

Ruby

从入门到精通

Beginning Ruby: From Novice to Professional



(美) Peter Cooper 著
仲田 等译

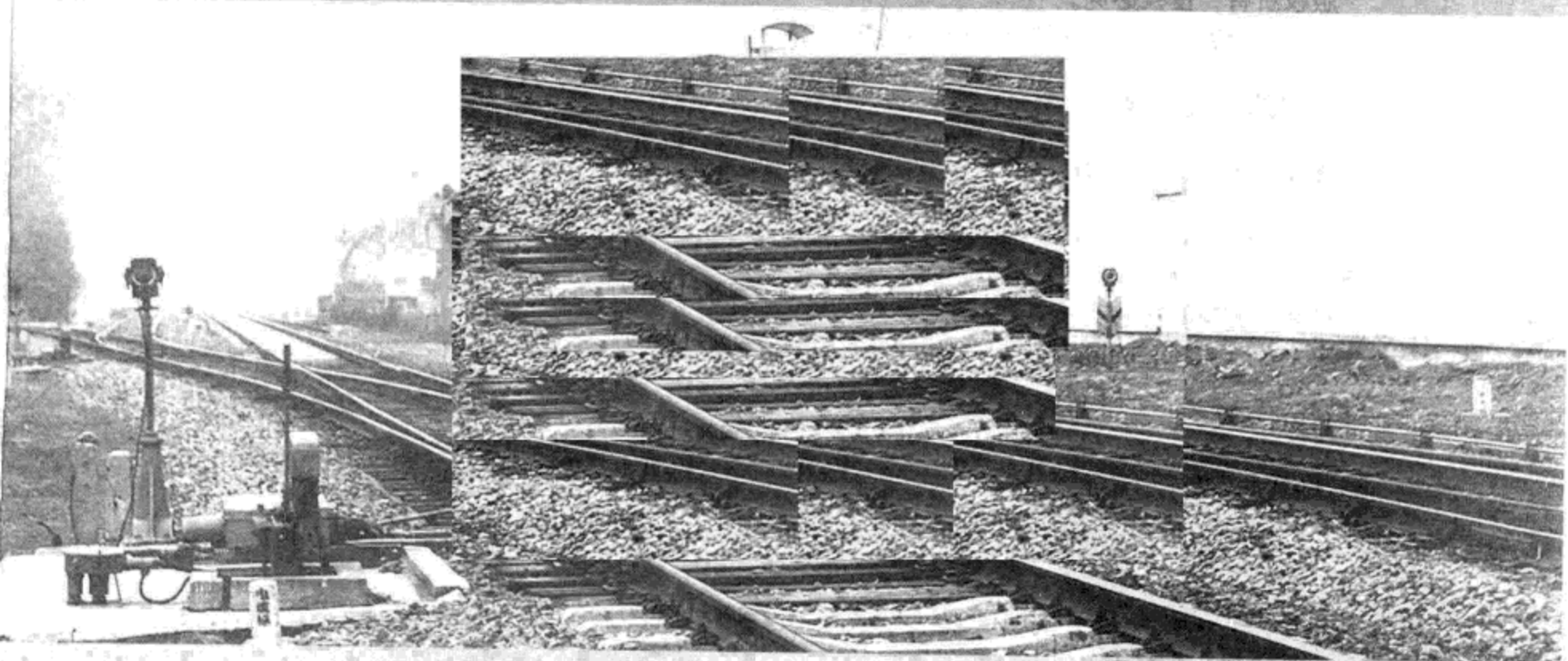


机械工业出版社
China Machine Press

Ruby

从入门到精通

Beginning Ruby: From Novice to Professional



(美) Peter Cooper 著
仲田 等译



机械工业出版社
China Machine Press

本书深入浅出地介绍了Ruby编程语言。全书分为三篇：第一篇介绍编程以及Ruby的基本概念，并用简单例子快速引导读者开发真正的Ruby应用程序；第二篇讲解Ruby语言的核心概念和语法，并综合这些概念和语法，以机器人小程序为例进行了实战开发；第三篇深入讲解Ruby on Rails开发，以及怎样用Ruby访问因特网和网络连接服务，最后还对大量非常有用的Ruby程序库和gem包作了简要介绍。本书最后给出三个附录，为有经验的开发人员提供了Ruby快速入门参考，为本书读者提供了Ruby语法参考索引，并介绍了可用于进一步学习Ruby的各种网络资源。

本书适合Ruby初学者、Web开发人员参考。

Beginning Ruby: From Novice to Professional (ISBN: 1-59059-766-4).

Original English language edition published by Apress L.P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA. Copyright © 2008 by Apress L.P. Simplified Chinese-language edition copyright © 2009 by China Machine Press. All rights reserved.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由Apress出版社出版。

本书简体字中文版由Apress出版社授权机械工业出版社独家出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括中国香港、台湾、澳门地区）销售发行，未经授权的本书出口将被视为违反版权法的行为。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2008-1858

图书在版编目（CIP）数据

Ruby从入门到精通 / (美) 库珀 (Cooper, P.) 著；仲田等译. —北京：机械工业出版社，2009.1

(Ruby和Rails技术系列)

书名原文：Beginning Ruby: From Novice to Professional

ISBN 978-7-111-25866-7

I. R… II. ① 库… ② 仲… III. 计算机网络—程序设计 IV. TP393.09

中国版本图书馆CIP数据核字 (2008) 第201312号

机械工业出版社 (北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑：王春华

北京诚信伟业印刷有限公司印刷

2009年2月第1版第1次印刷

186mm × 240mm · 25印张

标准书号：ISBN 978-7-111-25866-7

定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010) 68326294

译者序

几年前刚通过Python接触到Ruby时，我的第一感觉就是，和传统语言相比，Python已经够好了，但是Ruby比Python还好！它的语法简单易懂，灵活多变，而且实现了真正纯粹的面向对象，在Ruby中一切都是对象（例如`1.upto(10)`这种在传统语言中不可想像的语法）。它在程序语言设计的前人经验积累基础上，进行了大胆的组合与创新，已从量变到质变，达到了一个全新的高度。一直以来，我有一种观点：编程语言应该面向程序员，尽量为程序员提供便利，不能只为了机器编译的方便，而让程序员背上不必要的学习和使用负担。Ruby正是这样一门语言，它让我有一种感觉：我找到了！

Ruby语言自从诞生以来，由于没有“杀手级”应用，一直默默无闻地在小范围内传播，未得到广泛注意。直到2004年末，Ruby on Rails横空出世，世人这才惊觉，在耀眼夺目的Rails背后，有如此强大的Ruby。原来Ruby可以这么用，原来Ruby可以这么强！可以说，没有强大灵活的Ruby，就没有一鸣惊人的Rails！

值得一提的是，教授语言的大师Bruce Eckel（即《Thinking in C++》、《Thinking in Java》等获奖名著的作者）和面向对象设计大师Martin Fowler（即《Patterns of Enterprise Application Architecture》、《Refactoring》、《UML Distilled》、《Planning Extreme Programming》等获奖名著的作者）都对Ruby推崇备至，自从Ruby出现后，他们也像常人一样，从Python移情别恋到Ruby。

本书是一本覆盖全面且浅显易读的Ruby入门书籍，内容分为三篇。

第一篇是基础篇，介绍了编程以及Ruby的基本概念，并用简单例子快速引导读者开发真正的Ruby应用程序，另外，还介绍了Ruby的安装方法和发展历史。

第二篇是核心篇，讲解了Ruby语言的核心概念和语法，包括类、对象、模块、项目与程序库、文档化、出错处理和测试、文件与数据库、应用部署、高级功能等，最后综合这些概念和语法，以机器人小程序为例进行了实战开发。

第三篇是高级篇，深入讲解了炙手可热的Ruby on Rails开发，以及怎样用Ruby访问因特网和网络连接服务，最后还对大量非常有用的Ruby程序库和gem包进行了简要介绍。

本书末尾包括三个附录，为有经验的开发人员提供了Ruby快速入门参考，为本书读者提供了Ruby语法参考索引，并介绍了可用于进一步学习Ruby的各种网络资源。

本书由仲田、顾娟、吴畏和汪燕翻译。其中仲田负责翻译第6~16章和附录，顾娟负责翻译第1章和第2章，吴畏负责翻译第3章和第4章，汪燕负责翻译第5章，最后由仲田统稿。

翻译的过程也是学习的过程，本书让我弥补了许多知识点上的不足，希望也能让你感觉耳目一新，从中受益。

译者

2008年9月

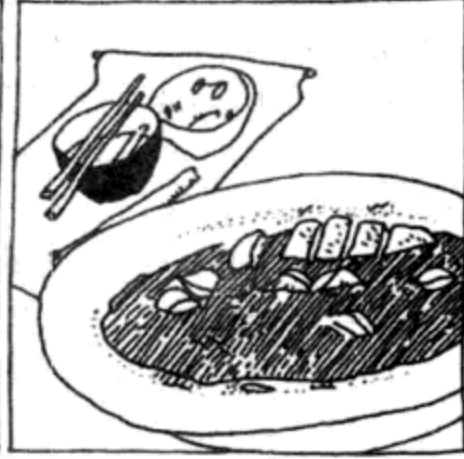
序 言



here is a nice mountain in japan.

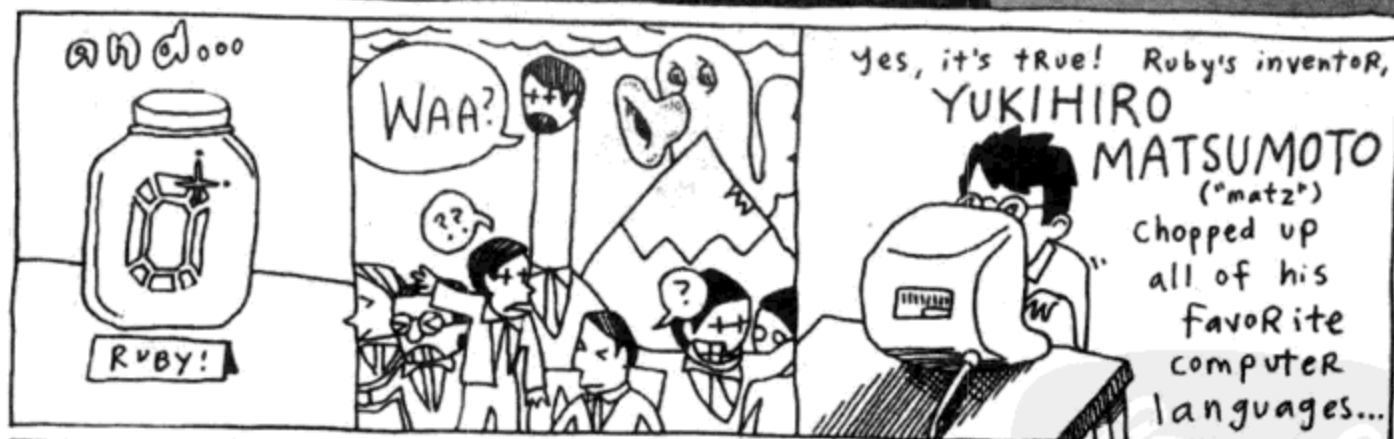
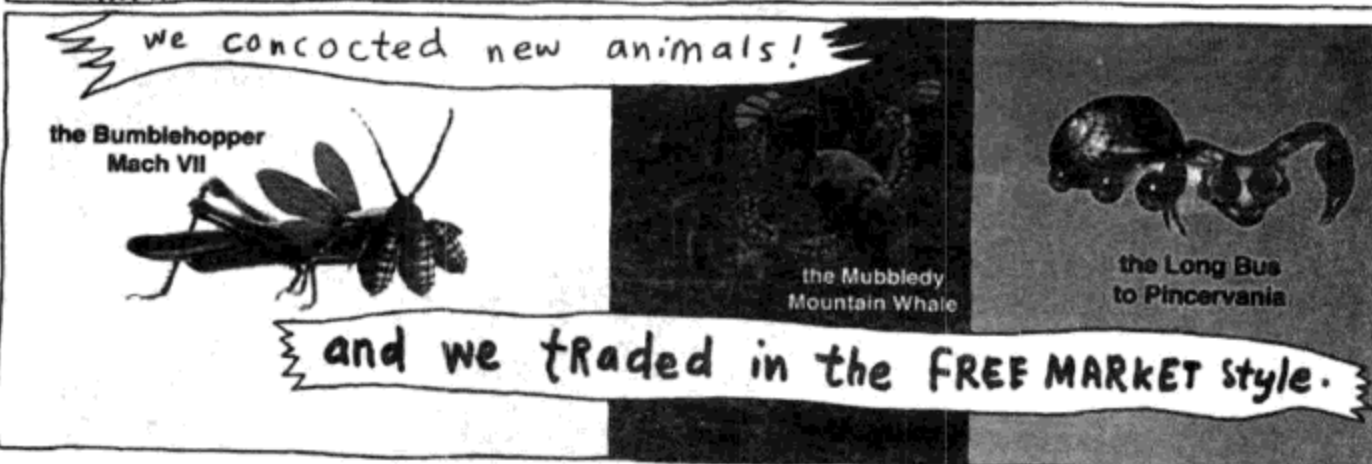
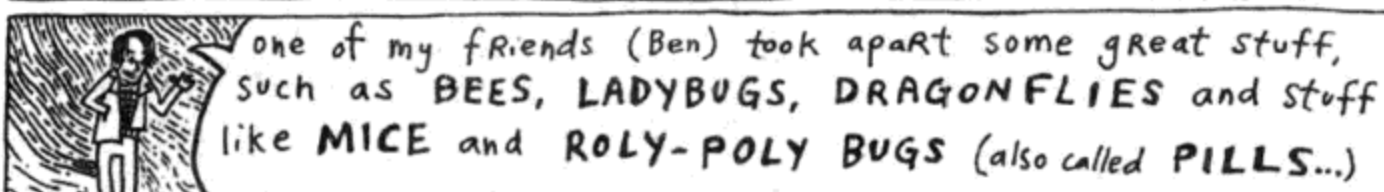


this one's just a meadow with a horse taking it easy.



oh, nice! a bowl of CURRY!
⚠️ so delicious...







why the lucky stiff

<http://whytheluckystiff.net/>

前言

我想最大程度地减少编程的难度，因此想最大程度地减少编程的劳动。这就是我设计Ruby的主要目标。我想让自己快乐地编程。

——松本行弘 (Matz)，Ruby之父

Ruby是“最好的那类”语言，它汇集了前辈语言最好和最强大的编程特点。

——Jim White

Ruby让我微笑。

——Amy Hoy (slash7.com)

Ruby是一个有趣的玩具，也是一门严肃的编程语言。Ruby是逗孩子们开心的乐呵呵大叔，但它也扎扎实实地每天花12小时泡在工地上。对千百万程序员来说，Ruby已经是个好朋友，是个值得依赖的服务员，而且它揭示了编程和软件开发的一种新思路。

就像吉他一样，Ruby这门语言常常得到“简单易学但很难精通”的评价。在一定前提下，我同意这种说法。如果你还不懂任何编程语言，Ruby就会令人惊讶地简单易学。如果你已经了解某些语言，例如PHP、Perl、BASIC、C或Pascal，就会很熟悉Ruby的某些概念，但Ruby对于解决问题所持的不同视角，可能会让你一时迷惑不已。如同人们交谈所用的各种语言的区别一样，Ruby与大多数其他编程语言的区别，不仅在于句法，更在于文化、语法和惯例。事实上，Ruby更接近于小众语言（如LISP和Smalltalk），而不是为人熟知的语言（如PHP和C++）。

虽然Ruby的根源可能与其他语言不同，但它在许多行业得到了广泛的应用和重视。以这样那样的方法使用或支持Ruby的公司中，有许多公司的名字如雷贯耳，例如Sun公司、英特尔公司、微软公司、苹果公司和亚马逊网站。Web框架Ruby on Rails是一套用来开发Web应用程序的系统，它以Ruby作为基础语言，目前已是成百上千个大型网站的支柱。Ruby还作为命令行方式的通用语言使用，在这方面更像Perl。语言学家、生物学家、数据库管理员以及千百类其他专业人士和业余人士，都用Ruby简化自己的工作。Ruby是真正的国际化语言，有几乎无限的应用。

本书是为了满足编程新手和有其他语言编程经验的编程人员的需要，由于Ruby的文化与其他语言有太大的不同，因此本书大部分内容将对这两类读者都有用。如果某些大段章节对熟练的程序员来说可以直接跳过，正文中将予以注明。在任何情况下，我都建议所有程序员至少快速浏览一遍自认为明白的章节，因为Ruby有许多令人惊奇的方法，与你以前的做法截然不同。

在阅读本书时，请做好心理准备，以便面对一些非正式的、有些奇怪的示例，以及大量实用主义的做法。Ruby是一种极其实用的语言，不太注重正规形式，而更注重简化开发和有效结果。我将不时展示怎样以“错误的”方式使用Ruby（仅仅是为了示例的目的），但大多数情况下，你将看到“以Ruby方式”完成任务的代码。在我开始学习Ruby时，我主要通过例子来学习，而

对于Ruby这么原创且充满惯用法的语言，这是为未来养成良好习惯的最简单方法。然而总是“有不只一种做事方法”，因此，如果你觉得本书某些代码可以用另一种方法重写，以便更适应你的思路，请你尽管尝试！

在开始阅读本书时，请做好心理准备，你将以全新的方式思考，并将由于既有趣又有益的Ruby，而产生编写代码的冲动。Ruby帮助了许多疲惫不堪的开发人员，让他们再次成为高效的程序员，因此，不管你是个编程新手还是这些疲劳者之一，都会发现Ruby既有趣又高效，这几乎是不可避免的。

最后，如果你来自现代脚本语言的阵营，例如Perl、PHP或Python，你可以在阅读第1章之前跳到附录A，它涵盖了Ruby和其他脚本语言的关键区别，或许有助于你更轻松地阅读本书的开头几章。

祝你好运，希望你享受本书的阅读之旅。我们第1章再见。



致 谢

常言道，写书是件孤独的事，但直到你自己写书才会发现，这个过程五味杂陈，但绝不是孤独。没有大型团队在背后对本书、对我个人的帮助和支持，我无法完成本书。

我首先要感谢Keir Thomas，是他建议我写一本关于Ruby的书。在书的范围和内容方面，他给了我巨大的自由。他还是让本书顺利出版的最大功臣。

我要特别感谢Apress出版公司的Beth Christmas，感谢她一流的项目管理能力和在写作本书期间持之以恒的支持。没有她的日程计划和对每件事按期完成的保证，我会变成紧张情绪的牺牲品。

Jonathan Gennick、Tim Fletcher和Peter Marklund值得高度赞扬，他们在开发工作的各个阶段，似乎永无休止地阅读、再阅读本书的各个章节。作为Ruby新手，Jonathan提供了一些特别有趣的观点，让本书更适合Ruby新手阅读。

我还要感谢Susannah Davidson Pfalzer，她费尽心血地编辑本书，修正我用的介词，删除我过度使用的“然而”、“因此”之类词语，并让本书变得轻松可读，让读者不致因为本书艰涩拗口而疯狂。由于这是我在Apress出版的第一本书，我极度依赖Susannah关于Apress习俗惯例的深厚知识。

自然，我还要感谢写作本书期间与我直接共事的所有人，不管他们是Apress公司职员还是个人。以下人名未特别排序：Jonathan Gennick、Keir Thomas、Beth Christmas、Tim Fletcher、Peter Marklund、Susannah Davidson Pfalzer、Jason Gilmore、Lori Bring、Nancy Sixsmith以及why the lucky stiff。

在本书之外，我还要感谢Ruby社区的许多人，感谢他们与我共事，或开发我所用的工具，或让Ruby语言变得更有魅力。以下人名未特别排序：why the lucky stiff（感谢其令人难忘的序言）、松本行弘（“Matz”）、Jamie van Dyke、Amy Hoy、Evan Weaver、Geoffrey Grosenbach、Obie Fernandez、Damien Tanner、Chris Roos、Martin Sadler、Zach Dennis、Pat Toner、Pat Eyler、Hendy Irawan、Ian Ozsvald、Nic Williams、Shane Vitarana、Josh Catone、Alan Bradburne、Jonathan Conway、Alex MacCaw、Benjamin Curtis和David Heinemeier Hansson。我很担心是不是遗漏了一些名字，如果上述名单漏掉了你的名字，我表示深深的歉意。

在私人生活中有很多人给了我很大支持，他们忍受我怪异的工作时间和烦人习惯，对本书提出问题，供我吃喝，或只是默默倾听，出于这一考虑，我想感谢（同上，人名未特别排序）Laura Craggs、Clive Cooper、Ann Cooper、David Sculley、Ed Farrow、Michael Wong、Bob Pardoe、Dave Hunt、Chris Ueland、Kelly Smith、Graham Craggs、Lorraine Craggs和Robert Smith。特别要感谢Laura Craggs，在写作本书期间，她不得不每天几乎24小时忍受我，她太了不起了。

衷心地感谢John Joyce，他仔细阅读了全书并给出勘误表。

最后，我必须感谢的是读者朋友，感谢你选择购买本书，因为如果没有人购买，这些感谢和这么多人在写作本书期间付出的辛勤劳动，都会白白浪费。感谢你！

作者简介

Peter Cooper 是经验丰富的Ruby开发者和培训师，还是最流行的Ruby新闻博客“Ruby内幕”(<http://www.rubyinside.com/>)的编辑。在2007年以前，他主要做Ruby培训和开发，现在是Feed Digest网站(<http://www.feeditdigest.com/>)的全职开发者和所有者，该网站是基于Ruby和Rails的RSS 订阅源(feed)处理和再分发服务，每月点击量超过2亿次，最近接受了《商业2.0》杂志的专访。

从2004年以来，Peter用Ruby和基于Ruby的Web框架Rails开发了许多商业网站。另外，他创建了Code Snippets网站(<http://www.bigbold.com/snippets/>)，该网站是因特网上最大的公共代码仓库；还创建了Congress（运用Ajax和Ruby on Rails技术的在线聊天客户端）。

在开发工作之余，自1998年以来，Peter还撰写了100多本关于各种开发技术和工具的专业书籍。他是WebDeveloper.com网站的编辑，并在因特网浪潮中，参与过iBoost.com和Webpedia.com网站的开发。

技术评审者简介

Tim Fletcher 是IBM公司的实习生。他喜欢Ruby语言，因为它很有趣。他没有孩子，也没有宠物，但有一个可敬的妹妹，名叫Sophie。当不写代码时，他喜欢阅读、吃东西、睡觉，以及尽可能多地滑雪。

Peter Marklund 是面向对象、Web开发、关系型数据库和测试方面的专家，有着丰富的经验。自2000年以来他一直用Java和Tcl语言做Web开发，是开源Web框架OpenACS的核心开发人员之一。2004年下半年他引入了Ruby on Rails，并用Rails完成了某个CRM系统和在线社区的开发。他是Ruby on Rails的自由作家，也帮助组织斯德哥尔摩的Ruby on Rails开发者社区的活动。他在<http://marklunds.com>网站有个私人博客，用于与其他开发者分享Rails开发建议。

译者简介

仲田 南京某软件公司项目经理，高级程序员、系统分析员，有多年软件开发与管理经验，从事过Delphi、J2EE、Rails应用开发，应用领域主要是企业管理应用，包括财务、审计、法律、商务、办公自动化等。目前正在研究Ruby语言和Rails框架。译者电子邮箱：ztian@163.com。

目 录

译者序
序言
前言
致谢

第一篇 基础与脚手架

第1章 让它跑起来：安装Ruby	1
1.1 安装Ruby	2
1.1.1 Windows平台	2
1.1.2 Apple Mac OS X平台	4
1.1.3 Linux平台	5
1.1.4 其他平台	7
1.2 小结	8
第2章 编程等于快乐：Ruby和面向对象 概览	9
2.1 初始步骤	9
2.1.1 irb：交互式Ruby	9
2.1.2 Ruby是计算机的通用语	10
2.1.3 为什么Ruby是如此杰出的编程语言	10
2.1.4 心灵小径	11
2.2 把思路转变成Ruby代码	13
2.2.1 Ruby怎么理解对象和类的概念	13
2.2.2 造人过程	13
2.2.3 基础变量	15
2.2.4 从人到宠物	15
2.3 一切都是对象	18
2.3.1 Kernel模块的方法	19
2.3.2 向方法传递数据	19
2.3.3 使用String类的方法	20
2.4 以非面向对象方式使用Ruby	21
2.5 小结	22
第3章 Ruby的构造元素：数据、表达式 和流程控制	24
3.1 数字与表达式	24

3.1.1 表达式基础知识	24
3.1.2 变量	24
3.1.3 比较运算符与表达式	25
3.1.4 用块和迭代子在数字中循环	27
3.1.5 浮点数	28
3.1.6 常量	29
3.2 文本与字符串	30
3.2.1 字面字符串	30
3.2.2 字符串表达式	31
3.2.3 插写	32
3.2.4 字符串方法	33
3.2.5 正则表达式与字符串操作	34
3.3 数组与列表	38
3.3.1 基本数组	38
3.3.2 字符串切分成数组	40
3.3.3 数组迭代	40
3.3.4 数组的其他方法	41
3.4 散列表	43
3.4.1 散列表的基础方法	43
3.4.2 散列表中的散列表	44
3.5 流程控制	45
3.5.1 if与unless	45
3.5.2 ?:, 三元运算符	46
3.5.3 elsif与case	47
3.5.4 while与until	48
3.5.5 代码块	49
3.6 其他有用的构造元素	51
3.6.1 日期与时间	51
3.6.2 大数字	53
3.6.3 范围	54
3.6.4 符号	55
3.6.5 类间转换	56
3.7 小结	57

第4章 开发基础的Ruby应用程序	59
4.1 处理源代码文件	59
4.1.1 创建测试文件	59
4.1.2 测试用源代码文件	60
4.1.3 运行源代码	61
4.2 我们的目标程序：文本分析器	63
4.2.1 基本功能需求	63
4.2.2 构建程序基本框架	64
4.2.3 获取哑文本	64
4.2.4 载入文本文件并统计行数	65
4.2.5 统计字符数	66
4.2.6 统计字数	66
4.2.7 统计句子和段落数	68
4.2.8 计算平均值	69
4.2.9 到目前为止的源代码	69
4.3 增加额外功能	70
4.3.1 “有用”字词的百分比	70
4.3.2 找出“有趣的”句子进行汇总	72
4.3.3 分析text.txt之外的其他文件	73
4.4 完整的程序	74
4.5 小结	76
第5章 Ruby生态系统	77
5.1 Ruby的历史	77
5.1.1 Ruby的起源	77
5.1.2 Ruby的影响	78
5.1.3 向西方流传	78
5.2 Ruby on Rails	80
5.2.1 Rails面世的由来	80
5.2.2 Web (2.0) 是怎样赢的	81
5.3 开源文化	82
5.4 如何获得帮助	83
5.4.1 邮件列表	83
5.4.2 Usenet新闻组	83
5.4.3 因特网中继聊天工具	83
5.4.4 文档	84
5.4.5 论坛	85
5.5 加入社区	85

5.5.1 向别人提供帮助	85
5.5.2 贡献代码	86
5.5.3 网络博客	86
5.6 小结	87

第二篇 Ruby的核心

第6章 类、对象和模块	89
6.1 为什么要用面向对象	89
6.2 面向对象基础知识	92
6.2.1 局部变量、全局变量、对象变量和类变量	92
6.2.2 类方法和对象方法	95
6.2.3 继承	97
6.2.4 覆写现有方法	99
6.2.5 对象方法的反射与发现	101
6.2.6 封装	102
6.2.7 多态	106
6.2.8 嵌套类	107
6.2.9 常量的作用域	108
6.3 模块、命名空间和掺入	109
6.3.1 命名空间	109
6.3.2 掺入	111
6.4 用对象构建“地下城”文本冒险游戏	117
6.4.1 地下城的概念	117
6.4.2 创建初始类	118
6.4.3 Structs：快捷简单的数据类	119
6.4.4 创建房间	121
6.4.5 让地下城运转起来	122
6.5 小结	125
第7章 项目与程序库	127
7.1 项目和使用其他文件的代码	127
7.1.1 基本的文件包含	127
7.1.2 从其他目录包含	129
7.1.3 有条件地包含代码	129
7.1.4 嵌套包含	130
7.2 程序库	130
7.2.1 标准程序库	131

7.2.2 RubyGems包	133	第10章 部署Ruby应用和程序库	194
7.3 小结	139	10.1 简单Ruby程序发布	194
第8章 文档编写、错误处理、调试和测试	140	10.1.1 shebang行	195
8.1 文档编写	140	10.1.2 关联Windows的文件类型	196
8.1.1 用RDoc生成文档	140	10.1.3 “编译”Ruby程序	196
8.1.2 RDoc技术	142	10.2 检测Ruby运行环境	197
8.2 调试与出错	144	10.2.1 用RUBY_PLATFORM作简单的操作系统检测	198
8.2.1 异常和出错处理	145	10.2.2 环境变量	198
8.2.2 Catch与Throw方法	147	10.2.3 读取命令行参数	200
8.2.3 Ruby调试器	148	10.3 以gem包形式发布Ruby程序库	200
8.3 测试	151	10.3.1 创建gem包	201
8.3.1 测试驱动开发的哲学	151	10.3.2 发布gem包	204
8.3.2 单元测试	153	10.3.3 RubyForge网站	204
8.3.3 更多的Test::Unit断言	154	10.4 以远程服务形式部署Ruby应用	205
8.4 性能基准度量和优化分析	155	10.4.1 CGI脚本	205
8.4.1 性能基准简单度量	156	10.4.2 常见HTTP服务器	207
8.4.2 性能优化分析	157	10.4.3 远程方法调用	210
8.5 小结	159	10.5 小结	214
第9章 文件和数据库	161	第11章 Ruby高级功能	216
9.1 输入与输出	161	11.1 动态代码执行	216
9.1.1 键盘输入	161	11.1.1 绑定	216
9.1.2 文件输入输出	162	11.1.2 eval的其他形式	217
9.2 数据库基础	173	11.1.3 创建attr_accessor	219
9.2.1 文本文件数据库	174	11.2 从Ruby中运行其他程序	220
9.2.2 对象和数据结构的存储	176	11.2.1 获得其他程序的运行结果	220
9.3 关系型数据库与SQL	179	11.2.2 向其他程序移交执行权	221
9.3.1 关系型数据库概念	179	11.2.3 同时运行两个程序	221
9.3.2 四大数据库: MySQL、PostgreSQL、Oracle和SQLite	180	11.2.4 与另一程序交互	222
9.3.3 安装SQLite	180	11.3 安全地掌控数据和危险方法	222
9.3.4 关于数据库基本操作和SQL的紧急教程	181	11.3.1 被感染的数据和对象	223
9.3.5 在Ruby中使用SQLite	184	11.3.2 安全级别	224
9.3.6 连接其他数据库系统	188	11.4 使用微软Windows	225
9.3.7 ActiveRecord简介	192	11.4.1 使用Windows API	225
9.4 小结	192	11.4.2 控制Windows程序	227
		11.5 线程	228
		11.5.1 基础Ruby线程实战	228

11.5.2 高级线程操作	229	13.1.2 安装Rails	272
11.6 其他语言嵌入Ruby	230	13.1.3 数据库方面的考虑	273
11.6.1 为什么用C作为嵌入语言	231	13.2 构建Rails简单应用	273
11.6.2 创建基础方法或函数	231	13.2.1 创建Rails空白应用	273
11.6.3 性能基准度量: C和Ruby	233	13.2.2 数据库初始化	277
11.7 对Unicode和UTF-8的支持	234	13.2.3 创建模型和迁移文件	279
11.8 小结	236	13.2.4 搭建脚手架	282
第12章 综合演练: 开发更大型的Ruby		13.2.5 控制器与视图	285
应用	238	13.2.6 路由	292
12.1 构建机器人小程序	238	13.2.7 模型间关系	293
12.1.1 什么是机器人小程序	238	13.2.8 会话与过滤器	295
12.1.2 为什么要构建机器人小程序	239	13.3 其他功能特性	296
12.1.3 怎样构建	239	13.3.1 界面布局	296
12.2 创建文本处理工具程序库	239	13.3.2 测试	298
12.2.1 构建WordPlay程序库	240	13.3.3 插件	299
12.2.2 测试该程序库	245	13.4 参考资料与演示应用	300
12.2.3 WordPlay程序库的源代码	247	13.4.1 参考站点和教程	300
12.3 构建机器人小程序的核心功能	249	13.4.2 Rails示例应用	300
12.3.1 程序的生命周期和组成部分	250	13.5 小结	301
12.3.2 机器人小程序的数据	250	第14章 Ruby与因特网	302
12.3.3 构建Bot类和数据载入器	254	14.1 HTTP与万维网	302
12.3.4 response_to方法	255	14.1.1 下载网页	302
12.3.5 试用机器人小程序	259	14.1.2 生成网页和HTML	309
12.4 机器人小程序主要代码清单	262	14.1.3 解析网页内容	313
12.4.1 bot.rb文件	263	14.2 电子邮件	317
12.4.2 basic_client.rb文件	265	14.2.1 用POP3协议接收邮件	317
12.5 扩展机器人小程序的功能	266	14.2.2 用SMTP协议发送邮件	319
12.5.1 用文本文件作为会话来源	266	14.2.3 用ActionMailer发送邮件	320
12.5.2 把机器人小程序连接到万维网	266	14.3 用FTP协议传输文件	321
12.5.3 机器人小程序之间的会话	269	14.3.1 FTP连接与基本操作	321
12.6 小结	270	14.3.2 下载文件	323
		14.3.3 上传文件	324
		14.4 小结	325
		第15章 网络连接、套接字与后台进程	326
		15.1 网络连接的概念	326
		15.1.1 TCP和UDP协议	326
		15.1.2 IP地址和DNS	327

第三篇 Ruby在线

第13章 Ruby on Rails: Ruby的杀手级

应用

13.1 第一步

13.1.1 Rails是什么, 为什么要用它

15.2 网络操作基础	327	16.6.3 更多信息	357
15.2.1 检查机器和服务是否可用	327	16.7 English程序库	357
15.2.2 进行DNS查询	328	16.7.1 安装	357
15.2.3 直接连接到TCP服务器	330	16.7.2 示例	358
15.3 服务器和客户端	332	16.7.3 更多信息	359
15.3.1 UDP客户端和服务端	332	16.8 ERB程序库	359
15.3.2 构建简单的TCP服务器	333	16.8.1 安装	359
15.3.3 多客户端TCP服务器	335	16.8.2 示例	359
15.3.4 GServer程序库	336	16.8.3 更多信息	361
15.3.5 基于GServer的聊天服务器	339	16.9 FasterCSV程序库	361
15.3.6 Web/HTTP服务器	341	16.9.1 安装	361
15.3.7 后台进程	341	16.9.2 示例	362
15.4 小结	343	16.9.3 更多信息	365
第16章 有用的Ruby程序库和gem包	344	16.10 iconv程序库	366
16.1 abbrev程序库	344	16.10.1 安装	366
16.1.1 安装	344	16.10.2 示例	366
16.1.2 示例	344	16.10.3 更多信息	367
16.1.3 更多信息	345	16.11 logger程序库	367
16.2 base64程序库	345	16.11.1 安装	367
16.2.1 安装	346	16.11.2 示例	367
16.2.2 示例	346	16.11.3 更多信息	369
16.2.3 更多信息	347	16.12 pp程序库	369
16.3 BlueCloth程序库	348	16.12.1 安装	369
16.3.1 安装	348	16.12.2 示例	369
16.3.2 示例	348	16.12.3 更多信息	370
16.3.3 更多信息	349	16.13 RedCloth程序库	371
16.4 cgi程序库	349	16.13.1 安装	371
16.4.1 安装	350	16.13.2 示例	371
16.4.2 示例	350	16.13.3 更多信息	372
16.4.3 更多信息	353	16.14 StringScanner程序库	372
16.5 chronic程序库	354	16.14.1 安装	372
16.5.1 安装	354	16.14.2 示例	373
16.5.2 示例	354	16.14.3 更多信息	375
16.5.3 更多信息	355	16.15 tempfile程序库	375
16.6 Digest程序库	355	16.15.1 安装	375
16.6.1 安装	355	16.15.2 示例	375
16.6.2 示例	356	16.15.3 更多信息	377

16.16 uri程序库	377
16.16.1 安装	377
16.16.2 示例	377
16.16.3 更多信息	380
16.17 zlib程序库	380
16.17.1 安装	380
16.17.2 示例	380

16.17.3 更多信息	381
--------------------	-----

附 录^①

附录A Ruby入门与回顾（开发人员
专用版）

附录B Ruby参考速查

附录C 有用的资源

^① 附录部分请到华章网站（www.hzbook.com）下载。



第一篇 基础与脚手架

本篇是Ruby的基础知识。学完本篇，你将能开发完整的（尽管是基本的）Ruby程序。本章主要介绍怎样让Ruby运行，面向对象的概念，怎样开发基本的程序，以及Ruby使用哪些数据类型，操作哪些控制结构。最后，我将带你从头至尾创建一个小程序。

第1章 让它跑起来：安装Ruby

Ruby是一门流行的编程语言，但很少有计算机默认支持它。为了让Ruby在计算机上运行，本章给出了必需的步骤。

作为开源语言，Ruby被转换（或用技术术语“移植”）为可以在许多不同计算机平台和架构体系上运行的语言。这表示如果你在某台机器上开发Ruby程序，那么应该也可以在别的机器上运行，无须作任何修改。在下列操作系统和平台上使用Ruby（以这样或那样的形式）：

- 微软Windows 95、98、XP和Vista（包含所有变体版本）
- Mac OS X（包含所有变体版本）
- Linux（包含所有变体版本）
- MS-DOS
- BSD（包含FreeBSD和OpenBSD）
- BeOS
- Acorn RISC OS
- OS/2
- Amiga
- Symbian Series 60手机
- 有Java虚拟机的任何平台（使用JRuby，而不是官方的Ruby解释器）

警告 Ruby的某些规格要求在不同平台上是有变化的，但本书大多数代码（特别是前面几章的代码）在所有版本中都可以运行。当我们开始考察更复杂的代码时，例如外部程序库，以及Ruby和其他系统的接口，你应该作好准备，对代码进行修改（以适应不同平台），或接受无法使用Ruby所有功能特性的现实。但如果你在x86体系架构下使用Windows、Linux或Mac OS X操作系统，则几乎所有代码都可以如本书所述地正常运行。

在开始把玩Ruby之前，你需要在计算机上安装Ruby的实现版本，以便让计算机支持Ruby语言，这是我们首先要讨论的主题。

1.1 安装Ruby

一般情况下，当你在计算机上安装Ruby时，安装的是“Ruby解释器”。它是一套程序，其功能是理解以Ruby编写的其他程序，并提供丰富的功能扩展和程序库，让你的Ruby功能更全面。但有些安装程序，例如本节谈到的Windows安装程序，还包含源代码编辑器和更容易阅读的文档，这是其他安装程序可能没有的。幸运的是，某版本包含而其他版本不包含的所有额外功能，都可以在以后单独安装。

为了让大部分读者不必再参考额外文档，我提供了在Windows、Mac OS X和Linux平台上安装和使用Ruby的全套指导说明，以及其他平台Ruby实现的链接。对于每种平台，我都提供了检查安装是否成功的指导说明，以便在进入第2章之前检查确认。

1.1.1 Windows平台

Ruby的初始设计目标是在UNIX和UNIX类操作系统（例如Linux）中使用的，但Windows用户也能享用到出色的“一键安装程序”，只需“点击一次鼠标键”，即可把Ruby、一组功能扩展、源代码编辑器和各种文档统统安装完毕。Ruby在Windows和其他操作系统中一样可靠和好用，而且Windows也为开发Ruby程序提供了一个良好的环境。

想尽快安装并运行Ruby，可遵循如下步骤：

1. 启动Web浏览器，访问<http://www.ruby-lang.org/en/downloads/>网址。
2. 下拉滚动条到“Ruby on Windows”，位置大约在整個页面的中部。
3. 在“Ruby on Windows”部分，你会看到几个链接，提供不同版本的Ruby，可在Windows平台下载。理想情况下，你会下载列表最顶部的链接文件，它称为“一键安装程序”。在编写本书时，最新版本是1.8.5。
4. 点击第3步找到的链接，并将其保存到计算机的桌面。
5. 下载完成后，查看桌面上刚刚下载的Ruby EXE文件，并双击之，载入安装程序。
6. 如果Windows提示“安全错误”对话框，请点击“运行”按钮，让其运行。
7. 此时出现典型的安装程序界面，其中有一些提示说明。在开始的界面中，点击“下一步”按钮。
8. 按界面提示进入到后续几个安装界面，保留安装文本编辑器SciTE和FreeRIDE、安装Ruby包管理器RubyGems（详见第7章）的打勾状态。除非有明确理由，否则应该让安装程序把Ruby安装到默认位置c:\ruby，并放在Windows“开始”菜单的默认程序组中。
9. 安装开始，你会看到一串文件名在屏幕上飞快闪过。安装过程要花几分钟，请等待，并开心地看这一情景，要安装那么多文件！
10. 安装完成时，安装程序显示“Installation Complete”（安装完毕），“下一步”按钮变得可点击。点击“下一步”按钮，然后再点击“完成”按钮，退出安装程序。

如果Ruby已按上述过程正确安装，恭喜你！点击“开始”菜单，然后点击“程序”或“所有程序”菜单。应该显示“Ruby”程序组，其中包含FreeRIDE、SciTE、卸载程序和其他一些图标。为了验证安装的Ruby能正常使用（为第2章的需要），你需要使用菜单列表中的“fxri-

现在可以学习第2章，开始把玩Ruby语言了。

1.1.2 Apple Mac OS X平台

与Windows不同，大多数运行着Mac OS X的现代苹果电脑，都自带安装了某个版本的Ruby，这说明你可以直接使用了。Mac OS X Panther (10.3.x) 默认包含Ruby 1.8.2，而OS X Tiger (10.4.x) 默认包含Ruby 1.8.4。

注 由于OS X Leopard是在2007年发布的，因此有可能带有最新版本的Ruby。如果你的机器运行该操作系统（在本书编写期间还未面世），可能已经一切就绪了！

本书的大部分代码都能在Ruby 1.8.2或更高版本运行，因此如果机器的操作系统是OS X Panther或Tiger，则无须再做任何特别调整。要想知道OS X是哪个版本，点击屏幕左上角的“Apple”菜单，并选择“About This Mac”菜单项，即可看到当前版本的显示。如果OS X版本超过10.3，则已经安装好Ruby了。

提示 如果你使用的是OS X Tiger (10.4.x)，请使用苹果的软件更新功能，升级到最新版本的OS X，因为苹果公司从OS X的10.4.6版本开始，把自带的Ruby版本进行了改进。如果不做这一升级，可能需要手工重新安装Ruby，才能让诸如Ruby on Rails等功能扩展正常运行。尽管这不是本书前面两部分考虑的，但会在后面给你带来一些困扰。

检查预装的Ruby版本

如果使用的是OS X Panther或OS X Tiger，可以用Terminal终端程序检查Ruby是否已安装。双击“Macintosh HD”图标（或任何硬盘名称），进入硬盘的Applications文件夹，再进入Utilities文件夹，你会找到名为Terminal的程序。双击图标，启动该程序。如果Terminal程序正常启动，将看到如图1-3所示的画面。

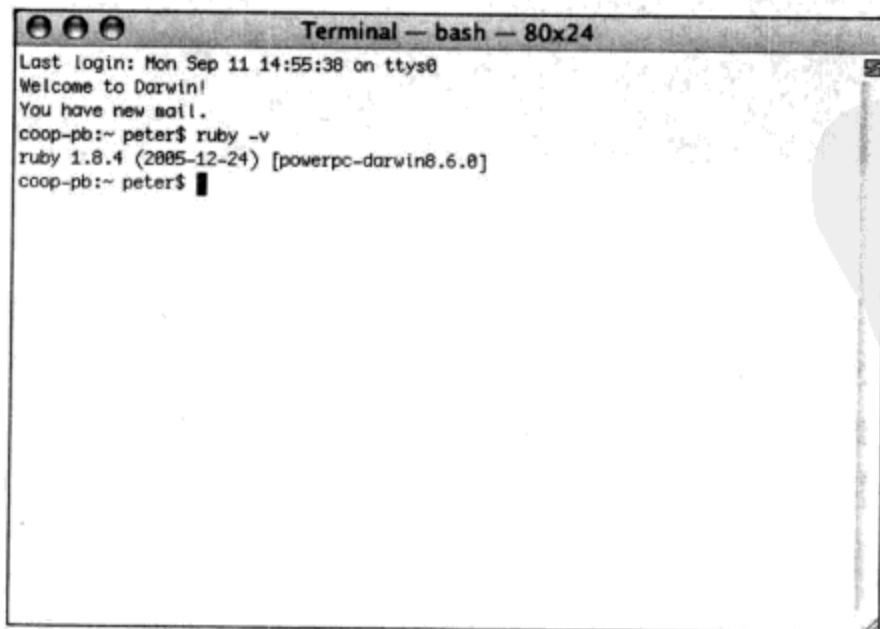


图1-3 Mac OS Tiger中的Mac OS X Terminal程序，Ruby已正常安装并验证

在Terminal程序中，你所在的位置叫做command prompt（命令提示行）或shell（系统

外壳)。从技术角度来说，当你输入命令时，就是在与计算机直接交谈。当你按下回车键时，计算机将立即执行输入的命令。

要想验证Ruby是否已经安装，请在Terminal的命令提示行输入以下命令（别忘记最后按回车键）：

```
ruby -v
```

如果命令执行成功，将看到如图1-3所示的结果，显示正在运行的Ruby的版本（理想地，这里应显示1.8.2或更高）。如果该命令奏效，可以试用名为“irb”的Ruby交互式解释器，在命令提示行输入如下命令即可：

```
irb
```

如果得到如图1-3所示的结果，就已经一切就绪，可以转到第2章了。如果需要在OS X上安装更新版本的Ruby，请继续阅读下一节。

在OS X上安装Ruby

在OS X上安装Ruby有几种方法。可以使用Fink或DarwinPorts之类的包管理器，从预先打包的安装包进行安装，或直接从Ruby源代码编译。如果已经使用Fink或DarwinPorts，那么参考相应网站的更多信息，但你会发现使用预制的安装包更容易。

最流行的安装包之一叫做Locomotive，可在网站<http://locomotive.raaum.org/>获取。

作为正规DMG OS X文件，你可以像其他OS X应用程序一样安装它（在PPC和x86架构体系均可）。与某些安装包不同，Locomotive包含了Ruby on Rails和LightTPD，这些工具不会立即派上用场，除非你准备现在就做Ruby on Rails开发，但到本书末尾，你会喜欢这些工具的。

在Mac OS X上通过源代码安装Ruby

在OS X上直接通过源代码安装Ruby的方法，与Linux上的方法相似，因此请继续阅读后续Linux小节中“通过源代码安装Ruby”。请注意这种安装方法与Locomotive之类的安装包相比，通过源代码安装Ruby，你得到的只有Ruby，以后还需要另外安装Rails之类的其他组件。

注 要在OS X上编译Ruby源代码，你需要安装OS X自带的Xcode开发工具。

1.1.3 Linux平台

作为开源编程语言，Ruby已经内置在许多Linux发行版本中。尽管还没有遍及所有Linux版本，但你可以用下一节的指导方法来检查Ruby是否安装。如果检查失败，那么这里的指导说明可以帮你安装Ruby。

检查Ruby是否已安装在Linux中

请从命令提示行（或终端窗口）运行以下命令，运行Ruby解释器：

```
ruby -v
```

如果Ruby已安装，则显示如下类似输出：

```
ruby 1.8.2 (2004-12-25) [i686-linux]
```


这表示Ruby 1.8.2已经安装在机器中。本书需要1.8.2作为最低要求，因此如果你安装的Ruby版本低于1.8.2，就需要继续阅读本章，安装更新的Ruby版本。但如果显示的是Ruby已安装且版本较新，则可以运行irb交互式Ruby解释器，命令如下：

```
irb
```

提示 在某些系统中，irb可能有稍微不同的名字。例如，它在Ubuntu中有时叫做irb 1.8，那么需要你按这个名字来运行。你可以`find / -name "irb" -maxdepth 4`命令来找到它。

运行了irb之后，你应该看到如下输出：

```
irb(main):001:0>
```

如果irb的运行结果是上述类似输出，你就可以转到第2章。（你可能想输入`exit`并按回车键来返回命令行！）否则，请继续阅读，安装新版本的Ruby。

用包管理器安装Ruby

在Linux上安装Ruby的过程，因Linux发行版本的不同而不同。某些发行版本，例如，Gentoo、Debian和Red Hat，提供了“包管理器”，让程序的安装很简单。其他发行版本则需要你直接根据源代码安装，或先安装包管理器。

如果你习惯使用`emerge`、`rpm`或`apt-get`，则可以用下列命令快速安装：

- RPM：下载Ruby RPM并用`rpm -Uvh ruby-*.rpm`命令安装。
- Gentoo：使用`emerge`，命令如下：`emerge ruby`。
- Debian：使用`apt-get`：`sudo apt-get install ruby`。
- Ubuntu：使用`apt-get`，命令同Debian。你可能还需要专门安装irb。对于Ruby 1.8，下面这行命令应该有用：

```
sudo apt-get install ruby rubyl.8 rubyl.8-dev rdoc ri irb
```

如果这些方法中有一种适应你，在安装完成后，可以用前节所述方法来试着运行Ruby和irb，如果一切正常则可以进入到第2章。另外，你可以搜索与你的操作系统版本相关的Ruby包程序库，因为在操作系统中Ruby包的名字可能是非标准的，或随时间而变化。但如果所有方法都不管用，你还可以直接通过Ruby源代码来安装，如下节所述。

通过源代码安装Ruby

如果你不在乎把手搞脏的话，那么从源代码安装Ruby是个绝妙的选择。这一安装过程对于各种形式的UNIX（不仅是Linux）都是相似的。以下是基本步骤：

1. 在机器中搜索“make”和“gcc”工具，确保你的Linux发行版本可以编译应用程序。在终端窗口，你可以用`which gcc`和`which make`命令检查这些开发工具是否已安装。如果没有安装，则需要安装这些开发工具。
2. 启动Web浏览器，访问<http://www.ruby-lang.org/>。
3. 在页面右边点击“Download Ruby”链接。如果页面设计有变化，则查找“downloading

Ruby” 链接。

4. 在下载页面点击“Ruby Source Code”稳定版本的链接。在本书编写期间，该版本是“ruby-1.8.5”。点击后将下载tar.gz文件，其中包含Ruby最新稳定版本的源代码。

5. 解压缩tar.gz文件。如果你使用命令提示行或终端窗口，则进入ruby-1.x.x.tar.gz文件所在目录，并运行tar xzvf ruby-1.x.x.tar.gz命令（这里ruby-1.x.x.tar.gz是刚刚下载的文件名）。

6. 进入解压缩所创建的Ruby文件夹。如果你此时没有使用命令提示行，请打开终端窗口，并进入该文件夹。

7. 运行./configure命令，生成Makefile和config.h文件。

8. 运行make命令，根据源代码编译出Ruby。这可能要花一些时间。

9. 运行make install命令，把Ruby安装到系统中正确的位置。你需要以超级用户（例如root）的身份来做此操作，因此可能要用sudo make install命令，并输入root用户的口令。

10. 如果在此阶段有错误发生，请阅读源代码文件附带的README文件，查找相关指示。否则，请用ruby -v命令，检查当前安装的Ruby版本。

如果此时显示的Ruby版本正如预期所示的，就可以转向第2章开始编程。如果你碰到错误，抱怨找不到Ruby，或安装了错误的Ruby版本，则Ruby安装的位置可能不在路径中（即操作系统查找运行文件的地方）。要修正这一错误，回到前面看看Ruby安装在什么位置（通常是/usr/local/bin或/usr/bin），并把相关目录加到路径中。关于路径的操作在不同的Linux发行版本或shell类型中均不相同，因此请参阅你Linux操作系统的文档中关于变更路径的说明。

一旦能够检查当前运行的Ruby版本，并且版本号是1.8.2或更高，就可以运行irb并看到Ruby解释器提示符，Ruby安装就完成了（就目前而言！），那么可以转到第2章。

1.1.4 其他平台

如果你用的操作系统平台不是Windows、Mac OS X或Linux，仍然可以使用Ruby，只要本章开头列出的清单当中有你的平台和体系架构就行。如果你是非常规平台的用户，我假定你有在系统中安装应用程序的基本知识，因此我只提供以下链接，分别指向不同的安装程序：

- MS-DOS: <http://ftp.ruby-lang.org/pub/ruby/binaries/djgpp/>。
- FreeBSD: 有多种Ruby版本，可作为标准发行版。
- OS/2: <http://hobbes.nmsu.edu/pub/os2/dev/misc/ruby-181.zip>。
- BeOS: 同前所述，可以采用与Linux相同的方式安装Ruby。
- Linspire或Lindows: 作为Linux的发行版本，同前所述，你可使用与Linux相同的方法进行安装。
- Symbian Series 60: <http://developer.symbian.com/main/tools/opensource/ruby/index.jsp>。
- Java虚拟机 (JVM): <http://jruby.codehaus.org/>。
- 其他UNIX版本: 参见前面Linux章节中“从源代码安装Ruby”的说明，它对所有版本都是一样的。

在许多情况下，与某些操作系统配套的Ruby版本可能过期或不再支持。如果碰到这种情况，你又有信心直接从源代码编译出你自己的Ruby版本，那就可以从<http://www.ruby-lang.org/en/20020102.html>下载源代码。

要验证Ruby安装是否已满足本书的要求，可以向Ruby询问其版本，以检查你安装的是Ruby的哪一个版本，命令如下：

```
ruby -v
```

你还需要能够访问Ruby的交互提示符程序irb，只须直接运行irb（假定它的目录处于你的查找路径中）即可，命令如下：

```
irb
```

如果Ruby或irb都不能正常运行，就需要寻求针对特定平台的帮助，附录C提供了有用资源的清单。

1.2 小结

本章重点关注Ruby的正确安装，以便能够运行下面几章用到的irb工具。

尽管Ruby是一门易学易用的语言，但我们 also 容易被Ruby本身的维护以及Ruby的安装升级搞得透不过气来。由于Ruby语言一直处于发展当中，因此本章讨论的内容将会过时，或在你的平台上可能有更容易的安装方法出现。

能够使用Ruby社区提供的各种资源，以及能够迅速地找到帮助，是Ruby开发人员的重要能力之一。在大多数情况下，Ruby社区能提供快捷的帮助，第5章和附录C提供了许多值得一试的可用资源。



第2章 编程等于快乐：Ruby和面向对象概览

编程既是一门科学又是一门艺术。用程序告诉计算机做什么，这需要编程人员既能以科学家，又能以艺术家的头脑思考。作为艺术家，要能提出重大思想，并足够灵活，敢于采取独特方法。作为科学家，则要能理解使用某种特定方法的原因和技巧，以及怎样从逻辑角度而非感情角度，进行测试和排错。

幸运的是，你无须现在就是艺术家或科学家。和身体锻炼一样，进行编程“锻炼”，并思考如何解决问题，这样的头脑训练可以让你成为更好的程序员。任何人都可以学编程。唯一可能造成阻碍的，是三心二意、不能专一地使用一门编程语言。Ruby是最容易学习的编程语言之一，因此无须担心不能专一。编程应该很有趣，甚至是快乐的，如果你能享受编程，那么专一就不求自来。

到本章结束时，我希望你可以品尝到开心的滋味，这滋味就隐藏在这门强大而令人信服的简单编程语言中，并开始感到兴奋，因为编程极乐世界就在前方！

注 本章不采取教学的形式（那是后续章节的事情），而是快速地从概念到概念，让你在深入后续章节的细节内容之前，对Ruby语言有个整体印象。

2.1 初始步骤

在第1章，我们的关注焦点是安装Ruby，以便让计算机可以理解这门语言。在第1章末尾，你载入名为irb的程序，对于微软Windows用户，则可以运行名为fxri的应用程序。fxri和irb都提供了相似的功能，因此当我提到irb时，如果你用的是微软Windows，请注意可以改用fxri，这是人所共知的正式做法。

2.1.1 irb：交互式Ruby

irb是“交互式Ruby”的缩写。“交互”表示当输入内容时，计算机会立即进行处理。因此，如果在irb中输入上一章的“`print 'hello, world!' 10 times`”源代码，并按回车键，则会立即看到结果。有时这种环境被称为即时或交互式环境。

注 如果你不记得怎样载入irb或fxri，请参阅第1章中关于计算机所用操作系统的相关说明。

启动irb（或fxri），确认出现了提示符，如下所示：

```
irb(main):001:0>
```

这个提示符并不像它看起来那么复杂。它的全部含义是，你目前正在irb程序中，正在输入第一

行 (001), 正处于0级代码深度。就目前来说, 你无须重视深度的概念。

在提示符之后输入以下内容, 并按回车键:

```
1 + 1
```

结果会迅速反馈出来: 2。整个过程看起来像这样:

```
irb(main):001:0> 1 + 1
```

```
=> 2
```

```
irb(main):002:0>
```

Ruby现在准备好接受下一个命令。

作为Ruby新手程序员, 你会把大量时间花在irb上, 用以验证概念, 并建立对Ruby内部机制的认识。它提供了一个完美的环境, 可以摆弄和测试这门语言, 因为你从irb内部不可能对它造成任何真正的破坏。当然, 如果你在irb中用代码显式要求Ruby删除磁盘上的文件, 这确实会发生, 但你不必担心计算机死机, 或破坏其他程序。

irb的交互式环境还提供了立即反馈的好处——这是学习的基本工具。不必在文本编辑器中输入代码, 保存文件, 让计算机运行这个文件, 然后再通过错误信息看到在哪儿搞错了, 而是直接在irb中输入小段代码, 按回车键, 并立即看到产生的结果。

如果想更进一步体验irb, 请试试其他算术表达式, 例如 $100 * 5$ 、 $57 + 99$ 或 $10 - 50$ (请注意, 在计算机中, 除号运算符是正斜杠 “/”)。

2.1.2 Ruby是计算机的通用语

通过与大多数人完全不同的方式, 计算机能够理解语言。作为无法理解微妙和模糊含义的逻辑设备, 如英语或法语之类的语言, 对计算机没有吸引力。计算机需要的语言要有逻辑结构和精心定义的语法, 以便在告诉计算机做什么时, 有清晰的逻辑。

清晰是必要的要求, 因为几乎所有在编程时告诉计算机的东西都是指令 (或命令)。指令是所有程序的基本构建单元, 而且为了让计算机能够正确运行 (或执行) 它们, 程序员的目的必须清楚而精确。成百上千的指令放在一起, 组成了程序, 它可以执行特定的任务, 这表示没有一点出错的空间。

还需要考虑, 其他程序员可能要维护你所编写的计算机程序。如果只是编着玩玩, 那这就不成问题, 但如果你的程序容易理解, 那么在以后回头来看程序时, 也能够理解它们。

2.1.3 为什么Ruby是如此杰出的编程语言

尽管用英语可能造就糟糕的编程语言 (由于其模糊性和复杂性), 但有时Ruby惊奇地给人一种类似英语的感觉。Ruby只是几百种编程语言的一种, 但它是特别的, 因为对许多程序员来说, 它的许多地方给人感觉像自然语言, 同时还有计算机需要的清晰性。来看下面的示例代码:

```
10.times do print "Hello, world!" end
```

大声读出这段代码 (很有用, 真的!), 它并不像英语读起来那么顺畅, 但其含义应该能立即清楚理解。它要求计算机在屏幕上做打印10次 “Hello, World!”。这段代码确实生效了。如果

运行irb，并输入上述代码再按回车键，可以看到结果如下：

```
Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!
Hello, world!Hello, world!Hello, world!Hello, world!
```

注 有经验的程序员可能奇怪，为什么上面代码示例的末尾没有分号。因为和诸如Perl、PHP、C或C++等许多其他语言不同，在Ruby中代码行末尾不需要分号（尽管如果你加上分号也没什么妨碍）。这在开始时可能要花一小段时间来习惯，但对于新程序员来说，这让Ruby更容易学会了。

下面是比较复杂的例子：

```
User.find_by_email('me@privacy.net').country = 'Belgium'
```

这段代码和“Hello, World!”示例没有半点相似之处，不像它那么直白易懂，但还是能把它要做什么猜出个大概。首先，它告诉计算机，想操作名为User（用户）的概念。其次，它试图找到有特定E-mail地址的用户。最后，它获取用户的国籍（country）信息，并将其修改为Belgium。就目前来说，先别担心用户的数据是怎样保存的，那是后面要讨论的事。

这是个相当高级的例子，但它通过可能有些复杂的应用程序展示了简单概念，你可以处理诸如“用户”等各种不同的概念。在本章末尾，你将看到，在Ruby中如何创建自己的实际生活概念，并以与本例相似的方法来操作它们。此时的代码也可以像英语一样易读。

2.1.4 心灵小径

学习本身可能是个有趣的活动，但只是阅读还不会让你成为专家。我阅读过一些菜谱书籍，但当我偶尔试着做菜时，我的厨艺似乎并未改善。所缺少的是试验和验证，没有它们，你的成就最多只能算是学术上的。

有了这一认识，从使用Ruby的第一天开始，就要养成随时试验和验证的心态，这是很重要的事情。贯穿本书，我将要求你尝试使用不同的代码块，把玩它们，看看会得到什么结果。偶尔会让自己感到惊奇，也许有时会追踪代码到死胡同，或常常挠头苦思（当然，如果你有头发的话！）。不管发生了什么，所有优秀程序员都从试验中学习，在学习时也只能通过试验来掌握语言和编程概念。相信我，这很好玩的！

本书将带你走过代码和概念的森林，但如果没有亲自测试和证明代码无误，你可能会很快迷失在森林中。利用irb和其他工具，我会尽可能频繁地讨论并试验代码，从而让你牢牢掌握相关知识。

在你的irb提示符后输入下列代码，并按回车键：

```
print "test"
```

结果很简单，如下所示：

```
test
=> nil
```

从逻辑上说, `print "test"` 让 `test` 打印显示在屏幕上, 但结果的第二行, 是你的代码作为表达式 (详见第3章) 的运算结果。这是因为在 Ruby 中几乎所有东西都是表达式。但 `print` 的功能是显示数据到屏幕, 而不是作为表达式返回什么值, 因此你得到的结果是 `nil`。更多内容详见第3章。

我们来试试别的:

```
print "2+2 is equal to" + 2 + 2
```

这个命令在表面上看, 似乎很合逻辑。如果 $2+2$ 等于 4, 那么就把这个结果加到 `"2+2 is equal to"` 的末尾, 然后会得到 `"2+2 is equal to 4"` 的结果, 对吗? 不幸的是, 你得到如下错误信息:

```
TypeError: can't convert Fixnum into String
    from (irb):45:in `+'
    from (irb):45
    from :0
```

当你犯错时, Ruby 就发出抱怨, 这里就是它的抱怨, 说你不能把数字转换成字符串 (“字符串”是文本的集合, 就像本句话一样)。数字和字符串不能像这样混合, 具体原因并不重要, 但这样试验一下, 会比仅仅阅读本书, 记住更多的 Ruby 内容。当类似的错误发生时, 可以利用错误信息作为解决方案的线索, 不管是在本书中发现的, 还是在因特网上, 或是通过询问其他开发人员得来的。

对于上述问题, 暂时的解决方案如下所示:

```
print "2+2 is equal to "
print 2 + 2
```

或是这样:

```
print "2+2 is equal to ", 2 + 2
```

我们再试一个例子。10除以3得到多少?

```
irb(main):002:0> 10 / 3
=> 3
```

人们以为计算机是精确的, 但是对于任何有基本算术能力的人都知道, 10除以3是3.33的循环, 而不是3!

这个古怪结果的原因是, 在默认情况下, Ruby 假定 10 或 3 之类的数字都是整数——完整的数字。在 Ruby 中用整数进行算术运算, 将给出整数结果, 因此需要向 Ruby 提供浮点数 (有小数点的数字), 才能得到浮点数结果, 例如 3.33。下面是具体的示例:

```
Irb(main):001:0> 10.0 / 3
=> 3.33333333333333
```

像这些不正常的结果, 使验证不仅是个良好的学习工具, 更是个大型程序开发的基本过程。到目前为止, 关于错误已经讲得够多了。我们来做点有用的东西!

2.2 把思路转变成Ruby代码

编程的艺术性，部分体现在它可以把你的思路转变成计算机程序。一旦熟练掌握了一门编程语言，就可以将思路直接转变成代码。但在做这件事之前，需要明白Ruby本身是怎样理解真实世界概念的，以及怎样才可以把思路转化成Ruby认可的形式。

2.2.1 Ruby怎么理解对象和类的概念

Ruby是个面向对象的编程语言。在最简单的意义上，这表示Ruby程序可以定义并操作真实世界风格的概念，可以包含诸如“人”、“盒子”、“票”、“地图”等概念，以及其他任何想处理的概念。面向对象语言让实现这些概念变得很容易，其方法是根据这些概念来创建对象。作为面向对象语言，Ruby可以按照你定义的方式，操作并理解这些概念的关系。

例如，你可能想创建某个应用程序，管理体育赛事的订票情况。涉及概念包括“赛事”、“人员”、“门票”、“举办地点”等，Ruby可以直接把这些概念放到程序中，创建这些概念的对象实例（“赛事”的实例可能是“超级杯”或“2010世界杯”），并对其执行操作，以及定义相互之间的关系。程序中有了所有这些概念，便可快速把“赛事”和“举办地点”、“门票”和“人员”关联起来，这样一来，代码从一开始就可以表达逻辑体系。

如果你以前没有做过编程，那么“把真实世界的概念直接用在计算机程序中”这种想法，对你来说似乎是明摆的，是一种让软件开发更简单的方法。但其实面向对象是一种相当新颖的软件开发思想（其概念诞生于1960年代，但只到1990年代才开始在主流编程中流行起来）。如果采用非面向对象语言，由于程序员在处理概念以及概念间关系方面拥有的灵活性不多，所以将会在此耗费大量开销。

2.2.2 造人过程

我们直接跳进源代码，演示一个简单概念，person（人）：

```
class Person
  attr_accessor :name, :age, :gender
end
```

前文的Ruby代码看起来很像英语，但在定义概念时，它就没那么像了。我们来一步一步地解析这段代码：

```
class Person
```

这行代码是定义“Person”这个概念的起始点。当用Ruby（或大多数其他面向对象语言）定义概念时，我们把概念称为类（class）。类是单个对象类型的定义。在Ruby中，类名永远以大写字母开头，因此你的程序最终由User、Person、Place、Topic、Message等名字的种类组成。

```
attr_accessor :name, :age, :gender
```

上一行代码为Person类提供了三个属性（attributes）。每个person都有name（名字）、age

(年龄)和gender(性别),因此该行代码创建了这些属性。attr代表“attribute”(属性),而accessor大概表示“让这些属性可访问,可被设置和修改”。这表示当在代码中操作某个Person对象时,可以修改这个person(人)的name(名字)、age(年龄)和gender(性别),或更准确地说,是name、age和gender属性。

```
end
```

end行的作用是明摆着的,它与第一行的类定义配对,告诉Ruby下面不再是Person类的定义。

回顾一下,类定义了概念(例如Person),而对象是基于类的单个东西(例如“Chris”或“Smith夫人”)。

那么,我们来体验一下Person类吧。进入irb提示符,并输入前文建立的Person类。输入应该如下所示:

```
irb(main):001:0> class Person
irb(main):002:1> attr_accessor :name, :age, :gender
irb(main):003:1> end
=> nil
irb(main):004:0>
```

我们注意到,在输入类代码时,每个irb提示行末尾的数字会变化。原因是当在class Person代码行按回车键时,Ruby知道现在是在类结构中,正在定义类,而不是在输入要立即执行的代码。其中的1表示当前处于概念嵌套的1级深度。如果你对此尚无概念,别担心,我会在后面详细谈到。

类定义输入完成之后,Ruby对输入内容进行处理,返回nil,这是因为类定义的结果是无返回值,而nil是Ruby表达“无”的方法。由于没有发生错误,你的Person类现已在Ruby中存在,因此我们用它来做点什么:

```
person_instance = Person.new
=> #<Person:0x358ea8>
```

第一行代码的功能是创建Person类的“新”实例,因此我们创建了一个“新人”,并将其赋给person_instance——这是表示新人的占位符,我们知道它其实是个变量(variable)。第二行代码是Ruby对创建新人的响应,其含义目前并不重要。在不同的计算机上,0x358ea8比特位也各不相同,它只表示Ruby赋给该新人的内部引用地址,你根本不必关心。

我们马上用person_instance来做点什么:

```
person_instance.name = "Robert"
```

在这个基础示例中,引用person_instance的name属性,并将其赋值为“Robert”。也就是说,我们刚刚给这个人起了一个名字。Person类还有另外两个属性:age和gender,我们来对其赋值:

```
person_instance.age = 52
person_instance.gender = "male"
```

很简单。这样一来,我们给了person_instance一个基本身份。把这个人的名字打印到屏幕上怎么样?

```
puts person_instance.name
```

当按回车键时，Robert显示在屏幕上。请用相同的方法，试试age（年龄）和gender（性别）。

注 在前文的示例中，我们用print命令把东西打印到屏幕。在上例中，用的是puts命令。print和puts的区别在于，puts自动把输出光标放到下一行，而print在上次的光标位置继续打印文本。一般情况下，你会使用puts，但我有更早的例子中使用print，是为了看起来更直观，读起来更上口。

2.2.3 基础变量

在上一节中，我们创建了一个person对象，并把它赋给一个名为person_instance的变量（变量是“占位符”的计算机术语）。

变量是编程的重要组成部分，它们容易理解，特别是在只有最少的算术知识情况下。请看下面这个例子：

```
x = 10
```

这行代码把10这个值赋给变量x。x现在等于10，你可以做如下操作：

```
x * 2
```

```
20
```

在Ruby中，变量可以包含Ruby可以理解的任何概念，例如数字、文本和其他数据结构。这一点将贯穿本书。在上一节中，person_instance是个变量，它指向Person类的对象实例，与“x是个包含数字10的变量”非常相似。更简单地说，可以把person_instance看成一个名字，它指向某个特定的、唯一的Person对象。

当想在程序中保存某个东西，并在不止一行代码中使用它，就会用到变量，让它作为所操作数据的临时存储空间。

2.2.4 从人到宠物

上文创建了一个简单的类（Person），创建了该类的对象，将其赋给person_instance变量，并给它一个用于查询的身份（称它为“Robert”）。如果这些概念对你来说太简单，那么你做得很好，这说明你已经理解了面向对象的非常基础的概念！如果还未理解，请重新阅读上一节，并务必在计算机上照做一遍，另外还要阅读本节，因为我准备稍微加深一点深度。

从Person类开始起步，现在需要掌握更复杂点的知识，因此我们来创建一些在Ruby中生活的“宠物”。我们将创建猫（cat）、狗（dog）和蛇（snake）。第一步是定义类，可以这么做：

```
class Cat
  attr_accessor :name, :age, :gender, :color
end
```

```
class Dog
  attr_accessor :name, :age, :gender, :color
```

```
end

class Snake
  attr_accessor :name, :age, :gender, :color
end
```

这和创建Person类很相似，只是为不同的动物重复了三次。可以继续下去，用`lassie = Dog.new`或`sammy = Snake.new`之类的代码创建动物，并用`lassie.age = 12`或`sammy.color = "Green"`之类的代码设置宠物的属性。如果愿意的话，可以在计算机上输入这些代码试试。

但是用这种方法创建类，将丢失面向对象编程的最好的一个特点：继承。

继承让不同的类一个个关联起来，并根据其相似之处形成团组的概念。在本例中，猫、狗和蛇都是宠物，用继承可以创建一个“父”类：Pet（宠物）类，并让Cat、Dog和Snake类继承所有宠物都具有的特征。

在真实世界中，几乎所有东西都与上面的类有相似的结构。猫是宠物，而宠物又是动物，动物又是生物，而生物又是存在于世界上的东西。类的层次谱系无处不在，面向对象的语言则让你可以在代码中定义这些关系。

注 第6章提供了一张有用的图形，显示了诸如哺乳动物、植物等不同形式生命之间的继承概念。

逻辑层次分明地构建你的宠物

我们已经想出了一些改进代码的思路，下面重新从零开始输入。为了完全清除和重新设置刚才输入的内容，可以重新启动irb。irb不会记得前一次使用中的信息。因此重新启动irb（要退出irb，请输入exit并按回车键），并重写类定义，如下所示：

```
class Pet
  attr_accessor :name, :age, :gender, :color
end

class Cat <Pet
end

class Dog <Pet
end

class Snake <Pet
end
```

注 本章所列代码，在类中的代码都被缩进，如上文Pet类中attr_accessor行所示。这只是一种编码风格，让代码更容易阅读。当你在irb中输入代码时，无须照搬缩进，可以直接输入。而当你是在文本编辑器中编写更长的程序时，就应该缩进代码以便更容易阅读，但现在来说并不重要。

在上面的代码中，首先创建Pet（宠物）类，并定义Pet对象可以使用的name（名字）、age（年龄）、gender（性别）和color（颜色）属性。然后根据Pet类直接继承，定义Cat（猫）、Dog（狗）和Snake（蛇）类。这表示猫、狗和蛇的对象都会有name、age、gender和color属性，但由于这些属性是从Pet类继承而来，因此不必在每个类中分别创建。这让代码更容易维护和更新（如果你想保存更多宠物信息，或想增加其他动物类型的话）。

那么不是每个动物都有的属性该怎么办？如果你想保存蛇的长度，但不想保存狗和猫的长度，该怎么办？幸运的是，继承提供了很多优点，但没有缺点。可以为特定类增加你想要的代码。我们重新输入Snake类，如下所示：

```
class Snake < Pet
  attr_accessor :length
end
```

现在Snake类有了length（长度）属性。但这只是加给继承自Pet的Snake类的，因此Snake类有name、age、gender、color和length这五个属性，而Cat和Dog类只有前面四个属性。你可以验证一下，如下所示（为简洁起见，去掉了一些输出行）：

```
irb(main):001:0> snake = Snake.new
irb(main):002:0> snake.name = "Sammy"
irb(main):003:0> snake.length = 500
irb(main):004:0> lassie = Dog.new
irb(main):005:0> lassie.name = "Lassie"
irb(main):006:0> lassie.age = 20
irb(main):007:0> lassie.length = 10
```

```
NoMethodError: undefined method 'length=' for #<Dog:0x32fddc @age=20,
@name="Lassie">
```

这里创建了一只狗和一条蛇，并规定蛇的长度为500，然后试图规定狗的长度为10（这里的长度单位并不重要）。给狗规定长度导致了“undefined method 'length='”错误，因为我们只对Snake类定义了length属性。

你可以试试其他属性，并创建其他“宠物”。试试不存在的属性，看看出错信息是什么。

控制你的宠物

到目前为止，我们已经创建了类和含有各种可修改属性的对象。属性是与各个对象相关的数据，例如蛇有长度，狗有名字，猫有颜色。那么我前面说过的指令是怎么回事？怎样向对象发出指令让它执行？答案是通过为每个类定义方法（method）。

方法在Ruby中很重要，它能让对象做事。例如，可能想向Dog类增加bark（吠叫）方法，当对Dog对象调用此方法时，向屏幕输出“Woof!”（“汪！”）。这段代码可以这么写：

```
class Dog < Pet
  def bark
    puts "Woof!"
  end
end
```


输入这段代码后，创建的任何狗对象都可以吠叫了。我们来试一下：

```
irb(main):0> a_dog = Dog.new
irb(main):0> a_dog.bark
```

```
Woof!
```

尤里卡！（译者注：希腊语，意为“我找到了！”，据传因阿基米德在浴缸发现浮力后欢呼着一跃而起而得名。）你会注意到，让狗吠叫只是引用这只狗（本例为`a_dog`）并加一个点号（“.”），后跟吠叫方法的名字，然后这只狗就“吠叫”起来。我们来解析一下，到底发生了什么事情。

首先，把`bark`方法加到`Dog`类，做法是定义这个`bark`方法。要定义方法，则使用`def`一词，并在其后跟要定义方法的名字。这就是`def bark`代码行的意思，表示“我现在开始在类中定义`bark`方法，到我说`end`为止”，其后代码行的功能只是在屏幕上输出“Woof!”（“汪！”）。方法的最后一行是结束该方法定义，最后的`end`结束类定义（这就是缩进的用途所在，让你可以看出哪个`end`行匹配哪个定义）。这时，`Dog`类就包含了名为`bark`的新方法，如前文所用。

可以想一下，该怎样为其他`Pet`子类或`Pet`类本身创建方法。有没有对所有宠物都通用的方法？如果有，这样的方法应该放在`Pet`类中。有没有猫类的专有方法？这样的方法应该放在`Cat`类中。

2.3 一切都是对象

在本章我们见识了Ruby怎样以类和对象的形式理解概念，创建了虚拟的猫和狗，对其命名，并调用其方法（例如`bark`方法）。这些基本概念构成了面向对象的核心，在本书中将经常运用这些概念。狗和猫只是面向对象所提供灵活性的示例，而我们到目前为止所用的概念，可以在大多数情况中运用，不管是给“票”发出变更其价格的命令，还是给“用户”发出变更其密码的命令。在构思要开发的程序时，请开始用其普通概念的术语来思考，并考虑怎样将这些概念，转化成可用Ruby操作的类。

即使在各种面向对象编程语言中，Ruby也是相当独特的，因为在Ruby语言中，几乎所有东西都是对象，甚至关于语言本身的概念也是对象。我们来看下面的代码行：

```
puts 1 + 10
```

如果在`irb`中输入这些代码并按回车键，会看到显示结果为11。对Ruby的要求是在屏幕上打印输出`1+10`的结果，这看似非常简单，但信不信由你，这个简单的代码行用到了两个对象。1是个对象，10也是。它们都是`Fixnum`类的对象，而这个内置类有定义好的方法，来执行数字类操作，例如加法和减法。

我们已经讨论过概念怎样关联到不同的类，前文的宠物类即是很好的例子。即便是定义类和对象的概念以便程序员用来编写计算机程序，也是如此。当编写例如`2+2`之类的简单汇总时，期望计算机把两个数字加起来，得到结果4。而Ruby用其面向对象的方式，把汇总操作的两个数字（2和2）视为数字对象，`2+2`只是一种快捷方式，要求第一个数字对象把第二个数字对象与自身相加。事实上，“+”号是个相加的方法！

可以证明，在Ruby中一切都是对象，因此只须询问它是哪个类的成员即可。在前文的宠物

示例中，可以用下面的代码，让a_dog告诉你它是哪个类的成员：

```
puts a_dog.class
```

```
Dog
```

class方法与bark方法不同，它并非自己创建的，而是Ruby默认提供给所有对象的方法。这表示我们可以用class方法询问任何对象，它是哪个类的成员。因此，当你输入puts a_dog.class代码时，得到的结果是Dog。

如果询问数字的所属类，会是什么结果？我们来试一下：

```
puts 2.class
```

```
Fixnum
```

数字2是Fixnum类的对象，这表示Ruby要做的，只是在Fixnum类中实现数字汇总相加的逻辑和代码，与在Dog类中创建bark方法很相似，然后Ruby就会知道怎样把两个数字加起来！更好的是，可以向Fixnum类增加自己的方法，让它用我们认可的方法来处理数字。

2.3.1 Kernel模块的方法

Kernel是个特殊的类（实际上，是个模块（module），但不必操心，到第6章再说！），它的方法在Ruby的每个类和有效范围中都可以使用。上面已经用过Kernel提供的一个关键方法。

我们来看puts方法，我们已经用过这个方法把数据打印输出到屏幕，如下所示：

```
puts "Hello, world!"
```

但与自己类的方法不同，puts方法没有完整的类或对象的前缀。由于puts是把文本放到屏幕上，因此完整的命令似乎应该是Screen.puts或Display.puts之类的写法。但实际上，puts是来自Kernel模块的方法，这个模块是特殊类型的类，其中装满了标准的、常用的方法，以便让代码更容易读写。

注 Ruby的Kernel模块与操作系统的kernel模块或Linux的kernel模块没有任何关系。与操作系统中kernel模块的地位相似，Ruby的Kernel模块也是Ruby的“核心”部分，但除此之外二者没有任何关系。

当输入puts "Hello, world!"时，Ruby发现没有指定任何类或对象，就在默认的、预定义的类和模块中查找名为puts的方法，在Kernel模块中进行查找，并进行调用。当看到没有明显指定类或对象的代码行时，请思考一下，这个调用方法转到哪里去了。

为了保证调用的一定是Kernel puts方法，可以显式地引用Kernel，尽管对于puts来说是极少使用的：

```
Kernel.puts "Hello, world!"
```

2.3.2 向方法传递数据

向狗发出吠叫的请求，或询问对象的所属类，用Ruby来做非常简单。只须引用类或对象，

后跟点号和方法名即可，例如`a_dog.bark`、`2.class`或`Dog.new`。但还有一些情况下，是不想发出简单命令，而是想关联一些数据给它。`puts`方法就是一个例子，我们来显式地调用它：

```
Kernel.puts "Hello, world!"
```

```
Hello, world!
```

对于`puts`方法，我们需要把要打印输出到屏幕的数据传递给它，这就是“`Hello, world!`”放在方法名之后的原因。

尽管可以在方法调用后面直接跟上相关数据，但这只是一种快捷方式，当把多个方法连接在一起时（在本章后面你将要这么做）就显得有些累赘。为了让方法和数据之间的关系清楚明白，常用的做法是把数据包含在括号中，放在方法调用之后，如下所示：

```
Kernel.puts("Hello, world!")
```

这行代码的含义与`puts "Hello, world!"`完全相同，执行方式也完全一样，唯一区别在于：

1. `puts`是`Kernel`模块的方法，而`Kernel`模块是被默认包含和搜索的，因此通常无须使用`Kernel.puts`的方式来引用。

2. `puts`方法只接受一个自由变量（argument）（即传递给方法的具体数据条目，也经常叫做参数（parameter）），后面极少跟其他方法或代码逻辑，因此括号不是严格必须的。但括号经常是需要的，因为在很多情况下，忽略括号让代码含糊难解，不够精确。

因此，以下所有代码行都是等价的：

```
Kernel.puts("Hello, world!")
Kernel.puts "Hello, world!"
puts("Hello, world!")
puts "Hello, world!"
```

以上各种情况，都是将“`Hello, world!`”数据传递给`Kernel.puts`的方法，只是调用风格各不相同。正如完成本章的某些示例一样，请用括号和/或直接用`Kernel`模块，都是使试验向屏幕输出数据的不同方法。

2.3.3 使用String类的方法

现在已经把玩了狗对象和数字对象，而文本行（字符串（strings））也很有趣，值得一试：

```
puts "This is a test".length
```

```
14
```

字符串“`This is a test`”是`String`类的对象（请用“`This is a test.class`”确认这一事实），请求它用`length`方法获得字符串的长度，并打印输出到屏幕上。所有字符串均可以调用`length`方法，因此可以把“`This is a test`”替换成你希望的任何文本，从而都可以得到合法的答案。

我们能做的，不只是询问字符串的长度。下面来看这行代码：

```
puts "This is a test".upcase
```

```
THIS IS A TEST
```

String类有许多方法，将在下一章详细介绍，但现在可以试验一下这些方法：capitalize、downcase、chop、hash、next、reverse、sum或swapcase。表2-1展示了字符串可用的一些方法。

表2-1 对"Test"字符串调用不同方法的结果

表达式	输出	表达式	输出
"Test" + "Test"	TestTest	"Test".reverse	tseT
"Test".capitalize	Test	"Test".sum	416
"Test".downcase	test	"Test".swapcase	tEST
"Test".chop	Tes	"Test".upcase	TEST
"Test".hash	-98625764	"Test".upcase.reverse	TSET
"Test".next	Tesu	"Test".upcase.reverse.next	TSEU

在表2-1中，有些例子很明显，例如改变文本的大小写，或逆转文本内容，但最后两个例子特别有意思。它们不是对文本进行单一方法处理，而是连接处理两到三个方法。之所以能这么做，是因为这些方法处理了原来的对象之后，将其作为结果返回，因此我们得到新鲜的String对象，并可以对其调用其他方法。"Test".upcase返回TEST，再对其调用reverse方法，结果返回TSET，再对其调用next方法，即“递增”最后一个字符，得到结果仍为TSEU。

在下一章，我们将更深入地考察字符串，此处介绍的链接式方法调用，可以得到快捷结果，它是Ruby的一种重要概念。你可以大声朗读上文示例，即可理解其含义。能提供如此瞬时熟悉度的编程语言并不多！

2.4 以非面向对象方式使用Ruby

到目前为止，本章考察了几个相当复杂的概念。对于某些编程语言而言，面向对象几乎是后配的，其入门书籍在读者理解语言基础之前，不会谈到面向对象的内容（特别是Perl和PHP这两种流行的Web开发语言）。但Ruby不是这样，因为Ruby是一种纯粹的面向对象语言，一旦理解了这些概念，你就能立刻从中得到重大收益。

当然Ruby也是从其他语言衍生而来的，它受到Perl和C等语言的重要影响，这两种语言都被视为过程式、非面向对象语言（尽管Perl有一些面向对象的功能特性），因此，尽管在Ruby中几乎所有东西都是对象，如果你喜欢，还是可以用非面向对象语言的相同方式，即使这种使用方式并不理想。

对于Perl或C之类的语言来说，常见的演示程序是创建子程序（subroutine）（从本质来说，它是未关联对象或类的某种方法）并调用之，与向Dog对象调用bark方法非常相似。下面是以Ruby编写的类似程序：

```
def dog_barking
  puts "Woof!"
```



```
end
```

```
dog_barking
```

这与以前的经验大不相同，即并非在类中定义方法，而是直接定义方法本身。这个方法非常通用，看起来不与任何特定类或对象关联，而是独立的。在Perl或C之类的语言中，这种方法称为过程（procedure）、函数（function）或子函数（subfunction），正如方法一词通常用于指向在对象身上发生的某个行为。

方法定义之后，我们仍称它为方法，尽管其他语言会将其视为子程序或函数——即可立即调用，无须使用任何类或对象名，就像puts方法可直接使用，无须引用Kernel模块一样。在调用此方法时，只须使用其方法名本身，如前例最后一行所示。在irb中输入前例代码，结果是dog_barking方法被调用，返回如下结果：

```
Woof!
```

然而，与Kernel.puts一样，dog_barking方法确实隶属于某个类。在Ruby中，几乎所有东西都是对象，包括魔术式的无类式方法！目前，对其内部机制的精确理解并不重要，但谨记Ruby的面向对象方法总是有用的，即使在试图不使用面向对象技术的情况下！

注 如果试验一下，你会发现dog_barking就是Object.dog_barking。

2.5 小结

在本章中，学到了几个重要概念，不仅用于Ruby编程，而且用于一般意义上的编程。如果你已经理解这些概念，那么你就已经踏上坚实的Ruby开发人员之路。在继续前进之前，我们来复习一下这些主要概念：

- 类：在诸如Ruby的面向对象语言中，类是对概念的定义。例如，我们可以创建名为Pet、Dog、Cat、Snake和Person的类。类可以从其他类中继承相应的功能特性，但仍有其自身独特的功能特性。
- 对象：对象是类的单个实例（或在某些情况下，是类自身的实例）。Person（人）类的实例是单个的人，Dog（狗）类的实例是单个的狗。可以把对象看成实际生活的对象，类是其分类，而对象是实际“东西”本身。
- 面向对象：面向对象是一种方法，它在诸如Ruby之类的编程语言中，运用类和对象对真实世界的概念进行建模。
- 变量：在Ruby中，变量是单个对象的占位符，这个对象可以是数字、字符串、列表，或是我们定义的类的实例，例如本章的Pet。
- 方法：方法代表类和/或对象中的一组代码（包含多个命令和语句）。例如，我们的Dog类对象有bark方法，该方法把“Woof!”打印输出到屏幕。方法也可以直接链接到类，例如fred = Person.new，这里的新是个方法，它根据Person类创建新对象。方法也可以接收数据——称为参数——包含在括号中，放在方法名之后，例如puts("Test")。
- 参数：包含在括号中传递给方法的数据（或在某些情况下，跟在方法名之后，不加括号，

例如`puts "Test"`)。

- **Kernel模块**: 某些方法无须类名即可使用, 例如`puts`。这些方法通常都是内置的常用方法, 与任何类都没有明显的关系。许多此类方法都被包含在Ruby的Kernel模块中, 该模块提供的功能可从任何位置的Ruby代码调用, 无须显式地引用模块名。
- **试验**: 编程最令人满足的事情之一, 就是可以把梦想变成现实。你需要哪些技能要因你的梦想而异, 但一般来说, 如果想开发某种应用程序或服务, 可以大胆尝试。大多数软件都源自需要或梦想, 因此请时刻关注可能想要开发的东西, 这种关注非常重要。更重要的在于, 当初次获得一门新语言的实用知识时, 例如正在阅读本书时, 如果有某个灵感, 请把它划分为可以用Ruby类来表示的最小组件, 然后观察是否可以用你目前学到的Ruby知识, 把这些构造元素组合起来。你的编程技能只能通过实践来提高。

在下面几章中, 我们将继续学习, 以便更深入地考察本章简要提到的几个主题。



第3章 Ruby的构造元素：数据、表达式和流程控制

计算机程序的所有运行时间几乎都花在操作数据上。我们输入字、词、数字，听音乐，看视频，而计算机则进行计算，作出决策，并将信息转发给我们。要编写计算机程序，必须理解关于数据的基础知识，以及怎样操作数据。Ruby让这些非常简单。

本章考察Ruby支持的数据的某些基本形式，以及怎样处理和操作这些数据。本章讨论的主题是提供大部分基础知识，我们以后开发的Ruby程序都是建立在这一基础之上。

3.1 数字与表达式

计算机在最底层是完全基于数字的，所有东西都用数字流表示。Ruby之类的语言将与你与计算机内部运算隔离开来，而Ruby中使用的数字与实际生活中使用的数字（例如统计、逻辑比较、算术等）几乎完全相同。我们来看一下，在Ruby中怎样以这些方法来使用数字，以及怎样用这些数字来进行处理。

3.1.1 表达式基础知识

在编程时，表达式（expression）是指数字、运算符（例如+或-）和变量的组合，当计算机理解此表达式时，将以某种回答形式得出结果。例如，以下都是表达式：

```
5
1 + 2
"a" + "b" + "c"
100 - 5 * (2 - 1)
x + y
```

开头4个表达式都可以在irb中直接输入并得到基本运算结果（1+2的结果是3，"a"+"b" + "c"的结果是abc等）。括号的用法与常规算术相同，括号中的内容先计算（或以技术性更强的说法，得到更高的优先级（precedence））。

注 你可以用Ruby即时解释器irb试验本章所有主题。如果卡住了，只需随时输入exit退出irb，再如第1章所示重新启动irb即可。

表达式普遍用于所有计算机程序，不仅仅是数字。然而，一旦理解了表达式和运算符对数字的处理方式，就可以立即明白它们怎样处理文本、列表和其他东西。

3.1.2 变量

在第2章中，我们走马观花地介绍了许多概念，包括变量的概念。变量是对象的占位符或引用，这里的对象包括数字、文本或所创建的任何对象。例如：

```
x = 10
puts x
```

10

这里把数值10赋值 (assign) 给名为x的变量。可以对变量任意命名，只有很少的限制。变量名必须是单个实体 (不含空格!)，必须以字母或下划线开头，必须只包含字母、数字或下划线，而且是区分大小写的。表3-1展示了合法与不合法的变量名：

变量是很重要的概念，因为有了变量，你就可以编写和使用操作各种数据的程序。例如，我们来看下面这个小程序，它的唯一任务是对两个数字做减法：

```
x = 100
y = 10
puts x - y
```

90

如果以上代码只是puts 100-10，会得到相同的结果，但不像上面代码这么灵活。使用变量则可以从用户、文件或其他来源，得到x和y的值。剩下的唯一代码逻辑是做减法。

由于变量是值和数据的占位符，因此还可以把表达式 (例如x=2-1) 的值赋给变量，还可用于表达式本身 (例如x-y+2)。下面是一个更复杂的例子：

```
x = 50
y = x * 100
x += y
puts x
```

5050

我们逐行来看这个例子。首先把x设为等于50，然后把y设置为x*100的值 (即50 * 10或5000)。下一步是把y加到x上，再把结果5050打印输出到屏幕。这些都很容易理解，但第三行代码的含义乍看起来不是那么明显。把y加到x，似乎用x=x+y看起来更符合逻辑，而不是x+=y。这是Ruby的另一个快捷方式，由于对变量本身执行某个操作在编程中过于频繁，因此将其进行缩减，把x=x+y缩为x+=y。同样可以对其他操作进行这种缩减，例如乘法和除法，即x*=y和x/=y也是合法的写法。变量值加1的常用方法是x+=1，它是x=x+1的快捷方式。

3.1.3 比较运算符与表达式

没有逻辑判断的程序是计算器，计算机不只对数据进行操作，还用逻辑判断来确定采取何种正确行为。逻辑判断的基本形式是在表达式中运用比较运算符，以作出决定。

表3-1 合法与不合法的变量名

变量名	合法或不合法
x	合法
y2	合法
_x	合法
7x	不合法 (以数字开头)
this_is_a_test	合法
this is a test	不合法 (不是单个单词)
this'is@a'test!	不合法 (包含非法字符: '、@和!)
this-is-a-test	不合法 (看起来像减法)

我们来看这样一个系统，它要求用户的年龄达到一定岁数：

```
age = 10
puts "You're too young to use this system" if age < 18
```

如果运行这段代码，会看到“You're too young to use this system”（你太年轻了，不能使用本系统）的提示，因为当age的值小于18时，则将这段文字打印输出在屏幕上。我们来做点更复杂的：

```
age = 24
puts "You're a teenager" if age > 12 && age < 20
```

这段代码没有任何反馈信息，因为某人的年龄表明他不是青少年。但如果age的值在13和19之间，则这段提示消息将会显示出来。这是一种两个小型表达式用&&连接在一起的情况，&&表示“与”。大声朗读这类表达式，是理解其含义的最好方式。这行代码的含义是，如果age大于12并且小于20，那么打印输出文本内容。

如果要得到相反的结果，可以用unless关键字：

```
age = 24
puts "You're NOT a teenager" unless age > 12 && age < 20
```

对于年龄24，这次你会看到提示消息，说你不是青少年。这是因为unless与if的含义正好相反。这行代码的含义是，显示这段提示消息，除非年龄处在青少年年龄段之中。

注 Ruby提供了另一种巧妙的手段，是用between?方法，如果对象处于（或等于）指定的两个值之间，则该方法返回true。例如，age.between?(12, 20)。

还可以测试是否相等：

```
age = 24
puts "You're 24!" if age == 24
```

请注意，“等于”的概念有两种不同含义，也有两种相应的表达方式。第一行代码明确了age等于24，表示要让age包含数字24，而第二行代码则是询问age是否“等同于”24。前者是要求，后者是询问。这一区别导致二者采用不同的运算符，以防止混淆。因此，是否等于的运算符是==，而赋值运算符只是=。用于数值比较的全套运算符列表如表3-2所示。

表3-2 Ruby数值比较运算符的全套列表

比较运算	含 义
<code>x > y</code>	大于
<code>x < y</code>	小于
<code>x == y</code>	等于
<code>x >= y</code>	大于或等于
<code>x <= y</code>	小于或等于
<code>x <=> y</code>	比较。如果x等于y则返回0，如果x更大则返回1，如果y更大则返回-1
<code>x != y</code>	不等于

如前文所见，可以把多个表达式放在一个表达式中，例如下面的代码：

```
puts "You're NOT a teenager" unless age > 12 && age < 20
```

&&用来强制age > 12和age < 20二者都是true，但也可以用||来检查二者之一为true，如下所示：

```
puts "You're either very young or very old" if age > 80 || age < 10
```

也可以灵活运用括号，把多个比较运算串在一起：

```
puts "You're a working age man" if gender == "male" && (age >= 18 && age <= 65)
```

本例的含义是检查gender（性别）是否等于"male"（男），并检查age是否在18和65之间。

3.1.4 用块和迭代子在数字中循环

几乎所有程序都需要某种反复循环的操作方式，以达成结果。如果像下面代码这样循环计数，是极其低效率（和不灵活！）的：

```
x = 1
puts x
x += 1
puts x
x += 1
puts x
...
...
```

在这种情况下，我们想做的是实现循环，这是一种让程序反复调用同一段代码的机制。下面是实现循环的基本方法：

```
5.times do puts "Test" end
```

```
Test
Test
Test
Test
Test
```

首先，选取数字5。然后，调用times方法，这是Ruby中所有数字都通用的方法。我们不向该方法传递数据，而是传递更多的代码，即位于do和end之间的代码。times方法随即连续五次调用这些代码，生成上述五行输出内容。

另一种编写方式是用花括号，而不是do和end。尽管对于多行代码块，推荐使用do和end，但对单行代码来说，花括号使代码更易读。因此，下面代码的功能是完全相同的：

```
5.times { puts "Test" }
```

从现在开始，我们将使用这种风格的单行代码，但对于更长代码块也会使用do和end。这是一种值得养成的良好习惯，因为几乎所有专业Ruby开发人员都遵循这种习惯（尽管总有例外）。

在Ruby中有一种创建循环的机制，叫做迭代子（iterator）。所谓迭代子，是指在列表条目中

逐步递进的东西。在本例中，迭代子在五个步骤中循环或称迭代，从而生成五行“Test”。对于数字来说，也可用其他迭代子，如下所示：

```
1.upto(5){...code to loop here...}
10.downto(5){...code to loop here...}
0.step(50,5){...code to loop here...}
```

第一个例子是从1“上溯到”5进行计数，第二个例子是从10“下溯到”5进行计数，最后一个例子调用数字0的step方法，表示从0步进到50，每步递进5。

目前还不清楚的是，在迭代过程中怎样保留每步被迭代的数字，以使用循环代码对它做些什么。如果想打印输出当前迭代数字，该怎么办呢？怎样编写用这些迭代子进行计数的程序呢？很简单，将迭代状态传递给循环代码作为参数即可，如下所示：

```
1.upto(5) { |number| puts number }
```

```
1
2
3
4
5
```

最简单的理解方法，是把do和end之间的代码看成是被循环的代码。在代码开头，“从1上溯到5”的计数数字顺着滑道，被发送给名为number的变量。可以把“滑道”想像成环绕着number两边的扶手，参数就是这样传递给无名代码块的（与类和对象中的方法不同，那些方法是有名字的）。在上面的代码中，要求Ruby从1到5计数，它随即从1开始，1被传递到代码块中，并用puts显示出来。再从2到5重复这一过程，最后产生所看到的输出结果。

请注意，Ruby（和irb）不关心（当然，总有例外！）我们是否把代码分布在多行。例如，下面的代码和上例的功能完全相同：

```
1.upto(5) do |number|
  puts number
end
```

3.1.5 浮点数

在第2章中，我们运行了10除以3的测试，如下所示：

```
puts 10 / 3
```

```
3
```

结果是3，尽管实际答案应该是3.33...的循环。原因在于，默认情况下，Ruby把没有浮点（也称为小数点）的任何数字都当成是整数。在说到10/3时，是让Ruby把两个整数（integer）相除，然后让Ruby返回整数结果。我们来稍稍改进一下这行代码：

```
puts 10.0 / 3.0
```

```
3.3333333333333333
```

现在得到了我们期望的结果，Ruby用Float类的数字对象来进行处理，并返回Float对象，从而提供了期望的精度级别。

有某些情况下，我们无法控制输入的数字是整数还是浮点数，但希望把它们视为浮点数。我们来看这样一个情况，用户输入两个待除的数字，并要求得到精确的答案：

```
x = 10
y = 3
puts x / y
```

```
3
```

两个输入数字都是整数，因此结果和前面一样，也是个整数。幸运的是，整数有个特别的方法，可以把它们实时转换成浮点数。只须把代码改写如下：

```
x = 10
y = 3
puts x.to_f / y.to_f
```

```
3.33333333333333
```

在此情况下，当进行除法运算时，x和y都已用Integer类的to_f方法，转换成浮点的等价形式。同样，浮点数也可以用to_i方法，反向转换成整数：

```
puts 5.7.to_i
```

```
5
```

我们将在3.6.5节用其他方法来考察这一技术方法。

3.1.6 常量

前文考察了用变量来分隔数据与逻辑的方法，并得出结论，需要把数据作为计算机程序的直接组成部分的情况极其罕见。这在大多数情况下是正确的，但考虑一下某些永不改变的值——例如，圆周率的值。这些不改变的值称为常量（constant），在Ruby中的表示形式是以大写字母开头的变量名：

```
Pi = 3.141592
```

如果在irb中输入上述代码，并尝试修改Pi的值，irb会允许修改，但给出警告提示：

```
Pi = 3.141592
Pi = 500
```

```
(irb): warning: already initialized constant Pi
```

对于常量的值，Ruby提供了完全控制权，但警告提示给出了清晰的消息。在未来版本中，Ruby可能会强化对常量的控制，因此请尊重这种使用方式，最好不要在程序中对常量重新赋值。

细心的读者会回想起第2章中，我们用Dog和Cat等以大写字母开头的名字来引用类，之所以这么做，是因为类一旦定义好之后，它就是程序的固化部分，与常量的性质类似。

3.2 文本与字符串

如果数字是计算机可以处理的最常见基本数据类型，那么文本就是排在第二位的常用数据。文本的应用无处不在，特别是在与用户沟通的情况下。在本节中，你将掌握怎样随心所欲地操作文本。

3.2.1 字面字符串

在前面的代码示例中，我们已经用过字符串，例如：

```
puts "Hello, world!"
```

字符串是一组任意长度文本字符（包括数字、字母、空格和符号）的集合。在Ruby中所有字符串都是String类的对象，正如调用字符串的class方法所发现的那样，它返回结果如下：

```
puts "Hello, world!".class
```

```
String
```

用前文所示的引号，把字符串直接嵌入在代码中，这种构造就称为字面字符串（string literal）。这与另一种字符串不同，后者的数据来自远程源，例如来自用户输入、文件或因特网，而预先嵌入在程序中的文本都是字面字符串。

与数字相似，我们可对字符串进行操作、相加和比较，也可以把字符串赋值给变量：

```
x = "Test"
y = "String"
puts "Success!" if x + y == "TestString"
```

```
Success!
```

把字面字符串包含在程序中有许多其他方法，例如，你可能想包含多行文本。用引号只适合于单行文本，但如果想扩展到多行，可以试试下面的方法：

```
x = %q{This is a test
of the multi
line capabilities}
```

在本例中，引号被替换成%q{和}。当然，不必非得用花括号，也可以用<和>、(和)，或只是自选的两个分界符，例如!和!。这段代码可以改写成如下形式，其含义完全相同：

```
x = %q!This is a test
of the multi
line capabilities!
```

然而，必须记住一点，如果使用惊叹号作为分界符，那么所引用正文中一旦有惊叹号，就会导致这种方法出错。如果字符串中有分界符符号，则字面字符串会提前终止，Ruby会把后面的文本内容视为错误内容。因此，请慎重选择所用的分界符！

另一种构建字面字符串的方法，是用引入文档（here document），这个概念出自许多其他编程语言。它的构建方式与上例类似，不同之处在于分界符可用多个字符。如下例所示：

```
x = <<END_MY_STRING_PLEASE
This is the string
And a second line
END_MY_STRING_PLEASE
```

在本例中，<<标记了字面字符串的开始，后跟所选择的分界符。字面字符串从下一行开始，以再次遇到分界符为结束。采用这种方法，就不大可能碰到选错分界符的问题，只要你有足够的创意即可！

3.2.2 字符串表达式

用+号可以把"Test"和"String"这两个字符串相加（连接起来），得到"TestString"，从而使下面的比较结果为true，进而向屏幕输出"Success!"：

```
puts "Success!" if "Test" + "String" == "TestString"
```

同样，也可以对字符串进行乘法操作。例如，假设你想把某个字符串复制五次，如下所示：

```
puts "abc" * 5
```

```
abccabccabccabcc
```

你也可以进行“大于”和“小于”的比较操作：

```
puts "x" > "y"
```

```
false
```

```
puts "y" > "x"
```

```
true
```

注 "x" > "y"和"y" > "x"都是表达式，用比较运算符可以得到true或false的结果。

在此情况下，Ruby对字符串中字符所用的数值进行比较。如前所述，字符以数值形式驻留在计算机内存中，因此每个字母和符号都有值，称为ASCII值。这些值本身没有特别重要的含义，但却让我们可以用这种方法，对字母甚至更长的字符串进行比较。如果想知道某个字符的值是多少，可以用如下方法：

```
puts ?x
```

```
120
```

```
puts ?A
```

```
65
```

问号后跟字符则返回整数，表示该字符在ASCII表中的位置，该表是表示字符值的标准。你可以用String类的chr方法，得到相反的效果。例如：

```
puts ?x
puts 120.chr
```

```
120
x
```

注 在此对ASCII字符集作更多解释，超出了本书的范围，但如果你想了解更多，可在网上找到许多资源，其中有个出色的资源是<http://en.wikipedia.org/wiki/ASCII>。

3.2.3 插写

在上面的例子中，用puts方法把代码的结果打印输出到屏幕上，但这个结果需要稍作解释。如果随便哪个用户过来使用你的代码，就不会很清楚将产生什么结果，因为他们没兴趣读你的源代码。因此，你的程序提供用户友好的输出是一件很重要的事。在本例中可以回头使用数字：

```
x = 10
y = 20
puts "#{x} + #{y} = #{x + y}"
```

```
10 + 20 = 30
```

这是幼儿级别的算术，但其结果展示了一个有意思的能力，即可以把表达式（甚至代码逻辑）直接嵌入到字符串中，这一过程称为插写（interpolation）。插写是指表达式结果插入字符串的过程。在字符串中插写的方式，是把表达式放在#{和}符号中。下面是更基础的例子：

```
puts "100 * 5 = #{100 * 5}"
```

```
100 * 5 = 500
```

代码段#{100 * 5}把100 * 5的结果（即500）插写到字符串中指定位置，产生如上所示的输出。再来看下面这行代码：

```
puts "#{x} + #{y} = #{x + y}"
```

首先插写x的值，然后是y的值，然后是x加y的值。每段代码都用相应的数学符号包围起来，然后，你便得到了完整的数学方程：

```
10 + 20 = 30
```

也可以插写其他字符串：

```
x = "cat"
puts "The #{x} in the hat"
```

```
The cat in the hat
```

或用下面更巧妙的办法：

```
puts "It's a #{'bad ' * 5}world"
```

```
It's a bad bad bad bad bad world
```

在本例中，插写一段重复的字符串，把"bad"重复五次。这当然比手工输入快多了！插写也可以用在字符串赋值中：

```
my_string = "It's a #{ "bad " * 5 }world"
puts my_string
```

```
It's a bad bad bad bad bad world
```

值得一提的是，不用插写方式，把表达式放到字符串之外，也可以得到和前面相同的结果。例如：

```
x = 10
y = 20
puts x.to_s + " + " + y.to_s + " = " + (x + y).to_s
puts "#{x} + #{y} = #{x + y}"
```

这两个puts代码行产生相同的输出。第一行代码用字符串相加 (+)，把几个不同的字符串连接起来，x和y的数值用其to_s方法转换成字符串。而第二行puts代码则用插写方式，不需要显式地把数字转换成字符串。

3.2.4 字符串方法

我们已经见识了在表达式中使用字符串，但除了相加和相乘，还可以对字符串做许许多多其他的操作。如在第2章所体验的，可以对字符串调用许多不同方法。表3-3提供了第2章见到的字符串方法概括说明。

表3-3 对字符串"Test"调用不同方法的结果

比较运算	输出结果	比较运算	输出结果
"Test" + "Test"	TestTest	"Test".reverse	tseT
"Test".capitalize	Test	"Test".sum	416
"Test".downcase	test	"Test".swapcase	tEST
"Test".chop	Tes	"Test".upcase	TEST
"Test".hash	-98625764	"Test".upcase.reverse	TSET
"Test".next	Tesu	"Test".upcase.reverse.next	TESU

在表3-3的每个例子中，都使用字符串提供的某个方法，无论是相加、转换成大写、反转，还是仅仅把最末字符增值。我们可以把方法链接起来，如表中最后一个例子所示。首先，创建字面字符串"Test"，然后将其转换成大写，返回TEST，然后再把它反转，返回TSET，再把它最末字符增值，返回TSEU。

在第2章用到的另一个方法是length，如下所示：

```
puts "This is a test".length
```


这些方法都很有用，但你无法用这些方法对字符串做什么特别令人耳目一新的事情。我们来看下一节，对文本本身直接做些有意思的事。

3.2.5 正则表达式与字符串操作

要在高级层面处理字符串，必须学会正则表达式 (regular expression)。从根本上说，正则表达式就是搜索查询，不要把它与本章讨论过的表达式混淆起来。如果在喜欢的搜索引擎中输入ruby，你是期望看到关于Ruby的信息。同样，如果你的正则表达式是ruby，并对某个很长的字符串运行该查询，则你期望得到的是相匹配的内容。因此，正则表达式是一个字符串，它描述在其他字符串中的匹配元素模式。

注 本节仅提供正则表达式的简单介绍。正则表达式是计算机科学的一个主要分支，有许多书籍和网站专门讨论它的用途。Ruby支持大部分标准正则表达式语法，因此从与Ruby无关的其他地方得到的关于正则表达式的知识，在Ruby中仍然有用。

替换

对于字符串，经常要做的操作之一，是把某些内容替换成其他内容，如下例：

```
puts "foobar".sub('bar', 'foo')
```

```
foofoo
```

在本例中，对字符串调用名为sub的方法，该方法把第一次遇到的'bar'（第一个参数），替换成'foo'（第二个参数），替换的结果为foofoo。sub方法只对发现的第一个匹配文本做一次替换，而gsub方法则对所有匹配文本进行多次替换，如下例所示：

```
puts "this is a test".gsub('i', '')
```

```
ths s a test
```

这里把所有字母'i'都替换成空字符串。试试更复杂的模式会怎么样？只匹配字母'i'不是真正的正则表达式。例如，假定我们要把字符串开头两个字符替换成'Hello'：

```
x = "This is a test"
puts x.sub(/^../, 'Hello')
```

```
Hellois is a test
```

在本例中，用sub方法进行了单个替换。传递给sub方法的第一个参数不是字符串，而是正则表达式，斜杠用于表示正则表达式的开头和结尾。在正则表达式中是^..，其中^称为锚，表示正则表达式将从字符串中的任一行开头进行匹配。至于两点，每个点都表示“任何字符”。综上所述，/^../表示“某行开始的开头两个字符”。因此，"This is a test"中的Th被替换成Hello。

同样，如果要改变最后两个字母，可以用另一种锚：

```
x = "This is a test"
```

```
puts x.sub(/..$/, 'Hello')
```

```
This is a teHello
```

这一次，正则表达式匹配字符串中任一行行尾的两个字符。

注 如果想锚到字符串的绝对开头和结尾，可以相应使用`\A`和`\Z`，而`^`和`$`则表示各行的开头和结尾。

用正则表达式进行迭代

在前文中，用过迭代子在数字集合中移动，例如从1计数到10。如果想在字符串中迭代，并分别访问其各个部分，应该怎么办？`scan`正是所需要的迭代子方法：

```
"xyz".scan(/./) { |letter| puts letter }
```

```
x  
y  
z
```

`scan`方法正如其名，它根据传递给它的正则表达式，在字符串中扫描，寻找所有匹配正则表达式的内容。在本例的情况下，提供的正则表达式是每次查找一个字符，这就是分别得到`x`、`y`、`z`输出的原因。查找到的每个字母都被送给代码块，并赋值给`letter`变量，然后打印到屏幕上。我们再试试下面这个更复杂的例子：

```
"This is a test".scan(/../) { |x| puts x }
```

```
Th  
is  
 i  
s  
a  
te  
st
```

这一次，你一次扫描两个字符。太容易了！当然，扫描所有字符会产生有些怪异的输出，如空格也混杂在内。我们来调整一下正则表达式，让它只匹配字母和数字，如下所示：

```
"This is a test".scan(/\w\w/) { |x| puts x }
```

```
Th  
is  
is  
te  
st
```

在正则表达式中，有一些特殊字符以反斜杠表示，它们有特殊含义。`\w`表示“字母表中的任何字符或下划线”。还有很多其他特殊字符，如表3-4所示。

表3-4 正则表达式中基本的特殊字符和符号

字 符	含 义	字 符	含 义
^	用于行开始的锚	\w	不匹配\w的任何内容
\$	用于行结束的锚	\d	任何数字
\A	用于字符串开始的锚	\D	不匹配\d的任何内容（非数字）
\Z	用于字符串结束的锚	\s	空白（空格、制表符、换行符等）
.	任意字符	\S	非空白（任何可见字符）
\w	任何字母、数字或下划线		

运用表3-4的知识，可以轻易地从字符串中解析出数字：

```
"The car costs $1000 and the cat costs $10".scan(/\d+/) do |x|
  puts x
end
```

```
1000
10
```

刚刚让Ruby从任意英语文本中解析出有意义的内容！scan方法的用法和前面一样，但这次传递给它的正则表达式，使用\d来匹配任何数字，而\d之后的+，让\d在一行中尽可能多地匹配数字。这表示它既匹配1000，也匹配10，而不是一次匹配一个数字。要证明这一点，试试下面的代码：

```
"The car costs $1000 and the cat costs $10".scan(/\d/) do |x|
  puts x
end
```

```
1
0
0
0
0
1
0
```

因此，在正则表达式中，字符之后的+表示匹配一个或多个此类字符。还有其他种类的修饰符，如表3-5所示。

表3-5 正则表达式字符和子表达式修饰符

修 饰 符	说 明
*	匹配零次或多次前面紧跟的字符，并尽量多地匹配
+	匹配一次或多次前面紧跟的字符，并尽量多地匹配
*?	匹配零次或多次前面紧跟的字符，并尽量少地匹配
++	匹配一次或多次前面紧跟的字符，并尽量少地匹配
?	要么匹配一次，要么全不匹配前面紧跟的字符
{x}	匹配x次前面紧跟的字符
{x,y}	匹配最少x次、最多y次前面紧跟的字符

关于正则表达式，目前需要理解的最后一个重要方面，是字符类别（character classes）。这些字符类别让可以针对特定字符集合进行匹配。例如，可以扫描字符串中所有元音字母：

```
"This is a test".scan(/[aeiou]/) { |x| puts x }
```

```
i
i
a
e
```

[aeiou]表示“匹配任何a、e、i、o或u”。还可以在方括号中指定字符的范围，如下所示：

```
"This is a test".scan(/[a-m]/) { |x| puts x }
```

```
h
i
i
a
e
```

这次扫描匹配a到m之间的所有小写字母。

正则表达式可以是复杂难解的，市面上有比本书还厚的专门讲解正则表达式的书。大多数编程人员只需要理解基础知识即可，因为更高级的技术方法将会随时间推移而慢慢清晰起来。但不管怎么说，如果对正则表达式有经验，并掌握它，那么它就是一种强大的工具。

在后续章节，将使用并扩展本节代码示例中讨论的技术方法。

匹配查询

做替换和从字符串中解析某些文本是很有用的，但有时我们只想检查某个字符串是否匹配所选择的模式。例如，可能想快速得知字符串是否包含元音字母：

```
puts "String has vowels" if "This is a test" =~ /[aeiou]/
```

在本例中，`=~`是运算符的另一种形式——匹配查询运算符。如果字符串与运算符后面的正则表达式能够匹配，则该表达式结果为`true`。当然，也可以做相反的操作：

```
puts "String contains no digits" unless "This is a test" =~ /[0-9]/
```

这一次，代码的含义是除非0到9的数字匹配测试字符串，否则就告诉用户，字符串中没有数字。

还可以使用String类提供的名为`match`的方法，`=~`是根据正则表达式是否匹配字符串，返回`true`或`false`，而`match`方法则提供了许多更强大的能力。下面是个简单的例子：

```
puts "String has vowels" if "This is a test".match(/[aeiou]/)
```

这行代码看起来与前面的例子几乎一样，但由于`match`方法不需要用正则表达式作为参数，它把提供给它的任何字符串都转换成正则表达式，因此下面这行代码同样可以：

```
puts "String has vowels" if "This is a test".match("[aeiou]")
```

如果正则表达式能够由用户提供，或从文件及其他外部来源载入，这种功能是非常有用的。

在正则表达式中，如果其中一部分由括号(和)包围，则该部分正则表达式所匹配的数据可以单独使用，与剩余的其他数据不相干。match方法让你可以访问这些数据：

```
x = "This is a test".match(/(\w+) (\w+)/)
puts x[0]
puts x[1]
puts x[2]
```

```
This is
This
is
```

match方法返回MatchData对象，该对象的访问方法与数组类似。第一个元素包含整个正则表达式所匹配的数据，但每个后续元素都只包含正则表达式每块单元所匹配的内容。在本例中，第一块(\w+)匹配This，而第二块(\w+)则匹配is。

注 匹配查询可能比上述内容更复杂，当你开始编写第一个完整的Ruby程序时，我将在下一章讨论更高级的用法。

3.3 数组与列表

到目前为止，本章创建了数字和字符串对象的单个实例，并对它们进行操作。下面我们将有必要创建这些对象的集合，并把它们当成列表来操作。在Ruby中，可以用数组(array)来表示对象的有序集合。

3.3.1 基本数组

下面是一个基本的数组：

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

这个数组有四个元素。每个元素都是整数，用逗号分隔相邻的两个元素。所有元素都包含在方括号中。

可通过元素的索引序号（即其在数组中的位置）来访问它们。要访问特定的元素、数组或包含数组的变量，可通过包含在方括号中的索引，称其为元素引用(element reference)。例如：

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

```
3
```

与大多数编程语言一样，Ruby的数组索引也是从0开始，因此数组的第一个元素是元素0，第二个元素是元素1，诸如此类。在我们的例子中，x[2]表示所谓的数组第三个元素，在本例中是指数字3的对象。要改变元素的值，可以直接对其赋予新值，也可以像本章前文中操作数字和字符串一样操作元素：

```
x[2] += 1
```

```
puts x[2]
```

4

或：

```
x[2] = "Fish" * 3
puts x[2]
```

FishFishFish

数组不需要设置预定义空间，或手工分配元素。我们可以创建空数组，如下所示：

```
x = []
```

这个数组是空数组，因此如果定位某个元素，例如`x[5]`，将返回空值。然而我们可以用压入数据的方法，向数组末尾加入内容，如下所示：

```
x = []
x << "Word"
```

在此操作之后，数组即包含单个元素：“Word”字符串。对于数组来说，`<<`是把数据放到数组末尾的运算符。也可以调用`push`方法，它们具有相同的效果。

还可以从数组中一个个地删除数据。传统上讲，数组是个“先进先出”系统，数据既能被压入数组末尾，也能从数组末尾弹出（弹出是从数组末尾检索数据并同时将其删除的过程）。

```
x = []
x << "Word"
x << "Play"
x << "Fun"
puts x.pop
puts x.pop
puts x.length
```

```
Fun
Play
1
```

把“Word”、“Play”和“Fun”压入到`x`所持有的数组中，然后在屏幕显示首先被“弹出”的元素。元素从数组末尾倒序弹出，因此Fun首先弹出，然后是Play。为了进行良好的度量，随即调用命名为`length`的方法（也可以用`size`方法，结果是一样的），此时数组的长度为1，因为当前数组中只有“Word”。

另一个有用的特点是，如果数组中全都是字符串，可以对该数组调用名为`join`的方法，如果把所有元素连接起来，则形成一个大字符串：

```
x = ["Word", "Play", "Fun"]
puts x.join
```

WordPlayFun

`join`方法可接受一个可选参数，放在结果字符串的每个元素之间：

```
x = ["Word", "Play", "Fun"]
puts x.join(',')
```

```
Word, Play, Fun
```

这一次，把数组元素连接起来，但在每个元素之间放了逗号和空格。这个输出结果看起来更简洁。

3.3.2 字符串切分成数组

在关于字符串的章节中，用scan方法在字符串内容中迭代扫描，查找匹配正则表达式模式的字符。对于scan方法，你使用了一个代码块，它接受每次查找字符的集合，并将其显示在屏幕上。然而，如果不用代码块进行扫描，它将返回一个数组，包含字符串中所有匹配的内容，如下所示：

```
puts "This is a test".scan(/\w/).join(',')
```

```
T,h,i,s,i,s,a,t,e,s,t
```

首先你定义一个字面字符串，然后对它进行扫描，查找字母表中的字符（用/\w/），最后把返回数组的所有元素连接起来，以逗号分隔。

如果不想扫描特定字符，而是想把字符串切分成几个部分，应该怎么办？我们可以使用split方法，告诉它把字符串按句点切分成字符串数组，如下所示：

```
puts "Short sentence. Another. No more.".split(/\./).inspect
```

```
["Short sentence", " Another", " No more"]
```

这里有几个要点。首先，如果你在正则表达式中使用.而不是\.,则会按每个字符，而不是按整句进行切分，因为在正则表达式中，.表示“任何字符”，因此你需要在前面加上反斜杠，将其转义（escape）（转义是特别标注某个字符，令其含义清晰的过程）。其次，无需连接并打印出结果，通过使用inspect方法，便可以得到更紧凑的结果。

在Ruby中，几乎所有内置类都有inspect方法，它给出对象的文本化表示形式。例如，从上文结果数组的输出可以看出，它与我们自己创建数组的输出是一样的。在摸索试验和调试程序时，inspect是个极其有用的方法！

split也能胜任按换行符或同时按多个字符进行切分，以得到更简洁的结果：

```
puts "Words with lots of spaces".split(/\s+/).inspect
```

```
["Words", "with", "lots", "of", "spaces"]
```

用Ruby和正则表达式，则再也不会遭受文本处理问题的困扰了！

3.3.3 数组迭代

在数组中进行迭代的方法很简单，用each方法即可。each方法遍历数组的每个元素，并把该元素当作参数，传递给你提供的代码块。例如：

```
[1, "test", 2, 3, 4].each { |element| puts element.to_s + "X" }
```

```
1X
testX
2X
3X
4X
```

尽管each方法可以在数组的元素中进行迭代，你还可以用collect方法，对数组进行实时转换：

```
[1, 2, 3, 4].collect { |element| element * 2 }
```

```
[2, 4, 6, 8]
```

collect方法在数组中逐个元素地进行迭代，并用代码块中表达式的结果，对该元素进行赋值。在本例中，把元素的值乘以2。

来自非动态语言、甚至非面向对象语言阵营的程序员，可能认为这些技术方法很时髦。如果有必要，也可以用“老式方法”编写Ruby程序：

```
a = [1, "test", 2, 3, 4]
i = 0

while (i < a.length)
  puts a[i].to_s + "X"
  i += 1
end
```

这与前面each例子类似，但对于传统程序员（例如用C或BASIC语言）来说，更熟悉这种在数组中循环的方法。然而，任何人都能立即看出，为什么迭代子、代码块和诸如each和collect等方法更适合Ruby，因为它们显著地提升了代码的易读性和可理解度。

3.3.4 数组的其他方法

数组有许多有意思的方法，本节介绍其中一些内容。

数组加法和串连

如果有两个数组，我们可以快速合并为一个：

```
x = [1, 2, 3]
y = ["a", "b", "c"]
z = x + y
puts z.inspect
```

```
[1, 2, 3, "a", "b", "c"]
```

数组减法和区分

也可以比较两个数组，方法是用一个数组减去另一个。这一方法结果是从主数组中删除两个数组中都有的元素：

```
x = [1, 2, 3, 4, 5]
y = [1, 2, 3]
z = x - y
puts z.inspect
```

```
[4, 5]
```

检查数组是否为空

如果打算在数组中迭代，可能想检查它是否还有元素。可以用`array.size`或`array.length`是否大于0的方法来检查，更流行的快捷方法是用`empty?`：

```
x = []
puts "x is empty" if x.empty?
```

```
x is empty
```

检查数组是否有某个元素

用`include?`方法，如果给定参数在数组中能找到则返回`true`，否则返回`false`：

```
x = [1, 2, 3]
puts x.include?("x")
puts x.include?(3)
```

```
false
true
```

访问数组中第一个和最后一个元素

访问数组中第一个和最后一个元素的方法很简单，可用`first`和`last`方法：

```
x = [1, 2, 3]
puts x.first
puts x.last
```

```
1
3
```

如果对`first`或`last`方法指定数字参数，则得到数组从开始或最末起的`n`个元素：

```
x = [1, 2, 3]
puts x.first(2).join("-")
```

```
1-2
```

反转数组元素的顺序

与字符串一样，数组也可以被反转：

```
x = [1, 2, 3]
puts x.reverse.inspect
```

```
[3, 2, 1]
```


3.4 散列表

数组是对象的集合，散列表也是。然而，散列表有不同的存储格式，以及在集合中定义每个对象的不同方法。散列表中的对象不在列表中给定位置，而是给定一个指向对象的键（key）。散列表更像字典，而不像列表，因为它没有确定的顺序，只有键和值之间的简单链接。下面是个基本的散列表，其中有两个元素：

```
dictionary = { 'cat' => 'feline animal', 'dog' => 'canine animal' }
```

保存散列表的变量名为dictionary，它包含两个元素，我们可以证明一下：

```
puts dictionary.size
```

```
2
```

其中一个元素的键为cat，值为feline animal；另一个元素的键为dog，值为canine animal。与数组类似，我们可以用方括号来引用要检索的元素。例如：

```
puts dictionary['cat']
```

```
feline animal
```

如你所见，散列表可被视为数组，其元素具有名字，而非位置序号。我们甚至可以用与数组相同的方法，来改变散列表元素的值：

```
dictionary['cat'] = "fluffy animal"
```

```
puts dictionary['cat']
```

```
fluffy animal
```

注 虽然以下内容不会马上派上用场，但值得一提。键和值都可以是任何类型的对象。因此，可以把数组（甚至是另一个散列表）用作键。当将来想处理更复杂的数据结构时，这一点可能派得上用场。

3.4.1 散列表的基础方法

和数组类似，散列表有许多有用的方法，本节对此进行介绍。

在散列表元素中迭代

对于数组，可以用each方法在数组元素中迭代。对散列表也可以进行此项操作，但由于散列表每个元素都用键，因此迭代的响应顺序不是固定的：

```
x = { "a" => 1, "b" => 2 }
```

```
x.each { |key, value| puts "#{key} equals #{value}" }
```

```
a equals 1
```

```
b equals 2
```

散列表的each迭代子方法是将两个参数传递给代码块：第一个是键，第二个是该键对应的

值。在本例中，把它们赋给名为key和value的变量，并用字符串插写方法，将其内容显示到屏幕上。

检索键

有时我们对散列表中的值并不感兴趣，而对散列表包含什么内容感兴趣。实现此目的的有效方法是查看键。Ruby提供了keys方法，可以很容易地立即查看任何散列表的键：

```
x = { "a" => 1, "b" => 2, "c" => 3 }
puts x.keys.inspect
```

```
["a", "b", "c"]
```

keys方法返回数组，其中包含散列表中的所有键。如果你状态良好的话，就能想到values方法也会返回数组，其中包含散列表中的所有值。但一般来说，我们会通过键查看值。

删除散列表中的元素

删除散列表元素很简单，用delete方法即可。只需传递键作为参数，该元素即可被删除：

```
x = { "a" => 1, "b" => 2 }
x.delete("a")
puts x.inspect
```

```
{"b"=>2}
```

有条件删除散列表中的元素

假设我们想删除散列表中值小于某个标准的所有元素：

```
x = { "a" => 100, "b" => 20 }
x.delete_if { |key, value| value < 25 }
puts x.inspect
```

```
{"a"=>100}
```

3.4.2 散列表中的散列表

散列表中有可能使用散列表（或任何类型的对象），甚至数组，甚至散列表中的散列表！因为一切都是对象，而散列表和数组可以包括任何类型的对象，因此可以用散列表和数组创建巨大的树结构，下面是个演示的例子：

```
people = {
  'fred'=>{
    'name'=>'Fred Elliott',
    'age'=>63,
    'gender'=>'male',
    'favorite painters'=>['Monet','Constable','Da Vinci']
  },
  'janet'=>{
    'name'=>'Janet S Porter',
    'age'=>55,
```

```
      'gender'=>'female'
    }
  }

  puts people ['fred']['age']
  puts people ['janet']['gender']

  puts people ['janet'].inspect
```

```
63
female
{"name"=>"Janet S Porter", "gender"=>"female", "age"=>55}
```

尽管这个散列表的结构初看起来有些令人迷惑，但当你把它划分为几部分时，就变得相当简单。'fred'和'janet'部分本身是简单的散列表，它们被包装在另一个巨型散列表中，并赋值给people。在查询该巨型散列表的代码时，你只是把每个查询串连起来，形成puts people['fred']['age']。这段代码首先得到people['fred']，返回Fred的散列表，然后再对此散列表访问['age']，得到结果63。

甚至访问嵌在Fred散列表中的数组也很简单：

```
puts people['fred']['favorite painters'].length
puts people['fred']['favorite painters'].join(", ")
```

```
3
Monet, Constable, Da Vinci
```

在后续章节中，我们将更多地使用这些技术方法，并作更深入的解释。

3.5 流程控制

在本章中，我们用if和unless，根据条件执行不同的操作，并对其进行比较。if和unless对单行代码很有效，当它们与大段代码合并使用时，将变得更有威力。本节将带你见识Ruby怎样用这些程序和其他语法结构，来控制程序流程。

3.5.1 if与unless

本章首次使用if的是下面的例子：

```
age = 10
puts "You're too young to use this system" if age < 18
```

如果age的值小于18，则把字符串打印输出到屏幕上。下面代码的功能与此完全相同：

```
age = 10
if age < 18
  puts "You're too young to use this system"
end
```

这段代码看起来似乎很熟悉，如果表达式为true就执行if和end之间代码，而不是把if表

达式放在单行代码的末尾。此外，这种结构可以在if语句和end行之间放置任意多行代码：

```
age = 10
if age < 18
  puts "You're too young to use this system"
  puts "So we're going to exit your program now"
  exit
end
```

值得一提的是，unless可以实现相同的效果，因为unless正是if的反义词：

```
age = 10
unless age >= 18
  puts "You're too young to use this system"
  puts "So we're going to exit your program now"
  exit
end
```

也可以嵌套代码逻辑，如下例所示：

```
age = 19
if age < 21
  puts "You can't drink in most of the United States"
  if age >= 18
    puts "But you can in the United Kingdom!"
  end
end
```

if和unless也提供了else条件，用来界定主表达式为false时想执行的代码：

```
age = 10
if age < 18
  puts "You're too young to use this system"
else
  puts "You can use this system"
end
```

3.5.2 ?:, 三元运算符

三元运算符可以让表达式包含小型if/else语句，构建这一运算符，完全是为了可选用途，有些开发人员完全把它遗忘了。但由于它可以用来生成紧凑的代码，因此值得早点学会。我们来看一个例子：

```
age = 10
type = age < 18 ? "child" : "adult"
puts "You are a " + type
```

第二行包含三元运算符，它是以把表达式的结果赋值给变量type开始，表达式是age<18 ? "child": "adult"，其结构规则如下：

```
<condition> ? <result if condition is true> : <result if condition is false>
```

在我们的例子中，age<18返回true，因此返回第一个结果"child"并赋值给type。但如

果`age < 18`是`false`，则返回`"adult"`结果。

我们来看另一种实现方法：

```
age = 10
type = 'child' if age < 18
type = 'adult' unless age < 18
puts "You are a " + type
```

两次比较让这段代码更难阅读。另一种实现方法是用多行的`if/else`选项：

```
age = 10
if age < 18
  type = 'child'
else
  type = 'adult'
end
puts "You are a " + type
```

三元运算符显示了其简洁而能立即得到的好处，由于它可以在一行代码中构建表达式，因此可以很容易地使用方法调用或在其他表达式中调用，这是`if`语句无法做到的。我们来看本节开头示例的更简单版本：

```
age = 10
puts "You are a " + (age < 18 ? "child" : "adult")
```

3.5.3 elsif与case

有时需要用同一个变量同作几次比较，也可以用前面提到的`if`语句来实现：

```
fruit = "orange"
color = "orange" if fruit == "orange"
color = "green" if fruit == "apple"
color = "yellow" if fruit == "banana"
```

如果`fruit`不等于`orange`、`apple`或`banana`，我们又想用`else`来赋值，那么很快就会弄得一团糟，因为你需要创建`if`块来检查这些词是否出现，然后执行和前面一样的比较。另一种方法是用`elsif`，表示“否则如果”：

```
fruit = "orange"
if fruit == "orange"
  color = "orange"
elsif fruit == "apple"
  color = "green"
elsif fruit == "banana"
  color = "yellow"
else
  color = "unknown"
end
```

`elsif`块和`else`块类似，区别在于你可以指定要执行的全新比较表达式，如果都不匹配，

则指定要执行的常规else块。

这种技术方法的另一种变化形式是用case块。把我们前面的例子用case块来改写，如下所示：

```
fruit = "orange"
case fruit
  when "orange"
    color = "orange"
  when "apple"
    color = "green"
  when "banana"
    color = "yellow"
  else
    color = "unknown"
end
```

除了语法更清晰外，这段代码与if块类似。case块首先处理表达式，然后找出匹配该表达式的when块并执行，如果找不到匹配的when块，则转而执行case块中的else块。

case还有另一个巧妙的手法。由于所有Ruby表达式都返回结果，因此可以把上例改得更短：

```
fruit = "orange"
color = case fruit
  when "orange"
    "orange"
  when "apple"
    "green"
  when "banana"
    "yellow"
  else
    "unknown"
end
```

在本例中，我们用了case块，然而把任何被执行内部块的结果直接赋值给color。

3.5.4 while与until

在前面的章节中，我们已经演示用迭代子方法进行循环，如下所示：

```
1.upto(5) { |number| puts number }
```

```
1
2
3
4
5
```

但也可以用其他方法来循环代码。while和until根据每次循环的比较结果，来循环代码：

```
x = 1
while x < 100
```

```
puts x
x = x * 2
end
```

```
1
2
4
8
16
32
64
```

在本例中，有个while块，它标示了一段循环代码，只要满足 $x < 100$ 表达式，这段代码就反复循环下去。因此，在每次循环之后， x 的值翻倍并打印输出到屏幕。一旦 x 等于或超过100，则循环结束。

until提供了相反的功能，直到某个条件满足才循环：

```
x = 1
until x > 99
  puts x
  x = x * 2
end
```

也可以在一行代码中使用while和until，与if和unless的用法类似：

```
i = 1
i = i * 2 until i > 1000
puts i
```

```
1024
```

i 的值一遍遍地翻倍，直到结果超过1000，此时循环才终止。

3.5.5 代码块

本章已经多次用到代码块，例如：

```
x = [1, 2, 3]
x.each { |y| puts y }
```

```
1
2
3
```

each方法接受单个代码块作为参数，该代码块的定义位于{和}符号之间，或位于do和end分界符之间：

```
x = [1, 2, 3]
x.each do |y|
  puts y
end
```

```
end
```

在{和}或do和end之间的代码是代码块，本质上是匿名的、无名的方法或函数。这段代码传递给each方法，由后者对于数组的每个元素运行该代码块。

我们可以编写自己的方法，来处理代码块。例如：

```
def each_vowel(&code_block)
  %w{a e i o u}.each { |vowel| code_block.call(vowel) }
end

each_vowel { |vowel| puts vowel }
```

```
a
e
i
o
u
```

each_vowel接受代码块参数，在其方法定义中，code_block参数名之前的宏符号(&)，表示该参数是代码块。该方法对字面数组%w{a e i o u}的每个元音字母进行迭代，并调用code_block的call方法，对每个元音字母执行代码块，将元音字母作为vowel变量参数，传递给call方法。

注 以这种方法传递的代码块，产生的结果是对象，它有许多自己的方法，例如call。请记住，在Ruby中一切都是对象！

另一种技术方法是用yield方法，它能自动检测传递给它的代码块，并将控制权移交给该代码块：

```
def each_vowel
  %w{a e i o u}.each { |vowel| yield vowel }
end

each_vowel { |vowel| puts vowel }
```

本例的功能与上例完全相同，尽管看起来不太明显，在函数定义中看不到它是如何接受任何代码块的。到底用哪种方法，选择权在你。

注 一次只能传递一个代码块，任何方法都不可能接受两个或多个代码块作为参数。但代码块本身可以接受零个、一个或多个参数。

也可以用lambda方法，把代码块存储在变量中：

```
print_parameter_to_screen = lambda { |x| puts x }
print_parameter_to_screen.call(100)
```

100

与方法接受代码块类似，用lambda对象的call方法来执行代码块，以及接受传递来的任

何参数。

注 用lambda一词是因为它在其他地方和其他编程语言中很流行。你当然可以继续称其为代码块，有时也叫它过程（Proc）。

3.6 其他有用的构造元素

到目前为止，本章讨论了数字、字符串、数组和散列表等主要的内置类。这几种类型的对象可以让你做很多事情，并将它们广泛用于所有程序中。第6章将对对象进行深入讨论，但在此之前，还有几个其他重点内容需要先介绍一下。

3.6.1 日期与时间

在许多计算机程序中，有个有用的概念叫做时间，其表示形式是日期和时间。Ruby提供了名为Time的类来处理这些概念。

Time的内部机制是用微秒值保存时间，这是根据UNIX的时间基准里程碑（UNIX time epoch）：格林尼治标准时间（GMT）/协调世界时（UTC）1970年元月1日0时0分0秒。有了这个时间基准里程碑，就可以很容易用标准的比较运算符，例如<和>，进行时间的比较。

我们来看怎样使用Time类：

```
puts Time.now
```

```
Tue Mar 27 00:00:00 +0100 2007
```

Time.now创建了Time类的实例，该实例的时间被设为当前时间。但由于我们要把它打印输出的屏幕，因此它被转换成上述字符串。

我们还可以用增加或减少秒数的方法来操作时间对象，例如：

```
puts Time.now
puts Time.now - 10
puts Time.now + 86400
```

```
Tue Mar 27 00:00:00 +0100 2007
Tue Mar 26 23:59:50 +0100 2007
Tue Mar 28 00:00:00 +0100 2007
```

在第一个例子中，我们先打印输出当前时间，然后把当前时间减去10秒，再把当前时间加上86 400秒（正好一整天）。由于时间如此容易操作，所以有些开发人员对Fixnum类进行扩展，加上某些辅助方法，让日期操作更容易：

```
class Fixnum
  def seconds
    self
  end
  def minutes
    self * 60
  end
end
```

```

end
def hours
  self *60 *60
end
def days
  self *60 *60 *24
end
end
end

puts Time.now
puts Time.now +10.minutes
puts Time.now +16.hours
puts Time.now -7.days

```

```

Tue Mar 27 00:00:00 +0100 2007
Tue Mar 27 00:10:00 +0100 2007
Tue Mar 27 16:00:00 +0100 2007
Mon Mar 19 23:00:00 +0000 2007

```

如果这段代码看起来不是很熟悉，且令人迷惑，请别担心，因为我们将在后续章节进一步讨论这种类型的技术方法。但请务必注意最后几个puts语句的风格。有了这些辅助方法，日期操作就变得很容易！

Time类还允许根据任意日期创建Time对象：

```
Time.local(year, month, day, hour, min, sec, msec)
```

上述代码根据当前（本地）时区创建Time对象，从month开始的所有参数都是可选的，默认值为1或0。你还可以指定月份数值（1~12）或英文月份名的三字母缩写。

```
Time.gm(year, month, day, hour, min, sec, msec)
```

上述代码根据GMT/UTC来创建Time对象，所需参数与Time.local相同。

```
Time.utc(year, month, day, hour, min, sec, msec)
```

上述代码与Time.gm同义，尽管有些人可能更喜欢这个方法的名字。

你还可以把Time对象转换成整数形式，其值是自UNIX时间基准里程碑以来的秒数：

```
Time.gm(2007, 05).to_i
```

```
1177977600
```

同样，你也可以把基准时间转换回Time对象。如果要把时间和日期保存到文件，或某种只需整数而非整个Time对象的格式，这种方法就很有用：

```

epoch_time = Time.gm(2007, 5).to_i
t = Time.at(epoch_time)
puts t.year, t.month, t.day

```

```

2007
5
1

```


除了展示在Time对象和基准时间之间的转换，这段代码还展示了Time对象拥有检索某个阶段日期/时间的方法。这些方法的列表如表3-6所示。

表3-6 Time对象用于访问日期/时间属性的方法

方 法	方法的返回值
hour	表示24小时格式的数字（例如21表示下午9点）
min	整点之后经过的分钟数
sec	整分钟之后经过的秒数
usec	整秒之后经过的微秒数（一秒钟有1百万微秒）
day	日期是该月的第几天
mday	与day方法同义，被视为“月份”天数
wday	按周计的天数（星期天是0，星期六是6）
yday	按年计的天数
month	日期的月份数值（例如11表示11月）
year	日期关联的年份
zone	返回时间关联的时区名
utc?	根据时间/日期是否在UTC/GMT时区，返回true或false
gmt?	与utc?方法同义，供喜欢GMT一词的人使用

请注意，这些方法是从日期或时间中检索其相应属性，而不能用于设置属性。如果你想修改日期或时间，要么增加或减少秒数，要么用Time.gm或Time.local方法创建新的Time对象。

注 在第16章中，你会看到有个名为Chronic（惯用法）的Ruby程序库，它能让你以自然的英语风格指定日期和时间，并将其转换成合法的Time对象。

3.6.2 大数字

关于国际象棋的发明，人们常说的故事是围绕大数字的解决。强权的国王征求游戏供自己打发闲暇时光，有个贫穷的数学家为他发明了（国际）象棋游戏，深受国王喜爱。国王打算奖赏这个数学家，要他提出自己想要的奖赏。数学家说他要麦子，分放在棋盘上。第一方格放1粒，第二方格放2粒，第三方格放4粒，如此继续下去，每格比前一格翻一倍，直至放满所有方格。国王认为数学家是个傻瓜，他觉得放满整个棋盘用到的麦子太少了。

我们通过Ruby用迭代子和一些插写功能，创建这个情况的模拟程序：

```
rice_on_square = 1
64.times do |square|
  puts "On square #{square + 1} are #{rice_on_square} grain(s)"
  rice_on_square *= 2
end
```

会得到如下结果：

```
On square 1 are 1 grain(s)
On square 2 are 2 grain(s)
```

```

On square 3 are 4 grain(s)
On square 4 are 8 grain(s)
On square 5 are 16 grain(s)
On square 6 are 32 grain(s)
On square 7 are 64 grain(s)
On square 8 are 128 grain(s)
[Results for squares 9 through 61 trimmed for brevity..]
On square 62 are 2305843009213693952 grain(s)
On square 63 are 4611686018427387904 grain(s)
On square 64 are 9223372036854775808 grain(s)

```

到第64方格，要为这个方格放上亿万粒麦子！这个故事的结局是国王意识到他无法满足这个要求。不过这也证明了Ruby能够处理极其巨大的数字，而且与许多其他编程语言不同，Ruby没有任何不方便的限制。

其他语言常常有数字大小的限制，通常是32个二进制位，即强制使用32位整数的语言最大值极限大约为42亿。大多数操作系统和计算机架构体系也有类似的限制。而Ruby与众不同，它能在计算机原生（即容易地）处理的数字，和需要更多预处理的数字之间，进行无缝地转换。此外，它是用不同的类来完成这一工作的，一个叫做Fixnum类，表示容易管理的较小数字；另一个叫做Bignum类，表示Ruby需要内部管理的“大”数字。在大多数系统中，边界值数字是1 073 741 823，你可以用irb来体验一下：

```
puts 1073741823.class
```

```
Fixnum
```

```
puts 1073741824.class
```

```
Bignum
```

如果得到的结果有所不同，请别担心。Ruby会帮你处理Bignum和Fixnum数字，也可以进行算术和其他运算，不会有任何问题。这一结果根据计算机体系架构而有所不同，但无须担心，这些变化都完成由Ruby来处理。

3.6.3 范围

在某些情况下，能够保存列表的概念，而不是保存其实际内容，是一件很有用的事。例如，如果想表示A到Z的所有字母，可以创建数组，如下所示：

```
x = ['A', 'B', 'C', 'D', 'E' .. and so on.. ]
```

这的确不错，尽管它只保存了“A到Z之间的所有内容”这一概念。而运用范围（Range）也可以做到。范围的表示方法如下：

```
('A'..'Z')
```

就其本身来说，范围没有太大用处，但Range类提供了简单的to_a方法，可把范围转换成数组。如下面的一行代码所示：

```
('A'..'Z').to_a.each { |letter| print letter }
```

这段代码很紧凑，但确实完成了工作。它把'A'到'Z'的范围转换成26个元素的数组，每个元素包含字母表中的一个字母，然后用前面数组章节用过的each方法，对每个元素进行迭代，把值传递给letter，然后把letter打印输出到屏幕。

注 因为用的是print方法而不是puts方法，字母将一个接一个地打印输出在同一行，而puts方法则每次从新行开始打印输出。

检验某数据是否包含在范围指定的对象集合中，有时也是很有用的。例如，有了('A'.....'Z')范围，你可以用include方法，检查R是否在此范围内，如下所示：

```
print "R is within A to Z!" if ('A'..'Z').include?('R')
```

```
R is within A to Z!
```

然而，“r”是小写字母，因此不在此范围内：

```
print "R is within A to Z!" if ('A'..'Z').include?('r')
```

```
=> nil
```

还可以把范围当成数组的索引，一次选择多个元素：

```
a = [2, 4, 6, 8, 10, 12]
puts a[1..3].inspect
```

```
[4, 6, 8]
```

同样，也可以用范围来一次设置多个元素：

```
a[1..3] = ["a", "b", "c"]
puts a.inspect
```

```
[2, "a", "b", "c", 10, 12]
```

此外，你可以对隶属许多不同类的对象使用范围，包括自己创建的类的对象。

3.6.4 符号

在主流语言中，符号（Symbol）是Ruby相当独特的概念（尽管LISP和Erlang语言也有类似的概念）。符号的威力强大，大多数专业Ruby开发人员都用它。尽管大多数初学者容易迷糊不解，然而它值得学习。我们直接来看一个演示的例子：

```
current_situation = :good
puts "Everything is fine" if current_situation == :good
puts "PANIC!" if current_situation == :bad
```

```
Everything is fine
```

在本例中，:good和:bad都是符号。符号不包含值或对象，不像变量那样，而是用作代码中固定的名字。例如，前面的代码中你可以方便地把符号替换成字符串，如下所示：

```
current_situation = "good"
puts "Everything is fine" if current_situation == "good"
puts "PANIC!" if current_situation == "bad"
```

这段代码的结果与上一段代码相同，但效率不高。在本例中，每次提到“good”和“bad”时，都在内存中分别创建了新对象，而符号则仅是引用值，只初始化一次。在第一个代码示例中，只存在`:good`和`:bad`，而第二个代码示例中，有“good”、“good”和“bad”三个完整的字符串占用内存。

符号还让代码在许多情况下看起来更简洁。有时你常常会用符号来指定方法的参数名。因为把不同的字符串数据和固定信息用符号来表示，会产生更容易阅读的代码。

你可能要把符号看成是无值的字面常量，但符号的名字是最重要的因素。如果你把`:good`符号赋值给变量，并在后面的代码中将该变量与`:good`相比较，将会得到匹配的结果。因此，在不需要保存实际值，只需保存概念或选项的情况下，符号很有用。

符号在创建散列表时特别有用，可以用它来区分键和值。例如：

```
s = { :key => 'value' }
```

这种方法还可用于规约或统一使用哪个键名的情况下：

```
person1 = { :name => "Fred", :age => 20, :gender => :male }
person2 = { :name => "Laura", :age => 23, :gender => :female }
```

Ruby提供的类中，有许多方法采用这种风格来传入信息（也常常用于返回值）。在本书中你将一直看到这种构造方法的例子。

3.6.5 类间转换

数字、字符串、符号和其他类型的数据，都是隶属于不同类的对象。数字属于`Fixnum`、`Bignum`、`Float`或`Integer`类，字符串是`String`类的对象，符号是`Symbol`类的对象，诸如此类。

在大多数情况下，我们可以在不同的类之间转换对象，因此数字可变成字符串，而字符串也可变成数字。我们来看下面的例子：

```
puts "12" + "10"
puts 12 + 10
```

```
1210
22
```

第一行代码把两个字符串加起来，它们正好表示数字，得到1210的结果。第二行把两个数字加起来，得到22的结果。

然而，可以把这些对象转换成不同的类的表示形式：

```
puts "12".to_i + "10".to_i
puts 12.to_s + 10.to_s
```

数据表用`to_`方法转换。`String`类提供`to_i`和`to_f`方法，把字符串相应转换成`Integer`或`Float`类的对象。`String`类还提供了`to_sym`方法，把字符串转换成符号。符号提供了反向方法，用`to_s`方法可以把它转换成字符串。

同样，数字类支持`to_s`方法，把自己转换成文本表示形式，同理`to_i`和`to_f`则是转换成整数和浮点数的方法。

3.7 小结

在本章中，你见识了所有计算机程序的关键构建元素——数据、表达式和逻辑，并掌握怎样用Ruby来实现它们。本章的主题为本书其他每一章节提供了重要的基础，因为将来编写的几乎每一行代码都包含表达式、迭代子或某种逻辑。

注 请务必记住，由于Ruby很宽泛，在此我无法涵盖所有各个类和方法的组合。在Ruby中还有更多的方法来做一件事，在进入本书后面更高级的技术之前，我们首先看到的是最简单的途径。

到目前为止，你还没有被Ruby中数据的不同类型搞得疲惫不堪。在第2章提到的对象和类，实际上也是数据的类型，尽管它们看起来好像不是。在第6章我们将直接操作对象和类，操作方法与本章操作数字和字符串类似，但总体思路将会变得清晰。

在第4章我们将开发一个完整而基础的Ruby程序，但在进入到第4章之前，我们来回顾一下到目前为止讨论的内容：

- 变量：我们已经在第2章中谈过，但本章扩展了已有知识。变量是对象的占位符，对象范围包括数字、文本、数组或任何自己创建的对象。
- 运算符：在表达式中用来操作对象，例如+（加）、-（减）、*（乘）、/（除）。也可以用运算符来进行比较，例如<、>和&&。
- 整数：完整的数字，例如5或923737。
- 浮点数：有小数点部分的数字，例如1.0或3.141592。
- 字符：单个字母、数字、空格或印刷符号。
- 字符串：诸如“Hello, World!”或“Ruby is cool.”等的字符集合。
- 常量：有固定值的变量。常量名以大写字母开头。
- 迭代子：诸如`each`、`upto`或`times`等的特别方法。它们在列表中逐个元素地进行步进，这一过程称为迭代，而`each`、`upto`或`times`称为迭代子方法。
- 插写：表达式与字符串的混合。
- 数组：有指定规则顺序的对象或值的集合。
- 散列表：与键关联的对象或值的集合。键可用于在散列表中查找其对应的值，但散列表中的元素没有特定顺序。它是一种查找表，更像书或字典的索引。
- 正则表达式：描述匹配和比较的文本模式。

- 流程控制：根据某个条件和状态，管理要执行哪一段代码的过程。
- 代码块：一段代码，常用于迭代子的参数，没有具体的名字，本身也不是方法，但可被以参数接收它的方法调用和处理。代码块还可被保存在变量中，作为Proc类的对象。
- 范围：从开始值到结束值整个范围的表示形式。
- 符号：Ruby符号是独特的引用。符号和变量不同，它不包含值，但可用于在代码中保持一致的引用。它可被视为无值的常量。

是让我们把这些基础元素放在一起的时候了，在第4章我们将开发一个完整的运行程序。



第4章 开发基础的Ruby应用程序

到现在为止，我们关注的焦点一直是Ruby语言基础知识，并考察这些知识的底层运作方式。本章我们将转移到真正的软件开发世界，开发一个完整的、基础的，具有一组基本功能的Ruby应用程序。一旦完成这一基础程序的开发和测试，我们将考察扩展这一程序的不同方法，让它变得更加有用。在我们的开发旅程中，将讨论本书目前尚未提到的一些新内容。

首先，在进入真正的编程之前，我们先来看一下基本的源代码组织方法。

4.1 处理源代码文件

到目前为止，本书一直用`irb`这个即时提示符程序来学习Ruby语言。但如果想开发重复使用的程序，必须把源代码保存到磁盘文件中（或发到因特网上、刻在光盘中等）。

创建并操作源代码文件的方式，与操作系统和个人喜好有关。在Windows桌面，你可能熟悉用内置的记事本程序创建和编辑文本文件。在Linux提示符界面，你可能用`vi`、`Emacs`或`pico/nano`程序。而Mac用户则使用随手可得的`TextEdit`程序。不管用什么程序，都需要创建新文件，并将其保存为纯文本，以供Ruby正确使用。在后续章节中，你将了解每个平台上有哪些特定的工具可用来进行Ruby程序开发。

4.1.1 创建测试文件

Ruby程序开发的第一步是熟悉文本编辑器。下面为每个主要平台分别提供一些指导说明。

如果你已经熟悉文本编辑器，并了解怎样用它来编写和保存源代码，请跳过本节，直接阅读4.1.2节。

Windows平台

如果遵循第1章的说明步骤下载并安装Ruby，就已经有两个文本编辑器，它们分别叫做`SciTE`和`FreeRIDE`，可在“开始”菜单的Ruby程序组菜单中找到。`SciTE`是个通用源代码编辑工具，而`FreeRIDE`是Ruby专用的源代码编辑器，是用Ruby编写的。`SciTE`更快一点，但`FreeRIDE`对一般开发任务来说不仅很快，而且还与Ruby有更好的集成。

一旦启动编辑器，就会看到一个空白文档，可在此开始输入Ruby源代码（在`FreeRIDE`中你需要用“File”菜单来创建新文档）。通过“文件”菜单，还可以把源代码保存到硬盘，在下节我们将做这一操作。如果用`FreeRIDE`，还可以把多个文件组成成单个项目（project）。

Mac OS X平台

Mac OS X有几个文本编辑器。`MacroMates`公司（<http://www.macromates.com/>）开发的`TextMate`如图4-1所示，它可能是Ruby社区最推崇的文本编辑器，但不是免费的，价格大约50美元。OSX开发工具中包含的`XCode`，也是值得一提的选择，但你需要了解怎样安装和使用这套开

发工具（在OS X安装光盘中可以找到）。根据Mac电脑的硬件配置情况，XCode有时会相当慢。

当然，OS X中也内置了免费的编辑器，是TextEdit。在Applications文件夹中双击TextEdit图标即可启动它。其初始模式不是纯文本编辑器，但在“Format”菜单选择“Make Plain Text”，即可进入纯文本编辑模式，可用来编辑Ruby源代码。

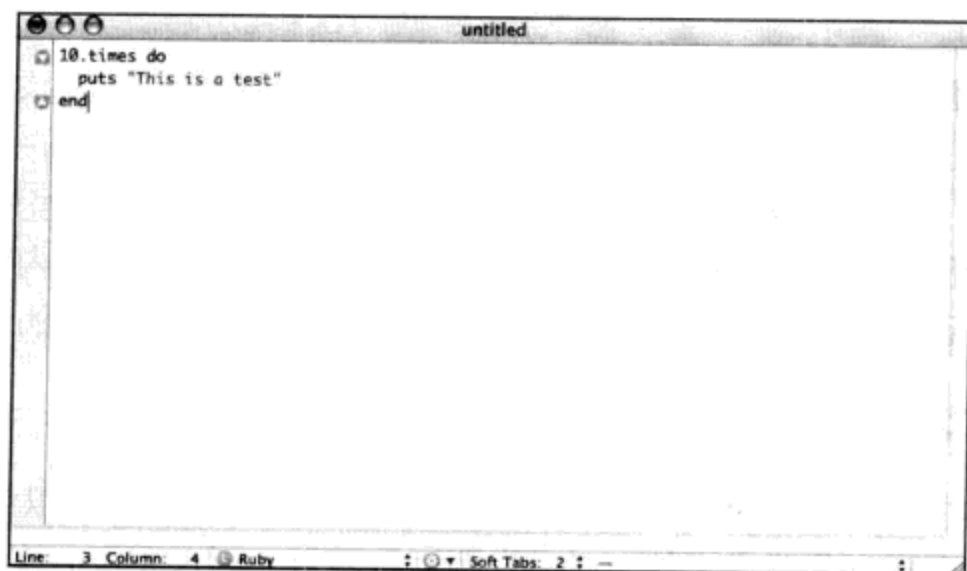


图4-1 使用TextMate

此时只需输入或粘贴Ruby代码，并用“File”→“Save”菜单选项，即可把文本保存到磁盘中。最好在home目录（即左边有你用户名的目录）创建名为ruby的文件夹，以便在此保存初始Ruby源代码文件，下节说明步骤中已经假定你创建了此文件夹。

Linux平台

Linux发行版本包含的文本编辑器常常各不相同，但至少有一个是固定的。如果你完全用shell或终端来操作，你可能会熟悉vi、Emacs、pico或nano，所有这些都用来编辑Ruby源代码。如果你用的是Linux图形界面，可能会有固定的Kate（KDE高级文本编辑器）和/或gedit（GNOME编辑器）。所有这些都是很棒的文本和源代码编辑器。

你也可以下载并安装FreeRIDE，这是个跨平台的源代码编辑器，专门为Ruby开发人员设计。它提供了从编辑器中单击即可运行代码的能力（如果你用的是X图形用户界面的话），并以不同颜色显示各种语法性质的代码，以便更容易阅读。如要了解更多FreeRIDE的信息，请访问<http://freeride.rubyforge.org/>网站。

在此阶段最好在home目录中创建一个名为ruby的文件夹，以便有个容易记忆的位置保存Ruby源代码文件。

4.1.2 测试用源代码文件

开发环境建立好之后，就可以编辑和保存文本文件了，请输入下面的代码：

```
x = 2
print "This application is running okay if 2 + 2 = #{x + x}"
```

注 如果你看不懂这段代码，那么可能是你跳过的章节太多了。所以请回头看第3章！

本章需要你完全掌握第3章的所有知识。

用`a.rb`文件名保存这段代码，放在所选择的目录。建议你在容易找到的地方，创建一个名为`ruby`的目录。在Windows平台上可以直接放在C盘根目录中，在OS X或Linux平台上可以放在`home`目录中。

注 `RB`是Ruby文件扩展名的事实标准，就像`PHP`是标准的`PHP`文件、`TXT`是普通的文本文件、`JPG`是`JPEG`图像文件的标准扩展名一样。

现在可以开始运行代码了。

4.1.3 运行源代码

创建了基本的Ruby源代码文件`a.rb`之后，就该让Ruby执行它。和以往一样，执行的过程也因操作系统而异。请阅读下面与你的操作系统相对应的特定小节。如果你的操作系统未在此列出，OS X和Linux的说明步骤应该最接近于你的操作系统平台。

无论何时，当本书请你“运行”程序时，下面都是你每次要做的步骤。

注 尽管本章开始让你开发应用程序，但还是有机会在本章使用`irb`，来试验测试代码，或探索基础理论的工作机制。请你自行判断，并在两种开发方法之间切换。`irb`对于验证小概念和简短代码块极其有用，它让你无须在文本编辑器和Ruby解释器之间麻烦地跳转。

Windows平台

如果你用的是Windows平台中随Ruby安装程序自带的SciTE或FreeRIDE程序，可以直接用它们来运行Ruby程序（参见图4-2）。在这两个程序中，都可以按F5键来运行Ruby代码。另一种方法是用菜单（SciTE是“Tools”→“Go”，而FreeRIDE是“Run”→“Run”）。不过，这么做之前，请务必确保已经保存了Ruby代码。如果没有，结果可能会不可预料（例如，可能会运行上次保存版本所生成的旧代码），或是系统提示你保存文件。

如果在输出面板中（位于SciTE的右边，FreeRIDE的底部），`a.rb`代码的运行结果给出令人满意的输出，你就可以进到下一节“我们的目标程序：文本分析器”。

另外还有一种方法，可以从命令提示行运行Ruby程序。首先启动命令提示行（“开始”菜单→“运行”→输入`cmd`并点击“确定”按钮），然后用`cd`命令进入`a.rb`文件所在的目录，输入`ruby a.rb`命令并按回车键。

不过，如果你不了解怎样从命令行进入硬盘相应目录，则不建议使用这种方法。还有另一种选择，如果你能熟练创建快捷方式，可以在Windows桌面创建Ruby可执行文件（`ruby.exe`）的快捷方式，然后把源代码文件拖放到这个快捷方式图标上即可。

Mac OS X平台

在Mac OS X平台上，运行Ruby程序的最简单方法是从Terminal终端程序运行，和`irb`的运行方式一样。第1章已介绍过Terminal程序，如果你按照前文的说明步骤操作成功，请继续按下面

的步骤操作：

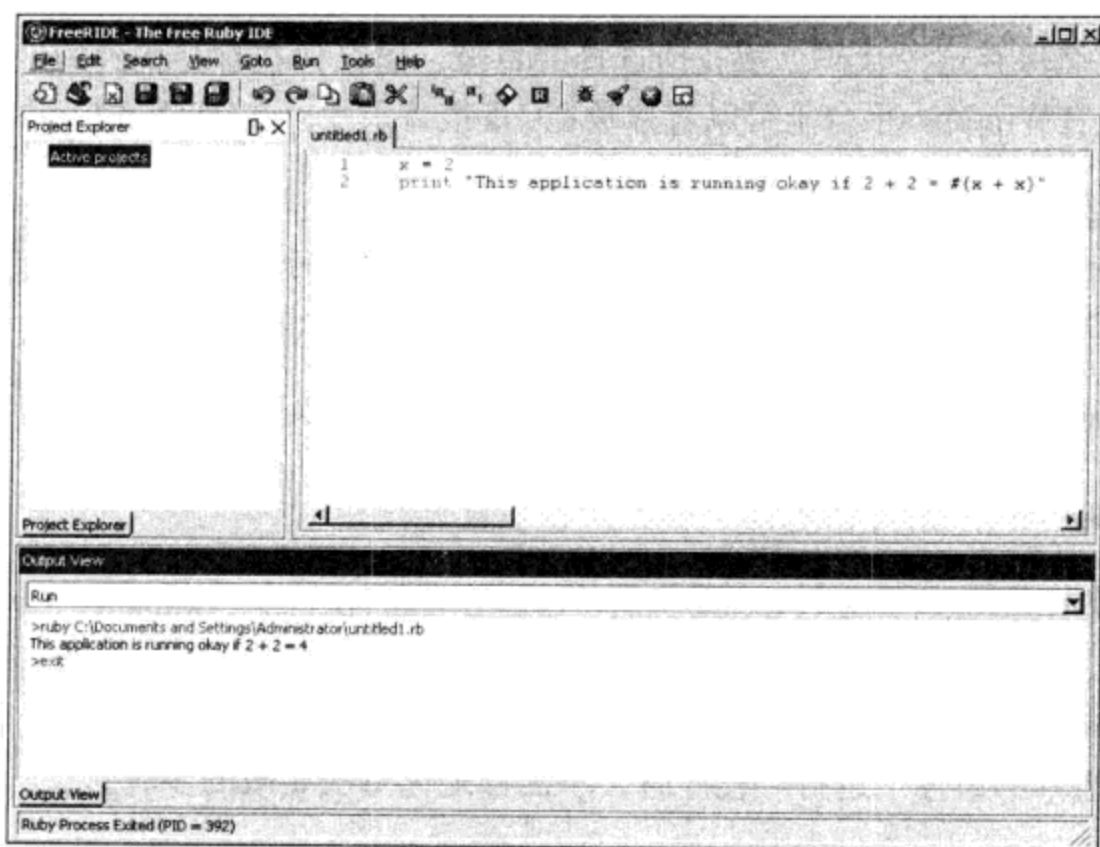


图4-2 在微软Windows平台的FreeRIDE中运行代码（请注意底部的输出面板）

1. 启动Terminal程序（在Applications/Utilities目录中）。
 2. 用cd命令进入存放a.rb文件的文件夹，例如：`cd ~/ruby`。这个命令告诉Terminal程序，进入home目录的ruby文件夹。
 3. 输入`ruby a.rb`并按回车键，即可执行a.rb这个Ruby脚本文件。
 4. 如果你看到错误信息，例如`ruby: No such file or directory -- a.rb (LoadError)`，则表示没有位于a.rb源文件你要找的文件夹，需要确定它被保存到哪里去了。
- 如果从a.rb得到令人满意的响应输出，则可以进入到下一节“我们的目标程序：文本分析器”。

Linux和其他基于UNIX的系统

在Linux和其他基于UNIX的系统中，是从shell程序运行Ruby程序的（也就是说，在终端窗口中），和你运行irb的方法一样。第1章已经介绍过运行irb的过程，因此如果你忘记了怎么做，就需要在继续下去之前回顾一下，如下所示：

1. 启动终端模拟器（例如xterm），以便得到Linux shell或命令提示符。
 2. 用cd命令进入a.rb所在目录（例如，用`cd ~/ruby`命令进入home目录下的ruby目录，通常是/home/yourusernamehere/）。
 3. 输入`ruby a.rb`命令并按回车键，让Ruby执行a.rb脚本。
- 如果从a.rb得到令人满意的输出结果，就可以继续阅读下面的章节。

文本编辑器 VS. 源代码编辑器

前文提到的源代码基本上就是纯文本，尽管可以在普通文本编辑器中编写代码，但如果用专用的源代码编辑器（或专用于开发的IDE——IDE是Integrated Development Environment的缩写，表示集成开发环境），则可以获得一些好处。

FreeRIDE就是Ruby开发人员专用编辑器的一个例子，它和其他文本编辑器一样，可以编辑文本，但提供了一些扩展功能，例如源代码高亮显示，并可以从编辑器中直接运行代码。

有些开发人员发现，源代码语法高亮显示的价值无法估量，它使代码更容易阅读。变量名、表达式、字面字符串和源代码中的其他元素均以不同颜色显示，可以一目了然。

选择源代码编辑器还是基本的文本编辑器，要根据你自己的喜好而定，但两种都值得一试。许多程序员喜欢常规文本编辑器带来的自由度，并在完成代码之后，再从命令行运行Ruby程序，而其他人则喜欢在单一环境中完成全部工作。

FreeRIDE可从<http://freeride.rubyforge.org/>网站获得，而另一个竞争对手是Ruby和Rails源代码编辑器，名叫RadRails，可从<http://www.radrails.org/>网站获得。在你的操作系统平台上，其他这些编辑器当然值得一试，因为它们可能提供更符合你期望的工作方式。

4.2 我们的目标程序：文本分析器

本章你要开发的程序是个文本分析器。该程序读入单独文件提供的文本内容，对其进行各种模式的分析和统计，并向用户打印输出分析结果。这不是个三维图形冒险程序，也不是个精美的网站，但文本处理程序是系统管理维护和大多数程序开发的基础，它是日志文件分析、网站用户提交文本的分析和其他文本数据分析的生命线。

Ruby非常适合做文本和文档分析，它的正则表达式功能，以及简单易用的scan和split函数，将在你的程序中得到大量使用。

注 对于本程序来说，重点是快速实现其功能，而不是开发一套详细的面向对象结构、相关文档或测试方案。第6章将深入讨论面向对象及其在大型程序中的使用，第8章将讨论文档生成和测试。

4.2.1 基本功能需求

文本分析器将提供以下基本统计数据：

- 字符数统计。
- 字符数统计（不含空格）。
- 行数统计。
- 字数（单词数）统计。
- 文句数统计。
- 段落数统计。

- 平均每句包含几个单词。
- 平均每段包含几个文句。

对于最后两个功能要求，统计结果可从其他统计数据中方便地计算出来。也就是说，有了总字数（单词数）和总语句数，要得到平均每句含几个单词，只是个简单的除法工作。

4.2.2 构建程序基本框架

在开始开发新程序时，想好需要哪些关键步骤是很有用的。过去常常用画流程图（flow chart）的方式，来展示计算机程序的操作是怎样流转的，但有了现代工具，例如Ruby，我们就可以很容易试验、修改，并保持敏捷的开发效率。我们先来勾勒出这个程序基本步骤，如下所示：

1. 载入包含待分析文本或文档的文件。
2. 在逐行载入文件时，统计有多少行（这是你关心的统计结果之一）。
3. 把文本放到字符串中，测量其长度以得到字符数统计。
4. 临时删除所有空白，并测量得到结果字符串的长度，得到不含空格的字符数。
5. 切分所有空白，得出有多少字（单词）。
6. 按句号切分，得出有多少文句。
7. 按双换行符切分，得出有多少段落。
8. 进行计算，得出平均数。

创建新的空白Ruby源文件，并保存为`analyzer.rb`，放在Ruby文件夹中。在下面几节中，我们将用代码填充这个文件。

4.2.3 获取哑文本

在开始编码之前，第一步是获取某些测试数据，供分析器进行分析。《雾都孤儿》（*Oliver Twist*）第1章是理想的文本，因为它开放了版权，而且容易获取，还有相当的长度。你可以从<http://www.rubyinside.com/book/oliver.txt>网页或http://www.dickens-literature.com/Oliver_Twist/0.html网页获得这段文本，把它拷贝到本地文本文件中，将该文件保存到与`a.rb`相同的目录，命名为`text.txt`。程序将默认从`text.txt`读取文本内容（当然，后面将把它改为更加动态化，并可以接受其他数据来源）。

提示 在你阅读本书时，如果上述网页无法访问，可以用你喜欢的搜索引擎搜索“twist workhouse rendered profound thingummy”文字，绝对能找到。还有另一种方法，是采用你能找到的任何大段文本。

如果你用《雾都孤儿》的文本，并希望你的结果与本章示例大致相同，请确保只拷贝粘贴以下两段文字之间的内容，从：

```
Among other public buildings in a certain town, which for many  
reasons it will be prudent to refrain from mentioning
```

到：

```
Oliver cried lustily. If he could have known that he was an
orphan, left to the tender mercies of church-wardens and
overseers, perhaps he would have cried the louder.
```

4.2.4 载入文本文件并统计行数

现在可以开始编码了！第一步是载入文件。Ruby通过File类提供了一套全面的文件操作方法。其他语言需要你先做几步准备工作，而Ruby则让接口相当简单。下面是打开text.txt文件的代码：

```
File.open("text.txt").each { |line| puts line }
```

在analyzer.rb文件中输入这段代码，并运行。如果text.txt文件在当前目录中，则将看到整个文件内容在屏幕上飞驰而过。

在这段代码中，要求File类打开text.txt文件，然后，与数组相似，你可以对该文件直接调用each方法，导致每行内容逐次传递给内部代码块，由puts将该行内容输出到屏幕（在第9章，你将看到文件访问和操作的内部细节，以及比本章所用方法更好的技术方法）。

编辑这段代码，改成如下形式：

```
line_count = 0
File.open("text.txt").each { |line| line_count += 1 }
puts line_count
```

在这段代码中，先初始化line_count变量，来保存行数统计值，然后打开文件，并迭代读取每一行内容，每次对line_count加1。完成之后，再把总计值打印输出到屏幕（如果你用的是《雾都孤儿》章节的话，大约是121）。现在，你得到了第一个统计值！

现在我们已经统计了行数，但还没有检查文件内容，以便统计字数（单词数）、段落数、句子数等统计值。这很容易修改，我们来稍微修改一下代码，加入text变量，收集各行内容：

```
text=''
line_count = 0
File.open("text.txt").each do |line|
  line_count += 1
  text << line
end

puts "#{line_count} lines"
```

注 请记住，用{和}来包含代码块，是单行代码块的标准风格，而do和end则更常用于多行代码块。不过这只是惯例，而非必须的要求。

与前面的代码相比，这段代码引入了text变量，并把每行内容逐次加到该变量中。当文件的迭代读取完成后——也就是说，读完所有行之后，text就把整个文件的内容包含在单个字符串中，供你后续使用。

这种把文件内容放到单个字符串中并统计行数的方法看起来很简洁，但File类还有其他更

快的读取文件方法。例如，可以把上述代码改写如下：

```
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join
```

```
puts "#{line_count} lines"
```

简洁多了吧！File类实现了readlines方法，可把整个文件逐行读取到数组中，我们既可以用它来统计行数，也可以把它们合并到单个字符串中。

4.2.5 统计字符数

第二个最简单的统计工作是算出文件中的字符数。由于整个文件内容已经被收集到text变量中，而且text还是个字符串，因此你可以用所有字符串都有的length方法，得到文件的确切大小，也就是相应的字符数。

在上述analyzer.rb代码的末尾，加入以下内容：

```
total_characters = text.length
puts "#{total_characters} characters"
```

如果现在用《雾都孤儿》文本运行analyzer.rb，将会得到如下输出：

```
121 lines
6165 characters
```

你想得到关于字符的第二个统计数，是空格除外的总字符数（上例给出的是包含空格的字符数）。如果你还有印象，应该能回想起第3章中，字符串都有用来进行全局替换（类似搜索与替换）的gsub方法。例如：

```
"this is a test".gsub(/t/, 'X')
```

```
Xhis is a XesX
```

同样，你可以用gsub方法来去掉text字符串中的空格，然后对新的“去空格的”text字符串，用length方法得出其长度。请把下面的代码加到analyzer.rb文件中：

```
total_characters_nospaces = text.gsub(/\s+/, '').length
puts "#{total_characters_nospaces} characters excluding spaces"
```

如果现在再对《雾都孤儿》文本运行analyzer.rb，会得到类似下面的结果：

```
121 lines
6165 characters
5055 characters (excluding spaces)
```

4.2.6 统计字数

各种文字处理软件通常都提供“字数统计”的功能，是对整篇文档或选中文本中的完整字

数（单词数）进行统计。这个统计值对于打印时计算整篇文档需要多少页是很有用的，许多赋值也需要字数（单词数）统计值，因此知道某段文本中有多少字数（单词数）是相当有用的。

可以用几种方法得到这一功能：

1. 用scan方法统计连续字符组的个数。
2. 用split方法按空格切分字符串，并用size方法统计切分之后的片断个数。

我们逐个来看这两种方法，看哪一种是最好的。回想一下，在第3章中，scan方法的工作机制是在文本字符串中迭代，并反复找出特定模式。例如：

```
puts "this is a test".scan(/\w/).join
```

```
thisisatest
```

在本例中，scan搜索字符串中匹配\w的内容，这里\w表示字母表中的任何字符（以及下划线），然后将找到的内容连接成一个字符串，并打印输出到屏幕。

你可以用文字和数字字符组实现同样的功能。在第3章中，我们学到用正则表达式来匹配多个字符，即字符后跟+。因此我们再试一次：

```
puts "this is a test".scan(/\w+/).join('-')
```

```
this-is-a-test
```

这次，scan搜索所有文字、数字字符组，并将其放入数组，然后用-作为分隔字符，把它们连接成一个字符串。

要得到字符串中的单词数，可以用length或size这两个数组方法，来统计数组中的元素个数，而非将其连接起来。

```
puts "this is a test".scan(/\w+/).length
```

```
4
```

太妙了！那么split方法又会得到怎样的效果？

split方法展现了Ruby（以及某些其他语言，特别是Perl）的核心原则，即“总有不止一种解决办法！”对同一个问题能找到多种不同的解决办法，是成为优秀程序员的决定因素，因为不同解决方法的功效是各不相同的。

我们把字符串按空白切分，并得出结果数组的长度，如下所示：

```
puts "this is a test".split.length
```

```
4
```

正如我们所知，split方法默认按空白（单个或多个空格、制表符、换行符等）进行切分，这样一来，与scan方法相比，代码就变得更短，而且更容易阅读。

那么，这两种方法的区别何在？很简单，一个是搜索并返回各个字（单词）供你进行统计，另一个是根据各个字（单词）之间的分隔符（即空白）来切分字符串，并得出该字符串被切分成多少个部分。有趣的是，这两种方法可得出不同的结果：


```
text = "First-class decisions require clear-headed thinking."
puts "Scan method: #{text.scan(/\w+/).length}"
puts "Split method: #{text.split.length}"
```

```
Scan method: 7
Split method: 5
```

scan方法是搜索所有文字数字字符块的，因此得出句子中有7个单词的统计结果。而如果按空白切分，则得到5个单词的统计结果。原因在于由连字号相连的单词。连字号不是“文字数字字符”，因此scan方法把“first”和“class”视为两个单独的单词。

回到analyzer.rb文件，我们把学到的方法应用起来，增加下面的代码：

```
word_count = text.split.length
puts "#{word_count} words"
```

运行完整的analyzer.rb，得到如下结果：

```
122 lines
6166 characters
5055 characters (excluding spaces)
1093 words
```

4.2.7 统计句子和段落数

理解了统计字数（单词数）的方法逻辑之后，统计文句数和段落数就变得很简单。我们不是按空白进行切分，因为文句和段落有另外的切分规则。

文句以句号、问号和惊叹号结尾，也可以用省略号和其他标点符号分开，但我们这里无须担心这些罕见情况。切分操作很简单，我们不再要求Ruby按一种类型的字符来切分文本，而是只按三种类型的字符进行切分，如下所示：

```
sentence_count = text.split(/\.|!|\?|!|/).length
```

正则表达式看起来很奇怪，但句号、问号和惊叹号可以看得很清楚。我们直接来看正则表达式：

```
/\.|!|\?|!|/
```

开头和结尾的斜杠是正则表达式的常用分界符，因此可以忽略。第一部分是\`\.`，表示句号。为什么不直接用`.`，而加反斜杠呢？原因在于，在正则表达式中表示“任何字符”（如第3章所述），因此需要用反斜杠进行转义，表示它是字面字符的句号。这也解释了为什么问号也用反斜杠进行转义，因为问号在正则表达式中表示“零个或一个前一字符”（也在第3章中有论述）。而`!`则未转义，因为它在正则表达式中没有其他特别含义。

管道符号（`|`字符）把三个主要字符分开，表示分别对待这三个字符，可匹配其中任一字符。这样一来，split即可同时按句号、问号和惊叹号进行切分。你可以测试一下，代码如下：

```
puts "Test code! It works. Does it? Yes.".split(/\.|!|\?|!|/).length
```

4

段落也可以用正则表达式来切分。由于在印刷书籍中，例如本书，段落之间一般没有空白分隔，而在计算机中输入的段落一般则有分隔，因此你可以用双换行符（用特别组合`\n\n`表示，它只是表示连续两个换行符），来得到分隔的段落。例如：

```
text = %q{
This is a test of
paragraph one.

This is a test of
paragraph two.

This is a test of
paragraph three.
}

puts text.split(/\n\n/).length
```

3

我们把这些概念都加到`analyzer.rb`中：

```
paragraph_count = text.split(/\n\n/).length
puts "#{paragraph_count} paragraphs"

sentence_count = text.split(/\.|\/|!|?|!|/).length
puts "#{sentence_count} sentences"
```

4.2.8 计算平均值

这个简单应用程序最后要统计的，是每句的平均单词数，以及每段的平均文句数。我们已经有了段落、文句、单词的统计数，即`word_count`、`paragraph_count`和`sentence_count`变量，因此只需做简单的算术计算，如下所示：

```
puts "#{sentence_count / paragraph_count} sentences per paragraph (average)"
puts "#{word_count / sentence_count} words per sentence (average)"
```

由于这些计算太过简单，因此可以直接放到输出命令中，无须预先进行计算。

4.2.9 到目前为止的源代码

随着本书讨论的进行，你也在不断更新源代码，每次都是把增加的逻辑代码放到显示结果的`puts`语句之后。但为了形成最终版本，最好把逻辑处理与数据展现的代码稍稍分开，并把计算放到单独的代码块中，置于屏幕输出的所有代码之前，这样安排更加紧凑。

这里不对逻辑处理代码作任何修改，但`analyzer.rb`最终源代码看起来更清爽一些，如下所示：

```

lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join
word_count = text.split.length
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length
paragraph_count = text.split(/\n\n/).length
sentence_count = text.split(/\.|!|?|!|/).length

puts "#{line_count}lines"
puts "#{character_count}characters"
puts "#{character_count_nospaces}characters excluding spaces"
puts "#{word_count}words"
puts "#{paragraph_count}paragraphs"
puts "#{sentence_count}sentences"
puts "#{sentence_count / paragraph_count}sentences per paragraph (average)"
puts "#{word_count / sentence_count}words per sentence (average)"

```

如果你已经编写了这么多代码，而且一切无误，那么应该得到祝贺。我们来看一下，怎样把我们的程序再扩展一点点，增加一些更有意思的统计。

4.3 增加额外功能

分析器程序有几个基本功能，但没有太大意思。虽然行、段落和字数（单词数）的统计很有用，但掌握了Ruby的威力，还可以从文本中分析出更多相当有意思的数据，唯一的限制是我们的想像力。本节将带你见识几个其他可以实现的功能，及其具体实现方式。

注 在开发软件时，永远值得思考的是软件未来扩展和修改的可能性，并提前为这些可能性作准备。许多开发瓶颈正是由于设计太死板，以致无法应对变化的情况！

4.3.1 “有用”字词的百分比

许多书面材料，包括本书，都包含大量特别的字词，这些字词尽管有其上下文环境和结构，但没有直接用处，或无法引起直接注意。在刚才这句话中，“其”、“和”、“是”、“或”不特别引人注意，但假如没有这些字词，这句话将不成句子。

这些字词一般称为“虚词”（stop words），对于专门进行文本分析和搜索的计算机系统来说，这些字词可忽略不计，因为大多数人不会用这些字词来作搜索关键字（例如，常用名词来搜索）。Google就是个完美的例证，它不想存储一般与搜索无关的信息来占用空间。

注 关于虚词（stop words）的更多信息，包括相关链接的完整列表，请访问如下网址：
http://en.wikipedia.org/wiki/Stop_words。

我们可以假设，更“有趣”的文本，或由更熟练作者所写的文本，其虚词所占百分比应该更低，有用或有趣的字词所占百分比则更高。你可以很容易扩展你的应用程序，算出目标文本

中，非虚词所占的百分比。

第一步是构建虚词列表。虚词可能有上百个，但你只用其中最常见。我们来创建数组，以便保存这些虚词：

```
stop_words = %w{the a by on for of are with just but and to the my I has some in}
```

这行代码产生一个虚词数组，赋值给stop_words变量。

提示 在第3章中，我们见过数组可以这样定义：`x=['a','b','c']`。和许多语言类似，Ruby也有快捷方式，即用字符串分隔的文本快速创建数组。这段代码可被缩减为等价形式`x=%w{a b c}`，如上面的虚词代码所示。

为了演示的目的，我们编写一个短小的、单独的程序，来验证这一概念：

```
text =%q{Los Angeles has some of the nicest weather in the country.}
stop_words =%w{the a by on for of are with just but and to the my I has some}

words =text.scan(/\w+/)
key_words =words.select {|word|!stop_words.include?(word)}

puts key_words.join('')
```

当运行这段代码，会得到如下结果：

```
Los Angeles nicest weather country
```

太酷了，是不是？在这段代码中，第一步先把一段文本放到程序中，然后是虚词列表。第二步从text数组中得到所有字词，放到名为words的数组中。然后进入魔术环节：

```
key_words = words.select { |word| !stop_words.include?(word) }
```

这行代码首先取得字词数组words，并以代码块调用select方法，这个代码块的功能是处理每个字词（和第3章中你用过的迭代子相似）。select方法是所有数组和散列表都有的方法，它返回该数组或散列表中，匹配代码块中表达式的元素。

在本例中，代码块中的代码通过word变量得到每个字词，并询问stop_words数组中是否包含与word变量相同的元素，这是通过stop_words.include?(word)实现的。

表达式之前的惊叹号(!)，把表达式的值取反（惊叹号把任何Ruby表达式取反）。这么做的原因是你不想选择属于stop_words数组中的字词，你想选择的是不属于其中的字词。

为了结束这段代码，再用select方法选择words数组中，不包含在stop_words数组的所有元素，并将选出的元素赋值给key_words数组。在你理解这句话之前，请不要继续读下去，因为这种单行代码构造形式在Ruby编程中很常见。

最后，用基本的算术方法，计算出非虚词占所有字词的百分比：

```
((key_words.length.to_f / words.length.to_f) * 100).to_i
```

使用.to_f方法的原因，是把数组长度看成浮点数，从而更精确地算出百分比。当得出真正的百分比（超出100），可以再把它转换成整数。

在本章末尾的最终版本中，你会看到这些处理方法是怎样结合在一起的。

4.3.2 找出“有趣的”句子进行汇总

诸如微软Word等文字处理软件一般都有汇总功能，可以对一长段文本进行处理，选出最佳文句，以便生成“一览表”汇总信息。随着这些年的发展，生成汇总的内部机制变得越来越复杂，但要开发自己的汇总功能，最简单的方法之一是用特定条件对文句进行扫描。

扫描方法之一是查找具备平均长度且包含名词的文句。短句不太可能包含有用的内容，长句又太长不便汇总。能够可靠地找到名词的系统远远超出了本书的范围，因此你可以“取巧”，只查找能够指示该句中有关名词出现的字词，例如“is”和“are”（例如：“名词is”、“名词are”、“There are x名词”）。

我们假定你想弃用2/3的文句，其中，去掉太短1/3的文句和太长1/3的文句，即只留下原来文句中理想的1/3，对你的任务来说规模正好合适。

为了便于开发，我们从零开始创建一个新程序，随后把代码逻辑转移到主程序中。请创建名为summarize.rb的新程序，并输入以下代码：

```
text =%q{
Ruby is a great programming language.It is object oriented
and has many groovy features.Some people don't like it,but that's
not our problem!It's easy to learn.It's great.To learn more about Ruby,
visit the official Ruby Web site today.
}

sentences =text.gsub(/\s+/, ' ').strip.split(/\.|\\?|\\!/ )
sentences_sorted =sentences.sort_by {|sentence|sentence.length }
one_third =sentences_sorted.length /3
ideal_sentences =sentences_sorted.slice(one_third,one_third +1)
ideal_sentences =ideal_sentences.select {|sentence|sentence =~/is|are/}
puts ideal_sentences.join(".")
```

为了得到良好的统计，我们运行这个程序，看看会有什么结果：

```
Ruby is a great programming language. It is object oriented and has many groovy
features
```

似乎很成功！我们仔细来看这段程序。

首先，定义变量text，用以保存有多个文句的长字符串，这与analyzer.rb很相似。然后把text变量切分到文句数组中，如下所示：

```
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\\?|\\!/ )
```

这里与analyzer.rb所用的方法略有不同，在调用链中有个额外的gsub方法，以及strip方法。gsub方法是去掉所有大块空白，代之以单个空白（\s+表示“一个或多个空白字符”），这是为了美化的目的。strip方法则是去掉字符串开头和结尾的所有额外空白。然后与分析器程序一样，用split方法进行同样的处理。

下一步根据文句的长度，对文句数组排序，因为想忽略最短的1/3和最长的1/3：

```
sentences_sorted = sentences.sort_by {|sentence| sentence.length }
```


数组和散列表都有`sort_by`方法，可用于你希望的任意顺序重排其中的元素。`sort_by`方法接受代码块参数，该代码块是个表达式，定义了排序的依据。在本例中，是对`sentences`数组进行排序。把每个文句放到`sentence`变量中，并调用其`length`方法，以便根据其长度排序。这一行代码执行后，`sentences_sorted`变量即包含按长度排序的文句。

下一步需要从`sentences_sorted`变量的按长度排序文句中，得到中间的1/3，因为这1/3是你认为可能最有趣的内容。为了实现这一目的，可把数组的长度除以3，得到1/3的元素个数，然后从数组的1/3处开始抓取（请注意，要额外抓取一个元素，以便弥补整数除法导致的舍入）。代码如下所示：

```
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
```

第一行代码取得数组的长度，并除以3，得到“1/3数组”的数量。第二行代码用`slice`方法把数组的一段内容“切出来”，并赋值给`ideal_sentences`变量。在本例中，假定`sentences_sorted`有6个元素。6除以3得2，因此1/3数组是两个元素长度。然后用`slice`方法从元素2开始，切出共2个（加1个）元素，从而得到元素2、3、4（请记住，数组元素从0开始）。这样一来，就从你所认为的理想长度文句中，得到“中间的1/3”。

倒数第二行代码则对文句进行检查，看它是否包含“is”或“are”，并只接受满足这一条件的文句：

```
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }
```

这里使用了`select`方法，和上节去除“虚词”的代码一样。代码块中的表达式使用了正则表达式，对`sentence`变量进行匹配，仅当`sentence`中有“is”或“are”时才返回`true`值。这表示`ideal_sentences`数组现在只包含长度适中，是中间的1/3，且包含“is”或“are”的文句。

最后一行代码只是把`ideal_sentences`数组的内容连接起来，形成单个字符串，用句号和空白分隔，以便阅读：

```
puts ideal_sentences.join(". ")
```

4.3.3 分析text.txt之外的其他文件

到目前为止，你的程序是把`text.txt`文件名写死在代码中的，这种方法可以接受，但如果在运行程序时指定要分析处理哪个文件，则会好得多。

注 如果在命令提示行或shell中运行`analyzer.rb`的话，例如，在Mac OS X或Linux操作系统中（如果你用的是Windows，则使用Windows命令提示行），这种方法只是用来演示。如果你在Windows中使用IDE开发环境，那么本节内容只要读读就可以了，不必亲自动手尝试。

一般情况下，如果从命令行启动程序，可以把参数加到命令末尾，程序将会处理这些参数。对于Ruby程序也可以这么做。

在启动Ruby程序时，Ruby把命令行末尾的参数自动放到特定的ARGV数组中。如果想要验证一下，请创建一个新的脚本文件，取名为`argv.rb`，并在其中输入如下代码：

```
puts ARGV.join('-')
```

从命令行运行这个脚本，如下所示：

```
ruby argv.rb
```

结果将会是空白，然后再试试下面的命令：

```
ruby argv.rb test 123
```

```
test-123
```

这一次，程序从ARGV数组中得到参数内容，用连字符连接起来，并显示在屏幕上。可以用这种方法，把`analyzer.rb`文件中的“`text.txt`”替换成`ARGV[0]`或`ARGV.first`（二者的意思完全相同——都是指ARGV数组的第一个元素）。这样一来，读取文件的代码行就变成下面的样子：

```
lines = File.readlines(ARGV[0])
```

现在如果要处理`text.txt`，得用下面这样的命令：

```
ruby analyzer.rb text.txt
```

关于程序的部署方法、怎样让程序对用户更加友好，以及ARGV的使用方法，我们将在第10章学到更多内容。

4.4 完整的程序

我们已经有了这个基础程序的完整源代码，但现在应该从前面几节中，把所有新增的扩展功能放到`analyzer.rb`文件中，生成这个文本分析器的最终版本。

注 请记住，本书的所有源代码均可从<http://www.apress.com>网站的Source Code/Download功能区获得，因此不必严格要求直接输入书中的代码。

所有代码如下所示：

```
#analyzer.rb --Text Analyzer
```

```
stop_words =%w{the a by on for of are with just but and to the my I has some in}
lines =File.readlines("text.txt")
line_count =lines.size
text =lines.join
```

```
#Count the characters
```

```
character_count =text.length
```

```
character_count_nospaces =text.gsub(/\s+/, '').length
```

```
#Count the words,sentences,and paragraphs
```

```
word_count =text.split.length
sentence_count =text.split(/\.|\\?|!\\/).length
paragraph_count =text.split(/\n \n/).length

#Make a list of words in the text that aren't stop words,
#count them,and work out the percentage of non-stop words
#against all words
all_words =text.scan(/\w+/)
good_words =all_words.select{|word|!stop_words.include?(word)}
good_percentage =((good_words.length.to_f /all_words.length.to_f)*100).to_i

#Summarize the text by cherry picking some choice sentences
sentences =text.gsub(/\s+/, '').strip.split(/\.|\\?|!\\/ )
sentences_sorted =sentences.sort_by {|sentence|sentence.length }
one_third =sentences_sorted.length /3
ideal_sentences =sentences_sorted.slice(one_third,one_third +1)
ideal_sentences =ideal_sentences.select {|sentence|sentence =~/is|are/}

#Give the analysis back to the user
puts "#{line_count}lines"
puts "#{character_count}characters"
puts "#{character_count_nospaces}characters (excluding spaces)"
puts "#{word_count}words"
puts "#{sentence_count}sentences"
puts "#{paragraph_count}paragraphs"
puts "#{sentence_count /paragraph_count}sentences per paragraph (average)"
puts "#{word_count /sentence_count}words per sentence (average)"
puts "#{good_percentage}%of words are non-fluff words"
puts "Summary:\n \n"+ideal_sentences.join(".")
puts "--End of analysis"
```

注 如果你是Windows用户，可能想把程序中的ARGV[0]引用替换成显式的"text.txt"引用，以确保在FreeRIDE或SciTE中运行正常。如果从命令提示行运行程序，则会运行正常，无须替换。

现在用《雾都孤儿》的文本来运行完整的analyzer.rb，得到如下所示的输出结果：

```
121 lines
6165 characters
5055 characters (excluding spaces)
1093 words
18 paragraphs
45 sentences
2 sentences per paragraph (average)
24 words per sentence (average)
76%of words are non-fluff words
Summary:
```

'The surgeon leaned over the body, and raised the left hand. Think what it is to be a mother, there's a dear young lamb do.' The old story, 'he said, shaking his head: 'no wedding-ring, I see. What an excellent example of the power of dress, young Oliver Twist was.' Apparently this consolatory perspective of a mother's prospects failed in producing its due effect. 'The surgeon had been sitting with his face turned towards the fire: giving the palms of his hands a warm and a rub alternately. 'You needn't mind sending up to me, if the child cries, nurse,' said the surgeon, putting on his gloves with great deliberation. She had walked some distance, for her shoes were worn to pieces; but where she came from, or where she was going to, nobody knows. 'He put on his hat, and, pausing by the bed-side on his way to the door, added, 'She was a good-looking girl, too; where did she come from

--End of analysis

用其他文本（也许可以用网页的内容）来试试analyzer.rb，看看是否可以改进程序的功能。等学会了后续几章之后，就可以对这个程序进行改进，因此如果你在寻找哪些代码值得改进，请把这件事记在心上。

代码注释

你可能注意到，在源代码中有些文本前面加了#符号的前缀。这些文本是注释（comments），它在程序中的用途一般是帮助原来的开发人员，以及任何可能需要阅读这些源代码的人。注释是特别用来注解为什么要做某些处理动作，以便将来忘记时作出提醒。

你可以把注释行放在Ruby源代码的任何行，甚至放在代码行的末尾。下面是几个Ruby合法的注释例子：

```
puts "2+2 = #{2+2}" # Adds 2+2 to make 4
# A comment on a line by itself
```

只要注释行单独占用一行，或放在某行的末尾，都算是正常合法的注释。充分运用注释，将使代码更容易理解。

4.5 小结

在本章中，我们开发了一个完整的、基础的应用程序，该程序有一组需求和所需功能。然后把这个程序进行了一些非必要、但很有用的扩展完善。用Ruby快速开发程序，这些将变得弹指即成。

通过在本章开发的应用程序，我们可以看出，如果你对手工处理许多文本或手工进行许多计算感到担心不已的话，Ruby可以消除这种紧张。

第4章标志着本书第一部分的实用化编程练习告一段落。第5章将带你了解Ruby的历史、Ruby的开发者社区、Ruby某些功能特点背后的历史原因，并学习怎样从Ruby社区获得帮助，并成为社区的一分子。在通向伟大程序员的道路上，编码只是一半的路程！

第5章 Ruby生态系统

和其他编程语言一样，Ruby有自己的文化和“生态系统”。Ruby的生态系统是由成千上万的开发人员、维护人员、文档人员、博客编写人员以及对Ruby开发和使用提供资助的人们组成。

有些新手程序员错误地认为，了解一门语言的历史和社区是无意义的，而最成功的开发人员则会快速理解生态系统并融入其中。因为语言背后的研发动机以及语言的用户，能提供重要的线索，它让我们得知在解决问题时采取何种最佳方式，而且理解其他开发人员的词汇上，则可以极大地有助于寻求帮助和建议。

本章让你从满是代码的导读中休息一下，带你赶上进度，下面我们将了解Ruby世界的运作情况，Ruby语言背后的动机，以及寻找帮助和融入社区的最佳途径。如果你是软件开发的新手，本章也会解释开发人员常用的与软件开发相关的一些术语和词句。

你还会走马观花地了解Ruby的历史、它的创造者、Ruby开发人员所用的相当独特的过程和术语，以及使Ruby从默默无闻一跃成为重要的一流编程语言所采用的技术。

5.1 Ruby的历史

在编程语言的世界，于1993年初次面世的Ruby是相对年轻的一门语言，与Perl和Python的“年龄”大致相同。而今天仍然使用的最流行编程语言中，例如，Fortran于1953年问世；C语言于1970年代早期问世；而BASIC则于1963年问世。但Ruby的年轻是一份资产，而非缺陷。从设计的第一天起，它就以面向对象为己任，随着时间的变迁，其语法保持了非同寻常的一致性。而其他旧语言为了引入面向对象、网络 and 图形环境等现代概念，被迫急剧增加其语法的复杂性。

许多语言的产生纯粹是出于研究的需要，而Ruby与之不同，它诞生的原因部分是由于对现有语言的厌倦感。尽管已经有如此之多的编程语言面世，一个有勇气的日本计算机科学家因为感觉开发工作变得愈加复杂和无聊，遂决定应该向编程语言的世界注入一些有趣的因素。

5.1.1 Ruby的起源

Ruby的生命起始于日本，它的创造者是松本行弘（Yukihiro Matsumoto），更广为人知的名字是“Matz”。和大多数语言开发者不同，Matz开发Ruby的动机是语言生动有趣并遵循“最小惊讶”原则，以便提高开发人员的总体生产率。由于无法找到符合自己期望的语言，他便运用自己对编程的见解，创造了Ruby（这个名字代表宝石，更是对Perl编程语言的一种合宜的致敬）。

作为长期的面向对象编程发烧友，Matz觉得面向对象是值得吸取的最佳模型，但与诸如Perl等其他语言不同，面向对象不是后来加入的元素，而是整个语言的核心基础。一切都是对象，而方法则充当了过程或函数的角色，以满足来自旧式过程化语言的开发人员的期望。正如Matz在2001年的一次访谈中所说的那样，“我想要一门比Perl更强大、比Python更面向对象的语言，这就是我决定自己设计语言的原因。”

在1995年12月，Matz发布了Ruby的第一版公共 α 版本，并很快在日本形成了一个社区。不过，尽管Ruby很快在日本流行起来，但在世界其他地方立足却花费了很大力气。

注 在软件开发中 α 、 β 和 γ 等名词用来表示软件的开发阶段。没有广泛使用的初始版本常常称为 α 版；实现了大多数所需功能、但可能并不完全测试或稳定的版本常常称为 β 版，尽管这个名词由于过多的Web应用程序而受到污染，其他完全发布的产品或服务也全都用“ β 版”的称呼。

1996年，Ruby的开发工作开放了一些，由核心开发人员组成的小型团队，与其他贡献者一起开始形成更广泛的Ruby开发者社区。Ruby 1.0版于1996年12月25日发布，这些核心开发人员帮助Matz开发Ruby，并向他提交补丁（即对代码的修补）和创意。Matz继续扮演“仁慈的独裁者”角色，尽管受到其他开发人员越来越广泛的影响，仍最终控制着这门语言的发展方向。

注 尽管独自开发软件的情况仍然很常见，但现在许多项目采用公开的方式进行，允许任何有资质的开发人员扩展和开发软件的功能。这样一来，在许多情况下，其他程序员可以产生项目的分支（基于现有代码，划分成他们自己的版本），但在实践中这是很罕见的。

5.1.2 Ruby的影响

Matz在开发Ruby的过程中，深深受到他所熟悉语言的影响。流行的Perl语言的开发者Larry Wall，就是Matz崇拜的英雄，而Perl“总有不只一种解决办法”的原则也在Ruby中得以体现。

有些语言，例如Python，喜欢为开发人员提供更严格的结构和整洁的方法，这样一来，为完成某个特定任务，只有很少的方法可以选择。而Ruby则允许开发人员用许多方法之中任一种来解决问题，这样就提供了巨大的灵活性，再与其面向对象的本质结合起来，让Ruby语言可以最大程度地按用户需要进行定制。

就其面向对象的本质来说，Ruby深受Smalltalk的影响，后者是1970年代开发出来的“桃李满天下”的面向对象语言。和Smalltalk一样，在Ruby中一切都是对象，而且Ruby还允许程序员在程序运行期间修改语言本身的许多详细功能，这一功能称为反射（reflection）。

Ruby也受到Python、LISP、Eiffel、ADA和C++的影响，不过受影响的程度小得多。这些影响体现出Ruby是一门胸怀广阔的语言，不怕从别的语言中吸取最佳思想，这也是Ruby语言如此强大和灵活的原因之一。Ruby语言实现了这些功能特性，也让程序员从其他语言迁移到Ruby变得相当容易。在很大程度上，学习Ruby意味着免费学习其他编程语言中最好的功能特性。（参见附录A中对Ruby和其他语言进行的比较）

5.1.3 向西方流传

作为一门当初只为Matz自己在日本使用的语言，原始文档完全是日文写的，把大多数非日本用户锁在外面。尽管它采用了英语单词作为关键字（例如print、puts、if等），但直到

1997年英文文档才编写出来。

随着1998年下半年ruby-talk邮件列表的建立, Matz才第一次用英语正式推广Ruby语言, 这个邮件列表目前仍然是讨论Ruby语言的最好去处之一, 也是很有用的资源, 在邮件列表网站 <http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml> 上有超过20万条消息。

注 你可以自己订阅ruby-talk, 只须将一封包含“subscribe”并在其后边跟上你姓名的电子邮件, 发到ruby-talk-ctl@ruby-lang.org信箱即可。

紧接着, 官方英文网站ruby-lang.org (<http://www.ruby-lang.org/>) 于1999年下半年很快建立起来, 该网站目前仍然是Ruby的官方英文网站 (参见图5-1和图5-2对那时和当下的官方网站比较)。

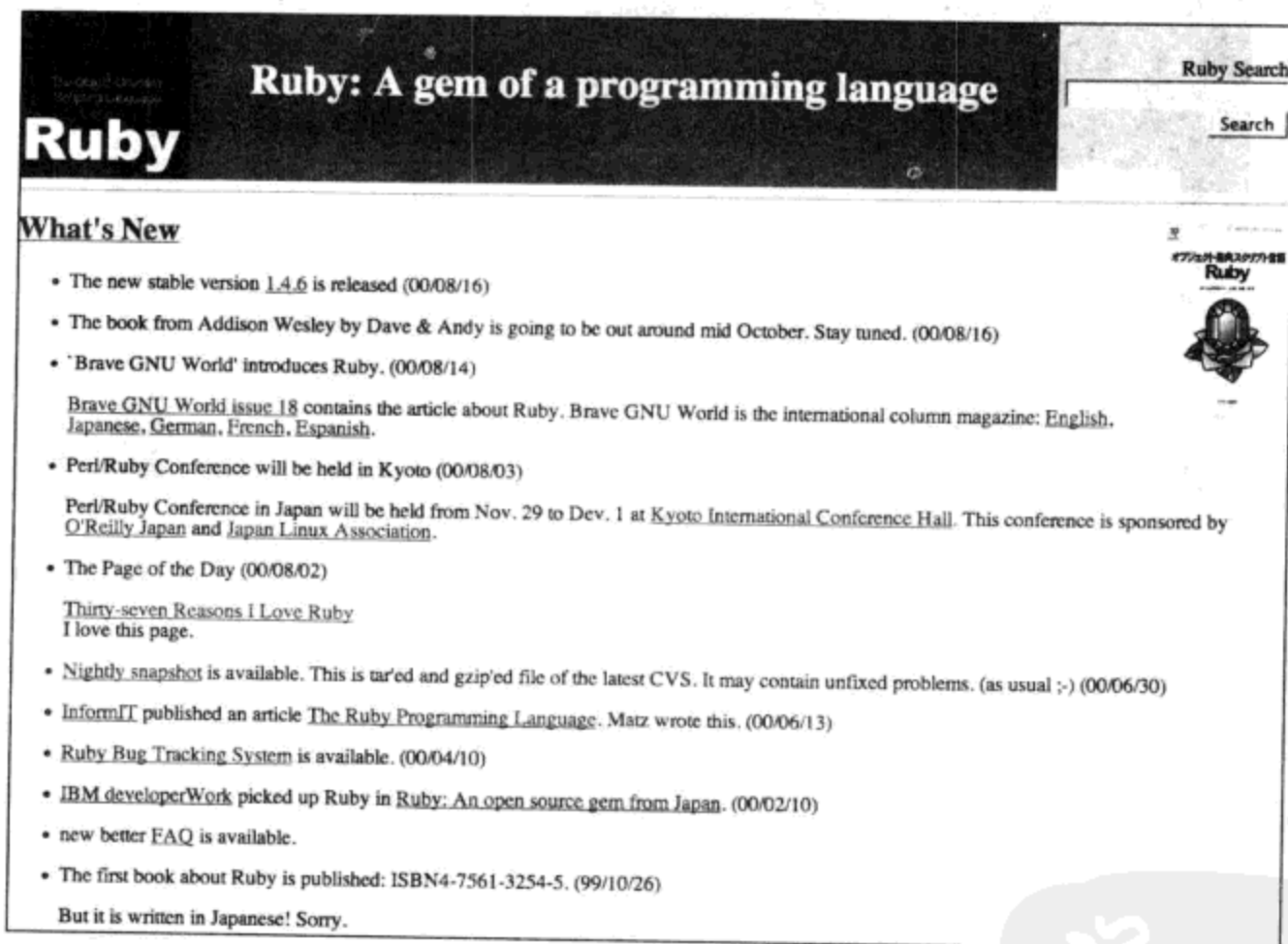


图5-1 2000年的Ruby英文官方主页

除了一些狂热的开发人员, Ruby没有引起太多人的注意, 这种情况一直持续到2000年至2001年 (Ruby主要的Usenet新闻组comp.lang.ruby于2000年5月建立), 甚至在那以后, 讲英语人群的Ruby社区仍然很小。然而Matz没把这种情况看得很严重, 他甚至感到惊讶, 竟然有人认为他的语言很有用, 因为Ruby语言只是为了满足他自己的思考习惯而创造的。

不过, Ruby仍然很少在更大范围软件开发人员面前亮相。IBM于2000年发表了一篇文章, 内容是对Ruby的简要介绍和对Matz的专访。而2001年更令人尊崇的《Dr.Dobb's Journal》杂志发表了Dave Thomas和Andy Hunt的文章, 也作了类似的介绍。

虽然Ruby有着显而易见的强大威力, 但在2004年以前, 它看起来似乎只是像Python和PHP

一样，作为通用脚本和Web语言，在争夺“下一种Perl语言”的桂冠（尽管在2000年Ruby就比Python更流行了）。但当一个丹麦年轻人发表了基于Ruby的工具，迅速改变了全世界开发社区对这门语言的印象时，一切就完全不同了。



图5-2 2006年下半年的Ruby官方主页

5.2 Ruby on Rails

自从2005年开始，要发表任何关于Ruby的书籍或文章，而不提到Ruby on Rails是不可能的。Ruby on Rails是一种Web应用程序框架，它推动Ruby走出日本，从一门只有少数默默无闻但狂热无比的开发人员所用的语言，变成目前成千上万开发人员感兴趣的语言。本节探讨Ruby on Rails，解说它的重要性，并讨论它的面世怎样改变了Ruby生态系统的整体运转状况。

注 应用程序框架是由一组惯例、结构和系统构成的，这些组件提供了底层结构，让应用程序开发变得更容易。Ruby on Rails就是这样一个用于Web应用程序开发的框架。

我们将在第13章讨论用Ruby on Rails进行开发，但首先要看一下这个框架背后的开发动机，以及它怎样改变了Ruby的整体图景。

5.2.1 Rails面世的由来

37signals (<http://www.37signals.com/>) 是个成功的Web软件公司，成立于1999年，开始是个Web设计事务所，在那时很流行的基于Flash的酷炫网站的基础上，向用户推荐采用清爽、快捷、有效的设计方案。整个公司只有两位创始人，他们很快意识到，需要某种工具来帮

助自己有效地处理生意。他们尝试了一些从商店买来的软件，但没有任何一种能满足自己的需要，大多数软件解决方案都臃肿不堪、复杂无比。他们觉得自己对Web设计的观点，也应该适用于应用程序，并于2003年中决定开发他们自己的项目管理工具。

由于37signals的定位是设计公司而非编码公司，因此他们转而找到丹麦哥本哈根的一名大学生，David Heinemeier Hansson，来为他们开发项目管理应用程序。Hansson相信，与其用那时流行的工具（例如Perl或PHP），不如用Ruby来开发这个应用程序，应该更加快捷和完整得多。Hansson以前是个PHP程序员，他已经感觉到用PHP开发大型Web应用程序的痛苦，认为应该寻找新的方向。

随着初期应用程序（名为“Basecamp”）的开发进展，开发团队成员向同行展示了开发成果，并从听到的反馈中很快意识到，他们应该公开发布这个应用程序，而非局限于自己专用。

Basecamp成功地在2004年2月（仅在项目启动后4个月）公开发布证明了37signals和Hansson所采用的开发方法的高效性，37signals开始迅速转变为应用程序开发公司，Hansson最终也成为这家公司的合伙人。

事实证明，Ruby是Basecamp得以快速开发的“银弹”（译者注：软件工程大师Frederick P. Brooks在其名著《人月神话》中，把大规模软件开发的难度比作“人狼”，把有效软件工程技术方法比作能真正杀死人狼的“银弹”，并预言几十年内不会有真正的“银弹”出现。“银弹”遂成为革命性软件工程技术方法的代名词），Hansson使用Ruby的面向对象和反射等功能特性，构筑了一个框架，使数据库驱动的Web应用程序开发比以前更加简单。这一框架就是为人所知的Ruby on Rails，于2004年7月首次公开发布。运用这个新框架的强大威力，37signals继续快速开发新产品。

和Ruby本身一样，Ruby on Rails框架没有立即爆炸性地流行开来，但有少数狂热的粉丝，他们认识到Ruby on Rails的强大威力，在很多情况下，都希望复制37signals的成功。

5.2.2 Web (2.0) 是怎样赢的

Ruby on Rails没有埋没多久。2005年是Ruby on Rails的史诗年份，Ruby也随之爆炸性地流行开来。Ruby on Rails的早期粉丝在博客中大谈特谈相关技术，以草根式营销宣传，无心插柳地为Ruby on Rails赢得了众多信徒。

2005年1月，在那时全世界最流行技术社区网站Slashdot上，发表了第一篇提到Ruby on Rails的帖子，自那时开始成篇累牍地讨论这一技术，鼓励现有PHP、Perl和Python开发人员都去试试Ruby on Rails。

2005年3月，Hansson宣布将写作第一本关于Rails的商业书，于当年5月以β版PDF格式发布。2005年9月，该书的印刷版上市，并立即登上亚马逊网站编程书籍排行榜首位。

在一年时间里，许多出版商启动了Rails书籍编写计划，并于当年发行；成千上万的博客帖子在讨论Rails技术；相关视频（可观看的屏幕操作录像，演示怎样使用Rails）被在线观看了数十万次；David Heinemeier Hansson赢得了数个奖项，包括Google和O'Reilly的“2005年度最佳黑客”。成千上万的开发人员突然间涌向Ruby on Rails，从而，也涌向了Ruby。

Ruby生态系统被迅速推到前台，暴露在聚光灯前，特别是打上了“Web 2.0”概念的烙印，

这个概念是个杜撰词，是指某种假定的第二代基于因特网的服务，常常用于指称博客文化、社交网络、网络百科（Wiki），以及其他根据用户内容驱动的网站。由于Ruby on Rails让这些网站易于开发，因此许多开发人员运用这些工具的优势，占领了Web 2.0领域的前沿。

5.3 开源文化

当Ruby刚开始开发时，Matz的心中并没有特定的开发文化。他开发这门语言，是为了自己使用，适合自己的思考习惯。在开头几年里，他敝帚自珍地把这门语言放在自己手里。当今关于怎样用Ruby开发软件的大多数文化，只是在最后几年里才逐渐出现，其部分思想也在其他编程语言中得到分享。

在Ruby开发文化中，常见的、必须理解的一大元素是开放源码（简称“开源”）的运动。

提示 如果你已经熟知开源的概念，完全可以跳过本节，直接阅读“从哪儿、怎样获取帮助”章节。

如果你曾用过Linux，或下载过某种类型的软件，就可能会熟悉“开源”一词。简单地说，开源表示对于某个应用程序或程序库的源代码，其他人可以公开查阅或使用。对于什么人可以对代码做什么，可能有一些限制（一般是通过许可证），但可以公开查阅。与Linux非常相似，Ruby及其几乎所有程序库，都是遵循开源许可证的规定而发布的——与其他软件，例如微软的Windows不同，它的源代码是不公开的。

Ruby许可证的条款规定，你用Ruby开发的应用程序，不必非得是开源的。你可以用Ruby开发“闭源”的专有应用程序，任何人都看不到代码。是否选择开源式发布，是一个艰难的抉择（你可在附录B中阅读Ruby许可证的全部文本）。

与闭源的决定相比，在开源中也常常有一些灰色阴影区域。37signals在开发其第一个基于Ruby on Rails的应用程序Basecamp时，并未公开发布源码，但却把Ruby on Rails框架从应用程序中剥离出来，并发布。结果是37signals公司得到了大量的公众关注，并招募了一些优秀的程序员，他们免费参与Ruby on Rails的开发，让所有人都受益。类似的软件产品还有流行的Apache Web服务器和MySQL数据库系统，都根据各有千秋的开源许可证开放源码，并常常得到义务程序员的改进。

开源社区是免费分享知识的地方，也是通力合作的地方，对我们大多数人都使用的系统和服务进行改进。尽管专有软件永远有一席之地，但在开发编程语言、程序库和其他通用类型软件时，开源方式迅速成为事实标准。

对开源的理解，是理解Ruby社区的重要线索。尽管许多开发人员不必开放他们的应用程序源代码，但可以经常向社区发布工具和代码技巧的源码，从而获取同行评审意见，并得以广泛流行。

把代码发布为开源，不一定是错误的商业决定。实际上，它可以改进代码和质量，并让你更了解这个行业。

5.4 如何获得帮助

本书将帮你学会所有基础的关于Ruby本身和相关的知识，但在编写代码时，如果能得到更新的或特定领域的帮助，常常更加有用。本节将考察几种从广大Ruby开发者社区中获取帮助的方法（附录C还有更简洁、更完整的资源列表，你将来可能用得上）。

5.4.1 邮件列表

关于编程语言的讨论，邮件列表永远是最流行的港湾，更偏爱编程语言文化的技术层面的人们，最爱在这里流连，这里可以询问关于语言内核和未来发展方向的问题，也可以询问只有真正的语言偏执狂才能回答的秘义问题。然而，这里不适合询问基础知识问题。

Ruby为讲英语者准备了如下三个官方邮件列表可供订阅：

- `ruby-talk`：讨论关于Ruby的一般主题和问题解答。
- `ruby-core`：讨论Ruby核心内容，特别是关于语言开发方面。
- `ruby-doc`：讨论关于Ruby的文档标准和工具（很少用到）。

关于这三个邮件列表的更多信息，请访问<http://www.ruby-lang.org/en/20020104.html>网页，另外`ruby-talk`邮件列表有个Web论坛风格的视图，请访问<http://www.nabble.com/ruby-talk-f13890.html>网页。

另外还有日语、法语和葡萄牙语的邮件列表，这些邮件列表与上段第一页内容很相似。讲英语的读者常常用翻译软件来阅读日语邮件列表，它是由一些最有经验的Ruby开发人员编纂的。关于这些内容的信息，可以在前面提到的网页中找到。

5.4.2 Usenet新闻组

直到2002年，新闻组系统（Usenet）还是大量有共同兴趣的人们分享和讨论知识的常用方式。

Google Groups的出现，让新闻组的访问变得简单，你常常可以得到其他人的优质答案。与邮件列表相比，新闻组更适合提出初级问题，但如果你的问题是个“经常问到的问题（FAQ）”，就会被警告。

Ruby主要的新闻组是`comp.lang.ruby`，如果你没有安装新闻组软件，你可以在<http://groups.google.com/group/comp.lang.ruby>网页上阅读。在2006年，这个新闻组每天仍然有20个新帖，得到大量Ruby开发人员的经常使用。

5.4.3 因特网中继聊天工具

因特网中继聊天工具（Internet Relay Chat, IRC）是个实时的因特网聊天系统，成千上万的人可以聚集在“频道”中，讨论各种话题。实时聊天的即时性，让IRC适合快速提问和回答，不过参与者对于解答更深入的问题，常常有令人惊讶的热情（参见图5-3，这是个IRC频道运行示例）。唯一的缺点是你可能根本得不到响应，只能阅读正在进行中的谈话。

事实表明，IRC在Ruby开发人员中很流行，有如下两个特别引人注意的频道，全天24小时

提供对Ruby和Ruby on Rails的支持:

- 关于Ruby语言的讨论: #ruby-lang, 位于irc.freenode.net服务器。
- 关于Ruby on Rails的讨论: #rubyonrails, 位于irc.freenode.net服务器。

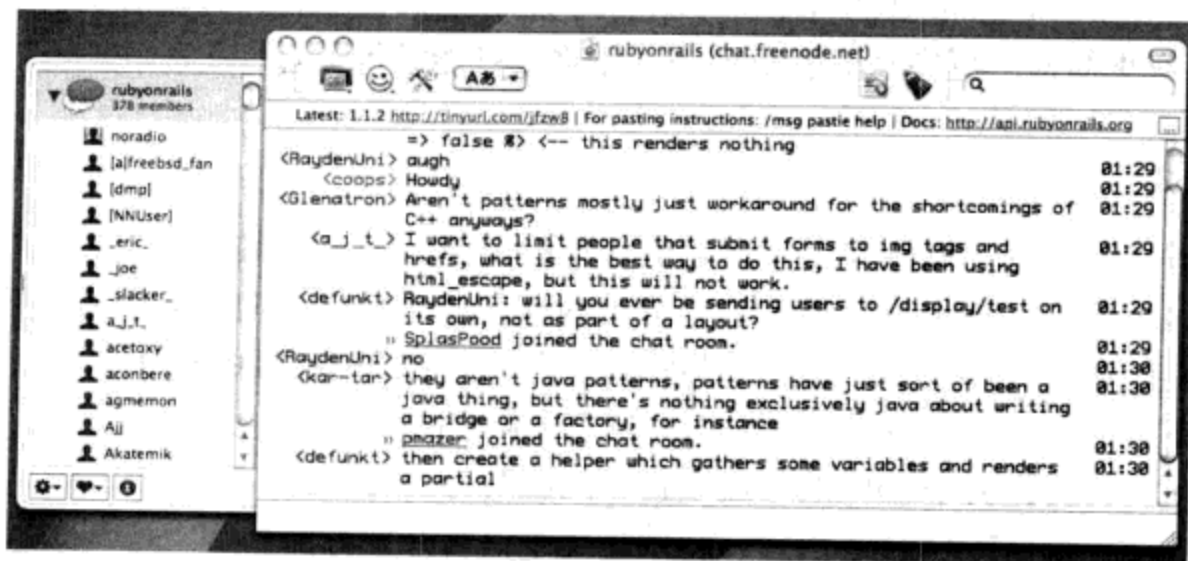


图5-3 IRC的Ruby on Rails频道谈话示例

要使用IRC, 必须下载IRC“客户端”软件, 并安装到你的计算机上, 用这个软件可以进入实时IRC频道。尽管安装这个软件超出了本书的范围, 我还是要推荐以下用于Windows、Linux和OS X的客户端:

- Windows: mIRC (<http://www.mirc.com>)。
- Mac OS X: Colloquy (<http://colloquy.info/>)。
- Linux/UNIX: XChat (<http://www.xchat.org/>)。

请务必尊重频道中的其他用户, 他们在那儿不是为了回答你的问题的, 否则你就会被看成“求助吸血鬼”, 并被大家当成空气般无视! 不过, 如果小心从事, 在这些频道中, 你也可以与Ruby界的大腕轻松交谈。

注 要了解IRC的更多信息, 请访问http://en.wikipedia.org/wiki/Internet_Relay_Chate网页。

5.4.4 文档

在网上有相当多的为Ruby开发人员准备的文档。最好的文档是位于<http://www.ruby-doc.org/>的官方参考文档, 完整(尽管常常只是基本)涵盖了Ruby标准类, 和最流行的Ruby程序库和插件。

Ruby当前稳定版本的API文档, 位于<http://www.ruby-doc.org/core/>。这是通过Ruby内置的文档工具rdoc, 从Ruby源代码中自动生成的文档, 文档结构不能一目了然。你通常可以选择阅读组成Ruby的某个文件的文档, 每个基类的文档, 或某个方法的文档。文档中没有逻辑顺序, 也没有深入的教程。这种文档仅为参考资料的用途而存在。

大多数Ruby程序库和应用系统都使用类似的文档风格, 在其官方网站上提供指向这类文档

的链接。例如，Ruby on Rails的API文档位于<http://api.rubyonrails.com/>。

5.4.5 论坛

在因特网中，一部分最流行的网站由论坛组成。与新闻组或邮件列表不同，它们一般活跃在更技术型人员的领域中，而论坛提供了网上非实时的讨论，极其容易访问。论坛是询问更基础问题并获得一般建议的好地方。

有几个Ruby论坛值得一看：

- Ruby-Forum.com (<http://www.ruby-forum.com/>)：Ruby-Forum.com为一些流行的Ruby邮件列表，提供了论坛式风格的界面。这意味着它不是严格意义上的真正论坛，但习惯用论坛的人将会喜欢它的结构。
- Ruby论坛 (<http://www.rubyforums.com/forumdisplay.php?f=1>)：Ruby论坛是大约15个更小论坛的组合，涵盖了Ruby的各个主题，从一般讨论到安装问题，到编辑器和Ruby on Rails。
- SitePoint Ruby论坛 (<http://www.sitepoint.com/forums/forumdisplay.php?f=227>)：SitePoint是个流行的Web开发网站，其中的论坛涵盖多个主题，其Ruby论坛于2005年10月建立。该论坛相当流行，非常友好。
- Rails论坛 (<http://railsforum.com/>)：Rails论坛是专注于Ruby on Rails的论坛，建立于2006年5月，其流行热度一直不减。该论坛对初学者特别友好。

5.5 加入社区

编程社区出现的原因之一，是人们可以从其他更有经验的人那里获得帮助，而且可以共同分享知识，开发有用的工具和文档。开发人员在接触一门新语言时，纯粹只从社区“获取”是很自然的，但重要的是当你掌握到一些知识之后，应该向社区回馈。Ruby开发人员对此很自豪，因为Ruby社区是最友好、最容易参与的社区之一，有好几种扬名立万的方法。

5.5.1 向别人提供帮助

在上一节中，我们考察了从其他Ruby开发人员那里获得帮助的方法，但如果你已经掌握一定程度Ruby知识，就能开始帮助别人。你可以参与IRC聊天室、论坛和邮件列表，并开始解答比你懂得更少的人们的问题。

帮助别人并不总是像看起来那么无私、费时。很多时候，要解答的问题，你只需要审视自己已知的知识，找出新知识点，或采用新方法，来解决已知问题即可。我的个人经验是，在IRC聊天室中帮助别人，可以持续拓展我的思路。尽管有时我可能有最佳答案，但有时我给出的答案也可能不准确或让人不明白，并被其他人纠正，这也让我获得新的认识。

在帮助别人这件事上，不要害怕尝试。如果你感到自己的答案是正确的，甚至并非是正确的，一般同时都会有好几个人提供帮助，而Ruby社区的人们一般都会原谅这种错误。在Ruby社区中，更看重努力，而不是技术水平。

5.5.2 贡献代码

当你开始开发自己的Ruby程序时，有时会发现系统或程序库中缺少某些你想要的功能，你可以自己开发这些功能，或对已有功能进行升级。如果项目是开源的，你就可以把自己的修改或更新提交给项目，这意味着你为整个社区改进了软件的质量。除了对别人有益，这还表示你的代码也可能被别人扩展或改进，这与把代码保留自用相比，将能从中获得更多的好处。

所有开源的Ruby程序库和应用系统，都有专人负责维护，如果你在项目网站上找不到相关功能的指示说明，请联系维护人员，看看能否贡献你的代码。

另外，如果你对自己的代码信心不足，但又看到项目文档中的巨大缺陷，甚至可能是Ruby本身的。如果你能提供文档，维护人员将会欣喜若狂。在第7章你将学会为Ruby程序生成文档的更多方法。许多程序员不善于写文档，或没有时间完成文档，因此如果你有这方面的技能，为项目贡献文档可能会让你大受欢迎！

5.5.3 网络博客

最近几年，开发人员写网络博客（weblog）（也称博客（blog）），已经是很平常的事。博客就像非正式的在线日记网站，其中充满了各种观点。原来仅供日记作者、哲学学者、开发人员使用的网络博客，现在变得极其流行。事实证明，它促进了Ruby的成功。

对于Ruby开发人员来说，在其网络博客中贴上一段新发现的知识，或有用的代码片断，这种事情并不少见。订阅这些网络博客，你的Ruby知识就可以一天天增长。看看许多其他程序员是怎样编码和解决问题的，将有助于拓展你的思路，而且你会发现，理想的代码片断或Ruby技巧会适时地出现在某人的网络博客上。

Ruby开发人员维护着许许多多的网络博客，而以下是其中最流行或最专注于Ruby的：

- **Ruby内幕** (<http://www.rubyinside.com/>)：本书的半官方站点。Ruby Inside博客每天贴出因特网上发现的有趣Ruby新闻、代码和教程的链接，是初学者和专家的理想去处。
- **红手** (<http://redhanded.hobix.com/>)：由魅力非凡的Ruby开发人员whytheluckystiff（为什么是幸运鬼）维护的博客，该博客专注于高超的Ruby技巧和前沿的Ruby新闻，是高级Ruby开发人员的最爱。
- **松本行弘的博客** (<http://www.rubyist.net/~matz/>)：由于Matz是Ruby的主要开发人员，很多用户都喜欢查看他的博客。尽管该博客是日文的，但常常有值得一看的有趣代码片断或演示。
- **Ruby On Rails星球** (<http://www.planetrubyonrails.com/>)：如果你没时间阅读大量的不同博客，那么Ruby On Rails星球将满足你的需要，该博客把大多数最佳Ruby博客的内容汇集在个页面里。尽管它的名字是Ruby On Rails，但非常侧重于Ruby，当然也关注Rails。

访问这些博客，你将快速了解许许多多的Ruby资源、技巧和源代码文档。如果你对这些站点进行评论，并开始根据自己的Ruby经验来更新你的博客，你将很快得到Ruby社区的认可。

5.6 小结

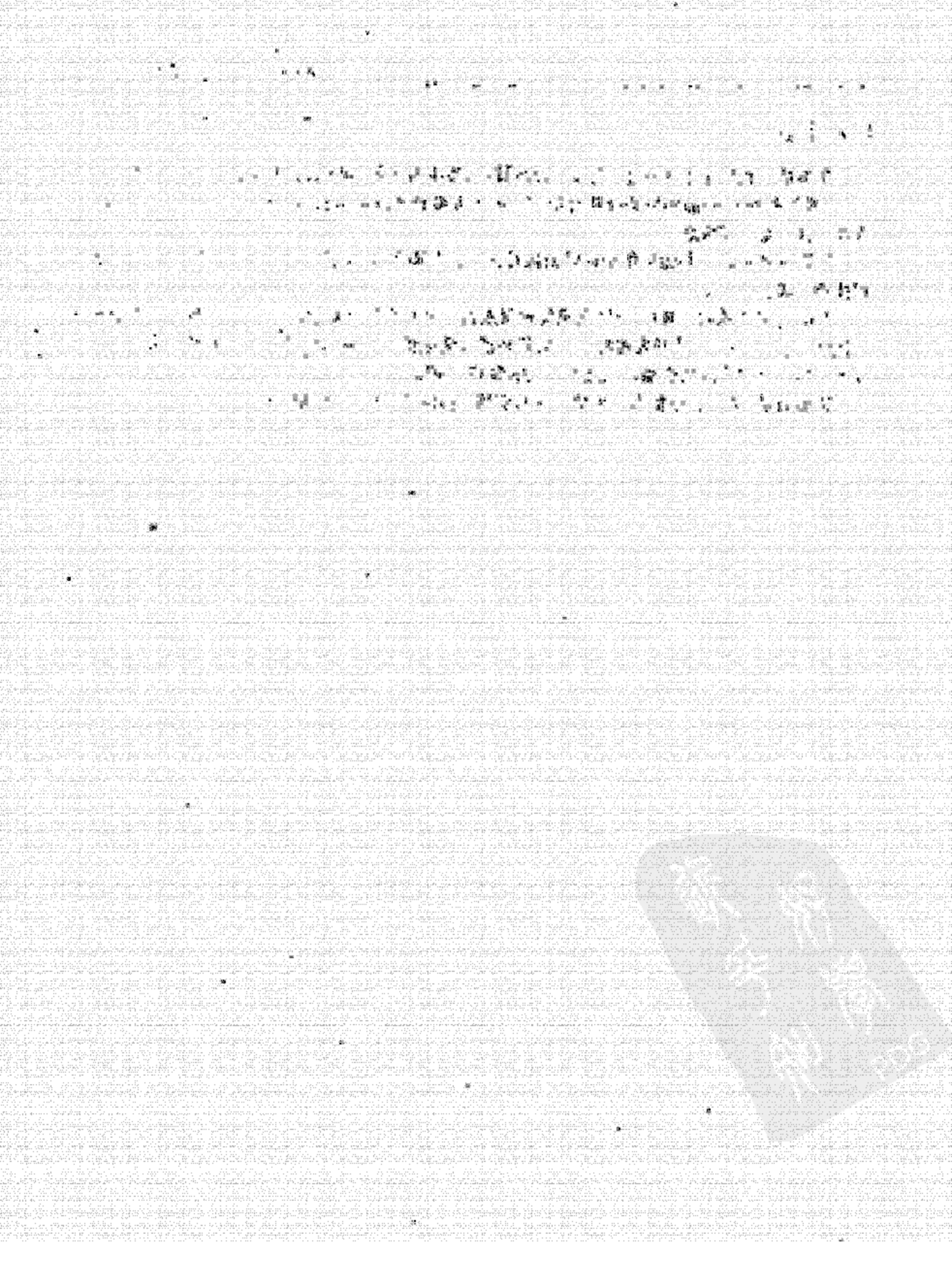
在本章，我们小小休息了一下，从代码转而关注围绕着Ruby语言的文化、社区和“生态系统”。理解Ruby的外部环境是极其有用的，因为大多数开发人员将从这个社区获得帮助、建议、代码，甚至是工作报酬。

能够得到帮助，并能提供帮助以回报社区，这是Ruby的发展动力，并将最终有助于自己编程技巧的成长。

对Ruby新人来说，Ruby社区很重要也很友善，因此当你开始学习Ruby时，最好尽快融入这个社区。在学习Ruby并开发程序时，请务必把社区提供的资源用到极致。一本书无法让你变成专家，但一组有价值的资源，以及对社区的参与则可以。

请参阅附录C，其中提供了大量URL和链接，指向其他Ruby在线资源。





第二篇 Ruby的核心

本篇带你领略Ruby其余的基础元素，并探讨前文提到的一些语言特性，深入其中的细节。到本篇结束时，你将能随心所欲地组织复杂的类和对象，进行Ruby应用程序开发，并知道怎样测试、写文档和部署，以及使用数据库和外部数据源为应用程序获取数据。

第6章 类、对象和模块

在第2章我们直接深入面向对象的原理，也就是Ruby用类和对象来表达概念的方法。接着我们介绍了Ruby的标准类，例如String和Array，并用它们进行了一些操作，然后又跳到Ruby的逻辑和其他核心功能等话题上。

本章关注的焦点重新回到面向对象，但不是远观总体概念，而是近距离考察细节。我们将考察类和对象为什么有那样的表现行为，面向对象为什么是行之有效的开发工具，怎样实现满足需求的类，以及怎样覆写（override）和扩展（extend）Ruby的默认类。最后，将以文本冒险游戏的形式，实现一个基本的“地下城”，来展现各种各样实际生活的概念，以及怎样合并到一组易于维护的、相互连接的类中。

6.1 为什么要用面向对象

面向对象不是软件开发的唯一开发方式。在面向对象方法之前，有过程（procedural）式编程方法，这种编程方法在C语言中仍然使用。面向对象强制要求必须以类的形式定义概念和过程，并根据类创建对象，而过程式编程则强调完成任务所需的步骤，对如何管理数据并不关心。

假定某个开发公司中有两个开发人员，他们在竞争成为公司中最有见识的程序员。为了利用他们的竞争，老板给两人安排了一组相同的任务，并择优选用他们的工作成果。两个程序员的唯一区别是，其中一人使用面向对象开发的原理，另一人使用过程式编程，不采用类和对象。

对于即将到来的项目，老板要求开发一段代码，算出不同形状的周长和面积，指定的形状包括正方形和三角形。

过程式程序员冲出门去，很快带回4个不同的函数：

```
def perimeter_of_square(side_length)
  side_length * 4
end
```

```
def area_of_square(side_length)
  side_length * side_length
```

```
end

def perimeter_of_triangle(side1,side2,side3)
  side1 +side2 +side3
end

def area_of_triangle(base_width,height)
  base_width *height /2
end
```

注 请记住，在Ruby中不必使用return来从方法中返回值。方法中最后一个表达式的值，将默认作为方法的返回值。

由于过程式程序员首先完成，因此他自信满满地认为，自己的代码将被选中。

面向对象程序员花了更长的时间。他认识到需求和规格将来可能会变更，因此先定义Shape类，再从该类继承创建出新类，这样会比较有用。也就是说，如果需要向形状中加入额外的通用功能，代码将是现成的。他提交了如下的初始解决方案：

```
class Shape
end

class Square <Shape
  def initialize(side_length)
    @side_length =side_length
  end

  def area
    @side_length *@side_length
  end

  def perimeter
    @side_length *4
  end
end

class Triangle <Shape
  def initialize(base_width,height,side1,side2,side3)
    @base_width =base_width
    @height =height
    @side1 =side1
    @side2 =side2
    @side3 =side3
  end

  def area
    @base_width *@height /2
  end
end
```

```

def perimeter
  @side1 + @side2 + @side3
end
end

```

注 目前这段代码看似复杂难懂，本章后面将介绍此处所用的技术。就目前来说，只需看懂类和方法的布局结构即可，这些在第2章中已有介绍。

过程式程序员对面向对象方案不屑一顾。“为什么到处都是无意义的数据赋值？”他嘲笑道，“面向对象的代码90%是框架结构，只有10%是逻辑处理！”

简短精炼的过程式代码给老板留下了深刻印象，但他想两种都试试。他很快发现了一个重大区别：

```

puts area_of_triangle(6,6)
puts perimeter_of_square(5)

```

```
18
```

```
20
```

```

my_square = Square.new(5)
my_triangle = Triangle.new(6, 6, 7.81, 7.81, 7.81)
puts my_square.area
puts my_square.perimeter
puts my_triangle.area
puts my_triangle.perimeter

```

```
25
```

```
20
```

```
18
```

```
23.43
```

老板注意到，用面向对象代码，他可以有条理地按自己需要创建任意多个形状，而过程式代码需要他自己记住要处理的形状。尽管如此，他还是有些担心。

“更多的代码行，代表着需要更多时间，”他说，“如果面向对象方法只是代表着更多代码行、更多时间和更多争论，那么它值得采用吗？”

面向对象程序员以前就听到过他的抱怨，并立即付诸行动。“请用大量的任意形状试试”，他说。

老板并没有完全跟上现代开发趋势，但当他发现，只须复制和粘贴现有类代码，并作少许修改，即可处理许多新的形状类型，他开始被说服了。他还发现，如果把形状存储在对象中，由一个变量引用，那么如果每个形状的都接受相同的方法，它到底是什么形状就无足轻重。他可以对任何形状调用`perimeter`或`area`方法，而无须担心出什么问题。而过程式代码则不同，只是各种方法的杂乱堆积，程序员被迫关注不同的形状类型，以便知道运行哪一个过程。`Shape`类也提供了一种途径，如果将来所有各种类型的形状都需要某种通用功能，则通过`Shape`类提供。老板现在知道该选择谁的代码了！

他说“面向对象代码需要稍多一些的搭建，但如果扩大代码规模，以适应实际生活的需求，它没有竞争对手！”

面向对象编程的基本优势在于，尽管在搭建代码时需要更多的结构，但非专家读者很容易理解类和对象的关系，也更容易维护和更新代码，以便适应实际生活的情况。

6.2 面向对象基础知识

我们来回顾一下在前几章学到的类和对象的基础知识：

类是对象的蓝图。我们只有一个名为Shape的类，但通过这个类可以创建多个形状实例（或称形状对象），所有实例都有Shape类中定义的方法和属性。

对象是类的实例（instance）。如果类是Shape，则`x = Shape.new`创建了新的Shape实例，并将其赋值给变量x。然后就可以称x是个Shape对象，或是个Shape类的对象。

6.2.1 局部变量、全局变量、对象变量和类变量

在第2章中，我们创建了一些类，并向其中加入了一些方法。为了回顾一下，我们来看一个简单的例子，演示包含两个方法的类及其使用方法。下面是类本身的定义代码：

```
class Square
  def initialize(side_length)
    @side_length = side_length
  end

  def area
    @side_length * @side_length
  end
end
```

下面，我们创建一些正方形（square）对象，并调用其area方法：

```
a = Square.new(10)
b = Square.new(5)
puts a.area
puts b.area
```

```
100
```

```
25
```

Square类第一个方法——当我说“第一个”，是指我们例子中的第一个方法。initialize与代码中的实际方法顺序无关。initialize是个特殊方法，在基于该类创建新对象时，调用这一方法。当调用Square.new(10)时，Square类就会创建一个新对象实例，然后调用该对象的initialize方法。

在本例中，initialize接受由Square.new(10)传递过来的side_length参数，将数字10赋值给名为@side_length的变量。此时变量名之前的@符号非常重要，但为什么呢？要理解有些变量名之前有某种符号前缀的原因，需要理解变量有多种类型，例如局部变量、全

局变量、对象变量和类变量。

局部变量

在上例中，我们已经创建过变量，例如：

```
x = 10
puts x
```

```
10
```

在Ruby中，这种基本变量称为局部变量 (local variable)，只能在变量定义的不同地方使用。如果转而调用对象的方法，或调用自己的独立方法，则变量x不再跟你在一起，它的作用域 (scope) 被视为局部的。也就是说，它只在局部代码区域内出现。下面是个演示的例子：

```
def basic_method
  puts x
end
```

```
x = 10
basic_method
```

本例定义x等于10，然后跳转到名为basic_method的局部方法。如果用irb运行这段代码，将得到如下的出错信息：

```
NameError: undefined local variable or method `x' for main:Object
from (irb):2:in `basic_method'
```

发生了什么事？当跳转到basic_method时，你不再处于创建变量x时所在的作用域。因为x是个局部变量，它只存在于定义的地方。为了避免这个问题，必须记住，只在直接用到局部变量的地方调用局部变量，这也正是其用途所在。

在下面的例子中，有两个局部变量，它们有相同的名字，但处在不同的作用域中：

```
def basic_method
  x = 50
  puts x
end
```

```
x = 10
basic_method

puts x
```

```
50
10
```

这个例子说明，局部变量只存在于它们原始的作用域。在主代码中，把x设为10，然后在方法中把x设为50，但当你回到原始范围时，x仍然是10。basic_method方法中的变量x与方法外的变量x不是一回事。它们是两个不同的变量，分别处在各自的作用域里。

全局变量

与局部变量直接相对的，Ruby还可以使用全局变量（global variables）。它的名字很大程度上暗示了，全局变量在程序的任何地方都可以访问，包括在类或对象中。

全局变量很有用，但在Ruby中并不常用。它与面向对象编程的思想无法水乳交融，因为一旦在程序中使用全局变量，代码就可能会依赖于它们。面向对象编程有一个有用的能力，是能把逻辑代码块相互隔离，因此全局变量在此并不受欢迎。不过，本书后面将会再次接触到全局变量，因此了解怎样构建全局变量，还是有用的。

通过在变量名之前加美元符号（\$），即可定义全局变量，如下所示：

```
def basic_method
  puts $x
end

$x = 10
basic_method
```

```
10
```

\$x被定义为全局变量，你可以在程序的任何地方调用它。

实例变量

局部变量被限制于局部作用域，全局变量有全局作用域，而对象变量（object variable）的得名，是由于其作用域内置于、关联于当前对象。本节开头的Square类代码演示了这个概念：

```
class Square
  def initialize(side_length)
    @side_length = side_length
  end

  def area
    @side_length * @side_length
  end
end
```

对象变量有个@符号前缀。在Square类中，把提供给类的side_length，赋值给@side_length。而@side_length作为对象变量，即可在该对象的任何其他方法中访问。这就是area方法为何能调用@side_length，来计算该对象所代表的正方形的面积：

```
a = Square.new(10)
b = Square.new(5)
puts a.area
puts b.area
```

```
100
```

```
25
```

两个正方形的面积计算结果是不同的，尽管计算面积的代码都是@side_length * @side_length。这是因为@side_length是个对象变量，只关联于当前对象或实例。

提示 如果你没有完全理解本章开头的Shape/Square/Triangle示例，现在是个好时机，可以回头再看一下，其中用了几个对象变量来实现其功能。

类变量

最后一种主要的变量类型是类变量 (class variable)。类变量的作用域处于整个类中，而不是处于该类的特定对象中。与对象变量的单个@符号相比，类变量以两个@符号 (@@) 作为前缀。

对于存储与某类所有对象都相关的信息，类变量特别有用。例如，可以把目前为止某个类已创建的对象数目保存在类变量中，如下所示：

```
class Square
  def initialize
    if defined?(@@number_of_squares)
      @@number_of_squares += 1
    else
      @@number_of_squares = 1
    end
  end
end
```

由于@@number_of_squares是个类变量，因此每次创建新对象时它已经存在（除了第一次，这正是检查它是否已存在的原因，如果不存在，则向其赋予初始值1）。

注 在第3章中你已经学过三元运算符，可用来简化上述方法中的代码，将其缩减为
 @@number_of_squares = defined?(@@number_of_squares) ?
 @@number_of_squares + 1 : 1。

6.2.2 类方法和对象方法

在Square类中，定义了两个方法：initialize和area。这两个方法都是对象方法，因为它们与对象相关，并直接对对象进行操作。下面再次列出这段代码：

```
class Square
  def initialize(side_length)
    @side_length = side_length
  end

  def area
    @side_length * @side_length
  end
end
```

当用s = Square.new(10)语句创建了正方形之后，即可用s.area语句得到s所代表的正方形的面积。area方法是Square类的所有对象都可以使用的方法，因此被视为对象方法。

不过，方法并不只在对象实例中有用，也可以直接对类本身操作。在上节中，我们使用过类变量，来保存已经创建多少正方形对象的统计值，因此，以某种方式访问@@number_of_

squares类变量，而非通过Square对象来访问，也是很有用处的。

下面是类方法的简单示例：

```
class Square
  def self.test_method
    puts "Hello from the Square class!"
  end

  def test_method
    puts "Hello from an instance of class Square!"
  end
end

Square.test_method
Square.new.test_method
```

```
Hello from the Square class!
Hello from an instance of class Square!
```

这个类有两个方法，尽管二者的名字相同，都是test_method，但第一个是类方法，第二个是实例方法。二者的区别在于，类方法由self.前缀标示，这里self表示当前类，因此def self.test_method定义的方法专用于该类。而没有前缀的方法，则自动成为实例方法。

还可以用另一种方式来定义类方法，如下所示：

```
class Square
  def Square.test_method
    puts "Hello from the Square class!"
  end
end
```

用哪种风格 (ClassName.method_name或self.method_name) 来定义类方法，取决于个人喜好。使用self.method_name方式 (如示例中的self.test_method) 不需要一次次重新声明类名，而用ClassName.method_name方式 (如示例中的Square.test_method) 则更贴近以后对该方法的调用。

注 本书后续章节会使用ClassName.method_name风格，但你在编写代码时可以用自己喜欢的方式。

类方法提供了一种机制，可以很好地实现前文所述的“对象计数器”，代码如下：

```
class Square
  def initialize
    if defined?(@@number_of_squares)
      @@number_of_squares += 1
    else
      @@number_of_squares = 1
    end
  end
end
```

```

def Square.count
  @@number_of_squares
end
end

```

我们来试一试：

```

a = Square.new
puts Square.count
b = Square.new
puts Square.count
c = Square.new
puts Square.count

```

```

1
2
3

```

请注意，在得到统计值时，你根本没有引用a、b、c对象，只是直接用Square.count类方法。可以把这一操作看成是“请求类”（而不是请求对象）做与整个类相关的事情。

6.2.3 继承

面向对象编程最有趣的概念之一是继承（inheritance），通过继承，可以生成一整套的类和对象。如果把所有生物都看成名为LivingThing（生物）的类（参见图6-1），那么在该类之下，可以有Plant（植物）和Animal（动物）类（生物学家们，请不要吹毛求疵，让我们简单一点吧！）在Animal类之下，可以有Mammal（哺乳类）、Fish（鱼类）和Amphibian（两栖类）等类。再深入Mammal类，可以得到Primate（灵长类）和Human（人类）。因此，Human是LivingThing，是Animal，是Mammal，……而每下一层，都比上一层更具体和精确。这就是类继承的实际应用！同样的体系也适用于Shape的例子，在这个例子中，Triangle和Square都直接从Shape继承而来。

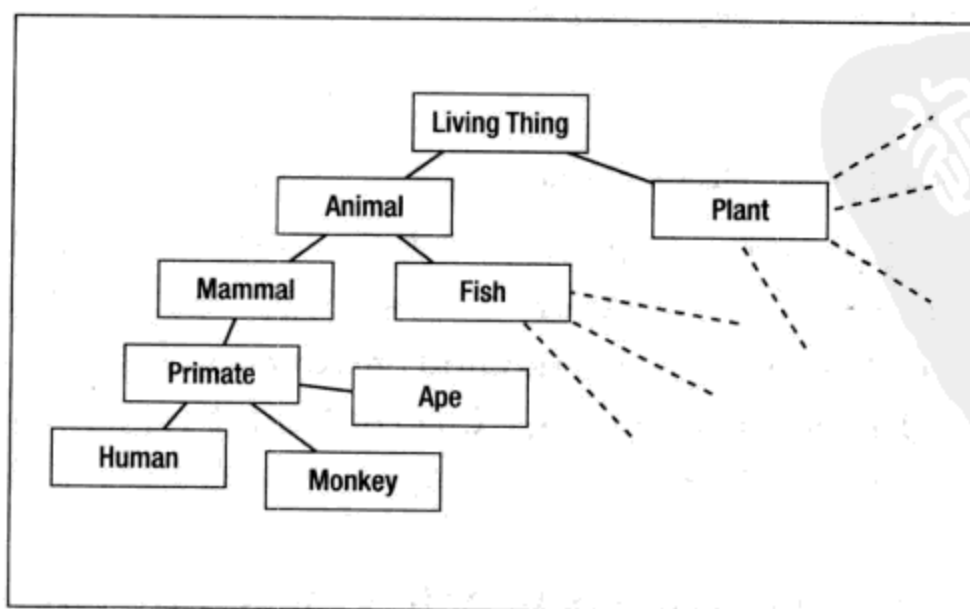


图6-1 “生物”的继承体系示例

继承的好处在于，底层类拥有上层类的功能，但也可以增加它们自有的功能。基本的“生物”类太泛泛了，你只能给它定义基本的live（活）或dead（死）方法。但在“动物”这一级，就可以增加eat（吃）、excrete（排泄）或breathe（呼吸）方法。在“人”这一级，就继承了所有这些方法，但还可以增加人类自有的方法和属性，例如sing（唱歌）、dance（跳舞）和love（爱）。

Ruby的继承功能同样简单。任何类都可以继承另一个类的功能特性，但只能从单个类继承。有些其他语言支持多重继承（multiple inheritance），即允许从多个类继承，但Ruby不支持。多重继承可能导致混乱的情况——例如，多重继承可能造成各个类之间无穷尽的循环——因此多重继承的效果是有争议的。

我们来看一下继承在代码中的表现形式：

```
class ParentClass
  def method1
    puts "Hello from method1 in the parent class"
  end

  def method2
    puts "Hello from method2 in the parent class"
  end
end

class ChildClass < ParentClass
  def method2
    puts "Hello from method2 in the child class"
  end
end

my_object = ChildClass.new
my_object.method1
```

```
Hello from method1 in the parent class
```

```
my_object.method2
```

```
Hello from method2 in the child class
```

首先创建ParentClass类，其中包含两个方法：method1和method2。然后创建ChildClass类，用ChildClass < ParentClass的写法，令其继承ParentClass。最后，创建ChildClass类的对象实例，并调用其method1和method2方法。

第一个调用方法完美地展示了继承的效果。ChildClass类自己没有method1方法，但由于它继承自ParentClass类，而ParentClass类有method1方法，因此ChildClass类可以调用这个方法。

但在第二个调用方法中，ChildClass类已经有method2方法，因此父类的method2方法被忽略了。在许多情况下，这是一种理想的行为方式，因为可以用更具体类的行为，覆写更一

般类的行为。但在某些情况下，你可能想让子类调用继承自父类的方法，并对调用结果进行一番处理。

我们来看下面的情况，基本类代表了不同的人：

```
class Person
  def initialize(name)
    @name = name
  end

  def name
    return @name
  end
end

class Doctor < Person
  def name
    "Dr. " + super
  end
end
```

在本例中有个Person（人）类，它实现存储和返回人名的基本功能。Doctor（医生）类继承自Person类，并覆写其name方法。在医生的name方法中，返回以“Dr.”开头、后跟正常姓名的字符串。这里使用了super，上溯继承链，并调用上一级的同名方法。在本例中，有两个层级，因此在name方法中使用super，将调用Person类的name方法。

这样使用继承的好处是，可以在一般类中实现通用功能，然后在更具体的子类中，只实现所需的特定功能。这样一方面节省了大量重复劳动，另一方面如果对父类进行修改，子类也将继承这些修改。修改Person类，增加两个参数firstname和lastname，就是很好的例子。此时Doctor类无须任何修改，即可支持这一变化。在本例中只有一个子类，这种好处似乎并不明显，但当你的应用程序中有上百个不同的类，就能减少大量的重复！

注 减少重复的概念通常称为DRY，即“Don't Repeat Yourself”（不重复自己）。如果你写一段代码，并在不同地方多次重用这段代码，这就是DRY的最好方法。

6.2.4 覆写现有方法

由于Ruby是一种动态语言，因此你可以对它做一件巧妙的事情，即覆写现有类和方法。例如，对于Ruby的String类，如第3章所示，创建字符串，就是创建了String类的对象。例如：

```
x = "This is a test"
puts x.class
```

String

对于x所保存String对象，可以对其调用许多方法：

```
puts x.length
```

```
puts x.upcase
```

```
14
```

```
THIS IS A TEST
```

我们来把水搅浑一些，覆写length方法：

```
class String
  def length
    20
  end
end
```

许多Ruby新手，甚至是有经验的开发人员，一开始都不敢相信这段代码真的管用，但结果正如代码所要求的那样：

```
puts "This is a test".length
puts "a".length
puts "A really long line of text".length
```

```
20
```

```
20
```

```
20
```

Ruby的一些程序库和扩展（插件）覆写了核心类的方法，以扩展Ruby的通用功能。但从这个例子可以看出，为什么总是要慎重小心，时刻注意应用程序中发生了什么事。如果我们依赖于length方法来测量字符串长度，而length方法却被覆写了，那就会遇到大麻烦！

另外值得一提的是，也可以覆盖自己的方法。事实上，通过在irb中输入这些示例代码，你可能已经做过很多这样的事了：

```
class Dog
  def talk
    puts "Woof!"
  end
end
```

```
my_dog = Dog.new
my_dog.talk
```

```
Woof!
```

```
class Dog
  def talk
    puts "Howl!"
  end
end
```

```
my_dog.talk
```

```
Howl!
```

在本例中，我们创建了基本类，包含了一个简单方法，然后重新打开该类，在运行中重新定义了方法。结果是重新定义的方法立即生效，`my_dog`对象不再轻吠，而是开始吼叫。

Ruby这种重新打开类并增加或重定义方法的能力，在各种面向对象语言中是相对独特的。尽管你可以用它来实现一些有趣的技巧（你会在后文中看到一些），但也会导致同一段代码的执行结果不同，这取决于你依赖的某个类是否被应用程序修改，如前文对String类的length方法所做的那样。

注 你可能已经注意到，在我们前面的例子中，已有这种重新打开类的技术方法的应用：首先在一个例子中创建方法，而在后续例子中只增加新方法。如果在irb或同一程序中运行这些代码，重新打开类可以增加新方法或修改旧方法，但不会丢失其他任何东西。

6.2.5 对象方法的反射与发现

反射是指计算机程序在运行和使用中，检视、分析并修改自身的过程。Ruby把反射用到极致，允许在运行自己的代码时，修改语言自身的大部分功能。

在Ruby中，可以查询几乎任何对象所定义的方法。这是反射的另一部分应用。

```
a = "This is a test"
puts a.methods.join('')
```

```
methods instance_eval %rindex map <<split any?dup sort strip size
instance_variables downcase min gsub!count include?succ!instance_of?extend
downcase!intern squeeze!eql?*next find_all each rstrip!each_line +id sub
slice!hash singleton_methods tr replace inject reverse taint unpack sort_by
lstrip frozen?instance_variable_get capitalize max chop!method kind_of?
capitalize!scan select to_a display each_byte type casecmp gsub protected_methods
empty?to_str partition tr_s tr!match grep rstrip to_sym instance_variable_set
next!swapcase chomp!is_a?swapcase!ljust respond_to?between?reject to_supto
hex sum class object_id reverse!chop <=>insert <tainted?private_methods ==
delete dump == __id__member?tr_s!>concat nil?untaint succ find strip!
each_with_index >=to_i rjust <=send index collect inspect slice oct all?clone
length entries chomp ==public_methods upcase sub!squeeze __send__upcase!crypt
delete!equal?freeze detect zip [] lstrip!!center []=to_f
```

对于任何对象，`methods`方法都返回该对象可用方法的数组（当然，除非`methods`方法被覆写了！）。由于Ruby极其面向对象的结构，这些方法的数量通常相当大，比自己定义的方法要多得多！

从以上结果中也可以看到另外一些反射方法。例如，`protected_methods`、`private_methods`和`public_methods`都以不同的封装方式，显示相关方法（详见下一节）。

另一个有意思的方法是`instance_variables`，该方法返回对象实例所关联的实例变量（而非类变量）。

```
class Person
```

```
attr_accessor :name, :age
end

p = Person.new
p.name = "Fred"
p.age = 20
puts p.instance_variables.inspect
```

```
["@age", "@name"]
```

目前你可能看不出这些反射方法的价值，但当你逐渐成为Ruby大师时，这些方法就越来越重要。本书不深入讨论元编程（metaprogramming）的艺术和高级反射技术，因为尽管这些主题都很有意思，但除非你达到相当的能力水平，一般不会经常用到，因此不在本书这本初学者书籍的范围内。

6.2.6 封装

封装是对象的一种能力，它让某些方法和属性变得公共可用（对任何位置的代码），但除此之外的其他方法和属性，只在该类自身或同一类的其他对象中可见。

注 在更技术的层面上，封装是指对象将其组成数据隐藏在抽象接口之后的能力，此处隐秘包含了这一能力。

封装的基本原理是向类之外尽可能少地暴露方法，这样一来，即使重写类的内部结构，也可以维持少量方法不变，这些方法是该类和对象与系统中其他元素之间的接口。封装有助于把大量功能放在类中，并提供安全性。

下面是个示例类，用以表达人（person）的概念：

```
class Person
  def initialize(name)
    set_name(name)
  end

  def name
    @first_name + ' ' + @last_name
  end

  def set_name(name)
    first_name, last_name = name.split(/\s+/)
    set_first_name(first_name)
    set_last_name(last_name)
  end

  def set_first_name(name)
    @first_name = name
  end
end
```



```

    def set_last_name(name)
      @last_name = name
    end
  end
end

```

在前文的示例中，你已经用单句`attr_accessor :name`实现了上面的功能，并把`name`赋值给一个对象变量。不幸的是，实际生活的限制常常需要我们采用另外的方法来解决。

在这一情况下，名和姓在每个`Person`对象中分别存储，放在名为`@first_name`和`@last_name`的对象变量中。当`Person`对象创建时，姓名被分为两半，通过相应的`set_first_name`和`set_last_name`方法，每一半都赋值给正确的对象变量。为什么要采用这样的实现方法，一个可能的原因是，尽管你想在程序中用全名，数据库设计却可能要求把名和姓放在两个字段中，因此你需要在类代码中进行处理，以便隐藏这一差异，如上述代码所示。

注 这种方法还有一个附带的好处，即可以在把数据赋值给对象变量之前，对数据进行检查。例如，在`set_first_name`和`set_last_name`方法中，我们可以检查姓名是否包含足够的字符以构成合法的名字。如果没有，则可以抛出错误。

下面的代码看起来一切正常：

```

p = Person.new("Fred Bloggs")
puts p.name

```

Fred Bloggs

但似乎仍然存在问题：

```

p = Person.new("Fred Bloggs")
p.set_last_name("Smith")
puts p.name

```

Fred Smith

麻烦了！你本想把关于名和姓的需求抽象出来，仅允许读写全名。但`set_first_name`和`set_last_name`仍然是公共方法，因此可以在拥有`Person`对象的任何代码中直接调用。幸运的是，封闭可以让你解决这一问题：

```

class Person
  def initialize(name)
    set_name(name)
  end

  def name
    @first_name + ' ' + @last_name
  end

  private

  def set_name(name)

```

```

    first_name, last_name = name.split(/\s+/)
    set_first_name(first_name)
    set_last_name(last_name)
  end

  def set_first_name(name)
    @first_name = name
  end

  def set_last_name(name)
    @last_name = name
  end
end

```

这个Person类与前文第一个版本的唯一区别，是增加了private关键字。private的作用是告诉Ruby，从此往下，本类中声明的任何方法都应保持“私有”状态。这表示只有对象内部的代码可以访问这些私有方法，而类之外的代码则无法访问。例如，下面的代码不再可行：

```

p = Person.new("Fred Bloggs")
p.set_last_name("Smith")

```

```

NoMethodError: private method 'set_last_name' called for #<Person:0x337b68
@last_name="Bloggs", @first_name="Fred">

```

private关键字的反义词是public。我们可以把private放在任何方法之前，然后在其后放上public，把后续方法转置为公共方法。如下所示：

```

class Person
  def anyone_can_access_this
    ...
  end

  private
  def this_is_private
    ...
  end

  public
  def another_public_method
    ...
  end
end

```

你还可以把private当成命令使用，向其传递一组符号参数，表示你想把这些符号表示的方法设置为私有的，如下所示：

```

class Person
  def anyone_can_access_this; ...; end

  def this_is_private; ...; end
end

```

```

def this_is_also_private; ...; end

def another_public_method; ...; end

private :this_is_private, :this_is_also_private
end

```

注 Ruby支持行尾加分号 (;), 并允许把多个代码放在一行中 (例如, `x=10; x+=1; puts x`)。本例采用这种方法来节省代码行, 但从产品的代码质量的角度来看, 这不是一种好的编码风格。

该命令告诉Ruby, `this_is_private`和`this_is_also_private`方法将被放到私有方法中。到底是在方法之前用`private`指令, 还是以命令的形式直接指定方法名, 完全取决于你的选择, 这也是作为Ruby程序员, 将要作出的许多技术上不太重要的、关于代码风格的决定之一。不过, 必须注意的是, 在上例中`private`声明必须放在方法定义之后。

Ruby支持封闭的第三种形式 (与`public`和`private`不同), 名为`protected`, 它让方法成为私有的, 但使用范围是在类的范围内, 而非仅在单个对象中。例如, 在对象及其方法的范围之外, 无法直接调用私有方法, 但可以在相同类的任何成员对象的方法中, 调用`protected`类型的方法:

```

class Person
  def initialize(age)
    @age = age
  end

  def age
    @age
  end

  def age_difference_with(other_person)
    (self.age - other_person.age).abs
  end

  protected :age
end

fred = Person.new(34)
chris = Person.new(25)
puts chris.age_difference_with(fred)
puts chris.age

```

```

9
:20: protected method `age' called for #<Person:0x1e5f28 @age=25>
(NoMethodError)

```

上面的例子使用了`protected`类型的方法, 因此无法直接调用`age`方法, 除非在`Person`

类对象的任何方法中调用。不过，如果age方法被设为private，则上例将会失败，因为other_person.age是非法调用。这是因为private让方法调用只对特定对象内的方法开放。

请注意，当在最后一行直接调用age方法时，Ruby会抛出一个异常。

6.2.7 多态

多态的概念是指编写代码可以同时适用于多种类型和类的对象。例如，+方法适用于数据相加、字符串相连，以及数组合并。+方法具体做什么操作，完全依赖于你要相加的东西是什么类型。

多态有一种常见的演示示例，以下是其Ruby表达形式：

```
class Animal
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

class Cat < Animal
  def talk
    "Meaow!"
  end
end

class Dog < Animal
  def talk
    "Woof!"
  end
end

animals = [Cat.new("Flossie"), Dog.new("Fido"), Cat.new("Tinkle")]
animals.each do |animal|
  puts animal.talk
end
```

```
Meaow!
Woof!
Meaow!
```

在本例中，定义了三个类：Animal（动物）类，以及从Animal继承的Dog（狗）和Cat（猫）类。在最后一段代码中，创建了一个数组，其中存放不同种类的动物对象：两个Cat对象和一个Dog对象（其名字都由来自Animal类的通用initialize方法处理）。

下一步，对每个动物进行迭代循环。在每次循环中，把动物对象放到局部变量animal中。最后，轮流对每个动物运行puts animal.talk。由于talk方法在Cat和Dog类中均有定义，

但输出结果不同，因此你得到了正确的输出：两个“Meaow!”和一个“Woof!”。

通过这一演示，我们可以看出，怎样循环处理不同类的对象，每次都得到正确的结果，假如每个类都实现相同方法的话。

如果你打算用继承在Cat和Dog类之下创建新类（例如`class Labrador < Dog`），那么`Labrador.new.talk`将仍然返回“Woof!”。感谢继承！

有些Ruby内置标准类（例如Array、Hash、String等）拥有自己的多态方法。例如，可以对许多内置类调用`to_s`方法，以字符串形式返回对象的内容：

```
puts 1000.to_s
puts [1,2,3].to_s
puts ({ :name => 'Fred', :age => 10 }).to_s
```

```
1000
123
age10nameFred
```

在本例中，输出结果并不是特别有用，但在许多情况下，大多数对象都可以依靠`to_s`方法来返回字符串，这是特别有用的。

6.2.8 嵌套类

在Ruby中，可以把类放入其他类之中，这样的类称为嵌套（nested）类。如果某个类依赖于其他类，而这些其他类仅在此处有用，在其他地方无用，在这样的情况下，嵌套类就很有用。如果想把类划分为类组，而不是各自独立时，嵌套类也很有用。示例如下：

```
class Drawing
  class Line
  end

  class Circle
  end
end
```

嵌套类与普通类的定义方法完全相同，但用法不同。

在Drawing类中，可以直接访问Line和Circle类，但在Drawing类之外，则只能以`Drawing::Line`和`Drawing::Circle`的形式访问Line和Circle。例如：

```
class Drawing
  def Drawing.give_me_a_circle
    Circle.new
  end

  class Line
  end

  class Circle
```



```

    def what_am_i
      "This is a circle"
    end
  end
end

a = Drawing.give_me_a_circle
puts a.what_am_i
a = Drawing::Circle.new
puts a.what_am_i
a = Circle.new
puts a.what_am_i

```

```

This is a circle
This is a circle
NameError: uninitialized constant Circle

```

`a = Drawing.give_me_a_circle`调用了`give_me_a_circle`类方法，返回`Drawing::Circle`的新实例。下一步，`a = Drawing::Circle.new`直接得到`Drawing::Circle`的新实例，而`a = Circle.new`并未成功，因为`Circle`类不存在。这是由于`Circle`是`Drawing`之下的嵌套类，只能以`Drawing::Circle`的形式为人所知。

在本章末尾的项目中，你将会习惯嵌套类，并且会看到它们怎样在整个程序的范围内有效工作。

6.2.9 常量的作用域

在第3章中，你已经见过常量：其值在程序范围内永不变更，因此它是一个特殊变量，例如`Pi = 3.141592`。示例如下：

```

def circumference_of_circle(radius)
  2 * Pi * radius
end

Pi = 3.141592
puts circumference_of_circle(10)

```

```

31.41592

```

在这种意义下，常量似乎与全局变量很像，但却不是。常量定义在当前类的作用域内，除非被覆写，否则所有子类均可访问。例如：

```

Pi = 3.141592

class OtherPlanet
  Pi = 4.5

  def OtherPlanet.circumference_of_circle(radius)

```

```

    radius * 2 * Pi
  end
end

puts OtherPlanet.circumference_of_circle(10)

```

```
90.0
```

```
puts OtherPlanet::Pi
```

```
4.5
```

```
puts Pi
```

```
3.141592
```

通过本例可以看出，常量的作用域是在类的环境中。OtherPlanet类有其自己的Pi定义。但如果其中未对Pi重新定义，则原来的Pi对OtherPlanet生效，因为OtherPlanet类是定义在全局的作用域内。

从上例的第二部分还可看出，可以直接查询其他类中的常量。OtherPlanet::Pi即直接引用OtherPlanet中的Pi常量。

6.3 模块、命名空间和掺入

模块提供了一种结构，用来把Ruby类、方法和常量收集到单独命名和定义的单元中。这样非常有用，可以避免与现有类、方法和常量发生冲突，而且还可以把模块的功能增加（掺入）到自己的类中。首先，我们来看一下怎样用模块来创建命名空间（namespace），以避免与名字相关的冲突。

6.3.1 命名空间

Ruby中经常使用的一种功能是把其他文件中代码包含到当前程序的能力（下一章对此有更深入的讨论）。当包含其他文件时，你很快就会碰到冲突，特别是当包含的文件或程序库又包含了多个文件时。你无法保证所包含的文件中（或在includes长长的包含链所包含的文件中），没有文件会与你已编写或已处理的代码发生冲突。

我们来看下面这个例子：

```

def random
  rand(1000000)
end

puts random

```

random方法返回0到999,999之间的一个随机数。该方法可能位于某个很容易遗忘的外部文件中，这样一来，如果用require包含的另一个文件中实现了如下所示的同名方法，就会发生问题。

```
def random
  (rand(26) + 65).chr
end
```

这个random方法返回随机的大写字符。

注 (rand(26) + 65).chr生成0到25之间的一个随机数字，再加上65，得到65到90范围内的数字。然后用chr方法将该数字转换成ASCII标准的字符，在该标准中，65代表A，到90代表Z。你可以访问<http://en.wikipedia.org/wiki/ASCII>网页，学习ASCII字符集的更多内容，或参阅第3章中对此主题的更详细讨论。

现在你有两个方法都叫random。如果第一个random方法位于名为number_stuff.rb文件中，第二个random方法位于名为letter_stuff.rb文件中，你将会碰到麻烦：

```
require 'number_stuff'
require 'letter_stuff'
```

```
puts random
```

究竟是哪个版本的random方法被调用？

注 require是Ruby语句，用来载入其他文件包含的代码。这方面的内容将在下一章详细介绍。

由于最后一个文件被载入，因此结果是后一个版本的random方法被调用，屏幕上显示一个随机的字母。不幸的是，这表示你的另一个random方法丢失了。

这一情况被称为命名冲突 (name conflict)，它甚至可能发生在比上面代码所示的简单示例更可怕的情况中。例如，类名可能同样发生冲突，你会无意中把两个同名类混杂在一起。如果名为Song的类定义在某个外部文件中，然后在另一个外部文件中也定义同名的类，那么在你的程序中，Song类将会是二者乱七八糟的混合体。有时可能有意这么做，但在其他情况下将导致一定的麻烦。

模块有助于解决这些冲突，它提供了命名空间，可以包含任何数量的类、方法和常量，并允许直接引用它们。例如：

```
module NumberStuff
  def NumberStuff.random
    rand(1000000)
  end
end

module LetterStuff
  def LetterStuff.random
    (rand(26) + 65).chr
  end
end
```

```
puts NumberStuff.random
puts LetterStuff.random
```

```
184783
X
```

注 由于所使用的rand方法带来的随机性，导致上述结果将因每次运行而不同。

在本例的最后两行代码中，你打算使用哪个版本的random，可以看得一清二楚。上述代码中定义的模块，看起来像是类，唯一区别在于定义用词为module而非class。但实际上，你无法定义模块的实例，因为它们不是真正的类，而且它们也不能从任何事物继承。模块只提供了把方法、类和常量组织到单独命名空间的途径。

下面是个更复杂的例子，涉及不同模块中的两个同名类：

```
module ToolBox
  class Ruler
    attr_accessor :length
  end
end

module Country
  class Ruler
    attr_accessor :name
  end
end

a = ToolBox::Ruler.new
a.length = 50
b = Country::Ruler.new
b.name = "Ghengis Khan from Moskau"
```

这里并没有发生两个Ruler（尺子）类争夺控制权，或最终形成一个变异的Ruler类，例如，既包含name属性也包含length属性的变异Ruler类（有多少计量用的尺子有名字？），而是把两个Ruler类分别放在ToolBox和Country命名空间中。

后面你将看到，为什么命名空间比这种情况更有用，那么首先你得了解模块如此有用的第二个原因。

6.3.2 掺入

在前文中，我们研究了继承：它是面向对象的一个功能特性，允许类（及其实例对象）从其他类中继承方法。你会发现，Ruby不支持多重继承（multiple inheritance），即同时从多个类中继承的能力。Ruby的继承功能只允许你创建简单的类树，避免了多重继承系统天生的混乱。

不过，在某些情况下，从迥然不同的类中共享功能，也是件很有用的事。在这种意义上，模块像某种“超级”类，可被包含到其他类中，用模块提供的功能来扩展其他类。例如：

```

module UsefulFeatures
  def class_name
    self.class.to_s
  end
end

class Person
  include UsefulFeatures
end

x = Person.new
puts x.class_name

```

```
Person
```

在这段代码中，UsefulFeatures模块看起来几乎就是个类，而且确实几乎就是。不过，模块本身是组织工具，而不是类。class_name方法位于模块中，因此随即被包含到Person类中。下面是另一个例子：

```

module AnotherModule
  def do_stuff
    puts "This is a test"
  end
end

include AnotherModule
do_stuff

```

```
This is a test
```

如你所见，即使不是直接在类中，也可以把模块的方法包含到当前作用域中。不过，你可以直接调用方法，此时有点像类：

```
AnotherModule.do_stuff
```

因此，include命令是把模块的内容放到当前作用域中。

Ruby自带了几个标准模块供你使用。例如，Kernel模块包含所有“标准”命令，在Ruby中使用时无须指定对象或类，例如load、require、exit、puts和eval。这些方法没有一个是直接在对象作用域内起效的（像自己程序的方法一样），但它们是特殊方法，通过Kernel模块，被默认包含所有类（包括主程序作用域）中。

不过，我们更感兴趣的是Ruby提供的、可以包含在自己程序中以立即获得更多功能的模块。这里介绍其中的两个模块，分别是Enumerable和Comparable。

Enumerable模块

在上一章，进行过迭代过程的操作，例如：

```
[1,2,3,4,5].each { |number| puts number }
```

在本例中，你创建了一个临时数组，其中包含1到5的数字，并用each迭代子，把数组的每

个值传递给代码块，代码块将每个值赋给number变量，然后用puts命令将其打印输出到屏幕。

each迭代子给你带来了巨大的能力，因为它让你遍历数组或散列表的所有元素，并用检索的数据算出数字数组的平均值，或数组中字符串的最长长度等。如下所示：

```
my_array = %w{this is a test of the longest word check}
longest_word = ''
my_array.each do |word|
  longest_word = word if longest_word.length < word.length
end
puts longest_word
```

longest

在本例中，循环遍历my_array，如果当前已知的最长单词比word的长度更短，则将其赋值给longest_word。当循环结束时，最长的单词存放在longest_word中。

相同的代码可以调整一下，来找出一组数字中最大（或最小）的数字：

```
my_array = %w{10 56 92 3 49 588 18}
highest_number = 0
my_array.each do |number|
  number = number.to_i
  highest_number = number if number > highest_number
end
puts highest_number
```

588

不过，Array类预先包含了Enumerable模块的方法，这个模块中提供了大约20个有用的统计和迭代相关的方法，包括collect、detect、find、find_all、include?、max、min、select、sort和to_a。所有这些方法都使用了Array类的each方法来完成其工作，如果你的类要实现each方法，则可以包含Enumerable模块，即可在你自己的类中，免费得到所有这些方法！

注 Enumerable模块提供的主要方法请参见附录B。

首先介绍Enumerable模块提供的一些方法示例：

```
[1,2,3,4].collect { |i| i.to_s + "x" }
```

```
=> ["1x", "2x", "3x", "4x"]
```

```
[1,2,3,4].detect { |i| i.between?(2,3) }
```

```
=> 2
```

```
[1,2,3,4].select { |i| i.between?(2,3) }
```

```
=> [2,3]
```

```
[4,1,3,2].sort
```

```
=> [1,2,3,4]
```

```
[1,2,3,4].max
```

```
=> 4
```

```
[1,2,3,4].min
```

```
=> 1
```

你可以创建自己的类，实现each方法，并“免费”得到这些方法：

```
class AllVowels
  @@vowels = %w{a e i o u}
  def each
    @@vowels.each { |v| yield v }
  end
end
```

实际上，这个类不需要提供多个对象，因为它只对元音字母进行枚举。不过，为了保持示例的简单性，这是个理想的例子。下面是其调用方法：

```
x = AllVowels.new
x.each { |v| puts v }
```

```
a
e
i
o
u
```

AllVowels类包含一个类变量数组，其中包含元音字母。用对象实例级的each方法对类变量数组@@vowels进行循环迭代，并让渡给each关联的代码块，用yield向代码块传递每个元音字母。我们让Enumerable模块也参与其中：

```
class AllVowels
  include Enumerable

  @@vowels = %w{a e i o u}
  def each
    @@vowels.each { |v| yield v }
  end
end
```

注 yield及其与代码块的关系，如果你需要刷新记忆，可以参阅第3章接近结束的地方的讨论。

现在我们再次试着使用Enumerable模块提供的方法。首先我们得到AllVowels对象：

```
x = AllVowels.new
```

现在你可以对x调用这些方法：

```
x.collect { |i| i + "x" }
```

```
=> ["ax", "ex", "ix", "ox", "ux"]
```

```
x.detect { |i| i > "j" }
```

```
=> "o"
```

```
x.select { |i| i > "j" }
```

```
=> ["o", "u"]
```

```
x.sort
```

```
=> ["a", "e", "i", "o", "u"]
```

```
x.max
```

```
=> "u"
```

```
x.min
```

```
=> "a"
```

Comparable模块

Comparable模块提供了很多方法，为其他类提供了比较运算符，例如<（小于）、<=（小于或等于）、==（等于）、>=（大于或等于）和>（大于），还有between?方法，当某值在指定的两个参数之间（包含参数值本身）时，则返回true（例如4.between?(3,10) == true）。

为了提供这些方法，对于包含Comparable模块的类，Comparable模块对该类使用了<=>运算符。如果对象的值小于指定参数，则<=>返回-1；如果等于，则返回0；如果大于，则返回1。例如：

```
1 <=> 2
```

```
-1
```

```
1 <=> 1
```

```
0
```

```
2 <=> 1
```

```
1
```

用<=>这个简单的方法，Comparable模块可以提供其他基本比较运算符和between?方法。你可以创建自己的类，并试一下：

```

class Song
  include Comparable

  attr_accessor :length
  def <=>(other)
    @length <=> other.length
  end

  def initialize(song_name, length)
    @song_name = song_name
    @length = length
  end
end

a = Song.new('Rock around the clock', 143)
b = Song.new('Bohemian Rhapsody', 544)
c = Song.new('Minute Waltz', 60)

```

下面是包含了Comparable模块的结果:

```
a < b
```

```
=> true
```

```
b >= c
```

```
=> true
```

```
c > a
```

```
=> false
```

```
a.between?(b,c)
```

```
=> true
```

你可以像比较数字一样比较歌曲。从技术上来说，确实是这么比较的。通过在Song类中实现<=>方法，具体的歌曲对象可以直接比较，用其长度进行比较。如果你喜欢，可以把<=>实现为按歌曲名字的长度或其他任何属性进行比较。

模块也有同样的能力，可以实现类似的通用功能集，然后可以将其应用于任何类。例如，可以创建一个模块，实现longest和shortest方法，返回列表中最长或最短的字符串，这两个方法可被Array、Hash或其他任何类包含。

通过命名空间和类进行掺入

在前文示例中，我们演示了怎样使用模块来定义命名空间，代码如下：

```

module ToolBox
  class Ruler
    attr_accessor :length

```

```

end
end

module Country
  class Ruler
    attr_accessor :name
  end
end

a = ToolBox::Ruler.new
a.length = 50
b = Country::Ruler.new
b.name = "Ghengis Khan of Moskau"

```

在这一情况下，Ruler类可通过相应的模块直接访问（例如ToolBox::Ruler和Country::Ruler）。

但如果你想临时假设Ruler（没有模块名前缀）就是Country::Ruler，那么要访问任何其他Ruler类就得直接引用，此时应该怎么办？include命令可以实现这一效果。

在前文章节中，你已经用过include命令，来把其他模块中的方法包含到当前类和访问作用域中，但除此之外，它同时也包含了模块中的类（如果有的话），并将这些类设为本地可访问。我们假定，在上面的代码之后，你又做了如下处理：

```

include Country
c = Ruler.new
c.name = "King Henry VIII"

```

成功了！Country模块的内容（在本例中，只有Ruler类）被带入当前访问作用域，还可以把Ruler当成本地类来使用。如果想使用ToolBox中的Ruler类，仍可以直接用ToolBox::Ruler来引用。

6.4 用对象构建“地下城”文本冒险游戏

到目前为止，本章带你深入探讨了面向对象的概念（主要从技术层面）。在此基础上，我们再把它应用到真实世界的场景中，这样做有利于扩展我们的知识。

在本节中，你将实现一个小型的文本冒险游戏/虚拟地下城。文本冒险游戏在1980年代很流行，但由于现代图形游戏的兴起，人们很快对它失去兴趣。不过，文本冒险游戏是用来体验类和对象的完美游乐场，因为以虚拟形式来复制真实世界，需要对真实世界概念与类的映射有透彻的理解。

6.4.1 地下城的概念

在进行类的开发之前，需要弄明白要对什么建模。地下城游戏一点也不复杂，但至少还要处理以下概念：

- **地下城**：需要一个一般类，用来封装地下城游戏的完整概念。
- **玩家**：玩家提供你与地下城之间的连接，地下城游戏的所有体验都由玩家而来。玩家可以

在地下城的各个房间之间来回移动。

- **房间**：地下城的房间供玩家在其间移动。各房间之间以多种形式连接（例如，北门、西门、东门、南门），并有相应描述。

完整的冒险游戏还应该有物品、敌人、其他角色、路点、咒语，以及各种谜题和结果的触发事件。以后如果你愿意，可以轻易地把你的开发成果扩展为更完整的游戏。

6.4.2 创建初始类

我们要开发的第一个概念，是地下城和游戏本身。在这一框架内，再引入其他概念，例如玩家和房间。

使用嵌套类，初始代码布局如下：

```
class Dungeon
  attr_accessor :player
  def initialize(player_name)
    @player = Player.new(player_name)
    @rooms = []
  end

  class Player
    attr_accessor :name, :location

    def initialize(player_name)
      @name = player_name
    end
  end

  class Room
    attr_accessor :reference, :name, :description, :connections

    def initialize(reference, name, description, connections)
      @reference = reference
      @name = name
      @description = description
      @connections = connections
    end
  end
end
```

这段代码给出了整个地下城游戏的框架。Dungeon（地下城）类封装了所有其他类，这一核心概念把所有东西都绑定在一起，因为在此情况下，如果没有Dungeon的包裹，Player（玩家）和Room（房间）类就没有任何用处。这并不是说，依赖于其他类的类必须嵌套，只是在此情况下，用这种方式来组织类是有道理的。

现在，地下城游戏有几个实例变量，分别保存玩家和房间的清单（@rooms = []创建了空Array，它与@rooms = Array.new同义）。

Player类让玩家对象可以跟踪自己的名字和当前位置。Room类让房间对象可以保存其名字、说明（例如，“刑讯室”和“这是一个黑暗的、让人不寒而栗的房间”），以及与其他房间的连接方式，以及相应的引用（供其他房间使用，得到双方的连接）。

当用Dungeon.new创建地下城时，需要提供玩家的名字，以便创建玩家，并将地下城实例变量赋给@player。这是因为玩家要和地下城连接在一起，因此在dungeon对象中保存player对象是有道理的。由于通过attr_accessor，player变量已经被设置为可读写，因此你可以很容易地访问玩家对象。例如：

```
my_dungeon = Dungeon.new("Fred Bloggs")
puts my_dungeon.player.name
```

Fred Bloggs

通过穿越地下城对象，可以直接访问玩家的功能。因为@player包含player对象，而且通过attr_accessor :player，@player已经被设置为公共可访问的，因此你得到了完全的访问权限。

6.4.3 Structs：快捷简单的数据类

目前列出的主代码中，可以很明显地看出一种现象：重复。Room和Player类只是作为基本的数据占位区，而不是具有逻辑和功能的真正类。在Ruby中有一种更简单的方法，只用一行名为Struct（结构）的类代码，即可创建这类专用于保存数据的特殊类。

结构是一种特殊类，其唯一职能就是拥有属性，保存数据。示例如下：

```
Person = Struct.new(:name, :gender, :age)
fred = Person.new("Fred", "male", 50)
chris = Person.new("Chris", "male", 25)
puts fred.age + chris.age
```

75

简单地说，Struct类的用途是构建存储数据的类。第一行代码创建了名为Person的新类，它有内置的姓名、性别和年龄属性。第二行代码创建了Person类的新实例，并在运行过程中设置属性。第一行代码等同于以下这段冗长的内容：

```
class Person
  attr_accessor :name, :gender, :age

  def initialize(name, gender, age)
    @name = name
    @gender = gender
    @age = age
  end
end
```

注 实际上，这段代码不完全等同于struct代码，因为在初始化Struct类时，参数是

可选的，而上述Person类代码需要提供三个参数（name、gender和age）。

这段代码创建Person类的方式比较冗长。如果只需要存储数据，那么struct技术更快捷更易读，当然，如果最终要向这个类增加更多功能，以冗长方式创建类就是值得的。不过，可以在开始时使用struct方式，然后在需要时再重新改写为完整的类，这是个好消息。这就是在开发地下城游戏时要采用的方式。我们来重写这段代码：

```
class Dungeon
  attr_accessor :player

  def initialize(player_name)
    @player = Player.new(player_name)
    @rooms = []
  end

  Player = Struct.new(:name, :location)
  Room = Struct.new(:reference, :name, :description, :connections)
end
```

这样简短多了，而且由于在创建Struct类的实例时，参数是可选的，因此仍然可以用Player.new(player_name)的方式，location属性只被设为nil。如果需要向Player或Room中增加方法，可以将其重写为普通类，并用attr_accessor把属性加回来。

属性读写权限设置器 (attr_accessor)

和第2章一样，在本章的代码中，类中使用了attr_accessor来为对象提供属性。用attr_accessor可以这样做：

```
class Person
  attr_accessor :name, :age
end

x = Person.new
x.name = "Fred"
x.age = 10
puts x.name, x.age
```

但实际上，attr_accessor并没有做什么魔术处理，它只是帮你写了一些代码。以下这段代码等同于上面Person类中的单行代码attr_accessor :name, :age:

```
class Person
  def name
    @name
  end

  def name=(name)
    @name = name
  end
end
```

```

def age
  @age
end

def age=(age)
  @age = age
end
end

```

这段代码定义了name和age方法，用来返回当前对象变量的相应属性，因此可以调用x.name和x.age（和前面的代码一样）。这段代码还定义了两个“写属性方法”，为@name和@age对象变量赋值。

如果你注意写属性方法的名字，会发现它们与读属性方法的名字相同，只是加了等于号(=)后缀。这表示可以用诸如x.name = "Fred"和x.age = 10这样的代码。在Ruby中，赋值正是对常规方法的调用！事实上，x.name = "Fred"只是x.name= ("Fred")的缩写。

6.4.4 创建房间

地下城游戏现在已有两个基本类，但还没有办法创建房间，因此我们来向Dungeon类增加方法：

```

class Dungeon
  def add_room(reference, name, description, connections)
    @rooms << Room.new(reference, name, description, connections)
  end
end

```

我们想做的是向地下城增加房间，因此向地下城对象增加方法是最合理的。这样一来，可以用下面的方法来创建房间（当然，是在my_dungeon已经定义好的前提下）：

```

my_dungeon.add_room(:largecave, "Large Cave", "a large cavernous cave", {
  :west => :smallcave })

my_dungeon.add_room(:smallcave, "Small Cave", "a small, claustrophobic cave", {
  :east => :largecave })

```

add_room方法接受reference（引用）、name（名字）、description（说明）和connections（连接）参数，用它们来创建新的Room对象，然后把这个对象放到@rooms数组的末尾。

reference、name和description参数很容易看明白，但connections参数则不然，它接受一个散列表，其中内容是某个特定房间与其他房间的连接。例如，{ :west => :smallcave }把两个符号(:west和:smallcave)连接到一起。地下城代码通过该链接把房间连接起来，根据{ :west => :smallcave, :south => :another_room }这样的连接散列表，将创建两个连接（一个指向西方，一个指向南方）。

6.4.5 让地下城运转起来

现在已经把所有房间载入到基本的地下城中了（并且只要愿意，可以用`add_room`方法增加更多的房间），但还没有办法启动地下城游戏本身。

第一步是在Dungeon中创建一个方法，把用户放到地下城中，并给出初始位置的说明，从而“启动”整个游戏：

```
class Dungeon
  def start(location)
    @player.location = location
    show_current_description
  end

  def show_current_description
    puts find_room_in_dungeon(@player.location).full_description
  end

  def find_room_in_dungeon(reference)
    @rooms.detect {|room| room.reference == reference }
  end

  class Room
    def full_description
      @name + "\n \nYou are in " + @description
    end
  end
end
```

在上面的代码中，在地下城类中定义了`start`方法，先设置玩家的`location`属性，然后调用地下城的`show_current_description`方法，后者是根据玩家的位置，找到当前所在房间，然后在屏幕上打印输出该位置的完整说明。`full_description`方法可以完成这一工作，接受位置名和说明，并将其转换成完整的、有用的说明。`find_room_in_dungeon`方法则正相反，它在`@rooms`数组中循环遍历，找出引用值匹配当前位置的房间。

但上述代码的问题是，`Room`是个struct结构，而不是完整的类，因此必须再次把它转换成完整类（如前文提示的那样）。这一变化需要一些关键修改，因此为了保持简洁，下面列出到目前为止的完整代码，以及把`Room`变为常规类的变化内容，和用于辅助地下城漫游的一些额外方法：

```
class Dungeon
  attr_accessor :player

  def initialize(player_name)
    @player = Player.new(player_name)
    @rooms = []
  end
```



```
def add_room(reference,name,description,connections)
  @rooms <<Room.new(reference,name,description,connections)
end

def start(location)
  @player.location =location
  show_current_description
end

def show_current_description
  puts find_room_in_dungeon(@player.location).full_description
end

def find_room_in_dungeon(reference)
  @rooms.detect {|room|room.reference ==reference }
end

def find_room_in_direction(direction)
  find_room_in_dungeon(@player.location).connections [direction ]
end

def go(direction)
  puts "You go "+direction.to_s
  @player.location =find_room_in_direction(direction)
  show_current_description
end

class Player
  attr_accessor :name,:location

  def initialize(name)
    @name =name
  end
end

class Room
  attr_accessor :reference,:name,:description,:connections

  def initialize(reference,name,description,connections)
    @reference =reference
    @name =name
    @description =description
    @connections =connections
  end

  def full_description
    @name +"\\n\\nYou are in "+@description
  end
end
```

```

    end
  end

end

#Create the main dungeon object
my_dungeon =Dungeon.new("Fred Bloggs")

#Add rooms to the dungeon
my_dungeon.add_room(:largecave,"Large Cave","a large cavernous cave",{
:west =>:smallcave })
my_dungeon.add_room(:smallcave,"Small Cave","a small,claustrophobic cave",{
:east =>:largecave })

#Start the dungeon by placing the player in the large cave
my_dungeon.start(:largecave)

```

Large Cave

You are in a large cavernous cave

这是一段很长的源代码，但其中大部分都能看明白。你已经再次把Room和Player改为真正的类，并实现了地下城游戏的基本功能。

如下两种特别有用的方法已经加入到Dungeon类中：

```

def find_room_in_direction(direction)
  find_room_in_dungeon(@player.location).connections [direction ]
end

def go(direction)
  puts "You go "+direction.to_s
  @player.location =find_room_in_direction(direction)
  show_current_description
end

```

go方法是实现地下城漫游的方法，它接受单个参数（漫游的方向）并用它把玩家的位置改变为那个方向的房间。为了实现这一功能，它调用了find_room_in_direction方法，该方法接受一个引用参数（该参数是当前房间在相应方向上的连接），并返回目标房间的引用。回忆一下，你定义了这样一个房间：

```

my_dungeon.add_room(:largecave,"Large Cave","a large cavernous cave",{
:west =>:smallcave })

```

如果当前房间是:largecave，那么find_room_in_direction(:west)将使用该房间的连接返回:smallcave，这一结果随即被赋值给@player.location，将其定义为新的当前位置。

为了检验地下城的漫游功能，如果你使用的是irb，可以直接输入go命令；如果你使用编辑器处理源代码文件，则需要把go命令加到源代码文件末尾，并重新运行。下面是运行结果：

```
my_dungeon.show_current_description
```

```
Large Cave
```

```
You are in a large cavernous cave
```

```
my_dungeon.go(:west)
```

```
You go west
```

```
Small Cave
```

```
You are in a small, claustrophobic cave
```

```
my_dungeon.go(:east)
```

```
You go east
```

```
Large Cave
```

```
You are in a large cavernous cave
```

这段代码没有任何出错检查（用`my_dungeon.go(:south)`试图进入不存在的房间），并缺少物品、仓库和文本冒险游戏的其他基本功能，但现在你已拥有一组表示地下城的可操作对象，并可以用基本的方式来漫游。

这段代码已经足够成熟，可供进一步扩展和操作。用另一个类和几个更多方法，你可以很容易地为游戏提供物品的支持，这些物品可放在不同的位置，可供拾取，并丢放在其他位置。

在第9章你将了解怎样与文件交互，和从键盘读取数据。到那时，你可以把地下城游戏扩展成交互式的，能够接受用户输入，验证方向是否合法（如合法则调用`go`方法）。有了这些扩展功能，并增加更多的房间，你就完成了可行的文本冒险游戏的大部分内容！

6.5 小结

本章讨论了面向对象的基础知识，以及Ruby为实现面向对象编码所提供的功能特性。我们考察了大多数语言所用的面向对象概念，例如继承、封装、类方法、实例方法，以及可用变量的各种类型。最后我们开发了一套基本的类，实现一个简单的地下城游戏。

我们来回顾一下本章涵盖的一些基本概念：

- 类：类是一些方法和数据的集合，用作创建与该类相关的多个对象的蓝图。
- 对象：对象是类的单个实例。`Person`类的对象是一个人，`Dog`类的对象是一条狗。如果把对象看成是实际生活中的物品，那么类就是物品的类别，而对象则是实际物品或是“东西”本身。
- 局部变量：只能在当前作用域内访问和使用的变量。
- 实例/对象变量：能在单个对象的作用域内访问和使用的变量。对象的所有方法均可访问对象的实例变量。
- 全局变量：能在当前程序任何位置访问和使用的变量。

- 类变量：能在类及其所有子对象的作用域内访问和使用的变量。
- 封装：这一概念允许方法在类或对象之外有不同程度的可见性。
- 多态：这一概念让方法可以处理不同数据类型，并提供更一般化的实现（例如Square类和Triangle类所提供的area和perimeter方法）。
- 模块：是一种组织元素，能把任意数量的类、方法和常量集合到单个命名空间之内。
- 命名空间：是一种组织命名元素，让类、方法和常量避免发生冲突。
- 掺入：是一种模块，可以将方法掺入到其他类中，以扩充该类功能。
- Enumerable模块：是一个掺入模块，为诸如collect、map、min和max等其他类提供了迭代子和列表相关方法，作为Ruby的标准实现。Ruby在Array和Hash类默认使用这一模块。
- Comparable模块：是一个掺入模块，为实现一般比较运算符<=>的类，提供了比较运算符（例如<、>和==），作为Ruby的标准实现。

在后续几章中，我将假定你已经了解类和对象的工作原理，以及不同的变量作用域（包括局部、全局、对象和类变量）的工作原理。



第7章 项目与程序库

在前面几章，我们从底层角度对Ruby进行考察，直接操作了类、对象和函数。到目前为止，我们在小项目中所用的每一行代码都是从零开始专门为该项目编写的。在本章中，我们将考察怎样用Ruby构建更大型的项目，以及怎样重用以前编写的代码。最后，我们将考察怎样在自己的应用程序中，使用由其他开发人员编写并打包的代码，从而无须在每次开发新程序时，都要重新发明轮子。

本章将介绍Ruby更广泛的应用：处理项目和程序库。

7.1 项目和使用其他文件的代码

随着你对Ruby越来越熟悉，并发现它越来越多的用途，也许你会不再想编写单个小程序（大约少于100行），而希望开发更复杂的、由多个部件组成的程序和系统。而更大型的程序和系统一般称为项目，其管理方式与单个文件的简单脚本有根本不同。

在Ruby中，进行功能划分的最常用办法，是将其放到不同文件的不同的类中。这样就可以编写用于多个项目的类，并且只需把相关文件拷贝到其他项目中即可。

7.1.1 基本的文件包含

我们来看下面这行代码：

```
puts "This is a test".vowels.join('-')
```

如果执行这行代码，将会碰到错误，提示对于String类的"This is a test"对象来说，`vowels`方法不可用。这样的结果是正确的，因为Ruby没有提供该方法。我们可以编写String类的扩展，来提供这一方法：

```
class String
  def vowels
    self.scan(/[aeiou]/i)
  end
end
```

如果这段代码与上面的`puts`代码在同一个文件中，将会产生如下的运行结果：

```
i-i-a-e
```

在此情况下，用`vowels`方法扩展了String类，该方法使用`scan`来返回元音字母的数组（正则表达式末尾的`i`选项令其忽略大小写）。

但你可能想编写一系列方法来扩展String类，以便用于多个程序中。因此，与其每次都拷贝和粘贴代码，不如将这些代码拷贝到单独的文件中，并用`require`命令把外部文件载入到当前程序中。例如，把这些代码放到名为`string_extensions.rb`的文件中：


```
class String
  def vowels
    self.scan(/[aeiou]/i)
  end
end
```

并把以下代码放到名为vowel_test.rb的文件中：

```
require 'string_extensions'
puts "This is a test".vowels.join('-')
```

如果运行vowel_test.rb，屏幕上将会显示期待的结果。第一行代码require 'string_extensions'的功能很简单，只是载入string_extensions.rb文件，并把其中的代码视为本地代码来处理。也就是说，在此情况下，vowels方法可以使用，这一切都由一行require代码实现。

也可以用load命令来载入外部源代码文件，其功能与require相同。例如，以下代码与上面代码的功能完全相同：

```
load 'string_extensions'
puts "This is a test".vowels.join('-')
```

二者采用了相同的处理方式，不过我们再试一下另外的例子，把以下代码放到a.rb文件中：

```
puts "Hello from a.rb"
```

再把以下代码放到b.rb文件中：

```
require 'a'
puts "Hello from b.rb"
require 'a'
puts "Hello again from b.rb"
```

运行ruby b.rb命令，得到以下结果：

```
Hello from a.rb
Hello from b.rb
Hello again from b.rb
```

在本例中，a.rb文件只被包含一次。它在第1行代码中，因此“Hello from a.rb”显示在屏幕上，但在b.rb的第3行再次被包含时，什么事也没有发生。与之相反的是：

```
load 'a'
puts "Hello from b.rb"
load 'a'
puts "Hello again from b.rb"
```

```
Hello from a.rb
Hello from b.rb
Hello from a.rb
Hello again from b.rb
```

用load命令，代码每次在载入时都被重新处理，而require命令则不然，只对外部代码处

理一次。

注 Ruby程序员一般使用`require`而非`load`。`load`的效果仅当外部文件的代码有变更时才有用，或外部文件包含的是活动代码，需要立即执行。不过，优秀的程序员会避免后者的情况，而且外部文件只包含类和模块，其内容一般来说极少发生变化。

7.1.2 从其他目录包含

`load`和`require`命令都可以接受本地路径或绝对路径的文件名。例如，`require 'a'`命令会首先在当前目录寻找`a.rb`文件，然后在硬盘的许多其他目录中寻找。默认情况下，这些其他目录是Ruby保存自有文件和程序库的不同目录。当然，如果有必要，你可以覆写这些默认设置。

Ruby把被包含文件的搜索路径（目录列表）保存在名为`$:`的特殊变量中。可以用`irb`查看`$:`包含的默认值：

```
$:.each { |d| puts d }
```

```
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-darwin8.8.1
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-darwin8.8.1
```

注 这一结果是在我的机器上（Mac OS X）运行的结果，与你机器上列出的目录可能有相当的差异，特别是如果你使用的是Windows平台，其路径布局完全不同，开头是盘符，而且用反斜杠，而不是正斜杠。

如果想增加额外的目录，方法很简单：

```
$:.push '/your/directory/here'
require 'yourfile'
```

`$:`是个数组，因此可以向其中增加额外的元素，或用`unshift`把元素加到列表开头（用于自己的目录在Ruby默认目录之前优先搜索的情况——如果想覆写Ruby标准程序库的话，这一点很有用）。

注 Ruby用文件名来跟踪`include`所包含的文件。如果同一个文件有两个路径，并用两个唯一的全路径名来包含它们，则Ruby将适时地把同一文件载入两次。

7.1.3 有条件地包含代码

在Ruby程序中，`require`和`load`命令和常规代码的效果一样，可以将其放在Ruby代码的任何位置，它们的行为表现似乎就是在那里被执行。例如：

```
$debug_mode = 0
```

```
require $debug_mode == 0 ? "normal-classes" : "debug-classes"
```

这个例子比较费解，它的功能是检查全局变量`$debug_mode`是否被设置为0。如果为0，则载入`normal-classes.rb`；如果不为0，则载入`debug-classes.rb`。这样一来，可以根据变量的值，来包含不同的源代码文件，一般用于应用程序是“常规”或“调试”模式的情况。你甚至可以编写这样的应用程序，一般情况下它的功能很完善，但一旦通过另外的`require`载入完全不同的一组文件时，就拥有了全新的或试验性的功能。

有个常用的快捷方式，是利用数组同时快速载入一组程序库。例如：

```
%w{file1 file2 file3 file4 file5}.each { |l| require l }
```

仅用两行代码，即可载入五个不同的外部文件或程序库。不过，有些程序员对此种风格并不热心，因为尽管更有效率，但这种风格让代码更难阅读。

7.1.4 嵌套包含

用`require`和`load`命令从其他文件中载入的代码，拥有与本地代码一样的自由度，仿佛它们就是粘贴到原文件中的。这表示载入的代码本身也可以调用`load`和`require`命令。例如，假设`a.rb`包含以下内容：

```
require 'b'
```

而`b.rb`又包含以下内容：

```
require 'c'
```

而`c.rb`则包含以下内容：

```
def example
  puts "Hello!"
end
```

而`d.rb`又包含以下内容：

```
require 'a'
example
```

```
Hello!
```

`d.rb`用`require`命令包含`a.rb`，`a.rb`包含`b.rb`，而`b.rb`又包含`c.rb`，这表示`example`方法对`d.rb`是可以访问的。

利用这种功能，可以很容易地用相互依赖的部件组成大型项目，其嵌套深度随你的意愿而定。

7.2 程序库

在计算机编程中，程序库（library）是指可被其他程序调用、且独立存在的例程集合。例如，你可以创建一个程序库，用来载入并处理数据文件，然后在无数其他程序中调用该程序库的例程。

在本章前半部分，我们考察了怎样用`require`命令把外部文件载入到你的Ruby程序中，然后考察了怎样用模块把功能元素划分到独立的命名空间中。在Ruby中创建程序库时，你可以联

合使用这两种概念。

在本章开头，我们开发了一个极其简单的程序库，名为`string_extensions.rb`，内容如下：

```
class String
  def vowels
    self.scan(/[aeiou]/i)
  end
end
```

然后用以下代码调用这个程序库：

```
require 'string_extensions'
puts "This is a test".vowels.join('-')
```

```
i-i-a-e
```

几乎所有程序库都比这个要复杂得多，但不管怎样，本例演示了程序库的基本原理。

下面我们将考察Ruby自带的标准程序库，并了解怎样下载和使用因特网上其他开发人员制作的程序库。

7.2.1 标准程序库

Ruby自带100个以上的程序库，作为标准程序库。这些程序库提供了“开箱即用”的功能，为Ruby提供了广泛的选择，包括从Web服务、网络工具，到数据加密、性能基准度量以及测试例程。

注 “标准程序库集”常常称为“标准库”，当你看到这一名词（它在第16章用得特别多），请务必记得，这是指一组程序库，而不是某个特定的程序库。

在本节中，我们只随机考察两个标准程序库（`net/http`和`OpenStruct`），以便让你熟悉相关使用方法，从而在后续章节中能够对其他程序库运用自如（其使用方法都是类似的）。

标准程序库的全部清单（含文档），可访问<http://www.ruby-doc.org/stdlib/>网页，其中相当数量的程序库在本书第16章有详细讲解。

注 有些用户可能会发现，Ruby自带的标准程序库数量可能有所减少，特别是在使用预装版本Ruby的情况下。不过，如果从源代码安装Ruby，则本节介绍的所有内容都能使用。

net/http程序库

HTTP是“超文本传输协议（HyperText Transfer Protocol）”的缩写，它是WWW万维网得以运转的主要协议，提供了网页、文件和其他媒体在Web服务器和客户端之间传输的机制。

Ruby通过`net/http`程序库，提供对HTTP的基本支持。例如，用来下载和打印特定网页的Ruby脚本，写起来非常容易：

```
require 'net/http'
Net::HTTP.get_print('www.rubyinside.com', '/')
```

如果运行这段代码，几秒钟之后，屏幕上就会飞速显示许多页的HTML代码。第一行代码的作用把net/http程序库载入到当前程序中，第二行代码则调用Net::HTTP类的类方法（这里Net是模块名，定义了Net命名空间，而HTTP是个子类），获取并打印（此方法取名为get_print）位于http://www.rubyinside.com/的网页内容。

把网页内容放到字符串中也很容易，以便稍后在程序中进一步处理：

```
require 'net/http'
url = URI.parse('http://www.rubyinside.com/')
response = Net::HTTP.start(url.host, url.port) do |http|
  http.get(url.path)
end
content = response.body
```

本例中，使用了URI程序库（这是另一个标准库，由net/http自动载入），对诸如http://www.rubyinside.com/等URL进行解析，将其分解各个组成部分，以便net/http程序库用来发出请求。一旦URL解析完成，即“启动”一个HTTP连接，在此连接的作用域内，用get方法发出GET请求（如果你看不懂，没关系，这是HTTP协议的部分工作原理）。最后，读取response.body这个字符串的内容，其中包含http://www.rubyinside.com/网页的内容。

注 net/http库只是个基本程序库，它要求输入的内容是经过预处理的，如前例所示。而URI程序库则很适合于完成这一预处理工作。

在第14章中，我们将深入考察net/http及其姐妹程序库，例如net/pop和net/smtp。

OpenStruct程序库

在第6章中，你操作了特殊类型的数据结构，名为Struct。Struct它可以在运行期间创建小型数据操作类，例如：

```
Person = Struct.new(:name, :age)
me = Person.new("Fred Bloggs", 25)
me.age += 1
```

Struct提供了丰富的功能，让你可以创建简单类，但无须以冗长的方式进行类定义。

由ostruct程序库提供的OpenStruct类，让这一切变得更加容易。它允许你创建数据对象，无须指定属性，并可在运行期间创建属性：

```
require 'ostruct'
person = OpenStruct.new
person.name = "Fred Bloggs"
person.age = 25
```

person是个变量，指向OpenStruct类的一个对象，而OpenStruct允许你在运行期间调用任何你想要的属性。这很像散列表的工作方式，但采用了对象的表示符号。

正如其名字所暗示的那样，OpenStruct比Struct更灵活，但它是更难阅读的代码，无法一目了然地看出对象用了哪些属性，但用传统的struct则可在创建时看出属性的名字。

正如你所见，程序库的使用方法非常简单。在大多数情况下，只需用require载入相关类和方法，然后即可开始直接使用。不过，对于更复杂的场景，请继续往下读！

7.2.2 RubyGems包

RubyGems是用于Ruby程序和程序库的一套打包系统，它让开发人员可以把自己的Ruby程序库打包成一种易于维护和安装的形式。RubyGems使得管理同一套程序库的不同版本变得非常容易，并让你可以在命令行界面用一行命令完成安装。

每个单独打包的Ruby程序库（或应用程序）被称为宝石（gem）或Ruby宝石（RubyGem）。Gem有名字、版本号和相关说明。你可以使用命令行gem命令，管理自己机器上安装的gem包。

安装RubyGem系统

由于RubyGems不是Ruby官方版本的组成部分，因此在使用之前，必须先进行安装（或确认它已经安装）。但是，由于它是Ruby程序库打包系统的事实标准，因此安装起来非常容易。

Windows平台

如果你的Ruby是用第1章所说的“一键安装程序”安装的，RubyGems就已经安装好了，因此可以跳过以下内容，直接阅读“查找Gem包”小节。

但如果你用的是Windows平台上未安装RubyGems的其他Ruby版本，可以参阅下面的OS X和Linux平台的说明，因为不同平台的安装方法是相当一致的。

Mac OS X、Linux和其他UNIX平台

RubyGems用纯Ruby代码开发，它在所有平台上的安装方法都是相似的。下面的说明针对OS X和Linux平台作了调整，但也可用于所有其他平台：

1. 访问RubyGems项目网站<http://rubyforge.org/projects/rubygems/>。
2. 找到“Download”链接（紧跟在“Latest File Releases”之后），进入下载页面。
3. 下载最新可用的.tar.gz或ZIP文件。最新版本应该会高亮显示。
4. 解压缩.tar.gz或ZIP文件。在OS X或Linux平台上，这一操作可用`tar xzvf rubygems-0.9.0.tgz`命令完成，其中文件名应该替换成你刚刚下载的文件名。
5. 用`cd rubygems-0.9.0`（或类似的）命令进入解压缩RubyGems文件时创建的文件夹。
6. 以超级用户（即root）身份运行`ruby setup.rb`命令，或用`sudo: sudo ruby setup.rb`，然后在提示符界面输入你的口令（OS X平台）或root用户口令（其他平台）。
7. RubyGems进行安装并报告成功（如遇出错，请检查出错提示信息）。

注 Linux用户请注意：如果你在安装过程中遇到错误，报告找不到某些程序库（例如YAML或Zlib），你使用的可能是Linux操作系统预装的Ruby版本。如果你不想从源代码安装最新的Ruby版本（这通常是最好的办法，详见第1章），可以用操作系统的包管理系统安装这些丢失的程序库。例如，在Ubuntu或Debian系统中，用`apt-get libyaml-ruby`和`apt-get libzlib-ruby`命令通常可以解决这一问题。

安装完成后，RubyGems的主程序gem应该存放在某个目录（例如/usr/bin或/usr/local/bin/），且该目录已位于搜索路径中，可以从命令行输入gem并按回车键立即运行。如果未成功，则需要找到gem安装到哪里去了，并将其目录放到搜索路径中。另外还有一种方法，以后使用gem时可以用前缀标示完整路径名（例如/usr/bin/gem）。

注 如果没有系统级安装权限，可以把RubyGems安装在本地用户目录中。更多相关的详细内容，请参见<http://docs.rubygems.org/>网站的RubyGems文档。

查找gem包

RubyGems系统可做的有用操作之一，是获取本机已安装的gems清单，以及可供下载安装的gems清单。具体做法是用gem的list命令。在命令行运行gem list命令，将得到与下面类似的结果：

```
*** LOCAL GEMS ***

sources (0.0.1)
  This package provides download sources for remote gem installation
```

数量并不多，但这是个开始。上面的清单显示有个“sources”gem包（版本是0.0.1）已安装，以及对该gem包的基本说明。

你可以查询远程gem服务器（目前位于RubyForge网站），代码如下所示：

```
gem list --remote
```

```
***REMOTE GEMS ***

abstract (1.0.0)
  a library which enables you to define abstract method in Ruby

ackbar (0.1.1,0.1.0)
  ActiveRecord KirbyBase Adapter

action_profiler (1.0.0)
  A profiler for Rails controllers

actionmailer (1.2.3,1.2.2,1.2.1,1.2.0,1.1.5,1.1.4,1.1.3,1.1.2,1.1.1,1.0.1,
1.0.0,0.9.1,0.9.0,0.8.1,0.8.0,0.7.1,0.7.0,0.6.1,0.6.0,0.5.0,0.4.0,
0.3.0)
  Service layer for easy email delivery and testing.

[...1,000s of lines about other gems removed for brevity...]
```

在大约1分钟以内，成千上百的gem包和相关说明在屏幕上飞驰而过（不过请允许几分钟的延迟，因为RubyForge网站常常极其繁忙）。

在这样一个巨量清单中徘徊是不现实的，但一般来说，在此操作之前，你会知道自己要安装哪个gem包。因特网上常常有人推荐gem包，本书或其他教程也会要求你安装特定的gem包。

不过，如果你希望“浏览”gem包，最好的方法是访问<http://rubyforge.org/>网站，这是RubyGems仓库的大本营。RubyForge提供了搜索工具，以及仓库中每个gem包的更多信息。

另外还有一种方法，可以直接使用gem程序提供的搜索功能，代码如下所示：

```
gem query --remote --name-matches class
```

```
***REMOTE GEMS ***
```

```
calibre-classinherit (2.1.0)
```

```
  Provides class-level inheritance for mixin modules.
```

```
calibre-classmethods (2.0.0,1.0.0)
```

```
  Provides class-level inheritance for included modules.
```

```
calibre-nackclass (0.5.1)
```

```
  Nack, which stands for Not-ACKnowledged, is a more efficient tool
  for deferable errors.
```

```
calibre-nullclass (1.0.0)
```

```
  Null is a alternate to Nil that's doesn't raise NoMethodError.
```

```
classifier (1.3.0,1.2.0,1.1.1,1.1,1.0)
```

```
  A general classifier module to allow Bayesian and other types of
  classifications.
```

```
classroom (0.0.2,0.0.1)
```

```
  Classroom is a 'class server' based on DRb
```

在本例中，你向仓库询问名字包含“class”一词的所有gem包。

注 在第16章中，我们将考察大量RubyGems和其他程序库，并了解它们的工作原理，以及怎样将其应用到自己的项目中。

安装一个简单的gem包

当你找到想安装的gem包的名字，即可在命令行用一行命令进行安装（在下例中，应把feedtools替换成想安装的gem包名字，当然feedtools也是值得试验的很好的gem包）：

```
gem install feedtools
```

警告 在UNIX类平台中（包括OS X和Linux），你可能很快得到出错信息，提示你没有安装gem包的权限。原因是Ruby通常作为系统程序安装，而你的当前用户则没有在系统级安装新程序库的特权。要解决这一问题，要么切换为root用户并重新运行gem install命令，要么用sudo: sudo gem install feedtools。

如果一切正常，你将看到与下面类似的输出结果：

```
Attempting local installation of 'feedtools'
Local gem file not found:feedtools*.gem
Attempting remote installation of 'feedtools'
Updating Gem source index for:http://gems.rubyforge.org
Successfully installed feedtools-0.2.26
Installing RDoc documentation for feedtools-0.2.26...
```


首先，RubyGems检查gem包在当前目录中是否存在（如果你愿意，可以保留自己的gem包存储位置）。如果不存在，则转向RubyForge网站下载该gem包并远程安装。最后，用rdoc（详见第8章）构建该程序库的文档，安装到此完成。几乎所有gem包的安装过程均相同。

注 在许多情况下，安装某个gem包，需要一并安装其他gem包。也就是说，你要安装的gem包可能需要其他gem包来完成自己的功能。如果碰到这种情况，gem会告诉你，并在你同意的情况下安装所需的gem包。

此时如果再次运行gem list命令，你的本地gem包列表将包含刚刚安装的gem包（在本例中，是feedtools）。

使用gem包

由于RubyGems系统不是内置集成在Ruby中的，因此需要告诉你的程序，你想使用和载入gem包。

为了演示怎样使用gem包，你将安装redcloth这一gem包，该程序库的功能是把特殊格式的文本转成可供网页使用的HTML。如前文所示，使用gem install redcloth命令（或sudo gem install redcloth命令，如果没有以root用户或超级用户的身份运行）来安装gem包。

安装完成后，运行irb或新建一个Ruby源代码文件，并使用redcloth包，如下所示：

```
require 'rubygems'
require 'RedCloth'
r = RedCloth.new("this is a *test* of _using RedCloth_")
puts r.to_html
```

```
<p>this is a <strong>test</strong> of <em>using RedCloth</em></p>
```

在本例中，首先载入RubyGems程序库，然后用require载入RedCloth程序库。当第一行代码载入RubyGems时，RubyGems程序库覆写了require方法，使其可以用来载入gem包，仿佛gem库是普通的本地程序库一样。

在此之后，可以使用RedCloth程序库，创建对象，调用该方法等。如果得到了HTML的输出结果，表示一切正常。就这样，你使用了自己的第一个gem包。

注 Windows用户用来安装Ruby和RubyGems的“一键安装程序”，让RubyGems默认载入到本机运行的所有Ruby程序中，因此require 'rubygems'就不再需要了，当然保留这行代码也没什么危害。

安装更复杂的gem包

我们来看安装gem包的第二个例子，看看实际情况中更复杂gem包的安装。

Hpricot是用于Ruby的HTML/网页处理程序库，为了提高速度，它使用了C语言编写的解析器。这就要求对C代码进行编译，尽管C编译器在UNIX/Linux和Mac OS X机器上很常见，但很少在Windows机器上看到。这就要求Hpricot的gem包发行两个版本：一个用于可编译C代码的机器；另一个包含预编译版本，用于Windows机器。

注 我们将在第14章使用Hpricot来处理网页内容，因此如果你准备按顺序学习本书，

建议你现在安装Hpricot。

可以用常规方式开始安装Hpricot的gem包：

```
gem install hpricot
```

```
Attempting local installation of 'hpricot'
Local gem file not found:hpricot*.gem
Attempting remote installation of 'hpricot'
Updating Gem source index for:http://gems.rubyforge.org
Select which gem to install for your platform (i686-darwin8.8.1)
 1.hpricot 0.4 (ruby)
 2.hpricot 0.4 (mswin32)
 3.Cancel installation
>
```

gem包的安装并未直接进行，而是由gem客户端提示，Hpricot 0.4版本有两个不同类型的gem包：一个版本标为“ruby”，另一个标为“mswin32”。标为“ruby”的gem包是通用gem包，用于安装在任何类UNIX环境且有C编译器的机器上（这类gem包包括Cygwin，它是用于Windows的UNIX行环境）。标为“mswin32”的版本则是专为纯Windows环境下Ruby用户设计的，包含预编译的HTML解析器二进制版本，用于32位x86的Window环境。

你可以输入数字来确定你的选择，并按回车键。

Hpricot安装完成后，可以很容易检查是否一切正常：

```
require 'rubygems'
require 'hpricot'
puts "Hpricot installed successfully" if Hpricot
```

如果想试试Hpricot更复杂的代码示例，请参阅第14章，其中使用Hpricot来处理网页的内容。

Hpricot的每个版本除了通用版和Windows版，还有特别的开发者版本，用来与开发人员最新的修改保持一致。尽管不时发布程序库的修正版本是一种常见做法，但测试驱动开发的出现，让我们可以相当安全地使用开发人员当前正在开发的更新版本，同时这也是实用的做法。因此，你可以选择从默认的gem服务器安装Hpricot的gem包的完整修正版本，或是选择从开发人员自己的gem服务器安装“几分钟前刚刚修改”的源代码版本。

Hpricot的开发者版本并未保存在默认的gem服务器上，而是放在由Hpricot开发人员自己维护的gem服务器上。要访问这一版本，只须稍稍调整gem install命令：

```
gem install hpricot --source code.whytheluckystiff.net
```

这一命令指示gem不再从默认的gem服务器上安装，而是从特定的code.whytheluckystiff.net安装。许多项目有其自己的gem服务器，因此如果想安装gem包和程序库的试验前沿版本，请到项目的网站查询安装前沿/源代码/试验版本的相关信息（所有以上词语均可用于描述前沿版本）。

运行上述命令，将得到更多的选项，比从默认gem服务器得到的选项多得多：

```
Select which gem to install for your platform (i686-darwin8.8.1):
 1.hpricot 0.4.52 (ruby)
```



```

2.hpricot 0.4.52 (mswin32)
3.hpricot 0.4.47 (ruby)
4.hpricot 0.4.43 (mswin32)
5.hpricot 0.4.43 (ruby)
6.hpricot 0.4 (mswin32)
7.hpricot 0.4 (ruby)
8.hpricot 0.3.32 (mswin32)
9.hpricot 0.3.32 (ruby)
10.hpricot 0.3.0 (mswin32)
11.hpricot 0.3 (ruby)
12.hpricot 0.2 (ruby)
13.hpricot 0.1 (ruby)
14.Cancel installation
>

```

由于gem版本号是递增的，主次版本号从左至右排列，因此这里的最佳可用版本是0.4.52，选项1和2相应用于通用版本和Windows版本。

注 相对于修正版本，Ruby on Rails也有“前沿”版本。许多开发人员选择使用前沿版本，因为这种版本提供的功能比最新官方版本要多得多。不过，Rails团队没有用Hpricot这样的版本号，而是用整数修订版本，例如5098或5200，数字越大，版本越新。相关内容详见第13章。

更新和卸载gem包

RubyGems的主要功能之一，是gem包可以很容易地更新。你可以用一行命令，更新当前安装的所有gem包：

```
gem update
```

这一命令让gem到远程gem仓库中寻找当前已安装gem包的新版本，如果有新版本则安装之。如果你只想更新特定的gem包，可在上面的命令之后加上该gem包的名字后缀。

卸载gem包是所有任务中最简单的。用uninstall命令（这里feedtools应该替换为想卸载的gem包的名字）：

```
gem uninstall feedtools
```

如果本机上同一个gem包有多个版本，则gem会询问你首先卸载哪一个版本（你可以让它一次卸载所有版本），如下例所示：

```
$ gem uninstall rubyforge
```

```
Select RubyGem to uninstall:
```

```

1. rubyforge-0.3.0
2. rubyforge-0.3.1
3. All versions

```

创建自己的gem包

也可以从自己的程序库和应用程序中创建gem包，这是很自然的事。创建gem包的完整过程

在第10章讲述，其中还包含其他方法，可供你把应用程序部署给其他用户（或全世界）。

7.3 小结

在本章中，我们考察了Ruby提供的一些方法，用来管理更大型的项目，以及为了减轻开发负担，对巨量的预编写代码库世界的访问。

除了从其他文件中包含代码，我们也考察了模块（及其命名空间）的用法，用来把存在潜在冲突的类、方法和常量划分到不同的组中。模块还提供了无须使用继承即可向其他类混入功能的方法。

在主流发行版本中，Ruby提供了有用的程序库财富，但运用诸如RubyGems的工具，可以访问成千上万其他Ruby开发者编写的代码，从而可以比其他方式更快实现更复杂的程序。

我们来回顾一下本章涵盖的主要概念：

- 项目：多个文件和子目录的集合，构成Ruby应用程序或程序库的单个实例。
- require：用来载入并处理其他文件中代码的方法，该文件中无论什么类、模块、方法和常量都被包含到当前作用域中。load与之相似，但并非只执行一次包含操作，而是每次调用load时都把包含的代码重新处理一次。
- 程序库：例程、类、方法和/或模块的集合，提供许多其他程序可以使用的功能。
- RubyGems：用于Ruby程序库和/或应用程序的打包系统，使开发人员安装和维护起来更容易。
- 前沿/源代码/开发版本：程序库和应用程序的特别版本，并非官方版本，而是反映程序库或应用程序开发人员的最新工作成果。不过，随着测试驱动开发的流行，许多这类前沿程序库用起来仍然很可靠，尽管其中大多数最新增加的功能可能还未提供完整文档或经过充分测试。
- gem包：用RubyGems系统打包的单个程序库（或应用程序），也可以叫做“RubyGem包”。

从这里开始，后面的许多章节都将运用程序库的威力，并用多个程序库组成单个应用程序。这样的例子之一是Ruby on Rails框架，我们将在第13章介绍，从本质上说这一框架本身是由多个程序库组成的巨型程序库！

在第16章中，我们将回到RubyGems，对一些最有用的gem包进行考察，了解其功能，及其使用方法。



第8章 文档编写、错误处理、调试和测试

在本章中，我们将详细考察怎样开发稳定可靠的程序，即对程序进行文档编写、错误处理、调试和测试。在大多数人的想像中，这些任务并不属于开发的范畴，但对于整个编码过程来说，这些任务是非常重要的。没有对代码进行文档化、调试和测试，别人就不大可能利用你的工作成果，而且你也将陷入发布的脚本和应用程序中有错误的风险。

本章展示怎样使用Ruby提供的工具来生成文档，处理程序中的错误，测试代码效率和确保（大部分）代码无缺陷。

8.1 文档编写

即使你是开发和使用某段Ruby代码的唯一人选，随着时间的推移，也不可避免会遗忘其架构组成和功能处理的具体细节。为了确保不发生这种“代码健忘症”，应该在开发过程中对代码进行文档编写。

传统上，文档编写常常由第三方而非开发者来完成，或在主体开发工作完成后进行。尽管人们总是希望开发人员在代码中留下注释，但真正高质量的文档不需要其他开发人员和用户阅读源代码即可理解，在此情况下，代码注释的重要性已经有所降低。

与其他语言的文档化功能相比较，Ruby让文档编写变得极其容易，用RDoc（它是Ruby Documentation（Ruby文档编写）的缩写）工具程序，也可以在编写代码的同时创建文档。

8.1.1 用RDoc生成文档

RDoc称自己为“Ruby源代码的文档生成器”，它是个工具程序，其功能是通过读Ruby源代码文件，并创建结构化HTML文档。RDoc是Ruby标准发行版中自带的，很容易找到并使用。如果你的Ruby安装版本中不知为何找不到RDoc，可以从RDoc官方网站下载，网址为<http://rdoc.sourceforge.net/>。

RDoc理解大量Ruby语法，并无须提示，即可为类、方法、模块和许多其他Ruby结构创建文档。

你可以在文档中留下注释，放在你想生成文档的类、方法或模块的定义之前，以供RDoc使用。例如：

```
#This class stores information about people.
class Person
  attr_accessor :name, :age, :gender

  #Create the person object and store their name
  def initialize(name)
    @name =name
  end
end
```

```
#Print this person's name to the screen
def print_name
  puts "Person called #{@name}"
end
end
```

这是个用注释进行文档编写的简单类，可读性很高，而RDoc可以在眨眼之间，将其转换成一组漂亮的HTML文档。

要使用RDoc，只需在命令行用`rdoc <源代码文件名>.rb`即可，如下所示：

```
rdoc person.rb
```

注 在Linux和OS X平台上，这一命令应该无须额外配置即可生效（只要包含RDoc的目录（一般是`/usr/bin`或`/usr/local/bin`）在搜索路径中）。而在Windows平台上，可能需要在`rdoc`之前加上完整路径的前缀。

该命令告诉RDoc，处理`person.rb`文件，并生成HTML文档。默认情况下，RDoc会在当前目录创建名为`doc`的子目录，并将相应的HTML和CSS文件放置其中。当RDoc处理完毕时，你可以打开`doc`目录中的`index.html`文件，并可以看到一些基本的文档内容，如图8-1所示。

Files	Classes	Methods
person.rb	Person	new (Person) print_name (Person)
<div> <div>Class Person</div> <div>In: person.rb</div> <div>Parent: Object</div> </div> <p>This class stores information about people.</p> <div> <div>Methods</div> <div>new print_name</div> </div> <div> <div>Attributes</div> <div>age [RW]</div> <div>gender [RW]</div> <div>name [RW]</div> </div> <div> <div>Public Class methods</div> <div>new(name)</div> <div>Create the person object and store their name</div> </div> <div> <div>Public Instance methods</div> <div>print_name()</div> <div>Print this person's name to the screen</div> </div> <div>[Validate]</div>		

图8-1 正如在浏览器中所见的基本的RDoc HTML输出

由图中可以看出，这一HTML文档的顶部由三个框架组成，其中包含指向相应文档文件、类和方法的链接。下面是一个主框架，其中包含当前查看的文档内容。顶部三个框架让你可以在不同类和方法之间实现一键跳转，在大型文档中，这一功能相当有用。

在查看`Person`类的文档时，其中显示了该类包含哪些方法，这些方法的文档内容，以及该类为其对象提供了哪些属性。RDoc根据源代码和你的注释，直接生成所有内容。

8.1.2 RDoc技术

在上一节中，用RDoc根据源代码文件中几个简单的注释，生成了文档。但RDoc对这么小的例子极少有用，它真正的威力在于大型项目的应用，以及它的高级功能。本节将介绍其中部分功能，以便在大型项目中能够正确编写代码注释。

注 下面章节只给出RDoc部分功能的基本概述。对RDoc完整文档的阅读和学习超出了本书的范围，请访问RDoc官方网站，网址为<http://rdoc.sourceforge.net/doc/>。

为整个项目生成文档

前文用rdoc后跟文件名的方式为单个文件生成文档。但在大型项目中，可能有成百上千个文件要处理。如果不带文件名使用rdoc命令，RDoc将处理当前目录及其所有子目录中的所有文件。和前文一样，结果文档全部放在doc目录中，整个文档都可以从index.html访问。

基本的格式化

对RDoc生成的文档进行格式化很容易。RDoc自动识别注释中的段落，甚至可以根据空格识别结构。下面是RDoc能够识别的格式化标志的一些例子：

```
#=RDoc Example
#
#==This is a heading
#
#*First item in an outer list
# *First item in an inner list
# *Second item in an inner list
#*Second item in an outer list
# *Only item in this inner list
#
#==This is a second heading
#
#Visit www.rubyinside.com
#
#==Test of text formatting features
#
#Want to see *bold*or _italic_text_?You can even embed
#+text that looks like code+by surrounding it with plus
#symbols.Indented code will be automatically formatted:
#
#   #class MyClass
#   #def method_name
#   #puts "test"
#   #end
#   #end
```

如果用RDoc处理这段注释，将会得到如图8-2所示的结果。要学习关于RDoc的一般格式化

功能的知识，最好的方法是阅读已作充分RDoc注释的现有代码，例如Ruby on Rails框架的原代码，或参考RDoc官方文档，网址为<http://rdoc.sourceforge.net/doc/>。

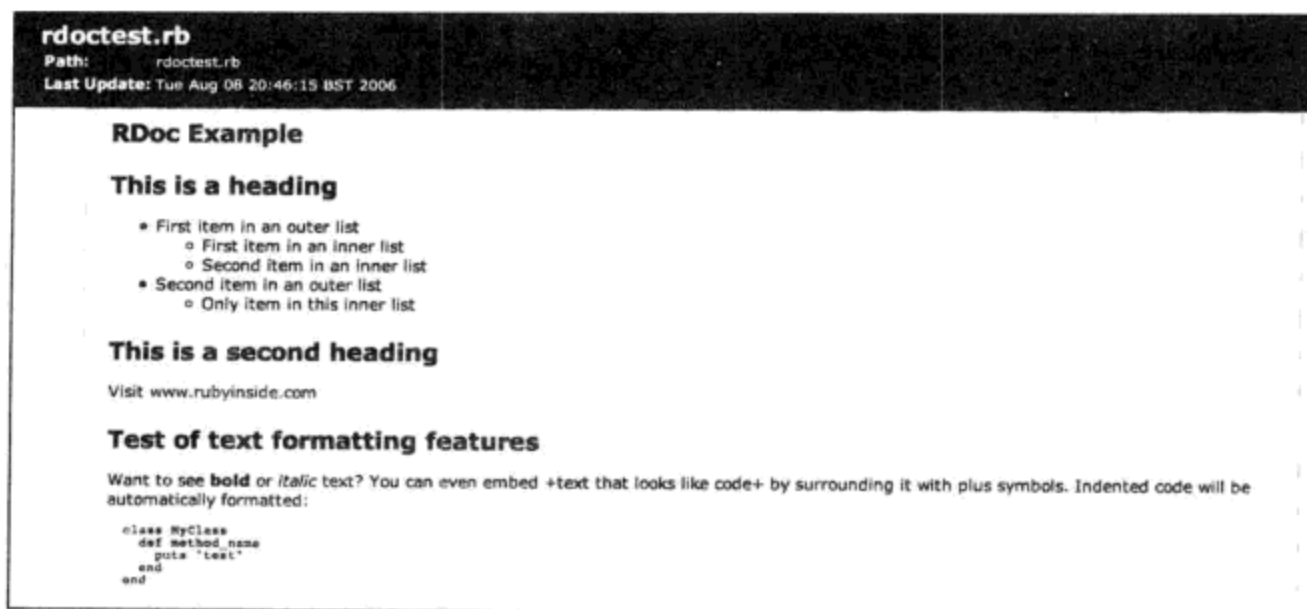


图8-2 RDoc生成的格式化功能测试文件

修饰符和选项

即使开发人员对RDoc一无所知，它也能正常工作。但为了最大程度地利用RDoc，需要了解其几项功能的工作原理，以及怎样进行定制调整。RDoc支持许多修饰符，以及大量命令行选项，可供添加在注释中。

:nodoc: 修饰符

在默认情况下，RDoc将试图利用一切相关内容来构建文档。但有时你想让RDoc忽略某些模块、类或方法，特别是在尚未编写相关文档的情况下。为了实现这一目的，只须在该模块、类或方法定义之后加上:nodoc:注释即可，如下所示：

```
#This is a class that does nothing
class MyClass
  #This method is documented
  def some_method
    end

  def secret_method #:nodoc:
    end
end
```

在本例中，RDoc将忽略secret_method方法。

:nodoc:只对直接紧跟的那一行元素有效。如果你希望把:nodoc:应用到当前元素及其所有子元素（例如，类中的所有方法），请按如下方法：

```
#This is a class that does nothing
class MyClass #:nodoc:all
  #This method is documented (or is it?)
  def some_method
    end
```

```

    def secret_method
    end
end

```

现在，MyClass类中没有任何内容被RDoc编入文档。

开启和关闭RDoc处理

你可以用#++和#--，让RDoc暂时停止处理注释，如下所示：

```

#This section is documented and read by RDoc.
#--
#This section is hidden from RDoc and could contain developer
#notes,private messages between developers,etc.
#++
#RDoc begins processing again here after the ++.

```

当你想留下仅供自己使用的注释时，这一功能特别有用，但不作广泛使用。

注 RDoc不处理方法内部的注释，因此常用的代码注释不会用于文档生成。

命令行选项

和包括Ruby在内的许多命令程序一样，RDoc也有许多命令行选项，如下所示：

- `--all`：通常RDoc只处理公共方法，但`--all`选项强制要求RDoc为源代码文件的所有方法生成文档。
- `--fmt <格式名>`：以某种格式生成文档（默认是html，但在某些配置环境中，也可用xml、yaml、chm和pdf）。
- `--help`：得到RDoc命令行选项的使用帮助，并找出可以使用哪种格式。
- `--inline-source`：通常，源代码以弹出窗口显示，但该选项强制代码嵌入在文档中一起显示。
- `--main <名字>`：把文档主索引页面所显示的类、模块或文件设置为<名字>（例如`rdoc --main MyClass`）。
- `--one-file`：让RDoc把所有文档内容放在一个文件中。
- `--op <目录名>`：把输出目录设置为<目录名>（默认为doc）。

在`rdoc`命令行选项之后，可附缀供RDoc生成文档的文件名字。另外，如果不指定任何命令行选项，RDoc将遍历当前目录及其所有子目录，并为整个项目生成文档。

注 RDoc支持更多的命令行选项，比这里介绍的多得多，在RDoc官方文档中有详细叙述。另外，在命令行用`rdoc --help`运行RDoc，可得到命令行选项的清单。

8.2 调试与出错

错误总会发生。你开发的程序不可避免会有缺陷，一般不会立即发现。在正则表达式中错误使用的字符，在数学符号中错误输入的内容，都会让程序的运行结果大为不同，让原本可靠的程序，变得不停抛出错误或生成未预期的结果。

8.2.1 异常和出错处理

异常是程序中发生错误时产生的事件。异常可能导致程序显示出错信息并立即退出，或可由程序中的错误处理（error handling）例程以合理的方式从错误中恢复。

例如，某程序可能依赖于网络连接（例如因特网），如果网络连接不可用，则当程序试图访问网络时，将产生一个错误。这时，程序代码不应该用一段令人费解的出错信息，并唐突地终止程序，而应该处理这个异常，并首先在屏幕上输出一段用户友好的出错信息。另外，程序可能有某种离线运行的机制，因此可以利用“访问不可用的网络或服务器”的异常，进入这种运行模式。

抛出异常

在Ruby中，异常被打包在Exception（异常）类或Exception的许多子类的对象中。Ruby有30种预定义的主要异常类，用于处理各种不同类型的错误，例如NoMemoryError（无内存错误）、RuntimeError（运行期错误）、SecurityError（安全错误）、ZeroDivisionError（除零错误）和NoMethodError（无此方法错误）。在使用irb的过程中，你可能已经见过这样一些出错信息。（附录B提供了一张表，其中列出了Ruby的所有标准异常类）

当异常被抛出（raised）时（当在程序运行过程中发生异常时，称为抛出异常），Ruby立即在调用当前程序的例程树（称为栈（stack））中回溯，查找可以处理这个特定异常的例程。如果找不到任何错误处理例程，则退出程序，返回原始的出错信息。例如：

```
irb(main):001:0> puts 10 / 0
```

```
ZeroDivisionError: divided by 0
    from (irb):1:in `/'
    from (irb):1
```

这段出错信息显示，有个ZeroDivisionError（除零错误）类型的异常被抛出，因为你试图用10除以0。

当你执行不正确的功能时，Ruby可以自动抛出异常，你也可以在自己的代码中抛出异常，即用raise方法加上现有的异常类，也可以从Exception类中继承，创建自己的异常类。

有个名为ArgumentError（参数错误）的标准异常类，用于提供给方法的参数完全错误时。如果自己的方法接受到错误数据时，可以用这个类作为异常：

```
class Person
  def initialize(name)
    raise ArgumentError, "No name present" if name.empty?
  end
end
```

如果用Person类创建新对象，并以空白的名字作为参数，将导致抛出异常：

```
fred = Person.new('')
```

```
ArgumentError: No name present
```

注 在调用raise时可以不加任何参数，此时将抛出通用的RuntimeError异常。这

不是好的做法，因为这样的异常将没有任何信息或意义。因此，只要有可能，永远要在`raise`之后附加异常类，并提供出错信息。

不过，如果你愿意，也可以创建自己的异常类型。例如：

```
class BadDataException < RuntimeError
end

class Person
  def initialize(name)
    raise BadDataException, "No name present" if name.empty?
  end
end
```

这一次，你从Ruby的标准异常类`RuntimeError`继承，创建了`BadDataException`（错误数据异常）类。

在目前看来，这个类似乎毫无意义，因为抛出这种不同类型的异常没有更多用处。但创建这个类的原因在于，可以用错误处理代码，以不同的方式处理不同类型的异常，如下文所示。

处理异常

在上一节中，我们了解了异常的工作原理。当异常被抛出时，程序执行被挂起，并从栈中回溯，找到可以处理异常的代码。如果找不到这类异常的处理程序，则程序终止运行，“死于”该异常相关的出错信息。

但在大多数情况下，因简单错误而终止程序是不必要的。错误可能很轻微，或有其他途径可选。因此，异常是可以处理的。在Ruby中，用`rescue`（挽救）语句，以及`begin`和`end`来定义处理异常的代码块。例如：

```
begin
  puts 10 / 0
rescue
  puts "You caused an error!"
end
```

```
You caused an error!
```

在此情况下，`begin`和`end`定义了一段代码，当异常被抛出时，将用`rescue`代码块内部的代码进行处理。首先，试图算出10除以0，这导致抛出`ZeroDivisionError`类的异常。但由于这段代码包含`rescue`段落，因此该异常将由`rescue`段落内部的代码行处理。因此，程序并未“死于”`ZeroDivisionError`，而是在屏幕上打印输出一段“You caused an error!”文本。

这对于依赖于外部数据源的程序来说，是相当重要的。我们来看下面这段伪代码：

```
data = ""
begin
  <..code to retrieve the contents of a Web page..>
  data =<..content of Web page..>
rescue
  puts "The Web page could not be loaded!Using default data instead."
```

```

    data =<..load data from local file..>
  end
  puts data

```

从这段代码可以看出，异常处理为什么会极其有用。如果检索网页内容失败（例如，没有连接到因特网），那么错误处理例程就会挽救异常，警告用户出现错误，然后转而从本地文件中载入一些数据。这当然比立即退出程序要好得多！

在上一节中，我们了解了怎样创建自己的异常类，这么做的动机是可以用不同的方式，挽救不同类型的异常。例如，对于代码中的致命错误，和“未连接到网络”这种轻微错误，你想有不同的反应方式。你也可能想要忽略某些错误，只对特定错误进行处理。

`rescue`的语法让不同异常的不同处理相当方便：

```

begin
  ...code here ...
rescue ZeroDivisionError
  ...code to rescue the zero division exception here ...
rescue YourOwnException
  ...code to rescue a different type of exception here ...
rescue
  ...code that rescues all other types of exception here ...
end

```

这段代码包含多个`rescue`代码块，其中每个都根据所抛出异常的类型而定。如果在`begin`和`rescue`块之间的代码抛出了`ZeroDivisionError`异常，则挽救`ZeroDivisionError`的代码将被执行，以处理这一异常。

处理被忽略的异常

除了可以用不同代码块处理不同类型的异常，还可以接收并使用异常。这是通过`rescue`代码块中一点额外语法实现的：

```

begin
  puts 10 / 0
rescue => e
  puts e.class
end

```

ZeroDivisionError

当异常被抛出时，并不是简单执行一段代码，而是把异常对象本身赋值给变量`e`，然后可以按自己的意愿来使用这一变量。这对于异常类中包含你关心的额外功能或属性的情况，是特别有用的。

8.2.2 Catch与Throw方法

尽管创建自己的异常类和异常处理程序对解决出错情况非常有用，但有时想在正常操作过程中，以类似异常的方式打破执行流程（例如循环），但并不一定真正产生错误了。Ruby为此提供了两个方法：`catch`和`throw`。

`catch`和`throw`的工作方式与`raise`和`rescue`有一点类似，但`catch`和`throw`是对符号进行处理，而不是异常。设计这两个方法的目的，是用于无错误发生的情况下，可以快速从嵌套循环、方法调用或其他类似情形中跳出。

```
catch(:finish) do
  1000.times do
    x = rand(1000)
    throw :finish if x == 123
  end

  puts "Generated 1000 random numbers without generating 123!"
end
```

上例用`catch`创建了一个代码块。当以符号`:finish`调用`throw`调用时，这个以符号`:finish`为参数的`catch`块将立即终止（并继续运行代码块之后的代码）。

在`catch`代码块中，生成1 000个随机数，一旦随机数是123，则用`throw :finish`立即跳出代码块。但如果你生成了1 000个随机数但其中没有123，则循环完成，该代码块也完成，你将看到提示信息。

`catch`和`throw`无须直接在同一个作用域内。在`catch`代码块中进行方法调用时，`throw`被启用：

```
def generate_random_number_except_123
  x = rand(1000)
  throw :finish if x == 123
end

catch(:finish) do
  1000.times { generate_random_number_except_123 }
  puts "Generated 1000 random numbers without generating 123!"
end
```

这段代码与第一段代码的运行方式完全相同。当`throw`在其当前作用域内无法找到使用`:finish`的代码块时，将从栈中向上跳回，直到找到这样的代码块。

8.2.3 Ruby调试器

调试是修正代码中缺陷的过程。这一过程可以很简单，例如修改一小段程序，可以先进行监控输出，然后再一遍遍地循环这一过程，直到得到正确的输出结果，并让程序的行为表现符合原先的期望。

不过，反复修改并重新运行程序，并不能让你对深藏在代码中的实际情况有任何洞察。有时在程序运行的某个特定时刻，你想知道每个变量分别包含什么内容，或想强制让某个变量包含特定值。当然，你也可以在程序中用`puts`语句来显示在特定时刻这些变量包含什么内容，但很快你的代码就会乱成一团糟，到处散布着调试用的代码。

Ruby提供了一个调试工具，可供一行一行地单步运行代码（如果你愿意的话）、设置断点（即程序执行停顿之处，供你检查内部细节）、和调试代码。这个调试工具和`irb`有点像，但不需

要手工输入整块程序。可以指定要调试的程序文件名，然后就可以随意操作，就像处在目标程序中一样。

例如，创建一个名为`debugtest.rb`的简单Ruby脚本，内容如下：

```
i = 1
j = 0
until i > 1000000
  i *= 2
  j += 1
end
puts "i = #{i}, j = #{j}"
```

如果用`ruby debugtest.rb`命令运行这段代码，将得到如下结果：

```
i = 1048576, j = 20
```

但假定用如下命令，使用Ruby调试器运行这段代码：

```
ruby -r debug debugtest.rb
```

则将得到类似下面的结果：

```
Debug.rb
Emacs support available

debugtest.rb:1:i = 1
(rdb:1)
```

开头两行表示调试器已载入，第三行显示你当前准备执行的代码（在本例中是第一行），第四行是个提示符，你可以在此输入命令。

调试器的功能和`irb`很相似，可以在提示符直接输入表达式和语句。不过，其主要优势不在于此，而是可以用特殊命令，一行一行地运行`debugtest.rb`，或设置断点并进行“观察”（断点依赖于某个为`true`的特定条件——例如，当`x`大于10时停止运行）。

以下是调试器提示符界面最有用的几个命令：

- **list**：列出当前正在运行的程序代码，该命令后面可以附加要查看的行号范围。例如，`list 2-4`显示第2行到第4行代码。如果没有任何参数，`list`显示当前执行点的局部程序。
- **step**：单行运行程序的下一行代码。该命令根据程序代码单行运行程序，一次执行一行。每行执行完毕后，你可以做检查变量、修改变量值等操作。这样你就可以跟踪确切的缺陷发生点。在`step`命令之后附加数字，可以一次执行几行（如果数字大于1的话），例如`step 2`则执行两行代码。
- **cont**：继续运行程序，不再单行运行。程序将持续运行，直到程序结束，或碰到断点，或碰到`watch`命令的某个条件成为`true`状态。
- **break**：在某行设置断点，例如用`break 3`在第3行代码设置断点。这表示如果你用`cont`继续运行，程序将运行到第3行再次停下来。当你想检查运行状态细节时，这个命令很有用。
- **watch**：设置条件断点。可以指定某个导致程序停止运行的条件，而不是在特定的程序行

设置断点。例如，如果想让程序在x大于10时停止，可用`watch x > 10`命令。对于找到缺陷发生的确切时点，这个命令非常完美，如果缺陷是在某个条件成为true的情况下发生的话。

- `quit`: 退出调试器。

以下是一个用`debugtest.rb`进行调试的简单调试进程：

```
#ruby -r debug debugtest.rb
Debug.rb
Emacs support available.

debugtest.rb:1:i =1
(rdb:1)list
[-4,5 ] in debugtest..rb
=>1 i =1
   2 j =0
   3 until i >1000000
   4 i *=2
   5 j +=1
(rdb:1)step
debugtest.rb:2:j =0
(rdb:1)i
1
(rdb:1)i =100
100
(rdb:1)step
debugtest.rb:3:until i >1000000
(rdb:1)step
debugtest.rb:4:i *=2
(rdb:1)step
debugtest.rb:5:j +=1
(rdb:1)i
200
(rdb:1)watch i >10000
Set watchpoint 1:i >10000
(rdb:1)cont
Watchpoint 1,toplevel at debugtest.rb:5
debugtest.rb:5:j +=1
(rdb:1)i
12800
(rdb:1)j
6
(rdb:1)quit
Really quit?(y/n)y
```

这一调试进程展示了单步运行代码、检查变量值、原地修改变量值、设置观测断点等操作。当调试代码时，99%的时间内都在使用这些工具，而且随着经验积累，调试环境将成为威力强大的工具，就像`irb`一样。

但许多Ruby开发人员并不特别经常用调试器，因为与测试驱动开发和单元测试等现代技术方法（这些将在后文谈到）相比，调试的工作方式和 workflows 看起来有点过时。如果调试器看起来比较可口，测试则会让你口水横流。

8.3 测试

测试是现代软件开发的一个基础组成部分，它能帮助你解决开发过程中许多意料之外的问题。如果没有配备合适的测试系统，你永远不能确信开发的系统是零缺陷的。如果有一套良好的测试系统，可能还会有1%的缺陷，但这已经是相当大的改善了。

前面我们考察了怎样处理显式的错误，但有时在某种情况下，程序可能会表现异常，例如，某些数据会导致算法返回不正确的结果，或产生非法数据，虽然这些数据不正确，但是不会引起明显的错误。

解决方法之一是调试代码（如前文所见），但调试只能解决一时的问题。调试代码有可能解决了一个问题，但却引起许多其他问题！因此，只用调试这一种手段，在解决问题时效果不佳，在这种情况下，对代码的功能进行全面测试变得尤为重要。

用户和开发人员过去可能进行过手工测试，即执行某些操作，并观察程序的运行效果。如果发生了错误，则对相关缺陷进行修正，再继续进行测试。事实上，曾有一段时期，开发人员把用户的反馈意见当作测试手段！

不过，随着测试驱动开发（又称为测试优先开发）的快速兴起和广泛流行，这些方法很快成为往事。测试驱动开发是一种全新的哲学，它扭转了软件开发实践的方向。Ruby开发人员已经站在这一技术方法的前沿，因此，有责任为推广和宣传这种开发方法尽自己的一份力量。

8.3.1 测试驱动开发的哲学

测试驱动开发是这样一种技术方法：开发人员在进行编码之前，首先创建一组必须通过系统的测试代码，然后严格地使用这些测试代码来保持程序代码的完整性。不过，从更轻量级的角度来说，“测试驱动开发”也可以指实现代码测试的技术方法，即使不需要在编写要测试的代码之前创建测试代码。

注 本节仅对测试驱动开发作基本介绍。这一主题的内涵极为深广，如果你希望了解更多，可以找到许多有关这一主题的书籍和资源。

例如，可以向String类加入一个简单方法，用来把文本开头字母变为大写，使之成为标题文字：

```
class String
  def titleize
    self.capitalize
  end
end
```

你的目的是创建一个方法，用来把“this is a test”变成“This Is A Test”，也就是说，创建一个让字符串看起来像标题文字的方法。因此，这个titleize方法调用capitalize方法把

当前字符串“大写化”。如果匆忙行事，或不想费心做代码测试，那么当这段代码随意发布时，灾难将很快降临。capitalize方法只把字符串的第一个字母变为大写，而不是整个字符串！

```
puts "this is a test".titleize
```

```
"This is a test"
```

这不是预期的效果！不过，如果使用测试驱动开发，就可以避免发布有缺陷代码的痛苦，只需首先写一些测试代码，来展现所期望的结果：

```
raise "Fail 1" unless "this is a test".titleize == "This Is A Test"
raise "Fail 2" unless "another test 1234".titleize == "Another Test 1234"
raise "Fail 3" unless "We're testing titleize".titleize == "We're Testing Titleize"
```

除非titleize方法的输出符合预期，否则这三行代码将抛出异常。

注 这些测试代码也被称为**断言** (assertion)，因为它可以断言某个条件是true。

如果titleize方法通过了这三个测试，就可以相信，其功能对于其他情况也是完好的。

注 用来测试单个组件或某套功能的一组测试或断言，称为**测试包** (test case)。

titleize方法的当前代码无法通过这套测试包的第一个测试，因此我们来改写这段代码，让它能正常工作：

```
class String
  def titleize
    self.gsub(/\b\w/) { |letter| letter.upcase }
  end
end
```

这段代码读取当前字符串，找到所有单词边界（用\b），传入每个单词的第一个字母（用\w获取），并将其转换成大写。工作完成了吗？我们可以再次运行三个测试即可知晓。

```
RuntimeError: Failed test 3
```

为什么第3个测试失败了？

```
puts "We're testing titleize".titleize
```

```
We'Re Testing Titleize
```

\b还不够聪明，没有检测到真正的单词边界，它只是用空格字符或“非单词”字符来区分单词和非单词。因此，对于“We're”，W和R都被变成大写字母。那么需要调整一下这段代码：

```
class String
  def titleize
    self.gsub(/\s\w/) { |letter| letter.upcase }
  end
end
```

如果你确认在需要大写的字母前面都是空格，那么可以保证它是真正的新单词。我们重新运行这个测试：


```
RuntimeError: Failed test 1
```

又回到了第一步。

你没有注意一件事，虽然查找单词之前的空格，但没有让每个字符串的第一个单词变成大写，因为这些字符串都是以字母开头，而非空格。这听起来没什么，但从中可以了解，简单的功能会变得多么复杂，以及为什么测试对于根除缺陷是如此重要。不过，最终解决方案很简单：

```
class String
  def titleize
    self.gsub(/(\A|\s)\w/){ |letter| letter.upcase }
  end
end
```

如果现在再运行这些测试，你将发现测试直接通过。成功了！

关于测试为什么如此重要，这个基本示例作了清晰的展示。对代码作小小修改，可能导致功能的重大变化，但如果有一套值得信赖的测试代码，则你的注意力可以集中在解决问题上，而不是担心目前的代码是否有缺陷。

与其编写代码并被动等待缺陷的出现，不如主动出击，确定代码应该做什么，然后在结果不满足预期时，立即做出反应。

8.3.2 单元测试

在上一节中，用raise、unless和==创建了一些基本的测试代码，并把方法调用的结果与预期结果相比较。我们可以用这样的方法做大量测试，但无须几个测试，代码就会很快变得杂乱无比，因为没有合适的地方来放置测试代码（你当然不想把测试代码包含在真正的功能代码中）。

幸运的是，Ruby自带了一个程序库，Test::Unit，它让测试简单易行，而且以清晰的结构对测试包进行组织管理。单元测试是测试驱动开发的主要组成部分，也就是说，你要做的是测试程序或系统中的每个具体功能单元。Test::Unit是Ruby进行单元测试的官方程序库。

用Test::Unit的一大好处是它提供了一个标准化的框架，供你编写和执行测试。它并未采用几种不统一的断言编写方式，而是提供了一组核心断言。

我们来把前文的titleize方法改写一下，以便演示Test::Unit的功能。首先创建一个名为test_titleize.rb的新文件：

```
class String
  def titleize
    self.gsub(/(\s\w)/){ |letter| letter.upcase }.gsub(/^\w/)do |letter|
      letter.upcase
    end
  end
end

require 'test/unit'

class TestTitleize < Test::Unit::TestCase
```

```

def test_basic
  assert_equal("This Is A Test", "this is a test".titleize)
  assert_equal("Another Test 1234", "another test 1234".titleize)
  assert_equal("We're Testing", "We're testing".titleize)
end
end

```

在这段代码中，首先包含对String类的扩展方法titleize（如果该方法放在另外的文件中，可用require命令包含之），然后用require载入Test::Unit类，最后从Test::Unit::TestCase继承创建一个测试包。在这个继承而来的测试包类中，只有一个方法（当然也可以有任意多的方法，随你意愿而定，以便条理分明地区分不同的测试），这个方法包含三条断言，与上一节的断言类似。

运行这段脚本，你将看到执行的测试结果。

```

Loaded suite test_titleize
Started
.
Finished in 0.000363 seconds.

1 tests, 3 assertions, 0 failures, 0 errors

```

这一输出结果显示，测试被启动，运行了一个测试方法（本例中是指test_basic），该测试方法有三条断言，均成功通过测试。

假定你向test_basic方法中加入一条必错无疑的断言，如下所示：

```
assert_equal("Let's make a test fail!", "foo".titleize)
```

并重新运行测试：

```

Loaded suite test_titleize
Started
F
Finished in 0.239156 seconds.

1) Failure:
test_basic(TestTitleize) [blah.rb:14]:
<"Let's make a test fail!"> expected but was
<"Foo">.

1 tests, 4 assertions, 1 failures, 0 errors

```

加入了一条必错无疑的断言，它也确实出错了。而Test::Unit也全面完整地解释了发生了什么事，利用这一信息，可以回头修正断言，或修正导致测试不通过的程序代码。在本例中，你是强制断言出错，但如果断言是正常创建的，这样的出错结果则表示程序代码有缺陷。

8.3.3 更多的Test::Unit断言

在上一节中，只用了一种类型的断言——assert_equal，它断言第一个和第二个参数是

相等的（不管是数字、字符串、数组或对象，或其他任何类型）。第一个参数是预期结果，第二个参数是准备生成的结果，如上节断言所示：

```
assert_equal("This Is A Test", "this is a test".titleize)
```

注 `assert_equal`也可以接受可选的第三个参数，作为断言失败时要显示的消息。事实证明，在某些情况下，人工消息比默认的断言失败消息有用得多。

其他几种断言也很有用，如下所示：

- `assert(<逻辑表达式>)`：仅当逻辑表达式不是`false`或`nil`时，断言才能通过（例如`assert 2 == 1`永远不会通过）。
- `assert_equal(预期值表达式, 实际值表达式)`：仅当预期值表达式和实际值表达式的值相等时，断言才能通过（如同用`==`比较）。例如，`assert_equal('A', 'a'.upcase)`将通过。
- `assert_not_equal(预期值表达式, 实际值表达式)`：与`assert_equal`正相反，当预期值表达式和实际值表达式的值相等时，断言将不通过。
- `assert_raise(异常类型1, ..) { <代码块> }`：仅当断言之后的代码块抛出参数指定类型的异常时，断言才能通过。例如，`assert_raise(ZeroDivisionError) { 2 / 0 }`将通过。
- `assert_nothing_raised(异常类型1, ..) { <代码块> }`：与`assert_raise`正相反，仅当列表中的异常没有抛出时，断言才能通过。
- `assert_instance_of(预期类, 对象)`：仅当对象是预期类的实例时，断言才能通过。
- `flunk`：`flunk`是一种特殊类型的断言，它永远不通过。如果想在未写完的测试代码中加入明显的备忘标记，就可以用这个断言，提示测试包还未编写完成！

注 上述所有断言，包括`flunk`，都可以像`assert_equal`一样接受可选的消息参数，作为最后一个参数。

我们将在第12章使用更多断言和单元测试，用于在程序库的构建过程中，开发一组测试代码。

8.4 性能基准度量和优化分析

当代码消除缺陷并正常运行之后，很自然地就会想到，可以发布程序了。但在真实世界中，代码的运行效率常常不高，或达不到所需的运行速度。正如1.8版本的Ruby解释器并不很快，而1.9和后续版本（包括2.0）则采用了全新的实现方式，运行速度有相当大的提升。因此，对代码进行性能基准度量（benchmark），以确保达到尽可能高的运行效率，永远是一件非常重要的事。

性能基准度量是指让代码或程序执行某个功能（常常成百上千次地连续运行，以消除偶然性偏差），并度量其所耗费的运行时间的过程。然后，当你在优化代码时，就可以引用这些时间度量数据。如果优化后的性能基准度量比优化前的更快，则表示优化方向是正确的。幸运的是，Ruby提供了许多工具，帮你代码进行性能基准度量。

8.4.1 性能基准简单度量

Ruby的标准程序库包含一个名为Benchmark的模块，该模块提供几种方法，可以度量你提供的代码的运行速度。例如：

```
require 'benchmark'
puts Benchmark.measure { 10000.times { print "." } }
```

这段代码对在屏幕上打印输出10 000个小数点的速度进行度量。不考虑输出的小数点，输出结果如下所示（这是在我机器上的结果，与你机器上的结果可能不一样）：

```
0.050000 0.040000 0.090000 ( 0.455168)
```

从左到右的各列，分别代表了用户CPU时间值、系统CPU时间值、总计CPU，以及“实际”耗时间。在本例中，尽管把10 000个小数点发送到屏幕花费了0.09秒的CPU时间，但在计算机执行的所有其他事务中，完成屏幕显示则花费了几乎半秒的时间。

由于measure方法接受代码块，因此你可以随心所欲地把代码块做得想怎么复杂精细都行。

```
require 'benchmark'
iterations = 1000000

b = Benchmark.measure do
  for i in 1..iterations do
    x = i
  end
end

c = Benchmark.measure do
  iterations.times do |i|
    x = i
  end
end

puts b
puts c
```

在本例中，对两个不同的从一到一百万的计数方法进行性能基准度量，结果大致如下所示：

```
0.800000 0.010000 0.810000 ( 0.949338)
0.890000 0.010000 0.900000 ( 1.033589)
```

结果显示二者的差别很小，除了用time方法耗费的CPU时间比用for稍微更多一些。你可以用同样的方式，测试相同结果不同计算方法的差异，并在代码中选用最快的方法，以实现性能优化。

Benchmark模块还包含一种方法，可以更方便地完成多个测试。你可以把上面的性能基准度量场景改写如下：

```
require 'benchmark'
iterations = 1000000
```

```
Benchmark.bm do |bm|
  bm.report("for:") do
    for i in 1..iterations do
      x = i
    end
  end
  bm.report("times:") do
    iterations.times do |i|
      x = i
    end
  end
end
```

使用bm方法的主要区别，是可以把一组性能基准测试收集在一起，并以更漂亮的方式显示结果。下面是前文代码的输出示例：

```
      user      system    total      real
for:  0.850000  0.000000  0.850000  (0.967980)
times:0.970000  0.010000  0.980000  (1.301703)
```

bm方法让结果更容易阅读，并为各列增加了标题。

另一个方法，bmbm则把性能基准度量重复做两次，第一次是“排练”，第二次才是真正的结果，因为在某些情况下，CPU缓存、内存缓存和其他因素会影响最终结果。因此，重复测试可以得到更精确的结果。把上例中bm方法替换成bmbm，即可得到如下结果：

```
Rehearsal -----
for:  0.780000  0.000001  0.780001  (0.958378)
times:0.100000  0.010000  0.110000  (1.342837)
-----total:0.890001sec

      user      system    total      real
for:  0.850000  0.000000  0.850000  (0.967980)
times:0.970000  0.010000  0.980000  (1.301703)
```

bmbm运行两次测试，并给出两次的结果，后者应该更精确。

8.4.2 性能优化分析

性能基准度量是度量总体完成时间并比较不同版本代码运行效果的过程，而性能优化分析则是告诉你哪些代码占用了多少时间。例如，在程序中可能有一行代码导致整体运行缓慢，那么通过性能优化分析可以立即看出，你的优化工作应该集中在哪一段代码上。

注 有些人认为性能优化分析是优化领域的圣杯，因此他们建议，不必一开始就考虑编写高效率的程序，而是只须先编写程序，然后进行性能优化分析，再修改最慢的那些代码段。这样做可以避免永无休止的优化。因为，你有可能会永无止境地优化一些并不真正值得的优化的东西，而忽略了可以让效率得到相对提升的代码段。

Ruby内置了一个代码性能优化分析器，你只须在代码开头加上require "profile"，或在源文件名开头用ruby --r profile命令，即可让代码自动得到分析。

下面是个简单的例子：

```
require 'profile'
class Calculator
  def self.count_to_large_number
    x = 0
    100000.times {x +=1 }
  end

  def self.count_to_small_number
    x = 0
    1000.times {x +=1 }
  end
end

Calculator.count_to_large_number
Calculator.count_to_small_number
```

```
% cumulative self      self      total
time seconds seconds calls ms/call  ms/call  name
70.76  7.38  7.38      2 3690.00  5215.00 Integer#times
29.24 10.43  3.05 101000  0.03    0.03 Fixnum#+
0.00  10.43  0.00      2  0.00    0.00
Kernel.singleton_method_added
0.00  10.43  0.00      1  0.00    110.00
Calculator#count_to_small_nu..
0.00  10.43  0.00      1  0.00  10320.00
Calculator#count_to_large_nu..
0.00  10.43  0.00      1  0.00    0.00 Class#inherited
0.00  10.43  0.00      1  0.00  10430.00 #toplevel
```

这里给出了大量信息，但很容易阅读。代码本身很简单，定义了两个类方法，其功能均为统计不同数字的汇总值。Calculator.count_to_large_number包含100 000次循环，而Calculator.count_to_small_number包含1 000次循环。

注 不采用更大的数字，例如1 000 000次循环作为性能基准度量测试的原因，是因为做性能优化分析给程序运行速度带来很大的开销，这与性能基准度量不太一样。尽管程序运行速度会变慢，但变慢的程度是始终如一的，因此可以保证性能优化分析的结果与此无关。

分析结果包含几个纵列。第一列表示在最右列指定方法中所花费的时间百分比。在上例中，性能优化分析器显示了总执行时间的70.76%花在Integer类的time方法中。第二列以秒数表示时间，而不是用百分比。

`calls`列表示该方法被调用了多少次。在我们的示例中，`times`方法只被调用了两次。的确如此，尽管传递给`times`的代码块运行了101 000次。后者的运行次数通过`Fixnum`类的加(+)方法调用次数来反映，总调用次数为101 000次。

可以利用性能优化分析器的结果，来找出程序中的“棘手”点，并解决过度占用CPU时间的低效方法。但对只消耗一般CPU时间的例程进行优化是不值得的，因此应该用性能优化分析器找出最占CPU时间的例程，然后对其进行专门优化。

提示 你还可以使用名为`profiler`的程序库（实际上，`profile`自己也是调用该程序库来完成工作的），对程序中的特定段落而非整个程序进行优化。具体使用方法如下：先用`require 'profiler'`语句，然后在相应位置调用`Profiler__:: start_profile`、`Profiler__:: stop_profile`和`Profiler__:: print_profile ($stdout)`命令。

8.5 小结

本章我们考察了文档化、错误处理、测试、性能基准度量和性能优化分析背后的处理过程，以及Ruby为此所提供的工具。

对于某个程序或某段代码，对其进行文档化、错误处理和测试的质量，体现了开发人员和程序的专业化程度。小型的、快速开发的脚本可能不需要这些工作环节，但如果你正在开发的是一个系统，目的是供他人使用，或用于关键任务，则必须理解错误处理和测试的基本常识，以避免你的代码产生错误、执行异常给你带来羞辱。

再进一步，就必须知道对代码进行性能基准度量和性能优化分析的重要性，以保证你的代码可以随时间的推移不断提升规模。一开始，你可能希望代码只执行一小部分功能——例如处理小型文件——但未来它可能需要用同一段代码来处理巨量的数据，并增加额外的、未曾预期的功能。现在花少量的时间进行性能基准度量和性能优化分析，并对相应代码进行优化，可以在将来获得减少运行时间的回报。

我们来回顾一下本章涵盖的主要概念：

- **RDoc**：是Ruby自带的一个工具，可以根据源代码的结构和注释生成HTML文档。
- **调试**：是指解决源代码中错误的过程，常常是通过单步执行并检查当时程序中数据状态的方法。
- **测试驱动开发/测试优先开发**：是指首先编写测试代码并强制要求某种期望结果，然后再编写实际代码来产生正确的结果的开发过程。
- **测试包**：是一组测试代码，用来测试和检查程序中某段代码（例如某个类或模块）的功能。
- **断言**：是检查某个状态或结果是否达成的单个测试代码，用来检查某段代码是否正常运行。
- **单元测试**：是指用断言对所有各个功能进行验证的代码测试过程，以确保整个系统工作正常。
- **性能优化**：是指通过改写算法、找出新的解决方法，提高代码效率的过程。
- **性能基准度量**：是指通过测试代码执行速度，看出在某种条件下，或用某种方法和算法的

情况下，代码运行速度有多快的过程。你可以用性能基准度量结果来比较不同版本的代码，或比较不同的编码方法。

- 性能优化分析：是指展示程序中哪个方法和例程花费最多执行时间的过程。

这些概念大多数都不直接用于本书的代码示例，因为它们主要与开发周期较长的项目或即将准备发布的代码有关。这并不是说这些概念不重要，而是作为较长开发过程的深层次组成部分，它们超出了其他章节代码示例的作用域。

在第12章，我们将再次简要考察测试方法，在开发程序库的过程中，创建一些简单的测试代码。



第9章 文件和数据库

本章我们将考察怎样用Ruby程序存储、处理外部数据源并与之交互。在第4章，我们简要了解了怎样载入文件并把数据读入应用程序，而本章的内容将远远超出此类基础知识，学完本章，你就能用Ruby程序从零开始创建文件。

在本章后半部分，我们将考察数据库——数据的特殊组织形式——以及怎样与之交互，并简要介绍与SQLite、MySQL、PostgreSQL、Oracle和微软SQL Server等流行数据库系统的交互方法。你可以用数据库来完成简单任务，例如保存关于少量物品或地址本的信息，但数据库也可用于最繁忙的数据处理环境。学习完本章，你将可以和全世界专业开发人员一样，以相同的方式（或至少以相似的方式）使用数据库。

9.1 输入与输出

在计算机术语中，“交互”与输入输出数据有关，或简单地说，与I/O有关。大多数编程语言都内置支持I/O，而Ruby对此尤为出色。

I/O流是所有Ruby输入输出操作的基础。每个I/O流都是一个导管或渠道，用于连接一端与另一端源头，并进行输入或输出操作。这通常是指Ruby程序与键盘之间，或Ruby程序与文件之间，输入和输出操作就在这个流中进行。在某些情况下，例如使用键盘时，只能获取输入信息，因为无法发送数据给键盘；而对于屏幕，只能发送数据给屏幕，而不能从中接收数据。

本节我们将考察怎样在Ruby中访问键盘、文件和其他形式的I/O，以及怎样使用这些形式的I/O。

9.1.1 键盘输入

程序获取外部数据的最简单方法是使用键盘。例如：

```
a = gets
puts a
```

`gets`从标准输入（此时是指键盘）获取一行数据，并将其赋值给`a`。然后即可使用`puts`方法，把该值打印输出到标准输出（此时是指屏幕）。

标准输入与输出

标准输入是许多操作系统提供的关于从用户获取数据的标准方式的默认流。在本例中，标准输入是键盘，但假如在UNIX类操作系统中，例如Linux或Mac OS X，你想把数据重定向给Ruby应用程序，那么标准输入将会是与程序连接的管道。例如：

```
ruby test.rb < somedata.txt
```

此时的输出应该是`somedate.txt`文件的第一行，因为`gets`从标准输入中获取一行数据，而标准输入此时已经变成`somedate.txt`文件的内容。

第9章 文件和数据库

本章我们将考察怎样用Ruby程序存储、处理外部数据源并与之交互。在第4章，我们简要了解了怎样载入文件并把数据读入应用程序，而本章的内容将远远超出此类基础知识，学完本章，你就能用Ruby程序从零开始创建文件。

在本章后半部分，我们将考察数据库——数据的特殊组织形式——以及怎样与之交互，并简要介绍与SQLite、MySQL、PostgreSQL、Oracle和微软SQL Server等流行数据库系统的交互方法。你可以用数据库来完成简单任务，例如保存关于少量物品或地址本的信息，但数据库也可用于最繁忙的数据处理环境。学习完本章，你将可以和全世界专业开发人员一样，以相同的方式（或至少以相似的方式）使用数据库。

9.1 输入与输出

在计算机术语中，“交互”与输入输出数据有关，或简单地说，与I/O有关。大多数编程语言都内置支持I/O，而Ruby对此尤为出色。

I/O流是所有Ruby输入输出操作的基础。每个I/O流都是一个导管或渠道，用于连接一端与另一端源头，并进行输入或输出操作。这通常是指Ruby程序与键盘之间，或Ruby程序与文件之间，输入和输出操作就在这个流中进行。在某些情况下，例如使用键盘时，只能获取输入信息，因为无法发送数据给键盘；而对于屏幕，只能发送数据给屏幕，而不能从中接收数据。

本节我们将考察怎样在Ruby中访问键盘、文件和其他形式的I/O，以及怎样使用这些形式的I/O。

9.1.1 键盘输入

程序获取外部数据的最简单方法是使用键盘。例如：

```
a = gets
puts a
```

`gets`从标准输入（此时是指键盘）获取一行数据，并将其赋值给`a`。然后即可使用`puts`方法，把该值打印输出到标准输出（此时是指屏幕）。

标准输入与输出

标准输入是许多操作系统提供的关于从用户获取数据的标准方式的默认流。在本例中，标准输入是键盘，但假如在UNIX类操作系统中，例如Linux或Mac OS X，你想把数据重定向给Ruby应用程序，那么标准输入将会是与程序连接的管道。例如：

```
ruby test.rb < somedata.txt
```

此时的输出应该是`somedate.txt`文件的第一行，因为`gets`从标准输入中获取一行数据，而标准输入此时已经变成`somedate.txt`文件的内容。

与之相反，**标准输出**通常是指屏幕或显示器，但如果Ruby脚本的结果要重定向到文件或另一程序，则目的文件或目的程序就变成标准输出的目标。

另外，可以使用`readlines`方法一次读入多行内容：

```
lines =.readlines
```

`readlines`方法接受一行行地输入内容，直到遇到终止符，最常用的终止符是EOF (End Of File文件末尾)。在大多数平台上都可以按Ctrl+D创建EOF。当遇到终止行时，所有输入的行都被放到赋值给`lines`的数组中。对于从标准输入接收管道或重定向数据的程序来说，这种方式是理想选择。例如，假定有个名为`linecount.rb`的脚本，内容只有一行：

```
puts.readlines.length
```

而且你把包含十行内容的文本文件传递给它：

```
ruby linecount.rb < textfile.txt
```

将得到如下结果：

```
10
```

但在现实情况中，这种方法极少使用，除非是编写UNIX提示符所用的shell脚本。在大多数情况下，你会直接读写文件，需要用到`gets`来获取键盘输入的情况很少。

9.1.2 文件输入输出

在第4章，我们用到了`File`类来打开文本文件，以便读入其内容，供程序处理。`File`类一般用作抽象类，实现对Ruby程序可访问的文件对象进行读写和操作。通过`File`类，你既可以写入纯文本文件，也可以写入二进制文件，而且可以利用一组方法来轻松处理文件。

打开和读取文件

最常见的与文件相关的操作，是读取文件的数据供程序使用。如第4章所介绍，这个操作很容易实现：

```
File.open("text.txt").each { |line| puts line }
```

`File`类的`open`方法用来打开文件`text.txt`，而`File`对象的`each`方法每次一行地返回文件内容。也可以这么做：

```
File.new("text.txt", "r").each { |line| puts line }
```

通过该方法可以把过程看得一清二楚。通过打开文件，创建了新的`File`对象。第二个参数“r”定义了以只读方式打开文件。这是默认模式，但用`File.new`，可以用这个模式参数表明你想对文件做什么操作。当在写入文件或从零创建新文件时，这个参数就特别有用。

对于打开和读取文件，`File.new`和`File.open`看起来一模一样，但用途不同。`File.open`可以接受代码块，当代码块运行结束时，文件将自动关闭。而`File.new`仅返回指向文件的`File`对象，要关闭文件，必须用`File`对象的`close`方法。我们来比较一下两个方法，首先来看`File.open`：

```
File.open("text.txt") do |f|
  puts f.gets
end
```

这段代码打开text.txt文件，并把文件句柄以f的形式传递给代码块。

puts f.gets从文件中取出一行数据，并打印输出到屏幕。现在，再看一下File.new的方式：

```
f = File.new("text.txt", "r")
puts f.gets
f.close
```

在本例中，把文件句柄/对象直接赋值给f，最后用close方法手工关闭文件句柄。

代码块方法和文件句柄方法都有各自的用途。代码块方式看起来比较清爽，可以快速打开单个文件并在一个地方进行处理，而用File.new赋值给File对象，则可以在整个当前作用域中引用文件，无须把文件操作代码放在单个代码块中。

注 你可能需要直接指定文件的位置，因为text.txt可能不在当前目录中。只须把f = File.new("text.txt", "r")换成File.new("c:\full\path\here\text.txt", "r")，包含所需的完整路径即可。另外，可以通过Dir::pwd方法的结果，看看当前工作目录中有什么文件，并把text.txt放在此处。

也可以把文件句柄赋值给类变量或实例变量：

```
class MyFile
  attr_reader :handle

  def initialize(filename)
    @handle = File.new(filename, "r")
  end

  def finished
    @handle.close
  end
end

f = MyFile.new("text.txt")
puts f.handle.gets
f.finished
```

这里只是概念验证，但确实证明了File.new在某些情况下有更大的用处。

读取文件的更多方法

在上节中，我们用File对象的each方法，在代码块中一行行读取文件内容。但你可以做更多的操作，我们假定text.txt文件中包含下面这些样例数据：

```
Fred Bloggs, Manager, Male, 45
Laura Smith, Cook, Female, 23
Debbie Watts, Professor, Female, 38
```

下面我们来看看另外一些读取文件的方法，以及相应的输出结果。首先，你可以用`each`方法一行行地读取I/O流：

```
File.open("text.txt").each { |line| puts line }
```

```
Fred Bloggs,Manager,Male,45
Laura Smith,Cook,Female,23
Debbie Watts,Professor,Female,38
```

注 从技术上说，`each`方法文件根据定界符读取文件，而标准定界符是“换行”符。也可以修改这个定界符，详情请参见附录B.4节。

你可以用自定义的定界符，通过`each`读取I/O流：

```
File.open("text.txt").each(',') { |line| puts line }
```

```
Fred Bloggs,
Manager,
Male,
45
Laura Smith,
Cook,
Female,
23
Debbie Watts,
Professor,
Female,
38
```

在本例中，向`each`传递一个可选参数，指定与默认“换行”符不同的定界符。这里用逗号分隔输入内容。

提示 你可以用设置特殊变量`$/`的方法，来把默认定界符覆写为你选择的任何定界符。

也可以用`each_byte`方法，逐字节地读取I/O流：

```
File.open("text.txt").each_byte { |byte| puts byte }
```

```
70
114
101
100
...many lines skipped for brevity...
51
56
10
```

注 当逐字节地读取数据时，每次得到的是每个字符的字节值，而不是字符本身，很

像类似puts "test"[0]的结果。要把它再转回文本字符，可以用chr方法。

下面是用gets方法，逐行地读取I/O流：

```
File.open("text.txt") do |f|
  2.times { puts f.gets }
end
```

```
Fred Bloggs,Manager,Male,45
Laura Smith,Cook,Female,23
```

gets方法不像each或each_byte那样，它不是迭代子，因此你必须多次调用它，才能得到多行数据。在本例中调用了两次，并把示例文件的开头两行读出。但和each方法一样，gets也接受可选的定界符参数：

```
File.open("text.txt") do |f|
  2.times { puts f.gets(',') }
end
```

```
Fred Bloggs,
Manager,
```

each_byte方法也有一个非迭代版本，名为getc：

```
File.open("text.txt") do |f|
  2.times { puts f.getc }
end
```

```
70
114
```

也可以用readlines方法把整个文件读入到数组中，按行分隔：

```
puts File.open("text.txt").readlines.join("--")
```

```
Fred Bloggs,Manager,Male,45
--Laura Smith,Cook,Female,23
--Debbie Watts,Professor,Female,38
```

最后但并非最不重要的一点，你可以用read方法，从文件中读取任意个字节放到某个变量中：

```
File.open("text.txt") do |f|
  puts f.read(6)
end
```

```
Fred B
```

注 所有这些方法可以用于任何文件，例如二进制文件（图像、可执行程序等），不仅仅是文本文件。不过，在Windows中，可能需要以二进制模式打开文件，详见下面“写

入文件”小节。

File类提供了一些方便的方法，要把文件内容读入字符串，你无须调用File.open(filename).read，而是像下面这样：

```
data = File.read(filename)
```

这是用标准read方法打开文件并随即关闭的快捷方式。

也可以这样做：

```
array_of_lines = File.readlines(filename)
```

太简单了！

一般来说，你应该尽量使用这些快捷方法，因为这样的代码更简短、更易读，也无须操心关闭文件。一切都在一步完成。当然，如果需要逐行读取文件（例如，假设要处理极其庞大的文件），你就可以用本章前文演示的逐行读取方法。

在文件中的指针位置

当读取文件时，如果能知道在文件中当前所处的位置，是很有用的事情。pos方法让你可以访问这一信息：

```
f = File.open("text.txt")
puts f.pos
puts f.gets
puts f.pos
```

```
0
Fred Bloggs,Manager,Male,45
28
```

从显示结果可知，在开始从文件读取任何文本之前，位置是0。一旦读入一行文本，位置即变为28。这是因为pos方法返回文件指针的位置（即当前从文件哪个位置开始读取），其单位是字节，从文件开头起算。

但pos可以双向操作，因为它有个姊妹方法pos=：

```
f = File.open("text.txt")
f.pos = 8
puts f.gets
puts f.pos
```

```
ggs,Manager,Male,45
28
```

本例在读取之前，先把文件指针放到第8个字节位置，这样一来就跳过了“Fred Blo”，只读取该行剩下的内容。

写入文件

跳过文件内容、根据定界符读取数据行、逐字节处理数据，这些让Ruby成为数据处理的理想工具，但我还未介绍怎样把新信息写到文件中，或对现有文件进行修改。

一般来说，在写入文件时，可以把大多数文件读取的技术方法进行镜像。例如：


```
File.open("text.txt", "w") do |f|
  f.puts "This is a test"
end
```

这段代码创建了名为text.txt的新文件（或覆写了现有文件），并把一行文本放入其中。前文单独用过puts方法，把数据输出到屏幕，而对File对象使用该方法，则是把数据写入到文件中。太简单了！

传递给File.open的第二个参数"w"，告诉Ruby以只写方式打开文件，创建新文件或覆写已有文件。这与"r"模式正好相反，它是以只读方式打开文件。

不过，你还可以用几种不同的文件模式，如表9-1所示。

表9-1 File.new可用的文件模式

文件模式	输入输出流的属性
r	只读。文件指针放在文件开头
r+	可读可写。文件指针放在文件开头
w	只写。创建新文件（或覆写旧文件）
w+	可读可写，但File.new从零创建新文件（或覆写旧文件）
a	写（以附加模式）。文件指针放在文件末尾，写操作将使文件增长
a+	可读可写（以附加模式）。文件指针放在文件末尾，写操作将使文件增长
b	二进制文件模式（仅Window需要）。可与上述其他模式并用

使用表9-1所述的附加模式，可以很简单地创建一个程序，每次运行即向某个文本文件附加一行：

```
f = File.new("logfile.txt", "a")
f.puts Time.now
f.close
```

如果把这段代码运行几次，logfile.txt将包含一个接一个的日期和时间。附加模式特别适用于日志文件的情况，此时新信息需要在不同时间加入进来。

读和写模式的使用方式简单。如果想以既可读又可写的方式打开文件，则可以使用如下代码：

```
f = File.open("text.txt", "r+")
puts f.gets
f.puts "This is a test"
puts f.gets
f.close
```

第二行代码从文件中读取第一行内容，这表示文件指针位于第二行数据的开头等待操作。而后续的f.puts语句则在文件该位置插入新的文本行，把原来第二行内容推到第三行。接着，再读取下一行内容，即此时的第三行文本。

puts方法用来输出文本行，而与getc和read方法等价的写操作是putc和write方法，用于输出字符和字节：

```
f = File.open("text.txt", "r+")
f.putc "X"
f.close
```

本例打开text.txt文件供读写，并把第一行的第一个字符改为x。同样，下面的代码按字节输出：

```
f = File.open("text.txt", "r+")
f.write "123456"
f.close
```

本例把第一行的前6个字符覆写为123456。

注 值得注意的是，putc和write方法只覆写文件的现有内容，而不是像puts方法那样插入新内容。

改名和删除文件

如果想修改文件名，可以用新名字创建新文件，并把原文件所有数据读入新文件。但不需要这样做，只须使用File.rename方法即可，如下所示：

```
File.rename("file1.txt", "file2.txt")
```

删除文件同样简单。你可以一次删除一个或多个文件：

```
File.delete("file1.txt")
File.delete("file2.txt", "file3.txt", "file4.txt")
File.unlink("file1.txt")
```

注 File.unlink所执行的操作与File.delete完全相同。

文件操作

File类提供了比读写文件更多的能力，也可以对文件进行许多检查和操作。

检查相同文件

检查两个文件是否相同的方法很简单：

```
puts "They're identical!" if File.identical?("file1.txt", "file2.txt")
```

创建平台无关的文件名

Windows和UNIX类操作系统采用了不同的方法来表示文件名。Windows文件名类似c:\directory\filename.ext，而UNIX类操作系统文件名类似/directory/filename.ext。如果你的Ruby脚本对文件名进行操作，并同时在两种操作系统中运行，则需要用到File类提供的join方法。

在两种操作系统中，文件名（和完整路径）都是由目录名和本地文件名组合而成。例如，在上面的例子中，目录名为directory，而Windows的反斜杠与UNIX的正斜杠恰好相反。

注 在Ruby的最近版本中，可以使用UNIX式路径名，即使用正斜杠作为目录分隔符，而不必非要按Windows风格在文件名中使用反斜杠。不过，本节的内容仅供参考备查，或用于某些不认同在其他操作系统中使用UNIX式路径名的程序库。

在Windows中，可以用File.join方法，把目录名和最终文件名合成为文件名：

```
File.join('full', 'path', 'here', 'filename.txt')
```

```
full\path\here\filename.txt
```

注 根据系统安装方式的不同,甚至可能看到上述Windows代码的正斜杠版本,尽管从技术上说,它是UNIX风格的路径。

在UNIX类操作系统中,例如Linux,这段代码仍然不变:

```
File.join('full', 'path', 'here', 'filename.txt')
```

```
full/path/here/filename.txt
```

`File.join`方法用起来很简单,你可以为两种系统编写同样的代码,无须在代码中选择反斜杠和正斜杠。

分隔符本身保存在名为`File::SEPARATOR`的常量中,因此你可以把文件名转换成绝对文件名(即包含绝对路径),方法是把目录分隔符放到开头,如下所示:

```
File.join(File::SEPARATOR, 'full', 'path', 'here', 'filename.txt')
```

```
full/path/here/filename.txt
```

同样,你可以用`File.expand_path`方法,把基本文件名转换成完整路径名。例如:

```
File.expand_path("text.txt")
```

```
/Users/peter/text.txt
```

注 `File.expand_path`的结果将因程序所在操作系统的不同而异。因为`text.txt`是个相对文件名,它被转换成绝对路径名,指向当前工作目录。

搜索

在上例中,用`pos=`修改了文件指针的位置。但这种方法只能指定文件指针的具体位置,如果想把指针向前移动一定距离,或移到距离文件末尾一定位置的地方,则需要使用`seek`方法:

`seek`有三种操作模式:

- `IO::SEEK_CUR`: 从当前位置向前移动若干字节。
- `IO::SEEK_END`: 移动到以文件末尾为基准的某个位置。这表示从末尾开始搜索,可能需要使用负数。
- `IO::SEEK_SET`: 移动到文件的绝对位置。与`pos=`完全相同。

因此,要把文件指针移动到从文件末尾倒数的第5个字节,并把该字符改为X,则应按下面的方法使用`seek`方法:

```
f = File.new("test.txt", "r+")
f.seek(-5, IO::SEEK_END)
f.putc "X"
f.close
```

注 请注意，因为你是写入文件，因此要用r+文件模式，这种模式既可读也可写。

或可以用它实现每5个字符打印输出一次：

```
f = File.new("test.txt", "r")
while a = f.getc
  puts a.chr
  f.seek(5, IO::SEEK_CUR)
end
```

找出文件上次修改时间

要确定文件的上次修改时间，可用File.mtime方法：

```
puts File.mtime("text.txt")
```

```
Fri Jan 11 18:25:42 2007
```

返回的时间是以Time对象的形式，因此可以直接得到更多信息：

```
t = File.mtime("text.txt")
puts t.hour
puts t.min
puts t.sec
```

```
18
25
42
```

注 关于Time类的更多信息及其方法，请参阅第3章。

检查文件是否存在

检查文件是否真实存在是很有用的，特别是当程序依赖于该文件或文件名是由用户提供时。如果文件不存在，则可以抛出用户友好的出错或异常。调用File.exist?方法，可以检查文件是否存在：

```
puts "It exists!" if File.exist?("file1.txt")
```

如果文件存在，则File.exist?返回true。也可以修改上例创建的MyFile类，在打开文件之前检查它是否存在，以避免可能抛出的异常，如下所示：

```
class MyFile
  attr_reader :handle

  def initialize(filename)
    if File.exist?(filename)
      @handle = File.new(filename, "r")
    else
      return false
    end
  end
end
```

得到文件大小

`File.size`方法以字节为单位返回文件大小。如果文件不存在则抛出异常，因此有必要用`File.exist?`先检查文件是否存在。

```
puts File.size("text.txt")
```

怎样知道已到文件末尾

在上面的例子中，要么用循环迭代给出文件的所有行或字节，要么只从文件的这里或那里取出几行。但如果能有十分简单的方法来得知文件指针是否到达或越过文件末尾，将是非常有用的。`eof?`方法提供这一功能：

```
f = File.new("test.txt", "r")
catch(:end_of_file) do
  loop do
    throw :end_of_file if f.eof?
    puts f.gets
  end
end
f.close
```

本例使用了“无限”循环，只能通过`catch`和`throw`（详见第8章）来跳出循环。仅当文件指针到达或越过文件末尾时，才调用`throw`。这个特别的例子不是特别有用，因为`f.each`可以执行与之类似的任务。但当你需要手工移动文件指针时，或在文件中作大范围跳跃时，对“文件末尾”进行检查则是非常有用的。

目录

所有文件都包含在不同的目录中，Ruby对处理目录也毫无问题。`File`类用来处理文件，目录则由`Dir`类来处理。

在目录中移动

要在Ruby程序中改变目录位置，可用`Dir.chdir`方法：

```
Dir.chdir("/usr/bin")
```

本例把当前目录改变到`/usr/bin`。

用`Dir.pwd`方法可以找出当前目录所在位置。例如，下面是在我机器上的运行结果：

```
puts Dir.pwd
```

```
/Users/peter
```

```
Dir.chdir("/usr/bin")
puts Dir.pwd
```

```
/usr/bin
```

你可以用`Dir.entries`方法，得到特定目录中的文件和目录列表：

```
puts Dir.entries("/usr/bin").join(' ')
```

```
...a2p aclocal aclocal-1.6 addftinfo afmtodit alias amlint ant appleping
```



```
appletviewer apply apropos apt ar arch as asa at at_cho_prn atlookup atos
atprint ...items removed for brevity...zless zmore znew zprint
```

Dir.entries方法返回一个数组，其中包含特定目录下的所有内容。Dir.foreach提供了相同的功能，但是以迭代子的方式：

```
Dir.foreach("/usr/bin") do |entry|
  puts entry
end
```

获取目录列表的更简洁方法是用Dir类的数组方法：

```
Dir["/usr/bin/*"]
```

```
["/usr/bin/a2p", "/usr/bin/aclocal", "/usr/bin/aclocal-1.6",
"/usr/bin/addftinfo",
"/usr/bin/afmtodit", "/usr/bin/alias", "/usr/bin/amlint", "/usr/bin/ant",
...items
removed for brevity...]
```

在本例中，每个条目都以绝对文件名形式返回，以便使用File类的方法按自己的意愿对每个条目进行检查。

你可以把这个过程再推进一步，变得更加与平台无关：

```
Dir[File.join(File::SEPARATOR, 'usr', 'bin', '*')]
```

注 当然，只有UNIX系统有/usr/bin目录，因此在此情况下这种方法未必能用，但在自己的程序中可能有用。

创建目录

要创建目录，可以用Dir.mkdir方法，如下所示：

```
Dir.mkdir("mynewdir")
```

目录创建好之后，可以用Dir.chdir移动到该目录。

在创建目录时，也可以指定其他目录下的绝对路径：

```
Dir.mkdir("/mynewdir")
Dir.mkdir("c:\\test")
```

不过，无法在不存在的目录之下创建目录。如果想创建整个目录结构，必须从顶向下逐个创建。

注 在UNIX类操作系统中，Dir.mkdir接受第二个可选参数：是个整数，用来指定对目录的操作权限。你可以用八进制指定该值，例如0666或0777，分别表示666和777模式。

删除目录

删除目录与删除文件很相似：

```
Dir.delete("testdir")
```

注 Dir.unlink和Dir.rmdir执行完全相同的功能，提供这个方法，是为了方便目的。

和Dir.mkdir一样，可以使用绝对路径名。

在删除目录时需要考虑一件事，即目录是否为空。如果目录不为空，则只用Dir.delete无法删除该目录。你需要先循环迭代每个子目录，并删除其中的文件和目录。可以用Dir.foreach来完成这一操作，对整个文件树进行递归循环，把要删除的目录和文件放在数组中。

在临时目录中创建文件

大多数操作系统都有“临时”目录的概念，供临时文件在此存放。临时文件是在程序执行过程中简单创建，不用作永久保存信息的用途。

Dir.tmpdir方法提供了当前操作系统的临时目录路径，不过该方法默认不可用。要令其可用，必须用require 'tmpdir'命令：

```
require 'tmpdir'
puts Dir.tmpdir
```

```
/tmp
```

你可以把Dir.tmpdir和File.join一起使用，以平台无关的方式创建临时文件：

```
require 'tmpdir'
tempfilename = File.join(Dir.tmpdir, "myapp.dat")
tempfile = File.new(tempfilename, "w")
tempfile.puts "This is only temporary"
tempfile.close
File.delete(tempfilename)
```

这段代码创建了一个临时文件，向其中写入数据，并删除该文件。

Ruby的标准程序库还包含名为Tempfile的程序库，可用来创建临时文件：

```
require 'tempfile'
f = Tempfile.new('myapp')
f.puts "Hello"
puts f.path
f.close
```

```
/tmp/myfile1842.0
```

与自己创建并管理临时文件相比，Tempfile有所不同，它会自动删除使用完毕的临时文件。在对两种方法作出选择时，这是个重要的考虑因素（关于临时文件和tempfile程序库的更多信息，详见第16章）。

9.2 数据库基础

许多应用程序需要保存、访问或操作数据。在某些情况下，通过载入文件、修改数据并把数据输出到屏幕或写回文件。但在许多情况下，需要使用数据库。

数据库是对计算机中数据进行系统化组织管理的系统。数据库可以简单到是个文本文件，

其中包括可程序化操作的数据，或复杂到有許多GB的数据，分布在上百个专用数据库服务器中。对于二者之间各种规模的数据库，Ruby都能游刃有余。

首先，我们要了解一下，怎样使用简单文本文件作为有组织数据的保存形式。

9.2.1 文本文件数据库

简单类型的数据库可以保存在CSV格式的文本文件中。CSV是逗号分隔值（Comma-Separated Values）的缩写，表示其中保存的每个数据条目都有多个属性，分别以逗号分隔。上节的样例数据文件text.txt就是使用CSV格式来保存数据。我们来回顾一下，text.txt起初包含如下内容：

```
Fred Bloggs,Manager,Male,45
Laura Smith,Cook,Female,23
Debbie Watts,Professor,Female,38
```

每一行表示不同的人，逗号分隔的属性与每个人相关。逗号让你可以单独读取（和修改）每个属性。

Ruby的标准程序库包含名为csv的程序库，让你可以用包含CSV数据的文本文件作为简单数据库，可以方便地读取、创建和操作。

读取和搜索CSV数据

csv程序库提供了CSV类，用来帮你管理和操作数据：

```
require 'csv'
CSV.open('text.txt', 'r') do |person|
  puts person.inspect
end
```

```
["Fred Bloggs", "Manager", "Male", "45"]
["Laura Smith", "Cook", "Female", "23"]
["Debbie Watts", "Professor", "Female", "38"]
```

在这段代码中，你用CSV.open打开text.txt文件，其中每行（即文件中的每个“人”）都逐次传递给代码块。inspect方法显示出每个条目都以数组形式表示，从而比纯文本形式更容易读取数据。

也可以把CSV和File类共用：

```
require 'csv'
people = CSV.parse(File.read('text.txt'))
puts people[0][0]
puts people[1][0]
puts people[2][0]
```

```
Fred Bloggs
Laura Smith
Debbie Watts
```

本例使用File类来打开并读取文件内容，而CSV.parse则立即用这些内容把数据转换成

由数组构成的数组。主数组的元素表示文件的每行，而每个元素中的子元素则表示该行的不同属性（或字段）。从而通过打印输出每个条目的第一个元素，即可只得到人们的名字。

把数据从CSV格式文件载入到数组的更精炼方法是用CSV.read:

```
puts CSV.read('text.txt').inspect
```

```
[["Fred Bloggs", "Manager", "Male", "45"], ["Laura Smith", "Cook", "Female", "23"],  
["Debbie Watts", "Professor", "Female", "38"]]
```

由Enumerable模块供给Array类的find和find_all方法，让你可以轻松地对数组中的数据进行搜索。例如，如果你想找出第一个名叫Laura的人，可以用下面这段代码：

```
require 'csv'  
people = CSV.read('text.txt')  
laura = people.find { |person| person[0] =~ /Laura/ }  
puts laura.inspect
```

```
["Laura Smith", "Cook", "Female", "23"]
```

用find方法配以代码块，该代码块搜索匹配名字中包含“Laura”的第一行数据，并返回正在搜索的数据。

find方法返回数组或散列表中第一个匹配的元素，而find_all则返回所有合法的匹配元素。我们假定你想在数据库找出年龄在20~40之间的人：

```
young_people = people.find_all do |p|  
  p[3].to_i.between?(20,40)  
end  
puts young_people.inspect
```

```
[["Laura Smith", "Cook", "Female", "23"], ["Debbie Watts", "Professor",  
"Female", "38"]]
```

该操作返回为一个数组，为你提供两个匹配的人，你可以对其进行迭代处理。

把数据保存回CSV文件

你已经能够读取和查询数据，下一步是可以修改它、删除它，并用新版本的数据重写CSV文件，以供未来使用。幸运的是，这个操作就像重新打开文件一样简单，将其设为写权限，并把数据“推”回文件即可。CSV模块帮你处理所有的内部转换。

```
require 'csv'  
people = CSV.read('text.txt')  
laura = people.find { |person| person[0] =~ /Laura/ }  
laura[0] = "Lauren Smith"  
  
CSV.open('text.txt', 'w') do |csv|  
  people.each do |person|  
    csv << person
```

```
end
end
```

在这段代码中，载入数据，找到要修改的人，修改其名字，然后打开CSV文件并把数据重写回去。但请注意，必须一个人一个人地写数据。完成之后，`text.txt`即被更新，其中的人名被修改。这就是怎样把CSV数据写回文件的方法（关于CSV的更多信息，以及CSV的更快速版本——以程序库形式提供的FasterCSV，请参阅第16章）。

9.2.2 对象和数据结构的存储

处理CSV十分容易，但给人的感觉并不非常顺畅。你总要处理数组，而不是用更好用的名字，例如`name`、`age`或`job`来处理不同的属性，因此不得不记住每个属性在数组的哪个位置。

还必须为每个条目保存简单数组。这里没有嵌套，没有办法把东西互相关联起来，并且与面向对象没有关系，另外数据是扁平的。这是处理简单数据的理想工具，但如果想把已经存在的Ruby数据结构（例如数组和散列表）的数据保存到磁盘中以备日后使用，该怎么办？

PStore

PStore是个核心的Ruby程序库，让你可以像使用常规Ruby对象和数据结构一样，将其保存到文件中，以后可以重新把对象从磁盘文件载入到内存。这种技术名为对象持久化（object persistence），它依赖于名为汇集（marshalling）的技术，即把标准数据结构转换成扁平数据的形式，可被保存到磁盘或通过网络传输，供以后重新构造数据结构。

我们创建一个类，来表示CSV示例中用到的数据结构：

```
class Person
  attr_accessor :name, :job, :gender, :age
end
```

你可以用下面的代码重新创建数据：

```
fred = Person.new
fred.name = "Fred Bloggs"
fred.age = 45

laura = Person.new
laura.name = "Laura Smith"
laura.age = 23
```

注 为简略起见，本例中你只对这两个对象进行处理。

现在无须把数据放到数组中，而是以完全面向对象的形式表示数据。也可以在`Person`类中创建方法以便操作对象。这样保存和操作数据的方式，是真正的Ruby方式，也是完全面向对象的。不过，到目前为止，你的对象只能生存到程序结束，而有了PStore则可以轻松地把它们写入到文件中：

```
require 'pstore'
store = PStore.new("storagefile")
store.transaction do
```



```

store[:people] ||= Array.new
store[:people] << fred
store[:people] << laura
end

```

在本例中，在名为`storagefile`的文件中，创建了新的PStore，然后开始事务处理（PStore文件中的数据只能在“事务”中被读取或更新，以防止数据损毁），在事务处理中，你要确保仓库的`:people`元素有内容，或被赋值给数组。接着，你把`fred`和`laura`对象推送到仓库的`:people`元素中，然后结束事务处理。

采用散列表式语法的原因，是因为PStore实际上是基于磁盘的散列表。因此可以把任何对象保存在散列表之中。在本例中，`store[:people]`创建了一个数组并把两个Person对象推送进来。

以后就可以从PStore数据库中检索数据：

```

require 'pstore'
store = PStore.new("storagefile")
people = []
store.transaction do
  people = store[:people]
end

#At this point the Person objects inside people can be treated
#as totally local objects.
people.each do |person|
  puts person.name
end

```

```

Fred Bloggs
Laura Smith

```

只需一个简单的存储和检索过程，PStore就为现有Ruby程序增加了简便的存储能力，让你可以把现有对象保存到PStore数据库中。对于许多类型的数据存储来说，对象持久化并非理想选择，但如果你的程序严重依赖于这些对象，而且你想把这些对象保存到磁盘供以后使用，那么PStore就提供了一种简单的方案。

YAML

YAML（是“YAML不是标记语言”的缩写）是一种特殊的基于文本的标记语言，用作一种可供人们阅读的数据序列化格式。可以用与PStore类似的方法把数据结构序列化，但与PStore数据不同的是，人们可以很容易阅读YAML数据，甚至可以用文本编辑器直接编辑（只需一点基本的YAML语法知识）。

YAML程序库是Ruby标准程序库的组成部分，因此很容易使用。但与PStore不同，YAML程序库对数据结构和YAML进行相互转换，但是不提供散列表，因此其技术略有不同。下面的例子展示了怎样把对象数组写入磁盘：

```
require 'yaml'
```

```

class Person
  attr_accessor :name, :age
end

fred = Person.new
fred.name = "Fred Bloggs"
fred.age = 45

laura = Person.new
laura.name = "Laura Smith"
laura.age = 23

test_data = [ fred, laura ]

puts YAML::dump(test_data)

```

```

---
-!ruby/object:Person
  age:45
  name:Fred Bloggs
-!ruby/object:Person
  age:23
  name:Laura Smith

```

在上面的代码中，用YAML::dump方法把Person对象数组转换成YAML数据，这些数据如你所见的那样，是极其可读的！YAML::load方法执行另一方向的操作，把YAML代码转换成可用的Ruby对象。例如，我们对YAML数据略作修改，看它是否能转回可用对象：

```

require 'yaml'

class Person
  attr_accessor :name, :age
end

yaml_string = <<END_OF_DATA
---
-!ruby/object:Person
  age:45
  name:Jimmy
-!ruby/object:Person
  age:23
  name:Laura Smith
END_OF_DATA

test_data = YAML::load(yaml_string)
puts test_data [0].name
puts test_data [1].name

```

PDF

```
Jimmy
Laura Smith
```

这里`YAML::load`方法成功地把YAML数据转回Person对象组成的`test_data`数组。

你可以使用YAML，在大多数种类的Ruby对象（包括基本类型，例如Array和Hash）和YAML之间来回转换。这让它成为保存应用程序所需数据（例如配置文件）的理想中间格式。

注 当处理序列化对象时，你必须仍然拥有这些对象使用的类，无论定义在程序的什么地方，否则这些对象将无法使用。

作为纯文本，YAML可以安全地通过电子邮件传输、以正常文本文件保存，比程序库（例如PStore）创建的二进制数据更容易在各处转移。

要了解关于YAML格式的更多信息，请参阅其维基百科条目<http://en.wikipedia.org/wiki/YAML>，或访问YAML官方网站<http://www.yaml.org/>。

9.3 关系型数据库与SQL

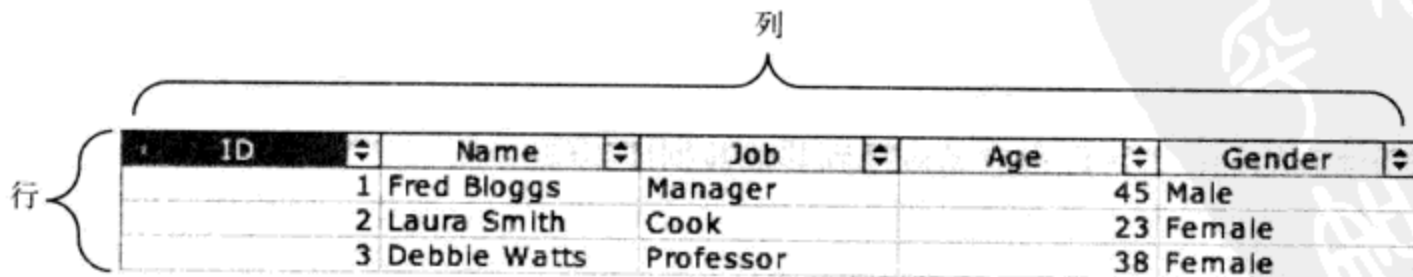
在上一节中，用文本文件和对象持久化创建了极其简化的数据库。文本文件当然有其局限性。如果许多进程同时使用文件，则文件是不可靠的。另外文件速度较慢，如果数据量不大，把CSV文件载入到内存的速度尚可接受，但当数据量增长时，直接处理文件的速度很快就会变得迟钝。

在开发更健壮的系统时，把数据库归档和管理的职能都移交给单独的程序或系统，你的应用程序只需连接到此数据库系统，来回传递数据即可。在上一节我们直接操作了数据库文件以及其中的数据，当性能和可靠性成为重要因素时，这种做法是不可接受的。

9.3.1 关系型数据库概念

使用专有数据库系统的一大主要好处是可以得到关系型数据库的支持。关系型数据库的数据按一个或多个表分组，这些表可以连接起来，每个表用来保存某一类事物的信息。例如，地址本数据库可由人（people）表、地址（addresses）表和电话号码（phonenumber）表组成，每个表分别保存关于人、地址和电话号码的信息。

people表可以有許多属性（在数据库领域，称为列），例如name、age和gender。表的每一行（即每个人）在每一列都有相关信息，如图9-1所示。



ID	Name	Job	Age	Gender
1	Fred Bloggs	Manager	45	Male
2	Laura Smith	Cook	23	Female
3	Debbie Watts	Professor	38	Female

图9-1 包含三个列的简单people表

图9-1的例子还包含名为id的列。这是关系型数据库的标准规程，大多数表都有id列，以便

唯一标识每行数据。尽管你可以根据其他列，例如name进行数据检索，但数值型ID在创建表间关系时很有用。

注 在图9-1中，表头是以一般方式书写的，如在常规地址本或地址表中看到的那样。但当在底层处理关系型数据库时，列名和表名一般全部都用小写字母。这也是本章的文字和代码示例中，只用小写字母指称表名和列名的原因。

使用关系型数据库的一大好处，是不同表中的行可以相互关联起来。例如，people表有个address_id列，保存了该用户地址的ID。如果想找到某人的地址，可以搜索其address_id，然后再到addresses表中查看对应的行。

采用这种关系的原因，是people数据库中的许多人可能共享相同的地址，而不是为每个人分别保存地址，这样只保存对地址的引用会更有效率。这也意味着如果将来更新地址信息，同时也为所有相关用户同时进行了更新。

关系也可以实现多对多关系。可以创建一个单独的表，名为related_people，其中有两列，分别是first_person_id和second_person_id。这个表可以保存成对ID，表示两个人相互关联。要查出某人与谁关联，只需搜索有这人ID的行，即可得到关联人员的ID。大多数数据库都使用这种关系，这也是关系型数据库如此有用的原因。

9.3.2 四大数据库：MySQL、PostgreSQL、Oracle和SQLite

当今有4种广为人知的数据库系统，可在Windows和UNIX操作系统中使用，分别是MySQL、PostgreSQL、Oracle和SQLite。其中每种数据库都与其他数据库有相当不同的特点，因此各有不同的用途。

注 微软SQL Server也很流行，不过只用于微软平台。

大多数Web开发人员都熟悉MySQL，因为它是大多数Web主机软件包和服务器自带的。因此，MySQL是因特网最常用的数据库引擎。它也是Ruby on Rails框架（详见第13章）的默认数据库引擎，因此你很可能会用到它。

PostgreSQL和Oracle也有它们自己的位置，Oracle是广为人知的企业级数据库，许可证费用颇高，但许多大型企业都使用它。

为了更好地掌握本章以下几节的内容，我们将使用名为SQLite的系统。与MySQL、PostgreSQL和Oracle不同，SQLite不是以“服务器在”形式运行，因此不需要任何特别资源。虽然MySQL、PostgreSQL和Oracle都以永久服务器程序形式运行，但SQLite是“按需应变”的系统，完全在本机运行。尽管如此，它仍然速度飞快，性能可靠，是局部数据库的理想选择。从SQLite所学到的知识，可以很容易应用到其他数据库系统。

不过到本章结束时，我们将介绍怎样连接到其他架构的数据库，这样你可以从Ruby程序直接访问现有数据库。

9.3.3 安装SQLite

让数据库系统快速安装运行的第一步是安装SQLite3——SQLite的最新版本。SQLite的下载

网页<http://www.sqlite.org/download.html>包含SQLite3程序库（Windows版为DLL，Linux版为共享库）的二进制下载版本，以及可在其他操作系统上进行编译的源代码。

Mac OS X DarwinPorts的用户可以在命令行用`sudo port install sqlite3`命令安装SQLite3，而某些Linux版本的用户可以用相应的包管理器安装SQLite3。

注 对于Windows用户，可参考<http://blip.tv/file/48664>关于SQLite3安装过程的屏幕录像视频。

SQLite3程序库安装完毕后，就可以安装gem包形式的Ruby程序库，以便可以访问SQLite3数据库。这个gem包名为`sqlite-ruby`，可用`gem install sqlite3-ruby`命令安装，如在UNIX类操作系统中，则以超级用户身份用`sudo install sqlite3-ruby`命令安装（关于安装Ruby gem包的详细信息，请参阅第7章）。

你可以用下面这段代码来检查是否一切安装正常：

```
require 'rubygems'
require 'sqlite3'
puts "It's all okay!" if defined?(SQLite3::Database)
```

```
It's all okay!
```

如果在安装过程中碰到问题，请访问附录C中提供的SQLite资源。

9.3.4 关于数据库基本操作和SQL的紧急教程

要使用任何一种数据库系统对数据库进行简单管理，具备一定的SQL命令知识是必不可少的。本节我们将介绍怎样创建表、向其中增加数据、检索数据、删除数据和修改数据。

在阅读本节的过程中，请把整个数据库想成是与Ruby分开的。关于Ruby怎样使用SQL来操作数据库，将在9.3.5节介绍。

注 如果你已经熟悉SQL，可以跳过后续几节内容，直接阅读9.3.5节，看看SQL是怎样在Ruby中发挥作用的。

SQL是什么

结构化查询语言（Structured Query Language，SQL）是一门特别的语言，常常称为查询语言，用于与数据库系统进行交互。你可以用SQL创建、检索、更新和删除数据，也可以创建和操作用来容纳数据的结构。其基本用途是支持客户端与数据库系统的交互。本节将介绍SQL入门级语法，以及怎样在Ruby中使用SQL。

请注意本节只是对SQL非常简单的介绍，因为对SQL作完整和深入的讲解超出了本书的范围。如果想了解更深入的SQL知识，请参阅附录C提到的相关资源。

请注意，不同数据库系统对SQL的使用和实现可能相差很大，这也是为什么下面章节只介绍标准语法，仅供进行基本数据操作。

CREATE TABLE（创建表）语句

在把数据放到数据库之前，必须先创建一个或多个表来容纳数据。要创建表，需要知道要

保存什么数据，怎样称呼这些数据，以及要保存数据的哪些属性。

对于people表，有name、job、gender和age列，以及唯一的id列，用于建立与其他表可能有的关系。创建表的语法格式如下：

```
CREATE TABLE table_name (
  column_name data_type options,
  column_name data_type options,
  ...,
  ...
)
```

注 有些数据库系统需要在每个SQL语句末尾加分号，不过本书中的示例都不包含分号。

因此，对于people表，应该用如下语句来创建：

```
CREATE TABLE people (
  id integer primary key,
  name varchar(50),
  job varchar(50),
  gender varchar(6),
  age integer)
```

这个SQL命令创建了people表，其中包含五个列。Name、job和gender列的数据类型都是VARCHAR，表示它们是变长字符字段。用简单的话说，表示它们可以包含字符串。括号中的数字表示这样的字符串可以有的最大长度，因此name列可以容纳最多50个字符。

注 SQLite是相当实用化的数据库，忽略了SQL相关的大多数惯例。几乎任何形式的数据都可以放在任意类型的列中。SQLite忽略这些VARCHAR字段的最大长度，这也是SQLite非常适合于快速轻松开发的原因之一，但不非常适合于至关重要的系统！

id列标注了primary key选项，这表示id列是对每行数据的主要引用，而且每行的id值必须是唯一的。在SQLite中，这表示SQLite将自动为每行数据赋予唯一的id值，因此在增加新行时，你无须每次自己指定id值。

INSERT INTO（插入数据）语句

你可以用INSERT命令向表中插入数据行：

```
INSERT INTO people (name,age,gender,job)VALUES ("Chris Scott",25,"Male",
"Technician")
```

首先指定要增加数据行的表，然后列出要填写的列，最后再提供要填写该行的值。如果在VALUES之后的数据是按正确顺序排列的，就可以忽略列名的清单。

```
INSERT INTO people VALUES ("Chris Scott", 25, "Male", "Technician")
```

警告 这个特别的INSERT语句在people表将导致错误！因为它漏掉了id列。

不过，更安全更方便的方法是手工指定列名，如第一个例子所示。第二个例子清楚地展示了原因，因为很难搞清楚每个数据项与哪个列相关。

未指定值的列将自动以CREATE TABLE语句中指定的默认值填充。对于本例的people表，id值将自动填充，并为每个新加的行提供唯一的ID值。

SELECT (选择数据) 语句

可以用SELECT语句来从表中检索数据，需要指定要检索哪个列（或用*作为通配符，表示检索所有列），要从哪个表中检索数据，并包含可选的条件，可以指定根据什么进行检索。例如，可以选择一个或多个匹配某个条件的特定数据行。

下面这条SQL语句从people表中检索所有行和所有列的数据：

```
SELECT * FROM people
```

下面这条SQL语句从people表中只检索所有行的name列（例如“Fred Bloggs”、“Chris Scott”、“Laura Smith”）：

```
SELECT name FROM people
```

下面这条SQL语句从people表中检索id列等于2的行（一般来说，由于id是含有唯一值的列，这样的查询只返回一行数据）：

```
SELECT * FROM people WHERE id = 2
```

下面这条SQL语句检索name列等于“Chris Scott”的行：

```
SELECT * FROM people WHERE name = "Chris Scott"
```

下面这条SQL语句检索年龄在20~40（含20和40）所有人的数据：

```
SELECT * FROM people WHERE age >= 20 AND age <= 40
```

SQL语句所用条件有些类似Ruby和其他编程语言，区别在于AND和OR等逻辑运算符以纯英文方式书写。而且和Ruby一样，也可以用括号把表达式进行分组，组成更复杂的请求语句。

可以用ORDER BY语句，把返回的结果以某种顺序排列，例如SQL查询中的ORDER BY column_name。你还可以进一步在列名后附加ASC，表示以增序排列，或附加DESC，表示以降序排列。例如，下面这行SQL语句从people表返回所有行，返回结果按name列降序排列（因此以Z开头的名字排在以A开头名字的前面）。

```
SELECT * FROM people ORDER BY name DESC
```

下面这条SQL语句返回所有年龄在20到40之间的人，返回结果按年龄顺序排列，年轻者在前：

```
SELECT * FROM people WHERE age >= 20 AND age <= 40 ORDER BY age ASC
```

SELECT命令的另一个有用选项是LIMIT。LIMIT让你可以设置单次查询返回的最多行数：

```
SELECT * FROM people ORDER BY name DESC LIMIT 5
```

LIMIT与ORDER并用时，可以找出数据的极限值。例如，找出年纪最大的人：

```
SELECT * FROM people ORDER BY age DESC LIMIT 1
```

这个SQL语句把结果按age降序排列，并返回第一个结果，即年龄最大的。要找到最年轻的人，只须把DESC排序改为ASC排序即可。

注 数据库引擎自动根据列的数据类型排序，字符串文本按字母顺序，而整数和其他数字列则按值排序。

DELETE语句

SQL的DELETE命令用来删除表中的行，也可以根据SQL条件进行删除。例如：

```
DELETE FROM people WHERE name="Chris"
DELETE FROM people WHERE age > 100
DELETE FROM people WHERE gender = "Male" AND age < 50
```

和SELECT语句一样，可以对删除的数目进行限制：

```
DELETE FROM people WHERE age > 100 LIMIT 10
```

在本例中，年龄超过100的人，只有10人可被删除。

可以把DELETE命令看成与SELECT很相似，一个是返回行，另一个是删除行，除此以外命令格式非常相似。

UPDATE语句

UPDATE语句提供了更新修改数据库中信息的能力。和DELETE一样，UPDATE的语法与SELECT相似。请看下面的例子：

```
SELECT * FROM people WHERE name = "Chris"
UPDATE people SET name = "Christopher" WHERE name = "Chris"
```

UPDATE首先接受要更新列的表名，然后接受一个或多个要更新的列，以及新数据，最后是可选的条件。下面是一些示例。

下面这条SQL语句找到name列当前值等于“Chris”的所有人，并将其name列都改为“Christopher”：

```
UPDATE people SET name = "Christopher" WHERE name = "Chris"
```

下面这条SQL语句找到name列当前值等于“Chris”的所有人，将其name列都改为“Christopher”，age列都改为44：

```
UPDATE people SET name = "Christopher", age = 44 WHERE name = "Chris"
```

下面这条SQL语句找到name列当前值等于“Chris”并且age列等于25的所有人，将其name列都改为“Christopher”。因此，名为Chris且年龄为21的人不会被这个SQL示例语句修改：

```
UPDATE people SET name = "Christopher" WHERE name = "Chris" AND age = 25
```

下面这条SQL语句把people表每行的name列都改为“Christopher”。从此可以看出，在构造SQL查询时为什么必须小心从事，因为短短的一条语句，可能产生巨大的效果！

```
UPDATE people SET name = "Christopher"
```

9.3.5 在Ruby中使用SQLite

现在已经安装了SQLite，我们也了解了基本的SQL语法，下面我们来简单展示一下怎样在Ruby中综合运用。你需要编写一个程序，操作本章到目前为止讨论的数据库people表。

第一步是编写载入或创建数据库的基本代码。有了SQLite-Ruby的gem包，这个操作很简单，只需使用SQLite3::Database.new方法。例如：

```
require 'rubygems'
require 'sqlite3'
$db = SQLite3::Database.new("dbfile")
$db.results_as_hash = true
```

从此时开始，可以像本章前面用过的文件句柄一样使用\$db对象。例如，正像关闭常规文件那样，\$db.close同样会关闭数据库文件。

\$db.results_as_hash = true这行代码强制SQLite以散列表格式返回数据，而不是放在属性数组中（像CSV那样）。这样返回结果更容易访问。

注 由于数据库句柄被赋值给全局变量\$db，因此你可以把这这段程序分为几个方法，但不必创建类。然后即可在任何地方访问数据库句柄\$db。

要处理关闭数据库的情况，可以创建一个方法，专门用来关闭数据库连接并结束程序：

```
def disconnect_and_quit
  $db.close
  puts "Bye!"
  exit
end
```

注 请记住，方法必须在使用之前定义，因此要把这些单独的方法放到源代码文件的开头。

现在我们来创建一个方法，使用SQL语句CREATE TABLE来创建用于保存数据的表：

```
def create_table
  puts "Creating people table"
  $db.execute %q{
    CREATE TABLE people (
      id integer primary key,
      name varchar(50),
      job varchar(50),
      gender varchar(6),
      age integer)
  }
end
```

数据库句柄让你可以用execute方法执行任意的SQL语句。你要做的只是把SQL语句以参数形式传递给它，SQLite即可对数据库执行SQL语句。

下面，我们来创建一个方法，请求用户输入，以便向数据库增加新人数据：

```
def add_person
  puts "Enter name:"
  name =gets.chomp
  puts "Enter job:"
```

```

    job =gets.chomp
    puts "Enter gender:"
    gender =gets.chomp
    puts "Enter age:"
    age =gets.chomp
    $db.execute("INSERT INTO people (name,job,gender,age)VALUES (?,?,?,?)",
        name,job,gender,age)
end

```

注 gets的chomp方法用来把字符串末尾的换行符去掉，这个字符串是用gets方法从键盘得到的。

add_person方法的开头很平常，用于轮流询问每个人的属性，并把它们赋值给变量。但这一次\$db.execute方法更引人注目。在上节中，SQL语句INSERT与主语句的数据一并出现，但在这个方法中，用问号(?)作为数据的占位符。

Ruby执行自动替换，把传递给execute方法的其他参数替换到占位符中。这是保护数据库的一种方法，因为如果把用户输入的内容直接放到SQL语句中，用户可能输入某些破坏性的查询。但如果使用占位符方式，SQLite-Ruby程序库将帮清理数据，确保数据对数据库是安全的。

现在需要访问输入数据的另一个方法该上场了！下面这段代码展示了怎样根据给定的姓名和ID来检索人员数据：

```

def find_person
  puts "Enter name or ID of person to find:"
  id =gets.chomp

  person =$db.execute("SELECT *FROM people WHERE name =?OR
    id =?",id,id.to_i).first

  unless person
    puts "No result found"
    return
  end

  puts %Q{Name:#{person ['name']}}
  Job:#{person ['job']}
  Gender:#{person ['gender']}
  Age:#{person ['age']}
end

```

find_person方法请用户输入待查找人员的姓名或ID，然后\$db.execute则巧妙地实现同时检查name和id列。这样一来，无论id或name匹配都行。如果找不到任何匹配数据，则用户将被告知，该方法提前结束。如果有匹配，则取出匹配人员信息并打印输出在屏幕上。

你可以把这些方法与主程序连接在一起，前文提到该主程序将为这四个方法实现菜单系统的功能。你已经有了数据库连接代码，因此创建菜单很简单：

```

loop do

```



```

puts %q{Please select an option:

1.Create people table
2.Add a person
3.Look for a person
4.Quit}

case gets.chomp
  when '1'
    create_table
  when '2'
    add_person
  when '3'
    find_person
  when '4'
    disconnect_and_quit
end
end

```

如果这段代码输入正确并运行，将显示如下典型的初始画面：

Please select an option:

```

1.Create people table
2.Add a person
3.Look for a person
4.Quit

```

1.

Creating people table

Please select an option:

```

1.Create people table
2.Add a person
3.Look for a person
4.Quit

```

2

Enter name:

Fred Bloggs

Enter job:

Manager

Enter gender:

Male

Enter age:

48

Please select an option:

```

1.Create people table
2.Add a person

```

```

3.Look for a person
4.Quit
3
Enter name or ID of person to find:
1
Name:Fred Bloggs
Job:Manager
Gender:Male
Age:48

Please select an option:

1.Create people table
2.Add a person
3.Look for a person
4.Quit
3
Enter name or ID of person to find:
Jane Smith
No result

```

这个快速开发、功能简单的程序，实现了从远程数据源增加数据和检索数据的方法，只用了几行代码！

9.3.6 连接其他数据库系统

在上一节我们了解了SQL，以及怎样用SQLite程序库来使用SQL，SQLite是为本机提供简单数据库系统的程序库。但更常见的情况是，你也许想连接到更主流的数据库服务器，例如运行着MySQL、PostgreSQL、MS SQL Server或Oracle的服务器。本节我们将简要介绍怎样连接到这些种类的数据库。

注 有个名为DBI程序库，提供了Ruby与数据库系统的通用接口。从理论上说，用DBI编写的程序可以访问任何数据库，并可轻松地在不同数据库之间切换。但在实际使用中并不总是这样。不过学习怎样使用DBI，将来可以使用MySQL、PostgreSQL和Oracle进行访问。附录C提供了一系列DBI资源的链接。

MySQL

MySQL是Web主机服务提供商最常用的数据库系统，这是因为MySQL是开源的品种，可以免费下载和使用。MySQL还有容易使用的赞誉，并已经形成了一个大型生态系统。

Ruby对MySQL的支持是通过MySQL程序库实现的，对UNIX和Windows均以RubyGem包形式提供。要安装这个程序库，请用以下代码：

```
gem install mysql
```

注 在OS X中安装mysql的gem包，说着容易做着难。可能发生许多问题，如<http://>

www.caboo.se/articles/2005/08/04/installing-ruby-mysql-bindings-2-6-on-tiger-troubleshooting和<http://bugs.mysql.com/bug.php?id=23201>所述。如果第一次没有安装成功，可以上网搜索，总有解决办法！

mysql的gem包向Ruby提供了一个类和一些方法，可用来连接到某个已安装的MySQL服务器。它本身不包含MySQL服务器！如果你没有可连接的MySQL服务器，需要向Web主机服务提供商咨询，或在你本机安装某个版本的MySQL。

mysql的gem包安装完毕后，从Ruby连接并使用MySQL服务器，几乎像SQLite数据库一样简单：

```
require 'rubygems'
require 'mysql'

#Connect to a MySQL database 'test' on the local machine
#using username of 'root' with no password.
db = Mysql.connect('localhost','root','','test')

#Perform an arbitrary SQL query
db.query("INSERT INTO people (name,age)VALUES('Chris',25)")

#Perform a query that returns data
begin
  query = db.query('SELECT *FROM people')

  puts "There were #{query.num_rows} rows returned"

  query.each_hash do |h|
    puts h.inspect
  end
rescue
  puts db.errno
  puts db.error
end

#Close the connection cleanly
db.close
```

这段代码展示了简单的任意SQL查询，以及返回数据的查询（以一行行数据的散列表格式），它还通过rescue代码块捕获异常，而是提供了基本的出错反馈机制，并使用MySQL程序库的错误检索方法。

注 你也可以用DBI程序库的数据库通用接口来访问MySQL数据库，详见本章后文介绍。

PostgreSQL

和MySQL一样，PostgreSQL（发音为post-gres-Q-L）也是免费的数据库服务器，它遵循开

源许可证，你可以免费下载和使用。

尽管PostgreSQL提供了许多与MySQL相同的功能，但有相当的不同。PostgreSQL用户声称，它的速度更快，运行更稳定，并提供更多功能。PostgreSQL也经常声称它更正确地遵循SQL标准，而MySQL则更实用主义（在其更倾向于打破既有标准的意义上来说），并为了自己的方便而对SQL语言进行扩展。

和MySQL一样，Ruby对PostgreSQL的访问也是通过程序库进行，该程序库也以RubyGem包形式提供。要安装该程序库，请用以下代码：

```
gem install postgres
```

在某些环境中，可能需要指定PostgreSQL的安装位置，如下所示：

```
gem install postgres -- --with-pgsql-dir=/var/pgsql
```

Postgres程序库的接口与MySQL或SQLite完全不同，完整文档详见<http://ruby.scripting.ca/postgres/rdoc/>。（你也可以用DBI数据库通用接口访问PostgreSQL数据库，详见下面的“DBI：通用数据库连接的程序库”。）

Oracle

Oracle（甲骨文）公司是提供数据库工具、服务和软件的公司，主要因开发了Oracle RDBMS这种关系型数据库管理系统而广为人知。Oracle数据库系统不是开源的，其完整版本也不可免费用于商业用途。但它在企业环境中相当流行，因此你需要了解怎样连接到Oracle数据库。

注 Oracle确实提供数据库系统的免费版本，但仅用于开发用途。

与其他数据库系统一样，对Oracle数据库访问也是通过程序库进行，这个程序库名为OCI8。

在其专门介绍Ruby与Oracle数据库连接的Web网站上，甲骨文公司提供了绝好的教程，适合于任何熟悉Oracle数据库的人。关于更多信息，请参阅<http://www.oracle.com/technology/pub/articles/haefel-oracle-ruby.html>。

也可以通过DBI系统的数据库通用接口，使用OCI8程序库，详见下面章节的介绍。

MS SQL Server

微软的SQL Server（有时称为MS SQL Server）是微软的关系型数据库管理系统软件，也是微软平台使用的主要数据库服务器（尽管前面几节介绍的大多数其他数据库系统也能在Windows中运行）。

连接到微软SQL Server的方式，因Ruby脚本所运行的操作系统而异。例如，在本书编写时，在Windows上你可以通过WIN32OLE使用ActiveX数据对象（ActiveX Data Object，简称ADO）来直接连接到服务器。在OS X上，你可以用iODBC。在Linux和其他UNIX类平台上，你可以用unixODBC。总之，从Ruby连接到MS SQL Server不是个简单任务，因为驱动程序经常变动，因此最好的参考资料是在线教程和博客。请用你喜欢的搜索引擎，搜索“ruby my sql”或“ruby Microsoft sql server”关键字，将会发现许多有用资源。

在本书编写时，以下这些页面提供了Ruby访问MS SQL Server的最好指导：

```
http://wiki.rubyonrails.org/rails/pages/HowtoConnectToMicrosoftSQLServer
```


<http://wiki.rubyonrails.org/rails/pages/HowtoConnectToMicrosoftSQLServerFromRailsOnOSX>

<http://wiki.rubyonrails.org/rails/pages/HowtoConnectToMicrosoftSQLServerFromRailsOnLinux>

如果你打算用DBI方式, 请阅读下节内容, 了解怎样使用Ruby DBI程序库的详细信息。

DBI: 通用数据库连接的程序库

数据库接口 (DataBase Interface, DBI) 程序库是数据库通用接口程序库, 它利用数据库驱动程序, 实现通用的数据库接口。这样在编写代码时, 无须考虑任何特定的数据库, 对几乎所有数据库都可使用相同的方法。这样一来, 你就失去了使用数据库驱动程序提供的更高级功能的能力, 但DBI让开发更简便, 并可跨数据库操作。

不幸的是, 在本书编写时, DBI程序库还不能以RubyGem包形式提供 (不过这种情况可能会改变, 你不妨快速搜索一下, 可以让安装容易许多)。因此, 要安装DBI程序库, 需要参考<http://ruby-dbi.rubyforge.org/>的指导和文件。这些指导和文件会定期更新, 因此我不在此重复, 因为在你阅读时它们可能已经过时。

注 如果你是Perl程序员, 会对DBI感到非常熟悉, 因为它遵循Perl版本的许多相同惯例。

Ruby DBI程序库以及你需要的任何数据库驱动程序安装完毕后, 对于这些驱动程序所支持的数据库, 通过DBI访问即可简单许多, 如下例所示:

```
require 'dbi'

#Connect to a database
db =DBI.connect('DBI:Mysql:db_name','username','password')

#Perform raw SQL statements with 'do', supports interpolation
db.do("INSERT INTO people (name,age)VALUES (?,?)",name,age)

#Construct and execute a query that will return data in
#the traditional way..
query =db.prepare('SELECT *FROM people')
query.execute

while row =query.fetch do
  puts row.inspect
end

query.finish
#Pull data direct from the database in a single sweep
#This technique is cleaner than the previous
db.select_all('SELECT *FROM people')do |row|
  puts row.inspect
end

db.disconnect
```


在本例中，用Mysql驱动程序连接到MySQL数据库，但如果安装了用于DBI的Oracle驱动程序，也可以简单地连接到Oracle数据库，即只须把DSN改为DBI:OCI8:db_name的形式。关于DBI功能的最新信息，以及怎样使用其他驱动程序，请参阅<http://ruby-dbi.rubyforge.org/>。

9.3.7 ActiveRecord简介

到目前为止，本章讨论了用全新语言SQL来直接访问数据库。与前文介绍的把数据放到文本文件中相比，这样访问数据库是更高效和更可靠的，而ActiveRecord可以让事情更加简单。ActiveRecord是Ruby on Rails框架的一个产品，可以完全独立地使用。ActiveRecord在第13章有详细讨论，但值得在这里简要介绍一下。

ActiveRecord把SQL和数据库设计的所有细节都抽象隐藏起来，可以用面向对象的方式与数据库中内容相关联，就像PStore的使用方法一样。在ActiveRecord中，对象对应于数据库中的行，类对应与数据库中的表，因此你可以用Ruby语法来操作数据，如下所示：

```
person = Person.find(:first, :conditions => ["name = ?", "Chris"])
person.age = 50
person.save
```

这段代码搜索people表，找出name列与“Chris”相匹配的行，并将该行数据放入关联的person对象中。通过ActiveRecord可以访问该行所有列，因此修改age列就像为对象属性赋值一样容易。不过，对象的值修改后，需要调用save方法将修改内容保存回数据库。

注 从Person类映射到复数形式的people表，是ActiveRecord自动完成的功能之一。

上述代码可以用SQL语句替换，如下所示：

```
SELECT * FROM people WHERE name = "Chris"
UPDATE people SET age = 50 WHERE name = "Chris"
```

即使熟悉Ruby的SQL大师也会发现，采用Ruby语法方式更自然，特别是在Ruby程序中。如果仅用Ruby即可提供双方功能，那么我们无须在一个程序中混用两种不同的语言。

ActiveRecord在第13章有详细讨论。

注 ActiveRecord不是提供对象与数据库表关联功能的唯一程序库。Og和Lafcadio (<http://lafcadio.rubyforge.org/>) 是另外两种可选方案，但它们远远不如ActiveRecord那样流行。

9.4 小结

本章我们介绍了数据怎样流入和流出Ruby程序。一开始我们介绍了I/O流的底层概念，然后快速移到实用的数据库。数据库以更抽象的方式提供了操作数据的方法，我们无须关心数据在计算机文件系统中的底层结构。事实上，数据库可以位于内存中，或完全位于另一台机器，而我们的代码可以保持不变。

我们来回顾一下本章涵盖的主要概念：

- I/O: 是输入/输出 (Input/Output) 的缩写, 这个概念是指以不同计算机媒介接受输入和发送输出, 通常是通过I/O流。
- I/O流: 是数据发送和/或接收的渠道。
- 标准输入 (stdin): 是把数据接收到程序中的默认流, 通常是键盘。
- 标准输出 (stdout): 是从程序输出数据的默认流, 通常是屏幕。
- 文件指针: 是对文件中当前“位置”的抽象引用。
- 数据库: 是数据的有组织集合, 其数据结构易于通过程序访问。
- CSV: 是逗号分隔值 (Comma-Separated Values) 的缩写, 它是一种结构化数据的方式, 其中每个属性都以逗号分隔。CSV可用纯文本文件保存。
- 汇集 (Marshalling): 是把活的数据结构或对象转换成扁平数据集, 以便保存到磁盘或通过网络传输, 然后在其他时间和地点, 重新构造成原来的数据结构或对象的过程。
- 表: 是一组数据集, 其中数据按行排布, 每行由多个列组成, 每列代表各行的不同属性。数据库中通常有多个表, 包含不同类型的数据。
- SQLite: 是一种开源的、公共领域的关系型数据库API和程序库, 可在本机以单用户方式使用。它支持SQL查询语言。
- MySQL: 是一种开源的关系型数据库系统, 有开发社区版和专业版, 由MySQL AB Web 主机服务公司维护, 通常提供MySQL数据库支持。
- PostgreSQL: 是一种免费、开源的关系型数据库系统, 遵循BSD许可证, 可以重新包装在商业产品中销售。通常认为PostgreSQL比MySQL性能更高, 更遵循SQL标准, 但不如MySQL得到广泛使用。
- Oracle: 是甲骨文公司开发的一种商用关系型数据库系统, 一般用于大型企业中管理非常巨大的数据集。
- 主键: 是表中的一个 (或多个) 列, 其数据唯一标识每一行。
- DBI: 是数据库接口 (DataBase Interface) 的缩写, 它是数据库通用接口程序库, 可以方便地实现Ruby与数据库系统的通信。
- SQL: 是结构化查询语言 (Structured Query Language) 的缩写, 它是专用于创建、修改、检索关系型数据库中数据的, 以及对数据进行其他操作的语言。
- ActiveRecord: 是一套程序库, 它用类和对象把数据库、行、列和SQL抽象成为标准Ruby语法。它是Ruby on Rails框架的主要组成部分, 在第13章有详细介绍。

有了载入、操作和保存数据的能力, 你可以开发的有用Ruby程序将成倍增长。很少有程序只依靠每次输入的数据, 而有了访问文件和数据库的能力, 就可以构建更强大的系统, 可以长期使用来管理数据。

请注意, 本章不是介绍Ruby数据库功能的最后一章。在第12章我们将在更深的层次运用这些数据库功能, 创建一个大型应用程序。

在下面第10章中, 我们将介绍怎样对外发布应用程序和程序库。

第10章 部署Ruby应用和程序库

本章我们将考察怎样部署和发布用Ruby编写的程序。

开发Ruby应用程序非常简单，你很快就会完成并想对外发布。如第5章所述，Ruby的社区和共享有令人骄傲的历史，几乎每个Ruby开发人员都曾经发布过代码，或曾经完成过应用程序开发。事实上，由于Ruby是解释型语言，在部署Ruby应用程序时，源代码也必须一并发布。如果你不希望这样，也有一些解决办法，我们将在本章介绍。

从实质上说，本章将带你一步步学会Ruby应用程序、程序库和远程可访问服务（通过HTTP后台进程，或作为CGI脚本）在部署时的注意事项和操作流程。

10.1 简单Ruby程序发布

Ruby是解释型语言，因此要发布Ruby程序，只需发布你编写的源代码文件即可。任何安装了Ruby的人都可以运行这个文件，和你的方式一样。

发布实际源代码的过程，正是Ruby等大多数脚本语言所编写程序的共享过程，但更传统的软件在发布时，是不含源代码的。C和C++等流行的桌面应用开发语言是编译型语言，其源代码被直接转换成可在某个硬件平台运行的机器代码。这样开发出来的软件，其发布方式是把编译的结果（即机器代码文件，而不是源代码文件）复制到各台机器上。不过Ruby不可能使用这种方法，因为目前还没有Ruby编译器，因此你必须以这样或那样的方式，把源代码发布给别人，以便可以运行你的程序。

注 本章后文将介绍通过网络提供Ruby程序功能，这种方法不需要把源代码发布给别人，不过仍需要把源代码拷贝到网络服务器上。

想看看Ruby源代码怎样发布，我们以一个Ruby文件作为示例，假定文件名为`test.rb`，其内容如下：

```
puts "Your program works!"
```

如果把`test.rb`拷贝到另一台安装了Ruby的计算机上，你可以像往常一样，直接用Ruby解释器运行这个程序：

```
ruby test.rb
```

```
Your program works!
```

如果在自己的机器和服务端之间传送文件，或把程序发布给别的开发人员，这种方法能够正常工作。只要其他用户和机器有供你的程序所用的相同Ruby程序库或gem包，你的程序就能运行得很好。这是解释型语言相对编译型语言的优势之一。如果不同平台上有相同的Ruby解释器版本，它就可以和你的Ruby解释器一样，运行同样的程序。而用编译的代码（专门针对特定平台编译而成的机器代码），就不会在所有平台上同样运行，事实上，通常不能运行！

如果想把Ruby程序发布给不熟悉Ruby解释器的人，该怎么办？根据目标操作系统（即用户运行的操作系统）的不同，有几种方法可以简化Ruby应用程序的发布。

10.1.1 shebang行

在UNIX类操作系统上（Linux、OS X、BSD等），可以让程序更简单地运行，加上shebang行即可（译注：shebang行是指以#!开头的行，由#和!组成，其读音sharp和bang组合而成shebang）。

注 在某些情况下，例如当使用Apache HTTP服务器时，shebang行可以在Windows环境下运行。你可以用#!ruby和#!c:\ruby\bin\ruby.exe等shebang行，让Ruby的CGI脚本可在Windows环境的Apache中运行。

例如，假定脚本内容如下：

```
#!/usr/bin/ruby

puts "Your program works!"
```

UNIX类操作系统支持用shebang行把解释器的名字放在文件的第一行，这里“shebang”是指磅符号（#）和感叹号（!）。

注 shebang行只需要放在第一个运行文件中，不需要放在主程序之外的其他程序库和支持文件中。

在此情况下，用/usr/bin/ruby即Ruby解释器来解释运行该文件的剩余内容。不过，可能遇到的一个问题，即Ruby解释器的路径名可能是/usr/bin/local/ruby，或用完全不同的名字。有个容易移植的方法可以解决这个问题。许多UNIX类操作系统（包括大多数Linux和OS X）都有个名为env的工具程序，用来保存某些应用程序的位置和设置。可以用它来载入Ruby，无须知道确切的位置。例如：

```
#!/usr/bin/env ruby

puts "Your program works!"
```

举例来说，你可以把这个示例复制到许多不同的Linux或OS X机器上，大部分都能正常运行（env不是所有操作系统都包含的）。

如果这个脚本名为test.rb并位于当前工作目录，你就可以直接在命令行运行之，如下所示：

```
./test.rb
```

注 在大多数UNIX类操作系统中，除了加上shebang行，还需要用chmod命令把Ruby脚本文件的属性设置为“可执行”，才能让上述示例正常运行，命令为：chmod +x test.rb。

如果你把脚本拷贝到其他地方（例如，/usr/bin），你很自然地可以直接访问它：

```
/usr/bin/test.rb
```

或如果脚本的位置在搜索路径中，就更简单了：

```
test.rb
```

10.1.2 关联Windows的文件类型

尽管UNIX类操作系统用shebang行，而Windows用户更熟悉文件扩展名（例如DOC、EXE、JPG、MP3或TXT），其用途是确定这类文件该如何处理。

如果用“我的电脑”或“Windows资源管理器”找到某个文件夹中包含Ruby文件，该文件可能已经关联到Ruby解释器（根据你安装的Ruby软件包而定）。另外，Ruby文件也可能被关联给文本编辑器。不管是哪种情况，如果在Windows中双击Ruby文件，就会让它们以常规Ruby程序的形式运行，你可以把文件扩展名改为RB（或其他任何你想用的扩展名），以改变其默认操作行为。

设置关联的最简单方法，是在表示Ruby文件的图标上点击右键，并从右键菜单中选择“打开方式”（或选“打开”，如果当前没有任何程序关联到该类文件），并把Ruby程序文件关联到机器上安装的Ruby解释器ruby.exe上，并选中“始终使用选择的程序打开这种文件”选项。这样一来，以后Ruby文件就可以直接被Ruby解释器执行。

注 微软公司提供了关于这种方法的更多信息，详见<http://support.microsoft.com/kb/307859>。

10.1.3 “编译” Ruby程序

无论是用shebang行还是关联文件类型选项，都要把应用程序相关的所有Ruby源代码提供给用户，让用户从命令行或创建文件关联来运行它。

对于非技术型普通用户，这些方法令人迷惑，与Linux、OS X或Windows常规应用程序的部署相比，Ruby程序的部署显得很专业。这是部署解释型语言所编写程序的本质特征，因为Ruby无法编译成一个小小的可执行文件，像其他可执行文件一样使用。

不过，有些聪明的程序员终于在几种不同的系统中解决了这个问题，可以创建单个编译的可执行文件，令人大开眼界。其中一大技巧是把Ruby解释器和源代码嵌在一个文件中，然后用这些组件让应用程序无障碍运行。

RubyScript2Exe

RubyScript2Exe是可以把Ruby源代码转换成可执行文件的程序，可主要用于Windows和Linux。它的功能是汇总源代码和应用程序用到的其他文件（包括Ruby及其程序库），并将其打包成与典型应用程序一样的单个文件。

在编写本书时，RubyScript2Exe已经在几种Linux版本中用不同Ruby版本通过测试，并在Windows 95、98、2000和XP中通过测试。对于OS X也有试验性的支持（不过对于OS X用户还有另一种工具，Platypus，我们将在下文介绍）。

要下载并学习使用RubyScript2Exe，请访问其官方网站<http://www.erikveen.dds.nl/rubyscript2exe/>。一旦你学会在自己选择的平台上创建简单的可执行文件，程序

部署就变得相当简单，因为你不再需要担心目标用户是否已经安装了Ruby。

Platypus

Platypus是用于Mac OS X的通用开发工具，可以根据Ruby或其他解释型语言（例如Perl、Python和PHP）编写的脚本，创建原生的、集成的应用程序。

它的功能比RubyScript2Exe更多，不过使用方式略有不同，是专门针对Mac OS X的。Platypus可以加密其输出文件，这样源代码就不再可见（尽管不作加密也十分简单），并提供拖放功能，可在程序中嵌入非Ruby文件（例如图像、SQLite数据库文件或音频文件），也可使用OS X的安全框架，让你的脚本在其运行机器上得到不受拘束的访问权限。

Platypus是免费软件（尽管作者要求用户捐款），可以从以下网址获取：<http://www.sveinbjorn.org/platypus>。Platypus（如图10-1所示）是个绝顶好用的工具，用于大量解释型程序，已让这些程序成为常规应用程序。

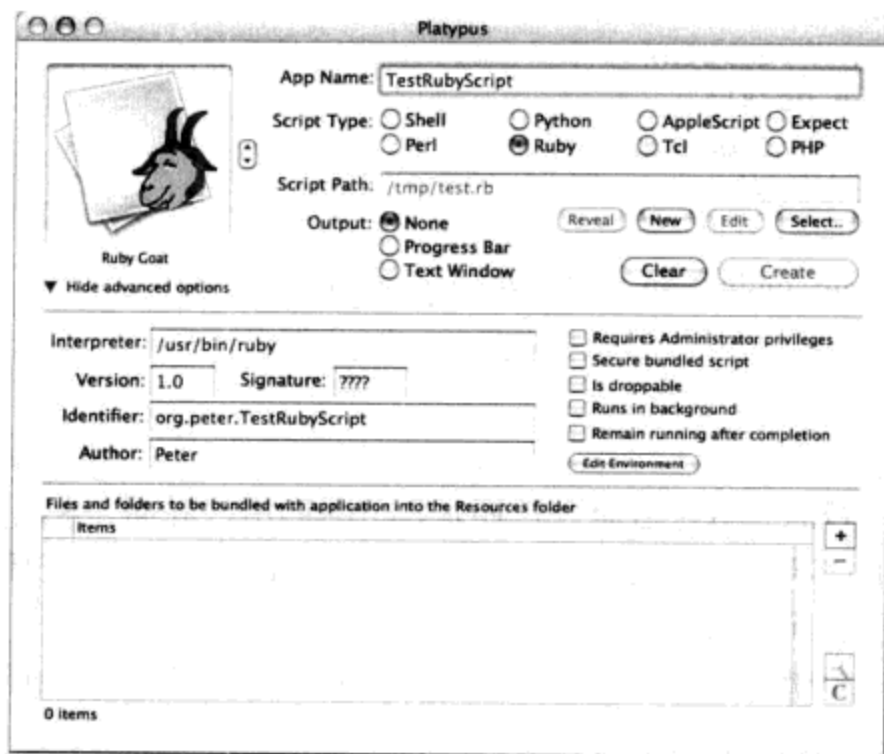


图10-1 Platypus软件主界面，其中已经组成了一个可执行程序包

10.2 检测Ruby运行环境

利用上节介绍的工具，部署Ruby程序可以变得更简单，但你也可以直接使用Ruby中的一些技术方法，让Ruby与周围环境的交互效果更好。

例如，可以检测Ruby脚本所运行机器的信息，并在运行中途改变程序的运行方式。你也可检索通过命令行传递给程序的参数。

在程序运行时检测运行环境，可以有效限制在特定平台上对用户的访问，如果你的程序与其他用户不相干的话；或有效调整程序的内部设置，以便让程序在用户的操作系统上运行得更好。它也是获取当前运行机器的系统特定信息（不是操作系统特定信息）的有用方法，因为这些信息可能影响程序的运行。常见的例子是检索当前用户的路径，即包含不同目录名的字符串，在搜索时用作文件的默认位置。另外还有一些环境变量，决定了临时文件的保存位置等。

10.2.1 用RUBY_PLATFORM作简单的操作系统检测

在Ruby可访问的无数特殊变量中，有个名为RUBY_PLATFORM的变量包含了正在运行的当前环境（操作系统）的名称。可以轻松查询该变量，以检测程序运行在什么操作系统上。如果想使用某种文件系统符号或功能，就可以先作如此检测，因为不同操作系统对文件系统符号的实现方式是不一样的。

在我的Windows机器上RUBY_PLATFORM包含“i386-mswin32”的内容，在我的OS X机器上它包含“powerpc-darwin8.6.0”，在我的Linux机器上它包含“i686-linux”，这样就让你立即可以根据操作系统信息，把程序的功能和设置区别开来。

```
if RUBY_PLATFORM =~ /win32/
  puts "We're in Windows!"
elsif RUBY_PLATFORM =~ /linux/
  puts "We're in Linux!"
elsif RUBY_PLATFORM =~ /darwin/
  puts "We're in Mac OS X!"
elsif RUBY_PLATFORM =~ /freebsd/
  puts "We're in FreeBSD!"
else
  puts "We're running under an unknown operating system."
end
```

10.2.2 环境变量

当程序在计算机中运行时，不管是命令行还是GUI图形用户界面，永远都包含在某种环境中。操作系统设置了一组名为环境变量的特殊变量，包含环境的信息。这些环境变量因操作系统而异，是检测环境的好方法，可对你的程序很有用。

通过irb使用特殊的ENV散列值，你可以迅捷简便地检查当前机器的环境变量（由操作系统提供）：

```
irb(main):001:0> ENV.each {|e| puts e.join(':')} }
```

```
TERM_PROGRAM:iTerm.app
TERM:vt100
SHELL:/bin/bash
USER:peter
PATH:
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/opt/local/bin:/usr/local/sbin
PWD:/Users/peter
SHLVL:1
HOME:/Users/peter
LOGNAME:peter
SECURITYSESSIONID:51bbd0
_:usr/bin/irb
LINES:32
COLUMNS:120
```

这些是我的机器上的特定结果，也许与你机器上的结果可能完全不同。例如，当我在Windows上用同样的代码运行时，得到如下结果：

```
ALLUSERSPROFILE:F:\Documents and Settings \All Users
APPDATA:F:\Documents and Settings \Peter \Application Data
CLIENTNAME:Console
HOMEDRIVE:F:
HOMEPATH:\Documents and Settings \Peter
LOGONSERVER:\\PSHUTTLE
NUMBER_OF_PROCESSORS:2
OS:Windows_NT
Path:F:\ruby \bin;F:\WINDOWS \system32;F:\WINDOWS
PATHEXT:.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.RB;.RBW
ProgramFiles:F:\Program Files
SystemDrive:F:
SystemRoot:F:\WINDOWS
TEMP:F:\DOCUME~1 \Peter \LOCALS~1 \Temp
TMP:F:\DOCUME~1 \Peter \LOCALS~1 \Temp
USERDOMAIN:PSHUTTLE
USERNAME:Peter
USERPROFILE:F:\Documents and Settings \Peter
windir:F:\WINDOWS
```

你可以用这些环境变量来确定临时文件的保存位置，或实时检查操作提供了哪些功能，与RUBY_PLATFORM的使用方法很相似：

```
tmp_dir = '/tmp'
if ENV ['OS'] =~ /Windows_NT/
  puts "This program is running under Windows NT/2000/XP!"
  tmp_dir = ENV ['TMP']
elsif ENV ['PATH'] =~ /\usr/
  puts "This program has access to a UNIX-style file system!"
else
  puts "I cannot figure out what environment I'm running in!"
  exit
end

[...do something here ...]
```

注 你也可以用ENV['variable_name'] = value的方式设置环境变量，但只能在有合理原因的情况下这么做。不过，在程序中设置环境变量只对本进程及其子进程有效，这表示变量的应用范围极其有限。

尽管ENV用起来像散列值，但从技术上讲，它是个特殊对象，你可以用它的.to_hash方法，即ENV.to_hash，将其转换成直接的散列值。

10.2.3 读取命令行参数

在第4章中使用过名为ARGV的特殊数组。ARGV是由Ruby解释器自动创建的数组，其中包含传递给Ruby程序的参数（不管是通过命令行还是其他途径）。例如，假定你创建了一个名为argvtest.rb的脚本，代码如下：

```
Drgvtest.rb:
puts ARGV.join('-')
```

你可以像下面这样运行这个脚本文件：

```
ruby argvtest.rb these are command line parameters
```

```
these-are-command-line-parameters
```

传递给程序的参数被放在ARGV数组中，你可以按自己的需要对其进行处理。对命令行工具来说，用ARGV是理想的方法，文件名和选项都可以用这种方式传递过来。

如果想直接调用脚本，ARGV仍然可以使用。在UNIX操作系统中，可以像下面这样来调整argvtest.rb：

```
#!/usr/bin/env ruby
puts ARGV.join('-')
```

你可以这样调用它：

```
./argvtest.rb these are command line parameters
```

```
these-are-command-line-parameters
```

命令行参数一般用来传递选项、设置和数据片段，在程序每次执行时这些参数有可能改变。例如，大多数操作系统的一个常用工具软件是copy或cp，用来拷贝文件。用法如下：

```
cp /directory1/from_filename /directory2/destination_filename
```

该命令可以把文件从文件系统的一个地方拷贝到另一个地方（并同时改名）。两个文件名都是命令行参数，而Ruby脚本可以用相同的方式来接收这些数据，如下所示：

```
#!/usr/bin/env ruby
from_filename = ARGV[0]
destination_filename = ARGV[1]
```

10.3 以gem包形式发布Ruby程序库

随着时间的推移，你会用Ruby开发出自己的程序库来解决各种问题，这样就无须一次次在不同的程序中重复编写相同的代码，只须调用程序库的支持即可。

通常你会希望这些程序库在其他地方也能使用，例如在其他机器上，在部署应用程序的服务器上，或供其他开发人员使用时。你甚至会想开放程序库的源代码，以便得到社区和更多开发人员的反馈。

如果你读过第5章，就会对Ruby为开源的贡献以及开源对Ruby开发人员的重要性很有好感。本节介绍怎样发布代码和程序库，以便对其他开发人员有用。

幸运的是，一般情况下，部署程序库比部署整个应用程序的问题要少得多，因为目标用户

都是其他开发人员，通常对安装程序库很熟悉。

在第7章我们考察了RubyGems，是Ruby的程序库安装和管理系统。我们考察了RubyGems怎样简化程序库的安装，但RubyGems还让你用自己的代码创建“gem”包变得简单易行。

10.3.1 创建gem包

我们首先来创建一个简单的程序库，对String类进行扩展，将其放到名为string_extend.rb的文件中：

```
class String
  def vowels
    scan(/[aeiou]/i)
  end
end
```

这段代码为String类增加了vowels方法，该方法返回一个数组，其内容为字符串中所有的元音字母：

```
"This is a test".vowels
```

```
["i", "i", "a", "e"]
```

如果是作为更大程序中的局部程序库，可以用require语句载入：

```
require 'string_extend'
```

但如果把它做成gem包的形式，就可以在任何地方使用。构建gem包涉及三个步骤。第一步是把代码和其他相关文件整理成特定结构，以便转换成gem。第二步是创建规格说明文件(specification file)，列出该gem的相关信息。第三步是用gem程序，根据源代码和规格说明文件生成gem包。

注 本节假定RubyGems已经完全安装，详见第7章。

整理文件

在构建gem包之前，必须把形成gem包的所有文件收集到一起，通常用标准结构来整理这些文件。到目前为止，你有个string_extend.rb文件，这是放在gem包中的唯一文件。

首先必须创建一个文件夹，其中包含所有gem的文件夹，因此你创建的文件夹名为string_extend。在此文件夹中，再创建下面几个其他文件夹：

- lib: 该目录包含与程序库有关的Ruby代码。
- pkg: 这是个临时目录，gem包在此生成。
- test: 该目录包含与程序库有关的单元测试脚本或其他测试脚本。
- doc: 这是个可选目录，用来包含与程序库有关的文档，特别是由rdoc创建的文档。
- bin: 这是另一个可选目录，用来包含与程序库有关的系统工具和命令行脚本。例如，RubyGems本身安装了gem命令行工具，这类工具将会放在bin目录中。

在极端情况下，至少应该有string_extend/lib、string_extend/pkg和string_extend/test三个目录。

在本例中,你应该把string_extend.rb文件放在string_extend/lib目录中。如果有测试脚本、文档或命令行脚本,也应该把它们放在相应的目录中。

注 上述目录名以UNIX风格书写,但Windows的表示方式需要略作调整:c:\gems\string_extend、c:\gems\string_extend\lib等。本节所有内容均适用这种调整。

创建规格说明文件

文件整理好之后,现在应该创建规格说明文件,描述gem中包含哪些内容并为RubyGems提供足够信息,来创建最终的gem包。请在string_extend主文件夹创建一个文本文件,取名为string_extend.gemspec (或其他符合你自己项目名称的名字),其中填写如下内容:

```
require 'rubygems'

spec = Gem::Specification.new do |s|
  s.name = 'string_extend'
  s.version = '0.0.1'
  s.summary = "StringExtend adds useful features to the String class"
  s.files = Dir.glob("**/*/*")
  s.test_files = Dir.glob("test/*_test.rb")
  s.autorequire = 'string_extend'
  s.author = "Your Name"
  s.email = "your-email-address@email.com"
  s.has_rdoc = false
  s.required_ruby_version = '>=1.8.2'
end
```

这是个基本的规格说明文件。规格说明文件实际上是简单的Ruby脚本,把相关信息传递给Gem::Specification。这些信息大部分很简单,我们来看几个关键的内容。

首先定义gem的名字,将其设为'string_extend':

```
s.name = 'string_extend'
```

然后定义版本号。一般情况下,Ruby项目(以及Ruby本身)的版本号包含三个部分,以重要程序顺序排列。软件在官方版本发布之前的早期版本,常常以0开头,例如这里的0.0.1:

```
s.version = '0.0.1'
```

摘要信息行是在gem list命令中显示的内容,可在安装gem包之前给出有用的信息。这里只对程序包/gem包提供了一段简短说明:

```
s.summary = "StringExtend adds useful features to the String class"
```

files属性接受一个数组,其内容是gem中包含的所有文件。在本例中,用Dir.glob得到当前目录所有文件的数组:

```
s.files = Dir.glob("**/*/*")
```

不过,你也可以在上一行代码中显式地声明数组中的每个文件。

test_files属性和files属性类似,接受由文件名组成的数组,此时是指与测试相关的

文件。即使你没有任何测试文件夹，也可以原封不动地保留这一行代码，因为`Dir.glob`将返回空数组。例如：

```
s.test_files = Dir.glob("test/*_test.rb")
```

`autorequire`参数指定了在gem包载入时自动载入的文件。这个参数不是特别重要，因为用户一般都在代码中使用`require 'string_extend'`。但如果你的gem包有多个Ruby文件需要载入，则可以很方便地指定：

```
s.autorequire = 'string_extend'
```

最后，有时程序库需要某个特定Ruby版本才能正常运行，你可以用`require_ruby_version`参数指定所需的Ruby版本。如果无须指定版本，你只须忽略这一行即可：

```
s.required_ruby_version = '>= 1.8.2'
```

注 关于RubyGems规格说明文件可用的完全参数列表，请参见<http://www.rubygems.org/read/chapter/20>。

构建gem包

规格说明文件编写完成后，构建最终的`.gem`文件非常简单，如下所示：

```
gem build <spec file>
```

在本例是：

```
gem build string_extend.gemspec
```

该命令创建最终的gem包文件，`string_extend-0.0.1.gem`。

注 以后如果修改和更新了程序库，只需把版本信息修改一下，再重新构建，就可以得到新的gem包，即可用于安装更新原来的gem包。

更简单的gem包创建方法

在本书编写时，Nic Williams博士开发并发布了一个名为`newgem`的新工具，它用一步过程即可创建用于生成gem包的文件结构和默认文件。`newgem`可用以下代码安装：

```
gem install newgem
```

安装完成后，你可以一步到位创建gem目录结构和默认文件：

```
newgem your_library_name
```

```
creating:your_library_name
creating:your_library_name/CHANGELOG
creating:your_library_name/README
creating:your_library_name/lib
creating:your_library_name/lib/your_library_name
creating:your_library_name/lib/your_library_name.rb
creating:your_library_name/lib/your_library_name/version.rb
creating:your_library_name/Rakefile
```

```
creating:your_library_name/test
creating:your_library_name/test/all_tests.rb
creating:your_library_name/test/test_helper.rb
creating:your_library_name/test/your_library_name_test.rb
creating:your_library_name/examples
creating:your_library_name/bin
```

newgem将来可能会更流行，因为它彻底地简化了创建gem包的过程。不过，你必须理解前面几节的内容，这样才会明白gem中有什么，即使你用的是自动化创建过程。

newgem在RubyForge官方网站的地址是<http://rubyforge.org/projects/newgem/>，另外作者还创建了一个有用的博客来讨论这个工具，地址是<http://drnicwilliams.com/2006/10/11/generating-new-gems/>。

10.3.2 发布gem包

gem包的发布方法很简单。可以把它上传到网站上，或用任何传输文件的方法来传递，然后就可以用gem install命令指向本地文件来安装。

不过，让gem客户可以下载并自动安装，这样的发布方式有一些难度。RubyGems系统自带了一个名为gem_server的脚本，用来在本地机器（或你选择的任何其他机器）上运行特殊服务器，它可以用来提供gem服务。要从特定服务器安装gem包，可以用--source选项：

```
gem install gem_name --source http://server-name-here/
```

注 以类似的方式，可以用常规Web服务器或Web主机软件包来容纳gem包，并可用--source选项来安装。要这么做，必须在Web网站上创建名为/gems的文件夹，并用RubyGems的generate_yaml_index.rb脚本来生成所需的元文件。

不过，发布gem包的最好方法，是可以通过因特网安装，且无须指定来源地址。例如：

```
gem install gem_name
```

该命令在因特网上搜索gem_name这个gem包，并将其下载到本地机器进行安装。但gem命令怎么知道到哪里去下载？默认情况下，如果未指定任何来源地址，则RubyGems在名为RubyForge的Ruby项目仓库中搜索gem包。下面我们将考察怎样用RubyForge把gem包放到默认数据库中。

10.3.3 RubyForge网站

RubyForge (<http://rubyforge.org/>) 是全球最大的Ruby项目和程序库的社区仓库，由Richard Kilmer和Tom Copeland维护。该网站包含成千上万的项目，担当着承载Ruby项目的中心主机的角色。几乎所有主要Ruby程序库都可在此找到，包括Ruby on Rails。

和基本的主机服务一样，RubyForge让项目维护者创建简单的网站，放在RubyForge的子域中（例如<http://mongrel.rubyforge.org/>），并为有需要的人提供源代码管理服务器（CVS和SVN）。

不过，很重要的一点是，RubyForge是gem包的默认来源。当用户运行`gem install rails`或`gem install mongrel`命令时，gem将在RubyForge搜索相应的gem包。因此，如果想让自己的gem包容易安装，关键在于把它放到RubyForge上。RubyForge主机服务是免费的，而且由于RubyForge在社区中的重要地位，把项目放在RubyForge上，可以让它看起来更合法。

要把项目放在RubyForge上，请先用<http://rubyforge.org/>主页上的链接创建新账号，完成之后就可以用它来创建新项目。你必须输入项目的一些信息，但在几天之内即被批准通过，可以上传文件。放在RubyForge上的任何项目的gem包，在几小时内即可通过RubyGems下载安装。

10.4 以远程服务形式部署Ruby应用

向用户提供源代码或将其打包在用户本地运行的另一种方法，是把程序功能作为远程服务，通过网络向外提供。这只适用于少量子功能，但提供远程功能可以让你对代码及其使用有更多控制。

Ruby的联网及Web功能将在第14章和第15章讨论，本节我们将介绍怎样用Ruby建立基本服务，让用户可以通过网络访问程序的功能。

10.4.1 CGI脚本

让脚本功能在线可用的常用方法，是把它们以CGI脚本的形式上传到Web主机上。通用网关接口（Common Gateway Interface, CGI）是一套标准，让Web服务器软件（例如Apache或微软IIS）可以调用程序，并在程序和Web客户端之间来回传递数据。

许多人把CGI一词与Perl语言关联在一起，因为Perl曾经是编写CGI脚本的最常用语言。不过，CGI是与语言无关的，你可以很容易用Ruby编写CGI脚本（事实上，用Ruby更容易！）。

基本CGI脚本

最基本的Ruby CGI脚本如下所示：

```
#!/usr/bin/ruby

puts "Content-type: text/html\n\n"
puts "<html><body>This is a test</body></html>"
```

如果把这段脚本命名为`test.cgi`并上传到基于UNIX操作系统的主机（这是最常见的类型），并赋予正确的权限，就可以把它当作CGI脚本使用。例如，假定你有个基于Linux Web主机的Web站点<http://www.example.com/>，并把`test.cgi`上传到主目录，将其设为可执行权限，然后访问<http://www.example.com/test.cgi>即可返回HTML页面，内容显示“This is a test.”

注 尽管上例中引用了`/usr/bin/ruby`，但对许多用户或Web主机来说，Ruby的位置可能是`/usr/local/bin/ruby`。因此请检查确认，或改为`usr/bin/env ruby`试试。

当Web浏览器发出对`test.cgi`的请求时，Web服务器在站点中寻找`test.cgi`文件，并用

Ruby解释器执行（根据shebang行——参见本章前文的介绍）。这段Ruby脚本返回的是基本的HTTP头（指定了内容类型为HTML），并返回基本的HTML文档。

注 关于生成HTML文档的更多信息，请参阅第14章。

Ruby自带了名为cgi的特殊程序库，可以比上述CGI脚本作更精细的交互。我们用它来创建一个基本的CGI脚本：

```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new

puts cgi.header
puts "<html><body>This is a test</body></html>"
```

本例中，创建了CGI对象，并用它来打印输出HTML头。比起记住HTML头是什么内容来说，这样做更简单，而且可以进行调整。不过，用cgi程序库的真正好处是，可以实现从Web浏览器（或HTML表单）接收数据并向用户返回更复杂数据之类的功能。

接受CGI变量

CGI脚本的一大好处，是可以处理从HTML表单传递给脚本的信息，或处理仅在URL中指定的信息。例如，如果有个提交给test.cgi的Web表单，其中有名为“text”的元素，你就可以访问传递过来的数据，如下所示：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

text = cgi['text']

puts cgi.header
puts "<html><body>#{text.reverse}</body></html>"
```

在此情况下，用户会看到他在表单中输入的文本被倒排了。你也可以把文本直接放在URL中来测试这段CGI脚本，例如<http://www.example.com/test.cgi?text=this+is+a+test>。

下面是个更完整的例子：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

from = cgi['from'].to_i
to = cgi['to'].to_i

number = rand(to-from+1) + from
```



```
puts cgi.header
puts "<html><body>#{number}</body></html>"
```

这段CGI脚本反馈一个随机数，取值范围由CGI变量from和to指定。用来发送正确数据的相关基本表单，它包含有如下HTML代码：

```
<form method="POST" action="http://www.example.com/test.cgi">
For a number between <input type="text" name="from" value="" />and
<input type="text" name="to" value="" /><input type="submit"
value="Click here!" /></form>
```

在第16章中，对CGI程序库有更深入的解释，并对HTTP cookie和会话(session)作了介绍。如果你对这种模式的部署有兴趣，请参阅相关章节了解更多信息和更长示例。

不过，一般来说，CGI运行正在逐渐淡出人们的视线，因为它速度较慢，且对每个请求都需要执行Ruby解释器，这让CGI不适用于高访问、重负载的情形。

10.4.2 常见HTTP服务器

HTTP是万维网的通讯协议。尽管它常用于网页在各处穿梭，但也可以用作内部网络甚至是单个机器中不同服务之间的通讯。

用Ruby程序创建HTTP服务器提供了一种途径，用户（甚至其他程序）可以对你的Ruby程序发出请求，这表示你无须再发布源代码，而是通过网络（例如因特网）提供程序功能。

本节不直接介绍这种功能的应用程序（在第14章有详细讨论），而是从实用角度介绍怎样用Ruby创建基本的Web/HTTP服务器。

WEBrick

WEBrick是一套Ruby程序库，用来通过Ruby构建HTTP服务器。大多数Ruby安装版本中都默认提供WEBrick（它是标准程序库的组成部分），因此通常只用几行代码即可创建基本的Web/HTTP服务器：

```
require 'webrick'

server = WEBrick::GenericServer.new(:Port =>1234 )

trap("INT"){server.shutdown }

server.start do |socket|
  socket.puts Time.now
end
```

这段代码在本地机器的1234端口创建了通用WEBrick服务器，如果进程被中断（常常是用Ctrl+C）则关闭服务器，对于每个新连接，均打印当前日期和时间。如果运行这段代码，可以在Web浏览器中访问http://127.0.0.1:1234/或http://localhost:1234/来查看运行结果。

警告 因为测试程序不提供合法的HTTP输出，因此对于大多数Web浏览器都可能出错，尤其是在Windows上。不过，如果你理解telnet程序的用法，可以用telnet 127.0.0.1

1234来查看结果。否则，请继续到下一个例子，其中提供了合法的HTTP以便返回给Web浏览器查看。

更强大的技术体现在创建servlet上，它位于自己的类中，对请求和反馈有更强的控制：

```
require 'webrick'

class MyServlet < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    response.status = 200
    response.content_type = "text/plain"
    response.body = "Hello, world!"
  end
end

server = WEBrick::HTTPServer.new(:Port => 1234)
server.mount "/", MyServlet
trap("INT") { server.shutdown }
server.start
```

这段代码更复杂，现在可以访问request和response对象，这两个对象代表了进来的请求和出去的反馈。

例如，你可以检查用户用浏览器要访问哪个URL：

```
response.body = "You are trying to load #{request.path}"
```

request.path包含URL中的路径（例如http://127.0.0.1:1234/abcd中的/abcd）表示你可以解释用户的请求内容，调用不同的方法，并提供正确的输出。

下面是个更复杂的例子：

```
require 'webrick'

class MyNormalClass
  def MyNormalClass.add(a, b)
    a.to_i + b.to_i
  end

  def MyNormalClass.subtract(a, b)
    a.to_i - b.to_i
  end
end

class MyServlet < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    if request.query ['a'] && request.query ['b']
      a = request.query ['a']
      b = request.query ['b']
      response.status = 200
      response.content_type = 'text/plain'
```



```

result =nil

case request.path
  when '/add'
    result =MyNormalClass.add(a,b)
  when '/subtract'
    result =MyNormalClass.subtract(a,b)
  else
    result ="No such method"
end

response.body =result.to_s +"\n"
else
  response.status =400
  response.body ="You did not provide the correct parameters"
end
end
end

server =WEBrick::HTTPServer.new(:Port =>1234)
server.mount '/',MyServlet
trap('INT'){server.shutdown }
server.start

```

在本例中，有个名为MyNormalClass的常规基本Ruby类，它实现了两个基本的算术方法。WEBrick的servlet使request对象从URL中检索参数，并从request.path得到所请求的Ruby方法。如果没有传递参数，则返回HTTP错误。

要使用上述脚本，需要用到如下的URL：

```
http://127.0.0.1:1234/add?a=10&b=20
```

```
30
```

```
http://127.0.0.1:1234/subtract?a=100&b=10
```

```
90
```

```
http://127.0.0.1:1234/subtract
```

```
You did not provide the correct parameters.
```

```
http://127.0.0.1:1234/abcd?a=10&b=20
```

```
No such method.
```

注 可以从<http://microjet.ath.cx/WebWiki/WEBrick.html>的《几诺米WEBrick指南》或本书的附录C了解WEBrick的更多信息。

Mongrel

Mongrel是一种快速HTTP服务器和Ruby程序库，目的是为Ruby应用程序提供主机服务。它与WEBrick很相似，但速度快得多，缺点是Ruby没有默认自带，需要额外安装。由于它的速度、稳定性和可靠性，许多评价甚高的Ruby on Rails网站都用Mongrel进行部署。

你可以用RubyGem安装Mongrel：

```
gem install --include-dependencies mongrel
```

注 和往常一样，如果你的操作系统需要的话，请记得在命令前加sudo。

和WEBrick一样，可以很容易地把Mongrel和现有Ruby应用程序绑定到一起。通过把Mongrel与句柄类进行关联，请求可以传递进来，由自己的代码进行处理。这些代码可以调用程序中的功能，并把结果返回给客户端。

下面是个Mongrel服务器的基本示例，当载入http://localhost:1234时，返回简单的HTML页面：

```
require 'rubygems'
require 'mongrel'

class BasicServer < Mongrel::HttpHandler
  def process(request, response)
    response.start(200) do |headers, output|
      headers["Content-Type"] = 'text/html'
      output.write('<html><body><h1>Hello!</h1></body></html>')
    end
  end
end

s = Mongrel::HttpServer.new("0.0.0.0", "1234")
s.register("/", BasicServer.new)
s.run.join
```

Mongrel::HttpServer.new也接受可选的第三个参数，指定用于处理请求所开启的线程数。例如：

```
s = Mongrel::HttpServer.new("0.0.0.0", "1234", 20)
```

上面的代码行创建20个处理器线程来处理请求。

如你所见，Mongrel与WEBrick非常相似，但有一些额外的优势。你可以访问Mongrel官方网站<http://mongrel.rubyforge.org/>了解更多信息。

10.4.3 远程方法调用

让远程程序可以访问本程序的功能，常用方法之一是使用远程方法调用（Remote Procedure Call, RPC）。与通过Web浏览器控制不同，RPC专用于一个程序使用其他程序开放的方法与功能。当RPC被使用得当时，使用远程程序开放的方法与功能，几乎和使用本地方法与功能一样简单。

Ruby内置支持两种最流行的RPC协议：XML-RPC和SOAP，也支持自己名为DRb的专用系统。

XML-RPC

XML-RPC是一种广为人知的RPC协议，它以XML作为消息载体，以HTTP作为传输协议。使用RPC的一大好处是可以用多种语言创建多个程序，但仍然可以用每种语言都理解的方式进行相互沟通。有了XML-RPC，我们就可以这样来进行开发：系统主体用PHP或Python编写，但可以调用Ruby程序提供的方法。

注 SOAP是另一种流行的RPC协议，但更加复杂。不过，它是Ruby原生支持的协议，如果想用的系统不支持其他任何协议，就只能用它了。其他系统，特别是简单得多的系统（例如REST）正在逐渐流行。但如果想用SAOP，Ruby自带了标准的SOAP程序库。

调用支持XML-RPC的方法

对支持XML-RPC的方法的调用极其简单：

```
require 'xmlrpc/client'

server = XMLRPC::Client.new2("http://xmlrpc-c.sourceforge.net/api/sample.php")
puts server.call("sample.sumAndDifference", 5, 3).inspect

{"difference"=>2, "sum"=>8}
```

注 该程序需要你的计算机可以访问因特网。而且如果XML-RPC示例服务器不可用，可能会得到出错消息。如果得不到结果，请多试几次，因为这个示例文件的负载压力通常都很大。

本例调用了一个远程程序（用PHP编写），提供了名为sample.sumAndDifference的方法。首先你要用XMLRPC::Client.new2创建一个指向远程程序的句柄，然后再用两个参数来调用这个方法。

调用结果（对你提供的两个参数进行汇总和相减）以散列表方式返回。因为调用远程程序可能会导致出错（连接失败、远程服务不可用等），因此必须对RPC返回的错误进行处理。XML-RPC提供了call2方法，让错误处理很简单：

```
require 'xmlrpc/client'

server =XMLRPC::Client.new2("http://xmlrpc-c.sourceforge.net/api/sample.php")
ok,results =server.call2("sample.sumAndDifference",5,3)

if ok
  puts results.inspect
else
  puts results.faultCode
  puts results.faultString
end
```


call2方法返回一个数组，其中包含“success”标志和结果。你可以检查数组的第一个元素（“success”标志）是否为true，如果不是，就可以检查错误信息。

开发支持XML-RPC的程序

调用支持XML-RPC的程序很容易，但开发自己的支持XML-RPC的程序也很简单：

```
require 'xmlrpc/server'

server =XMLRPC::Server.new(1234)
server.add_handler("sample.sumAndDifference")do |a,b|
  {"sum"=>a.to_i +b.to_i,
   "difference"=>a.to_i - b.to_i }
end
trap("INT"){server.shutdown }
server.serve
```

本程序在本地机器的1234端口运行一个XML-RPC服务器（基于WEBrick），并与上节客户端使用的sample.php采用同样的方式进行处理。下面的客户端代码可以调用上面服务器开放的sample.sumAndDifference方法：

```
require 'xmlrpc/client'

server =XMLRPC::Client.new2("http://127.0.0.1:1234/")
puts server.call("sample.sumAndDifference",5,3).inspect
```

在服务器端，只须增加更多的add_handler代码块来处理请求。你可以用require来载入与程序相关的类，并使简单的XML-RPC服务器就位，让程序的功能可以被远程访问。例如：

```
require 'xmlrpc/server'
require 'string_extend'

server =XMLRPC::Server.new(1234)

server.add_handler("sample.vowel_count")do |string|
  string.vowels
end

trap("INT"){server.shutdown }
server.serve
```

该XML-RPC服务器把string_extend程序库功能对远程开放。你可以用下面的方式来调用：

```
require 'xmlrpc/client'

server =XMLRPC::Client.new2("http://127.0.0.1:1234/")
puts server.call("sample.vowel_count","This is a test").inspect

["i","i","a","e"]
```

和WEBrick和Mongrel一样，XML-RPC服务器也可以直接使用其他类，如WEBrick的servlet所示。例如：

```

class OurClass
  def some_method
    "Some test text"
  end
end

require 'xmlrpc/server'

server =XMLRPC::Server.new(1234)
server.add_handler(XMLRPC::iPIMethods('sample'),OurClass.new)

trap("INT"){server.shutdown }
server.serve

```

对于该服务器，这些方法自动与XML-RPC服务器关联。从XML-RPC客户端对sample.some_method发出的调用，将自动路由给OurClass开放的some_method实例方法。

DRb

DRb是“分布式Ruby”（Distributed Ruby）的缩写，它是仅用于Ruby的RPC程序库。从表面上看，DRb与XML-RPC没有太大区别，但如果你只需在Ruby程序之间相互通讯，DRb则更加强大。与XML-RPC不同，DRb是面向对象的，同时连接到DRb服务器，即向客户端提供了一个类的实例，该实例位于DRb服务器上。你可以像本地方法一样，调用该类开放的方法。

DRb客户端可以像下面这样简单：

```

require 'drb'

remote_object = DRbObject.new nil, 'druby://:51755'
puts remote_object.some_method

```

只需一行代码，即可检索来自远程DRb服务器的类的实例。用XML-RPC首先要创建指向服务器的句柄，而用DRb则创建指向类实例的句柄。在调用了DRbObject.new之后，remote_object就是指向对象的句柄，该对象由特定DRb服务器（在本例中是本地服务器）提供服务。

我们来看与客户端关联的服务器：

```

require 'drb'

class OurClass
  def some_method
    "Some test text"
  end
end

DRb.start_service nil,OurClass.new
puts "DRb server running at #{DRb.uri}"
trap("INT"){DRb.stop_service }
DRb.thread.join

```



没有比这更简单的了。`DRb.start_service`把`OurClass`的一个实例与DRb服务器关联起来，它把发送给DRb服务器的URL打印出来，然后启动DRb服务器，等待客户端的连接。

注 你需要在客户端示例代码中修改`druby://`这个URL，以便匹配由服务器示例代码输出的URL。

有了DRb，数据结构几乎可以无障碍的穿越网络连接。如果远程类要返回复杂的散列表，这完全可以做到，并会完美地在客户端得到展现。在一定程度上，通过DRb连接来使对象是透明的，远程对象表现得仿佛是本地对象。

这只是DRb基本功能的简单概述。不过，用DRb的基本RPC功能术语来说，这个简单的服务器—客户端示例，展示了DRb的核心功能集，你可以将其扩展到任意的复杂度。如果代码已经用类写好，就可以在几分钟之内放入DRb功能，并立即让你的程序功能在远程可以访问。

提示 关于其他DRb基础入门教程，请访问<http://www.chadfowler.com/ruby/drb.html>。关于最新文档，请访问<http://www.ruby-doc.org/stdlib/libdoc/drb/rdoc/index.html>。

10.5 小结

本章我们考察了怎样部署Ruby程序和程序库，以及怎样通过网络向Web浏览器和其他应用程序开放自身的功能。我们还审视了运行环境，从而可以根据每个不同的操作系统，在程序中采用不同的技术方法。

我们来回顾一下本章涵盖的主要概念：

- Shebang行：是源代码文件开头的特殊行，用来确定处理本文件的解释器。主要用于UNIX类操作系统，当用于Apache Web服务器时，shebang行也能在Windows环境下使用。
- `RUBY_PLATFORM`：是由Ruby预置的特殊变量，包含当前平台（环境）的名称。
- 环境变量：是由操作系统或其他进程设置的特殊变量，包含与当前执行环境以及操作系统相关的信息。
- RubyForge：是专门容纳并发布Ruby项目及程序库的中心仓库和Web站点，可在<http://rubyforge.org/>找到。
- CGI：是通用网关接口（Common Gateway Interface）的缩写，它是一套标准，让Web服务器可以执行脚本，并在Web用户和服务器脚本之间提供接口。
- WEBrick：是用于Ruby的一套简单易用的HTTP服务器程序库，由Ruby作为标准程序库默认自带。
- Mongrel：由Zed Shaw开发，是用于Ruby的更强大的HTTP服务器程序库，比WEBrick在速度、稳定性和总体性能上有相当大的改进。
- RPC：是远程方法调用（Remote Procedure Call）的缩写，它是通过网络（局域网或因特网）、传输协议（例如HTTP）和消息协议（例如XML）调用其他程序功能的方法。
- XML-RPC：是一种RPC协议，使用HTTP和XML作为传输和消息机制。

- SOAP: 是简单对象访问协议 (Simple Object Access Protocol) 的缩写, 它是另一种RPC协议, 使用HTTP和XML作为传输和消息机制。
- DRb: 是分布式Ruby (Distributed Ruby) 的缩写, 它只用于Ruby的RPC机制, 可在不同的Ruby脚本之间实现对象操作。

在第15章, 我们将回过头来考察网络服务器, 不过是以不同的方式。首先在第11章, 我们将考察更高级Ruby主题, 以进一步丰富目前已学内容。



第11章 Ruby高级功能

本章我们将考察前面章节没有涉及的一些Ruby高级技术，本章也是本书第二篇的最后一章。我们将在第三篇介绍有用的程序库、框架和Ruby相关技术，而本章则是熟练Ruby程序员必备知识的圆满结束。这意味着尽管本章涵盖几个不同的主题，但每个主题都是成为专业Ruby开发人员所必不可少的。

本章涵盖了许多主题，包括怎样在运行过程中动态创建Ruby代码，怎样让Ruby代码更安全，怎样对操作系统发出命令，怎样与微软Windows集成，以及怎样创建用于其他编程语言的Ruby程序库。从根本上说，本章的目的是涵盖一系列具体而重要的主题，这些主题超出了其他章节的范围，但却是开发中可能需要用到的。

11.1 动态代码执行

作为一门动态的解释型语言，Ruby也可以执行动态创建的代码，方法是用eval方法。例如：

```
eval "puts 2 + 2"
```

```
4
```

请注意，当4显示之后，4并非作为整个eval表达式的返回值，puts永远返回nil。如要从eval返回4，可以这么做：

```
puts eval("2 + 2")
```

```
4
```

下面是个更复杂的例子，使用了字符串和插写：

```
my_number = 15
my_code = %Q{#{my_number} * 2}
puts eval(my_code)
```

```
30
```

eval方法仅对传递给它的代码进行执行（或称评估运算），并返回结果。第一个例子让eval招待puts 2 + 2，而第二个例子则是用字符串插写，构建了15 * 2的表达式，然后再对这个表达式进行评估运算，并用puts打印输出到屏幕。

11.1.1 绑定

在Ruby中，绑定（binding）是指对上下文环境、作用域或执行状态的引用。例如，当前变量值和其他执行状态详细信息，都可以是绑定所包含的内容。

可以把绑定传递给eval并让eval执行该绑定所提供的代码，而不是执行当前代码。这样，

你就可以让代码在eval中执行，与程序执行的主环境互不相干。

举例如下：

```
def binding_elsewhere
  x = 20
  return binding
end

remote_binding = binding_elsewhere

x = 10
eval("puts x")
eval("puts x", remote_binding)
```

```
10
20
```

这段代码展示了eval接受可选的第二个参数，它是个绑定，在本例中是从binding_elsewhere方法返回的值。remote_binding变量所包含的，是对binding_elsewhere方法（而不是主代码）所处的执行环境的引用。因此，当打印输出x时，则显示20，因为在binding_elsewhere中定义x等于20！

注 你可以用Kernel模块的binding方法，随时获取对当前作用域的绑定。

上例很容易扩展，如下所示：

```
eval("x = 10")
eval("x = 50", remote_binding)
eval("puts x")
eval("puts x", remote_binding)
```

```
10
50
```

在本例中涉及两个绑定：默认绑定值和remote_binding值（来自binding_elsewhere方法）。

因此，即使把x先设为10，然后设为50，两次处理的并非同一个x。其中一个x是当前环境的局部变量，而另一个x则位于binding_elsewhere的环境中。

11.1.2 eval的其他形式

尽管eval在当前环境（或由某个绑定值指定的环境）中执行代码，而class_eval、module_eval和instance_eval则可以分别在类、模块和对象实例的环境中执行代码。

class_eval是向类中动态增加方法的理想选择：

```
class Person
end
```

```
def add_accessor_to_person(accessor_name)
  Person.class_eval %Q{
    attr_accessor :#{accessor_name}
  }
end
```

```
person = Person.new
add_accessor_to_person :name
add_accessor_to_person :gender
person.name = "Peter Cooper"
person.gender = "male"
puts "#{person.name} is #{person.gender}"
```

```
Peter Cooper is male
```

在本例中，用`add_accessor_to_person`方法向`Person`类中动态加入属性读写方法，在使用`add_accessor_to_person`方法之前，`Person`中既没有`name`也没有`gender`这两个属性读写方法。

请注意，这段代码的关键部分`class_eval`方法，在为`Person`创建所需代码时，使用了字符串插写操作：

```
Person.class_eval %Q{
  attr_accessor :#{accessor_name}
}
```

字符串插写让`eval`方法成为强大的工具，可以在运行过程中生成不同的功能。这一能力在主流编程语言中是看不见的，在Ruby on Rails等系统中被用到极致（详见第13章）。

可以把上例再向前推进一大步，通过把巧妙的`class_eval`放到`Class`类的新方法中（以便让所有其他类继承），从而为其中每个类增加`add_accessor`方法。

```
class Class
  def add_accessor(accessor_name)
    self.class_eval %Q{
      attr_accessor :#{accessor_name}
    }
  end
end
```

```
class Person
end
```

```
person = Person.new
Person.add_accessor :name
Person.add_accessor :gender
person.name = "Peter Cooper"
person.gender = "male"
puts "#{person.name} is #{person.gender}"
```

本例向`Class`类加入`add_accessor`方法，从而向程序中每个类都增加了这个方法。这样

就可以对任何类动态增加属性读写方法，只要调用`add_accessor`即可（如果你搞不清这种方法的代码逻辑，请务必亲自动手试一下，一步步完成每个过程，并证实每步执行的结果）。

上例所用的方法，也可用来定义下面这样的类：

```
class SomethingElse
  add_accessor :whatever
end
```

因为`add_accessor`在某类中使用，调用该方法将回溯至`Class`类中定义的`add_accessor`方法。

我们回到更简单的方法，使`instance_eval`有点像常规的`eval`，但其运行环境隶属于某个对象（而不是某个方法）。在本例中，你看到`instance_eval`在对象的作用域内执行代码。

```
class MyClass
  def initialize
    @my_variable = 'Hello, world!'
  end
end

obj = MyClass.new
obj.instance_eval { puts @my_variable }
```

```
Hello, world!
```

11.1.3 创建`attr_accessor`

到目前为止，我们已经在类中用过`attr_accessor`方法，以便为实例变量快速生成属性读写方法。例如，以常规方式，你可能编写这样的代码：

```
class Person
  def name
    @name
  end

  def name=(name)
    @name = name
  end
end
```

这段代码可以这样使用，例如`puts person.name`和`person.name = 'Fred'`。不过，还可以改用`attr_accessor`：

```
class Person
  attr_accessor :name
end
```

这个版本的类更加简洁，与上面常规版本的功能完全相同。我们不禁要问：`attr_accessor`的功能是怎样实现的？

实际上`attr_accessor`并不像看起来那么神秘，用`eval`可以非常容易实现你自己的版本。

我们来看下面的代码：

```
class Class
  def add_accessor(accessor_name)
    self.class_eval %Q{
      def #{accessor_name}
        @#{accessor_name}
      end

      def #{accessor_name}=(value)
        @#{accessor_name} = value
      end
    }
  end
end
```

乍看起来，这段代码很复杂，但其实与上节创建的add_accessor代码非常相似，是用class_eval为当前类的属性定义了读方法和写方法。

如果accessor_name等于“name”，那么class_eval执行的代码就等于下面的代码：

```
def name
  @name
end

def name=(value)
  @name = value
end
```

这样就复制了attr_accessor的功能。

可以用这种方法创建多种不同的“代码生成器”，以及创建像“宏”语言一样的方法，这种“宏”在Ruby中可以完成的功能，改用其他方式则需要输入大量代码才能实现。

11.2 从Ruby中运行其他程序

在自己的程序中可以运行操作系统的其他程序，这是一种有用的能力。这样一来，你就可以减少自己程序必须实现的功能总量，因为你可以把相关工作移交给已经编写好的程序。另外，能够把自己编写的多个程序连接起来，让相关功能在每个程序中都可以使用，这也是非常有用的能力。与此同时无须上章介绍的远程方法调用（RPC）系统，只需运用Ruby提供的一两个方法，即可在自己的程序中运行其他程序。

11.2.1 获得其他程序的运行结果

在Ruby中调用其他程序有三种方法：system命令（在Kernel模块中定义）、反引号语法（```）以及定界输入字面值（delimited input literals）（`%x{}`）。如果不关心调用外部程序的输出结果，则使用system命令是个理想选择。如果需要外部程序返回输出结果，则应使用反引号。

下面几行代码展示了运行操作系统date程序的两种方法：

```
x = system("date")
```

```
x = 'date'
```

对于第一行代码，`x`等于`true`，而第二行代码的`x`则包含`date`命令的输出。使用哪种方法取决于你想要什么结果。如果在Ruby脚本所运行的屏幕上，你不想看到其他程序的输出，则应使用反引号（或字面值`%x{}`）。

注 `%x{}`在功能上与反引号等价，例如`%x{date}`。

11.2.2 向其他程序移交执行权

有时需要立即跳转到其他程序，并终止当前程序的运行。在某些情况下这样做很有用，例如，你要完成多步过程，并已编写应用程序实现每个步骤。要终止当前程序并启动另一个，只需使用`exec`命令取代`system`命令。例如：

```
exec "ruby another_script.rb"
puts "This will never be displayed"
```

在本例中，执行权被移交给另一程序，并且当前程序立即终止，而第二行代码永远不会运行。

11.2.3 同时运行两个程序

分支（Forking）操作是指程序的实例（进程）复制自身，导致该程序的两个进程并发运行。你可以在第二个进程调用`exec`命令运行其他程序，而第一个（父）进程则继续原来的运行路线。

`fork`是Kernel模块提供的方法，可以创建当前进程的分支子进程，该方法在父进程中返回子进程的进程ID，但在子进程中则返回`nil`，你可以据此确定脚本运行在哪个进程当中。下面的示例把当前进程分支为两个进程，只在子进程（即分支生成的进程）中运行`exec`命令：

```
if fork.nil?
  exec "ruby some_other_file.rb"
end

puts "This Ruby script now runs alongside some_other_file.rb"
```

如果另一个程序（即被`exec`命令所运行的程序）在某个时刻被终止，并且你想让父程序等它运行结束，则可以使用`Process.wait`来等待所有子进程终止，然后再继续进行。示例如下：

```
child = fork do
  sleep 3
  puts "Child says 'hi'!"
end

puts "Waiting for the child process..."
Process.wait child
puts "All done!"
```

```
Waiting for the child process...
<3 second delay>
Child says 'hi'!
```



```
All done!
```

注 分支操作对Windows版的Ruby不可使用，因为POSIX风格的分支操作不受Windows平台支持。不过，本章后文介绍的线程，提供了很好的备选方案。

11.2.4 与另一程序交互

在其运行过程中不需要以任何方式与其直接交互的情况下，上面的方法对简单情况很有效，即在只需从外部程序得到基本结果。但有时可能想让数据在两个独立程序之间来回传递。

Ruby的IO模块有个

下面是个简单的只读功能示例：

```
ls = IO.popen("ls", "r")
while line = ls.gets
  puts line
end
ls.close
```

在本例中，用ls命令（是列出当前目录内容的UNIX命令，如果你用的是微软Windows，请改用dir命令）开启I/O流，然后和其他形式的I/O流一样，一行行读取内容，并在完毕后关闭流。

同样，也可以用可读可写的I/O流打开程序，并处理双向数据：

```
handle = IO.popen("other_program", "r+")
handle.puts "send input to other program"
handle.close_write
while line = handle.gets
  puts line
end
```

注 使用handle.close_write的原因，是为了关闭I/O流的写出流，从而把等待写出给外部程序的所有数据都发送出去。如果需要写出流不关闭，还可以使用IO类的flush方法。

11.3 安全地掌控数据和危险方法

对于Ruby应用程序来说，有时程序的操作依赖于外部源头的的数据，这样的情况很常见。这些数据无法永远值得信任，因此保护机器和环境免受由恶意数据或代码导致的意外情况的打击，是一件很有用的事。Ruby有两种方法可以让自己变得更安全：一是检查外部数据是否被感染，二是设置安全级别，从而让Ruby解释器限制某些功能不得被代码执行。

11.3.1 被感染的数据和对象

在Ruby中，如果数据来自外部源头，或如果Ruby没有办法证实数据是否安全，则一般认为这些数据被感染了。例如，从命令行收集的数据可能不安全，因此被视为感染数据。从外部文件或通过网络连接读取的数据也是被感染的。不过，数据以硬编码方式写在程序中，例如字面字符串，则被视为未感染的。

我们来看下面的简单程序，它展示了为什么检查感染数据很重要：

```
while x = gets
  puts "=> #{eval(x)}"
end
```

这段代码运行起来像个微型版本的irb，它接受一行行用户输入，并立即执行：

```
10+2
=> 12
"hello".length
=> 5
```

但如果某人想捣乱，输入'`rm -rf /*`'会发生什么情况？这行命令还会运行！

警告 不要把上面这行代码输入到程序中！在UNIX类操作系统中，如果具备恰当权限，运行`rm -rf /*`将删除硬盘大部分内容！

很明显，在某些情况下，你需要检查数据是否被外部世界感染。

也可以用`tainted?`方法检查对象是否被感染：

```
x = "Hello, world!"
puts x.tainted?

y = [x, x, x]
puts y.tainted?

z = 20 + 50
puts z.tainted?

a = File.open("somefile").readlines.first
puts a.tainted?

b = ENV["PATH"]
puts b.tainted?

c = [a, b]
puts c.tainted?
```

```
false
false
false
true
```

```
true
false
```

注 上面几个例子中，有一个例子需要somefile文件在本地目录真实存在才能运行。

前三个例子都是对已经定义在程序中的数据（字面数据），因此认为是未感染的。最后三个例子都涉及外部源头的的数据（a包含文件的第一行内容，b包含操作系统的环境信息），那么，为什么最后一个例子被认为是未感染的？

c被认为是未感染的，是因为c只是个数组，包含对a和b的引用。尽管a和b都是感染的，但包含a和b的数组则不然。因此，必须检查所用到的每片数据是否感染，而不是检查整体数据结构。

注 相对于数据检查，另一种可选方法是设置Ruby解释器的“安全级别”，从而让任何潜在的危險操作都被禁止。这种方法在下节介绍。

也可以对对象调用untaint方法，强制对象被视为未感染的。例如，下面是个对Ruby解释器极其安全的版本：

```
while x = gets
  next if x.tainted?
  puts "=> #{eval(x)}"
end
```

不过，这段代码毫无用处，因为从用户接收的所有数据都被视为感染的，因此什么都不会运行。这纯粹是“不作为，保平安”！但我们在此假定，你有个方法可以确定某个操作是否安全，如下所示：

```
def code_is_safe?(code)
  code =~ /[^\;*-]/?false :true
end

while x =gets
  x.untaint if code_is_safe?(x)
  next if x.tainted?
  puts "=>#{eval(x)}"
end
```

警告 code_is_safe?仅检查代码行是否包含反引号、分号、星号、横杠连字符，并把符合条件的代码视为不安全的。这并非检查代码安全的合理方法，只是用作演示。

在本例中，你显式地把认为安全的代码标为未感染的，因此eval将执行所有“安全”代码。

注 同样，可以调用对象的taint方法，显式地将其标为感染的。

11.3.2 安全级别

尽管可以检查数据是否被感染，并采取保护性动作将其清除，但还有一种更强形式的保护

方法，即Ruby自带的“安全级别”。安全级别可以指定Ruby可以使用哪些功能，以及怎样处理感染的数据库。

当前安全级别由\$SAFE变量表示。默认情况下，\$SAFE被设为0，即提供最低级别的安全性，最高级别的自由度。但也可以使用其他4种模式，如表11-1所示。

表11-1 Ruby的安全级别，由\$SAFE表示

\$SAFE的值	说 明
0	没有任何限制。这是默认的安全级别
1	可能不安全的方法无法使用感染的数据库。而且，当前目录不加入到Ruby的搜索路径，该路径用于载入程序库
2	对安全级别1进行限制，增加不让Ruby从文件系统的全局可写位置载入任何外部程序文件的限制。这是为了防止黑客通过上传攻击性代码，并操纵现有程序，载入这些代码的方式进行攻击
3	对安全级别2进行限制，增加自动把程序中新创建对象视为感染的限制。并且无法消除对象的感染状态
4	对安全级别3进行限制，增加无法修改其他安全级别所创建的非感染对象的限制。你可以借此建立一个低安全模式的运行环境，程序可以继续执行，但保护原来的对象和环境不受影响

要改变安全级别，只须把\$SAFE的值设置为你想要的安全级别。但务必注意，一旦设置了安全级别，只能增加安全级别，无法减少它。原因在于：如果允许减少安全级别，那么只须降低它，就可以用eval运行不安全的代码，并导致重大破坏！

11.4 使用微软Windows

到目前为止，本书的例子都是通用的，稍稍偏向基于UNIX的操作系统。对于微软Windows世界来说，Ruby是个相对迟到的后来者，但现在它有了一些程序库，可以轻松地直接使用Windows的API。

本节考察怎样在Ruby中使用Windows API和OLE功能，不过这需要你拥有这些主题的深厚知识，如果想编写更高级代码的话。

11.4.1 使用Windows API

微软Windows提供了一套应用编程接口（Application Programming Interface, API），它像Windows核心功能的程序库，可用来访问Windows内核、图形界面、控制程序库、网络连接服务和用户界面。Ruby的Win32API程序库（包含在标准程序库中），为开发人员提供了访问原汁原味的Windows API功能。

注 本节的所有代码都只能在Windows中运行，而非在其他任何操作系统中运行，并且在Windows 98之前的版本可能也无法运行。

打开对话框是个相当简单的操作：

```
require 'Win32API'
```

```
title = "My Application"
```

```
text = "Hello, world!"
```

```
Win32API.new('user32','MessageBox',%w{L P P L},'I').call(0,text,title,0)
```

首先，把Win32API程序库载入程序。然后设置变量，提供对话框所需标题和内容。接着，创建对Windows API提供的MessageBox函数的引用，然后用设置好的文本和标题来调用该引用。结果如图11-1所示。

Win32API.new的参数说明如下：

1. 包含所需调用函数的系统DLL的名字。
2. 想要调用的函数的名字。
3. 描述每个参数格式的数组，以便传递给该函数。
4. 表示该函数返回何种数据类型的字符。

在本例中，指定了想要调用由user32.dll文件提供的MessageBox函数，并为该函数提供4个参数（一个数字、两个字符串和另一个数字——L代表数字，P代表字符串），并期望返回整数（I代表整数）。

有了对函数的引用，就可以用call方法以4个参数调用该函数。对于MessageBox函数，4个参数含义如下：

1. 对父窗口的引用（本例未涉及）。
2. 显示在消息窗口的文本。
3. 消息窗口的标题。
4. 消息窗口的显示类型（0代表基本的“确定”按钮式对话框）。

call方法返回的整数在本例中没有用到，但该数字用来表示用户在对话框中点击了哪一个按钮。

当然，也可以创建更复杂的例子：

```
require 'Win32API'

title = "My Application"
text = "Hello,world!"

dialog = Win32API.new('user32','MessageBox','LPPL','I')
result = dialog.call(0,text,title,1)

case result
  when 1:
    puts "Clicked OK"
  when 2:
    puts "Clicked Cancel"
  else
    puts "Clicked something else!"
end
```

本例使用了MessageBox函数的返回值，并用它来确定哪个按钮被点击。在本例中，以第4个参数为1调用MessageBox函数，表示对话框包含“确定”和“取消”两个按钮。结果如图11-2所示。



图11-1 基本的对话框

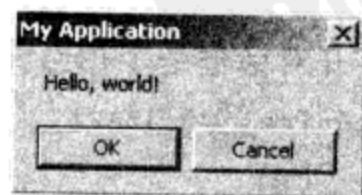


图11-2 “确定” / “取消”式对话框

如果“确定”按钮被点击，则`dialog.call`返回1；而如果“取消”被点击，则返回2。

注 仅用`MessageBox`函数，就可以创建出许多不同类型的对话框。详细内容请参阅微软关于`MessageBox`函数的文档。

Windows API提供了成百上千的函数，可以做一切事情，例如打印、改变桌面背景图片、创建复杂窗口等。在理论上说，你甚至可以用原始的Windows API函数创建整个Windows程序，不过这是一个浩大的工程。关于Windows API的更多信息，去维基百科是个很好的开始，入口地址为http://en.wikipedia.org/wiki/Windows_API。

11.4.2 控制Windows程序

尽管Windows API允许你访问Windows操作系统的底层函数，但也可以访问操作系统中其他程序开放的功能，这非常有用。让这一切成为可能的技术名为Windows自动化（Automation）。Windows自动化为程序提供了一种途径，可以激活另一个程序的功能，以便对前者进行某些功能操作。

对Windows自动化的访问是通过Ruby的WIN32OLE程序库（也包含在标准程序库中）实现的。如果你已经了解Windows自动化、COM或OLE技术，就会立刻感到Ruby的接口非常熟悉。即使你不熟悉，也能迅速读懂下面的代码：

```
require 'win32ole'

web_browser = WIN32OLE.new('InternetExplorer.Application')
web_browser.visible = true
web_browser.navigate('http://www.rubyinside.com/')
```

这段代码载入WIN32OLE程序库，并创建`web_browser`变量，它是对名为'`Internet-Explorer.Application`'的OLE自动化服务器的引用。该服务器由Windows自带的IE浏览器提供，而OLE自动化服务器让你可以远程控制浏览器的功能。在本例中，你让Web浏览器首先显示在屏幕上，然后再根据指令载入某个页面。

WIN32OLE本身不实现`visible`和`navigate`方法。这些动态方法由`method_missing`方法（这是个特殊方法，当在类中找不到预定义方法时则运行）在运行过程中处理，并传递给OLE自动化服务器。因此，可在Ruby中直接使用OLE自动化服务器开放的任何方法！

也可以对本例进行扩展，充分利用IE浏览器开放的更多方法：

```
require 'win32ole'

web_browser = WIN32OLE.new('InternetExplorer.Application')
web_browser.visible = true
web_browser.navigate('http://www.rubyinside.com/')

while web_browser.ReadyState != 4
  sleep 1
end
```

```
puts "Page is loaded"
```

本例使用ReadyState属性来确定IE浏览器载入页面是否完毕。如果页面还未载入，则Ruby睡眠一秒钟再继续检查。这样就可以等待远程操作完毕，再继续运行。

一旦页面载入完毕，IE浏览器即开放文档属性，让你可以对已载入网页的文档对象模型(Document Object Model, DOM)进行全面访问，和JavaScript的访问方式非常相似。例如：

```
puts web_browser.document.getElementById('header').innerHTML.length
```

1056

注 许多Windows应用程序实现了OLE自动化，可被Ruby以这种方式远程控制，但这是关于Windows开发高级指南的内容，超出了本书的范围。Win32Utils项目提供了更进一步的Windows相关程序库，请访问<http://rubyforge.org/projects/win32utils/>。

本节的目的是展示，尽管Ruby发源于UNIX类操作系统，但对Windows有相当的支持。你可以访问Windows API、使用OLE和OLE自动化，并访问DLL文件。许多Windows相关功能属于高级功能，超出了本书的范围。但我希望本节能够刺激你的欲望，去研究这一开发领域的更多内容。

11.5 线程

线程是执行线程的缩写。线程用来把程序的执行划分为多个可并发运行部分。例如，用于向成千上万人程序同时发送电子邮件的程序，可以把任务切分成20个不同线程，同时发送邮件。这样的并行机制比一个个串行快得多，特别是在不止一个CPU的系统上，因为此时各个线程可以在不同处理器上执行。另外，由于不必浪费时间等待远程机器的回应，而是可以继续转做其他操作，因此也加快了运行速度。

Ruby目前不支持传统意义上的线程。一般来说，线程能力由操作系统提供，不同的系统各有差别。但Ruby解释器为Ruby直接提供了线程能力，这表示在不同平台上Ruby的线程都能运行得很好，但与传统的系统级别线程相比，也缺少一些能力。

Ruby线程不是“真正”操作系统级别的线程，这是它的主要缺点之一，如果线程需要调用操作系统并等待回应，整个Ruby线程调度程序就会暂停下来。但对于一般操作，Ruby的线程系统还是很好用的。

注 在Ruby的未来版本中，可能会实现系统级别的线程。

11.5.1 基础Ruby线程实战

下面是Ruby线程实战的简单演示：

```
threads = []

10.times do
  thread = Thread.new do
```

```

    10.times {|i|print i;$stdout.flush;sleep rand(2)}
  end

  threads <<thread
end

threads.each {|thread|thread.join }
```

在上面的代码中，创建了一个数组来保存Thread对象，这样就可以轻松地跟踪管理这些线程。然后创建了10个线程，对于每个线程，把要执行的代码块发送给Thread.new方法，并把每个生成的线程加入到数组中。

注 当你创建线程时，它可以访问此时所处作用域的所有变量。不过，随后在线程内部创建的局部变量，则完全是该线程的局部变量。这与其他种类代码块的行为表现很相似。

线程创建后，在程序结束前要等待所有线程执行完毕。等待方法是对threads数组中所有线程对象进行循环，并调用每个线程的join方法。join方法让主程序等待，直到线程执行完毕再继续。这样一来，就可以保证所有线程都执行完毕，再退出程序。

上述程序的输出结果近似如下（如有变化是与随机的睡眠时间有关）：

```
00101200010010101212312124232512323453234336634544365546744548776557886689756765
67979789878889899999
```

本例创建了10个Ruby线程，其唯一任务是统计并随机睡眠。结果就是上述伪随机的输出。

线程可以做抓取网页、执行数学运算或发送邮件等操作，而不是睡眠。事实上，对几乎所有需要单个程序并发的情况，Ruby的线程都是理想工具。

注 在第15章你将用线程来创建服务器，该服务器为每个客户连接创建新的执行进程，从而可以实现简单的聊天系统。

11.5.2 高级线程操作

如你所见，创建和运行简单线程相当简单，但线程也提供了一系列高级功能。这些功能在下面子节中讨论：

等待线程结束回归

当用join方法等待线程结束时，可以指定等待的超时值（单位是秒）。如果线程在指定时间内未结束，则join方法返回nil。下面的例子展示了每个线程只给一秒钟运行：

```
threads.each do |thread|
  puts "Thread #{thread.object_id}didn't finish within 1s"unless thread.join(1)
end
```

获取全部线程列表

用Thread.list可以获取程序中运行的所有线程的全局列表。事实上，如果不想自己保存

线程，可以重写前面“基础Ruby线程实战”的例子，改成下面两行代码：

```
10.times {Thread.new {10.times {|i|print i;$stdout.flush;sleep rand(2)}}}
Thread.list.each {|thread|thread.join }
```

不过，如果你想有更多的控制，而不是只能操作整个程序的线程组，那么保留自己的线程列表就很重要，而且当需要用到join或其他功能时，你就会想把线程分别单独保存。

线程列表也包含主线程，它是表示主程序的执行线程。你可以把每个线程对象与Thread.main相比较，来确定哪个线程是主线程，如下所示：

```
Thread.list.each {|thread|thread.join unless thread == Thread.main }
```

在线程内部的线程操作

线程只是小巧而沉默的代码段，它们可以与Ruby线程调度程序进行沟通，并提供自身状态的更新信息。例如，线程可以停止自身：

```
Thread.new do
  10.times do |i|
    print i
    $stdout.flush
    Thread.stop
  end
end
```

本例中的线程每次被创建时，即向屏幕打印输出一串数字，然后停止自身的运行。随后，如果父程序要重新启动和恢复它，只能调用线程的run方法，如下所示：

```
Thread.list.each { |thread| thread.run }
```

线程也可告诉Ruby线程调度程序，它想把执行权转交给另一个线程。这种把控制权主动转让给另一线程的技术，通常称为协作式多任务（cooperative multitasking），因为线程或进程自己说可以把执行权转移给另一线程或进程。如果恰当地运用这种方法，可以让线程运行得更高效，因此你可以编写代码，让线程在理想的位置转移控制权。下面的例子展示了怎样从线程转让控制权：

```
2.times { Thread.new { 10.times { |i| print i; $stdout.flush; Thread.pass } } }
Thread.list.each { |thread| thread.join unless thread == Thread.main }
```

```
00112233445566778899
```

在本例中，执行流在两个线程之间来回跳转，从而形成特别的模式，如结果所示。

11.6 其他语言嵌入Ruby

作为动态和面向对象的编程语言，Ruby的设计目标不是传统意义上的高性能语言。现如今这不是特别的考虑要素，因为大多数任务都不是计算机密集型的。但在某些情况下，有些子功能还是需要纯粹的高性能。

在某些情况下需要极其高的性能，因此可以用更强大但表达力较弱的语言，编写计算密集型代码，然后从Ruby中调用这段代码。幸运的是，Ruby有个名为RubyInline的程序库，由Ryan

Davis和Eric Hodel开发，可以在Ruby代码中使用其他更强大的语言编写代码。它最常用的用途是用C或C++语言来编写高性能代码，本节我们将进行专门讨论。

用RubyGems在UNIX类平台（例如Linux和OS X）安装RubyInline非常简单：

```
gem install RubyInline
```

如果你没有安装gcc（一种C语言编译器），RubyInline对C语言的支持将无法使用，而且RubyInline本身也可能无法安装。请参阅你操作系统的文档，了解怎样安装gcc。

注 在编写本书时，有报道说RubyInline可以在微软Windows上运行，但需要作相当多的调整（尽管它可以在Cygwin环境中完美运行）。不过，这些只是高级用户尝试的东西，在本书出版时可能已经自动集成到程序库中。如果你是Windows用户又想用RubyInline，要么用Cygwin实现运行，要么查阅其官方网站<http://www.zenspider.com/ZSS/Products/RubyInline/>的相关说明。

11.6.1 为什么用C作为嵌入语言

C是一种通用的过程化编译型编程语言，由Dennis Ritchie在1970年代开发。它是世界上最广为使用的编程语言，也是当前几乎所有主流操作系统的基础。由于其纯粹的速度和灵活性，C（及其面向对象的姊妹语言C++）仍然是流行的编程语言。尽管Ruby这类语言的设计目的是轻松开发，而C则提供了大量底层访问功能，以及炫目的速度。这让C成为编写性能密集型程序库和函数的完美语言，让这些程序库和函数可供Ruby等其他编程语言调用。

注 本节不是C语言入门，因为这是一整本书的内容，如要学习C编程语言的更多内容，请访问http://en.wikipedia.org/wiki/C_programming_language。

11.6.2 创建基础方法或函数

对RubyInline以及C的威力的理想演示，是创建一个简单的方法（即C语言的函数）来计算阶乘。某个数的阶乘是从该数到1的所有数字的乘积。例如，8的阶乘是 $8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$ ，即40 320。

用Ruby计算阶乘很容易：

```
class Fixnum
  def factorial
    (1..self).inject { |a, b| a * b }
  end
end

puts 8.factorial
```

```
40320
```

你可以用性能基准度量的知识（详见第8章）来测试这个方法有多快：


```
require 'benchmark'

Benchmark.bm do |bm|
  bm.report('ruby:')do
    100000.times do
      8.factorial
    end
  end
end
```

```
      user  system    total    real
ruby:0.930000 0.010000 0.940000 (1.537101)
```

结果显示循环运行100 000次子程序，计算8的阶乘大约花1.5秒的时间，大约每秒运行66 666次循环。

我们再通过RubyInline用C语言编写阶乘方法：

```
class CFactorial
  class <<self
    inline do |builder|
      builder.c %q{
        long factorial(int value){
          long result =1,i =1;
          for (i =1;i <=value;i++){
            result *=i;
          }
          return result;
        }
      }
    end
  end
end
```

首先创建CFactorial类来容纳这个新方法，然后用inline do |builder|启动RubyInline环境，用builder.c处理%q{和}之间多行字符串组成的C代码。这样层层深入的原因，是RubyInline可以同时使用多种语言，因此需要先进入RubyInline环境，然后再指定代码关联到某种特定语言。

上例中实际的C代码由builder.c行开始。我们来专门看一下：

```
long factorial(int value){
  long result =1,i =1;
  for (i =1;i <=value;i++){
    result *=i;
  }
  return result;
}
```

这段代码定义了名为factorial的C函数，接受一个整数参数，并返回一个整数值。内部

逻辑是从1统计到指定数字，并把每个数字相乘，得到阶乘值。

11.6.3 性能基准度量：C和Ruby

我们已编写了基于C代码的阶乘函数，我们来对它做个性能基准度量，并与Ruby代码的版本相比较。下面是对两个不同函数（C和Ruby）进行性能基准度量的完整程序：

```
require 'rubygems'
require 'inline'
require 'benchmark'

class CFactorial
  class <<self
    inline do |builder|
      builder.c %q{
        long factorial(int value){
          long result =1,i =1;
          for (i =1;i <=value;i++){
            result *=i;
          }
          return result;
        }
      }
    end
  end
end

class Fixnum
  def factorial
    (1..self).inject {|a,b|a *b }
  end
end

Benchmark.bm do |bm|
  bm.report('ruby:')do
    100000.times {8.factorial }
  end

  bm.report('c:')do
    100000.times {CFactorial.factorial(8)}
  end
end
```

	user	system	total	real
ruby:	0.930000	0.010000	3.110000	(1.571207)
c:	0.020000	0.000000	0.120000	(0.044347)

C的阶乘函数快多了，转瞬即完！快了至少30倍。当然还可以对两种版本的代码都进行性能

改进，但这次度量足以展示编译型和解释型代码在性能上的巨大差异，以及Ruby的面向对象在性能方面的开销。

提示 要了解更多RubyInline的内容，请访问RubyInline官方网站<http://www.zenspider.com/ZSS/Products/RubyInline/>。

11.7 对Unicode和UTF-8的支持

对Ruby常见的抱怨是它对国际字符集的支持不怎么好。世界是多语言的，有时Ruby代码需要反映这一点。

Unicode是表示世界上各种文字系统的行业标准方法。在合理的标准环境下，对多种不同字母和字符集进行管理，Unicode是唯一可行的办法。

当人们抱怨Ruby对国际字符的支持时，他们通常是在抱怨缺少对Unicode的支持。Ruby 1.8版本确实如此，不过也有一些变通办法，我将在本节介绍。但在Ruby 1.9和2.0版本中这个问题已被解决了，Ruby原生支持Unicode和多字节字符，因此你可能不需要阅读本节内容。

注 关于Unicode的完整解释，以及它与软件开发的关系，请参阅<http://www.joelonsoftware.com/articles/Unicode.html>。Unicode的官方网站是<http://unicode.org/>，其中也有规范定义文本和进一步的详细内容。

主要问题是Ruby 1.8版本在默认情况下，把字符串中的字符只当成8比特位字符。这对拉丁/英语字母当然没有问题，因为大多数文本都可以表示为由8比特位字符组成。但对于中文或日文等语言，符号数量巨大，字符可能占用2个、3个甚至4个比特位。但Ruby只把每个字节看成一个字符，而不是把更大的字节组看成一个字符。由Ruby 1.8版本导致的这一问题很容易暴露出来，如下面的示例所示。

注 下面的示例在Ruby 2.0也能使用，和Ruby 1.8一样，不过在本书编写时Ruby 2.0还未发布。你可以在以下网址找到关于Ruby 2.0的多字节字符能力的内容：<http://redhanded.hobix.com/inspect/futurismUnicodeInRuby.html>。

对于英语（或更准确地说，拉丁字母字符），取出字符串的第一个字符很容易：

```
"test"[0].chr
```

```
t
```

但对于日语，则并非如此：

```
"権限の"[0].chr
```

```
<..nothing or a junk result..>
```

由于每个日语字符都用不止一个字节表示，因此Ruby无法对其正常操作，而只是用[0]取出第一个字节，而不是第一个字符，并试图把这个无意义的字节值转换回字符形式。

Ruby 1.8提供了一种名为jcode的变通办法。这种方法是Ruby自带的，让Ruby只支持UTF-8

(或其他字符编码,大多与日文有关,但我们在此不考虑其他字符编码)。UTF-8是用来表示Unicode字符的最常用系统,默认情况下,UTF-8字符只占用一个字节(例如英语字母字符),但如果需要,可以占用多于一个字节(例如日文、中文等)。

使用jcode可以让正则表达式对UTF-8生效:

```
$KCODE = 'u'
require 'jcode'

"権限の".scan(/./) do |character|
  puts character
end
```

随之而来的结果是:



```
権
限
の
```

把\$KCODE全局变量设为'u'(代表UTF-8)并载入jcode,让正则表达式意识到UTF-8字符的存在,并在scan循环中正确地返回每个多字节字符。不幸的是,这种意识仅对正则表达式有效,而其他Ruby方法,例如length、first、last和从字符串中提取单个字符的[]方法,都无法正确地处理这些字符串。

目前有些好项目可以让字符串的各种方法不仅对默认字符串有效,对UTF-8字符串也同样有效。这些项目大多采用正则表达式来达到这一效果,因此比Ruby内置方法慢得多。例如:

```
$KCODE = 'u'
require 'jcode'

class String
  def reverse
    scan(/./).reverse.join
  end
end
```

```
puts "権限の".reverse
```

下面是运行结果:

```
の限権
```

在本例中,覆写了String类中的reverse方法,改以使用scan的版本实现。这对UTF-8编码的字符串将会产生正确的结果。

注 关于这种技术方法的更多信息,请参阅<http://redhanded.hobix.com/inspect/closingInOnUnicodeWithJcode.html>。

在2006年中期,出现了一个为Ruby 1.8开发的更彻底的变通方法,名为ActiveSupport::Multibyte,现在已是Ruby on Rails框架的标准组成部分(也可以在<https://fngtps.com/>

projects/multibyte_for_rails下载这个程序库的独立版本)。ActiveSupport::Multibyte提供了名为chars的简单代理方法，可以访问字符串中的真正字符（而不是仅为每块8比特位）。这样你就可以编写下面的代码：

```
puts " 権限の ".chars.reverse
```

得到如下结果：

```
の 限 権
```

或

```
puts " 権限の ".chars[1..2]
```

结果如下：

```
限 の
```

关于ActiveSupport::Multibyte的完整讨论，请参阅<http://www.ruby-forum.com/topic/81976>。

注 在不同字符编码之间进行转换的功能，由iconv程序库提供，详见第16章。

11.8 小结

本章我们考察了一系列高级Ruby主题，从动态代码生成，到以C编程语言编写高性能函数。本章是Ruby程序员都应熟悉的一般Ruby知识的最后一章。在第12章我们将采取另一种方式，来开发一个完整的Ruby应用程序，与第4章的做法非常相似。

我们来回顾一下本章涵盖的主要概念：

- 绑定：绑定是以对象形式表示的某个作用域的（执行）环境。
- 分支：当程序的实例把自身复制为两个进程时，其中一个为父进程，另一个为子进程，两个进程都继续执行。
- 感染的数：来源无法完全信任或来源未知的数据。
- 安全级别：不同的安全级别，导致Ruby解释器对代码可以处理和执行什么功能，有不同的限制。
- Win32API：是一套Ruby程序库，可以访问Windows API。Windows API是一组程序库，提供了可以访问Windows内核、图形界面、控制程序库、网络服务和用户界面的函数。
- Windows自动化（也叫OLE自动化）：是这样一套系统，它允许Windows应用程序把自己注册为服务器，可以让其他程序远程控制。如想了解更多内容，请访问http://en.wikipedia.org/wiki/OLE_Automation。
- 线程：是独立的执行“线”，相互之间同步运行。Ruby的线程完全由Ruby解释器实现，但一般情况下线程也可在操作系统级别运行，是应用程序开发的一种常用工具。
- C语言：是1970年代开发的一种高性能的编译型语言，用在世界上大多数操作系统和底层软件中。也可在Ruby中通过RubyInline程序库使用C代码。
- RubyInline：是由Ryan Davis开发的一套Ruby程序库，可以轻松地在Ruby代码中嵌入C代

码，从而让Ruby可以轻松地访问高性能的C函数。

- 字符编码：是一种系统，也是一种编码，用于把字符（不管是罗马字母、中文汉字还是阿拉伯字母）编码为一组数字，以便让计算机用来表示字符。
- UTF-8：是Unicode的8字节变体，这种字符编码支持Unicode标准的所有字符。它支持可变长字符，原用于支持ASCII编码，也可以用最多四个字节，来表示其他字符集的字符。

现在可以进入到第12章，我们将运用本书到目前为止介绍的知识，开发一个完整的Ruby程序。



第12章 综合演练：开发更大型的Ruby应用

本章我们将退后一步，从对Ruby单方面功能的关注，转到用我们目前为止已经学到的知识，开发一个完整的程序。我们将关注应用开发在结构方面的考虑，并考察灵活的程序结构将为我们带来怎样的长期利益。

在阅读本章时，重要的一点是，记住应用程序本身不如开发理念更重要。我们将快速介绍程序开发的大部分相关领域，例如流程图、测试和简单的重构。这些技术方法有助于创建一定规模的应用程序。

12.1 构建机器人小程序

在开始编码之前，我们要了解一下将要构建什么，为什么要构建，以及打算怎样构建。

12.1.1 什么是机器人小程序

本章我们将构建一个机器人（robot），这不是像《太空漂流记》（Lost in Space）中的那样的科幻意义上的机器人，而是一个可以与我们的计算机程序。这种类型的程序通常称为机器人小程序（bot）或交谈机器人小程序（chatbot）。近来许多不同的软件和工具都有机器人小程序，AOL流行的即时消息软件就内置两个机器人小程序，作为配置定义的默认朋友。可以向它们提问，咨询礼品建议和电影场次。简言之，这有点像与客户服务接待员交谈，只不过这个接待员完全是自动化的。

你可能熟悉你自己机器上的机器人小程序。微软Office自带了Office助手“Clippy”并默认开启，许多网站也有自动化交谈机器人小程序（宜家的网站提供了绝好的例子）。

机器人小程序起源于20世纪60年代，麻省理工学院（MIT）有个名叫Joseph Weizenbaum的科学家开发了一个名为ELIZA的机器人小程序，最终极为流行，1980年至1990年大多数计算机用户都见过这样或那样的表现形式，即许多流行的“与你的计算机交谈”式程序。

与ELIZA式机器人小程序可以交谈的内容虽然不激动人心，但也相当有趣，如图12-1所示。ELIZA机器人小程序所用的一般机制，是把你说的内容扭转一下，变成陈述句或疑问句。例如，如果你说“我很烦”，ELIZA会回应“你烦了多久了？”或“你为什么烦？”。这样把用户输入的原句反弹回来，看起来有点粗鲁，但只因其深思熟虑式的本质，人们常常被骗得相信，以为他们正与某个有智能的东西交谈（这叫做ELIZA效应）。

我们的机器人小程序不会像ELIZA一样——也就是说，不会是ELIZA的克隆——而会带有一些相同的功能，并使用一些类似的技术。我们也将考察未来怎样用其他功能扩展这个机器人小程序。

注 如果想了解或尝试一些因特网版本的ELIZA，请访问<http://en.wikipedia.org/wiki/ELIZA>。

```
Eliza: Hi. I'm Eliza. Tell me your problems!
You : I don't have any problems!
Eliza: We have to find the reason why you don't have any problems!
You : How can there be a reason for that?
Eliza: Why do you ask?
You : It was a rhetorical question.
Eliza: Perhaps you would like to talk about a problem?
You : Okay, let's talk about my book.
Eliza: What does this mean for you, saying " Okay, let's talk about your book " ?
You : Huh?
Eliza: I understand!
You : I'm glad about that.
Eliza: How long are you glad about that?
You : Five seconds?
Eliza: I'm not sure wether I understand you!
You : And you sure can't spell.
Eliza: Come on! Explain your thinkings!
```

图12-1 与在线ELIZA机器人小程序交谈的演示

12.1.2 为什么要构建机器人小程序

开发机器人小程序的好处在于，它既可以很简单，也可以很复杂，随意愿而定。到本章结束时，我们将介绍扩展机器人小程序的方法，其初始结构相当简单。

在构建机器人小程序时，将使用本书到目前为止介绍的大多数技术方法。你会做一些测试和文档化，也会用到类和复杂数据结构。还会用到文件来保存机器人小程序所用信息，并考察怎样用HTTP服务器和CGI脚本，让机器人小程序对大众开放。

机器人小程序可以有一些趣味和体验，而开发联系人信息管理工具（举例）则不那么有趣，即使此类系统使用了与机器人小程序类似的技术。你还可以实现测试、文档化、类和存储系统，但最终是一个有趣的结果，有无限的扩展和改进空间。

12.1.3 怎样构建

本章主要关注怎样使机器人小程序的每个功能段相互耦合，这是在开发某类应用程序时的重要设计决策，如果将来准备扩展它们的话。这个机器人小程序的开发目标是尽可能地容易扩展或修改，从而可以对它进行定制修改、增加功能，或制作自己的机器人小程序。

用交谈机器人小程序的一般操作术语来说，你要开发的机器人小程序将位于某个类中，你可以通过创建新实例，轻松地复制机器人小程序。在创建机器人小程序时，除了内含的类的代码逻辑（它可能是“空白的”），你可以传送一个特殊的数据文件，为它提供一套知识集和一套响应集，以便与用户交谈时使用。用户通过键盘进行输入，但输入机制将保持足够的灵活性，以便机器人小程序可以轻松地使用网站或其他媒介。

机器人小程序开始只有几个公共方法。它需要具备把数据文件载入内存、接受用户输入并返回响应信息的能力。在表相之下，机器人小程序需要能解析用户“说”什么，并能够构建出相应的回应。因此，第一步是开始语言处理和字词识别。

12.2 创建文本处理工具程序库

接受“我很烦”之类的用户输入，并将其转换成“你为什么烦？”之类的回应，需要几个

步骤。第一步是进行一些预处理——即让文本更容易解析——例如整理文本，进行词语扩展，例如把“I'm”扩展为“I am”，把“you're”扩展为“you are”等。下一步是把输入内容切分为句子和字词，选择最佳的回应语句，最后从数据文件中寻找与输入内容相匹配的回应。

这些基本步骤如图12-2的流程图所示。

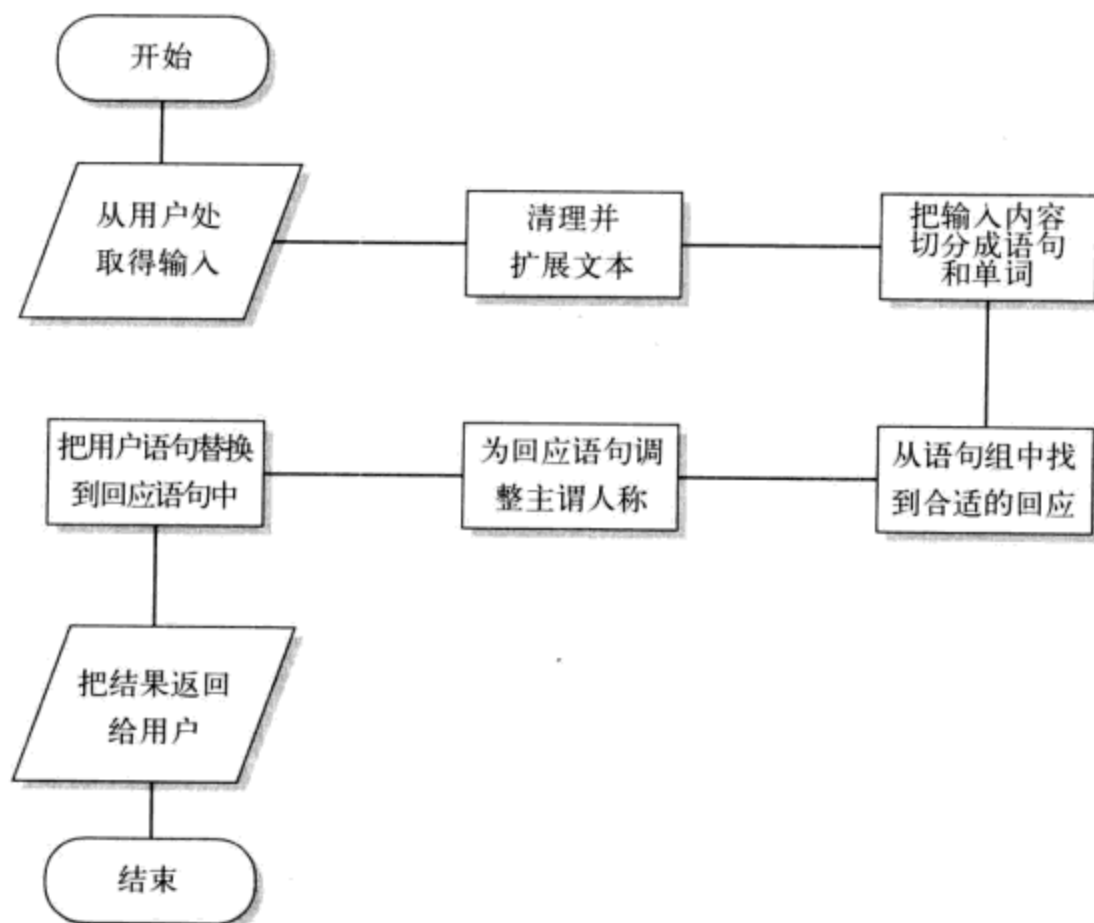


图12-2 机器人小程序的文本处理系统简单操作的基本流程图

注 流程图是某个系统（如计算机程序）中相关步骤的图形化表示。形成如图12-2所示的流程图，可以帮助定义过程中的步骤，从而更容易把程序的构想与实际代码连接起来。关于流程图及其所用符号，以及怎样使用流程图，请参阅<http://en.wikipedia.org/wiki/Flowchart>。

在这些语言任务中，有些是足够通用的，可以用于其他应用程序，因此你可以为其开发一个简单的程序库。这样就让机器人小程序的代码更简单，还形成了程序库，可在需要时用在其他应用程序中。专用于机器人小程序的代码逻辑和方法，可以放在机器人小程序的源代码中，而处理文本的通用方法则可以放在程序库中。

本节介绍简单程序库开发，包括测试和文档化。

12.2.1 构建WordPlay程序库

你可以把这个文本操作和处理的程序库命名为WordPlay，因此请创建名为wordplay.rb文件，其中是一个简单的类：

```
class WordPlay
```

```
end
```

现在已经建立了程序库的主文件，下面要开始实现一些文本操作和处理功能，将用于机器人小程序，但也是与具体程序无关的通用功能。（第6章已深入介绍了类的构造方法。）

把文本切分成语句

机器人小程序和其他程序一样，只对单行语句输入感兴趣。因此，必须只接受每个输入行的第一句。不过与其具体取出第一句，不如把输入内容切成成语句，然后再选择第一句。这么做的原因是可以使用某个通用的语句切分方法，无须每次都创建独立的解决方案。

为Ruby的String类创建sentences方法，以便让程序代码保持清爽。当然也可以在WordPlay类中创建类方法，如WordPlay.sentences(our_input)，但不如our_input.sentences看起来更直观、更面向对象，因为这里的sentences是String类的方法。

```
class String
  def sentences
    gsub(/\n|\r/, ' ').split(/\.\s*/)
  end
end
```

你可以简单测试如下：

```
%q{Hello. This is a test of
basic sentence splitting. It
even works over multiple lines.}.sentences
```

```
["Hello", "This is a test of basic sentence splitting", "It even works over
multiple lines"]
```

把语句切分成字词

还需要这个程序库把语句切分成字词。和sentences方法一样，也向String类增加words方法：

```
class String
  def words
    scan(/\w[\w\'\-]*/)
  end
end
```

```
"This is a test of words' capabilities".words
```

```
["This", "is", "a", "test", "of", "words'", "capabilities"]
```

可以把words方法与sentences方法并用进行测试：

```
%q{Hello. This is a test of
basic sentence splitting. It
even works over multiple lines}.sentences[1].words[3]
```

```
test
```


该测试用`sentences[1]`挑选出第二个句子，然后用`words[3]`挑选出第四个字词——请记住，数组都是从零开始计数的。（本节介绍的切分方法在第3章也有讲述。）

字词匹配

把新方法 with 数组的现有方法并用，从而取出匹配某词的语句，如下例所示：

```
hot_words = %w{test ruby}
my_string = "This is a test.Dull sentence here.Ruby is great.So is cake."
my_string.sentences.find_all do |s|
  s.downcase.words.any?{|word|hot_words.include?(word)}
end
```

在本例中，定义了两个“热”词，用于找到包含热词的语句，并搜索`my_string`中的语句，查找是否包含这两个热词。具体做法是看语句中是否有`hot_words`数组中包含的字词。

有经验的读者会猜想，在此情况下是否可以使用正则表达式。当然可以，但这里的重点的是清晰表达出代码逻辑，以便容易扩展和修改。如果愿意，还可以根据原字词数组和去掉热词后字词数组的不同长度，来排出哪一个匹配的热词更多。如果打算修改机器人小程序（或其他任何使用WordPlay程序库的软件），挑出并处理最重要语句，而不是只对第一个语句进行操作，则这种方法很有用。例如：

```
def self.best_sentence(sentences,desired_words)
  ranked_sentences =sentences.sort_by do |s|
    s.words.length - (s.downcase.words - desired_words).length
  end

  ranked_sentences.last
end
```

这个类方法接受语句数组和“目标字词”数组作为参数，然后按每个语句与目标字词有多少差异排序。如果差异很大，则表示语句中有很多目标字词。在`best_sentence`方法的最后，返回匹配次数最多的语句。

切换主语和宾语的代词

切换代词是指交换“你”和“我”、“我”和“你”、“我的”和“你的”以及“你的”和“我的”。这一简单交换让语句很容易用于回应。考虑一下，如果只把用户输入内容的代词进行切换，并反馈回去，会是什么情况？下表展示了一些例子：

输 入	回 应
我的 (My) 猫病了	你的 (Your) 猫病了
我 (I) 讨厌我的 (My) 车	你 (You) 讨厌你的 (Your) 车
你是 (You are) 个丑恶的机器人小程序	我是 (I are) 个丑恶的机器人小程序

这些对话都不复杂，前两句回应是合法的英语，正是机器人小程序可以使用的。第三句回应暴露出还需注意的地方：需要根据“I”和“you”调整“am”和“are”。

你将把代词切换功能作为类方法放在WordPlay类中，由于该方法不会链接到其他方法，不需要特别简洁，你可以把它放到WordPlay类中，而不是继续向String类增加更多方法。

```

def self.switch_pronouns(text)
  text.gsub(/\b(I am|You are|I|You|Your|My)\b/i)do |pronoun|
    case pronoun.downcase
      when "i"
        "you"
      when "you"
        "I"
      when "i am"
        "you are"
      when "you are"
        "i am"
      when "your"
        "my"
      when "my"
        "your"
    end
  end
end

```

该方法接受字符串文本参数，并对每个“I am”、“you are”、“I”、“you”、“your”或“my”执行替换操作。然后，用case语句构造把每个代词用对应身份的代词替换。（case/when语法初次出现在第3章中，其中也可找到其工作原理的深入解说。）

这样执行替换的原因是，对于每个代词你只能修改一次。如果你用了4次gsub来把所有“I’s”改为“you’s”、“you’s”改为“I’s”等，上一个gsub所作的修改就会被后一个覆盖。因此必须用一次gsub对输入内容的代词进行一个个扫描，而不是连续进行几次空白替换。

我们来检查一下结果：

```
WordPlay.switch_pronouns("Your cat is fighting with my cat")
```

```
my cat is fighting with your cat
```

```
WordPlay.switch_pronouns('You are my robot')
```

```
I am your robot
```

很容易发现这些结果中有一个异常：

```
WordPlay.switch_pronouns("I gave you life")
```

```
you gave I life
```

当“you”或“I”是句子的宾语而非主语时，“you”要变成“me”，“me”要变成“you”；而作为主语时，“I”要变成“you”，“you”要变成“I”。

我们不必陷入复杂的语句处理，确定哪个指的是主语，哪个指的是宾语，而是假定每个不在语句开头的“you”都是宾语，应变成“me”，如果“you”在语句开头，则应假定它是主语，并用“I”替换。这个新规则要求该方法作小小的改动：

```

def self.switch_pronouns(text)
  text.gsub(/\b(I am|You are|I|You|Me|Your|My)\b/i)do |pronoun|

```

```

case pronoun.downcase
  when "i"
    "you"
  when "you"
    "me"
  when "me"
    "you"
  when "i am"
    "you are"
  when "you are"
    "i am"
  when "your"
    "my"
  when "my"
    "your"
end
end.sub(/^me\b/i, 'i')
end

```

本例所作的修改，从表面上看起来很奇怪。switch_pronouns用于处理代词并进行正确的修改，原来在语句开头把“you”改为“me”，现在把其中的“me”改为“I”。这是通过末尾的链式sub实现的。

我们来试验一下：

```
WordPlay.switch_pronouns('Your cat is fighting with my cat')
```

```
my cat is fighting with your cat
```

```
WordPlay.switch_pronouns('My cat is fighting with you')
```

```
your cat is fighting with me
```

```
WordPlay.switch_pronouns('You are my robot')
```

```
i am your robot
```

```
WordPlay.switch_pronouns('I gave you hope')
```

```
you gave me hope
```

```
WordPlay.switch_pronouns('You gave me hope')
```

```
i gave you hope
```

成功了！

如果你极其倾向于此，可以只用这个方法，创建一个非常烦人的机器人小程序。考虑下面的简单例子：

```

while input = gets
  puts '>> ' + WordPlay.switch_pronouns(input).chomp + '?'
end

```

```
end
```

```
I am ready to talk
>> you are ready to talk?
yes
>> yes?
You are a dumb computer
>> I am a dumb computer?
```

很明显，下面还有事情要做！

12.2.2 测试该程序库

当构建大型应用程序，或构建其他应用程序所依赖的程序库时，必须确保一切都通过完整测试。在第8章中介绍了用Ruby的单元测试功能进行简单测试，在此你可以用相同的方法对WordPlay进行测试。

你将使用与第8章相同的过程。请创建一个名为test_wordplay.rb文件，与wordplay.rb放在同一目录，并编写以下基本代码结构：

```
require 'test/unit'
require 'wordplay'

class TestWordPlay < Test::Unit::TestCase
end
```

运行这段脚本将出现错误，因为还没有定义测试代码。我们来编写一些。

测试语句划分

要向test_wordplay.rb中加入成组的测试断言，只须创建以test_开头命名的方法。创建一个简单测试方法来测试语句划分是很容易的：

```
def test_sentences
  assert_equal(["a","b","c d","e f g"],"a.b.c d.e f g.".sentences)

  test_text =%q{Hello.This is a test
of sentence separation.This is the end
of the test.}
  assert_equal("This is the end of the test",test_text.sentences [2 ])
end
```

第一个断言测试样例语句"a. b. c d. e f g."是否被分隔成各组成成分的“语句”。第二个断言使用更长的预定义文本字符串，并确保第三个语句被正确标识出来。

注 理想情况下，应该用几个更多测试和更复杂的例子（例如以多个句号、逗号和其他奇怪符号结束的语句），对这个简单断言组合进行扩展。由于这些额外示例不会展示更多Ruby功能，因此在此不作介绍，但请自由尝试！

测试字词划分

测试words方法是否工作正常，比测试sentences方法更简单：

```
def test_words
  assert_equal(%w{this is a test},"this is a test".words)
  assert_equal(%w{these are mostly words},"these are,mostly,words".words)
end
```

这些断言都很简单。把语句拆分成字词，并与由这些字词组成的预定义数组进行比较，这些断言都能测试通过。

测试优先式开发为什么是个好主意，最引人注意的一个原因，是容易看出先编写测试代码，然后再用测试通过或失败作为指示，表明是否正确完成了words代码的开发。这是一个高级的编程理念，如果这样编写测试代码“打动”你的话，它是值得牢记于心的。

测试最佳语句选择

还需要测试WordPlay.best_sentence方法，因为机器人小程序将用它来选择与用户输入最密切相关的语句：

```
def test_sentence_choice
  assert_equal('This is a great test',
    WordPlay.best_sentence(['This is a test',
                           'This is another test',
                           'This is a great test'],
                           %w{test great this}))
  assert_equal('This is a great test',
    WordPlay.best_sentence(['This is a great test'],
                           %w{still the best}))
end
```

该测试方法执行一个简单断言，即正确的语句是从三个选项中选择。三个语句提供给WordPlay.best_sentence方法，并提供了目标关键字“test”、“great”和“this”，因此第三个语句应该是最佳匹配。第二个断言确保WordPlay.best_sentence方法即使没有匹配也返回一个语句，因为此时任何语句都是“最佳”匹配。

测试代词切换

在开发switch_pronouns方法时，用到一些模糊的语法规则，因此测试是确保这些规则至少对基本语句有效的根本保障：

```
def test_basic_pronouns
  assert_equal("i am a robot", WordPlay.switch_pronouns("you are a robot"))
  assert_equal("you are a person", WordPlay.switch_pronouns("i am a person"))
  assert_equal("i love you", WordPlay.switch_pronouns("you love me"))
end
```

这些简单断言证明，“you are”、“I am”、“you”和“me”短语都被正确切换。也可以创建单独的测试方法，来执行一些更复杂的断言：

```
def test_mixed_pronouns
  assert_equal("you gave me life", WordPlay.switch_pronouns("i gave you life"))
  assert_equal("i am not what you are", WordPlay.switch_pronouns("you are not
  what i am"))
  assert_equal("i annoy your dog", WordPlay.switch_pronouns("you annoy my dog"))
end
```



```
end
```

这些例子更复杂，但证明了switch_pronouns方法可以处理多个代词的一些更复杂情况。构建让switch_pronouns失败的测试代码：

```
def test_complex_pronouns
  assert_equal("yes, i rule", WordPlay.switch_pronouns("yes, you rule"))
  assert_equal("why do i cry", WordPlay.switch_pronouns("why do you cry"))
end
```

这些测试都失败了，因为它们回避了用来确保“you”在正确情况下被转为“me”和“I”的技巧。在这些情况下，它们应该都变成“I”，但因“I”不是位于语句的开头，因此变成了“me”。必须注意，简单语句一般都能正常，而疑问句或更复杂的语句则会失败。不过，对于本机器人小程序的目的来说，这些基本替换已经足够了。

如果你只关注怎样生成精确的语言处理程序，则可以用这些测试来指导开发，也可能在开发程序库时用这些技术，以便在你自己的项目中处理这样的边界情况（edge cases）。

12.2.3 WordPlay程序库的源代码

对目前来说，你的WordPlay萌芽程序库已经完成，可进一步优化源代码，使其更简单、更易读。下一步我将展示该程序库的源代码，以及关联的单元测试文件。另外，代码也包括每个类定义和方法定义之前的注释，因此你可以用RDoc来生成HTML文档文件，如第8章所示。

注 请记住，本书源代码可在<http://www.apress.com>网站的Source Code/Download区找到，因此不必直接从书中输入代码。

wordplay.rb

下面是WordPlay程序库的代码：

```
class String
  def sentences
    self.gsub(/\n|\r/, '').split(/\.\s*/)
  end
  def words
    self.scan(/\w [\w \'\-]*/)
  end
end

class WordPlay
  def self.switch_pronouns(text)
    text.gsub(/\b(I am|You are|I|You|Me|Your|My)\b/i) do |pronoun|
      case pronoun.downcase
      when "i"
        "you"
      when "you"
        "me"
      when "me"

```

```

        "you"
      when "i am"
        "you are"
      when "you are"
        "i am"
      when "your"
        "my"
      when "my"
        "your"
      end
    end.sub(/^me \b/i, 'i')
  end

  def self.best_sentence(sentences, desired_words)
    ranked_sentences = sentences.sort_by do |s|
      s.words.length - (s.downcase.words - desired_words).length
    end

    ranked_sentences.last
  end
end

```

test_wordplay.rb

下面是与WordPlay程序库相关的测试包代码：

```

require 'test/unit'
require 'wordplay'

#Unit testing class for the WordPlay library
class TestWordPlay <Test::Unit::TestCase

  #Test that multiple sentence blocks are split up into individual
  #words correctly
  def test_sentences
    assert_equal(["a", "b", "c d", "e f g"], "a.b.c d.e f g.".sentences)

    test_text = %q{Hello.This is a test
of sentence separation.This is the end
of the test.}
    assert_equal("This is the end of the test", test_text.sentences [2 ])
  end

  #Test that sentences of words are split up into distinct words correctly
  def test_words
    assert_equal(%w{this is a test}, "this is a test".words)
    assert_equal(%w{these are mostly words}, "these are,mostly,words".words)
  end
end

```

```
#Test that the correct sentence is chosen,given the input
def test_sentence_choice
  assert_equal('This is a great test',
    WordPlay.best_sentence(['This is a test',
                           'This is another test',
                           'This is a great test'],
                           %w{test great this}))
  assert_equal('This is a great test',
    WordPlay.best_sentence(['This is a great test'],
                           %w{still the best}))
end

#Test that basic pronouns are switched by switch_pronouns
def test_basic_pronouns
  assert_equal("i am a robot",WordPlay.switch_pronouns("you are a robot"))
  assert_equal("you are a person",WordPlay.switch_pronouns("i am a person"))
  assert_equal("i love you",WordPlay.switch_pronouns("you love me"))
end

#Test more complex sentence switches using switch_pronouns
def test_mixed_pronouns
  assert_equal("you gave me life",
    WordPlay.switch_pronouns("i gave you life"))

  assert_equal("i am not what you are",
    WordPlay.switch_pronouns("you are not what i am"))
end
end
```

12.3 构建机器人小程序的核心功能

在上节中，你把WordPlay程序库组装起来，提供了机器人小程序需要的一些功能，例如基本的语句和字词划分。现在可以开始向机器人小程序本身添加细节了。

用Bot类创建机器人小程序，用该类可以创建多个机器人小程序实例，并赋予不同的名字和数据集，它们可以互相独立地运行。这是最清晰的结构，因为这样可以把机器人小程序的代码逻辑与其交互方式分离开来。例如，如果编写的Bot类放在bot.rb文件中，即可编写Ruby程序，让用户通过机器人小程序反转键盘输入，这么做非常简单，如下所示：

```
require 'bot'

bot =Bot.new(:name =>"Botty",:data_file =>"botty.bot")

puts bot.greeting
while input =gets and input.chomp !='goodbye'
  puts ">>"+bot.response_to(input)
end
puts bot.farewell
```

你将用这个简短的客户端程序作为创建Bot类的标准。在上例中，创建了bot对象并向其传递了一些参数，这样就可以使用机器人小程序的方法，将键盘输入的内容进行反转，并返回给用户。

在某些情况下，首先编写高层次的、更抽象的来表达最终用途代码，然后再编写底层的代码来满足高层代码，这是很用的的编程方法。这与测试优化式开发不同，不过原理是类似的。首先编写最简单的、最抽象的代码，然后再逐步添加细节。

下面我们通过常规对话来看一下，机器人小程序如何操作，并开始一个个地开发所需功能。

12.3.1 程序的生命周期和组成部分

在图12-2中，显示了机器人小程序被询问并回应用户输入的情况。但在图12-3中显示了机器人小程序总体的生命周期，以及用户对它的访问，这正是我们要开发的。

整个应用程序将由四部分组成：

1. Bot类，放在bot.rb文件中，其中包含机器人小程序的所有代码逻辑和子类。
2. WordPlay程序库，放在wordplay.rb文件中，其中包含WordPlay类和对String类的扩展。
3. 简单的“客户端”程序，它创建机器人小程序并让用户可与之交互。首先创建简单的键盘输入客户端，本章后文也将考察一些替代方式。
4. 一个辅助程序，用来轻松生成机器人小程序的数据文件。

图12-3展示了客户端示例程序及其关联的机器人小程序对象的基本生命周期。客户端程序创建机器人小程序实例，然后持续要求用户输入，并将输入内容传递给机器人小程序。回应内容被打印输出到屏幕，该循环一直继续到用户决定退出。

下面将开始组装Bot类，并了解机器人小程序怎样找到数据并进行处理。

12.3.2 机器人小程序的数据

关于机器人小程序，第一个考虑事项就是它到哪里获取数据。机器人小程序的数据包括字词替换的信息，以便进行预处理，以及可用来生成回应的无数关键字和短语。

数据结构

机器人小程序的数据放在散列表中，如下所示：

```
bot_data = {
  :presubs =>[
    ["dont", "don't"],
    ["youre", "you're"],
    ["love", "like"]
  ],

  :responses =>{
    :default =>[
      "I don't understand.",
      "What?",
      "Huh?"
    ]
  }
}
```

```

    ],
    :greeting =>["Hi.I'm [name ].Want to chat?"],
    :farewell =>["Good bye!"],
    'hello' =>[
      "How's it going?",
      "How do you do?"
    ],
    'i like *'=>[
      "Why do you like *?",
      "Wow!I like *too!"
    ]
  }
}

```

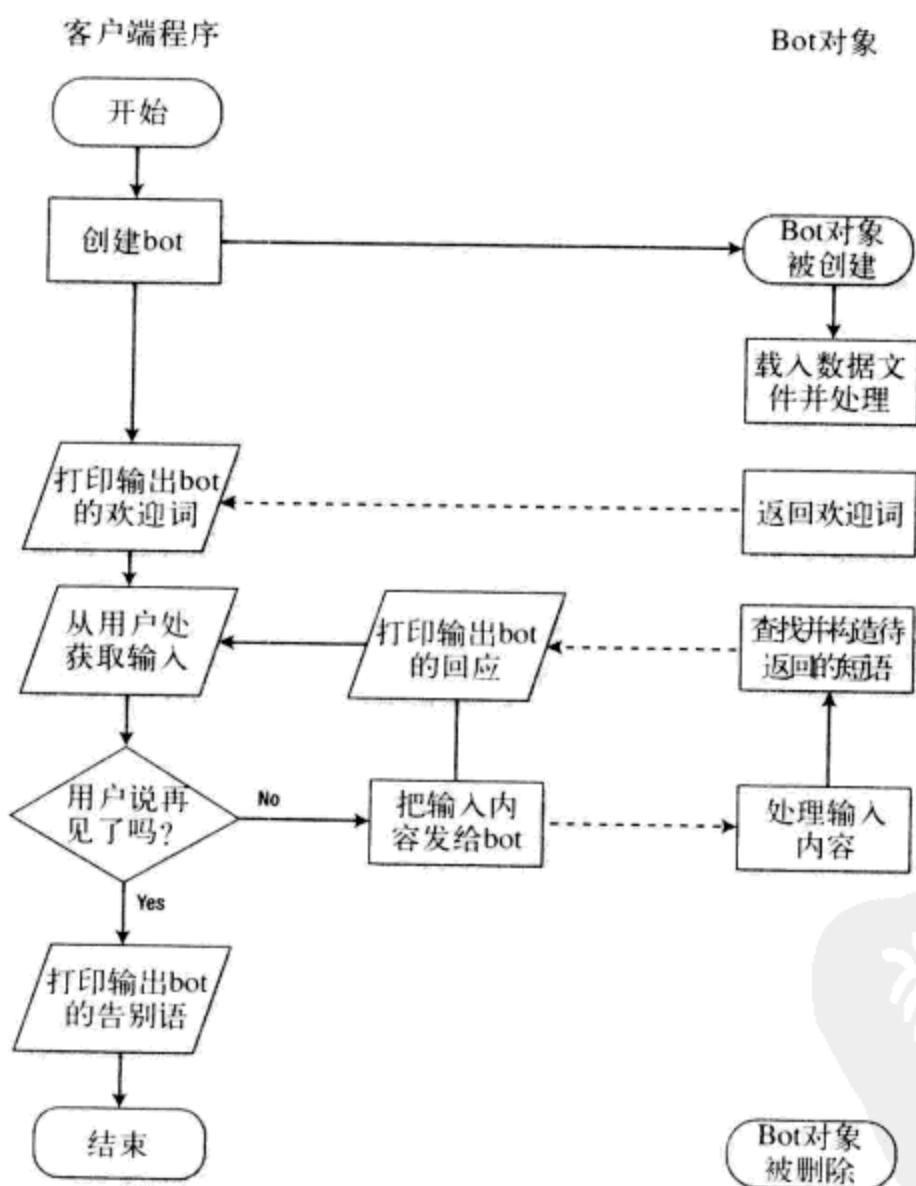


图12-3 关于机器人小程序客户端和对象的示例生命周期的简单流程图

主散列表的两个父元素：`:presubs`和`:responses`。`:presubs`元素指向一个由数组构成的数组，其中包含在机器人小程序形成回应之前，对用户输入进行的替换。在本例中，机器人小程序将对一些缩写进行扩展，也会把“love”改为“like”。为什么要这么做，当你看到`:responses`时就会明白了。

注 这个数据结构有意采用少量数据，以节省讨论空间。到本章结束时，你将有更完整的数据供机器人小程序使用。这种风格的数据结构在第3章也有介绍。

`:responses`指向另一个散列表，它包含`:default`、`:greeting`、`:farewell`、`'hello'`和`'i like *'`等名字的散列表。这个散列表包含各种不同短语，供机器人小程序在生成回应时直接使用，或将它们作为模板来创建完整的短语。赋值给`:default`的数组包含一些短语，供机器人小程序在无法根据输入内容判断应该说什么的时候，从而在这里随机挑选使用。与`:greeting`和`:farewell`关联的内容则包含一般的欢迎词与告别语。

更有趣的是与`'hello'`和`'i like *'`关联的数组，当输入内容与每个数组的键相匹配时，则使用这些短语。例如，如果用户说“hello computer”，则与`'hello'`相匹配，即从该数组中随机选择回应。如果用户说“i like computers”，则与`'i like *'`相匹配，星号用于代替用户输入的后续内容（在“i like”之后），并放在机器人小程序的输出短语里。如果用了第二个短语的话，就可能产生诸如“Wow! I like computers too”的输出结果。

在外部保存数据

用散列表可以方便地访问数据（与其他相比，例如数据库），在选择语句和执行匹配时速度也很快。不过，因为机器人小程序类需要处理多个数据集，因此需要把每个机器人小程序的散列表数据放在文件中保存，在机器人小程序启动时再选择载入。

在第9章你了解了对象持久化的概念，即Ruby数据可被“冻结”并保存。用到一个名叫PStore的程序库，它可以把Ruby数据结构放在不可读的二进制格式中保存，而另一个程序库名为YAML，可以把数据结构放在可读的特别格式文件中保存。对于本项目来说，要用YAML，因为你想在运行过程中对数据文件进行修改，以便无须每次都要重新构造全新文件，才能改变机器人小程序要说的话，或测试新短语的效果。

可以手工创建数据文件，并让Bot类载入。但为了简化工作，可以创建一个小程序，帮你创建初始数据文件，如第9章所述的那样。这个小程序的理想文件名应该是`bot_data_to_yaml.rb`：

```
require 'yaml'

bot_data = {
  :presubs => [
    ["dont", "don't"],
    ["youre", "you're"],
    ["love", "like"]
  ],

  :responses => {
    :default => [
      "I don't understand.",
      "What?",
      "Huh?"
    ],

    :greeting => ["Hi.I'm [name ].Want to chat?"],
    :farewell => ["Good bye!"],
```

```

    'hello' =>[
      "How's it going?",
      "How do you do?"
    ],
    'i like *'=>[
      "Why do you like *?",
      "Wow! I like *too!"
    ]
  }
}

#Show the user the YAML data for the bot structure
puts bot_data.to_yaml

#Write the YAML data to file
f =File.open(ARGV.first || 'bot_data', "w")
f.puts bot_data.to_yaml
f.close

```

这个简短的程序把机器人小程序的数据定义在bot_data散列表中，然后在屏幕上展示YAML表示方式，再写入到文件中。文件名由命令行指定，如果没有指定则用默认的bot_data。

ruby bot_data_to_yaml.rb

```

---
:presubs:
--dont
--don't
--youre
--you're
--love
--like
:responses:
  i like *:
    -Why do you like *?
    -Wow! I like *too!
  :default:
    -I don't understand.
    -What?
    -Huh?
  hello:
    -How's it going?
    -How do you do?
  :greeting:
    -Hi. I'm [name ]. Want to chat?
  :farewell:
    -Good bye!

```

请注意，YAML数据是纯文本，你可以直接编辑文件，或修改bot_data结构再重新运行

bot_data_to_yaml.rb程序。从此处开始，我们假定已经运行过该程序，并在当前目录中生成上述名为bot_data的YAML文件。

现在有了简单的数据文件，可以开始构造Bot类及其初始化方法了。

12.3.3 构建Bot类和数据载入器

我们来创建bot.rb文件，并开始编写Bot类：

```
require 'yaml'
require 'wordplay'

class Bot
  attr_reader :name

  def initialize(options)
    @name = options[:name] || "Unnamed Bot"
    begin
      @data = YAML.load(File.read(options[:data_file]))
    rescue
      raise "Can't load bot data"
    end
  end
end
```

Initialize方法搭建每个新创建的对象，并用可选的散列表来填充两个类变量@name和@data。对@name的外部访问由彬彬有礼的attr_reader提供。File.open和read方法一起，打开数据文件并读入全部内容供YAML程序库处理。YAML.load把YAML数据转换成原始的散列表数据结构，并将其赋值给@data类变量。如果YAML数据文件打开失败，或YAML处理失败，则抛出异常，因为机器人小程序没有数据无法工作。

现在可以创建greeting和farewell方法，它根据机器人小程序的数据集，随机显示欢迎词和告别语。当人们开始使用机器人小程序时，或在机器人小程序客户端退出前，将使用这两个方法。

```
def greeting
  @data[:responses][:greeting][rand(@data[:responses][:greeting].length)]
end

def farewell
  @data[:responses][:farewell][rand(@data[:responses][:farewell].length)]
end
```

哎呀！这一点都不好看。你通过@data[:responses]来访问欢迎词（和告别语），但选择随机单个短语很快就变得很丑陋。看起来应该创建一个私有方法，用它在选定的回应组中检索随机短语：

```
private

def random_response(key)
```

```

    random_index = rand(@data[:responses][key].length)
    @data[:responses][key][random_index].gsub(/\[name\]/, @name)
  end
end

```

这个方法简化了从@data特定短语集中选择随机短语的工作。random_response方法的第二行代码执行替换，让包含[name]的回应把[name]替换成机器人小程序的名字。例如，示例欢迎词为“Hi. I'm [name]. Want to chat?”，但如果你创建的机器人小程序对象有指定的名字叫“Fred”，则输出内容将会是“Hi. I'm Fred. Want to chat?”。

注 请记住，私有方法是无法在类之外调用的方法。因为random_response方法只需在类的内部使用，因此把它设为私有方法再好不过了。

我们用random_response修改一下greeting和farewell方法：

```

def greeting
  random_response :greeting
end

def farewell
  random_response :farewell
end

```

把常用功能放到单独方法中不是很妙吗？这两个方法现在看起来简单多了，与前面的混乱代码相比，现在一眼就能看出其中的意义。

注 这种方法也可用于“丑陋”或复杂无比的代码，只需将其隐藏到一个方法中，再调用该方法即可。请尽量把复杂代码藏在后台，让前台代码看起来越简单越好。

12.3.4 response_to方法

Bot类的核心是response_to方法。它用于把用户输入传递给机器人小程序，得到回应并返回。不过，该方法本身应该比较简单，每个所需操作只占用一行代码（即对私有方法的调用，由这些私有方法完成具体操作）。

response_to方法必须执行下面几个动作：

1. 接受用户输入。
 2. 执行预处理替换，如机器人小程序数据文件章节所述。
 3. 把输入内容划分成语句，并选择关键词最多的语句。
 4. 根据回应语句集的键，搜索匹配内容。
 5. 对用户输入内容执行代词切换。
 6. 选择匹配的随机短语（如果没有匹配则选择默认短语），并对用户输入执行替换，再放到结果中。
 7. 返回完整的输出短语。
- 我们逐个来看每个动作。

接受输入并执行替换

首先，接受输入内容作为`response_to`方法的基本参数：

```
def response_to(input)
end
```

然后开始进行预处理，根据机器人小程序数据文件中的`:presubs`数组，对字词和短语进行替换。回想一下，`:presubs`是由数组构成的数组，其中指定了需要改为其他形式的字词和短语。这么做的原因是单个短语可以对应处理多个条目。例如，如果把所有“yes”都替换成“yeah”，那么不管用户说的是“yeah”还是“yes”，即使短语只匹配“yes”，返回结果都会显示相应短语。

因为你的目的是让`response_to`方法尽量简单，因此只需用一个方法调用来完成预处理即可：

```
def response_to(input)
  prepared_input = preprocess(input).downcase
end
```

现在可以实现私有方法`preprocess`的功能了：

private

```
def preprocess(input)
  perform_substitutions input
end
```

然后再实现`substitution`方法本身的功能：

```
def perform_substitutions(input)
  @data[:presubs].each { |s| input.gsub!(s[0], s[1]) }
  input
end
```

这段代码循环处理`:presubs`数组中定义每个替换，并用`gsub!`对输入内容进行替换。

此时值得考虑一下，为什么要用一串方法最终才调用到`perform_substitutions`？为什么不直接在`response_to`方法中调用它？

这么做的原因是想让代码逻辑与程序中其他逻辑尽量分开，这是大型程序的开发方式，可以更容易地扩展。例如，如果将来想执行更多的预处理工作，你只需为其创建方法，并从`preprocess`中调用即可，无须对`response_to`方法进行修改。尽管这看起来不够高效，但实际上从长期效果来看，代码更容易扩展和阅读。小小的冗余度，带来巨大的灵活性。你会在其他Ruby程序中看到许多类似的方法，这也是为什么在此强制使用这种方法。

选择最佳语句

按你的布置对输入内容完成预处理之后，该把它划分成语句并选择最佳语句了。你可在`response_to`方法中再加一行代码：

```
def response_to(input)
  prepared_input = preprocess(input).downcase
  sentence = best_sentence(prepared_input)
end
```


然后再实现私有方法best_sentence的功能：

```
def best_sentence(input)
  hot_words =@data [:responses ].keys.select do |k|
    k.class ==String &&k =~/^\\w+$/
  end

  WordPlay.best_sentence(input.sentences,hot_words)
end
```

首先，best_sentence根据:responses散列表的键，通过查看所有是字符串的键（你不想混入:default、:greeting或:farewell符号），或是单个字词的键，并收集形成由单个字词组成的数组。然后将这个列表用于本章前文开发的WordPlay.best_sentence方法，从用户输入中选择匹配最多“热词”的语句（如果有匹配的话）。

可以用自己喜欢的任何方式重写该方法，如果只想选择用户输入中的第一个语句，则很容易办到：

```
def best_sentence(input)
  input.sentences.first
end
```

或选择最长的语句怎么样？

```
def best_sentence(input)
  input.sentences.sort_by { |s| s.length }.last
end
```

通过把选择最佳语句的小段代码逻辑放在单独的方法中，我们再次修改了程序的工作方式，不再使用混乱的大型方法。

寻找匹配的短语

现在有了要解析的语句，也执行了替换，下一步是找出适合于回应的短语，并随机选择一个。我们对response_to方法再次进行扩展：

```
def response_to(input)
  prepared_input = preprocess(input.downcase)
  sentence = best_sentence(prepared_input)
  responses = possible_responses(sentence)
end
```

并实现possible_responses方法：

```
def possible_responses(sentence)
  responses =[]

  #Find all patterns to try to match against
  @data [:responses ].keys.each do |pattern|
    next unless pattern.is_a?(String)

    #For each pattern,see if the supplied sentence contains
    #a match.Remove substitution symbols (*)before checking.
```



```

    #Push all responses to the responses array.
    if sentence.match('\b'+pattern.gsub(/\*/, ' ')+'\b')
      responses <<@data [:responses ][pattern ]
    end
  end
end

#If there were no matches,add the default ones
responses <<@data [:responses ][:default ] if responses.empty?
#Flatten the blocks of responses to a flat array
responses.flatten
end

```

`possible_responses`方法接受一个语句作为参数，并用`:responses`散列表中的字符串键来检查是否有匹配。一旦发现匹配，则各个合适的回应内容被压入`responses`数组，然后把这个数组扁平化，返回单个数组。

如果未找到任何特定匹配，则使用默认值（在`:responses`用`:default`键）。

组装成最终的短语

现在`response_to`方法有了所有组成部分，可以把它组装成最终的回应短语了。我们从`response`中选择随机短语来使用：

```

def response_to(input)
  prepared_input = preprocess(input.downcase)
  sentence = best_sentence(prepared_input)
  responses = possible_responses(sentence)
  responses[rand(responses.length)]
end

```

如果要对替换过代词的语句进行任何切换，这个`response_to`版本将是最终版。不过，机器人小程序可以把用户输入的一部分放到回应中。以下是机器人小程序的一小段示例数据：

```

'i like *' => [
  "Why do you like *?",
  "Wow! I like * too!"
]

```

当用户说“I like”则匹配这个规则。第一个可选回应是“Why do you like *?”，其中包含一个星号，可用来把用户语句中的一部分替换进来，并与前文WordPlay中的代词切换方法并用。

例如，用户可能说“I like to talk to you”，如果切换了代词则为“You like to talk to me”。如果“You like”之后的部分被替换到第一个可选回应中，就得到“Why do you like to talk to me?”这是个绝妙的回应，强制用户继续输入并展示代词替换方法的威力。

因此，如果选中的回应包含星号（用作回应短语中的占位符），就需要把原来语句中的相应部分替换进来，并对这部分执行代词切换。

下面是`possible_responses`方法的新版本，修改的内容以粗体字显示：

```

def possible_responses(sentence)
  responses =[]

```

```

#Find all patterns to try to match against
@data [:responses ].keys.each do |pattern|
  next unless pattern.is_a?(String)

  #For each pattern,see if the supplied sentence contains
  #a match.Remove substitution symbols (*)before checking.
  #Push all responses to the responses array.
  if sentence.match('\b'+pattern.gsub(/\*/,'')+'\b')
    #If the pattern contains substitution placeholders,
    #perform the substitutions
    if pattern.include?('*')
      responses <<@data [:responses ][pattern ].collect do |phrase|
        #First,erase everything before the placeholder
        #leaving everything after it
        matching_section =sentence.sub(/^.*#{pattern}\s+/, '')

        #Then substitute the text after the placeholder,with
        #the pronouns switched
        phrase.sub('*',WordPlay.switch_pronouns(matching_section))
      end
    else
      #No placeholders?Just add the phrases to the array
      responses <<@data [:responses ][pattern ]
    end
  end
end

#If there were no matches,add the default ones
responses <<@data [:responses ][:default ] if responses.empty?

#Flatten the blocks of responses to a flat array
responses.flatten
end

```

`possible_responses`方法的这个新版本检查模式中是否含有星号，如有则抽取原语句相应部分内容，放到`matching_section`中，并切换其中的代词，然后把整块内容替换到每个相关短语中。

12.3.5 试用机器人小程序

你已经实现了Bot类的基本方法，下面我们来试用一下，然后再考虑怎样进一步扩展。第一步是准备一套更好的数据集供机器人小程序使用，以便对话能够更加迷人，不像本章前文的样例数据那样沉闷。

Fred：机器人小程序的人格

本节你将修改前面创建的`bot_data_to_yaml.rb`脚本，来为第一个机器人小程序生成YAML文件。它的名字叫Fred，你要生成的机器人小程序数据文件名为`fred.bot`。下面是

bot_data_to_yaml.rb的扩展版本，其中包含了更好的短语和替换内容：

```
require 'yaml'

bot_data = {
  :presubs =>[
    ["dont", "do not"],
    ["don't", "do not"],
    ["youre", "you're"],
    ["love", "like"],
    ["apologize", "are sorry"],
    ["dislike", "hate"],
    ["despise", "hate"],
    ["yeah", "yes"],
    ["mom", "family"]
  ],

  :responses =>{
    :default =>[
      "I don't understand.",
      "What?",
      "Huh?",
      "Tell me about something else.",
      "I'm tired of this.Change the subject."
    ],
    :greeting =>[
      "Hi.I'm [name ].Want to chat?",
      "What's on your mind today?",
      "Hi.What would you like to talk about?"
    ],
    :farewell =>["Good bye!", "Au revoir!"],
    'hello' =>[
      "How's it going?",
      "How do you do?",
      "Enough of the pleasantries!"
    ],
    'sorry' =>["There's no need to apologize."],
    'different' =>[
      "How is it different?",
      "What has changed?"
    ],
    'everyone *'=>["You think everyone *?"],
    'do not know'=>["Are you always so indecisive?"],
    'yes' =>[
      "At least you're positive about something!",
      "Great."
    ],
    'family' =>["Tell me about your family"],
```

```

'you are *'=>[
    "What makes you think I am *?",
    "Are you so sure I am *?"
],
'i am *'    =>[
    "Is it normal for you to be *?",
    "Do you like being *?"
],
'i do not *' =>["Why don't you *?"],
'what'      =>["Why do you ask?", "Why?", "I don't know.Do you?"],
'no'        =>[
    "Really?",
    "Fair enough."
],
'why does *' =>[
    "I don't know why *",
    "Do you already know the answer?"
],
'why can \'t i *'=>["Do you want to *?"],
'why can \'t you *'=>["Do you want me to *?"],
'hates *'    =>[
    "Why do you think they hate *?"
],
'hate *'     =>[
    "Why don't you like *?",
    "Why the dislike of *?"
],
'i like *'   =>[
    "Why do you like *?",
    "Wow!I like *too!"
]
}
}

```

```

#Show the user the YAML data for the bot structure
puts bot_data.to_yaml

```

```

#Write the YAML data to file
f =File.open(ARGV.first || 'bot_data', "w")
f.puts bot_data.to_yaml
f.close

```

如果用`ruby bot_data_to_yaml.rb fred.bot`来运行上面的程序，将得到`fred.bot`文件，它包含了与简单机器人对话所必需的数据。

初次真正的对话

下面的简单程序使用`Bot`类来读取用户键盘输入，并把机器人小程序的回应打印输出到屏幕：


```

require 'bot'

bot = Bot.new(:name => 'Fred', :data_file => 'fred.bot')

puts bot.greeting

while input = gets and input.chomp != 'end'
  puts '>> ' + bot.response_to(input)
end

puts bot.farewell

```

这与我们在实现Bot类之前构想的脚本几乎完全相同，用程序规定的方法名，并恰到好处地使用。（如果你想参考，Bot类的全套代码在下节提供。）

不必费什么力气，请试一下看看能否进行一场对话。下面是运行演示：

```

Hi.What would you like to talk about?
I don't know
>>Are you always so indecisive?
Yes,usually
>>Great.
I'm sorry!
>>There's no need to apologize.
So why can't you talk to me properly?
>>Do you want me to talk to you properly?
Yes
>>At least you're positive about something!
I don't know about that
>>Are you always so indecisive?
Woe is me,why can't I make decisions?
>>Do you want to make decisions?
Everyone does
>>You think everyone does?

```

你的机器人可以正常工作！这段对话可能有点不自然，并有意使用了数据集中包含的短语和字词，但有了这样的基本机制，并进一步扩展数据集，就可以进行相当复杂的对话。不幸的是，提供大型数据集超出了本书的范围。

下一节展示了简单机器人小程序的最终代码，可以看看怎样进一步扩展机器人小程序的功能。

12.4 机器人小程序主要代码清单

本节提供bot.rb文件中的全套Bot类源代码，包括可用于RDoc的额外文档。源代码中还包含一个简单的机器人小程序客户端，可以用它来与机器人小程序通过命令行键盘进行一对一的对话。

与本章到目前为止的示例不同，这些代码加了详细注释，推荐你至少浏览一遍下面的代码，以便对整个程序的运作有个直观感受。

注 也可以从<http://www.apress.com/>的Source Code/Download区找到这些代码清单并下载。

12.4.1 bot.rb文件

下面是主体Bot类的源代码：

```
require 'yaml'
require 'wordplay'

#A basic implementation of a chatterbot
class Bot
  attr_reader :name

  #Initializes the bot object, loads in the external YAML data
  #file and sets the bot's name. Raises an exception if
  #the data loading process fails.
  def initialize(options)
    @name = options[:name] || "Unnamed Bot"
    begin
      @data = YAML.load(File.open(options[:data_file]).read)
    rescue
      raise "Can't load bot data"
    end
  end

  #Returns a random greeting as specified in the bot's data file
  def greeting
    random_response(:greeting)
  end

  #Returns a random farewell message as specified in the bot's
  #data file
  def farewell
    random_response(:farewell)
  end

  #Responds to input text as given by a user
  def response_to(input)
    prepared_input = preprocess(input.downcase)
    sentence = best_sentence(prepared_input)
    reversed_sentence = WordPlay.switch_pronouns(sentence)
    responses = possible_responses(sentence)
    responses[rand(responses.length)]
  end

  private
```

```

#Chooses a random response phrase from the :responses hash
#and substitutes metadata into the phrase
def random_response(key)
  random_index =rand(@data [:responses ][key ].length)
  @data [:responses ][key ][random_index ].gsub(/\[name \]/,@name)
end

#Performs preprocessing tasks upon all input to the bot
def preprocess(input)
  perform_substitutions(input)
end

#Substitutes words and phrases on supplied input as dictated by
#the bot's :presubs data
def perform_substitutions(input)
  @data [:presubs ].each {|s|input.gsub!(s [0 ],s [1 ])}
  input
end

#Using the single word keys from :responses,we search for the
#sentence that uses the most of them,as it's likely to be the
#best sentence to parse
def best_sentence(input)
  hot_words =@data [:responses ].keys.select do |k|
    k.class ==String &&k =~/^\\w+$/
  end

  WordPlay.best_sentence(input.sentences,hot_words)
end

#Using a supplied sentence,go through the bot's :responses
#data set and collect together all phrases that could be
#used as responses
def possible_responses(sentence)
  responses =[]

  #Find all patterns to try to match against
  @data [:responses ].keys.each do |pattern|
    next unless pattern.is_a?(String)

    #For each pattern,see if the supplied sentence contains
    #a match.Remove substitution symbols (*)before checking.
    #Push all responses to the responses array.
    if sentence.match('\\b'+pattern.gsub(/\*/,')+\\b')
      #If the pattern contains substitution placeholders,
      #perform the substitutions
      if pattern.include?('*')
        responses <<@data [:responses ][pattern ].collect do |phrase|

```

```

    #First,erase everything before the placeholder
    #leaving everything after it
    matching_section =sentence.sub(/^.*#{pattern}\s+/, '')

    #Then substitute the text after the placeholder,with
    #the pronouns switched
    phrase.sub('*',WordPlay.switch_pronouns(matching_section))
  end
else
  #No placeholders?Just add the phrases to the array
  responses <<@data [:responses ][pattern ]
end
end
end

#If there were no matches,add the default ones
responses <<@data [:responses ][:default ] if responses.empty?

#Flatten the blocks of responses to a flat array
responses.flatten
end

end

```

12.4.2 basic_client.rb文件

这个简单的客户端接受用户键盘输入，并将机器人小程序的回应打印输出到屏幕。这可能是最简单的客户端。

```

require 'bot'

bot =Bot.new(:name =>ARGV [0 ],:data_file =>ARGV [1 ])

puts bot.greeting

while input =$stdin.gets and input.chomp !='end'
  puts '>>'+bot.response_to(input)
end

puts bot.farewell

```

客户端使用方法如下：

```
ruby basic_client.rb <bot name> <data file>
```

注 在12.5节中，可以找到简单的Web客户端、机器人小程序互为客户端，以及文本文件客户端的代码清单。

12.5 扩展机器人小程序的功能

把机器人小程序的各个功能分别有条理地放在自己的类中，并使用多个互操作的方法，这样做的好处之一是可以很容易地修改和增加功能。本节我们将介绍怎样轻松扩展机器人小程序的基本功能，以便处理键盘之外的其他输入来源。

在创建核心Bot类时，我们看过一个示例客户端程序，它使用键盘作为输入来源，把输入内容传递给机器人小程序，并打印输出回应内容。这个简单结构展示了怎样抽象分离程序的各个部分，形成松耦合的类，让应用程序更容易修改和扩展。可以用这种松耦合方法，创建使用其他形式输入的客户端。

注 在设计大型应用程序时，要牢记松耦合的用处，它把各部分代码分离开来，如果将来需求有变化，则无须重写大量代码即可达到目的。

12.5.1 用文本文件作为会话来源

你可以在文本文件中创建完全单向的对话，并将其传递给机器人小程序，检验不同的机器人小程序对同一段对话的回应。我们来看下面的例子：

```
require 'bot'

bot = Bot.new(:name => ARGV [0 ], :data_file => ARGV [1 ])
user_lines = File.readlines(ARGV [2 ], 'r')

puts "#{bot.name}says:" + bot.greeting

user_lines.each do |line|
  puts "You say:" + line
  puts "#{bot.name}says:" + bot.response_to(line)
end
```

这段程序接受机器人小程序的名字、数据文件名和对话文件名作为命令行参数，并把用户端对话内容读取到数组中，循环遍历该数据，把每行内容轮流传递给机器人小程序。

12.5.2 把机器人小程序连接到万维网

许多应用程序都把自身挂接到万维网，这样任何人都可以使用它。运用第10章介绍的WEBrick程序库，这是个相当简单的过程。

```
require 'webrick'
require 'bot'

#Class that responds to HTTP/Web requests and interacts with the bot
class BotServlet < WEBrick::HTTPServlet::AbstractServlet
  #A basic HTML template consisting of a basic page with a form
  #and text entry box for the user to converse with our bot. It uses
  #some placeholder text (%RESPONSE%) so the bot's responses can be
```



```

#substituted in easily later.
@@html =%q{
<html><body>
  <form method="get">
    <h1>Talk To A Bot</h1>
    %RESPONSE%
    <p>
      <b>You say:</b><input type="text" name="line" size="40"/>
      <input type="submit"/>
    </p>
  </form>
</body></html>
}

def do_GET(request,response)
  #Mark the request as successful and set MIME type to support HTML
  response.status =200
  response.content_type = "text/html"

  #If the user supplies some text,respond to it
  if request.query ['line'] &&request.query ['line'].length >1
    bot_text =$bot.response_to(request.query ['line'].chomp)
  else
    bot_text =$bot.greeting
  end

  #Format the text and substitute into the HTML template
  bot_text =%Q{<p><b>I say:</b>#{bot_text}</p>}
  response.body =@@html.sub(/\%RESPONSE \%/ ,bot_text)
  end
end

#Create an HTTP server on port 1234 of the local machine
#accessible via http://localhost:1234/or http://127.0.0.1:1234/
server =WEBrick::HTTPServer.new(:Port =>1234 )
$bot =Bot.new(:name =>"Fred",:data_file =>"fred.bot")
server.mount "/",BotServlet
trap("INT"){server.shutdown }
server.start

```

运行这段脚本之后，可用Web浏览器访问<http://127.0.0.1:1234/>或<http://localhost:1234/>，与机器人小程序交谈。交谈示例如图12-4所示。

另外，也可以创建CGI脚本（名为bot.cgi或类似的名字），可用在任何Web主机服务中，只要它支持Ruby语言的脚本即可：

```

#!/usr/bin/env ruby

require 'bot'
require 'cgi'

```

```

#A basic HTML template creating a basic page with a forum and text
#entry box for the user to converse with our bot.It uses some
#placeholder text (%RESPONSE%)so the bot's responses can be
#substituted in easily later
html =%q{
  <html><body>
    <form method="get">
      <h1>Talk To A Bot</h1>
      %RESPONSE%
      <p>
        <b>You say:</b><input type="text" name="line" size="40" />
        <input type="submit" />
      </p>
    </form>
  </body></html>
}

#Set up the CGI environment and make the parameters easy to access
cgi =CGI.new
params =cgi.params
line =params ['line'] &&params ['line'].first

bot =Bot.new(:name =>"Fred",:data_file =>"fred.bot")

#If the user supplies some text,respond to it
if line &&line.length >1
  bot_text =bot.response_to(line.chomp)
else
  bot_text =bot.greeting
end

#Format the text and substitute into the HTML template
#as well as sending the MIME header for HTML support
bot_text =%Q{<p><b>I say:</b>#{bot_text}</p>}
puts "Content-type:text/html \n \n"
puts html.sub(/\%RESPONSE \%/ ,bot_text)

```

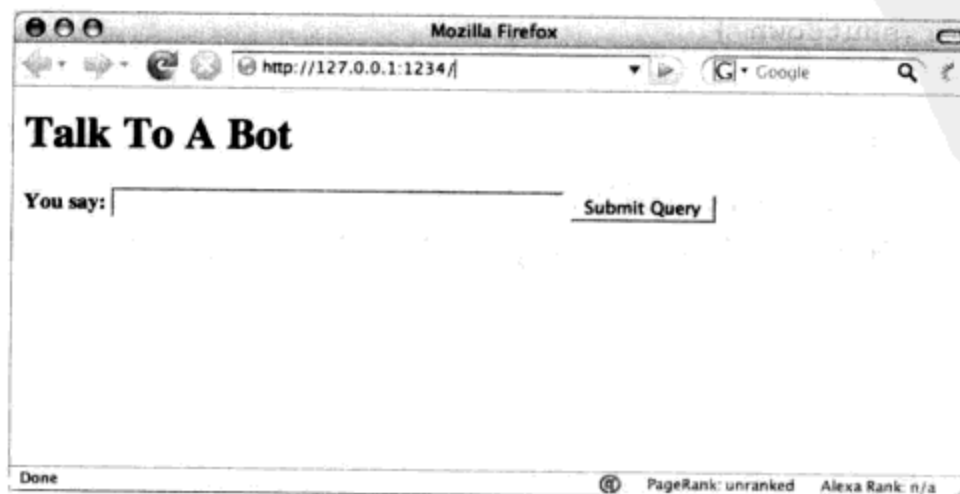


图12-4 通过Web浏览器访问机器人小程序的Web客户端

注 你还需要确认已上传bot.rb、wordplay.rb和机器人小程序数据文件。

12.5.3 机器人小程序之间的会话

可以让人与机器人小程序进行交互，也可以让机器人小程序互相交互！因为机器人小程序收集回应只需一个方法，因此可以用管道在两个机器人小程序之间来回传递回应，只须以下几行代码：

```
require 'bot'

fred = Bot.new(:name => 'Fred', :data_file => 'fred.bot')
chris = Bot.new(:name => 'Chris', :data_file => 'fred.bot')
r = fred.greeting
10.times do
  puts "#{fred.name}said:" + r
  r = chris.response_to(r)
  puts "#{chris.name}said:" + r
  r = fred.response_to(r)
end
```

其结果可以如下面对话所示（由于数据文件中有多个可选回应，因此结果可能随机变化）：

```
Fred said:Hi.What would you like to talk about?
Chris said:Why?
Fred said:What?
Chris said:I don't know.Do you?
Fred said:What?
Chris said:Why do you ask?
Fred said:I don't understand.
Chris said:Tell me about something else.
Fred said:Tell me about something else.
Chris said:Tell me about something else.
Fred said:I'm tired of this.Change the subject.
Chris said:What?
Fred said:Why?
Chris said:Tell me about something else.
Fred said:I don't understand.
Chris said:What?
Fred said:Why do you ask?
Chris said:What?
Fred said:Why?
Chris said:Huh?
```

这不是我们看过的最棒对话，但看着两个假治疗师互搞乌龙确实有点意思。当然，如果你能开发出两个能进行真正引人入胜对话的机器人小程序，你将成为人工智能领域的明星人物！

机器人小程序数据的关键问题，是默认数据不包含任何可从其他短语中摘取的关键字，因此两个机器人小程序被锁在一个循环中，互相抛出默认短语。如果你想让机器人小程序表现得

令人惊叹，就必须相当重视扩展基本数据集。

12.6 小结

本章我们介绍了开发简单的交谈机器人小程序，并同时开发了一个程序库，为程序库生成了测试代码，编写代码把机器人小程序的词汇保存到外部文件，并考察了用数据库或网站对项目进行扩展的几种方法。

本章标志着本书第二篇的结束，现在你应该有足够的Ruby知识，可以成为中等Ruby开发人员。你可以理解大部分在线Ruby文档，也可以用Ruby进行专业化开发，或为自己的兴趣进行开发。

本书第三篇深入介绍Ruby的程序库和框架，范围从Ruby on Rails到Web，再到一般网络连接程序库的使用。第16章介绍各种各样的Ruby程序库及其用法，对你开发自己的程序特别有用，因为你不必经常重新发明轮子！

第三篇 Ruby在线

本篇介绍Ruby的因特网和网络连接功能。本篇涵盖的知识并非开发一般Ruby程序所必须的，但随着因特网和Web快速成为现代软件开发的重要范围，你会觉得这些章节很有用。本篇以一个参考资料章节结束，该章节介绍了大量Ruby程序库及其功能。

第13章 Ruby on Rails: Ruby的杀手级应用

本章将介绍Ruby on Rails框架，这是一种高级Web应用开发框架。我们将详细讲解简单Rails应用的开发和基于数据库的运行，然后再介绍一些更高级的Web开发主题。

尽管本书是本Ruby书，而不是Rails书（Ruby on Rails简称Rails），但Rails已经变成Ruby世界如此重要的组成部分，甚至Ruby初学者也要关注这一主题。不过，Apress出版公司确实有一系列专门讲解Ruby on Rails和Web开发的书籍，如果你希望进一步了解这方面的开发，可以选购阅读。

13.1 第一步

在开始用Rails开发Web应用程序之前，必须首先知道它是什么，为什么用，以及怎样让它运行，因为与其他Ruby程序库相比，它的安装过程有更多细节内容。

13.1.1 Rails是什么，为什么要用它

Ruby on Rails是一个开源的Web应用开发框架。它让Web应用开发变得相当简单。关于Rails背后的非技术历史，包括其开发动机，请参阅第5章。

Rails的目标是可以用轻松直接的方式，并用尽量少的代码，来开发Web应用。默认情况下，Rails有许多假定和默认配置，对大多数Web应用都有效。当然，覆盖默认配置也很容易，但这些默认配置的目的，就是让应用开发从一开始就简单。

Rails遵循模型—视图—控制器（Model-View-Controller, MVC）架构模式。这表示Rails应用主要分为三个部分：模型、视图和控制器。在Rails中，这些组件有下列角色：

- 模型：用于表示应用的数据，其中包含操作和检索数据的逻辑。在Rails中，模型以类的方式表示。可以把模型想成控制器与数据之间的抽象而理想化的接口。
- 视图：是Web应用用户可见的模板和HTML代码。视图将数据转换成用户可视的格式，可以用HTML（用于Web浏览器）、XML、RSS和其他格式输出数据。
- 控制器：控制器结构是把模型、数据和视图绑定在一起的代码逻辑。它能处理输入数据，

并发送输出数据。控制器是调用模型的方法，并将其提交给视图。控制器包含名为action的方法，一般情况下，action表示与该控制器相关的动作，例如“显示”、“隐藏”、“查看”、“删除”等。

各个组件之间的基本关系如图13-1所示。

注 如要了解更多MVC范式内容，请访问<http://en.wikipedia.org/wiki/Model-view-controller>。

使用Rails的最常见动机，是它去掉了用其他技术开发Web应用所需的大量基础工作。例如数据库访问、动态页面元素（用Ajax——Asynchronous JavaScript and XML（异步JavaScript和XML）的缩写）、模板生成和数据校验等功能，要么已经预先配置好，要么只需几行代码即可配置。

Rails还鼓励养成良好的开发习惯。所有Rails应用程序都自带支持单元测试（以及其他形式的测试），Rails的指导原则是“绝不重复（Don't Repeat Yourself, DRY）”和“惯例优于配置（Convention Over Configuration）”。

13.1.2 安装Rails

Rails框架由几个不同程序库组成，但安装很简单，因为所有组成部分都以RubyGems包形式提供。为了完整起见，下面列出Rails各组成部分的程序库清单：

- Rails：是Ruby on Rails框架的核心程序库，用来把其他程序库连接在一起。
- ActionMailer：该程序库让Rails应用可以很容易发送电子邮件。关于ActionMailer的简要介绍，以及怎样用它在Rails之外发送电子邮件，请参阅第14章。
- ActionPack：该程序库提供了有用的方法，用于视图和控制器生成HTML和动态页面元素（例如Ajax和JavaScript），或管理数据对象。
- ActionWebService：该程序库提供的方法可以轻松地把Rails应用的功能，作为Web服务的形式发布。Rails这部分功能在Rails 1.2之后删除了，因为改用了另一种技术，但这里为了完整起见仍然提一下。
- ActiveRecord：是对象—关系映射工具（object-relational mapper, ORM），把数据库表和类连接起来。如果你有个ActiveRecord对象，指向数据库表的某一行，则可以像其他Ruby对象一样操作该对象（使用其属性和方法），修改将会保存到相应的数据库表中。关于ActiveRecord的简要介绍，请参阅第9章。
- ActiveSupport：该程序库收集了许多支持和工具类，用于各个Rails功能。例如，

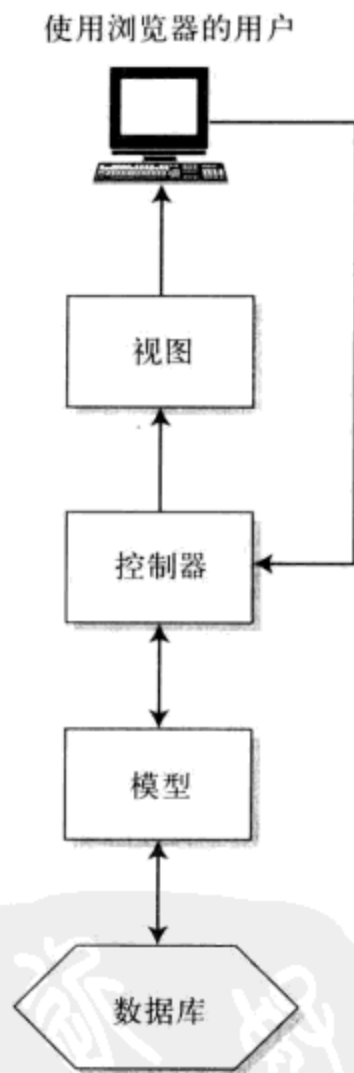


图13-1 用户、视图、控制器和模型之间的交互

ActiveSupport实现了许多有用的方法，来操作时间、数字、数组和散列表。

一般来说，无须知道或关心其中的每个组成元素具体是哪个程序库，因为你可以用RubyGems一次安装全部内容，如下所示：

```
gem install rails
```

gem会询问你是否安装每个程序库（可以跳过询问直接安装，方法是改用`gem install --include-dependencies rails`命令），然后gem即安装每个程序库及其文档。

如果你此时对安装Ruby程序库很自信，那么这些命令可以足够完成基本的Rails框架安装。不过在Windows和Mac OS X中还有更简单的安装方法，可以简化这一过程。

Windows用户可以安装Instant Rails (<http://instantrails.rubyforge.org/>)，这是一站式Rails安装方案，其中包含Ruby、Rails、Apache和MySQL，所有组件都预先配置好，可以“开箱即用”。这些组成元素与你常规安装的版本互不相干，因此你可以立即开始Rails开发。如果你在正常安装Rails和/或MySQL时遇到困难，该系统可以帮你解决。

Mac用户可以安装Locomotive (<http://locomotive.raaum.org/>)，它提供了与Instant Rails相同功能，但用于OS X平台。Locomotive将在Mac上运行Rails，不会破坏任何现有配置或已安装的工具。

13.1.3 数据库方面的考虑

由于Rails主要用于开发数据驱动的Web应用，因此你的计算机上有必要拥有一套数据库系统。Instant Rails和Locomotive用户将自动安装MySQL，但如果没有安装数据库系统，则需要获取一套。

数据库引擎在第9章介绍，你可以用其中任何一种（MySQL、SQLite、PostgreSQL、Oracle和微软SQL Server）在Ruby on Rails上运行。不过，大多数开发人员都用MySQL或PostgreSQL，因为Rails对这两种数据库引擎的支持最好。在使用其他数据库引擎时，常常需要做些“破解”并修改Rails程序才能让一切正常运行，这些内容则超出了本章的范围。

本章假定你本机安装有MySQL服务器，可供Rails应用使用。如果没有，可以访问<http://dev.mysql.com/downloads/mysql/5.0.html>，免费下载并安装MySQL的“社区”版。

13.2 构建Rails简单应用

如上节所述，Rails因其可以轻松开发Web应用而流行。在本节中，我将展示这一特点，演示怎样生成简单的Web应用，并介绍底层的工作原理。

13.2.1 创建Rails空白应用

由于Rails可以用来开发小型或大型应用，因此相应不同类型的文件都组织存放在不同的目录中，以便隔离各组成元素，保持大型项目的整洁。在新建的空白Rails项目中，也包含许多预先创建的文件。浏览这些文件和总体目录结构的最快方法，是立即动手创建一个Rails新项目。

在本章中，你的项目是创建一个过于简单的在线日记系统，类似于博客（或称网络博客）。

系统开发完成后，可以让你查看日记，并增加、删除或编辑指定日记条目。创建、读取、更新和删除（Create、Read、Update、Delete）条目的基本功能，又称为CRUD功能，而大多数类型的Web应用都提供某种层次的CRUD机制。例如，照片画廊网站允许你增加、查看、编辑和删除照片，这就是全套CRUD操作。因此，开发简单日记工具的技巧，可以很容易地转变为开发大多数其他类型Web应用的技巧。

rails命令行工具

当Rails安装完毕后，也安装了一个名为rails的脚本。可以用rails脚本创建新Rails新项目，及其默认文件，以及目录结构。要使用该命令，请在文件系统中进入要保存Rails项目的地方（可能需要创建一个rails目录），运行rails命令，并把应用的名字作为唯一参数：

```
rails mydiary
```

```
create
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create components
create db
create doc
create lib
create lib/tasks
create log
create public/images
create public/javascripts
create public/stylesheets
create script/performance
create script/process
create test/fixtures
create test/functional
create test/integration
create test/mocks/development
create test/mocks/test
create test/unit
create vendor
create vendor/plugins
create tmp/sessions
create tmp/sockets
create tmp/cache
create Rakefile
create README
create app/controllers/application.rb
create app/helpers/application_helper.rb
create test/test_helper.rb
create config/database.yml
```



```
create config/routes.rb
create public/.htaccess
create config/boot.rb
create config/environment.rb
create config/environments/production.rb
create config/environments/development.rb
create config/environments/test.rb
create script/about
create script/breakpointer
create script/console
create script/destroy
create script/generate
create script/performance/benchmark
create script/performance/profiler
create script/process/reaper
create script/process/spawner
create script/runner
create script/server
create script/plugin
create public/dispatch.rb
create public/dispatch.cgi
create public/dispatch.fcgi
create public/404.html
create public/500.html
create public/index.html
create public/favicon.ico
create public/robots.txt
create public/images/rails.png
create public/javascripts/prototype.js
create public/javascripts/effects.js
create public/javascripts/dragdrop.js
create public/javascripts/controls.js
create public/javascripts/application.js
create doc/README_FOR_APP
create log/server.log
create log/production.log
create log/development.log
create log/test.log
```

此时生成了许多文件和目录，但不要被淹没的感觉！在本章中，无须使用其中大数文件，因为它们都是为更高级用途而创建的，与简单Web应用开发无关。在大多数情况下，Rails提供了明智的默认配置，因此你无须修改其中的许多文件，除非你要做十分特别的事。不过，将在下一节查看这些目录是做什么用的。

注 在你机器上创建的具体文件和目录结构，可能因运行的Rails版本而异。上面的结构由Rails 1.1.6生成，这是2007年初的Rails官方产品化版本。

Rails应用中的文件和目录

本节我们将逐一介绍rails命令创建的目录和文件，并了解它们的用途。请不要对本节内容感觉喘不过气，如果你不理解某些内容，请继续往下读，因为这里提到的大多数新名词和概念，将在本章对其使用过程中解释。

rails命令生成以下主要文件夹：

- **app**: 该文件夹包含大多数与应用直接相关的Ruby源代码和输出的模板。它包含几个其他子文件夹，将在下面介绍。
- **app/controllers**: 包含控制器文件。在空项目中，其中只有`application.rb`文件。`application.rb`是应用级的控制器，可在此定义其他所有控制器都能继承的方法。
- **app/helpers**: 包含辅助文件——这是Ruby源代码文件，供视图使用。
- **app/models**: 包含应用中的模型文件，一个模型对应一个文件。在空项目中，还没有定义模型，因此该目录是空的。
- **app/view**: 包含应用的输出模板（视图）。一般每个控制器都在`app/view`之下有其自己的文件夹，其中有模板文件。还有一个`layout`文件夹，Rails用它来保存通用的应用级模板。
- **components**: 包含具体的MVC“组件”应用。该功能不再常用，主要为历史原因而保留（实际上，该文件夹在Rails项目中甚至可能不存在，因所用Rails的版本而定）。本章末尾介绍的插件，已经很大程度上取代了组件。
- **config**: 这是个重要的文件夹，其中包含应用的配置文件。`database.yml`是个YAML文件，其中是关于应用所需数据库的信息。`environment.rb`和`boot.rb`是预制文件，通常不需要修改，除非你对应用启动的细节非常了解。`routes.rb`在本章后文“路由”一节介绍。
- **db**: 该文件夹用于保存数据库映像、备份和迁移文件。
- **doc**: 包含为应用生成的RDoc格式文档。在空项目中，这里包含一个简单的文本文件，名为`README_FOR_APP`，可用作纯文本文档文件，也可包含怎样安装应用的说明。
- **lib**: 包含第三程序库和Rake任务文件。在大多数Rails应用开发中，无须用到该目录。插件已经很大程度上取代了lib中程序库的功能。
- **log**: 包含关于应用运行的日志文件。
- **public**: 包含非动态文件，可在应用的URL模式下访问。例如，可包含JavaScript程序库、图像和CSS样式表。该文件夹还包括几个“分发”文件和一个`.htaccess`文件，可以设置你的应用在Apache和LightTPD等Web服务器下如何运行。
- **script**: 包含脚本和命令行工具，可用于构造和部署Rails应用。`console`是个类似`irb`的工具，可以预先载入Rails应用的环境，并提供命令提示符。`generate`是个脚本，可帮你从模板中生成某种类型的Rails代码。`server`用于运行基本的WEBrick或LightTPD服务器，以便从Web浏览器访问应用。其他脚本目前对你来说还用不到。
- **test**: 包含Rails应用的测试子系统，本章后文“测试”一节将介绍关于该文件夹的更多内容。
- **tmp**: 临时保存数据的地方，供Rails应用创建并保存数据。
- **vender**: 该文件夹用于保存你应用所绑定的Rails框架的版本信息，以及保存插件（在`vender/plugins`目录）。

在本章后续内容中，为了开发简单应用而在文件夹中创建文件，我将再次简要提到这里的许多文件夹。

13.2.2 数据库初始化

前文提到Rails应用一般都是依赖于数据库的。鉴于此，必须在数据库服务器为应用创建数据库。

为应用开发而创建数据库的方法，根据数据库类型和数据库服务器的安装情况而定。在本节中，我将假设你安装了MySQL服务器，并下载了管理MySQL数据库的工具，或可以用标准MySQL命令行客户端。

注 如果你使用了其他类型的数据库，必须参考相关文档和数据库系统关联的程序，找出创建数据库和从你的应用中登录到数据库的方法。

从MySQL命令行客户端可以快速简单地创建数据库。下面的例子展示了怎样创建数据库，并将其与某个用户名和密码关联起来，以备日后访问：

```
~/rails/mydiary $mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 10 to server version: 5.0.27-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>CREATE DATABASE mydiary;
Query OK, 1 row affected (0.08 sec)

mysql>GRANT ALL PRIVILEGES ON mydiary.* TO mydiary@localhost
IDENTIFIED BY 'mypassword';
Query OK, 0 rows affected (0.30 sec)

mysql>QUIT
Bye
```

在本例中，创建了名为mydiary的数据库，然后把全部权限授予名为mydiary的用户，其密码为mypassword。可以通过MySQL命令行客户端，用以下命令访问mydiary数据库，检查数据库和用户是否已成功创建。

```
~/rails/mydiary $mysql -u mydiary -p
Enter password:<type mypassword at this point>
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 12 to server version: 5.0.27-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>QUIT
Bye
```

如果MySQL不提示任何错误，一般情况下表示一切正常。如果看到出错提示说非法用户名，则上面授权操作没有成功，因此请检查那一步操作的出错信息。

如果你使用MySQL的GUI客户端，或使用完全不同的数据库系统的客户端，请确认已经创建了名为mydiary的数据库，并让它对用户开放。客户端应该可以对此进行简单的测试。

数据库创建好之后，可以编辑config/database.yml文件，告诉Rails应用这个数据库已存在。下面是该文件在空项目情况下的默认内容：

```
#MySQL (default setup).Versions 4.1 and 5.0 are recommended.
#
#Install the MySQL driver:
#gem install mysql
#On MacOS X:
#  gem install mysql -- --include=/usr/local/lib
#On Windows:
#  There is no gem for Windows.Install mysql.so from RubyForApache.
#  http://rubyforge.org/projects/rubyforapache
#
#And be sure to use new-style password hashing:
#  http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
  adapter:mysql
  database:mydiary_development
  username:root
  password:
  host:localhost

#Warning:The database defined as 'test'will be erased and
#regenerated from your development database when you run 'rake'.
#Do not set this db to the same as development or production.
test:
  adapter:mysql
  database:mydiary_test
  username:root
  password:
  host:localhost

production:
  adapter:mysql
  database:mydiary_production
  username:root
  password:
  host:localhost
```

忽略注释，你会注意到database.yml中有三个主要段落，分别是“development”、“test”和“production”。这表示应用有三种不同的运行环境。例如，在开发时，你希望应用返回充分的出错信息，并自动检测对代码的修改。在投运时（更好的想法是在“部署”环境），想要速度、缓存和简要的出错信息。而测试环境是用另外的数据库进行测试，不影响正常数据。

目前要关心的是“development”段落。我们需要修改本段落的详细内容，以反映怎样连接到刚刚创建的数据库。例如：

```
development:
  adapter:mysql
  database:mydiary
  username:mydiary
  password:mypassword
  host:localhost
```

只作此修改并保存该文件，不要修改其他任何内容。

13.2.3 创建模型和迁移文件

在进行应用开发之前的最后一个关键步骤，是生成数据库表，以表示应用要操作的模型。在本例中，我们开始时尽量简单化，完全关注日记条目（entries），因此把这个数据库表称为entries。

在Rails中，模型和数据库表一般有直接关系。如果有个名为entries的数据表，则它会直接关联到Rails应用中名为Entry的模型类。

注 默认情况下，表名是复数形式，而模型名是单数。Rails能对单数和复数名字进行自动转换。不过，有时也可能要手工强制表名适应不同的情况（例如，表名为diary_entries，而模型类名为Entry），但这超出了本章的范围。

此时可以用两种方法，在mydiary数据库中创建entries表。

第一种选择是通过MySQL命令行客户端（或你使用的其他客户端），用SQL语句手工生成表，如下所示：

```
CREATE TABLE entries (
  id int auto_increment,
  title varchar(255),
  content text,
  created_at datetime,
  PRIMARY KEY(id)
);
```

这段SQL语句创建了entries表，其中包含四列：id列、用来保存日记条目标题的title列、content列和用来保存条目创建时间的created_at列。

用SQL或GUI客户端创建表的方法很有效，但还有个更好的选择，是用Rails提供的迁移（migrations）系统。迁移提供了管理数据库模式和数据的程序化方式，它让你可以管理数据库模式的版本变迁，可以“回滚”数据库模式，或在新数据库进行一次全部创建。

注 关于迁移的全面信息，请访问<http://www.rubyonrails.org/api/classes/ActiveRecord/Migration.html>。

我们用script 文件夹中的generate脚本，创建迁移文件来构建entries表：

```
ruby script/generate migration AddEntriesTable
```

```
create db/migrate
create db/migrate/001_add_entries_table.rb
```

注 在OS X或Linux中，可以用更短的格式来使用script文件夹中的脚本，例如./script/generate。不过上面的格式显式指定了Ruby解释器，因此可在所有平台使用。

generate脚本为你创建了db/migrate文件夹（这是所有迁移文件保存的地方），并创建了名为001_add_entries_table.rb的Ruby文件，在该文件中可以定义你想要数据库做什么。

注 尽管为该迁移文件用了AddEntriesTable的名字，但它的名字可以是任何内容。此处用这一名字只是因为它是迁移命名的常用风格。

我们来看一下001_add_entries_table.rb文件的内容：

```
class AddEntriesTable < ActiveRecord::Migration
  def self.up
    end

    def self.down
    end
end
```

这是个空的迁移文件，其中有两个方法：up和down。up用来创建东西并执行迁移所需操作，down用来“回滚”状态，从迁移完成后的状态，回到迁移运行之前的状态。

在up方法中使用Rails提供的一些方法来生成entries表，在down方法中使用删除表的方法。最终001_add_entries_table.rb如下所示：

```
class AddEntriesTable < ActiveRecord::Migration
  def self.up
    create_table :entries do |table|
      table.column :title, :string
      table.column :content, :text
      table.column :created_at, :datetime
    end
  end

  def self.down
    drop_table :entries
  end
end
```

在类方法up中使用了create_table方法来创建entries表，在代码块中用column方法来创建不同类型的列。

注 你无须显式创建id列，因为已经自动创建了。

要执行迁移（从而实际创建迁移文件中的entries表），需要用到名为db:migrate 的 Rake任务（关于它的更多信息请参阅下面的“Rake任务”）：

```
rake db:migrate
```

```
(in /Users/peter/rails/mydiary)
==AddEntriesTable:migrating =====
--create_table(:entries)
  ->0.3683s
==AddEntriesTable:migrated (0.3685s)=====
```

迁移中的第一步都显示在上述结果中。AddEntriesTable迁移被运行，从而创建了entries表。这里没有显示错误信息，则mydiary数据库现在含有正常可用的entries表。

RAKE任务

Rake任务是与应用相关的管理维护任务，由Rake工具进行管理。Rake表示“Ruby的Make”，是用来对Ruby项目和代码进行处理和激发动作的工具，通常用在Rails项目中，完成诸如单元测试、执行迁移等事务。

要执行Rake任务，只须运行rake命令后跟任务名：

```
rake <task name>
```

你也可以获取可用Rake任务的列表清单：

```
rake --tasks
```

在Rails 1.1.6中有41个默认任务。为节省篇幅，不在这里列出，但值得一看，可以让你获得有哪些任务的总体印象。

一般更倾向于使用迁移，而不是对数据库直接执行SQL语句，因为迁移大多为数据库无关的（Rails根据迁移文件中的方法，将针对所用数据库引擎输出正确的SQL语句），而且数据库操作被长久保存在代码中，而不是在短暂的SQL语句操作中。

迁移也让修改数据库更容易。例如，如果想为增加entries表增加一列，只须创建一个新的迁移，并在新迁移文件的up和down方法中，分别使用add_column和remove_column方法即可。例如：

```
ruby script/generate migration AddUpdatedAtColumnToEntries
```

```
exists db/migrate
```

```
create db/migrate/002_add_updated_at_column_to_entries.rb
```

然后可以编写002_add_updated_at_column_to_entries.rb，如下所示：

```
class AddUpdatedAtColumnToEntries < ActiveRecord::Migration
  def self.up
    add_column :entries, :updated_at, :datetime
  end
  def self.down
    remove_column :entries, :updated_at
  end
end
```



```
end
end
```

然后实际执行迁移:

```
rake db:migrate
```

```
(in /Users/peter/rails/mydiary)
==AddUpdatedAtColumnToEntries:migrating =====
--add_column(:entries,:updated_at,:datetime)
->0.2381s
==AddUpdatedAtColumnToEntries:migrated (0.2383s)=====
```

新迁移向entries表增加了updated_at DATETIME列。

如果你需要,可以“回滚”迁移,回到第一次迁移的状态:

```
rake db:migrate VERSION=1
```

```
(in /Users/peter/rails/mydiary)
==AddUpdatedAtColumnToEntries:reverting =====
--remove_column(:entries,:updated_at)
->0.0535s
==AddUpdatedAtColumnToEntries:reverted (0.0536s)=====
```

注 在某些情况下,迁移文件自动创建,你只需填写内容即可。例如,如果用Rails模型生成器创建模型时,将会自动创建迁移文件,以创建与新建模型相关的表。但在本章中,我们采用相反的方向。这正体现了Ruby的核心原则:总有不只一种解决方法!

13.2.4 搭建脚手架

在上面几节中,创建了数据库和迁移,生成了正常可用的entries表。在本节中,将创建足够的代码,来建立可以基于entries表执行CRUD操作的简单Web应用。

Rails提供了名为脚手架(scaffolding)的机制,可以生成默认通用代码,为任何模型提供CRUD操作。然后可以在此基础上构建自己的视图和控制器。脚手架的目的是快速启动,无须从零开始编码(不过如果你愿意,也可以这么做,特别是当你雄心勃勃地想创建与脚手架完全不同的功能时)。

要为entries表生成脚手架,请再次使用generate脚本:

```
ruby script/generate scaffold Entry
```

```
exists app/controllers/
exists app/helpers/
create app/views/entries
exists test/functional/
dependency model
exists app/models/
exists test/unit/
```

```
exists test/fixtures/  
create app/models/entry.rb  
create test/unit/entry_test.rb  
create test/fixtures/entries.yml  
create app/views/entries/_form.rhtml  
create app/views/entries/list.rhtml  
create app/views/entries/show.rhtml  
create app/views/entries/new.rhtml  
create app/views/entries/edit.rhtml  
create app/controllers/entries_controller.rb  
create test/functional/entries_controller_test.rb  
create app/helpers/entries_helper.rb  
create app/views/layouts/entries.rhtml  
create public/stylesheets/scaffold.css
```

在创建脚手架时, Rails查看与模型关联的、要建立脚手架的数据库表(本例是与Entry模型关联的entries表), 并生成反映该表结构的控制器、视图和模型文件。生成的文件显示在上面的结果列表中。

注 脚手架是根据数据库表的结构创建的, 因此在创建脚手架之前, 永远必须先创建并运行迁移文件。

脚手架生成器还创建了布局文件、几个测试相关文件和用于脚手架布局的样式表文件。另外如果所需目录不存在, 还生成相关目录。

你只须这么做, 就得到了可以运行的应用! 如要试运行, 需要运行server脚本, 它提供了基本的WEBrick Web服务器, 我们通过它来访问应用程序:

```
ruby script/server
```

```
=>Booting WEBrick...  
=>Rails application started on http://0.0.0.0:3000  
=>Ctrl-C to shutdown server;call with --help for options  
[2007-03-19 19:37:48 ] INFO WEBrick 1.3.1  
[2007-03-19 19:37:48 ] INFO ruby 1.8.5 ((2006-08-32)[i686-darwin8.8.1 ]  
[2007-03-19 19:37:48 ] INFO WEBrick::HTTPServer#start:pid=10999 port=3000
```

此时应用程序运行起来, 但什么也不做。这是因为它正在等待来自Web浏览器的请求。

注 如果安装了Mongrel程序库, 可以用它来运行Rails应用, 只须运行mongrel_rails start命令, 而不是ruby script/server命令。

使用Web浏览器, 用WEBrick输出的URL(本例是http://0.0.0.0:3000/, 在你的机器上也可以用http://localhost:3000/或http://127.0.0.1:3000/), 即可访问本应用。你应该看到如图13-2所示的页面。

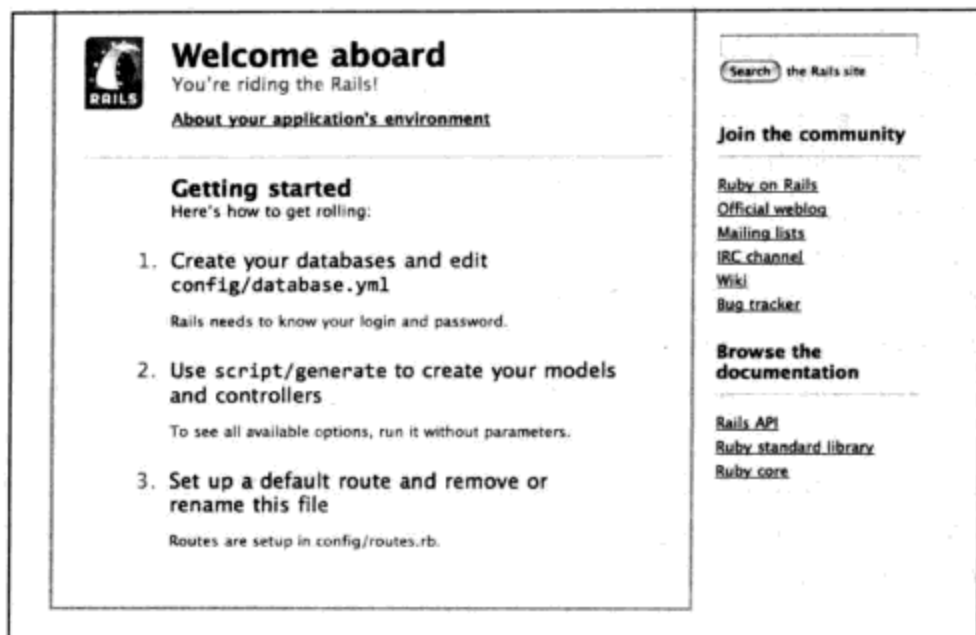


图13-2 Rails应用的默认index.html页面

你看到的页面是public文件夹中的index.html文件。这是因为如果在浏览器载入的URL中，没有关联任何Rails应用的action，则Rails应用返回public文件夹中的文件（如果有匹配文件的话），或返回出错信息。由于Web服务器默认页面通常是index.html，因此返回public/index.html。

当你为Entry模型生成了脚手架之后，也创建了名为entries的控制器。默认情况下，用http://<hostname>/controller/action的URL格式来访问Rails应用中的控制器方法。

因此，对于该应用来说，应载入http://localhost/entries（localhost可换成你本机所用的其他主机名）。无须指定action的名称，但默认情况下假定使用index的action名，而脚手架已实现了这些action。如果载入成功，可看到条目的简单列表，如图13-3所示。

图13-3所示的条目列表是一片醒目的空白，这是因为entries表中还没有数据。但列标题很显眼（Title、Content和Created at），并有一个“New entry”（新建条目）可用链接。

点击“New entry”链接，进入http://localhost/entries/new（这是entries控制器中的new方法），并显示页面，其中包含一个表单，可以在此填写条目数据。该视图如图13-4所示。

从此开始，可以创建新条目，返回列表，编辑条目（其表单与图13-4看起来很像）和删除条目。这正好涵盖了所有CRUD功能！

有了脚手架功能，只须在命令行输入一行命令，即可获



图13-3 entries脚手架index视图的简单列表

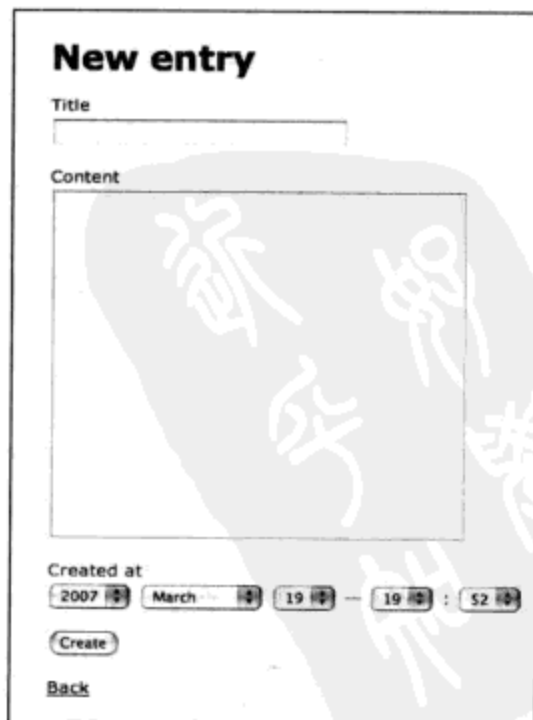


图13-4 entries控制器的new方法，用来创建新条目

得简单但完整的数据驱动Web应用。但下面需要看一下脚手架生成器实际生成的内容，并了解怎样定制模型、控制器和视图，来创建想要的應用。

13.2.5 控制器与视图

在上一节中，你组装了一个简单的Web应用，可以用来创建、修改、列表和删除日记条目。你用到了脚手架，因此无须编写任何代码即可建立完整的可运行应用。在本节中，你将了解脚手架生成了什么内容，它的工作原理，以及用自己的方法和视图来扩展应用的功能。

控制器的action

你用来访问应用的第一个URL是<http://localhost/entries/list>。这个URL调用了`entries`控制器的`list`方法。我们来看一下`app/controllers/entries_controller.rb`文件，找到这个方法：

```
class EntriesController < ApplicationController
  def index
    list
    render :action => 'list'
  end

  #GETs should be safe (see http://www.w3.org/2001/tag/doc/whenToUseGet.html)
  verify :method => :post, :only => [ :destroy, :create, :update ],
        :redirect_to => { :action => :list }

  def list
    @entry_pages, @entries = paginate :entries, :per_page => 10
  end

  def show
    @entry = Entry.find(params[:id])
  end

  def new
    @entry = Entry.new
  end

  def create
    @entry = Entry.new(params[:entry])
    if @entry.save
      flash[:notice] = 'Entry was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end

  def edit
```

```

    @entry = Entry.find(params[:id])
  end

  def update
    @entry = Entry.find(params[:id])
    if @entry.update_attributes(params[:entry])
      flash[:notice] = 'Entry was successfully updated.'
      redirect_to :action => 'show', :id => @entry
    else
      render :action => 'edit'
    end
  end

  def destroy
    Entry.find(params[:id]).destroy
    redirect_to :action => 'list'
  end
end

```

这段代码显示Ruby控制器以类的形式实现，它继承自ApplicationController（如app/controllers/application.rb所示），后者又继承自Rails核心类，ActionController::Base。

当用户访问entries控制器的list方法时，控制被转交给EntriesController类的list方法（或action），如下所示：

```

def list
  @entry_pages, @entries = paginate :entries, :per_page => 10
end

```

这段代码执行的操作很简单，它依赖于Rails提供的分页（paginate）方法，该方法从特定模型（本例是从条目的模型）得到数据记录，并将其按10个（本例是10个）为一组进行分页。分页的目的在于，如果系统包含1 000个数据记录，都显示在一页，这显得笨拙无比，而用paginate方法按10个为一组，并根据Web浏览器（通过URL）传递的页码变量，只显示正确的10个一组的数据记录。

但可以重写list方法，载入所有的条目，如下所示：

```

def list
  @entries = Entry.find(:all)
end

```

Entry是模型类，继承自ActiveRecord::Base，后者提供了适合在模型的关联表中浏览和查找数据的方法。因此，Entry.find(:all)从entries表中（以对象形式）返回所有行，并将它们放在@entries数组中。

视图和嵌入式Ruby代码

上一节讨论了list这个控制器action，现在我们要来看一下它所对应的视图。视图的模板放在app/views/entries/list.rhtml文件中：


```

<h1>Listing entries</h1>

<table>
  <tr>
    <%for column in Entry.content_columns %>
      <th><%=column.human_name %></th>
    <%end %>
  </tr>

  <%for entry in @entries %>
    <tr>
      <%for column in Entry.content_columns %>
        <td><%=h entry.send(column.name)%></td>
      <%end %>
      <td><%=link_to 'Show',:action =>'show',:id =>entry %></td>
      <td><%=link_to 'Edit',:action =>'edit',:id =>entry %></td>
      <td><%=link_to 'Destroy',{:action =>'destroy',:id =>entry },
        :confirm =>'Are you sure?',:post =>true %></td>
    </tr>
  <%end %>
</table>
<%=link_to 'Previous page',{:page =>@entry_pages.current.previous }
  if @entry_pages.current.previous %>
<%=link_to 'Next page',{:page =>@entry_pages.current.next }
  if @entry_pages.current.next %>

<br />

<%=link_to 'New entry',:action =>'new'%>

```

如果你既熟悉Ruby又熟悉HTML，将会注意到这个视图基本上是HTML代码，其中嵌入了Ruby代码（Ruby代码放在<%和%>标签中）。

注 可以嵌入Ruby代码的HTML视图，其文件扩展名为RHTML，而不是HTML。

在上面的视图中，第一个动态段落如下所示：

```

<% for column in Entry.content_columns %>
  <th><%= column.human_name %></th>
<% end %>

```

这段代码和正常Ruby循环一样，for循环迭代遍历Entry.content_columns的结果（这是个ActiveRecord方法，返回对象的每一列）。

在这个应用的示例中，列名是Title、Content和Created at（这是title、content和created_at的人性化版本），因此上述循环是调用每一列的human_name方法，导致生成下面的HTML，并返回给访问的Web浏览器：

```

<th>Title</th>
<th>Content</th>

```

```
<th>Created at</th>
```

列表视图的核心部分包含如下代码：

```
<%for entry in @entries %>
  <tr>
    <%for column in Entry.content_columns %>
      <td><%=h entry.send(column.name)%></td>
    <%end %>
    <td><%=link_to 'Show',:action =>'show',:id =>entry %></td>
    <td><%=link_to 'Edit',:action =>'edit',:id =>entry %></td>
    <td><%=link_to 'Destroy',{:action =>'destroy',:id =>entry },
      :confirm =>'Are you sure?',:post =>true %></td>
  </tr>
<%end %>
```

这段视图代码对页面的主要部分进行渲染：生成实际的条目列表。我不会逐行代码地进行解说，但有几个关键值得一提。整段代码是对@entries中每个元素的循环（这是对for entry in @entries是@entries.each do |entry|的另一种说法），你可以回忆一下，控制器代码从数据库中把Entry对象放到@entries数组，因此这里的视图代码对每个元素（或每个条目）进行循环迭代。接着，在主循环中还有另一个循环，是对这些条目关联的每一列进行循环。

条目数据显示之后，接着碰到下面的代码：

```
<td><%= link_to 'Show', :action => 'show', :id => entry %></td>
<td><%= link_to 'Edit', :action => 'edit', :id => entry %></td>
<td><%= link_to 'Destroy', { :action => 'destroy', :id => entry },
  :confirm => 'Are you sure?', :post => true %></td>
```

这里值得一看的重要内容是对link_to方法的调用。link_to是Rails提供的特殊方法，它生成HTML链接，指向应用中的另一个控制器和/或action。我们来看第一行：

```
<td><%= link_to 'Show', :action => 'show', :id => entry %></td>
```

在视图中，一般Ruby代码放在<%和%>标签中，而生成内容渲染文档（显示在网页中）的Ruby代码则放在<%=和%>标签中。link_to方法接受链接的文本，然后接受散列表格式的参数，表示链接最终要指向哪里。在本例中，创建的链接指向当前控制器的show方法（你可以用:controller => 'controllername'这样的选项，指定不同的控制器），并附以当前迭代条目的ID。

例如，我们假定指向某Entry对象的entry有如下属性和数据：

```
id: 3
title: Example Entry
content: This is an example entry.
```

此时entry.id等于3，entry.title等于Example Entry，而entry.content等于This is an example entry。

我们通过几个步骤来构建link_to示例，展示一些样例视图代码，以及本例将会渲染生成什么内容：

```
<%= entry.id %>
```

3

```
<%= entry.content %>
```

```
This is an example entry.
```

```
<%= link_to 'Show', :action => 'show' %>
```

```
<a href="/entries/show">Show</a>
```

```
<%= link_to entry.title, :action => 'show', :id => entry.id %>
```

```
<a href="/entries/show/3">Example Entry</a>
```

```
<%= link_to 'Show', :action => 'show', :id => entry %>
```

```
<a href="/entries/show/3">Show</a>
```

你必须理解这些示例的工作原理，因为Ruby渲染的许多视图元素都包含此类模式，不管是生成链接，还是包含图像，或是创建用于提交数据的表单。

注 最后一个例子使用`:id => entry`而不是`:id => entry.id`。可以这样写，是因为在没有提供其他内容的情况下，创建的链接将使用`id`作为默认列。

创建新的action和视图

我们用到目前为止已掌握的基本知识，从零开始创建自己的action和视图，实现以日记或博客风格的布局，在一页中显示所有日记条目。

创建新方法很容易，只要在控制器中加入选择的方法即可。请在`app/controllers/entries_controller.rb`中加入以下方法：

```
def view_all
  @entries = Entry.find(:all, :order => 'created_at DESC')
end
```

这段代码定义了一个名为`view_all`的方法（也是控制器action），其中只有一行代码，功能是从数据库检索所有的条目，以发生时间倒序排列（像博客一样）。排序方式在可选的`:order`参数中定义。ActiveRecord的方法，例如`find`，有许多有用的此类可选参数，以便得到你想要的结果。因为这方面内容太多，无法在这里一一介绍，你可以从Ruby on Rails官方文档中了解更多信息。

有了这个action，就可以在访问`http://localhost/entries/view_all`时响应，但如果访问这个URL，会得到如下错误信息：

```
Template is missing
Missing template script/../../config/../../app/views/entries/view_all.rhtml
```

这段错误信息提示你，尽管有了action，但还没有与该action相关的视图。要创建相关视图，只须在`app/views/entries`目录中创建一个名为`view_all.rhtml`的新文件。在`view_all`。

rhtml文件中编写以下代码：

```
<% @entries.each do |entry| %>
  <h1><%= entry.title %></h1>
  <p><%= entry.content %></p>
  <p><em>Posted at <%= entry.created_at %></em></p>
<% end %>

<%= link_to 'Add New entry', :controller => 'entries', :action => 'new' %>
```

代码编写完毕后，即可成功访问 `http://localhost/entries/view_all`，生成的结果页面如图13-5所示。

我们来看一下，在新action和新视图中的代码做了什么事情。当请求 `http://localhost/entries/view_all` 时，entries控制器中的view_all方法随即运行：

```
def view_all
  @entries = Entry.find(:all, :order => 'created_at DESC')
end
```

Entries表中所有条目都通过Entry模型获得，以Entry对象形式放在@entries数组中，然后该数组被传递给关联的视图 `app/views/entries/view_all.rhtml`，其中包含如下代码：

```
<%@entries.each do |entry|%>
  <h1><%=link_to entry.title,:action =>'show',:id =>entry.id %></h1>
  <p><%=entry.content %></p>
  <p><em>Posted at <%=entry.created_at %></em></p>
<%end %>

<%=link_to 'Add New entry',:controller =>'entries',:action =>'new'%>
```

第一行用@entries的each方法，对每个元素进行循环迭代，并把每个条目放到局部变量entry中。循环中，在<h1>标签中显示了条目的标题（通过entry.title），然后显示条目的内容和创建日期。循环结束后，来自数据库的所有条目都被渲染呈现出来，最终再渲染一个指向entries 控制器中new方法的链接，这样用户可以向系统提交更多的日记条目。

参数

在上一节中，你创建了action和视图，在系统的单个类似博客网页中展示了所有日记条目。在view_all视图中有如下代码：

```
<h1><%= link_to entry.title, :action => 'show', :id => entry.id %></h1>
```

这行代码创建了一个包含链接的标题，指向显示该条目的show方法，其渲染的最终HTML如下所示：

```
<a href="/entries/show/1">This is a test</a>
```

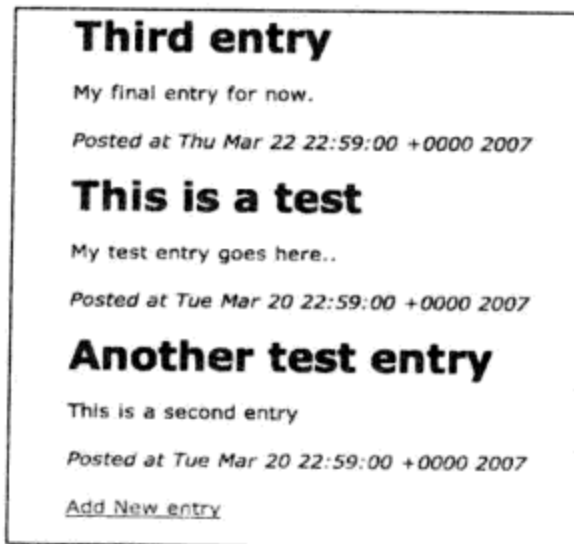


图13-5 view_all方法显示entries表中的测试条目

如果你点击该链接，将转向entries控制器的show方法，ID是1也将传递给该方法。我们来看show方法中用ID做了什么：

```
def show
  @entry = Entry.find(params[:id])
end
```

show方法很简单，它所做的一切只是从数据库检索单个条目（和往常一样，是用继承自ActiveRecord::Base的Entry模型所提供的find方法）。通过params散列表，从URL中检索出ID值，这个params散列表是在数据通过URL传递给Rails应用时，自动填充生成的。

如果用单个参数（整数ID）调用find方法，则该行数据将从与对象关联的相关表中检索出来，并以单个记录的形式返回。在本例中，检索出的条目与@entry关联，然后位于app/views/entries/show.rhtml的视图将渲染出你看到的页面。

下面是URL关联参数的一些示例，参数放在params散列表中：

http://localhost/entries/show/1

```
params[:controller] == 'entries'
params[:action] == 'show'
params[:id] == '1'
```

http://localhost/entries/another_method/20?formfield1=test&formfield2=hello

```
params[:controller] == 'entries'
params[:action] == 'another_method'
params[:id] == '20'
params[:formfield1] == 'test'
params[:formfield2] == 'hello'
```

http://localhost/test/test2/?formfield1=test&formfield2=hello

```
params[:controller] == 'test'
params[:action] == 'test2'
params[:formfield1] == 'test'
params[:formfield2] == 'hello'
```

这些示例展示了怎样通过URL把数据传递给方法（或POST请求，例如通过HTML表单提交的，把数据直接放在HTTP请求中的），然后由控制器（或视图）通过params参数获取数据。

如果查看create方法的代码——它是用于创建新条目的action，调用形式如http://localhost/entries/new，你将看到params怎样用来创建新条目：

```
def create
  @entry = Entry.new(params[:entry])
  if @entry.save
    flash[:notice] = 'Entry was successfully created.'
    redirect_to :action => 'list'
  else
    render :action => 'new'
  end
end
```



```
end
end
```

在该方法中，用`Entry.new(params[:entry])`创建一个新条目，由`ActiveRecord`提供的`new`方法接受整个`params`散列表，并用它来创建新数据行。下一行代码`@entry.save`把该行保存到数据库中，该方法返回`true`或`false`。如果是`true`（即保存成功），用户将被重定向到`list`方法（也可以改为`view_all`）。如果保存不成功，`new`方法对应的视图将用`render: action => 'new'`进行渲染。请注意，对`redirect_to`和`render`的使用方法，不必一定是常规的控制器→action→视图循环，例如把用户重定向到别处，或渲染与控制器action关联的视图。

结论观点

本节仅涵盖控制器和视图的简单用途，但这些是最根本的内容。视图和控制器提供的其他功能依赖于本节介绍的概念。URL被所需的控制器和action解析，用户提供的任何其他数据通过`params`散列表传递给action，当action代码执行完毕后，即渲染某个视图。

下一节你将考察怎样定制URL解析系统，以便可以把自己的URL格式转换成想要的控制器和方法模式。这一技术名为路由。

13.2.6 路由

当Rails应用请求不在`public`文件夹的某个页面时，Rails应用将尽量找出你想访问哪个控制器和action。在上节我们看过`http://localhost/entries/view_all`这样的URL，它表示请求是指向`entries`控制器的`view_all`方法。

也可以用路由来覆盖这一默认假设，即假设URL采用`controller_name/action_name/id`的形式。Rails应用的路由配置放在`config/routes.rb`文件中。我们来看一下它的默认内容：

```
ActionController::Routing::Routes.draw do |map|
  #The priority is based upon order of creation: first created -> highest priority

  #Sample of regular route:
  #map.connect 'products/:id', :controller => 'catalog', :action => 'view'
  #Keep in mind you can assign values other than :controller and :action
  #Sample of named route:
  #map.purchase 'products/:id/purchase', :controller => 'catalog',
    :action => 'purchase'
  #This route can be invoked with purchase_url(:id => product.id)

  #You can have the root of your site routed by hooking up ''
  #--just remember to delete public/index.html.
  #map.connect '', :controller => "welcome"

  #Allow downloading Web Service WSDL as a file with an extension
  #instead of a file named 'wsdl'
  map.connect ':controller/service.wsdl', :action => 'wsdl'

  #Install the default route as the lowest priority.
```

```
map.connect ':controller/:action/:id'
end
```

在这个文件中有许多注释，描述了怎样使用路由。这些注释值得一读，可以用它来理解当前使用的Rails版本中的路由工作原理。下面我们来看一个简单的例子。

请注意文件末尾附近的这行代码：

```
map.connect ':controller/:action/:id'
```

这是所有新Rails应用的默认路由，它只定义了URL采用controller_name/action_name/id的形式，如前文所示。请注意其中使用的:controller、:action和:id等符号，以及它们怎样与params散列表中的数据相关联。你可以用这种方法创建自己的路由，这样URL的不同部分将通过参数，用不同的元素键名传递给控制器。

我们假定你想创建一个路由，让直接对http://localhost/的请求被传递给entries控制器的view_all方法，也可以像下面这样编写路由：

```
map.connect '', :controller => 'entries', :action => 'view_all'
```

这行路由代码定义了如果URL中没有其他内容（即在URL路径的主机名之后未指定任何内容），则把entries 控制器名和view_all方法名自动加到请求中，从而传递到正确的位置。

注 新路由必须放在默认路由map.connect ':controller/:action/:id'之上，这样才能让它有较高优先级。在routes.rb文件中，路由的处理是按从上到下的优先顺序进行的。

在使用新路由之前，请务必删除public文件夹中的index.html文件。否则http://localhost/将导致public/index.html被返回，因为在匹配URL请求时，public文件夹的文件比Rails应用的优先级更高。一旦该文件被删除，路由就会正确运行，把http://localhost/请求传递给view_all方法。

路由有许多更高级的功能，但这些技术方法因要做的目标而有巨大差异。关于路由的工作原理以及怎样创建自己的高级路由，在<http://manuals.rubyonrails.com/read/chapter/65>有大量讲解。

13.2.7 模型间关系

到目前为止，你的应用只有一个模型：Entry，它与日记条目相关。而ActiveRecord程序库的一大主要优势，就是提供了轻松关联各个模型的能力。例如，可以创建另一个名为User的模型，与在系统中提交日记条目的各个人员关联。

关于ActiveRecord和模型关系（又叫关联）的全面深入的讲解，可以占用整本书的篇幅，因此超出了本书入门级的范围，但在本节中，我们将用一个简单的例子，来介绍ActiveRecord的模型怎样相互关联。

在本章上面几节中，你看过ActiveRecord对象怎样在基本层面运行。例如：

```
entry = Entry.find(1)
entry.title = 'Title of the first entry'
entry.save
```

ActiveRecord把数据与对象关联的方式是逻辑分明、直接了当的。列改变成对象的可读写属性，用对象的save方法，可把整个对象保存回数据库。

但我们假定你有个User模型，包含用户名、电子邮件地址和其他用户相关信息的列。假设你在应用中把用户表和日记条目表直接关联，可能希望实现以下操作：

```
entry = Entry.find(1)
entry.user.name = 'Name of whoever posted the entry'
entry.user.email = 'Their e-mail address'
```

事实上，这正是ActiveRecord在一对多关系上实现的功能。设置这样的模型间关系很简单，我们考虑两个模型，分别是app/models/entry.rb和app/models/user.rb：

```
class Entry < ActiveRecord::Base
  belongs_to :user
end
```

对于User模型则用下面的代码：

```
class User < ActiveRecord::Base
  has_many :entries
end
```

ActiveRecord的设计方式，让我们用几乎是自然语言的机制来定义模型间关系。在我们的Entry模型中，我们说Entry对象“belongs_to（属于）”User对象。在User模型中，我们说User对象“has_many（有许多）”关联的Entry对象。

除了模型间关系本身，还需设置的唯一一件事，是设置entries表的某一列，从而让这个关系可以运作。你需要把关联用户的id保存到每个Entry对象中，因此需要在entries表中增加名为user_id的列。也可以创建新的迁移，用add_column :entries, :user_id, :integer指令来增加该列，或用SQL（或其他客户端）手工增加该列。

在完成模型关系定义和表间数据关系设置之后，就可以很简单地进行关联操作，例如entry.user = User.find(1)，可以通过关系访问数据。如果在视图中显示条目，可以用视图代码：

```
<p>Posted by <%= entry.user.name %> at <%= entry.created_at %></p>
```

ActiveRecord还支持多对多关系。例如，考虑虚拟的Student（学生）和Class（课程）模型，学生可以同时关联多个课程，而每个课程可以包含许多学生。用ActiveRecord可以用连接表和has_and_belongs_to_many关系来定义这些关系，或通过Enrollment等中介模型来定义，该中介模型定义了Student和Class的链接。

注 值得指出一点，名为Class的模型在Rails中是不可接受的，因为Ruby已内置一个名为Class的类。因此，在起名时请注意不要与无处不在的关键字发生冲突！

ActiveRecord支持的各种关系在Ruby on Rails官方文档中有详细叙述，请访问<http://www.rubyonrails.org/api/classes/ActiveRecord/Associations/ClassMethods.html>。

13.2.8 会话与过滤器

Rails应用提供的“开箱即用”的有用功能之一，是提供了对会话的支持。当Web浏览器向你的应用发出请求时，Rails不声不响地向浏览器发回一个cookie，其中包含一个唯一标识，这样下次获得该浏览器的请求时，应用就知道是上次某个访问者发出的另一个请求。可以利用会话来保存与特定访问者相关的信息，以便用于处理未来的请求。

会话是网站常用的功能，用于暂存购物车信息，或跟踪某页面被你访问了多少次。例如，如果在某个电子商务网站把某个商品放入购物车，选中商品即保存到与会话ID相关的数据仓储中。当结账时，网站即在会话系统的数据仓储中查找会话ID，找出在购物车中放了哪些商品。

为了在Rails应用中展示简单的会话存储，你可以统计并向用户显示，对于应用的某个action，他访问了多少次。为了实现这一功能，你需要有某种方法，对发给应用的每个请求都执行这一代码逻辑。你当然可以在每个控制器action中都加入这段代码逻辑，但还有一种更简单的方法，是使用名为`before_filter`的过滤器方法。

`before_filter`方法可以用在控制器的类级别，以定义应该在当前请求的action执行之前，先执行某个方法（事实上，或许是很多方法）。过滤器可以在每个请求之前（或在某组方法或某个控制器之前）执行通用活动。

注 在Rails中，过滤器的常见用途是确认访问者是否已被认证或授权，可以访问某个控制器并执行某个action。如果有个名为`AdminController`的类，则可能需要加入一个`before_filter`，确保访问者是以管理员身份登录到网站，然后才能让他使用危险性较大的方法！

在本例中，用`before_filter`在应用的每个请求之前执行代码逻辑。要实现这一功能，要在`app/controllers/application.rb`中加入一些代码，这样应用中的每个控制器（尽管本例中只有一个控制器`entries`）都服从该过滤器约束。

下面是新代码加入之前的`app/controllers/application.rb`文件内容：

```
#Filters added to this controller will be run for all controllers in the
#application.
#Likewise, all the methods added will be available for all controllers.
class ApplicationController < ActionController::Base
end
```

提示 请注意，默认文件中的注释，它们通常提供了很有用的信息，如上面的代码所示。

下面是加入了所需统计代码之后的`app/controllers/application.rb`文件内容：

```
class ApplicationController < ActionController::Base
  before_filter :count_requests_in_session

  def count_requests_in_session
    session[:requests] ||= 0
    session[:requests] += 1
  end
end
```

在这段代码中，用`before_filter`后跟一个符号作为参数，这个符号表示`count_requests_in_session`方法。

在`count_requests_in_session`方法中用到了`session`，它是Rails提供的散列表。`session`自动且永远是当前会话相关信息的仓储空间，因此向它写入或读出的信息，永远与当前会话有关。

在本例中，如果`session[:requests]`还未定义，则将其初始化为0，然后在下一行代码增加统计值。现在可以从视图轻松地访问这一信息，我们打开`app/views/entries/view_all.rhtml`文件，并在开头加入下面这行代码：

```
<%= session[:requests] %>
```

如果现在再载入`http://localhost/entries/view_all` (或`http://localhost/`，如果你遵循上面“路由”小节代码的话)，则将在页面开头看到“1”。重新载入此页面，该数字随着每次载入而增长。会话生效了！

如果完全关闭Web浏览器，然后再打开，再做上面的操作，你会注意到数字又变成了1。这是因为在默认情况下，会话信息只保存到浏览器关闭为止。但如果你需要，也可以覆盖这一默认方式，只须在`config/environment.rb`文件中放入一些配置信息即可。也可以访问<http://errtheblog.com/post/24>了解更多内容。(关于更深层次的在Rails之外怎样使用会话，16.4节还有进一步的文档。)

13.3 其他功能特性

尽管到目前为止，已经创建了一个简单的、可运行的Rails应用，但我只介绍了基础知识。本节将介绍几个关键领域的稍稍深入的知识，可让Rails更加强大。

13.3.1 界面布局

在本章前文的Rails应用开发中，让脚手架帮你创建视图。然后再查看创建视图，了解其工作原理。此时，可能已经注意到所用的HTML代码非常简单。其中的代码只是专门用于渲染具体的页面或视图，其中没有常规HTML的头尾代码。例如，大多数HTML文档都以如下内容开始：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Page Title Here</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen"/>
  </head>
  <body>
```

并且，至少通常的HTML文档以如下内容结束：

```
</body>
</html>
```


这里的代码没有包含在看到的任何视图中。但如果用浏览器访问Rails应用时使用“查看源文件”菜单，可以清楚地看到头尾代码都有。这是因为视图是在布局中渲染的。

在Rails中，布局是特殊的通用包装模板，可用于多个视图。无须在每个视图中重复HTML头尾代码，只须把每个视图的输出嵌入到布局中即可。默认情况下，如果在`app/views/layouts`目录中有个文件名字与当前控制器相同，并以RHTML作为扩展名，则该文件用作布局文件。

下面是前文应用的`app/views/layouts/entries.rhtml`内容：

```
<html>
<head>
  <title>Entries:<%=controller.action_name %></title>
  <%=stylesheet_link_tag 'scaffold'%>
</head>
<body>

<p style="color:green"><%=flash[:notice] %></p>

<%=yield %>

</body>
</html>
```

这个布局完美地展示了布局的用途。它包含简单的HTML头尾代码，但也用一些特殊的Rails代码来把当前action名（不管它是什么）放到页面的标题中。它还用ActionPack提供的名为`stylesheet_link_tag`的辅助程序，其中包含`<link>`标签，可从`public/stylesheets/scaffold.css`载入`scaffold.css`文件，供页面内部使用。

`<p style="color: green"><%= flash[:notice] %></p>`代码的功能是：如果有`flash[:notice]`（这里的`flash`是特殊的Rails数据仓储（有点像`session`），用于返回控制器action运行过程中出现的消息），则对它进行渲染。把这段代码放到布局中而非视图中，表示无论在`entries`控制器的何处产生消息，均会正确显示在`entries`控制器渲染的当前页面上。

最后，`<%= yield %>`代码把渲染过程让渡给当前action的视图，因此当前视图的内容在此位置进行渲染。

`entries`的布局被自动使用，因为它的名字是`entries.html`，因此由`entries`控制器生成的结果将自动使用这一布局。不过，可以强制视图显示时不用布局，即只须在`entries`控制器相关方法或action进行渲染时增加一行代码，如下所示：

```
render :layout => false
```

例如，我们创建一个action，其视图是独立于布局的。在`entries`控制器中，加入下面代码：

```
def special_method_without_layout
  render :layout => false
end
```

在`app/views/entries/special_method_without_layout.rhtml`中，有如下代码：

```
<html>
```

```
<body>
<h1>This is a standalone page!</h1>
</body>
</html>
```

当`entries/special_method_without_layout`的action在渲染时，只使用视图中的代码，而布局被忽略。

也可以提供布局名，指定改用另外的布局进行渲染：

```
render :layout => 'some_other_layout'
```

这样就用`app/views/layouts/some_other_layout.rhtml`作为该视图的布局。

注 关于布局的更多内容，请访问<http://api.rubyonrails.org/classes/ActionController/Layout/ClassMethods.html>。

13.3.2 测试

在第8章我们了解了Ruby的单元测试能力，在第12章我们用它们测试了自己开发的程序库。测试让你可以指定预期结果，然后对不同功能元素（大多是方法的结果）进行测试，看实际结果是否符合预期。如果符合，则可以假定代码是正确的（或尽可能近似正确！），但如果碰到错误，可用结果来调试代码。

Rails的测试功能特别知名，大多是源自第8章和第12章介绍的Ruby的良好测试程序库。测试包在应用的根目录用Rake命令运行，测试在“测试”环境中执行，这表示测试操作是在与实际数据无关的其他数据库上进行的。

目前Rails支持三种主要的测试类型，如下所示：

- 单元测试：Rails中用单元测试来测试模型。为Rails测试系统提供“填充数据”（也就是样例数据，用它来填充到测试数据库中），并为每个模型提供一套测试包，根据数据执行操作，然后根据操作结果执行断言。这些测试包都放在`test/unit`目录中，而填充数据以YAML格式放在`test/fixtures`目录中。
- 功能测试：功能测试用来测试控制器。为每个控制器提供一套测试包，通过代码向控制器action发送请求，并根据响应结果进行断言。功能测试放在`test/functional`目录中。
- 集成测试：这是最高层的测试，集成测试用来测试整个应用。对控制器action发出请求，跟随重定向，并根据响应结果进行断言，然后继续发出更多不同类型的请求。集成测试常常作为故事级别的测试，因为可以从开头到结束测试一整块功能，像用户通过Web浏览器与应用的交互一样。集成测试定义在`test/integration`目录中。

上述测试技术的运用因开发人员而异。当下载了开源Rails应用后，发现根本没有任何测试代码，这样的情况很常见。养成坚持测试的习惯很难，更难的是学会在编写实际代码之前先编写测试代码。不过，这仍然是值得鼓励的，因为一旦把这种技术方法用到极致、你的代码将会光彩无瑕，你也会对自己的代码十分放心，知道它们容易测试，安全可靠。

注 关于Rails向标准Test::Unit断言中增加了哪些特殊类型的断言，可以访问<http://>

api.rubyonrails.org/classes/Test/Unit/Assertions.html了解更多信息。

在实际使用中，高层次的Rails开发人员倾向于先写测试代码，而初学者则不然。这也是为什么这里未对测试作深入介绍的原因。但有个很好的资源，“Rails测试指南”，可以从中学学习测试Rails应用的方方面面，访问地址为<http://manuals.rubyonrails.com/read/book/5>。这个指南的内容有点旧，但涵盖了所有基本知识。正如集成测试如何在后期阶段加入，这里也不作介绍。

不过，最好首先了解开发Rails应用的全套方法，生成几个“用完即扔”的应用，然后在熟悉了整个框架之后，再回过头来关注测试。

13.3.3 插件

在Rails中，插件是特殊的程序库，它向Rails框架中增加功能，影响范围是安装插件的应用本身。Rails应用可以使用插件所增加的功能。

Rails插件有成百上千个，随着Rails开发人员不断产生新点子，这一数字还在持续增长。可以获取插件，并为你的应用创建图形，为代码增加标签功能，甚至增加一大块功能，例如为应用增加整个认证系统。

安装插件甚至比安装gem包还简单，请使用script/plugin脚本，如下所示：

```
ruby script/plugin install <url of plugin here>
```

注 与gem包一样，插件脚本支持install、remove和update操作，可对插件执行相关操作。关于插件支持的各种操作，运行ruby script/plugin -h命令可了解更多内容。

待安装插件的URL由插件作者提供，一般在他们自己的网页上，或在得知该插件的任何地方。

以下展示了acts_as_commentable插件的安装：

```
ruby script/plugin install http://juixe.com/svn/acts_as_commentable/
```

```
+./acts_as_commentable/MIT-LICENSE
+./acts_as_commentable/README
+./acts_as_commentable/init.rb
+./acts_as_commentable/install.rb
+./acts_as_commentable/lib/acts_as_commentable.rb
+./acts_as_commentable/lib/comment.rb
+./acts_as_commentable/tasks/acts_as_commentable_tasks.rake
+./acts_as_commentable/test/acts_as_commentable_test.rb
```

运行script/plugin，结果显示哪些文件被加入到项目插件，保存在vendor/plugins文件夹中，因此这个插件位于vendor/plugins/acts_as_commentable。

当下次启动Rails应用时，acts_as_commentable插件将自动载入（和位于vendor/plugins的所有其他插件一样），你就可以在你自己的应用中使用其功能。

插件的使用方法因插件不同而有相当差异，而寻找插件的好地方是<http://www.agilewebdevelopment.com/plugins>和<http://plugins.radrails.org/>。

注 你可从<http://wiki.rubyonrails.org/rails/pages/Plugins>了解更多插件知识。

13.4 参考资料与演示应用

自从2004年末以来，Rails已经变得非常流行。它吸引了成千上万开发人员的兴趣，他们中的许多人在博客上谈论这个框架，或免费发布自己的Rails应用源代码。也可以看看一些鼓舞人心的大规模Rails应用。

学习Rails的最好办法（基础知识除外），是跟踪正在开发的最新功能，阅读其他人开发的源代码，以及亲自动手试验。Rails并不是可以轻松掌握的东西。

本节提供一些链接，指向几个有用的参考资源和示例应用，供你学习研究。

13.4.1 参考站点和教程

下面是一些有用的参考资料网站和教程，可帮你从Rails起步：

- Ruby on Rails官方API (<http://api.rubyonrails.org/>)：这是Ruby on Rails框架的官方文档，Rails几乎为每个类和方法都提供了文档。
- Ruby on Rails官方维基 (<http://wiki.rubyonrails.org/>)：这是一套有用的、访问者可更新的Ruby on Rails文档。提供了几百篇文章，讨论Rails的各个不同方面。

警告 由于维基可由任何访问者更新，因此维基的页面可能损坏或包含恶劣语言，在访问之前请注意。

- Ruby on Rails屏幕录像 (<http://www.rubyonrails.org/screencasts>)：这里提供视频录像，展示Rails应用怎样开发。这些录像是很有价值的材料，可以帮你巩固本章学到的内容，甚至可以显示同一问题有不同解决方法。
- Ruby on Rails的摸爬滚打 (<http://www.onlamp.com/pub/a/onlamp/2006/12/14/revisiting-ruby-on-rails-revisited.html/>)：这是对Ruby on Rails的基础介绍，由Bill Walton和Curt Hibbs制作，涵盖与本章开头几节类似的基础知识。但也能用来帮你巩固已学知识。

13.4.2 Rails示例应用

下面是一些应用，可以下载、试用、修改它们，从而对Ruby on Rails了解更多：

- Tracks (<http://www.rousette.org.uk/projects/>)：这是用Rails开发的开源时间管理系统，它是极好的早期示例，通读可以了解更多Rails知识。
- Typo (<http://www.typosphere.org/>)：这是用Ruby on Rails开发的开源博客引擎。
- Mephisto (<http://mephistoblog.com/>)：这是另一个Rails开源博客引擎。

- Instiki (<http://www.instiki.com/>): 这是用Ruby on Rails开发的一套维基系统, 开始是由Rails的创始人David Heinemeier Hansson开发。
- Ruby资产管理器 (<http://www.locusfoc.us/ram/>): 这是用Ruby on Rails开发的一套资产管理器, 提供了文件上传、RSS、存储和导出功能。

13.5 小结

本章我们介绍了怎样用Ruby on Rails框架开发简章的Web应用。Rails框架提供了强大的“开箱即用”能力, 并可以在较短时间内开发出全功能的Web应用。

本章我们仅仅了解了皮毛, 因为Ruby on Rails是个复杂的大型框架(尽管用起来很简单, 但它有许多复杂细节, 供高级用途使用)。关于Rails可以写一整本书, 比本书还厚, 因此本章仅提供浅尝辄止的体验。你可以用上一节提供的参考资源来学习该框架的更多内容, 或可以选购Apress公司出版的Rails图书。

Rails一开始看起来很复杂, 但它复杂的目录结构和rails工具创建的默认文件, 只是为了让你的开发工作更简单。一旦熟悉了Rails的布局 and 工具, 开发Web应用就变成了简单而有条不紊的过程。

我们来回顾一下本章涵盖的主要概念:

- Ruby on Rails: 是个基于Ruby的Web应用开发框架, 由David Heinemeier Hansson开发。关于Ruby on Rails的幕后历史, 请参阅第5章。
- 框架: 是一组程序库和工具, 可用于应用开发的基础。
- 模型: 是表示数据的类, 这些数据是应用所需的, 且包含操作和检索的代码逻辑。
- 视图: 是模板和HTML代码(更具体地说, 代码既包含HTML, 也包含嵌入式Ruby代码), 用来生成Web应用用户可见的页面。视图可以用HTML输出数据(当用户是Web浏览器时), 也可以输出XML、RSS和其他格式。
- 控制器: 是处理用户输入, 并控制发送什么数据给视图供其输出的类。控制器包含模型、数据和视图代码的绑定逻辑。
- action: 是控制器中包含的方法。
- CRUD: 是创建、读取、更新、删除(Create、Read、Update和Delete)的缩写。有四种基本的action, 可用于最常见的Web应用。在Rails 1.2和更高版本中, 这些操作将对应于PUT、GET、POST和DELETE这四种操作。
- ActiveRecord: 是把数据库、行、列和SQL抽象成标准的Ruby语法, 用类和对象来表示。它是Ruby on Rails框架的主要组成部分。
- 路由: 通过路由模式, 把URL翻译成所需的控制器和action的过程。
- 会话: 给新用户赋予唯一ID, 并在后续每个请求中都来加传递该ID, 因此可以用它来跟踪用户信息。
- 插件: Ruby on Rails框架的Ruby程序库。插件可覆写Rails的默认行为, 或用新功能扩展框架, 例如认证系统。插件是以预安装的方式安装的, 而不是Rails框架整体的一部分。

本章我们介绍了基于框架开发Web应用, 但在下一章我们将介绍更直接地使用因特网协议。你可以把第14章的技术与Rails应用结合起来, 从而可以与电子邮件、FTP等其他在线服务, 以及与其他网站的数据进行通信。

第14章 Ruby与因特网

本章我们将考察怎样用Ruby通过Web、电子邮件和文件传输，使用因特网和因特网上各种可用服务。

因特网对软件开发来说已变得不可或缺，而Ruby提供了相当多的程序库来处理不可计数的因特网可用服务。本章我们专门讨论几个较流行的服务：Web、电子邮件（POP3和SMTP）以及FTP，并介绍怎样处理从这些服务获取的数据。

在第15章我们将介绍怎样使用Ruby和底层网络功能，例如，ping操作、TCP/IP和套接字，来开发真实的服务器和后台进程代码。但本章只关注怎样访问因特网并处理获取的数据，而不是Ruby网络连接功能的细节。

14.1 HTTP与万维网

超文本传输协议（HyperText Transfer Protocol, HTTP）是一种因特网协议，它定义了Web服务器和Web客户端（例如Web浏览器）怎样相互通讯。HTTP以及普通意义上Web的基本原理，是每个资源（例如网页）在Web上都有唯一的统一的资源定位符（Uniform Resource Locator, URL），而Web客户端可以用诸如GET、POST、PUT和DELETE等HTTP“动词”来检索或操作这些资源。例如，当Web浏览器检索网页时，即是对正确的Web服务器发出对该网页的GET请求，服务器随即返回该网页的内容。

在第10章我们简要了解了HTTP，并开发了一些简单的Web服务器应用程序，以展示Ruby应用程序在因特网上怎样提供可用功能。本节我们将考察怎样从Web检索数据、解析数据并生成Web兼容的内容。

14.1.1 下载网页

对Web操作的最基本动作，就是下载一个网页或文档。首先我们将考察怎样使用最常用的Ruby HTTP程序库net/http，然后再介绍一些值得一看的同类程序库。

net/http程序库

net/http程序库是Ruby标准程序库中自带的，也是用来访问Web网站最常用的程序库。下面是个简单的例子：

```
require 'net/http'

Net::HTTP.start('www.rubyinside.com') do |http|
  req = Net::HTTP::Get.new('/test.txt')
  puts http.request(req).body
end
```

```
Hello Beginning Ruby reader!
```

本例载入net/http程序库，并连接到www.rubyinside.com 的Web服务器（这是本书的半官方博客，请看一下！），然后对/test.txt执行HTTP的GET请求。该文件内容随即被返回，并显示在屏幕上。与该请求等价的URL是http://www.rubyinside.com/test.txt，如果在Web浏览器中载入该URL，将得到与上面Ruby程序相同的反馈结果。

注 http://www.rubyinside.com/test.txt是有效文档，已为本书读者专门创建，你可以在所有HTTP请求的测试中安全使用。

如上例所示，net/http程序库的使用有点原始。不是光把URL传递给它就行了，还要必须把要连接的Web服务器传递给它，然后再指定该服务器上的本地文件名。还得指定HTTP请求类型为GET，并用request方法激活请求。可以用Ruby自带的URI程序库来简化这些工作，该程序库提供许多方法，可以把URL转成不同net/http所需的各种片断。示例如下：

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.txt')

Net::HTTP.start(url.host,url.port)do |http|
  req = Net::HTTP::Get.new(url.path)
  puts http.request(req).body
end
```

本例用URI类（由net/http自动载入）来解析提供的URL，返回结果是一个对象，其host、port和path方法分别提供URL的不同部分，供Net::HTTP使用。请注意，在本例中你向Net::HTTP.start主方法提供了两个参数：URL的主机名和URL的端口号。端口号是可选的，但URI很聪明，可以返回HTTP默认端口号80。

我们甚至还可以做一个更简单的例子：

```
require 'net/http'

url = URI.parse('http://www.rubyinside.com/test.txt')
response = Net::HTTP.get_response(url)
puts response.body
```

这里虽然没有创建HTTP连接和发出GET请求，但是用Net::HTTP.get_response方法一步到位地完成请求。在某些情况下这么做可能不够灵活，但如果只是想通过Web检索文档，这个方法就是理想的选择。

检查错误并重定向

到目前的示例都假定用的是合法的URL，采访问真实存在的文档。根据请求是否成功，或客户是否被重定向到其他URL，Net::HTTP将返回不同的响应，也可以对此进行检查。在下面的例子中，创建了名为get_web_document的方法，该方法接受一个URL作为参数，对该URL进行解析，尝试获取所需的文档，然后根据case/when代码块的判定返回结果：

```
require 'net/http'
```

```

def get_web_document(url)
  uri =URI.parse(url)
  response =Net::HTTP.get_response(uri)

  case response
  when Net::HTTPSuccess:
    return response.body
  when Net::HTTPRedirection:
    return get_web_document(response ['Location'])
  else
    return nil
  end
end

puts get_web_document('http://www.rubyinside.com/test.txt')
puts get_web_document('http://www.rubyinside.com/non-existent')

```

```

Hello Beginning Ruby reader!
nil

```

如果响应结果属于`Net::HTTPSuccess`类，则返回响应内容；如果响应结果是个重定向（由返回的`Net::HTTPRedirection`对象表示），则再次调用`get_web_document`方法，并以远程服务器返回的重定向目标作为URL参数；如果响应结果既不是成功也不是重定向，则激活某种类型的错误，并返回`nil`。

如果你愿意，可以用更具体的方式检查错误。例如，404错误表示“文件未找到”，特别用于试图请求远程服务器上某个没有的文件的情况。当发生此错误时，`Net::HTTP`返回`Net::HTTPNotFound`类的响应结果。但在处理403错误（即“禁止访问”）时，`Net::HTTP`返回`Net::HTTPForbidden`类的响应结果。

注 HTTP错误列表及其相关联的`Net::HTTP`响应类，可在以下网址访问：[http://www.ruby-doc.org/stdlib/libdoc/net/http/rdoc/classes/ Net/HTTP.html](http://www.ruby-doc.org/stdlib/libdoc/net/http/rdoc/classes/Net/HTTP.html)。

基础认证

除了基本的文档检索功能，`net/http`还支持基础认证（Basic Authentication）模式，许多Web服务器都用它来保护位于密码保护区内的文档。这里的示例展示了用`Net::HTTP.start`执行整个请求的灵活性这非常有用：

```

require 'net/http'

url =URI.parse('http://www.rubyinside.com/test.txt')

Net::HTTP.start(url.host,url.port)do |http|
  req =Net::HTTP::Get.new(url.path)
  req.basic_auth('username','password')

```

```
puts http.request(req).body
end
```

本例对于RubyInside网站的URL仍然有用，因为在请求非保护URL时将忽略认证。但如果要访问由基础认证保护的URL，则`basic_auth`方法允许你指定证书。

提交表单数据

在目前的示例中，我们仅从Web检索获取数据。另一种形式的交互是把数据发送给Web服务器，对此最常见的例子是在网页填写表单。也可以用Ruby完成同样的操作，例如：

```
require 'net/http'

url =URI.parse('http://www.rubyinside.com/test.cgi')

response =Net::HTTP.post_form(url,{'name'=>'David','age'=>'24'})
puts response.body
```

```
You say David is 24 years old.
```

在本例中，用`Net::HTTP.post_form`方法，向指定URL执行HTTP的POST请求，并以散列表参数中的数据作为表单数据。

注 `test.cgi`是个特殊程序，它根据`name`和`age`表单字段的值，返回包含这些字段值的字符串，并生成上例的输出结果。我们在第10章讨论过怎样创建CGI脚本。

和基本文档检索示例一样，完成同样的事情还有更复杂、更底层的方式，可以对表单提交过程的每一步进行控制：

```
require 'net/http'

url =URI.parse('http://www.rubyinside.com/test.cgi')

Net::HTTP.start(url.host,url.port)do |http|
  req =Net::HTTP::Post.new(url.path)
  req.set_form_data({'name'=>'David','age'=>'24'})
  puts http.request(req).body
end
```

如果需要的话，也允许你使用`basic_auth`方法。

使用HTTP代理

当HTTP请求不直接在客户端和HTTP服务器之间传递，而是通过第三方路径传递时，称为使用代理。在某些情况下可能需要为HTTP请求使用HTTP代理，这是学校和办公室的常见情况，其Web访问通常受到限制或过滤。

`net/http`支持代理操作，方法是创建HTTP代理类，并用它来执行常规的HTTP方法。要创建代理类，请用`Net::HTTP::Proxy`，例如：

```
web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)
```

对`Net::HTTP::Proxy`的调用生成一个HTTP代理，它使用位于某个特定主机名端口号为

8080的代理。可以用下面的方式来使用这个代理：

```
require 'net/http'

web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)

url = URI.parse('http://www.rubyinside.com/test.txt')

web_proxy.start(url.host, url.port) do |http|
  req = Net::HTTP::Get.new(url.path)
  puts http.request(req).body
end
```

在本例中，当调用start方法时，web_proxy取代了Net::HTTP。也可以用前面用过的简单方法get_response：

```
require 'net/http'

web_proxy = Net::HTTP::Proxy('your.proxy.hostname.or.ip', 8080)
url = URI.parse('http://www.rubyinside.com/test.txt')

response = web_proxy.get_response(url)
puts response.body
```

本例说明，如果你的程序可能需要HTTP请求的代理支持，即使在各种情况下都不需要代理，就值得生成代理式系统。举例如下：

```
require 'net/http'

http_class = ARGV.first ? Net::HTTP::Proxy(ARGV[0], ARGV[1]) : Net::HTTP
url = URI.parse('http://www.rubyinside.com/test.txt')

response = http_class.get_response(url)
puts response.body
```

如果该程序运行，且程序的命令行参数提供了HTTP代理的主机名和端口，则将某个HTTP代理类赋给http_class。如果没有指定任何代理。则http_class将只引用到Net::HTTP。这样一来，当发出请求时，就可以在Net::HTTP的位置使用http_class，以便在代理和非代理情况都能正常运行，并且两种情况的代码编写是完全一样。

用HTTPS加强HTTP的安全

HTTP是纯文本、非加密的协议，这样一来它就不适合传输敏感数据，例如信用卡信息。HTTPS是这个问题的解决方案，因为它与HTTP相同，但通过安全套接字层（Secure Socket Layer, SSL）进行路由，这就让任何第三方都无法窥视。

Ruby的net/https程序库可以访问HTTPS的URL，也可以把Net::HTTP实例的use_ssl属性设为true，从而让net/http以半透明的方式使用HTTPS，如下所示：

```
require 'net/http'
require 'net/https'
```



```
url =URI.parse('https://example.com/')

http =Net::HTTP.new(url.host,url.port)
http.use_ssl =true if url.scheme =='https'
```

```
request =Net::HTTP::Get.new(url.path)
puts http.request(request).body
```

请注意，上面使用了url的scheme方法，来检测远程URL是不是需要激活SSL的URL。把提交表单的代码混合进来并不复杂，这样可以用安全的方式把敏感信息发送给远程服务器：

```
require 'net/http'
require 'net/https'
```

```
url =URI.parse('https://example.com/')

http =Net::HTTP.new(url.host,url.port)
http.use_ssl =true if url.scheme =='https'
```

```
request =Net::HTTP::Post.new(url.path)
request.set_form_data({'credit_card_number'=>'1234123412341234'})
puts http.request(request).body
```

net/https程序库还可以把请求与你的客户端认证以及认证目录关联起来，并可检索服务器的对等认证。不过，这些都是高级功能，只在很少的情况下使用，也超出了本节范围。如要了解更多信息，请参阅附录C的链接。

open-uri程序库

open-uri是一个程序库，它把net/http、net/https和net/ftp的功能封装在一个程序包中。尽管它缺少一些直接使用底层程序库的原始能力，但大大简化了所有主要功能的执行。

open-uri的关键部分是它对常用因特网操作的抽象方式，允许对因特网操作使用文件I/O方法。从Web检索文档变得像打开本机文件一样：

```
require 'open-uri'

f =open('http://www.rubyinside.com/test.txt')
puts f.readlines.join
```

```
Hello Beginning Ruby reader!
```

与File::open一样，上面的open方法返回一个I/O对象（从技术上说，是StringIO对象），而且可以用each_line、readlines和read等方法，与第9章的做法一样。

```
require 'open-uri'

f =open('http://www.rubyinside.com/test.txt')

puts "The document is #{f.size}bytes in length"

f.each_line do |line|
```

```
puts line
end
```

```
The document is 29 bytes in length
Hello Beginning Ruby reader!
```

而且，与File类的方式类似，也可以用代码块的方式使用open方法：

```
require 'open-uri'

open('http://www.rubyinside.com/test.txt')do |f|
  puts f.readlines.join
end
```

注 HTTPS和FTP的URL对open方法是透明的，你可以用任何HTTP、HTTPS或FTP的URL。

open方法除了作为可在任何地方使用的基础方法，还可以在URI对象直接调用：

```
require 'open-uri'

url = URI.parse('http://www.rubyinside.com/test.txt')
url.open { |f| puts f.read }
```

如果你力图编写出最短的open-uri代码，也可以用下面这段代码：

```
require 'open-uri'
puts URI.parse('http://www.rubyinside.com/test.txt').open.read
```

注 Ruby开发人员通常使用快捷代码，例如上面的例子，但为了有效地捕捉到错误，推荐用begin/ensure/end结构把这样的单行代码包裹起来，以便捕获任何异常。

除了操作与I/O对象类似，open-uri还可以使用返回对象的方法，以找到HTTP（或FTP）响应本身的特殊内容。例如：

```
require 'open-uri'

f = open('http://www.rubyinside.com/test.txt')

puts f.content_type
puts f.charset
puts f.last_modified
```

```
text/plain
iso-8859-1
Sun Oct 15 02:24:13 +0100 2006
```

最后，可以在HTTP请求中发送额外的头域，方法是在调用open时提供可选的散列参数：

```
require 'open-uri'
```

```
f = open('http://www.rubyinside.com/test.txt',
        {'User-Agent' => 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)'})

puts f.read
```

在本例中，“user agent”头附加在HTTP请求中一起发送，看起来仿佛在用IE浏览器请求远程文件。如果你访问的Web站点对不同类型的浏览器返回不同的信息，那么发送“用户代理”头信息就是一种有用的方法。但在理想情况下，应该使用User-Agent头，以反映程序的名字。

14.1.2 生成网页和HTML

网页可以用不同技术创建，最流行的是用超文本标记语言（HyperText Markup Language, HTML）。HTML是一种语言，以纯文本表示，由许多标签（tag）组成，这些标签分别表示文档各个部分的意义。例如：

```
<html>
  <head>
    <title>This is the title</title>
  </head>
  <body>
    <p>This is a paragraph</p>
  </body>
</html>
```

标签以如下样式开始：

```
<tag>
```

并以如下样式结尾：

```
</tag>
```

在开始标签和结束标签之间的任何内容，在语义上都属于这个标签。因此，在<p>和</p>标签之间的文本都是单个段落（<p>是表示段落的HTML标签）的组成部分。这就解释了为什么整个文档由<html>和</html>包围起来，因为整个文档都是HTML。

Web应用程序，以及其他需要输出数据到Web的应用程序，通常都需要生成HTML来渲染其输出内容。Ruby提供了许多程序库来简化这一工作，让你无须以原始的字符串方式生成HTML。本节将介绍此类程序库中的两个：Markaby和RedCloth。

Markaby——Ruby代码式标记

Markaby是由Tim Fletcher (<http://tfletcher.com/>) 和“why the lucky stiff” (<http://whytheluckystiff.net/>) 开发的程序库，允许用Ruby方法和结构来创建HTML。Markaby以gem包形式提供，因此用gem客户端来安装很简单，如下所示：

```
gem install markaby
```

Markaby安装好完毕后，就可以试试下面的例子，它们展示了用Markaby生成HTML的基本原理：

```
require 'rubygems'
```

```

require 'markaby'

m =Markaby::Builder.new

m.html do
  head {title 'This is the title'}

  body do
    h1 'Hello world'
    h2 'Sub-heading'
    p %q{This is a pile of stuff showing off Markaby's features}
    h2 'Another sub-heading'
    p 'Markaby is good at:'
    ul do
      li 'Generating HTML from Ruby'
      li 'Keeping HTML structured'
      li 'Lots more..'
    end
  end
end

puts m

```

```

<html><head><meta content="text/html; charset=utf-8" http-equiv="Content-
Type"/><title>This is the title</title></head><body><h1>Hello world</h1><h2>Sub-
heading</h2><p>This is a pile of stuff showing off Markaby's features</p><h2>
Another sub-heading</h2><p>Markaby is good at:</p><ul><li>Generating HTML from
Ruby</li><li>Keeping HTML structured</li><li>Lots more..</li></ul></body></html>

```

输出结果是简单的HTML，可用任何Web浏览器查看。Markaby的工作原理是把方法调用解释为HTML标签，这样就可以只用方法调用来创建HTML标签。如果方法调用传递给代码块（如上例中m.html和body所示），则代码块中的代码将生成HTML，这些HTML位于父标签的开始与结束标签之间。

因为Markaby提供了纯Ruby的方式来生成HTML，因此可以把Ruby代码逻辑和流程控制放在HTML生成过程中：

```

require 'rubygems'
require 'markaby'

m =Markaby::Builder.new

items =['Bread','Butter','Tea','Coffee']

m.html do
  body do
    h1 'My Shopping List'
    ol do

```

```

        items.each do |item|
          li item
        end
      end
    end
  end
end

puts m

```

```

<html><body><h1>My Shopping
List</h1><ol><li>Bread</li><li>Butter</li><li>Tea</li>
<li>Coffee</li></ol></body></html>

```

如果用常规Web浏览器查看这些输出内容，看起来应该像下面这样：

My Shopping List

1. Bread
2. Butter
3. Tea
4. Coffee

你经常用到的一个HTML常用功能是给元素赋以类名或ID名。要给元素赋以类名，可以用该元素附加的方法调用。要给元素赋以ID，可以用该元素附加的方法调用后跟感叹号(!)结尾。例如：

```

div.posts!do
  div.entry do
    p.date Time.now.to_s
    p.content "Test entry 1"
  end

  div.entry do
    p.date Time.now.to_s
    p.content "Test entry 2"
  end
end

```

```

<div id="posts"><div class="entry"><p class="date">Mon Oct 16 02:48:06 +0100
2006</p><p class="content">Test entry 1</p></div><div class="entry"><p
class="date">Mon Oct 16 02:48:06 +0100 2006</p><p class="content">Test entry
2</p></div></div>

```

在本例中，父div元素以"posts"的id创建，两个子div元素以"entry"的类名创建，其中分别包含了两个类名为"date"和"content"的段落。

必须注意的是，由Markaby生成的HTML不一定是合法的HTML。对标签和结构的正确使用，是开发者的责任。

注 要了解Markaby的更多内容, 请参阅<http://markaby.rubyforge.org/>的Markaby官方主页和文档。

RedCloth

RedCloth是一套程序库, 提供了Textile标记语言的Ruby实现。Textile标记语言是对纯文本的一种特殊格式方法, 可被转换成HTML。

下面是Textile的示例, 后跟Textile解释器生成的HTML代码:

```
h1.This is a heading.
```

```
This is the first paragraph.
```

```
This is the second paragraph.
```

```
h1.Another heading
```

```
h2.A second level heading
```

```
Another paragraph
```

```
<h1>This is a heading.</h1>
<p>This is the first paragraph.</p>
<p>This is the second paragraph.</p>
<h1>Another heading</h1>
<h2>A second level heading</h2>
<p>Another paragraph</p>
```

Textile提供了更加友好易读的语言, 转换成HTML非常容易。RedCloth为Ruby提供了这一功能。

RedCloth以RubyGem形式提供, 可用常规方法安装 (例如`gem install redcloth`), 更多信息请参见第7章。要使用RedCloth, 请创建RedCloth类的一个实例, 并把要使用的Textile代码传递给这个实例:

```
require 'rubygems'
require 'redcloth'

text =%q{h1.This is a heading.

This is the first paragraph.

This is the second paragraph.

h1.Another heading

h2.A second level heading

Another paragraph}
```

```
document = RedCloth.new(text)
puts document.to_html
```

RedCloth类是String类的基本扩展，因此可以对RedCloth对象调用常规字符串方法，也可以用to_html方法把RedCloth/Textile文档转换成HTML。

Textile语言是一种强大的标记语言，但其语法超出了本章的范围。它提供了简单的方式，把纯文本转换成包含表、实体、图像和结构元素的复杂HTML。要了解RedCloth和Textile的更多内容，请参阅RedCloth官方网站<http://redcloth.rubyforge.org/>。

注 BlueCloth是Ruby的另一种标记语言程序库，以gem包形式存在。你可以在第16章或BlueCloth官方网站<http://www.deveiate.org/projects/BlueCloth>了解其更多内容。

14.1.3 解析网页内容

如前文所见，用Ruby从Web检索数据是小菜一碟。一旦检索到数据，下面就该对这些数据进行一些操作。用正则表达式和常规Ruby字符串方法解析来自Web的数据，是一种选择，但已经有几种程序可以简化这些工作，它们能对不同形式的Web内容进行专门处理。本节我们将介绍几种处理HTML和XML（包含RSS和Atom等订阅源（feed）格式）的最好方法。

用Hpricot解析HTML

在上一节，我们用Markaby和RedCloth，从Ruby代码和数据生成了HTML。本节我们将考察怎样做相反的工作，以结构化方式从HTML代码中分解数据。

Hpricot是由“why the lucky stiff”开发的Ruby程序库，用于实现快速、简单和有趣的HTML解析。它以gem包形式存在，通过gem install hpricot命令安装。尽管为了速度的需要，它依赖于C语言编写的编译扩展，但有个Windows特供版本，通过RubyGems包含了预编译的扩展。

安装完毕后，Hpricot很容易使用。下面的例子是载入Hpricot程序库，把一些基本的HTML放到字符串中，创建一个Hpricot对象，搜索H1标签（用search方法），然后检索第一个元素（用first方法，因为search返回数组），并查看其中的HTML（用inner_html方法）。

```
require 'rubygems'
require 'hpricot'

html = <<END_OF_HTML
<html>
<head>
  <title>This is the page title</title>
</head>

<body>
  <h1>Big heading!</h1>
  <p>A paragraph of text.</p>
  <ul><li>Item 1 in a list</li><li>Item 2</li><li class="highlighted">Item
```

```
3</li></ul>
</body>
</html>
END_OF_HTML
```

```
doc = Hpricot(html)
puts doc.search("h1").first.inner_html
```

Big heading!

Hpricot可以直接用于open-uri程序库，以便从远程文件载入HTML，如下例所示：

```
require 'rubygems'
require 'hpricot'
require 'open-uri'

doc = Hpricot(open('http://www.rubyinside.com/test.html'))
puts doc.search("h1").first.inner_html
```

注 <http://www.rubyinside.com/test.html> 包含与前文示例相同的HTML代码。

通过使用搜索方法的组合，可以在HTML中搜索列表（由标签定义，其中表示列表中的每个元素），并从列表中分解出每个元素：

```
list = doc.search("ul").first
list.search("li").each do |item|
  puts item.inner_html
end
```

```
Item 1 in a list
Item 2
Item 3
```

除了搜索并返回数组，Hpricot还可以用at方法，只搜索某元素的第一个实例：

```
list = doc.at("ul")
```

不过，Hpricot可以搜索元素或标签名之外的更多内容，它也支持XPath和CSS表达式。这些查询风格超出了本章的范围，这里仅对通过CSS类查找某类元素作小小展示：

```
list = doc.at("ul")
highlighted_item = list.at("/.highlighted")
puts highlighted_item.inner_html
```

```
Item 3
```

本例找到了HTML文件中的第一个列表，然后查找类名为highlighted的子元素。highlighted是查找类名为highlighted的元素，而#highlighted则是查找ID为highlighted的元素。

注 可以在CSS表达式前加正斜杠 (/)。

可以访问<http://code.whytheluckystiff.net/hpricot/>官方网站，了解Hpricot及其语法和所支持风格的更多内容。Hpricot正在不断发展中，在本书出版时，它的功能集很可能已经有所增长。

用REXML解析XML

可扩展的标记语言（Extensible Markup Language，简称XML）是一种简单灵活的纯文本数据格式，可用来表达许多种类的数据结构。最简单的XML文档看起来像是HTML：

```
<people>
  <person>
    <name>Peter Cooper</name>
    <gender>Male</gender>
  </person>
  <person>
    <name>Fred Bloggs</name>
    <gender>Male</gender>
  </person>
</people>
```

这段极度简化的XML文档定义了一组人，其中包含两个人，每人都有名字和性别。在前面的章节中，我们以相似的方式用过YAML，尽管YAML对Ruby更简单更易用，而XML在Ruby世界之外更流行。

随着因特网对共享数据的需要，XML逐渐流行开，因为它的方式容易被机器解析，特别是在用API和机器可访问的在线服务时，例如雅虎的搜索API以及为其他语言提供的在线服务接口。由于XML很流行，因此值得了解一下用Ruby怎样解析XML。

Ruby主要的XML程序库名为REXML，它是Ruby默认自带的，是标准程序库的组成部分。

REXML支持两种不同方式来处理XML文件：树解析和流解析。树解析是把整个XML文件放到单个数据结构中，然后可进行搜索、遍历和其他操作。而流搜索则是对XML文件边处理边解析，当在文件中发现任何有用内容时，即调用特殊的回调函数。流解析在大多数情况下不如树解析有用，尽管速度略微更快。本节我们将专门讨论树解析，因为它对大多数情况更有意义。

下面是对解析XML文件、寻找某种元素的基本展示，

```
require 'rexml/document'

xml = <<END_XML
<people>
  <person>
    <name>Peter Cooper</name>
    <gender>Male</gender>
  </person>
  <person>
    <name>Fred Bloggs</name>
    <gender>Male</gender>
  </person>
</people>
END_XML
```

```
tree = REXML::Document.new(xml)

tree.elements.each("people/person") do |person|
  puts person.get_elements("name").first
end
```

```
<name>Peter Cooper</name>
<name>Fred Bloggs</name>
```

本例首先新建REXML::Document对象，以构建XML元素树。使用tree的elements方法返回由XML文件中每个元素组成的数组，each方法接受XPath查询（即XML搜索查询的一种形式），并把匹配的元素传递给关联的代码块。本例的目标是在<people>元素中寻找每个<person>元素。

一旦得到person中的每个<person>元素，就用get_elements方法在数组中检索<name>元素，然后取出第一个。因为在XML数据中每个人只有一个名字，因此可以从数据中得到正确的名字。

REXML支持大多数基本XPath规约，因此如果熟悉XPath，就可以在XML中搜索任何需要的内容。

注 可以访问<http://en.wikipedia.org/wiki/XPath>了解XPath的更多内容及其语法。REXML也支持XQuery，可从<http://en.wikipedia.org/wiki/XQuery>了解更多内容。

关于处理XML和在Ruby中使用REXML与XPath的进一步资源，请参阅附录C。

用FeedTools解析Web订阅源

Web订阅源（feed）（有时称为新闻订阅源，更常见的称呼就是“订阅源”）是特殊的XML文件，专用于包含多个内容条目（例如新闻）。通常用于博客和新闻站点供用户订阅。订阅源阅读器从用户订阅的站点读取RSS和Atom源内容（这是两种最流行的订阅源格式），一旦订阅源中出现新条目，定期监视的订阅源客户端就会立即通知用户。大多数订阅源允许用户阅读条目梗概，点击链接后再访问更新的站点。

注 订阅源的另一种常见用途，是发送音频广播，这是通过广播订阅类型格式，在线发布音频内容的流行方法。

处理RSS和Atom订阅源，已经变成Ruby这类语言的流行任务。因为订阅源采用机器友好的格式，因此更容易被程序处理和使用，而不是通过不稳定的HTML进行扫描。

FeedTools (<http://sporkmonger.com/projects/feedtools/>) 是一个Ruby程序库，用于处理RSS和Atom订阅源。它以RubyGem形式存在，用gem install feedtools命令安装。它是一种自由式订阅源解析器，这表示它会尽量容忍读取的订阅源中许多错误和格式问题。因此它是处理订阅源的理想选择，而不是通过REXML或其他XML程序库，手工创建你自己的解析器。

本节的示例将使用RubyInside.com这个流行的Ruby博客所提供的RSS订阅源。我们来看一下怎样快速处理订阅源，即从Web读取条目，并以不同的详细程度打印输出条目及其子条目的内容：

```
require 'rubygems'
require 'feed_tools'

feed = FeedTools::Feed.open('http://www.rubyinside.com/feed/')

puts "This feed's title is #{feed.title}"
puts "This feed's Web site is at #{feed.link}"

feed.items.each do |item|
  puts item.title + "\n---\n" + item.description + "\n \n"
end
```

解析订阅源甚至比下载网页更简单，因为FeedTools处理了所有技术层面的事务。它负责下载订阅源，甚至把订阅源的内容缓存一段较短时间，以便不向订阅源提供方发送大量请求。

上例是打开订阅源，打印输出订阅源的标题，打印输出与订阅源关联站点的URL，然后用feed.items中的条目数组，打印输出标题和订阅源中每个条目的简要说明。

除了description和title方法，feed条目（即FeedTools::FeedItem类的对象）还提供author、categories、comments、copyright、enclosures、id、images、itunes_author、itunes_duration、itunes_image_link、itunes_summary、link、published、rights、source、summary、tags、time和updated等方法。

对订阅源所有词汇术语进行解释，超出了本书的范围。如果想了解更多，请访问维基百科关于Web订阅源的章节，地址是http://en.wikipedia.org/wiki/Web_feed。也可以找到当前最主流的订阅源网站，因此用Ruby脚本处理新闻可以很轻松地变为现实。

14.2 电子邮件

电子邮件在因特网诞生之前即存在于世，它仍然是最重要和最流行的在线功能之一。本节你将看到怎样检索和管理POP3服务器上的电子邮件，以及怎样用SMTP服务器发送电子邮件。

14.2.1 用POP3协议接收邮件

邮局协议（Post Office Protocol 3，POP3）是从邮件服务器检索电子邮件的最流行协议。如果你的电子邮件程序安装在本机（与Web邮件相反，例如Hotmail或雅虎邮件），可能就会使用POP3协议与邮件服务器沟通，它是通过邮件服务器帮你从外部世界收取邮件的。

通过Ruby可以使用net/pop程序库来完成电子邮件客户端相同的职能，例如预览、检索或删除邮件。如果你有创造的激情，甚至可以用net/pop开发你自己的反垃圾邮件工具。

注 本节中的例子如果不作调整是无法运行的，因为需要对真正的邮件账户进行操作。如果你想运行这些程序，需要把POP3/邮件账号的服务器名、用户名和密码替换成真实的。在理想情况下，如果想尝试这些示例程序，可以创建测试用邮件账号，或先对你的邮件作备份，以防止不可预知的错误。这是因为尽管通过本地邮件程序无法直接删

除邮件，但可能会删除邮件服务器上等待接收的新邮件。直到你对自己的代码和要完成的目标有信心，再修改配置信息，对真实的账号进行操作。

用POP3服务器可以做的基本操作，是连接到服务器，接收某账号包含的邮件信息，查看邮件，删除邮件并关闭连接。首先，我们要连接到POP3服务器，看看那儿是否有可下载的消息，如果有，再看看共有多少：

```
require 'net/pop'

mail_server = Net::POP3.new('mail.mailservernamehere.com')

begin
  mail_server.start('username','password')
  if mail_server.mails.empty?
    puts "No mails"
  else
    puts "#{mail_server.mails.length}mails waiting"
  end
rescue
  puts "Mail error"
end
```

这段代码首先创建一个指向服务器的对象，然后start方法来连接。与邮件服务器连接并操作的整段程序代码，被包含在begin/ensure/end块中，以便让连接错误得以处理，不致于让程序崩溃，或显示莫名奇妙的错误信息。

当start方法连接到POP3服务器，mail_server.mails包含了由Net::POPMail对象组成的数组，这些对象指向服务器上等待的每个消息。可用Array的empty?方法查看是否有邮件，如果有则根据数组的大小，得出有多少邮件正在等待。

可以用Net::POPMail对象的方法来操作和收集已到服务器的邮件。下载所有邮件很简单，只需对每个Net::POPMail对象调用pop方法。

```
mail_server.mails.each do |m|
  mail = m.pop
  puts mail
end
```

随着每个邮件从服务器上被检索（或被弹出，如果你愿意这么说的话），整个邮件的内容，包含邮件头和邮件体文本，都被放到mail变量中，然后再显示在屏幕上。

要删除邮件，可以用delete方法，不过邮件只是被作上标记，等到会话结束时再删除：

```
mail_server.mails.each do |m|
  m.delete if m.pop =~ /\bthis is a spam e-mail\b/i
end
```

这段代码遍历帐号中每个消息，并对包含this is a spam e-mail字符串的邮件作删除标记。

你也可以只检索邮件头，如果你要查找特定主题的邮件，或查找来自特定邮件地址的邮件，

这是一种有用的方法。pop方法返回整个邮件（其大小有时可达数十兆甚至上百兆），而header方法只从服务器返回邮件头。下面的例子展示怎样删除主题包含“medicines”字符串的邮件：

```
mail_server.mails.each do |m|
  m.delete if m.header =~ /Subject:.*?medicines\b/i
end
```

要构建基本的反垃圾邮件过滤器，可以同时并用邮件检索和邮件删除的方法，连接到邮件帐号，在邮件客户端看到邮件之前，删除那些不想要的邮件。考虑一下，如果下载邮件，将其传递给几个正则表达式，然后根据匹配结果删除特定邮件，这样会达到什么样的效果？

14.2.2 用SMTP协议发送邮件

POP3负责处理客户端对邮件的检索、删除和预览操作，而简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）则负责处理发送邮件以及在邮件服务器之间路由邮件。本节不介绍路由的使用，而是介绍一下用SMTP如何实现把邮件发送给某个邮件地址。

net/smtp可以与SMTP服务器直接沟通。在许多UNIX机器上，特别是因特网的服务器上，可以把邮件发送给本机运行的SMTP服务器，该邮件随即通过因特网传送给目标邮箱。在此情况下，发送电子邮件相当简单，如下所示：

```
require 'net/smtp'

message =<<MESSAGE_END
From:Private Person <me@privacy.net>
To:Author of Beginning Ruby <test@rubyinside.com>
Subject:SMTP e-mail test

This is a test e-mail message.
MESSAGE_END

Net::SMTP.start('localhost')do |smtp|
  smtp.send_message message, 'me@privacy.net', 'test@rubyinside.com'
end
```

这段代码用引入文档（here document）把一段简单邮件内容放在message变量中，并仔细编写正确的邮件头（如上代码所示，邮件需要From、To和Subject等头标志，并与邮件正文以空行分开），要发送邮件，可用Net::SMTP连接到本机的SMTP服务器，然后调用send_message方法，附上消息、发送方地址、接收方地址作为参数（尽管发送方和接收方地址已经放在邮件内容里，但仍需要专门提出，供邮件路由使用）。

如果本机没有运行SMTP服务器，可以用Net::SMTP连接到远程SMTP服务器。除非你用的邮件系统是web邮件服务（例如Hotmail或雅虎邮件），否则你的邮件服务商都会为你提供出口邮件服务器详细说明，可用于Net::SMTP，如下所示：

```
Net::SMTP.start('mail.your-domain.com')
```

这行代码连接到mail.your-domain.com这个SMTP服务器的25端口，无须使用任何用户名和密码。但如果你需要，可以指定端口号和其他细节。例如：

```
Net::SMTP.start('mail.your-domain.com',25,'localhost','username','password',➡
:plain)
```

本例通过用户名和密码，以纯文本格式连接到位于mail.your-domain.com的SMTP服务器，并以localhost作为客户的主机名。

注 Net::SMTP也支持LOGIN和CRAM-MD5认证模式。要使用这些认证，请使用:login或:cram_md5作为传递给start的第六个参数。

14.2.3 用ActionMailer发送邮件

ActionMailer

(<http://wiki.rubyonrails.org/rails/pages/ActionMailer>)可以在更高层面发送邮件，而不是直接使用SMTP协议（或net/smtp）。你不再与SMTP服务器直接对话，而是创建ActionMailer::Base的子类，实现一个方法来设置邮件的主题、接收方和其他细节，然后调用该方法发出邮件。

ActionMailer是Ruby on Rails框架的组成部分（详见第13章），但可以单独使用。如果你的计算机上还没有安装Ruby on Rails，可以安装ActionMailer的gem包，命令为gem install actionmailer。

下面是使用ActionMailer的简单示例：

```
require 'rubygems'
require 'action_mailer'

class EMailer < ActionMailer::Base
  def test_email(email_address,email_body)
    recipients(email_address)
    from "me@privacy.net"
    subject "This is a test e-mail"
    body email_body
  end
end

EMailer.deliver_test_email('me@privacy.net','This is a test e-mail!')
```

以上代码从ActionMailer::Base继承，定义了EMailer类。test_email方法利用ActionMailer的辅助方法，对邮件的接收方、发送方地址、主题和正文进行设置，但你永远不必直接调用该方法。要发送邮件，可调用EMailer类名为deliver_test_email的动态类方法（或用deliver_后跟任何内容，形成该类的方法名）。

在上例中，ActionMailer使用默认设置来发出邮件，即试图连接到本机的SMTP服务器。如果本机没有安装并运行SMTP服务器，可以要求ActionMailer寻找其他地方的SMTP服务器，如下所示：

```
ActionMailer::Base.server_settings = {
  :address => "mail.your-domain.com",
```

```

:port =>25,
:authentication =>:login,
:user_name =>"username",
:password =>"password",
}

```

这些设置与Net::SMTP很相似，可以稍作修改，以符合你的配置情况。

14.3 用FTP协议传输文件

文件传输协议（File Transfer Protocol, FTP）是个基本的网络连接协议，用于在任何TCP/IP网络传输文件。尽管Web也可以传输文件，但FTP仍然常用于大型文件的传输，或访问与Web不相干的大型文件仓库。使用FTP的一大好处，在于它内置了认证和访问控制机制。

FTP系统的核心部分是FTP服务器，它是运行在文件服务器上的一个程序，允许FTP客户端在该机器下载和/或上传文件。

在本章上节的“open-uri程序库”段落中，我们考察了怎样使用open-uri程序库从因特网轻松地检索文件。open-uri支持HTTP、HTTPS和FTP的URL，如果想用尽量少的代码从FTP服务器下载文件，open-uri是个理想的选择。示例如下：

```

require 'open-uri'

output =File.new('1.8.2-patch1.gz','w')
open('ftp://ftp.ruby-lang.org/pub/ruby/1.8/1.8.2-patch1.gz')do |f|
  output.print f.read
end
output.close

```

本例从FTP下载文件，并将其内容保存到本地文件。

注 本例在你的机器上可能无法运行，因为你的网络连接可能不支持主动FTP，或许可能需要被动FTP连接。这个问题在本节后文将给出解答。

不过，对于大多数更复杂的操作，net/ftp程序库是理想的选择，因为和net/http对HTTP请求的支持一样，net/ftp提供了底层访问FTP连接的能力。

14.3.1 FTP连接与基本操作

用net/ftp通过FTP的URL连接到FTP服务器是个简单操作：

```

require 'net/ftp'
require 'uri'

uri =URI.parse('ftp://ftp.ruby-lang.org/')

Net::FTP.open(uri.host)do |ftp|
  ftp.login 'anonymous','me@privacy.net'
  ftp.passive =true
  ftp.list(uri.path){|path|puts path }
end

```



```
end
```

```
drwxrwxr-x  2 0 103 6 Sep 10 2005 basecamp
drwxrwxr-x  3 0 103 41 Oct 13 04:53 pub
```

用URI.parse方法解析基本FTP的URL，并用Net::FTP.open方法连接到FTP服务器。一旦连接打开，就可以用ftp对象的login方法指定登录证书信息（与Net::HTTP所用的认证证书非常相似）。再设置连接类型为被动式（这是FTP的一个选项，让防火墙之发出的FTP连接更容易连接成功），然后请求FTP服务器返回URL指定目录（本例是指FTP服务器的根目录）中的文件列表。

Net::FTP提供了login方法，可针对Net::FTP对象使用，如下所示：

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.ruby-lang.org')
ftp.passive = true
ftp.login
ftp.list('*') {|file| puts file }
ftp.close
```

注 如果你打算连接到匿名FTP服务器（只需一般证书即可登录的公共服务器），就不需要用login方法指定任何证书。这正是上例的情形。

本例展示了用Net::FTP连接到FTP服务器的完全不同方法。和Net::HTTP和File类一样，可以在代码块结构中使用Net::FTP，或通过引用对象（在此情况下是ftp对象），手工打开和关闭连接。

由于未提供用户名和密码，login方法对ftp.ruby-lang.org进行了匿名连接。请注意，本例根据主机名连接到FTP服务器的。不过，如果需要用户名和口令，可改用下面的代码：

```
ftp.login(username, password)
```

连接成功后，即可用ftp对象的list方法，获取服务器当前目录的所有文件列表。因为未指定要变更到哪个目录，因此当前目录就是FTP服务器的默认目录。如果要变更目录，可用chdir方法：

```
ftp.chdir('pub')
```

也可以变更目录到远程文件系统的任何位置：

```
ftp.chdir('/pub/ruby')
```

如果你有这么做的权限（根据你在FTP服务器的帐号而定），也可以创建目录。通过mkdir来实现这一操作：

```
ftp.mkdir('test')
```

在没有合适权限的FTP服务器上执行该操作将导致异常，因此最好把这种不稳定的动作放在代码块中，以捕捉可能抛出的异常。

同样，可以删除文件，也可以修改文件名：

```
ftp.rename(filename, new_name)
ftp.delete(filename)
```

这些操作只在有合适权限的前提下才能使用。

14.3.2 下载文件

从FTP服务器下载文件很简单，如果你知道自己要下载的文件名和文件类型的话。Net::FTP提供了两种有用的方法来下载文件，分别是getbinaryfile和gettextfile。由于纯文本文件和二进制文件（例如图像、音频或程序）的传送方式不一样，因此下载时必须选用正确的方法。在大多数情况下，你会提前注意到需要哪种方法。下面的例子展示了怎样从Ruby官方FTP服务器下载二进制文件：

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.ruby-lang.org')
ftp.passive = true
ftp.login
ftp.chdir('/pub/ruby/1.8')
ftp.getbinaryfile('1.8.2-patch1.gz')
ftp.close
```

getbinaryfile接受的几个参数，只有一个是必须的。第一个参数是远程文件的名称（本例是1.8.2-patch1.gz），可选的第二个参数是写入本地文件的名称，可选的第三个参数是文件块的大小，它是指定每次按多大（字节）块下载文件。如果你忽略第二个参数，则下载的文件将写到本地目录的同名文件中，如果你想把远程文件写到特定的本地路径，则可以指定这个参数。

这样使用getbinaryfile有一个问题，它会把程序锁定，直到下载完毕。不过，如果为getbinaryfile提供一个代码块，则每次下载的数据将传送给代码块，并保存到文件：

```
ftp.getbinaryfile('stable-snapshot.tar.gz', 'local-filename', 102400) do |blk|
  puts "A 100KB block of the file has been downloaded"
end
```

每当下载了文件的另一块100KB之后，这段代码都把一段字符串打印输出到屏幕。你可以用这种方法来向用户提供更新信息，而不是让他们傻傻等待，却不知道文件到底下载好了没有。

也可以像这样分块下载文件，并在代码块中即时处理下载的数据，如下所示：

```
ftp.getbinaryfile('stable-snapshot.tar.gz', 'local-filename', 102400) do |blk|
  ..do something with blk here ..
end
```

这样每次下载的100KB文件块都被传递到代码块中进行处理。不幸的是，文件仍然会保存到本地文件中，这不是我们想要的结果，可以通过Tempfile（详见第9章）使用临时文件，然后立即将其删除。

下载文本或ASCII码文件的方法与上述代码相同，但需要改用gettextfile方法。唯一的区别在于，gettextfile不接受第三个块大小参数，而是一行行地把数据返回给代码块。

14.3.3 上传文件

向FTP服务器上传文件，仅当你对服务器中你要上传的目录拥有写权限，才可能实现。因此，本节的所有示例代码未经修改无法运行，因为你无法提供具有写权限的FTP服务器（原因很明显！）。

上传与下载正好相反，net/ftp程序库提供了putbinaryfile和puttextfile方法，它们接受的参数与getbinaryfile和gettextfile完全相同。第一个参数是要上传的本地文件名，可选的第二个参数是远程服务器上的文件名（如果省略此参数，默认与上传文件名相同），可选的第三个参数仅用于putbinaryfile方法，是用来上传的文件块的。下面是个上传的例子：

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.domain.com')
ftp.passive = true
ftp.login
ftp.chdir('/your/folder/name/here')
ftp.putbinaryfile('local_file')
ftp.close
```

与getbinaryfile和gettextfile一样，如果提供了代码块，上传的文件块将传递给代码块，可用于告知用户上传过程的进展。

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.domain.com')
ftp.passive = true
ftp.login
ftp.chdir('/your/folder/name/here')

count = 0

ftp.putbinaryfile('local_file', 'local_file', 100000) do |block|
  count += 100000
  puts "#{count}bytes uploaded"
end

ftp.close
```

如果要上传刚刚由Ruby脚本生成、还未放在文件中的数据，就需要用Tempfile创建临时文件，并用它来上传。例如：

```
require 'net/ftp'
require 'tempfile'

tempfile = Tempfile.new('test')

my_data = "This is some text data I want to upload via FTP."
tempfile.puts my_data
```

```

ftp =Net::FTP.new('ftp.domain.com')
ftp.passive =true
ftp.login
ftp.chdir('/your/folder/name/here')

ftp.puttextfile(tempfile.path,'my_data')

ftp.close
tempfile.close

```

14.4 小结

本章我们考察了Ruby对各种因特网系统和协议的支持，Ruby怎样操作Web，怎样处理和操作从因特网获取的数据。

我们来回顾一下本章涵盖的主要概念：

- HTTP：是超文本传输协议（HyperText Transfer Protocol）的缩写，它定义了Web浏览器与Web服务器通过因特网之类的网络相互沟通的方式。
- HTTPS：是HTTP的安全版本，确保数据在双向传递时只能在两端可读，任何拦截HTTS流的人都无法破译。该协议常常用于电子商务，以及在Web上传输财务数据。
- HTML：是超文本标记语言（HyperText Markup Language）的缩写，它是用于表示网页文本格式化和布局的语言。
- WEBrick：是一套HTTP服务器工具包，由Ruby默认自带。WEBrick可以快速简单地组成基本的Web服务器。
- Mongrel：是另一种HTTP服务器程序库，由Zed Shaw开发，以gem包形式提供。它比WEBrick运行更快，更可伸缩扩展。
- Markaby：是一套Ruby程序库，可用于从Ruby方法和代码逻辑中直接生成HTML。
- RedCloth：是Textile标记语言的Ruby实现。Textile语言可方便地根据特殊格式的纯文本生成HTML文档。
- Hpricot：是自夸为“快速和怡人的”HTML解析器，用于在Ruby中轻松处理和解析HTML。
- POP3：是邮局协议（Post Office Protocol 3）的缩写，是检索电子邮件的常用邮件服务器协议。
- SMTP：是简单邮件传输协议（Simple Mail Transfer Protocol）的缩写，是向邮件服务器、或在邮件服务器之间传输邮件的常用邮件服务器协议。SMTP用于发送邮件，而不是接收邮件。
- FTP：是文件传输协议（File Transfer Protocol）的缩写，它是一套因特网协议，提供对服务器上文件的访问，并允许用户下载或上传。

本章我们讨论了各种因特网相关功能，在第15章我们将介绍网络连接、服务器和网络服务的更深入主题。第15章介绍的大部分内容也适用于因特网，但与FTP或Web的运用相比，处于更低层的位置。

第15章 网络连接、套接字与后台进程

本章我们将考察怎样使用Ruby来执行网络相关操作，怎样创建服务器和网络服务，以及怎样创建可以响应网络查询的持久化进程。

第14章从较高层面讨论了Ruby的因特网能力，例如处理对Web网站的查询，处理HTML，解析XML，以及通过FTP管理文件。与之相反，本章从较低层面考察网络连接和网络服务，并动手创建自己的基本协议和永远运行的服务进程。

我们首先来看一下本章要用到的网络连接的基本概念。

15.1 网络连接的概念

网络是一组以某种方式连接在一起的计算机。如果你家里有几台计算机，共用一台有线或无线路由器，这就叫局域网（local area network, LAN）。你的计算机可能还连接到因特网，或其他形式的网络。网络连接是指两台或多台计算机或设备之间进行通讯的总体概念，本章考察怎样用Ruby来进行网络相关操作，不管是在局域网还是在因特网。

注 如果你对网络和TCP、UDP以及IP协议有经验，可以跳过本节内容。

15.1.1 TCP和UDP协议

网络的类型有很多，但我们最感兴趣的网络类型是TCP/IP网络。TCP/IP是传输控制协议（Transmission Control Protocol, TCP）和网际协议（Internet Protocol, IP）这两种协议的合称。TCP定义了计算机相互连接的概念，确保机器可以成功地按正确顺序发送与接收数据包（packet），而IP则是关于数据在各机器之间实际路由的协议。IP是大多数局域网和因特网的底层基础，而TCP是确保连接可靠的上层协议。

用户数据报协议（User Datagram Protocol, UDP）是与TCP类似的另一种协议，但与TCP不同之处在于，它不考虑可靠性，不保证远程机器收到所发送的数据。当用UDP发送数据时，只得希望它能到达目的地，因为收不到任何成功或失败的确认。尽管如此，因为UDP速度快开销低，仍然被用于各种非关键任务。

一般情况下，两台机器之间需要持久连接（不管是长时间还是短时间）的操作，都使用TCP和基于TCP的协议。例如，几乎所有需要认证的服务，如电子邮件访问，都使用基于TCP的协议，这样认证信息只发送一次——在连接开始时——然后连接的双方都认同该连接已通过认证。

对于连接不很重要或很容易重建连接的操作，例如把域名和主机名转换成IP地址以及逆向操作，可以在UDP上运行。如果在指定时间内没有收到对询问的答复，只要再发送另一个询问即可。因为UDP的低开销和低延迟，有时也被用于视频流和音频流的传输。

15.1.2 IP地址和DNS

连接到基于IP网络的每台机器，都有一个或多个唯一IP地址。当数据在网络中发送给特定IP地址时，具有该地址的机器将接收到数据。

当使用万维网访问诸如<http://www.apress.com>之类的Web网站时，计算机首先询问域名服务（Domain Name Service，简称DNS）服务器，查询与主机名www.apress.com相关的IP地址。一旦得到原始地址的反馈（对于本例是65.19.150.101），Web浏览器即在80端口建立与该机器的连接。计算机可以在不同的TCP（或UDP）端口（取值范围为0—65536）建立和接收连接，而不同端口被赋给不同类型的服务。例如，80端口是Web服务器的默认端口。

本章下文将考察怎样通过基于IP的网络进行某些操作，例如检查网络中机器是否可用，我们还将创建基本的TCP和UDP客户端和服务端。

15.2 网络操作基础

网络编程通常是困难的，它涉及最底层的许多神秘术语，并需要与古老的程序库进行接口。不过，Ruby并非寻常，Ruby的程序库拿掉了与网络编程有关的大部分复杂性。

本节我们将考察怎样实现一些基本的网络连接操作，例如检查服务器是否在网络上，查看数据怎样在网络的端点之间路由，以及怎样直接连接到远程机器提供的服务。

15.2.1 检查机器和服务是否可用

最基本的网络操作之一是执行ping，这是对另一台机器是否在网络中或其服务是否可用的简单检查。

Ruby标准程序库包含ping，它是检查机器网络可用性的基本程序库：

```
require 'ping'
puts "Pong!" if Ping.pingecho('localhost', 5)
```

Pong!

ping程序库非常基础，只提供一个Ping类，以及一个类方法pingecho。第一个参数是待检查机器的主机名、域名或IP地址，第二个参数是等待响应的最大秒数。由于ping操作可能要花几秒钟才能得到响应，因此可以考虑使用Ruby的线程支持，来同时执行多个ping，如果要执行许多ping操作的话。

在许多系统中，上述程序将返回“Pong!”的响应，因为pingecho可以对本机进行ping操作。在有些系统中，可能需要把'localhost'改为'127.0.0.1'，这永远是本机解析的默认IP地址。不过，你会发现没有任何响应。在检查原因之前，我们来看另一个例子：

我们用ping程序库来检查服务器是否在线：

```
require 'ping'

puts "Pong!" if Ping.pingecho('www.google.com', 5)
```

在本书编写时，这个ping操作得不到任何响应，在5秒钟之后程序默默退出。而如果用命令行工具对www.google.com进行ping操作，就会得到响应。

原因在于ping程序库只执行TCP的ping回声操作，而不是ICMP（Internet Control Message Protocol，网际控制消息协议）的回声操作。TCP回声很少用，被许多机器阻止（或被某些网络整个阻止），特别是因特网上的机器。尽管TCP回声操作可以在局域网中正常使用，但很少得到在线支持。因此，你需要寻找其他方法。

另一种可用的ping程序库名为net-ping，它以gem包形式提供，用gem install net-ping命令安装。net-ping也支持TCP回声操作，但也可以用更可靠的技术与操作系统的ping命令接口，从而得到响应。它也支持直接连接到远程机器提供的服务，来测试它是否可以响应请求。

```
require 'rubygems'
require 'net/ping'

if Net::PingExternal.new('www.google.com').ping
  puts "Pong!"
else
  puts "No response"
end
```

Pong!

不过，如果你想检查某个特定服务是否可用，与其检查整个机器是否可用，不如用net-ping连接到TCP或UDP的特定端口：

```
require 'rubygems'
require 'net/ping'

if Net::PingTCP.new('www.google.com',80).ping
  puts "Pong!"
else
  puts "No response"
end
```

本例就是Web浏览器直接连接到www.google.com的HTTP端口，一旦得到连接就立即关闭连接。这样就可以验证到www.google.com可以接收HTTP连接。

15.2.2 进行DNS查询

大多数Ruby网络连接程序库允许在与远程服务器交互时，可以指定域名和主机名，并自动将这些名字解析（resolve）为IP地址。不过，这样做增加了小小的开销，因此在某些情况下可能希望自己提前解析IP地址。

也可以用DNS查询来检查不同主机名是否存在，或域名是否激活，甚至是否指向Web服务器。

resolv是Ruby标准程序库中的一个程序库，提供几种有用方法，可对主机名和IP地址进行相互转换：

```
require 'resolv'
puts Resolv.getaddress("www.google.com")
```

66.102.9.104

这段代码对Google网站返回66.102.9.104的IP地址。不过，如果把同样的代码运行几次，可能得到不同的反馈。这时因为像Google这种大型网站，通常把请求分发到多个Web服务器，以便提高响应速度。如果想得到与主机名关联的所有地址，可以改用each_address方法：

```
require 'resolv'

Resolv.each_address("www.google.com") do |ip|
  puts ip
end
```

```
66.102.9.104
66.102.9.99
66.102.9.147
```

你也可以用getname方法把IP地址转换成主机名：

```
require 'resolv'

ip = "192.0.34.166"
begin
  puts Resolv.getname(ip)
rescue
  puts "No hostname associated with #{ip}"
end
```

```
www.example.com
```

必须注意的是，不是所有IP地址都可以反向解析为主机名，因为这是DNS系统的可选需求。

除了可以在IP地址和主机名之间相互转换，resolv还可以从DNS服务器检索其他信息，例如与特定主机或域名关联的邮件服务器。“哪个IP地址与哪个主机名关联”的记录被称为A记录，而“哪台邮件服务器与主机名关联”被称为MX记录。

在上述示例中，用到了Resolv类提供的特殊辅助方法，而搜索MX记录需要直接用到Resolv::DNS类，因此你可以把需要搜索的不同类型的记录放在额外选项中：

```
require 'resolv'

Resolv::DNS.open do |dns|
  mail_servers = dns.getresources("google.com", Resolv::DNS::Resource::IN::MX)
  mail_servers.each do |server|
    puts "#{server.exchange.to_s}-#{server.preference}"
  end
end
```

```
smtp2.google.com -10
smtp3.google.com -10
smtp4.google.com -10
smtp1.google.com -10
```

本例直接使用Resolv::DNS，以更详细的方式执行了DNS请求，而不是用方便的Resolv。

`getname`和`Resolv.getaddress`辅助方法，这样就可以用`Resolv::DNS::Resource::IN::MX`选项指定MX请求。

注 具备DNS术语常识的读者，可能想在上述选项中使用CNAME、A、SOA、PTR、NS和TXT等变体，该程序库对这些选项都支持。

如果想发送电子邮件，但又没有任何用来发送邮件的SMTP服务器，MX记录就很用，因为你可以直接使用`Net::SMTP`（详见第14章），把邮件发送给目的域名邮件服务器。例如，如果你想给某人发送电子邮件，他的邮件地址末尾是`@google.com`，则可以用`Net::SMTP`直接连接到`smtp2.google.com`（或其他任何可选邮件服务器地址），并直接把邮件发送给此人。

```
require 'resolv'
require 'net/smtp'

from = "your-email@example.com"
to = "another-email@example.com"

message = <<MESSAGE_END
From:#{from}
To:#{to}
Subject:Direct e-mail test

This is a test e-mail message.
MESSAGE_END

to_domain = to.match(/\@(.+)/)[1]

Resolv::DNS.open do |dns|
  mail_servers = dns.getresources(to_domain, Resolv::DNS::Resource::IN::MX)
  mail_server = mail_servers[rand(mail_servers.size)].exchange.to_s

  Net::SMTP.start(mail_server) do |smtp|
    smtp.send_message message, from, to
  end
end
```

注 如想了解DNS的更多内容，请访问http://en.wikipedia.org/wiki/Domain_Name_System。

15.2.3 直接连接到TCP服务器

最重要的网络连接操作之一，是连接到另一台机器（或在某种情况下，甚至是本机！）提供的服务，并以某种方式与其交互。在第14章我们考察了一些高层次的方法，例如通过Ruby程序库，更简便地使用Web或FTP进行这些操作。

不过，也可以在TCP层面直接连接到远程服务，并以原始格式与其通讯。这种方法可以有效

地检查不同协议的工作机制（因为你将需要用到并理解协议的原始数据），或用于创建自己的简单协议。

要直接连接到TCP端口，可以使用名为Telnet的工具程序。Telnet是一套协议，提供了通用、双向、8比特位、面向字节的通讯能力。其名字来源于“telecommunication network（远程通讯网络）”，我们只关心其连接到原始TCP端口的便捷能力即可。正如你所期望的那样，Ruby标准程序库自带了一个Telnet程序库：net/telnet。

我们来用net/telnet连接到Web站点，并用直接HTTP协议检索网页：

```
require 'net/telnet'

server = Net::Telnet::new('Host'=>'www.rubyinside.com',
                          'Port'=>80,
                          'Telnetmode'=>false)

server.cmd("GET /HTTP/1.1 \nHost:www.rubyinside.com \n")do |response|
  puts response
end
```

```
HTTP/1.1 200 OK
Date:Wed,01 Nov 2006 03:46:11 GMT
Server:Apache
X-Powered-By:PHP/4.3.11
X-Pingback:http://www.rubyinside.com/xmlrpc.php
Status:200 OK
Transfer-Encoding:chunked
Content-Type:text/html;charset=UTF-8
```

..hundreds of lines of HTML source code for the page removed ..

注 几秒钟之后，可能会发生了超时错误。这是因为你不知道数据何时能从Web服务器接收完毕。通常情况下，如果没有更多数据进来，就可以在超时发生时关闭连接。这也是使用合适的HTTP程序库的原因之一，因为它帮你处理了所有这一切！

Net::Telnet连接到www.rubyinside.com的80端口（标准HTTP端口），并发出以下命令：

```
GET / HTTP/1.1
Host: www.rubyinside.com
```

这些命令是HTTP协议的组成部分，它们告诉远程Web服务器，请返回www.rubyinside.com的主页。反馈的内容随即被打印输出到屏幕，其中前8行（或左右）是HTTP头，这也是HTTP协议的另一组成部分。

当使用open-uri和Net::HTTP程序库时（如第14章所示），这些方法都被屏蔽了，因为程序库创建了正确的HTTP命令，并处理HTTP反馈。不过，如果需要创建自己的程序库来处理新的或Ruby目前还不支持的协议，就可以需要用到Net::Telnet或类似的程序库，以便直接访问TCP原始数据。

15.3 服务器和客户端

客户端和服务端是使用网络的两种主要类型的软件。客户端连接到服务器，而服务器负责处理信息，并管理来自客户端的连接和数据，以及将数据返回给客户端。本节将带你创建服务器，还可用本章和第14章介绍的net/telnet及其他客户端程序库来连接到这些服务器。

15.3.1 UDP客户端和服务端

在上节中，我们考察了怎样用net/telnet创建基本的TCP客户端。但为了演示基本的客户端/服务器系统，UDP是个理想的开始。与TCP不同，UDP没有连接的概念，因此用于简单系统中，消息在一处传递到另一处，并不保证一定到达。TCP像是打电话，UDP像是通过邮局寄明信片。

创建UDP服务器非常简单，我们来创建一段脚本，取名为udpserver.rb:

```
require 'socket'

s = UDPSocket.new
s.bind(nil, 1234)

5.times do
  text, sender = s.recvfrom(16)
  puts text
end
```

这段代码使用了Ruby的socket（套接字）程序库，它提供了对操作系统最底层网络功能的访问。套接字对UDP很适用，本例中你创建了新的UDP套接字，并将其绑定到本机的1234端口。然后循环五次，从套接字接收16字节组成的数据块，并打印输出到屏幕。

注 只循环五次的原因，是让脚本可以在接收五次简短消息之后优雅地结束。后文我们将看到，怎样让服务器永远运行。

有了服务器，现在需要一个客户端，来向服务器发送数据。我们来创建udpclient.rb:

```
require 'socket'

s = UDPSocket.new
s.send("hello", 0, 'localhost', 1234)
```

这段代码创建了一个UDP套接字，但不是用于监听数据，而是把字符串"hello"发送给localhost（本机）1234端口的UDP服务器。如果与udpclient.rb同时运行udpserver.rb，则当udpserver.rb运行时，“hello”将显示在屏幕上。这样就成功地通过网络（尽管是在同一台机器上）用UDP从客户端向服务器发送了数据。

当然，如果你有多台机器，也可以在不同机器上分别运行客户端和服务端，只需把send方法中的'localhost'改为udpserver.rb运行所在机器的主机名或IP地址即可。

如你所见，UDP很简单，但可以在它上面层叠更高级的功能。例如，因为没有涉及到连接，你可以用一个程序在客户端和服务端模式间切换，实现双向效果。这样做很容易，只需编制一个程序，实现向自身发送并从自身接收UDP数据即可：

```
require 'socket'

host = 'localhost'
port = 1234

s = UDPSocket.new
s.bind(nil, port)
s.send("1", 0, host, port)

5.times do
  text, sender = s.recvfrom(16)
  remote_host = sender[3]

  puts "#{remote_host} sent #{text}"

  response = (text.to_i * 2).to_s
  puts "We will respond with #{response}"

  s.send(response, 0, host, port)
end
```

```
127.0.0.1 sent 1
We will respond with 2
127.0.0.1 sent 2
We will respond with 4
127.0.0.1 sent 4
We will respond with 8
127.0.0.1 sent 8
We will respond with 16
127.0.0.1 sent 16
We will respond with 32
```

注 在实际应用中，一般会用到两个脚本，它们分别运行在不同机器上，并相互之间进行通信。但本例为了便于测试，在一台机器上展示了实现这一效果所必须的代码逻辑。

UDP在速度和所需资源方面有一些优势，但因为它缺少连接状态和数据传输的可靠性，一般更多使用TCP。下一节我们将考察怎样创建简单的TCP服务器，你可用net/http和其他应用程序来连接到这些服务器。

15.3.2 构建简单的TCP服务器

TCP服务器是大多数因特网服务的基础。尽管轻量级的时间服务器和DNS服务器可用UDP进行工作，但在发送网页和电子邮件时，必须与远程服务器建立连接，发送请求，并发送和接收数据。本节将带你构建一个可以响应Telnet请求的基本的TCP服务器，然后我们再学习怎样创建更复杂的服务器。

我们来看一个基本的服务器，它在1234端口运行，接收连接并把客户端发送来的任何文本打印输出到屏幕上，以及向客户端发回确认信息：

```
require 'socket'

server =TCPServer.new(1234)

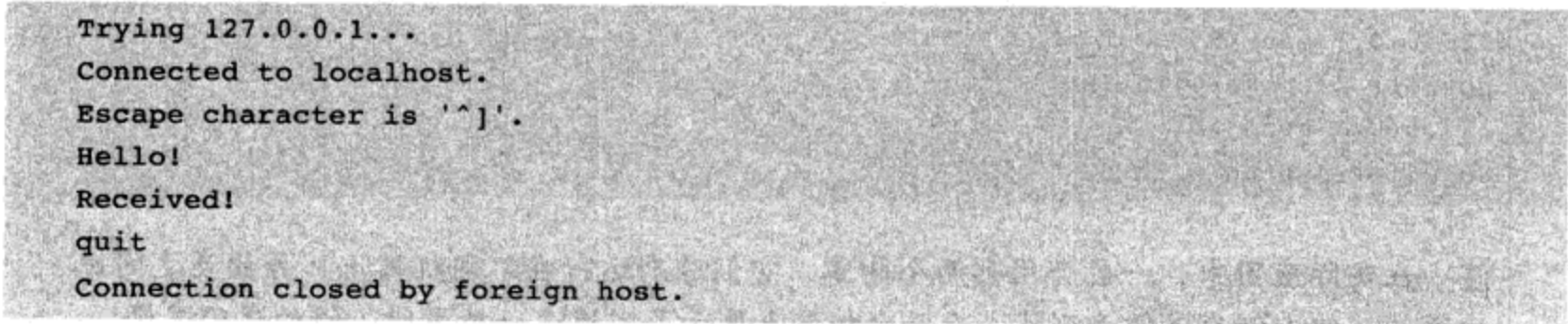
while connection =server.accept
  while line =connection.gets
    break if line =~/quit/
    puts line
    connection.puts "Received!"
  end

  connection.puts "Closing the connection.Bye!"
  connection.close
end
```

和用UDP创建客户端和服务端一样，创建TCP服务器和客户端也用到套接字。本例在本机的1234端口创建一个TCPServer对象，然后进入循环，在接受来自客户端的新连接时，调用TCPServer对象的accept方法。一旦连接建立，服务器即接收一行行地输入信息，且仅在某行包含单词quit时才关闭连接。

要测试这个客户端，可以用操作系统的telnet客户端程序（OS X、Linux和Windows均自带该程序，可在命令行以telnet调用），如下所示：

```
telnet 127.0.0.1 1234
```



```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello!
Received!
quit
Connection closed by foreign host.
```

另外，你也可以用net/telnet创建自己的基本客户端：

```
require 'net/telnet'

server =Net::Telnet::new('Host'=>'127.0.0.1',
                        'Port'=>1234,
                        'Telnetmode'=>false)

lines_to_send =['Hello!','This is a test','quit']

lines_to_send.each do |line|
  server.puts(line)

  server.waitFor(/./)do |data|
```

```

    puts data
  end
end

```

和UDP客户端与服务器示例一样，这个客户端和服务程序可以（而且通常会）放在两台机器上。即使服务器放在地球的另一端，客户端在本机上运行，这些测试程序的运行结果没有任何区别，只要它们都连接在因特网上。

不过，这个TCP服务器的一大缺陷是它一次只接受一个连接。在用telnet连接它并开始输入时，如果有另一个连接试图进入，则在它开始连接时，收不到任何响应信息。发生这种情况的原因，是你的TCP服务器在当前状况下，一次只能接受一个连接。下一节我们将考察怎样创建可以同时处理多个客户端的更高级服务器。

15.3.3 多客户端TCP服务器

因特网上的大多数服务器都能同时处理大量客户端请求，一次只能提供一个文件的Web服务器很快就会成为世界上最慢的Web网站，因为用户会排成长队等待服务！上节的TCP服务器就是这样，一般被称为“单线程”或“排队式”服务器。

用Ruby的Thread类，可以很容易地创建多线程的服务器——即在接受请求时立即创建新执行线程来处理这个连接，同时允许主程序等待更多连接的服务器。

```

require 'socket'

server =TCPServer.new(1234)

loop do
  Thread.start(server.accept)do |connection|
    while line =connection.gets
      break if line =~/quit/
      puts line
      connection.puts "Received!"
    end

    connection.puts "Closing the connection.Bye!"
    connection.close
  end
end

```

本例中有个永久循环，当server.accept响应时，即创建一个新线程，并立即启动该线程来处理刚刚接受的连接（即传递给该线程的connection对象）。而主程序可以立即继续循环，等待新的连接请求。

这样使用Ruby线程，让程序代码容易移植，其在Linux、OS X和Windows上的运行方式完全相同。不过，线程化也不是没有缺点。Ruby的线程不是真正操作系统级别的线程，在程序等待系统提供数据时会发生锁死现象。另外在为每个连接创建新线程并令其执行时，也有一定的开销。

在兼容POSIX的操作系统中（例如OS X和Linux，但不包含Windows），可以对程序进行fork分支操作，这样就创建了一个单独的进程，而不是单独的线程。不过，与其在接受连接时创建

分支进程，不如预先创建一批监听进程，以增加可同时处理的最大连接数：

```
require 'socket'

server =TCPServer.new(1234)

5.times do
  fork do
    while connection =server.accept
      while line =connection.gets
        break if line =~/quit/
        puts line
        connection.puts "Received!"
      end

      connection.puts "Closing the connection.Bye!"
      connection.close
    end
  end
end
```

注 对于不支持POSIX式分支操作的操作系统，例如Windows，这段代码无法运行。不过，用Ruby线程的服务器可以在Ruby支持的所有操作系统上运行。

本例分出5个独立进程，每个进程按顺序分别接受一个连接，这样就允许5个客户端同时连接到服务器。每个进程均独立于主进程，因此即使主进程在分支操作完成后立即终止，客户端进程仍然可以继续运行。

注 因为分支出来的进程可以继续运行，因此要关闭它们，需要用`killall ruby`等命令（适用于Linux和OS X）杀死进程。

尽管使用分支操作，可以获得并行运行多个相同服务器的能力，但管理子进程是件非常麻烦的事。必须手工杀死进程，而且如果某个进程死亡或运行出错，也无法用新服务器进程来取代它。这意味着如果因为某种原因所有子进程都死亡，将残留一个无法运行的服务！

虽然可以自己编写代码逻辑，处理所有这些子进程维护所必须的管理功能，但Ruby自带了一个名为GServer的程序库，为我们省了许多事，而且它能在所有平台上运行。

15.3.4 GServer程序库

GServer是Ruby标准程序库中的一个程序库，它实现了“通用服务器”的功能，提供了线程池管理、日志、以及同时管理多个服务器的工具。GServer以类的形式提供，你可以从它继承得到服务器类。

除了简单管理，GServer还提供了在不同端口同时运行多个服务器的功能，只需几行代码，整套服务即可整装待发。线程管理完全被GServer接管，当然如果你愿意，也可以参与这一过程。GServer还实现了日志功能，同样，如果你愿意，也可以用你自己的代码来实现该功能。

我们来看用GServer可以实现的最简单TCP服务器：

```
require 'gserver'

class HelloServer <GServer
  def serve(io)
    io.puts("Hello!")
  end
end

server =HelloServer.new(1234)
server.start
server.join
```

这段代码实现了一个基本的服务器，其功能只是对连接到1234端口的任何客户端输出“Hello!”一词。如果你用telnet连接到1234端口（甚至用Web浏览器也可以，使用http://127.0.0.1:1234/地址），就会看到在连接关闭之前，返回了字符串“Hello!”。

本例通过从GServer继承，创建了一个名为HelloServer的服务器类。GServer实现了线程管理和连接管理的全部功能，让你无须操心。在这个简单的示例中，你只是创建了一个服务器进程，告诉它使用1234端口，并立即启动该进程。

不过，即使这么简单的示例，也可以在多客户端的情况下运行，如果用telnet多次同时访问它，你会发现所有请求都被成功处理了。但也可以为new方法提供更多参数，设置最大允许连接数：

```
require 'gserver'

class HelloServer <GServer
  def serve(io)
    io.puts("Say something to me:")
    line =io.gets
    io.puts("You said '#{line.chomp}'")
  end
end

server =HelloServer.new(1234,'127.0.0.1',1)
server.start
server.join
```

GServer的new方法接受几个参数，按顺序分别是：服务器运行的端口号、服务器运行的主机名或接口名、允许同时处理的最大连接数（本例将其设置为1）、用来发送日志消息的文件句柄，以及将日志功能打开/关闭的true/false标志。

如前文所述，可以同时创建多个服务器：

```
require 'gserver'

class HelloServer <GServer
  def serve(io)
    io.puts("Say something to me:")
    line =io.gets
```

```

        io.puts("You said '#{line.chomp}'")
      end
    end

    server =HelloServer.new(1234,'127.0.0.1',1)
    server.start

    server2 =HelloServer.new(1235,'127.0.0.1',1)
    server2.start
    sleep 10

```

创建多个服务器和创建HelloServer（或任何继承自GServer的类）的新实例一样简单，将其赋值给变量，并调用其start方法即可。

本例与前例的另一个区别，是没有调用server.join方法。对于GServer对象，join方法的使用方式与Thread对象相同，后者用join方法等待线程完成职能，再继续执行下去。在第一个GServer示例中，你的程序将永远等待下去，直到手工退出（例如用Ctrl+C）。但在上例中，你没有调用任何join方法，只用sleep 10延时了10秒钟。这表示你创建的服务器在运行后只对1234和1235端口开放10秒钟，然后程序及其创建的子线程都同时退出。

由于GServer允许多个服务器同时运行而不会阻止主程序的运行，因此可以对当前运行的服务器进行管理，即使用GServer提供的方法来启动、停止和检查服务器：

```

require 'gserver'

class HelloServer <GServer
  def serve(io)
    io.puts("To stop this server,type 'shutdown'")
    self.stop if io.gets =~/shutdown/
  end
end

server =HelloServer.new(1234)
server.start

loop do
  break if server.stopped?
end

puts "Server has been terminated"

```

这一次把主程序放在等待服务器停止的循环中，如果某人连接到服务器并输入shutdown，则触发服务器的stop方法，导致整个服务器程序终止。

还可以用in_service?类方法，无须拥有对象引用，即可检查GServer是否在某个端口运行：

```

if GServer.in_service?(1234)
  puts "Can't create new server. Already running!"
else
  server = HelloServer.new(1234)
end

```

15.3.5 基于GServer的聊天服务器

有了上节的知识，要用GServer构建实用的应用程序，只须在复杂性上小小地前进一步。本节你将构建一个简单的聊天服务器，允许许多客户端连接并可以进来互相聊天。

第一步是从GServer继承出一个新类：ChatServer，并用自己的代码覆写其初始化方法，以便设置类变量，为所有客户共享保存客户ID和聊天记录：

```
class ChatServer <GServer
  def initialize(*args)
    super(*args)

    #Keep an overall record of the client IDs allocated
    #and the lines of chat
    @@client_id = 0
    @@chat = []
  end
end
```

程序的主体部分与其他基于GServer的应用程序很相似，有一个基本的初始化程序和一个循环，仅在服务器关闭自身时退出循环：

```
server = ChatServer.new(1234)
server.start

loop do
  break if server.stopped?
end
```

注 请记住，你可以指定取得服务的主机名，作为ChatServer.new的第二个参数。如果你想在因特网上使用这个聊天服务器，可能需要指定远程可访问的IP地址作为第二个参数，否则你的服务器可能只对局域网机器有用。

现在基本功能已经就位，需要创建serve方法，其功能是把下一个可用客户ID（用类变量@@client_id）分配给连接，向用户发送欢迎信息，接受用户输入的文本行，并随时向其显示其他用户输入的最新文本行。

本例serve方法的代码特别长，下面显示聊天服务器的完整源代码，包括注释：

```
require 'gserver'

class ChatServer <GServer
  def initialize(*args)
    super(*args)

    #Keep an overall record of the client IDs allocated
    #and the lines of chat
    @@client_id = 0
    @@chat = []
  end
```

```
def serve(io)
  #Increment the client ID so each client gets a unique ID
  @@client_id +=1
  my_client_id =@@client_id
  my_position =@@chat.size

  io.puts("Welcome to the chat,client #{@@client_id}!")

  #Leave a message on the chat queue to signify this client
  #has joined the chat
  @@chat <<[my_client_id,"<joins the chat>"]

  loop do
    #Every 5 seconds check to see if we are receiving any data
    if IO.select([io ],nil,nil,2)
      #If so,retrieve the data and process it...
      line =io.gets

      #If the user says 'quit',disconnect them
      if line =~/quit/
        @@chat <<[my_client_id,"<leaves the chat>"]
        break
      end

      #Shut down the server if we hear 'shutdown'
      self.stop if line =~/shutdown/

      #Add the client's text to the chat array along with the
      #client's ID
      @@chat <<[my_client_id,line ]
    else
      #No data,so print any new lines from the chat stream
      @@chat [my_position..(@@chat.size -1)].each_with_index do |line,index|
        io.puts("#{line [0]}says:#{line [1]}")
      end

      #Move the position to one past the end of the array
      my_position =@@chat.size
    end
  end

end

end

server =ChatServer.new(1234)
server.start

loop do
```

```
break if server.stopped?
end
```

该聊天服务器的运行主要依靠一个简单的循环，用下面这行代码持续检查是否有数据正在等待接收：

```
if IO.select([io], nil, nil, 2)
```

`IO.select`是个特殊的方法，可以（按照接收缓冲区、发送缓冲区、异常/错误缓冲区的顺序）检查I/O流的各个缓冲区是否有数据。如果客户连接有还未处理的数据，则`IO.select([io], nil, nil, 2)`返回值，但对是否存在发送数据或错误数据则予以忽略。最后一个参数2，指定了两秒钟的超时限制，因此在处理成功或失败之前将等待两秒钟。这表示程序每过两秒钟就执行else块，把聊天记录中任何新消息发送给客户。

如果用telnet连接到该聊天服务器，连接会话将如下所示：

```
$telnet 127.0.0.1 1234

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome to the chat,client 1!
1 says:<joins the chat>
2 says:<joins the chat>
Hello 2!
1 says:Hello 2!
2 says:Hello 1!
2 says:I'm going now..bye!
2 says:<leaves the chat>
quit
Connection closed by foreign host.
```

通过本节和上节介绍的GServer原理，你可以创建遵循自定协议的服务器，甚至创建可以响应古老协议的服务器。所有这些，只需有接收数据、处理数据、把数据发回给客户端的能力。使用这些方法，可以创建邮件服务器、Web服务器或其他任何类型的在线服务器。

15.3.6 Web/HTTP服务器

如上节所示，Web服务器也是TCP服务器，用到最后几节介绍的许多技术，例如分支进程和线程化。Web服务器只是用HTTP沟通的常规TCP服务器。

不过，这里不打算直接考察HTTP服务器，因为在第10章已有介绍，因此如果你想回顾一下怎样用Ruby通过WEBrick或Mongrel构建基本的Web服务器，请参阅第10章最后几节内容。

15.3.7 后台进程

在上面的示例中，服务器都以常规应用程序的形式，在命令行运行。它们可以打印输出到屏幕，如果用Ctrl+C则可以关闭它们。不过，服务器一般都以后台（daemon）进程的形式运行，

独立于任何shell或终端程序。

注 本节与Windows用户无关，因为Windows的概念是服务而非后台进程。关于Windows服务的信息请参阅<http://www.tacktech.com/display.cfm?ttid=304>。

后台进程是持续运行并在机器后台保持静默的程序，不由用户直接运行。后台进程常常用于运行服务器，因为服务器是持续运行的，不需要与本机有任何交互。

在15.3.3节创建了基本的服务器，它分支为五个独立进程，用来监听并处理客户连接。结果是进程都运行在后台，但并非真正的独立，它们仍然把错误消息输出到当前屏幕，仍然挂接到父进程（即使父进程已经死亡）。

要让程序真正作为后台进程运行，需要执行一些操作：

1. 对进程进行分支操作，允许初始进程退出。
2. 调用操作系统的setsid函数，创建独立于任何终端或shell的新会话。
3. 再次对进程进行分支操作，确保新的后台进程是个孤儿进程，完全由其自我控制。
4. 改变工作目录到根目录，这样后台进程就不会阻止对原来当前工作目录的删除操作。
5. 确保标准输出、标准输入和标准出错文件句柄（STDIN、STDOUT和STDERR）没有连接到任何流。
6. 设置一个信号句柄来抓取任何TERM信号，以便让后台进程在需要时退出。

我们来看一下怎样用Ruby做这些事：

```
def daemonize
  fork do
    Process.setsid
    exit if fork
    Dir.chdir('/')
    STDIN.reopen('/dev/null')
    STDOUT.reopen('/dev/null','a')
    STDERR.reopen('/dev/null','a')
    trap("TERM"){exit }
    yield
  end
end

daemonize do
  #You can do whatever you like in here and it will run in the background
  #entirely separated from the parent process.
end

puts "The daemon process has been launched!"
```

daemonize方法执行上面列表的所有操作，然后把执行权移交给指定的代码块。这意味着代码块中的代码将在daemonize方法调用之后，建立后台化进程。在这里你可以创建GServer服务器，创建线程，或用Ruby做任何你想做的事，与启动初始程序的shell或终端完全不相干。

15.4 小结

本章我们考察了Ruby对构建底层网络连接工具和服务器的支持，并用Ruby开发了后台进程和其他持久运行进程。

我们来回顾一下本章涵盖的主要概念：

- 网络：是一组连接起来的计算机，通过连接可以互相发送和接收数据。
- TCP：是传输控制协议（Transmission Control Protocol）的缩写，是负责处理基于IP网络的两台机器之间的连接，并确保数据包按正确的顺序成功地发送与接收的协议。
- UDP：是用户数据报协议（User Datagram Protocol）的缩写，是允许两台计算机在无“连接”的情况下相互发送和接收消息，并不保证数据是否被远端接收的协议。
- IP：是网际协议（Internet Protocol）的缩写，是基于数据包的协议，用于通过网络发送数据。IP也为连接到网络的每台机器提供一个或多个IP地址。
- DNS：是域名服务（Domain Name Service）的缩写，是一套把主机或机器名和不同IP地址关联起来，并在二者之间相互转换的系统。例如，DNS服务器会把apress.com转换成IP地址65.19.150.101。
- Ping：是通过发送小数据包并等待回应，以检验特定IP地址的机器是否合法并接收请求的过程。
- 服务器：是运行在某台机器上并响应来自其他机器的客户端连接的进程，例如Web服务器。
- 客户端：是连接到服务器，传输并接收数据，然后在任务完成后断开连接的进程。Web浏览器是个客户端的简单例子。
- GServer：是一套Ruby程序库，简化了网络服务器和服务的开发。它负责线程管理和连接管理，允许对GServer类进行继承生成子类以创建服务器。
- 后台进程：是持续运行并在机器后台保持静默的进程，不由用户直接运行。后台进程常常用于运行服务器，因为它们持续运行，无须与本机进行交互。被转换到后台的进程常常被称为“后台化”。

此处标志着叙述性、教学性内容的最后终结，因为第16章是参考指南式的内容，介绍了大量Ruby程序库（包括标准程序库和以gem包形式提供的程序库）。因此，所有参与编写本书的人员向你表示感谢，感谢你读到这里，希望你会觉得下面的参考指南章节和索引的内容很有用。

在此致以最良好的祝愿，愿你继续Ruby世界之旅，开心快乐！请务必翻阅下面的参考指南章节和索引，以进一步丰富你的Ruby知识。

第16章 有用的Ruby程序库和gem包

本章是基本参考章节，提供了一组有用的Ruby程序库和RubyGems包，可供你在自己的程序中使用。我们考察的程序库将涵盖各种各样的功能，从网络联接到因特网访问，乃至文件解析和压缩。本章按字母顺序介绍程序库，为便于浏览，每个程序库以新的一页开始，页头标注该程序库的名字。下面是每个程序库的标题，后面是几个小节：

- 概述：描述程序库做什么，程序库的基本功能，以及为什么会想使用它。概述没有任何题头，但紧接在程序库名的下面。
- 安装：提供去哪里寻找、怎样安装、怎样在大多数系统中运行该程序库的信息。
- 示例：提供一个或多个程序库的使用示例，展示其功能的不同方面。示例中也包含运行结果。本节可能分为多个小节，每个小节包含单独的示例，展示如何使用特别的功能方面。
- 更多信息：提供关于该程序库更多信息的链接或指示，包含在线参考和教程。

与本书其他主要章节不同，本章是参考资料章节，你现在可能并不需要阅读，但随着时间的推移，当你想要找出怎样实现某个功能时，就会发现本章很有用。不管是什么情况，请务必至少浏览一下程序库列表，了解有哪些种类可用Ruby程序库，以便当你想实现某个程序库已有的功能时，不必再重新发明轮子！

注 我已尽我所能，确保文中链接引用的网站应该可以运行很多年，但有些网站可能已被转移或下线。在此情况下，最好的办法是用Google等搜索引擎，搜索“ruby”和程序库的名字。

16.1 abbrev程序库

abbrev程序库只提供了一个方法，用于对每个字符串组参数，算出一组唯一的缩写。

16.1.1 安装

abbrev包含在标准程序库中，由Ruby默认自带。要使用它，只需把下面这行代码放在程序的开头位置附近：

```
require 'abbrev'
```

16.1.2 示例

abbrev提供了一个单独的方法，可用两种方法来访问：一个是直接通过`Abbrev::abbrev`来调用，另一个是作为Array类的附加方法。我们先来看最基本的例子：

```
require 'abbrev'  
require 'pp'
```

```
pp Abbrev::abbrev(%w{Peter Patricia Petal Petunia})
```

```
{ "Patrici" => "Patricia",
  "Patric"  => "Patricia",
  "Petal"   => "Petal",
  "Pat"     => "Patricia",
  "Petu"    => "Petunia",
  "Patri"   => "Patricia",
  "Patricia" => "Patricia",
  "Peter"   => "Peter",
  "Petun"   => "Petunia",
  "Petuni"  => "Petunia",
  "Peta"    => "Petal",
  "Pa"      => "Patricia",
  "Patr"    => "Patricia",
  "Petunia" => "Petunia",
  "Pete"    => "Peter" }
```

如果有一个输入需求，需要许多可猜测的答案，那么abbrev对很有用，因为你可以更容易地检测部分输入或错误输入的数据。例如：

```
require 'abbrev'

abbrevs = %w{Peter Paul Patricia Petal Pauline}.abbrev
puts "Please enter your name:"
name = gets.chomp

if a = abbrevs.find {|a,n| a.downcase == name.downcase }
  puts "Did you mean #{a.join('or ')}?"
  name = gets.chomp
end
```

```
Please enter your name:
paulin
Did you mean Paulin or Pauline?
pauline
```

因为abbrev给出的结果是可能的最长唯一缩写，因此如果输入数据集更小，就可以更多地依赖于这个程序库。

16.1.3 更多信息

abbrev官方文档：<http://www.ruby-doc.org/stdlib/libdoc/abbrev/rdoc/index.html>

16.2 base64程序库

base64是一种对8比特位二进制数据进行编码的方法，其结果以7比特位数据格式表示。它只

利用A~Z、a~z、0~9、+和/这些字符来表示数据（也用=来填充数据）。一般情况下，这种编码方法把三个8比特位字节转换成四个7比特位字节，导致数据长度增加33%。这样做的主要好处在于，base64方法让二进制数据能够以普通文本的方式表示，因此可以更加可靠地用于电子邮件传送、数据库存储或用于文本格式，例如YAML和XML。

注 base64技术标准详见RFC 2045 (<http://www.faqs.org/rfcs/rfc2045.html>)。

16.2.1 安装

base64程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'base64'
```

16.2.2 示例

下面两个例子展示了怎样把二进制数据转换成base64表示形式，以及如何转换回来。然后我们将考察第三个例子，即展示怎样通过压缩来更有效地使用base64表示法。

把二进制数据转换成base64数据

base64程序库包含单个模块base64，其中提供了encode64和decode64方法。要把数据转换成base64格式，请使用encode64方法：

```
require 'base64'
puts Base64.encode64('testing')
```

```
dGVzdGluZW==
```

在本例中，只对已经是可打印的数据进行转换（尽管在技术上说，其内部格式还是8比特位数据），这是可接受的。不过，一般情况下，你编码的二进制数据来自文件，或是在其他地方生成的：

```
require 'base64'
puts Base64.encode64(File.read('/bin/bash'))
```

```
yv66vgAAAAIAAAHAAAAAwAAEAAAB4xQAAADAAABIAAAAAAegAAAIrywA
AAAMAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
<hundreds of lines skipped for brevity>
```

注 本例在OS X和Linux操作系统可以正常运行。对于Windows机器，你可以试试把/bin/bash替换成c:\windows\system\cmd.exe，从而得到类似的结果。

把base64数据转换成二进制数据

要把base64格式编码的数据转换回原始数据，请使用decode64方法：


```
require 'base64'
puts Base64.decode64(Base64.encode64('testing'))
```

```
testing
```

请注意，如果你试图解码非base64格式，不会得到任何错误反馈，只是从decode64方法得到没有任何意义的返回数据。

用压缩让base64更高效

尽管base64把一段数据的长度增加了33%，但可以在转换成base64之前压缩数据，然后在转换回二进制数据之后再解压缩。

注 并非所有二进制数据的压缩效果都好，不过大多数情况下都至少缩小5%，通常还会缩小更多。

要压缩和解压缩，你可以用zlib程序库，本章后文将会介绍，如下所示：

```
require 'base64'
require 'zlib'

module Base64
  def Base64.new_encode64(data)
    encode64(Zlib::Deflate.deflate(data))
  end

  def Base64.new_decode64(data)
    Zlib::Inflate.inflate(decode64(data))
  end
end

test_data = 'this is a test'*100

data = Base64.encode64(test_data)
puts "The uncompressed data is #{data.length}bytes long in Base64"

data = Base64.new_encode64(test_data)
puts "The compressed data is #{data.length}bytes long in Base64"
```

```
The uncompressed data is 1900 bytes long in Base64
```

```
The compressed data is 45 bytes long in Base64
```

在本例中，向Base64模块加入两个新方法，在把数据转换成base64格式之前，先用zlib对数据进行压缩，把数据从base64格式转换回来之后，再解压缩数据。这样，就在一定程度上节省了空间。关于zlib操作细节的更多信息，请参阅16.17节。

16.2.3 更多信息

下面是关于base64以及base64程序库的一些链接，它们提供了优秀的信息资源：

- base64的标准程序库文档: <http://www.ruby-doc.org/stdlib/libdoc/base64/rdoc/index.html>
- 关于Base64标准的一般信息: <http://en.wikipedia.org/wiki/Base64>
- 关于Base64用法的实际考察: http://email.about.com/cs/standards/a/base64_encoding.htm

16.3 BlueCloth程序库

BlueCloth是用来把特殊格式(名为Markdown格式)的文本文档转换成合法HTML的程序库。使用Markdown这类语言的原因,是大多数用户喜欢以清爽的格式编写文档,而不是非得到处使用HTML标签,这会导致文档读起来不像通顺的文本。Markdown可以让标注后的文档像通顺文档一样漂亮,也可以快速转换成HTML格式,以便在万维网上使用。这个优点让Markdown等语言在在线发贴与回贴系统中非常流行,甚至许多博客作者首先用Markdown等语言编写贴子,然后再发布。

16.3.1 安装

BlueCloth不是Ruby标准程序库的组成部分,它以RubyGem包的形式提供。要安装BlueCloth,请使用常规的gem包安装过程(详见第7章),如下所示:

```
gem install BlueCloth
```

或

```
sudo gem install BlueCloth
```

16.3.2 示例

Markdown文档示例如下:

```
This is a title
=====
```

```
Here is some _text_that's formatted according to [Markdown][1]
*specifications*.And how about a quote?
```

```
[1]:http://daringfireball.net/projects/markdown/
```

```
>This section is a quote..a block quote
>more accurately..
```

Lists are also possible:

```
*Item 1
*Item 2
*Item 3
```

在下面的示例中,为了节省篇幅,我们假定该文档已经赋值给变量markdown_text。

BlueCloth通过继承String形成了子类，并加入to_html方法，用来把Markdown标签转换成HTML。这样就可以对BlueCloth对象使用String类的标准方法。

下面是把Markdown语法转换成HTML的例子：

```
require 'rubygems'
require 'bluecloth'

bluecloth_obj = BlueCloth.new(markdown_text)
puts bluecloth_obj.to_html
```

```
<h1>This is a title</h1>

<p>Here is some <em>text</em>that's formatted according to <a
href="http://daringfireball.net/projects/markdown/">Markdown</a>
<em>specifications</em>.And how about a quote?</p>

<blockquote>
  <p>This section is a quote...a block quote
  more accurately..</p>
</blockquote>

<p>Lists are also possible:</p>

<ul>
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
</ul>
```

用Web浏览器观看，可以看到输出的HTML正确地模仿了Markdown语法。

如要了解Markdown格式及其语法的更多内容，请访问Markdown官方主页，如下节所示。

16.3.3 更多信息

- BlueCloth官方主页：<http://www.deveiate.org/projects/BlueCloth>
- Markdown格式官方主页：<http://daringfireball.net/projects/markdown/>

16.4 cgi程序库

CGI是通用网关接口（Common Gateway Interface）的缩写（尽管很少这么叫），它是一套系统，用来在Web服务器上运行脚本，并允许脚本传入付出数据，以生成响应内容，并反馈给通过Web浏览器访问该脚本的用户。

CGI已今非昔比，不像过去那么常用，但仍然在某些情况下使用，例如用一小段脚本来执行简单任务，而且针对每个请求，载入Ruby程序的开销不是什么大问题。

Ruby的CGI程序库不仅提供了页头和CGI数据管理等方法，还提供管理cookie和session的工具，让这些处理变得更简单。

16.4.1 安装

cgi程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'cgi'
```

16.4.2 示例

本节你将看到cgi可以做几件独特的事情。通过创建一些简单的CGI脚本，展示怎样利用cgi程序库生成Ruby脚本以响应Web输入，并在网页范围内执行动态运算（CGI执行在第10章有介绍）。

基本的CGI脚本

我们来创建一个基本的CGI脚本，该脚本可被上传到Web空间，该空间位于基于Linux的Web主机环境中（大多如此）。下面是CGI脚本的示例，它生成一个简单的网页：

```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new

puts cgi.header
puts "<html><body>This is a test</body></html>"
```

如果把这个Ruby脚本命名为test.cgi，上传到前文提到的Web主机，并设置为可执行文件，即可通过<http://www.your-website.com/test.cgi>来访问，并看到“This is a test”的响应内容。

注 上述脚本使用了shebang行，其中假定Ruby解释器位于/usr/bin/ruby。如果位置不对，该脚本将执行失败。万一碰到这种情况，请修改为正确的路径名，并参阅第10章关于怎样解决此类问题的文档说明。

上例的工作原理是，Web主机提供的Web服务器识别出用户请求的是CGI文件，遂执行相应的CGI脚本。脚本第一行告诉Web服务器，该脚本是Ruby脚本，然后载入cgi程序库并打印输出页头，同时将结果返回给Web服务器，然后在发送自己创建的一些HTML内容。

还可以用更直接的方式使用cgi程序库，即把准备返回给Web浏览器的一串数据发送给cgi程序库，让它处理页头输出以及与请求相关的任何数据。示例如下：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

cgi.out do
  "<html><body>This is a test</body></html>"
end
```

用cgi.out来返回数据给Web浏览器的一大主要好处，是out方法会自动处理页头输出，无须

费心。这种方法将在“cookie”小节用得更多，out方法也将通过请求，自动发回其他形式的数据。

注 如要进一步了解out方法及其支持的其他功能，请访问<http://www.ruby-doc.org/stdlib/libdoc/cgi/rdoc/classes/CGI.html#M000078>。

接受CGI变量

CGI脚本的一大好处是可以处理由HTML页面表单传送过来的数据，或在URL中指定的数据。如果你有个Web表单要传送给test.cgi，表单中包含名为“text”的元素，可以按下面的方式访问传递给它的数据：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

text = cgi['text']

puts cgi.header
puts "<html><body>#{text.reverse}</body></html>"
```

在此情况下，用户会看到他在表单中输入的文本被反转了。你还可以把文本直接放在URL中来测试CGI脚本，例如<http://www.mywebsite.com/test.cgi?text=this+is+a+test>。

下面是个更完整的例子：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

from = cgi['from'].to_i
to = cgi['to'].to_i

number = rand(to - from + 1) + from

puts cgi.header
puts "<html><body>#{number}</body></html>"
```

该CGI脚本返回一个随机数，取值范围在两个CGI变量from和to之间。用来发送正确数据的相关基本表单应该有HTML代码，如下所示：

```
<form method="POST" action="http://www.mywebsite.com/test.cgi">
For a number between <input type="text" name="from" value="" />and
<input type="text" name="to" value="" /><input type="submit"
value="Click here!" /></form>
```

cookie

cookie是小段数据，可在Web服务器和浏览器之间传送。如果从程序发送cookie到Web浏览器，该cookie将保存在用户的Web浏览器中，并在后续请求中发送回来（通常是这样，但有些人

禁用了cookie功能)。

例如, cookie可以在用户的计算机中保存一个数字, 将来每次请求同一个页面(大多数情况下, 是同一个网站)时都发送回来。你可以在每次请求时把这个数字每次递增1, 以显示该用户已访问某个页面多少次。

用cgi程序库创建和操作cookie非常简单。在本例中, 你将设置用户计算机中的cookie, 然后检索该cookie, 如果它在将来请求中出现的话:

```
#!/usr/bin/ruby

require 'cgi'
cgi =CGI.new

cookie =cgi.cookies ['count']

#If there is no cookie,create a new one
if cookie.empty?
  count =1
  cookie =CGI::Cookie.new('count',count.to_s)
else
  #If there is a cookie,retrieve its value (note that cookie.value results
  #in an Array)
  count =cookie.value.first

  #Now send back an increased amount for the cookie to store
  cookie.value =(count.to_i +1).to_s
end

cgi.out("cookie"=>[cookie ])do
  "<html><body>You have loaded this page #{count}times</body></html>"
end
```

在第一次请求本例所示脚本时, 你会看到如下结果:

```
You have loaded this page 1 times
```

在后续的每个请求中, 一将递增为二, 二将递增为三, 如此逐次递增下去。这是因为如果脚本检测到名为count的cookie, 就会检索并调整cookie的值, 然后对每个请求, 都用传递给cgi.out的参数, 把它cookie发回去。

cookie有许多用途, 远远超出了本例所示的情况。它们常常用于保存用户名、位置和其他小段信息, 用来控制特定网页应该显示什么内容。它们也常常用于跟踪“会话”, 我们将在下面介绍。

会话 (session)

cookie就像沉默的数据片断, 在客户端和Web服务器的每次请求时来回传递。而会话则不然, 它为Web浏览器和Web服务器提供了更容易管理和更抽象的关系。客户端不再来回发送实际数据, 而是只需要发送一个会话ID, 任何关于该用户或该会话的数据, 都存储在Web服务器上, 由CGI脚本自己管理。利用会话, 可以在各次请求之间“管理状态”, 这里的状态可能是巨大的数据集, 而非小小的cookie所能承载。

有了cgi程序库，在CGI脚本中加入会话功能变得非常简单。你无须使用cookie功能以文件或数据库的形式手工实现会话（这仍然是可选方案，如果你需要如此级别的控制的话），而是让CGI::Session类为你处理一切。

下面是个使用CGI::Session的例子，为每个访问该页面的用户都定义一个数据存储区：

```
#!/usr/bin/ruby

require 'cgi'
require 'cgi/session'
require 'cgi/session/pstore'

cgi = CGI.new
session = CGI::Session.new(cgi,
                           :session_key => 'count_app',
                           :database_manager => CGI::Session::PStore,
                           :prefix => 'session_id'
                          )

if session ['count'] && session ['count'].to_i > 0
  session ['count'] = (session ['count'].to_i + 1).to_s
else
  session ['count'] = 1
end

cgi.out do
  "<html><body>You have loaded this page #{session ['count']}times</body></html>"
end

session.close
```

在本例中，与cookie代码一样，执行了相同的计数运算，但本例可以为每个会话存储许多KB的数据，例如购物车信息、二进制数据或其他诸如YAML和XML文档的元数据形式。

请注意，上面的代码与Ruby on Rails处理会话的代码类似。这个session变量就像特殊的散列表，对每个独立用户都创建并保存。但与Rails不同的是，在会话使用完毕后调用了close方法，这样任何新数据都可以安全地写入磁盘。

注 你可以把上例代码载入各种不同的Web浏览器，以测试代码的有效性（例如，用Firefox和IE浏览器，而不是同一个浏览器的不同窗口）。

如想了解CGI::Session的更多内容，请访问<http://www.ruby-doc.org/core/classes/CGI/Session.html>，其中包含怎样用不同方式让CGI::Session保存会话数据（例如保存在内存中，或保存为纯文本格式）。

16.4.3 更多信息

- cgi程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/cgi/rdoc/index.html>

- 关于CGI的更多信息: <http://www.w3.org/CGI/>
- 关于HTTP cookie的更多信息: http://en.wikipedia.org/wiki/HTTP_cookie

16.5 chronic程序库

chronic程序库可以简化日期的转换,把几乎以任何格式书写的日期和时间,转换成Ruby可以内部正确识别的日期和时间格式。该程序库接受'tomorrow'、'last tuesday 5pm'等字符串,并将其转换成合法的Time对象。

16.5.1 安装

chronic程序库不是Ruby标准程序库的组成部分,它以RubyGem包的形式提供。要安装该程序库,请使用常规的gem包安装过程(详见第7章),如下所示:

```
gem install chronic
```

或

```
sudo gem install chronic
```

16.5.2 示例

chronic可接受以自然语言格式书写的日期和时间参数,并返回合法的Time对象。下面是一些基本的例子:

```
puts Chronic.parse('last tuesday 5am')
```

```
Tue Nov 07 05:00:00 +0000 2006
```

```
puts Chronic.parse('last tuesday 5:33')
```

```
Tue Nov 07 17:33:00 +0000 2006
```

```
puts Chronic.parse('last tuesday 05:33')
```

```
Tue Nov 07 05:33:00 +0000 2006
```

```
puts Chronic.parse('last tuesday lunchtime')
```

```
Tue Nov 07 17:33:00 +0000 2006
```

```
puts Chronic.parse('june 29th at 1am')
```

```
Fri Jun 29 01:00:00 +0100 2007
```

```
puts Chronic.parse('in 3 years')
```

```
Thu Nov 12 03:50:00 +0000 2009
```

```
puts Chronic.parse('sep 23 2033')
```

```
Fri Sep 23 12:00:00 +0100 2033
```

```
puts Chronic.parse('2003-11-10 01:02')
```

```
Mon Nov 10 01:02:00 +0000 2003
```

如果日期或时间无法识别，`Chronic.parse`将返回`nil`。

注 标准程序库提供了`Time`类的扩展，也可以解析时间，不过要求参数采用的正规程度略高。相关详细信息请参见<http://stdlib.rubyonrails.org/libdoc/time/rdoc/index.html>。标准程序库中还有个名为`ParseDate`的程序库，它提供了一个方法，可以把文本化的日期转换成数组值，以表示指定日期的不同方面。如要了解`ParseDate`的更多信息，请参见<http://www.ruby-doc.org/stdlib/libdoc/parsedate/rdoc/index.html>。

16.5.3 更多信息

- `chronic`的文档：<http://chronic.rubyforge.org/>
- `chronic`的更多示例：<http://www.yup.com/articles/2006/09/10/a-natural-language-date-time-parser-for-ruby-chronic>

16.6 Digest程序库

一段文摘（digest）（经常被称为散列表——尽管与Ruby用来存储数据结构的散列表不是相同类型）——是从另一组数据中生成的几个或一串数据。与原始数据相比，文摘相当短，而且被用作原始数据的校验和（checksum）。文摘的生成方式是这样：其他合法数据不太可能生成相同的值，如果想要创建另一段合法数据，使其文摘结果与此文摘结果相同，即使不是“不可能”，起码是相当困难。

散列表或文摘的常见用途，是在数据库中安全地保存密码。一般不以明文形式保存密码（因为可能被人窥视），而是创建密码的文摘并保存，这样当需要校验密码是否正确时，只须比较密码的文摘。你将在“示例”小节看到这样的例子。

16.6.1 安装

Ruby中生成文摘的程序库名为`digest/sha1`或`digest/md5`。这两个程序库都是标准程序库的组成部分，因此均由Ruby默认自带。要使用这两个程序库，只须把下面这行代码放在程序的开头位置附近：

```
require 'digest/sha1'
```

或

```
require 'digest/md5'
```

16.6.2 示例

我们来看一段数据的文摘大概是什么样子：

```
require 'digest/sha1'
puts Digest::SHA1.hexdigest('password')
```

```
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
```

你可以用十六进制（对Digest::SHA1和Digest::MD5均可——更多内容详见本节后文）来生成任何数据的文摘。这个文摘是由20个十六进制8比特位数字组成的字符串，在此情况下，文摘比输入数据长了很多。在实际使用中，输入数据一般比文摘结果更长。不管是哪种情况，通过Digest::SHA1生成的任何文摘都是完全相同的长度。例如，下面是由4 000个字符组成的输入字符串生成的文摘：

```
require 'digest/sha1'
puts Digest::SHA1.hexdigest('test' * 1000)
```

```
52fcb8acabb0a5ad7865350249e52bb70666751d
```

Digest::SHA1使用SHA-1散列算法，这是目前已知并广为使用的安全可靠的散列运算方法。该算法生成的输出结果为160位（通过hexdigest形成20位十六进制数字），这表示有1 461 501 637 330 902 918 203 684 832 716 283 019 655 932 542 976种可能的散列值。这几乎保证了对于单一领域的有效数据，不会有冲突的散列值。

Ruby提供的另一种散列机制是MD5散列算法。MD5生成128比特位散列值，从而提供了340 282 366 920 938 463 463 374 607 431 768 211 456种组合。一般认为MD5的安全性比SHA-1略低，因为有可能生成“散列冲突”，即两组合法数据可能生成相同的散列值。散列冲突可用来攻击基于MD5散列算法的认证系统。不过，MD5仍然是流行的散列机制，因此Ruby支持MD5对开发人员来说是很有用的。可以用与SHA-1完全相同的方法来使用Digest::MD5。

```
require 'digest/md5'
puts Digest::MD5.hexdigest('test' * 1000)
```

```
b38968b763b8b56c4b703f93f510be5a
```

在需要密码的场合使用文摘非常容易：

```
require 'digest/sha1'

puts "Enter the password to use this program:"
password = gets
if Digest::SHA1.hexdigest(password) == '24b63c0840ec7e58e5ab50d0d4ca243d1729eb65'
  puts "You've passed!"
else
  puts "Wrong!"
  exit
end
```


在此情况下，密码以SHA-1十六进制数字保存，对任何输入的密码都进行散列运算，以检查是否相同。这样做也不知道密码到底是什么，用上述程序没有任何办法获知密码，甚至看了源代码也没用！

注 我悬赏50美元征求第一个（只给第一个！）对上例密码编写出反散列解码版本的人，请通过<http://www.rubyinside.com/>联系我。

还可以生成原始数据，不必用digest方法转成十六进制字符串，如下所示：

```
Digest::SHA1.digest('test' * 1000)
```

由于生成结果是20字节的8比特位数据，如果以字符形式将其打印输出到屏幕，你对输出结果可能不太满意，但可以证明结果值存在：

```
Digest::SHA1.digest('test' * 1000).each_byte do |byte|
  print byte, "-"
end
```

```
82-252-184-172-171-176-165-173-120-101-53-2-73-229-43-183-6-102-117-29-
```

值得一提的是，如果想以文本格式保存文摘，但又想比40个十六进制数字字符占用更少的空间，那么base64程序库可以提供帮助：

```
require 'base64'
require 'digest/sha1'

puts Digest::SHA1.hexdigest('test')
puts Base64.encode64(Digest::SHA1.digest('test'))
```

```
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
qUqP5cyxm6YcTAhz05Hph5gvu9M=
```

16.6.3 更多信息

- 关于SHA-1的更多信息：<http://en.wikipedia.org/wiki/SHA-1>。
- 关于MD5的更多信息：<http://en.wikipedia.org/wiki/MD5>。

16.7 English程序库

在阅读本书的过程中，经常用到Ruby提供的用于各种用途的特殊变量。例如，\$!包含程序抛出的最新出错消息的字符串，\$\$返回当前程序的进程ID，\$/可以用来调整gets方法所用的默认行分隔符或记录分隔符。English程序库则可以用英语表示的名字来访问这些Ruby特殊变量，而不是用符号来访问。这样一来，这些变量就更容易记忆。

16.7.1 安装

English程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面

这行代码放在程序的开头位置附近：

```
require 'English'
```

16.7.2 示例

用`require 'English'`语句（注意首字母要大写，这与其他程序库采用的标准化全小写字母不同），即可为Ruby特殊变量创建英语别名，下面介绍其中的一些名字：

- `$DEFAULT_OUTPUT`（是`$>`的别名）：是`print`和`puts`等命令输出目标的别名。默认情况下，该目标指向`$stdout`，即标准输出设备，一般是指屏幕或当前终端（更多信息请参见第9章边栏“标准输入与输出”）。
- `$DEFAULT_INPUT`（是`$<`的别名）：是一个对象，其功能有点像`File`对象，代表从命令行发送给脚本的数据，或如果该数据缺失，则代表标准输入设备（一般是指键盘或当前终端）。该变量是只读变量。
- `$ERROR_INFO`（是`$!`的别名）：指向传递给`raise`的异常对象，或更实际地说，它可以包含最近一次的出错消息。它的初始化形式在`rescue`代码块中很有用。
- `$ERROR_POSITION`（是`$@`的别名）：返回上一次异常所生成的调用栈，栈的格式与`Kernel.caller`提供的格式相同。
- `$OFS`和`$OUTPUT_FIELD_SEPARATOR`（是`$,`的别名）：可读写，包含`print`方法和`Array`的`join`方法在输出时用到的默认分隔符。其默认值是`nil`，用`%w{a b c}.join`即可确认这一点，结果是“abc”。
- `ORS`和`$OUTPUT_RECORD_SEPARATOR`（是`$\`的别名）：可读写，包含`print`和`IO.write`等方法在输出时用到的默认分隔符。其默认值是`nil`，因此当需要在输出数据之后加换行时，要改用`puts`方法。
- `$FS`和`$FIELD_SEPARATOR`（是`$;`的别名）：可读写，包含`String`的`split`方法用到的默认分隔符。修改该值再对字符串调用`split`，如果不加切分正则表达式或切分字符的话，将会得到意料之外的结果。
- `$RS`和`$INPUT_RECORD_SEPARATOR`（是`$/`的别名）：可读写，包含`gets`等方法在输入时用到的默认分隔符。其默认值是换行符（`\n`），效果是`gets`每次接收一行数据。如果把该值设为`nil`，则`gets`会把整个文件或数据流一次读入。
- `$PID`和`$PROCESS_ID`（是`$$`的别名）：返回当前程序的进程ID。该ID对计算机中每个程序或程序实例都是独一无二的，因此`tempfile`用它来构造临时文件的名称。该变量是只读变量。
- `$LAST_MATCH_INFO`（是`$~`的别名）：返回`MatchData`对象，其中包含最近一次成功模式匹配的结果。
- `$IGNORECASE`（是`$=`的别名）：是个可读写的标志，用于确定在进行正则表达式和模式匹配时，默认情况下是否忽略大小写。不推荐使用该特殊变量，它在Ruby 2中可能会被去掉。一般情况下，如果需要该功能，可以改为在正则表达式末尾加`/i`标志。
- `$MATCH`（是`$&`的别名）：包含在当前范围内最近一次正则表达式成功匹配的整个字符串。如果没有任何匹配，该变量的值为`nil`。

- \$PREMATCH (是\$`的别名): 包含在当前范围内最近一次正则表达式成功匹配的字符串之前的字符串。如果没有任何匹配, 该变量的值为nil。
- \$POSTMATCH (是\$'的别名): 包含在当前范围内最近一次正则表达式成功匹配的字符串之后的字符串。如果没有任何匹配, 该变量的值为nil。

16.7.3 更多信息

English程序库的标准文档:<http://www.ruby-doc.org/stdlib/libdoc/English/rdoc/index.html>

16.8 ERB程序库

ERB是Ruby的一套模板程序库, 让你可以把其他内容与Ruby代码混用。ERB是Ruby on Rails渲染RHTML视图所用的主要模板系统(更多信息详见第13章)。在一套强大的模板系统中把Ruby代码和其他内容混合在一起, 不禁让人有点怀旧地想起PHP。

16.8.1 安装

ERB程序库是标准程序库的组成部分, 由Ruby默认自带。要使用该程序库, 只需把下面这行代码放在程序的开头位置附近:

```
require 'erb'
```

16.8.2 示例

ERB通过接收以ERB模板语言编写的模板, 将其转换成Ruby代码, 来生成所需的输出, 并执行这些代码。

基本的模板和内容渲染

基本的ERB脚本看起来像下面这样:

```
<% 1.upto(5) do |i| %>
  <p>This is iteration <%= i %></p>
<% end %>
```

在该模板中, Ruby代码和HTML代码混合在一起。要执行的Ruby代码放在<%和%>标签中, 要运算并“打印输出”的Ruby代码放在<%=和%>标签中, 而正常内容则保持原样。通过ERB运行上述模板, 输出结果如下:

```
<p>This is iteration 1</p>
<p>This is iteration 2</p>
<p>This is iteration 3</p>
<p>This is iteration 4</p>
<p>This is iteration 5</p>
```

注 由于模板的空格排列, 输出结果的空格看起来可能比较奇怪。通常在HTML或

XHTML中加空格是没有作用的，但如果用ERB输出其他格式的数据，在编写模板时可能需要注意空格的使用。

利用ERB程序库，可以从Ruby程序中渲染ERB代码：

```
require 'erb'

template = <<EOF
<% 1.upto(5) do |i| %>
  <p>This is iteration <%= i %></p>
<% end %>
EOF
```

```
puts ERB.new(template).result
```

`result`方法并不直接打印输出数据，而是把渲染的模板返回给调用方，然后就可以用`puts`把结果打印输出到屏幕。如果想让ERB直接打印输出到屏幕，可以用`run`方法：

```
ERB.new(template).run
```

访问外部变量

ERB模板也可以在当前范围内访问外部变量。例如：

```
require 'erb'

array_of_stuff = %w{this is a test}

template = <<EOF
<%array_of_stuff.each_with_index do |item,index| %>
  <p>Item <%=index %>:<%=item %></p>
<%end %>
EOF

puts ERB.new(template).result
```

```
<p>Item 0:this</p>
<p>Item 1:is</p>
<p>Item 2:a</p>
<p>Item 3:test</p>
```

注 如果想让ERB访问其他范围所定义的变量，或想把模板访问的变量放在“沙箱”内，则`result`和`run`方法也接受可选的参数绑定。如果允许模板默认访问主绑定，请记住模板内的代码可能会修改当前变量的值，如果模板的作者愿意的话。

安全级别

由于ERB可以执行混合在其他内容中Ruby代码，因此允许并不信任的用户在所控制的系统中创建或修改ERB模板是一件不智的事。这是因为他们可能执行任意代码，这些代码可能访问

文件系统、删除数据，或对你的系统产生其他损害（回想一下，Ruby可以用backticks（两个`符号之间的系统命令）来运行计算机中当前用户可访问的任何程序）。

在第11章，你了解了Ruby提供的“安全级别”概念，即可以抑制代码的能力，特别是关于运行任意程序，或用eval等“危险”命令处理受感染的数据。

ERB.new把安全级别作为可选的第二个参数，从而提供了很大帮助，让模板渲染过程变得更加安全：

```
require 'erb'

template =<<EOF
Let's try to do something crazy like access the filesystem..
<%= `ls` %>
EOF

puts ERB.new(template,4).result      #Using safe level 4!

/usr/local/lib/ruby/1.8/erb.rb:739:in `eval':Insecure:can't modify
trusted binding (SecurityError)
```

安全级别只对运行ERB模板的代码起作用，但如果前面已经使用了安全级别，就无法再降低了。其工作原理是当安全级别应用于ERB时，ERB为处理这段ERB代码创建了新线程，其安全级别相对于主代码是独立的。

如要了解每种安全级别提供哪些能力，请参阅第11章或附录B。

16.8.3 更多信息

- ERB程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/index.html>
- Merb——使用ERB的一种轻量级应用服务器：<http://merb.rubyforge.org/>

16.9 FasterCSV程序库

在第9章你见识了逗号分隔值（Comma-Separated Value）数据库，这是存储数据的一种极其粗鲁的方式，CSV用逗号区分数据字段，用换行符区分记录。例如：

```
Clive,53,male,UK
Ann,55,female,France
Eugene,29,male,California
```

你还见识了Ruby提供的csv程序库，并用它来处理这种形式的数据。而由James Edward Gray II创建的FasterCSV程序库实现与之相同的功能，目的是成为速度更快的替代品。它的接口与标准csv程序库略有不同，不过如果有必要，它也可以模拟标准接口。

16.9.1 安装

FasterCSV不是Ruby标准程序库的组成部分（尽管这种情况未来可能会改变），它以

RubyGem包的形式提供。要安装该程序库，请使用常规的gem包安装过程，如下所示：

```
gem install fastercsv
```

或

```
sudo gem install fastercsv
```

16.9.2 示例

在以下示例中，我们假定该CSV数据放在名为data.csv的文件中：

```
Clive,53,male,UK
Ann,55,female,France
Eugene,29,male,California
```

逐行解析字符串

如果你想逐行迭代处理CSV数据，通过常规字符串可以很容易实现：

```
require 'rubygems'
require 'fastercsv'
FasterCSV.parse(File.read("data.csv")) do |person|
  puts person.inspect
end
```

把字符串解析到数组构成的数组中

在上例中，对CSV数据进行了“实时”处理，但FasterCSV也可以把CSV数据转换成另一种数据结构（数组构成的数组），每次以便轻松地比较和处理一个数据集：

```
require 'rubygems'
require 'fastercsv'
array_of_arrays = FasterCSV.parse(File.read("data.csv"))
array_of_arrays.each do |person|
  puts person.inspect
end
```

从文件中逐行解析CSV数据

在上例中，对CSV数据进行迭代处理，一行行放到字符串中，但FasterCSV也可以直接对文件进行操作：

```
require 'rubygems'
require 'fastercsv'
FasterCSV.foreach("data.csv") do |person|
  puts person.inspect
end
```

把整个CSV文件解析到数组构成的数组

让FasterCSV名声大振的能力是在一个方法调用中，把整个CSV文件读入，并转换到数组构成的数组中：

```
require 'rubygems'
```

```
require 'fastercsv'
array_of_arrays = FasterCSV.read("data.csv")
```

生成CSV数据

除了可以读入数据，FasterCSV也可以创建CSV格式的数据。要做这一操作，必须把要输出的数据以正确的顺序放在数组中。下面的例子展示了怎样把散列表构成的数组转换成CSV格式的输出：

```
require 'rubygems'
require 'fastercsv'

people = [
  { :name => "Fred", :age => 10, :gender => :male },
  { :name => "Graham", :age => 34, :gender => :male },
  { :name => "Lorraine", :age => 29, :gender => :female }
]

csv_data = FasterCSV.generate do |csv|
  people.each do |person|
    csv << [person[:name], person[:age], person[:gender]]
  end
end

puts csv_data
```

```
Fred,10,male
Graham,34,male
Lorraine,29,female
```

FasterCSV类还提供了open方法，可以直接写入到文件：

```
FasterCSV.open("data.csv", "w") do |csv|
  people.each do |person|
    csv << [person[:name], person[:age], person[:gender]]
  end
end
```

注 如果以模式"a"而非模式"w"打开文件，那么新增数据都会附加到现有数据的末尾。如果用模式"w"，新数据将会完全替换原数据。

FasterCSV还为String和Array提供了便捷方法，可以轻松地把单行字符串或单维数组直接与CSV相互转换：

```
puts ["Fred", 10, "male"].to_csv
```

```
Fred,10,male
```

```
puts "Fred,10,male".parse_csv.inspect
```

```
["Fred", 10, "male"]
```

FasterCSV二维表

FasterCSV还支持二维表数据结构。它可以把CSV文件第一行当作一组列名，以便用这些列名更方便地访问表中其他数据。如要让FasterCSV以二维表形式读取数据，并把第一行作为标题，请将FasterCSV类的读取方法的:headers选项设为true。

我们假定data.csv文件包含以下内容，以便让后续示例可以正常运行：

```
Name, Age, Gender, Location
Clive, 53, male, UK
Ann, 55, female, France
Eugene, 29, male, California
```

现在用下面的代码把data.csv当成二维表读入：

```
require 'rubygems'
require 'fastercsv'
require 'pp'

csv =FasterCSV.read("data.csv",:headers =>true)
pp csv
```

```
#<FasterCSV::Table:0x5b7ed0
  @mode=:col_or_row,
  @table=
[#<FasterCSV::Row:0x5b72f0
  @header_row=false,
  @row=
    [{"Name","Clive"},
     {"Age","53"},
     {"Gender","male"},
     {"Location","UK"}]>,
#<FasterCSV::Row:0x5b6c60
  @header_row=false,
  @row=
    [{"Name","Ann"},
     {"Age","55"},
     {"Gender","female"},
     {"Location","France"}]>,
#<FasterCSV::Row:0x5b651c
  @header_row=false,
  @row=
    [{"Name","Eugene"},
     {"Age","29"},
     {"Gender","male"},
     {"Location","California"}]>]>
```

注 在上例中用pp显示表结构，比puts csv.inspect的显示格式更漂亮，二者除了空格的区别，输出内容基本是相同的。pp程序库在本章后文介绍。

当使用表头时，并非导入数组构成的数组，而是创建了FasterCSV::Table对象，其中每一行都包含FasterCSV::Row对象。还可以看到每个列名都与每行的正确数据相关联。

FasterCSV::Table提供了一些有用的方法（这些例子假定csv包含FasterCSV::Table对象，像上例那样），如下所示：

- `csv.to_s`返回一个字符串，其内容是以CSV格式的整个表。你可以用这个方法把数据写回到文件中。
- `csv.to_a`返回一个数组构成的数组，其内容是表，和FasterCSV.read在未设置:headers选项时的结果相同。
- `csv <<`可用于把新行压入到表的末尾（例如，`csv << ['Chris', 26, 'male', 'Los Angeles']`）。
- `csv.headers`返回一个数组，其内容是表头行。
- `csv.delete('Name')`从每一行删除Name列。
- `csv.delete(n)`删除第n行内容。
- `csv[n]`返回第n行内容。
- `csv.each`用代码块迭代处理每一行。

表中各行（各FasterCSV::Row对象）也有其自己的方法来访问数据，如下例所示：

```
csv.each do |row|
  puts row['Name']
end
```

```
Clive
Ann
Eugene
```

每行数据作为FasterCSV::Row，而不是数组，就可以使用列标题名检索信息。同样，也可以设置列值等于其他内容：

```
csv.each do |row|
  row['Location'] = "Nowhere"
end

puts csv.to_csv
```

```
Name, Age, Gender, Location
Clive, 53, male, Nowhere
Ann, 55, female, Nowhere
Eugene, 29, male, Nowhere
```

注 在下面“更多信息”小节提供的FasterCSV官方文档链接中，你还可以找到其他更少使用的方法。

16.9.3 更多信息

- FasterCSV的官方文档：<http://fastercsv.rubyforge.org/>

- 关于CSV的更多信息: http://en.wikipedia.org/wiki/Comma-separated_values

16.10 iconv程序库

iconv是Ruby和UNIX的iconv工具程序的接口,可以在不同字符编码之间转换字符串。在第11章你了解了字符编码,以及怎样在Ruby中使用字符编码。而iconv为Ruby提供了在不同编码之间转换字符串的功能。

注 iconv工具程序和程序库(是指操作系统程序库,不是Ruby程序库)作为标准功能在所有Linux版本中均予提供(通常是BSD版本,例如Mac OS X),但不在Windows中提供。尽管iconv附于Windows的Cygwin环境中,但由于Windows环境的限制,该程序库还是不太可能正常运行。

16.10.1 安装

iconv程序库是标准程序库的组成部分,由Ruby默认自带。要使用该程序库,只需把下面这行代码放在程序的开头位置附近:

```
require 'iconv'
```

16.10.2 示例

用iconv在不同字符编码之间转换字符串,有三种主要方法。一是以句柄形式,二是以直接形式,三是以代码块形式。例如,你可以用句柄形式从UTF-8编码转换到ISO-8859-1编码,如下所示:

```
require 'iconv'
converter = Iconv.new('utf-8', 'iso-8859-1')
utf8_string = "This is a test"
iso_string = converter.iconv(utf8_string)
```

注 源编码在Iconv.new的第一个参数提供,而目标编码在第二个参数提供。

以代码块形式使用iconv,与使用File对象的代码块形式类似,如下所示:

```
require 'iconv'
Iconv.open('utf-8', 'iso-8859-1') do |converter|
  utf8_string = "This is a test"
  iso_string = converter.iconv(utf8_string)
end
```

可以用直接形式使用iconv,用一行代码转换字符串:

```
require 'iconv'
Iconv.iconv("utf-8", "iso-8859-1", "This is a test").to_s
```


16.10.3 更多信息

- iconv程序库的标准文档: <http://www.ruby-doc.org/stdlib/libdoc/iconv/rdoc/index.html>
- 关于iconv常规信息的维基百科文章: <http://en.wikipedia.org/wiki/Iconv>

16.11 logger程序库

logger (日志记录器) 是由Hiroshi Nakamura和Gavin Sinclair开发的程序库, 为Ruby应用程序提供了全面丰富的日志功能。它支持自动日志翻转和多重紧急级别, 并可输出到文件、标准输出设备或标准出错句柄。logger是Ruby on Rails的主要日志系统, 也可以从其他任何Ruby应用程序使用它。

16.11.1 安装

logger程序库是标准程序库的组成部分, 由Ruby默认自带。要使用该程序库, 只需把下面这行代码放在程序的开头位置附近:

```
require 'logger'
```

16.11.2 示例

要使用logger, 请先创建Logger对象, 然后用该对象提供的方法, 在程序运行过程中报告发生的事件。第一步是获取Logger对象。

建立logger对象

logger可以写入到标准输出设备、标准出错设备或文件中, 只需为Logger.new指定文件句柄或文件名即可。例如, 下面是怎样把日志信息直接写到屏幕或终端的方法:

```
require 'logger'
logger = Logger.new(STDOUT)
```

用下面的代码可以把日志信息写入文件:

```
logger = Logger.new('mylogfile.log')
logger = Logger.new('/tmp/some_log_file.log')
```

还可以指定日志文件每天、每周或每月更新 (旧日志文件加日期后缀):

```
logger = Logger.new('mylogfile.log', 'daily')
logger = Logger.new('mylogfile.log', 'weekly')
logger = Logger.new('mylogfile.log', 'monthly')
```

最后, 还可以创建指定日志文件最大空间的logger。一旦日志文件达到大小限制, logger就把现有日志文件复制到另一个文件名, 然后开启新的日志文件。这也被称为日志翻转 (log rotation):

```
logger = Logger.new('mylogfile.log', 10, 100000)
```

该logger的日志文件名为mylogfile.log, 当它到达100 000字节长度时, logger则将其改

名（加数字后缀）并创建新的mylogfile.log。该logger保留最近十个未使用的日志文件。

日志级别

日志级别有5种，按严重级别顺序排列如下：

- **DEBUG**：最低级别，用于开发人员调试信息。
- **INFO**：提供程序、程序库或系统运行的一般信息。
- **WARN**：提供程序状态的非致命错误警告。
- **ERROR**：提供可处理错误（例如有rescue代码块的异常）的信息。
- **FATAL**：提供不可修复的、强制程序立即结束的错误信息。

启动logger后，可以指定要跟踪的消息级别。如果消息属于指定级别或更高级别，则记入日志。如果低于指定级别，则忽略消息。这种功能很有用，这样一来，在开发时可以记录每个debug消息，而当程序在实际环境中运行时，只记录重要的消息。

要指定logger的严重级别，请使用logger的sev_threshold方法。下面这个级别确保只记录FATAL消息：

```
logger.sev_threshold = Logger::FATAL
```

下面这个级别确保所有级别的每个消息都记入日志：

```
logger.sev_threshold = Logger::DEBUG
```

把消息记入日志

每个Logger对象都提供了几个方法，可用来把消息发送到日志。最常用的方法是debug、info、warn、error和fatal方法，它们根据各自的严重程序创建日志消息。

```
require 'logger'
logger = Logger.new(STDOUT)
logger.debug "test"
logger.info "test"
logger.fatal "test"
```

```
D,[2007-03-12T11:06:06.805072 #9289 ] DEBUG ---:test
I,[2007-03-12T11:06:06.825144 #9289 ] INFO  ---:test
F,[2007-03-12T11:06:06.825288 #9289 ] FATAL ---:test
```

日志消息由以单个字母表示的严重程度、创建日志的日期和时间、创建日志的进程ID、日志的严重级别标签、以及实际消息组成。另外，如果在记日志方法中指定的话，也可以提供程序的名字，与正常消息放在一个块中，如下所示：

```
logger.info("myprog") { "test" }
```

```
I, [2007-03-12T11:09:32.284956 #9289] INFO -- myprog: test
```

也可以动态定义某个日志消息的严重程度，如下所示：

```
logger.add(Logger::FATAL) { "message here" }
```

```
F, [2007-03-12T11:13:06.880818 #9289] FATAL -- : message here
```

要使用不同的严重程序，只须把严重程度类（Logger::FATAL、Logger::DEBUG、

Logger::INFO等) 加入到参数中。

关闭logger

关闭logger的方法和关闭文件或其他I/O设备一样:

```
logger.close
```

16.11.3 更多信息

- logger程序库的标准文档: <http://www.ruby-doc.org/stdlib/libdoc/logger/rdoc/index.html>
- Log4r——另一个更复杂的Ruby日志程序库: <http://log4r.sourceforge.net/>

16.12 pp程序库

pp是“漂亮打印机”(pretty printer)的缩写,它提供比`puts something.inspect`更漂亮的输出结果。和`inspect`的输出不同,它通过适当的表格对齐和留空,为数据结构提供更干净、整齐的外观。

16.12.1 安装

pp程序库是标准程序库的组成部分,由Ruby默认自带。要使用该程序库,只需把下面这行代码放在程序的开头位置附近:

```
require 'pp'
```

16.12.2 示例

要使用pp,只需使用pp方法,后跟要显示数据结构的对象。下面是`inspect`和pp显示效果的基本比较:

```
person1 = {:name => "Peter", :gender => :male }
person2 = {:name => "Laura", :gender => :female }
people = [person1, person2, person1, person1, person1 ]
puts people.inspect
```

```
[{:name=>"Peter",:gender=>:male},{:name=>"Laura",:gender=>:female},
{:name=>"Peter",:gender=>:male},{:name=>"Peter",:gender=>:male},
{:name=>"Peter",:gender=>:male}]
```

```
pp people
```

```
[{:name=>"Peter",:gender=>:male},
{:name=>"Laura",:gender=>:female},
{:name=>"Peter",:gender=>:male},
{:name=>"Peter",:gender=>:male},
{:name=>"Peter",:gender=>:male}]
```

如上所示，对于数据内容超出一行的复杂对象，pp最为有用。下面是个更生造的例子：

```
require 'pp'

class TestClass
  def initialize(count)
    @@a = defined?(@@a)?@@a + 1 : 0
    @c = @@a
    @d = [:a => {:b => count }, :c => :d ] *count
  end
end

pp TestClass.new(2), STDOUT, 60
pp TestClass.new(3), $>, 60
pp TestClass.new(4), $>, 60
```

```
#<TestClass:0x357000
@c=0,
@d=[{:a=>{:b=>2},:c=>:d},{:a=>{:b=>2},:c=>:d}]>
#<TestClass:0x354364
@c=1,
@d=
[{:a=>{:b=>3},:c=>:d},
{:a=>{:b=>3},:c=>:d},
{:a=>{:b=>3},:c=>:d}]>
#<TestClass:0x3503f4
@c=2,
@d=
[{:a=>{:b=>4},:c=>:d},
{:a=>{:b=>4},:c=>:d},
{:a=>{:b=>4},:c=>:d},
{:a=>{:b=>4},:c=>:d}]>
```

只要有可能，pp就把数据放在一行对齐，但当数据超出一行空间时，pp就根据数据内容进行格式调整和留空。

请注意，上例中调用pp的格式如下：

```
pp TestClass.new(4), $>, 60
```

在没有任何参数时，pp假定显示宽度为79个字符。但pp支持两个可选参数，可设置输出的目标设备，以及输出字段的宽度。本例输出到标准输出设备，并假定按60个字符的宽度回卷。

16.12.3 更多信息

pp程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/pp/rdoc/index.html>

16.13 RedCloth程序库

RedCloth程序库用来把特殊格式的文本文档（名为Textile格式）转换成合法的HTML格式。它与本章前文介绍的BlueCloth程序库在许多方面都很相似。

使用Textile这类语言的原因，是大多数用户喜欢以清爽的格式编写文档，而不是非得到处使用HTML标签，这样会导致文档读起来不像通顺的文本。Textile可以让标注后的文档像通顺文档一样漂亮，也可以快速转换成HTML格式，以便在万维网上使用。

与BlueCloth所用的Markdown标记语言相比，Textile对HTML输出结果提供了更多一些的控制，并提供了某些简单易用的高级功能。不过，在我看来，尽管它确实大有用处，但更技术化，不像Markdown那么流行。

RedCloth由“why the lucky stiff”开发，Textile由Dean Allen开发。

16.13.1 安装

RedCloth不是Ruby标准程序库的组成部分，它是以RubyGem包的形式提供。要安装该程序库，请使用常规的gem包安装过程，如下所示：

```
gem install RedCloth
```

或

```
sudo gem install RedCloth
```

16.13.2 示例

RedCloth和BlueCloth的使用方法几乎完全相同，如本章前文所述。它有个直接从String类继承而来的RedCloth类，因此可以对RedCloth对象使用所有String类的常规方法。

RedCloth的基本示例与BlueCloth很相似：

```
require 'rubygems'
require 'redcloth'

redcloth_text =<<EOF
h1.This is a title

Here is some _text_that's formatted according to
"Textile":http://hobix.com/textile/*specifications*.
And how about a quote?
bq.This section is a quote..a block quote
more accurately..

Lists are also possible:

*Item 1
*Item 2
*Item 3
EOF
```



```
redcloth_obj = RedCloth.new redcloth_text
puts redcloth_obj.to_html
```

```
<h1>This is a title</h1>
  <p>Here is some <em>text</em>that's formatted according to
<a href="http://hobix.com/textile/">Textile</a><strong>specifications</strong>.
And how about a quote?</p>

  <blockquote>
    <p>This section is a quote..a block quote
more accurately..</p>
  </blockquote>

  <p>Lists are also possible:</p>

  <ul>
    <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
  </ul>
```

注 在上面的输出结果中，去掉了一些代码行的空格，但横向格式未作任何变动。

值得一提的是，乍看起来，RedCloth的输出不像BlueCloth那么清爽，但仍然是合法的HTML格式。

16.13.3 更多信息

- RedCloth官方主页：<http://whytheluckystiff.net/ruby/redcloth/>
- Textile官方主页：<http://www.textism.com/tools/textile/>

16.14 StringScanner程序库

StringScanner程序库提供“步进匹配”字符串的功能，每次匹配一个，每次只对剩余还未作匹配的数据进行匹配。这与标准scan方法完全相反，后者是立即自动返回所有匹配内容。

16.14.1 安装

StringScanner程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'strscan'
```

注 请务必注意，这里的文件名与程序库（在此情况下是类）的名字并不相同。尽管大多数程序库开发人员倾向于保持二者的统一，但并不是所有人都这么做！

16.14.2 示例

了解StringScanner有哪些功能的最好办法是看看它的实际使用：

```
require 'strscan'

string =StringScanner.new "This is a test"
puts string.scan(/\w+/)
puts string.scan(/\s+/)
puts string.scan(/\w+/)
puts string.scan(/\s+/)
puts string.rest
```

```
This
is
a test
```

本例对字符串进行步进匹配，首先用scan方法匹配一个单词，再匹配空格，然后是另一个单词，接下来又是空格，最后要求StringScanner用rest方法给出剩余的字符串内容。

不过，仅当指定模式正好匹配字符串的当前位置，scan才返回内容。例如，下面的代码不会检索出每个单词：

```
puts string.scan(/\w+/)
puts string.scan(/\w+/)
puts string.scan(/\w+/)
puts string.scan(/\w+/)
```

```
This
nil
nil
nil
```

在第一次扫描之后，string的指针指向“This”之后的空格，而scan必须匹配空格才能继续往下进行。解决这个问题的一种办法是这样：

```
puts string.scan(/\w+\s*/)
puts string.scan(/\w+\s*/)
puts string.scan(/\w+\s*/)
puts string.scan(/\w+\s*/)
```

上例对单词和每个单词后的任何空格进行检索。当然，这可能并不是原先期望的，因此StringScanner还提供了其他有用的扫描字符串方法。

scan_until方法从当前位置扫描，直到碰到指定的模式。从扫描开始的所有数据，直到匹配的位置，包括匹配内容，都被该方法返回。下面的例子执行了常规扫描，并取出第一个单词，然后用scan_until扫描所有剩余内容，直到碰到数字：

```
string =StringScanner.new "I want to live to be 100 years old!"
```

```
puts string.scan(/\w+/)
puts string.scan_until(/\d+/)
```

```
I
want to live to be 100
```

还可以用scan_until方法，对上文“扫描每个单词”问题给出不同的解决方案：

```
require 'strscan'
string = StringScanner.new("This is a test")
puts string.scan_until(/\w+/)
puts string.scan_until(/\w+/)
puts string.scan_until(/\w+/)
puts string.scan_until(/\w+/)
```

另一个有用的方法是unscan，它提供回滚上一次扫描效果的机会：

```
string =StringScanner.new "I want to live to be 100 years old!"
puts string.scan(/\w+/)
string.unscan
puts string.scan_until(/\d+/)
string.unscan
puts string.scan_until(/live/)
```

```
I
I want to live to be 100
I want to live
```

也可以检索扫描指针在字符串中的当前位置：

```
string =StringScanner.new "I want to live to be 100 years old!"
string.scan(/\w+/)
string.unscan
puts string.pos
string.scan_until(/\d+/)
puts string.pos
string.unscan
string.scan_until(/live/)
puts string.pos
```

```
0
24
14
```

此外，还可以用pos方法来设置或改写扫描指针的位置：

```
string = StringScanner.new "I want to live to be 100 years old!"
string.pos = 12
puts string.scan(/...../)
```

```
ve to
```

注 StringScanner不是String的子类，因此不能使用String类的常规方法。不过，StringScanner确实实现了String类的一些方法，例如<<方法，可以把数据追加到字符串末尾。

16.14.3 更多信息

StringScanner程序库标准文档：<http://www.ruby-doc.org/stdlib/libdoc/strscan/rdoc/index.html>

16.15 tempfile程序库

临时文件是指用一次的文件，它们生命短暂，用于临时保存信息，并被迅速删除。在第9章你了解了创建临时文件的几种方法，而tempfile提供了创建和操作临时文件的简单而标准的方法。

16.15.1 安装

tempfile程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'tempfile'
```

16.15.2 示例

tempfile对临时文件的创建和操作进行管理，它针对你的操作系统，在正确的位置创建临时文件，并给临时文件独一无二的名字，以便不使你分心，可以专注处理程序的主逻辑。

要创建临时文件，请用Tempfile.new方法：

```
require 'tempfile'
f = Tempfile.new('myapp')
f.puts "Hello"
puts f.path
f.close
```

```
/tmp/myapp1842.0
```

Tempfile.new方法用给定的字符串作为前缀创建一个临时文件，文件名格式为<指定名>-<程序进程ID>.<唯一数字>。该方法返回的对象是Tempfile对象，其大多数方法均代理给常用的File和IO类，你可以用自己熟悉的文件方法进行操作，例如上例的f.puts。

要使用临时文件中的数据，可以先关闭临时文件再迅速重新打开：

```
f.close
f.open
```

如果f.open方法未指定任何参数，则将重新打开该对象关联的临时文件。此时你可以向临时文件写入数据，或从中读出数据：

```
require 'tempfile'
```

```
f = Tempfile.new('myapp')
f.puts "Hello"
f.close
f.open
puts f.read
f.close!
```

Hello

上述代码创建了一个临时文件，向其中写入数据，并关闭临时文件（即从内存缓冲区把数据写回磁盘），并重新打开该文件以备读入。

最后一行代码使用了`close!`方法而不是`close`，前者强制关闭临时文件并将其永久删除。

当然，你可以手工清理缓冲区，从而可以使用同一个临时文件进行读写，任何时候都无须关闭：

```
require 'tempfile'

f =Tempfile.new('myapp')
f.puts "Hello"
f.pos =0
f.print "Y"
f.pos =f.size -1
f.print "w"
f.flush
f.pos =0
puts f.read
f.close!
```

Yellow

注 默认情况下，临时文件以`w+`模式打开。

在某些情况下，你可能想使用临时文件，但不想把临时文件放在可被其他程序或用户看到的地方。`Tempfile.new`方法接受可选的第二个参数，用来指定想在何处创建临时文件：

```
f = Tempfile.new('myapp', '/my/secret/temporary/directory')
```

与其他文件相关类一样，可以用代码块形式使用`Tempfile`：

```
require 'tempfile'

Tempfile.open('myapp')do |f|
  f.puts "Hello"
  f.pos =0
  f.print "Y"
  f.pos =f.size -1
  f.print "w"
  f.flush
  f.pos =0
```



```
puts f.read
end
```

Yellow

注 当用到代码块时，要用`Tempfile.open`而不是`Tempfile.new`方法。

在此情况下，用代码块形式的好处在于，临时文件将被自动删除，无须调用关闭方法。但如果想在整个程序的范围内使用临时文件，代码块形式可能就不是个好选择。

16.15.3 更多信息

tempfile程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/tempfile/rdoc/index.html>

16.16 uri程序库

uri程序库用来管理统一资源标识符（Uniform Resource Identifier, URI），一般又称为统一资源定位符（Uniform Resource Locator, URL）。URL是一个地址，例如<http://www.rubyinside.com/>，<ftp://your-ftp-site.com/directory/filename>，甚至是<mailto:your-email-address@privacy.net>。uri让这些地址的检测、创建、解析和操作变得极其简单。

16.16.1 安装

uri程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'uri'
```

16.16.2 示例

在本节中，将考察几个用uri程序库执行URL相关基本功能的例子。

从文本中析取URL

`URI.extract`是个类方法，用来从指定字符串中析取URL并放到数组中：

```
require 'uri'
puts URI.extract('Check out http://www.rubyinside.com/or e-mail ➡
mailto:me@privacy.net').inspect
```

```
["http://www.rubyinside.com/", "mailto:me@privacy.net"]
```

也可以限制析取的URL类型：

```
require 'uri'
puts URI.extract('http://www.rubyinside.com/and mailto:me@privacy.net', ➡
['http']).inspect
```

```
[ "http://www.rubyinside.com/" ]
```

如果想每析取出一个URL就立即使用，可以用代码块调用`extract`方法：

```
require 'uri'
```

```
email = %q{Some cool Ruby sites are http://www.ruby-lang.org/and ➡
http://www.rubyinside.com/and http://redhanded.hobix.com/}
```

```
URI.extract(email, ['http', 'https']) do |url|
  puts "Fetching URL #{url}"
  #Do some work here ...
end
```

解析URL

字符串中的URL可以很有用，例如特别是在用`open-uri`或`net/http`程序库的情况下，但把URL切分成各个组成成分也很有用。用正则表达式来做这件事可能会产生不一致的结果，在非常规情况下也容易出错，因此URI类提供了必要的工具，可以轻松地把URL切分成几部分。

```
URI.parse('http://www.rubyinside.com/')
```

```
=> #<URI::HTTP:0x2d071c URL:http://www.rubyinside.com/>
```

`URI.parse`方法解析字符串中提供的URL，并返回基于URI的对象。针对FTP、HTTP、HTTPS、LDAP和MailTo等URL，URI有具体的子类，但对于URL格式字符串中无法识别的URL，均返回`URI::Generic`对象。

URI对象有几个方法可以用来访问URL相关信息：

```
require 'uri'
a = URI.parse('http://www.rubyinside.com/')
puts a.scheme
puts a.host
puts a.port
puts a.path
puts a.query
```

```
http
www.rubyinside.com
80
/
nil
```

请注意，`URI::HTTP`很聪明，知道如果没有在HTTP URL中指定端口号，则必须使用默认的80端口。其他URI类，例如`URI::FTP`和`URI::HTTPS`，也有类似的假定。

对于更复杂的URL，可以访问扩展数据：

```
require 'uri'
url = 'http://www.x.com:1234/test/1.html?x=y&y=z#top'
puts URI.parse(url).port
```

```
puts URI.parse(url).path
puts URI.parse(url).query
puts URI.parse(url).fragment
```

```
1234
/test/1.html
x=y&y=z
top
```

uri程序库也提供了方便的方法，让解析URL变得更容易：

```
u = URI('http://www.test.com/')
```

在此情况下，`URI(url)`是`URI.parse`的同义方法。

和`URI.parse`一样，可以用`URI.split`把URL切分成各个组成成分，无须用到URI对象：

```
URI.split('http://www.x.com:1234/test/1.html?x=y&y=z#top')
```

```
=>["http",nil,"www.x.com","1234",nil,"/test/1.html",nil,
    "x=y&y=z","top"]
```

`URI.split`方法按顺序返回以下内容：模式、用户信息、主机名、端口号、注册信息、路径、不透明属性、查询以及片断。缺失元素均以`nil`表示。

注 用`URI.split`的唯一好处是，不需要创建URI对象，因此可以减少内存和处理器的使用。不过，一般人更愿意使用`URI()`或`URI.parse`，这样可以按名字指定不同元素，而不是必须按照数组中元素的顺序（对于不同版本的程序库，这个顺序可能有变化）。

创建URL

你还可以用uri创建合格的URL。最简单的方式是用URI针对每种协议的子类，传入一个由URL组成元素构成的散列表，来生成URL：

```
require 'uri'
u = URI::HTTP.build( :host => 'rubyinside.com', :path => '/' )
puts u.to_s
puts u.request_uri
```

```
http://rubyinside.com/
/
```

请注意，`to_s`方法返回整个URL，而`request_uri`则返回URL中主机名之后的部分。这是因为`net/http`等程序库会用到`request_uri`的数据，而`open-uri`等程序库则使用整个URL。

下面是创建FTP URL的例子：

```
ftp_url = URI::FTP.build( :userinfo => 'username:password',
  :host => 'ftp.example.com',
  :path => '/pub/folder',
  :typecode => 'a')
```

```
puts ftp_url.to_s
```

```
ftp://username:password@ftp.example.com/pub/folder?type=a
```

另外请注意，uri擅长以安全的方式调整URL，因此你可以读取和设置各种属性的值：

```
require 'uri'
my_url = "http://www.test.com/something/test.html"
url = URI.parse(my_url)
url.host = "www.test2.com"
url.port = 1234
puts url.to_s
```

```
http://www.test2.com:1234/something/test.html
```

16.16.3 更多信息

- uri程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/uri/rdoc/index.html>
- 关于URL和URI的更多信息：<http://en.wikipedia.org/wiki/URL>

16.17 zlib程序库

zlib是个开源的数据压缩程序库。zlib也是数据压缩领域的重要标准，几乎在每个操作系统平台上都可以操作zlib压缩文档。值得注意的是，zlib常常用于Web服务器和浏览器之间压缩网页，用于Linux内核，成为许多操作系统程序库的关键组成部分。

你可以在Ruby中使用zlib，作为压缩和解压缩数据的方法机制。

16.17.1 安装

zlib程序库是标准程序库的组成部分，由Ruby默认自带。要使用该程序库，只需把下面这行代码放在程序的开头位置附近：

```
require 'zlib'
```

16.17.2 示例

在zlib术语中，压缩和解压缩称为放瘪（deflating）和充胀（inflating）。最快的压缩（放瘪）数据的方法是直接使用Zlib::Deflate类：

```
require 'zlib'

test_text = 'this is a test string'*100
puts "Original string is #{test_text.length}bytes long"
compressed_text = Zlib::Deflate.deflate(test_text)
puts "Compressed data is #{compressed_text.length}bytes long"
```

```
Original string is 2100 bytes long
```

```
Compressed data is 46 bytes long
```

这段测试文本压缩效果极好，因为它由同一段字符串重复100次组成。不过，对于常规数据，更实际的压缩率在10%~50%之间。

恢复压缩数据需要Zlib::Inflate类：

```
require 'zlib'

test_text = 'this is a test string' * 100
puts "Original string is #{test_text.length} bytes long"
compressed_text = Zlib::Deflate.deflate(test_text)
puts "Compressed data is #{compressed_text.length} bytes long"
uncompressed_text = Zlib::Inflate.inflate(compressed_text)
puts "Uncompressed data is back to #{uncompressed_text.length} bytes in length"
```

```
Original string is 2100 bytes long
Compressed data is 46 bytes long
Uncompressed data is back to 2100 bytes in length
```

注 zlib返回的被压缩数据由若干8比特位数据组成，因此可能不适用于电子邮件或需要常规纯文本格式的场合。为了解决这一问题，你可以先用zlib压缩数据，然后用base64程序库把压缩结果转换成纯文本。

zlib还提供了可直接操作压缩文件的类。用zlib算法压缩的文件通常称为gzip文件，而用Zlib::GzipWriter和Zlib::GzipReader可以轻松创建、读入这些文件。

```
require 'zlib'

Zlib::GzipWriter.open('my_compressed_file.gz') do |gz|
  gz.write 'This data will be compressed automatically!'
end

Zlib::GzipReader.open('my_compressed_file.gz') do |my_file|
  puts my_file.read
end
```

```
This data will be compressed automatically!
```

16.17.3 更多信息

zlib程序库的标准文档：<http://www.ruby-doc.org/stdlib/libdoc/zlib/rdoc/index.html>

Ruby从入门到精通

Beginning Ruby: From Novice to Professional

作为极其流行的Ruby on Rails Web开发框架的底层引擎，Ruby已经广为人知，而它本身是一种极其强大的全能型编程语言。Ruby关注的焦点是减轻开发的负担，以及提供完全的面向对象环境。

本书是一本彻底而全面的最新指南，适合于各类Ruby读者，不管是编程初学者、Web开发人员，还是Ruby新手。本书从解说面向对象编程背后的原理开始，只通过几章的讲解，就构造出了真正的Ruby应用程序。

本书还讲解了Ruby关键内容（如类、对象、项目、模板和程序库）以及Ruby的其他方面（如数据库访问）。另外，本书深入介绍了Ruby on Rails。本书附录也提供了重要的参考信息，为经验丰富的程序员提供了Ruby快速入门。

附录部分请到华章网站（www.hzbook.com）下载。

作者简介：

Peter Cooper 是经验丰富的Ruby开发者和培训师，还是最流行的Ruby新闻博客“Ruby内幕”（<http://www.rubyinside.com/>）的编辑。在2007年以前，他主要做Ruby培训和开发，现在是Feed Digest网站（<http://www.feeditdigest.com/>）的全职开发者和所有者。

译者简介：

仲田 南京某软件公司项目经理，高级程序员、系统分析员，有多年软件开发与管理经验，从事过Delphi、J2EE、Rails应用开发，应用领域主要是企业管理应用，包括财务、审计、法律、商务、办公自动化等，目前正在研究Ruby语言和Rails框架。

Apress®

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

封面设计：王建敏



上架指导：计算机/程序设计

ISBN 978-7-111-25866-7



9 787111 258667

定价：59.00 元