - экземпляра, создание 333

B

- Базы данных, внешние соединения
 - LDAP 323
 - MySQL 319
 - Oracle 325
 - ORM 327
 - PostgreSQL 321
 - SQLite 317
 - обертки вокруг DBI 326
- Байты
 - интерпретация 136
 - и символы 139
 - порядок следования 179
 - составные и монолитные формы 141
- Библиотеки
 - feedtools 510
 - Open-URI 590
 - PDF::Writer 521
 - метод quadrant 522
 - пример документа 523
 - RSS, стандартная 508
 - ruby-breakpoint 541
- Блоги, посвященные Ruby 658
- Блоки
 - do-end 56
 - Kernel#trace_var, метод 388
 - set_trace_func, метод 387
 - в фигурных скобках 56, 72

- метод initialize 333
- отложенная интерполяция строк 96
- передача итератору 72, 76
- передача методам 56
- синтаксис 63
- Булевские типы 64

B

- Ввод/вывод
 - \$stdout, переменная 287
 - ARGF, переменная 287
 - fcntl, метод 295
 - fileno, метод 295
 - ioctl, метод 295
 - readpartial, метод 296
 - select, метод 295
 - STDERR, константа 287
 - STDIN, константа 287
 - STDOUT, константа 287
 - syscall, метод 295
 - буферизованный
 - и небуферизованный 288
 - каналы 294
 - каталоги. См. Каталоги
 - неблокирующий 296
 - перенаправление 483
 - программы
 - чтение встроенных данных 302
 - чтение исходного текста 302
 - файлы. См. Файлы
- Ввод символов с клавиатуры 300
- Ветвящегося равенства оператор (==) 43, 341
- Високосные секунды 207
- Високосный год, проверка 210
- Висячие указатели 61
- Вложения, кодирование и декодирование 582
- Внешние программы, запуск
 - манипулирование процессами 477
 - метод exec 475
 - метод system 475
 - перехват выводимой информации 476
 - стандартный ввод/вывод 479
- Возведение в степень 159
- Временные файлы 303
- Временные штампы 290
- Встроенная документация 37, 63
- Встроенные данные, чтение 302
- Встроенные документы 81

Выражения

- включение в строку 96
- интерполяция 37
- ориентация на выражения 75
- регулярные 76
- синтаксические ошибки 62

Г

Генераторы

- реализация с помощью продолжений 364
- чисел Фибоначчи 365

Глиф

- определение 137
- составные и монолитные формы 142

Глобальные константы 287

Глобальные методы, синонимы 69

Глобальные переменные 36

- \$KCODE 139
- \$SAFE 452
- \$stdout 287
- переменные окружения, импорт 486
- синонимы 69

Глубокое копирование 308, 345

Град, единица измерения углов 181

Графема, определение 137

Графический интерфейс пользователя

- Apollo 446
- FXRuby. См. FXRuby
- QtRuby. См. QtRuby
- Ruby/GTK2. См. Ruby/GTK2
- Ruby/Tk. См. Ruby/Tk

Графы

- ресурсы в Web 280
- связность 277
- эйлеров путь 279
- эйлеров цикл 278

Д

Дата и время

- mktime, метод 203
- strftime, метод 204
- Time.gm, метод 203
- Time.utc, метод 204
- to_a, метод 204
- wday, метод экземпляра 204
- високосные секунды 207
- всеобщее скоординированное время (UTC) 202, 216
- вывод на печать 215

вычисление n-ого дня недели

в месяце 205

вычисление интервала 212

извлечение из строки 214

контроль 208

летнее время 202

определение даты Пасхи 204

определение дня недели 204

определение недели в году 209

определение порядкового номера дня в году 208

определение текущего времени 203

определение числа дней в месяце 216

преобразование к точке отсчета 207

преобразования 213

прибавление интервала 211

разбиение месяца на недели 216

секунд в дни, часы, минуты 206

сопоставление с помощью регулярных выражений 131

сравнение 211

среднее время по Гринвичу 202, 216

точка отсчета 202, 213

форматирование 215

часовые пояса

определение 210

преобразование 216

юлианская дата 208

Двоеточие – теги, распознаваемые rdoc 554

Двоичные деревья

реализация 269

сортировка 271

справочные таблицы 272

Двоичные файлы 285

Действия, утилита rake 645

Делегирование

вызовы методов 370

и наследование 372

Деревья 268

двоичные

реализация 269

сортировка 271

справочные таблицы 272

преобразование в массив 273

преобразование в строку 273

разбор XML-документа 505

Деструкторы 30

Диалоговые окна, создание в FXRuby 432

Динамические средства

const_get, метод 377

const_missing, метод 382

define_method, метод 378

динамическая интерпретация
 кода 375

классы

- отслеживание изменений 390
- получение списка закрытых
 методов 385
- получение списка защищенных
 методов 386
- получение списка модулей 385
- получение списка открытых
 методов 386
- получение списка переменных 385
- получение списка предков 385

константы, получение списка 385

методы

- const_missing 382
- вызовы несуществующих 389
- добавление в класс или объект 378

модули

- отслеживание добавлений 391
- получение списка 385

объекты

- чистильщики 393

определения, удаление 383

переменные экземпляра, получение
 и установка 378

просмотр стека вызовов 387

функция eval 375

Дисперсия (наборы данных) 183

Дубликаты, удаление из строки 102

Дуговой градус 181

Ж

Журналы онлайнные о Ruby 658

З

Заглушки (Ruby on Rails) 601

Заглядывание вперед, регулярные
 выражения 116

Замораживание объектов 354

Замыкание 73

И

Изменяемые объекты 50

Именованные параметры 56

Именованные соответствия,
 регулярные выражения 127

Индексная переменная, модификация 66

Инкапсуляция, определение 30

Интернационализация 134

- jcode, библиотека 141
- jcount, метод 141
- mbchar?, метод 141
- UCS-2, кодировка 138
- Unicode
 - кодировки 137
 - терминология 136
- UTF-16, кодировка 138
- UTF-8, кодировка. См. Интернацио-
 нализация, строки
- байты и символы 139
- кодировки
 - преобразование 148
 - распознавание 144
- регулярные выражения 143
- составные символы 141
- справочники сообщений. См.
 Справочники сообщений
- строки
 - UTF-8, кодировка 138
 - \$KCODE, глобальная
 переменная 139
 - unpack, метод 142
 - интерпретация байтов 136, 139
 - квадратные скобки 142
 - кодовые позиции 140
 - конкатенация 140
 - нормализация 144
 - проверка вхождения подстроки 140
 - упорядочение 146

Интерполяция 37, 96

Исключения

- блоки begin-end 46
- ключевое слово retry 47
- переменная err 47
- предложение raise 46
- предложение rescue 47

Итераторы

- each
 - конструкция for 71
 - строки 83
- each_byte
 - строки 84
- retry 72
- блоки 72, 76
- вызовы методов 72
- параллельные, потоки 471
- строки 73
- циклы 45

К

Канальный уровень, сетевое
программирование 561

Каталоги

изменение корня 304
изменение текущего 303
обход 304
поиск 305
получение содержимого 304
рекурсивное удаление 305
цепочка 304

Кванторы, регулярные выражения 114

* 115

Oniguruma 126

Классы

Complex 171

Date

акцессор cweek 209
контроль даты и времени 208
преобразование в DateTime 213

Deflate, сжатие строк 101

DelegateClass, метод 370

FalseClass 64

File

atime, метод 290
basename, метод 297
binmode, метод 285
chmod, метод 289
chown, метод 289
ctime, метод 290
delete, метод 298
dirname, метод 297
flock, метод 286
join, метод 297
link, метод 299
mtime, метод 290
new, метод 282
open, метод 283
path, метод 297
putc, метод 287
rename, метод 299
rewind, метод 284
seek, метод 284
split, метод 297
symlink, метод 299
tell, метод 284
truncate, метод 299
umask, метод 290
utime, метод 291

Inflate, сжатие строк 101

Module 53, 60

Pathname 297

PGlarge 322

PGresult 323

PGconn 321

Rational 166

self, псевдопеременная 53

SimpleDelegator 370

SingleForwardable 372

SizedQueue, потоки 463

String. См. String, класс

StringScanner 86

Struct 354

Time. См. Time, класс

TrueClass 64

абстрактные 34, 55

вложенные 361

закрытые методы 54

защищенные методы 54

имена 51

конкретный тип 34

конструкторы 393

методы

добавление 378

закрытые 337

объявление 335

отслеживание добавлений 391

получение списка 385

создание 334

модули 34

включение 368

получение списка 385

наследование 31, 55

объекты 71

опрос типа 340

создание 49

определение 30, 51

открытые методы 54

отслеживание изменений

в определении 390

параметрические, создание 361

переменные 64

получение списка 385

префикс @@ 65

создание 335

получение значения константы 377

производные 32

синглетные 78

содержащие только данные 353

список предков 385

структурные шаблоны 354

суперклассы, наследование 338

является, отношение 32

Классы символов, регулярные
выражения 119

Ключевые слова

begin 63
break 45
defined? 59
do 63
elsif 62
end 63
next 45
redo 45
retry 45
then 63
yield 45

как имена методов 62

список 36

Кнопки (графический интерфейс)

FXRuby 425

QtRuby 438

Ruby/GTK2 412

Ruby/Tk 399

Кодировки (Unicode) 137

Кодовая позиция

определение 137

составные и монолитные формы 142

строки 140

Код завершения,

получение и установка 500

Командная строка

аргументы, ARGV[0] 65

разбор флагов 479

Комментарии 37

Комплексные числа 171

Конвейерная компиляция, Nitro 612

Конкретные типы 34

Константы 36

изменение значения 70

отсутствующие 382

получение списка 385

удаление 384

числовые 37

Конструкторы 30, 393

Контроллеры, Nitro 606

Конференции по Ruby 659

Концевой маркер, встроенные документы 82

Копирование

глубокое 308, 345

дерева каталогов 495

объектов 345

поверхностное 345

файлов 299

Коэффициент корреляции 184

Критические секции, синхронизация
потоков 459

Куки, программирование CGI 595

Л

Левенштейна расстояние, строки 105

Летнее время 202

Логарифмы 178, 182

Локализация 134

Локальные переменные 36

и атрибуты объекта 66

примеры 36

М

Маршалинг

YAML 312

глубокое копирование 308

нестандартный 307

простой 306

Массивы. См. также Хэши

ассоциативные. См. Хэши

битовые 177

деревья и 273

и переменные окружения 486

метод pack, двоичные файлы 286

параметров, сворачивание 56

преобразование

объекта в массив 352

строки в массив 352

примеры 38

строки 38

base64-кодирование 107

разложение строки в массив 84

чтение файла в массив 300

Матрица смежности 274

Матрицы 167

Медиана, наборы данных 183

Меню, создание

FXRuby 428

Ruby/GTK2 418

Метаданные, определение 559

Методы

abort, отмена транзакции 309

accessor_string 359

add, множества, форматирование даты
и времени 262

add_bool_config 557

add_path_config 557

alias_method 55, 392

- alive?, потоки 452
- allocate, неинициализированные объекты 347
- append_features 348, 393
- asctime, форматирование даты и времени 215
- atime 290
- attr, создание атрибутов экземпляра 333
- basename 297
- binmode, двоичные файлы 285
- blksize 292
- blockdev? 292
- blocks 292
- calcc 364
- caller, определение вызвавшего метода 387
- capitalize, класс String 88
- casecmp, класс String 88
- cattr_reader 372
- cattr_writer 372
- center, форматирование строк 87
- chardev? 292
- children 298
- chmod 289
- chomp, строки 95
- chop, строки 94
- chr, чтение файла 301
- chroot 304
- class_variables 385
- classify, множества 262
- clone
 - MD5-свертка 104
 - копирование объектов 345
- close, временные файлы 303
- coerce 352
- column_size, матрицы 169
- commit, фиксация транзакции 309
- compare_file 299
- config_key 557
- const_get 377
- const_missing 382
- constants 385
- copy 299
- count, строки 101
- covector 170
- crc32, строки 103
- crypt, шифрование строк 100
- ctime
 - форматирование даты и времени 215
- ctime, временные штампы файла 290
- curr_objdir, идентификация потока 557
- curr_srcdir, идентификация потока 557
- current, идентификация потока 451
- Date.leap? 210
- deep_copy 346
- def_delegator 371
- define_method, добавление метода в класс/объект 378
- DelegateClass 370
- delete
 - удаление корней 310
 - удаление символов из строки 102
 - файлы 298
- deq, потоки 464
- dev, низкоуровневая информация о файле 293
- digest, MD5-свертка 104
- Dir.chdir 303
- Dir.entries 304
- Dir.getws 303
- Dir.pwd 303
- dirname 297
- divide, множества 262
- downcase, класс String 88
- dump, печать специальных символов 102
- dup, копирование объектов 345
- each
 - обход каталога 304
 - чтение файла 301
- empty?, потоки 464
- enq, потоки 464
- eq?, проверка объектов на равенство 342
- equal?, проверка объектов на равенство 342
- exec, PostgreSQL 322
- execute_query 313
- exist, проверка существования файла 291
- exit, потоки 452
- extend 51, 360
- extend_object 393
- fcntl 295
- fileno 295
- Float, преобразование строки в число 98
- flock 286
- foreach, обход каталога 304
- fork, создание потока 450
- ftype 293
- get_config_key 557
- getmyvar 52
- gregorian_leap? 210
- gsub, подстановка в строку 90
- hexdigest, MD5-свертка 104
- hook_method 391
- include, поиск в строке 91

- included_modules 385
- index, поиск в строке 91
- inherited 392
- initialize 72, 333
- initialize_copy 346
- inner_product 171
- ino 293
- insert, двоичные деревья 269
- inspect 352
- install 299
- instance_methods 385
- instance_of? 340
- Integer, преобразование строки в число 98
- integrate 180
- intersection, множества 261
- IO
 - построчное чтение из файла 301
 - чтение файла в память 300
- IO.popen 294
- ioctl 295
- is_a? 340
- join
 - поток 454
 - путевые имена файлов 297
- julian_leap? 210
- Kernel#trace_var 388
- key?, доступ к переменным потока 450
- kill, поток 452
- kind_of? 340
- length, строки 83
- link 299
- list, поток 457
- ljust, форматирование строк 87
- load 51
- lstrip 96
- main, главный поток программы 451
- makedirs 304
- md5 104
- method_added 391
- method_missing 389
- mktime 203
- Module#ancestors 385
- move 299
- mtime 290
- myvar= 52
- num_waiting 464
- ObjectSpace.each_object 389
- offset 119
- open 283
- pack
 - base64-кодирование 107
 - двоичные файлы 286
- polar 171
- pop, поток 464
- prime_division 172
- printf 178
- private 54, 343
- private_class_method 337
- private_instance_methods 385
- private_methods 386
- protected 54, 344
- protected_instance_methods 385
- protected_methods 386
- public 54, 344
- public_instance_methods 385
- public_methods 386
- push, поток 464
- putc 287
- puts 93
- query, PostgreSQL 322
- raise, поток 454
- rdev 293
- readpartial 296
- Regexp.compile 112
- Regexp.escape 113
- Regexp.last_match 91, 118
- remove_const 384
- remove_method 383
- rename 299
- require 51
- respond_to? 59, 341
- reverse, строки 102
- rewind, прямой доступ к файлу 284
- rindex, поиск в строке 91
- rjust, форматирование строк 87
- rm_r, рекурсивное удаление каталога 305
- rmtree, рекурсивное удаление каталога 305
- root? 298
- rot13, кодирование строк 99
- row_size, vfnhbws 169
- rstrip 96
- run, возобновление потока 453
- safe_level 452
- safe_unlink 299
- scan, сопоставление с регулярным выражением 86, 91
- scanf 99
- seek, прямой доступ к файлу 284
- select 295
- set_config_key 557
- set_trace_func, мониторинг выполнения программы 387
- setmyvar 52

- shift, потоки 464
- similar?, расстояние Левенштейна 106
- singleton_method_added 392
- singleton_methods 386
- size? 291
- sort_by 356
- split
 - путевые имена файлов 297
 - разбиение строки на лексемы 85
- sprintf
 - преобразование системы счисления 178
 - форматирование строк 87
- squeeze, удаление дубликатов 102
- srcfiles 557
- start, потоки 452
- status, потоки 452
- stop, потоки 452
- stop?, потоки 452
- strftime
 - определение дня недели 204
 - определение недели в году 209
 - печать часов, минут и секунд 211
 - преобразование часовых поясов 216
 - форматирование даты и времени 215
- strip 95
- sub, подстановка строк 90
- succ, генерирование последовательности строк 103
- swapcase, класс String 88
- symlink 299
- sync, буферизация ввода/вывода 288
- syscall 295
- ThreadGroup.new, создание группы потоков 457
- Time.at 207
- Time.gm 203
- Time.local 203
- Time.utc 204
- to_ary 352
- to_f, преобразование строки в число 98
- to_hash 314
- to_i, преобразование строки в число 98
- to_int, неявные преобразования 175
- to_s 92, 177, 215, 314
- to_set 261
- to_str 92, 351
- truncate 299
- tty? 292
- type 59
- umask 290
- undef_method 383
- ungetc 288
- union, множества 261
- unpack, двоичные файлы 286
- update, MD5-свертка 104
- upto, генерирование последовательности строк 103
- utime 291
- value, потоки 455
- wakeup, возобновление потока 453
- zero, матрицы 168
- zero?, проверка файла на пустоту 292
- Zlib, методы сжатия 101
- вызовы
 - делегирование 370
 - перенаправление 370
 - скобки 61
- глобальные, синонимы 69
- именованные аргументы 56
- как операторы 65
- класса
 - закрытые 337
 - объявление 335
 - создание 334
- определение 30
- отслеживание добавлений 391
- отсутствующие 60
- передача блока 56
- переопределение 32, 55
- синглетные 56, 79
- синонимы, создание 55
- сцепление 56
- управление доступом 343
- установки 71, 332
- хэш, передача 62
- чтения 332
- экземпляра 51
- executable_real? 289
- gid 289
- grpowned? 289
- mode 289
- MysqlField 320
- MysqlRes 320
- owned? 289
- readable_real? 289
- stat 291
- to_a 204
- uid 289
- wday 204
- writable_real? 289
- zero? 291

Множества математические
 более сложные операции 262
 перечисляемые объекты,
 преобразование 261
 простые операции 261
 создание 261
 Мода, наборы данных 183
 Модальные диалоги, создание в
 FXRuby 432
 Модификаторы,
 регулярные выражение 111
 Модули 34
 append_features, метод 348
 Math, управление пространством
 имен 51
 ParseDate 214
 включенные в класс, получение
 списка 385
 вложенные 361
 добавление функций 51
 методы экземпляра 51
 определение 51, 461
 отслеживание добавления методов 391
 подмешивание к объектам 360
 получение списка констант 377
 примеси 348
 Мониторинг выполнения программы 387
 Мультимодальные наборы данных 183
 Мьютексы 461

Н

Наборы данных
 дисперсия 183
 медиана 183
 мода 183
 среднее 182
 Наборы символов, глобальная переменная
 \$KCODE 139
 Наследование 31
 и делегирование 372
 множественное 32
 отношение «является» 339
 полиморфизм 33
 ромбовидное 32
 символ < 72
 Незапоминаемые группы 118
 Немодальные диалоги, создание в
 FXRuby 432
 Неявные преобразования
 строк 92
 чисел 174

Новостные группы (Ruby) 657
 Нормализация Unicode-строк 144



Обратные ссылки, регулярные
 выражения 117
 Объектно-реляционные отображения
 (ORM) 327
 Оглядывание назад, регулярные
 выражения 125
 ООП (объектно-ориентированное
 программирование)
 coerce, метод 352
 Struct, класс 354
 UnboundMethod, объект 367
 АОП 374
 атрибуты экземпляра, создание 332
 без классов 373
 вложенные классы 361
 замораживание объектов 354
 классы
 включение модулей 368
 опрос 340
 параметрические, создание 361
 содержащие только данные 353
 методы
 делегирование 370
 перенаправление 370
 управление доступом 343
 методы класса
 закрытые 337
 объявление 335
 создание 334
 методы чтения и установки уровня
 класса 372
 модули
 append_features, метод 348
 включение в классы 368
 методы экземпляра 348
 подмешивание к объекту 350, 360
 примеси 348
 наследование 338
 на основе прототипов 373
 несколько конструкторов 331
 объекты
 Proc 366
 копирование 345
 неинициализированные 347
 специализация 357
 сравнение на равенство 342
 параметры, заданные по умолчанию 369

паттерны проектирования 374
 переменные класса
 создание 335
 преобразование
 объекта в массив 352
 объекта в представление
 для печати 352
 объекта в строку 351
 строки в массив 352
 продолжения, реализация
 генераторов 364
 проектирование по контракту 374
 сложные конструкторы 333
 функциональное
 программирование 373
 экстремальное программирование 374
Операторы
 && 65
 || 65
 деления по модулю 64
 диапазона 69
 для объекта nil 71
 как методы 65
 отношения 77
 перегрузка 55, 71
 перечень 39
 повтора строк 96
 приоритет 65
 разрешения области видимости 71
 расширения массива 78
 составного присваивания 70
 сравнения 77, 85
 тернарный 63, 76
Операционная система, определение 501
Определения
 исполняемые 71
 удаление 383
Отладка
 drb, инструмент 542
 irb, инструмент 541
 отладчик Ruby 538
 приложений Ruby on Rails 601
Отражение 59, 385, 386
Очереди 264, 267

П

Параметры
 заданные по умолчанию 369
 массивы, сворачивание 56
 передача потоку 449
Паттерны проектирования 374

Переключатели, создание
 FXRuby 428
 QtRuby 438
 Ruby/GTK2 418
 Ruby/Tk 407
Переключатели (графический интерфейс)
 FXRuby 428
 QtRuby 438
 Ruby/GTK2 418
 Ruby/Tk 407
Переменные
 \$&, подстановка строк 90
 \$KCODE 139
 \$MATCH, подстановка строк 90
 \$SAFE, потоки 452
 @@, префикс 65
 егг, исключения 47
 глобальные 36
 \$stdout 287
 примеры 36
 синонимы 69
 интерполяция 37
 класса 36
 получение списка 385
 создание 335
 локальные 36
 и атрибуты объекта 66
 и имена методов 66
 примеры 36
 массивы 38
 обмен значений 70
 объявление 64
 окружения 485
 импорт как глобальных 486
 хранение в виде массива или хэша 486
 чтение и установка 485
 потока 449
 присваивание значений 70
 псевдопеременные 63
 self 53
 примеры 36
 условные, потоки 465
 хэши 39
 экземпляра 36
 класса 74
 получение и установка 378
 получение списка 385
 примеры 36
 экземпляра класса 79, 337
Переменные окружения 485
 импорт в виде глобальных
 переменных 486

- справочники сообщений 151
 - хранение в виде массива или хэша 486
 - чтение и установка 485
 - Переменные экземпляра
 - экземпляра класса 337
 - Перенаправление вызовов методов 370
 - Подключаемые модули (Ruby on Rails) 603
 - Покрывание кода, измерение 542
 - Потоки 448
 - Monitor, примесь 467
 - ввод/вывод, продолжение обработки 471
 - возбуждение исключений 454
 - возобновление 453
 - главный
 - ожидание других 455
 - получение ссылки 451
 - группы
 - возобновление 457
 - добавление потока 457
 - завершение 457
 - получение списка потоков 457
 - создание 457
 - идентификация 451
 - мьютексы 461
 - обработка исключений 455
 - опрос уровня безопасности 452
 - параллельные итераторы 471
 - передача параметров 449
 - перечисление 451
 - планировщик, передача управления 453
 - приоритеты 453
 - приостановка 452
 - синхронизация 458
 - доступа к ресурсам 460
 - критические секции 459
 - очереди, предопределенные классы 463
 - события, ожидание 469
 - создание 449
 - состояние 452
 - тайм-аут 469
 - тупиковая ситуация 448
 - условные переменные 465
 - Поэтический режим 77
 - Предки класса, получение списка 385
 - Преобладающие объекты, Madeleine 313
 - Преобразование
 - кодировок 148
 - объекта в массив 352
 - объекта в представление для печати 352
 - объекта в строку 351
 - строки в массив 352
 - Приведение типов 175
 - Прикладной уровень (сетевое программирование) 561
 - Примеси 348
 - Monitor, потоки 467
 - определение 51
 - Присваивания операторы составные 70
 - Проблема ромбовидного наследования 32
 - Продолжение, реализация генераторов 364
 - Проектирование по контракту 374
 - Производительность, измерение 544
 - Benchmark.bm, метод 546
 - Benchmark.measure, метод 546
 - Производные классы 32
 - Пропуски, синтаксические ошибки 62
 - Пространства кортежей 636
 - Псевдопеременные 63
 - self 53
 - примеры 36
 - Путевые имена файлов 297
- Р**
- Разработки средства
 - IDE
 - ArachnoRuby 655
 - FreeRIDE 654
 - Komodo 655
 - RDE 655
 - RDT 655
 - irb, утилита 647
 - rake, утилита 644
 - ri, утилита 652
 - RubyGems
 - запуск gem-сервера 644
 - получение информации
 - об установленных пакетах 644
 - установка 643
 - поддержка со стороны редакторов 653
 - Распределенный Ruby (drb) 630
 - клиент 631
 - обнаружение сервисов 640
 - пространства кортежей (Rinda) 636
 - сервер 630
 - списки контроля доступа 632
 - Расширения массива оператор 78
 - Регулярные выражения
 - Oniguruma. См. Oniguruma
 - внутренние модификаторы 122
 - внутренние подвыражения 122
 - заглядывание вперед 116
 - интернационализация 143
 - кванторы 114

классы символов 119
 кодировка UTF-8 140
 компиляция 112
 литеральные 113
 модификаторы 111
 многострочности 76
 обнаружение повторяющихся слов 132
 обобщенные 120
 обратные ссылки 117
 простые примеры 110
 римские цифры 130
 синтаксис 110
 соответствия нулевой длины 76
 сопоставление со строкой 86
 сопоставление с IP-адресом 129
 сопоставление с датой и временем 131
 сопоставление с номером версии 132
 сопоставление с номером телефона 133
 сопоставление с парой ключ-значение 130
 сопоставление с почтовым индексом 133
 сопоставление с символом конца строки 121
 сопоставление с числовыми константами 131
 экранирование специальных символов 113
 якоря 113

Редакторы

emacs 654
 jEdit 653
 SciTe 653
 TextMate 602, 653

С

Сборка мусора, алгоритм пометки и удаления 393
 Сериализация объектов 306
 Сетевой уровень, сетевое программирование 561
 Сжатие строк 101
 Символические ссылки 293
 Синглеты 358
 Singleton, паттерн 78
 классы 78
 методы 79
 определение 56
 создание 72
 Синонимы
 fail 71
 kind_of? 59

глобальных методов 69
 глобальных переменных 69
 методов, создание 55
 Система счисления, преобразование 177
 Случайные числа, генерирование 185
 Списки рассылки, посвященные Ruby 658
 Справочники сообщений
 gettext, библиотека 151
 МО-файлы 152
 РО-файлы 152
 RubyGems 156
 переменные окружения 151
 текстовые домен 152
 Сравнение файлов 299
 Сравнения оператор (<=>) 77, 85
 Среднее, наборы данных 182
 Стек 263
 вызовов 387
 и рекурсия 266
 обнаружение несбалансированных скобок в выражениях 265
 строгая реализация 264
 Стенд (Ruby on Rails) 601
 Строки
 %q, конструкция 81
 base64-кодирование 106
 each, итератор 83
 each_byte, итератор 84
 MD5-свертка 104
 StringScanner, класс 86
 unpack, метод 142
 встроенные документы 81
 вычисление 32-разрядного CRC 103
 в двойных кавычках 81
 в одиночных кавычках 81
 генерирование последовательности 103
 данные, разделенные запятыми 97
 длина 83
 извлечение даты и времени 214
 интернационализация, библиотека
 jcode 140
 интернационализация, глобальная
 переменная \$KCODE 139
 интернационализация, конкатенация в
 кодировке UTF-8 140
 интерполяция отложенная 96
 как объекты IO 87
 как файлы 301
 кодовая позиция 140
 конкатенация с числом 351
 массивы 38, 84
 нормализация 144



обращение 102
 односимвольные 92
 оператор << 94
 парные ограничители 81
 перехват вывода команды 37
 повтор 96
 подстановка 90
 подстроки
 вычленение и замена 88
 и кодировка UTF-8 140
 подсчет числа символов 101
 поиск 91
 преобразования
 в массив 352
 в число 98
 между символами и кодами ASCII 92
 объектов в 351
 символов в коды ASCII
 и обратно 92
 явные и неявные 92
 разбиение на лексемы 85
 расстояние Левенштейна 105
 регистр, определение 88
 сжатие 101
 символы, преобразование в строку 50
 сопоставление с регулярным
 выражением 86
 специализированное сравнение 84
 табуляторы 107
 удаление лишних пропусков 95
 удаление хвостовых символов 94
 форматирование 87, 163
 цитирование текста 108
 шифрование
 сгрупп 100
 rot13 99
 Структурные шаблоны 354
 Сцепление вызовов 56

T

Текстовые поля
 FXRuby 427
 QtRuby 439
 Ruby/GTK2 413
 Ruby/Tk 402
 Текущее время, определение 203
 Тестирование
 Test::Unit, библиотека 531
 flunk, метод 534
 комплект тестов 534
 утверждения 533

ZenTest
 autotest, программа 538
 multiruby, программа 538
 unit_diff, программа 537
 тестируемый класс 535
 флаги в командной строке 537
 приложений Ruby on Rails 601
 Тестируемый класс, ZenTest 535
 Тождественная матрица 168
 Точка отсчета 202
 более ранние даты 213
 контроль даты и времени 208
 микросекунды 207
 преобразование 207
 Транзакции 309
 Транспортный уровень, сетевое
 программирование 561
 Трема 145
 Тригонометрические функции 180
 Тупиковая ситуация 448

У

Удаление
 каталогов рекурсивное 305
 констант 384
 корней 310
 определений 383
 файлов 298, 497
 Унимодальные наборы данных 183
 Упорядочение строк 146
 Управляющие конструкции
 redo, ключевое слово 45
 retry, ключевое слово 45
 until 43
 while 43
 yield, ключевое слово 45
 ветвление 42
 циклы
 break, ключевое слово 45
 next, ключевое слово 45
 итераторы 45
 модуль Kernel 44
 примеры 43
 с предварительной проверкой
 условия 43
 Условные переменные, потоки 465
 Условные предложения 42
 Установка
 Microsoft Windows, моментальный
 инсталлятор 493
 Ruby, быстрая 498

Утверждения 533
«Утилизация» 78



Файлы

umask 290
ассоциированы ли с терминалом 292
блокировка 286
блочные устройства 292
временные 303
временные штампы 290
двоичные 285
дописывание в конец 284
идентификатор пользователя 289
информация низкого уровня 293
копирование 299
обновление 283
открытие и закрытие 282
переименование 299
перемещение 299
поиск 305
получение индексного узла 293
получение информации об устройстве 293
получение размера 291
посимвольный вывод 287
построчное чтение 301
права владения 288
проверка на пустоту 291
проверка существования 291
прямой доступ 284
путевые имена 297
разрешения 289
символические ссылки 293
сравнение 299
строки как 301
считывание в память 300
удаление 298, 497
усечение 299
установка 299
физические ссылки 293

Фибоначчи числа, генератор 365

Фильтры текстовые 495

Флажки

QtRuby 441
Ruby/GTK2 416
Ruby/Tk 406

Форматирование

даты и времени 215
для программы RDoc 554
строк 87
чисел 163

Форматы данных

Atom
библиотека feedtools 510
стандартная библиотека rss 508

PDF::Writer 521

метод quadrant 522
пример документа 523

REXML, разбор XML 503

RMagick 512

API рисования 517
масштабирование изображения 514
получение информации
об изображении 512
специальные эффекты
и трансформации 515

примеры 503

Форматы файлов. См. Форматы данных

Функции

atan2 181
dtms2sec, прибавление интервала
к моменту времени 212
eval, динамическая интерпретация
кода 375
sec2dhms, вычисление разности
между моментами времени 212
кэширование, memoize 186
Функциональное программирование 373

Х

Хэш. См. также Массивы
константа 38
передача методу 62
переменная 39
переменные окружения, хранение
в виде хэша 486

Ц

Цепочка каталогов, создание 304

Циклы. См. Управляющие конструкции,
циклы

Ч

Части (Nitro) 612

Числа

BigDecimal, класс 164
Fixnum 158
библиотека mathn 172
простые числа 173
разложение на простые множители 172

- битовые массивы 177
- большие целые 164
- возведение в степень 159
- вычисление определенного интеграла 180
- деление 159
- извлечение корней 178
- комплексные 171
- конкатенация со строкой 351
- коэффициент корреляции 184
- кэширование результатов
 вычисления 186
- логарифмы 178
- матрицы 167
- подчерки 158
- поразрядные операции 176
- порядок байтов 178
- представление в Ruby 158
- преобразование в строку 98, 177
- преобразование системы счисления 177
- приведение типов 175
- простые 173
- разложение на простые множители, НОД
 и НОК 172
- рациональные 166
- римские 130
- случайные, генерирование 185
- с плавающей точкой
 - округление 160
 - создание 159
 - сравнение 162

А

- abort, метод, отмена транзакции 309
- abort_on_exception, флаг, потоки 455
- accessor_string, вспомогательная функция
 уровня класса 359
- ACL, список контроля доступа 632
- ActiveLDAP 324
- add, метод, математические множества 262
- add_bool_config, метод 557
- add_path_config, метод 557
- Ajax, поддержка в Nitro 613
- alias_method, метод 55, 392

- тригонометрические функции 180
- форматирование
 - вставка разделителей 163
 - для вывода 163
- явные и неявные преобразования 174
- Чистильщики 393

Ш

- Шварца преобразование 356
- Шифрование
 - строк
 - crypt 100
 - rot13 99

Э

- Эйлеров путь, графы 279
- Эйлеров цикл, графы 278
- Электронная почта
 - вложения, кодирование
 и декодирование 582
 - отправка по протоколу SMTP 578
 - шлюз в конференции 584

Я

- Явные преобразования, строки 92
- Якоря 113

- alive?, метод экземпляра, потоки 452
- allocate, метод, неинициализированные
 объекты 347
- Amrita, библиотека 593
- Apache 622
- Apollo 447
- app.cnf, конфигурационный файл 618
- append_features, метод 348, 393
- ArachnoRuby 655
- ARGF, глобальная константа 287, 481
- ArgumentError, исключение 46
- Array#each, метод, измерение
 производительности 546

ASCII, кодировка
 интернационализация 135
 преобразование символов 92
 asctime, метод, форматирование даты
 и времени 215
 atan2, функция 181
 atime, метод, временные штампы файла 290
 Atom, формат данных 508
 библиотека feedtools 510
 attr, метод, создание атрибутов
 экземпляра 333
 autotest, программа (ZenTest) 538

В

base64, кодирование
 и декодирование 106
 basename, метод, путевые имена
 файлов 297
 begin, ключевое слово 63
 begin-end, блоки
 ensure 48
 исключения 47
 Benchmark.bm, метод, измерение
 производительности 546
 Benchmark.measure, метод, измерение
 производительности 546
 BigDecimal, класс 165
 Bignum 164
 binmode, метод, двоичные файлы 285
 blksize, метод, размер файла 292
 blockdev?, метод 292
 blocks, метод 292
 break, ключевое слово, циклы 45
 break, предложение 68

С

calculate, метод 57
 callcc, метод 364
 caller, метод 387
 capitalize, метод, класс String 88
 case, предложение 43, 66
 break 68
 ветви 68
 ключевое слово then 63
 оператор отношения (---) 67
 cascmp, метод, класс String 88
 cattr_reader, метод 372
 cattr_writer, метод 372
 center, метод, форматирование строк 87

Cerise 593
 CGI, программирование 592
 Amrita 593
 Cerise 593
 cgi.rb, библиотека 593
 ruby-web, библиотека 593
 вывод и обработка форм 595
 куки 595
 методы доступа к URL 593
 сеансы пользователей 596
 cgi.rb, библиотека 593
 chardev?, метод 292
 children, метод 298
 chmod, метод 289
 chompl, метод 40
 chop, метод 94
 chr, метод, чтение из файла 301
 chroot, метод 304
 class_variables, метод 385
 classify, метод, множества 262
 clone, метод
 глубокое копирование 308
 копирование объектов 345
 строки, вычисление
 MD5-свертки 104
 close, метод, временные файлы 303
 coerce, метод 352
 column_size, метод, матрицы 169
 commit, метод, фиксация
 транзакций 309
 Comparable, модуль, сравнение дат 211
 compare_file, метод 299
 Complex, класс 171
 config_key, метод 557
 const_get, метод
 динамическое создание экземпляра
 класса 377
 получение значений констант класса
 или модуля 377
 const_missing, метод 382
 constants, метод, получение списка
 констант 385
 copy, метод 299
 count, метод, строки 101
 covector, метод 170
 crc32, метод, строки 103
 crypt, метод, шифрование строк 100
 CSV, формат данных 310
 csv.rb, модуль 310
 ctime, метод
 временные штампы файлов 290
 форматирование даты и времени 215



curr_objdir, метод 557
 curr_srcdir, метод 557
 currrent, метод, идентификация потока 451
 cweek, аксессор, определение недели
 в году 209

D

Date, класс
 cweek, аксессор 209
 контроль даты и времени 208
 методы
 Date.leap? 210
 gregorian_leap? 210
 julian_leap? 210
 преобразование в DateTime 213
 DateTime, класс, преобразования 213
 DBI, обертки 326
 DBM, библиотека 314
 deep_copy, метод 346
 def_delegator, метод, вызовы методов 371
 define_method, добавление метода
 в класс/объект 378
 defined?, ключевое слово 59
 Deflate, класс, сжатие строк 101
 delete, метод
 корни 310
 удаление символов из строки 102
 файлы 298
 Delphi, Apollo 447
 deq, метод, потоки 464
 dev, метод, получение низкоуровневой
 информации о файле 293
 dhms2sec, функция, прибавление
 интервала к моменту времени 212
 DHTML, поддержка в Nitro 613
 digest, метод, вычисление
 MD5-свертки 104
 Dir.chdir, метод, каталоги файлов 303
 Dir.entries, метод, получение содержимого
 каталога 304
 Dir.getwd, метод, каталоги файлов 303
 Dir.pwd, метод, каталоги файлов 303
 dirname, метод, путевые имена
 файлов 297
 divide, метод, множества 262
 do, ключевое слово 63
 do-end, блоки 56
 downcase, метод, класс String 88
 drb. См. Распределенный Ruby
 dump, метод 102, 306
 dup, метод 308, 345

E

each, итератор
 конструкция for 71
 строки 83
 циклы 45
 each, метод
 обход каталога 304
 чтение файла 301
 each_byte, итератор, строки 84
 else, ветвь предложения 42, 43
 emacs 654
 empty?, метод, потоки 464
 end, ключевое слово 63
 enq, метод, потоки 464
 ensure, часть блока begin-end 48
 eql?, метод 342
 equal?, метод, проверка объектов
 на равенство 342
 erb, встроенный Ruby 622
 err, объект-исключение 47
 Etc, модуль 502
 eval, функция 375
 exes, метод
 PostgreSQL 322
 запуск внешней программы 475
 executable_real?, метод 289
 execute_command, метод 313
 execute_query, метод 313
 exist?, метод, проверка существования
 файла 291
 exit!, метод, управление
 процессами 478
 exit, метод
 получение и установка кода
 завершения 500
 потоки 452
 extend, метод
 добавление функций из модуля 51
 подмешивание модуля к объекту 360
 extend_object, метод 393

F

fail, синоним 71
 false, значение 64, 65
 FalseClass 64
 FastCGI 597
 FasterCSV, библиотека 310
 fcntl, метод 295
 feedtools, библиотека 510
 File::Stat, объект 289, 292, 293

File::Separator, объект, путевое имя файла 297
 fileno, метод 295
 FileTest, модуль
 blockdev?, метод 292
 chardev?, метод 292
 exist?, метод 291
 size?, метод 291
 symlink?, метод 293
 zero?, метод 291
 FileUtils, библиотека
 mkdirs, метод 304
 rm_r, метод, рекурсивное удаление каталогов 305
 Filter, класс, перенаправление ввода/вывода 483
 Fixnum 65, 158, 164
 Float, метод, преобразование строки в число 98
 flock, метод 286
 flunk, метод 534
 for, синтаксическая конструкция 71
 foreach, метод, обход каталога 304
 fork, метод
 манипулирование процессами 477
 создание потока 450
 forwardable, библиотека 370
 FreeRIDE 654
 ftype, метод 293
 FXRuby
 графические форматы 436
 диалоговые окна 432
 кнопки 425
 меню 427
 обзор 422
 переключатели 428
 пример 424
 сообщения 436
 списки 430
 текстовые поля 427

G

Gem-сервер 644
 get_config_key, метод 557
 getmyvar, метод 52
 gettext, библиотека
 справочники сообщений 151
 установка 151
 gid, метод, права владения файлом 289
 gregorian_leap?, метод 210

grpowned?, метод 289
 gsub, метод, подстановка строк 90

H

hexdigest, метод, вычисление MD5-свертки 104
 hook_method, метод 391

I

iconv, библиотека 148
 if, модификатор 41
 if, предложение 42
 elsif, ключевое слово 62
 then, ключевое слово 63
 IMAP, протокол 581
 include?, метод, поиск в строке 91
 included_modules, метод, получение списка включенных модулей 385
 index, метод, поиск в строке 91
 Inflate, класс, сжатие строк 101
 inherited, метод 392
 initialize, метод 72, 333
 initialize_copy, метод 346
 inner_product, метод 171
 ino, метод, получения индексного узла файла 293
 insert, метод, двоичные деревья 269
 inspect, метод, преобразование к виду для печати 352
 install, метод 299
 instance_methods, метод 385
 instance_of?, метод 340
 InstantRails 602
 Integer, метод, преобразование строки в число 98
 integrate, метод 180
 intersection, метод 261
 io/nonblock, библиотека 296
 ioctl, метод 295
 IOWA, каркас для разработки Web-приложений 617
 app.cnf, конфигурационный файл, IOWA 618
 mapfile.cnf, конфигурационный файл 619
 запросы 619
 компоненты, передача управления 621
 отслеживание состояния сеанса 619
 переменная oid 619
 смешение HTML и компонентов 619

установка 618
шаблоны 620
IP-адрес, сопоставление с регулярным
выражением 129
irb, интерактивный Ruby 647
завершение по клавише Tab 650
и отладка 541
IRC-каналы 659
is_a?, метод 340

J

jcode, библиотека 140
jcount, метод, интернационализация 141
jEdit, редактор 653
join, метод
 потоки 454
 путевые имена файлов 297
julian_leap?, метод 210

K

key?, метод, потоки 450
kill, метод
 манипулирование процессами 478
 потоки 452
kind_of?, метод 59, 340
KirbyBase 315
Komodo 655

L

LDAP (интерфейс с LDAP) 323
length, метод, строки 83
list, метод, потоки 457
ljust, метод, форматирование строк 87
Locomotive, Ruby On Rails 602
lstrip, метод, строки 96

M

m, модификатор многострочности 121
Madeleine, преобладающие объекты 313
main, метод, потоки 451
makedirs, метод, создание цепочки
 каталогов 304
marfile.cnf, конфигурационный файл,
 IOWA 619
Marshal, модуль 306, 346
 _dump, метод, нестандартный
 маршalling 307

 _load, метод, нестандартный
 маршalling 307
clone, метод, глубокое
 копирование 308
dup, метод 308
MatchData, объект 118
Math, модуль, управление пространством
 имен 51
Math.sqrt 178
mathn, библиотека 172
 простые числа 173
 разложение на простые
 множители 172
mbchar?, метод,
 интернационализация 141
MD5-свертка 104
Method, класс 367
method_added, метод 391
method_missing, метод 389
Microsoft Windows
 ActiveScriptRuby 492
 Win32API, библиотека 488
 Win32OLE, расширение 488
 моментальный инсталлятор 493
MIME, формат почтового сообщения 583
mktime, метод 203
mod_ruby 622
Mongrel 628
multiruby, инструмент (ZenTest) 538
MVC (Модель-Вид-Контроллер), паттерн
 проектирования, Nitro 604
MySQL (интерфейс с MySQL) 319
MysqlField, методы экземпляра 320, 321
myvar=, метод 52

N

nil, определение операторов 71
Nitro 603
 MVC, паттерн проектирования 604
Og 609
 классы контроллеров 606
 конвейерная компиляция 612
 кэширование действий 613
 локаль, определение 614
 поддержка Ajax 613
 поддержка DHTML 613
 подстановка содержимого в шаблон 614
 ресурсы в Web 615
 создание простого приложения 603
 части 612
 элементы 610



ObjectSpace, модуль 60
 oid, переменная (IOWA) 619
 Oniguruma, библиотека 123
 именованные соответствия 127
 кванторы 126
 новые возможности 125
 оглядывание назад 125
 проверка наличия 123
 рекурсивные регулярные выражения 128
 сборка 124
 Open-URI, библиотека 590
 Oracle (интерфейс с Oracle) 325

P

ParseDate, модуль 214
 Pathname, класс 297
 pid, метод 478
 POP, протокол 576
 popdir, метод 484
 PostgreSQL (интерфейс с PostgreSQL) 321
 ppid, метод 478
 priority, метод, потоки 453
 Proc, объект 366
 PStore, библиотека 308
 pushdir, метод 484

Q

QtRuby 436
 переключатели 442
 простое оконное приложение 437
 текстовые поля 439
 флажки 442

R

RAA (архив приложений Ruby) 559
 RadRails (Ruby on Rails) 603
 raise, метод, потоки 454
 rake, утилита
 действия 645
 задания 645
 Rational, класс 166
 rspec, утилита измерения покрытия кода 543
 RCR (запрос на изменение Ruby) 659
 RDE (Ruby Development Environment) 655
 RDoc
 простая разметка 552

 сложное форматирование 554
 теги 555
 RDT (Ruby Development Tool) 655
 readpartial, метод 296
 redo, ключевое слово, циклы 45
 rescue, ключевое слов, исключения 47
 retry, ключевое слово
 исключения 47
 циклы 45
 reverse, метод, строки 102
 REXML, разбор XML-файлов 504
 XPath 507
 древовидное представление 505
 поточковый 506
 Rinda, пространства кортежей 636
 Rinda::RingFinger, класс 641
 Rinda::RingProvider, класс 640
 Rinda::RingServer, класс 640
 ri (Ruby index), утилита 652
 rjust, метод, форматирование строк 87
 RMagick, библиотека
 API рисования 517
 обработка изображений 512
 масштабирование 514
 получение информации 512
 специальные эффекты
 и трансформации 515
 RSS, формат данных 510
 feedtools, библиотека 510
 rss, стандартная библиотека 508
 rstrip, метод, строки 96
 ruby-breakpoint, библиотека
 drb 542
 irb 541
 ruby-web, библиотека 593
 Ruby/GTK2, библиотека 409
 Ruby-GNOME2 420
 кнопки 412
 меню 418
 переключатели 418
 простое оконное приложение 411
 текстовые поля 413
 флажки 416
 Ruby/Tk 397
 кнопки 399
 переключатели 407
 простое оконное приложение 398
 списки 409
 текстовые поля 402
 флажки 406
 холст 409
 RubyForge 560

RubyGems 557
 запуск gem-сервера 644
 справочники сообщений 156
 установка 643
 установка пакетов 643
 Ruby on Rails 599
 InstantRails 602
 Locomotive 602
 MVC, паттерн проектирования 599
 RadRails 603
 TextMate 602
 базовые расширения 602
 стенд 601
 заглушки 601
 обстраивание 600
 операции со временем 602
 подключаемые модули 603
 прерыватель 601
 тестирование и отладка 601
S
 safe_level, метод 452
 safe_unlink, метод 299
 SciTe, редактор 653
 select, метод 295
 self, псевдопеременная 53
 set_config_key, метод 557
 setmyvar, метод 52
 Shell, библиотека
 Filter, класс 483
 popdir, метод 484
 pushdir, метод 484
 transact, метод 484
 перенаправление ввода/вывода 483
 SimpleDelegator, класс 370
 SingleForwardable, класс 372
 singleton_method_added, метод 392
 singleton_methods, метод 386
 SizedQueue, класс 463
 SMTP, протокол 578
 sort_by, метод 356
 SQLite (интерфейс с SQLite) 317
 start, метод, потоки 452
 STDERR, константа 287
 STDIN, константа 287
 STDOUT, константа 287
 stop, метод, потоки 452
 StringIO, библиотека 301
 StringScanner, класс 86
 Struct, класс 354
 super, ключевое слово 55

swapcase, метод, класс String 88
 sync.rb, библиотека, потоки 468
 syscall, метод 295

T

Tempfile, библиотека 303
 TextMate, редактор 602
 then, ключевое слово 63
 Time, класс
 Comparable, модуль 211
 методы
 mktime 203
 strftime 204
 Time.at 207
 Time.gm 203
 Time.utc 204
 to_a 204
 wday 204
 определение високосности года 210
 часовые пояса 210
 to_a, метод, дата и время 204
 to_ary, метод 352
 to_f, метод, преобразование строки
 в число 98
 to_hash, метод 314
 to_i, метод, преобразование строки
 в число 98
 to_int, метод, неявные преобразования 175
 to_s, метод 314
 преобразование системы счисления 177
 преобразования строк 92
 форматирование даты и времени 215
 to_set, метод 261
 to_str, метод 92
 tracer, библиотека 388
 transact, метод 484
 trap, метод 478
 TrueClass 64
 truncate, метод 299

U

UCS-2, кодировка 138
 uid, метод, права владения файлом 289
 UnboundMethod, объект 367
 undef_method, метод, удаление
 определений 383
 Unicode
 UTF-8, кодировка
 jcount, метод 141

mbchar?, метод 141
unpack, метод 142
байты и символы 139
библиотека jcode 140
глобальная переменная \$KCODE 139
квадратные скобки 142
конкатенация строк 140
перебор символов в строке 141
преобразование в другую
кодировку 148
распознавание кодировки 144
регулярные выражения 140, 143
байт, определение 137
глиф, определение 137
графема, определение 137
кодовая позиция, определение 137
семейство кодировок 137
составные и монолитные формы 141
строки
нормализация 144
упорядочение 146
терминология 136
union, метод, множества 261
unpack, метод
двоичные файлы 286
строки, интернационализация 142
upcase, метод, класс String 88
upto, метод, генерирование
последовательности строк 103
UTF-16, кодировка 138
UTF-8, кодировка. См. Unicode, UTF-8

V

vim, редактор 653

W

Web-серверы 622
erb 624
mod_ruby 623
Mongrel 628
WEBrick 626
Wee (Web Engineering Easy)
HTML-код 616
автоматическая перезагрузка
файлов 616
ассоциирование состояния с URL 616
компоненты 616
ресурсы в Web 617
Win32API, сценарии для Microsoft
Windows 488
Win32OLE, сценарии для Microsoft
Windows 488
Windows API 447
wxWidgets 446

X

XML, разбор 503
X Window System 446

Y

YAML (YAML Ain't Markup Language) 312
yield, ключевое слово 45, 70

Z

ZenTest. См. Тестирование, ZenTest
Zlib, библиотека 101

Программирование на языке Ruby

Ruby – прагматичный объектно-ориентированный язык, позаимствовавший лучшие особенности языков LISP, Smalltalk, Perl, C++ и других. За пять лет, прошедших с момента выхода первого издания этой книги, его популярность необычайно возросла.

Эта книга ответит на любой возникший у вас вопрос по программированию в Ruby. Приведено свыше 400 примеров, разбитых по темам. В каждом примере дается ответ на конкретный вопрос. Автор приводит описание задачи и обсуждает технические детали, а затем дает отличное решение с подробными пояснениями.

Рассматриваемые темы:

- Обзор терминологии и основных принципов языка Ruby
- Операции над низкоуровневыми типами данных (числа, строки, регулярные выражения, даты)
- Новая библиотека регулярных выражений (Oniguruma)
- Интернационализация и справочники сообщений в Ruby
- Операции над хэшами, массивами и другими структурами данных: стеками, очередями, деревьями и графами
- Общие принципы ввода/вывода, файлы и устойчивые объекты
- Базы данных, включая MySQL, SQLite, Oracle, DBI и другие
- Особенности Ruby в части объектно-ориентированного и динамического программирования
- Графические интерфейсы (Tk, GTK+, Fox и Qt)
- Многопоточное программирование в Ruby
- Написание рутинных сценариев и системное администрирование
- Работа с графическими форматами: PDF, YAML, XML, RSS
- Тестирование, отладка, профилирование и создание диспетчеризованных пакетов

Scanned: x-code

special for

infanata.org

Хэл Фултон больше 15 лет работал с различными версиями ОС UNIX, в том числе AIX, Solaris и Linux. С языком Ruby впервые познакомился в 1999 году, а в 2001 приступил к работе над первым изданием этой книги – второй книги на английском языке, посвященной Ruby. Он присутствовал на многих конференциях по Ruby, не раз выступал, в том числе и на первой европейской конференции по языку Ruby, состоявшейся в Карлсруэ в Германии. Обладатель двух ученых степеней по информатике, полученных в Университете штата Миссисипи. Четыре года преподавал информатику в колледже, пока не переехал в Остин, штат Техас, для работы по контрактам с различными компаниями, в частности, с отделением IBM в Остине. Сейчас он работает в компании Broadwing Communications, базирующейся в Остине и занимающейся вопросами, связанными с большим хранилищем данных и относящимися к нему телекоммуникационными приложениями. В работе он использует язык C++, СУБД Oracle и, конечно, Ruby.

Internet-магазин: www.aliants-kniga.ru

Книга – почтой:

Россия, 123242, Москва, а/я 20

e-mail: books@aliants-kniga.ru

Оптовая продажа: «Альянс-книга»

Тел./факс: (495) 258-9195

e-mail: books@aliants-kniga.ru

ISBN 5-94074-357-9



9 785940 743576



DMK
ПРЕСС