

Programming Ruby 中文版

第2版

Programming Ruby: The Pragmatic Programmers' Guide, Second Edition

[美] Dave Thomas
Chad Fowler
Andy Hunt
著

孙 勇
姚延栋
张海峰
译



電子工業出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Second Edition
Includes Ruby
1.8

Programming Ruby 第2版 中文版

Ruby是一门非常强大而有用的语言，无论何时我都用它工作，这本书也总在我身边。

— Martin Fowler, 首席科学家, ThoughtWorks

Ruby是一门跨平台、面向对象的编程语言，它使编程这门手艺（craft）变得更容易了。Ruby体现了表达的一致性和简单性，它不仅是一门编程语言，更是表达想法的一种简练方式。Ruby支持“自然的智能（natural intelligence）。” — 本书的作者

欢迎来到PickAxe(镐头书，由封面上的工具得名)。这是一本关于Ruby的权威著作。

书中包含：

- 广受赞誉的Ruby使用教程；
- 完全的Ruby语言参考；
- 所有内建类、模块和方法的文档；
- 所有98个标准库的描述。

在现实中使用一把镐头是艰苦的工作，但是这本“镐头书”让您能够轻松地：

- 学习Ruby的基础 — 熟悉例如类、对象和异常等构成（construct），例如迭代器（对“越界（off-by-one）”的错误说再见）等高级特性， mixin（多继承的简化方式）以及线程；
- 使用CGI脚本、XML、SOAP和模板技术建立Web应用；
- 创建跨平台的GUI应用程序；
- 访问Microsoft Windows的自有API，并自动化Windows应用。

如果您阅读过第1版……

第2版有超过200页的新内容，以及对原有内容的大量修订，涵盖了Ruby 1.8中新的和改进的特性以及标准库模块。

新增和扩充的内容包括以下要点：

- 安装和打包；
- 文档化Ruby的源代码；
- 线程与同步；
- 使用C语言编写的扩展；
- 单元测试。

内建库的参考文档记录了从Ruby 1.6以来超过250个新增或改进的方法。

使用Ruby进行Web编程非常简单，并且第2版包括了关于XML/RPC、SOAP、分布式Ruby和Web模板系统的新内容。

Dave Thomas 是Ruby社区的一根顶梁柱，并且亲自负责Ruby许多创新性方向的探讨和开创工作。他和原来的合著者Andy Hunt是*Pragmatic Programmer*以及*Pragmatic Bookshelf*的创始人。Chad Fowler是Ruby Central, Inc. 的共同董事，并且始终是Ruby社区中活跃且起推进作用的力量。

您可以在www.pragmaticprogrammer.com上联系这些作者。

网上订购：www.dearbook.com.cn
第二书店 第一服务

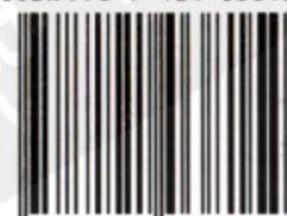


策划编辑：方 舟
责任编辑：陈元玉



图书分类：程序设计语言

ISBN 978-7-121-03815-0



9 787121 038150 >

定价：99.00元

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

Programming Ruby

中文版, 第2版

Programming Ruby: The Pragmatic Programmers' Guide, Second Edition

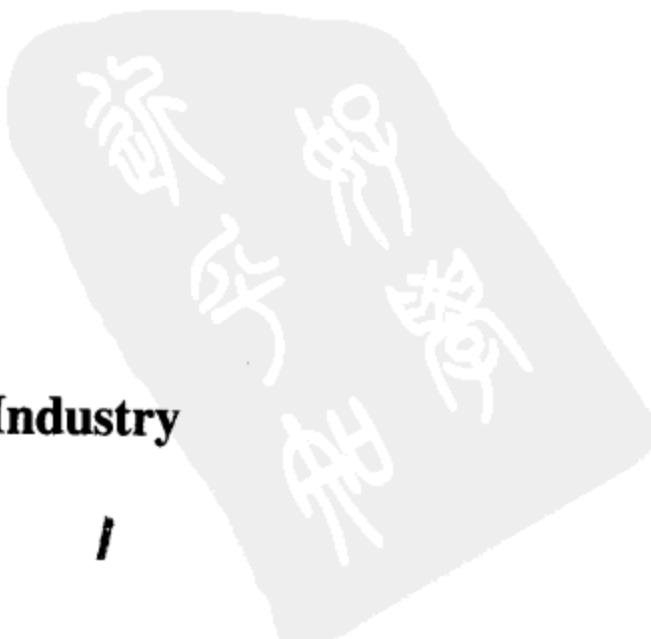
Dave Thomas
[美] Chad Fowler 著
Andy Hunt
孙 勇 姚延栋 张海峰 译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

1



内 容 简 介

Ruby 是一种跨平台、面向对象的动态类型编程语言。Ruby 体现了表达的一致性和简单性，它不仅是一门编程语言，更是表达想法的一种简练方式。它不仅受到广大程序员的欢迎，无数的软件大师亦为其倾倒。*Programming Ruby* 是关于 Ruby 语言的一本权威著作，也被称为 PickAxe Book（镐头书，由封面上的工具得名）。本书是它的第 2 版，其中包括超过 200 页的新内容，以及对原有内容的修订，涵盖了 Ruby 1.8 中新的和改进的特性以及标准库模块。它不仅是您学习 Ruby 语言及其丰富特性的一本优秀教程，也可以作为日常编程时类和模块的参考手册。

本书适合各种程度的 Ruby 程序员，无论新手还是老兵，都会从中得到巨大的帮助。

0-9745140-5-5 Programming Ruby: The Pragmatic Programmers' Guide, Second Edition by Dave Thomas, with Chad Fowler, Andy Hunt

All rights reserved. Authorized translation from the English language edition published by The Pragmatic Programmers, LLC.

本书简体中文专有翻译出版权由 The Pragmatic Programmers, LLC. 授予电子工业出版社未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2006-2241

图书在版编目 (CIP) 数据

Programming Ruby 中文版：第 2 版 / (美) 托马斯 (Thomas,D.), (美) 弗沃尔 (Fowler,C.), (美) 亨特 (Hunt,A.) 著；孙勇，姚延栋，张海峰译. —北京：电子工业出版社，2007.3

书名原文: Programming Ruby: The Pragmatic Programmers' Guide, Second Edition

ISBN 978-7-121-03815-0

I. P… II. ①托…②孙…③姚…④张… III. 计算机网络—程序设计 IV.TP393.092

中国版本图书馆 CIP 数据核字 (2007) 第 010837 号

策划编辑：方 舟

责任编辑：陈元玉

印 刷：北京市天竺颖华印刷厂

装 订：三河市金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：54.75 字数：1 000 千字

印 次：2007 年 3 月第 1 次印刷

定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：(010) 68279077；邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

世界各地的开发者对 *Programming Ruby* 和 Ruby 语言的评论

“Ruby 是一门非常强大而有用的语言，无论何时我都用它工作，而这本书也总在我身边。”

► Martin Fowler, 首席科学家, ThoughtWorks

“如果你的世界像我一样以 Java 为主，那么你需要这本杰出的书来学习所有你错过的精彩内容。只要看一眼，你的 Java 世界就会被撼动。实际上，读了仅仅几页 *Programming Ruby* 之后，再使用 Ruby 之外的语言编程感觉就像做无用功（push rope）。”

► Mike Clark, 作家、顾问

Ruby 是巧妙、优雅而有趣的语言，值得写一本巧妙、优雅而有趣的书。*Programming Ruby* 的第 1 版就是这样一本书，而第 2 版则更为出色。

► James Britt, 管理员, <http://ruby-doc.org>

“学习一门新的编程语言的原因是学习以不同的方式思考。学习以 Ruby 的方式去思考的最好方式是读 *Programming Ruby*。几年前我就是用本书的第 1 版这样做的。从那时起，我拥有了持续的令人愉快的 Ruby 编程体验。这源自学习此语言时所用的高质量的资源。据我所知，我并不是唯一一个声称每种语言都需要像这样一本高质量的书的人。”

► Chad Fowler, Ruby Central 公司董事

“这本书使我开始学习 Ruby。它至今仍旧是我遇到问题时转而求助的第一本书。”

► Ryan Davis, Seattle.rb 的创立者

“这本书改变了我的生活。听起来有点陈词滥调，但这是事实。使用了 6 年 Java，写了 300 000 行代码后，我需要一个转变了。当我读到本书的第 1 版时开始了这种转变。借助可靠的社区支持以及持续增长的强大类库基础，我创建了一个主要以应用 Ruby 解决现实问题来盈利的公司。Ruby 已经准备好步入黄金时段，而这本书的最新版将会向等待它的世界展示 Ruby 是一块多么瑰丽的宝石。”

► Rich Kilmer, InfoEther 有限责任公司首席执行官

“本书的第 1 版已经是我多年的桌旁伴侣。第 2 版将是它急切等待的替代者。”

► Tom Enebo, JRuby 开发者

“*Programming Ruby* 的第 1 版在日本之外得到了大规模的介绍，逐渐成为发布语言参考的事实标准，而且成了清晰、高效的科技写作再三引用的典型。扩充的第 2 版的出现令世界各地的 Ruby 程序员倍感兴奋，而且必定会引起学习这一优雅而又强大的语言的新浪潮。”

► David A. Black, 博士, Ruby Central 公司董事

“对我来说，Ruby 绝对是解决脚本和原型问题的首选，这本书将会帮助你发现它的用途和优雅之处。除此之外，阅读本书也是一种乐趣。”

► Robert Klemme

“当本书的第 1 版发布时我买了一本，用它来学习 Ruby 极其出色。于是我买了第二本放在家中。但是从那时起 Ruby 发生了很多变化。我很高兴看到 *Programming Ruby* 的第 2 版面世，它将再一次帮助程序员来学习这个奇妙而漂亮的语言。这不仅对 Ruby 新手来说是好消息，像我这样的 Ruby 开发者也想要一本（不，是两本），以便 Ruby 的所有最新细节都变得唾手可得。”

► Glenn Vanderburg, Countrywide Financial 软件架构师

Ruby 是一门可以花一个下午来学习使用而花多年（可能是一生）来深入掌握的伟大语言之一。在 C 中，我总是要解决语言的一些局限；而在 Ruby 中，我总是能发现解决问题的更简洁、更优雅的方式。*Programming Ruby* 是阐释 Ruby 语言最精彩的读物。它不仅教你语法，还教你语言的精神和对它的感觉。

► Ben Giddings

孔子说，“你听到，会忘记”。他还说，“你做了，会理解”。但实际上“做”并不容易，除非使用支持快速简洁原型设计的强大语言。对我来说，这种语言就是 Ruby！谢谢！”

► Michael Neumann

推荐序一

孟 岩

如果你想掌握 Ruby，这本书是最好的起点。如果你想运用 Ruby，这本书也是案头必备。所以，如果你已经决定要走入 Ruby 的世界，那么这本书是必经之路，而本不需要一篇“推荐序”。

问题在于，我们为什么还要学习一种新的语言？特别是当 Ruby 整体上仍然是一个没有完全成熟的“小语种”的时候，为什么要把宝贵的精力投入到 Ruby 中？这才是我想讨论的问题。

跟很多人一样，我学习程序设计是从 Basic 语言开始的。然而在初步了解了程序设计的基本概念之后，我便迅速地转向了 C 语言，并且在上面下了一番苦功夫。是 C 语言帮助我逐步理解了计算机系统以及算法、数据结构等基础知识，从而迈入程序设计的大门之中的。C 语言出色地实现了真实计算机系统的抽象，从而表现出极佳的适应性和强大的系统操控能力。直到今天，我仍认为认真理解和实践 C 语言是深入理解计算机系统的捷径。然而，当我尝试着用 C 语言来干点实事的时候，立刻发现一些问题：C 的抽象机制过于简单，容易犯错，开发效率低下，质量难以稳定。这时，有一位老师向我介绍了 Visual Basic，他盛赞 VB 是“最高级的开发语言”，代表未来的发展方向。正好当时有一个学校的项目摆在我面前，我几乎没有学习，凭着自己那点 Quick Basic 的底子，借助在线帮助，迅速地将项目完成，并且获得了好评。

通过这些初级的实践，我体会到 VB 在开发应用程序方面所具有的惊人的高效率，进而意识到，C 与 VB 是两类设计目标完全不同的语言。以 C 语言为代表的系统编程语言的优势在于能够充分发挥机器的性能，而像 VB 这样的语言的优势则是充分提高人的效率。事实上，执行性能与开发效率是软件开发中的一对矛盾，所有的程序设计语言都必须面对这个矛盾，作出自己的选择。

在当时，大多数新语言的选择是上下通吃。它们一方面提供了丰富多彩的高级抽象，另一方面又提供了强有力的底层操作能力，希望由此实现高性能与高效率的统一。C++、Java、C# 和 Delphi 都是走的这条路线，甚至 VB 从 5.0 开始也强化了底层操作机制，并提供了编译模型，不落人后。

我当时选择了 C++。对于熟悉 C 语言的我来说，这是一个自然而然的选择。深入学习 C++ 是一个漫长而艰苦的过程，不但要理解伴随面向对象和泛型而来的大量概念，牢记各种奇怪的语法规则，还要了解实践应用中大量存在的陷阱，掌握一系列“模式”、“惯用法”和“条例”。在克服这种种困难的过程中，我对程序设计的认识确实得到了强有力的提升，但是 C++ 真的实现了执行性能和开发效率的双丰收了吗？很遗憾，答案是“否”。我自己的体会是，使用 C++ 要花费大量精力来进行试探性设计，还可能要投入巨大精力来消除代码缺陷，因此开发效率不高。一些权威的调查显示，在新项目的执行中，C++ 的开发效率甚至低于 C，这不得不发人深思。C++ 提供了大量的语言机制，又存在一些陷阱，想要实现良好的运行性能，就必须遵守一系列清规戒律，这就带来了思想上的不自由，其结果往往是效率的低下。

随后流行的 Java 和 C# 等语言，有效地消除了 C++ 中存在的一些陷阱和缺陷，并且提供了很好的基础设施和开发环境，大大提高了开发效率。但是它们都设定了严整的结构和规则，进行严格的编译期检查，强调规范、纪律和计划。这些语言的拥护者们骨子里都认为，构造大规模软件是一个严肃的工程，必须施加强有力的约束和规范，时时刻刻预防和纠正开发者可能犯下的错误。当然，这样必然会导致对开发者思维的束缚和思考效率的限制，从而导致宏观上的低效。但他们认为，这是构造大型高质量软件所必须付出的代价。

然而，世界上另外有一些人的想法完全不同。他们认为，一种“可上九天揽月，可下五洋捉鳖”的终极语言即使能够实现，也注定是低效的。既然这个世界本身就是丰富多彩的，为什么不保留编程语言的多样性，各取所需，各得其所，彼此协同合作，不亦乐乎？既然 C 语言在充分发挥机器性能方面已经登峰造极，那么就应该尽力创造能够充分发挥人脑效能的程序设计语言。当人的效率充分提高的时候，所有的问题都会迎刃而解，或者至少大为简化。

1998 年，我读到一本薄薄的小书，叫做《Perl 入门》。这本书介绍了一种完全陌生的语言，不但看上去稀奇古怪，而且骨子里透露出来一股与 C++、Java 等完全不同的气质：狂放、不羁、乖张、散漫，无法无天。对于当时的我来说，这一切令人诧异，难以接受。而 Perl 的拥护者似乎也懒得挑战“主流意识形态”，他们自嘲地说，Perl 就是骆驼，样子不好看，气味也不好闻，但就是能干活。随后进入视野的 Python 外表温文尔雅，简朴工整，但其实质与 Perl 一样，也是崇尚自由灵活，追求简单直接。伴随着 Web 走来的 JavaScript 和 PHP，虽然外表上差别很大，但是总体上看，与 Perl、Python 一样，都是把自由放在结构之上，把人放在机器之上的语言。人们称它们为“动态语言”或者“脚本语言”。这些语言的出现和流行，强有力地挑战了过去 20 年来人们深信不疑的传

统观念。传统认为编译阶段的类型检查至关重要，可是动态语言却把这类检查推迟到执行阶段的最后一刻才进行；传统认为严整的结构和严肃的开发过程是必不可少的，可是动态语言却能够用一种看上去随意自由的风格去“堆砌”系统；传统把精致的模型和多层次的设计视为最佳实践，而动态语言却往往是蛮不讲理地直来直去；传统把频繁的变化和修改视为不良过程的标志，而动态语言却将此视为自然而然的工作方式；传统认为必须在机器性能与人的效率之间取得折中，动态语言却偏执地强调人的效率，而把机器性能这档子事情抛到九霄云外。动态语言离经叛道，却又大获成功，不由分说地把人们好不容易搭建起来的那个严谨的、精致的圣殿冲击得摇摇欲坠。人们怀着复杂的心情观察着动态语言的发展，猜度着它的方向和影响。

这时候我发现了 Ruby。

很多转向 Ruby 的人，在谈论这种语言的时候，都提到“乐趣”这个字眼，我也不例外。使用 Ruby 编程，你会体验到一种难以名状的趣味，这种感觉，就好像是突然掌握了十倍于从前的力量，同时又挣脱了长久以来一直束缚在身上的枷锁一般。“白日放歌须纵酒，青春作伴好还乡”，当年临摹“Hello World”时的淳朴热情仿佛又回到了身上。Ruby 实现了最纯粹意义上的面向对象，让 Smalltalk、Perl 和 Lisp 的灵魂在新的躯壳里高歌。相比于 Python，Ruby 的思想更加清晰一致，形式更加灵活；相比于 Perl，Ruby 更简单质朴，绝少光怪陆离之举；相比于 Smalltalk 和 Lisp，Ruby 更富有现代感和实干气质；相比于庙堂之上的“工业语言”，Ruby 自由挥洒、轻快锐利；而相比于 JavaScript 和 PHP，Ruby 从 Smalltalk 继承而来的深厚底蕴又大占优势。面对执行性能与开发效率的谜题，Ruby 毫不犹豫地选择了开发效率，选择了对人脑的友好。Ruby 的基本思想非常简单淳朴，对于基本原则的坚持非常彻底，毫不打折扣，而在具体应用中又集各家所长，实现了巧妙的平衡。从我自己的体验来讲，使用 Ruby 写程序，与使用 C++ 等主流语言感觉完全不同，没有战战兢兢的规划和设计，没有紧绷的神经，没有一大堆清规戒律，有的是一种闲庭信步的悠然，有的是时不时灵光一闪的洋洋自得，有的是抓住问题要害之后猛冲猛打的快感。我的水平还不高，还不能够设计更精妙的框架和 DSL（领域语言），但我相信我已经初步体会到了 Ruby 的乐趣。

自从这本书的第 1 版出版以来，Ruby 的发展越来越快。最初人们用 Ruby 来完成一些临时的任务，然后就迅速被它迷住了，越来越多有用和有趣的东西被开发出来，越来越多有才华的人加入了 Ruby 的阵营，从而形成了一股潜流。这潜流在 2003 年以后就已经出现了，只是还没有引起外部世界的注意。然而自 2005 年以后，潜流变成了潮流，越来越多的人被卷进来。整个社群围绕着 Ruby 的质朴与自由，创造了大量的珍宝（gems）。有人用 200 行 Ruby 代码写了一个飞快的全文搜索引擎，有人把(X)HTML 的

解析变成了 CSS Selector 的有趣复用，有人用 13 行 Ruby 代码完成了一个 P2P 文件共享程序，有人把 Google 引以为豪的 MapReduce 算法轻巧地实现在一个小小的 Ruby 程序库中。当然，最为人津津乐道的还是 Ruby on Rails 在 Web 开发领域掀起的风暴。所有这一切，都令人对 Ruby 报以越来越热烈的掌声。

今天，Ruby 已经成为世界上发展最快的程序设计语言之一，一个充满热情和创造力的社群围绕着它，开展着种种激动人心的工作。在这里没有什么豪言壮语，但是所有的工作都在扎实地推进，人们被自己内心的力量驱动着，而这种力量来自于 Ruby 质朴和自由的乐趣，它是近于纯粹的。

我深深地知道，Ruby 今天还不是“主流”，其前景究竟如何，也不是没有争论。在今天这个时代，选择技术路线是一件关乎生计和名利的事情，是不纯粹也不能纯粹的事情。在各种场合，年轻的或是年长的程序员们头头是道地分析着 IT 大局，掰着指头计算着技术的“钱”途，这些都无可厚非，甚至非常必要。但是，作为一个把程序设计当成自己的事业，或者至少是职业的人，总应该在自己的内心中留下那么一小片空间，在那里抛开一切功利，回归质朴，追求纯粹的编程的乐趣。如果你跟我一样，希望体会这样一种内心的乐趣，我热情地邀请你翻开这本书，加入 Ruby 社群。也请你相信，当越来越多的人为着自己的乐趣而共同努力的时候，我们就能创造历史。

孟 岩
《程序员》杂志技术主编
2006 年 12 月于北京

推荐序二

熊 节

根据我的观察，习惯于 Java 或者 C# 的程序员在初初接触 Ruby 时，最能打动他们的往往就是像本文标题这样的一句代码：原本熟悉的字符串或者整数突然摇身一变，有了很多新的行为，甚至让整个 Ruby 语言都似乎变了个样。尽管“改变标准库的行为”并不总是值得推荐的做法，但如果使用得当，你能够在 Ruby 的基础上创造出一种贴近项目需求、易写易读的方言——也有人把这些方言叫做“领域专用语言”（DSL，Domain Specific Language）。更多的程序员是因为 Rails 这个框架才开始对 Ruby 语言产生兴趣，而 Rails 在很大程度上正是一种针对 Web 应用开发的 DSL。

能够创造 DSL，这是 Ruby 语言最大的魅力之一。但仅仅这一点并不足以解释为何有那么多优秀的程序员如此盛赞 Ruby 语言，更不足以解释为何它会突然间红透半边天——毕竟，在元编程方面更具实力的 LISP 和 Smalltalk 并没有像如今的 Ruby 这样流行。作为一个 Java 程序员的 Mike Clark 给了我们一个有趣的比喻：推绳子——他说“读了仅仅几页 Programming Ruby 之后，再使用 Ruby 之外的语言编程感觉就像是在‘推绳子’（push rope）。”把一根软绵绵的绳子往前推，那种有劲使不上的感觉，正是用惯 Ruby 之后再回到 Java/C# 时的真实感受。

灵活、优雅、巧妙、便利……这些溢美之词我们已经听得太多了。但在我看来，Ruby 最大的特点就一个字：快。这不仅意味着你能够很快地为自己的问题找到现成的解决办法，更意味着你能够直观地描述自己心中的想法，并且在改变想法时能够很快地调整你的程序。这种能力对于今天的软件开发者而言显得尤为重要，因为世界在飞快地改变，软件项目的需求在飞快地改变。对于今天的软件客户来说，尽快得到可以工作的软件、尽快反馈、尽快看到调整的效果，比一个完美但尚未实现的设计要有价值得多。而 Ruby 这种“快速实现想法”的能力，正是众多开发者对之青睐有加的根源所在。

Ruby 能够帮你描述心中所想——这句话，在某种意义上，也意味着你需要熟悉 Ruby 的思考方式。尽管自称是面向对象的脚本语言，Ruby 的精神仍然与函数式编程（functional programming）一脉相承。这种精神不仅体现在语法层面上，还体现在构建系统的思路上。Ruby 社群很少会一开始就把要实现的目标想得清清楚楚，或是首先制定种

种规范标准；相反，他们会充分利用 Ruby 的灵活与简洁、优雅与巧妙，从一个简单的、能够工作的软件开始，逐步增加更多的功能，并通过不断重构和优化让良好的设计逐渐浮现。

是以，跨进 Ruby 的世界，也许你首先需要学会的是这种“渐进式”的思维方式——不仅仅是编写软件，就连“学习 Ruby 语言”本身也一样。你无须读 18 本书或者参加半年培训来学会 Ruby 编程——另一方面即便你这么做了也未必就能学会，如果你没有使用 Ruby 来编写真正有用的程序的话。所以，如果你对 Ruby 产生了兴趣，稍微了解一下，然后就开始写吧：把编写 shell 脚本的首选语言从 Perl 改为 Ruby，用 Rake 来构建你的项目，或者——像大多数人那样——用 Rails 来开发一个小网站。你会遇到无数的问题，解决这些问题的过程就是对你的技术进行重构的过程。

但你至少还需要通过某种途径来“稍微了解”Ruby 语言，而且在遇到问题时也需要一本手册来帮你排疑解难。在你手上的这本 *Programming Ruby* 正是为此而出的一本书。书中的精彩内容无须我在这里赘述，你大可以自己去发掘。我唯一想要告诉你的是：如果你想要开采最瑰丽的“红宝石”宝藏，这本书就是你不可或缺的“镐头”。锻造这柄镐头的是两位大名鼎鼎的“实用主义程序员”Dave Thomas 和 Andy Hunt，这两位撰写过一系列 C++/Java/.NET 技术图书的开发者最终选择用 Ruby on Rails 来开发他们自己的网站（PragmaticProgrammer.com），本身就已经证明了 Ruby 的价值，同时也让我们对本书的实用性更有信心。

所以，你还在犹豫什么呢？既然已经拿起了这本书，既然已经对 Ruby 产生了兴趣，就不要再浪费时间了。翻开书，跟着这两位讲求实效的作者一道，现在就开始你的 Ruby 编程之旅吧。Ruby 已经向你说过“hello”了，你将会如何回应它呢？

最后，和以往一样，祝你在 Ruby 的世界里，编程快乐！

熊 节

ThoughtWorks 咨询师

2006 年 12 月于西安

译序

关于 Ruby 语言及相关技术，非常感谢孟岩兄和熊节兄在前面所做的精辟入里的分析与推荐，这里就不再赘述了。相信您读了之后，一定已经怦然心动而跃跃欲试了。

借此机会，我们还要感谢许多人。

首先要感谢博文视点公司引进此书，并将如此重要的一部书交托给我们来翻译，希望能幸不辱命。也要感谢本书的两位编辑方舟和陈元玉，没有二位认真、辛苦的工作，本书不可能有现在的翻译质量。我们也从他们的技术和文字校订中学到了很多。

另外译者也要相互感谢一下。在本书翻译的四个月期间，姚延栋刚做了父亲，张海峰也是家有幼子，并开始了新的职业征途，但两位都竭力保证了翻译的进度和质量。孙勇则不辞辛苦地对全书进行了统稿。当然，一定要感谢家人对我们的宽容和理解。

由于译者对 Ruby 语言也是新学，水平有限，难免在译稿中存有这样那样的错误。如果您有技术或文字方面的问题，欢迎致信 progruby.cn@gmail.com，我们会尽力帮您解答。

译者

2006 年 12 月于北京

第 1 版序

Foreword to the First Edition

人受创造之驱动；我清楚自己非常喜欢创造。尽管我不善于油画、素描或者音乐，但是我可以写软件。

在接触计算机后不久，我就对编程语言产生了浓厚的兴趣。我坚信一定可以实现一种理想的编程语言，而我希望能成为其设计者。后来在积累了一些经验之后，我意识到设计这种理想的、通用的语言比我想象的要难得多，但是我依旧希望能设计一种能解决我日常遇到的大多数问题的语言，这是我学生时代的梦想。

多年后，我和同事谈起了脚本语言以及它们的处理能力和潜力。作为一个超过 15 年的面向对象的爱好者，我认为面向对象编程也非常适合脚本语言。我在网络上做了一段时期的调查，但是我找到的候选，例如 Perl 和 Python，都不是我真正需要的。我需要一个比 Perl 更强大、比 Python 更面向对象的语言。

后来，我想起了以前的梦想，于是决定自己设计一种语言。起初我只是在工作时用来自娱。但是，逐渐地它成长为一个足以替代 Perl 的工具。我命名为 Ruby —— 先前的名字叫 Red Stone —— 并在 1995 年公开发布。

从那时起，越来越多的人开始对 Ruby 感兴趣。不管你相信与否，现在 Ruby 在日本实际上已经比 Python 更流行了。我希望最终它能在全世界被广泛地接受。

我认为生活的目的是快乐，至少部分是快乐。基于这种信念，Ruby 被设计成一种使编程不但容易而且有趣的语言。它能让你面对更少的压力，集中精力于程序设计中的有创造性的一面。如果你不相信我，请阅读此书并试用一下 Ruby。我确信你自己会体会到这一点。

我非常感谢那些参加 Ruby 社区的人；他们给了我莫大的帮助。我总是觉得 Ruby 是我的一个孩子，但实际上它是许多人共同努力的结晶。没有他们的帮助，Ruby 不可能发展成现在这样。

我要特别感谢此书的作者，Dave Thomas 和 Andy Hunt。Ruby 一直不是一个文档齐全的语言。因为我喜欢写程序胜于写文档，所以 Ruby 手册没有应有的那样完善。你必须通过阅读源代码来获悉语言的准确行为。但是现在 Dave 和 Andy 为你做了这一工作。

他们对来自远东的一个不怎么出名的语言产生了兴趣。他们研究它，阅读了成千上万行源代码，编写了不计其数的测试脚本和电子邮件，澄清了语言的某些不明确的行为，发现了许多 bug（甚至修正了其中的许多），并最终编写了这本出色的书。毫无疑问，Ruby 因此已经成了一个文档完善的语言。

他们在本书上花的精力绝不是微不足道的。当他们在写此书时，我对语言本身进行了一些修改。我们一起为这些更新而工作，并使此书尽量准确。

我希望 Ruby 和本书能使你编程更容易和有趣。工作得开心！

Yukihiro Matsumoto, a.k.a. “Matz”

まつもとゆきひろ

2000 年 10 月于日本

第 2 版序

Foreword to the Second Edition

在 1993 年，没有人会相信一个由日本业余语言设计者创造的面向对象语言能最终在世界范围内被广泛使用并变得几乎像 Perl 那样流行。这种想法被认为是极其愚蠢的，我承认这一点，我也不相信。

但是这确实发生了，它大大超出了我的期望。这是——至少部分是——由本书的第 1 版引起的。著名的 *Pragmatic Programmers* 选择了一个在日本之外没有名气的动态语言，并写了一本关于它的好书。奇迹就这样发生了。

那都是过去的事情了，未来将从现在开始。我们有了 *Programming Ruby* 的第 2 版，它比第 1 版更好。这已不再是奇迹。这次，持续增长的 Ruby 社区帮助编写了此书。而我只须参与社区一起工作。

我真的很感谢 Pragmatic Programmers, Dave Thomas 和 Andy Hunt 以及来自社区的其他帮助编写此书的人（抱歉不能一个个列出）。我非常热爱和睦的 Ruby 社区。这是我所见过的最好的软件社区之一。我也非常感谢全世界使用 Ruby 的每一位程序员。

宝石已经开始旋转，它的光芒将闪耀整个地球。

Yukihiro Matsumoto, a.k.a. “Matz”

まつもとゆきひろ

2004 年 8 月于日本

前言

Preface

本书是 *Programming Ruby* 的第 2 版，*Programming Ruby* 已广为 Ruby 爱好者所知。它是 Ruby 编程语言的教程和参考文献。如果你有本书的第 1 版，你会发现这一版有了重大变化。

当 Andy 和我写第 1 版的时候，我们不得不介绍 Ruby 的背景和吸引力。其中我们写道：“当发现 Ruby 时，我们意识到找到了一直在寻找的东西。和我们曾经用过的任何语言相比，Ruby 都称得上是罕见的。你可以集中精力于解决手头上的问题而不是与编译和语言本身周旋。这就是它使你成为一位更好的程序员的方法：通过让你将时间花在为用户创建解决方案上而不是编译上。”

今天这种信念变得更强了。4 年后，Ruby 仍旧是我们的选择：我用它来开发客户端程序，用它来运行我们的出版商务系统，还用它来做所有使系统运行更流畅的小程序。

在这 4 年中，Ruby 发展迅速。大量的方法被加入内建的类和模块，标准库（那些包含在 Ruby 发行版中的库）也有了重要发展。现在社区有了标准的文档系统（RDoc），而 RubyGems 也成了为打包发行 Ruby 代码的系统选择。

这些改变是令人激动的，但是这也使得本书第 1 版变得有点过时。第 2 版弥补了这一缺陷：像第 1 版一样，它是基于 Ruby 的最新版而编写的。

Ruby 版本 Ruby Versions

本书阐述的是 Ruby 1.8（特别涵盖了集成入 Ruby 1.8.2 的改变）。¹

¹ Ruby 版本号遵循其他许多开源项目的模式。子版本号是偶数的版本（1.6, 1.8 等），也是稳定的公开发布版。这些版本的 Ruby 被预先打好包并放在各种各样的 Ruby 网站上以供下载。软件开发版的子版本号是奇数，例如 1.7 和 1.9。对于这种版本，你必须用本书第 4 页介绍的方法来下载并自己编译。

用来写本书的 Ruby 版本到底是什么？让我们来问 Ruby 吧。

```
% ruby -v  
ruby 1.8.2 (2004-12-30) [powerpc-darwin7.7.0]
```

这演示了重要的一点。你在本书中看到的大多数代码例子都在我排版本书时被实际执行过。当你看到一个程序的输出时，该输出是通过实际运行代码并将结果插入到本书而成的。

本书的变动

Changes in the Book

和第 1 版相比，除了更新到 Ruby 1.8，你还会发现一些其他的变化。

在本书的前半部分，我新加了 6 章。与第 1 版相比，入门一章对设置和运行 Ruby 做了更完整的介绍。第二个新加入的单元测试章反映了 Ruby 爱好者对测试越来越重视。第三个新的章节涵盖了 Ruby 程序员使用的三个工具：用来体验 Ruby 的 *irb*，用来给代码写文档的 *RDoc* 以及用来打包发布代码的 *RubyGems*。新添的最后一章介绍了 *duck typing*，这种编程哲学和 Ruby 背后的思想非常一致。

新加入的还不止这些。你还会发现线程一章大大扩展了对同步的讨论，而编写 Ruby 扩展一章的大部分也被重写了。关于网页编程一章讨论了一个可选的模板系统并加入了 SOAP 一节。语言参考一章也被大大扩展了（特别是关于 *block*、*procs*、*break* 和 *return* 新规则的部分）。

本书的后半部分对内建的类和模块进行了介绍，该部分包含的重要改动超过了 250 处。其中很多是方法的更新，一些是更新过时的老方法，而另一些是具有新行为的方法。你还会发现加入了许多对新模块和类的介绍。

最后，本书用了一节来介绍标准库。自 Ruby 1.6 以来标准库有了极大的发展，现在它的内容如此之多，以至于没有上千页篇幅根本就不可能对之进行较为详细的介绍。同时，Ruby Documentation 项目正在忙于为库代码添加 RDoc 文档（第 199 页的第 16 章介绍了 RDoc）。这意味着你可以使用 Ruby 发行版自带的 *ri* 工具来获得关于库模块的精确

的、最新的文档。综合上述原因，我决定改变介绍库文档的风格——现在该部分是可用库的一个路线图，显示了一些代码样例并介绍了大致用法。我把底层的细节留给了 RDoc。

贯穿本书，我使用页边空白中的一个小符号，就像这里的这个一样，来尽量标明 1.6 和 1.8 之间的变化。没变的一点是：在本书的正文中当谈论到作者时我仍旧使用我们一词。里面的很多词都源自第 1 版，我当然不想侵占 Andy 在那本书上的工作成果。

总之，本书是对第 1 版的重大更新。我希望你觉得它有用。

资源 Resources

访问 Ruby 网站 <http://www.ruby-lang.org>，看看有什么新内容。在新闻组和邮件列表中（参见附录 C）可以和其他 Ruby 用户进行讨论。

而且我也非常希望能得到你的反馈。欢迎任何评论，提出建议，以及对书中的错误和例子中的问题给予指正。请发邮件到：

rubybook@pragmaticprogrammer.com

如果你告诉我们书中的错误，我会将它们加入到位于以下地址的勘误表中：

<http://www.pragmaticprogrammer.com/titles/ruby/errata.html>

从下面的地址中你会发现书中几乎所有样例代码的源代码链接：

<http://www.pragmaticprogrammer.com/titles/ruby>

致谢 Acknowledgments

我曾在 Ruby 邮件列表中询问是否有人愿意帮助审查此书的第 2 版。几乎有一百名志愿者响应了我的请求。为了便于管理，我不得不依据先来先得的原则限制人数。即使这样，那些出色的审查员还是给了我超过 1.5MB 的建议。他们指出了许多问题，从放错地方的逗号到遗漏的方法。我不可能获得比这更好的帮助了，所以非常“感谢” Richard Amacker, David A. Black, Tony Bowden, James Britt, Warren Brown, Mike Clark, Ryan Davis（感谢那个日文 PDF!），Guy Decoux, Friedrich Dominicus, Thomas Enebo, Chad Fowler, Hal Fulton, Ben Giddings, Johan Holmberg, Andrew Johnson, Rich Kilmer, Robert Klemme, Yukihiro

Matsumoto, Marcel Molina Jr., Roeland Moors, Michael Neumann, Paul Rogers, Sean Russell, Hugh Sasse, Gavin Sinclair, Tanaka Akira, Juliet Thomas, Glenn Vanderburg, Koen Vervloesem 和 Austin Ziegler.

Chad Fowler 写了 RubyGems 一章。实际上，他写了两次。写第一次的时候他在欧洲休假，不幸在回家的路上，他的 Powerbook 被偷了，而他所写的书稿也丢失了，所以回去以后，他只好又坐下来重写了一遍。对此我感激不尽。

Kim Wimpsett 做了编辑这一费力不讨好的工作。她做了大量的工作（而且超过了记录），而本书中行业术语的数量和我对语言的组织能力之差使得这项任务更加困难。Ed Giddens 设计了出色的封面，该封面极好地混合了新老封面。谢谢他们！

最后，我仍想对 Ruby 的创建者 Yukihiro “Matz” Matsumoto 致以深深的谢意。在这段成长和变化的时间内，他一直以快乐和专一的精神来改进 Ruby 语言。Ruby 社区的友善和开发精神是他个人精神的直接体现。

谢谢各位。Domo arigato gozaimasu.

Dave Thomas
THE PRAGMATIC PROGRAMMERS
<http://www.pragmaticprogrammer.com>

符号约定

Notation Conventions

在本书中，我们使用如下排版符号。

代码样例使用等宽字体显示。

```
class SampleCode
  def run
    ...
  end
end
```

在正文中，`Fred#do_something` 是对 `Fred` 类的实例方法（这里是 `do_something`）的引用，`Fred.new2` 是一个类方法，而 `Fred::EOF` 是一个类常量。使用井号符来表示实例方法的决定是很棘手的：它不是合法的 Ruby 语法，但是我们认为区分一个类的实例方法和类方法是很重要的。当你看到 `File.read` 时，你知道我们在讨论类方法 `read`。而当你看到 `File#read` 时，我们在引用实例方法 `read`。

本书包含了很多 Ruby 代码片断。如果可能，我们会尽量显示运行它们的结果。一个最简单的例子，我们将表达式的值和表达式显示在同一行上。例如：

```
a = 1
b = 2
a + b → 3
```

在这里，你能看到 `a + b` 的计算结果 `3` 显示在了箭头的右边。注意如果你只是运行这个程序，那么你不会看到输出结果 `3`——你需要使用类似 `puts` 这样的方法将其输出。

有时我们对赋值语句的值感兴趣，这种情况下我们也会把它显示出来。

```
a = 1 → 1
b = 2 → 2
a + b → 3
```

如果程序输出更复杂，那么我们将其显示在程序代码下面。

```
3.times { puts "Hello!" }
```

输出结果：

```
Hello!
Hello!
Hello!
```

² 在其他 Ruby 文档中，你可能会看到类方法被写为 `Fred::new`。这是完全合法的 Ruby 语法；我们只是认为 `Fred.new` 更易于阅读。

在某些库文档中，我们需要在输出中显示空格。你会看到空格被表示成“”字符。

命令行调用使用等宽字体表示，而参数使用斜体表示。可选元素被放在中括号中。

```
ruby [flags ...] [progname] [arguments ...]
```



路线图

Road Map

本书主要包括 4 个部分，每一部分有其不同的侧重点，阐述了 Ruby 语言的不同方面。

第 1 部分 *Facets of Ruby* 中有一个 Ruby 教程。它首先介绍了如何在你的系统上运行 Ruby，然后用一个短小的章节讲解了 Ruby 特有的术语和概念，该章也包含了足够的基础知识以帮助理解其他章节。教程的后面对语言由上到下做了介绍。其中我们讨论了类、对象、类型、表达式和语言的其他一些特性。最后介绍了单元测试，以帮助你解决实际遇到的问题。

Ruby 很突出的一点是它和环境的集成程度。第 2 部分 *Ruby in Its Setting* 探讨了这些方面。在这里你会发现使用 Ruby 的很多实用信息：使用解释器选项、使用 irb、文档化 Ruby 代码以及打包 Ruby 以方便使用。本部分也包含了一些使用 Ruby 解决常用任务的教程：使用 Ruby 处理网页；使用 Tk 创建 GUI 应用；在 Microsoft Windows 环境中（包括本机系统 API 调用，COM 集成和 Windows 自动化）使用 Ruby。通过本部分你还会发现扩展 Ruby 和在自己的代码中嵌入 Ruby 是多么容易的事。

第 3 部分 *Ruby Crystallized* 包含了更高级的话题。这里介绍了语言的所有细节，*duck typing* 的概念、元类模型、tainting、reflection 和 marshaling。第一次读这部分时你可能读得很快，但是我们相信一旦你真正开始使用 Ruby，你会回来再读这一部分。

第 4 部分是 *Ruby Library Reference*。^{1.8} 这部分内容很多，首先介绍了 48 个内建的类和模块中 950 多个方法（超过了第 1 版中介绍的 40 个类和模块中的 800 个方法）。之后又介绍了包含在标准 Ruby 发行版中的一些库模块（其中的 98 个）。

那么我们该如何来阅读本书呢？根据你的编程水平，特别是 OO 编程水平，你可能只想读本书的某些部分。下面是我们的建议。

如果你是初学者，可能需要从第 1 部分的教程开始。当开始写程序时把库参考放在身边。熟悉像 Array、Hash 和 String 这样的基本类。当你对 Ruby 环境更熟悉后，可

能想去研究一下第 3 部分的某些高级话题。

如果你已经对 Perl、Python、Java 或 Smalltalk 比较熟悉，那么我们建议你先读第 1 章的第 3 页，那里我们讨论了如何安装和运行 Ruby，然后阅读第 2 章。从那里，你可以采用较慢的方法顺序阅读后面的部分，也可以跳到第 3 部分的详细语言介绍，最后阅读第 4 部分的库参考。

专家、高手和那些“不需要烂教程”的人可以直接跳到第 317 页的第 22 章的语言参考，撕去库参考，然后就可以用本书来做（相当吸引人的）咖啡垫了。

当然你也完全可以从头开始，一页一页地按你的方式来阅读。

另外不要忘记，当你遇到无法解决的问题时还可求助于“帮助”。关于使用“帮助”的详细信息请参见第 783 页的附录 C。

目录

Contents

第1版序	xv
第2版序	xvii
前言	xiv
路线图	xxxv
第1部分 Ruby面面观	1
第1章 入门	3
1.1 安装 Ruby	3
1.2 运行 Ruby	5
1.3 Ruby 文档: RDoc 和 ri	8
第2章 Ruby.new	11
2.1 Ruby 是一门面向对象语言	11
2.2 Ruby 的一些基本知识	13
2.3 数组和散列表	16
2.4 控制结构	18
2.5 正则表达式	19
2.6 Block 和迭代器	21
2.7 读/写文件	23
2.8 更高更远	24
第3章 类、对象和变量	25
3.1 继承和消息	27
3.2 对象和属性	29
3.3 类变量和类方法	33
3.4 访问控制	37
3.5 变量	39

第 4 章 容器、Blocks 和迭代器.....	43
4.1 容器.....	43
4.2 Blocks 和迭代器.....	49
4.3 处处皆是容器.....	57
第 5 章 标准类型.....	59
5.1 数字.....	59
5.2 字符串.....	61
5.3 区间.....	66
5.4 正则表达式.....	68
第 6 章 关于方法的更多细节.....	79
6.1 定义一个方法.....	79
6.2 调用方法.....	81
第 7 章 表达式.....	87
7.1 运算符表达式.....	88
7.2 表达式之杂项.....	89
7.3 赋值.....	90
7.4 条件执行.....	93
7.5 Case 表达式.....	98
7.6 循环.....	100
7.7 变量作用域、循环和 Blocks.....	105
第 8 章 异常，捕获和抛出	107
8.1 异常类.....	107
8.2 处理异常.....	108
8.3 引发异常.....	112
8.4 捕获和抛出.....	114
第 9 章 模块.....	117
9.1 命名空间.....	117
9.2 Mixin.....	118
9.3 迭代器与可枚举模块.....	120
9.4 组合模块.....	120
9.5 包含其他文件.....	123
第 10 章 基本输入和输出	127
10.1 什么是 IO 对象	127
10.2 文件打开和关闭.....	128

10.3 文件读写.....	129
10.4 谈谈网络.....	133
第 11 章 线程和进程	135
11.1 多线程.....	135
11.2 控制线程调度器.....	140
11.3 互斥.....	141
11.4 运行多个进程.....	147
第 12 章 单元测试.....	151
12.1 Test::Unit 框架.....	152
12.2 组织测试.....	156
12.3 组织和运行测试.....	159
第 13 章 当遇到麻烦时.....	163
13.1 Ruby 调试器.....	163
13.2 交互式 Ruby	164
13.3 编辑器支持.....	165
13.4 但是它不运作.....	167
13.5 然而它太慢了.....	170
第 2 部分 Ruby 与其环境.....	175
第 14 章 Ruby 和 Ruby 世界.....	177
14.1 命令行参数.....	177
14.2 程序终止.....	180
14.3 环境变量.....	181
14.4 从何处查找它的模块.....	182
14.5 编译环境.....	183
第 15 章 交互式 Ruby Shell	185
15.1 命令行.....	185
15.2 配置.....	190
15.3 命令.....	194
15.4 限制.....	196
15.5 rtags 与 xmp.....	196
第 16 章 文档化 Ruby	199
16.1 向 Ruby 代码中添加 RDoc.....	199
16.2 向 C 扩展中添加 RDoc.....	207

16.3 运行 RDoc	211
16.4 显示程序用法信息.....	212
第 17 章 用 RubyGems 进行包的管理	215
17.1 安装 RubyGems.....	216
17.2 安装程序 Gems	216
17.3 安装和使用 Gem 库.....	218
17.4 创建自己的 Gems	223
第 18 章 Ruby 与 Web	235
18.1 编写 CGI 脚本.....	235
18.2 Cookies.....	244
18.3 提升性能.....	247
18.4 Web 服务器的选择	247
18.5 SOAP 及 Web Services	249
18.6 更多信息.....	253
第 19 章 Ruby Tk.....	255
19.1 简单的 Tk 应用程序	255
19.2 部件.....	256
19.3 绑定事件.....	260
19.4 画布.....	261
19.5 滚动.....	263
19.6 从 Perl/Tk 文档转译.....	265
第 20 章 Ruby 和微软 Windows 系统.....	267
20.1 得到 Ruby for Windows	267
20.2 在 Windows 下运行 Ruby.....	268
20.3 Win32API	268
20.4 Windows 自动化	269
第 21 章 扩展 Ruby.....	275
21.1 你的第一个扩展.....	275
21.2 C 中的 Ruby 对象	278
21.3 Jukebox 扩展	284
21.4 内存分配.....	293
21.5 Ruby 的类型系统.....	294
21.6 创建一个扩展.....	296
21.7 内嵌 Ruby 解释器.....	301

21.8 将 Ruby 连接到其他语言	304
21.9 Ruby C 语言 API	305
第 3 部分 Ruby 的核心	315
第 22 章 Ruby 语言	317
22.1 源代码编排	317
22.2 基本类型	319
22.3 名字	328
22.4 变量和常量	330
22.5 表达式	338
22.6 方法定义	345
22.7 调用方法	348
22.8 别名	351
22.9 类定义	352
22.10 模块定义	354
22.11 访问控制	356
22.12 Blocks, Closures 和 Proc 对象	356
22.13 异常	360
22.14 Catch 和 Throw	362
第 23 章 Duck Typing	365
23.1 类不是类型	366
23.2 像鸭子那样编码	370
23.3 标准协议和强制转换	371
23.4 该做的做，该说的说	377
第 24 章 类与对象	379
24.1 类和对象是如何交互的	379
24.2 类和模块的定义	387
24.3 顶层的执行环境	393
24.4 继承与可见性	393
24.5 冻结对象	394
第 25 章 Ruby 安全	397
25.1 安全级别	398
25.2 受污染的对象	399
第 26 章 反射, ObjectSpace 和分布式 Ruby	403
26.1 看看对象	404

26.2 考察类.....	405
26.3 动态地调用方法.....	407
26.4 系统钩子.....	410
26.5 跟踪程序的执行.....	412
26.6 列集和分布式 Ruby.....	414
26.7 编译时？运行时？任何时候.....	419
第 4 部分 Ruby 库的参考.....	421
第 27 章 内置的类和模块.....	423
27.1 字母顺序列表.....	424
Array	427
Bignum.....	441
Binding.....	444
Class.....	445
Comparable.....	447
Continuation.....	448
Dir	449
Enumerable	454
Errno.....	460
Exception	461
FalseClass.....	464
File	465
File::Stat	477
FileTest.....	483
Fixnum	484
Float	487
GC	491
Hash	492
Integer	501
IO	503
Kernel.....	516
Marshal.....	535
MatchData.....	537
Math	540
Method	543
Module	545
NilClass.....	561
Numeric.....	562

Object	567
ObjectSpace	578
Proc	580
Process	583
Process::GID	589
Process::Status	591
Process::Sys	594
Process::UID	596
Range	597
Regexp	600
Signal	604
String	606
Struct	626
Struct::TMS	630
Symbol	631
Thread	633
ThreadGroup	640
Time	642
TrueClass	650
UnboundMethod	651
第 28 章 标准库	653
Abbrev	655
Base64	656
Benchmark	657
BigDecimal	658
CGI	659
CGI::Session	661
Complex	662
CSV	663
Curses	664
Date/DateTime	665
DBM	666
Delegator	667
Digest	668
DL	669
dRuby	670
English	671
Enumerator	672

erb.....	673
Etc	675
expect.....	676
Fcntl	677
FileUtils.....	678
Find	679
Forwardable	680
ftools.....	681
GDBM.....	682
Generator.....	683
GetoptLong	684
GServer	685
Iconv.....	686
IO/Wait.....	687
IPAddr.....	688
jcode	689
Logger	690
Mail	691
mathn.....	692
Matrix.....	694
Monitor.....	695
Mutex	696
Mutex_m	697
Net::FTP.....	698
Net::HTTP.....	699
Net::IMAP.....	701
Net::POP	702
Net::SMTP	703
Net::Telnet	704
NKF.....	705
Observable	706
open-uri	707
Open3	708
OpenSSL	709
OpenStruct	710
OptionParser.....	711
ParseDate	713
Pathname	714

PP	715
PrettyPrint	716
Profile	717
Profiler_	718
PStore	719
PTY	720
Rational	721
readbytes	722
Readline	723
Resolv	724
REXML	725
Rinda	727
RSS	728
Scanf	729
SDBM	730
Set	731
Shellwords	732
Singleton	733
SOAP	734
Socket	735
StringIO	736
StringScanner	737
Sync	738
Syslog	740
Tempfile	741
Test::Unit	742
thread	743
ThreadsWait	744
Time	745
Timeout	746
Tk	747
tmpdir	748
Tracer	749
TSort	750
un	751
URI	752
WeakRef	753
WEBrick	754

Win32API	755
WIN32OLE	756
XMLRPC	757
YAML	758
Zlib	759
第 5 部分 附录	761
附录 A Socket 库	653
附录 B MKMF 参考	779
附录 C 支持	783
附录 D 书目	787
索引 (Index)	789



表 目 录

List of Tables

表 2.1 变量和类名称样例.....	17
表 5.1 字符类缩写	72
表 7.1 常用的比较操作符.....	95
表 11.1 在一个竞态条件中的两个线程.....	143
表 13.1 调试器命令	173
表 14.1 Ruby 使用的环境变量.....	182
表 15.1 irb 命令行选项.....	187
表 17.1 版本操作符	218
表 18.1 erb 的命令行选项.....	243
表 21.1 C/Ruby 数据类型转换函数和宏.....	280
表 22.1 常规分隔输入.....	318
表 22.2 双引号字符串中允许的替换	321
表 22.3 保留字.....	329
表 22.4 Ruby 操作符（优先级从高到低）	339
表 25.1 安全级别的定义	401
表 27.1 Array#pack 的模板字符	435
表 27.2 匹配模式常量	468
表 27.3 路径分隔符常量（和平台相关）	470
表 27.4 open 模式常量	472
表 27.5 锁模式常量	476

表 27.6 模式字符串	504
表 27.7 sprintf 标志字符	531
表 27.8 sprintf 域类型	532
表 27.9 参数只有一个的文件测试	533
表 27.10 参数有两个的文件测试	533
表 27.11 定义在 Numeric 类以及其子类中的方法。打勾意味着 这个方法定义在相应的类中	564
表 27.12 模 (modulo) 和余数 (remainder) 之间的区别。 模运算符 (“%”) 总是有除数 (divisor) 的符号， 然而 remainder 有被除数 (dividend) 的符号	565
表 27.13 替换字符串中的反斜线序列	614
表 27.14 String#unpack 指令	624
表 27.15 Time#strftime 指令	648
表 28.1 ERB 的指令	674
表 28.2 选项定义参数	712

图目录

List of Figures

图 3.1 变量保存对象引用.....	41
图 4.1 如何对数组进行索引操作.....	45
图 8.1 Ruby 异常层次结构.....	109
图 12.1 产生罗马数字（有 bug）.....	153
图 13.1 irb 会话的示范.....	166
图 13.2 使用 benchmark 比较变量访问代价	171
图 16.1 浏览类 Counter 的 RDoc 输出.....	200
图 16.2 浏览带注释的源代码的 RDoc 输出.....	201
图 16.3 使用 ri 来读取文档	202
图 16.4 由 Rdoc/ri 生成的 Proc 类的文档	203
图 16.5 用 RDoc 文档化的 Ruby 源文件.....	208
图 16.6 用 RDoc 文档化的 C 源文件.....	210
图 16.7 使用 RDoc::usage 的范例程序.....	213
图 16.8 由范例程序产生的帮助信息	214
图 17.1 MomLog 包结构.....	232
图 18.1 简单的 CGI 表单	238
图 18.2 使用 erb 处理带有循环的文件.....	245
图 19.1 在 Tk 的 Canvas (画布) 上绘制	262
图 21.1 将 C 数据类型包装为对象	286
图 21.2 构建扩展的过程	297

图 22.1 boolean 区间 (range) 的状态变迁	342
图 24.1 一个基本对象, 以及它的类和超类	380
图 24.2 添加 Guitar 的 metaclass	381
图 24.3 向对象中添加一个虚拟类	384
图 24.4 被包含的模块以及它的代理类	386
图 27.1 标准异常等级结构	462
图 27.2 实际运行 Method#arity	544



第1部分 Ruby面面观

Part I Facets of Ruby



Programming Ruby 中文版, 第2版

Getting Started

在开始讨论 Ruby 语言之前，我们将先帮助你在计算机上运行 Ruby，这对学习它是非常有益的。那样你就可以一边学习一边试运行样例代码和你自己的代码。我们也会演示运行 Ruby 的几种不同方式。

1.1 安装 Ruby

Installing Ruby

通常你根本不需要自己下载 Ruby。许多 Linux 发行版已经预装了 Ruby，Mac OS X 也预装了 Ruby（尽管 OS X 预装的 Ruby 的次版本号一般比 Ruby 的最新版本老一些）。试着在命令行提示符下输入 `ruby -v`，可能会有意外的惊喜。

如果你的系统上还没有安装 Ruby，或者想更新它到较新的版本，那么你可以很方便地安装它。但是首先需要作出一个选择：是使用二进制发行版，还是从源码编译？

1.1.1 二进制发行版

Binary Distributions

Ruby 的二进制发行版很容易安装，一旦安装就可以运行。二进制发行版是为特定操作系统环境而预先编译的，如果你不想为从源代码编译浪费时间，它是非常方便的。二进制发行版的不足之处是你只能用它提供的东西：它可能比最新的代码低一个或两个次版本号，也可能不包含你需要的可选的软件包。如果你能容忍这些不足，那么你还需要找到适合你的操作系统和机器体系结构的二进制发行版。

对基于 RPM 的 Linux 系统，可以到 <http://www.rpmfind.net> 去搜索一个合适的 Ruby RPM 包。输入 Ruby 作为搜索关键字，并从列出的版本号、体系结构和发行版中选择。例如，`ruby-1.8.2.i386` 是 Ruby 1.8.2 在 Intel x86 体系结构上的一个二进制发行版。

对 Debian 这样基于 dpkg 的 Linux 系统，你可以使用 apt-get 系统去查找和安装 Ruby。也可以使用 apt-cache 命令去搜索 Ruby 包。

```
# apt-cache search ruby interpreter
libapache-mod-ruby - Embedding Ruby in the Apache web server
libber-ruby 1.6 - Tiny eRuby for Ruby 1.6
libber-ruby 1.8 - Tiny eRuby
ruby - An interpreter of object-oriented scripting language Ruby
ruby1.7 - Interpreter of object-oriented scripting language Ruby
ruby1.8 - Interpreter of object-oriented scripting language Ruby
```

你可以使用 apt-get 来安装这些包中的任意一个。

```
# apt-get install ruby1.8
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  libruby1.8
Suggested packages:
  ruby1.8-examples
The following NEW packages will be installed:
  libruby1.8 ruby1.8
  :  :
```

注意在 Unix 和 Linux 系统上安装全局软件包需要超级用户权限，这也是为什么我们用#作为提示符的原因。

如果你运行的是 Microsoft Windows，在 <http://rubyinstaller.rubyforge.org> 上可以找到 Ruby One-Click 安装程序的主页。

1.1.2 从源码编译 Ruby

Building Ruby from Source

因为 Ruby 是一个开放源代码的项目，你可以下载 Ruby 解释器的源代码，并在自己的系统上编译。与使用二进制发行版相比，这种方式给予了对安装选项的更多控制，并且能保持安装是最新的。不便之处在于需要管理编译和安装过程。虽然这个过程并不繁琐，但是如果你从来没有从源代码安装过开源软件，也是有点令人发怵的。

第一件要做的事是下载源代码。从 <http://www.ruby-lang.org> 上下载源代码有 3 种选择：

1. *Tarball* 格式的稳定版（stable release）。Tarball 是一个归档文件，很像 zip 文件。点击 Download Ruby 链接，然后点击 stable release 链接即可获得。
2. 稳定版快照（stable snapshot）。这是每天晚上创建的基于 Ruby 稳定版开发分支的最新源代码的 tarball 文件。稳定分支（stable branch）是用于产品发布的代码，一般来说比较可靠。但是由于 snapshot 是每天产生一个，所以新的特性可能没有经过全面的测试——通常上述第 1 点中的稳定 tarball 更可靠。

3. **Nightly 开发版快照** (*nightly development snapshot*)。这也是每天晚上创建的 tarball。和第 1 点、第 2 点中的稳定代码不同的是，这些代码是最新主流代码，因为这是从主开发分支中获得的。这种版本有可能会出现无法预料的问题。

如果你打算经常下载上面提到的两种快照之一，那么直接订阅源码库（source repository）更方便。下一页提供了关于它的更详细的信息。

下载了 tarball 归档文件后，还需要把它解压缩还原为原来的格式。可以用 `tar` 命令做这项工作（如果你的系统上没有安装 `tar` 命令，你可以尝试其他的归档工具，现在有很多工具支持 `tar` 格式的文件）。

```
% tar xzf snapshot.tar.gz
ruby/
ruby/bcc32/
ruby/bcc32/Makefile.sub
ruby/bcc32/README.bcc32
:
```

这会安装 Ruby 源代码树（source tree）到 `ruby/` 子目录中。在那个目录中可以找到一个名为 `README` 的文件，它详细解释了安装过程。总的说来，在基于 POSIX 的系统上编译 Ruby，使用的 4 个命令与编译大多数开放源代码程序所使用的命令相同：`./configure`、`make`、`make test` 和 `make install`。也可以在别的环境上（包括 Windows）编译 Ruby，这需要使用一个像 cygwin¹这样的 POSIX 模拟环境，或者使用自带的编译器——参照发行版的 `win32` 子目录中的 `README.win32`。

1.1.3 本书的源代码

Source Code from This Book

我们已经将本书所含的源码放到我们的网站 <http://pragmaticprogrammer.com/titles/ruby/code> 上以供下载。有的地方列出的代码是完整的，有的地方列出的代码只是完整源代码文件的一部分——它可能还包括额外的骨架代码以使其可以编译。

1.2 运行 Ruby

Running Ruby

现在安装了 Ruby，你可能想运行一些程序。和编译式语言不同，你有两种方式运行 Ruby：（1）以交互的方式输入代码直接执行；或者（2）先创建程序文件，然后再运行。以交互方式运行代码是体验 Ruby 语言的一种绝佳方式，但对于较复杂的代码，或者想多次运行代码，则需要创建程序文件，然后运行它们。

¹ 详见 <http://www.cygwin.com>.

最新的 Ruby

The Very Latest Ruby

对那些想获得刚出炉、未经测试的最新代码的人（比如撰写这本书的笔者们）来说，可以直接从开发者的工作仓库中获得 Ruby 开发版。

Ruby 开发者使用 CVS (Concurrent Version System，可以从 <https://www.cvshome.org> 自由获得) 作为版本控制系统。你可以执行下面的命令，以匿名用户将文件从其归档 (archive) 中检出：

```
% cvs -z4 -d :pserver:anonymous@cvs.ruby-lang.org:/src login
(Logging in to anonymous@cvs.ruby-lang.org)
CVS password: ENTER
% cvs -z4 -d :pserver:anonymous@cvs.ruby-lang.org:/src checkout ruby
```

完整的源代码树（包含开发者所作的最新改动）将会被拷贝到你的本地机器上的 ruby 子目录。

这个命令将会导出源代码开发树的主分支。如果你想导出 Ruby 1.8 分支，在第二个命令的 checkout 后面加入 -r ruby_1_8。

如果你想用 CVSup 镜像工具（可以从 <http://www.cvsup.org> 自由获得），你可以从 ruby-lang 的网站 <http://cvs.ruby-lang.org/cvsup> 上找到 Ruby 的 supfiles。

1.2.1 交互式 Ruby

Interactive Ruby

要以交互方式运行 Ruby，只须在 Shell 提示符下输入 **ruby**。下面我们输入了一个 puts 表达式和文件结束符（在我们的系统中是 Ctrl+D）。这种方式确实可以工作，但是如果输入发生错误将会很麻烦，而且也不能真正看到按键的结果。

```
% ruby
puts "Hello, world!"
^D
Hello, world!
```

对大多数人来说，irb (Interactive Ruby) 是以交互方式执行 Ruby 的首选。irb 是一个 Ruby Shell，包括命令行历史 (command-line history) 功能、行编辑 (line-editing) 功能和作业控制 (job control)。实际上，从第 185 页开始有专门的章节讨论 irb。可以从命令行运行 irb，一旦启动，就可以键入 Ruby 代码。irb 将会对每个表达式求值并将结果显示出来。

```
% irb
irb(main):001:0> def sum(n1, n2)
irb(main):002:1>   n1 + n2
irb(main):003:1> end
=> nil
irb(main):004:0> sum(3, 4)
=> 7
irb(main):005:0> sum("cat", "dog")
=> "catdog"
```

我们建议你熟练使用 `irb`, 以便能以交互方式运行我们提到的程序。

有个小技巧可以让你使用 `irb` 来运行已经存在文件中的代码示例。比方说你想试试第 208 页的 Fibonacci 模块。为此, 你需要在 `irb` 中先装载程序文件, 然后调用其中包含的方法。在本例中, 程序文件是 `code/rdoc/fib_example.rb`。

```
% irb
irb(main):001:0> load "code/rdoc/fib_example.rb"
=> true
irb(main):002:0> Fibonacci.upto(20)
=> [1, 1, 2, 3, 5, 8, 13]
```

1.2.2 Ruby 程序 Ruby Programs

你可以像运行任意 shell 脚本、Perl 或 Python 程序那样从文件中运行 Ruby 程序, 只需运行 Ruby 解释器, 并以脚本文件名作为参数即可。

```
% ruby myprog.rb
```

你也可以用 Unix 的“shebang”符号作为程序文件的第一行。²

```
#!/usr/local/bin/ruby -w
puts "Hello, world!"
```

如果将此源代码文件设置为可执行 (比如使用 `chmod +x myprog.rb`) , Unix 就可将其作为程序来运行。

```
% ./myprog.rb
Hello, world!
```

在微软 Windows 操作系统上使用文件关联可以做类似的事情, 而且可以通过在浏览器中双击文件名来运行 Ruby 图形用户界面程序。

² 如果系统支持的话, 你可以使用`#!/usr/bin/env ruby` (系统将会搜索 Ruby 路径并执行之), 避免在“shebang”一行上硬性指定 Ruby 路径。

1.3 Ruby 文档: RDoc 和 ri

Ruby Documentation: RDoc and ri

Ruby 库数量的增加,使得不可能在一本书中对所有库进行说明;Ruby 自带的标准库含有的方法 (methods) 已超过 9 000 个。幸运的是,还存在一种纸制书籍以外的方式来阐述这些方法(以及类和模块)。许多库由内部一个称为 RDoc 的系统来进行文档化。

如果使用 RDoc 来文档化一个源文件,那么其中的文档可以被提取出来,并转换成 HTML 或者 ri 格式。

有多个网站含有 Ruby 的完整的 RDoc 文档,其中 <http://www.ruby-doc.org> 可能是最知名的。大体浏览一下,你至少能发现一些 Ruby 库的文档形式。网站还在不断地加入新文档。

ri 是一种本地命令行工具,用来阅读 RDoc 格式的 Ruby 文档。大多数的 Ruby 发行版也会安装 ri 程序所要使用的资源。

输入 ri 类名,就可以得到该类的文档。例如下面列出了 GC 类的概要信息(输入 ri-c 可以找到 ri 文档中的所有类的列表)。

```
% ri GC
-----Class: GC
The GC module provides an interface to Ruby's mark and sweep
garbage collection mechanism. Some of the underlying methods are
also available via the ObjectSpace module.

-----
Class methods:
  disable, enable, start

Instance methods:
  garbage_collect
```

以方法名字作为参数,可以获得该方法的信息。

```
% ri enable
-----GC::enable
GC.enable => true or false
-----
Enables garbage collection, returning true if garbage collection
was previously disabled.

  GC.disable #=> false
  GC.enable  #=> true
  GC.enable  #=> false
```

如果传给 `ri` 的方法名称存在于多个类或模块当中, `ri` 会列出所有的可能。在方法名字前加上类名字和一个点号, 重新调用该命令。

```
% ri start
More than one method matched your request. You can refine
your search by asking for information on one of:

    Date#new_start, Date#start, GC::start, Logger::Application#start,
    Thread::start

% ri GC.start
-----GC::start-----
GC.start                => nil
gc.garbage_collect      => nil
ObjectSpace.garbage_collect => nil
-----
Initiates garbage collection, unless manually disabled.
```

使用“`ri --help`”可以获得使用 `ri` 的帮助信息。特别是你可能想体验一下“`--format`”选项, 该选项告诉 `ri` 如何显示修饰性的文本符号 (例如节头)。如果你的终端程序支持 ANSI 转义序列, 使用“`--format ansi`”就可显示好看且多彩的画面。一旦找到了你喜欢的一组选项, 就可以把它们设置到 `RI` 环境变量中。在笔者使用的 shell(zsh), 可以这样做:

```
% export RI="--format ansi --width 70"
```

如果一个类或者模块还没有 RDoc 格式的文档, 可以发请求到 `suggestions@ruby-doc.org` 让人添加它。

如果你不经常使用 shell, 那么前面的命令行操作可能有些令人不愉快。然而实际上这并不难, 而且你会发现这种串联各种命令在一起处理问题的能力常常令人惊讶。坚持使用它, 就会走上掌握 Ruby 和你的计算机的正途。



Ruby.new

Ruby.new

最初写本书时，我们有一个雄心勃勃的计划（那时我们还年轻）。我们想自顶而下地介绍 Ruby 这门语言。从类和对象开始，最后以枯燥乏味的语法细节结束。那时候这主意看起来不错。毕竟，Ruby 里几乎所有东西都是对象，因此从对象开始讲起是不无道理的。

我们曾经这么想过。

不幸的是，我们发现用这种方式去描述一门语言是一件很困难的事情。如果不介绍字符串（strings）、`if` 语句、赋值以及其他语法细节，那么写一个类（class）的示例都很困难。在整个由上而下的描述中，我们不得不一次次地对底层细节进行描述，以使示例代码变得有意义。

因此，我们开始了另外一个雄心勃勃的计划（我们不是平白无故被称为 *pragmatic programmers* 的）。本书仍旧从顶层开始介绍 Ruby，但是在开始之前，我们添加了篇幅不长的一章，用 Ruby 特有的词汇描述了例子中要用到的所有常见语言特性，它就像一本迷你教程那样，把各位引入到这本书的其他部分。

2.1 Ruby 是一门面向对象语言

Ruby Is an Object-Oriented Language

重新说一遍，Ruby 是真正的面向对象语言。你所操作的每件东西都是对象，操作结果本身也是对象。当然，很多语言同样声称它们是真正的面向对象语言，而其使用者常会对“面向对象究竟是什么”有不同的解释，也用不同的术语来说明他们所使用的概念。

在深入细节之前，我们先简单地看看本书所要用到的术语和程序语言的表示法。

编写面向对象的代码时，通常你得根据真实世界对概念进行建模。在这个过程中，要挖掘出需要在代码中所表达的事物种类。在点唱机系统（jukebox）中，“歌曲”的概念可能就是这么一个种类。在 Ruby 里，需要定义类（*class*）来表示实体。类是状态（*state*，比如歌曲名称）和使用这些状态的方法（*method*，可能是一个播放歌曲的方法）的组合。

一旦建立了这些类，通常要为每个类创建若干个实例（*instances*）。在点唱机系统中可以包含一个称为 Song 的类，你可能会为诸如“Ruby Tuesday”、“Enveloped in Python”、“String of Pearls”、“Small Talk”之类的歌曲分别创建单独的实例。“对象（*object*）”这个概念和“类的实体（*class instance*）”等同互用（我们比较懒，因为 object 长度更短些，所以可能会更常使用“对象”这个词）。

在 Ruby 中，通过调用构造函数（*constructor*）来创建对象，这是一种与类相关联的特殊方法。标准的构造函数被称为 new。

```
song1 = Song.new("Ruby Tuesday")
song2 = Song.new("Enveloped in Python")
# and so on
```

这些实例都是从相同的类派生出来的，不过它们具有各异的特征。首先，每个对象有一个唯一的对象标识符（*object identifier*，缩写为 *object ID*）。其次，可以定义一些实例变量（*instance variables*），这些变量的值对于每个实例来说是唯一的。这些实例变量都持有对象的状态（*state*）。例如，每首歌曲可能都会用一个实例变量来保存歌曲的标题。

可以为每个类定义实例方法（*instance methods*）。每个方法是一组功能，它们可能会在类的内部或从类的外部（依赖于访问约束）被调用。这些实例方法反过来访问对象的实例变量及其状态。

方法是通过向对象发送消息（*message*）来唤起调用的。消息包含方法名称以及方法可能需要的参数。¹当对象接收到一条消息时，它在自己的类中查找相应的方法。如果找到了，该方法会被执行。如果没有找到……唔，我们后面会讲到这一点。

方法和消息之间的关系可能听起来挺复杂，但实际上这是非常自然的。让我们来看一些方法调用。

```
"gin joint".length → 9
"Rick".index("c") → 2
-1942.abs → 1942
sam.play(song) → "duh dum, da dum de dum ..."
```

¹ 将方法调用表述为消息形式的想法来自于 Smalltalk。

(记住，在本书的代码范例中，箭头指向的是表达式的值。执行 `-1942.abs` 的结果是 `1942`。如果将这段代码放到一个文件中，用 Ruby 运行它，你不会看到任何输出，因为该语句本身并没有告诉 Ruby 去显示求值结果。如果使用 `irb`，你就会看到在本书中显示的这些结果值。)

在这里点号（“.”）之前的东西被称为接收者（*receiver*），点号后面的名字是被调用的方法。第一个范例用来取得字符串的长度，第二个范例要求另外一个字符串找到字母 `c` 的索引号，第三行代码计算一个数值的绝对值。最后一行代码中，我们要求 `Sam` 演奏一首歌曲。

值得注意的是：Ruby 和大多数别的语言之间有一个很大的区别。例如在 Java 中，是通过调用另外一个函数来得到某些成员数据的绝对值，并把结果传回数据本身。你可以写成

```
number = Math.abs(number) // Java 代码
```

而在 Ruby 里，确定绝对值的能力内建在数字中——处理细节在内部实现中。只要发送 `abs` 消息到一个数字对象，让它去得到绝对值即可。

```
number = number.abs
```

同样的情况应用于所有的 Ruby 对象：用 C 语言你得写 `strlen(name)`，但是用 Ruby 它是 `name.length`，诸如此类。这就是我们所说的——Ruby 是一门真正的面向对象语言。

2.2 Ruby 的一些基本知识

Some Basic Ruby

开始学习一种新语言时没有多少人喜欢阅读成堆的、令人心烦的语法规则，所以我们暂时绕开一下。本节会讲到一些重要的规则，这些规则是编写 Ruby 程序所必须掌握的。在后面的章节中（第 22 章，从第 317 页开始），我们再介绍全部细节。

让我们从一个简单的 Ruby 程序开始。编写一个方法，让它返回一个亲切的、个性化的问候。然后我们会调用这个方法若干次。

```
def say_goodnight(name)
  result = "Good night, " + name
  return result
end
# time for bed...
puts say_goodnight("John-Boy")
puts say_goodnight("Mary-Ellen")
```

就像这个范例所显示的那样，Ruby 语法干净。只要每个语句放在单独的行上，就不需要在语句结束处加上分号。Ruby 注释以一个`#`字符开始，在行尾结束。代码布局任由

你决定：缩进编排并不重要（但是如果打算要发布你的代码的话，使用两个字符的缩进编排会让你在社区内交很多朋友）。

方法（method）用关键字 `def` 来定义，后面跟着方法名称（在本例中，它是 `say_goodnight`）和在括号中的方法参数（实际上，括号是可选的，但我们喜欢使用它们），Ruby 没有使用花括号来界定复杂的语句和定义的程序体。相反，可以用关键字 `end` 结束这个程序体。这个方法的程序体相当地简单。第一行连接"Good night, "字符串和参数 `name`，并把这个结果赋值给局部变量 `result`。下一行将结果返回给调用者。注意，我们不必声明 `result` 这个变量：当赋值给它时，它便存在了。

这个方法在定义之后被调用了两次，两次都是把结果传递给 `puts` 方法。`puts` 的功能就是输出其参数，后面跟一个回车换行符（移动到下一行）。

```
Good night, John-Boy
Good night, Mary-Ellen
```

`puts say_goodnight ("John-Boy")` 这行代码包含两个方法调用，一个是 `say_goodnight`，另外一个是 `puts`。为什么一个调用把参数放在括号里，而另外一个没有呢？这纯粹是个人的喜好。下面两行是等同的。

```
puts say_goodnight("John-Boy")
puts(say_goodnight("John-Boy"))
```

当然，生活不是总这么简单的，优先级规则使得我们难以判断哪个参数属于哪个方法调用。所以建议除了最简单的情况以外，都请使用括号。

这个范例也展示了一些 Ruby 字符串对象。创建字符串对象有多种途径，最常用的可能是使用字符串字面量（literals），即一组单引号或双引号之间的字符序列。这两种形式的区别在于，当构造字面量时，Ruby 对字符串所做处理的多寡有所不同。Ruby 对单引号串处理得很少。除了极少的一些例外，键入到字符串字面量的内容就构成了这个字符串的值。

Ruby 对双引号字符串有更多的处理。首先，它寻找以反斜线开始的序列，并用二进制值替换它们。其中最常见的是`\n`，它会被回车换行符替换掉。当一个包含回车换行符的字符串输出时，`\n` 会强制换行。

```
puts "And good night,\nGranma"
```

输出结果：

```
And good night,
Granma
```

Ruby 对双引号字符串所做的第二件事情是字符串内的表达式内插（expression interpolation），`#{表达式}`序列会被“表达式”的值替换。可以用这种方式重写前面的方法。

```
def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Pa')
```

输出结果：

```
Good night, Pa
```

Ruby 构建这个字符串对象时，它找到 `name` 的当前值并把它替换到字符串中。任意复杂的表达式都允许放在`#{...}`结构中。这里调用在所有字符串中定义的 `capitalize` 方法，将参数的首字母改为大写之后输出。

```
def say_goodnight(name)
  result = "Good night, #{name.capitalize}"
  return result
end
puts say_goodnight('uncle')
```

输出结果：

```
Good night, Uncle
```

为了方便起见，如果表达式只是一个全局实例或类变量（马上会讲到），则不需要提供花括号。

```
$greeting = "Hello"      # $greeting 是全局变量
@name     = "Prudence"    # @name 是实例变量
puts "#$greeting, #$name"
```

输出结果：

```
Hello, Prudence
```

有关字符串更多的信息和其他 Ruby 标准类型，请阅读从第 59 页开始的第 5 章。

最后，可以进一步地简化这个方法。Ruby 方法所返回的值，是最后一个被求值的表达式的值，所以可以把这个临时变量和 `return` 语句都去掉。

```
def say_goodnight(name)
  "Good night, #{name}"
end
puts say_goodnight('Ma')
```

输出结果：

```
Good night, Ma
```

我们保证本节要简略，最后再讲一个内容：Ruby 的名称。为简短起见，我们会使用一些没有在这里定义的术语（比如类变量，*class variable*）。当然，我们现在就会开始使用这些术语，这样在后面谈到类变量等东西的时候，你就不会感到陌生了。

Ruby 使用一种命名惯例来区分名称的用途：名称的第一个字符显示这个名称如何被使用。局部变量、方法参数和方法名称都必须以小写字母或下画线开始。全局变量都有美元符号 (\$) 为前缀，而实例变量以“at” (@) 符号开始。类变量以两个“at” @@ 符号开始。最后，类名称、模块名称和常量都必须以一个大写字母开始。各种名称的示例在图 2.1 中给出。

从上述规定的初始字符之后开始，名称可以是字母、数字和下画线的任意组合（但跟在@符号之后的符号不能是数字）。但是按惯例，包含多个单词的实例变量名称在词与词之间使用下画线连接，包含多个单词的类变量名称使用混合大小写（每个单词首字母大写）。方法名称可以？！和=字符结束。

2.3 数组和散列表

Arrays and Hashes

Ruby 的数组（arrays）和散列表（hashes）是被索引的收集（indexed collections）²。两者都存储对象的集合，通过键（key）来访问。数组的键是整数，而散列表支持以任何对象作为它的键。数组和散列表会按需调整大小来保存新的元素。访问数组元素是高效的，但是散列表提供了灵活性。任何具体的数组或散列表可以保存不同类型的对象；马上就会看到，一个数组可以包含一个整数、一个字符串和一个浮点数。

使用数组字面量（array literal）——即方括号之间放一组元素——可以创建和初始化新的数组对象。有了数组对象，在方括号之间提供索引便可以访问单个元素，如下例所示。注意 Ruby 数组的索引从零开始。

```
a = [ 1, 'cat', 3.14 ]      # 有三个元素的数组
# 访问第一个元素
a[0] → 1
# 设置第三个元素
a[2] = nil
# 显示这个数组
a → [1, "cat", nil]
```

你可能已注意到在这个例子中使用了 `nil` 这个特别的值。许多语言中 `nil`（或 `null`）的概念是指“没有对象”。在 Ruby 中，这是不一样的：`nil` 是一个对象，与别的对象一样，只不过它是用来表示没有任何东西的对象。让我们继续介绍数组和散列表。

² 译注：“collection”作为程序语言的术语，目前国内尚无统一译法，常见的译法有“群集”、“集合”等。本书中，表达程序语言特殊概念的“collection”译为“收集”，“set”译为“集合”，表达普通概念的“collection”在不与“set（集合）”概念相混淆的情况下译为“集合”，敬请读者注意辨析两者的区别。

表 2.1 变量和类名称样例

局部变量	全局变量	实例变量	类变量	常量和类名称
name	\$debug	@name	@@total	PI
fish_and_chips	\$CUSTOMER	@point_1	@@syntab	FeetPerMile
x_axis	\$_	@x	@@N	String
thx1138	\$plan9	@_	@@x_pos	MyClass
_26	\$Global	@plan9	@@SINGLE	JazzSong

有时候创建一组单词的数组是一件痛苦的事情——要处理许多引号和逗号。幸运的是，Ruby 有一种快捷方式：`%w` 能够完成我们想做的事情。

```
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
a[0] → "ant"
a[3] → "dog"
# this is the same:
a = %w{ant bee cat dog elk}
a[0] → "ant"
a[3] → "dog"
```

Ruby 的散列表与数组相似。散列表字面量（hash literal）使用花括号而不是方括号。这个字面量必须为每一项提供两个对象：一个键（key）和一个值（value）。

例如，你可能想将乐器映射到它们所属的交响乐章节，可以用散列表这么来做：

```
inst_section = {
  'cello'      => 'string',
  'clarinet'   => 'woodwind',
  'drum'        => 'percussion',
  'oboe'        => 'woodwind',
  'trumpet'    => 'brass',
  'violin'     => 'string'
}
```

=>的左边是键（key），右边是其对应的值（value）。在一个散列表里面，键必须是唯一的（不能有两个“drum”项）。散列表里面的键和值可以是任意对象——你可能会有这样的散列表，它的值是数组或别的散列表等。

散列表使用与数组相同的方括号表示法来进行索引。

```
inst_section['oboe'] → "woodwind"
inst_section['cello'] → "string"
inst_section['bassoon'] → nil
```

正如上例所示，默认情况下，如果用一个散列表没有包含的键进行索引，散列表就返回 `nil`。通常这样是很方便的，比如在条件表达式中 `nil` 就意味着 `false`。而有时候你可能想改变这个默认动作。比如使用散列表来计算每个键出现的次数时，如果这个默认值是 `0` 的话就会很方便。这很容易做到：当创建一个新的空散列表时，可以指定一个默认值。

```
histogram = Hash.new(0)
histogram['key1'] → 0
histogram['key1'] = histogram['key1'] + 1
histogram['key1'] → 1
```

数组和散列表对象包含大量有用的方法：详见第 43 页开始的讨论，以及第 427 页和第 492 页开始的参考手册。

2.4 控制结构

Control Structures

Ruby 具有所有常见的控制结构，如 `if` 语句和 `while` 循环。Java、C 和 Perl 程序员可能会对这些语句的程序体“缺乏花括号”不太适应。Ruby 是使用 `end` 关键字来表明程序体的结束。

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
  puts "Enter a number"
end
```

同样地，`while` 语句以 `end` 结束。

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

大多数 Ruby 语句会返回值，这意味着可以把它们当条件使用。例如，`gets` 方法从标准输入流返回下一行，或者当到达文件结束时返回 `nil`。因为 Ruby 在条件中把 `nil` 当作一个假值 (`false`) 对待，可以像下面这样来处理文件中的行。

```
while line = gets
  puts line.downcase
end
```

在这里赋值语句要么把变量 `line` 设置为下一行，要么是 `nil`，然后 `while` 语句测试这个赋值的结果，如果它是 `nil`，则终止这个循环。

如果 `if` 或 `while` 语句的程序体只是一个表达式，Ruby 的语句修饰符 (*statement modifiers*) 是一种有用的快捷方式。只要写出表达式，后面跟着 `if` 或 `while` 和条件。比如，这是 `if` 语句的例子。

```
if radiation > 3000
  puts "Danger, Will Robinson"
end
```

用语句修饰符重新编写了同样这个例子。

```
puts "Danger, Will Robinson" if radiation > 3000
```

同样地，下面是 `while` 循环语句

```
square = 2
while square < 1000
  square = square*square
end
```

用语句修饰符，这个语句变得更简洁了。

```
square = 2
square = square*square while square < 1000 .
```

Perl 程序员应该对这些语句修饰符很熟悉。

2.5 正则表达式 Regular Expressions

程序员对很多 Ruby 内建的类型是熟悉的，大多数语言都有字符串、整数、浮点和数组等等类型。但是正则表达式的内建支持通常只限于脚本语言如 Ruby, Perl 和 awk 等。这是一个耻辱：尽管正则表达式很神秘，但它是一个强大的文本处理工具。通过内建而不是通过程序库接口来支持它，有很大的不同。

有很多书通篇都是介绍正则表达式的（例如《精通正则表达式》*Mastering Regular Expressions* [Fri02]³），因此我们不试图在这有限的篇幅里涵盖所有方面。相反，我们只会看看一些实用的正则表达式的例子。从第 68 页开始我们会对正则表达式有一个全面的介绍。

正则表达式只是一种指定字符模式的方法，这个字符模式会在字符串中进行匹配。在 Ruby 中，通常在斜线之间 (`/pattern/`) 编写模式 (`pattern`) 来创建正则表达式。同时，Ruby 就是 Ruby，正则表达式是对象并且可以当作对象来操作。

比如，可以使用如下的正则表达式来编写模式，它会匹配包含 Perl 或 Python 的字符串。

```
/Perl|Python/
```

³ *Mastering Regular Expressions*, Third Edition 已经于 2006 年 8 月由 O'Reilly Media, Inc. 出版，中文版由电子工业出版社于 2007 年出版。—编者

前面那个斜线界定了这个模式，模式由要匹配的两个子字符串组成，它们被管道符 (`|`) 分开。管道符意味着“要么右边的字符串，要么左边的字符串”。在这个例子中，它们分别是 *Perl* 或 *Python*。就像在算术表达式中的那样，可以在模式中使用括号，因此可以把模式写成

```
/P(erl|y)thon/
```

也可以在模式中指定**重复** (*repetition*)。`/ab+c/` 匹配一个含有 *a*，后面跟着一个或多个 *b*，其后是 *c* 的字符串。把模式中的加号改成星号，`/ab*c/` 创建了一个匹配含有 *a*，零个或多个 *b* 和然后是 *c* 的正则表达式。

也可以在模式中匹配一组或多组字符。一些常见的例子是**字符类** (*character classes*) 如`\s`，它匹配空白字符（空格符、制表符、回车换行符等等）；`\d` 匹配任何数字；还有`\w`，它匹配会出现在一个词内的任何字符。一个点`(.)` 匹配几乎任意字符。第 72 页有一个这些字符类的表。

可以把它们用一起来产生一些有用的正则表达式。

<code>/\d\d:\d\d:\d\d/</code>	# 时间如 12:34:56
<code>/Perl.*Python/</code>	# Perl，零个或多个别的字符，然后是 Python
<code>/Perl Python/</code>	# Perl，一个空格和 Python
<code>/Perl *Python/</code>	# Perl，零个或多个空格，和 Python
<code>/Perl +Python/</code>	# Perl，一个或多个空格，和 Python
<code>/Perl\s+Python/</code>	# Perl，空格字符，然后是 Python
<code>/Ruby (Perl Python)/</code>	# Ruby，一个空格，然后是 Perl 或者 Python

一旦创建了模式，不去用它总不好意思。`=~` 匹配操作符可以用正则表达式来匹配字符串。如果在字符串中发现了模式，`=~` 返回模式的开始位置，否则它返回 `nil`。这意味着可以在 `if` 和 `while` 语句中把正则表达式当作条件使用。比如，如果字符串包含了 *Perl* 或 *Python*，下面的代码输出一条消息。

```
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end
```

正则表达式匹配到的字符串部分，可以使用 Ruby 的其中一种替换方法，替换为其他文本。

```
line.sub(/Perl/, 'Ruby')          # 用 'Ruby' 替换第一个 'Perl'
line.gsub(/Python/, 'Ruby')       # 用 'Ruby' 替换所有的 'Python'
```

使用下面语句可以用 *Ruby* 替换出现 *Perl* 和 *Python* 的每个地方。

```
line.gsub(/Perl|Python/, 'Ruby')
```

在本书中，还会讲到正则表达式的许多方面。

2.6 Block 和迭代器

Blocks and Iterators

本节会简单地描述 Ruby 的一个独特特性。Block，一种可以和方法调用相关联的代码块，几乎就像参数一样。这是一个不可思议的功能强大的特性。一位评论家曾经说：“这个特性是相当的有趣和重要，如果以前没有注意到，从现在开始你应当要注意了。”必须同意他的观点是正确的。

可以用 `block` 实现回调（但它比 Java 的匿名内部（anonymous inner）类更简单），传递一组代码（但它远比 C 的函数指针灵活），以及实现迭代器。

Block 只是在花括号或者 `do...end` 之间的一组代码。

```
{ puts "Hello" }          # this is a block
do                      #####
  club.enroll(person)    # and so is this
  person.socialize       #
end                      #####
```

为什么有两种分界符？部分原因是有人觉得有时候用一种分界符比另外一种感觉更自然。另外一部分原因是它们有不同的优先级：花括号比 `do/end` 绑定的更紧密些。在本书中，我们尝试遵循正在成为 Ruby 标准的一个约定俗成，单行 `block` 用花括号，多行 `block` 用 `do/end`。

一旦创建了 `block`，就可以与方法的调用相关联。把 `block` 的开始放在含有方法调用的源码行的结尾处，就可以实现关联。比如，在下面的代码中，含有 `puts "Hi"` 的 `block` 与 `greet` 方法的调用相关联。

```
greet { puts "Hi" }
```

如果方法有参数，它们出现在 `block` 之前。

```
verbose_greet("Dave", "loyal customer") { puts "Hi" }
```

然后使用 Ruby 的 `yield` 语句，方法可以一次或多次地调用（`invoke`）相关联的 `block`。可以把 `yield` 想象成比如方法调用，它调用含有 `yield` 语句的方法所关联的 `block`。

下面的例子显示了如何使用 `yield` 语句。定义了一个方法，它会调用 `yield` 两次。然后调用这个方法，把 `block` 放在同一行，在方法调用之后（并在方法的所有参数之后）。⁴

⁴ 有些人喜欢把 `block` 和方法的关联看作是一种参数传递。一方面讲这是正确的，但这并不是全部。最好把 `block` 和方法想象成协同例程（coroutines），在它们之间来回地转换控制权。

```

def call_block
  puts "Start of method"
  yield
  yield
  puts "End of method"
end

call_block { puts "In the block" }

```

输出结果：

```

Start of method
In the block
In the block
End of method

```

看看（`puts "In the block"`）block 中的代码如何被执行两次，每次调用 `yield` 时，代码都会被执行。

可以提供参数给对 `yield` 的调用：参数会传递到 block 中。在 block 中，竖线（|）之间给出参数名来接受这些来自 `yield` 的参数。

```

def call_block
  yield("hello", 99)
end

call_block {|str, num| ... }

```

在 Ruby 库中大量使用了 block 来实现迭代器：迭代器是从某种收集（collection）如数组中连续返回元素的方法。

```

animals = %w( ant bee cat dog elk ) # 创建一个数组
animals.each {|animal| puts animal } # 迭代它的内容

```

输出结果：

```

ant
bee
cat
dog
elk

```

让我们看一下如何实现应用在前面例子中的 `Array` 类中的 `each` 迭代器。`each` 迭代器循环处理数组中的元素，对每个元素调用 `yield`。在伪码中，它可能写成：

```

# 在 Array 类中.....
def each
  for each element # <-- 无效的 Ruby 语句
    yield(element)
  end
end

```

许多内建于 C 和 Java 等语言的循环结构在 Ruby 中只是方法调用，这些方法会零次或多次地调用相关联的 block。

```
[ 'cat', 'dog', 'horse' ].each { |name| print name, " " }
5.times { print "*" }
3.upto(6) { |i| print i }
('a'...'e').each { |char| print char }
```

输出结果：

```
cat dog horse *****3456abcde
```

上面的代码要求对象 5 五次调用 block；然后要求对象 3 调用一个 block，并传入一个连续的值，直到这个值到达 6 为止。最后对 *a* 到 *e* 的字符区间（range），使用 each 方法调用 block。

2.7 读/写文件

Reading and Writing

Ruby 有一个完备的 I/O 库。但是本书的大多数例子只使用其中一些简单的方法。已经碰到了两个用来输出的方法。`.puts` 输出它的参数，并在每个参数后面添加回车换行符。`.print` 也输出它的参数，但没有添加回车换行符。它们都可以用来向任何 I/O 对象进行输出，但在默认情况下，它们输出到标准输出。

另外一个常用的输出方法是 `printf`，它在一个格式化字符串的控制下打印出它的参数（就象 C 或 Perl 中的 `printf`）。

```
printf("Number: %5.2f,\nString: %s\n", 1.23, "hello")
```

输出结果：

```
Number: 1.23,
String: hello
```

在这个例子中，"Number: %5.2f,\nString: %s\n" 格式化字符串，告诉 `printf` 替换一个浮点数（最多允许 5 个字符，并且 2 个在小数点后面）和一个字符串。注意到回车换行符（\n）嵌入到格式化串中；回车换行符把输出移动到下一行。

有许多方式可以把输入读到程序中。最传统的方式是使用 `gets` 函数，它从程序的标准输入流中读取下一行。

```
line = gets
print line
```

Ruby 揭开了新篇章

Ruby Escapes Its Past

早些时期 Ruby 从 Perl 语言借鉴了许多。其中的一个特性是与全局变量打交道时的某种“神奇的魔术”，可能没有别的全局变量比 `$_` 更神奇的了。例如，`gets` 方法有副作用：它返回刚读取的这行的同时，也把这行保存到 `$_` 变量中。如果不带任何参数地调用 `print`，它会打印出 `$_` 中的内容。如果用正则表达式作为条件编写 `if` 或 `while` 语句，正则表达式会与 `$_` 匹配。因为所有这些神奇之处，导致你可能写出下面的程序，在文件中寻找含有“Ruby”的所有行。

```
while gets
  if /Ruby/
    print
  end
end
```

但是在完美主义者（purists）的努力下，这种 Ruby 编程风格正在迅速地衰落。Matz 恰好就是这样一个完美主义者，Ruby 就会为这些特殊的使用发出警告：将来这些特性会消失。

但是这并不意味着编程时要敲入更多字符。“Ruby 风格”会使用迭代器和预定义的对象 `ARGF`，`ARGF` 表示程序的输入文件。

```
ARGF.each { |line| print line if line =~ /Ruby/ }
```

甚至可以写得更简洁些：

```
print ARGF.grep(/Ruby/)
```

总的来说，Ruby 社区正在逐渐脱离某些 Perl 主义的风格。如果用 `-w` 选项运行程序来显示警告信息（你确实启用警告信息来运行程序，不是吗？），你会发现 Ruby 解释器捕捉到了大部分的此类用法。

2.8 更高更远

Onward and Upward

好了，我们已经介绍完了 Ruby 的一些基本特性。已经知道了对象、方法、字符串、容器和正则表达式，看到了一些简单的控制结构和一些小巧的迭代器。本章的内容构成了学习本书其他部分的基本知识。

随着时间的推移，该是前进到更高层次的时候了。下面我们会介绍类和对象。这两个概念是 Ruby 最高级别的构成部分，同时也是整个语言的支柱。

类、对象和变量

Classes, Objects, and Variables

从我们目前已经演示的示例来看，你可能会对之前我们关于“Ruby 是一门面向对象语言”的论断有所疑虑。好吧，本章就是为了证明这一论断。我们将会看到，在 Ruby 中如何创建类和对象，以及和其他大多数面向对象语言相比更强大的一些设施。其间，我们还将实现下一个价值十亿美元的产品——基于 Internet 的爵士和布鲁斯自动点唱机——的一部分。

在忙碌数月之后，我们重金聘请的研发人员作出决定，自动点唱机需要歌曲。因此，首先编写一个表示歌曲的 Ruby 类，似乎是一个好主意。我们知道现实中的歌曲有名字、演唱者和时长，那么，我们需要确保程序中的歌曲对象也包括这些信息。

我们将首先创建一个基础的类 `Song`¹，它包括了一个方法，`initialize`。

```
class Song
  def initialize(name, artist, duration)
    @name = name
    @artist = artist
    @duration = duration
  end
end
```

在 Ruby 程序中，`initialize` 是一个特殊的方法。当你调用 `Song.new` 创建一个新的 `song` 对象时，Ruby 首先分配一些内存来保存未初始化的对象，然后调用对象的

¹ 如我们在第 16 页提到的，类名以一个大写字母开始，而方法名通常以一个小写字母开始。

`initialize` 方法，并把调用 `new` 时所使用的参数传入该方法。这为你提供了机会来编写代码设置对象的状态。

对 `Song` 类来说，`initialize` 方法接收 3 个参数。这些参数的作用同方法内的局部变量一样，因此它们遵循了局部变量命名的约定，即以小写字母开头。

每个对象都表示自己对应的歌曲，因此我们需要每个 `Song` 对象带有自己的歌曲名、演唱者和时长。这意味着我们需要将这些值作为 **实例变量** (*instance variables*) 保存在对象中。对象内的所有方法都可以访问实例变量，每个对象都有实例变量的一份拷贝。

在 Ruby 中，实例变量就是一个由@符开头的名字。在我们的示例中，参数 `name` 被赋值给实例变量`@name`，`artist` 被赋值给`@artist`，`duration`（以秒计算的歌曲长度）被赋值给`@duration`。

让我们测试一下这个绝妙的类。

```
song = Song.new("Bicyclops", "Fleck", 260)
song.inspect → #<Song:0x1c8ac8, @duration=260, @artist="Fleck",
                 @name="Bicyclops">
```

好的，它似乎可以工作了。`inspect` 方法（可以发送给任何对象）默认将对象的 ID 和实例变量格式化。看起来我们已经正确地设置了它们。

经验告诉我们，在开发过程中，我们经常需要将 `Song` 对象的内容输出出来，而 `inspect` 默认的格式化有时并不合用。值得庆幸的是，Ruby 有一个标准消息（message）`to_s`，它可以发送给任何一个想要输出字符串表示的对象。让我们用歌曲对象尝试一下。

```
song = Song.new("Bicyclops", "Fleck", 260)
song.to_s → "#<Song:0x1c8ce4>"
```

这输出没什么用处——它只是报告了对象的 ID。因此，让我们覆写（override）`Song` 类中的 `to_s` 方法。同时，应该用一点时间来说明一下，我们在本书中是如何表示类的定义的。

在 Ruby 中，类永远都不是封闭的：你总可以向一个已有的类中添加方法。这适用于你自己编写的类，同样适用于标准的内建（built-in）类。只要打开某个已有类的类定义，你就可以将指定的新内容添加进去。

这对我们来说太棒了。在本章中，当我们为类添加新特性（feature）时，我们只需展示新方法的定义；而那些原有的方法依然存在。这避免了我们在每个示例中重复冗余的内容。不过，显然，如果你从头编写代码，最好把所有方法都放到单独的一个类定义中。

说明得足够细致了！让我们向 Song 类添加一个 `to_s` 方法吧。我们使用#字符将 3 个实例变量的值插入到字符串中。

```
class Song
  def to_s
    "Song: #{@name}--#{@artist} (#@duration)"
  end
end
song = Song.new("Bicyclops", "Fleck", 260)
song.to_s → "Song: Bicyclops--Fleck (260)"
```

好极了，我们取得了很大的进展。不过，我们没有深究某些微妙的东西。我们说过，Ruby 的所有对象都支持 `to_s`，但我们并没有介绍如何支持。答案涉及继承（inheritance）、子类化（subclassing），以及当你向一个对象发送消息时 Ruby 如何判定运行哪个方法。这是下一节的主题，因此且听下回分解……

3.1 继承和消息

Inheritance and Messages

继承允许你创建一个类，作为另一个类的精炼（refinement）和特化（specialization）。例如，在我们的自动点唱机系统中，有“歌曲”这一概念，被封装在 `Song` 类中。然后，随着市场的成长，我们需要提供卡拉OK的支持。一首卡拉OK歌曲和其他歌曲没什么两样（它只是没有主唱的音轨，对此我们不必关心）。不过，它还包括对应的一套歌词以及时间信息。当我们的自动点唱机在播放一首卡拉OK歌曲时，歌词应该随音乐滚动显示在点唱机前的屏幕上。

解决这个问题的一种方法是定义一个新的类 `KaraokeSong`，就是 `Song` 加上歌词。

```
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end
```

类定义一行中的“`< Song`”告诉 Ruby，`KaraokeSong` 是 `Song` 的子类（subclass）。当然，这也意味着 `Song` 是 `KaraokeSong` 的超类（superclass）。人们通常称之为“父子”关系，因此 `KaraokeSong` 的父类是 `Song`。现在，不必过于担心 `initialize` 方法；我们将稍后讨论 `super` 调用。

让我们创建一个 `KaraokeSong` 对象，并查看代码是否正常工作（在最终的系统中，歌词保存在一个对象中，其中包括文本以及时间信息）。不过，为了测试我们的类，这里还是使用字符串。动态类型语言的另一个好处是——在代码开始运行之前，我们无须定义任何东西。

```
song = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
song.to_s      → "Song: My Way--Sinatra (225)"
```

好的，它可以运行。但是为什么 `to_s` 方法没有显示歌词呢？

这和我们在向一个对象发送消息时，Ruby 判定调用哪个方法的机制有关。在程序代码的初始解析（parse）期间，当 Ruby 遇到方法调用 `song.to_s` 时，它并不知道从何处找到 `to_s` 方法，而是将判定推迟直至程序开始运行时再进行。在那时，Ruby 查看 `song` 所属的类。如果该类实现了和消息名称相同的方法，就运行这个方法。否则，Ruby 就查看其父类中的方法，然后是祖父类，凡此以往追溯整个祖先链。如果最终它在祖先类中没有找到合适的方法，Ruby 会产生一种特殊的行为，通常是导致引发一个错误。²

回到我们的示例。我们向 `song`——即 `KaraokeSong` 类的一个对象——发送消息 `to_s`。Ruby 在 `KaraokeSong` 类中查找要调用的方法 `to_s`，但是没有找到。然后解释器（interpreter）查看 `KaraokeSong` 的父类 `Song` 类，结果找到了我们在第 26 页中定义的 `to_s` 方法。这就是 Ruby 输出了歌曲的细节但却没有歌词的原因——`Song` 类并不知道歌词。

让我们通过实现 `KaraokeSong#to_s` 来解决这个问题，你有许多方法可以完成它。让我们从最糟糕的方法开始，我们将 `to_s` 方法从 `Song` 类中拷贝出来并添加 `lyrics` 信息。

```
class KaraokeSong
  #
  def to_s
    "KS: #@name--#@artist (#@duration) [#@lyrics]"
  end
end
song = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
song.to_s → "KS: My Way--Sinatra (225) [And now, the...]"
```

我们正确地显示了实例变量 `@lyrics` 的值。但使用这种方法，子类需要直接访问其祖先的实例变量。那么为什么这是实现 `to_s` 的一种糟糕方式呢？

答案与良好的编程风格有关（有时被称为解耦）。直接戳进父类的内部结构，并且显式地检验它的实例变量，会使得我们和父类的实现紧密地绑在一起。假如我们决定修改 `Song` 类以毫秒保存时长。突然间，`KaraokeSong` 开始报告非常荒谬的时长。“`My Way`”的卡拉OK版本有 3 750 分钟长，这想法太恐怖了。

² 实际上，你可以拦截这个错误，让你可以在运行时伪造方法的调用。这在第 572 页的 `Object#method_missing` 一节中描述。

我们通过让每个类处理其自身实现细节的方法来解决这个问题。当调用 `KaraokeSong#to_s` 时，我们调用其父类的 `to_s` 方法来得到歌曲的细节。然后，将歌词信息添加上去，并返回结果。这里使用的技巧是 Ruby 的关键字 `super`。当你调用 `super` 而不使用参数时，Ruby 向当前对象的父类发送一个消息，要求它调用子类中的同名方法。Ruby 将我们原先调用方法时的参数传递给父类的方法。现在，我们可以实现改进后新的 `to_s` 方法。

```
class KaraokeSong < Song
  # Format ourselves as a string by appending
  # our lyrics to our parent's #to_s value.
  def to_s
    super + " #{@lyrics}"
  end
end
song = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
song.to_s → "Song: My Way--Sinatra (225) [And now, the...]"
```

我们明确地告诉 Ruby，`KaraokeSong` 是 `Song` 的子类，但是我们并没有指定 `Song` 类本身的父类是什么。如果你在定义一个类时没有指定其父类，Ruby 默认以 `Object` 类作为其父类。这意味着所有类的始祖都是 `Object`，并且 `Object` 的实例方法对 Ruby 的所有对象都可用。回到第 26 页，我们说过 `to_s` 对所有对象都是可用的。现在我们知道原因了：`Object` 类中有超过 35 个的实例方法，而 `to_s` 是其中之一。完整的列表从第 567 页开始。

在本章中，目前我们已经考察了类以及方法。现在，是将目标转移到对象的时候了，比如 `Song` 类的实例。

3.2 对象和属性

Objects and Attributes

我们现在所创建的 `Song` 对象有内部的状态（例如歌曲名称和演唱者）。这些状态是对象所私有的——其他对象无法访问一个对象的实例变量。总体上讲，这是一件好事。这意味着只有对象才有责任维护其自身的一致性。

不过，完全密封的对象实在没什么用处——你可以创建它，但接下来你对它什么都不能做。你通常会定义一些方法来访问及操作对象的状态，让外部世界得以与之交互。一个对象的外部可见部分被称为其属性（attribute）。

对我们的 `Song` 对象来说，需要做的第一件事情是找出歌曲的名称和演唱者（这样我们可以在歌曲播放时显示它们）以及时长（这样我们可以显示某种进度条）。

继承与 Mixin

Inheritance and Mixins

某些面向对象语言（例如 C++）支持多继承，在这些语言中，一个类可以有多于一个的直接父类，并继承每者的功能。虽然强大，但这种技术是有危险的，会使继承结构变得混乱。

其他语言，例如 Java 和 C#，仅支持单继承。其中，一个类只能有一个直接的父类。虽然很清爽（并且容易实现），但是单继承也有缺点——在现实世界中，对象通常从多个源继承属性（例如，足球是一个有弹性的、圆球状的东西）。

Ruby 提供了一种有趣且强大的折中，给了你单继承的简单性以及多继承的强大功能。Ruby 类只有一个直接的父类，因此 Ruby 是一门单继承语言。不过，Ruby 类可以从任何数量的 *mixin*（类似于一种部分的类定义）中引入（*include*）功能。这提供了可控的、类似多继承的能力，而没有多继承的缺点。我们将在第 118 页更详细地探讨 *mixin*。

```
class Song
  def name
    @name
  end
  def artist
    @artist
  end
  def duration
    @duration
  end
end
song = Song.new("Bicyclops", "Fleck", 260)
song.artist      →      "Fleck"
song.name        →      "Bicyclops"
song.duration   →      260
```

这里，我们定义了 3 个访问方法来得到这 3 个实例变量的值。举例来说，方法 `name()` 返回实例变量 `@name` 的值。因为这是一个常见的惯用手法，Ruby 提供了一种方便的快捷方式：`attr_reader` 为你创建这些访问方法³。

```
class Song
  attr_reader :name, :artist, :duration
end
```

³ 译注：`attr_reader` 实际上是 Ruby 的一个方法，而`:name` 等符号（Symbol）是其参数。它会通过代码求解（code evaluation）动态地在 `Song` 类中加入实例方法体。这也是 Ruby meta-programming 的一个示例。

```

song = Song.new("Bicyclops", "Fleck", 260)
song.artist      →      "Fleck"
song.name       →      "Bicyclops"
song.duration   →      260

```

这个示例引入了一些新的内容。构成本体:`:artist` 是一个表达式，它返回对应 `artist` 的一个 `Symbol` 对象。你可以将`:artist` 看作是变量 `artist` 的名字，而普通的 `artist` 是变量的值。在这个示例中，我们将访问方法命名为 `name`、`artist` 和 `duration`。对应的实例变量`@name`、`@artist` 和`@duration` 会自动被创建。这些访问方法和我们早先手写的一些是等同的。

3.2.1 可写的属性

Writable Attributes

有些时候，你需要能够在一个对象外部设置它的属性。例如，让我们假定某一首歌的时长最初是预估得来的（可能从 CD 或 MP3 数据的信息中产生而来）。当我们第一次播放这首歌时，会得到它实际的长度，并将这个新的值保存回 `Song` 对象中。

在类如 C++ 和 Java 等语言中，你需要用 `setter` 方法来完成这个任务。

```

class JavaSong {                                // Java 代码
    private Duration _duration;
    public void setDuration(Duration newDuration) {
        _duration = newDuration;
    }
}
s = new JavaSong(...);
s.setDuration(length);

```

在 Ruby 中，访问对象属性就像访问其他变量一样。我们在前面已经看到，例如 `song.name` 语句。因此，当你想要设置某个属性的值时，对这些变量直接赋值似乎更自然些。在 Ruby 中，你可以通过创建一个名字以等号结尾的方法来达成这一目标。这些方法可以作为赋值操作的目标。

```

class Song
  def duration=(new_duration)
    @duration = new_duration
  end
end
song = Song.new("Bicyclops", "Fleck", 260)
song.duration → 260
song.duration = 257    # set attribute with updated value
song.duration → 257

```

赋值操作 `song.duration = 257` 调用了 `song` 对象中的 `duration=` 方法，并将 257 作为参数传入。实际上，定义一个以等号结尾的方法名，便使其能够出现在赋值操作的左侧。

同样，Ruby 又提供了一种快捷方式来创建这类简单的属性设置方法。

```
class Song
  attr_writer :duration
end
song = Song.new("Bicyclops", "Fleck", 260)
song.duration = 257
```

3.2.2 虚拟属性

Virtual Attributes

这类属性访问的方法并不必须是对象实例变量的简单包装（wrapper）。例如，你可能希望访问以分钟为单位，而不是我们已经实现的以秒为单位的时长。

```
class Song
  def duration_in_minutes
    @duration/60.0 # force floating point
  end
  def duration_in_minutes=(new_duration)
    @duration = (new_duration*60).to_i
  end
end
song = Song.new("Bicyclops", "Fleck", 260)
song.duration_in_minutes → 4.333333333333333
song.duration_in_minutes = 4.2
song.duration → 252
```

这里我们使用属性方法创建了一个虚拟的实例变量。对外部世界来说，`duration_in_minutes` 就像其他属性一样。然而，在内部它没有对应的实例变量。

这远非出于讨巧。在 Bertrand Meyer 的划时代著作 *Object-Oriented Software Construction* [Mey97] 中，他将它称为统一访问原则（Uniform Access Principle）。通过隐藏实例变量与计算的值的差异，你可以向外部世界屏蔽类的实现。将来你可以自由地更改对象如何运作，而不会影响使用了你的类的、数以百万行计的代码。这是一种很大的成功。

3.2.3 属性、实例变量及方法

Attributes, Instance Variables, and Methods

这种对属性的描述，可能会让你认为它们无外乎就是方法——为什么我们要为它们发明一个新奇的名字呢？从某种程度上，这绝对是正确的。属性就是一个方法。某些时候，属性简单地返回实例变量的值。某些时候，属性返回计算后的结果。并且某些时

候，那些名字以等号结尾的古怪方法，被用来更新对象的状态。那么问题是，哪里是属性止而一般方法始的地方呢？是什么使之成为属性，而非普通老式的方法呢？总之，这是一个值得着墨的问题。下面是我个人的看法。

当你设计一个类的时候，你决定其具有什么样的内部状态，并决定这内部状态对外界（类的用户）的表现形式。内部状态保存在实例变量中。通过方法暴露出来的外部状态，我们称之为**属性**（*attributes*）。你的类可以执行的其他动作，就是一般方法。这并非是一个非常重要的区别，但是通过把一个对象的外部状态称为**属性**，可以帮助人们了解你所编写的类。

3.3 类变量和类方法

Class Variables and Class Methods

到目前为止，所有我们创建的类都包括有实例变量和实例方法：变量被关联到类的某个特定实例，以及操作这些变量的方法。有时候，类本身需要它们自己的状态。这是类变量的领地。

3.3.1 类变量

Class Variables

类变量被类的所有对象所共享，它与我们稍后描述的类方法相关联。对一个给定的类来说，类变量只存在一份拷贝。类变量由两个@符开头，例如`@@count`。与全局变量和实例变量不同，类变量在使用之前必须被初始化。通常，初始化就是在类定义中的简单赋值。

例如，我们的点唱机可能希望记录每首歌被播放的次数。这个数目可能是`Song`对象的一个实例变量。当一首歌被播放时，实例中的值增加。但是，假如我们还想要了解一共播放了多少首歌。通过搜索所有`Song`对象并累加它们的播放次数，或者冒天下之大不韪使用全局变量来完成统计；或者，让我们使用类变量。

```
class Song
  @@plays = 0
  def initialize(name, artist, duration)
    @name      = name
    @artist    = artist
    @duration  = duration
    @plays     = 0
  end
```

```

def play
  @plays += 1    # same as @plays = @plays + 1
  @@plays += 1
  "This song: #{@plays} plays. Total #{@@@plays} plays."
end
end

```

出于调试的目的，我们还让 `Song#play` 返回一个字符串，其中包括该歌曲被播放的次数，以及所有歌曲播放的总次数。我们可以很容易地测试它。

```

s1 = Song.new("Song1", "Artist1", 234) # test songs..
s2 = Song.new("Song2", "Artist2", 345)
s1.play      →      "This song: 1 plays. Total 1 plays."
s2.play      →      "This song: 1 plays. Total 2 plays."
s1.play      →      "This song: 2 plays. Total 3 plays."
s1.play      →      "This song: 3 plays. Total 4 plays."

```

类变量对类及其实例都是私有的。如果你想让它们能够被外部世界访问，你需要编写访问方法。这个方法要么是一个实例方法，或者是紧接着在下一节要介绍的类方法。

3.3.2 类方法 Class Methods

有时，类需要提供不束缚于任何特定对象的方法。我们已经见过一个这样的方法。`new` 方法创建一个新的 `Song` 对象，但是 `new` 方法本身并不与一个特定的歌曲对象相关联。

```
song = Song.new(....)
```

我们会发现类方法遍布于 Ruby 库中。例如，`File` 类的对象用来表示在底层文件系统中打开的一个文件。不过，`File` 类还提供了几个类方法来操作文件，而它们并未打开文件，因此也没有相应的 `File` 对象。如果你想要删除一个文件，你可以调用类方法 `File.delete`，传入文件名作为参数。

```
File.delete("dommed.txt")
```

类方法和实例方法是通过它们的定义区别开来的；通过在方法名之前放置类名以及一个句点，来定义类方法（请参见第 36 页的注解）。

```

class Example
  def instance_method      # instance method
  end
  def Example.class_method # class method
  end
end

```

点唱机按播放的每首歌曲收费，而非按分钟收费。这使得短歌曲比长歌更有助于盈收。我们可能希望避免那些过长的歌曲出现在 SongList 中。我们可以在 SongList 中定义一个类方法来检查，是否某一首歌超过了时限。我们使用一个类的常量来设置时限，就是一个在类代码体中初始化的常量（记得常量吗？它们使用一个大写字母开头）。

```
class SongList
  MAX_TIME = 5*60          # 5 minutes

  def SongList.is_too_long(song)
    return song.duration > MAX_TIME
  end
end
song1 = Song.new("Bicyclops", "Fleck", 260)
SongList.is_too_long(song1) → false
song2 = Song.new("The Calling", "Santana", 468)
SongList.is_too_long(song2) → true
```

3.3.3 单件与其他构造函数

Singletons and Other Constructors

有些时候，你希望覆写（override）Ruby 默认创建对象的方式。举例来说，看看我们的点唱机。因为我们将有许多点唱机分布在全国各地，我们希望让维护工作尽可能简单。其中一个需求是记录点唱机发生的所有事情：播放的歌曲、收到的点歌费、倾倒进去的各种奇怪液体等等。因为我们希望为音乐保留网络带宽，因此我们将日志文件保存在本地。这意味着我们需要一个类来处理日志。不过，我们希望每个点唱机只有一个记录日志的对象，并且我们希望其他所有对象都共享同一个日志对象。

这里适用 Singleton 模式，在《*Design Patterns* (设计模式)》[GHJV95] 一书中记载。我们将进行一些调整，使得只有一种方式来创建日志对象，那就是调用 MyLogger.create，并且我们还保证只有一个日志对象被创建。

```
class MyLogger
  private_class_method :new
  @@logger = nil
  def MyLogger.create
    @@logger = new unless @@logger
    @@logger
  end
end
```

通过将 MyLogger 的 new 方法标记为 private (私有)，我们阻止所有人使用传统的构造函数来创建日志对象。相反，我们提供了一个类方法，MyLogger.create。这个方法使用了类变量@@logger 来保存唯一一个日志对象实例的引用，并在每次被调用时返回

类方法定义

Class Method Definitions

回到第 32 页，我们说过，通过将类名和一个句点放在方法名之前来定义类方法。这实际上是最简单的一种形式（但它在任何时候都有效）。

实际上，你可以用很多方式来定义类方法，但是理解这些方式为什么有效，我们需要等到第 24 章。现在，我们只向你演示人们惯用的技法，供你在 Ruby 代码中遇到它们时参考。

下面是在类 Demo 内定义类方法。

```
class Demo
  def Demo.meth1
    # ...
  end
  def self.meth2
    # ...
  end
  class <<self
    def meth3
      # ...
    end
  end
end
```

这个实例⁴。我们可以通过查看方法返回对象的标识符来检验。

MyLogger.create.object_id	→	946790
MyLogger.create.object_id	→	946790

使用类方法作为伪构造函数（pseudo-constructors），可以让使用类的用户更轻松些。举个简单例子，让我们看一个表示等边多边形的 Shape 类。通过将所需的边数和周长传入构造函数来创建 Shape 的实例。

```
class Shape
  def initialize(num_sides, perimeter)
    # ...
  end
end
```

⁴ 我们这里演示的 singleton 实现并非是线程安全的；如果多个线程在运行，有可能会创建多个日志对象。与其我们自己添加线程安全，不如使用 Ruby 提供的 Singleton mixin，将在第 733 页介绍。

不过，几年之后，另一个不同的应用使用了这个类，其中的程序员习惯于通过名字、并指定每条边的长度而不是周长来创建图形。只需要在 `shape` 中添加若干类方法。

```
class Shape
  def Shape.triangle(side_length)
    Shape.new(3, side_length*3)
  end
  def Shape.square(side_length)
    Shape.new(4, side_length*4)
  end
end
```

类方法有许多有趣并强大的用途，为了使我们的点唱机产品尽可能快地完成，这里就不再继续深入探究了，让我们继续前进吧。

3.4 访问控制

Access Control

当我们设计类的接口时，需要着重考虑你的类想要向外部世界暴露何种程度的访问。如果你的类允许过度的访问，会增加应用中耦合的风险——类的用户可能会依赖于类实现的细节，而非逻辑性的接口。好消息是，在 Ruby 中改变一个对象的状态，唯一简单方式是调用它的方法。控制对方法的访问，你就得以控制对对象的访问。一个经验法则是，永远不要暴露会导致对象处于无效状态的方法。Ruby 为你提供了三种级别的保护。

- **Public**（公有）方法可以被任何人调用，没有限制访问控制。方法默认是 `public` 的（`initialize` 除外，它是 `private` 的）。
- **Protected**（保护）方法只能被定义了该方法的类或其子类的对象所调用。整个家族均可访问。
- **Private**（私有）方法不能被明确的接收者调用，其接收者只能是 `self`。这意味着私有方法只能在当前对象的上下文中被调用；你不能调用另一个对象的私有方法。

“`protected`” 和 “`private`” 之前的区别很微妙，并且和其他大多数普通的面向对象语言都不同。如果方法是保护的，它可以被定义了该方法的类或其子类的实例所调用。如果方法是私有的，它只能在当前对象的上下文中被调用——不可能直接访问其他对象的私有方法，即便它与调用者都属同一个类的对象。

Ruby 和其他面向对象语言的差异，还体现在另一个重要的方面。访问控制是在程序运行时动态判定的，而非静态判定。只有当代码试图执行受限的方法，你才会得到一个访问违规。

3.4.1 指定访问控制 Specifying Access Control

你可以使用 `public`、`protected` 和 `private` 3 个函数（function），来为类或模块定义内的方法（method）指定访问级别。你可以以两种不同的方式使用每个函数。

如果使用时没有参数，这 3 个函数设置后续定义方法的默认访问控制。如果你是 C++ 或 Java 程序员，这应该是你熟悉的行为，你同样使用类似 `public` 的关键字来取得相同的效果。

```
class MyClass
  def method1 # default is 'public'
    ...
  end
  protected      # subsequent methods will be 'protected'
  def method2 # will be 'protected'
    ...
  end
  private        # subsequent methods will be 'private'
  def method3 # will be 'private'
    ...
  end
  public         # subsequent methods will be 'public'
  def method4 # and this will be 'public'
    ...
  end
end
```

另外，你可以通过将方法名作为参数列表传入访问控制函数来设置它们的访问级别。

```
class MyClass
  def method1
  end
  # ... and so on
  public    :method1, :method4
  protected :method2
  private   :method3
end
```

是时候来看一些示例了。假设我们要为一个会计系统建立模型，其中每个借方都有对应的信用。因为我们希望确保没有人能破坏这个规则，所以我们让有关借方和信用的方法均成为私有，并定义外部的交易接口。

```

class Accounts
  def initialize(checking, savings)
    @checking = checking
    @savings = savings
  end
  private
  def debit(account, amount)
    account.balance -= amount
  end
  def credit(account, amount)
    account.balance += amount
  end
  public
  #...
  def transfer_to_savings(amount)
    debit(@checking, amount)
    credit(@savings, amount)
  end
  #...
end

```

当对象需要访问同属类的其他对象的内部状态时，使用保护访问（protected access）方式。例如，我们希望单个的 `Account` 对象能够比较它们的原始余额，而对其余所有对象隐藏这些余额（可能因为我们要以一种不同的形式表现它们）。

```

class Account
  attr_reader :cleared_balance      # accessor method 'cleared_balance'
  protected :cleared_balance        # and make it protected
  def greater_balance_than(other)
    return @cleared_balance > other.cleared_balance
  end
end

```

因为属性 `balance` 是 `protected`，只有 `Account` 的对象才可以访问它。

3.5 变量

Variables

现在我们已经辗转创建了所有这些对象，让我们确保没有丢掉它们。变量用来保存这些对象的印迹；每个变量保存一个对象的引用。

让我们通过下面的代码来验证。

```

person = "Tim"
person.object_id → 938678
person.class → String
person → "Tim"

```

第一行代码，Ruby 使用值“Tim”创建了一个 String 对象。这个对象的一个引用（reference）被保存在局部变量 person 中。接下去的快速检查展示了这个变量具备字符串的特性，它具有对象的 ID、类和值。

那么，变量是一个对象吗？在 Ruby 中，答案是“不”。变量只是对象的引用。对象漂浮在某处一个很大的池中（大多数时候是堆，即 heap 中），并由变量指向它们。

让我们看一个稍复杂一些的例子。

```
person1 = "Tim"
person2 = person1

person1[0] = 'J'

person1 →      "Jim"
person2 →      "Jim"
```

发生了什么？我们改变了 person1 的第一个字符，但是 person1 和 person2 都从“Tim”改为了“Jim”。

这都归结于变量保存的是对象引用而非对象本身这一事实。将 person1 赋值给 person2 并不会创建任何新的对象；它只是将 person1 的对象引用拷贝给 person2，因此 person1 和 person2 都指向同一个对象。我们在下一页的插图 3.1 中展示了这一操作的结果。

赋值别名（alias）对象，潜在地给了你引用同一对象的多个变量。但这不会在你的代码中导致问题吗？它会的，但是并不像你想象的频繁（例如 Java 中的对象，也以相同的方式运作）。例如，在插图 3.1 的示例中，你可以通过使用 String 的 dup 方法来避免创建别名，它会创建一个新的、具有相同内容的 String 对象。

```
person1 = "Tim"
person2 = person1.dup
person1[0] = "J"
person1 →      "Jim"
person2 →      "Tim"
```

你可以通过冻结一个对象来阻止其他人对其进行改动（我们将在第 394 页详细地讨论冻结对象）。试图更改一个被冻结的对象，Ruby 将引发（raise）一个 TypeError 异常。

```
person1 = "Tim"
person2 = person1
person1.freeze # prevent modifications to the object
person2[0] = "J"
```

输出结果：

```
prog.rb:4:in '[]=': can't modify frozen string (TypeError)
from prog.rb:4
```

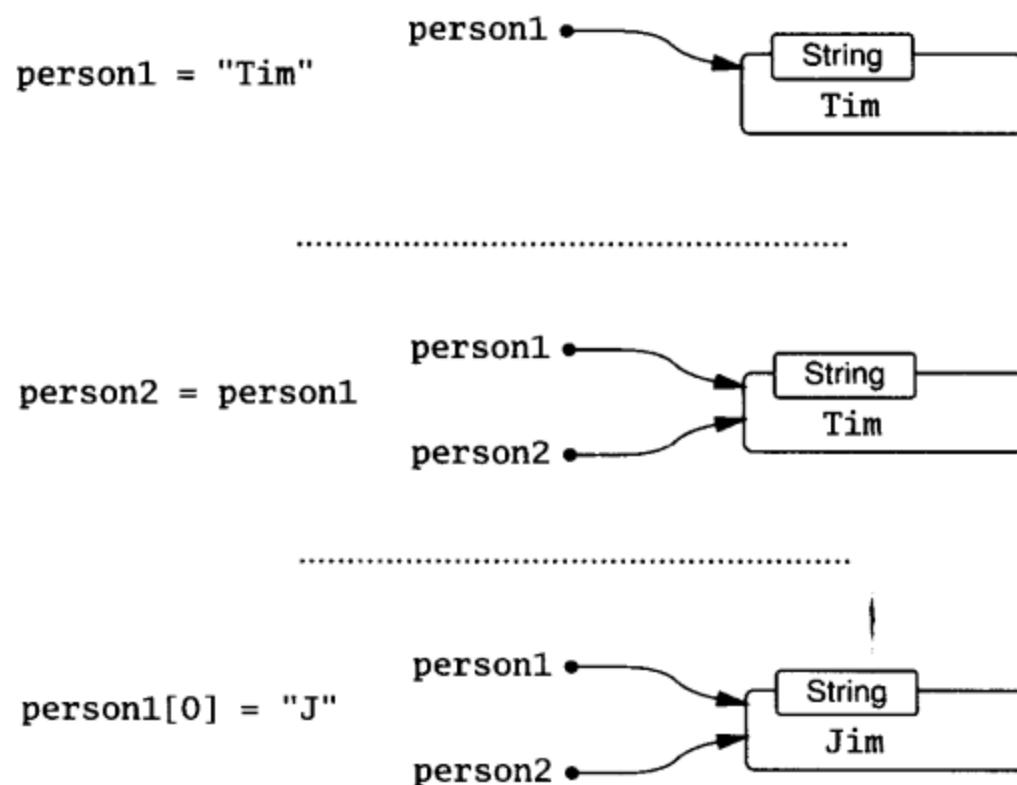


图 3.1 变量保存对象引用

本章总结了我们对 Ruby 中类和对象的考察。下面一点很重要：你在 Ruby 中操作的任何东西都是一个对象。当我们使用对象时，创建它们的集合是最常见的动作之一。但这是我们下一章的主题。





容器、Blocks 和迭代器

Containers, Blocks, and Iterators

只有一首歌曲的唱机是不可能受欢迎的（除非是在非常非常怪的酒吧里），所以我们应该开始考虑建立一个歌曲目录和待播放歌曲的列表。它们都是容器（*containers*），所谓容器是指含有一个或多个对象引用的对象。

目录和播放列表需要一组相似的方法：添加一首歌曲，删除一首歌曲，返回歌曲列表等等。播放列表可能还需要执行额外的任务，例如偶尔插播广告或者记录累计的播放时间，不过我们在后面才考虑这些问题。现在看来，开发一个通用的 `SongList` 类，然后将其特化（*specialize*）为目录和播放列表类，似乎是个好主意。

4.1 容器

Containers

在开始实现之前，我们需要决定如何在 `SongList` 对象中存储歌曲列表。目前有 3 个明显的选择：(1) 使用 Ruby 的 `Array`（数组）；(2) 使用 Ruby 的 `Hash`（散列表）；(3) 自定义列表结构。让我们偷一下懒，先来看看 `Array` 和 `Hash`，然后选择一个用来实现我们的类。

4.1.1 数组

Arrays

数组类含有一组对象引用。每个对象引用占据数组中的一个位置，并由一个非负的整数索引来标识。

可以通过使用字面量（*literal*），或显式地创建 `Array` 对象，来创建数组。字面量数组（*literal array*）只不过是处于方括号中的一组对象。

```

a = [ 3.14159, "pie", 99 ]
a.class      → Array
a.length     → 3
a[0]         → 3.14159
a[1]         → "pie"
a[2]         → 99
a[3]         → nil

b = Array.new
b.class      → Array
b.length     → 0
b[0] = "second"
b[1] = "array"
b           → ["second", "array"]

```

数组由`[]`操作符来进行索引。和 Ruby 的大多数操作符一样，它实际上是一个方法（`Array`类的一个实例方法），因此可以被子类重载。如上面例子所示，数组的下标从 0 开始。使用非负整数访问数组，将会返回处于该整数位置上的对象，如果此位置上没有对象，则返回`nil`。使用负整数访问数组，则从数组末端开始计数。

```

a = [ 1, 3, 5, 7, 9 ]
a[-1]      → 9
a[-2]      → 7
a[-99]     → nil

```

下一页的图 4.1 更详细地阐述了这种索引模式。

你也可以使用一对数字`[start, count]`来访问数组。这将返回一个包含从`start`开始的`count`个对象引用的新数组。

```

a = [ 1, 3, 5, 7, 9 ]
a[1..3]    → [3, 5, 7]
a[3..1]    → [7]
a[-3..2]   → [5, 7]

```

最后，你还可以使用`range`来对数组进行索引，其开始和结束位置被两个或 3 个点分割开。两个点的形式包含结束位置，而 3 个点的形式不包含。

```

a = [ 1, 3, 5, 7, 9 ]
a[1..3]    → [3, 5, 7]
a[1...3]   → [3, 5]
a[3..3]    → [7]
a[-3..-1]  → [5, 7, 9]

```

`[]`操作符有一个相应的`[]=`操作符，它可以设置数组中的元素。如果下标是单个整数，那么其位置上的元素将被赋值语句右边的东西所替换。造成的任何间隙将由`nil`来填充。

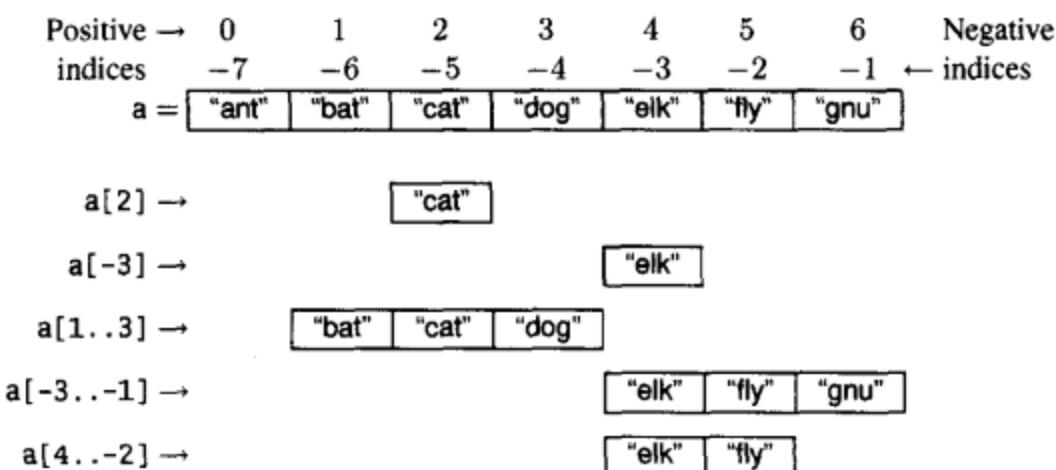


图 4.1 如何对数组进行索引操作

```

a = [ 1, 3, 5, 7, 9 ] → [1, 3, 5, 7, 9]
a[1] = ' bat' → [1, "bat", 5, 7, 9]
a[-3] = 'cat' → [1, "bat", "cat", 7, 9]
a[3] = [ 9, 8 ] → [1, "bat", "cat", [9, 8], 9]
a[6] = 99 → [1, "bat", "cat", [9, 8], 9, nil, 99]

```

如果`[] =`的下标是两个数字（起点和长度）或者是`range`，那么原数组中的那些元素将被赋值语句右边的东西所替换。如果长度是 0，那么赋值语句右边的东西将被插入到数组的起点位置之前，且不会删除任何元素。如果右边本身是一个数组，那么其元素将替换掉原数组对应位置上的东西。如果索引下标选择的元素个数和赋值语句右边的元素个数不一致，那么数组会自动调整其大小。

```

a = [ 1, 3, 5, 7, 9 ] → [1, 3, 5, 7, 9]
a[2, 2] = 'cat' → [1, 3, "cat", 9]
a[2, 0] = 'dog' → [1, 3, "dog", "cat", 9]
a[1, 1] = [ 9, 8, 7 ] → [1, 9, 8, 7, "dog", "cat", 9]
a[0..3] = [] → ["dog", "cat", 9]
a[5..6] = 99, 98 → ["dog", "cat", 9, nil, nil, 99, 98]

```

数组还有大量其他有用的方法。使用这些方法，你可以用数组来实现栈（stack）、收集（set）、队列（queue）、双向队列（dequeue）和先进先出队列（fifo）。从第 427 页开始列出了数组方法的完整列表。

4.1.2 散列表

Hashes

Hashes（也称关联数组、图或者词典）和数组的相似之处在于它们都是被索引的对象引用的集合。不过数组只能用整数来进行索引，而`hash`可以用任何类型的对象来进行索引，比如字符串、正则表达式等等。当你将一个值存入`hash`时，其实需要提供两个对象，一个是索引（通常称为键（key）），另一个是值。随后你可以通过键去索引`hash`

以获得其对应的值。`hash` 中的值可以是任意类型的对象。

下面的例子使用了 `hash` 字母符号表示法：处于花括号之间的 `key => value` 配对的列表。

```

h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine' }

h.length → 3
h['dog'] → "canine"
h['cow'] = 'bovine'
h[12] = 'dodecine'
h['cat'] = 99
h → {"cow"=>"bovine", "cat"=>99, 12=>"dodecine",
      "donkey"=>"asinine", "dog"=>"canine"}

```

和数组相比，`hashes` 有一个突出的优点：可以用任何对象做索引。然而它也有一个突出的缺点：它的元素是无序的，因此很难使用 `hash` 来实现栈和队列。

你会发现 `hash` 是 Ruby 最常用的数据结构之一。从第 492 页开始有 `Hash` 类的完整方法列表。

4.1.3 实现一个 SongList 容器

Implementing a SongList Container

在简单介绍了数组和 `hash` 后，该来实现点唱机的 `SongList` 了。让我们来设计一组 `SongList` 类所需的基本方法。之后我们会逐步扩充，但现在这些已经足够了。

`append(song) → list`

添加给定的歌曲到列表中。

`delete_first() → song`

删除列表的第一首歌曲，并返回该歌曲。

`delete_last() → song`

删除列表的最后一首歌曲，并返回该歌曲。

`[index] → song`

返回 `index` 处的歌曲。

`with_title(title) → song`

返回指定名字的歌曲。

这个列表为如何实现 `SongList` 给出了提示。既能在尾部添加歌曲，又能在头尾部删除歌曲，这提示我们使用双向队列（即有两个头部的队列）可以使用 `Array` 来实现它。同样，数组也支持返回列表中整数位置歌曲。

然而我们还需要使用歌名来检索歌曲，这可以通过以歌名为键、歌曲为值的 hash 来实现。我们可以用 hash 吗？也许可以，但会带来问题。首先，hash 是无序的，所以我们可能需要一个辅助数组来跟踪歌曲列表。第二，更重要的是 hash 不支持多个键对应一个相同的值。这对实现播放列表不利，因为播放列表可能需要播放同一首歌多次。因此，目前我们先使用数组来实现，并在需要的时候搜索歌名。如果这成为性能瓶颈，我们可以添加一些基于 hash 的搜索功能。

`SongList` 类的实现以一个基本的 `initialize` 方法开始，该方法会创建容纳歌曲的数组，并将该数组的引用存储到实例变量`@songs` 中。

```
Class SongList
  def initialize
    @songs = Array.new
  end
end
```

`SongList#append` 方法将给定的歌曲添加到`@songs` 数组的尾部。它会返回 `self`，即当前 `SongList` 对象的引用。这是一个很有用的惯用法，可以让我们把对 `append` 的多个调用链接在一起。我们会在后面看到它的例子。

```
class SongList
  def append(song)
    @songs.push(song)
    self
  end
end
```

接着我们来添加 `delete_first` 和 `delete_last` 方法，它们分别用 `Array#shift` 和 `Array#pop` 来实现。

```
class SongList
  def delete_first
    @songs.shift
  end
  def delete_last
    @songs.pop
  end
end
```

到目前为止，都还不错。下一个要实现的方法是`[]`，用它通过下标来访问元素。这种简单的代理形式的方法在 Ruby 代码中经常可见：如果你的代码含有大量一两行的方法，不用担心——那是你设计正确的迹象。

```
class SongList
  def [](index)
    @songs[index]
  end
end
```

现在需要做个简单的测试了。我们将使用 Ruby 标准发行版自带的一个称为 TestUnit 的测试框架来做这项工作。但是不会在这里详细介绍它（第 151 页的单元测试一章有详细介绍）。`assert_equal` 方法检查它的两个参数是否相等，如果不等则立刻报错。同样，`assert_nil` 方法在它的参数不为 `nil` 时也会报错。我们将会使用这些断言来保证从列表中删除恰当的歌曲。

测试需要必要的初始化，以告诉 Ruby 使用 TestUnit 测试框架，并告诉该框架我们在写一些测试代码。然后创建一个 `SongList` 对象和 4 首歌曲对象，并添加歌曲到列表中（趁机炫耀一下，我们利用了“`append` 返回 `SongList` 对象”这一特点来链接方法调用）。接着我们测试 `[]` 方法，验证它是否返回了指定下标处的正确的歌曲（或者 `nil`）。最后我们从列表的首尾删除歌曲，并验证返回正确的歌曲。

```
require 'test/unit'
class TestSongList < Test::Unit::TestCase
  def test_delete
    list = SongList.new
    s1 = Song.new('title1', 'artist1', 1)
    s2 = Song.new('title2', 'artist2', 2)
    s3 = Song.new('title3', 'artist3', 3)
    s4 = Song.new('title4', 'artist4', 4)

    list.append(s1).append(s2).append(s3).append(s4)

    assert_equal(s1, list[0])
    assert_equal(s3, list[2])
    assert_nil(list[9])

    assert_equal(s1, list.delete_first)
    assert_equal(s2, list.delete_first)
    assert_equal(s4, list.delete_last)
    assert_equal(s3, list.delete_last)
    assert_nil(list.delete_last)
  end
end
```

输出结果：

```
Loaded suite -
Started
.
Finished in 0.002314 seconds.

1 tests, 8 assertions, 0 failures, 0 errors
```

测试运行结果表明 1 种测试方法执行了 8 个断言，并且都通过了验证。我们开始踏上制作点唱机的征途了。

是时候来添加“以歌名来查找歌曲”的功能了。这要求遍历列表中的所有歌曲，并检查每首歌的名字。为了实现这项功能，我们先花几页的篇幅来看一下 Ruby 的一个最新特性：迭代器。

4.2 Blocks 和迭代器

Blocks and Iterators

实现 SongList 的下一个问题是实现 `with_title` 方法，该方法接受一个字符串参数，并返回以此为歌名的歌曲。看起来有个直接的实现方式：因为我们有歌曲数组，所以可以遍历该数组的所有元素，并查找出匹配的元素。

```
class SongList
  def with_title(title)
    for i in 0...@songs.length
      return @songs[i] if title == @songs[i].name
    end
    return nil
  end
end
```

这种方式确实可行，而且也相当常见：用 `for` 循环遍历数组。但是有没有更自然的方式呢？

的确有更自然的方式。在某种程度上，`for` 循环和数组的耦合过于紧密：需要知道数组的长度，然后依次获得其元素的值，直到找到一个匹配为止。为什么不只是请求数组对它的每一个元素执行一个测试呢？这正是数组的 `find` 方法要做的事情。

```
class SongList
  def with_title(title)
    @songs.find { |song| title == song.name }
  end
end
```

`find` 方法是一种迭代器，它反复调用 `block` 中的代码。迭代器和 `block` 是 Ruby 最有趣的特性之一，所以我们花一点时间来了解它们（随着逐渐深入，我们也会弄清 `with_title` 方法中的代码实际做了些什么）。

4.2.1 实现迭代器

Implementing Iterators

Ruby 的迭代器只不过是可以调用 `block` 的方法而已。乍看之下，Ruby 的代码区块和 C、Java、C#、Perl 的代码区块很相似。实际上这有点蒙蔽人——Ruby 的 `block` 不是传统意义上的、将语句组织在一起的一种方式。

首先，`block` 在代码中只和方法调用一起出现；`block` 和方法调用的最后一个参数处于同一行，并紧跟在其后（或者参数列表的右括号的后面）。其次，在遇到 `block` 的时候

并不立刻执行其中的代码。Ruby 会记住 block 出现时的上下文（局部变量、当前对象等）然后执行方法调用。这正是神奇之处。

在方法内部，block 可以像方法一样被 yield 语句调用。每执行一次 yield，就会调用 block 中的代码。当 block 执行结束时，控制返回到紧随 yield 之后的那条语句。¹ 我们来看个简单的例子。

```
def three_times
  yield
  yield
  yield
end
three_times { puts "Hello" }
```

输出结果：

```
Hello
Hello
Hello
```

block（花括号内的代码）和对方法 three_times 的调用联合在一起。该方法内部，连续 3 次调用了 yield。每次调用时，都会执行 block 中的代码，并且打印出一条欢迎信息。更有趣的是，你可以传递参数给 block，并获得其返回值。例如，我们可以写个简单的函数返回低于某个值的所有 Fibonacci 数列项。²

```
def fib_up_to(max)
  i1, i2 = 1, 1          # 并行赋值(i1 = 1 and i2 = 1)
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fib_up_to(1000) {|f| print f, " "}
```

输出结果：

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

¹ 程序语言迷们会很乐意看到 yield 关键字，它模仿了 Liskov 的 CLU 语言中的 yield 函数，CLU 是一个超过 20 年之久的语言，其包含的特性至今还未被完全开发出来。

² 基本 Fibonacci 数列是以两个 1 开头的整数序列，其中的每一项都是它的前两项的和。它常被用在排序算法和自然现象分析中。

在这个例子中，`yield` 语句带有一个参数。参数值将传送给相关联的 `block`。在 `block` 定义中，参数列表位于两个竖线（管道符）之间。在这个例子中，变量 `f` 收到 `yield` 的参数的值，所以 `block` 能够输出数列中的下一个项（这个例子也展示了并行赋值的用法，在第 91 页会详细介绍）。尽管通常 `block` 只有一个参数，但这不是必然的；`block` 可以有任意数量的参数。

如果传递给 `block` 的参数是已存在的局部变量，那么这些变量即为 `block` 的参数，它们的值可能会因 `block` 的执行而改变。同样的规则适用于 `block` 内的变量：如果它们第一次出现在 `block` 内，那么它们就是 `block` 的局部变量。相反，如果它们先出现在 `block` 外，那么 `block` 就与其外部环境共享这些变量。³

在下面这个（人为设计的）例子中，我们看到 `block` 从外围环境中继承了变量 `a` 和 `b`，而 `c` 是 `block` 的局部变量（`defined?`方法在其参数没有定义时返回 `nil`）。

```
a = [1, 2]
b = 'cat'
a.each {|b| c = b * a[1] }
a → [1, 2]
b → 2
defined?(c) → nil
```

`block` 也可以返回值给方法。`block` 内执行的最后一条表达式的值被作为 `yield` 的值返回给方法。这也是 `Array` 类的 `find` 方法的工作方式。⁴它的实现类似于下面的代码。

```
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end

[1, 3, 5, 7, 9].find { |v| v*v > 30 } → 7
```

上面的代码把数组的元素依次传递给关联的 `block`。如果 `block` 返回真，那么方法返回相应的元素。如果没有元素匹配，方法则返回 `nil`。这个例子展示了这种实现迭代器

³ 尽管有时非常有用，此特性也会造成意外的行为，并在 Ruby 社区中引起了激烈的争论。Ruby 2.0 有可能改变这种 `block` 继承局部变量的方式。

⁴ 实际上 `find` 方法是在 `Enumerable` 模块中定义的，而 `Array` 类包含了这个模块。

的方式的优点。数组类处理它擅长的事情，例如访问数组元素，而让应用程序代码集中精力处理特殊需求（本例子中的特殊需求是找到满足某些算术标准的数组项）。

一些迭代器是 Ruby 的许多收集 (*collections*) 类型所共有的。我们已经看了 `find` 方法。另外两个是 `each` 和 `collect`。`each` 可能是最简单的迭代器，它所做的就是连续访问收集的所有元素。

```
[ 1, 3, 5, 7, 9 ].each { |i| puts i }
```

输出结果：

```
1
3
5
7
9
```

`each` 迭代器在 Ruby 中有独特的作用：在第 103 页我们会描述它如何被用来实现 Ruby 语言的 `for` 循环，在第 120 页我们会看到如何定义 `each` 方法来自动地为类添加一大堆额外的功能。

另一个常用的迭代器是 `collect`，它从收集中获得各个元素并传递给 `block`。`block` 返回的结果被用来生成一个新的数组，例如：

```
["H", "A", "L"].collect { |x| x.succ } → ["I", "B", "M"]
```

迭代器并不仅局限于访问数组和 hash 中的已有数据。从 Fibonacci 的例子中，我们已经看到迭代器可以返回得到的值。这个功能被 Ruby 的输入/输出类所使用，这些类实现了一个迭代器接口以返回 I/O 流中的连续相继的行（或字节）。下面的例子使用了 `do...end` 来定义 `block`。这种方式定义 `block` 和使用花括号定义 `block` 的唯一区别是优先级：`do...end` 的绑定低于`{...}`。我们将在第 356 页讨论这种影响。

```
f = File.open("testfile")
f.each do |line|
  puts line
end
f.close
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

让我们再来看一个有用的迭代器。`inject`（名字有点难理解）方法（定义在 1.8 `Enumerable` 模块中）让你可以遍历收集的所有成员以累计出一个值。例如，使用下面的

代码你可以将数组中的所有元素加起来，并获得它们的累加和。

```
[1,3,5,7].inject(0) { |sum, element| sum+element} → 16
[1,3,5,7].inject(1) { |product, element| product*element} → 105
```

`inject` 是这样工作的：`block` 第一次被执行时，`sum` 被置为 `inject` 的参数，而 `element` 被置为收集的第一个元素。接下来每次执行 `block` 时，`sum` 被置为上次 `block` 被调用时的返回值。`inject` 的最后结果是最后一次调用 `block` 返回的值。还有一个技巧：如果 `inject` 没有参数，那么它使用收集的第一个元素作为初始值，并从第二个元素开始迭代。这意味着我们可以把前面的例子写成：

```
[1,3,5,7].inject { |sum, element| sum+element} → 16
[1,3,5,7].inject { |product, element| product*element} → 105
```

内迭代器和外迭代器

Ruby 实现迭代器的方式与其他如 C++ 和 Java 等语言实现迭代器的方式，值得我们来作一下比较。在 Ruby 中，迭代器集成于收集内部——它只不过是一个方法，和其他方法不同的是，每当产生新的值的时候调用 `yield`。使用迭代器的不过是和该方法相关联的一个代码 `block` 而已。

在其他语言中，收集本身没有迭代器。它们生成外部辅助对象（例如，Java 中基于 `Iterator` 接口的对象）来传送迭代器状态。从这点看来（当然还可从很多方面来看），Ruby 是一种透明的语言。你在写程序的时候，Ruby 语言能使你集中精力在你的工作上，而不是在语言本身上费神。

我们也值得花点时间看看为什么 Ruby 的内部迭代器并不总是最好的解决方案。当你需要把迭代器本身作为一个对象时（例如，将迭代器传递给一个方法，而该方法需要访问由迭代器返回的每一个值），它的表现就欠佳了。另外，使用 Ruby 内建的迭代器模式^{1.8} 也难以实现并行迭代两个收集。幸运的是，Ruby 1.8 提供了 `Generator` 库（第 683 页有详细描述），该库为解决这些问题实现了外部迭代器。

4.2.2 事务 Blocks

Blocks for Transactions

尽管 `block` 通常是和迭代器合用，但它还有其他用处。我们来看其中几个用法。

`block` 可以用来定义必须运行在事务控制环境下的代码。比如，你经常需要打开一个文件，对其内容作些处理，然后确保在处理结束后关闭文件。尽管可以用传统方式来实现，但也存在“应该由文件负责自身的关闭”这样的观点。我们可以用 `block` 来实现这种

需求。如下是一个简单且忽略了错误处理的例子：

```
class File
  def File.open_and_process(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end

File.open_and_process("testfile", "r") do |file|
  while line = file.gets
    puts line
  end
end
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

`Open_and_process` 是一个类方法，它可以独立于任何 `file` 对象来被使用。我们希望它接受与传统的 `File.open` 一样的参数，但并不关心这些参数到底是什么。所以我们用 `*args` 表示参数，这意味着“把传递给这个方法的实际参数收集到名字为 `args` 的数组中。”然后我们调用 `File.open`，并以 `*args` 作为参数。它将把数组参数扩展成独立的参数。最终的结果是，`open_and_process` 透明地将它所接收到的任意参数都传递给了 `File.open`。

一旦文件被打开，`open_and_process` 将调用 `yield`，并传递打开的文件对象给 `block`。当 `block` 返回时，文件即被关闭。通过这种方式，关闭打开文件的责任从文件使用者身上转移到了文件本身。

让文件管理它自己的生命周期的技术是如此重要，以至于 Ruby 的 `File` 类直接支持了这项技术。如果 `File.open` 有个关联的 `block`，那么该 `block` 将被调用，且参数是该文件对象，当 `block` 执行结束时文件会被关闭。这非常有趣，因为它意味着 `File.open` 有两种不同的行为：当和 `block` 一起调用时，它会执行该 `block` 并关闭文件；当单独调用时，它会返回文件对象。使得这种行为成为可能的是 `Kernel.block_given?` 方法，当某方法和 `block` 关联在一起调用时，`Kernel.block_given?` 将返回真。使用该方法，可以用下面的代码（同样也忽略了错误处理）实现类似于标准的 `File.open` 方法。

```

class File
  def File.my_open(*args)
    result = file = File.new(*args)
    # If there's a block, pass in the file and close
    # the file when it returns
    if block_given?
      result = yield file
      file.close
    end

    return result
  end
end

```

还有一点不足：前面的例子在使用 `block` 来控制资源时，我们还没有解决错误处理问题。如果想完整实现这些方法，那么即使处理文件的代码由于某种原因异常中断，我们也需要确保文件被关闭。后面谈到的异常处理（在第 107 页中）可以解决这个问题。

4.2.3 Blocks 可以作为闭包 Blocks Can Be Closures

让我们再回到点唱机上（还记得点唱机的例子吧）。在某些时候，我们需要处理用户界面——用户用来选择歌曲和控制点唱机的按钮。我们需要将行为关联到这些按钮上：当按“开始”按钮时，开始播放音乐。事实证明，Ruby 语言的 `block` 是实现这种需求的合适方式。假设点唱机的硬件制造商实现了一个 Ruby 扩展，该扩展提供了一个基本的按钮类（从第 275 页开始我们会讨论扩展 Ruby）。

```

start_button = Button.new("Start")
pause_button = Button.new("Pause")
# ...

```

当用户按其中一个按钮时会发生什么呢？硬件开发人员做了埋伏，使得按钮按下时，调用 `Button` 类的回调函数 `button_pressed`。向按钮类中添加功能的一个显而易见的方式是创建 `Button` 类的子类，并让每个子类实现自己的 `button_pressed` 方法。

```

class StartButton < Button
  def initialize
    super("Start")      # invoke Button's initialize
  end
  def button_pressed
    # do start actions...
  end
end

start_button = StartButton.new

```

这样做有两个问题。首先，这会导致大量的子类。如果 `Button` 类的接口发生变化，维护代价将会很高。其次，按下按钮引发的动作所处层次不当：它们不是按钮的功能，而是使用按钮的点唱机的功能。使用 `block` 可以解决这些问题。

```
songlist = SongList.new

class JukeboxButton < Button

  def initialize(label, &action)
    super(label)
    @action = action
  end

  def button_pressed
    @action.call(self)
  end
end

start_button = JukeboxButton.new("Start") { songlist.start }
pause_button = JukeboxButton.new("Pause") { songlist.pause }
```

上面代码的关键之处在于 `JukeboxButton#initialize` 的第二个参数。如果定义方法时在最后一个参数前加一个`&`（例如`&action`），那么当调用该方法时，Ruby 会寻找一个 `block`。`block` 将会被转化成 `Proc` 类的一个对象，并赋值给参数。这样可以像任意其他变量一样处理该参数。在上面的例子中，我们将它赋给了实例变量`@action`。这样当回调函数 `button_pressed` 被调用时，我们可以 `Proc#call` 方法去调用相应的 `block`。

但是当我们创建 `Proc` 对象时，到底获得了什么呢？有趣的是，我们得到的不仅仅是一堆代码。和 `block`（以及 `Proc` 对象）关联在一起的还有定义 `block` 时的上下文，即 `self` 的值、作用域内的方法、变量和常量。Ruby 的神奇之处是，即使 `block` 被定义时的环境早已消失了，`block` 仍然可以使用其原始作用域中的信息。在其他语言中，这种特性称之为闭包（closure）。

让我们来看一个故意设计的例子。该例使用了 `lambda` 方法，该方法将一个 `block` 转换成了 `Proc` 对象。

```
def n_times(thing)
  return lambda {|n| thing * n }
end

p1 = n_times(23)
p1.call(3)      → 69
p1.call(4)      → 92
p2 = n_times("Hello ")
p2.call(3)      → "Hello Hello Hello "
```

`n_times` 方法返回引用了其参数 `thing` 的 `Proc` 对象。尽管 `block` 被调用时，这个参数已经出了其作用域，但是 `block` 仍然可以访问它。

4.3 处处皆是容器

Containers Everywhere

容器、`block` 和迭代器是 Ruby 的核心概念。用 Ruby 写的代码越多，你就会发现自己对传统循环结构使用得越少。你会更多地写支持迭代自身内容的类，而且你会发现这些代码精简易读并易于维护。



标准类型

Standard Types

到目前为止，我们已经对点唱机有了一些好玩儿的实现，但同时有所取舍。前面我们讲到了数组、散列表和 proc。但还没有真正地谈到 Ruby 中的其他一些基本类型：数字（number）、字符串、区间（range）和正则表达式。在接下来的几页中会介绍这些 Ruby 语言的基石。

5.1 数字

Numbers

Ruby 支持整数和浮点数。整数可以是任何长度（其最大值取决于系统可用内存的大小）。一定范围内的整数（通常是 -2^{30} 到 $2^{30}-1$ 或 -2^{62} 到 $2^{62}-1$ ）在内部以二进制形式存储，它们是 Fixnum 类的对象。这个范围之外的整数存储在 Bignum 类的对象中（目前实现为一个可变长度的短整型集合）。这个处理是透明的，Ruby 会自动管理它们之间的来回转换。

```
num = 81
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end
```

输出结果：

```
Fixnum: 81
Fixnum: 6561
Fixnum: 43046721
Bignum: 1853020188851841
Bignum: 3433683820292512484657849089281
Bignum: 11790184577738583171520872861412518665678211592275841109096961
```

在书写整数时，你可以使用一个可选的前导符号，可选的进制指示符（0 表示八进制，0d 表示十进制[默认]，0x 表示十六进制或者 0b 表示二进制），后面跟一串符合适当进制的数字。下画线在数字串中被忽视（一些人在更大的数值中使用它们来代替逗号）。

```

123456          => 123456 # Fixnum
0d123456        => 123456 # Fixnum
123_456         => 123456 # Fixnum - 忽略了下画线
-543            => -543  # Fixnum - 负数
0xaabb          => 43707 # Fixnum - 十六进制
0377             => 255   # Fixnum - 八进制
-0b10_1010       => -42   # Fixnum - 二进制 (负数)
123_456_789_123_456_789 => 123456789123456789 # Bignum

```

控制字符的整数值可以使用?`\C-x` 和?`\cx` (`x` 的 control 版本, 是`x & 0x9f1`) 生成。元字符 (`x | 0x802`) 可以使用?`\M-x` 生成。元字符和控制字符的组合可以使用?`\M-\C-x` 生成。可以使用?`\\"序列` 得到反斜线字符的整数值。

```

?a              => 97    # ASCII character
?\n             => 10   # code for a newline (0x0a)
?\C-a           => 1     # control a = ?A & 0x9f = 0x01
?\M-a           => 225  # meta sets bit 7
?\M-\C-a        => 129  # meta and control a
?\C-?           => 127  # delete character

```

与原生体系结构的 `double` 数据类型相对应, 带有小数点和/或幂的数字字面量被转换成浮点对象。你必须在小数点之前和之后都给出数字 (如果把 `1.0e3` 写成 `1.e3`, Ruby 会试图调用 `Fixnum` 类的 `e3` 方法)。

所有数字都是对象, 并且可以对各种形式的消息 (它们在第 441, 484, 487, 501 和 562 页列出) 作出响应。因此, 与 (比如说) C++ 不一样, Ruby 使用 `num.abs` 而不是 `abs(num)` 去得到数字的绝对值。

整数也支持几种有用的迭代器。我们已经看到了一种: 前一页的代码示例中的 `6.times`。别的迭代器还有 `upto` 和 `downto`, 它们在两个整数之间分别向上和向下迭代。另外 `Numeric` 类提供了更通用的 `step` 方法, 它更像传统的 `for` 循环。

```

3.times { print "X "}
1.upto(5) { |i| print i, " " }
99.downto(95) { |i| print i, " " }
50.step(80, 5) { |i| print i, " " }

```

输出结果:

```
X X X 1 2 3 4 5 99 98 97 96 95 50 55 60 65 70 75 80
```

最后, 给 Perl 用户提出一个警告。那些只包含数字的字符串, 当在表达式中使用时, 不会被自动转换成数字。所以从文件中读取数字时常常会出现问题。比如, 也许我们想得到文件中每行中两个数字的和:

```

3 4
5 6
7 8

```

¹ 译注: 也就是 Ctrl+X 的值。

² 译注: 在 PC 的英文键盘上通常对应于 Alt 键。

下面的代码不会工作。

```
some_file.each do |line|
  v1, v2 = line.split # split line on spaces
  print v1 + v2, " "
end
```

输出结果：

34 56 78

这里的问题是因为输入是作为字符串而不是作为数字被读取的。加号操作符把两个字符串连接起来，这正是我们在输出中所看到的。可以使用 `Integer` 方法把字符串转换成整数来解决这个问题。

```
some_file.each do |line|
  v1, v2 = line.split
  print Integer(v1) + Integer(v2), " "
end
```

输出结果：

7 11 15

5.2 字符串

Strings

Ruby 字符串是 8 比特字节的序列。通常它们包含可打印字符，但这不是一个必要条件；字符串也可以包含二进制数据。字符串是 `String` 类的对象。

字符串字面量是处于分界符之间的字符序列，常常使用它来创建字符串。可以在字符串字面量中放置各种转义序列，要不然的话，很难在程序源文件中表示二进制数据。当程序被编译时，它们会被相应的二进制值替换。字符串分界符的类型决定了要被替换的程度。在单引号字符串中，两个连续的反斜线会被一个反斜线替换，而后面跟有一个单引号的反斜线变成一个单引号。

'escape using \"\"'	→ escape using "
'That\'s right'	→ That's right

双引号字符串支持更多的转义序列。最常用的恐怕是`\n`，回车换行符。第 321 页的表 22.2 给出了一个完整的转义序列表。除此之外，可以使用`#{expr}` 序列把任何 Ruby 代码的值放进字符串中。如果代码只是全局变量、类变量或实例变量的话，花括号可以忽略。

"Seconds/day: #{24*60*60}"	→ Seconds/day: 86400
"#{'Ho! '*3}Merry Christmas!"	→ Ho! Ho! Ho! Merry Christmas!
"This is line #\$."	→ This is line 3

1.8 要进行插入替换的代码可以是一条或多条语句，而不仅仅是一个表达式。

```
puts "now is # {def the(a)
           'the ' + a
         end
         the('time')
       }for all good coders..."
```

输出结果：

```
now is the time for all good coders...
```

另外还有3种方式去构建字符串字面量：`%q`，`%Q` 和 *here documents*。

`%q` 和 `%Q` 分别开始界定单引号和双引号的字符串（可以把 `%q` 看成薄的引号'，把 `%Q` 看成厚的引号"）。

```
%q/general single-quoted string/ → general single-quoted string
%Q/general double-quoted string/ → general double-quoted string
%Q{Seconds/day: #{24*60*60}} → Seconds/day: 86400
```

跟在 `q` 或 `Q` 后面的字符是分界符。如果它是开始（opening）的方括号“[”，花括号“{”，括号“(”或小于号“<”，字符串被一直读取直到发现相匹配的结束符号。

否则，字符串会被一直读取，直到出现下一个相同的分界符。分界符可以是任何一个非字母数字的单字节字符。

1.8 字母数字的单字节字符。

最后，可以使用 *here document* 构建字符串。

```
string = <<END_OF_STRING
The body of the string
is the input lines up to
one ending with the same
text that followed the '<<'
END_OF_STRING
```

here document 由源文件中的那些行但没有包含在<<字符后面指明终结字符串的行组成。一般情况下，终结符（terminator）必须在第一列出现。当然，如果把一个减号放在<<字符后面，就可以缩进编排终结符。

```
print <<-STRING1, <<-STRING2
Concat
STRING1
enate
STRING2
```

输出结果：

```
Concat
enate
```

注意：在这些例子中，Ruby 没有从这些字符串中去掉这些前导空格。

5.2.1 操作字符串

Working with Strings

字符串可能是 Ruby 中最大的内建类，它有 75 个以上的标准方法。在这里我们不会介绍所有的方法；程序库参考有一个完整的方法列表。反之，我们看看那些常用的字符串惯用技法，在日常的编码中很可能会用上它们。

回到点唱机。尽管被设计成与互联网相连，它也在本地硬盘上保存一些流行歌曲的备份。用这种方式，如果一只松鼠咬断了网络连接，我们仍然可以娱乐客户。

由于历史上的原因（难道还有别的原因？），歌曲列表在纯文本文件中按行保存。每行有歌曲文件的名称、歌曲的持续时间、作者和标题，它们都放在以竖线分割开的各个字段中。一个典型的文件可能是：

```
/jazz/j00132.mp3 | 3:45 | Fats Waller | Ain't Misbehavin'  
/jazz/j00319.mp3 | 2:58 | Louis Armstrong | Wonderful World  
/bgrass/bg0732.mp3 | 4:09 | Strength in Numbers | Texas Red  
: : : :
```

看看这些数据，在根据它们创建 `Song` 对象之前，很明显我们会使用 `String` 类的一些方法去抽取和清理这些字段。至少我们需要做：

- 把每行分割成各个字段；
- 把播放时间从 `mm:ss` 转换成秒；
- 删掉歌曲演唱者名字中的多余空格。

首先把每行分割成各个字段，`String#split` 可以很好地完成这件事情。在这个例子中，我们将`/\s*\|\s*/`正则表达式传递给 `split` 方法，无论 `split` 在哪里找到了竖线，竖线都可能会被多个空格围绕，正则表达式会把这行分割成各个字元（token）。因为从文件读取的行尾含有一个回车换行符，在应用 `split` 之前，我们要使用 `String#chomp` 去除它。

```
File.open("songdata") do |song_file|
  songs = SongList.new
  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    songs.append(Song.new(title, name, length))
  end

  puts songs[1]
end
```

输出结果：

```
Song: Wonderful World--Louis Armstrong (2:58)
```

不幸的是，无论是谁创建了这个文件，他在输入演唱者名字的时候，有时会包含多余空格。这在我们高科技的超级环绕的 Day-Glo 平板显示器上会显得很难看。所以进一

步往前走之前，最好是删除这些多余空格。有许多方式可以删除多余空格，但是最简单的方式可能是 `String#squeeze`，它修剪（`trim`）重复字符。我们会使用这个方法的 `squeeze!` 形式在字符串上进行修改。

```
File.open("songdata") do |song_file|
  songs = SongList.new

  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    name.squeeze!(" ")
    songs.append(Song.new(title, name, length))
  end

  puts songs[1]
end
```

输出结果：

```
Song: Wonderful WorldLouis Armstrong (2:58)
```

最终有了分秒格式的时间：文件说 2:58，我们需要秒数，它是 178。可以再次使用 `split` 把冒号周围的时间字段分割出来。

```
mins, secs = length.split(/:/)
```

相反，我们会使用其他相关的方法。`String#scan` 类似于 `split`，因为它根据模式（`pattern`）把字符串分成几块。但是与 `split` 不一样，使用 `scan` 可以指定希望这些块去匹配的模式。在这个例子中，我们想为分和秒都匹配一个或多个数字。这种匹配一个或多个数字的模式是 `/\d+/-`。

```
File.open("songdata") do |song_file|
  songs = SongList.new

  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    name.squeeze!(" ")
    mins, secs = length.scan(/\d+/)
    songs.append(Song.new(title, name, mins.to_i*60+secs.to_i))
  end

  puts songs[1]
end
```

输出结果：

```
Song: Wonderful World-Louis Armstrong (178)
```

点唱机有关键字搜索的能力。从歌曲标题或歌曲演唱者名字中给定一个词，它会列出所有匹配的记录。键入 `fats`，它也许会搜索出 Fats Domino、Fats Navarro 和 Fats Waller 的歌曲。我们会创建索引类来实现它。送入一个对象和一些字符串，它会用出现在这些字符串中的每个词（有两个或多个字符）来索引这个对象。这会展现 `String` 类的更多方法。

```

class WordIndex
  def initialize
    @index = {}
  end
  def add_to_index(obj, *phrases)
    phrases.each do |phrase|
      phrase.scan(/\w[-\w']+/) do |word| # extract each word
        word.downcase!
        @index[word] = [] if @index[word].nil?
        @index[word].push(obj)
      end
    end
  end
  def lookup(word)
    @index[word.downcase]
  end
end

```

`String#scan` 方法从字符串中抽取出匹配正则表达式的元素。在这个例子中，`\w[-\w']+模式`匹配可以在词中出现任意的字符，其后跟着在方括号内指定一个或多个字符（连字符，另一个组词字符，或单引号）。从第 68 页开始我们会更详细地讲述正则表达式。为了让搜索与大小写无关，在查找的时候，我们把抽取出来的词以及用作键的词都变成小写。注意第一个 `downcase!`方法名称结尾处的感叹号。就像与原先用到的 `squeeze!`方法一样，这个标识用来表示方法会在适当的位置修改接收者，在这个例子中，它把字符串变成小写。³

我们扩展了 `SongList` 类，这样当添加歌曲时它会索引歌曲，我们还添加了方法，其根据给定的词去查询歌曲。

```

class SongList
  def initialize
    @songs = Array.new
    @index = WordIndex.new
  end
  def append(song)
    @songs.push(song)
    @index.add_to_index(song, song.name, song.artist)
    self
  end
  def lookup(word)
    @index.lookup(word)
  end
end

```

³ 这个代码样例包含一个小错误：“Gone, Gone, Gone” 这首歌会被索引三次。能够给出解决办法吗？

最终我们测试了所有这些方法。

```
songs = SongList.new
song_file.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  mins, secs = length.scan(/\d+/)
  songs.append(Song.new(title, name, mins.to_i*60+secs.to_i))
end
puts songs.lookup("Fats")
puts songs.lookup("ain't")
puts songs.lookup("RED")
puts songs.lookup("WoRlD")
```

输出结果：

```
Song: Ain't Misbehavin'--Fats Waller (225)
Song: Ain't Misbehavin'--Fats Waller (225)
Song: Texas Red--Strength in Numbers (249)
Song: Wonderful World--Louis Armstrong (178)
```

在前面的这段代码中，`lookup` 方法返回了匹配的数组。当把数组传递给 `puts` 时，它只是简单地依次输出每个元素，这些元素由回车换行符分隔开。

我们完全可以再用 50 页篇幅来描述 `String` 类中的所有方法，不过还是让我们继续前进，看看更简单的数组类型：区间（range）。

5.3 区间

Ranges

区间无处不在：1 月到 12 月，0 到 9，`rare` 到 `well-done`，第 50 行到第 67 行等等。如果 Ruby 帮助我们对现实世界建模，看起来它自然会支持这些区间。实际上 Ruby 做得更好：它使用区间去实现 3 种不同的特性：序列（sequences）、条件（conditionals）和间隔（intervals）。

5.3.1 区间作为序列

Ranges as Sequences

区间的第一个且可能最自然的用法是：表达序列。序列有起点、终点以及在序列中产生连续值的方法。在 Ruby 中，使用“..”和“...”区间操作符来创建序列。两个点的形式是创建闭合的区间（包括右端的值），而 3 个点的形式是创建半闭半开的区间（不包括右端的值）。

```
1..10
'a'...'z'
my_array = [ 1, 2, 3 ]
0...my_array.length
```

与 Perl 的早期版本不一样，在 Ruby 中，区间没有在内部用列表（list）表示：1..100000 序列被存储为 Range 对象，它包含对两个 Fixnum 对象的引用。如果需要，可以使用 `to_a` 方法把区间转换成列表。

```
(1..10).to_a → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a → ["bar", "bas", "bat"]
```

区间实现了许多方法可以让我们迭代它们，并且以多种方式测试它们的内容。

```
digits = 0..9
digits.include?(5) → true
digits.min → 0
digits.max → 9
digits.reject { |i| i < 5 } → [5, 6, 7, 8, 9]
digits.each { |digit| dial(digit) } → 0..9
```

到现在为止，我们已显示了数字和字符串的区间。当然，就像对一个面向对象语言所期望的那样，Ruby 可以根据你所定义的对象来创建区间。唯一的限制是这些对象必须返回在序列中的下一个对象作为对 `succ` 的响应，而且这些对象必须是可以使用`<=>`来比较的。有时`<=>`也被称为太空船（*spaceship*⁴）操作符，它比较两个值，并根据第一个值是否小于、等于或大于第二个值，分别返回-1，0 或+1。

下面一个简单的类表示了几行#符号。我们可以将它用作点唱机基于文本界面的声量控制。

```
class VU
  include Comparable
  attr :volume
  def initialize(volume) # 0..9
    @volume = volume
  end
  def inspect
    '#' * @volume
  end
  # Support for ranges
  def <=>(other)
    self.volume <=> other.volume
  end
  def succ
    raise(IndexError, "Volume too big") if @volume >= 9
    VU.new(@volume.succ)
  end
end
```

因为 `VU` 类实现了 `succ` 和`<=>`方法，因此它可以作为区间。

```
medium_volume = VU.new(4)..VU.new(7)
medium_volume.to_a → [#####
medium_volume.include?(VU.new(3)) → false
```

⁴ 译注：概因形状相似而得名。

5.3.2 区间作为条件

Ranges as Conditions

除了表达序列之外，区间也可以当作条件表达式来使用。在这里它们表现得就像某种双向开关——当区间第一部分的条件为 `true` 时，它们就打开，当区间第二部分的条件为 `true` 时，它们就关闭。例如，下面的代码段，打印从标准输入得到的行的集合，每组的第一行包含 `start` 这个词，最后一行包含 `end` 这个词。

```
while line = gets
  puts line if line =~ /start/ .. line =~ /end/
end
```

在幕后区间跟踪了每种测试的状态。从第 100 页开始的循环描述中和第 342 页的语言小节中会给出一些它的例子。

1.8 早期的 Ruby 版本中，裸区间（bare range）可以在 `if`, `while` 和类似的语句中作为条件来使用。比如，可能会把先前的代码写成。

```
while gets
  print if /start/..end/
end
```

这种写法不再支持。不幸的是，它不会引发任何错误；且每次测试都会成功。

5.3.3 区间作为间隔

Ranges as Intervals

区间这个多面手的最后一一种用法是用间隔测试：看看一些值是否会落入区间表达的间隔内。使用`==`即 `case equality` 操作符可以做到这一点。

(1..10)	<code>==</code>	5	→ true
(1..10)	<code>==</code>	15	→ false
(1..10)	<code>==</code>	3.14159	→ true
('a'..'j')	<code>==</code>	'c'	→ true
('a'..'j')	<code>==</code>	'z'	→ false

第 98 页的一个 `case` 表达式的例子显示了这种测试，它判断了某一年的爵士风格。

5.4 正则表达式

Regular Expressions

回到第 63 页，当从文件中创建歌曲列表时，我们使用了正则表达式去匹配输入文件的字段分界符。我们声称 `line.split (/\\s*\\|\\s*/)` 表达式匹配一条可能被空格围绕的竖线。让我们更详细地探索正则表达式，看看这个声称为什么是准确的。

正则表达式被用来根据模式对字符串进行匹配。Ruby 提供了内建的支持，使得模式匹配和替换变得更方便和更简明。本节会介绍正则表达式的所有主要特性。在这里不会谈到一些细节：更多的信息请看第 324 页。

正则表达式是 `Regexp` 类型的对象。可以通过显式地调用构造函数或使用字面量形式 `/pattern/` 和 `%r{pattern}` 来创建它们。

```
a = Regexp.new('^\s*[a-z]') → /^\s*[a-z]/
b = /^s*[a-z]/ → /^s*[a-z]/
c = %r{^\s*[a-z]} → /^s*[a-z]/

```

一旦有了正则表达式对象，可以使用 `Regexp#match (string)` 或匹配操作符 `=~`（肯定匹配）和 `!~`（否定匹配）对字符串进行匹配。匹配操作符对 `String` 和 `Regexp` 对象均有定义。^{1.8} 匹配操作符至少有一个操作数必须为正则表达式。（早期版本的 Ruby 当中，这两个操作数可能都是字符串，且第二个操作数会暗中被转换成正则表达式。）

```
name = "Fats Waller"
name =~ /a/ → 1
name =~ /z/ → nil
/a/ =~ name → 1

```

匹配操作符返回匹配发生的字符位置。它们也有副作用，会设置一些 Ruby 变量。`$&` 得到与模式匹配的那部分字符串，`$`` 得到匹配之前的那部分字符串，而`$'` 得到匹配之后的那部分字符串。可以使用这些变量来编写 `show_regexp` 方法，以说明具体的模式在何处发生匹配。

```
def show_regexp(a, re)
  if a =~ re
    "#{$`}<<#${$&}>>#${$'}"
  else
    "no match"
  end
end

show_regexp('very interesting', /t/) → very in<<t>>eresting
show_regexp('Fats Waller', /a/) → F<<a>>ts Waller
show_regexp('Fats Waller', /ll/) → Fats Wa<<ll>>er
show_regexp('Fats Waller', /z/) → no match

```

这个匹配也设置了线程局部变量（thread-local variables），`$~` 与 `$1` 直到 `$9`。`$~` 变量是 `MatchData` 对象（从第 537 页开始描述），它持有你想知道的有关匹配的所有信息。`$1` 等持有匹配各个部分的值，我们在后面会谈到这些。对那些看到这种类似 Perl 变量名就心怀畏惧的人来讲，请少安毋躁，在本节结束时会有好消息。

5.4.1 模式

Patterns

每个正则表达式包含一种模式，用来对字符串进行正则表达式的匹配。

在模式内，除了.，|，(，)，[，]，{，}，+，\，^，\$，*和?字符之外，所有字符均匹配它们本身。

```
show-regexp('kangaroo', /angar/) → k<<angar>>oo
show-regexp('!@%&-_=+', /%&/) → !@<<%&>>-_=+
```

在这些特殊字符之前放置一个反斜线便可以匹配它们的字面量。这解释了我们曾经用到的分割歌曲行的/\s*\|\s*/模式。|\意味着“匹配竖线”。没有反斜线，|字符指的是替换（后面会讲到它）。

```
show-regexp('yes | no', /\|/) → yes <<|>> no
show-regexp('yes (no)', /\(no\)/) → yes <<(no)>>
show-regexp('are you sure?', /e\?/) → are you sur<<e?>>
```

反斜线后面跟着一个分母数字的字符被用来引入一个特殊的匹配构造，后面我们会介绍到。另外，正则表达式可能包含#{...}表达式替换。

锚点

默认情况下，正则表达式会试图发现模式在字符串中出现的第一个匹配。对“Mississippi”字符串匹配/iss/，它会找出从位置1开始的“iss”子字符串。但是如果想强迫模式只匹配字符串的开始或结束部分，怎么办？

^和\$模式分别匹配行首和行尾。它们常常被用来锚定（anchor）模式匹配：例如，只要option出现在行的开始处，/^option/就会匹配这个词。\\A序列匹配字符串的开始，而\\z和\\Z匹配字符串的结尾（实际上，除非字符串以\\n结束，\\Z才会匹配字符串的结尾，在这个例子中，它会在\\n之前匹配）。

```
show-regexp("this is\nthe time", /^the/) → this is\n<<the>> time
show-regexp("this is\nthe time", /is$/i) → this <<is>>\nthe time
show-regexp("this is\nthe time", /\Athis/) → <<this>> is\nthe time
show-regexp("this is\nthe time", /\Athe/) → no match
```

同样地，\\b和\\B模式分别匹配词的边界和非词（nonword）的边界。组词字符可以是字母、数字和下画线。

```
show-regexp("this is\nthe time", /\bis\b/) → this <<is>>\nthe time
show-regexp("this is\nthe time", /\Bis\b/) → th<<is>> is\nthe time
```

字符类

字符类是处于方括号之间字符的集合：[*characters*] 匹配方括号之间的任何单个字符。[aeiou] 会匹配元音，[.,;!?] 匹配标点符号等等。特殊正则表达式字符的意义——. | () | {+^\$*?}——在方括号里面是关闭的。不过，正常的字符串替换仍然发生，因此（例如）\b 表示回退空格，而\n 表示回车换行符（见第 321 页表 22.2）。此外，可以使用在下页表 5.1 中显示的缩写形式，例如\s 匹配任何空格，而不仅仅是字面量空格。这个表中第二部分的 POSIX 字符类，对应于 ctype(3) 中的那些同名宏。

```
show-regexp('Price $12.', /[aeiou]/)           → Pr<<i>>ce $12.
show-regexp('Price $12.', /[\s]/)                → Price<< >>$12.
show-regexp('Price $12.', /[:digit:]/)           → Price $<<1>>2.
show-regexp('Price $12.', /[:space:]/)            → Price<< >>$12.
show-regexp('Price $12.', /[:punct:]aeiou/)       → Pr<<i>>ce $12.
```

在方括号内， c_1 - c_2 序列表达 c_1 和 c_2 之间的所有字符，并包含 c_2 。

```
a = 'see [Design Patterns-page 123]'
show-regexp(a, /[A-F]/)             → see [<<D>>esign Patterns-page 123]
show-regexp(a, /[A-Fa-f]/)          → s<<e>>e [Design Patterns-page 123]
show-regexp(a, /[0-9]/)             → see [Design Patterns-page <<1>>23]
show-regexp(a, /[0-9][0-9]/)        → see [Design Patterns-page <<12>>3]
```

如果想在字符类包含字面量字符]和-，它们必须出现在开始处。把^直接放在开始的方括号后面会对字符类求反：[^a-z] 匹配任何非小写字母的字符。

```
a = 'see [Design Patterns-page 123]'
show-regexp(a, /[]/)               → see [Design Patterns-page 123<<]>>
show-regexp(a, /[-]/)              → see [Design Patterns<<->> page 123]
show-regexp(a, /[^a-z]/)            → see<< >>[Design Patterns-page 123]
show-regexp(a, /[^a-z\s]/)          → see <<[>>Design Patterns-page 123]
```

一些字符类被使用得如此频繁以至于 Ruby 为它们提供了缩写形式。这些缩写形式在接下来的表 5.1 中列出——它们可以用在方括号内和模式体内。

```
show-regexp('It costs $12.', /\s/)      → It<< >>costs $12.
show-regexp('It costs $12.', /\d/)        → It costs $<<1>>2.
```

最后，出现在方括号外面的点号(.) 表示除回车换行符之外的任何字符（尽管在多行模式下它也匹配回车换行符）。

```
a = 'It costs $12.'
show-regexp(a, /c.s/)    → It <<cos>>ts $12.
show-regexp(a, /./)       → <<I>>t costs $12.
show-regexp(a, /\./)      → It costs $12<<. >>
```

表 5.1 字符类缩写

序 列	是[...]	含 义
\d	[0-9]	数字字符
\D	[^0-9]	除数字之外的任何字符
\s	[\t\r\n\f]	空格字符
\S	[^\t\r\n\f]	除空格之外的任何字符
\w	[A-Za-z0-9_]	组词字符
\W	[^A-Za-z0-9_]	除组词字符之外的任何字符
POSIX 字符类		
[:alnum:]		字母和数字
[:alpha:]		大写或小写字母
[:blank:]		空格和制表符
[:cntrl:]		控制字符（至少 0x00–0x1f, 0x7f）
[:digit:]		数字
[:graph:]		除了空格的可打印字符
[:lower:]		小写字符
[:print:]		任何可打印字符（包括空格）
[:punct:]		除了空格和字母数字的可打印字符
[:space:]		空格（等同于\s）
[:upper:]		大写字母
[:xdigit:]		16 进制数字（0–9, a–f, A–F）

重复

当指定 /\s*\|\s*/ 模式来分割歌曲列表的行时，也就是说我们想匹配被任意数目的空格围绕着的竖线。现在我们知道 \s 序列匹配单个空格，因此看起来星号 (*) 指的是“任意数目”。实际上，星号是多个修饰符中的一个，它允许对模式进行多次匹配。

如果 *r* 代表模式内先前的正则表达式，那么

- r** 匹配零个或多个 *r* 的出现
- r+* 匹配一个或多个 *r* 的出现
- r?* 匹配零个或一个 *r* 的出现
- r{m,n}* 匹配至少 "m" 次和最多 "n" 次 *r* 的出现
- r{m,}* 匹配至少 "m" 次 *r* 的出现
- r{m}* 只匹配 "m" 次 *r* 的出现

重复构造有高的优先级——它们只绑定到模式内先前的正则表达式。*/ab+/*a** 匹配一个 *a* 后面跟着一个或多个 *b*，而不是 *ab* 序列。必须小心使用 * 构造——*/a*/* 模式会匹配任何

字符串；每个字符串都会有零个或多个 *a*。

这种模式被称作贪心 (*greedy*) 模式。因为默认情况下它们会匹配尽可能多的字符串。你可以通过添加问号后缀让它们匹配最少的字符串来改变这个默认行为。

```
a = "The moon is made of cheese"
show_regex(a, /\w+/)           → <<The>> moon is made of cheese
show_regex(a, /\s.*\s/)         → The<< moon is made of >>cheese
show_regex(a, /\s.*?\s/)        → The<< moon >>is made of cheese
show_regex(a, /[aeiou]{2,99}/)   → The m<<oo>>n is made of cheese
show_regex(a, /mo?o/)          → The <<moo>>n is made of cheese
```

替换

竖线是特别字符，因此行分割模式必须使用反斜线去转义它，未转义的竖线 (|) 要么是匹配在它之前的正则表达式，要么是匹配在它之后的正则表达式。

```
a = "red ball blue sky"
show_regex(a, /die/)           → r<<e>>d ball blue sky
show_regex(a, /al|lu/)          → red b<<al>>l blue sky
show_regex(a, /red ball|angry sky/) → <<red ball>> blue sky
```

对粗心大意的人来说，这里有个陷阱。竖线的优先级非常低，最后这个例子匹配 *red ball* 或 *angry sky*，但不会匹配 *red ball sky* 或 *red angry sky*。为了匹配 *red ball sky* 或 *red angry sky*，需要使用编组 (grouping) 来重载默认的优先级。

编组

你可以使用括号在正则表达式中编组 (group) 词目。组内的所有东西被当作单个正则表达式对待。

```
show_regex('banana', /an*/)    → b<<an>>ana
show_regex('banana', /(an)*/)   → <<>>banana
show_regex('banana', /(an)+/)   → b<<anan>>a

a = 'red ball blue sky'
show_regex(a, /blue|red/)       → <<red>> ball blue sky
show_regex(a, /(blue|red) \w+/)  → <<red ball>> blue sky
show_regex(a, /(red|blue) \w+/)  → <<red ball>> blue sky
show_regex(a, /red|blue \w+/)    → <<red>> ball blue sky

show_regex(a, /red (ball|angry) sky/) → no match
a = 'the red angry sky'
show_regex(a, /red (ball|angry) sky/) → the <<red angry sky>>
```

括号也收集模式匹配的结果。Ruby 计算开始括号的数目，保存每个开始括号和相应的关闭括号之间部分匹配的结果。你可以在模式的剩余部分和 Ruby 程序中使用这种部分匹配。在模式内部，\1 序列指的是第一个组的匹配，\2 序列指的是第二个组的匹配。在

模式外面，特殊变量 \$1 和 \$2 等起到相同的作用。

```
"12:50am" =~ /(\d\d):(\d\d)(..)/      → 0
"Hour is #$1, minute #$2"              → "Hour is 12, minute 50"
"12:50am" =~ /((\d\d):(\d\d))(..)/    → 0
"Time is #$1"                         → "Time is 12:50"
"Hour is #$2, minute #$3"             → "Hour is 12, minute 50"
"AM/PM is #$4"                        → "AM/PM is am"
```

在匹配的剩余部分使用当前部分匹配的能力，能够让你在字符串中寻找各种形式的重复。

```
# match duplicated letter
show_regexp('He said "Hello"', /(\w)\1/)      → He said "He<<ll>>o"
# match duplicated substrings
show_regexp('Mississippi', /(\w+)\1/)           → M<<ississ>>ippi
```

也可以使用向后引用去匹配分界符。

```
show_regexp('He said "Hello"', /(["']).*?\1/) → He said
                                                <<"Hello">>
show_regexp("He said 'Hello'", /(["']).*?\1/) → He said
                                                <<'Hello'>>
```

5.4.2 基于模式的替换

Pattern-Based Substitution

在字符串中能够发现模式有时就已经足够好了。如果你的朋友给你出难题，让你找出顺序包含字母 *a*, *b*, *c*, *d* 和 *e* 的词。你可能会使用模式/a.*b.*c.*d.*e/去查找词表并找到 *abjectedness*, *absconded*, *ambuscade* 和 *carbacidometer* 这些词等等。这种事情肯定是有价值的。

当然，有时候得根据模式匹配去改变一些东西。让我们回到歌曲列表文件中，不管是谁，都会以小写方式输入歌曲演唱者的名字。以混合大小写方式在点唱机的屏幕上显示它们会更好看。怎么样才能把每个词的首字符改成大写呢？

`String#sub` 和 `String#gsub` 方法查找能够匹配第一个参数的那部分字符串，同时用第二个参数替换这它们。`String#sub` 执行一次替换，而 `String#gsub` 在匹配每次出现时都进行替换。两个方法都返回了对含有这些替换字符串的新的拷贝。`Mutator` 版本的 `String#sub!` 和 `String#gsub!` 会直接修改原先的字符串。

```
a = "the quick brown fox"
a.sub(/[aeiou]/, '*')   → "th* quick brown fox"
a.gsub(/[aeiou]/, '*') → "th* q**ck br*w n f*x"
a.sub(/\s\S+/, '')     → "the brown fox"
a.gsub(/\s\S+/, '')   → "the"
```

两个函数的第二个参数可以是 String 或 block。如果使用 block，匹配的子字符串会被传递给 block，同时 block 的结果值会被替换到原先的字符串中。

```
a = "the quick brown fox"  
a.gsub(/./) {|match| match.upcase }           → "The quick brown fox"  
a.gsub(/[aeiou]/) {|vowel| vowel.upcase }     → "thE qUIck brOwn fOx"
```

所以，对如何转换歌曲演唱者名字的这个问题，我们给出了答案。匹配词的首字符的模式是`\b\w`——它寻找后面跟着词字符（word character）的词边界。把这个模式和`gsub`结合起来，可以改变歌曲演唱者的名字了。

```
def mixed_case(name)
  name.gsub(/\b\w/) {|first| first.upcase }
end

mixed_case("fats waller")      → "Fats Waller"
mixed_case("louis armstrong") → "Louis Armstrong"
mixed_case("strength in numbers") → "Strength In Numbers"
```

替换中的反斜线序列

早些时候我们注意到序列`\1` 和`\2` 等在模式中可用，它们代表至今为止第 n 个已匹配的组。相同的序列也可以作为 `sub` 和 `gsub` 的第二个参数。

```
"fred:smith".sub(/(\w+):(\w+)/, '\2, \1')      → "smith, fred"  
"nercpcyitno".gsub(/(.) (.)/, '\2\1')          → "encryption"
```

其他的反斜线序列在替换字符串中起的作用：`\&`（最后的匹配），`\+`（最后匹配的组），`\``（匹配之前的字符串），`\``（匹配之后的字符串）和`\\"`（字面量反斜线）。

如果想在替换中包含字面量反斜线，我们会变得很困惑。显然可以写成：

```
str.gsub(/\V/, 'WW')
```

很清楚，代码正试图用两个反斜线替换 `str` 中的每个反斜线。程序员在替换文本中用了双倍的反斜线，因为知道它们在语法分析阶段会被转换成`\\"`。但是当替换发生时，正则表达式引擎对字符串又执行了一遍，并把`\\"`转换成`\`，所以最终结果是使用另外一个反斜线替换每个反斜线。需要把它写成 `gsub ('\\\\\\', '\\\\\\\\\\\\\\\\\\')`！

```
str = 'a\b\c'          → "a\b\c"  
str.gsub(/\b/, '\\\\\\\\') → "a\\\b\\\c"
```

但是，如果利用\&被替换为已匹配的字符串这个事实，也可以写成：

```
str = 'a\b\c'          → "a\b\c"  
str.gsub(/\\\/, '\\&\&') → "a\\b\\c"
```

如果使用 `gsub` 的 `block` 形式，用来替换的字符串会被和仅被分析一次（在语法分析阶段），这正是所希望看到的结果。

```
str = 'a\b\c'           → "a\b\c"
str.gsub(/\b/) { '\b' } → "a\\b\\c"
```

最后，下面的例子极好地表达了结合使用正则表达式和 `block`，请参见由 Wakou Aoyama 编写的 CGI 库模块的代码片断。这段代码接受包含 HTML 转义序列的字符串并把它转换成普通的 ASCII。为了支持日本用户，它在正则表达式上使用了 `n` 修饰符关闭了宽字符处理。它也说明了 Ruby 的 `case` 表达式，我们将从第 98 页开始讨论它。

```
def unescapeHTML(string)
  str = string.dup
  str.gsub!(/&(.*)?;/n) {
    match = $1.dup
    case match
    when /\Aamp\z/n      then '&'
    when /\Aquot\z/n     then '"'
    when /\Agt\z/n       then '>'
    when /\Alt\z/n       then '<'
    when /\A#(\d+)\z/n   then Integer($1).chr
    when /\A#[x([0-9a-f]+)\z/n then $1.hex.chr
    end
  }
  str
end

puts unescapeHTML("1<2 && 4>3")
puts unescapeHTML(""A" = &#65; = &#x41;")
```

输出结果：

```
1<2 && 4>3
"A" = A = A
```

5.4.3 面向对象的正则表达式

Object-Oriented Regular Expressions

必须承认，尽管所有这些古怪的变量用起来十分方便，但是它们不是面向对象的，同时它们很神秘。难道我们没有说 Ruby 的所有物体都是对象吗？哪里出错了？

真的没有错。在 Matz 设计 Ruby 时，他设计了一个完全面向对象的正则表达式处理系统。然后在它之上包装（wrap）所有这些\$变量，使它们看起来对 Perl 程序员很熟悉。对象和类还在这里，只不过就在表面下。让我们花点时间把它们挖掘出来吧。

实际上已经遇到过一个类：正则表达式字面量创建了 `Regexp` 类的实例（从第 600 页开始描述）。

```
re = /cat/
re.class → Regexp
```

`Regexp#match` 方法对字符串匹配正则表达式。如果失败了，它返回 `nil`。如果成功，则返回 `MatchData` 类（描述在第 537 页）的一个实例。`MatchData` 对象让你访问关于这次匹配的所有可用信息。所有这些好东西是通过`$`变量得到的，它们绑定在一个小巧方便的对象里。

```
re = /(\d+):(\d+)/ # match a time hh:mm
md = re.match("Time: 12:34am")
md.class → MatchData
md[0] # == $& → "12:34"
md[1] # == $1 → "12"
md[2] # == $2 → "34"
md.pre_match # == $` → "Time: "
md.post_match # == $' → "am"
```

匹配数据存储在它自己的对象里，这样可以同时保留两个或多个模式匹配的结果，这些事情使用那些`$`变量是做不出来的。在下面的例子中，对两个字符串匹配相同的 `Regexp` 对象，每次匹配返回了唯一的 `MatchData` 对象，通过检验两个子模式字段来验证它。

```
re = /(\d+):(\d+)/ # match a time hh:mm
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
md1[1, 2] → ["12", "34"]
md2[1, 2] → ["10", "30"]
```

那么这些`$`变量如何装备好的呢？好吧，每次模式匹配发生后，Ruby 把对结果（`nil` 或一个 `MatchData` 对象）的引用保存到线程局部变量中（可通过`$~`访问）。然后所有其他的正则表达式变量都是从这个对象派生出来的。尽管想象不出下面代码的实际使用，但是它说明了所有其他与 `MatchData` 相关的那些`$`变量真的是从`$~`中得到的。

```
re = /(\d+):(\d+)/
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
[ $1, $2 ] # last successful match → ["10", "30"]
$~ = md1
[ $1, $2 ] # previous successful match → ["12", "34"]
```

讲了所有这些之后，我们不得不心悦诚服。我们通常使用这些`$`变量，而无须担心 `MatchData` 对象。在日常使用中它们让我们的生活变得更方便。有时候我们不由自主地变成了实用主义者。

关于方法的更多细节

More about Methods

迄今为止，在本书中我们虽已经定义并使用了许多方法，但没有深入地思考。现在，是让我们考察其细节的时候了。

6.1 定义一个方法 Defining a Method

正如我们已经看到的，可以使用关键字 `def` 来定义一个方法。方法名必须以一个小写字母开头。¹表示查询的方法名通常以`?结尾`，例如 `instance_of?`。“危险的”或者会修改接收者对象（receiver）的方法，可以用`!结尾`。例如，`String` 提供了 `chop` 和 `chop!` 方法。第一个方法返回一个修改后的字符串；而第二个则就地修改对象。可以被赋值的方法（我们已经在第 31 页讨论过的一个特性）以一个等号（`=`）结尾。只有`?、!和=`这几个怪异的字符能够作为方法名的后缀。

现在，我们已经为新方法指定了一个名字，可能还需要声明某些参数（parameter，形参）。它们就是括号中列出的局部变量。（方法参数两边的括号是可选的；我们的惯例是，当方法有参数时则使用括号，否则即忽略它们。）

```
def my_new_method(arg1, arg2, arg3)      #3 arguments
  # Code for the method would go here
end
def my_other_new_method                  # No arguments
  # Code for the method would go here
end
```

Ruby 可以让你指定方法参数（argument，实参）的默认值——如果调用者在传入参数时没有明确指定所使用的值。为此你可以使用赋值操作符。

¹ 如果你使用大写字母，并不会立即得到一个错误，但是当 Ruby 看到你调用这个方法时，它首先猜测这是一个常量，而不是一个方法调用，且结果是 Ruby 可能错误地解析这个调用。

```

def cool_dude(arg1="Miles", arg2="Coltrane", arg3="Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end

cool_dude                                → "Miles, Coltrane, Roach."
cool_dude("Bart")                         → "Bart, Coltrane, Roach."
cool_dude("Bart", "Elwood")                → "Bart, Elwood, Roach."
cool_dude("Bart", "Elwood", "Linus")       → "Bart, Elwood, Linus."

```

- 方法体内是普通的 Ruby 表达式，你不能在方法内定义非单件（nonsingleton）类或模块。如果你在一个方法内定义另一个方法，内部的方法只有在外部方法执行时才得到定义。方法的返回值是执行的最后一个表达式的值，或者 return 表达式显式返回的值。

6.1.1 可变长度的参数列表

Variable-Length Argument Lists

但是如果你希望传入可变个数的参数（argument）、或者想用一个形参（parameter）接收多个参数，在“普通”的参数名前放置一个星号（*）即可。

```

def varargs(arg1, *rest)
  "Got #{arg1} and #{rest.join(', ')}"
end

varargs("one")                            → "Got one and "
varargs("one", "two")                     → "Got one and two"
varargs "one", "two", "three"             → "Got one and two, three"

```

在这个示例中，和往常一样第一个参数赋值给方法的第一个形参。不过，第二个形参的前缀为星号，因此所有剩余的参数被装入到一个新的 Array 中，然后赋值给第二个形参。

6.1.2 方法和 Block

Methods and Blocks

如我们在第 49 页的“Block 与迭代器”一节中讨论的，当调用一个方法时，可以用一个 block 与之相关联。通常，你可以使用 yield 从方法内部调用这个 block。

```

def take_block(p1)
  if block_given?
    yield(p1)
  else
    p1
  end
end

take_block("no block")                   → "no block"
take_block("no block") {|s| s.sub(/no /, '')} → "block"

```

不过，如果方法定义的最后一个参数前缀为`&`，那么所关联的 `block` 会被转换为一个 `Proc` 对象，然后赋值给这个参数。

```
class TaxCalculator
  def initialize(name, &block)
    @name, @block = name, block
  end
  def get_tax(amount)
    "#@name on #{amount} = #{@block.call(amount)}"
  end
end

tc = TaxCalculator.new("Sales tax") { |amt| amt * 0.075 }

tc.get_tax(100)      → "Sales tax on 100 = 7.5"
tc.get_tax(250)      → "Sales tax on 250 = 18.75"
```

6.2 调用方法

Calling a Method

你可以通过指定接收者、方法的名称、可选的参数及 `block`，来调用一个方法。

```
connection.download_MP3("jitterbug") { |p| show_progress(p) }
```

在这个示例中，`connection` 对象是接收者，`download_MP3` 是方法的名称，“`jitterbug`”是参数，花括号中的内容是相关联的 `block`。

对类方法或模块方法来说，接收者是类或模块的名字。²

```
File.size("testfile")      → 66
Math.sin(Math::PI/4)       → 0.707106781186548
```

如果你省略了接收者，其默认为 `self`，也就是当前的对象。

```
self.class      → Object
self.frozen?    → false
frozen?        → false
self.object_id → 978140
object_id      → 978140
```

`Ruby` 正是用这种默认机制实现私有方法调用的。我们无法调用某个接收者对象的私有方法，它们只在当前对象（`self`）中是可用的。

而在前面的示例中，我们调用 `self.class` 时必须要指定一个接收对象。这是因为 `class` 在 `Ruby` 中是一个关键字（它引入类的定义），因此单独使用它会产生语法错误。

² 译注：在 `Ruby` 中类和模块也是对象。

可选的参数跟随在方法名之后。如果没有二义性，在调用方法时，你可以省略参数列表两侧的括号。³不过，除非最简单的情况，我们并不推荐这样做——某些微妙的问题可能会让你犯错误。⁴我们的规则很简单：只要你有任何疑虑，就使用括号。

```
a = obj.hash      # Same as
a = obj.hash()    # this.

obj.some_method "Arg1", arg2, arg3      # Same thing as
obj.some_method("Arg1", arg2, arg3)     # with parentheses.
```

早先的 Ruby 版本允许你在方法名和左括号之间放置空格，使这个问题更为严重。这使得解析变得困难：括号是参数列表的起始呢，还是表达式的起始？在 Ruby 1.8 中，如果你在方法名和左括号之间置入空格，你会得到一个警告。

6.2.1 方法返回值

Method Return Values

每个被调用的方法都会返回一个值（尽管没有规定说你必须要使用这个值）。方法的值，是在方法执行中最后一个语句执行的结果。Ruby 有一个 `return` 语句，可以从当前执行的方法中退出。`return` 的返回值是其参数的值。如果不需要 `return` 就省略之，这是 Ruby 的一个惯用技法。

```
def meth_one
  "one"
end
meth_one      → "one"

def meth_two(arg)
  case
  when arg > 0
    "positive"
  when arg < 0
    "negative"
  else
    "zero"
  end
end

meth_two(23)      → "positive"
meth_two(0)       → "zero"
```

³ 其他的 Ruby 文档有时将这种没有括号的方法调用称为命令 (*command*)。

⁴ 特别地，当一个方法的参数列表中放置另一个方法调用（除非是最后一个参数）时，必须要使用括号。

```
def meth_three
  100.times do |num|
    square = num*num
    return num, square if square > 1000
  end
end
meth_three → [32, 1024]
```

最后一种情况演示了，如果你给 `return` 多个参数，方法会将它们以数组的形式返回。你可以使用并行赋值来收集返回值。

```
num, square = meth_three
num → 32
square → 1024
```

在方法调用中的数组展开

我们早先看到，在方法定义中，如果你在一个正规参数前加上星号，那么传入这个方法调用的多个参数将会被装入一个数组。当然，也有某些操作是相反的方式。

当你调用一个方法时，你可以分解一个数组，这样每个成员都被视为单独的参数。在数组参数（必须在所有普通参数的后面）前加一个星号可以完成这一点。

```
def five(a, b, c, d, e)
  "I was passed #{a} #{b} #{c} #{d} #{e}"
end

five(1, 2, 3, 4, 5) → "I was passed 1 2 3 4 5"
five(1, 2, 3, *['a', 'b']) → "I was passed 1 2 3 a b"
five(*(10..14).to_a) → "I was passed 10 11 12 13 14"
```

让 block 更加动态

我们已经看到如何为方法调用关联一个 block。

```
list_bones("aardvark") do |bone|
  # ...
end
```

一般来说，这已经足够好了——你可以将一个固定的 block 关联到方法，就如同在 `if` 或 `while` 语句之后编写的一大块代码。

但是，某些时候你希望能更灵活些。例如，我们希望教授一些算术技法。⁵学生们可能想要一个 n 次相加或相乘的表。如果学生需要一个 2 倍次的表，我们就输出 2, 4, 6, 8

⁵ 当然，Andy 和 Dave 可能先要学习数学技巧。Conrad Schneiker 提醒我们世界上有两类人：会数数和不会数数的。

等。（下面的代码并不检测可能的输入错误。）

```
print "(t)imes or (p)lus: "
times = gets
print "number: "
number = Integer(gets)
if times =~ /^t/
  puts((1..10).collect {|n| n*number }).join(", ")
else
  puts((1..10).collect {|n| n+number }).join(", ")
end
```

输出结果：

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

这可以工作，不过在每个 `if` 语句之后重复实质上等价的代码，颇为丑陋。如果我们
可以将完成计算的 `block` 抽取出来，代码就漂亮多了。

```
print "(t)imes or (p)lus: "
times = gets
print "number: "
number = Integer(gets)
if times =~ /^t/
  calc = lambda {|n| n*number }
else
  calc = lambda {|n| n+number }
end
puts((1..10).collect(&calc).join(", "))
```

输出结果：

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

如果方法的最后一个参数前有`&`符号，Ruby 将认为它是一个 `Proc` 对象。它会将其从
参数列表中删除，并将 `Proc` 对象转换为一个 `block`，然后关联到该方法。

收集散列参数

某些语言支持关键字参数——也就是你可以以任意顺序传入参数的名称与值，而非
按指定顺序传入参数。Ruby 1.8 并不支持关键字参数（这让我们成了说谎话的人，因为
我们在本书的上一版本中提到 Ruby 1.8 将会支持这一特性。也许要到 Ruby 2.0）。同时，
人们可以使用散列表来取得相同的效果。例如，可以考虑为我们的 `SongList` 加入更
强大的按名搜索功能。

```

class SongList
  def create_search(name, params)
    # ...
  end
end
list.create_search("short jazz songs",
  {
    'genre'          => "jazz",
    'duration_less_than' => 270
  })

```

第一个参数是搜索的名字，第二个是一个散列式，其中包括搜索的参数。使用散列表意味着我们可以模拟关键字：查询类型为“爵士乐”，且时长小于 4 分半的歌曲。不过，这种方式有些笨重，并且一组大括号容易被误写为一个和方法关联的 block。因此，Ruby 提供了一种快捷方式。只要在参数列表中，散列数组在正常的参数之后，并位于任何数组或 block 参数之前，你就可以只使用 *key => value* 对儿。所有这些对儿会被集合到一个散列数组中，并作为方法的一个参数传入。无须使用大括号。

```

list.create_search('short jazz songs',
  'genre'          => 'jazz',
  'duration_less_than' => 270)

```

最后，Ruby 的一种惯用技法是，你可以使用符号（symbol）而非字符串作为参数，符号清楚地表达了你所引用的是某个事物的名字。

```

list.create_search('short jazz songs',
  :genre          => :jazz,
  :duration_less_than => 270)

```

一个潜心编写的 Ruby 程序通常包括许多方法，每个都很短小，因此熟悉定义和使用 Ruby 方法的各种选择是非常值得的。

表达式

Expressions

到目前为止，我们对 Ruby 中表达式的使用相当随意。毕竟，`a=b+c` 是非常标准的表达式形式。因此即使不看本章，你也应该能写出完整的一堆 Ruby 代码来。

但那样就不怎么有趣了：◎

Ruby 和其他语言的一个不同之处就是任何东西都能返回一个值：几乎所有东西都是表达式。在实际中，这有什么意义呢？

明显的一个好处是能够实现链式语句。

```
a = b = c = 0          → 0
[ 3, 1, 7, 0 ].sort.reverse → [7, 3, 1, 0]
```

不太明显的好处是，C 和 Java 中的普通语句在 Ruby 中也是表达式。例如，`if` 和 `case` 语句都返回最后执行的表达式的值。

```
song_type = if song.mp3_type == MP3::Jazz
              if song.written < Date.new(1935, 1, 1)
                Song::TradJazz
              else
                Song::Jazz
              end
            else
              Song::Other
            end

rating = case votes_cast
         when 0...10  then Rating::SkipThisOne
         when 10...50 then Rating::CouldDoBetter
         else           Rating::Rave
         end
```

从第 96 页开始我们会详细讨论 `if` 和 `case`。

7.1 运算符表达式

Operator Expressions

Ruby 提供了基本的运算符集（如+、-、*、/ 等等），也提供了几个独特的运算符。第 339 页的表 22.4 给出了运算符及其优先级的完整列表。

实际上，Ruby 中的许多运算符是由方法调用来实现的。例如，当你执行 `a*b+c` 时，实际上你是请求 `a` 对象执行方法 `*`，传入的参数是 `b`。然后请求返回的结果对象执行 `+` 方法，传入的参数是 `c`。这等价于：

```
(a.*(b)).+(c)
```

因为任何东西都是对象，而且你可以重新定义实例方法，所以你可以重新定义任何不满足你需求的基本算术方法。

```
class Fixnum
  alias old_plus + # we can reference the original '+' as 'old_plus'

  # Redefine addition of Fixnums. This
  # is a BAD IDEA!
  def +(other)
    old_plus(other).succ
  end
end

1 + 2          → 4
a = 3
a += 4         → 8
a + a + a     → 26
```

更有用的是，你写的类可以像内建对象那样参与到运算符表达式中。比如，你可能想从歌曲中间剪辑一段，这可以用索引操作来实现。

```
class Song
  def [](from_time, to_time)
    result = Song.new(self.title + " [extract]",
                      self.artist,
                      to_time - from_time)
    result.set_start_time(from_time)
    result
  end
end
```

这段代码扩展了类 `Song`：提供了 `[]` 方法，该方法有两个参数（开始时间和结束时间）。它返回对应给定时间间隔的一个新的 `song` 对象。我们可以通过下面的代码来试听一首歌：

```
song[0, 15].play
```

7.2 表达式之杂项

Miscellaneous Expressions

除了支持明显的运算符表达式和方法调用，以及不太明显的语句表达式（例如 `if` 和 `case`）外，Ruby 的表达式还支持更多的东西。

7.2.1 命令展开

Command Expansion

如果你用反引号 (`)，或者以`%x` 为前缀的分界形式，括起一个字符串，默认情况下它会被当作底层操作系统的命令来执行。表达式的返回值就是该命令的标准输出。由于没有去除新行符，所以你获得的返回值结尾可能会有回车符或者换行符。

```
'date'           → "Thu Aug 26 22:36:31 CDT 2004\n"
`ls'.split[34]   → "book.out"
%x{echo "Hello there"} → "Hello there\n"
```

你也可以在命令字符串中使用表达式展开和所有普通的转义序列。

```
for i in 0..3
  status = `dbmanager status id=#{i}`
  #
end
```

命令的退出状态 (`exit status`) 保存在全局变量`$?`中。

重定义反引号

在前面的描述中，我们说反引号括起的字符串“默认”被当作命令来执行。实际上，字符串被传递给了名为 `Kernel.'`方法（单反引号）。如果你愿意，可以重载它。

```
alias old_backquote '
def '(cmd)
  result = old_backquote(cmd)
  if $? != 0
    fail "Command #{cmd} failed: #$?"
  end
  result
end
print 'date'
print 'data'
```

输出结果：

```
Thu Aug 26 22:36:31 CDT 2004
prog.rb:10: command not found: data
prog.rb:5:in ``': Command data failed: 32512 (RuntimeError)
from prog.rb:10
```

7.3 赋值

Assignment

到目前为止，本书给出的每个例子几乎都有赋值语句的影子。或许到该说说它的时候了。

赋值语句将左侧的变量或者属性（左值）设置为右侧的值（右值），然后返回该值作为赋值表达式的结果。这意味着你可以链接赋值语句，并可以在某些特殊的地方执行赋值操作。

```
a = b = 1 + 2 + 3
a      →   6
b      →   6
a = (b = 1 + 2) + 3
a      →   6
b      →   3
File.open(name = gets.chomp)
```

Ruby 的赋值语句有两种基本形式。第一种是将一个对象引用赋值给变量或者常量。这种形式的赋值在 Ruby 语言中是直接执行的（hardwired）。

```
instrument = "piano"
MIDDLE_A   = 440
```

第二种形式等号左边是对象属性或者元素的引用。

```
song.duration      = 234
instrument["ano"] = "ccolo"
```

这种形式的特殊之处在于，它是通过调用左值的方法来实现的，这意味着你可以重载它们。

我们已经看了如何定义一个可写的对象属性：只要简单地定义一个以等于号结尾的方法即可。这个方法以其右值作为它的参数。

```
class Song
  def duration=(new_duration)
    @duration = new_duration
  end
end
```

这种设置属性的方法不必和内部的实例变量相对应，并且具有赋值方法的属性也并非必须要有读取该属性的方法。

```
class Amplifier
  def volume=(new_volume)
    self.left_channel = self.right_channel = new_volume
  end
end
```

在老版本的 Ruby 中，赋值语句的返回值是设置该属性的方法的返回值。在 Ruby 1.8 中，赋值语句的值总是参数的值而方法的返回值将被丢掉。

```
class Test
  def val=(val)
    @val = val
    return 99
  end
end

t = Test.new
a = t.val = 2
a → 2
```

在老版本中，`a` 将被赋值语句设置为 99，而在 Ruby 1.8 中，它的值是 2。

7.3.1 并行赋值 Parallel Assignment

在编程语言课程的第一周（或者在比较散漫的“派对”学校的第二学期），你可能需要写代码来交换两个变量的值。

```
int a = 1;
int b = 2;
int temp;

temp = a;
a = b;
b = temp;
```

在 Ruby 中你可以用下面这样简洁的代码来实现。

```
a, b = b, a
```

Ruby 的赋值实际是以并行方式执行的，所以赋值语句右边的值不受赋值语句本身的影响。在左边的任意一个变量或属性被赋值之前，右边的值按它们出现的顺序被计算出来。下面这个人为设计的例子说明了这一点。第二行将表达式 `x`、`x+=1` 和 `x+=1` 的值分别赋值给变量 `a`、`b` 和 `c`。

```
x = 0 → 0
a, b, c = x, (x += 1), (x += 1) → [0, 1, 2]
```

当赋值语句有多于一个左值时，赋值表达式将返回由右值组成的数组。如果赋值语句的左值多于右值，那么多余的左值将被忽略。如果右值多于左值，那么额外的右值将被忽略。如果赋值语句仅有一个左值而有多个右值，那么右值将被转换成数组，然后赋值给左值。

在类中使用访问方法

Using Accessors within a Class

为什么第 90 页的例子中使用 `self.left_channel`? 其实, 可写属性有个隐藏的陷阱。通常, 类中的方法可以通过函数形式 (即带一个隐式 `self` 作为接受者) 调用同一个类的其他方法和它的父类的方法。然而这不适用于属性赋值函数: Ruby 看到赋值语句时, 会认为左边的名字是局部变量, 而不是一个为属性赋值的方法调用。

```
class BrokenAmplifier
  attr_accessor :left_channel, :right_channel
  def volume=(vol)
    left_channel = self.right_channel = vol
  end
end

ba = BrokenAmplifier.new
ba.left_channel = ba.right_channel = 99
ba.volume = 5
ba.left_channel → 99
ba.right_channel → 5
```

在上面的赋值语句中, 我们忘了给 `left_channel` 加 “`self.`” 前缀, 所以 Ruby 存储新值给 `volume=` 方法的局部变量, 因而根本没有更新对象属性。这会成为难以追踪的缺陷。

使用 Ruby 的并行赋值操作, 你可以叠起和展开数组。如果最后一个左值有一个 ‘*’ 前缀, 那么所有多余的右值将被集合在一起, 并作为一个数组赋给左值。同样地, 如果最后一个右值是一个数组, 你可以在它的前面加一个 ‘*’ , 它将被适当地展开成其元素的值 (如果右边只有一个右值, 那么这就没有必要了——数组会自动展开的)。

<code>a = [1, 2, 3, 4]</code>	
<code>b, c = a</code>	→ <code>b == 1, c == 2</code>
<code>b, *c = a</code>	→ <code>b == 1, c == [2, 3, 4]</code>
<code>b, c = 99, a</code>	→ <code>b == 99, c == [1, 2, 3, 4]</code>
<code>b, *c = 99, a</code>	→ <code>b == 99, c == [[1, 2, 3, 4]]</code>
<code>b, c = 99, *a</code>	→ <code>b == 99, c == 1</code>
<code>b, *c = 99, *a</code>	→ <code>b == 99, c == [1, 2, 3, 4]</code>

嵌套赋值

并行赋值还有一个值得一提的特性: 赋值语句的左边可以含有一个由括号括起来的变量列表。Ruby 视这些变量为嵌套赋值语句。在处理更高层级的赋值语句前, Ruby 会

提取出对应的右值，并赋值给括起来的变量。

```
b, (c, d), e = 1,2,3,4      →  b == 1, c == 2, d == nil,    e == 3
b, (c, d), e = [1,2,3,4]    →  b == 1, c == 2, d == nil,    e == 3
b, (c, d), e = 1,[2,3],4    →  b == 1, c == 2, d == 3,      e == 4
b, (c, d), e = 1,[2,3,4],5  →  b == 1, c == 2, d == 3,      e == 5
b, (c,*d), e = 1,[2,3,4],5 →  b == 1, c == 2, d == [3, 4], e == 5
```

7.3.2 赋值语句的其他形式

Other Forms of Assignment

和许多其他语言一样，Ruby 有一个句法的快捷方式：`a = a + 2` 可以写成 `a+=2`。

内部处理时，会将第二种形式先转换成第一种形式。这意味着在你自己类中作为方法定义的操作符，和你预期的效果是一样的。

```
class Bowdlerize
  def initialize(string)
    @value = string.gsub(/[aeiou]/, '*')
  end
  def +(other)
    Bowdlerize.new(self.to_s + other.to_s)
  end
  def to_s
    @value
  end
end

a = Bowdlerize.new("damn ")      →      d*m*
a += "shame"                   →      d*m* sh*m*
```

Ruby 不支持 C 或者 Java 中的自加（`++`）和自减（`--`）运算符，如果想用，使用 `+=` 和 `-=` 替代之。

7.4 条件执行

Conditional Execution

Ruby 对条件代码的执行有几种不同的机制：大多数的形式比较常见，并且对许多形式作了一些简化。不过在我们学习它们之前，我们还是需要花点时间来看看布尔表达式。

7.4.1 布尔表达式

Boolean Expressions

Ruby 对“真值”（truth）的定义很简单：任何不是 `nil` 或者常量 `false` 的值都为真。你会发现 Ruby 库程序常常利用这一事实。例如 `IO#gets` 方法返回文件的下一行，如果遇到文件尾则返回 `nil`，利用它可以写出如下循环。

```
while line = gets
  #process line
end
```

然而，C, C++ 和 Perl 程序员可能会落入陷阱：数字 0 不被解释为假值，长度为 0 的字符串也不是假值。这可是一个难以克服的顽固的习惯。

Defined?、与、或和非

Ruby 支持所有的标准布尔操作符，并引入了一个新操作符 `defined?`。

仅当 `and` 和 `&&` 的两个操作数都为真时结果才为真。并且仅当第一个操作数为真时才求解第二个操作数的值（这也称为短路求解，*shortcircuit evaluation*）。这两种形式的唯一区别在于优先级不同（`and` 低于 `&&`）。

同样，如果有一个操作数为真，那么 `or` 和 `||` 的结果为真。且仅当第一个操作数为假时才求解第二个操作数。和 `and` 一样，`or` 和 `||` 的唯一区别是它们的优先级。

需要注意的是 `and` 和 `or` 有相同的优先级，而 `&&` 的优先级高于 `||`。

`not` 和 `!` 返回它们的操作数的相反值（如果操作数为真，则返回假；如果操作数为假，则返回真）。你可能已经想到了，是的，`not` 和 `!` 的唯一区别是优先级不同。

第 339 页的表 22.4 总结了所有这些优先级规则。

如果参数（可以是任意表达式）未被定义，`defined?` 操作符返回 `nil`；否则返回对参数的一个描述。如果参数是 `yield`，而且有一个 `block` 和当前上下文相关联，那么 `defined?` 返回字符串“`yield`”。

<code>defined? 1</code>	→ "expression"
<code>defined? dummy</code>	→ nil
<code>defined? printf</code>	→ "method"
<code>defined? String</code>	→ "constant"
<code>defined? \$_</code>	→ "global-variable"
<code>defined? Math::PI</code>	→ "constant"
<code>defined? a = 1</code>	→ "assignment"
<code>defined? 42.abs</code>	→ "method"

除了布尔表达式，Ruby 对象还支持如下比较方法：`==`, `==>`, `<=>`, `=~`, `eql?` 和 `equal?`（参见下一页的表 7.1）。除了 `<=>`，其他方法都是在类 `Object` 中定义的，但是经常被子类重载以提供适当的语义。例如，类 `Array` 重定义了 `==`：当两个数组对象有相同的元素个数，且对应的元素也都相等时，才认为它们相等。

表 7.1 常用的比较操作符

操作符	含 义
<code>==</code>	测试值相等与否
<code>==></code>	用来比较 case 语句的目标和每个 when 从句的项
<code><=></code>	通用比较操作符。根据接受者小于、等于或者大于其参数，返回 1, 0 或 +1
<code><, <=, >=, ></code>	小于、小于等于、大于等于、大于比较操作符
<code>=~</code>	正则表达式模式匹配操作符
<code>eq1?</code>	如果接受者和参数有相同的类型和相等的值，则返回真。 <code>1==1.0</code> 返回真，但 <code>eq1?(1,0)</code> 为假
<code>equal?</code>	如果接受者和参数有相同的对象 ID，则返回真

`==` 和 `=~` 都有相反的形式：`!=` 和 `!~`。不过，在 Ruby 读取程序的时候，会对它们进行转换：`a != b` 等价于 `!(a == b)`；`a !~ b` 等价于 `!(a =~ b)`。这意味着如果你写的类重载了 `==` 或者 `=~`，那么也会自动得到 `!=` 和 `!~`。不过另一方面，这也意味着你无法独立于 `==` 和 `=~` 而分别定义 `!=` 和 `!~`。

你还可以用 Ruby 的 `range` 作为布尔表达式。像 `exp1..exp2` 这样的 `range`，在 `exp1` 变为真之前，它的值为假；在 `exp2` 变为真之前，`range` 被求解为真。一旦 `exp2` 变为真，`range` 将重置，准备再次重新计算。我们在第 100 页展示了几个这样的例子。

1.8. Ruby 1.8 之前的版本中，可以用裸正则表达式（bare regular expression）作为布尔表达式。现在这种方式已经过时了。不过你仍然可以用 `~` 操作符将 `$_` 和一个模式进行匹配。

7.4.2 逻辑表达式的值

The Value of Logical Expressions

在上面，我们说“如果两个操作数都为真，则 `and` 语句的值为真。”但实际上比这更微妙。操作符 `and`, `or`, `&&` 和 `||` 实际上返回首个决定条件真伪的参数的值。听着很复杂。那到底是什么意思？

比方说表达式“`val1 and val2`”，如果 `val1` 为假或者 `nil`，我们知道这个表达式不可能为真。在这个例子中，`val1` 的值决定了表达式的值，所以返回它的值。如果 `val1` 为其他值，那么表达式的值依赖于 `val2`，所以返回它的值。

```

nil and true      →    nil
false and true   →    false
99 and false     →    false
99 and nil       →    nil
99 and "cat"     →    "cat"

```

注意：不管怎样，表达式的最终值是正确的。

or 值的计算与此类似（除了当 val1 的值为真时，才能提前知道 or 表达式的值）。

```

false or nil      →    nil
nil   or false    →    false
99   or false    →    99

```

Ruby 的一个惯用技法利用了这一特性。

```

words[key] ||= []
words[key] << word

```

第一行等价于 `words[key] = words[key] || []`。如果散列表 `words` 中 `key` 对应的项没有值 (`nil`)，那么 `||` 的值是第二个操作数：一个新的空数组。这样，这两行代码将会把一个数组赋值给没有值的散列表元素，对已经有值的元素则原封不动。有时，你也会看到将它们写到一行的情况：

```
(words[key] ||= []) << word
```

7.4.3 If 和 Unless 表达式

If and Unless Expressions

Ruby 的 if 表达式和其他语言的“if”语句非常相似。

```

if song.artist == "Gillespie" then
  handle = "Dizzy"
elsif song.artist == "Parker" then
  handle = "Bird"
else
  handle = "unknown"
end

```

如果将 if 语句分布到多行上，那么可以不用 then 关键字。

```

if song.artist == "Gillespie"
  handle = "Dizzy"
elsif song.artist == "Parker"
  handle = "Bird"
else
  handle = "unknown"
end

```

不过，如果你想让你的代码更紧凑些，你可以使用 then 关键字来区分布尔表达式和它后面的语句。

```
if song.artist == "Gillespie" then handle = "Dizzy"
elsif song.artist == "Parker" then handle = "Bird"
else handle = "unknown"
end
```

1.8 使用冒号（:）来替代 `then` 可以使得代码更简洁。

```
if song.artist == "Gillespie": handle = "Dizzy"
elsif song.artist == "Parker": handle = "Bird"
else handle = "unknown"
end
```

你可以用零个或者多个 `elsif` 从句和一个可选的 `else` 从句。

正如我们前面所说，`if` 是表达式而不是语句——它返回一个值。你不必非要使用 `if` 表达式的值，但它迟早能派上用场。

```
handle = if song.artist == "Gillespie" then
          "Dizzy"
        elsif song.artist == "Parker" then
          "Bird"
        else
          "unknown"
        end
```

Ruby 还有一个否定形式的 `if` 语句。

```
unless song.duration > 180
  cost = 0.25
else
  cost = 0.35
end
```

最后，给 C 的爱好者一个惊喜，Ruby 也支持 C 风格的条件表达式。

```
cost = song.duration > 180 ? 0.35 : 0.25
```

条件表达式返回的是冒号前面表达式的值还是冒号后面表达式的值，依赖于问号前面布尔表达式值的真伪。在这个例子中，如果歌曲的长度大于 3 分钟，则表达式返回 0.35，否则返回 0.25。不管结果是什么，它将被赋给 `cost` 变量。

If 和 Unless 修饰符

Ruby 和 Perl 都有一个灵活的特性：语句修饰符允许你将条件语句附加到常规语句的尾部。

```
mon, day, year = $1, $2, $3 if date =~ /(\d\d)-(\d\d)-(\d\d)/
puts "a = #{a}" if debug
print total unless total.zero?
```

对于 `if` 修饰符，仅当条件为真时才计算前面的表达式的值。`unless` 与此相反。

```
File.foreach("/etc/fstab") do |line|
  next if line =~ /^#/          # Skip comments
  parse(line) unless line =~ /^$/ # Don't parse empty lines
end
```

因为 `if` 本身也是一个表达式，下面的用法会让人感到晦涩难懂：

```
if artist == "John Coltrane"
  artist = "'Trane"
end unless use_nicknames == "no"
```

这种用法将令人抓狂。

7.5 Case 表达式

Case Expressions

Ruby 的 `case` 表达式非常强大：它相当于多路的 `if`。而它有两种形式，使得它更加强大。

第一种形式接近于一组连续的 `if` 语句：它让你列出一组条件，并执行第一个为真的条件表达式所对应的语句。例如闰年必然能被 400 整除，或者能被 4 整除但不能被 100 整除。

```
leap = case
  when year % 400 == 0: true
  when year % 100 == 0: false
  else year % 4 == 0
end
```

`case` 语句的第二种形式可能更常见。在 `case` 语句的顶部指定一个目标，而每个 `when` 从句列出一个或者多个比较条件。

```
case input_line
when "debug"
  dump_debug_info
  dump_symbols
when /p\s+(\w+)/
  dump_variable($1)
when "quit", "exit"
  exit
else
  print "Illegal command: #{input_line}"
end
```

和 `if` 一样，`case` 返回执行的最后一个表达式的值；而且如果表达式和条件在同一行上的话，你可以用 `then` 关键字来加以区分。

```
kind = case year
  when 1850..1889 then "Blues"
  when 1890..1909 then "Ragtime"
  when 1910..1929 then "New Orleans Jazz"
  when 1930..1939 then "Swing"
  when 1940..1950 then "Bebop"
  else
    "Jazz"
end
```

1.8. 和 if 语句一样，也可以使用冒号（:）来替代 then 关键字。

```
kind = case year
  when 1850..1889: "Blues"
  when 1890..1909: "Ragtime"
  when 1910..1929: "New Orleans Jazz"
  when 1930..1939: "Swing"
  when 1940..1950: "Bebop"
  else
    "Jazz"
end
```

case 通过比较目标（case 关键字后面的表达式）和 when 关键字后面的比较表达式来运作。这个测试通过使用 *comparison === target* 来完成。只要一个类为 *==*（内建的类都有）提供了有意义的语义，那么该类的对象就可以在 case 表达式中使用。

例如，正则表达式将 *==* 定义为一个简单的模式匹配。

```
case line
when /title=(.*)/
  puts "Title is #$1"
when /track=(.*)/
  puts "Track is #$1"
when /artist=(.*)/
  puts "Artist is #$1"
end
```

Ruby 的所有类都是类 Class 的实例，它定义了 *==* 以测试参数是否为该类或者其父类的一个实例。所以（放弃了多态的好处，并把重构的“福音”带到你的耳侧），你可以测试对象到底属于哪个类。

```
case shape
when Square, Rectangle
  # ...
when Circle
  # ...
when Triangle
  # ...
else
  # ...
end
```

7.6 循环

Loops

不要告诉任何人，Ruby 内建的循环结构相当原始。

只要条件为真，`while` 循环就会执行循环体。比如，下面这个常用法读取输入直到输入结束。

```
while line = gets
  # ...
end
```

`until` 循环与此相反，它执行循环体直到循环条件变为真。

```
until play_list.duration > 60
  play_list.add(song_list.pop)
end
```

与 `if` 和 `unless` 一样，你也可以用这两种循环做语句的修饰符。

```
a = 1
a *= 2 while a < 100
a -= 10 until a < 100
a → 98
```

在第 95 页的布尔表达式一节中，我们说过 `range` 可以作为某种后空翻：当某个事件发生时变为真，并在第二个事件发生之前一直保持为真。这常被用在循环中。在下面的例子中，我们读一个含有前十个序数（“first”，“second”等等）的文本文件，但只输出匹配“third”和“fifth”之间的行。

```
file = File.open("ordinal")
while line = file.gets
  puts(line) if line =~ /third/ .. line =~ /fifth/
end
```

输出结果：

```
third
fourth
fifth
```

你可能会发现熟悉 Perl 的人，他们的写法和前面的代码稍有不同。

```
file = File.open("ordinal")
while file.gets
  print if ~/third/ .. ~/fifth/
end
```

输出结果：

```
third
fourth
fifth
```

这段代码背后有点小戏法：`gets` 将读取的最后一行赋值给全局变量 `$_`，`~` 操作符对 `$_` 执行正则表达式匹配，而不带参数的 `print` 将输出 `$_`。在 Ruby 社区中，这种类型的代码已经过时了。

用在布尔表达式中的 `range` 的起点和终点本身也可以是表达式。每次求解总体布尔表达式时就会求解起点和终点表达式的值。例如，下面的代码利用了变量 `$.` 包含当前输入行号这一事实，来显示 1 到 3 行以及位于 /eig/ 和 /nin/ 之间的行。

```
File.foreach("ordinal") do |line|
  if (($. == 1) || line =~ /eig/) .. (($. == 3) || line =~ /nin/)
    print line
  end
end
```

输出结果：

```
first
second
third
eighth
ninth
```

当使用 `while` 和 `until` 做语句修饰符时，有一点需要注意：如果被修饰的语句是一个 `begin/end` 块，那么不管布尔表达式的值是什么，块内的代码至少会执行一次。

```
print "Hello\n" while false
begin
  print "Goodbye\n"
end while false
```

输出结果：

```
Goodbye
```

7.6.1 迭代器

Iterators

如果你读了前面小节的开头部分，当时可能会很失望，因为我们在那儿提到：“Ruby 内建的循环结构很原始”。不过，不要失望，亲爱的读者，因为我们有好消息：Ruby 不需要内建任何复杂的循环，因为迭代器实现了所有有趣的东西。

例如，Ruby 没有“for”循环——至少没有类似于 C, C++ 和 Java 中的 for 循环。但是 Ruby 使用各种内建的类中定义的方法来提供类似且更健壮的功能。

让我们看几个例子。

```
3.times do
  print "Ho! "
end
```

输出结果：

```
Ho! Ho! Ho!
```

这种代码易于避免长度错误和差异：这种循环会执行 3 次。除了 `times` 函数，通过调用 `downto` 和 `upto` 函数，整数还可以在指定的 `range` 上循环，而且所有数字都可以使用 `step` 来循环。例如，传统的从 0 到 9 的“for”循环（类似 `i=0; i<10; i++`）可以写成：

```
0.upto(9) do |x|
  print x, " "
end
```

输出结果：

```
0 1 2 3 4 5 6 7 8 9
```

从 0 到 12，步长为 3 的循环可以写成：

```
0.step(12, 3) {|x| print x, " "}
```

输出结果：

```
0 3 6 9 12
```

类似地，使用 `each` 方法使得遍历数组和其他容器变得十分简单。

```
[1, 1, 2, 3, 5].each {|val| print val, " "}
```

输出结果：

```
1 1 2 3 5
```

并且如果一个类支持 `each` 方法，那么也会自动支持 `Enumerable` 模块（第 454 页有其文档，第 120 页有其概述）中的方法。例如，`File` 类提供了 `each` 方法，它依次返回文件中的行。使用 `Enumerable` 中的 `grep` 方法，我们可以只迭代那些满足某些条件的行：

```
File.open("ordinal").grep(/\d$/) do |line|
  puts line
end
```

输出结果：

```
second
third
```

最后也可能是最简单的：Ruby 提供了一个内建的称为 `loop` 的迭代器。

```
loop do
  # block ...
end
```

loop 循环一直执行给定的块（或者直到你跳出循环，后面会讲到如何做）。

7.6.2 For ... In

前面我们说到 Ruby 内建的循环原语只有 while 和 until。那么 for 呢？是的，for 基本上是一个语法块，当我们写：

```
for song in songlist
  song.play
end
```

Ruby 把它转换成：

```
songlist.each do |song|
  song.play
end
```

for 循环和 each 形式的唯一区别是循环体中局部变量的作用域。第 105 页对此进行了讨论。

你可以使用 for 去迭代任意支持 each 方法的类，例如 Array 或者 Range。

```
for i in ['fee', 'fi', 'fo', 'fum']
  print i, " "
end
for i in 1..3
  print i, " "
end
for i in File.open("ordinal").find_all{|line| line =~ /d$/}
  print i.chomp, " "
end
```

输出结果：

```
fee fi fo fum 1 2 3 second third
```

只要你的类支持 each 方法，你就可以使用 for 循环去遍历它的对象。

```
class Periods
  def each
    yield "Classical"
    yield "Jazz"
    yield "Rock"
  end
end
```

```
periods = Periods.new
for genre in periods
  print genre, " "
end
```

输出结果：

```
Classical Jazz Rock
```

7.6.3 Break, Redo 和 Next

Break, Redo, and Next

循环控制结构 `break`, `redo` 和 `next` 可以让你改变循环或者迭代的正常流程。

`break` 终止最接近的封闭循环体，然后继续执行 `block` 后面的语句。`redo` 从循环头重新执行循环，但不重计算循环条件表达式或者获得迭代中的下一个元素。`next` 跳到本次循环的末尾，并开始下一次迭代。

```
while line = gets
  next if line =~ /^#\s*/ # skip comments
  break if line =~ /^END/ # stop at end
    # substitute stuff in backticks and try again
  redo if line.gsub!(//'.*?'//) { eval($1) }
  # process line ...
end
```

这些关键字还可以和任意基于迭代的循环机制一起使用。

```
i=0
loop do
  i += 1
  next if i < 3
  print i
  break if i > 4
end
```

输出结果：

```
345
```

1.8 在 Ruby 1.8 中，可以传递一个值给 `break` 和 `next`。在传统的循环中，这可能只对 `break` 有意义，此时 `break` 将设置循环的返回值。（传递给 `next` 的任意值会被丢掉。）如果传统循环根本没有执行 `break`，那么它的值是 `nil`。

```
result = while line = gets
  break(line) if line =~ /answer/
end
process_answer(result) if result
```

如果你想了解 `break` 和 `next` 是如何与 `block` 和 `proc` 协同工作的本质细节，可以参考

第 358 页的描述。如果你想从嵌套的 block 或者循环中退出，请参考第 362 页和第 519 页描述的 Kernel.catch。

7.6.4 Retry

`redo` 语句使得一个循环重新执行当前的迭代。但是有时你需要从头重新执行一个循环。`retry` 语句就是做这件事的，它重新执行任意类型的迭代式循环。

```
for i in 1..100
  print "Now at #{i}. Restart? "
  retry if gets =~ /^y/i
end
```

交互式地运行这段代码，你会看到：

```
Now at 1. Restart? n
Now at 2. Restart? y
Now at 1. Restart? n
.
.
```

`retry` 在重新执行之前会重新计算传递给迭代的所有参数。下面是一个 DIY 的 `until` 循环的例子。

```
def do_until(cond)
  break if cond
  yield
  retry
end

i = 0
do_until(i > 10) do
  print i, " "
  i += 1
end
```

输出结果：

```
0 1 2 3 4 5 6 7 8 9 10
```

7.7 变量作用域、循环和 Blocks

Variable Scope, Loops, and Blocks

`while`, `until` 和 `for` 循环内建到了 Ruby 语言中，但没有引入新的作用域：前面已存在的局部变量可以在循环中使用，而循环中新创建的局部变量也可以在循环后使用。

被迭代器使用的 `block`（比如 `loop` 和 `each`）与此略有不同。通常，在这些 `block` 中创建的局部变量无法在 `block` 外访问。

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

输出结果：

```
prog.rb:4: undefined local variable or method 'x' for
main:Object (NameError)
```

然而，如果执行 `block` 的时候，一个局部变量已经存在且与 `block` 中的变量同名，那么 `block` 将使用此已有的局部变量。因而，它的值在块后面仍然可以使用。如下面例子所示，此即适用于 `block` 中的普通变量也适用于 `block` 的参数。

```
x = nil
y = nil
[ 1, 2, 3 ].each do |x|
  y = x + 1
end
[ x, y ] → [3, 4]
```

注意在外部作用域中变量不必有值：Ruby 解释器只需要看到它即可。

```
if false
  a = 1
end
3.times { |i| a = i }

a → 2
```

变量作用域和 `block` 的问题在 Ruby 社区中引起了广泛的讨论。当前的模式有一定的问题（特别是当 `block` 中的变量和外部变量意外同名时），但是没有人提出更好的且被社区广泛接受的新方案。Matz 希望在 Ruby 2.0 中对此进行改进，但目前，我们只能提几条建议以减少局部变量和 `block` 变量相互干扰带来的问题。

- 使你的方法和 `block` 尽量简短。变量越少，则它们相互干扰的机会就越少。而且也易于阅读，以检查是否有名字冲突。
- 为局部变量和 `block` 参数使用不同的命名方式。例如，你可能不想用“`i`”作为局部变量的名字，但作为 `block` 参数它还是可以接受的。

实际上，这个问题不像你想象的那样频繁发生。

异常，捕获和抛出

Exceptions, Catch, and Throw

到目前为止，我们都是在 Pleasantville（欢乐谷）中开发代码，在这个奇妙的地方不会发生任何错误。每个程序库的调用都是成功的，用户从来没有输入不准确的数据，这儿的资源也是丰富而廉价的。好的，该有些变化了，欢迎回到现实的世界来！

在现实世界中，错误会发生。好的程序（和程序员）可以预见它们的发生，然后优雅地处理这些错误。但这并非总像听起来那么简单。通常检测到错误出现的那部分代码，缺少有关如何处理它的上下文信息。比如，试图去打开一个并不存在的文件，在一些情况下是可行的，但在别的情况下却是致命的错误。文件处理模块要如何做呢？

传统的做法是使用返回码。`open` 方法在失败时会返回一些特定值。然后这个值会沿着调用例程的层次往回传播，直到有函数想要处理它。

这种做法的问题是，管理所有这些错误码是一件痛苦的事情。如果函数首先调用 `open`，然后调用 `read`，最后调用 `close` 方法，而且每个方法都有可能返回错误标识。那么当函数将返回码返回给调用者时，该如何区分这些错误码呢？

异常在很大程度上解决了这个问题。异常允许把错误信息打包到一个对象中，然后该异常对象被自动传播回调用栈（calling stack），直到运行系统找到明确声明知道如何处理这类异常的代码为止。

8.1 异常类

The Exception Class

含有异常信息的数据包（package）是 `Exception` 类、或其子类的一个对象。Ruby

预定义了一个简洁的异常层次结构，见下页的图 8.1。我们在后面会看到，这个层次结构使得处理异常变得相当简单。

当需要引发 (`raise`) 异常时，可以使用某个内建的 `Exception` 类，或者创建自己的异常类。如果创建自己的异常类，可能你希望它从 `StandardError` 类或其子类派生。否则，你的异常在默认情况下不会被捕获。

每个 `Exception` 都关联有一个消息字符串和栈回溯信息（`backtrace`）。如果定义自己的异常，可以添加额外的信息。

8.2 处理异常

Handling Exceptions

我们的点唱机用 TCP 套接字从互联网下载歌曲。它的基本代码很简单（假设文件名和套接字都已经创建好）。

```
op_file = File.open(opfile_name, "w")
while data = socket.read(512)
  op_file.write(data)
end
```

如果下载过程中得到一个致命错误，会发生什么呢？我们肯定不想在歌曲列表中存储一首不完整的歌曲。“I Did it My *刺刺拉拉的声音*。”

让我们添加一些处理异常的代码，看看它是如何帮助处理异常的。在一个 `begin/end` 块中，使用一个或多个 `rescue` 语句告诉 Ruby 希望处理的异常类型。在这个特定的例子中，我们感兴趣的是捕获 `SystemCallError` 异常（同时暗含着任何 `SystemCallError` 子类的异常），所以它就是出现在 `rescue` 行的异常类型。在这个错误处理 `block` 中，我们报告了这个错误，关闭和删除了输出文件，同时重新引发异常。

```
op_file = File.open(opfile_name, "w")
begin
  # 这段代码引发的异常会被
  # 下面的 rescue 语句捕获
  while data = socket.read(512)
    op_file.write(data)
  end

  rescue SystemCallError
    $stderr.print "IO failed: " + $!
    op_file.close
    File.delete(opfile_name)
    raise
  end
```

当异常被引发时，Ruby 将相关 `Exception` 对象的引用放在全局变量 `$!` 中，这与任何随后的异常处理不相干。

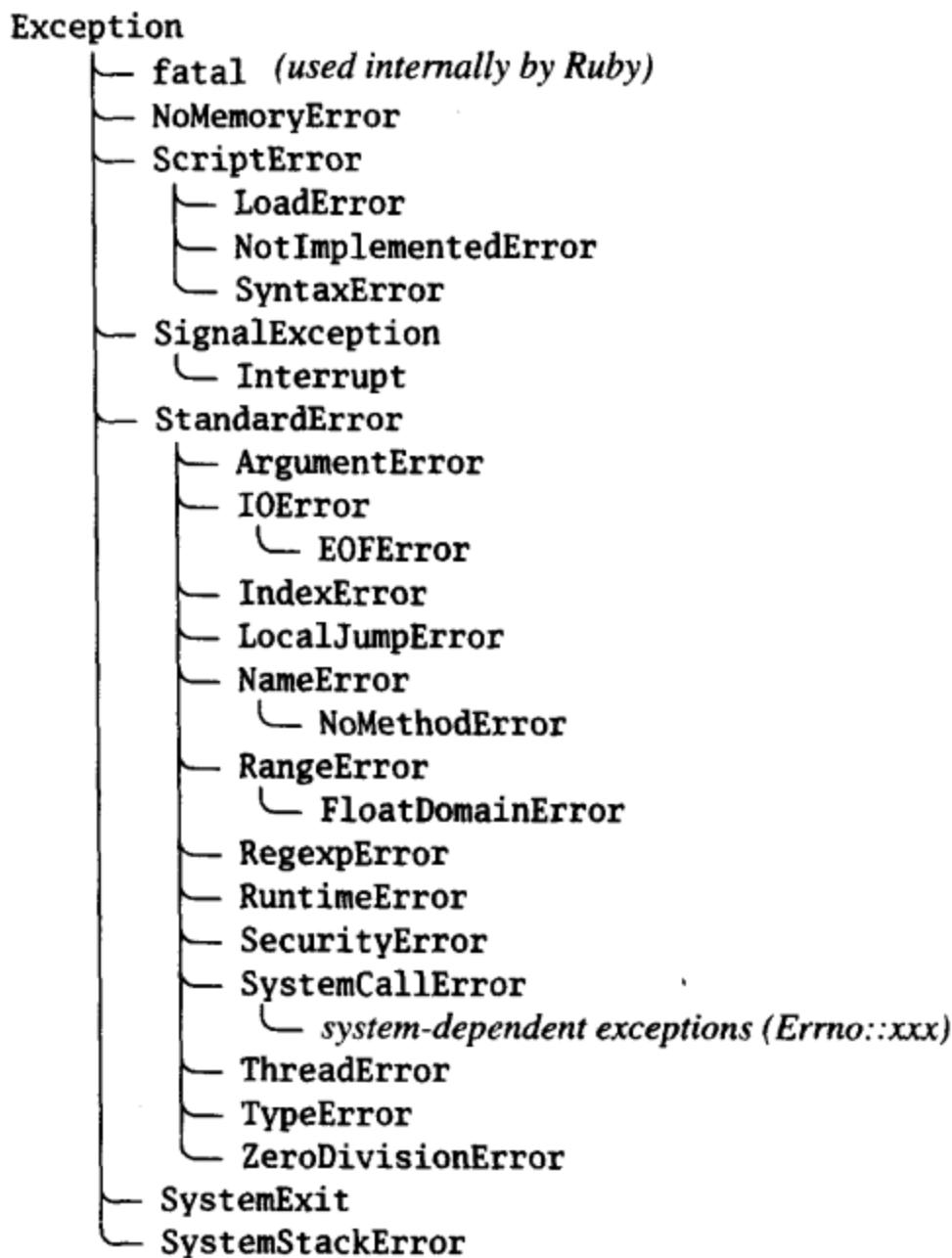


图 8.1 Ruby 异常层次结构

（这个感叹号大概反映出了我们的惊讶，我们的代码竟然会导致错误！），在前面的例子中，我们用\$!变量去格式化错误信息。

关闭和删除文件后，我们可以不带任何参数来调用 `raise`，它会重新引发\$!中的异常。这是一个有用的技术，它允许我们先编写代码过滤掉一些异常，再把不能处理的异常传递到更高的层次。这几乎就像实现了一个错误处理的继承层次结构。

在 `begin` 块中可以有多个 `rescue` 子句（clause），每个 `rescue` 子句可以指示捕获多个异常。在 `rescue` 子句的结束处，你可以提供一个 Ruby 的局部变量名来接收匹配

的异常。许多人发现这比到处使用`$!`有更好的可读性。

```
begin
  eval string
rescue SyntaxError, NameError => boom
  print "String doesn't compile: " + boom
rescue StandardError => bang
  print "Error running script: " + bang
end
```

Ruby 如何决定执行哪个 `rescue` 子句呢？这个处理非常类似于对 `case` 语句的处理。`Ruby` 用引发的异常依次比较 `begin` 块中每个 `rescue` 子句的每个参数。如果引发的异常匹配了一个参数，`Ruby` 就执行 `rescue` 的程序体，同时停止比较。匹配是用 `parameter===$!` 完成的。对于大多数异常来说，如果 `rescue` 子句给出的类型与当前引发的异常的类型相同，或者它是引发异常的超类（superclass），这意味着匹配是成功的。¹如果编写一个不带参数表的 `rescue` 子句，它的默认参数是 `StandardError`。

如果没有任何 `rescue` 子句与之匹配，或者异常在 `begin/end` 块外面被引发，`Ruby` 就沿着调用栈向上查找，在调用者上寻找异常的处理器，接着在调用者的调用者上寻找，依此类推。

尽管 `rescue` 子句的参数通常是 `Exception` 类的名称，实际上它们可以是任何返回 `Exception` 类的表达式（包括方法调用）。

8.2.1 系统错误

System Errors

当对操作系统的调用返回错误码时，会引发系统错误。在 POSIX 系统上，这些错误名称有诸如 `EAGAIN` 和 `EPERM` 等（在 Unix 机器上，键入 `man error`，你会得到这些错误的列表）。

`Ruby` 得到这些错误，把每个错误包装（wrap）到特定的对象中。每个错误都是 `SystemCallError` 的子类，定义在 `Errno` 模块中。这意味着，你会发现类名如 `Errno::EAGAIN`, `Errno::EIO` 和 `Errno::EPERM` 等的异常。如果想得到低层的系统错误码，则每个 `Errno` 异常对象有一个 `Errno`（有点令人困惑）的类常量（class constant），它包含相应的系统错误码。

¹ 可以这样进行比较的原因是：异常是类，而类进而是某种 `Module`. `==` 方法是为模块定义的，如果操作数的类与接收者相同或者是接收者的祖先，这个方法返回 `true`。

```

Errno::EAGAIN::Errno      → 35
Errno::EPERM::Errno       → 1
Errno::EIO::Errno         → 5
Errno::EWOULDBLOCK::Errno → 5

```

注意到 `EWOULDBLOCK` 和 `EAGAIN` 有相同的错误码。这是我电脑上的操作系统的一个特性——两个常量映射到相同的错误码。为了处理这种情况，Ruby 作出了安排，让 `Errno::EAGAIN` 和 `Errno::EWOULDBLOCK` 在 `rescue` 子句中被等同地对待。如果你要求 `rescue` 其中一个错误，那么另一个也会被 `rescue`。通过重定义 `SystemCallError#==` 可以做到这点，因此，如果要比较 `SystemCallError` 的两个子类，是比较它们的错误码而不是它们在层次结构中的位置。

8.2.2 善后

Tidying Up

有时候你需要保证一些处理在 `block` 结束时能够被执行，而不管是否有异常引发。比如，也许在 `block` 的入口处（`entry`）打开了一个文件，需要确保当 `block` 退出时它会被关闭。

`ensure` 子句就是做这个的。`ensure` 跟在最后的 `rescue` 子句后面，它包含一段当 `block` 退出时总是要被执行的代码。不管 `block` 是否正常退出，是否引发并 `rescue` 异常，或者是否被未捕获的异常终止——这个 `ensure` 块总会得到运行。

```

f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close unless f.nil?
end

```

尽管不是那么有用，`else` 子句是一个类似 `ensure` 子句的构造。如果存在的话，它会出现在 `rescue` 子句之后和任何一个 `ensure` 子句之前。`else` 子句的程序体，只有当主代码体没有引发任何异常时才会被执行。

```

f = File.open("testfile")
begin
  # .. 处理
rescue
  # .. 处理错误
else
  puts "Congratulations-- no errors!"
ensure
  f.close unless f.nil?
end

```

8.2.3 再次执行

Play It Again

有时候也许可以纠正异常的原因。在这些例子中，你可以在 `rescue` 子句中使用 `retry` 语句去重复执行整个 `begin/end` 区块。显然这很可能会导致无限循环，所以使用这个特性应当倍加小心（同时把一根手指轻轻放在键盘的中断键上，随时准备着）。

下面的例子在出现异常时会重新执行，它来自 Minero Aoki 的 `net/smtp.rb` 库。

```
@essmtp = true

begin
  # 首先尝试扩展登录。如果因为服务器不支持而失败，
  # 则使用正常登录

  if @essmtp then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end

  rescue ProtocolError
    if @essmtp then
      @essmtp = false
      retry
    else
      raise
    end
  end
end
```

这段代码首先使用 `EHLO` 命令试图连接 SMTP 服务器，而这个命令并没有被广泛支持。如果连接尝试失败了，则设置 `@essmtp` 变量为 `false`，同时重试连接。如果第二次连接也失败了，则引发异常给它的调用者。

8.3 引发异常

Raising Exceptions

到目前为止，我们一直都处于守势，处理那些被别人引发的异常。该是轮到我们进攻的时候了（有些人说本书这些温和的作者总是咄咄逼人，但那是另外一本书）。

可以使用 `Kernel.raise` 方法在代码中引发异常（或者它有点判决意味的同义词，`Kernel.fail`）。

```
raise
raise "bad mp3 encoding"
raise InterfaceException, "Keyboard failure", caller
```

第一种形式只是简单地重新引发当前异常（如果没有当前异常的话，引发 RuntimeError）。这种形式用于首先截获异常再将其继续传递的异常处理方法中。

第二种形式创建新的 RuntimeError 异常，把它的消息设置为指定的字符串。然后异常随着调用栈向上引发。

第三种形式使用第一个参数创建异常，然后把相关联的消息设置给第二个参数，同时把栈信息（trace）设置给第三个参数。通常，第一个参数是 Exception 层次结构中某个类的名称，或者是某个异常类的对象实例的引用。² 通常使用 Kernel.caller 方法产生栈信息。

下面是使用 raise 的一些典型例子。

```
raise
raise "Missing name" if name.nil?
if i >= names.size
  raise IndexError, "#{i} >= size (#{names.size})"
end
raise ArgumentError, "Name too big", caller
```

最后这个例子从栈回溯信息删除当前函数，这在程序库模块中十分有用。可以更进一步：下面的代码通过只将调用栈的子集传递给新异常，从而达到从栈回溯信息中删除两个函数的目的。

```
raise ArgumentError, "Name too big", caller[1...-1]
```

8.3.1 添加信息到异常

Adding Information to Exceptions

你可以定义自己的异常，保存任何需要从错误发生地传递出去的信息。例如，取决于外部环境，某种类型的网络错误可能是暂时的。如果这种错误发生了，而环境是适宜的，则可以在异常中设置一个标志，告诉异常处理程序重试这个操作可能是值得的。

```
class RetryException < RuntimeError
  attr :ok_to_retry
  def initialize(ok_to_retry)
    @ok_to_retry = ok_to_retry
  end
end
```

² 从技术层面上讲，这个参数可以是任何对象，只要它能响应消息 exception，且这个消息返回一个能够满足 object.kind_of?(Exception) 为真的对象。

在下面的代码里面，发生了一个暂时的错误。

```
def read_data(socket)
  data = socket.read(512)
  if data.nil?
    raise RetryException.new(true), "transient read error"
  end
  # .. 正常处理
end
```

在上一级的调用栈处理了异常。

```
begin
  stuff = read_data(socket)
  # .. process stuff
rescue RetryException => detail
  retry if detail.ok_to_retry
  raise
end
```

8.4 捕获和抛出

Catch and Throw

尽管 `raise` 和 `rescue` 的异常机制对程序出错时终止执行已经够用了，但是如果在正常处理过程期间能够从一些深度嵌套的结构中跳转出来，则是很棒的。`catch` 和 `throw` 应运而生，可以方便地做到这点。

```
catch (:done) do
  while line = gets
    throw :done unless fields = line.split(/\t/)
    songlist.add(Song.new(*fields))
  end
  songlist.play
end
```

`catch` 定义了以给定名称（可能是符号或字符串）为标签的 `block`。这个 `block` 会正常执行直到遇到 `throw` 为止。

当 Ruby 碰到 `throw`，它迅速回溯（*zip back*）调用栈，用匹配的符号寻找 `catch` 代码块。当发现它之后，Ruby 将栈清退（*unwind*）到这个位置并终止该 `block`。所以在前面的例子中，如果输入没有包含正确格式化的行，`throw` 会跳到相应 `catch` 代码块的结束处，不仅终止了 `while` 循环，而且跳过了歌曲列表的播放。如果调用 `throw` 时指定了可选的第二个参数，这个值会作为 `catch` 的值返回。

在下面的例子中，如果在响应任意提示符时键入！，使用 `throw` 终止与用户的交互。

```
def prompt_and_get(prompt)
  print prompt
  res = readline.chomp
  throw :quit_requested if res == "!"
  res
end

catch :quit_requested do
  name = prompt_and_get("Name: ")
  age  = prompt_and_get("Age: ")
  sex  = prompt_and_get("Sex: ")
  #
  # 处理信息
end
```

这个例子说明了 `throw` 没必要出现在 `catch` 的静态作用域中。



模块

Modules

模块是一种将方法、类与常量组织在一起的方式。模块给你提供了两个主要的好处：

1. 模块提供了命名空间（namespace）来防止命名冲突。
2. 模块实现了 mixin 功能。

9.1 命名空间

Namespaces

当你开始编写越来越大的 Ruby 程序时，你自然会发现自己编写了许多可重用的代码——将相关的例程（routine）组成一个库通常是合适的。你会希望将这些代码分解到不同的文件，使其内容可以被其他不同的 Ruby 程序共享。

通常代码会被组织为类，你可能会让一个类（或一组相关的类）对应一个文件。

不过，有时你想要把那些无法自然构成类的部分集合到一起。

一种初步的方法是将所有内容放到一个文件中，然后简单地在任何需要它的程序中加载（load）它。这是 C 语言工作的方式。不过，这种方式有一个问题。假设你要编写一组三角函数 `sin`、`cos` 等等。你将它们全部塞到一个文件 `trig.rb` 中，为后世享用。同时，Sally 想要模拟善良和邪恶，并且她编写了一组对自己有用的例程，包括 `be_good` 和 `sin1`，并将它们放到 `moral.rb` 中。Joe 想要编写一个程序来找出针尖上有多少跳舞的天使²，需要在他的程序中加载 `trig.rb` 和 `moral.rb`。但是两者都定义了名为 `sin` 的方法。糟了。

¹ 译注：罪恶。

² 译注：中世纪的神学讨论课题。

答案是使用模块机制。模块定义了一个 *namespace*（命名空间），它是一个沙箱（sandbox），你的方法和常量可以在其中任意发挥，而无须担心被其他方法或常量干扰。三角函数可以放到一个模块中。

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
    #
  end
  def Trig.cos(x)
    #
  end
end
```

而“品行”好坏的方法可以放到另一个模块中。

```
module Moral
  VERY_BAD = 0
  BAD = 1
  def Moral.sin(badness)
    #
  end
end
```

模块常量的命名和类常量一样，都以大写字母开头。方法定义同样看起来很相似：这些模块方法就类似于类方法的定义。

如果第三方的程序想要使用这些模块，它可以简单地加载这两个文件（使用 Ruby 的 `require` 语句，我们将在第 123 页讨论）并引用它们的完整名称（qualified name）。

```
require 'trig'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

同类方法一样，你可以用模块名和句点来调用模块方法，使用模块名和两个冒号来引用常量。

9.2 Mixin

模块有另外一个妙用。它提供了一种称为 *mixin* 的功能，以雷霆之势，极大地消除了对多重继承的需要。

在上一节的示例中，我们定义了模块方法，它们的名字都以模块名为前缀。如果这让你想到类方法，你接下来的想法可能是“如果我在模块内定义实例方法会怎样？”

好问题。模块并没有实例，因为模块并不是类。不过，你可以在类的定义中 *include*（包含）一个模块。当包含发生时，模块所有的实例方法瞬间在类中也可使用了。它们被混入（*mix in*）了。实际上，所混入的模块其实际行为就像是一个超类。

```
module Debug
  def who_am_i?
    "#<#{self.class.name} (\#\{self.object_id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
class EightTrack
  include Debug
  # ...
end
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")

ph.who_am_i?      →      "Phonograph (#945760): West End Blues"
et.who_am_i?      →      "EightTrack (#945760): Surrealistic Pillow"
```

通过包含 `Debug` 模块，`Phonograph` 和 `EightTrack` 都得以访问 `who_am_i?` 这个实例方法。

在继续前行之前，我们将探讨关于 `include` 语句的几点问题。首先，`include` 与文件无关。`C` 程序员使用叫做`#include` 的预处理器指令，在编译期将一个文件的内容插入到另一个文件中。`Ruby` 语句只是简单地产生一个指向指定模块的引用。如果模块位于另一个文件中，在使用 `include` 之前，你必须使用 `require`（或者不那么常用的旁系，`load`）将文件加载进来。第二点，`Ruby` 的 `include` 并非简单地将模块的实例方法拷贝到类中。相反，它建立一个由类到所包含模块的引用。如果多个类包含这个模块，它们都指向相同的内容。即使当程序正在运行时，如果你改变模块中一个方法的定义，所有包含这个模块的类都会表现出新的行为。³

`Mixin` 为你向类中添加功能，提供了一种控制精巧的方式。不过，它们的真正力量，当 `mixin` 中的代码和使用它的类中的代码开始交互时，它们会一齐迸发出来。让我们以标准的 `Ruby mixin`——`Comparable` 为例。你可以使用 `Comparable mixin` 向类中添加比较操作符（`<`, `<=`, `==`, `>=` 和 `>`）以及 `between?` 方法。为了使其能够工作，`Comparable` 假定任何使用它的类都定义了`<=>`操作符。这样，作为类的一个编写者，你定义一个方法 `<=>`，再包含 `Comparable`，然后就可以免费得到 6 个比较函数。让我们用 `Song` 类来尝试

³ 当然，我们这里所指的只是方法。比如，实例变量总是属于每个对象所有的。

一下，令它们基于时长来进行比较。我们所要做的就是包含 Comparable 模块并实现比较操作符`<=>`。

```
class Song
  include Comparable
  def initialize(name, artist, duration)
    @name      = name
    @artist    = artist
    @duration = duration
  end
  def <=>(other)
    self.duration <=> other.duration
  end
end
```

我们可以用几首测试歌曲来检查一下结果。

```
song1 = Song.new("My Way", "Sinatra", 225)
song2 = Song.new("Bicyclops", "Fleck", 260)

song1 <=> song2  → -1
song1 < song2   → true
song1 == song1  → true
song1 > song2   → false
```

9.3 迭代器与可枚举模块

Iterators and the Enumerable Module

你可能已经注意到 Ruby 收集（collection）类支持大量针对收集的各种操作：遍历、排序等等。你可能会想，“哎呀，如果我自己的类也能支持这些出色的特性，那就太好了！”（如果你的确这么想，大概是时候停止收看 20 世纪 60 年代电视节目的重播了。）

当然，你的类可以支持所有这些出色的特性，感谢 mixin 和 Enumerable 模块的魔力。你所要做的就是编写一个称为 `each` 的迭代器，由它依次返回收集中的元素。包含入 Enumerable，然后你的类瞬间支持诸如 `map`、`include?` 和 `find_all` 等操作。如果在你收集的对象中使用`<=>`方法实现了有意义的排序语义，你还会得到诸如 `min`、`max` 和 `sort` 等方法。

9.4 组合模块

Composing Modules

回到第 52 页，我们讨论了 Enumerable 的 `inject` 方法。Enumerable 是另一个标准的

`mixin`, 它基于宿主类 (`host class`) 中的 `each` 实现了许多方法。因此, 我们可以在任何包括了 `Enumerable` 模块并定义了 `each` 方法的类中使用 `inject`。许多内建的类都是如此。

```
[ 1, 2, 3, 4, 5 ].inject {|v,n| v+n } → 15
('a'...'m').inject {|v,n| v+n } → "abcdefghijklm"
```

我们还可以定义自己的类以包含 `Enumerable`, 继而得到 `inject` 的支持。

```
class VowelFinder
  include Enumerable

  def initialize(string)
    @string = string
  end

  def each
    @string.scan(/aeiou/) do |vowel|
      yield vowel
    end
  end
end

vf = VowelFinder.new("the quick brown fox jumped")

vf.inject {|v,n| v+n } → "euioooue"
```

注意, 我们使用了和前面示例中调用 `inject` 的相同模式——使用它来求和。当作用于数字时, 它返回算术和, 当作用于字符串时, 返回串联的字符串。我们也可以使用一个模块来封装这个功能。

```
module Summable
  def sum
    inject {|v,n| v+n }
  end
end

class Array
  include Summable
end

class Range
  include Summable
end

class VowelFinder
  include Summable
end

[ 1, 2, 3, 4, 5 ].sum → 15
('a'...'m').sum → "abcdefghijklm"

vf = VowelFinder.new("the quick brown fox jumped")
vf.sum → "euioooue"
```

9.4.1 Mixin 中的实例变量

Instance Variables in Mixins

从 C++ 转向 Ruby 的人经常问我们，“ mixin 中的实例变量会如何呢？在 C++ 中，我必须兜好几个圈子才能控制如何在多重继承中共享变量。Ruby 是如何处理的呢？”

我们告诉他们，好吧，对初学者来说，这根本不是什么问题。回忆一下 Ruby 中实例变量是如何工作的：当前缀为 @ 的变量第一次出现时，即在当前对象（也就是 `self`）中创建实例变量。

对 mixin 来说，这意味着你要混入客户类中的模块，可能会在客户对象中创建实例变量，并可能使用 `attr_reader` 或类似方法，定义这些实例变量的访问方法。例如，下面实例中的 `Observable` 模块，会向包含它的类中添加实例变量 `@observer_list`。

```
module Observable
  def observers
    @observer_list ||= []
  end

  def add_observer(obj)
    observers << obj
  end

  def notify_observers
    observers.each { |o| o.update }
  end
end
```

不过，这种行为也给我们带来了风险。一个 mixin 中的实例变量可能会和其宿主类或其他 mixin 中的实例变量相冲突。下面的实例演示使用了 `Observer` 模块的一个类，但是不幸的是，它也有一个名为 `@observer_list` 的实例变量。在运行时，程序可能会产生某些难以诊断的错误行为。

```
class TelescopeScheduler
  # other classes can register to get notifications
  # when the schedule changes
  include Observable

  def initialize
    @observer_list = [] # folks with telescope time
  end

  def add_viewer(viewer)
    @observer_list << viewer
  end

  # ...
end
```

多数时候， mixin 模块并不带有它们自己的实例数据——它们只是使用访问方法从客户对象中取得数据。但是，如果你要创建的 mixin 不得不持有它们自己的状态。确保这

个实例变量具有唯一的名字，可以对系统中其他的 `mixin` 区别开来（也许使用模块名作为变量名的一部分）。或者，模块可以使用模块一级的散列表，以当前对象的 ID 作为索引，来保存特定于实例的数据，而不必使用 Ruby 的实例变量。

```
module Test
  State = {}
  def state=(value)
    State[object_id] = value
  end
  def state
    State[object_id]
  end
end

class Client
  include Test
end

c1 = Client.new
c2 = Client.new
c1.state = 'cat'
c2.state = 'dog'

c1.state      → "cat"
c2.state      → "dog"
```

9.4.2 解析有歧义的方法名

Resolving Ambiguous Method Names

关于 `mixin`，人们经常问到的另一个问题是，方法查找是如何处理的？特别地，如果类、父类，以及类所包含的 `mixin` 中，都定义有相同名字的方法时，会发生什么？

答案是，Ruby 首先会从对象的直属类（immediate class）中查找，然后是类所包含的 `mixin`，之后是超类以及超类的 `mixin`。如果一个类有多个混入的模块，最后一个包含的模块将会被第一个搜索。

9.5 包含其他文件

Including Other Files

因为 Ruby 可以使我们轻松地编写良好的、模块化的代码，你经常会发现自己编写了许多含有大量自包含功能的小文件——比如 `x` 的接口、完成 `y` 的算法，等等。典型地，你会将这些文件组织为类或库。

在产生这些文件之后，你希望在新的程序中结合使用它们。Ruby 有两个语句来完成这一点。每次当 `load` 方法执行时，都会将指定的 Ruby 源文件包含进来。

```
load 'filename.rb'
```

更常见的是使用 `require` 方法来加载指定的文件，且只加载一次。⁴

```
require 'filename'
```

被加载文件中的局部变量不会蔓延到加载它们所在的范围中。例如，这里有一个名为 `included.rb` 的文件。

```
a=1
def b
  2
end
```

下面是当我们把它包含到另一个文件中时，将会发生什么。

```
a = "cat"
b = "dog"
require 'included'
a      →      "cat"
b      →      "dog"
b()    →      2
```

`require` 有额外的功能：它可以加载共享的二进制库。两者都可以接受相对或绝对路径。如果指定了一个相对路径（或者只是一个简单的名字），它们将会在当前加载路径中（`load path`——`$:`，将在第 183 页讨论）的每个目录下搜索这个文件。

使用 `load` 或 `require` 所加载的文件，当然也可以包含其他文件，而这些文件又可包含别的文件，依此类推。可能不那么明显的是，`require` 是一个可执行的语句——它可能在一个 `if` 语句内、或者可能包含一个刚刚拼合的字符串。搜索路径也可以在运行时更改。只需将你希望的目录加入到`$:`数组中。

因为 `load` 会无条件地包含源文件，你可以使用它来重新加载一个在程序开始执行后可能更改的源文件。下面是一个人为设计的示例。

⁴ 这并不是严格为真的。Ruby 在数组`$"`中保存了被 `require` 所加载的文件列表。不过，这个列表只包括了调用 `require` 时所指定的文件名。欺骗 Ruby 让它多次加载同一个文件，是有可能的。

```
require '/usr/lib/ruby/1.9/English.rb'
require '/usr/lib/ruby/1.9/rdoc/../English.rb'
```

```
$" → ["/usr/lib/ruby/1.9/English.rb", "/usr/lib/ruby/1.9/rdoc/../English.rb"]
```

在这种情况下，两个 `require` 语句都最终加载了同一个文件，不过使用了不同的路径。某些人认为这是个 bug，而这个行为在以后的版本中可能会改变。

```
5.times do |i|
  File.open("temp.rb","w") do |f|
    f.puts "module Temp"
    f.puts "  def Temp.var"
    f.puts "    #{i}"
    f.puts "  end"
    f.puts "end"
  end
  load "temp.rb"
  puts Temp.var
end
```

输出结果：

```
0
1
2
3
4
```

对于这个功能，可以考虑一个不太实际的例子：Web 应用重新加载正在运行的模块。这让它能够动态地更新自己；它不需要重新启动来集成软件的新版本。这是使用例如 Ruby 等动态语言的众多好处之一。



基本输入和输出

Basic Input and Output

Ruby 提供了两套乍看之下完全不同的 I/O 例程（routine）。第一套接口很简单——至今我们已经多次使用了。

```
print "Enter your name:\n"
name = gets
```

Kernel 模块实现了一整套 I/O 相关的方法：gets, open, print, printf, putc, puts, readline, readlines 和 test，它们使得 Ruby 编程简便。这些方法通常对标准输入和标准输出进行操作，因而很适合用它们来编写过滤器。你可以从第 516 页找到其相关文档。

第二种方式是使用 `IO` 对象，这种方式可以对 `IO` 进行更多的控制。

10.1 什么是 `IO` 对象

What Is an `IO` Object

Ruby 定义了一个 `IO` 基类来处理输入和输出。类 `File` 和 `BasicSocket` 都是该基类的子类，虽然它们提供了更具体的行为，但是基本原则是相同的。`IO` 对象是 Ruby 程序和某些外部资源¹之间的一个双向通道。当然，一个 `IO` 对象不止是这些显而易见的东西，但最终你所要做的只是向之写入或者从中读取。

本章我们会重点介绍 `IO` 类和它最常用的子类 `File`。如果想获得更多关于使用套节字类进行网络操作的信息，请参见第 763 页。

¹ 对那些想知道实现细节的人来说，这意味着一个 `IO` 对象可以管理操作系统的多个文件描述符。例如，如果你打开一对管道，那么一个 `IO` 对象可以既包括读管道也可以包括写管道。

10.2 文件打开和关闭

Opening and Closing Files

正如你所想，你可以使用 `File.new` 创建一个新的文件对象。

```
file = File.new ("testfile", "r")
#... 处理该文件
file.close
```

根据打开模式，你可以创建一个用来读、写或者兼有两者的文件对象（这里我们使用“`r`”模式来打开 `testfile` 以从中读取数据。我们也可以用“`w`”模式来表示写文件，而“`r+`”表示读写。第 504 页有完整的模式列表）。当创建一个文件时，你还可以指定文件的许可权限，详细信息请参见第 470 页中 `File.new` 的描述。打开文件后，我们可以写入或者读取所需的数据。最后，作为一个负责任的软件开发人员，我们需要关闭该文件，以确保所有缓存的数据被写入文件，并释放所有相关的资源。

但 Ruby 能使这些操作更简单。方法 `File.open` 也可以打开文件，通常情况下，它和 `File.new` 行为相似，但是当和 `block` 一起调用时就有区别了。与返回一个新的 `File` 对象不同，`open` 会以刚打开的文件作为参数调用相关联的 `block`，当 `block` 退出时，文件会被自动关闭。

```
File.open("testfile", "r") do |file|
  # ... 文件处理
end
```

第二种方式有额外的优势。在较前面的例子中，处理文件的过程中如果发生异常，可能 `file.close` 就不会被调用。一旦 `file` 变量出了其作用域，垃圾收集器最终会把它关闭，但可能一时半会儿不会发生。而与此同时，资源一直被占用。

以 `block` 调用 `File.open` 的形式没有这个问题。如果 `block` 内引发了异常，那么在异常传递给调用者之前文件会被关闭。看似 `open` 方法实现了类似下面的代码：

```
class File
  def File.open(*args)
    result = f = File.new(*args)
    if block_given?
      begin
        result = yield f
      ensure
        f.close
      end
    end
    return result
  end
end
```

10.3 文件读写

Reading and Writing Files

我们用于“简单”I/O的所有方法都适用于文件对象。`gets`从标准输入读取一行（或者从执行脚本的命令行上指定的任意文件），而`file.gets`从文件对象`file`中读取一行。

例如，我们可以创建一个程序`copy.rb`。

```
while line = gets
  puts line
end
```

如果不带参数运行此程序，将从终端读取行然后再复制回终端显示。注意一旦按下面键，每行就会回显出来。（在这个例子以及以后的例子中，我们用粗体表示用户输入。）

```
% ruby copy.rb
These are lines
These are lines
that I am typing
that I am typing
^D
```

我们也可以从命令行传入一个或多个文件名，这种情况下`gets`将依次从文件中读入。

```
% ruby copy.rb testfile
This is line one
This is line two
This is line three
And so on...
```

最后我们可以显式地打开文件，并从中读取数据。

```
File.open("testfile") do |file|
  while line = file.gets
    puts line
  end
end
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

和`gets`一样，I/O对象还有一组额外的访问方法，它们使得Ruby编程更容易。

10.3.1 读取迭代器

Iterators for Reading

你既可以使用普通的循环从 `IO` 流中读取数据，也可以使用各种各样的 Ruby 迭代器来读取数据。`IO#each_byte` 将以从 `IO` 对象中获得的下一个 8 位字节为参数，调用相关联的 `block`（在下面的例子中，对象类型是 `File`）。

```
File.open("testfile") do |file|
  file.each_byte{|ch| puts ch; print ".."}
end
```

输出结果：

```
T.h.i.s. .i.s. .l.i.n.e. .o.n.e.
.T.h.i.s. .i.s. .l.i.n.e. .t.w.o.
.T.h.i.s. .i.s. .l.i.n.e. .t.h.r.e.e.
.A.n.d. .s.o. .o.n.....
.
```

`IO#each_line` 以文件的每一行为参数调用相关联的 `block`。在下面的例子中，我们将用 `String#dump` 来显示换行符，这样一来你就知道我们没有骗人了。

```
File.open("testfile") do |file|
  file.each_line{|line| puts "Got #{line.dump}"}
end
```

输出结果：

```
Got "This is line one\n"
Got "This is line two\n"
Got "This is line three\n"
Got "And so on...\n"
```

你可以将任意字符序列传递给 `each_line` 作为行分隔符，然后它会根据该字符序列相应地分隔输入字符串，并返回分割后的每一行。这也是为什么在前面的例子中你能看到 `\n` 字符的原因。在下面的例子中，我们将使用字符 `e` 作为行分隔符。

```
File.open("testfile") do |file|
  file.each_line("e"){|line| puts "Got #{line.dump}"}
end
```

输出结果：

```
Got "This is line"
Got " one"
Got "\nThis is line"
Got " two\nThis is line"
Got " thre"
Got "e"
Got "\nAnd so on...\n"
```

如果你将迭代器思想和 `block` 自动关闭文件的特性结合在一起，你就获得了 `IO.foreach`。这个方法以 I/O 数据源的名字作为参数，以读模式打开它，并以文件中的每一行为参数调用关联的迭代器，最后会自动关闭该文件。

```
IO.foreach("testfile") { |line| puts line }
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

另外，如果你喜欢，你还可以将整个文件的内容读取到一个字符串或者一个行数组中。

```
# 读到字符串中
str = IO.read("testfile")
str.length → 66
str[0, 30] → "This is line one\nThis is line "

# 读到一个数组中
arr = IO.readlines("testfile")
arr.length → 4
arr[0] → "This is line one\n"
```

不要忘了在不确定的世界里，I/O 也必然是不确定的——大多数的错误会引发异常，你需要采取适当的行动以从错误中恢复。

10.3.2 写文件

Writing to Files

到目前为止，我们已经多次调用过 `puts` 和 `print`，并传递任意已存在的对象作为参数，我们相信 Ruby 会对之进行正确的处理（当然，它也是这么做的）。但它到底做了些什么呢？

答案很简单：除了少数几个例外，你传递给 `puts` 和 `print` 的任意对象都会被该对象的 `to_s` 方法转换成一个字符串。如果由于某种原因，`to_s` 方法未能返回一个合法的字符串，含有对象类名和 ID 的一个字符串就会被创建，类似于`#<ClassName:0x123456>`。

也有例外的情况。`nil` 对象会输出字符串“`nil`”，而传递给 `puts` 的数组会依次将它的元素打印出来，就像每个元素被分别传递给 `puts` 一样。

如果你想写入二进制数据而不想被 Ruby 干扰该怎么办？通常来说可以调用 `IO#print`，并以包括待写入字节的字符串作为参数。不过你也可以使用底层的输入输出例程，如果你真的想这么做——请参见第 514 页的 `IO#sysread` 和 `IO#syswrite` 的文档。

我们如何才能将二进制数据存储到字符串中呢？常用的有 3 个方法：字符串的字面量，一个字节一个字节地存入，或使用 `Array#pack`。

```

str1 = "\001\002\003"      →      "\001\002\003"
str2 = ""
str2 << 1 << 2 << 3      →      "\001\002\003"
[ 1, 2, 3 ].pack("c*")    →      "\001\002\003"

```

然而我怀念 C++ 的 `iostream`

有时实在无法计较个人的喜好……不过，就如你可以使用 `<<` 操作符添加一个对象到数组一样，你也可以添加对象到 IO 输出流。

```

endl = "\n"
STDOUT << 99 << " red balloons" << endl

```

输出结果：

```
99 red balloons
```

同样，方法 `<<` 使用 `to_s` 将它的参数转换成字符串，然后再按照它的方式传递。

尽管一开始我们鄙视可怜的`<<`操作符，但确实有一些使用它的理由。因为其他类（例如 `String` 和 `Array`）也实现了相似语义的`<<`操作符，所以你可以使用`<<`编写代码来附加某些东西，而不必关心是添加到数组、文件还是字符串。这种灵活性也使得单元测试比较简单。我们将在第 365 页开始的“duck typing”一章更详细地讨论这种思想。

1.8 使用字符串 I/O

很多时候，你需要处理假定读写一个或者多个文件的代码。但问题是：数据并不位于文件中。它或许是 SOAP 服务产生的数据，又或是从命令行传递来的参数。也可能你正在运行单元测试，而你并不想改变真正的文件系统。

来看看 `StringIO` 对象，它们的行为很像其他 I/O 对象，不同的是它们读写的是字符串而不是文件。如果你为读而打开一个 `StringIO` 对象，你需要提供一个字符串给它，在此 `StringIO` 对象上进行的所有读操作都会从该字符串中读。同样，当你想向 `StringIO` 对象写入时，你要传递一个待填充的字符串。

```

require 'stringio'

ip = StringIO.new("now is\nthe time\n\tto learn\nRuby!")
op = StringIO.new("", "w")

ip.each_line do |line|
  op.puts line.reverse
end
op.string → "\nsi won\n\nemit eht\n\nrael ot\n!ybuR\n"

```

10.4 谈谈网络

Talking to Networks

Ruby 善于处理网络协议，无论是底层协议还是高层协议。

对那些需要处理网络层的人来说，Ruby 的套节字库提供了一组类（请参见第 763 页）。这些类让你可以访问 TCP、UDP、SOCK、Unix 域套节字，以及在你的体系结构上支持的任意其他套节字类型。库中还提供了辅助类，这使得写服务器程序更容易。下面这个简单的例子使用 finger 协议获取本地机器上用户“mysql”的信息。

```
require 'socket'
client = TCPSocket.open('127.0.0.1', 'finger')
client.send("mysql\n", 0)          # 0 means standard packet
puts client.readlines
client.close
```

输出结果：

```
Login: mysql           Name: MySQL Server
Directory: /var/empty      Shell: /usr/bin/false
Never logged in.
No Mail.
No Plan.
```

在较高的层次上，lib/net 库提供了一组模块来处理应用层协议（目前支持 FTP，HTTP，POP，SMTP 和 telnet），详细请参见第 698 页。例如，下面的程序将列出 Pragmatic Programmer 主页上显示的所有图片。

```
require 'net/http'
h = Net::HTTP.new('www.pragmaticprogrammer.com', 80)
response = h.get('/index.html', nil)
if response.message == "OK"
  puts response.body.scan(//m).uniq
end
```

输出结果：

```
images/title_main.gif
images/dot.gif
images/Bookshelf_1.5_in_green.png
images/sk_all_small.jpg
images/new.jpg
```

尽管这个例子简单得令人着迷，但是还有很大的改进空间。特别是，它还没有做任何错误处理。它需要报告“Not Found”错误（声名狼藉的 404），还需要能处理重定向（当 web 服务器为客户端请求的页面给出一个替代地址时）。

我们还可以在更高层进行处理。通过装载 `open-uri` 库到程序中，方法 `Kernel.open` 立刻就可以识别文件名中的 `http://` 和 `ftp://` 等 URL。不仅如此，它还能自动处理重定向。

```
require 'open-uri'  
open('http://www.pragmaticprogrammer.com') do |f|  
  puts f.read.scan(/18</sup> 如果不想永远阻塞，可以给予 `join` 一个超时参数——如果超时在线程终止之前到期，`join` 返回 `nil`。`join` 的另一个变种，`Thread#value` 方法返回线程执行的最后语句的值。

除了 `join` 之外，另有一些便利函数用来操作线程。使用 `Thread.current` 总是可以得到当前线程。可以使用 `Thread.list` 得到一个所有线程的列表，返回包含所有可运行或被停止的线程对象。可以使用 `Thread#status` 和 `Thread#alive?` 去确定特定线程的状态。

另外，可以使用 `Thread#priority=` 调整线程的优先级。更高优先级的线程会在低优先级的线程前面运行。我们稍后就会更多地讨论线程调度、停止和启动线程。

## 线程变量

线程通常可以访问在它创建时其作用范围内的任何变量。线程 `block` 里面的局部变量是线程的局部变量，它们没有被共享。

但是如果需要线程局部变量能被别的线程访问——包括主线程，怎么办？`Thread` 类提供了一种特别的设施，允许通过名字来创建和访问线程局部变量。可以简单地把线程对象看作一个散列表，使用 `[ ]=` 写入元素并使用 `[ ]` 把它们读出。在下面的例子中，通过键 `mycount`，线程把 `count` 变量的当前值记录到线程局部变量中。在读取时，代码使用字符串“`mycount`”对线程对象进行索引（这里存在一个竞态条件（*race condition*）<sup>1</sup>，但是因为我们还没有讨论同步，所以现在只是悄悄地忽略它）。

---

<sup>1</sup> 竞态条件出现在当两段或多段代码（或硬件）都试图访问一些共享资源时，在这里结果会随着它们执行的次序而改变。在这个例子中，有可能一个线程将其 `mycount` 变量的值设置给 `count`，但是在它有机会增加 `count` 之前，这个线程被调度了出去，而另外一个线程重用了相同的 `count` 值。这个问题可以通过对共享资源的访问（如 `count` 变量）进行同步解决。

```

count = 0
threads = []
10.times do |i|
 threads[i] = Thread.new do
 sleep(rand(0.1))
 Thread.current["mycount"] = count
 count += 1
 end
end
threads.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"

```

输出结果：

```
4, 1, 0, 8, 7, 9, 5, 6, 3, 2, count = 10
```

主线程等待子线程结束，然后打印出每个子线程获得的 `count` 值。纯粹是为了让它变得更有趣些，在记录这个值之前，我们让每个线程都随机等待了一段时间。

### 11.1.2 线程和异常

#### Threads and Exceptions

如果线程引发（`raise`）了未处理的异常，会发生些什么呢？这依赖于 `abort_on_exception` 标志（将在第 633 页和第 636 页说明）和解释器 `debug` 标志（将在第 178 页说明）的设置。

如果 `abort_on_exception` 是 `false`, `debug` 标志没有启用（默认条件），未处理的异常会简单地杀死当前线程——而所有其他线程继续运行。实际上，除非对引发这个异常的线程调用了 `join`，你甚至根本不知道这个异常存在。

在下面的例子中，线程 2 自爆了，同时没有任何输出。但是你仍然可以从其他线程看到蛛丝马迹。

```

threads = []
4.times do |number|
 threads << Thread.new(number) do |i|
 raise "Boom!" if i == 2
 print "#{i}\n"
 end
end
threads.each {|t| t.join }

```

输出结果：

```

0
1
3
prog.rb:4: Boom! (RuntimeError)
from prog.rb:8:in `join'
from prog.rb:8
from prog.rb:8:in `each'
from prog.rb:8

```

当线程被 `join` 时，我们可以 `rescue` 这个异常。

```
threads = []
4.times do |number|
 threads << Thread.new(number) do |i|
 raise "Boom!" if i == 2
 print "#{i}\n"
 end
end
threads.each do |t|
 begin
 t.join
 rescue RuntimeError => e
 puts "Failed: #{e.message}"
 end
end
```

输出结果：

```
0
1
3
Failed: Boom!
```

但是，设置 `abort_on_exception` 为 `true`，或者使用 `-d` 选项去打开 `debug` 标志，未处理的异常会杀死所有正在运行的线程。一旦线程 2 退出，不会产生更多的输出。

```
Thread.abort_on_exception = true
threads = []
4.times do |number|
 threads << Thread.new(number) do |i|
 raise "Boom!" if i == 2
 print "#{i}\n"
 end
end
threads.each { |t| t.join }
```

输出结果：

```
0
1
prog.rb:5: Boom! (RuntimeError)
from prog.rb:4:in `initialize'
from prog.rb:4:in `new'
from prog.rb:4
from prog.rb:3:in `times'
from prog.rb:3
```

这段代码也说明了一个容易犯错的地方（gotcha）。在循环里面，线程使用 `print` 而不是 `puts` 去输出数字，为什么呢？这是因为，在幕后 `puts` 的工作被分为两部分：输出其参数，然后再输出回车换行符。在这两个动作之间，一个线程可能得到调度，这样它们的输出可能会交织在一起。用已经包含回车换行符的字符串作为参数调用 `print`，则规避了这个问题。

## 11.2 控制线程调度器

### Controlling the Thread Scheduler

在设计良好的程序中，你通常只是让线程做它们该做的事情；多线程程序中建立时间依赖性（timing dependency）通常被认为是糟糕的设计，因为这会导致代码复杂化同时阻碍线程调度器优化程序的执行。

但是有时候需要显式地控制线程。也许点唱机有一个显示光束的线程。音乐停止时需要暂时停止它。因而这两个线程可能处于一种经典的生产者-消费者关系中，如果生产者的订单没有完成，则消费者必须暂停。

`Thread` 类提供了若干种控制线程调度器的方法。调用 `Thread.stop` 停止当前线程，调用 `Thread#run` 安排运行特定的线程。`Thread.pass` 把当前线程调度出去，允许运行别的线程，`Thread#join` 和 `Thread#value` 挂起调用它们的线程，直到指定的线程结束为止。

下面的程序说明了这些特性。它创建了 `t1` 和 `t2` 两个子线程，每个子线程运行 `Chaser` 类的一个实例。`chase` 方法对 `count` 加 1，但不会让它比另外一个线程里的 `count` 值多两个以上。为了防止差距大于 2，这个方法调用了 `Thread.pass`，它允许线程中的 `chase` 赶上来。为了变得有趣些，我们一开始就让这些线程把自己挂起，然后随机地启动其中一个线程。

```
class Chaser
 attr_reader :count
 def initialize(name)
 @name = name
 @count = 0
 end
 def chase(other)
 while @count < 5
 while @count-other.count > 1
 Thread.pass
 end
 @count += 1
 print "#{@name}: #{@count}\n"
 end
 end
end
c1 = Chaser.new("A")
c2 = Chaser.new("B")
threads = [
 Thread.new { Thread.stop; c1.chase(c2) },
 Thread.new { Thread.stop; c2.chase(c1) }
]
```

```

start_index = rand(2)
threads[start_index].run
threads[1-start_index].run
threads.each {|t| t.join }

```

输出结果：

```

A: 1
A: 2
B: 1
B: 2
B: 3
B: 4
A: 3
A: 4
A: 5
B: 5

```

但是在实际的代码中使用这些原语（primitive）实现同步并不是一件容易的事情——竞态条件总是等着趁机咬你一口。同时处理共享数据时，竞态条件总是会带来漫长且令人沮丧的调试阶段。实际上，前面例子包含了这样的错误：有可能在一个线程中 `count` 被增加了，但是在 `count` 被输出之前，第二个线程得到调度并且输出了 `count`。这个输出结果将会是乱序的。

值得庆幸的是，线程有辅助的设施——互斥的概念。使用互斥，可以实现许多安全的同步机制。

## 11.3 互斥

### Mutual Exclusion

这是最底层的阻止其他线程运行的方法，它使用了全局的线程关键（*thread-critical*）条件。当条件被设置为 `true`（使用 `Thread.critical=` 方法）时，调度器将不会调度现有的线程去运行。但是这不会阻止创建和运行新线程。某些线程操作（如停止或杀死线程，在当前线程中睡眠和引发异常）会导致即使线程处于一个关键区域内也会被调度出去。

直接使用 `Thread.critical=` 当然是可能的，但是它不是很方便。实际上，我们强烈建议不要使用它，除非你是一个多线程编程（和喜欢长时间调试）的黑带高手（black belt）。值得庆幸的是，Ruby 有多种变通方法。马上会看到其中的一种，`Monitor` 库。你也许想看一下 `Sync` 库（第 738 页）、`Mutex_m` 库（第 697 页开始）和实现在 `thread` 库中的 `Queue` 类（第 743 页）。

### 11.3.1 监视器 Monitors

尽管这些线程原语提供了基本的同步机制，但是熟练使用它们需要很多技巧。这些年来，各种各样的人提出了更高级别的替代方法。尤其在面向对象系统的领域中，其中比较成功的一个方法是监视器（monitor）的概念。

监视器用同步函数对包含一些资源的对象进行了封装。为了看看在实际中如何使用它们，考察一个被两个线程访问的简单计数器。

```
class Counter
 attr_reader :count
 def initialize
 @count = 0
 end
 def tick
 @count += 1
 end
end

c = Counter.new

t1 = Thread.new { 100_000.times { c.tick } }
t2 = Thread.new { 100_000.times { c.tick } }

t1.join
t2.join

c.count #⇒ 130082
```

也许你会惊讶，`count` 不等于 200 000。下面一行是罪魁祸首。

```
@count += 1
```

这一行实际上比看起来要复杂得多。Ruby 解释器会把它分成：

```
val = fetch_current(@count)
add 1 to val
store val back into @count
```

想象一下如果两个线程同时执行这段代码。下一页的表 11.1 显示了线程号（`t1` 和 `t2`）所执行的代码以及计数器的值（被初始化为 0）。

尽管基本的 `load/add/store` 指令集被执行了 5 次，可是最终的 `count` 是 3。这是因为线程 1 在中间中断了线程 2 的执行，当线程 2 恢复运行时，它把一个不新鲜（旧）的值保存到`@count` 中。

表 11.1 在一个竞态条件中的两个线程

| 线程  | 执行……                         | 结果         |
|-----|------------------------------|------------|
| t1: | val = fetch_current (@count) | @count = 0 |
| t1: | add 1 to val                 | 0          |
| t1: | store val back into @count   | @count = 1 |
| t2: | val = fetch_current (@count) | 1          |
| t2: | add 1 to val                 | 1          |
| t2: | store val back into @count   | @count = 2 |
| t1: | val = fetch_current (@count) | 2          |
| t2: | val = fetch_current (@count) | 2          |
| t1: | add 1 to val                 | 2          |
| t1: | store val back into @count   | @count = 3 |
| t1: | val = fetch_current (@count) | 3          |
| t1: | add 1 to val                 | 3          |
| t1: | store val back into @count   | @count = 4 |
| t2: | add 1 to val                 | 4          |
| t2: | store val back into @count   | @count = 3 |

解决办法是作出妥善安排，以确保每次只有一个线程能够在 `tick` 方法中执行这个增加。使用监视器可以容易做到这点。

```
require 'monitor'
class Counter < Monitor
 attr_reader :count
 def initialize
 @count = 0
 super
 end
 def tick
 synchronize do
 @count += 1
 end
 end
end

c = Counter.new
t1 = Thread.new { 100_000.times { c.tick } }
t2 = Thread.new { 100_000.times { c.tick } }

t1.join; t2.join
c.count #~ 200000
```

通过把计数器变成监视器，它获得了 `synchronize` 方法。对于特定的监视器对象，每次只有一个线程能够执行 `synchronize block` 中的代码，所以再也不会有两个线程同时缓存中间结果，这样的计数值才是我们所期望的。

为了获得这些益处，没有必要让我们的类变成 `Monitor` 的子类。也可以 mix in 使用监视器的一个变体，`MonitorMixin`。

```
require 'monitor'

class Counter
 include MonitorMixin
 ...
end
```

前面的例子把同步放在需要同步的资源里面。当需要对类中所有对象的所有访问进行同步时，这是恰当的方式。但是如果在某些情况下才需要对对象的访问进行同步，或者如果对一组对象进行同步，最好是使用一个外部的监视器。

```
require 'monitor'

class Counter
 attr_reader :count
 def initialize
 @count = 0
 end
 def tick
 @count += 1
 end
end

c = Counter.new
lock = Monitor.new

t1 = Thread.new { 100_000.times { lock.synchronize { c.tick } } }
t2 = Thread.new { 100_000.times { lock.synchronize { c.tick } } }
t1.join; t2.join

c.count → 200000
```

甚至可以把特定的对象放入监视器。

```
require 'monitor'

class Counter
 # as before...
end

c = Counter.new
c.extend(MonitorMixin)

t1 = Thread.new { 100_000.times { c.synchronize { c.tick } } }
t2 = Thread.new { 100_000.times { c.synchronize { c.tick } } }

t1.join; t2.join
c.count → 200000
```

在此，因为 Counter 类在被定义的时候不知道它是一个监视器，所以必须在外部执行同步（这个例子是通过封装 c.tick 的调用）。显然这很危险：如果别的代码调用了 tick 但是没有意识到需要同步，则我们又回到开始面临的境地了。

### 11.3.2 队列 Queues

本节中的大多数例子使用 Monitor 类进行同步。当然，也可以用别的技术，尤其当需要同步生产者和消费者之间的工作时。线程库中的 Queue 类实现了一个线程安全的队列机制。多个线程可以添加和从队列中删除对象，保证了每次添加和删除是原子性的操作。第 743 页的线程库描述中有它的样例。

### 11.3.3 条件变量 Condition Variables

监视器提供了一半我们需要的特性，但是这里有一个问题。假设两个线程访问共享的队列。一个要添加条目，而另一个要读取它们（在点唱机中，也许那个列表中是等待被播放的歌曲，用户选取的歌曲被添加到列表，然后歌曲播放后被清空）。

我们知道需要对访问进行同步，所以试着这么写：

```
require 'monitor'

playlist = []
playlist.extend(MonitorMixin)

播放者线程
Thread.new do
 record = nil
 loop do
 playlist.synchronize do # < < BUG!!!
 sleep 0.1 while playlist.empty?
 record = playlist.shift
 end
 play(record)
 end
end

Customer request thread
Thread.new do
 loop do
 req = get_customer_request
 playlist.synchronize do
 playlist << req
 end
 end
end
```

但是这段代码有一个问题。播放器线程获得对监视器的访问，然后循环等待直到歌曲被添加到 `playlist` 为止。但是因为拥有了监视器，客户线程将永远无法进入它的同步块，永远无法把东西添加到 `playlist`。我们陷入了困境。我们需要能够发出信号通知别人 `playlist` 已经有内容，并基于这个条件在线程之间提供同步，所有这些动作都处于监视器的安全范围。更一般地讲，需要能够暂时放弃对关键区域的排他性使用，同时告诉别人我们正在等待资源。当资源变得可用时，我们需要能够抓住它并重新获得关键区域的锁，这些动作要在一个步骤里完成。

这是条件变量大展身手的地方。条件变量是一种在两个线程之间交换事件（或条件）的受控方式。一个线程等待条件，而另外一个发信号通知条件。例如，可以使用条件变量重写点唱机（我们会编写存根 [stub] 方法，让它们使用条件变量去接受客户请求和播放歌曲。还必须添加一个标志通知播放器关闭；一般情况下它会一直运行）。

```
require 'monitor'

SONGS = [
 'Blue Suede Shoes',
 'Take Five',
 'Bye Bye Love',
 'Rock Around The Clock',
 'Ruby Tuesday'
]
START_TIME = Time.now

def timestamp
 (Time.now - START_TIME).to_i
end

Wait for up to two minutes between customer requests

def get_customer_request
 sleep(120 * rand)
 song = SONGS.shift
 puts "#{timestamp}: Requesting #{song}" if song
 song
end

Songs take between two and three minutes

def play(song)
 puts "#{timestamp}: Playing #{song}"
 sleep(120 + 60*rand)
end

ok_to_shutdown = false
and here's our original code
playlist = []
playlist.extend(MonitorMixin)
```

```

plays_pending = playlist.new_cond
Customer request thread
customer = Thread.new do
 loop do
 req = get_customer_request
 break unless req
 playlist.synchronize do
 playlist << req
 plays_pending.signal
 end
 end
end

Player thread
player = Thread.new do
 loop do
 song = nil
 playlist.synchronize do
 break if ok_to_shutdown && playlist.empty?
 plays_pending.wait_while { playlist.empty? }
 song = playlist.shift
 end
 break unless song
 play(song)
 end
end

customer.join
ok_to_shutdown = true
player.join

```

输出结果：

```

25: Requesting Blue Suede Shoes
26: Playing Blue Suede Shoes
70: Requesting Take Five
195: Requesting Bye Bye Love
205: Playing Take Five
294: Requesting Rock Around The Clock
305: Requesting Ruby Tuesday
385: Playing Bye Bye Love
551: Playing Rock Around The Clock
694: Playing Ruby Tuesday

```

## 11.4 运行多个进程

### Running Multiple Processes

有时候也许需要把任务分成几个进程来处理——或者也许需要运行的不是用 Ruby 编写的独立进程。这不是一个问题：Ruby 有好几个方法去衍生（spawn）和管理独立进程。

## 11.4.1 衍生新进程

### Spawning New Processes

有好几种方式可以衍生单独的进程；最简单的方式是运行一些命令然后等待它结束。你可能发现，要运行单独的命令或者从主机系统取回数据时，你也会这么做。Ruby 使用 `system` 命令和反引号（或 `backtick`）方法来实现它。

```
system("tar xzf test.tgz") → true
result = `date` → "Wed May 3 16:56:19 CDT 2006\n"
```

`Kernel.system` 方法在子进程中执行给定的命令，如果命令被找到和正确执行了，它会返回 `true`，否则返回 `false`。如果失败了，子进程的退出码保存在全局变量`$?`中。

使用 `System` 的问题是，这个命令和程序使用相同的输出流，这可能是我们不希望看到的。就像前面例子中的`date`那样，可以使用反引号字符得到子进程的标准输出。回想一下，可以使用 `String#chomp` 从结果中删除行结束字符。

好的，对简单的例子这种处理没问题——运行一些别的进程然后得到它们的返回状态。但是很多情况下我们需要更多的控制：跟子进程进行对话，发送数据给它和从它那儿得到数据。`IO.popen` 方法可以这么做。`popen` 方法在子进程里运行命令，把子进程的标准输入和标准输出连接到 Ruby `IO` 对象。它把数据写入到 `IO` 对象，子进程可以从它的标准输入中读到。不论子进程写入什么数据，Ruby 程序通过读取 `IO` 对象来得到它。

例如，`pig` 是系统中最有用的一个工具，它从标准输入读出单词，同时以 `pig latin` 的形式（蓄意颠倒英语字母顺序拼凑而成的行话）把它们打印出来。当我们的 Ruby 程序发送那些需要瞒过五岁小孩儿的消息时，可以使用这个工具。

```
pig = IO.popen("/usr/local/bin/pig", "w+")
pig.puts "ice cream after they go to bed"
pig.close_write
puts pig.gets
```

输出结果：

```
iceway eamcray afterway eythay ogay otay edbay
```

这个例子说明，当通过管道驱使子进程时，同时涉及到了表面上的简单性和现实世界的复杂性。这段代码看起来非常简单：打开管道，写入一句话同时读回响应。但我们发现 `pig` 程序并没有刷新它的输出。我们最初尝试在 `pig.puts` 后面跟着 `pig.gets`，但

是这会造成程序的永久挂起。`pig` 程序处理了输入，但是它的响应却永远不会写入到管道中。我们不得不插入 `pig.close_write` 代码行来解决这个问题。它会发送一个文件结束符到 `pig` 的标准输入，这样我们盼望的输出就随着 `pig` 的终止而被刷新。

`popen` 还有另外一个值得注意的地方。如果被传递的命令是单个减号（-），`popen` 会创建（fork）新的 Ruby 解释器。这个解释器和原先的解释器在 `popen` 返回后都会继续运行。原先的进程会接受到 `IO` 对象，而子进程得到 `nil`。这个函数只能在支持 `fork(2)` 调用（因而不包括 Windows）的操作系统上工作。

```
pipe = IO.popen("-", "w+")
if pipe
 pipe.puts "Get a job!"
 STDERR.puts "Child says '#{pipe.gets.chomp}'"
else
 STDERR.puts "Dad says '#{gets.chomp}'"
 puts "OK"
end
```

输出结果：

```
Dad says 'Get a job!'
Child says 'OK'
```

除了 `popen` 方法，一些平台支持 `Kernel.fork`，`Kernel.exec` 和 `IO.pipe` 方法。如果把！作为文件名的第一个字符（详见第 503 页的 `IO` 类介绍），许多符合文件命名惯例的 `IO` 方法和 `Kernel.open` 也会衍生子进程。注意不能使用 `File.new` 创建管道，它只是用来创建文件。

## 11.4.2 独立子进程

### Independent Children

有时候并不需要如此大动干戈：我们愿意把任务交给子进程，然后继续做我们的事情。我们在晚些时候查看子进程是否已经结束了。比如，我们也许想运行一个需要长时间运行的外部排序。

```
exec("sort testfile > output.txt") if fork.nil?
The sort is now running in a child process
carry on processing in the main program
... dum di dum ...
then wait for the sort to finish
Process.wait
```

`Kernel.fork` 调用会在父进程中返回进程 ID，在子进程中返回 `nil`，这样子进程会执行 `Kernel.exec` 并运行 `sort`。然后我们调用 `Process.wait` 等待 `sort` 结束（同时返回它的进程 ID）。

如果你更愿意在子进程结束时收到通知，而不是在那儿傻等，可以使用 `Kernel.trap`（在第 534 页描述）设置信号处理方法。这里在 `SIGCLD` 上设置了 `trap`，子进程终止时 `SIGCLD` 信号会被发给父进程。

```
trap("CLD") do
 pid = Process.wait
 puts "Child pid #{pid}: terminated"
end

exec("sort testfile > output.txt") if fork.nil?
do other stuff...
```

输出结果：

```
Child pid 25816: terminated
```

关于使用和控制外部进程的更多信息，请参见 `Kernel.open` 和 `IO.popen` 文档以及第 583 页的 `Process` 模块一节。

### 11.4.3 block 和子进程

#### Blocks and Subprocesses

`IO.popen` 与 `block` 相伴工作，很大程度上和 `File.open` 所做的一样。如果把命令传递给它，例如 `date`，一个 `IO` 对象会作为参数传递给 `block`。

```
IO.popen("date") { |f| puts "Date is #{f.gets}" }
```

输出结果：

```
Date is Thu Aug 26 22:36:55 CDT 2004
```

就像 `File.open` 那样，`block` 结束时 `IO` 对象会被自动关闭。

如果用 `Kernel.fork` 关联 `block`，`block` 中的代码会在 Ruby 子进程中运行，父进程会在 `block` 后面继续运行。

```
fork do
 puts "In child, pid = #$$"
 exit 99
end
pid = Process.wait
puts "Child terminated, pid = #{pid}, status = #{$?.exitstatus}"
```

输出结果：

```
In child, pid = 25823
Child terminated, pid = 25823, status = 99
```

`$?` 是全局变量，它包含子进程结束时的信息。更多信息请看第 591 页开始的 `Process::Status` 一节。

## 单元测试

### Unit Testing

单元测试（在下一页的侧栏描述）是一项帮助开发者编写更好的代码的技术。测试在代码被实际编写之前就有帮助，因为它自会引导你创建更好、更解耦的设计。当你编写代码时，它也很有帮助，可以向你及时反馈代码是否准确无误。在你完成代码之后，测试同样很有帮助，因为它能够让你检查代码是否可以正常工作，并且还帮助其他人理解如何使用你的代码。

单元测试是好事情。

但是，在一本 Ruby 的书籍中，为什么将关于单元测试的一章放在中间部分呢？因为单元测试对类如 Ruby 这样的动态语言，似乎携手而来。Ruby 的灵活性使得编写测试非常简单，而测试让你能够更容易地验证你的代码。一旦渐入佳境，你会发现自己通常以这样一种步骤工作：先编写一些代码，再编写一两个测试，验证所有事情都完美正确，然后编写更多的代码。

单元测试还很琐碎——运行一个程序来调用应用的部分代码，得到若干结果，然后检查结果是否如你所愿。

假定我们测试一个罗马数字类。目前代码还很简单：它只是让我们创建一个表示特定数字的对象，并将该对象以罗马数字显示。第 153 页中的插图 12.1，演示了在我们实现中的第一个刺痛。

我们通过编写另一个程序来测试这段代码，如下所示。

```
require 'roman'

r = Roman.new(1)
fail "'i' expected" unless r.to_s == "i"

r = Roman.new(9)
fail "'ix' expected" unless r.to_s == "ix"
```

不过，随着项目中测试数量的增长，这种自由散漫（ad-hoc）的方式会使测试变得复杂而难于管理。多年以来，出现了许多单元测试框架来帮助组织测试过程。Ruby 自带了一个预安装的、由 Nathaniel Talbott 编写的 Test::Unit 框架。

## 何为单元测试

### What is Unit Testing

单元测试专注于小块（单元）的代码，一般是单个方法或方法中的几行。这和其他的将整个系统视为一体的测试形式区别开来。

为何要专注于斯呢？因为最后所有软件都是以层次而架构的：某一层次的代码依赖于下一层代码的正确操作。如果下层的代码结果包含了 bug，那么所有上面的层次都会有潜在的影响。这是一个大问题。Fred 可能每星期编写的代码包含一个 bug，而两个月以后，你可能会间接地调用了它。当你的代码产生不正确的结果时，你可能颇费一段时间来追溯，问题产生在 Fred 的方法中。当你询问 Fred 为什么这样编写这个方法时，回答很可能是“我不记得了，都过去两个月了。”

如果 Fred 对它的代码进行了单元测试，可能发生两种情况。第一，当代码在头脑中依然鲜活时，他就找出了 bug。第二，因为单元测试只查看他刚刚编写的代码，当 bug 真的发生时，他只需要查看少数代码行就可以发现，而不必在整个代码基（code base）上考古了。

## 12.1 Test::Unit 框架

### Test::Unit Framework

Test::Unit 框架基本上是将 3 个功能包装到一个整洁的包中。

- 它提供了一种表示单个测试的方式。
- 它提供了一个框架来组织测试。
- 它提供了灵活的方式来调用测试。

#### 12.1.1 断言 == 预期的结果

##### Assertions == Expected Results

与其让你在测试中编写一系列单独的 `if` 语句，还不如让 Test::Unit 提供一系列断言（assertion）来达到同样的目标。虽然存在许多不同风格的断言，但是它们基本上都遵循相同的模式。每个断言都向你提供指定预想的结果或输出、以及传入实际输出的方式。如果实际结果和预期的不同，断言会输出一条漂亮的消息，并将此次记录为一次失败。

例如，我们可以使用 Test::Unit 重写之前对于 Roman 类的测试。现在，忽略开始和结束处的框架（scaffolding）代码，只考察 `assert_equal` 方法。

```

class Roman
 MAX_ROMAN = 4999

 def initialize(value)
 if value <= 0 || value > MAX_ROMAN
 fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
 end
 @value = value
 end

 FACTORS = [["m", 1000], ["cm", 900], ["d", 500], ["cd", 400],
 ["c", 100], ["xc", 90], ["l", 50], ["xl", 40],
 ["x", 10], ["ix", 9], ["v", 5], ["iv", 4],
 ["i", 1]]

 def to_s
 value = @value
 roman = ""
 for code, factor in FACTORS
 count, value = value.divmod(factor)
 roman << code unless count.zero?
 end
 roman
 end
end

```

图 12.1 产生罗马数字（有 bug）

```

require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
 def test_simple
 assert_equal("i", Roman.new(1).to_s)
 assert_equal("ix", Roman.new(9).to_s)
 end
end

```

输出结果：

```

Loaded suite -
Started

Finished in 0.003655 seconds.

1 tests, 2 assertions, 0 failures, 0 errors

```

第一个断言表示的是，我们期待 1 的 Roman 数字字符串表示为“i”，而第二个测试表示我们希望 9 对应为“ix”。我们很幸运，两个预期都应验了，并且追踪结果报告我们的测试通过了。

让我们添加多些测试。

```

require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
 def test_simple
 assert_equal("i", Roman.new(1).to_s)
 assert_equal("ii", Roman.new(2).to_s)
 assert_equal("iii", Roman.new(3).to_s)
 assert_equal("iv", Roman.new(4).to_s)
 assert_equal("ix", Roman.new(9).to_s)
 end
end

```

输出结果：

```

Loaded suite -
Started
F
Finished in 0.021877 seconds.

1) Failure:
<"ii"> expected but was
<"i">.

1 tests, 2 assertions, 1 failures, 0 errors
test_simple(TestRoman) [prog.rb:6]:

```

啊呀！第二个断言失败了。看看错误消息如何利用断言所了解的预期值和实际值信息：它预期得到“ii”，相反得到的是“i”。查看我们的代码，你可以看到 `to_s` 中有一个明显的 bug。如果用 `factor` 除得到的值大于 0，则我们应该输出多个罗马数字。而现有的代码只输出一个。修改很简单。

```

def to_s
 value = @value
 roman = ""
 for code, factor in FACTORS
 count, value = value.divmod(factor)
 roman << (code * count)
 end
 roman
end

```

现在，让我们再一次运行测试。

```

Loaded suite -
Started
.
Finished in 0.002161 seconds.

1 tests, 5 assertions, 0 failures, 0 errors

```

看起来不错。我们现在可以再进一步，同时删除一些重复。

```

require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
 NUMBERS = [
 [1, "i"], [2, "ii"], [3, "iii"],
 [4, "iv"], [5, "v"], [9, "ix"]
]
 def test_simple
 NUMBERS.each do |arabic, roman|
 r = Roman.new(arabic)
 assert_equal(roman, r.to_s)
 end
 end
end

```

输出结果：

```

Loaded suite -
Started

Finished in 0.001179 seconds.

1 tests, 6 assertions, 0 failures, 0 errors

```

我们还可以测试些什么呢？好吧，构造函数检查我们传入的数字，是否可以被表示为罗马数字，如果不能则引发一个异常。让我们测试异常吧。

```

require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
 def test_range
 assert_raise(RuntimeError) { Roman.new(0) }
 assert_nothing_raised() { Roman.new(1) }
 assert_nothing_raised() { Roman.new(4999) }
 assert_raise(RuntimeError) { Roman.new(5000) }
 end
end

```

输出结果：

```

Loaded suite -
Started

Finished in 0.001125 seconds.

1 tests, 4 assertions, 0 failures, 0 errors

```

虽说我们可以对 `Roman` 类多做一些测试，不过让我们迈向更强大更美好的内容。但是在继续前行之前，应该强调我们只是触及了 `Test::Unit` 断言的皮毛。第 162 页中的插图 12.2 给出了完整的列表。每个断言的最后一个参数是一条消息，会在任何错误消息之前

输出。一般它是不需要的，因为 `Test::Unit` 的消息通常已经很适用了。测试 `assert_not_nil` 是一个例外，它的消息“`<nil> expected to not be nil`”帮不上什么忙。在这种情况下，你可能希望添加自己的一些注解。

```
require 'test/unit'
class TestsWhichFail < Test::Unit::TestCase
 def test_reading
 assert_not_nil(ARGF.read, "Read next line of input")
 end
end
```

输出结果：

```
Loaded suite -
Started
F
Finished in 0.033581 seconds.

1) Failure:
Read next line of input.
<nil> expected to not be nil.

1 tests, 1 assertions, 1 failures, 0 errors
test_reading(TestsWhichFail) [prog.rb:4]:
```

## 12.2 组织测试

### Structuring Tests

之前我们让你忽略测试外围的框架代码，现在是时候来考察它们了。

在你的单元测试中，使用下面的代码行包含 `Test::Unit` 的功能。

```
require 'test/unit'
```

单元测试，可以很自然地被组织成更高层的形式，叫做测试用例 (*test case*)；或者分解成较底层的形式，也就是测试方法本身。测试用例通常包括和某个特定功能或特性相关的所有测试。我们的 `Roman` 数字类非常简单，因此所有测试只需放在一个单独的测试用例中。在这个测试用例中，我们可能希望将你的断言组织成一系列的测试方法，每个方法都包括针对某种测试类型的断言：例如一个方法可能检查一般的数字转换，而另一个可以测试错误处理，等等。

表示测试的类必须是 `Test::Unit::TestCase` 的子类。含有断言的方法名必须以 `test` 开头。这是很重要的：`Test::Unit` 使用反射（reflection）来查找要运行的测试，而只有以 `test` 开头的方法才符合条件。

你经常可以发现，测试用例中的所有测试方法都会设置一个特定的场景（scenario）。每个测试方法考察该场景的某些方面。最终，每个方法会总结得到自己的结果。例如，我们要测试一个从数据库提取点唱机播放列表的类。

```
require 'test/unit'
require 'playlist_builder'
require 'dbi'

class TestPlaylistBuilder < Test::Unit::TestCase
 def test_empty_playlist
 db = DBI.connect('DBI:mysql:playlists')
 pb = PlaylistBuilder.new(db)
 assert_equal([], pb.playlist())
 db.disconnect
 end

 def test_artist_playlist
 db = DBI.connect('DBI:mysql:playlists')
 pb = PlaylistBuilder.new(db)
 pb.include_artist("krauss")
 assert(pb.playlist.size > 0, "Playlist shouldn't be empty")
 pb.playlist.each do |entry|
 assert_match(/krauss/i, entry.artist)
 end
 db.disconnect
 end

 def test_title_playlist
 db = DBI.connect('DBI:mysql:playlists')
 pb = PlaylistBuilder.new(db)
 pb.include_title("midnight")
 assert(pb.playlist.size > 0, "Playlist shouldn't be empty")
 pb.playlist.each do |entry|
 assert_match(/midnight/i, entry.title)
 end
 db.disconnect
 end

 # ...
end
```

输出结果：

```
Loaded suite -
Started
...
Finished in 0.004809 seconds.

3 tests, 23 assertions, 0 failures, 0 errors
```

每个测试首先连接数据库，然后创建一个新的播放列表生成器。最后每个测试和数据库断开。（在单元测试中使用真实数据库的想法是很成问题的，因为单元测试假定是

快速运行、上下文无关并且易于设定的，但是它说明了这一点。）

我们可以把这些通用的代码提取到 `setup` 和 `teardown` 方法中。在一个 `TestCase` 类中，一个叫做 `setup` 的方法将在每个测试方法之前运行，而叫做 `teardown` 的方法在每个测试方法结束之后运行。让我们强调一下：`setup` 和 `teardown` 方法是对每个测试、而非每个测试用例运行一次。

我们的测试可以变为：

```
require 'test/unit'
require 'playlist_builder'
require 'dbi'

class TestPlaylistBuilder < Test::Unit::TestCase
 def setup
 @db = DBI.connect('DBI:mysql:playlists')
 @pb = PlaylistBuilder.new(@db)
 end

 def teardown
 @db.disconnect
 end

 def test_empty_playlist
 assert_equal([], @pb.playlist())
 end

 def test_artist_playlist
 @pb.include_artist("krauss")
 assert(@pb.playlist.size > 0, "Playlist shouldn't be empty")
 @pb.playlist.each do |entry|
 assert_match(/krauss/i, entry.artist)
 end
 end

 def test_title_playlist
 @pb.include_title("midnight")
 assert(@pb.playlist.size > 0, "Playlist shouldn't be empty")
 @pb.playlist.each do |entry|
 assert_match(/midnight/i, entry.title)
 end
 end
 # ...
end
```

输出结果：

```
Loaded suite -
Started
...
Finished in 0.00691 seconds.

3 tests, 23 assertions, 0 failures, 0 errors
```

## 12.3 组织和运行测试

### Organizing and Running Tests

我们目前所演示的测试用例都是可以运行的 `Test::Unit` 程序。例如，如果 `Roman` 类的测试用例位于文件 `test_roman.rb` 中，我们可以使用下面的形式从命令行运行测试：

```
% ruby test_roman.rb
Loaded suite test_roman
Started
.
Finished in 0.039257 seconds.

2 tests, 9 assertions, 0 failures, 0 errors
```

`Test::Unit` 足够聪明，可以发现没有主程序，因此它会将所有的测试用例类集合起来并依次运行。

如果我们希望如此，可以让它运行一个特定的测试方法。

```
% ruby test_roman.rb --name test_range
Loaded suite test_roman
Started
.
Finished in 0.006445 seconds.

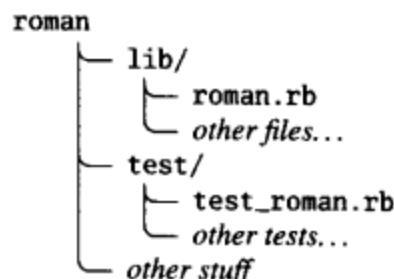
1 tests, 4 assertions, 0 failures, 0 errors
```

#### 12.3.1 何处存放测试

##### Where to Put Tests

一旦你习惯了单元测试，你可能会发现自己产生的测试代码几乎和产品代码一样多。所有这些测试都必须放在某个地方。问题是，如果你将它们和一般的产品代码源文件放在一起，你的目录会变得很臃肿——因为实际上你最终为每个产品源文件配备了两个文件。

一般的解决方法是建立一个 `test/` 目录，将你所有的测试源文件放置其中。这个目录和保存开发代码的目录平行放置。例如，对 `Roman` 数字类，我们具有下面的目录结构：



这是一个可用的组织文件的方式，但是留给你一个小问题：如何告诉 Ruby 到哪里查找要测试的库文件？例如，如果我们的 `TestRoman` 测试代码在 `test/` 子目录中，Ruby 如何知道从哪里查找我们要测试的 `roman.rb` 源文件呢？

一种不太可靠的方法是，将路径放到测试文件的 `require` 语句中，然后从 `test/` 子目录中运行测试。

```
require 'test/unit'
require '../lib/roman'
class TestRoman < Test::Unit::TestCase
 # ...
end
```

为什么这种方法无法奏效呢？第一，因为我们的 `roman.rb` 文件本身也可能需要类库中编写的其他文件。它将使用 `require`（而没有“`..lib/`”前缀）将它们加载进来，因为它们并不位于 Ruby 的 `$LOAD_PATH` 中，因此无法找到。我们的测试因而也无法运行。第二，一个不那么直接的问题是，我们将无法使用相同的测试来测试安装到目标系统上的类，所以我们应该简单地使用 `require 'roman'` 来引用它们。

一种稍好些的解决方法是，从被测试库的父目录中运行测试。因为当前的目录是在加载路径中，测试代码将可以找到它。

```
% ruby ../test/test_roman.rb
```

不过，当你想要运行系统中其他地方的测试时，这种方法就无法奏效了。可能，你所规划的构建过程要查找并执行名为 `test_xxx` 的文件来为应用中的所有软件运行测试。在这种情况下，你需要一点加载路径的魔法。在你测试代码的头部（例如在 `test_roman.rb` 中），添加下面的代码行：

```
$:.unshift File.join(File.dirname(__FILE__), "..", "lib")
require ...
```

这种技法可以工作，是因为测试代码相对被测试代码的存放位置是已知的。它首先得出测试文件运行所在的目录，然后构造被测试文件的路径。这个目录被加入加载路径（变量`$:`）的前部。之后，类似如 `require 'roman'` 的代码将首先搜索到被测试的库。

### 12.3.2 测试套件

#### Test Suites

一段时间之后，你的应用的测试用例集合可能会有可观的增长。你可能会发现类聚的倾向：一组用例测试某个特定的功能集合，而另一组用例测试另一个功能集合。这样的话，你可以将这些测试用例一并组成测试套件（*test suite*），然后以组的形式运行它们。

这在 `Test::Unit` 中是很容易的。你所要做的，就是创建一个要求加载 `test/unit` 的 Ruby 文件，然后要求加载每个你希望成组的、包含测试用例的文件。在这种方式下，你为自己建立了测试资料的一种层次结构。

- 你可以通过名字来运行单个的测试。
- 你可以通过某个文件来运行其中的所有测试。
- 你可以将多个文件组成一个测试套件，作为运行测试的单元。
- 你可以将多个测试套件组成另外的测试套件。

这让你可以在所掌控的任何粒度级别运行单元测试，只测试一个方法，或者测试整个应用。

至此，值得我们考虑一下命名约定。Nathaniel Talbott，`Test::Unit` 的作者，使用这样的约定，测试用例在以 `tc_XXX` 命名的文件中，而测试套件在 `ts_XXX` 命名的文件中。

```
file ts_dbaccess.rb
require 'test/unit'
require 'tc_connect'
require 'tc_query'
require 'tc_update'
require 'tc_delete'
```

现在，如果你运行 `ts_dbaccess.rb`，将会执行所要求加载的 4 个文件中的全部测试用例。

这就是全部了么？还不是，如果你希望，你可以使它更复杂。你可以手工地创建并填充 `TestSuite` 对象，但是在实践中好像没什么意义。如果你想要查找更多信息，`ri Test::Unit` 应该可以帮助你。

`Test::Unit` 还带有许多时髦的 GUI 测试运行器。不过，因为真正的程序员使用命令行，这里就不介绍它们了。再次，关于细节请参见文档。

`assert(boolean, [ message ])`  
如果 `boolean` 为 `false` 或 `nil`, 则失败.

`assert_nil(obj, [ message ])`  
`assert_not_nil(obj, [ message ])`  
预期 `obj` 为 (或不为) `nil`.

`assert_equal(expected, actual, [ message ])`  
`assert_not_equal(expected, actual, [ message ])`  
使用 `==` 来判定 `actual` 和 `expected` 是否相等.

`assert_in_delta(expected_float, actual_float, delta, [ message ])`  
预期实际的浮点值处于预期值的 `delta` (偏差) 之内.

`assert_raise(Exception, ...) { block }`  
`assert_nothing_raised(Exception, ...) { block }`  
预期 `block` 将会 (或不会) 引发参数列出的一个异常.

`assert_instance_of(klass, obj, [ message ])`  
`assert_kind_of(klass, obj, [ message ])`  
预期 `obj` 是 `klass` 的子类或实例.

`assert_respond_to(obj, message, [ message ])`  
预期 `obj` 能够响应 `message` (一个符号).

`assert_match-regexp, string, [ message ])`  
`assert_no_match-regexp, string, [ message ])`  
预期 `string` 符合 (或不符合) 正则表达式 `regexp`.

`assert_same(expected, actual, [ message ])`  
`assert_not_same(expected, actual, [ message ])`  
预期 `expected.equal?(actual)`.

`assert_operator(obj1, operator, obj2, [ message ])`  
预期用 `obj2` 为参数调用 `obj1` 的 `operator`, 且结果为 `true`.

`assert_throws(expected_symbol, [ message ]) { block }`  
预期 `block` 抛出指定的符号.

`assert_send(send_array, [ message ])`  
将 `send_array[1]` 中的消息发送给 `send_array[0]` 中的接收者, 将 `send_array` 的其余部分作为参数. 预期返回值为 `true`.

`flunk(message="Flunked")`  
永远失败.

图 12.2 Test::Unit 支持的断言

## 当遇到麻烦时

### When Trouble Strikes

很遗憾的是，用 Ruby 也可能写出有缺陷的程序。

但是不用担心！Ruby 有几个特性可以帮助你调试程序。我们会逐一介绍这些特性，还列出了使用 Ruby 常犯的一些错误，以及如何修正它们。

#### 13.1 Ruby 调试器 Ruby Debugger

Ruby 自带一个调试器，并被集成到 Ruby 的基本系统中。你可以通过在调用解释器时使用 `-r debug` 选项、连同 Ruby 的其他选项以及脚本名称，来运行调试器。

```
Ruby -r debug [调试选项] [程序文件] [程序参数]
```

调试器支持你所期望的常用功能，包括设置断点、步入或者跳过方法调用以及显示调用栈和变量。它还能列出某个对象或类中定义的实例方法，也能列出并控制 Ruby 中单独的线程。第 173 页的表 13.1 列出了调试器中可用的所有命令。

如果你在安装 Ruby 时启用了 `readline` 功能，那么就可以用方向键在命令的历史记录中来回选择，还可以使用行编辑命令来修正前面的输入。

下面的例子会话（session）可以让你对 Ruby 调试器有个初步的印象（粗体表示用户输入）。

```
% ruby -r debug t.rb
Debug.rb
Emacs support available.
t.rb:1:def fact(n)
(rdb:1) list 1-9
[1, 10] in t.rb
```

```

=> 1 def fact(n)
 2 if n <= 0
 3 1
 4 else
 5 n * fact(n-1)
 6 end
 7 end
 8
 9 p fact(5)
(rdb:1) b 2
Set breakpoint 1 at t.rb:2
(rdb:1) c
breakpoint 1, fact at t.rb:2
t.rb:2: if n <= 0
(rdb:1) disp n
 1: n = 5
(rdb:1) del 1
(rdb:1) watch n==1
Set watchpoint 2
(rdb:1) c
watchpoint 2, fact at t.rb:fact
t.rb:1:def fact(n)
1: n = 1
(rdb:1) where
--> #1 t.rb:1:in `fact'
 #2 t.rb:5:in `fact'
 #3 t.rb:5:in `fact'
 #4 t.rb:5:in `fact'
 #5 t.rb:5:in `fact'
 #6 t.rb:9
(rdb:1) del 2
(rdb:1) c
120

```

## 13.2 交互式 Ruby

### Interactive Ruby

如果你想练习使用 Ruby，我们建议使用交互式 Ruby——简称 irb。irb 实际上是 Ruby 的“shell”，它类似于操作系统的 shell（具备任务控制）。使用它提供的环境，你可以实时地“把玩”Ruby 语言。可以在命令行提示符下运行 irb。

`irb [ irb 选项 ] [ ruby 脚本 ] [ 程序参数 ]`

一旦你输入完一个表达式，irb 就会显示它的值。例如：

```
% irb
irb(main):001:0> a=1+
irb(main):002:0> 2*3/
irb(main):003:0> 4%5
```

```
=> 2
irb(main):004:0> 2+2
=> 4
irb(main):005:0> def test
irb(main):006:1> puts "Hello, world!"
irb(main):007:1> end
=> nil
irb(main):008:0> test
Hello, world!
=> nil
irb(main):009:0>
```

irb 中可以创建子会话，每个会话还可以有自己的上下文。例如，你可以创建一个和原始会话有相同上下文的子会话，也可以在某个类或实例的上下文中创建子会话。下一页的图 13.1 中显示的例子虽然有点长，但是它展示了如何创建子会话，以及如何在它们之间来回切换。

如果想得到 irb 支持的所有命令的完整描述，请参见从第 185 页开始的索引。

和调试器一样，如果你的 Ruby 版本内建了 GNU 的 readline 功能，那么就可以使用方向键（和 Emacs 一样）或 vi 风格的快捷键来编辑单个的输入行、也可返回到前面执行的命令来重新执行或编辑它——就像命令行的 shell 一样。

irb 是一个非常棒的工具：它非常便于你快速地尝试你的想法，以知道它是否可行。

## 13.3 编辑器支持 Editor Support

Ruby 解释器被设计成一次读一个程序：这意味着你可以通过管道把整个程序作为标准输入传递给解释器，而它就可以很好地工作。

我们可以利用这个优点在编辑器内部运行 Ruby 代码。例如在 Emacs 中，你可以选择一段 Ruby 代码，并使用 Meta-**!** 来执行它。Ruby 解释器会将选择的区域作为标准输入，并将输出写入名为 \*Shell Command Output\* 的缓冲区中。这个特性对我们编写本书非常方便——仅需要在段落中间选择几行 Ruby 代码就可以试着运行它！

在 vi 编辑器中你也可以做类似的事情：**:!ruby** 会用代码运行的结果替换程序文本，**:w !ruby** 会显示输出而不影响缓冲区。其他编辑器也有类似的特性。

因为我们正在讨论编辑器支持这个话题，所以这里也许有最合适的地方来指明，Ruby 源代码 misc/ 目录下的 ruby-mode.el 文件中包含了 Emacs 的 Ruby 模式。

```
% irb
irb(main):001:0> irb
irb#1(main):001:0> jobs
#0->irb on main (#<Thread:0x401bd654>: stop)
#1->irb#1 on main (#<Thread:0x401d5a28>: running)
irb#1(main):002:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
@context=#<IRB::Context:0x401ca86c>>
irb(main):002:0> class VolumeKnob
irb(main):003:1> end
=> nil
irb(main):004:0> irb VolumeKnob
irb#2(VolumeKnob):001:0> def initialize
irb#2(VolumeKnob):002:1> @vol=50
irb#2(VolumeKnob):003:1> end
=> nil
irb#2(VolumeKnob):004:0> def up
irb#2(VolumeKnob):005:1> @vol += 10
irb#2(VolumeKnob):006:1> end
=> nil
irb#2(VolumeKnob):007:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
@context=#<IRB::Context:0x401ca86c>>
irb(main):005:0> jobs
#0->irb on main (#<Thread:0x401bd654>: running)
#1->irb#1 on main (#<Thread:0x401d5a28>: stop)
#2->irb#2 on VolumeKnob (#<Thread:0x401c400c>: stop)
irb(main):006:0> VolumeKnob.instance_methods
=> ["up"]
irb(main):007:0> v = VolumeKnob.new
#<VolumeKnob: @vol=50>
irb(main):008:0> irb v
irb#3(#<VolumeKnob:0x401e7d40>):001:0> up
=> 60
irb#3(#<VolumeKnob:0x401e7d40>):002:0> up
=> 70
irb#3(#<VolumeKnob:0x401e7d40>):003:0> up
=> 80
irb#3(VolumeKnob):004:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
@context=#<IRB::Context:0x401ca86c>>
irb(main):009:0> kill 1,2,3
=> [1, 2, 3]
irb(main):010:0> jobs
#0->irb on main (#<Thread:0x401bd654>: running)
irb(main):011:0> exit
```

In this same irb session,  
we'll create a new  
subsession in the context  
of class VolumeKnob.

We can use fg 0 to  
switch back to the main  
session, take a look at all  
current jobs, and see what  
instance methods  
VolumeKnob defines.

Make a new VolumeKnob  
object, and create a new  
subsession with that  
object as the context.

Switch back to the main  
session, kill the  
subsessions, and exit.

图 13.1 irb 会话的示范

你也可以从网络上找到 vim (vi 编辑器的增强版), jed 和其他编辑器的语法高亮显示模块。查看 Ruby FAQ (<http://www.rubygarden.org/iowa/faqtotum>)，可以获得最新的信息和资源列表。

## 13.4 但是它不运作

### But It Doesn't Work

到这里你已经读了足够的章节，可以开始写自己的 Ruby 程序，但有时会遇到问题。下面列出了一些常见问题和技巧。

- 首先也是最重要的：运行你的脚本时启用警告（-w 命令行选项）。
- 如果碰巧忘记了参数列表中的“,”——特别是用于输出时——你会招致一些非常奇怪的错误信息。
- 源代码最后一行的解析错误经常是由于某处遗忘了 end 关键字造成的，有时会是很前面的地方。
- 没有调用属性设置方法。在类定义中，`setter=`被解释为局部变量的赋值语句，而不是方法调用。如果使用 `self.setter=`形式则表示方法调用。

```
class Incorrect
 attr_accessor :one, :two
 def initialize
 one = 1 # incorrect - sets local variables
 self.two = 2
 end
end

obj = Incorrect.new
obj.one → nil
obj.two → 2
```

- 对象看起来没有被正确设置，通常是由于 `initialize` 方法拼写错误造成的。

```
class Incorrect
 attr_reader :answer
 def initialise # < < < spelling error
 @answer = 42
 end
end

ultimate = Incorrect.new
ultimate.answer → nil
```

如果写错实例变量名也会造成同样的错误。

```

class Incorrect
attr_reader :answer
def initialize
 @anwser = 42 #<< spelling error
end
end

ultimate = Incorrect.new
ultimate.answer → nil

```

- Block 的参数和局部变量处于相同的作用域中。当执行 block 时，如果 block 的参数存在已定义的同名局部变量，那么该变量可能被 block 修改。这可能是优点也可能不是。

```

c = "carbon"
i = "iodine"
elements = [c, i]
elements.each_with_index do |element, i|
 # do some chemistry
end
c → "carbon"
i → 1

```

- 注意优先级问题，特别是当使用 {} 而不是用 do/end 时。

```

def one(arg)
 if block_given?
 "block given to 'one' returns #{yield}"
 else
 arg
 end
end

def two
 if block_given?
 "block given to 'two' returns #{yield}"
 end
end

result1 = one two {
 "three"
}

result2 = one two do
 "three"
end

puts "With braces, result = #{result1}"
puts "With do/end, result = #{result2}"

```

输出结果：

```

With braces, result = block given to 'two' returns three
With do/end, result = block given to 'one' returns three

```

- 终端输出可能被缓存。这意味着你可能无法立即看到输出的信息。另外，如果你同时向 \$stdout 和 \$stderr 输出信息，那么输出的顺序与你所期望的可能有所不同。始终用非缓冲的 I/O 来处理调试用信息（设置 sync=true）。
- 若数字来路不正规，则它们实际上可能是字符串。从文件中读取的文本是 String，Ruby 不会自动把它转换成数字。调用 Integer 会进行转换（如果输入的不是合法的整数形式将会产生异常）。Perl 程序员常犯的一个错误是：

```
while line = gets
 num1, num2 = line.split(/,/)
 # ...
end
```

你可以重写为：

```
while line = gets
 num1, num2 = line.split(/,/)
 num1 = Integer(num1)
 num2 = Integer(num2)
 # ...
end
```

或者你可以使用 map 来一次性转换所有的字符串。

```
while line = gets
 num1, num2 = line.split(/,/).map {|val| Integer(val) }
 # ...
End
```

- 无意的别名——如果你使用一个对象作为散列表的关键字，确保它不会改变其 hash 值（或者改变后调用 Hash#rehash）。

```
arr = [1, 2]
hash = { arr => "value" }
hash[arr] → "value"
arr[0] = 99
hash[arr] → nil
hash.rehash → {[99, 2] => "value"}
hash[arr] → "value"
```

- 确保你所使用对象的类是你预想的，如有疑虑，则使用 puts my\_obj.class 查看。
- 确保方法名字以小写字母开头，而类名和常量名以大写字母开头。
- 如果方法调用结果与所期望的不符，确保将参数放到小括号内部。

- 确保方法参数列表的左括号紧跟在方法名字后，且中间没有空格。
- 使用 `irb` 和调试器。
- 使用 `Object#freeze`。如果你怀疑某段未知的代码会将变量设置为一个虚妄的值，则试着冻结该变量，那么当罪魁祸首试图改变此变量时就可以抓到它。

使得编写 Ruby 代码既容易又有趣的一项主要技术是：**渐增式地开发应用程序**。写几行代码，运行之，还可以使用 `Test::Unit` 写一些测试。接着写更多的代码，并运行它们。动态类型语言的一个主要好处是你不必等到代码完善之后才运行它们。

## 13.5 然而它太慢了 But It's Too Slow

Ruby 是一门解释性的高级语言，因此它可能没有低级语言例如 C 的运行速度快。下面小节中，我们将看看提高性能的一些基本技术；关于其他的指示信息，请查看索引中的 *Performance* 部分。

通常，运行速度慢的程序会有一两个性能瓶颈，它们浪费了执行时间。找到并改进这些瓶颈，将使得整个程序的效率立刻得到改善。难点在于如何找到它们。`Benchmark` 模块和 Ruby 的剖析器（profiler）可以帮助我们。

### 13.5.1 Benchmark

你可以使用第 657 页描述的 `Benchmark` 模块来对代码段计时。例如，我们可能想知道一个大的循环到底是使用循环内的局部变量速度快、还是使用外部已存在的变量速度快。下一页的图 13.2 展示了如何使用 `Benchmark` 来解决这个问题。

使用 `benchmark` 时需要很小心，因为垃圾回收常常会导致 Ruby 程序运行速度变慢。垃圾回收可能在程序运行的任何时候发生，因此 `benchmark` 可能会给你错误的结果：`benchmark` 显示一段代码运行很慢，而实际上速度变慢是由于执行该代码时垃圾回收恰好被触发所造成的。`Benchmark` 模块的 `bmbm` 方法会运行测试两次，一次作为预演，一次实际测量性能，力图将垃圾回收带来的影响降到最低。`Benchmark` 测试过程本身相对设计得比较优雅——它不会使程序运行慢太多。

```

require 'benchmark'
include Benchmark

LOOP_COUNT = 1_000_000

bm(12) do |test|
 test.report("normal:") do
 LOOP_COUNT.times do |x|
 y = x + 1
 end
 end
 test.report("predefine:") do
 x = y = 0
 LOOP_COUNT.times do |x|
 y = x + 1
 end
 end
end

```

输出结果：

|            | user     | system   | total    | real        |
|------------|----------|----------|----------|-------------|
| normal:    | 3.110000 | 0.000000 | 3.110000 | ( 4.954929) |
| predefine: | 2.560000 | 0.000000 | 2.560000 | ( 3.009354) |

图 13.2 使用 `benchmark` 比较变量访问代价

## 13.5.2 剖析器

### The Profiler

Ruby 自带一个代码剖析器（Profiler，其文档在第 717 页）。剖析器可以告诉你程序中的每个方法调用的次数以及 Ruby 运行这些方法的平均时间和累计时间。

你可以使用`-r profile` 命令行选项，也可以在代码中使用`require 'profile'` 将`profile` 添加到代码中。例如：

```

require 'profile'
count = 0
words = File.open("/usr/share/dict/words")
while word = words.gets
 word = word.chomp!
 if word.length == 12
 count += 1
 end
end
puts "#{count} twelve-character words"

```

第一次我们不使用剖析器，在大约含 235 000 个词的词典上运行这段代码，大约需要几秒钟才能完成。这看起来有点过分，所以我们添加`-r profile` 命令行选项并重新运行。最后我们得到的输出类似如下。

| 20460 twelve-character words |            |         |        |         |           |                          |
|------------------------------|------------|---------|--------|---------|-----------|--------------------------|
| %                            | cumulative | self    |        | self    | total     |                          |
| time                         | seconds    | seconds | calls  | ms/call | ms/call   | name                     |
| 7.76                         | 12.01      | 12.01   | 234937 | 0.05    | 0.05      | String#chomp!            |
| 7.75                         | 24.00      | 11.99   | 234938 | 0.05    | 0.05      | IO#gets                  |
| 7.71                         | 35.94      | 11.94   | 234937 | 0.05    | 0.05      | String#length            |
| 7.62                         | 47.74      | 11.80   | 234937 | 0.05    | 0.05      | Fixnum#=                 |
| 0.59                         | 48.66      | 0.92    | 20460  | 0.04    | 0.04      | Fixnum#+                 |
| 0.01                         | 48.68      | 0.02    | 1      | 20.00   | 20.00     | Profiler__.start_profile |
| 0.00                         | 48.68      | 0.00    | 1      | 0.00    | 0.00      | File#initialize          |
| 0.00                         | 48.68      | 0.00    | 1      | 0.00    | 0.00      | Fixnum#to_s              |
| 0.00                         | 48.68      | 0.00    | 1      | 0.00    | 0.00      | File#open                |
| 0.00                         | 48.68      | 0.00    | 1      | 0.00    | 0.00      | Kernel.puts              |
| 0.00                         | 48.68      | 0.00    | 2      | 0.00    | 0.00      | IO#write                 |
| 0.00                         | 48.68      | 0.00    | 1      | 0.00    | 154800.00 | #toplevel                |

首先需注意的是上面显示的耗时，比不使用剖析器运行程序要慢得多。剖析对性能影响很大，但是我们假设该影响是全局的，因此获得的相对的值仍然是有意义的。这个具体的程序明显地花费了很多时间在循环上，该循环大约执行了 235 000 次。如果我们能够降低循环体的运行代价或者完全不用循环，那么就可以大大地提高性能。实现第二种方法的一种方式是读取整个单词列表到一个长字符串中，然后使用模式去匹配并提取所有 12 个字母的词。

```
require 'profile'
words = File.read("/usr/share/dict/words")
count = words.scan(PATT= /^.....\n/).size
puts "#{count} twelve-character words"
```

我们的剖析结果显示的数据好多了（当我们不使用剖析时程序运行比以前快 5 倍）。

| 20460 twelve-character words |            |         |       |         |         |                          |
|------------------------------|------------|---------|-------|---------|---------|--------------------------|
| %                            | cumulative | self    |       | self    | total   |                          |
| time                         | seconds    | seconds | calls | ms/call | ms/call | name                     |
| 96.67                        | 0.29       | 0.29    | 1     | 290.00  | 290.00  | String#scan              |
| 6.67                         | 0.31       | 0.02    | 1     | 20.00   | 20.00   | Profiler__.start_profile |
| 0.00                         | 0.31       | 0.00    | 2     | 0.00    | 0.00    | Array#size               |
| 0.00                         | 0.31       | 0.00    | 1     | 0.00    | 0.00    | Kernel.puts              |
| 0.00                         | 0.31       | 0.00    | 1     | 0.00    | 110.00  | IO#write                 |
| 0.00                         | 0.31       | 0.00    | 1     | 0.00    | 0.00    | Fixnum#to_s              |
| 0.00                         | 0.31       | 0.00    | 1     | 0.00    | 300.00  | #toplevel                |
| 0.00                         | 0.31       | 0.00    | 1     | 0.00    | 0.00    | File#read                |

请记住，随后如果不使用剖析，检查一下代码——有时候剖析器引起的速度降低可能会掩盖其他问题。

Ruby 是个非常透明的、表现极佳的语言，但这并不会意味着程序员可以忽视一些基本的常识：创建不必要的对象，或者执行不必要的工作，以及代码过于臃肿都会造成程序速度下降，而不管编程语言是什么。

表 13.1 调试器命令

|                                        |                                                      |
|----------------------------------------|------------------------------------------------------|
| <code>b[reak] [file class:]line</code> | 在 <i>file</i> (默认为当前文件) 或者 <i>class</i> 的给定行设置断点     |
| <code>b[reak] [file class:]name</code> | 在 <i>file</i> 或者 <i>class</i> 的 <i>method</i> 中设置断点  |
| <code>b[reak]</code>                   | 显示断点和监视点                                             |
| <code>wat[ch] expr</code>              | 当表达式为真时中止                                            |
| <code>del[ete] [nnn]</code>            | 删除断点 <i>nnn</i> (默认为全部)                              |
| <code>cat[ch] exception</code>         | 当发生 <i>exception</i> 时异常停止运行                         |
| <code>cat[ch]</code>                   | 列出当前捕获的异常                                            |
| <code>tr[ace] (on off) [all]</code>    | 开关对当前线程或所有线程的执行跟踪                                    |
| <code>disp[lay] expr</code>            | 每当调试器获得控制时即显示表达式的值                                   |
| <code>disp[lay]</code>                 | 显示当前 <i>display</i>                                  |
| <code>undisp[lay] [nnn]</code>         | 删除 <i>display</i> (默认为所有)                            |
| <code>c[ont]</code>                    | 继续执行                                                 |
| <code>s[tep] nnn=1</code>              | 执行下面的 <i>nnn</i> 行, 遇到方法则进入执行                        |
| <code>n[ext] nnn=1</code>              | 执行下面的 <i>nnn</i> 行, 遇到方法则跨过执行                        |
| <code>fin[ish]</code>                  | 完成当前函数的执行                                            |
| <code>q[uit]</code>                    | 退出调试器                                                |
| <code>w[here]</code>                   | 显示当前的调用栈                                             |
| <code>f[rame]</code>                   | 与 <i>where</i> 同义                                    |
| <code>l[ist] [start-end]</code>        | 列出源代码的 <i>start</i> 到 <i>end</i> 行                   |
| <code>up nnn=1</code>                  | 在调用栈中上移 <i>nnn</i> 层                                 |
| <code>down nnn=1</code>                | 在调用栈中下移 <i>nnn</i> 层                                 |
| <code>v[ar] g [lobal]</code>           | 显示全局变量                                               |
| <code>v[ar] l [ocal]</code>            | 显示局部变量                                               |
| <code>v[ar] i [stance] obj</code>      | 显示 <i>obj</i> 的实例变量                                  |
| <code>v[ar] c[onst] Name</code>        | 显示给定类或者模块中的常量                                        |
| <code>m[ethod] i [nstance] obj</code>  | 显示 <i>obj</i> 的实例方法                                  |
| <code>m[ethod] Name</code>             | 显示给定类或者模块的实例方法                                       |
| <code>th[read] l[ist]</code>           | 列出所有线程                                               |
| <code>th[read] [c[ur]rent]]</code>     | 显示当前线程的状态                                            |
| <code>th[read] [c[ur]rent]] nnn</code> | 使线程 <i>nnn</i> 成为当前线程, 并停止它                          |
| <code>th[read] stop nnn</code>         | 使线程 <i>nnn</i> 成为当前线程, 并停止它                          |
| <code>th[read] resume nnn</code>       | 继续执行线程 <i>nnn</i>                                    |
| <code>th[read] [swITCH] nnn</code>     | 切换线程上下文为 <i>nnn</i>                                  |
| <code>[p] expr</code>                  | 在当前上下文中执行 <i>expr</i> . <i>expr</i> 时可能会包含变量赋值或者方法调用 |
| <code>h[elp]</code>                    | 显示命令概要                                               |
| <code>empty</code>                     | 重复上次执行的命令                                            |



---

## 第2部分 Ruby与其环境

---

## Part II Ruby in Its Setting

---



*Programming Ruby 中文版, 第2版*



# Ruby 和 Ruby 世界

## Ruby and Its World

我们的程序必须和这个庞大而糟糕的世界打交道，这是生活中一个不幸的事实。在本节我们看看 Ruby 如何与它的环境交互。Microsoft Windows 用户可能也想看一下第 267 页开头与平台相关的一些信息。

### 14.1 命令行参数

#### Command-Line Arguments

“In the beginning was the command line”（最初始于命令行）。<sup>1</sup>无论 Ruby 部署在什么系统上，不管它是超级高端计算图形工作站或者是嵌入式 PDA 设备，我们总是要运行 Ruby 解释器的，所以有机会将命令行参数传递给它。

Ruby 命令行由 3 部分组成：传递给 Ruby 解释器的选项，要运行的程序名称（可选）和程序的一组参数（可选）。

```
ruby [options] [--] [programfile] [arguments]
```

Ruby 本身的选项截止到命令行中首个以连字符（-）开始的单词、或者特别标志--（两个连字符）。

如果命令行上没有指定文件名，或者指定的文件名是单个连字符（-），Ruby 会从标准输入读入程序源码。

程序的参数跟在程序名称后面。例如，

```
% ruby -w - "Hello World"
```

Ruby 解释器会启用警告，从标准输入读入程序，同时把"Hello World"字符串作为程序的参数传递给程序。

<sup>1</sup> 这是 Neal Stephenson 一篇绝妙短文的标题（可从 <http://www.spack.org/index.cgi/InTheBeginningWasTheCommandLine> 在线获得）。

## 14.1.1 命令行选项

### Command-Line Options

#### -0[八进制]

标志 0（数字零）指定记录分隔字符（如果后面没有跟其他数字，则分隔字符是\0）。

-00 指示段落模式（paragraph mode）：记录会被两个连续的默认记录分隔字符分开。

-777 一次性读取整个文件（因为它是非法字符）。设置\$/全局变量。

-a 与-n 或者-p一同使用时为自动分割（autosplit）模式；等同于在每个循环迭代开始处执行\$F = \$\_.split语句。

#### -C directory

在执行程序之前将工作目录改为directory。

-c 只做语法检查，不执行程序。

#### --copyright

打印版权通告并退出。

#### -d, --debug

1.8 设置\$DEBUG 和\$VERBOSE 为 true。可以被程序使用来打开额外的信息跟踪。

#### -e 'command'

1.8 将 command 作为一行 Ruby 源码来执行。允许多个-e 选项，这些 command 被视为同一程序中的多行代码。如果使用-e 时省略了 programfile 参数，运行完这些-e 命令后解释器会停止。使用-e 选项运行的程序可以使用区间和正则表达式在条件语句中的老式行为——整数区间与当前输入行号进行比较，而正则表达式对\$\_ 变量进行匹配。

#### -F 模式

指定输入字段的分隔符（\$;），作为 split() 的默认分隔符（影响-a 选项）。

#### -h, --help

显示简短帮助信息。

#### -I 目录

指定前插到\$LOAD\_PATH (\$:) 的目录。可以出现多个-I 选项。每个-I 选项后面可以有多个目录，它们在类 Unix 系统中以冒号（:）分开，在 DOS/Windows 系统中以分号（;）分开。

#### -i [extension]

直接编辑 ARGV 文件。对于 ARGV 中的每个文件，写入到标准输出的任何内容都会保存为该文件的内容。如果提供了扩展名（extension），会生成文件的一个备份拷贝。

```
% ruby -pi.bak -e "gsub(/Perl/, 'Ruby')" *.txt
```

**-k *kcode***

指定编码集。这个选项主要用在日语处理上。*kcode* 可能是日语的一个编码：*e*, *E* 表示 EUC; *s*, *S* 表示 SJIS; *u*, *U* 表示 UTF-8; 或者 *a*, *A*, *n*, *N* 表示 ASCII。

- l** 启用自动行结束处理；将\$\*\*设置为\$/的值，并自动截断(chop)每个输入行。
- n** 认为程序包含在 while gets; ...; end 循环中。比如，简单的 grep 命令可以如下实现：

```
% ruby -n -e "print if /wombat/" *.txt
```

- p** 把程序代码放在 while gets; ...; print; end 循环中。

```
% ruby -p -e "$_.downcase!" *.txt
```

**-r *库***

在执行之前 require 指定的程序库。

- S** 使用 RUBYPATH 或 PATH 环境变量查找程序文件。
- s** 出现在程序文件之后，但是在任何文件名参数或者--之前的命令行开关会从 ARGV 中删除，同时把它设置给用选项开关名字命名的全局变量中。在下面例子中，它的效果就是把\$opt 变量设置为“electric”。

```
% ruby -s prog -opt=electric ./mydata
```

**-T [级别]**

设置安全级别，启用 tainting 检查（见第 397 页）。设置\$SAFE 变量。

**-v, --verbose**

- 1.8** 设置SVERBOSE 为 true，启用 verbose 模式。同时打印版本号。在 verbose 模式中，打印编译警告。如果命令行中没有指定程序文件，Ruby 退出。

**--version**

显示 Ruby 版本号并退出。

- 1.8** 启用 verbose 模式。与-v 选项不一样，如果命令行中没有指定程序文件，它从标准输入中读取程序源码。建议使用-w 选项来运行 Ruby 程序。

**-w 级别**

设置发出的警告级别。指定一个与-w 选项等效的级别，或者 2（没有指定时的默认值）——会给出额外的警告。如果级别为 1，运行在标准（默认的）警告级别。使用-w0，绝对不会给出任何警告（包括使用 Kernel.warn 发出的那些警告）。

**-X directory**

执行程序之前将工作目录改为 *directory*。与 **-C directory** 相同。

**-x [directory]**

删除`#!ruby` 行之前的文本，如果给出了 *directory*，则将工作目录改为 *directory*。

**-y, --yydebug**

在语法分析器中启用 yacc 调试（太多的信息）。

### 14.1.2 ARGV

程序文件后面出现的任何命令行参数在 Ruby 程序中可用，它们保存在全局数组 `ARGV` 中。比如，假设 `test.rb` 包含如下程序：

```
ARGV.each { |arg| p arg }
```

用下面的命令行调用它：

```
* ruby -w test.rb "Hello World" a1 1.6180
```

会产生如下输出：

```
"Hello World"
"a1"
"1.6180"
```

对于所有 C 程序员来说这里有一个经常出错的问题（gotcha）——`ARGV[0]` 是程序的第一个参数而不是程序名称。当前程序的名称位于全局变量 `$0` 中。注意到 `ARGV` 中所有的值都是字符串。

如果你的程序试图从标准输入（或者使用特别文件 `ARGF`，将在第 336 页中描述）中读取，`ARGV` 中的程序参数会被当作文件名称，然后 Ruby 会从这些文件中读取。如果程序把参数和文件名称混合在一起，那就确保从这些文件读取之前已清空了 `ARGV` 数组中的非文件名参数。

## 14.2 程序终止

### Program Termination

`Kernel#exit` 方法会终止程序，返回一个状态值给操作系统。但是与有些语言不一样，`exit` 没有立即终止程序。`Kernel#exit` 首先抛出可以捕获的 `SystemExit` 异常，然后执行若干个清理动作，其中包括运行任何已注册的 `at_exit` 方法和对象的终止方法（finalizers）。详细内容请参考第 521 页开始的 `Kernel#exit`。

## 14.3 环境变量

### Environment Variables

可以使用预定义的变量 `ENV` 来访问操作系统环境变量。它响应与 `Hash` 相同的方法。<sup>2</sup>

```
ENV['SHELL'] → "/bin/sh"
ENV['HOME'] → "/Users/dave"
ENV['USER'] → "dave"
ENV.keys.size → 32
ENV.keys[0..7] → ["MANPATH", "TERM_PROGRAM", "TERM", "SHELL",
 "SAVEHIST", "HISTSIZE", "MAKEFLAGS"]
```

某些环境变量的值在 Ruby 第一次运行时被读取。正如下页的表 14.1 所示，这些变量会改变解释器的行为。

#### 14.3.1 写入环境变量

##### Writing to Environment Variables

Ruby 程序可能写入 `ENV` 对象。在大多数系统上这会改变相应环境变量的值。当然，这种变化仅限于作出这种改变的进程，以及随后被它创建的那些子进程中。下面的代码说明了环境变量的继承。一个子进程改变一个环境变量，这种变化会被它随后产生的一个进程继承。但是，父进程看不到这种变化（这正好证明了：父母永远无法真正知道孩子在做什么）。

```
puts "In parent, term = #{ENV['TERM']}"
fork do
 puts "Start of child 1, term = #{ENV['TERM']}"
 ENV['TERM'] = "ansi"
 fork do
 puts "Start of child 2, term = #{ENV['TERM']}"
 end
 Process.wait
 puts "End of child 1, term = #{ENV['TERM']}"
end
process.wait
puts "Back in parent, term = #{ENV['TERM']}
```

输出结果：

```
In parent, term = xterm-color
Start of child 1, term = xterm-color
Start of child 2, term = ansi
End of child 1, term = ansi
Back in parent, term = xterm-color
```

<sup>2</sup> `ENV` 实际上不是散列表，但是如果需要的话，可以使用 `ENV#to_hash` 把它转换成 `Hash`。

表 14.1 Ruby 使用的环境变量

| 变量名称              | 描述                                                               |
|-------------------|------------------------------------------------------------------|
| DLN_LIBRARY_PATH  | 动态装入模块的查找路径                                                      |
| HOME              | 指向用户主目录。在文件名和目录名中展开~时使用它                                         |
| LOGDIR            | 如果没有设置\$HOME, 它是用户主目录的 fallback 指针。只被 Dir.chdir 使用               |
| 1.8. OPENSSL_CONF | 指定 OpenSSL 配置文件的位置                                               |
| RUBYLIB           | Ruby 程序的附加查找路径 (\$SAFE 必须为 0)                                    |
| RUBYLIB_PREFIX    | (只限 Windows) 通过将前缀添加到 RUBYLIB 的每个组成部分, 来改编 (mangle) RUBYLIB 查找路径 |
| RUBYOPT           | Ruby 的附加命令行选项; 在解析完实际的命令行选项之后被检查 (\$SAFE 必须为 0)                  |
| RUBYPATH          | 使用 -S 选项时 Ruby 程序的查找路径 (默认为 PATH)                                |
| RUBYSHELL         | 在 Windows 下创建执行进程时要用到的 shell; 如果没有设置, 则会检查 SHELL 或 COMSPEC 环境变量  |
| RUBY_TCL_DLL      | 覆盖 (override) TCL 共享库或 DLL 的默认名称                                 |
| RUBY_TK_DLL       | 覆盖 Tk 共享库或 DLL 的默认名称。使用它或者 RUBY_TCL_DLL 时都必须要设置这两个选项             |

## 14.4 从何处查找它的模块

### Where Ruby Finds Its Modules

使用 `require` 或 `load` 把程序库模块装入到程序中。有些模块是 Ruby 自带的, 有些可能是从 Ruby 应用归档 (Ruby Application Archive) 安装的, 有些可能是自己开发的。那么 Ruby 如何找到它们呢?

为特定机器编译 Ruby 时, 它预定义了一组标准目录去保存 Ruby 程序库。在哪里保存它们是和所讨论的机器相关的。可以使用类似下面这个命令行得到答案:

```
% ruby -e 'puts $:'
```

- 1.8. 在典型的 Linux 机器上, 你可能会发现下面的一些目录。注意 Ruby 1.8 中这些目录的顺序已经发生改变——体系结构相关的目录跟在与机器无关的目录后面。

```
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
```

`site_ruby` 目录是为了保存已经添加的那些模块和扩展。与体系结构相关的目录（这个例子中的 `i686-linux`）保存特定于这种机器的可执行文件和其他东西。所有这些目录会自动地被包括在 Ruby 的模块查找路径中。

有时这还不够。可能你正忙于一个以 Ruby 编写的大型项目上，你和同事已经编译了大量 Ruby 代码的程序库。你希望组里的每个人都可以访问所有这些代码。可以有几种选择来实现它。如果程序运行在安全级别 0 上（请参见从第 397 页开始的第 25 章），可以把环境变量 `RUBYLIB` 设置成一个包含一个或多个查找目录的列表。<sup>3</sup>如果程序没有 `setuid`，可以使用命令行参数`-I`来做同样的事情。

Ruby 变量`$:`是一个目录数组，用来查找已装入的文件。正如我们所见，这个变量被初始化为标准目录表，加上用 `RUBYLIB` 和`-I` 选项指定的所有附加目录。程序运行过程中随时可以将附加目录添加到这个数组中。

更有趣的是，一种新的程序库管理方法刚好及时出现，并在本书中介绍。第 215 页的第 17 章将要描述 RubyGems，这是一个支持网络的包管理系统。

## 14.5 编译环境

### Build Environment

为特定的体系结构编译 Ruby 时，所有用来编译它的相关设置（包括用来编译 Ruby 的机器的体系结构，编译器选项和源代码目录等等）都被写入到库文件 `rbconfig.rb` 中的 `Config` 模块。Ruby 安装之后，任何 Ruby 程序可以使用这个模块得到编译 Ruby 的细节。

```
require 'rbconfig'
include Config
CONFIG["host"] → "powerpc-apple-darwin7.7.0"
CONFIG["libdir"] → "/Users/dave/ruby1.8/lib"
```

扩展库可以使用这个配置文件在给定的体系结构上正确地编译和链接。详细信息请参见第 275 页开始的第 21 章和第 779 页开始的 `mkmf`。

---

<sup>3</sup> 目录之间的分隔符与平台相关。在 Windows 上分隔符是分号`(;)`；在 Unix 上是冒号`(:)`。



# 交互式 Ruby Shell

## Interactive Ruby Shell

回到第 164 页，我们介绍了 irb，一个帮助你交互式地进入 Ruby 程序并立时看到结果的 Ruby 模块。本章将深入有关使用和定制 irb 的更多细节。

### 15.1 命令行 Command Line

irb 是从命令行运行的。

```
irb [irb-options] [ruby_script] [program arguments]
```

irb 的命令行选项在下一页的表 15.1 中列出。通常你在运行 irb 时并不指定任何选项，但是如果你希望运行一个脚本并实时查看每一步的记录，你可以指定 Ruby 脚本的名称以及该脚本的选项。

开始执行之后，irb 显示一个提示符并等待输入。在下面的示例中，我们使用 irb 的默认提示符，它显示当前的绑定（binding）、缩进（嵌套）级别以及行号。

在提示符后，你可以键入 Ruby 代码。irb 包括一个 Ruby 的解析器，因此它知道语句尚未结束。这时，提示符会变成一个星号。你可以通过键入 `exit` 或 `quit`，或者通过输入一个文件结束符号（如果没有设置 `IGNORE_EOF` 模式）来退出 irb。

```
% irb
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0> 3 +
irb(main):003:0*> 4
=> 7
irb(main):004:0> quit
%
```

表 15.1 irb 命令行选项

| 选 项                              | 描 述                                                           |
|----------------------------------|---------------------------------------------------------------|
| --back-trace-limit <i>n</i>      | 显示最顶部和尾部的 <i>n</i> 项回溯信息。默认值为 16                              |
| -d                               | 设置 #DEBUG 为 true (同 ruby -d 一样)                               |
| -f                               | 禁止读取 ~/.irbrc                                                 |
| -I <i>path</i>                   | 指定 \$LOAD_PATH 目录                                             |
| --inf-ruby-mode                  | 设置 irb 以 Emacs 的 inf-ruby-mode 模式运行。更改提示符并废止 readline         |
| --inspect                        | 使用 Object#inspect 来格式化输出 (默认方式, 除数学模式外)                       |
| --irb_debug <i>n</i>             | 将内部调试级别设置为 <i>n</i> (只对 irb 的开发有用)                            |
| -m                               | 数学模式 (支持分数和矩阵)                                                |
| --noinspect                      | 不使用 inspect 进行输出                                              |
| --noprompt                       | 不显示提示符                                                        |
| --noreadline                     | 不要使用 Readline 扩展模块                                            |
| --prompt <i>prompt-mode</i>      | 切换提示符。预定义的模式包括 null, default, classic, simple, xmp 和 inf-ruby |
| --promtp-mode <i>prompt-mode</i> | 同 --prompt                                                    |
| -r <i>load-module</i>            | 同 ruby -r                                                     |
| --readline                       | 使用 readline 扩展模块                                              |
| --simple-promtp                  | 使用简单的提示符                                                      |
| --tracer                         | 显示命令执行的跟踪                                                     |
| -v, --version                    | 输出 irb 的版本号                                                   |

在 irb 的会话中, 你所做的工作在 irb 工作区 (workspace) 中累积起来。你设置的变量、定义的方法和创建的类, 都被记忆下来并可以被后续使用。

```

irb(main):001:0> def fib_up_to(n)
irb(main):002:1> f1, f2 = 1, 1
irb(main):003:1> while f1 <= n
irb(main):004:2> puts f1
irb(main):005:2> f1, f2 = f2, f1+f2
irb(main):006:2> end
irb(main):007:1> end
=> nil
irb(main):008:0> fib_up_to(4)
1
1
2
3
=> nil

```

注意返回值 `nil`。它们分别是定义方法和运行方法的结果。这个方法输出 Fibonacci (斐波纳契) 数列，然后返回 `nil`。

`irb` 的一个重要用途在于，体验你刚刚编写的代码。可能你想追踪一个 `bug`，或者你只是想把玩一下。如果你将程序加载到 `irb` 中，就可以创建它定义的类的实例并调用其方法。例如，文件 `code/fib_up_to.rb` 包括了下面的方法定义。

```
def fib_up_to(max)
 i1, i2 = 1, 1
 while i1 <= max
 yield i1
 i1, i2 = i2, i1+i2
 end
end
```

我们可以把它加载到 `irb` 中，并尝试调用这个方法。

```
% irb
irb(main):001:0> load 'code/fib_up_to.rb'
=> true
irb(main):002:0> result = []
=> []
irb(main):003:0> fib_up_to(20) {|val| result << val}
=> nil
irb(main):004:0> result
=> [1, 1, 2, 3, 5, 8, 13]
```

在这个示例中，我们使用 `load` 而不是 `require`，将文件包含到我们的会话中。我们这样做是出于实践的考量：`load` 允许我们多次加载同一个文件，这样如果我们发现了一个 `bug` 然后编辑文件，可以将它重新加载到 `irb` 会话中。

### 15.1.1 Tab 补齐 Tab Completion

如果你的 Ruby 安装支持 `readline`，可以使用 `irb` 的完成功能。在加载之后（我们稍后将介绍如何加载），当你在 `irb` 提示符后键入表达式时，“完成功能”改变了 **Tab** 键的含义。当你在词的中间按下 **Tab** 键时，`irb` 会查找所有在此刻有意义的候选完成。如果只有一个候选，`irb` 将会自动填充它。如果有多个选择，则 `irb` 最初什么也不做。不过，如果你再次按下 **Tab** 键，它会显示一个当前有效的完成列表。

例如，如果你在一个 `irb` 会话中，刚刚将一个字符串对象赋值给变量 `a`。

```
irb(main):002:0> a = "cat"
=> "cat"
```

你现在想尝试调用这个对象的 `String#reverse` 方法。你开始键入 `a.re` 然后按下 **Tab** 键两次。

```
irb(main):003:0> a.re TAB TAB
a.reject a.replace a.respond_to? a.reverse a.reverse!
```

irb 列出了该对象所支持的名字以“re”开头的全部方法。我们看到了想要的那个，reverse，并输入它名字的下一个字符，v，接着按下 Tab 键。

```
irb(main):003:0> a.rev TAB
irb(main):003:0> a.reverse
=> "tac"
irb(main):004:0>
```

irb 通过尽可能地扩展名称来响应 Tab 键，于是完成了单词 reverse。如果你此时键入 Tab 两次，它会向我们展示当前的选择，reverse 和 reverse!。不过，既然 reverse 是我们想要的那一个，继而按下回车键，然后这行代码就被执行了。

Tab 补齐并不仅限于内建的名称。如果你在 irb 中定义了一个类，当你想要调用它的一个方法时，tab 补齐也有效。

```
irb(main):004:0> class Test
irb(main):005:1> def my_method
irb(main):006:2> end
irb(main):007:1> end
=> nil
irb(main):008:0> t = Test.new
=> #<Test:0x35b724>
irb(main):009:0> t.my TAB
irb(main):009:0> t.my_method
```

Tab 补齐是以一个扩展库来实现的，即 irb/completion。当调用 irb 时，你可以从命令行加载它。

```
% irb -r irb/completion
```

你还可以在 irb 运行时加载补齐库。

```
irb(main):001:0> require 'irb/completion'
=> true
```

如果你总是使用 tab 补齐，最便捷的方式可能是将 require 命令放到你的 .irbrc 文件中。

```
require 'irb/completion'
```

## 15.1.2 子会话

### Subsessions

irb 支持多个、并发的会话。当前会话只有一个；其他的在被激活前处于休眠状态。在 irb 内输入 irb 命令会创建一个子会话，输入 jobs 命令列出所有的会话，输入 fg 则激活一个特定的休眠会话。

下面的示例还演示了 -r 命令行选项，它会在 irb 启动之前加载指定的文件。

```
% irb -r code/fib_up_to.rb
irb(main):001:0> result = []
=> []
irb(main):002:0> fib_up_to(10) {|val| result << val }
=> nil
irb(main):003:0> result
=> [1, 1, 2, 3, 5, 8]
irb(main):004:0> # Create a nested irb session
irb(main):005:0*> irb
irb#1(main):001:0> result = %w{ cat dog horse }
=> ["cat", "dog", "horse"]
irb#1(main):002:0> result.map {|val| val.upcase }
=> ["CAT", "DOG", "HORSE"]
irb#1(main):003:0> jobs
=> #0->irb on main (#<Thread:0x331740>: stop)
#1->irb#1 on main (#<Thread:0x341694>: running)
irb#1(main):004:0> fg 0
irb(main):006:0> result
=> [1, 1, 2, 3, 5, 8]
irb(main):007:0> fg 1
irb#1(main):005:0> result
=> ["cat", "dog", "horse"]
```

### 15.1.3 子会话与绑定

#### Subsessions and Bindings

如果当你创建子会话时指定了一个对象，它会成为绑定中 `self` 的值。这是体验对象的一种便捷的方式。在下面的示例中，我们使用字符串“wombat”作为默认对象，创建了一个子会话。没有指明接收者的方法会被该对象执行。

```
% irb
irb(main):001:0> self
=> main
irb(main):002:0> irb "wombat"
irb#1(wombat):001:0> self
=> "wombat"
irb#1(wombat):002:0> upcase
=> "WOMBAT"
irb#1(wombat):003:0> size
=> 6
irb#1(wombat):004:0> gsub(/[^aeiou]/, '*')
=> "w*mb*t"
irb#1(wombat):005:0> irb_exit
irb(main):003:0> self
=> main
irb(main):004:0> upcase
NameError: undefined local variable or method `upcase' for main:Object
```

## 15.2 配置

### Configuration

`irb` 是高度可配置的。你可以在命令行或者从初始化文件中，甚至在 `irb` 内时，设置配置选项。

#### 15.2.1 初始化文件

##### Initialization File

`irb` 使用一个初始化文件，你可以在其中设置常用的选项、或者执行任何所需的 Ruby 语句。当 `irb` 开始运行时，它会尝试加载一个初始化文件，按下面的顺序查找：`~/.irbrc`、`.irbrc`、`irb.rc` 和 `$irbrc`。

在初始化文件中，你可以运行任意的 Ruby 代码。你还可以设置配置项的值。配置变量的列表从第 192 页开始给出——在初始化文件中使用的值是符号（Symbol，以一个冒号开头）。你使用这些符号来设置散列表 `IRB.conf` 中的值。例如，让 `SIMPLE` 成为所有 `irb` 会话的默认提示符，你可以在初始化文件中设置如下。

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
```

配置 `irb` 一个有趣的技法是，你可以将 `IRB.conf[:IRB_RC]` 设置为一个 `Proc` 对象。只要 `irb` 的上下文有所变动，这个 `Proc` 就会被调用，并且以该上下文的配置作为参数。你可以使用这项功能，基于上下文动态地改变配置。例如，下面的 `.irbrc` 文件对提示符进行了设置，虽然主提示符只显示 `irb` 的层次，但是连接提示符和结果仍会排列好。

```
IRB.conf[:IRB_RC] = proc do |conf|
 leader = " " * conf.irb_name.length
 conf.prompt_i = "#{conf.irb_name} --> "
 conf.prompt_s = leader + ' \-\- '
 conf.prompt_c = leader + ' \-\+ '
 conf.return_format = leader + " ==> %s\n\n"
 puts "Welcome!"
end
```

使用该 `.irbrc` 文件的 `irb` 会话，看起来像如下所示。

```
% irb
Welcome!
irb --> 1 + 2
==> 3

irb --> 2 +
\-\+ 6
==> 8
```

## 15.2.2 扩展 irb

### Extending irb

因为你在 irb 中键入的内容被当作 Ruby 代码来解释，你可以通过定义新的顶层（top-level）方法来扩展 irb。例如，你可能希望能够在 irb 中查询某个类或方法的参考文档。如果你在 .irbrc 文件中添加下面的代码，将会添加一个叫做 ri 的方法，它使用传入的参数调用外部的 ri 命令（在 Windows 下你需要使用一个 ri.bat 命令）。

```
def ri(*names)
 system(%{ri #{names.map{|name| name.to_s}.join(" ")}})
end
```

当你下一次启动 irb 时，你将能够使用这个方法来得到参考文档。

```
irb(main):001:0> ri Proc
----- Class: Proc
 Proc objects are blocks of code that have been bound to a set of
 local variables. Once bound, the code may be called in different
 contexts and still access those variables.
 and so on...

irb(main):002:0> ri :strftime
----- Time#strftime
 time.strftime(string) => string

 Formats time according to the directives in the given format
 string. Any text not listed as a directive will be passed through
 to the output string.

 Format meaning:
 %a - The abbreviated weekday name ('`Sun``')
 %A - The full weekday name ('`Sunday``')
 %b - The abbreviated month name ('`Jan``')
 %B - The full month name ('`January``')
 %c - The preferred local date and time representation
 %d - Day of the month (01..31)
 and so on...

irb(main):003:0> ri "String.each"
----- String#each
 str.each(separator=$/) |substr| block => str
 str.each_line(separator=$/) |substr| block => str

 Splits str using the supplied parameter as the record separator
 ($/ by default), passing each substring in turn to the supplied
 block. If a zero-length record separator is supplied, the string
 is split on \n characters, except that multiple successive
 newlines are appended together.

 print "Example one\n"
 "hello\nworld".each |s| p s
 and so on...
```

### 15.2.3 交互式配置

#### Interactive Configuration

多数配置值是你在运行 `irb` 时可设置的。本页开始的列表展示了形如 `conf.xxx` 的这些值。例如，要将你的提示符改为 `SIMPLE` 并改回 `DEFAULT`，你可以使用下面的步骤。

```
irb(main):001:0> 1 +
irb(main):002:0*> 2
=> 3
irb(main):003:0> conf.prompt_mode = :SIMPLE
=> :SIMPLE
>> 1 +
?> 2
=> 3
```

### 15.2.4 irb 的配置选项

#### irb Configuration Options

在接下来的描述中，形如`:xxx` 的标记表示在初始化文件中 `IRB.conf` 散列表内所使用的 `key`（键），而 `conf.xxx` 表示可以在交互中设置的值。描述最后部分方括号中的值，是该选项的默认值。

##### `:AUTO_INDENT / conf.auto_indent_mode`

如果为真，`irb` 将会在你输入嵌套结构时进行缩进。`[false]`

##### `:BACK_TRACE_LIMIT / conf.back_trace_limit`

显示回溯的起始和结束 `n` 行。`[16]`

##### `:CONTEXT_MODE`

对新工作区所使用的绑定：0→在顶层的 `proc`，1→绑定于一个加载的、匿名文件中，2→每个线程绑定于一个加载的文件，3→绑定于一个顶层的函数。`[3]`

##### `:DEBUG_LEVEL/conf.debug_level`

将内部的调试级别设置为 `n`。只有当你调试 `irb` 的词法分析器时才有用。`[0]`

##### `:IGNORE_EOF/conf.ignore_eof`

指定当输入接收到文件结尾时 `irb` 的行为。如果为真，它会被忽略；否则，`irb` 将退出。`[false]`

##### `:IGNORE_SIGINT/conf.ignore_sigint`

如果为假，则`^C` (`Ctrl+c`) 将会退出 `irb`。如果为真，在输入中途按下`^C` 将会取消输入并返回到顶层；当执行时，`^C` 将会取消当前的操作。`[true]`

##### `:INSPECT_MODE/conf.inspect_mode`

指定如何显示值：`true` 则意味着使用 `inspect`，`false` 则使用 `to_s`，`nil` 在非数学模式下使用 `inspect`，而在数学模式下使用 `to_s`。`[nil]`

**:IRB\_RC**

可以被设置为一个 proc 对象，当一个 irb 会话或子会话开始时被调用。[nil]

**conf.last\_value**

irb 输出的最后一个值。[...]

**:LOAD\_MODULES/conf.load\_modules**

通过 -r 命令行选项加载模块的列表。[[]]

**:MODULE\_MODE/conf.math\_mode**

如果为真，irb 运行所加载的 mathn 库（参见第 692 页）。[false]

**conf.prompt\_c**

连接语句的提示符（例如，在 “if” 之后立即显示）。[依赖于配置]

**conf.prompt\_i**

标准的、顶层的提示符。[依赖于配置]

**:PROMPT\_MODE/conf.prompt\_mode**

提示符的显示风格。[:DEFAULT]

**conf.prompt\_s**

连续字符串的提示符。[依赖于配置]

**:PROMPT**

参见第 195 页的配置提示符[(…)]

**:RC/conf.rc**

如果为 false，不要加载初始化文件。[true]

**conf.return\_format**

用来显示交互中输入表达式结果的格式。[依赖于配置]

**:SINGLE\_IRB**

如果为 true，嵌套的 irb 会话会共享相同的绑定；否则会根据 :CONTEXT\_MODE 的值创建一个新的绑定。[nil]

**conf.thread**

当前执行 Thread 对象的只读引用。[当前线程]

**:USE\_LOADER/conf.use\_loader**

指定了是否使用 irb 自己的文件读取方法来进行 load/require。[false]

**:USE\_READLINE/conf.use\_readline**

如有存在，irb 使用 readline 库（参见第 723 页），除非该选项被设置为 false，此时 readline 永远不会使用；如果为 nil，readline 将不会在 inf-ruby-mode 中使用。  
[依赖于配置]

**:USE\_TRACER/conf.use\_tracer**

如果为 true，则跟踪语句的执行。[false]

**:VERBOSE/conf.verbose**

理论上，在该选项为 true 时会打开附加的跟踪；但实践中并没有输出什么额外的跟踪结果。[true]

## 15.3 命令

### Commands

在 irb 提示符下，你可以输入任何有效的 Ruby 表达式并查看结果。你还可以使用下面的任何命令来控制 irb 会话。

**exit, quit, irb\_exit, irb\_quit**

退出 irb 的会话或子会话。如果你使用 cb 更改了绑定（参见下面的命令），则从绑定模式退出。

**conf, context, irb\_context**

显示当前的配置。通过调用 conf 的方法来修改配置。第 192 页开始的列表展示了可用的 conf 设置。例如，要将默认的提示符设置为服从性的含义，可以使用：

```
irb(main):001:0> conf.prompt_i = "Yes, Master? "
=> "Yes, Master? "
Yes, Master? 1 + 2
```

**cb, irb\_change\_binding <obj>**

创建并进入一个新的绑定，它有自己的局部变量范围。如果提供了 obj，它将作为新绑定中的 self。

**irb <obj>**

开始一个 irb 子会话。如果提供了 obj，它会被用作 self。

**jobs, irb\_jobs**

列出 irb 的子会话。

**fg n, irb\_fg n**

切换到指定的 irb 子会话。n 可能是下列之一：irb 的子会话号，线程的 ID，一个 irb 对象，或者当子会话开始时的 self 对象。

**kill n, irb\_kill n**

杀死一个 irb 子会话。n 的可能值同 irb\_fg 中描述的一样。

### 15.3.1 配置提示符

#### Configuring the Prompt

你有很多灵活性来配置 `irb` 所使用的提示符。提示符的集合保存在提示符的散列表 `IRB.conf[:PROMPT]` 中。

例如，要建立一个名为“`MY_PROMPT`”的新提示符，你可以输入下面的语句（直接在 `irb` 提示符之后，或在 `.irbrc` 文件中）。

```
IRB.conf[:PROMPT][:MY_PROMPT] = { # name of prompt mode
 :PROMPT_I => '-->',
 :PROMPT_S => '--''',
 :PROMPT_C => '--+',
 :RETURN => " ==>%s\n"
}
```

在你定义了提示符之后，你必须通知 `irb` 来使用它。从命令行，你可以使用`--prompt` 选项（注意提示符模式的名称是如何自动地转换为大写形式、同时横线改为下画线的）。

```
%irb --prompt my-prompt
```

如果你想要在未来所有的 `irb` 会话中都使用这个提示符，你可以将它作为一个 `.irbrc` 文件中的配置值。

```
IRB.conf[:PROMPT_MODE] = :MY_PROMPT
```

符号 `PROMPT_I`、`PROMPT_S` 和 `PROMPT_C` 指定了每个提示符字符串的格式。在格式化字符串中，特定的“%”序列会被展开。

---

#### 标 志 描 述

---

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| <code>%N</code>  | 当前命令                                                                                   |
| <code>%m</code>  | 主对象 ( <code>self</code> ) <code>to_s</code> 的结果                                        |
| <code>%M</code>  | 主对象 ( <code>self</code> ) <code>inspect</code> 的结果                                     |
| <code>%l</code>  | 分隔符类型。在多行的字符串中， <code>%l</code> 显示用来开始字符串的分隔符类型，这样你知道如何结束它。分隔符是“`”、“`”、“/”或`中的一个       |
| <code>%ni</code> | 缩进级别。可选的数字 <code>n</code> 用来作为 <code>printf</code> 的宽度规格，例如 <code>printf("%nd")</code> |
| <code>%nn</code> | 当前的行号 ( <code>n</code> 用作缩进级别)                                                         |
| <code>%%</code>  | 字面的百分号                                                                                 |

---

例如，默认的提示符模式定义如下。

```
IRB.conf[:PROMPT_MODE][:DEFAULT] = {
 :PROMPT_I => "%N(%m):%03n:%i>",
 :PROMPT_S => "%N(%m):%03n:%i%l",
 :PROMPT_C => "%N(%m):%03n:%i*",
 :RETURN => "%s\n"
}
```

## 15.4 限制

### Restrictions

因为 irb 的工作方式，它和标准的 Ruby 解释器稍有不兼容。问题在于局部变量的判定。

通常，Ruby 查看赋值语句来判定某个名字是否为一个变量——如果一个名字没有被赋值过，那么 Ruby 假定该名字是一个方法调用。

```
eval "var = 0"
var
```

输出结果：

```
prog.rb:2: undefined local variable or method `var'
for main:Object (NameError)
```

在这种情况下，赋值出现在一个字符串中，因此 Ruby 并不会对它有所考虑。

而另一方面，irb 在我们输入时执行语句。

```
irb(main):001:0> eval "var = 0"
0
irb(main):002:0> var
0
```

在 irb 中，赋值在第二行之前被执行，因此 var 被正确地标识为一个局部变量。

如果你需要更紧密地匹对 Ruby 的行为，你可以将这些语句放在 begin/end 对儿中。

```
irb(main):001:0> begin
irb(main):002:1* eval "var = 0"
irb(main):003:1> var
irb(main):004:1> end
NameError: undefined local variable or method `var'
(irb):3:in `irb_binding'
```

## 15.5 rtags 与 xmp

### rtags and xmp

假如还嫌 irb 不够复杂，让我们再加点料。随着 irb 的主程序，irb 套件还包括了其他一些内容。在本节中，我们将考察其中的两个：rtags 和 xmp。

#### rtags

rtags 是一个用来创建 Emacs 或 vi 编辑器所使用的 TAGS 文件的工具。

```
rtags [-vi] [files] ...
```

默认的，`rtags` 创建一个适合 Emacs 使用的 TAGS 文件（参见 `etags.el`）。`-vi` 选项生成适合 vi 使用的 TAGS 文件。

`rtags` 的安装方式和 `irb` 相同（也就是说，你需要把 `irb` 安装到库路径中，并创建一个由 `irb/rtags.rb` 到 `bin/rtags` 的链接）。

### xmp

`irb` 的 `xmp` 是一个“示例打印程序”——它是一个非常好的打印程序，在运行时显示每个表达式的值（很像我们为格式化本书示例所编写的脚本）。在 `irb` 归档中也有另一个独立的 `xmp`。

`xmp` 可以按如下使用。

```
require 'irb/xmp'

xmp <<END
artist = "Doc Severinsen"
artist.upcase
END
```

输出结果：

```
artist = "Doc Severinsen"
=> "Doc Severinsen"
artist.upcase
=> "DOC SEVERINSEN"
```

或者，`xmp` 可以作为一个对象实例。以这种方式使用，`xmp` 对象维护调用之间的上下文。

```
require 'irb/xmp'

x = XMP.new
x.puts 'artist = "Louis Prima"'
x.puts 'artist.upcase'
```

输出结果：

```
artist = "Louis Prima"
=> "Louis Prima"
artist.upcase
=> "LOUIS PRIMA"
```

你可以用任何形式明确地提供一个绑定；否则 `xmp` 使用调用者的环境。

```
xmp code_string, abinding
XMP.new(abinding)
```

注意 `xmp` 不支持多线程。



# 文档化 Ruby

## Documenting Ruby

1.8. 从 1.8 版本开始，Ruby 内建了 RDoc 工具，它可以提取并格式化嵌套在 Ruby 源代码文件中的文档。该工具被用来为 Ruby 内建的类和模块生成文档。不断增长的库和扩展也使用这种方式来写文档。

RDoc 做两件事情。第一，它分析 Ruby 和 C 源文件以寻找需要文档化的信息<sup>1</sup>。第二，它把得到的这些信息转换成某种方便阅读的格式。RDoc 可以生成两种格式的输出：HTML 和 ri。下一页的图 16.1 显示了在浏览器窗口中以 HTML 格式输出的 RDoc。这是将一个没有额外文档的 Ruby 源文件传递给 RDoc 的结果——RDoc 非常出色地生成了一些有意义的信息。如果我们的源代码包括注释，Rdoc 会使用它们来丰富产生的文档。通常来说，元素前面的注释被用来生成该元素的文档，正如第 201 页的图 16.2 所示。

RDoc 也可以生成能被 ri 命令行工具识别的文档。例如，如果用这种方式来为图 16.2 中的代码生成 RDoc 文档，我们则可以像 202 页中图 16.3 所示那样，使用 ri 命令访问该文档。新的 Ruby 发行版中内建的类和模块（以及某些库）都是用这种方法来建立文档的。203 页的图 16.4 显示了 ri Proc 的结果。

### 16.1 向 Ruby 代码中添加 RDoc Adding RDoc to Ruby Code

RDoc 解析 Ruby 源代码以提取其主要元素（类、模块、方法、属性等等）。通过在文件中的元素前添加注释块，你可以为该元素添加额外的文档。

<sup>1</sup> RDoc 也能为 Fortran 77 程序生成文档。

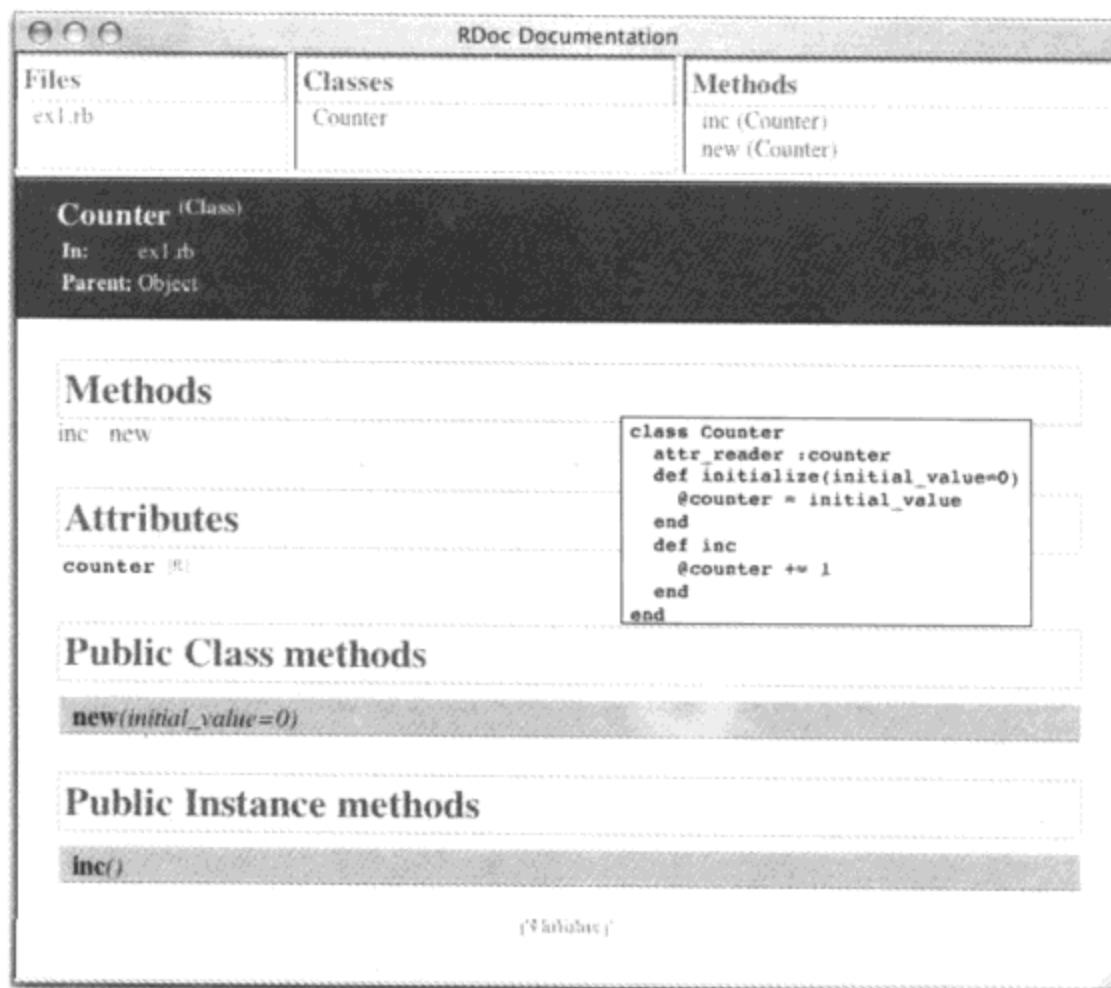


图 16.1 浏览类 Counter 的 RDoc 输出

此图显示了浏览器窗口中的 RDoc 输出。图上的方块显示了产生此文档输出的源代码。尽管源代码中没有任何内部文档，RDoc 仍旧从中提取出了令人感兴趣的信息。屏幕顶端有 3 个面板（pane），分别显示了已生成文档的文件、类和方法。对类 Counter 而言，RDoc 显示了其属性和方法（包括方法的原型特征——signature）。并且如果我们点击一个方法签名，RDoc 会弹出一个包含其源代码的窗口。

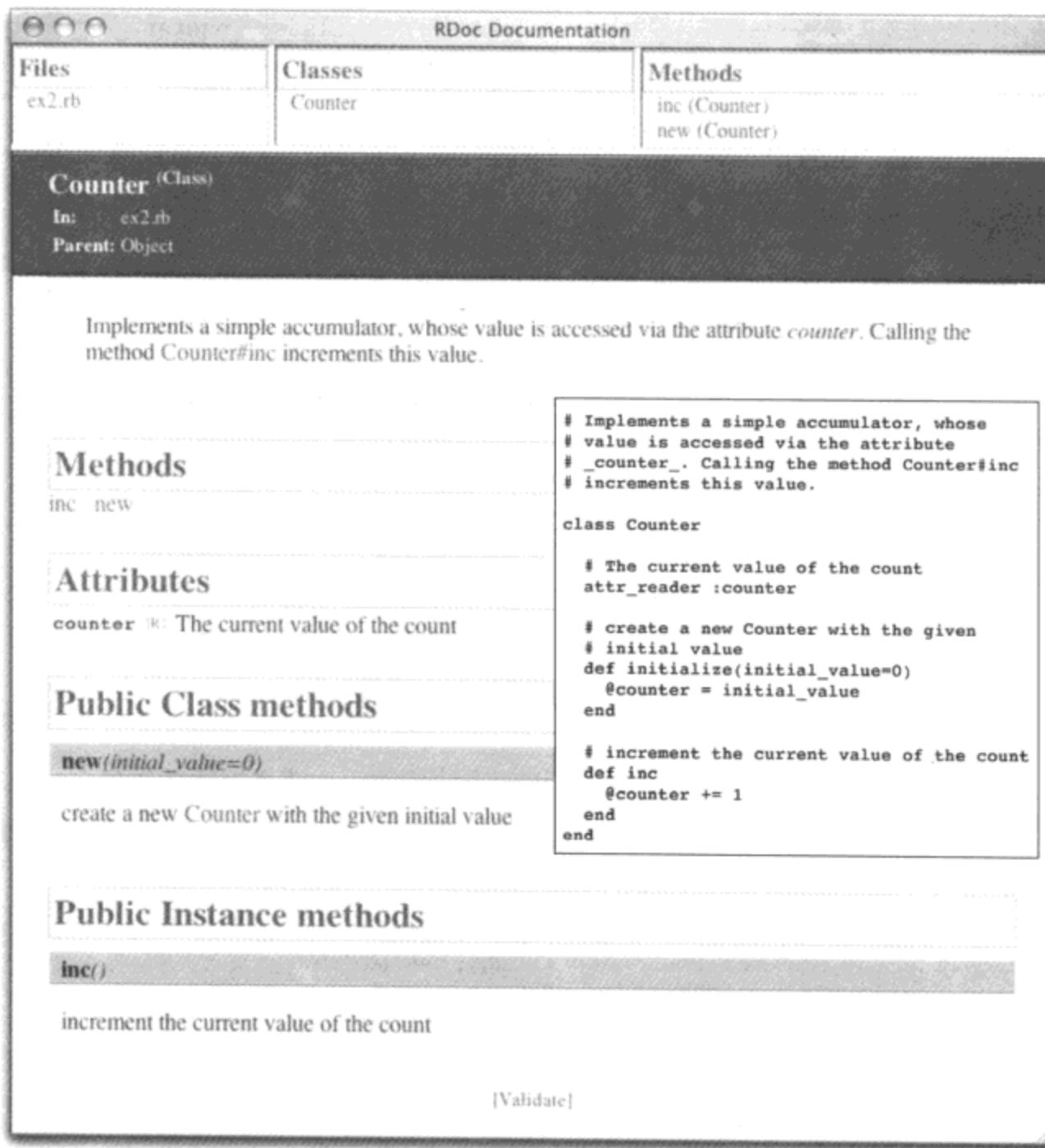


图 16.2 浏览带注释的源代码的 RDoc 输出

注意每个元素前面的注释是如何输出在 RDoc 中并重新格式化生成 HTML 的。不太明显的一点是，RDoc 会检测出注释中超链接的可能性：在类级别的注释中，对 `Counter#inc` 的引用会转化成指向该方法描述的超链接，而在 `new` 方法的注释中，对类 `Counter` 的引用将链接回到类文档。这是 RDoc 的关键特性：它被设计为对 Ruby 源文件无侵入性，并通过当生成结果时施行尽可能明智的处理来达成这一点。

```
% ri Counter

----- Class: Counter
Implements a simple accumulator, whose value is
accessed via the attribute counter. Calling the
method Counter#inc increments this value.

Class methods:
new

Instance methods:
inc

Attributes:
counter

% ri Counter.inc

----- Counter#inc
inc()

increment the current value of the count
```

图 16.3 使用 ri 来读取文档

注释块可以很自然地书写，或者在连续的注释行前加 #，或者把注释放到 =begin ... =end 块中。如果你使用第二种形式，=begin 行必须带 rdoc 标签，以和其他风格的文档相区分。

```
=begin rdoc
Calculate the minimal-cost path though the graph
using Debrinkski's algorithm, with optimized
inverse pruning of isolated leaf nodes.
=end
def calculate_path
 .
 .
end
```

在文档注释中，段落中的行具有相同的左边距。缩进超过此空白部分的文本被逐字地格式化。

有些文本可以被标记起来。`_word_`, `*word*` 和 `+word+` 将分别以斜体、粗体和印刷体来显示其中的词。如果想处理多个词或者含有非词字符的文本，你可以用 `<em>` `multiple words</em>`, `<b>more words</b>` 和 `<tt>yet more words</tt>`。把反斜线加入到内嵌的标记前将使它不被解释。

```
% ri Proc
----- Class: Proc
Proc objects are blocks of code that have been
bound to a set of local variables. Once bound,
the code may be called in different contexts and
still access those variables.

def gen_times(factor)
 return Proc.new { |n| n*factor }
end

times3 = gen_times(3)
times5 = gen_times(5)

times3.call(12) #=> 36
times5.call(5) #=> 25
times3.call(times5.call(4)) #=> 60

Class methods:
new

Instance methods:
==, [], arity, binding, call, clone, eql?, hash,
to_proc, to_s
```

图 16.4 由 Rdoc/ri 生成的 Proc 类的文档

如果 RDoc 发现注释行以 `#--` 开头，那么它将停止处理注释。这可以用来区分外部注释和内部注释，或者禁止将注释关联到方法、类或者模块上。以 `#++` 开头的行重新打开文档处理功能。

```
Extract the age and calculate the
date of birth.
#--
FIXME: fails if the birthday falls on
February 29th, or if the person
was born before epoch and the installed
Ruby doesn't support negative time_t
#++
The DOB is returned as a Time object.
#--
But should probably change to use Date.

def get_dob(person)
 ...
end
```

### 16.1.1 超链接

#### Hyperlinks

以下画线或减号作前缀的类名、源代码文件以及任何方法名都会自动把注释文本转化成指向它们描述的链接。

以 `http:`, `mailto:`, `ftp:` 和 `www:` 开始的网络链接也能被识别。引用外部图像文件的 HTTP URL 将被转换成内嵌的 `<IMG ...>` 标记。以 `link:` 开头的链接被认为是引用本地文件，且其路径是相对于当前操作路径（即存储输出文件的路径）的相对路径。

超链接也可以是 `label[url]` 形式，在这种情况下 `label` 是被显示的文本，而 `url` 是被链接的目标。如果 `label` 中含有多个词，则可以将它们放到花括号中：`{two words}[url]`。

### 16.1.2 列表

#### Lists

列表是缩进的段落，且带有

- 一个 \* 或者 - (用于无序列表)。
- 后跟一个点的数字，用于编号列表。
- 后跟一个点的大写或小写字母，用于字母列表。

例如，你可以使用下面的注释产生类似上面文本的文档。

```
Lists are typed as indented paragraphs with
* a * or -(for bullet lists),
* a digit followed by a period for
numbered lists,
* an upper or lower case letter followed
by a period for alpha lists.
```

注意一个列表项中后续的行是如何缩进以和第一行对齐的。

带标签的列表（labeled list，有时也称为描述列表）使用方括号将标记括起。

```
[cat] small domestic animal
[+cat+] command to copy standard input
to standard output
```

带标签的列表也可通过在标记后加两个冒号来生成。这将得到表格形式的输出，所以描述都会排列整齐。

```
cat::: small domestic animal
+cat+::: command to copy standard input
to standard output
```

对两种类型的标记列表而言，如果主体文本的开头和标记在同一行，那么开头的位置决定主体文本剩余部分的缩进量。主体文本也可以从标记的下一行开始，并从标记的开头缩进。这常用在标记比较长的情况下。下面所示都是合法的标记列表项。

```
<tt>--output</tt> <i>name [, name]</i>::
specify the name of one or more output files. If multiple
files are present, the first is used as the index.

<tt>--quiet:</tt>:: do not output the names, sizes, byte counts,
index areas, or bit ratios of units as
they are processed.
```

### 16.1.3 标题

#### Headings

以等于号开头的行表示标题。等于号个数越多，则标题的级别越高。

```
= Level One Heading
== Level Two Heading
and so on...
```

3个或更多的连字符表示水平线。

```
and so it goes...

The next section...
```

### 16.1.4 文档修饰符

#### Documentation Modifiers

方法的参数列表将被提取出来和方法的描述一块显示。如果一个方法调用 `yield`，那么传递给 `yield` 的参数也将被显示出来。例如，考虑如下代码：

```
def fred
 ...
 yield line, address
```

这将产生如下的文档：

```
fred() { |line, address| ... }
```

当方法定义在同一行时，使用含有 `:yields: ...` 的注释，可以改变这种默认行为。

```
def fred # :yields: index, position
 ...
 yield line, address
```

将获得如下文档：

```
fred() { |index, position| ... }
```

`:yields:` 是一个文档修饰符的例子。它们紧跟在所修饰的文档元素后面，并与文档元素在同一行上。

其他修饰符包括：

**:nodocz: [all]**

不要在文档中包含此元素。对于类和模块，直接处于其中的方法、别名、常量和属性也不会出现在文档中。但是默认情况下在其中定义的模块或者类仍将会被文档化。可以使用 `all` 修饰符关闭此默认行为。例如在下面的代码中，只有类 `SM::Input` 将被文档化。

```
module SM #:nodoc:
 class Input
 end
end
module Markup #:nodoc: all
 class Output
 end
end
```

**:doc:**

强制文档化方法或者属性，否则将不被文档化。如果你想把一种特殊的私有方法放到文档中的话，这会很有用。

**:notnew:**

(仅适用于 `initialize` 实例方法。) 通常 RDoc 认为一个类的 `#initialize` 方法的文档和参数实际对应的是 `new` 方法，并会为该类构造一个 `new` 方法。`:notnew:` 修饰符将阻止 RDoc 这么做。记住 `#initialize` 是被保护的方法，所以你不会看到它的文档，除非使用 `-a` 命令行选项。

## 其他指令

注释块还可以包含其他指令。

**:call-seq: lines ...**

用此指令当生成文档时，持续到下一空白注释行的文本被认为是对调用序列（略过对方法参数列表的解析）的解释。即使某一行以`#`开头也被认为是空白行。对此指令而言，开头的冒号可以省略。

**:include: filename**

添加指定文件的内容到此处。RDoc 将从 `-include` 命令行选项列出的目录列表中搜索文件，默认情况下在当前目录搜索。文件的内容将移动到和`:include:` 指令的：具有相同量的缩进。

**:title: text**

设置文档的标题。等价于 `--title` 命令行参数（命令行参数会覆盖源代码中的任意 `:title:` 指令）。

**:main: name**

等价于 `--main` 命令行参数，设置此文档的初始显示页面。

**:stopdoc: / :startdoc:**

停止和开始添加新文档元素到当前容器。例如，如果类含有一些你不想为之添加文档的常量，那么可以把 `:stopdoc:` 放到第一个常量前，而把 `:startdoc:` 放到最后一个常量前。如果直到容器结束都没有 `:startdoc:`，那么整个类或者模块将不被文档化。

**:enddoc:**

在当前词汇级别，不再为后面任何东西做文档。

下一页的图 16.5 显示了一个用 RDoc 文档化源文件的更完整的例子。

## 16.2 向 C 扩展中添加 RDoc

### Adding RDoc to C Extensions

RDoc 也能理解很多用 C 为 Ruby 写扩展的惯例。

大多数 C 扩展有一个 `Init_Classname` 的方法。RDoc 认为这是类的定义——`Init_` 方法前的任何 C 注释都被当作该类的文档。

`Init_` 函数通常被用来关联 C 函数和 Ruby 方法名。例如，`Cipher` 扩展可能会定义一个名为 `salt=` 的 Ruby 方法，该方法是通过调用如下函数来由 C 函数 `salt_set` 实现的。

```
rb_define_method(cCipher, "salt=", salt_set, 1);
```

RDoc 会解析此调用，添加 `salt=` 方法到类文档中。然后搜索函数 `salt_set` 的 C 源代码。如果该函数前面有注释，那么 RDoc 会用该注释作为 `salt=` 方法的文档。

除了在函数的注释中写普通文档之外，这种基本模式不需要任何其他额外负担即可工作。然而 RDoc 不能识别相应的 Ruby 方法的调用序列。在这个例子中，RDoc 仅会显

```

This module encapsulates functionality related to the
generation of Fibonacci sequences.
#-
Copyright (c) 2004 Dave Thomas, The Pragmatic Programmers, LLC.
Licensed under the same terms as Ruby. No warranty is provided.
module Fibonacci

 # Calculate the first _count_ Fibonacci numbers, starting with 1,1.
 #
 # :call-seq:
 # Fibonacci.sequence(count) -> array
 # Fibonacci.sequence(count) {|val| ... } -> nil
 #
 # If a block is given, supply successive values to the block and
 # return +nil+, otherwise return all values as an array.
 def Fibonacci.sequence(count, &block)
 result, block = setup_optional_block(block)
 generate do |val|
 break if count <= 0
 count -= 1
 block[val]
 end
 result
 end

 # Calculate the Fibonacci numbers up to and including _max_.
 #
 # :call-seq:
 # Fibonacci.upto(count) -> array
 # Fibonacci.upto(count) {|val| ... } -> nil
 #
 # If a block is given, supply successive values to the
 # block and return +nil+, otherwise return all values as an array.
 def Fibonacci.upto(max, &block)
 result, block = setup_optional_block(block)
 generate do |val|
 break if val > max
 block[val]
 end
 result
 end

 private

 # Yield a sequence of Fibonacci numbers to a block.
 def Fibonacci.generate
 f1, f2 = 1, 1
 loop do
 yield f1
 f1, f2 = f2, f1+f2
 end
 end

 # If a block parameter is given, use it, otherwise accumulate into an
 # array. Return the result value and the block to use.
 def Fibonacci.setup_optional_block(block)
 if block.nil?
 [result = [], lambda {|val| result << val }]
 else
 [nil, block]
 end
 end
end

```

图 16.5 用 RDoc 文档化的 Ruby 源文件

示一个名为“arg1”（有点无意义）的参数。你可以通过在函数注释中使用 `call-seq` 指令解决此问题。`call-seq` 后面的行（直到空行）被用来文档化方法的调用序列。

```
/*
 * call-seq:
 * cipher.salt = number
 * cipher.salt = "string"
 *
 * Sets the salt of this cipher to either a binary +number+ or
 * bits in +string+.
 */
static VALUE
salt_set(cipher, salt)
...
```

如果方法返回一个有意义的值，则应当用 `call-seq` 中的字符`->`后面的串来注释。

```
/*
 * call-seq:
 * cipher.keylen -> Fixnum or nil
 */
```

尽管在为简单的扩展寻找类和方法注释时 RDoc 能以启发式的方式工作良好，但对较复杂的扩展实现它却不能总是做到运行正确。在这种情况下，你可以使用 `Document-class:` 和 `Document-method:` 指令来指明一个 C 注释分别和给定的类或者方法相关联。这两个修饰符以被注释的 Ruby 类或方法的名字作为参数。

```
/*
 * Document-method: reset
 *
 * Clear the current buffer and prepare to add new
 * cipher text. Any accumulated output cipher text
 * is also cleared.
 */
```

最后，我们也可以在 `Init`\_方法中将 Ruby 方法与处于不同 C 源文件中的 C 函数相关联。但是没有你的帮助，RDoc 也无法找到这个函数。你可以通过为 `rb_define_method` 方法调用添加注释来指明包含函数定义的文件。下面的例子告诉 RDoc 到名为 `md5.c` 的文件中去找与 `md5` 方法相对应的函数（以及相关文档）。

```
rb_define_method(cCipher, "md5", gen_md5, -1); /* in md5.c */
```

下一页的图 16.6 显示了一个用 RDoc 来文档化的 C 源文件。注意，为了节省空间，几个内部函数的函数体被省略了。

```

#include "ruby.h"
#include "cdjukebox.h"

static VALUE cCDPlayer;
static void cd_free(void *p) { ... }
static VALUE cd_alloc(VALUE klass) { ... }
static void progress(CDJukebox *rec, int percent) { ... }

/* call-seq:
 * CDPlayer.new(unit) -> new_cd_player
 *
 * Assign the newly created CDPlayer to a particular unit
 */
static VALUE cd_initialize(VALUE self, VALUE unit) {
 int unit_id;
 CDJukebox *jb;

 Data_Get_Struct(self, CDJukebox, jb);

 unit_id = NUM2INT(unit);
 assign_jukebox(jb, unit_id);

 return self;
}

/* call-seq:
 * player.seek(int_disc, int_track) -> nil
 * player.seek(int_disc, int_track) {|percent| } -> nil
 *
 * Seek to a given part of the track, invoking the block
 * with the percent complete as we go.
 */
static VALUE
cd_seek(VALUE self, VALUE disc, VALUE track) {
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);

 jukebox_seek(jb, NUM2INT(disc), NUM2INT(track), progress);
 return Qnil;
}

/* call-seq:
 * player.seek_time -> Float
 *
 * Return the average seek time for this unit (in seconds)
 */
static VALUE
cd_seek_time(VALUE self)
{
 double tm;
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);
 tm = get_avg_seek_time(jb);
 return rb_float_new(tm);
}

/* Interface to the Spinzelot[http://spinzelot.cd]
 * CD Player library.
 */

void Init_CDPlayer() {
 cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
 rb_define_alloc_func(cCDPlayer, cd_alloc);
 rb_define_method(cCDPlayer, "initialize", cd_initialize, 1);
 rb_define_method(cCDPlayer, "seek", cd_seek, 2);
 rb_define_method(cCDPlayer, "seek_time", cd_seek_time, 0);
}

```

图 16.6 用 RDoc 文档化的 C 源文件

## 16.3 运行 RDoc

### Running RDoc

你可以用下面的命令来运行 RDoc:

```
% rdoc [options] [filenames ...]
```

输入 `rdoc --help` 可以得到最新的选项概述。

在生成任何输出之前，将先对文件进行解析，并收集它们所含有的信息。这样就可以处理所有文件之间的交叉索引。如果文件是一个目录，那么会遍历其含有的所有文件。如果没有提供文件名，那么会处理当前目录（及其子目录）的所有 Ruby 文件。

典型的用法是为 Ruby 源代码包（例如 RDoc 自身）生成文档。

```
% rdoc
```

本命令会为当前目录及其子目录内的所有 Ruby 源文件和 C 源文件生成 HTML 文档。这些文档会被存储在 `doc/` 子目录下的文档树中。

RDoc 使用文件后缀名来确定如何处理每个文件。名字以 `.rb` 和 `.rbw` 结尾的文件被认为是 Ruby 源文件。以 `.c` 结尾的文件被当作 C 文件来处理。其他所有文件被认为只含有标记（有或者没有`#`开头的注释标记）。如果目录名被传递给 RDoc，那么只递归扫描其中的 C 和 Ruby 源文件。假若想包含像 `README` 这样的非源代码文件到文档分析流程中，则需要在命令行中显式给出它们的文件名。

在写 Ruby 库时，常常会有一部分源代码文件实现了公共接口，但是大多数文件是仅为内部使用的，文档读者对它们并不关心。在这种情况下，可以在你的项目目录下创建一个 `.document` 文件。如果 RDoc 进入含有 `.document` 文件的目录，那么 RDoc 仅处理该文件内列出的文件。该文件的每一行可以是文件名、目录名或者是一个通配符（文件系统的“glob”模式）。例如，如果想包括所有名字以 `main` 开头的 Ruby 文件和 `constants.rb` 文件，你可以含有如下内容的 `.document` 文件：

```
main*.rb
constants.rb
```

有些项目通常在顶层的 `README` 文件中查找文档。以 RDoc 格式编写此文件，并用 `:include:` 指令包含它到主类中是一种很方便的做法。

### 16.3.1 为 ri 创建文档

#### Create Documentation for ri

RDoc 也可以创建能被 ri 命令显示的文档。

当你运行 ri 时，默认情况下它会到 3 个地方去搜索文档<sup>2</sup>：

1. *system* 文档目录：其中含有 Ruby 发布版自带的文档，它们是在安装 Ruby 时创建的。
2. *site* 目录：其中含有本地添加的系统范围内的文档。
3. 用户的目录：存储在每个用户自己的主目录中。

你可以在如下位置找到这 3 个目录。

- \$datadir/ri/<ver>/system/...
- \$datadir/ri/<ver>/site/...
- ~/.rdoc/....

\$datadir 变量是安装 Ruby 时配置的数据目录，你可以使用下面的命令找到本地的 *datadir*：

```
ruby -r rbconfig -e 'p Config:::CONFIG["datadir"]'
```

为了添加文档到 ri，你需要告诉 RDoc 使用哪个输出目录。如果自己用的话，最方便的方式是使用 --ri 选项。

```
% rdoc --ri file1.rb file2.rb
```

如果想安装文档到系统范围内，则使用 --ri-site 选项。

```
% rdoc --ri-site file1.rb file2.rb
```

--ri-system 选项通常仅被用来为 Ruby 内建的类和标准库安装文档。你可以从 Ruby 源代码发行版（不是从安装的库本身）中重新创建这些文档。

```
% cd <ruby source base>/lib
% rdoc --ri-system
```

### 16.4 显示程序用法信息

#### Displaying Program Usage

大多数的命令行程序用某种工具来显示它们的正确用法：如果收到不合法的参数，它们会报告一个简短的错误信息，后跟其实际选项的纲要。如果你使用 RDoc，你很可能

---

<sup>2</sup> 你可以使用 RDoc 的 -op 选项后用 ri 的 --oc-dir 选项覆盖默认目录位置。

```

== Synopsis
#
Display the current date and time, optionally honoring
a format string.
#
== Usage
#
ruby showtime.rb [-h | --help] [-f | --fmt fmtstring]
#
fmtstring::
A +strftime+ format string controlling the
display of the date and time. If omitted,
use "%Y-%M-%d %H:%m"
#
== Author
Dave Thomas, The Pragmatic Programmers, LLC
#
== Copyright
Copyright (c) 2004 The Pragmatic Programmers.
Licensed under the same terms as Ruby.

require 'optparse'
require 'rdoc/usage'

fmt = "%Y-%M-%d %H:%m"
opts = OptionParser.new
opts.on("-h", "--help") { RDoc::usage }
opts.on("-f", "--fmt FMTSTRING") {|str| fmt = str }
opts.parse(ARGV) rescue RDoc::usage('usage')

puts Time.now.strftime(fmt)

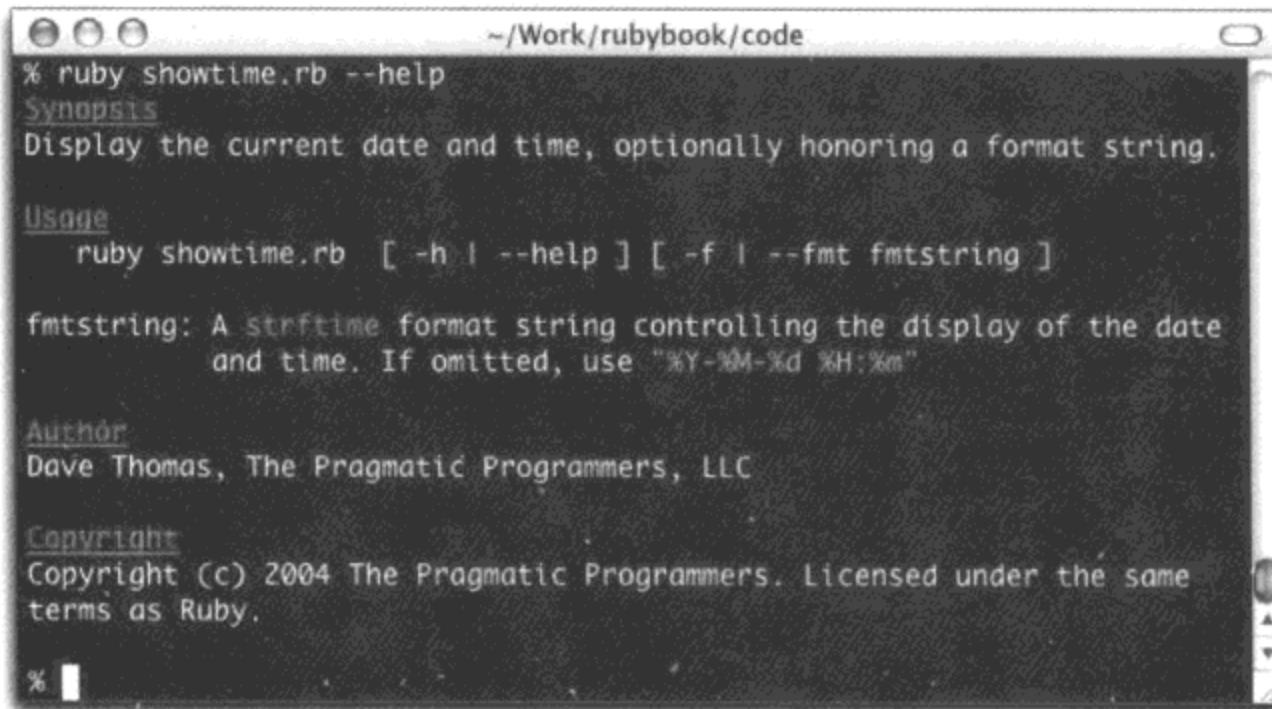
```

图 16.7 使用 RDoc::usage 的范例程序

已经在主程序开始处的 RDoc 注释中描述了如何使用本程序。你可以用 RDoc::usage 直接从注释命令中将这些信息提取出来显示给用户，而不用在程序的某个地方使用 puts 重复这些信息。

你可以传递一些字符串参数给 RDoc::usage。如果有参数，则它只从注释块中提取以参数命名的小节（该小节以和参数相同的字符串做标题，忽略大小写）。如果没有字符串参数，那么 RDoc::usage 将显示整个文档。此外，RDoc::usage 在显示完用法信息后会立即终止程序。如果 RDoc::usage 的第一个参数是一个整数，那么它将作为程序的返回码（否则它会返回 0 错误码）。如果你不想在显示用法信息后退出程序，则调用 RDoc::usage\_no\_exit。

图 16.7 展示了一个显示时间的小程序。如果用户请求帮助信息，则它使用 RDoc::usage 来显示整个注释块；如果用户给的参数不合法，则它只显示用法信息。图 16.8 展示了 --help 选项所生成的输出。



The screenshot shows a terminal window with the following content:

```
~/Work/rubybook/code
% ruby showtime.rb --help
Synopsis
Display the current date and time, optionally honoring a format string.

Usage
ruby showtime.rb [-h | --help] [-f | --fmt fmtstring]

fmtstring: A strftime format string controlling the display of the date
and time. If omitted, use "%Y-%M-%d %H:%m"

Author
Dave Thomas, The Pragmatic Programmers, LLC

Copyright
Copyright (c) 2004 The Pragmatic Programmers. Licensed under the same
terms as Ruby.

%
```

图 16.8 由范例程序产生的帮助信息

RDoc::usage 使用 RI 环境变量，该变量可以设置显示宽度和输出风格。图 16.8 的输出是在 RI 变量设置为“-f ansi”的情况下产生的。如果你是在黑白色的书中看这个表，可能看起来不太明显，但小节标题、代码字体和加重字体使用了 ANSI 转义序列定义的不同的颜色。

Chad Fowler 是 Ruby 社区的一位领军人物。  
他是 Ruby Central Inc. 董事会成员。  
他是 RubyConf 的一个组织者。同时  
他是 RubyGems 的开发者之一。所有  
这一切使他成为唯一胜任编写这一章的人。

## 第 17 章

# 用 RubyGems 进行包的管理

---

## Package Management with RubyGems

RubyGems<sup>1</sup>是一个库和程序的标准化打包以及安装框架，它使定位、安装、升级和卸载 Ruby 包变得容易。它给使用者和开发者提供了 4 种主要功能。

1. 一种标准化的包格式。
2. 使用一个中央仓库来存储这种格式的包。
3. 安装和管理同时安装的多个版本的相同库。
4. 最终用户工具以查询、安装、卸载以及别的方式来操作这些包。

RubyGems 出现之前，安装新的程序库会涉及：搜索互联网、下载程序库包并试图安装它——结果可能只是发现它的依赖性尚未满足。如果程序库是使用 RubyGems 打包的，那么，现在可以简单地让 RubyGems 去安装它（以及它所有的依赖包）。每件事情 RubyGems 都为你完成了。

在 RubyGems 的世界里，开发人员把他们的程序和库包裹（bundle）到一个 *gem* 文件中。这些文件遵循标准化的格式，同时 RubyGems 系统提供命令行工具去操作这些 *gem* 文件，这个工具被恰当地命名为 *gem*。

本章我们会看到如何：

1. 在电脑上安装 RubyGems。
2. 使用 RubyGems 去安装别的程序和库。
3. 编写自己的 *gem*。

---

<sup>1</sup> 译注：书翻译时的最新版本是 0.9.0。

## 17.1 安装 RubyGems

### Installing RubyGems

为了使用 RubyGems，首先需要从 <http://rubygems.rubyforge.org> 项目主页去下载和安装 RubyGems 系统。解开下载的包之后，可以使用其中的安装脚本来进行安装。

```
% cd rubygems-0.7.0
% ruby install.rb
```

根据你的操作系统的具体情况，你可能需要将文件写入到 Ruby 的 `site_ruby/` 和 `bin/` 目录的适当权限。

测试 RubyGems 是否已成功安装的最佳办法，碰巧也是我们要学习的最重要的命令。

```
% gem help
RubyGems is a sophisticated package manager for Ruby. This is
a basic help message containing pointers to more information.

Usage:
gem -h/ --help
gem -v/ --version
gem command [arguments...] [options...]

Examples:
gem install rake
gem list --local
gem build package.gemspec
gem help install

Further help:
gem help commands list all 'gem' commands
gem help examples show some examples of usage
gem help <COMMAND> show help on COMMAND(e.g. 'gem help install')

Further information:
http://rubygems.rubyforge.org
```

因为 RubyGems 帮助信息非常地丰富，所以本节我们不会详细讨论每个可用的 RubyGems 命令和选项。

## 17.2 安装程序 Gems

### Installing Application Gems

让我们开始使用 RubyGems 来安装以 Ruby 编写的程序。Jim Weirich 的 Rake 程序 (<http://rake.rubyforge.org>) 享有第一个以 `gem` 方式发布的应用的殊荣。不仅如此，它还是一个手边常用的好工具，它类似于 Make 和 Ant 的一个构建 (build) 工具。实际上，你甚至可以使用 Rake 去构建 gems！

用 RubyGems 定位和安装 Rake 是简单的。

```
% gem install -r rake
Attempting remote installation of 'Rake'
Successfully installed rake, version 0.4.3
% rake --version
rake, version 0.4.3.
```

RubyGems 下载 Rake 包并安装它。因为 Rake 是一个程序，RubyGems 下载了 Rake 库和命令行程序 rake。

使用子命令可以控制 gem 程序，每个子命令有它自己的选项和帮助信息。在这个例子中，使用了 install 命令和 -r 选项，-r 选项告诉 gem 远程操作。（许多 RubyGems 操作可以在本地或远程执行。比如，可以使用 query 命令去显示所有远程可用的 gems 文件或显示已经安装在本地的 gems 列表。正因为如此，子命令接受 -r 和 -l 选项去指定一个操作是在远程还是本地执行。）

因为某些原因——可能由于一个潜在的兼容问题，你需要一个旧版本的 Rake，可以使用 RubyGems 的版本需求操作符（version requirement operator）去指定选择版本的条件。

```
% gem install -r rake -v"< 0.4.3"
Attempting remote installation of 'rake'
Successfully installed rake, version 0.4.2
% rake --version
rake, version 0.4.2
```

下一页的表 17.1 列出了版本需求操作符。前面例子中的 -v 选项要求 Rake 的最高版本低于 0.4.3。

如果使用 RubyGems 安装相同程序的不同版本时，这里有一个微妙的问题。即使 RubyGems 保存了程序各个版本的库文件，但是它没有提供不同版本的命令行程序。其结果是，程序的每次安装实际上都覆盖了先前的安装。

在安装的时候也可以添加 -t 选项到 RubyGems 的 install 命令，这会让 RubyGems 运行这个 gem 的测试套件（test suite 如果已经创建了一个）。如果测试失败了，安装程序会提示保留或丢弃这个 gem。这是一种好的方式，我们可以获得更多的一点自信以确信刚下载的这个 gem 会按其作者设想的那样在系统上工作。

```
% gem install SomePoorlyTestedProgram -t
Attempting local installation of 'SomePoorlyTestedProgram-1.0.1'
Successfully installed SomePoorlyTestedProgram, version 1.0.1
23 tests, 22 assertions, 0 failures, 1 errors...keep Gem? [Y/n] n
Successfully uninstalled SomePoorlyTestedProgram version 1.0.1
```

如果作出默认选择并保留安装好的 gem，那么我们可以检查 gem 来试图确定测试失败的原因。

表 17.1 版本操作符

| 说 明 |                                                                                                |
|-----|------------------------------------------------------------------------------------------------|
| 操作符 | 描 述                                                                                            |
| =   | 精确版本匹配。Major（主），minor（次）和patch（补丁）级别必须相同                                                       |
| !=  | 任何不是指定版本的版本                                                                                    |
| >   | 任何比指定版本高的版本（甚至在patch级别）                                                                        |
| <   | 任何比指定版本低的版本                                                                                    |
| >=  | 任何比指定版本高或相等的版本                                                                                 |
| <=  | 任何比指定版本低或相等的版本                                                                                 |
| ->  | “Boxed”（装箱式）版本操作符。版本必须高于或等于指定版本，同时应该低于指定版本在其次版本号加1后得到的结果。 <sup>*</sup> 这是为了避免不同次版本之间 API 的不兼容性 |

## 17.3 安装和使用 Gem 库

### Installing and Using Gem Libraries

使用 RubyGems 安装完整的程序，是一种熟悉并开始学习使用 `gem` 命令的好方式。但在大多数情况下，是因为程序中需要一些 Ruby 库，你才使用 RubyGems 来安装它们的。既然 RubyGems 允许安装和管理相同库的多个版本，当代码需要这些库时，你还需要做一些新颖的、只有 RubyGems 才特有的事情。

也许你的妈妈要求你去编写程序，帮助她维护和发布日记。你决定以 HTML 格式发布她的日记，但是担心妈妈可能理解不了 HTML 标记的所有细节。所以你倾向于从众多 Ruby 可用的优秀模板化包中选择一种。经过一番调查之后，你发现 Michael Granger 的 BlueCloth 使用起来非常简单，于是决定使用它。<sup>2</sup>

首先需要找到和安装 BlueCloth gem。

```
%gem query -rn Blue
*** REMOTE GEMS ***
BlueCloth (0.0.4, 0.0.3, 0.0.2)
 BlueCloth is a Ruby implementation of Markdown, a text-to-HTML
 conversion tool for web writers. Markdown allows you to write using
 an easy-to-read, easy-to-write plain text format, then convert it
 to structurally valid XHTML (or HTML).
```

<sup>2</sup> 译注：例如  $1.0.1 > 1.0$ ，但是  $1.1.2 > 1.0$  不成立。

`query` 命令使用`-n` 选项在中央 `gem` 仓库中查找任何名字匹配正则表达式/`Blue/`的 `gem` 文件。结果显示存在 3 个可用版本的 `BlueCloth` (0.0.4, 0.0.3 和 0.0.2)。因为想安装最近的版本，所以没有在 `install` 命令中给出一个明确的版本；默认情况下最近的版本会被下载。

```
%gem install -r BlueCloth
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
```

### 17.3.1 生成 API 文档

#### Generating API Documentation

这是第一次使用 `BlueCloth`，你还不知道如何使用它。需要从它的一些 API 文档开始熟悉。好在添加`--rdoc` 选项到 `install` 命令后，RubyGems 会为安装的 `gem` 生成 RDoc 文档。关于 RDoc 更多的信息，见第 199 页的第 16 章。

```
%gem install -r BlueCloth --rdoc
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
Installing RDoc documentation for BlueCloth-0.0.4...
WARNING: Generating RDoc on .gem that may not have RDoc.
 bluecloth.rb: cc.....
Generating HTML...
```

那么在生成了所有这些有用的 HTML 文档后，如何阅读它们呢？至少有两种办法。困难一点的办法（尽管它真的没有这么困难）是进入 RubyGems 的文档目录，直接浏览这些文档。与 RubyGems 大多数事情一样，每个 `gem` 的文档存储在一个集中受保护的位置，这是 RubyGems 所特有的。这个位置可能会随着系统以及所明确选择安装那些 `gem` 的位置而改变。查找文档最可靠的办法是执行 `gem` 命令，询问 RubyGems 主目录放在哪里。比如：

```
% gem environment gemdir
/usr/local/lib/ruby/gems/1.8
```

RubyGems 把生成的文档存储到这个目录的 `doc/` 子目录中，在这个例子里它是 `/usr/local/lib/ruby/gems/1.8/doc`。可以打开文件 `index.html` 并阅读这个文档。如果发现自己经常使用这个路径，可以创建快捷方式。这里是 Mac OS X 机器上的一种方式。

```
% gemdoc=`gem environment gemdir`/doc
% ls $gemdoc
BlueCloth-0.0.4
% open $gemdoc/BlueCloth-0.0.4/rdoc/index.html
```

为了节省时间，可以在登录 shell 的 profile 或 rc 文件里声明 \$gemdoc。

第二种（更容易些）阅读 gems RDoc 文档的办法是使用 RubyGems 自带的 gem\_server 实用工具。为了启动 gem\_server，简单地键入：

```
% gem_server
[2004-07-18 11:28:51] INFO WEBrick 1.3.1
[2004-07-18 11:28:51] INFO ruby 1.8.2 (2004-06-29) [i386-mswin32]
[2004-07-18 11:28:51] INFO WEBrick::HTTPServer#start: port=8808
```

gem\_server 启动一个 Web 服务器，运行在启动它的电脑上。默认情况下，它运行在端口 8808 上，从默认的 RubyGems 安装目录为那些 gems 以及它们的文档提供服务。端口号和 gem 目录可以分别通过使用命令行的 -p 和 -d 选项去重新设置。

一旦启动了 gem\_server 程序，如果运行在本地机器上，用浏览器浏览 <http://localhost:8808> 就可以访问那些已安装的 gem 文档。在这里会看到一个列表，它包含已安装的 gems，它们的描述以及到 RDoc 文档的链接。

### 17.3.2 开始编码

Let's Code

安装了 BlueCloth 并知道如何使用它后，现在你可以准备去编写一些代码了。使用 RubyGems 下载了 BlueCloth 库后，也可以把这个库装入到我们的程序中。在 RubyGems 出现之前，可以写成：

```
require 'bluecloth'
```

通过使用 RubyGems 后，就可以利用它的打包和版本支持。用 require\_gem 替换 require 便可以实现它。

```
require 'rubygems'
require_gem 'BlueCloth', ">=0.0.4"
doc = BlueCloth::new <<MARKUP
 This is some sample [text][1]. Just learning to use [BlueCloth][1].
 Just a simple test.
 [1]: http://ruby-lang.org
MARKUP
puts doc.to_html
```

输出结果：

```
<p>This is some sample text. Just
learning to use BlueCloth.
Just a simple text.</p>
```

头两行是 RubyGems 特有的代码。第一行装入 RubyGems 的核心库，我们需要它来使用那些已安装的 gems。

```
require 'rubygems'
```

第二行是神奇发生的地方。

```
require_gem 'BlueCloth', '>=0.0.4'
```

这一行把 `BlueCloth` gem 添加到 Ruby 的 `$LOAD_PATH` 中，同时使用 `require` 装入 gem 作者指定要自动装入的所有库。让我们稍稍换种说法再重述一遍。

每个 gem 被认为是一组资源。它可能包含一个或一百个库文件。在老式的非 RubyGems 库中，所有这些文件会被拷贝到 Ruby 程序库树中的某个共享位置，这个位置包括在 Ruby 预定义的装载路径中。

RubyGems 不是以这种方式工作的。相反，它在自包含的目录树中保存每个版本的 gem。gems 并没有被存储到标准的 Ruby 库目录中。其结果是，RubyGems 需要一些技巧才能让你得到那些文件。它是通过将 gem 的目录树添加到 Ruby 的装载路径来实现的。从正在运行的程序内容看，这个效果是一样的：`require` 工作得很好。从外部看，RubyGems 对要装入到程序中的库文件提供了更好的控制。

在 `BlueCloth` 例子中，这个模板化代码作为文件 `bluecloth.rb` 被分发；这就是 `require_gem` 要装入的那个文件。`require_gem` 的第二个参数可选，它指定一个版本需求。在这个例子中，为了使用这个代码，已经指定了必须安装 `BlueCloth` 版本 0.0.4 或其更高的版本。如果需要版本 0.0.5 或更高版本，这个程序会失败，因为已安装的版本太低而不能满足程序的需求。

```
require 'rubygems'
require_gem 'BlueCloth', '>= 0.0.5'
```

输出结果：

```
/usr/local/lib/ruby/site_ruby/rubygems.rb:30:
 in 'require_gem': (LoadError)
RubyGem version error: BlueCloth(0.0.4 not >= 0.0.5)
from prog.rb:2
```

正如我们前面说过的，版本需求参数是可选参数，显而易见这个例子是故意编写的。但是我们可能，容易去想象这个特性会多么有用，因为若干不同项目可能依赖于同一个库的多个版本，而不同版本之间可能会不兼容。

### 17.3.3 依赖于 RubyGems

#### Dependent on RubyGems

细心的读者（你们都是）会注意到我们至今编写的这些代码都依赖于已安装的 RubyGems 包。从长远来看，这会是一个相当安全的赌注（我们猜测 RubyGems 会最终进入 Ruby 核心发行版中）。然而由于现在 RubyGems 还不是标准 Ruby 发行版的一部分，所以使用你的软件的人可能没有把 RubyGems 安装到他们的电脑里。如果我们的发行代码有 `require 'rubygems'` 语句，那么代码会出错。

## 幕后的代码

### The Code Behind the Curtain

那么当调用魔术般的 `require_gem` 方法时，幕后究竟会发生些什么呢？

第一，`gems` 库修改 `$LOAD_PATH` 包括已经添加到 `gemspec` 中的 `require_paths` 的任何目录。第二，它对 `gemspec` 的 `autorequires` 属性（在第 224 页描述）中指定的任何文件调用 Ruby 的 `require` 方法。正是这个 `$LOAD_PATH` 的修改行为允许 RubyGems 管理相同库的多个已安装版本。

至少可以使用两种技术去规避这个问题。首先，可以把 RubyGems 特有的代码封装到代码块中，并使用 Ruby 的异常处理去 `rescue` 这个在 `require` 时因为没有找到 RubyGems 而引发的 `LoadError`。

```
begin
 require 'rubygems'
 require_gem 'BlueCloth', ">= 0.0.4"
rescue LoadError
 require 'bluecloth'
end
```

这段代码首先试图在 RubyGems 库中请求。如果失败，则 `rescue` 代码段会被调用，同时程序会试图使用传统的 `require` 方法来装入 `BlueCloth`。如果 `BlueCloth` 没有安装，那么 `require` 会失败，用户此刻看到的状态和他们没有使用 RubyGems 时是一样的。

在 RubyGems 0.8.0 中，请求 `rubygems.rb` 会安装一个重载版本的 Ruby `require` 方法。安装了 RubyGems 框架之后，可以写成：

```
require 'bluecloth'
```

尽管这看起来像传统的代码，幕后 RubyGems 实际上会从当前安装的 `gems` 列表中找到第一个匹配并将 `bluecloth.rb` 装入。

重载的 `require` 方法几乎可以让程序避免使用任何 RubyGems 特有的代码。一个例外是，在请求这些通过 `gem` 安装的库之前，RubyGems 库必须先被装载。

可以用 `-r` 开关调用 Ruby 解释器来避免依赖 RubyGems。

```
ruby -rubygems myprogram.rb
```

这会导致解释器装入 RubyGems 框架，从而安装了 RubyGems 重载版本的 `require` 方法。为了在给定系统上确保当每次调用 Ruby 解释器时 RubyGems 会被装入，可以设置 `RUBYOPT` 环境变量：

```
% export RUBYOPT=rubygems
```

这样当你运行 Ruby 解释器时无须显式装载 RubyGems 框架，那些以 `gem` 方式安装的库对需要它们的那些应用是可用的。

使用重载的 `require` 方法的最大的缺点是，失去了管理同一库的多个已安装版本的能力。如果需要特定版本的库，最好使用先前描述的 `LoadError` 方式。

## 17.4 创建自己的 Gems

### Creating Your Own Gems

现在你已经看到，对应用或库的用户来说，RubyGems 使事情变得那么简单，你可能准备好了去创建自己的 `gem`。如果你正在编写将在开源社区共享的代码，RubyGems 是让最终用户去发现、安装和卸载你的代码的一种理想的方式。它们（`gems`）也提供了一种强有力手段去管理内部项目、公司项目或者个人项目，因为它们使得升级和回滚（`rollbacks`）变得如此简单。最终，更多的 `gems` 会使得 Ruby 社区更强大。这些 `gems` 必须来自四面八方；在这里我们准备告诉你如何把自己的 `gem` 贡献给 Ruby 社区。

假设你终于实现了妈妈的在线日记应用 `MomLog`，并决定在一个开源许可证下发布它。毕竟，别的程序员也有妈妈。自然而然地，你希望把 `MomLog` 当作一个 `gem` 来发布（妈妈会很高兴，如果你提供给她们的是 `gems`）。

#### 17.4.1 包的布局

##### Package Layout

创建 `gem` 的首要任务是在一个有意义的目录结构中组织代码。在创建典型的 `tar` 或 `zip` 档案时所使用的一些规则，同样适用于对软件包的组织。下面是一些通常的惯例。

- 把所有的 Ruby 源文件放在 `lib/` 子目录中。后面会显示当用户装入这个 `gem` 时，如何确保这个目录会添加到 Ruby 的 `$LOAD_PATH` 中。
- 如果对你的项目合适，那么包含一个 `lib/yourproject.rb` 的文件，它会执行必要的 `require` 命令去装入项目的大部分功能。在 RubyGems 的 `autorequire` 特性出现之前，这使得别人更方便地使用库。即使有了 RubyGems，它使得别人可以更便利地探索你的代码，因为你给了它们一个显见的起点。

- 总是包括 `README` 文件，其中包含项目概要、作者联系信息以及入门知识的获取。这个文件使用 `RDoc` 格式，这样可以把它添加到 `gem` 安装时生成的文档中。记住 `README` 文件应当包含版权和许可证，因为很多商业用户不会轻易使用一个包，除非它的许可证条款是清晰的。
- 测试用例应当放在 `test/` 目录中。很多开发者把库的单元测试当作一个用法指南来使用。把它们放在某些可预见的地方是不错的做法，这让别人可以很容易地找到它们。
- 任何可执行的脚本应当放在 `bin/` 子目录中。
- Ruby 扩展源代码应当放在 `ext/` 中。
- 如果 `gem` 包含非常多的文档，把这些文档保存在它们自己的 `docs/` 子目录中是好的做法。如果 `README` 文件处在包的顶层目录，一定要对读者提及这个位置。

第 232 页的图 17.1 说明了包的布局。

#### 17.4.2 Gem 规范 The Gem Specification

现在以所希望的方式组织安排了那些文件，是讲述 `gem` 创建核心的时候了：`gem` 规范或 `gemspec`。`gemspec` 是 Ruby 或 YAML（见第 758 页）形式的元数据集，它们提供了关于 `gem` 的关键信息。`gemspec` 作为构建 `gem` 流程的输入。可以使用几种不同的机制来创建 `gem`，但它们在概念上都是相同的。下面是你的第一个基本的 `gem`，`MomLog`。

```
require 'rubygems'

SPEC = Gem::Specification.new do |s|
 s.name = "MomLog"
 s.version = "1.0.0"
 s.author = "Jo Programmer"
 s.email = "jo@joshost.com"
 s.homepage = "http://www.joshost.com/MomLog"
 s.platform = Gem::Platform::RUBY
 s.summary = "An online Diary for families"
 s.candidates = Dir.glob("{bin,docs,lib,test}/**/*")
 s.files = candidates.delete_if do |item|
 item.include?("CVS") || item.include?("rdoc")
 end
 s.require_path = "lib"
 s.autorequire = "momlog"
 s.test_file = "test/ts_momlog.rb"
 s.has_rdoc = true
 s.extra_rdoc_files = ["README"]
 s.add_dependency("BlueCloth", ">= 0.0.4")
end
```

让我们快速地介绍一下这个例子。`gem` 的元数据保存在 `Gem::Specification` 类的对象中。`gemspec` 可以用 YAML 或 Ruby 代码来表达。这里我们会显示 Ruby 版本的 `gemspec`，因为通常它更容易构造和更灵活地使用。规范里的最初 5 个属性给出了基本信息诸如 `gem` 名称、版本，以及作者姓名、电子邮件和主页。

在这个例子中，下一个属性是 `gem` 可以运行的平台。因为这个 `gem` 是一个纯 Ruby 库，没有对操作系统的特别需求，所以设置平台为 `RUBY`。如果 `gem` 只是为 Windows 开发的，平台应当是 `WIN32`。现在这个字段只是提供信息用的，但是将来 `gem` 系统会使用它智能地选择已经预编译好的本地扩展 `gems`。

`gem` 的概述是对软件包的简短描述，运行 `gem query` 时会显示它（就像在前面的 `BlueCloth` 例子中）。

`files` 属性是一个文件数组，编译 `gem` 时会包括这些文件。在这个例子中，我们使用了 `Dir.glob` 来生成这个表，并过滤掉那些 CVS 和 RDoc 文件。

### 17.4.3 运行时魔术

#### Runtime Magic

接下来的两个属性 `require_path` 和 `autorequire` 让你指定当 `require_gem` 装入 `gem` 时，会被添加到 `$LOAD_PATH` 中的目录，以及通过 `require` 被自动装入的所有文件。在这个例子中，`lib` 指的是 `MemLog` gem 目录中的一个相对路径，并且 `autorequire` 会导致当调用 `require_gem "MemLog"` 时装入 `lib/memlog.rb`。RubyGems 也提供了 `require_path` 的复数版本，`require_paths`。它接受一个数组，允许指定多个将要包含在 `$LOAD_PATH` 中的目录。

### 17.4.4 添加测试和文档

#### Adding Tests and Documentation

`test_file` 属性保存 `gem` 中的单个相对路径文件，这个文件应当作为 `Test::Unit` 测试套件（`test suite`）被装入。（可以使用复数形式的 `test_files` 去引用一个包含测试的文件数组。）关于如何创建测试套件的详细描述，见第 151 页第 12 章的单元测试。

结束这个例子前，我们还有最后两个属性，它们控制了 `gem` 本地文档的生成。`has_rdoc` 属性表明你已经在代码中添加了 `RDoc` 注释。在完全没有注释的代码上运行 `RDoc` 是可行的，只要提供一个其接口的可浏览视图，但是很显然，与在有着良好注释的代码上运行 `RDoc` 相比，这失去了太多有价值的东西。`has_doc` 让你告诉世界，“是的，为这个 `gem` 生成文档是值得的。”

RDoc 格式无论以源文件或其渲染后的形式来看，可读性都非常好，这使得 Rdoc 格式成为 README 文件的一个极好的选择。默认情况下，`rdoc` 命令只会运行在源代码文件上。`extra_rdoc_file` 属性是 `gem` 中非源代码文件的一个数组，你可能希望当生成 RDoc 文档时包括这些文件。

## 17.4.5 添加依赖

### Adding Dependencies

为了让你的 `gem` 可以正常地工作，用户需要安装 `BlueCloth`。

先前我们已经看到如何设置库在装入时的版本依赖。现在我们需要把这个依赖告诉 `gemspec`，这样安装程序会确保当安装 `MomLog` 时它是存在的。可以通过添加对 `Gem::Specification` 对象的一个方法调用来实现它。

```
s.add_dependency("BlueCloth", ">= 0.0.4")
```

`add_dependency` 方法的参数等同于 `require_gem` 的参数，这些参数在前面已解释过。

生成了 `gem` 后，当试图在干净的系统上安装它时，会看到类似下面的消息。

```
% gem install pkg/MomLog1.0.0.gem
Attempting local installation of 'pkg/MomLog1.0.0.
gem'
/usr/local/lib/ruby/site_ruby/1.8/rubygems.rb:50:in `require_gem':
(LoadError)
Could not find RubyGem BlueCloth (>= 0.0.4)
```

因为正在从文件中执行本地安装，RubyGems 不会试图解决这个依赖。相反，它会大声嚷嚷失败了，告诉你它需要 `BlueCloth` 去完成这次安装。然后你可能就像我们前面所做的那样去安装 `BlueCloth`，在下一次尝试安装 `MomLog` gem 时，事情会进行得很顺利。

如果已经将 `MomLog` 上传到中央 RubyGems 仓库中，然后试着在干净的系统上安装它，作为 `MomLog` 安装的一部分，它会提示你自动安装 `BlueCloth`。

```
% gem install rMomLog
Attempting remote installation of 'MomLog'
Install required dependency BlueCloth? [Yn] y
Successfully installed MomLog, version 1.0.0
```

现在已经把 `BlueCloth` 和 `MomLog` 都安装好了，妈妈就可以开始愉快地发布她的日记。如果选择不安装 `BlueCloth`，这次安装会失败，就像试图本地安装时所发生的那样。

当把更多的特性添加到 `MomLog` 时，你可能会发现自己需要请求更多的外部 `gems` 去支持这些特性。可以在一个 `gemspec` 中多次调用 `add_dependency` 方法，来支持你需要它去解决的那些依赖关系。

## 17.4.6 Ruby 扩展 Gems

### Ruby Extension Gems

到目前为止，我们看到的所有例子都是纯 Ruby 代码。当然，许多 Ruby 库是作为本地扩展方式（native extension）来创建的（见第 275 页的第 21 章）。作为一个 gem，有两种方式可以去打包和分发这种类型的库。可以用源码格式分发 gem，同时让安装程序在安装时编译代码。另外，可以预编译这个扩展，为每个要支持的独立平台分发一个 gem。

RubyGems 会为源码 gems 提供附加的 `Gem::Specification` 属性，称为 `extensions`。这个属性是一个 Ruby 文件的数组，它们会生成 `Makefiles`。创建其中一个程序的最典型方式是使用 Ruby 的 `mkmf` 库（见第 275 页的第 21 章以及第 779 页的关于 `mkmf` 的附录）。这些文件尽管实际上任何名字都可以，但是按惯例被命名为 `extconf.rb`。

妈妈有一个电子版本的菜谱数据库，这几乎是她的心肝宝贝。她在数据库中保存了那些菜谱已经好多年了，为了她的朋友和家庭也能用到这些菜谱，你可能希望使她有能力在 Web 上发布这些菜谱。你发现 `MenuBuild` 菜谱程序有一个相当好的 native API，所以决定编写一个 Ruby 扩展去封装它。既然这个扩展可能会对别人有用，他们可能不需要使用 `MomLog`，所以你决定把它打包为单独的 gem，同时把它作为 `MemLog` 的附加依赖。

这里是这个 `gemspec`。

```
require 'rubygems'

spec = Gem::Specification.new do |s|
 s.name = "MenuBuilder"
 s.version = "1.0.0"
 s.author = "Jo Programmer"
 s.email = "jo@joshost.com"
 s.homepage = "http://www.joshost.com/projects/MenuBuilder"
 s.platform = Gem::Platform::RUBY
 s.summary = "A Ruby wrapper for the MenuBuilder recipe database."
 s.files = ["ext/main.c", "ext/extconf.rb"]
 s.require_path = "."
 s.autorequire = "MenuBuilder"
 s.extensions = ["ext/extconf.rb"]
end
if $0 == __FILE__
 Gem::manage_gems
 Gem::Builder.new(spec).build
end
```

注意到必须在这个规范的 `files` 列表中包含那些源文件，这样它们会包含在分发的 gem 包中。

当安装源码 gem 时，RubyGems 运行其中的每个 `extensions` 程序，并执行进而产生的 `Makefile`。

```
% gem install MenuBuilder-1.0.0.gem
Attempting local installation of 'MenuBuilder1.0.0.gem'
ruby extconf.rb inst MenuBuilder1.0.0.gem
creating Makefile
make
gcc -fPIC -g -O2 -I. -I/usr/local/lib/ruby/1.8/i686-linux \
 -I/usr/local/lib/ruby/1.8/i686-linux -I. -c main.c
gcc -shared -L"/usr/local/lib" -o MenuBuilder.so main.o \
 -ldl -lcrypt -lm -lc
make install
install -c -p -m 0755 MenuBuilder.so \
 /usr/local/lib/ruby/gems/1.8/gems/MenuBuilder-1.0.0/.
成功安装 MenuBuilder, 版本 1.0.0
```

RubyGems 没有能力检测源码 gems 可能有的系统库依赖。如果源码 gems 依赖于一个没有安装的系统库，gem 安装会失败，同时会显示 make 命令的一个错误输出。

分发源码的 gem 显然需要这个 gem 的使用者有一套开发工具。至少他们需要某种类型的 make 程序和编译器。尤其对 Windows 用户来说，他们可能没有这些工具。可以通过分发预编译的 gems 规避这种不足。

创建预编译的 gems 很简单，只要把已编译的共享库文件（share object，在 Windows 上是 DLL）添加到 gemspec 的 files 列表中，同时确信这些文件在 gem 的 require\_path 属性的某个路径中。与纯 Ruby gems 一样，require\_gem 命令会修改 Ruby 的\$LOAD\_PATH，共享库会通过 require 来访问。

既然这些 gems 是平台特有的，也可以使用 platform 属性（还能从第一个 gemspec 例子中记起它吗？）指定 gem 的目标平台。Gem::Specification 类为 Windows, Intel Linux, Macintosh 和纯 Ruby 平台定义了相应常量。对于那些没有包含在这个表中的平台，可以使用 RUBY\_PLATFORM 变量的值。这个属性现在纯粹只是用来提供信息的，但是使用它是一个应当有的良好习惯。将来的 RubyGems 发行会根据安装程序正在运行的平台，使用 platform 属性去智能地选择已预编译的 gems。

### 17.4.7 编译 Gem 文件 Building the Gem File

刚刚创建的 MomLog gemspec 是一个可运行的 Ruby 程序。调用它会生成一个 gem 文件，MomLog-0.5.0.gem。

```
% ruby momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

另外，可以使用 `gem build` 命令生成 gem 文件。

```
% gem build momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

现在有了 gem 文件，你可以像任何其他包一样来分发它。可以把它放在 FTP 服务器上或 Web 站点上供人下载或通过电子邮件发送给你的朋友。一旦你的朋友在他们的本地电脑上得到了这个文件（必要的话，可以从你的 FTP 服务器上下载），他们可以通过调用下面的命令来安装这个 gem（假设他们已经把 RubyGems 也安装上了）。

```
% gem install MomLog-0.5.0.gem
Attempting local installation of 'MomLog-0.5.0.gem'
Successfully installed MomLog, version 0.5.0
```

如果想要把你的 gem 发行到 Ruby 社区，最简便的办法是使用 RubyForge (<http://rubyforge.org>)。RubyForge 是一个管理开源项目的 Web 站点。它也存储中央 gem 仓库。任何使用 RubyForge 文件发行功能发布的 gem 文件，每天会有多次被自动选取并添加到中央 gem 仓库中。对使用你的软件的潜在用户来说，其好处是，通过 RubyGems 的远程查询和安装操作来获得它，这使得安装更加容易。

## 17.4.8 用 Rake 编译

### Building with Rake

最后但是肯定不是至少，我们可以使用 Rake 来编译 gems（记得 Rake 这个纯 Ruby 编译工具在第 216 页提及过）。Rake 使用一个称为 `Rakefile` 的命令行文件去控制编译过程。它定义了（用 Ruby 语法！）一组规则和任务。Make 中以规则驱动的概念和 Ruby 能力的结合，创建了一个自动构建和发行的梦幻般的环境。哪个 Ruby 项目的发布在没有生成 gem 之前而算是彻底完成的呢？

关于如何使用 Rake 的细节，请参见这个项目的 Web 页面 <http://rake.rubyforge.org>。它的文档是完备的并总是最新的。在这里我们只是侧重于介绍如何使用 Rake 来编译 gem。下面这段摘自 Rake 文档：

任务是 `Rakefile` 中主要的工作单元。任务有一个名称（通常以符号或字符串方式给出），一个前提条件的列表（更多的符号或字符串）和一个动作列表（以 `block` 的方式给出）。

一般情况下，你可以使用 Rake 内建的 `task` 方法在 `Rakefile` 中定义自己命名的任务。对一些特殊的例子，提供辅助（helper）代码来自动化执行一些重复性的工作是有意义的，要不然你得自己亲自做。创建 Gem 是这些具体例子中的一个。Rake 提供了特别的

*TaskLib*, 它被称为 *GemPackage Task*, 帮助把 *gem* 创建整合到你的自动化构建和发行阶段的剩余部分。

为了在 *Rakefile* 中使用 *GemPackageTask*, 如我们前面所做的那样创建 *gemspec*, 但是这次是把它放在 *Rakefile* 中。然后我们把这个规范传送给 *GemPackageTask*。

```
require 'rubygems'
Gem::manage_gems
require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|
 s.name = "MomLog"
 s.version = "0.5.0"
 s.author = "Jo Programmer"
 s.email = "jo@joshost.com"
 s.homepage = "http://www.joshost.com/MomLog"
 s.platform = Gem::Platform::RUBY
 s.summary = "An online Diary for families"
 s.files = fileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
 s.require_path = "lib"
 s.autorequire = "momlog"
 s.test_file = "tests/ts_momlog.rb"
 s.has_rdoc = true
 s.extra_rdoc_files = ["README"]
 s.add_dependency("BlueCloth", ">= 0.0.4")
 s.add_dependency("MenuBuilder", ">= 1.0.0")
end

Rake::GemPackageTask.new(spec) do |pkg|
 pkg.need_tar = true
end
```

注意必须请求 *rubygems* 包到 *Rakefile* 中。你也会注意到我们使用了 *Rake* 的 *FileList* 类而不是 *Dir.glob* 去生成文件列表。*FileList* 比 *Dir.glob* 更智能, 它会自动忽视通常不会使用到的那些文件(比如 *CVS* 目录, 它是 *CVS* 版本控制工具留下来的)。

*GemPackageTask* 在内部以下如下标识符生成一个 *Rake* 目标,

```
package_directory/gemname-gemversion.gem
```

在这个例子中, 这个标识符是 *pkg/MomLog-0.5.0.gem*。可以从放置 *Rakefile* 文件的相同目录中调用这项任务。

```
% rake pkg/MomLog-0.5.0.gem
(in /home/chad/download/gembook/code/MomLog)
 Successfully built RubyGem
 Name: MomLog
 Version: 0.5.0
 File: MomLog-0.5.0.gem
```

现在，你已经搞定了项任务，可以像任何其他 Rake 任务那样去使用它，为其添加依赖或者将它添加到另一项任务的依赖列表中，比如部署或发行打包。

### 17.4.9 维护 Gem (最后看看 MomLog)

#### Maintaining Your Gem (and One Last Look at MomLog)

MomLog 已经发布，并每周都在吸引新的崇拜者。你已经非常小心干净地打包，同时使用 Rake 来编译 gem。

这个带有你联系方式的 gem 开始迅速“蹿红”，你知道，开始从用户那儿收到特性请求（和 MomLog 迷的邮件！）只是时间的问题。但是，第一个请求来自于一个电话，不是别人，正是亲爱的老妈妈打过来的。她刚从佛罗里达度假回来，问你如何可以把她的假期照片放到她的日记中。你肯定不会认为花时间向她解释 FTP 命令行工具是有效的，作为永远孝顺的儿女，你会花一晚上为 MomLog 编写一个好的相册模块。

既然已经把新的功能添加到程序中（而不仅仅是解决了一个错误），你就可以决定把 MomLog 的版本号从 1.0.0 升级到 1.1.0。同时也为新功能添加一组测试和文档，去解释如何设置相片上传功能。

下页的图 17.1 显示了最终的 MomLog 1.1.0 包的完全目录结构。最终的 gem 规范（从 Rakefile 抽取出来的）看起来像这个。

```
spec = Gem::Specification.new do |s|
 s.name = "MomLog"
 s.version = "1.1.0"
 s.author = "Jo Programmer"
 s.email = "jo@joshhost.com"
 s.homepage = "http://www.joshhost.com/MomLog"
 s.platform = Gem::Platform::RUBY
 s.summary = "An online diary, recipe publisher, " +
 "and photo album for families."
 s.files = FileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
 s.require_path = "lib"
 s.autorequire = "momlog"
 s.test_file = "tests/ts_momlog.rb"
 s.has_rdoc = true
 s.extra_rdoc_files = ["README", "docs/DatabaseConfiguration.rdoc",
 "docs/Installing.rdoc", "docs/PhotoAlbumSetup.rdoc"]
 s.add_dependency("BlueCloth", ">= 0.0.4")
 s.add_dependency("MenuBuilder", ">= 1.0.0")
end
```

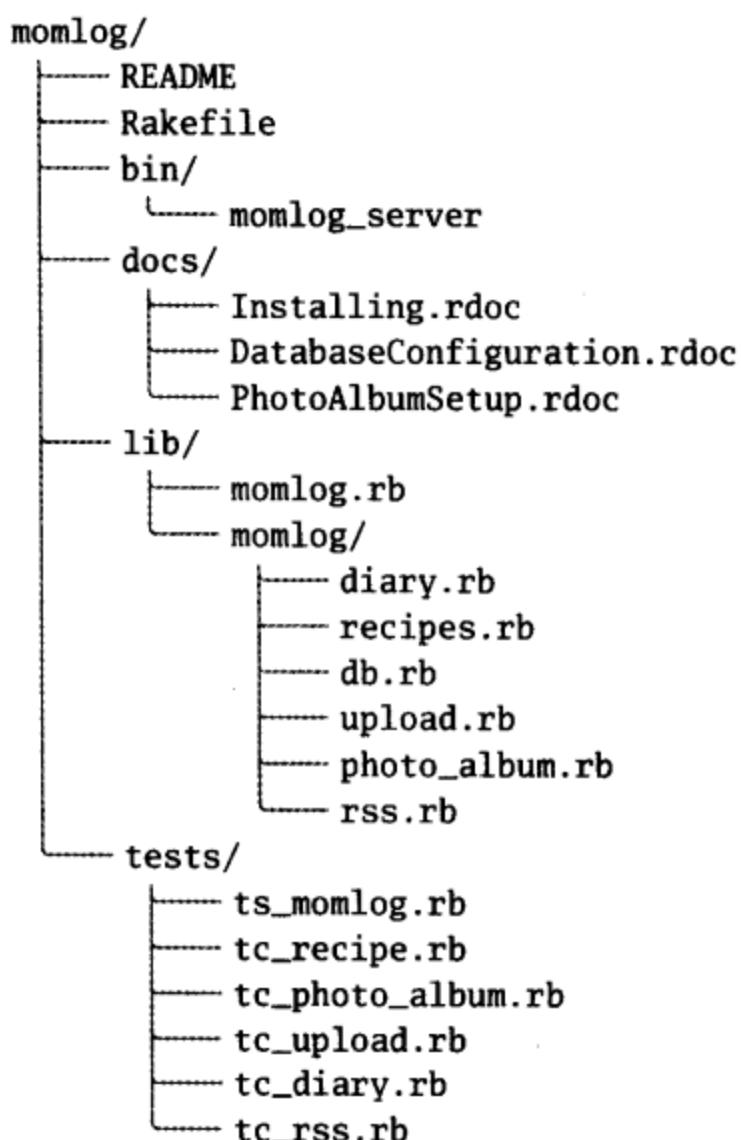


图 17.1 MomLog 包结构

从 `Rakefile` 运行 `Rake`，它会生成更新的 `MomLog` gem，现在你可以发布这个新版本了。登录到你的 RubyForge 账户，将 `gem` 上传到你项目中的“文件”区。在等待 RubyGems 自动将这个 `gem` 发布到中央 `gem` 仓库的过程中，你撰写了发布通告并张贴到你的 RubyForge 项目中。

大约一个小时之内，你登录到妈妈的 Web 服务器，为她安装了这个新软件。RubyGems 使事情变得很简单，但是我们必须特别照顾一下妈妈。

```
% gem query -rn MomLog
*** REMOTE GEMS ***
MomLog (1.1.0, 1.0.0)
 An online diary, recipe publisher, and photo album for families.
```

太好了！这个查询表明这里现在有两个版本的 MomLog 可用。你在键入 `install` 命令时并没有指定版本参数，是因为你知道在默认条件下会安装最近的版本。

```
% gem install -r MomLog
Attempting remote installation of 'MomLog'
Successfully installed MomLog, version 1.1.0
```

你并没有改变任何的 MomLog 依赖，所以现有的 BlueCloth 和 MenuBuilder 安装满足 MomLog 1.1.0 的需求。

妈妈现在很快乐，是去尝试一下她最近张贴菜谱的时候了。





# Ruby 与 Web

## Ruby and the Web

Ruby 对 Internet 并不陌生。你不仅可以用 Ruby 编写 SMTP 服务器、FTP 守护程序或者 Web 服务器，而且还可以用 Ruby 做一些更一般性的工作，例如用于 CGI 编程或者替代 PHP。

使用 Ruby 来实现 Web 应用有许多可用的方式，而以一章的篇幅无法充分介绍它们。但我们可以尝试涉足其中的一些亮点，并且指引你相关的库和资源。

让我们从一些简单的事情开始：运行 Ruby 程序作为通用网关接口（Common Gateway Interface, CGI）程序。

### 18.1 编写 CGI 脚本

#### Writing CGI Scripts

使用 Ruby 编写 CGI 脚本非常容易。要让 Ruby 脚本产生 HTML 输出，你只需编写如下的代码：

```
#!/usr/bin/ruby
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

将这个脚本放到 CGI 目录中，使其有可执行的权限，然后你可以通过浏览器来访问它。（如果你的 Web 服务器并不自动添加头信息[header]，你需要自己添加响应头信息。）

```
#!/usr/bin/ruby
print "HTTP/1.0 200 OK\r\n"
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

不过这种编程方式太底层了。你需要编写自己的请求解析、会话管理、cookie 操作、输出转义，等等。幸运的是，有其他方式可以使它们更简单。

### 18.1.1 使用 cgi.rb

#### Using cgi.rb

类 CGI 提供了编写 CGI 脚本的支持。使用它，你可以操作表单、cookie 和环境；维护有状态的会话，等等。它是一个相当大的类，不过我们在这里只快速浏览一下它的功能。

### 18.1.2 引用

#### Quoting

当要处理 URL 和 HTML 编码时，你一定要小心引用某些特定的字符。例如，斜杠字符 (/) 在 URL 中有特殊的意义，所以如果它是路径名的一部分，就必须要进行转义（escaped）。也就是说，URL 查询部分的任何/都要被转换为%2F，而在你使用它时再转回成/。空格和&字符也是特殊字符。要处理这种转换，CGI 提供了 CGI.escape 和 CGI.unescape 两个例程（routine）。

```
require 'cgi'
puts CGI.escape("Nicholas Payton/Trumpet & Flugel Horn")
```

输出结果：

```
Nicholas+Payton%2FTrumpet+%26+Flugel+Horn
```

更常见的，你可能想要转义 HTML 特殊字符。

```
require 'cgi'
puts CGI.escapeHTML("a < 100 && b > 200")
```

输出结果：

```
a < 100 && b > 200
```

更奇妙的是，你可以决定只对字符串中某个特定的 HTML 元素进行转义。

```
require 'cgi'
puts CGI.escapeElement('<hr>Click Here
', 'A')
```

输出结果：

```
<hr>Click Here

```

这里只有 A (Anchor, 锚点) 元素被转义了；而其他元素则保持原样。这些方法都有一个“un-”（反向）的版本来恢复原来的字符串。

```
require 'cgi'
puts CGI.unescapeHTML("a < 100 && b > 200")
```

输出结果：

```
a < 100 && b > 200
```

### 18.1.3 查询参数 Query Parameters

从浏览器到你应用的 HTTP 请求中可能包括参数，或者作为 URL 的一部分，或者将数据嵌入到请求体中。

因为在一个请求中相同名字的值可能会返回多次，因此处理这些参数是很复杂的。例如，假定我们要编写一个调查以找出为什么人们喜欢 Ruby。我们的 HTML 表单看起来大概是下面这个样子。

```
<html>
 <head><title>Test Form</title></head>
 <body>
 I like Ruby because:
 <form target="cgi-bin/survey.rb">
 <input type="checkbox" name="reason" value="flexible" />
 It's flexible

 <input type="checkbox" name="reason" value="transparent" />
 It's transparent

 <input type="checkbox" name="reason" value="perlish" />
 It's like Perl

 <input type="checkbox" name="reason" value="fun" />
 It's fun
 <p>
 Your name: <input type="text" name="name">
 </p>
 <input type="submit"/>
 </form>
 </body>
</html>
```

当某人填写这个表单时，他可能选择多种喜爱 Ruby 的原因（参见下一页的图 18.1）。这种情况下，对应 `reason` 字段的表单数据可能有 3 个值，分别对应所选择的选项。

类 `CGI` 为你提供了几种方式来访问表单数据。首先，你可以把 `CGI` 对象看作是一个散列表，通过字段名（`field`）索引并返回字段的值。

```
require 'cgi'
cgi = CGI.new
cgi['name'] → "Dave Thomas"
cgi['reason'] → "flexible"
```

但是这对 `reason` 字段来说并不能很好地工作：只能看到 3 个值中的一个。我们可以 18 通过使用 `CGI#params` 方法来要求全部查看它们。`params` 返回的值类似一个散列表，保存有请求的参数。你可以读写这个散列表（写操作可以让你修改与这个请求关联的数据）。注意，散列表中的每个值实际上都是一个数组。

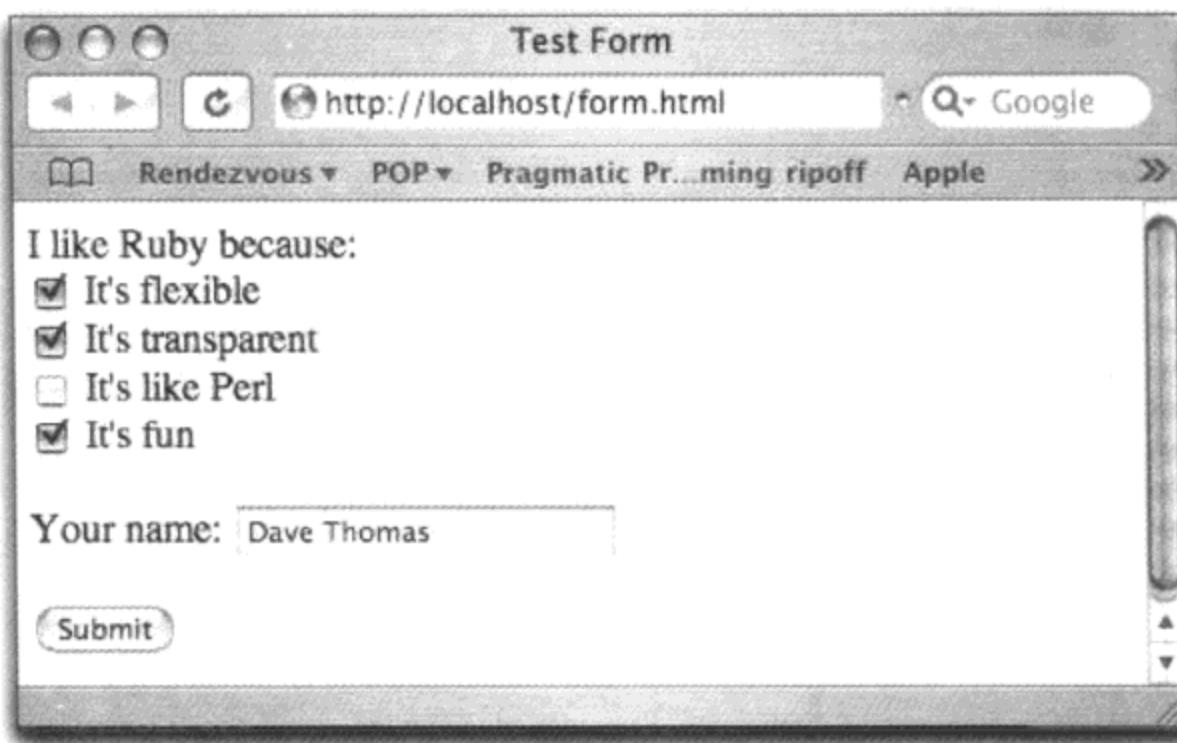


图 18.1 简单的 CGI 表单

```
require 'cgi'
cgi = CGI.new
cgi.params → {"name"=>["Dave Thomas"],
 "reason"=>["flexible", "transparent",
 "fun"]}
cgi.params['name'] → ["Dave Thomas"]
cgi.params['reason'] → ["flexible", "transparent", "fun"]
cgi.params['name'] = [cgi['name'].upcase]
cgi.params → {"name"=>["DAVE THOMAS"],
 "reason"=>["flexible", "transparent",
 "fun"]}
```

你可以使用 `CGI#has_key?` 来判断请求中是否包括某个特定参数。

```
require 'cgi'
cgi = CGI.new
cgi.has_key?('name') → true
cgi.has_key?('age') → false
```

### 生成 HTML

CGI 包括大量的方法用来创建 HTML——每个元素一个方法。要启用这些方法，你必须调用 `CGI.new` 创建一个 CGI 对象，传入需要的 HTML 标准级别。在后面的示例中，我们使用 `html3`。

为了让元素嵌套更容易些，这些方法将代码 block 作为其内容。这些代码 block 应该返回一个 String，作为该元素的内容。对下面的示例来说，我们添加了一些不必要的断行来输出符合页面的宽度。

```
require 'cgi'
cgi = CGI.new("html3") # add HTML generation methods
cgi.out {
 cgi.html {
 cgi.head { "\n"+cgi.title("This Is a Test") } +
 cgi.body { "\n"+
 cgi.form { "\n"+
 cgi.hr +
 cgi.h1 { "A Form: " } + "\n"+
 cgi.textarea("get_text") +" \n"+
 cgi.br +
 cgi.submit
 }
 }
 }
}
```

输出结果：

```
Content-Type: text/html
Content-Length: 302

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><HEAD>
<TITLE>This Is a Test</TITLE></HEAD><BODY>
<FORM METHOD="post" ENCTYPE="application/x-www-form-urlencoded">
<HR><H1>A Form: </H1>
<TEXTAREA NAME="get_text" ROWS="10" COLS="70"></TEXTAREA>

<INPUT TYPE="submit"></FORM></BODY></HTML>
```

这段代码会以 “This Is a Test” 为标题产生一个 HTML 表单，之后是一条横线，一个 H1 的标题，一个文本输入框，最后是一个提交按钮。当提交返回时，你将得到一个名为 get\_text 的 CGI 参数，其中包括用户所输入的文本。

虽然有趣，但是这个生成 HTML 的方法还是相当费力的，并且可能在实践中也不多使用。多数人倾向直接书写 HTML，使用模板系统，或者使用应用框架，例如 Iowa。不幸的是，我们没有更多的篇幅来讨论 Iowa——参见 <http://enigo.com/projects/iowa> 的在线文档，或者查看《Ruby 开发者指南》（*The Ruby Developer's Guide*）的第 6 章[FJN02]——不过我们可以考察一下模板。

## 模板系统

模板系统可以让你将应用的表示和逻辑分隔开来。好像所有用 Ruby 编写 Web 应用

的人都在同时编写一个模板系统：RubyGarden 的 wiki 列出了许多，<sup>1</sup>甚至这个列表也还不完整。现在，让我们只查看其中的 3 个：RDoc 模板、Amrita 和 erb/eruby。

## RDoc 模板

RDoc 文档系统（在第 199 页的第 16 章中介绍过）含有一个非常简单的模板系统，用来将它的 XML 文档产生 HTML 输出。因为 RDoc 是作为标准 Ruby 的一部分而分发的，安装了 Ruby 1.8.2 或之后的版本，所以可以获得这个模板系统。不过，这个模板系统并不使用传统的 HTML 或 XML 标记（因为它有意被用来产生许多不同格式的输出），因此有 RDoc 模板标记的文件，可能不容易用传统的 HTML 编辑工具来编辑。

```
require 'rdoc/template'
HTML = %{Hello, %name%.
<p>
The reasons you gave were:

START:reasons
 %reason_name% (%rank%)
END:reasons

}

data = {
 'name' => 'Dave Thomas',
 'reasons' => [
 { 'reason_name' => 'flexible', 'rank' => '87' },
 { 'reason_name' => 'transparent', 'rank' => '76' },
 { 'reason_name' => 'fun', 'rank' => '94' },
]
}

t = TemplatePage.new(HTML)
t.write_html_on(STDOUT, data)
```

输出结果：

```
Hello, Dave Thomas.
<p>
The reasons you gave were:

 flexible (87)
 transparent (76)
 fun (94)

```

---

<sup>1</sup> <http://www.rubygarden.org/ruby?HtmlTemplates>.

向构造函数传入一个包含了要应用的模板的字符串。然后将一个包括名字和值的散列表传入 `write_html_on` 方法。如果模板包括`%xxxx%`序列，则查找散列表，而对应名字`xxx`的值会被替换进去。如果模板包括`START:yyy`，对应`yyy`的散列值被认为是一个散列数组。在`START:yyy`和`END:yyy`之间的模板行重复应用于数组的每个元素。模板还支持条件：如果散列表中有`zzz`的 key，则`IF:zzz`和`ENDIF:zzz`之间的行会被包含到输出中。

### Amrita

Amrita<sup>2</sup>是生成 HTML 文档的库，其模板本身也是有效的 HTML。这使得 Amrita 对现有的 HTML 编辑器非常友好。它还意味着 Amrita 模板也可以作为独立的 HTML 页面正常显示。

Amrita 使用 HTML 元素的 `id` 标记来判断值是否要替换。如果对应给定名字的值是`nil`或`false`，HTML 元素不会包含到输出结果中。如果值是一个数组，它迭代对应的 HTML 元素。

```
require 'amrita/template'
include Amrita

HTML = %{<p id="greeting" />
<p>The reasons you gave were:</p>

 <li id="reasons">,

}

data = {
 :greeting => 'Hello, Dave Thomas',
 :reasons => [
 { :reason_name => 'flexible', :rank => '87' },
 { :reason_name => 'transparent', :rank => '76' },
 { :reason_name => 'fun', :rank => '94' },
]
}

t = TemplateText.new(HTML)
t.prettyprint = true
t.expand(STDOUT, data)
```

输出结果：

```
<p>Hello, Dave Thomas</p>
<p>The reasons you gave were:</p>
```

---

<sup>2</sup> <http://www.brain-tokyo.jp/research/amrita/rdocs/>.

```

 flexible, 87
 transparent, 76
 fun, 94

```

## erb 和 eruby

现在我们已经考察了使用 Ruby 来创建 HTML 输出，不过我们也可以将问题反过来考虑：实际上我们可以将 Ruby 嵌入到 HTML 文档中。

有很多包可以让你将 Ruby 语句嵌入到其他形式的文档中，特别是 HTML 页面。通常这被称为“eRuby”。特别地，eRuby 存在很多不同的实现，包括 eruby 和 erb。eruby 由 Shugo Maeda 编写，可以从 Ruby Application Archive 下载。erb，它的表兄，是用纯 Ruby 编写的，并且包含在标准发布中。我们现在先考察 erb。

在 HTML 中嵌入 Ruby 是非常强大的概念——它基本上提供了同 ASP、JSP 或 PHP 对等的工具，但是具有 Ruby 的全部能力。

### 使用 erb

erb 通常被作为一个过滤器。输入文件中的文本除了以下情况外，其他不会被更改。

表达式	描述
<code>&lt;% ruby code %&gt;</code>	执行分隔符之间的 Ruby 代码
<code>&lt;%= ruby expression %&gt;</code>	对 ruby 表达式求值，并用表达式的值替换序列
<code>&lt;%# ruby code %&gt;</code>	忽略分隔符之间的 Ruby 代码（对测试有用）
<code>% line of ruby code</code>	由百分号开头的文本行被认为只包括 Ruby 代码

你可以如下调用 erb：

```
erb [options] [document]
```

如果忽略 *document* 参数，erubg 将从标准输入中读取。erb 的命令行选项在下一页的表 18.1 中给出。

让我们看看简单的例子，对下面的输入执行 erb 程序。

```
% a = 99
<%= a %> bottles of beer...
```

表 18.1 erb 的命令行选项

选项	描述
-d	设置\$DEBUG 为真
-Kkcode	指定其他的编码系统（参见第 179 页）
-n	显示生成的 Ruby 脚本（带有行号）
-r library	加载指定的 library
-P	不对以%开头的行进行 erb 处理
-S level	设置 safe level（安全级别）
-T mode	设置 trim mode（修剪模式）
-v	打开细节信息模式
-x	显示结果的 Ruby 脚本

由百分号开始的行简单地执行给定的 Ruby 语句。下一行包括序列<%= a %>，它会被 a 的值替换。

```
erb f1.erb
```

输出结果：

```
99 bottles of beer...
```

erb 是通过将它的输入改写为一个 Ruby 脚本并执行该脚本来工作的。你可以使用 -n 或 -x 选项来查看 erb 生成的 Ruby 脚本。

```
erb -x f1.erb
```

输出结果：

```
_erbout = ''; a = 99
_erbout.concat((a).to_s); _erbout.concat " bottles of beer...\n"
```

注意 erb 是如何构造字符串\_erbout 的，它包括来自模板的静态字符串，以及表达式的执行结果（本例中是 a 的值）。

当然，你可以将 Ruby 嵌入到更复杂的文本类型中，例如 HTML。第 245 页的图 18.2 演示了在 HTML 文档中的几个循环。

### 18.1.4 在 Apache 中安装 eruby

#### Installing eruby in Apache

如果你想在有一定访问量的 Web 站点中使用类似 erb 的页面生成技术，你可能要转换到使用 eruby，它有较好的性能。然后你可以配置 Apache Web 服务器自动使用 eRuby 来解析内嵌 Ruby 的文档，和 PHP 的方式一样。你可以用.rhtml 作为后缀创建内嵌 Ruby 的文件，然后配置 Web 服务器对这些文档运行 eruby 程序来产生想要的 HTML 输出。

为了在 Apache Web 服务器中使用 eruby，你需要执行下面的步骤。

1. 将 eruby 的二进制文件拷贝到 cgi-bin 目录中。

2. 在 httpd.conf 中添加下面两行。

```
AddType application/x-httpd-eruby .rhtml
Action application/x-httpd-eruby /cgi-bin/eruby
```

3. 如果需要，你还可以添加或替换 DirectoryIndex 指令，使其包括 index.rhtml。这让你可以使用 Ruby 来为没有 index.html 的目录建立目录列表。例如，下面的指令将令内嵌 Ruby 脚本的 index.rhtml，当目录中没有 index.html 或 index.shtml 时，作为候补。

```
DirectoryIndex index.html index.shtml index.rhtml
```

当然，你也可以提供一个站点级别的 Ruby 脚本。

```
DirectoryIndex index.html index.shtml /cgi-bin/index.rb
```

## 18.2 Cookies

Cookies 是让 Web 应用得以在用户机器上保存状态的一种方式。虽然让某些人皱眉头，cookies 仍然是记忆会话信息的一种便捷的（也是不可靠的）方式。

Ruby CGI 类为你处理加载和保存 cookies。你可以使用 CGI#cookies 方法来访问与当前请求相关联的 cookies，你可以通过将 CGI#out 的 cookies 参数设置为单个 cookie 或 cookies 数组的引用，将 cookies 返回给浏览器。

```
#!/usr/bin/ruby
COOKIE_NAME = 'chocolate chip'
require 'cgi'
cgi = CGI.new
values = cgi.cookies[COOKIE_NAME]
if values.empty?
 msg = "It looks as if you haven't visited recently"
else
 msg = "You last visited #{values[0]}"
end
cookie = CGI::Cookie.new(COOKIE_NAME, Time.now.to_s)
cookie.expires = Time.now + 30*24*3600 # 30 days
cgi.out("cookie" => cookie) { msg }
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>eruby example</title>
</head>
<body>
<h1>Enumeration</h1>

%5.times do |i|
 number <%=i%>
%end

<h1>"Environment variables starting with "T"</h1>
<table>
%ENV.keys.grep(/^T/).each do |key|
 <tr><td><%=key%></td><td><%=ENV[key]%></td></tr>
%end
</table>
</body>
</html>
```

输出结果：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>eruby example</title>
</head>
<body>
<h1>Enumeration</h1>

 number 0
 number 1
 number 2
 number 3
 number 4

<h1>"Environment variables starting with "T"</h1>
<table>
 <tr><td>TERM_PROGRAM</td><td>iTerm.app</td></tr>
 <tr><td>TERM</td><td>xterm-color</td></tr>
</table>
</body>
</html>
```

图 18.2 使用 erb 处理带有循环的文件

## 18.2.1 会话 Sessions

Cookies 本身还需要一点工作才能有用。我们需要会话 (*session*)：一种持久化的特定 Web 浏览器和请求之间的信息。会话是由 `CGI::Session` 处理的，它使用 `cookies` 但是提供了一种更高层次的抽象。

使用 `cookies`，会话模拟了一种类似散列表的行为，让你可以关联 `value` 和 `key`。与 `cookies` 不同，`session` 将大部分数据保存在服务器，使用浏览器驻留的 cookie 简单地作为服务器端数据的唯一标识。你还可以选择会话的存储技术：可以保存在一般文件中，在 `PStore`（参见第 719 页的描述）中，在内存中，甚至在你自定义的存储中。

会话在使用之后应该关闭，这可以确保它们的数据被写入到存储中。当你最终完成了一个会话之后，你应该删除它。

```
require 'cgi'
require 'cgi/session'

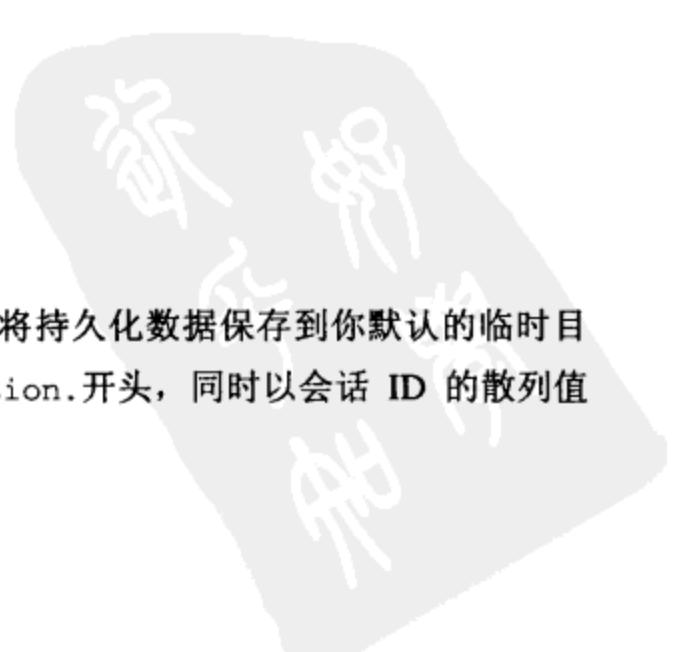
cgi = CGI.new("html3")
sess = CGI::Session.new(cgi,
 "session_key" => "rubyweb",
 "prefix" => "web-session."
)

if sess['lastaccess']
 msg = "You were last here #{sess['lastaccess']}."
else
 msg = "Looks like you haven't been here for a while"
end

count = (sess["accesscount"] || 0).to_i
count += 1

msg << "<p>Number of visits: #{count}"
sess["accesscount"] = count
sess["lastaccess"] = Time.now.to_s
sess.close

cgi.out {
 cgi.html {
 cgi.body {
 msg
 }
 }
}
```

上面的示例代码使用了会话的默认存储机制：将持久化数据保存到你默认的临时目录中（参见 `Dir.tmpdir`）。文件名均以 `web-session.` 开头，同时以会话 ID 的散列值结尾。更多信息请参见 `ri CGI::Session`。  


## 18.3 提升性能

### Improving Performance

你可以使用 Ruby 编写 Web 的 CGI 程序，但是，与大部分的 CGI 程序一样，默认的配置是每访问 cgi-bin 的一个页面，就启动一个新的 Ruby 程序拷贝。这对机器的利用而言是很昂贵的，而且对 Web 访问者来说慢得极其痛苦。Apache Web 服务器通过支持可加载的模块来解决这个问题。

一般来说，这些模块是动态加载的，并成为 Web 服务器运行进程的一部分——你不必一次又一次地衍生解释器进程来响应请求；Web 服务器便是解释器。

因此我们有赖于 mod\_ruby（可以从应用归档中获得），它是 Apache 的一个模块，将一个完整的 Ruby 解释器链接到 Apache Web 服务器本身。mod\_ruby 所带的 README 提供了如何编译及安装的细节。

在安装和配置好之后，你可以运行 Ruby 脚本，也如同没有使用 mod\_ruby 一样，唯一的差是它运行得更快些。你还可以利用 mod\_ruby 提供额外的功能（例如与 Apache 请求处理的紧密集成）。

不过，你还有一些事情须小心。因为在请求之间解释器始终保持在内存中，它可能会处理多个应用的请求。这些应用的库可能会产生冲突（特别当不同的库包括同名类时）。你无法假定来自浏览器会话的请求序列都是由同一个解释器处理的——Apache 使用内部的算法来分配处理程序进程。

使用 FastCGI 协议可以解决部分问题。这是一个很有趣的 hack，不止对 Ruby，对所有 CGI 类型的程序都适用。它使用了一个非常简单的代理程序，通常作为 Apache 的一个模块。当请求到达时，这个代理会将它们转发到一个特定的、始终运行的进程，它像普通的 CGI 脚本那样进行响应。结果会返回给代理，然后发送回浏览器。FastCGI 和运行 mod\_ruby 有同样的优势，解释器总是在后台运行。对如何将请求分配给解释器，它还给了你更多的控制。你可以在 <http://www.fastcgi.com> 找到更多信息。

## 18.4 Web 服务器的选择

### Choice of Web Servers

- 现在，我们已经在 Apache Web 服务器下运行 Ruby 脚本。不过，Ruby 1.8 和之后的版本都捆绑了 WEBrick，一个灵活的、纯 Ruby 的 HTTP 服务器工具。基本上，它是一个基于插件的框架，让你可以编写服务器来处理 HTTP 请求及响应。下面是一个基本的 HTTP 服务器，负责文档和目录的索引。

```

#!/usr/bin/ruby
require 'webrick'
include WEBrick

s = HTTPServer.new(
 :Port => 2000,
 :DocumentRoot => File.join(Dir.pwd, "/html")
)
trap("INT") { s.shutdown }
s.start

```

HTTPServer 构造函数在端口 2000 上创建了一个新的 Web 服务器。代码设置文档目录为当前目录下的 html/ 子目录。然后在服务器开始运行之前调用 Kernel.trap，使其在遇到中断信号时，可以完整地关闭。如果用浏览器打开 <http://localhost:2000>，你将看到 html 子目录的列表。

WEBrick 并不仅限于支持这种静态内容的服务。你可以像 Java servlet 容器那样使用它。下面的代码架设了地址为 /hello 的一个简单的 servlet。当请求到达时，do\_GET 方法被调用。它使用响应对象来显示用户的 agent 信息，以及请求中的参数。

```

#!/usr/bin/ruby
require 'webrick'
include WEBrick

s = HTTPServer.new(:Port => 2000)
class HelloServlet < HTTPServlet::AbstractServlet
 def do_GET(req, res)
 res['Content-Type'] = "text/html"
 res.body = %{
 <html><body>
 Hello. You're calling from a #{req['User-Agent']}
 <p>
 I see parameters: #{req.query.keys.join(', ')}
 </body></html>
 }
 end
end
s.mount("/hello", HelloServlet)
trap("INT"){ s.shutdown }
s.start

```

关于 WEBrick 的更多信息可以在 <http://www.webrick.org> 中获得。你可以在那里找到许多有用的 servlet，包括可以让你用 Ruby 编写的 SOAP 服务器。

## 18.5 SOAP 及 Web Services

### SOAP and Web Services

1.8 谈到 SOAP, Ruby 目前也提供了 SOAP<sup>3</sup>的一种实现。它可以让你编写使用 Web Services 的服务器和客户端。这些应用可以从本地或跨网络远程访问, 这是它们的本性。SOAP 应用也不知道网络另一端是用何种语言实现的, 因此 SOAP 是将 Ruby 应用和其他语言 (例如 Java、Visual Basic 或 C++) 编写的应用进行互联的一种便捷方式。

SOAP 基本上是一种列集 (marshaling) 机制, 它使用 XML 在网络的两个节点间发送数据。它基本上被用来实现分布进程间的远程方法调用, RPC (remote procedure call)。SOAP 服务器发布一个或多个接口。这些接口是由数据类型以及使用这些类型的方法所定义的。然后 SOAP 客户端创建本地的代理, 通过 SOAP 协议连接到服务器的接口。对代理中方法的调用, 会被传递到服务器对应的接口上。服务器上方法产生的返回值通过代理传递回客户端。

让我们开始编写一个简单的 SOAP 服务: 编写一个计算收益的对象。最开始, 它提供了一种单独的方法, compound, 给定本金、收益率、每年收益复利的次数以及年数, 算出复利收益。为达到管理的目的, 我们跟踪这个方法调用了多少次, 并通过访问方法让外界可以得到这个计数。注意这个类就是一般的 Ruby 代码——它并不知道自己运行于 SOAP 环境中。

```
class InterestCalculator
 attr_reader :call_count
 def initialize
 @call_count = 0
 end
 def compound(principal, rate, freq, years)
 @call_count += 1
 principal*(1.0 + rate/freq)**(freq*years)
 end
end
```

现在我们让对象可以通过 SOAP 服务器来访问, 这将让客户端程序能够通过网络调用对象的方法。我们在这里使用独立的服务器, 在测试时非常方便, 因为我们可以从命令行启动它。你还可以将 Ruby SOAP 服务器作为 CGI 脚本或在 mod\_ruby 之下运行。

---

<sup>3</sup> SOAP 以前代表 Simple Object Access Protocol (简单对象访问协议)。当人们无法忍受这种反讽, 这种字母缩写的解释被丢弃了, 而 SOAP 就只是个名字了。

```

require 'soap/rpc/standaloneServer'
require 'interestcalc'

NS = 'http://pragprog.com/InterestCalc'
class Server2 < SOAP::RPC::StandaloneServer
 def on_init
 calc = InterestCalculator.new
 add_method(calc, 'compound', 'principal', 'rate', 'freq', 'years')
 add_method(calc, 'call_count')
 end
end

svr = Server2.new('Calc', NS, '0.0.0.0', 12321)
trap('INT') { svr.shutdown }
svr.start

```

这段代码定义了一个类，它实现了一个独立的 SOAP 服务器。当它初始化时，这个类创建了一个 `InterestCalculator` 对象（我们刚编写类的一个实例）。然后它使用 `add_method` 将这个类实现的两种方法 `compound` 和 `call_count` 加入到服务器中。最后，代码创建了服务器类的一个实例并运行之。构造函数的参数是应用的名字、默认的命名空间、接口使用的地址以及端口号。

其次，我们需要编写一些客户端代码来访问这个服务器。客户端为服务器上的 `InterestCalculator` 服务创建一个本地代理，加入它想要使用的方法，然后调用它们。

```

require 'soap/rpc/driver'
proxy = SOAP::RPC::Driver.new("http://localhost:12321",
 "http://pragprog.com/InterestCalc")
proxy.add_method('compound', 'principal', 'rate', 'freq', 'years')
proxy.add_method('call_count')
puts "Call count: #{proxy.call_count}"
puts "5 years, compound annually: #{proxy.compound(100, 0.06, 1, 5)}"
puts "5 years, compound monthly: #{proxy.compound(100, 0.06, 12, 5)}"
puts "Call count: #{proxy.call_count}"

```

要测试它，我们可以在一个终端窗口运行服务器（下面的输出已经稍作排版，以适应页面的宽度）。

```

% ruby server.rb
I, [2004-07-26T10:55:51.629451 #12327] INFO
 -- Calc: Start of Calc.
I, [2004-07-26T10:55:51.633755 #12327] INFO
 -- Calc: WEBrick 1.3.1
I, [2004-07-26T10:55:51.635146 #12327] INFO
 -- Calc: ruby 1.8.2 (2004-07-26) [powerpc-darwin]
I, [2004-07-26T10:55:51.639347 #12327] INFO
 -- Calc: WEBrick::HTTPServer#start: pid=12327 port=12321

```

最后，我们在另一个窗口中运行客户端。

```
% ruby client.rb
Call count: 0
5 years, compound annually: 133.82255776
5 years, compound monthly: 134.885015254931
Call count: 2
```

看起来不错！运行成功，然后我们调用所有方法再运行一次。

```
% ruby client.rb
Call count: 2
5 years, compound annually: 133.82255776
5 years, compound monthly: 134.885015254931
Call count: 4
```

注意当我们第二次运行客户端时，调用次数从 2 开始。服务器创建了单独一个 `InterestCalculator` 对象来服务进入的请求，且这个对象为每个请求所重用。

### 18.5.1 SOAP 和 Google

#### SOAP and Google

明显地，SOAP 的真正好处是，让你可以和其他的 Web 服务交互。作为一个示例，让我们编写一些 Ruby 代码向 Google 的 Web API 发送查询。

在我们向 Google 发送查询之前，你需要一个开发者密钥。请到 <http://www.google.com/apis>，按照步骤 2 “创建一个 *Google* 账号”的指导。在你填写 email 地址并提供一个口令之后，Google 会将一个开发者密钥发送给你。在下面的示例中，我们假定你已经将这个密钥保存在主目录的 `.google_key` 文件中。

让我们从基础级开始。查看 Google API 方法 `doGoogleSearch` 的文档，我们发现它有 10 (!) 个参数。

<code>key</code>	开发者密钥
<code>q</code>	查询字符串
<code>start</code>	结果起始的索引
<code>maxResults</code>	每次查询返回的最大结果个数
<code>filter</code>	如果开启，压缩结果，这样类似的页面或者同一领域的页面只显示一次
<code>restrict</code>	限制搜索为 Google Web 索引的一个子集
<code>safeSearch</code>	如果开启，从结果中删除可能的成人内容
<code>lr</code>	限制搜索指定语言集合的文档
<code>ie</code>	忽略（输入编码）
<code>oe</code>	忽略（输出编码）

我们可以使用 `add_method` 调用为 `doGoogleSearch` 方法构造一个 SOAP 代理。下面的示例就是这样做的，假设你向 Google 搜索单词 *pragmatic*，打印返回的第一项。

```

require 'soap/rpc/driver'
require 'cgi'
endpoint = 'http://api.google.com/search/beta2'
namespace = 'urn:GoogleSearch'
soap = SOAP::RPC::Driver.new(endpoint, namespace)
soap.add_method('doGoogleSearch', 'key', 'q', 'start',
 'maxResults', 'filter', 'restrict',
 'safeSearch', 'lr', 'ie', 'oe')
query = 'pragmatic'
key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
result = soap.doGoogleSearch(key, query, 0, 1, false, nil,
 false, nil, nil, nil)
printf "Estimated number of results is %d.\n",
 result.estimatedTotalResultsCount
printf "Your query took %.6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.URL
puts CGI.unescapeHTML(first.snippet)

```

运行它，你会看到类似下面的输出（注意查询的单词是如何被 Google 加亮的）。

```

Estimated number of results is 550000.
Your query took 0.123762 seconds.
The Pragmatic Programmers, LLC
http://www.pragmaticprogrammer.com/
Home of Andrew Hunt and David Thomas's best-selling book 'The
Pragmatic Programmer'
 and The 'Pragmatic Starter Kit
(tm)' series. ... The Pragmatic Bookshelf TM. ...

```

不过 SOAP 支持动态地发现服务器上对象的接口。这是使用 WSDL (Web Services Description Language) 来完成的。WSDL 文件是一个 XML 文档，描述了一个 Web 服务接口的类型、方法以及访问的机制。SOAP 客户端可以读取 WSDL 文件来自动创建访问服务器的接口。

Google 在 <http://api.google.com/GoogleSearch.wsdl> 发布了其接口的 WSDL 描述。你可以修改上面的搜索应用来读取这个 WSDL，消除了显式添加 doGoogleSearch 方法的需要。

```

require 'soap/wsdlDriver'
require 'cgi'
WSDL_URL = "http://api.google.com/GoogleSearch.wsdl"
soap = SOAP::WSDLDriverFactory.new(WSDL_URL).createDriver
query = 'pragmatic'
key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
result = soap.doGoogleSearch(key, query, 0, 1, false,
 nil, false, nil, nil)

```

```

printf "Estimated number of results is %d.\n",
 result.estimatedTotalResultsCount
printf "Your query took %6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.URL
puts CGI.unescapeHTML(first.snippet)

```

最后，我们可以使用 Ian Macdonald 的 Google 库（可以从 RAA 得到）来更进一步。它将 Web 服务 API 封装到一个漂亮的接口后面（原因无外乎是它消除了所有额外的参数）。这个库还有方法来构造 Google 查询的数据范围和其他限制，并提供了 Google 缓存和拼写检查功能的接口。下面的代码使用 Ian 库来完成我们的“pragmatic”搜索。

```

require 'google'
require 'cgi'

key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
google = Google::Search.new(key)
result = google.search('pragmatic')
printf "Estimated number of results is %d.\n",
 result.estimatedTotalResultsCount
printf "Your query took %6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.url
puts CGI.unescapeHTML(first.snippet)

```

## 18.6 更多信息

### More Information

Ruby 的 Web 编程是一个很大的课题。要深入研究，你可能需要查看 “*The Ruby Way*” [Ful01] 的第 9 章，你可以在那里找到网络和 Web 编程的更多示例；查看 “*The Ruby Developer's Guide*” [FJN02] 的第 6 章，你可以在其中找到关于组织 CGI 应用的优秀示例，以及 Iowa 的一些示例代码。

如果 SOAP 让你感到有些复杂，你可以考察使用 XML-RPC，在第 757 页有简略的描述。

Ruby Web 开发框架在网络上有很多。这是一个动态发展的区域：新的竞争者持续地出现，难以在一本印刷的书中完整而准确地讲解。不过，Ruby 社区中的两个框架正在吸引大量的人气。<sup>4</sup>

- Rails (<http://www.rubyonrails.org>)，and
- CGIKit ([http://www.spice-of-life.net/cgikit/index\\_en.html](http://www.spice-of-life.net/cgikit/index_en.html))

---

<sup>4</sup> 译注：特别是 Rails，可以说是 Rails 的火热带动了 Ruby 的学习浪潮。



# Ruby Tk

## Ruby Tk

Ruby 应用程序归档（Application Archive）含有几个提供图形用户界面（GUI）的扩展，包括对 Fox, GTK 及其他一些部件库（widget）的支持。

Tk 扩展被集成到了 Ruby 主发行版中，既可以运行在 Unix 上，也可以运行在 Windows 系统上。要使用它，需要在系统上先安装 Tk。Tk 是一个很大的系统，而且有专门的书籍介绍它，所以在这里我们不会花时间和资源来深究 Tk 本身，而是集中精力于如何从 Ruby 中访问 Tk 的特性。你可能需要一本参考书以更有效地使用 Tk 和 Ruby。我们用的绑定（binding）和 Perl 绑定非常相近，所以你可能想搞到一本 *Learning Perl/Tk* [Wal99] 或者 *Perl/Tk Pocket Reference* [Lid98]。

TK 以一种组合模型（composition model）工作——也就是说，你开始先创建一个容器（例如一个 TkFrame 或 TkBoot），然后在其中创建部件（widget，GUI 组件的另一种说法），例如按钮或者标签（label）。当你准备好启动 GUI 时，调用 Tk.mainloop。最后 Tk 引擎会获得程序的控制权，显示各个部件并根据 GUI 事件调用相应的代码。

### 19.1 简单的 Tk 应用程序

#### Simple Tk Application

在 Ruby 中一个简单的 Tk 应用程序看起来如下所示：

```
require 'tk'
root = TkRoot.new { title "Ex1" }
TkLabel.new(root) do
 text 'Hello, World!'
 pack('padx' => 15, 'pady' => 15, 'side' => 'left')
end
Tk.mainloop
```



让我们来详细看看这段代码。装载了 Tk 扩展模块后，我们使用 TkRoot.new 创建一个根窗框（frame）。然后创建一个 TkLabel 部件作为根窗框的子部件，并为这个标签

<sup>1</sup> 译注：截至本书出版时，Ruby 1.8.5 Windows 版本的 One-Click Installer 安装之后，Tk 可能无法正常运作，运行 require “tk” 可能会报告找不到 DLL 或其他错误信息。这似乎是此版本的一个 bug。解决此问题的方法之一是安装 ActiveTcl ([www.activestate.com](http://www.activestate.com))。

设置几个选项。最后组装（pack）好根窗框，并进入 GUI 的主事件循环。

明确指定根窗框是个好习惯，但是你也可以省略它——以及额外的选项——由此可以将上面的代码缩减为 3 行。

```
require 'tk'
TkLabel.new { text 'Hello, World!'; pack }
Tk.mainloop
```

Tk 程序也不过如此而已！再来一本本章开头推荐的 Perl/Tk 专著，你就可以立即构建你所需要的高级 GUIs 了。但是如果你想深入了解更多的细节，那就接着读吧。

## 19.2 部件 Widgets

创建部件很容易。使用 Tk 文档中给出的部件的名字，并在其前面加“Tk”即可。例如，部件 Label、Button 和 Entry 变成了类 TkLabel、TkButton 和 TkEntry。像创建任何其他对象一样，使用 new 方法可以创建部件实例。如果不为给定的部件指定一个父部件，那么父部件默认为根窗框。通常我们需要指定部件的父亲，以及其他一些选项——颜色、大小等等。当程序运行的时候，我们也要通过设定回调函数（当某种事件发生时调用的例程）和共享数据以从部件中取回信息。

### 19.2.1 设置部件选项 Setting Widget Options

如果你看一下 Tk 参考手册（例如 Perl/Tk 手册），你会注意到部件的选项列表通常带一个连字符——就像命令行选项那样。在 Perl/Tk 中，选项被传递给部件的一个 Hash 中。在 Ruby 中你也可以这样做，但也可以使用代码 block 来传递选项：在 block 中选项的名字被用作方法名，选项的参数作为方法的参数。部件以它的父部件作为第一个参数，后跟可选的选项 hash 或者可选的选项 block。这样，下面的两种形式是等价的。

```
TkLabel.new(parent_widget) do
 text 'Hello, World!'
 pack('padx' => 5,
 'pady' => 5,
 'side' => 'left')
end
or
TkLabel.new(parent_widget, 'text' => 'Hello, World!).pack(...)
```

使用代码 `block` 形式时有一点需要注意：变量的作用域并不是你所想的那样。`Block` 实际不是在调用者上下文而是在部件对象上下文中被处理的。这意味着不能在 `block` 中访问调用者的实例变量，但是可以访问外围作用域中的局部变量和全局变量。我们会在后面的例子中展示如何使用这两种方式来处理选项。

距离（譬如例子中的 `padx` 和 `pady` 选项）的单位默认为像素，但可以使用不同的后缀来表示不同的单位：`c`（厘米），`i`（英寸），`m`（毫米）或者`p`（点）。例如“`12p`”表示 12 个像素点。

## 19.2.2 获取部件数据

### Getting Widget Data

通过使用回调函数和绑定变量，我们可以从部件中获取信息。

回调函数很容易设置。`command` 选项（以下例子的 `TkButton` 调用中）以 `Proc` 对象为参数，当回调函数被触发时，该对象将被调用。这里我们将与方法相关联的 `block` 作为传入的 `proc`，但我们也同样可以使用 `Kernel.lambda` 来显式生成一个 `Proc` 对象。

```
require 'tk'
TkButton.new do
 text "EXIT"
 command { exit }
 pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

我们还可以使用 `TkVariable` 代理，将某个 Ruby 变量绑定到 Tk 部件的值。这样的安排使得每当部件的值发生变化时，Ruby 变量也会自动被更新，而当变量发生变化时，部件也会改变以反映新的值。

下面的例子显示了这一点。注意 `TkCheckButton` 是如何设置的；文档中说 `variable` 选项以 *var reference* 作为参数。因此，我们使用 `TkVariable.new` 创建一个 Tk 变量的引用。根据 `checkbox` 是否选中，`mycheck.value` 将返回字符串“0”或者“1”。同样的机制可以应用到任意支持变量引用的 Tk 部件中，例如单选按钮和文本域。

```
require 'tk'
packing = { 'padx'=>5, 'pady'=>5, 'side' => 'left' }
checked = TkVariable.new
def checked.status
 value == "1" ? "Yes" : "No"
end
```

```

status = TkLabel.new do
 text checked.status
 pack(packing)
end

TkCheckButton.new do
 variable checked
 pack(packing)
end

TkButton.new do
 text "Show status"
 command { status.text(checked.status) }
 pack(packing)
end

Tk.mainloop

```

### 19.2.3 动态地设置/获取选项

#### Setting/Getting Options Dynamically

除了在创建部件时可以设置选项外，还可以在运行时重新配置部件。每个部件都支持 `configure` 方法，它和 `new` 一样以一个 Hash 对象或者代码 block 作为参数。我们可以修改第一个例子，使得点击按钮时改变标签文本。

```

require 'tk'
root = TkRoot.new { title "Ex3" }

top = TkFrame.new(root) { relief 'raised'; border 5 }

lbl = TkLabel.new(top) do
 justify 'center'
 text 'Hello, World!'
 pack('padx'=>5, 'pady'=>5, 'side' => 'top')
end

TkButton.new(top) do
 text "Ok"
 command { exit }
 pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end

TkButton.new(top) do
 text "Cancel"
 command { lbl.configure('text'=>"Goodbye, Cruel World!") }
 pack('side'=>'right', 'padx'=>10, 'pady'=>10)
end

top.pack('fill'=>'both', 'side' =>'top')

Tk.mainloop

```

现在每当点击 Cancel 按钮时，标签中文本立即从 “Hello, World!” 变成 “Goodbye, Cruel World!”

你也可以使用 `cget` 来查询部件的具体选项。

```

require 'tk'
b = TkButton.new do
 text "OK"
 justify "left"
 border 5
end
b.cget('text') → "OK"
b.cget('justify') → "left"
b.cget('border') → 5

```

## 19.2.4 示例应用程序

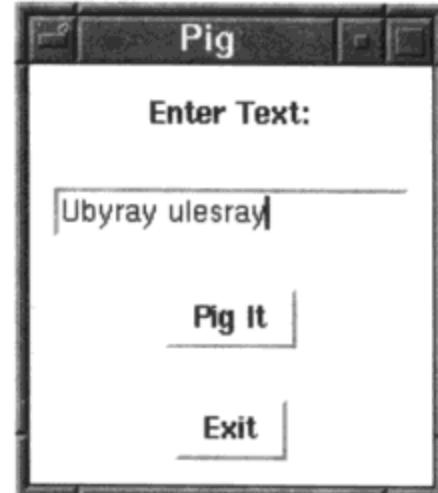
### Sample Application

下面的例子稍微有点长，它显示了一个真实的应用程序——一个 pig latin<sup>2</sup>生成器。输入 **Ruby rules**，然后点击 **Pig It** 按钮会立即把它翻译成 pig latin。

```

require 'tk'
class PigBox
 def pig(word)
 leading_cap = word =~ /^[A-Z]/
 word.downcase!
 res = case word
 when /^[aeiouy]/
 word+"way"
 when /^([aeiouy]+)(.*)/
 $2+$1+"ay"
 else
 word
 end
 leading_cap ? res.capitalize : res
 end
 def show_pig
 @text.value = @text.value.split.collect{|w| pig(w)}.join(" ")
 end
 def initialize
 ph = { 'padx' => 10, 'pady' => 10 } # common options
 root = TkRoot.new { title "Pig" }
 top = TkFrame.new(root) { background "white" }
 TkLabel.new(top) { text 'Enter Text:' ; pack(ph) }
 @text = TkVariable.new
 TkEntry.new(top, 'textvariable' => @text).pack(ph)
 pig_b = TkButton.new(top) { text 'Pig It'; pack ph}
 pig_b.command { show_pig }
 exit_b = TkButton.new(top) { text 'Exit'; pack ph}
 exit_b.command { exit }
 top.pack('fill'=>'both', 'side' =>'top')
 end
end
PigBox.new
Tk.mainloop

```



<sup>2</sup> 译注：故意颠倒英语字母顺序拼凑而成的行话。

## 布局管理

### Geometry Management

在本章的例子中，你会看到多次调用部件方法 `pack`。这是一个非常重要的方法调用——如果不使用它，你就见不到任何部件。`pack` 命令告诉布局管理器会根据我们指定的条件来布局部件。布局管理器能识别 3 个命令：

命 令	位置说明
-----	------

<code>pack</code>	灵活的，基于限制的位置
-------------------	-------------

<code>place</code>	绝对位置
--------------------	------

<code>grid</code>	表格（行/列）位置
-------------------	-----------

因为 `pack` 是最常用的命令，所以我们在例子中用了它。

## 19.3 绑定事件

### Binding Events

我们的部件暴露在真实的世界中；它们会被点击，鼠标会在其上移过，也可以用 tab 键来遍历；所有这些操作，以及更多的操作都会产生我们能够捕获的事件。你可以使用部件的 `bind` 方法创建一个从特定部件事件到代码 block 的绑定。

例如，假设我们创建了一个显示图像的按钮部件。当用户鼠标越过按钮时，我们希望改变图像。

```
require 'tk'

image1 = TkPhotoImage.new { file "img1.gif" }
image2 = TkPhotoImage.new { file "img2.gif" }

b = TkButton.new(@root) do
 image image1
 command { exit }
 pack
end

b.bind("Enter") { b.configure('image'=>image2) }
b.bind("Leave") { b.configure('image'=>image1) }

Tk.mainloop
```

首先，使用 `TkPhotoImage` 从磁盘上的文件创建两个 GIF 图像对象；接着创建一个按钮（很明智命名为“`b`”）来显示图像 `image1`；最后我们绑定 `Enter` 事件以使得当鼠标移到按钮上时动态地改变按钮显示的图像为 `image2`，而当鼠标移离按钮时恢复成 `image1`。

这个例子展示了 Enter 和 Leave 两个简单的事件。但在 bind 函数的参数中，事件名字可以被短线分割为几个子串的组成部分，顺序为修饰符-修饰符-类型-细节 (*modifier-modifier-type-detail*)。Tk 参考中列出了修饰符，包括 Button1, Control, Alt, Shift 等等。类型是事件的名字（使用 X11 中的命名传统），包括 ButtonPress, KeyPress 和 Expose。细节可以是 1 到 5 的按钮号，也可以是键盘输入的 keysym。例如，当按下 control 键并释放鼠标的 button 1 时，触发的绑定可以记为：

```
Control-Button1-ButtonRelease
```

或者

```
Control-ButtonRelease-1
```

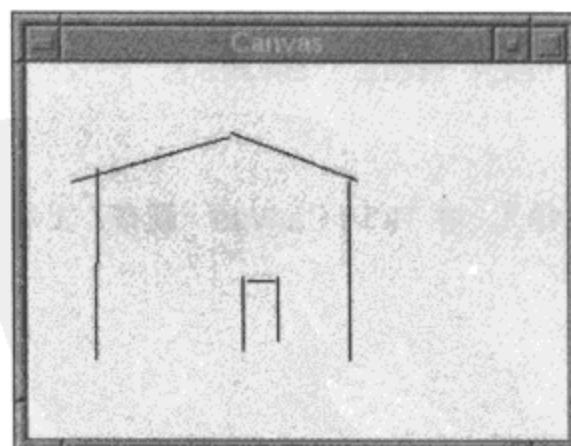
事件本身可以含有一些域 (field)，如事件发生的时间、*x* 和 *y* 位置等。bind 可以使用 *event filed code* 将这些信息传递给回调函数。使用它们的方式类似于 printf 规范。例如，为了获得鼠标移动的 *x* 和 *y* 坐标，你需要使用 3 个参数调用 bind。第二个参数是回调函数的 Proc，而第三个参数是事件域字符串。

```
canvas.bind("Motion", lambda {|x, y| do_motion (x, y)}, "%x %y")
```

## 19.4 画布 Canvas

Tk 提供了一个 Canvas 部件，使用它你可以绘制并生成 PostScript 输出。下一页的图 19.1 显示了绘制直线的一小段简单代码（根据发行版的代码改写的）。按下 button 1 开始绘制直线，当你四处移动鼠标时，会看到一条线，当你释放 button 1 时，直线将被绘制下来。

点击几下鼠标，你就有一个即兴杰作了。



如他们所言，“因为我们找不到画家，所以就停止继续画了……”

```

require 'tk'

class Draw
 def do_press(x, y)
 @start_x = x
 @start_y = y
 @current_line = TkcLine.new(@canvas, x, y, x, y)
 end

 def do_motion(x, y)
 if @current_line
 @current_line.coords @start_x, @start_y, x, y
 end
 end

 def do_release(x, y)
 if @current_line
 @current_line.coords @start_x, @start_y, x, y
 @current_line.fill 'black'
 @current_line = nil
 end
 end

 def initialize(parent)
 @canvas = TkCanvas.new(parent)
 @canvas.pack
 @start_x = @start_y = 0
 @canvas.bind("1", lambda {|e| do_press(e.x, e.y)})
 @canvas.bind("B1-Motion",
 lambda {|x, y| do_motion(x, y)}, "%x %y")
 @canvas.bind("ButtonRelease-1",
 lambda {|x, y| do_release(x, y)},
 "%x %y")
 end
end

root = TkRoot.new { title 'Canvas' }
Draw.new(root)
Tk.mainloop

```

图 19.1 在 Tk 的 Canvas（画布）上绘制

## 19.5 滚动

### Scrolling

除非你打算绘制一些非常小的图，否则前面的例子没有太大用处。TkCanvas, TkListbox 和 TkText 都可以被设置成使用滚动条，这样你就可以在一幅“巨画”的一小部分上工作了。

滚动条和部件之间的通信是双向的。移动滚动条意味着部件视图将改变；而当通过其他方式改变部件视图时，滚动条也会相应地移动以反映新的位置。

由于我们还没有使用过列表，下面将我们的滚动条例子使用可滚动的文本列表。在下面的代码片断里，我们先创建了一个简单的 TkListbox 和与之相关联的 TkScrollbar。滚动条的回调函数（使用 command 设置）将会调用列表部件的 yview 方法，该方法会改变列表 y 方向上可见部分的值。

设置完回调函数后，我们再建立逆向关联：当列表需要滚动时，我们使用 TkScrollbar#set 函数设置滚动条的适当范围。在下一节功能完整的例子中我们还会使用这段代码。

```
list_w = TkListbox.new(frame) do
 selectmode 'single'
 pack 'side' => 'left'
end

list_w.bind("ButtonRelease-1") do
 busy do
 filename = list_w.get(*list_w.curselection)
 tmp_img = TkPhotoImage.new { file filename }
 scale = tmp_img.height / 100
 scale = 1 if scale < 1
 image_w.copy(tmp_img, 'subsample' => [scale, scale])
 image_w.pack
 end
end

scroll_bar = TkScrollbar.new(frame) do
 command { |*args| list_w.yview *args }
 pack 'side' => 'left', 'fill' => 'y'
end

list_w.yscrollcommand { |first, last| scroll_bar.set(first, last) }
```

#### 19.5.1 还有一件事

Just One More Thing

我们还可以继续介绍 Tk 几百页，但那将会成为另一本书。下面的例子是我们最终的 Tk 例子——一个简单的 GIF 图像查看器。你可以从滚动列表中选择一个 GIF 图像文件名，然后其缩略图将会被显示出来。下面我们将再指出几点。

你曾经见过创建了“忙光标”而忘记重置它的应用程序吗？在 Ruby 中有一个优雅的技巧可以避免此类事情发生。还记得 `File.new` 是怎么使用 `block` 来确保用完文件后将其关闭的吗？如下面例子所示，我们可以使用 `busy` 方法做类似的事情。

这个例子也演示了 `TkListbox` 的一些基本操作——向列表中添加元素，为鼠标按钮释放事件<sup>3</sup>添加回调函数，以及查询当前的选择。

迄今为止，我们使用 `TkPhotoImage` 来直接显示图像，但是你也可缩放、二次抽样或仅显示部分图像。这里我们使用二次抽样来缩小图像以方便查看。

```
require 'tk'

class GifViewer
 def initialize(filelist)
 setup_viewer(filelist)
 end

 def run
 Tk.mainloop
 end

 def setup_viewer(filelist)
 @root = TkRoot.new {title 'Scroll List'}
 frame = TkFrame.new(@root)

 image_w = TkPhotoImage.new
 TkLabel.new(frame) do
 image image_w
 pack 'side'=>'right'
 end

 list_w = TkListbox.new(frame) do
 selectmode 'single'
 pack 'side' => 'left'
 end

 list_w.bind("ButtonRelease-1") do
 busy do
 filename = list_w.get(*list_w.curselection)
 tmp_img = TkPhotoImage.new { file filename }
 scale = tmp_img.height / 100
 scale = 1 if scale < 1
 image_w.copy(tmp_img, 'subsample' => [scale, scale])
 image_w.pack
 end
 end
 end
end
```



<sup>3</sup> 因为只有当鼠标释放时才算作选中部件，所以你需要按钮释放事件而不是按下事件。

```

filelist.each do |name|
 list_w.insert('end', name) # Insert each file name into the list
end

scroll_bar = TkScrollbar.new(frame) do
 command {[*args] list_w.yview *args }
 pack 'side' => 'left', 'fill' => 'y'
end

list_w.yscrollcommand {|first,last| scroll_bar.set(first,last) }
frame.pack
end

Run a block with a 'wait' cursor
def busy
 @root.cursor "watch" # Set a watch cursor
 yield
ensure
 @root.cursor "" # Back to original
end
end

viewer = GifViewer.new(Dir["screenshots/gifs/*.gif"])
viewer.run

```

## 19.6 从 Perl/Tk 文档转译

### Translating from Perl/Tk Documentation

就是这么一回事，现在你可以独立了。很大程度上，你可以很容易地将 Perl/Tk 文档转译为 Ruby 文档。但也有几个异常：某些方法没有实现，也可能某些额外的功能没有文档。在 Ruby/Tk 书籍面世之前，最好的方法是在邮件列表中询问或者阅读源代码。

但是通常很容易看出端倪来。记住选项可能以散列表的方式或者代码 block 的风格出现，并且代码 block 的作用域是在使用它的 TkWidget 内而不是类的实例内。

#### 19.6.1 对象创建

##### Object Creation

在 Perl/Tk 世界中，父部件负责创建它的子部件。而在 Ruby 中，父部件被作为第一个参数传递给子部件的构造函数。

Perl/Tk:	<code>\$widget = \$parent-&gt;Widget( [ option =&gt; value ] )</code>
Ruby:	<code>widget = TkWidget.new(parent, option-hash)</code>
	<code>widget = TkWidget.new(parent) { code block }</code>

有时你可能不需要保存新建部件的返回值，但如果需要的话，它就在那儿。不要忘了布局部件（或者使用其他的几何位置函数），否则它将不会被显示出来。

## 19.6.2 选项

### Options

```
Perl/Tk: -background => color
Ruby: 'background' => color
 { background color }
```

记住代码 block 的作用域是不同的。

## 19.6.3 变量引用

### Variable References

```
Perl/Tk: -textvariable => \$variable
 -textvariable => varRef
Ruby: ref = TkVariable.new
 'textvariable' => ref
 { textvariable ref }
```

使用 `TkVariable` 将一个 Ruby 变量和一个部件的属性值关联起来。然后你可以使用 `TkVariable` 中的值访问方法 (`TkVariable#value` 和 `TkVariable#value=`) 来直接更改部件的内容。

## Ruby 和微软 Windows 系统

## Ruby and Microsoft Windows

Ruby 运行在不同的环境中。其中一些环境是基于 Unix 的，而另外一些是基于不同版本的微软 Windows 系统的。Ruby 来自于以 Unix 为中心的人，但是这些年来它也已经在 Windows 世界开发了许多有用的特性。本章我们会介绍这些特性，同时分享在 Windows 下有效地使用 Ruby 的一些秘密。

### 20.1 得到 Ruby for Windows

#### Getting Ruby for Windows

两种版本的 Ruby 在 Windows 环境下可用。

第一种是运行在本地的 Ruby 版本，即它只是另外一个 Windows 程序。得到这个发行版最简单的方式是运行 One-Click Installer，它会将已经编译好的二进制发行版安装到机器上。在 <http://rubyinstaller.rubyforge.org/> 页面可以得到它的最新版本。

如果你更喜欢冒险，或者需要编译那些未在二进制发行中提供的库，可以从源文件编译 Ruby。你需要微软的 VC++ 编译器和相关工具来完成编译。从 <http://www.ruby-lang.org> 下载 Ruby 的源代码，或使用 CVS 检出（check out）最新的开发版本。然后阅读 win32\README.win32 文件。

第二种版本使用称为 Cygwin 的仿真层，它在 Windows 系统上提供了类似 Unix 的环境。Cygwin 版本的 Ruby 与运行在 Unix 平台上的 Ruby 最接近，但是运行它意味着也必须要安装 Cygwin。如果你想采取这种途径，可以从 <http://ftp.ruby-lang.org/pub/ruby/binaries/cygwin/> 下载 Cygwin 版本的 Ruby。当然你也需要 Cygwin 本身。下载页面中有一个链接指向所需动态链接库（DLL），或者可以到 <http://www.cygwin.com> 下载整个包（但是请小心点：需要确保得到的版本和下载的 Ruby 兼容）。

- 1.8 该选择哪个版本的 Ruby 呢？当本书第 1 版推出时，Cygwin 版本的 Ruby 是首选的发行版本。但是现在情况发生了变化：随着时间的推移，Ruby 的本地（native）版本具有越来越多的功能，以至于它现在成了我们在 Windows 系统上的首选。

## 20.2 在 Windows 下运行 Ruby

### Running Ruby Under Windows

在 Ruby 的 Windows 发行版中，你会发现两个可执行文件。

`ruby.exe` 用于命令行提示（DOS shell），这和 Unix 版本中的 Ruby 可执行程序是一样的。对于读写标准输入和输出的程序来说，是挺不错的。但是，这也意味着，任何时候运行 `ruby.exe`，你都会得到一个 DOS shell，即使你不需要它——Windows 也会创建新的命令行提示窗口，并当 Ruby 正在运行时显示它。某些时候可能这不是恰当的行为，比如，如果双击了使用图形界面接口（如 Tk）的一个 Ruby 脚本，或者你要以后台任务方式或者从其他程序内部运行 Ruby 脚本时。

在这些情况下，你可能希望使用 `rubyw.exe`。它与 `ruby.exe` 一样，除了不会提供标准输入、标准输出或标准错误外，并且当运行时也不会启动 DOS shell。

安装程序会默认设置文件关联，让`.rb` 后缀的文件自动使用 `rubyw.exe`。这样可以双击 Ruby 脚本，它们只会运行程序而不会同时提示 DOS shell。

## 20.3 Win32API

如果你计划要编写的 Ruby 程序，需要直接使用一些 Windows 32 API 函数，或者需要使用其他 DLL 中的某些入口函数（entry point），我们有好消息告诉你——使用 Win32API 库。

作为一个例子，这里的代码来自于一个更大的 Windows 应用，我们的预约实施系统使用这个应用来下载并打印发票及收据。Web 应用生成 PDF 文件，运行在 Windows 上的 Ruby 脚本会把它下载到本地文件中。脚本使用 Windows 中的 shell 命令 `print` 来打印这个文件。

```
arg = "ids=#{resp.intl_orders.join(',')}"
fname = "/temp/invoices.pdf"
site = Net::HTTP.new(HOST, PORT)
site.use_ssl = true
http_resp, = site.get2("/fulfill/receipt.cgi?" + arg,
 'Authorization' => 'Basic ' +
 ["name:passwd"].pack('m').strip)
```

```

File.open(fname, "wb") {|f| f.puts(http_resp.body) }
shell = Win32API.new("shell32", "ShellExecute",
 ['L', 'P', 'P', 'P', 'P', 'L'], 'L')
shell.Call(0, "print", fname, 0, 0, SW_SHOWNORMAL)

```

这里创建了一个 `Win32API` 对象，它表示对特定 DLL 入口函数的调用，这是由函数名称、含有该函数的 DLL 名称以及函数原型特征（参数类型和返回类型）来指定的。在前面的例子中，`shell` 变量包装（wrap）了 `shell32` DLL 中 `ShellExecute` 这个 Windows 函数。它接受 6 个参数（1 个数字，4 个字符串指针和 1 个数字）并返回 1 个数字（这些参数类型将在第 755 页描述）。我们也可以调用这个返回的对象去打印下载的文件。

DLL 函数的许多参数是某种形式的二进制结构。`Win32API` 通过使用 Ruby 的 `String` 对象来回传递这些二进制数据。必要的话你需要对这些字符串打包和拆包（`pack` 和 `unpack`，参见第 755 页的例子）。

## 20.4 Windows 自动化

### Windows Automation

如果对摆弄这些底层的 Windows API 不感兴趣，也许你对 Windows 自动化会有兴趣——Masaki Suketa 编写的一个称为 `WIN32OLE` 的 Ruby 扩展，使得可以把 Ruby 当作 18. Windows 自动化的一个客户程序来使用。`Win32OLE` 是标准 Ruby 发行版的一部分。

Windows 自动化允许自动化控制器（automation controller，亦即客户端程序）对自动化服务器发出命令和查询，自动化服务器可能是微软 Excel、Word、PowerPoint 等等。

若要执行自动化服务器中的方法，可以通过从 `WIN32OLE` 对象中调用相同名称的方法来实现。比如，可以创建一个新的 `WIN32OLE` 客户程序，运行全新的 Internet Explore (IE) 拷贝，并命令它访问其主页。

```

ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = true
ie.gohome

```

也可以让它浏览特定的网页。

```

ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = true
ie.navigate("http://www.pragmaticprogrammer.com")

```

那些 `WIN32OLE` 不知道的方法（比如 `visible`, `gohome` 或 `navigate`）会被传递到 `WIN32OLE#invoke` 方法，会将正确的命令发送给服务器。

### 20.4.1 得到和设置属性

#### Getting and Setting Properties

使用普通的 Ruby 散列表表示法可以从服务器中设置和得到属性。比如，为了在 Excel 图表中设置 Rotation 属性，可以这样写：

```
excel = WIN32OLE.new("excel.application")
excelchart = excel.Charts.Add()
...
excelchart['Rotation'] = 45
puts excelchart['Rotation']
```

OLE 对象的参数作为 WIN32OLE 对象的属性会被自动设置。这意味着可以通过对一个对象属性进行赋值来设置参数。

```
excelchart.rotation = 45
r = excelchart.rotation
```

下面的例子是修改后的示例文件 `excel.rb`（可在 `ext/win32/samples` 目录中找到）。它先启动 Excel，创建一个图表，然后在屏幕上旋转它。Pixar<sup>1</sup>，当心了！

```
require 'win32ole'
- 4100 is the value for the Excel constant xl3DColumn.
ChartTypeVal = - 4100;
excel = WIN32OLE.new("excel.application")
Create and rotate the chart
excel['Visible'] = TRUE
excel.Workbooks.Add()
excel.Range("a1")['Value'] = 3
excel.Range("a2")['Value'] = 2
excel.Range("a3")['Value'] = 1
excel.Range("a1:a3").Select()
excelchart = excel.Charts.Add()
excelchart['Type'] = ChartTypeVal
30.step(180, 5) do |rot|
 excelchart.rotation = rot
 sleep(0.1)
end
excel.ActiveWorkbook.Close(0)
excel.Quit()
```

### 20.4.2 命名参数

#### Named Arguments

其他自动化客户语言如 Visual Basic 有命名参数这个概念。假设有一个 Visual Basic 函数，它有如下标签：

```
Song(artist, title, length): rem Visual Basic
```

---

<sup>1</sup> 译注：美国著名的动画制片公司。

与顺序指定所有 3 个参数相反，你可以使用命名参数来调用它：

```
Song title := 'Get It On': rem Visual Basic
```

这等同于 `Song(nil, 'Get It on', nil)` 调用。

在 Ruby 中，可以通过传递带有命名参数的散列表来使用这个特性。

```
Song.new('title' => 'Get It On')
```

### 20.4.3 for each

Visual Basic 有“for each”语句，它对服务器中项（item）的收集（collection）进行迭代，WIN32OLE 对象有 `each` 方法（它接受一个代码块）去完成同样的事情。

```
require 'win32ole'
excel = WIN32OLE.new("excel.application")
excel.Workbooks.Add
excel.Range("a1").Value = 10
excel.Range("a2").Value = 20
excel.Range("a3").Value = "=a1+a2"
excel.Range("a1:a3").each do |cell|
 p cell.Value
end
```

### 20.4.4 事件

#### Events

以 Ruby 编写的自动化客户程序可以注册自己去接收来自其他程序的事件。这是使用 `WIN32OLE_EVENT` 类来实现的。这个例子（基于 Win32OLE 0.1.1 发布中的代码）演示了使用事件槽（event sink）记录用户使用 Internet Explorer 浏览过的 URLs。

```
require 'win32ole'
$urls = []
def navigate(url)
 $urls << url
end
def stop_msg_loop
 puts "IE has exited..."
 throw :done
end
def default_handler(event, *args)
 case event
 when "BeforeNavigate"
 puts "Now Navigating to #{args[0]}..."
 end
end
```

```

ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = TRUE
ie.gohome
ev = WIN32OLE_EVENT.new(ie, 'DWebBrowserEvents')
ev.on_event { |*args| default_handler(*args)}
ev.on_event("NavigateComplete") { |url| navigate(url)}
ev.on_event("Quit") { |*args| stop_msg_loop}
catch(:done) do
 loop do
 WIN32OLE_EVENT.message_loop
 end
end
puts "You Navigated to the following URLs: "
$urls.each_with_index do |url, i|
 puts "#{(i+1)} #{url}"
end

```

## 20.4.5 优化

### Optimizing

与大多数高级语言（即使不是所有语言）一样，在代码中可以非常容易地掺杂进那些慢得让人无法忍受的代码，但是只要稍加考虑，就会很容易解决它们。

使用 `WIN32OLE`，需要当心那些不必要的动态查找（`lookup`）。只要有可能，最好把 `WIN32OLE` 赋值给变量，然后使用它来引用其元素，而不是创建一长链的“.”表达式。

比如，不要写成：

```

workbook.Worksheets(1).Range("A1").value = 1
workbook.Worksheets(1).Range("A2").value = 2
workbook.Worksheets(1).Range("A3").value = 4
workbook.Worksheets(1).Range("A4").value = 8

```

可以通过把表达式的前面部分保存到临时变量中，以去掉这些共同的子表达式，再从这个变量那儿进行调用。

```

worksheet = workbook.Worksheets(1)
worksheet.Range("A1").value = 1
worksheet.Range("A2").value = 2
worksheet.Range("A3").value = 4
worksheet.Range("A4").value = 8

```

也可以为特定的 Windows 类型库创建 Ruby 存根（`stub`）。这些存根把 OLE 对象包装到 Ruby 类中，其中 OLE 对象中的每个入口函数都在该类中有一种对应的方法。在内部，存根方法使用入口函数的编号而不是它的名称，这加快了访问速度。

给出类型库的名称，使用 `ext\win32ole\samples` 目录中的 `olegen.rb` 脚本可以生成这个封装类。

```
C:\> ruby olegen.rb 'NetMeeting 1.1 Type Library' >netmeeting.rb
```

类型库的外部方法和事件会作为 Ruby 方法写入到指定的文件中。你可以把它包含在你的程序中，并直接调用这些方法。我们试着比较一下用时。

```
require 'netmeeting'
require 'benchmark'
include Benchmark

bmbr(10) do |test|
 test.report("Dynamic") do
 nm = WIN32OLE.new('NetMeeting.App.1')
 10000.times { nm.Version }
 end

 test.report("Via proxy") do
 nm = NetMeeting_App_1.new
 10000.times { nm.Version }
 end
end
```

输出结果：

```
Rehearsal -----
Dynamic 0.600000 0.200000 0.800000 (1.623000)
Via proxy 0.361000 0.140000 0.501000 (0.961000)
-----total: 1.301000sec

 user system total real
Dynamic 0.471000 0.110000 0.581000 (1.522000)
Via proxy 0.470000 0.130000 0.600000 (0.952000)
```

这个代理版本比执行动态查找的代码快出 40%以上。

## 20.4.6 更多的帮助

### More Help

如果需要用 Ruby 访问 Windows NT、2000 或 XP 的话，你可以看一下 Daniel Berger 的 Win32Utils 项目（<http://rubyforge.org/projects/win32utils/>）。在那里会找到访问 Windows 剪贴板、事件日志和调度器等的模块。

另外，DL 库（将在第 669 页简单介绍）允许 Ruby 程序调用动态装入的共享库中的方法。在 Windows 上，这意味着 Ruby 代码可以装入和调用 Windows DLL 中的入口函数。比如，下面的代码取自标准 Ruby 发行版中的 DL 源码，它在 Windows 机器上弹出消息框，同时判断用户点击了哪个按钮。

```
require 'dl'
User32 = DL.dlopen("user32")
MB_OKCANCEL = 1
```

```
message_box = User32['MessageBoxA', 'ILSSI']
r, rs = message_box.call(0, 'OK?', 'Please Confirm', MB_OKCANCEL)
case r
when 1
 print("OK!\n")
when 2
 print("Cancel!\n")
end
```

打开 User32 DLL 这段代码。创建 message\_box 这个 Ruby 对象，该对象包装 (wrap) 了 MessageBoxA 的入口函数。第二个参数 “ILSSI” 声明这个方法会返回一个整数，同时接受一个 Long、两个字符串和一个整数作为它的参数。

这个包装 (wrapper) 对象被用来调用在 DLL 中的消息框入口函数。返回值是用户交互的结果（在本例中，它是用户按下按钮的标识符），以及传入参数的一个数组（我们忽略了它）。

# 扩展 Ruby

---

## Extending Ruby

用 Ruby 编写代码来扩展 Ruby 新的特性很简单。但是你偶尔也需要和底层系统打交道。如果你用 C 编写的底层代码扩展 Ruby，更有无限的可能性。话虽如此，但本章的内容比较高阶，你可以在第一次通读本书时跳过它。

用 C 扩展 Ruby 很简单。例如，假定我们要为 Sunset Diner 和 Grill 定制一个支持 Internet 的点唱机。它可以从硬盘播放 MP3 音频文件，或者从 CD 唱机播放音乐 CD。我们希望能从 Ruby 程序控制点唱机硬件，硬件供应商为我们提供了 C 的头文件和要使用的二进制库；我们的工作是构造一个 Ruby 对象来调用适当的 C 函数。

本章的大部分信息是从 Ruby 发布所包含的 `README.EXT` 文件中选取的。如果你计划编写一个 Ruby 扩展，那么你可能需要参考该文件来获取更多的细节和最近的更新。

### 21.1 你的第一个扩展

#### Your First Extension

为了介绍编写扩展，先让我们动手编写一个吧。这个扩展只是为了测试一个过程——它并没有做什么无法单纯用 Ruby 来完成的事情。我们所演示的部分内容没有过多的阐述——所有这些烦乱的细节稍后给出。

我们要编写的扩展与下面的 Ruby 类有相同的功能。

```
class MyTest
 def initialize
 @arr = Array.new
 end
 def add(obj)
 @arr.push(obj)
 end
end
```

换言之，我们要用 C 编写一个扩展，让它和上面的 Ruby 类做到插入兼容（plug-compatible）。等价的 C 代码看起来和下面的类似。

```
#include "ruby.h"
static int id_push;
static VALUE t_init(VALUE self)
{
 VALUE arr;
 arr = rb_ary_new();
 rb_iv_set(self, "@arr", arr);
 return self;
}
static VALUE t_add(VALUE self, VALUE obj)
{
 VALUE arr;
 arr = rb_iv_get(self, "@arr");
 rb_funcall(arr, id_push, 1, obj);
 return arr;
}
VALUE cTest;
void Init_my_test() {
 cTest = rb_define_class("MyTest", rb_cObject);
 rb_define_method(cTest, "initialize", t_init, 0);
 rb_define_method(cTest, "add", t_add, 1);
 id_push = rb_intern("push");
}
```

我们仔细查看这个示例，发现它阐明了本章许多重要的概念。首先，我们需要包含头文件 `ruby.h` 获得所需的 Ruby 定义。

现在查看最后一个方法 `Init_my_test`。每个扩展都定义一个名为 `Init_name` 的 C 全局函数，这个函数在解释器第一次加载（或者静态链接）名为 `name` 的扩展时被调用。它被用来初始化扩展，并加入到 Ruby 的环境中。（至于 Ruby 如何知道扩展的名称是 `name`，我们稍后再涉及。）本例中，我们定义了一个新的类 `MyTest`，是 `Object`（由外部符号 `rb_cObject` 表示；其他请参见 `ruby.h`）的子类。

接下来我们将 `add` 和 `initialize` 设置为 `MyTest` 的两个实例方法。调用 `rb_define_method` 会在 Ruby 方法名和实现它的 C 函数间建立一个绑定。如果 Ruby 代码调用对象实例的 `add` 方法，解释器会调用 C 函数 `t_add`，并传入一个参数。

类似地，当调用该类的 `new` 方法时，Ruby 会构造一个基本的对象，然后调用 `initialize`，这里我们定义了它要调用的 C 函数 `t_init`，而没有（Ruby 类型的）参数。

现在让我们返回并看看 `t_init` 的定义。虽然我们说它没有参数，但它其实是一个参数的！除了所有的 Ruby 参数之外，每个方法会被传入一个初始的 `VALUE` 参数，其中包括该方法的接收者（`receiver`，等同于 Ruby 代码中的 `self`）。

在 `t_init` 中我们要做的第一件事是创建一个 Ruby 数组，并设置让实例变量 `@arr` 来指向它。正如你所预期的，如果你编写的 Ruby 代码引用一个不存在的实例变量，则会创建它。然后我们得到一个指针。

**警告：**每个从 Ruby 调用的 C 函数必须返回一个 `VALUE`，即使它只是 `Qnil`。否则，结果很可能是 core dump（或 GPF，General Protection Fault，一般性保护错误）。

最后，`t_add` 方法从当前对象得到实例变量 `@arr`，并调用 `Array#push` 将传入的参数值存入数组。当以这种方式访问实例变量时，前缀 `e` 是必须的——否则变量虽可创建但无法从 Ruby 内引用。

尽管 C 带来了额外和笨重的语法，你还是在使用 Ruby 来编写程序——可以调用任何你知道和喜爱的方法来操作对象，得到的好处是在需要时能够手工编写紧凑且快速的代码。

### 21.1.1 构建我们的扩展 Building Our Extension

稍后我们会对构建扩展有更多阐述。不过现在，我们所要做的就是遵循下面的步骤。

1. 在 C 源文件 `my_test.c` 的同一目录下，创建一个名为 `extconf.rb` 的文件。  
`extconf.rb` 应该包括以下两行。

```
require 'mkmf'
create_makefile("my_test")
```

2. 运行 `extconf.rb`。这会生成一个 `Makefile`。

```
* ruby extconf.rb
creating Makefile
```

3. 使用 `make` 来构建扩展。这是在 Mac OS X 系统中所产生的结果。

```
* make
gcc -fno-common -g -O2 -pipe -fno-common -I.
-I/usr/lib/ruby/1.9/powerpc-darwin7.4.0
-I/usr/lib/ruby/1.9/powerpc-darwin7.4.0 -I. -c my_test.c
cc -dynamic -bundle -undefined suppress -flat_namespace
-L'/usr/lib' -o my_test.bundle my_test.o -ldl -lobjc
```

构建这个扩展的所有结果，会被组装成一个共享库（一个 `.so`、`.dll` 或 `.bundle` [在 OS X 上]）。

## 21.1.2 运行我们的扩展

### Running Our Extension

我们可以简单地通过在运行时动态 `require` 来使用它（适用于大部分系统）。我们可以将它放到一个测试中，来验证它如我们预期地那样工作。

```
require 'my_test'
require 'test/unit'

class TestTest < Test::Unit::TestCase
 def test_test
 t = MyTest.new
 assert_equal(Object, MyTest.superclass)
 assert_equal(MyTest, t.class)
 t.add(1)
 t.add(2)
 assert_equal([1,2], t.instance_eval("@arr"))
 end
end
```

输出结果：

```
Finished in 0.012271 seconds.
1 tests, 3 assertions, 0 failures, 0 errors
```

在看到扩展可以正常工作后，然后我们可以通过运行 `make install` 将它在全局安装。

## 21.2 C 中的 Ruby 对象

### Ruby Objects in C

当我们编写第一个扩展时，我们作了假，因为它实际没有做任何和 Ruby 对象有关的事情——例如它并没有对 Ruby 的数字类型进行计算。在操作 Ruby 对象之前，我们需要找出如何从 C 表示和访问 Ruby 的数据对象。

在 Ruby 中一切皆对象，所有变量都是对象的引用。当我们从 C 代码中考察 Ruby 对象时，情况还是一样的。大部分 Ruby 对象，被表示为一个 C 指针，它指向了包括对象数据和其他实现细节的一块内存区域。在 C 代码中，所有这些引用都是 VALUE 类型的变量，因此当你传递 Ruby 对象时，所传递的是 VALUE。

但有一个例外。出于性能的原因，Ruby 将 `Fixnum`、`Symbol`、`true`、`false` 和 `nil` 实现为所谓的立即值 (*immediate value*)。它们的值也保存在 VALUE 类型的变量中，但是它们不是指针。相反地，它们的值直接保存在变量中。

因此，某些时候 `VALUE` 是指针，而某些时候它们是直接值。解释器如何实现这一魔法呢？它依赖于这样的一个事实，所有指向某个内存区域的指针，都是 4 或 8 字节边界对齐的。这意味着我们可以确保指针的低两位总是 0。当它想要储存一个立即值时，它会安排至少将其中一位设置为 1，其余的解释器代码可以将指针和直接值区别开来。虽然这听起来非常诡谲，但实际上很容易在实践中使用，主要是因为解释器有很多宏和方法，来简化这个类型系统的处理。

这是 Ruby 如何在 C 中实现面向对象的代码：Ruby 对象是内存中分配的一个结构，包括了实例变量的表（table）和有关这个类的信息。类本身是另一个对象（也是内存中分配的结构），包括了类的实例方法表。Ruby 是在此基础上建立的。

### 21.2.1 操作直接对象

#### Working With Immediate Objects

如我们上面提到的，直接值并非指针：`Fixnum`、`Symbol`、`true`、`false` 和 `nil` 是直接以 `VALUE` 存储的。

`Fixnum` 的值是一个 31 位的数字<sup>1</sup>，它将原来的数字左移一位，并设置 LSB (least significant bit, 最低有效位, bit 0) 为 1。当 `VALUE` 被作为一个指针指向某个具体的 Ruby 结构时，总能够确保 LSB 的值为 0；其他直接值的 LSB 也为 0。因此，一个简单的位测试可以告诉你它是否是一个 `Fixnum`。`FIXNUM_P` 的宏包装了这个测试。类似的测试可以让你检查其他的直接值。

<code>FIXNUM_P(value)</code>	→ 如果 <code>value</code> 是 <code>Fixnum</code> , 则非 0
<code>SYMBOL_P(value)</code>	→ 如果 <code>value</code> 是 <code>Symbol</code> , 则非 0
<code>NIL_P(value)</code>	→ 如果 <code>value</code> 是 <code>nil</code> , 则非 0
<code>RTEST(value)</code>	→ 如果 <code>value</code> 是 <code>nil</code> 或 <code>false</code> , 则非 0

下一页的表 21.1 列出了若干有用的数字以及其他标准数据类型的转换宏。

其他的直接值 (`true`、`false` 和 `nil`) 在 C 中被分别表示为 `Qtrue`、`Qfalse` 和 `Qnil` 的常量。你可以直接测试 `VALUE` 变量，或者使用转换宏（由它进行适当的转型）。

### 21.2.2 操作字符串

#### Working with Strings

在 C 中，我们使用 `null` 结尾的字符串。不过，Ruby 字符串更一般化，可能包括若干内嵌的 `null`。因此，操作 Ruby 字符串最安全的方式是，按照解释器所做的那样，同时使

<sup>1</sup> 或者在更广泛的 CPU 体系结构中可能是 63 位。

表 21.1 C/Ruby 数据类型转换函数和宏

C 数据类型转换为 Ruby 对象		
INT2NUM( <i>int</i> )	→ <i>Fixnum</i> 或 <i>Bignum</i>	
INT2FIX( <i>int</i> )	→ <i>Fixnum</i> (速度较快)	
LONG2NUM( <i>long</i> )	→ <i>Fixnum</i> 或 <i>Bignum</i>	
LONG2FIX( <i>int</i> )	→ <i>Fixnum</i> (速度较快)	
LL2NUM( <i>long long</i> )	→ <i>Fixnum</i> 或 <i>Bignum</i> (如果本地系统支持 <i>long long</i> 类型)	
ULL2NUM( <i>long long</i> )	→ <i>Fixnum</i> 或 <i>Bignum</i> (如果本地系统支持 <i>long long</i> 类型)	
CHR2FIX( <i>char</i> )	→ <i>Fixnum</i>	
rb_str_new2( <i>char</i> *)	→ <i>String</i>	
rb_float_new( <i>double</i> )	→ <i>Float</i>	
Ruby 对象转换为 C 数据类型		
<i>int</i>	NUM2INT( <i>Numeric</i> )	(包括类型检查)
<i>int</i>	FIX2INT( <i>Fixnum</i> )	(速度较快)
<i>unsigned int</i>	NUM2UINT( <i>Numeric</i> )	(包括类型检查)
<i>unsigned int</i>	FIX2UINT( <i>Fixnum</i> )	(包括类型检查)
<i>long</i>	NUM2LONG( <i>Numeric</i> )	(包括类型检查)
<i>long</i>	FIX2LONG( <i>Fixnum</i> )	(速度较快)
<i>unsigned long</i>	NUM2ULONG( <i>Numeric</i> )	(包括类型检查)
<i>char</i>	NUM2CHR( <i>Numeric</i> 或 <i>String</i> )	(包括类型检查)
<i>double</i>	NUM2DBL( <i>Numeric</i> )	
	关于字符串的部分请参见 正文……	

用指针和长度。实际上，Ruby *String* 对象是 *RString* 结构的一个引用，而 *RString* 结构包括指针和长度成员。你可以通过 *RSTRING* 宏来访问这个结构。

```
VALUE str;
RSTRING(str)->len → Ruby 字符串的长度
RSTRING(str)->ptr → 字符串的存储指针
```

- 然而，生活远比这复杂。当你需要一个字符串值的时候，与其直接使用 *VALUE* 对象，不如调用 *StringValue*，并以原来的值作为参数。它会返回一个对象，你可以对其使用 *RSTRING* 宏，或者如果它无法从原来的值得到一个字符串时，则抛出异常。这是 Ruby 1.8 的 duck typing<sup>2</sup> 提案的一部分，在第 294 页和第 365 页有更详细的描述。*StringValue* 方法检查其操作数是否是一个 *String*，如果不是，则它尝试调用对象的 *to\_str*，如果调用失败，则抛出一个 *TypeError* 异常。

<sup>2</sup> 译注：动态类型的术语，有两个含义，一是说 Ruby 避开了类型系统，二是说来自 Dave Thomas 的隐喻：如果某物走起来像鸭子，叫起来也像鸭子，那么就可认为它是一只鸭子。

因此，如果你想编写迭代 `String` 对象中所有字符的代码，可以这样编写：

```
static VALUE iterate_over(VALUE original_str) {
 int i;
 char *p;
 VALUE str = StringValue(original_str);
 p = RSTRING(str)->ptr; // may be null
 for (i = 0; i < RSTRING(str)->len; i++, p++) {
 // process *p
 }
 return str;
}
```

如果你想跳过长度信息，而只访问底层的字符串指针，你可以使用便捷方法 `StringValuePtr`，它得到字符串的引用，然后返回指向其内容的 C 指针。

如果你想使用字符串来访问或控制某些外部资源，那么你可能希望钩入（hook）Ruby 的 `tainting` 机制中。在本例中，你可以使用 `SafeStringValue` 方法，它像 `StringValue` 一样工作，但如果它的参数是 `tainted`（不可信的）同时安全级别大于 0 的，则抛出一个异常。

### 21.2.3 操作其他对象

#### Working with Other Objects

当 `VALUE` 不是直接对象时，它们指向定义了 Ruby 对象结构的指针——你不能让 `VALUE` 指向任意内存区域。基本内建类的结构在 `ruby.h` 中定义，命名为 `RClassname`：例如 `RArray`、`RBignum`、`RClass`、`RData`、`RFile`、`RFloat`、`RHash`、`RObject`、`RRegexp`、`RString` 和 `RStruct`。

你有多种方式可以检查某个特定的 `VALUE` 是何种类型的结构。宏 `TYPE (obj)` 会返回一个常量，表示给定对象的 C 类型：`T_OBJECT`、`T_STRING` 等。内建类的常量在 `ruby.h` 中定义。注意，这里所说的 `type` 是实现中的细节——和对象的类不同。

如果你想确保 `VALUE` 指针指向一个特定的结构，你可以使用宏 `Check_Type`，如果 `value` 不是预期的 `type` (`T_STRING`、`T_FLOAT` 这样的常量)，则它会抛出一个 `TypeError` 异常。

```
Check_Type(VALUE value, int type)
```

再次，注意我们所说的“`type`”是表示某个特定内建类型的 C 结构。对象的类是完全不同的另一只怪兽。内建类的 `class` 对象保存在名为 `rb_cClassname` 的 C 全局变量中（例如，`rb_cObject`），模块的命名为 `rb_mModulename`。

## Ruby 1.8 之前的字符串访问

### Pre1.8 String Access

在 Ruby 1.8 之前，如果认为 VALUE 包含一个字符串，那么你直接访问 RSTRING 成员即可。然而，在 1.8 中，随着 duck typing 以及各种优化的逐步引入，这种方法可能无法如往常那样工作。特别地，STRING 对象的 ptr 成员可能用 null 来表示长度为 0 的字符串。如果你使用 1.8 中的 `StringValue` 方法，则它会处理这种情况，将 null 指针重新指向一个唯一的、共享的空字符串。

因此，如何编写一个在 Ruby 1.6 和 1.8 中都可以工作的扩展呢？请谨慎地使用宏。也许可以使用下面的方式。

```
#if !defined(StringValue)
define StringValue(x) (x)
#endif
#ifndef !defined(StringValuePtr)
define StringValuePtr(x) ((STR2CSTR(x)))
#endif
```

这段代码为较老的 1.6 版本，定义了 1.8 中的 `StringValue` 和 `StringValuePtr` 宏。如果你用这些宏来编写代码，那么较老的和新的解释器都可以编译并运行。

如果希望你的代码即使在 1.6 中运行，也具有 1.8 中 duck-typing 的行为，那么你可能希望稍有不同地定义 `StringValue`。这和之前的实现的区别在第 294 页中描述。

```
#if !defined(StringValue)
define StringValue(x) do {
 if (TYPE(x) != T_STRING) x = rb_str_to_str(x);
} while (0)
#endif
```

直接更改这些 C 结构中的数据是不可取的，不过——你可以查看，但不要触动它们。相反，你通常使用提供的 C 函数来操作 Ruby 数据（我们稍后将有更多讨论）。

然而，你可能有兴趣深入到这些结构中来得到数据。要解引用（dereference）这些 C 结构的成员，你必须将一般化的 VALUE 转型为适当的结构类型。`ruby.h` 包含了许多宏为你执行适当的转型，让你可以很容易地对结构成员解引用。这些宏的名称为 `RCLASSNAME`，例如 `RSTRING` 或 `RARRAY`。我们已经在处理字符串时看到了 `RSTRING` 的使用。你可以对数组使用相同的方式。

```

VALUE arr;
RARRAY(arr)->len → length of the Ruby array
RARRAY(arr)->capa → capacity of the Ruby array
RARRAY(arr)->ptr → pointer to array storage

```

对散列表（RHASH）、文件（RFILE）等也有类似的访问方法。话虽如此，但你也要小心，在扩展代码中不要过于依赖对类型的检查。我们将在第 294 页更多地讨论扩展以及 Ruby 的类型系统。

## 21.2.4 全局变量

### Global Variables

多数时候，由你的扩展实现类，而 Ruby 代码使用这些类。你在 Ruby 和 C 代码中共享的数据，会整洁地包装（wrap）到类的对象中。它应当如此。

不过，有时候你希望实现全局变量，在 C 扩展和 Ruby 代码中都可以访问。

最简单的方法是，让变量作为一个 VALUE（也就是说，一个 Ruby 对象）。然后你可以将 C 变量的地址绑定到一个 Ruby 变量的名字。在这种情况下，\$前缀是可选的，但是它可以帮助你澄清这是一个全局变量。并且请记住：让栈中的变量作为 Ruby 的全局变量，不会（长期）有效。

```

static VALUE hardware_list;
static VALUE Init_SysInfo() {
 rb_define_class(...);
 hardware_list = rb_ary_new();
 rb_define_variable("$hardware", &hardware_list);
 ...
 rb_ary_push(hardware_list, rb_str_new2("DVD"));
 rb_ary_push(hardware_list, rb_str_new2("CDPlayer1"));
 rb_ary_push(hardware_list, rb_str_new2("CDPlayer2"));
}

```

Ruby 端可以用 \$hardware 来访问 C 变量 hardware\_list。

```
$hardware → ["DVD", "CDPlayer1", "CDPlayer2"]
```

不过，有时生活会更复杂些。也许你要定义的全局变量，它的值在访问时必须通过计算才能得到。你可以通过定义 *hooked* 和虚拟（virtual）变量来完成。*hooked* 变量是一个真实的变量，当访问到对应的 Ruby 变量时，由一个具名（named）的函数初始化。虚拟变量也有些类似，但并没有真正的存储：它们的值来自于对 hook 函数的求值。更多细节请参见第 308 页开始的 API 章节。

如果你从 C 创建一个 Ruby 对象，把它保存在一个 C 全局变量中，而不将它暴露给 Ruby，你至少要将它告知给垃圾收集器，以免因疏忽而忘记回收。

```
static VALUE obj;
// ...
obj = rb_ary_new();
rb_global_variable(obj);
```

## 21.3 Jukebox 扩展

### The Jukebox Extension

我们已经覆盖了大部分的基本内容，现在回到我们的点唱机示例——连接 Ruby 和 C 代码，并且在两个世界中共享数据和行为。

#### 21.3.1 包装 C 结构

##### Wrapping C Structures

我们已经得到了供应商控制音乐 CD 点唱机单元的库，并已经准备好将它连接到 Ruby 中。供应商的头文件类似如下所示。

```
typedef struct _cdjb {
 int statusf;
 int request;
 void *data;
 char pending;
 int unit_id;
 void *stats;
} CDJukebox;

// Allocate a new CDJukebox structure
CDJukebox *new_jukebox(void);

// Assign the Jukebox to a player
void assign_jukebox(CDJukebox *jb, int unit_id);

// Deallocate when done (and take offline)
void free_jukebox(CDJukebox *jb);

// Seek to a disc, track and notify progress
void jukebox_seek(CDJukebox *jb,
 int disc,
 int track,
 void (*done)(CDJukebox *jb, int percent));

// ... others...

// Report a statistic
double get_avg_seek_time(CDJukebox *jb);
```

这是供应商提供的全部操作：虽然他们可能不承认，代码是用面向对象的方式编写的。我们并不知道 CDJukeBox 接口中所有成员的含义，但这不碍事——我们可以将它看作是不透明的一大堆数据。供应商的代码知道如何操作它；我们只需要将这些数据携带进去。

任何时候，你想要如 Ruby 对象那样来处理一个 C 结构，你需要将它包装（wrap）成一个特殊的 Ruby 内部类，称为 DATA（类型为 T\_DATA）。有两个宏完成这一包装，且其中一个宏会再次返回你的结构。

### 21.3.2 API: C 数据类型的包装

#### API: C Data Type Wrapping

```
VALUE Data_Wrap_Struct(VALUE class, void (*mark)(),
 void (*free)(), void *ptr)
```

包装给定的 C 数据类型 *ptr*，注册两个垃圾收集例程（参见下面的描述），并返回一个指向实际 Ruby 对象的 VALUE 指针。返回对象的 C 类型是 T\_DATA，且它的 Ruby 类是 *class*。

```
VALUE Data_Make_Struct(VALUE class, c-type, void (*mark)(),
 void (*free)(), c-type *)
```

分配指定类型的结构体并清零，然后继续 Data\_Wrap\_Struct 的工作。*c-type* 是你要包装的 C 数据类型的名字，而不是类型的变量。

```
Data_Get_Struct(VALUE obj, c-type, c-type *)
```

返回原本的指针。这个宏是对宏 DATA\_PTR (*obj*) 的类型安全的包装，它会对指针进行评估。

Data\_Wrap\_Struct 创建的对象是普通的 Ruby 对象，唯一不同的是它有额外的、无法从 Ruby 中访问的 C 数据类型。你可以从下一页的图 21.1 中看到，C 数据类型和对象所包含的实例变量是分开的。但由于它是独立的，当垃圾收集器回收这个对象时，该怎样清除它呢？如果你必须释放某些资源（关闭文件、清除锁或其他 IPC 机制等等），又该如何呢？

Ruby 使用标记和清扫（mark and sweep）的垃圾回收方式。在标记阶段，Ruby 查找指向内存区域的指针。它标记这些区域为“使用中”（因为有东西指向它们）。如果这些区域本身包括更多的指针，这些指针所指向的内存也会被标记，以此类推。在标记的最后阶段，所有被引用的内存已经被标记了，而所有孤立的区域则不会有标记。当清扫阶段开始时，释放那些没有被标记的内存。

为了参与到 Ruby 标记后清扫的垃圾回收过程，你必须定义一个例程来释放你的结构，或者还需要一个例程来完成从你的结构到其他结构的引用标记。两个例程都接收 void 指针，它指向了你的结构。*mark* 例程在垃圾收集器处于标记阶段时被调用。如果你的结构引用了其他的 Ruby 对象，那么你的标记函数需要用 rb\_gc\_mark (*value*) 来标识出这些对象。如果结构没有引用其他的 Ruby 对象，你可以简单地以 0 作为函数的指针。

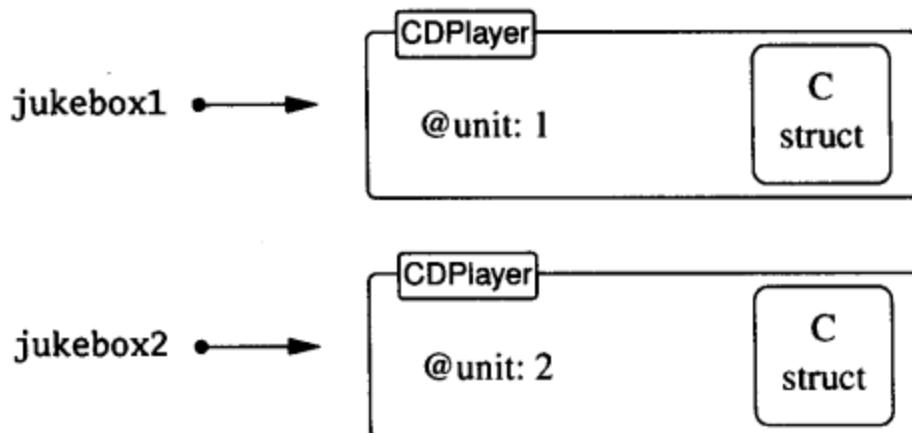


图 21.1 将 C 数据类型包装为对象

当对象需要清除时，垃圾收集器将调用 `free` 例程来释放它。如果你自己分配了某些内存（例如，使用 `Data_Make_Struct`），你需要传入一个释放函数——即便它就是标准 C 库中的 `free` 例程。对你分配的复杂结构，你的释放函数可能需要遍历这个结构来释放所有分配的内存。

让我们看看 CD 播放器的接口。供应商的库将信息以及各种函数保存在一个 `CDJukebox` 结构中。这个结构表示点唱机的状态，因此是将之包装成 Ruby 类的一个好的候选。我们通过调用库中的 `CDPlayerNew` 方法来创建该结构的一个新实例，然后将这个创建的结构包装到一个新的 `CDPlayer` Ruby 对象中。下面的代码片段完成了这一操作（稍后我们将讨论神秘的 `klass` 参数）。

```
CDJukebox *jukebox;
VALUE obj;

// Vendor library creates the Jukebox
jukebox = new_jukebox();

// then we wrap it inside a Ruby CDPlayer object
obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
```

在代码执行后，`obj` 应该保存了一个新分配的 `CDPlayer` Ruby 对象（包装了新建的 `CDJukebox` C 结构）的引用。当然，要编译这段代码，我们还需要多作些工作。我们必须定义 `CDPlayer` 的类，并将它的引用保存在 `cCDPlayer` 变量中。我们还必须定义用来释放对象的函数 `cdplayer_free`。这很简单：只需调用供应商库中的清除方法即可。

```
static void cd_free(void *p) {
 free_jukebox(p);
}
```

不过，代码片段还不足成事。我们需要以某种方式将所有这些东西打包集成到解释器中。为此，我们还需要查看解释器使用的某些约定。

### 21.3.3 对象创建 Object Creation

1.8 Ruby 1.8 完善了对象的创建和初始化。虽然旧有的方式也可以工作，但是使用分配函数的新方式，要整洁得多（而且将来也不大会被废弃）。

基本的想法很简单。假说你要在 Ruby 程序中创建一个 `CDPlayer` 类的对象。

```
cd = CDPlayer.new
```

在幕后，解释器调用 `CDPlayer` 类方法的 `new`。因为 `CDPlayer` 没有定义 `new`，所以 Ruby 查看它的父类，`Class` 类。

实现 `Class` 类中的 `new` 相当简单：它为新的对象分配内存，然后调用对象的 `initialize` 方法来初始化内存。

因此，如果我们的 `CDPlayer` 扩展要成为一个 Ruby 的好公民，它应该能在这种框架下工作。这意味着，我们需要实现一个分配函数以及一个初始化方法。

### 21.3.4 分配函数 Allocation Functions

分配函数负责创建对象使用的内存。如果你所实现的对象，并不需要除 Ruby 实例变量之外的数据，那么你无须编写一个分配函数——Ruby 默认的分配器就可以满足了。但是如果你的类包装了一个 C 结构，你需要在分配函数中为这个结构分配空间。分配函数将要分配对象的类（`klass`）作为参数。在我们的示例中，它十有八九是 `cCDPlayer`，但是我们直接使用给出的参数，因为这意味着作为 `klass` 的子类我们也可以正确地工作。

```
static VALUE cd_alloc(VALUE klass) {
 CDJukebox *jukebox;
 VALUE obj;

 // Vendor library creates the Jukebox
 jukebox = new_jukebox();

 // then we wrap it inside a Ruby CDPlayer object
 obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);
 return obj;
}
```

你还需要在类的初始化代码中注册你的分配函数。

```
void Init_CDPlayer() {
 cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
 rb_define_alloc_func(cCDPlayer, cd_alloc);
 // ...
}
```

大部分对象可能也需要定义一个初始化函数。分配函数创建一个空的、未初始化的对象，而我们需要填入具体的值。以 CD 播放器为例，调用构造函数可指定与该对象所关联的播放器单元个数。

```
static VALUE cd_initialize(VALUE self, VALUE unit) {
 int unit_id;
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);
 unit_id = NUM2INT(unit);
 assign_jukebox(jb, unit_id);
 return self;
}
```

这种多步对象创建协议的原因是，让解释器能够处理必须以“后门方式”创建对象的情况。一个例子是，当对象要从列集的（marshaled）形式中反序列化时。此时，解释器需要创建一个空的对象（通过调用分配器），但是它不会调用初始化函数（因为它不知道要使用的参数）。另一个常见的情况是，当对象被复制或克隆时。

这里潜伏了一个更深入的问题。因为用户可以选择绕过构造函数，你需要确保分配代码使返回的对象处于有效状态。它可能并没有包括曾经由`#initialize`设置的全部信息，但是至少应该是可用的。

### 21.3.5 克隆对象

#### Cloning Objects

所有 Ruby 对象都可以使用 `dup` 或 `clone` 方法来拷贝。这两个方法是类似的：两者都会通过调用分配函数产生调用类的一个新实例。然后，它们从原来的对象中拷贝实例变量。`clone` 会拷贝得更深入些，它会拷贝原来对象的单体类（singleton class，如果它有的话）和标志（例如指示对象被冻结的标志）。你可以认为 `dup` 是内容的拷贝，而 `clone` 是整个对象的拷贝。

不过，Ruby 解释器并不知道如何处理你编写的 C 扩展中对象的内部状态。例如，如果你的对象包装了一个 C 结构，其中包括一个打开的文件描述符，这取决于实现的语言，这个描述符是否应该简单地拷贝到新对象中，或者是否应该打开一个新的文件描述符。

## 1.8 版本之前的对象分配

### Pre-1.8 Object Allocation

1.8

在 Ruby 1.8 之前，如果你想要在对象中分配额外的空间，要么你将代码放到 `initialize` 方法中，要么你必须为你的类定义一个 `new` 方法。Guy Decoux 建议使用下面的混合方法，来做到 1.6 和 1.8 扩展间最大的兼容。

```
static VALUE cd_alloc(VALUE klass) {
 // same as before
}

static VALUE cd_new(int argc, VALUE *argv, VALUE klass) {
 VALUE obj = rb_funcall2(klass,
 rb_intern("allocate"), 0, 0);
 rb_obj_call_init(obj, argc, argv);
 return obj;
}

void init_CDPlayer() {
 // ...

#if HAVE_RB_DEFINE_ALLOC_FUNC
 // 1.8 allocation
 rb_define_alloc_func(cCDPlayer, cd_alloc);
#else
 // define manual allocation function for 1.6
 rb_define_singleton_method(cCDPlayer, "allocate",
 cd_alloc, 0);
#endif
 rb_define_singleton_method(cCDPlayer, "new", cd_new, -1);
 // ...
}
```

如果你要编写的代码必须运行在 Ruby 最新和较老的版本上，那么你需要使用与此类似的方法。不过，你可能还需要处理克隆和复制，并考虑当你的对象被列集时会如何。

为了处理它，解释器将拷贝对象内部状态的职责委托给你自己的代码。在拷贝对象的实例变量之后，解释器调用新对象的 `initialize_copy` 方法，传入原对象的引用。这取决于你在该方法中实现有意义的语义。

对我们的 `CDPlayer` 类来说，对克隆问题采取十分简单的方法：我们简单地从原对象中拷贝 `CDJukebox` 结构。

这是本例的一段短小而奇怪的代码。要测试我们确实可以克隆原对象，代码检查原对象：

1. 具有 T\_DATA 的 TYPE（意味着它是一个非核心的对象），并且
2. 具有和我们相同的释放函数。

这是一种相对性能较好的方式，来验证原对象和我们自己的对象是兼容的（只要你没有在多个类间共享同样的释放函数）。另一种方式，稍稍慢些，是使用 rb\_obj\_is\_kind\_of 并对类进行直接的测试。

```
static VALUE cd_init_copy(VALUE copy, VALUE orig) {
 CDJukebox *orig_jb;
 CDJukebox *copy_jb;
 if (copy == orig)
 return copy;
 // we can initialize the copy from other CDPlayers
 // or their subclasses only
 if (TYPE(orig) != T_DATA ||
 RDATA(orig)->dfree != (RUBY_DATA_FUNC)cd_free) {
 rb_raise(rb_eTypeError, "wrong argument type");
 }
 // copy all the fields from the original
 // object's CDJukebox structure to the
 // new object
 Data_Get_Struct(orig, CDJukebox, orig_jb);
 Data_Get_Struct(copy, CDJukebox, copy_jb);
 MEMCPY(copy_jb, orig_jb, CDJukebox, 1);
 return copy;
}
```

我们的拷贝方法无须分配一个被包装的结构来接收原对象的 CDJukebox 结构：  
cd\_alloc 方法已经为我们分配了。

注意，使用基于类的类型检查是正确的：我们需要原对象含有一个包装的 CDJukebox 结构，并且只有从类 CDPlayer 派生的对象才有该结构。

### 21.3.6 集成到一起 Putting It All Together

好的，最后，我们已经准备好编写 CDPlayer 类的所有代码了。

```
#include "ruby.h"
#include "cdjukebox.h"
static VALUE cCDPlayer;
```

```

// Helper function to free a vendor CDJukebox
static void cd_free(void *p) {
 free_jukebox(p);
}

// Allocate a new CDPlayer object, wrapping
// the vendor's CDJukebox structure
static VALUE cd_alloc(VALUE klass) {
 CDJukebox *jukebox;
 VALUE obj;

 // Vendor library creates the Jukebox
 jukebox = new_jukebox();

 // then we wrap it inside a Ruby CDPlayer object
 obj = Data_Wrap_Struct(klass, 0, cd_free, jukebox);

 return obj;
}

// Assign the newly created CDPLayer to a
// particular unit
static VALUE cd_initialize(VALUE self, VALUE unit) {
 int unit_id;
 CDJukebox *jb;

 Data_Get_Struct(self, CDJukebox, jb);
 unit_id = NUM2INT(unit);
 assign_jukebox(jb, unit_id);

 return self;
}

// Copy across state (used by clone and dup). For jukeboxes, we
// actually create a new vendor object and set its unit number from
// the old
static VALUE cd_init_copy(VALUE copy, VALUE orig) {
 CDJukebox *orig_jb;
 CDJukebox *copy_jb;

 if (copy == orig)
 return copy;

 // we can initialize the copy from other CDPlayers or their
 // subclasses only
 if (TYPE(orig) != T_DATA ||
 RDATA(orig)->dfree != (RUBY_DATA_FUNC)cd_free) {
 rb_raise(rb_eTypeError, "wrong argument type");
 }

 // copy all the fields from the original object's CDJukebox
 // structure to the new object
 Data_Get_Struct(orig, CDJukebox, orig_jb);
 Data_Get_Struct(copy, CDJukebox, copy_jb);
 MEMCPY(copy_jb, orig_jb, CDJukebox, 1);

 return copy;
}

```

```

// The progress callback yields to the caller the percent complete
static void progress(CDJukebox *rec, int percent) {
 if (rb_block_given_p()) {
 if (percent > 100) percent = 100;
 if (percent < 0) percent = 0;
 rb_yield(INT2FIX(percent));
 }
}

// Seek to a given part of the track, invoking the progress callback
// as we go
static VALUE
cd_seek(VALUE self, VALUE disc, VALUE track) {
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);
 jukebox_seek(jb,
 NUM2INT(disc),
 NUM2INT(track),
 progress);
 return Qnil;
}

// Return the average seek time for this unit
static VALUE
cd_seek_time(VALUE self)
{
 double tm;
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);
 tm = get_avg_seek_time(jb);
 return rb_float_new(tm);
}

// Return this player's unit number
static VALUE
cd_unit(VALUE self) {
 CDJukebox *jb;
 Data_Get_Struct(self, CDJukebox, jb);
 return INT2NUM(jb->unit_id);
}

void Init_CDPlayer() {
 cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
 rb_define_alloc_func(cCDPlayer, cd_alloc);
 rb_define_method(cCDPlayer, "initialize", cd_initialize, 1);
 rb_define_method(cCDPlayer, "initialize_copy", cd_init_copy, 1);
 rb_define_method(cCDPlayer, "seek", cd_seek, 2);
 rb_define_method(cCDPlayer, "seek_time", cd_seek_time, 0);
 rb_define_method(cCDPlayer, "unit", cd_unit, 0);
}

```

现在，我们可以用一种美好的、面向对象的方式，从 Ruby 中控制点唱机。

```
require 'CDPlayer'
p = CDPlayer.new(13)
puts "Unit is #{p.unit}"
p.seek(3, 16) {|x| puts "#{x}% done" }
puts "Avg. time was #{p.seek_time} seconds"
p1 = p.dup
puts "Cloned unit = #{p1.unit}"
```

输出结果：

```
Unit is 13
26% done
79% done
100% done
Avg. time was 1.2 seconds
Cloned unit = 13
```

这个实例演示了迄今我们所讨论的大部分内容，还有一个附加的功能。供应商库提供了一个回调例程——一个函数指针，当硬件播放碟片时会频繁调用。我们在此已经设置停当，运行作为 `seek` 参数传入的一个代码 `block`。在 `progress` 函数中，我们查看在当前上下文中是否有一个迭代器，如果有，以当前完成的百分比为参数运行它。

## 21.4 内存分配

### Memory Allocation

有时你可能需要在扩展中分配内存，但不用于对象的存储——也许你已经得到了 Bloom 滤波器（Bloom filter）一个巨大的位图（bitmap）、一幅图像或者 Ruby 并不直接使用的许多小的数据结构。

为了让垃圾收集器正确地工作，你应该使用下面的内存分配例程。这些例程比标准的 `malloc` 稍稍多做了些工作。例如，如果 `ALLOC_N` 判断它无法分配得到所需数量的内存，将会调用垃圾收集器来尝试收回一些空间。如果无法分配或者所需数量的内存不可获得，它将引发一个 `NoMemError` 异常。

#### 21.4.1 API：内存分配

##### API: Memory Allocation

`type * ALLOC_N (c-type, n)`

分配 `n` 个 `c-type` 的对象，其中 `c-type` 是 C 类型的字面名称，而非该类型的变量。

`type * ALLOC (c-type)`

分配一个 `c-type`，并且将结果转型为该类型的指针。

```
REALLOC_N(var, c-type, n)
```

重新分配 *n* 个 *c-type*, 并将结果复制给指针 *var*, 它指向类型为 *c-type* 的变量。

```
type * ALLOCA_N(c-type, n)
```

在栈上分配 *c-type* 的 *n* 个对象——这部分内存会在调用 **ALLOCA\_N** 的函数返回时自动释放。

## 21.5 Ruby 的类型系统

### Ruby Type System

1.8 在 Ruby 中, 我们很少依赖于一个对象的类型 (或 class), 而更多地关注于它的能力, 这被称为 *duck typing*。我们将在第 365 页的第 23 章中详细描述它。如果你查看解释器的代码, 你会发现许多例证。例如, 下面的代码实现了 `kernel.exec` 方法。

```
VALUE
rb_f_exec(argc, argv)
 int argc;
 VALUE *argv;
{
 VALUE prog = 0;
 VALUE tmp;
 if (argc == 0) {
 rb_raise(rb_eArgError, "wrong number of arguments");
 }
 tmp = rb_check_array_type(argv[0]);
 if (!NIL_P(tmp)) {
 if (RARRAY(tmp)->len != 2) {
 rb_raise(rb_eArgError, "wrong first argument");
 }
 prog = RARRAY(tmp)->ptr[0];
 SafeStringValue(prog);
 argv[0] -= RARRAY(tmp)->ptr[1];
 }
 if (argc == 1 && prog == 0) {
 VALUE cmd = argv[0];
 SafeStringValue(cmd);
 rb_proc_exec(RSTRING(cmd)->ptr);
 }
 else {
 proc_exec_n(argc, argv, prog);
 }
 rb_sys_fail(RSTRING(argv[0])->ptr);
 return Qnil; /* dummy */
}
```

该方法的第一个参数是一个字符串或者一个含有两个字符串的数组。然而，代码并不显式地检查参数的类型。相反，它先对传入的参数调用 `rb_check_array_type`。这个方法会做些什么呢？让我们看一看。

```
VALUE
rb_check_array_type(ary)
 VALUE ary;
{
 return rb_check_convert_type(ary, T_ARRAY, "Array", "to_ary");
}
```

这一小段不足以澄清。让我们继续跟踪到 `rb_check_convert_type`。

```
VALUE
rb_check_convert_type(val, type, tname, method)
 VALUE val;
 int type;
 const char *tname, *method;
{
 VALUE v;
 /* always convert T_DATA */
 if (TYPE(val) == type && type != T_DATA) return val;
 v = convert_type(val, tname, method, Qfalse);
 if (NIL_P(v)) return Qnil;
 if (TYPE(v) != type) {
 rb_raise(rb_eTypeError, "%s#%s should return %s",
 rb_obj_classname(val), method, tname);
 }
 return v;
}
```

现在我们将面临这样的处境。如果对象是正确的类型（本例中为 `T_ARRAY`），那么返回原本的对象。否则，我们并不就此放弃。相反，我们调用原对象，询问它是否可以将自己表现为一个数组（我们调用它的 `to_ary` 方法）。如果可以，我们很高兴并继续。代码所表述的意思是“我并不需要一个 `Array`，我只是需要一个能够表现为数组的对象。”这意味着 `kernel.exec` 可以接收任何实现了 `to_ary` 方法的参数。我们从第 371 页开始会详细地讨论这些转换协议（但是从 Ruby 的角度）。

作为一个扩展的编写者，这对你意味着什么呢？有两个含义。其一，尝试避免检查传递给你的参数类型。相反，看看是否有类似 `rb_check_xxx_type` 的方法帮你将参数转换到你需要的类型。如果没有，查找现有的某个可以帮助你进行转换的函数（例如 `rb_Array`、`rb_Float` 或 `rb_Integer`）。其二，如果你要编写一个扩展，其中实现的某些对象可以有意义地用作一个 Ruby 字符串或数组，考虑实现 `to_str` 或 `to_ary` 方法，允许扩展所实现的对象被用于字符串或数组的上下文中。

## 21.6 创建一个扩展

### Creating an Extension

在编写完扩展的源代码后，现在我们需要编译它来为 Ruby 所用。我们或者将它实现为一个可以在运行时动态加载的共享库，或者静态地连接到 Ruby 的主解释器中。基本的步骤是相同的。

1. 在指定的目录中创建 C 源文件。
2. 在 lib 子目录中创建所有 Ruby 的支持文件（可选）。
3. 创建 extconf.rb。
4. 运行 extconf.rb 来为目录中的 C 文件创建 Makefile。
5. 运行 make。
6. 运行 make install。

#### 21.6.1 通过 extconf.rb 创建 Makefile

##### Creating a Makefile with extconf.rb

下一页的图 21.2 演示了构建扩展的总体流程。整个过程的关键是，你作为开发者所创建的 extconf.rb 程序。在 extconf.rb 中，你编写一个简单的程序来判断在用户系统中有哪些特性可用，以及这些特性位于何处。执行 extconf.rb 会建立一个定制的 Makefile，它是根据你的应用和编译时所用的系统来裁减的。当你对该 Makefile 运行 make 命令时，你的扩展被构建并（可选地）被安装。

最简单的 extconf.rb 可能只有两行，而对许多扩展来说已经足够了。

```
require 'mkmf'
create_makefile("Test")
```

第一行引入了 mkmf 库模块（其描述始于 779 页）。它包括了所有我们要使用的命令。第二行为名为“Test”的扩展创建一个 Makefile。（注意这个“Test”是扩展的名字；而 makefile 的名字则总是 Makefile。）Test 将会从当前目录中的 C 源文件构建。当你的代码被加载时，Ruby 调用它的 Init\_Test 方法。

假说我们在只有一个源文件（main.c）的目录中运行 extconf.rb 程序。其结果是用来构建我们扩展的一个 makefile。在 Linux 机器上，它执行下面的命令。

```
gcc -fPIC -I/usr/local/lib/ruby/1.8/i686-linux -g -O2 \
 -c main.c -o main.o
gcc -shared -o Test.so main.o -lc
```

编译的结果是 Test.so，可以通过 require 在运行时动态地链接到 Ruby 中。

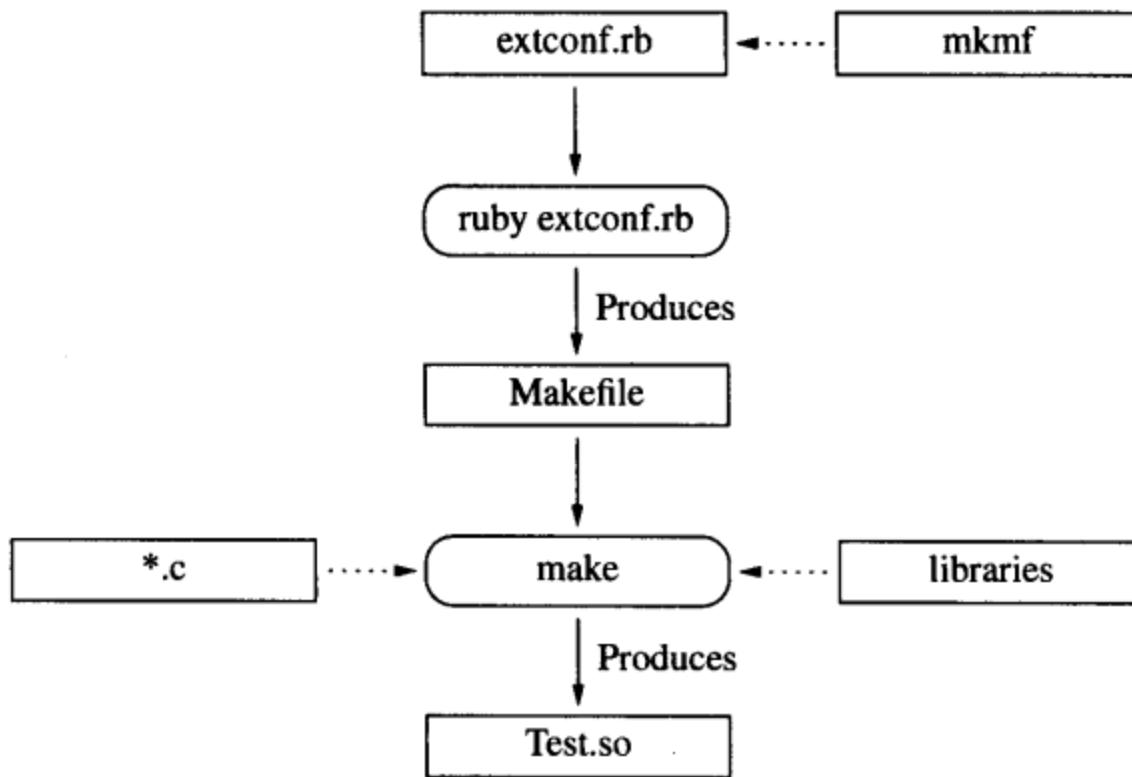


图 21.2 构建扩展的过程

在 Mac OS X 中，命令有所不同，但是结果是相同的：都创建了一个共享库（Mac 上称为 *bundle*）。

```

gcc -fno-common -g -O2 -pipe -fno-common \
-I/usr/lib/ruby/1.8/powerpc-darwin \
-I/usr/lib/ruby/1.8/powerpc-darwin -c main.c
cc -dynamic -bundle -undefined suppress -flat_namespace \
-L'/usr/lib' -o Test.bundle main.o -ldl -lobjc

```

看看 `mkmf` 命令如何自动地定位特定于平台的库，并使用特定于本地编译器的选项。很整洁，不是吗？

虽然这个基本的 `extconf.rb` 程序对许多简单的扩展都适用，但如果您的扩展需要默认编译环境所没有的某些头文件或库，或者您根据某些库或函数的存在与否来进行条件编译，您还需要多做些工作。

一个常见的需求是指定查找头文件和库的非标准目录。这是一个两步的过程。首先，您的 `extconf.rb` 应该包括一个或多个 `dir\_config` 命令。这指定了一个目录集合的标记。然后，当你运行 `extconf.rb` 程序时，告诉 `mkmf` 在当前系统中对应的物理目录在什么地方。

## 划分命名空间

### Dividing Up the Namespace

逐渐地，扩展编写者都变成了好公民。与其将他们的工作直接安装到 Ruby 的某个库目录中，还不如使用子目录将文件组织起来。这对 extconf.rb 很简单。如果 create\_makefile 调用的参数包括斜线，那么 mkmf 会认为在最后一个斜线之前的是目录名，而余下的是扩展的名字。扩展会被安装到指定的目录中（相对 Ruby 的目录树）。在下面的示例中，扩展依然被命名为 Test。

```
require 'mkmf'
create_makefile("wibble/Test")
```

然而，当你在 Ruby 程序中需要这个类时，需要如此编写代码：

```
require 'wibble/Test'
```

如果 extconf.rb 包括 dir\_config (*name*) 的代码行，你可以用命令行选项给出对应目录的位置。

**--with-name-include=directory**

将 *directory/include* 添加到编译器的选项中。

**--with-name-lib=directory**

将 *directory/lib* 添加到链接器的选项中。

如果（而且很常见）你的头文件和库文件的目录分别是某个目录下的 include 和 lib 子目录，你可以使用简捷的方式。

**--with-name-dir=directory**

将 *directory/lib* 和 *directory/include* 分别添加到链接器和编译器的选项中。

和你在运行 extconf.rb 时使用--with 选项一样，也可以使用为你的机器构建 Ruby 时所使用的--with 选项。这意味着，你可以发现并使用 Ruby 本身使用库的位置。

具体起见，假说你开发的 CD 播放器需要使用供应商的 CDJukebox 库和头文件。你的 extconf.rb 可能包括：

```
require 'mkmf'
dir_config('cdjukebox')
.. more stuff
create_makefile("CDPlayer")
```

然后你可能像下面这样运行 extconf.rb:

```
% ruby extconf.rb---with-cdjukebox-dir=/usr/local/cdjb
```

产生的 Makefile 会认为 /usr/local/cdjb/lib 含有库文件，而 /usr/local/cdjb/include 含有头文件。

dir\_config 命令添加用于搜索库和头文件的位置列表。但它并不会将这些库链接到你的应用。为了要这样做，你需要使用 have\_library 或 find\_library 命令。

have\_library 查找某个具名库的指定入口点。如果它找到这个入口点，便将这个库添加到链接扩展所使用库的列表中。find\_library 与之类似，它允许你指定一个目录列表来搜索库。下面是我们链接 CD 播放器的 extconf.rb 内容。

```
require 'mkmf'
dir_config("cdjukebox")
have_library("cdjukebox", "new_jukebox")
create_makefile("CDPlayer")
```

某个特定库的位置可能会因主机系统的不同而不同。例如，X Window 系统在这一点上是很闻名的，它在不同的系统上位于不同的目录中。find\_library 命令会搜索所提供的目录列表，来查找正确的路径（这和 have\_library 不同，have\_library 只使用配置信息进行搜索）。例如，要创建使用 X Windows 和 JPEG 库的 Makefile，extconf.rb 可能包括：

```
require 'mkmf'
if have_library("jpeg", "jpeg_mem_init") and
 find_library("X11", "XOpenDisplay",
 "/usr/X11/lib", # list of directories
 "/usr/X11R6/lib", # to check
 "/usr/openwin/lib") # for library
then
 create_makefile("XThing")
else
 puts "No X/JPEG support available"
end
```

我们已经为这个程序添加了一些附加的功能。所有的 mkmf 命令在失败时返回 false。这意味着我们可以编写一个 extconf.rb，它只有当万事俱备时，才会生成 Makefile。标准的 Ruby 发布就是这样做的，它只尝试编译你的系统所支持的那些扩展。

你可能还希望扩展的代码能够根据目标使用环境的不同，来配置其特性。例如，我们的 CD 点唱机可以使用高性能的 MP3 解码器，如果用户最终已有安装的话。我们可以通过查找它的头文件来检验。

```
require 'mkmf'
dir_config('cdjukebox')
have_library('cdjb', 'CDPlayerNew')
have_header('hp_mp3.h')
create_makefile("CDJukeBox")
```

我们还可以检查目标环境中的所有库是否支持某个特定的函数。例如，`setpriority` 调用可能是有用的，但是并非在所有的系统上都存在。我们可以通过下面的代码来检验：

```
require 'mkmf'
dir_config('cdjukebox')
have_func('setpriority')
create_makefile("CDJukeBox")
```

如果 `have_header` 或 `have_func` 找到了目标，它们均会定义预处理器的常量。名字的形式是，将目标名转换为大写字母且在前面加上 `HAVE_`。你的 C 代码则可以利用这一宏定义，例如：

```
#if defined(HAVE_HP_MP3_H)
include <hp_mp3.h>
#endif
#if defined(HAVE_SETPRIORITY)
 err = setpriority(PRIOR_PROCESS, 0, -10)
#endif
```

如果你有特殊的需要，而这些 `mkmf` 的命令无法满足时，你的程序可以直接向全局变量 `$CFLAGS` 和 `$LFLAGS` 中添加选项，这些选项会被分别传入到编译器和链接器。

有时，你创建的 `extconf.rb` 好像是无法工作的。你提供给它库名，但它却断定并不存在这样的库。你调整了又调整，`mkmf` 还是无法找到你需要的库。如果你可以找出幕后发生的确切状况，就好了。好的，你可以。当你每次运行 `extconf.rb` 脚本时，`mkmf` 会产生一个日志文件，包括了它完成的详细步骤。如果你查看 `mkmf.log`，会发现程序曾试图查找你所请求库的步骤。有时候，尝试手工地运行这些步骤，会帮助你解决问题。

## 安装目标

你的 `extconf.rb` 所产生的 `Makefile`，会包括一个“`install`”目标。它会将你的共享库对象正确拷贝到你（或你用户）的本地文件系统中。目标与你运行 `extconf.rb` 时使用的 Ruby 解释器的安装位置有关。如果你的系统中安装了多个 Ruby 解释器，你的扩展会被安装到那个运行 `extconf.rb` 的 Ruby 解释器的目录树中。

除了安装共享库之外，`extconf.rb` 还会查看是否存在 `lib/` 子目录。如果有，它会将这些 Ruby 文件随着共享库一同安装。如果你将编写扩展的工作分成两部分，包括底层的 C 代码和高层的 Ruby 代码时，这是很有用处的。

## 21.6.2 静态链接 Static Linking

最后，如果你的系统不支持动态链接，或者如果你有一个扩展模块希望静态链接到 Ruby 本身中，编辑 Ruby 发布中的 `ext/Setup` 将扩展的目录列表添加到该文件中。在扩展的目录中，创建一个名为 `MANIFEST` 的文件，其中包括扩展的所有文件（源文件、`extconf.rb`、`lib/` 等等）。然后重新构建 Ruby。`Setup` 中列出的扩展将会被静态地链接到 Ruby 可执行程序中。如果你想禁止所有的动态链接，并静态地链接所有的扩展，编辑 `ext/Setup` 加入下面的选项。

```
option nodynamic
```

## 21.6.3 一个捷径 A Shortcut

如果你希望扩展一个现有的用 C 或 C++ 编写的库，你可以调查一下 SWIG (<http://www.swig.org>)。SWIG 是一个接口生成器：它使用库的定义（通常从头文件获得）并自动生成从其他语言访问这个库的粘合（glue）代码。SWIG 支持 Ruby，意味着它可以生成将外部库包装到（wrap）Ruby 类的 C 源文件。

# 21.7 内嵌 Ruby 解释器 Embedding a Ruby Interpreter

除了通过添加 C 代码来扩展 Ruby 外，你还可以把问题反过来，并把 Ruby 嵌入到你的应用中。你有两种方式来这样做。一是通过调用 `ruby_run` 来让解释器得到控制权。这是最简单的方式，但是有一个显著的劣势——解释器永远不会从 `ruby_run` 调用中返回。下面是一个示例。

```
#include "ruby.h"
int main(void) {
 /* ... our own application stuff ... */
 ruby_init();
 ruby_init_loadpath();
 ruby_script("embedded");
 rb_load_file("start.rb");
 ruby_run();
 exit(0);
}
```

为了初始化 Ruby 解释器，你可以调用 `ruby_init()`。但是在某些平台上，你需要在此之前施行一些特殊的步骤。

```
#if defined(NT)
 NtInitialize(&argc, &argv);
#endif
#if defined(__MACOS__) && defined(__MWERKS__)
 argc = ccommand(&argv);
#endif
```

查看 Ruby 发布中的 `main.c`，看看你的平台需要哪些特殊的宏定义或设置。

你需要 Ruby 的头文件和库文件来编译这些嵌入的代码。在我的机器上（Mac OS X），我将 Ruby 1.8 安装到一个私有目录中，所以我的 `Makefile` 看起来如下所示。

```
WHERE=/Users/dave/ruby1.8/lib/ruby/1.8/powerpc-darwin/
CFLAGS=-I$(WHERE) -g
LDFLAGS=-L$(WHERE) -lruby -ldl -lobjc
embed: embed.o
 $(CC) -o embed embed.o $(LDFLAGS)
```

第二种嵌入 Ruby 的方式是，允许 Ruby 代码和你的 C 代码相互交互：C 代码调用 Ruby 代码，接着 Ruby 代码作出响应。你和寻常一样初始化解释器。然后，不再进入解释器的主循环，相反你调用 Ruby 代码中的特定方法。当这些方法返回时，你的 C 代码则收回控制权。

不过，其中有一些问题。如果 Ruby 代码引发了一个异常而没有被捕捉，那么你的 C 程序会退出。为了克服这一点，你需要像解释器那样保护所有可能导致异常的调用。这会变得很棘手。`rb_protect` 方法可以包装对另一个 C 函数的调用。后者应该调用我们的 Ruby 方法。然而，`rb_protect` 包装的方法被定义为只能接收一个参数。要传入多个参数将牵扯到某些丑陋的 C 转型。

让我们考察一个示例。这里是一个简单的 Ruby 类，它实现了一个方法返回从 1 到 `max` 的和。

```
class Summer
 def sum(max)
 raise "Invalid maximum #{max}" if max < 0
 (max*max + max)/2
 end
end
```

让我们编写一个 C 程序以让它多次调用这个类的实例。要创建它的实例，我们需要得到类对象（通过查找一个顶层的常量，其名字亦是类的名字）。然后我们可以让 Ruby 创建这个类的实例——实际上 `rb_class_new_instance` 等同于 `Class.new`。（开头两个

为 0 的参数是变量数和指向变量本身的空指针。) 一旦我们得到这个对象，就可以使用 `rb_funcall` 调用它的 `sum` 方法。

```
#include "ruby.h"
static int id_sum;
int Values[] = { 5, 10, 15, -1, 20, 0 };
static VALUE wrap_sum(VALUE args) {
 VALUE *values = (VALUE *)args;
 VALUE summer = values[0];
 VALUE max = values[1];
 return rb_funcall(summer, id_sum, 1, max);
}
static VALUE protected_sum(VALUE summer, VALUE max) {
 int error;
 VALUE args[2];
 VALUE result;
 args[0] = summer;
 args[1] = max;
 result = rb_protect(wrap_sum, (VALUE)args, &error);
 return error ? Qnil : result;
}

int main(void) {
 int value;
 int *next = Values;
 ruby_init();
 ruby_init_loadpath();
 ruby_script("embedded");
 rb_require("sum.rb");
 // get an instance of Summer
 VALUE summer = rb_class_new_instance(0, 0,
 rb_const_get(rb_cObject, rb_intern("Summer")));
 id_sum = rb_intern("sum");
 while (value = *next++) {
 VALUE result = protected_sum(summer, INT2NUM(value));
 if (NIL_P(result))
 printf("Sum to %d doesn't compute!\n", value);
 else
 printf("Sum to %d is %d\n", value, NUM2INT(result));
 }
 ruby_finalize();
 exit(0);
}
```

最后一件事：Ruby 解释器并非为嵌入到其他语言而编写的。最大的问题可能是它在全局变量中维护其状态，因此它不是线程安全的。你可以嵌入 Ruby——每个进程只有一个解释器。

在 C++ 程序中嵌入 Ruby 的一个很好的资源位于 <http://metaeditor.sourceforge.net/embed/>。这个页面还包括其他嵌入 Ruby 示例的链接。

### 21.7.1 API: 嵌入 Ruby API

#### API: Embedded Ruby API

`void ruby_init()`

设置并初始化解释器。这个函数应该在任何相关 Ruby 的函数之前被调用。

`void ruby_init_loadpath()`

初始化 \$: (加载路径) 变量；如果你的代码需要加载库模块，则有必要设置它。

`void ruby_options(int argc, char **argv)`

为解释器提供命令行选项。

`void ruby_script(char *name)`

将 Ruby 脚本的名字 (\$0) 设置为 *name*。

`void rb_load_file(char *file)`

将指定的文件加载到解释器中。

`void ruby_run()`

运行解释器。

`void ruby_finalize()`

关闭解释器。

在另一个程序中嵌入 Ruby 解释器的其他示例，请参见 `eruby`，描述开始于第 242 页。

## 21.8 将 Ruby 连接到其他语言

### Bridging Ruby to Other Languages

迄今为止，我们已经讨论了通过添加用 C 编写的例程来扩展 Ruby。不过，你也可以用任何语言来编写扩展，只要能将两个语言用 C 连接起来。几乎任何事都是可能发生的，包括 Ruby 和 C++、Ruby 和 Java 等笨拙的联姻。

你可能完成同样的事而无须诉诸于 C 代码。例如，你可以使用中间件（例如 SOAP 或 COM）将 Ruby 连接到其他语言。更多细节请参见 SOAP 一节（第 249 页）以及开始于第 269 页的 Windows 自动化一节。

## 21.9 Ruby C 语言 API

### Ruby C Language API

最后，但是绝非不重要，有一些 C 级别的函数，你可能发现在编写扩展时它们非常有用。

某些函数需要一个 ID：你可以使用 `rb_intern` 得到某个字符串对应的 ID，并使用 `rb_id2name` 从 ID 重建这个字符串名字。

因为大部分 C 函数在 Ruby 中有对等的方法，并且已经在书中的其他地方详细描述了，这里对它们的描述比较简略。

下面的列表是不完整的。还有更多的函数——实在因太多而无法全部记录在这里。如果你需要的函数无法在这里找到，请查看 `ruby.h` 或 `intern.h` 寻找类似候选。而且，在每个源文件的底部或接近底部，是一组描述了从 Ruby 方法到 C 函数之间绑定的方法定义。你也可以直接调用 C 函数，或查找包装了你所需调用的函数。下面的列表，基于 `README.EXT` 中的列表，显示了解释器中的主要源文件。

#### Ruby 语言核心

`class.c, error.c, eval.c, gc.c, object.c, parse.y, variable.c`

#### 实用函数

`dln.c, regex.c, st.c, util.c`

#### Ruby 解释器

`dmyext.c, inits.c, keywords main.c, ruby.c, version.c`

#### 基础库

`array.c, bignum.c, compar.c, dir.c, enum.c, file.c, hash.c, io.c, marshal.c, math.c, numeric.c, pack.c, prec.c, process.c, random.c, range.c, re.c, signal.c, sprintf.c, string.c, struct.c, time.c`

### 21.9.1 API：定义类

#### API: Defining Classes

`VALUE rb_define_class (char *name, VALUE superclass)`

用给定的 `name` 和 `superclass`（对 `Object` 类使用 `rb_cObject`）定义一个顶层的类。

`VALUE rb_define_module (char *name)`

用给定的 `name` 定义一个新的顶层模块。

`VALUE rb_define_class_under (VALUE under, char *name, VALUE superclass)`

在 `under` 类或模块之下定义一个嵌套类。

`VALUE rb_define_module_under (VALUE under, char *name)`

在 *under* 类或模块之下定义一个嵌套模块。

`void rb_include_module (VALUE parent, VALUE module)`

将指定的 *module* 包含到 *parent* 类或模块中。

`void rb_extend_object (VALUE obj, VALUE module)`

用 *module* 扩展 *obj*。

`VALUE rb_require (const char *name)`

等同于 `require name`。返回 `Qtrue` 或 `Qfalse`。

## 21.9.2 API: 定义结构

### API: Defining Structures

`VALUE rb_struct_define (char *name, char *attribute..., NULL)`

用给定的属性定义一个新的结构。

`VALUE rb_struct_new (VALUE sClass, VALUE args..., NULL)`

用给定的属性值创建一个 *sClass* 的实例。

`VALUE rb_struct_aref (VALUE struct, VALUE idx)`

返回由 *idx* 索引或命名的元素。

`VALUE rb_struct_aset (VALUE struct, VALUE idx, VALUE val)`

设置由 *idx* 索引或命名的属性值为 *val*。

## 21.9.3 API: 定义方法

### API: Defining Methods

在下面的某些方法定义中，参数 *argc* 指定了 Ruby 方法接收参数的个数。它可能有下面的值。

<i>argc</i>	函数原型
0..17	<code>VALUE func (VALUE self, VALUE arg...)</code> 使用这么多的实际参数来调用 C 函数
-1	<code>VALUE func (int argc, VALUE *argv, VALUE self)</code> 提供给 C 函数的可变个数参数是以 C 数组形式传入的
-2	<code>VALUE func (VALUE self, VALUE args)</code> 提供给 C 函数的可变个数参数是以 Ruby 数组形式传入的

在一个接收可变个数参数的函数中，你可以使用 C 函数 `rb_scan_args` 来解决问题（参见下面的描述）。

```
void rb_define_method(VALUE classmod, char *name,
 VALUE(*func)(), int argc)
```

用给定的 *name* 在 *classmod* 类或模块中定义一个实例方法，由 C 函数 *func* 实现并接收 *argc* 个参数。

```
void rb_define_alloc_func(VALUE classmod, VALUE(*func)())
```

为 *classmod* 标识分配器。

```
void rb_define_module_function(VALUE module, char *name,
 VALUE(*func)(), int argc)
```

用给定的 *name* 在 *module* 模块中定义一个方法，由 C 函数 *func* 实现并接收 *argc* 个参数。

```
void rb_define_global_function(char *name, VALUE(*func)(), int argc)
```

用给定的 *name* 定义一个全局函数（Kernel 的私有方法），由 C 函数 *func* 实现并接收 *argc* 个参数。

```
void rb_define_singleton_method(VALUE classmod, char *name,
 VALUE(*func)(), int argc)
```

用给定的 *name* 在类 *classmod* 中定义一个单体（singleton）方法，由 C 函数 *func* 实现并接收 *argc* 个参数。

```
int rb_scan_args(int argc, VALUE *argv, char *fmt, ...)
```

扫描参数列表，并赋值给类似 *scanf* 中的变量：*fmt* 是一个字符串，包括 0、1 或两个数字和后面的标志字符。第一个数字指明了必需参数的个数；第二个是可选参数的个数。*\** 意味着将剩余的参数装入到一个 Ruby 数组中。*&* 意味着会传入一个相关联的代码 *block*，并赋值给指定的参数（如果没有给出代码 *block*，会赋值为 *Qnil*）。在 *fmt* 字符串之后，给出了若干的 VALUE 指针（和 *scanf* 一样），对应要赋值的参数。

```
VALUE name, one, two, rest;
rb_scan_args(argc, argv, "12", &name, &one, &two);
rb_scan_args(argc, argv, "1*", &name, &rest);
```

```
void rb_undef_method(VALUE classmod, const char *name)
```

在给定的 *classmod* 类或模块中，取消给定方法 *name* 的定义。

```
void rb_define_alias(VALUE classmod, const char *newname,
 const char *oldname)
```

为 *classmod* 类或模块中的 *oldname* 定义一个别名。

## 21.9.4 API: 定义变量和常量

### API: Defining Variables and Constants

```
void rb_define_const(VALUE classmod, char *name, VALUE value)
```

使用给定的 *name* 和 *value* 定义 *classmod* 类或模块中的常量。

```
void rb_define_global_const(char *name, VALUE value)
```

用给定的 *name* 和 *value* 定义一个全局常量。

```
void rb_define_variable(const char *name, VALUE *object)
```

将 C 中创建的 *object* 地址以 *name* 为符号名暴露给 Ruby 命名空间。从 Ruby 看来，这是一个全局变量，因此 *name* 应该以一个\$开头。切记要尊重 Ruby 允许的变量名规则；违规命名的变量将无法从 Ruby 中访问。

```
void rb_define_class_variable(VALUE class, const char *name,
 VALUE val)
```

在指定的 *class* 中定义一个类变量 *name*（必须以@@为前缀），初始化为 *value*。

```
void rb_define_virtual_variable(const char *name,
 VALUE(*getter)(),
 void(*setter)())
```

将一个虚拟变量以\$name 暴露到 Ruby 命名空间。这个变量没有实际的存储；尝试得到或设置这个值，将调用原型如下的指定函数。

```
VALUE getter(ID id, VALUE *data,
 struct global_entry *entry);
void setter(VALUE value, ID id, VALUE *data,
 struct global_entry *entry);
```

你可能不需要使用 *entry* 参数，并可以安全地在函数声明中忽略它。

```
void rb_define_hooked_variable(const char *name,
 VALUE *variable,
 VALUE(*getter)(),
 void(*setter)())
```

定义当读写 *variable* 时调用的函数。参见 `rb_define_virtual_variable`。

```
void rb_define_READONLY_VARIABLE(const char *name,
 VALUE *value)
```

和 `rb_define_variable` 类似，但是在 Ruby 中是只读的。

```
void rb_define_attr(VALUE variable, const char *name, int read,
 int write)
```

使用指定的 *name* 为指定的 *variable* 创建访问方法。如果 *read* 非 0，创建一个读取方法；如果 *write* 非 0，创建一个写入方法。

```
void rb_global_variable(VALUE *obj)
```

将指定的地址注册给垃圾收集器。

## 21.9.5 API: 调用方法

### API: Calling Methods

```
VALUE rb_class_new_instance((int argc, VALUE *argv,
 VALUE klass))
```

返回 *klass* 类的一个新实例。*argv* 是 *argc* 个参数的数组。

```
VALUE rb_funcall(VALUE recv, ID id, int argc, ...)
```

调用 *recv* 对象中由 *id* 指定的方法，*argc* 是参数的个数，后面是实际的参数（可能没有参数）。

```
VALUE rb_funcall2(VALUE recv, ID id, int argc, VALUE *args)
```

调用 *recv* 对象中由 *id* 指定的方法，*argc* 是参数的个数，参数本身保存在 C 数组 *args* 中。

```
VALUE rb_funcall3(VALUE recv, ID id, int argc, VALUE *args)
```

和 *rb\_funcall2* 一样，但是不会调用私有的方法。

```
VALUE rb_apply(VALUE recv, ID name, int argc, VALUE args)
```

调用 *recv* 对象中由 *id* 指定的方法，*argc* 是参数的个数，而参数本身保存在 Ruby 数组 *args* 中。

```
ID rb_intern(char *name)
```

返回指定 *name* 对应的 ID。如果名字不存在，会为它创建一个符号表项 (symbol table entry)。

```
char * rb_id2name(ID id)
```

返回指定 *id* 的名字。

```
VALUE rb_call_super(int argc, VALUE *args)
```

调用与当前方法同名的超类中的方法。

## 21.9.6 API: 异常

### API: Exceptions

```
void rb_raise(VALUE exception, const char *fmt, ...)
```

引发一个 *exception*。指定的字符串 *fmt* 和余下参数的解释方式与 `printf` 相同。

```
void rb_fatal(const char *fmt, ...)
```

引发一个 `Fatal` 异常，并终止进程。不会调用 `rescue` 的 `block`，但是 `ensure block` 会被调用。指定的字符串 *fmt* 和余下参数的解释方式与 `printf` 相同。

```
void rb_bug(const char *fmt, ...)
```

立即终止进程——不会调用任何形式的处理函数。指定的字符串 *fmt* 和余下参数的解释方式与 `printf` 相同。你只有遇到致命 `bug` 时才应该调用这个函数。你不应该编写致命的 `bug`，是不是？

```
void rb_sys_fail(const char *msg)
```

使用指定的 *msg*，引发一个特定于平台的异常，对应最后得知的系统错误。

```
VALUE rb_rescue(VALUE (*body)(), VALUE args, VALUE(*rescue)(),
 VALUE rargs)
```

用给定的 *args* 执行 *body*。如果引发了 `StandardError` 异常，则用给定的 *rargs* 执行 *rescue*。

```
VALUE rb_ensure(VALUE(*body)(), VALUE args, VALUE(*ensure)(),
 VALUE eargs)
```

用给定的 *args* 执行 *body*。不管是否引发了异常，在 *body* 完成之后，均使用给定的 *eargs* 执行 *ensure*。

```
VALUE rb_protect(VALUE (*body)(), VALUE args, int *result)
```

用给定的 *args* 执行 *body*，如果期间引发了任何异常，则在 *result* 中返回非 0 的值。

```
void rb_notimplement()
```

引发一个 `NotImpError` 异常，指示外围的函数没有完成，或者在该平台上无法得到。

```
void rb_exit(int status)
```

用指定的 *status* 退出 Ruby。引发一个 `SystemExit` 异常，并调用注册的退出函数和完成函数。

```
void rb_warn(const char *fmt, ...)
```

无条件地向标准错误输出中发出一个警告消息。指定的字符串 *fmt* 和余下参数的解释方式与 printf 相同。

```
void rb_warning(const char *fmt, ...)
```

有条件地向标准错误输出中发出一个警告消息，如果调用 Ruby 时使用了 -w 选项。指定的字符串 *fmt* 和余下参数的解释方式与 printf 相同。

## 21.9.7 API: 迭代器

### API: Iterators

```
void rb_iter_break()
```

中断外围的迭代器 block。

```
VALUE rb_each(VALUE obj)
```

调用指定 *obj* 对象的 each 方法。

```
VALUE rb_yield(VALUE arg)
```

将执行转移到当前上下文的迭代器 block，将 *arg* 作为参数传入。多个值可以作为一个数组来传入。

```
int rb_block_given_p()
```

如果 yield 可以执行当前上下文的 block，则返回 true——也就是说，如果当前方法传入了一个代码 block 并可以被调用。

```
VALUE rb_iterate(VALUE (*method)(), VALUE args,
 VALUE (*block)(), VALUE arg2)
```

使用参数 *args* 和 *block* 调用 *method*。该方法中的 yield 将使用传给它的参数和第二个参数 *arg2* 调用 *block*。

```
VALUE rb_catch(const char *tag, VALUE (*proc)(), VALUE value)
```

等同于 Ruby 的 catch。

```
void rb_throw(const char *tag, VALUE value)
```

等同于 Ruby 的 throw。

## 21.9.8 API: 访问变量

### API: Accessing Variables

```
VALUE rb_iv_get(VALUE obj, char *name)
```

返回指定对象 *obj* 中的实例变量 *name*（必须以 @ 为前缀）。

```

VALUE rb_ivar_get(VALUE obj, ID name)

 返回指定对象 obj 中的实例变量 name。

VALUE rb_iv_set(VALUE obj, char *name, VALUE value)

 将指定对象 obj 中的实例变量 name（必须以@为前缀）的值设置为
 value。返回 value。

VALUE rb_ivar_set(VALUE obj, ID name, VALUE value)

 将指定对象 obj 中的实例变量 name 的值设置为 value。返回 value。

VALUE rb_gv_set(const char *name, VALUE value)

 将全局变量 name（$前缀是可选的）设置为 value。返回 value。

VALUE rb_gv_get(const char *name)

 返回全局变量 name（$前缀是可选的）。

void rb_cvar_set(VALUE class, ID name, VALUE val, int unused)

 将指定 class 中的类变量 name 设置为 value。

VALUE rb_cvar_get(VALUE class, ID name)

 返回指定 class 中的类变量 name。

int rb_cvar_defined(VALUE class, ID name)

 如果 class 中定义了类变量 name，则返回 Qtrue；否则，返回 Qfalse.

void rb_cv_set(VALUE class, const char *name, VALUE val)

 将指定 class 中的类变量 name（必须以@@为前缀）为 value。

VALUE rb_cv_get(VALUE class, const char *name)

 返回指定 class 中的类变量 name（必须以@@为前缀）。

```

### 21.9.9 API：对象状态

API: Object Status

```

OBJ_TAINT(VALUE obj)

 将给定对象 obj 标记为 tainted。

int OBJ_TAINTED(VALUE obj)

 如果给定的 obj 是 tainted，则返回一个非 0 值。

OBJ_FREEZE(VALUE obj)

 将给定对象 obj 标记为被冻结了。

```

```
int OBJ_FROZEN(VALUE obj)
```

如果给定的 *obj* 被冻结了，则返回非 0 值。

```
SafeStringValue(VALUE str)
```

1.8 如果当前的安全级别>0 并且 *str* 是 tainted（不可信的），则引发 SecurityError 异常，如果 *str* 不是 T\_STRING 或\$SAFE>=4，则引发 TypeError 异常。

```
int rb_safe_level()
```

返回当前的安全级别。

```
void rb_secure(int level)
```

如果 *level*<=当前的安全级别，则引发 SecurityError 异常。

```
void rb_set_safe_level(int newlevel)
```

将当前的安全级别设置为 *newlevel*。

### 12.9.10 API: 常用的方法

#### API: Commonly Used Methods

```
VALUE rb_ary_new()
```

返回默认大小的新 Array。

```
VALUE rb_ary_new2(long length)
```

返回一个长度为 *length* 的新 Array。

```
VALUE rb_ary_new3(long length, ...)
```

返回一个长度为 *length* 的新 Array，并用余下的参数填充它。

```
VALUE rb_ary_new4(long length, VALUE *values)
```

返回一个长度为 *length* 的新 Array，并用 C 数组的值填充它。

```
void rb_ary_store(VALUE self, long index, VALUE value)
```

将 *value* 保存到数组 *self* 的 *index* 元素中。

```
VALUE rb_ary_push(VALUE self, VALUE value)
```

将 *value* 压入到数组 *self* 的底部。返回 *value*。

```
VALUE rb_ary_pop(VALUE self)
```

从数组 *self* 中删除并返回最后一个元素。

```
VALUE rb_ary_shift(VALUE self)
```

从数组 *self* 中删除并返回第一个元素。

VALUE **rb\_ary\_unshift**( VALUE self, VALUE value )  
 将 *value* 压入到数组 *self* 的前部。返回 *value*。

VALUE **rb\_ary\_entry**( VALUE self, long index )  
 返回 *self* 数组中第 *index* 个元素。

int **rb\_respond\_to**( VALUE self, ID method )  
 如果 *self* 可以响应 *method*, 则返回非 0 值。

VALUE **rb\_thread\_create**( VALUE (\*func)(), void \*data )  
 在一个新线程中运行 *func*, 将 *data* 作为参数传入。

VALUE **rb\_hash\_new**( )  
 返回一个新的、空的 Hash。

VALUE **rb\_hash\_aref**( VALUE self, VALUE key )  
 返回 *self* 中与 *key* 对应的元素。

VALUE **rb\_hash\_aset**( VALUE self, VALUE key, VALUE value )  
 设置 *self* 中 *key* 对应的值。返回 *value*。

VALUE **rb\_obj\_is\_instance\_of**( VALUE obj, VALUE klass )  
 如果 *obj* 是 klass 的一个实例, 则返回 Qtrue。

VALUE **rb\_obj\_is\_kind\_of**( VALUE obj, VALUE klass )  
 如果 *klass* 是 *obj* 的类, 或者 *klass* 是 *obj* 的一个超类。

VALUE **rb\_str\_new**( const char \*src, long length )  
 返回一个新的 String, 用 *src* 中 *length* 个字符初始化。

VALUE **rb\_str\_new2**( const char \*src )  
 返回一个新的 String, 用一个 null 结尾的 C 字符串初始化。

VALUE **rb\_str\_dup**( VALUE str )  
 返回一个从 *str* 复制得到的新 String 对象。

VALUE **rb\_str\_cat**( VALUE self, const char \*src, long length )  
 将字符串 *src* 中的 *length* 个字符串联到 String *self*。返回 *self*。

VALUE **rb\_str\_concat**( VALUE self, VALUE other )  
 将 *other* 串联到字符串 *self*。返回 *self*。

VALUE **rb\_str\_split**( VALUE self, const char \*delim )  
 返回 *self* 被 *delim* 分隔后得到的一个 String 对象的数组。

---

## 第3部分 Ruby的核心

---

## Part III Ruby Crystallized

---



*Programming Ruby 中文版, 第2版*



## Ruby 语言

# The Ruby Language

本章是对 Ruby 语言从头到脚的一个介绍。本章的大部分内容是语言本身的语法和语义——基本上忽略了内建的类和模块（这些内容将从第 423 页开始详细介绍）。然而，有时 Ruby 用库实现的某些特性，对大多数语言来说可能属于其基本语法。本章也包含了这样的方法，并在页边空白上用“库”将它们标出。

本章的内容看着很面熟——因为大部分内容我们已经在前面的章节中讲述了。你可以把本章看成是 Ruby 语言核心的完备参考。

### 22.1 源代码编排 Source Layout

Ruby 程序是用 7 位 ASCII、Kanji（使用 EUC 或者 SJIS）或者 UTF-8 来表示的。如果要使用非 7 位 ASCII 的其他字符集，那么必须适当地设置 KCODE 选项，如第 179 页所述。

Ruby 是基于行的语言，Ruby 的表达式和语句都以行尾结束，除非解析器能够确定语句是不完整的——比如行最后一个符号是操作符或者逗号。分号可以区分一行中的多个表达式。你也可以在行尾加一个反斜线表示延续到下一行。注释以#开始，到物理行结束为止。在语法分析时注释将被忽略。

```
a = 1
b = 2; c = 3
d = 4 + 5 +
 6 + 7 # 不需要 '\'
e = 8 + 9 \
 + 10 # 需要 '\'
```

Ruby 会忽略以`=begin` 开头的和以`=end` 开头的行之间的物理行，它们可以被用来注释掉一段代码或者用在源代码中嵌入文档。

Ruby 一次性读入整个程序，所以你可以用管道将程序传递给 Ruby 解释器的标准输入流。

```
echo 'puts "Hello"' | ruby
```

在源代码的任意位置，如果 Ruby 遇到只含有“`__END__`”且前后没有空格的行，那么 Ruby 将认为该行是程序的结束行——后面的行都不会被当作程序代码。不过，使用全局的 `IO` 对象 `DATA`，可以将这些行读入到运行的程序中，第 337 页有详细介绍。

### 22.1.1 BEGIN 和 END Blocks BEGIN and END Blocks

Ruby 的每个源代码文件都可以声明当自己被装载时要执行的代码 `block` (`BEGIN block`) 和程序运行结束后要执行的 `block` (`END block`)。

```
BEGIN {
 开始代码
}

END {
 结束代码
}
```

一个程序可以包含多个 `BEGIN` 和 `END block`。`BEGIN block` 的执行顺序和它出现的顺序相同。`END block` 的执行顺序与此相反。

### 22.1.2 常规分隔输入 General Delimited Input

除了常用的引用 (quoting) 机制外，还可以用其他形式来表示字符串字面量 (literal)、数组、正则表达式和 shell 命令，这就是常规分隔语法。所有这些字面量都是以一个百分号开头，后跟一个指明字面量类型的字符的。表 22.1 概述了这些字符；实际的字面量在本章后面的章节有相应的描述。

类型字符后面的是分隔字符，分隔字符可以是任意非字母或非多字节的字符。如果分隔符是`(`、`[`、`{`或`<`其中之一，那么从它到对应的闭合分隔符之间的字符，属字面量所有，同时嵌套的分隔符对也考虑在内。对于其他的分隔符形式，在分隔符下一次出现之前的所有字符，属字面量所有。

```
%q/this is a string/
%q-string-
%q(a (nested) string)
```

表 22.1 常规分隔输入

类 型	意 义	参 见
%q	单引号字符串	320
%Q, %	双引号字符串	320
%w, %W	字符串数组	322
%r	正则表达式模式	324
%x	Shell 命令	338

分隔字符可以跨越多行；行结束符以及后续行开始处的空格，都将被包含到字符串中。

```
meth = %q{def fred(a)
 a.each {|i| puts i }
 end}
```

## 22.2 基本类型

### The Basic Types

Ruby 的基本类型包括数字、字符串、数组、散列表（hash）、区间（range）、符号和正则表达式。

#### 22.2.1 整数和浮点数

##### Integer and Floating-Point Numbers

Ruby 的整数是 Fixnum 类或 Bignum 类的对象。Fixnum 对象可以容纳比本机字长少一位的整数。当一个 Fixnum 超过这个范围时，它将会自动转换成 Bignum 对象，Bignum 对象的表示范围仅受可用内存大小的限制。如果 Bignum 对象的操作结果可以用 Fixnum 表示，那么结果将以 Fixnum 类型返回。

- 1.8 整数由一个可选的符号标记、一个可选的进制指示符（0 代表八进制，0d 代表十进制，0x 代表十六进制，0b 代表二进制）和一个相应进制的字符串组成。数字串中的下画线字符将被忽略。

123456	=> 123456	# Fixnum
0d123456	=> 123456	# Fixnum
123_456	=> 123456	# Fixnum -underscore ignored
-543	=> -543	#Fixnum -negative number
0xaabb	=> 43707	# Fixnum -hexadecimal
0377	=> 255	# Fixnum -octal
-0b10_1010	=> -42	# Fixnum -binary (negated)
123_456_789_123_456_789	=> 123456789123456789	# Bignum

你可以在一个 ASCII 字符前加一个问号来获得其对应的整数值。Ctrl 组合键字符可以由?\\c-x 和?\\cx (Ctrl+x 的组合键是 x&0x9f) 来产生。Meta 字符 (x |0x80) 可以由

?M-x 来生成。Meta 和 Ctrl 的组合键可以由?M-\C-x 生成。使用?\序列你能得到反斜线字符的整数值。

```
?a => 97 # ASCII character
?\n => 10 # code for a newline (0x0a)
?\C-a => 1 # control a = ?A & 0x9f = 0x01
?\M-a => 225 # meta sets bit 7
?\M-\C-a => 129 # meta and control a
?\C-? => 127 # delete character
```

一个带有小数点和/或指数的数字字面量被认为是 Float 对象，Float 对象和本机上的 double 数据类型大小一样。小数点后必须跟一个数字，因为像 1.e3 这样的串将试图调用 Fixnum 类的 e3 方法。Ruby 1.8 还要求在小数点前至少要有一个数字。

```
12.34 → 12.34
-0.1234e2 → -12.34
1234e-2 → 12.34
```

## 字符串

Ruby 提供了多种机制来生成字面量字符串。每种机制都产生 string 类型的对象。不同机制的区别在于如何分隔字符串以及字面量内容中会进行哪些替换。

单引号引起的字符串字面量（例如'stuff' 和 %q/stuff/）执行的替换最少。这两者都会将 \\ 序列转换为单个反斜线，单引号的形式还会将 \' 转换为单引号。所有其他的反斜线都不进行转换。

```
'hello' → hello
'a backslash \'\\\'\' → a backslash '\'
%q/simple string/ → simple string
%q(nesting (really) works) → nesting (really) works
%q no_blanks_here ; → no_blanks_here
```

双引号字符串（例如"stuff"，%Q/stuff/ 和 %/stuff/）还执行额外的替换，详见下页的表 22.2。

```
a = 123
"\123mile" → Smile
"Say \"Hello\""
%Q!"I said 'nuts'," I said!
%Q{Try #{a + 1}, not #{a-1}}
%<Try #{a + 1}, not #{a-1}>
"Try #{a + 1}, not #{a-1}"
%{ #(a = 1; b = 2; a + b) } → Try 124, not 122
→ Try 124, not 122
→ Try 124, not 122
→ 3
```

字符串可以跨行，在这种情况下串中包含换行符。也可以使用 here documents 来表示很长的字符串字面量。每当 Ruby 遇到 <<identifier 或者 <<quoted string 时，将用后续的逻辑输入行来生成字符串字面量，直到遇到以 identifier 或者 quoted string 开头的行时才结束。

表 22.2 双引号字符串中允许的替换

\a	Bell/alert (0x07)	\nnn	Octal <i>nnn</i>
\b	Backspace (0x08)	\xnn	Hex <i>nn</i>
\e	Escape (0x1b)	\cx	Control-x
\f	Formfeed (0x0c)	\C-x	Control-x
\n	Newline (0x0a)	\M-x	Meta-x
\r	Return (0x0d)	\M-\C-x	Meta-control-x
\s	Space (0x20)	\x	x
\t	Tab (0x09)	# {code}	Value of code
\v	Vertical tab (0x0b)		

你也可以在<<字符后紧跟一个减号，让终止符可以从左侧页空白缩进。如果终止符是由引号引起的字符串，那么该引号对应的替换规则将被应用到 here document，否则使用双引号替换规则。

```
print <<HERE
Double quoted \
here document.
It is #{Time.now}
HERE

print <<-'THERE'
 This is single quoted.
 The above used #{Time.now}
THERE
```

输出结果：

```
Double quoted here document.
It is Thu Aug 26 22:37:12 CDT 2004
 This is single quoted.
 The above used #{Time.now}
```

输入中相连的单引号字符串和双引号字符串将被连接成一个 String 对象。

```
'Con' "cat" 'en' "ate" → "Concatenate"
```

字符串被保存在 8 位字节<sup>1</sup>序列中，每个字节可以含有 256 个 8 位值中的任意一个，包括 null 和换行符。表 22.2 中的替换序列可以让我们方便且可移植地将不可打印字符插入到字符串中。

<sup>1</sup> 为了在日本使用，jcode 库支持对 EUC、SJIS 或者 UTF-8 编码的字符串进行操作，不过，底层的字符串仍作为字节序列被访问。

当字符串字面量用于赋值语句或者作为参数时，一个新的 `String` 对象将被创建。

```
3.times do
 print 'hello'.object_id, " "
end
```

输出结果：

```
937140 937110 937080
```

从第 606 页开始有 `String` 类的详细文档。

## 22.2.2 区间 Ranges

除了用在条件表达式中，`expr..expr` 和 `expr...expr` 还能构成 `Range` 对象。两个点的形式是闭合区间，而三点的形式是半开半闭的。详情请参考第 597 页对 `Range` 类的描述。如果想获得 `Range` 的其他用法，请参考第 342 页条件表达式的描述。

## 22.2.3 数组 Arrays

数组类的字面量是在方括号间由逗号分隔的一连串对象引用组成的。尾部的逗号将被忽略。

```
arr = [fred, 10, 3.14, "This is a string", barney("pebbles"),]
```

1.8 数组也可以用简写形式 `%w` 和 `%W` 来构成。小写字母的形式将空格隔开的 `token` 提取为连续的数组元素。在单个字符串内不执行替换。大写字母的形式虽然也把单词列表转换成数组，但是对每个词执行和双引号字符串一样的替换规则。词之间的空格可以用反斜线转义。这是第 318—319 页描述的常规分隔输入的一种形式。

```
arr = %w(fred wilma barney betty great\ gazoo)
arr → ["fred", "wilma", "barney", "betty", "great gazoo"]
arr = %w(Hey!\tIt is now -#{Time.now}-)
arr → ["Hey!\\tIt", "is", "now", "-\\#{Time.now}-"]
arr = %W(Hey!\tIt is now -#{Time.now}-)
arr → ["Hey!\\tIt", "is", "now", "-Thu Aug 26 22:37:13 CDT 2004-"]
```

## 22.2.4 散列表 Hashes

Ruby 的 Hash 字面量可以由花括号中的键/值对列表构成，由逗号分隔键/值对，键和值之间由`=>`序列分隔。尾部的逗号将被忽略。

```
colors = { "red" => 0xf00,
 "green" => 0x0f0,
 "blue" => 0x00f
 }
```

并不要求一个具体的散列表中的所有“key”和/或“值”必须为相同类型。

### 对散列表键的要求

散列表键必须能够响应 `hash` 消息并返回一个散列码（hash code），且某个键对应的散列码不能改变。散列表中使用的键也必须能用 `eq?` 来比较。如果 `eq?` 在比较两个键时返回真，那么这两个键必定具有相同的散列码。这意味着某些类（例如数组和散列表）不适于做键，因为它们的 `hash` 值可能会随其内容而发生变化。

如果你保存了键对象的一个外部引用，并使用该引用修改了对象，那么这也将修改它的散列码，进而基于该键的查询将无效。

因为字符串是最常用的键，且字符串内容会经常变化，所以 Ruby 会对字符串键进行特别处理。如果你使用 `String` 对象作为 `hash` 键，则 `hash` 将在内部复制该字符串，并使用该拷贝作为键。此拷贝将被冻结，对原字符串的任意改变都不会影响 `hash`。  
1.8

如果你实现了自定义的类，并用该类的对象实例作为 `hash` 键，那么你需要确保（a）一旦对象被创建，它的散列码就不再改变，或者（b）每当键的散列码发生变化时都调用 `Hash#rehash` 方法重新对散列表进行索引。

## 22.2.5 符号 Symbols

Ruby 符号是一个对应字符串（通常是一个名字）的标识符。你可以通过在名字前加一个冒号来构造该名字的符号，也可以在任意字符串字面量前加一个冒号来创建该字符串的符号。在双引号字符串中会发生替换。不管程序如何使用名字，一个具体的名字或者字符串总是产生同样的符号。  
1.8

<code>:Object</code>		
<code>:my_variable</code>		
<code>:"Ruby rules"</code>		
<code>a = "cat"</code>		
<code>:'catsup'</code>	→	<code>:catsup</code>
<code>:"#(a)sup"</code>	→	<code>:catsup</code>
<code>:'#{a}sup'</code>	→	<code>:"\#{a}sup"</code>

其他语言称这个过程为 *interning*，而称符号为原子。

## 22.2.6 正则表达式

### Regular Expressions

正则表达式的字面量 (literal) 是 `Regexp` 类型的对象。可以通过调用 `Regexp.new` 构造函数显式创建正则表达式，也可以使用字面量形式/*pattern*/ 和 `%r{pattern}` 隐式创建。`%r` 构造体 (construct) 是常规分隔输入的一种形式（第 318—319 页有其描述）。

```
/pattern/
/pattern/ options
%r{pattern}
%r{pattern} options
Regexp.new('pattern' [, options])
```

#### 正则表达式选项

正则表达式可以包含一个或多个选项来修改模式和字符串的匹配方式。如果你使用字面量 (literal) 来创建 `Regexp` 对象，那么一个或者多个字符可以作为选项紧跟在结束符后面。如果你使用 `Regexp.new`，那么这些选项是作为构造函数中第二个参数的若干常量。

- 1.8.
  - i **大小写无关。** 模式匹配将忽略模式和字符串中字符的大小写。通过设置`$=`来使匹配大小写无关的方式已经过时了。
  - o **替换一次。** 正则表达式中的任意#...替换仅在第一次求解 (evaluate) 它的时候执行替换。否则，替换在每次字面量生成 `Regexp` 对象时执行。
  - m **跨行模式。** 通常，“.”匹配换行符之外的任意字符。使用`/m` 选项，“.”将匹配任意字符。
  - x **扩展模式。** 复杂的正则表达式可读性较差。 `x` 选项允许你向模式中插入空格、换行符和注释以提高它的可读性。

还有另外一组选项可以设置正则表达式的语言编码。如果没有这些选项，那么将使用解释器的默认编码（可以用`-K` 或者 `$KCODE` 设置）。

n: 没有编码 (ASCII)	e: EUC
s: SJIS	u: UTF-8

#### 正则表达式模式

**常规字符** 除了`.`, `|`, `(`, `)`, `[`, `\`, `^`, `{`, `+`, `$`, `*`, 和`?`外的所有字符都与它们自身相匹配。要匹配这些特殊字符，需要在其前面加反斜线字符。

^	匹配行首。
\$	匹配行尾。
\A	匹配字符串的开始。

\z	匹配字符串的结尾。
\Z	匹配字符串的结尾，如果字符串以\n结尾，那么匹配\n前面的那个字符。
\b, \B	分别匹配词边界和非词边界。
\G	前面重复性搜索结束的位置（仅在某些情况下）。参考后面的详细描述。
[ <i>characters</i> ]	方括号表达式匹配方括号中列出的字符列表中的任意一个字符。在其他模式中具有特殊意义的字符 .   () [{+^\$*?} 在方括号中将失去其特殊意义。序列 \nnn, \xnn, \cx, \C-x, \M-x 和 \M-\C-x 仍有如第 321 页表 22.2 所示的特殊含义。序列 \d, \D, \s, \S, \w 和 \W 是一组字符的缩写，如第 72 页的表 5.1 所示。序列 [: <i>class</i> :] 匹配一个 POSIX 字符类别，如第 72 页的表 5.1 所示（注意左右方括号是类别表示法的一部分，所以模式 /[_[:digit:]]/ 匹配一个数字或者一个下画线）。序列 <i>c</i> <sub>1</sub> - <i>c</i> <sub>2</sub> 表示 <i>c</i> <sub>1</sub> 和 <i>c</i> <sub>2</sub> 之间的所有字符，包括 <i>c</i> <sub>1</sub> 和 <i>c</i> <sub>2</sub> 。字符 ] 和 - 必须紧跟在左括号的后面。一个脱字符号 (^) 紧跟在左括号后面将使得匹配反义——模式匹配任意不在字符类中的字符。
\d, \s, \w	分别是匹配数字、空格和词的字符类的缩写形式。第 72 页的表 5.1 有其概述。
\D, \S, \W	\d, \s 和 \w 的反义形式，分别与非数字字符、非空格字节和非词字符匹配。
. (句点)	出现在方括号外面，匹配换行符以外的任意字符（如果指定了/m 选项，它也可以匹配换行符）。
<i>re</i> *	匹配 0 个或多个 <i>re</i> 。 <sup>2</sup>
<i>re</i> +	匹配 1 个或多个 <i>re</i> 。
<i>re</i> {m, n}	匹配最少 “m” 个最多 “n” 个 <i>re</i> 。
<i>re</i> {m, }	匹配最少 “m” 个 <i>re</i> 。
<i>re</i> {m}	匹配刚好 “m” 个 <i>re</i> 。
<i>re</i> ?	匹配 0 个或 1 个 <i>re</i> 。默认情况下，*, + 和 {m, n} 都是最长匹配，在其后面加一个问号使它们执行最短匹配。
<i>re1</i>   <i>re2</i>	匹配 <i>re1</i> 或者 <i>re2</i> 。  的优先级很低。

<sup>2</sup> 译注：这里与后面的“re”均是 regular expression 的缩写，表示正则表达式的一个构造体 [construct]。

- (...) 括号用来组合正则表达式。比如模式 /abc+/ 和包含 1 个 *a*, 1 个 *b* 及一个或多个 *c* 的字符串相匹配。`/(\abc)+/` 匹配 1 个或者多个 *abc* 序列。括号也可以用来收集模式匹配的结果。Ruby 保存每一个左括号和其对应的右括号之间的匹配结果作为连续的组。在同一个模式中, `\1` 代表第一组匹配的结果, `\2` 代表第二组匹配的结果, 依此类推。在模式之外, 特殊变量 `$1`, `$2` 等起同样的作用。
- 1.8 锚点 `\G` 用在全局匹配方法 `String#gsub`、`String#gsub!`、`String#index` 和 `String#scan` 中。在重复匹配中, 它代表字符串内迭代中最后一个匹配结束的位置。开始时 `\G` 指向字符串开始的位置 (或者 `String#index` 第二个参数引用的字符)。

```
"a01b23c45 d56".scan(/ [a-z] \d+ /) → ["a01", "b23", "c45", "d56"]
"a01b23c45 d56".scan(/ \G [a-z] \d+ /) → ["a01", "b23", "c45"]
"a01b23c45 d56".scan(/ \A [a-z] \d+ /) → ["a01"]
```

## 替换

`#{...}` 像在字符串那样, 执行表达式替换。默认情况, 每次求解正则表达式时都执行该替换。如果设置了 `/o` 选项, 那么仅在第一次求解时执行替换。

`\0, \1, \2, ... \9, \&, \^, \', \+`

替换第 *n* 组子表达式匹配的值, 或者整个匹配、匹配之前、匹配之后, 或者最高的那一组。

## 正则表达式扩展

和 Perl、Python 一样, Ruby 的正则表达式也提供了传统 Unix 正则表达式之外的一些扩展。所有这些扩展都位于字符 “(?” 和 “)” 之间。括起这些扩展的括号是组, 但它们不生成后向引用 (back reference) : 它们不设置 `\1` 和 `$1` 等的值。

`(?# comment)`

插入注释到模式中。当模式匹配时, 其中的内容将被忽略。

`(?:re)` 使 *re* 成为组, 但不产生后向引用。当你想对结构进行分组, 但又不想为该组产生 `$1` 或者任何其他特殊变量时很有用。下面例子中的两个模式都能匹配用冒号或者空格分隔年月日的日期。第一种形式存储分隔字符在 `$2` 和 `$4` 中, 而第二个模式根本不在外部变量中存储这些分隔符。

```

date = "12/25/01"
date =~ %r{(\d+)(/|\:)(\d+)(/|\:)(\d+)}
[$1,$2,$3,$4,$5] → ["12", "/", "25", "/", "01"]
date =~ %r{(\d+)(?:/|\:)(\d+)(?:/|\:)(\d+)}
[$1,$2,$3] → ["12", "25", "01"]

```

(?*re*) 在此处匹配 *re*, 但并不耗用 (consume) 它 (也被称为零宽度正向查看——*zero-width positive lookahead*)。这可以让你向前寻找一个匹配的上下文而不影响\$&。在下面的例子中, `scan` 方法匹配后跟逗号的词, 但逗号不被包括在匹配的结果中。

```

str = "red, white, and blue"
str.scan(/([a-z]+(?:,))/) → ["red", "white"]

```

(?*!re*) 如果此处不匹配 *re*, 则认为匹配。且不消耗匹配 (零宽度正向查看)。例如/*hot* (?!*dog*) (*\w+*)/与含有字符 *hot* 且后不跟 *dog* 的词相匹配, 并保存词的结尾部分到\$1中。

(?>*re*) 在当前锚点的第一个正则表达式中嵌套一个独立的正则表达式。被其耗用的字符, 就不会被上层的正则表达式访问到。因此这个结构可以抑制回溯, 从而提高性能。例如, 模式/*a.\*b.\*a*/在匹配含有一个*a*后跟多个*b*但并不以*a*结尾的字符串时, 需要指数级时间复杂度。不过有时候这可以通过使用嵌套的正则表达式 /*a(?>.\*)b.\*a*/来避免。在这种形式中, 嵌套的表达式将耗用掉所有的输入字符直到最后一个可能的*b*。当检查尾部*a*匹配失败时, 就没必要回退, 并且模式匹配将迅速以失败告终 (当匹配不应当直接查到最后一个*b*时, 这个模式和原来的模式有不同的语义)。

```

require 'benchmark'
include Benchmark

str = "a" + ("b" * 5000)

bm(8) do |test|
 test.report("Normal:") { str =~ /^a.*b.*a/ }
 test.report("Nested:") { str =~ /^a(?>.*)b.*a/ }
end

```

输出结果:

	user	system	total	real
Normal:	0.460000	0.010000	0.470000	( 0.466996)
Nested:	0.000000	0.000000	0.000000	( 0.000435)

(?imx) 开启*i*, *m* 或者 *x* 选项。如果用在组中, 则只影响该组。

(?-imx) 关闭 i, m, 或者 x 选项。

(?imx:re) 为 re 开启选项 i, m, 或者 x。

(?-imx:re) 为 re 关闭选项 i, m, 或者 x。

## 22.3 名字

### Names

Ruby 名字用来引用常量、变量、方法、类和模块。名字中的第一个字符帮助 Ruby 来区分它的用法。某些名字，如下一页的表 22.3 所示，是保留字，不能将它们用作变量、方法、类或者模块的名字。

从第 345 页开始的章节里有方法名字的详细描述。

在下面的描述中，小写字母指从 a 到 z 的字母，以及下画线\_。大写字母指从 A 到 Z 的字母，而数字指 0 到 9 的数字字符。一个名字是一个大写字母、小写字母或者下画线，后跟任意个命名用字符：大写字母、小写字母、下画线或者数字的任意组合。

**局部变量名**由小写字母后跟命名用字符组成。通常来说，使用下画线而不是 camelCase<sup>3</sup>来书写含多个词的名字，不过对此解释器并不强制要求。

```
fred anObject _x three_two_one
```

**实例变量名**以一个“at”符 (@) 开始，后跟一个名字。在 @ 后用小写字母是个不错的做法。

```
@name @_ @size
```

**类变量名**以两个“at”符 @@ 开始，后跟一个名字。

```
@@name @@_ @@Size
```

**常量名**以一个大写字母开始，后跟多个命名用字符。类名字和模块名字都是常量，所以遵守常量的命名传统。习惯上，常量对象引用一般由大写字母和下画线组成，而类和模块名字是 MixedCase。

```
module Math
 ALMOST_PI = 22.0/7.0
end
class BigBlob
end
```

**全局变量**，以及一些特殊的系统变量，由美元符 (\$) 后跟命名用字符组成。此外，Ruby 还定义了一组由两个字母组成的全局变量名，其中第二个字符是一个标点符号。

---

<sup>3</sup> 译注：除第一个词外，其他词的首字母大写，像驼峰起伏，Java 采用的就是这种命名规范。

表 22.3 保留字

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

从第 333 页开始有这些预定义变量的详细列表。最后，全局变量名也可以由 \$-后跟一个字符或者下画线组成。后面的这些变量通常用来反映命令行选项的相应设置（详见第 335 页的表格）。

\$params    \$PROGRAM        \$!        \$\_        \$-a        \$-K

### 22.3.1 变量/方法二义性

#### Variable/Method Ambiguity

在表达式中，当 Ruby 看到像 a 这样的名字时，它需要判断 a 是一个局部变量引用还是对没有参数的方法 a 的调用。Ruby 使用一种启发式的方法来判断这种情况。当 Ruby 解析源代码文件时，它会记录所有已经被赋值的符号。它认为这些符号是变量。以后当遇到一个既可以是变量又可以是方法调用的符号时，Ruby 会检查是否已经对该符号进行了赋值。如果是，那么把该符号当作变量，否则当作方法调用。下面是描述这种情况的一个人为设计的例子，该例子由 Clemens Hintze 提供。

```
def a
 print "Function 'a' called\n"
 99
end
for i in 1..2
 if i == 2
 print "a=", a, "\n"
 else
 a = 1
 print "a=", a, "\n"
 end
end
```

输出结果：

```
a=1
Function 'a' called
a=99
```

当解析的时候，Ruby 看到第一个 print 语句使用了 a，并且由于还没有遇到对 a 的任意赋值语句，所以把它当作方法调用。但是当解析到第二个 print 语句时，由于 Ruby 遇到了对 a 的一个赋值语句，所以把它当作变量。

注意赋值语句不一定被执行——只要 Ruby 看到它了就可以。下面的程序不会导致错误。

```
a = 1 if false; a
```

## 22.4 变量和常量

### Variables and Constants

Ruby 的变量和常量含有对象的引用。变量本身没有内在的类型。变量的类型仅仅由变量引用的对象所能响应的消息决定。<sup>4</sup>

Ruby 常量也是对对象的引用。常量在它第一次被赋值的时候创建（通常是在类或模块的定义中）。和不灵活的语言不同，Ruby 允许你改变常量的值，尽管这会导致警告。

```
MY_CONST = 1
MY_CONST = 2 # generates a warning
```

输出结果：

```
prog.rb:2: warning: already initialized constant MY_CONST
```

注意尽管应当不改变常量的值，但是可以改变它所引用的对象的内部状态。

```
MY_CONST = "Tim"
MY_CONST[0] = "J" # alter string referenced by constant
MY_CONST → "Jim"
```

赋值会潜在地为对象定义别名，使得一个对象有不同的名字。

#### 22.4.1 常量和变量的作用域

##### Scope of Constants and Variables

在类或者模块内的任意位置都可以直接访问此类或模块中定义的常量。在类或者模块之外，可以通过在域作用符`::`前面加上一个返回适当类或者模块对象的表达式来访问。不属于任意类或者模块的常量，可以直接访问或者使用不带前缀的域作用符来访问。  
1.8 常量可以在方法体外定义。通过在常量名之前使用类或模块名和域作用符，还可以将常量从外面添加到已存在的类或者模块中。

```
OUTER_CONST = 99
```

---

<sup>4</sup> 当我们说变量没有类型的时候，是指任意一个变量都能在不同时候引用不同类型的对象。

```

class Const
 def get_const
 CONST
 end
 CONST = OUTER_CONST + 1
end

Const.new.get_const → 100
Const::CONST → 100
::OUTER_CONST → 99
Const::NEW_CONST = 123

```

**全局变量**是贯穿整个程序的变量。每次对某个特定全局名字的引用总是返回同一个对象。引用一个未被初始化的全局变量将会返回 `nil`。

**类变量**是贯穿类或者模块体的变量。它必须在使用之前初始化。一个类变量被类的所有实例共享，且只在类中可以使用。

```

class Song
 @@count = 0
 def initialize
 @@count += 1
 end
 def Song.get_count
 @@count
 end
end

```

类变量属于包含该变量的最内层的类或者模块。在顶层使用的类变量在 `Object` 内中定义，这种类变量类似于全局变量。在 `singleton` 方法中定义的类变量属于顶层类（尽管这种使用方式已经过时了，如果你用这种方式，会产生警告）。在 Ruby 1.9 中，类变量将是其类的私有变量。

```

class Holder
 @@var = 99
 def Holder.var=(val)
 @@var = val
 end
 def var
 @@var
 end
end
@@var = "top level variable"
a = Holder.new
a.var → 99
Holder.var = 123
a.var → 123

```

```
This references the top-level object
def a.get_var
 @@var
end
a.get_var → "top level variable"
```

类变量被定义该变量的类的所有子类所共享。

```
class Top
 @@A = 1
 def dump
 puts values
 end
 def values
 "#{self.class.name}: A = #{@A}"
 end
end
class MiddleOne < Top
 @@B = 2
 def values
 super + ", B = #{@B}"
 end
end
class MiddleTwo < Top
 @@B = 3
 def values
 super + ", B = #{@B}"
 end
end
class BottomOne < MiddleOne; end
class BottomTwo < MiddleTwo; end

Top.new.dump
MiddleOne.new.dump
MiddleTwo.new.dump
BottomOne.new.dump
BottomTwo.new.dump
```

输出结果：

```
Top: A = 1
MiddleOne: A = 1, B = 2
MiddleTwo: A = 1, B = 3
BottomOne: A = 1, B = 2
BottomTwo: A = 1, B = 3
```

**实例变量**可以被类内的所有实例方法使用。引用一个未初始化的实例变量会返回 nil。类的每个实例都有一组独特的实例变量。类方法中不能使用实例变量（尽管类和模块也可以有实例变量——请参考第 388 页）。

**局部变量**的独特之处在于它们的作用域是静态确定的，然而却是动态创建的。

局部变量是在程序执行时为其第一次赋值的时候动态创建的。然而，局部变量的作用域却是静态确定的，其作用域是最内层的 block、方法定义、类定义、模块定义或者顶层程序。引用一个在作用域内但未创建的局部变量会产生 `NameError` 异常。如果同名的局部变量不在同一个作用域内，那么它们是不同的变量。

**方法参数**是局部于其方法的变量。

**Block 参数**在执行 block 时被赋值。

```
a = [1, 2, 3]
a.each {|i| puts i } # i local to block
a.each {|$i| puts $i } # assigns to global $i
a.each {@i| puts @i } # assigns to instance variable @i
a.each {I| puts I } # generates warning assigning to constant
a.each {|b.meth| } # invokes meth= in object b
sum = 0
var = nil
a.each {|var| sum += var } # uses sum and var from enclosing scope
```

如果局部变量（包括 block 参数）是在 block 内第一次被赋值的，那么它是该 block 的局部变量。相反，如果在执行 block 时同名的变量已经被创建，那么 block 会继承该变量。

Block 可以使用在它创建时已经存在的局部变量。这形成绑定的一部分。注意，尽管此时变量的绑定是固定的，但是在执行的时候，block 可以访问这些变量的当前值。即使原本外围的作用域被销毁了，绑定依然会保持这些变量。

`while`, `until` 和 `for` 的循环体和循环体外的代码属于同一作用域：前面已经存在的局部变量可以在循环中使用，而且新创建的任意局部变量也可以在循环体后继续使用。

## 22.4.2 预定义变量

### Predefined Variables

下面的变量是 Ruby 解释器预定义的变量。在以下描述中，符号[r/o]表示变量是只读的：如果程序试图修改只读变量将会出现错误。而且，你可能不想在执行程序的半途中改变 `true` 的意思（除非你是政客）。带有[thread]标记的变量是局部于线程的变量。

许多全局变量看起来有点怪：例如`$_`, `$!`, `$&`等等。这是有“历史”原因的：这类变量大多数来自于 Perl。如果你觉得很难记住这些标点，你可以看看名为 `English` 的库文件，该库为常用的全局变量提供了更具描述性的名字，第 671 页有其详细文档。

在下面的变量和常量表中，我们列出了变量名字，所引用的对象类型以及一些描述。

### 异常信息

\$!	异常	传递给 <code>raise</code> 的异常对象。[thread]
\$@	数组	最后一个异常产生时的调用栈。详见第 518 页的 <code>Kernel#caller</code> 。[thread]

### 模式匹配变量

这些变量（除了`$=`）在模式匹配失败后被置为 `nil`。

\$&	String	匹配的字符串（匹配成功之后）。此变量局部于当前作用域。 [r/o, thread]
\$+	String	成功模式匹配产生的最高序号的组的内容。这样，在 "cat" =~ /(c a)(t z)/ 中，\$+ 为 "t"。这个变量局部于当前作用域。
\$`	String	匹配字符串之前的串。此变量局部于当前作用域。[r/o, thread]
\$`	String	匹配字符串之后的串。此变量局部于当前作用域。[r/o, thread]
<u>1.8.</u>	<u>\$=</u> Object	过时的用法。如果设置为除了 <code>nil</code> 或 <code>false</code> 之外的任意值，所有的模式匹配将会大小写无关，字符串比较将忽略大小写，而且字符串的 <code>hash</code> 值也会忽略大小写。
<u>\$1 到 \$9</u>	String	模式匹配中连续匹配的组的内容。在 "cat" =~ /(c a)(t z)/ 匹配后，\$1 为 "a"，\$2 为 "t"。这些变量局部于当前作用域。[r/o, thread]
\$~	MatchData	一个封装成功匹配结果的对象。变量 \$&, \$`, \$` 和 \$1 到 \$9 都是从 \$~ 中派生的。对 \$~ 的赋值会改变这些派生的变量的值。此变量局部于当前作用域。[thread]

### 输入/输出变量

\$/	String	输入记录分隔符（默认为换行符）。像 <code>Kernel#gets</code> 这样的例程用该值来区分记录边界。如果设置为 <code>nil</code> ， <code>gets</code> 将读入整个文件。
\$-0	String	\$/ 的同义词。
\$\	String	附加到方法调用（例如 <code>Kernel#print</code> 和 <code>IO#write</code> ）的输出结果的字符串。默认值为 <code>nil</code> 。
\$,	String	像 <code>Kernel#print</code> , <code>Array#join</code> 这样的方法的参数输出时的分隔字符串。默认为 <code>nil</code> ，不添加任何文本。
\$.	Fixnum	从当前输入文件中读入的最后一行的行号。
\$;	String	<code>String#split</code> 使用的默认分隔模式。也可以在命令行用 <code>-F</code> 选项设置。

\$<	<b>Object</b>	一个可以访问作为命令行参数给出或者 \$stdin (当没有参数的时候) 给出的所有文件的内容对象。\$<支持的方法和 File 对象类似: binmod,close,closed?,each,each_byte,each_line,eof,eof?,file,filename,fileno,getc,gets,lineno,lineno=,path, pos, pos=, read, readchar, readline, readlines, rewind,seek, skip, tell, to_a,to_i, to_io, to_s 以及 Enumerable 中的方法。file 方法返回表示当前正在读的文件的 File 对象。返回的对象可能会改变, 因为 \$<会依次读取命令行上给出的文件。[r/o]
\$>	<b>IO</b>	Kernel#print 和 Kernel#printf 的输出目标。默认值为 \$stdout。
\$_	<b>String</b>	Kernel#gets 或者 Kernel#readline 读入的最后一行。在 Kernel 模块中很多字符串相关的函数默认情况下都对 \$_ 操作。此变量局部于当前作用域。[thread]
<u>1.8</u>	<b>\$defout IO</b>	和 \$> 同义。已被弃用: 使用 \$stdout 替代。
<u>1.8</u>	<b>\$deferr IO</b>	和 STDERR 同义。已被弃用: 使用 \$stderr 替代。
	<b>\$-F String</b>	和 \$; 同义。
	<b>\$stderr IO</b>	当前标准错误输出。
	<b>\$stdin IO</b>	当前标准输入。
<u>1.8</u>	<b>\$stdout IO</b>	当前标准输出。对 \$stdout 赋值的用法已经过时, 使用 \$stdout.reopen 替代。

## 执行环境变量

\$0	<b>String</b>	被执行的 Ruby 程序的名字。通常是程序的文件名。在某些操作系统中, 对此变量赋值将会改变 ps(1) 命令报告的进程名字。
\$*	<b>Array</b>	一个包含调用程序时给出的命令行选项的字符串数组。被 Ruby 解释器使用的选项已经被删除。[r/o]
\$"	<b>Array</b>	一个包含 require 装载的文件名或者模块名的数组。[r/o]
\$\$	<b>Fixnum</b>	被执行的程序的进程号。[r/o]
\$?	<b>Process::Status</b>	最后一个终止的子进程的退出状态。[r/o, thread]
\$:	<b>Array</b>	一个字符串数组, load 和 require 方法将在每个字符串指定的目录中搜索 Ruby 脚本和二进制扩展。初始值为 -I 命令行选项传入的参数, 后跟安装时定义的标准库的路径、当前路径 (“.”)。在程序中可以重新设置此变量的值以调整默认搜索路径; 通常, 程序使用 \$: << dir 以将 dir 添加到路径中。[r/o]
\$-a	<b>Object</b>	如果命令行中给出了 -a 选项, 则为真。[r/o]

\$-d	Object	\$DEBUG 的同义词。	
\$DEBUG	Object	如果命令行中给出了 -d 选项，则为真。	
<u>_FILE_</u>	String	当前源文件的名字。[r/o]	
\$F	Array	如果使用了 -a 命令行选项，则此数组表示分隔的输入行。	
\$FILENAME	String	当前输入文件的名字。等价于 \$<.filename>。[r/o]	
\$-i	String	如果启动了 <i>in-place</i> 编辑模式（可能使用了 -i 命令行选项），则 \$-i 保存了用来保存备份文件的扩展。如果设置一个值给 \$-i，则将启用 <i>in-place</i> 编辑模式。详见第 178 页。	
\$-I	Array	与 \$: 同义。[r/o]	
\$-K	String	设置字符串和正则表达式的多字节编码系统。等价于 -K 命令行选项，详见第 179 页。	
\$-l	Object	如果命令行中给出了 -l 选项（该选项启用 <i>line-end</i> 处），则为真。详见第 179 页。[r/o]	
<u>_LINE_</u>	String	源代码文件中当前行的行号。[r/o]	
\$LOAD_PATH	Array	与 \$: 同义。[r/o]	
\$-p	Object	如果命令行中有 -p 选项（该选项将程序置于一个隐含的 while gets...end 循环中），则为真。详见第 179 页。[r/o]	
\$SAFE	Fixnum	当前的安全级别（详见第 398 页）。不能用赋值语句来减小该变量的值。[thread]	
1.8	\$VERBOSE	Object	如果命令行中给出了 -v, --version, -w 或者 -w 选项，则为真。如果没有这些选项，或者有 -w1，则为假。如果有 -w0 选项则为 nil。将此选项设置为真将使得解释器和一些库例程输出更多额外信息。设置为 nil 将禁止所有警告（包括 Kernel.warn 的输出）。
\$-v	Object	与 \$VERBOSE 同义。	
\$-w	Object	与 \$VERBOSE 同义。	

### 标准对象

ARGF	Object	与 \$< 同义。
ARGV	Array	与 \$* 同义。
ENV	Object	一个类似于 hash 的包含程序环境变量的对象。类 Object::ENV 的实例实现了 Hash 的所有方法。该对象可以用来查询和设置环境变量的值，例如 ENV["PATH"] 和 ENV["term"] = "ansi"。
false	FalseClass	类 FalseClass 的 Singleton 实例。[r/o]
nil	NilClass	类 NilClass 的 Singleton 实例。表示未初始化的实例变量和全局变量的值。[r/o]

<code>self</code>	<code>Object</code>	当前方法的接受者（对象）。[r/o]
<code>true</code>	<code>TrueClass</code>	<code>TrueClass</code> 类的 singleton 实例。[r/o]

### 22.4.3 全局常量

#### Global Constants

下面的常量是 Ruby 解释器定义的。

<code>DATA</code>	<code>IO</code>	如果主程序文件含有 <code>__END__</code> 指令，那么常量 <code>DATA</code> 将会被初始化为源代码中 <code>__END__</code> 之后的行。
<code>FALSE</code>	<code>FalseClass</code>	和 <code>false</code> 同义。
<code>NIL</code>	<code>NilClass</code>	和 <code>nil</code> 同义。
<code>RUBY_PLATFORM</code>	<code>String</code>	运行程序的平台标志符。这个字符串和 GNU <code>configure</code> 工具用的平台标志符有相同的形式（并不与之完全一致）。
<code>RUBY_RELEASE_DATE</code>	<code>String</code>	版本发布的日期。
<code>RUBY_VERSION</code>	<code>String</code>	解释器的版本号。
<code>STDERR</code>	<code>IO</code>	程序实际的标准错误流。初始值为 <code>\$stderr</code> 。
<code>STDIN</code>	<code>IO</code>	程序实际的标准输入流。初始值为 <code>\$stdin</code> 。
<code>STDOUT</code>	<code>IO</code>	程序实际的标准输出流。初始值为 <code>\$stdout</code> 。
<code>SCRIPT_LINES__</code>	<code>Hash</code>	如果常量 <code>SCRIPT_LINES__</code> 被定义，并引用一个 <code>Hash</code> ，那么 Ruby 会保存它解析的每个文件的内容为 <code>Hash</code> 的一项，键是文件的名字而值是一个字符串数组。请参考第 528 页的 <code>Kernel.require</code> 的实例。
<code>TOPLEVEL_BINDING</code>	<code>Binding</code>	一个 <code>Binding</code> 对象表示在 Ruby 顶层上的绑定——程序被初始执行的层次。
<code>TRUE</code>	<code>TrueClass</code>	和 <code>true</code> 同义。

常量`__FILE__`和变量`$0`常常被合用，以使得仅当文件直接由用户执行时才运行文件中的代码。例如，库程序员经常需要在库中加入一些测试代码，且当库代码被直接运行时才运行这些测试代码，而库被装载入其他程序时则不运行测试代码。

```
Library code
#
...
if __FILE__ == $0
 # tests...
end
```

## 22.5 表达式

### Expressions

表达式中用的术语可以是下面任意一个：

- **字面量。** Ruby 字面量可以是数字、字符串、数组、散列表（hash）、区间（range）、符号（symbol）和正则表达式。详见第 319 页。
- **Shell 命令。** Shell 命令是封装在反引号之间字符串或以%x 开头的常规分隔字符串（第 318 页）。字符串的值是在宿主操作系统的标准 shell 上运行字符串表示的命令标准输出。执行后也会将命令的退出状态保存在\$?变量中。

```
filter = "*.c"
files = `ls #{filter}`
files = %x{ls #{filter}}
```

- 1.8
- **符号产生器。** 符号对象是通过在操作符、字符串、变量、常量、方法、类或模块名字前加一个冒号生成的。对应不同名字的符号对象是唯一的，但是它并不指向名字的具体实例，所以不管上下文是什么，符号例如:fred 都是同一个。符号和其他高级语言的原子概念很相似。
  - **变量引用或常量引用。** 变量通过使用其名字来引用它。根据作用域的不同（参见第 330 页），常量可以通过使用名字来引用，也可以使用包含此常量的类或模块的名字加域操作符(::) 来引用。
- ```
barney      # 变量引用
APP_NAMR    # 常量引用
Math::PI     # 使用域操作符的常量引用
```
- **方法调用。** 第 348 页有方法调用的各式各样方式的描述。

22.5.1 操作符表达式

Operator Expressions

表达式可以用操作符来组合。下一页上的表 22.4 按优先级顺序列出了 Ruby 的操作符。在方法一栏有个✓的操作符是由方法来实现的，可以被覆写（override）。

22.5.2 关于赋值的更多内容

More on Assignment

赋值操作符将一个或者多个 *rvalues* (*r* 表示“右”，因为右值倾向于出现在赋值语句的右边) 赋给一个或者多个 *lvalues* (“左”值)。赋值的意义依赖于每个单独的左值。

表 22.4 Ruby 操作符 (优先级从高到低)

| 方 法 | 操作符 | 描 述 |
|-----|------------------------|----------------------------------|
| ✓ | [] []= | 引用元素、设置元素 |
| ✓ | ** | 求幂 |
| ✓ | ! ~ + - | 非、求补、一元加和一元减 (最后两个的方法名为+@和-@) |
| ✓ | * / % | 乘、除和求模 |
| ✓ | + - | 加和减 |
| ✓ | >> << | 右移和左移 |
| ✓ | & | “与” (对整数是一位一位的操作) |
| ✓ | ^ | 异或和或 (对整数是一位一位的操作) |
| ✓ | <= < > >= | 比较操作符 |
| ✓ | <=> == === != =~ !~ | 等于和模式匹配操作符 (!= 和 !~ 可能不是作为方法定义的) |
| | && | 逻辑 “与” |
| | | 逻辑 “或” |
| | | Range (包含边界和不包含边界) |
| | ? : | 三元运算符 if-then-else |
| | = %= /= -= += = &= | 赋值运算符 |
| | >>= <<= *= &&= = **= | |
| | defined? | 检查符号是否被定义 |
| | not | 逻辑非 |
| | or and | 逻辑比较 |
| | if unless while until | 表达式修饰符 |
| | begin/end | Block 表达式 |

如果左值是一个变量名或者常量名，那么该变量或者常量获得相应左值的引用。

```
a = /regexp/
b,c,d = 1, "cat", [3, 4, 5]
```

如果左值是一个对象属性，那么接收对象中对应的属性设置函数将被调用，并以右值作为参数。

```
obj = A.new;
obj.value = "hello" # 等价于 obj.value = ("hello")
```

如果左值是数组元素引用，那么 Ruby 将调用接收对象的元素赋值操作符 ([]=)，并以方括号的下标和右值为参数。下表演示了这一点。

| 元素引用 | 实际的方法调用 |
|---|---|
| <code>obj[] = "one"</code> | <code>obj.[]("one")</code> |
| <code>obj[1] = "two"</code> | <code>obj.[](1, "two")</code> |
| <code>obj["a", /<i>cat</i>/] = "three"</code> | <code>obj.[]("a", /<i>cat</i>/, "three")</code> |

1.8 赋值表达式本身的值是它的右值。即使是对返回不同值的属性方法赋值，也是如此。

并行赋值

赋值表达式可以有一个或者多个左值，也可以有一个或者多个右值。本节将阐述 Ruby 是如何处理具有不同参数组合的赋值的。

1. 如果最后一个右值前面有一个星号，且实现了 `to_ary` 函数，那么该右值将被数组的元素取代，每个元素形成一个独立的右值。
2. 如果赋值语句含有多个左值和一个右值，那么右值将被转换成一个数组，而且这个数组会如（1）中所述那样扩展成一组右值。
3. 连续的右值赋值给左值。这种赋值其实是并行进行的，所以（例如）`a,b = b,a` 将交换 `a` 和 `b` 的值。
4. 如果左值个数多于右值个数，那么超出的部分被赋为 `nil`。
5. 如果右值个数多于左值个数，那么多余的部分将被忽略。
6. 如果最后一个左值前面有个星号，那么这些规则需要微调。在赋值语句中，这个左值总是赋值为一个数组。此数组将包含正常情况下赋值给该左值的右值，以及多余的右值（如果有的话）。
7. 如果左值含有括号括起来的列表，则此列表被当作嵌套赋值语句，而且应用这些规则来根据相应的右值进行赋值。

第 91 页有赋值语句的示例。

22.5.3 Block 表达式 Block Expressions

```
begin
  body
end
```

表达式可以成组地放入 `begin` 和 `end` 之间。Block 表达式的值是最后一个表达式的值。

Block 表达式在异常处理中也起作用，第 360 页会对此进行讨论。

22.5.4 布尔表达式

Boolean Expressions

Ruby 预定义了全局的 `false` 和 `nil`。在布尔表达式上下文中，这两个都被当作假。所有其他值都被当作真。当你需要使用显式的“真”值时，可以使用常量 `true`。

与、或、非和 Defined

`and` 和 `&&` 操作符先求解它们的第一个操作数，如果为假，那么表达式返回第一个操作符的值；否则，表达式返回第二个操作数的值。

```
expr1 and expr2
expr1 && expr2
```

`or` 和 `||` 操作符先求解第一个操作数的值，如果为真，则表达式返回它们的第一个操作数的值；否则，表达式将返回第二个操作数的值。

```
expr1 or expr2
expr1 || expr2
```

`not` 和 `!` 操作符求解它们操作数的值，如果为真，则表达式返回假；如果为假，则表达式返回真。由于历史的原因，字符串、正则表达式或者 `range` 不能作为 `not` 或者 `!` 的参数。

这些操作符的单词形式 (`and`, `or` 和 `not`) 比相应的符号形式 (`&&`, `||` 和 `!`) 优先级低。详见第 339 页的表 22.4。

`defined?` 操作符的参数可以是任意表达式，如果这个参数未定义，则 `defined?` 返回 `nil`。否则，它将返回参数的一个描述。请参见本教程第 94 页上的例子。

比较操作符

Ruby 的语法定义了比较操作符 `==`, `==>`, `<=`, `<`, `<=`, `>`, `>=`, `~=`。所有这些操作符都是由方法来实现的。习惯上，Ruby 语言也使用标准方法 `eq?` 和 `equal?`（请参见第 95 页的表 7.1）。尽管这些操作符的意义很直观，但是其比较语义是由实现它们的类来决定的。从第 423 页开始的库文档阐述了内置类的比较语义。依据 `<=`，模块 `Comparable` 为实现操作符 `==`, `<`, `<=`, `>`, `>=` 和方法 `between?` 提供了支持。操作符 `==>` 被用在 `case` 表达式中，详见第 343 页。

`==` 和 `~=` 都有相反的形式 `!=` 和 `!~`。在语法分析中，Ruby 会对它们进行转换：`a!=b` 映射为 `!(a==b)`，而 `a!~b` 映射为 `!(a =~ b)`。没有和 `!=` 与 `!~` 相对应的方法。

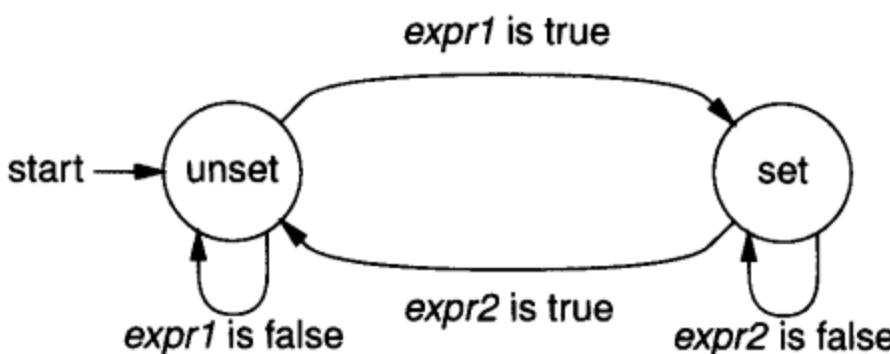


图 22.1 boolean 区间 (range) 的状态变迁

布尔表达式中的 Ranges

```
if expr1 .. expr2
while expr1 ... expr2
```

在布尔表达式中使用的 range 具有双态触发器 (flip-flop) 的行为。它有两种状态：开和关，而且初始状态是关。如图 22.1 中的状态机所示，每调用一次 range 变迁一次状态。如果当调用结束时状态机处于开状态，则 range 表达式返回 true；否则返回 false。

两点形式的 range 和三点形式的 range 行为稍有不同。两点的 range 当第一次从关变成开时，它会立即求解结束条件，并相应地变迁状态。这意味着如果 *expr1* 和 *expr2* 在一个调用中都为真，那么两点形式的调用将结束于关状态。不过，该调用仍旧返回真。

三点形式的 range 不会在进入开状态后立即求解结束条件。

下面的代码说明了这些差别。

```
a = (11..20).collect { |i| (i%4 == 0)..(i%3 == 0) ? i : nil}
a → [nil, 12, nil, nil, nil, 16, 17, 18, nil, 20]
a = (11..20).collect { |i| (i%4 == 0)...(i%3 == 0) ? i : nil}
a → [nil, 12, 13, 14, 15, 16, 17, 18, nil, 20]
```

布尔表达式中的正则表达式

- 1.8** 在 Ruby 1.8 之前，布尔表达式中的单个正则表达式将会和变量 \$_ 的当前值进行匹配。现只有在命令行 -e 参数中出现的条件中才支持这种行为。在通常的代码中，对隐含操作符和 \$_ 的使用已经慢慢废止了，所以最好使用匹配变量的显式匹配。如果需要匹配 \$_，则使用：

```
if ~ /re/ ... 或 if $_ =~ /re/ ...
```

22.5.5 if 和 unless 表达式

if and unless Expressions

```
if 布尔表达式 [ then | : ]
  代码体
[ elsif 布尔表达式 [ then | : ]
  代码体 , ... ]
[ else
  代码体 ]
end

unless 布尔表达式 [ then | : ]
  代码体
[ else
  代码体 ]
end
```

关键字 `then`（或者冒号）将代码体和条件分隔开来。如果代码体和条件不在一行上，则不需要这些分隔符。无论哪个代码体被执行，`if` 或者 `unless` 表达式的值是最后一个被执行的表达式的值。

if 和 unless 修饰符

`表达式 if 布尔表达式`
`表达式 unless 布尔表达式`

仅当布尔表达式为真（对 `if`）或者假（对 `unless`）时才求解表达式。

22.5.6 三元运算符

Ternary Operator

`布尔表达式 ? expr1 : expr2`
 如果布尔表达式为真，则返回 `expr1`，否则返回 `expr2`。

22.5.7 case 表达式

case Expressions

Ruby 有两种形式的 `case` 语句。第一种会求解一系列条件的值，并执行第一个为真的条件对应的代码。

```
case
when 条件 [, 条件 ]... [ then | : ]
  代码体
when 条件 [, 条件 ]... [ then | : ]
  代码体
...
[ else
  代码体 ]
end
```

第二种形式的 `case` 表达式，在 `case` 关键字后面有一个目标表达式。它从第一个（左上）`comparison` 开始，通过执行 `comparison == target` 寻找一个匹配。

```

case target
when comparison [, comparison]... [ then |: ]
  body
when comparison [, comparison]... [ then |: ]
  body
...
[ else
  body ]
end

```

被比较目标可以是跟在星号后的一个数组引用，在这种情况下，在执行测试之前，数组会扩展成其元素。当比较返回真时，停止搜索，并执行与该比较目标相关联的代码体（不需要 `break`）。然后 `case` 返回执行的最后一个表达式的值。如果没有 `comparison` 匹配：如果有 `else` 从句，则执行其中的代码，否则将返回 `nil`。

`then` 关键字（或冒号）分隔 `when` 比较目标和它的代码体，如果代码体从一行开始，则不需要 `then` 和冒号。

循环

```

while 布尔表达式 [ do | : ]
  循环体
end

```

只要布尔表达式为真，则一直执行循环体。

```

until 布尔表达式 [ do | : ]
  循环体
end

```

只要表达式为假，则一直执行循环体。

上面的两种形式中，`do` 或冒号将布尔表达式和循环体分隔开，如果循环体在另一行上，则可以省略它们。

```

for name [, name ]... in expression [ do | : ]
  循环体
end

```

`for` 循环的执行很像下面的 `each` 循环，除了在 `for` 循环体中定义的局部变量在循环体外仍旧可以使用，而在迭代 `block` 中定义变量在循环体外无效。

```

expression.each do |name [, name ]... |
  循环体
end

```

Library `loop`，迭代执行与之相关联的 `block`，不是一个语言结构——它是 `Kernel` 模块中定义的一个方法。

```

loop do
  print "Input: "
  break unless line = gets
  process(line)
end

```

while 和 until 修饰符

表达式 while 布尔表达式

表达式 until 布尔表达式

如果表达式不是 begin/end block，则只要布尔表达式为真（对 while）或假（对 until）就一直执行表达式。

如果表达式是 begin/end block，则 block 至少会被执行一次。

break, redo, next 和 retry

break, redo, next 和 retry 会改变 while, until, for 或者迭代器控制的循环正常执行流程。

break 终止直接封装它的循环——控制会转到 block 后面的语句。redo 从头重新执行循环，但并不重新求解条件表达式或取迭代器的下一个元素。next 关键字跳到循环体的末尾，相当于开始下一次迭代。retry 重新执行循环，并重新求解条件表达式。

1.8 break 和 next 可以有一个或多个可选的参数。如果 break 和 next 用在 block 中，则给定的参数作为 yield 的值返回。如果用在 while, until 和 for 循环中，传递给 break 的值将作为本语句的值返回，而传递给 next 的值将被忽略。如果没有调用 break，或者调用了它但没有参数值，则循环返回 nil。

```

match = while line = gets
  next if line =~ /^#/
  break line if line =~ /ruby/
end

match = for line in ARGF.readlines
  next if line =~ /^#/
  break line if line =~ /ruby/
end

```

22.6 方法定义

Method Definition

```

def defname [ ([ arg [ =val ] ,... ] [ , *vararg ] [ , &blockarg ] ) ]
  body
end

```

defname 既是方法的名字，也是其所处的合法的上下文的名字。

```
defname → methodname
constant.methodname
(expr).methodname
```

methodname 既可以是可重定义的操作符（参见第 339 页的表 22.4），也可以是个名字。如果 *methodname* 是个名字，那么它应当以一个小写字母（或者下画线）开头，后面可以跟大写和小写字母、下画线和数字。*Methodname* 还可以以问号（?）、感叹号（!）或者等于号（=）结尾。问号和感叹号仅仅是名字的一部分。等于号也是名字的一部分，但也表明了该方法可以用作左值（参见第 31 页）。

在类或者模块定义中，由普通方法名定义的方法是实例方法。一个实例方法仅能通过发送他的名字给一个接受者来调用，且该接受者必须是定义它的类（或者其子类）的一个实例。

在类或者模块定义外，由普通方法名定义的方法将作为私有方法加入到类 `Object` 中，因此无须指定显式的接受者就可以在任意上下文中调用它。

使用 `constant.methodname` 形式或者更一般的`(expr).methodname` 形式作为方法名，将创建一个与常量或者表达式引用的对象相关联的方法；该方法只能通过以表达式引用的对象作为接受者来调用。这种定义风格将创建单个对象的方法或称单例（*singleton*）方法。

```
class MyClass
  def MyClass.method      # 定义
  end
end

MyClass.method          # 调用

obj = Object.new
def obj.method          # 定义
end

obj.method              # 定义

def (1.class).fred     # 接受者可以是表达式
end

Fixnum.fred            # 调用
```

- 1.8 方法定义可能不包含类或模块定义。它们可以含有嵌套的实例或者单例方法定义。当执行外部的方法时，内部的方法被定义。在被嵌套方法的上下文中，内部方法不是一个闭包（closure）——它是自包含的。

```

def toggle
  def toggle
    "subsequent times"
  end
  "first time"
end

toggle → "first time"
toggle → "subsequent times"
toggle → "subsequent times"

```

方法体的行为如同是一个 begin/end 的 block，其中可能包括异常处理的语句（rescue、else 和 ensure）。

22.6.1 方法参数

Method Arguments

方法定义可以含有零或多个普通参数，一个可选的数组参数和可选的 block 参数。参数之间用逗号分隔，而且参数列表可以用括号括起来。

普通参数是一个局部变量，后可跟一个等于号和作为默认值的表达式。表达式在方法被调用的时候被求解。表达式是从左向右求解的。表达式可以引用参数列表中它前面已定义的参数（parameter，形参）。

```

def options(a=99, b=a+1)
  [ a, b ]
end
options → [99, 100]
options 1 → [1, 2]
options 2, 4 → [2, 4]

```

可选的数组参数必须在所有的普通参数后面，而且可能没有默认值。当方法被调用时，Ruby 设置数组参数使其引用一个数组类的新对象。如果调用方法时给定的参数超过了普通参数的个数，那么所有额外的参数将被放入这个新创建的数组中。

```

def varargs(a, *b)
  [ a, b ]
end
varargs 1 → [1, []]
varargs 1, 2 → [1, [2]]
varargs 1, 2, 3 → [1, [2, 3]]

```

如果数组参数跟在带有默认值的参数后面，那么参数将优先被用来覆盖（override）默认值。剩下的参数将会用来形成数组。

```

def mixed(a, b=99, *c)
  [a, b, c]
end
mixed 1      →      [1, 99, []]
mixed 1, 2    →      [1, 2, []]
mixed 1, 2, 3 →      [1, 2, [3]]
mixed 1, 2, 3, 4 →     [1, 2, [3, 4]]

```

可选的 `block` 参数必须是参数列表中的最后一个。无论何时调用该方法，Ruby 都会检查是否有关联的 `block`。如果有 `block`，它将被转换成 `Proc` 类的对象，然后赋值给 `block` 参数。如果没有 `block`，该参数将被设置为 `nil`。

```

def example(&block)
  puts block.inspect
end

example
example { "a block" }

```

输出结果：

```

nil
#<Proc:0x001c9940@->:6>

```

22.7 调用方法

Invoking a Method

```

[receiver.]name[parameters][block]
[receiver::]name[parameters][block]

parameters ← ([param, ...][, hashlist][*array][&a_proc])

block      ← {blockbody}
            do blockbody end

```

开头的形参将被赋值给方法的实际参数。这些形参后面可以有一个 `key=>value` 配对的列表。这些 `key=>value` 对将被收集到一个新的 Hash 对象中，并作为一个形参传入方法。

这些形参后面可以是一个前面带星号的参数。如果这个形参是一个数组，那么 Ruby 将用相应的数组元素，将它替换为零或多个形参。

```

def regular(a, b, *c)
  # ..
end
regular 1, 2, 3, 4
regular(1, 2, 3, 4)
regular(1, *[2, 3, 4])

```

`block` 可以关联一个方法调用，该调用可以使用字面量形式的 `block`（它必须和方法调用的最后一行在同一行上），也可以关联一个参数，该参数包含对带`&`符号的 `Proc` 或 `Method` 对象的引用。不管 `block` 参数存在与否，Ruby 使用全局函数 `Kernel.block_given?` 的值来判断是否存在与本调用相关联的 `block`。

```
a_proc      = lambda { 99 }
an_array    = [ 98, 97, 96 ]
def block
  yield
end
block { }
block do
  end
block(&a_proc)

def all(a, b, c, *d, &e)
  puts "a = #{a.inspect}"
  puts "b = #{b.inspect}"
  puts "c = #{c.inspect}"
  puts "d = #{d.inspect}"
  puts "block = #{yield(e).inspect}"
end

all('test', 1 => 'cat', 2 => 'dog', *an_array, &a_proc)
```

输出结果：

```
a = "test"
b = {1=>"cat", 2=>"dog"}
c = 98
d = [97, 96]
block = 99
```

可以通过将方法的名字传递给接受者来调用一个方法。如果没有指明接受者，则使用 `self`。接受者首先在它自己的类中查看是否有方法定义，然后依次在它的父类中查找。被包含的模块的实例方法看起来像是包含它们的类的匿名超类。如果没找到方法，Ruby 将调用接受者的 `method_missing` 方法。`Kernel.method_missing` 定义的默认行为是报告错误并终止程序。

当显式指明方法调用的接受者时，可以使用句点“.”或两个冒号“::”把它和方法名分隔开。这两种形式的唯一区别仅在方法名以大写字母开始时才会出现。在这种情况下，Ruby 会认为 `receiver::Thing` 方法调用是试图访问接受者中的称为 `Thing` 的常量，除非方法调用带有一个包含在括号中的参数列表。

```

Foo.Bar()          # 方法调用
Foo.Bar           # 方法调用
Foo::Bar()        # 方法调用
Foo::Bar          # 访问常量

```

方法的返回值是执行的最后一个表达式的值。

```
return [expr, ...]
```

`return` 表达式会立刻退出方法。如果不带参数调用 `return`，则 `return` 的返回值为 `nil`，如果带一个参数，则返回该参数的值，如果参数多于一个，则返回一个包含所有参数值的数组。

22.7.1 super

```
super [([param, ...] [*array])] [block]
```

在方法体内，调用 `super` 就像调用原方法一样，不过是在含有原方法的对象的超类中搜索方法体。如果没有参数（且未加括号）传递给 `super`，则原方法的参数将作为 `super` 的参数，否则，`super` 的参数将被传递。

22.7.2 操作符方法

Operator Methods

```

expr1 operator
operator expr1
expr1 operator expr2

```

如果操作符表达式中的操作符是一个可重新定义的方法（参见第 339 页的表 22.4），Ruby 像调用如下表达式那样执行操作符表达式。

```
(expr1).operator() 或
(expr1).operator(expr2)
```

22.7.3 属性赋值

Attribute Assignment

```
receiver.attrname = rvalue
```

- 当 `receiver.attrname` 作为左值出现时，Ruby 将调用接受者中名为 `attrname=` 的方法，并以右值作为其唯一参数。这种赋值语句的返回值总是 `rvalue` 的值——`attrname=` 的返回值将被丢弃。如果你想访问返回值（多半情况下不是 `rvalue` 的值），那么向方法发送一个显式的消息。

```

class Demo
  attr_reader :attr
  def attr=(val)
    @attr = val
    "return value"
  end
end

d = Demo.new

# 在如下情形中, @attr 都被设置为 99
d.attr = 99          →      99
d.attr=(99)         →      99
d.send(:attr=, 99)   →      "return value"
d.attr             →      99

```

22.7.4 元素引用操作符

Element Reference Operator

receiver[expr[, expr]...]
receiver[expr[, expr]...] = rvalue

当用作右值时, 元素引用调用接受者的`[]`方法, 并以方括号中的表达式作为参数传递。

当用作左值时, 元素引用将调用接受者的`[]=`方法, 并以方括号中的表达式和跟在其后被赋值的 *rvalue* 作为参数传递。

22.8 别名

Aliasing

`alias new_name old_name`

将创建一个引用已有的方法、操作符、全局变量或正则表达式向后引用 (`$&`, `$``, `$'` 和 `$+`) 的新名字。局部变量、实例变量、类变量和常量不能有别名。`alias` 的参数可以是名字或符号。

```

class Fixnum
  alias plus +
end
1.plus(3)       →      4

alias $prematch $`
$string" =~ /i/   →      3
$prematch        →      "str"

alias :cmd :
cmd "date"       →      "Thu Aug 26 22:37:16 CDT 2004\n"

```

当为方法起别名时，新的名字将指向原方法体的一个拷贝。即使后来方法被重新定义了，别名仍旧会调用原来的方法实现代码。

```
def meth
  "original method"
end
alias original meth
def meth
  "new and improved"
end
meth      →      "new and improved"
original →      "original method"
```

22.9 类定义

Class Definition

```
class [scope::] classname [< superexpr>]
  body
end

class << obj
  body
end
```

Ruby 的类定义通过执行类代码体创建或者扩展类 Class 的对象。在第一种形式中，一个命名类将被创建或者被扩展。生成的 Class 对象将被赋给名为 *classname* 的常量（参见下面的作用域规则）。这个名字应该以一个大写字母打头。在第二种形式中，一个匿名（单例）类会和指定的对象相关联。

如果 *superexpr* 存在，那么它应当是一个以 Class 对象为结果的表达式，而且它也将是被定义的类的超类。如果省略了 *superexpr*，则默认为类 Object。

在方法体内，随着各种定义代码的读入，大多数 Ruby 表达式将被执行。然而：

- 方法定义将在类的对象的一个表中注册该方法。
- 嵌套的类和模块定义将被存储在类的常量中，而不是全局常量中。在定义嵌套的类和模块的类外可以通过使用 “`::`” 修饰其名字来访问它们。

```
module NameSpace
  class Example
    CONST = 123
  end
end
obj = NameSpace::Example.new
a = NameSpace::Example::CONST
```

- 方法 `Module#include` 将把命名的模块添作被定义的类的匿名超类。

使用域作用符 (::) 可以为类定义中的 *classname* 前置一个已存在的类或模块名。

- 1.8 这种语法会将新的定义插入到前面已定义的模块和/或类的名字空间中，但不是在这些外部类的作用域中解释此定义。前面带有域作用符的 *classname* 将使其类或模块处于顶层作用域中。

在下面的例子中，类 C 被插入到模块 A 的作用域，但并不在 A 的上下文中进行解释。结果，对 CONST 的引用被解释成该名字对应的顶层常量而不是 A 的常量。而且我们必须使用单例方法的全名，因为在 A::C 的上下文中，c 本身不是一个已知的常量。

```
CONST = "outer"

module A
  CONST = "inner" # This is A::CONST
end

module A
  class B
    def B.get_const
      CONST
    end
  end
end

A::B.get_const → "inner"

class A::C
  def (A::C).get_const
    CONST
  end
end

A::C.get_const → "outer"
```

值得我们强调的是类定义是可执行的代码。在类定义中使用的许多指令（例如 attr 和 include）只不过是类 Module 的私有实例方法（参见第 554 页的文档）。

从第 379 页开始的第 24 章描述了 Class 对象是如何与环境的其他部分进行交互细节的。

22.9.1 从类中创建对象

Creating Objects from Classes

obj = *classexpr*.new [([*args*, ...])]

- 类 Class 定义的实例方法 Class#new 将创建接受者（在这个语法例子中是 *classexpr*）对应的类的对象。这是通过调用 *classexpr*.allocate 来完成的。你可以重载此方法，但是你的实现必须返回正确的类的对象。然后它调用新创建的对象的 initialize，并将传递给 new 的参数传递给 initialize。

如果类定义中重载了类方法 `new`，并且 `new` 没有调用 `super`，那么将无法创建该类的对象，并且调用 `new` 将返回 `nil`。

和其他方法一样，`initialize` 应当调用 `super`，以保证父类被适当的初始化。如果父类是 `Object`，则不需要这么做，因为 `Object` 类不做任何实例相关的初始化。

22.9.2 类属性声明

Class Attribute Declarations

Library

类属性声明不是 Ruby 语法的一部分：它们不过是定义在类 `Module` 中的方法，该方法会自动创建访问类属性的方法。

```
class name
  attr attribute [, writable]
  attr_reader attribute [, attribute]...
  attr_writer attribute [, attribute]...
  attr_accessor attribute [, attribute]...
end
```

22.10 模块定义

Module Definitions

```
module name
  body
end
```

模块基本上是一个不能实例化的类。和类一样，在定义过程中模块体将被执行，生成的 `Module` 对象被存储在一个常量中。模块中可以含有类方法和实例方法，也可以定义常量和类变量。和类一样，通过使用 `Module` 对象作为接受者调用模块方法，通过使用“`::`”域作用符来访问常量。在模块定义中的名字可以以封装它的类或者模块做前缀。

```
CONST = "outer"
module Mod
  CONST = 1
  def Mod.method1    # module method
    CONST + 1
  end
end
module Mod::Inner
  def (Mod::Inner).method2
    CONST + " scope"
  end
end
Mod::CONST          →      1
Mod.method1         →      2
Mod::Inner::method2 →      "outer scope"
```

22.10.1 Mixins——混入模块

Mixins—Including Modules

```
class | module name
  include expr
end
```

使用 `include` 方法可以将一个模块包含到另一个模块或者类的定义中。含有 *Library* `include` 的模块或类定义可以访问它所包含模块的常量、类变量和实例方法。

如果一个模块被包括到一个类定义中，那么模块的常量、类变量和实例方法实际上被绑定到该类的一个匿名（且不可访问）超类中。类的对象会响应发送给模块实例方法的消息。如果调用未在类中定义的方法，那么在传递此方法给任何父类之前会先传递给该类包括的模块。模块可以定义一个 `initialize` 方法，当创建包括模块的类的对象时，如果满足下面条件之一，那么该方法将被调用：(a) 类没有定义它自己的 `initialize` 方法，或 (b) 类的 `initialize` 方法调用了 `super`。

模块也可以在顶层被包括，在这种情况下，顶层可以访问模块的常量、类变量和实例方法。

模块函数

尽管 `include` 在提供 mixin 功能时很有用，它也可以把模块的常量、类变量和实例方法带入到另一个名字空间中。然而，实例方法定义的功能不能通过模块方法来实现。

```
module Math
  def sin(x)
    #
  end
end

# Only way to access Math.sin is...
include Math
sin(1)
```

方法 `Module#module_function` 通过拷贝一个或多个模块实例方法的定义来创建相 *Library* 应的模块方法来解决这个问题。

```
module Math
  def sin(x)
    #
  end
  module_function :sin
end

Math.sin(1)
include Math
sin(1)
```

实例方法和模块方法是两个不同的方法：方法定义被 `module_function` 拷贝出来而不是建立别名。

22.11 访问控制

Access Control

Ruby 为模块和类的常量和方法定义了 3 种保护级别。

- **Public**。任意访问。
- **Protected**。仅可以被类或者子类的对象访问。
- **Private**。仅可以使用函数形式访问（也就是说以隐含的 `self` 作为接受者）。因此私有方法仅能在本类中和同一个对象的直接后代中被访问。参见第 37 页的讨论。

```
private [symbol, ...]
protected [symbol, ...]
public [symbol, ...]
```

Library

每个函数有两种使用方式。

1. 如果没有参数，那么这三个函数为随后定义的方法设置默认的访问控制。
2. 如果有参数，那么为参数中给出的方法或常量设置访问控制。

方法被调用时执行访问控制。

22.12 Blocks, Closures 和 Proc 对象

Blocks, Closures, and Proc Objects

代码 `block` 是包括在花括号或 `do/end` 对中的一组语句和表达式。Block 可以以包括在垂直条中的参数列表开始。它只能紧随在方法调用后才出现。Block 的开始部分（花括号或 `do`）必须和方法调用处于同一逻辑行上。

```
invocation do | a1, a2, ... |
end

invocation { | a1, a2, ... | }
```

花括号具有高优先级，而 `do` 的优先级较低。如果方法调用有不被括在括号内的参数，那么花括号形式的 `block` 将会绑定到最后一个参数，而不是整个调用上。`do` 形式会绑定到调用上。

在被调用的方法的方法体内，可以使用 `yield` 关键字调用代码 `block`。传递给 `yield` 的参数将赋值给 `block` 的参数。如果 `yield` 将多个参数传递给只能接受一个参数的 `block`，将会产生一个警告。`yield` 的返回值是 `block` 中执行的最后一条语句的值或者传递给在 `block` 中执行的 `next` 语句的值。

`block` 是 *closure*；它能记住其被定义时的上下文，并在被调用时使用该上下文。上下文中包含 `self` 的值、常量、类变量、局部变量和任意被截获的 `block`。

```
class Holder
  CONST = 100
  def call_block
    a = 101
    @a = 102
    @@a = 103
    yield
  end
end

class Creator
  CONST = 0
  def create_block
    a = 1
    @a = 2
    @@a = 3
    proc do
      puts "a = #{a}"
      puts "@a = #{@a}"
      puts "@@a = #{@@@a}"
      puts yield
    end
  end
end

block = Creator.new.create_block { "original" }
Holder.new.call_block(&block)
```

输出结果：

```
a = 1
@a = 2
@@a = 3
original
```

22.12.1 Proc 对象, break 和 next

Proc Objects, break, and next

Ruby 的 `block` 是和方法相关联的一堆代码，并在它们被定义的上下文中运作。`Block` 不是对象，但是它们能被转换成类 `Proc` 的对象。有 3 种方式将 `block` 转换成 `Proc` 对象。

- 通过传递 `block` 给一个方法，该方法的最后一个参数前有一个地址符`&`。该参数会接受 `block` 作为 `Proc` 对象。

```
def meth1(p1, p2, &block)
  puts block.inspect
end
meth1(1,2) { "a block" }
meth1(3,4)
```

输出结果：

```
#<Proc:0x001c9940@-:4>
nil
```

Library

- 通过调用 `Proc.new`，再将它和 `block` 关联。

```
block = Proc.new { "a block" }
block → #<Proc:0x001c9ae4@-:1>
```

Library

- 通过调用方法 `Kernel.lambda`（或者等价的有点过时的方法 `Kernel.proc`）关联 `block` 到方法调用。

```
block = lambda { "a block" }
block → #<Proc:0x001c9b0c@-:1>
```

前两种风格的 `Proc` 对象在使用时是相同的。我们称这些对象为 *raw procs*。第三种风格，由 `lambda` 生成，为 `Proc` 对象添加了一些额外的功能，我们很快就能看到。我们称这种对象为 *lambdas*。

无论在哪种 `block` 内，`next` 语句将退出 `block`。`block` 的值是传递给 `next` 的值，如果没有值传递给 `next`，则为 `nil`。

```
def meth
  res = yield
  "The block returns #{res}"
end

meth { next 99 } → "The block returns 99"
pr = Proc.new { next 99 }
pr.call → 99
pr = lambda { next 99 }
pr.call → 99
```

在 *raw proc* 中，`break` 语句可以终止调用 `block` 的方法。方法的返回值为传递给 `break` 的参数。

22.12.2 Return 和 Block Return and Blocks

处于作用域 `block` 内的 `return` 和该作用域的 `return` 作用一样。`block` 内原上下文不再

有效的 `return` 会引发异常（因上下文不同会引发 `LocalJumpError` 或 `ThreadError` 异常）。下面的例子演示了第一种情况。

```
def meth1
  (1..10).each do |val|
    return val  # 从方法中返回
  end
end
meth1      →      1
```

下面的例子展示了由于 `block` 的上下文不存在而导致返回失败。

```
def meth2(&b)
  b
end

res = meth2 { return }
res.call
```

输出结果：

```
prog.rb:5: unexpected return (LocalJumpError)
from prog.rb:5:in `call'
from prog.rb:6
```

下面例子中返回操作失败是由于 `block` 是在一个线程中创建的而在另一个线程中被调用。

```
def meth3
  yield
end

t = Thread.new do
  meth3 { return }
end

t.join
```

输出结果：

```
prog.rb:6: return can't jump across threads (ThreadError)
from prog.rb:9:in `join'
from prog.rb:9
```

`Proc` 对象的情景稍微有点复杂。如果使用 `Proc.new` 从 `block` 中创建一个 `proc` 对象，该 `proc` 行为类似于 `block`，则前面的规则也适用。

```
def meth4
  p = Proc.new { return 99 }
  p.call
  puts "Never get here"
end

meth4      →      99
```

如果 Proc 对象是使用 Kernel.proc 或 Kernel.lambda 来创建的，则它的行为更像自立的方法体：return 会从 block 中返回到 block 的调用者。

```
def meth5
  p = lambda { return 99 }
  res = p.call
  "The block returned #{res}"
end

meth5 → "The block returned 99"
```

因此，如果你使用 Module#define_method，你可能想传递一个用 lambda 而不是 Proc.new 创建的 proc 给该方法，因为前面形式中的 return 会取得希望的效果，而后面的形式会产生 LocalJumpError。

22.13 异常 Exceptions

Ruby 异常是类 Exception 或其子类的对象（第 462 页的图 27.1 给出了内建异常的完整列表）。

22.13.1 引发异常 Raising Exceptions

Library Kernel.raise 方法可以引发一个异常。

```
raise
raise string
raise thing [, string [ stack trace ]]
```

第一种形式重新引发 \$! 中存储异常，如果 \$! 是 nil，则引发一个新的 RuntimeError。

第二种形式创建一个新的 RuntimeError 异常，并设置其消息为给定的字符串。

第三种形式通过在其第一个参数上调用 exception 方法创建一个异常对象，然后设置异常消息和调用栈为第二个和第三个参数。

类 Exception 和其对象含有一个称为 exception 的工厂方法，所以异常类的名字和实例可以用作 raise 的第一个参数。

当异常发生时，Ruby 会将异常对象的引用存储在全局变量 \$! 中。

22.13.2 处理异常

Handling Exceptions

在如下地方可以处理异常。

- 在 `begin/end block` 的作用域内，

```
begin
  code...
  code...
  [rescue [parm, ...] [=> var] [then]
   error handling code... , ... ]
  [else
   no exception code... ]
  [ensure
   always executed code... ]
end
```

- 在方法体内，

```
def method and args
  code...
  code...
  [rescue [parm,...] [ => var ] [ then ]
   error handling code... , ...]
  [else
   no exception code... ]
  [ensure
   always executed code... ]
end
```

- 以及一个可执行语句的后面。

```
statement [rescue statement, ...]
```

一个 `block` 或方法可以有多个 `rescue` 语句，每个 `rescue` 语句可以指定零个或多个异常参数。不带参数的 `rescue` 语句被当作它好像带一个 `StandardError` 参数。这意味着一些底层的异常无法被不带参数的 `rescue` 类捕获。如果你想捕获任何异常，使用：

```
rescue Exception => e
```

当发生异常时，Ruby 向上扫描调用栈，直至找到该异常外层的 `begin/end block`、方法体或带 `rescue` 修饰符的语句。对 `block` 中的每个 `rescue` 语句，Ruby 依次比较当前被引发的异常和 `rescue` 的每个参数；每个参数被 `parameter === $1` 测试。如果引发的异常和 `rescue` 的一个参数匹配，则 Ruby 执行 `rescue` 的代码体，并停止查找。如果匹配的 `rescue` 语句以 `=>` 和一个变量名结束，那么该变量将被设置为 `$!`。

尽管 `rescue` 语句的参数通常是异常类的名字，但实际上它们可以是返回适当类的任意表达式（包括方法调用）。

如果没有 `rescue` 语句和发生的异常匹配，Ruby 将向上一层栈中搜索匹配异常的高一级的 `begin/end block`。如果一个异常被传递到主线程的顶层还没有被捕获，那么程序将输出一个消息并终止运行。

如果 `else` 语句存在，且 `code` 没有引发异常，那么其代码体将被执行。当执行 `else` 语句时发生的异常不会被和 `else` 在同一个 `block` 中的 `rescue` 语句捕获。

如果 `ensure` 语句存在，当 `block` 退出时它的代码体总会被执行（即使存在未被捕获的异常处理）。

在 `rescue` 语句中，不带参数的 `raise` 将重新引发`$!` 异常。

Rescue 语句修饰符

一条语句可以含有一个可选的 `rescue` 修饰符，该修饰符后跟另一条语句（还可以跟另一个 `rescue` 修饰符，等等）。如果 `rescue` 修饰符不含异常参数，则 `rescue` 将捕获 `StandardError` 及其子类。

如果异常在 `rescue` 修饰符的左边发生，那么左边的语句将被放弃，而且整行的值为右边语句的值。

```
values = [ "1", "2.3", /pattern/ ]
result = values.map { |v| Integer(v) rescue Float(v) rescue String(v) }
result → [1, 2.3, "(?-mix:pattern)"]
```

对 Block 进行 Retry

`retry` 语句可以用在 `rescue` 语句中以从头重新运行 `begin/end block`。

22.14 Catch 和 Throw

Catch and Throw

Library

方法 `Kernel.catch` 将执行与之相关联的 `block`。

```
catch ( symbol | string ) do
  block...
end
```

Library

方法 `Kernel.throw` 将中断语句的正常执行。

```
throw( symbol | string [, obj] )
```

当 `throw` 执行时，Ruby 在调用栈中向上搜索直到找到匹配符号或字符串的第一个 `catch block`。如果找到，则搜索结束，并从被捕获的 `block` 后面继续执行。如果 `throw` 的第二个参数存在，那么它的值将作为 `catch` 的值返回。当搜索对应的 `catch` 时，Ruby 检查它遇到的任何 `block` 表达式的 `ensure` 语句。

如果没有匹配 `throw` 的 `catch block`，Ruby 将会在 `throw` 的位置引发一个 `NameError` 异常。



Duck Typing

Duck Typing

你应该已经注意到在 Ruby 中我们没有声明变量和方法的类型——一切都是某种对象。

现在，似乎人们对此有两种反应。一些人喜欢这种灵活性，对用动态类型的变量和方法编写代码感觉心应手。如果你是这样的人，你可能希望跳到下页的“类不是类型（Classes aren't Types）”章节。然而，另一些人对不受限制地使用所有这些变量会感到紧张不安。如果你是从 C# 或 Java 等语言转向 Ruby 的，你会觉得 Ruby 太不严谨了，以至于无法用它来编写“真正”的应用，在那些语言中，你已经习惯了为所有的变量和方法都指定一个类型。

不！

我们会用几个段落的篇幅来试图说服你，缺乏静态类型对编写可靠的应用来说并不是一个问题。在这里我们不是试图谴责别的语言。相反，我们只是对比方法上的不同。

现实情况是，在大多数主流语言中，静态类型系统在程序安全方面没有真正地起到太大作用。如果 Java 的类型系统可靠，举例来说，Java 就无须实现 ClassCastException 异常。但是这个异常是需要的，因为 Java 中存在运行时的类型非确定性（与 C++、C# 和别的语言一样）。静态类型（typing）有助于代码优化，并通过工具提示（tooltip）帮助那些集成开发环境（IDE）做得更聪明些，但是我们还没有看到很多证据表明它促进了编写更为安全可靠的代码。

另一方面，一旦用了 Ruby 一段时间，你意识到动态类型的变量实际上在很多方面提高了你的生产率。你也会惊讶地发现，对类型混乱感到惶恐其实是毫无根据的。大型的、长时间运行的 Ruby 程序执行一些关键应用，却并不抛出任何类型相关的错误。为什么？

部分原因是：这是一个常识问题。如果用 Java 编程（Java 1.5 以前版本），所有容器实际上都是无类型的：容器里的任何东西都是 `Object`，当抽取出元素时需要把它转换成所需的类型。当运行这些程序时，可能永远无法看到 `ClassCastException` 异常。你的代码结构不允许：你放入 `Person` 对象，然后取出 `Person` 对象。你不会编写以其他方式工作的程序。

好吧，在 Ruby 中也如此。如果你出于某种目的使用了一个变量，当你接下来再次使用它时，几乎总可能是因为相同的原因而使用。所谓可能会发生的混乱根本就不会发生。

在这之上，有丰富 Ruby 编程经验的人往往倾向于采用某种编程风格。他们编写了很多简短的方法，并一边编写一边测试。简短的方法意味着大多数变量的作用范围是受限制的：这样，程序因类型而出错的情况就少些。同时测试会帮助尽早发现那些愚蠢的错误：输入错误（typo）以及类型错误，使它们没有机会在代码中扩散开来。

结论是：在“类型安全”中的“安全”常常是一种错觉，用动态语言如 Ruby 编程是安全的，同时有很高的生产率。因此，如果你为 Ruby 缺乏静态类型感到紧张不安，我们建议你暂时把担心放在一旁，并试着用 Ruby 一段时间。一旦开始挖掘出使用动态类型的能力，那么你会惊讶地看到，你很少会见到因为类型问题而带来的错误，并会惊讶地发现你的生产率是如此之高。

23.1 类不是类型

Classes Aren't Types

类型的问题，实际上比强类型拥护者和支持动态类型的叛逆青年之间正在进行的争吵，要复杂得多。真正的问题是，类型究竟是什么？

如果有使用传统类型语言编程的经验，你可能被这样教育过，对象的类型（*type*）是它的类（*class*）——所有对象都是某个类的实例，类（*class*）是对象的类型（*type*）。类定义了对象能够支持的操作（方法），同时定义了这些方法所操作的状态（实例变量）。看看 Java 代码。

```
Customer c;
c = database.findCustomer("dave"); /* Java */
```

这段代码声明了变量 `c` 是 `Customer` 类型，把它设置为对 Dave 客户对象的引用，这个对象是从某些数据库记录中创建的。所以 `c` 里面的对象类型是 `Customer`，对吗？

可能是。但是，即使在 Java 中这个问题也比我们想象得要复杂。Java 支持 *interface* 概念，*interface* 是某种弱化的（emasculated）抽象基类。一个 Java 类可以声明其实现了某个接口。使用这个功能，你可以用如下方式定义你的类。

```

public interface Customer {
    long getID();
    Calendar getDateOfLastContact();
    // ...
}

public class Person
    implements Customer {
    public long getID() { ... }
    public Calendar getDateOfLastContact() { ... }
    // ...
}

```

所以即使在 Java 中，类并不总是类型——有时候类型是类的子集，并且有时候对象实现了多个类型。

在 Ruby 中，类从来（好吧，几乎从来）不是类型。相反，对象类型更多是根据对象能够做什么决定的。在 Ruby 中，它被称为 *duck typing*。如果对象能够像鸭子那样行走，像鸭子那样呱呱叫的话，那么解释器会很高兴地把它当成鸭子来对待的。

看一个例子，也许我们编写了一个方法，它把客户名字写入到一个已打开文件的后面。

```

class Customer
    def initialize(first_name, last_name)
        @first_name = first_name
        @last_name = last_name
    end
    def append_name_to_file(file)
        file << @first_name << " " << @last_name
    end
end

```

作为好的程序员，我们写了一个单元测试。需要注意的是，它相当地凌乱（但是马上我们就会改进它）。

```

require 'test/unit'
require 'addcust'

class TestAddCustomer < Test::Unit::TestCase
    def test_add
        c = Customer.new("Ima", "Customer");
        f = File.open("tmpfile", "w") do |f|
            c.append_name_to_file(f)
        end
        f = File.open("tmpfile") do |f|
            assert_equal("Ima Customer", f.gets)
        end
    ensure
        File.delete("tmpfile") if File.exist?("tmpfile")
    end
end

```

输出结果：

```
Finished in 0.003473 seconds.
1 tests, 1 assertions, 0 failure, 0 errors
```

我们所要做的，就是创建一个文件，然后重新打开它，读取其内容并验证是否写入了正确的字符串。等一切结束后还需要把文件删除（当然它还存在的话）。

然而，可以依赖 duck typing。需要某种言行有如文件的东西，我们把它传递给要测试的方法。在这种情况下，意味着我们需要这么一个对象，它通过附加一些特性来响应 <<方法。什么东西合适呢？普通的 String 怎么样？

```
require 'test/unit'
require 'addcust'

class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = ""
    c.append_name_to_file(f)
    assert_equal("Ima Customer", f)
  end
end
```

输出结果：

```
Finished in 0.001951 seconds.
1 tests, 1 assertions, 0 failures, 0 errors
```

被测试的方法认为它正写入到文件中，但是相反它只是添加到一个字符串中。最后我们可以检测写入的内容是正确的。

没有必要非得使用字符串，用数组也不错。

```
require 'test/unit'
require 'addcust'

class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = []
    c.append_name_to_file(f)
    assert_equal(["Ima", " ", "Customer"], f)
  end
end
```

输出结果：

```
Finished in 0.001111 seconds.
1 tests, 1 assertions, 0 failures, 0 errors
```

实际上，如果想检查插入的单个对象是否正确，用数组可能会更方便。

因此，*duck typing* 适合用来测试，但是用在程序体内会怎么样呢？在先前的例子中它使得测试变得容易，同样它也使得编写灵活的程序代码变得容易。

实际上，Dave 有过一次有趣的经历，*duck typing* 把他和他的一位客户拯救出来。他编写过一个大型的基于 Ruby 的 Web 应用，这个应用有一个数据库表，里面全部是某次竞赛中所有参赛者的个人信息。这个系统提供了一种逗号分隔值（comma-separated value, CSV）的下载功能，它允许管理员把这些信息导入到他们的本地电子制表（spreadsheet）中。

就在比赛将要开始时，电话铃响了。原来一直工作地好好的下载功能出问题了，由于需要太长的时间来响应请求以至于请求会超时。那时候紧张到了极点，因为管理员必须使用这些信息去作出赛程安排，同时发出邮件。

一番简单调试后，发现问题出在得到数据查询的结果并生成 CSV 下载的函数中。这个代码有点像如下的代码：

```
def csv_from_row(op, row)
  res = ""
  until row.empty?
    entry = row.shift.to_s
    if /[,"]/ =~ entry
      entry = entry.gsub(/"/, '""')
    res << '"' << entry << '"'
    else
      res << entry
    end
    res << "," unless row.empty?
  end
  op << res << CRLF
end
result = ""
query.each_row {|row| csv_from_row(result, row)}
http.write result
```

当处理中等规模的数据集时，这段代码表现得不错。但是当输入到达一定大小时，它的性能突然间马上慢下来了。谁是罪魁祸首？垃圾回收！这种方法会生成成百上千的临时字符串，同时它构造了一个很大的结果字符串，每次一行。当字符串不断变大时，就需要更多的空间，这时候垃圾回收会被触发，它扫描并删除所有这些临时的字符串。

解决办法简单而令人惊讶地有效。不是在处理的时候构造结果字符串，而是修改代码，把每个 CSV 行作为数组的元素保存到数组中。这意味着这些临时字符串仍然被引

用，它们不再是垃圾。这也意味着不再构造一个不断变大的字符串，以避免强迫触发垃圾回收。由于 **duck typing**，这个改变是微不足道的。

```
def csv_from_row(op, row)
  # as before
end
result = []
query.each_row {|row| csv_from_row(result, row)}
http.write result.join
```

我们把数组传递到 `csv_from_row` 方法中。因为它（隐含地）使用了 **duck typing**，方法本身没有改变：它不关心参数究竟是什么类型，继续把它产生的数据添加到它的参数中。在方法返回了结果后，我们把所有那些单个行连接到一个大的字符串中。这个改变把运行时间从 3 分钟以上减少到只需几秒钟。

23.2 像鸭子那样编码 Coding like a Duck

如果你想使用 **duck typing** 哲学来编写程序，只需要记住一件事情：对象的类型是根据 1.8 它能够做什么而不是根据它的类决定的。（实际上，正是由于此，Ruby 1.8 目前不赞成使用 `Object#type` 方法而使用 `Object#class` 方法：这个方法返回接受者的类，因为 `type` 这个名字会让人产生误解。）

在实践中这有什么意义？在某种层面上，它意味着测试对象的类通常没什么价值。

比如，你可能编写函数去把歌曲信息添加到一个字符串中。如果有编写 C# 或 Java 的经验，你倾向于把它写成：

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end

  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song) → "I Got Rhythm (Gene Kelly)"
```

拥抱和使用 Ruby 的 **duck typing** 吧，这样你会编写出非常简洁的代码。

```

def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song) → "I Got Rhythm (Gene Kelly)"

```

无须检查参数类型。如果它们支持`<<`（在这个例子中它是`result`）或`title`和`artist`（在这个例子中它是`song`），一切都运行得挺好。如果不支持，方法总之会抛出一个异常（就如同你检查其类型时会发生的那样）。但是取消这个检查的话，方法突然间变得非常灵活：可以向其传入数组、字符串、文件或任何别的使用`<<`来添加的对象，这个方法就可以工作。

有时候你想在使用这种自由的（*laissez-faire*）编程风格之余，对参数进行检查。你可能有好的理由检查参数是否可以做你所需的。如果因为检查了参数是所需的类，你会被`duck typing`俱乐部抛弃吗？不，你不会。¹但是你会考虑是根据对象的能力而不是它的类来做这个检查。

```

def append_song(result, song)
  # test we're given the right parameters
  unless result.respond_to?(:<<)
    fail TypeError.new("'result' needs '<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("'song' needs 'artist' and 'title'")
  end

  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song) → "I Got Rhythm (Gene Kelly)"

```

当然，在沿着这条路走下去之前，确保你真正从中受益——需要编写和维护很多额外的代码。

23.3 标准协议和强制转换

Standard Protocols and Coercions

尽管技术上不属于 Ruby 语言的一部分，但解释器和标准库会使用各种各样的协议来处理其他语言在使用类型时遇到的一些问题。

¹ 总之，这`duck typing`俱乐部并不会检查你是否是其中的一员……

一些对象会有多种自然表示。比如，你可能会编写一个类来表示罗马数字（I, II, III, IV, V 等等）。这个类并不一定是 `Integer` 的子类，因为其对象是数字的表示，自己未必就是数字。同时它们真的有点类似整数（integer）的特点。无论何时当 Ruby 预期见到整数时，如果能够使用我们罗马数字类的对象，就太好了。

为此，Ruby 提供了转换协议（*conversion protocols*）的概念——对象可以选择把自己转换成另外一个类的对象。Ruby 有三种标准的方式来实现它。

我们已经遇到了第一种方式。诸如 `to_s` 和 `to_i` 方法分别把它们的接收者转换成字符串和整数。这些转换方法不是非常严格的：比如，如果对象有某种得体的字符串表示，它可能会有 `to_s` 方法。为了返回数字的字符串表示（比如 VII），我们的 `Roman` 类可能会实现 `to_s` 方法。

第二种形式的转换函数使用名字如 `to_str` 和 `to_int` 的方法。它们是严格的转换函数：只有当对象能够很自然地用在字符串或者整数能够使用的任何地方，才应该实现它。比如，罗马数字的对象可以清楚地表示一个整数，因此应该实现 `to_int` 方法。但是涉及字符串的严格转换时，我们得好好想一想。

罗马数字显然可以用字符串来表示，但是这些数字是字符串吗？我们能在任何使用字符串的地方使用这些数字吗？不，可能不。从逻辑上讲，它们是数字的表示。你可以把它们表示成字符串，但是它们和字符串并非是插入兼容（plug-compatible）。因为这个原因，罗马数字不应实现 `to_str`——它不是真正的字符串。总结一下：使用 `to_s`，罗马数字可以转换成字符串，但是它本质上不是字符串，因此没有实现 `to_str`。

为了看看这实际上是如何工作的，我们考察一下打开的文件。`File.new` 的第一个参数可能是已有的文件描述符（用数字表示）或是要打开的文件名。当然，Ruby 不是只看第一个参数还检查类型是否是 `Fixnum` 或 `String`。相反，它给被传递进来的对象一个机会，让这个对象以数字或字符串来表示自己。如果用 Ruby 编写，它会写成这样：

```
class File
  def File.new(file, *args)
    if file.respond_to?(:to_int)
      IO.new(file.to_int, *args)
    else
      name = file.to_str
      # call operating system to open file 'name'
    end
  end
end
```

如果想传递保存在罗马数字中的文件描述符整数到 `File.new` 中，我们看看到底发生了什么。因为我们的类实现了 `to_int`，第一个 `respond_to?` 测试会成功、我们把数字的整数表示传递到 `IO.open`，同时返回文件描述符，所有这一切都被包装到新的 `IO` 对象中。

少数几个严格的转换函数被内建到标准库中。

`to_ary → Array`

用在当解释器需要把对象转换成数组以传递参数或进行多个赋值时。

```
class OneTwo
  def to_ary
    [ 1, 2 ]
  end
end

ot = OneTwo.new
a, b = ot
puts "a = #{a}, b = #{b}"
printf("%d--%d\n", *ot)
```

输出结果：

```
a = 1, b = 2
1 -- 2
```

`to_hash → Hash`

用在当解释器期待看到散列表时（唯一已知的用法是 `Hash#replace` 的第二个参数）。

`to_int → Integer`

用在当解释器期待看到整数值时（如文件描述符或 `Kernel.Integer` 的参数）。

`to_io → IO`

用在当解释器期待 `IO` 对象时（比如，`IO#reopen` 或 `IO.select` 的参数）。

`to_proc → Proc`

用在转换方法调用中的前缀有`&`符号的对象。

```
class OneTwo
  def to_proc
    proc { "one-two" }
  end
end

def silly
  yield
end

ot = OneTwo.new
silly(&ot) → "one-two"
```

to_str → String

普遍用在任何解释器寻找 String 值的地方。

```
class OneTwo
  def to_str
    "one-two"
  end
end

ot = OneTwo.new
puts("count: " + ot)
File.open(ot) rescue puts $!.message
```

输出结果：

```
count: one-two
No such file or directory -one-two
```

请注意，当然 `to_str` 使用的不是很普遍——一些想要字符串参数的方法不会调用 `to_str`。

```
File.join("/user", ot) → "/user/#<OneTwo:0x1c974c>"
```

to_sym→Symbol

表示接收者是符号。解释器没有使用它进行转换并且可能在用户代码中是无用的。

最后要说明的是：诸如 `Integer` 和 `Fixnum` 的类实现了 `to_int` 方法，`String` 类实现了 `to_sym` 方法。这样可以多态地调用这些严格的转换函数：

```
# it doesn't matter if obj is a Fixnum or a
# Roman number, the conversion still succeeds
num = obj.to_int
```

23.3.1 数字强制转换

Numeric Coercion

在第 372 页讲过解释器可以执行三种类型的转换。已经讨论了宽松的和严格的转换。第三种是数字强制转换（numeric coercion）。

问题是这样的。当写“`1+2`”时，Ruby 知道在对象 `1` (`Fixnum`) 上调用 `+`，把 `Fixnum 2` 作为参数传递给它。但是当写“`1+2.3`”时，同样的`+`方法现在接受 `Float` 参数。它如何知道该做些什么呢（尤其详细检查参数的类型违背了 duck typing 的精神）？

答案在 Ruby 中的基于 `coerce` 方法的强制（coercion）协议。`coerce` 的基本操作很简单。它接受两个数字（一个是接受者，另外一个是参数）。它返回包含两个元素的数组，分别是两个数字的表示（但是参数在前，后面跟着接受者）。`coerce` 方法保证两个

对象会有相同的类，因此可以对它们进行相加（或乘，或比较，或任何别的什么操作）。

```
1.coerce(2)          → [2, 1]
1.coerce(2.3)        → [2.3, 1.0]
(4.5).coerce(2.3)    → [2.3, 4.5]
(4.5).coerce(2)      → [2.0, 4.5]
```

技巧在于接受者调用其参数的 `coerce` 方法来生成数组。这种技术被称为两次分发 (*double dispatch*)，它允许方法根据它的类并且还有参数的类来改变自身的行为。在这个例子中，我们让参数决定到底应该对哪个类的对象进行相加（或乘，除等等）。

假设正在编写新的类来进行算术计算。为了使用强制转换，需要实现 `coerce` 方法。它接受一些其他类型的数字作为参数，并返回一个数组，其中包含同一类的两个对象，它们的值与其参数及其自身相等。

对于罗马数字类来说这相当简单。每个罗马数字对象在内部把它的实际值以 `Fixnum` 方式保存在实例变量 `@value` 中。`coerce` 方法查看参数的类是否也是 `Integer`。如果是，则返回参数和这个内部值。如果不是，则优先把两者都转换成浮点数。

```
class Roman
  def initialize(value)
    @value = value
  end

  def coerce(other)
    if Integer === other
      [other, @value]
    else
      [Float(other), Float(@value)]
    end
  end

  # .. other Roman stuff
end

iv = Roman.new(4)
xi = Roman.new(11)

3 * iv → 12
1.1 * xi → 12.1
```

当然，以此种方式实现的 `Roman` 类还不知道如何相加：无法在先前例子中编写 “`xi + 3`”，因为 `Roman` 类没有 “`plus`” 方法。也许它就应当这样。但是不管这些，让我们为罗马数字实现相加方法吧。

```

class Roman
  MAX_ROMAN = 4999
  attr_reader :value
  protected :value
  def initialize(value)
    if value <= 0 || value > MAX_ROMAN
      fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
    end
    @value = value
  end
  def coerce(other)
    if Integer === other
      [other, @value]
    else
      [Float(other), Float(@value)]
    end
  end
  def +(other)
    if Roman === other
      other = other.value
    end
    if Fixnum === other && (other + @value) < MAX_ROMAN
      Roman.new(@value + other)
    else
      x, y = other.coerce(@value)
      x + y
    end
  end
  FACTORS = [["m", 1000], ["cm", 900], ["d", 500], ["cd", 400],
             ["c", 100], ["xc", 90], ["l", 50], ["xl", 40],
             ["x", 10], ["ix", 9], ["v", 5], ["iv", 4],
             ["i", 1]]
  def to_s
    value = @value
    roman = ""
    for code, factor in FACTORS
      count, value = value.divmod(factor)
      roman << (code * count)
    end
    roman
  end
  iv = Roman.new(4)
  xi = Roman.new(11)

  iv + 3          →    vii
  iv + 3 + 4       →    xi
  iv + 3.14159    →    7.14159
  xi + 4900        →    mmmmmcmxi
  xi + 4990        →    5001

```

最后，应当小心使用 `coerce`——尝试总是强制转换到更通用的类型，否则可能最终造成强制转换循环，在那里 A 试图强制转换到 B，而 B 试图强制转换回到 A。

23.4 该做的做，该说的说

Walk the Walk, Talk the Talk

Duck typing 会引起争论。有人不时地在邮件列表中发生一通争吵，或有人在博客上支持或反对这个概念。很多参与到讨论中的人有一些相当极端的观点。

但是，根本上讲 duck typing 不是一组规则；它只是一种编程风格。当设计程序时，请在偏执和灵活中寻求平衡。如果觉得需要限制对象的类型，问一下自己为什么要限制。如果期待 `String` 但却得到 `Array` 时，试着确定在哪里出错了。有时候这个差异是特别地重要。尽管往往它不重要。试着使用这种宽松的编程风格并容许犯错一段时间，看看有什么糟糕的事情发生。如果没有的话，也许 duck typing 就不只是用于形容禽类的词汇了。





类与对象

Classes and Objects

类与对象无疑是 Ruby 的中心，但是第一眼看上去可能让人有点困惑。似乎有太多的概念：类、对象、class 对象、实例方法、类方法、单例类和虚拟类。然而，其实，Ruby 只有一个底层的类和对象结构，我们将在本章中详细讨论。实际上，基本模型是如此简单，以至于我们只需一段文字便可尽述了。

一个 Ruby 对象有三个部分：一组标志，一些实例变量，以及相关联的类。Ruby 类是 class 的一个对象，包括一个对象所具备的所有内容、外加方法列表和一个超类的引用（超类本身是另一个类）。Ruby 中的所有方法调用必须指派一个接收者（默认为 `self`，即当前对象）。Ruby 通过在接收者的类中查找方法列表，来找到要调用的方法。如果在其中没有找到，则在其包含的模块中查找，然后是父类，父类中包含的模块，继而是父类的父类，凡此以往。如果方法最终没有在接收者类或者其任何祖先中找到，Ruby 调用最初始接收者的 `method_missing` 方法。

就是这样——全部的解释。涵盖了本章的全部。

“但是等一下”，你喊道，“我为这一章是花了银子的。那么其他的内容呢——虚拟类、类方法等等。它们是如何工作的？”问得好。

24.1 类和对象是如何交互的

How Classes and Objects Interact

所有类/对象的交互，都是使用上面给出的简单模型解释的：对象引用类，而类引用零个或多个超类。不过，实现细节可能有些诡谲。

我们已经找到了最简单的方式，可视化地将 Ruby 实现的实际结构描绘出来。因此，在后面的篇幅中，我们查看类与对象的所有可能组合。注意，它们并非如 UML 的类图；我们所演示的是内存中的结构以及它们之间的指针。

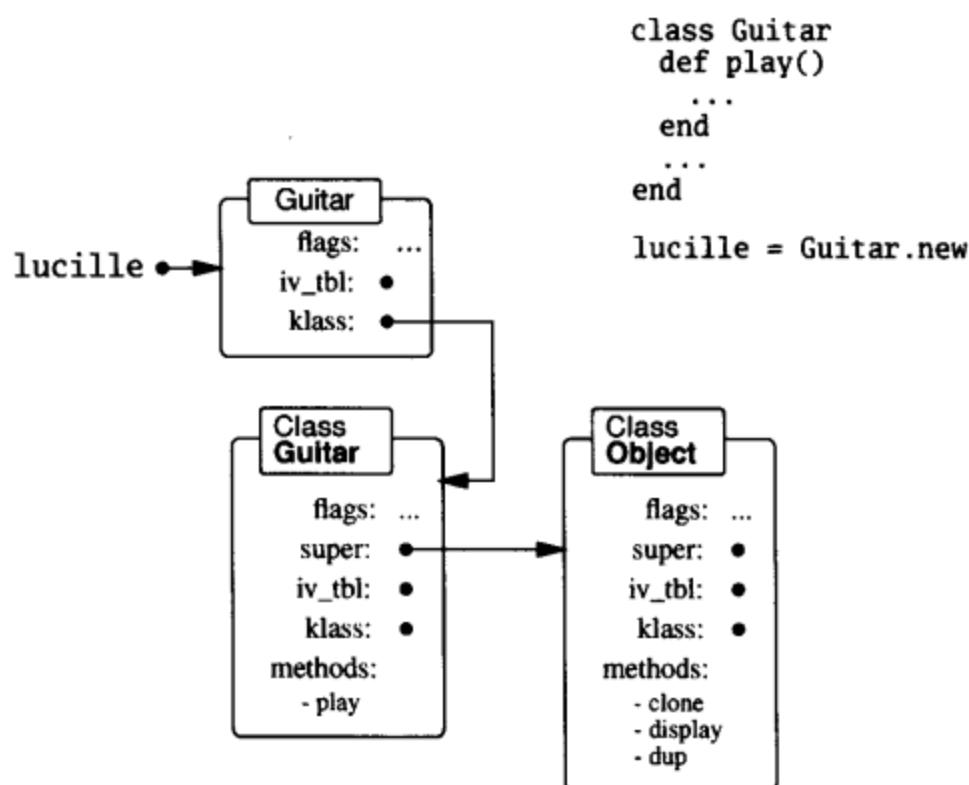


图 24.1 一个基本对象，以及它的类和超类

24.1.1 你的基本的、日常对象

Your Basic, Everyday Object

首先让我们查看一个从简单类创建的对象。图 24.1 演示了由变量 `lucille` 引用的一个对象 `Guitar`，以及类的超类 `Object`。注意，对象的类引用 `klass` 是如何指向 `class` 对象的，并且 `super` 指针如何指向父类。

如果我们调用 `lucille.play()` 方法，Ruby 找到接收者 `lucille`，然后沿着 `klass` 引用得到 `class` 对象 `Guitar`。它搜索方法表，找到 `play`，然后调用它。

如果我们转而调用 `lucille.display()`，Ruby 使用相同的方法，但是在 `Guitar` 类的方法表中无法找到 `display`。然后它沿着 `super` 引用得到 `Guitar` 的超类，在其中它会找到并调用这个方法。

24.1.2 什么是 Meta

What's the Meta?

敏锐的读者（是的，你们都不例外）会注意到，图 24.1 中 `Class` 对象的 `klass` 成员没有指向任何有意义的东西。现在我们具有所需的全部信息来找出它应该指向什么。

当你调用 `lucille.play()` 时，Ruby 沿着 `lucille` 的 `klass` 指针找到要查找实例方

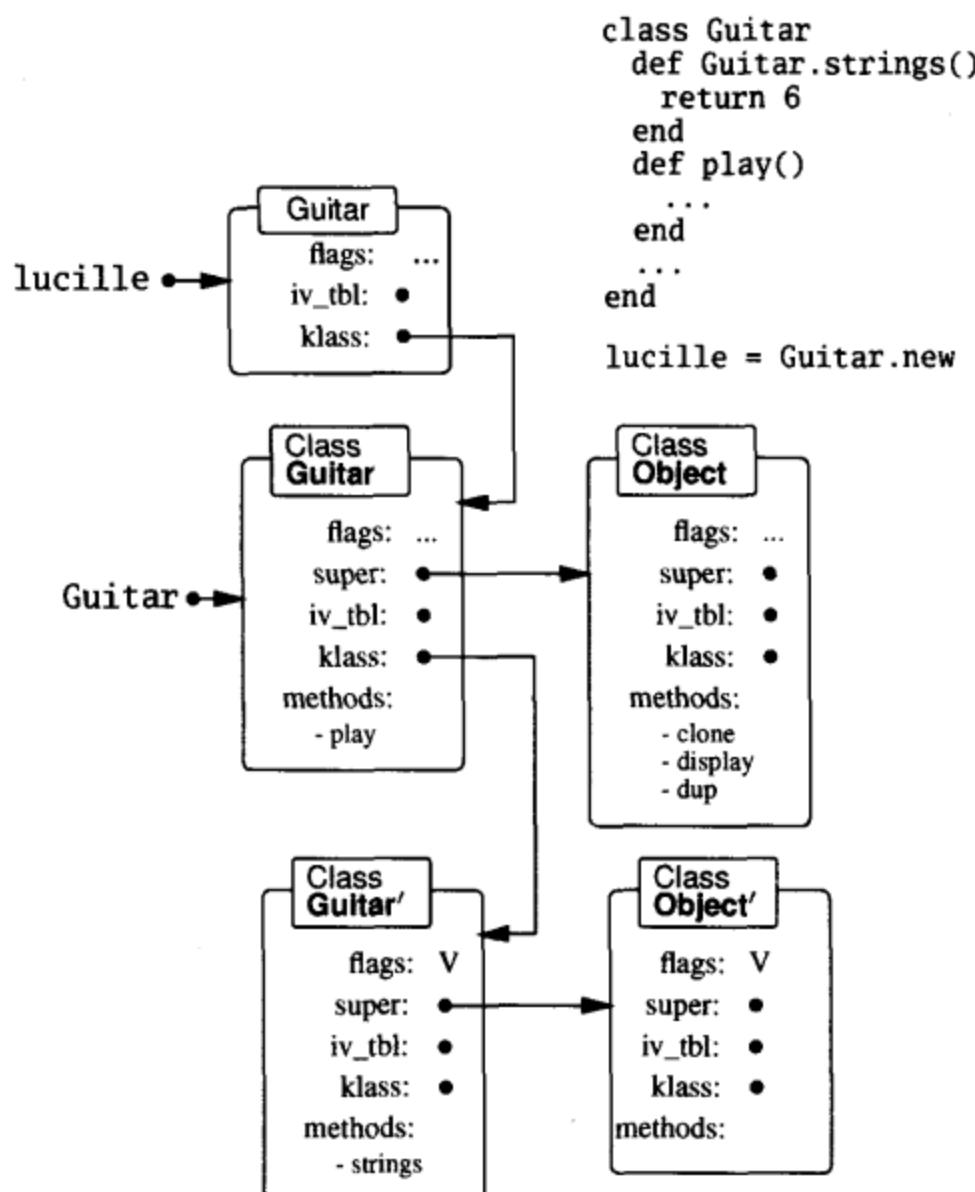


图 24.2 添加 Guitar 的 metaclass

法的 `class` 对象。当你调用类方法——例如 `Guitar.strings(...)`——时会发生什么呢？这里的接收者是 `class` 对象 `Guitar` 本身。因此，为了一致性，我们需要把这些方法放到其他类中，并由 `Guitar` 的 `klass` 指针所引用。这个新的类将包括 `Guitar` 的所有类方法。虽然术语还未确定，我们将它称为 *metaclass*（参见下一页的旁白）。我们把 `Guitar` 的 metaclass 记为 `Guitar'`。但是这还不是全部。因为 `Guitar` 是 `Object` 的子类，它的 metaclass `Guitar'` 则是 `Object` 的 metaclass (`Object'`) 的子类。在图 24.2 中，我们显示了这些额外的 metaclass。

当 Ruby 执行 `Guitar.strings()` 时，它沿用与之前一样的过程：找到接收者 `Guitar` 类；沿着 `klass` 引用得到 `Guitar'` 类；然后找到方法。

最后，注意 `Guitar'` 类的标志中有一个 `V`。这些由 Ruby 自动创建的类，被内在标记

Metaclass 和 Singleton (单例) 类

Metaclass and Singleton Classes

在本书的审校期间，对 *metaclass* 术语的使用引起了很多讨论，因为 Ruby 的 metaclass 和类似 Smalltalk 语言中的不同。最后，Matz 作出了下面的抉择：

你可以称之为 *metaclass*，但是和 Smalltalk 不同，它并非是一个关于类的类（class of a class）；它是一个类的 *singleton* 类。

- Ruby 中的每个对象都有自己的属性（方法、常量等等），在其他语言中被保存在类中。就如同每个对象都有其自己的类。
- 为了处理每个对象的属性，Ruby 为每个对象提供了一个好像类的东西，某些时候被称为 *singleton* 类。
- 在当前的实现中，*singleton* 类是具有特别标志的类，与对象和它们的类不同。如果语言的实现者愿意，可以把它们作为虚拟类。
- 类的 *singleton* 类的行为如同 Smalltalk 的 metaclass。

为虚拟类（virtual class）。虚拟类在 Ruby 内被不同地对待。最明显的不同是对外界来说它们实际上是不可见的：它们永远不会出现在像 `Module#ancesstors` 或 `ObjectSpace.each_object` 等方法返回的对象列表中，而你也无法使用 `new` 来创建其实例。

24.1.3 特定于 Object 的类 Object-Specific Classes

Ruby 允许你创建一个和特定对象绑定的类。在下面的示例中，我们创建了两个 `String` 对象。然后我们将一个匿名类关联到其中一个对象，覆写（override）对象基类中的一个方法并添加一个新的。

```
a = "hello"
b = a.dup

class <<a
  def to_s
    "The value is '#{self}'"
  end
  def two_times
    self + self
  end
end
```

```
a.to_s      →      "The value is 'hello'"
a.two_times →      "hellohello"
b.to_s      →      "hello"
```

这个示例使用 `class <<obj` 定义，基本的意思是“为对象 `obj` 构建一个新类”。我们也可以将它改写为：

```
a = "hello"
b = a.dup
def a.to_s
  "The value is '#{self}'"
end
def a.two_times
  self + self
end

a.to_s      →      "The value is 'hello'"
a.two_times →      "hellohello"
b.to_s      →      "hello"
```

两者的效果是相同的：向对象 `a` 中添加一个类。有关 Ruby 的实现，这给了我们强烈的提示：创建虚拟类并插入作为 `a` 的直接类。`a` 的原有类 `String`，作为虚拟类的超类。下一页中的图 24.3 演示了之前和之后的状况。

记住 Ruby 中的类是永不关闭的；你可以随时打开一个类并添加新的方法。这对虚拟类也适用。如果对象的 `klass` 引用已经指向了一个虚拟类，则并不会创建一个新的虚拟类。这意味着，前面示例中定义的第一个方法创建了一个虚拟类，而第二个为其添加了一个方法。

`Object#extend` 方法将其参数中的方法添加到调用该方法的接收者中，因此，如果需要的话，它也创建一个虚拟类。`obj.extend(Mod)` 基本和下面的等价：

```
class <<obj
  include Mod
end
```

24.1.4 Mixin 模块 Mixin Modules

当类包含一个模块时，模块的实例方法就变为类的实例方法。就好像模块变成类的超类。无须惊诧，这就是它工作的方式。当你包含一个模块时，Ruby 创建了一个指向该模块的匿名代理类，并将这个代理插入到实施包含的类中作为其直接超类。代理类包含有指向模块实例变量和实例方法的引用。这很重要：同一个模块可能被包含到不同的类中，并出现在许多不同的继承链中。不过，感谢代理类，它指向唯一的底层模块：改变

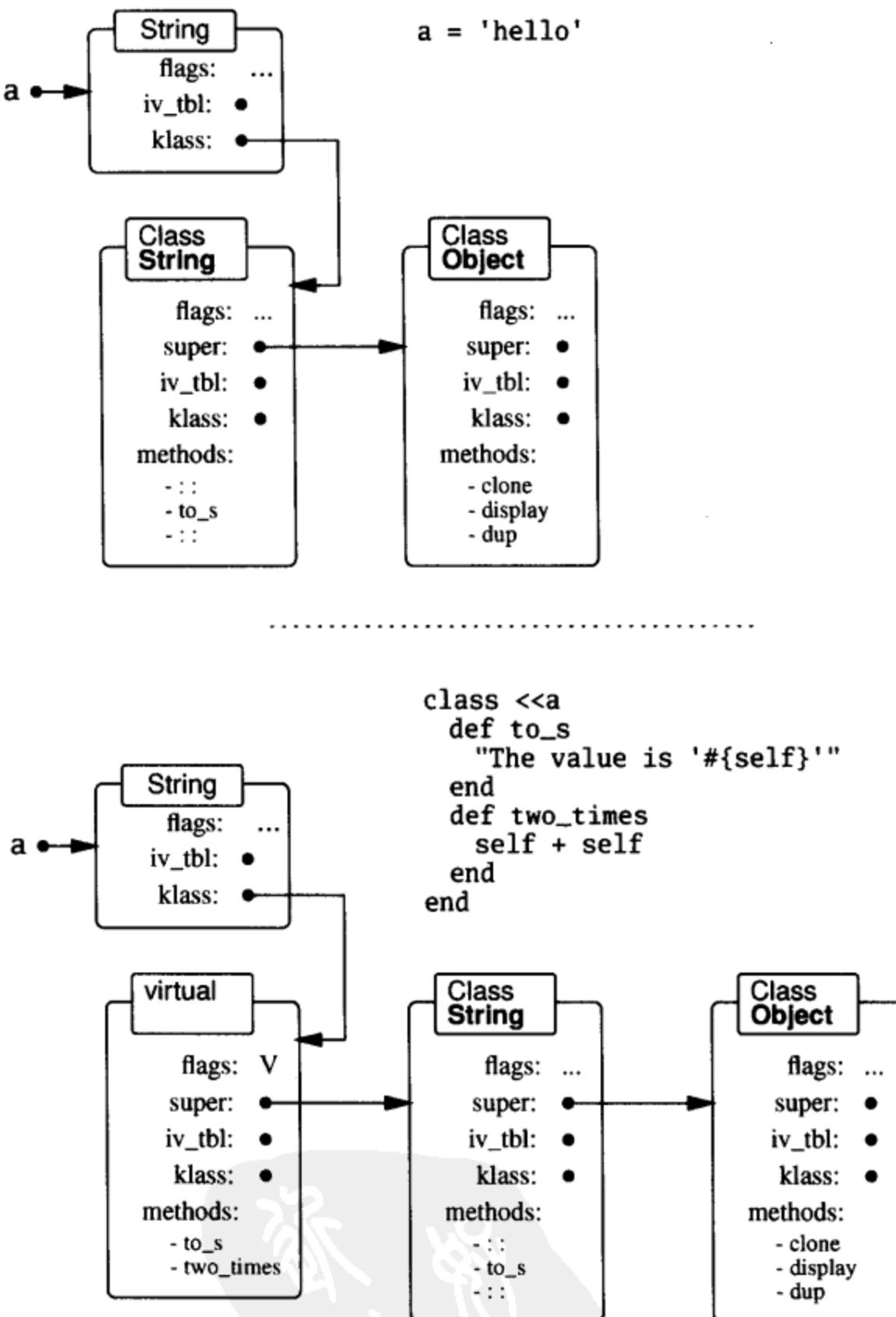


图 24.3 向对象中添加一个虚拟类

模块中某个方法的定义，它会改变所有包含该模块的类，无论过去或未来。

```
module SillyModule
  def hello
    "Hello."
  end
end

class SillyClass
  include SillyModule
end

s = SillyClass.new
s.hello      →      "Hello."
module SillyModule
  def hello
    "Hi, there!"
  end
end

s.hello      →      "Hi, there!"
```

下一页的图 24.4 演示了类和所包含 `mixin` 模块之间的关系。如果包含了多个模块，它们按次序插入到继承链中。

如果模块本身包含了其他模块，代理类的链会加入到包含该模块的所有类中，每个直接或间接包含的模块都对应有一个代理。

24.1.5 扩展对象 Extending Objects

就同你使用 `class <<obj` 为对象定义一个匿名类一样，你可以使用 `Object#extend` 将一个模块混合到对象中。例如：

```
module Humor
  def tickle
    "hee, hee!"
  end
end

a = "Grouchy"
a.extend Humor
a.tickle → "hee, hee!"
```

这是使用 `extend` 一个很有趣的技巧。如果你在一个类定义中使用它，模块的方法会变为类方法。这是因为调用 `extend` 和 `self.extend` 是等价的，因此方法被添加到 `self` 中，而在类定义中则是添加到类本身。

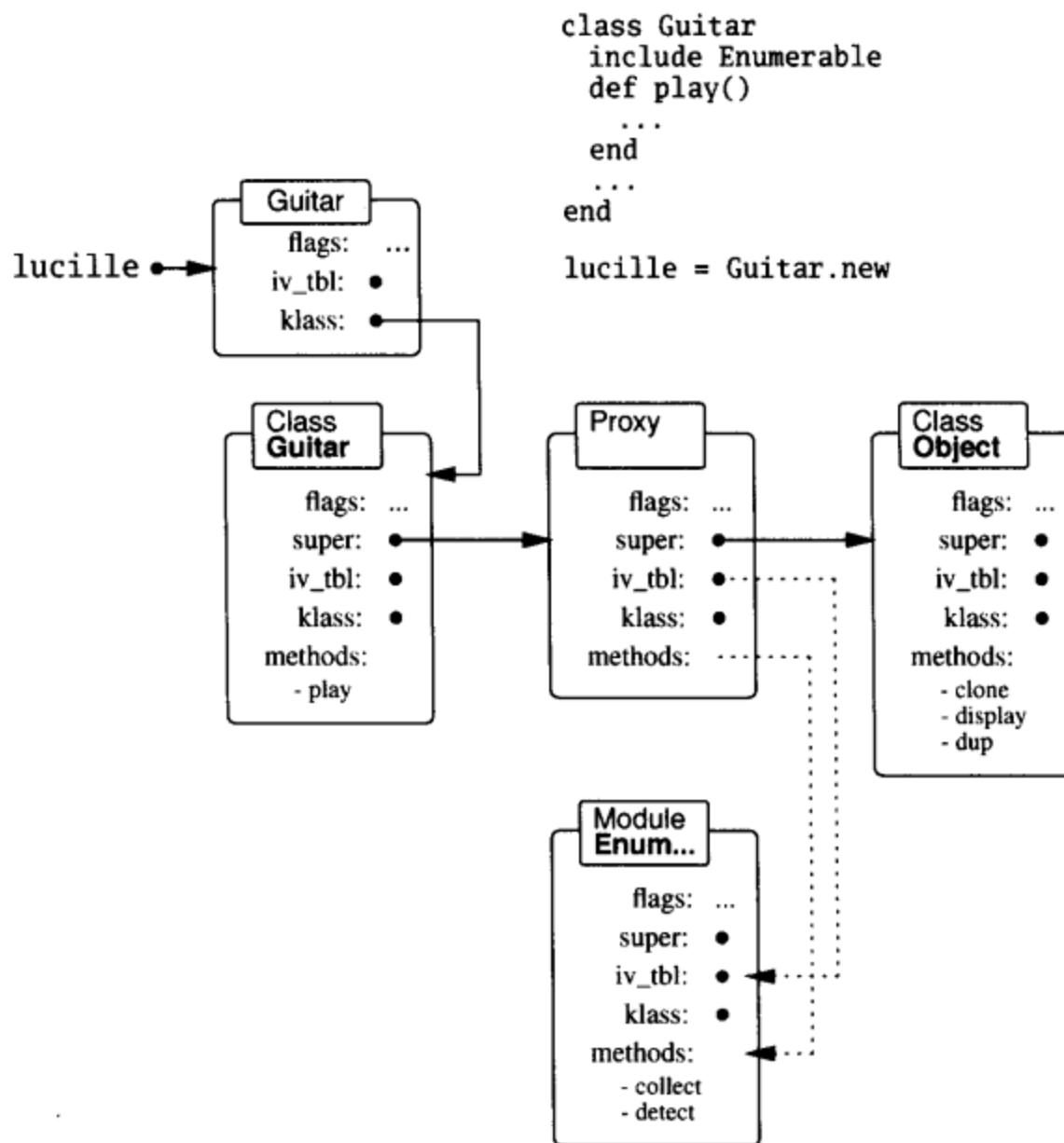


图 24.4 被包含的模块以及它的代理类

下面的示例将模块中的方法添加到类的级别。

```

module Humor
  def tickle
    "hee, hee!"
  end
end
class Grouchy
  include Humor
  extend Humor
end

Grouchy.tickle
a = Grouchy.new
a.tickle

```

→ "hee, hee!"
→ "hee, hee!"

24.2 类和模块的定义

Class and Module Definitions

在费心了解类和对象的组合之后，我们可以（谢天谢地）通过考察类和模块定义的基本成分，回归到编程的话题。

在如 C++ 或 Java 的语言中，类定义是在编译期处理的：编译期创建符号表，计算出需要分配多少存储空间，构造分发表（dispatch table），以及我们无须过多考虑的所有其他隐秘事情。

Ruby 则不同。在 Ruby 中，类和模块的定义是可执行的代码。虽然是在编译期进行解析，但当遇到定义时，类和模块是在运行时创建的。（这对方法定义也是成立的。）这可以让你可以比传统语言更动态地构架你的程序。你可以在类要被定义时，而非每次使用类的对象时，做出决定。下面示例中的类，当它要被定义时，决定要创建哪个版本的解密例程。

```
module Tracing
  # ...
end

class MediaPlayer
  include Tracing if $DEBUG

  if ::EXPORT_VERSION
    def decrypt(stream)
      raise "Decryption not available"
    end
  else
    def decrypt(stream)
      # ...
    end
  end
end
```

如果类定义是可执行的代码，这意味着它是在某个对象的上下文中执行的：`self` 必定指向了某个东西。让我们把它找出来。

```
class Test
  puts "Class of self = #{self.class}"
  puts "Name of self = #{self.name}"
end
```

输出结果：

```
Class of self = Class
Name of self = Test
```

这意味着类定义在执行时就是以这个类作为当前对象。回到第 380 页中有关 `metaclass`

的章节，我们可以看到这意味着 metaclass 及其超类中的方法，当执行方法定义时是可用的。让我们来看一看。

```
class Test
  def Test.say_hello
    puts "Hello from #{name}"
  end
  say_hello
end
```

输出结果：

```
Hello from Test
```

在这个示例中，我们定义了一个类方法，`Test.say_hello`，并且在类定义的代码体中调用它。在 `say_hello` 内部，我们调用 `name`，`Module` 类的一个实例方法。因为 `Module` 是 `Class` 的祖先类，因此在类定义中调用其实例方法，无须明确指定接收者。

24.2.1 类实例变量

Class Instance Variables

如果类定义是在某个对象的上下文中执行的，这意味着这个类可能具有实例变量。

```
class Test
  @cls_var = 123
  def Test.inc
    @cls_var += 1
  end
end

Test.inc      -> 124
Test.inc      -> 125
```

如果类有它们自己的实例变量，我们可以使用 `attr_reader` 等方法去访问它们吗？可以，但是我们必须在正确的地方运行这些方法。对一般的实例变量来说，属性访问方法是在类一级定义的。对类实例变量来说，我们必须在 metaclass 中定义访问函数。

```
class Test
  @cls_var = 123
  class <<self
    attr_reader :cls_var
  end
end

Test.cls_var -> 123
```

这把我们导向一个有趣的结论。许多我们在定义类或模块时使用的指令，例如 `alias_method`, `attr` 和 `public`，都是 `Module` 类中的方法。这创建了某些隐秘的可能性

——你可以通过编写 Ruby 代码来扩展类或模块定义的功能。让我们看几个示例。

第一个示例，让我们查看向模块和类添加基本的文档化功能。这可以让我们将一个字符串关联到所编写的模块和类，当程序运行时可以得到这个字符串。我们选择一个简单的语法。

```
class Example
  doc "This is a sample documentation string"
  # .. rest of class
end
```

我们要让 `doc` 方法在任何模块或类中都可使用，所以我们需要让它作为 `Module` 的一个实例方法。

```
class Module
  @@docs = {}

  # Invoked during class definitions
  def doc(str)
    @@docs[self.name] = self.name + ":\\n" + str.gsub(/^\\s/, '')
  end

  # invoked to get documentation
  def Module::doc(aClass)
    # If we're passed a class or module, convert to string
    # ('<=' for classes checks for same class or subtype)
    aClass = aClass.name if aClass.class <= Module
    @@docs[aClass] || "No documentation for #{aClass}"
  end
end

class Example
  doc "This is a sample documentation string"
  # .. rest of class
end

module Another
  doc <<-edoc
    And this is a documentation string
    in a module
  edoc
  # rest of module
end

puts Module::doc(Example)
puts Module::doc("Another")
```

输出结果：

```
Example:
This is a sample documentation string
Another:
And this is a documentation string
in a module
```

第二个实例是基于 Tadayoshi Funaba 的 date 模块（在第 665 页描述）的性能增强。假如我们有一个类表示某种底层的数量（本例中是日期）。这个类可能有许多属性，以多种不同方式表示底层的日期：Julian 天数¹、字符串、[年，月，日]的三元组等等。每个值都表示相同的日期，并且可能设计相当复杂的计算推导。这样我们希望当第一次访问某个属性时，只一次计算它的值。

手工的方式是为每个访问方法添加一个测试。

```
class ExampleDate
  def initialize(day_number)
    @day_number = day_number
  end

  def as_day_number
    @day_number
  end

  def as_string
    unless @string
      # complex calculation
      @string = result
    end
    @string
  end

  def as_YMD
    unless @ymd
      # another calculation
      @ymd = [y, m, d]
    end
    @ymd
  end
  #
end
```

这是一个笨重的方法——让我们看看能否搞出一些更性感的方法。

我们的目标是一个指令，它表明某个特定的方法体只应该被调用一次。第一次调用返回的值应该被缓存起来。此后，调用同一方法应该返回缓存的值，而无须再次重入方法体。这和 Eiffel 的例程修饰语 once 类似。我们希望用下面的方式编写代码：

```
class ExampleDate
  def as_day_number
    @day_number
  end

  def as_string
    # complex calculation
  end
```

¹ 译注：朱利安·凯撒制定的历法，即当前使用的阳历。

```

def as_YMD
  # another calculation
  [ y, m, d ]
end

once :as_string, :as_YMD
end

```

通过编写 `once` 作为 `ExampleDate` 的一个类方法，我们可以使用它作为一个指令，但是内部看起来应该如何呢？对每个方法，它创建原有代码的一个别名，然后用相同的名字创建一个新方法。下面是 Tadayoshi Funaba 的代码，稍稍重新做了格式化。

```

def once(*ids) # :nodoc:
  for id in ids
    module_eval <<-end;
      alias_method :__#{id.to_i}__, :#{id.to_s}
      private :__#{id.to_i}__
      def #{id.to_s}(*args, &block)
        (@__#{id.to_i}__ ||= [__#{id.to_i}__(*args, &block)))[0]
      end
    end;
  end
end

```

这段代码使用 `module_eval` 在调用它的模块（或者以本例来说，调用它的类）的上下文中执行一段代码。原来的方法被重命名为`__nnn__`，其中 `nnn` 部分是方法名符号 ID 的一个整数表示。代码使用相同的名字来缓冲实例变量。然后用原来的名字定义了一个方法。如果缓存的实例变量有值，则返回这个值；否则调用原来的方法，缓存并返回它的返回值。

理解了这段代码，你才真正迈向精通 Ruby 的道路。

不过，我们可以更进一步。查看 `date` 模块，你会看到 `once` 的代码稍有不同。

```

class Date
  class << self
    def once(*ids)
      # ...
    end
  end
  # ...
end

```

这里有趣的事情是内联类的定义，`class << self`。它基于 `self` 对象定义一个类，并且 `self` 恰巧是 `Date` 的 `class` 对象。结果呢？内联类定义中的每个方法，都自动成为 `Date` 的类方法。

`once` 特性有很强的适用性——它应该对所有类都可以工作。如果你将 `once` 提取出来并作为 `Module` 类的私有实例方法，那么所有的 Ruby 类都可以使用它。（当然你可以这样做，因为 `Module` 类是开放的，你可以自由地向其中添加方法。）

24.2.2 类名是常量

Class Names Are Constants

我们提到过当你调用一个类方法时，你所要做的是向 `Class` 对象本身发送一个消息。当你调用例如 `String.new("gumby")` 的方法时，你向 `String` 的 `class` 对象发送消息 `new`。但是 Ruby 如何处理它呢？总之，消息的接收者应该是一个对象引用，隐含着某处必定有一个名为 `String` 的常量，持有 `String` 对象的引用。²而且实际上，这就是实际所发生的事情。所有的内建类，以及你所定义的类，都有一个对应的全局常量，它的名字和你的类名相同。这很直接且微妙。微妙的地方在于，在系统中有两个名为 `String` 的东西。一个是指向 `String` 类的常量（`Class` 类的一个对象），另一个是（类）对象本身。

类名为常量的事实，意味着你可以把类和其他 Ruby 对象一样来对待：你可以拷贝它们，将它们作为参数传入方法，或者在表达式中使用它们。

```
def factory(klass, *args)
  klass.new(*args)
end

factory(String, "Hello")           →      "Hello"
factory(Dir, ".")                 →      #<Dir:0x1c90e4>

flag = true
(flag ? Array : Hash)[1, 2, 3, 4]   →      [1, 2, 3, 4]
flag = false
(flag ? Array : Hash)[1, 2, 3, 4]   →      {1=>2, 3=>4}
```

还有另一个侧面：如果一个无名的类被赋值给一个常量，Ruby 将常量作为类名。

```
var = Class.new
var.name      →      ""
Wibble = var
var.name      →      "Wibble"
```

² 它是一个常量，不是一个变量，因为 `String` 是由大写字母开头的。

24.3 顶层的执行环境

Top-Level Execution Environment

我们在本书中多次宣称 Ruby 中所有事物都是对象。不过，我们一次又一次使用的一样东西看起来与此相悖——Ruby 顶层的执行环境。

```
puts "Hello, World"
```

看起来不是一个对象。我们好像编写的是某种 Fortran 或 BASIC 的变体。但是深入探究，你就会发现即便是最简单的代码也潜藏着对象和类。

我们知道，字面上的"Hello, World"产生一个 Ruby String 对象，所以它是一个对象。我们还知道直接调用 puts 方法实际等同于 self.puts。那么何为 self?

```
self.class → Object
```

在顶层，我们在某个预定义的对象上下文中执行代码。当我们定义方法时，我们实际创建了 Object 类的一个（私有的）实例方法。这十分微妙；因为它们在 Object 类中，这些方法在任何地方都可以获得。而且因为我们在 Object 的上下文中，能够以函数的形式使用 Object 的方法（包括从 Kernel 中 mixin 的方法）。这解释了为什么我们可以在顶层（实际上在 Ruby 的任何地方）调用 Kernel 中的方法，例如 puts；这些方法是每个对象的一部分。

顶层的实例变量也属于这个顶层对象。

24.4 继承与可见性

Inheritance and Visibility

关于类继承最后一个诡秘的地方，十分地含糊。

在一个类定义中，你可以修改祖先类中方法的可见性。例如，你可以像下面那样：

```
class Base
  def aMethod
    puts "Got here"
  end
  private :aMethod
end

class Derived1 < Base
  public :aMethod
end

class Derived2 < Base
end
```

在这个示例中，我们可以调用 `Derived1` 类实例中的 `aMethod`，但是不能通过 `Base` 或 `Derived2` 的实例来调用它。

那么 Ruby 是如何辗转让一个方法有两个不同的可见性呢？很简单，它使诈。

如果一个子类改变了父类中某个方法的可见性，Ruby 实际上在子类中插入了一个隐藏的代理方法，使用 `super` 调用原有的方法。然后，它将这个代理的可见性设置为你所需要的。这意味着，下面的代码：

```
class Derived1 < Base
  public :aMethod
end
```

实际等同于：

```
class Derived1 < Base
  def aMethod(*args)
    super
  end
  public :aMethod
end
```

`super` 调用可以访问父类中的方法，而不管其可见性，所以重写方法可以让子类覆写父类中的可见性规则。相当骇人，嗯？

24.5 冻结对象

Freezing Objects

有时候你费尽力气才让你的对象完全正确，如果允许其他人可以改变它，你会受到惩罚。也许你需要通过某个第三方的对象，在你的两个类之间传递某种不知类型（opaque）的对象，而你希望确保它在到达时是未经改动的。也许你希望使用一个对象作为散列表的 `key`，并确保当它被用作 `key` 时没有人能修改它。也许有什么东西破坏了你的一个对象，而你想让 Ruby 在改变发生时立即抛出一个异常。

Ruby 提供了一种非常简单的机制来帮助实现上述需求。任何对象都可以通过调用 `Object#freeze` 被冻结。一个被冻结的对象是不能修改的：你不能改变它的实例变量（无论直接或间接），你不能为其关联单例方法，而且，如果它是类或模块，你不能添加、删除或更改它的方法。一旦被冻结，对象将始终保持冻结状态：没有 `Object#thaw` 方法可用。你可以使用 `Object#frozen?` 来判断对象是否被冻结。

当你拷贝一个被冻结的对象时，会发生些什么呢？这取决于你使用的方法。如果你调用对象的 `clone` 方法，整个对象的状态（包括它是否被冻结）都会被拷贝到新对象中。另一方面，`dup` 通常只拷贝对象的内容——新拷贝不会继承冻结的状态。

```
str1 = "hello"
str1.freeze          →      "hello"
str1.frozen?        →      true
str2 = str1.clone
str2.frozen?        →      true
str3 = str1.dup
str3.frozen?        →      false
```

虽然冻结对象最初看起来是个好主意，不过在你遇到实际需求之前，最好不要这样做。冻结是在纸面上看起来十分必要而在实践中所用不多的想法之一。



Ruby 安全

Locking Ruby in the Safe

Walter Webcoder 有一个非常棒的想法：设计一个 Web 算术页面。该页面是含有一个文本域以及一个按钮的简单 Web 表单，并被各种各样的非常酷的数学链接和横幅广告包围，使得看起来丰富多彩。用户输入一个算术表达式到文本域中，并按下按钮，然后结果就会被显示出来。一夜之间，世界上所有计算器都变得无用了；Walter 大大获利，然后他退休并把他的余生用于收集车牌号。

Walter 认为实现这样一个计算器很容易。他可以用 Ruby 的 CGI 库访问表单域中的内容，再用 eval 方法把字符串当作表达式来计算。

```
require 'cgi'

cgi = CGI.new("html4")

# Fetch the value of the form field "expression"
expr = cgi["expression"].to_s

begin
  result = eval(expr)
rescue Exception => detail
  # handle bad expressions
end

# display result back to user...
```

Walter 把这个应用程序放到网上才几秒钟，来自 Waxahachie 的一个 12 岁的小孩在表单中输入了 `system("rm")`，随他计算机上的文件一起，Walter 的美梦一下子破灭了。

Walter 得到了一个重要的教训：所有外部数据都是有危险的。不要让它们靠近那些可能改动你的系统的接口。在这个案例中，表单中的内容是外部数据，而对 eval 的调用正是一个安全漏洞。

幸运的是，Ruby 为减少这种风险提供了支持。所有来自外部的数据都可以标记为被污染的（*tainted*，或不受信的）。当运行在安全模式下时，传递被污染的对象给一个具有潜在威胁的方法会引发 `SecurityError`。

25.1 安全级别

Safe Levels

变量`$SAFE`决定了 Ruby 的安全级别。第 401 页的表 25.1 列出了每种安全级别下所执行的检查的详细列表。

| <code>\$SAFE</code> | 限 制 |
|----------------------------|---------------------------------|
| 0 | 对外部提供（受污染）的数据不做检查。这是 Ruby 的默认模式 |
| ≥ 1 | Ruby 不允许在具有潜在威胁的操作中使用受污染的数据 |
| ≥ 2 | Ruby 禁止从外部可写的位置装载程序文件 |
| ≥ 3 | 所有新创建的对象都被认为受污染了 |
| ≥ 4 | Ruby 有效地将运行的程序分为两部分。未污染的对象不能被修改 |

在大多数情况下，`$SAFE` 的默认值是 0。然而，如果 Ruby 运行 `setuid` 或 `setgid`¹ 的脚本，或者在 `mod_ruby` 下执行，它的安全级别会自动变为 1。安全级别还可以通过`-T` 命令行选项设置，也可以通过在程序中对`$SAFE` 重新赋值来设置。但是通过对`$SAFE` 赋值不能降低其值的大小。

当创建新的线程时，`$SAFE` 的当前值将被继承下来。然而，在每个线程中，可以修改`$SAFE` 的值而不影响其他线程中该变量的值。使用这种特性可以实现安全“沙箱（sandboxes）”，在其中可以安全地运行外部代码而不影响应用程序或系统的其他部分。通过把从外部文件中装载的代码封装在一个它自己的匿名模块中可以实现这点。这可以保护你的程序的名字空间不受意外的改动。

```
f=open(filename, "w")
f.print ...      # write untrusted program into file.
f.close
Thread.start do
  $SAFE = 4
  load(filename, true)
end
```

因为`$SAFE` 等于 4，你只能装载被封装的文件。详见第 524 页中关于 `Kernel.load` 的描述。

这个概念被 Clemens Wyss 用来实现 Channel(<http://www.ruby.ch>)。在这个网站上，你可以运行本书第 1 版中的代码。你也可以在窗口中输入并执行 Ruby 代码。而且该网站是全天候的，因为它在一个沙箱中运行你的代码。

¹ Unix 脚本可以被标志为以不同于运行它的用户的用户 ID 或组 ID 来执行。这使得脚本可以获得它的使用者不具有的权限；该脚本可以访问通常它无权访问的资源。这种脚本被称为 `setuid` 或者 `setgid`。

从 http://www.approximity.com/cgi-bin/rubybuch_wiki/wpage.rb?nd=214 中你可以看到沙箱的源代码列表。

- 1.8 当创建 `Proc` 对象时，实际的安全级别被存储在该对象中。如果 `Proc` 被污染了，且当前的安全级别大于创建 `block` 时的实际级别，则 `Proc` 不能作为参数传递给方法。

25.2 受污染的对象 Tainted Objects

任何从外部资源（例如，从文件中读取的字符串或环境变量）派生的 Ruby 对象都被自动标记为被污染。如下面代码所示，如果程序从受污染的对象中派生了一个新的对象，那么这个新对象也将被污染。以前在某个位置获得外部数据的对象也将被污染。不管当前的安全级别是什么，此污染判断过程总会被执行。使用 `Object#tainted?` 方法可以看到一个对象是否已被污染。

| | |
|------------------------------|-------------------------------|
| # internal data | # external data |
| <code># =====</code> | <code># =====</code> |
| <code>x1 = "a string"</code> | <code>y1 = ENV["HOME"]</code> |
| <code>x1.tainted?</code> | <code>y1.tainted?</code> |
| <code>→ false</code> | <code>→ true</code> |
| <code>x2 = x1[2, 4]</code> | <code>y2 = y1[2, 4]</code> |
| <code>x2.tainted?</code> | <code>y2.tainted?</code> |
| <code>→ false</code> | <code>→ true</code> |
| <code>x1 =~ /([a-z])/</code> | <code>y1 =~ /([a-z])/</code> |
| <code>\$1.tainted?</code> | <code>\$1.tainted?</code> |
| <code>→ 0</code> | <code>→ 2</code> |
| <code>→ false</code> | <code>→ true</code> |

`taint` 方法可以把任意对象强制转换成受污染的对象。如果安全级别低于 3，`untaint`² 方法可以把对象的受污染标记去除。轻易不能做这种操作。

显而易见，Walter 应当在 1 安全级别运行他的 CGI 脚本。当程序试图传递表单数据给 `eval` 方法时将引发一个异常。一旦发生这种情况，Walter 会有几个可选解决方案。他可以实现一个合适的表达式解析器，以避免使用 `eval` 的潜在威胁。然而，由于人的惰性，更可能的是对表单数据执行一些简单的检查，如果无危害，就去掉数据的污染标记。

² 即使不用 `untaint` 方法也可以用其他技巧来做到这一点。我们将留给你自己去发现如何做。

```
require 'cgi';
$SAFE = 1

cgi = CGI.new("html4")
expr = cgi["expression"].to_s

if expr =~ %r{(\A[-+*/\d\seE.()]*\z)}
  expr.untaint
  result = eval(expr)
  # display result back to user...
else
  # display error message...
end
```

从个人角度来说，我们认为 Walter 仍有些冒险。我们可能更希望看到一个真正的解析器，但是实现这样一个解析器对阐述对象污染没有什么意义，所以我们将进入其他话题。

表 25.1 安全级别的定义

\$SAFE >= 1

- 不处理环境变量 RUBYLIB 和 RUBYOPT。且不添加当前路径到路径列表中。
 - 不允许 -e, -i, -I, -r, -s, -S 和 -x 命令行选项。
 - 如果\$PATH 中的任何目录是任何人可写的，则不能从\$PATH 中运行程序。
 - 不能处理或 chroot 到一个名字为被污染的字符串的目录。
 - 不能 glob 受污染的字符串。
 - 不能 eval 受污染的字符串。
- 1.8.
- 不能 load 或 require 名字是被污染的字符串的文件（除非 load 被封装了）。
 - 不能处理或查询名字是受污染的字符串的文件或管道。
 - 不能执行系统命令或 exec 来自外部字符串的程序。
 - 不能传递受污染的字符串给 trap。

\$SAFE >= 2

- 不能改变、创建或删除目录，不能使用 chroot。
- 不能从任何人可写的目录中装载文件。
- 不能装载名字是以~开头的受污染的字符串的文件。
- 不能使用 File#chmod, File#chown, File#lstat, File.stat, File#truncate, File.umask, File#flock, O#ioctl, IO#stat, Kernel#fork, Kernel#syscall, Kernel#trap, Process.setpgid, Process.setsid, Process.setpriority, 或 Process.egid=。
- 不能用 trap 处理信号。

\$SAFE >= 3

- 所有对象在创建时就被污染。
- 不能 untaint 对象。

\$SAFE >= 4

- 不能修改未被污染的数组、hash 或字符串。
- 不能修改全局变量。
- 不能访问未污染对象的实例变量。
- 不能改变环境变量。
- 不能关闭或重新打开未污染的文件。
- 不能 freeze 未污染的对象。
- 不能改变 (private/public/protected) 方法的可见性。
- 在未污染的类或模块内不能创建别名。
- 不能获得元信息（例如方法或变量列表）。
- 不能在未污染的类或模块内定义、重定义、删除或取消定义方法。
- 不能修改 Object。
- 不能从未污染的对象中删除实例变量或常量。
- 除了当前线程不能处理线程、终止线程，或者设置 abort_on_exception。
- 没有线程内的局部变量。
- 不能在具有较低\$SAFE 值的线程内引发异常。
- 不能在 ThreadGroups 间移动线程。
- 不能调用 exit, exit! 或者 abort。
- 只能装载被封装的文件，且在未污染的类或模块中不能 include 模块。
- 不能把符号标志符转换成对象引用。
- 不能向文件或管道中写东西。
- 不能使用 autoload。
- 不能 taint 对象。

反射, ObjectSpace 和分布式 Ruby

Reflection, ObjectSpace, and DistributedRuby

能够内省 (*introspect*) 是 Ruby 等动态语言的诸多优点之一，那就是在程序内部自己检验程序的方方面面。Java 把这个特性称为反射 (*reflection*)，而 Ruby 在这方面的本领超过了 Java。

反射这个词让我们的头脑中浮现出一幅在镜子中观察自身的景象——也许你正在研究自己头顶上的秃点在无情地扩散。这是十分恰当的类比：使用反射去检验程序中那些通常从我们所在位置看不到的部分。

在极度内省的情绪中，当我们正气沉丹田并积蓄能量时（小心不要把这两个任务弄颠倒了），对程序内省我们会了解到什么呢？我们可能会发现：

- 包括哪些对象。
- 类的层次结构。
- 对象的属性和方法。
- 有关方法的信息。

有了这些信息后，就可以查看特定的对象并决定在运行时调用它们的那个方法——即使当开始编写代码时这些对象的类不存在。我们也可以开始做些更聪明的事情，比如当程序正在运行的时候修改它。

听起来挺可怕吗？不用害怕。实际上那些反射 (*reflection*) 本领让我们可以做一些非常有用的事情。在本章后面我们会看到分布式 Ruby 和列集 (*marshaling*)，这两个基于反射的技术可让我们穿越时空发送对象。

26.1 看看对象

Looking at Objects

你曾渴望在程序中拥有遍历所有现存 (living) 对象的本领吗? 我们有这个本领! 使用 Ruby 的 `ObjectSpace.each_object` 就可以遍历对象。可以使用它去实作各种各样优雅的窍门。

比如, 在 `Numeric` 类型的所有对象上迭代, 可以这么写:

```
a = 102.7
b = 95.1
ObjectSpace.each_object(Numeric) { |x| p x }
```

输出结果:

```
95.1
102.7
2.71828182845905
3.14159265358979
2.22044604925031e-16
1.79769313486232e+308
2.2250738585072e-308
```

嗨, 所有这些额外的数字来自何方? 我们没有在程序中定义它们。如果你查看第 487 页和第 540 页, 你会看到 `Float` 类为最大和最小浮点数定义了常量与 `epsilon`, 这是区分两个浮点数的最小差距。`Math` 模块为 `e` 和 `π` 定义了常量。既然程序要检查系统中所有现存的对象, 所以它们也被显示出来。

用不同的数字运行这个例子。

```
a = 102
b = 95
ObjectSpace.each_object(Numeric) { |x| p x }
```

输出结果:

```
2.71828182845905
3.14159265358979
2.22044604925031e-16
1.79769313486232e+308
2.2250738585072e-308
```

没有出现我们创建的那些 `Fixnum` 对象。这是因为 `ObjectSpace` 并不知晓这些具有立即值的对象: `Fixnum`, `Symbol`, `true`, `false` 和 `nil`。

26.1.1 窥入对象

Looking Inside Objects

一旦发现感兴趣的对象, 你可能会倾向于找出它能够做什么。在静态语言中变量类型决定了它的类以及它所支持的方法。与静态语言不一样, Ruby 支持自由的

(liberated) 对象。我们只有看看它的帽子下面到底有什么，才能知道对象的确切能力。¹ 第 365 页开始的 *Duck Typing* 一章对它进行了讨论。

例如，可以得到对象能响应的所有方法的一个列表。

```
r = 1..10 # Create a Range object
list = r.methods
list.length → 68
list[0..3] → ["collect", "to_a", "instance_eval", "all?"]
```

或者，可以看看对象是否支持某个具体的方法。

```
r.respond_to?("frozen?") → true
r.respond_to?(:has_key?) → false
"me".respond_to?("==") → true
```

可以确定对象的类以及它的唯一对象 ID，并测试它与其他类的关系。

```
num = 1
num.id → 3
num.class → Fixnum
num.kind_of? Fixnum → true
num.kind_of? Numeric → true
num.instance_of? Fixnum → true
num.instance_of? Numeric → false
```

26.2 考察类

Looking at Classes

知晓对象是反射 (reflection) 的一部分，但是为了得到全貌，你也需要能查看类——它们所包含的方法和常量。

考察类的层次结构很容易。可以使用 `Class#superclass` 得到任何类的父类。对类和模块来说，`Module#ancestors` 会同时列出超类和混入的 (mixed-in) 模块。

```
klass = Fixnum
begin
  print klass
  klass = klass.superclass
  print " < " if klass
end while klass
puts
p Fixnum.ancestors
```

输出结果：

```
Fixnum < Integer < Numeric < Object
[Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
```

¹ 对在大西洋东岸创建的对象来说，应该要看看它的苏格兰圆帽 (bonnet) 下面有些什么。

如果想构建一个完整的类层次结构，只要对系统中的每个类运行那段代码就可以了，可以使用 ObjectSpace 来迭代所有 Class 对象。

```
ObjectSpace.each_object(Class) do |klass|
  # ...
end
```

26.2.1 窥入类

Looking Inside Classes

可以在一个特别对象中找出关于其方法和常量的更多信息。它不是检查对象是否响应了一个给定的消息，可以通过访问级别（access level）来查找方法，可以只查找单例 18. (singleton) 方法，也可以查看对象的常量、局部变量和实例变量。

```
class Demo
  @@var = 99
  CONST = 1.23

  private
    def private_method
    end
  protected
    def protected_method
    end
  public
    def public_method
      @inst = 1
      i = 1
      j = 2
      local_variables
    end

    def Demo.class_method
    end
  end

  Demo.private_instance_methods(false)          → ["private_method"]
  Demo.protected_instance_methods(false)         → ["protected_method"]
  Demo.public_instance_methods(false)            → ["public_method"]
  Demo.singleton_methods(false)                  → ["class_method"]
  Demo.class_variables                         → ["@@var"]
  Demo.constants - Demo.superclass.constants → ["CONST"]

  demo = Demo.new
  demo.instance_variables                      → []
  # Get 'public_method' to return its local variables
  # and set an instance variable
  demo.public_method                          → ["i", "j"]
  demo.instance_variables                   → ["@inst"]
```

`Module.constants` 返回模块中所有可用的常量，它包括模块超类中的常量。我们此刻对这些常量不感兴趣，所以从列表中删除了它们。

- 1.8 你也许奇怪前面代码中那些 `false` 参数是什么。在 Ruby 1.8 中，默认情况下这些反射（reflection）方法会递归（reurse）进入父类中，同样它们的父类会沿着祖先链递归进入它们的父类中。传递 `false` 参数会阻止这种刨根问底（prying）。

给定一个方法名称的列表，可能会诱惑我们尝试调用它们。幸运的是，在 Ruby 中这是很容易的。

26.3 动态地调用方法

Calling Methods Dynamically

C 和 Java 程序员经常发现他们自己编写的某种类型的分发表（dispatch table）：根据命令调用函数。当需要把字符串转换成函数指针时，就想想典型的 C 的习惯用法。

```
typedef struct {
    char *name;
    void (*fptr)();
} Tuple;

Tuple list[] = {
    { "play", fptr_play },
    { "stop", fptr_stop },
    { "record", fptr_record },
    { 0, 0 },
};

...

void dispatch(char *cmd) {
    int i = 0;
    for (; list[i].name; i++) {
        if (strncmp(list[i].name, cmd, strlen(cmd)) == 0) {
            list[i].fptr();
            return;
        }
    }
    /* 没有发现 */
}
```

在 Ruby 中，可以只用一行代码就完成所有事情。把所有的命令函数塞到一个类中，然后创建类的一个实例（我们把它叫做 `commands`），并要求那个对象用 `command_string` 去执行同一名称的方法。

```
Commands.send(command_string)
```

哦，顺便说一下，它比 C 版本做的事情还要多——它是动态的。这个 Ruby 的版本可以很容易地找到在运行时添加的新方法。

我们无须编写特别的 `command` 类来支持 `send` 方法：它适用于任何对象。

```
"John Coltrane".send(:length) → 13
"Miles Davis".send("sub", /iles/, '.') → "M.Davis"
```

另一种动态调用方法的方式是使用 `Method` 对象。`Method` 对象类似 `Proc` 对象：它表示一段代码以及执行它所在的上下文。在这个例子中，代码是方法的程序体，而上下文是创建了这个方法的对象。一旦有了自己的 `Method` 对象，我们可以在后面通过发送消息 `call` 来执行它。

```
trane = "John Coltrane".method(:length)
miles = "Miles Davis".method("sub")

trane.call → 13
miles.call(/iles/, '.') → "M. Davis"
```

你可以像对待其他对象一样到处传递 `Method` 对象，当调用 `Method#call` 时，那个方法就会运行，就像是在原有对象上调用的那样。它类同 C 语言中的函数指针，且是以完全面向对象的风格来实现的。

你也可以和迭代器一起使用 `Method` 对象。

```
def double(a)
  2*a
end

mObj = method(:double)

[ 1, 3, 5, 7 ].collect(&mObj) → [2, 6, 10, 14]
```

`Method` 对象绑定到一个特别的对象。你可以创建非绑定方法（它的类是 1.8 `UnboundMethod`），然后将它们绑定到一个或多个对象。这个绑定创建新的 `Method` 对象。如别名（`alias`）一样，非绑定方法所指向的是其被创建时的有效方法定义。

```
unbound_length = String.instance_method(:length)
class String
  def length
    99
  end
end
str = "cat"
str.length → 99
bound_length = unbound_length.bind(str)
bound_length.call → 3
/* 译注：unbound_length 指向的依然是原来的 String.length 方法 */
```

俗话说：好事成双，这里还有另外一种方法可以动态地调用方法。`eval` 方法（和它的变种如 `class_eval`, `module_eval` 和 `instance_eval`）会解析和执行合法 Ruby 源码的任意字符串。

```

trane = %q("John Coltrane".length)
miles = %q("Miles Davis".sub(/iles/, '.'))

eval trane      → 13
eval miles     → "M. Davis"

```

使用 eval 时，明确指出表达式求值是使用的上下文而非当前上下文，是有帮助的。可以在想要的位置调用 Kernel#binding 来得到上下文。

```

def get_a_binding
  val = 123
  binding
end

val = "cat"

the_binding = get_a_binding
eval("val", the_binding)      → 123
eval("val")                  → "cat"

```

对 val 求值的第一个 eval，其执行所在的上下文，就像在 get_a_binding 方法中执行那样。在这个绑定中，val 变量的值是 123。第二个 eval 在顶层（toplevel）绑定中对 val 求值，在这里它的值是“cat”。

26.3.1 性能考虑

Performance Considerations

正如本节中所看到的，Ruby 提供了好几几种可以调用对象的任意方法：Object#send、Method#call 和各种各样的 eval 变种。

根据需要，也许你更喜欢使用其中的某种技术，但是应该意识到使用 eval 会比别的技术慢很多（或者可以对乐观的读者这么说，send 和 call 比 eval 快很多）。

```

require 'benchmark'
include Benchmark
test = "Stormy Weather"
m = test.method(:length)
n = 100000
bm(12) { |x|
  x.report("call") { n.times { m.call } }
  x.report("send") { n.times { test.send(:length) } }
  x.report("eval") { n.times { eval "test.length" } }
}

```

输出结果：

| | user | system | total | real |
|------|----------|----------|----------|-------------|
| call | 0.250000 | 0.000000 | 0.250000 | (0.340967) |
| send | 0.210000 | 0.000000 | 0.210000 | (0.254237) |
| eval | 1.410000 | 0.000000 | 1.410000 | (1.656809) |

26.4 系统钩子

System Hooks

钩子是一个跟踪 (trap) 一些 Ruby 事件的技术, 比如跟踪对象创建。在 Ruby 中, 最简单的 hook 技术是截取程序执行的系统命令。可能你想记录程序所执行的所有操作系统命令。只要重命名 Kernel.system 方法, 用你自己的 Kernel.system 替换它就可以了, 它会把系统命令记入日志并调用原来的 Kernel 方法。

```
module Kernel
  alias_method :old_system, :system
  def system(*args)
    result = old_system(*args)
    puts "system(#{args.join(' ', '})) returned #{result}"
    result
  end
end

system("date")
system("kangaroo", "-hop 10", "skippy")
```

输出结果:

```
Thu Aug 26 22:37:22 CDT 2004
system(date) returned true
system(kangaroo, -hop 10, skippy) returned false
```

一个更强大的钩子是捕获对象的创建。如果当创建每个对象时你都能在现场, 就可以完成各种有趣的事情: 你可以包装 (wrap) 它们, 向其中添加方法, 从其中删除方法, 以及把它们添加到容器中以实现持久性, 随你自由发挥。在这里给出一个简单的例子: 当每个对象创建时, 我们会为其添加时间戳。首先, 把 timestamp 属性添加到系统中的每个对象。这可以通过 hack Object 类本身来做到。

```
class Object
  attr_accessor :timestamp
end
```

然后, 我们需要 hook 对象的创建以添加时间戳。一种方式是在 Class#new 上使用方法重命名这个小技巧, 被调用的方法会为新的对象分配空间。这项技术不是完美无缺的——某些如字面量 (literal) 的内建对象并不是调用 new 来构造的——但是处理我们编写的对象没有问题。

```
class Class
  alias_method :old_new, :new
  def new(*args)
    result = old_new(*args)
    result.timestamp = Time.now
    result
  end
end
```

最后，我们可以测试一下。每隔几个毫秒便创建一个对象，同时检查它的时间戳。

```
class Test
end

obj1 = Test.new
sleep(0.002)
obj2 = Test.new

obj1.timestamp.to_f → 1116425239.97018
obj2.timestamp.to_f → 1116425239.98023
```

方法重命名技术是挺不错的，它真的起作用了，但是你应该要了解它可能会导致的问题。如果一个子类作同样的事情，用同样的名字对方法重命名，那么它最终会进入一个无限循环。你可以通过将方法的别名指定为一个唯一的符号名，或采用一致的命名规则来避免这个问题。

还有其他更好的方法可以进入到运行中的程序内部。Ruby 提供了几种回调方法，让你以受控的方式去跟踪某些事件。

26.4.1 运行时回调函数

Runtime Callbacks

无论何时发生了以下事件，你就会被通知。

| 事 件 | 回 调 函 数 |
|------------------------|-----------------------------------|
| 添加实例方法 | Module#method_added |
| 1.8 删除实例方法 | Module#method_removed |
| 1.8 取消实例方法的定义 | Module#method_undefined |
| 添加 singleton 方法 | Kernel.singleton_method_added |
| 1.8 删除 singleton 方法 | Kernel.singleton_method_removed |
| 1.8 取消 singleton 方法的定义 | Kernel.singleton_method_undefined |
| 子类化某个类 | Class#inherited |
| Mixin 一个模块 | Module#extend_object |

默认情况下，这些方法什么也不做。如果在类中回调函数被定义了，则它会自动地被调用。它们的实际调用顺序在每个回调函数的 Ruby 库文档描述中详细阐明了。

持续追踪方法和类的创建以及对模块的使用，可以让你精确刻画出程序的动态状态，这可能是很重要的。比如，你编写代码把所有的方法包装（wrap）到一个类中，可能是为了添加事务处理支持或实现某种形式的授权。但是任务只完成了一半：Ruby 的动态特性意味着类的用户可以在任何时候为其添加新的方法。使用回调函数，你就可以编写代码当创建那些新方法时对它们进行包装。

26.5 跟踪程序的执行

Tracing Your Program's Execution

当我们饶有兴趣地探查 (reflecting) 程序中的所有对象和类时, 不要忘记那些普普通通的语句, 正是它们才让代码实际地运作。我们发现 Ruby 也可以让我们跟踪语句。

首先, 你可以在解释器执行代码时观察 (watch) 它。只要是执行新的代码行、调用方法或创建对象等, `set_trace_func` 就可以执行一个 `Proc`, 并提供各种丰富的调试信息。你可以在第 529 页找到它的详细描述, 这里只是一个简单的尝试。

```
class Test
  def test
    a = 1
    b = 2
  end
end
set_trace_func proc {|event, file, line, id, binding, classname|
  printf "%8s %s:%-2d %10s %8s\n", event, file, line, id, classname
}
t = Test.new
t.test
```

输出结果:

| | | | |
|----------|------------|------------|--------|
| line | prog.rb:11 | | false |
| c-call | prog.rb:11 | new | Class |
| c-call | prog.rb:11 | initialize | Object |
| c-return | prog.rb:11 | initialize | Object |
| c-return | prog.rb:11 | new | Class |
| line | prog.rb:12 | | false |
| call | prog.rb:2 | test | Test |
| line | prog.rb:3 | test | Test |
| line | prog.rb:4 | test | Test |
| return | prog.rb:4 | test | Test |

`trace_var` 方法 (描述在第 532 页) 让你为全局变量添加一个钩子; 无论何时对这个全局变量赋值, `Proc` 对象都会被调用。

26.5.1 我们如何到这儿的

How Did We Get Here

一个公平的问题, 我们会经常这么问自己。尽管江郎才尽, 在 Ruby 中至少可以使用 `caller` 方法正确地找到 “*how you got there*”, 它返回 `String` 对象的数组, 表示当前的调用栈。

```
def cat_a
  puts caller.join("\n")
end
def cat_b
  cat_a
end
```

```
def cat_c
  cat_b
end
cat_c
```

输出结果：

```
prog.rb:5:in `cat_b'
prog.rb:8:in `cat_c'
prog.rb:10
```

一旦你已经了解是如何到这里的，下一步往哪儿走就取决于你自己了。

26.5.2 源代码

Source Code

Ruby 从纯文本文件中执行程序。你可以使用多种技术，通过查看文件来检验构成程序的代码。

`__FILE__` 这个特殊变量包含当前源文件的名称。我们有了一个相当短的（如果骗人的话）Quine——它可以将自身的源代码输出出来。

```
print File.read(__FILE__)
```

`Kernel.caller` 方法返回调用栈——当调用该方法时存在的栈帧（frame）的列表。表中的每一项以文件名、冒号以及文件中的行号开始。可以解析这些信息以显示源代码。在下面的例子中，我们有主程序 `mail.rb`，调用了另一个文件 `sub.rb` 中的方法。这个方法反过来调用一个代码块，在那里我们遍历调用栈并写出相应的源代码行。注意到这里使用了文件内容的散列表，它以文件名为索引。

这段代码转储（dump）了调用栈，其中包含源代码的信息。

```
def dump_call_stack
  file_contents = {}
  puts "File"                                Line Source Line"
  puts "-----+-----+-----+-----"
  caller.each do |position|
    next unless position =~ /\A(.*):(\d+)/
    file = $1
    line = Integer($2)
    file_contents[file] ||= File.readlines(file)
    printf("%-25s:%3d -%s", file, line,
          file_contents[file][line-1].lstrip)
  end
end
```

产生结果：这个（微不足道的）`sub.rb` 文件中包含了一个方法。

```
def sub_method(v1, v2)
  main_method(v1**3, v2**6)
end
```

这里是主程序，在被 `submethod` 回调后调用栈转储方法。

```
require 'sub'
require 'stack_dumper'
def main_method(arg1, arg2)
  dump_call_stack
end
sub_method(123, "cat")
```

输出结果：

| File | Line | Source Line |
|----------------------|------|---------------------------|
| code/caller/main.rb | : 5 | - dump_call_stack |
| ./code/caller/sub.rb | : 2 | - main_method(v1*3, v2*6) |
| code/caller/main.rb | : 8 | - sub_method(123, "cat") |

`SCRIPT_LINES` 常量与这个技术密切相关。如果程序用散列表初始化了 `SCRIPT_LINES` 常量，散列表会得到每个文件的源代码，这些文件是通过使用 `require` 或 `load` 依次被载入到解释器中的。详细的示例请见第 528 页的 `Kernal.require`。

26.6 列集和分布式 Ruby

Marshaling and Distributed Ruby

Java 提供了序列化 (*serialize*) 对象的特性，可以在某处保存对象，并且需要的话可以重建 (*reconstitute*) 它们。比如可以使用这个功能保存一个对象树，这些对象表示某部分程序的状态——一份文档，一张 CAD 绘图或一小段音乐等等。

Ruby 把这种类型的序列化称为列集 (*marshaling*)（想想铁路的装货场，在那里每辆汽车依次被装配到火车中，然后发送到别的地方）。可以使用 `Marshal.dump` 方法来保存一个对象和它的部分或所有组成对象 (*component*)。通常你从某些给定的对象作为起点，对其整个对象树进行转储，然后你可以使用 `Marshal.load` 方法来重建对象。

下面是一个简短的例子。这里有一个 `Chord` 类，它保存有一组音乐的音符 (`note`)。我们希望把一个特别美妙的旋律存储下来，这样可以通过电子邮件把它发送给数百个亲密的朋友。他们可以把它加载到自己的 Ruby 中并尽情地欣赏它。让我们从 `Note` 和 `Chord` 的类开始。

```
Note = Struct.new(:value)
Class Note
  def to_s
    value.to_s
  end
end
```

```

class Chord
  def initialize(arr)
    @arr = arr
  end
  def play
    @arr.join('-')
  end
end

```

现在开始创建我们的大作，并使用 `Marshal.dump` 把它的序列化版本保存到磁盘中。

```

c = Chord.new( [ Note.new("G"),
                 Note.new("Bb"),
                 Note.new("Db"),
                 Note.new("E") ] )
File.open("posternity", "w+") do |f|
  Marshal.dump(c, f)
end

```

最终我们的子孙读到它，并会为我们创作的美妙而欣喜若狂。

```

File.open("posternity") do |f|
  chord = Marshal.load(f)
end

chord.play → "G-Bb-Db-E"

```

26.6.1 定制序列化策略

Custom Serialization Strategy

并不是所有的对象都可以转储：绑定（binding）、过程（procedure）对象、IO 类的实例以及 singleton 对象，不能在正运行的 Ruby 环境之外保存（如果试着这么做，会抛出 `TypeError`）。即使你的对象并不包括这些问题对象，你也可能想自己控制对象的序列化。

1.8 `Marshal` 提供了你所需要的钩子（hook）。只要在需要定制序列化的对象中实现两个实例方法：一个是 `marshal_dump`，它把对象写到字符串，另一个是 `marshal_load`，它读取刚才创建的字符串并用它来初始化新分配的对象（在早期的 Ruby 版本中，你得使用 `_dump` 和 `_load` 方法，但是新版本的方法和 Ruby 1.8 的新分配方案（scheme）合作得更好）。`marshal_dump` 实例方法应该返回一个对象，它表示要被转储的状态。当接下来使用 `Marshal.load` 方法重新构建对象时，会对这个对象调用 `marshal_load` 来设置其状态——该方法执行时以目标加载类的已分配但没有初始化的对象为上下文。

比如，这里是一个示范类，它定义了自己的序列化。不管什么原因，`Special` 不希望把它其中的一个内部数据成员 `@volatile` 保存下来。编写者决定将其他两个实例变量序列化到一个数组中。

```

class Special
  def initialize(valuable, volatile, precious)
    @valuable = valuable
    @volatile = volatile
    @precious = precious
  end
  def marshal_dump
    [ @valuable, @precious ]
  end
  def marshal_load(variables)
    @valuable = variables[0]
    @precious = variables[1]
    @volatile = "unknown"
  end
  def to_s
    "#@valuable #@volatile #@precious"
  end
end
obj = Special.new("Hello", "there", "World")
puts "Before: obj = #{obj}"
data = Marshal.dump(obj)
obj = Marshal.load(data)
puts "After: obj = #{obj}"

```

输出结果：

```

Before: obj = Hello there World
After: obj = Hello unknown World

```

更多细节，请见从第 535 页开始的 `Marshal` 参考章节。

26.6.2 用 YAML 来列集

YAML for Marshaling

`Marshal` 模块内建在解释器中，它使用了二进制格式在外部保存对象。尽管速度快，这个二进制格式有一个很大的不足：如果解释器有了非常大的改变，这个 `marshal` 二进制格式可能也会改变，那么老的转储文件可能再也不能被载入。

另外一个办法是使用不那么严谨（less fussy）的外部格式，这个格式更适合用文本文件而不是二进制文件来表示。其中一个选择是 `YAML`²，Ruby 1.8 有一个标准库来支持它。

可以改写先前的 `marshal` 例子来使用 `YAML` 格式。而不是实现特定的载入和转储模块来控制 `marshal` 处理，在这里我们只是简单地定义了 `to_yaml_properties` 模块，它返回要被保存的实例变量列表。

² <http://www.yaml.org>. YAML 代表“YAML 不是标记语言”，但是这并不重要。

```

require 'yaml'
class Special
  def initialize(valuable, volatile, precious)
    @valuable = valuable
    @volatile = volatile
    @precious = precious
  end
  def to_yaml_properties
    %w{ @precious @valuable }
  end
  def to_s
    "#@valuable #@volatile #@precious"
  end
end
obj = Special.new("Hello", "there", "World")
puts "Before: obj = #{obj}"
data = YAML.dump(obj)
obj = YAML.load(data)
puts "After: obj = #{obj}"

```

输出结果：

```

Before: obj = Hello there World
After: obj = Hello World

```

可以看看作为对象的序列化格式，YAML 究竟创建了什么——非常简单。

```

obj = Special.new("Hello", "there", "World")
puts YAML.dump(obj)

```

输出结果：

```

--- !ruby/object:Special
precious: World
valuable: Hello

```

26.6.3 分布式 Ruby Distributed Ruby

既然可以把对象或一组对象序列化到一种适合进程外（out-of-process）存储的形式，也可以使用这个能力把对象从一个进程发送到另一个进程。把这个本领和联网的威力结合起来，瞧：我们有一个分布式对象系统了。为了避免自己写代码的麻烦，我们建议使用 Masatoshi Seki 的分布式 Ruby 库（drb），它现在是一个标准的 Ruby 库。
1.8

使用 drb 的 Ruby 进程可能会作为服务器、客户机或者两者都是。drb 服务器是对象的源，而客户机是对象的使用者。对客户机来说，这些对象表现为本地对象，但是实际上代码仍然在远程执行。

服务器通过将对象与一个给定的端口相关联来启动服务。线程在内部被创建来处理对这个端口的请求，因此当退出程序时，请记住等待（join）drb 线程终止。

```
require 'drb'
class TestServer
  def add(*args)
    args.inject{|n,v| n + v}
  end
end
server = TestServer.new
DRb.start_service('druby://localhost:9000', server)
DRb.thread.join # Don't exit just yet!
```

一个简单的 drb 客户机只是创建一个本地的 drb 对象，并把它与在远程服务器上的对象关联起来，这个本地对象只是一个代理（proxy）。

```
require 'drb'
DRb.start_service()
obj = DRbObject.new(nil, 'druby://localhost:9000')
# Now use obj
puts "Sum is: #{obj.add(1, 2, 3)}"
```

客户机连接上服务器并调用 add 方法，这个方法使用 inject 魔法（magic）对它的参数求和。返回结果并由客户机把它打印出来。

```
Sum is: 6
```

DRbObject 的初始 nil 参数表明我们希望附着（attach）到一个新的分布式对象。也可以使用现有的对象。

Ho hum（哼哼）。这听起来像 Java 的 RMI, CORBA 或别的什么。是的。它是一个全功能的分布式对象机制——但是它仅用几百行 Ruby 代码写成。没有任何 C 代码，也没有任何花哨的东西，它们只是普普通通的旧的 Ruby 代码。当然，它没有命名（naming）服务或交易（trader）服务，或任何你能够在 CORBA 中看到的东西，但是它简单，运行起来相当快。在 2.5GHz 的 Power Mac 机器上，这段示范代码每秒大约可以调用 1300 个远程的消息。

同时，如果你喜欢 Sun 的 JavaSpaces 样子（JINI 架构的基础），你会有兴趣了解 drb 是以一个短小的模块来发行的，它完成类似的事情。JavaSpaces 是基于一种被称为 Linda 的技术。为了显示 drb 日本开发者的幽默感，Ruby 版本的 Linda 被称为 Rinda。

- 1.8 如果你喜欢臃肿、麻木但支持互操作的远程消息调用，可以看看 Ruby 分发中 SOAP 库。³

³ 这是对 SOAP 的一个评论，它很早以前就抛弃了其首字母简略词中的 Simple。Ruby 的 SOAP 实现是了不起的一项工作。

26.7 编译时? 运行时? 任何时

Compile Time? Runtime? Anytime

使用 Ruby 要记住的一件重要事情是：在“编译时”和“运行时”之间没有太大的差别。它们都是相同的。可以向一个正运行的进程添加代码。可以动态地重新定义方法，把它们的作用范围从 `public` 改变成 `private` 等等。甚至可以修改像 `Class` 和 `Object` 那样的基本类型。

一旦习惯了这种灵活性，再让你回到如 C++ 的静态语言中甚至如 Java 的半静态语言，会是很困难的一件事情。

但是为什么你要回头呢？



第4部分 Ruby库的参考

Part IV Ruby Library Reference



Programming Ruby 中文版, 第2版

内置的类和模块

Built-in Classes and Modules

本章介绍内置在标准 Ruby 语言中的类和模块。它们自动被任意一个 Ruby 程序所拥有；不需要使用 `require`。本章不包含各种预定义的变量和常量；在第 333 页有其列表。

从第 427 页开始的描述中，我们展示了每个方法的用例。

`new`

`String.new(some_string) → new_string`

该描述展示了一个名为 `String.new` 的类方法。用斜体书写的参数表明传入了一个字符串，箭头表明该方法返回另一个字符串。因为返回值和参数的名字不同，因此它表示不同的对象。

当演示实例方法时，我们用斜体表示的伪（dummy）对象名作为接受者来展示方法调用。

`each`

`str.each(sep=$/) {|record| block} → str`

`String#each` 的参数具有默认值；当不带参数调用 `each` 时，将使用 `$/` 的值。这个方法是一个迭代器，所以调用后跟一个 `block`。`String#each` 返回它的接受者（receiver¹），所以接受者的名字（在这个例子中是 `str`）在箭头后再次出现。

有些方法具有可选参数。我们将这种参数置于尖括号中，例如`<xxx>`（此外，我们用符号`<xxx>*`表示 `xxx` 出现零次或多次，用`<xxx>+`表示 `xxx` 出现一次或多次）。

`index`

`self.index(str <,offset>) → pos 或 nil`

最后，对那些具有多种不同调用方式的方法，我们将各种形式列在不同的行上。

¹ 译注：即调用该方法的字符串对象。

27.1 字母顺序列表

Alphabetical Listing

标准类按字母顺序一一列出，其后是标准模块。在每个类或模块中，我们先列出类（或模块）方法，然后列出实例方法。

内置类概述

Array (427 页)：类方法：[], new. 实例方法：&, *, +, -, <<, <=>, ==, [], []=, !, assoc, at, clear, collect!, compact, compact!, concat, delete, delete_at, delete_if, each, each_index, empty?, eql?, fetch, fill, first, flatten, flatten!, include?, index, indexes, indices, insert, join, last, length, map!, nitems, pack, pop, push, rassoc, reject!, replace, reverse, reverse!, reverse_each, rindex, shift, size, slice, slice!, sort, sort!, to_a, to_ary, to_s, transpose, uniq, uniq!, unshift, values_at.

Bignum (441 页)：实例方法：Arithmetic operations, Bit operations, <=>, ==, [], abs, div, divmod, eql?, modulo, quo, remainder, size, to_f, to_s.

Binding (444 页)

Class (445 页)：类方法：inherited, new. 实例方法：allocate, new, superclass.

Continuation (448 页)：实例方法：call.

Dir (449 页)：类方法：[], chdir, chroot, delete, entries, foreach, getwd, glob, mkdir, new, open, pwd, rmdir, unlink. 实例方法：close, each, path, pos, pos=, read, rewind, seek, tell.

Exception (461 页)：类方法：exception, new. 实例方法：backtrace, exception, message, set_backtrace, status, success?, to_s, to_str.

FalseClass (464 页)：实例方法：&, ^, !.

File (465 页)：类方法：atime, basename, blockdev?, chardev?, chmod, chown, ctime, delete, directory?, dirname, executable?, executable_real?, exist?, exists?, expand_path, extname, file?, fnmatch, fnmatch?, ftype, grpowned?, join, lchmod, lchown, link, lstat, mtime, new, open, owned?, pipe?, readable?, readable_real?, readlink, rename, setgid?, setuid?, size, size?, socket?, split, stat, sticky?, symlink, symlink?, truncate, umask, unlink, utime, writable?, writable_real?, zero?. 实例方法：atime, chmod, chown, ctime, flock, lchmod, lchown, lstat, mtime, path, truncate.

File::Stat (477 页)：实例方法：<=>, atime, blksize, blockdev?, blocks, chardev?, ctime, dev, dev_major, dev_minor, directory?, executable?, executable_real?, file?, ftype, gid, grpowned?, ino, mode, mtime, nlink, owned?, pipe?, rdev, rdev_major, rdev_minor, readable?, readable_real?, setgid?, setuid?, size, size?, socket?, sticky?, symlink?, uid, writable?, writable_real?, zero?.

Fixnum (484 页)：类方法：. 实例方法：Arithmetic operations, Bit operations, <=>, [], abs, div, divmod, id2name, modulo, quo, size, to_f, to_s, to_sym, zero?.

Float (487 页)：实例方法：Arithmetic operations, <=>, ==, abs, ceil, divmod, eql?, finite?, floor, infinite?, modulo, nan?, round, to_f, to_i, to_int, to_s, truncate, zero?.

Hash (492 页)：类方法：[], new. 实例方法：==, [], []=, clear, default, default=, default_proc, delete, delete_if, each, each_key, each_pair, each_value, empty?, fetch, has_key?, has_value?, include?, index, indexes, indices, invert, key? keys, length, member?, merge, merge!, rehash, reject, reject!, replace, select, shift, size, sort, store, to_a, to_hash, to_s, update, value?, values, values_at.

Integer (501 页)：实例方法：ceil, chr, downto, floor, integer?, next, round, succ, times, to_i, to_int, truncate, upto.

IO (503页)：类方法：for_fd,foreach,new,open,pipe,popen,read,readlines,select,sysopen. 实例方法：<<,binmode,clone,close,close_read,close_write,closed?,each,each_byte, each_line,eof.eof?,fcntl,fileno, flush, fsync,getc,gets,iioctl,isatty,lineno,lineno=,pid, pos, pos=,print,printf,putc,puts,read,readchar, readline,readlines, reopen, rewind,seek,stat,sync, sync=,sysread,sysseek,syswrite,tell,to_i,to_io,tty?,ungetc,write.

MatchData (537页)：实例方法：[], begin, captures, end, length, offset, post_match, pre_match, select, size, string, to_a, to_s, values_at.

Method (543页)：实例方法：[], ==, arity, call, eql?, to_proc, unbind.

Module (545页)：类方法：constants, nesting, new. 实例方法：<, <=, >, >=, <=>, ===, ancestors, autoload,autoload?, class_eval, class_variables, clone, const_defined?, const_get, const_missing, const_set, constants, include?, included_modules, instance_method, instance_methods, method_defined?, module_eval, name, private_class_method, private_instance_methods, private_method_defined?, protected_instance_methods, protected_method_defined?, public_class_method, public_instance_methods, public_method_defined?. Private:alias_method, append_features, attr, attr_accessor, attr_reader, attr_writer, define_method, extend_object, extended, include, included, method_added, method_removed, method_undefined, module_function, private, protected, public, remove_class_variable, remove_const, remove_method, undef_method.

NilClass (561页)：实例方法：&, ^, I, nil?, to_a, to_f, to_i, to_s.

Numeric (562页)：实例方法：+@,-@,<=>,abs,ceil,coerce,div,divmod,eql?,floor, integer?, modulo, nonzero?, quo, remainder, round, step,to_int,truncate,zero?.

Object (567页)：实例方法：==, ===, =~, _id_, _send_, class, clone, display, dup, eql?, equal?, extend, freeze, frozen?, hash, id, initialize_copy, inspect, instance_eval, instance_of?, instance_variable_get, instance_variable_set, instance_variables, is_a?, kind_of?, method, method_missing, methods, nil?, object_id, private_methods, protected_methods, public_methods, respond_to?, send, singleton_methods, taint, tainted?, to_a, to_s, type, untaint. Private:initialize, remove_instance_variable, singleton_method_added, singleton_method_removed, singleton_method_undefined.

Proc (580页)：类方法：new. 实例方法：[], ==, arity, binding, call, to_proc, to_s.

Process::Status (591页)：实例方法：==, &, >>, coredump?, exited?, exitstatus, pid, signaled?, stopped?, success?, stopsig, termsig, to_i, to_int, to_s.

Range (597页)：类方法：new. 实例方法：==, ===, begin, each, end, eql?, exclude_end?, first, include?, last, member?, step.

Regexp (600页)：类方法：compile, escape, last_match, new, quote. 实例方法：==, ===, =~, ~, casefold?, inspect, kcode, match, options, source, to_s.

String (606页)：类方法：new. 实例方法：%*, +, <<, <=>, ==, ===, =~, [], []=, ~, capitalize, capitalize!,casecmp, center, chomp,.chomp!, chop, chop!, concat, count, crypt, delete, delete!, downcase, downcase!, dump, each_byte, each_line, empty?, gsub, gsub!, hex, include?, index, insert, intern, length, ljust, lstrip, lstrip!, match, next, next!, oct, replace, reverse, reverse!, rindex, rjust, rstrip, rstrip!, scan, size, slice, slice!, split, squeeze, squeeze!, strip, strip!, sub, sub!, succ, succ!, sum, swapcase, swapcase!, to_f, to_i, to_s, to_str, to_sym, tr, tr!, tr_s, tr_s!, unpack, upcase, upcase!, upto.

Struct (626页)：类方法：new, new, [], members. 实例方法：==, [], []=, each, each_pair, length, members, size, to_a, values, values_at.

Struct::Tms (630页)

Symbol (631 页) : 类方法: `all_symbols`. 实例方法: `id2name, inspect, to_i, to_int, to_s, to_sym`.

Thread (633 页) : 类方法: `abort_on_exception, abort_on_exception=, critical, critical=, current, exit, fork, kill, list, main, new, pass, start, stop`. 实例方法: `[], []=, abort_on_exception, abort_on_exception=, alive?, exit, group, join, keys, key?, kill, priority, priority=, raise, run, safe_level, status, stop?, terminate, value, wakeup`.

ThreadGroup (640 页) : 类方法: `new`. 实例方法: `add, enclose, enclosed?, freeze, list`.

Time (642 页) : 类方法: `at, gm, local, mktime, new, now, times, utc`. 实例方法: `+, -, <=, asctime, ctime, day, dst?, getgm, getlocal, getutc, gmt?, gmtime, gmt_offset, gmtoff, hour, isdst, localtime, mday, min, mon, month, sec, strftime, to_a, to_f, to_i, to_s, tv_sec, tv_usec, usec, utc, utc?, utc_offset, wday, yday, year, zone`.

TrueClass (650 页) : 实例方法: `&, ^, |`.

UnboundMethod (651 页) : 实例方法: `arity, bind`.

Summary of Built-in Modules

Comparable (447 页) : 实例方法: `Comparisons, between?`.

Enumerable (454 页) : 实例方法: `all?, any?, collect, detect, each_with_index, entries, find, find_all, grep, include?, inject, map, max, member?, min, partition, reject, select, sort, sort_by, to_a, zip`.

Errno (460 页)

FileTest (483 页)

GC (491 页) : 类方法: `disable, enable, start`. 实例方法: `garbage_collect`.

Kernel (516 页) : 类方法: `Array, Float, Integer, String, ` (backquote), abort, at_exit, autoload, autoload?, binding, block_given?, callcc, caller, catch, chomp, chomp!, chop, chop!, eval, exec, exit, exit!, fail, fork, format, gets, global_variables, gsub, gsub!, iterator?, lambda, load, local_variables, loop, open, p, print, printf, proc, puts, raise, rand, readline, readlines, require, scan, select, set_trace_func, sleep, split, sprintf, srand, sub, sub!, syscall, system, test, throw, trace_var, trap, untrace_var, warn`.

Marshal (535 页) : 类方法: `dump, load, restore`.

Math (540 页) : 类方法: `acos, acosh, asin, asinh, atan, atanh, atan2, cos, cosh, erf, erfc, exp, frexp, hypot, ldexp, log, log10, sin, sinh, sqrt, tan, tanh`.

ObjectSpace (578 页) : 类方法: `_id2ref, define_finalizer, each_object, garbage_collect, undefined_finalizer`.

Process (583 页) : 类方法: `abort, detach, egid, egid=, euid, euid=, exit, exit!, fork, getpgid, getpgrp, getpriority, gid, gid=, groups, groups=, initgroups, kill, maxgroups, maxgroups=, pid, ppid, setpgid, setpgrp, setpriority, setsid, times, uid, uid=, wait, waitall, wait2, waitpid, waitpid2`.

Process::GID (589 页) : 类方法: `change_privilege, eid, eid=, grant_privilege, re_exchange, re_exchangeable?, rid, sid_available?, switch`.

Process::Sys (594 页) : 类方法: `getegid, geteuid, getgid, getuid, issetugid, setegid, seteuid, setgid, setregid, setresgid, setresuid, setreuid, setrgid, setruid, setuid`.

Process::UID (596 页) : 类方法: `change_privilege, eid, eid=, grant_privilege, re_exchange, re_exchangeable?, rid, sid_available?, switch`.

Signal (604 页) : 类方法: `list, trap`.

Class **Array** < **Object**

数组是有序的、由整数索引的对象组成的集合（collection）。和 C 或 Java 一样，数组索引从 0 开始。负数下标是相对于数组尾部的，也就是说，-1 下标表示数组的最后一个元素，-2 下标表示倒数第二个元素，依此类推。

Mixes in

Enumerable:

```
all?, any?, collect, detect, each_with_index, entries, find, find_all,
grep, include?, inject, map, max, member?, min, partition, reject, select,
sort, sort_by, to_a, zip
```

类方法

[]

Array[<obj>*] → *an_array*

返回一个由给定对象生成的新数组。等价于操作符形式 `Array.[](...)`

| | | |
|---|---|----------------------------|
| <code>Array.[](1, 'a', /^A/)</code> | → | [1, "a", / ^A /] |
| <code>Array[1, 'a', /^A/]</code> | → | [1, "a", / ^A /] |
| <code>[1, 'a', /^A/]</code> | → | [1, "a", / ^A /] |

new

| | | |
|---|---|-----------------|
| <code>Array.new</code> | → | <i>an_array</i> |
| <code>Array.new(size=0, obj=nil)</code> | → | <i>an_array</i> |
| <code>Array.new(array)</code> | → | <i>an_array</i> |
| <code>Array.new(size) { <i>block</i> }</code> | → | <i>an_array</i> |

返回一个新数组。第一种形式生成的新数组是空的。第二种形式生成的数组由 *size* 个 *obj* 对象的拷贝组成（也就是对同一个对象 *obj* 的 *size* 个引用）。第三种形式创建参数数组（数组由调用参数的 `to_ary` 方法生成）的一个拷贝。最后一种形式将创建一个给定大小的数组。其中每个元素是运行相关联 *block* 的返回值（以元素下标为参数）。

| | | |
|--------------------------------|---|---------------------------|
| <code>Array.new</code> | → | [] |
| <code>Array.new(2)</code> | → | [nil, nil] |
| <code>Array.new(5, "A")</code> | → | ["A", "A", "A", "A", "A"] |

```
# 只创建了默认对象的一个实例
a = Array.new(2, Hash.new)
a[0]['cat'] = 'feline'
a   → [{"cat"=>"feline"}, {"cat"=>"feline"}]
a[1]['cat'] = 'Felix'
a   → [{"cat"=>"Felix"}, {"cat"=>"Felix"}]
```

```
a = Array.new(2) { Hash.new } # Multiple instances
a[0]['cat'] = 'feline'
a → [{"cat"=>"feline"}, {}]

squares = Array.new(5) {|i| i*i}
squares → [0, 1, 4, 9, 16]

copy = Array.new(squares) # initialized by copying
squares[5] = 25
squares → [0, 1, 4, 9, 16, 25]
copy → [0, 1, 4, 9, 16]
```

实例方法

&

arr & other_array → *an_array*

交集——返回一个含有两个数组共有元素的新数组，且其中没有重复的元素。比较元素的规则和比较 hash 键的规则一样。如果需要类似集合（set）的行为，请参见第 731 页的 Set 类。

```
[1, 1, 3, 5] & [1, 2, 3] → [1, 3]
```

*

*arr * int* → *an_array*
*arr * str* → *a_string*

重复——如果参数实现了 `to_str`，则等价于 `arr.join(str)`。否则返回由串联 *int* 个 *arr* 拷贝组成的一个新数组。

```
[1, 2, 3] * 3 → [1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3] * "--" → "1--2--3"
```

+

arr + other_array → *an_array*

串联——返回通过串联已有的两个数组而形成的一个新数组。

```
[1, 2, 3] + [4, 5] → [1, 2, 3, 4, 5]
```

-

arr - other_array → *an_array*

数组差集——返回一个新数组，该数组是原数组的拷贝，并删除了出现在 *other_array* 中的元素。如果需要类似于集合的差集操作，请参见第 731 页的类 Set。

```
[1, 1, 2, 2, 3, 3, 4, 5] - [1, 2, 4] → [3, 3, 5]
```

<<

arr << obj → *arr*

添加——将给定的对象添加到数组的末尾。这个表达式返回数组本身，所以可以链接多个添加操作。参见 `Array#push`。

```
[1, 2] << "c" << "d" << [3, 4] → [1, 2, "c", "d", [3, 4]]
```

<=>

arr <=> other_array → -1, 0, +1

比较——根据数组小于、等于或大于 *other_array*, 分别返回 -1, 0 或 +1。每个数组中的每个对象都被比较（使用 `<=>`）。如果任何值不相等, 那么这次比较的结果将作为返回值返回。如果所有的值都相等, 则返回值由数组长度的比较结果确定。因此, 根据 `Array#<=>` 当且仅当两个数组长度相同且对应的元素分别相等时才认为它们相等。

```
[ "a", "a", "c" ]      <=> [ "a", "b", "c" ] → -1
[ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]           → 1
```

==

arr == obj → true 或 false

等价——如果两个数组含有相同数目的元素, 并且对应的元素分别相等 (根据 `Object#==`), 则它们相等。如果 *obj* 不是数组, 则使用 `to_ary` 将它转换成数组, 然后返回 *obj == arr* 的结果。

```
[ "a", "c" ] == [ "a", "c", 7 ] → false
[ "a", "c", 7 ] == [ "a", "c", 7 ] → true
[ "a", "c", 7 ] == [ "a", "d", "f" ] → false
```

[]

arr[int] → obj 或 nil
arr[start,length] → an_array 或 nil
arr[range] → an_array 或 nil

1.8.

元素引用——分别返回下标 *int* 处的元素, 从下标 *start* 开始由 *length* 个元素组成的子数组, 或者返回由 *range* 指定的子数组。负数下标从数组尾部开始计算 (-1 是最后一个元素)。如果被选的第一个元素的下标大于数组的大小, 则返回 `nil`。如果开始下标等于数组的大小, 并且给出了 *length* 或 *range* 参数, 则返回一个空数组。等价于 `Array#slice`。

```
a = [ "a", "b", "c", "d", "e" ]
a[2] + a[0] + a[1] → "cab"
a[6] → nil
a[1..2] → [ "b", "c" ]
a[1..3] → [ "b", "c", "d" ]
a[4..7] → [ "e" ]
a[6..10] → nil
a[-3..3] → [ "c", "d", "e" ]
```

特殊情况

```
a[5] → nil
a[5..1] → []
a[5..10] → []
```

[]=

$$\begin{aligned} arr[int] &= obj \rightarrow obj \\ arr[start, length] &= obj \rightarrow obj \\ arr[range] &= obj \rightarrow obj \end{aligned}$$

元素赋值——设置下标 *int* 处的元素，替换从 *start* 下标开始由 *length* 个元素组成的子数组，或者替换由 *range* 指定的子数组。如果 *int* 大于数组的当前容积，那么数组会自动增长。负数 *int* 将从数组的末尾开始计数。如果 *length* 为 0 则插入元素。如果 *obj* 为 *nil*，则从 *arr* 中删除元素。如果 *obj* 是一个数组，只有一个下标的赋值形式将插入数组到 *arr* 中，而带长度参数的形式或以 *range* 为参数的形式将用数组内容替换 *arr* 中的给定元素。如果负数下标超过了数组的开头，则引发 *IndexError*。

参见 *Array#push* 和 *Array#unshift*。

| | |
|------------------------------------|--|
| <i>a</i> = <i>Array</i> .new | → [] |
| <i>a</i> [4] = "4"; | <i>a</i> → [nil, nil, nil, nil, "4"] |
| <i>a</i> [0] = [1, 2, 3]; | <i>a</i> → [[1, 2, 3], nil, nil, nil, "4"] |
| <i>a</i> [0..3] = ['a', 'b', 'c']; | <i>a</i> → ["a", "b", "c", nil, "4"] |
| <i>a</i> [1..2] = [1, 2]; | <i>a</i> → ["a", 1, 2, nil, "4"] |
| <i>a</i> [0..2] = "?"; | <i>a</i> → ["?", 2, nil, "4"] |
| <i>a</i> [0..2] = "A"; | <i>a</i> → ["A", "4"] |
| <i>a</i> [-1] = "Z"; | <i>a</i> → ["A", "Z"] |
| <i>a</i> [1..-1] = nil; | <i>a</i> → ["A"] |

arr | other_array → an_array

联合——返回由本数组和 *other_array* 联合而成的一个新数组，并去除重复的元素。元素比较的规则和 *hash* 键比较的规则相同。如果需要类似于集合的并集操作，请参见第 731 页的类 *Set*。

```
[ "a", "b", "c" ] | [ "c", "d", "a" ] → ["a", "b", "c", "d"]
```

assoc

arr.assoc(*obj*) → an_array 或 nil

通过使用 *obj*.== 对 *obj* 和子数组的第一个元素进行比较，在数组中搜索子数组元素。返回匹配的第一个子数组（也就是第一个 *associated* 的数组），如果没有匹配则返回 *nil*。参见 *Array#rassoc*。

```
s1 = [ "colors", "red", "blue", "green" ]
s2 = [ "letters", "a", "b", "c" ]
s3 = "foo"
a = [ s1, s2, s3 ]
a.assoc("letters") → ["letters", "a", "b", "c"]
a.assoc("foo") → nil
```

at

arr.at(*int*) → *obj* 或 nil

返回 *int* 下标处的元素。如果下标为负数，则从 *arr* 的末尾开始计算。如果下标超

出了范围则返回 nil。参见 Array#[] (Array#at 比 Array#[]稍微快点，因为它不接受 range 参数等等)。

```
a = [ "a", "b", "c", "d", "e" ]
a.at(0) → "a"
a.at(-1) → "e"
```

clear*arr.clear* → *arr*

删除 *arr* 的所有元素。

```
a = [ "a", "b", "c", "d", "e" ]
a.clear → []
```

collect!*arr.collect! {obj|block}* → *arr*

对 *arr* 的每个元素调用 *block*，并用 *block* 的返回值替换原元素。参见 Enumerable#collect。

```
a = [ "a", "b", "c", "d" ]
a.collect! { |x| x + "!" } → ["a!", "b!", "c!", "d!"]
a → ["a!", "b!", "c!", "d!"]
```

compact*arr.compact* → *an_array*

返回 *arr* 的一个拷贝，并删除其中的 nil 元素。

```
[ "a", nil, "b", nil, "c", nil ].compact → ["a", "b", "c"]
```

compact!*arr.compact!* → *arr* 或 nil

删除 *arr* 中的所有 nil 元素。如果没有改变内容，则返回 nil。

```
[ "a", nil, "b", nil, "c" ].compact! → ["a", "b", "c"]
[ "a", "b", "c" ].compact! → nil
```

concat*arr.concat (other_array)* → *arr*

将 *other_array* 的元素附加到 *arr*。

```
[ "a", "b" ].concat( [ "c", "d" ] ) → ["a", "b", "c", "d"]
```

delete*arr.delete(obj)* → *obj* 或 nil*arr.delete(obj){block}* → *obj* 或 nil

删除 *arr* 中与 *obj* 相等的元素。如果没有找到这样的元素则返回 nil。如果有相关联的 *block*，且没有找到相应的元素，则返回 *block* 的结果。

```
a = [ "a", "b", "b", "b", "c" ]
a.delete("b") → "b"
a → ["a", "c"]
a.delete("z") → nil
a.delete("z") { "not found" } → "not found"
```

delete_at*arr.delete_at(index) → obj 或 nil*

删除指定下标处的元素，并返回该元素，如果下标值超出了数组的下标范围，则返回 `nil`。参见 `Array#slice!`

```
a = %w( ant bat cat dog )
a.delete_at(2)      → "cat"
a                  → ["ant", "bat", "dog"]
a.delete_at(99)    → nil
```

delete_if*arr.delete_if {|item| block} → arr*

删除数组 `arr` 中那些能使 `block` 返回 `true` 的元素。

```
a = [ "a", "b", "c" ]
a.delete_if { |x| x >= "b" } → ["a"]
```

each*arr.each { |item| block} → arr*

对 `arr` 数组中的每个元素调用 `block`，并以该元素做 `block` 的参数。

```
a = [ "a", "b", "c" ]
a.each { |x| print x, "--"}
```

输出结果：

```
a -- b -- c --
```

each_index*arr.each_index { |index| block} → arr*

除了传递的是元素的下标而不是元素本身之外，其他和 `Array#each` 相同。

```
a = [ "a", "b", "c" ]
a.each_index { |x| print x, "--"}
```

输出结果：

```
0 -- 1 -- 2 --
```

empty?*arr.empty? → true 或 false*

如果数组 `arr` 没有元素，则返回 `true`。

```
[].empty?      → true
[ 1, 2, 3 ].empty? → false
```

eq?**arr.eq?(other) → true 或 false*

如果 `arr` 和 `other` 是同一个对象，或者如果 `other` 是一个与 `arr` 长度相同并且内容也相同的 `Array` 对象，则返回 `true`。数组中的元素使用 `Object#eq?` 进行比较。参见 `Array#<=>`。

```
[ "a", "b", "c" ].eq?([ "a", "b", "c" ]) → true
[ "a", "b", "c" ].eq?([ "a", "b" ])      → false
[ "a", "b", "c" ].eq?([ "b", "c", "d" ]) → false
```

fetch

arr.fetch(index) → obj
arr.fetch(index.default) → obj
arr.fetch(index){|lil block|} → obj

- 1.8. 返回下标 *index* 处的元素。如果下标值超出了数组的范围，那么第一种形式将会引发 (raise) 一个 IndexError 异常，第二种形式返回 *default*，而第三种形式返回以下标为参数调用 *block* 的结果。负数下标从数组的末尾开始计算。

```
a = [ 11, 22, 33, 44 ]
a.fetch(1)           → 22
a.fetch(-1)          → 44
a.fetch(-1,'cat')   → 44
a.fetch(4, 'cat')   → "cat"
a.fetch(4){|lil i*i|} → 16
```

fill

arr.fill(obj) → arr
arr.fill(obj, start <, length >) → arr
arr.fill(obj, range) → arr
arr.fill{|lil block|} → arr
arr.fill(start <,length >){|i| block} → arr
arr.fill(range){|lil block|} → arr

- 1.8. 前三种形式将 *arr* 中指定的元素（可以是整个数组）设置为 *obj*。如果 *start* 为 nil，则相当于 0。如果 *length* 为 nil，则相当于 *arr.length*。后三种形式使用 *block* 的值填充数组。传递给 *block* 的参数是被填充元素下标的绝对值。

```
a = [ "a", "b", "c", "d" ]
a.fill("x")           → ["x", "x", "x", "x"]
a.fill("z", 2, 2)     → ["x", "x", "z", "z"]
a.fill("y", 0..1)     → ["y", "y", "z", "z"]
a.fill{|lil i*i|}    → [0, 1, 4, 9]
a.fill(-3){|lil i+100|} → [0, 101, 102, 103]
```

first

arr.first → obj 或 nil
arr.first(count) → an_array

- 1.8. 返回数组的第一个元素，或前 *count* 个元素。如果数组为空，第一种形式返回 nil，而第二种形式返回一个空数组。

```
a = [ "q", "r", "s", "t" ]
a.first      → "q"
a.first(1)   → ["q"]
a.first(3)   → ["q", "r", "s"]
```

flatten

arr.flatten → an_array

- 返回一个新数组，该数组由原数组元素（递归）平铺而成。也就是说，如果某个数组元素是一个数组，则提取其中的元素并添加到新数组中。

```
s = [ 1, 2, 3 ]           → [1, 2, 3]
t = [ 4, 5, 6, [7, 8] ]   → [4, 5, 6, [7, 8]]
a = [ s, t, 9, 10 ]       → [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
a.flatten                 → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

flatten!*arr.flatten! → arr 或 nil*

除了直接修改原数组，和 `Array#flatten` 一样。如果未做任何改动则返回 `nil`（例如：`arr` 不包含子数组）。

```
a = [ 1, 2, [3, [4, 5] ] ]
a.flatten!    → [1, 2, 3, 4, 5]
a.flatten!    → nil
a            → [1, 2, 3, 4, 5]
```

include?*arr.include?(obj) → true 或 false*

如果所给对象出现在数组中（也就是说如果数组的任何对象 `== obj`），则返回 `true`，否则返回 `false`。

```
a = [ "a", "b", "c" ]
a.include?("b")  → true
a.include?("z")  → false
```

index*arr.index(obj) → int 或 nil*

返回数组 `arr` 中第一个等于 `obj` 的对象下标。如果没有匹配的对象，则返回 `nil`。

```
a = [ "a", "b", "c" ]
a.index("b")  → 1
a.index("z")  → nil
```

indexes*arr.indexes(i1, i2, ... iN) → an_array*

1.8 已废弃；使用 `Array#values_at`。

indices*arr.indices(i1, i2, ... iN) → an_array*

1.8 已废弃；使用 `Array#values_at`。

insert*arr.insert(index, <obj>+) → arr*

1.8 如果 `index` 是负数，则将给定的值插入到给定下标所对应的元素的前面。如果 `index` 为-1，则添加值到 `arr` 的末尾。其他情况下，将给定的值插入到给定下标对应的元素的后面。

```
a = %w{ a b c d }
a.insert(2, 99)      → ["a", "b", 99, "c", "d"]
a.insert(-2, 1, 2, 3) → ["a", "b", 99, "c", 1, 2, 3, "d"]
a.insert(-1, "e")    → ["a", "b", 99, "c", 1, 2, 3, "d", "e"]
```

表 27.1 Array#pack 的模板字符

| 指 令 含 义 | |
|----------|---------------------------------------|
| @ | 移到绝对位置 |
| A | ASCII 字符串（以空格补齐，计数符表示宽度） |
| a | ASCII 字符串（以 null 补齐，计数符表示宽度） |
| B | 比特串（降序位） |
| b | 比特串（升序位） |
| C | 无符号字符 |
| c | 有符号字符 |
| D, d | 本机格式的双精度浮点数 |
| E | 双精度浮点数，小端（little-endian）字节序 |
| e | 单精度浮点数，小端字节序 |
| F, f | 本机格式的单精度浮点数 |
| G | 双精度浮点数，网络（大端，big-endian）字节序 |
| g | 单精度浮点数，网络（大端）字节序 |
| H | 十六进制字符串（高四位在前） |
| h | 十六进制字符串（低四位在前） |
| I | 无符号整数 |
| i | 有符号整数 |
| L | 无符号长整数 |
| l | 有符号长整数 |
| M | Quoted-printable, MIME 编码（参见 RFC2045） |
| m | Base64 编码字符串 |
| N | 网络（大端）字节序的长整型 |
| n | 网络（大端）字节序的短整型 |
| P | 指向结构体（定长字符串）的指针 |
| p | 指向以 null 结尾的字符串 |
| 1.8 Q, q | 64-位整数 |
| S | 无符号短整型 |
| s | 短整型 |
| U | UTF-8 |
| u | UU-编码字符串 |
| V | 小端字节序的长整型 |
| v | 小端字节序的短整型 |
| 1.8 w | BER-压缩的整数 ² |
| X | 后退一个字节 |
| x | Null 字节 |
| Z | 和 A 一样 |

² BER 压缩整型包括多个 8 位字节，表示一个以 128 为基数的无符号整型，开头是最高有效数位（most significant digit），其后的数位尽可能少。除了最后一个字节，每个字节的第八位（高位）被设置为 1（自描述二进制数据表示，MacLeod）。

join*arr.join(separator=\$,) → str*

将数组中的每个元素连接成一个字符串，使用分隔符来分割每个元素，并返回该字符串。

```
[ "a", "b", "c" ].join      → "abc"
[ "a", "b", "c" ].join("-") → "a-b-c"
```

last

*arr.last → obj 或 nil**arr.last(count) → an_array*

1.8 返回数组 *arr* 的最后一个元素，或最后 *count* 个元素。如果数组为空，第一种形式返回 *nil*，第二种形式返回一个空的数组。

```
[ "w", "x", "y", "z" ].last      → "z"
[ "w", "x", "y", "z" ].last(1)   → ["z"]
[ "w", "x", "y", "z" ].last(3)   → ["x", "y", "z"]
```

length

arr.length → int

返回 *arr* 中元素的个数。参见 *Array#nitems*。

```
[ 1, nil, 3, nil, 5 ].length → 5
```

map!

arr.map! { | obj | block } → arr

与 *Array#collect!* 同义。

nitems

arr.nitems → int

返回 *arr* 中非 *nil* 元素的个数。参见 *Array#length*。

```
[ 1, nil, 3, nil, 5 ].nitems → 5
```

pack

arr.pack(template) → binary_string

1.8 根据模板 *template* 中的指令将 *arr* 的内容打包成二进制字节序（参见上一页的表 27.1）。指令 A, a 和 z 后可跟一个计数符，该计数符说明了生成字段的宽度。其余的指令也可以后跟一个计数符以表明被转换的数组元素的个数。如果计数符是星号 (*)，则剩余的所有数组元素将被转换。“sSiLL”指令中的任何一个都可后跟一个下画线 (_) 以使用底层平台的本地大小来表示指定的类型；否则使用平台无关的大小。模板字符串中的空格将被忽略。以#开头到下一个新行或字符串结尾的注释也被忽略。参见第 623 页的 *String#unpack*。

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
a.pack("A3A3A3") → "a\u0000b\u0000c\u0000"
a.pack("a3a3a3") → "a\u0000\u0000b\u0000\u0000c\u0000\u0000"
n.pack("ccc") → "ABC"
```

pop*arr.pop → obj 或 nil*

删除 *arr* 中的最后一个元素，并返回该元素，如果数组为空，则返回 *nil*。

```
a = [ "a", "m", "z" ]
a.pop      → "z"
a          → [ "a", "m" ]
```

push

arr.push(< obj >) → arr*

将给定参数添加到数组中。

```
a = [ "a", "b", "c" ]
a.push("d", "e", "f") → [ "a", "b", "c", "d", "e", "f" ]
```

rassoc

arr.rassoc(key) → an_array 或 nil

在数组中搜索本身是数组的元素。使用 `==` 比较 *key* 和数组元素的第二个元素。返回第一个匹配该 *key* 的数组元素。参见 `Array#assoc`。

```
a = [ [ 1, "one"], [2, "two"], [3, "three"], ["ii", "two"] ]
a.rassoc("two") → [2, "two"]
a.rassoc("four") → nil
```

reject!

arr.reject! { block } item → arr 或 nil

等价于 `Array#delete_if`，但如果没做任何改变，则返回 *nil*。参见 `Enumerable#reject`。

replace

arr.replace(other_array) → arr

使用 *other_array* 的内容替换 *arr* 的内容，并根据需要进行截断或扩展。

```
a = [ "a", "b", "c", "d", "e" ]
a.replace([ "x", "y", "z" ]) → [ "x", "y", "z" ]
a                         → [ "x", "y", "z" ]
```

reverse

arr.reverse → an_array

反序 *arr* 的元素，并返回该新数组。

```
[ "a", "b", "c" ].reverse → [ "c", "b", "a" ]
[ 1 ].reverse           → [ 1 ]
```

reverse!

arr.reverse! → arr

1.8.

反序 *arr* 的元素。

```
a = [ "a", "b", "c" ]
a.reverse!           → [ "c", "b", "a" ]
a                   → [ "c", "b", "a" ]
[ 1 ].reverse!     → [ 1 ]
```

reverse_each*arr.reverse_each { |item| block } → arr*

和 `Array#each` 一样，但是按逆序遍历 *arr*。

```
a = [ "a", "b", "c" ]
a.reverse_each { |x| print x, " " }
```

输出结果：

```
c b a
```

rindex

arr.rindex(obj) → int 或 nil

返回数组 *arr* 中最后一个对象，且该对象 $\equiv\ obj$ 。如果没有匹配，则 *nil* 返回 *nil*。

```
a = [ "a", "b", "b", "b", "c" ]
a.rindex("b") → 3
a.rindex("z") → nil
```

shift

arr.shift → obj 或 nil

返回 *arr* 的第一个元素，并将它从数组中删除（其他所有元素左移一个）。如果数组为空，则返回 *nil*。

```
args = [ "-m", "-q", "filename" ]
args.shift → "-m"
args → ["-q", "filename"]
```

size

arr.size → int

与 `Array#length` 同义。

slice

*arr.slice(int) → obj**arr.slice(start, length) → an_array**arr.slice(range) → an_array*

与 `Array#[]` 同义。

| | |
|---|-------------------|
| <i>a = ["a", "b", "c", "d", "e"]</i> | |
| <i>a.slice(2) + a.slice(0) + a.slice(1)</i> | → "cab" |
| <i>a.slice(6)</i> | → nil |
| <i>a.slice(1, 2)</i> | → ["b", "c"] |
| <i>a.slice(1..3)</i> | → ["b", "c", "d"] |
| <i>a.slice(4..7)</i> | → ["e"] |
| <i>a.slice(6..10)</i> | → nil |
| <i>a.slice(-3, 3)</i> | → ["c", "d", "e"] |
| <i># 特殊情况</i> | |
| <i>a.slice(5)</i> | → nil |
| <i>a.slice(5, 1)</i> | → [] |
| <i>a.slice(5..10)</i> | → [] |

slice!

arr.slice!(int) → obj 或 nil
arr.slice!(start, length) → an_array 或 nil
arr.slice!(range) → an_array 或 nil

删除给定索引（可后跟一个可选的长度参数）处的元素或给定 range 的元素。
 返回被删除的对象、子数组，如果下标超出数组范围，则返回 nil。等价于：

```
def slice!(*args)
  result = self[*args]
  self[*args] = nil
  result
end

a = [ "a", "b", "c" ]
a.slice!(1)      → "b"
a               → ["a", "c"]
a.slice!(-1)    → "c"
a               → ["a"]
a.slice!(100)   → nil
a               → ["a"]
```

sort

arr.sort → an_array
arr.sort {|a,b| block} → an_array

返回通过排序 arr 创建的一个新数组。排序所用的比较操作用 `<=>` 操作符或可选的代码 block 实现。block 实现 *a* 和 *b* 之间的比较，返回 -1, 0 或 +1。参见 Enumerable#sort_by。

```
a = [ "d", "a", "e", "c", "b" ]
a.sort           → ["a", "b", "c", "d", "e"]
a.sort { |x,y| y <=> x } → ["e", "d", "c", "b", "a"]
```

sort!

arr.sort! → arr
arr.sort! { |a,b| block } → arr

对数组 arr 进行排序（参见 Array#sort）。在排序的过程中 arr 将被冻结。

```
a = [ "d", "a", "e", "c", "b" ]
a.sort! → ["a", "b", "c", "d", "e"]
a       → ["a", "b", "c", "d", "e"]
```

to_a

arr.to_a → arr
array_subclass.to_a → array

如果 arr 是一个数组，则返回 arr。如果 arr 是数组的子类，则调用 to_ary，并用结果创建一个新数组对象。

to_ary

arr.to_ary → arr

返回 arr。

to_s *arr.to_s → str*

返回 *arr.join*.

```
[ "a", "e", "i", "o" ].to_s → "aeio"
```

transpose *arr.transpose → an_array*

1.8 假设 *arr* 是一个由数组组成的数组，并调换其行和列。

```
a = [[1,2], [3,4], [5,6]]  
a.transpose → [[1, 3, 5], [2, 4, 6]]
```

uniq *arr.uniq → an_array*

删除 *arr* 中的重复元素，并返回结果数组，重复元素是通过使用 *eql?* 和 *hash* 比较检测出来的。

```
a = [ "a", "a", "b", "b", "c" ]  
a.uniq → [ "a", "b", "c" ]
```

uniq! *arr.uniq! → arr 或 nil*

和 *Array#uniq* 一样，但是直接修改数组而不是创建一个新数组。如果没做任何改动（也就是说没有重复元素），则返回 *nil*。

```
a = [ "a", "a", "b", "b", "c" ]  
a.uniq! → [ "a", "b", "c" ]  
b = [ "a", "b", "c" ]  
b.uniq! → nil
```

unshift *arr.unshift(< obj >*) → arr*

添加对象到 *arr* 的首部。

```
a = [ "b", "c", "d" ]  
a.unshift("a") → [ "a", "b", "c", "d" ]  
a.unshift(1, 2) → [ 1, 2, "a", "b", "c", "d" ]
```

values_at *arr.values_at(< selector >*) → an_array*

1.8 返回一个用给定选择符从 *arr* 中选取的元素所组成的数组。选择符可以是整数下标，也可以是 *range*。

```
a = %w{ a b c d e f }  
a.values_at(1, 3, 5) → [ "b", "d", "f" ]  
a.values_at(1, 3, 5, 7) → [ "b", "d", "f", nil ]  
a.values_at(-1, -3, -5, -7) → [ "f", "d", "b", nil ]  
a.values_at(1..3, 2...5) → [ "b", "c", "d", "c", "d", "e" ]
```

Class **Bignum < Integer**

`Bignum` 对象容纳超出 `Fixnum` 范围的整数。当整数计算的结果超出 `Fixnum` 范围时，会自动创建一个 `Bignum` 对象。当包含 `Bignum` 对象的计算结果能被 `Fixnum` 表示时，结果将自动转换为 `Fixnum`。

为了实现位操作和`[]`，`Bignum` 被当作是一个以 2 的补码表示的无限长度的位串。

虽然 `Fixnum` 值是立即数，但 `Bignum` 对象不是——对象的引用可以进行赋值操作或当作参数进行传递，但对象本身不可以这样做。

实例方法

算术操作

可以对 `big` 执行各种各样的算术操作。

| | |
|----------------------------|------|
| <code>big + number</code> | 加 |
| <code>big - number</code> | 减 |
| <code>big * number</code> | 乘 |
| <code>big / number</code> | 除 |
| <code>big % number</code> | 取模 |
| <code>big ** number</code> | 指数操作 |
| <code>big -@</code> | 一元减 |

位操作

在 `Bignum` 的二进制表示上执行各种位操作。

| | |
|----------------------------------|----------------------------------|
| <code>~ big</code> | 按位取反 |
| <code>big number</code> | 位或 |
| <code>big & number</code> | 位与 |
| <code>big ^ number</code> | 位异或 |
| <code>big << number</code> | 左移 <code>number</code> 位 |
| <code>big >> number</code> | 右移 <code>number</code> 位 (带符号扩展) |

`<=>`

`big <=> number → -1, 0, +1`

比较——根据 `big` 小于、等于或大于 `number`，分别返回`-1`，`0` 或`+1`。这是 `Comparable` 中测试的基础。

`==`

`big == obj → true 或 false`

仅当 `obj` 和 `big` 有相同的值时才返回 `true`。可以和 `Bignum#eql?` 进行比较，它需要 `obj` 是 `Bignum`。

```
68719476736 == 68719476736.0 → true
```

[]

big[n] → 0, 1

位引用——返回 *big* (假设) 的二进制表示的第 *n* 位, 其中 *big[0]* 是最低位。

```
a = 9**15
50.downto(0) do |n|
  print a[n]
end
```

输出结果:

```
00010111011010000011100001111001010011100010111001
```

abs*big.abs → bignum*

返回 *big* 的绝对值。

```
1234567890987654321.abs → 1234567890987654321
-1234567890987654321.abs → 1234567890987654321
```

div*big.div(number) → other_number*1.8

与 *Bignum# /* 同义。

```
-1234567890987654321.div(13731) → -89910996357706
-1234567890987654321.div(13731.0) → -89910996357705.5
-1234567890987654321.div(-987654321) → 1249999989
```

divmod*big.divmod(number) → array*

参见第 565 页的 *Numeric#divmod*。

eql?*big.eql?(obj) → true 或 false*

仅当 *obj* 是一个和 *big* 有相同值的 *Bignum* 时返回 *true*。可以参考 *Bignum#==*, 它会执行类型转换。

```
68719476736.eql? 68719476736 → true
68719476736 == 68719476736 → true
68719476736.eql? 68719476736.0 → false
68719476736 == 68719476736.0 → true
```

modulo*big.modulo(number) → number*1.8

与 *Bignum#%* 同义。

quo*big.quo(number) → float*1.8

返回用 *number* 除 *big* 的浮点结果。

```
-1234567890987654321.quo(13731) → -89910996357705.5
-1234567890987654321.quo(13731.0) → -89910996357705.5
-1234567890987654321.div(-987654321) → 1249999989
```

remainder *big.remainder(number) → other_number*

1.8 返回 *big* 除以 *number* 的余数。

```
-1234567890987654321.remainder(13731) → -6966  
-1234567890987654321.remainder(13731.24) → -9906.22531493148
```

size *big.size → integer*

返回 *big* 的机器表示所用的字节数。

```
(256**10 - 1).size → 12  
(256**20 - 1).size → 20  
(256**40 - 1).size → 40
```

to_f *big.to_f → float*

把 *big* 转换为 Float。如果 *big* 不能用 Float 表示，则结果无限大。

to_s *big.to_s(base=10) → str*

1.8 返回包含 *big* 的 *base* (2 到 36) 进制表示的字符串。

```
12345654321.to_s → "12345654321"  
12345654321.to_s(2) → "10110111110110111011110000110001"  
12345654321.to_s(8) → "133766736061"  
12345654321.to_s(16) → "2dfdbbc31"  
12345654321.to_s(26) → "1dp1pc6d"  
78546939656932.to_s(36) → "rubyrules"
```

Class Binding < Object

类 `Binding` 的对象封装了代码在某个位置的执行上下文，并保留该上下文以备后用。在该上下文中可能被访问的变量、方法、`self` 的值以及可能的迭代 `block` 都将被保存下来。`Binding` 对象由 `Kernel#binding` 来创建，并可用在 `Kernel#set_trace_func` 的回调函数中。

绑定对象可以作为 `Kernel#eval` 方法的第二个参数，以建立该方法的执行环境。

```
class Demo
  def initialize(n)
    @secret = n
  end
  def get_binding
    return binding()
  end
end

k1 = Demo.new(99)
b1 = k1.get_binding
k2 = Demo.new(-3)
b2 = k2.get_binding

eval("@secret", b1)      → 99
eval("@secret", b2)      → -3
eval("@secret")          → nil
```

`Binding` 对象没有它自己的类定义的方法。

Class Class < Module

在 Ruby 中类是最常见的对象——每个类都是类 Class 的实例。

当定义一个新的类时（通常是使用 `class Name ... end`），Class 类型的一个对象将被创建，并赋值给一个常量（在这个例子中是 `Name`）。当调用 `Name.new` 创建一个新对象时，在运行新对象的 `initialize` 方法前，默认情况下 class 的实例方法 `new` 将运行，它会调用 `allocate` 来为对象分配内存。
1.8

类方法

inherited

`cls.inherited(sub_class)`

当创建 `cls` 的子类时，该方法将被 Ruby 调用。该方法的参数是将被创建的子类。

```
class Top
  def Top.inherited(sub)
    puts "New subclass: #{sub}"
  end
end
class Middle < Top
end
class Bottom < Middle
end
```

输出结果：

```
New subclass: Middle
New subclass: Bottom
```

new

`Class.new(super_class=Object) <{ block }> → cls`

1.8

根据给定的超类（如果没有给定参数，则使用 `Object`）创建一个新的匿名（未命名的）类。如果传入一个 block，则该 block 将作为新类的定义体。

```
p = lambda do
  def hello
    "Hello, Dave"
  end
end

FriendlyClass = Class.new(&p)
f = FriendlyClass.new
f.hello → "Hello, Dave"
```

实例方法

allocate

cls.allocate → *obj*

- 1.8 为 *cls* 类的新对象分配空间。返回的对象必须是 *cls* 的一个实例。一般来说，调用 `new` 创建对象和调用类方法 `allocate` 分配新对象，然后调用新对象的 `initialize` 方法是一样的。在普通程序中你无法重载 `allocate`，Ruby 将调用该方法而不会进入传统的方法分派（`dispatch`）。

```
class MyClass
  def MyClass.another_new(*args)
    o = allocate
    o.send(:initialize, *args)
    o
  end
  def initialize(a, b, c)
    @a, @b, @c = a, b, c
  end
end

mc = MyClass.another_new(4, 5, 6)
mc.inspect → "#<MyClass:0x1c9378 @c=6, @b=5, @a=4>"
```

new

cls.new(<args>)* → *obj*

- 调用 `allocate` 创建 *cls* 类的一个新对象，然后调用新创建的对象 `initialize` 方法，并以 *args* 作为参数。

superclass

cls.superclass → *super_class* 或 *nil*

返回 *cls* 的超类或 *nil*。

| | | |
|--------------------------------|---|---------------------|
| <code>Class.superclass</code> | → | <code>Module</code> |
| <code>Object.superclass</code> | → | <code>nil</code> |

Module Comparable

依赖: `<=>`

`Comparable` mixin 可被那些“其对象可以排序”的类使用。类必须定义 `<=>` 操作符，它比较接受者和另外一个对象，并根据接受者小于、等于或大于另一个对象返回 `-1`, `0` 或 `+1`。`Comparable` 使用 `<=>` 来实现传统的比较操作符 (`<`, `<=`, `==`, `>=` 和 `>`) 和 `between?` 方法。

```
class CompareOnSize
  include Comparable
  attr :str
  def <=>(other)
    str.length <=> other.str.length
  end
  def initialize(str)
    @str = str
  end
end

s1 = CompareOnSize.new("Z")
s2 = CompareOnSize.new([1,2])
s3 = CompareOnSize.new("XXX")

s1 < s2          → true
s2.between?(s1, s3) → true
s3.between?(s1, s2) → false
[s3, s2, s1].sort → ["Z", [1, 2], "XXX"]
```

实例方法

Comparisons

| |
|--|
| <i>obj < other_object</i> → true 或 false |
| <i>obj <= other_object</i> → true 或 false |
| <i>obj == other_object</i> → true 或 false |
| <i>obj >= other_object</i> → true 或 false |
| <i>obj > other_object</i> → true 或 false |

基于接受者的 `<=>` 方法比较两个对象。

between?

obj.between?(min, max) → true 或 false

如果 *obj <=> min* 小于 0 或者 *obj <=> max* 大于 0，则返回 `false`；否则返回 `true`。

| | |
|------------------------------|---------|
| 3.between?(1, 5) | → true |
| 6.between?(1, 5) | → false |
| 'cat'.between?('ant', 'dog') | → true |
| 'gnu'.between?('ant', 'dog') | → false |

Class Continuation < Object

Continuation 对象由 Kernel#callcc 产生。它包含返回地址和执行上下文，允许从程序的任何地方返回到 callcc block 的结尾处。Continuation 在一定程度上像 C 的 setjmp/longjmp 的结构化版本（由于它含有更多的状态，所以你可能认为它更接近线程）。

下面这个人为设计的例子允许最内层的循环提前结束处理。

```
callcc do |cont|
  for i in 0..4
    print "\n#{i}: "
    for j in i*5...(i+1)*5
      cont.call() if j == 7
      printf "%3d", j
    end
  end
  print "\n"
```

输出结果：

```
0: 0 1 2 3 4
1: 5 6
```

下面的例子显示了方法的调用栈被保存在了 continuation 中。

```
def strange
  callcc {|continuation| return continuation}
  print "Back in method, "
end
print "Before method. "
continuation = strange()
print "After method. "
continuation.call if continuation
```

输出结果：

```
Before method. After method. Back in method, After method.
```

实例方法

call

cont.call(< args >)*

调用 continuation。程序从 callcc block 的末尾继续执行。如果没有参数，则原来的 callcc 返回 nil。如果有一个参数，callcc 返回它。否则返回包含 args 的一个数组。

| | |
|---|-----------------------------|
| <pre>callcc { cont cont.call } callcc { cont cont.call 1 } callcc { cont cont.call 1, 2, 3 }</pre> | → nil
→ 1
→ [1, 2, 3] |
|---|-----------------------------|

Class Dir < Object

类 `Dir` 的对象是表示底层文件系统目录的目录流。它们提供了各种各样的方法来列出目录和它的内容。请参见第 465 页的 `File`。

这些例子中用的目录含有两个普通文件 (`config.h` 和 `main.rb`)，父目录 (...) 和目录本身 (.)。

Mixes in

Enumerable:

```
all?, any?, collect, detect, each_with_index, entries, find,
find_all, grep, include?, inject, map, max, member?, min, partition,
reject, select, sort, sort_by, to_a, zip
```

类方法

`[]` `Dir[glob_pattern] → array`

1.8 等价于调用 `Dir.glob(glob_pattern, 0)`。

`chdir` `Dir.chdir(<dir>) → 0`
`Dir.chdir(<dir>){|path|block} → obj`

改变进程的当前工作目录给给定的字符串。当不带参数调用此函数时，改变目录为环境变量 `HOME` 或 `LOGDIR` 的值。如果目标目录不存在，则引发 `SystemCallError`（也可能是 `Errno::ENOENT`）。

1.8 当给出 `block` 时，Ruby 会传递新的工作目录名字给 `block`，并以该目录为当前目录执行 `block`。当 `block` 退出时恢复原来的工作目录。`chdir` 的返回值为 `block` 的返回值。`Chdir` `block` 可以嵌套，但在多线程的程序中，如果一个线程试图打开其他线程，则已经打开 `chdir` 的 `block` 会出现错误。这是因为底层的操作系统只能理解这样的概念，即在任何时刻只有一个当前工作目录。

```
Dir.chdir("/var/log")
puts Dir.pwd
Dir.chdir("/tmp") do
  puts Dir.pwd
  Dir.chdir("/usr") do
    puts Dir.pwd
  end
  puts Dir.pwd
end
puts Dir.pwd
```

输出结果:

```
/var/log
/tmp
/usr
/tmp
/var/log
```

chroot**Dir.chroot(*dirname*) → 0**

改变进程的文件系统根目录。只有有权限的进程才能调用此方法。不是所有平台都支持此方法。在 Unix 系统上，参考 `chroot(2)` 以获得更多信息。

```
Dir.chdir("/production/secure/root")
Dir.chroot("/production/secure/root")    → 0
Dir.pwd                                → "/"
```

delete**Dir.delete(*dirname*) → 0**

删除给定的目录。如果目录不为空，则引发 `SystemCallError` 的子类的异常。

entries**Dir.entries(*dirname*) → array**

返回给定目录中所有文件名组成的数组。如果给定的目录不存在，则引发 `SystemCallError` 异常。

```
Dir.entries("testdir")      → [".", "..", "config.h", "main.rb"]
```

foreach**Dir.foreach(*dirname*) { |filename| *block* } → nil**

对给定目录中的每个项调用 *block*，并传递每个项的文件名作为 *block* 的参数。

```
Dir.foreach("testdir") { |x| puts "Got #{x}" }
```

输出结果:

```
Got .
Got ..
Got config.h
Got main.rb
```

getwd**Dir.getwd → *dirname***

返回包含进程的当前工作目录的规范路径的字符串。注意在某些操作系统上，这个名字可能不是传递给 `Dir.chdir` 的名字。例如在 OS X 上，`/tmp` 是一个符号连接文件。

```
Dir.chdir("/tmp")      → 0
Dir.getwd            → "/private/tmp"
```

| | |
|-------------------|--|
| <code>glob</code> | <code>Dir.glob(<i>glob_pattern</i>, <<i>flags</i>>) → array</code>
<code>Dir.glob(<i>glob_pattern</i>, <<i>flags</i>>) { filename <i>block</i> } → false</code> |
|-------------------|--|

1.8 扩展给定的模式 *glob_pattern*，并返回找到的文件名，返回的文件名或者作为 *array* 数组的元素，或者作为 *block* 的参数。注意这里的模式不是正则表达式（它更像 shell `glob`）。关于 *flags* 参数的意义请参考第 468 页的 `File.fnmatch`。大小写敏感性依赖于你的系统（所以 `File::FNM_CASEFOLD` 将被忽略）。模式中的原字符有：

- * 文件名中的任意字符序列：“*”会匹配所有文件，“c*”匹配所有以“c”开头的文件，“*c”匹配所有以“c”结尾的文件，而“*c*”匹配所有名字含有字母“c”的文件。
- ** 匹配 0 个或多个目录（所以“*/fred”匹配在当前目录或其子目录中名为“fred”的文件）。
- ? 匹配文件名中的任意一个字符。
- [*chars*] 匹配 *chars* 中的任意一个。如果 *chars* 的第一个字符是^，则匹配任意不在其中的字符。
- {*patt*, ...} 匹配花括号中给出的模式之一。这些模式可以含有其他原字符。

\ 去除下一个字符的特殊含义。

| | |
|---|--|
| <code>Dir.chdir("testdir")</code> | → 0 |
| <code>Dir["config.?"]</code> | → ["config.h"] |
| <code>Dir.glob("config.?")</code> | → ["config.h"] |
| <code>Dir.glob("*. [a-z] [a-z]*")</code> | → ["main.rb"] |
| <code>Dir.glob("*. [^r]*")</code> | → ["config.h"] |
| <code>Dir.glob("*. (rb,h)")</code> | → ["main.rb", "config.h"] |
| <code>Dir.glob("*)")</code> | → ["config.h", "main.rb"] |
| <code>Dir.glob("*, File::FNM_DOTMATCH)</code> | → [".", "..", "config.h", "main.rb"] |
|
 | |
| <code>Dir.chdir(..")</code> | → 0 |
| <code>Dir.glob("code/**/fib*.rb")</code> | → ["code/fib_up_to.rb",
"code/rdoc/fib_example.rb"] |
| <code>Dir.glob("**/rdoc/fib*.rb")</code> | → ["code/rdoc/fib_example.rb"] |

| | |
|--------------------|---|
| <code>mkdir</code> | <code>Dir.mkdir(<i>dirname</i> <, <i>permissions</i> >) → 0</code> |
|--------------------|---|

创建名为 *dirname* 的一个新目录，权限有可选的参数 *permission* 指定。权限可以被 `File.umask` 的值改变，在 Windows 上将被忽略。如果不能创建指定的目录，则引发 `SystemCallError` 异常。有关权限的讨论请参见第 465 页。

| | |
|------------------|---|
| <code>new</code> | <code>Dir.new(<i>dirname</i>) → <i>dir</i></code> |
|------------------|---|

为指定名字的目录返回一个新的目录对象。

| | |
|------|--|
| open | <code>Dir.open(<i>dirname</i>) → <i>dir</i></code>
<code>Dir.open(<i>dirname</i>) { <i>dir</i> <i>block</i> } → <i>obj</i></code> |
|------|--|

不带 `block` 调用时, `open` 和 `Dir.new` 同义。如果带有 `block`, 将传递 `dir` 作为
1.8. `block` 的参数。当 `block` 结束时将关闭目录, 且 `Dir.open` 返回 `block` 的值。

| | |
|-----|---------------------------------------|
| pwd | <code>Dir.pwd → <i>dirname</i></code> |
|-----|---------------------------------------|

与 `Dir.getwd` 同义。

| | |
|-------|--|
| rmdir | <code>Dir.rmdir(<i>dirname</i>) → 0</code> |
|-------|--|

与 `Dir.delete` 同义。

| | |
|--------|---|
| unlink | <code>Dir.unlink(<i>dirname</i>) → 0</code> |
|--------|---|

与 `Dir.delete` 同义。

实例方法

| | |
|-------|-------------------------------------|
| close | <code><i>dir</i>.close → nil</code> |
|-------|-------------------------------------|

关闭目录流。之后对 `dir` 的任何访问都会引发 `IOError` 异常。

```
d = Dir.new("testdir")
d.close → nil
```

| | |
|------|--|
| each | <code><i>dir</i>.each { <i>filename</i> <i>block</i> } → <i>dir</i></code> |
|------|--|

对目录中每个项都调用 `block`, 并以项中含有的文件名作为 `block` 的参数。

```
d = Dir.new("testdir")
d.each { |name| puts "Got #{name}" }
```

输出结果:

```
Got .
Got ..
Got config.h
Got main.rb
```

| | |
|------|---|
| path | <code><i>dir</i>.path → <i>dirname</i></code> |
|------|---|

1.8. 返回传递给 `dir` 的构造函数的路径参数。

```
d = Dir.new("..")
d.path → ".."
```

| | |
|-----|-----------------------------------|
| pos | <code><i>dir</i>.pos → int</code> |
|-----|-----------------------------------|

1.8. 与 `Dir#tell` 同义。

pos=*dir.pos(int) → int*

1.8.

除了返回位置参数之外，和 Dir#seek 同义。

```
d = Dir.new("testdir")      → #<Dir:0x1c9378>
d.read                      → "."
i = d.pos                   → 1
d.read                      → ".."
d.pos = i                  → 1
d.read                      → ".."
```

read

*dir.read → filename 或 nil*读取 *dir* 中的下一项，并作为字符串返回。如果到底流的末尾，则返回 nil。

```
d = Dir.new("testdir")
d.read → "."
d.read → ".."
d.read → "config.h"
```

rewind

*dir.rewind → dir*重新定位 *dir* 到第一项。

```
d = Dir.new("testdir")
d.read → "."
d.rewind → #<Dir:0x1c96e8>
d.read → ".."
```

seek

*dir.seek(int) → dir*定位到 *dir* 中指定的位置。*int* 必须是 Dir#tell 返回的值（它不必是项中的一个简单索引）。

```
d = Dir.new("testdir")      → #<Dir:0x1c9378>
d.read                      → "."
i = d.tell                   → 1
d.read                      → ".."
d.seek(i)                   → #<Dir:0x1c9378>
d.read                      → ".."
```

tell

*dir.tell → int*返回 *dir* 中的当前位置。参见 Dir#seek。

```
d = Dir.new("testdir")
d.tell → 1
d.read → "."
d.tell → 2
```

Module Enumerable

依赖于 `each`, `<=>`

`Enumerable` mixin 提供了具有遍历与搜索方法和排序能力的集合类。类必须提供 `each` 方法，该方法获得集合中连续的成员。如果使用了 `Enumerable#max`、`#min`、`#sort` 或者 `sort_by`，则集合中的对象必须实现一个有意义的`<=>`操作符，因为这些方法依赖于集合成员之间的顺序。

实例方法

all?

enum.all? <{ |obj| block} > → true 或 false

1.8

依次传递集合中的各个元素给 `block`。如果 `block` 一直没有返回 `false` 或 `nil`，则本方法返回 `true`。如果没有给定 `block`，那么 Ruby 自动创建一个隐式的`{|obj| obj}` `block`（也就是说，如果集合中没有为 `false` 或 `nil` 的成员，则 `all?` 返回 `true`）。

```
%w{ ant bear cat}.all? { |word| word.length >= 3 } → true
%w{ ant bear cat}.all? { |word| word.length >= 4 } → false
[ nil, true, 99 ].all? → false
```

any?

enum.any? <{ |obj| block} > → true 或 false

1.8

一次传递集合中的每个元素给 `block`。如果 `block` 返回了 `false` 或 `nil` 之外的值，则 `any?` 方法返回 `true`。如果没有 `block`，那么 Ruby 会自动创建 `{|obj| obj}` `block`（也就是说，如果集合中有一个成员不是 `false` 或 `nil`，则 `any?` 返回 `true`）。

```
%w{ ant bear cat}.any? { |word| word.length >= 3 } → true
%w{ ant bear cat}.any? { |word| word.length >= 4 } → true
[ nil, true, 99 ].any? → true
```

collect

enum.collect { |obj| block } → array

以 `enum` 中的每个元素为参数运行 `block`，并返回由 `block` 的运行结果组成的一个新数组。

```
(1..4).collect { |i| i*i } → [1, 4, 9, 16]
(1..4).collect { "cat" } → ["cat", "cat", "cat", "cat"]
```

detect

enum.detect(ifnone = nil) { |obj| block } → obj 或 nil

1.8

依次传递 `enum` 中的每个项给 `block`。返回使 `block` 不为 `false` 的第一个元素。除非给定了 `proc ifnone`，否则如果没有匹配的对象，则返回 `nil`，在给定了 `ifnone` 的情况下它将被调用，并返回其结果。

```
(1..10).detect { |i| i % 5 == 0 and i % 7 == 0 }      → nil
(1..100).detect { |i| i % 5 == 0 and i % 7 == 0 }     → 35
sorry = lambda { "not found" }
(1..10).detect(sorry) { |i| i > 50}                  → "not found"
```

each_with_index*enum.each_with_index { |obj, i| block } → enum*

对 *enum* 中的每项，以该项和其下标为参数调用 *block*。

```
hash = Hash.new
%w(cat dog wombat).each_with_index do |item, index|
  hash[item] = index
end
hash → {"cat"=>0, "wombat"=>2, "dog"=>1}
```

entries*enum.entries → array*

与 `Enumerable#to_a` 同义。

find*enum.find(ifnone = nil) { |obj| block } → obj 或 nil*

与 `Enumerable#detect` 同义。

find_all*enum.find_all { |obj| block } → array*

返回由 *enum* 中使 *block* 不为假的元素组成的数组（参见 `Enumerable#reject`）。

```
(1..10).find_all { |i| i % 3 == 0 } → [3, 6, 9]
```

grep*enum.grep(pattern) → array**enum.grep(pattern) { |obj| block } → array*

返回由 *enum* 中满足 `pattern==element` 的元素组成的数组。如果提供了 *block*，那么将传递每个匹配的元素给该 *block*，并将 *block* 返回的结果存储在输出数组中。

```
(1..100).grep 38..44 → [38, 39, 40, 41, 42, 43, 44]
c = IO.constants
c.grep(/SEEK/) → ["SEEK_CUR", "SEEK_SET", "SEEK_END"]
res = c.grep(/SEEK/) { |v| IO.const_get(v) }
res → [1, 0, 2]
```

include?*enum.include?(obj) → true 或 false*

如果 *enum* 中的任何成员等于 *obj*，则返回 `true`。使用 `==` 测试等价性。

```
IO.constants.include? "SEEK_SET" → true
IO.constants.include? "SEEK_NO_FURTHER" → false
```

| | |
|--------|---|
| inject | <i>enum.inject(initial) { memo, obj block } → obj</i>
<i>enum.inject { memo, obj block } → obj</i> |
|--------|---|

1.8 通过以计数器 (*memo*) 和 *enum* 中的元素为参数调用 *block* 来组合 *enum* 中的每个元素。在每一步中，*memo* 被设置为 *block* 的返回值。第一种形式允许你为 *memo* 提供一个初始值。第二种形式使用集合的第一个元素做 *memo* 的初始值（而且迭代时跳过该元素）。

```
# Sum some numbers
(5..10).inject(0) { |sum, n| sum + n } → 45
# Multiply some numbers
(5..10).inject(1) { |product, n| product * n } → 151200

# find the longest word
longest = %w{ cat sheep bear }.inject do |memo, word|
  memo.length > word.length ? memo : word
end
longest → "sheep"

# find the length of the longest word
longest = %w{ cat sheep bear }.inject(0) do |memo, word|
  memo >= word.length ? memo : word.length
end
longest → 5
```

| | |
|-----|---|
| map | <i>enum.map { obj block } → array</i> |
|-----|---|

与 `Enumerable#collect` 同义。

| | |
|-----|--|
| max | <i>enum.max → obj</i>
<i>enum.max { a,b block } → obj</i> |
|-----|--|

返回 *enum* 中值最大的对象。第一种形式假设 *enum* 中的所有对象都实现了 `<=>`；第二种形式使用 *block* 返回 `a<=>b`。

```
a = %w(albatross dog horse)
a.max → "horse"
a.max { |a,b| a.length <=> b.length } → "albatross"
```

| | |
|---------|---|
| member? | <i>enum.member?(obj) → true 或 false</i> |
|---------|---|

与 `Enumerable#include?` 同义。

| | |
|-----|--|
| min | <i>enum.min → obj</i>
<i>enum.min { a,b block } → obj</i> |
|-----|--|

返回 *enum* 中值最小的对象。第一种形式假设 *enum* 中的所有对象都实现了 `Comparable`；第二种形式使用 *block* 返回 `a<=>b`。

```
a = %w(albatross dog horse)
a.min → "albatross"
a.min { |a,b| a.length <=> b.length } → "dog"
```

partition *enum.partition { |obj| block } → [true_array, false_array]*

1.8 返回两个数组，第一个数组含有 *enum* 中使 *block* 为真的元素，第二个数组含有其余的元素。

```
(1..6).partition { |i| (i&1).zero? } → [[2, 4, 6], [1, 3, 5]]
```

reject *enum.reject { |obj| block } → array*

返回由 *enum* 中使 *block* 为假的元素组成的数组（参见 `Enumerable#find_all`）。

```
(1..10).reject { |i| i % 3 == 0 } → [1, 2, 4, 5, 7, 8, 10]
```

select *enum.select { |obj| block } → array*

与 `Enumerable#find_all` 同义。

sort *enum.sort → array*
enum.sort { |a, b| block } → array

对 *enum* 中的元素进行排序，并返回由排序后的元素组成的数组，排序可以用它自己的 `<=>` 方法，也可以使用给定 *block* 的结果。根据 *a* 和 *b* 的比较结果 *block* 应该返回 -1, 0 或 +1。在 Ruby 1.8 中，`Enumerable#sort_by` 实现了一个内建的 Schwarzenegger 变换，当计算和比较代价高昂时该变换很有用。

```
%w(rhea kea flea).sort → ["flea", "kea", "rhea"]
```

```
(1..10).sort { |a,b| b <=> a } → [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

sort_by *enum.sort_by { |obj| block } → array*

1.8 使用给定的 *block* 对 *enum* 中的元素计算的键值进行排序，并使用 *block* 的结果进行元素比较。

```
sorted = %w{ apple pear fig }.sort_by { |word| word.length}
```

```
sorted → ["fig", "pear", "apple"]
```

`.sort_by` 在内部会生成由原集合元素和映射值的元组所构成的一个数组。这使得 `sort_by` 在键集（keyset）比较简单时开销相当大。

```
require 'benchmark'
include Benchmark

a = (1..100000).map { rand(100000) }
bm(10) do |b|
  b.report("Sort") { a.sort }
  b.report("Sort by") { a.sort_by { |a| a } }
end
```

输出结果：

| | user | system | total | real |
|---------|----------|----------|----------|------------|
| Sort | 0.070000 | 0.010000 | 0.080000 | (0.085860) |
| Sort by | 1.580000 | 0.010000 | 1.590000 | (1.811626) |

然而，考虑一下比较操作复杂的情形。下面的代码使用基本 `sort` 方法根据文件的修改时间来对文件进行排序。

```
files = Dir["*"]
sorted = files.sort { |a,b| File.new(a).mtime <=> File.new(b).mtime}
sorted → ["mon", "tues", "wed", "thurs"]
```

这种排序效率不高：每次比较都会生成两个新的 `File` 对象。一个稍微好点的技术使用 `Kernel#test` 方法直接获得文件的修改时间。

```
files = Dir["*"]
sorted = files.sort do |a,b|
  test(?M, a) <=> test(?M, b)
end
sorted → ["mon", "tues", "wed", "thurs"]
```

这仍旧会生成许多不必要的 `Time` 对象。一个更有效的方法是在排序前缓存排序用的键（在这个例子中是修改时间）。Perl 用户以 Randal Schwartz 命名这种方法，称为 `Schwartzian` 变换。下面我们创建一个临时数组，其中的每个元素是由排序键和文件名组成的一个数组。然后对此数组进行排序，并从排序后的结果中提取出文件名。

```
sorted = Dir["*"].collect { |f|
  [test(?M, f), f]
}.sort.collect { |f| f[1] }
sorted → ["mon", "tues", "wed", "thurs"]
```

这正是 `sort_by` 在内部所做的。

```
sorted = Dir["*"].sort_by { |f| test(?M, f)}
sorted → ["mon", "tues", "wed", "thurs"]
```

`sort_by` 对多级排序也很有用。一个技巧是让 `block` 返回一个由比较键组成的数组，不过这需要数组是由一个元素一个元素进行比较的。例如，如果想先根据词的长度排序，再依据字母顺序排序，可以这样实现：

```
words = %w{ puma cat bass ant aardvark gnu fish }
sorted = words.sort_by { |w| [w.length, w] }
sorted → ["ant", "cat", "gnu", "bass", "fish", "puma", "aardvark"]
```

to_a*enum.to_a→array*

返回由 *enum* 中的项组成的数组。

| | | |
|---------------------------------|---|--------------------------------|
| (1..7).to_a | → | [1, 2, 3, 4, 5, 6, 7] |
| { 'a'=>1, 'b'=>2, 'c'=>3 }.to_a | → | [["a", 1], ["b", 2], ["c", 3]] |

zip

*enum.zip(<arg>+)→array**enum.zip(<arg>+){|arr|block}→nil*

把参数转换成数组，然后将 *enum* 中的元素和每个参数中对应的元素合并起来。结果是一个和 *enum* 具有相同数目元素的数组。其中的每个元素是一个含有 *n* 个元素的数组，这里的 *n* 比参数个数大 1。如果任意一个参数的元素个数小于 *enum* 的元素个数，则使用 nil。如果带有 block，则以每个输出的数组为参数调用它，否则返回由结果数组组成的数组。

```
a = [ 4, 5, 6 ]
b = [ 7, 8, 9 ]
```

| | | |
|---------------------|---|-----------------------------------|
| (1..3).zip(a, b) | → | [[1, 4, 7], [2, 5, 8], [3, 6, 9]] |
| "cat\ndog".zip([1]) | → | [["cat\n", 1], ["dog", nil]] |
| (1..3).zip | → | [[1], [2], [3]] |

Module Errno

Ruby 异常对象是 `Exception` 的子类。然而，操作系统通常使用单纯的整数来报告错误。Ruby 动态创建模块 `Errno` 以映射操作系统错误到 Ruby 类，且每个错误号都对应产生 `SystemCallError` 的一个子类。因为子类是在模块 `Errno` 中创建的，所以它的名字以 `Errno::` 打头。

```
Exception
  StandardError
  SystemCallError
  Errno::xxx
```

`Errno::` 类的名字依赖于 Ruby 运行的环境。在典型的 Unix 或 Windows 平台上，你会看到 Ruby 有 `Errno::EACCES`, `Errno::EAGAIN`, `Errno::EINTR` 等 `Errno` 类。

某个具体错误对应的整数操作系统错误号可以从类常量 `Errno::error::Errno` 中获得。

```
Errno::EACCES::Errno → 13
Errno::EAGAIN::Errno → 35
Errno::EINTR::Errno → 4
```

你平台上的操作系统错误的完整列表可以从 `Errno` 的常量中获得。在这个模块（包括已有异常的子类）中用户自定义的任意异常也必须定义一个 `Errno` 常量。

```
Errno.constants → E2BIG, EACCES, EADDRINUSE, EADDRNOTAVAIL,
                  EAFNOSUPPORT, EAGAIN, EALREADY, ...
```

- 1.8. Ruby 1.8 中，在 `rescue` 语句中使用 `Module#==` 来匹配异常。`SystemCallError` 类重载了 `==` 方法以基于 `Errno` 值进行比较。这样，如果两个不同的 `Errno` 类内部有相同的 `Errno` 值，那么 `rescue` 语句将把它们当作同一个异常对待。

Class **Exception** < Object

`Exception` 类的子类被用来在 `raise` 方法和 `begin/end block` 中的 `rescue` 语句间进行通信。`Exception` 对象含有关于异常的信息——它的类型（异常类的名字），一个可选的描述性的字符串和一个可选的追踪（trace）信息。

下一页的图 27.1 列出了标准库定义的异常。也请参考上页关于 `Errno` 的描述。

类方法

| | |
|---|---|
| <code>exception</code> | <code>Exception.exception(< message >) → exc</code> |
| 创建并返回一个新的异常对象，如果有 <code>message</code> 参数，则设置异常消息为 <code>message</code> 。 | |
| <code>new</code> | <code>Exception.new(< message >) → exc</code> |
| 创建并返回一个新的异常对象，如果有 <code>message</code> 参数，则设置异常消息为 <code>message</code> 。 | |

实例方法

| | |
|---|---|
| <code>backtrace</code> | <code>exc.backtrace → array</code> |
| 返回和异常相关的追踪信息。追踪信息是一个字符串数组，每个元素含有 <code>filename:line: in 'method'</code> 或 <code>filename:line</code> 。 | |
| <code>def a
 raise "boom"
end

def b
 a()
end

begin
 b()
rescue => detail
 print detail.backtrace.join("\n")
end</code> | |
| 输出结果： | |
| | <code>prog.rb:2:in `a'
prog.rb:6:in `b'
prog.rb:10</code> |

| | |
|---|---|
| <code>exception</code> | <code>exc.exception(< message >) → exc 或 exception</code> |
| 如果没有参数，则返回接受者。否则，创建一个新异常对象，它和接受者属于相同的类，但所含消息不同。 | |

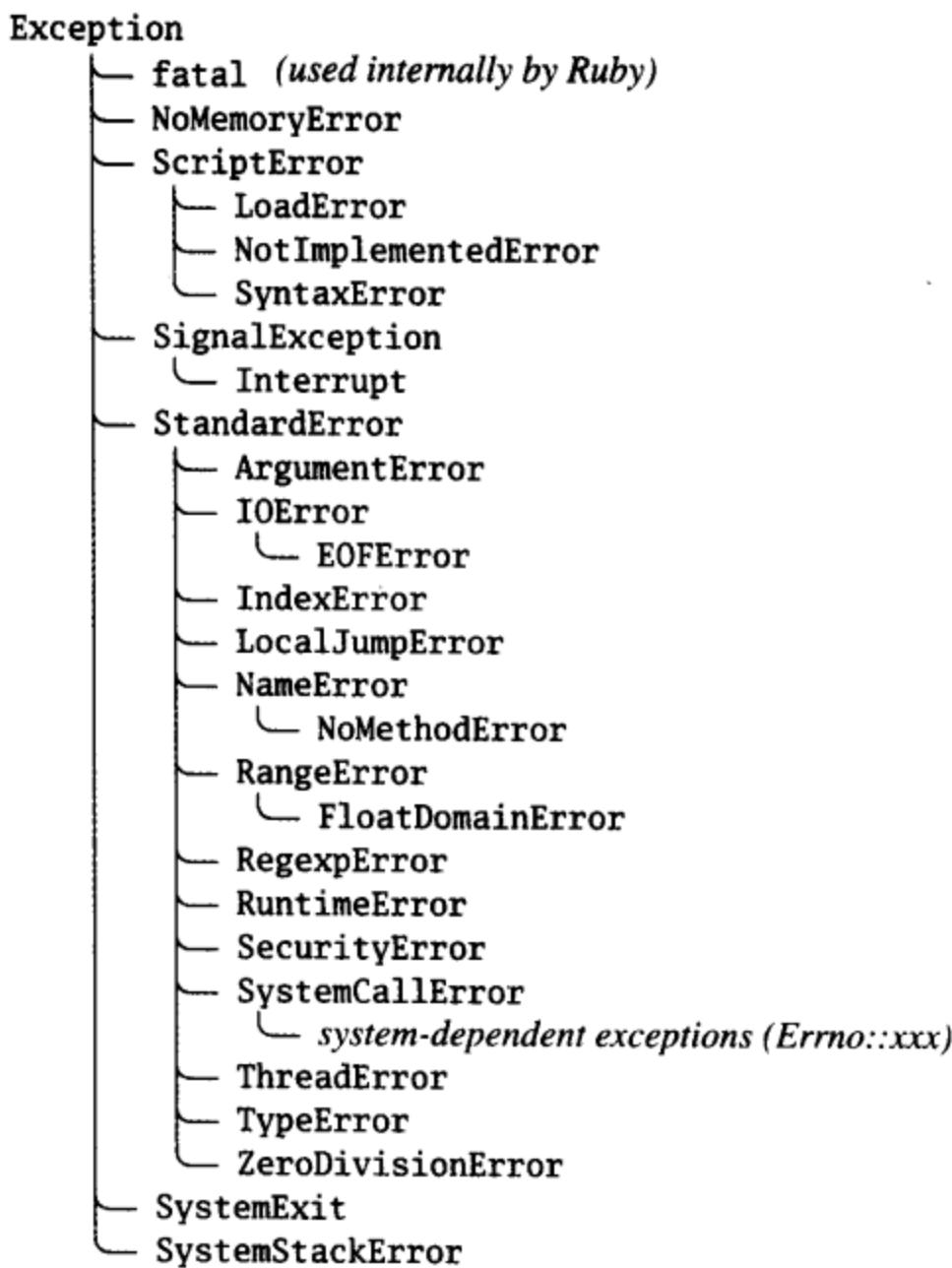


图 27.1 标准异常等级结构

| | |
|--|--|
| message | <i>exc.message</i> → <i>msg</i> |
| 返回与异常对应的消息。 | |
| set_backtrace | <i>exc.set_backtrace(array)</i> → <i>array</i> |
| 设置和 <i>exc</i> 相关联的追踪信息。参数必须是由符合 <code>Exception#backtrace</code> 所描述的格式的 <code>String</code> 对象组成的数组。 | |
| status | <i>exc.status</i> → <i>status</i> |
| 1.8 | (只用于 <code>SystemExit</code>) 返回和 <code>SystemExit</code> 异常相联系的退出状态。通常这个状态是由 <code>Kernel#exit</code> 设置的。 |

```

begin
  exit(99)
rescue SystemExit => e
  puts "Exit status is: #{e.status}"
end

```

输出结果:

```
Exit status is: 99
```

| | |
|----------|------------------------------------|
| success? | <i>exc.success? → true 或 false</i> |
|----------|------------------------------------|

18. (仅在 SystemExit 中提供) 如果退出状态为 nil 或 0, 则返回 true。

```

begin
  exit(99)
rescue SystemExit => e
  print "This program "
  if e.success?
    print "did"
  else
    print "did not"
  end
  puts " succeed"
end

```

输出结果:

```
This program did not succeed
```

| | |
|------|-----------------------|
| to_s | <i>Exc.to_s → msg</i> |
|------|-----------------------|

返回和异常相关的信息 (如果没有设置消息, 则返回异常的名字)。

```

begin
  raise "The message"
rescue Exception => e
  puts e.to_s
end

```

输出结果:

```
The message
```

| | |
|--------|-------------------------|
| to_str | <i>exc.to_str → msg</i> |
|--------|-------------------------|

返回和异常相关的信息 (如果没有设置消息, 则返回异常的名字)。实现 to_str 使异常具有类似字符串的行为。

Class FalseClass < Object

全局值 `false` 是 `FalseClass` 类的唯一实例，在逻辑表达式中它表示逻辑假。这个类提供了操作符以使 `false` 可以正确用在逻辑表达式中。

实例方法**&****`false & obj` → `false`**

与——返回 `false`。因为 `obj` 是方法调用的参数，所以总是会计算它的值——在这种情况下不遵守短路运算规则。换句话说，下面列出的使用`&&` 的代码不会调用 `lookup` 方法。

```
def lookup(val)
  puts "Looking up #{val}"
  return true
end
false && lookup("cat")
```

然而，如果在这段代码中使用`&`，则程序将如下所示：

```
false & lookup("cat")
```

输出结果：

```
Looking up cat
```

`false ^ obj` → `true` 或 `false`

异或——如果 `obj` 是 `nil` 或 `false`，则返回 `false`；否则返回 `true`。

`false | obj` → `true` 或 `false`

或——如果 `obj` 是 `nil` 或 `false`，则返回 `false`，否则返回 `true`。

Class **File** < **IO**

`File` 是可以被程序访问的任何文件对象的抽象，它和第 503 页描述的 `IO` 类紧密相关。`File` 包含了模块 `FileTest` 的方法作为其类方法，这时你可以编写（例如）这样的代码 `File.exist?("foo")`。

在本节中，权限位指和平台相关的表明文件访问权限的位。在基于 Unix 的系统上，权限位被认为是三个八位的字节，分别表示拥有者、组和其他人对文件的访问权限。对每个访问者，文件的访问权限可以设置为读、写或执行。

| Owner | Group | Other |
|-------|-------|-------|
| r w x | r w x | r w x |
| 4 2 1 | 4 2 1 | 4 2 1 |

权限位 0644（八进制）会被解释为允许文件的拥有者读写，而组成员和其他人只能读。高位可以用来表示文件的类型（普通文件、目录，管道、socket 等等）和其他特殊意义。如果权限是为目录设置的，那么执行位的含义变为：如果设置，则该目录可以被搜索。

每个文件有三个相关的时间。`atime` 是文件最后被访问的时间。`ctime` 是文件状态（不必是文件的内容）最后被改变的时间。最后 `mtime` 是文件数据最后被修改的时间。在 Ruby 中所有这些时间都作为 `Time` 的对象返回。

在非 POSIX 系统上，只能标志文件为只读或读/写。在这种情况下，其余的权限位将被合成为相似的典型值。例如在 Windows 上默认权限位是 0644，这意味着文件主可以读/写，而其他人只能读。唯一可做的改动是使文件只读，权限位对应的是 0444。

参见第 714 页的 `Pathname`。

类方法

atime

`File.atime(filename) → time`

返回一个包含所给文件最后访问时间的 `Time` 对象，如果文件未被访问过，则返回纪元时间。

`File.atime("testfile") → Thu Aug 26 22:36:07 CDT 2004`

basename

`File.basename(filename <, suffix >) → string`

1.8 返回给定文件名 `filename` 的最后一部分。如果有 `suffix` 参数，且它出现在 `filename` 的末尾，则它将被删除。通过使用 “`.*`” 可以去除任意扩展名。

```
File.basename("/home/gumby/work/ruby.rb")      → "ruby.rb"
File.basename("/home/gumby/work/ruby.rb", ".rb") → "ruby"
File.basename("/home/gumby/work/ruby.rb", ".*)") → "ruby"
```

blockdev?**File.blockdev?(filename) → true 或 false**

如果给定的文件是块设备，则返回 `true`，如果不是块设备或者如果操作系统不支持此特性，则返回 `false`。

```
File.blockdev?("testfile") → false
```

chardev?**File.chardev?(filename) → true 或 false**

如果给定的文件是字符设备，则返回 `true`，如果不是或者操作系统不支持此特性，则返回 `false`。

```
File.chardev?("/dev/tty") → true
```

chmod**File.chmod(permission <, filename >+) → int**

将给定文件的权限位改为 `permission`。实际的效果依赖于操作系统（参见本节开头）。在 Unix 系统上，若想获得更多细节可以参考 `chmod(2)`。返回被处理的文件的个数。

```
File.chmod(0644, "testfile", "out") → 2
```

chown**File.chown(owner, group <, filename >+) → int**

将给定文件的所有者和/或所属的组改为给定数字表示的所有者和组 ID。仅具有超级用户权限的进程可以改变文件的所有者。文件当前的拥有者可以改变文件的组为拥有者所属的任何组。如果拥有者或组 ID 为 `nil` 或 `-1`，则它们将被忽略。返回被处理的文件的个数。

```
File.chown(nil, 100, "testfile")
```

ctime**File.ctime(filename) → time**

返回一个含有给定文件最后状态修改时间的 `Time` 对象。

```
File.ctime("testfile") → Thu Aug 26 22:37:31 CDT 2004
```

delete**File.delete(<filename >+) → int**

删除给定的文件。返回被处理的文件的个数。参见 `Dir.rmdir`。

```
File.open("testrm", "w+")
File.delete("testrm") → 1
```

directory? File.directory?(*path*) → true 或 false

如果给定的文件是目录，则返回 true，否则返回 false。

File.directory?(".")

→ true

dirname File.dirname(*filename*) → *filename*

返回给定文件 *filename* 中除了最后一部分的其余部分。

File.dirname("/home/gumby/work/ruby.rb") → "/home/gumby/work"
File.dirname("ruby.rb") → ".."

executable? File.executable?(*filename*) → true 或 false

如果给定文件是可执行文件，则返回 true。用进程的有效拥有者来实现的测试。

File.executable?("testfile") → false

executable_real? File.executable_real?(*filename*) → true 或 false

和 File#executable? 一样，可使用进程的真实拥有者来进行测试。

exist? File.exist?(*filename*) → true 或 false

如果给定的文件或目录存在，则返回 true。

File.exist?("testfile") → true

exists? File.exists?(*filename*) → true 或 false

与 File.exist? 同义。

expand_path File.expand_path(*filename* <, *dirstring* >) → *filename*

转换路径名为绝对路径。除非有 *dirstring* 参数，相对路径是相对于进程的当前工作目录而言的，如果给定了 *dirstring*，那么它将作为路径的起点。给定的路径可能以 ~ 开头，它将被扩展为进程主人的主目录（环境变量 HOME 必须正确设置）。~user 扩展为给定用户的主目录。

File.expand_path("~testuser/bin") → "/Users/testuser/bin"
File.expand_path("../..bin", "/tmp/x") → "/bin"

extname File.extname(*path*) → *string*

1.8 返回文件的扩展名（文件名 *path* 中句点后面的部分）。

File.extname("test.rb") → ".rb"
File.extname("a/b/d/test.rb") → ".rb"
File.extname("test") → ""

表 27.2 匹配模式常量

| | |
|--------------|---|
| FNM_NOESCAPE | glob 中的反斜线不转义任何特殊字符，模式中的反斜线必须匹配文件名中的一个反斜线 |
| FNM_PATHNAME | 文件名中的斜线被认为是路径分隔符，因此必须和模式中的字符明确地匹配 |
| FNM_DOTMATCH | 如果没有指定本选项，那么开头包含一个点 “.” 的文件名只和含有一个点的模式匹配。开头的点是指文件的第一个字符是点或者（如果指定了 FNM_PATHNAME）后跟一个斜线 |
| FNM_CASEFOLD | 文件名匹配忽略大小写 |

file?`File.file?(filename) → true 或 false`

如果给定的文件是一个普通文件（不是设备文件、目录、管道、Socket 等等），则返回 `true`。

```
File.file?("testfile") → true
File.file?(".") → false
```

fnmatch`File.fnmatch(glob_pattern, path, <flags>) → true 或 false`

如果 `path` 和 `glob_pattern` 匹配，则返回 `true`。这里的模式不是正则表达式，不过它遵守类似 shell 文件名通配符的规则。因为 `fnmatch` 是由底层的操作系统实现的，因而它可能和 `Dir.glob` 的含义有所不同。一个 `glob_pattern` 可以含有如下元字符。

| | |
|-------------|---|
| ** | 递归式匹配子目录 |
| * | 匹配零个或多个字符 |
| ? | 匹配任意单个字符 |
| [charset] | 匹配给定字符集合中的任意一个字符。字符范围可以写为 <code>from-to</code> 形式。如果开头有个脱字符 (^)，则表示非此集合中的任意字符 |
| \ | 转义下一个字符的特殊含义 |

`flags` 可以是本页列出的 `FNM_xxx` 参数的位或。参见第 451 页的 `Dir.glob`。

```
File.fnmatch('cat',      'cat')      → true
File.fnmatch('cat',      'category') → false
File.fnmatch('c(at,ub)s', 'cats')    → false
File.fnmatch('c(at,ub)s', 'cubs')    → false
File.fnmatch('c(at,ub)s', 'cat')     → false
File.fnmatch('c?t',       'cat')     → true
File.fnmatch('c\?t',       'cat')     → false
```

```

File.fnmatch('c??t',      'cat')           → false
File.fnmatch('c*',        'cats')          → true
File.fnmatch('c/**/t',    'c/a/b/c/t') → true
File.fnmatch('c*t',       'cat')           → true
File.fnmatch('c\at',      'cat')           → true
File.fnmatch('c\at',      'cat', File::FNM_NOESCAPE) → false
File.fnmatch('a?b',        'a/b')           → true
File.fnmatch('a?b',        'a/b', File::FNM_PATHNAME) → false

File.fnmatch('*',         '.profile')       → false
File.fnmatch('*',         '.profile', File::FNM_DOTMATCH) → true
File.fnmatch('*',         'dave/.profile') → true
File.fnmatch('*',         'dave/.profile', File::FNM_DOTMATCH) → true
File.fnmatch('*',         'dave/.profile', File::FNM_PATHNAME) → false
File.fnmatch('*/*',       'dave/.profile', File::FNM_PATHNAME) → false
STRICT = File::FNM_PATHNAME | File::FNM_DOTMATCH
File.fnmatch('*/*',       'dave/.profile', STRICT)           → true

```

fnmatch? `File.fnmatch?(glob_pattern, path, <flags>)` → (true 或 false)

1.8 与 `File#fnmatch` 同义。

ftype `File.ftype(filename)` → filetype

确定给定文件的类型。返回的字符串可以是 `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, `socket` 或 `unknown`。

```

File.ftype("testfile") → "file"
File.ftype("/dev/tty") → "characterSpecial"
system("mkfifo wibble") → true
File.ftype("wibble") → "fifo"

```

grpowned? `File.grpowned?(filename)` → true 或 false

如果进程的有效组 ID 和给定文件的组 ID 相同，则返回 `true`。在 Windows 上返回 `false`。

```
File.grpowned?("/etc/passwd") → false
```

join `File.join(<string>+)` → filename

使用 `File::SEPARATOR` 连接字符串，并返回该新结果字符串。下页的表 27.3 列出了各种各样的分隔符。

```
File.join("usr", "mail", "gumby") → "usr/mail/gumby"
```

lchmod `File.lchmod(permission, <filename>+)` → 0

1.8 等价于 `File.chmod`，但不跟随符号链接（所以它会改变符号连接文件本身的权限，而不是链接所指向的文件的权限）。不是在所有平台上可用。

表 27.3 路径分隔符常量（和平台相关）

| | |
|----------------|-------------------------|
| ALT_SEPARATOR | 其他路径分割符 |
| PATH_SEPARATOR | 在搜索路径中文件名分隔符（例如 : 或者 ;） |
| SEPARATOR | 文件名中路径分隔符（例如 \ 或 /） |
| Separator | SEPARATOR 的别名 |

| | |
|--------|--|
| lchown | File.lchown(<i>owner</i> , <i>group</i> , < <i>filename</i> >+) → 0 |
|--------|--|

1.8 等价于 File.chown，但是不跟随符号连接（所以它会改变链接文件本身的拥有者，而不是链接所指目标文件）。常常不可用。

| | |
|------|--|
| link | File.link(<i>oldname</i> , <i>newname</i>) → 0 |
|------|--|

使用硬链接为已有文件创建一个新的名字。如果新名字 *newname* 已经存在，则不会覆盖它（这种情况下 link 会引发一个 SystemCallError 的子类的一个错误）。不是在所有平台上可用。

```
File.link("testfile", "testfile.2")      → 0
f = File.open("testfile.2")
f.gets                                     → "This is line one\n"
File.delete("testfile.2")
```

| | |
|-------|--------------------------------------|
| lstat | File.lstat(<i>filename</i>) → stat |
|-------|--------------------------------------|

以 File::Stat 类型的对象返回文件 *file* 的状态信息。和 IO#stat 一样（参见第 513 页），但是不跟随符号链接。相反它报告链接文件本身的信息。

```
File.symlink("testfile", "link2test") → 0
File.stat("testfile").size           → 66
File.lstat("link2test").size        → 8
File.stat("link2test").size         → 66
```

| | |
|-------|--------------------------------------|
| mtime | File.mtime(<i>filename</i>) → time |
|-------|--------------------------------------|

返回包含文件被修改时间的一个 Time 对象。

```
File.mtime("testfile")      → Tue May 17 16:26:41 CDT 2005
File.mtime("/tmp")          → Wed May 18 09:07:03 CDT 2005
```

| | |
|-----|--|
| new | File.new(<i>filename</i> , <i>modestring</i> ="r") → file
File.new(<i>filename</i> <, <i>modenum</i> <, <i>permission</i> >>) → file
File.new(<i>fd</i> <, <i>modenum</i> <, <i>permission</i> >>) → file |
|-----|--|

根据模式 *modestring*（默认为 r）打开名为 *filename* 的文件（或关联已经打开的文件描述符 *fd*），并返回一个新的 File 对象。第 504 页的表 27.6 对模式字符串进行了描述。文件模式也可以通过把第 472 页的表 27.4 描述的标志“或”在一起形成的

`Fixnum` 来指定。可选的权限位可以由 `permission` 指定。这些模式和权限位是平台相关的；在 Unix 系统上详情可以参考 `open(2)`。

```
f = File.new("testfile", "r")
f = File.new("newfile", "w+")
f = File.new("newfile", File::CREAT|File::TRUNC|File::RDWR, 0644)
```

| | |
|-------------------|--|
| <code>open</code> | <code>File.open(filename, modestring="r") → file</code>
<code>File.open(filename <, modenum <, permission >>) → file</code>
<code>File.open(fd <, modenum <, permission >>) → file</code>
<code>File.open(filename, modestring="r") { file block } → obj</code>
<code>File.open(filename <, modenum <, permission >>) { file block } → obj</code>
<code>File.open(fd <, modenum <, permission >>) { file block } → obj</code> |
|-------------------|--|

如果没有关联的 `block`，那么 `open` 和 `File.new` 同义。如果提供了代码 `block`，它将传递 `file` 作为参数，并且当 `block` 执行结束时文件将被自动关闭。在这种情况下，`File.open` 返回 `block` 的值。

| | |
|---------------------|---|
| <code>owned?</code> | <code>File.owned?(filename) → true 或 false</code> |
|---------------------|---|

如果进程的有效用户 ID 和文件的拥有者相同，则返回 `true`。

```
File.owned?("/etc/passwd") → false
```

| | |
|--------------------|--|
| <code>pipe?</code> | <code>File.pipe?(filename) → true 或 false</code> |
|--------------------|--|

如果操作系统支持管道且给定的文件是管道文件，则返回 `true`，否则返回 `false`。

```
File.pipe?("testfile") → false
```

| | |
|------------------------|--|
| <code>readable?</code> | <code>File.readable?(filename) → true 或 false</code> |
|------------------------|--|

如果本进程的有效用户 ID 可以读给定的文件，则返回 `true`。

```
File.readable?("testfile") → true
```

| | |
|-----------------------------|---|
| <code>readable_real?</code> | <code>File.readable_real?(filename) → true 或 false</code> |
|-----------------------------|---|

如果本进程的真实用户 ID 可以读给定的文件，则返回 `true`。

```
File.readable_real?("testfile") → true
```

| | |
|-----------------------|---|
| <code>readlink</code> | <code>File.readlink(filename) → filename</code> |
|-----------------------|---|

以字符串返回给定的符号链接。不是在所有平台上可用。

```
File.symlink("testfile", "link2test") → 0
File.readlink("link2test") → "testfile"
```

表 27.4 open 模式常量

| | |
|----------|---|
| APPEND | 以追加模式打开文件；对文件的所有写操作都从文件末尾开始 |
| CREAT | 打开文件时如果文件不存在，则创建文件 |
| EXCL | 和 CREATE 合用时，如果文件已存在，则 open 将失败 |
| NOCTTY | 当打开终端设备时（参见第 510 页的 IO#isatty），不允许它变为控制终端 |
| NONBLOCK | 以非阻塞模式打开文件 |
| RDONLY | 以只读模式打开文件 |
| RDWR | 以读写模式打开文件 |
| TRUNC | 如果文件已经存在，则截断文件使其长度为 0 |
| WRONLY | 以只写模式打开文件 |

rename `File.rename(oldname, newname) → 0`

重命名给定的文件或者目录为新名字。如果不能成功重命名，则引发 `SystemCallError`。

```
File.rename("afile", "afile.bak") → 0
```

setgid? `File.setgid?(filename) → true 或 false`

如果设置了给定文件的 set-group-id 位，则返回 `true`，如果没有设置或者操作系统不支持这种特性，则返回 `false`。

```
File.setgid?("/usr/sbin/lpc") → false
```

setuid? `File.setuid?(filename) → true 或 false`

如果设置了给定文件的 set-group-id 位，则返回 `true`，如果没有设置或者操作系统不支持这种特性，则返回 `false`。

```
File.setuid?("/bin/su") → false
```

size `File.size(filename) → int`

返回文件的字节大小。

```
File.size("testfile") → 66
```

size? `File.size?(filename) → int 或 nil`

如果给定文件的长度为 0，则返回 `nil`，否则返回其大小。在测试的条件语句中很有用。

```
File.size?("testfile") → 66
File.size?("/dev/zero") → nil
```

socket?**File.socket?(filename) → true 或 false**

如果给定文件是 socket 文件，则返回 true，如果不是或者操作系统不支持这种特性，则返回 false。

split**File.split(filename) → array**

把给定的字符串分割成目录和文件，并返回包含它们的具有两个元素的数组。

参见 `File.dirname` 和 `File.basename`。

```
File.split("/home/gumby/.profile") → ["/home/gumby", ".profile"]
File.split("ruby.rb") → [".", "ruby.rb"]
```

stat**File.stat(filename) → stat**

返回给定文件的一个 `File::Stat` 对象（参见第 477 页的 `File::Stat`。）

```
stat = File.stat("testfile")
stat.mtime → Thu Aug 26 12:33:23 CDT 2004
stat.blockdev? → false
stat.ftype → "file"
```

sticky?**File.sticky?(filename) → true 或 false**

如果给定文件设置了粘滞位，则返回 true，如果没有设置或者操作系统不支持这种特性，则返回 false。

symlink**File.symlink(oldname, newname) → 0 或 nil**

为文件 `oldname` 创建一个名为 `newname` 的符号链接。在不支持符号链接的平台上将返回 nil。

```
File.symlink("testfile", "link2test") → 0
```

symlink?**File.symlink?(filename) → true 或 false**

如果给定文件是符号链接文件，则返回 true，如果不是或者操作系统不支持这种特性，则返回 false。

```
File.symlink?("testfile", "link2test") → 0
File.symlink?("link2test") → true
```

truncate**File.truncate(filename, int) → 0**

截断文件 `filename`，使其长度至多 `int` 字节。不是在所有平台上可用。

```
f = File.new("out", "w")
f.write("1234567890") → 10
f.close → nil
File.truncate("out", 5) → 0
File.size("out") → 5
```

umask**File.umask(< int >) → int**

返回当前进程的 umask 值。如果有参数，则设置 umask 值为参数值，并返回原来的值。Umask 值将从默认权限中移除，所以 0222 umask 值使得文件对所有人只读。参见第 465 页关于权限的讨论。

```
File.umask(0006) → 18
File.umask → 6
```

unlink**File.unlink(< filename >+) → int**

与 File.delete 同义。参见 Dir.rmdir。

```
File.open("testrm", "w+") {} → nil
File.unlink("testrm") → 1
```

utime**File.utime(accesstime, modtime <, filename >+) → int**

改变多个文件的访问和修改时间。时间必须是类 Time 的实例或表示自纪元开始的秒数的数字。返回被处理的文件个数。不是在所有平台上可用。

```
File.utime(0, 0, "testfile") → 1
File.mtime("testfile") → Wed Dec 31 18:00:00 CST 1969
File.utime(0, Time.now, "testfile") → 1
File.mtime("testfile") → Thu Aug 26 22:37:33 CDT 2004
```

writable?**File.writable?(filename) → true 或 false**

如果给定的文件可以被进程的有效用户 ID 写，则返回 true。

```
File.writable?("/etc/passwd") → false
File.writable?("testfile") → true
```

writable_real?**File.writable_real?(filename) → true 或 false**

如果给定的文件可以被进程的真实用户 ID 写，则返回 true。

zero?**File.zero?(filename) → true 或 false**

如果给定的文件长度为 0，则返回 true，否则返回 false。

```
File.zero?("testfile") → false
File.open("zerosize", "w") {}
File.zero?("zerosize") → true
```

实例方法

| | |
|---|--|
| atime | <i>file.atime</i> → <i>time</i> |
| 返回含有文件 <i>file</i> 最后访问时间的一个 <i>Time</i> 对象，如果文件未被访问过，则返回纪元时间。 | |
| | <code>File.new("testfile").atime → Wed Dec 31 18:00:00 CST 1969</code> |
| <hr/> | |
| chmod | <i>file.chmod(permission)</i> → 0 |
| 修改文件 <i>file</i> 的权限位为 <i>permission</i> 表示的位模式。其实际的效果与平台相关；在 Unix 系统上，详细介绍参见 <code>chmod(2)</code> 。此方法跟随符号链接。参见第 465 页中关于权限的讨论。另参见 <code>File#lchmod</code> 。 | |
| | <code>f = File.new("out", "w");
f.chmod(0644) → 0</code> |
| <hr/> | |
| chown | <i>file.chown(owner, group)</i> → 0 |
| 改变文件 <i>file</i> 的拥有者和组位给定的数字拥有者和组 ID。只有具有超级用户权限的进程可以改变文件的拥有者。文件的当前拥有者可以改变文件的组为拥有者所属的组之一。本方法将忽略为 nil 或 -1 拥有者或组 id。而且本方法跟随符号链接。参见 <code>File#lchown</code> 。 | |
| | <code>File.new("testfile").chown(502, 1000)</code> |
| <hr/> | |
| ctime | <i>file.ctime</i> → <i>time</i> |
| 返回含有文件 <i>file</i> 状态最后一次被修改时间的 <i>Time</i> 对象。 | |
| | <code>File.new("testfile").ctime → Thu Aug 26 22:37:33 CDT 2004</code> |
| <hr/> | |
| flock | <i>file.flock(locking_constant)</i> → 0 或 false |
| 根据 <i>locking_constant</i> （由下一页的表 27.5 中的值组成的逻辑或）锁住或开锁文件。如果指定了 <code>File::LOCK_NB</code> ，则返回 <code>false</code> ，否则操作将阻塞。不是在所有平台上可用。 | |
| | <code>File.new("testfile").flock(File::LOCK_UN) → 0</code> |
| <hr/> | |
| lchmod | <i>file.lchmod(permission)</i> → 0 |
| 1.8 | 等价于 <code>File#chmod</code> ，但不跟随符号链接（因此它将会改变链接文件本身的权限而不是被链接的文件的权限）。不是在所有平台上可用。 |
| <hr/> | |
| lchown | <i>file.lchown(owner, group)</i> → 0 |
| 1.8 | 等价于 <code>File#chown</code> ，但不跟随符号链接（因此它将会改变链接文件本身的拥有者，而不是被链接的文件的拥有者）。不是在所有平台上可用。 |

表 27.5 锁模式常量

| | |
|---------|-----------------------------|
| LOCK_EX | 排他锁。在一个时刻仅有一个进程可以获得给定文件的排他锁 |
| LOCK_NB | 当加锁时不阻塞。可以使用逻辑或与其他选项组合 |
| LOCK_SH | 共享锁。在一个时刻可以有多个进程获得给定文件的共享锁 |
| LOCK_UN | 解锁 |

| | |
|--|------------------------------------|
| lstat | <i>file.lstat</i> → <i>stat</i> |
| 与 IO#stat一样，但不跟随符号链接而是报告链接本身的状态。 | |
| <pre>File.symlink("testfile", "link2test") → 0 File.stat("testfile").size → 66 f = File.new("link2test") f.lstat.size → 8 f.stat.size → 66</pre> | |
| mtime | <i>file.mtime</i> → <i>time</i> |
| 返回包含文件 <i>file</i> 修改时间的一个 Time 对象。 | |
| <pre>File.new("testfile").mtime → Thu Aug 26 22:37:33 CDT 2004</pre> | |
| path | <i>file.path</i> → <i>filename</i> |
| 返回创建文件 <i>file</i> 时的路径名字符串。不对名字进行标准化。 | |
| <pre>File.new("testfile").path → "testfile" File.new("/tmp/..tmp/xxx", "w").path → "/tmp/..tmp/xxx"</pre> | |
| truncate | <i>file.truncate(int)</i> → 0 |

截断文件 *filename*，使其长度至多 *int* 字节。文件必须以写模式被打开。不是在所有平台上可用。

```
f = File.new("out", "w")
f.syswrite("1234567890") → 10
f.truncate(5) → 0
f.close() → nil
File.size("out") → 5
```

Class **File::Stat** < Object

类 `File::Stat` 对象封装了 `File` 对象的普通状态信息。信息是在 `File::Stat` 对象被创建时记录的；且不受以后对文件的修改的影响。`IO#stat`、`File.stat`、`File#lstat` 和 `File.lstat` 会返回 `File::Stat` 对象。这些方法中许多方法会返回平台相关的值，而且返回的值并不是在所有的平台上都有意义。参见第 531 页的 `Kerenl#test`。

Mixes in

Comparable:

```
<, <=, ==, >=, >, between?
```

实例方法

<=>

`statfile <=> other_stat → -1, 0, 1`

比较 `File::Stat` 对象，比较基于它们各自的修改时间。

```
f1 = File.new("f1", "w")
sleep 1
f2 = File.new("f2", "w")
f1.stat <=> f2.stat      → -1
# Methods in Comparable are also available
f1.stat > f2.stat        → false
f1.stat < f2.stat        → true
```

atime

`statfile.atime → time`

返回含有 `statfile` 最后访问时间的一个 `Time` 对象，如果文件未被访问过，则返回纪元时间。

```
File.stat("testfile").atime      → Wed Dec 31 18:00:00 CST 1969
File.stat("testfile").atime.to_i  → 0
```

blksize

`statfile.blksize → int`

返回本地文件系统的块大小。在不支持这类信息的平台上将返回 `nil`。

```
File.stat("testfile").blksize → 4096
```

blockdev?

`statfile.blockdev? → true 或 false`

如果文件是块设备，则返回 `true`。如果不是或操作系统不支持这种特性，则返回 `false`。

```
File.stat("testfile").blockdev?    → false
File.stat("/dev/disk0").blockdev?  → true
```

| | |
|--|---|
| blocks | <i>statfile.blocks</i> → <i>int</i> |
| 返回为文件分配的本地文件系统的块的个数，如果底层操作系统不支持这种特性，则返回 nil。 | |
| File.stat("testfile").blocks | → 8 |
| chardev? | |
| 如果文件是字符设备，则返回 true。如果不是或操作系统不支持这种特性，则返回 false。 | |
| File.stat("/dev/tty").chardev? | → true |
| File.stat("testfile").chardev? | → false |
| ctime | <i>statfile.ctime</i> → <i>time</i> |
| 返回含有 <i>statfile</i> 相关联文件的修改状态的一个 Time 对象。 | |
| File.stat("testfile").ctime | → Thu Aug 26 22:37:33 CDT 2004 |
| dev | <i>statfile.dev</i> → <i>int</i> |
| 返回表示保存 <i>statfile</i> 的设备的一个整数。设备整数中的位常常由主设备信息和次设备信息组合而成。 | |
| File.stat("testfile").dev | → 234881033 |
| "%x" % File.stat("testfile").dev | → "e000009" |
| dev_major | <i>statfile.dev_major</i> → <i>int</i> |
| <u>1.8.</u> | 返回 File::Stat#dev 的主设备部分。如果操作系统不支持这种特性，则返回 nil。 |
| File.stat("testfile").dev_major | → 14 |
| dev_minor | <i>statfile.dev_minor</i> → <i>int</i> |
| <u>1.8.</u> | 返回 File::Stat#dev 的次设备部分。如果操作系统不支持这种特性，则返回 nil。 |
| File.stat("testfile").dev_minor | → 9 |
| directory? | <i>statfile.directory?</i> → true 或 false |
| 如果 <i>statfile</i> 是目录，则返回 true，否则返回 false。 | |
| File.stat("testfile").directory? | → false |
| File.stat(".").directory? | → true |

| | |
|--|---|
| executable? | <i>statfile.executable?</i> → true 或 false |
| 如果 <i>statfile</i> 是可执行文件或者操作系统不区分文件执行与否，则返回 true。
测试是用进程的有效拥有者进行的。 | |
| <code>File.stat("testfile").executable? → false</code> | |
| executable_real? | <i>statfile.executable_real?</i> → true 或 false |
| 和 executable?一样，但测试使用的是进程的真实拥有者。 | |
| file? | <i>statfile.file?</i> → true 或 false |
| 如果 <i>statfile</i> 是普通文件（不是设备文件、管道、socket 等等），则返回 true。 | |
| <code>File.stat("testfile").file? → true</code> | |
| ftype | <i>statfile.ftype</i> → type_string |
| 确定 <i>statfile</i> 的类型。返回的字符串有：file、directory、character-Special、blockSpecial、fifo、link、socket 或 unknown。
<code>File.stat("/dev/tty").ftype → "characterSpecial"</code> | |
| gid | <i>statfile.gid</i> → int |
| 返回 <i>statfile</i> 的拥有者的数字组 ID。
<code>File.stat("testfile").gid → 502</code> | |
| grpowned? | <i>statfile.grpowned?</i> → true 或 false |
| 如果进程的有效组 ID 和 <i>statfile</i> 的组 ID 相同，则返回 true。在 Windows 上返回 false。
<code>File.stat("testfile").grpowned? → true</code>
<code>File.stat("/etc/passwd").grpowned? → false</code> | |
| ino | <i>statfile.ino</i> → int |
| 返回 <i>statfile</i> 的 inode 号。
<code>File.stat("testfile").ino → 422829</code> | |
| mode | <i>statfile.mode</i> → int |
| 返回表示 <i>statfile</i> 的权限位的整数。位的含义是平台相关的；在 Unix 系统上，参见 stat(2)。
<code>File.chmod(0644, "testfile") → 1</code>
<code>File.stat("testfile").mode.to_s(8) → "100644"</code> | |

| | |
|--|--|
| mtime | <i>statfile.mtime</i> → <i>time</i> |
| 返回包含 <i>statfile</i> 的修改时间的一个 <i>Time</i> 对象。 | |
| <code>File.stat("testfile").mtime</code> → Thu Aug 26 22:37:33 CDT 2004 | |
| nlink | <i>statfile.nlink</i> → <i>int</i> |
| 返回 <i>statfile</i> 的硬链接的数。 | |
| <code>File.stat("testfile").nlink</code> → 1
<code>File.link("testfile", "testfile.bak")</code> → 0
<code>File.stat("testfile").nlink</code> → 2 | |
| owned? | <i>statfile.owned?</i> → true 或 false |
| 如果进程的有效用户 ID 和 <i>statfile</i> 的拥有者相同，则返回 true。 | |
| <code>File.stat("testfile").owned?</code> → true
<code>File.stat("/etc/passwd").owned?</code> → false | |
| pipe? | <i>statfile.pipe?</i> → true 或 false |
| 如果操作系统支持管道文件且 <i>statfile</i> 是一个管道文件，则返回 true。 | |
| rdev | <i>statfile.rdev</i> → <i>int</i> |
| 返回一个表示 <i>statfile</i> 所在设备的类型的一个整数， <i>statfile</i> 需要的是一个特殊文件。如果操作系统不支持这种特性，则返回 nil。 | |
| <code>File.stat("/dev/disk0s1").rdev</code> → 234881025
<code>File.stat("/dev/tty").rdev</code> → 33554432 | |
| rdev_major | <i>statfile.rdev_major</i> → <i>int</i> |
| <u>1.8</u> | 返回 <code>File::Stat#rdev</code> 的主设备号部分，如果操作系统不支持这种特性，则返回 nil。 |
| <code>File.stat("/dev/disk0s1").rdev_major</code> → 14
<code>File.stat("/dev/tty").rdev_major</code> → 2 | |
| rdev_minor | <i>statfile.rdev_minor</i> → <i>int</i> |
| <u>1.8</u> | 返回 <code>File::Stat#rdev</code> 的次设备号部分，如果操作系统不支持这种特性，则返回 nil。 |
| <code>File.stat("/dev/disk0s1").rdev_minor</code> → 1
<code>File.stat("/dev/tty").rdev_minor</code> → 0 | |
| readable? | <i>statfile.readable?</i> → true 或 false |
| 如果进程的有效用户 ID 对 <i>statfile</i> 有读权限，则返回 true。 | |
| <code>File.stat("testfile").readable?</code> → true | |

| | |
|---|---|
| readable_real? | <i>statfile.readable_real? → true 或 false</i> |
| 如果进程的真实用户 ID 对 <i>statfile</i> 有读权限，则返回 true。 | |
| File.stat("testfile").readable_real? → true
File.stat("/etc/passwd").readable_real? → true | |
| setgid? | <i>statfile.setgid? → true 或 false</i> |
| 如果 <i>statfile</i> 设置了 set-group-id 权限位，则返回 true，如果没有设置或者操作系统不支持此特性，则返回 false。 | |
| File.stat("testfile").setgid? → false
File.stat("/usr/sbin/postdrop").setgid? → true | |
| setuid? | <i>statfile.setuid? → true 或 false</i> |
| 如果 <i>statfile</i> 设置了 set-user-id 权限位，则返回 true，如果没有设置或者操作系统不支持此特性，则返回 false。 | |
| File.stat("testfile").setuid? → false
File.stat("/usr/bin/su").setuid? → true | |
| size | <i>statfile.size → int</i> |
| 以字节为单位返回 <i>statfile</i> 的大小。 | |
| File.stat("/dev/zero").size → 0
File.stat("testfile").size → 66 | |
| size? | <i>statfile.size? → int 或 nil</i> |
| 如果 <i>statfile</i> 文件长度为 0，则返回 nil，否则返回文件大小。在测试条件下很有用。 | |
| File.stat("/dev/zero").size? → nil
File.stat("testfile").size? → 66 | |
| socket? | <i>statfile.socket? → true 或 false</i> |
| 如果 <i>statfile</i> 是 socket 文件，则返回 true，如果不是或者操作系统不支持此特性，则返回 false。 | |
| File.stat("testfile").socket? → false | |
| sticky? | <i>statfile.sticky? → true 或 false</i> |
| 如果 <i>statfile</i> 设置了粘滞位，则返回 true，如果没有设置或者操作系统不支持此特性，则返回 false。 | |
| File.stat("testfile").sticky? → false | |

symlink?*statfile.symlink?* → true 或 false

如果 *statfile* 是符号链接文件，则返回 true，如果不是或者操作系统不支持此特性，则返回 false。因为 File.stat 会自动跟随符号链接文件，所以对 File.stat 返回的对象，symlink? 总是 false。

```
File.symlink("testfile", "alink") → 0
File.stat("alink").symlink? → false
File.lstat("alink").symlink? → true
```

uid

statfile.uid → int

返回 *statfile* 的拥有者的数字用户 ID。

```
File.stat("testfile").uid → 502
```

writable?

statfile.writable? → true 或 false

如果进程的有效用户 ID 读 *statfile* 有写权限，则返回 true。

```
File.stat("testfile").writable? → true
```

writable_real?

statfile.writable_real? → true 或 false

如果进程的真实用户 ID 读 *statfile* 有写权限，则返回 true。

```
File.stat("testfile").writable_real? → true
```

zero?

statfile.zero? → true 或 false

如果 *statfile* 文件的长度为 0，则返回 true；否则返回 false。

```
File.stat("testfile").zero? → false
```

Module FileTest

FileTest 实现了文件测试操作，这些操作与 `File::Stat` 所用的操作类似。FileTest 中的方法复制了类 `File` 中的方法。这里我们列出了方法的名字，并指引你参考第 465 页开始的关于 `File` 的文档，而不再重复其文档。FileTest 看起来是一个有点多余的模块。

FileTest 方法有：

`blockdev?, chardev?, directory?, executable?, executable_real?, exist?, exists?, file?, grpowned?, owned?, pipe?, readable?, readable_real?, setgid?, setuid?, size, size?, socket?, sticky?, symlink?, world_readable?, world_writable?, writable?, writable_real? 和 zero?`

1.8

Class Fixnum < Integer

Fixnum 所保存的 Integer 值是用本机机器字长（减一位）表示的。如果 Fixnum 上的任意操作超出了这个范围，那么值自动被转换成 Bignum。

Fixnum 对象含有立即数。这意味着当用它们进行赋值或作为参数传递时，操作的是对象本身而不是对象的引用。赋值操作不会为 Fixnum 对象起别名。因为对任何一个给定的整数值，实际上只有一个 Fixnum 对象实例，所以你不能添加单例方法到 Fixnum 中。

实例方法

算术操作

在 fix 上执行各种算术操作。

| | |
|----------------|-----|
| fix + numeric | 加 |
| fix - numeric | 减 |
| fix * numeric | 乘 |
| fix / numeric | 除 |
| fix % numeric | 求模 |
| fix ** numeric | 幂运算 |
| fix -@ | 一元减 |

位操作

在 Fixnum 的二进制表示上执行各种操作。

| | |
|----------------|--------------------|
| ~fix | 按位取反 |
| fix numeric | 按位或 |
| fix & numeric | 按位与 |
| fix ^ numeric | 按位异或 |
| fix << numeric | 左移 numeric 位 |
| fix >> numeric | 右移 numeric 位 (带符号) |

<=>

fix <=> numeric → -1, 0, +1

比较——根据 fix 小于、等于或大于 numeric 分别返回 -1, 0 或 +1。这是 Comparable 中测试操作的基础。

| | | |
|-----------|---|----|
| 42 <=> 13 | → | 1 |
| 13 <=> 42 | → | -1 |
| -1 <=> -1 | → | 0 |

[]

fix[n]→0,1

位引用——返回 *fix* 的二进制表示的第 *n* 位，其中 *fix[0]* 是最低位。

```
a = 0b11001100101010
30.downto(0) {|n| print a[n]}
```

输出结果：

```
000000000000000011001100101010
```

abs

fix.abs→int

返回 *fix* 的绝对值。

```
-12345.abs → 12345
12345.abs → 12345
```

div

fix.div(numeric)→integer

1.8

与 Fixnum#/ 同义。整数相除总是产生整数结果。

```
654321.div(13731) → 47
654321.div(13731.34) → 47
```

divmod

fix.divmod(numeric)→array

参见第 565 页的 Numeric#divmod。

id2name

fix.id2name→string 或 nil

返回符号 ID 为 *fix* 的对象的名字。如果符号表中没有这样的符号，则返回 nil。id2name 和方法 Object.object_id 没有任何关系。参见第 615 页的 Fixnum#to_sym、String#intern 和第 631 页的 Symbol 类。

```
symbol = :@inst_var → :@inst_var
id      = symbol.to_i → 9866
id.id2name → "@inst_var"
```

modulo

fix.modulo(numeric)→numeric

1.8

与 Fixnum#% 同义。

```
654321.modulo(13731) → 8964
654321.modulo(13731.24) → 8952.72000000001
```

quo

fix.quo(numeric)→float

1.8

返回用 *numeric* 除 *fix* 的浮点结果。

```
654321.quo(13731) → 47.6528293642124
654321.quo(13731.24) → 47.6519964693647
```

| | |
|--|---|
| size | <i>fix.size → int</i> |
| 返回 Fixnum 的机器表示的字节数。 | |
| <code>1.size → 4</code> | |
| <code>-1.size → 4</code> | |
| <code>2147483647.size → 4</code> | |
| <hr/> | |
| to_f | <i>fix.to_f → float</i> |
| 把 <i>fix</i> 转换为 <i>Float</i> 。 | |
| <hr/> | |
| to_s | <i>fix.to_s(base=10) → string</i> |
| <u>1.8</u> | 返回 <i>fix</i> 以 <i>base</i> (2 到 36) 为基数的值的字符串。 |
| <code>12345.to_s</code> | → "12345" |
| <code>12345.to_s(2)</code> | → "11000000111001" |
| <code>12345.to_s(8)</code> | → "30071" |
| <code>12345.to_s(10)</code> | → "12345" |
| <code>12345.to_s(16)</code> | → "3039" |
| <code>12345.to_s(36)</code> | → "9ix" |
| <code>84823723233035811745497171.to_s(36)</code> | → "anotherrubyhacker" |
| <hr/> | |
| to_sym | <i>fix.to_sym → symbol</i> |
| <u>1.8</u> | 返回整数值是 <i>fix</i> 的符号。参见 Fixnum#id2name。 |
| <code>fred = :fred.to_i</code> | |
| <code>fred.id2name → "fred"</code> | |
| <code>fred.to_sym → :fred</code> | |
| <hr/> | |
| zero? | <i>fix.zero? → true 或 false</i> |
| 如果 <i>fix</i> 是 0，则返回 <i>true</i> 。 | |
| <code>42.zero? → false</code> | |
| <code>0.zero? → true</code> | |

Class **Float** < Numeric

`Float` 对象使用本机体系结构的双精度浮点数表示实数。

类常量

| | |
|------------|--|
| DIG | Float 的精度（以十进制表示） |
| EPSILON | 使 $1.0 + \text{EPSILON} \neq 1.0$ 的最小 Float |
| MANT_DIG | 尾数的个数（基于 RADIX） |
| MAX | 最大的 Float |
| MAX_10_EXP | 使 10^x 为有限 Float 的最大整数 x |
| MAX_EXP | 使 $\text{FLT_RADIX}^{(x-1)}$ 为有限 Float 的最大整数 x |
| MIN | 最小的 Float |
| MIN_10_EXP | 使 10^x 为有限 Float 的最小整数 x |
| MIN_EXP | 使 $\text{FLT_RADIX}^{(x-1)}$ 为有限 Float 的最小整数 x |
| RADIX | 浮点表示的根 |
| ROUNDS | 浮点操作的舍入模式。可能的值有：
-1 如果是混合模式
0 如果向 0 舍入
1 如果向最接近的可表示的值舍入
2 如果向 $+\infty$ 舍入
3 如果向 $-\infty$ 舍入 |

实例方法

算术操作

在 `flt` 上可执行的各种算术操作。

| | | |
|---------------------|----------------------|-----|
| <code>flt +</code> | <code>numeric</code> | 加 |
| <code>flt -</code> | <code>numeric</code> | 减 |
| <code>flt *</code> | <code>numeric</code> | 乘 |
| <code>flt /</code> | <code>numeric</code> | 除 |
| <code>flt %</code> | <code>numeric</code> | 求模 |
| <code>flt **</code> | <code>numeric</code> | 幂 |
| <code>flt -@</code> | | 一元减 |

 <=>

 $flt \leqslant numeric \rightarrow -1, 0, +1$

根据 `flt` 小于、等于或大于 `numeric` 分别返回 -1, 0 或 +1。这是 Comparable 中测试操作的基础。

==*flt == obj* → true 或 false

如果 *obj* 和 *flt* 的值相同，则返回 true。和 Float#eq? 相比，Float#eq? 要求 *obj* 也是一个 Float。

```
1.0 == 1.0      →  true
(1.0).eq?(1.0) →  true
1.0 == 1        →  true
(1.0).eq?(1)   →  false
```

abs*flt.abs* → numeric

返回 *flt* 的绝对值。

```
(-34.56).abs    →  34.56
-34.56.abs     →  34.56
```

ceil*flt.ceil* → int

返回大于或等于 *flt* 的最小整数。

```
1.2.ceil       →  2
2.0.ceil       →  2
(-1.2).ceil   →  -1
(-2.0).ceil   →  -2
```

divmod*flt.divmod(numeric)* → array

参见第 565 页的 Numeric#divmod。

eq?*flt.eq?(obj)* → true 或 false

如果 *obj* 是和 *flt* 有相同值的 Float，则返回 true。与 Float#==相比，Float#==会执行类型转换。

```
1.0.eq?(1)      →  false
1.0 == 1        →  true
```

finite?*flt.finite?* → true 或 false

如果 *flt* 是合法的 IEEE 浮点数，则返回 true (*flt* 不是无限数，且 nan? 为 false)。

```
(42.0).finite? →  true
(1.0/0.0).finite? →  false
```

floor*flt.floor* → int

返回小于或等于 *flt* 的最小整数。

```
1.2.floor      →  1
2.0.floor      →  2
(-1.2).floor   →  -2
(-2.0).floor   →  -2
```

infinite?

flt.infinite? → nil, -1, +1

根据 *flt* 是有限数、 $-\infty$ 还是 $+\infty$ 分别返回 *nil*, *-1* 或 *+1*。

| | | |
|----------------------|---|-----|
| (0.0).infinite? | → | nil |
| (-1.0/0.0).infinite? | → | -1 |
| (+1.0/0.0).infinite? | → | 1 |

modulo

flt.modulo(numeric) → numeric

1.8. 与 *Float#%* 同义。

| | | |
|------------------------|---|-------------------|
| 6543.21.modulo(137) | → | 104.21 |
| 6543.21.modulo(137.24) | → | 92.92999999999996 |

nan?

flt.nan? → true 或 false

如果 *flt* 是个不合法的 IEEE 浮点数，则返回 *true*。

| | | |
|----------------|---|-------|
| (-1.0).nan? | → | false |
| (0.0/0.0).nan? | → | true |

round

flt.round → int

舍入 *flt* 为最接近的整数，等价于：

```
def round
  case
  when self > 0.0 then (self+0.5).floor
  when self < 0.0 then return (self-0.5).ceil
  else 0
  end
end

1.5.round      → 2
(-1.5).round   → -2
```

to_f

flt.to_f → flt

返回 *flt*。

to_i

flt.to_i → int

截断 *flt* 为整数，并返回该整数。

| | | |
|-------------|---|----|
| 1.5.to_i | → | 1 |
| (-1.5).to_i | → | -1 |

to_int

flt.to_int → int

与 *Float#to_i* 同义。

to_s*flt.to_s→string*

返回一个包含自身表示的字符串。和定点数和指数形式一样，本调用可能返回 NaN、Infinity 和 -Inifnity。

truncate*flt.truncate→int*

1.8.

与 `Float#to_i` 同义。**zero?***flt.zero?→true 或 false*

如果 `flt` 是 0.0，则返回 `true`。

Module GC

GC 模块为 Ruby 的标记清除（mark-sweep）垃圾回收机制提供了一个接口。里面的一些方法也可以通过第 578 页描述的 ObjectSpace 模块获得。

模块方法

| | |
|----------------|----------------------------------|
| disable | GC.disable → true 或 false |
|----------------|----------------------------------|

禁用垃圾回收，如果垃圾回收已经被禁止，则返回 true。

```
GC.disable → false
GC.disable → true
```

| | |
|---------------|---------------------------------|
| enable | GC.enable → true 或 false |
|---------------|---------------------------------|

启用垃圾回收，如果垃圾回收已经被禁用，则返回 true。

```
GC.disable → false
GC.enable → true
GC.enable → false
```

| | |
|--------------|-----------------------|
| start | GC.start → nil |
|--------------|-----------------------|

初始化垃圾回收机制，除非已被手动禁止。

```
GC.start → nil
```

实例方法

| | |
|------------------------|------------------------------|
| garbage_collect | garbage_collect → nil |
|------------------------|------------------------------|

与 GC.start 等价。

```
include GC
garbage_collect → nil
```

Class Hash < Object

Hash 是键/值对组成的集合。除了可以用任意对象类型作为键 (key) 而不仅仅是用整数进行索引之外，它和 Array 相似。基于 hash 内容的各种迭代器返回的键和 /或值的顺序是随意的，而且一般来说和插入的顺序不同。

散列表具有默认值。当我们试图用散列表中并不存在的 key 来访问它时，就会返回这个值。这个值默认为 nil。

Mixes in**Enumerable:**

```
all?, any?, collect, detect, each_with_index, entries, find, find_all, grep,
include?, inject, map, max, member?, min, partition, reject, select, sort, sort_
by, to_a, zip
```

类方法**[]**

`Hash[<key => value >*] → hsh`

根据给定的对象创建一个新的散列表。和用形式 `{key=>value, ...}` 创建散列表等价。键和值成对出现，所以必须有偶数个参数。

```
Hash["a", 100, "b", 200] → {"a"=>100, "b"=>200}
Hash["a" => 100, "b" => 200] → {"a"=>100, "b"=>200}
{ "a" => 100, "b" => 200 } → {"a"=>100, "b"=>200}
```

new

`Hash.new → hsh`

`Hash.new(obj) → hsh`

`Hash.new { |hash, key| block } → hsh`

1.8.

返回一个新的空散列表。如果随后在访问散列表时使用了没有对应项的键，返回的值依赖于创建散列表时所用的 new 形式。对第一种形式，访问返回 nil。如果指定了 obj，该对象将被用作所有值的默认值。如果给定了 block，将以散列表对象和键为参数调用 block，且 block 会返回默认值。如果需要，block 负责在散列表中存储值。

```
h = Hash.new("Go Fish")
h["a"] = 100
h["b"] = 200
h["a"] → 100
h["c"] → "Go Fish"
# The following alters the single default object
h["c"].upcase! → "GO FISH"
h["d"] → "GO FISH"
h.keys → ["a", "b"]
```

```
# While this creates a new default object each time
h = Hash.new{|hash, key| hash[key] = "Go Fish: #{key}"}
h["c"]           →      "Go Fish: c"
h["c"].upcase!   →      "GO FISH: C"
h["d"]           →      "Go Fish: d"
h.keys          →      ["c", "d"]
```

实例方法

==

hsh == obj → true 或 false

等价性——如果两个散列表有相同的默认值，含有相同数目的键且每个键在第一个散列表中对应的值等于（使用`==`）在第二个散列表中的值，则两个散列表相等。如果 *obj* 不是散列表，则试图使用 `to_hash` 方法先将它转换成散列表然后返回 *obj == hsh*。

```
h1 = { "a" => 1, "c" => 2 }
h2 = { 7 => 35, "c" => 2, "a" => 1 }
h3 = { "a" => 1, "c" => 2, 7 => 35 }
h4 = { "a" => 1, "d" => 2, "f" => 35 }
h1 == h2      →  false
h2 == h3      →  true
h3 == h4      →  false
```

[]

hsh[key] → *value*

元素引用——检索 *key* 对应的 *value*。如果没找到 *key*，则返回默认值（详情参见 `Hash.new`）。

```
h = { "a" => 100, "b" => 200 }
h["a"]        →  100
h["c"]        →  nil
```

[]=

hsh[key] = value → *value*

元素赋值——将 *value* 值和给定的键 *key* 相关联。当 *key* 被用作键时，它的值不能改变（作为键的 `String` 将被复制或冻结）。

```
h = { "a" => 100, "b" => 200 }
h["a"] = 9
h["c"] = 4
h → {"a"=>9, "b"=>200, "c"=>4}
```

clear

hsh.clear → *hsh*

从 *has* 中删除所有的键/值对。

```
h = { "a" => 100, "b" => 200 }      →  {"a"=>100, "b"=>200}
h.clear                           →  {}
```

default *hsh.default(key=nil)→obj*

1.8 返回默认值，即当 *key* 在 *hsh* 中不存在时由 *hsh[key]* 返回的那个值。也可以参见 Hash.new 和 Hash#default=。

```

h = Hash.new          → {}
h.default           → nil
h.default(2)         → nil

h = Hash.new("cat")   → {}
h.default           → "cat"
h.default(2)         → "cat"

h = Hash.new{|h,k| h[k] = k.to_i*10} → {}
h.default           → 0
h.default(2)         → 20

```

default= *hsh.default=obj→hsh*

设置默认值，即当键在散列表中不存在时返回的值。不能设置默认值为 Proc，在每次搜索键时它将被执行。

```

h = { "a" => 100, "b" => 200 }
h.default = "Go fish"
h["a"]   → 100
h["z"]   → "Go fish"
# This doesn't do what you might hope...
h.default = proc do |hash, key|
  hash[key] = key + key
end
h[2]     → #<Proc:0x001c94e0@-:6>
h["cat"] → #<Proc:0x001c94e0@-:6>

```

default_proc *hsh.default_proc→obj 或 nil*

1.8 如果调用 Hash.new 时带有一个 block，则返回那个 block；否则返回 nil。

```

h = Hash.new{|h,k| h[k] = k*k} → {}
p = h.default_proc           → #<Proc:0x001c997c@-:1>
a = []
p.call(a, 2)                 → []
a                           → [nil, nil, 4]

```

delete *hsh.delete(key)→value*
hsh.delete(key){|key|block}→value

从 *hsh* 中删除键为 *key* 的项，并返回对应的值。如果没有找到键，则返回 nil。

1.8 如果提供了一个 block 且没有找到 *key*，则将 *key* 传递给 block 并返回 block 的结果。

```

h = { "a" => 100, "b" => 200 }
h.delete("a")                                → 100
h.delete("z")                                → nil
h.delete("z") { |el| "#{el} not found" }      → "z not found"

```

delete_if*hsh.delete_if { |key, value| block } → hsh*

从 *hsh* 中删除所有使 *block* 为真的键/值对。

```

h = { "a" => 100, "b" => 200, "c" => 300 }
h.delete_if { |key, value| key >= "b" } → {"a"=>100}

```

each*hsh.each { |key, value| block } → hsh*

对 *hsh* 中的每个键调用一次 *block*, 并传递键和值作为 *block* 的参数。

```

h = { "a" => 100, "b" => 200 }
h.each { |key, value| puts "#{key} is #{value}" }

```

输出结果:

```

a is 100
b is 200

```

each_key*hsh.each_key { |key| block } → hsh*

对散列表中的每个键调用一次 *block*, 并传递键作为 *block* 的参数。

```

h = { "a" => 100, "b" => 200 }
h.each_key { |key| puts key }

```

输出结果:

```

a
b

```

each_pair*hsh.each_pair { |key, value| block } → hsh*

与 Hash#each 同义。

each_value*hsh.each_value { |value| block } → hsh*

对 *hsh* 中的每个键调用一次 *block*, 并传递键对应的值作为 *block* 的参数。

```

h = { "a" => 100, "b" => 200 }
h.each_value { |value| puts value }

```

输出结果:

```

100
200

```

empty?*hsh.empty? → true 或 false*

如果 *hsh* 没有任何键/值对, 则返回 *true*。

```
{ }.empty? → true
```

| | |
|--------------|--|
| fetch | <i>hsh.fetch(key <, default >) → obj</i>
<i>hsh.fetch(key) { key block } → obj</i> |
|--------------|--|

从散列表中返回给定键对应的值。如果找不到键，则有几种可能：如果没有其他参数，则引发一个 `IndexError` 异常；如果有 `default` 参数，则返回该参数；如果有关联的 `block`，则该 `block` 会运行，并返回其结果。`fetch` 不会计算创建散列表时提供的任何默认值——只在散列表中查找键。

```
h = { "a" => 100, "b" => 200 }
h.fetch("a") → 100
h.fetch("z", "go fish") → "go fish"
h.fetch("z") { |el| "go fish, #{el}" } → "go fish, z"
```

下面的例子演示了当找不到键且没有提供默认值时将引发一个异常。

```
h = { "a" => 100, "b" => 200 }
h.fetch("z")
```

输出结果：

```
prog.rb:2:in `fetch': key not found (IndexError)
from prog.rb:2
```

| | |
|-----------------|---|
| has_key? | <i>hsh.has_key?(key) → true 或 false</i> |
|-----------------|---|

如果给定键在 `hsh` 中存在，则返回 `true`。

```
h = { "a" => 100, "b" => 200 }
h.has_key?("a") → true
h.has_key?("z") → false
```

| | |
|-------------------|---|
| has_value? | <i>hsh.has_value?(value) → true 或 false</i> |
|-------------------|---|

如果给定值在 `hsh` 中是某个键对应的值，则返回 `true`。

```
h = { "a" => 100, "b" => 200 }
h.has_value?(100) → true
h.has_value?(999) → false
```

| | |
|-----------------|---|
| include? | <i>hsh.include?(key) → true 或 false</i> |
|-----------------|---|

与 `Hash#has_key?` 同义。

| | |
|--------------|---------------------------------|
| index | <i>hsh.index(value) → key</i> |
|--------------|---------------------------------|

在散列表中搜索值`==value` 的项，并返回相应的键。如果有多个项含有此值，则随机返回其中之一。如果没有找到满足条件的项，则返回 `nil`。

```
h = { "a" => 100, "b" => 200 }
h.index(200) → "b"
h.index(999) → nil
```

| | |
|----------------|--|
| indexes | <i>hsh.indexes(<key>+) → array</i> |
| 1.8 | 由于 Hash#values_at 的出现，此函数已不推荐使用。 |
| indices | <i>hsh.indices(<key>+) → array</i> |
| 1.8 | 由于 Hash#values_at 的出现，此函数已不推荐使用。 |
| invert | <i>hsh.invert → other_hash</i> |
| | 返回一个使用 <i>hsh</i> 的值做键、键做值的新散列表。如果 <i>hsh</i> 有重复的值，新散列表只使用其中一个作为键——到底用哪个是不可预测的。 |
| | <pre>h = { "n" => 100, "m" => 100, "y" => 300, "d" => 200, "a" => 0 } h.invert → {0=>"a", 100=>"n", 200=>"d", 300=>"y"}</pre> |
| key? | <i>hsh.key?(key) → true 或 false</i> |
| | 与 Hash#has_key? 同义。 |
| keys | <i>hsh.keys → array</i> |
| | 返回有散列表的键组成的数组。参见 Hash#values。 |
| | <pre>h = { "a" => 100, "b" => 200, "c" => 300, "d" => 400 } h.keys → ["a", "b", "c", "d"]</pre> |
| length | <i>hsh.length → fixnum</i> |
| | 返回散列表中键/值对的个数。 |
| | <pre>h = { "d" => 100, "a" => 200, "v" => 300, "e" => 400 } h.length → 4 h.delete("a") → 200 h.length → 3</pre> |
| member? | <i>hsh.member?(key) → true 或 false</i> |
| | 与 Hash#has_key? 同义。 |
| merge | <i>hsh.merge(other_hash) → result_hash</i>
<i>hsh.merge(other_hash) { key, old_val, new_val block } → result_hash</i> |
| 1.8 | 返回即含有 <i>other_hash</i> 的内容又含有 <i>hsh</i> 的内容的新散列表。如果没有 <i>block</i> 参数，在 <i>hsh</i> 和 <i>other_hash</i> 有重复键的情况下将使用 <i>other_hash</i> 中的项覆盖 <i>hsh</i> 的项。如果给定了 <i>block</i> ，将以重复的键和两个散列表的值为参数调用此 <i>block</i> 。 <i>Block</i> 的返回值将作为新散列表的值。 |

```

h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge(h2)           → {"a"=>100, "b"=>254, "c"=>300}
h1.merge(h2) {|k,o,n| o} → {"a"=>100, "b"=>200, "c"=>300}
h1                         → {"a"=>100, "b"=>200}

```

merge!

hsh.merge!(other_hash) → hsh
hsh.merge!(other_hash) { |key, old_val, new_val| block } → hsh

- 将 *other_hash* 的内容加入到 *hsh*，如果有重复的键，则使用 *other_hash* 的值覆盖 *hsh* 的值。

```

h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge!(h2)           → {"a"=>100, "b"=>254, "c"=>300}
h1 = { "a" => 100, "b" => 200 }
h1.merge!(h2) {|k,o,n| o} → {"a"=>100, "b"=>200, "c"=>300}
h1                         → {"a"=>100, "b"=>200, "c"=>300}

```

rehash*hsh.rehash → hsh*

根据每个键的当前值重新构建散列表。如果在键对象插入散列表后其值发生了变化，使用此方法来为 *hsh* 重新索引。如果在用迭代遍历 *hash* 时调用了 Hash#rehash，则迭代内会引发 IndexError 异常。

```

a = [ "a", "b" ]
c = [ "c", "d" ]
h = { a => 100, c => 300 }
h[a]      → 100
a[0] = "z"
h[a]      → nil
h.rehash → {[ "z", "b" ]=>100, [ "c", "d" ]=>300}
h[a]      → 100

```

reject*hsh.reject { |key, value| block } → hash*

和 Hash#delete_if 一样，但是在 *hsh* 的拷贝上处理。等价于 *hsh.dup.delete_if*。

reject!*hsh.reject! { |key, value| block } → hsh 或 nil*

等价于 Hash#delete_if，但如果没做任何改动，则返回 nil。

replace*hsh.replace(other_hash) → hsh*

使用 *other_hash* 的内容替换 *hsh* 的内容。

```

h = { "a" => 100, "b" => 200 }
h.replace({ "c" => 300, "d" => 400 }) → {"c"=>300, "d"=>400}

```

| | |
|----------------|--|
| select | <i>hsh.select { key, value block } → array</i> |
| | 返回由令 <code>block</code> 返回 <code>true</code> 的 <code>[key, value]</code> 对组成的一个数组。参见 <code>Hash#values_at</code> 。 |
| | <pre>h = { "a" => 100, "b" => 200, "c" => 300 } h.select { k,v k > "a" } → [["b", 200], ["c", 300]] h.select { k,v v < 200} → [["a", 100]]</pre> |
| shift | <i>hsh.shift → array 或 nil</i> |
| <u>1.8</u> | 从 <code>hsh</code> 中删除一个键/值对，并返回由它们组成的含有两个元素的数组 <code>[key,value]</code> 。如果 <code>hash</code> 为空，则返回默认值，并调用默认的 <code>proc</code> （以 <code>nil</code> 键）或返回 <code>nil</code> 。 |
| | <pre>h = { 1 => "a", 2 => "b", 3 => "c" } h.shift → [1, "a"] h → {2=>"b", 3=>"c"}</pre> |
| size | <i>hsh.size → fixnum</i> |
| | 与 <code>Hash#length</code> 同义。 |
| sort | <i>hsh.sort → array</i>
<i>hsh.sort { a, b block } → array</i> |
| | 将 <code>hsh</code> 转换成一个由 <code>[key, value]</code> 数组组成的嵌套数组，并使用 <code>Array#sort</code> 对它进行排序。 |
| | <pre>h = { "a" => 20, "b" => 30, "c" => 10 } h.sort → [[{"a": 20}, {"b": 30}, {"c": 10}] h.sort { a,b a[1]<=>b[1]} → [{"c": 10}, {"a": 20}, {"b": 30}]</pre> |
| store | <i>hsh.store(key, value) → value</i> |
| | 和元素赋值(<code>Hash#[] =</code>)同义。 |
| to_a | <i>hsh.to_a → array</i> |
| | 将 <code>hsh</code> 转换成一个数组，数组元素是由 <code>[key, value]</code> 组成的数组。 |
| | <pre>h = { "c" => 300, "a" => 100, "d" => 400, "c" => 300 } h.to_a → [{"a": 100}, {"c": 300}, {"d": 400}]</pre> |
| to_hash | <i>hsh.to_hash → hsh</i> |
| | 参见第 372 页。 |
| to_s | <i>hsh.to_s → string</i> |
| | 将散列表转换成一个由 <code>[key, value]</code> 对组成的数组，然后使用默认分隔符通过 <code>Array#join</code> 将数组转换成字符串。 |
| | <pre>h = { "c" => 300, "a" => 100, "d" => 400, "c" => 300 } h.to_s → "a100c300d400"</pre> |

| | |
|------------------|---|
| update | <i>hsh.update(other_hash) → hsh</i>
<i>hsh.update(other_hash) { key, old_val, new_val block } → hsh</i> |
| 1.8 | 与 Hash#merge! 同义。 |
| value? | <i>hsh.value?(value) → true 或 false</i> |
| | 与 Hash#has_value? 同义。 |
| values | <i>hsh.values → array</i> |
| | 返回一个由 <i>hsh</i> 的值组成的数组。参见 Hash#keys。

h = { "a" => 100, "b" => 200, "c" => 300 }
h.values → [100, 200, 300] |
| values_at | <i>hsh.values_at(<key>+) → array</i> |
| 1.8 | 返回由给定键对应的值组成的数组。对于没有找到的键，则插入创建数组时指定的默认值。

h = { "a" => 100, "b" => 200, "c" => 300 }
h.values_at("a", "c") → [100, 300]
h.values_at("a", "c", "z") → [100, 300, nil]
h.default = "cat"
h.values_at("a", "c", "z") → [100, 300, "cat"] |

Class Integer < Numeric

子类: Bignum, Fixnum

`Integer` 是表示整数的两个具体的类 `Bignum` 和 `Fixnum` 的基础（如果你在查找迭代器 `step`, 它的文档在第 566 页）。

实例方法

| | |
|-------------------|---------------------------------|
| <code>ceil</code> | <code>int.ceil → integer</code> |
|-------------------|---------------------------------|

与 `Integer#to_i` 同义。

| | |
|------------------|-------------------------------|
| <code>chr</code> | <code>int.chr → string</code> |
|------------------|-------------------------------|

返回接受者的值所表示的 ASCII 字符组成的字符串。

```
65.chr → "A"  
?a.chr → "a"  
230.chr → "\346"
```

| | |
|---------------------|--|
| <code>downto</code> | <code>int.downto(integer) { i block } → int</code> |
|---------------------|--|

迭代 `block`, 并传递从 `int` 到 `integer` (包括 `integer`) 递减的值作为参数。

```
5.downto(1) { |n| print n, "... " }  
print "Liftoff!\n"
```

输出结果:

```
5..4..3..2..1.. Liftoff!
```

| | |
|--------------------|----------------------------------|
| <code>floor</code> | <code>int.floor → integer</code> |
|--------------------|----------------------------------|

返回小于或等于 `int` 的最大整数。等价于 `Integer#to_i`。

```
1.floor → 1  
(-1).floor → -1
```

| | |
|-----------------------|----------------------------------|
| <code>integer?</code> | <code>int.integer? → true</code> |
|-----------------------|----------------------------------|

总是返回 `true`。

| | |
|-------------------|---------------------------------|
| <code>next</code> | <code>int.next → integer</code> |
|-------------------|---------------------------------|

返回等于 `int + 1` 的 `Integer`。

```
1.next → 2  
(-1).next → 0
```

| | |
|--------------------|----------------------------------|
| <code>round</code> | <code>int.round → integer</code> |
|--------------------|----------------------------------|

与 `Integer#to_i` 同义。

| | |
|---|--|
| succ | <i>int.succ → integer</i> |
| 与 Integer#next 同义。 | |
| times | <i>int.times { i block } → int</i> |
| 迭代 block <i>int</i> 次，并传递从 0 到 <i>int</i> - 1 的值作为参数。 | |
| | 5.times do i
print i, " "
end |
| 输出结果：
0 1 2 3 4 | |
| to_i | <i>int.to_i → int</i> |
| 返回 <i>int</i> 。 | |
| to_int | <i>int.to_int → integer</i> |
| 与 Integer#to_i 同义。 | |
| truncate | <i>int.truncate → integer</i> |
| 与 Integer#to_i 同义。 | |
| upto | <i>int.upto(integer) { i block } → int</i> |
| 迭代 block，并传递从 <i>int</i> 到 <i>integer</i> （包括 <i>integer</i> ）递增的值作为参数。 | |
| | 5.upto(10) { i print i, " " } |
| 输出结果：
5 6 7 8 9 10 | |

Class **IO** < Object

子类: File

在 Ruby 中类 `IO` 是所有输入和输出操作的基础。一个 `IO` 流可以复用（也就是双向），所以可以使用多个本机操作系统流。

本节中的很多例子使用了类 `File`，它是 `IO` 的唯一一个标准子类。这两个类联系很紧密。

如本节所用，`portname` 有如下几种形式：

- 表示适用于底层操作系统的文件名的一个普通字符串。
- 以 `+` 开头的字符串表示一个子进程。`+` 后面的其余字符串作为一个进程被调用，并为之连接适当的输入 / 输出通道。
- 等于 `|-` 的字符串将创建另外一个 Ruby 实例子进程。

类 `IO` 使用了 Unix 文件描述符，文件描述符是表示已打开文件的小整数。传统上，标准输入的文件描述符是 0，标注输出的文件描述符是 1 而标准错误的文件描述符是 2。

如果可能的话，Ruby 会根据不同操作系统的用法转换路径名。例如在 Windows 系统上文件名 `/gumby/ruby/test.rb` 将作为 `\gumby\ruby\test.rb` 打开。当在双引号字符串中指定 Windows 风格的文件名时，记住要转义反斜线。

`"c:\\gumby\\\\ruby\\\\test.rb"`

这里我们的例子使用 Unix 风格的斜线；可以用 `File::SEPARATOR` 来获得平台相关的路径分隔符。

有几种不同的打开 I/O 的方式，本节中用 `modestring` 来表示这些模式。模式字符串必须是下页的表 27.6 列出的值之一。

Mixes in

Enumerable:

`all?, any?, collect, detect, each_with_index, entries, find, find_all, grep, include?, inject, map, max, member?, min, partition, reject, select, sort, sort_by, to_a, zip`

表 27.6 模式字符串

| 模 式 | 含 义 |
|-----|--|
| r | 只读，从文件的开头开始读（默认模式） |
| r+ | 读/写，从文件的开头开始 |
| w | 只写，将已存在的文件的长度截断为 0 或者为写操作创建一个新文件 |
| w+ | 读/写，将已存在的文件的长度截断为 0 或者为读/写操作创建一个新文件 |
| a | 只写，如果文件存在，则从文件结尾开始写；否则为写操作创建一个新文件 |
| a+ | 读/写，如果文件存在，则从文件结尾开始写；否则为读/写操作创建一个新文件 |
| b | （只用于 DOS/Windows）二进制文件模式（可以和前面列出的任意字符组合使用） |

类方法

| | |
|---------------------|--|
| <code>for_fd</code> | <code>IO.for_fd(int, modestring) → io</code> |
| <code>1.8</code> | 和 <code>IO.new</code> 同义。 |

| | |
|---|---|
| <code>foreach</code> | <code>IO.foreach(portname, separator=\$/) { line block } → nil</code> |
| 为给定名字 I/O 端口中的每一行执行相关联的 block，其中的行分隔符是 <code>separator</code> 。 | |

```
IO.foreach("testfile") { |x| puts "GOT: #{x}" }
```

输出结果：

```
GOT: This is line one
GOT: This is line two
GOT: This is line three
GOT: And so on...
```

| | |
|--|---|
| <code>new</code> | <code>IO.new(int, modestring) → io</code> |
| 根据给定的整数文件描述符和模式字符串返回一个新的 IO 对象（一个流）。参见 <code>IO#fileno</code> 和 <code>IO.for_fd</code> 。 | |

```
a = IO.new(2, "w")    # '2' is standard error
STDERR.puts "Hello"
a.puts "World"
```

输出结果：

```
Hello
World
```

open

IO.open(*int, modestring*) → *io*
IO.open(*int, modestring*) { | *io* | *block* } → *obj*

如果没有关联 *block*, *open* 和 *IO.new* 同义。如果给出了代码 *block*, 那么将以 *io* 为参数传递给 *block*, 并在 *block* 结束时自动关闭 *io* 对象。在这种情况下, *IO.open* 返回 *block* 的值。

```
IO.open(1, "w") do |io|
  io.puts "Writing to stdout"
end
```

输出结果:

```
Writing to stdout
```

pipe

IO.pipe → array

创建一对管道端点（互相连接），并返回由它们组成的一个含有两个元素的数组: [*read_file*, *write_file*]、*write_file* 将自动处于同步模式。不是在所有的平台上可用。

下面的例子中，两个进程关闭了它们不用的管道端点。这样做不是为了美观。如果有任意一个写者打开了管道，那么管道的读端不会产生 end-of-file 条件。在这个例子的父进程中，如果不先调用 *wr.close*, 那么 *rd.read* 就永远不会返回。

```
rd, wr = IO.pipe
if fork
  wr.close
  puts "Parent got: <#{rd.read}>"
  rd.close
  Process.wait
else
  rd.close
  puts "Sending message to parent"
  wr.write "Hi Dad"
  wr.close
end
```

输出结果:

```
Sending message to parent
Parent got: <Hi Dad>
```

popen

IO.popen(*cmd, modestring="r"*) → *io*
IO.popen(*cmd, modestring="r"*) { | *io* | *block* } → *obj*

以子进程运行指定的命令字符串；该子进程的标准输入和输出将连接到返回的 *IO* 对象上。参数 *cmd* 可以是一个字符串或者（在 Ruby 1.9 中）字符串数组。在后一种情况下，数组将被作为新进程的 *argv* 参数，且在字符串上不执行任何特殊 shell 处理。

如果 *cmd* 是一个字符串，它将会被 shell 扩展。如果 *cmd* 字符串以减号 (-) 开头，且操作系统支持 fork(2)，那么当前的 Ruby 进程将被克隆。新文件对象的默认模式是 r，但是 *modestring* 可以为 504 页表 27.6 中列出的任意模式。

如果带有 block，Ruby 将以通过管道连接到 Ruby 的一个子进程来运行命令。管道的 Ruby 这一端将作为参数传递给 block。在这种情况下，`IO.popen` 返回 block 的值。

如果带有 block，且 *cmd* 字符串是“-”，那么 block 将在两个进程中运行：一个是父进程，一个是子进程。父进程中将传递管道对象作为 block 的参数，子进程中传递 nil 作为 block 的参数，并且子进程的标准输入和标准输出将通过管道链接到父进程。不是在所有平台上都支持。参见第 708 页的 Open3 库和第 521 页的 Kernel#exec。

```
pipe = IO.popen("uname")
p(pipe.readlines)
puts "Parent is #{Process.pid}"
IO.popen("date"){|pipe| puts pipe.gets}
IO.popen("-"){|pipe| STDERR.puts "#{Process.pid} is here, pipe=#{pipe}"}
```

输出结果：

```
["Darwin\n"]
Parent is 9778
Wed May 18 09:07:29 CDT 2005
9778 is here, pipe=<IO:0x1ce418>
9781 is here, pipe=
```

read`IO.read(portname, <length=$/<, offset>>) → string`**1.8**

打开文件，将文件指针移动到给定的偏移处，并返回 *length* 字节（默认是文件的剩余部分）。`read` 保证返回之前关闭文件。

| | | |
|--|---|--|
| <code>IO.read("testfile")</code> | → | "This is line one\nThis is line two\nThis is line three\nAnd so on...\n" |
| <code>IO.read("testfile", 20)</code> | → | "This is line one\nThi" |
| <code>IO.read("testfile", 20, 10)</code> | → | "ne one\nThis is line " |

readlines`IO.readlines(portname, separator=$/) → array`

读取 *portname* 指定的整个文件，把文件内容作为一个个的行，并返回由这些行组成的一个数组。行分隔符为 *separator*。

```
a = IO.readlines("testfile")
a[0] → "This is line one\n"
```

select IO.select(*read_array* <, *write_array* <, *error_array* <, *timeout* >>) → *array* 或 *nil*

参见第 528 页的 Kernel#select。

sysopen IO.sysopen(*path*, < *mode* >, *perm* >>) → *int*

1.8 打开给定的路径，以 Fixnum 返回底层的文件描述符。

```
IO.sysopen("testfile") → 3
```

实例方法

<< *io* << *obj* → *io*

字符串输出——输出 *obj* 到 *io*。*obj* 将被 *to_s* 转换成一个字符串。

```
STDOUT << "Hello " << "world!\n"
```

输出结果：

```
Hello world!
```

binmode *io.binmode* → *io*

使 *io* 进入二进制模式。这只在 MS-DOS/Windows 环境中有用。一旦一个流进入二进制模式，那么它将不能被重新设置为非二进制模式。

clone *io.clone* → *io*

通过拷贝 *io* 的所有属性，创建一个新的 I/O 流。文件位置指针也会被共享，因此从克隆中读也会改变原文件对象的文件指针，反之亦然。

close *io.close* → *nil*

关闭 *io*，并将所有未写入的数据写到操作系统中。此后该流将无法进行任何数据操作；如果试图操作，则该流将引发 IOError。当进行垃圾回收时，未关闭的 IO 流将自动被关闭。

close_read *io.close_read* → *nil*

关闭双向 I/O 流（包含读流和写流的流，例如管道）的读取端。如果流不是双向的，则会引发 IOError。

```
f = IO.popen("/bin/sh", "r+")
f.close_read
f.readlines
```

输出结果：

```
prog.rb:3:in `readlines': not opened for reading (IOError)
from prog.rb:3
```

close_write*io.close_write* → nil

关闭双向 I/O 流（包含读流和写流的流，例如管道）的写入端。如果流不是双向的，则会引发 IOError。

```
f = IO.popen("/bin/sh", "r+")
f.close_write
f.print "nowhere"
```

输出结果：

```
prog.rb:3:in `write': not opened for writing (IOError)
from prog.rb:3:in `print'
from prog.rb:3
```

closed?

io.closed? → true 或 false

如果 *io* 被完全关闭（对双向流而言，包括读和写），则返回 true，否则返回 false。

```
f = File.new("testfile")
f.close      → nil
f.closed?    → true
f = IO.popen("/bin/sh", "r+")
f.close_write → nil
f.closed?    → false
f.close_read → nil
f.closed?    → true
```

each

io.each(separator=\$/){|line|block|} → io

对 *io* 中的每一行执行 *block*，其中行之间由 *separator* 分割。*io* 必须已经以读模式打开，否则引发 IOError。

```
f = File.new("testfile")
f.each{|line| puts "#{f.lineno}: #{line}"}
```

输出结果：

```
1: This is line one
2: This is line two
3: This is line three
4: And so on...
```

each_byte

io.each_byte{|byte|block|} → nil

对 *io* 中的每一个字节（0 到 255 之间的一个 Fixnum），以该字节为参数调用给定的 *block*。*io* 必须已经以读模式打开，否则引发 IOError。

```
f = File.new("testfile")
checksum = 0
f.each_byte{|x| checksum ^= x} → #<File:testfile>
checksum                   → 12
```

| | |
|------------------|--|
| each_line | <i>io.each_line(separator=\$/) { line block } → io</i> |
|------------------|--|

与 `IO#each` 同义。

| | |
|------------|------------------------------|
| eof | <i>io.eof → true 或 false</i> |
|------------|------------------------------|

如果 `io` 处于文件末尾，则返回 `true`。流必须已经以读模式打开，否则引发 `IOError`。

```
f = File.new("testfile")
dummy = f.readlines
f.eof → true
```

| | |
|-------------|-------------------------------|
| eof? | <i>io.eof? → true 或 false</i> |
|-------------|-------------------------------|

与 `IO#eof` 同义。

| | |
|--------------|-----------------------------------|
| fcntl | <i>io.fcntl(cmd, arg) → int</i> |
|--------------|-----------------------------------|

提供一种机制来调用底层的命令来控制或查询面向文件的 I/O 流。命令（是整数）、参数和结果是平台相关的。如果 `arg` 是一个数字，它的值将被直接传递。如果是一个字符串，它将被解释为二进制字节序。在 Unix 平台上，详情参见 `fctnl(2)`。Fcntl 模块为第一个参数提供了符号名字（参见第 677 页）。仍未在所有平台上实现。

| | |
|---------------|------------------------|
| fileno | <i>io.fileno → int</i> |
|---------------|------------------------|

返回表示 `io` 文件描述符的数字整数。

```
STDIN.fileno → 0
STDOUT.fileno → 1
```

| | |
|--------------|----------------------|
| flush | <i>io.flush → io</i> |
|--------------|----------------------|

将 `io` 内所有缓冲的数据写入底层的操作系统（注意这只对 Ruby 的内部缓冲；OS 也可能缓冲数据）。

```
STDOUT.print "no newline"
STDOUT.flush
```

输出结果：

```
no newline
```

| | |
|--------------|---------------------------|
| fsync | <i>io.fsync → 0 或 nil</i> |
|--------------|---------------------------|

立即将 `io` 中缓存的所有数据写入磁盘。如果底层的操作系统不支持 `fsync(2)`，则返回 `nil`。注意 `fsync` 和使用 `IO#sync=` 不同。后者保证 Ruby 自己缓存的数据被写入操作系统，但是不能保证底层的操作系统真正将数据写入到磁盘。

getc*io.getc* → int 或 nil

从 *io* 中获得下一个 8 位字节 (0..255)。如果到达文件末尾，则返回 nil。

```
f = File.new("testfile")
f.getc → 84
f.getc → 104
```

gets

io.gets(separator=\$/) → string 或 nil

从 I/O 流中读取下一“行”；行之间由 *separator* 分割。如果分隔符为 nil，则读取整个文件；如果分隔符长度为 0，则一次读一段（输入中两个连续的新行分割段落）。流必须已经以读模式打开，否则将引发 IOError。读入的行将被返回，并被赋值给 *\$_*。如果到达文件末尾，则返回 nil。

```
File.new("testfile").gets → "This is line one\n"
$_ → "This is line one\n"
```

ioctl

io.ioctl(cmd, arg) → int

提供一种机制来调用底层的命令来控制或查询 I/O 设备。命令（是整数）、参数和结果是平台相关的。如果 *arg* 是一个数字，它的值将被直接传递。如果是一个字符串，它将被解释为二进制字节序。在 Unix 平台上，详情参见 ioctl(2)。仍未在所有平台上实现。

isatty*io.isatty* → true 或 false

如果 *io* 被关联到过终端设备 (tty)，则返回 true；否则返回 false。

```
File.new("testfile").isatty → false
File.new("/dev/tty").isatty → true
```

lineno

io.lineno → int

返回 *io* 的当前行号。流必须已经以读模式打开。lineno 计算 gets 被调用的次数，它并不计算遇到的新行的个数。如果调用 gets 时的分隔符不是新行符，那么这两个值会不同。参见 *\$.* 变量。

```
f = File.new("testfile")
f.lineno → 0
f.gets → "This is line one\n"
f.lineno → 1
f.gets → "This is line two\n"
f.lineno → 2
```

lineno=*io.lineno = int → int*

人为将当前行号设置为给定值。仅在下次读的时候才更新\$..

```
f = File.new("testfile")
f.gets           → "This is line one\n"
$.              → 1
f.lineno = 1000
f.lineno         → 1000
$. # lineno of last read → 1
f.gets           → "This is line two\n"
$. # lineno of last read → 1001
```

pid

io.pid → int

返回和 *io* 相关联的子进程的进程 ID。它是由 *IO.popen* 设置的。

```
pipe = IO.popen("-")
if pipe
  STDERR.puts "In parent, child pid is #{pipe.pid}"
else
  STDERR.puts "In child, pid is #{ $$ }"
end
```

输出结果：

```
In child, pid is 26884
In parent child, pid is 26884
```

pos

io.pos → int

返回 *io* 的当前偏移（以字节为单位）。

```
f = File.new("testfile")
f.pos   → 0
f.gets   → "This is line one\n"
f.pos   → 17
```

pos=

io.pos = int → 0

设置 *io* 的文件指针为给定值（以字节为单位）。

```
f = File.new("testfile")
f.pos = 17
f.gets   → "This is line two\n"
```

print

io.print(<obj=\$_>) → nil*

将给定的对象写入 *io* 中。流必须以写模式打开。如果输出记录分隔符(\$\n)不为 nil，它则将附加到输出流中。如果没有参数，则输出\$_。不是字符串的对象将调用它的 to_s 方法并将之转换成字符串。返回 nil。

```
STDOUT.print("This is ", 100, " percent.\n")
```

输出结果:

```
This is 100 percent.
```

printf*io.printf(format <, obj >*) → nil*

格式化并写入 *io*, 根据控制字符串的格式转化对应的参数。详情参见第 529 页的 Kernel#sprintf。

putc*io.putc(obj) → obj*

把给定的字符（从 String 或 Fixnum 中获得）写入 *io*。

```
STDOUT.putc "A"
STDOUT.putc 65
```

输出结果:

```
AA
```

puts*io.puts(<obj>*) → nil*

与 IO#print 一样，把给定的对象写入 *io*。如果任何一个被写入的对象不以新行符结尾，则输出一个新行符。如果以数组为参数，则每个元素都输出到单独的行上。如果不带参数调用，则输出一个新行。

```
STDOUT.puts("this", "is", "a", "test")
```

输出结果:

```
this
is
a
test
```

read*io.read(<int>, buffer >>) → string 或 nil*

从 I/O 流中至多读取 *int* 字节；如果 *int* 省略了，则读到文件末尾。如果已经到达文件末尾，则返回 nil。如果提供了 *buffer*（一个 String），则它将相应地重新调整大小，并直接将读取的内容存储其中。
1.8

```
f = File.new("testfile")
f.read(16)      →      "This is line one"
str = "cat"
f.read(10, str) →      "\nThis is 1"
str            →      "\nThis is 1"
```

readchar*io.readchar → int*

和 IO#getc 一样读取一个字符，但是如果到达文件末尾，则引发 EOFError。

readline*io.readline(separator=\$/) → string*

和 IO#gets 一样读取一行，但如果到达文件末尾，则引发 EOFError。

readlines*io.readlines(separator=\$/) → array*

读取 *io* 中的所有行，返回由这些行组成的一个数组。行之间由可选的 *separator* 参数分割。流必须以读模式打开，否则将引发 *IOError*。

```
f = File.new("testfile")
f.readlines → ["This is line one\n", "This is line two\n", "This
is line three\n", "And so on...\n"]
```

reopen

*io.reopen(other_io) → io**io.reopen(path, modestring) → io*

重新关联 *io* 到给定的流 *other_io* 或在 *path* 上打开的新流。这可能会动态地改变流的实际类。

```
f1 = File.new("testfile")
f2 = File.new("testfile")
f2.readlines[0] → "This is line one\n"
f2.reopen(f1) → #<File:testfile>
f2.readlines[0] → "This is line one\n"
```

rewind

io.rewind → 0

重置 *io* 到输入流的开始，且重置 *lineno* 为 0。

```
f = File.new("testfile")
f.readline → "This is line one\n"
f.rewind → 0
f.lineno → 0
f.readline → "This is line one\n"
```

seek

io.seek(int, whence=SEEK_SET) → 0

根据 *whence* 的值，使流指针移动给定的偏移量 *int*。

`IO::SEEK_CUR` 移动到当前位置加 *int* 处

`IO::SEEK_END` 从流末尾移动 *int* (需要一个负数 *int*)

`IO::SEEK_SET` 移动到绝对地址 *int* 处

```
f = File.new("testfile")
f.seek(-13, IO::SEEK_END) → 0
f.readline → "And so on...\n"
```

stat

io.stat → stat

以 `File::Stat` 类型对象返回 *io* 的状态信息。

```
f = File.new("testfile")
s = f.stat
"%o" % s.mode      →      "100644"
s.blksize          →      4096
s.atime            →      Thu Aug 26 22:37:41 CDT 2004
```

sync*io.sync* → true 或 false

返回 *io* 的当前“同步模式”。当同步模式为真时，所有输出将立即传递给底层操作系统，Ruby 不会在内部缓存任何输出数据。参见 `IO#fsync`。

```
f = File.new("testfile")
f.sync      →      false
```

sync=*io.sync = bool* → true 或 false

设置“同步模式”为 true 或 false。当同步模式为真时，所有输出将立即传递给底层操作系统，Ruby 不会在内部缓存任何输出数据。返回新状态。参见 `IO#fsync`。

```
f = File.new("testfile")
f.sync = true
```

sysread*io.sysread(int <, buffer >)* → string

1.8.

使用底层的读函数从 *io* 中读 *int* 个字节，并以字符串返回。如果提供了 *buffer* 参数（一个 `String`），输入将直接读入其中。不要和 *io* 读取数据的其他函数相混，否则会得到意想不到的结果。遇到错误将引发 `SystemCallError`，到文件结尾将引发 `EOFError`。

```
f = File.new("testfile")
f.sysread(16)      →      "This is line one"
str = "cat"
f.sysread(10, str) →      "\nThis is 1"
str              →      "\nThis is 1"
```

sysseek*io.sysseek(offset, whence=SEEK_SET)* → int

1.8.

根据 *whence* 的值，使流指针移动到给定的偏移量 *offset*（关于 *whence* 的可能值参见 `IO#seek`）。返回文件的新偏移。

```
f = File.new("testfile")
f.sysseek(-13, IO::SEEK_END) →      53
f.sysread(10)                 →      "And so on."
```

syswrite*io.syswrite(string)* → int

使用底层的写函数将给定的字符串写入到 *io* 中。返回写入的字节数。不要和 *io* 的其他写函数混淆，否则会有意想不到的结果。遇到错误将引发 `SystemCallError`。

```
f = File.new("out", "w")
f.syswrite("ABCDEF") → 6
```

tell *o.tell* → *int*

与 `IO#pos` 同义。

to_i *io.to_i* → *int*

与 `IO#fileno` 同义。

to_io *io.to_io* → *io*

返回 *io*。

tty? *io.tty?* → `true` 或 `false`

与 `IO#isatty` 同义。

ungetc *io.ungetc(int)* → `nil`

将一个字符退还给 *io*，以使随后启用了缓冲的读操作再一次返回它。在读操作之前，只能退还一个字符（也就是说，你只能读取被退回的几个字符中的最后一个）。

对未使用缓冲的读取无效（例如 `IO#sysread`）。

```
f = File.new("testfile") → #<File:testfile>
c = f.getc → 84
f.ungetc(c) → nil
f.getc → 84
```

write *io.write(string)* → *int*

将给定的字符串写入 *io*。流必须以写模式打开。如果参数不是字符串，使用 `to_s` 函数将之转换成字符串。返回写入的字节数。

```
count = STDOUT.write( "This is a test\n" )
puts "That was #{count} bytes of data"
```

输出结果：

```
This is a test
That was 15 bytes of data
```

Module Kernel

类 Object 包含了 Kernel 模块，所以任意 Ruby 对象都包含其函数。从第 567 页的类 Object 开始有 Kernel 的实例方法的文档。本节只阐述它的模块方法。调用这些方法可以不写接受者，因此可以以函数形式调用。

模块函数

| | |
|--------------|---|
| Array | Array(<i>arg</i>) → <i>array</i> |
|--------------|---|

以一个数组返回参数 *arg*。首先调用 *arg.to_array*，然后调用 *arg.to_a*。如果都失败，则创建一个只包含一个元素 *arg* 的数组（除非 *arg* 为 nil）。

```
Array(1..5) → [1, 2, 3, 4, 5]
```

| | |
|--------------|---|
| Float | Float(<i>arg</i>) → <i>float</i> |
|--------------|---|

转换 *arg* 成浮点数，并返回。数值类型将被直接转换，其他类型则使用 1.8. *arg.to_f* 转换。在 Ruby 1.8 中，转换 nil 将导致 TypeError。

```
Float(1) → 1.0
Float("123.456") → 123.456
```

| | |
|----------------|---|
| Integer | Integer(<i>arg</i>) → <i>int</i> |
|----------------|---|

将 *arg* 转换成 Fixnum 或 Bignum。数值类型将被直接转换（浮点数将被截断）。如果 *arg* 更是一个字符串，开头可以有基数指示符（0, 0b 和 0x）。对其他类型将调用 *to_int* 和 *to_i*。这种行为和 String#to_i 不同。

```
Integer(123.999) → 123
Integer("0x1a") → 26
Integer(Time.new) → 1116425264
Integer(nil) → 0
```

| | |
|---------------|---|
| String | String(<i>arg</i>) → <i>string</i> |
|---------------|---|

通过调用 *to_s* 方法将 *arg* 转换成一个 String。

```
String(self) → "main"
String(self.class) → "Object"
String(123456) → "123456"
```

| | |
|---------------|------------------------------|
| `(反引号) | `cmd` → <i>string</i> |
|---------------|------------------------------|

在子 shell 中运行 *cmd*，并返回其标准输出。第 89 页描述的内置语法 %x{...} 使用了此方法。设置 \$? 为进程的状态。

```
'date'           → "Thu Aug 26 22:38:08 CDT 2004\n"
`ls testdir`.split[1] → "main.rb"
`echo oops && exit 99` → "oops\n"
$?.exitstatus      → 99
```

abort

abort
abort(msg)

- 1.8 立即终止执行，退出码为 1。如果带字符串类型参数，则在程序终止前将输出该串到标准错误。

at_exit

at_exit { block } → proc

将 *block* 转换成一个 Proc 对象（并且在调用处绑定该 *block*），并注册该 *block* 以使之在程序退出时运行。如果注册了多个函数，则它们以和注册时相反的顺序执行。

```
def do_at_exit(str1)
  at_exit { print str1 }
end
at_exit { puts "cruel world" }
do_at_exit("goodbye ")
exit
```

输出结果：

```
goodbye cruel world
```

autoload

autoload(name, file_name) → nil

注册 *file_name*，以使得当第一次访问模块 *name*（可以是 String 或符号）时（使用 Kernel.require）装载 *file_name*。

```
autoload(:MyModule, "/usr/local/lib/modules/my_module.rb")
```

- 1.8 在 Ruby 1.8 之前，*name* 参数处于顶层名字空间。在 Ruby 1.8 中，新方法 Module.autoload 允许在某个名字空间内定义自动转载钩子。在下面的代码中，Ruby 1.6 会在访问:xxx 时装载 xxx.rb，而 Ruby 1.8 是在访问 x::xxx 时装载该文件。

```
module X
  autoload :XXX, "xxx.rb"
end
```

注意 xxx.rb 需要在正确的名字空间中定义类。例如在这个例子中 xxx.rb 应当包含：

```
class X::XXX
  # ...
end
```

autoload?autoload?(*name*)→*file_name* 或 nil

返回当顶级上下文中的字符串或符号 *name* 被访问时将被自动装载的文件的名字。如果没有相关联的自动装载文件，则返回 nil。

```
autoload(:Fred, "module_fred") → nil
autoload?(:Fred) → "module_fred"
autoload?(:Wilma) → nil
```

binding

binding→*a_binding*

返回一个 Binding 对象，该对象描述了此函数被调用时绑定的变量和方法。当调用 eval 时，可以使用这个对象，以其作为执行求解后命令的环境。关于 Binding 类的更多描述请参考第 444 页。

```
def get_binding(param)
  return binding
end
b = get_binding("hello")
eval("param", b) → "hello"
```

block_given?

block_given?→true 或 false

如果在当前上下文中进行 yield，能够执行到一个 block，就返回 true。

```
def try
  if block_given?
    yield
  else
    "no block"
  end
end
try → "no block"
try { "hello" } → "hello"
try do "hello" end → "hello"
```

callcc

callcc{|*cont*|*block*}→*obj*

产生一个 Continuation 对象，并传递给相关联的 block。执行 *cont.call* 将导致 callcc 返回（即执行至 block 的结尾）。callcc 的返回值为 block 的值或者传递给 *cont.call* 的值。关于更多细节，参见第 448 页的 Continuation 一节。另一种展开调用栈的机制，可以参见 Kernel.throw。

caller

caller(<*int*>)→*array*

返回当前的执行栈——一个字符串数组，每个元素的格式为 *file:line* 或 *file:line:in 'method'*。可选的 *int* 参数决定在结果中要忽略的调用栈的最初几项。

```

def a(skip)
  caller(skip)
end
def b(skip)
  a(skip)
end
def c(skip)
  b(skip)
end
c(0) → ["prog:2:in `a'", "prog:5:in `b'", "prog:8:in `c'",
          "prog:10"]
c(1) → ["prog:5:in `b'", "prog:8:in `c'", "prog:11"]
c(2) → ["prog:8:in `c'", "prog:12"]
c(3) → ["prog:13"]

```

catch**catch(*symbol*) { *block* } → *obj***

`catch` 执行它的 `block`。如果执行过程中遇到 `throw`, Ruby 将在栈中向上搜索标记 (`tag`) 与 `throw` 的 `symbol` 相对应的 `catch` `block`。如果找到, 该 `block` 将被终止, 并且 `catch` 返回传给 `throw` 的值。如果没有遇到 `throw` 语句, `block` 将正常终止, 且 `catch` 的返回值是执行最后一个表达式的值。`catch` 表达式可以嵌套, 且 `throw` 不一定在同一作用域中。

```

def routine(n)
  puts n
  throw :done if n <= 0
  routine(n-1)
end
catch(:done) { routine(4) }

```

输出结果:

```

4
3
2
1
0

```

chomp**chomp(<*rs*>) → *\$_* 或 *string***

除了当 `chomp` 没有改变`$_`时不会执行赋值操作外, `chomp` 与`$_=$_.chomp(rs)`等价。参见第 610 页的 `String#chomp`。

```

$_ = "now\n"
chomp → "now"
chomp "ow" → "n"
chomp "xxx" → "n"
$_ → "n"

```

chomp!**chomp!(<*rs*>) → *\$_* 或 *nil***

等价于 `$_.chomp!(rs)`。参见 `String#chomp!`

```
$_. = "now\n"
chomp!      →      "now"
$_          →      "now"
chomp! "x"  →      nil
$_          →      "now"
```

chop**chop → string**

除了当 `chop` 不做任何操作时不会改变`$_`且不会返回 `nil` 之外，`chop` 几乎与 `($.dup).chop!` 等价。参见第 610 页的 `String#chop!`。

```
$_ = a = "now\r\n"
chop      →      "now"
$_          →      "now"
chop      →      "no"
chop      →      "n"
chop      →      ""
a          →      "now\r\n"
```

chop!**chop! → \$_ 或 nil**

等价于`$.chop!`。

```
$_ = a = "now\r\n"
chop!     →      "now"
chop!     →      "no"
chop!     →      "n"
chop!     →      ""
chop!     →      nil
$_          →      ""
a          →      ""
```

eval**eval(string <, binding <, file <, line >>>) → obj**

执行 `string` 中的 Ruby 表达式。如果带有 `binding` 参数，则在其上下文中执行计算。给定的绑定参数可以是 `Binding` 对象或 `Proc` 对象。如果带有 `file` 和 `line` 参数，当遇到语法错误时，它们将被用来报告错误。

```
def get_binding(str)
  return binding
end
str = "hello"
eval "str + ' Fred'"           →      "hello Fred"
eval "str + ' Fred'", get_binding("bye") →      "bye Fred"
```

1.8 在 Ruby 1.8 中，在 `eval` 内赋值的局部变量，仅当它在执行 `eval` 之前的外部作用域中曾定义过，才会在 `eval` 之后仍然有效。在这种方式下，`eval` 和 `block` 有相同的作用域规则。

```
a = 1
eval "a = 98; b = 99"
puts a
puts b
```

输出结果：

```
98
prog.rb:4: undefined•local variable or method `b' for
main:Object (NameError)
```

exec**exec(command <, args >)**

通过运行给定的外部命令替换当前的进程。如果 `exec` 只有一个参数，那么在执行之前 Shell 将对该参数进行扩展。如果 `command` 含有换行符或 *?{}[]<>()~\&|\\$;`中的任意一个字符，或者如果在 Windows 下 `command` 是 shell 的内部命令，那么 `command` 将在 shell 中运行。在 Unix 系统上，Ruby 通过添加 `sh -c` 到 `command` 的头部来实现。在 Windows 上，它使用 RUBYSHELL 或者 COMSPEC 指定的 shell。

如果有多个参数，那么第二个参数和随后的参数将作为参数传递给 `command`，且不执行 shell 扩展。如果第一个参数是含有两个元素的数组，那么第一个元素是将被执行的命令，第二个参数为 `argv[0]` 的值，用来在进程列表中显示进程的名字。在 MSDOS 环境中，命令将在子 shell 中运行；在其他系统中，`exec(2)` 系统调用中的一个将被使用，所以运行的命令可能会从原程序中继承一些环境变量（包括打开的文件描述符）。如果不能执行 `command` 命令，则引发 `SystemCallError`（通常是 `Errno::ENOENT`）。

```
exec "echo *" # echoes list of files in current directory
# never get here
exec "echo", "*" # echoes an asterisk
# never get here
```

exit**exit(true|false|status=1)**

1.8.

终止 Ruby 脚本的运行。如果是在一个异常处理程序的作用域中被调用，则引发 `SystemExit` 异常，此异常可以被捕获。否则使用 `exit(2)` 终止进程的运行。可选的参数被用来返回一个状态码给调用环境。如果参数为 `true`，则退出状态为 0。如果参数为 `false`，或者没有参数，则退出状态为 1，否则以给定的退出状态码退出。注意在 Ruby 1.8 中默认的退出状态值从 -1 变成了 +1。

```
fork { exit 99 }
Process.wait
puts "Child exits with status: #{$?.exitstatus}"
begin
  exit
  puts "never get here"
rescue SystemExit
  puts "rescued a SystemExit exception"
end
puts "after begin block"
```

输出结果：

```
Child exits with status: 99
rescued a SystemExit exception
after begin block
```

在即将退出之前，Ruby 会执行任意 `at_exit` 函数，以及任意对象清理函数（参见第 578 页的 `ObjectSpace`）。

```
at_exit { puts "at_exit function" }
ObjectSpace.define_finalizer("string", lambda { puts "in finalizer" })
exit
```

输出结果：

```
at_exit function
in finalizer
```

| | |
|-------|---|
| exit! | <code>exit!(true false status=1)</code> |
|-------|---|

1.8 和 `Kernel.exit` 类似，但是将跳过异常处理、`at_exit` 函数和对象清理函数。

| | |
|------|---|
| fail | <code>fail</code>
<code>fail(message)</code>
<code>fail(exception<, message<, array>>)</code> |
|------|---|

与 `Kernel.raise` 同义。

| | |
|------|--|
| fork | <code>fork<{block}> → int 或 nil</code> |
|------|--|

创建一个子进程。如果提供了 `block`，则将在子进程中运行该 `block`，并且子进程的退出状态为 0。否则 `fork` 调用将返回两次，一次在父进程中，返回子进程的进程 ID，一次在子进程中，返回 `nil`。子进程可以调用 `Kernel.exit!` 来避免调用任意的 `at_exit` 函数。父进程需要用 `Process.wait` 来获取子进程的退出状态，或者使用 `Process.detach` 来忽略它们的退出状态；否则操作系统的僵死进程将不断增加。

```
fork do
  3.times { |i| puts "Child: #{i}" }
end
3.times { |i| puts "Parent: #{i}" }
Process.wait
```

输出结果：

```
Child: 0
Parent: 0
Child: 1
Parent: 1
Child: 2
Parent: 2
```

| | |
|---------------|--|
| format | format(<i>format_string</i> <, <i>arg</i> >*) → <i>string</i> |
|---------------|--|

与 Kernel.printf 同义。

| | |
|-------------|--|
| gets | gets(<i>separator</i>=\$/) → <i>string</i> 或 nil |
|-------------|--|

返回 ARGV (或 \$*) 给出的文件列表中文件的下一行，并赋给 \$_，如果命令行未提供文件，则返回标准输入的下一行。如果到达了文件结尾，则返回 nil。可选的参数是记录分割符。分隔符包含在每个记录的内容中。如果分隔符为 nil，则读取全部内容；如果分割符长度为 0，则一次读取一段，其中的段由两个连续的换行符分割。如果 ARGV 中有多个文件，gets(nil) 一次读取一个文件。

```
ARGV << "testfile"
print while gets
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

使用 \$_ 作为隐含参数的编程风格，逐渐被 Ruby 社区所弃用。

| | |
|-------------------------|---------------------------------|
| global_variables | global_variables → array |
|-------------------------|---------------------------------|

返回全局变量组成的一个数组。

```
global_variables.grep /std/ → ["$stdout", "$stdin", "$stderr"]
```

| | |
|-------------|---|
| gsub | gsub(<i>pattern</i>, <i>replacement</i>) → <i>string</i> |
| | gsub(<i>pattern</i>) { <i>block</i> } → <i>string</i> |

除了在发生替换时会更新 \$_ 的值之外，与 \$_.gsub(...) 等价。

```
$_ = "quick brown fox"
gsub /[aeiou]/, '*' → "q**ck br*wn f**x"
$_ → "q**ck br*wn f**x"
```

| | |
|--------------|--|
| gsub! | gsub!(<i>pattern</i>, <i>replacement</i>) → <i>string</i> 或 nil |
| | gsub!(<i>pattern</i>) { <i>block</i> } → <i>string</i> 或 nil |

等价于 \$_.gsub!(...).

```
$_ = "quick brown fox"
gsub! /cat/, '*' → nil
$_ → "quick brown fox"
```

| | |
|------------------|---------------------------------|
| iterator? | iterator? → true 或 false |
|------------------|---------------------------------|

与 Kernel.block_given? 同义，但是已过时。

| | |
|---------------|--------------------------------|
| lambda | lambda { block } → proc |
|---------------|--------------------------------|

根据给定的 `block` 创建一个新的过程对象。对于使用 `lambda` 和使用 `Proc.new` 创建过程对象的区别，请参见第 357 页。注意现在推荐尽量使用 `lambda` 而不是 `proc`。

```
prc = lambda { "hello" }
prc.call → "hello"
```

| | |
|-------------|---|
| load | load(file_name, wrap=false) → true |
|-------------|---|

装载和执行 Ruby 程序文件 `file_name`。如果文件名不是绝对路径，则 Ruby 将在 \$: 列出的库路径中搜索。如果可选的参数 `wrap` 为 `true`，被装载的脚本将在一个匿名的模块中执行，以保护调用程序的全局名字空间。无论如何，被装载文件的任何局部变量都不会被传播到装载环境中。

| | |
|------------------------|--------------------------------|
| local_variables | local_variables → array |
|------------------------|--------------------------------|

返回当前局部变量的名字。

```
fred = 1
for i in 1..10
  #
end
local_variables → ["fred", "i"]
```

注意局部变量和绑定相关联。

```
def fred
  a = 1
  b = 2
  binding
end
freds_binding = fred
eval("local_variables", freds_binding) → ["a", "b"]
```

| | |
|-------------|-----------------------|
| loop | loop { block } |
|-------------|-----------------------|

重复执行 `block`。

```
loop do
  print "Input: "
  break if (line = gets).nil? or (line =~ /^[qQ]\/)
  #
end
```

| | |
|------|---|
| open | <code>open(name <, modestring <, permission >>) → io 或 nil</code> |
| | <code>open(name <, modestring <, permission >>) { io block } → obj</code> |

创建一个连接到给定流、文件或子进程的 `IO` 对象。

如果 `name` 不是以管道符 (`|`) 开头，那么 Ruby 把它当作要打开的文件的名字，默认的打开模式是 “`r`”（参见第 504 页有效模式表）。如果需要创建文件，那么它的初始权限可以通过第三个整数参数来设置。如果提供了第三个参数，则使用底层的 `open(2)` 而不是 `fopen(3)` 调用打开文件。

如果带有 `block`，则以 `IO` 对象为参数调用该 `block`，当 `block` 终止时，`IO` 对象将自动被关闭。在这种情况下返回 `block` 的值。

如果 `name` 以管道符打头，则将创建一个子进程，并通过一个管道连接到调用者。可以使用返回的 `IO` 对象向子进程的标准输出写或从其标准输入读。如果 `|` 字符后面的命令是一个减号，那么 Ruby 将创建一个子进程，且此子进程将连接到父进程。在子进程中，`open` 调用返回 `nil`。如果命令不是 “`-`”，那么子进程将运行该命令。如果有多个 `block` 关联到了 `open("|-")`，那么该 `block` 将被运行两次——一次在父进程中一次在子进程中。在父进程中 `block` 的参数是一个 `IO` 对象，而在子进程中参数为 `nil`。父进程的 `IO` 对象将被关联到子进程的 `STDIN` 和 `STDOUT`。在 `block` 结束时子进程也将终止。

```
open("testfile") do |f|
  print f.gets
end
```

输出结果：

```
This is line one
```

Open 一个子进程并读取其输出。

```
cmd = open("|date")
print cmd.gets
cmd.close
```

输出结果：

```
Thu Aug 26 22:38:10 CDT 2004
```

Open 一个运行相同 Ruby 程序的子进程。

```
f = open("|-", "w+")
if f.nil?
  puts "in Child"
  exit
else
  puts "Got: #{f.gets}"
end
```

输出结果:

```
Got: in Child
```

使用接受 I/O 对象的 block 来打开一个子进程。

```
open("|-") do |f|
  if f.nil?
    puts "in Child"
  else
    puts "Got: #{f.gets}"
  end
end
```

输出结果:

```
Got: in Child
```

p

$p(<obj>^*) \rightarrow \text{nil}$

对每个对象，输出 `obj.inspect` 后跟当前输出记录分隔符到程序的标准输出。

参见第 716 页的 PrettyPrint 库。

```
S = Struct.new(:name, :state)
s = S['dave', 'TX']
p s
```

输出结果:

```
#<struct S name="dave", state="TX">
```

print

$\text{print}(<obj>^*) \rightarrow \text{nil}$

依次打印每个对象到 STDOUT。如果输出域分隔符 (`$,`) 不为 nil，那么它的内容将出现在每个域之间。如果输出记录分隔符 (`$\n`) 不为 nil，那么将添加它到输出末尾。如果没有任何参数，则打印`$_`。如果对象不是字符串，则调用其 `to_s` 方法把它先转换成字符串。

```
print "cat", [1,2,3], 99, "\n"
$, = ","
$\ = "\n"
print "cat", [1,2,3], 99
```

输出结果:

```
cat12399
cat, 1, 2, 3, 99
```

printf

$\text{printf}(io, format <, obj>^*) \rightarrow \text{nil}$

$\text{printf}(format <, obj>^*) \rightarrow \text{nil}$

等价于:

```
io.write sprintf(format, obj ...)
```

或

```
STDOUT.write sprintf(format, obj ...)
```

proc *proc { block } → a_proc*

使用给定的 *block* 创建一个新的过程对象。由于 Kernel#lambda 的出现，此方法有点过时了。¹⁸

```
prc = proc {|name| "Goodbye, #{name}" }
prc.call('Dave') → "Goodbye, Dave"
```

putc *putc(int) → int*

等价于 STDOUT.puts(*int*)。

puts *puts(<arg>*) → nil*

等价于 STDOUT.puts(*arg...*)。

raise *raise
raise(message)
raise(exception <, message <, array > >)*

如果没有参数，则引发 \$! 变量保存的异常；如果 \$! 为 nil，则引发 RuntimeError；如果有一个 String 参数，则引发 RuntimeError 异常，并以该字符串作为消息。否则第一个参数需要 Exception 类的名字（或者当发送异常时返回 Exception 的一个对象）。可选的第二个参数设置和异常相关联的消息，而第三个参数是含有调用栈信息的数组。异常将被 begin ... end block 的 rescue 语句所捕获。

```
raise "Failed to create socket"
raise ArgumentError, "No parameters", caller
```

rand *rand(max=0) → number*

使用 *max*₁ = *max*.to_i.abs 将 *max* 转化成一个整数。如果返回的结果为 0，则返回一个大于等于 0.0 且小于 1.0 的一个伪随机浮点数；否则返回一个大于等于 0，小于 *max*₁ 的一个伪随机整数。可以使用 Kernel.srand 来保证每次运行程序时获得不同的随机数。目前 Ruby 使用介于 $2^{19937} - 1$ 之间的一个 Mersenne Twister³ 随机数。

| | | |
|---------------------------------------|---|---------------------------------------|
| <code>srand 1234</code> | → | 0 |
| <code>[rand, rand]</code> | → | [0.191519450163469, 0.49766366626136] |
| <code>[rand(10), rand(1000)]</code> | → | [6, 817] |
| <code>srand 1234</code> | → | 1234 |
| <code>[rand, rand]</code> | → | [0.191519450163469, 0.49766366626136] |

readline *readline(<separator=\$/ >) → string*

除了 readline 会在遇到文件末尾引发 EOFError 之外，与 Kernel.gets 等价。

³ 译注：一种随机数生成算法。

readlines**readlines(< separator=\$/ >) → array**

返回一个数组，该数组包含调用 `Kernel.gets(separator)` 直到遇到文件末尾获得的所有行。

require**require(library_name) → true 或 false**

Ruby 试图装载 `library_name`，如果成功，则返回 `true`。如果文件名不能解析成绝对路径，那么将在`$:`列出的目录中搜索它。如果文件含有`.rb` 扩展，则作为源文件被装入；如果扩展名是`.so`, `.o` 或 `.dll`⁴，Ruby 将把这些共享库作为 Ruby 扩展加以装载。否则，Ruby 会试图将`.rb`, `.so` 等扩展名加到文件名之后。被装载的代码对应的文件名字将被添加到数组`$"`中。如果其名字已经出现在`$"`⁵ 中，则它将不被装载。如果成功装载，则返回 `true`。

```
require 'my-library.rb'
require 'db-driver'
```

1.8.

可以使用 `SCRIPT_LINES__` 常量来获取使用 `require` 读取的源代码。

```
SCRIPT_LINES__ = {}
require 'code/scriptlines'
puts "Files: #{SCRIPT_LINES__.keys.join(', ')}"
SCRIPT_LINES__[ './code/scriptlines.rb' ].each do |line|
  puts "Source: #{line}"
end
```

输出结果：

```
3/8
Files:./code/scriptlines.rb,/Users/dave/ruby1.8/lib/ruby/1.8/rational.rb
Source: require 'rational'
Source:
Source: puts Rational(1,2)*Rational(3,4)
```

scan**scan(pattern) → array****scan(pattern) { block } → \$_**

与调用`$_.scan`等价。参见第 617 页的 `String#scan`。

select**select(read_array <, write_array <, error_array <, timeout >>>) → array 或 nil**

执行底层的 `select` 调用，该函数从输入/输出设备上等待数据。前三个参数是 `IO` 对象数组或者 `nil`。最后一个参数是以秒为单位的超时值，该值可以是 `Integer` 或者 `Float`。本函数等待，直到 `read_array` 中任意一个 `IO` 对象有数据可以读取，或

⁴ 或者当前平台上的任意默认共享库扩展名。

⁵ 因为名字不被转换成绝对路径，所以 `require'a';require'./a'` 将装载 `a.rb` 两次。这是一个备受争论的 bug。

者 `write_array` 中的某个设备其缓冲区已经被完全清空得以继续向其写入数据，或者 `error_array` 中的任意设备出现错误。如果这些条件有一个或多个满足，那么调用将返回一个含有三个元素的数组，每个数据元素均由已就绪的 `IO` 对象所组成的数组。否则，如果状态在 `timeout` 时间内没有变化，则返回 `nil`。如果所有的参数都是 `nil`，则当前线程将永远进入睡眠。

```
select( [STDIN], nil, nil, 1.5 ) → [[#<IO:0x1cfac>], [], []]
```

| | |
|-----------------------------|--|
| <code>set_trace_func</code> | <code>set_trace_func(proc) → proc</code>
<code>set_trace_func(nil) → nil</code> |
|-----------------------------|--|

设置 `proc` 为追踪处理函数，如果参数为 `nil`，则禁用追踪。`proc` 接纳 6 个参数：事件名、文件名、行号、对象 ID、绑定和类名。每当事件发生时调用 `proc`。事件有 `c-call`（调用一个 C 语言例程）、`c-return`（从一个 C 语言例程返回）、`call`（调用 Ruby 方法）、`class`（开始类或模块定义）、`raise`（引发一个异常）和 `return`（从 Ruby 方法返回）。追踪在 `proc` 的上下文中将被禁用。

更多信息请参考第 412 页的例子。

| | |
|--------------------|--|
| <code>sleep</code> | <code>sleep(numeric=0) → fixnum</code> |
|--------------------|--|

挂起当前线程 `numeric` 秒（可以是 `Float` 以表示浮点的秒数）。返回实际睡眠的秒数（舍入后的整数），如果线程被 `SIGALRM` 中断或者另外一个线程调用了 `Thread#run`，则返回的秒数可能小于请求的描述。参数为 0 将导致线程无限期睡眠。

```
Time.now → Thu Aug 26 22:38:10 CDT 2004
sleep 1.9 → 2
Time.now → Thu Aug 26 22:38:12 CDT 2004
```

| | |
|--------------------|---|
| <code>split</code> | <code>split(<pattern <, limit > >) → array</code> |
|--------------------|---|

等价于 `$.split(pattern, limit)`。参见第 619 页的 `String#split`。

| | |
|----------------------|--|
| <code>sprintf</code> | <code>sprintf(format_string <, arguments >*) → string</code> |
|----------------------|--|

应用 `format_string` 到其他参数，并返回结果字符串。在格式字符串内部，除了格式序列的其他任意字符将直接被拷贝到结果中。

一个格式序列由百分号，可选的标志（flag）、宽度和精度描述符，以及一个域类型字符组成。域类型控制对相应的 `sprintf` 参数如何解释，而标志可以修改此解释。下页的表 27.7 列出了标志字符，表 27.8 列出了字段类型字符。

可选的字段宽度是一个整数，后可以跟一个点和精度。宽度指明了将该字段写入到结果字符串占用的最小字符个数。对于数值字段，精度控制要显示的小数的个数。对字符串字段，精度指明了将要从字符串中拷贝的字符的最大个数（这样，格式序列 `%10.10s` 将总是拷贝 10 个字符到结果中）。

| | |
|---|----------------------|
| <code>sprintf("%d %04x", 123, 123)</code> | → "123 007b" |
| <code>sprintf("%08b '%4s'", 123, 123)</code> | → "01111011 '123'" |
| <code>sprintf("%1\$*2\$s %2\$d %1\$s", "hello", 8)</code> | → "hellohello8hello" |
| <code>sprintf("%1\$*2\$s %2\$d", "hello", -8)</code> | → "hellooooo8" |
| <code>sprintf("%+g:% g:%-g", 1.23, 1.23, 1.23)</code> | → "+1.23:-1.23:1.23" |

strand`strand(<number>) → old_seed`

设置伪随机数生成器的种子为 `number.to_i.abs`。如果 `number` 被省略了或者为 0，则用当前时间、进程 ID 和一个序号为基础生成种子（如果未先调用 `strand` 就调用 `Kernel.rand`，则也用这种方法来生成种子，不过不用序号）。通过设置种子为一个预知的值，在测试过程，使用 `rand` 的脚本具有确定性。先前的种子值将被返回。参见第 527 页的 `Kernel.rand`。

sub`sub(pattern, replacement) → $_
sub(pattern) { block } → $_`

除了在发生替换时会更新 `$_` 外，等价于 `$_.sub(args)`。

sub!`sub!(pattern, replacement) → $_ 或 nil
sub!(pattern) { block } → $_ 或 nil`

等价于 `$_.sub!(args)`。

syscall`syscall(fixnum <, args >*) → int`

调用 ID 为 `fixnum` 的操作系统函数。参数必须是 `String` 对象或者是可以用本机 `long` 类型表示的 `Integer` 对象。至多可以传递 9 个参数。以 `fixnum` 为 ID 的函数是系统相关的。在某些 Unix 系统上，该数字可以从头文件 `syscall.h` 中获得。

```
syscall 4, 1, "hello\n", 6 # '4' is write(2) on our system
```

输出结果：

```
hello
```

system`system(command <, args >*) → true 或 false`

在子 shell 中执行 `command`，如果找到命令并成功运行，则返回 `true`，否则返

表 27.7 sprintf 标志字符

| 标 志 | 应 用 到 | 含 义 |
|------------|--------------|---|
| (空
格) | bdEefGgiouXx | 在正数前面留一个空格 |
| Digit\$ | 全 部 | 为这个字段指定绝对的参数个数。在 sprintf 字符串中不能同时使用绝对和相对参数个数 |
| # | beEfgGoxX | 使用其他格式。对 b, o, X 和 x, 在结果前面分别加 b, 0, 0X, 0x。对于 E, e, f, G 和 g, 即使没有小数也输出小数点。对于 G 和 g, 不去除尾部的零 |
| + | bdEefGgiouXx | 在正数的前面加一个加号 |
| - | 全 部 | 使结果左对齐 |
| 0 (零) | bdEefGgiouXx | 不用空格而用 0 补齐 |
| * | 全 部 | 使用下一个参数作为字段的宽度。如果是负数, 则令结果左对齐。如果星号后跟一个数和一个美元符, 则使用指定的参数作为宽度 |

回 `false`。错误状态保存在 `$?` 中。参数的处理方式和第 521 页的 `Kernel.exec` 一样。如果无法执行 `command`, 则引发 `SystemCallError` (通常是在 `Errno::ENOENT`)。

```
system("echo *")
system("echo", "*")
```

输出结果:

```
config.h main.rb
*
```

| | |
|------|--|
| test | <code>test(cmd, file1 <, file2 >) → obj</code> |
|------|--|

使用数值 `cmd` 在 `file1` 或 `file1` (第 533 页的表 27.9) 和 `file2` (表 27.10) 上执行各种测试。

| | |
|-------|---|
| throw | <code>throw(symbol <, obj >)</code> |
|-------|---|

将控制权转移到等待 `symbol` 的活跃 `catch block` 的结尾。如果没有对应此符号的 `catch`, 则引发 `NameError`。如果有第二个参数, 则该参数为 `catch block` 提供了返回值; 如果没有第二个参数, 则返回 `nil`。参见第 519 页关于 `Kernel.catch` 的例子。

表 27.8 sprintf 域类型

| 域 转 换 | |
|-------|---|
| b | 把参数转换成二进制数 |
| c | 参数是单个字符的数值码 |
| d | 把参数转换成十进制数 |
| E | 等价于 e, 但使用大写的 E 来表示指数 |
| e | 把浮点数参数转换成指数形式, 使得小数点前只有一个数字。精度确定了小数的个数 (默认为 6) |
| f | 把浮点数参数转换成 [-]ddd.ddd, 精度确定了小数点后面数字的个数 |
| G | 等价于 g, 但是使用大写的 E 表示指数 |
| g | 如果指数小于 -4 或大于等于精度, 则把浮点数转换成指数形式, 否则使用 d.dddd 形式 |
| i | 等价于 d |
| o | 把参数转换成八进制数 |
| 1.8 | p <i>argument.inspect</i> 的值 |
| s | 参数是要被替换的字符串。如果格式序列含有精度, 那么至多有精度那么多字符被拷贝 |
| u | 把参数当作是无符号十进制数 |
| X | 把参数转换成十六进制数, 使用大写字母。负数前面将显示两个点 (表示前面有无限个 FF) |
| x | 把参数转换成十六进制数。负数前面将显示两个点 (表示前面有无限个 FF) |

| | |
|-----------|---|
| trace_var | trace_var(<i>symbol</i> , <i>cmd</i>) → nil
trace_var(<i>symbol</i>) { <i>val</i> <i>block</i> } → nil |
|-----------|---|

控制对全局变量赋值操作的追踪。参数 *symbol* 表示被追踪的变量（或者是字符串名或者是符号标志符）。每当对变量赋值时, *cmd* (可以是字符串也可以是 Proc 对象) 或关联的 *block* 将被执行, 并以变量的新值作为参数。仅追踪显式的赋值。

参见 Kernel.untrace_var。

```
trace_var :$_, lambda { |v| puts "$_ is now '#{v}'" }
$_ = "hello"
sub(/ello/, "i")
$_ += " Dave"
```

输出结果:

```
$ _ is now 'hello'
$_ is now 'hi Dave'
```

表 27.9 参数只有一个的文件测试

| 标 志 描 述 | 返 回 值 |
|---|---------------|
| ?A <i>file1</i> 的最后访问时间 | Time |
| ?b 如果 <i>file1</i> 是块设备，则为 true | true 或 false |
| ?c 如果 <i>file1</i> 是字符设备，则为 true | true 或 false |
| ?C <i>file1</i> 的最后修改时间 | Time |
| ?d 如果 <i>file1</i> 存在且是目录，则为 true | true 或 false |
| ?e 如果 <i>file1</i> 存在，则为 true | true 或 false |
| ?f 如果 <i>file1</i> 存在且是普通文件，则为 true | true 或 false |
| ?g 如果 <i>file1</i> 设置了 setgid 位，则为 true (在 NT 下为 false) | true 或 false |
| ?G 如果 <i>file1</i> 存在且其所属的组等于调用者的组，则为 true | true 或 false |
| ?k 如果 <i>file1</i> 存在且设置了粘滞位 (sticky)，则为 true | true 或 false |
| ?l 如果文件 <i>file1</i> 存在且是符号链接，则为 true | true 或 false |
| ?M <i>file1</i> 的最后更改时间 | Time |
| ?o 如果 <i>file1</i> 存在且被调用者的有效 UID 拥有，则为 true | true 或 false |
| ?O 如果 <i>file1</i> 存在且被调用者的 UID 拥有，则为 true | true 或 false |
| ?p 如果 <i>file1</i> 存在且是 fifo ⁶ ，则为 true | true 或 false |
| ?r 如果 <i>file1</i> 存在且可以被调用者的有效 UID/GID 读，则为 true | true 或 false |
| ?R 如果 <i>file1</i> 存在且可以被调用者的真实 UID/GID 读，则为 true | true 或 false |
| ?s 如果 <i>file1</i> 的大小不为 0，则返回其大小，否则返回 nil | Integer 或 nil |
| ?S 如果 <i>file1</i> 存在且是套接字，则为 true | true 或 false |
| ?u 如果 <i>file1</i> 设置了 setuid 位，则为 true | true 或 false |
| ?w 如果 <i>file1</i> 存在且可以被有效 UID/GID 写，则为 true | true 或 false |
| ?W 如果 <i>file1</i> 存在且可以被真实 UID/GID 写，则为 true | true 或 false |
| ?x 如果 <i>file1</i> 存在且可以被有效 UID/GID 执行，则为 true | true 或 false |
| ?X 如果 <i>file1</i> 存在且可以被真实 UID/GID 执行，则为 true | true 或 false |
| ?z 如果 <i>file1</i> 存在且长度为 0，则为 true | true 或 false |

表 27.10 参数有两个的文件测试

| 标 志 描 述 |
|--|
| ?- 如果 <i>file1</i> 是到 <i>file2</i> 的硬链接，则为 true |
| ?= 如果 <i>file1</i> 和 <i>file2</i> 的修改时间相同，则为 true |
| ?< 如果 <i>file1</i> 的修改时间在 <i>file2</i> 的修改时间之前，则为 true |
| ?> 如果 <i>file1</i> 的修改时间在 <i>file2</i> 的修改时间之后，则为 true |

⁶ 译注，又成为“具名管道”

trap

trap(signal, proc) → obj
trap(signal) { block } → obj

参见第 604 页的 Singal 模块。

untrace_var

untrace_var(symbol <, cmd >) → array 或 nil

删除全局变量上指定的追踪命令并返回 *nil*。如果没提供命令参数，则删除该变量上的所有追踪命令，并返回由实际被删除的命令组成的数组。

warn

warn msg

1.8 将给定的消息输出到 STDERR（除非 \$VERBOSE 为 *nil*，这可能因为指定了 -w0 命令行选项）。

```
warn "Danger, Will Robinson!"
```

输出结果：

```
Danger, Will Robinson!
```

Module Marshal

序列化库将 Ruby 对象集合转换成字节流，以使它们可以存储在当前脚本之外。随后这些数据可以再次被读取，并重新构造原来的对象。第 414 页有对序列化的描述。也请参考第 758 页的 YAML 库。

除了对象信息本身，被序列化的数据还有主、次版本号。通常的用法中，序列化仅能装载那些具有相同主版本号且具有相同或较低次版本号的数据。如果设置了 Ruby 的“verbose”标志（通常使用 `-d`, `-v`, `-w` 或 `-verbose`），那么主、次版本号必须精确匹配。序列化的版本号和 Ruby 的版本号是独立的。通过提取被序列化的数据的前两个字节可以获得版本号。

```
str = Marshal.dump("thing")
RUBY_VERSION → "1.8.2"
str[0] → 4
str[1] → 8
```

有些对象不能转储：如果将被转储的对象包含绑定、过程或方法对象，类 `IO` 的实例或者单例对象，又或者如果试图转储匿名类或模块，则将引发 `TypeError`。
1.8

如果你的类有特殊的序列化需求（例如假设你想以特定的格式来序列化），或者如果它含有其他情况下无法序列化的对象，那么你可以实现自定义的序列化策略。在 Ruby 1.8 之前，需要定义 `_dump` 和 `_load`。

Ruby 1.8 使用实例方法 `marshal_dump` 和 `marshal_load` 为实现自定义的序列化提供了一个更灵活的接口：如果将要序列化的对象响应 `marshal_dump`，则调用该方法而不是 `_dump`。`marshal_dump` 可以返回任意类（不止是 `String`）的对象。一个实现了 `marshal_dump` 的类也必须实现 `marshal_load`，该方法作为新分配的对象的实例方法被调用，并以 `marshal_dump` 最初创建的对象为参数。
1.8

下面的代码使用这一新的框架将一个 `Time` 对象存储到其序列化对象中。当装载的时候，该新对象被传递给 `marshal_load`，此函数将其转换成可打印的形式，并将结果存储在一个实例变量中。

```
class TimedDump
  attr_reader :when_dumped
  def marshal_dump
    Time.now
  end
  def marshal_load(when_dumped)
    @when_dumped = when_dumped.strftime("%I:%M%p")
  end
end
```

```
t = TimedDump.new
t.when_dumped → nil

str = Marshal.dump(t)

newt = Marshal.load(str)
newt.when_dumped → "10:38PM"
```

模块常量

| | |
|---------------|---------------------|
| MAJOR_VERSION | marshal 格式版本号的主版本号。 |
| MINOR_VERSION | marshal 格式版本号的次版本号。 |

模块方法

| | |
|------|--|
| dump | dump(<i>obj</i> < , <i>io</i> > , <i>limit</i> =-1) → <i>io</i> |
|------|--|

序列化 *obj* 及其所有派生对象。如果带有 *io* 参数，序列化的数据将被写入其中；否则数据将作为 String 返回。如果带有 *limit*，则对子对象的遍历深度限于此值。如果 *limit* 是负数，则不检查遍历深度。

```
class Klass
  def initialize(str)
    @str = str
  end
  def say_hello
    @str
  end
end

o = Klass.new("hello\n")
data = Marshal.dump(o)
obj = Marshal.load(data)
obj.say_hello → "hello\n"
```

| | |
|------|---|
| load | load(<i>from</i> < , <i>proc</i> >) → <i>obj</i> |
|------|---|

转换 *from* 中存储的序列化数据为 Ruby 对象（可能带有相关的子对象），并返回转换结果。*from* 可以是 IO 的实例或能响应 *to_str* 的对象。如果带有 *proc*，那么反序列化时每个对象将作为参数传递给 *proc*。

| | |
|---------|--|
| restore | restore(<i>from</i> < , <i>proc</i> >) → <i>obj</i> |
|---------|--|

与 Marshal.load 同义。

Class MatchData < Object

所有的模式匹配都会设置特殊变量`$~`为一个包含匹配信息的 `MatchData`。方法 `Regexp#match` 和 `Regexp#last_match` 也会返回一个 `MatchData` 对象。该对象封装了一个模式匹配的所有结果，这些结果可以通过特殊变量`$&`, `$'`, `$``, `$1`, `$2` 等等访问（参见第 334 页的列表）。类 `MatchData` 也称作 `MatchingData`。

实例方法

[]

match[i] → obj
match[start, length] → array
match[range] → array

匹配引用 —— `MatchData` 和数组一样可以通过使用通常的下标技术来访问。
`match[0]` 等价于殊变量`$&`，其中含有整个匹配的部分。`Match[1]`、`match[2]`等保存后向匹配的值（括号括起来的模式部分）。参见 `MatchData#select` 和
1.8 `MatchData#values_at`。

```
m = /(.)(.)(\d+)(\d)/.match("THX1138.")
m[0]      →      "HX1138"
m[1, 2]    →      ["H", "X"]
m[1..3]    →      ["H", "X", "113"]
m[-3, 2]   →      ["X", "113"]
```

begin

match.begin(n) → int

返回字符串中匹配数组的第 *n* 个元素的开始字符的偏移。

```
m = /(.)(.)(\d+)(\d)/.match("THX1138.")
m.begin(0)    →      1
m.begin(2)    →      2
```

captures

match.captures → array

1.8 返回所有匹配组构成的数组。和 `MatchData#to_a` 相比，`MatchData#to_a` 返回完成匹配的字符串以及所有的匹配组。

```
m = /(.)(.)(\d+)(\d)/.match("THX1138.")
m.captures   →      ["H", "X", "113", "8"]
```

当在赋值语句中提取匹配中的部分内容时很有用。

```
f1, f2, f3 = /(.)(.)(\d+)(\d)/.match("THX1138.").captures
f1      →      "H"
f2      →      "X"
f3      →      "113"
```

end*match.end(n) → int*

返回匹配数组中第 *n* 个元素结尾后的字符，在字符串中的偏移。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.end(0) → 7
m.end(2) → 3
```

length*match.length → int*

返回匹配数组中元素的个数。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.length → 5
m.size → 5
```

offset*match.offset(n) → array*

返回一个数组，包含第 *n* 个匹配的开始偏移和结束偏移这两个元素。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.offset(0) → [1, 7]
m.offset(4) → [6, 7]
```

post_match*match.post_match → string*

返回原字符串中当前匹配后面的部分。等价于特殊变量 \$`。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138: The Movie")
m.post_match → ": The Movie"
```

pre_match*match.pre_match → string*

返回原字符串中当前匹配前面的部分。等价于特殊变量 \$`。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.pre_match → "T"
```

select*match.select { |val| block } → array*

1.8

返回 *match* 中那些使 *block* 为 true 的元素所组成的数组。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138: The Movie")
m.to_a → ["HX1138", "H", "X", "113", "8"]
m.select { |v| v =~ /\d\d/ } → ["HX1138", "113"]
```

size*match.size → int*

与 MatchData#length 同义。

| | |
|---------------|------------------------------|
| string | <i>match.string → string</i> |
|---------------|------------------------------|

返回传递给 `match` 的字符串的一个冻结拷贝。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.string → "THX1138."
```

| | |
|-------------|---------------------------|
| to_a | <i>match.to_a → array</i> |
|-------------|---------------------------|

返回匹配数组。和 `MatchData#captures` 不同，它将返回匹配的整个字符串。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.to_a → ["HX1138", "H", "X", "113", "8"]
```

| | |
|-------------|----------------------------|
| to_s | <i>match.to_s → string</i> |
|-------------|----------------------------|

返回匹配的整个字符串。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138.")
m.to_s → "HX1138"
```

| | |
|------------------|---|
| values_at | <i>match.values_at(<index>) → array</i> |
|------------------|---|

1.8 使用每个 `index` 访问匹配值，返回由相应的匹配组成的数组。

```
m = /(..)(..)(\d+)(\d)/.match("THX1138: The Movie")
m.to_a → ["HX1138", "H", "X", "113", "8"]
m.values_at(0, 2, -2) → ["HX1138", "X", "113"]
```

Module Math

Math 模块包含基本三角函数和超越函数等模块方法。第 487 页的 `Float` 类含有 Ruby 浮点数精度的常量列表。

模块常量

`E` e (自然对数的根) 的值。

`PI` π 的值。

模块方法

| | |
|--------------------|--|
| <code>acos</code> | <code>Math.acos(x)→float</code> |
| <code>1.8</code> | 计算 x 的反余弦值，返回值范围为 $0.. \pi$ |
| <code>acosh</code> | <code>Math.acosh(x)→float</code> |
| <code>1.8</code> | 计算 x 的反双曲余弦值。 |
| <code>asin</code> | <code>Math.asin(x)→float</code> |
| <code>1.8</code> | 计算 x 的反正弦值，返回值范围为 $0.. \pi$ 。 |
| <code>asinh</code> | <code>Math.asinh(x)→float</code> |
| <code>1.8</code> | 计算 x 的反双曲正弦值。 |
| <code>atan</code> | <code>Math.atan(x)→float</code> |
| <code>1.8</code> | 计算 x 的反正切值，返回值范围为 $-\pi/2 .. \pi/2$ 。 |
| <code>atanh</code> | <code>Math.atanh(x)→float</code> |
| <code>1.8</code> | 计算 x 的反双曲正切值。 |
| <code>atan2</code> | <code>Math.atan2(y, x)→float</code> |
| | 返回给定 y 和 x 的反正切值，返回值的范围为 $-\pi .. \pi$ 。 |
| <code>cos</code> | <code>Math.cos(x)→float</code> |
| | 计算以弧度表示的 x 的余弦，返回值的范围为 $-1..1$ 。 |
| <code>cosh</code> | <code>Math.cosh(x)→float</code> |
| <code>1.8</code> | 计算以弧度表示的 x 的双曲余弦。 |

| | |
|--------------|---|
| erf | Math.erf(<i>x</i>)→float |
| <u>1.8.</u> | 返回 <i>x</i> 的误差函数。 |
| | $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ |
| erfc | Math.erfc(<i>x</i>)→float |
| <u>1.8.</u> | 返回 <i>x</i> 的补误差函数。 |
| | $\text{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ |
| exp | Math.exp(<i>x</i>)→float |
| | 返回 e^x . |
| frexp | Math.frexp(<i>numeric</i>)→[<i>fraction</i>, <i>exponent</i>] |
| | 返回一个两元素数组，其中包含 <i>numeric</i> 的尾数部分（一个 Float）和指数部分（一个 Fixnum）。 |
| | <pre>fraction, exponent = Math.frexp(1234) → [0.6025390625, 11] fraction * 2**exponent → 1234.0</pre> |
| hypot | Math.hypot(<i>x</i>, <i>y</i>)→float |
| <u>1.8.</u> | 返回 $\sqrt{x^2 + y^2}$ ，直角边为 <i>x</i> 和 <i>y</i> 的直角三角形的斜边长度。 |
| | <code>Math.hypot(3, 4) → 5.0</code> |
| ldexp | Math.ldexp(<i>float</i>, <i>integer</i>)→float |
| | 返回 <i>float</i> $\times 2^{integer}$ 的值。 |
| | <pre>fraction, exponent = Math.frexp(1234) Math.ldexp(fraction, exponent) → 1234.0</pre> |
| log | Math.log(<i>numeric</i>)→float |
| | 返回 <i>numeric</i> 的自然对数。 |
| log10 | Math.log10(<i>numeric</i>)→float |
| | 返回 <i>numeric</i> 的以 10 为根的对数。 |
| sin | Math.sin(<i>numeric</i>)→float |
| | 计算以弧度表示的 <i>numeric</i> 的正弦。返回值范围为 -1..1。 |

| | |
|-------------|---|
| sinh | Math.sinh(<i>numeric</i>)→float |
| 1.8. | 计算以弧度表示的 <i>numeric</i> 的双曲正弦。 |
| sqrt | Math.sqrt(<i>numeric</i>)→float |
| | 返回 <i>numeric</i> 的非负平方根。如果 <i>numeric</i> 小于 0，则因为 ArgError。 |
| tan | Math.tan(<i>numeric</i>)→float |
| | 返回以弧度表示的 <i>numeric</i> 的正切。 |
| tanh | Math.tanh(<i>numeric</i>)→float |
| 1.8. | 计算以弧度表示的 <i>numeric</i> 的双曲正切。 |

Class Method < Object

Method 对象是由 `Object#method` 方法创建的。它们和特定的对象（不仅仅和类）相关联。可以使用它们来调用对象保存的方法或者作为与迭代器相关联的 block。可以解除它们对一个对象的关联（创建一个 `UnboundMethod`），并关联到其他对象上。

```
def square(n)
  n*n
end
meth = self.method(:square)

meth.call(9)          → 81
[1, 2, 3].collect(&meth) → [1, 4, 9]
```

实例方法

[]

meth[<args>]→object*

与 `Method.call` 同义。

==

meth==other→true 或 false

1.8

如果 *meth* 和 *other* 是同一个函数，则返回 `true`。

```
def fred()
  puts "Hello"
end

alias bert fred → nil

m1 = method(:fred)
m2 = method(:bert)
m1 == m2 → true
```

arity

meth.arity→fixnum

返回方法接受的参数的个数。参见下页的图 27.2。

call

meth.call(<args>)→object*

以给定的参数调用 *meth*，返回 *meth* 方法的返回值。

```
m = 12.method("+")
m.call(3) → 15
m.call(20) → 32
```

eq?

meth.eql?(other)→true 或 false

1.8

如果 *meth* 是和 *other* 一样的方法，则返回 `true`。

对接受固定个数参数的方法而言，`Method#arity` 将返回一个非负整数。对那些接受可变个数参数的 Ruby 方法而言，返回 $-n - 1$ ，这里的 n 是要求的参数的个数。对于用 C 编写的方法，若其参数个数可变，则返回 -1 。

```
class C
  def one; end
  def two(a); end
  def three(*a); end
  def four(a, b); end
  def five(a, b, *c); end
  def six(a, b, *c, &d); end
end
c = C.new
c.method(:one).arity → 0
c.method(:two).arity → 1
c.method(:three).arity → -1
c.method(:four).arity → 2
c.method(:five).arity → -3
c.method(:six).arity → -3

"cat".method(:size).arity → 0
"cat".method(:replace).arity → 1
"cat".method(:squeeze).arity → -1
"cat".method(:count).arity → -1
```

图 27.2 实际运行 `Method#arity`

```
def fred()
  puts "Hello"
end

alias bert fred → nil

m1 = method(:fred)
m2 = method(:bert)
m1.eql?(m2) → false
```

to_proc

meth.to_proc→prc

- 1.8 返回与此方法相对应的一个 `Proc` 对象。因为当传递 `block` 参数时解释器将调用 `to_proc`，所以可以使用跟在地址符`&`后面的方法对象将 `block` 传递给另外一个方法调用。参见本节开始的 `Thing` 例子。

unbind

meth.unbind→unbound_method

- 1.8 解除 `meth` 和当前接受者的关联。返回的 `UnboundMethod` 随后可以关联到相同类的一个新对象上（参见第 651 页的 `UnboundMethod`）。

Class Module < Object

子类: Class

模块 (Module) 是方法 (method) 和常量 (constant) 的一个集合 (collection)。模块内的方法可能是实例方法或者是模块方法。当包含模块时, 它的实例方法会作为方法出现在一个类中; 而模块方法则不会出现。相反, 可以在未创建混合对象实例的情况下调用模块方法, 而实例方法则不可以。同样参见第 558 页的 `Module#module_function`。

在接下来的描述中, 参数 `symbol` 指的是一个符号, 它可能是一个引号字符串或者是一个 `Symbol` (例如: `:name`)。

```
module Mod
  include Math
  CONST = 1
  def meth
    # ...
  end
end
Mod.class          → Module
Mod.constants      → ["E", "CONST", "PI"]
Mod.instance_methods → ["meth"]
```

类方法

constants

`Module.constants` → array

返回定义在系统内的所有常量名称的数组。这个列表包含了所有模块和类的名称。

```
p Module.constants.sort[1..5]
```

输出结果:

```
["ARGV", "ArgumentError", "Array", "Bignum", "Binding"]
```

nesting

`Module.nesting` → array

返回在当前调用点上嵌套的模块列表。

```
module M1
  module M2
    $a = Module.nesting
  end
end
$a           → [M1::M2, M1]
$a[0].name   → "M1::M2"
```

new

Module.new → mod

Module.new{|mod|block} → mod

- 1.8 创建新的匿名 (anonymous) 模块。如果给定一个 block，模块对象会传递给它，并在这个模块的上下文中使用 module_eval 方法对 block 进行求解 (evaluate)。

```
Fred = Module.new do
  def meth1
    "hello"
  end
  def meth2
    "bye"
  end
end
a = "my string"
a.extend(Fred)      → "my string"
a.meth1            → "hello"
a.meth2            → "bye"
```

实例方法

<,<=,>,>=

mod relop module → true 或 false

层次结构查询——如果一个模块包含在别的模块中或者它是别的模块的一个父类，这个模块被认为大于另外一个模块。同样依次定义了别的操作符。如果两个模块之间没有任何关系，则所有操作符都返回 false。

```
module Mixin
end

module Parent
  include Mixin
end

module Unrelated
end

Parent > Mixin      → false
Parent < Mixin      → true
Parent <= Parent    → true
Parent < Unrelated  → nil
Parent > Unrelated  → nil
```

<=>

mod <=> other_mod → -1, 0, +1

比较 (comparison) ——如果 mod 包含了 other_mod，则返回 -1。如果 mod 和 other_mod 是相同的模块，则返回 0。如果 mod 被包含在 other_mod 中或者两者没有任何关系，则返回 +1。

====

mod === obj → true 或 false

Case 等式 (case equality) ——如果 obj 是 mod 或者其派生类的一个实例，则返回 true。很少用于模块，但是可以用在 case 语句中来通过类测试对象。

ancestors*mod.ancestors* → array

返回包括在 *mod* 中的模块列表（包括 *mod* 本身）。

```
module Mod
  include Math
  include Comparable
end

Mod.ancestors → [Mod, Comparable, Math]
Math.ancestors → [Math]
```

autoload

mod.autoload(name, file_name) → nil

- 1.8 注册 *file_name*, 这样当 *name* (可能是一个 String 或者一个 Symbol) 模块第一次在 *mod* 的名字空间内被访问时, 会载入 (使用 Kernel.require) *file_name*。请注意: 自动载入的文件是在顶层 (top-level) 上下文中被求解。在这个例子中,

module_b.rb 包括了:

```
module A::B # in module_b.rb
  def doit
    puts "In Module A::B"
  end
  module_function :doit
end
```

然后别的代码可以自动地包括这个模块。

```
module A
  autoload(:B, "module_b")
end

A::B.doit      # autoloads "module_b"
```

输出结果:

```
In Module A::B
```

autoload?

mod.autoload?(name) → *file_name* 或 nil

- 1.8 返回在 *mod* 的上下文中引用 *name* 字符串或 *name* 符号时会自动载入的文件名, 或者如果没有关联的自动载入, 则返回 nil。

```
module A
  autoload(:B, "module_b")
end

Aautoload?(:B) → "module_b"
Aautoload?(:C) → nil
```

class_eval

mod.class_eval(string <, file_name <, line_number >>) → *obj**mod.class_eval { block }* → *obj*

等同于 *Module.module_eval*。

class_variables*mod.class_variables* → *array*

返回在 *mod* 及其祖先中的类变量名称的数组。

```
class One
  @@var1 = 1
end
class Two < One
  @@var2 = 2
end
One.class_variables → ["@@var1"]
Two.class_variables → ["@@var2", "@@var1"]
```

clone*mod.clone* → *other_mod*

创建模块的新拷贝。

```
m = Math.clone → #<Module:0x1c976>
m.constants → ["PI", "E"]
m == Math → false
```

const_defined?*mod.const_defined?(symbol)* → true 或 false

如果 *mod* 定义了给定名称的常量，则返回 true。

```
Math.const_defined? "PI" → true
```

const_get*mod.const_get(symbol)* → *obj*

返回 *mod* 中的给定常量的值。

```
Math.const_get :PI → 3.14159265358979
```

const_missing*const_missing(symbol)* → *obj*

1.8. 引用 *mod* 中的未定义常量时，*const_missing* 会被调用。它接收这个未定义常量的符号，并返回为这个常量所用的值。下面代码的风格实在是很糟糕。如果对一个未定义常量作出了引用，它试图载入一个文件，其文件名是这个常量的小写版本（因而，假设 Fred 类是在 fred.rb 文件中）。如果发现了这个文件，它返回这个被载入类的值。因此它实现了一种不正当的自动载入装置。

```
def Object.const_missing(name)
  @looked_for ||= {}
  str_name = name.to_s
  raise "Class not found: #{name}" if @looked_for[str_name]
  @looked_for[str_name] = 1
  file = str_name.downcase
  require file
  klass = const_get(name)
  return klass if klass
  raise "Class not found: #{name}"
end
```

| | |
|------------------------|---|
| <code>const_set</code> | <code>mod.const_set(symbol, obj) → obj</code> |
|------------------------|---|

把给定的常量设置给指定的对象，同时返回这个对象。如果指定名称的常量先前不存在，则创建新的常量。

```
Math.const_set("HIGH SCHOOL PI", 22.0/7.0) → 3.14285714285714
Math::HIGH SCHOOL PI - Math::PI → 0.00126448926734968
```

| | |
|------------------------|------------------------------------|
| <code>constants</code> | <code>mod.constants → array</code> |
|------------------------|------------------------------------|

返回 `mod` 中可访问的常量的名称数组。这包括了包含在 `mod` 中的任何模块的常量名称（参见本节开始的例子）。

| | |
|-----------------------|---|
| <code>include?</code> | <code>mod.include?(other_mod) → true 或 false</code> |
|-----------------------|---|

1.8 如果 `other_mod` 被包括在 `mod` 或者其祖先中，返回 `true`。

```
module A
end

class B
  include A
end

class C < B
end

B.include?(A) → true
C.include?(A) → true
A.include?(A) → false
```

| | |
|-------------------------------|---|
| <code>included_modules</code> | <code>mod.included_modules → array</code> |
|-------------------------------|---|

返回包括在 `mod` 中的模块列表。

```
module Mixin
end

module Outer
  include Mixin
end

Mixin.included_modules → []
Outer.included_modules → [Mixin]
```

| | |
|------------------------------|---|
| <code>instance_method</code> | <code>mod.instance_method(symbol) → unbound_method</code> |
|------------------------------|---|

1.8 返回在 `mod` 中表示这个给定实例方法的 `UnboundMethod`。

```
class Interpreter
  def do_a() print "there, "; end
  def do_d() print "Hello "; end
  def do_e() print "!\n"; end
  def do_v() print "Dave"; end
```

```

Dispatcher = {
  ?a => instance_method(:do_a),
  ?d => instance_method(:do_d),
  ?e => instance_method(:do_e),
  ?v => instance_method(:do_v)
}
def interpret(string)
  string.each_byte{|b| Dispatcher[b].bind(self).call }
end
end

interpreter = Interpreter.new
interpreter.interpret('dave')

```

输出结果：

```
Hello there, Dave!
```

| | |
|-------------------------|---|
| instance_methods | <i>mod.instance_methods(inc_super=true) → array</i> |
|-------------------------|---|

- 1.8 返回一个数组，包含在接受者中的 `public` 实例方法的名称。对于模块，它是 `public` 方法；对于类，它是实例方法（不是 `singleton` 方法）。当没有任何参数或者参数是 `true` 时，它会返回在 `mod` 以及 `mod` 的超类中的方法的名称。当一个模块作为接受者被调用或者参数是 `false` 时，它返回 `mod` 中的实例方法（在 2004 年 1 月之前的 Ruby 版本中，这个参数默认是 `false`）。

```

module A
  def method1()
  end
end

class B
  def method2()
  end
end

class C < B
  def method3()
  end
end

A.instance_methods
B.instance_methods(false)
C.instance_methods(false)
C.instance_methods(true).length

```

| | |
|------------------------|---|
| method_defined? | <i>mod.method_defined?(symbol) → true 或 false</i> |
|------------------------|---|

如果 `mod`（或者它包括的模块；如果 `mod` 是类，或者 `mod` 的祖先）定义了给定的方法，则返回 `true`。匹配 `public` 和 `protected` 方法。

```

module A
  def method1() end
end
class B
  def method2() end
end
class C < B
  include A
  def method3() end
end

A.method_defined? :method1 → true
C.method_defined? "method1" → true
C.method_defined? "method2" → true
C.method_defined? "method3" → true
C.method_defined? "method4" → false

```

| | |
|--------------------|--|
| module_eval | <i>mod.class_eval(string<, file_name <, line_number>>)→obj</i> |
| | <i>mod.module_eval { block } →obj</i> |

在 *mod* 的上下文中对 *string* 或 *block* 进行求解。这可以用来把方法添加到一个类中。*module_eval* 对它的参数求解后把结果返回。可选的 *file_name* 和 *line_number* 参数设置出错信息文本。

```

class Thing
end
a = %q{def hello() "Hello there!" end}
Thing.module_eval(a)
puts Thing.new.hello()
Thing.module_eval("invalid code", "dummy", 123)

```

输出结果：

```

Hello there!
dummy:123:in `module_eval': undefined local variable
or method `code' for Thing:Class

```

| | |
|----------------------|------------------------|
| name | <i>mod.name→string</i> |
| 返回 <i>mod</i> 模块的名称。 | |

| | |
|-----------------------------|--|
| private_class_method | <i>mod.private_class_method(<symbol>*)→nil</i> |
|-----------------------------|--|

使得已有类方法变成 *private*。常常用于隐藏默认构造函数 *new*。

```

class SimpleSingleton # Not thread safe
  private_class_method :new
  def SimpleSingleton.create(*args, &block)
    @me = new(*args, &block) if ! @me
    @me
  end
end

```

private_instance_methods *mod.private_instance_methods(inc_super=true) → array*

- 1.8 返回定义在 *mod* 中的 *private* 实例方法的列表。如果可选参数是 *true*, *mod* 所有祖先的方法也会包括在列表中（在 2004 年 1 月之前的 Ruby 版本中，这个参数默认是 *false*）。

```
module Mod
  def method1() end
  private :method1
  def method2() end
end
Mod.instance_methods      → ["method2"]
Mod.private_instance_methods → ["method1"]
```

private_method_defined? *mod.private_method_defined?(symbol) → true 或 false*

- 1.8 如果 *mod*（或者它包括的模块；如果 *mod* 是类，或者 *mod* 的祖先）定义了给定的 *private* 方法，则返回 *true*。

```
module A
  def method1() end
end
class B
  private
  def method2() end
end
class C < B
  include A
  def method3() end
end

A.method_defined? :method1      → true
C.private_method_defined? "method1" → false
C.private_method_defined? "method2" → true
C.method_defined? "method2"     → false
```

protected_instance_methods *mod.protected_instance_methods(inc_super=true) → array*

- 1.8 返回定义在 *mod* 中的 *protected* 实例方法列表。如果可选参数是 *true*, *mod* 任何祖先的方法会包括在列表中（在 2004 年 1 月之前的 Ruby 版本中，这个参数默认是 *false*）。

protected_method_defined? *mod.protected_method_defined?(symbol) → true 或 false*

- 1.8 如果 *mod*（或者它包括的模块；如果 *mod* 是类，或者 *mod* 的祖先）定义了给定的 *protected* 方法，则返回 *true*。

```

module A
  def method1() end
end
class B
  protected
  def method2() end
end
class C < B
  include A
  def method3() end
end

A.method_defined? :method1      → true
C.protected_method_defined? "method1" → false
C.protected_method_defined? "method2" → true
C.method_defined? "method2"     → true

```

public_class_method *mod.public_class_method(<symbol>+)* → nil

使得一组已有的类方法变成 public。

public_instance_methods *mod.public_instance_methods(inc_super=true)* → array

- 1.8 返回定义在 *mod* 中的 public 实例方法列表。如果可选参数是 true, *mod* 任何祖先的方法被包括在列表中（在 2004 年 1 月之前的 Ruby 版本中，这个参数默认是 false）。

public_method_defined? *mod.public_method_defined?(symbol)* → true 或 false

- 1.8 如果 *mod*（或者它包括的模块；如果 *mod* 是类，或者 *mod* 的祖先）定义了给定的 public 方法，则返回 true。

```

module A
  def method1() end
end
class B
  protected
  def method2() end
end
class C < B
  include A
  def method3() end
end

A.method_defined? :method1      → true
C.public_method_defined? "method1" → true
C.public_method_defined? "method2" → false
C.method_defined? "method2"     → true

```

Private 实例方法

alias_method

`alias_method(new_id, old_id) → mod`

使得 `new_id` 成为 `old_id` 方法的新拷贝。这可以用来保留对被覆写 (overridden) 方法的访问。

```
module Mod
  alias_method :orig_exit, :exit
  def exit(code=0)
    puts "Exiting with code #{code}"
    orig_exit(code)
  end
end
include Mod
exit(99)
```

输出结果：

```
Exiting with code 99
```

append_features

`append_features(other_mod) → mod`

当这个模块包含在别的模块中时，Ruby 调用这个模块中的 `append_features`，并把接受者模块放在 `other_mod` 中传递给它。如果这个模块还没有添加到 `other_mod` 中或它的任何祖先，Ruby 的默认实现是把这个模块的常量、方法以及模块变量添加到 `other_mod` 中。在 Ruby 1.8 版本之前，用户代码常常重定义了 `append_features` 添加自己的功能，然后调用 `super` 去处理实际的添加 (`include`)。在 Ruby 1.8 版本中，相反你应当实现 `Module#included` 方法。同时参见第 556 页的 `Module#include`。

attr

`attr(symbol, writable=false) → nil`

为这个模块定义给定的属性，属性名称是 `symbol.id2name`，并创建了一个实例变量 (@name) 和一个相应的访问方法去读取这个属性。如果可选的 `writable` 参数是 `true`，同时也创建 `name=` 方法以设置这个属性。

```
module Mod
  attr :size, true
end
```

等同于：

```
module Mod
  def size
    @size
  end
  def size=(val)
    @size = val
  end
end
```

attr_accessorattr_accessor(<*symbol*>+) → nil

相当于对每个 *symbol* 依次调用 “attr *symbol*, true”。

```
module Mod
  attr_accessor(:one, :two)
end6
Mod.instance_methods.sor → ["one", "one=", "two", "two="]
```

attr_reader

attr_reader(<*symbol*>+) → nil

创建实例变量和相应的方法，这个方法返回每个实例变量的值。相当于对每个名称依次调用 attr:*name*。

attr_writerattr_writer(<*symbol*>+) → nil

创建 accessor 方法，允许对这个 *symbol*.id2name 属性进行赋值。

define_methoddefine_method(*symbol*, *method*) → *method*
define_method(*symbol*) { *block* } → *proc*

1.8. 在接受者中定义实例方法。*method* 参数可以是 Proc 或者 Method 对象。如果给定了 *block*，则它会被用作这个方法的程序体。*block* 是通过使用 instance_eval 来求解的。很困难 (tricky) 去作出示范，因为 define_method 是 private 方法（这是为什么我们在这个例子中求助于 send 方法的原因）。

```
class A
  def fred
    puts "In Fred"
  end
  def create_method(name, &block)
    self.class.send(:define_method, name, &block)
  end
  define_method(:wilma) { puts "Charge it!" }
end
class B < A
  define_method(:barney, instance_method(:fred))
end
b = B.new
b.barney
b.wilma
b.create_method(:betty) { p self }
b.betty
```

输出结果：

```
In Fred
Charge it!
#<B:0x1c9134>
```

| | |
|----------------------|---|
| extend_object | extend_object(<i>obj</i>)→<i>obj</i> |
|----------------------|---|

通过添加这个模块的常量和方法（作为 singleton 方法被添加）来扩展指定的对象。Object#extend 会使用这个回调（callback）方法。

```
module Picky
  def Picky.extend_object(o)
    if String === o
      puts "Can't add Picky to a String"
    else
      puts "Picky added to #{o.class}"
      super
    end
  end
(s = Array.new).extend Picky # Call Object.extend
(s = "quick brown fox").extend Picky
```

输出结果：

```
Picky added to Array
Can't add Picky to a String
```

| | |
|-----------------|-------------------------------------|
| extended | extended(<i>other_mod</i>) |
|-----------------|-------------------------------------|

- 1.8 无论何时当这个接受者用来扩展一个对象，extended 回调方法会被调用。这个对象作为参数被传入。当一个模块用来扩展一个对象时，如果你的代码想执行某些动作，你应当优先于 Module#extend_object 而考虑使用这个方法。

```
module A
  def A.extended(obj)
    puts "#{self} extending '#{obj}'"
  end
end
"cat".extend(A)
```

输出结果：

```
A extending 'cat'
```

| | |
|----------------|--|
| include | include(< <i>other_mod</i> >+)→<i>mod</i> |
|----------------|--|

对每个参数（以颠倒顺序）调用 Module.append_features（在第 554 页说明了）。相当于下面的代码。

```
def include(*modules)
  modules.reverse_each do |mod|
    mod.append_features(self)
    mod.included(self)
  end
end
```

| | |
|-----------------|-------------------------------------|
| included | included(<i>other_mod</i>) |
|-----------------|-------------------------------------|

- 1.8 无论何时当这个接受者包括在别的模块或者类中，included 回调方法会被调用。

当一个模块包括在别的模块中时，如果你的代码想执行某些动作，你应当优先于 `Module#append_features` 而考虑使用这个方法。

```
module A
  def A.included(mod)
    puts "#{self} included in #{mod}"
  end
end
module Enumerable
  include A
end
```

输出结果：

```
A included in Enumerable
```

method_added

method_added(*symbol*)

无论何时当一个方法添加到这个接受者中，`method_added` 回调方法会被调用。

```
module Chatty
  def Chatty.method_added(id)
    puts "Adding #{id.id2name}"
  end
  def one() end
end
module Chatty
  def two() end
end
```

输出结果：

```
Adding one
Adding two
```

method_removed

method_removed(*symbol*)

1.8 无论何时当一个方法从这个接受者中删除，`method_removed` 回调方法会被调用。

```
module Chatty
  def Chatty.method_removed(id)
    puts "Removing #{id.id2name}"
  end
  def one() end
end
module Chatty
  remove_method(:one)
end
```

输出结果：

```
Removing one
```

method_undefined

method_undefined(*symbol*)

1.8 无论何时当一个方法在这个接受者中没有定义（`undefined`），`method_undefined` 回调方法会被调用。

```
module Chatty
  def Chatty.method_undefined(id)
    puts "Undefining #{id.id2name}"
  end
  def one() end
end
module Chatty
  undef_method(:one)
end
```

输出结果：

```
Undefining one
```

module_function

module_function(<symbol>*)→ mod

为给定的方法创建模块函数（module function）。把这个模块作为一个接受者，这些函数可以被调用，它们作为实例方法，可以被混合在这个模块中的那些类访问。模块函数是原始函数的拷贝，所以它们能够独立地被修改。这些实例方法版本的函数被变成 private。如果不带任何参数调用 module_function 时，随后定义的方法就成为模块函数。

```
module Mod
  def one
    "This is one"
  end
  module_function :one
end
class Cls
  include Mod
  def call_one
    one
  end
end
Mod.one      →  "This is one"
c = Cls.new
c.call_one  →  "This is one"
module Mod
  def one
    "This is the new one"
  end
end
Mod.one      →  "This is one"
c.call_one  →  "This is the new one"
```

private

private(<symbol>*)→ mod

不带任何参数调用时，把随后定义的方法的默认可见范围（visibility）设置为 private。带有参数调用时，把给定方法的可见范围设置为 private。参见从第 536 页开始的“访问控制”。

```
module Mod
  def a() end
  def b() end
  private
  def c() end
  private :a
end
Mod.private_instance_methods → ["c", "a"]
```

protected **protected(<symbol>*)→mod**

不带任何参数调用时，把随后定义的方法的默认可见范围设置为 `protected`。带有参数调用时，把给定方法的可见范围设置为 `protected`。参见从第 356 页开始描述的“访问控制”。

public **public(<symbol>*)→mod**

不带任何参数调用时，把随后定义的方法的默认可见范围设置为 `public`。带有参数调用时，把给定方法的可见范围设置为 `public`。参见从第 356 页开始描述的“访问控制”。

remove_class_variable **remove_class_variable(symbol)→obj**

1.8. 删除 `symbol` 定义，并返回这个常量的值。

```
class Dummy
  @@var = 99
  puts @@var
  remove_class_variable(:@@var)
  puts(define? @@var)
end
```

输出结果：

```
99
nil
```

remove_const **remove_const(symbol)→obj**

删除给定常量的定义，并返回这个常量的值。预定义的类和 `singleton` 对象（比如 `true`）不能被删除。

remove_method **remove_method(symbol)→mod**

从当前的类删除 `symbol` 标识的方法。举个例子，参见 `Module#undef_method`。

undef_method **undef_method(<symbol>*)→mod**

阻止当前类响应对这些给定方法的调用。这与 `remove_method` 方法形成鲜明对比：`remove_method` 从这个特别的类删除方法，而 Ruby 将仍然会从它的超类以及 `mixed-in` 模块中查找，以找到一个可能的接受者。

```
class Parent
  def hello
    puts "In parent"
  end
end
class Child < Parent
  def hello
    puts "In child"
  end
end
c = Child.new
c.hello
class Child
  remove_method :hello # remove from child, still in parent
end
c.hello
class Child
  undef_method :hello # prevent any calls to 'hello'
end
c.hello
```

输出结果：

```
In child
In parent
prog.rb:23: undefined method `hello' for #<Child:0x1c92ec> (NoMethodError)
```

Class NilClass < Object

singleton 对象 nil 的类。

实例方法

&

$\text{nil} \& \text{obj} \rightarrow \text{false}$

与 (And) ——返回 false。当 obj 是方法调用的一个参数时，它总是会被求解；在这种情况下，不会进行短路 (short-circuit) 求解。

```
nil && puts("logical and")
nil & puts("and")
```

输出结果：

```
and
```

^

$\text{nil} \wedge \text{obj} \rightarrow \text{true 或 false}$

异或 (Exclusive Or) ——如果 obj 是 nil 或者 false，则返回 false；否则返回 true。

|

$\text{nil} \mid \text{obj} \rightarrow \text{true 或 false}$

或 (Or) ——如果 obj 是 nil 或者 false 返回 false；否则返回 true。

```
nil | false → false
nil | 99 → true
```

nil?

$\text{nil.nil?} \rightarrow \text{true}$

总是返回 true。

to_a

$\text{nil.to_a} \rightarrow []$

总是返回空数组。

```
nil.to_a → []
```

to_f

$\text{nil.to_f} \rightarrow 0.0$

1.8

总是返回零。

```
nil.to_f → 0.0
```

to_i

$\text{nil.to_i} \rightarrow 0$

总是返回零。

```
nil.to_i → 0
```

to_s

$\text{nil.to_s} \rightarrow ""$

总是返回空字符串。

```
nil.to_s → ""
```

Class Numeric < Object

子类: Float, Integer

Numeric 是抽象类 Integer 和具体数字类比如 Float、Fixnum 和 Bignum 的基础类型。Numeric 中的许多方法会在其子类中被覆写 (override)，同时 Numeric 能够自由调用这些子类中的方法。第 564 页的表 27.11 列出了所有这五个类中定义的方法的完全列表。

Mixes in

Comparable:

`<, <=, ==, >=, >, between?`

实例方法

+@ `+num → num`

一元 (Unary) 加——返回这个接受者的值。

-@ `-num → numeric`

一元 (Unary) 减——返回这个接受者的值，求反。

<=> `num <=> other → 0 或 nil`

如果 `num` 等于 `other`，则返回 0；否则返回 nil。

abs `num.abs → numeric`

返回 `num` 的绝对值。

```
12.abs      → 12
(-34.56).abs → 34.56
-34.56.abs   → 34.56
```

ceil `num.ceil → int`

返回大于或等于 `num` 的最小 Integer。Numeric 类通过把它自己转换成一个 Float，然后调用 Float#ceil 来实现这个功能。

```
1.ceil      → 1
1.2.ceil    → 2
(-1.2).ceil → -1
(-1.0).ceil → -1
```

coerce `num.coerce(numeric) → array`

coerce 是 Numeric 类的实例方法，同时也是类型转换协议的一部分。当一个数字被要求执行一个操作，而传入的参数其类与其数字本身的类不同，它必须首先把

自己和这个参数都强制转换成同一个类，然后这个操作才有意义。比如，在表达式 `1 + 2.5` 中，`Fixnum` `1` 必须被强制转换成一个浮点数，使得它和 `2.5` 保持一致。这个转换是由 `coerce` 执行的。对于所有的数字对象来说，`coerce` 是一目了然的：如果 `numeric` 的类型与 `num` 的类型相同，则返回一个含有 `numeric` 和 `num` 的数组；否则返回一个数组，其中 `numeric` 和 `num` 都是以 `Float` 对象来表示的。

```
1.coerce(2.5)    → [2.5, 1.0]
1.2.coerce(3)   → [3.0, 1.2]
1.coerce(2)     → [2, 1]
```

如果一个 `numeric` 对象被要求对一个 `non-numeric` 对象进行操作，它会试图对那个对象调用 `coerce`。比如，如果写成：

```
1 + "2"
```

Ruby 实际上以如下方式执行这个代码：

```
n1, n2 = "2".coerce(1)
n2 + n1
```

在更一般的情况下，这行不通，因为大多数 `non-numeric` 对象并没有定义这个 `coerce` 方法。当然，你可以使用它（如果你倾向于此）来部分地实现 Perl 表达式中字符串到数字的自动转换功能。

```
class String
  def coerce(other)
    case other
    when Integer
      begin
        return other, Integer(self)
      rescue
        return Float(other), Float(self)
      end
    when Float
      return other, Float(self)
    else super
    end
  end
end

1 + "2"      → 3
1 - "2.3"   → -1.3
1.2 + "2.3" → 3.5
1.5 - "2"   → -0.5
```

第 374 页更进一步讨论了 `coerce`。

div

`num.div(numeric) → int`

1.8

使用 / 来执行除法，然后把结果转换成一个整数。`Numeric` 没有定义这个 / 操作符，留给了它的子类来定义。

| | Numeric | Integer | Fixnum | Bignum | Float |
|-----------|---------|---------|--------|--------|-------|
| % | - | - | ✓ | ✓ | ✓ |
| & | - | - | ✓ | ✓ | - |
| * | - | - | ✓ | ✓ | ✓ |
| ** | - | - | ✓ | ✓ | ✓ |
| + | - | - | ✓ | ✓ | ✓ |
| +@ | ✓ | - | - | - | - |
| - | - | - | ✓ | ✓ | ✓ |
| -@ | ✓ | - | ✓ | ✓ | ✓ |
| / | - | - | ✓ | ✓ | ✓ |
| < | - | - | ✓ | - | ✓ |
| << | - | - | ✓ | ✓ | - |
| <= | - | - | ✓ | - | ✓ |
| <=> | ✓ | - | ✓ | ✓ | ✓ |
| = | - | - | ✓ | ✓ | ✓ |
| > | - | - | ✓ | - | ✓ |
| >= | - | - | ✓ | - | ✓ |
| >> | - | - | ✓ | ✓ | - |
| [] | - | - | ✓ | ✓ | - |
| ^ | - | - | ✓ | ✓ | - |
| abs | ✓ | - | ✓ | ✓ | ✓ |
| ceil | ✓ | ✓ | - | - | ✓ |
| chr | - | ✓ | - | - | - |
| coerce | ✓ | - | - | ✓ | ✓ |
| div | ✓ | - | ✓ | ✓ | - |
| divmod | ✓ | - | ✓ | ✓ | ✓ |
| downto | - | ✓ | - | - | - |
| eq? | ✓ | - | - | ✓ | ✓ |
| finite? | - | - | - | - | ✓ |
| floor | ✓ | ✓ | - | - | ✓ |
| hash | - | - | - | ✓ | ✓ |
| id2name | - | - | ✓ | - | - |
| infinite? | - | - | - | - | ✓ |
| integer? | ✓ | ✓ | - | - | - |
| modulo | ✓ | - | ✓ | ✓ | ✓ |
| nan? | - | - | - | - | ✓ |
| next | - | ✓ | - | - | - |
| nonzero? | ✓ | - | - | - | - |
| quo | ✓ | - | ✓ | ✓ | - |
| remainder | ✓ | - | - | ✓ | - |
| round | ✓ | ✓ | - | - | ✓ |
| size | - | - | ✓ | ✓ | - |
| step | ✓ | - | - | - | - |
| succ | - | ✓ | - | - | - |
| times | - | ✓ | - | - | - |
| to_f | - | - | ✓ | ✓ | ✓ |
| to_i | - | ✓ | - | - | ✓ |
| to_int | ✓ | ✓ | - | - | ✓ |
| to_s | - | - | ✓ | ✓ | ✓ |
| to_sym | - | - | ✓ | - | - |
| truncate | ✓ | ✓ | - | - | ✓ |
| upto | - | ✓ | - | - | - |
| zero? | ✓ | - | ✓ | - | ✓ |
| ~ | - | - | ✓ | ✓ | - |

表 27.11 定义在 Numeric 类以及其子类中的方法。打勾意味着这个方法定义在相应的类中

表 27.12 模 (modulo) 和余数 (remainder) 之间的区别。模运算符 (“%”) 总是有除数 (divisor) 的符号, 然而 remainder 有被除数 (dividend) 的符号

| a | b | a.divmod(b) | a / b | a.modulo(b) | a.remainder(b) |
|-------|----|-------------|--------|-------------|----------------|
| 13 | 4 | 3, 1 | 3 | 1 | 1 |
| 13 | -4 | -4, -3 | -4 | -3 | 1 |
| -13 | 4 | -4, 3 | -4 | 3 | -1 |
| -13 | -4 | 3, -1 | 3 | -1 | -1 |
| 11.5 | 4 | 2.0, 3.5 | 2.875 | 3.5 | 3.5 |
| 11.5 | -4 | -3.0, -0.5 | -2.875 | -0.5 | 3.5 |
| -11.5 | 4 | -3.0, 0.5 | -2.875 | 0.5 | -3.5 |
| -11.5 | -4 | 2.0, -3.5 | 2.875 | -3.5 | -3.5 |

divmod *num.divmod(numeric) → array*

返回商 (quotient) 和模 (modulus) 的数组, 对 *num* 除以 *numeric* 可以得到它们。如果 $q, r = x.divmod(y)$, $q = \text{floor}(float(x)/float(y))$ 和 $x = q \times y + r$, 这个商取整到负无穷大 ($-\infty$)。例子请参见表 27.12。

eql? *num.eql?(numeric) → true 或 false*

如果 *num* 和 *numeric* 的类型相同并且值相等, 则返回 *true*。

```
1 == 1.0      → true
1.eql?(1.0)   → false
(1.0).eql?(1.0) → true
```

floor *num.floor → int*

返回小于或者等于 *num* 的最大整数。Numeric 类通过把 *int* 转换成一个 Float, 然后调用 Float#floor 来实现这个功能。

```
1.floor      → 1
(-1).floor   → -1
```

integer? *num.integer? → true 或 false*

如果 *num* 是一个 Integer (包括 Fixnum 和 Bignum), 则返回 *true*。

modulo *num.modulo(numeric) → numeric*

等同于 *num.divmod(numeric)[1]*。

nonzero? *num.nonzero? → num 或 nil*

如果 *num* 不是 0, 则返回 *num*; 否则返回 *nil*。当把比较串联在一起时, 这个作法很有用。

```
a = %w( z Bb bB bb BB a aA Aa AA A )
b = a.sort { |a,b| (a.downcase <=> b.downcase).nonzero? || a <=> b }
b → ["A", "a", "AA", "Aa", "aA", "BB", "Bb", "bB", "bb", "z"]
```

quo*num.quo(numeric) → numeric*

1.8. 等同于 `Numeric# /`，但是在子类中会被覆写。`quo` 的意图是返回除法的（在上下文中）尽可能精确的结果。因此 `1.quo(2)` 等于 `0.5`，而 `1/2` 等于 `0`。

remainder*num.remainder(numeric) → numeric*

如果 `num` 和 `numeric` 的符号不同，则返回 `mod-numeric`；否则返回 `mod`。在这两种情况下，`mod` 的值为 `num.modulo(numeric)`。`remainder` 和 `modulo (%)` 的区别显示在前一页的表 27.12 中。

round*num.round → int*

将 `num` 取整（`round`）到最近的整数。`Numeric` 类通过把 `int` 转换成一个 `Float`，然后调用 `Float#round` 来实现这个功能。

step*num.step(end_num, step) {|i| block} → num*

1.8. 用从 `num` 开始的数字序列来调用 `block`，每次调用时，这个数字会增加 `step`。当传递给 `block` 的值大于 `end_num`（如果 `step` 是正数）或者小于 `end_num`（如果 `step` 是负数），循环结束。如果所有的参数都是整数，则这个循环使用一个整数计数器来计数。如果其中任何一个参数是浮点数，它会把所有参数都转换成浮点数，同时循环会执行 $[n + n * \infty] + 1$ 次，在这里 $n = (\text{end_num}-\text{num})/\text{step}$ 。否则循环从 `num` 开始，使用`<或者>`操作符来对这个计数器和 `end_num` 进行比较，同时使用`+`操作符增加它自己。

```
1.step(10, 2) { |i| print i, " " }
Math::E.step(Math::PI, 0.2) { |f| print f, " " }
```

输出结果：

```
1 3 5 7 9
2.71828182845905 2.91828182845905 3.11828182845905
```

to_int*num.to_int → int*

调用子类的 `to_i` 方法把 `num` 转换成整数。

truncate*num.truncate → int*

把 `num` 截断成一个整数，返回它。`Numeric` 类通过把它的值转换成浮点数并调用 `Float#truncate` 来实现它。

zero?*num.zero? → true 或 false*

如果 `num` 为 0，则返回 `true`。

Class Object

它的子类：Array、Binding、Continuation、Data（用在解释器内部）、Dir、Exception、FalseClass、File::Stat、Hash、IO、MatchData、Method、Module、NilClass、Numeric、Proc、Process::Status、Range、Regexp、String、Struct、Symbol、Thread、ThreadGroup、Time、TrueClass、UnboundMethod。

`Object` 是 Ruby 中所有类的父类。因此它的方法对所有对象都可用，除非它们显式地在子类中被覆写（override）。

`Object` 混合在 `Kernel` 模块中，使得这些内建的内核函数变得可用的全局函数。尽管 `Kernel` 模块定义了 `Object` 的实例方法，我们选择在这里说明它们以避免任何混淆。

在接下来的描述中，参数 `symbol` 指的是一个符号，它是一个引用字符串或者一个 `Symbol`（比如：`:name`）。

实例方法

`==` $obj == other_obj \rightarrow \text{true 或 false}$

等式（Equality）——在 `Object` 层次，只有当 `obj` 和 `other_obj` 对象相同，`==` 才返回 `true`。通常它会在派生类中被覆写，以提供类特定的功能（含义）。

`====` $obj === other_obj \rightarrow \text{true 或 false}$

Case 等式——等同于 `Object#==`，但是通常会在派生类中被覆写，以在 `case` 语句中提供有意义的语义。

`=~` $obj =~ other_obj \rightarrow \text{false}$

模式匹配（pattern match）——在派生类（尤其是 `Regexp` 和 `String` 类）中被覆写，以提供有意义的模式匹配语义。

`__id__` $obj.__id__ \rightarrow \text{fixnum}$

等同于 `Object#object_id`。

`__send__` $obj.__send__(symbol <, args >^<, &block >) \rightarrow other_obj$

等同于 `Object#send`。

`class` $obj.class \rightarrow klass$

返回 `obj` 的类，现在优先于 `Object#type` 使用，因为一个 Ruby 对象的类型只是松耦合到这个对象的类。由于 `class` 也是 Ruby 的一个保留词，所以这个方法总是必须用一个显式的接受者来调用。

| | | |
|-------------------------|---------------|---------------------|
| <code>1.class</code> | \rightarrow | <code>Fixnum</code> |
| <code>self.class</code> | \rightarrow | <code>Object</code> |

clone*obj.clone* → *other_obj*

生成 *obj* 的一个浅 (shallow) 拷贝——拷贝了 *obj* 的实例变量，但是没有拷贝它引用的那些对象。拷贝了 *obj* 的 frozen 和 tainted 的状态。同时参见 Object#dup 的描述。

```
class Klass
  attr_accessor :str
end
s1 = Klass.new      → #<Klass:0x1c9170>
s1.str = "Hello"   → "Hello"
s2 = s1.clone       → #<Klass:0x1c90d0 @str="Hello">
s2.str[1,4] = "i"   → "i"
s1.inspect         → "#<Klass:0x1c9170 @str=\"Hi\">"
s2.inspect         → "#<Klass:0x1c90d0 @str=\"Hi\">"
```

display

obj.display(port=\$>) → nil

在指定端口 (默认端口是 \$>) 打印出 *obj*。相当于下面的代码：

```
def display(port=$>)
  port.write self
end
```

比如：

```
1.display
"cat".display
[ 4, 5, 6 ].display
puts
```

输出结果：

```
1cat456
```

dup

obj.dup → *other_obj*

生成 *obj* 的一个浅 (shallow) 拷贝——拷贝了 *obj* 的实例变量，但是没有拷贝它引用的那些对象。dup 拷贝了 *obj* 的 tainted 状态。同时参见 Object#clone 的描述。一般而言，在派生类中 clone 和 dup 或许有不同的语义。一般使用 clone 来复制一个对象以及它的内部状态，而 dup 通常使用派生对象 (descendent object) 的类来创建新的实例。

eql?

obj.eql?(other_obj) → true 或 false

如果 *obj* 和 *other_obj* 的值相同，则返回 true。Hash 使用这个方法来测试它的成员是否相等。对 Object 类的对象而言，eql? 等同于 ==。Object 子类一般都会延续这个传统，但是也有例外情形。比如，Numeric 类型通过 == 而不是 eql? 方法来转换类型。所以：

```
1 == 1.0      → true
1.eql? 1.0    → false
```

equal?*obj.equal?(other_obj) → true 或 false*

如果 *obj* 和 *other_obj* 的对象 ID 相同，则返回 *true*。它不能够在子类中被覆盖。

```
a = ['cat', 'dog']
b = ['cat', 'dog']
a == b                      → true
a.object_id == b.object_id   → false
a.eql?(b)                   → true
a.equal?(b)                  → false
```

extend

obj.extend(<mod>+) → obj

把作为参数传入模块中的每个实例方法添加到 *obj* 中。同时参见 `Module#extend_object`。

```
module Mod
  def hello
    "Hello from Mod.\n"
  end
end

class Klass
  def hello
    "Hello from Klass.\n"
  end
end

k = Klass.new
k.hello      →      "Hello from Klass.\n"
k.extend(Mod) →      #<Klass:0x1ce300>
k.hello      →      "Hello from Mod.\n"
```

编写 `obj.extend(Mod)` 基本上与下面的代码相同。

```
class <>obj
  include Mod
end
```

freeze

obj.freeze → obj

阻止对 *obj* 的进一步修改。如果试图修改 *obj*，则会引发 `TypeError`。你不能解冻（`unfreeze`）一个已冻结（`frozen`）的对象。同时参见 `Object#frozen?`。

```
a = ["a", "b", "c"]
a.freeze
a << "z"
```

输出结果：

```
prog.rb:3:in `<<': can't modify frozen array (TypeError)
from prog.rb:3
```

frozen?*obj.frozen? → true 或 false*

返回 *obj* 的 freeze 状态。

```
a = [ "a", "b", "c" ]
a.freeze      → [ "a", "b", "c" ]
a.frozen?     → true
```

hash

obj.hash → fixnum

生成这个对象的一个 Fixnum 散列值。这个函数必须包含一个属性，它的 *a.eql?(b)* 指的是 *a.hash == b.hash*。这个散列值会被 Hash 类使用。任何超出 Fixnum 范围的散列值在使用之前会被截断。

id*obj.id → fixnum*

1.8.

Object#object_id 为将要废弃的版本。

initialize_copy*obj.initialize_copy(other) → other_obj 或 obj*

1.8.

作为 *Object#dup* 和 *Object#clone* 所使用协议的一部分，*initialize_copy* 作为一个回调函数会被调用，它应当拷贝无法经由 *dup* 和 *clone* 自身拷贝的所有状态信息。通常这只有在编写 C 的扩展时才有用。可以把 *initialize_copy* 想象为某种拷贝构造函数。

inspect*obj.inspect → string*

返回表示 *obj* 的一个可读字符串。如果没有被覆写，则它使用 *to_s* 方法来生成这个字符串。

```
[ 1, 2, 3..4, 'five' ].inspect → "[1, 2, 3..4, \"five\"]"
Time.new.inspect → "Thu Aug 26 22:37:49 CDT 2004"
```

instance_eval

obj.instance_eval(string <, file <, line >>) → other_obj
obj.instance_eval { block } → other_obj

在这个接受者 (*obj*) 的上下文中，对含有 Ruby 代码的字符串或者给定的 *block* 进行求解。为了设置上下文，当执行代码时 *self* 变量被设置为 *obj*，这样代码可以访问 *obj* 的实例变量。在接受 *String* 为参数的 *instance_eval* 版本中，可选的第二和第三个参数提供了文件名以及起始行号，这个行号用来报告任何编译错误。

```
class Klass
  def initialize
    @secret = 99
  end
end
k = Klass.new
k.instance_eval { @secret } → 99
```

instance_of? *obj.instance_of?(klass) → true 或 false*

如果 *obj* 是指定类的实例，则返回 *true*。同时参见 *Object#kind_of?*。

instance_variable_get *obj.instance_variable_get(symbol) → other_obj*

- 1.8 返回指定实例变量的值（或者抛出 *NameError* 异常）。对常规的实例变量而言，*symbol* 应当包括变量名的@部分。

```
class Fred
  def initialize(p1, p2)
    @a, @b = p1, p2
  end
end
fred = Fred.new('cat', 99)
fred.instance_variable_get(:@a)          → "cat"
fred.instance_variable_get("@b")         → 99
```

instance_variable_set *obj.instance_variable_set(symbol, other_obj) → other_obj*

- 1.8 将名为 *symbol* 的实例变量设置为 *other_obj*，因而打击了类作者尝试提供适当封装的积极性。

```
class Fred
  def initialize(p1, p2)
    @a, @b = p1, p2
  end
end
fred = Fred.new('cat', 99)
fred.instance_variable_set(:@a, 'dog')   → "dog"
fred.inspect                          → "#<Fred:0x1ce4e0 @b=99,
@a='dog'>"
```

instance_variables *obj.instance_variables → array*

返回这个接受者实例变量名的数组。请注意：仅定义访问方法（accessor）并不会创建一个相应的实例变量。

```
class Fred
  attr_accessor :a1
  def initialize
    @iv = 3
  end
end
Fred.new.instance_variables → ["@iv"]
```

is_a? *obj.is_a?(klass) → true 或 false*

等同于 *Object#kind_of?*。

kind_of?*obj.kind_of?(klass) → true 或 false*

如果 *klass* 是 *obj* 的类或者 *obj* 的一个超类，或者是包含在 *obj* 中的一个模块，则返回 *true*。

```
module M; end
class A
  include M
end
class B < A; end
class C < B; end
b = B.new
b.instance_of? A    → false
b.instance_of? B    → true
b.instance_of? C    → false
b.instance_of? M    → false
b.kind_of? A        → true
b.kind_of? B        → true
b.kind_of? C        → false
b.kind_of? M        → true
```

method

obj.method(symbol) → meth

在 *obj* 中查询给定的方法，返回一个 *Method* 对象（或者引发 *NameError*）。这个 *Method* 对象作为 *obj* 对象实例中的一个 *closure*（闭包），因此它依然可以访问实例变量和 *self* 的值。

```
class Demo
  def initialize(n)
    @iv = n
  end
  def hello()
    "Hello, @iv = #{@iv}"
  end
end

k = Demo.new(99)
m = k.method(:hello)
m.call → "Hello, @iv = 99"

l = Demo.new('Fred')
m = l.method("hello")
m.call → "Hello, @iv = Fred"
```

method_missing

*obj.method_missing(symbol <, *args >) → other_obj*

当向 *obj* 发送一个它不能处理的消息时，Ruby 会调用 *method_missing*。*symbol* 是这个要被调用方法的符号，同时 *args* 是传递给它的所有参数。下面的例子创建了 *Roman* 类，它响应方法名中包括罗马数字的方法，并返回相应的整数值。*method_missing* 非常典型的用法是实现代理（proxy）、委托程序（delegator）和转发程序（forwarder）。

```

class Roman
  def roman_to_int(str)
    # ...
  end
  def method_missing(method_id)
    str = method_id.id2name
    roman_to_int(str)
  end
end

r = Roman.new
r.iv      → 4
r.xxiii   → 23
r.mm      → 2000

```

methods *obj.methods(regular=true) → array*

- 1.8 如果 *regular* 是 true，则返回 *obj* 以及其祖先中的公开（publicly）可访问的方法名称列表；否则返回 *obj* 的单例（singleton）方法列表。

```

class Klass
  def my_method()
  end
end
k = Klass.new
def k.single
end
k.methods[0..9] → ["dup", "hash", "single", "private_methods",
                   "nil?", "tainted?", "class", "my_method",
                   "singleton_methods", "=~"]
k.methods.length → 42
k.methods(false) → ["single"]

```

nil? *obj.nil? → true 或 false*

除了 nil 以外，所有的对象都返回 false。

object_id *obj.object_id → fixnum*

- 1.8 返回 *obj* 的整数标识符。对于一个给定的对象，无论合适 *object_id* 调用都返回一个相同的数字，没有两个活跃的（active）对象会共享一个 ID。Object#object_id 是一个与 :name 表示法不相同的概念，后者返回 name 的符号 ID。它取代了过时的 Object#id 方法。

private_methods *obj.private_methods → array*

返回 *obj* 中可访问的 private 方法列表。这将包括 *obj* 祖先中的 private 方法，以及任何 mixed-in 模块函数。

protected_methods *obj.protected_methods → array*

返回 *obj* 可访问的 protected 方法列表。

| | |
|-----------------------|-----------------------------------|
| public_methods | <i>obj.public_methods → array</i> |
|-----------------------|-----------------------------------|

等同于 `Object#methods`。

| | |
|--------------------|---|
| respond_to? | <i>obj.respond_to?(symbol, include_priv=false) → true 或 false</i> |
|--------------------|---|

如果 `obj` 能够响应给定的方法，则返回 `true`。只有当可选的第二个参数被求解为 `true` 时，才会在搜索中包括 `private` 方法。

| | |
|-------------|--|
| send | <i>obj.send(symbol <, args >*<, &block >) → other_obj</i> |
|-------------|--|

调用由 `symbol` 标识的方法，把任何参数和 `block` 传递给它。如果 `send` 这个名字和 `obj` 中的已有方法产生了冲突，建议使用 `_send_`。

```
class Klass
  def hello(*args)
    "Hello " + args.join(' ')
  end
end
k = Klass.new
k.send :hello, "gentle", "readers" → "Hello gentle readers"
```

| | |
|--------------------------|--|
| singleton_methods | <i>obj.singleton_methods(all=true) → array</i> |
|--------------------------|--|

返回 `obj` 的单例（`singleton`）方法名称数组。如果可选的参数 `all` 为 `true`，这个列表会包括那些被 `obj` 包括的模块中的方法（在 2004 年 1 月之前的 Ruby 版本中，
1.8 这个参数默认是 `false`）。

```
module Other
  def three() end
end

class Single
  def Single.four() end
end

a = Single.new

def a.one() end

class << a
  include Other
  def two() end
end

Single.singleton_methods → ["four"]
a.singleton_methods(false) → ["two", "one"]
a.singleton_methods(true) → ["two", "one", "three"]
a.singleton_methods → ["two", "one", "three"]
```

| | |
|-------|-------------------------------|
| taint | <i>obj.taint</i> → <i>obj</i> |
|-------|-------------------------------|

把 *obj* 标记为 tainted（非受信的）。如果 \$SAFE 级别大于 0，一些对象会在创建时被标记为 tainted。参见从第 397 页开始的第 25 章。

| | |
|----------|------------------------------------|
| tainted? | <i>obj.tainted?</i> → true 或 false |
|----------|------------------------------------|

如果这个对象是非受信的（tainted），则返回 true。

```
a = "cat"
a.tainted?      →  false
a.taint         →  "cat"
a.tainted?      →  true
a.untaint       →  "cat"
a.tainted?      →  false
```

| | |
|------|-------------------------|
| to_a | <i>obj.to_a</i> → array |
|------|-------------------------|

1.8 返回 *obj* 的数组表示。对于 Object 类以及其他没有显式地覆写这个方法的那些类而言，返回的数组中包括了 self。在 Ruby 1.9 中，to_a 将不再由 Object 类实现——由各个 Object 子类来决定是否提供它们自己的 to_a 实现。

```
self.to_a          →  -:1: warning: default `to_a' will be
                      obsolete\n[main]
"hello".to_a      →  ["hello"]
Time.new.to_a     →  [50, 37, 22, 26, 8, 2004, 4, 239, true, "CDT"]
```

| | |
|------|--------------------------|
| to_s | <i>obj.to_s</i> → string |
|------|--------------------------|

返回 *obj* 的字符串表示。默认的 to_s 打印出这个对象的类和对象 ID 的编码。一个特别的情况是，作为 Ruby 程序的初始执行上下文，这个顶层（top-level）对象会返回“main”。

| | |
|------|-------------------------|
| type | <i>obj.type</i> → klass |
|------|-------------------------|

1.8 等同于 Object#class，不被推荐使用。

| | |
|---------|---------------------------------|
| untaint | <i>obj.untaint</i> → <i>obj</i> |
|---------|---------------------------------|

去除 *obj* 的 taint 状态。

Private 实例方法

| | |
|------------|---------------------------------|
| initialize | <i>initialize(<arg>*)</i> |
|------------|---------------------------------|

initialize 作为对象构造的第三步也是最后一步被调用，它负责设置新对象的初始状态。你可以像在其他语言中使用构造函数那样，来使用 initialize 方法。如果你继承了 Object 之外的类，你可能想调用 super 以调用父类的 initializer⁷。

⁷ 译注：super 将调用父类中的同名方法，参见 super。

```

class A
  def initialize(p1)
    puts "Initializing A: p1 = #{p1}"
    @var1 = p1
  end
end
class B < A
  attr_reader :var1, :var2
  def initialize(p1, p2)
    super(p1)
    puts "Initializing B: p2 = #{p2}"
    @var2 = p2
  end
end
b = B.new("cat", "dog")
puts b.inspect

```

输出结果：

```

Initializing A: p1 = cat
Initializing B: p2 = dog
#<B:0x1c9224 @var2=" dog ", @var1=" cat ">

```

remove_instance_variable

remove_instance_variable(symbol)→ other_obj

- 1.8 从 *obj* 删除给定的实例变量，并返回这个变量的值。

```

class Dummy
  attr_reader :var
  def initialize
    @var = 99
  end
  def remove
    remove_instance_variable(:@var)
  end
end
d = Dummy.new
d.var → 99
d.remove → 99
d.var → nil

```

singleton_method_added

singleton_method_added(symbol)

- 1.8 无论何时向这个接受者添加一个单例 (singleton) 方法，*singleton_method_added* 作为回调函数会被调用。

```

module Chatty
  def Chatty.singleton_method_added(id)
    puts "Adding #{id.id2name} to #{self.name}"
  end
  def self.one() end
  def two() end
end
def Chatty.three() end

```

```

obj = "cat"
def obj.singleton_method_added(id)
  puts "Adding #{id.id2name} to #{self}"
end

def obj.speak
  puts "meow"
end

```

输出结果：

```

Adding singleton_method_added to Chatty
Adding one to Chatty
Adding three to Chatty
Adding singleton_method_added to cat
Adding speak to cat

```

singleton_method_removed

singleton_method_removed(symbol)

- 1.8 无论何时从这个接受者删除了一个 `singleton` 方法，`singleton_method_removed` 作为回调函数会被调用。

```

module Chatty
  def Chatty.singleton_method_removed(id)
    puts "Removing #{id.id2name}"
  end
  def self.one()      end
  def two()         end
  def Chatty.three() end
  class <<self
    remove_method :three
    remove_method :one
  end
end

```

输出结果：

```

Removing three
Removing one

```

singleton_method_undefined

singleton_method_undefined(symbol)

- 1.8 当一个 `singleton` 方法在这个接受者中没有定义时，`singleton_method_undefined` 作为回调函数会被调用。

```

module Chatty
  def Chatty.singleton_method_undefined(id)
    puts "Undefining #{id.id2name}"
  end
  def Chatty.one() end
  class << self
    undef_method(:one)
  end
end

```

输出结果：

```

Undefining one

```

Module ObjectSpace

`ObjectSpace` 模块包含了许多与垃圾回收设施相互作用的函数，它们允许你使用迭代器来遍历所有活跃（living）对象。

`ObjectSpace` 也提供了对对象析构器（finalizers）的支持。当特定的对象即将被垃圾回收销毁时，相应的 `procs` 将会被调用。

```
include ObjectSpace

a, b, c = "A", "B", "C"
puts "a's id is #{a.object_id}"
puts "b's id is #{b.object_id}"
puts "c's id is #{c.object_id}"

define_finalizer(a, lambda {|id| puts "Finalizer one on #{id}" })
define_finalizer(b, lambda {|id| puts "Finalizer two on #{id}" })
define_finalizer(c, lambda {|id| puts "Finalizer three on #{id}" })
```

输出结果：

```
a's id is 936150
b's id is 936140
c's id is 936130
Finalizer three on 936130
Finalizer two on 936140
Finalizer one on 936150
```

模块方法

| | |
|----------------|--|
| <u>_id2ref</u> | ObjectSpace._id2ref(<i>object_id</i>) → <i>obj</i> |
|----------------|--|

把对象 ID 转换成对象的引用。可能会对传入析构器的对象 ID 参数，调用该方法。

```
s = "I am a string"          →      "I am a string"
oid = s.object_id             →      936550
r = ObjectSpace._id2ref(oid)   →      "I am a string"
r                           →      "I am a string"
r.equal?(s)                  →      true
```

| | |
|-------------------------|---|
| <u>define_finalizer</u> | ObjectSpace.define_finalizer(<i>obj</i> , <i>a_proc=proc()</i>) |
|-------------------------|---|

把 *a_proc* 作为析构器（finalizer）添加了，它在 *obj* 即将被销毁时会被调用。

| | |
|--------------------|--|
| <u>each_object</u> | ObjectSpace.each_object(< <i>class_or_mod</i> >) { <i>obj</i> <i>block</i> } → <i>fixnum</i> |
|--------------------|--|

对 Ruby 进程中每个活跃（living）且非立即值的（nonimmediate）对象调用一次 *block*。如果指定了 *class_or_mod*，它只会对那些匹配 *class_or_mod* 或 *class_or_mod* 子类的类或者模块调用 *block*。返回找到的对象个数。永远不会返回立即值的（immediate）对象（Fixnums、Symbols `true`、`false` 和 `nil`）。在下面的例子中，`each_object` 返回了我们定义的数字和定义在 `Math` 模块中的几个常量。

```
a = 102.7
b = 95    # Fixnum: won't be returned
c = 12345678987654321
count = ObjectSpace.each_object(Numeric) { |x| p x }
puts "Total count: #{count}"
```

输出结果：

```
12345678987654321
102.7
2.71828182845905
3.14159265358979
2.22044604925031e-16
1.79769313486232e+308
2.2250738585072e-308
Total count: 7
```

garbage_collect

ObjectSpace.garbage_collect→nil

启动垃圾回收（参见第 491 页的 GC 模块）。

undefine_finalizer

ObjectSpace.undefine_finalizer(*obj*)

删除 *obj* 中的所有析构器。



Class **Proc** < **Object**

`Proc` 对象是绑定到一组局部变量的代码 `block`。一旦绑定，这些代码可能会在不同的上下文中被调用，同时它们仍然可以访问这些变量。

```
def gen_times(factor)
  return Proc.new {|n| n*factor }
end

times3 = gen_times(3)
times5 = gen_times(5)

times3.call(12)          → 36
times5.call(5)          → 25
times3.call(times5.call(4)) → 60
```

类方法

| | |
|------------|--|
| new | <code>Proc.new { block } → a_proc</code>
<code>Proc.new → a_proc</code> |
|------------|--|

创建新的 `Proc` 对象，把它绑定到当前的上下文。只有在附带有（attached）`block` 的方法中，才可以不带 `block` 调用 `Proc.new`，在这种情况下，这个 `block` 会转换成 `Proc` 对象。

```
def proc_from
  Proc.new
end
proc = proc_from { "hello" }
proc.call → "hello"
```

实例方法

| | |
|-----------|---|
| [] | <code>prc[<params>*] → obj</code> |
|-----------|---|

等同于 `Proc.call`。

| | |
|-----------|--|
| == | <code>prc==other → true 或 false</code> |
|-----------|--|

1.8 如果 `prc` 和 `other` 相同，则返回 `true`。

| | |
|--------------|----------------------------------|
| arity | <code>prc.arity → integer</code> |
|--------------|----------------------------------|

1.8 返回 `block` 所需的参数个数。如果这个 `block` 声明为不带任何参数，则返回 0。如果确切知道 `block` 接受 `n` 个参数，则返回 `n`。如果 `block` 有可选参数，则返回 `-(n+1)`，这里 `n` 是必需参数的个数。没有参数声明的 `proc` 也返回 `-1`，因为它可以接受（和忽视）任意个数目的参数。

```

Proc.new {}.arity      → -1
Proc.new {||}.arity    → 0
Proc.new {|a|}.arity   → 1
Proc.new {|a,b|}.arity → 2
Proc.new {|a,b,c|}.arity → 3
Proc.new {|*a|}.arity   → -1
Proc.new {|a,*b|}.arity → -2

```

1.8 在 Ruby 1.9 中, `arity` 被定义为无法被忽视的参数的个数。

在 Ruby 1.8 中, `Proc.new{}.arity` 返回-1, 而在 1.9 中它返回 0。

| | |
|----------------|-------------------------------------|
| binding | <i>prc.binding</i> → <i>binding</i> |
|----------------|-------------------------------------|

1.8 返回与 `prc` 关联的绑定。请注意: `Kernel#eval` 接受一个 `Proc` 或者绑定对象作为它的第二个参数。

```

def fred(param)
  lambda {}
end

b = fred(99)
eval("param", b.binding) → 99
eval("param", b) → 99

```

| | |
|-------------|---|
| call | <i>prc.call(<params>*)</i> → <i>obj</i> |
|-------------|---|

调用这个 `block`, 使用某种接近于方法调用的语义把这个 `block` 的参数设置为 `params` 的值。返回 `block` 中最后一个被求解的表达式的值。

```

a_proc = Proc.new {|a, *b| b.collect{|i| i*a}}
a_proc.call(9, 1, 2, 3) → [9, 18, 27]
a_proc[9, 1, 2, 3] → [9, 18, 27]

```

1.8 如果被调用的这个 `block` 显式声明了只接受一个参数, `call` 会抛出一个警告, 除非我们只为它指定一个参数; 否则, 它会很高兴地接受给定的任何参数, 忽视传入的多余参数, 并且把未设置的 `block` 参数设置为 `nil`。

```
a_proc = Proc.new {|a| a}
a_proc.call(1,2,3)
```

输出结果:

```
prog.rb:1: warning: multiple values for a block parameter (3 for 1)
from prog.rb:2
```

如果你希望 `block` 接受任意数目的参数, 则把它定义成接受`*args`。

```
a_proc = Proc.new {|*a| a}
a_proc.call(1,2,3) → [1, 2, 3]
```

使用 `Kernel.lambda` 创建的 `block` 会检查, 它们调用时使用了完全一致的参数个数。

```
p_proc = Proc.new {|a,b| puts "Sum is: #{a + b}" }
p_proc.call(1,2,3)
p_proc = lambda {|a,b| puts "Sum is: #{a + b}" }
p_proc.call(1,2,3)
```

输出结果：

```
Sum is: 3
prog.rb:3: wrong number of arguments (3 for 2) (ArgumentError)
from prog.rb:3:in `call'
from prog.rb:4
```

| | |
|---------|--------------------------|
| to_proc | <i>prc.to_proc → prc</i> |
|---------|--------------------------|

1.8 将对象转换成 Proc 对象协议的一部分。Proc 类的实例简单地返回自己。

| | |
|------|--------------------------|
| to_s | <i>prc.to_s → string</i> |
|------|--------------------------|

返回 *prc* 的描述，包括在何处定义它的信息。

```
def create_proc
  Proc.new
end

my_proc = create_proc { "hello" }
my_proc.to_s → "#<Proc:0x001c7abc@prog.rb:5>"
```

Module Process

Process 模块是一个用来操作进程的方法集合（collection）。希望操作真实和有效用户 ID 以及真实和有效进程组 ID 的程序，也应当看看 Process::GID 和 Process::UID 模块。很多在这里介绍的功能都会在 Process::Sys 模块中重复介绍。

模块常量

| | |
|---------------|-------------------------------------|
| PRIOR_PGRP | 进程组优先级。 |
| PRIOR_PROCESS | 进程优先级。 |
| PRIOR_USER | 用户优先级。 |
| WNOHANG | 如果没有子进程退出，则不会阻塞（block）。不是在所有平台上都可用。 |
| WUNTRACED | 返回被停止的子进程。不是在所有平台上都可用。 |

模块函数

| | |
|-------|-----------------------|
| abort | abort
abort(msg) |
| 1.8 | 等同于 Kernel.abort。 |

| | |
|--------|---|
| detach | Process.detach(pid) → thread |
| 1.8 | 一些操作系统会一直保留被终止子进程的状态，直到父进程收集（collect）这个状态（通常使用 wait() 的某些变种）为止。如果父进程永远不会收集它，子进程会作为一个僵死（zombie）进程一直存在。Process.detach 通过运行一个独立的 Ruby 线程来防止出现这种情况，这个线程唯一的任务是当进程 pid 终止时，收集它的状态。只有当你不打算明确等待子进程终止时，才使用 detach。detach 只是定时地检查状态（当前是每秒检查一次）。 |

在第一个例子中，没有收集第一个子进程的状态，所以它在进程状态中显示为一个僵死进程。

```
pid = fork { sleep 0.1 }
sleep 1
system("ps -o pid,state -p #{pid}")
```

输出结果：

```
PID  STAT
27836  ZN+
```

在下面的例子中，使用 Process.detach 来自动地收集子进程——不会有子进程一直运行。

```
pid = fork { sleep 0.1 }
Process.detach(pid)
sleep 1
```

```
system("ps -o pid,state -p #{pid}")
```

输出结果：

```
PID STAT
```

| | |
|-------------|---------------------------|
| egid | Process.egid → int |
|-------------|---------------------------|

返回进程的有效组 ID。

```
Process.egid → 502
```

| | |
|--------------|--------------------------------|
| egid= | Process.egid= int → int |
|--------------|--------------------------------|

设置进程的有效组 ID。

| | |
|-------------|---------------------------|
| euid | Process.euid → int |
|-------------|---------------------------|

返回进程的有效用户 ID。

```
Process.euid → 502
```

| | |
|--------------|--------------------------|
| euid= | Process.euid= int |
|--------------|--------------------------|

设置进程的有效用户 ID。不是在所有平台上都可用。

| | |
|-------------|------------------------------|
| exit | Process.exit(int=0) |
|-------------|------------------------------|

1.8 等同于 Kernel.exit。

| | |
|--------------|---|
| exit! | Process.exit!(true false status=1) |
|--------------|---|

1.8 等同于 Kernel.exit!。不运行任何 exit 处理程序（handler）。把 0、1 或者 status 作为退出状态返回给底层系统。

```
Process.exit!(0)
```

| | |
|-------------|---|
| fork | Process.fork < { block } > → int 或 nil |
|-------------|---|

参见第 522 页的 Kernel.fork。

| | |
|----------------|------------------------------------|
| getpgid | Process.getpgid(int)→ int |
|----------------|------------------------------------|

返回给定进程 id 的进程组 ID。不是在所有平台上都可用。

```
Process.getpgid(Process.ppid()) → 25122
```

| | |
|----------------|------------------------------|
| getpgrp | Process.getpgrp → int |
|----------------|------------------------------|

返回进程的进程组 ID。不是在所有平台上都可用。

```
Process.getpgid(0) → 25122
Process.getpgrp → 25122
```

| | |
|---|--|
| getpriority | Process.getpriority(<i>kind</i>, <i>int</i>) → <i>int</i> |
| 得到指定的进程、进程组或用户的调度优先级。 <i>kind</i> 指出要查找的实体类型：
Process::PRIO_PGRP、Process::PRIO_USER 或者 Process::PRIO_PROCESS 其中之一。 <i>int</i> 指出特定的进程 ID、进程组 ID 或者用户 ID（ID 为 0，指的是当前的进程、进程组或者用户）。较低优先级的进程会优先得到调度。不是在所有平台上都可用。 | |
| <code>Process.getpriority(Process::PRIO_USER, 0) → 19</code>
<code>Process.getpriority(Process::PRIO_PROCESS, 0) → 19</code> | |
| gid | Process.gid → <i>int</i> |
| 返回进程的组 ID。

<code>Process.gid → 502</code> | |
| gid= | Process.gid= <i>int</i> → <i>int</i> |
| 设置进程的组 ID。

<code>Process.gid= 502 → 502</code> | |
| groups | Process.groups → <i>groups</i> |
| <u>1.8</u> | 返回一个辅助（supplementary）组 ID 的整数数组。不是在所有平台上都可用。
同时参见 <code>Process.maxgroups</code> 。

<code>Process.groups → [502, 79, 80, 81]</code> |
| groups= | Process.groups = <i>array</i> → <i>groups</i> |
| <u>1.8</u> | 用给定的数组设置这个辅助组 ID，这个数组可能含有进程组名称（作为字符串）的个数。不是在所有平台上都可用。只有超级用户才可以使用这个方法。同时参见 <code>Process.maxgroups</code> 。

<code>Process.groups= [502] → 502</code> |
| initgroups | Process.initgroups(<i>user</i>, <i>base_group</i>) → <i>groups</i> |
| <u>1.8</u> | 使用操作系统的 <code>initgroups</code> 调用来初始化进程组访问列表。不是在所有平台上都可用。可能需要超级用户特权。

<code>Process.initgroups("dave", 500)</code> |
| kill | Process.kill(<i>signal</i>, <<i>pid</i>>+) → <i>int</i> |
| 发送给定信号到这个指定的进程 ID，如果 <i>pid</i> 是 0，则发送给当前进程。 <i>signal</i> 可能是一个整数信号数字或者 POSIX 信号名（带或者不带 SIG 前缀）。如果 <i>signal</i> 是负数（或者以一个减号开始），杀死（kill）进程组而不是进程。不是所有信号在所有平台上都可用。

<code>pid = fork do</code>
<code>Signal trap("USR1") { puts "Ouch!"; exit }</code>
<code># ... do some work ...</code>
<code>end</code> | |

```
# ...
Process.kill("USR1", pid)
Process.wait
```

输出结果：

```
-:2: SIGUSR1 (SignalException)
from -:1:in `fork'
from -:1
```

maxgroups**Process.maxgroups→count**

1.8 Process 模块对于它在 `Process.groups` 和 `Process.groups=` 调用中支持的辅助组的个数有一个上限。`maxgroups` 调用返回这个上限值（默认是 32），而 `maxgroups=` 调用设置它。

```
Process.maxgroups      → 32
Process.maxgroups = 64
Process.maxgroups      → 64
```

maxgroups=**Process.maxgroups=limit→count**

1.8 设置可以被 `groups` 和 `groups=` 方法处理的辅助组 ID 的最大值。如果给定的上限大于 4096，将使用 4096。

pid**Process.pid→int**

返回这个进程的进程 ID。不是在所有平台上都可用。

```
Process.pid → 27864
```

ppid**Process.ppid→int**

返回这个进程的父进程的进程 ID。在 Windows 上总是返回 0。不是在所有平台上都可用。

```
puts "I am #{Process.pid}"
Process.fork { puts "Dad is #{Process.ppid}" }
```

输出结果：

```
I am 27866
Dad is 27866
```

setpgid**Process.setpgid(pid, int)→0**

把 `pid` 进程的进程组 ID 设置为 `int` 参数。不是在所有平台上都可用。

setpgrp**Process.setpgrp→0**

等同于 `setpgid(0, 0)`。不是在所有平台上都可用。

| | |
|--------------------|--|
| setpriority | Process.setpriority(<i>kind, int, int_priority</i>) → 0 |
| | 参见 Process#getpriority。 |
| | Process.setpriority(Process::PRIO_USER, 0, 19) → 0
Process.setpriority(Process::PRIO_PROCESS, 0, 19) → 0
Process.getpriority(Process::PRIO_USER, 0) → 19
Process.getpriority(Process::PRIO_PROCESS, 0) → 19 |
| setsid | Process.setsid → <i>int</i> |
| | 把这个进程设置为新的会话和进程组组长，同时没有控制终端（controlling tty）。返回会话 ID。不是在所有平台上都可用。 |
| | Process.setsid → 27877 |
| times | Process.times → struct_tms |
| <u>1.8.</u> | 返回一个 Tms 结构（参见第 630 页的 Struct::Tms），它含有这个进程的用户时间和系统 CPU 时间。
t = Process.times
[t.utime, t.stime] → [0.01, 0.01] |
| uid | Process.uid → <i>int</i> |
| | 返回这个进程的用户 ID。
Process.uid → 502 |
| uid= | Process.uid= <i>int</i> → numeric |
| | 设置这个进程的（整数）用户 ID。不是在所有平台上都可用。 |
| wait | Process.wait → <i>int</i> |
| | 等待任何子进程退出并返回子进程的进程 ID。同时将\$?变量设置为含有子进程信息的 Process::Status 对象。如果不存在子进程，则引发 SystemError。不是在所有平台上都可用。
Process.fork { exit 99 } → 27878
Process.wait → 27878
\$?.exitstatus → 99 |
| waitall | Process.waitall → [[<i>pid1,status</i>], ...] |
| <u>1.8.</u> | 等待所有子进程退出，返回一个 <i>pid/status</i> 对数组（这里 <i>status</i> 是 Process::Status 类的一个对象）。 |

```

fork { sleep 0.2; exit 2 }      → 27881
fork { sleep 0.1; exit 1 }      → 27882
fork { exit 0 }                → 27883
Process.waitall                → [[27883, #<Process::Status:  

                                         pid=27883,exited(0)>], [27882,  

                                         #<Process::Status:  

                                         pid=27882,exited(1)>], [27881,  

                                         #<Process::Status:  

                                         pid=27881,exited(2)>]]

```

wait2**Process.wait2→[pid, status]**

- 1.8 等待任何子进程退出，并返回一个数组，含有退出子进程的进程 ID 以及它的退出状态（一个 `Process::Status` 对象）。如果不存在子进程，引发 `SystemError`。

```

Process.fork { exit 99 }      → 27886
pid, status = Process.wait2
pid                      → 27886
status.exitstatus         → 99

```

waitpid**Process.waitpid(pid, int=0)→pid**

根据 `pid` 的值来等待子进程退出：

- <-1 任何进程组 ID 等于 `pid` 绝对值的子进程。
- 1 任何子进程（与 `wait` 方法相同）。
- 0 任何进程组 ID 等于当前进程的进程组 ID 的子进程。
- >0 给定 PID 的子进程。

`int` 参数可能是标志 `Process::WNOHANG`（如果没有已退出的子进程，不会阻塞）或者 `Process::WUNTRACED`（返回未报告的已停止子进程）的逻辑或。并非所有的标志值在所有平台上都可用，但是值为 0 的标志在所有平台上都可以工作。

```

include Process
pid = fork { sleep 3 }          → 27889
Time.now                         → Thu Aug 26 22:38:17 CDT 2004
waitpid(pid, Process::WNOHANG)  → nil
Time.now                         → Thu Aug 26 22:38:17 CDT 2004
waitpid(pid, 0)                 → 27889
Time.now                         → Thu Aug 26 22:38:20 CDT 2004

```

waitpid2**Process.waitpid2(pid, int=0)→[pid, status]**

等待给定的子进程退出，返回这个子进程的进程 ID 以及退出状态（一个 `Process::Status` 对象）。`int` 参数可能是 `Process::WNOHANG`（如果没有任何已退出的子进程，不会阻塞）或者 `Process::WUNTRACED`（返回未报告的已停止子进程）的逻辑或。不是所有的标志在所有平台上都可用，但是值为 0 的标志在所有平台上都可以工作。

Module Process::GID

对底层操作系统真实的、有效的和保存的（saved⁸）组 ID 概念，提供了更高级别的（同时更易移植的）接口。讨论这些 ID 的语义远远超出了本书的范围：对此感兴趣的读者应当参考 POSIX 文档或者阅读最近任何 Unix 平台上的 `intro(2)` 手册页。如果宿主操作系统没有支持一个足够的调用集合，所有那些方法会抛出 `NotImplementedError`。接下来的描述是基于 Hidetoshi Nagai 在 ruby-talk:76218 中给出的注解。

模块方法

| | |
|-------------------------|---|
| change_privilege | Process::GID.change_privilege(<i>gid</i>) → <i>gid</i> |
| <u>1.8</u> | 将真实的、有效的和保存的组 ID 设置为 <i>gid</i> ，如果失败了，则引发一个异常（在这种情况下，不确定这些 ID 的状态）。 |
| | 这个方法与 <code>Process.gid=</code> 方法不兼容。 |
| eid | Process::GID.eid → <i>egid</i> |
| <u>1.8</u> | 返回这个进程的有效组 ID。等同于 <code>Process.egid</code> 。 |
| eid= | Process::GID.eid = <i>egid</i> |
| <u>1.8</u> | 等同于 <code>Process::GID.grant_privilege</code> 。 |
| grant_privilege | Process::GID.grant_privilege(<i>egid</i>) → <i>egid</i> |
| <u>1.8</u> | 把有效的组 ID 设置为 <i>egid</i> ，如果失败了，则引发一个异常。在某种环境下，这可能也会改变保存的组 ID（参见 <code>re_exchangeable?</code> ）。 |
| re_exchange | Process::GID.re_exchange → <i>egid</i> |
| <u>1.8</u> | 互换实际的和有效的组 ID，同时把保存的组 ID 设置为新的有效组 ID。返回新的有效组 ID。 |
| re_exchangeable? | Process::GID.re_exchangeable → true 或 false |
| <u>1.8</u> | 如果在宿主操作系统上实际的和有效的组 ID 可以互换，则返回 true；否则返回 false。 |
| rid | Process::GID.rid → <i>gid</i> |
| <u>1.8</u> | 返回这个进程的实际组 ID。等同于 <code>Process.gid</code> 。 |
| sid_available? | Process::GID.sid_available? → true 或 false |
| <u>1.8</u> | 如果底层平台支持保存的组 ID，则返回 true；否则返回 false。当前，如果 |

⁸ 译注：指的是当 `exec` 执行一个设置了用户或组 ID 的程序文件之后，有效用户或组 ID 将被设置为程序文件的所有者或组 ID，而之前的有效用户或组 ID 则被保存起来。

操作系统有 `setresgid(2)` 或者 `setegid(2)` 调用，或者配置包括了 `POSIX_SAVED_IDS` 标志，Ruby 会假定底层系统支持保存的组 IDs。

| | |
|---------------|--|
| switch | Process::GID.switch→ <i>egid</i>
Process::GID.switch { <i>block</i> }→ <i>obj</i> |
|---------------|--|

- 1.8 处理组特权的转换 (toggling)。在带有 `block` 的形式中，原来的诸 ID 在 `block` 终止时会自动转换回去（但是前提是 `block` 没有调用那些可能会造成干扰的 `Process::GID` 方法）。不带 `block` 的形式返回初始的有效组 ID。

Class Process::Status < Object

`Process::Status` 封装了一个正运行或已终止的系统进程的状态信息。内建的`$?` 变量为 `nil` 或者一个 `Process::Status` 对象。

```
fork { exit 99 } → 27186
Process.wait → 27186
$?.class → Process::Status
$?.to_i → 25344
$? >> 8 → 99
$?.stopped? → false
$?.exited? → true
$?.exitstatus → 99
```

POSIX 系统使用一个 16 位的整数记录进程信息。它的低位记录了进程状态（已停止、退出或被信号中断），高位可能包含附加信息（比如，在进程已退出的情况下，程序的返回码）。在 Ruby 1.8 版本之前，Ruby 程序可以直接访问这些位。现在 Ruby 把它们封装到一个 `Process::Status` 对象中。当然，为了最大可能地与已有 Ruby 程序兼容，这些对象保留了面向位（bit）的处理接口。在接下来的描述中，当谈到 `stat` 的整数值时，我们指的是这个 16 位的整数值。

实例方法

`==` *stat == other* → true 或 false

如果 `stat` 的整数值与 `other` 相等，则返回 `true`。

`&` *stat & num* → fixnum

`stat` 中的位（bit）和 `num` 的逻辑与。

```
fork { exit 0x37 }
Process.wait
sprintf('%04x', $?.to_i) → "3700"
sprintf('%04x', $? & 0x1e00) → "1600"
```

`>>` *stat >> num* → fixnum

把 `stat` 中的位（bit）右移 `num` 位。

```
fork { exit 99 } → 27192
Process.wait → 27192
$?.to_i → 25344
$? >> 8 → 99
```

`coredump?` *stat.coredump* → true 或 false

`stat` 终止时，如果产生了 `coredump`（核心转储），则返回 `true`。不是在所有平台上都可用。

| | |
|----------------------|--|
| <code>exited?</code> | <code>stat.exited? → true 或 false</code> |
|----------------------|--|

如果 `stat` 正常退出，则返回 `true`（比如，使用 `exit` 调用或者结束程序）。

| | |
|-------------------------|---|
| <code>exitstatus</code> | <code>stat.exitstatus → fixnum 或 nil</code> |
|-------------------------|---|

返回 `stat` 返回码的低 8 位。只有当 `exits?` 为 `true` 时可用。

```
fork { }      → 27195
Process.wait  → 27195
$?.exited?   → true
$?.exitstatus → 0

fork { exit 99 } → 27196
Process.wait    → 27196
$?.exited?    → true
$?.exitstatus  → 99
```

| | |
|------------------|--------------------------------|
| <code>pid</code> | <code>stat.pid → fixnum</code> |
|------------------|--------------------------------|

返回与这个 `status` 对象相关联的进程的 ID。

```
fork { exit }  → 27199
Process.wait   → 27199
$?.pid        → 27199
```

| | |
|------------------------|--|
| <code>signaled?</code> | <code>stat.signaled? → true 或 false</code> |
|------------------------|--|

如果 `stat` 因为未捕获的信号而终止，则返回 `true`。

```
pid = fork { sleep 100 }
Process.kill(9, pid)  → 1
Process.wait         → 27202
$?.signaled?        → true
```

| | |
|-----------------------|---|
| <code>stopped?</code> | <code>stat.stopped? → true 或 false</code> |
|-----------------------|---|

如果这个进程已停止，则返回 `true`。只有当相应的 `wait` 调用设置了 `WUNTRACED` 标志时，它才会返回。

| | |
|-----------------------|---|
| <code>success?</code> | <code>stat.success? → nil 或 true 或 false</code> |
|-----------------------|---|

如果 `stat` 指的是一个已成功退出的进程，则返回 `true`。如果指的是退出时失败的进程，则返回 `false`。如果指的是还没有退出的进程，则返回 `nil`。

| | |
|----------------------|--|
| <code>stopsig</code> | <code>stat.stopsig → fixnum 或 nil</code> |
|----------------------|--|

返回导致 `stat` 停止的信号个数（若其本身未停止，则返回 `nil`）。

termsig*stat.termsig*→fixnum 或 nil

返回导致 *stat* 终止的信号个数（若其本身不是因为未捕获的信号终止的，则返回 nil）。

to_i

stat.to_i→fixnum

把 *stat* 中的这些位（bits）返回为一个 Fixnum。查看这些位是与平台相关的。

```
fork { exit 0xab }      → 27205
Process.wait           → 27205
sprintf('%04x', $?.to_i) → "ab00"
```

to_int

stat.to_int→fixnum

等同于 `Process::Status#to_i`。

to_s

stat.to_s→string

等同于 *stat.to_i.to_s*。



Module **Process::Sys**

`Process::Sys` 为进程用户和进程组环境提供了系统调用级别的访问。很多调用是 `Process` 模块中的那些方法的别名，我们把它们打包在这个模块中来提供一个完全的方法列表。对于更高级别的（同时更易移植的）接口，同时参见 `Process::GID` 和 `Process::UID`。

模块方法

| | |
|--------------------|---|
| getegid | <code>Process::Sys.getegid → gid</code> |
| <small>1.8</small> | 返回这个进程的有效组 ID。等同于 <code>Process.egid</code> 。 |
| geteuid | <code>Process::Sys.geteuid → uid</code> |
| <small>1.8</small> | 返回这个进程的有效用户 ID。等同于 <code>Process.euid</code> 。 |
| getgid | <code>Process::Sys.getgid → gid</code> |
| <small>1.8</small> | 返回这个进程的组 ID。等同于 <code>Process.gid</code> 。 |
| getuid | <code>Process::Sys.getuid → uid</code> |
| <small>1.8</small> | 返回这个进程的用户 ID。等同于 <code>Process.uid</code> 。 |
| issetugid | <code>Process::Sys.issetugid → true 或 false</code> |
| <small>1.8</small> | 如果这个进程因为最后一个 <code>execve()</code> 系统调用而导致 <code>setuid</code> 或者 <code>setgid</code> 被调用，则返回 <code>true</code> ；如果没有，则返回 <code>false</code> 。在不支持 <code>issetugid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setegid | <code>Process::Sys.setegid(gid)</code> |
| <small>1.8</small> | 把有效的组 ID 设置为 <code>gid</code> ，如果底层的系统调用失败，则 <code>setegid</code> 调用会失败。在不支持 <code>setegid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| seteuid | <code>Process::Sys.seteuid(uid)</code> |
| <small>1.8</small> | 把有效的用户 ID 设置为 <code>uid</code> ，如果底层的系统调用失败，则 <code>seteuid</code> 调用会失败。在不支持 <code>seteuid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setgid | <code>Process::Sys.setgid(gid)</code> |
| <small>1.8</small> | 把组 ID 设置为 <code>gid</code> ，如果底层的系统调用失败，则 <code>setgid</code> 调用会失败。在不支持 <code>setgid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setregid | <code>Process::Sys.setregid(rgid, egid)</code> |
| <small>1.8</small> | 把真实的和有效的组 ID 分别设置为 <code>rgid</code> 和 <code>egid</code> ，如果底层的系统调用失败， <code>setregid</code> 调用会失败。在不支持 <code>setregid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |

| | |
|------------------|--|
| setresgid | Process::Sys.setresgid(<i>rgid</i>, <i>egid</i>, <i>sgid</i>) |
| 1.8 | 把真实的、有效的和保存的 (saved) 组 ID 分别设置为 <i>rgid</i> 、 <i>egid</i> 和 <i>sgid</i> ，如果底层的系统调用失败， <code>setresgid</code> 调用会失败。在不支持 <code>setresgid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setresuid | Process::Sys.setresuid(<i>ruid</i>, <i>euid</i>, <i>suid</i>) |
| 1.8 | 把真实的、有效的和保存的用户 ID 分别设置为 <i>ruid</i> 、 <i>euid</i> 和 <i>suid</i> ，如果底层的系统调用失败， <code>setresuid</code> 调用会失败。在不支持 <code>setresuid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setreuid | Process::Sys.setreuid(<i>ruid</i>, <i>euid</i>) |
| 1.8 | 把真实的和有效的用户 ID 分别设置为 <i>ruid</i> 和 <i>euid</i> ，如果底层的系统调用失败， <code>setreuid</code> 调用会失败。在不支持 <code>setreuid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setrgid | Process::Sys.setrgid(<i>rgid</i>) |
| 1.8 | 把真实的组 ID 设置为 <i>rgid</i> ，如果底层的系统调用失败， <code>setrgid</code> 调用会失败。在不支持 <code>setrgid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setruid | Process::Sys.setruid(<i>ruid</i>) |
| 1.8 | 把真实的用户 ID 设置为 <i>ruid</i> ，如果底层的系统调用失败， <code>setruid</code> 调用会失败。在不支持 <code>setruid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |
| setuid | Process::Sys.setuid(<i>uid</i>) |
| 1.8 | 把用户 ID 设置为 <i>uid</i> ，如果底层的系统调用失败， <code>setuid</code> 调用会失败。在不支持 <code>setuid(2)</code> 的系统上，抛出 <code>NotImplementedError</code> 。 |



Module Process::UID

对底层操作系统的真实的、有效的和保存的用户 ID 概念，提供了更高级别的（同时更易移植的）接口。关于更多的信息，请参见第 589 页的 `Process::GID` 介绍。

模块方法

| | |
|-------------------------------|---|
| <code>change_privilege</code> | <code>Process::UID.change_privilege(uid) → uid</code> |
| <code>1.8</code> | 把真实的、有效的和保存的用户 ID 设置为 <code>uid</code> ，如果失败了，引发一个异常（在这种情况下，不确定这些 ID 的状态）。它与 <code>Process.uid=</code> 方法不兼容。 |
| <code>eid</code> | <code>Process::UID.eid → euid</code> |
| <code>1.8</code> | 返回这个进程的有效用户 ID。等同于 <code>Process.euid</code> 。 |
| <code>eid=</code> | <code>Process::UID.eid = euid</code> |
| <code>1.8</code> | 等同于 <code>Process::UID.grant_privilege</code> 。 |
| <code>grant_privilege</code> | <code>Process::UID.grant_privilege(euid) → euid</code> |
| <code>1.8</code> | 把有效的用户 ID 设置为 <code>euid</code> ，如果失败了，引发一个异常。在某种环境下，这可能也会改变保存的用户 ID。 |
| <code>re_exchange</code> | <code>Process::UID.re_exchange → euid</code> |
| <code>1.8</code> | 互换真实的和有效的用户 ID，同时把保存的用户 ID 设置为新的有效用户 ID。
返回新的有效用户 ID。 |
| <code>re_exchangeable?</code> | <code>Process::UID.re_exchangeable → true 或 false</code> |
| <code>1.8</code> | 如果在宿主操作系统上真实的和有效的用户 ID 可以互换，则返回 <code>true</code> ；否则返回 <code>false</code> 。 |
| <code>rid</code> | <code>Process::UID.rid → uid</code> |
| <code>1.8</code> | 返回这个进程的实际用户 ID。等同于 <code>Process.uid</code> 。 |
| <code>sid_available?</code> | <code>Process::UID.sid_available? → true 或 false</code> |
| <code>1.8</code> | 如果底层平台支持保存的用户 ID，则返回 <code>true</code> ；否则返回 <code>false</code> 。当前，如果操作系统有 <code>setresuid(2)</code> 或者 <code>seteuid(2)</code> 调用，或者配置包括了 <code>POSIX_SAVED_IDS</code> 标志，Ruby 会假定底层系统支持保存的用户 ID。 |
| <code>switch</code> | <code>Process::UID.switch → euid</code>
<code>Process::UID.switch { block } → obj</code> |
| <code>1.8</code> | 处理用户特权的转换（toggling）。在带有 <code>block</code> 的形式中，当 <code>block</code> 终止时会自动地转换回原来的诸 ID（前提是 <code>block</code> 没有调用可能会造成干扰的其他 <code>Process::UID</code> 方法）。不带 <code>block</code> 的形式，返回初始的有效用户 ID。 |

Class Range < Object

一个区间表示一个间隔——带有开头 (start) 和结尾 (end) 的一组值。区间可能是使用 `s..e` 和 `s...e` 字面量或者 `Range.new` 来构造的。使用 `..` 构造的区间把从开头到结尾的所有值都包括了。使用 `...` 创建的区间把结尾值排除在外。当区间用作一个迭代器时，它会返回这个序列中的每个值。

```
(-1...-5).to_a → []
(-5...-1).to_a → [-5, -4, -3, -2, -1]
('a'...'e').to_a → ["a", "b", "c", "d", "e"]
('a'...'e').to_a → ["a", "b", "c", "d"]
```

可以用任何类型的对象来构造区间，只要这些对象可以使用它们的 `<=>` 操作符来进行比较，同时它们支持 `succ` 方法，返回序列中的下一个对象。

```
class Xs      # represent a string of 'x's
  include Comparable
  attr :length
  def initialize(n)
    @length = n
  end
  def succ
    Xs.new(@length + 1)
  end
  def <=>(other)
    @length <=> other.length
  end
  def to_s
    sprintf "%2d #{inspect}", @length
  end
  def inspect
    'x' * @length
  end
end

r = Xs.new(3)..Xs.new(6) → xxx..xxxxxx
r.to_a                  → [xxx, xxxx, xxxxx, xxxxxx]
r.member?(Xs.new(5))     → true
```

在前面的代码例子中，`Xs` 类包含了 `Comparable` 模块。这是因为 `Enumerable#member?` 使用 `==` 方法来检查相等。包含 `Comparable` 就确保了会根据 `Xs` 中 `<=>` 方法来定义 `==` 方法。

Mixes in

`Enumerable`:

```
all?, any?, collect, detect, each_with_index, entries, find, find_all,
grep, include?, inject, map, max, member?, min, partition, reject,
select, sort, sort_by, to_a, zip
```

类方法

| | |
|------------|---|
| new | Range.new(start, end, exclusive=false) → rng |
|------------|---|

使用给定的 *start* 和 *end* 来构造区间。如果省略了第三个参数或者它是 *false*，这个区间会包括这个 *end* 对象；否则，它会被排除在外。

实例方法

| | |
|-----------|----------------------------------|
| == | rng == obj → true 或 false |
|-----------|----------------------------------|

如果 *obj* 区间的开头和结尾分别和 *rng* 的开头和结尾相同（使用 `==` 方法进行比较），同时它的 *exclusive* 标志和 *rng* 的相同，返回 *true*。

| | |
|-------------|-----------------------------------|
| ==== | rng === val → true 或 false |
|-------------|-----------------------------------|

如果 *rng* 把它的结尾值排除在外，返回 $rng.start \leq val < rng.end$ 。如果 *rng* 包括了它的结尾值，返回 $rng.start \leq val \leq rng.end$ 。请注意：这意味着 *val* 不需要是 *rng* 区间的成员之一（比如，一个浮点数可以落在一个整数区间的开头和结尾值之间）。`====` 操作符可以方便地使用在 `case` 语句中。

```
case 74.95
when 1...50  then puts "low"
when 50...75 then puts "medium"
when 75...100 then puts "high"
end
```

输出结果：

```
medium
```

| | |
|--------------|------------------------|
| begin | rng.begin → obj |
|--------------|------------------------|

返回 *rng* 的第一个对象。

| | |
|-------------|-------------------------------------|
| each | rng.each { i block } → rng |
|-------------|-------------------------------------|

在元素 *rng* 上迭代，把每个元素依次传入到 *block*。使用 `succ` 方法来产生连续元素。

```
(10..15).each do |n|
  print n, ' '
end
```

输出结果：

```
10 11 12 13 14 15
```

| | |
|------------|----------------------|
| end | rng.end → obj |
|------------|----------------------|

返回定义 *rng* 结尾的对象。

| | | |
|---------------------------|---|----|
| <code>(1..10).end</code> | → | 10 |
| <code>(1...10).end</code> | → | 10 |

| | |
|------------|------------------------------------|
| eq? | <i>rng.eq?(obj) → true 或 false</i> |
|------------|------------------------------------|

如果 *obj* 区间的开头和结尾分别和 *rng* 的开头和结尾相同（使用 *eq?* 方法进行比较），同时它的 *exclusive* 标志与 *rng* 的相同，返回 *true*。

| | |
|---------------------|--|
| exclude_end? | <i>rng.exclude_end? → true 或 false</i> |
|---------------------|--|

如果 *rng* 把它的结尾值排除在外，则返回 *true*。

| | |
|--------------|------------------------|
| first | <i>rng.first → obj</i> |
|--------------|------------------------|

等同于 *Range#begin*。

| | |
|-----------------|---|
| include? | <i>rng.include?(val) → true 或 false</i> |
|-----------------|---|

等同于 *Range#==*。

| | |
|-------------|-----------------------|
| last | <i>rng.last → obj</i> |
|-------------|-----------------------|

等同于 *Range#end*。

| | |
|----------------|--|
| member? | <i>rng.member?(val) → true 或 false</i> |
|----------------|--|

如果 *val* 是 *rng* 中的一个值（也就是说，如果 *Range#each* 在某个时候会返回 *val*），则返回 *true*。

```
r = 1..10
r.include?(5)      →  true
r.member?(5)       →  true
r.include?(5.5)    →  true
r.member?(5.5)    →  false
```

| | |
|-------------|--|
| step | <i>rng.step(n=1){ obj block} → rng</i> |
|-------------|--|

- 1.8 在 *rng* 区间上迭代，把它的每 *n* 个元素传递给 *block*。如果这个区间包含数字，每次加 1 来产生连续元素。否则，*step* 调用 *succ* 方法对这个区间中的元素进行迭代。下面的代码使用定义在本节开始处的 *Xs* 类。

```
range = Xs.new(1)..Xs.new(10)
range.step(2){|x| puts x}
range.step(3){|x| puts x}
```

输出结果：

```
1 x
3 xxx
5 xxxxx
7 xxxxxxx
9 xxxxxxxxx
1 x
4 xxxx
7 xxxxxxx
10 xxxxxxxxx
```

Class **Regexp** < Object

`Regexp` 保存有一个正则表达式，它用来对字符串匹配一个模式。可以使用`/.../`和`%r...`字面量和`Regexp.new`构造函数来创建`Regexp`。本节讲述了 Ruby 1.8 正则表达式。Ruby 1.9 及以后的版本使用不同的正则表达式引擎。

类常量

| | |
|------------|-----------------------|
| EXTENDED | 忽视 regexp 中的空格和回车换行符。 |
| IGNORECASE | 匹配是大小写无关的。 |
| MULTILINE | 回车换行符被当作任意别的字符对待。 |

类方法

| | |
|---------|--|
| compile | <code>Regexp.compile(pattern <, options <, lang >>) → rxp</code> |
|---------|--|

等同于`Regexp.new`。

| | |
|--------|---|
| escape | <code>Regexp.escape(string) → escaped_string</code> |
|--------|---|

对任何可能在正则表达式中有特殊含义的字符进行转义。对任何字符串，`Regexp.new(Regexp.escape(str)) =~ str` 会是 true。

```
Regexp.escape('\\\\[*?{}.]') → '\\\\[\\]\\*\\?\\{\\}\\>.
```

| | |
|------------|--|
| last_match | <code>Regexp.last_match → match</code>
<code>Regexp.last_match(int) → string</code> |
|------------|--|

第一种形式返回一个`MatchData` 对象，它是由最后一个成功的模式匹配所产生的。这等同于读取全局变量`$~`的值。第 537 页描述了`MatchData` 对象。第二种形式返回这个`MatchData` 对象中的第 *n* 个字段。

```
/c(.)t/ =~ 'cat'      → 0
Regexp.last_match     → #<MatchData:0x1ce468>
Regexp.last_match(0)  → "cat"
Regexp.last_match(1)  → "a"
Regexp.last_match(2)  → nil
```

| | |
|-----|---|
| new | <code>Regexp.new(string <, options <, lang >>) → rxp</code>
<code>Regexp.new(regexp) → new_regexp</code> |
|-----|---|

用 *string* 或 *regexp* 构造新的正则表达式。在第二种形式中，*regexp* 的 *options* 会传播，可以不用指定新的 *options*（这是 Ruby 1.8 中的一个变化）。如果 *options* 是一个`Fixnum`，它应当是一个或多个`Regexp::EXTENDED`、`Regexp::IGNORECASE` 和`Regexp::MULTILINE` 的逻辑或。否则，如果 *options* 参数不是`nil`，*regexp* 将会是大

小写无关的。*lang* 参数启用对 regexp 的多字节支持: n, N, 或 nil = none, e, E = EUC, s, S = SJIS, u, U =UTF-8。

| |
|--|
| r1 = Regexp.new('^[a-z]+:\s+\w+') → /^[a-z]+:\s+\w+/
r2 = Regexp.new('cat', true) → /cat/i
r3 = Regexp.new('dog', Regexp::EXTENDED) → /dog/x
r4 = Regexp.new(r2) → /cat/i |
|--|

quote**Regexp.quote(string) → escaped_string**

等同于 `Regexp.escape`。

实例方法**==*****rxp == other_regexp* → true 或 false**

等同性 (equality) ——如果两个 `regexp` 的模式相同, 字符集编码相同以及它们的 `casefold?` 值相同, 那么这两个 `regexp` 是相等的。

| |
|--------------------------|
| /abc/ == /abc/x → false |
| /abc/ == /abc/i → false |
| /abc/u == /abc/n → false |

====***rxp === string* → true 或 false**

Case 等式——等同于在 `case` 语句中使用 `Regexp#=~` 方法。

```
a = "HELLO"
case a
when /^[a-z]*$/; print "Lower case\n"
when /^[A-Z]*$/; print "Upper case\n"
else               print "Mixed case\n"
end
```

输出结果:

```
Upper case
```

=~***rxp =~ string* → int 或 nil**

匹配——对 `string` 匹配 `rxp`, 返回匹配开始处的偏移, 或者如果匹配失败, 返回 `nil`。把 `$~` 设置为相应的 `MatchData` 对象或 `nil`。

| |
|------------------------------|
| /SIT/ =~ "insensitive" → nil |
| /SIT/i =~ "insensitive" → 5 |

-***~ rxp* → int 或 nil**

匹配——对 `$~` 内容匹配 `rxp`。等同于 `rxp =~ $~`。

```
$_ = "input data"
~ /at/ → 7
```

| | |
|------------------|-------------------------------------|
| casefold? | <i>rxp.casefold? → true 或 false</i> |
|------------------|-------------------------------------|

返回大小写相关性标志的值。

| | |
|----------------|-----------------------------|
| inspect | <i>rxp.inspect → string</i> |
|----------------|-----------------------------|

返回一个可读版本的 *rxp*。

```
/cat/mi.inspect → "/cat/mi"
/cat/mi.to_s → "(?mi-x:cat)"
```

| | |
|--------------|---------------------------|
| kcode | <i>rxp.kcode → string</i> |
|--------------|---------------------------|

返回 *regexp* 的字符集编码。

```
/cat/.kcode → nil
/cat/s.kcode → "sjis"
```

| | |
|--------------|--|
| match | <i>rxp.match(string) → match 或 nil</i> |
|--------------|--|

返回描述这次匹配的一个 *MatchData* 对象（参见第 537 页），如果没有匹配上，则返回 *nil*。这相当于在一次正常的匹配后，获得特别变量 \$~ 的值。

```
/(..)(..).match("abc")[2] → "b"
```

| | |
|----------------|--------------------------|
| options | <i>rxp.options → int</i> |
|----------------|--------------------------|

1.8 返回对应创建 *Regexp* 时所用 *options* 的位的集合（详细信息请参见 *Regexp.new*）。请注意：可能会在返回的 *options* 中设置附加的位，正则表达式的代码会在内部使用它们。如果把 *options* 传递给 *Regexp.new*，这些附加的位会被忽视。

```
# Let's see what the values are...
Regexp::IGNORECASE → 1
Regexp::EXTENDED → 2
Regexp::MULTILINE → 4

/cat/.options → 0
/cat/ix.options → 3
Regexp.new('cat', true).options → 1
Regexp.new('cat', 0, 's').options → 48

r = /cat/ix
Regexp.new(r.source, r.options) → /cat/ix
```

| | |
|---------------|----------------------------|
| source | <i>rxp.source → string</i> |
|---------------|----------------------------|

返回初始的字符串模式。

```
/ab+c/ix.source → "ab+c"
```

to_s*rxp.to_s→string*

返回一个字符串，它包含正则表达式和它的 options（使用这个(`?xx:yyy`)表示法）。这个字符串可以作为正则表达式传递给 `Regexp.new`，它与原来的正则表达式有相同的语义（但是，当对它们进行比较时，`Regexp#==`可能不会返回 `true`，因为正则表达式的源（`source`）本身就可能会不相同，如下面的例子所示）。

`Regexp#inspect` 产生一个通常更具可读性版本的 `rxp`。

```
r1 = /ab+c/ix      →  /ab+c/ix
s1 = r1.to_s        →  "(?ix-m:ab+c)"
r2 = Regexp.new(s1) →  /(?ix-m:ab+c)/
r1 == r2           →  false
r1.source          →  "ab+c"
r2.source          →  "(?ix-m:ab+c)"
```

Module Signal

很多操作系统允许发送信号给正运行的进程。一些信号对进程有明确的影响，而别的信号可能会在用户代码中被捕获（trapped），同时作出相应的处理。比如，你的进程可能捕获 USR1 信号，使用它来进行切换调试，或者使用 TERM 信号来发起一次受控的关机。

```
pid = fork do
  Signal trap("USR1") do
    $debug = !$debug
    puts "Debug now: #$debug"
  end
  Signal trap("TERM") do
    puts "Terminating..."
    shutdown()
  end
  # . . . do some work . . .
end
Process.detach(pid)
# Controlling program:
Process.kill("USR1", pid)
# ...
Process.kill("USR1", pid)
# ...
Process.kill("TERM", pid)
```

输出结果：

```
Debug now:true
Debug now:false
Terminating...
```

这个可用信号名以及信号名解释的列表是与系统密切相关的。信号发送机制也可能会因系统而异；尤其是信号发送可能不总是可靠的。

模块方法

list

Signal.list → hash

返回一个信号名列表，这些信号名映射到相应的底层操作系统信号数值。

```
Signal.list → { "ABRT"=>6, "ALRM"=>14, "BUS"=>10, "CHLD"=>20,
  "CLD"=>20, "CONT"=>19, "EMT"=>7, "FPE"=>8, "HUP"=>1,
  "ILL"=>4, "INFO"=>29, "INT"=>2, "IO"=>23, "IOT"=>6,
  "KILL"=>9, "PIPE"=>13, "PROF"=>27, "QUIT"=>3,
  "SEGV"=>11, "STOP"=>17, "SYS"=>12, "TERM"=>15,
  "TRAP"=>5, "TSTP"=>18, "TTIN"=>21, "TTOU"=>22,
  "URG"=>16, "USR1"=>30, "USR2"=>31, "VTALRM"=>26,
  "WINCH"=>28, "XCPU"=>24, "XFSZ"=>25}
```

trap

Signal trap(signal, proc) → obj
Signal trap(signal) { block } → obj

指定信号的处理方式。第一个参数是信号名（例如 SIGALRM、SIGUSR1 等等那样的字符串）或信号数。信号名可能会省略 SIG 字符。这个命令或 block 指定了当这个信号引发时要运行的特定代码。如果这个命令是 IGNORE 或 SIG_IGN 字符串，这个信号会被忽视。如果这个命令是 DEFAULT 或 SIG_DFL，将调用操作系统默认的处理函数。如果命令是 EXIT，进程将会被这个信号终止。否则，将运行给定的命令或者 block。

特殊的信号名 EXIT 或者信号数值 0 将会在程序即将终止之前被触发。

trap 返回给定信号的原有处理函数。

```
Signal trap(0, lambda { puts "Terminating: #{ $$ }" })
Signal trap("CLD") { puts "Child died" }
fork && Process.wait
```

输出结果：

```
Terminating: 27913
Child died
Terminating: 27912
```

Class String < Object

`String` 对象保存并操作一个字节（byte）的序列，它通常表示若干字符（character）。可以使用 `String.new` 或者以字面量来创建 `String` 对象（参见第 320 页）。

因为存在别名问题，使用字符串时，你应当了解那些会修改 `String` 对象内容的方法。通常名称以`!`结尾的方法会修改它们的接收者；相反，那些不带有`!`的方法返回新的 `String`。当然，也存在特例，比如 `String#[] =`。

Mixes in

Comparable:

`<, <=, ==, >=, >, between?`

Enumerable:

`all?, any?, collect, detect, each_with_index, entries, find, find_all, grep, include?, inject, map, max, member?, min, partition, reject, select, sort, sort_by, to_a, zip`

类方法

new

`String.new(val="") → str`

1.8. 返回新的字符串对象，它含有 `val` 的一个拷贝（`val` 应当是一个 `String` 或者实现了 `to_str` 方法）。

```
str1 = "wibble"
str2 = String.new(str1)
str1.object_id      → 946790
str2.object_id      → 946790
str1[1] = "o"
str1                → "wobble"
str2                → "wibble"
```

实例方法

%

`str % arg → string`

格式化（format）——把 `str` 当作格式化规范，然后返回作用于 `arg` 之后的结果。如果这个格式化规范包含多个替换（substitution），那么 `arg` 必须是一个数组，包含要被替换的值。关于格式化字符串的详细信息请参见第 529 页的 `Kernel.printf`。

```
"%05d" % 123           → "00123"
"%-5s:%08x" % [ "ID", self.object_id ] → "ID_____000eecdc"
```

* *str * int → string*

拷贝——返回新的 String，含有这个接收者的 *int* 个拷贝。

```
"Ho! " * 3 → "Ho! Ho! Ho! "
```

+ *str + string → string*

串联 (concatenation) ——把 string 串联到 *str*，返回这个新的 String。

```
"Hello from " + self.to_s → "Hello from main"
```

<< *str << fixnum → str*
str << obj → str

附加 (append) ——把给定的对象串联到 *str*。如果这个对象是一个值在 0 和 255 之间的 Fixnum，在串联之前，会把它转换成一个字符。

```
a = "hello "
a << "world" → "hello world"
a << 33 → "hello world!"
```

<=> *str <=> other_string → -1, 0, +1*

比较——如果 *str* 小于 *other_string*，返回 -1；等于 *other_string*，返回 0；大于 *other_string*，返回 +1。如果这两个字符串有不同的长度，且比较至二者最短长度时相等，那么我们认为长的字符串要大于短的字符串。如果变量 \$= 是 false，比较是根据字符串中每个字符的二进制值来进行的。在旧的 Ruby 版本中，设置 \$= 允许进行大小写无关的比较；现在这种方式已废弃，请使用 String#casecmp。

<=> 是 <、<=、>、>= 和 between? 方法的基础，这些方法是从 Comparable 模块中包括进来的。而 String#== 方法没有使用 Comparable#==。

```
"abcdef" <=> "abcde" → 1
"abcdef" <=> "abcdef" → 0
"abcdef" <=> "abcdefg" → -1
"abcdef" <=> "ABCDEF" → 1
```

== *str == obj → true 或 false*

等同性 (equality) ——如果 *obj* 是一个 String 并且 *str* <=> *obj* 等于 0，则返回 true；否则返回 false。如果 *obj* 不是一个 String，但是它会对 to_s 作出响应，则返回 *obj* == *str*；否则返回 false。

```
"abcdef" == "abcde" → false
"abcdef" == "abcdef" → true
```

==== *str === obj → true 或 false*

Case 等式——等同于 String#==。

 =~

str =~ *regexp* → *int* 或 *nil*

匹配——相当于 *regexp* =~ *str*。Ruby 1.8 之前的版本允许 =~ 有一个任意的操作数；现在已经不推荐使用了。返回匹配开始的位置，或者如果没有匹配上，则返回 nil。

```
"cat o' 9 tails" =~ /\d/ → 7
```

[]

str[*int*] → *int* 或 *nil**str*[*int*, *int*] → *string* 或 *nil**str*[*range*] → *string* 或 *nil**str*[*regexp*] → *string* 或 *nil**str*[*regexp*, *int*] → *string* 或 *nil**str*[*string*] → *string* 或 *nil*

元素引用 (element reference) ——如果传入单个 *int*，返回在这个位置的字符码。如果传入两个 *int*，返回一个子字符串，它是从第一个 *int* 指定的偏移开始，长度由第二个 *int* 参数给定。如果给定一个区间，返回一个子字符串，它包含了在这个区间指定的偏移处的那些字符。在所有这三种情况中，如果偏移是负数，则它从 *str* 的结尾处开始数起。如果初始的偏移落在字符串之外，没有给定长度，长度是负数或者区间的开头大于结尾，返回 nil。

如果提供了 *regexp*，则返回 *str* 匹配的那部分。如果正则表达式后面跟有一个数字参数，则相反，它返回 MatchData 对象的那个成分。如果给定一个 String 并且它出现在 *str* 中，则返回这个字符串。在这两种情况下，如果没有匹配上，都返回 nil。

| | |
|----------------------|---------|
| a = "hello there" | |
| a[1] | → 101 |
| a[1,3] | → "ell" |
| a[1..3] | → "ell" |
| a[1...3] | → "el" |
| a[-3,2] | → "er" |
| a[-4...-2] | → "her" |
| a[-2...-4] | → "" |
| a[/[aeiou](.)\1/] | → "ell" |
| a[/[aeiou](.)\1/, 0] | → "ell" |
| a[/[aeiou](.)\1/, 1] | → "l" |
| a[/[aeiou](.)\1/, 2] | → nil |
| a[/(..)e/] | → "the" |
| a[/(..)e/, 1] | → "th" |
| a["lo"] | → "lo" |
| a["bye"] | → nil |

[]=

`str[int] = int`
`str[int] = string`
`str[int, int] = string`
`str[range] = string`
`str[regexp] = string`
`str[regexp, int] = string`
`str[string] = string`

1.8.

元素赋值 (element assignment) —— 替换 `str` 中的部分或者全部内容。使用与 `String#[]` 同样的标准来决定要被替换的那部分字符串。如果替换字符串的长度与要替换的文本长度不相同，替换字符串会相应地做出调整。如果用作索引的正则表达式或者字符串没有匹配上字符串中的任何位置，引发 `IndexError`。如果使用正则表达式形式，可选的第二个 `int` 参数允许你指定要替换的匹配部分（实际上使用 `MatchData` 的索引规则）。如果 `Fixnum` 的值超出范围，接受一个 `Fixnum` 的形式会引发 `IndexError`。这个区间形式会引发 `RangeError`，而 `Regexp` 和 `String` 形式会默默地忽视这个赋值。

```
a = "hello"
a[2] = 96
a[2, 4] = "xyz"
a[-4, 2] = "xyz"
a[2..4] = "xyz"
a[-4..-2] = "xyz"
a[/[aeiou](.)\1(.)/] = "xyz"
a[/[aeiou](.)\1(.)/, 1] = "xyz"
a[/[aeiou](.)\1(.)/, 2] = "xyz"
a["l"] = "xyz"
a["ll"] = "xyz"
a[2, 0] = "xyz"

(a → "he`lo")
(a → "hexyz")
(a → "hxyzlo")
(a → "hexyz")
(a → "hxyz")
(a → "hxyz")
(a → "hexyzlo")
(a → "hellxyz")
(a → "hexyzlo")
(a → "hexyzo")
(a → "hexyzllo")
```

~ `str → int 或 nil`

等同于 `$_. =~ str`。

`capitalize`

`str.capitalize → string`

返回 `str` 的一个拷贝，把它的首字符转换成大写，同时把其余字符都转换成小写。

```
"hello".capitalize → "Hello"
"HELLO".capitalize → "Hello"
"123ABC".capitalize → "123abc"
```

`capitalize!`

`str.capitalize! → str 或 nil`

通过把首字符转换成大写，同时把其余字符都转换成小写来修改 `str`。如果没有做出任何修改，则返回 `nil`。

```
a = "hello"
a.capitalize!      →  "Hello"
a                  →  "Hello"
a.capitalize!      →  nil
```

casecmp*str.casecmp(string)→ -1, 0, +1*

1.8 大小写无关的 String#<=>版本。

```
"abcdef".casecmp("abcde")      →  1
"abcdef".casecmp("abcdef")     →  0
"ABcDeF".casecmp("abcdef")     →  0
"abcdef".casecmp("abcdefg")    →  -1
"abcdef".casecmp("ABCDEF")     →  0
```

center*str.center(int, pad=" ")→ string*

如果 *int* 大于 *str* 的长度，返回新的 String，*int* 是它的长度，把 *str* 放置在中间，左右两边用给定的 padding（默认是空格）来填充；否则返回 *str*。

```
"hello".center(4)            →  "hello"
"hello".center(20)           →  "         hello         "
"hello".center(4, "-^-")     →  "hello"
"hello".center(20, "-^-")    →  "-^-_-^-hello_-^-_-^-"
"hello".center(20, "-")      →  "-----hello-----"
```

chomp*str.chomp(rs=\$/)→ string*

在 *str* 的结尾处删除给定的记录分隔符（如果有的话），返回新的 String。如果未将\$/从默认的 Ruby 记录分隔符更改为其他字符，那么 chomp 也会删除回车换行符（也就是说会删除\n、\r 和\r\n）。

```
"hello".chomp          →  "hello"
"hello\n".chomp         →  "hello"
"hello\r\n".chomp        →  "hello"
"hello\n\r".chomp        →  "hello\n"
"hello\r".chomp          →  "hello"
"hello \n there".chomp   →  "hello \n there"
"hello".chomp("llo")     →  "he"
```

chomp!*str.chomp!(rs=\$/)→ str 或 nil*

以 String#chomp 所描述的方式直接对 *str* 进行修改，同时返回它，或者如果没有做出任何修改，则返回 nil。

chop*str.chop → string*

返回新的 String，删除它的最后一个字符。如果这个字符串以\r\n结尾，则两个字符都会被删除。把 chop 应用于空串，则会返回空串。通常 String#chomp 是一个更安全的选择，因为如果字符串不是以一个记录分隔符结束，它并不会修改这个字符串。

```
"string\r\n".chop      →      "string"
"string\n\r".chop      →      "string\n"
"string\n".chop       →      "string"
"string".chop         →      "strin"
"x".chop.chop        →      "
```

chop!*str.chop! → str 或 nil*

就像 `String#chop` 那样对 `str` 进行处理，返回它；或者如果 `str` 是空串，返回 `nil`。同时参见 `String#chomp!`

concat

str.concat(int) → str
str.concat(obj) → str

等同于 `String#<<`。

count*str.count(< string >+) → int*

每个 `string` 参数都定义了要被计数的字符集合。这些集合的交集定义了 `str` 中要被计数的所有字符。任何以`^`符号开始的参数会被取反。`c1–c2` 序列指的是在 `c1` 和 `c2` 之间的所有字符。

```
a = "hello world"
a.count "lo"           → 5
a.count "lo", "o"      → 2
a.count "hello", "^l"  → 4
a.count "ej-m"         → 4
```

crypt*str.crypt(settings) → string*

通过调用标准库函数 `crypt` 来把单向的加密散列算法应用于 `str`。这个参数某种程度上是与系统相关的。在传统的 Unix 机器上，它常常是两个字符的 `salt` 字符串。在更现代的机器上，它也可能控制例如 DES 加密参数等一些事情。关于详细的信息，请参见 `crypt(3)` 手册页。

```
# standard salt
"secret".crypt("sh")          → "shRK3aVg8FsI2"
# On OSX: DES, 2 interactions, 24-bit salt
"secret".crypt("...0abcd")     → "...0abcdROn65JNDj12"
```

delete*str.delete(< string >+) → new_string*

返回 `str` 的一个拷贝，其中在参数交集中出现的所有字符均被删除。使用了与 `String#count` 相同的规则来定义字符集。

```
"hello".delete("l","o")       → "heo"
"hello".delete("lo")         → "he"
"hello".delete("aeiou", "^e") → "hell"
"hello".delete("ej-m")       → "ho"
```

delete!*str.delete!(<string>+) → str 或 nil*

直接对 *str* 执行删除操作，同时返回它；如果 *str* 没有任何修改，则返回 *nil*。

```
a = "hello"
a.delete!("l","lo")      →  "heo"
a                      →  "heo"
a.delete!("l")          →  nil
```

downcase*str.downcase → string*

返回 *str* 的一个拷贝，把它的所有大写字母都替换成小写字母。这个操作是与语言环境（locale）无关的——只会影响到从 A 到 Z 的所有字符。会跳过多字节（multibyte）字符。

```
"hEllO".downcase →  "hello"
```

downcase!*str.downcase! → str 或 nil*

把 *str* 中的大写字母都替换成小写字母，如果 *str* 没有做出任何改变，则返回 *nil*。

dump*str.dump → string*

产生 *str* 用 \nnn 表示法替换其所有非打印字符、同时转义所有特殊字符的版本。

each*str.each(sep=\$/) { |substr| block } → str*

把提供的参数当作记录分隔符（默认是 \$/）来分割 *str*，依次把每个子字符串传递给这个给定的 *block*。如果提供的记录分隔符长度为 0，会把这个字符串分割成段落，其中每个段落由多个 \n 字符结束。

```
print "Example one\n"
"hello\nworld".each{|s| p s}
print "Example two\n"
"hello\nworld".each('l') {|s| p s}
print "Example three\n"
"hello\n\n\nworld".each('') {|s| p s}
```

输出结果：

```
Example one
"hello\n"
"world"
Example two
"hel"
"l"
"o\nworl"
"d"
Example three
"hello\n\n\n"
"world"
```

| | |
|------------------|--|
| each_byte | <i>str.each_byte { int block } → str</i> |
|------------------|--|

把 *str* 中的每个字节传入给定的 *block*。

```
"hello".each_byte { |c| print c, ' ' }
```

输出结果：

```
104 101 108 108 111
```

| | |
|------------------|--|
| each_line | <i>str.each_line(sep=\$/) { substr block } → str</i> |
|------------------|--|

等同于 *String#each*。

| | |
|---------------|----------------------------------|
| empty? | <i>str.empty? → true 或 false</i> |
|---------------|----------------------------------|

如果 *str* 长度为 0，返回 *true*。

```
"hello".empty? → false
"".empty? → true
```

| | |
|-------------|---------------------------------------|
| eql? | <i>str.eql?(obj) → true 或 false</i> |
|-------------|---------------------------------------|

如果 *obj* 字符串的内容与 *str* 的内容相同，返回 *true*。

```
"cat".eql?("cat") → true
```

| | |
|-------------|---|
| gsub | <i>str.gsub(pattern, replacement) → string</i> |
| | <i>str.gsub(pattern) { match block } → string</i> |

返回 *str* 的一个拷贝，其中所有 *pattern* 的出现处都用 *replacement* 或者 *block* 的值来替换。通常 *pattern* 是一个 *Regexp*；如果它是一个 *String*，那么正则表达式元字符将不会被解释（也就是说 /\d/ 将匹配一个数字，但是 '\d' 将匹配一条反斜线，后面跟一个 d）。

如果一个字符串被用作 *replacement*，由匹配得到的那些特殊变量（例如 \$& 和 \$1）将不能被替换到这个字符串中，因为在这个字符串中进行的替换发生在模式匹配开始之前。当然，可以使用 \1、\2 等等序列，在匹配中插入连续的组（group）。下页的表 27.13 显示了这些序列。

带有 *block* 的形式，当前的匹配会作为参数传入 *block*，并且会适当地设置 \$1、\$2、\$`、\$& 和 \$' 等等变量。每次调用传入的匹配都将被 *block* 的返回值所替换。

返回的结果继承了在初始字符串或者任何给定的 *replacement* 字符串中的 tainting 状态。

```
"hello".gsub(/ [aeiou] /, '*') → "h*ll*"
"hello".gsub(/ ([aeiou]) /, '<\1>') → "h<e>ll<o>"
"hello".gsub(/ ./) { |s| s[0].to_s + ' ' } → "104 101 108 108 111 "
```

表 27.13 替换字符串中的反斜线序列

| 序 列 | 被替换的文本 |
|-----------------|---------------------------------------|
| \1, \2, ..., \9 | 被第 <i>n</i> 组子表达式匹配的值 |
| \& | 最后的匹配 |
| \` | 匹配之前的部分字符串 |
| \' | 匹配之后的部分字符串 |
| \+ | 匹配最大编号的组 (The highest-numbered group) |

| | |
|-------|---|
| gsub! | <i>str.gsub!(pattern, replacement) → str 或 nil</i>
<i>str.gsub!(pattern) { match block } → str 或 nil</i> |
|-------|---|

直接执行 `String#gsub` 替换，返回 `str`；如果没有发生任何替换，返回 `nil`。

| | |
|---|----------------------|
| hex | <i>str.hex → int</i> |
| 把 <code>str</code> 的那些前导字符看作一个十六进制数字（带有可选的符号标记和可选的 <code>0x</code> ）的字符串，返回相应的数值。如果出错，返回 <code>0</code> 。 | |
| "0x0a".hex | → 10 |
| "-1234".hex | → -4660 |
| "0".hex | → 0 |
| "wombat".hex | → 0 |

| | |
|----------|---|
| include? | <i>str.include?(string) → true 或 false</i>
<i>str.include?(int) → true 或 false</i> |
|----------|---|

如果 `str` 包含了给定的字符串或者字符，返回 `true`。

```
"hello".include? "lo"    → true
"hello".include? "ol"    → false
"hello".include? ?h     → true
```

| | |
|-------|---|
| index | <i>str.index(string <, offset >) → int 或 nil</i>
<i>str.index(int <, offset >) → int 或 nil</i>
<i>str.index(regexp <, offset >) → int 或 nil</i> |
|-------|---|

返回给定的子字符串，字符或者模式首次出现在 `str` 中的索引。如果未发现，返回 `nil`。如果提供了第二个参数，它指定了在 `string` 内开始搜寻的位置。

```
"hello".index('e')          → 1
"hello".index('lo')         → 3
"hello".index('a')          → nil
"hello".index(101)          → 1
"hello".index(/[aeiou]/, -3) → 4
```

insert*str.insert(index, string)→ str*

1.8.

在给定 *index* 位置的字符前面把 *string* 插入，同时修改 *str*。如果 *index* 是一个负数，就从 *string* 的结尾处开始数起，并在给定的字符后面把 *string* 插入。插入完成后，*str* 包含了从 *index* 位置开始的 *string*。

```
"abcd".insert(0, 'X')      → "Xabcd"
"abcd".insert(3, 'X')      → "abcXd"
"abcd".insert(4, 'X')      → "abcdX"
"abcd".insert(-3, 'X')     → "abXcd"
"abcd".insert(-1, 'X')     → "abcdX"
```

intern*str.intern → symbol*

1.8.

返回对应于 *str* 的 *symbol*，如果原先不存在这个符号，就创建它。可以 *intern* 任何字符串，而不仅仅是标识符。参见第 631 页的 #id2name 符号。

```
"Koala".intern    → :Koala
sym = "$1.50 for a soda!?!?".intern
sym.to_s         → "$1.50 for a soda!?!?"
```

length*str.length → int*

返回 *str* 的长度。

ljust*str.ljust(width, padding=" ")→ string*

1.8.

如果 *width* 大于 *str* 的长度，返回新的 *String*，长度为 *width*，其中 *str* 左对齐，用 *padding* 填充剩下的空间；否则，返回 *str* 的一个拷贝。

```
"hello".ljust(4)          → "hello"
"hello".ljust(20)         → "hellooooooooooooo"
"hello".ljust(20, "*")    → "hello*****"
"hello".ljust(20, " dolly") → "hello dolly dolly do"
```

lstrip*str.lstrip → string*

1.8.

返回 *str* 的一个拷贝，删除其中的前导空格字符。同时参见 String#rstrip 和 String#strip 方法。

```
" hello ".lstrip        → "hello"
"\000 hello ".lstrip    → "\000hello"
"hello".lstrip           → "hello"
```

lstrip!*str.lstrip! →**self 或 nil*

1.8.

删除 *str* 中的前导空格字符，如果没有做出任何修改，返回 *nil*。同时参见 String#rstrip! 和 String#strip! 方法。

```
" hello ".lstrip!       → "hello"
"hello".lstrip!          → nil
```

match *str.match(pattern) → match_data 或 nil*

1.8. (如果 *pattern* 还不是一个 `Regexp`) 把它转换成一个 `Regexp`, 同时对 *str* 调用这个 `Regexp` 的 `match` 方法。

```
'hello'.match('(.)\1')      → #<MatchData:0x1c9468>
'hello'.match('(.)\1')[0]    → "ll"
'hello'.match(/(.)\1/)[0]    → "ll"
'hello'.match('xx')         → nil
```

next *str.next → string*

等同于 `String#succ`。

next! *str.next! → str*

等同于 `String#succ!`。

oct *str.oct → int*

把 *str* 开头的若干字符看作一个八进制数字（带有可选的符号标记）的字符串，同时返回相应的数字。如果转换失败，返回 0。

```
"123".oct      → 83
"-377".oct    → -255
"bad".oct      → 0
"0377bad".oct → 255
```

replace *str.replace(string) → str*

用 *string* 的相应值替换 *str* 的内容和非受信度（taintedness）。

```
s = "hello"      → "hello"
s.replace "world" → "world"
```

reverse *str.reverse → string*

返回新的字符串，反转 *str* 的字符。

```
# 每个问题有它自己的解决办法...
"stressed".reverse → "desserts"
```

reverse! *str.reverse! → str*

直接反转 *str*。

rindex *str.rindex(string <, int >) → int 或 nil*

str.rindex(int <, int >) → int 或 nil

str.rindex(regexp <, int >) → int 或 nil

返回给定的子字符串，字符或模式在 *str* 中最后出现的索引。如果未发现，返回 `nil`。如果给出了第二个参数，它指定了在这个字符串中搜寻的结束位置——超过这个位置的字符将不会被考虑。

```
"hello".rindex('e')          → 1
"hello".rindex('l')          → 3
"hello".rindex('a')          → nil
"hello".rindex(101)          → 1
"hello".rindex(/[aeiou]/, -2) → 1
```

rjust *str.rjust(width, padding=" ")→ string*

1.8. 如果 *width* 大于 *str* 的长度，返回新的 String，长度为 *width*，其中 *str* 右对齐，并且用 *padding* 填充剩余的空间；否则，返回 *str* 的一个拷贝。

```
"hello".rjust(4)          → "hello"
"hello".rjust(20)          → "          hello"
"hello".rjust(20, "-")    → "-----hello"
"hello".rjust(20, "padding") → "paddingpaddingphello"
```

rstrip *str.rstrip→ string*

1.8. 返回 *str* 的一个拷贝，删除第一个结尾的 NULL 字符，然后删除结尾的空格字符。同时参见 String#lstrip 和 String#strip。

```
" hello ".rstrip      → "hello"
" hello \000 ".rstrip → "hello\000"
" hello \000".rstrip  → "hello"
"hello".rstrip        → "hello"
```

rstrip! *str.rstrip!→ self 或 nil*

1.8. 删除 *str* 的结尾 NULL 字符，然后删除结尾的空格字符。如果没有做出任何修改，返回 nil。同时参见 String#lstrip! 和 #strip!。

```
" hello ".rstrip! → "hello"
"hello".rstrip! → nil
```

scan *str.scan(pattern)→array*
str.scan(pattern){|match,...| block }→str

这两种形式都通过匹配 *pattern*（它可能是一个 Regexp 或者一个 String），来迭代 *str*。每次匹配会生成一个结果，然后把结果添加到这个结果数组中，或者传递给 *block*。如果 *pattern* 没有包含任何组，每个结果由匹配的字符串 (\$&) 组成。如果 *pattern* 包含了组，每个结果本身就是一个数组，其中每组是一项。如果 *pattern* 是一个 String，就从字面意义上解释它（即不会把它当作一个正则表达式）。

```
a = "cruel world"
a.scan(/\w+/)          → ["cruel", "world"] .
a.scan(/.../)          → ["cru", "el ", "wor"]
a.scan(/(...)/)        → [["cru"], ["el "], ["wor"]]
a.scan(/(..)(..)/)    → [["cr", "ue"], ["l ", "wo"]]
```

下面是 *block* 的形式：

```
a.scan(/\w+/) { |w| print "<<#{w}>> " }
puts
a.scan(/(.)(.)/) { |a,b| print b, a }
puts
```

输出结果：

```
<<cruel>> <<world>>
rceu lowlr
```

| | |
|-------------|-----------------------|
| size | <i>str.size → int</i> |
|-------------|-----------------------|

等同于 `String#length`。

| | |
|--------------|-------------------------------------|
| slice | <i>str.slice(int) → int 或 nil</i> |
|--------------|-------------------------------------|

str.slice(int, int) → string 或 nil

str.slice(range) → string 或 nil

str.slice(regexp) → string 或 nil

str.slice(match_string) → string 或 nil

等同于 `String#[]`。

```
a = "hello there"
a.slice(1)           → 101
a.slice(1,3)         → "ell"
a.slice(1..3)        → "ell"
a.slice(-3,2)        → "er"
a.slice(-4..-2)      → "her"
a.slice(-2..-4)      → ""
a.slice(/th[aeiou]/) → "the"
a.slice("lo")         → "lo"
a.slice("bye")        → nil
```

| | |
|---------------|--------------------------------------|
| slice! | <i>str.slice!(int) → int 或 nil</i> |
|---------------|--------------------------------------|

str.slice!(int, int) → string 或 nil

str.slice!(range) → string 或 nil

str.slice!(regexp) → string 或 nil

str.slice!(match_string) → string 或 nil

从 `str` 删除指定的部分子字符串，同时返回被删除部分。如果指定的值超出了字符串范围，接受一个 `Fixnum` 的形式会引发 `IndexError`；而接受一个 `Range` 的形式会引发 `RangeError`，接受 `Regexp` 和 `String` 的形式则默然不会修改字符串。

```
string = "this is a string"
string.slice!(2)           → 105
string.slice!(3..6)         → " is "
string.slice!(/s.*t/)       → "sa st"
string.slice!("r")          → "r"
string                      → "thing"
```

split

str.split(pattern=\$;, <limit>) → array

基于分隔符 (delimiter) 把 *str* 分隔成若干子字符串，组成一个数组并返回。

如果 *pattern* 是一个 String，那么它的内容被当作这个分隔符来对 *str* 进行分隔。如果 *pattern* 是单个空格字符，*str* 会在空格处分开，忽视前导空格和后面连续的空格字符。

如果 *pattern* 是一个 Regexp，*str* 会在匹配这个模式的位置分开。无论何时当这个模式匹配了一个长度为 0 的字符串，*str* 会被分隔成单个字符。如果 *pattern* 包含了组 (groups)，那些组将会包含在返回的值中。

如果省略了 *pattern* 参数，它会使用变量 \$; 的值。如果 \$; 是 nil (这是默认值)，那么 *str* 会在空格处分开，就如同指定 “\s” 为分隔符。

如果省略了 *limit* 参数，它会抑制 (suppress) 结尾的空字段。如果 *limit* 是正数，最多可能返回那个数目的字段数 (如果 *limit* 是 1，整个字符串会作为数组的唯一元素被返回)。如果 *limit* 是负数，对于要返回的字段数不会有任何限制，同时也不会抑制结尾的空字段。

```
" now's the time".split      → ["now's", "the", "time"]
" now's the time".split(' ') → ["now's", "the", "time"]
" now's the time".split(/ /) → [ "", "now's", "", "", "the", "time"]
"a@1bb@2ccc".split(/@\d/)   → ["a", "bb", "ccc"]
"a@1bb@2ccc".split(/@(\d)/) → ["a", "1", "bb", "2", "ccc"]
"1, 2.34, 56, 7".split(/,\s*/) → ["1", "2.34", "56", "7"]
"hello".split("//")          → ["h", "e", "l", "l", "o"]
"hello".split("//, 3)        → ["h", "e", "llo"]
"hi mom".split(/\s*/)       → ["h", "i", "m", "o", "m"]
"".split                      → []
"mellow yellow".split("ello") → ["m", "w y", "w"]
"1,2,,3,4,,".split(',')     → ["1", "2", "", "3", "4"]
"1,2,,3,4,,".split(',', 4)  → ["1", "2", "", "3,4,,"]
"1,2,,3,4,,".split(',', -4) → ["1", "2", "", "3", "4", "", ""]
```

squeeze

str.squeeze(<string>) → squeezed_string

使用第 611 页描述的 String#count 步骤，用 *string* 参数构造一个字符集合。返回新的字符串，出现在这个字符集合中的相同连续字符会被单个字符替代。如果没有给出任何参数，字符串中所有相同的连续字符都会被单个字符替代。

```
"yellow moon".squeeze           → "yellow mon"
" now is the".squeeze(" ")     → " now is the"
"putters putt balls".squeeze("m-z") → "putters putt balls"
```

squeeze!*str.squeeze!(<string>*) → str 或 nil*

直接对 *str* 进行挤压 (squeeze)，同时返回 *str*。如果没有做出任何修改，则返回 nil。

strip*str.strip → string*

1.8

返回 *str* 的一个拷贝，删除它开头的空格，结尾的 NULL 和空格字符。

```
"hello ".strip → "hello"
"\tgoodbye\r\n".strip → "goodbye"
"goodbye \000".strip → "goodbye"
"goodbye \000 ".strip → "goodbye \000"
```

strip!

str.strip! → str 或 nil

1.8

删除 *str* 开头的空格，结尾的 NULL 和空格字符。如果 *str* 没有改变，返回 nil。

sub*str.sub(pattern, replacement) → string**str.sub(pattern) {|match| block} → string*

返回 *str* 的一个拷贝，用 *replacement* 或者 *block* 的值替换出现在 *str* 中的第一个 *pattern*。*pattern* 通常是一个 Regexp；如果它是一个 String，那么将不会解释任何正则表达式元字符（也就是说 /\d/ 会匹配一个数字，但是 '\d' 会匹配一条反斜线，后面跟有一个 d 字符）。

如果这个方法调用指定了 *replacement*，那些特殊变量比如 \$& 将变得没有用处。因为在这个字符串中的替换发生在模式匹配开始之前。但是，可以使用在第 614 页表 27.13 中列举的 \1、\2 等等序列。

在 *block* 形式中，当前的匹配作为参数传递给 *block*，并且适当地设置 \$1、\$2、\$`、\$& 和 \$' 等等变量。每次调用传入的匹配都将被 *block* 的返回值所替换。

```
"hello".sub(/aeiou/, '*') → "h*llo"
"hello".sub(/([aeiou])/, '<\1>') → "h<e>llو"
"hello".sub(/./) {|s| s[0].to_s + ' '} → "104 ello"
```

sub!

*str.sub!(pattern, replacement) → str 或 nil**str.sub!(pattern) {|match| block} → str 或 nil*

直接对 *str* 执行 String#sub 替换，同时返回它。如果没有做出任何替换，则返回 nil。

succ*str.succ → string*

返回 *str* 的 successor。通过增加在这个字符串中最右边的字母数字（如果不是字母数字，或者最右边的字符）来得到这个 successor。增加一个数字总是会导致产生另外一个数字，同时增加一个字母总是会导致产生大小写相同的另外一个字母。使用底层的字符集的排序序列来增加非字母数字字符。

如果这个增加产生一次“进位 (carry)”，它会继续增加其左边的字符。这个过程会一直重复，直到没有进位产生为止，如果必要的话，会添加一个附加的字符。

```
"abcd".succ      → "abce"
"THX1138".succ  → "THX1139"
"<<koala>>".succ → "<<koalb>>"
"1999zzz".succ  → "2000aaa"
"ZZZ9999".succ  → "AAAA0000"
"****".succ     → "***+"
```

succ! *str.succ! → str*

等同于 `String#succ`，但直接修改这个接收者。

sum *str.sum(n=16) → int*

返回 *str* 中字符一个基本的 *n* 位校验和，这里 *n* 是可选的参数，默认是 16。对 1.8 *str* 中的每个字符的二进制值简单求和，然后用 $2^n - 1$ 求模，得到这个校验和。这不是一个特别好的校验和——要使用更好的校验和，请参见第 668 页的 `digest` 库。

```
"now is the time".sum      → 1408
"now is the time".sum(8)   → 128
```

swapcase *str.swapcase → string*

返回 *str* 的一个拷贝，把其中的大写字母都转换成小写，同时把小写字母都转换成大写。

```
"Hello".swapcase        → "hELLO"
"cYbEr_PuNk11".swapcase → "CyBeR_pUnK11"
```

swapcase! *str.swapcase! → str 或 nil*

等同于 `String#swapcase`，但直接对 *str* 进行修改，同时返回它。如果没有做出任何修改，则返回 `nil`。

to_f *str.to_f → float*

返回对 *str* 中开头若干字符进行解释后得到的浮点数结果。忽视有效数字部分之后的那些无关字符。如果 *str* 的起始字符不是一个有效的数字，则返回 `0.0`。这个方法从来不会引发异常（使用 `Kernel.Float` 验证数字）。

```
"123.45e1".to_f      → 1234.5
"45.67 degrees".to_f → 45.67
"thx1138".to_f       → 0.0
```

to_i *str.to_i(base=10)→int*

1.8 对 *str* 中开头的若干字符进行解释，返回以 *base* 为进制（2 到 36）的整数。如果给定的进制为 0，*to_i* 将会寻找 0、0b、0o、0d 或者 0x 前缀，并且相应地设置进制。开头的空格会被忽视，同时接受开头的加号或者减号。忽视有效数字部分之后的那些无关字符。如果 *str* 的首字符不是一个有效数字，则返回 0。这个方法从来不会引发异常。

| | | |
|------------------------|---|-----------|
| "12345".to_i | → | 12345 |
| "99 red balloons".to_i | → | 99 |
| "0a".to_i | → | 0 |
| "0a".to_i(16) | → | 10 |
| "0x10".to_i | → | 0 |
| "0x10".to_i(0) | → | 16 |
| "-0x10".to_i(0) | → | -16 |
| "hello".to_i | → | 0 |
| "hello".to_i(30) | → | 14167554 |
| "1100101".to_i(2) | → | 101 |
| "1100101".to_i(8) | → | 294977 |
| "1100101".to_i(10) | → | 1100101 |
| "1100101".to_i(16) | → | 17826049 |
| "1100101".to_i(24) | → | 199066177 |

to_s *str.to_s→str*

返回这个接收者。

to_str *str.to_str→str*

等同于 *String#to_s*。*String#concat* 方法会使用 *to_str* 把它们的参数都转换成字符串。与 *to_s* 不同，几乎所有的类都支持 *to_s*，而 *to_str* 方法一般只是实现那些类似字符串的类中。在内建的类中，只有 *Exception* 和 *String* 实现了 *to_str*。

to_sym *str.to_s→symbol*

返回 *str* 的符号。这可以创建那些不能使用:*xxx* 表示法来表示的符号。等同于 *String#intern*。

| | | |
|----------------------|---|----------------|
| s = 'cat'.to_sym | → | :cat |
| s == :cat | → | true |
| 'cat and dog'.to_sym | → | :"cat and dog" |
| s == :'cat and dog' | → | false |

tr *str.tr(from_string, to_string)→string*

返回 *str* 的一个拷贝，用 *to_string* 的相应字符替换 *from_string* 的字符。如果 *to_string* 的长度比 *from_string* 短，就会用 *to_string* 的最后一个字符来填充。这两个字符串可能都使用 *c₁–c₂* 表示法来表示字符的区间，同时 *from_string* 可能以一个^字符开始，表示除了这些给定字符之外的所有字符。

```
"hello".tr('aeiou', '*')      → "h*ll*"
"hello".tr('^aeiou', '*')    → "*e**o"
"hello".tr('el', 'ip')       → "hippo"
"hello".tr('a-y', 'b-z')     → "ifmmp"
```

tr!*str.tr!(from_string, to_string) → str 或 nil*

使用与 `String#tr` 同样的规则直接对 `str` 进行修改，同时返回它。如果没有做出任何修改，则返回 `nil`。

tr_s*str.tr_s(from_string, to_string) → string*

就像 `String#tr` 描述的那样处理 `str` 的一个拷贝，然后在替换发生的区域删除重复字符。

```
"hello".tr_s('l', 'r')      → "hero"
"hello".tr_s('el', '*')    → "h*o"
"hello".tr_s('el', 'hx')   → "hhxo"
```

tr_s!*str.tr_s!(from_string, to_string) → str 或 nil*

直接对 `str` 进行 `String#tr_s` 处理，同时返回它。如果没有做出任何修改，则返回 `nil`。

unpack*str.unpack(format) → array*

基于 `format` 参数对 `str`（可能包含二进制数据）进行解码（decode），返回一个含有析取值的数组。这个格式化字符串由单个字符指令（directive）序列组成，下页的表 27.14 总结了所有这些单字符指令。每个指令后面可能跟有一个数字，它指出重复执行这个指令的次数。星号 (*) 将处理所有剩余的元素。`sSiIIL` 指令后面可能跟有一个下画线（_），以使用这种指定类型在底层平台上的本机大小（native size）。否则，它们会使用一个与平台无关的一致大小。在这个格式化字符串中，空格会被忽视。注释以#开始，直到遇到下一个回车换行符或者字符串结束为止，注释也会被忽略。同时请参见第 436 页的 `Array#pack`。

```
"abc \0\0abc \0\0".unpack('A6Z6') → ["abc", "abc "]
"abc \0\0".unpack('a3a3')      → ["abc", " \000\000"]
"aa".unpack('b8B8')           → ["10000110", "01100001"]
"aaa".unpack('h2H2c')         → ["16", "61", 97]
"\xfe\xff\xfe\xff".unpack('ss') → [-257, 65279]
"now=20is".unpack('M*')       → ["now is"]
"whole".unpack('xax2aX2aX1aX2a') → ["h", "e", "l", "l", "o"]
```

upcase*str.upcase → string*

返回 `str` 的一个拷贝，把其中所有小写字母都用相应的大写字母代替。这个操作是与语言环境（locale）无关的——只有 *a* 到 *z* 之间的 ASCII 字符会受影响。

```
"hEllo".upcase → "HELLO"
```

表 27.14 String#unpack 指令

| 格 式 功 能 | 返 回 值 |
|--|---------|
| A 一个以 NULL 结尾并且删除了空格的字符串 | String |
| a 字符串 | String |
| B 提取每个字符的位 (MSB 在前) | String |
| b 提取每个字符的位 (LSB 在前) | String |
| C 将一个字符作为无符号整型提取出来 | Fixnum |
| c 将一个字符作为整型提取出来 | Fixnum |
| d,D 将 <code>sizeof(double)</code> 个字符视为本机的双精度浮点数 | Float |
| E 将 <code>sizeof(double)</code> 个字符视为小端 (little-endian) 字节序的双精度浮点数 | Float |
| e 将 <code>sizeof(float)</code> 个字符视为小端字节序的双精度浮点数 | Float |
| f,F 将 <code>sizeof(float)</code> 个字符视为本机的浮点数 | Float |
| G 将 <code>sizeof(double)</code> 个字符视为网络字节序的双精度浮点数 | Float |
| g 将 <code>sizeof(float)</code> 个字符视为网络字节序的浮点数 | Float |
| H 提取每个字符的 4 位十六进制值 (最高有效位在前) | String |
| h 提取每个字符的 4 位十六进制值 (最低有效位在前) | String |
| I 将 <code>sizeof(int)</code> ¹ 的连续字符视为一个无符号的本机整型 | Integer |
| i 将 <code>sizeof(int)</code> ¹ 的连续字符视为一个有符号的本机整型 | Integer |
| L 将四个连续字符视为一个无符号的本机整型 | Integer |
| l 将四个连续字符视为一个有符号的本机整型 | Integer |
| M 提取一个引用可打印 (quoted-printable) 的字符串 | String |
| m 提取一个 Base64 编码的字符串 | String |
| N 将四个字符视为一个网络字节序的无符号长整型 | Fixnum |
| n 将两个字符视为一个网络字节序的无符号短整型 | Fixnum |
| P 将 <code>sizeof(char *)</code> 个字符视为一个指针，并从指向的位置返回 <code>len</code> 个字符 | String |
| p 将 <code>sizeof(char *)</code> 个字符视为一个指针，指向一个 null 结尾的字符串 | String |
| Q 将 8 个字符视为一个无符号的四个字 (quad word, 64 位) | Integer |
| q 将 8 个字符视为一个有符号的四个字 (quad word, 64 位) | Integer |
| S 将两个 ⁹ 连续的字符视为一个本机字节序的无符号短整型 | Fixnum |
| s 将两个连续的字符视为一个本机字节序的有符号短整型 | Fixnum |
| U 将一个 UTF-8 字符提取为无符号整型 | Integer |
| u 提取一个 uu 编码的字符串 | String |
| V 将四个字符视为一个小端字节序的无符号长整型 | Fixnum |
| v 将两个字符视为一个小端字节序的无符号短整型 | Fixnum |
| w BER 压缩的整型 (更多信息请参见 <code>Array#pack</code>) | Integer |
| X 向后跳过一个字符 | — |
| x 向前跳过一个字符 | — |
| Z 删除了 NULL 结尾的字符串 | String |
| @ 跳过由长度参数指定的偏移 | — |

⁹ 可能因为向指令附加 “_” 而改变。

upcase!

str.upcase! → str 或 nil

把 *str* 的内容变成大写，如果没有做出任何修改，则返回 *nil*。

upto

str.upto(string) { |s| block } → str

在从 *str* 到参数 *string*（包括 *string*）的连续值上迭代，并且依次把每个值传递给 *block*。使用 *String#succ* 方法来产生这些值。

```
"a8".upto("b6") { |s| print s, ' ' }
for s in "a8".."b6"
  print s, ' '
end
```

输出结果：

```
a8 a9 b0 b1 b2 b3 b4 b5 b6
a8 a9 b0 b1 b2 b3 b4 b5 b6
```



Class**Struct < Object**

子类: Struct::Tms。

`Struct` 是一种将许多属性捆绑在一起的便捷方式, 可以使用 `accessor` 方法访问这些属性, 而无需为之编写一个显式的类。

`Struct` 类是产生特定类的生成器, 每个生成的类被定义为存有一组变量以及访问这些变量的 `accessor` 方法。在这些例子中, 我们会把这些生成的类称作 `Customer`, 同时演示该类的一个示范实例 `joe`。

同时参见第 710 页的 `OpenStruct`。

在接下来的描述中, `symbol` 参数指的是一个符号, 它是一个引号字符串或者一个 `Symbol` (例如`:name`)。

Mixes in**Enumerable:**

```
all?, any?, collect, detect, each_with_index, entries, find, find_all,
grep, include?, inject, map, max, member?, min, partition, reject,
select, sort, sort_by, to_a, zip
```

类方法**new**

`Struct.new(<string><, symbol>+)` → `Customer`

[1.9] `Struct.new(<string><, symbol>+) { block }` → `Customer`

创建一个名为 `string` 的新类, 包含了那些给定符号对应的 `accessor` 方法。如果省略了 `string` 参数, 将会创建一个匿名的结构类。否则, 这个结构的名字将作为一个 `Struct` 类中的一个常量, 所以它必须对系统中生成的所有结构类是唯一的, 并且应当以一个大写字母开始。把一个结构类赋值给一个常量, 这样有效地把常量的名称提供给这个类。

`Struct.new` 返回新的 `Class` 对象, 然后可以用它来创建这个新结构的具体实例。下面列出的方法 (类和实例的方法), 为这个新生成的类所定义。请参见位于示例之后的描述。

Ruby 1.9 和之后的版本允许将 `block` 传递给 `struct` 的构造函数。这个 `block` 会在新结构类的上下文中被求解, 因此可以让你方便地把实例方法添加到这个新结构中。

```
# Create a structure with a name in Struct
Struct.new("Customer", :name, :address) →      Struct::Customer
Struct::Customer.new("Dave", "123 Main") →      #<struct
                                                Struct::Customer
                                                name="Dave",
                                                address="123 Main">
```

```
# Create a structure named by its constant
Customer = Struct.new(:name, :address) → Customer
Customer.new("Dave", "123 Main") → #<struct Customer
                                         name="Dave", address="123
                                         Main">
```

new*Customer.new(<obj>⁺) → joe*

创建一个结构的新实例（由 `Struct.new` 创建的类）。实际的参数个数必须小于或等于定义在这个类中的属性个数；未设置的参数默认是 `nil`，传入太多的参数会引发 `ArgumentError`。

```
Customer = Struct.new(:name, :address, :zip)

joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.name → "Joe Smith"
joe.zip → 12345
```

[]*Customer[<obj>⁺] → joe*

等同于 `new`（对这个生成的结构而言）。

```
Customer = Struct.new(:name, :address, :zip)

joe = Customer["Joe Smith", "123 Maple, Anytown NC", 12345]
joe.name → "Joe Smith"
joe.zip → 12345
```

members*Customer.members → array*

返回一个字符串数组，其中字符串表示实例变量的名称。

```
Customer = Struct.new("Customer", :name, :address, :zip)
Customer.members → ["name", "address", "zip"]
```

实例方法**==***joe == other_struct → true 或 false*

等同性（equality）——如果 `other_struct` 与这个结构相等，返回 `true`：它们必须都是由 `Struct.new` 生成的相同类，同时它们的所有实例变量的值必须相等（根据 `Object#==`）。

```
Customer = Struct.new(:name, :address, :zip)

joe      = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joejr   = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
jane    = Customer.new("Jane Doe", "456 Elm, Anytown NC", 12345)

joe == joejr → true
joe == jane  → false
```

| | |
|----|---|
| [] | <i>joe[symbol] → obj</i>
<i>joe[integer] → obj</i> |
|----|---|

属性引用 (attribute reference) —— 返回以 *symbol* 命名的或者由 *int* 索引的 (*0..length -1*) 实例变量的值。如果不存在给定的变量，引发 `NameError`，或者如果索引超出了范围，引发 `IndexError`。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe["name"] → "Joe Smith"
joe[:name] → "Joe Smith"
joe[0] → "Joe Smith"
```

| | |
|------|---|
| []= | <i>joe[symbol] = obj → obj</i>
<i>joe[int] = obj → obj</i> |
|------|---|

属性赋值 (attribute assignment) —— 把 *obj* 的值赋值给以 *symbol* 命名的或者由 *int* 索引的实例变量，同时返回 *obj* 值。如果不存在给定的变量，引发 `NameError`；或者如果索引超出了范围，引发 `IndexError`。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe["name"] = "Luke"
joe[:zip] = "90210"
joe.name → "Luke"
joe.zip → "90210"
```

| | |
|------|---|
| each | <i>joe.each { obj block } → joe</i> |
|------|---|

为每个实例变量调用 *block* 一次，把实例变量的值作为参数传入。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.each { |x| puts(x) }
```

输出结果：

```
Joe Smith
123 Maple, Anytown NC
12345
```

| | |
|-----------|--|
| each_pair | <i>joe.each_pair { symbol, obj block } → joe</i> |
|-----------|--|

1.8 为每个实例变量调用 *block* 一次，把它的名称（作为一个符号）和值作为参数传入。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.each_pair { |name, value| puts("#{name} => #{value}") }
```

输出结果：

```
name => Joe Smith
address => 123 Maple, Anytown NC
zip => 12345
```

length*joe.length* → int

返回实例变量的个数。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.length → 3
```

members

joe.members → array

返回一个字符串数组，其中字符串表示实例变量的名称。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.members → ["name", "address", "zip"]
```

size

joe.size → int

等同于 Struct#length。

to_a

joe.to_a → array

将该实例的值以一个数组返回。

```
Customer = Struct.new(:name, :address, :zip)
joe = Customer.new("Joe Smith", "123 Maple, Anytown NC", 12345)
joe.to_a[1] → "123 Maple, Anytown NC"
```

values

joe.values → array

等同于 to_a。

values_at

joe.values_at(<selector>)* → array

- 1.8 返回一个数组，含有在 *joe* 中与给定索引相对应的那些元素。这些 selector 可能是整数索引或者区间。

```
Lots = Struct.new(:a, :b, :c, :d, :e, :f)
l = Lots.new(11, 22, 33, 44, 55, 66)
l.values_at(1, 3, 5) → [22, 44, 66]
l.values_at(0, 2, 4) → [11, 33, 55]
l.values_at(-1, -3, -5) → [66, 44, 22]
```

Class Struct::Tms < Struct

这个结构由 `Process.times` 返回。它含有在支持平台上的进程时间的信息。并非所有值在所有平台上都有效。这个结构包含下面的实例变量和相应的访问方法：

`utime` 用户 CPU 时间数量，以秒计

`stime` 系统 CPU 时间数量，以秒计

`cutime` 已结束子进程的用户 CPU 时间总数，以秒计（在 Windows 平台上总是 0）

`cstime` 已结束子进程的系统 CPU 时间总数，以秒计（在 Windows 平台上总是 0）

同时参见第 626 页的 `struct` 和第 587 页的 `Process.times`。

```
def eat_cpu
  100_000.times { Math.sin(0.321) }
end
3.times { fork { eat_cpu } }
eat_cpu
Process.waitall
t = Process::times
[ t.utime, t.stime]      → [0.28, 0.03]
[ t.cutime, t.cstime ]   → [0.74, 0.01]
```

Class Symbol < Object

在 Ruby 解释器内部，Symbol 对象表示名称。可以使用 :name 字面量语法以及各种各样的 to_sym 方法来生成这些 Symbol 对象。在程序执行期间，对给定名称的字符串会创建相同的 Symbol 对象，而无论名称所处的上下文或者其含义如何。因此，如果 Fred 在某个上下文中是常量，在另外一个上下文中是方法，而在第三个上下文中是类，Symbol :Fred 在这三个上下文中是同一个对象。

```
module One
  class Fred
  end
  $f1 = :Fred
end
module Two
  Fred = 1
  $f2 = :Fred
end
def Fred()
end
$f3 = :Fred
$f1.object_id → 2526478
$f2.object_id → 2526478
$f3.object_id → 2526478
```

类方法

| all_symbols | Symbol.all_symbols → array |
|-------------|----------------------------|
|-------------|----------------------------|

1.8 返回一个数组，包含了在当前 Ruby 符号表中的所有符号。

```
Symbol.all_symbols.size → 913
Symbol.all_symbols[1,20] → [:floor, :ARGV, :Binding, :symlink,
                           :chown, :EOFError, :$/, :String,
                           :LOCK_SH, :"setuid?", :$<,
                           :default_proc, :compact, :extend, :Tms,
                           :getwd, :$=, :ThreadGroup, :"success?",
                           :wait2]
```

实例方法

| id2name | sym.id2name → string |
|---------|----------------------|
|---------|----------------------|

1.8 返回 sym 的字符串表示。在 Ruby 1.8 版本之前，符号通常表示名称；现在它们可以是任意字符串。

```
:fred.id2name → "fred"
:"99 red balloons! .id2name → "99 red balloons!"
```

inspect*sym.inspect → string*

以一个符号字面量返回 *sym* 的表示。

```
:fred.inspect          → :fred
:"99 red balloons!".inspect → :"99 red balloons!"
```

to_i

sym.to_i → fixnum

返回一个整数，在程序的一次特定执行中，每个符号的整数值是唯一的。

```
:fred.to_i           → 9857
"fred".to_sym.to_i → 9857
```

to_int

sym.to_int → fixnum

等同于 `Symbol#to_i`。允许符号有类似整数的行为。

to_s

sym.to_s → string

等同于 `Symbol#id2name`。

to_sym

sym.to_sym → sym

符号的 `Symbol!`¹⁰

¹⁰ 译注：返回的 `sym` 对象和原对象的 `object_id` 相同，因此是同一对象。

Class Thread < Object

线程封装了一次线程的执行行为，包括了 Ruby 脚本的主（main）线程。参见第 11 章从第 135 页开始的指南。

在接下来的描述中，参数 *symbol* 指的是一个符号，它是一个引号字符串或者一个 Symbol（例如：`:name`）。

类方法

| | |
|---------------------------------|---|
| <code>abort_on_exception</code> | <code>Thread.abort_on_exception</code> → true 或 false |
|---------------------------------|---|

返回全局“`abort on exception`（异常发生时中止执行）”条件的状态。默认为 `false`。当把它设置为 `true`，或者如果全局 `$DEBUG` 标志为 `true`（或许因为使用了 `-d` 命令行选项），如果在任何线程中引发了异常，所有线程将会中止（这个进程会调用 `exit(0)`）。同时参见 `Thread.abort_on_exception=`。

| | |
|----------------------------------|---|
| <code>abort_on_exception=</code> | <code>Thread.abort_on_exception</code> = <i>bool</i> → true 或 false |
|----------------------------------|---|

当设置为 `true` 时，如果引发了异常，所有线程将会中止。返回新的状态。

```
Thread.abort_on_exception = true
t1 = Thread.new do
  puts "In new thread"
  raise "Exception from thread"
end
sleep(1)
puts "not reached"
```

输出结果：

```
In new thread
prog.rb:4: Exception from thread (RuntimeError)
from prog.rb:2:in `initialize'
from prog.rb:2:in `new'
from prog.rb:2
```

| | |
|-----------------------|---|
| <code>critical</code> | <code>Thread.critical</code> → true 或 false |
|-----------------------|---|

返回全局“`thread critical`（线程临界）”条件的状态。

| | |
|------------------------|---|
| <code>critical=</code> | <code>Thread.critical</code> = <i>bool</i> → true 或 false |
|------------------------|---|

设置全局“`thread critical`（线程临界）”条件的状态并返回它。当将之设置为 `true` 时，会禁止调度任何现有的线程。但是不会阻碍创建和运行新的进程。某些线程操作（例如停止或者杀死一个线程，在当前线程中睡眠或者引发异常）可能会导致即使是在一个关键区域（`critical section`），线程也会被调度下来。`Thread.critical` 不常在日常编程中使用：它主要是支持编写线程库的程序员。

| | |
|----------------|--|
| current | Thread.current → thread |
| | 返回正在执行的线程。
Thread.current → #<Thread:0x1d5790 run> |
| exit | Thread.exit |
| | 中止当前正运行的线程，并调度另外的线程来运行。如果这个线程已经标记为要被杀死，则 exit 返回这个 Thread。如果这是主线程或者最后一个线程，则退出进程。 |
| fork | Thread.fork { block } → thread |
| | 等同于 Thread.start。 |
| kill | Thread.kill(thread) |
| | 导致给定的线程退出（参见 Thread.exit）。
count = 0
a = Thread.new { loop { count += 1 } }
sleep(0.1) → 0
Thread.kill(a) → #<Thread:0x1c947c dead>
count → 39410
a.alive? → false |
| list | Thread.list → array |
| | 返回一个 Thread 对象数组，包含了所有可运行或者已停止的线程。
Thread.new { sleep(200) }
Thread.new { 1000000.times { i i*i } }
Thread.new { Thread.stop }
Thread.list.each { thr p thr } |
| | 输出结果：
#<Thread:0x1c960c sleep>
#<Thread:0x1c9698 run>
#<Thread:0x1c96fc sleep>
#<Thread:0x1d5790 run> |
| main | Thread.main → thread |
| | 返回这个进程的主线程。
Thread.main → #<Thread:0x1d5790 run> |
| new | Thread.new(< arg >*) { args block } → thread |
| | 创建并运行一个新的线程，它执行在 block 中给出的指令。任何传递给 Thread.new 的参数会传递给这个 block。 |

```
x = Thread.new { sleep 0.1; print "x"; print "y"; print "z" }
a = Thread.new { print "a"; print "b"; sleep 0.2; print "c" }
x.join; a.join # wait for threads to finish
```

输出结果:

```
abxyzc
```

pass

Thread.pass

调用线程调度器，把执行权让给别的线程。

```
a = Thread.new { print "a"; Thread.pass; print "b" }
b = Thread.new { print "x"; Thread.pass; print "y" }
a.join; b.join
```

输出结果:

```
axby
```

start

Thread.start(<args>*){|args|block} → thread

基本上与 `Thread.new` 相同。但是，如果 `Thread` 类被子类化，那么在子类中调用 `start` 不会调用子类的 `initialize` 方法。

stop

Thread.stop

停止当前线程的执行，把它置于“睡眠”状态，同时调度另外的线程来运行。

把“critical”条件重置为 `false`。

```
a = Thread.new { print "a"; Thread.stop; print "c" }
Thread.pass
print "b"
a.run
a.join
```

输出结果:

```
abc
```

实例方法

[]

thr[symbol] → obj 或 nil

属性引用 (attribute reference) —— 使用符号或者字符串名称，返回线程局部 (thread-local) 变量的值。如果指定的变量不存在，则返回 `nil`。

```
a = Thread.new { Thread.current["name"] = "A"; Thread.stop }
b = Thread.new { Thread.current[:name] = "B"; Thread.stop }
c = Thread.new { Thread.current["name"] = "C"; Thread.stop }
Thread.list.each {|x| puts "#{x.inspect} : #{x[:name]}"}  
#<Thread:0x1c92b0 sleep>: C
#<Thread:0x1c9328 sleep>: B
#<Thread:0x1c93b4 sleep>: A
#<Thread:0x1d5790 run>:
```

输出结果:

```
#<Thread:0x1c92b0 sleep>: C
#<Thread:0x1c9328 sleep>: B
#<Thread:0x1c93b4 sleep>: A
#<Thread:0x1d5790 run>:
```

[]=

thr[symbol] = obj→obj

属性赋值 (attribute assignment) —— 使用符号或者字符串，设置或者创建线程局部变量的值。同时参见 Thread#[]。

abort_on_exception*thr.abort_on_exception→true 或 false*

返回 *thr* 线程局部的 “abort on exception” 条件的状态。默认是 *false*。同时参见 Thread.abort_on_exception=。

abort_on_exception=*thr.abort_on_exception=true 或 false→true 或 false*

当被设置为 *true* 时，如果在 *thr* 中引发了异常，导致所有线程（包括主程序）中止。这个进程实际上将调用 `exit(0)`。

alive?*thr.alive?→true 或 false*

如果 *thr* 正运行或者处于睡眠状态，则返回 *true*。

```
thr = Thread.new { }
thr.join           → #<Thread:0x1c9bfc dead>
Thread.current.alive? → true
thr.alive?         → false
```

exit*thr.exit→thr 或 nil*

终止 *thr* 线程，同时调度另外的线程来运行。如果这个线程已经标记为要被杀死，*exit* 返回这个 `Thread`。如果这是主线程或者是最后一个线程，退出这个进程。

group*thr.group→thread_group*

1.8

返回 *thr* 所属的 `ThreadGroup`，或者返回 *nil*。

```
thread = Thread.new { sleep 99 }
Thread.current.group.list      → [#<Thread:0x1c9418 sleep>,
                                  #<Thread:0x1d5790 run>]
new_group = ThreadGroup.new
thread.group.list              → [#<Thread:0x1c9418 sleep>,
                                  #<Thread:0x1d5790 run>]
new_group.add(thread)
thread.group.list              → [#<Thread:0x1c9418 sleep>]
Thread.current.group.list     → [#<Thread:0x1d5790 run>]
```

join*thr.join→thr**thr.join(limit)→thr*

1.8

调用者线程会挂起 (`suspend`)，同时运行 *thr* 线程。它直到 *thr* 退出或者已经运行了 *limit* 秒时间才会退出。如果时间 *limit* 到期了，则返回 *nil*；否则返回 *thr*。

当主程序退出时，任何没有被 `join` 的线程将会被杀死。如果 `thr` 先前已经引发了一个异常，同时没有设置 `abort_on_exception` 和 `$DEBUG` 标志（因此这个异常还未处理），这个时候异常将会被处理。

```
a = Thread.new { print "a"; sleep(10); print "c" }
x = Thread.new { print "x"; Thread.pass; print "y"; print "z" }
x.join # Let x thread finish, a will be killed on exit.
```

输出结果：

```
axyz
```

下面的例子说明了 `limit` 参数。

```
y = Thread.new { 4.times { sleep 0.1; print "tick...\n" } }
print "Waiting\n" until y.join(0.15)
```

输出结果：

```
tick...
Waiting
tick...
tick...
Waiting
tick...
```

| | |
|-------------|-------------------------|
| keys | <i>thr.keys → array</i> |
|-------------|-------------------------|

1.8 返回线程局部变量的名称数组（以符号的形式）。

```
thr = Thread.new do
  Thread.current[:cat] = 'meow'
  Thread.current["dog"] = 'woof'
end
thr.join →      #<Thread:0x1c965c dead>
thr.keys →      [:dog, :cat]
```

| | |
|-------------|--|
| key? | <i>thr.key?(symbol) → true 或 false</i> |
|-------------|--|

如果线程局部变量中存在给定的字符串（或者符号），则返回 `true`。

```
me = Thread.current
me[:oliver] = "a"
me.key?(:oliver) → true
me.key?(:stanley) → false
```

| | |
|-------------|-----------------|
| kill | <i>thr.kill</i> |
|-------------|-----------------|

等同于 `Thread#exit`。

| | |
|-----------------|---------------------------|
| priority | <i>thr.priority → int</i> |
|-----------------|---------------------------|

返回 `thr` 的优先级。默认是 0；更高优先级的线程将在较低优先级的线程之前运行。

```
Thread.current.priority → 0
```

priority=*thr.priority= int → thr*

把 *thr* 的优先级设置为 *integer*。更高优先级的线程将在较低优先级的线程之前运行。

```
count1 = count2 = 0
a = Thread.new do
  loop { count1 += 1 }
end
a.priority = -1

b = Thread.new do
  loop { count2 += 1 }
end
b.priority = -2
sleep 1
Thread.critical = 1
count1 → 451372
count2 → 4514
```

raise

thr.raise

thr.raise(message)
thr.raise(exception <, message <, array >>)

- 从 *thr* 引发一个异常（关于详细的信息，请参见第 527 页的 *Kernel.raise*）。调用者不用非得是 *thr*。

```
Thread.abort_on_exception = true
a = Thread.new { sleep(200) }
a.raise("Gotcha")
```

输出结果：

```
prog.rb:3: Gotcha (RuntimeError)
from prog.rb:2:in `initialize'
from prog.rb:2:in `new'
from prog.rb:2
```

run*thr.run → thr*

唤醒 *thr*，使它可以参与调度。如果不是在关键区域（critical section），则调用调度器。

```
a = Thread.new { puts "a"; Thread.stop; puts "c" }
Thread.pass
puts "Got here"
a.run
a.join
```

输出结果：

```
a
Got here
c
```

safe_level*thr.safe_level* → *int*

返回 *thr* 的有效安全级别。设置线程局部安全级别，可以有助于实现用来运行不安全代码的 *sandbox*。

```
thr = Thread.new { $SAFE = 3; sleep }
Thread.current.safe_level      → 0
thr.safe_level                 → 3
```

status*thr.status* → *string, false* 或 *nil*

返回 *thr* 的状态：如果 *thr* 正在睡眠或者在 I/O 上等待，返回 *sleep*；如果 *thr* 正在执行，返回 *run*；如果 *thr* 将要中止，返回 *aborting*；如果 *thr* 已正常退出，返回 *false*；如果 *thr* 由于异常已经被中止，返回 *nil*。

```
a = Thread.new { raise("die now") }
b = Thread.new { Thread.stop }
c = Thread.new { Thread.exit }
d = Thread.new { sleep }
Thread.critical = true
d.kill                      →      #<Thread:0x1c8e00 aborting>
a.status                     →      nil
b.status                     →      "sleep"
c.status                     →      false
d.status                     →      "aborting"
Thread.current.status        →      "run"
```

stop?*thr.stop?* → *true* 或 *false*

如果 *thr* 已退出或者正在睡眠，返回 *true*。

```
a = Thread.new { Thread.stop }
b = Thread.current
a.stop? → true
b.stop? → false
```

terminate*thr.terminate*

等同于 *Thread#exit*。

value*thr.value* → *obj*

等待 *thr* 结束（通过 *Thread#join*），同时返回它的值。

```
a = Thread.new { 2 + 2 }
a.value → 4
```

wakeup*thr.wakeup* → *thr*

把 *thr* 标记为可调度（当然，它可能依然还阻塞在 I/O 上）。不会调用调度器（参见 *Thread#run*）。

Class ThreadGroup < Object

ThreadGroup 可以追踪很多线程。线程在某个时候只能属于一个 ThreadGroup；把线程添加到一个线程组，将会把它从当前线程组中删除。新创建的线程属于创建它们的那个线程的线程组。

ThreadGroup 常量

Default 默认线程组。

类方法

new

ThreadGroup.new → thgrp

返回一个新创建的 ThreadGroup。这个组初始是空的。

实例方法

add

thgrp.add(thread) → thgrp

把给定的线程添加到这个线程组，从它先前属于的线程组中删除这个线程。

```
puts "Default group is #{ThreadGroup::Default.list}"
tg = ThreadGroup.new
t1 = Thread.new { sleep }
t2 = Thread.new { sleep }
puts "t1 is #{t1}, t2 is #{t2}"
tg.add(t1)
puts "Default group now #{ThreadGroup::Default.list}"
puts "tg group now #{tg.list}"
```

输出结果：

```
Default group is #<Thread:0x1d6790>
t1 is #<Thread:0x1ca24c>, t2 is #<Thread:0x1cale8>
Default group now #<Thread:0x1cale8>#<Thread:0x1d6790>
tg group now #<Thread:0x1ca24c>
```

enclose

thgrp.enclose → thgrp

1.8.

阻止向（从） thgrp 添加线程或删除线程。依然可以启动新的线程。

```
thread = Thread.new { sleep 99 }
group = ThreadGroup.new
group.add(thread)
group.enclose
ThreadGroup::Default.add(thread)
```

输出结果：

```
prog.rb:5:in `add': can't move from the enclosed thread group (Thread Error)
from prog.rb:5
```

| | |
|------------------|-------------------------------------|
| enclosed? | <i>thgrp.enclose</i> → true 或 false |
|------------------|-------------------------------------|

1.8. 如果这个线程组已经 enclosed, 返回 true。

| | |
|---------------|---------------------|
| freeze | <i>thgrp.freeze</i> |
|---------------|---------------------|

停止向（从） *thgrp* 添加线程或删除线程。也不会启动新线程。

| | |
|-------------|---------------------------|
| list | <i>thgrp.list</i> → array |
|-------------|---------------------------|

返回一个 Thread 对象数组，包含了属于这个线程组的所有现有 Thread 对象。

```
ThreadGroup::Default.list → [#<Thread:0x1d6790 run>]
```

Class Time < Object

`Time` 是日期和时间的一个抽象。Ruby 在内部将时间存储为自新纪元——1970 年元月 1 日 00:00 UTC——以来的秒数和微秒数。在一些操作系统上，该时间偏移允许是负数。同时参见分别在第 665 页和第 713 页描述的库模块 `Date` 和 `ParseDate`。

`Time` 类同等地对待 `GMT`（格林威治标准时间）和 `UTC`（Coordinated Universal Time）¹¹。`GMT` 是指向该基线时间的较早方式，但是它一直存在于 `POSIX` 系统调用的名称中。

所有的时间都保存有微秒数。当比较两个时间时，请小心这个事实。它们在显示时明显是相等的，但是它们在比较时却可能不同。

Mixes in

Comparable:

`<`, `<=`, `==`, `>=`, `>`, `between?`

类方法

at

`Time.at(time) → time`

`Time.at(seconds <, microseconds >) → time`

使用由 `time` 给定的值或者自新纪元以来的秒数（和可选的微秒数）来创建新的 1.8. 时间对象。在某些系统上这个时间偏移可以是负数，这不是一个可移植的特性。

| | | |
|----------------------------------|---|------------------------------|
| <code>Time.at(0)</code> | → | Wed Dec 31 18:00:00 CST 1969 |
| <code>Time.at(946702800)</code> | → | Fri Dec 31 23:00:00 CST 1999 |
| <code>Time.at(-284061600)</code> | → | Sat Dec 31 00:00:00 CST 1960 |

gm

`Time.gm(year <, month, day, hour, min, sec, usec >) → time`

`Time.gm(sec, min, hour, day, month, year, wday, yday, isdst, tz) → time`

基于给定的值创建一个时间对象，解释为 `UTC`（`GMT`）。必须指定 `year` 参数。别的值默认是那个域（可能是 `nil` 或者被省略）的最小值。可以使用 1 到 12 之间的数字或者三个字母的英文月份名称来指定 `Month`。可以使用 24 制的小时数（0..23）指定 `Hour`。如果任何一个值超出了它的范围，会引发 `ArgumentError`。也可以接受十个以 `Time#to_a` 的输出顺序传入的参数。

`Time.gm(2000, "jan", 1, 20, 15, 1) → Sat Jan 01 20:15:01 UTC 2000`

¹¹ 是的，`UTC` 真的代表 Coordinated Universal Time。有一个委员会涉及此事。

| | |
|-------|---|
| local | <code>Time.local(year <, month, day, hour, min, sec, usec >) → time</code> |
| | <code>Time.local(sec, min, hour, day, month, year, wday, yday, isdst, tz) → time</code> |

与 `Time.gm` 相同，但是以当地时区（local time zone）来解释这些值。第一种形式使用 `ParseDate#parsedate`（在第 713 页描述）返回的结果，用来构建一个 `Time` 对象。

```
require 'parsedate'
Time.local(2000, "jan", 1, 20, 15, 1) → Sat Jan 01 20:15:01 CST 2000
res = ParseDate.parsedate("2000-01-01 20:15:01")
Time.local(*res) → Sat Jan 01 20:15:01 CST 2000
```

| | |
|--------|---|
| mktime | <code>Time.mktime(year, month, day, hour, min, sec, usec) → time</code> |
|--------|---|

等同于 `Time.local`。

| | |
|-----|------------------------------|
| new | <code>Time.new → time</code> |
|-----|------------------------------|

返回一个 `Time` 对象，把它初始化为当前系统时间。请注意：将会使用你系统上可用的时钟分辨率来创建这个对象，所以可能会包含秒的小数位（fractional second）。

```
a = Time.new → Thu Aug 26 22:38:02 CDT 2004
b = Time.new → Thu Aug 26 22:38:02 CDT 2004
a == b → false
"%."6f" % a.to_f → "1093577882.29 2375"
"%."6f" % b.to_f → "1093577882.29 3183"
```

| | |
|-----|------------------------------|
| now | <code>Time.now → time</code> |
|-----|------------------------------|

等同于 `Time.new`。

| | |
|-------|--------------------------------------|
| times | <code>Time.times → struct_tms</code> |
|-------|--------------------------------------|

1.8. 不推荐使用，请使用第 587 页描述的 `Process.times`。

| | |
|-----|---|
| utc | <code>Time.utc(year <, month, day, hour, min, sec, usec >) → time</code> |
| | <code>Time.utc(sec, min, hour, day, month, year, wday, yday, isdst, tz) → time</code> |

等同于 `Time.gm`。

```
Time.utc(2000, "jan", 1, 20, 15, 1) → Sat Jan 01 20:15:01 UTC 2000
```

实例方法

+

time + numeric → time

增加 (addition) —— 把若干秒数 (可能含有小数) 增加给 *time*, 同时以一个新的时间返回它的值。

```
t = Time.now      → Thu Aug 26 22:38:02 CDT 2004
t + (60 * 60 * 24) → Fri Aug 27 22:38:02 CDT 2004
```

-

*time - time → float**time - numeric → time*

差 (difference) —— 返回新的 *time*, 它表示了两个 *time* 之间的差, 或者从 *time* 中减去给定的 *numeric* 秒数。

```
t = Time.now      → Thu Aug 26 22:38:02 CDT 2004
t2 = t + 2592000 → Sat Sep 25 22:38:02 CDT 2004
t2 - t           → 2592000.0
t2 - 2592000     → Thu Aug 26 22:38:02 CDT 2004
```

<=>

*time <=> other_time → -1, 0, +1**time <=> numeric → -1, 0, +1*

比较 (comparison) —— 使用 *other_time* 或者 *numeric* 与 *time* 进行比较, *numeric* 是自新纪元以来的秒数 (可能含有小数)。

```
t = Time.now      → Thu Aug 26 22:38:02 CDT 2004
t2 = t + 2592000 → Sat Sep 25 22:38:02 CDT 2004
t <=> t2          → -1
t2 <=> t          → 1
t <=> t           → 0
```

asctime*time.asctime → string*

返回 *time* 的规整字符串表示。

```
Time.now.asctime → "Thu Aug 26 22:38:02 2004"
```

ctime*time.ctime → string*

等同于 *Time#asctime*。

day*time.day → int*

返回 *time* 在这个月中的天数 (1..n)。

```
t = Time.now → Thu Aug 26 22:38:02 CDT 2004
t.day        → 26
```

dst?*time.dst? → true 或 false*

1.8.

等同于 *Time#isdst*。

```
Time.local(2000, 7, 1).dst? → true
Time.local(2000, 1, 1).dst? → false
```

getgm*time.getgm*→*time*

1.8 返回新的 Time 对象，它以 UTC 表示时间。

```
t = Time.local(2000,1,1,20,15,1) → Sat Jan 01 20:15:01 CST 2000
t.gmt? → false
y = t.getgm → Sun Jan 02 02:15:01 UTC 2000
y.gmt? → true
t == y → true
```

getlocal*time.getlocal*→*time*

1.8 返回新的 Time 对象，它以当地时间（使用这个进程有效的当地时区）来表示 *time*。

```
t = Time.gm(2000,1,1,20,15,1) → Sat Jan 01 20:15:01 UTC 2000
t.gmt? → true
l = t.getlocal → Sat Jan 01 14:15:01 CST 2000
l.gmt? → false
t == l → true
```

getutc*time.getutc*→*time*

1.8 等同于 Time#getgm。

gmt?*time.gmt?*→true 或 false

如果 *time* 以 UTC (GMT) 表示，返回 true。

```
t = Time.now → Thu Aug 26 22:38:02 CDT 2004
t.gmt? → false
t = Time.gm(2000,1,1,20,15,1) → Sat Jan 01 20:15:01 UTC 2000
t.gmt? → true
```

gmtime*time.gmtime*→*time*

把 *time* 转换成 UTC (GMT)，同时修改这个接受者。

```
t = Time.now → Thu Aug 26 22:38:02 CDT 2004
t.gmt? → false
t.gmtime → Fri Aug 27 03:38:02 UTC 2004
t.gmt? → true
```

gmt_offset*time.gmt_offset*→int

1.8 返回在 *time* 和 UTC 时区之间的秒数偏移。

```
t = Time.gm(2000,1,1,20,15,1) → Sat Jan 01 20:15:01 UTC 2000
t.gmt_offset → 0
l = t.getlocal → Sat Jan 01 14:15:01 CST 2000
l.gmt_offset → -21600
```

| | |
|------------------|--|
| gmtoff | <i>time.gmtoff</i> → <i>int</i> |
| 1.8 | 等同于 <i>Time#gmt_offset</i> 。 |
| hour | <i>time.hour</i> → <i>int</i> |
| | 返回 <i>time</i> 在这一天的小时数 (0..23)。
<pre>t = Time.now → Thu Aug 26 22:38:02 CDT 2004 t.hour → 22</pre> |
| isdst | <i>time.isdst</i> → <i>true</i> 或 <i>false</i> |
| | 如果 <i>time</i> 位于其所在时区的夏令时 (Daylight Saving Time) 期内，返回 <i>true</i> 。
<pre>Time.local(2000, 7, 1).isdst → true Time.local(2000, 1, 1).isdst → false</pre> |
| localtime | <i>time.localtime</i> → <i>time</i> |
| | 把 <i>time</i> 转换成当地时间（使用这个进程有效的当地时区），同时修改这个接收者。
<pre>t = Time.gm(2000, "jan", 1, 20, 15, 1) t.gmt? → true t.localtime → Sat Jan 01 14:15:01 CST 2000 t.gmt? → false</pre> |
| mday | <i>time.mday</i> → <i>int</i> |
| | 等同于 <i>Time#day</i> 。 |
| min | <i>time.min</i> → <i>int</i> |
| | 返回 <i>time</i> 在这个小时的分钟数 (0..59)。
<pre>t = Time.now → Thu Aug 26 22:38:03 CDT 2004 t.min → 38</pre> |
| mon | <i>time.mon</i> → <i>int</i> |
| | 返回 <i>time</i> 在这年中的月份 (1..12)。
<pre>t = Time.now → Thu Aug 26 22:38:03 CDT 2004 t.mon → 8</pre> |
| month | <i>time.month</i> → <i>int</i> |
| | 等同于 <i>Time#mon</i> 。 |

sec*time.sec* → *int*

返回 *time* 在这分钟的秒数 (0..60)¹²。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.sec → 3
```

strftime*time.strftime(format)→string*

根据给定 *format* 字符串中的指令 (directive) 对 *time* 进行格式化。关于可用的值，参见下页的表 27.15。任何以非指令方式给出的文本将会直接传入到 (pass through) 输出字符串。

```
t = Time.now
t.strftime("Printed on %m/%d/%Y") → "Printed on 08/26/2004"
t.strftime("at %I:%M%p") → "at 10:38PM"
```

to_a*time.to_a→array*

返回表示 *time* 值的十个元素的数组: [sec, min, hour, day, month, year, wday, yday, isdst, zone]。关于每个值有效范围的说明，请参见各个方法。这十个元素可以直接传递给 *Time.utc* 或者 *Time.local* 方法来创建新的 *Time*。

```
now = Time.now → Thu Aug 26 22:38:03 CDT 2004
t = now.to_a → [3, 38, 22, 26, 8, 2004, 4, 239, true, "CDT"]
```

to_f*time.to_f→float*

以浮点数方式返回的 *time* 值 (自新纪元以来的秒数)。

```
t = Time.now
"%10.5f" % t.to_f → "1093577883.33171"
t.to_i → 1093577883
```

to_i*time.to_i→int*

以整数方式返回的 *time* 值 (自新纪元以来的秒数)。

```
t = Time.now
"%10.5f" % t.to_f → "1093577883.37052"
t.to_i → 1093577883
```

to_s*time.to_s→string*

返回表示 *time* 的字符串。等同于用 %a%b%d %H:%M:%S%Z%Y 格式字符串调用 *Time#strftime*。

```
Time.now.to_s → "Thu Aug 26 22:38:03 CDT 2004"
```

¹² 是的，秒真的可以处于 0 到 60 之间。这可以让系统随时插入闰秒 (leap second) 来校正下面这个事实：年并非是整长的小时数。

表 27.15 Time#strftime 指令

| 格 式 含 义 | |
|---------|--|
| %a | 缩写的星期名称 (“Sun”) |
| %A | 完全的星期名称 (“Sunday”) |
| %b | 缩写的月份名称 (“Jan”) |
| %B | 完全的月份名称 (“January”) |
| %c | 首选当地日期和时间表示 |
| %d | 一月的天数 (01..31) |
| %H | 一天的小时数, 24-小时时钟 (00..23) |
| %I | 一天的小时数, 12-小时时钟 (01..12) |
| %j | 一年中的天数 (001..366) |
| %m | 月份 (01..12) |
| %M | 一小时中的分钟数 (00..59) |
| %p | 上下午的指示 (“AM”或者“PM”) |
| %S | 一分钟中的秒数 (00..60) |
| %U | 今年的星期数 (00..53); 从第一个星期日开始数起, 它作为第一个星期的第一天 |
| %W | 今年的星期数 (00..53); 从第一个星期一开始数起, 它作为第一个星期的第一天 |
| %w | 一个星期中的天数 (星期日是 0, 0..6) |
| %x | 首选的日期表示; 只有日期, 没有时间 |
| %X | 首选的时间表示; 只有时间, 没有日期 |
| %y | 不带世纪 (century) 的年数 (00..99) |
| %Y | 带有世纪的年数 |
| %Z | 时区名称 |
| %% | %字符的字面量 |

tv_sec *time.tv_sec* → int

等同于 Time#to_i。

tv_usec *time.tv_usec* → int

等同于 Time#usec。

usec *time.usec* → int

只是返回 *time* 的微秒数。

| | | |
|-------------------|---|------------------------------|
| t = Time.now | → | Thu Aug 26 22:38:03 CDT 2004 |
| "%10.6f" % t.to_f | → | "1093577883.448204" |
| t.usec | → | 448204 |

| | |
|------------|------------------------|
| utc | <i>time.utc → time</i> |
|------------|------------------------|

等同于 Time#gmtime。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.utc? → false
t.utc → Fri Aug 27 03:38:03 UTC 2004
t.utc? → true
```

| | |
|-------------|---------------------------------|
| utc? | <i>time.utc? → true 或 false</i> |
|-------------|---------------------------------|

如果 *time* 以 UTC (GMT) 表示, 返回 true。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.utc? → false
t = Time.gm(2000, "jan", 1, 20, 15, 1) → Sat Jan 01 20:15:01 UTC 2000
t.utc? → true
```

| | |
|-------------------|--------------------------------------|
| utc_offset | <i>_offset time.utc_offset → int</i> |
|-------------------|--------------------------------------|

| | |
|------------|----------------------|
| <u>1.8</u> | 等同于 Time#gmt_offset。 |
|------------|----------------------|

| | |
|-------------|------------------------|
| wday | <i>time.wday → int</i> |
|-------------|------------------------|

返回一个表示星期中天数的整数, 0..6, 其中 Sunday == 0。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.wday → 4
```

| | |
|-------------|------------------------|
| yday | <i>time.yday → int</i> |
|-------------|------------------------|

返回一个表示年的天数的整数, 1..366。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.yday → 239
```

| | |
|-------------|------------------------|
| year | <i>time.year → int</i> |
|-------------|------------------------|

返回 *time* 的年数 (包含世纪)。

```
t = Time.now → Thu Aug 26 22:38:03 CDT 2004
t.year → 2004
```

| | |
|-------------|---------------------------|
| zone | <i>time.zone → string</i> |
|-------------|---------------------------|

| | |
|------------|--|
| <u>1.8</u> | 返回 <i>time</i> 使用的时区名称。对于 UTC 时间, Ruby 1.8 版本返回 “UTC” 而不是 “GMT”。 |
|------------|--|

```
t = Time.gm(2000, "jan", 1, 20, 15, 1)
t.zone → "UTC"
t = Time.local(2000, "jan", 1, 20, 15, 1)
t.zone → "CST"
```

Class TrueClass < Object

全局变量 `true` 只是 `TrueClass` 类的一个实例，它在布尔表达式中表示一个逻辑为真的值。`TrueClass` 类提供了一些操作符，允许在逻辑表达式中使用 `true`。

实例方法
&`true & obj → true 或 false`

与 (And) ——如果 `obj` 是 `nil` 或者 `false`，返回 `false`；否则返回 `true`。

^`true ^ obj → true 或 false`

异或 (Exclusive Or) ——如果 `obj` 是 `nil` 或者 `false`，返回 `false`；否则返回 `true`。

|`true | obj → true`

或 (Or) ——返回 `true`。当 `obj` 是方法调用的一个参数时，它总是会被求解；在这种情况下，不会执行短路 (short-circuit) 求解。

```
true | puts("or")
true || puts("logical or")
```

输出结果：

```
or
```

Class UnboundMethod < Object

1.8 Ruby 支持两种方法对象化 (objectified) 的形式。Method 类用来表示那些与特定对象相关联的方法：这些方法对象被绑定到这个对象。可以使用 Object#method 创建与一个对象相绑定的方法对象。

Ruby 也支持未绑定方法，它们是那些没有与特定对象相关联的方法对象。可以对一个已绑定的对象调用 unbind 或者调用 Module#instance_method 来创建这些方法对象。

未绑定方法只有当它们绑定到一个对象后才能被调用。未绑定方法的原始类必须是这个对象的类或者超类（通过 kind_of?）。

```
class Square
  def area
    @side * @side
  end
  def initialize(side)
    @side = side
  end
end

area_unbound = Square.instance_method(:area)

s = Square.new(12)
area = area_unbound.bind(s)
area.call → 144
```

未绑定方法是当该方法被对象化时对它的一个引用：对这个底层类的后续修改不会影响该未绑定方法。

```
class Test
  def test
    :original
  end
end

um = Test.instance_method(:test)
class Test
  def test
    :modified
  end
end
t = Test.new
t.test → :modified
um.bind(t).call → :original
```

实例方法

arity

umeth.arity → fixnum

参见第 543 页的 `Method#arity`。

bind

umeth.bind(obj) → method

- 1.8 把 *umeth* 绑定到 *obj*。如果 *umeth* 最初是从 `Klass` 类中获得的，则 `obj.kind_of?(Klass)` 一定为 `true`。

```
class A
  def test
    puts "In test, class = #{self.class}"
  end
end
class B < A
end
class C < B
end
um = B.instance_method(:test)
bm = um.bind(C.new)
bm.call
bm = um.bind(B.new)
bm.call
bm = um.bind(A.new)
bm.call
```

输出结果：

```
In test, class = C
In test, class = B
prog.rb:16:in `bind': bind argument must be an instance of B (TypeError)
from prog.rb:16
```

标准库

Standard Library

Ruby 解释器带有大量内建的类、模块和方法——它们成为运行程序的一部分。如果你需要的功能不属于这个内建体系（repertoire），通常你能在某一个库中找到它，并把这个库包括（require）到你的程序中。

互联网上有大量的 Ruby 库。比如 Ruby Application Archive¹ 和 RubyForge² 站点有极多的索引（indices）和大量源码。

当然，Ruby 也提供了数量众多的标准库。其中一些库是用纯 Ruby 语言编写的，因此可以在所有 Ruby 平台上使用它们。还有一些库是 Ruby 的扩展，其中某些扩展只有当你的系统支持它们所需的资源时才是可用的。所有标准库都可以通过使用 require，把它们包括到你的 Ruby 程序中。同时，与互联网上发现的那些库不同，你几乎可以确保所有 Ruby 用户都已经把这些标准库安装到了他们的机器上。

在本章中，我们会以一种新的瑞典式自助餐（smorgasbord）格式来介绍标准库。不会在一些库的细枝末节上花费太多笔墨，相反，本章介绍了标准库的全部内容，每页介绍一个库。我们会为每个库给出一些介绍性的注解（note），同时会给出一到两个使用样例。在这里你将找不到详细的方法描述：有关这方面的内容请参考库本身的文档。

“请参考库本身的文档”是一个相当好的建议，但是你可以在哪里找到它呢？答案是“视情况而定”。一些库已经使用了 RDoc 文档（参见第 16 章）。这意味着你可以使用 ri 命令来得到它们的文档。比如，你也许能够从命令行上看到如下的关于 Base64 标准库中 decode64 方法的文档。

¹ <http://raa.ruby-lang.org>.

² <http://rubyforge.org>.

```
% ri Base64.decode64
-----
-----Base64#decode64
decode64(str)
-----
>Returns the Base64-decoded version of str.

    require 'base64'
    str ='VGhpcyBpcyBsaW5lIG9uZQpUaGlzIG' +
          'lzIGxpbmUgdHdvClRoaXMgaXMgbGlu' +
          'ZSB0aHJ1ZQpBbmQgc28gb24uLi4K'
    puts Base64.decode64(str)

Generates:

This is line one
This is line two
This is line three
And so on...
```

如果没有 Rdoc 文档可用，下一个要查找的地方是这个库本身。如果你有 Ruby 的发行源码，它们位于 ext/ 和 lib/ 子目录中。如果只安装了二进制文件，你依然可以找到那些纯 Ruby 库模块的源码（通常是在你 Ruby 安装路径的 lib/ruby/1.8/ 子目录中）。库的源码目录常常包含了一些原作者还未来得及把它们转换成 Rdoc 格式的文档。

如果你依然没有找到任何文档，那么求助于 Google。许多 Ruby 标准库作为外部项目单独存在。库的作者独立地开发它们，然后定期地把代码整合到标准的 Ruby 分发中。比如，如果你想得到 YAML 库 API 的详细信息，用 google 搜索 “yaml ruby” 会把你引向 <http://yaml4r.sourceforge.net>。欣赏一番 *why the lucky stiff* 的艺术佳作之后，点击一下 Doc 链接，你会看到 40 多页的参考手册。

下一个求助的地方是 rubytalk 邮件列表。在那里提出一个（有礼貌地）问题，如果运气好的话，在短短几个小时之内，你就可以得到一个知识渊博的回答。关于如何订阅邮件列表，请参见第 784 页。

如果你依然没有找到任何文档，你总是可以遵循 Obi Wan 的忠告，做我们以前写 Ruby 文档时所做的事情——直接阅读源码（use the source）。你会惊讶地发现，阅读 Ruby 库的实际源码同时弄清用法的细节，是多么简单。

给定一组字符串，计算出这些字符串的明确的缩写集合，并返回一个散列表，其键（key）是所有可能的缩写而值是相应的完整字符串。因此，给定输入为“car”和“cone”，指向“car”的键可能是“ca”和“car”，而指向“cone”的键可能是“co”、“con”和“cone”。

可以指定一个可选的模式或者字符串——只有那些匹配这个模式或者以这个指定字符串开头的输入字符串才会被考虑添加到输出散列表中。

将包含 Abbrev 库添加 abbrev 方法到 Array 类中。

- 显示某些词的缩写集合。

```
require 'abbrev'

Abbrev::abbrev(['ruby', 'rules']) → {"rules"=>"rules",
                                         "ruby"=>"ruby",
                                         "rul"=>"rules",
                                         "rub"=>"ruby",
                                         "rule"=>"rules"}

%w{ car cone }.abbrev → {"co"=>"cone",
                           "con"=>"cone",
                           "cone"=>"cone",
                           "ca"=>"car", "car"=>"car"}

%w{ car cone }.abbrev("ca") → {"ca"=>"car",
                                 "car"=>"car"}
```

- 使用缩写的一个普通的命令循环。

```
require 'abbrev'

COMMANDS = %w{ sample send start status stop }.abbrev
while line = gets
  line = line.chomp
  case COMMANDS[line]
  when "sample": # ...
  when "send":   # ...
  # ...
  else
    STDERR.puts "Unknown command: #{line}"
  end
end
```

使用 Base64 表示法对二进制数据进行编码和解码。这可以让你用纯可打印字符表示任何二进制数据。在 RFC 2045 (<http://www.faqs.org/rfcs/rfc2045.html>) 中详细说明了这种编码格式。

1.8. 在 Ruby 1.8.2 之前的版本中，这些方法被添加到全局的名字空间。现在已经不推荐使用了；相反，这些方法应当作为 Base64 模块的成员来访问。

- 对已编码的字符串进行解码。

```
require 'base64'
str = 'VGhpcyBpcyBsaW5lIG9uZQpUaGlzIG' +
      'lzIGxpbmUgdHdvClRoaXMgaXMgbGlu' +
      'ZSB0aHJlZQpBbmQgc28gb24uLi4K'
puts Base64.decode64(str)
```

输出结果：

```
This is line one
This is line two
This is line three
And so on...
```

- 转换和返回字符串。

```
require 'base64'
puts Base64.encode64("Now is the time\n to learn Ruby")
```

输出结果：

```
Tm93IGlzIHRoZSB0aW1lCnRvIGx1YXJuIFJ1Ynk=
```

- 把字符串转换成 Base64 编码格式，并且把它打印到标准输出。

```
require 'base64'
Base64.b64encode("Now is the time\n to learn Ruby")
```

输出结果：

```
Tm93IGlzIHRoZSB0aW1lCnRvIGx1YXJuIFJ1Ynk=
```

允许对代码的执行进行计时，并以表格形式给出计时结果。如果把 Benchmark 模块包含在你的顶层（top-level）环境中，使用这个模块变得更加容易。

同时参见：Profile（第 717 页）

- 比较三种类型的 dispatch 方法的成本。

```
require 'benchmark'
include Benchmark
string = "Stormy Weather"
m = string.method(:length)
bm(6) do |x|
  x.report("call") { 10_000.times { m.call } }
  x.report("send") { 10_000.times { string.send(:length) } }
  x.report("eval") { 10_000.times { eval "string.length" } }
end
```

输出结果：

| | user | system | total | real |
|------|----------|----------|----------|-------------|
| call | 0.020000 | 0.000000 | 0.020000 | (0.045998) |
| send | 0.040000 | 0.000000 | 0.040000 | (0.051318) |
| eval | 0.130000 | 0.000000 | 0.130000 | (0.177950) |

- 哪种方法更好：一次性地读取字典的所有内容，然后分隔出各个词条；或者每次只读取一行内容，然后分隔词条？在计时之前，使用 bmbm 进行一次预演。

```
require 'benchmark'
include Benchmark
bmbm(6) do |x|
  x.report("all") do
    str = File.read("/usr/share/dict/words")
    words = str.scan(/[-\w']+/)
  end
  x.report("lines") do
    words = []
    File.foreach("/usr/share/dict/words") do |line|
      words << line.chomp
    end
  end
end
```

输出结果：

| | user | system | total | real |
|------------------------|----------|----------|----------|-------------|
| Rehearsal----- | | | | |
| all | 0.980000 | 0.070000 | 1.050000 | (1.256552) |
| lines | 2.310000 | 0.120000 | 2.430000 | (2.720674) |
| -----total:1.010000sec | | | | |
| | user | system | total | real |
| all | 0.870000 | 0.030000 | 0.900000 | (0.949623) |
| lines | 1.720000 | 0.030000 | 1.750000 | (1.926910) |

Library **BigDecimal****高精度十进制数**

Ruby 的标准 `Bignum` 类支持有很多数字位的整数。`BigDecimal` 类支持有很多小数位的十进制数字。这个标准库支持了所有标准的算术运算。`BigDecimal` 也提供了一些扩展库。

bigdecimal/ludcmp

对矩阵执行 LU 分解。

bigdecimal/math

连同计算 π 和 e 的函数一起，提供了超越函数（transcendental function） $\sqrt{}$, \sin , \cos , \tan , \exp 和 \log 。所有函数都接受一个任意的精度参数。

bigdecimal/jacobian

构建给定函数的雅可比（Jacobian）行列式（枚举偏导数的一个矩阵）。不依赖于 `BigDecimal`。

bigdecimal/newton

使用牛顿方法解决非线性函数的根。不依赖于 `BigDecimal`。

bigdecimal/nlsolve

为 `BigDecimal` 方程式包装 `bigdecimal/newton` 库。

可以在 Ruby 源码分发的 `ext/bigdecimal/bigdecimal_en.html` 文件中找到其英文文档。

```
require 'bigdecimal'
require 'bigdecimal/math'
include BigMath

pi = BigMath::PI(20) # 20 is the number of decimal digits
radius = BigDecimal("2.1415987652974674392")
area = pi * radius**2

area.to_s → "0.14408354044685604417672003380667956168
8599846410445032583215824758780405545861
780909930190528E2"

# The same with regular floats
radius = 2.1415987652974674392

Math::PI * radius**2 → 14.4083540446856
```

在一个 Web 服务器中，CGI 类提供了对 CGI (Common Gateway Interface) 脚本程序的支持。CGI 对象使用来自环境的数据和来自 HTTP 请求的数据来初始化，同时它们为访问表单数据和 cookies 提供了便利设施。它们也可以使用各种各样的存储机制管理会话 (session)。CGI 类也提供了生成 HTML 的基本设施，同时提供了类方法对请求和 HTML 进行转义和反向转义。

1.8 注意：CGI 1.8 实现改变了访问表单数据的方式。详细信息请参见 `CGI#[]` 和 `CGI#params` 的 ri 文档。

同时参见：`CGI::Session` (第 661 页)

- 转义和反向转义 URL 和 HTML 中的特殊字符。如果 `$KCODE` 变量被设置为 "u" (为 UTF8)，这个库会把 HTML 的 Unicode 转换成 UTF8。

```
require 'cgi'
CGI.escape('c:\My Files')           →      c%3A%5CMy+Files
CGI.unescape('c%3a%5cMy+Files')     →      c:\My Files
CGI::escapeHTML('"a"<b & c')       →      &quot;a&quot;&lt;b &amp; c

$KCODE = "u" # Use UTF8
CGI.unescapeHTML('&quot;a&quot;&lt;=>b') →  "a"=>b
CGI.unescapeHTML('&#65;&#x41;')        →      AA
CGI.unescapeHTML('&#x3c0;r&#178;')     →      πr2
```

- 访问进入 (incoming) 请求中的信息。

```
require 'cgi'
c = CGI.new
c.auth_type      →      "basic"
c.user_agent     →      "Mozscape Explorari V5.6"
```

- 访问输入请求中的表单字段。假设下面的脚本已经以 `test.cgi` 文件安装在系统中，那么用户可以使用 `http://mydomain.com/test.cgi?fred=10&barney=cat` 链接到它。

```
require 'cgi'
c = CGI.new
c['fred']      →      "10"
c.keys         →      ["barney", "fred"]
c.params       →      {"barney"=>["cat"], "fred"=>["10"]}
```

- 如果表单有多个名字相同的字段，这些字段相应的值会以一个数组返回给这个脚本。`[]` 访问方法只是返回它的第一个值——应当对 `params` 方法的结果进行索引以得到所有的值。在这个例子中，假设这个表单有三个 “name” 字段。

```
require 'cgi'
c = CGI.new
c['name']      →      "fred"
c.params['name'] →      ["fred", "wilma", "barney"]
c.keys         →      ["name"]
c.params       →      {"name"=>["fred", "wilma", "barney"]}
```

- 发送一个响应到浏览器（没有多少人使用这种形式来生成 HTML。请考虑使用其中一个模板化（templating）库——参见第 239 页）。

```
require 'cgi'
cgi = CGI.new("html4Tr")
cgi.header("type" => "text/html", "expires" => Time.now + 30)
cgi.out do
  cgi.html do
    cgi.head{ cgi.title("Hello World!") } +
    cgi.body do
      cgi.pre do
        CGI::escapeHTML(
          "params: " + cgi.params.inspect + "\n" +
          "cookies: " + cgi.cookies.inspect + "\n")
      end
    end
  end
end
```

- 在客户端浏览器中存储 cookie。

```
require 'cgi'
cgi = CGI.new("html4")
cookie = CGI::Cookie.new('name' => 'mycookie',
                        'value' => 'chocolate chip',
                        'expires' => Time.now + 3600)
cgi.out('cookie' => cookie) do
  cgi.head + cgi.body { "Cookie stored" }
end
```

- 得到先前存储的 cookie。

```
require 'cgi'
cgi = CGI.new("html4")
cookie = cgi.cookies['mycookie']
cgi.out('cookie' => cookie) do
  cgi.head + cgi.body { "Flavor: " + cookie[0] }
end
```

CGI::Session 在 CGI 环境中为 Web 用户维护了一个持久的状态。会话 (session) 可能驻留在内存中或者存储在硬盘内。详细的信息请参见第 246 页的讨论。

同时参见: CGI (第 659 页)

```
require 'cgi'
require 'cgi/session'

cgi = CGI.new("html3")
sess = CGI::Session.new(cgi,
                       "session_key" => "rubyweb",
                       "prefix" => "web-session."
                      )

if sess['lastaccess']
  msg = "You were last here #{sess['lastaccess']}."
else
  msg = "Looks like you haven't been here for a while"
end

count = (sess["accesscount"] || 0).to_i
count += 1

msg << "<p>Number of visits: #{count}</p>"

sess["accesscount"] = count
sess["lastaccess"] = Time.now.to_s
sess.close

cgi.out {
  cgi.html {
    cgi.body {
      msg
    }
  }
}
```

Library Complex

复数

Complex 类表示复数。把 Complex 类包括到你的程序中会改变 Numeric 类（及其子类），这让人产生错觉：以为所有的数字都变成了复数（因为它们添加了 real, image, arg, polar, conjugate 和 power! 方法）。

```
require 'complex'
include Math

v1 = Complex(2,3)      → Complex(2, 3)
v2 = 2.im               → Complex(0, 2)
v1 + v2                → Complex(2, 5)
v1 * v2                → Complex(-6, 4)
v2**2                  → Complex(-4, 0)
cos(v1)                → Complex(-4.18962569096881, -9.10922789375534)
v1 < v2                → false
v2**2 == -4            → true

# Euler's theorem
E**(PI*Complex::I)    → Complex(-1.0, 1.22464679914735e-16)
```

以逗号分隔的数据文件，常常用于传递表格式信息（它们是一种通用语 (*lingua franca*)，用来导入和导出电子表格和数据库信息）。

Ruby 的 CSV 库处理数组（对应于 CSV 文件中的一行）和字符串（对应于一行中的元素）。如果某行缺少了一个元素，这个元素在 Ruby 中用 `nil` 表示。

用以下示例中使用的数据文件：

csvfile:

```
12,eggs,2.89,  
2,"shirt, blue",21.45,special  
1,"""Hello Kitty"" bag",13.99
```

csvfile_hdr:

```
Count, Description, Price  
12,eggs,2.89,  
2,"shirt, blue",21.45,special  
1,"""Hello Kitty"" bag",13.99
```

- 读取含有 CSV 数据的文件，一行接一行地处理。

```
require 'csv'  
CSV.open(" csvfile", "r") do |row|  
  qty = row[0].to_i  
  price = row[2].to_f  
  printf "%20s: $%5.2f %s\n", row[1], qty*price, row[3] || "---"  
end
```

输出结果：

```
eggs:    $34.68 ---  
shirt, blue: $42.90 special  
"Hello Kitty" bag: $13.99 ---
```

- 有些 CSV 文件有一个标题 (header) 行。读取它，然后处理文件的剩余部分。

```
require 'csv'  
reader = CSV.open(" csvfile_hdr", "r")  
header = reader.shift  
reader.each {|row| process(header, row) }
```

- 把 CSV 数据写入到一个已打开的流（在这个例子中，它是 `STDOUT`）。把 `|` 作为列分隔符 (column separator)。

```
require 'csv'  
CSV::Writer.generate(STDOUT, '|') do |csv|  
  csv << [ 1, "line 1", 27 ]  
  csv << [ 2, nil, 123 ]  
  csv << [ 3, "|bar|", 32.5 ]  
end
```

输出结果：

```
1|line 1|27  
2||123  
3|"|bar|"|32.5
```

仅当在目标环境中安装有 Curses 或 ncurses 时才适用。

Curses 库是对 C 语言 curses 或者 ncurses 库的一个相当精简的包装，它为程序提供了一种在控制台或其他类终端设备上、与设备无关的绘图 (drawing) 方式。作为对面向对象的推崇，curses 窗口和鼠标事件都是以 Ruby 对象来表示的。另外，在 Curses 模块中简单地定义了标准的 curses 调用和常量。

```
# Draw the paddle of a simple game of 'pong'. It moves
# in response to the up and down keys

require 'curses'
include Curses

class Paddle
  HEIGHT = 4
  PADDLE = " \n" + "| \n" * HEIGHT + " "
  def initialize
    @top = (Curses::lines - HEIGHT)/2
    draw
  end
  def up
    @top -= 1 if @top > 1
  end
  def down
    @top += 1 if (@top + HEIGHT + 1) < lines
  end
  def draw
    setpos(@top-1, 0)
    addstr(PADDLE)
    refresh
  end
end

init_screen
begin
  crmode
  noecho
  stdscr.keypad(true)
  paddle = Paddle.new
  loop do
    case getch
    when ?Q, ?q          : break
    when Key::UP           : paddle.up
    when Key::DOWN         : paddle.down
    else beep
    end
    paddle.draw
  end
ensure
  close_screen
end
```

date 库实现了 Date 和 DateTime 类，它们提供了一组完备的设施来保存、操作和转换那些带时间部分或者不带时间部分的日期。这些类可以表示和操作始于公元前（BCE）4713 年元月 1 日开始的民用日（civil）、年日（ordinal）、历周日（commercial）、儒略日（Julian）和标准日期。DateTime 用小时、分、秒和小数秒（fractional second）扩展了 Date，它提供了对时区的一些支持。这些类也提供了对解析和格式化 date 和 datetime 字符串的支持。这些类的接口很丰富——参考 ri 文档以得到详细的信息。在 lib/date.rb 文件中的介绍性注解也非常值得一读。

同时参见：ParseDate（第 713 页）

- 感受各种各样的日期表示。

```
require 'date'

d = Date.new(2000, 3, 31)      → #<Date:
                                4903269/2,0,2299161>
[d.year, d.yday, d.wday]       → [2000, 91, 5]
[d.month, d.mday]             → [3, 31]
[d.cwyear, d.cweek, d.cwday]   → [2000, 13, 5]
[d.jd, d.mjd]                 → [2451635, 51634]
d1 = Date.commercial(2000, 13, 7) → #<Date:
                                4903273/2,0,2299161>
d1.to_s                        → "2000-04-02"
[d1.cwday, d1.wday]            → [7, 0]
```

- 关于圣诞节的基本信息。

```
require 'date'

now = DateTime.now
year = now.year
year += 1 if now.month == 12 && now.day > 25
xmas = DateTime.new(year, 12, 25)
diff = xmas - now
puts "It's #{diff.to_i} days to Christmas"
h,m,s,frac = Date.day_fraction_to_time(diff)
s += frac.to_f
puts "That's #{h} hours, #{m} minutes, #{s} seconds"
puts "Christmas falls on a #{xmas.strftime('%A')}"
```

输出结果：

```
It's 119 days to Christmas
That's 2876 hours, 12 minutes, 28.0000094433912 seconds
Christmas falls on a Saturday
```

Library

DBM

访问 DBM 数据库的接口

DBM 文件实现了简单的、类似散列表的持久存储。现在已经有很多 DBM 的实现——可以配置这个 Ruby 库来使用任何一种 DBM 库 db、dbm (ndbm)、gdbm 和 qdbm。除了 DBM 的键和值是字符串外，DBM 的文件接口与 Hash 类相似。这可能会引起混淆，因为当写入数据时，数据会悄悄地被转换成字符串。DBM 库是对较低底层访问方法的一个包装。关于真正低级别的对数据库访问，请参见 GDBM 和 SDBM 库。

仅当在目标环境中安装有 DBM 库时才适用。

同时参见：gdbm（第 682 页），sdbm（第 730 页）

- 创建一个简单的 DBM 文件，然后以只读方式重新打开它，并读取一些数据。
请注意 date 对象到其字符串形式的转换。

```
require 'dbm'
require 'date'

DBM.open("data.dbm") do |dbm|
  dbm['name'] = "Walter Wombat"
  dbm['dob'] = Date.new(1997, 12, 25)
end

DBM.open("data.dbm", nil, DBM::READER) do |dbm|
  p dbm.keys
  p dbm['dob']
  p dbm['dob'].class
end
```

输出结果：

```
["name", "dob"]
"1997-12-25"
String
```

- 读取系统的 *aliases* 文件。请注意所有字符串结尾处的 null 字节。

```
require 'dbm'

DBM.open("/etc/aliases", nil) do |dbm|
  p dbm.keys
  p dbm["postfix\000"]
end
```

输出结果：

```
["postmaster\000", "daemon\000", "ftp-bugs\000",
 "operator\000", "abuse\000", "decode\000", "@\000",
 "mailer-daemon\000", "bin\000", "named\000", "nobody\000",
 "uucp\000", "www\000", "postfix\000", "manager\000",
 "dumper\000"]
"root\000"
```

`delegation` 对象是一种在运行时组合对象的方式——使用其他对象的能力来扩展一个对象。`Ruby Delegator` 类实现了一个既简单功能又强大的委托机制 (`delegation scheme`)，它会自动地把请求从主 (`master`) 类转发到这些受托者 (`delegate`) 或者它们的祖先，同时可以使用单个方法调用在运行时改变受托者 (`delegate`)。

同时参见：`Forwardable` (第 680 页)

- 最简单的情况是受托者 (`delegate`) 类是固定不变的，而使主类成为 `DelegateClass` 的一个子类，把要委托类的名称作为参数传入。在主类的 `initialize` 方法中，把要委托的对象传递给超类。

```
require 'delegate'

class Words < DelegateClass(Array)
  def initialize(list = "/usr/share/dict/words")
    words = File.read(list).split
    super(words)
  end
end

words = Words.new
words[9999]           → "anticritique"
words.size            → 234937
words.grep(/matz/)    → ["matzo", "matzoon", "matzos", "matzoth"]
```

- 使用 `SimpleDelegator` 来委托给一个特定的对象（它可以改变）。

```
require 'delegate'

words = File.read("/usr/share/dict/words").split
names = File.read("/usr/share/dict/propernames").split

stats = SimpleDelegator.new(words)
stats.size           → 234937
stats[226]           → "abidingly"
stats.__setobj__(names)
stats.size           → 1323
stats[226]           → "Dave"
```

Library**Digest****MD5, RIPEMD-160 SHA1, and SHA2 Digests**

`Digest` 模块包含许多实现了安全摘要算法的类：MD5、RIPEMD-160、SHA1 和 SHA2（256、384 和 512 位）。所有这些类的接口是一样的。

- 通过调用类方法 `digest` 或 `hexdigest`，可以为给定的字符串创建二进制或者十六进制摘要。
- 也可以创建一个对象（可传入一个可选的初始字符串），同时调用 `digest` 或者 `hexdigest` 实例方法来得到这个对象的散列值（hash）。在这种情况下，你可以使用 `update` 方法把散列值附加到这个字符串中，然后重新获得一个更新的散列值。
- 计算某些 MD5 和 SHA1 散列值。

```
require 'digest/md5'
require 'digest/sha1'

for hash_class in [ Digest::MD5, Digest::SHA1 ]
  puts "Using #{hash_class.name}"
  # Calculate directly
  puts hash_classhexdigest("hello world")
  # Or by accumulating
  digest = hash_class.new
  digest << "hello"
  digest << " "
  digest << "world"
  puts digesthexdigest
  puts
end
```

输出结果：

```
Using Digest::MD5
5eb63bbbe01eeed093cb22bb8f5acdc3
5eb63bbbe01eeed093cb22bb8f5acdc3

Using Digest::SHA1
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

Library**DL****访问动态加载的库 (.dll 和 .so)**

只有当
Windows
或系统支持
动态库时才
适用。

DL 模块用来访问底层操作系统的动态加载功能。在 Windows 系统中，它可以用来访问 DLL 中的函数（替代了 Win32API 类——dl/win32 提供了一个兼容的包装库）。在 Unix 上，它可以加载共享库。因为 Ruby 没有具体类型的方法参数和返回值，你必须通过指定方法的原型特征 (signature)，定义它所期许的类型。这可以通过使用一种类似 C 的语法来完成（如果你使用 dl/import 中的高层方法）或在较底层的 DL 模块级别使用明确的类型指示符。在源码树 ext/dl/doc 目录中提供了很好的文档。

同时参见：Win32API (第 755 页)

- 下面是一个简单的 C 程序，我们要把它构建 (build) 为共享库。

```
#include <stdio.h>
int print_msg(text, number) {
    return printf("Text: %s (%d)\n", text, number);
}
```

- 产生一个代理来访问共享库中的 print_msg 方法。本书使用的方式，共享库放在 code/dl 子目录中；这个目录必须被添加到查找动态对象的搜索目录中。

```
require 'dl'

Message = DL.dlopen("code/dl/lib.so")
print_msg = Message[ "print_msg", "ISI" ]
msg_size, args = print_msg.call("Answer", 42)
puts "Just wrote #{msg_size} bytes"
```

输出结果：

```
Text: Answer (42)
Just wrote 18 bytes
```

- 我们还可以把这个方法包装到一个模块中。这里我们使用环境变量来设置共享对象的路径。这是特定于操作系统的。

```
ENV['DYLD_LIBRARY_PATH'] = ":code/dl" # Mac OS X
require 'dl/import'
module Message
  extend DL::Importable
  dllload "lib.so"
  extern "int print_msg(char *, int)"
end

msg_size = Message.print_msg("Answer", 42)
puts "Just wrote #{msg_size} bytes"
```

输出结果：

```
Text: Answer (42)
Just wrote 18 bytes
```

dRuby 可以让 Ruby 对象跨越网络连接成为分布式对象。虽然还使用客户端与服务器这些术语，但是一旦建立起最初的连接，协议实际上是对称的：每一端都可以调用另一端对象中的方法。通常，远程调用传递参数和返回结果的对象是按值传递的；在对象中包含 DRbUndumped 模块，可以强迫它按引用传递（当实现回调时很有用）。

同时参见：Rinda（第 727 页），XMLRPC（第 757 页）

- 下面的这个服务器程序是可观察的——当计数的值发生改变时，通知所有注册的侦听器。

```
require 'drb'
require 'drb/observer'

class Counter
  include DRb::DRbObservable

  def run
    5.times do |count|
      changed
      notify_observers(count)
    end
  end
end

counter = Counter.new
DRb.start_service('druby://localhost:9001', counter)
DRb.thread.join
```

- 下面的客户端程序和服务器进行交互，在调用服务器的 `run` 方法之前，注册了一个侦听器对象来接受回调。

```
require 'drb'

class Listener
  include DRbUndumped

  def update(value)
    puts value
  end
end

DRb.start_service
counter = DRbObject.new(nil, "druby://localhost:9001")
listener = Listener.new
counter.add_observer(listener)
counter.run
```

在 ruby 脚本中引入 English 库文件后，你可以使用较有含义的名字，来引用例如`$_`的全局变量，如下表所列。

| | | | |
|---------------------|---------------------------------------|----------------------|--|
| <code>\$*</code> | <code>\$ARGV</code> | <code>\$"</code> | <code>\$LOADED_FEATURES</code> |
| <code>\$?</code> | <code>\$CHILD_STATUS</code> | <code>\$&</code> | <code>\$MATCH</code> |
| <code>\$<</code> | <code>\$DEFAULT_INPUT</code> | <code>\$.</code> | <code>\$NR</code> |
| <code>\$></code> | <code>\$DEFAULT_OUTPUT</code> | <code>\$,</code> | <code>\$OFS</code> |
| <code>\$!</code> | <code>\$ERROR_INFO</code> | <code>\$\</code> | <code>\$ORS</code> |
| <code>\$@</code> | <code>\$ERROR_POSITION</code> | <code>\$,</code> | <code>\$OUTPUT_FIELD_SEPARATOR</code> |
| <code>\$;</code> | <code>\$FIELD_SEPARATOR</code> | <code>\$\</code> | <code>\$OUTPUT_RECORD_SEPARATOR</code> |
| <code>\$;</code> | <code>\$FS</code> | <code>\$\$</code> | <code>\$PID</code> |
| <code>\$=</code> | <code>\$IGNORECASE</code> | <code>\$'</code> | <code>\$POSTMATCH</code> |
| <code>\$.</code> | <code>\$INPUT_LINE_NUMBER</code> | <code>\$`</code> | <code>\$PREMATCH</code> |
| <code>\$/</code> | <code>\$INPUT_RECORD_SEPARATOR</code> | <code>\$\$</code> | <code>\$PROCESS_ID</code> |
| <code>\$~</code> | <code>\$LAST_MATCH_INFO</code> | <code>\$0</code> | <code>\$PROGRAM_NAME</code> |
| <code>\$+</code> | <code>\$LAST_PAREN_MATCH</code> | <code>\$/</code> | <code>\$RS</code> |
| <code>\$_</code> | <code>\$LAST_READ_LINE</code> | | |

```
require 'English'
$OUTPUT_FIELD_SEPARATOR = ' -- '
"waterbuffalo" =~ /buff/
print $LOADED_FEATURES, $POSTMATCH, $PID, "\n"
print $", $', $$, "\n"
```

输出结果：

```
English.rb -- alo -- 28035 --
English.rb -- alo -- 28035 --
```

Library Enumerator

定义外部的迭代器

Ruby 的约定是，可枚举的对象必须定义一个名为 `each` 的方法，当每次调用时返回其中的一项。`each` 方法以及内建的 `for` 循环，是 `Enumerable` 模块的基础。即使类定义了多个枚举方法，`Enumerable` 只能使用 `each`。

`Enumerator` 模块基于现有的对象创建一个新的可迭代的对象，将新对象中的 `each` 方法映射到原来对象中的任意方法。这让你可以对任意方法使用标准的 Ruby 枚举技术。

同时参见：`Enumerable`（第 454 页），`Generator`（第 683 页）

- 定义一个外部迭代器返回散列表的所有 `key`。

```
require 'enumerator'

hash = { "cow" => "bovine", "cat" => "feline", "dog" => "canine" }
key_iter = Enumerable::Enumerator.new(hash, :each_key)
puts "Max key is #{key_iter.max}"
for key in key_iter
  puts "Key is #{key}"
end
```

输出结果：

```
Max key is dog
Key is cat
Key is cow
Key is dog
```

- `to_enum` 和 `enum_for` 方法也创建 `Enumerator` 对象。

```
require 'enumerator'

hash = { "cow" => "bovine", "cat" => "feline", "dog" => "canine" }
key_iter = hash.enum_for(:each_key)
key_iter.min      →      "cat"
key_iter.max     →      "dog"
```

- `each_slice` 和 `each_cons` 方法每一次返回 n 个枚举元素。`each_slice` 返回不相交的集合，而 `each_cons` 返回集合一个移动的窗口。

```
require 'enumerator'
(1..7).each_slice(3) {|slice| print slice.inspect, " " }
puts
(1..7).each_cons(3) {|cons| print cons.inspect, " " }
```

输出结果：

```
[1, 2, 3] [4, 5, 6] [7]
[1, 2, 3] [2, 3, 4] [3, 4, 5] [4, 5, 6] [5, 6, 7]
```

ERB 是一个轻量级的模板系统，允许你将 Ruby 代码和纯文本相互混合。通常这是以一种方便的方式来创建 HTML 文档，但是对其他纯文本的情况也非常有用。关于其他模板方案，请参见第 239 页。

ERB 将它的输入文本分解为普通文本和程序段落。然后它建立一个 Ruby 程序，它在运行时输出结果的文本，并执行程序段。程序段用`<%`和`%>`标记包围起来。对这些段落的实际解释，取决于左标记`<%`，如下一页中的表 28.1 所示。

```
require 'erb'
input = %{
<% high.downto(low) do |n| # set high, low externally %>
<%= n %> green bottles, hanging on the wall
<%= n %> green bottles, hanging on the wall
And if one green bottle should accidentally fall
There'd be <%= n-1 %> green bottles, hanging on the wall
<% end %>
}
high,low = 10, 8
erb = ERB.new(input)
erb.run
```

输出结果：

```
10 green bottles, hanging on the wall
10 green bottles, hanging on the wall
And if one green bottle should accidentally fall
There'd be 9 green bottles, hanging on the wall
...
...
```

`ERB.new` 的可选的第二个参数可以设置求解表达式的安全级别。如果为 `nil`，表达式在当前线程中被求解；否则创建一个新线程，并且它的`$SAFE` 级别被设置为参数的值。

`ERB.new` 的可选的第三个参数可以对输入的解释以及向输出添加空格的方式进行一定的控制。如果第三个参数是一个字符串，并且字符串包括一个百分号，那么 ERB 会对百分号开始的行有特别的对待。对待一个由单百分号开头的行，如同它们在`<%...%>`中一样。由双百分号开始的行，会被拷贝到输出中，由一个百分号开头。

```
str = %{
% 2.times do |i|
  This is line <%= i %>      =>
%end
%% done}
ERB.new(str, 0, '%').run
```

输出结果：

```
This is line 0
This is line 1
% done
```

表 28.1 ERB 的指令

| 序 列 | 动 作 |
|-----------------|---|
| <% ruby 代码 %> | 将此处指定的 Ruby 代码插入到产生的程序中。如果有任何的输出，将它包含到结果中 |
| <%= ruby 表达式 %> | 求解表达式，并将它的值插入到产生程序的输出中 |
| <%#...%> | 注释（忽略） |
| <%% 与 %%> | 在输出中分别替换为<%和%> |

如果第三个参数包括字符串<>，而输入行由一个 ERB 指令开头并由%>结尾，那么新折行不会被输出。如果修剪的参数包括>，而输入行以%>结尾，那么新换行不会被输出。

```
str1 = %{\n* <%= "cat" %>\n<%= "dog" %>\n}\nERB.new(str1, 0, ">").run\nERB.new(str1, 0, "<>").run
```

输出结果：

```
* catdog* cat\ndog
```

erb 库还定义了辅助模块 ERB::Util，包括两个方法：html_escape（别名为 h）和 url_encode（别名为 u）。这分别对应 CGI 方法 escapeHTML 和 escape（不同的是，escape 将空格编码为加号，url_encode 则使用%20）。

```
include ERB::Util\nstr1 = %{\n  h(a) = <%= h(a) %>\n  u(a) = <%= u(a) %>\n}\na = "< a & b >"\nERB.new(str1).run
```

输出结果：

```
h(a) = &lt; a & b &gt;\nu(a) = %3C%20a%20%26%20b%20%3E
```

Ruby 发布提供了命令行工具 erb。对输入文件运行 erb 时你可以使用对等的选项；更多细节请参见 erb --help。

仅在 Unix
或 Cygwin
中可用。

Etc 模块提供了许多方法来查询 Unix 系统上 passwd 和 group 的功能。

- 找出当前登录的用户信息

```
require 'etc'

name = Etc.getlogin
info = Etc.getpwnam(name)
info.name      →      "dave"
info.uid       →      502
info.dir       →      "/Users/dave"
info.shell     →      "/bin/bash"

group = Etc.getgrgid(info.gid)
group.name     →      "dave"
```

- 返回编写本书所用系统的所有用户名和组名。

```
require 'etc'

users = []
Etc.passwd {|passwd| users << passwd.name }
users.join(", ") → "nobody, root, daemon, unknown, smmsp, lp,
                     postfix, www, eppc, mysql, sshd, qtss,
                     cyrus, mailman, appserver dave, testuser"

groups = []
Etc.group {|group| groups << group.name }
groups.join(", ") → "nobody, nogroup, wheel, daemon, kmem,
                      sys, tty, operator, mail, bin, staff,
                      smmsp, lp, postfix, postdrop, guest, utmp,
                      uucp, dialer, network, www, mysql, sshd,
                      qtss, mailman, appserverusr, admin,
                      appserveradm, unknown, dave, testuser"
```

Library **expect****IO 对象的 expect 方法**

`expect` 库为所有的 `IO` 对象添加了 `expect` 方法。这让你可以编写代码，来等待 I/O 流中出现了某个特定的字符串或模式。`expect` 方法对 `pty` 对象（参见第 720 页）和访问远程服务器的网络连接来说特别有用，它可以用来协调对外部交互进程的使用。

如果全局变量`$expect_verbose` 为 `true`，`expect` 方法将所有从 I/O 流中读取的字符写入到 `STDOUT`（标准输出）中。

同时参见：`pty`（第 720 页）

- 连接到本地的 FTP 服务器，登录并输出用户的目录名（注意：使用 `net/ftp` 库来编写可能容易得多）。

```
# This code might be specific to the particular
# ftp daemon.

require 'expect'
require 'socket'

$expect_verbose = true
socket = TCPSocket.new('localhost', 'ftp')

socket.expect("ready")
socket.puts("user testuser")
socket.expect("Password required for testuser")
socket.puts("pass secret")
socket.expect("logged in.\r\n")
socket.puts("pwd")
puts(socket.gets)
socket.puts "quit"
```

输出结果：

```
220 localhost FTP server (lukemftpd 1.1) ready.
331 Password required for testuser.
230-
      Welcome to Darwin!
230 User testuser logged in.
257 "/Users/testuser" is the current directory.
```

Fcntl 模块为主机系统上可用的 fcntl 常量（在 fcntl.h 中定义）提供了一个符号名。也就是说，如果主机系统的 fcntl.h 中有一个名为 F_GETLK 的常量，那么 Fcntl 模块将有一个对应的常量 Fcntl::F_GETLK 和头文件中#define 有相同的值。

不同的操作系统有不同的 Fcntl 常量。同一名称的常量在不同的平台上，所对应的值也有可能是不同的。下面是在我 Mac OS X 系统中的值。

```
require 'fcntl'

Fcntl.constants.sort.each do |name|
  printf "%10s: %04x\n", name, Fcntl.const_get(name)
end
```

输出结果：

```
FD_CLOEXEC: 0001
F_DUPFD: 0000
F_GETFD: 0001
F_GETFL: 0003
F_GETLK: 0007
F_RDLCK: 0001
F_SETFD: 0002
F_SETFL: 0004
F_SETLK: 0008
F_SETLKW: 0009
F_UNLCK: 0002
F_WRLCK: 0003
O_ACCMODE: 0003
O_APPEND: 0008
O_CREAT: 0200
O_EXCL: 0800
O_NDELAY: 0004
O_NOCTTY: 0000
O_NONBLOCK: 0004
O_RDONLY: 0000
O_RDWR: 0002
O_TRUNC: 0400
O_WRONLY: 0001
```

Library **FileUtils**

文件和目录操作

`FileUtils` 是操作文件和目录的方法的集合。虽然适用一般的情况，但这个模块在编写安装脚本时特别有用。

许多方法有 `src` 和 `dest` 参数。如果 `dest` 是一个目录，那么 `src` 可能是一个文件名或者一个文件名数组。例如，下面的代码拷贝文件 `a`、`b` 和 `c` 到 `/tmp` 中。

```
cp( %w{ a b c }, "/tmp" )
```

大部分函数有一组选项。可以有零或多个

| 选 项 | 含 义 |
|------------------------|---|
| <code>:verbose</code> | 追踪每个函数的执行（默认输出到 <code>STDERR</code> ，不过可以通过设置类变量 <code>@fileutils_output</code> 来覆盖） |
| <code>:noop</code> | 不要执行函数的动作（对测试脚本很有用处） |
| <code>:force</code> | 覆盖方法的某种比较保守的行为（例如，覆盖一个已有的文件） |
| <code>:preserve</code> | 尝试在 <code>dest</code> 中保留 <code>src</code> 的 <code>atime</code> 、 <code>mtime</code> 和状态信息（ <code>setuid</code> 和 <code>setgid</code> 标志则总会被清除） |

为了最大的可移植性，使用前斜线（`/`）来分割文件名中的目录名，即使在 Windows 上也如此。

`FileUtils` 包括三个子模块重复了顶层的方法，但是有不同的默认选项：模块 `FileUtils::Verbose` 设置了 `verbose` 选项，模块 `FileUtils::NoWrite` 设置了 `noop` 选项，而 `FileUtils::DryRun` 设置了 `verbose` 和 `noop` 选项。
1.8

同时参见：`un`（第 751 页）

```
require 'fileutils'
include FileUtils::Verbose
cd("/tmp") do
  cp("/etc/passwd", "tmp_passwd")
  chmod(0666, "tmp_passwd")
  cp_r("/usr/include/net/", "headers")
  rm("tmp_passwd")      # Tidy up
  rm_rf("headers")
end
```

输出结果：

```
cd /tmp
cp /etc/passwd tmp_passwd
chmod 666 tmp_passwd
cp -r /usr/include/net/ headers
rm tmp_passwd
rm -rf headers
cd -
```

Find 模块支持自顶向下地遍历一组由 `find` 方法的参数所指定的文件路径。如果参数是一个文件，它的名字被传入到 `block` 中。如果它是一个目录，那么它的名字及其所有子目录的名字会被传入。

在 `block` 内，方法 `prune` 可能被调用，它会跳过当前的文件或目录，使用下一个目录重新开始循环。如果当前文件是一个目录，这个目录不会被递归地进入。

```
require 'find'
Find.find("/etc/passwd", "code/cdjukebox") do |f|
  type = case
    when File.file?(f): "F"
    when File.directory?(f): "D"
    else "?"
  end
  puts "#{type}: #{f}"
  Find.prune if f =~ /CVS/
end
```

输出结果：

```
F: /etc/passwd
D: code/cdjukebox
F: code/cdjukebox/Makefile
F: code/cdjukebox/libcdjukebox.a
D: code/cdjukebox/CSV
F: code/cdjukebox/cdjukebox.o
F: code/cdjukebox/cdjukebox.h
F: code/cdjukebox/cdjukebox.c
```

Library **Forwardable**

对象委托

`Forwardable` 提供了一种机制，允许类将某个已命名的方法的调用委托给其他对象。

同时参见：`Delegator`（第 667 页）

- 这个简单的符号表使用了散列表，暴露了散列表的部分方法。

```
require 'forwardable'

class SymbolTable
  extend Forwardable
  def_delegator(:@hash, [], :lookup)
  def_delegator(:@hash, [], :add)
  def_delegators(:@hash, :size, :has_key?)
  def initialize
    @hash = Hash.new
  end
end

st = SymbolTable.new
st.add('cat', 'feline animal') →      "feline animal"
st.add('dog', 'canine animal') →      "canine animal"
st.add('cow', 'bovine animal') →      "bovine animal"
st.has_key?('cow') →                  true
st.lookup('dog') →                  "canine animal"
```

- 通过让对象从 `SingleForwardable` 派生，也可以为个别对象定义转发。虽然很难想象有什么好的原因来使用这一特性。这里有一个无聊的示例……

```
require 'forwardable'
TRICKS = [ "roll over", "play dead" ]
dog = "rover"
dog.extend SingleForwardable
dog.def_delegator(:TRICKS, :each, :can)
dog.can do |trick|
  puts trick
end
```

输出结果：

```
roll over
play dead
```

ftools 库为 File 类添加了一些方法，主要用于移动和拷贝文件的程序，例如安装程序。现在提倡使用 FileUtils 库而非 ftools。

同时参见：fileutils（第 678 页）

- 将 testfile 文件安装到 /tmp 目录中。不用担心目标文件是否存在并且和原来的相同。

```
require 'ftools'

def install_if_different(source, dest)
  if File.exist?(dest) && File.compare(source, dest)
    puts "#{dest} is up to date"
  else
    File.copy(source, dest)
    puts "#{source} copied to #{dest}"
  end
end

install_if_different('testfile', '/tmp/testfile')
puts "Second time..."
install_if_different('testfile', '/tmp/testfile')
puts "Done"
```

输出结果：

```
testfile copied to /tmp/testfile
Second time...
/tmp/testfile is up to date
Done
```

- 使用 FTool 的 install 方法完成同样的工作（日志稍有不同）。

```
require 'ftools'

File.install('testfile', '/tmp', 0644, true)
puts "Second time..."
File.install('testfile', '/tmp', 0644, true)
puts "Done"
```

输出结果：

```
testfile -> /tmp/testfile
chmod 0644 /tmp/testfile
Second time...
Done
```



Library GDBM**GDBM 数据库的接口**

访问 `gdbm` 数据库函数库的接口。³ 虽然 `DBM` 库提供了对 `gdbm` 数据库一般性的访问，但是它并没有暴露出 `gdbm` 完整接口的部分特性。GDBM 库让你可以访问底层的 `gdbm` 特性，例如缓存大小、同步模式、重组和加锁。只有一个进程可以打开 GDBM 数据库进行写入（除非禁止了加锁）。

仅当有
`gdbm` 库时
才适用。

同时参见：DBM（第 666 页），SDBM（第 730 页）

- 将某些值保存到数据库中，然后把它们读取回来。`open` 方法的第二个参数指定了文件的模式，下一个参数使用两个标志，其中（1）指示如果数据库不存在则创建它，（2）强制所有的写操作被同步到磁盘上。在打开时创建是 Ruby `gdbm` 的默认行为。

```
require 'gdbm'
GDBM.open("data.dbm", 0644, GDBM::WRCREAT | GDBM::SYNC) do |dbm|
  dbm['name'] = "Walter Wombat"
  dbm['dob'] = "1969-12-25"
  dbm['uses'] = "Ruby"
end
GDBM.open("data.dbm") do |dbm|
  p dbm.keys
  p dbm['dob']
  dbm.delete('dob')
  p dbm.keys
end
```

输出结果：

```
["uses", "dob", "name"]
"1969-12-25"
["uses", "name"]
```

- 以只读方式打开数据库。注意删除一个 key 的尝试会失败。

```
require 'gdbm'
GDBM.open("data.dbm", 0, GDBM::READER) do |dbm|
  p dbm.keys
  dbm.delete('name')
end
```

输出结果：

```
["uses", "name"]
prog.rb:4:in `delete': Reader can't delete (GDBMError)
from prog.rb:4
from prog.rb:2:in `open'
```

³ <http://www.gnu.org/software/gdbm/gdbm.html>.

基于 Enumerable 对象，或者 yield 值的 block，generator 库实现了外部迭代器（像 Java 和 C++ 那样）。Generator 类是一个简单的迭代器。这个库还包括 SyncEnumerator，它可以在创建一个 Enumerable 对象的同时迭代多个集合。

同时参见：Enumerable（第 454 页），Enumerator（第 672 页）

- 迭代一个 Enumerable 对象。

```
require 'generator'
gen = Generator.new(1..4)
while gen.next?
  print gen.next, "--"
end
```

输出结果：

1--2--3--4--

- 迭代一个 block。

```
require 'generator'
gen = Generator.new do |result|
  result.yield "Start"
  3.times { |i| result.yield i }
  result.yield "done"
end
while gen.next?
  print gen.next, "--"
end
```

输出结果：

Start--0--1--2--done--

- 同时迭代两个集合。

```
require 'generator'
gen = SyncEnumerator.new(1..3, "a".."c")
gen.each { |num, char| print num, "(", char, ")" }
```

输出结果：

1(a) 2(b) 3(c)

Library

GetoptLong

解析命令行选项

`GetoptLong` 类支持 GNU 风格的命令行选项解析。选项可能是一个减号（-）后面跟一个字符，或者两个减号（--）后跟一个名字（长选项）。长选项通常被精炼到最短且无歧义的长度。

内部的一个选项可能有多个外部的表示。例如，控制详细输出的选项可能是下面中的任何一个`-v`、`--verbose` 或者`--details`。某些选项可能还接受相关联的值。

每个内部选项都以一个数组传递给 `GetoptLong`，包括表示选项的外部形式和标志的字符串。标志指定了 `GetoptLong` 如何为这个选项关联参数（`NO_ARGUMENT`、`REQUIRED_ARGUMENT` 或者 `OPTIONAL_ARGUMENT`）。

如果设置了环境变量 `POSIXLY_CORRECT`，所有选项必须在命令行的前部。否则，`GetoptLong` 的默认行为会重组命令行，将选项放到前面。可以通过设置 `GetoptLong#ordering` 为 `PERMUTE`、`REQUIRE_ORDER` 或 `RETRUN_IN_ORDER`，来改变这一行为。`POSIXLY_CORRECT` 可以被覆写。

同时参见：`OptionParser`（第 711 页）

```
# 使用"ruby example.rb --size 10k -v -q a.txt b.doc"来运行该程序
require 'getoptlong'

# specify the options we accept and initialize
# the option parser

opts = GetoptLong.new(
  [ "--size",      "-s",           GetoptLong::REQUIRED_ARGUMENT ],
  [ "--verbose",   "-v",           GetoptLong::NO_ARGUMENT ],
  [ "--query",     "-q",           GetoptLong::NO_ARGUMENT ],
  [ "--check",     "--valid",    "-c", GetoptLong::NO_ARGUMENT ]
)

# 处理解析后的选项
opts.each do |opt, arg|
  puts "Option: #{opt}, arg #{arg.inspect}"
end

puts "Remaining args: #{ARGV.join(' ', '')}"
```

输出结果：

```
Option: --size, arg "10k"
Option: --verbose, arg ""
Option: --query, arg ""
Remaining args: a.txt, b.doc
```

编写 TCP 服务器的一个简单框架。继承 GServer 类，先在构造函数中设置端口号（以及可能的其他参数），然后实现 `serve` 方法来处理进入的请求。

GServer 为进入的请求管理一个线程池，所以你的 `serve` 方法可能并行运行在多个线程中。

你可以在不同的端口运行同一应用的多个 GServer 拷贝。

- 当连接建立在 2000 端口时，用当前时间的字符串作为响应。在三次请求过后终止。

```
require 'gserver'

class TimeServer < GServer
  def initialize
    super(2000)
    @count = 3
  end
  def serve(client)
    client.puts Time.now.to_s
    @count -= 1
    stop if @count.zero?
  end
end

server = TimeServer.new
server.audit = true      # enable logging
server.start
server.join
```

- 你可以通过 telnet 本地的 2000 端口来测试这个服务器。

```
% telnet localhost 2000
```

输出结果：

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
Thu Aug 26 22:38:41 CDT 2004
Connection closed by foreign host.
```

Library **Iconv****字符编码转换**

Iconv 类是访问 Open Group 的 `iconv` 函数库的接口，支持将字符串在不同字符编码间的转换。如果要了解你的平台所支持的编码列表，可以参见系统上 `iconv_open` 的 man page（手册页）⁴。

仅当安装有 Libiconv 时才适用。

Iconv 对象封装了转换描述符，它包括了从一个编码转换到另一个编码所需的信息。转换方法可以多次使用，直至关闭。

`iconv` 的转换方法可以被调用多次，来转换输入的字符串。最后，你应该使用 `nil` 参数来调用它，以清空余下的输出。

- 将 ISO-8859-1 转换到 UTF-16。

```
require 'iconv'

conv = Iconv.new("UTF-16", "ISO-8859-1")
result = conv.iconv("hello")
result << conv.iconv(nil)
result → "\376\377\000h\000e\0001\0001\000o"
```

- 使用类方法完成相同的转换。注意我们使用的 `Iconv.conv` 返回一个字符串，和 `Iconv.iconv` 不同（它返回一个字符串数组）。

```
require 'iconv'
result = Iconv.conv("UTF-16", "ISO-8859-1", "hello")
result → "\376\377\000h\000e\0001\0001\000o"
```

- 将 *olé* 从 UTF-8 转换到 ISO-8859-1。

```
require 'iconv'
result = Iconv.conv("ISO-8859-1", "UTF-8", "ol\303\251")
result → "ol\351"
```

- 将 *olé* 从 UTF-8 转换为 ASCII。这会抛出一个异常，因为 ASCII 没有字符 *é*。

```
require 'iconv'
result = Iconv.conv("ASCII", "UTF-8", "ol\303\251")
```

输出结果：

```
prog.rb:2:in `conv': "\303\251" (Iconv::IllegalSequence)
from prog.rb:2
```

- 这一次，转换为转译的 ASCII，显示所缺少字符的近似字符。

```
require 'iconv'
result = Iconv.iconv("ASCII//TRANSLIT", "UTF-8", "ol\303\251")
result → ["ol'e"]
```

⁴ 译注：在类 UNIX 平台上，可以运行 `iconv -l` 来得到这个列表。

Library IO/Wait

检查是否有未读取的数据

仅在 ioctl(2) 包含库 io/wait，会在标准 IO 类中添加方法 IO#ready? 和 IO#wait。这让我们支持 FIONREAD 可以查询一个打开了流（非文件）的 IO 对象，是否有数据可以被读取，而无须实际读取它；并且可以等待，直至可读取的数据到达一定数量的字节。

- 在两个进程间建立一个管道（pipe），并且每次向其中写入 10 个字节。周期性地查看有多少数据可以读取。

```
require 'io/wait'

reader, writer = IO.pipe

if (pid = fork)
  writer.close
  8.times do
    sleep 0.03
    len = reader.ready?
    puts "Ready? = #{len.inspect}"
    puts(reader.sysread(len)) if len
  end
  Process.waitpid(pid)
else
  reader.close
  5.times do |n|
    sleep 0.04
    writer.write n.to_s * 10
  end
  writer.close
end
```

输出结果：

```
Ready? = 10
0000000000
Ready? = nil
Ready? = 10
1111111111
Ready? = 10
2222222222
Ready? = 10
3333333333
Ready? = nil
Ready? = 10
4444444444
Ready? = nil
```

Library

IPAddr

表示和操作 IP 地址

类 `IPAddr` 可以保存并操作 Internet Protocol (Internet 协议, IP) 地址。每个地址包括三部分：地址、掩码和地址族类（family）。族类通常是 `AF_INET`，表示为 IPv4 和 IPv6 的地址。这个类中的方法包括，提取地址的组成部分、检查 IPv4 的兼容地址（和 IPv4 映射到 IPv6 的地址），测试地址是否位于某个子网，以及许多其他的功能。另外有趣的是，它还包括自己的单元测试数据。

```
require 'ipaddr'

v4 = IPAddr.new('192.168.23.0/24')
v4                         →      #<IPAddr: IPv4:192.168.23.0/ 255.255.255.0>
v4.mask(16)                 →      #<IPAddr: IPv4:192.168.0.0/ 255.255.0.0>
v4.reverse     →      "0.23.168.192.in-addr.arpa"
v6 = IPAddr.new('3ffe:505:2::1')
v6                         →      #<IPAddr:
                                IPv6:3ffe:0505:0002:0000:0000:0000:0001/
                                ffff:ffff:ffff:ffff:ffff:ffff:ffff>
v6.mask(48)                 →      #<IPAddr:
                                IPv6:3ffe:0505:0002:0000:0000:0000:0000/
                                ffff:ffff:ffff:0000:0000:0000:0000>

# the value for 'family' is OS dependent. This
# value is for OS X
v6.family      →      30
other = IPAddr.new("192.168.23.56")
v4.include?(other)   →      true
```

加载 jcode 库扩充了内建的 string 类，使其支持 EUC 和 SJIS 的日文编码以及 UTF8⁵。只有当 \$KCODE 是 EUC、SJIS 或 UTF8 之一时，它才起作用。下列的方法会被更新：chop!、chop、delete!、delete、squeeze!、squeeze、succ!、succ、tr!、tr、tr_s! 和 tr_s。

例如字符串 "\342\210\202x/\342\210\202y" 包括 9 个 8 位的字符。不过，序列 \343\210\202 也可以被解释为一个 UTF-8 字符（数学 delta 符，构成字符串 $\delta x / \delta y$ ）。如果你不告诉 Ruby 它的编码，它将字符串的每个字节当作单独的字符。

- 没有编码支持时，字符串包含字节。

```
$KCODE = "NONE"
require 'jcode'

str = "\342\210\202x/\342\210\202y"
str.length          → 9
str.jlength        → 9
str.jcount("\210") → 2
str.chop!           → "\342\210\202x/\342\210\202"
str.chop!           → "\342\210\202x/\342\210"

str.each_char {|ch| print ch.inspect, " "}
```

输出结果：

```
"\342"  "\210"  "\202"  "x"  "/"  "\342"  "\210"  "\202"  "y"
```

- 不过，当告诉 Ruby 它处理的是 UTF8 字符串时，结果就改变了。

```
$KCODE = 'UTF8'
require 'jcode'

str = "\342\210\202x/\342\210\202y"
str.length          → 9
str.jlength        → 5
str.jcount("\210") → 0
str.chop!           → "\u03b4x/\u03b4"
str.chop!           → "\u03b4x/"

str = "\342\210\202x/\342\210\202y"
str.each_char {|ch| print ch.inspect, " "}
```

输出结果：

```
"\u03b4"  "x"  "/"  "\u03b4"  "y"
```

⁵ 译注：Nikolai Weibull 为 Ruby 编写了一个全新的 UTF-8 库，值得推荐。

Library **Logger**
应用日志

向文件或流中写入日志消息。支持基于时间或大小自动轮换的日志文件。可以指派消息的严重性，只有处于或高于当前日志报告级别的消息才会被记录。

- 当开发时，你可能希望看到所有的消息。

```
require 'logger'
log = Logger.new(STDOUT)
log.level = Logger::DEBUG
log.datetime_format = "%H:%M:%S"
log.info("Application starting")
3.times do |i|
  log.debug("Executing loop, i = #{i}")
  temperature = some_calculation(i) # defined externally
  if temperature > 50
    log.warn("Possible overheat. i = #{i}")
  end
end
log.info("Application terminating")
```

输出结果：

```
I, [09:08:05#11118] INFO -- : Application starting
D, [09:08:05#11118] DEBUG -- : Executing loop, i = 0
D, [09:08:05#11118] DEBUG -- : Executing loop, i = 1
D, [09:08:05#11118] DEBUG -- : Executing loop, i = 2
W, [09:08:05#11118] WARN -- : Possible overheat. i = 2
I, [09:08:05#11118] INFO -- : Application terminating
```

- 在部署时，你可以关闭低于 INFO 的所有消息。

```
require 'logger'
log = Logger.new(STDOUT)
log.level = Logger::INFO
log.datetime_format = "%H:%M:%S"
# as above...
```

输出结果：

```
I, [09:08:05#1120] INFO -- : Application starting
W, [09:08:05#1120] WARN -- : Possible overheat. i = 2
I, [09:08:05#1120] INFO -- : Application terminating
```

- 记录到一个文件，当到达 10k 字节时进行轮换。最多保持 5 个旧文件。

```
require 'logger'
log = Logger.new("application.log", 5, 10*1024)
log.info("Application starting")
# ...
```

Mail 类提供了对 e-mail 消息的基本解析。它可以从指定的文件中读取一个消息，或者重复调用它以从打开的 mbox 格式的文件流中读取消息。每个 Mail 对象表示一个 e-mail 消息，可以分为信头（header）和信体（body）。信体是由行组成的一个数组，而信头是由信头名索引组成的哈希表。Mail 可以正确地连接多行信头。

- 从一个文件中读取一个 e-mail。

```
require 'mailread'

MAILBOX = "/Users/dave/Library/Mail/Mailboxes/Ruby/Talk.mbox/mbox"
msg = Mail.new(MAILBOX)
msg.header.keys → ["Status", "List-software", "Message-id",
                    "Subject", "Received",
                    "X-spambayes-classification",
                    "List-unsubscribe", "Posted",
                    "X-spam-level", "Content-type", "From",
                    "X-virus-scanned", "List-post",
                    "X-spam-status",
                    "Content-transfer-encoding", "X-mlserver",
                    "To", "In-reply-to", "X-ml-info",
                    "X-mail-count", "Date", "List-owner",
                    "X-ml-name", "References", "Reply-to",
                    "Delivered-to", "List-help", "Lines",
                    "Mime-version", "X-spam-checker-version",
                    "List-id", "Precedence"]
msg.body[0] → "On Sat, 14 Aug 2004 03:02:42 +0900, Curt
Hibbs <curt@hibbs.com> wrote:\n"
msg.body[1] → "> We've change the name of the project from
\"Ruby Installer for Windows\" to\n"
msg.body[2] → "> the \"One-Click Ruby Installer\" because
we are branching out more platforms\n"
```

- 从 mbox 格式文件中读取连续的消息。

```
require 'mailread'

MAILBOX = "/Users/dave/Library/Mail/Mailboxes/Ruby/Talk.mbox/mbox"
mbox = File.open(MAILBOX)
count = 0
lines = 0
while !mbox.eof?
  msg = Mail.new(mbox)
  count += 1
  lines += msg.header['Lines'].to_i
end
count → 180
lines → 5927
```

Library **mathn**

统一的数字系统

`mathn` 库尝试为 Ruby 提供统一的数字系统，让 `Bignum`、`Complex`、`Fixnum`、`Integer` 和 `Rational` 类能更好地在一起工作。

- 类型将以一种更自然的方式相互进行转换（例如，`Complex::I` 的平方值为 `-1`，而不是 `Complex[-1, 0]`）。
- 除操作将产生更精确的结果。传统的除操作符（`/`）被重定义为使用 `quo`，不会舍入（`quo` 的文档在第 566 页）。
- 和前面的一点有关，尽可能优先使用有理数，而不是浮点数。用 1 除以 2 得到有理数 `1/2`，而不是 `0.5`（或者 0，一般整数除法的结果）。

同时参见：`Matrix`（第 694 页）、`Rational`（第 721 页）、`Complex`（第 662 页）

- 不使用 `mathn` 的情况。

```
require 'matrix'
require 'complex'

36/16          →      2
Math.sqrt(36/16)   →      1.4142135623731

Complex::I * Complex::I →      Complex(-1, 0)

m = Matrix[[1,2],[3,4]]
i = m.inv

i*m           →       $\begin{pmatrix} 1 & 0 \\ -2 & -2 \end{pmatrix}$ 

(36/16)**-2    →      0.25
(36.0/16.0)**-2 →      0.197530864197531
(-36/16)**-2   →      0.1111111111111111

(36/16)**(1/2)  →      1
(-36/16)**(1/2) →      1

(36/16)**(-1/2) →      0.5
(-36/16)**(-1/2) →      -0.3333333333333333

Matrix.diagonal(6,7,8)/3 →       $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$ 
```

● 使用 mathn:

```

require 'mathn'
require 'matrix'
require 'complex'

36/16          →      9/4
Math.sqrt(36/16)   →      3/2

Complex::I * Complex::I   →      -1

m = Matrix[[1,2],[3,4]]
i = m.inv

i*m           →       $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 

(36/16)**-2      →      16/81
(36.0/16.0)**-2    →      0.197530864197531
(-36/16)**-2      →      16/81

(36/16)**(1/2)      →      3/2
(-36/16)**(1/2)    →      Complex(9.18485099360515e-17, 1.5)

(36/16)**(-1/2)     →      2/3
(-36/16)**(-1/2)   →      Complex(4.08215599715784e-17,
                                -0.666666666666667)

Matrix.diagonal(6,7,8)/3 →       $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 7/3 & 0 \\ 0 & 0 & 8/3 \end{pmatrix}$ 

```

● mathn 库还扩展了数字类来包括新的功能，并添加了一个新的类 Prime。

```

require 'mathn'
primes = Prime.new
3.times { puts primes.succ }
primes.each { |p| puts p; break if p > 20 }

```

输出结果：

```

2
3
5
7
11
13
17
19
23

```

Library Matrix

矩阵和矢量操作

matrix 库定义了类 `Matrix` 和 `Vector`，分别代表矩阵和矢量。和普通的算术操作一样，它们提供了特定于矩阵的函数（例如秩、逆矩阵和行列式）和许多构造方法（创建特例的矩阵——零阶、单位阵、对等阵、奇异阵和向量）。

因为默认的整型算术会截断，整数矩阵的行列式可能被错误地计算，除非你加载 `mathn` 库。

同时参见：`mathn`（第 692 页）、`Rational`（第 721 页）

```
require 'matrix'
require 'mathn'

m1 = Matrix[ [2, 1], [-1, 1] ]           →  ⎛ 2 1 ⎞
                                            ⎝ -1 1 ⎠

m1[0,1]                                →  1

m1.inv                                 →  ⎛ 1/3 -1/3 ⎞
                                            ⎝ 1/3 2/3 ⎠

m1 * m1.inv                            →  ⎛ 1 0 ⎞
                                            ⎝ 0 1 ⎠

m1.determinant                         →  3

m1.singular?                           →  false

m2 = Matrix[ [1,2,3], [4,5,6], [7,8,9] ] →  ⎛ 1 2 3 ⎞
                                            ⎝ 4 5 6 ⎠
                                            ⎝ 7 8 9 ⎠

m2.minor(1, 2, 1, 2)                  →  ⎛ 5 6 ⎞
                                            ⎝ 8 9 ⎠

m2.rank                                →  2

v1 = Vector[3, 4]                      →  Vector[3, 4]

v1.covector                            →  ( 3 4 )

m1 * v1                               →  Vector[10, 1]
```

Monitor 是一种互斥机制的形式，在 1974 年提出。它们可以让不同的线程定义互斥访问的共享资源，并提供了一种可控的机制让某个线程等待获得资源。

monitor 库实际上定义了三种使用 monitor 的不同方式：作为父类、作为 mixin、以及作为某个特定对象的扩展。这三种情况的示例（以及在实践中使用 monitor 的其他代码）在第 142 页的开始。在本节中我们记录了 Monitor 的模块形式。这和类的形式实际上等同的。以类的形式，以及在已有的类中包含 MonitorMixin，本质上都是在类的 initialize 方法中调用 super。

同时参见：Mutex（第 696 页）、Sync（第 738 页）、Thread（第 633 页）

```
require 'monitor'
require 'mathn'

numbers = []
numbers.extend(MonitorMixin)
number_added = numbers.new_cond

# Reporter thread
Thread.new do
  loop do
    numbers.synchronize do
      number_added.wait_while { numbers.empty? }
      puts numbers.shift
    end
  end
end

# Prime number generator thread
generator = Thread.new do
  p = Prime.new
  5.times do
    numbers.synchronize do
      numbers << p.succ
      number_added.signal
    end
  end
end
generator.join
```

输出结果：

```
2
3
5
7
11
```

Library Mutex

线程同步支持

`Mutex` 类可以让线程对某些共享资源进行互斥访问。也就是说，在任何时刻只有一个线程持有锁。其他线程可以选择等待直至锁可以获得，或者立即得到一个错误，表明锁不可获得。这个库还实现了条件变量，让线程当得到一个 `mutex` 时可以放弃控制权，然后在资源变为可获取时重新得到这个锁；还实现了队列，让线程可以安全传递消息。我们在第 135 页开始的第 11 章中介绍了线程，在第 142 页讨论了另一种同步机制 *monitor*。

同时参见：`Monitor`（第 695 页）、`Sync`（第 738 页）、`Queue`（第 743 页）、`Thread`（第 633 页）

```
require 'thread'
class Resource
  attr_reader :left, :times_had_to_wait
  def initialize(count)
    @left = count
    @times_had_to_wait = 0
    @mutex = Mutex.new
    @empty = ConditionVariable.new
  end
  def use
    @mutex.synchronize do
      while @left <= 0
        @times_had_to_wait += 1
        @empty.wait(@mutex)
      end
      @left -= 1
    end
  end
  def release
    @mutex.synchronize do
      @left += 1
      @empty.signal if @left == 1
    end
  end
end

def do_something_with(resource)
  resource.use
  sleep 0.001 # to simulate doing something that takes time
  resource.release
end

resource = Resource.new(2)
user1 = Thread.new { 100.times { do_something_with(resource) } }
user2 = Thread.new { 100.times { do_something_with(resource) } }
user3 = Thread.new { 100.times { do_something_with(resource) } }
user1.join; user2.join; user3.join

resource.times_had_to_wait → 152
```

Library **Mutex_m****Mutex 的 Mix-in**

`mutex_m` 是 `Mutex` 类（在前一页 `thread` 库的文档中介绍）的变体，可让互斥的功能被 mixin 到任何对象。

`Mutex_m` 模块定义了和 `Mutex` 中一一对应的方法，不过都以 `mu_` 开头（因此 `lock` 被定义为 `mu_lock`, 等等）。然后它将这些名称作为原来 `Mutex` 中方法的别名。

同时参见：`Mutex`（第 696 页）、`Sync`（第 738 页）、`Thread`（第 633 页）

```
require 'mutex_m'

class Counter
  include Mutex_m
  attr_reader :count
  def initialize
    @count = 0
    super
  end
  def tick
    lock
    @count += 1
    unlock
  end
end

c = Counter.new

t1 = Thread.new { 10000.times { c.tick } }
t2 = Thread.new { 10000.times { c.tick } }

t1.join
t2.join

c.count → 20000
```



Library Net ::FTP**FTP 客户端**

net/ftp 库实现了一个文件传输协议（File Transfer Protocol, FTP）客户端。除了数据传输命令（getbinaryfile、gettextfile、list、putbinaryfile 和 puttextfile）外，该库还支持完整的服务器命令（acct、chdir、delete、mdtm、mkdir、nlst、rename、rmdir、pwd、size、status 和 system）。支持匿名和基于密码认证的会话。连接可以是主动或被动的。

同时参见：open-uri（第 707 页）

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.netlab.co.jp')
ftp.login
ftp.chdir('pub/lang/ruby/contrib')
files = ftp.list('n*')
ftp.getbinaryfile('nif.rb-0.91.gz', 'nif.gz', 1024)
ftp.close
```

net/http 库提供了一个简单的客户端，使用 HTTP 协议取得一个 HTTP 头信息和 Web 页面的内容。

接口 `get`、`post` 和 `head` 方法从 Ruby 1.6 到 1.8 发生了改变。现在，只返回一个 `response` 对象，它的内容可以通过 `response` 的 `body` 方法来得到。此外，这些方法当遇到可恢复的错误时不再抛出异常。

同时参见： OpenSSL（第 709 页）、open-uri（第 707 页）、URI（第 752 页）

- 打开一个连接，并取得一个页面，显示响应的代码和消息，头信息，以及部分内容体。

```
require 'net/http'
```

```
Net::HTTP.start('www.pragmaticprogrammer.com') do |http|
  response = http.get('/index.html')
  puts "Code = #{response.code}"
  puts "Message = #{response.message}"
  response.each {|key, val| printf "%-14s = %-40.40s\n", key, val }
  p response.body[400, 55]
end
```

输出结果：

```
Code = 200
Message = OK
last-modified      = Fri, 27 Aug 2004 02:25:48 GMT
content-type       = text/html;
etag              = "b00d226-35b4-412e9bac
date              = Fri, 27 Aug 2004 03:38:47 GMT
server             = Rapidsite/Apa/1.3.31 (Unix) FrontPage/5.
content-length    = 13748
accept-ranges     = bytes
"-selling book 'The Pragmatic Programmer' and The\n "
```

- 取得一个页面，显示响应代码和消息，头信息和部分内容体。

```
require 'net/http'
```

```
response = Net::HTTP.get_response('www.pragmaticprogrammer.com',
                                   '/index.html')
puts "Code = #{response.code}"
puts "Message = #{response.message}"
response.each {|key, val| printf "%-14s = %-40.40s\n", key, val }
p response.body[400, 55]
```

输出结果：

```
Code = 200
Message = OK
last-modified      = Fri, 27 Aug 2004 02:25:48 GMT
```

```

content-type      = text/html; charset=iso-8859-1
etag             = "b00d226-608b-43cbbebf"
date             = Tue, 17 Jan 2006 03:47:51 GMT
server           = Rapidsite/Apa/1.3.31 (Unix) FrontPage/5.
content-length   = 24715
accept-ranges    = bytes
"lling book 'The Pragmatic Programmer' and The\n"

```

- 跟进重定向 (`open-uri` 库会自动完成这一点)。代码取自 RDoc 文档。

```

require 'net/http'
require 'uri'

def fetch(uri_str, limit=10)
  fail 'http redirect too deep' if limit.zero?
  puts "Trying: #{uri_str}"
  response = Net::HTTP.get_response(URI.parse(uri_str))
  case response
  when Net::HTTPSuccess
    response
  when Net::HTTPRedirection
    fetch(response['location'], limit-1)
  else
    response.error!
  end
end

response = fetch('http://www.ruby-lang.org')
p response.body[500, 50]

```

输出结果:

```

Trying: http://www.ruby-lang.org
Trying: http://www.ruby-lang.org/en/
"rg\>\n\t<link rel=\\"start\\" title=\\"Top\\" href=\\"..\\">\n\t\n\t"

```

- 通过提交表单数据并读取响应来搜索 Dave 的 blog (不过对我的新 blog 不起作用……)。

```

require 'net/http'

Net::HTTP.start('blogs.pragprog.com') do |query|
  response = query.post("/pragdave", "terms=jolt&handler=searching")
  response.body.scan(%r{<span class="itemtitle">(.*)</span>})m) do
    |title|
    puts title
  end
end

```

输出结果:

```

We're Jolt Finalists
We Got a Jolt Award!

```

Library Net::IMAP
[访问 IMAP 邮件服务器](#)

Internet 邮件访问协议（Internet Mail Access Protocol, IMAP）可以让邮件客户端来访问邮件服务器。它支持明文登录、IMAP 登录和 CRAM-MD5 验证机制。一旦连接完成，这个库则支持多线程，因此在同一时间可以和服务器进行多个交互。

下面的示例取自这个库源文件的 RDoc 文档，并稍加修改。

同时参见：Net::POP（第 702 页）

- 列出 INBOX（收件箱）中发送给“dave”的邮件的发送者和主题。

```
require 'net/imap'

imap = Net::IMAP.new('my.mailserver.com')
imap.authenticate('LOGIN', 'dave', 'secret')
imap.examine('INBOX')
puts "Message count: #{imap.responses['EXISTS']}"
imap.search(["TO", "dave"]).each do |message_id|
  envelope = imap.fetch(message_id, "ENVELOPE")[0].attr["ENVELOPE"]
  puts "#{envelope.from[0].name}: \t#{envelope.subject}"
end
```

- 将 Mail/sent-mail 文件夹中在 2003 年 4 月发送的所有消息移到 Mail/sent-apr03。

```
require 'net/imap'
imap = Net::IMAP.new('my.mailserver.com')
imap.authenticate('LOGIN', 'dave', 'secret')
imap.select('Mail/sent-mail')
if not imap.list('Mail/', 'sent-apr03')
  imap.create('Mail/sent-apr03')
end
imap.search(["BEFORE", "01-May-2003",
            "SINCE", "1-Apr-2003"]).each do |message_id|
  imap.copy(message_id, "Mail/sent-apr03")
  imap.store(message_id, "+FLAGS", [:Deleted])
end
imap.expunge
```



Library Net::POP

访问 POP 邮件服务器

`net/pop` 库提供了一个简单的客户端，从 POP (Post Office Protocol) 服务器上取回或删除邮件。

类 `Net::POP3` 被用来访问一个 POP 服务器，返回 `Net::POPMail` 对象的一个列表，对应保存在服务器上的每个消息。然后可以使用这些 `POPMail` 对象来读取或删除每个消息。

这个库还提供了 `APOP` 类，一个 `POP3` 的替代类，它可以执行验证。

```
require 'net/pop'
pop = Net::POP3.new('server.ruby-stuff.com')
pop.start('joe', 'secret') do |server|
  msg = server.mails[0]
  # Print the 'From:' header line
  from = msg.header.split("\r\n").grep(/^From: /)[0]
  puts from
  puts
  puts "Full message:"
  text = msg.pop
  puts text
end
```

输出结果：

```
From: dave@facet.ruby-stuff.com (Dave Thomas)

Full message:
Return-Path: <dave@facet.ruby-stuff.com>
Received: from facet.ruby-stuff.com (facet.ruby-stuff.com [10.96.0.122])
          by pragprog.com (8.11.6/8.11.6) with ESMTP id i2PJMW701809
          for <joe@carat.ruby-stuff.com>; Thu, 25 Mar 2004 13:22:32 -0600
Received: by facet.ruby-stuff.com (Postfix, from userid 502)
          id 4AF228B1BD; Thu, 25 Mar 2004 13:22:36 -0600 (CST)
To: joe@carat.ruby-stuff.com
Subject: Try out the new features!
Message-Id: <20040325192236.4AF228B1BD@facet.ruby-stuff.com>
Date: Thu, 25 Mar 2004 13:22:36 -0600 (CST)
From: dave@facet.ruby-stuff.com (Dave Thomas)
Status: RO

Ruby 1.8 has a boatload of new features, both in
the core language and in the supplied libraries.

Try it out!
```

net/smtp 库提供了一个简单的客户端，使用简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）来发送电子邮件。它并不帮助邮件的创建——它只是把一个构造好的 RFC822 消息发送出去。

- 从一个字符串发送 e-mail。

```
require 'net/smtp'

msg = "Subject: Test\n\nNow is the time\n"
Net::SMTP.start('pragprog.com') do |smtp|
  smtp.send_message(msg, 'dave@pragprog.com', ['dave'])
end
```

- 使用 SMTP 对象和适配器发送 e-mail。

```
require 'net/smtp'

Net::SMTP::start('pragprog.com', 25, "pragprog.com") do |smtp|
  smtp.open_message_stream('dave@pragprog.com', # from
                           [ 'dave' ]                      # to
                           ) do |stream|
    stream.puts "Subject: Test1"
    stream.puts
    stream.puts "And so is this"
  end
end
```

- 向一个需要 CRAM-MD5 验证的服务器发送 e-mail。

```
require 'net/smtp'

msg = "Subject: Test\n\nNow is the time\n"
Net::SMTP.start('pragprog.com', 25, 'pragprog.com',
               'user', 'password', :cram_md5) do |smtp|
  smtp.send_message(msg, 'dave@pragprog.com', ['dave'])
end
```

Library Net::Telnet**Telnet 客户端**

`net/telnet` 库为 `telnet` 客户端提供了一个完整的实现，并包括其他一些特性使我们与非 `telnet` 服务的交互更方便。

类 `Net::Telnet` 委托给 `Socket` 类。结果，`Socket` 及其父类 `IO` 中的方法在 `Net::Telnet` 对象中都可以调用。

- 连接到 `localhost`，运行 `date` 命令，然后断开。

```
require 'net/telnet'
tn = Net::Telnet.new({})
tn.login "guest", "secret"
tn.cmd "date"      → "date\nThu Aug 26 22:38:56 CDT 2004\n%"
tn.close           → nil
```

- `new`、`cmd`、`login` 和 `waitFor` 可以接收一个可选的 `block`。如果存在，通过这些调用从服务器得到的输出会传入到 `block`。这可以用来提供实时的输出，而无须等待（例如）整个登录完成之后才能显示服务器的响应。

```
require 'net/telnet'
tn = Net::Telnet.new({})          {|str| print str }
tn.login("guest", "secret")       {|str| print str }
tn.cmd("date")                   {|str| print str }
tn.close
```

输出结果：

```
Connected to localhost.
Darwin/BSD (wire less-2.local. thomases.com) (ttyp1)
login: guest
Password:Last login: Thu Aug 26 22:38:56 from localhost
Welcome to Darwin!
% date
Thu Aug 26 22:38:57 CDT 2004
%
```

- 从 NTP 服务器上获得时间。

```
require 'net/telnet'
tn = Net::Telnet.new('Host'      => 'time.nonexistent.org',
                     'Port'      => 'time',
                     'Timeout'   => 60,
                     'Telnetmode'=> false)
atomic_time = tn.recv(4).unpack('N')[0]
puts "Atomic time: " + Time.at(atomic_time - 2208988800).to_s
puts "Local time: " + Time.now.to_s
```

输出结果：

```
Atomic time: Thu Aug 26 22 38:56 CDT 2004
Local time: Thu Aug 26 22 38:59 CDT 2004
```

NKF 模块包装了 Itaru Ichikawa 的 Network Kanji Filter (NKF) 库 (1.7 版本)。它提供了猜测 JIS、EUC 和 SJIS 流的编码，以及将一种编码转换为另一种编码的函数。

- 与解释器使用字符串来表示编码不同的是，NKF 使用整型常量。

```
require 'nkf'
NKF::AUTO → 0
NKF::JIS → 1
NKF::EUC → 2
NKF::SJIS → 3
```

- 猜测一个字符串的编码（感谢 Nobu Nakada 提供本页的示例）。

```
require 'nkf'

NKF.guess("Yukihiro Matsumoto") → 5
NKF.guess("\e$B$^$D$b$H$f$-$R$m\ e(B") → 1
NKF.guess("\244\336\244\304\244\342\244\310\244\346\244\255\244\322\244\355") → 2
NKF.guess("\202\334\202\302\202\340\202\306\202\344\202\253\202\320\202\353") → 3
```

- NKF.nfk 方法接收两个参数。第一个是一组选项，传递给 NKF 库。第二个是要翻译的字符串。下面的示例假定你的控制台可以显示日文字符。三个 ruby 命令之后的文本是 Yukihiro Matsumoto。

```
$ ruby -e 'p *ARGV' まつもと ゆきひろ
"\244\336\244\304\244\342\244\310\244\346\244\255\244\322\244\355"

$ ruby -rnkf -e 'p NKF.nkf(*ARGV)' --Es まつもと ゆきひろ
"\202\334\202\302\202\340\202\306\202\344\202\253\202\320\202\353"

$ ruby -rnkf -e 'p NKF.nkf(*ARGV)' --Ej まつもと ゆきひろ
"\e$B$^$D$b$H$f$-$R$m\ e(B"
```

Library Observable**Observer（观察者）模式**

Observer 模式，也被称为发布/订阅，提供了一种简单的机制，让一个对象（资源）在其状态发生改变时，通知给感兴趣的一组第三方对象（参见《设计模式》（*Design Patterns*），[GHJV95]）。在 Ruby 实现中，处理通知的类 mixin Observable 模块，该模块提供了管理相关联的观察者对象的方法。观察者对象必须实现 update 方法来接收通知。

```
require 'observer'

class CheckWaterTemperature # Periodically check the water
  include Observable

  def run
    last_temp = nil
    loop do
      temp = Temperature.fetch # external class...
      puts "Current temperature: #{temp}"
      if temp != last_temp
        changed                      # notify observers
        notify_observers(Time.now, temp)
        last_temp = temp
      end
    end
  end
end

class Warner
  def initialize(&limit)
    @limit = limit
  end
  def update(time, temp)          # callback for observer
    if @limit.call(temp)
      puts "--- #{time.to_s}: Temperature outside range: #{temp}"
    end
  end
end

checker = CheckWaterTemperature.new
checker.add_observer(Warner.new { |t| t < 80 })
checker.add_observer(Warner.new { |t| t > 120 })
checker.run
```

输出结果：

```
Current temperature: 83
Current temperature: 75
--- Thu Aug 26 22:38:59 CDT 2004: Temperature outside range: 75
Current temperature: 90
Current temperature: 134
--- Thu Aug 26 22:38:59 CDT 2004: Temperature outside range: 134
Current temperature: 134
Current temperature: 112
Current temperature: 79
--- Thu Aug 26 22:38:59 CDT 2004 Temperature outside range: 79
```

Library

open-uri

将 FTP 和 HTTP 资源视为文件

open-uri 库扩展了 Kernel#open，可以让它访问 FTP 或 HTTP 的 URI，像本地的文件名那样。一旦打开完毕，这些资源可以被看作是本地文件，可以使用传统的 IO 方法来访问它们。传递给 open 方法的 URI，可以是一个包括 HTTP 或 FTP URL 的字符串，或是一个 URI 对象（在第 752 页描述）。当打开一个 HTTP 资源时，open 方法会自动处理重定向和代理。当使用 FTP 资源时，方法使用匿名用户来登录。

这种情况下对 open 返回的 IO 对象进行了扩展，以支持若干从请求中返回元信息（meta-information）的方法：content_type、charset、content_encoding、last_modified、status、base_uri 和 meta。

同时参见：URI（第 752 页）

```
require 'open-uri'
require 'pp'

open('http://localhost/index.html') do |f|
  puts "URI: #{f.base_uri}"
  puts "Content-type: #{f.content_type}, charset: #{f.charset}"
  puts "Encoding: #{f.content_encoding}"
  puts "Last modified: #{f.last_modified}"
  puts "Status: #{f.status.inspect}"
  pp f.meta
  puts "----"
  3.times { |i| puts "#{i}: #{f.gets}" }
end
```

输出结果：

```
URI: http://localhost/index.html
Content-type: text/html, charset: iso-8859-1
Encoding:
Last modified: Wed Jul 23:44:21 UTC 2001
Status: ["200", "OK"]
{"vary"=>"negotiate,accept-Language,accept-charset",
 "last-modified"=>"Wed, 18 Jul 2001 23:44:21 GMT",
 "content-location"=>"index.html.en",
 "date"=>"Fri, 27 Aug 2004 03:38:59 GMT",
 "etag"=>"\\"6657-5b0-3b561f55;411edab5\\\"",
 "content-type"=>"text/html",
 "content-language"=>"en",
 "server"=>"Apache/1.3.29 (Darwin) ",
 "content-length"=>"1456",
 "tcn"=>"choice",
 "accept-ranges"=>"bytes"}
----
0: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
1:   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2: <html xmlns="http://www.w3.org/1999/xhtml">
```

Library

Open3

运行子进程并连接所有的流

在子进程中运行一个命令。写入到 *stdin* 中的数据可以被子进程读取，子进程中写入到标准输出和标准错误输出的数据，可以从 *stdout* 和 *stderr* 流中得到。子进程实际上作为一个孙子进程在运行，因此无法使用 `Process#waitall` 来等待它的终止（因此在下面的示例中使用 `sleep`）。

```
require 'open3'

Open3.popen3('bc') do |stdin, stdout, stderr|
  Thread.new { loop { puts "Err stream: #{stderr.gets}" } }
  Thread.new { loop { puts "Output stream: #{stdout.gets}" } }

  stdin.puts "3 * 4"
  stdin.puts "1 / 0"
  stdin.puts "2 ^ 5"
  sleep 0.1
end
```

输出结果：

```
Output stream: 12
Err stream:     Runtime error (func=(main), adr=3): Divide by zero
Output stream: 32
Err stream:
```

Library**OpenSSL****SSL 库**

仅在支持
OpenSSL 库
的系统中可用
(<http://www.openssl.org>)

Ruby 的 OpenSSL 扩展包装了可自由获得的 OpenSSL 函数库。它提供了 Secure Sockets Layer (安全套接字层, SSL) 和 Transport Layer Security (传输层安全, TLS) 协议，支持网络上的安全通信。这个库提供了函数来进行证书的创建和管理、消息签名、以及加密和解密。它还提供封装来简单地访问 https 服务器，以及安全 FTP。这个库的接口非常庞大（大概有 330 个方法），但是一般的 Ruby 用户大概只使用库全部功能的一个小的子集。

同时参见：Net::FTP (第 698 页)、Net::HTTP (第 699 页)、Socket (第 735 页)

- 使用 HTTPS 访问安全的 Web 站点。注意使用 SSL 作为和站点通信的通道，但是所请求的页面还需要标准的 HTTP 基础验证。

```
require 'net/https'
USER = "xxx"
PW   = "yyy"
site = Net::HTTP.new("www.securestuff.com", 443)
site.use_ssl = true
response = site.get2("/cgi-bin/cokerecipe.cgi",
                     'Authorization' => 'Basic ' +
                     ["#{USER}:#{PW}"].pack('m').strip)
```

- 使用 SSL 创建一个套接字 (socket)。这并非是访问 Web 站点的一个好的示范。不过，它演示了套接字是如何加密的。

```
require 'socket'
require 'openssl'
socket = TCPSocket.new("www.secure-stuff.com", 443)
ssl_context = OpenSSL::SSL::SSLContext.new()
unless ssl_context.verify_mode
  warn "warning: peer certificate won't be verified this session."
  ssl_context.verify_mode = OpenSSL::SSL::VERIFY_NONE
end
sslsocket = OpenSSL::SSL::SSLSocket.new(socket, ssl_context)
sslsocket.sync_close = true
sslsocket.connect
sslsocket.puts("GET /secret-info.shtml")
while line = sslsocket.gets
  p line
end
```

Library **OpenStruct**

开放(动态)结构

开放结构是一个对象，它的属性是在第一次赋值时动态创建的。换言之，如果 *obj* 是 OpenStruct 的一个实例，语句 *obj.abc=1* 会在 *obj* 中创建属性 *abc*，然后将值 1 赋值给它。

```
require 'ostruct'

os = OpenStruct.new( "f1" => "one", :f2 => "two" )
os.f3 = "cat"
os.f4 = 99
os.f1      →      "one"
os.f2      →      "two"
os.f3      →      "cat"
os.f4      →      99
```

Library OptionParser

选项解析

OptionParser 是解析命令行参数的一种灵活的和可扩展的方式。它对选项的概念有特别丰富的抽象。

- 选项可以有多个短名称（由一个横线开头的选项）和多个长名称（由两个横线开头）。因此，显示帮助的选项可能是`-h`、`-?`、`--help` 和`--about`。用户可以将长选项精简到最短的无歧义的前缀。
- 选项可以指定为无参数，参数可选，或者必须指定参数。参数可以通过模式或有效值的列表来验证。
- 参数可以被返回为对象或其他任何类型（不仅仅是字符串）。参数类型系统是可扩展的（我们在示例中添加`Data`的处理）。
- 参数可以有一行或多行描述属性文本，在生成用法信息时使用。

选项可以使用`on` 和`def` 方法来指定。这些方法可以接收可变个数的参数，逐步完成对每个选项的定义。这些方法可以接收的参数在下一页的表 28.2 中列出。

同时参见：`GetoptLong`（第 684 页）

```
require 'optparse'
require 'date'

# Add Dates as a new option type
OptionParser.accept(Date, /(\d+)-(\d+)-(\d+)/) do |d, mon, day, year|
  Date.new(year.to_i, mon.to_i, day.to_i)
end

opts = OptionParser.new
opts.on("-x") { |val| puts "-x seen" }
opts.on("-s", "--size VAL", Integer) { |val| puts "-s #{val}" }
opts.on("-a", "--at DATE", Date) { |val| puts "-a #{val}" }

my_argv = [ "--size", "1234", "-x", "-a", "12-25-2003", "fred", "wilma" ]
rest = opts.parse(*my_argv)
puts "Remainder = #{rest.join(' ')}"
puts opts.to_s
```

输出结果：

```
-s 1234
-x seen
-a 2003-12-25
Remainder = fred, wilma
Usage: myprog [options]
      -x
      -s, --size VAL
      -a, --at DATE
```

表 28.2 选项定义参数

"-x" "-xARG" "-x=ARG" "-x[OPT]" "-x[=OPT]" "-x PLACE"

选项具有短名 x。第一种形式没有参数，接下来的两个具有必备参数，再接下来的两个具有可选的参数，最后一个指定了选项后的参数。短名也可以被指定为一个范围（例如“-[a-c]”）。

--switch" "--switch=ARG" "--switch=[OPT]" "--switch PLACE"

选项有长名称 switch。第一种形式没有参数，接下来的两个具有必备参数，再接下来的两个具有可选的参数，最后一个指定了选项后的参数。

--no-switch"

定义一个默认值为 false 的选项。

"=ARG" "= [OPT]"

这个选项的参数是必备的或可选的。例如，下面代码表达的是某个别名为 -x、 -y 和 -z 的选项必须接收一个参数，在用法信息中该参数显示为 N。

```
opt.on("-x", "-y", "-z", "=N")
```

"description"

任何不是由 - 或 = 开始的字符串，可以作为选项的概要描述。可以指定多个描述；它们将显示在其他的行。

/pattern/

所有参数必须匹配给定的模式。

array

参数必须是来自数组的一个值。

proc 或 method

参数类型的转换是通过指定的 proc 或方法来进行的（而不是和 on 或 def 方法调用所关联的 block）。

ClassName

参数必须匹配 ClassName 的定义，可能是预定义或通过 OptionParser.accept 添加的。内建的参数类是：

Object: 任何字符串。无转换。这是默认的。

String: 任何非空的字符串。无转换。

Integer: 类 Ruby/C 的整型，有可选的符号（0ddd 是八进制，0bddd 是二进制，0xdddd 是十六进制）。转换为 Integer。

Float: 浮点数格式。转换为 Float。

Numeric: 一般的数字格式。整型转换为 Integer，浮点数转换为 Float。

Array: 参数必须是由逗号隔开的字符串列表。

OptionParser::DecimalInteger: 十进制整型。转换为 Integer。

OptionParser::OctalInteger: 类 Ruby/C 的八进制/十六进制/二进制整型。

OptionParser::DecimalNumeric: 十进制的整型和浮点数。整数转换为 Integer，浮点数转换为 Float。

TrueClass, FalseClass: 布尔开关。

Library ParseDate

解析日期字符串

ParseDate 模块定义了唯一一个方法 `ParseDate.parsedate`, 将一个日期和/或时间字符串转换为一个 `Fixnum` 值的数组, 表示日期和/或时间的组成成分(年、月、日、时、分、秒、时区和星期)。如果某个字段无法由字符串解析得到, 则返回 `nil`。如果年的值小于 100, 而 `guess` 参数设置为 `true`, `parsedate` 会返回和 `year+2000` 等同的年份, 如果 `year` 小于 69, 返回的年份等同 `year+1900`。

同时参见: `Date` (第 665 页)

| 字符串 | guess | ParseDate::parsedate (string, guess) | | | | | | | | |
|-----------------------------|-------|---------------------------------------|----|----|----|-----|-----|-------|----|--|
| | | yy | mm | dd | hh | min | sec | zone | wd | |
| 1999-09-05 23:55:21+0900 | F | 1999 | 9 | 5 | 23 | 55 | 21 | +0900 | - | |
| 1983-12-25 | F | 1983 | 12 | 25 | - | - | - | - | - | |
| 1965-11-10 T13:45 | F | 1965 | 11 | 10 | 13 | 45 | - | - | - | |
| 10/9/75 1:30pm | F | 75 | 10 | 9 | 13 | 30 | - | - | - | |
| 10/9/75 1:30pm | T | 1975 | 10 | 9 | 13 | 30 | - | - | - | |
| Wed Feb 2 17:15:49 CST 2000 | F | 2000 | 2 | 2 | 17 | 15 | 49 | CST | 3 | |
| Tue, 02-Mar-99 11:20:32 GMT | F | 99 | 3 | 2 | 11 | 20 | 32 | GMT | 2 | |
| Tue, 02-Mar-99 11:20:32 GMT | T | 1999 | 3 | 2 | 11 | 20 | 32 | GMT | 2 | |
| 12-January-1990, 04:00 WET | F | 1990 | 1 | 12 | 4 | 0 | - | WET | - | |
| 4/3/99 | F | 99 | 4 | 3 | - | - | - | - | - | |
| 4/3/99 | T | 1999 | 4 | 3 | - | - | - | - | - | |
| 10th February, 1976 | F | 1976 | 2 | 10 | - | - | - | - | - | |
| March 1st, 84 | T | 1984 | 3 | 1 | - | - | - | - | - | |
| Friday | F | - | - | - | - | - | - | - | 5 | |

Library Pathname

表示文件路径

Pathname 表示一个文件的绝对或相对路径。它有两个不同的用途。首先，它允许操作文件路径的某些部分（提取组成部分、建立新的路径等等）。其次（并且让人有些迷惑），它是 Dir、File 类和 FileTest 模块中某些方法的一个 facade（门面），将对“Pathname 对象命名的文件”的调用转发给它。

同时参见：File（第 465 页）

- 路径名操作。

```
require 'pathname'

p1 = Pathname.new("/usr/bin") → #<Pathname:/usr/bin>
p2 = Pathname.new("ruby")      → #<Pathname:ruby>
p3 = p1 + p2                 → #<Pathname:/usr/bin/ruby>
p4 = p2 + p1                 → #<Pathname:/usr/bin>
p3.parent                      → #<Pathname:/usr/bin>
p3.parent.parent                → #<Pathname:/usr>
p1.absolute?                    → true
p2.absolute?                    → false
p3.split                         → [#<Pathname:/usr/bin>,
                                    <Pathname:ruby>]

p5 = Pathname.new("testdir")   → #<Pathname:testdir>
p5.realpath          → #<Pathname:/Users/dave/Work/rubybook/testdir>
p5.children           → [#<Pathname:testdir/config.h>,
                        #<Pathname:testdir/main.rb>]
```

- Pathname 作为文件和目录状态请求的代理。

```
require 'pathname'

p1 = Pathname.new("/usr/bin/ruby")
p1.file?                      → true
p1.directory?                  → false
p1.executable?                 → true
p1.size                        → 1913444

p2 = Pathname.new("testfile")   → #<Pathname:testfile>
p2.read                         → "This is line one\nThis is
                                line two\nThis is line
                                three\nAnd so on...\n"
p2.readlines                     → ["This is line one\n", "This
                                is line two\n", "This is line
                                three\n", "And so on...\n"]
```

PP 使用 PrettyPrint 库来格式化 Ruby 对象的检视结果。除了一些类方法，它还定义了一个全局方法 pp，该方法和现有的 p 方法类似，但是对输出进行了格式化。

PP 对所有 Ruby 对象有一个默认的布局。不过，你可以通过定义 pretty_print 方法（以一个 PP 对象作为参数）来覆写它对类的处理方式。它应该使用 PP 对象的方法 text、breakable、nest、group 和 pp 来格式化它的输出（细节请参见 PrettyPrint）。

同时参见：PrettyPrint（第 716 页）、YAML（第 758 页）

- 比较“p”和“pp.”。

```
require 'pp'
Customer = Struct.new(:name, :sex, :dob, :country)
cust = Customer.new("Walter Wall", "Male", "12/25/1960", "Niue")
puts "Regular print"
p cust
puts "\nPretty print"
pp cust
```

输出结果：

```
Regular print
#<struct Customer name="Walter Wall", sex="Male", dob="12/25/1960",
country="Niue">

Pretty print
#<struct Customer
name="Walter Wall",
sex="Male",
dob="12/25/1960",
country="Niue">
```

- 你可以告诉 PP 不要显示已经显示的对象。

```
require 'pp'
a = "string"
b = [a]
c = [b, b]
PP.sharing_detection = false
pp c
PP.sharing_detection = true
pp c
```

输出结果：

```
[["string"], ["string"]]
[["string"], [...]]
```

Library PrettyPrint

通用完美打印程序

PrettyPrint 为结构化文本实现了一个完美的打印程序。它处理换行、分组和缩进的细节。PP 库使用 PrettyPrint 来产生更可读的 Ruby 对象的内部信息输出。

同时参见：PP（第 715 页）

- 下面的程序打印 Ruby 类的图表，在父类后显示子类的括号列表。为了节约版面，我们只显示继承树中的 Numeric 分支。

```
require 'prettyprint'
require 'complex'
require 'rational'
@children = Hash.new { |h,k| h[k] = Array.new }
ObjectSpace.each_object(Class) do |cls|
  @children[cls.superclass] << cls if cls <= Numeric
end
def print_children_of(printer, cls)
  printer.text(cls.name)
  kids = @children[cls].sort_by { |k| k.name }
  unless kids.empty?
    printer.group(0, " [", "]") do
      printer.nest(3) do
        printer.breakable
        kids.each_with_index do |k, i|
          printer.breakable unless i.zero?
          print_children_of(printer, k)
        end
      end
      printer.breakable
    end
  end
end
printer = PrettyPrint.new($stdout, 30)
print_children_of(printer, Object)
printer.flush
```

输出结果：

```
Object [
  Numeric [
    Complex
    Float
    Integer [
      Bignum
      Fixnum
    ]
    Rational
  ]
]
```

Library Profile**剖析 Ruby 程序的执行**

`profile` 库是 `Profiler` 模块的一个普通包装，使之可以很容易对整个程序的执行进行剖析。剖析可以使用命令行选项`-rprofile`，或者在源文件中加载 `profile` 模块来启动。

同时参见：Benchmark（第 657 页）、`Profiler_`（第 718 页）

```
require 'profile'
def ackerman(m, n)
  if m == 0 then n+1
  elsif n == 0 and m > 0 then ackerman(m-1, 1)
  else ackerman(m-1, ackerman(m, n-1))
  end
end
ackerman(3, 3)
```

输出结果：

| % | cumulative | self | | self | total | |
|-------|------------|---------|-------|---------|---------|-------------------------|
| time | seconds | seconds | calls | ms/call | ms/call | name |
| 75.14 | 2.75 | 2.75 | 2432 | 1.13 | 46.92 | Object#ackerman |
| 13.39 | 3.24 | 0.49 | 3676 | 0.13 | 0.13 | Fixnum#== |
| 7.65 | 3.52 | 0.28 | 2431 | 0.12 | 0.12 | Fixnum#- |
| 3.83 | 3.66 | 0.14 | 1188 | 0.12 | 0.12 | Fixnum#+ |
| 0.55 | 3.68 | 0.02 | 1 | 20.00 | 20.00 | Profiler_.start_profile |
| 0.00 | 3.68 | 0.00 | 1 | 0.00 | 0.00 | Kernel.puts |
| 0.00 | 3.68 | 0.00 | 1 | 0.00 | 0.00 | Module#method_added |
| 0.00 | 3.68 | 0.00 | 2 | 0.00 | 0.00 | IO#write |
| 0.00 | 3.68 | 0.00 | 57 | 0.00 | 0.00 | Fixnum#> |
| 0.00 | 3.68 | 0.00 | 1 | 0.00 | 3660.00 | #toplevel |

Library Profiler--**控制执行的剖析**

Profiler--模块可以用来收集一个 Ruby 程序中方法的调用次数、所花时间等总结信息。输出按照每个方法所花费的总时间进行排序。profile 库是一个方便的 wrapper，使剖析整个程序更容易。

同时参见：Benchmark（第 657 页）、profile（第 717 页）

```
require 'profiler'
# Omit definition of connection and fetching methods

def calc_discount(qty, price)
  case qty
  when 0..10 then 0.0
  when 11..99 then price * 0.05
  else price * 0.1
  end
end

def calc_sales_totals(rows)
  total_qty = total_price = total_disc = 0
  rows.each do |row|
    total_qty += row.qty
    total_price += row.price
    total_disc += calc_discount(row.qty, row.price)
  end
end

connect_to_database
rows = read_sales_data
Profiler__::start_profile
calc_sales_totals(rows)
Profiler__::stop_profile
Profiler__::print_profile($stdout)
```

输出结果：

| % | cumulative | self | self | total | | |
|-------|------------|---------|-------|---------|---------|--------------------------|
| time | seconds | seconds | calls | ms/call | ms/call | name |
| 31.19 | 0.34 | 0.34 | 1 | 34.00 | 1090.00 | Array#each |
| 20.18 | 0.56 | 0.22 | 648 | 0.34 | 0.80 | Range#== |
| 15.60 | 0.73 | 0.17 | 648 | 0.26 | 0.39 | Fixnum#<=> |
| 10.09 | 0.84 | 0.11 | 324 | 0.34 | 2.01 | Object#calc_discount |
| 6.42 | 0.91 | 0.07 | 648 | 0.11 | 0.11 | Float#coerce |
| 5.50 | 0.97 | 0.06 | 1296 | 0.05 | 0.05 | Float#<=> |
| 3.67 | 1.01 | 0.04 | 969 | 0.04 | 0.04 | Float#+ |
| 2.75 | 1.04 | 0.03 | 648 | 0.05 | 0.05 | S#price |
| 2.75 | 1.07 | 0.03 | 648 | 0.05 | 0.05 | S#qty |
| 1.83 | 1.09 | 0.02 | 324 | 0.06 | 0.06 | Float#* |
| 0.92 | 1.10 | 0.01 | 1 | 10.00 | 10.00 | Profiler__.start_profile |
| 0.00 | 1.10 | 0.00 | 1 | 0.00 | 1090.00 | #toplevel |
| 0.00 | 1.10 | 0.00 | 1 | 0.00 | 1090.00 | Object#calc_sales_totals |
| 0.00 | 1.10 | 0.00 | 3 | 0.00 | 0.00 | Fixnum#+ |

PStore 类为 Ruby 对象提供了事务的、基于文件的持久化存储。每个 PStore 可以存储多个对象层次。每个层次由一个根、一个键（key，通常是一个字符串）来标识。在一个 PStore 事务的开始时，这些对象层次从磁盘文件读出，并可以为 Ruby 程序所使用。当事务结束时，对象层次被写回到文件中。这些对象层次中发生的任何改变都会被保存到磁盘上，并在使用该文件的下一次事务开始时读取出来。

对一般的使用，创建一个 PStore 对象，然后一次或多次使用它来控制一个事务。在事务主体中，以前保存的对象层次如今可以访问了，对对象层次的任何改变，以及任何新的对象层次，都在事务结束时写回到文件中。

- 下面的示例在一个 PStore 中保存了两个对象层次。第一个，由 key "names" 来标识，是一个字符串数组。第二个，由 key "tree" 来标识，是一个简单的二叉树。

```
require 'pstore'
require 'pp'
class T
  def initialize(val, left=nil, right=nil)
    @val, @left, @right = val, left, right
  end
  def to_a
    [ @val, @left.to_a, @right.to_a ]
  end
end

store = PStore.new("/tmp/store")
store.transaction do
  store['names'] = [ 'Douglas', 'Barenberg', 'Meyer' ]
  store['tree'] = T.new('top',
    T.new('A', T.new('B')),
    T.new('C', T.new('D', nil, T.new('E'))))
end

# now read it back in
store.transaction do
  puts "Roots: #{store.roots.join(', ')}"
  puts store['names'].join(', ')
  pp store['tree'].to_a
end
```

输出结果：

```
Roots: names, tree
Douglas, Barenberg, Meyer
["top",
 ["A", ["B", [], []], []],
 ["C", ["D", [], ["E", [], []]], []]]
```

Library **PTY**

伪终端接口：与外部进程交互

许多 UNIX 平台支持伪终端 (*pseudo-terminal*) ——一对设备，一端模拟一个在传统终端运行的进程，另一端可以读写这个终端，好像用户在看着一个屏幕并用键盘输入。

PTY 库提供了 `spawn` 方法，启动一个指定的命令（由默认的 shell 启动），将它连接到伪终端的一端。然后它返回和这个终端连接的读写流，让你的进程可以同运行的程序交互。

使用伪终端可能很有技巧。更方便的方法，请参见第 676 页的 `IO#expect`，它可以让你的生活简单些。你还可以追踪到 Ara T. Howard 的 `Session` 模块，它是一种驱动子进程更简单的方法。⁶

另见：`expect`（第 676 页）。

在子 shell 中运行 `ivb`，并要求它将字符串 “`cat`” 转换为大写。

```
require 'pty'
require 'expect'

$expect_verbose = true
PTY.spawn("ruby /usr/Local/bin/irb") do |reader, writer, pid|
  reader.expect(/irb.*:0> /)
  writer.puts "'cat'.upcase"
  reader.expect("> ")
  answer = reader.gets
  puts "Answer = #{answer}"
end
```

输出结果：

```
irb(main):001:0> 'cat'.upcase
=> Answer = "CAT"
```

⁶ 现在可以从 <http://www.codeforpeople.com/lib/ruby/session/> 找到。

有理数可以表示为两个整数的比率。当分母可以完全除尽分子时，有理数实际是一个整数。有理数能够准确地表示一个分数，但是某些实际值可能无法精确地表达，并且有些无法表示为一个有理数。

通常类 `Rational` 相对独立于其他数字类：两个整数相除的结果通常是一个（被截断的）整型。不过，如果 `mathn` 库被加载到程序中，整数除法将产生一个 `Rational` 的结果。

同时参见：`mathn`（第 692 页）、`Matrix`（第 694 页）、`Complex`（第 662 页）

- 作为独立类的 `Rational`。

```
require 'rational'
r1 = Rational(3, 4)      → Rational(3, 4)
r2 = Rational(2, 3)      → Rational(2, 3)
r1 * 2                  → Rational(3, 2)
r1 * 8                  → Rational(6, 1)
r1 / 6                  → Rational(1, 8)
r1 * r2                → Rational(1, 2)
r1 + r2                → Rational(17, 12)
r1 ** r2               → 0.825481812223657
```

- 使用 `mathn` 后和整数集成的 `Rational`。注意 `mathn` 是如何更改数字字符串表示的。

```
require 'rational'
require 'mathn'
r1 = Rational(3, 4)      → 3/4
r2 = Rational(2, 3)      → 2/3
r1 * 2                  → 3/2
r1 * 8                  → 6
5/3                     → 5/3
5/3 * 6                 → 10
5/3 * 6/15              → 2/3
Math::sin(r1)           → 0.681638760023334
```

Library **readbytes****固定大小的读取**

为类 `IO` 添加了 `readbytes` 方法。这个方法可以保证从一个流中读取固定字节数量的数据，遇到文件结尾时抛出 `EOFError`，或者如果流中剩余的数据少于所请求的大小，则抛出 `TruncatedDataError`。

- 通常，`readbytes` 用于网络连接领域。我们这里使用普通文件来阐明它的使用。

```
require 'readbytes'
File.open("testfile") do |f|
begin
  loop do
    data = f.readbytes(10)
    p data
  end
rescue EOFError
  puts "End of File"
rescue TruncatedDataError => td
  puts "Truncated data: read '#{td.data.inspect}'"
end
end
```

输出结果：

```
"This is li"
"ne one\nThi"
"s is line "
"two\nThis i"
"s line thr"
"ee\nAnd so "
Truncated data: read '"on...\\n"'
```

仅当
GNUreadli
ne 存在时
才适用。

Readline 模块可以让程序为用户输入提供符，并接收所输入的行。模块允许在得到输入行时进行编辑，命令行历史允许回忆和编辑之前的命令。历史可以被搜索，例如，可以让用户回忆起之前包括 *ruby* 字样的命令。命令的自动补齐允许上下文敏感的快捷方式：输入的单词根据所调用的应用，可以在命令行展开。在典型的 GNU 方式中，底层的 readline 库支持远超过一般用户需求的更多选项，它还可以模拟 vi 和 emacs 的键绑定。

- 下面这个无意义的程序实现了一个简单的解释器，可以增加和减小一个值。它使用了 Abbrev 模块（在第 655 页描述），当按下 tab 键时，展开缩写的命令。

```
require 'readline'
include Readline

require 'abbrev'
COMMANDS = %w{ exit inc dec }
ABBREV = COMMANDS.abbrev

Readline.completion_proc = proc do |string|
  ABBREV[string]
end

value = 0
loop do
  cmd = readline("wibble [#{value}]: ", true)
  break if cmd.nil?
  case cmd.strip
  when "exit"
    break
  when "inc"
    value += 1
  when "dec"
    value -= 1
  else
    puts "Invalid command #{cmd}"
  end
end

% ruby code/readline.rb
wibble [0]: inc
wibble [1]: <up-arrow> => inc
wibble [2]: d<tab>      => dec
wibble [1]: ^r i          => inc
wibble [2]: exit
%
```

Library Resolv

DNS 客户端库

`resolv` 库是 DNS 客户端一个纯 Ruby 的实现——可以使用它来将域名转换为相应的 IP 地址。它还支持反查询，以及解析本地 `hosts` 文件中的域名。

`resolv` 库是为了克服和标准操作系统 DNS 查询以及 Ruby 的线程机制交互的一个问题。在大多数操作系统中，名字解析是同步的：你发起了名字查询的调用，当得到地址时调用返回。因为查询通常牵扯网路通信，并且 DNS 服务器可能很慢，这个调用可能需要很长（相对而言）的时间。在这段时间中，发起这个调用的线程实际上被挂起了。因为 Ruby 并不使用操作系统线程，这意味着，如果任何 Ruby 运行的线程执行了一个 DNS 请求，Ruby 解释器实际上被挂起了。有些时候这是无法接受的。转而使用 `resolv` 库。因为它是用 Ruby 编写的，它自动参与到 Ruby 线程机制中，因此，当一个线程进行 DNS 查询时，其他的 Ruby 线程可以照常运行。

加载附加库 `resolv-replace` 会将 `resolv` 库插入到 Ruby 的 `socket` 库中（参见第 735 页）。

- 使用标准的 `socket` 库来查询一个域名。在一个单独线程中运行的计数，当查询发生时被挂起。

```
require 'socket'

count = 0
Thread.critical = true
thread = Thread.new { Thread.pass; loop { count += 1; } }
IPSocket.getaddress("www.ruby-lang.org")      →      "210.163.138.100"
count                                         →      0
```

- 重复这个实验，但是使用 `resolv` 库让 Ruby 线程可以并发运行。

```
require 'socket'
require 'resolv-replace'

count = 0
Thread.critical = true
thread = Thread.new { Thread.pass; loop { count += 1; } }
IPSocket.getaddress("www.ruby-lang.org")      →      "210.163.138.100"
count                                         →      370141
```

REXML 是一个由纯 Ruby 编写的 XML 处理库，包括符合 DTD 的文档解析，XPath 查询，以及文档生成。它支持基于树和基于流的文档处理。因为它是用 Ruby 编写的，所以它在 Ruby 支持的所有平台都可获得。REXML 有完整但是复杂的接口——本节只包括一些小的示例。

- 假定 demo.xml 文件包括：

```
<classes language="ruby">
<class name="Numeric">
  Numeric represents all numbers.
  <class name="Float">
    Floating point numbers have a fraction and a mantissa.
  </class>
  <class name="Integer">
    Integers contain exact integral values.
    <class name="Fixnum">
      Fixnums are stored as machine ints.
    </class>
    <class name="Bignum">
      Bignums store arbitrary-sized integers.
    </class>
  </class>
</classes>
```

- 读取和处理 XML。

```
require'rexml/document'
xml = REXML::Document.new(File.open("demo.xml"))

puts "Root element: #{xml.root.name}"
puts "\nThe names of all classes"
xml.elements.each("//class") {|c| puts c.attributes["name"] }
puts "\nThe description of Fixnum"
p xml.elements["//class[@name='Fixnum']").text
```

输出结果：

```
Root element: classes

The names of all classes
Numeric
Float
Integer
Fixnum
Bignum

The description of Fixnum
"\n      Fixnums are stored as machine ints.\n"
```

- 读入一个文档，在写回之前，添加并删除一些元素，并操作属性。

```

require 'rexml/document'
include REXML

xml = Document.new(File.open("demo.xml"))
cls = Element.new("class")
cls.attributes["name"] = "Rational"
cls.text = "Represents complex numbers"

# Remove Integer's children, and add our new node as
# the one after Integer
int = xml.elements["//class[@name='Integer']"]
int.delete_at(1)
int.delete_at(2)
int.next_sibling = cls

# Change all the 'name' attributes to class_name
xml.elements.each("//class") do |c|
  c.attributes['class_name'] = c.attributes['name']
  c.attributes.delete('name')
end

# and write it out with a XML declaration at the front
xml << XMLDecl.new
xml.write(STDOUT, 0)

```

输出结果：

```

<?xml version='1.0'?>
<classes language='ruby'>
  <class class_name='Numeric'>
    Numeric represents all numbers.
    <class class_name='Float'>
      Floating point numbers have a fraction and a mantissa.
    </class>
    <class class_name='Integer'>
      Integers contain exact integral values.

    </class>
    <class class_name='Rational'>Represents complex numbers</class>
  </class>
</classes>

```

Tuplespace 是一个分布式的黑板系统。进程可能向黑板添加元组，而其他进程可能从黑板上删除匹配某个特定模式的元组。最初由 David Gelernter 提出，tuplespace 为异构进程间的分布式合作提供了一种有趣的方式。

Rinda，是 tuplespace 的 Ruby 实现，并增加了一些有趣的东西。特别地，Rinda 实现使用`==`操作符来匹配元组。这意味着，tuples 可以使用正则表达式、元素的类以及元素的值来进行匹配。

同时参见：DRb（第 670 页）

- 黑板是一个 DRb 服务器，提供共享的 tuplespace。

```
require 'drb/druby'
require 'rinda/tuplespace'
require 'my_uri'      # Defines the constant MY_URI
DRb.start_service(MY_URI, Rinda::TupleSpace.new)
Drb.thread.join
```

- 算术代理接收一个包括算术运算符和两个数字的消息。然后它把结果保存在黑板上。

```
require 'drb/druby'
require 'rinda/rinda'
require 'my_uri'

DRb.start_service
ts = Rinda::TupleSpaceProxy.new(DRbObject.new(nil, MY_URI))
loop do
  op, v1, v2 = ts.take([%r{^[-+/*]}, Numeric, Numeric])
  ts.write(["result", v1.send(op, v2)])
end
```

- 客户端在黑板上放置一系列元组，然后读回每一个的结果。

```
require 'drb/druby'
require 'rinda/rinda'
require 'my_uri'

DRb.start_service
ts = Rinda::TupleSpaceProxy.new(DRbObject.new(nil, MY_URI))
queries = [[ "+", 1, 2 ], [ "*", 3, 4 ], [ "/", 8, 2 ]]
queries.each do |q|
  ts.write(q)
  ans = ts.take(["result", nil])
  puts "#{q[1]} #{q[0]} #{q[2]} = #{ans[1]}"
end
```

Rich (或 RDF) Site Summary、Really Simple Syndication，随你选一个。RSS 是特意用于在 Internet 上传播新闻的一种协议。Ruby 的 RSS 库支持创建和解析符合 RSS 0.9、RSS 1.0 和 RSS 2.0 规范的流。

- 读取并总结 <http://ruby-lang.org> 上近期的新闻。

```
require 'rss/1.0'
require 'open-uri'

open('http://ruby-lang.org/en/index.rdf') do |http|
  response = http.read

  result = RSS::Parser.parse(response, false)
  puts "Channel: " + result.channel.title
  result.items.each_with_index do |item, i|
    puts "#{i+1}. #{item.title}" if i < 4
  end
end
```

输出结果：

```
Channel: Ruby Home Page
1. Brad Cox to Keynote RubyConf 2004
2. Download Ruby
3. RubyConf 2004 registration now open
4. ruby 1.8.2 preview1 released
```

- 生成 RSS 信息。

```
require 'rss/0.9'

rss = RSS::Rss.new("0.9")
chan = RSS::Rss::Channel.new
chan.description = "Dave's Feed"
chan.link = "http://pragprog.com/pragdave"
rss.channel = chan

image = RSS::Rss::Channel::Image.new
image.url = "http://pragprog.com/pragdave.gif"
image.title = "PragDave"
image.link = chan.link
chan.image = image
3.times do |i|
  item = RSS::Rss::Channel::Item.new
  item.title = "My News Number #{i}"
  item.link = "http://pragprog.com/pragdave/story_#{i}"
  item.description = "This is a story about number #{i}"
  chan.items << item
end

puts rss.to_s
```

实现了 C 库 `scanf` 函数的一个 Ruby 版本，按照格式指示符从字符串中提取输入值。

Ruby 版本的库为 `IO` 类和 `String` 类添加了 `scanf` 方法。`IO` 类中的版本，将格式化字符串作用于读取出的下一行。`String` 中的版本，将格式化字符串作用于调用字符串对象本身。这个库还添加了全局方法 `Kernel scanf`，将标准输入的下一行作为格式化源。

使用 `scanf` 来断开一个字符串和使用正则表达式相比有一个主要的好处：正则表达式提取出字符串，而 `scanf` 会将对象转换为正确的类型。

- 将一个日期字符串分解成它的组成对象。

```
require 'scanf'

date = "2004-12-15"
year, month, day = date scanf("%4d-%2d-%2d")
year → 2004
month → 12
day → 15
year.class → Fixnum
```

- `scanf` 的 `block` 形式，可以对输入字符串进行多次格式化，将结果的每个集合返回给 `block`。

```
require 'scanf'
data = "cat:7 dog:9 cow:17 walrus:31"
data scanf("%[^:]:%d") do |animal, value|
  puts "A #{animal.strip} has #{value*1.4}"
end
```

输出结果：

```
A cat has 9.8
A dog has 12.6
A cow has 23.8
A walrus has 43.4
```

- 提取十六进制的数字。

```
require 'scanf'

data = "decaf bad"
data scanf("%3x%2x%x") → [3564, 175, 2989]
```

Library SDBM**访问 SDBM 数据库的接口**

SDBM 数据库实现了一个简单的 key/value 的持久化机制。因为底层的 SDBM 库本身是由 Ruby 提供的，没有外部的依赖，所以 SDBM 在 Ruby 支持的所有平台上都可以获得。SDBM 数据库的 key 和 value 必须是字符串。SDBM 数据库实际上类似于散列表。

同时参见：DBM（第 666 页），GDBM（第 682 页）

- 将记录保存到一个新的数据库中，然后将其取回。和 DBM 库不同，SDBM 的所有值必须是字符串（或者实现了 `to_str`）。

```
require 'sdbm'
require 'date'

SDBM.open("data.dbm") do |dbm|
  dbm['name']      = "Walter Wombat"
  dbm['dob']        = Date.new(1997, 12, 25).to_s
  dbm['uses']       = "Ruby"
end

SDBM.open("data.dbm", nil) do |dbm|
  p dbm.keys
  p dbm['dob']
  p dbm['dob'].class
end
```

输出结果：

```
["name", "dob", "uses"]
"1997-12-25"
String
```

Set 是一个唯一值的集合（唯一性是由 `eql?` 和 `hash` 判定的）。有方便的方法可以让你将 set 绑定到可枚举的对象。

- 基本的 set 操作。

```
require 'set'

set1 = Set.new([:bear, :cat, :deer])

set1.include?(:bat)      →      false
set1.add(:fox)          →      #<Set: {:cat, :deer, :fox, :bear}>

partition = set1.classify{|element| element.to_s.length}

partition              →      {3=>#<Set: {:cat, :fox}>, 4=>#<Set: {:deer, :bear}>}

set2 = [ :cat, :dog, :cow ].to_set
set1 | set2            →      #<Set: {:cat, :dog, :deer, :cow, :fox, :bear}>
set1 & set2            →      #<Set: {:cat}>
set1 - set2            →      #<Set: {:deer, :fox, :bear}>
set1 ^ set2            →      #<Set: {:dog, :deer, :cow, :fox, :bear}>
```

- 将`/etc/passwd`文件中的用户信息隔离到子集中，并且每个子集的成员都具有相邻的用户 ID。

```
require 'etc'
require 'set'

users = []
Etc.passwd{|u| users << u}

related_users = users.to_set.divide do |u1, u2|
  (u1.uid - u2.uid).abs <= 1
end

related_users.each do |relatives|
  relatives.each{|u| print "#{u.uid}/#{u.name} " }
  puts
end
```

输出结果：

```
75/sshd 79/appserver 78/mailmam 77/cyrus 76/qtss 74/mysql
503/testuser 502/dowe
27//postfix 25/smmsp 26/lp
70/www 71/eppc
99/wnknown
-2nobody
1/daemon 0/root
```

Library **Shellwords****使用 POSIX 语义将命令行分解为单词**

给定一个表示 shell 命令行的字符串，依据 POSIX 语义，将它分解到单词词条（token）。

- 双引号或单引号之间的空白符被视为单词的组成部分。
- 双引号可以使用反斜线来转义。
- 由反斜线转义的空白符不会被用以分割单词。
- 否则由空白符分割的词条被视为单词。

```
require 'shellwords'  
include Shellwords  
  
line = %{Code Ruby, Be Happy!}  
shellwords(line) → ["Code", "Ruby,", "Be", "Happy!"]  
  
line = %{"Code Ruby", 'Be Happy'!}  
shellwords(line) → ["Code Ruby,", "Be Happy!"]  
  
line = %q{Code\ Ruby, \"Be Happy\"!}  
shellwords(line) → ["Code Ruby,", "\\"Be", "Happy\"!"]
```

Singleton（单例）设计模式确保某个特定的类，在程序的生命期内只能创建一个唯一的实例（参见 *Design Patterns* [GHJV95]）。

`singleton` 库使得它非常容易实现。将 `Singleton` 模块混入到需要成为单例的类中，而且这个类中的 `new` 方法会修改为私有方法。在它的作用下，这个类的用户可以调用 `instance` 方法，返回这个类的单体实例。

在下面的示例中，`MyClass` 的两个实例是同一个对象。

```
require 'singleton'

class MyClass
  attr_accessor :data
  include Singleton
end

a = MyClass.instance      →      #<MyClass:0x1c20dc>
b = MyClass.instance      →      #<MyClass:0x1c20dc>

a.data = 123              →      123
b.data                  →      123
```

Library SOAP**SOAP 的客户端和服务器端的实现**

SOAP 库实现了 SOAP 协议的客户端和服务器端，包括 WSDL 的支持，Web Services Description Language（Web 服务描述语言）。

关于 SOAP 库更完整的讨论，包括访问 Google 搜索服务的示例，在第 249 页开始。

- 创建一个简单的 SOAP 服务，以字符串返回当前的本地时间。

```
require 'soap/rpc/standaloneServer'
module TimeServant
  def TimeServant.now
    Time.now.to_s
  end
end

class Server < SOAP::RPC::StandaloneServer
  def on_init
    servant = TimeServant
    add_method(servant, 'now')
  end
end

if __FILE__ == $0
  svr = Server.new('Server',
    'http://pragprog.com/TimeServer',
    '0.0.0.0',
    12321)
  trap('INT') { svr.shutdown }
  svr.start
end
```

- 使用一个简单的 SOAP 客户端查询服务器。

```
require 'soap/rpc/driver'
proxy = SOAP::RPC::Driver.new("http://localhost:12321",
  "http://pragprog.com/TimeServer")
proxy.add_method("now")
p proxy.now
```

输出结果：

"Thu Aug 26 22:39:14 CDT 2004"

Library Socket

访问 IP、TCP、UNIX 和 SOCKS 套接字

socket 扩展定义了 9 个类来访问底层系统的 socket 层的通信。所有这些类都（间接）是 IO 类的子类，意味着 IO 类中的方法可以用于 socket 连接。

socket 类的层次反映了网络编程的现实，因此有些令人困惑。BasicSocket 类包括了大部分用于套接字连接的数据传输的通用方法。可以对其子类化来提供特定协议的实现：

IPSocket、UNIXSocket（UNIX 域套接字）和（间接地）TCPSocket、UDPSocket 和 SOCKSSocket。

Socket 也是 BasicSocket 类的子类，Socket 类本身是一个更通用的接口来进行面向套接字的互联。诸如 TCPSocket 这样的类是特定于协议的，而使用 Socket 对象加上一些额外的工作，就可以做到不依赖特定协议。

TCPSocket、SOCKSSocket 和 UNIXSocket 都是面向连接的。每个都有一个对应的 xxxxServer 类，实现连接的服务器方。

你可能永远不会直接使用 socket 库。不过，如果你使用它们，就需要知道很多细节。由于这个原因，我们包含了一个参考章节来涵盖 socket 库的方法，在第 763 页开始的附录 A 中。

下面的代码演示了一个简单的 UDP 服务器和客户端。关于更多的示例，参见附录 A。

```
# Simple logger prints messages
# received on UDP port 12121
require 'socket'
socket = UDPSocket.new
socket.bind("127.0.0.1", 12121)
loop do
  msg, sender = socket.recvfrom(100)
  host = sender[3]
  puts "#{Time.now}: #{host} '#{msg}'"
end
```

```
# Exercise the logger
require 'socket'
log = UDPSocket.new
log.connect("127.0.0.1",
12121)
log.print "Up and Running!"
# process ... process ...
log.print "Done!"
```

输出结果：

```
Wed Jun 30 17:30:24 CDT 2004: 127.0.0.1 'Up and Running!'
Wed Jun 30 17:30:24 CDT 2004: 127.0.0.1 'Done!'
```

Library **StringIO****将字符串视作 IO 对象**

在某种程度上，字符串和文件内容之间的区别是人为的：文件的内容基本上是一个保存在磁盘上（而非内存中）的字符串。StringIO 库的目的是统一这两个概念，让字符串具有打开 IO 对象的行为。一旦字符串被包装成 StringIO 对象之后，你就可以从中读取或向其写入，好像它是一个打开的文件。这可以让单元测试简单得多。你可以把字符串传入到原本编写为操作文件的类和方法中。

- 对字符串进行读写。

```
require 'stringio'

sio = StringIO.new("time flies like an arrow")

sio.read(5)           →      "time "
sio.read(5)           →      "flies"
sio.pos = 18
sio.read(5)           →      " arro"
sio.rewind            →      0
sio.write("fruit")    →      5
sio.pos = 16
sio.write("a banana") →      8
sio.rewind            →      0
sio.read              →      "fruitflies like a banana"
```

- 为测试目的使用 StringIO。

```
require 'stringio'
require 'csv'
require 'test/unit'

class TestCSV < Test::Unit::TestCase
  def test_simple
    StringIO.open do |op|
      CSV::Writer.generate(op) do |csv|
        csv << [ 1, "line 1", 27 ]
        csv << [ 2, nil, 123 ]
      end
      assert_equal("1,line 1,27\n2,,123\n", op.string)
    end
  end
end
```

输出结果：

```
Loaded suite -
Started

.
Finished in 0.001857 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Library StringScanner

基本的字符串分词程序 (Tokenizer)

StringScanner 对象处理一个字符串，匹配（并可能返回）符合给定模式的词条（token）。和内建的扫描方法不同，StringScanner 对象包括字符串当前检验位置的指针，因此，每次调用都沿用上一次调用留下的这个位置。模式匹配定位在上一次的位置。

- 实现一个简单的语言。

```
require 'strscan'

# Handle the language:
#   set <var> = <value>
#   get <var>

values = {}

loop do
  line = gets or break
  scanner = StringScanner.new(line.chomp)
  scanner.scan(/(get|set)\s+/) or fail "Missing command"
  cmd = scanner[1]

  var_name = scanner.scan(/\w+/) or fail "Missing variable"
  case cmd
  when "get"
    puts "#{var_name} => #{values[var_name].inspect}"
  when "set"
    scanner.skip(/\s+=\s+/) or fail "Missing '='"
    value = scanner.rest
    values[var_name] = value
  else
    fail cmd
  end
end
```

输出结果：

```
% ruby code/strscan.rb
set a = dave
set b = hello
get b
b => "hello"
get a
a => "dave"
```

Library Sync

用共享区域进行线程同步

sync 库同步多个并发线程对共享数据的访问。和 Monitor 不同，sync 库支持排他性访问和共享（只读）访问。

同时参见：Monitor（第 695 页）、Mutex（第 696 页）、Thread（第 633 页）

- 无同步时，下面的代码有竞争条件（race condition）：inc 方法可能插入到取得计数和保存递加值的操作之间，结果是更新的结果被丢失了。

```
require 'thwait'

class Counter
  attr_reader :total_count
  def initialize
    @total_count = 0
  end
  def inc
    @total_count += 1
  end
end

count = Counter.new
waiter = ThreadsWait.new([])

# create 10 threads that each inc() 10,000 times
10.times do
  waiter.join_nowait(Thread.new { 10000.times { count.inc } })
end

waiter.all_waits
count.total_count → 62449
```

- 添加排他性同步来保证计数是正确的。

```
require 'thwait'
require 'sync'

class Counter
  attr_reader :total_count
  def initialize
    @total_count = 0
    @sync = Sync.new
  end
  def inc
    @sync.synchronize(:EX) do
      @total_count += 1
    end
  end
end
```

```

count = Counter.new
waiter = ThreadsWait.new([])

# create 10 threads that each inc() 10,000 times
10.times do
  waiter.join_nowait(Thread.new { 10000.times { count.inc } })
end

waiter.all_waits      →      nil
count.total_count     →      100000

```

- 添加共享区确保读取者得到一致的图像。

```

require 'thwait'
require 'sync'

class Counter
  attr_reader :total_count
  def initialize
    @total_count = 0
    @count_down = 0
    @sync = Sync.new
  end
  def inc
    @sync.synchronize(:EX) do
      @total_count += 1
      @count_down -= 1
    end
  end
  def test_consistent
    @sync.synchronize(:SH) do
      fail "Bad counts" unless @total_count + @count_down == 0
    end
  end
end

count = Counter.new
waiter = ThreadsWait.new([])

# create 10 threads that each inc() 10,000 times
10.times do
  waiter.join_nowait(Thread.new { 10000.times do
    count.inc
    count.test_consistent
  end })
end

waiter.all_waits      →      nil
count.total_count     →      100000

```

Library Syslog

访问 Unix 系统日志的接口

Syslog 类是 Unix `syslog(3)` 函数库的一个简单包装。它可以让消息以不同的安全级别写入到日志守护进程中，按照 `syslog.conf` 中的配置传播。下面的示例假定日志文件为 `/var/log/system.log`。

只在 Unix
系统上可
用。

- 添加到我们的本地系统日志。我们将记录所有级别的用户消息（除了调试消息之外）。

```
require 'syslog'
log = Syslog.open("test") # "test" is the app name
log.debug("Warm and fuzzy greetings from your program")
log.info("Program starting")
log.notice("I said 'Hello!'")
log.warning("If you don't respond soon, I'm quitting")
log.err("You haven't responded after %d milliseconds", 7)
log.alert("I'm telling your mother...")
log.emerg("I'm feeling totally crushed")
log.crit("Aarrgh....")
system("tail -7 /var/log/system.log")
```

输出结果：

```
Aug 26 22:39:38 wireless_2 test[28505]: Program starting
Aug 26 22:39:38 wireless_2 test[28505]: I said 'Hello!'
Aug 26 22:39:38 wireless_2 test[28505]: If you don't respond soon, I'm quitting
Aug 26 22:39:38 wireless_2 test[28505]: You haven't responded after 7 milliseconds
Aug 26 22:39:38 wireless_2 test[28505]: I'm telling your mother...
Aug 26 22:39:38 wireless_2 test[28505]: I'm feeling totally crushed Aarrgh....
Aug 26 22:39:38 wireless_2 test[28505]: Aarrgh....
```

- 只记录上面的错误消息。

```
require 'syslog'
log = Syslog.open("test")
log.mask = Syslog::LOG_UPTO(Syslog::LOG_ERR)
log.debug("Warm and fuzzy greetings from your program")
log.info("Program starting")
log.notice("I said 'Hello!'")
log.warning("If you don't respond soon, I'm quitting")
log.err("You haven't responded after %d milliseconds", 7)
log.alert("I'm telling your mother...")
log.emerg("I'm feeling totally crushed")
log.crit("Aarrgh....")
system("tail -7 /var/log/system.log")
```

输出结果：

```
Jan 26 22:39:38 wireless_2 test[28510]: You haven't responded after 7 milliseconds
Jan 26 22:39:38 wireless_2 test[28510]: I'm telling your mother...
Jan 26 22:39:38 wireless_2 test[28510]: I'm feeling totally crushed
Jan 26 22:39:38 wireless_2 test[28510]: Aarrgh....
```

Library Tempfile
临时文件的支持

类 `tempfile` 创建受控的临时文件。虽然它们和其他 `IO` 对象的行为相同，但是当 Ruby 程序结束时会自动删除这些临时文件。当 `Tempfile` 对象被创建时，底层的文件可能多次被打开和关闭。

`Tempfile` 并不直接继承于 `IO`。相反，它将所有调用委托给一个 `File` 对象。从程序员的角度来看，除了不同的 `new`、`open` 和 `close` 语义外，`Tempfile` 对象的行为就像是一个 `IO` 对象。

当你创建它们时如果没有指定保存临时文件的目录，则会使用 `tmpdir` 来得到一个系统相关的位置。

同时参见： `tmpdir`（第 748 页）

```
require 'tempfile'

tf = Tempfile.new("afile")
tf.path                         → "/tmp/afile28519.0"
tf.puts("Così Fan Tutte")        → nil
tf.close                         → nil
tf.open                          → #<File:/tmp/afile28519.0>
tf.gets                          → "Così Fan Tutte\n"
tf.close(true)                  → #<File:/tmp/afile28519.0 (closed)>
```

Library Test::Unit**单元测试框架**

`Test::Unit` 是一个基于原有 `SUnit` Smalltalk 框架的单元测试框架。它提供了用以组织、选取和运行测试的一种结构。测试可以从命令行运行，或使用某个图形界面的接口。

在第 151 页的第 12 章，包括了 `Test::Unit` 的入门教程。

我有一个简单的播放列表类，设计用来保存和获取歌曲。

```
require 'code/testunit/song.rb'
require 'forwardable'

class Playlist
  extend Forwardable
  def_delegator(:@list, :<<, :add_song)
  def_delegator(:@list, :size)
  def initialize
    @list = []
  end

  def find(title)
    @list.find{|song| song.title == title}
  end
end
```

我们可以编写一个单元测试来检验这个类。`Test::Unit` 框架非常聪明，当没有提供主程序时，则运行测试类中的测试项目。

```
require 'test/unit'
require 'code/testunit/playlist.rb'

class TestPlaylist < Test::Unit::TestCase
  def test_adding
    pl = Playlist.new
    assert_equal(0, pl.size)
    assert_nil(pl.find("My Way"))
    pl.add_song(Song.new("My Way", "Sinatra"))
    assert_equal(1, pl.size)
    s = pl.find("My Way")
    assert_not_nil(s)
    assert_equal("Sinatra", s.artist)
    assert_nil(pl.find("Chicago"))
    # .. and so on
  end
end
```

输出结果：

```
Loaded suite -
Started

Finished in 0.002046 seconds.

1 tests, 6 assertions, 0 failures, 0 errors
```

thread 库为支持线程添加了一些实用函数和类。其中大部分已经被 Monitor 类所取代了，不过它包括的两个类 Queue 和 SizedQueue，依然很有用处。这两个类都实现了线程安全的队列，可以用来在多线程环境下的生产者和消费者之间传递对象。Queue 对象实现了一个无大小限制的队列。SizedQueue 有指定的容积；当队列已满时，试图向其中添加对象的生产者会被阻塞，直至有一个消费者取出一个对象。

- 下面的示例由 Robert Kellner 提供。其中有三个消费者从没有大小限制的队列中取得对象。这些对象是由两个生产者提供的，每个添加三项。

```
require 'thread'
queue = Queue.new

consumers = (1..3).map do |i|
  Thread.new("consumer #{i}") do |name|
    begin
      obj = queue.deq
      print "#{name}: consumed #{obj.inspect}\n"
      sleep(rand(0.05))
    end until obj == :END_OF_WORK
  end
end

producers = (1..2).map do |i|
  Thread.new("producer #{i}") do |name|
    3.times do |j|
      sleep(0.1)
      queue.enq("Item #{j} from #{name}")
    end
  end
end

producers.each { |th| th.join}
consumers.size.times { queue.enq(:END_OF_WORK) }
consumers.each { |th| th.join}
```

输出结果：

```
consumer 1: consumed "Item 0 from producer 1"
consumer 2: consumed "Item 0 from producer 2"
consumer 3: consumed "Item 1 from producer 1"
consumer 3: consumed "Item 1 from producer 2"
consumer 3: consumed "Item 2 from producer 2"
consumer 2: consumed "Item 2 from producer 1"
consumer 3: consumed :END_OF_WORK
consumer 2: consumed :END_OF_WORK
consumer 1: consumed :END_OF_WORK
```

Library ThreadsWait

等待多个线程终止

类 `threadsWait` 处理一组线程对象的终止。它提供了若干方法，让你可以检查某个受控的线程是否终止，并等待所有受控的线程终止。

下面的示例启动了许多线程，每个线程等待一小段时间，然后退出并返回它们的线程号。使用 `ThreadsWait`，我们可以在线程退出时单独地或者成组地捕捉到这些线程。

```
require 'thwait'

group = ThreadsWait.new

# construct 10 threads that wait for 1 second, .9 second, etc.
# add each to the group
9.times do |i|
  thread = Thread.new(i) { |index| sleep 1.0 - index/10.0; index }
  group.join_nowait(thread)
end

# any threads finished?
group.finished?          →      false

# wait for one to finish
group.next_wait.value    →      8

# wait for 5 more to finish
5.times { group.next_wait } →      5

# wait for next one to finish
group.next_wait.value    →      2

# and then wait for all the rest
group.all_waits          →      nil
```

Library Time
扩展 Time 类的功能

time 库为内建的 Time 类添加了功能，支持 RFC 2822（邮件）、RFC 2616（HTTP）和 ISO 8601（XML schema 的子集）使用的日期和/或时间格式。

```
require 'time'
Time.rfc2822("Thu, 1 Apr 2004 16:32:45 CST")
→ Thu Apr 01 16:32:45 CST 2004

Time.rfc2822("Thu, 1 Apr 2004 16:32:45 -0600")
→ Thu Apr 01 16:32:45 CST 2004

Time.now.rfc2822
→ Wed,18 May 2005 09:08:37-0500

Time.httpdate("Thu, 01 Apr 2004 16:32:45 GMT")
→ Thu Apr 01 16:32:45 UTC 2004

Time.httpdate("Thursday, 01-Apr-04 16:32:45 GMT")
→ Thu Apr 01 16:32:45 UTC 2004

Time.httpdate("Thu Apr 1 16:32:45 2004")
→ Thu Apr 01 16:32:45 UTC 2004

Time.now.httpdate
→ Wed,18 May 2005 14:08:37 GMT

Time.xmlschema("2004-04-01T16:32:45")
→ Thu Apr 01 16:32:45 CST 2004

Time.xmlschema("2004-04-01T16:32:45.12-06:00")
→ Thu Apr 01 22:32:45 UTC 2004

Time.now.xmlschema
→ 2005-05-18T09:08:3-05:00
```

Library Timeout**使用 Timeout 运行一个 block**

`Timeout.timeout` 方法接收一个表示超时时间段（以秒为单位）的参数和一个可选的异常参数，以及一个 `block`。`block` 和定时器同时运行。如果 `block` 在超时之前结束，则 `timeout` 返回 `block` 的值。否则，抛出异常（默认为 `Timeout::Error`）。

```
require 'timeout'
for snooze in 1..2
  puts "About to sleep for #{snooze}"
  begin
    Timeout::timeout(1.5) do |timeout_length|
      puts "Timeout period is #{timeout_length}"
      sleep(snooze)
      puts "That was refreshing"
    end
  rescue Timeout::Error
    puts "Woken up early!!"
  end
end
```

输出结果：

```
About to sleep for 1
Timeout period is 1.5
That was refreshing
About to sleep for 2
Timeout period is 1.5
Woken up early!!
```

仅当有安装 Tk 库时才适用。

在所有使用 Ruby 创建图形用户界面的选项中，Tk 库可能是支持最广泛的，可以运行在 Windows、Linux、Mac OS X 和其他类 Unix 的平台上。⁷ 虽然它产生的界面并不是最漂亮的，但对程序来说 Tk 是实用并相对简单的。我们在第 255 页的第 19 章中，更完整地记录了 Tk 扩展。

```
require 'tk'
include Math

TkRoot.new do |root|
  title "Curves"
  geometry "400x400"

  TkCanvas.new(root) do |canvas|
    width 400
    height 400
    pack('side'=>'top', 'fill'=>'both', 'expand'=>'yes')

    points = []

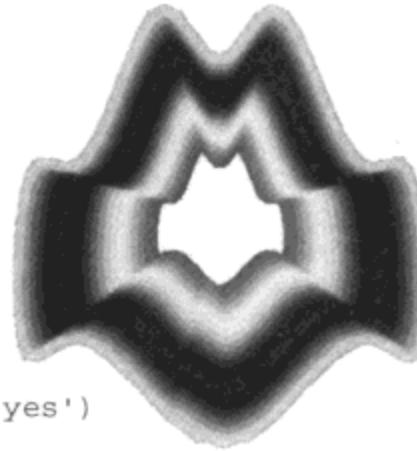
    10.upto(30) do |scale|
      (0.0).step(2*PI, 0.1) do |i|
        new_x = 5*scale*sin(i) + 200 + scale*sin(i*2)
        new_y = 5*scale*cos(i) + 200 + scale*cos(i*6)
        points << [ new_x, new_y ]

        f = scale/5.0
        r = (Math.sin(f)+1)*127.0
        g = (Math.cos(2*f)+1)*127.0
        b = (Math.sin(3*f)+1)*127.0

        col = sprintf("#%02x%02x%02x", r.to_i, g.to_i, b.to_i)

        if points.size == 3
          TkLine.new(canvas,
                     points[0][0], points[0][1],
                     points[1][0], points[1][1],
                     points[2][0], points[2][1],
                     'smooth'=>'on',
                     'width'=> 7,
                     'fill'     => col,
                     'capstyle' => 'round')
          points.shift
        end
      end
    end
  end
end

Tk.mainloop
```



⁷ 不过所有这些环境都需要先安装 Tcl/Tk，才能使用 Ruby 的 Tk 扩展。

Library tmpdir**系统无关的临时目录位置**

tmpdir 库为 Dir 类添加了 tmpdir 方法。这个方法返回一个临时目录的路径，这个路径对当前进程来说应该是可写的（如果没有一个众所周知的临时目录是可写的，并且当前的工作目录也不可写，则这种情况是不可能出现的）。候选的目录包括 TMPDIR、TMP、TEMP 和 USERPROFILE 环境变量所指的目录，/tmp 目录和（在 Windows 机器上）Windows 或 System 目录下的 temp 子目录。

```
require 'tmpdir'

Dir.tmpdir      →      "/tmp"

ENV['TMPDIR'] = "/wibble"  # doesn't exist
ENV['TMP']    = "/sbin"    # not writable
ENV['TEMP']   = "/Users/dave/tmp" # just right

Dir.tmpdir      →      "/Users/dave/tmp"
```

Library Tracer

追踪程序的执行

tracer 库使用 Kernel.set_trace_func 来追踪 Ruby 程序所有部分的执行。追踪得到的输出行显示线程号、文件、行号、类、事件和源代码行。输出的事件中，“-”表示换行，“>”表示调用，“<”表示返回，“C”表示类定义，而“E”表示定义结束。

- 你可以通过从命令中包含 tracer 库来追踪整个程序。

```
class Account
  def initialize(balance)
    @balance = balance
  end
  def debit(amt)
    if @balance < amt
      fail "Insufficient funds"
    else
      @balance -= amt
    end
  end
  acct = Account.new(100)
  acct.debit(40)
```

ruby -r tracer account.rb
 #0:account.rb:1::-- class Account
 #0:account.rb:1:Class:>: class Account
 #0:account.rb:1:Class:<: class Account
 #0:account.rb:1::C: class Account
 #0:account.rb:2::-- def initialize(balance)
 #0:account.rb:2:Module:>: def initialize(balance)
 #0:account.rb:2:Module:<: def initialize(balance)
 #0:account.rb:5::-- def debit(amt)
 #0:account.rb:5:Module:>: def debit(amt)
 #0:account.rb:5:Module:<: def debit(amt)
 #0:account.rb:1::E: class Account
 #0:account.rb:13::-- acct = Account.new(100)
 #0:account.rb:13:Class:>: acct = Account.new(100)
 #0:account.rb:2:Account:>: def initialize(balance)
 #0:account.rb:3:Account:-: @balance = balance
 #0:account.rb:13:Account:<: acct = Account.new(100)
 #0:account.rb:13:Class:<: acct = Account.new(100)
 #0:account.rb:14::-- acct.debit(40)
 #0:account.rb:5:Account:>: def debit(amt)
 #0:account.rb:6:Account:-: if @balance < amt
 #0:account.rb:6:Account:-: if @balance < amt
 #0:account.rb:6:Fixnum:>: if @balance < amt
 #0:account.rb:6:Fixnum:<: if @balance < amt
 #0:account.rb:9:Account:-: @balance -= amt
 #0:account.rb:9:Fixnum:>: @balance -= amt
 #0:account.rb:9:Fixnum:<: @balance -= amt
 #0:account.rb:9:Account:<: @balance -= amt

- 你还可以使用 tracer 对象只追踪代码的某个部分，并使用过滤器来选择要追踪的内容。

```
require 'tracer'
class Account
  def initialize(balance)
    @balance = balance
  end
  def debit(amt)
    if @balance < amt
      fail "Insufficient funds"
    else
      @balance -= amt
    end
  end
  tracer = Tracer.new
  tracer.add_filter lambda {|event, *rest| event == "line" }
  acct = Account.new(100)
  tracer.on do
    acct.debit(40)
  end
```

Library TSort**拓扑 (Topological) 排序**

给定各个节点之间的依赖关系（每个节点依赖 0 或多个其他节点，并且在依赖关系图中没有闭环），拓扑排序将返回一个排好序的节点列表，其中每个节点的后续节点都和该节点没有依赖关系。它的一个用途是调度任务，其中的顺序意味着在你开始一个任务之前，必须先完成它所依赖的任务。`make` 程序使用拓扑排序来规划它的执行步骤。

要使用这个库，你需要 mixin `TSort` 模块，并定义两个方法：`tsort_each_node`，依次返回每个节点；`tsort_each_child`，它在给出一个节点时，返回它所依赖的每个节点。

- 给出完成 piña colada⁸ 步骤间的依赖关系，看看什么是制作它的最佳步骤？

```
require 'tsort'

class Tasks
  include TSort
  def initialize
    @dependencies = {}
  end

  def add_dependency(task, *relies_on)
    @dependencies[task] = relies_on
  end

  def tsort_each_node(&block)
    @dependencies.each_key(&block)
  end

  def tsort_each_child(node, &block)
    deps = @dependencies[node]
    deps.each(&block) if deps
  end
end

tasks = Tasks.new
tasks.add_dependency(:add_rum, :open_blender)
tasks.add_dependency(:add_pc_mix, :open_blender)
tasks.add_dependency(:add_ice, :open_blender)
tasks.add_dependency(:close_blender, :add_rum, :add_pc_mix, :add_ice)
tasks.add_dependency(:blend_mix, :close_blender)
tasks.add_dependency(:pour_drink, :blend_mix)
tasks.add_dependency(:pour_drink, :open_blender)
puts tasks.tsort
```

输出结果：

<code>open_blender</code>	打开搅拌器
<code>add_pc_mix</code>	加入菠萝切片
<code>add_ice</code>	加冰
<code>add_rum</code>	加入朗姆酒
<code>close_blender</code>	扣好搅拌器
<code>blend_mix</code>	搅拌混合
<code>pour_drink</code>	到处饮料

⁸译注：一种有椰子味的酒精饮料。

为什么需要 `un`? 因为当你使用 Ruby 的`-r` 选项从命令行调用它时, 它拼写为`-run`。这个双关语提示你此库的意图: 它让你可以从命令行运行命令 (这种情况下, 是 `FileUtils` 的一个方法子集)。理论上, 这为你提供了与操作系统无关的文件操作命令, 当编写可移植的 `Makefile` 时特别有用。

同时参见: `FileUtils` (第 678 页)

- 可用的命令包括:

```
% ruby -run -e cp -- <options> source dest
% ruby -run -e ln -- <options> target linkname
% ruby -run -e mv -- <options> source dest
% ruby -run -e rm -- <options> file
% ruby -run -e mkdir -- <options> dirs
% ruby -run -e rmdir -- <options> dirs
% ruby -run -e install -- <options> source dest
% ruby -run -e chmod -- <options> octal_mode file
% ruby -run -e touch -- <options> file
```

注意, `--`的使用是告诉 Ruby 解释器, 选项是供程序使用的。

你可以使用下面的命令得到所有可用命令的列表:

```
% ruby -run -e help
```

要得到某个特定命令的帮助, 在后面加上命令的名称:

```
% ruby -run -e help mkdir
```

Library**URI****RFC 2396 统一资源标识符 (Uniform Resource Identifier, URI) 的支持**

URI 代表统一资源标识符 (Uniform Resource Identifier) 的概念，一种指定某种（可能是网络上的）资源的方式。URI 是 URL 的超集：URL（例如 Web 页面的地址）可以按位置指定地址，URI 还可以按名字指定地址。

URI 包括一个模式（例如 `http`、`mailto`、`ftp` 等等），其后的结构化数据指示这种模式下的资源。

URI 有工厂方法，接收一个 URI 字符串，并返回一个对应特定模式的子类。这个库显式地支持 `ftp`、`http`、`https`、`ldap` 和 `mailto` 模式；其他的则被视为普通的 URI。这个模块还有一些便捷的方法，对 URI 进行转义和反转义。`Net::HTTP` 类原本期望接受 URL 参数的地方，也可以接受 URI 对象。

同时参见：`open-uri`（第 707 页）、`Net::HTTP`（第 699 页）

```
require 'uri'

uri = URI.parse("http://pragprog.com:1234/mypage.cgi?q=ruby")
uri.class          → URI::HTTP
uri.scheme         → "http"
uri.host           → "pragprog.com"
uri.port           → 1234
uri.path           → "/mypage.cgi"
uri.query          → "q=ruby"

uri = URI.parse("mailto:ruby@pragprog.com?Subject=help&body=info")
uri.class          → URI::MailTo
uri.scheme         → "mailto"
uri.to             → "ruby@pragprog.com"
uri.headers        → [["Subject", "help"], ["body", "info"]]

uri = URI.parse("ftp://dave@anon.com:/pub/ruby?type=i")
uri.class          → URI::FTP
uri.scheme         → "ftp"
uri.host           → "anon.com"
uri.port           → 21
uri.path           → "/pub/ruby"
uri.typecode       → "i"
```

在 Ruby 中，如果对象还有指向它的引用存在，是无法被垃圾回收的。通常，这是件好事情——如果一个对象在你正使用它时莫名蒸发了，那真是太窘了。不过，有时候你希望多一些灵活性。例如，你想要实现一个位于内存的常用文件内容的缓存。当你读取更多文件时，缓存随之增长。过些时候，你的可用内存数量可能过低了。垃圾回收器会被调用，但是缓存中的对象均被缓存数据结构本身所引用，因此无法被删除。

弱引用的行为和普通的对象引用基本相同，唯一重要的例外是——即使当引用存在时，被引用的对象依然可以被垃圾回收。以缓存的示例来说，如果被缓存的文件是用弱引用访问的，当可用内存过少时，它们会被垃圾回收掉，释放内存让应用程序的其他部分使用。

- 弱引用引入了少许复杂性。因为被引用的对象可以在任何时候被垃圾收集器删除掉，访问这些对象的代码必须小心确保引用是有效的。可以使用两种技术。第一，代码可以像平常一样引用这些对象。任何尝试引用已经被垃圾回收的对象，都会抛出 `WeakRef::RefError` 异常。

```
require 'weakref'
ref = "fol de rol"
puts "Initial object is #{ref}"
ref = WeakRef.new(ref)
puts "Weak reference is #{ref}"
ObjectSpace.garbage_collect
puts "But then it is #{ref}"
```

输出结果：

```
Initial object is fol de rol
Weak reference is fol de rol
prog.rb:8:Illegal Reference-probably recycled(WeakRef::RefError)
```

- 另外一种方法是，使用 `WeakRef#weakref_alive?` 方法来检查引用在使用之前是否是有效的。当测试和后续引用这个对象时，垃圾回收必须被禁止。在一个单线程的程序中，你可以这样使用它：

```
ref = WeakRef.new(some_object)
# ... some time later
gc_was_disabled = GC.disable
if ref.weakref_alive?
  # do stuff with 'ref'
end
GC.enable unless gc_was_disabled
```

Library

WEBrick

Web 服务器工具包

WEBrick 是一个由纯 Ruby 编写的用来实现基于 HTTP 服务器的框架。标准库包括了 WEBrick 服务来实现一个标准的 Web 服务器（支持列出文件和目录的服务），以及若干 servlet 来支持 CGI、erb、文件下载和挂接（mount）Ruby 的 lambda 函数。

在第 247 页开始有更多的 WEBrick 示例。

- 下面的代码在 Web 服务器上挂接了两个 Ruby 的 Proc 对象。请求 URI `http://localhost:2000/hello` 则运行第一个 proc，请求 `http://localhost:2000/bye` 则运行另一个。

```
#!/usr/bin/ruby
require 'webrick'
include WEBrick

hello_proc = lambda do |req, resp|
  resp['Content-Type'] = "text/html"
  resp.body = %{
    <html><body>
      Hello. You're calling from a #{req['User-Agent']}
    <p>
      I see parameters: #{req.query.keys.join(', ')}
    </body></html>
  }
end

bye_proc = lambda do |req, resp|
  resp['Content-Type'] = "text/html"
  resp.body = %{
    <html><body>
      <h3>Goodbye!</h3>
    </body></html>
  }
end

hello      = HTTPServlet::ProcHandler.new(hello_proc)
bye       = HTTPServlet::ProcHandler.new(bye_proc)

s = HTTPServer.new(:Port => 2000)
s.mount("/hello", hello)
s.mount("/bye",   bye)
trap("INT"){ s.shutdown }
s.start
```

Library Win32API**访问 Windows DLL 的入口点**

仅在
Windows
上可用。

Win32API 模块允许访问任意的 Windows 32 函数。许多这类函数接收并返回一个 Pointer 的数据类型——一个对应 C 字符串或结构类型的内存区域。

在 Ruby 中，这些指针是用 String 类表示的，它包括 8 位的一个字节序列。如何将数据位打包到 string 并解包，取决于你。更多细节请参见第 623 页的 unpack 和第 436 页的 pack。

new 调用的第 3、4 个参数指定了所调用方法的参数和返回类型。类型指示符 n 和 l 表示数字，i 表示整型，p 表示指向数据（保存在一个字符串中）的指针，以及 v 表示 void 类型（只用于暴露参数）。这些字符串是大小写不敏感的。方法参数是以字符串数组的形式指定的，并且返回类型是一个字符串。

Win32API 的功能，还可以通过使用 dl/win32 库来提供。因为 DL 库更新一些，意味着原来的 Win32API 可能会随着时间慢慢淡出了。

同时参见：DL（第 669 页）

- 下面的示例来自于 Ruby 的发布，在 ext/Win32API 中。

```
require 'Win32API'

get_cursor_pos = Win32API.new("user32", "GetCursorPos", ['P'], 'V')
lpPoint = " " * 8 # store two LONGs
get_cursor_pos.Call(lpPoint)
x, y = lpPoint.unpack("LL") # get the actual values
print "x: ", x, "\n"
print "y: ", y, "\n"
ods = Win32API.new("kernel32", "OutputDebugString", ['P'], 'V')
ods.Call("Hello, World\n")

GetDesktopWindow = Win32API.new("user32", "GetDesktopWindow", [], 'L')
GetActiveWindow = Win32API.new("user32", "GetActiveWindow", [], 'L')
SendMessage = Win32API.new("user32", "SendMessage", ['L'] * 4, 'L')
SendMessage.Call(GetDesktopWindow.Call, 274, 0xf140, 0)
```

Library **WIN32OLE****Windows 自动化**

访问 Windows 自动化的接口，可以让 Ruby 与 Windows 应用进行交互。Ruby 仅在 Windows 上可用。
访问 Windows 的接口已经在第 267 页的第 20 章详细讨论过了。

同时参见：Win32API（第 755 页）

- 打开 IE 浏览器，要求它显示我们的主页。

```
ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = true
ie.navigate("http://www.pragmaticprogrammer.com")
```

- 在微软的 Excel 表格软件中，创建一个新的图表，然后旋转它。

```
require 'win32ole'
# -4100 is the value for the Excel constant xl3DColumn.
ChartTypeVal = -4100;
excel = WIN32OLE.new("excel.application")
# Create and rotate the chart
excel['Visible'] = TRUE
excel.Workbooks.Add()
excel.Range("a1")['Value'] = 3
excel.Range("a2")['Value'] = 2
excel.Range("a3")['Value'] = 1
excel.Range("a1:a3").Select()
excelchart = excel.Charts.Add()
excelchart['Type'] = ChartTypeVal
30.step(180, 5) do |rot|
  excelchart.rotation = rot
  sleep(0.1)
end
excel.ActiveWorkbook.Close(0)
excel.Quit()
```

XMLRPC 库可以让客户端使用 XML-RPC 协议，调用网路另一端服务器上的方法。通信使用的是 HTTP 协议。服务器可能运行在一个 Web 服务器的环境中，这种情况通常使用 80 或 443（适用于 SSL）端口。服务器也可能是单独运行的。Ruby 的 XML-RPC 服务器实现支持以下运作方式：作为 CGI 脚本、mod_ruby 脚本、WEBrick 的处理方法、以及单独的服务器。支持基本的认证，并且客户端可以通过代理（proxy）和服务器通信。服务器可能会抛出 `FaultException` 错误——这些错误会在客户端产生相应的异常（或者作为调用返回的状态标志）。

同时参见：SOAP（第 249 页）、dRuby（第 670 页）、WEBrick（第 754 页）

- 下面这个简单的服务器，接收摄氏度并将它转换为华氏度。它运行于 WEBrick Web 服务器的环境中。

```
require 'webrick'
require 'xmlrpc/server'

xml_servlet = XMLRPC::WEBrickServlet.new
xml_servlet.add_handler("convert_celcius") do |celcius|
  celcius*1.8 + 32
end

xml_servlet.add_multicall # Add support for multicall
server = WEBrick::HTTPServer.new(:Port => 2000)
server.mount("/RPC2", xml_servlet)
trap("INT"){ server.shutdown }
server.start
```

- 客户端可以调用温度转换服务器。注意我们在结果中显示了服务器的日志以及客户程序的输出。

```
require 'xmlrpc/client'

server = XMLRPC::Client.new("localhost", "/RPC2", 2000)
puts server.call("convert_celcius", 0)
puts server.call("convert_celcius", 100)
puts server.multicall(['convert_celcius', -10],
                      ['convert_celcius', 200])
```

输出结果：

```
[2004-04-16 06:57:02] INFO WEBrick 1.3.1
[2004-04-16 06:57:02] INFO WEBrick::HTTPServer#start: pid=11956 port=2000
localhost - - [16/Apr/2004:06:57:13 PDT] "POST /RPC2 HTTP/1.1" 200 124 - -> /RPC2
32.0
localhost - - [16/Apr/2004:06:57:13 PDT] "POST /RPC2 HTTP/1.1" 200 125 - -> /RPC2
212.0
localhost - - [16/Apr/2004:06:57:14 PDT] "POST /RPC2 HTTP/1.1" 200 290 - -> /RPC2
14.0
392.0
```

Library **YAML**

对象序列化和反序列化

YAML 库（在第 416 页的教程中也有描述）可以将 Ruby 对象树序列化为一个外部的、可读的、纯文本格式文件，或从这个文件进行反序列化。YAML 可以用作一种可移植的对象列集（marshaling）方案，可以让对象以纯文本的方式在两个隔离的 Ruby 进程间传递。某些情况下，对象也可以在 Ruby 程序和其他语言编写的、同样支持 YAML 的程序之间进行交换。

- YAML 可以用来将对象树保存在一个扁平的文件中。

```
require 'yaml'
tree = { :name => 'ruby',
         :uses => [ 'scripting', 'web', 'testing', 'etc' ]
       }
File.open("tree.yaml", "w") {|f| YAML.dump(tree, f)}
```

- 在保存之后，它可以被另一个程序读取。

```
require 'yaml'
tree = YAML.load_file("tree.yaml")
tree[:uses][1] → "web"
```

- YAML 格式也是为程序保存配置信息的一种方便的方式。因为它是可读的，可以使用常见的编辑器来维护它，并且让程序以对象的形式读取出来。例如，一个配置文件可能包括：

```
---
username: dave
prefs:
  background: dark
  foreground: cyan
  timeout: 30
```

我们可以在一个程序中这样使用它：

```
require 'yaml'

config = YAML.load_file("code/config.yaml")
config["username"] → "dave"
config["prefs"]["timeout"] * 10 → 300
```

Library**Zlib****读写压缩的文件**

仅在支持 zlib 函数库的系统上可用。

Zlib 模块是处理压缩和解压缩流的 Ruby 类、以及处理 gzip 格式压缩文件的 Ruby 类的家园。它们还可以计算 zip 的校验和。

- 将/etc/passwd 压缩为一个 gzip 文件，并将结果读取回来。

```
require 'zlib'

# These methods can take a filename
Zlib::GzipWriter.open("passwd.gz") do |gz|
  gz.write(File.read("/etc/passwd"))
end

system("ls -l /etc/passwd passwd.gz")

# or a stream
File.open("passwd.gz") do |f|
  gzip = Zlib::GzipReader.new(f)
  data = gzip.read.split(/\n/)
  puts data[15,3]
end
```

输出结果：

```
-rw-r--r-- 1 root wheel 1374 8 Dec 2003 /etc/passwd
-rw-r--r-- 1 dave dave 635 18 May 09:08 passwd.gz
daemon:*:1:1:System Services:/var/root:/usr/bin/false
smmsp:*:25:25:Sendmail User:/private/etc/mail:/usr/bin/false
lp:*:26:26:Printing Services:/var/spool/cups:/usr/bin/false
```

- 压缩在两个进程间发送的数据。

```
require 'zlib'

rd, wr = IO.pipe

if fork
  rd.close
  zipper = Zlib::Deflate.new
  zipper << "This is a string "
  data = zipper.deflate("to compress", Zlib::FINISH)
  wr.write(data)
  wr.close
  Process.wait
else
  wr.close
  unzipper = Zlib::Inflate.new
  unzipper << rd.read
  puts "We got: #{unzipper.inflate(nil)}"
end
```

输出结果：

```
We got: This is a string to compress
```

第5部分 附录

Part V Appendixes



Programming Ruby 中文版, 第 2 版

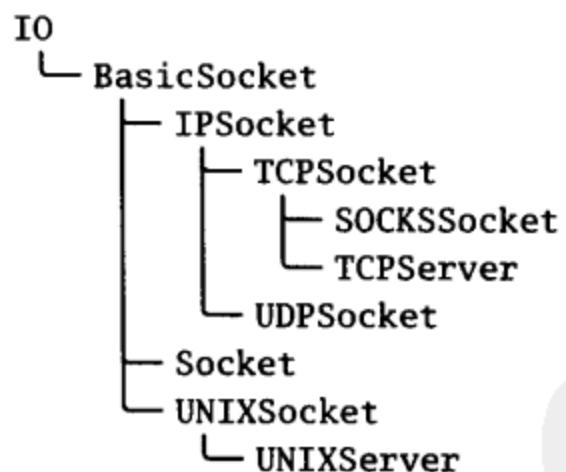
附录 A

Socket 库

Socket Library

因为 socket 和 network 库对于将 Ruby 应用集成到网络是如此重要的部分，所以我们决定比其他标准库更细致地来介绍它们。

socket 类的层次结构如下图所示。



Class BasicSocket < IO

BasicSocket 是其他所有 socket 类的抽象基类。

这个类和它的子类，常常使用 `struct sockaddr` 的结构来操作地址，它实际上是一个不透明的二进制字符串。¹

类方法

<code>do_not_reverse_lookup</code>	<code>BasicSocket.do_not_reverse_lookup → true 或 false</code>
------------------------------------	---

返回全局反向查找标志。

<code>do_not_reverse_lookup=</code>	<code>BasicSocket.do_not_reverse_lookup = true 或 false</code>
-------------------------------------	---

设置全局反向查找标志。如果设置为 `true`，对远程地址的查询将返回数字式地址，而不是主机名。

`socket` 库默认会在连接时执行反向查找。如果由于某种原因查找过慢或超时，连接到一个主机将花费很长时间。将这个选项设置为 `false` 可以解决这个问题。

<code>for_fd</code>	<code>BasicSocket.for_fd(fd) → sock</code>
---------------------	--

将一个已打开的文件描述符包装（wrap）为一个 `socket` 对象。

<code>lookup_order=</code>	<code>BasicSocket.lookup_order = int</code>
----------------------------	---

设置全局地址查找的顺序。

实例方法

<code>close_read</code>	<code>sock.close_read → nil</code>
-------------------------	------------------------------------

关闭这个 `socket` 的可读连接。

<code>close_write</code>	<code>sock.close_write → nil</code>
--------------------------	-------------------------------------

关闭这个 `socket` 的可写连接。

<code>getpeername</code>	<code>sock.getpeername → string</code>
--------------------------	--

返回 `socket` 连接另一端所关联的 `struct sockaddr` 结构。

<code>getsockname</code>	<code>sock.getsockname → string</code>
--------------------------	--

返回和 `sock` 关联的 `struct sockaddr` 结构。

¹ 实际上它被映射到底层 C 语言 `struct sockaddr` 的一组结构，在手册页（manpage）中和 Stevens 的书（译注：指 Richard Stevens 的《UNIX 网络编程》一书）中均有记载。

getsockopt	<i>sock.getsockopt(level, optname) → string</i>
	返回指定选项的值。
recv	<i>sock.recv(len, <, flags>) → string</i>
	从 <i>sock</i> 接收至多 <i>len</i> 个字节。
send	<i>sock.send(string, flags, <, to>) → int</i>
	通过 <i>sock</i> 发送 <i>string</i> 。如果指定了目标， <i>to</i> 是一个表示接收地址的 <i>struct sockaddr</i> 结构。 <i>flags</i> 是某个或某几个 <i>MSG_</i> 选项的和（在下一页列出）。返回发送的字符数目。
setsockopt	<i>sock.setsockopt(level, optname, optval) → 0</i>
	设置一个 socket 选项。 <i>level</i> 是一个 socket 层级的选项（在下一页中列出）。 <i>optname</i> 和 <i>optval</i> 是特定于协议的——有关细节请参见你的系统文档。
shutdown	<i>sock.shutdown(how=2) → 0</i>
	关闭 socket 的接收者（ <i>how==0</i> ），发送者（ <i>how==1</i> ），或同时关闭（ <i>how==2</i> ）。

Class **Socket** < **BasicSocket**

`Socket` 类提供了对底层操作系统 `socket` 实现的访问。它可以用来提供更多特定于操作系统而不是特定于协议的功能，但是以更多的复杂性为代价。特别地，处理地址的类将 `struct sockaddr` 结构打包（`pack`）为 Ruby 字符串，让我们可以愉快地处理它。

类常量

`Socket` 类定义了在整个 `socket` 库中使用的常量。个别常量只有在支持相应功能的架构上可用。

类型:

`SOCK_DGRAM`, `SOCK_PACKET`, `SOCK_RAW`, `SOCK_RDM`, `SOCK_SEQPACKET`, `SOCK_STREAM`

协议族:

`PF_APPLETALK`, `PF_AX25`, `PF_INET6`, `PF_INET`, `PF_IPX`, `PF_UNIX`, `PF_UNSPEC`

地址族:

`AF_APPLETALK`, `AF_AX25`, `AF_INET6`, `AF_INET`, `AF_IPX`, `AF_UNIX`, `AF_UNSPEC`

查询顺序选项:

`LOOKUP_INET6`, `LOOKUP_INET`, `LOOKUP_UNSPEC`

发送/接收选项:

`MSG_DONTROUTE`, `MSG_OOB`, `MSG_PEEK`

Socket 层级选项:

`SOL_ATALK`, `SOL_AX25`, `SOL_IPX`, `SOL_IP`, `SOL_SOCKET`, `SOL_TCP`, `SOL_UDP`

Socket 选项:

`SO_BROADCAST`, `SO_DEBUG`, `SO_DONTROUTE`, `SO_ERROR`, `SO_KEEPALIVE`, `SO_LINGER`,
`SO_NO_CHECK`, `SO_OOBINLINE`, `SO_PRIORITY`, `SO_RCVBUF`, `SO_REUSEADDR`,
`SO_SNDBUF`, `SO_TYPE`

QOS (Quality of Service) 选项:

`SOPRI_BACKGROUND`, `SOPRI_INTERACTIVE`, `SOPRI_NORMAL`

多播:

`IP_ADD_MEMBERSHIP`, `IP_DEFAULT_MULTICAST_LOOP`, `IP_DEFAULT_MULTICAST_TTL`,
`IP_MAX_MEMBERSHIPS`, `IP_MULTICAST_IF`, `IP_MULTICAST_LOOP`, `IP_MULTICAST_TTL`

TCP 选项:

`TCP_MAXSEG`, `TCP_NODELAY`

getaddrinfo 错误代码:

EAI_ADDRFAMILY, EAI_AGAIN, EAI_BADFLAGS, EAI_BADHINTS, EAI_FAIL,
 EAI_FAMILY, EAI_MAX, EAI_MEMORY, EAI_NODATA, EAI_NONAME, EAI_PROTOCOL,
 EAI_SERVICE, EAI_SOCKTYPE, EAI_SYSTEM

ai_flags 的值:

AI_ALL, AI_CANONNAME, AI_MASK, AI_NUMERICHOST, AI_PASSIVE,
 AI_V4MAPPED_CFG

类方法**getaddrinfo**

Socket.getaddrinfo(*hostname, port,*
 <*family*<, *socktype*<, *protocol*<, *flags*>>>) → array

返回描述指定主机和端口（可以使用如上所示的限定参数）数组的一个数组。每个子数组，包括地址族、端口号、主机名、主机 IP 地址、协议族、socket 类型和协议。

```
for line in Socket.getaddrinfo('www.microsoft.com', 'http')
  puts line.join(", ")
end
```

输出结果:

```
AF_INET, 80, 207.46.134.221, 207.46.134.221, 2, 2, 17
AF_INET, 80, 207.46.134.221, 207.46.134.221, 2, 1, 6
AF_INET, 80, origin2.microsoft.com, 207.46.144.188, 2, 2, 17
AF_INET, 80, origin2.microsoft.com, 207.46.144.188, 2, 1, 6
AF_INET, 80, microsoft.com, 207.46.230.219, 2, 1, 6
AF_INET, 80, microsoft.net, 207.46.130.14, 2, 1, 6
```

gethostbyaddr

Socket.gethostbyaddr(*addr, type=AF_INET*) → array

返回给定地址的主机名、地址族、sockaddr 组成部分。

```
a = Socket.gethostbyname("161.58.146.238")
res = Socket.gethostbyaddr(a[3], a[2])
res.join(' ', ') → "www.pragmaticprogrammer.com, , 2, \241:\222\356"
```

gethostbyname

Socket.gethostbyname(*hostname*) → array

返回一个四个元素的数组，包括规格化的主机名、主机别名的子数组、地址族，以及 sockaddr 结构的地址部分。

```
a = Socket.gethostbyname("63.68.129.130")
a.join(' ', ') → "63.68.129.130, , 2, ?D\201\202"
```

gethostname

Socket.gethostname → string

返回当前主机的名称。

```
Socket.gethostname → "wireless_2.local.thomases.com"
```

getnameinfo**Socket.getnameinfo(*addr* <, *flags* >) → array**

查询给定的地址，可能是一个包括 `sockaddr` 的字符串，要么是一个三个或四个元素的数组。如果 `addr` 是一个数组，它应该包括地址族字符串、端口（或 `nil`）和主机名或 IP 地址。如果存在第四个元素或者非 `nil`，它将被作为主机名。以数组返回一个规格化的主机名（或地址）和端口号。

```
Socket.getnameinfo(["AF_INET", '23', 'www.ruby-lang.org'])
```

getservbyname**Socket.getservbyname(*service*, *proto*='tcp') → int**

- 返回给定服务和协议对应的端口号。

```
Socket.getservbyname("telnet") → 23
```

new**Socket.new(*domain*, *type*, *protocol*) → sock**

使用给定的参数创建一个 socket。

open**Socket.open(*domain*, *type*, *protocol*) → sock**

等同于 `Socket.new`。

pack_sockaddr_in**Socket.pack_sockaddr_in(*port*, *host*) → str_address**

1.8 给定一个端口和主机，以字符串的形式，返回 `sockaddr` 结构（系统相关的）。

```
require 'socket'
addr = Socket.pack_sockaddr_in(80, "pragprog.com")
# Pragprog.com is 216.87.136.211
addr.unpack("CCnC4") → [16, 2, 80, 68, 238, 145, 243]
```

pack_sockaddr_un**Socket.pack_sockaddr_in(*path*) → str_address**

1.8 给定一个 Unix socket 的路径，以字符串的形式，返回 `sockaddr` 结构（系统相关）。仅在支持 Unix 地址族的系统上可用。

```
require 'socket'
addr = Socket.pack_sockaddr_un("/tmp/sample")
addr[0,20] → "\000\001/tmp/sample\000\000\000\000\000\000\000"
```

pair**Socket.pair(*domain*, *type*, *protocol*) → array**

返回一个数组，其中包括用给定的域、类型和协议连接的一对匿名的 Socket 对象。

socketpair**Socket.socketpair(*domain*, *type*, *protocol*) → array**

等同于 `Socket.pair`。

unpack_sockaddr_in *Socket.pack_sockaddr_in(string_address) → [port, host]*

1.8. 给定一个含有二进制 `addrinfo` 结构的字符串，返回端口和主机。

```
require 'socket'
addr = Socket.pack_sockaddr_in(80, "pragprog.com")
Socket.unpack_sockaddr_in(addr) → [80, "216.87.136.211"]
```

unpack_sockaddr_un *Socket.pack_sockaddr_in(string_address) → [port, host]*

1.8. 给定一个含有二进制 `sock_addr_un` 结构的字符串，返回 Unix socket 的路径。仅在支持 Unix 地址族的系统上可用。

```
require 'socket'
addr = Socket.pack_sockaddr_in(80, "pragprog.com")
Socket.unpack_sockaddr_in(addr) → [80, "216.87.136.211"]
```

实例方法

accept *sock.accept → [socket, address]*

接受一个进入的连接，返回一个数组，其中包括一个新的 `Socket` 对象，以及包括发起者的 `struct sockaddr` 信息的字符串。

bind *sock.bind(sockaddr) → 0*

绑定到指定的、保存在字符串中的 `struct sockaddr`。

connect *sock.connect(sockaddr) → 0*

连接到指定的、保存在字符串中的 `struct sockaddr`。

listen *sock.listen(int) → 0*

侦听连接，使用指定的 `int` 作为最大连接数目。

recvfrom *sock.recvfrom(len <, flags>) → [data, sender]*

从 `sock` 接收至多 `len` 个字节。`flags` 为 0 或多个的 `MSG_` 选项的和。结果的第一个元素是接收到的数据。第二个元素包括发送端的协议相关信息。

sysaccept *sock.sysaccept → [socket_fd, address]*

1.8. 接收一个进入的连接。返回一个数组，包括进入连接的（整型）文件描述符，和一个字符串包括调用者的 `struct sockaddr` 信息。

Class **IPSocket < BasicSocket**

IPSocket 类是使用 IP 作为传输层的 socket 的基类。TCPSocket 和 UDPSocket 均基于此类。

类方法

getaddress

IPSocket.getaddress(*hostname*) → string

返回 *hostname* 的点段式 IP 地址。

```
a=IPSocket.getaddress('www.ruby-lang.org')
a → "210.251.121.210"
```

实例方法

addr

sock.addr → array

将 *sock* 的域 (domain)、端口、名称和 IP 地址作为有四个元素的数组返回。如果 `do_not_reverse_lookup` 标志为 `true`, 名字将以地址返回。

```
u = UDPSocket.new
u.bind('localhost', 8765)
u.addr → ["AF_INET", 8765, "localhost", "127.0.0.1"]
BasicSocket.do_not_reverse_lookup = true
u.addr → ["AF_INET", 8765, "127.0.0.1", "127.0.0.1"]
```

peeraddr

sock.peeraddr → array

返回对方的域、端口、名称和 IP 地址。

recvfrom

sock.recvfrom(*len* <, *flags*) → [*data*, *sender*]

从连接中接收至多 *len* 个字节。*flags* 是 0 或多个 `MSG_` 选项的和（在第 766 页列出）。^{1.8.} 返回一个两个元素的数组。第一个元素是接收到的数据，第二个是包括对方信息的数组。例如在我的 Mac OS X 机器中，`recvfrom()` 方法并不会返回 TCP 连接对方的信息，第二个元素的数组为 `nil`。

```
require 'socket'
t = TCPSocket.new('localhost', 'ftp')
data = t.recvfrom(40)
data → ["220 localhost FTP server (lukemftpd 1.1", nil]
t.close → nil
```

Class **TCPSocket** < **IPSocket**

```
t = TCPSocket.new('localhost', 'ftp')
t.gets → "220 localhost FTP server (lukemftpd 1.1) ready.\r\n"
t.close → nil
```

类方法

gethostbyname	TCPSocket.gethostbyname(<i>hostname</i>) → <i>array</i>
----------------------	--

查询 *hostname*, 并返回它的规格化 (canonical) 名称、别名的数组、AF_INET 的地址类型和点段式的 IP 地址。

```
a = TCPSocket.gethostbyname('ns.pragprog.com')
a → ["pragprog.com", [], 2, "216.87.136.211"]
```

new	TCPSocket.new(<i>hostname</i>, <i>port</i>) → <i>sock</i>
------------	--

打开一个到 *hostname* 和 *port* 的 TCP 连接。

open	TCPSocket.open(<i>hostname</i>, <i>port</i>) → <i>sock</i>
-------------	---

等同于 `TCPSocket.new`。

Class

SOCKSSocket < TCPSocket

SOCKSSocket 支持基于 SOCKS 协议的连接。

类方法

new `SOCKSSocket.new(hostname, port) → sock`

打开一个到 *hostname* 上 *port* 的 SOCKS 连接。

open `SOCKSSocket.open(hostname, port) → sock`

等同于 `SOCKSSocket.new`。

实例方法

close `sock.close → nil`

关闭这个 SOCKS 连接。

Class **TCPServer** < **TCPsocket**

TCPServer 接收进入的 TCP 连接。下面是一个 Web 服务器，在指定端口上侦听，并返回时间。

```
require 'socket'
port = (ARGV[0] || 80).to_i
server = TCPServer.new('localhost', port)
while (session = server.accept)
  puts "Request: #{session.gets}"
  session.print "HTTP/1.1 200/OK\r\nContent-type: text/html\r\n\r\n"
  session.print "<html><body><h1>#{Time.now}</h1></body></html>\r\n"
  session.close
end
```

类方法

new

TCPServer.new(<hostname,> port) → sock

在指定的接口（通过 *hostname* 和 *port* 标识）上创建一个新的 socket。如果省略了 *hostname*，服务器会侦听当前主机的所有接口（等价于地址 0.0.0.0）。

open

TCPServer.open(<hostname,> port) → sock

等同于 `TCPServer.new`。

实例方法

accept

sock.accept → tcp_socket

等待 *sock* 上进入的连接，并返回一个连接到调用者的新 *tcp_socket*。参见本页中的示例。



Class UDPsocket < IPSocket

UDP socket 发送和接收数据报（datagram）。要接收数据，必须将 socket 绑定到一个特定的端口。你在发送数据时有两个选择：你可以连接到一个远程的 UDP socket，然后向这个端口发送数据报，或者你可以为发送的每个包指定主机和端口。下面的示例是一个 UDP 服务器，打印它接收到的消息。无连接和基于连接的客户端都可以呼叫它。

```
require 'socket'
$port = 4321
server_thread = Thread.start do # run server in a thread
  server = UDPSocket.open
  server.bind(nil, $port)
  2.times { p server.recvfrom(64) }
end

# Ad-hoc client
UDPSocket.open.send("ad hoc", 0, 'localhost', $port)

# Connection based client
sock = UDPSocket.open
sock.connect('localhost', $port)
sock.send("connection-based", 0)
server_thread.join
```

输出结果：

```
["ad hoc", ["AF_INET", 55668, "localhost", "127.0.0.1"]]
["connection-based", ["AF_INET", 55669, "localhost", "127.0.0.1"]]
```

类方法

new	<code>UDPSocket.new(family = AF_INET) → sock</code>
------------	---

创建一个 UDP 端点（endpoint），可以指定一个地址族。

open	<code>UDPSocket.open(family = AF_INET) → sock</code>
-------------	--

等同于 `UDPSocket.new`。

实例方法

bind	<code>sock.bind(hostname, port) → 0</code>
-------------	--

将 UDP 连接的本地端关联到一个指定的 `hostname` 和 `port`。第一个参数是主机名，可以是 "`<broadcast>`" 或 ""（空字符串），以分别绑定到 `INADDR_BROADCAST` 和 `INADDR_ANY`。服务器必须使用它来建立可访问的端点（endpoint）。

connect	<code>sock.connect(hostname, port) → 0</code>
----------------	---

创建一个连接，到给定的 `hostname` 和 `port`。如果后续的 `UDPSocket#send` 请求

没有修改接收者，将使用这个连接。可以对 *sock* 调用多个 *connect* 请求：*send* 将使用最近的一个。第一个参数是主机名，可以是“<broadcast>”或“”（空字符串），以分别绑定到 `INADDR_BROADCAST` 和 `INADDR_ANY`。

recvfrom

sock.recvfrom(len <, flags >) → [data, sender]

从 *sock* 接收至多 *len* 个字节。*flag* 可以是 0 或多个 `MSG_` 选项的合（在第 766 页列出）。结果是一个有两个元素的数组，包括接收到的数据和发送者的信息。参见上一页中的范例。

send

sock.send(string, flags) → int

sock.send(string, flags, hostname, port) → int

两个参数的 *send* 使用一个已建立的连接来发送 *string*。四个参数的形式将 *string* 发送到 *hostname* 的 *port* 端口。



Class **UNIXSocket** < **BasicSocket**

类 `UNIXSocket` 支持使用 Unix 域协议（Unix domain protocol）进行进程间通讯。虽然底层的协议支持数据报和流连接，Ruby 库只提供了基于流的连接。

```
require 'socket'
SOCKET = "/tmp/sample"
server_thread = Thread.start do # run server in a thread
  sock = UNIXServer.open(SOCKET)
  s1 = sock.accept
  p s1.recvfrom(124)
end
client = UNIXSocket.open(SOCKET)
client.send("hello", 0)
client.close
server_thread.join
```

输出结果：

```
["hello", ["AF_UNIX", "q\1240"]]
```

类方法

<code>new</code>	<code>UNIXSocket.new(path) → sock</code>
用指定的 <code>path</code> 打开一个新的域 socket， <code>path</code> 必须是一个路径名。	
<code>open</code>	<code>UNIXSocket.open(path) → sock</code>
等同于 <code>UNIXSocket.new</code> 。	

实例方法

<code>addr</code>	<code>sock.addr → array</code>
返回该 socket 的地址族和路径。	
<code>path</code>	<code>sock.path → string</code>
返回该域 socket 的路径。	
<code>peeraddr</code>	<code>sock.peeraddr → array</code>
返回连接服务器端的地址族和路径。	
<code>recvfrom</code>	<code>sock.recvfrom(len <, flags>) → array</code>
从 <code>sock</code> 接受至多 <code>len</code> 个字节。 <code>flags</code> 可能是 0 或多个 <code>MSG_</code> 选项的和（在第 766 页列出）。返回数组的第一个元素是接收到的数据，第二个包括发送者的（最少）信息。	

Class **UNIXServer** < **UNIXSocket**

UNIXServer 类提供了一个简单的 Unix 域 socket 服务器。示例代码参见 **UNIXSocket**。

类方法

new	UNIXServer.new(path) → sock
------------	--------------------------------------

用指定的 *path* 创建一个服务器。对应文件在调用这个方法时必须是不存在的。

open	UNIXServer.open(path) → sock
-------------	---------------------------------------

等同于 `UNIXServer.new`。

实例方法

accept	sock.accept → unix_socket
---------------	----------------------------------

等待服务器 `socket` 上接收到的连接，并为这个连接返回一个新的 `socket` 对象。
参见上一页 `UNIXSocket` 中的示例。



MKMF 参考

MKMF Reference

Ruby 扩展模块使用 `mkmf` 库来帮助创建 `Makefile`。第 21 章，从第 275 页开始，描述了这些扩展是如何创建和构建（build）的。本附录描述了 `mkmf` 库的细节。

Module `mkmf`

`require "mkmf"`

当编写扩展时，你创建一个名为 `extconf.rb` 的程序，它可能像下面这样简单。

```
require 'mkmf'
create_makefile("Test")
```

在运行时，这个脚本会产生一个适合目标平台的 `Makefile`。它还会产生一个日志文件 `mkmf.log`，可以帮助你诊断构建时的问题。

`mkmf` 包括若干方法，你可以使用它们来找出为编译器设置的库和头文件标志。

`mkmf` 可以从多个源得到配置信息：

- 在构建 Ruby 时使用的配置。
- 环境变量 `CONFIGURE_ARGS`，一个 `key=value` 对。
- `Key=value` 或 `--key=value` 形式的命令行参数。

你可以通过倾列（dump）变量 `$configure_args` 来检查构建的配置。

```
% export CONFIGURE_ARGS="ruby=ruby18 --enable-extras"
% ruby -rmkmf -rpp -e 'pp $configure_args' -- --with-cflags=-O3
{ "--topsrcdir"=>".",
  "--topdir"=>"/Users/dave/Work/rubybook/tmp",
  "--enable-extras"=>true,
  "--with-cflags"=>"-O3",
  "--ruby"=>"ruby18"}
```

下面的配置选项是被认可的。

CFLAGS

传递给 C 编译器的标志（可以被--with-cflags 覆写）。

CPPFLAGS

传递给 C++ 编译器的标志（可以被--with-cppflags 覆写）。

curdir

设置全局\$curdir，可以在 extconf.rb 脚本中使用；否则无效。

disable-xxx

关闭特定于扩展的选项 xxx。

enable-xxx

启用特定于扩展的选项 xxx。

LDFLAGS

传递给链接器的标志（可以被--with-ldflags 覆写）。

ruby

设置在 Makefile 中使用的 Ruby 解释器的名字和/或路径。

srcdir

设置 Makefile 中的源代码目录路径。

with-cflags

传递给 C 编译器的标志。覆写 CFLAGS 环境变量。

with-cppflags

传递给 C++ 编译器的标志。覆写 CPPFLAGS 环境变量。

with-ldflags

传递给链接器的标志。覆写 LDFLAGS 环境变量。

with-make-prog

设置 make 程序的名字。如果在 Windows 上运行，make 程序的选择会影响所产生 Makefile 文件的语法（nmake vs. Borland make）。

with-xxx-{dir|include|lib}

控制 dir_config 方法查找的位置。

实例方法

create_makefile

`create_makefile(target, srcprefix=nil)`

为名为 *target* 的扩展创建一个 Makefile。*srcprefix* 可以覆写默认的源代码目录。如果没有调用这个方法，则不会创建 Makefile。

dir_config

`dir_config(name)`

查找 *name* 对应的目录配置选项，作为程序或者原本构建 Ruby 的参数。这些参数可以是下列之一：

```
--with-name-dir=directory  
--with-name-include=directory  
--with-name-lib=directory
```

所给出的目录会被加入 Makefile 内适当的搜索路径（头文件或链接）中。

enable_config

`enable_config(name, default=nil) → true 或 false 或 default`

测试选项中是否含有 --enable-*name* 或 --disable-*name*。如果给出的是 enable 选项，则返回 true；如果给出的是 disable 选项则返回 false；否则返回默认的值。

find_library

`find_library(name, function, <path>+) → true 或 false`

和 have_library 一样，但是还会在给出的目录路径中搜索。

have_func

`have_func(function) → true 或 false`

如果指定的方法在标准编译环境中存在，则将指令-DHAVE_FUNCTION 添加到 Makefile 的编译命令行中，并返回 true。

have_header

`have_header(header) → true 或 false`

如果指定的头文件可以在标准的搜索路径中找到，将指令-DHAVE_HEADER 添加到 Makefile 的编译命令行中，并返回 true。

have_library

`have_library(library, function) → true 或 false`

如果在给定的库（必须位于标准搜索路径，或者由 dir_config 添加的路径内）中存在指定的函数，将这个库添加到 Makefile 的链接命令中，并返回 true。

支持

Support

开放源码项目的一大特色之一是技术支持。各大媒体上的文章常常批评开放源码的努力，认为开源没有提供像商业产品那样的技术支持。不过伙计，这是个好事情！与其打电话给那些过劳且人手不足的服务部门，并享受一小时的等候音乐，甚至这样都无法得到你要的解答，我们有更好的方案：Ruby 社区。Ruby 的作者，本书的作者，以及许多 Ruby 的用户，都愿意并且能够在你需要时帮你一把。

Ruby 的语法保持着相当的稳定，但是随着软件的进化，新的特性不断地被加入。结果印刷的书籍和在线文档都可能会落在后面。所有的软件都有 bug，Ruby 也不例外。虽然没有很多，但是它们的确有时突然出现。

如果你遇到 Ruby 的问题，尽管在邮件列表或新闻组中发问（也就花上你 1 分钟时间）。通常，你可以及时地从 Matz 本人（Ruby 语言的作者）、其他高手和那些已经解决了与你相似问题的人那里，得到解答。

你可以在邮件列表或新闻组中查找相似的问题，并且在你发问之前从头到尾看一下最近的帖子，是一个良好的“网络礼仪”。如果你不能找到你需要的答案，发问，而正确的答案通常会很快并以非常高的精确性，出现在你的眼前。

C.1 Web 站点

Web Sites

因为 Web 的改变太快速，我们尽量让这个列表简短。访问下面列出的某个连接，你会发现许多其他有价值的 Ruby 在线资源。

Ruby 的官方主页是 <http://www.ruby-lang.org>。

你可以在 RubyForge (<http://www.rubyforge.org>) 上找到许多 Ruby 的类库和应用。

RubyForge 为 Ruby 开发者提供了开放源码项目的空间。每个项目都有一个 CVS 仓库 (repository)、保存发行归档的空间、bug 和特性 (feature) 请求的追踪系统、WikiWiki Web 系统和邮件列表。RubyForge 还可下载 RubyGems 的仓库。

Ruby Production Archive (Ruby 产品化归档, RPA) 位于 <http://www.ruby-archive.org>, 提供了许多预先打包好的 Ruby 库和应用。这个站点意图提供类似 Debian 或 FreeBSD 为它们各自社区提供的服务, 但是面向的是 Ruby 用户。这个站点在本书付梓时才出现, 我们没有任何使用它的直接经验。

Rubygarden 包括一个门户 (<http://www.rubygarden.org>) 以及一个 WikiWiki 站点 (<http://www.rubygarden.org>), 两者都提供了许多有用的 Ruby 信息。

<http://www.ruby-doc.org> 是各种 Ruby 文档资源的门户。

当你遇到麻烦时, 顺便访问一下 <http://www.pragmaticprogrammer.com>, 看看我们能帮些什么。

C.2 下载站点

Download Sites

你可以从下面的站点下载 Ruby 最新的版本:

<http://www.ruby-lang.org/en/>

预先编译好的 Windows 发布, 可以从下面的站点获得:

<http://rubyinstaller.rubyforge.org/>

这个项目也计划发布一个 Mac OS X 上的一站式安装程序, 但是当本书送到出版社时还未就绪。

C.2.1 Usenet 新闻组

Usenet Newsgroup

Ruby 有它自己的新闻组, `comp.lang.ruby`。在这个新闻组上的讨论, 被归档并镜像到 `ruby-talk` 邮件列表。

C.3 邮件列表

Mailing Lists

你可以发现许多关于 Ruby 的邮件列表。下面列出的前三个是英文的, 其余则大部分是日文的, 但是其中也有一些英文语言的帖子。

<code>ruby-talk@ruby-lang.org</code>	关于 Ruby 的英文讨论（镜像到 <code>comp.lang.ruby</code> ）
<code>ruby-doc@ruby-lang.org</code>	标准和工具的文档
<code>ruby-cvs@ruby-lang.org</code>	Ruby 源代码 CVS 提交的通知
<code>ruby-core@ruby-lang.org</code>	关于核心实现主题的英/日文混合讨论
<code>ruby-list@ruby-lang.org</code>	Ruby 的日文讨论
<code>ruby-dev@ruby-lang.org</code>	Ruby 开发者的列表
<code>ruby-ext@ruby-lang.org</code>	为 ruby 编写或用 Ruby 编写扩展的开发者列表
<code>ruby-math@ruby-lang.org</code>	用于计算的 Ruby 讨论

关于如何加入邮件列表，参见 <http://www.ruby-lang.org> 下面的“Mailing Lists”（邮件列表）标题。

邮件列表被归档到下面的地址，并可以进行搜索：

<http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml>,

或使用：

<http://www.ruby-talk.org>。





附录 D

书目

Bibliography

- [FJN02] Robert Feldt, Lyle Johnson, and Micheal Neuman. *The Ruby Developer's Guide*. Syngress Publishing, Inc, Rockland, MA, 2002.
- [Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools*. O'Reilly & Associates, Inc., Sebastopol, CA, second edition, 2002.
- [Ful01] Hal Fulton. *The Ruby Way*. Sams Publishing, 2001.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [Lid98] Stephen Lidie. *Perl/Tk Pocket Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, 1998.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.
- [Wal99] Nancy Walsh. *Learning Perl/Tk: Graphical User Interfaces with Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1999.



索引

Index

本书描述的每个内建和库方法都至少被索引了两次，一次是在方法的名下，另一次是在包括这个方法的类或模块的名下。这些项中的方法和类/模块名使用 typewriter 字体，并且在其后有 method、class 或 module 的字样。如果希望了解 String 类包括的方法，你可以在索引中查找“String class”。如果你想要了解哪些类和模块支持名为 index 的方法，则查找“index method”。这些方法所列的加黑页号，显示了参考章节的入口。

当某个类或模块的名字对应一个宽泛的概念（例如 String）时，我们将在这个概念之外单独索引该类。

符号是用 ASCII 排序的。右面的表可以帮助那些想不起标点字符顺序的人（鄙视一下你们）。

Symbols

! (logical not) 94, 341
!= (not equal) 95, 341
!~ (does not match) 69, 95, 341
(comment) 317
#! (shebang) 7
#{...}
 substitute in pattern 70, 326
 substitute in string 61, 321
\$ (global variable prefix) 328
\$ (in pattern) 70, 324
\$ variables
 \$! 108, 334, 360, 362
 \$\$ 124, 335
 \$\$ 335
 \$& 69, 334, 537
 \$* 335, 523
 \$+ 334
 \$, 334, 436, 526
 \$-0 334
 \$-F 335
 \$-I 336

\$-K 336
\$-a 335
\$-d 336
\$-i 336
\$-l 336
\$-p 336
\$-v 336
\$-w 336
\$. 334, 510
\$/ 178, 179, 334, 610
\$: 124, 160, 178, 183, 304,
 335
\$: 178, 334
\$< 335
\$= 324, 334
\$> 335
\$? 89, 148, 150, 335, 338,
 516, 531, 587, 591
\$@ 334
\$\ 179, 334, 511, 526
\$_ 24, 95, 101, 178, 335,
 342, 510, 523
\$` 69, 334, 538

\$~ 69, 77, 334, 537, 600–602
\$1 to \$9 334
\$1...\$9 69, 74, 326
\$0 180, 335, 337
\$configure_args 779
\$DEBUG 178, 336, 633, 637
\$deferr 335
\$defout 335
\$expect_verbose 676
\$F 178, 336
\$FILENAME 336
\$KCODE 324, 659, 689
\$LOAD_PATH 160, 178, 186,
 221–223, 225, 228, 336
\$SAFE 179, 313, 336, 398,
 575, 673
\$stderr 335
\$stdin 335
\$stdout 335
\$' 69, 334, 538
\$VERBOSE 178, 179, 336,
 534
English names 333, 671

<p>% method</p> <ul style="list-style-type: none"> class Bignum 441 class Fixnum 484 class Float 487 class String 606 %W{...} (array of words) 322 %q{...}, %Q{...} (string literal) 62, 320 %r{...} (regexp) 69, 324 %w{...} (array of words) 16, 322 %x{...} (command expansion) 89, 338, 516 %{...} (string literal) 62, 320 & (block parameter to method) 56, 81, 349 & method <ul style="list-style-type: none"> class Array 428 class Bignum 441 class FalseClass 464 class Fixnum 484 class NilClass 561 class Process::Status 591 class TrueClass 650 && (logical and) 94, 341 (...) (in pattern) 73, 326 (?...) (regexp extensions) 326 * (array argument) 347 * (in pattern) 72, 325 * method <ul style="list-style-type: none"> class Array 428 class Bignum 441 class Fixnum 484 class Float 487 class String 607 ** method <ul style="list-style-type: none"> class Bignum 441 class Fixnum 484 class Float 487 + (in pattern) 72, 325 + method <ul style="list-style-type: none"> class Array 428 class Bignum 441 class Fixnum 484 class Float 487 class String 607 class Time 644 +@ method <ul style="list-style-type: none"> class Numeric 562 - method <ul style="list-style-type: none"> class Array 428 class Bignum 441 class Fixnum 484 	<p>class Float 487</p> <ul style="list-style-type: none"> class Time 644 <p>-@ method</p> <ul style="list-style-type: none"> class Bignum 441 class Fixnum 484 class Float 487 class Numeric 562 (in pattern) 325 .. and ... (range) 66, 342 <p>/ method</p> <ul style="list-style-type: none"> class Bignum 441 class Fixnum 484 class Float 487 /.../ (regexp) 69, 324 <p>:</p> <ul style="list-style-type: none"> (symbol creation) 323, 338 (then replacement) 97, 99 <p>:: (scope resolution) 330, 338, 352, 354</p> <p>vs. ":" 349</p> <p>; (line separator) 317</p> <p>< (superclass) 352</p> <p>< method</p> <ul style="list-style-type: none"> module Comparable 447 <p><, <=, >, >= method</p> <ul style="list-style-type: none"> class Module 546 <p><= method</p> <ul style="list-style-type: none"> module Comparable 447 <p><=> (comparison operator) 67, 95, 447, 454</p> <p><=> method</p> <ul style="list-style-type: none"> class Array 429 class Bignum 441 class File::Stat 477 class Fixnum 484 class Float 487 class Module 546 class Numeric 562 class String 607 class Time 644 <p><<</p> <ul style="list-style-type: none"> here document 62, 321 singleton object 352, 382 <p><< method</p> <ul style="list-style-type: none"> class Array 428 class Bignum 441 class Fixnum 484 class IO 132, 507 class String 607 <p>= (assignment) 90, 338</p> <p>== (equals) 95</p> <p>== method</p> <ul style="list-style-type: none"> class Array 429 class Bignum 441 class Float 488 	<p>class Hash 493</p> <ul style="list-style-type: none"> class Method 543 class Object 567 class Process::Status 591 class Proc 580 class Range 598 class Regexp 601 class String 607 class Struct 627 <p>module Comparable 447</p> <p>===(case equals) 95, 99, 110, 343</p> <p>==== method</p> <ul style="list-style-type: none"> class Module 546 class Object 567 class Range 598 class Regexp 601 class String 607 <p>=></p> <ul style="list-style-type: none"> hash creation 46, 322 in argument list 85, 348 rescue clause 110, 361 <p>=begin...=end 318</p> <ul style="list-style-type: none"> embedded documentation 202 <p>=~ (match) 69, 95</p> <p>=~ method</p> <ul style="list-style-type: none"> class Object 567 class Regexp 601 class String 608 <p>> method</p> <ul style="list-style-type: none"> module Comparable 447 <p>>= method</p> <ul style="list-style-type: none"> module Comparable 447 <p>>> method</p> <ul style="list-style-type: none"> class Bignum 441 class Fixnum 484 class Process::Status 591 <p>? (in pattern) 72, 325</p> <p>? (ternary operator) 97</p> <p>@ (instance variable prefix) 328</p> <p>@@ (class variable prefix) 328</p> <p>[] method</p> <ul style="list-style-type: none"> class Array 44, 427, 429 class Bignum 442 class Dir 449 class Fixnum 485 class Hash 492, 493 class MatchData 537 class Method 543 class Proc 580 class String 608 class Struct 627, 628 class Thread 635
--	--	---

[]= method
 class Array 44, 430
 class Hash 493
 class String 609
 class Struct 628
 class Thread 636
 [...]
 array literal 16, 322
 bracket expression 325
 character class 71
 \$\ variable 179, 334, 511, 526
 \ (line continuation) 317
 \& (in substitution) 75
 \` (in substitution) 75
 \+ (in substitution) 75
 \1... \9
 in pattern 74, 326
 in substitution 75
 \A (in pattern) 324
 \B (in pattern) 325
 \D (in pattern) 325
 \G (in pattern) 325, 326
 \S (in pattern) 325
 \W (in pattern) 325
 \Z (in pattern) 325
 * (in substitution) 75
 \b (in pattern) 325
 \d (in pattern) 325
 \n (newline) 14, 321
 \s (in pattern) 325
 \w (in pattern) 325
 \z (in pattern) 325
 ^ (in pattern) 70, 71, 324
 ^ method
 class Bignum 441
 class FalseClass 464
 class Fixnum 484
 class NilClass 561
 class TrueClass 650
 _id__ method
 class Object 567
 _send__ method
 class Object 567
 _id2ref method
 module ObjectSpace 578
 \$` variable 69, 334, 538
 ` (backquote) method
 module Kernel 89, 148, 516
 {...}
 hash literal 17, 322
 in pattern 72, 325
 see also Block
 --verbose (Ruby option) 535
 |

in file name 149
 in pattern 73, 325
 | method
 class Array 430
 class Bignum 441
 class FalseClass 464
 class Fixnum 484
 class NilClass 561
 class TrueClass 650
 || (logical or) 94, 341
 \$~ variable 69, 77, 334, 537,
 600–602
 ~ method
 class Bignum 441
 class Fixnum 484
 class Regexp 601
 class String 609
 -0[octal] (Ruby option) 178

A

-a (Ruby option) 178, 335, 336
 \$-a variable 335
 Abbrev module 655, 723
 Abbreviations, calculating 655
 Abort *see* Exception
 abort method
 module Kernel 517
 module Process 583
 abort_on_exception method
 class Thread 138, 633, 636
 abort_on_exception= method
 class Thread 633, 636
 abs method
 class Bignum 442
 class Fixnum 485
 class Float 488
 class Numeric 562
 accept method
 class Socket 769
 class TCPServer 773
 class UNIXServer 777
 Access control 37, 356
 method 551, 553, 558, 559
 overriding in subclass 393
see also File, permission
 Accessor method 29, 92
 acos method
 module Math 540
 acosh method
 module Math 540
 ActiveX *see* Microsoft
 Windows, automation
 Ad hoc testing 151

add method
 class ThreadGroup 640
 add_observer method
 module Observable 706
 addr method
 class IPStream 770
 class UNIXStream 776
 AF_INET class 688
 Alias 40, 169, 330
 alias 351
 alias_method method
 class Module 554
 module Kernel 410
 alive? method
 class Thread 636
 all? method
 module Enumerable 454
 all_symbols method
 class Symbol 631
 ALLOC 293
 ALLOC_N 293
 ALLOCA_N 294
 allocate method
 class Class 446
 Allocation 287
 Amrita
 templates 241
 Ancestor 27
 ancestors method
 class Module 382, 405, 547
 and (logical and) 94, 341
 Anonymous class 382, 445
 any? method
 module Enumerable 454
 Aoki, Minero 112
 Aoyama, Wakou 76
 Apache Web server 243
 mod_ruby 247
 API
 Microsoft Windows 268, 755
 Ruby *see* Extend Ruby
 APOP authentication 702
 append_features method
 class Module 554
 ARGF constant 180
 ARGF variable 24, 336
 Argument, command-line *see*
 Command line
 Argument, method 79, 80
 ARGV variable 178–180, 336,
 523, 684, 711
 Arithmetic 692, 721
 Arithmetic operations
 method

<p>class Bignum 441 class Fixnum 484 class Float 487 arity method class Method 543 class Proc 580 class UnboundMethod 652 Array associative <i>see</i> Hash creating 43 expanding as method parameter 83, 348 indexing 44 literal 16, 322 method argument 347 Array class 373, 427 & 428 * 428 + 428 - 428 <=> 429 << 428 == 429 [] 44, 427, 429 []= 44, 430 430 assoc 430 at 430 clear 431 collect! 431 compact 431 compact! 431 concat 431 delete 431 delete_at 432 delete_if 432 each 432 each_index 432 empty? 432 eql? 432 fetch 433 fill 433 first 433 flatten 433 flatten! 434 include? 434 index 434 indexes 434 indices 434 insert 434 join 436 last 436 length 436 map! 436</p>	<p>new 427 nitems 436 pack 131, 435, 436 pop 437 push 437 rassoc 437 reject! 437 replace 437 reverse 437 reverse! 437 reverse_each 438 rindex 438 scanf 729 shift 438 size 438 slice 438 slice! 439 sort 439 sort! 439 to_a 439 to_ary 439 to_s 440 transpose 440 uniq 440 uniq! 440 unshift 440 values_at 440 Array method module Kernel 516 ASCII 317 character literal 60, 320 convert integer to 501 asctime method class Time 644 asin method module Math 540 asinh method module Math 540 ASP <i>see</i> eruby assert_equal method 152 assert_not_nil method 156 Assertions <i>see</i> Test::Unit, assertions Assignment 90, 338 attribute 350 parallel 91, 340 assoc method class Array 430 Associative array <i>see</i> Hash Asynchronous I/O 687 at method class Array 430 class Time 642 at_exit method</p>	<p>module Kernel 517 atan method module Math 540 atan2 method module Math 540 atanh method module Math 540 atime method class File::Stat 477 class File 465, 475 Atom <i>see</i> Symbol attr method 354 class Module 554 attr_accessor method 354 class Module 555 attr_reader method 30, 354, 388 class Module 555 attr_writer method 354 class Module 555 Attribute 29 assignment 167, 350 virtual 32 writable 31 <i>see also</i> Class attribute autoload method class Module 547 module Kernel 517 autoload? method class Module 547 module Kernel 518 Automation, Windows 269, 756 Autosplit mode 178</p>
--	---	--

B

Backquote character *see*
 `(backquote)
Backreferences (in regular
 expressions) 73–75, 326,
 351
Backtrace *see* \$@, caller
backtrace method
 class Exception 461
Backup files, creating 178
Base
 number 443, 486, 622
Base (numeric) *see* to_s
 methods,
 Kernel::Integer,
 String#to_i
Base64 module 656
base_uri method 707
basename method

class File 465
BasicSocket class 127, 735, 764
 close_read 764
 close_write 764
 do_not_reverse_lookup 764
 do_not_reverse_lookup= 764
 for_fd 764
 getpeername 764
 getsockname 764
 getsockopt 765
 lookup_order= 764
 recv 765
 send 765
 setsockopt 765
 shutdown 765
BEGIN {...} 318
begin method
 class MatchData 537
 class Range 598
 =begin...=end 318
 begin...end 101, 108, 340, 361
Benchmark module 170, 409, 657
 Berger, Daniel 273
between? method
 module Comparable 447
BigDecimal class 658
BigMath module 658
Bignum class 59, 441, 484, 658, 692
 % 441
 & 441
 * 441
 ** 441
 + 441
 - 441
 -@ 441
 / 441
 <=> 441
 << 441
 == 441
 >> 441
 [] 442
 ^ 441
 | 441
 ~ 441
 abs 442
 Arithmetic operations 441
 Bit operations 441

div 442
 divmod 442
 eql? 442
 literal 59, 319
 modulo 442
 quo 442
 remainder 443
 size 443
 to_f 443
 to_s 443
 Binary data 131, 321, 436, 623
 Binary notation 59, 319
bind method
 class Socket 769
 class UDPSocket 774
 class UnboundMethod 652
Binding
 in block 333
 GUI events 260
Binding class 337, 444
binding method
 class Proc 581
 module Kernel 409, 444, 518
binmode method
 class IO 507
Bit operations method
 class Bignum 441
 class Fixnum 484
blksize method
 class File::Stat 477
Block 21, 49, 356
 break and next 357
 as closure 55
 and files 128
 for busy cursor 263
 fork, popen, and subprocess 150, 506, 525, 708
 with method 408
 as parameter to method 80, 348, 349
 parameters 21, 51
 return from 359
 as transaction 53
 variable scope 105, 136, 333
see also iterator
block_given? method
 module Kernel 54, 349, 518
blockdev? method
 class File::Stat 477
 class File 466
blocks method
 class File::Stat 478
BlueCloth 218

Boolean expressions 341
see also FalseClass, TrueClass
 Bottlenecks 170
 break 104, 345, 357
 Breakpoint 163
 Buffering problems 169
 Bug *see* Testing
 Build environment *see* Config module
 Busy cursor 263

C

-c (Ruby option) 178
 -C directory (Ruby option) 178
C language *see* Extend Ruby
C language API
 ALLOC 293
 ALLOC_N 293
 ALLOCA_N 294
 Data_Get_Struct 285
 Data_Make_Struct 285
 Data_Wrap_Struct 285
 OBJ_FREEZE 312
 OBJ_FROZEN 313
 OBJ_TAINT 312
 OBJ_TAINTED 312
 rb_apply 309
 rb_ary_entry 314
 rb_ary_new 313
 rb_ary_new2 313
 rb_ary_new3 313
 rb_ary_new4 313
 rb_ary_pop 313
 rb_ary_push 313
 rb_ary_shift 313
 rb_ary_store 313
 rb_ary_unshift 314
 rb_block_given_p 311
 rb_bug 310
 rb_call_super 309
 rb_catch 311
 rb_class_new_instance 309
 rb_cv_get 312
 rb_cv_set 312
 rb_cvar_defined 312
 rb_cvar_get 312
 rb_cvar_set 312
 rb_define_alias 307
 rb_define_alloc_func 307

rb_define_attr 309
 rb_define_class 305
 rb_define_class_under 305
 rb_define_class_variable 308
 rb_define_const 308
 rb_define_global_const 308
 rb_define_global_function 307
 rb_define_hooked_variable 308
 rb_define_method 307
 rb_define_module 305
 rb_define_module_function 307
 rb_define_module_under 306
 rb_define_READONLY_VARIABLE 308
 rb_define_singleton_method 307
 rb_define_VARIABLE 308
 rb_define_virtual_VARIABLE 308
 rb_each 311
 rb_ensure 310
 rb_exit 310
 rb_extend_object 306
 rb_fatal 310
 rb_funcall 309
 rb_funcall2 309
 rb_funcall3 309
 rb_global_VARIABLE 309
 rb_gv_get 312
 rb_gv_set 312
 rb_hash_aref 314
 rb_hash_aset 314
 rb_hash_new 314
 rb_id2name 309
 rb_include_module 306
 rb_intern 309
 rb_iter_break 311
 rb_iterate 311
 rb_iv_get 311
 rb_iv_set 312

rb_ivar_get 312
 rb_ivar_set 312
 rb_load_file 304
 rb_notimplement 310
 rb_obj_is_instance_of 314
 rb_obj_is_kind_of 314
 rb_protect 310
 rb_raise 310
 rb_require 306
 rb_rescue 310
 rb_respond_to 314
 rb_safe_level 313
 rb_scan_args 307
 rb_secure 313
 rb_set_safe_level 313
 rb_str_cat 314
 rb_str_concat 314
 rb_str_dup 314
 rb_str_new 314
 rb_str_new2 314
 rb_str_split 314
 rb_struct_aref 306
 rb_struct_aset 306
 rb_struct_define 306
 rb_struct_new 306
 rb_sys_fail 310
 rb_thread_create 314
 rb_throw 311
 rb_undef_method 307
 rb_warn 311
 rb_warning 311
 rb_yield 311
 REALLOC_N 294
 ruby_finalize 304
 ruby_init 304
 ruby_init_loadpath 304
 ruby_options 304
 ruby_run 304
 ruby_script 304
 SafeStringValue 313
 call method
 class Continuation 448
 class Method 408, 543
 class Proc 581
 :call-seq: (RDoc) 206, 209
 Callback
 from GUI widget 257
 Ruby runtime 411
 see also Block, closure
 callcc method
 module Kernel 448, 518
 caller method

module Kernel 113, 412, 413, 518
 CamelCase 328
 capitalize method
 class String 609
 capitalize! method
 class String 609
 captures method
 class MatchData 537
 case expression 98, 343
 Case insensitive
 string comparison 610
 Case insensitive (regexp) 324
 casecmp method
 class String 610
 casefold? method
 class Regexp 602
 catch method
 module Kernel 105, 114, 362, 519
 ceil method
 class Float 488
 class Integer 501
 class Numeric 562
 center method
 class String 610
 CFLAGS (mkmf) 780
 CGI class 236, 659
 cookies 244
 has_key? 238
 params 237
 CGI programming 235–253
 cookies 244
 embedding Ruby (eruby) 242
 forms 237
 generate HTML 238
 mod_ruby 247
 query parameters 237
 quoting 236
 session 246
 WEBrick 247
 see also Network protocols, Templates
 CGI::Session class 661
 CGIKit, Web framework 253
 change_privilege method
 module Process::GID 589
 module Process::UID 596
 changed method
 module Observable 706
 changed? method
 module Observable 706
 Character

convert integer to 501
literal 60, 320
Character class 71
chardev? method
 class File::Stat 478
 class File 466
charset method 707
chdir method
 class Dir 182, 449
Checksum 621, 668
Child process *see* Process
chmod method
 class File 466, 475
chomp method
 class String 63, 610
 module Kernel 519
chomp! method
 class String 610
 module Kernel 519
chop method
 class String 610, 689
 module Kernel 520
chop! method
 class String 611, 689
 module Kernel 520
chown method
 class File 466, 475
chr method
 class Integer 501
chroot method
 class Dir 450
Class
 anonymous 382, 445
 attribute 29, 354
 defining 352, 387
 extending 26
 generator 626
 hierarchy 546
 instance 12, 353
 listing hierarchy 405
 metaclass 380
 method 34, 385
 mixing in module 355
 naming 16, 392
 object specific 382
 vs type 366
 variable 33
 virtual 381, 382
Class class 379, 445
 allocate 446
 inherited 411, 445
 new 353, 445, 446
 superclass 405, 446
class method

 class Object 370, 567
 class_eval method
 class Module 547
 class_variables method
 class Module 548
Classes
 list of methods 424
 AF_INET 688
 Array 373, 427
 BasicSocket 127, 735, 764
 BigDecimal 658
 Bignum 59, 441, 484, 658,
 692
 Binding 337, 444
 CGI 236, 659
 CGI::Session 661
 Class 379, 445
 Complex 662, 692
 Continuation 448, 518
 CSV 663
 CSV::Row 663
 Date 665
 DateTime 665
 DBM 666
 Delegator 667
 Dir 449, 714
 DRb 670
 ERB 673
 Exception 107, 360, 461
 FalseClass 464
 File 127, 465, 483, 681,
 714
 File::Stat 477
 Fixnum 59, 484, 692
 Float 60, 487
 GDBM 682
 Generator 683
 GetoptLong 684
 GServer 685
 Hash 373, 492
 Iconv 686
 Integer 373, 501, 692
 IO 127, 373, 503, 676, 729,
 735, 736
 IPAddr 688
 IPSocket 735, 770
 Logger 690
 Mail 691
 MatchData 69, 77, 537, 600,
 602, 608
 MatchingData 537
 Matrix 694
 Method 408, 543, 555
 Module 545

Monitor 142, 144, 738, 743
Mutex 696, 697
Net::FTP 698
Net::HTTP 699, 752
Net::IMAP 701
Net::POP3 702
Net::SMTP 703
Net::Telnet 704
NilClass 561
Numeric 562, 662
Object 29, 393, 567
OpenStruct 626, 710
OptionParser 711
Pathname 714
PP 715
PrettyPrint 526, 715, 716
Proc 56, 81, 357, 359, 373,
 544, 555, 580
Process::Status 150, 587,
 588, 591
PStore 719
Queue 141, 145, 743
Range 67, 322, 597
Rational 692, 721
Regexp 76, 600
SDBM 730
Set 428, 430, 731
SimpleDelegator 667
SizedQueue 743
Socket 735, 766
SOCKSSocket 735, 772
String 61, 320, 374, 606,
 689, 729
StringIO 132, 736
StringScanner 737
Struct 626
Struct::Tms 630
Symbol 31, 338, 374, 615,
 631
Sync 738
SyncEnumerator 683
Syslog 740
TCPServer 773
TCPSocket 735, 771
Tempfile 741
Test::Unit 742
Thread 633
ThreadGroup 636, 640
ThreadsWait 744
Time 465, 642, 745
Tk 747
TrueClass 650
UDPSocket 735, 774

UnboundMethod 408, 543, 544, 549, 651
 UNIXServer 777
 UNIXSocket 735, 776
 URI 752
 Vector 694
 WeakRef 753
 Win32API 669, 755
 WIN32OLE 756
clear method
 class Array 431
 class Hash 493
 Client/Server 417, 670, 685, 734
clone method
 class IO 507
 class Module 548
 class Object 288, 568
close method
 class Dir 452
 class IO 507
 class SOCKSSocket 772
close_read method
 class BasicSocket 764
 class IO 507
close_write method
 class BasicSocket 764
 class IO 508
closed? method
 class IO 508
 Closure 55. *see* Block
 Code profiler 171
 Coding system (ASCII, EUC, SJIS, UTF-8) 179, 317n, 321n, 686, 689, 705
coerce method 374
 class Numeric 374, 562
 Coercion 374
 Coffee coaster
 attractive xxxii
collect method
 module Enumerable 52, 431, 454
collect! method
 class Array 431
 COM *see* Microsoft Windows, automation
 Comma-separated data 663
 Command (type of method) 82n
 Command expansion 89
see also ` (backquote)
 Command line 129, 177
 options 178–180
 parsing 684, 711

see also ARGV
 Command line, parsing 732
 Command, editing with readline 723
 Comment 317
 for RDoc 199
 regular expression 326
 Common Gateway Interface
see CGI programming
compact method
 class Array 431
compact! method
 class Array 431
Comparable module 119, 447
 < 447
 <= 447
 == 447
 > 447
 >= 447
 between? 447
 Comparisons 447
Comparison operators 341
see also <=>
Comparisons method
 module Comparable 447
compile method
 class Regexp 600
 Completion, trb 187
 Complex class 662, 692
 Compression, gzip 759
 COMSPEC 182, 521
concat method
 class Array 431
 class String 611
Condition variable *see* Thread, condition variable (and Thread, synchronization)
Conditional expression 97, 343
see also Range
Config module 183
CONFIGURE_ARGS 779
\$configure_args variable 779
connect method
 class Socket 769
 class UDPSocket 774
const_defined? method
 class Module 548
const_get method
 class Module 548
const_missing method
 class Module 548
const_set method
 class Module 549
Constant 330

class name 392
 listing in module 406
 scope 330
Constants
 ARGF 180
 DATA 318, 337
 Errno 110
 FALSE 337
 false 93, 336, 341
 __FILE__ 337
 NIL 337
 nil 16, 93, 336, 341
 RUBY_PLATFORM 337
 RUBY_RELEASE_DATE 337
 RUBY_VERSION 337
 SCRIPT_LINES__ 337, 414, 528
 STDERR 337, 534
 STDIN 337, 525
 STDOUT 337, 525, 526
 TOLEVEL_BINDING 337
 TRUE 337
 true 93, 337, 341
constants method
 class Module 545, 549
Constructor 12, 25
 initialize method 575
 private 35
 Contact, authors' e-mail xxvii
 Containers *see* Array and Hash
content_encoding method 707
content_type method 707
Continuation class 448, 518
 call 448
Control character
 \n etc. 60, 320, 321
 Conventions, typographic xxix
 Conversion protocols 372
 Cookies *see* CGI programming, cookies
cookies method
 class CGI 244
 Cookies, HTTP 244
 Coordinated Universal Time 642
 --copyright (Ruby option) 178
 CORBA *see* Distributed Ruby
coredump? method
 class Process::Status 591
cos method
 module Math 540
cosh method

module Math 540
 count method
 class String 611
 count_observers method
 module Observable 706
 CPPFLAGS (mkmf) 780
 CPU times 630
 CRAM-MD5 authentication 701
 create_makefile method
 module mkmf 296, 781
 Critical section *see* Thread, synchronization
 critical method
 class Thread 633
 critical= method
 class Thread 141, 633
 crypt method
 class String 611
 Cryptographic Hashes 668
 CSV class 663
 CSV::Row class 663
 ctime method
 class File::Stat 478
 class File 466, 475
 class Time 644
 curdir (mkmf) 780
 Current directory 450
 current method
 class Thread 634
 Curses module 664
 CVS access to Ruby 6
 CVSup 6
 cygwin32 267

D

-d (Ruby option) 138, 336, 535
 -d, --debug (Ruby option) 178
 \$-d variable 336
 DATA constant 318, 337
 Data_Get_Struct 285
 Data_Make_Struct 285
 Data_Wrap_Struct 285
 Database *see* dbm, gdbm, qdbm, sdbm
 Datagram *see* Network protocols, UDP
 Date class 665
 parsing 713
 see also Time class
 Date module 642
 DateTime class 665

day method
 class Time 644
 DBM class 666
 dbm 666
 DCOM *see* Microsoft Windows, automation
 Deadlock *see* Thread
 Debian installation 4
 \$DEBUG variable 178, 336, 633, 637
 Debug mode 138, 178
 Debugger 163
 commands 173f
 Decimal notation 59, 319
 Decoupling 28
 Decoux, Guy 289
 def (method definition) 79
 Default (ThreadGroup constant) 640
 Default parameters 79, 347
 default method
 class Hash 494
 default= method
 class Hash 494
 default_proc method
 class Hash 494
 \$deferr variable 335
 define_finalizer method
 module ObjectSpace 578
 define_method method
 class Module 555
 module Module 360
 defined? operator 94, 341
 \$defout variable 335
 Delegation 667, 680
 Delegator class 667
 delete method
 class Array 431
 class Dir 450
 class File 466
 class Hash 494
 class String 611, 689
 delete! method
 class String 612, 689
 delete_at method
 class Array 432
 delete_if method
 class Array 432
 class Hash 495
 delete_observer method
 module Observable 706
 delete_observers method
 module Observable 706
 Delimited string 318

Dependency, RubyGems 215
 Design Pattern *see* Patterns
 detach method
 module Process 583
 detect method
 module Enumerable 454
 Determinant, matrix 694
 dev method
 class File::Stat 478
 dev_major method
 class File::Stat 478
 dev_minor method
 class File::Stat 478
 Dictionary *see* Hash
 DIG (Float constant) 487
 Digest module 668
 Dir
 match modes 468f
 Dir class 449, 714
 [] 449
 chdir 182, 449
 chroot 450
 close 452
 delete 450
 each 452
 entries 450
 foreach 450
 getwd 450
 glob 451
 mkdir 451
 new 451
 open 452
 path 452
 pos 452
 pos= 453
 pwd 452
 read 453
 rewind 453
 rmdir 452
 seek 453
 tell 453
 tmpdir 246, 748
 unlink 452
 see also Find module
 dir_config method
 module mkmf 297, 781
 Directories
 include and library for extensions 297
 lib/ 296
 pathname 714
 search path 298
 searched 183
 temporary 748

working 178
directory? method
 class `File::Stat` 478
 class `File` 467
dirname method
 class `File` 467
disable method
 module `GC` 491
disable-xxx (mkmf) 780
Dispatch table 407
display method
 class `Object` 568
Distributed Ruby 417, 670, 727, 757
Distribution *see* RubyGems
div method
 class `Bignum` 442
 class `Fixnum` 485
 class `Numeric` 563
Division, accuracy 692, 721
divmod method
 class `Bignum` 442
 class `Fixnum` 485
 class `Float` 488
 class `Numeric` 565
DL module 669
DL library 273
DLL, accessing API 268, 669, 755
DLN_LIBRARY_PATH 182
DNS 724
do (in loops) 344
do...end *see* Block
do_not_reverse_lookup
 method
 class `BasicSocket` 764
do_not_reverse_lookup=
 method
 class `BasicSocket` 764
:doc: (RDoc) 206
Document Type Definition 725
Document-class: (RDoc) 209
Document-method: (RDoc)
 209
Documentation
 doc string example 389
 embedded 199, 318
 modifiers 205
see also RDoc
doGoogleSearch method 251
Domain Name System 724
Dotted quad *see* Network
 protocols
Double dispatch 375

Double-quoted string 61, 320
downcase method
 class `String` 612
downcase! method
 class `String` 612
Download
 Ruby 3
 source from book 5
Download Ruby
 sites 784
downto method
 class `Integer` 102, 501
dpkg installation 4
DRb
see also Distributed Ruby
DRb class 670
DRbUndumped module 670
dst? method
 class `Time` 644
DTD 725
Duck typing 294, 365–377
_dump 415, 535
dump method
 class `String` 612
 module `Marshal` 414, 536
dup method
 class `Object` 288, 568
Dynamic
 compilation 520
 definitions 387
 linking 301, 669
 method invocation 407
see also Reflection

E

E (Math constant) 540
-e 'command' (Ruby option)
 178
E-mail
 date/time formats 745
each method 672
 class `Array` 432
 class `Dir` 452
 class `Hash` 495
 class `IO` 508
 class `Range` 598
 class `String` 612
 class `Struct` 628
 module `Enumerable` 52, 454
each_byte method
 class `IO` 130, 508
 class `String` 613
each_cons method 672

each_index method
 class `Array` 432
each_key method
 class `Hash` 495
each_line method
 class `IO` 130, 509
 class `String` 613
each_object method
 module `ObjectSpace` 382, 404, 406, 578
each_pair method
 class `Hash` 495
 class `Struct` 628
each_slice method 672
each_value method
 class `Hash` 495
each_with_index method
 module `Enumerable` 455
Editor
 run Ruby in 165
egid method
 module `Process` 584
egid= method
 module `Process` 584
eid method
 module `Process::GID` 589
 module `Process::UID` 596
eid= method
 module `Process::GID` 589
 module `Process::UID` 596
Eiffel
 once modifier 391
Element reference ([]) 351
else (exceptions) 111, 362
see also if, case
elsif 343
Emacs 165
 tag file 196
Emacs key binding 723
E-mail
 address for feedback xxvii
 fetching with IMAP 701
 fetching with POP 702
 parsing 691
 sending with SMTP 703
Embed Ruby
 in HTML etc. *see* eruby
 interpreter in application 301
Embedded documentation 199, 318
empty? method
 class `Array` 432
 class `Hash` 495
 class `String` 613

enable method
 module GC 491
 enable-xxx (mkmf) 780
 enable_config method
 module mkmf 781
 enclose method
 class ThreadGroup 640
 enclosed? method
 class ThreadGroup 641
 Encodings, character 686, 689, 705
 Encryption 611
 __END__ 318, 337
 END {...} 318
 End of line 130
 end method
 class MatchData 538
 class Range 598
 :enddoc: (RDoc) 207
 English library 671
 English names for \$ variables 333, 671
 ensure (exceptions) 111, 362
 entries method
 class Dir 450
 module Enumerable 455
 enum_for method 672
 Enumerable class
 Enumerator 672
 Enumerable module 52, 120, 454, 672, 683
 all? 454
 any? 454
 collect 52, 431, 454
 convert to Set 731
 detect 454
 each 52, 454
 each_with_index 455
 entries 455
 find 455
 find_all 455
 grep 455
 include? 455
 inject 52, 120, 456
 map 456
 max 456
 member? 456
 min 456
 partition 457
 reject 437, 457
 select 457
 sort 457
 sort_by 457
 to_a 459

zip 459
 Enumerator module 672
 ENV variable 181, 336
 Environment variables 181
 COMSPEC 182, 521
 DLN_LIBRARY_PATH 182
 HOME 182, 449, 467
 LOGDIR 182, 449
 OPENSSL_CONF 182
 PATH 179
 POSIXLY_CORRECT 684
 RI 9, 214
 RUBY_TCL_DLL 182
 RUBY_TK_DLL 182
 RUBYLIB 182, 183, 401
 RUBYLIB_PREFIX 182
 RUBYOPT 182, 401
 RUBYPATH 179, 182
 RUBYSHELL 182, 521
 SHELL 182
 see also ENV variable
 eof method
 class IO 509
 eof? method
 class IO 509
 Epoch 642
 EPSILON (Float constant) 487
 eql? method 95
 class Array 432
 class Bignum 442
 class Float 488
 class Method 543
 class Numeric 565
 class Object 568
 class Range 599
 class String 613
 equal? method 95
 class Object 569
 ERB class 673
 erb 242, 673
 ERB::Util module 674
 erf method
 module Math 541
 erfc method
 module Math 541
 Errno constant 110
 Errno module 110, 460, 461
 Error handling *see* Exception
 Errors in book, reporting xxvii
 eruby 242–244
 in Apache 243
 see also CGI programming
 escape method
 class Regexp 600

Escaping characters *see*
 Quoting
 Etc module 675
 EUC 317, 324, 686, 689, 705
 euid method
 module Process 584
 euid= method
 module Process 584
 eval method
 module Kernel 408, 444, 520
 Event binding *see* GUI programming
 Example code, download 5
 Example printer 197
 Exception 107–115, 360
 ClassCastException 366
 EOFError 722
 in extensions 310
 FaultException 757
 handling 108
 hierarchy 109f
 IndexError 433, 609
 LoadError 222
 LocalJumpError 359
 NameError 333, 363
 raising 112, 527
 RuntimeError 113, 360
 SecurityError 397
 StandardError 108, 110, 361
 stored in \$! 334
 SystemCallError 110, 460, 521, 531
 SystemExit 180, 462, 463, 521
 testing 155
 in thread 138, 633
 ThreadError 359
 Timeout::Error 746
 TruncatedDataError 722
 TypeError 40, 280, 415, 516
 Exception class 107, 360, 461
 backtrace 461
 exception 461
 message 462
 new 461
 set_backtrace 462
 status 462
 success? 463
 to_s 463
 to_str 463
 exception method

class Exception 461
 exclude_end? method
 class Range 599
 exec method
 module Kernel 149, 506, 521
 executable? method
 class File::Stat 479
 class File 467
 executable_real? method
 class File::Stat 479
 class File 467
 Execution
 environment 393
 profiler 171
 tracing 412
 exist? method
 class File 467
 exists? method
 class File 467
 Exit status *see \$?*
 exit method
 class Thread 634, 636
 module Kernel 180, 462, 521
 module Process 584
 exit! method
 module Kernel 522
 module Process 584
 exited? method
 class Process::Status 592
 exitstatus method
 class Process::Status 592
 exp method
 module Math 541
 expand_path method
 class File 467
 expect library 676
 expect method
 class IO 676, 720
\$expect_verbose variable 676
 Expression 87–106, 338–345
 boolean 93, 341
 case 98, 343
 if 96, 343
 range as boolean 95
 substitution in string 321
 ternary 97, 343
 unless *see if*
 extconf.rb 277, 296
 see also mkmf module
 Extend Ruby 275–315, 779
 allocation 287
 building extensions 296

see also mkmf module
 call method API 309
 clone and dup 288
 create object 276, 287
 data type conversion API 280
 data type wrapping API 285
 define classes API 305
 define methods API 306
 define structures API 306
 documentation (RDoc) 207
 embedded Ruby API 304
 embedding 301
 example code 290
 exception API 310
 garbage collection 285
 initialize 276
 internal types 278
 iterator API 311
 linking 301
 memory allocation API 293
 object status API 312
 strings 280
 variable API 308, 311
 variables 283
 extend method
 class Object 383, 385, 569
 extend_object method
 class Module 411, 556
 EXTENDED (Regexp constant) 600
 Extended mode (regexp) 324
 extended method
 class Module 556
 Extending classes 26
 External iterator 53, 683
 extname method
 class File 467

F

\$F variable 178, 336
 -F pattern (Ruby option) 178, 334
 \$-F variable 335
 Factory method 36
 fail method
 module Kernel 112, 522
 FALSE constant 337
 false constant 93, 336, 341
 FalseClass class 464
 & 464
 ^ 464
 | 464

Fcntl module 509, 677
 fcntl method
 class IO 509
 FD (file descriptor) 507
 Feedback, e-mail address xxvii
 fetch method
 class Array 433
 class Hash 496
 Fibonacci series (fib_up_to) 50
 Field separator *see \$;*
 __FILE__ constant 337
 File
 associations under Windows 268
 and blocks 128
 descriptor 507
 directory operations *see Dir*
 class
 directory traversal 679
 expanding names 467, 468
 FNM_NOESCAPE 468
 including source 124, 178, 182
 lock modes 476f
 match modes 468f
 modes 504f
 open modes 472f
 opening 128
 owner 466, 475, 479, 481, 482
 pathname 470f, 503, 714
 permission 465, 474
 reading 129
 temporary 741
 tests 531
 writing 131
 File class 127, 465, 483, 681, 714
 atime 465, 475
 basename 465
 blockdev? 466
 chardev? 466
 chmod 466, 475
 chown 466, 475
 ctime 466, 475
 delete 466
 directory? 467
 dirname 467
 executable? 467
 executable_real? 467
 exist? 467
 exists? 467
 expand_path 467

extname 467	directory? 478	& 484
file? 468	executable? 479	* 484
flock 475	executable_real? 479	** 484
fnmatch 451, 468	file? 479	+ 484
fnmatch? 469	ftype 479	- 484
ftools extension 681	gid 479	-@ 484
ftype 469	grpowned? 479	/ 484
grpowned? 469	ino 479	<= 484
join 469	mode 479	<< 484
lchmod 469, 475	mtime 480	>> 484
lchown 470, 475	nlink 480	[] 485
link 470	owned? 480	^ 484
lstat 470, 476	pipe? 480	484
mtime 470, 476	rdev 480	- 484
new 128, 470	rdev_major 480	abs 485
open 54, 128, 471	rdev_minor 480	Arithmetic operations 484
owned? 471	readable? 480	Bit operations 484
path 476	readable_real? 481	div 485
pipe? 471	setgid? 481	divmod 485
readable? 471	setuid? 481	id2name 485
readable_real? 471	size 481	literal 59, 319
readlink 471	size? 481	modulo 485
rename 472	socket? 481	quo 485
setgid? 472	sticky? 481	range of 59
setuid? 472	symlink? 482	size 486
size 472	uid 482	to_f 486
size? 472	writable? 482	to_s 486
socket? 473	writable_real? 482	to_sym 486
split 473	zero? 482	zero? 486
stat 473	file? method	flatten method
sticky? 473	class File::Stat 479	class Array 433
symlink 473	class File 468	flatten! method
symlink? 473	\$FILENAME variable 336	class Array 434
truncate 473, 476	fileno method	Float class 60, 487
umask 474	class IO 509	% 487
unlink 474	FileTest module 483, 714	* 487
utime 474	FileUtils module 678, 751	** 487
writable? 474	fill method	+ 487
writable_real? 474	class Array 433	- 487
zero? 474	Find module 679	-@ 487
File Transfer Protocol <i>see</i>	find method	/ 487
Network protocols, FTP	module Enumerable 455	<= 487
File, reading 722	find_all method	== 488
File::Stat class 477	module Enumerable 455	abs 488
<= 477	find_library method	Arithmetic operations 487
atime 477	module mkmf 299, 781	ceil 488
blksize 477	Finger client 133	divmod 488
blockdev? 477	finite? method	eql? 488
blocks 478	class Float 488	finite? 488
chardev? 478	first method	floor 488
ctime 478	class Array 433	infinite? 489
dev 478	class Range 599	literal 60, 320
dev_major 478	Fixnum class 59, 484, 692	
dev_minor 478	% 484	

modulo 489
nan? 489
round 489
to_f 489
to_i 489
to_int 489
to_s 490
truncate 490
zero? 490
Float method
 module Kernel 516, 621
flock method
 class File 475
floor method
 class Float 488
 class Integer 501
 class Numeric 565
flush method
 class IO 509
FNM_xxx
 filename match constants
 468
fnmatch method
 class File 451, 468
fnmatch? method
 class File 469
for...in loop 103, 344, 672
for_fd method
 class BasicSocket 764
 class IO 504
foreach method
 class Dir 450
 class IO 131, 504
Fork see Process
fork method
 class Thread 634
 module Kernel 149, 150,
 522
 module Process 584
format method
 module Kernel 523
Forms see CGI programming,
 forms
Forms (Web) 237
 Fortran, documentation 199n
Forwardable module 680
 Forwarding 667, 680
 Fowler, Chad xxviii, 215
freeze method
 class Object 170, 394, 569
 class ThreadGroup 641
frexp method
 module Math 541
frozen? method

class Object 570
fsync method
 class IO 509
 ftools library 681
 FTP *see* Network protocols,
 FTP
 FTP site for Ruby 3
ftype method
 class File::Stat 479
 class File 469
 Funaba, Tadayoshi 390
Function *see* Method
 Function pointer 408

G

Garbage collection 369, 491,
 578, 753
 internals 285
garbage_collect method
 module GC 491
 module ObjectSpace 579
GC module 491
disable 491
enable 491
garbage_collect 491
start 491
GDBM class 682
gdbm 666, 682
 Gelernter, David 727
Gem *see* RubyGems
gem_server 220
 gemspec 224–226
 General delimited string 318
Generator class 683
 Generator library 53
 Geometry management 260
get method 699
getaddress method
 class IP Socket 770
getaddrinfo method
 class Socket 767
getc method
 class IO 510
getegid method
 module Process::Sys 594
geteuid method
 module Process::Sys 594
getgid method
 module Process::Sys 594
getgm method
 class Time 645
gethostbyaddr method
 class Socket 767

gethostbyname method
 class Socket 767
 class TCPSocket 771
gethostname method
 class Socket 767
getlocal method
 class Time 645
getnameinfo method
 class Socket 768
GetoptLong class 684
getpeername method
 class BasicSocket 764
getpgid method
 module Process 584
getpgrp method
 module Process 584
getpriority method
 module Process 585
gets method
 class IO 510
 module Kernel 335, 523
getservbyname method
 class Socket 768
getsockname method
 class BasicSocket 764
getsockopt method
 class BasicSocket 765
Getter method 29
getuid method
 module Process::Sys 594
getutc method
 class Time 645
getwd method
 class Dir 450
gid method
 class File::Stat 479
 module Process 585
gid= method
 module Process 585
GIF 260, 264
Glob *see* File, expanding names
glob method
 class Dir 451
Global variables *see* Variables
global_variables method
 module Kernel 523
gm method
 class Time 642
GMT 642
gmt? method
 class Time 645
gmt_offset method
 class Time 645
gmtime method

class Time **645**
 gmtoff method
 class Time **646**
 GNU readline **723**
 Google
 developer key **251**
 Web API **251**
 WSDL **252**
 Granger, Michael **218**
 grant_privilege method
 module Process:::GID **589**
 module Process:::UID **596**
 Graphic User Interface *see* GUI
 programming
 Greedy patterns **73**
 Greenwich Mean Time **642**
 grep method
 module Enumerable **455**
 group method
 class Thread **636**
 Grouping (regular expression)
 73
 groups method
 module Process **585, 586**
 groups= method
 module Process **585, 586**
 grpowned? method
 class File:::Stat **479**
 class File **469**
 GServer class **685**
 gsub method
 class String **74, 326, 613**
 module Kernel **523**
 gsub! method
 class String **326, 614**
 module Kernel **523**
 GUI programming **255–266,**
 747
 callback from widget **257**
 events **260**
 geometry management **260**
 scrolling **263**
 widgets **256–259**
 GZip compression **759**

H

-h, --help (Ruby option) **178**
 has_key? method
 class CGI **238**
 class Hash **496**
 has_value? method
 class Hash **496**
 Hash **45**

creating **46**
 default value **18**
 indexing **46**
 key requirements **323**
 literal **17, 322**
 as method parameter **84, 348**
 Hash class **373, 492**
 == **493**
 [] **492, 493**
 []= **493**
 => **46**
 clear **493**
 default **494**
 default= **494**
 default_proc **494**
 delete **494**
 delete_if **495**
 each **495**
 each_key **495**
 each_pair **495**
 each_value **495**
 empty? **495**
 fetch **496**
 has_key? **496**
 has_value? **496**
 include? **496**
 index **496**
 indexes **497**
 indices **497**
 invert **497**
 key? **497**
 keys **497**
 length **497**
 member? **497**
 merge **497**
 merge! **498**
 new **492**
 rehash **323, 498**
 reject **498**
 reject! **498**
 replace **373, 498**
 select **499**
 shift **499**
 size **499**
 sort **499**
 store **499**
 to_a **499**
 to_hash **499**
 to_s **499**
 update **500**
 value? **500**
 values **500**
 values_at **500**
 Hash functions **668**

hash method
 class Object **570**
 have_func method
 module mkmf **300, 781**
 have_header method
 module mkmf **299, 781**
 have_library method
 module mkmf **299, 781**
 head method **699**
 Heading, RDoc **205**
 Here document **62, 321**
 Hex notation **59, 319**
 hex method
 class String **614**
 Hintze, Clemens **329**
 HOME **182, 449, 467**
 Hook **410**
 hour method
 class Time **646**
 Howard, Ara T. **720**
 HTML *see* CGI programming
 HTML, documentation **199**
 HTTP *see* Network protocols,
 HTTP
 HTTPS protocol **709**
 Hyperlink in documentation
 204
 hypot method
 module Math **541**

L

/i regexp option **324**
 -i [extension] (Ruby option)
 178, 336
 -I directories (Ruby option)
 178, 335
 \$-I variable **336**
 \$-i variable **336**
 Ichikawa, Itaru **705**
 Iconv class **686**
 id method
 class Object **570**
 id2name method
 class Fixnum **485**
 class Symbol **631**
 Identifier
 object ID **12, 405**
 see also Variable
 IEEE floating point **487**
 -Idirectories (Ruby option)
 183
 if expression **96, 343**
 as modifier **97, 343**

IGNORECASE (Regexp constant) 600	module Enumerable 52, 120, 456	class Numeric 565
Igpay atinlay <i>see</i> Pig latin	ino method	Interactive Ruby <i>see</i> irb
in (for loop) 344	class File::Stat 479	Intern <i>see</i> Symbol
In-place edit mode 178	Input/Output <i>see</i> I/O	intern method
:include: (RDoc) 206	insert method	class String 615
include method 119	class Array 434	Internal iterator 53
class Module 355, 556	class String 615	Internet <i>see</i> Network.protocols
include? method	inspect method	Internet Mail Access Protocol
class Array 434	class Object 570 , 715	(IMAP) <i>see</i> Network
class Hash 496	class Regexp 602	protocols, IMAP
class Module 549	class Symbol 632	Interval <i>see</i> Range
class Range 599	Installation script 678, 751	Introspection <i>see</i> Reflection
class String 614	Installing Ruby 3	Inverse, matrix 694
module Enumerable 455	Instance	invert method
included method	class instance method <i>see</i>	class Hash 497
class Module 556	Object	Invoking <i>see</i> Method, calling
included_modules method	method method <i>see</i> Method	I/O class 127, 373, 503, 676,
class Module 549	variable <i>see</i> Variable	729, 735, 736
Including source files <i>see</i> File,	instance_eval method	<< 132, 507
including source	class Object 570	binmode 507
Incremental development 170	instance_method method	clone 507
Indentation 14	class Module 549 , 651	close 507
index method	instance_methods method	close_read 507
class Array 434	class Module 550	close_write 508
class Hash 496	instance_of? method	closed? 508
class String 326 , 614	class Object 571	each 508
indexes method	instance_variable_get	each_byte 130, 508
class Array 434	method	each_line 130, 509
class Hash 497	class Object 571	eof 509
Indexing	instance_variable_set	eof? 509
array 44	method	expect 676, 720
hash 46	class Object 571	fcntl 509
indices method	instance_variables method	fileno 509
class Array 434	class Object 571	flush 509
class Hash 497	Integer class 373, 501, 692	for_fd 504
infinite? method	ceil 501	foreach 131, 504
class Float 489	chr 501	fsync 509
Inheritance 27, 352	downto 102, 501	getc 510
and access control 393	floor 501	gets 510
method lookup 349, 380	integer? 501	ioctl 510
single <i>versus</i> multiple 30	next 501	isatty 510
<i>see also</i> Delegation;	round 501	lineno 510
Module, mixin	succ 502	lineno= 511
inherited method	times 102, 502	new 504
class Class 411, 445	to_i 502	open 505
initgroups method	to_int 502	pid 511
module Process 585	truncate 502	pipe 149, 505
initialize method 25, 37,	upto 102, 502	popen 148, 505
353	<i>see also</i> Fixnum, Bignum	pos 511
class Object 575	Integer method	pos= 511
initialize_copy method	module Kernel 373, 516	print 511
class Object 289, 570	integer? method	printf 512
inject method 52	class Integer 501	putc 512
		puts 512

read 506, 512
 readbytes 722
 readchar 512
 readline 512
 readlines 506, 513
 ready? 687
 reopen 373, 513
 rewind 513
 seek 513
 select 373, 507
 stat 513
 StringIO 736
 sync 514
 sync= 514
 sysopen 507
 sysread 514
 sysseek 514
 syswrite 514
 tell 515
 to_i 515
 to_io 515
 tty? 515
 ungetc 515
 wait 687
 write 515
 I/O 127–134
 binary data 131
 buffering problems 169
 see also classes File, IO,
 and Network Protocols
 io/wait library 687
 ioctl method
 class IO 510
 Iowa, Web framework 239
 IP address representation 688
 IP, IPv4, IPv6 *see* Network
 protocols
 IPAddress class 688
 IPSocket class 735, 770
 addr 770
 getaddress 770
 peeraddr 770
 recvfrom 770
 irb 6, 164, 185–196
 commands 194
 configuration 190
 extending 191
 load files into 187
 options 186f
 prompt 190, 195
 adding ri 191
 subsession 188
 tab completion 187

.irbrc, _irbrc, irb.rc,
 \$irbrc 190
 is_a? method
 class Object 571
 isatty method
 class IO 510
 isdst method
 class Time 646
 ISO 8601 date 713, 745
 issetugid method
 module Process::Sys 594
 Iterator 21, 49, 101
 on arbitrary method 672
 in extension 311
 external, internal 53, 683
 for reading files 130
 see also Block
 iterator? method
 module Kernel 524

J

JavaSpaces *see* Distributed
 Ruby
 jcode library 689
 JINI *see* Distributed Ruby
 JIS 317, 686, 689, 705
 join method
 class Array 436
 class File 469
 class Thread 137, 636
 JSP *see* eruby
 Jukebox example 25–34, 55–57,
 284–293

K

-K kcode (Ruby option) 179,
 336
 \$-K variable 336
 Kanji 705
 \$KCODE variable 324, 659, 689
 kcode method
 class Regexp 602
 Kellner, Robert 743
 Kernel module 516
 ` (backquote) 89, 148,
 516
 abort 517
 alias_method 410
 Array 516
 at_exit 517
 autoload 517
 autoload? 518
 binding 409, 444, 518
 block_given? 54, 349, 518
 callcc 448, 518
 caller 113, 412, 413, 518
 catch 105, 114, 362, 519
 chomp 519
 chomp! 519
 chop 520
 chop! 520
 eval 408, 444, 520
 exec 149, 506, 521
 exit 180, 462, 521
 exit! 522
 fail 112, 522
 Float 516, 621
 fork 149, 150, 522
 format 523
 gets 335, 523
 global_variables 523
 gsub 523
 gsub! 523
 Integer 373, 516
 iterator? 524
 lambda 56, 358, 360, 524,
 581
 load 182, 335, 398, 524
 local_variables 524
 loop 102, 524
 method_missing 349, 379
 open 134, 525, 707
 p 526
 pp 715
 print 335, 526
 printf 335, 526
 proc 358, 360, 527
 putc 527
 puts 527
 raise 112, 360, 527
 rand 527
 readline 335, 527
 readlines 528
 require 182, 335, 528, 547
 scan 528
 scanf 729
 select 528
 set_trace_func 412, 444,
 529, 749
 singleton_method_added
 411
 singleton_method_
 removed
 411
 singleton_method_
 undefined
 411

sleep 529
 split 178, 529
 sprintf 529
 srand 530
 String 516
 sub 530
 sub! 530
 syscall 530
 system 148, 530
 test 531
 throw 114, 362, 531
 trace_var 532
 trap 150, 534
 untrace_var 534
 warn 179, 336, 534
 see also Object class
 key? method
 class Hash 497
 class Thread 637
 keys method
 class Hash 497
 class Thread 637
 Keyword argument 84
 Keywords 329
 kill method
 class Thread 634, 637
 module Process 585
 kind_of? method
 class Object 572

L

-1 (Ruby option) 179, 336
 \$-1 variable 336
 lambda method
 module Kernel 56, 358,
 360, 524, 581
 last method
 class Array 436
 class Range 599
 last_match method
 class Regexp 600
 last_modified method 707
 Latent types *see* Duck typing
 Layout, source code 317
 lchmod method
 class File 469, 475
 lchown method
 class File 470, 475
 ldexp method
 module Math 541
 LDFLAGS (mkmf) 780
 Leap seconds 647n
 Leap year 98

length method
 class Array 436
 class Hash 497
 class MatchData 538
 class String 615
 class Struct 629
 Library
 Abbrev 655
 Base64 656
 Benchmark 657
 BigDecimal 658
 BigMath 658
 CGI 659
 CGI::Session 661
 Complex 662
 CSV 663
 CSV::Row 663
 Curses 664
 Date 665
 DateTime 665
 DBM 666
 Delegator 667
 Digest 668
 DL 273, 669
 DRb 670
 English 671
 Enumerator 672
 ERB 673
 expect 676
 Fcntl 677
 FileUtils 678
 Find 679
 Forwardable 680
 ffools 681
 GDBM 682
 Generator 53, 683
 GetoptLong 684
 GServer 685
 Iconv 686
 io/wait 687
 IPAddr 688
 jcode 689
 Logger 690
 Mail 691
 mathn 193, 692, 721
 Matrix 694
 mkmf 779
 Monitor 695
 monitor 142
 MonitorMixin 695
 Mutex 696
 Mutex_m 697
 net/http 136
 Net::FTP 698
 Net::HTTP 699
 Net::IMAP 701
 Net::POP3 702
 Net::SMTP 703
 Net::Telnet 704
 NKF 705
 Observable 706
 open-uri 134, 707
 Open3 708
 OpenSSL 709
 OpenStruct 710
 OptionParser 711
 ParseDate 713
 Pathname 714
 PP 715
 PrettyPrint 716
 profile 171, 717
 Profiler__ 718
 PStore 719
 PTY 720
 Queue 743
 Rational 721
 readbytes 722
 Readline 723
 readline 163, 187, 193
 resolv 724
 resolv-replace 724
 REXML 725
 Rinda 727
 RSS 728
 scanf 729
 SDBM 730
 Set 731
 Shellwords 732
 SimpleDelegator 667
 Singleton 733
 SizedQueue 743
 SOAP 734
 Socket 735
 standard 653–759
 StringIO 736
 StringScanner 737
 Sync 738
 SyncEnumerator 683
 Syslog 740
 Tempfile 741
 Test::Unit 742
 thread 141, 145
 ThreadsWait 744
 time 745
 Timeout 746
 Tk 747
 tmpdir 741, 748
 tracer 749

TSort 750
 un 751
 URI 752
 Vector 694
 WeakRef 753
 WEBrick 754
 Win32API 268, 755
 WIN32OLE 269, 756
 XMLRPC 757
 YAML 416, 535, 654, 758
 Zlib 759
see also RubyGems
 lib/ directory 296
 Linda *see* Distributed Ruby.
 Rinda
 Line continuation 317
 Line separator *see* End of line
 lineno method
 class IO 510
 lineno= method
 class IO 511
 link method
 class File 470
 List *see* Array
 RDoc 204
 list method
 class ThreadGroup 641
 class Thread 634
 module Signal 604
 listen method
 class Socket 769
 Listener *see* Observer
 Literal
 array 322
 ASCII 60, 320
 Bignum 59, 319
 Fixnum 59, 319
 Float 60, 320
 hash 322
 range 66, 322
 regular expression 68, 324
 String 61, 320
 symbol 323
 ljust method
 class String 615
 _load 415, 535
 load method 119, 124
 module Kernel 182, 335,
 398, 524
 module Marshal 414, 415,
 536
 \$LOAD_PATH variable 160, 178,
 186, 221–223, 225, 228,
 336

Local variable *see* Variable
 local method
 class Time 643
 local_variables method
 module Kernel 524
 localtime method
 class Time 646
 Locking *see* File class, flock
 Locking (file) 475
 log method
 module Math 541
 log10 method
 module Math 541
 LOGDIR 182, 449
 Logger
 system 740
 Logger class 690
 lookup_order= method
 class BasicSocket 764
 Loop 501, 502, 566
see also Iterator
 loop method 102, 344
 module Kernel 102, 524
 lstat method
 class File 470, 476
 lstrip method
 class String 615
 lstrip! method
 class String 615
 Lvalue 90, 338

M

/m regexp option 324
 Macdonald, Ian 253
 Maeda, Shugo 242
 Mail class 691
 Mailing lists 784
 :main: (RDoc) 207
 Main program 393
 main method
 class Thread 634
 MAJOR_VERSION (Marshal
 constant) 536
 MANIFEST file 301
 MANT_DIG (Float constant) 487
 map method
 module Enumerable 456
 map! method
 class Array 436
 Marshal module 414–416,
 535–536
 dump 414, 536
 limitations 535

load 414, 415, 536
 restore 536
see also YAML
 marshal_dump method 415,
 535
 marshal_load method 535
 match method
 class Regexp 69, 77, 602
 class String 616
 MatchData class 69, 77, 537,
 600, 602, 608
 [] 537
 begin 537
 captures 537
 end 538
 length 538
 offset 538
 post_match 538
 pre_match 538
 select 538
 size 538
 string 539
 to_a 539
 to_s 539
 values_at 539
see also \$~
 MatchingData class 537
 Math module 540
 acos 540
 acosh 540
 asin 540
 asinh 540
 atan 540
 atan2 540
 atanh 540
 cos 540
 cosh 540
 erf 541
 erfc 541
 exp 541
 frexp 541
 hypot 541
 ldexp 541
 log 541
 log10 541
 sin 541
 sinh 542
 sqrt 542
 tan 542
 tanh 542
 mathn library 193, 692, 721
 Matrix class 694
 Matsumoto, Yukihiro xxii, xxiii,
 xxviii, 76

Matz *see* Matsumoto, Yukihiro
MAX (Float constant) 487
max method
 module Enumerable 456
MAX_10_EXP (Float constant)
 487
MAX_EXP (Float constant) 487
maxgroups method
 module Process 586
maxgroups= method
 module Process 586
mbox (e-mail file) 691
MD5 hash 668
mday method
 class Time 646
member? method
 class Hash 497
 class Range 599
 module Enumerable 456
members method
 class Struct 627, 629
merge method
 class Hash 497
merge! method
 class Hash 498
Message
 receiver 13
 sending 12, 28
Message box, Windows 273
message method
 class Exception 462
Meta character 60, 320
meta method 707
Metaclass 380, 382
Metaprogramming *see*
 Reflection
Method 85
 access control 37, 551, 553,
 558, 559
 aliasing 351
 ambiguity 123
 arguments 347
 array parameter 83
 block as parameter 80
 call, in extension 309
 calling 81, 348
 calling dynamically 407
 class 34, 385
 defining 79, 80, 345
 in extension 306
 getter 29
 instance 12
 with iterator 408
 keyword argument 84

module 118
naming 16, 79, 346
nested definition 80
nested method definition 346
object 408, 572
as operator 88
parameters 79, 80
private 81
renaming 410
return value 80, 82, 350
setter 31, 92
vs. variable name 329
variable-length arguments 80
Method class 408, 543, 555
 == 543
 [] 543
 arity 543
 call 408, 543
 eq? 543
 to_proc 544
 unbind 544
Method module 651
method method
 class Object 543, 572
method_added method
 class Module 411, 557
method_defined? method
 class Module 550
method_missing method
 class Object 572
 module Kernel 349, 379
method_removed method
 class Module 411, 557
method_undefined method
 class Module 411, 557
methods method
 class Object 405, 573
Meyer, Bertrand 32
Microsoft Windows 267–274
accessing API 268, 755
automation 269, 756
file associations 268
installing Ruby 4, 784
message box 273
printing under 268
running Ruby 268
scripting *see* automation
 (above)
MIN (Float constant) 487
min method
 class Time 646
 module Enumerable 456
MIN_10_EXP (Float constant)
 487

MIN_EXP (Float constant) 487
MINOR_VERSION (Marshal
 constant) 536
Mirroring, using CVSup 6
MixedCase 16, 328
Mixin *see* Module
mkdir method
 class Dir 451
mkmf module 779
 building extensions with 296
 create_makefile 296, 781
 dir_config 297, 781
 enable_config 781
 find_library 299, 781
 have_func 300, 781
 have_header 299, 781
 have_library 299, 781
mkmf library 779
mktime method
 class Time 643
mod_ruby 247
 safe level 398
mode method
 class File::Stat 479
Module 117–125
 constant 118
 creating extension *see*
 Extend Ruby
 defining 354
 function 355
 include 119
 instance variable 122
 load 119
 as mixin 118, 355, 383
 as namespace 117
 naming 16
 require 119
 wrap 398
Module class 545
 <, <=, >, >= 546
 <=> 546
 == 546
alias_method 554
ancestors 382, 405, 547
append_features 554
attr 554
attr_accessor 555
attr_reader 555
attr_writer 555
autoload 547
autoload? 547
class_eval 547
class_variables 548
clone 548

const_defined? 548
 const_get 548
 const_missing 548
 const_set 549
 constants 545, 549
 define_method 555
 extend_object 411, 556
 extended 556
 include 355, 556
 include? 549
 included 556
 included_modules 549
 instance_method 549, 651
 instance_methods 550
 method_added 411, 557
 method_defined? 550
 method_removed 411, 557
 method_undefined 411,
 557
 module_eval 551
 module_function 355, 558
 name 551
 nesting 545
 new 546
 private 558
 private_class_method
 35, 551
 private_instance_
 methods
 552
 private_method_defined?
 552
 protected 559
 protected_instance_
 methods
 552
 protected_method_
 defined?
 552
 public 559
 public_class_method 553
 public_instance_methods
 553
 public_method_defined?
 553
 remove_class_variable
 559
 remove_const 559
 remove_method 559
 undef_method 559
 Module module
 define_method 360
 module_eval method
 class Module 551

module_function method
 class Module 355, 558
 Modules
 list of methods 426
 Abbrev 655, 723
 Base64 656
 Benchmark 170, 409, 657
 BigMath 658
 Comparable 119, 447
 Config 183
 Curses 664
 Date 642
 Digest 668
 DL 669
 DRbUndumped 670
 Enumerable 52, 120, 454,
 672, 683
 Enumerator 672
 ERB::Util 674
 Errno 110, 460, 461
 Etc 675
 Fcntl 509, 677
 FileTest 483, 714
 FileUtils 678, 751
 Find 679
 Forwardable 680
 GC 491
 Kernel 516
 Marshal 535
 Math 540
 Method 651
 mkmf 779
 Monitor 695
 MonitorMixin 144, 695
 Mutex_m 141, 697
 NKF 705
 ObjectSpace 578
 Observable 706
 Open3 506, 708
 ParseDate 642, 713
 Process 150, 583, 594
 Process::GID 589, 594
 Process::Sys 594
 Process::UID 594, 596
 Profiler 717
 Profiler_ 718
 PTY 720
 Readline 723
 REXML 725
 Rinda 727
 RSS 728
 Session 720
 Shellwords 732
 Signal 534, 604

SingleForwardable 680
 Singleton 733
 SOAP 734
 Sync 141
 Timeout 746
 TSort 750
 WEBrick 754
 XMLRPC 757
 Zlib 759
 modulo method
 class Bignum 442
 class Fixnum 485
 class Float 489
 class Numeric 565
 mon method
 class Time 646
 Monitor class 142, 144, 738,
 743
 Monitor module 695
 monitor library 142
 MonitorMixin module 144,
 695
 month method
 class Time 646
 mswin32 267
 mtime method
 class File::Stat 480
 class File 470, 476
 MULTILINE (Regexp constant)
 600
 Multiline mode (regexp) 324
 Multiple inheritance 30
 see also Module, mixin
 Multithreading see Thread
 Music on hold 783
 Mutex class 696, 697
 Mutex_m module 141, 697
 Mutual exclusion see Thread,
 synchronization
 "My Way" 27

N

-n (Ruby option) 179
 Nagai, Hidetoshi 589
 Nakada, Nobuyoshi 705
 name method
 class Module 551
 Namespace see Module
 Naming conventions 16, 328
 file pathnames 503
 method names 79
 Test::Unit 161
 nan? method

class **Float** 489
Native thread *see Thread*
ncurses *see Curses*
ndbm 666
 Nested assignment 92
 nesting method
 class **Module** 545
 net/http library 136
 Net::FTP class 698
 Net::HTTP class 699, 752
 Net::IMAP class 701
 Net::POP3 class 702
 Net::SMTP class 703
 Net::Telnet class 704
 Network protocols
 DNS 724
 domain socket 776
 finger 133
 ftp 698, 707, 752
 secure 709
 generic server for 685
 HTTP 699, 707, 752
 HTTPS 709, 752
 IMAP 701
 IP 770
 IP address representation 688
 IPv4/IPv6 688
 LDAP 752
 POP 702
 server 773, 777
 SMTP 703
 socket 133, 735, 764, 766
 SOCKS 772
 TCP 771
 telnet 704
 UDP 774
 new method
 see also Constructor
 new method
 class **Array** 427
 class **Class** 353, 445, 446
 class **Dir** 451
 class **Exception** 461
 class **File** 128, 470
 class **Hash** 492
 class **IO** 504
 class **Module** 546
 class **Proc** 524, 580
 class **Range** 598
 class **Regexp** 600
 class **Socket** 768
 class **SOCKSSocket** 772
 class **String** 606

class **Struct** 626, 627
 class **TCPServer** 773
 class **TCPSocket** 771
 class **ThreadGroup** 640
 class **Thread** 136, 634
 class **Time** 643
 class **UDPSocket** 774
 class **UNIXServer** 777
 class **UNIXSocket** 776
 Newline (\n) 14, 321
 Newsgroup 784
 next 104, 345, 357
 next method
 class **Integer** 501
 class **String** 616
 next! method
 class **String** 616
 nfk method
 module NKF 705
 NIL constant 337
 nil constant 16, 93, 336, 341
 nil? method
 class **NilClass** 561
 class **Object** 573
 NilClass class 561
 & 561
 ^ 561
 | 561
 nil? 561
 to_a 561
 to_f 561
 to_i 561
 to_s 561
 nitems method
 class **Array** 436
 NKF module 705
 nfk 705
 nlink method
 class **File::Stat** 480
 No-wait mode I/O 687
 :nodoc: (RDoc) 206
 nonzero? method
 class **Numeric** 565
 not (logical not) 94, 341
 Notation xxix
 binary, decimal, hex, octal 59, 319
 notify_observers method
 module Observable 706
 :notnew: (RDoc) 206
 now method
 class **Time** 643
 NTP (Network Time Protocol) 704

Numbers, unifying 692
 Numeric class 562, 662
 +@ 562
 -@ 562
 <=> 562
 abs 562
 ceil 562
 coerce 374, 562
 div 563
 divmod 565
 eql? 565
 floor 565
 integer? 565
 mathn 692
 modulo 565
 nonzero? 565
 quo 566
 Rational 721
 remainder 566
 round 566
 step 102, 566
 to_int 566
 truncate 566
 zero? 566

O

/o regexp option 324
 OBJ_FREEZE 312
 OBJ_FROZEN 313
 OBJ_TAINT 312
 OBJ_TAINTED 312
 Object 12
 aliasing 40, 169, 330
 allocation 287
 creation 25, 353, 410
 extending 382, 385
 finalizer 578
 freezing 394
 ID 12, 405
 immediate 279, 404, 484
 listing active 404
 listing methods in 405
 object_id 578
 persistence 719
 tainting 399
 Object class 29, 393, 567
 == 567
 ==== 567
 =~ 567
 __id__ 567
 __send__ 567
 class 370, 567
 clone 288, 568

display 568
 dup 288, 568
 eql? 568
 equal? 569
 extend 383, 385, 569
 freeze 170, 394, 569
 frozen? 570
 hash 570
 id 570
 initialize 575
 initialize_copy 289, 570
 inspect 570, 715
 instance_eval 570
 instance_of? 571
 instance_variable_get 571
 instance_variable_set 571
 instance_variables 571
 is_a? 571
 kind_of? 572
 method 543, 572
 method_missing 572
 methods 405, 573
 nil? 573
 object_id 573
 private_methods 573
 protected_methods 573
 public_methods 574
 remove_instance_variable 576
 respond_to? 405, 574
 send 574
 singleton_method_added 576
 singleton_method_removed 577
 singleton_method_undefined 577
 singleton_methods 574
 taint 575
 tainted? 575
 to_a 575
 to_s 26, 575
 to_str 280
 type 370, 575
 untaint 575
see also Kernel module
 Object-oriented terminology 11
 object_id method
 class Object 573

ObjectSpace module 578
 _id2ref 578
 define_finalizer 578
 each_object 382, 404, 406, 578
 garbage_collect 579
 undefine_finalizer 579
 Observable module 706
 add_observer 706
 changed 706
 changed? 706
 count_observers 706
 delete_observer 706
 delete_observers 706
 notify_observers 706
 Observer pattern 706
 oct method
 class String 616
 Octal notation 59, 319
 offset method
 class MatchData 538
 OLE *see* Microsoft Windows automation
 olegem.rb 272
 once example 391
 Once option (regexp) 324
 One-Click Installer 4, 267, 784
 OO *see* Object-oriented
 open method
 class Dir 452
 class File 54, 128, 471
 class IO 505
 class Socket 768
 class SOCKSSocket 772
 class TCPServer 773
 class TCPSocket 771
 class UDPSocket 774
 class UNIXServer 777
 class UNIXSocket 776
 module Kernel 134, 525, 707
 open-uri library 134, 707
 Open3 module 506, 708
 OpenSSL library 709
 OPENSSL_CONF 182
 OpenStruct class 626, 710
 Operating system errors 460
 Operator
 as method call 88, 350
 precedence 339
 Optimizing *see* Performance
 Option, command line *see* Command line
 OptionParser class 711

options method
 class Regexp 602
 or (logical or) 94, 341
 owned? method
 class File::Stat 480
 class File 471
 Ownership, file *see* File, owner

P
 -p (Ruby option) 179, 336
 p method
 module Kernel 526
 \$-p variable 336
 pack method
 class Array 131, 435, 436
 pack_sockaddr_in method
 class Socket 768
 pack_sockaddr_un method
 class Socket 768
 Packaging *see* RubyGems
 pair method
 class Socket 768
 Paragraph mode 178
 Parallel assignment 91, 340
 Parameter
 default 79
 to block 21
 params method
 class CGI 237
 Parent-child 27
 Parse error 167
 ParseDate module 642, 713
 parsedate 643
 see also Time class and library
 parsedate method
 module ParseDate 643
 partition method
 module Enumerable 457
 pass method
 class Thread 635
 PATH 179
 path method
 class Dir 452
 class File 476
 class UNIXSocket 776
 Pathname *see* File, pathname
 Pathname class 714
 Pattern *see* Regular expression
 Patterns
 factory 36
 observer 706
 singleton 35, 733

peeraddr method class IPSocket 770 class UNIXSocket 776	Pre-defined variables <i>see</i> Variables	module Kernel 358, 360, 527
Performance 170, 369, 657 caching method values 390 CGI 247 dynamic method invocation 409 profiling 171, 717, 718 windows automation 272	pre_match method class MatchData 538	return from 360 and safe level 399
Perl/Tk <i>see</i> GUI programming	Precedence do...end vs {} 168, 356 of operators 339	Process 147–150
Perlisms 24, 76, 100	Pretty printing 715, 716	block 150
Permission <i>see</i> File, permission	pretty_print method 715	creating 147, 503, 506, 525, 708
Persistent object storage 719	PrettyPrint class 526, 715, 716	exec 521
PHP <i>see</i> eruby	Print under Windows 268	ID (<i>see also</i> \$\$) 511
PI (Math constant) 540	print method class IO 511	priority 585, 587
pid method class IO 511 class Process::Status 592 module Process 586	module Kernel 335, 526	Ruby subprocess 149, 150, 503, 506, 708
Pig latin 148, 259	printf method class IO 512	setting name 335
Pipe <i>see</i> IO.pipe, IO.popen	module Kernel 335, 526	termination 150, 180, 517, 522, 584, 587
pipe method class IO 149, 505	PRIOR_PGRP (Process constant) 583	times 630
pipe? method class File::Stat 480	PRIOR_PROCESS (Process constant) 583	Process module 150, 583, 594
class File 471	PRIOR_USER (Process constant) 583	abort 583
pop method class Array 437	priority method class Thread 637	detach 583
popen method class IO 148, 505	priority= method class Thread 638	egid 584
pos method class Dir 452	Private <i>see</i> Access control	egid= 584
class IO 511	private method 38	euid 584
pos= method class Dir 453	class Module 558	euid= 584
class IO 511	private_class_method method	exit 584
POSIX character classes 71	class Module 35, 551	exit! 584
error codes 110	private_instance_methods method	fork 584
POSIXLY_CORRECT 684	class Module 552	getpgid 584
Post Office Protocol (POP) <i>see</i> Network protocols, POP	private_method_defined? method	getpgrp 584
post method 699	class Module 552	getpriority 585
post_match method class MatchData 538	private_methods method class Object 573	gid 585
PP class 715	Proc class 56, 81, 357, 359, 373, 544, 555, 580	gid= 585
pp method module Kernel 715	== 580	groups 585, 586
ppid method module Process 586	[] 580	groups= 585, 586
Pragmatic Programmer e-mail address xxvii	arity 580	initgroups 585
	binding 581	kill 585
	call 581	maxgroups 586
	new 524, 580	maxgroups= 586
	to_proc 582	pid 586
	to_s 582	ppid 586
	proc method	setpgid 586
		setpgrp 586
		setpriority 587
		setsid 587
		times 587, 630
		uid 587
		uid= 587
		wait 149, 587
		wait2 588
		waitall 587
		waitpid 588
		waitpid2 588
		Process::GID module 589, 594

change_privilege 589
 eid 589
 eid= 589
 grant_privilege 589
 re_exchange 589
 re_exchangeable? 589
 rid 589
 sid_available? 589
 switch 590
Process::Status class 150,
 587, 588, 591
 & 591
 == 591
 >> 591
 coredump? 591
 exited? 592
 exitstatus 592
 pid 592
 signaled? 592
 stopped? 592
 stopsig 592
 success? 592
 termsig 593
 to_i 593
 to_int 593
 to_s 593
Process::Sys module 594
 getegid 594
 geteuid 594
 getgid 594
 getuid 594
 issetugid 594
 setegid 594
 seteuid 594
 setgid 594
 setregid 594
 setresgid 595
 setresuid 595
 setreuid 595
 setrgid 595
 setruid 595
 setuid 595
Process::UID module 594,
 596
 change_privilege 596
 eid 596
 eid= 596
 grant_privilege 596
 re_exchange 596
 re_exchangeable? 596
 rid 596
 sid_available? 596
 switch 596
 profile library 171, 717

Profiler 171
 Profiler module 717
 Profiler__ module 718
 Program *see* Process
 Protected *see* Access control
 protected method 38
 class Module 559
 protected_instance_ methods
 method
 class Module 552
 protected_method_defined?
 method
 class Module 552
 protected_methods method
 class Object 573
 Protocols 372
 Pseudo terminal 720
PStore class 719
PTY module 720
 Public *see* Access control
 public method 38
 class Module 559
 public_class_method method
 class Module 553
 public_instance_methods
 method
 class Module 553
 public_method_defined?
 method
 class Module 553
 public_methods method
 class Object 574
 Publish/subscribe 706
 push method
 class Array 437
 putc method
 class IO 512
 module Kernel 527
 puts method
 class IO 512
 module Kernel 527
 pwd method
 class Dir 452

Q

qdbm 666
Queue class 141, 145, 743
 quo method
 class Bignum 442
 class Fixnum 485
 class Numeric 566
 quote method

class Regexp 601
Quoting
 characters in regexp 70, 600
 URLs and HTML 236

R

-r (Ruby option) 751
 -r library (Ruby option)
 179
 Race condition 137
 RADIX (Float constant) 487
 Radix *see* to_s methods,
 Kernel.Integer,
 String#to_i
 Rails, Web framework 253
 raise method
 class Thread 638
 module Kernel 112, 360,
 527
 Rake 229
 Rake (build tool) 216
 rand method
 module Kernel 527
 Range 66
 as condition 68, 95, 100, 342
 as interval 68
 literal 66, 322
 as sequence 66
Range class 67, 322, 597
 == 598
 === 598
 begin 598
 each 598
 end 598
 eql? 599
 exclude_end? 599
 first 599
 include? 599
 last 599
 member? 599
 new 598
 step 599
 Rank, matrix 694
 rassoc method
 class Array 437
Rational class 692, 721
 rb_apply 309
 rb_ary_entry 314
 rb_ary_new 313
 rb_ary_new2 313
 rb_ary_new3 313
 rb_ary_new4 313
 rb_ary_pop 313
 rb_ary_push 313

rb_ary_shift 313
 rb_ary_store 313
 rb_ary_unshift 314
 rb_block_given_p 311
 rb_bug 310
 rb_call_super 309
 rb_catch 311
 rb_create_new_instance
 309
 rb_cv_get 312
 rb_cv_set 312
 rb_cvar_defined 312
 rb_cvar_get 312
 rb_cvar_set 312
 rb_define_alias 307
 rb_define_alloc_func 307
 rb_define_attr 309
 rb_define_class 305
 rb_define_class_under 305
 rb_define_class_variable
 308
 rb_define_const 308
 rb_define_global_const
 308
 rb_define_global_function
 307
 rb_define_hooked_variable
 308
 rb_define_method 307
 rb_define_module 305
 rb_define_module_function
 307
 rb_define_module_under
 306
 rb_define_READONLY_
 variable
 308
 rb_define_singleton_
 method
 307
 rb_define_VARIABLE 308
 rb_define_VIRTUAL_
 variable
 308
 rb_each 311
 rb_ensure 310
 rb_exit 310
 rb_extend_object 306
 rb_fatal 310
 rb_funcall 309
 rb_funcall2 309
 rb_funcall3 309
 rb_global_VARIABLE 309
 rb_gv_get 312

rb_gv_set 312
 rb_hash_aref 314
 rb_hash_aset 314
 rb_hash_new 314
 rb_id2name 309
 rb_include_module 306
 rb_intern 309
 rb_iter_break 311
 rb_iterate 311
 rb_iv_get 311
 rb_iv_set 312
 rb_ivar_get 312
 rb_ivar_set 312
 rb_load_file 304
 rb_notimplement 310
 rb_obj_is_INSTANCE_OF 314
 rb_obj_is_KIND_OF 314
 rb_protect 310
 rb_raise 310
 rb_require 306
 rb_rescue 310
 rb_respond_to 314
 rb_safe_level 313
 rb_scan_args 307
 rb_secure 313
 rb_set_SAFE_Level 313
 rb_str_cat 314
 rb_str_concat 314
 rb_str_dup 314
 rb_str_new 314
 rb_str_new2 314
 rb_str_split 314
 rb_struct_aref 306
 rb_struct_aset 306
 rb_struct_define 306
 rb_struct_new 306
 rb_sys_fail 310
 rb_thread_create 314
 rb_throw 311
 rb_undef_method 307
 rb_warn 311
 rb_warning 311
 rb_yield 311
 rbconfig.rb 183
 rbconfig.rb *see* Config
 module
 rdev method
 class File::Stat 480
 rdev_major method
 class File::Stat 480
 rdev_minor method
 class File::Stat 480
 RDoc 8, 199–212
 C extensions 207

:call-seq: 206, 209
 comment format 199–205
 :doc: 206
 Document-class: 209
 Document-method: 209
 documentation modifiers
 205
 embedding in Ruby 199
 :enddoc: 207
 heading 205
 hyperlink 204
 :include: 206
 lists 204
 :main: 207
 :nodoc: 206
 :notnew: 206
 including README 211
 generate ri documentation
 212
 for RubyGems 219
 rules 205
 running 211
 :startdoc: 207
 :stopdoc: 207
 templates 240
 :title: 207
 yield parameters 205
 :yields: 206
 RDoc::usage method 213
 RDoc::usage_no_exit method
 213
 rdtool 318
 re_exchange method
 module Process:::GID 589
 module Process:::UID 596
 re_exchangeable? method
 module Process:::GID 589
 module Process:::UID 596
 read method
 class Dir 453
 class IO 506, 512
 readable? method
 class File::Stat 480
 class File 471
 readable_real? method
 class File::Stat 481
 class File 471
 readbytes library 722
 readbytes method
 class IO 722
 readchar method
 class IO 512
 Readline module 723
 readline library 163, 187, 193

<p>readline method class I0 512 module Kernel 335, 527</p> <p>readlines method class I0 506, 513 module Kernel 528</p> <p>readlink method class File 471</p> <p>README 211</p> <p>ready? method class I0 687</p> <p>REALLOC_N 294</p> <p>Really Simple Syndication 728</p> <p>Receiver 13, 81, 349, 379</p> <p>Record separator <i>see \$/</i></p> <p>recv method class BasicSocket 765</p> <p>recvfrom method class IPSocket 770 class Socket 769 class UDPSocket 775 class UNIXSocket 776</p> <p>redo 104, 345</p> <p>Reference to object 39 weak 753</p> <p>Reflection 403–414 callbacks 411</p> <p>Regexp class 76, 600 == 601 == 601 =~ 601 ~ 601 casefold? 602 compile 600 escape 600 inspect 602 kcode 602 last_match 600 match 69, 77, 602 new 600 options 602 quote 601 source 602 to_s 603</p> <p>Regular expression 68–77, 324–328 alternation 73 anchor 70 character class 71 as condition 342 extensions 326 greedy 73 grouping 73</p>	<p>literal 68, 324</p> <p>nested 327</p> <p>object-oriented 76</p> <p>options 324, 327, 600</p> <p>pattern match variables 334</p> <p>quoting within 70</p> <p>repetition 72</p> <p>substitution 74, 614</p> <p>rehash method class Hash 323, 498</p> <p>reject method class Hash 498 module Enumerable 437, 457</p> <p>reject! method class Array 437 class Hash 498</p> <p>remainder method class Bignum 443 class Numeric 566</p> <p>Remote Procedure Call <i>see</i> Distributed Ruby, SOAP, XMLRPC</p> <p>remove_class_variable method class Module 559</p> <p>remove_const method class Module 559</p> <p>remove_instance_variable method class Object 576</p> <p>remove_method method class Module 559</p> <p>rename method class File 472</p> <p>reopen method class I0 373, 513</p> <p>replace method class Array 437 class Hash 373, 498 class String 616</p> <p>require method 119, 124 loading extensions 278 module Kernel 182, 335, 528, 547</p> <p>require_gem 220</p> <p>rescue 108, 361, 460</p> <p>Reserved words 329</p> <p>resolv library 724</p> <p>resolv-replace library 724</p> <p>respond_to? method class Object 405, 574</p> <p>restore method module Marshal 536</p>	<p>retry in exceptions 112, 113, 362 in loops 105, 345</p> <p>return from block 359 from lambda/proc 360 from Proc 359 <i>see also</i> Method, return value</p> <p>reverse method class Array 437 class String 616</p> <p>reverse! method class Array 437 class String 616</p> <p>reverse_each method class Array 438</p> <p>rewind method class Dir 453 class I0 513</p> <p>REXML module 725</p> <p>RFC 2045 (base 64) 656</p> <p>RFC 2396 (URI) 752</p> <p>RFC 2616 (HTTP) 745</p> <p>RFC 2882 (e-mail) 745</p> <p>.rhtml (eruby) 243</p> <p>RI 9, 214</p> <p>ri 8, 199–212 add to irb 191 directories 212 sample output 203 <i>see also</i> RDoc</p> <p>Rich Site Summary 728</p> <p>rid method module Process::GID 589 module Process::UID 596</p> <p>Rinda module 727</p> <p>rinda <i>see</i> Distributed Ruby</p> <p>rindex method class Array 438 class String 616</p> <p>RIPEMD-160 hash 668</p> <p>rjust method class String 617</p> <p>rmdir method class Dir 452</p> <p>RMI <i>see</i> Distributed Ruby</p> <p>Roll, log files 690</p> <p>Roman numerals 151 example 372</p> <p>round method class Float 489 class Integer 501 class Numeric 566</p>
---	---	---

ROUNDS (Float constant) 487
 RPM installation 3
 RSS module 728
 RSTRING macro 280
 rstrip method
 class String 617
 rstrip! method
 class String 617
 rtags 196
 Ruby
 debugger 163
 distributed 417–418
 download 784
 embed in application 301
 installing 3, 298
 language reference 317–363
 and Perl 24, 76, 100
 versions xxv
 Web sites xxvii, 783
 ports to Windows 267
 ruby (mkrf) 780
 Ruby Documentation Project 9, 784
 Ruby mode (emacs) 165
 Ruby On Rails 253
 Ruby Production Archive (RPA) 784
 ruby-doc.org 9
 ruby-mode.el 165
 ruby.exe and rubyw.exe 268
 ruby_finalize 304
 ruby_init 304
 ruby_init_loadpath 304
 ruby_options 304
 ruby_run 304
 ruby_script 304
 RUBY_TCL_DLL 182
 RUBY_TK_DLL 182
 RUBY_PLATFORM constant 337
 RUBY_PLATFORM variable 228
 RUBY_RELEASE_DATE constant 337
 RUBY_VERSION constant 337
 RubyForge 229, 784
 RubyGarden 784
 RubyGems 215–233
 creating 223
 documentation 219
 extensions 227
 gem_server 220
 gemspec 224–226
 installing applications 216
 installing library 218
 installing RubyGems 216

package layout 223
 repository 784
 require_gem 220
 stub 222
 test on install 217
 versioning 217, 218f, 221
 RUBYLIB 182, 183, 401
 RUBYLIB_PREFIX 182
 RUBYOPT 182, 401
 RUBYPATH 179, 182
 RUBYSHELL 182, 521
 Rule, RDoc 205
 run method
 class Thread 638
 Runtime Type Information (RTTI) *see* Reflection
 Rvalue 90, 338

S

-S (Ruby option) 179
 -s (Ruby option) 179
 \$SAFE variable 179, 313, 336, 398, 575, 673
 Safe level 397–400
 in extensions 312
 list of constraints 401f
 and proc 399
 setting using -T 179
 and tainting 399
 safe_level method
 class Thread 639
 SafeStringValue 313
 SafeStringValue method 281
 Sandbox 398, 399, *see* Safe level
 chroot 450
 scan method
 class String 64, 65, 326, 617, 737
 module Kernel 528
 scanf library 729
 scanf method
 class Array 729
 class String 729
 module Kernel 729
 Scheduler, thread 140
 Schneiker, Conrad 83n
 Schwartz, Randal 458
 Schwartzian transform 458
 Scope of variables 105, 330
 Screen output *see* Curses
 SCRIPT_LINES__ constant 337, 414, 528

SDBM class 730
 sdbm 730
 Search path 183, 298
 sec method
 class Time 647
 seek method
 class Dir 453
 class IO 513
 Seki, Masatoshi 417
 select method
 class Hash 499
 class IO 373, 507
 class MatchData 538
 module Enumerable 457
 module Kernel 528
 self variable 81, 122, 337, 349, 379
 in class definition 387
 Semaphore *see* Thread, synchronization
 Send message 12, 28
 send method
 class BasicSocket 765
 class Object 574
 class UDPSocket 775
 Sequence *see* Range
 Serialization *see* Marshal
 Server 685
 Session *see* CGI programming, session
 Session module 720
 Session leader 587
 Session, HTTP 246
 Set class 428, 430, 731
 Set operations *see* Array class
 set_backtrace method
 class Exception 462
 set_trace_func method
 module Kernel 412, 444, 529, 749
 setegid method
 module Process::Sys 594
 seteuid method
 module Process::Sys 594
 setgid, setuid 398
 setgid method
 module Process::Sys 594
 setgid? method
 class File::Stat 481
 class File 472
 setpgid method
 module Process 586
 setpgrp method
 module Process 586

setpriority method	Simple Mail Transfer Protocol	slice method
module Process 587	<i>see</i> Network protocols, SMTP	class Array 438
setregid method	Simple Object Access protocol	class String 618
module Process::Sys 594	<i>see</i> SOAP	slice! method
setresgid method	SimpleDelegator class 667	class Array 439
module Process::Sys 595	sin method	class String 618
setresuid method	module Math 541	Smalltalk 12n, 382
module Process::Sys 595	Sinatra, Frank 27	SMTP <i>see</i> Network protocols, SMTP
setreuid method	Single inheritance 30	SOAP 249, 418, 734
module Process::Sys 595	Single-quoted string 61, 320	SOAP module 734
setrgid method	SingleForwardable module	Socket <i>see</i> Network protocols
module Process::Sys 595	680	Socket class 735, 766
setruid method	Singleton module 733	accept 769
module Process::Sys 595	Singleton class 382	bind 769
setsid method	Singleton pattern 35, 733	connect 769
module Process 587	singleton_method_added	getaddrinfo 767
setsockopt method	method	gethostbyaddr 767
class BasicSocket 765	class Object 576	gethostbyname 767
Setter method <i>see</i> Method,	module Kernel 411	gethostname 767
setter	singleton_method_removed	getnameinfo 768
setuid method	method	getservbyname 768
module Process::Sys 595	class Object 577	listen 769
setuid? method	module Kernel 411	new 768
class File::Stat 481	singleton_method_	open 768
class File 472	undefined	pack_sockaddr_in 768
setup method 158	method	pack_sockaddr_un 768
SHA1/2 hash 668	class Object 577	pair 768
Shallow copy 568	singledown_method_	recvfrom 769
Shared library, accessing 669	undefined	socketpair 768
Shebang (#!) 7	method	sysaccept 769
SHELL 182	class Object 577	unpack_sockaddr_in 769
Shell glob <i>see</i> File, expanding	singledown_methods	unpack_sockaddr_un 769
names	method	socket? method
Shellwords module 732	class Object 574	class File::Stat 481
shift method	sinh method	class File 473
class Array 438	module Math 542	socketpair method
class Hash 499	site_ruby directory 183	class Socket 768
shutdown method	size method	SOCKS <i>see</i> Network protocols
class BasicSocket 765	class Array 438	SOCKSSocket class 735, 772
sid_available? method	class Bignum 443	close 772
module Process::GID 589	class File::Stat 481	new 772
module Process::UID 596	class File 472	open 772
SIGALRM 529	class Fixnum 486	Sort
SIGCLD 150	class Hash 499	topological 750
Signal	class MatchData 538	sort method
handling 150	class String 618	class Array 439
sending 585	class Struct 629	class Hash 499
<i>see also</i> trap method	size? method	module Enumerable 457
Signal module 534, 604	class File::Stat 481	Schwartzian transform 458
list 604	class File 472	sort! method
trap 605	SizedQueue class 743	class Array 439
signaled? method	SJIS 317, 324, 689	sort_by method
class Process::Status 592	sleep method	module Enumerable 457
	module Kernel 529	

Source code	\$stdout variable 335	concat 611
layout 317	step method	count 611
reflecting on 413	class Numeric 102, 566	crypt 611
Source code from book 5	class Range 599	delete 611, 689
source method	Stephenson, Neal 177n	delete! 612, 689
class Regexp 602	sticky? method	downcase 612
Spaceship <i>see</i> <=>	class File::Stat 481	downcase! 612
Spawn <i>see</i> Process, creating	class File 473	dump 612
spawn method 720	stiff, why the lucky 654	each 612
split method	stop method	each_byte 613
class File 473	class Thread 635	each_line 613
class String 63, 619	stop? method	empty? 613
module Kernel 178, 529	class Thread 639	eql? 613
sprintf method	:stopdoc: (RDoc) 207	gsub 74, 326, 613
field types 532	stopped? method	gsub! 326, 614
flag characters 531	class Process::Status 592	hex 614
module Kernel 529	stopsig method	include? 614
sqrt method	class Process::Status 592	index 326, 614
module Math 542	store method	insert 615
squeeze method	class Hash 499	intern 615
class String 64, 619, 689	strftime method	length 615
squeeze! method	class Time 647	ljust 615
class String 620, 689	String 61	lstrip 615
srand method	#{} 61	lstrip! 615
module Kernel 530	%... delimiters 318	match 616
srcdir (mkmf) 780	control characters \n etc.	new 606
Stack	321	next 616
execution <i>see</i> caller	conversion for output 131,	next! 616
method	526	oct 616
operations <i>see</i> Array class	expression interpolation 15	replace 616
unwinding 110, 114, 361	here document 62, 321	reverse 616
Stack frame 163	literal 14, 61, 320	reverse! 616
Standard Library 653–759	concatenation 321	rindex 616
start method	String class 61, 320, 374, 606,	rjust 617
class Thread 635	689, 729	rstrip 617
module GC 491	% 606	rstrip! 617
:startdoc: (RDoc) 207	* 607	scan 64, 65, 326, 617, 737
stat method	+ 607	scanf 729
class File 473	<= 607	size 618
class IO 513	<< 607	slice 618
Statement modifier	== 607	slice! 618
if/unless 97, 343	=== 607	split 63, 619
while/until 100, 345	=~ 608	squeeze 64, 619, 689
Static linking 301	[] 608	squeeze! 620, 689
Static method <i>see</i> Class, method	[]= 609	strip 620
Static typing <i>see</i> Duck typing	~ 609	strip! 620
status method 707	capitalize 609	sub 74, 620
class Exception 462	capitalize! 609	sub! 620
class Thread 639	casecmp 610	succ 620, 689
STDERR constant 337, 534	center 610	succ! 621, 689
\$stderr variable 335	chomp 63, 610	sum 621
STDIN constant 337, 525	chomp! 610	swapcase 621
\$stdin variable 335	chop 610, 689	swapcase! 621
STDOUT constant 337, 525, 526	chop! 611, 689	to_f 621

to_i 622
 to_s 622
 to_str 622
 to_sym 622
 tr 622, 689
 tr! 623, 689
 tr_s 623, 689
 tr_s! 623, 689
 unpack 623
 upcase 623
 upcase! 625
 upto 625
String method
 module Kernel 516
string method
 class MatchData 539
StringIO class 132, 736
StringScanner class 737
StringValue method 280
StringValuePtr method 281
strip method
 class String 620
strip! method
 class String 620
Struct class 626
 == 627
 [] 627, 628
 []= 628
 each 628
 each_pair 628
 length 629
 members 627, 629
 new 626, 627
 OpenStruct 710
 size 629
 to_a 629
 values 629
 values_at 629
 struct sockaddr 764
Struct::Tms class 630
Stub
 RubyGems 222
 WIN32OLE 272
sub method
 class String 74, 620
 module Kernel 530
sub! method
 class String 620
 module Kernel 530
Subclass 27
 Subnet, testing address in 688
Subprocess see Process
Subroutine see Method

Substitution see Regular expression
succ method
 class Integer 502
 class String 620, 689
 for generating sequences 67
succ! method
 class String 621, 689
success? method
 class Exception 463
 class Process::Status 592
Suites, test 160
 Suketa, Masaki 269
sum method
 class String 621
super 29, 350, 575
Superclass 27, 379, 405
see also Module, mixin
superclass method
 class Class 405, 446
swapcase method
 class String 621
swapcase! method
 class String 621
SWIG 301
switch method
 module Process::GID 590
 module Process::UID 596
Symbol
 literal 323
Symbol class 31, 338, 374, 615, 631
 all_symbols 631
 id2name 631
 inspect 632
 to_i 632
 to_int 632
 to_s 632
 to_sym 632
symlink method
 class File 473
symlink? method
 class File::Stat 482
 class File 473
Sync class 738
Sync module 141
sync method
 class IO 514
sync= method
 class IO 514
SyncEnumerator class 683
Synchronization see Thread, synchronization
sysaccept method

class Socket 769
syscall.h 530
syscall method
 module Kernel 530
Syslog class 740
sysopen method
 class IO 507
sysread method
 class IO 514
sysseek method
 class IO 514
system method
 module Kernel 148, 530
syswrite method
 class IO 514

T

-T[level] (Ruby option) 179
Tab completion
 irb 187
Tag file 196
taint method
 class Object 575
Tainted objects 281, 399, 575
see also Safe level
tainted? method
 class Object 575
 Talbott, Nathaniel 151, 161
tan method
 module Math 542
tanh method
 module Math 542
Tcl/Tk see GUI programming
TCP see Network protocols
TCPServer class 773
 accept 773
 new 773
 open 773
TCPSocket class 735, 771
 gethostbyname 771
 new 771
 open 771
teardown method 158
Technical support 783
tell method
 class Dir 453
 class IO 515
Telnet see Network protocols, telnet
Tempfile class 741
Templates 239–244
 Amrita 241
 BlueCloth 218

eruby 242, 673	abort_on_exception 138, 633, 636	extensions to 745
RDoc 240	abort_on_exception= 633, 636	getgm 645
Temporary directory 748	alive? 636	getlocal 645
Temporary file 741	critical 633	getutc 645
Terminal	critical= 141, 633	gm 642
pseudo 720	current 634	gmt? 645
terminate method	exit 634, 636	gmt_offset 645
class Thread 639	fork 634	gmtime 645
termsig method	group 636	gmtoff 646
class Process::Status 593	join 137, 636	hour 646
Ternary operator 97, 343	key? 637	isdst 646
Test case 156	keys 637	local 643
Test suites 160	kill 634, 637	localtime 646
test method	list 634	mday 646
module Kernel 531	main 634	min 646
Test::Unit 152–161	new 136, 634	mktme 643
exceptions 155	pass 635	mon 646
assertions 152, 162f	priority 637	month 646
cases 156	priority= 638	new 643
naming conventions 161	Queue 743	now 643
setup 158	raise 638	sec 647
suites 160	run 638	strftime 647
teardown 158	safe_level 639	times 643
<i>see also</i> Testing	SizedQueue 743	to_a 647
Test::Unit class 742	start 635	to_f 647
Testing 151–161	status 639	to_i 647
ad hoc 151	stop 635	to_s 647
assertions 152	stop? 639	tv_sec 648
exceptions 155	terminate 639	tv_usec 648
gem 217	value 137, 639	usec 648
Roman numerals 151	wakeup 639	utc 643, 649
using StringIO 736	thread library 141, 145	utc? 649
structuring tests 156	ThreadGroup class 636, 640	utc_offset 649
what is a unit test? 152	add 640	wday 649
where to put files 159	enclose 640	yday 649
\$' variable 69, 334, 538	enclosed? 641	year 649
then 343	freeze 641	zone 649
Thread 135–147	list 641	time library 745
condition variable 145	new 640	Timeout module 746
creating 135	ThreadsWait class 744	times method
exception 138	throw method	class Integer 102, 502
group 640	module Kernel 114, 362, 531	class Time 643
queue 743	Time class 465, 642, 745	module Process 587, 630
race condition 137	+ 644	:title: (RDoc) 207
scheduling 140	- 644	Tk <i>see</i> GUI programming
synchronization 141–147,	<=> 644	Tk class 747
695–697, 738, 743	asctime 644	tmpdir library 741, 748
variable 137	at 642	tmpdir method
variable scope 136	ctime 644	class Dir 246, 748
waiting for multiple 744	day 644	to_a method
Thread class 633	dst? 644	class Array 439
[] 635		class Hash 499
[]= 636		class MatchData 539
		class NilClass 561

class Object 575
 class Struct 629
 class Time 647
 module Enumerable 459
 to_ary method 340, 373, 429,
 439
 class Array 439
 to_enum method 672
 to_f method
 class Bignum 443
 class Fixnum 486
 class Float 489
 class NilClass 561
 class String 621
 class Time 647
 to_hash method 373, 493
 class Hash 499
 to_i method
 class Float 489
 class Integer 502
 class IO 515
 class NilClass 561
 class Process::Status 593
 class String 622
 class Symbol 632
 class Time 647
 to_int method 372, 373
 class Float 489
 class Integer 502
 class Numeric 566
 class Process::Status 593
 class Symbol 632
 to_io method 373
 class IO 515
 to_proc method 373
 class Method 544
 class Proc 582
 to_s method 372
 class Array 440
 class Bignum 443
 class Exception 463
 class Fixnum 486
 class Float 490
 class Hash 499
 class MatchData 539
 class NilClass 561
 class Object 26, 575
 class Process::Status 593
 class Proc 582
 class Regexp 603
 class String 622
 class Symbol 632
 class Time 647
 and print 131, 526

to_str method 372, 374, 606
 class Exception 463
 class Object 280
 class String 622
 to_sym method 374
 class Fixnum 486
 class String 622
 class Symbol 632
 to_yaml_properties method
 416
 Top-level environment 393
 TOLEVEL_BINDING constant
 337
 Topological sort 750
 tr method
 class String 622, 689
 tr! method
 class String 623, 689
 tr_s method
 class String 623, 689
 tr_s! method
 class String 623, 689
 trace_var method
 module Kernel 532
 tracer library 749
 Tracing 412, *see* Logger
 Transactions 53
 Transcendental functions 540
 Transparent language 53, 59
 transpose method
 class Array 440
 trap method
 module Kernel 150, 534
 module Signal 605
 Trigonometric functions 540
 Troubleshooting 167
 TRUE constant 337
 true constant 93, 337, 341
 TrueClass class 650
 & 650
 ^ 650
 | 650
 truncate method
 class File 473, 476
 class Float 490
 class Integer 502
 class Numeric 566
 TSort module 750
 tsort_each_child method
 750
 tsort_each_node method 750
 tty? method
 class IO 515
 Tuning *see* Performance

Tuplespace *see* Distributed Ruby, Rinda
 tv_sec method
 class Time 648
 tv_usec method
 class Time 648
 type method
 class Object 370, 575
 Types *see* Duck typing
 Typographic conventions xxix

U

UDP *see* Network protocols
 UDPSocket class 735, 774
 bind 774
 connect 774
 new 774
 open 774
 recvfrom 775
 send 775
 uid method
 class File::Stat 482
 module Process 587
 uid= method
 module Process 587
 umask method
 class File 474
 un library 751
 Unary minus, unary plus 562
 unbind method
 class Method 544
 UnboundMethod class 408, 543,
 544, 549, 651
 arity 652
 bind 652
 undef_method method
 class Module 559
 undefine_finalizer method
 module ObjectSpace 579
 ungetc method
 class IO 515
 Unicode 317
Uniform Access Principle 32
 uniq method
 class Array 440
 uniq! method
 class Array 440
 Unit test *see* Testing
 UNIXServer class 777
 accept 777
 new 777
 open 777
 UNIXSocket class 735, 776

addr 776	-v, --verbose (Ruby option) 179, 336	\$; 178, 334
new 776	\$< 335	\$< 335
open 776	\$= 324, 334	\$= 324, 334
path 776	\$> 335	\$> 335
peeraddr 776	\$? 89, 148, 150, 335, 338, 516, 531, 587, 591	\$? 89, 148, 150, 335, 338, 516, 531, 587, 591
recvfrom 776	\$@ 334	\$@ 334
unless <i>see if expression</i>	\$DEBUG 178, 336, 633, 637	\$DEBUG 178, 336, 633, 637
unlink method	\$F 178, 336	\$F 178, 336
class Dir 452	\$FILENAME 336	\$FILENAME 336
class File 474	\$KCODE 324, 659, 689	\$KCODE 324, 659, 689
unpack method	\$LOAD_PATH 160, 178, 186, 221–223, 225, 228, 336	\$LOAD_PATH 160, 178, 186, 221–223, 225, 228, 336
class String 623	\$SAFE 179, 313, 336, 398, 575, 673	\$SAFE 179, 313, 336, 398, 575, 673
unpack_sockaddr_in method	\$VERBOSE 178, 179, 336, 534	\$VERBOSE 178, 179, 336, 534
class Socket 769	\$\ 179, 334, 511, 526	\$\ 179, 334, 511, 526
unpack_sockaddr_un method	\$_ 24, 95, 101, 178, 335, 342, 510, 523	\$_ 24, 95, 101, 178, 335, 342, 510, 523
class Socket 769	\$` 69, 334, 538	\$` 69, 334, 538
unshift method	\$configure_args 779	\$configure_args 779
class Array 440	\$deferr 335	\$deferr 335
untaint method	\$defout 335	\$defout 335
class Object 575	\$expect_verbose 676	\$expect_verbose 676
until <i>see while loop</i>	\$stderr 335	\$stderr 335
untrace_var method	\$stdin 335	\$stdin 335
module Kernel 534	\$stdout 335	\$stdout 335
upcase method	\$' 69, 334, 538	\$' 69, 334, 538
class String 623	\$~ 69, 77, 334, 537, 600–602	\$~ 69, 77, 334, 537, 600–602
upcase! method	@fileutils_output 678	@fileutils_output 678
class String 625	__FILE__ 413	__FILE__ 413
update	ARGF 24, 336	ARGF 24, 336
Observable callback 706	ARGV 178–180, 336, 523, 684, 711	ARGV 178–180, 336, 523, 684, 711
update method	ENV 181, 336	ENV 181, 336
class Hash 500	environment <i>see</i> Environment variables	environment <i>see</i> Environment variables
upto method	__FILE__ 336	__FILE__ 336
class Integer 102, 502	__LINE__ 336	__LINE__ 336
class String 625	predefined 333	predefined 333
URI class 752	English names 333, 671	English names 333, 671
URI, opening as file 707	RUBY_PLATFORM 228	RUBY_PLATFORM 228
Usage, message 213	self 81, 122, 337, 349, 379	self 81, 122, 337, 349, 379
usec method	Vector class 694	Vector class 694
class Time 648	\$VERBOSE variable 178, 179, 336, 534	\$VERBOSE variable 178, 179, 336, 534
Usenet 784	--version (Ruby option) 179	--version (Ruby option) 179
UTC 642	Versions of Ruby xxv	Versions of Ruby xxv
utc method	vi and vim 165	vi and vim 165
class Time 643, 649	tag file 196	tag file 196
utc? method	vi key binding 723	vi key binding 723
class Time 649	Virtual	Virtual
utc_offset method		
class Time 649		
UTF 317		
UTF8 324, 689		
utime method		
class File 474		
V		
-v (Ruby option) 535		

class 381
Virtual attribute 32

W

-w (Ruby option) 179, 336, 535
-W level (Ruby option) 179, 534
\$-w variable 336
wait method
 class IO 687
 module Process 149, 587
wait2 method
 module Process 588
waitall method
 module Process 587
waitpid method
 module Process 588
waitpid2 method
 module Process 588
wakeup method
 class Thread 639
Walk directory tree 679
warn method
 module Kernel 179, 336, 534
Warnings 179
 ARGV[0] is not \$0 180
 be careful with tainted data 397
 C functions must return VALUE 277
 strings aren't numbers 60, 169
wday method
 class Time 649
Weak reference 753
WeakRef class 753
 weakref_alive? 753
weakref_alive? method
 class WeakRef 753
Web *see* CGI programming
Web framework
 CGIKit 253
 Iowa 239
 Rails 253

Web server
 trivial 773
WEBrick 247, 754
 see also Apache
Web services 249
 description language 252
 Google 251
Web sites for Ruby xxvii, 783
Webcoder, Walter 397
WEBrick 247
WEBrick module 754
Weirich, Jim 216
when (in case) 343
while loop 100, 344
 as modifier 100, 345
why the lucky stiff 654
Widget *see* GUI programming
Wildcard *see* fnmatch and glob
Win32API class 669, 755
Win32API library 268
WIN32OLE class 756
WIN32OLE library 269
Windows *see* Microsoft Windows, GUI programming
with-cflags (mkmf) 780
with-cppflags (mkmf) 780
with-ldflags (mkmf) 780
with-make-prog (mkmf) 780
WNOHANG (Process constant) 583
Words
 array of 16, 322
Working directory 178, 450
Wrap *see* Module, wrap
writable? method
 class File::Stat 482
 class File 474
writable_real? method
 class File::Stat 482
 class File 474
write method
 class IO 515
WSDL 252, 734

Google interface 252
WUNTRACED (Process constant) 583
Wyss, Clemens 398

X

/x regexp option 324
-x [directory] (Ruby option) 180
-X directory (Ruby option) 180
XML 725, 757
XMLRPC module 757
xmp 197

Y

-y, --yydebug (Ruby option) 180
YAML library 416, 535, 654, 758
yday method
 class Time 649
year method
 class Time 649
yield 50, 357
 arguments 21, 51
 and RDoc 205
:yields: (RDoc) 206
Yukihiro, Matsumoto 382

Z

zero? method
 class File::Stat 482
 class File 474
 class Fixnum 486
 class Float 490
 class Numeric 566
Zip compression 759
zip method
 module Enumerable 459
Zlib module 759
zone method
 class Time 649

Array#pack 的模板字符

指 令 含 义	
@	移动到绝对位置
A	ASCII 字符串（用空格补齐，计数符表示宽度）
a	ASCII 字符串（用 null 补齐，计数符表示宽度）
B	比特串（低位序）
b	比特串（高位序）
C	无符号字符
c	有符号字符
D, d	双精度浮点数，本机格式
E	双精度浮点数，小端（little-endian）字节序
e	双精度浮点数，小端字节序
F, f	单精度浮点数，本机格式
G	双精度浮点数，网络（大端，big-endian）字节序
g	单精度浮点数，网络（大端）字节序
H	十六进制字符串（高四位在前）
h	十六进制字符串（低四位在前）
I	无符号整型
i	有符号整型
L	无符号长整型
l	有符号长整型
M	Quoted-printable、MIME 编码（参见 RFC2045）
m	Base64 编码的字符串
N	长整型，网络（大端）字节序
n	短整型（short），网络（大端）字节序
P	指向结构体（定长字符串）的指针
p	以 null 结尾字符串的指针
1.8 Q, q	64 位数字
S	无符号短整型
s	短整型
U	UTF-8
u	UU 编码的字符串
V	长整型，小端字节序
v	短整型，小端字节序
1.8 w	BER 压缩的整型 ¹
X	后退一个字节
x	Null 字节
Z	与 A 相同

¹ BER 压缩整型包括多个 8 位字节，表示一个以 128 为基数的无符号整型，开头是最高有效数位（most significant digit），其后的数字位尽可能少。除了最后一个字节，每个字节的第八位（高位）被设置为 1（自描述二进制数据表示，MacLeod）。

String#unpack 的模板字符

格式功能	返回值
A 一个以 NULL 结尾并且删除了空格的字符串	String
a 字符串	String
B 提取每个字符的位 (MSB 在前)	String
b 提取每个字符的位 (LSB 在前)	String
C 将一个字符作为无符号整型提取出来	Fixnum
c 将一个字符作为整型提取出来	Fixnum
d, D 将 <code>sizeof(double)</code> 个字符视为本机的双精度浮点数	Float
E 将 <code>sizeof(double)</code> 个字符视为小端字节序的双精度浮点数	Float
e 将 <code>sizeof(float)</code> 个字符视为小端字节序的双精度浮点数	Float
f, F 将 <code>sizeof(float)</code> 个字符视为本机的浮点数	Float
G 将 <code>sizeof(double)</code> 个字符视为网络字节序的双精度浮点数	Float
g 将 <code>sizeof(float)</code> 个字符视为网络字节序的浮点数	Float
H 提取每个字符的 4 位十六进制值 (最高有效位在前)	String
h 提取每个字符的 4 位十六进制值 (最低有效位在前)	String
I 将 <code>sizeof(int)</code> ² 的连续字符视为一个无符号的本机整型	Integer
i 将 <code>sizeof(int)</code> ² 的连续字符视为一个有符号的本机整型	Integer
L 将四个连续字符视为一个无符号的本机整型	Integer
l 将四个连续字符视为一个有符号的本机整型	Integer
M 提取一个引用可打印 (quoted-printable) 的字符串	String
m 提取一个 Base64 编码的字符串	String
N 将 4 个字符视为一个网络字节序的无符号长整型	Fixnum
n 将 2 个字符视为一个网络字节序的无符号短整型	Fixnum
P 将 <code>sizeof(char *)</code> 个字符视为一个指针，并从指向的位置返回 <code>len</code> 个字符	String
p 将 <code>sizeof(char *)</code> 个字符视为一个指针，指向一个 null 结尾的字符串	String
Q 将 8 个字符视为一个无符号的 4 个字 (quad word, 64 位)	Integer
q 将 8 个字符视为一个有符号的 4 个字 (quad word, 64 位)	Integer
S 将 2 个 ¹ 连续的字符视为一个本机字节序的无符号短整型	Fixnum
s 将 2 个 ¹ 连续的字符视为一个本机字节序的有符号短整型	Fixnum
U 将一个 UTF-8 字符提取为无符号整型	Integer
u 提取一个 uu 编码的字符串	String
V 将 4 个字符视为一个小端字节序的无符号长整型	Fixnum
v 将 2 个字符视为一个小端字节序的无符号短整型	Fixnum
w BER 压缩的整型 (更多信息请参见 <code>Array#pack</code>)	Integer
X 向后跳过一个字符	—
x 向前跳过一个字符	—
Z 删除了 NULL 结尾的字符串	String
@ 跳过由长度参数指定的偏移	—

² 可能因为向指令附加 “_” 而改变。

字符类别的简写

序 列	代 表 [...]	含 义
\d	[0-9]	数字字符
\D	[^0-9]	除数字字符之外的任意字符
\s	[s\t\r\n\f]	空白字符
\S	[^s\t\r\n\f]	除空白字符之外的任意字符
\w	[A-Za-z0-9_]	构成单词的字符
\W	[^A-Za-z0-9_]	除单词字符之外的任意字符
POSIX 字符类		
[:alnum:]		字母和数字
[:alpha:]		大写或小写字母
[:blank:]		空白和 tab
[:cntrl:]		控制字符 (至少包括 0x00-0x1f, 0x7f)
[:digit:]		数字
[:graph:]		除空格之外的可打印字符
[:lower:]		小写字符
[:print:]		任意可打印字符 (包括空格)
[:punct:]		除空格和字母数字之外的可打印字符
[:space:]		空白字符 (与 \s 相同)
[:upper:]		大写字符
[:xdigit:]		十六进制数字 (0-9, a-f, A-F)

sprintf 的标志字符

标 志	作用于	含 义
<code> </code> (<i>space</i>)	<code>bdeEfGgiouXx</code>	在正数的前面留下一个空格
<code>digit\$</code>	<code>all</code>	为这个字段指定绝对的参数个数。在 <code>sprintf</code> 字符串中不能同时使用绝对和相对参数个数
<code>#</code>	<code>beFfgGoxX</code>	使用备用格式。对转换 <code>b</code> 、 <code>o</code> 、 <code>X</code> 和 <code>x</code> 来说，分别在结果上加上前缀 <code>b</code> 、 <code>0</code> 、 <code>0X</code> 、 <code>0x</code> 。对 <code>E</code> 、 <code>e</code> 、 <code>f</code> 、 <code>G</code> 和 <code>g</code> 而言，即使没有小数位也输出小数点。对 <code>G</code> 和 <code>g</code> 来说，不会删除结尾的 <code>0</code>
<code>+</code>	<code>bdeEfGgiouXx</code>	在正数的前面加上一个加号
<code>-</code>	<code>all</code>	将转换的结果左对齐
<code>0 (zero)</code>	<code>bdeEfGgiouXx</code>	使用 <code>0</code> 而不是空格进行补齐
<code>*</code>	<code>all</code>	使用下一个参数作为字段的宽度。如果为负数，将结果左对齐。如果星号后面跟着一个数字和一个美元符，则使用指定的参数作为宽度

sprintf 的字段类型**字 段 转 换**

b	将参数转换为一个二进制数字
c	参数是一个单字符的数值码
d	将参数转换为一个十进制数字
E	等价于 e, 但是使用大写的 E 来表示指数
e	将浮点参数转换为指数形式, 并且在小数点前只有一个数字。精度决定小数的位数 (默认为 6)
f	将浮点参数转换为 [-]ddd.ddd, 其中精度决定小数点之后的数字位数
G	等价于 g, 但是使用大写 E 的指数形式
g	转换一个浮点参数, 如果指数小于 -4 或者大于等于精度, 则使用指数形式, 否则使用 d.dddd 形式
i	等同于 d
o	将参数转换为一个八进制数
1.8. p	<i>argument.inspect</i> 的值
s	参数是要被替换的字符串。如果格式化序列包括一个精度值, 最多会有精度值这么多的字符会被拷贝
u	将参数视为一个无符号的十进制数
X	将参数转换为大写字母的十六进制数。当显示负数时会在其前面加两个点 (表示前面有无限个 FF)
x	将参数转换为一个十六进制的数。在显示负数时会在其前面加两个点 (表示前面有无限个 FF)

Time#strftime 指令

格式	含义
%a	星期名称的缩写 ("Sun")
%A	完整的星期名称 ("Sunday")
%b	月份名称的缩写 ("Jan")
%B	完整的月份名称 ("January")
%c	首选的本地日期和时间表示
%d	一个月中的第几天 (01..31)
%H	一天中的第几个小时, 24 小时时钟 (00..23)
%I	一天中的第几个小时, 12 小时时钟 (01..12)
%j	一年中的第几天 (001..366)
%m	一年中的第几个月 (00..12)
%M	一小时中的第几分钟 (00..59)
%p	上下午的指示 ("AM"或"PM")
%S	一分钟的第几秒 (00..60)
%U	本年的第几个星期, 以第一个星期天作为首个星期的首天 (00..53)
%W	本年的第几个星期, 以第一个星期一作为首个星期的首天 (00..53)
%w	一个星期中的第几天 (星期天为 0, 0..6)
%x	首选的日期表示, 不包括时间
%X	首选的时间表示, 不包括日期
%y	不带有世纪的年份 (00..99)
%Y	带有世纪的年份
%Z	时区的名称
%%	%字符的字面值

单个参数的文件检测

标志	描述	返回值
?A	<i>file1</i> 的最近访问时间	Time
?b	如果 <i>file1</i> 是一个块设备，则返回真	true 或 false
?c	如果 <i>file1</i> 是一个字符设备，则返回真	true 或 false
?C	<i>file1</i> 的最后更改时间	Time
?d	如果 <i>file1</i> 存在并且是一个目录，则返回真	true 或 false
?e	如果 <i>file1</i> 存在，则返回真	true 或 false
?f	如果 <i>file1</i> 存在并是一个一般文件，则返回真	true 或 false
?g	如果 <i>file1</i> 设置了 setgid 位，则返回真（在 NT 下返回 false）	true 或 false
?G	如果 <i>file1</i> 存在，并且组所有权和调用者的组相同，则返回真	true 或 false
?k	如果 <i>file1</i> 存在，并设置了 sticky（粘滞）位，则返回真	true 或 false
?l	如果 <i>file1</i> 存在，并且是一个符号链接，则返回真	true 或 false
?M	<i>file1</i> 的最后修改时间	Time
?o	如果 <i>file1</i> 存在，并且所有权属于调用者的有效 UID，则返回真	true 或 false
?O	如果 <i>file1</i> 存在，并且所有权属于调用者的真实 UID，则返回真	true 或 false
?p	如果 <i>file1</i> 存在，并且是一个 fifo（译注，亦称为“具名管道”）	true 或 false
?r	如果 <i>file1</i> 对调用者的有效 UID/GID 来说可读，则返回真	true 或 false
?R	如果 <i>file1</i> 对调用者的真实 UID/GID 来说可读，则返回真	true 或 false
?s	如果 <i>file1</i> 的长度不为 0，则返回其大小；否则返回 nil	Integer 或 nil
?S	如果 <i>file1</i> 存在并且是一个套接字（socket），则返回真	true 或 false
?u	如果 <i>file1</i> 存在，并且设置了 setuid 位，则返回真	true 或 false
?w	如果 <i>file1</i> 存在，并且对有效的 UID/GID 来说可写，则返回真	true 或 false
?W	如果 <i>file1</i> 存在，并且对真实的 UID/GID 来说可写，则返回真	true 或 false
?x	如果 <i>file1</i> 存在，并且对有效的 UID/GID 来说可执行，则返回真	true 或 false
?X	如果 <i>file1</i> 存在，并且对真实的 UID/GID 来说可执行，则返回真	true 或 false
?z	如果 <i>file1</i> 存在，并且长度为 0，则返回真	true 或 false

两个参数的文件检测

格式	描述
?-	如果 <i>file1</i> 是 <i>file2</i> 的硬链接，则返回真
?=	如果 <i>file1</i> 和 <i>file2</i> 的修改时间相同，则返回真
?<	如果 <i>file1</i> 的修改时间在 <i>file2</i> 之前，则返回真
?>	如果 <i>file1</i> 的修改时间在 <i>file2</i> 之后，则返回真

作 者： [美] DAVE THOMAS CHAD FOWLER ANDY HUNT

著 孙勇 姚延栋 张海峰译

形态项： 830

出版项： 电子工业出版社， 2007

I S B N号： 7 - 121 - 03815 - 3 / TP312

原书定价： 99.00

主题词： 计算机网络 程序设计

参考文献格式： [美] DAVE THOMAS CHAD FOWLER ANDY HUNT著 孙勇 姚延栋 张海峰译. PROGRAMMING RUBY中文版（第二版）. 电子工业出版社， 2007 .

读秀号： 000004890574

读秀d值： 6595424F8EFC139F5316B455BD9E76C0

S S号： 11824644

封面

书名

版权

第1版序

第2版序

前言

路线图

目录

第1部分 Ruby面面观

第1章 入门

- 1.1 安装Ruby
- 1.2 运行Ruby
- 1.3 Ruby文档：RDoc和ri

第2章 Ruby.new

- 2.1 Ruby是一门面向对象语言
- 2.2 Ruby的一些基本知识
- 2.3 数组和散列表
- 2.4 控制结构
- 2.5 正则表达式
- 2.6 Block和迭代器
- 2.7 读/写文件
- 2.8 更高更远

第3章 类、对象和变量

- 3.1 继承和消息
- 3.2 对象和属性
- 3.3 类变量和类方法
- 3.4 访问控制
- 3.5 变量

第4章 容器、Blocks和迭代器

- 4.1 容器
- 4.2 Blocks和迭代器
- 4.3 处处皆是容器

第5章 标准类型

- 5.1 数字
- 5.2 字符串
- 5.3 区间
- 5.4 正则表达式

第6章 关于方法的更多细节

- 6.1 定义一个方法
- 6.2 调用方法

第7章 表达式

- 7.1 运算符表达式
- 7.2 表达式之杂项
- 7.3 赋值
- 7.4 条件执行
- 7.5 Case表达式
- 7.6 循环
- 7.7 变量作用域、循环和Blocks

第8章 异常，捕获和抛出

- 8 . 1 异常类
- 8 . 2 处理异常
- 8 . 3 引发异常
- 8 . 4 捕获和抛出

第9章 模块

- 9 . 1 命名空间
- 9 . 2 M i x i n
- 9 . 3 迭代器与可枚举模块
- 9 . 4 组合模块
- 9 . 5 包含其他文件

第10章 基本输入和输出

- 1 0 . 1 什么是 I O 对象
- 1 0 . 2 文件打开和关闭
- 1 0 . 3 文件读写
- 1 0 . 4 谈谈网络

第11章 线程和进程

- 1 1 . 1 多线程
- 1 1 . 2 控制线程调度器
- 1 1 . 3 互斥
- 1 1 . 4 运行多个进程

第12章 单元测试

- 1 2 . 1 T e s t : : U n i t 框架
- 1 2 . 2 组织测试
- 1 2 . 3 组织和运行测试

第13章 当遇到麻烦时

- 1 3 . 1 R u b y 调试器
- 1 3 . 2 交互式 R u b y
- 1 3 . 3 编辑器支持
- 1 3 . 4 但是它不运作
- 1 3 . 5 然而它太慢了

第2部分 R u b y 与其环境

第14章 R u b y 和 R u b y 世界

- 1 4 . 1 命令行参数
- 1 4 . 2 程序终止
- 1 4 . 3 环境变量
- 1 4 . 4 从何处查找它的模块
- 1 4 . 5 编译环境
- 1 5 . 1 命令行
- 1 5 . 2 配置
- 1 5 . 3 命令
- 1 5 . 4 限制
- 1 5 . 5 r t a g s 与 x m p

第16章 文档化 R u b y

- 1 6 . 1 向 R u b y 代码中添加 R D o c
- 1 6 . 2 向 C 扩展中添加 R D o c
- 1 6 . 3 运行 R D o c
- 1 6 . 4 显示程序用法信息

第17章 用Ruby Gems进行包的管理

- 17.1 安装Ruby Gems
- 17.2 安装程序Gems
- 17.3 安装和使用Gem库
- 17.4 创建自己的Gems
- 18.1 编写CGI脚本
- 18.2 Cookies
- 18.3 提升性能
- 18.4 Web服务器的选择
- 18.5 SOAP及Web Services
- 18.6 更多信息

第19章 Ruby Tk

- 19.1 简单的Tk应用程序
- 19.2 部件
- 19.3 绑定事件
- 19.4 画布
- 19.5 滚动
- 19.6 从Perl/Tk文档转译

第20章 Ruby和微软Windows系统

- 20.1 得到Ruby for Windows
- 20.2 在Windows下运行Ruby
- 20.3 Win32 API
- 20.4 Windows自动化

第21章 扩展Ruby

- 21.1 你的第一个扩展
- 21.2 C中的Ruby对象
- 21.3 Jukebox扩展
- 21.4 内存分配
- 21.5 Ruby的类型系统
- 21.6 创建一个扩展
- 21.7 内嵌Ruby解释器
- 21.8 将Ruby连接到其他语言
- 21.9 Ruby C语言API

第3部分 Ruby的核心

第22章 Ruby语言

- 22.1 源代码编排
- 22.2 基本类型
 - 整数和浮点数
 - 字符串
 - 区间
 - 数组
 - 散列表
 - 符号
 - 正则表达式
 - 正则表达式扩展
- 22.3 名字
- 22.4 变量和常量
- 22.5 表达式

22.6	方法定义
22.7	调用方法
22.8	别名
22.9	类定义
22.10	模块定义
22.11	访问控制
22.12	Blocks, Closures和Proc对象
22.13	异常
22.14	Catch和Throw
第23章	Duck Typing
23.1	类不是类型
23.2	像鸭子那样编码
23.3	标准协议和强制转换
23.4	该做的做，该说的说
第24章	类与对象
24.1	类和对象是如何交互的
24.2	类和模块的定义
24.3	顶层的执行环境
24.4	继承与可见性
24.5	冻结对象
第25章	Ruby 安全
25.1	安全级别
25.2	受污染的对象
第26章	反射，Object Space 和分布式Ruby
26.1	看看对象
26.2	考察类
26.3	动态地调用方法
26.4	系统钩子
26.5	跟踪程序的执行
26.6	列集和分布式Ruby
26.7	编译时？运行时？任何时候
第4部分	Ruby库的参考
第27章	内置的类和模块
27.1	字母顺序列表
	Array
	Bignum
	Binding
	Class
	Comparable
	Continuation
	Dir
	Enumerable
	Errno
	Exception
	FalseClass
	File
	File::Stat
	FileTest

```
Fixnum
Float
GC
Hash
Integer
IO
Kernel
Marshal
MatchData
Math
Method
Module
NilClass
Numeric
Object
ObjectSpace
Proc
Process
Process::GID
Process::Status
Process::Sys
Process::UID
Range
Regexp
Signal
String
Struct
Struct::TMS
Symbol
Thread
ThreadGroup
Time
TrueClass
UnboundMethod
```

第28章 标准库

```
Abbrev
Base64
Benchmark
BigDecimal
CGI
CGI::Session
Complex
CSV
Curses
Date/DateTime
DBM
Delegator
Digest
```

D L
dRuby
English
Enumerator
erb
Etc
expect
Fcntl
FileUtils
Find
Forwardable
f tools
GDBM
Generator
GetoptLong
GServer
Iconv
IO/Wait
IPAddr
jcode
Logger
Mail
mathn
Matrix
Monitor
Mutex
Mutex_m
Net::FTP
Net::HTTP
Net::IMAP
Net::POP
Net::SMTP
Net::Telnet
NKF
Observable
open-uri
Open3
OpenSSL
OpenStruct
OptionParser
ParseDate
Pathname
PP
PrettyPrint
Profile
Profiler_—
PStore
PTY

Rational
readbytes
Readline
Resolve
REXML
Rinda
RSS
Scanf
SDBM
Set
Shellwords
Singleton
SOAP
Socket
StringIO
StringScanner
Sync
Syslog
Tempfile
Test::Unit
thread
ThreadsWait
Time
Timeout
Tk
tmpdir
Tracer
TSort
un
URI
WeakRef
WEBbrick
Win32 API
WIN32 OLE
XMLRPC
YAML
Zlib

第5部分 附录

- 附录A Socket库
- 附录B MKMF参考
- 附录C 支持
- 附录D 书目
- 索引 (Index)

表目录

- 表2.1 变量和类名称样例
- 表5.1 字符类缩写
- 表7.1 常用的比较操作符
- 表11.1 在一个竞态条件中的两个线程

- 表13.1 调试器命令
 表14.1 Ruby使用的环境变量
 表15.1 irb命令行选项
 表17.1 版本操作符
 表18.1 erb的命令行选项
 表21.1 C/Ruby数据类型转换函数和宏
 表22.1 常规分隔输入
 表22.2 双引号字符串中允许的替换
 表22.3 保留字
 表22.4 Ruby操作符(优先级从高到低)
 表25.1 安全级别的定义
 表27.1 Array#pack的模板字符
 表27.2 匹配模式常量
 表27.3 路径分隔符常量(和平台相关)
 表27.4 open模式常量
 表27.5 锁模式常量
 表27.6 模式字符串
 表27.7 sprintf标志字符
 表27.8 sprintf域类型
 表27.9 参数只有一个的文件测试
 表27.10 参数有两个的文件测试
 表27.11 定义在Numeric类以及其子类中的方法。打勾意味着这个方法定义在相应的类中

表27.12 模(modulo)和余数(remainder)之间的区别。模运算符(“%”)总是有除数(divisor)的符号,然而remainder有被除数(dividend)的符号

- 表27.13 替换字符串中的反斜线序列
 表27.14 String#unpack指令
 表27.15 Time#strftime指令
 表28.1 ERB的指令
 表28.2 选项定义参数

图目录

- 图3.1 变量保存对象引用
 图4.1 如何对数组进行索引操作
 图8.1 Ruby异常层次结构
 图12.1 产生罗马数字(有bug)
 图13.1 irb会话的示范
 图13.2 使用benchmark比较变量访问代价
 图16.1 浏览类Counter的RDoc输出
 图16.2 浏览带注释的源代码的RDoc输出
 图16.3 使用ri来读取文档
 图16.4 由Rdoc/ri生成的Proc类的文档
 图16.5 用RDoc文档化的Ruby源文件
 图16.6 用RDoc文档化的C源文件
 图16.7 使用RDoc::usage的范例程序
 图16.8 由范例程序产生的帮助信息
 图17.1 MomLog包结构
 图18.1 简单的CGI表单

- 图18.2 使用e r b处理带有循环的文件
- 图19.1 在Tk的Canvas(画布)上绘制
- 图21.1 将C数据类型包装为对象
- 图21.2 构建扩展的过程
- 图22.1 boolean区间(range)的状态变迁
- 图24.1 一个基本对象，以及它的类和超类
- 图24.2 添加Guitar的metaclass
- 图24.3 向对象中添加一个虚拟类
- 图24.4 被包含的模块以及它的代理类
- 图27.1 标准异常等级结构
- 图27.2 实际运行Method#arity