The ruby way

前言

这是一本关于秘诀的书籍，其中包括：对一般问题的解决方案、复制和粘贴代码段、解释、示例和简短指南。

本书意欲为读者节省时间。人们总是说，时间就是金钱，但时间也是个人生命的构成。我们的生命应当花在创造新事物上，而不是用于抗击我们自己的错误或者解决那些已经解决过的问题。我们提供此书的愿望是：它的所有读者所节省的时间远远超过我们编写它所花掉的时间。

Ruby 编程语言本身是一种非常节省时间的工具，与其他编程语言相比，它能产生更高的生产率，因为用户会花费更多的时间让计算机做自己想做的事情，而思考语言本身的时间则较少。但是，对于一名 Ruby 程序员，可能有很多情况即使没做任何事情也花费了很多时间，下面是我们遇到过的各个方面：

- 将时间花在编写通用算法的 Ruby 实现上。
- 将时间花在调试通用算法的 Ruby 实现上。
- 将时间花在发现和修正 Ruby 特有的缺陷上。
- 将时间花在应当自动执行的重复性任务上（包括重复性编程任务！）。
- 将时间花在重复其他人已经公开实现过的工作上。
- 将时间花在搜索运行 X 的库上。
- 将时间花在对运行 X 的多个库进行评估和确定上。
- 将时间花在由于文档的匮乏或过时而学习如何使用库上。
- 将时间浪费在由于惧怕而不敢接触有用的技术上。

我们以及本书的许多投稿人还清楚地记得我们自己浪费的那些时日。我们将自己的经历提炼进本书中，从而让读者不再浪费自己的时间——或者至少让读者愉快地将时间花在其他更有趣的问题上。

我们的另一个目标是扩大读者的兴趣。如果读者阅读本书后希望能够使用 Ruby 生成算法音乐，没问题，12.14 节将会节省读者的时间。迄今为止，读者更可以不必考虑可能性问题。本书中每个秘诀的形成和编写都在理念上带有如下这样两个目标：节省读者时间以及让读者的头脑对新观点保持活跃。

本书读者对象

本书的目标读者是那些至少懂一点 Ruby 或者对一般性编程相当了解的人。这不是一本 Ruby 指南（要查看某些真正的指南，请参见下面的“其他资源”一节），但是，如果读者已经熟悉一些其他的编程语言，那么应当能够通过阅读本书的前 10 章并在阅读过程中实践列出的代码，从而能够做到对 Ruby 无师自通。

我们已经包含的秘诀适用于各级读者，从那些刚刚开始使用 Ruby 的用户到那些需要偶尔进行参考的专家。我们主要致力于一般的编程技术，但也涵盖了特定的应用程序框架（类似于 Rails 和 GUI 库上的 Ruby）和最佳实践（类似于单元测试）。

即使读者只准备将本书作为参考用书，我们也建议读者通读一遍全书，以便了解我们所解决的问题。这本书很厚，但并没有解决所有问题。如果读者在阅读后发现找不到自己问题的解决方案，或者对自己问题可能有所帮助的信息，那么就是在浪费时间了。

如果读者事先通读本书，那么会对本书中我们所涉及的问题有一个清晰的概念，从而能获得更好的命中率。读者可以知道何时本书能够帮助自己，以及何时应当参考其他书籍、进行 Web 搜索、询问朋友或者从其他途径获得帮助。

本书组织结构

本书共有 23 章，每一章集中在一类编程或一种特殊数据类型。目前对章节的概述应当能够令读者了解我们如何划分各类秘诀。每一章还包括对章节本身有些冗长的介绍，给出章中各节更详细的描述。我们强烈建议读者浏览章节介绍和内容列表。

我们用前 6 章介绍 Ruby 的内置数据结构。

•第 1 章字符串，包括构建、处理和操作文本字符串的秘诀。我们特别介绍了若干正则表达式的秘诀（1.17 节至 1.19 节），但是，我们重点关注的是特定的 Ruby 问题，而正则表达式则是一种十分常见的工具。如果读者尚未使用过它，或者对使用它感到恐惧，那么我们建议读者阅读一本在线指南或由 Jeffrey Friedl 撰写的 Mastering Regular Expressions 一书（O'Reilly 出版）。

•第 2 章数字，介绍了对不同类型的数字的表示，包括：实数、复数、任意精度的小数，等等。它还包括了一般的数学和统计学算法的 Ruby 实现，并解释了在创建自己的数字类型时会遇到的一些 Ruby 独有的特性（2.13 节和 2.14 节）。

•第 3 章日期与时间，涉及 Ruby 在时间处理上的两个接口：一个基于 C 的 time 库，这在其他编程语言中比较常见；另一个以纯 Ruby 实现，更合乎语言习惯。

•第 4 章数组，介绍 Ruby 最简单的复合数据类型：数组。许多数组的方法实际上是 Enumerable 混入的方法，这意味着用户可以在散列或其他数据类型上应用这些秘诀。Enumerable 的某些特性在本章中有所涉猎（4.4 节和 4.6 节），还有一些特性在第 7 章中介绍。

•第 5 章散列，介绍了 Ruby 的另一种基本复合数据类型：散列。散列令对象与其名字相关联从而便于以后的查找（散列有时被称为“查找表”或“字典”，两个都是有效的名字）。结合使用散列与数组可以轻松地构建深奥、复杂的数据结构。

•第 6 章文件与目录，包括读、写和操作文件的技术。Ruby 的文件访问接口基于标准 C 的文件库，因此读者可能很熟悉。本章还涉及了 Ruby 用于搜索和操作文件系统的标准库，许多秘诀将在第 23 章中再次给出。

前 6 章处理特定的算法问题，接下来的 4 章会更加抽象：它们涉及到 Ruby 的方言和基本原理。如果用户无法令 Ruby 语言本身去做自己想做的事，或者在按 Ruby“应当”具有的外观来书写 Ruby 代码时遇到困难，那么下面这 4 章中的秘诀会有所帮助。

•第 7 章代码块与迭代，包括了研究 Ruby 代码块（也称为闭包）各种可能性的秘诀。

•第 8 章对象与类，介绍了 Ruby 承担的面对对象编程的任务。它包括了书写不同类型的类和方法的秘诀，以及少量以示例说明所有 Ruby 对象能力的秘诀（例如冻结和克隆）。

•第 9 章模块与命名空间，介绍了 Ruby 的模块。这些结构用于将新行为加入现有类中，并将功能分隔进不同的命名空间中。

•第 10 章反射与元编程，介绍了以编程手段研究和修改 Ruby 类定义的技术。

第 6 章介绍了基本的文件访问，但没有触及更多的特定文件格式。我们将用下面 3 章内容介绍流行的数据存储方法。

•第 11 章 XML 和 HTML，给出如何处理最流行的数据交换格式。本章主要涉及分析其他人的 XML 文档和 Web 网页（参见 11.9 节）。

•第 12 章图形与其他文件格式，介绍除 XML 和 HTML 之外的其他数据交换格式，特别关注了图形的生成与操作。

•第 13 章数据库和持久性，涉及 Ruby 对数据存储格式最好的接口，不管是串行地将 Ruby 对象存入磁盘，还是在数据库中存储结构化数据。从串行化数据和索引文本到 Ruby 针对流行 SQL 数据库的客户库，再到类似于 ActiveRecord、可完全避免书写 SQL 语句的完全成熟的抽象层，本章对其不同方法皆给出了相应的示例。

当前对 Ruby 最流行的使用是在网络应用程序中（大多数是通过 Ruby on Rails）。我们用下面 3 章的内容介绍不同类型的应用程序：

•第 14 章 Internet 服务，通过举例说明从 Ruby 库编写的大量客户端和服务端开始介绍我们的网络覆盖。

•第 15 章 Web 开发：Ruby on Rails，介绍这个正强有力地提升 Ruby 声望的 Web 应用程序框架。

•第 16 章 Web 服务及分布式编程，介绍 Ruby 编程中在计算机间共享信息的两种技术。为了使用 Web 服务，用户要向其他计算机上发出程序的 HTTP 请求，这通常是用户无法控制的。Ruby 的 DRb 库可使用户共享运行位于一组计算机上的程序之间的 Ruby 数据结构，一切皆可在控制中。

然后我们用下面 3 章的内容介绍出现在项目的主要编程工作周围的辅助性任务。

•第 17 章测试、调试、优化以及文档化，重点介绍处理异常条件以及为用户代码创建单元测试，还有若干关于调试和优化过程的秘诀。

•第 18 章打包和发布软件，主要处理 Ruby 的 Gem 打包系统和宿主许多 gem 文件的 RubyForge 服务器。其他章中的许多秘诀要求读者安装特殊的 gem，因此如果读者不熟悉 gem，那么我们建议读者特别阅读 18.2 节。本章还给出如何为自己的项目创建和发布 gem。

•第 19 章用 Rake 自动执行任务，介绍了最流行的 Ruby 构建工具。使用 Rake 可以像运行单元测试一样编写普通任务，或者像 gem 一样打包自己的代码。尽管它通常用在 Ruby 项目中，但它是一种通用的构建语言，用户可将其用在任何可以使用 Make 的地方。

我们用最后 4 章介绍其他主题。

•第 20 章多任务和多线程，给出如何使用线程同时做更多的事，以及如何用 Unix 子进程运行外部命令。

•第 21 章用户界面，介绍用户界面（除 Web 界面之外，其在第 15 章进行了介绍）。我们讨论了命令行界面、带 Curses 和 HighLine 的基于字符的 GUI、用于各种平台的 GUI 工具包，以及更多种无名的用户界面（21.11 节）。

•第 22 章用其他语言扩展 Ruby，主要介绍为了性能或获得对更多库的访问，将 Ruby 与其他语言相连接。本章大部分内容着力于获得对 C 库的访问，但有一个关于 JRuby 的秘诀，它是运行在 Java 虚拟机上的 Ruby 实现（22.5 节）。

•第 23 章系统管理，全面介绍完成管理任务的自包含程序，通常要使用其他章的技术。这些秘诀重点介绍了 Unix 管理，但也有一些针对 Windows 用户（23.2 节）和某些交叉平台脚本的资源。

代码清单的工作原理

学习本书意味着要执行秘诀。我们的秘诀中有些定义了大量的 Ruby 代码，读者只需简单地将代码放入自己的程序中并使用即可，而无需真正地理解它们（19.8 节就是一个很好的例子）。但是，大多数秘诀对技术进行了举例说明，而学习一项技术最好的途径就是实践它。

我们本着此种精神编写秘诀及其代码清单，大部分清单类似于对秘诀中描述内容进行单元测试：它们翻来覆去地测试对象并给出结果。

目前 Ruby 安装文件带有一个称为 irb 的交互式解释器。在一个 irb 会话内，用户可以键入 Ruby 代码行并立刻能够看到输出，而不必创建一个 Ruby 程序文件并通过解释器来运行它。

我们的秘诀中大部分以用户可以直接在 irb 会话中键入或进行复制/ 粘贴的方式存在。要想进一步研读秘诀，我们建议读者从 irb 会话开始并在阅读过程中运行代码清单，这样做可以使得对概念的理解能够比仅进行阅读要深刻一些。完成这一步后，读者可以进一步试验那些运行代码清单时自己定义的对象。

有时我们希望将读者的注意力引向 Ruby 表达式所希望的结果，因此，我们使用了 Ruby 注释，它包含指向表达式期望值的 ASCII 箭头。这与 irb 用于告知用户每个由用户键入的表达式的值所使用的箭头相同。

我们还使用文本注释来解释某些代码段。下面是某个秘诀中按我希望的格式进行注释的 Ruby 代码段：

```
1 + 2               # => 3

# On a long line, the expected value goes on a new line:
Math.sqrt(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10)
# => 7.41619848709566
```

为了显示 Ruby 表达式的期望输出，我们使用一个不含 ASCII 箭头的注释，并且它总会另起一行：

```
puts "This string is self-referential."
# This string is self-referential.
```

如果在 irb 中键入这两段代码而忽略注释，那么读者可以根据文本进行回查，以便验证是否得到了与我们相同的结果：

```
$ irbirb(main):001:0> 1 + 2=> 3irb(main):002:0> Math.sqrt(1 + 2 + 3 +
 4 + 5 + 6 + 7 + 8 + 9 + 10)=> 7.41619848709566irb(main):003:0> p
uts "This string is self-referential."This string is self-referential.=> nil
```

如果读者阅读的是本书的电子版，那么可以将代码段复制并粘贴至 irb 中。Ruby 解释器会忽略注释，但是，你可以用它们来确保自己的结果与我们的结果相匹配，而不必回头查看文本（但是，读者应当知道，如果自己键入代码，最起码在第一次键入代码时有利于自己更好地理解这些代码）。

```
$ irbirb(main):001:0> 1 + 2 # => 3=> 3irb(main):002:0>irb(main):003:0* # On
 a long line, the expected value goes on a new line:irb(main):004:0* Math.sqrt(1
 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10)=> 7.41619848709566irb(main):005:
0> # => 7.41619848709566irb(main):006:0*irb(main):007:0* puts "This string is
self-referential."This string is self-referential.=> nilirb(main):008:0> # This string i
s self-referential.
```

我们没有走捷径。我们秘诀中的大部分皆从头至尾完整地举例说明了 irb 会话，并且包括了我们试图举例说明的观点的所有重要性和初始必要性。如果读者正确地按秘诀运行代码，那么应当能够获得与我们相同的结果（注 1）。这适合于我们所谓的代码示例应当

注 1：当一个程序的行为依赖于当前时间、随机数生成器或磁盘上某些文件的存在时，可能无法得到与我们给出的相同结果，但应当是相似的。

是其下内容的单元测试的理念。事实上，我们像单元测试一样使用一种 Ruby 脚本分析秘诀文本并运行代码清单，对我们的代码示例进行了测试。

irb 会话技术并不总是有效的。Rails 秘诀必须运行在 Rails 中，Curses 秘诀取代屏幕，使用 irb 无法良好运行，为此，我们有时以下列格式给出独立的文件：

```
#!/usr/bin/ruby -w
# sample_ruby_file.rb: A sample file

1 + 2
Math.sqrt(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10)
puts "This string is self-referential."
```

只要有可能，我们还会给出运行这一程序将会得到的结果：可能是一个 GUI 程序的截屏图，或者是程序在 Unix 命令行下运行的输出记录：

```
$ ruby sample_ruby_file.rb
This string is self-referential.
```

要注意的是，sample_ruby_file.rb 的输出与键入 irb 相同代码的输出不同。在这里没有加法和平方根运算的痕迹，因为它们没有产生输出。

安装软件

Ruby 预安装在 Mac OS X 和大多数 Linux 安装程序中。Windows 没有自带 Ruby，但可以很容易地获得单击一次式安装程序（One-Click Installer）：参见 http://rubyforge.org/ projects/rubyinstaller/。

如果用户使用 Unix/Linux 系统而没有安装 Ruby（或者用户希望进行升级），那么产品的打包系统可以提供一个 Ruby 程序包。在 Debian GNU/Linux 上，它作为 ruby-[version]程序包提供：例如，ruby-1.8 或 ruby-1.9。Red Hat Linux 称其为 ruby，Mac OS X 上的 DarwinPorts 系统也是如此。

如果所有这些尝试都失败了，那么可以自己下载 Ruby 源代码并进行编译。用户可通过访问 http://www.ruby-lang.org/ 的 FTP 或 HTTP 获得 Ruby 源代码。

本书的许多秘诀都要求用户以 Ruby gem 形式安装第三方库。我们一般倾向于使用单独的解决方案（仅使用 Ruby 标准库）来解决 gem 的使用问题，使用基于 gem 的解决方案来解决要求使用其他类型的第三方软件的问题。

如果用户不熟悉 gem，那么可以根据自己的需要参考第 18 章。开始学习时，用户要做的只是懂得首先在 http://rubyforge.org/projects/rubygems/上下载 Rubygems 库而已（在网页中选择最新版本）。解压缩 tarball 或 ZIP 文件，变成 rubygems-[version]目录，并以超级用户权限运行下列命令：

```
$ ruby setup.rb
```

Rubygems 库包括在 Windows 的单击一次式安装程序中，因此用户在 Windows 上不必担心这一步。Rubygems 库安装完毕后，就可以很容易地安装其他的 Ruby 代码。如果某个秘诀中有类似于“Ruby on Rails 与 rails gem 一起提供”之类的话，那么用户可以在命令行中键入下列命令（再次强调，要作为超级用户）：

```
$ gem install rails --include-dependencies
```

RubyGems 库会下载并自动安装 rails gem（以及其他相互依赖的 gem）。然后，用户就能够完全如其所示地运行秘诀中的代码。

对于新安装的 Ruby 来说，三个最有用的 gem 是 rails（如果用户试图创建 Rails 应用程序的话）和 Ruby Facets 项目提供的两个 gem：facets_core 和 facets_more。Facets Core 库用通常情况下有用的方法扩展了 Ruby 标准库的类。Facets More 库添加了全新的类和模块。Ruby Fecets 主页（http://facets.rubyforge.org/）上有完整的索引。

有些 Ruby 库（特别是一些比较老的库）没有像 gem 一样打包封装。本书中提到的大多数非 gem 库在 Ruby 程序与库的目录 Ruby Application Archive（http://raa.ruby-lang.org/）中都有记录。

在多数情况下，用户可以在 RAA 上下载 tarball 或 ZIP 文件，并使用 18.8 节中描述的技术进行安装。

平台差异、版本差异与其他问题

除非有特别注明，各个秘诀描述的都是跨平台概念，代码本身应当以相同方式在 Windows、Linux 和 Mac OS X 上运行。大多数平台差异和特定于平台的秘诀在最后几章中给出，这几章包括：第 20 章、第 21 章和第 23 章（但需参见第 6 章中对 Windows 文件名的介绍）。

我们使用 Ruby 1.8.4 版和 Rails 1.1.2 版编写和测试秘诀，这是编写本书时最新的稳定版本。如果用户运行的是 Ruby 1.9（它是编写本书时不稳定的最新版本）或 2.0，那么在某些地方我们会提示用户应当对代码进行修改。

尽管我们尽了最大努力，但本书仍可能存在没有标记出来的特定于平台的代码、未提及的老 bug。我们为它们的出现而感到抱歉。如果读者对某个秘诀有疑问，可以查看本书的勘误表（参见下面的内容）。

在本书的若干秘诀中，我们为添加新方法而修改了类似 Array 的标准 Ruby 类（例如，参见 1.10 节，它定义了一种称为 String#capitalize_first_letter 的新方法）。之后，这些方法被用于用户程序的每个类实例。这在 Ruby 中是相当常用的技术：前面提到的 Rails 和 Facets Core 库都这样做了。尽管尚存在部分争论，并且可能会导致问题（参见 8.4 节的更深层讨论），但我们感觉应当在前言中提出这一点，尽管它对于那些新接触 Ruby 的人来讲过于技术化了。如果读者不愿意修改标准类，那么可以将我们给出的示例方法放入某个子类，或者在 Kernel 命名空间中定义它们：也就是说，定义 capitalize_first_letter_of_string 来代替重新打开 String 并在其中定义 capitalize_first_letter。

其他资源

如果读者需要学习 Ruby，那么标准参考用书是 Dave Thomas、Chad Fowler 和 Andy Hunt （务实的程序员）撰写的 Programming Ruby: The Pragmatic Programmer's Guide。该书第一版以 HTML 格式在线提供（http://www.rubycentral.com/book/），但已经过时。第二版更好一些，并以印刷书或 PDF（http://www.pragmaticprogrammer.com/titles/ruby/）形式提供。比试图阅读第一版更好的主意是购买第二版并使用第一版作为手头的参考书。

由“why the lucky stiff”提供的“Why's (Poignant) Guide to Ruby”用故事来讲授 Ruby，像一本英语初级读本一样，它是具有创造力的初学者的最佳用书（http:// poignantguide.net/ruby/）。

对于 Rails 而言，标准读物是 Dave Thomas、David Hansson、Leon Breedt 和 Mike Clark （务实的程序员）撰写的 Agile Web Development with Rails。与此类似的还有两本专门针对 Rails 的书：Rob Orsini 撰写的 Rails Cookbook（O'Reilly）和 Chad Fowler（务实的程序员）撰写的 Rails Recipes。

某些常见的 Ruby 缺陷在 Ruby FAQ（http://www.rubycentral.com/faq/，从第 4 部分开始）和“Things That Newcomers to Ruby Should Know”（http://www.glue.umd.edu/ ~billtj/ruby.html）中进行了解释。

许多人在学习 Ruby 前就已经通晓一种或多种编程语言。读者也许会发现使用一本认为必须教会自己编程和 Ruby 的厚书会妨碍学习 Ruby。对于这类读者，我们推荐 Ruby 创始人 Yukihiro Matsumoto 撰写的 Ruby User's Guide（http://www.ruby-doc.org/docs/ UsersGuide/rg/）。这是一本简短的读物，主要介绍 Ruby 与其他编程语言的差异，它的术语稍有些过时，并且其代码示例运行在独立的 eval.rb 程序上（使用 irb 替代），但它是我们所知最好的简介。

有一些文章特别适合于希望学习 Ruby 的 Java 程序员：Jim Weirich 的“10 Things Every Java Programmer Should Know About Ruby”、

（http://onestepback.org/articles/10things/）Francis Hwang 的博客文章“Coming to Ruby from Java”（http://fhwang.net/blog/40.html）和 Chris Williams 的链接收藏“From Java to Ruby (With Love)”（http:// cwilliams.textdriven.com/pages/java_to_ruby）。不管名字如何，C++ 程序员也会从这些文章中受益匪浅。

Ruby Bookshelf（http://books.rubyveil.com/books/Bookshelf/Introduction/Bookshelf）以易读的 HTML 格式提供了许多关于 Ruby 的免费书籍，包括上面提过的许多图书。

最后，Ruby 的内置模块、类和方法都有极好的文档记录（大多数最初为 Ruby 编程而编写）。读者可以在 http://www.ruby-doc.org/core/ 和 http://www.ruby-doc.org/stdlib/ 上在线阅读这些文档，还可以在自己安装的 Ruby 上使用 ri 命令进行查看。通过类或方法的名字，ri 将给出相应的文档。下面是几个示例：

```
$ ri Array                  # A class
$ ri Array.new              # A class method
$ ri Array#compact          # An instance method
```

本书使用的约定

下面是本书使用的印刷约定：

普通文本表示菜单题头、菜单选项、菜单按钮和键盘加速器（例如 Alt 与 Ctrl）。斜体（Italic）表示新术语、URL、电子邮件地址和 Unix 功能。

等宽字体（Constant Width）表示命令、选项、开关、变量、属性、键、函数、类型、类、命名空间、方法、模块、性质、参数、值、对象、事件、事件句柄、XML 标记、HTML 标记、宏、程序、库、文件名、路径名、目录、文件内容、或者命令输出。

等宽粗体（Constant Width Bold）给出应当由用户键入的命令或其他文本。等宽斜体（*Constant Width Italic*）给出应当由用户的提供值代替的文本。

使用代码示例

本书在这里要帮助读者完成自己的工作。一般而言，读者可以在自己的程序和文档中使用本书的代码。除非读者要重新生成代码的重要部分，否则无需联系我们获取许可。例如，使用本书的某

些代码段来编写程序无需请求许可。出售或发布 O'Reilly 书中示例的 CD-ROM 无需请求许可。引用本书及示例代码来回答问题无需请求许可。将本书的若干示例代码加入自己产品的文档无需请求许可。

我们鼓励但不要求列出参考出处。一个参考出处通常包括标题、作者、出版社及 ISBN。例如：“Ruby Cookbook by Lucas Carlson and Leonard Richardson，Copyright 2006 O'Reilly Media, Inc., 0-596-52369-6”。

如果您觉得自己对代码示例的使用超出了正当使用或上述许可的范围，望不吝联系我们：permissions@oreilly.com。

意见与问题

请通过以下地址把关于本书的评论和问题发送给出版社：美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室奥莱理软件（北京）有限公司

本书的 Web 页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/rubyckbk
http://www.oreilly.com.cn/book.php?bn=978-7-302-14770-1

如果想就本书的技术问题发表评论或咨询，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于本书、会议、资源中心和 O'Reilly 网络的更多信息，请访问 O'Reilly 的 Web 站点：

http://www.oreilly.com

http://www.oreilly.com.cn

感谢

首先要感谢我们的编辑 Michael Loukides，感谢他的帮助以及默许我们在秘诀示例代码中使用他的名字，即便是我们将他写成一只会说话的青蛙。制作编辑 Colleen Gorman 也提供了许多帮助。

如果没有我们的投稿作者，这本书的编写时间将会更长，并且会缺乏趣味，他们总共编写了 60% 以上的秘诀。名单包括：Steve Arniel、Ben Bleything、Antonio Cangiano、Mauro Cicio、Maurice Codik、Thomas Enebo、Pat Eyler、Bill Froelich、Rod Gaither、Ben Giddings、Michael Granger、James Edward Gray II、Stefan Lang、Kevin Marshall、Matthew Palmer、Chetan Patil、Alun ap Rhisiart、Garrett Rooney、John-Mason Shackelford、Phil Tomson 和 John Wells。他们自己对各种 Ruby 主题的丰富知识节约了我们的时间，并以其思想极大地丰富了本书内容。

如果没有我们的技术审核员，那么本书的质量可能会低得令人毛骨悚然，他们指出了大量的 bug、特定平台问题以及概念错误，他们是：John N. Alegre、Dave Burt、Bill Dolinar、Simen Edvardsen、Shane Emmons、Edward Faulkner、Dan Fitzpatrick、Bill Guindon、Stephen Hildrey、Meador Inge、Eric Jacoboni、Julian I. Kamil、Randy Kramer、Alex LeDonne、Steven Lumos、Keith Rosenblatt、Gene Tani 和 R Vrajmohan。

最后要感谢 Ruby 社区的程序员和作者，不管是如 Yukihiro Matsumoto、Dave Thomas、Chad Fowler 和“why”这样的名人，还是其工作被加入本书举例所用库中的成百上千位无名英雄，他们的能力与耐心会持续不断地将更多的人带入 Ruby 社区。

参加本书翻译的人员有：王欣轩、郑路长、陈宗斌、蔡京平、李毅、毕蓉蓉、祁海生、张贺乾、史宁、刘绿生、孙雷、蔡加双、安东辉、米翔娟、刘颜、王宇宇、沈程亮、陆晓萍、金国良、俞群、李正智、赵敏、陈征、陈红霞、张景友、易小丽、陈婷、管学岗、王新彦、金惠敏、张海峰、徐晔、戴锋、张德福、张士华、张锁玲、杜明宗、高玉琢、王涛、刘晓捷、董礼、何永利、李楠、陈宁、房金萍、黄骏衡、黄绪民、焦敬俭、李军、刘瑞东、潘曙光、蒲书箴、邵长凯、郁琪琳、张广东、梁永翔、刘冀得、孙先国、张淑芝、张路红、程明、李大成、张春林、刘淑妮、侯经国、宫飞、高德杰、李振国、孙玲、申川。

第 3 章 日期与时间

3．1 查找当前日期]

如果没有时间的概念，我们的生活将会是一团糟。没有软件程序坚持不懈地好好管理和记录我们宇宙的这一奇妙方面，我们实际上可能会过得好一些。但是，为什么要冒这个险呢？

有些程序代表人们管理现实时间，否则人们就必须自己进行管理：日历、时间表，以及科学实验的数据收集器。另一些程序为了自己的目的而使用人类的时间概念：它们可能要运行自己的实验、根据微秒变化量进行决策。有时为与时间毫无关系的对象提供时间戳以记录创建或上一次修改它们的时间。就基本数据类型而言，时间只有一种，它直接对应于现实世界中的时间。

Ruby 支持用户过去习惯使用的来自其他编程语言的日期和时间接口，但是这些接口的顶层是令编程更简单的 Ruby 专用方言。在本章中，我们将说明如何使用这些接口和方言，以及如何便捷地弥合该语言带来的差异。

Ruby 实际上有两种不同的时间实现。有一组用 C 语言编写的时间库，它们已经出现了数十年。与大多数现代编程语言一样，Ruby 自带有对这些 C 库的接口。这些库功能强大、实用并且可靠，但是它们也有缺点，为此，Ruby 补充了另一个以纯 Ruby 编写的时间库。纯 Ruby 库不能用于所有地方，因为它比 C 接口要慢，而且缺少某些深藏于 C 库中的特性，比如，夏令时的管理。

Time 类包含 Ruby 对 C 库的接口，也是大多数应用程序所需要的。Time 类上附带有许多 Ruby 方言，但大多数方法拥有奇怪的非 Ruby 类的名字，如 strftime 和 strptime。这有利于已经用过 C 库或者它的其他接口（类似于 Perl 或 Python 中的接口）的用户。

Time 对象的内部表示是“time zero（时间零点）”之前或之后的秒数。Ruby 的时间零点是 Unix 纪元：GMT 1970 年 1 月 1 日第 1 秒。使用 Time.now 或者使用 Time.at 创建一个来自 seconds-since-epoch 的 Time 对象可以获得当前本地时间。

```
Time.now                                    # => Sat Mar 18 14:49:30 EST
2006
Time.at(0)                                  # => Wed Dec 31 19:00:00 EST
1969
```

这一时间的数字内部表示作为人类可读的表示不是那么有用。可以如前所示获得 Time 的字符串表示，或者调用 accessor 方法根据人类计算时间的方式分割时间实例：

```
t = Time.at(0)t.sec # => 0t.min # => 0t.hour # => 19t.day # => 31t.month # =
> 12t.year # => 1969t.wday # => 3 # Numeric day ofweek; Sunday is 0t.yday #
=> 365 # Numeric day ofyeart.isdst # => false # Is Daylight SavingTime in
                                                # effect?
t.zone # => "EST" # Time zone
```

有关更多将 Time 对象分割为人类可读的方式请参见 3.3 节。

除了难用的方法和成员名字之外，Time 类的最大缺点在于它的基本实现在 32 位系统上无法处理 1901 年 12 月之前或 2037 年 1 月之后的日期。

```
Time.local(1865, 4, 9)
# ArgumentError: time out of range
```

```
Time.local(2100, 1, 1)
# ArgumentError: time out of range
```

为了表示这些时间，需要将其转换为 Ruby 的其他时间实现：Date 和 DateTime 类。可以在所有地方都使用 DateTime，而根本无需使用 Date：

```
require 'date'DateTime.new(1865, 4, 9).to_s # => "1865-04-09T00:00:00Z"D
ateTime.new(2100, 1, 1).to_s # => "2100-01-01T00:00:00Z"
```

回忆一下，Time 对象存储为 1970 年“时间零点”之后的相对秒数。Date 或 DateTime 对象的内部表示是天文儒略历日期：从 6,000 多年前 4712 BCE 的“时间零点”开始的相对天数。

```
# Time zero for the date library:
DateTime.new.to_s # => "-4712-01-01T00:00:00Z"

# The current date and time:
DateTime::now.to_s # => "2006-03-18T14:53:18-0500"
```

一个 DateTime 对象可以精确地表示远自古老的宇宙、或者遥至宇宙可预测生命的时间。当 DateTime 处理历史日期时，它需要考虑近 500 年来席卷西方世界的历法改革运动。有关创建 Date 和 DateTime 对象的更多信息请参见 3.1 节。

DateTime 在天文和历史应用中显然比 Time 要高级，但是，用户可以在大多数日常程序中使用 Time。下表应当给出了 Time 对象和 DateTime 对象的相对优势图。

	Time	DateTime
日期范围在 32 位系统上为	1901?2037	实际上地无限
处理夏令时	Yes	No
处理历法改革	No	Yes
时区转换	使用 tz gem 很简单	除非只考虑时区偏移，否则很难
类似 RFC822 的普通时间格式内置		自己编写
速度	较快	较慢

Time 和 DateTime 对象皆支持类似迭代和日期算法的细节：基本上可以将它们视为数字，因为它们在内部被存储为数字。但是，回忆一下，Time 对象被存储为秒数，而 DateTime 对象被存储为天数，因此，相同的操作会作用于 Time 和 DateTime 对象上的不同时间度量。更多信息请参见 3.4 节和 3.5 节。

到目前为止，我们已经讨论了编写代码及时管理特定时刻：过去、未来、或者当下的某个时刻。时间的另一种用途是持续时间，即两个时期的关系：“开始”与“结束”、“之前”与“之后”。可以通过用一个 DateTime 对象减去另一个 DateTime 对象或者用一个 Time 对象减去另一个 Time 对象来测量持续时间：可以得到按天或秒计算的测量结果（参见 3.5 节）。如果用户希望自己的程序实际经历持续时间（现在与未来某个时间之间的不同），那么可以让某个线程睡眠特定时间量：参见 3.12 节和 3.13 节。在开发过程中也许会经常需要使用持续时间。基准测试与 profiling 能够度量用户程序开始运行的时间长度以及运行时间最长的部分。这些主题在第 17 章中介绍：参见 17.12 节与 17.13 节。

3.1 查找当前日期问题

要创建一个对象来表示当前日期和时间、或者未来或过去某个时间。

解决方案

工厂方法 Time.now 创建一个包含当前本地时间的 Time 对象。如果用户愿意，可以随后通过调用 Time#gmtime 将其转换为 GML 时间。尽管 gmtime 方法没有遵循 Ruby 对该类方法的命名惯例（其名字应当类似 gmtime!），但实际上是它修改下面的时间对象。

```
now = Time.now # => Sat Mar 18 16:58:07 EST 2006now.gmtime # => Sat Mar 18 21:58:07 UTC 2006

#The original object was affected by the time zone conversion.
now # => Sat Mar 18 21:58:07 UTC 2006
```

要为当前本地时间创建一个 DateTime 时间，可使用工厂方法 DateTime.now。调用不带参数的 DateTime#new_offset 可以将 DateTime 对象转换为 GMT，该方法与 Time#gmtime 不同，它返回另一个 DateTime 对象，而不是就地修改原始对象。

```
require 'date'now = DateTime.now# => #<DateTime: 70669826362347677/28800000000,-5/24,2299161>now.to_s # => "2006-03-18T16:58:07-0500"now.new_offset.to_s # => "2006-03-18T21:58:07Z"
#The original object was not affected by the time zone conversion.
now.to_s # => "2006-03-18T16:58:07-0500"
```

讨论

Time 与 DateTime 对象皆为西方历法和时钟划分时间的基本方法提供了访问器方法。两个类皆提供了 year、month、day、hour（24 小时格式）、min、sec 和 zone 访问器。Time#isdst 让用户知道一个 Time 对象的时间是否被其时区中的夏令时所修改。DateTime 佯装夏令时不存在。

```
now_time = Time.new

now_datetime = DateTime.now


now_time.year # => 2006now_datetime.year # => 2006now_time.hour # => 18now_datetime.hour # => 18
now_time.zone # => "EST"now_datetime.zone # => "-0500"now_time.isdst # => false
```

可以看出 Time#zone 与 DateTime#zone 有些许不同。Time#zone 返回时区名或其缩写，而 DateTime#zone 返回字符串形式的自 GMT 的偏移数。可以调用 DateTime#offset 得到 GMT 偏移数：相对天数。

```
now_datetime.offset # => Rational(-5, 24) # -5 hours
```

两个类也都能够表示秒的分数，使用 Time#usec 和 DateTime#sec_fraction 可以访问（即，msec 或微秒）。在上例中，DateTime 对象在 Time 对象后被创建，因此即使两个对象在同一秒内被创建，该数字也是不同的。

```
now_time.usec # => 247930# That is, 247930 microsecondsnow_datetime.sec_fraction # => Rational(62191, 21600000000)# That is, about 287921 microseconds
```

date 库提供了一个类似 DateTime 的不带时间的 Date 类。要创建一个包含当前日期的 Date 对象，最好的策略是创建一个 DateTime 对象并使用对 Date 工厂方法的调用结果。DateTime 实际是 Date 的子类，因此如果要精炼时间数据以确保它未得到使用，那么只需要这样做即可。

```
class Date
  def Date.now
    return Date.jd(DateTime.now.jd)
  end
end
puts Date.now
# 2006-03-18
```

除了上述这样创建一个 time 对象之外，可以由一个字符串（参见 3.2 节）或由另一个 time 对象（参见 3.5 节）创建一个 time 对象，还可以使用工厂方法根据其日历和时钟部分创建 time 对象：年、月、日，等等。

工厂方法 Time.local 和 Time.gm 此时需要以 Time 对象为参数。对于本地时间，要使用 Time.local；对于 GMT，要使用 Time.gm。year 之后的所有参数是可选的，默认值为 0。

```
Time.local(1999, 12, 31, 23, 21, 5, 1044)

# => Fri Dec 31 23:21:05 EST 1999


Time.gm(1999, 12, 31, 23, 21, 5, 22, 1044)


# => Fri Dec 31 23:21:05 UTC 1999
Time.local(1991, 10, 1)
# => Tue Oct 01 00:00:00 EDT 1991

Time.gm(2000)
# => Sat Jan 01 00:00:00 UTC 2000
```

Time.local 的 DateTime 等价方法是内部工厂方法，其参数大部分与 Time.local 相同，但不完全相同：

```
[year, month, day, hour, minute, second, timezone_offset,date_of_calendar_reform].
```

Time.local 与 Time.gmt 的主要不同点在于：

- 没有用于秒分数的单独 usec 参数。通过传递用于秒的有理数可以表示相对秒数。
- 所有参数皆是可选的。但是，默认的年份为 4712 BCE，这对用户而言可能没什么用处。
- 除了为不同时区提供不同方法外，用户传递 GMT 偏移作为相对天数。默认值为零，表示不带时区调用 DateTime.civil 会给出 GMT 时间。

```
DateTime.civil(1999, 12, 31, 23, 21, Rational(51044, 100000)).to_s
# => "1999-12-31T23:21:00Z"
```

```
DateTime.civil(1991, 10, 1).to_s
# => "1991-10-01T00:00:00Z"

DateTime.civil(2000).to_s
# => "2000-01-01T00:00:00Z"
```

获得关于本地时区的 GMT 偏移的最简单方法是在 DateTime.now 的结果上调用 offset，然后将偏移传递给 DateTime.civil：

```
my_offset = DateTime.now.offset              # => Rational(-5, 24)

DateTime.civil(1999, 12, 31, 23, 21, Rational(51044, 100000),
my_offset).to_s
# => "1999-12-31T23:21:00-0500"
```

哦，还有历法改革的问题。回忆一下，Time 对象只能表示有限范围内的时间（在 32 位系统上，可表示 20 世纪和 21 世纪的日期）。DateTime 对象能够表示任何时间。得到这一更大范围的代价是 DateTime 在处理历史日期时需要考虑历法改革。如果使用老日期，那么可能会出现从儒略历（它规定每四年有一个闰年）至更精确的公历（它有时会跳过闰年）转换而导致的时差。这一转换在不同国家的不同时期发生，当地日历在承受由于许多世纪以来使用儒略历而导致的额外闰日时会产生大小不一的时差。某个国家的时差内所生成的日期对该国来说是不合法的。

默认情况下，Ruby 假设用户创建的 Date 对象是相对于儒略历的，它于 1582 年被转换为公历。对于美国和英吉利共和国用户而言，Ruby 提供了常数 Date::ENGLAND，它对应于采用公历的英国及其殖民地的日期。DateTime 的构造器和工厂方法将接受 Date::ENGLAND 或 Date::ITALY 为额外参数，表示历法改革在该国的起始时间。历法改革参数还可以是任意老儒略历日期，以便处理来自任意国家的老日期：

```
#In Italy, 4 Oct 1582 was immediately followed by 15 Oct 1582.
#
Date.new(1582, 10, 4).to_s
# => "1582-10-04"
Date.new(1582, 10, 5).to_s
# ArgumentError: invalid date
Date.new(1582, 10, 4).succ.to_s
# => "1582-10-15"

#In England, 2 Sep 1752 was immediately followed by 14 Sep 1752.
#
Date.new(1752, 9, 2, Date::ENGLAND).to_s
# => "1752-09-02"
Date.new(1752, 9, 3, Date::ENGLAND).to_s
# ArgumentError: invalid date
Date.new(1752, 9, 2, DateTime::ENGLAND).succ.to_s
# => "1752-09-14"
Date.new(1582, 10, 5, Date::ENGLAND).to_s
# => "1582-10-05"
```

可能无需使用 Ruby 的公历转换特性：计算机应用程序需要处理同时以精确和与特定地点相关而著称的老日期的情况并不常见。

3.2 精确或模糊地分析日期问题

要将一个描述日期或日期/ 时间的字符串转换为一个 Date 对象，其中，时间前面的字符串格式可能并不知道。

解决方案

最好的解决方案是将日期字符串传递给 Date.parse 或 DateTime.parse，这些方法使用启发式算法猜测字符串格式，并且能够很好地完成任务：

```
require 'date'

Date.parse('2/9/2007').to_s
# => "2007-02-09"

DateTime.parse('02-09-2007 12:30:44 AM').to_s
# => "2007-09-02T00:30:44Z"

DateTime.parse('02-09-2007 12:30:44 PM EST').to_s
# => "2007-09-02T12:30:44-0500"

Date.parse('Wednesday, January 10, 2001').to_s
# => "2001-01-10"
```

讨论

parse 方法能够节约大量其他编程语言中与分析次数相关的工作，但它们有时无法给出所需要的结果。要注意第一个示例中 Date.parse 假设 2/9/2007 是美式日期（月份在前）而不是欧式日期（日期在前）的方法。parse 还会错误地解释两位数的年份：

```
Date.parse('2/9/07').to_s # => "0007-02-09"
```

我们说 Date.parse 无法工作，但是，读者知道所有要处理的日期都会以某种方式进行格式化。可以使用标准 strftime 指令创建一个格式字符串并与日期字符串一起传递给 DateTime.strptime 或 Date.strptime。如果日期字符串与格式字符串匹配，那么将返回一个 Date 或 DateTime 对象。由于许多语言以及 Unix 的 date 命令都以这种方式进行日期格式化，因此，读者可能已经熟悉了这种技术。

某些常见的日期和时间格式包括：

```
american_date = '%m/%d/%y'Date.strptime('2/9/07', american_date).to_s # =>
 "2007-02-09"DateTime.strptime('2/9/05', american_date).to_s # => "2005-02-0
9T00:00:00Z"Date.strptime('2/9/68', american_date).to_s # => "2068-02-09"Dat
e.strptime('2/9/69', american_date).to_s # => "1969-02-09"

european_date = '%d/%m/%y'Date.strptime('2/9/07', european_date).to_s # =

> "2007-09-02"Date.strptime('02/09/68', european_date).to_s # => "2068-09-0

2"Date.strptime('2/9/69', european_date).to_s # => "1969-09-02"

four_digit_year_date = '%m/%d/%Y'Date.strptime('2/9/2007', four_digit_year_date).to
_s # => "2007-02-09"Date.strptime('02/09/1968', four_digit_year_date).to_s # => "
1968-02-09"Date.strptime('2/9/69', four_digit_year_date).to_s # => "0069-02-09"
```

```
date_and_time = '%m-%d-%Y %H:%M:%S %Z'
DateTime.strptime('02-09-2007 12:30:44 EST', date_and_time).to_s
# => "2007-02-09T12:30:44-0500"
DateTime.strptime('02-09-2007 12:30:44 PST', date_and_time).to_s
# => "2007-02-09T12:30:44-0800"
DateTime.strptime('02-09-2007 12:30:44 GMT', date_and_time).to_s
# => "2007-02-09T12:30:44Z"
```

```
twelve_hour_clock_time = '%m-%d-%Y %I:%M:%S %p'
DateTime.strptime('02-09-2007 12:30:44 AM', twelve_hour_clock_time).to_s
# => "2007-02-09T00:30:44Z"
DateTime.strptime('02-09-2007 12:30:44 PM', twelve_hour_clock_time).to_s
# => "2007-02-09T12:30:44Z"
```

```
word_date = '%A, %B %d, %Y'
Date.strptime('Wednesday, January 10, 2001', word_date).to_s
# => "2001-01-10"
```

如果日期字符串是有限数目的格式中的一种，那么可以在一定范围的格式字符串上进行迭代并使用每种格式依次试着分析日期字符串。这会带来 Date.parse 的某些灵活性，但要求忽略它所制定的假设。Date.parse 仍然是较快的，因此只要它能工作，最好使用它。

```
Date.parse('1/10/07').to_s # => "0007-01-10"
Date.parse('2007 1 10').to_s
# ArgumentError: 3 elements of civil date are necessary
```

```
TRY_FORMATS = ['%d/%m/%y', '%Y %m %d']

def try_to_parse(s)
  parsed = nil
  TRY_FORMATS.each do |format|
    begin
      parsed = Date.strptime(s, format)
      break

    rescue ArgumentError
    end
  end
  return parsed
```

```
end

try_to_parse('1/10/07').to_s # => "2007-10-01"try_to_parse('2007 1 10').to_s # =>
"2007-01-10"
```

有些常见的日期格式无法可靠地由 strptime 格式字符串表示。Ruby 定义了 Time 的类方法来分析这些日期字符串，因此不必自己编写代码。下列每个方法皆返回一个 Time 对象。Time.rfc822 分析 Internet 邮件标准 RFC822/RFC2822 格式的日期字符串，在 RFC2822 日期中，月份与周日期总以英语表示（例如，“Tue”和“Jul”），即使使用某种其他语言也一样。

```
require 'time'
mail_received = 'Tue, 1 Jul 2003 10:52:37 +0200'
Time.rfc822(mail_received)
# => Tue Jul 01 04:52:37 EDT 2003
```

要分析 HTTP 标准 RFC2616 格式的日期需使用 Time.httpdate。RFC2616 日期指在 HTTP 头部看到的类似 Last-Modified 的日期种类。使用 RFC2822 时，月份与日期的缩写始终是英语：

```
last_modified = 'Tue, 05 Sep 2006 16:05:51 GMT'
Time.httpdate(last_modified)
# => Tue Sep 05 12:05:51 EDT 2006
```

要分析 XML Schema 的 ISO8601 格式的日期，需使用 Time.iso8601 或 Time.xmlschema：

```
timestamp = '2001-04-17T19:23:17.201Z't = Time.iso8601(timestamp) # => Tue Apr 17
 19:23:17 UTC 2001t.sec # => 17t.tv_usec # => 201000
```

不要将 Time 的这些类方法与同名的实例方法相混淆。类方法由字符串创建 Time 对象，而实例方法用其他方法将现有的 Time 对象格式化为字符串：

```
t = Time.at(1000000000) # => Sat Sep 08 21:46:40 EDT 2001t.rfc822 # => "S
at, 08 Sep 2001 21:46:40 -0400"t.httpdate # => "Sun, 09 Sep 2001 01:46:40 G
MT"t.iso8601 # => "2001-09-08T21:46:40-04:00"
```

参考

- Time#strftime 方法的 RDoc 列出大多数支持的 strftime 指令（ri Time#strftime），有关更详细和完整的列表请参见 3.3 节“打印日期”中的列表

3.3 打印日期问题

要将日期对象打印为字符串。

解决方案

如果只想看到某个日期，那么可以调用 Time#to_s 或 Date#to_s 而不必为喜爱的格式费心：

```
require 'date'Time.now.to_s # => "Sat Mar 18 19:05:50 EST2006"DateTime.now.to_s # => "2006-03-18T19:05:50-0500"
```

如果需要特殊格式的日期，那么需要将该格式定义为包含时间格式化指令的字符串。将格式字符串传递给 Time#strftime 或 Date#strftime，将会返回一个字符串，其中的格式化指令已被 Time 或 DateTime 对象的对应部分代替。

格式化指令看起来类似于一个百分号和一个字母：%x。格式字符串中所有不是格式化指令的内容皆被视为文字：

```
Time.gm(2006).strftime('The year is %Y!') # => "The year is 2006!"
```

讨论小节中列出了 Time#strftime 和 Date#strftime 定义的所有时间格式化指令。下面是某些常见的时间格式化字符串，以日期为 2005 年最后一天 GMT 下午 1:30 为例。

```
time = Time.gm(2005, 12, 31, 13, 22, 33)american_date = '%D'time.strftime(american_date) # => "12/31/05"european_date = '%d/%m/%y'time.strftime(european_date) # => "31/12/05"four_digit_year_date = '%m/%d/%Y'time.strftime(four_digit_year_date) # => "12/31/2005"date_and_time = '%m-%d-%Y %H:%M:%S %Z'time.strftime(date_and_time) # => "12-31-2005 13:22:33 GMT"twelve_hour_clock_time = '%m-%d-%Y %I:%M:%S %p'time.strftime(twelve_hour_clock_time) # => "12-31-2005 01:22:33 PM"word_date = '%A, %B %d, %Y'time.strftime(word_date) # => "Saturday, December 31,2005"
```

讨论

铅印格式纸、分析器和人对日期的格式都非常挑剔。拥有标准格式的日期可以令日期易读并且易于查找错误。遵循某种格式还能避免歧义（4/12 指 12 月 4 日还是 4 月 12 日？）。

如果需要“时间”，那么 Time 对象会针对常用日期表示标准产生特殊目的格式化方法：Time#rfc822、Time#httpdate 和 Time#iso8601。这些能够容易地以遵循邮件、HTTP 和 XML 标准的格式打印日期：

```
require 'time'

time.rfc822 # => "Sat, 31 Dec 2005

13:22:33 -0000"time.httpdate # => "Sat, 31 Dec 2005 13:22:33GMT"time.iso8601 # => "2005-12-31T13:22:33Z"
```

DateTime 仅提供了上述三种格式中的一种。ISO8601 是 DateTime 对象的默认字符串表示（调用#to_s 所得到的结果）。这意味着无需将 DateTime 对象转换为 Time 对象就可以容易地将其打印在 XML 文档中。

对于其他两种格式，最好的策略是将 DateTime 转换为 Time 对象（细节请参见 3.9 节）。即使在使用 32 位时间计数器的系统上，DateTime 对象也适合 Time 所支持的 1901~2037 范围，因为 RFC822 和 HTTP 日期几乎都使用近期或不远将来的日期。

有时需要定义自定义的日期格式。Time#strftime 和 Date#strftime 定义了许多用于格式化字符串的指令。下面的大表列出了它们的工作，可以在一个格式化字符串中任意地联合使用。
用户可能熟悉其中某些来自其他编程语言的指令，事实上，自从 C 包括了一个使用其中某些指令的 strftime 实现之后，所有语言皆包括了它们，其中某些指令是 Ruby 独有的。

格式化指令	工作内容	以2005年12月31日13:22:33为例
%A	英语表示周日期	“Saturday”
%a	英语缩写表示周日期	“Sat”
%B	英语表示年的月份	“December”
%b	英语表示年的月份	“Dec”
%C	年份的世纪数，必要时补零	“20”
%c	以类似 Time 的默认字符串表示方式打印日期和时间，但不带时区。等价于 %a %b %e %H:%M:%S %Y'	"Sat Dec 31 13:22:33 2005"
%D	带两位数年份的美式短日期格式。等价于“%m/%d/%y”	12/31/05
%d	月份的各天，补零	“31”
%e	月份的各天，不补零	“31”
%F	带四位数年份的短日期格式。等价于“%Y-/%m/%d”	2005-12-31
%G	带世纪数的商业财政年度，	“2005”

	用零补全至四位数，对于BCE日期在前面预加一个减号（参见3.11节。对于公历使用%Y）	
%g	不带世纪数的年份，用零补全至两位数	“05”
%H	日期的小时数，24小时时钟，用零补全至两位数	“13”
%h	年的缩写月份，与“%b”相同	“Dec”
%I	日期的小时数，12小时时钟，用零补全至两位数	“01”
%j	年的儒略历天数，补全至三位数（从001至366）	“365”
%k	日期的小时数，24小时时钟，无零补全，类似%H但不带补全	“13”
%l	日期的小时数，12小时时钟，无零补全，类似%I但不带补全	“1”
%M	小时的分钟数，补全为两位数	“22”
%m	年的月份，补全为两位数	“12”
%n	换行符，不要使用它，它仅在格式化字符串中放置一个换行符	“\n”
%P	小写正午指示器（“am”或“pm”）	“pm”
%p	大写正午指示器。类似%P，但给出“AM”或“PM”；是的，大写P给出小写的正午指示器，反之亦然	“PM”
%R	短24小时时间格式，等价于“%H:%M”	“13:22”
%r	长12小时时间格式，等价于	“01:22:33 PM”

	“%I:%M:%S %p”	
%S	分钟的秒数，用零补全至两位数	“33” \
%s	自 Unix 纪元以来的秒数	“1136053353”
%T	长 24 小时时间格式，等价于“%H:%M:%S”	“13:22:33”
%t	一个制表符，不要使用它，它只在格式化字符串中放置一个制表符	“\t”
%U	年的历法周数：假设年的第一周开始于第一个星期一；如果日期来自于该年的第一个星期日之前，那么它被计算为“第零周”的一部分并返回“00”	“52”
%u	年的商用周日期，从 1 至 7，星期一为第 1 日	“6”
%V	年的商用周数（参见 3.11 节）	“52”
%W	与 %V 相同，但是，如果日期“52”在该年第一个星期一之前，那么它被计算为“第零周”的一部分并返回“00”	
%w	周的历日，从 0 至 6，星期日为第 0 日	“6”
%X	时间的首选表示，等价于“%H:%M:%S”	“13:22:33”
%x	日期的首选表示，等价于“%m/%d/%y”	“12/31/05”
%Y	带世纪的年份，用零补全至四位数，并在 BCE 日期前加一个减号	“2005”
%y	不带世纪的年份，用零补全至两位数	“05”
%Z	时区缩写（Time）或偏移（Date）。如果时间是	GMT对Time为“GMT”，对Date为“Z”

	GMT 时间，那么 Date 使用“Z”代替“+0000”	
%z	作为 GMT 偏移的时区“+0000”	
%%	文字百分号	“%”
%v	带月份缩写的欧式日期格式，等价于“%e-%b-%Y”	31-Dec-2005
%+	打印 Date 对象，仿佛它是一个被转换为字符串的 Time 对象；类似于 %c，但包括时区信息；等价于“%a %b %e %H:%M:%S %Z %Y”	13:22:33 Z 2005

Date 定义了两种在 Time#strftime 中完全不起作用的格式化指令。这两种都是格式化手工创建的字符串的捷径。

如果需要一种日期格式但没有适用的格式化指令，那么应当能够通过编写 Ruby 代码进行弥补。例如，假设要将我们的示例格式化为“The 31st of December”。没有专门的格式化指令将日期打印为序数，但是，可以使用 Ruby 代码构建一个给出正确答案的格式化字符串。

```
class Time
  def day_ordinal_suffix
  if day == 11 or day ==
  = 12
  return "th"

  else

  case day % 10

  when 1 then return "st"

  when 2 then return "nd"

  when 3 then return "rd"

  else return "th"

  end

  end

end

end
```

```
time.strftime("The %e#{time.day_ordinal_suffix} of %B") # => "The 31st of
December"
```

实际的格式化字符串根据日期而有所不同。在这里，它以“The %est of %B”结束，但对其他日期可能会是“The %end of %B”、“The %erd of %B”或“The %eth of %B”。

参考

•Time 对象能够分析并打印常见的日期格式；有关如何分析 strftime、rfc822、httpdate 和 iso8661 的输出请参见 3.2 节“精确或模糊地分析日期”

3.4 日期上的迭代问题

给定一个时间点，希望得到其他时间点。

解决方案

Ruby 的所有时间对象都可像数字一样用在值域中。Date 和 DateTime 对象按天递增，而 Time 对象按秒递增：

```
require 'date'
(Date.new(1776, 7, 2)..Date.new(1776, 7, 4)).each { |x| puts x }
# 1776-07-02
# 1776-07-03
# 1776-07-04
```

```
span = DateTime.new(1776, 7, 2, 1, 30, 15)..DateTime.new(1776, 7, 4, 7, 0,
0)
span.each { |x| puts x }
# 1776-07-02T01:30:15Z
# 1776-07-03T01:30:15Z
# 1776-07-04T01:30:15Z
```

```
(Time.at(100)..Time.at(102)).each { |x| puts x }
# Wed Dec 31 19:01:40 EST 1969
# Wed Dec 31 19:01:41 EST 1969
# Wed Dec 31 19:01:42 EST 1969
```

Ruby 的 Date 类定义了 step 和 upto 两种方便的由数字使用的迭代器：

```
the_first = Date.new(2004, 1, 1)
the_fifth = Date.new(2004, 1, 5)

the_first.upto(the_fifth) { |x| puts x }

# 2004-01-01

# 2004-01-02

# 2004-01-03

# 2004-01-04

# 2004-01-05
```

讨论

Ruby 的日期对象在内部被存储为数字，并且一定范围的这种对象可被视为一定范围的数字。对于 Date 和 DateTime 对象而言，内部表示是儒略历日期：每次在一定范围的这种对象上累加一天。对于 Time 对象而言，内部表示是自 Unix 纪元以来的秒数：每次在一定范围的 Time 对象上累加一秒。Time 没有定义 step 和 upto 方法，但添加它们也很简单：

```
class Time

  def step(other_time, increment)

    raise ArgumentError, "step can't be 0" if increment == 0
    increasing = self < other_time
    if (increasing && increment < 0) || (!increasing && increment > 0)

      yield self

      return
    end
    d = self
    begin

    yield d
    d += increment
    end while (increasing ? d <= other_ti
    me : d >= other_time)
    end

  def upto(other_time)
    step(other_time, 1) { |x| yi
    eld x }
    end
  end

the_first = Time.local(2004, 1, 1)
the_second = Time.local(2004, 1, 2)
the_first.step(the_second, 60 * 60 * 6) { |x| puts x }
# Thu Jan 01 00:00:00 EST 2004
# Thu Jan 01 06:00:00 EST 2004
# Thu Jan 01 12:00:00 EST 2004
# Thu Jan 01 18:00:00 EST 2004
# Fri Jan 02 00:00:00 EST 2004

the_first.upto(the_first) { |x| puts x }
# Thu Jan 01 00:00:00 EST 2004
```

参考

- 2.15 节“生成数字序列”

3.5 计算日期问题

要知道两个日期之间流逝了多少时间，或者为一个日期添加某个数以得到较早或较迟的另一日期。

解决方案

为一个 Time 对象加上或减去某个数等于为对象加上或减去数字大小的秒数。为一个 Date 对象加上或减去某个数等于为对象加上或减去数字大小的天数：

```
require 'date'y2k = Time.gm(2000, 1, 1) # => Sat Jan 01 00:00:00 UTC2000y2k + 1 # => Sat Jan 01 00:00:01 UTC2000y2k - 1 # => Fri Dec 31 23:59:59 UTC1999y2k + (60 * 60 * 24 * 365) # => Sun Dec 31 00:00:00 UTC2000

y2k_dt = DateTime.new(2000, 1, 1)(y2k_dt + 1).to_s # => "2000-01-02T00:00:00Z"(y2k_dt - 1).to_s # => "1999-12-31T00:00:00Z"(y2k_dt + 0.5).to_s # => "2000-01-01T12:00:00Z"(y2k_dt + 365).to_s # => "2000-12-31T00:00:00Z"
```

用一个 Time 对象减去另一个 Time 对象将给出两个日期之间间隔的秒数。用一个 Date 对象减去另一个 Date 对象会给出间隔的天数（注 1）：

```
day_one = Time.gm(1999, 12, 31)day_two = Time.gm(2000, 1, 1)day_two - day_one # => 86400.0day_one - day_two # => -86400.0
day_one = DateTime.new(1999, 12, 31)day_two = DateTime.new(2000, 1, 1)day_two - day_one # => Rational(1, 1)day_one - day_two # => Rational(-1, 1)
# Compare times from now and 10 seconds in the future.
before_time = Time.now
before_datetime = DateTime.now
sleep(10)
Time.now - before_time # => 10.003414
```

注 1：Facts More 库也是如此计算。

```
DateTime.now - before_datetime # => Rational(5001557, 43200000000)
```

Ruby on Rails 必备的 activesupport gem 在 Numeric 和 Time 上定义了许多用于操控时间的有用函数：

```
require 'rubygems'
require 'active_support'

10.days.ago # => Wed Mar 08 19:54:17 EST20061.month.from_now # => Mon Apr 17 20:54:17 EDT20062.weeks.since(Time.local(2006, 1, 1)) # => Sun Jan 15 00:00:00 EST2006

y2k - 1.day # => Fri Dec 31 00:00:00 UTC1999y2k + 6.3.years # => Thu Apr 20 01:48:00 UTC20066.3.years.since y2k # => Thu Apr 20 01:48:00 UTC2006
```

讨论

Ruby 的日期算法利用了 Ruby 将时间对象在内部存储为数字这一事实。添加日期或计算日期间隔可通过增加或减去相应数字完成。这就是向 Time 加 1 等于加 1 秒而向 DateTime 加 1 等于加 1 天的原

因：Time 被存储为自时间零点以来的秒数，而 Date 或 DateTime 被存储为自（另一不同）时间零点以来的天数。

并非所有算法操作都对日期有意义：可以用相应数字的乘法计算“两个日期相乘”，但这对实际的时间而言是没有意义的，因此 Ruby 没有定义这种运算符。一旦数字在真实世界中具有了意义，对该数字所进行的合理操作就有了限制。

下面是增加或减去大量时间的一种捷径：在 Date 或 DateTime 对象上使用 right-shift 或 left-shift 运算符可以在日期上增加或减去一定数量的月数。

(y2k_dt >> 1).to_s # => "2000-02-01T00:00:00Z"(y2k_dt << 1).to_s # => "1999-12-01T00:00:00Z"

使用 activesupport 的 Numeric#month 方法可以执行类似的行为，但该方法假设一个“月”有 30 天，而不是根据特定月份的长度进行处理：

```
y2k + 1.month # => Mon Jan 31 00:00:00 UTC2000y2k - 1.month # => Thu Dec 02 00:00:00 UTC1999
```

相比而言，如果终止月份没有足够的天数（例如，起始月份有 31 天，然后要转到只有 30 天的另一个月），那么标准库将使用新月份的最后一天：

```
# Thirty days hath September...halloween = Date.new(2000, 10, 31)(halloween << 1).to_s # => "2000-09-30"(halloween >> 1).to_s # => "2000-11-30"(halloween >> 2).to_s # => "2000-12-31"
leap_year_day = Date.new(1996, 2, 29)(leap_year_day << 1).to_s # => "1996-01-29"(leap_year_day >> 1).to_s # => "1996-03-29"(leap_year_day >> 12).to_s # => "1997-02-28"(leap_year_day << 12 * 4).to_s # => "1992-02-29"
```

参考

- 3.4 节“日期上的迭代”
- 3.6 节“从任意日期开始计算天数”
    - Rails 的 ActiveSupport::CoreExtensions::Numeric::Time 模块的 RDoc

3.6 从任意日期开始计算天数问题

要知道从某一特定日期开始过了多少天，或者到未来某个日期为止还有多少天。

解决方案

用较晚的日期减去较早的日期。如果使用 Time 对象，结果将会是浮点数的秒数，因此要除以每天总共的秒数：

```
def last_modified(file)
    t1 = File.stat(file).ctime
    t2 = Time.now
    elapsed = (t2-t1)/(60*60*24)
    puts "#{file} was last modified #{elapsed} days ago."
end

last_modified("/etc/passwd")
# /etc/passwd was last modified 125.873605469919 days ago.
last_modified("/home/leonardr/")
# /home/leonardr/ was last modified 0.113293513796296 days ago.
```

如果使用 DateTime 对象，那么结果将会是一个有理数。希望将它转换为整数或浮点数以便显示：

```
require 'date'

def advent_calendar(date=DateTime.now)
    christmas = DateTime.new(date.year, 12, 25)
    christmas = DateTime.new(date.year+1, 12, 25) if date > christmas
    difference = (christmas-date).to_i
    if difference == 0
      puts "Today is Christmas."
    else
    puts "Only #{difference} day#{"s" unless difference==1} until
    Christmas."
    end
end

advent_calendar(DateTime.new(2006, 12, 24))
# Only 1 day until Christmas.
advent_calendar(DateTime.new(2006, 12, 25))
```

```
# Today is Christmas.
advent_calendar(DateTime.new(2006, 12, 26))
# Only 364 days until Christmas.
```

讨论

由于时间在内部被存储为数字，因此用一个时间减去另一个时间将会得到一个数字。由于两个数字衡量的是相同的事物（从某个时间零点开始的时间流逝），因此该数字实际表示着某些事物：时间轴上两个时间点相隔的秒数或天数。

当然，这对其他时间间隔也起作用。为了显示按小时计算的差，对于 Time 对象而言，要用该差除以每小时的秒数（3,600，如果使用的是 Rails，则除以 1.hour），对于 DateTime 对象而言，要用该差除以每小时代表的天数（也即，用该差乘以 24）：

```
sent = DateTime.new(2006, 10, 4, 3, 15)
received = DateTime.new(2006, 10, 5, 16, 33)
elapsed = (received-sent) * 24
puts "You responded to my email #{elapsed.to_f} hours after I sent it."
# You responded to my email 37.3 hours after I sent it.
```

甚至可以在某个时间间隔上使用 divmod 将它分割为更小的时间片。我从前在学校时编写过一个脚本，显示到我应当进行学习的期末所剩余的时间。该方法给出到某个预定事件为止的天、小时、分钟和秒的倒数计秒：

```
require 'date'

def remaining(date, event)
  intervals = [["day", 1], ["hour", 24], ["minute", 60], ["second", 60]]
  elapsed = DateTime.now - date
  tense = elapsed > 0 ? "since" : "until"

  interval = 1.0

  parts = intervals.collect do |name, new_interval|
    interval /= new_interval
    number, elapsed = elapsed.abs.divmod(interval)

    "#{number.to_i} #{name}#{"s" unless number == 1}"
  end
  puts "#{parts.join(", ")} #{tense} #{event}."
end

remaining(DateTime.new(2006, 4, 15, 0, 0, 0, DateTime.now.offset),
```

```
            "the book deadline")
# 27 days, 4 hours, 16 minutes, 9 seconds until the book deadline.
remaining(DateTime.new(1999, 4, 23, 8, 0, 0, DateTime.now.offset),

            "the Math 114A final")
# 2521 days, 11 hours, 43 minutes, 50 seconds since the Math 114A final.
```

参考

- 3.5 节“计算日期”

3.7 时区转换问题

要更改某个时间对象以便它能表示某个其他时区的相同时间。

解决方案

最常见的时区转换是将系统本地时间转换为 UTC 以及将 UTC 转换为本地时间。这些转换对于 Time 和 DateTime 对象都很简单。

Time#gmtime 方法就地修改 Time 对象，将其转换为 UTC。Time#gmtime 方法反方向进行转换：

```
now = Time.now # => Sat Mar 18 20:15:58 EST 2006now = now.gmtime # => Sun Mar 19 01:15:58 UTC 2006now = now.localtime # => Sat Mar 18 20:15:58 EST 2006
```

DateTime.new_offset 方法将一个 DateTime 对象从一个时区转换为另一时区，必须传递目的时区与 UTC 的偏移，如果将本地时间转换为 UTC 则传递零。由于 DateTime 对象是不可变的，因此该方法会创建一个除了时区偏移之外其余皆与老 DateTime 对象相同的新对象：

```
require 'date'

local = DateTime.now

local.to_s # => "2006-03-18T20:15:58-0500"


utc = local.new_offset(0)
utc.to_s # => "2006-03-19T01:15:58Z"
```

要将一个 UTC DateTime 对象转换为本地时间，需要调用 DateTime#new_offset 并传递关于本地时区的偏移。获得该偏移的最简单方法是在一个已知位于本地时间的 DateTime 对象上调用 offset，offset 通常是个分母为 24 的有理数：

```
local = DateTime.now
utc = local.new_offset

local.offset # => Rational(-5, 24)local_from_utc = utc.new_offset(local.offset)local_from_
utc.to_s # => "2006-03-18T20:15:58-0500"local == local_from_utc # => true
```

讨论

用 Time.at、Time.local、Time.mktime、Time.new 和 Time.now 创建的 Time 对象是使用当前系统时区创建的。用 Time.gm 和 Time.utc 创建的 Time 对象是使用 UTC 时区创建的。Time 对象可以表示任意时区，但是除了本地时间或 UTC 之外很难使用带 Time 的时区。

假设需要将本地时间转换为除 UTC 之外的其他时区。如果知道目的时区与 UTC 的偏移，那么可以将其表示为天的分数并传递给 DateTime#new_offset：

```
#Convert local (Eastern) time to Pacific time
eastern = DateTime.now
eastern.to_s # => "2006-03-18T20:15:58-0500"

pacific_offset = Rational(-7, 24)
pacific = eastern.new_offset(pacific_offset)
pacific.to_s # => "2006-03-18T18:15:58-0700"
```

DateTime#new_offset 能够在任意时区偏移间进行转换，因此，对于时区转换而言，最简单的方法是使用 DateTime 对象并在必要时将其转换回 Time 对象。但是，DateTime 对象只根据数字 UTC 偏移

来理解时区。当知道时区被称为“WET”、“Zulu”或“Asia/Taskent”时，如何才能将日期和时间转换为 UTC 呢？

在 Unix 系统上可以临时为当前进程修改“系统”时区。下面 Time 类的 C 库知道许多种时区（如果要查看可用的时区，那么“zoneinfo”数据库通常位于/usr/share/ zoneinfo/）。要利用这一技巧，需要将环境变量 TZ 设置为适当的值，强制 Time 类的行为令计算机仿佛位于某个其他时区。下面是一个使用该技巧将一个 Time 对象转换为下面的 C 库所支持的任意时区的方法：

```
class Time

  def convert_zone(to_zone)

    original_zone = ENV["TZ"]

    utc_time = dup.gmtime

    ENV["TZ"] = to_zone

    to_zone_time = utc_time.localtime

    ENV["TZ"] = original_zone

    return to_zone_time

  end
end
```

下面我们将一些本地（东部）时间转换为世界各地其他时区：

```
t = Time.at(1000000000) # => Sat Sep 08 21:46:40 EDT 2001

t.convert_zone("US/Pacific") # => Sat Sep 08 18:46:40 PDT 2001

t.convert_zone("US/Alaska") # => Sat Sep 08 17:46:40 AKDT 2001t.convert_zone("UTC") # => Sun Sep 09 01:46:40 UTC 2001t.convert_zone("Turkey") # => Sun Sep 09 04:46:40 EEST 2001
```

请注意某些时区（如印度）只比相邻时区偏移半个小时：

```
t.convert_zone("Asia/Calcutta") # => Sun Sep 09 07:16:40 IST 2001
```

通过在创建 Time 对象前设置 TZ 环境变量，就可以表示任意时区的时间。下列代码将拉各斯时间转换为新加坡时间，而忽略“真正”的时区。

```
ENV["TZ"] = "Africa/Lagos"
t = Time.at(1000000000) # => Sun Sep 09 02:46:40 WAT 2001
ENV["TZ"] = nil

t.convert_zone("Singapore") # => Sun Sep 09 09:46:40 SGT 2001

# Just to prove it's the same time as before:
t.convert_zone("US/Eastern") # => Sat Sep 08 21:46:40 EDT 2001
```

由于 TZ 环境变量对进程而言是全局的，因此，当有多个线程试图同时转换时区时会出问题。

参考

- 3.9 节“Time 与 DateTime 对象之间的转换”
- 3.8 节“检查夏令时是否起效”

3.8 检查夏令时是否起效

问题

要查看当前位置的当前时间是正常时间还是夏令时。

解决方案

创建一个 Time 对象并检查其 isdst 方法：

```
Time.local(2006, 1, 1) # => Sun Jan 01 00:00:00 EST 2006Time.local(2006, 1, 1).isdst # => falseTime.local(2006, 10, 1) # => Sun Oct 01 00:00:00 EDT 2006Time.local(2006, 10, 1).isdst # => true
```

讨论

表示 UTC 时间的 Time 对象在调用 isdst 时总是返回 false，因为 UTC 是全年不变的。其他 Time 对象会参考关于用于创建 Time 对象的时间位置的夏令时规则。这通常用来创建它的计算机所在的系统位置：有关修改它的信息请参见 3.7 节。下列代码列举了关于美国夏令时的某些规则：

```
eastern = Time.local(2006, 10, 1) # => Sun Oct 01 00:00:00 EDT 2006eastern.isdst # => true

ENV['TZ'] = 'US/Pacific'pacific = Time.local(2006, 10, 1) # => Sun Oct 01 00:00:00 PDT 2006pacific.isdst # => true

# Except for the Navajo Nation, Arizona doesn't use Daylight Saving Time.ENV['TZ'] = 'America/Phoenix'arizona = Time.local(2006, 10, 1) # => Sun Oct 01 00:00:00 MST 2006arizona.isdst # => false

# Finally, restore the original time zone.
ENV['TZ'] = nil
```

Ruby 的 Time 类所基于的 C 库处理历史上特定时区或地区用于夏令时的复杂规则。例如，全美于 1918 年批准使用夏令时，但大多数地区很快就废止了。C 库所使用的“zoneinfo”文件包含了这一信息以及其他许多规则：

```
# Daylight saving first took effect on March 31, 1918.Time.local(1918, 3, 31).isdst # => falseTime.local(1918, 4, 1).isdst # => trueTime.local(1919, 4, 1).isdst # => true

# The federal law was repealed later in 1919, but some places

# continued to use Daylight Saving Time.
ENV['TZ'] = 'US/Pacific'
Time.local(1920, 4, 1) # => Thu Apr 01 00:00:00 PST 1920

ENV['TZ'] = nil
Time.local(1920, 4, 1) # => Thu Apr 01 00:00:00 EDT 1920
```

```
# Daylight Saving Time was reintroduced during the Second World War.Time.local (1942,2,9) # => Mon Feb 09 00:00:00 EST 1942Time.local(1942,2,10) # => Tue Feb 10 00:00:00 EWT 1942# EWT stands for "Eastern War Time"
```

于 2005 年通过的美国法律将夏令时扩展至 3 月和 11 月，从 2007 年开始执行。在 2007 年及其后创建的 Time 对象会根据用户 zoneinfo 文件的时间长短来选择反映或不反映新法律。

```
Time.local(2007, 3, 13) # => Tue Mar 13 00:00:00 EDT 2007
# Your computer may incorrectly claim this time is EST.
```

这是普通的观点。没有比通过法律条文更令人们选举出的官员们热衷的了，因此，不应当依赖于 is dst 来保证任何表示未来一年以上的 Time 对象是精确的。当时间真正到来时，你所在地方的夏令时可能会遵循不同的规则。

Date 类不是基于 C 库，对时区或地区一无所知，因而对夏令时也一无所知。

参考

- 3.7 节“时区转换”

3.9 Time 与 DateTime 对象之间的转换问题

同时使用由 Ruby 的两种标准日期/ 时间库所创建的 DateTime 和 Time 对象，因此在进行比较、迭代或日期计算时不能混淆这些对象，因为它们是不兼容的。要将所有对象转换为这种或那种形式以便于以相同方式同等对待。

解决方案

要将一个 Time 对象转换为 DateTime，需要使用类似下列的代码：

```
require 'date'
class Time
 def to_datetime

    # Convert seconds + microseconds into a fractional number of seconds
    seconds = sec + Rational(usec, 10**6)

    # Convert a UTC offset measured in minutes to one measured in a
    # fraction of a day.
    offset = Rational(utc_offset, 60 * 60 * 24)
    DateTime.new(year, month, day, hour, min, seconds, offset)

  end
end

time = Time.gm(2000, 6, 4, 10, 30, 22, 4010)
# => Sun Jun 04 10:30:22 UTC 2000
time.to_datetime.to_s
# => "2000-06-04T10:30:22Z"
```

将一个 DateTime 对象转换为 Time 是相似的，只需要确定 Time 对象是使用本地时间还是 GMT。下列代码为 DateTime 的超类 Date 添加了转换方法，因此它能够同时在 Date 和 DateTime 对象上运行。

```
class Date
  def to_gm_time
  to_time(new_offset, :g
 m)
  end

 def to_local_time
   to_time(new_offset(DateTime.now.offset-offse
  t), :local)
   end

  private

  def to_time(dest, method)
    #Convert a fraction of a day to a number of microseconds
    usec = (dest.sec_fraction * 60 * 60 * 24 * (10**6)).to_i
    Time.send(method, dest.year, dest.month, dest.day, dest.hour,
```

```
dest.min,
                des
                t.se
                c, u
                sec)
                 en
                d
                end
```

```
(datetime = DateTime.new(1990, 10, 1, 22, 16, Rational(41,2))).to_s
# => "1990-10-01T22:16:20Z"
datetime.to_gm_time
# => Mon Oct 01 22:16:20 UTC 1990
datetime.to_local_time
# => Mon Oct 01 17:16:20 EDT 1990
```

讨论

Ruby 表示日期和时间的两种方法无法良好共存。但是，由于任何一种都无法完全替代另一种，因为在使用 Ruby 时可能会同时使用这两种。转换方法通过简单地将一种类型转换为另一种类型从而避免了不兼容性：

```
time < datetime
# ArgumentError: comparison of Time with DateTime failed
time.to_datetime < datetime
# => false
time < datetime.to_gm_time
# => false

time - datetime# TypeError: can't convert DateTime into Float(time.to_datetime - dateti
me).to_f# => 3533.50973962975 # Measured in daystime - datetime.to_gm_time# =>
305295241.50401 # Measured in seconds
```

上述定义的方法是可逆的：可以在 Date 和 DateTime 对象之间来回转换而不会损失精确度。

```
time # => Sun Jun 04 10:30:22 UTC2000time.usec # => 4010

time.to_datetime.to_gm_time # => Sun Jun 04 10:30:22 UTC2000time.to_datetime.
to_gm_time.usec # => 4010
```

```
datetime.to_s # => "1990-10-01T22:16:20Z"datetime.to_gm_time.to_datetime.to_s
# => "1990-10-01T22:16:20Z"
```

只要能够在 Time 和 DateTime 对象之间进行转换，那么可以简单地编写规格化一个混合数组的代码，使得数组的所有元素在结束时都具有相同类型。该方法试图将一个混合数组转换为仅包含 Time 对象的数组。如果它遇到一个日期不符合 Time 类的约束，它会停止并将数组转换为 DateTime 对象的数组（因此会丢失关于夏令时的信息）：

```
def normalize_time_types(array)
  # Don't do anything if all the objects are already of the same type.
  first_class = array[0].class
  first_class = first_class.super if first_class == DateTime
  return unless array.detect { |x| !x.is_a?(first_class) }

  normalized = array.collec
    t do |t|
    if t.is_a?(Date)
    begin
    t.to_local_time
    rescue ArgumentError
    # Time out of range;
    convert to DateTimes
instead.
          convert_to = DateTime
          break

    end
  else
  t
  end
  end

  unless normalized

    normalized = array.collect { |t| t.is_a?(Time) ? t.to_datetime : t }
  end
  return normalized

end
```

如果一个混合数组中的所有对象都可以表示为 Time 或 DateTime 对象，那么该方法会将它们都转换为 Time 对象：

```
mixed_array = [Time.now, DateTime.now]
# => [Sat Mar 18 22:17:10 EST 2006,
# #<DateTime: 23556610914534571/9600000000,-5/24,2299161>]
normalize_time_types(mixed_array)
# => [Sat Mar 18 22:17:10 EST 2006, Sun Mar 19 03:17:10 EST 2006]
```

如果某个 DateTime 对象无法表示为 Time，那么 normalize_time_types 会将所有对象转换为 DateTime 实例。下面代码运行在带 32 位时间计数器的系统上：

```
mixed_array << DateTime.civil(1776, 7, 4)
normalize_time_types(mixed_array).collect { |x| x.to_s }
```

```
# => ["2006-03-18T22:17:10-0500", "2006-03-18T22:17:10-0500",
# => "1776-07-04T00:00:00Z"]
```

参考

- 3.1 节“查找当前日期”

3.10 查找周日期问题

要查找特定日期的周日期。

解决方案

使用 wday 方法（Time 与 DateTime 皆支持）查找周日期为 0 至 6 之间的某个数字。星期日是 0 日。下列代码产生的代码块是两个日期之间所有星期日的日期，它使用 wday 查找从起始日期向后的第一个星期日（要记住起始日期本身可能是个星期日），然后每次增加 7 天获得后续的星期日：

```
def every_sunday(d1, d2)
  # You can use 1.day instead of 60*60*24 if you're using Rails.
  one_day = d1.is_a?(Time) ? 60*60*24 : 1

  sunday = d1 + ((7-d1.wday) % 7) * one_day

  while sunday < d2
    yield sunday
    sunday += one_day * 7

  end
end
```

```
def print_every_sunday(d1, d2)
  every_sunday(d1, d2) { |sunday| puts sunday.strftim
  e("%x")}
  end
```

```
print_every_sunday(Time.local(2006, 1, 1), Time.local(2006, 2, 4))
# 01/01/06
# 01/08/06
# 01/15/06
# 01/22/06
# 01/29/06
```

讨论

时间最常用的部分是其日历和时钟读数：年、日、小时，等等。Time 和 DateTime 可以访问这些，但它们还可以访问时间的一些其他方面：年的儒略历日期（yday）以及更有用的周日期（wday）。every_Sunday 方法可接收两个 Time 对象或两个 DateTime 对象。唯一的不同在于需要增加对象的数字是按天计算的。如果只准备使用一类对象，那么可以稍稍简化代码。

要获得英语字符串表示的周日期，可使用 strftime 指令%A 和%a：

```
t = Time.local(2006, 1, 1)t.strftime("%A %A %A!") # => "Sunday Sunday Sunday!"
t.strftime("%a %a %a!") # => "Sun Sun Sun!"
```

可以查找周日期或年日期，但 Ruby 没有用于查找年周数的内置方法（有一个查找年的商用周的方法，请参见 3.11 节）。如果需要这种方法，使用年日期和周日期与创建一个也不是难事。下面代码在某个模块中定义了 week 方法，它可以混合使用 Date 和 Time：

```
require 'date'
module Week
 def week
 (yday + 7 - wday)
 / 7
 end
end

class Date
   include Week
  end

class Time


   include Week
end

saturday = DateTime.new(2005, 1, 1)saturday.week # => 0(saturday+1).week # => 1
#Sunday, January 2(saturday-1).week # => 52 #Friday, December31
```

参考

- 3.3 节“打印日期”
- 3.5 节“计算日期”
- 3.11 节“处理商用日期”

3.11 处理商用日期问题

在编写商业或金融应用程序时，需要处理商用日期而不是民用日期或历法日期。

解决方案

DateTime 提供了某些方法可处理商用日期。Date#cwday 给出商用周日期，Date#cweek 给出商用年周数，而 Date#cwyear 给出商业财政年。

考虑 2006 年 1 月 1 日。这是日历 2006 年的第一天，但由于它是个星期日，因为它是商用 2005 年最后一天：

```
require 'date'sunday = DateTime.new(2006, 1, 1)sunday.year # => 2006sunday.cwyear # => 2005sunday.cweek # => 52sunday.wday # => 0sunday.cwday # => 7
```

商用 2006 年开始于 2006 年的第一个平日：

```
monday = sunday + 1monday.cwyear # => 2006monday.cweek # => 1
```

讨论

除非要编写需要使用商用日期的应用程序，否则用户可能不会关心这些，但它是一类非常有趣的事情（如果认为日期是有趣的话）。商用周开始于星期一，而不是星期日，因为星期日是周末的一部分。DateTime#cwday 就像 DateTime#wday 一样，只是它赋予星期日的值是 7 而不是 0。

这说明 DateTime#cwday 的范围是从 1～7，而不是从 0～6：

```
(sunday...sunday+7).each do |d|

  puts "#{d.strftime("%a")} #{d.wday} #{d.cwday}"

end

# Sun 0 7

# Mon 1 1

# Tue 2 2

# Wed 3 3

# Thu 4 4

# Fri 5 5

# Sat 6 6
```

cweek 和 cwyear 方法必须作用于以每年的第一个星期一为起始日的商用年。所有在第一个星期一之前的日子都被认为是前一个商用年的一部分。上述解决方案中的示例就说明了这一点：2006 年 1 月 1 日是个星期日，因此按商用计算它属于 2005 年的最后一周。

参考

- 有关使用 strftime 指令打印商用日期各部分的信息请参见 3.3 节“打印日期”

3.12 周期性运行代码块问题

要按某种时间间隔重复地运行某些 Ruby 代码（例如一次对 shell 命令的调用）。

解决方案

创建一个运行代码块的方法，然后睡眠直至该代码块再次运行的时间为止：

```
def every_n_seconds(n)
  loop do
    before = Time.now
    yield
    interval = n-(Time.now-before)
    sleep(interval) if interval > 0
  end
end

every_n_seconds(5) do
  puts "At the beep, the time will be #{Time.now.strftime("%X")}... beep!"
end
# At the beep, the time will be 12:21:28... beep!
# At the beep, the time will be 12:21:33... beep!
# At the beep, the time will be 12:21:38... beep!
# ...
```

讨论

希望周期性地运行某些代码的时期主要有两个。第一个是实际希望某些事情按特定时间间隔发生时：例如每 10 秒将自己的状态附加至日志文件。另一个时期是希望某些事情持续发生、但把它置于死循环会令系统性能变差时。在这种情况下，通过在循环中放置某些空闲时间使得自己的代码不会一直运行来进行折衷。

every_n_seconds 的实现在睡眠时间中扣除花费在运行代码块上的时间，这样能够确保对该代码块的调用具有均匀地间隔时间，尽可能接近于期望的时间间隔。如果告诉 every_n_seconds 每 5 秒调用一次代码块，但代码块运行要花费 4 秒，那么 every_n_seconds 只睡眠 1 秒。如果代码块运行要花费 6 秒，那么 every_n_seconds 就根本不会睡眠：它从对代码块的调用返回，然后立即再次产生代码块。

如果希望始终睡眠某个时间间隔，而不管代码块的运行时间开销，那么可以简化代码：

```
def every_n_seconds(n)
```

```
    loop do
      yield
      sleep(n)
    end
  end
```

在大多数情况下，不希望 every_n_seconds 接管程序的主循环。下面是 every_n_seconds 的一种版本，它产生一个单独的线程来运行用户任务。如果用户代码块使用 break 关键字停止循环，那么线程会停止运行：

```
  def every_n_seconds(n)
    thread = Thread.new do
      while true
        before = Time.now
        yield
        interval = n-(Time.now-before)
        sleep(interval) if interval > 0
      end
    end
    return thread
  end
```

下列代码段使用 every_n_seconds 监视一个文件，等待用户修改它：

```
  def monitor_changes(file, resolution=1)
    last_change = Time.now
    every_n_seconds(resolution) do
      check = File.stat(file).ctime
```

```
      if check > last_change
        yield file
        last_change = check

      elsif Time.now - last_change > 60
        puts "Nothing's happened for a minute, I'm bored."
        break
      end
    end
  end
end
```

如果系统上的某用户在文件/tmp/foo 上工作，那么该示例会给出类似下列的输出：

```
thread = monitor_changes("/tmp/foo") { |file| puts "Someone changed
#{file}!" }
# "Someone changed /tmp/foo!"
# "Someone changed /tmp/foo!"
# "Nothing's happened for a minute; I'm bored."
thread.status # => false
```

参考

- 3.13 节“等待固定长度的时间”
- 23.4 节“不用 cron 或 at 运行周期性任务”

3.13 等待固定长度的时间问题

要暂停程序或其某个单独的线程一段固定长度的时间。

解决方案

Kernel#sleep 方法接收一个浮点数并令当前线程睡眠该数量（可能是小数）的秒数：

```
3.downto(1) { |i| puts "#{i}..."; sleep(1) }; puts "Go!"
# 3...
# 2...
# 1...
# Go!

Time.new # => Sat Mar 18 21:17:58 EST 2006

sleep(10)Time.new # => Sat Mar 18 21:18:08 EST 2006sleep(1)Time.new # => Sat Mar 18 21:18:09 EST 2006
# Sleep for less then a second.Time.new.usec # => 377185sleep(0.1)Time.new.usec # => 479230
```

讨论

计时器通常用于程序需要与比计算机 CPU 运行速度慢的资源相结合时：网络管道或人的手眼。Ruby 程序可以在每次轮询之间睡眠几秒钟，将运行机会留给 CPU 上的其他程序，而不是一直轮询查找新数据。从人的标准来看并没有多少时间，但几秒钟的睡眠可以大大提高系统的总体性能。

可以传递任意浮点数用于sleep，但这夸大了可以控制线程睡眠时间的精细程序。例如，睡眠时间不能在 10^{-50} 秒内，因为它在物理上是不可能的（它小于普朗克时间）；睡眠时间也不能为Float::EPSILON秒，因为这几乎小于计算机计时器的时间分辨力。

甚至可能无法可靠地 sleep 一微秒，尽管大多数现代计算机时钟具有微秒级精确度。在 sleep 命令被 Ruby 解释器执行并且线程实际开始等待其计时器停止期间，少量的时间就已经流逝掉了。对于每个小时间间隔，这一时间可能会大于早先要求 Ruby 睡眠的时间。

下面是一个简单的基准测试，给出系统实际能够令一个线程 sleep 的时间。它以 1 秒的 sleep 时间间隔开始，这已经相当精确了，然后依次逐渐减少时间间隔，精确度也随之减小：

```
interval = 1.0

10.times do |x|

  t1 = Time.new

  sleep(interval)

  actual = Time.new - t1

  difference = (actual-interval).abs
```

```
    percent_difference = difference / interval * 100

    printf("Expected: %.9f Actual: %.6f Difference: %.6f (%.2f%%)\n",
                   interval, actual, difference, percent_difference)
    interval /= 10
end
# Expected: 1.000000000 Actual: 0.999420 Difference: 0.000580 (0.06%)
# Expected: 0.100000000 Actual: 0.099824 Difference: 0.000176 (0.18%)
# Expected: 0.010000000 Actual: 0.009912 Difference: 0.000088 (0.88%)
# Expected: 0.001000000 Actual: 0.001026 Difference: 0.000026 (2.60%)
# Expected: 0.000100000 Actual: 0.000913 Difference: 0.000813 (813.00%)
# Expected: 0.000010000 Actual: 0.000971 Difference: 0.000961 (9610.00%)

# Expected: 0.000001000 Actual: 0.000975 Difference: 0.000974 (97400.00%)
# Expected: 0.000000100 Actual: 0.000015 Difference: 0.000015 (14900.00%)
# Expected: 0.000000010 Actual: 0.000024 Difference: 0.000024 (239900.00%)
# Expected: 0.000000001 Actual: 0.000016 Difference: 0.000016
(1599900.00%)
```

少量报告的时间来源于创建第二个 Time 对象的开销，但不足以影响到这些结果。在我的系统上，如果我告诉 Ruby 睡眠 1 微秒，那么运行 sleep 调用所花费的时间远远大于早先希望的 sleep 时间！根据这一基准测试，我能够要求 sleep 的精确睡眠时间最少为一秒的 1/100。

读者可能会认为通过将 CPU 置于带有一定数量循环的死循环中会得到好一些的睡眠时间分辨力。除了显而易见的问题（这会损害系统性能，并且由于计算机总是越来越快的，因此相同的循环运行也会越来越快）之外，这样做甚至是不可靠的。

操作系统并不知道用户试图运行一个定时的循环：它只看到用户使用 CPU，并且可以在任意时刻打断循环任意长的时间让其他进程使用 CPU。除非使用可以精确控制 CPU 运转的嵌入式操作系统，否则唯一等待特定时间间隔的可靠方法是使用 sleep。

迟早唤醒

sleep 方法在调用它的线程调用其 run 方法时会提早结束。如果希望线程睡眠至另一个线程唤醒它为止，那么可以使用 Thread.stop：

```
alarm = Thread.new(self) { sleep(5); Thread.main.wakeup }
puts "Going to sleep for 1000 seconds at #{Time.new}..."
sleep(10000); puts "Woke up at #{Time.new}!"
# Going to sleep for 1000 seconds at Thu Oct 27 14:45:14 PDT 2005...
# Woke up at Thu Oct 27 14:45:19 PDT 2005!

alarm = Thread.new(self) { sleep(5); Thread.main.wakeup }
puts "Goodbye, cruel world!";
Thread.stop;
puts "I'm back; how'd that happen?"
```

```
# Goodbye, cruel world!
# I'm back; how'd that happen?
```

参考

- 3.12 节“周期性运行代码块”
- 第 20 章
- 21.11 节“使得键盘指示灯闪烁”中的摩尔斯电码示例给出睡眠的有趣用途

3.14 为长期运行的操作添加超时问题

给定正在运行某个可能要花很长时间才能完成、或者可能永远都无法完成的代码，要在它执行时间过长时中断代码运行。

解决方案

使用内置 timeout 库。Timeout.timeout 方法要求一个代码块和一个截止期限（以秒计算）。如果代码块及时结束运行，那么它返回 true。如果截止期限到达而代码块仍在运行，那么 Timeout.timeout 中止代码块并引发一个异常。

如果没有对 timeout 的调用，下列代码可能永远不会结束运行。但是，timeout 在 5 秒钟之后会引发 Timeout::Error 并终止执行：

```
# This code will sleep forever... OR WILL IT?

require 'timeout'

before = Time.now

begin


  status = Timeout.timeout(5) { sleep }

rescue Timeout::Error


  puts "I only slept for #{Time.now-before} seconds."
end
# I only slept for 5.035492 seconds.
```

讨论

有时必须建立一个网络连接，或者执行其他某些可能会非常慢、甚至可能根本无法完成的动作。使用超时可以为操作执行强加一个上限。如果它失败，那么可以随后再试或者在获得试图得到的信息时逐步迈进。即使无法得到恢复，也可以报告错误并优雅地退出程序，而不是傻坐着等待操作完成。

默认情况下，Timeout.timeout 引发 Timeout::Error。可以传递一个自定义的异常类作为 Timeout.timeout 的第二参数：这样可以避免必须解除 Timeout:Error，让用户可以引发其他应用程序知道如何进行处理的错误。

如果代码块有副作用，那么在 timout 终止代码块之后它们仍是可见的：

```
def count_for_five_seconds

  $counter = 0

  begin


    Timeout::timeout(5) { loop { $counter += 1 } }

  rescue Timeout::Error
```

```
  puts "I can count to #{$counter} in 5 seconds."
 end
end
```

```
count_for_five_seconds
# I can count to 2532825 in 5 seconds.
$counter # => 2532825
```

这说明用户数据集目前处于不一致状态。

参考

- ri Timeout
    - 3.13 节“等待固定长度的时间”
    - 14.1 节“抓取 Web 页面的内容”

15. 1 编写简单的Rails应用程序显示系统状态

Ruby on Rails 是无可非议的 Ruby 杀手锏应用程序。它提供很多保障将 Ruby 从其日文本土的隐晦中解脱出来。没有其他任何一门编程语言可以以这样一个简单的 Web 应用程序框架而自豪，该框架也吸取了大部分该语言开发者的思想（注 1）。本章论证重要的基础 Rails 使用原理（类似 15.6 节中），给出普通 Web 应用程序模式的 Rails 实现（15.4 节和 15.8 节）并且讨论如何在 Rails 内部使用标准的 Ruby 工具（15.22 节和 15.23 节）。

不谈其质量和流行程度，Rails 并没有为 Web 开发带来其他新鲜的东西。其基础在于标准的编程模式，比如 ActiveRecord 以及 Model-View-Controller。它重用很多已经存在的 Ruby 库（比如 Rake 和 ERb）。Rails 的功能在于集成这些标准的技术自动化完成任务，并且会断言合理的默认行为。

如果 Rails 有秘密的话，那就在于命名惯例。最为广泛使用的 Web 应用程序是 CRUD 应用程序：从数据库创建、阅读、更新并且删除信息。在这些类型的应用程序中，Rails 的功劳很是夺目。可能一开始只有一个数据库计划而没有任何代码，但是 Rails 可以将很多段代码通过命名惯例和快捷方式联系起来。这些使得可以十分快速地创建应用程序。

因为这么多的设定和命名可以显式地从其他信息片断里继承，Rails 比其他框架需要更少的“文书工作”。数据隐式包含在代码里，或者数据库方案并不需要在其他地方指定。本系统的必需部分是针对复数名词的 ActiveSupport 系统（15.7 节）。

在命名惯例不起作用的地方，Rails 使用 decorator 方法来声明对象间的关系，而不是在膨胀的 XML 配置文件里。结果是更小，更易于理解，并且更灵活的应用程序。

正如上面所说，Rails 架构在普通 Ruby 库之上，这些库在本书的很多地方都有所涉及，它们包括 ActiveRecord（13 章的大部分，尤其在 13.11 节中）、ActionMailer（14.5 节）、ERb（1.3 节）、Rake（19 章）和 Test::Unit（17.7 节）。其中某些出现在 Rails 之前，某些是为 Rails 编写却在外部无法使用的。反过来也是正确的：因为 Rails 应用程序可以被用来实现很多目的，几乎本章的每一节在 Rails 程序中都很有用。

Rails 可用作 rails gem，它包含库以及 rails 的命令行程序。这就是运行来创建 Rails 应用程序的程序。当调用该程序（比如，用 rails mywebapp）时，Rails 为 Web 应用程序产生一个目录结构，通过 WEBrick 测试服务器以及单元测试框架完成。当使用 script/generate 脚本来跳过应用程序的创建时，Rails 会用更多文件组装该目录结构。这些脚本生成的代码很小，并且等同于启动项目时大部分 IDE 生成的代码。

Rails 的架构是流行的 Model-View-Controller 架构。这就将 Web 应用程序分割成 3 个可预先命名的部分。本章会详细讲述，但是先有一个引导性的介绍。

模型（model ）是应用程序使用的数据形式的代表。通常是一组 Ruby 类，ActiveRecord::Base 的子类，每个都对应于应用程序数据库的一个表。本章第一个重要的模型将在 15.6 节重点介绍。要生成某个特定数据库表的模型，通过表名调用 script/generate 模型，比如：

```
$ script/generate model users
```

这会创建一个名为 app/models/users.rb 的文件，它定义了 User ActiveRecord 类以及单元测试该模块的基本结构。并不会创建实际的数据库表。

控制器（controller）是 Ruby 的一个类（ActionController::Base 的子类），它的方法在模块上定义了操作。每个操作当作 controller 的方法被定义。

要生成一个控制器，用控制器的名称以及要做的操作来调用 script/generate controller：

```
$ script/generate controller user add delete login logout
```

该命令创建文件 app/controllers/user_controller.rb，它定义了一个类 UserController。该类定义了 4 个 stub 方法：add、delete、login 以及 logout，每个对应于终端用户可以在重要的 User 模型上执行的操作。它也创建了功能单元测试控制器的模板。

控制器在本章的首节讨论（15.1 节）。视图（view）是应用程序的用户界面。它包含在一组 ERb 模板中，保存在.rhtml 文件里。更重要的是，对于每个控制器的操作通常都会有一个.rhtml 文件：这是针对该特殊操作的 Web 界面。上述创建了 UserController 类的同一条命令也创建了 app/views/user/ 下的 4 个文件：add.rhtml、delete.rhtml、login.rhtml 以及 logout.rhtml。这是由于 UserController 类，这些被当作 stub 文件启动。我们的工作是自定义它们来表示应用程序的接口。

就像控制器，视图在本章首节所展现：15.1 节，而 15.3 节、15.5 节以及 15.14 节讲述如何自定义属于自己的视图。

这样的分节不是随意的。如果限制改变数据库为模型的代码，那么就将容易进行单元测试该代码并且审查其安全问题。通过移动所有处理代码到控制器中，可以将用户界面的显示与其内部的工作分离开。这样做的最为明显的好处在于可以拥有一个 UI 设计器来更改视图模板，而无需占用很多 Ruby 代码。

要想学习 Model-View-Controller 如何工作的话，参考 15.2 节，该节讲述控制器和视图之间的关系，15.16 节则将三者都集成起来。

如下有更多可以参考的资源来帮助我们快速地学习 Rails：

•本书的姐妹篇，Rails Cookbook by Rob Orsini (O'Reilly)，更详细地讨论了 Rails 问题，正如 Chad Fowler（注重实效的编程者）在 Rails 节所述

•Dave Thomas、David Hansson、Leon Breedt、Mike Clark、Thomas Fuchs 以及 Andrea Schwarz（注重实效的编程者）所著 Agile Web Development with Rails 是 Rails 程序员的标准参考书

•Rails Web 站点 http://www.rubyonrails.com/ 上的 Ruby，尤其是 RDoc 文档（http:// api.rubyonrails.org/）以及 wiki（http://wiki.rubyonrails.com/）

15.1 编写简单的 Rails 应用程序显示系统状态问题

想通过创建很简单的应用程序来开始 Rails 的学习。

解决方案

该例显示了 Unix 系统上运行着的进程。如果开发在 Windows 上进行，可以用其他命令代替（比如 dir 的输出）或者只是使得应用程序输出静态消息。首先，确保安装了 rails gem。

要创建 Rails 应用程序，运行 rails 命令并且传递进入应用程序的名称。应用程序名为“status”。

```
$ rails status

                                create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
...
```

Rails 应用程序至少需要两部分：一个控制器和一个视图。控制器能够得到系统信息，视图则可以显示它。
能够产生控制器以及用 generate 脚本生成相应的视图。如下调用定义了控制器和视图，它们实现名为 index 的单一操作。这就是应用程序的主视图（也是唯一的）。

```
$ cd status
```

```
$ ./script/generate controller status index exists app/controllers/ exists app/helpers/ create
      app/views/status exists test/functional/ create app/controllers/status_controller.rb cre
     ate test/functional/status_controller_test.rb create app/helpers/status_helper.rb create
      app/views/status/index.rhtml
```

生成的控制器在 Ruby 源文件 app/controllers/status_controller.rb 里。该文件定义了 StatusControlle r 类，实现 index 操作，就像名为 index 的空白方法一样。填满 index 方法，从而可以显示想要在视图里使用的对象：

```
class StatusController < ApplicationController
```

```
  def index
    # This variable won't be accessible to the view, since it is local
    # to this method
    time = Time.now
```

```
  # These variables will be accessible in the view, since they are

  # instance variables of the StatusController.

  @time = time

  @ps = `ps aux`


    end

end
```

生成的视图在 pp/views/status/index.rhtml 里。它由一段静态 HTML 开始，改变它为 ERb 模板，使用 StatusController#index 设置的实例变量：

```
<h1>Processes running at <%= @time %></h1>
<pre><%= @ps %></pre>
```

现在应用程序完整了。要想运行它，用如下命令启动 Rails 服务器：

```
$ ./script/server

=> Booting WEBrick...

=> Rails application started on http://0.0.0.0:3000
```

```
=> Ctrl-C to shutdown server; call with --help for options
...
```

可以通过访问 http://localhost:3000/status/ 查看该应用程序。当然，不需要将该应用程序发布到外部世界，因为这样可能会将系统信息提供给攻击者。

讨论

首先，需要注意的与 Rails 应用程序有关的事情是不要为每个 URL 创建单独的代码。Rails 使用一种架构，在该架构中，控制器（Ruby 源文件）以及视图（.rhtml 文件里的 ERb 模板）组合起来服务于一系列操作。每个操作处理站点上的部分 URL。

考虑类似 http://www.example.com/hello/world 的 URL。为了在 Rails 应用程序里服务该 URL，需要创建 hello 控制器并且给它一个称为 world 的操作。

```
$ ./script/generate controller hello world
```

控制器类需要有一个 world 方法，views/hello 目录则需要一个包含视图的 world.rhtml 文件。

```
class HelloController < ApplicationController

  def world

  end

end
```

访问 http://www.example.com/hello/world 会调用 HelloController#world 方法，翻译 world.rhtml 模板，得到一些 HTML 输出，并且将这些输出送至客户端。

控制器的默认操作是 index，就好比静态 Web 服务器目录里的默认页面是 index.html。因此访问 http://www.example.com/hello/和访问 http://www.example.com/hello/index/会得到同样的效果。

如上所述，视图文件仅仅是 Rails 服务的最终页面的主要部分。它不是完整的 HTML 页面，不可以在其中放入<html>或者<body>标签（参考 15.3 节）。因为视图文件是 ERB 模板，也就不可以在视图里调用 puts 或者 print。ERB 的介绍见 1.3 节，但是很值得在 Rails 应用程序上下文里进行扫描。

要想插入 Ruby 表达式的值到 ERB 模板里，使用<%= %>指示。如下是 hello 操作的可能存在的 world.rhtml 视图：

```
<p>Several increasingly silly ways of displaying "Hello world!":</p>

<p><%= "Hello world!" %></p>

<p><%= "Hello" + "world!" %></p>

<p><%= w = "world"
```

```
"Hello #{w}!" %></p>
```

```
<p><%= 'H' + ?e.chr + ('l' * 2) %><%=('o word!').gsub('d', 'ld')%></p>
```

最后这个例子是额外的，但是它证明了一点。没有必要在视图模板里放入这么多的 Ruby 代码（可能必须进入控制器，或者需要以类似 PHP 的代码结束），但是如果需要这么做的话也是可以实现的。在 ERb 指示里等号意味着将会打印出输出。如果想要执行某个没有输出的命令，忽略等号并且使用<% %>指示。

```
<% hello = "Hello" %>
<% world = "world!" %>
<%= hello %> <%= world %>
```

视图和控制器只是基于一些从 Ruby 代码里得到的数据（比如当前时间以及 ps aux 的输出）。但是大多数真实世界的视图和控制器是基于模型的：一组数据库表，包含被视图显示并且被控制器操作的数据。这就是著名的“Model-View-Controller”架构，并且这无疑是 Rails 独一无二的。

参考

- 1.3 节“将变量代入现有的字符串”有更多 ERB 相关信息
- 15.3 节“创建页眉和页脚的布局”

15.2 从控制器传递数据到视图问题

想要在控制器和视图之间传递数据。

解决方案

视图是 ERB 模板，该模板在其控制器对象的上下文中得到解释。视图不会调用任何控制器的方法，但是它可以访问控制器的实例变量。传递数据到视图，设置控制器的一个实例变量。

这是一个 NovelController 类，被放至 app/controllers/novel_controller.rb。可以通过运行 script/generate controller novel index 为其生成 stub。

```
class NovelController < ApplicationController

  def index
    @title = 'Shattered View: A Novel on Rails'
    one_plus_one = 1 + 1
    increment_counter one_plus_one
  end

  def helper_method
    @help_message = "I see you've come to
    me for help."
    end

  private

  def increment_counter(by)
    @counter ||= 0
    @counter += by

  end
end
```

因为这是 Novel 控制器和 index 操作，相应的视图在 app/views/novel/index.rhtml。

```
<h1><%= @title %></h1>

<p>I looked up, but saw only the number <%= @counter %>.</p>

<p>"What are you doing here?" I asked sharply. "Was it <%=
@counter.succ %> who sent you?"</p>
```

该视图在 NovelController#index 运行后被解释。下面说明视图可以和不可以访问的地方：

•可以访问实例变量@title 和@counter，因为它们在 NovelController#index 运行结束时被定义在 NovelController 对象上。

•可以调用实例变量@title 和@counter 的实例方法。

•不可以访问实例变量@help_message，因为改变量是由 helper_method 方法定义的，而该方法永远也不会被调用。

•不可以访问变量 one_plus_one，因为这不是一个实例变量：它对于 index 方法来说是本地的。

•即使其运行在 NovelController 的上下文中，也不可以调用 NovelController 的任何方法，不管是 helper_method 或者是 set_another_variable，都不可以。而且不可以再次调用 index。

讨论

控制器的操作方法负责创建和存储（在实例变量里）所有视图工作涉及的对象。这些变量可能很简单，如字符串，也可能是很复杂的 helper 类。每一种形式下，大部分的应用程序逻辑都必须在控制器里。这在类似数据结构迭代的视图里没有什么问题，但是大部分工作会发生在控制器，或者某个在实例变量间发布的对象里。

Rails 为每个请求实例化新的 NovelController 对象。这意味着无法通过将其放入控制器实例变量来在请求间维持数据。不管重新加载多少次页面，@counter 变量都不会超过 2。每次调用 increment_counter 时，都是在一个全新的 NovelController 对象里实现的。

类似任何 Ruby 类，Rails 控制器可以定义类变量和常数，但是它们对于视图而言不可用。考虑一个类似如下的 NovelController：

```
class NovelController < ApplicationController

  @@numbers = [1, 2, 3]

  TITLE = 'Revenge of the Counting Numbers'

end
```

@@numbers 和 TITLE 在这个控制器的任何视图里都不可以被访问。它们只能够被控制器方法调用。然而，在控制器上下文之外定义的常数是可以被每个视图访问的。当想在某个容易改变的位置声明 Web 站点名字时，这很是有用。config/environment.rb 文件是定义这些常数的好地方：

```
# config/environment.rb
AUTHOR = 'Lucas Carlson'
...
```

在基于对象的编程中使用全局变量总是一个坏主意。但是 Ruby 的确使用了它们，一个全局变量一旦定义后在所有的视图里都可用。它们是全局可用的，而不用管它们是否定义在某个操作、控制器的作用域里或是任何作用域外。

```
$one = 1
```

```
class NovelController < ApplicationController

  $two = 2

  def sequel

    $three = 3
  end
end
```

如下是视图，sequel.rhtml，它们使用 3 个全局变量：

```
Here they come, the counting numbers, <%= $one %>, <%= $two %>, <%= $three %>.
```

15.3 创建页眉和页脚的布局问题

想要为 Web 应用程序上的每一页创建页眉和页脚。某页必须有特定的页眉页脚，想要动态决定对于给定请求该采用哪种页眉页脚。

解决方案

多数 Web 应用程序要求定义页眉页脚文件，并且自动包含这些文件到每一页的顶端和底部。Rails 转化了这种模式。被调用的单一文件包含页眉和页脚，并且每个特殊页的内容被插入进这些文件中。
在 Web 应用程序中应用每一页的布局，创建名为 app/views/layouts/application.rhtml 的文件。类似如下：

```
<html>
  <head>
  <title>My We
 bsite</title>
  </head>
  <body>
  <%= @conte
 nt_for_layout
 %>
  </body>
 </html>
```

任何布局文件里的关键信息是指示<%= content_for_layout %>。这会被每个单独页面的内容所替代。
可以为每个控制器创建自定义的布局，通过在 app/views/layouts 文件夹里独立创建文件来实现。比如，app/views/layouts/status.rhtml 是 status 控制器、StatusController 的布局。PriceController 的布局文件是 price.rhtml。

重载 site-wide layout 来自定义布局，它们并没有加上这个。

讨论

正如主视图模板一样，布局模板已经访问了所有操作设定的实例变量。任何可以在视图里做的事情，都可以在布局模板里完成。这意味着可以完成这样的事情，比如在操作中动态设定页面标题，然后在布局里使用它们：

```
class StatusController <
 ActionController:Base
  def index
  @title = "System Stat
 us"
  end
 end
```

现在 application.rhtml 文件可以访问@title，类似如下：

```
<html>

  <head>
```

```
    <title>My Website - <%= @title %></title>
  </head>
  <body>

    <%= @content_for
_layout %>
  </body>
</html>
```

application.rhtml 并不是仅仅针对 Rails 应用程序的控制器而发生在默认的布局模板里。它这样发生是因为每个控制器都是从 ApplicationController 继承而来。默认来说，布局的名字是从控制器类名得到的。因此 ApplicationController 转变为 application.rhtml。如果有一个名为 MyFunkyController 的控制器，那么布局的默认文件名会是 app/views/layouts/my_funky.rhtml。如果这个文件并不存在，Rails 会查找对应于 MyFunkyController 父类的布局，并且会在 app/views/ layouts/application.rhtml 里找到。

要想改变一个控制器的布局文件，调用其布局方法：

```
class FooController < ActionController:Base

  # Force the layout for /foo to be app/views/layouts/bar.rhtml,

  # not app/view/layouts/foo.rhtml.

  layout 'bar'


end
```

如果在某次操作中使用 render 方法（参考 15.5 节），可以传递:layout 变量到 render 里，并且给此次操作一个不同于其他控制器的布局。在本例中，FooController 的大多数操作使用 bar.rhtml 作为它们的布局，但是 count 操作使用的是 count.rhtml：

```
class FooController < ActionController:Base
  layout 'bar'

  def count

    @data = [1,2,3]
    render :layout => 'count'
  end
end
```

甚至可以得到没有布局的操作。本段代码赋给 FooController 的所有操作一个 bar.html 的布局，除去 count 操作，它根本没有任何布局：它负责自己所有的 HTML。

```
class FooController < ActionController:Base
  layout 'bar', :except => 'count'
end
```

如果需要动态计算布局文件，传递一个方法符号到布局方法中。这告诉布局在每次请求时都调用该方法。该方法的返回值定义了布局文件。方法可以调用 action_name 来决定当前请求的操作名字。

```
class FooController < ActionController:Base
  layout :figure_out_layout

private

  def figure_out_layout
    if action_name =~ /pretty/
      'pretty'              # use pretty.rhtml for the layout
    else
      'standard'            # use standard.rhtml
    end
  end
end
```

最后，布局接受 lambda 函数作为一个参数。这使得可以用更少的代码动态决定布局：

```
class FooController < ActionController:Base

  layout lambda { |controller| controller.logged_in? ? 'user' : 'guest' }
end
```

对于程序员和设计者来说，使用布局文件取代单独的页眉和页脚，这都是很好的事情：更加容易查看这个图片。但是如果需要使用显式的页眉页脚，那么也可以。创建名为 app/views/layouts/_header.rhtml 和 app/views/layouts/_footer.rhtml 的文件。这里的下划线表明它们是“局部变量”（参考 15.14 节）。要使用它们，设置操作使之不使用任何布局，并且在视图文件里编写下述代码：

```
<%= render :partial => 'layouts/header' %>

... your view's content goes here ...

<%= render :partial => 'layouts/footer' %>
```

参考

- 15.5 节“用 render 显示模板”
- 15.14 节“重构视图为视图的部分片断”

15.4 重新定位不同的位置问题

想重新定位用户到应用程序的另一个操作中，或者到某个外部 URL。

解决方案

ActionController::Base 类（ApplicationController 的父类）定义了名为 redirect_to 的方法，它实现 HTTP 重新定位。要重新定位到另一个站点，可以将其当作一个字符串传递到一个 URL 里。要重新定位到应用程序里的不同操作，传递给它一个指定的控制器、操作和 ID 的散列。

这是一个 BureaucracyController，在不同的操作间来回搅乱进入的请求，最终发送客户端到一个外部站点：

```
class BureaucracyController < ApplicationCo
  ntroller
   def index
   redirect_to :controller => 'bureaucracy', :
  action =>
  'reservation_window'
   end
```

```
def reservation_window
   redirect_to :action => 'claim_your_form', :id
  => 123
   end
```

```
def claim_your_form
   redirect_to :action => 'fill_out_your_form', :id =>
    params[:id]
   end
```

```
def fill_out_your_form
   redirect_to :action => 'form_processing
  '
   end
```

```
def form_processing

   redirect_to "http://www.d

  mv.org/"

   end

  end
```

如果运行 Rails 服务器并且在浏览器上单击 http://localhost:3000/bureaucracy/，会最终定位到 http://www.dmv.org/。Rails 服务器记录会显示链接到那里的 HTTP 请求：

```
"GET /bureaucracy HTTP/1.1" 302
"GET /bureaucracy/reservation_window HTTP/1.1" 302
"GET /bureaucracy/claim_your_form/123 HTTP/1.1" 302
```

```
"GET /bureaucracy/fill_out_your_form/123 HTTP/1.1" 302
"GET /bureaucracy/form_processing HTTP/1.1" 302
```

不需要为每个操作创建视图模板，因为 HTTP 重新定位的主体不会被 Web 浏览器显示。

讨论

redirect_to 方法使用简练的默认方式。如果给它一个没有指定控制器的散列，它会假定用户想要在同一个控制器里移动至另一个操作。如果不考虑操作，会假定用户涉及的是 index 操作。

由“解决方案”里所示简单的重新定位可见，可能会认为 redirect_to 的确中止了操作方法，并且执行了一次立即的 HTTP 重新定位。这是不对的。操作方法会继续运行直到其结束，或者该调用返回。redirect_to 方法无法实现一次重新定位：它告诉 Rails 一旦操作方法结束运行的话，那么实现一次重新定位。

如下是该问题的解释。可能会认为如下 redirect_to 的调用会阻止 do_something_ dangerous 方法的调用。

```
class DangerController < ApplicationController

  def index

    redirect_to (:action => 'safety') unless params[:i_like_danger]


    do_something_dangerous

  end

  # ...
end
```

实际上不会。唯一能够在运行中中止一次操作方法的方式是调用 return（注 2）。真正需要做的是：

```
class DangerController < ApplicationCon
  troller def index redirect_to (:action
  => 'safety') and return unlessparams
  [:i_like_danger] do_something_danger
  ous end
```

注 2：可以抛出异常，但是当重新定位没有发生时，用户会看到一个异常屏幕。

```
end
```

注意 redirect_to 结尾的 and return。在告诉 Rails 重新定位用户到另一页面后，又要执行代码，这种情况通常很少见。为了避免出现问题，形成添加的习惯，并且在 redirect_to 或者 render 调用结束后返回。

参考

•ApplicationController::Base#redirect_to 和 ApplicationController::Base#url_for 方法生成的 RDoc

15.5 用 render 显示模板问题

Rails 的操作方法与视图模板之间的默认映射对于用户而言灵活性不够。想要自定义一个模板，该模板通过直接调用 Rails 的渲染代码来实现特定操作的渲染实现。

解决方案

渲染发生在 ActionController::Base#render 方法里。Rails 的默认操作是在操作方法运行后调用 render，将此方法映射到相应的视图模板。Foo 操作映射到 foo.rhtml 模板。可以在操作方法内调用 render 来使得 Rails 渲染不同的模板。这个控制器定义了两种操作，都是用 shopping_list.rhtml 模板来实现渲染的：

```
class ListController < ApplicationController

  def index

    @list = ['papaya', 'polio vaccine']


    render :action => 'shopping_list'

  end

  def shopping_list
    @list = ['cotton balls', 'amino
     acids', 'pie']
    end
   end
```

默认来说，render 假定用户在与控制器通话，并且在操作 render 被调用时已经在运行。如果调用无参数的 render，Rails 会以正常方式处理。但是要指定'shopping_list' 作为视图重载默认方式，使得 index 操作使用 shopping_list.rhtml 模板，就像 shopping_list 操作一样。

讨论

尽管使用的是相同的模板，访问 index 操作和访问 shopping_list 操作却并不相同。它们显示不同的列表，因为 index 定义的列表和 shopping_list 不同。

回顾 15.4 节，redirect 方法没有实现立即的 HTTP 重新定位。它告诉 Rails 一旦当前操作方法运行结束，那么实现一次重新定位。相似地，render 方法并不立即实现渲染。它只是告诉 Rails 在操作完成后，渲染哪一个模板。

考虑如下例子：

```
class ListController < ApplicationController

  def index

    render :action => 'shopping_list'
```

```
    @budget = 87.50
  end

  def shopping_list
    @list = ['lizard food', 'baki
  ng soda']
    end
  end
```

可能会认为调用 index 设置@list 而不是@budget。实际上，反过来才是正确的。调用 index 设置@budget 而不是@list。
@budget 变量被设置是因为 render 不会停止当前操作的执行。调用 render 更像是将某条信息装进信封，供 Rails 在今后某个时间点打开阅读。仍然可以自由设置实例变量，并且调用其他方法。一旦操作方法返回了，Rails 会打开信封并且使用其中包含的渲染策略。
@list 变量不被设置，因为 render 调用不会调用 shopping_list 操作。它只是实现现有的操作，index，使用 shopping_list.rhtml 模板而不是 index.rhtml 模板。这里甚至不需要 shopping_list 操作：只需要一个名为 shopping_list.rhtml 的模板。
如果的确想要从另一个操作中调用某个操作，可以显式调用该操作方法。代码会使得 index 设置@budget 和@list：

```
class ListController < ApplicationController

  def index

    shopping_list and render :action => 'shopping_list'

    @budget = 87.50

  end

end
```

这样的“封装”行为的另一个结果是在一次单一的客户端请求里永远不可以调用 render 两次（这对于很类似 render 的 redirect_to 也是一样的，redirect_to 也是在信封里封装了某条信息）。
如果编写如下代码，Rails 会罢工。用户给了它两个封装了的信封，但它不知道该打开哪一个：

```
class ListController < ApplicationController

  def plain_and_fancy

    render :action => 'plain_list'

    render :action => 'fancy_list'
```

```
  end
end
```

但是下述代码是正确的，因为任何给定请求只会触发 if/else 语句的某一个分支。不管发生了什么，render 在每次请求里只会被调用一次。

```
class ListController < Applica
  tionController
   def plain_or_fancy
   if params[:fancy]
   render :action => 'fancy_l
  ist'
   else
   render :action => 'plain_li
  st'
   end
   end
  end
```

使用 redirect_to 时，如果要强制操作方法停止运行，可以在调用 render 后，立刻加入一个 return 语句。本代码没有设置@budget 变量，因为执行永远不会越过 return 语句：

```
class ListController < ApplicationController

  def index

    render :action => 'shopping_list' and return

    @budget = 87.50 # This line won't be run.

  end
end
```

参考

- 15.4 节“重新定位不同的位置”
- 15.14 节“重构视图为视图的部分片断”给出了在视图模板里调用 render 的示例

15.6 集成数据库到 Rails 应用程序中问题

想要 Web 应用程序在一个相关数据库里保存持久数据。

解决方案

最困难的部分是设置所有东西：创建数据库并且使得 Rails 与之建立联系。一旦这些完成了，数据库的访问就像编写 Ruby 代码一样简单。

告诉 Rails 如何访问数据库，打开应用程序的 config/database.yml 文件。假定 Rails 应用程序叫做 mywebapp，看上去会类似如下：

```
development:
adapter: mysql
database: mywebapp_development
host: localhost
username: root
password:

test:
adapter: mysql
database: mywebapp_test
host: localhost
username: root
password:

production:
adapter: mysql
database: mywebapp
host: localhost
username: root
password:
```

现在，只需确保 development 部分包含有效的用户名和密码，这是指针对用户数据库不同类型的正确适配器的名称（查看第 13 章的列表）。

现在创建一个数据库表。如果遵照如下惯例，Rails 会自动完成很多数据库工作。如果有必要，可以重载这些惯例，但是现在更简单的方法是很好地使用它们。

表的名称必须是复数形式的名词：比如，“people”、“tasks”、“items”。

表必须包含一个称为 id 的自动递增的主键字段。

如在以下的例子里，使用数据库工具或者 CREATE DATABASE SQL 命令来创建 mywebapp_development 数据库（如果需要帮助，查看第 13 章的介绍章节）。然后在此数据库里创建名为 people 的表。如下 SQL 在 MySQL 里创建 people 表，可以为数据库改编它。

```
use mywebapp_development;

DROP TABLE IF EXISTS 'people';
CREATE TABLE 'people' (
'id' INT(11) NOT NULL AUTO_INCREMENT,
'name' VARCHAR(255),
'email' VARCHAR(255),
```

```
PRIMARY KEY (id)
) ENGINE=InnoDB;
```

现在回到命令行，改变到 Web 应用程序的根目录，并且键入./script/generate model Person。这会生成一个 Ruby 类，它知道如何操作 people 表。

```
$ ./script/generate model Person
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/person.rb
create test/unit/person_test.rb
create test/fixtures/people.yml
```

注意模型被命名为 Person，即使表被命名为 people。如果遵守这样的惯例，Rails 会自动处理这些复数形式的名词（查看 15.7 节）。

Web 应用程序现在已经通过 Person 类访问了 people 表。再次通过命令行，运行如以下命令：

```
$ ./script/runner 'Person.create(:name => "John Doe", \
:email => "john@doe.com")'
```

这段代码在 people 表里创建一个新的记录项。（如果已经阅读过 13.11 节，会知道这是 ActiveRecord 代码。）

要想在应用程序里访问 peoson，创建一个新的控制器和视图：

```
$ ./script/generate controller people list
exists app/controllers/
exists app/helpers/
create app/views/people
exists test/functional/
create app/controllers/people_controller.rb
create test/functional/people_controller_test.rb
create app/helpers/people_helper.rb
create app/views/people/list.rhtml
```

编辑 app/view/people/list.rhtml，看上去如下：

```
<!-- list.rhtml -->
<ul>
<% Person.find(:all).each do |person| %>
<li>Name: <%= person.name %>, Email: <%= person.email %
></li>
<% end %>
</ul>
```

启动 Rails 服务器，访问 http://localhost:3000/people/list/，会看到排列好的 John Doe。

Person 模型类可以在 Rails 应用程序的所有部分对其访问：控制器、视图、helper 以及 mailer。

讨论

到现在为止，本节创建的应用程序只是用到控制器和视图（注 3）。Person 类，以及其重要的数据库表，给出了 Model-View-Controller 三角关系中的 Model 部分的首次示例。

相关数据库通常是存储真实世界模型的最佳选择，但是很难直接编程相关数据库。Rails 使用 ActiveRecord 库隐藏 people 表到 Person 类之后。类似 Person.find 的方法使得可以不用编写 SQL 就能实现 person 数据库表的搜索。结果会自动转变为 Person 对象。ActiveRecord 的基础在 13.11 节里有所介绍。

Person.find 方法有很多可选的参数。如果给其传递一个整数，会查询唯一 ID 是整数的 person 记录项，并且返回一个合适的 Person 对象。:all 和:first 符号从表（Person 对象数组）里抓取所有的记录项，或者仅仅是匹配的第一个 person。可以通过指定:limit 或者:order 来限制或者排序数据库项，甚至可以通过:conditions 设置原始 SQL 条件。

如下介绍如何在包含邮件地址的 people 表里找到前 5 个记录项。结果会是包含 5 个 Person 对象的列表，以它们的名字字段排序。

```
Person.find(:all,
    :limit => 5,
    :order => 'name',
    :conditions => 'email IS NOT NULL')
```

config/database.yml 的 3 个不同的部分指定 Rails 应用程序不同的时间里使用的 3 个不同的数据库：

注 3：更确切地说，我们的模型嵌入在 controller 里，就像添加 hoc 数据结构比如很难编写的 shopping 列表。

开发数据库运行应用程序时使用的数据库。通常填充的是测试数据。

测试数据库应用程序运行测试时，单元测试框架使用的数据库。其数据是单元测试框架自动组装的。

产品数据库 Web 站点运行动态数据时使用的数据库模式。

除非显式设置 Rails 运行在产品或者测试模式里，否则默认为开发模式。因此要想启动，仅仅需要确保 database.yml 的开发部分的设置正确。

参考

- 第 13 章
- 13.11 节“用 ActiveRecord 使用对象关系映射”
- 13.13 节“以编程方式构建查询”

•13.14 节“用 ActiveRecord 确认数据”

•ActiveRecord 并不能完成 SQL 可以完成的任何事情。对于复杂的数据库操作，需要使用 DBI 或者绑定至特定类型数据库的一种 Ruby。这些话题在 13.15 节“阻止 SQL 注入攻击”里都已有所介绍，该节给出了关于 database.yml 文件格式的更多细节

15.7 理解复数规则问题

想要理解和自定义 Rails 的自动复数化名词的规则。

解决方案

可以在应用程序的任何部分使用 Rails 的复数化功能，但是 ActiveRecord 是 Rails 自动复数化的唯一主要部分。ActiveRecord 通常期望表名是复数名词形式，对应的模式类是同一个名词的单数形式。因此当创建一个模型类时，通常需要使用单数名称。Rails 自动将其复数化：• 模型的相应表名

- has_many 关系
- has_and_belongs_to_many 关系

比如，如果创建一个 LineItem 模型，表名自动变为 line_items。也要注意表名全变为小写，并且原始名词现在被下划线分割开。
如果接着创建一个 Order 模型，相应的表需要被命名为 orders。如果想要描述有很多行项目的排序，代码如下所示：

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

正如参考的表名，has_many 关系里使用的符号被复数化并被下划线分隔开。对于表间

的其他关系也有类似的机制，比如 has_and_belongs_to_many 关系。

讨论

ActiveRecord 复数化这些名称，使得代码读起来更像是英文句子：has_many :line_items 可以被读成“has many line items”。如果复数化会造成困惑，可以通过设定 ActiveRecord::Base.pluralize_table_names 为 false 来禁止该功能。在 Rails 里，完成这个的最简单方法就是将如下代码加入到 config/environment.rb:里：

```
Rails::Initializer.run do |config|
  config.active_record.pluralize_table_names = false
end
```

如果应用程序知道一些特定的单词是 ActiveRecord 不知道如何复数化的，则可以通过操作 Inflector 类定义属于自己的复数化规则。设定“foo”的复数形式是“fooze”，构建应用程序来管理 fooze。在 Rails 里，可以通过添加如下代码到 config/environment.rb 来指定这样的转换：

```
Inflector.inflections do |inflect|

  inflect.plural /^(foo)$/i, '\1ze'

  inflect.singular /^(foo)ze/i, '\1'


end
```

在这种情况下，更简单的方法是使用 irregular 方法：

```
Inflector.inflections do |inflect|
  inflect.irregular 'foo', 'fooze'
end
```

如果有些名词不能够被改变（通常因为它们是聚集名词，或者因为它们的复数形式和其单数形式相同），那么可以将其传递给 uncountable 方法：

```
Inflector.inflections do |inflect|
  inflect.uncountable ['status', 'furniture', 'fish', 'sheep']
end
```

Inflector 类是 activesupport gem 的一部分，可以在 ActiveRecord 或 Rails 的外部使用它，并将其作为复数化英文单词的通用方式。如下是标准的 Ruby 程序：

```
require 'rubygems'
require 'active_support/core_ext'

'blob'.pluralize # => "blobs"'child'.pluralize # => "children"'octopus'.pluralize # => "octopi"'octopi'.singularize # => "octopus"'people'.singularize # => "person"
'goose'.pluralize # => "gooses"Inflector.inflections { |i| i.irregular 'goose', 'geese' }'goose'.pluralize # => "geese"
'moose'.pluralize # => "mooses"Inflector.inflections { |i| i.uncountable 'moose' }'moose'.pluralize # => "moose"
```

参考

- 13.11 节“用 ActiveRecord 使用对象关系映射”

15.8 创建登录系统

问题

想使得应用程序支持基于用户账户的登录系统。用户使用唯一的用户名和密码登录，就像大多数商业和社区 Web 站点使用的一样。

解决方案

创建一个包含非空用户名和密码字段的 users 表。创建该表的 SQL 看上去就像如下的 MySQL 示例：

```
use mywebapp_development;

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

  `id` INT(11) NOT NULL AUTO_INCREMENT,

  `username` VARCHAR(255) NOT NULL,

  `password` VARCHAR(40) NOT NULL,

  PRIMARY KEY (`id`)

);
```

进入应用程序的主目录并且生成对应于该表的 User 模型：

```
$ ./script/generate model User exists app/models/ exists test/unit/ exists test/fixture
      s/ create app/models/user.rb create test/unit/user_test.rb create test/fixtures/u
      sers.yml
```

打开生成的文件 app/models/user.rb 并且编辑它使之看上去类似：

```
class User < ActiveRecord::Base
  validates_uniqueness_of :username
  validates_confirmation_of :password, :on => :create
  validates_length_of :password, :within => 5..40

  # If a user matching the credentials is found, returns the User object.
  # If no matching user is found, returns nil.
  def self.authenticate(user_info) find_by_username_and_password(user_info[:use
rname],
user_info[:password])
end
end
```

现在已经得到一个代表用户账户的 User 类，一种基于数据库存储项目来确认用户名和密码的方式。

讨论

“解决方案”里给出的简单 User 模型定义了一种完成用户名/ 密码确认的方法，以及一些强加限制到存储数据上的确认规则。这些确认规则告诉 User：

- 保证每个用户名是唯一的。两个不同的用户不能使用相同的用户名。
- 保证不管设置的是什么样的密码属性，password_confirmation 属性都具有相同的值。
- 保证密码属性值介于 5～40 字符长度之间

现在来为该模型创建一个控制器。它有一个显示登录页面的 login 操作，一个检查用户名和密码的 process_login 操作，以及一个鉴别登录会话的 logout 操作。因为用户账户会实际完成某些事情，也需要添加一个 my_account 操作：

```
$ ./script/generate controller user login process_login logout my_account
```

```
  exists  app/controllers/
  exists  app/helpers/
  create  app/views/user
  exists  test/functional/
  create  app/controllers/user_controller.rb
  create  test/functional/user_controller_test.rb
  create  app/helpers/user_helper.rb
  create  app/views/user/login.rhtml
  create  app/views/user/process_login.rhtml
  create  app/views/user/logout.rhtml
```

编辑 app/controllers/user_controller.rb 来定义 3 个操作：

```
class UserController < ApplicationController

  def login
    @user = User.new
    @user.username = params[:username]
  end

  def process_login
    if user = User.authenticate(params[:user])
      session[:id] = user.id # Remember the user's id during this session
      redirect_to session[:return_to] || '/'
```

```
    else
      flash[:error] = 'Invalid login.'
      redirect_to :action => 'login', :username =>
        params[:user][:username]
      end
    end

    def logout
      reset_session
      flash[:message] = 'Logged out.'
      redirect_to :action => 'login'
    end

    def my_account
    end
end
```

现在针对视图。process_login 和 logout 操作只是重新定位到其他操作，因此只需针对 login 和 my_account 的视图。如下是针对 login 的视图：

```
<!-- app/views/user/login.rhtml -->
<% if @flash[:message] %><div><%= @flash[:message] %></div><% end %>
<% if @flash[:error] %><div><%= @flash[:error] %></div><% end %>

<%= form_tag :action => 'process_login' %>

  Username: <%= text_field "user", "username" %>&#x00A;
  Password: <%= password_field "user", "password" %>&#x00A;
  <%= submit_tag %>

<%= end_form_tag %>
```

@flash 实例变量是一个类散列的对象，用于为操作间的用户存储临时消息。当 logout 操作设置 flash[:message]以及 login 的重新定位，或者 process_login 设置 flash[:error] 以及 login 的重新定位时，结果对于 login 操作的视图是可用的。然后它们被清除。如下是 my_account 的简单视图：

```
<!-- app/views/user/my_account.rhtml -->
<h1>Account Info</h1>

<p>Your username is <%= User.find(session[:id]).username %>
```

在 users 表里创建记录项，启动服务器，会发现可以从 http://localhost:3000/user/login 登录，从 http://localhost:3000/user/my_account 查看账户信息。

```
$ ./script/runner 'User.create(:username => "johndoe", \
                                :password => "changeme")'
```

这里还有一个遗漏的部分：即使没有登录也可以访问 my_account 操作。没有方法可以关闭未经鉴定的用户的操作。添加如下代码到 app/controllers/application.rb 文件：

```
class ApplicationController < ActionController::Base
  before_filter :set_user

  protected
    def set_user
    @user = User.find(session[:i
  d]) if @user.nil? && session[:
  id]
    end

  def login_required

    return true if @user

    access_denied

    return false


  end

  def access_denied
    session[:return_to] = request.request_uri
    flash[:error] = 'Oops. You need to login before you can view that

page.'
      redirect_to :controlle
    r => 'user', :action
    => 'login'
     end
    end
```

这段代码定义了两个过滤器，set_user 和 login_required，可以应用它们到操作或者控制器上。set_user 过滤器在每个操作上运行（因为我们传递它到 ApplicationController 的 before_filter，而 ApplicationController 是所有 controller 的父类）。set_user 方法在用户已经登录时设置实例变量@user。现在有关登录用户的信息在应用程序里就可用了。操作方法和视图可以正常使用这个实例变量。这对于不要求登录的操作尤其有用：比如，主布局视图可能会在每页上显示登录用户的名字。可以通过为 login_required 方法传递符号到 before_filter，来禁止未经鉴别的用户使用特定的操作或者控制器。如下显示如何保护定义在 app/controllers/ user_controller.rb 里的 my_account 操作：

```
class UserController < ApplicationController
  before_filter :login_required, :only => :my_account
end
```

现在如果在未登录时试图使用 my_account 操作，会被重新定位到登录页面。

参考

- 13.14 节“用 ActiveRecord 确认数据”
- 15.6 节“集成数据库到 Rails 应用程序中”
- 15.9 节“保存散列化的用户密码到数据库中”
- 15.11 节“设置并找回会话信息”
- 不用再由自己完成这部分工作，可以安装 login_generator gem，并且使用其 login 产生器：它会给应用程序一个 User 模型以及实现基于密码的验证系统的控制器。查看 http://wiki.rubyonrails.com/rails/pages/LoginGenerator，以及 http:// wiki.rubyonrails.com/rails/pages/AvailableGenerators 有其他产生器的信息（包括久负盛名的 model_security_generator）

15.9 保存散列化的用户密码到数据库中问题

15.8 节定义的数据库表以普通文本保存用户的密码。这不是个好方法：如果某人侵入数据库，就能够得到所有用户的密码。所以最好保存为密码的安全散列。这样的话，没有密码（因此没有人能够窃取它），但是可以确保用户知道自己的密码。

解决方案

重新创建 15.8 节的 users 表，取代密码字段，它有一个 hashed_password 字段。如下是完成这些的 MySQL 代码：

```
use mywebapp_development;

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

  `id` INT(11) NOT NULL AUTO_INCREMENT,

  `username` VARCHAR(255) NOT NULL,

`hashed_password` VARCHAR(40) NOT
 NULL,
 PRIMARY KEY (id)
);
```

打开 15.8 节创建的文件 app/models/user.rb，并且类似如下编辑它：

```
require 'sha1'

class User < ActiveRecord::Base

  attr_accessor :password

  attr_protected :hashed_password

  validates_uniqueness_of :username

  validates_confirmation_of :password,

   :if => lambda { |user| user.new_record? or not user.password.blank?
 }
validates_length_of :password, :within => 5..40,
:if => lambda { |user| user.new_record? or not user.password.blank?
 }

def self.hashed(str)
  SHA1.new(str).to_s
  end
```

```
# If a user matching the credentials is found, returns the User object.
# If no matching user is found, returns nil.
def self.authenticate(user_info)
```

```
user = find_by_username(user_info[:usern
ame])
if user && user.hashed_password == has
hed(user_info[:password])
return user
end
end
```

```
private
before_save :update_password
```

```
# Updates the hashed_password if a
plain password was provided.
def update_password
if not password.blank?
self.hashed_password = self.class.has
hed(password)
end
end
end
```

一旦完成了这些，应用程序会像以前一样起作用（虽然需要转变任何预先存在的用户账户为新的密码格式）。不需要修改任何控制器或者视图代码，因为 User.authenticate 方法采取和以前一样的工作方式。这是分离商业逻辑和表示逻辑的优点之一。

讨论

现在我们的用户模型有 3 段。第一段是加强了的确认代码。用户模型如下：

- 提供获取器和设置器密码属性。
- 确保数据库的 hashed_password 字段无法从外部访问。
- 确保每个用户有唯一的用户名。

当新的用户被创建，或者密码被更改时，User 确保：

- password_confirmation 属性的值等于密码属性的值。
- 密码在 5～40 字符长度之间。

代码的第二部分像以前一样定义 User 类。添加一个新的类级别的方法，hashed，在纯文本密码上执行散列化功能。如果想要在以后改变散列化机制，仅仅需要改变这个方法即可（同时移动所有现有密码）。

模型中代码的第三部分是私有的实例方法，update_password，它实现数据库中纯文本密码属性与散列化版本之间的同步。before_save 的调用设置该方法在 User 对象保存至数据库之前被调用。这样可以通过设置密码为纯文本值来改变用户的密码，而不用自己来完成这样的散列。

参考

- 13.14 节“用 ActiveRecord 确认数据”
- 15.8 节“创建登录系统”

15.10 转义显示用的 HTML 和 JavaScript 问题

想要显示可能包含 HTML 或 JavaScript 的数据，并且不让浏览器将其当作 HTML 渲染或者翻译该段 JavaScript。这在显示用户输入的数据时尤其重要。

解决方案

传递数据字符串到 h( )辅助函数来转义其 HTML 记录项。也就是说，代替如下：

```
<%= @data %>
```

为：

```
<%=h @data %>
```

h( )帮助器函数转变字符为 HTML 记录项等式：与符号（&）、双引号（”），左三角括号（<）以及右三角括号（>）。

讨论

在 Rails 源代码中找不到 h( )帮助器函数的定义，因为这是 ERb 的内置帮助器函数 html_escape( )的快捷方式。

JavaScript 在类似<SCRIPT>的 HTML 标签里被展开，因此转义 HTML 字符串会压制 HTML 里的所有 JavaScript。然而，有时仅仅需要转义 JavaScript 为字符串。Rails 添加了可以使用的名为 escape_javascript( )的帮助器函数。该函数的功能不多：只是转变行转换符为字符串“\n”，同时在单和双引号前添加反斜杠。这在想要在自己的 JavaScript 代码里使用任意数据时很有用：

```
<!-- index.rhtml -->
<script lang="javascript">
var text = "<%= escape_javascript @javascript_alert_text %>";
alert(text);
</script>
```

参考

- 第 11 章

15.11 设置并找回会话信息问题

想要关联数据到每个使用应用程序的独特 Web 客户端。数据需要在 HTTP 请求间保持恒定。

解决方案

可以使用 cookie（参考 15.12 节），但是将数据放到用户的会话中通常更简单。Rails 站点的每个访问者会被自动给予一个会话 cookie。Rails 以 cookie 值作为键，在服务器上创建一个任意数据的散列。遍历全部的 Rails 应用程序，控制器、视图、helper 以及 mailer，可以通过调用称为 session 的方法来访问这个散列。保存在该散列里的对象在同一个 Web 浏览器的请求间保持恒定。控制器里的下述代码跟踪客户端首次访问 Web 站点的时间：

```
class IndexController < Applicati
  onController
   def index
   session[:first_time] ||= Time.
  now
   end
  end
```

在视图里，可以编写如下代码显示时间（注 4）：

```
<!-- index.rhtml -->
You first visited this site on <%= session[:first_time] %>.

That was <%= time_ago_in_words session[:first_time] %> ago.
```

讨论

Cookie 和会话很是相似。它们都保存有关站点访问者的持久数据。也都在 HTTP 顶端实现正式的操作，该 HTTP 没有属于自己的状态。cookie 和会话的主要区别在于 cookie 里，所有数据保存在访问者计算机的小型 cookie 文件中，而会话里，所有数据保存在 Web 服务器里。客户端只是保存一个小型的会话 cookie，它包含联系到服务器上数据的唯一 ID。任何私人数据都保存在访问者的计算机上。有几个理由使得可能需要使用会话代替 cookie：

•cookie 只能保存 4K 比特数据。

•cookie 只能保存字符串值。

•如果在 cookie 里保存个人信息，它会被截取，除非所有客户端的请求都被 SSL 封装。即使这样，站点间的脚本攻击可能会阅读客户端 cookie 并且得到敏感信息。

另一方面，cookie 在如下情况很有用：

- 信息既不敏感，也不是很大。
- 不需要在服务器上保存每个访问者的会话信息。
- 需要应用程序的速度，并不是每一页都需要访问会话数据。通常来说，使用会话优于在 cookie 里保存数据。

注 4：帮助函数 tine_ago_in_words()计算某个特定时间起至返回英文文本之间的时间，文本类似为“about a minute”或“5 hours”或“2 days”。这是一种简单直观的方式告诉用户日期的含义。

可以在会话里包含模型对象：这会在每个请求里从数据库找回相同的对象时省去很多麻烦。然而，如果想要完成这些，可以在应用程序控制器里列出所有将要放到会话里的模型。这会在从会话存储里找回数据时降低 Rails 不能解串行对象的风险。

```
class ApplicationController < ActionController::Base
  model :user, :ticket, :item, :history
end
```

然后可以将 ActiveRecord 对象放到会话里：

```
class IndexController < Application
  Controller
  def index
  session[:user] ||= User.find(para
  ms[:id])
  end
end
```

如果站点不需要保存任何会话信息，可以禁止使用这个特性，通过添加如下代码到 app/controllers/application.rb 文件实现：

```
class ApplicationController < ActionControlle
  r::Base
  session :off
end
```

也许可以猜到，还可以使用 session 方法将会话转变为单一的控制器：

```
class MyController < ApplicationContro
  ller
  session :off
end
```

甚至可以将其降为操作级别：

```
class MyController < ApplicationController
  session :off, :only => ['index']

  def index
    #This action will not have any
    sessions available to it
    end
  end
```

会话界面提供给需要在很多操作间保持的数据，可能是通过用户记录项访问站点。如果只是需要传递一个对象（比如状态消息）到下一个操作，那么使用 15.8 节介绍的 flash 构造会更简单：

```
flash[:error] = 'Invalid login.'
```

默认来说，Rails 会话通过 PStore 机制保存到服务器上。该机制使用 Marshal 串行化会话数据到临时文件里。这种方法对于小型站点很有效，但是如果站点有大量的访问者或者需要在多台服务器上并发运行 Rails 应用程序，那么就需要研究另外的方法。

3 种主要可以考虑的方法为 ActiveRecordStore、DRbStore 和 MemCacheStore。ActiveRecordStore 保存会话信息到数据库表中：可以通过在命令行运行 rake create_sessions_table 来设置表。DRbStore 和 MemCacheStore 都创建一个可以在网络上访问的内存散列，但是它们使用不同的库。
Ruby 有一个称为 DRb 的标准库，允许在网络上共享对象（包括散列）。Ruby 也有到 Memcached 守护进程的绑定，它帮助衡量 Web 站点比如 Slashdot 和 LiveJournal。Memcached 工作起来类似直接存储到 RAM，可以自动分布到不同计算机上而不用任何特殊的配置。
要改变会话存储机制，编辑 config/environment.rb 文件类似如下：

```
Rails::Initializer.run do |config|

  config.action_controller.session_store = :active_record_store

end
```

参考

- 15.8 节“创建登录系统”，有一个使用 flash 的例子
- 15.12 节“设置并找回 cookie”
- 16.10 节“在任意数目的计算机间共享散列”
- 16.16 节“用 MemCached 在分布式 RAM 上保存数据”
    - http://wiki.rubyonrails.com/rails/pages/HowtoChangeSessionOptions

15.12 设置并找回 Cookie 问题

想要在 Rails 里设置 cookie

解决方案

回顾 15.11 节，所有 Rails 的控制器、视图、helper 以及 mailer 访问名为 sessions 的方法，返回当前客户端会话信息的散列。控制器、helper 以及 mailer（不是用户的视图）也访问名为 cookie 的方法，返回当前客户端 HTTP cookie 的散列。

要为用户设置一个 cookie，只需简单地在该散列里设置键/ 值对。比如，要保持跟踪访问者查看过的页面数目，可以设置“visits”cookie：

```
class ApplicationController < ActionController::Base
  before_filter :count_visits

  private

  def count_visits
    value = (cookies[:visits] || '0').to_i
    cookies[:visits] = (value + 1).to_s
    @visits = cookies[:visits]
  end
end
```

调用 before_filter 告诉 Rails 在调用任何操作方法之前运行该方法。私有声明使得 Rails 不会将 count_visits 方法当作可以公开查看的操作方法。

因为 cookie 对于视图不是直接可用的，count_visits 使得:visits cookie 的值可以当作实例变量@visits 使用。这个变量可以从视图里访问：

```
<!-- index.rhtml -->
You've visited this website's pages <%= @visits %> time(s).
```

HTTP cookie 值只能是字符串。Rails 可以自动转化某些值为字符串，但是只把值存储到 cookie 里，这是最安全的方式。如果需要保存无法简单与字符串来回转变的对象，很可能需要将它们保存到会话散列里。

讨论

在很多情况下需要更多的 cookie 上的控制权。比如，Rails cookie 默认在用户关闭浏览器会话时失效。如果想要改变浏览器失效时间，可以赋给 cookie 一个包含:expires 关键值的散列，以及一个 cookie 失效的时间。如下面的 cookie 会在一个小时之后失效

（注 5）：

```
cookies[:user_id] = { :value => '123', :expires => Time.now + 1.hour}
```

还有另外一些可以传递进 cookie 的 cookie 散列的选项。

Cookie 请求的域名：

```
:domain
```

注 5：Rails 扩展 Ruby 的数值类，使之包含一些十分有用的方法（比如此处所示的 hour 方法）。这些方法转变给定的单元为秒。比如，Time.nor+1.hour 和 Time.nor+3600 是一样的，因为 1.hour 返回一个小时的秒数。其他有用的方法还包括 minutes、hours、days、months、weeks 以及 years。因为它们全都被转变为秒数，甚至可以把它们都加起来，比如 1.week+3.days。

cookie 请求的 URL 路径（默认来说，cookie 请求到入口域：这意味着如果在相同域名处运行多个应用程序，它们的 cookie 可能会冲突）：

```
:path
```

不管该 cookie 是否安全（安全的 cookie 只在 HTTPS 连接里传输，默认值为 false）：

```
:secure
```

最后，Rails 提供一个简单快捷的删除 cookie 的方式：

```
cookies.delete :user_id
```

当然，每个 Ruby 散列实现一种 delete 方法，但是 cookie 散列有一点不同。它包含特殊的代码，因此调用 delete 不仅会从 cookie 散列里移除键/ 值对，也会从用户浏览器里移除相应的 cookie。

参考

- 3.5 节“计算日期”
- 15.11 节“设置并找回会话信息”讨论了何时使用 cookie，以及何时使用会话

15.13 提取代码到辅助函数中问题

视图被 Ruby 代码混淆了。

解决方案

创建一个有着十分复杂视图的控制器，查看会发生些什么：

```
$ ./scripts/generate controller list index exists app/controllers/ exists app/helpers/ create
    app/views/list exists test/functional/ create app/controllers/list_controller.rb create
    test/functional/list_controller_test.rb create app/helpers/list_helper.rb create app/vi
   ews/list/index.rhtml
```

编辑 app/controllers/list_controller.rb 类似如下：

```
class ListController < ApplicationC
  ontroller
   def index
   @list = [1, "string", :symbol,
  ['list']]
   end
  end
```

编辑 app/views/list/index.rhtml 包含如下代码。它迭代@list 里的每个元素，并且打印出索引及其对象 ID 的 SHA1 散列：

```
<!-- app/views/list/index.rhtml -->

<ul>

<% @list.each_with_index do |item, i| %>

  <li class="<%= i%2==0 ? 'even' : 'odd' %>"><%= i %>:

    <%= SHA1.new(item.id.to_s) %></li>
<% end %>
</ul>
```

这显得很是混乱，但是如果已经做过很多 Web 编程工作，它们看上去会很相似。

要清除这段代码，需要为控制器移动其中一些到帮助器里。该例中，控制器称为 list，因此其帮助器依赖于 app/helpers/list_helper.rb。

创建名为 create_li 的帮助器函数。给定一个对象及其在列表里的位置，该函数创建适合在 index 视图里使用的<LI>标签：

```
module ListHelper
   def create_li(item, i)
   %{<li class="#{ i%2==0 ? 'e
  ven' : 'odd' }">#{i}:
   #{SHA1.new(item.id.to_s)}</li>}
   end
  end
```

list 控制器的视图访问了所有定义在 ListHelper 里的函数。可以类似如下来清除 index 视图：

```
<!-- app/views/list/index.rhtml -->

<ul>

<% @list.each_with_index do |item, i| %>

  <%= create_li(item, i) %>
<% end %>
</ul>
```

帮助器函数可以完成所有通常从视图里完成的事情，因此这是一种很好的提取重要部分的方法。

讨论

帮助器函数的目的是创建维护性更强的代码，并且在程序员和 UI 设计者之间执行更好的工作分工。可维护代码使得程序员更容易维护，当它在帮助器函数里时，它不会妨碍设计人员，他们能够来回 tweak HTML，而无需过滤代码。

何时使用帮助器的一个很好的规则是大声朗读代码。如果它对于熟悉 HTML 的人来说，听上去没有什么意义，或者它由多于一个的英文短句所组成，那么应该将它隐藏到帮助器后。

这样做的负面作用是必须最小化帮助器里生成的 HTML 的数量。UI 设计者，或者其他熟悉 HTML 的人员采取的方式不会混乱代码，就能找出何处可以找到需要 tweak 的 HTML 数据。

虽然帮助器函数很有用并且经常会被用到，Rails 也提供了 partial，另一种提取代码为转化成更小部分的方法。

参考

- 15.14 节“重构视图为视图的部分片断”中更详细地介绍了 partial

15.14 重构视图为视图的部分片断问题

视图不包含很多 Ruby 代码，但是它仍然变得越来越复杂。想要重构视图的逻辑到分散的、可重用的模板里。

解决方案

可以重构视图模板到称为partial的多个模板中。一个模板可以通过调用render方法来包含其他模板，render 方法在 15.5 节首次出现。

从 15.5 节所示视图的更加复杂的版本开始讨论：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

  <ul>
  <% @list.each do |item| %>
   <li><%= item.name %>
  <%= link_to 'Delete', {:action
  => 'delete', :id => item.id},
   :post => true %>
   </li>
  <% end %>

</ul>


<h2>Add a new item</h2>

<%= form_tag :action => 'new' %>

  Item: <%= text_field "product", "name" %>&#x00A;

  <%= submit_tag "Add new item" %>

<%= end_form_tag %>
```

如下是相应的控制器类，以及一个虚拟的作为模型的 ListItem 类：

```
  # app/controllers/list_controller.rb
  class ListController < ActionController::Base
   def shopping_list
   @list = [ListItem.new(4, 'aspirin'), ListItem.new(19
  9, 'succotash')]
   end

  # Other actions go here: add, d
elete, etc.
  # ...
end
```

```
class ListItem
    def initialize(id, name)
     @id, @name = id, n
    ame
     end
    end
```

该视图包括两部分：第一部分列出所有项，第二部分打印出表格添加一个新项。很明显第一步需要分离新项表格。

可以通过创建一个部分视图来打印出新项表格。要完成这个，需要在 app/views/list/ 里创建新的名为 _new_item_form.rhtml 的文件。文件名前的下划线表明这是一个部分视图，不是 new_item_form 操作的完整的视图。如下是部分文件。

```
<!-- app/views/list/_new_item_form.rhtml -->

<%= form_tag :action => 'new' %>
Item: <%= text_field "item", "value" %>&#x00A;
<%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

要包含一个部分，从某个模板里调用 render 方法。如下是集成到主视图里的_new_item_form 部分。视图看上去几乎一样，但是代码被更好地组织了。

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

 <ul>
 <% @list.each do |item|
 %>
  <li><%= item.name %>

 <%= link_to 'Delete', {:a
 ction => 'delete', :id =>
 item.id},

                :post => true %>

  </li>
<% end %>
</ul>

<%= render :partial => 'new_item_form' %>
```

即使文件名以下划线开始，当调用部分时，仍可能会忽略下划线。

讨论

部分视图继承了控制器提供的所有实例变量，因此可以像访问父视图一样访问相同的实例变量。这就是为什么没有必要为_new_item_form 部分而改变所有表格的代码。

可以创建第二部分来构建为每个列表项目打印出<LI> 标签的代码。如下是_list_item.rhtml：

```
<!-- app/views/list/_list_item.rhtml -->

<li><%= list_item.name %>

<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},

            :post => true %>
</li>
```

如下是修改了的主视图：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>


<ul>

<% @list.each do |item| %>

  <%= render :partial => 'list_item', :locals => {:list_item => item} %>
<% end %>
</ul>

<%= render :partial => 'new_item_form' %>
```

部分视图没有从父视图继承本地变量，因此项目变量需要传递进部分，在特殊的称为:locals 的散列里。它在部分里作为 list_item 可以被访问，因为这就是它在散列中被赋予的名称。
这种情况，在 Enumerable 上的迭代，为每个元素翻译一个部分，这在 Web 应用程序中十分常见，因此 Rails 为此提供了快捷方式。可以更加简化主视图，通过传递数组到 render 里（作为:collection 参数）实现，使之完成这样的迭代：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>

  <%= render :collection => @list, :partial => 'list_item' %>

</ul>


<%= render :partial => 'new_item_form' %>
```

部分针对@list 里的每个元素被翻译一次。每个列表元素作为本地变量 list_item 可用。万一还没有猜出来，这个名字来自于部分自身的名字：render 自动给_foo.rhtml 一个名为 foo 的本地变量。
list_item_counter 是另一个自动设置的变量（再一次，名字反映了模板的名字）。list_item_counter 是迭代集合的当前项的索引。在想改变列表项以不同的风格显示时，这个变量很是有用：

```
<!-- app/views/list/_list_item.rhtml -->
```

```
<li><%= list_item.name %>

<% css_class = list_item_counter % 2 == 0 ? 'a' : 'b' %>

<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},

                {'class' => css_class}, :post => true %>

</li>
```

在当前没有集合的地方，可以通过指定 render 的:object 参数，来传递一个单一对象到部分里。这比创建只需传递一个对象的:locals 所需的整个散列要简单得多。通过:collection，对象被当作本地变量使用，该本地对象的名称基于此部分的名称。
如下有一个例子：发送购物列表到 new_item_form.rhtml 部分，因此新项的表格可以打印出更详细的消息。这给 shopping_list.rhtml 发挥作用提供了机会：

```
<%= render :partial => 'new_item_form', :object => @list %>
```

如下是_new_item_form.rhtml 的新版本：

```
<!-- app/views/list/_new_item_form.rhtml -->
<h2>Add a new item to the <%= new_item_form.size %> already in this
list</h2>

<%= form_tag :action => 'new' %>

  Item: <%= text_field "product", "name" %>

  <%= submit_tag "Add new item" %>

<%= end_form_tag %>
```

参考

- 15.5 节“用 render 显示模板”

15.15 用 script.aculo.us 添加 DHTML 效果问题

想要添加奇特的效果，比如应用程序的淡出，而无需编写任何 JavaScript。

解决方案

每个 Rails 应用程序和一些 JavaScript 库绑定，允许创建 Ajax 和 DHTML 效果。甚至不需要在 Rails 的 Web 站点编写 JavaScript 使能 DHTML。

首先编写主布局模板（查看 15.3 节），在<HEAD>标签调用 javascript_include_tag：

```
<!-- app/views/layouts/application.rhtml -->

<html>

  <head>

    <title>My Web App</title>

    <%= javascript_include_tag "prototype", "effects" %>

  </head>
  <body>
  <%= @content_for_
layout %>
  </body>
</html>
```

现在在视图里，可以调用 visual_effect 方法来实现 script.aculo.us 库里的 DHTML 窍门。

如下是“高亮”效果的例子：

```
<p id="important">Here is some important text, it will be highlighted
when the page loads.</p>

<script type="text/javascript">
<%= visual_effect(:highlight, "important", :duration => 1.5) %>
</script>
```

如下是“淡去”效果的例子：

```
<p id="deleted">Here is some old text, it will fade away when the page
loads.</p>

<script type="text/javascript">

<%= visual_effect(:fade, "deleted", :duration => 1.0) %>

</script>
```

讨论

如上示例代码段在页面加载时被触发，因为它们被封装在<SCRIPT>标签里。在实际应用程序中，可能需要响应用户操作显示文本效果：删除了的项目会淡去消失，或者选中某项时会高亮相关项目。如下是一幅图像，当点击其下方的链接时会发出咯吱的声音：

```
<img id="to-squish" src="bug.jpg">
<%=link_to_function("Squish the bug!", visual_effect(:squish, "to
squish"))%>
```

visual_effect 生成的 JavaScript 代码看上去很想传递进方法的参数。比如，Rails 视图的一段代码如下：

```
<script type="text/javascript">
<%= visual_effect(:fade, 'deleted-text', :duration => 1.0) %>
</script>
```

生成如下 JavaScript：

```
<script type="text/javascript">
new Effect.Fade("deleted-text", {duration:1.0});
</script>
```

这样的相似性意味着 script.aculo.us 库的文档几乎可以直接应用到 visual_effect。这也意味着如果更愿意编写直接的 JavaScript，那么代码对于懂得 visual_effect 的人来说仍然很容易理解。
如下表格列出 Rails 1.0 里的很多可用的效果。

JavaScript 初始化	Rails 初始化
new Effect.Highlight	visual_effect(:highlight)
new Effect.Appear	visual_effect(:appear)
new Effect.Fade	visual_effect(:fade)
new Effect.Puff	visual_effect(:puff)
new Effect.BlindDown	visual_effect(:blind_down)
new Effect.BlindUp	visual_effect(:blind_up)
new Effect.SwitchOff	visual_effect(:switch_off)
new Effect.SlideDown	visual_effect(:slide_down)
new Effect.SlideUp	visual_effect(:slide_up)
new Effect.DropOut	visual_effect(:drop_out)
new Effect.Shake	visual_effect(:shake)
new Effect.Pulsate	visual_effect(:pulsate)
new Effect.Squish	visual_effect(:squish)
new Effect.Fold	visual_effect(:fold)
new Effect.Grow	visual_effect(:grow)
new Effect.Shrink	visual_effect(:shrink)
new Effect.ScrollTo	visual_effect(:scroll_to)

JavaScript 初始化 Rails 初始化

参考

- script.aculo.us 演示（http://wiki.script.aculo.us/scriptaculous/show/ CombinationEffectsDemo）
- 15.3 节“创建页眉和页脚的布局”中有更多关于布局模板的信息
- 15.17 节“创建 Ajax 表格”

15.16 生成操作模型对象的表格

问题

想要定义操作，使得用户创建或者编辑保存在数据库里的对象。

解决方案

让我们创建一个简单的模型，然后为其构建表格。如下是一段 MySQL 代码，创建一个键/ 值对的表：

```
use mywebapp_development;

DROP TABLE IF EXISTS items;

CREATE TABLE `items` (

  `id` int(11) NOT NULL auto_increment,

  `name` varchar(255) NOT NULL default '',

  `value` varchar(40) NOT NULL default '[empty]',

  PRIMARY KEY (`id`)

);
```

现在，从命令行，创建模型类，以及控制器和视图：

```
$ ./script/generate model Item exists app/models/ exists test/unit/ exists test/fixtures/
    create  app/models/item.rb
    create  test/unit/item_test.rb
    create  test/fixtures/items.yml
    create  db/migrate
    create  db/migrate/001_create_items.rb
```

```
$ ./script/generate controller items new create edit

exists app/controllers/

exists app/helpers/

create app/views/items

exists test/functional/

create app/controllers/items_controller.rb

create test/functional/items_controller_test.rb
```

```
create  app/helpers/items_helper.rb

create  app/views/items/new.rhtml

    create  app/views/items/edit.rhtml
```

第一步是要自定义视图。从 app/views/items/new.rhtml 开始。编辑它类似如下：

```
<!--  app/views/items/new.rhtml  -->

<%=  form_tag  :action  =>  "create"  %>

 Name:  <%=  text_field  "item",  "name"  %><br  />

 Value:  <%=  text_field  "item",  "value"  %><br  />

   <%=  submit_tag  %>
<%=  end_form_tag  %>
```

所有这些方法调用生成 HTML：form_tag，打开一个<FORM>标签，submit_tag 生成一个提交按钮，等等。可以手工键入相同的 HTML，Rails 不会介意这些，但是构建方法调用会更容易，也会使得模板更加简洁。

text_field 调用会涉及一些。它创建一个<INPUT>标签，作为文本记录项字段显示在 HTML 表格里。但是它同时绑定该字段的值到@item 实例变量的成员之一。本段代码创建一个文本记录项字段，绑定到@item 的 name 成员：

```
<%=  text_field  "item",  "name"  %>
```

但是@item 实例变量是什么呢？是的，它还没有定义，因为我们使用的是已经生成的控制器。如果现在尝试访问页面/items/new page，可能会得到一个错误，提示预期外的 nil 值。nil 值指的就是@item 变量，它还没有定义就被使用了（在 text_field 调用里）。

让我们自定义 ItemsController 类，从而 new 操作会恰当设置@item 实例变量。同时实现 create 操作，从而当用户单击生成的表格上的提交按钮时确实会发生预想的事情。

```
class  ItemsController  <  ApplicationController

   def  new

      @item  =  Item.new
   end

   def  create

      @item  =  Item.create(params[:item])

      redirect_to  :action  =>  'edit',  :id  =>  @item.id
```

```
    end
end
```

如果访问/items/new page，会看到预期的效果：一个有两个文本记录项字段的表格。“Name”字段是空的，“Value”字段包含“[empty]”的默认数据库值。

填写该表格并提交，会在 items 表里创建新行。会被重新定位到 edit 操作，这个操作并不存在。现在创建它。如下是控制器部分（注意上述 ItemsController#edit 和 ItemsController#create 间的相似性）：

```
class ItemsController < Appl
  icationController
   def edit
   @item = Item.find(param
  s[:id])

  if request.post?
    @item.update_attributes(params[:item])
    redirect_to :action => 'edit', :id => @item.id

  end
 end
end
```

实际上，edit 操作和 create 操作十分相似，它们的表格几乎都是一样的。唯一不同的地方在于 form_tag 的参数：

```
<!-- app/views/items/edit.rhtml -->

<%= form_tag :action => "edit", :id => @item.id %>
  Name: <%= text_field "item", "name" %><br />
  Value: <%= text_field "item", "value" %><br />
  <%= submit_tag %>

<%= end_form_tag %>
```

讨论

这几乎是 Web 开发人员每天都会遇到的常见问题。这样的问题如此普遍，所以 Rails 有一个名为 scaffold 的工具，可以生成这样的代码。如果想要这样调用 generate 而不用上述给定的参数，Rails 会为“解决方案”里给出的操作生成代码，并再加上一些：

```
$ ./script/generate scaffold Items
```

用 scaffold 开始并不意味着可以不必知道 Rails 表格如何生成，因为仍然需要明确自定义 scaffold 代码。代码中有两处是神奇之处。第一个是视图中的 text_field 调用，它在“解决方案”里已涉及了。它绑定对象成员（比如@item.name）到一个 HTML 表格控制。如果查看/ items/new 页面的源代码，会看到表格字段类似如下：

```
Name: <input type="text" name="item[name]" value="" /><br />Value <in
put type="text" name="item[value]" value="[empty]" /><br />
```

这些特殊的字段名称被第二个神奇之处使用，定位在 Item.create（在 new 里）和 Item#update_att ributes 的调用里。两种情况下，Item 对象被填入新值的散列作为其成员。该散列被嵌入到 params 散列里，它包含 CGI 格式值。

HTML 表格字段的名称（item[name]和 item[value]）翻译到 params 散列，类似如下：

```
{

  :item => {

    :name => "Name of the item",

    :value => "Value of the item"


  },

  :controller => "items",

  :action => "create"

}
```

因此这样一行代码：

```
Item.create(params[:item])
```

和下面这行代码一样高效：

```
Item.create(:name => "Name of the item", :value => "Value of the item")
```

在 edit 的操作里 Item#update_attributes 的调用几乎以相同的方式起作用。如上所述，edit 和 new 的视图很是相似，只是表格的目的地不同而已。用很少的重构操作，可以彻底移除视图文件之一。没有任何参数的<%= form_tag %>调用设置表格目的地为当前的 URL。适当地修改 new.rhtml 文件：

```
<!-- app/views/items/new.rhtml -->

<%= form_tag %>

Name: <%= text_field "item", "name" %>&#x00A;
Value: <%= text_field "item", "value" %>&#x00A;

<%= submit_tag %>

<%= end_form_tag %>
```

现在 new.rhtml 视图适合于 new 和 edit 的使用。只需改变 new 操作来调用 create 方法（因为表格不会到那里），并且改变 edit 操作来渲染 new.rhtml 而不是 edit.rhtml（它可能会被移除）：

```
class ItemsController < Appli
  cationController
   def new
   @item = Item.new

  create if request.post?

end

def edit
  @item = Item.find(params[:id])

  if request.post?

    @item.update_attributes(params[:item])

    redirect_to :action => 'edit', :id => @item.id and return
  end

  render :action => 'new'

  end
end
```

回顾 15.5 节，render 调用近几年指定会使用的模板文件。edit 里的 render 调用不会实际调用 new 方法，因此不需要担心 new 方法会覆盖@item 的值。

在现实生活里，add 和 edit 表格的内容会有很大的不同来为每个操作分离视图。然而，这些表格之间又有足够的相似性使得它们能够被重构到两个视图共享着的单一的部分视图里（查看 15.14 节）。这是一个关于 DRY（Don't Repeat Yourself）规则的很好的例子。如果 add 和 edit 视图都各自有一个单表格，它更简单也很少出错，那么倾向于在数据库方案改变时维护表格。

参考

- 15.5 节“用 render 显示模板”
- 15.14 节“重构视图为视图的部分片断”

15.17 创建 Ajax 表格问题

想要构建一个能够做出响应并且易于使用的 Web 应用程序。不会让用户耗费大量时间等待浏览器重画屏幕。

解决方案

可以使用 JavaScript 来使得浏览器的 XMLHTTPRequest 对象发送数据给服务器，而不会令用户不得不忍受普遍的（但是缓慢的）页面更新。这样的技术叫做 Ajax（注 6），Rails 使得可以很简单地使用 Ajax，而无需编写或熟悉 JavaScript。

在 Web 应用程序里实现 Ajax 之前，必须编辑应用程序的主布局模板，使得在<HEAD> 标签里调用 javascript_include_tag 方法。这和 15.15 节做出的改变是一样的：

```
<!-- app/views/layouts/application.rhtml -->

<html>
  <hea
 d>
  <title
 >My
 Web
 App<
 /title>

 <%= javascript_include_tag "prototype", "effects" %>

 </head>
 <body>
 <%= @content
_for_layout %>
 </body>
</html>
```

从 15.16 节的基础上改变应用程序，这里 new 操作是 AJAX 使能的（如果依据那一节的所有方式，并使得 edit 操作使用 new.rhtml 而不是 edit.rhtml，那么不需要那些改变并使得 edit 使用其自己的视图模板）。

从视图模板开始。编辑 app/views/items/new.rhtml 类似如下：

```
<!-- app/views/items/new.rhtml -->

<div id="show_item"></div>

<%= form_remote_tag :url => { :action => :create },
:update => "show_item",
:complete => visual_effect(:highlight, "show_item") %>
```

```
Name: <%= text_field "item", "name" %><br />

Value: <%= text_field "item", "value" %><br />

<%= submit_tag %>

<%= end_form_tag %>
```

这些细小的改动使得标准的 HTML 格式变为 Ajax 格式。主要区别在于调用 form_remote_tag 而不是 form_tag。另一个区别在于传递进方法的参数。
第一个改变是将:action 参数放进一个散列，该散列传递进:url 选项。Ajax 格式比起正常的格式有着更多的相关选项，因此像 form_tag 一样简单描述其格式操作。

注 6：这并不能完全代表 Asynchronous JavaScript 和 XML。单词 Ajax 的原型现在是计算机神话的一部分，并非只是首字母的缩写。

当用户单击提交按钮时，表格值被串行化并且发送到后台的目的操作里（本例中，为 create）。create 操作像之前一样处理表格的提交，并且返回一段 HTML。
这段 HTML 是怎么回事呢？这是:update 选项需要的。它告诉 Rails 得到表格提交的结果，并且将其放入 HTML ID 为“show_item”的元素里。这是为什么我们在模板顶部添加 标签的原因：这是服务器响应的位置。
最后 new.rhtml 视图的改变是:complete 选项。这是一个回调参数：它指定 JavaScript 代码的字符串，并会在 Ajax 请求结束时运行。我们在服务器请求出现时高亮它们。
这就是视图。也需要修改控制器里的 create 操作，使得当构建一次 Ajax 表格提交时，服务器返回一段 HTML。这就是会插入到浏览器端“show_item”元素里的片断。如果对于一次标准（非 Ajax）的表格提交，服务器的行为类似 15.16 节，发送一次 HTTP 重新定位（注 7）。控制器类看上去需要类似如下：

```
class ItemsController < Ap
  plicationController
   def new
   @item = Item.new
   end

def create

  @item = Item.create(params[:item])

   if request.xml_http_request?
    render :action => 'show', :layout
   => false
    else
    redirect_to :action => 'edit', :id =
   > @item.id
    end

end
```

```
def edit
  @item = Item.find(params[:id])

  if request.post?

    @item.update_attributes(params[:item])

    redirect_to :action => 'edit', :id => @item.id

  end

end
```

这段代码参考一个新的视图，show。这是服务器返回的小型 HTML 片断，被 Web 浏览器插入到“show_element”标签里。需要定义它：

```
<!-- app/views/items/show.rhtml -->

Your most recently created item:<br />

Name: <%= @item.name %><br />Value: <%= @item.value %><br /> <hr>
```

注 7：这会发生在有人在 JavaScript 关闭时使用应用程序的情况下。

现在当使用 http://localhost:3000/items/new 来添加新项到数据库时，不会被重新定位到 edit 操作。会仍然停留在新的页面上，表格提交的结果会显示在表格上方。这使得一次创建很多新项更简单。

讨论

15.16 节介绍了如何以传统方式提交数据给表格：用户单击“提交”按钮，浏览器发送请求到服务器，服务器返回响应页面，浏览器渲染相应页面。

最近，类似 Gmail 和 Google Maps 这样的站点普及一种无需页面刷新而发送并接受数据的技术。相同地，这些技术被称为 Ajax。Ajax 是很实用的工具可以改进应用程序的响应时间及可用性。Ajax 请求是某个应用程序操作的真实 HTTP 请求，可以像对待任何其他请求一样处理它。然而，大部分时间里，并不希望返回的是一整页 HTML，而只是数据的片断。Web 浏览器会在全部 Web 页面（早些时候服务的）内容里发送 Ajax 请求，该页面知道如何处理请求片断。

可以在一次 Ajax 请求生命周期的好几处定义 JavaScript 回调函数。一个回调函数，:complete，被用来在插入某片断到页面后高亮它。下表列出其他回调函数。

回调函数名称	回调函数描述
:loading	当 Web 浏览器设计用来加载远程文档时被调用
:loaded	当浏览器完成远程文件加载时被调用
:interactive	当用户可以与远程文档交互，即使其还没有完成加载时被调用
:success	当 XMLHttpRequest 完成，HTTP 状态代码在 2XX 范围时被调用

:failure	当 XMLHttpRequest 完成，HTTP 状态代码不在 2XX 范围时被调用
:complete	当 XMLHttpRequest 完成时被调用。如果:success 和/ 或:failure 也正运行着，那么该函数在它们之后运行

15.18 在 Web 站点上发布 Web 服务问题

想要在自己的 Web 应用程序里提供 SOAP 和 XML-RPC Web 服务。

解决方案

Rails 有一个内置的 Web 服务生成器，它使得很容易实现将控制器的操作当作 Web 服务发布。不需要花时间编写 WSDL 文件或者设置，不需要知道 SOAP 和 XML-RPC 是如何工作的。

如下是一个简单的例子。首先，按照 15.16 节的指导，创建一个名为 items 的数据库表，并且为该表生成一个模型。不要生成控制器。

现在，在命令行运行该表格：

```
./script/generate web_service Item add edit fetch
```

```
      create  app/apis/
      exists  app/controllers/
      exists  test/functional/
      create  app/apis/item_api.rb
      create  app/controllers/item_controller.rb
      create  test/functional/item_api_test.rb
```

这会创建一个 item 控制器，它支持 3 种操作：add、edit 和 fetch。但是不使用有.rhtml 视图的 Web 应用程序，这些是通过 SOAP 或 XML-RPC 访问的 Web 服务操作。

Ruby 方法不关心对象接收为参数的数据类型，或者其返回值的数据类型。但是 SOAP 或 XML-RPC Web 服务方法会关心这些。要想通过 SOAP 或 XML-RPC 接口发布 Ruby 方法，需要为其签名定义类型信息。打开文件 app/apis/item_api.rb 并对它做类似如下的编辑：

```
class ItemApi < ActionWebService::API::Base

 api_method :add, :expects => [:string, :string], :returns => [:int]

   api_method :edit, :expects => [:int, :string, :string], :returns =>
[:bool]

   api_method :fetch, :expects => [:int], :returns => [Item]
end
```

现在需要实现实际的 Web 服务界面。打开 app/controllers/item_controller.rb，并做类似如下的编辑：

```
   class ItemController < ApplicationController
     wsdl_service_name 'Item'

   def add(name, value)
     Item.create(:name => name, :value => value).id
     end
```

```
def edit(id, name, value)
  Item.find(id).update_attributes(:name => name, :value => value)
  end
```

```
def fetch(id)
  Item.find(id)
  end
 end
```

讨论

现在 item 控制器为 items 表实现了 SOAP 和 XML-RPC Web 服务。这个控制器和 items 的控制器同时存在，后者实现传统的 Web 界面（注 8）。

XML-RPC API 的 URL 是 http://www.yourserver.com/item/api，SOAP API 的 URL 是 http://www.yourserver.com/item/service.wsdl。要测试这些服务，如下是一段简短的 Ruby 脚本，它通过 SQAP 客户端调用 Web 服务器方法：

```
require 'soap/wsdlDriver'

wsdl = "http://localhost:3000/item/service.wsdl"
item_server = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver

item_id = item_server.add('foo', 'bar')
if item_server.edit(item_id, 'John', 'Doe') puts 'Hey, it worked!' else puts 'Back to the drawing board...'
end # Hey, it worked!

item = item_server.fetch(item_id) item.class
SOAP::Mapping::Object item.name item.value                    # => # => "John" # => "Doe"
```

如下是 XML-RPC 等效程序：

```
require 'xmlrpc/client'
item_server = XMLRPC::Client.new2('http://localhost:3000/item/api')

item_id = item_server.call('Add', 'foo', "bar")
```

```
if item_server.call('Edit', item_id, 'John', 'Doe')
  puts 'Hey, it worked!'

else
  puts 'Back to the drawing board...'
```

注 8：甚至可以添加 Web 界面操作到 ItemController 类里。然后单一的 controller 可以实现传统 Web 界面以及 Web 服务界面。但是无法将 Web 应用程序操作的名称定义为和 Web 服务操作相同的名字，因为 controller 类只能包含一种给定名称的方法。

```
end
# Hey, it worked!

item = item_server.call('Fetch', item_id)
# => {"name"=>"John", "id"=>2, "value"=>"Doe"}
```

参考

- Matt Biddulph 的文章“REST on Rails”描述了如何在 Rails 顶端创建 REST 类型的 Web 服务（http://www.xml.com/pub/a/2005/11/02/rest-on-rails.html）
- 16.3 节“编写 XML-RPC 客户端”，以及 16.4 节“编写 SOAP 客户端”
- 16.5 节“编写 SOAP 服务器”提出了一种 SOAP Web 服务的非 Rails 的实现

15.19 用 Rails 发送邮件问题

想要从 Rails 应用程序里发送邮件：可能是一次配置或者一个排序，或者通知某个操作已经被当作是用户行为。

解决方案

首先要做的是生成一些 mailer 基础构造。进入应用程序的基础目录，键入如下命令：

```
./script/generate mailer Notification welcome exists app/models/ create app/views/notification exists test/unit/ create test/fixtures/notification create app/models/notification.rb create test/unit/notification_test.rb create app/views/notification/welcome.rhtml create test/fixtures/notification/welcome
```

向应用程序的邮件中心提供名称“Notification”，这和 Web 界面里的控制器有某种程度的类似。设置 mailer 来生成单一的邮件，名为“welcome”：这和视图模板的一个操作很是类似。

现在打开 app/models/notification.rb，并做类似如下的编辑：

```
class Notification < ActionMailer::Base

  def welcome(user, sent_at=Time.now) @subject = 'A Friendly Welcome' @recipients = user.email @from = 'admin@mysite.com'

  @sent_on = sent_at

  @body = {
    :user => user,
    :sent_on => sent_at

  }

attachment 'text/plain' do
  |a|
   a.body = File.read('rules.txt')
   end
   end
  end
```

邮件的主题是“A Friendly Welcome”，它从地址 admin@mysite.com 处发送到用户的邮箱地址。它有一个附件，是磁盘文件 rules.txt（相对于 Rails 应用程序的根目录而言）。

虽然文件 notification.rb 在 models/ 目录下，它的功能类似于控制器，其每个邮箱消息都有相关联的视图模板。Welcome 邮件的视图在 app/views/notification/ welcome.rhtml 里，它和正常控制器的视图功能相同。

最重要的区别在于 mailer 视图不需要访问 mailer 的实例变量。要为 mailer 设置实例变量，传递这些变量的散列到 body 方法。键称为实例变量的名称，值称为它们的值。在 notification.rb 里，为 welcome 视图构造两个可用的实例变量，@user 和@sent_on。如下是视图代码本身：

```
<!-- app/views/notification/welcome.rhtml -->
```

```
Hello, <%= @user.name %>, and thanks for signing up at <%= @sent_on
%>. Please print out the attached set of rules and keep them in a
prominent place; they help keep our community running smoothly. Be
sure to pay special attention to sections II.4 ("Assignment of
Intellectual Property Rights") and XIV.21.a ("Dispute Resolution
Through Ritual Combat").
```

从 Rails 应用程序里发送 welcome 邮件，添加下述代码到控制器、模型或者观察器里均可：

```
Notification.deliver_welcome(user)
```

这里，user 变量可以是任何实现#name 和#email 的对象，这两个方法在 welcome 方法和模板里被调用。

讨论

永远不会直接调用 Notification#welcome 方法。实际上，Notification#welcome 是不可用的，因为它是一个实例方法，所以永远无法直接实例化 Notification 对象。

ActionMailer::Base 类定义了 method_missing 的实现，它查看所有未定义类方法的调用。这就是为什么即使没有定义过 deliver_welcome 但仍然可以调用它的原因。

上述给定的 welcome.rhtml 模板生成普通文本的邮件。要发送 HTML 邮件，只需简单添加下述代码到 Notification#welcome：

```
content_type 'text/html'
```

现在模板可以生成 HTML，邮件客户端会识别出邮件的格式并适当地加以渲染。

有时会想得到有关传输过程的更多的控制权，比如，当单元测试 ActionMailer 类时。不用调用 deliver_welcome 来发送邮件，可以调用 create_welcome 得到作为 Ruby 对象的邮件。这些“create”方法返回 TMail 对象，有必要时可以检查和操作它们。

如果本地 Web 服务器无法发送邮件，可以修改 environment.rb 来联系远程的 SMTP 服务器：

```
Rails::Initializer.run do |config|

  config.action_mailer.server_settings = {

    :address => 'someserver.com',

    :user_name => 'uname',

    :password => 'passwd',

    :authentication => 'cram_md5'

  }
```

```
end
```

参考

- 10.8 节“响应对未定义方法的调用”
- 14.5 节“发送邮件”中介绍了有关 ActionMailer 和 SMTP 设置方面的更多信息

15.20 自动发送错误信息到邮箱问题

想要在每次用户遇到应用程序错误时都收到一封描述性的邮件信息。

解决方案

应用程序运行中发生的错误会被发送到 ActionController::Base#log_error 方法。如果已经设置了 mailer（如 15.19 节所示），那么可以重载该方法并且使其发送邮件。代码类似如下：

```
class ApplicationController < ActionController::Base

private

  def log_error(exception)
    super
    Notification.deliver_error_message(exception,
      clean_backtrace(exception),
      session.instance_variable_get("@data"),
      params,
      request.env
    )
  end
end
```

代码收集失败发生时 Rails 请求的全方位状态信息。它捕获异常对象、相应的踪迹、会话数据、CGI 请求参数以及所有环境变量的值。

重载了的 log_error 调用 Notification.deliver_error_messsage，它假定已经创建了名为“Notification”的 mailer，并且定义了 Notification.error_message 方法。如下是实现代码：

```
class Notification < ActionMailer::Base
  def error_message(exception, trace, session, params, env, sent_on =
Time.now) @recipients = 'me@mydomain.com' @from = 'error@mydomain.com' @subject
    = "Error message: #{env['REQUEST_URI']}" @sent_on = sent_on @body = {
      :exception => exception,
      :trace => trace,
      :session => session,
      :params => params,
      :env => env
    }
  end
end
```

该邮件的模板如下：

```
<!-- app/views/notification/error_message.rhtml -->

Time: <%= Time.now %>

Message: <%= @exception.message %>

Location: <%= @env['REQUEST_URI'] %>

Action: <%= @params.delete('action') %></td></tr>

Controller: <%= @params.delete('controller') %></td></tr>

Query: <%= @env['QUERY_STRING'] %></td></tr>

Method: <%= @env['REQUEST_METHOD'] %></td></tr>

SSL: <%= @env['SERVER_PORT'].to_i == 443 ? "true" : "false" %>

Agent: <%= @env['HTTP_USER_AGENT'] %>

Backtrace
<%= @trace.to_a.join("</p>\n<p>") %>

Params

<% @params.each do |key, val| -%>
  * <%= key %>: <%= val.to_yaml %>
<% end -%>

  Session

<% @session.each do |key, val| -%>

  * <%= key %>: <%= val.to_yaml %>
<% end -%>

  Environment

<% @env.each do |key, val| -%>

  * <%= key %>: <%= val %>
<% end -%>
```

讨论

ActionController::Base#log_error 提供了根据自己喜好处理错误的灵活性。这在 Rails 应用程序工作在访问受限的主机上时尤其有效：可以要求错误发送出来，而不是写入到一个可能无法查阅的文件里。或者更愿意记录错误到数据库里，因此可以查找选用相应的模式。

ApplicationController#log_error 方法被私有调用来避免二义性。如果它不是私有的，所有控制器会认为它们已经定义了 log_error。用户能够访问/<controller>/ log_error 并使得 Rails 工作异常。

参考

- 15.19 节“用 Rails 发送邮件

15.21 文档化 Web 站点问题

想要文档化 Web 应用程序的控制器、模型和 helper，开发人员需要负责维护应用程序，使其理解该如何去工作。

解决方案

和其他任何 Ruby 程序一样，通过向代码添加特定格式的命令来文档化 Rails 应用程序。如下说明如何向 FooController 类及其方法添加文档：

```
# The FooController controller contains miscellaneous functionality
# rejected from other controllers.
class FooController < ApplicationController

  # The set_random action sets the @random_number instance variable
  # to a random number.
  def set_random

    @random_number = ra
nd*rand
  end
end
```

类和方法的文档在其声明之前出现，而不是在之后。

当已经完成应用程序文档命令的添加时，回到 Rails 应用程序根目录下并处理 rake appdoc 命令：

```
$ rake appdoc
```

这个 Rake 任务为 Rails 应用程序运行 RDoc，并且生成名为 doc/app 的目录。该目录包含一个 Web 站点，它是所有文档命令的集合，与源代码互为参考。在任何 Web 浏览器中打开 doc/app/index.rhtml，可以浏览生成的文档。

讨论

RDoc 文档可以包含 markup 和特殊的指示：可以在定义列表里描述参数，通过:nodoc: 指令从文档里隐藏类或方法。这在 17.11 节里有具体的描述。

Rails 应用程序和其他 Ruby 程序唯一的不同之处在于 Rails 有一个定义了 appdoc 任务的 Rakefile。不需要自己查找或编写它。

可能已经在方法里加入了内联命令，在操作发生时描述它们。因为 RDoc 文档包含了源代码的格式化版本，这些命令对于查阅 RDoc 的人来说是可见的。然而，这些命令被格式化成 Ruby 源代码，而不是 RDoc markup。

参考

- 17.11 节“文档化应用程序”
- 第 19 章，尤其是 19.2 节“自动生成文档”
- RDoc 的 RDoc（http://rdoc.sourceforge.net/doc/index.html）

15.22 Web 站点的单元测试问题

想要创建自动化测试套件来测试 Rails 应用程序的功能。

解决方案

Rails 无法编写比视图和控制器更多的测试代码，但是它的确使得组织和运行自动化测试变得简单。当使用./script/generate 命令创建控制器和模型时，不仅可以节省时间，而且得到了生成的针对单元和功能测试的框架。通过自己编写的功能测试来填写该框架，可以得到相当理想的测试覆盖率。

至今为止，本章的所有例子都是在 Rails 应用程序开发数据库上运行的，因此只需要确保正确设置了 config/database.yml 文件的开发部分。单元测试代码基于应用程序测试数据库运行，因此现在需要同时设置测试部分。mywebapp_test 数据库无需包含任何表，但是它必须存在而且能够被 Rails 使用。

当用 generate 脚本生成模型时，Rails 也会在 test 目录生成针对该模型的单元测试脚本。它同时创建一个 fixture、YAML 文件，包含可以加载进 mywebapp_test 数据库的测试数据。这就是单元测试基于运行的数据：

```
./script/generate model User
```

```
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/user.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
create  db/migrate
create  db/migrate/001_create_users.rb
```

当用 generate 创建一个控制器时，Rails 为控制器创建了功能测试脚本：

```
./script/generate users list

exists app/controllers/

exists app/helpers/

create app/views/users

exists test/functional/

create app/controllers/users_controller.rb

create test/functional/users_controller_test.rb

create app/helpers/users_helper.rb create app/views/users/list.rhtml
```

因为在模型和控制器类里编写代码，需要在这些文件里编写相应的测试。

要运行单元和功能测试，在主目录下调用 rake 命令。默认的 Rake 任务会运行所有测试。如果在生成测试文件之后立即运行它，会产生如下效果：

```
$ rake
(in /home/lucas/mywebapp)
/usr/bin/ruby1.8 "test/unit/user_test.rb"
Started
.
Finished in 0.048702 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
/usr/bin/ruby1.8 "test/functional/users_controller_test.rb"
Started
.
Finished in 0.024615 seconds.

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

讨论

所有用其他语言或在其他 Ruby 程序里编写的单元测试知识都可以应用到 Rails 里。Rails 为我们完成一些计算，并且它定义了一些很是实用的断言（如下所示），但是仍然需要我们自己完成一些工作。回报也是相同的：可以自信地修改和重构代码，因为知道如果什么中断了，那么测试也会中断。可以立即知道问题所在并且可以快速修改。

一起看看 Rails 生成了什么。如下是生成的 test/unit/user_test.rb：

```
require File.dirname(__FILE_ _) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  # Replace this with
your real tests.
  def test_truth
  assert true
  end
end
```

不错的开始，但是 test_truth 有些同义反复。如下是更理性的测试：

```
class UserTest
    def test_first
    assert_kind_of Use
   r, users(:first)
    end
  end
```

该代码从 users 表得到第一个元素，并且断言 ActiveRecord 将其转变为 User 对象。这样测试 User 代码（没有编写任何代码）并没有测试 Rails 和 ActiveRecord 效果好，但是它显示了针对良好单元测试的断言。

但是 users(:first)是如何返回什么的呢？测试套件在 mywebapp_test 数据库上运行，甚至没有在其中添加任何表，以及少许示例数据。

我们没有，但是 Rails 有。当运行测试套件时，Rails 复制开发数据库方案到测试数据库中。不用针对存在于开发数据库的每一个数据都运行测试，Rails 从名为 fixtures 的 YAML 文件里加载特殊的测试数据。fixture 文件包含需要测试的所有数据库数据：被测试删除而存在的对象、不同表行间的特殊关系、或者其他任何需要的东西。

上述示例中，users 表的 fixture 被 fixtures :users 这一行代码加载。如下是为 User 模型生成的 fixture，在 test/fixtures/users.yml 里：

```
first:
  id: 1
another:
  id: 2
```

在运行单元测试之前，Rails 阅读该文件，在 users 表里创建两行，并且为它们定义别名（:first 和:another），因此可以在单元测试里引用它们。然后定义 users 方法（正如其他所示，该方法名称基于模型的名称）。在 test_first 里，users(:first) 的调用找回 User 对象，它对应 fxture 里的:first：ID 1 的那个对象。

如下是另一个单元测试：

```
class UserTest

  def test_another
    assert_kind_of User, users(:another)
    assert_equal 2, users(:another).id
    assert_not_equal users(:first), users(:another)
  end
end
```

Rails 添加如下 Rails 特定的断言到 Ruby 的 Test::Unit：

- assert_dom_equal
- assert_dom_not_equal
- assert_generates
- assert_no_tag

- assert_recognizes
- assert_redirected_to
- assert_response
- assert_routing
- assert_tag
- assert_template
- assert_valid

参考

•“测试 Rails”是 Rails 单元和功能测试的指导（http://manuals.rubyonrails.com/read/ book/5）

•Rails 1.1 通用支持集成测试，针对 controller 和操作之间交互的测试，查看 http:// rubyonrails.com/rails/classes/ActionController/IntegrationTest.html 和 http:// jamis.jamisbuck.org/articles/2006/03/09/integration-testing-in-rails-1-1

•ZenTest 库包含 Test::Rails，它使得可以为视图和 controller 编写单独的测试（http:// rubyforge.org/projects/zentest/）

•在 http://ar.rubyonrails.org/classes/Fixtures.html 阅读有关 fixture 的信息

•在 http://rails.rubyonrails.com/classes/Test/Unit/Assertions.html 阅读有关 assertions that Rails adds to Test::Unit 的信息

•15.6 节“集成数据库到 Rails 应用程序中”

•17.7 节“编写单元测试”

•第 19 章

15.23 在 Web 应用程序中使用断点问题

Rails 应用程序有无法通过记录消息找到的 bug。需要一种耐受力强的调试工具，可以内部审查应用程序在任何给定点的全部状态。

解决方案

Breakpoint 库使得可以停止代码流程并且进入 irb，一个交互的 Ruby 会话。在 irb 里，可以内部审查当前作用域中的本地变量，修改这些变量，并且继续代码正常流程的执行。如果曾经耗费很长时间试图通过在某处放置记录消息来跟踪 bug ，会发现 breakpoint 使得一切变得简单并且可以更直观地进行调试。

但是如何从 Web 应用程序里运行一个交互控制台程序呢？答案是必须事先运行该控制台程序，侦听 Rails 服务器的调用。

第一步是从命令行运行./script/breakpointer。这个命令启动服务器，它侦听网络上的 Rails 服务器的 breakpoint 调用。在终端窗口一直运行该程序：这就是 irb 会话开启之处：

```
$ ./script/breakpointer
No connection to breakpoint service at druby://localhost:42531
Tries to connect will be made every 2 seconds...
```

要触发一次 irb 会话，可以在 Rails 中的任何地方调用 breakpoint 方法，在模型、控制器或者帮助器方法里。当执行到这一点时，进入的客户端请求的处理会停止，irb 会话会在终端启动。当退出会话时，请求处理会继续。

讨论

有一个例子。假定已经编写了如下控制器，在修改 Item 对象名称属性时遇到困难。

```
class ItemsController < ApplicationController

  def update
    @item = Item.find(params[:id])
    @item.value = '[default]'
    @item.name = params[:name]
    @item.save
    render :text => 'Saved'
  end
end
```

可以在 Item 类中放置一个 breakpoint 调用，如下：

```
class Item < ActiveRecord::Base
  attr_accessor :name, :value
```

```
    def name=(name)
      super
      breakpoint
    end
end
```

访问 URL http://localhost:3000/items/update/123?name=Foo 调用 ItemController#update，它发现 Item 号码 123 并且随后调用其 name=方法。name=的调用触发断点。不用翻译文本“Saved”，站点挂起并且不响应请求。但是如果返回终端运行 breakpointer 服务器，可以看到已经启动了一个交互的 Ruby 会话。该会话允许在断点调用处处理所有本地变量和方法：

```
Executing break point "Item#name=" at item.rb:4 in `name='

irb:001:0> local_variables
=> ["name", "value", "_", "_ _"]

irb:002:0> [name, value]
=> ["Foo", "[default]"]

irb:003:0> [@name, @value]
=> ["Foo", "[default]"]

irb:004:0> self
=> #<Item:0x292fbe8 @name="Foo", @value="[default]">

irb:005:0> self.value = "Bar"
=> "Bar"

irb:006:0> save
=> true

irb:006:0> exit
Server exited. Closing connection...
```

一旦结束了，键入 exit 中止交互的 Ruby 会话。Rails 应用程序从刚才断开处继续运行，按照预期翻译“Saved”。

默认来说，断点是根据其所在的方法命名的。可以传递一个字符串进 breakpoint，得到一个更具描述性的名称。这在某个方法中包含多个断点时很有用：

```
breakpoint "Trying to set Item#name, just called super"
```

不必直接调用 breakpoint，也可以调用 assert，该方法得到一个代码块。如果该块计算后得到 false，Ruby 调用 breakpoint，否则继续正常的流程。使用 assert 使得可以设置这样的断点，它们只在运行有误时才被调用（在传统的调试器里，这是“条件断点”）：

```
1.upto 10 do |i|

  assert { Person.find(i) }

  p = Person.find(i)

  p.update_attribute(:name, 'Lucas')

end
```

如果所有要求的 Person 对象都被找出，breakpoint 不会被调用，因为 Person.find 会一直返回 true。如果有一个 Person 对象丢失了，那么 Ruby 调用 breakpoint 方法，会得到有待调查的 irb 会话。

断点是功能强大的工具，能够可观地简化调试过程。要理解其真正的功能很难，除非亲身体会，因此用自己的代码来验证这个解决方案吧。

参考

- 17.10 节的“使用断点审查并改变应用程序的状态”中更详细地讲解了有关断点的更多知识
- http://wiki.rubyonrails.com/rails/show/HowtoDebugWithBreakpoint