

原书第 3 版 · 涵盖 Rails 4.2

Ruby on Rails 教程

Ruby on Rails Tutorial 中文版



原书第 3 版

Ruby on Rails 教程

通过 Rails 学习 Web 开发

Michael Hartl 著

安道 译

样 章

目录

致中国读者

序

致谢

作者译者简介

版权和代码授权协议

第 1 章 从零开始，完成一次部署	1
1.1. 简介	2
1.2. 搭建环境	5
1.3. 第一个应用	7
1.4. 使用 Git 做版本控制	20
1.5. 部署	29
1.6. 小结	33
1.7. 练习	33
第 2 章 玩具应用	35
2.1. 规划应用	35
2.2. 用户资源	38
2.3. 微博资源	49
2.4. 小结	58
2.5. 练习	59
第 3 章 基本静态的页面	63
3.1. 创建演示应用	63
3.2. 静态页面	65
3.3. 开始测试	73
3.4. 有点动态内容的页面	78
3.5. 小结	87
3.6. 练习	88
3.7. 高级测试技术	90
第 4 章 Rails 背后的 Ruby	97
4.1. 导言	97
4.2. 字符串和方法	100

4.3. 其他数据类型	106
4.4. Ruby 类	115
4.5. 小结	123
4.6. 练习	123

致中国读者

Ruby 是一门优美的计算机语言，其设计原则是“让编程人员快乐”。David Heinemeier Hansson 就是看重了这一点，才在开发 Rails 框架时选择了 Ruby。Rails 常被称作 Ruby on Rails，它让 Web 开发变得从未这么快速，也从未这么简单。在过去的几年中，《Ruby on Rails Tutorial》这本书被视为介绍使用 Rails 进行 Web 开发的先驱者。

在这个全球互联的世界中，计算机编程和 Web 应用开发都在迅猛发展，我很期待能为中国的开发者提供 Ruby on Rails 培训。学习英语这门世界语言是很重要的，但先通过母语学习往往会有更有效果。正因为这样，当看到安道把《Ruby on Rails Tutorial》翻译成中文时，我很高兴。

我从未到过中国，但一定会在未来的某一天到访。希望我到中国时能见到本书的一些读者！

衷心的祝福你们，

Michael Hartl
《Ruby on Rails Tutorial》的作者

附原文：

Ruby is a delightful computer language explicitly designed to make programmers happy. This philosophy influenced David Heinemeier Hansson to pick Ruby when implementing the Rails web framework. Ruby on Rails, as it's often called, makes building custom web applications faster and easier than ever before. In the past few years, the Ruby on Rails Tutorial has become the leading introduction to web development with Rails.

In our interconnected world, computer programming and web application development are rapidly rising in importance, and I am excited to support Ruby on Rails in China. Although it is important to learn English, which is the international language of programming, it's often helpful at first to learn in your native language. It is for this reason that I am grateful to Andor Chen for producing the Chinese-language edition of the Ruby on Rails Tutorial book.

I've never been to China, but I definitely plan to visit some day. I hope I'll have the chance to meet some of you when I do!

Best wishes and good luck,

Michael Hartl
Author
The Ruby on Rails Tutorial

序

我之前工作的公司（CD Baby）是大张旗鼓转用 Ruby on Rails 最早的企业之一，然后又更加惹眼地换回了 PHP（在 Google 中搜索我的名字，能搜到关于这场闹剧的文章）。很多人都强烈推荐 Michael Hartl 的这本书，所以我不得不读一下，读完《Ruby on Rails Tutorial》后，我又开始使用 Rails 做开发了。

我读过很多 Rails 相关的书，但是这本真正让我入门了。书里的一切都很符合“Rails 之道”，我以前觉得这个“道”很不自然，但是读完这本书，却感觉自然无比。本书也是唯一一本自始至终都使用“测试驱动开发”（Test-driven Development，简称 TDD）理念的 Rails 书籍。很多行家都推荐使用 TDD，但是在本书出版之前从没有人如此清楚地介绍过这个理念。书中的演示应用还用到了 Git、Bitbucket 和 Heroku，作者真是让你体验了一把开发真正能用的应用是什么感觉，而且书中用到的代码并不是凭空捏造出来的。

线性叙述是很好的模式。我花了三天的时间¹ 阅读书，完成了书中所有的演示应用，也做了全部练习。从头至尾，循序渐进，不要跳着读，这样才能从中受益。

享受这本书吧！

Derek Sivers (sivers.org)
CD Baby 创始人

1. 这可不常见，读完整本书花的时间往往比三天长很多。

致谢

《Ruby on Rails 教程》很大程度上归功于我以前写的一本书——RailsSpace，因此以前的合著人 [Aurelius Prochazka](#) 也有很大功劳。我要感谢 Aure，他不仅为前一本书做出了贡献，而且也给予了这本书支持。我还要感谢 Debra Williams Cauley，她是 RailsSpace 和这本书的编辑，只要她还带我去玩棒球，我就会继续为她写书。

我要感谢很多 Ruby 高手，在过去这些年，他们教我知识，也给我启迪。他们是：David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Mark Bates, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Pratik Naik, Sarah Mei, Sarah Allen, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, Pivotal Labs 公司的好心人们，Heroku 团队，thoughtbot 公司的小伙伴，以及 GitHub 的全体员工。最后，还有很多很多读者（太多了，无法一一列举）在本书写作过程中反馈了众多问题，还给了我很多建议，我由衷地感谢这些人的帮助，以及努力让这本书变得更好。

作者译者简介

本书英文版原作者是 [Michael Hartl](#)，把 Ruby on Rails Web 开发介绍给世人的先行者之一，也是自出版平台 [SoftCover](#) 的联合创始人。他之前曾经写作并开发了 [RailsSpace](#)，一本很过时的 Rails 教程；也曾使用 Ruby on Rails 开发过一个名为 [Insoshi](#) 的社交网络平台，这个平台曾经很流行，现在已经过气了。因为他对 Ruby 社区的贡献，于 2011 年被授予了 [Ruby Hero 奖](#)。他毕业于哈佛学院，并获得了[加州理工学院的物理学博士学位](#)。他还是 [Y Combinator](#) 创业者项目的毕业生。

本书简体中文版由[安道](#)翻译。他是一名翻译爱好者，一直在学习使用 Ruby，已经翻译多本 Rails 相关的书籍，例如《[使用 RSpec 测试 Rails 程序](#)》和《[Rails 程序部署之道](#)》等。

版权和代码授权协议

本书是《Ruby on Rails Tutorial: Learn Web Development with Rails (Third Edition)》一书的简体中文版，由作者 Michael Hartl 授权安道翻译和销售。版权归 Michael Hartl 和安道所有。

本书受版权法保护，任何组织或个人不得以任何形式分发或做商业使用。

书中代码基于 [MIT 协议](#) 和 [Beerware 协议](#) 发布。

The MIT License

Copyright (c) 2014 Michael Hartl

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
/*
 * -----
 * "THE BEERWARE LICENSE" (Revision 43):
 * Michael Hartl wrote this code. As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.
 * -----
 */
```


第 1 章 从零开始，完成一次部署

欢迎阅读《Ruby on Rails 教程：通过 Rails 学习 Web 开发》。本书的目的是教你如何开发 Web 应用，而我们选择的工具是流行的 Ruby on Rails Web 框架。如果你刚接触这一领域，本书会向你详细介绍 Web 应用开发的方方面面，包括 Ruby、Rails、HTML 和 CSS、数据库、版本控制、测试，以及部署的基本知识。学会这些知识足够为你赢得一份 Web 开发者的工作，或者还可以让你成为一名技术创业者。如果你已经了解 Web 开发，阅读本书能快速学会 Rails 框架的基础，包括 MVC 和 REST、生成器、迁移、路由，以及嵌入式 Ruby。不管怎样，读完本书之后，以你所掌握的知识，已经能够阅读讨论更高级话题的图书和博客，或者观看视频。这些都是旺盛的编程教学生态圈的一部分。¹

本书采用一种综合式方法讲授 Web 开发，在学习的过程中我们要开发三个演示应用：第一个最简单，叫 `hello_app`（[1.3 节](#)）；第二个功能多一些，叫 `toy_app`（[第 2 章](#)），第三个是真正的演示程序，叫 `sample_app`（[第 3 章到第 12 章](#)）。从这三个应用的名字可以看出，书中开发的应用不限定于某种特定类型的网站。不过，最后一个演示应用有点儿类似某个流行的[社会化微博网站](#)（很巧，这个网站一开始也是使用 Rails 开发的）。本书的重点是介绍通用原则，所以不管你想开发什么样的 Web 应用，读完本书后，都能建立扎实的基础。

人们经常会问，我要具备多少背景知识才能阅读本书学习 Web 开发。[1.1.1 节](#)对此做了详细分析。Web 开发是个具有挑战性的学科，对没有任何背景知识的初学者来说挑战更大。我最初为本书设定的阅读对象是已经具有一定编程和 Web 开发经验的开发者，但后来发现读者中有很多都刚开始接触开发。所以，现在你看到的本书第三版做出了很多努力，尽量降低了入门 Rails 的门槛。

旁注 1.1：降低门槛

本书第三版采取了很多措施，降低入门 Rails 的门槛：

- 使用云端标准的开发环境（[1.2 节](#)），规避了安装和配置新系统涉及到的很多问题；
- 合理利用 Rails 默认提供的组件，例如原生的 MiniTest 测试框架；
- 删掉了很多外部依赖件（RSpec，Cucumber，Capybara，Factory Girl）；
- 使用一种更轻量级、更灵活的测试方式；
- 延后介绍，或者删除了较为复杂的配置选项（Guard，Spork，RubyTest）；
- 弱化某个 Rails 版本特有的功能，更加强调 Web 开发的通用原则。

我希望这些变化能让本书第三版获得比前一版更多的读者。

这第一章，我们要安装 Ruby on Rails 以及需要的所有软件，而且还要架设开发环境（[1.2 节](#)）。然后创建第一个 Rails 应用，`hello_app`。本书旨在介绍优秀的软件开发习惯，所以在创建第一个应用之后，我们会立即

1. 本书最新版可以在本书网站上获取，地址是 <http://railstutorial-china.org>。如果你看的是离线版，一定要访问在线版，获取最近的更新。

将它纳入版本控制系统 Git 中（[1.4 节](#)）。你可能不相信，在这一章，我们还要部署这个应用（[1.5 节](#)），把它放到外网上。

[第 2 章](#)会创建第二个项目，演示 Rails 应用的一些基本操作。为了速度，我们会使用脚手架（[旁注 1.2](#)）创建这个应用（名为 `toy_app`）。因为生成的代码很丑也很复杂，所以[第 2 章](#)将集中精力在浏览器中，使用 URI（经常称为 URL）² 和这个应用交互。

本书剩下的章节将集中精力开发一个真实的大型演示应用（名为 `sample_app`），所有代码都从零开始编写。在开发这个应用的过程中，我们会用到模拟技术，“测试驱动开发”（Test-driven Development，简称 TDD）理念，以及“集成测试”（integration test）。[第 3 章](#)创建静态页面，然后增加一些动态内容。[第 4 章](#)会简要介绍一下 Rails 使用的 Ruby 程序语言。[第 5 章](#)到[第 10 章](#)将逐步完善这个应用的低层结构，包括网站的布局，用户数据模型，完整的注册和认证系统（含有账户激活和密码重设功能）。最后，[第 11 章](#)和[第 12 章](#)将添加微博和社交功能，最终开发出一个可以正常运行的演示网站。

旁注 1.2：脚手架——更快，更简单，更诱人

Rails 出现伊始就吸引了众多目光，特别是 Rails 创始人 David Heinemeier Hansson 录制的著名的“[15分钟开发一个博客程序](#)”视频。这个视频及其衍生版本是窥探 Rails 强大功能一种很好的方式，我推荐你看一下这些视频。不过事先提醒一下，这些视频中的演示能控制在 15 分钟以内，得益于一种叫做“脚手架”（scaffold）的功能，通过 Rails 命令 `generate scaffold` 生成大量的代码。

写作本书时，我也想过使用脚手架，因为它[更快、更简单、更诱人](#)。不过脚手架生成的大量且复杂的代码会让初学者困惑。虽然能学会脚手架的用法，但并不明白到底发生了什么事。使用脚手架，你只是一个脚本生成器的使用者，无法提升你对 Rails 的认识。

本书将采用一种不同的方式，虽然[第 2 章](#)会用脚手架开发一个小型的玩具应用，但本书的核心是从[第 3 章](#)起开发的演示应用。在开发这个演示应用的每个阶段，我们只会编写少量的代码，易于理解但又具有一定的挑战性。通过这一过程，最终你会对 Rails 有较为深刻的理解，而且能灵活运用，开发几乎任何类型的 Web 应用。

1.1. 简介

Ruby on Rails（或者简称“Rails”）是一个 Web 开发框架，使用 Ruby 编程语言开发。自 2004 年出现之后，Rails 就迅速成为动态 Web 应用开发领域功能最强大、最受欢迎的框架之一。使用 Rails 的公司有很多，例如 [Airbnb](#)、[Basecamp](#)、[Disney](#)、[Github](#)、[Hulu](#)、[Kickstarter](#)、[Shopify](#)、[Twitter](#) 和 [Yellow Pages](#)。还有很多 Web 开发工作室专门从事 Rails 应用开发，例如 [ENTP](#)、[thoughtbot](#)、[Pivotal Labs](#)、[Hashrocket](#) 和 [HappyFunCorp](#)。除此之外还有无数独立顾问，培训人员和项目承包商。

Rails 为何如此成功呢？首先，Rails 完全开源，基于 [MIT 协议](#) 发布，可以免费下载、使用。Rails 的成功很大程度上得益于它优雅而紧凑的设计。Rails 熟谙 Ruby 语言的可扩展性，开发了一套用于编写 Web 应用的“[领域特定语言](#)”（Domain-specific Language，简称 DSL）。所以 Web 编程中很多常见的任务，例如生成 HTML，创建数据模型和 URL 路由，在 Rails 中都很容易实现，最终得到的应用代码简洁而且可读性高。

Rails 还会快速跟进 Web 开发领域最新的技术和框架设计方式。例如，Rails 是最早使用 REST 架构风格组织 Web 应用的框架之一（这个架构贯穿本书）。当其他框架开发出成功的新技术后，Rails 的创建者 [David](#)

2. URI 是“统一资源标识符”（Uniform Resource Identifier）的简称，较少使用的 URL 是“统一资源定位符”（Uniform Resource Locator）的简称。在实际使用中，URL 一般和浏览器地址栏中的内容一样。

Heinemeier Hansson 及其核心开发团队会毫不犹豫的将其吸纳进来。或许最典型的例子是 Rails 和 Merb 两个项目的合并，从此 Rails 继承了 Merb 的模块化设计、稳定的 API，性能也得到了提升。

最后一点，Rails 有一个活跃而多元化的社区。社区中有数以百计的开源项目贡献者，以及与会者众多的开发者大会，而且还开发了大量的 gem（代码库，一个 gem 解决一个特定的问题，例如分页和图片上传），有很多内容丰富的博客，以及一些讨论组和 IRC 频道。有如此众多的 Rails 程序员也使得处理程序错误变得简单了：在谷歌中搜索错误消息，几乎总能找到一篇相关的博客文章或讨论组中的话题。

1.1.1. 预备知识

阅读本书不需要具备特定的预备知识。本书不仅介绍 Rails，还涉及底层的 Ruby 语言，Rails 默认使用的测试框架（MiniTest），Unix 命令行，HTML、CSS，少量的 JavaScript，以及一点 SQL。我们要掌握的知识很多，所以我一般建议阅读本书之前先具备一些 HTML 和编程知识。说是这么说，但也有相当数量的初学者使用本书从零开始学习 Web 开发，所以即便你的经验有限，我还是建议你读一下试试。如果你招架不住了，随时可以翻回这里，使用下面列出的某个资源，从头学起。多位读者告诉我，他们建议跟着教程做两遍，第一遍毕竟学到的知识有限，但再做第二遍就简单多了。

学习 Rails 时经常有人问，要不要先学 Ruby。这个问题的答案取决于你个人的学习方式以及编程经验。如果你希望较为系统地彻底学习，或者你以前从未编程过，那么先学 Ruby 或许更合适。学习 Ruby，我推荐阅读 Chris Pine 写的《Learn to Program》和 Peter Cooper 写的《Ruby 入门》。很多 Rails 初学者很想立即开始开发 Web 应用，而不是在此之前读完一本介绍 Ruby 的书。如果你是这类人，我推荐你在 Try Ruby 上学习一些简短的交互式教程，以便在阅读本书之前对 Ruby 有个大概的了解。如果你还是觉得本书太难，或许可以先看 Daniel Kehoe 写的《Learn Ruby on Rails》，或者学习 One Month Rails 课程——它们更适合没有任何背景知识的初学者。

不管你从哪里开始，读完本书后都应该可以学习 Rails 中高级知识了。以下是我推荐的一些学习资源：

- [Code School](#)：很好的交互式编程课程；
- [Tealeaf Academy](#)：很好的在线 Rails 开发训练营（包含高级知识）；
- [Thinkful](#)：在线课程，和本书的难度差不多；
- Ryan Bates 主持的 [RailsCasts](#)：优秀的 Rails 视频（大多数免费）；
- [RailsApps](#)：很多针对特定话题的 Rails 项目和教程，说明详细；
- [Rails 指南](#)：按话题编写的 Rails 参考，经常更新。³

1.1.2. 排版约定

本书使用的排版方式，很多都不用再做解释。本节我要说一下那些意义不是很清晰的排版。

书中很多代码清单用到了命令行命令。为了行文简便，所有命令都使用 Unix 风格命令行提示符（一个美元符号），例如：

```
$ echo "hello, world!"  
hello, world!
```

3. 译者注：Rails 指南已由本书译者翻译成中文，电子书购买地址：<https://selfstore.io/products/13>。

在 1.2 节我会提到，不管你使用哪种操作系统（尤其是 Windows），我都建议使用云端开发环境（1.2.1 节），这种环境都内置了 Unix（Linux）命令行。命令行十分有用，因为 Rails 提供了很多可以在命令行中运行的命令。例如，在 1.3.2 节中，我们会使用 `rails server` 命令启动本地的 Web 开发服务器：

```
$ rails server
```

和命令行提示符一样，本书也会使用 Unix 惯用的目录分隔符（即斜线 /）。例如，演示应用中的配置文件 `production.rb`，它的路径是：

```
config/environments/production.rb
```

这个文件路径相对于应用的根目录。在不同的系统中，根目录会有差别。在云端 IDE 中，根目录像下面这样：

```
/home/ubuntu/workspace/sample_app/
```

所以，`production.rb` 的完整路径是：

```
/home/ubuntu/workspace/sample_app/config/environments/production.rb
```

为了行文简洁，我一般都会省略应用的路径，写成 `config/environments/production.rb`。

本书经常需要显示一些来自其他程序（shell 命令，版本控制系统，Ruby 程序等）的输出。因为系统之间存在细微的差异，你看到的输出结果可能和书中显示的不完全一致，但是无需担心。而且，有些命令在某些操作系统中可能会导致错误，本书不会一一说明这些错误的解决方法，你可以在谷歌中搜索错误消息，自己尝试解决——这也是为现实中的软件开发做准备。如果你在阅读本书的过程中遇到了问题，我建议你看一下[本书网站帮助区](#)中列出的资源。

在这个教程中我们要测试 Rails 应用，所以最好知道某段代码能让测试组件失败（使用红色表示）还是通过（使用绿色表示）。为了方便，导致测试失败的代码使用“RED”标记，能让测试通过的代码使用“GREEN”标记。

每一章都有一些练习，你可以自己决定要不要做，但推荐做。为了区分正文和练习，练习的解答不会和后续的内容混在一起。如果后面需要使用某个练习中的代码，我会在正文中指出来，并给出解答方法。

最后，为了方便，本书使用两种排版方式，让代码清单更易理解。第一种，有些代码清单中包含一到多个高亮的代码行，如下所示：

```
class User < ActiveRecord::Base
  validates :name, presence: true
  validates :email, presence: true
end
```

高亮的代码行一般用于标出这段代码清单中最新的新代码，偶尔也用来表示当前代码清单和前一个代码清单的差异。第二种，为了行文简洁，书中很多代码清单中都有竖排的点号，如下所示：

```
class User < ActiveRecord::Base
  .
  .
  .
  has_secure_password
end
```

这些点号表示省略的代码，不要直接复制。

1.2. 搭建环境

就算对经验丰富的 Rails 开发者来说，安装 Ruby、Rails，以及相关的所有软件，也要几经波折。这些问题是由环境的多样性导致的。不同的操作系统，版本号，文本编辑器的偏好设置和“集成开发环境”（Integrated Development Environment，简称 IDE）等，都会导致环境有所不同。如果你已经在本地电脑中配置好了开发环境，可以继续使用你的环境。但对于初学者，我更鼓励使用云端集成开发环境（[旁注 1.1](#) 中说过），这样可以避免安装和配置出现问题。云端 IDE 运行在普通的 Web 浏览器中，因此在不同的平台中表现一致，这对 Rails 开发一直很困难的操作系统（例如 Windows）来说尤其有用。如果你不怕挑战，仍想在本地开发环境中学习书中的教程，我建议你按照 [InstallRails.com](#) 中的说明搭建环境。⁴

1.2.1. 开发环境

不同的人有不同的喜好，每个 Rails 程序员都有一套自己的开发环境。为了避免问题复杂化，本书使用一个标准的云端开发环境，Cloud9。而且，为了第三版我还和 Cloud9 合作，专为本书量身打造了一个开发环境。这个开发环境预先安装好了 Rails 开发所需的大多数软件，包括 Ruby、RubyGems 和 Git（其实，唯有 Rails 要单独安装，而且这么做是有目的的，详情参见 [1.2.2 节](#)）。这个云端 IDE 还包含 Web 应用开发所需的三个基本组件：文本编辑器，文件系统浏览器，以及命令行终端（如图 1.1）。云端 IDE 中的文本编辑器功能很多，其中一项是“在文件中查找”的全局搜索功能⁵，我觉得这个功能对大型 Rails 项目来说是必备的。

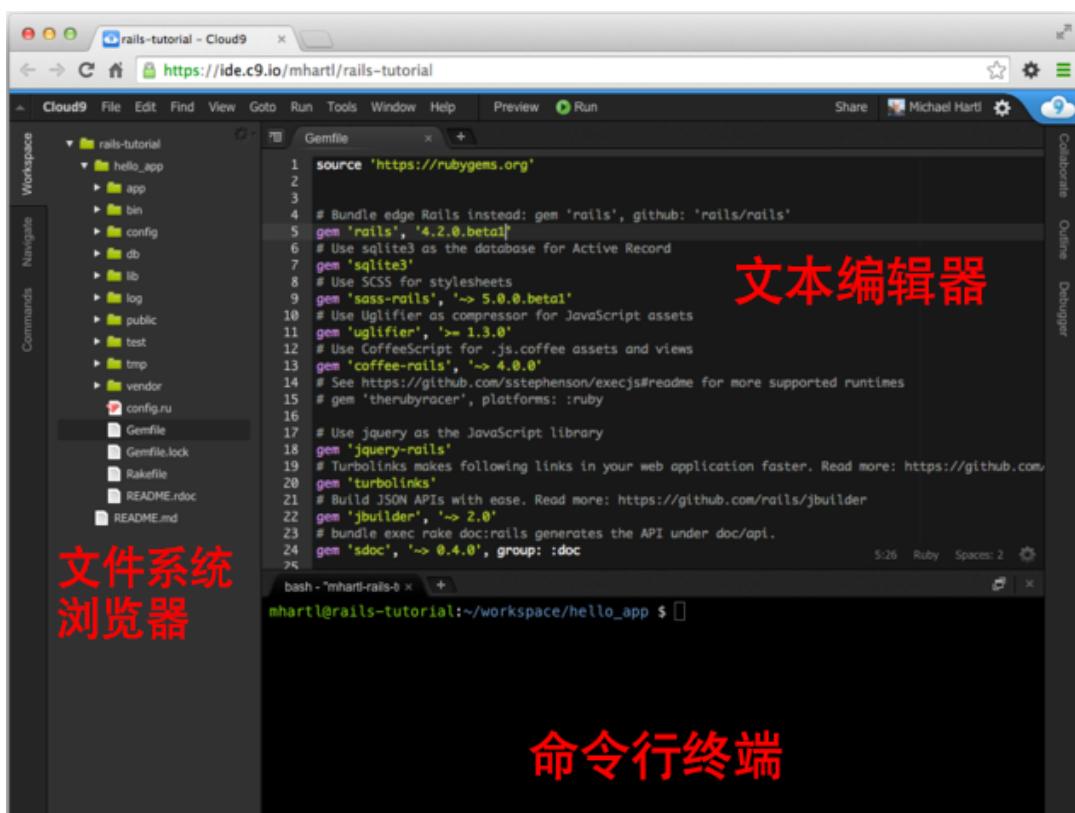


图 1.1：云端 IDE 的界面布局

这个云端开发环境的使用步骤如下：

4. 就算这样，我还是要提醒 Windows 用户，InstallRails 推荐使用的 Rails 安装包往往版本过旧，可能跟本书的教程不兼容。
5. 例如，要想找到 `foo` 函数的定义体，可以全局搜索“`def foo`”。

1. 在 Cloud9 中[注册一个免费账户](#)；
2. 点击“Go to your Dashboard”（进入控制台）；
3. 选择“Create New Workspace”（新建工作空间）；
4. 创建一个名为“rails-tutorial”（不是“rails_tutorial”）的工作空间，勾选“Private to the people I invite”（仅对我邀请的人开放），然后选择表示 Rails 教程的图标（不是表示 Ruby on Rails 那个图标），如图 1.2 所示。；
5. 点击“Create”（创建）；
6. Cloud9 配置工作空间完成后，选择这个工作空间，然后点击“Start editing”（开始编辑）。

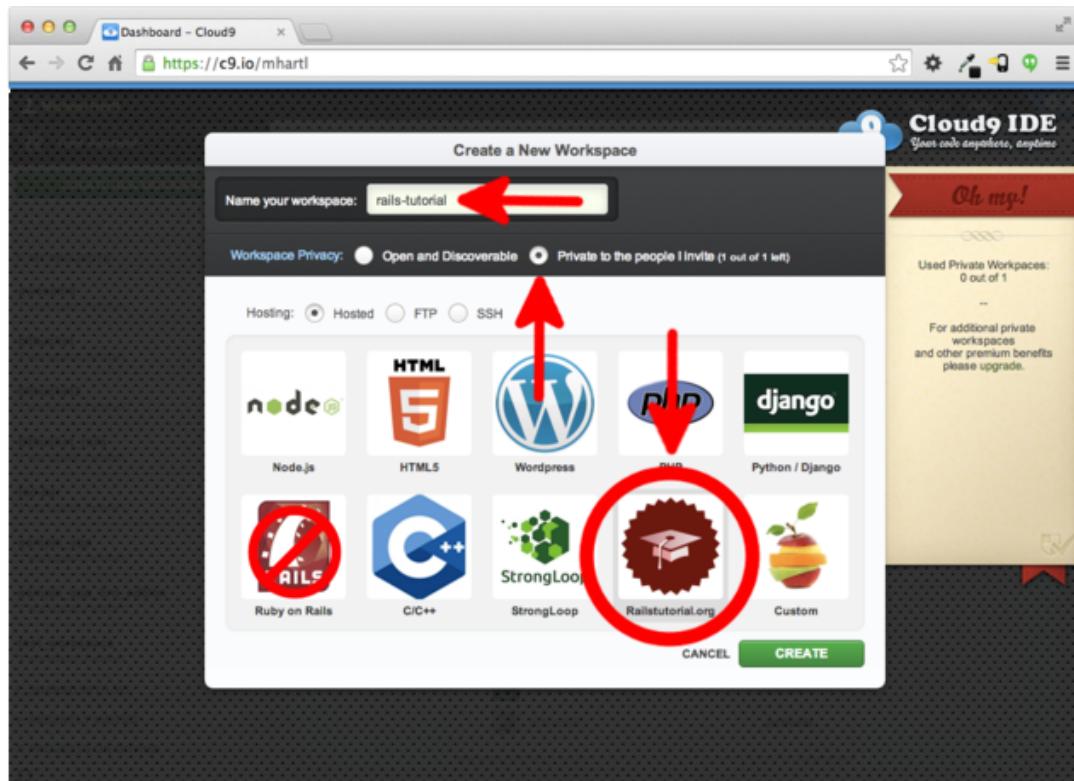


图 1.2：在 Cloud9 中新建一个工作空间

因为使用两个空格缩进几乎是 Ruby 圈通用的约定，所以我建议你修改编辑器的配置，把默认的四个空格改为两个。配置方法是，点击右上角的齿轮图标，然后选择“Code Editor (Ace)”（Ace 代码编辑器），编辑“Soft Tabs”（软制表符）设置，如图 1.3 所示。（注意，修改设置后立即生效，无需点击“Save”按钮。）

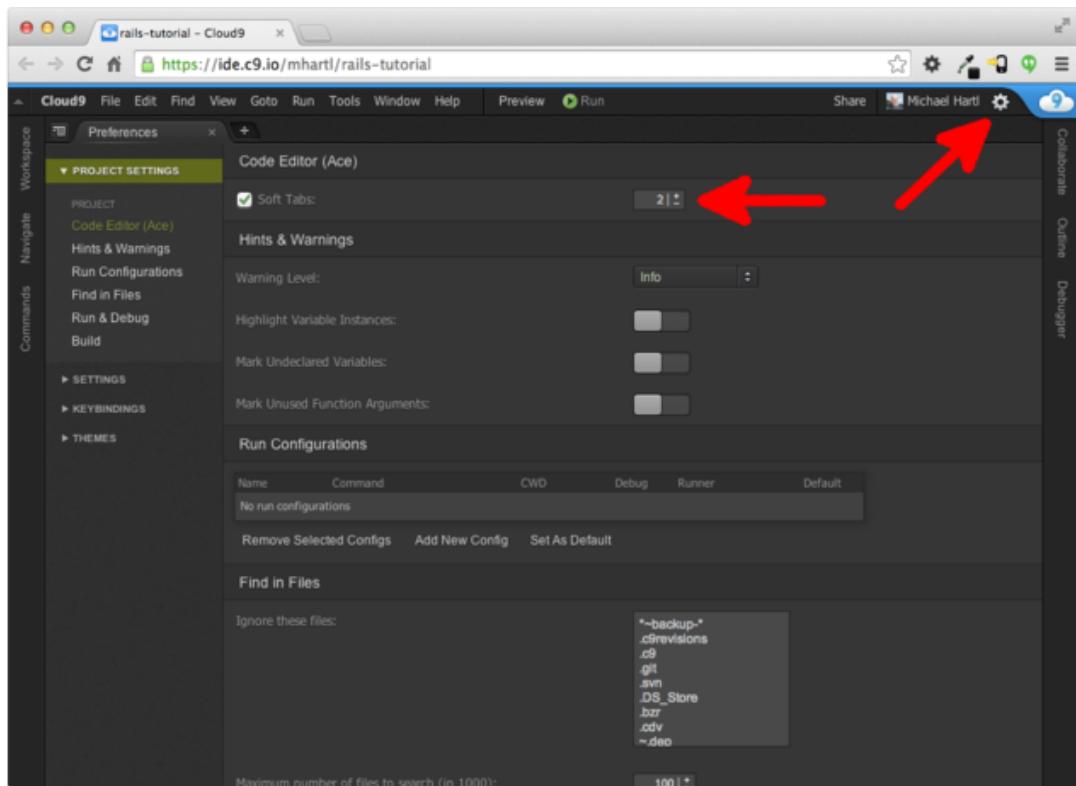


图 1.3：让 Cloud9 使用两个空格缩进

1.2.2. 安装 Rails

前一节创建的开发环境包含所有软件，但没有 Rails。⁶为了安装 Rails，我们要使用包管理器 RubyGems 提供的 `gem` 命令，在命令行终端里输入代码清单 1.1 所示的命令。（如果在本地系统中开发，在终端窗口中输入这个命令；如果使用云端 IDE，在图 1.1 中的“命令行终端”输入这个命令。）

代码清单 1.1：安装 Rails，指定版本

```
$ gem install rails -v 4.2.0.beta4
```

-v 旗标的作用是指定安装哪个 Rails 版本。你使用的版本必须和我一样，这样学习的过程中，你我得到的结果才相同。

1.3. 第一个应用

按照计算机编程领域长期沿用的传统，第一个应用的目的是编写一个“hello, world”程序。具体说来，我们要创建一个简单的应用，在网页中显示字符串“hello, world！”，在开发环境（1.3.4 节）和线上网站中（1.5 节）都是如此。

Rails 应用一般都从 `rails new` 命令开始，这个命令会在你指定的文件夹中创建一个 Rails 应用骨架。如果没有使用 1.2.1 节推荐的 Cloud9 IDE，首先你要新建一个文件夹，命名为 `workspace`，然后进入这个文件夹，如代

6. Cloud9 目前提供的 Rails 版本有点儿旧，和本书不兼容，所以才要自己动手安装。

码清单 1.2 所示。（代码清单 1.2 中使用了 Unix 命令 `cd` 和 `mkdir`，如果你不熟悉这些命令，可以阅读旁注 1.3。）

代码清单 1.2：为 Rails 项目新建一个文件夹，命名为 `workspace`（在云端环境中不用这一步）

```
$ cd          # 进入家目录  
$ mkdir workspace # 新建 workspace 文件夹  
$ cd workspace/ # 进入 workspace 文件夹
```

旁注 1.3：Unix 命令行速成课

使用 Windows 和 Mac OS X（数量较少，但增长势头迅猛）的用户可能对 Unix 命令行不熟悉。如果使用推荐的云端环境，很幸运，这个环境提供了 Unix（Linux）命令行——在标准的 [shell 命令行界面](#) 中运行的 [Bash](#)。

命令行的基本思想很简单：使用简短的命令执行很多操作，例如创建文件夹（`mkdir`），移动和复制文件（`mv` 和 `cp`），以及变换目录浏览文件系统（`cd`）。主要使用图形化界面（Graphical User Interface，简称 GUI）的用户可能觉得命令行落后，其实是被表象蒙蔽了：命令行是开发者最强大的工具之一。其实，你经常会看到经验丰富的开发者开着多个终端窗口，运行命令行 shell。

这是一门很深的学问，但在本书中只会用到一些最常用的 Unix 命令行命令，如表 1.1 所示。若想更深入地学习 Unix 命令行，请阅读 Mark Bates 写的《Conquering the Command Line》（可以[免费在线阅读](#)，也可以[购买电子书和视频](#)）。

表 1.1：一些常用的 Unix 命令

作用	命令	示例
列出内容	<code>ls</code>	<code>\$ ls -l</code>
新建文件夹	<code>mkdir <dirname></code>	<code>\$ mkdir workspace</code>
变换目录	<code>cd <dirname></code>	<code>\$ cd workspace/</code>
进入上层目录		<code>\$ cd ..</code>
进入家目录		<code>\$ cd ~ 或 \$ cd</code>
进入家目录中的文件夹		<code>\$ cd ~/workspace/</code>
移动文件（重命名）	<code>mv <source> <target></code>	<code>\$ mv README.rdoc README.md</code>
复制文件	<code>cp <source> <target></code>	<code>\$ cp README.rdoc README.md</code>
删除文件	<code>rm <file></code>	<code>\$ rm README.rdoc</code>
删除空文件夹	<code>rmdir <directory></code>	<code>\$ rmdir workspace/</code>
删除非空文件夹	<code>rm -rf <directory></code>	<code>\$ rm -rf tmp/</code>
连结并显示文件的内容	<code>cat <file></code>	<code>\$ cat ~/.ssh/id_rsa.pub</code>

不管在本地环境，还是在云端 IDE 中，下一步都是使用代码清单 1.3 中的命令创建第一个应用。注意，在这个代码清单中，我们明确指定了 Rails 版本（4.2.0.beta4）。这么做的目的是，确保使用代码清单 1.1 中安装

的 Rails 版本来创建这个应用的文件夹结构。（执行[代码清单 1.3](#) 中的命令时，如果返回“Could not find ‘railties’”这样的错误，说明你没安装正确的 Rails 版本，再次确认你安装 Rails 时执行的命令和[代码清单 1.1](#) 一模一样。）

代码清单 1.3：执行 `rails new` 命令（明确指定版本号）

```
$ cd ~/workspace
$ rails _4.2.0.beta4_ new hello_app
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  .
  .
  .
  create  test/test_helper.rb
  create  tmp/cache
  create  tmp/cache/assets
  create  vendor/assets/javascripts
  create  vendor/assets/javascripts/.keep
  create  vendor/assets/stylesheets
  create  vendor/assets/stylesheets/.keep
    run  bundle install
Fetching gem metadata from https://rubygems.org/.....
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake 10.3.2
Using i18n 0.6.11
.
.
.
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
  run  bundle exec spring binstub --all
* bin/rake: spring inserted
* bin/rails: spring inserted
```

如代码清单 1.3 所示，执行 `rails new` 命令生成所有文件之后，会自动执行 `bundle install` 命令。我们会在[1.3.1 节](#)说明这个命令的作用。

留意一下 `rails new` 命令创建的文件和文件夹。这个标准的文件夹结构（如[图 1.4](#)）是 Rails 的众多优势之一——让你从零开始快速的创建一个可运行的简单应用。而且，所有 Rails 应用都使用这种文件夹结构，所以阅读他人的代码时很快就能理清头绪。这些文件的作用如[表 1.2](#) 所示，在本书的后续内容中将介绍其中大多数文件和文件夹。从[5.2.1 节](#)开始，我们将介绍 `app/assets` 文件夹，它是 Asset Pipeline 的一部分。Asset Pipeline 把组织、部署 CSS 和 JavaScript 等资源文件变得异常简单。

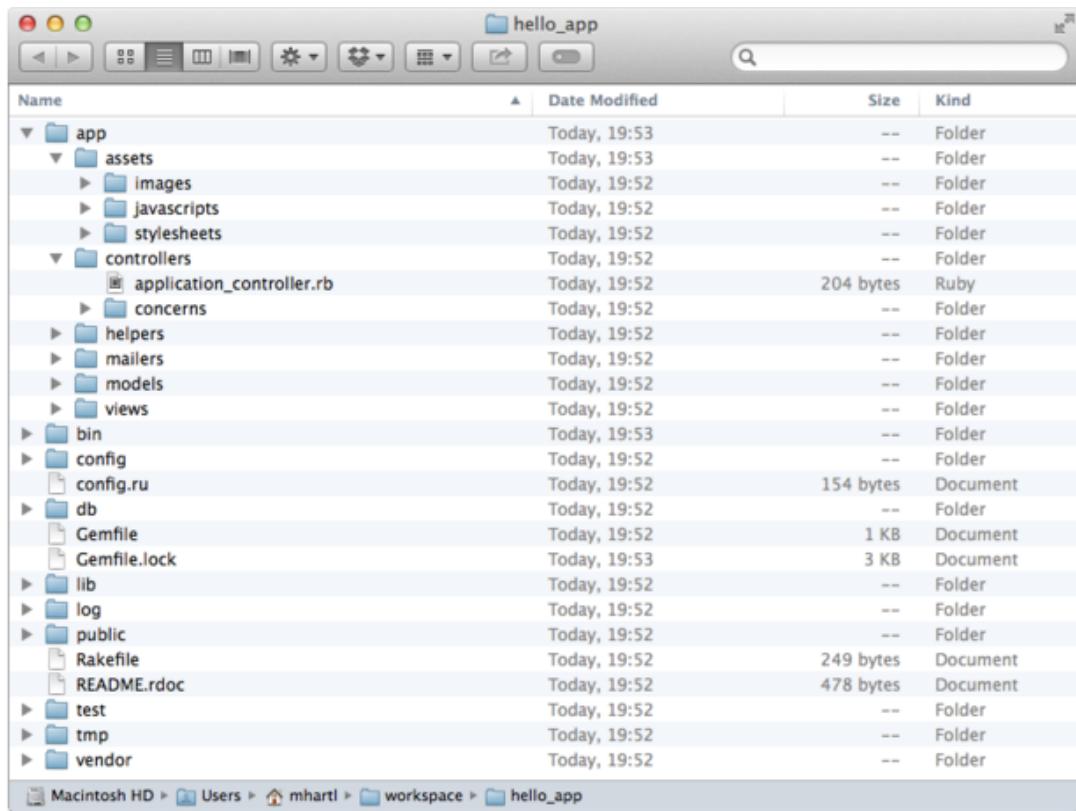


图 1.4：新建 Rails 应用的文件夹结构

表 1.2：Rails 文件夹结构简介

文件/文件夹	作用
app/	应用的核心文件，包含模型、视图、控制器和辅助方法
app/assets	应用的资源文件，例如层叠样式表（CSS）、JavaScript 和图片
bin/	可执行二进制文件
config/	应用的配置
db/	数据库相关的文件
doc/	应用的文档
lib/	代码库模块文件
lib/assets	代码库的资源文件，例如 CSS、JavaScript 和图片
log/	应用的日志文件
public/	公共（例如浏览器）可访问的文件，例如错误页面
bin/rails	生成代码、打开终端会话或启动本地服务器的程序
test/	应用的测试
tmp/	临时文件

文件/文件夹	作用
vendor/	第三方代码，例如插件和 gem
vendor/assets	第三方资源文件，例如 CSS、JavaScript 和图片
README.rdoc	应用简介
Rakefile	使用 rake 命令执行的实用任务
Gemfile	应用所需的 gem
Gemfile.lock	gem 列表，确保这个应用的副本使用相同版本的 gem
config.ru	Rack 中间件的配置文件
.gitignore	Git 忽略的文件

1.3.1. Bundler

创建完一个新的 Rails 应用后，下一步是使用 Bundler 安装和包含该应用所需的 gem。在 1.3 节简单提到过，执行 rails new 命令时会自动运行 Bundler（通过 bundle install 命令）。不过这一节，我们要修改应用默认使用的 gem，然后再次运行 Bundler。首先，在文本编辑器中打开文件 Gemfile，虽然具体的版本号和内容或许有所不同，但大概与代码清单 1.4 和图 1.5 差不多。（这个文件中的内容是 Ruby 代码，现在先不关心语法，第 4 章会详细介绍 Ruby。）如果你没看到如图 1.5 所示的文件和文件夹，点击文件浏览器中的齿轮图标，然后选择“Refresh File Tree”（刷新文件树）。（如果没出现某个文件或文件夹，就可以刷新文件树。）

代码清单 1.4: hello_app 中默认生成的 Gemfile

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.0.beta4'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0.0.beta1'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'
# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes following links in your web application faster. Read more:
# https://github.com/rails/turbolinks
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0', group: :doc
```

```

# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Rails Html Sanitizer for HTML sanitization
gem 'rails-html-sanitizer', '~> 1.0'

# Use Unicorn as the app server
# gem 'unicorn'

# Use Capistrano for deployment
# gem 'capistrano-rails', group: :development

group :development, :test do
  # Call 'debugger' anywhere in the code to stop execution and get a
  # debugger console
  gem 'byebug'

  # Access an IRB console on exceptions page and /console in development
  gem 'web-console', '~> 2.0.0.beta2'

  # Spring speeds up development by keeping your application running in the
  # background. Read more: https://github.com/rails/spring
  gem 'spring'
end

```

其中很多行代码都用 `#` 符号注释掉了，这些代码放在这里是为了告诉你一些常用的 gem，也是为了展示 Bundler 的句法。现在，除了这些默认的 gem 之外，我们还不需要其他的 gem。

如果没在 `gem` 指令中指定版本号，Bundler 会自动最新版。下面就是一例：

```
gem 'sqlite3'
```

还有两种常用的方法，用来指定 gem 版本的范围，一定程度上控制 Rails 使用的版本。首先看下面这行代码：

```
gem 'uglifier', '>= 1.3.0'
```

这行代码的意思是，安装版本号大于或等于 1.3.0 的最新版 `uglifier`（作用是压缩 Asset Pipeline 中的文件），就算是 7.2 版也会安装。第二种方法如下所示：

```
gem 'coffee-rails', '~> 4.0.0'
```

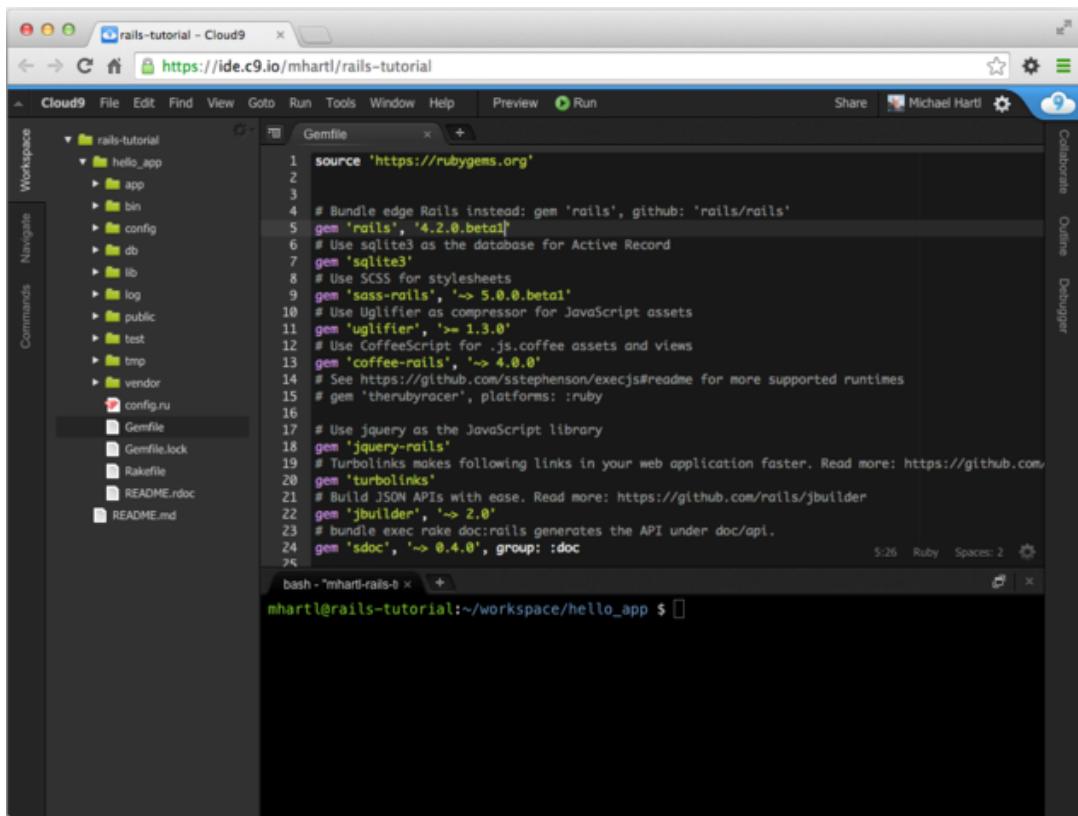


图 1.5：在文本编辑器中打开默认生成的 Gemfile

这行代码的意思是，安装版本号大于 4.0.0，但小于 4.1 的 coffee-rails。也就是说，`>=` 符号的意思是始终安装最新版；`~> 4.0.0` 的意思是只安装补丁版本号变化的版本（例如从 4.0.0 到 4.0.1），而不安装次版本或主版本的更新（例如从 4.0 到 4.1）。不过，经验告诉我们，即使是补丁版本的升级也可能导致错误，所以在本教程中我们基本上会为所有的 gem 都指定精确的版本号。你可以使用任何 gem 的最新版本，还可以在 Gemfile 中使用 `~>`（一般推荐有经验的用户使用），但事先提醒你，这可能会导致本教程开发的应用表现异常。

修改[代码清单 1.4](#) 中的 Gemfile，换用精确的版本号，得到的结果如[代码清单 1.5](#) 所示。注意，借此机会我们还变动了 sqlite3 的位置，只在开发环境和测试环境（[7.1.1 节](#)）中安装，避免和 Heroku 所用的数据库冲突（[1.5 节](#)）。

代码清单 1.5：每个 Ruby gem 都使用精确版本号的 Gemfile

```
source 'https://rubygems.org'

gem 'rails',                  '4.2.0.beta4'
gem 'sass-rails',              '5.0.0.beta1'
gem 'uglifier',                '2.5.3'
gem 'coffee-rails',             '4.0.1'
gem 'jquery-rails',             '4.0.0.beta2'
gem 'turbolinks',               '2.3.0'
gem 'jbuilder',                 '2.2.3'
gem 'rails-html-sanitizer',     '1.0.1'
gem 'sdoc', '0.4.0', group: :doc
```

```
group :development, :test do
  gem 'sqlite3',      '1.3.9'
  gem 'byebug',       '3.4.0'
  gem 'web-console',   '2.0.0.beta3'
  gem 'spring',        '1.1.3'
end
```

把代码清单 1.5 中的内容写入应用的 `Gemfile` 文件之后，执行 `bundle install` 命令⁷ 安装这些 gem：

```
$ cd hello_app/
$ bundle install
Fetching source index for https://rubygems.org/
.
.
.
```

`bundle install` 命令可能要执行一会儿，不过结束后我们的应用就可以运行了。

1.3.2. rails server

运行完 1.3 节中的 `rails new` 命令和 1.3.1 节中的 `bundle install` 命令之后，我们的应用就可以运行了，但是怎么运行呢？Rails 自带了一个命令行程序（或叫脚本），可以运行一个本地服务器，协助我们的开发工作。这个命令具体怎么执行，取决于你使用的环境：在本地系统中，直接执行 `rails server` 命令就行（[代码清单 1.6](#)）；而在 Cloud9 中，还要指定绑定的 IP 地址和端口号，告诉 Rails 服务器外界可以通过哪个地址访问应用（[代码清单 1.7](#)）。⁸（Cloud9 使用特殊的环境变量 `$IP` 和 `$PORT` 动态指定 IP 地址和端口号。如果想查看这两个环境变量的值，可以在命令行中输入 `echo $IP` 和 `echo $PORT`。）

代码清单 1.6：在本地设备中运行 Rails 服务器

```
$ cd ~/workspace/hello_app/
$ rails server
=> Booting WEBrick
=> Rails application starting on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

代码清单 1.7：在云端 IDE 中运行 Rails 服务器

```
$ cd ~/workspace/hello_app/
$ rails server -b $IP -p $PORT
=> Booting WEBrick
=> Rails application starting on http://0.0.0.0:8080
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

不管使用哪种环境，我都建议你在另一个终端选项卡中执行 `rails server` 命令，这样你就可以继续在第一个选项卡中执行其他命令了，如图 1.6 和图 1.7 所示。（如果你已经在第一个选项卡中启动了服务器，可以按 `Ctrl-C` 键关闭服务器。）在本地环境中，在浏览器中打开 <http://localhost:3000/>；在云端 IDE 中，打开

7. 如表 3.1 所示，可以省略 `install`，因为 `bundle` 是 `bundle install` 的别名。

8. 一般情况下，网站使用 80 端口，但这个端口需要特别的权限。所以，为了方便，开发服务器最好使用没有限制的大数端口。

“Share”（分享）面板，点击“Application”后的地址即可打开应用（如图 1.8）。在这两种环境中，显示的页面应该都和图 1.9 类似。

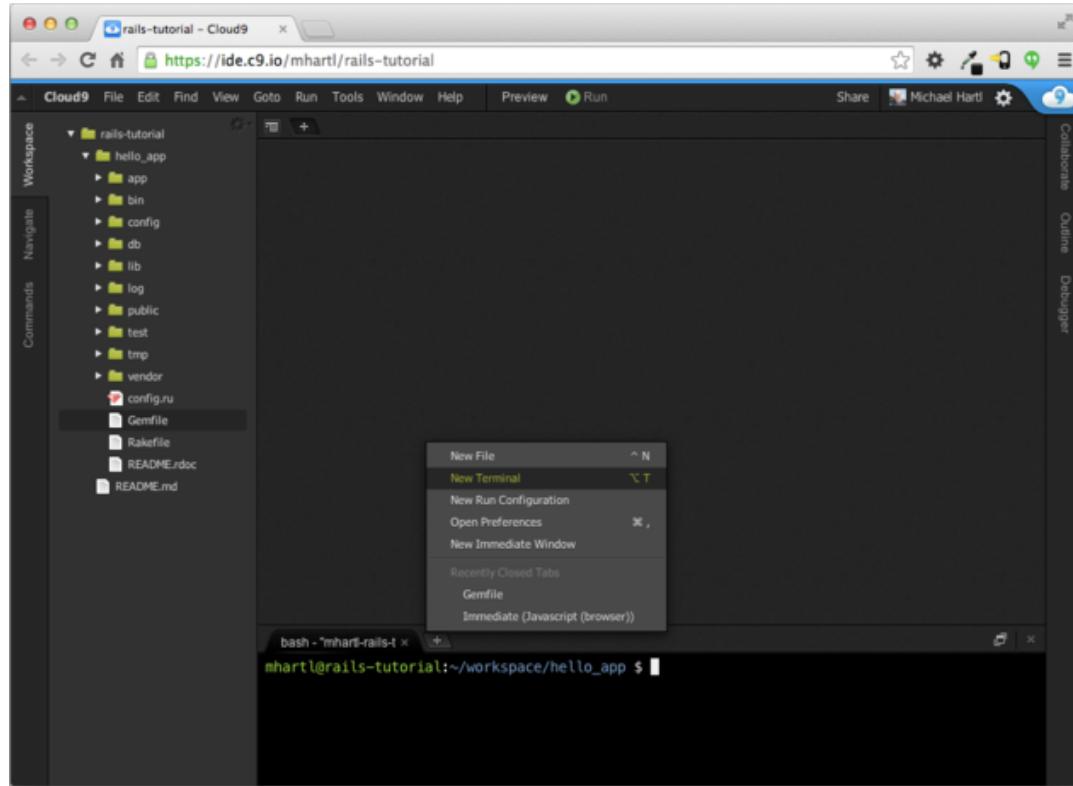


图 1.6：再打开一个终端选项卡

点击“About your application’s environment”可以查看应用的信息。你看到的版本号可能和我的不一样，但和图 1.10 差不多。当然了，从长远来看，我们不需要这个 Rails 默认页面，不过现在看到这个页面说明 Rails 可以正常运行了。我们会在1.3.4 节删除这个页面，替换成我们自己写的首页。

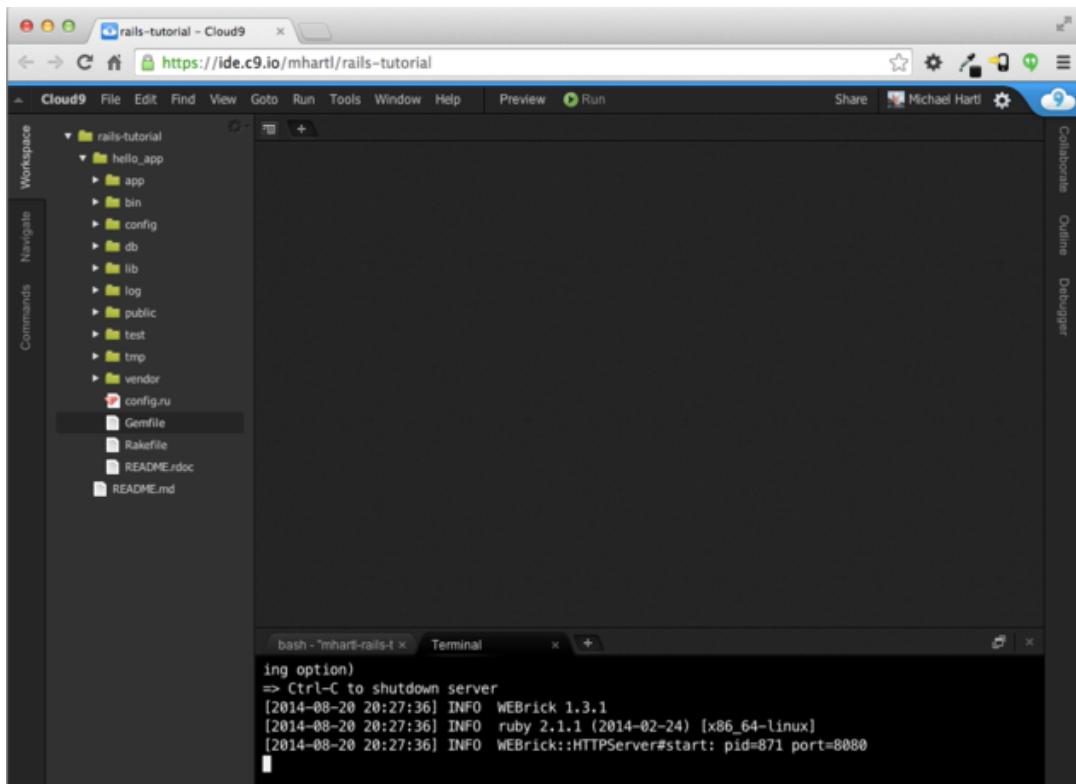


图 1.7：在另一个选项卡中运行 Rails 服务器

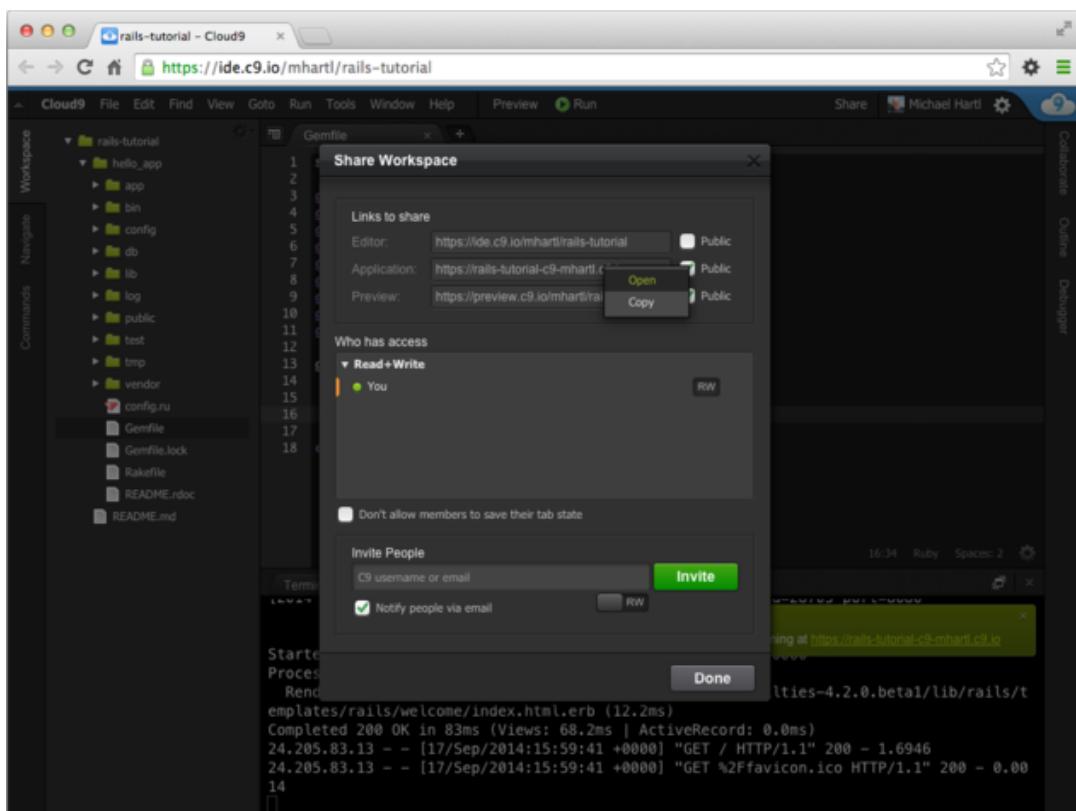


图 1.8：分享运行在云端工作空间中的本地服务器

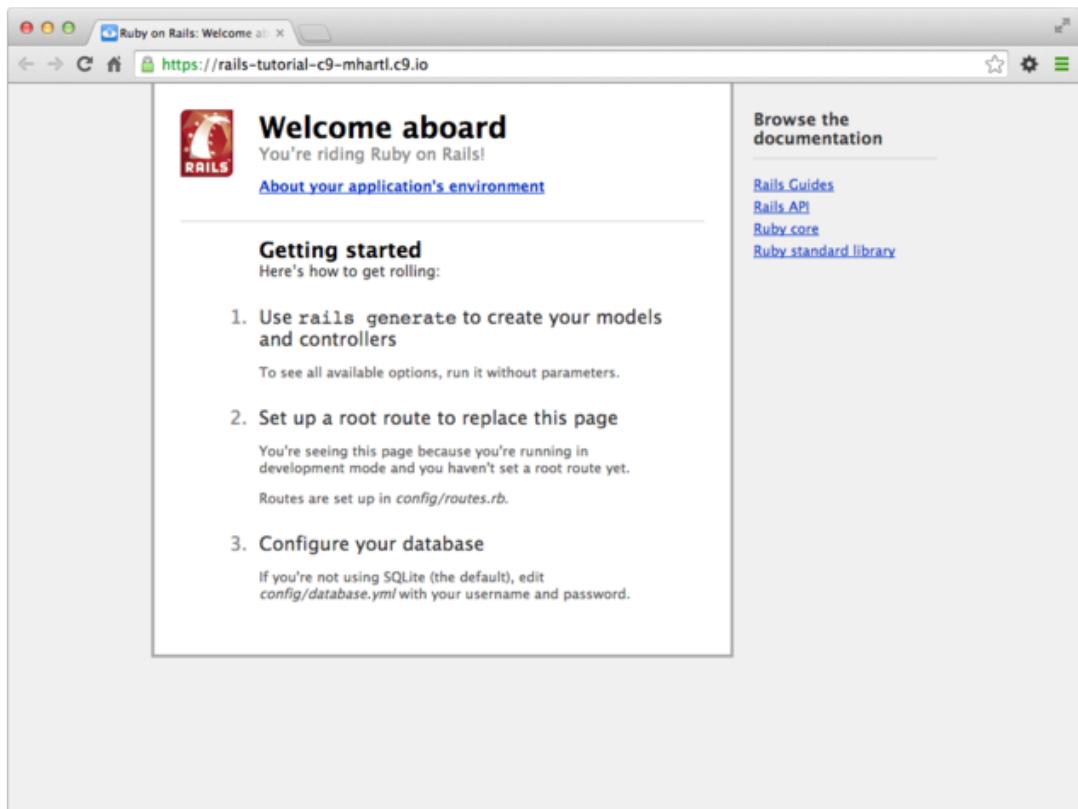
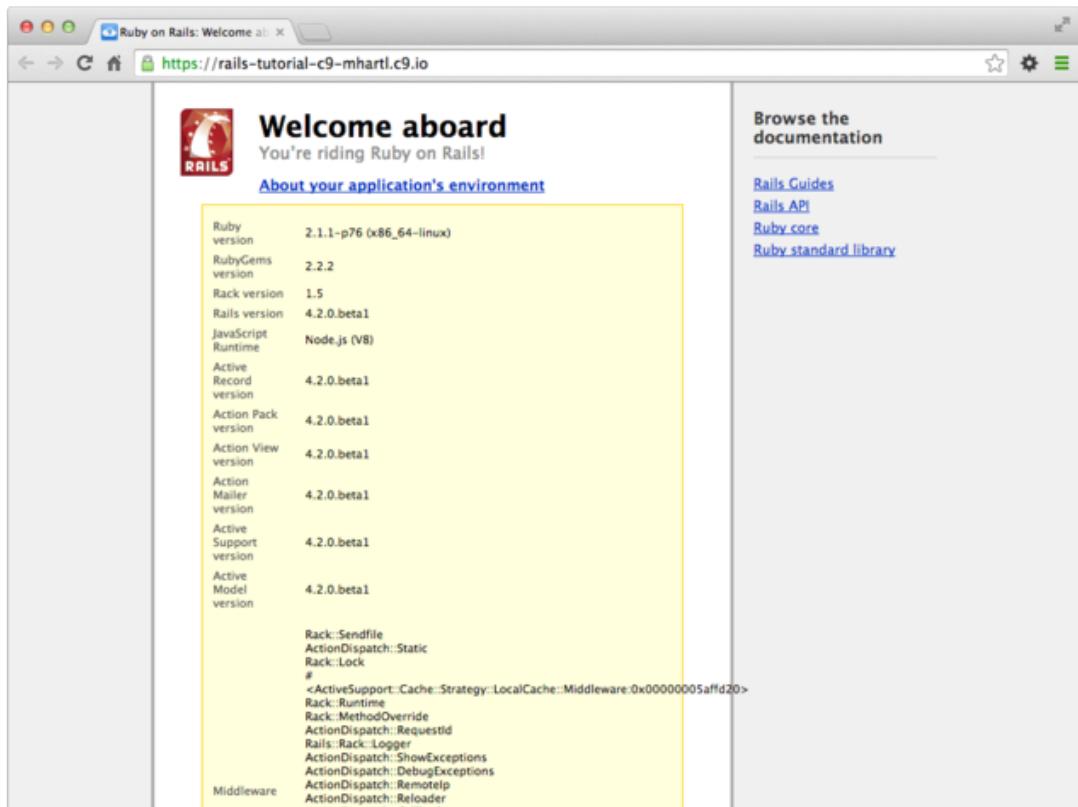


图 1.9：执行 `rails server` 命令后看到的 Rails 默认页面



Ruby version	2.1.1-p76 (x86_64-linux)
RubyGems version	2.2.2
Rack version	1.5
Rails version	4.2.0.beta1
JavaScript Runtime	Node.js (V8)
Active Record version	4.2.0.beta1
Action Pack version	4.2.0.beta1
Action View version	4.2.0.beta1
Action Mailer version	4.2.0.beta1
Active Support version	4.2.0.beta1
Active Model version	4.2.0.beta1
Middleware	<pre>Rack::Sendfile ActionDispatch::Static Rack::Lock # <ActiveSupport::Cache::Strategy::LocalCache:0x00000005affd20> Rack::Runtime Rack::MethodOverride ActionDispatch::Requestid Rack::rack ActionDispatch::ShowExceptions ActionDispatch::DebugExceptions ActionDispatch::RemoteIp ActionDispatch::Reloader ActionDispatch::Callbacks</pre>

图 1.10：默认页面中显示应用的环境信息

1.3.3. 模型-视图-控制器

在初期阶段，概览一下 Rails 应用的工作方式（图 1.11）多少会有些帮助。你可能已经注意到了，在 Rails 应用的标准文件夹结构中有一个文件夹叫 `app/`（图 1.4），其中有三个子文件夹：`models`、`views` 和 `controllers`。这暗示 Rails 采用了“模型-视图-控制器”（简称 MVC）架构模式，这种模式把“域逻辑”（domain logic，也叫“业务逻辑”（business logic））与图形用户界面相关的输入和表现逻辑强制分开。在 Web 应用中，“域逻辑”一般是“用户”、“文章”和“商品”等数据模型，GUI 则是浏览器中的网页。

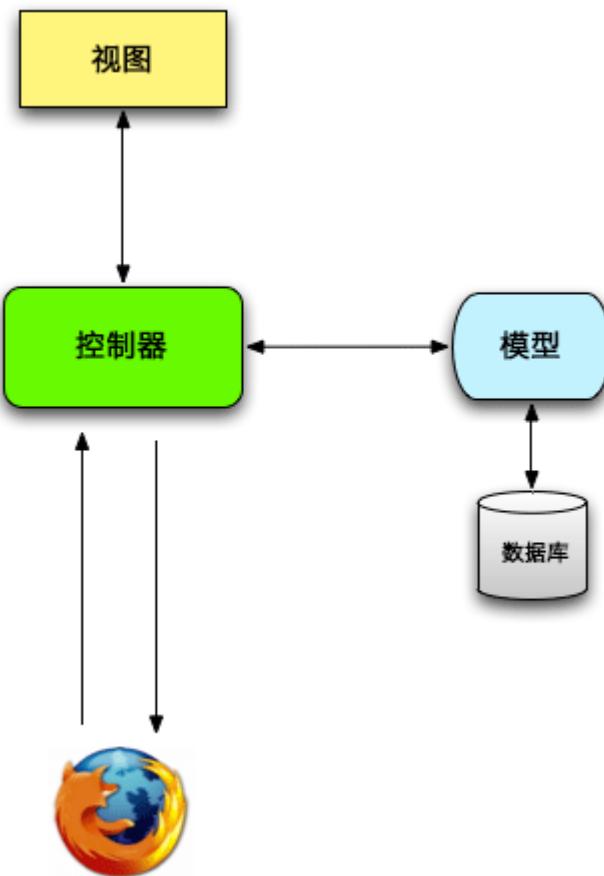


图 1.11：MVC 架构图解

和 Rails 应用交互时，浏览器发出一个请求（request），Web 服务器收到这个请求之后将其传给 Rails 应用的控制器，由控制器决定下一步该做什么。某些情况下，控制器会立即渲染视图（view），生成 HTML，然后发送给浏览器。对于动态网站来说，更常见的是控制器和模型（model）交互。模型是一个 Ruby 对象，表示网站中的一个元素（例如一个用户），并且负责和数据库通信。和模型交互后，控制器再渲染视图，并把生成的 HTML 返回给浏览器。

如果你觉得这些内容有点抽象，不用担心，后面会经常讲到 MVC。在 1.3.4 节中，首次使用 MVC 架构编写应用；在 2.2.2 节中，会以一个应用为例较为深入地讨论 MVC；在最后那个演示应用中会使用完整的 MVC 架构。从 3.2 节开始，介绍控制器和视图；从 6.1 节开始，介绍模型；7.1.2 节则把这三部分放在一起使用。

1.3.4. Hello, world!

接下来我们要对这个使用 MVC 框架开发的第一个应用做些小改动——添加一个控制器动作，渲染字符串“hello, world!”。（从 [2.2.2 节](#) 开始会更深入的介绍控制器动作。）这一节的目的是，使用显示“hello, world!”的页面替换 Rails 默认的首页（[图 1.9](#)）。

从“控制器动作”这个名字可以看出，动作在控制器中定义。我们要在 `ApplicationController` 中定义这个动作，并将其命名为 `hello`。其实，现在我们的应用只有 `ApplicationController` 这一个控制器。执行下面的命令可以验证这一点（从 [第 2 章](#) 开始，我们会创建自己的控制器。）：

```
$ ls app/controllers/*_controller.rb
```

`hello` 动作的定义体如[代码清单 1.8](#) 所示，调用 `render` 函数返回文本“hello, world!”。（现在先不管 Ruby 的句法，[第 4 章](#) 会详细介绍。）

代码清单 1.8：在 `ApplicationController` 中添加 `hello` 动作

`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  def hello
    render text: "hello, world!"
  end
end
```

定义好返回所需字符串的动作之后，我们要告诉 Rails 使用这个动作，不再显示默认的首页（[图 1.10](#)）。为此，我们要修改 Rails 路由。路由在控制器之前（[图 1.11](#)），决定浏览器发给应用的请求由哪个动作处理。

（简单起见，[图 1.11](#) 中省略了路由，从 [2.2.2 节](#) 开始会详细介绍路由。）具体而言，我们要修改默认的首页，也就是根路由。这个路由决定根 URL 显示哪个页面。根 URL 是 `http://www.example.com/` 这种形式，所以一般简化使用 /（斜线）表示。

如[代码清单 1.9](#) 所示，Rails 应用的路由文件（`config/routes.rb`）中有一行注释，说明如何编写根路由。其中，“welcome”是控制器名，“index”是这个控制器中的动作名。去掉这行前面的 `#` 号，解除注释，这样根路由就可以定义了，然后再把内容替换成[代码清单 1.10](#) 中的代码，告诉 Rails 把根路由交给 `ApplicationController` 中的 `hello` 动作处理。（在 [1.1.2 节](#) 说过，竖排的点号表示省略的代码，不要直接复制。）

代码清单 1.9：生成的默认根路由（在注释中）

`config/routes.rb`

```
Rails.application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  .
  .
  .
end
```

代码清单 1.10：设定根路由

`config/routes.rb`

```
Rails.application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  root 'application#hello'
  .
  .
  .
end
```

有了[代码清单 1.8](#) 和[代码清单 1.10](#) 中的代码，根路由就会按照我们的要求显示“hello, world!”了，如图 1.12 所示。

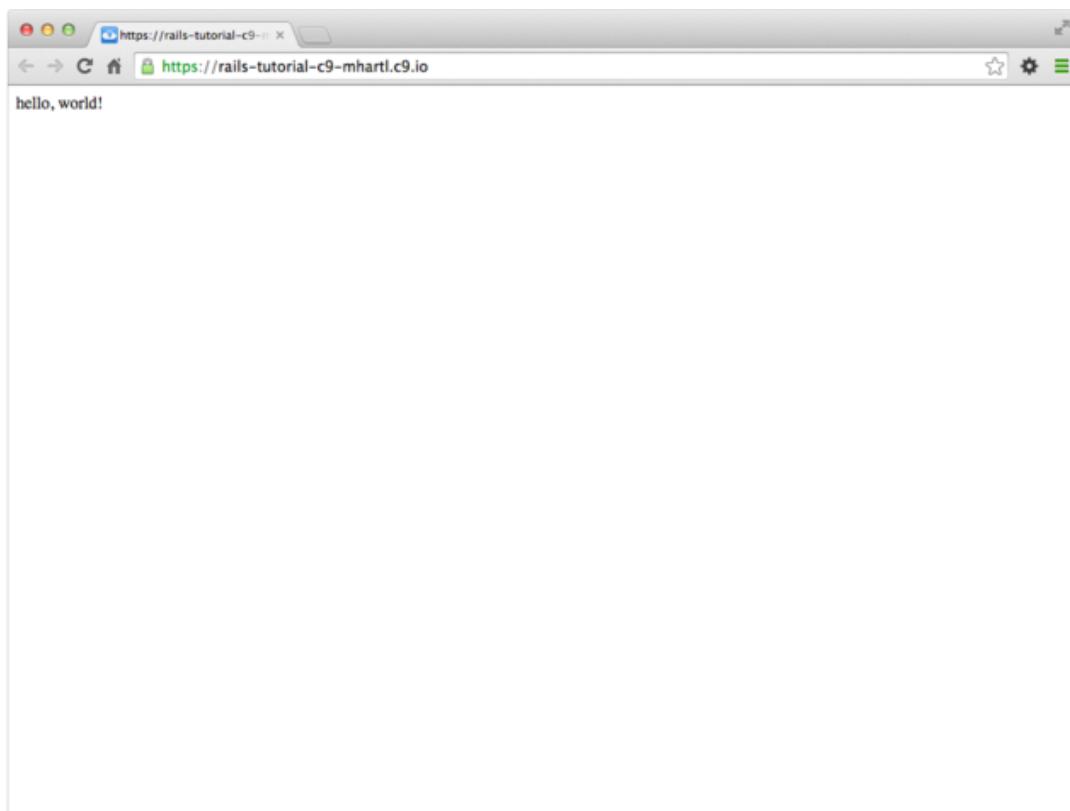


图 1.12：在浏览器中查看显示“hello, world!”的页面

1.4. 使用 Git 做版本控制

我们已经开发了一个可以运行的 Rails 应用，接下来要花点时间来做一件事。虽然这件事不是必须的，但是经验丰富的软件开发者都认为这是最基本的事情，即把应用的源代码纳入版本控制。版本控制系统可以跟踪项目中代码的变化，便于和他人协作，如果出现问题（例如不小心删除了文件）还可以回滚到以前的版本。每个专业级软件开发者都应该学习使用版本控制系统。

版本控制系统种类很多，Rails 社区基本都使用 Git。Git 由 Linus Torvalds 开发，最初目的是存储 Linux 内核代码。Git 相关的知识很多，本书只会介绍一些皮毛。网络上有很多免费的资料，我特别推荐 Scott Chacon 写的《[Pro Git](#)》。⁹之所以推强烈推荐使用 Git 做版本控制，不仅因为 Rails 社区都在用，还因为使用 Git 分享代码更简单（[1.4.3 节](#)），而且也便于应用的部署（[1.5 节](#)）。

1.4.1. 安装和设置

[1.2.1 节](#) 推荐使用的云端 IDE 默认已经集成 Git，不用再安装。如果你没使用云端 IDE，可以参照 [Install-Rails.com](#) 中的说明，在自己的系统中安装 Git。

第一次运行前要做的系统设置

使用 Git 前，要做一些一次性设置。这些设置对整个系统都有效，因此一台电脑只需设置一次：

```
$ git config --global user.name "Your Name"  
$ git config --global user.email your.email@example.com  
$ git config --global push.default matching  
$ git config --global alias.co checkout
```

注意，在 Git 配置中设定的名字和电子邮件地址会在所有公开的仓库中显示。（前两个设置必须做。第三个设置是为了向前兼容未来的 Git 版本。第四个设置是可选的，如果设置了，就可以使用 `co` 代替 `checkout` 命令。为了最大程度上兼容没有设置 `co` 的系统，本书仍将继续使用全名 `checkout`，不过在现实中我基本都用 `git co`。）

第一次使用仓库前要做的设置

下面的步骤每次新建仓库时都要执行。首先进入第一个应用的根目录，然后初始化一个新仓库：

```
$ git init  
Initialized empty Git repository in /home/ubuntu/workspace/hello_app/.git/
```

然后执行 `git add -A` 命令，把项目中的所有文件都放到仓库中：

```
$ git add -A
```

这个命令会把当前目录中的所有文件都放到仓库中，但是匹配特殊文件 `.gitignore` 中模式的文件除外。
`rails new` 命令会自动生成一个适用于 Rails 项目的 `.gitignore` 文件，而且你还可以添加其他模式。¹⁰

加入仓库的文件一开始位于“暂存区”（staging area），这一区用于存放待提交的内容。执行 `status` 命令可以查看暂存区中有哪些文件：

```
$ git status  
On branch master  
  
Initial commit  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)
```

9. 译者注：中文版在线阅读地址：<http://git-scm.com/book/zh>。也可以阅读译者制作的电子书：<https://selfstore.io/products/46>。

10. 本书基本不会修改这个文件，不过 [3.7 节](#) 的高级测试配置中演示了如何修改。

```
new file: .gitignore
new file: Gemfile
new file: Gemfile.lock
new file: README.rdoc
new file: Rakefile
.
.
.
```

(显示的内容很多，所以我使用竖排点号省略了一些内容。)

如果想告诉 Git 保留这些改动，可以使用 `commit` 命令：

```
$ git commit -m "Initialize repository"
[master (root-commit) df0a62f] Initialize repository
.
.
.
```

旗标 `-m` 的意思是为这次提交添加一个说明。如果没指定 `-m` 旗标，Git 会打开系统默认使用的编辑器，让你在其中输入说明。（本书所有的示例都会使用 `-m` 旗标。）

有一点很重要要注意：Git 提交只发生在本地，也就是说只在执行提交操作的设备中存储内容。[1.4.4 节](#)会介绍如何把改动推送（使用 `git push` 命令）到远程仓库中。

顺便说一下，可以使用 `log` 命令查看提交的历史：

```
$ git log
commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Wed August 20 19:44:43 2014 +0000

    Initialize repository
```

如果仓库的提交历史很多，可能需要输入 `q` 退出。

1.4.2. 使用 Git 有什么好处

如果以前从未用过版本控制，现在可能不完全明白版本控制的好处。那我举个例子说明一下吧。假如你不小心做了某个操作，例如把重要的 `app/controllers/` 文件夹删除了：

```
$ ls app/controllers/
application_controller.rb  concerns/
$ rm -rf app/controllers/
$ ls app/controllers/
ls: app/controllers/: No such file or directory
```

我们用 Unix 中的 `ls` 命令列出 `app/controllers/` 文件夹里的内容，然后用 `rm` 命令删除这个文件夹。旗标 `-rf` 的意思是“强制递归”，无需明确征求同意就递归删除所有文件、文件夹和子文件夹等。

查看一下状态，看看发生了什么：

```
$ git status
On branch master
```

```
Changed but not updated:  
(use "git add/rm <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)  
  
deleted:    app/controllers/application_controller.rb  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

可以看出，删除了一个文件。但是这个改动只发生在“工作树”中，还未提交到仓库。所以，我们可以使用 `checkout` 命令，并指定 `-f` 旗标，强制撤销这次改动：

```
$ git checkout -f  
$ git status  
# On branch master  
nothing to commit (working directory clean)  
$ ls app/controllers/  
application_controller.rb  concerns/
```

删除的文件夹和文件又回来了，这下放心了！

1.4.3. Bitbucket

我们已经把项目纳入 Git 版本控制系统了，接下来可以把代码推送到 Bitbucket 中。Bitbucket 是一个专门用来托管和分享 Git 仓库的网站。（本书前几版使用 GitHub，换用 Bitbucket 的原因参见旁注 1.4。）在 Bitbucket 中放一份 Git 仓库的副本有两个目的：其一，对代码做个完整备份（包括所有提交历史）；其二，便于以后协作。

旁注 1.4：GitHub 和 Bitbucket

目前，托管 Git 仓库最受欢迎的网站是 GitHub 和 Bitbucket。这两个网站有很多相似之处：都可托管仓库，也可以协作，而且浏览和搜索仓库很方便。但二者之间有个重要的区别（对本书而言）：GitHub 为开源项目提供无限量的免费仓库，但私有仓库收费；而 Bitbucket 提供了无限量的私有仓库，仅当协作者超过一定数量时才收费。所以，选择哪个网站，取决于具体的需求。

本书前几版使用 GitHub，因为它对开源项目来说有很多好用的功能，但我越来越关注安全，所以推荐所有 Web 应用都放在私有仓库中。因为 Web 应用的仓库中可能包含潜在的敏感信息，例如密钥和密码，可能会威胁到使用这份代码的网站的安全。当然，这类信息也有安全的处理方法，但是容易出错，而且需要很多专业知识。

本书开发的演示应用可以安全地公开，但这只是特例，不能推广。因此，为了尽量提高安全，我们不能冒险，还是默认就使用私有仓库保险。既然 GitHub 对私有仓库收费，而 Bitbucket 提供了不限量的免费私有仓库，就我们的需求来说，Bitbucket 比 Github 更合适。

Bitbucket 的使用方法很简单：

1. 如果没有账户，先[注册一个 Bitbucket 账户](#)；
2. 把[公钥](#)复制到剪切板。云端 IDE 用户可以使用 `cat` 命令查看公钥，如[代码清单 1.11](#) 所示，然后选中公钥，复制。如果你在自己的系统中，执行[代码清单 1.11](#) 中的命令后没有输出，请参照“[如何在你的 Bitbucket 账户中设定公钥](#)”；

3. 点击右上角的头像，选择“Manage account”（管理账户），然后点击“SSH keys”（SSH 密钥），如图 1.13 所示。

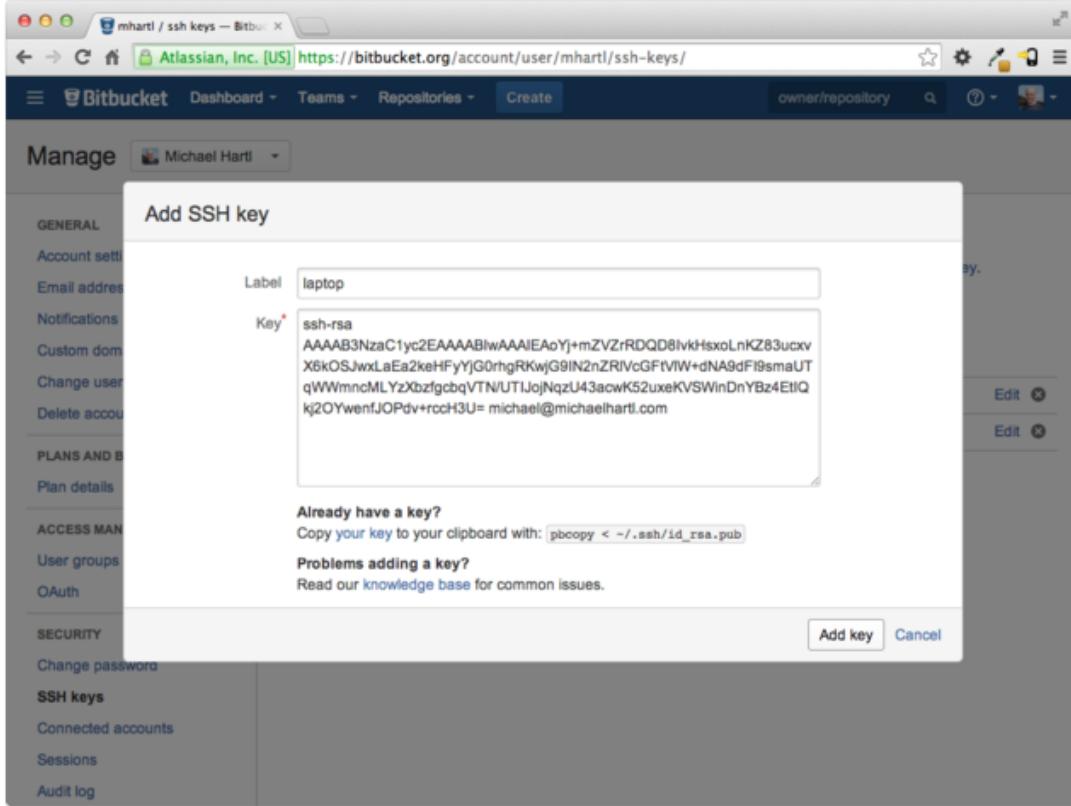


图 1.13：添加 SSH 公钥

代码清单 1.11：使用 cat 命令打印公钥

```
$ cat ~/.ssh/id_rsa.pub
```

添加公钥之后，点击“Create”（创建）按钮，新建一个仓库，如图 1.14 所示。填写项目的信息时，记得要选中“This is a private repository”（这是私有仓库）。填完后点击“Create repository”（创建仓库）按钮，然后按照“Command line > I have an existing project”（命令行 > 现有项目）下面的说明操作，如代码清单 1.12 所示。（如果与代码清单 1.12 不同，可能是公钥没正确添加，我建议你再试一次。）推送仓库时，如果询问“Are you sure you want to continue connecting (yes/no)?”（确定继续连接吗？），输入“yes”。

代码清单 1.12：添加 Bitbucket，然后推送仓库

```
$ git remote add origin git@bitbucket.org:<username>/hello_app.git
$ git push -u origin --all # 第一次推送仓库
```

这段代码的意思是，先告诉 Git，你想添加 Bitbucket，作为这个仓库的源，然后再把仓库推送到这个远端的源。（别管 -u 旗标的意思，如果好奇，可以搜索“git set upstream”。）当然了，你要把 <username> 换成你自己的用户名。例如，我运行的命令是：

```
$ git remote add origin git@bitbucket.org:mhartl/hello_app.git
```

推送完毕后，在 Bitbucket 中会显示一个 hello_app 仓库的页面。在这个页面中可以浏览文件、查看完整的提交信息，除此之外还有很多其他功能（如图 1.15 所示）。

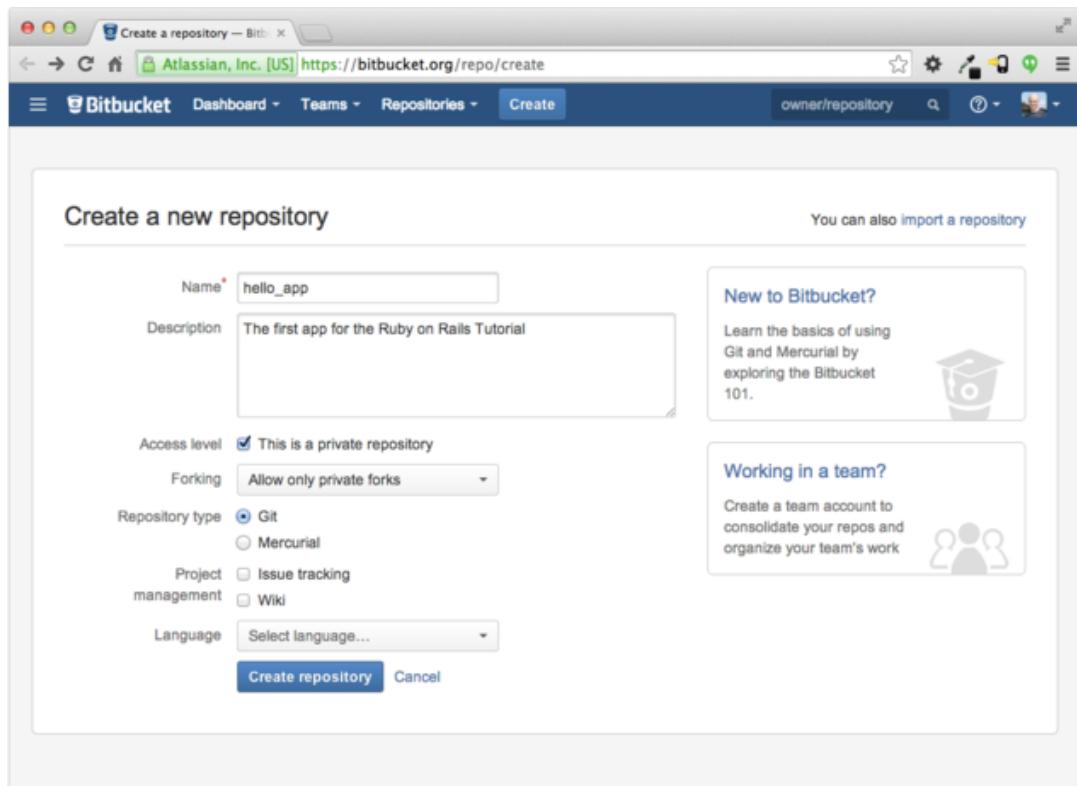


图 1.14：在 Bitbucket 中创建存放这个应用的仓库

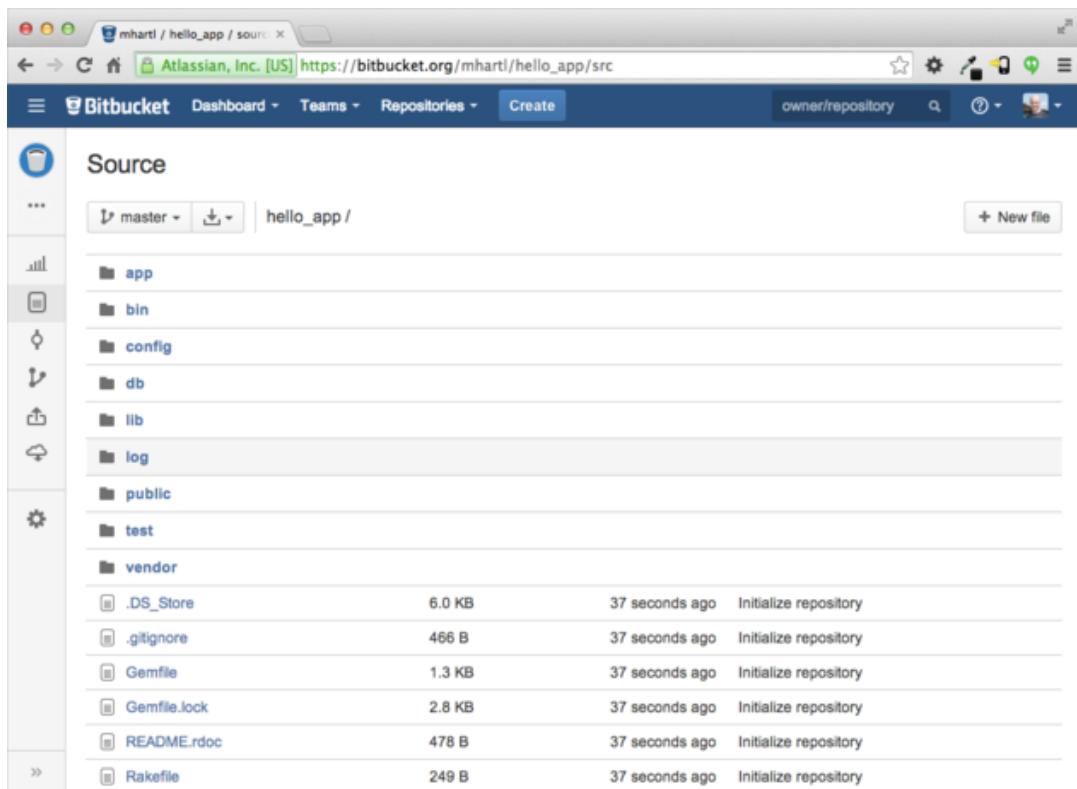


图 1.15：一个 Bitbucket 仓库的页面

1.4.4. 分支，编辑，提交，合并

如果你跟着 1.4.3 节中的步骤做，可能注意到了，Bitbucket 没有自动识别仓库中的 README.rdoc 文件，而在仓库的首页提醒没有 README 文件，如图 1.16 所示。这说明 rdoc 格式不常见，所以 Bitbucket 不支持。其实，我以及我认识的几乎所有人都使用 Markdown 格式。这一节，我们要把 README.rdoc 文件改成 README.md，顺便还要在其中添加一些针对本书的内容。在这个过程中，我们将首次演示我推荐在 Git 中使用的工作流程，即“分支，编辑，提交，合并”。¹¹

The screenshot shows the Bitbucket repository overview page. At the top, there's a navigation bar with 'Bitbucket', 'Dashboard', 'Teams', 'Repositories', 'Create', and a download icon. To the right of the download icon are 'SSH' and 'git@bitbuck' options. On the left, there's a sidebar with various icons: a bucket, three dots, a bar chart, a document, a gear, a person, a plus sign, and a cloud. The main content area has a title 'Overview'. Below it, there's a summary table:

Last updated just now	1 Branch	0 Tags
Language —	0 Forks	1 Watcher
Access level Admin		

Below the table, there's a large empty box containing a sad face icon and the text 'There isn't a README yet'. Underneath, it says 'Create one and tell people where to start and how to contribute.' At the bottom of this box is a button labeled 'Create a README'.

图 1.16：Bitbucket 提示没有 README 文件

分支

Git 中的分支功能很强大。分支是对仓库的高效复制，在分支中所做的改动（或许是实验性质的）不会影响父级文件。大多数情况下，父级仓库是 master 分支。我们可以使用 `checkout` 命令，并指定 `-b` 旗标，创建一个新“主题分支”（topic branch）：

```
$ git checkout -b modify-README  
Switched to a new branch 'modify-README'  
$ git branch
```

11. 如果想形象化的查看 Git 仓库，可以使用 Atlassian 开发的应用 [SourceTree](#)。

```
master
* modify-README
```

其中，第二个命令 `git branch` 的作用是列出所有本地分支。星号 (*) 表示当前所在的分支。注意，`git checkout -b modify-README` 命令先创建一个新分支，然后再切换到这个新分支——`modify-README` 分支前面的星号证明了这一点。（如果你在 1.4 节中设置了别名 `co`，就要使用 `git co -b modify-README`。）

只有多个开发者协作开发一个项目时，才能看出分支的全部价值。¹² 如果只有一个开发者，分支也有作用。一般情况下，要把主题分支的改动和主分支隔离开，这样即便搞砸了，随时都可以切换到主分支，然后删除主题分支，丢掉改动。本节末尾会介绍具体做法。

顺便说一下，像这种小改动，我一般不会新建分支。现在我这么做是为了让你养成好习惯。

编辑

创建主题分支后，我们要编辑 README 文件，让其更好地描述我们的项目。较之默认的 RDoc 格式，我更喜欢 [Markdown 标记语言](#)。如果文件扩展名是 `.md`，Bitbucket 会自动排版其中的内容。首先，使用 Git 提供的 Unix `mv` 命令修改文件名：

```
$ git mv README.rdoc README.md
```

然后把[代码清单 1.13](#) 中的内容写入 `README.md`。

代码清单 1.13：新 README 文件，`README.md`

```
# Ruby on Rails Tutorial: "hello, world!"  
  
This is the first application for the  
[*Ruby on Rails Tutorial*](http://www.railstutorial.org/)  
by [Michael Hartl](http://www.michaelhartl.com/).
```

提交

编辑后，查看一下该分支的状态：

```
$ git status
On branch modify-README
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:   README.rdoc -> README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:  README.md
```

这里，我们本可以使用 1.4.1.2 节中用过的 `git add -A`，但是 `git commit` 提供了 `-a` 旗标，可以直接提交现有文件中的全部改动（以及使用 `git mv` 新建的文件，对 Git 来说这不算新文件）：

12. 详情参阅《Pro Git》书中“[Git 分支](#)”一章。

```
$ git commit -a -m "Improve the README file"
2 files changed, 5 insertions(+), 243 deletions(-)
delete mode 100644 README.rdoc
create mode 100644 README.md
```

使用 `-a` 旗标一定要小心，千万别误用了。如果上次提交之后项目中添加了新文件，应该使用 `git add -A`，先告诉 Git 新增了文件。

注意，我们使用现在时（严格来说是祈使语气）编写提交消息。Git 把提交当做一系列补丁，在这种情况下，说明现在做了什么比说明过去做了什么要更合理。而且这种用法和 Git 命令生成的提交信息相配。详情参阅《[Shiny new commit styles](#)》。

合并

我们已经改完了，现在可以把结果合并到主分支了：

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating 34f06b7..2c92bef
Fast forward
 README.rdoc      | 243 -----
 README.md        |    5 +
 2 files changed, 5 insertions(+), 243 deletions(-)
 delete mode 100644 README.rdoc
 create mode 100644 README.md
```

注意，Git 命令的输出中经常会出现 `34f06b7` 这样的字符串，这是 Git 内部对仓库的指代。你得到的输出结果不会和我的一模一样，但大致相同。

合并之后，我们可以清理一下分支——如果不用这个主题分支了，可以使用 `git branch -d` 命令将其删除：

```
$ git branch -d modify-README
Deleted branch modify-README (was 2c92bef).
```

这一步可做可不做，其实一般都会留着这个主题分支，这样就可以在两个分支之间来回切换，并在合适的时候把改动合并到主分支中。

前面提过，还可以使用 `git branch -D` 命令放弃主题分支中的改动：

```
# 仅作演示之用，千万别这么做
$ git checkout -b topic-branch
$ <really screw up the branch>
$ git add -A
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

和旗标 `-d` 不同，如果指定旗标 `-D`，即使没合并分支中的改动，也会删除分支。

推送

我们已经更新了 `README` 文件，现在可以把改动推送到 Bitbucket，看看改动的效果。之前我们已经推送过一次（[1.4.3 节](#)），在大多数系统中都可以省略 `origin master`，直接执行 `git push`：

```
$ git push
```

正如前面所说，Bitbucket 对使用 Markdown 编写的文件做了精美排版，如图 1.17 所示。

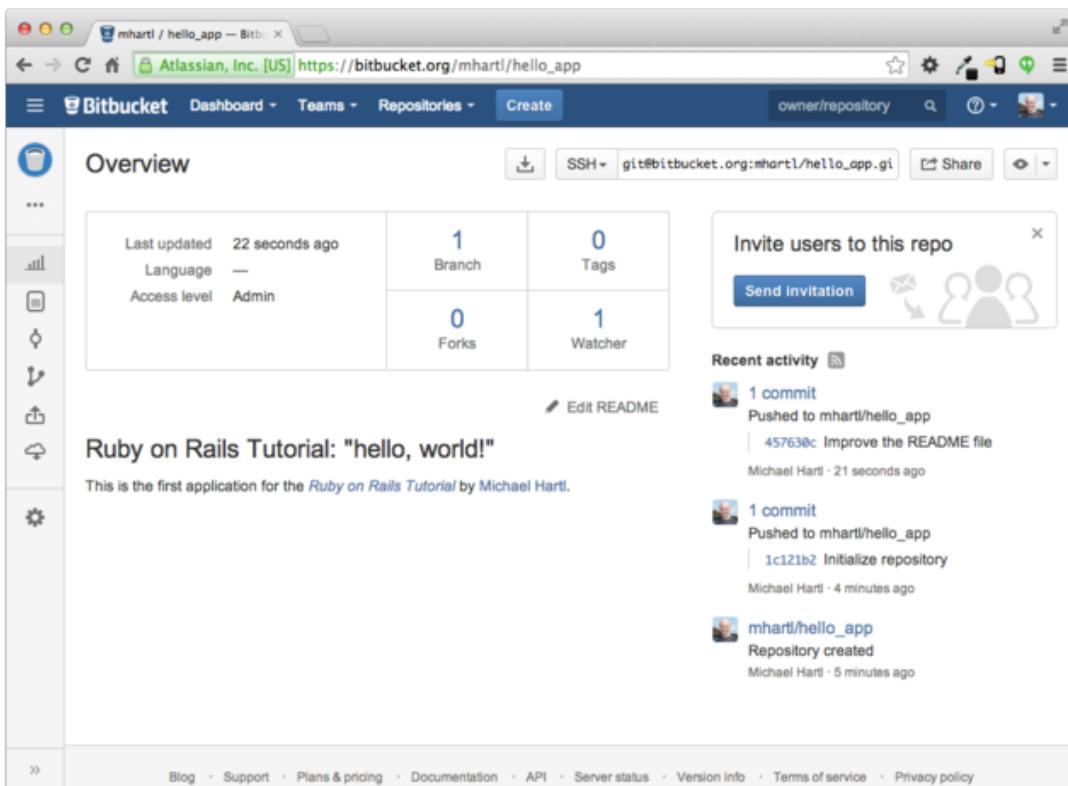


图 1.17：使用 Markdown 格式重写的 README 文件

1.5. 部署

即使现在还处在早期阶段，我们还是要把（几乎没什么内容）的 Rails 应用部署到生产环境。这一步可做可不做，不过在开发过程中尽早且频繁地部署，可以尽早发现开发中的问题。在开发环境中花费大量精力之后再部署，往往会在发布时遇到严重的集成问题。¹³

以前，部署 Rails 应用是件痛苦的事。但最近几年，Rails 开发生态系统不断成熟，已经出现很多好的解决方案了，包括使用 [Phusion Passenger](#)（Apache 和 Nginx¹⁴ Web 服务器的一个模块）的共享主机和虚拟私有服务器，[Engine Yard](#) 和 [Rails Machine](#) 这种提供全方位部署服务的公司，以及 [Engine Yard Cloud](#)、[Ninefold](#) 和 [Heroku](#) 这种云部署服务。

我最喜欢使用 Heroku 部署 Rails 应用。Heroku 专门用于部署 Rails 和其他 Web 应用，部署 Rails 应用的过程异常简单——只要源码纳入了 Git 版本控制系统就好。（这也是为什么要按照 1.4 节介绍的步骤设置 Git。如果你还没有照着做，现在赶紧做吧。）本节下面的内容专门介绍如何把我们的第一个应用部署到 Heroku 中。其中一些操作相对高级，如果没有完全理解也不要紧。本节的重点是把应用部署到线上环境中。

13. 如果你担心不小心公开了应用，可以参照 1.5.4 节中的做法。不过对本书开发应用来说，这份担心是多余的。

14. 读作“Engine X”。

1.5.1. 搭建 Heroku 部署环境

Heroku 使用 PostgreSQL¹⁵ 数据库，所以我们要把 pg 加入生产组，这样 Rails 才能和 PostgreSQL 通信：¹⁶

```
group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

注意，我们还添加了 rails_12factor，Heroku 使用这个 gem 伺服静态资源，例如图片和样式表。最终得到的 Gemfile 如[代码清单 1.14](#) 所示。

代码清单 1.14：增加 gem 后的 Gemfile

```
source 'https://rubygems.org'

gem 'rails',           '4.2.0.beta4'
gem 'sass-rails',      '5.0.0.beta1'
gem 'uglifier',         '2.5.3'
gem 'coffee-rails',     '4.0.1'
gem 'jquery-rails',     '4.0.0.beta2'
gem 'turbolinks',       '2.3.0'
gem 'jbuilder',          '2.2.3'
gem 'rails-html-sanitizer', '1.0.1'
gem 'sdoc',             '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',        '1.3.9'
  gem 'byebug',          '3.4.0'
  gem 'web-console',    '2.0.0.beta3'
  gem 'spring',          '1.1.3'
end

group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

为了准备好部署环境，下面要运行 bundle install 命令，并且指定一个特殊的选项，禁止在本地安装生产环境使用的 gem（即 pg 和 rails_12factor）：

```
$ bundle install --without production
```

因为我们在[代码清单 1.14](#) 中只添加了用于生产环境的 gem，所以现在执行这个命令其实不会在本地安装任何新的 gem，但是又必须执行这个命令，因为我们要把 pg 和 rails_12factor 添加到 Gemfile.lock 中。然后提交这次改动：

```
$ git commit -a -m "Update Gemfile.lock for Heroku"
```

15. 读作 post-gres-cue-ell，经常简称 Postgres。

16. 一般来说，开发环境和生产环境要尽量一致，包括使用相同的数据库。但在本书中，本地一直使用 SQLite，生产环境则使用 PostgreSQL。详情参见 [3.1 节](#)。

接下来我们要注册并配置一个 Heroku 新账户。第一步是[注册 Heroku 账户](#)。然后检查系统中是否已经安装 Heroku 命令行客户端：

```
$ heroku version
```

使用云端 IDE 的读者应该会看到 Heroku 客户端的版本号，这表明可以使用命令行工具 `heroku`。在其他系统中，可能需要使用 [Heroku Toolbelt](#) 安装。

确认 Heroku 命令行工具已经安装之后，使用 `heroku` 命令登录，然后添加 SSH 密钥：

```
$ heroku login  
$ heroku keys:add
```

最后，执行 `heroku create` 命令，在 Heroku 的服务器中创建一个文件夹，用于存放演示应用，如[代码清单 1.15](#) 所示。

代码清单 1.15：在 Heroku 中创建一个新应用

```
$ heroku create  
Creating damp-fortress-5769... done, stack is cedar  
http://damp-fortress-5769.herokuapp.com/ | git@heroku.com:damp-fortress-5769.git  
Git remote heroku added
```

`heroku` 命令会为你的应用分配一个二级域名，立即生效。当然，现在还看不到内容，我们开始部署吧。

1.5.2. Heroku 部署第一步

部署应用的第一步是，使用 Git 把主分支推送到 Heroku 中：

```
$ git push heroku master
```

(可能会看到一些提醒消息，现在先不管，[7.5 节](#)会解决。)

1.5.3. Heroku 部署第二步

其实没有第二步了。我们已经完成部署了。现在可以通过 `heroku create` 命令给出的地址（参见[代码清单 1.15](#)，如果没用云端 IDE，在本地可以执行 `heroku open` 命令）查看刚刚部署的应用，如[图 1.18](#) 所示。看到的页面和[图 1.12](#)一样，但是现在这个应用运行在生产环境中。

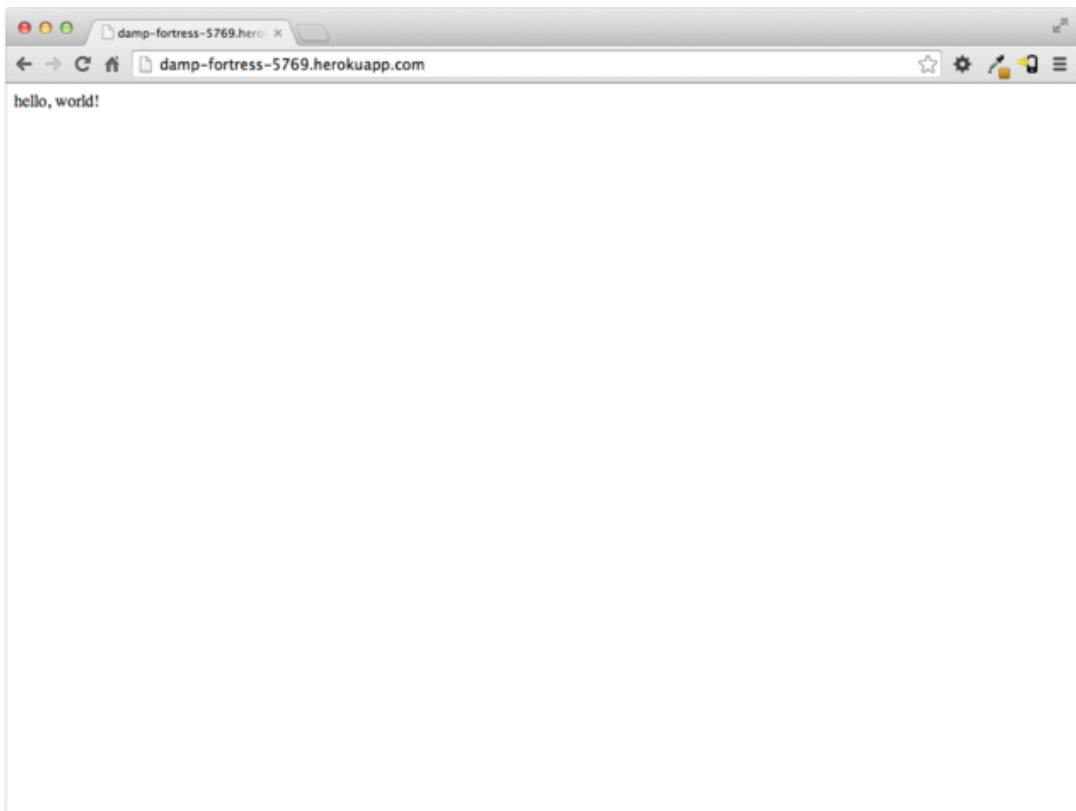


图 1.18：运行在 Heroku 中的第一个应用

1.5.4. Heroku 命令

Heroku 提供了[很多命令](#)，本书只简单介绍了几个。下面花几分钟再介绍一个命令，其作用是重命名应用：

```
$ heroku rename rails-tutorial-hello
```

你别再使用这个名字了，我已经占用了。或许，现在你无需做这一步，使用 Heroku 提供的默认地址就行。不过，如果你真想重命名应用，基于安全考虑，可以使用一些随机或难猜到的二级域名，例如：

```
hwpcbmze.herokuapp.com  
seyjhflo.herokuapp.com  
jhyicevg.herokuapp.com
```

使用这样随机的二级域名，只有你将地址告诉别人他们才能访问你的网站。顺便让你一窥 Ruby 的强大，下面是我用来生成随机二级域名的代码，很精妙吧。

```
('a'..'z').to_a.shuffle[0..7].join
```

除了支持二级域名，Heroku 还支持自定义域名。其实本书的网站¹⁷就放在 Heroku 中。如果你阅读的是在线版，现在就在浏览一个托管于 Heroku 中的网站。在 [Heroku 文档](#)中可以查看更多关于自定义域名的信息以及 Heroku 相关的其他话题。

17. 英文原版的网站托管在 Heroku 中，你现在阅读的中文版托管在 Github Pages 中，地址是 <http://railstutorial-china.org>。

1.6. 小结

这一章做了很多事：安装，搭建开发环境，版本控制以及部署。下一章会在这一章的基础上开发一个使用数据库的应用，让我们看看 Rails 真正的本事。

如果此时你想分享阅读本书的进度，可以发一条推文或者更新 Facebook 状态，写上类似下面的内容：

我正在阅读《Ruby on Rails 教程》学习 Ruby on Rails！

<http://railstutorial-china.org/>

建议你加入 [Rails 教程邮件列表](#)，以便及时收到本书重要的更新和优惠码。

1.6.1. 读完本章学到了什么

- Ruby on Rails 是一个使用 Ruby 编程语言开发的 Web 开发框架；
- 在预先配置好的云端环境中安装 Rails、新建应用，以及编辑文件都很简单；
- Rails 提供了命令行命令 `rails`，可用于新建应用（`rails new`）和启动本地服务器（`rails server`）；
- 添加了一个控制器动作，并且修改了根路由，最终开发出一个显示“hello, world!”的应用；
- 为了避免丢失数据，也为了协作，我们把应用的源码纳入 Git 版本控制系统，而且还把最终得到的代码推送到 Bitbucket 一个私有仓库中；
- 使用 Heroku 把应用部署到生产环境中。

1.7. 练习

1. 把 `hello` 动作（[代码清单 1.8](#)）中的“hello, world!”改成“holá, mundo!”。加分项：使用倒置的感叹号（例如“¡Hola, mundo!”中的第一个字符），证明 Rails 支持非 ASCII 字符。¹⁸ 结果如图 1.19 所示。
2. 按照编写 `hello` 动作的方式（[代码清单 1.8](#)），再添加一个动作，命名为 `goodbye`，渲染文本“goodbye, world!”。然后修改路由文件（[代码清单 1.10](#)），把根路由改成 `goodbye`。结果如图 1.20 所示。

18. 在你的编辑器中可能会显示一个消息，提示“invalid multibyte character”（无效的多字节字符），别去管它。如果你想让这个消息消失，可以在[谷歌中搜索这个错误消息](#)。

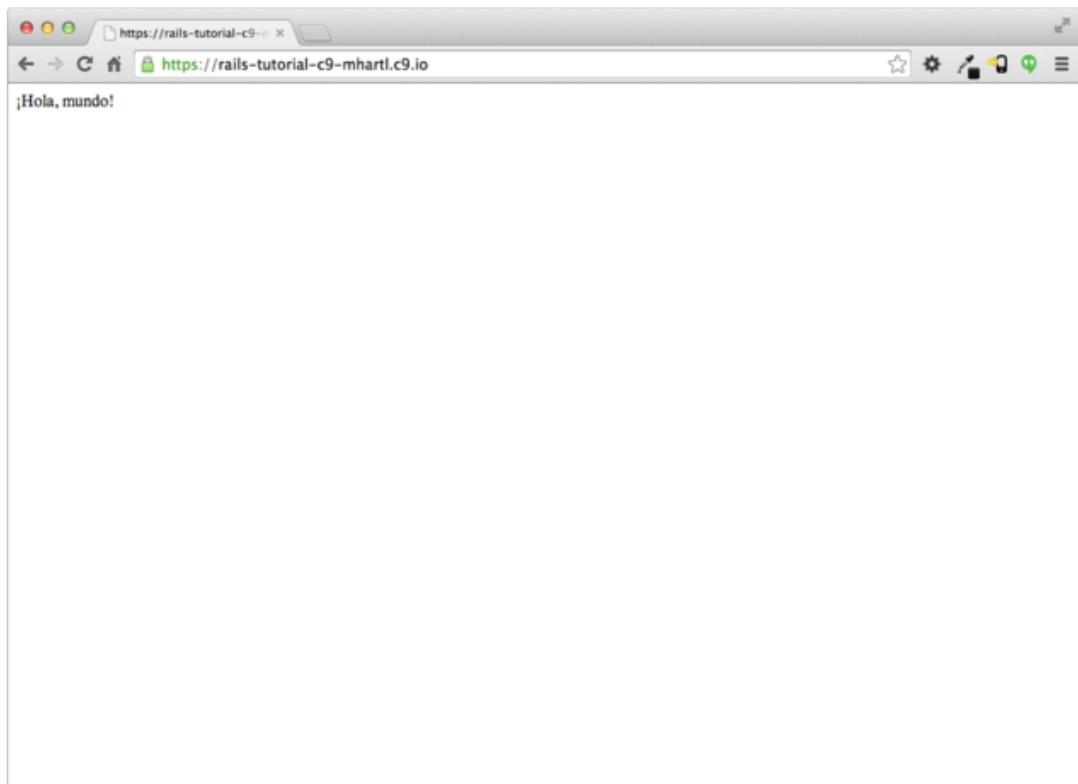


图 1.19：修改 hello 动作，显示文本“¡Hola, mundo!”

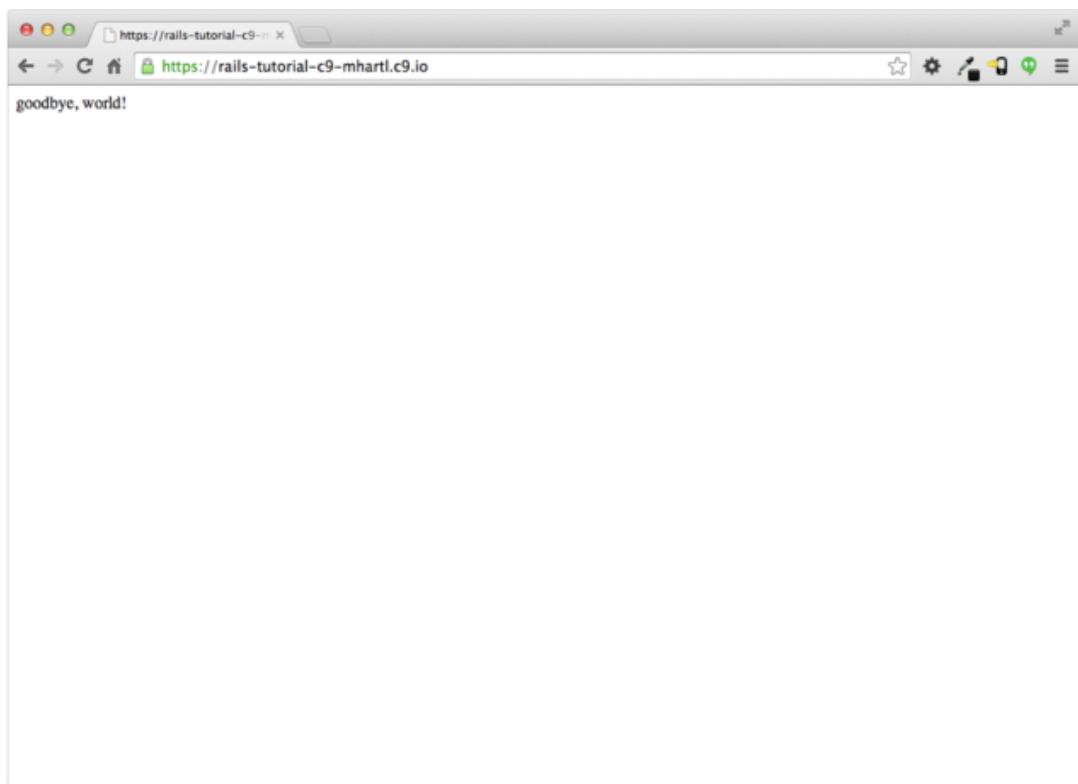


图 1.20：修改根路由，显示文本“goodbye, world!”

第 2 章 玩具应用

本章我们要开发一个简单的演示应用，展示 Rails 强大的功能。我们会使用脚手架快速生成程序，这样就能站在一定高度上概览 Ruby on Rails 编程的过程（也能大致了解 Web 开发）。正如[旁注 1.2](#) 中所说，本书将采用与众不同方法，循序渐进开发一个完整的演示应用，遇到新的概念都会详细说明。不过为了快速概览（也为了寻找成就感），无需对脚手架避而不谈。我们可以通过 URL 和最终开发出来的玩具应用交互，了解 Rails 应用的结构，也第一次演示 Rails 使用的 REST 架构。

和后面的演示应用类似，这个玩具应用中有用户（users）和用户的微博（microposts），因此算是一个小型的 Twitter 类应用。应用的功能还需要后续开发，而且开发过程中的很多步骤看起来很神秘，不过暂时不用担心：从[第 3 章](#)起将从零开始再开发一个类似的完整应用，我还会提供大量的资料供后续阅读。你要有些耐心，不要怕多犯错误，本章的主要目的就是让你不要被脚手架的神奇迷惑住了，而要更深入的了解 Rails。

2.1. 规划应用

这一节，我们要规划一下这个玩具应用。和[1.3 节](#)一样，我们先使用 `rails new` 命令生成应用的骨架。

```
$ cd ~/workspace  
$ rails _4.2.0.beta4_ new toy_app  
$ cd toy_app/
```

如果执行 `rails new` 命令后看到“Could not find ‘railties’”这样的错误，说明你安装的 Rails 版本不对。再次确认安装 Rails 时执行的命令和[代码清单 1.1](#) 一模一样。（注意，如果使用[1.2.1 节](#)推荐的云端 IDE，这个应用可以在第一个应用所在的工作空间中创建，没必要再新建一个工作空间。如果没看到文件，可以点击文件浏览器中的齿轮图标，然后选择“Refresh File Tree”（刷新文件树）。）

然后，在文本编辑器中修改 `Gemfile`，写入[代码清单 2.1](#) 中的内容。

代码清单 2.1：这个玩具应用的 `Gemfile`

```
source 'https://rubygems.org'  
  
gem 'rails', '4.2.0.beta4'  
gem 'sass-rails', '5.0.0.beta1'  
gem 'uglifier', '2.5.3'  
gem 'coffee-rails', '4.0.1'  
gem 'jquery-rails', '4.0.0.beta2'  
gem 'turbolinks', '2.3.0'  
gem 'jbuilder', '2.2.3'  
gem 'rails-html-sanitizer', '1.0.1'  
gem 'sdoc', '0.4.0', group: :doc  
  
group :development, :test do  
  gem 'sqlite3', '1.3.9'  
  gem 'byebug', '3.4.0'  
  gem 'web-console', '2.0.0.beta3'  
  gem 'spring', '1.1.3'
```

```

end

group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
end

```

注意，[代码清单 2.1](#) 和[代码清单 1.14](#) 的内容一样。

和 [1.5.1 节](#)一样，安装 gem 时要指定 `--without production` 选项，不安装生产环境所需的 gem:

```
$ bundle install --without production
```

最后，把这个玩具应用纳入 Git 版本控制系统:

```

$ git init
$ git add -A
$ git commit -m "Initialize repository"

```

你还可以在 Bitbucket 网站中点击“Create”（新建）按钮[创建一个新仓库](#)（图 2.1），然后把代码推送到这个远程仓库中:

```

$ git remote add origin git@bitbucket.org:<username>/toy_app.git
$ git push -u origin --all # 首次推送这个仓库

```

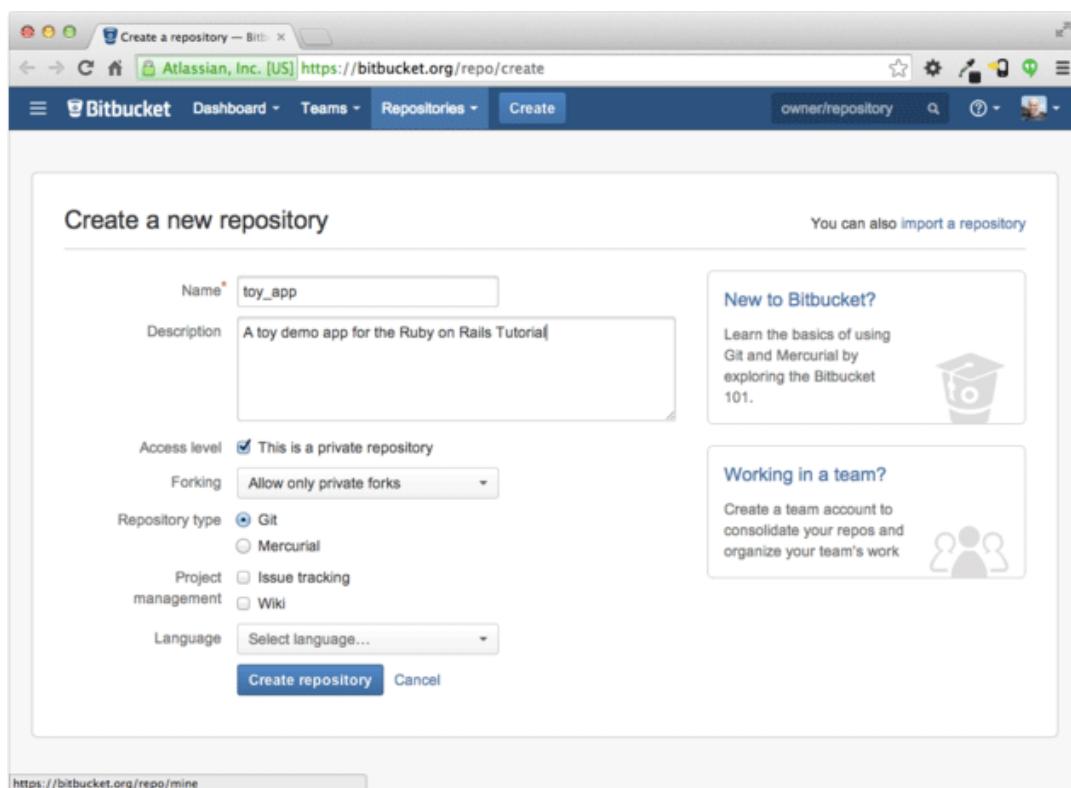


图 2.1：在 Bitbucket 中为这个玩具应用创建一个仓库

越早部署应用越好。我建议把[代码清单 1.8](#) 和[代码清单 1.9](#) 中的内容复制过来，¹ 然后提交改动，再推送到 Heroku:

```
$ git commit -am "Add hello"  
$ heroku create  
$ git push heroku master
```

(和 1.5 节一样，可能会看到一些提醒消息，现在先不去管它。7.5 节会解决。) 除了 Heroku 为应用提供的地址之外，输出的内容应该和图 1.18 一样。

下面要开发这个应用了。一般来说，开发 Web 应用的第一步是创建数据模型（data model）。模型表示应用所需的结构。这个玩具应用是个微博客，只有用户和简短的文章（微博）。那么我们先为这个应用添加用户模型（2.1.1 节），然后再添加微博模型（2.1.2 节）。

2.1.1. 用户模型

网络中有多少不同的注册表单，就有多少定义用户数据模型的方式。我们要使用一种最简单的。这个玩具应用的用户有一个唯一的标识 `id` (`integer` 类型)，一个公开的名字 `name` (`string` 类型)，以及一个电子邮件地址 `email` (也是 `string` 类型)。电子邮件地址也作为用户名使用。用户模型的结构如图 2.2。

users	
<code>id</code>	<code>integer</code>
<code>name</code>	<code>string</code>
<code>email</code>	<code>string</code>

图 2.2：用户数据模型

在 6.1.1 节会看到，图 2.2 中的 `users` 对应于数据库中的一个表；`id`、`name` 和 `email` 是表中的列。

2.1.2. 微博模型

微博数据模型的核心比用户模型还要简单：微博只要一个 `id` 和表示微博内容的 `content` (`text` 类型) 字段即可。² 不过还有一个比较复杂的字段要实现，这个字段把微博和用户关联起来。我们使用 `user_id` 存储微博的属主。最终得到的微博数据模型如图 2.3 所示。

microposts	
<code>id</code>	<code>integer</code>
<code>content</code>	<code>text</code>
<code>user_id</code>	<code>integer</code>

图 2.3：微博数据模型

2.3.3 节会介绍怎样使用 `user_id` 字段简单的实现一个用户拥有多个微博的功能。在第 11 章中有更完整的说明。

-
- 之所以这么做是因为，Rails 提供的默认页面在 Heroku 中不能正常显示，因此很难判断部署是否成功。
 - 微博的内容很短，`string` 类型就足够了，但使用 `text` 类型更能表明我们的意图，而且也便于以后放宽微博长度限制。

2.2. 用户资源

这一节我们要实现 2.1.1 节设定的用户数据模型，还会为这个模型创建 Web 界面。二者结合起来就是一个“用户资源”（Users Resource）。“资源”的意思是把用户设想为对象，可以通过 [HTTP 协议](#) 在网页中创建（create）、读取（read）、更新（update）和删除（delete）。正如前面提到的，用户资源使用 Rails 内置的脚手架生成。我建议你先不要细看脚手架生成的代码，这时看只会让你更困惑。

把 scaffold 传给 rails generate 就可以使用 Rails 的脚手架了。传给 scaffold 的参数是资源名的单数形式（这里是 User）³，后面可以再跟着一些可选参数，指定数据模型中的字段：

```
$ rails generate scaffold User name:string email:string
  invoke  active_record
  create    db/migrate/20140821011110_create_users.rb
  create    app/models/user.rb
  invoke  test_unit
  create    test/models/user_test.rb
  create    test/fixtures/users.yml
  invoke  resource_route
  route    resources :users
  invoke  scaffold_controller
  create    app/controllers/users_controller.rb
  invoke  erb
  create    app/views/users
  create    app/views/users/index.html.erb
  create    app/views/users/edit.html.erb
  create    app/views/users/show.html.erb
  create    app/views/users/new.html.erb
  create    app/views/users/_form.html.erb
  invoke  test_unit
  create    test/controllers/users_controller_test.rb
  invoke  helper
  create    app/helpers/users_helper.rb
  invoke  test_unit
  create    test/helpers/users_helper_test.rb
  invoke  jbuilder
  create    app/views/users/index.json.jbuilder
  create    app/views/users/show.json.jbuilder
  invoke  assets
  invoke  coffee
  create    app/assets/javascripts/users.js.coffee
  invoke  scss
  create    app/assets/stylesheets/users.css.scss
  invoke  scss
  create    app/assets/stylesheets/scaffolds.css.scss
```

我们在执行的命令中加入了 name:string 和 email:string，这样就可以实现图 2.2 中的用户模型了。注意，没有必要指定 id 字段，Rails 会自动创建并将其设为表的主键（primary key）。

接下来我们要用 Rake（参见 [旁注 2.1](#)）来迁移（migrate）数据库：

3. 脚手架中使用的名字和模型一致，是单数；而资源和控制器使用复数。因此，这里要使用 User，不是 Users。

```
$ bundle exec rake db:migrate
-- CreateUsers: migrating =====
-- create_table(:users)
-> 0.0017s
-- CreateUsers: migrated (0.0018s) =====
```

上面的命令会使用新定义的用户数据模型更新数据库。（[6.1.1 节](#)会详细介绍数据库迁移）注意，为了使用 `Gemfile` 中指定的 Rake 版本，我们要通过 `bundle exec` 执行 `rake`。在很多系统中，包括云端 IDE，都不必使用 `bundle exec`，但某些系统必须使用，所以为了命令的完整，我会一直使用 `bundle exec`。

然后，执行下面的命令，在另一个选项卡中运行本地 Web 服务器（[图 1.7](#)）：⁴

```
$ rails server -b $IP -p $PORT # 在本地设备中只需执行 `rails server`
```

现在，这个玩具应用应该可以通过本地服务器访问了（[1.3.2 节](#)）。如果使用云端 IDE，要在一个新的浏览器选项卡中打开网页，别在 IDE 中打开。

旁注 2.1: Rake

在 Unix 中，把源码编译成可执行的程序时，`make` 扮演了很重要的角色。很多程序员的身体甚至已经对下面的代码产生了条件反射：

```
$ ./configure && make && sudo make install
```

在 Unix 中（包括 Linux 和 Mac OS X），这个命令一般用来编译代码。

Rake 是 Ruby 版的 `make`，用 Ruby 语言编写的类 `make` 程序。Rails 灵活的运用了 Rake 的功能，提供了很多开发基于数据库的 Web 应用所需的管理任务。`rake db:migrate` 或许是最常用的。除此之外还有很多其他命令，运行 `rake -T db` 可以查看所有数据库相关的任务：

```
$ bundle exec rake -T db
```

如果想查看所有 Rake 任务，运行：

```
$ bundle exec rake -T
```

任务列表看起来有点让人摸不着头脑，不过现在无需担心，你不需要知道所有（甚至大多数）命令。学完本教程后，你会知道所有重要的任务。

2.2.1. 浏览用户相关的页面

如果访问根 URL <http://localhost:3000/> 看到的还是 Rails 默认页面（[图 1.9](#)）。不过使用脚手架生成用户资源时生成了很多用来处理用户的页面。例如，列出所有用户的页面地址是 `/users`，创建新用户的地址是 `/users/new`。本节的目的就是走马观花地浏览一下这些用户相关的页面。浏览时你会发现表 [表 2.1](#) 很有用，表中显示了页面和 URL 之间的对应关系。

4. `rails` 脚本经过特殊处理，无需使用 `bundle exec`。

表 2.1：用户资源中页面和 URL 的对应关系

URL	动作	作用
/users	index	列出所有用户
/users/1	show	显示 ID 为 1 的用户
/users/new	new	创建新用户
/users/1/edit	edit	编辑 ID 为 1 的用户

我们先来看一下显示所有用户的页面，这个页面叫“索引页”。和预期一样，目前还没有用户，如图 2.4 所示。

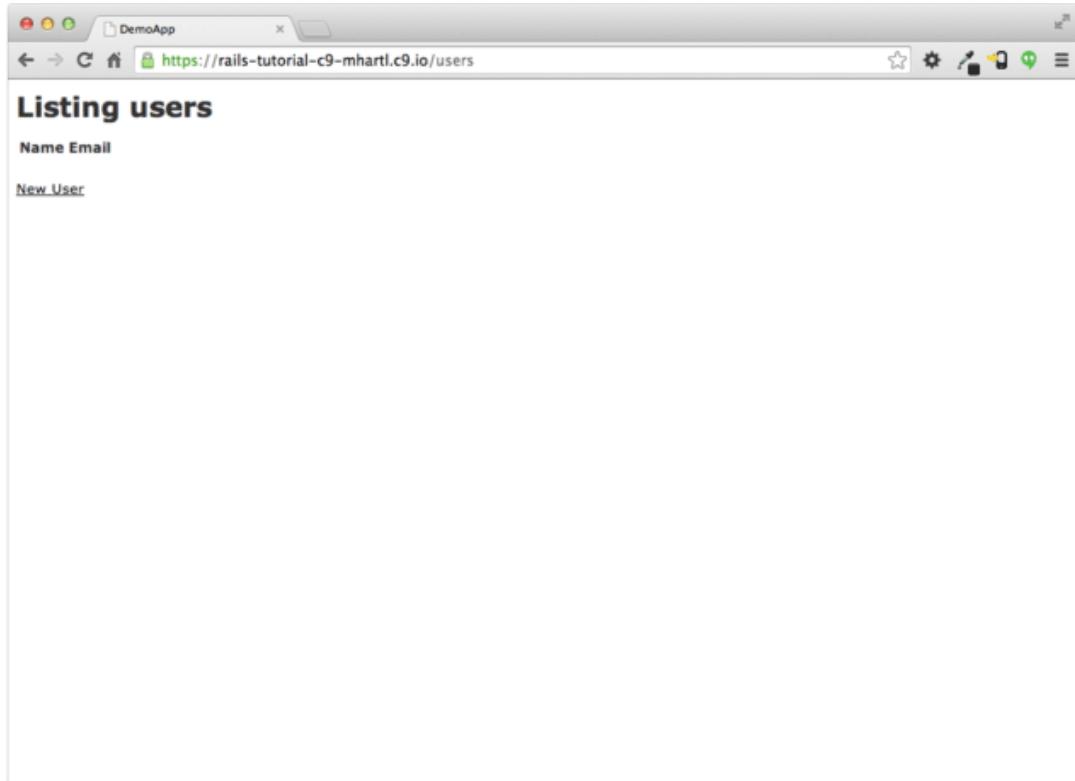


图 2.4：用户资源的索引页（/users）

如果想创建新用户要访问“[新建用户](#)”页面，如图 2.5 所示。（在本地开发时，地址的前面部分都是 <http://localhost:3000> 或云端 IDE 分配的地址，因此在后面的内容中我会省略这一部分。）第 7 章会把这个页面改造成用户注册页面。

我们可以在表单中填入名字和电子邮件地址，然后点击“Create User”（创建用户）按钮创建一个用户。然后就会显示这个用户的页面，如图 2.6 所示。页面中显示的绿色文字是“闪现消息”（flash message），7.4.2 节会介绍。注意，这个页面的 URL 是 /users/1。你可能猜到了，这里的 1 就是图 2.2 中的用户 id。7.1 节会把这个页面打造成用户的资料页。

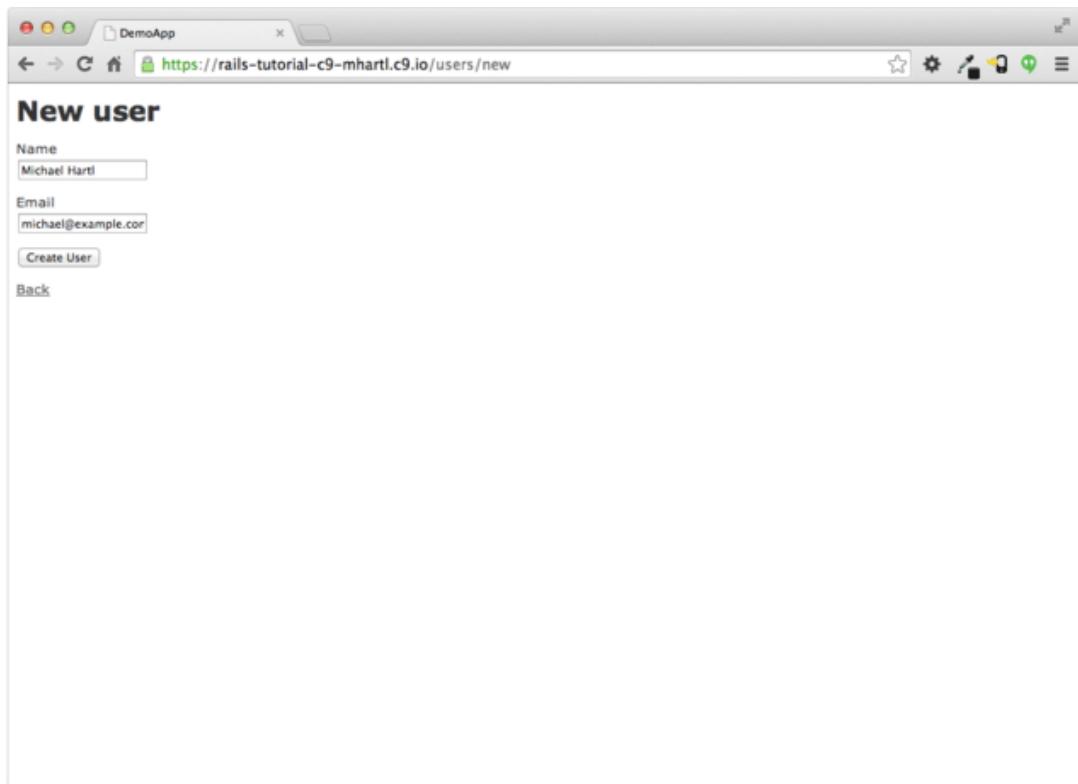


图 2.5：新建用户页面（/users/new）

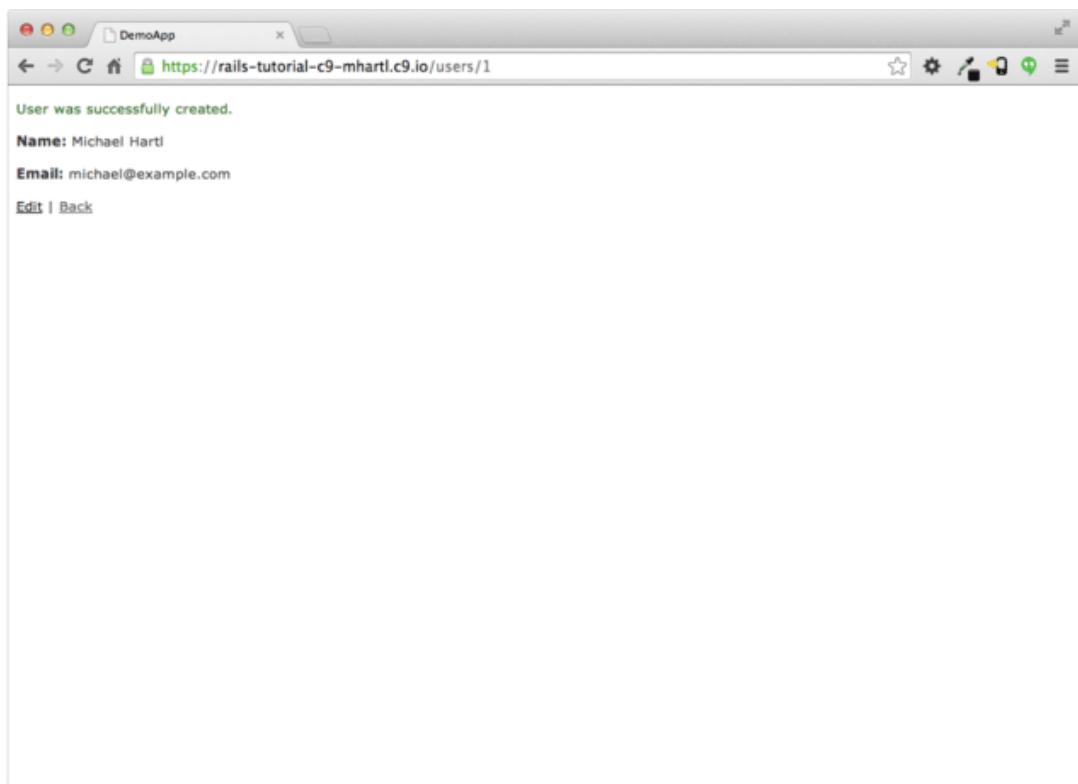


图 2.6：显示某个用户的页面（/users/1）

如果想修改用户的信息，要访问“编辑页面”（图 2.7）。修改用户信息后点击“Update User”（更新用户）按钮就更改了这个玩具应用中该用户的信息（图 2.8）。第 6 章会详细介绍，用户的信息存储在后端的数据库中。我们会在 9.1 节为演示应用添加编辑和更新用户信息的功能。

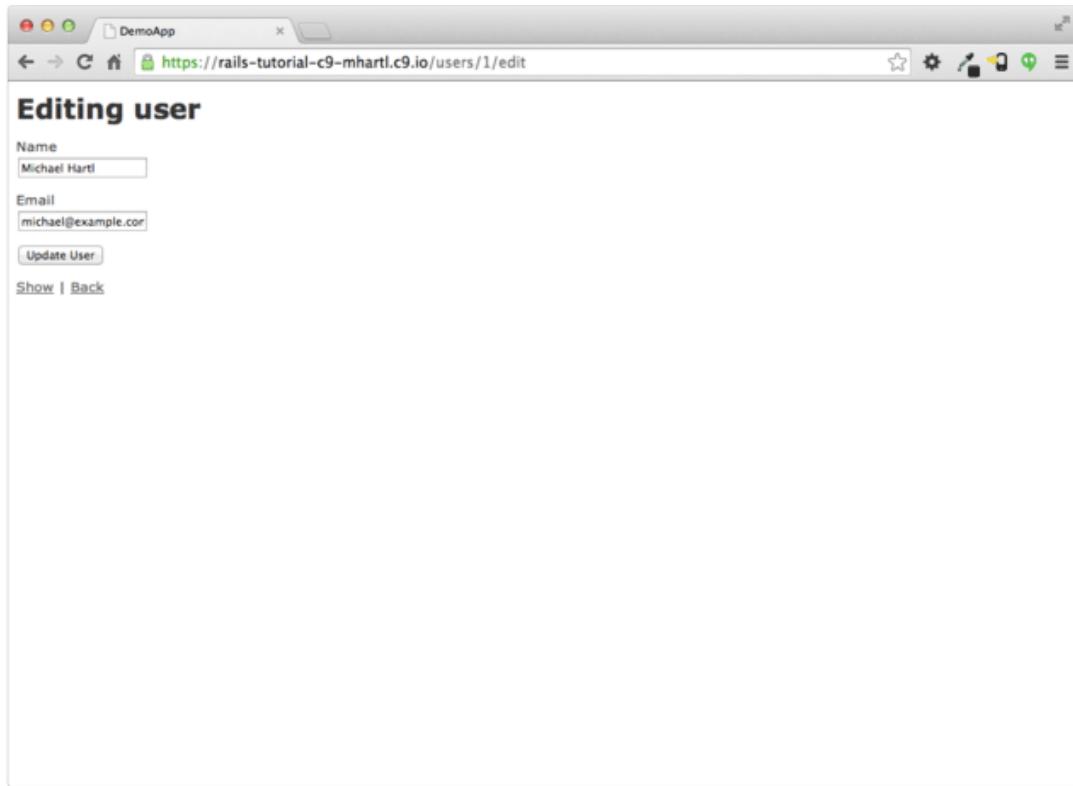


图 2.7：编辑用户信息的页面（/users/1/edit）

现在回到创建新用户的页面，提交表单创建第二个用户。然后访问用户索引页，结果如图 2.9 所示。7.1 节会美化这个显示所有用户的页面。

我们已经看了创建、显示和编辑用户的页面，最后要看删除用户的页面（图 2.10）。点击图 2.10 中所示的链接后，会删除第二个用户，索引页面就只剩一个用户了。如果这个操作不成功，确认浏览器是否启用了 JavaScript。Rails 通过 JavaScript 发起删除用户的请求。9.4 节会为演示应用实现用户删除功能，而且仅限于管理员级别的用户才能执行这项操作。

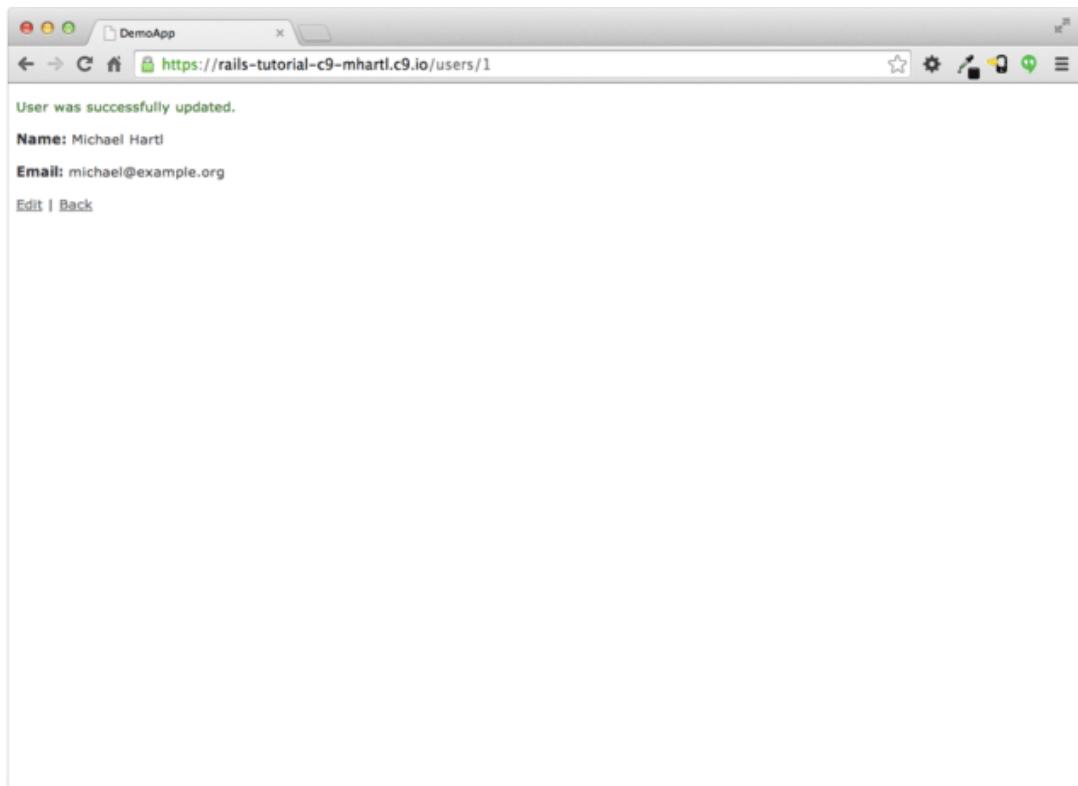


图 2.8：更新信息后的用户页面

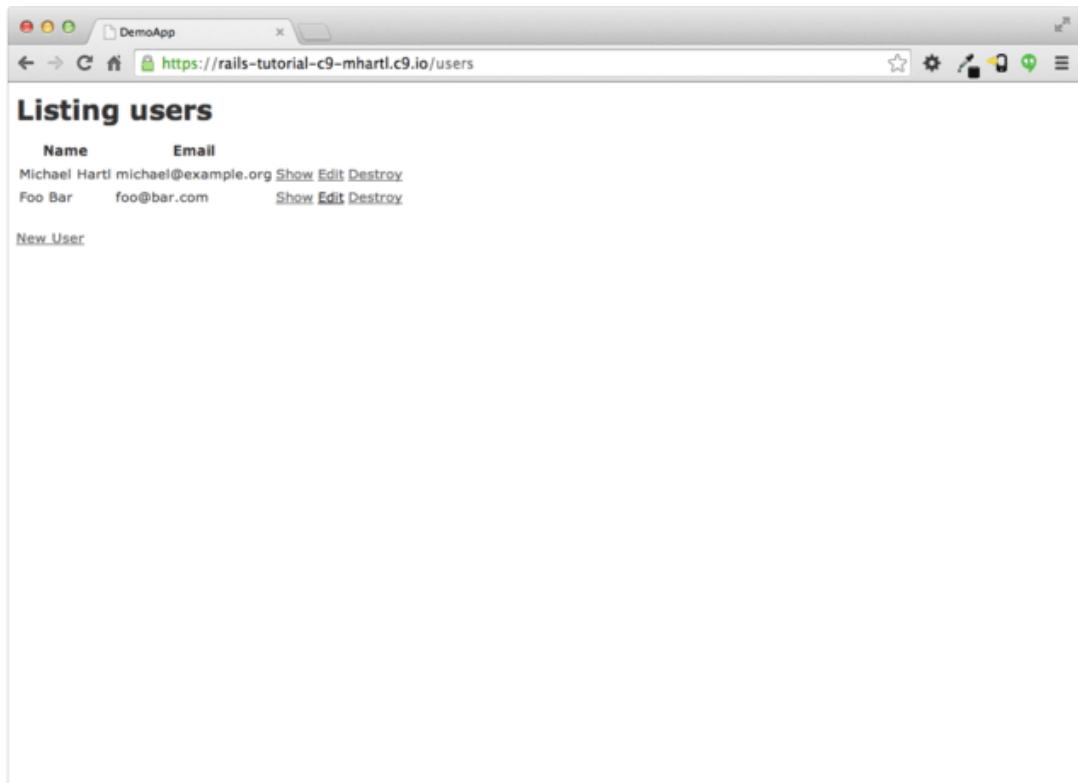


图 2.9：创建第二个用户后的用户索引页（/users）

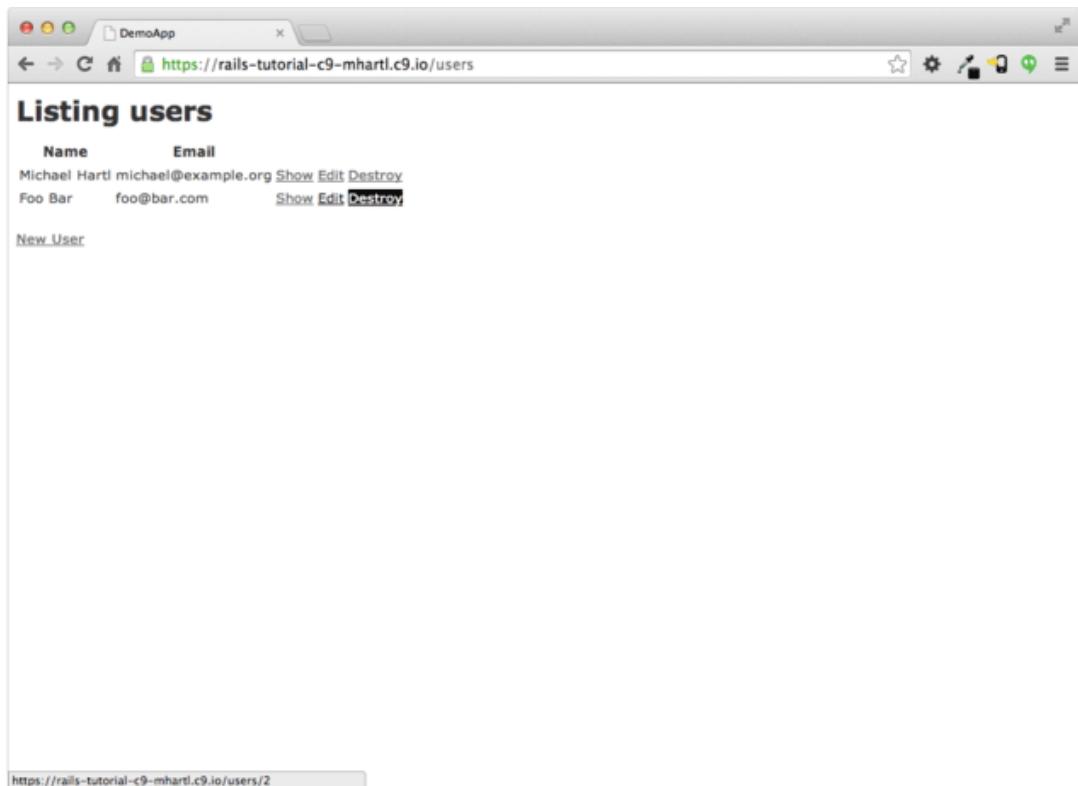


图 2.10：删除一个用户

2.2.2. MVC 实战

我们已经快速概览了用户资源，下面我们从 MVC（[1.3.3 节](#)）的视角出发，审视其中某些特定部分。我们要分析在浏览器中访问用户索引页的过程，了解一下 MVC（[图 2.11](#)）。

图中各步的说明如下：

1. 浏览器向 /users 发起一个请求；
2. Rails 的路由把 /users 交给 UsersController 中的 index 动作处理；
3. index 动作要求用户模型读取所有用户 (`User.all`)；
4. 用户模型从数据库中读取所有用户；
5. 用户模型把所有用户组成的列表返回给控制器；
6. 控制器把所有用户赋值给 `@users` 变量，然后传入 index 视图；
7. 视图使用嵌入式 Ruby 把页面渲染成 HTML；
8. 控制器把 HTML 发送回浏览器。⁵

5. 有些文章说视图直接把 HTML 返回给浏览器（通过 Web 服务器，例如 Apache 或 Nginx）。不管实现的细节如何，我更相信控制器是一个中枢，应用中所有信息都会经由它传递。

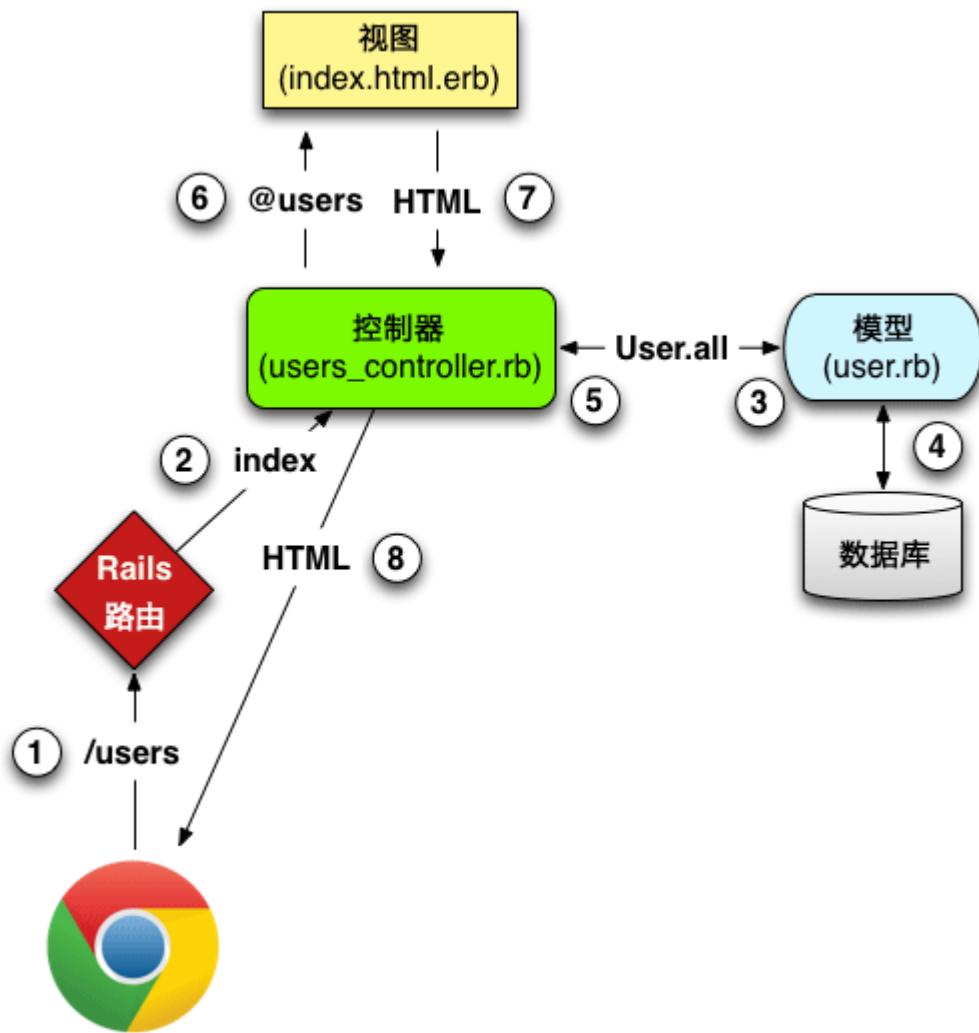


图 2.11: Rails 中的 MVC 架构详解

下面详细分析这个过程。首先，从浏览器中发起一个请求（第 1 步）。可以直接在浏览器地址栏中输入地址，也可以点击网页中的链接。请求到达 Rails 路由（第 2 步），根据 URL（以及请求的类型，参见[旁注 3.2](#)）将其分发给合适的控制器动作。把用户资源中相关的 URL 映射到控制器动作的代码如[代码清单 2.2](#) 所示。这行代码会按照[表 2.1](#) 中的对应关系做映射。`:users` 这个符号很奇怪，它是一个符号（Symbol），[4.3.3](#) 节会介绍。

代码清单 2.2: Rails 路由，其中定义了用户资源的规则

`config/routes.rb`

```
Rails.application.routes.draw do
  resources :users
  .
  .
  .
end
```

既然打开了路由文件，那就花点儿时间把根路由改为用户索引页吧，修改之后，访问根地址就会显示 /users 页面。在[代码清单 1.10](#) 中，我们把

```
# root 'welcome#index'
```

改成了

```
root 'application#hello'
```

让根路由指向 `ApplicationController` 中的 `hello` 动作。现在我们想使用 `UsersController` 中的 `index` 动作，要按照[代码清单 2.3](#) 所示的方式修改。如果本章开头在 `ApplicationController` 中添加了 `hello` 动作，我建议现在把这个动作删除。

代码清单 2.3: 把根路由指向 `UsersController` 中的动作

config/routes.rb

```
Rails.application.routes.draw do
  resources :users
  root 'users#index'

  .
  .
  .

end
```

[2.2.1 节](#) 中浏览的页面对应于 `UsersController` 中的不同动作。脚手架生成的控制器代码摘要如[代码清单 2.4](#) 所示。注意 `class UsersController < ApplicationController` 这种写法，在 Ruby 中表示类继承。[2.3.4 节](#) 会简要介绍继承，[4.4 节](#) 再做详细介绍。

代码清单 2.4: 用户控制器代码摘要

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def index
  .
  .
  .
  end

  def show
  .
  .
  .
  end

  def new
  .
  .
  .
  end

  def edit
  .
  .
  .
  end
```

```

end

def create
  .
  .
  .

end

def update
  .
  .
  .

end

def destroy
  .
  .

end
end

```

你可能注意到了，动作的数量比我们看过的页面数量多，`index`、`show`、`new` 和 `edit` 对应于 2.2.1 节介绍的页面。不过还有一些其他动作，`create`、`update` 和 `destroy` 等。这些动作一般不直接渲染页面（不过有时也会），只会修改数据库中保存的用户数据。表 2.2 列出了控制器的全部动作，这些动作就是 Rails 对 REST 架构（参见旁注 2.2）的实现。REST 架构由计算机科学家 Roy Fielding 提出，意思是“表现层状态转化”（Representational State Transfer）。⁶ 注意表 2.2 中的内容，有些部分有重叠。例如 `show` 和 `update` 两个动作都映射到 `/users/1` 这个地址上。二者的区别是，使用的 HTTP 请求方法不同。3.3 节会更详细地介绍 HTTP 请求方法。

表 2.2：代码清单 2.2 生成的符合 REST 架构的路由

HTTP 请求	URL	动作	作用
GET	/users	<code>index</code>	列出所有用户
GET	/users/1	<code>show</code>	显示 ID 为 1 的用户
GET	/users/new	<code>new</code>	显示创建新用户页面
POST	/users	<code>create</code>	创建新用户
GET	/users/1/edit	<code>edit</code>	显示编辑 ID 为 1 的用户页面
PATCH	/users/1	<code>update</code>	更新 ID 为 1 的用户
DELETE	/users/1	<code>destroy</code>	删除 ID 为 1 的用户

6. 加州大学欧文分校 2000 年 Roy Thomas Fielding 的博士论文《Architectural Styles and the Design of Network-based Software Architectures》。

旁注2.2：表现层状态转化（REST）

如果你阅读过一些 Ruby on Rails Web 开发相关的资料，会看到很多地方都提到了“REST”，它是“表现层状态转化”（REpresentational State Transfer）的简称。REST 是一种架构方式，用来开发分布式、基于网络的系统和软件程序，例如 WWW 和 Web 应用。REST 理论很抽象，在 Rails 应用中，REST 意味着大多数组件（例如用户和微博）都会被模型化，变成资源（resource），可以创建（create）、读取（read）、更新（update）和删除（delete）。这些操作与[关系型数据库中的 CRUD 操作](#)和[HTTP 请求方法](#)（POST, GET, PATCH⁷ 和 DELETE）对应。[3.3 节](#)，特别是[旁注 3.2](#)，将更详细地介绍 HTTP 请求。

作为 Rails 应用开发者，REST 开发方式能帮助你决定编写哪些控制器和动作：你只需简单的把可以创建、读取、更新和删除的资源理清就可以了。对本章的“用户”和“微博”来说，这一过程非常明确，因为它们都是很自然的资源形式。在[第 12 章](#)将看到，使用 REST 架构可以通过一种自然而便捷的方式解决很棘手的问题（“关注用户”功能）。

为了探明用户控制器和用户模型之间的关系，我们看一下简化后的 index 动作，如[代码清单 2.5](#) 所示。（脚手架生成的代码很粗糙，所以我做了简化。）

代码清单 2.5：这个玩具应用中简化后的 index 动作

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

index 动作中有一行代码，`@users = User.all`（[图 2.11](#) 中的第 3 步），要求用户模型从数据库中取出所有用户（第 4 步），然后把结果赋值给 `@users` 变量（读作“at-users”，第 5 步）。用户模型的代码参见[代码清单 2.6](#)。代码看似简单，但是通过继承具备了很多功能（参见 [2.3.4 节](#) 和 [4.4 节](#)）。具体而言，调用 Rails 中的 Active Record 库后，`User.all` 就能获取数据库中的所有用户。

代码清单 2.6：玩具应用中的用户模型

app/models/user.rb

```
class User < ActiveRecord::Base
end
```

定义 `@users` 变量后，控制器再调用视图（第 6 步）。视图的代码如[代码清单 2.7](#) 所示。以 @ 开头的变量是“实例变量”（instance variable），在视图中自动可用。在本例中，`index.html.erb` 视图的代码（[代码清单 2.7](#)）遍历 `@users`，为每个用户生成一行 HTML。（你现在可能读不懂这些代码，这里只是让你看一下视图代码是什么样子。）

7. Rails 早期版本使用 PUT 请求更新数据，但是根据 HTTP 标准，PATCH 方法更合适。

代码清单 2.7：用户索引页的视图代码

app/views/users/index.html.erb

```
<h1>Listing users</h1>



| Name             | Email             |                      |                      |                                                      |
|------------------|-------------------|----------------------|----------------------|------------------------------------------------------|
| <%= user.name %> | <%= user.email %> | <a href="#">Show</a> | <a href="#">Edit</a> | <a data-confirm="Are you sure?" href="#">Destroy</a> |

<%= link_to 'New User', new_user_path %>
```

视图把代码转换成 HTML（第 7 步），然后控制器将其返回给浏览器，再显示出来（第 8 步）。

2.2.3. 这个用户资源的不足

脚手架生成的用户资源虽然能够让你大致了解 Rails，但也有一些不足：

- 没验证数据。用户模型会接受空名字和无效的电子邮件地址，而不报错。
- 没有认证机制。没实现登录和退出功能，随意一个用户都可以进行任何操作。
- 没有测试。也不是完全没有，脚手架会生成一些基本的测试，不过很粗糙也不灵便，没有针对数据验证和认证的测试，更别说针对其他功能的测试了。
- 没样式，没布局。没有共用的样式和网站导航。
- 没真正理解。如果你能读懂脚手架生成的代码，就不需要阅读这本书了。

2.3. 微博资源

我们已经生成并浏览了用户资源，现在要生成微博资源。阅读本节时，我推荐你和 2.2 节对比一下。你会发现两个资源在很多方面都是一致的。通过这样重复生成资源，我们可以更好地理解 Rails 中的 REST 架构。在这样的早期阶段看一下用户资源和微博资源的相同之处，也是本章的主要目的之一。

2.3.1. 概览微博资源

和用户资源一样，我们使用 `rails generate scaffold` 命令生成微博资源的代码，不过这一次要实现图 2.3 中的数据模型：⁸

```
$ rails generate scaffold Micropost content:text user_id:integer
  invoke  active_record
  create    db/migrate/20140821012832_create_microposts.rb
  create    app/models/micropost.rb
  invoke  test_unit
  create    test/models/micropost_test.rb
  create    test/fixtures/microposts.yml
  invoke  resource_route
    route    resources :microposts
  invoke  scaffold_controller
  create    app/controllers/microposts_controller.rb
  invoke  erb
    create      app/views/microposts
    create      app/views/microposts/index.html.erb
    create      app/views/microposts/edit.html.erb
    create      app/views/microposts/show.html.erb
    create      app/views/microposts/new.html.erb
    create      app/views/microposts/_form.html.erb
  invoke  test_unit
    create      test/controllers/microposts_controller_test.rb
  invoke  helper
    create      app/helpers/microposts_helper.rb
  invoke  test_unit
    create      test/helpers/microposts_helper_test.rb
  invoke  jbuilder
    create      app/views/microposts/index.json.jbuilder
    create      app/views/microposts/show.json.jbuilder
  invoke  assets
  invoke  coffee
    create      app/assets/javascripts/microposts.js.coffee
  invoke  scss
    create      app/assets/stylesheets/microposts.css.scss
  invoke  scss
  identical   app/assets/stylesheets/scaffolds.css.scss
```

如果看到 Spring 相关的错误，再次执行这个命令即可。

然后，和 2.2 节一样，我们要执行迁移，更新数据库，使用新建的数据模型：

```
$ bundle exec rake db:migrate
==  CreateMicroposts: migrating =====
-- create_table(:microposts)
  -> 0.0023s
==  CreateMicroposts: migrated (0.0026s) =====
```

8. 和生成用户资源使用的脚手架命令一样，生成微博资源的脚手架也要使用单数形式，因此要用 `generate Micropost`。

现在我们就可以使用类似 2.2.1 节中介绍的方法来创建微博了。你可能猜到了，脚手架还会更新 Rails 的路由文件，为微博资源加入一条规则，如[代码清单 2.8](#) 所示。⁹ 和用户资源类似，`resources :microposts` 把微博相关的 URL 映射到 `MicropostsController`，如表 2.3 所示。

代码清单 2.8：Rails 的路由，有一条针对微博资源的新规则

`config/routes.rb`

```
Rails.application.routes.draw do
  resources :microposts
  resources :users
  .
  .
  .
end
```

表 2.3：[代码清单 2.8](#) 中微博资源生成的符合 REST 架构的路由

HTTP 请求	URL	动作	作用
GET	/microposts	<code>index</code>	列出所有微博
GET	/microposts/1	<code>show</code>	显示 ID 为 1 的微博
GET	/microposts/new	<code>new</code>	显示创建新微博的页面
POST	/microposts	<code>create</code>	创建新微博
GET	/microposts/1/edit	<code>edit</code>	显示编辑 ID 为 1 的微博页码
PATCH	/microposts/1	<code>update</code>	更新 ID 为 1 的微博
DELETE	/microposts/1	<code>destroy</code>	删除 ID 为 1 的微博

`MicropostsController` 的代码简化后如[代码清单 2.9](#) 所示。注意，除了把 `UsersController` 换成 `MicropostsController` 之外，这段代码和[代码清单 2.4](#) 没什么区别。这说明了两个资源在 REST 架构中的共同之处。

代码清单 2.9：简化后的 `MicropostsController`

`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  .
  .
  .
  def index
  .
  .
  .
  end

  def show
  .
  .
  .
end
```

9. 脚手架生成的代码和[代码清单 2.8](#) 相比，可能会有额外的空行。无须担心，因为 Ruby 会忽略额外的空行。

```
    .
    .
    .
end

def new
    .
    .
    .
end

def edit
    .
    .
    .
end

def create
    .
    .
    .
end

def update
    .
    .
    .
end

def destroy
    .
    .
    .
end
end
```

我们在发布微博的页面（/microposts/new）输入一些内容，发布一篇微博，如图 2.12 所示。

既然已经打开这个页面了，那就多发布几篇微博，并且确保至少把一篇微博的 user_id 设为 1，把微博赋予 2.2.1 节中创建的第一个用户。结果应该和图 2.13 类似。

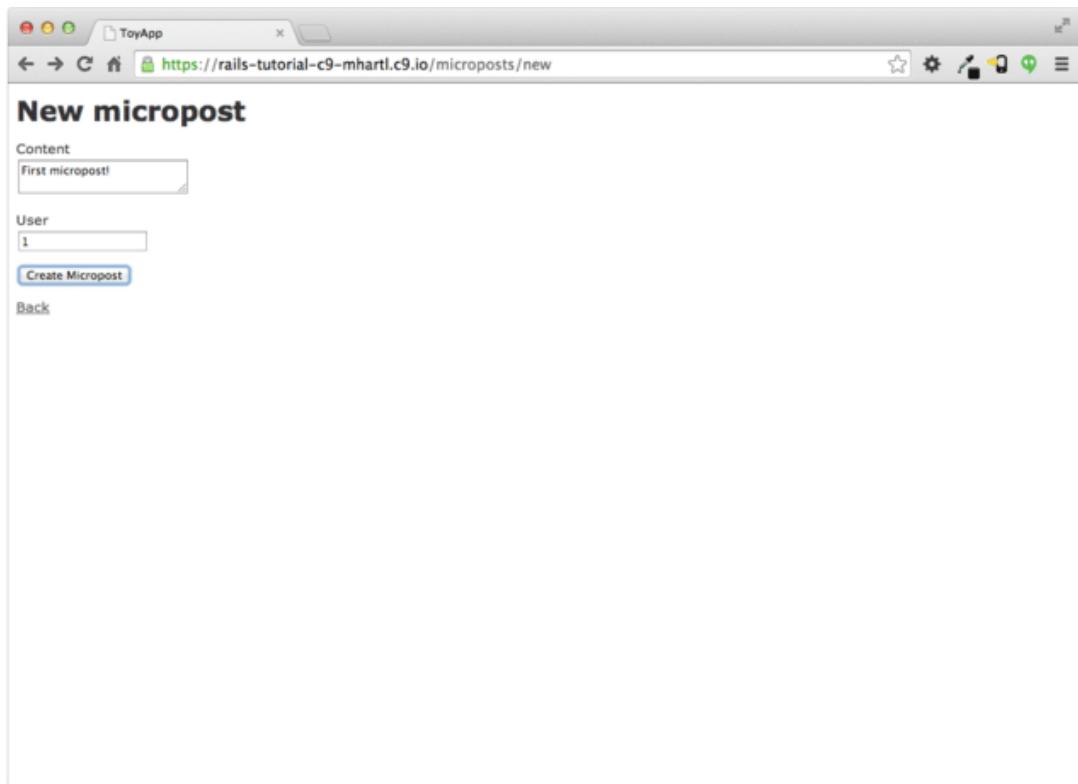


图 2.12：发布微博的页面

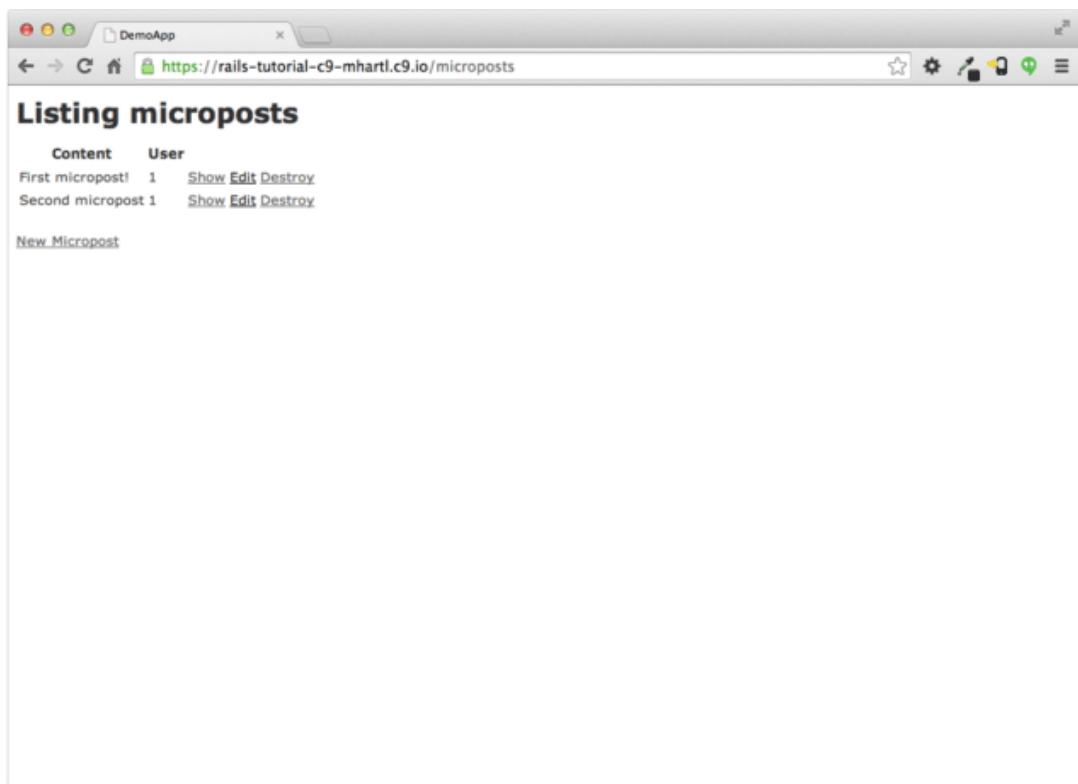


图 2.13：微博索引页（/microposts）

2.3.2. 限制微博内容的长度

如果要称得上“微博”这个名字，就要限制内容的长度。在 Rails 中实现这种限制很简单，使用验证（validation）功能即可。要限制微博的长度最大字数为 140 个字符（就像 Twitter 一样），我们可以使用长度验证。在文本编辑器或 IDE 中打开 `app/models/micropost.rb`，写入代码清单 2.10 中的代码。

代码清单 2.10：限制微博的长度最多为 140 个字符

`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  validates :content, length: { maximum: 140 }
end
```

这段代码看起来可能很神秘，我们会在 6.2 节详细介绍验证。如果我们在发布微博的页面输入超过 140 个字符的内容，就能看出这个验证的作用了。如图 2.14 所示，Rails 会渲染错误信息，提示微博的内容太长了。（7.3.3 节会详细介绍错误信息。）

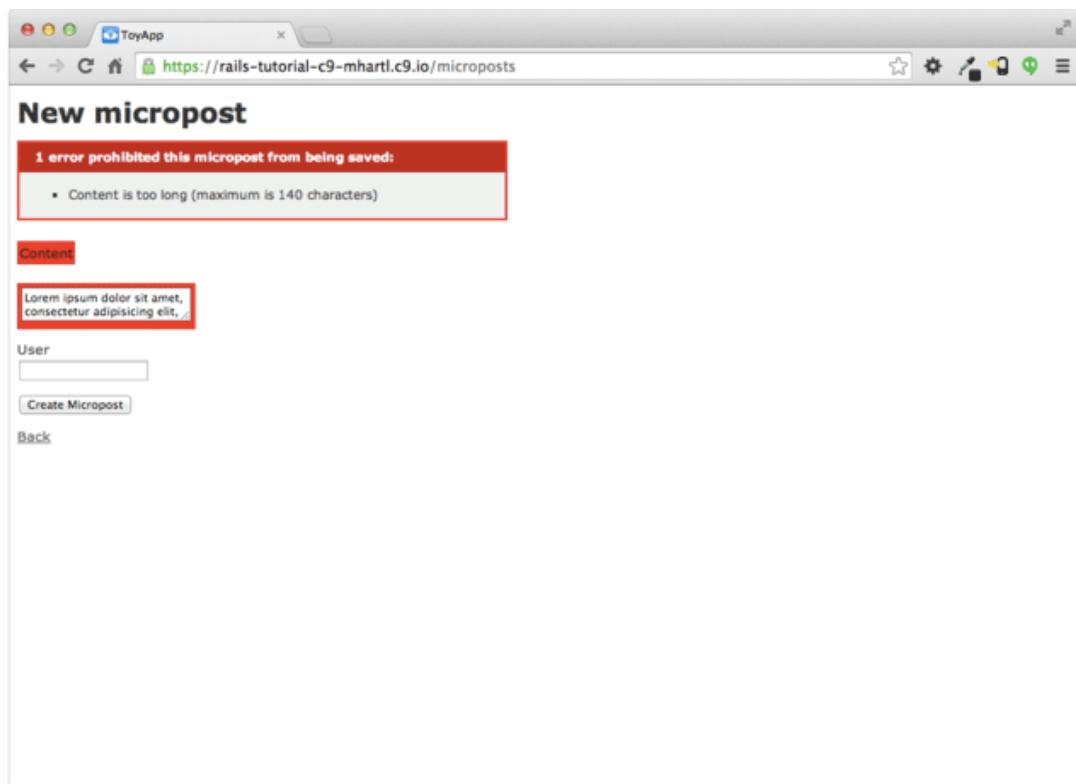


图 2.14：发布微博失败时显示的错误消息

2.3.3. 一个用户拥有多篇微博

Rails 最强大的功能之一是，可以在不同的数据模型之间建立关联（association）。对本例中的用户模型而言，每个用户可以拥有多篇微博。我们可以更新用户模型（参见代码清单 2.11）和微博模型（参见代码清单 2.12）的代码实现这种关联。

代码清单 2.11：一个用户拥有多篇微博

`app/models/user.rb`

```
class User < ActiveRecord::Base
  has_many :microposts
end
```

代码清单 2.12：一篇微博属于一个用户

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :content, length: { maximum: 140 }
end
```

我们可以把这种关联用图 2.15 中的图标表示出来。因为 `microposts` 表中有 `user_id` 这一列，所以 Rails（通过 Active Record）能把微博和各个用户关联起来。

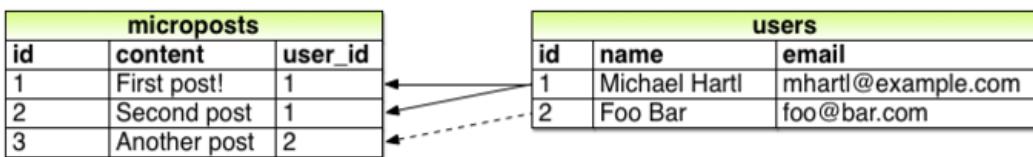


图 2.15：微博和用户之间的关联

在第 11 章和第 12 章，我们会使用微博和用户之间的关联显示一个用户的所有微博，还会生成一个和 Twitter 类似的微博列表。现在，我们可以在控制台（console）中检查用户与微博之间的关联。控制台是和 Rails 应用交互很有用的工具。在命令行中执行 `rails console` 命令，启动控制台。然后输入 `User.first`，从数据库中读取第一个用户，并把得到的数据赋值给 `first_user` 变量：¹⁰

```
$ rails console
>> first_user = User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2014-07-21 02:01:31", updated_at: "2014-07-21 02:01:31">
>> first_user.microposts
=> [#<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2014-07-21 02:37:37", updated_at: "2014-07-21 02:37:37">, #<Micropost id: 2,
content: "Second micropost", user_id: 1, created_at: "2014-07-21 02:38:54",
updated_at: "2014-07-21 02:38:54">]
>> micropost = first_user.microposts.first    # Micropost.first would also work.
=> #<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2014-07-21 02:37:37", updated_at: "2014-07-21 02:37:37">
>> micropost.user
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2014-07-21 02:01:31", updated_at: "2014-07-21 02:01:31">
>> exit
```

我在这段代码的最后一行加上了 `exit`，告诉你如何退出终端。在大多数系统中也可以按 Ctrl-D 键。¹¹ 我们使用 `first_user.microposts` 获取这个用户发布的微博。Active Record 会自动返回 `user_id` 和 `first_user` 的 ID（1）相同的所有微博。在第 11 章和第 12 章中，我们会更深入地学习关联的这种用法。

10. 你的控制台可能会显示类似 2.1.1 :001 > 的提示符，但示例中使用 >> 代替，因为不同的 Ruby 版本显示的提示符不同。

11. 和“Ctrl-C”一样，虽然实际按的是“Ctrl-d”，但习惯写成“Ctrl-D”。

2.3.4. 继承体系

接下来简要介绍 Rails 中控制器和模型的类集成。如果你有面向对象编程（Object-oriented Programming，简称 OOP）的经验，能更好地理解这些内容。如果你未接触过 OOP 的话，可以跳过这一节。一般来说，如果你不熟悉类的概念（[4.4 节](#) 中会介绍），我建议你以后再回过头来读这一节。

我们先介绍模型的继承结构。对比一下[代码清单 2.13](#) 和[代码清单 2.14](#)，可以看出，User 和 Micropost 都（通过 < 符号）继承自 ActiveRecord::Base，这是 Active Record 为模型提供的基类。[图 2.16](#) 列出了这种继承关系。通过继承 ActiveRecord::Base，模型对象才能与数据库通讯，才能把数据库中的列看做 Ruby 中的属性，等等。

代码清单 2.13：User 类中的继承

app/models/user.rb

```
class User < ActiveRecord::Base  
.  
. . .  
end
```

代码清单 2.14：Micropost 类中的继承

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base  
.  
. . .  
end
```

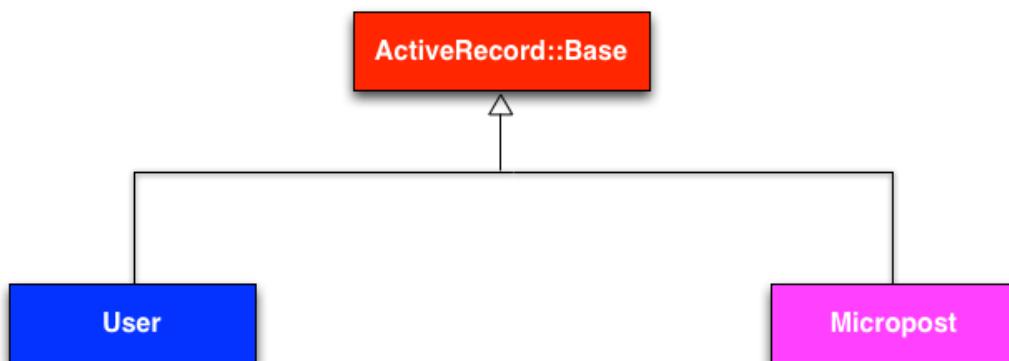


图 2.16：用户模型和微博模型中的继承体系

控制器的继承结构稍微复杂一些。对比[代码清单 2.15](#) 和[代码清单 2.16](#)，可以看出，UsersController 和 Microposts Controller 都继承自 ApplicationController。如[代码清单 2.17](#) 所示，ApplicationController 继承自 ActionController::Base。ActionController::Base 是 Rails 中 Action Pack 库为控制器提供的基类。这些类之间的关系如[图 2.17](#) 所示。

代码清单 2.15：UsersController 类中的继承

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
.
.
.
end
```

代码清单 2.16: MicropostsController 类中的继承

app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController
.
.
.
end
```

代码清单 2.17: ApplicationController 类中的继承

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
.
.
.
end
```

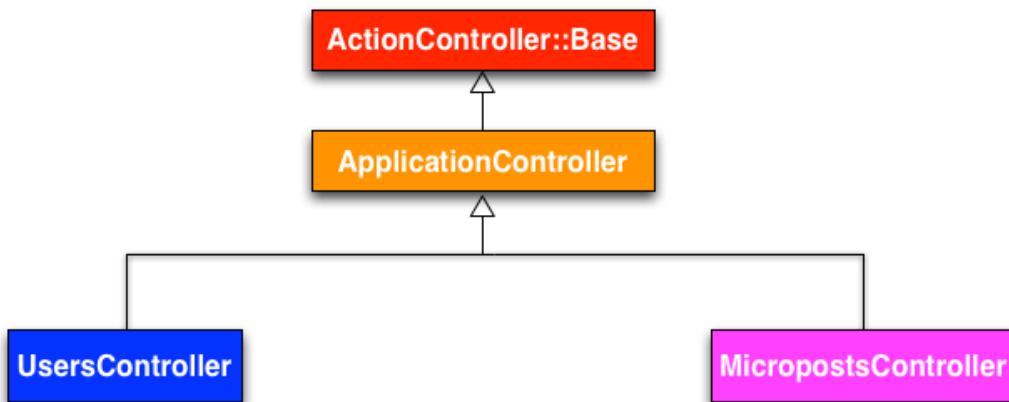


图 2.17: UsersController 和 MicropostsController 中的继承体系

和模型的继承类似，通过继承 `ActionController::Base`，`UsersController` 和 `MicropostsController` 获得了很
多功能。例如，处理模型对象的能力，过滤输入的 HTTP 请求，以及把视图渲染成 HTML。Rails 应用中的所
有控制器都继承 `ApplicationController`，所以其中定义的规则会自动运用于应用中的的每个动作。例如，[8.4](#)
节会介绍如何在 `ApplicationController` 中引入辅助方法，为整个应用的所有控制器都加上登录和退出功能。

2.3.5. 部署这个玩具应用

完成微博资源之后，是时候把代码推送到 Bitbucket 的仓库中了：

```
$ git status
$ git add -A
```

```
$ git commit -m "Finish toy app"  
$ git push
```

通常情况下，你应该经常做一些很小的提交，不过对于本章来说，最后做一次大提交也无妨。

然后，你也可以按照 1.5 节介绍的方法，把这个应用部署到 Heroku：

```
$ git push heroku
```

执行这个命令的前提是，你已经按照 2.1 节中的说明创建了 Heroku 应用。否则，应该先执行 `heroku create`，然后再执行 `git push heroku master`。

然后还要执行下面的命令迁移生产环境的数据库，这样应用才能使用数据库：

```
$ heroku run rake db:migrate
```

这个命令会按照用户和微博的数据模型更新 Heroku 中的数据库。迁移数据库之后，就可以在生产环境中使用这个应用了，如图 2.18 所示，而且这个应用使用 PostgreSQL 数据库。

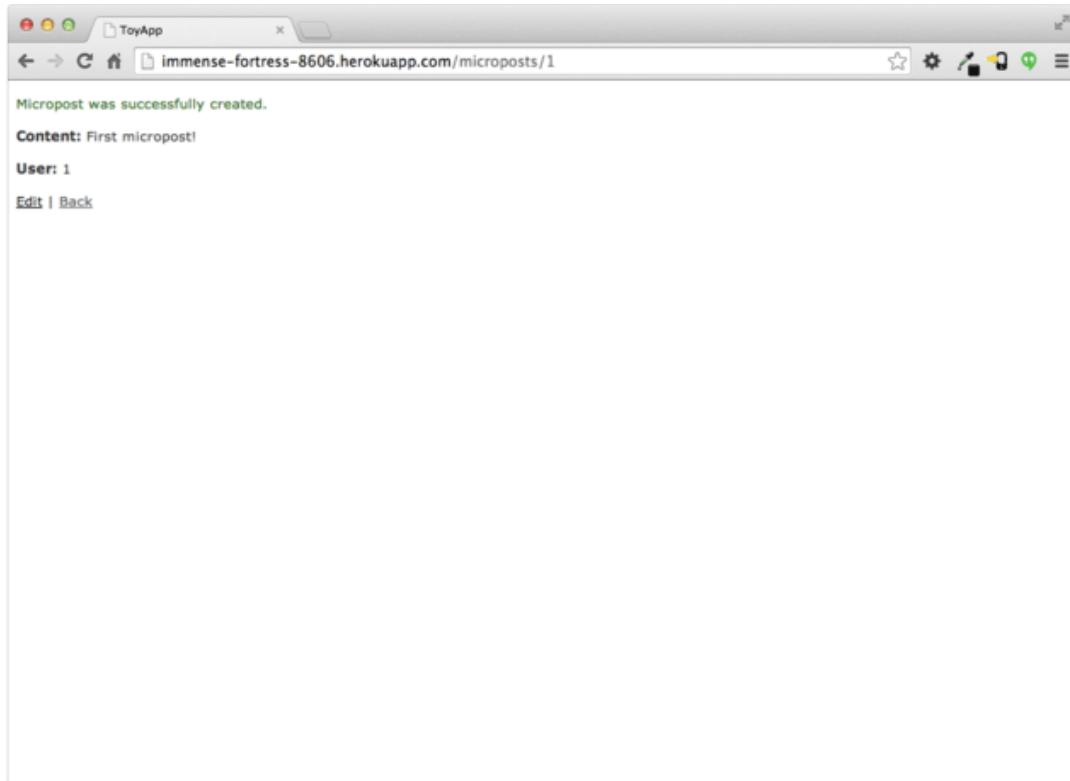


图 2.18：运行在生产环境中的玩具应用

2.4. 小结

至此，对这个 Rails 应用的概览结束了。本章开发的玩具应用有优点也有缺点。

优点

- 概览了 Rails

- 介绍了 MVC
- 第一次体验了 REST 架构
- 开始使用数据模型了
- 在生产环境中运行了一个基于数据库的 Web 应用

缺点

- 没自定义布局和样式
- 没有静态页面（例如“首页”和“关于”）
- 没有用户密码
- 没有用户头像
- 没登录功能
- 不安全
- 没实现用户和微博之间的自动关联
- 没实现“关注”和“被关注”功能
- 没实现微博列表
- 没编写有意义的测试
- 没有真正理解所做的事情

本书后续的内容建立在这些优点之上，而且会改善缺点。

2.4.1. 读完本章学到了什么

- 使用脚手架自动生成模型的代码，然后通过 Web 界面和应用交互；
- 脚手架有利于快速上手，但生成的代码不易理解；
- Rails 使用“模型-视图-控制器”（MVC）模式组织 Web 应用；
- 借由 Rails 我们得知，为了和数据模型交互，REST 架构制定了一套标准的 URL 和控制器动作；
- Rails 支持数据验证，约束数据模型的属性可以使用什么值；
- Rails 内建支持定义数据模型之间关系的功能；
- 可以使用 Rails 控制台在命令行中与 Rails 应用交互。

2.5. 练习

1. 代码清单 2.18 为微博内容添加了一个存在性验证，以此确保微博不能为空。验证这个规则确实能实现如图 2.19 所示的效果。
2. 修改代码清单 2.19，把 `FILL_IN` 改成合适的代码，验证用户模型的 `name` 和 `email` 属性都存在（图 2.20）。

代码清单 2.18：微博内容的存在性验证

`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :content, length: { maximum: 140 },
    presence: true
end
```

代码清单 2.19：在用户模型中加入存在验证

app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts
  validates FILL_IN, presence: true
  validates FILL_IN, presence: true
end
```

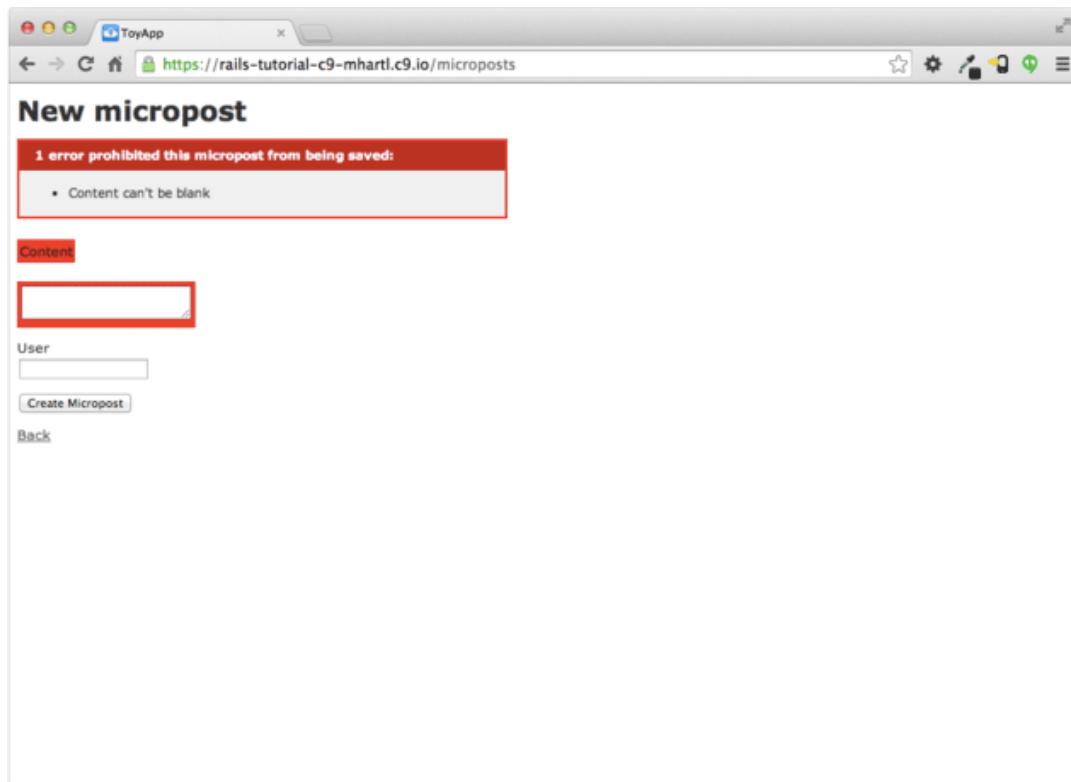


图 2.19：微博模型存在性验证的效果

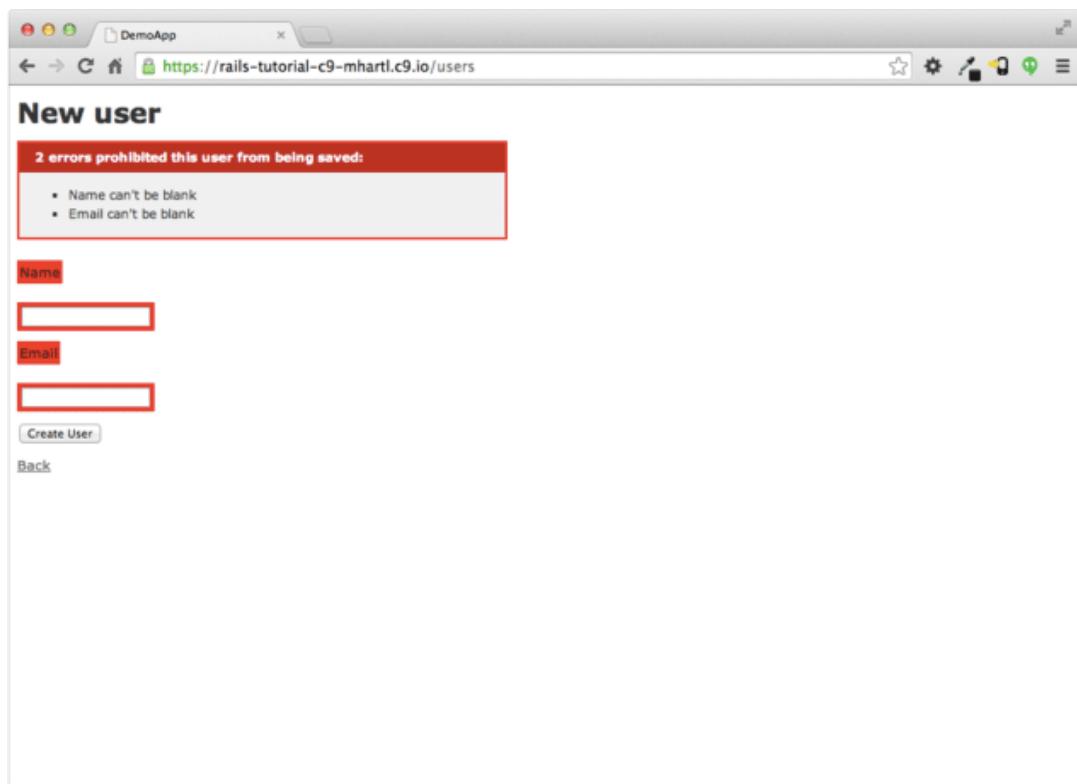


图 2.20：用户模型中存在性验证的效果

第 3 章 基本静态的页面

从本章开始，我们要开发一个专业级演示应用，本书后续内容会一直开发这个应用。最终完成的应用包含用户、微博功能，以及完整的登录和用户身份认证系统，不过我们先从一个看似功能有限的话题出发——创建静态页面。这看似简单的一件事却是一个很好的锻炼，极具意义，对这个初建的应用而言也是个很好的开端。

虽然 Rails 被设计出来是为了开发基于数据库的动态网站，不过它也能胜任使用纯 HTML 创建的静态页面。其实，使用 Rails 创建静态页面有一个好处：添加少量动态内容十分容易。这一章就教你怎么做。在这个过程中，我们会一窥自动化测试（automated testing）的面目，自动化测试可以让我们相信自己编写的代码是正确的。而且，编写一个好的测试组件还可以让我们信心十足地重构代码，修改实现过程但不影响功能。

3.1. 创建演示应用

和第 2 章一样，我们要先创建一个新 Rails 项目，名为 `sample_app`，如[代码清单 3.1](#) 所示：¹

代码清单 3.1：创建一个新应用

```
$ cd ~/workspace  
$ rails _4.2.0.beta4_ new sample_app  
$ cd sample_app/
```

（和[2.1 节](#)一样，如果使用云端 IDE，可以在同一个工作空间中创建这个应用，没必要再新建一个工作空间。）

类似[2.1 节](#)，接下来我们要用文本编辑器打开并编辑 `Gemfile`，写入应用所需的 `gem`。[代码清单 3.2](#) 与[代码清单 1.5](#) 和[代码清单 2.1](#) 一样，不过 `test` 组中的 `gem` 有所不同，稍后会做进一步设置（[3.7 节](#)）。注意，如果你现在你想安装这个应用使用的所有 `gem`，要写入[代码清单 11.66](#) 中的内容。

代码清单 3.2：演示应用的 `Gemfile`

```
source 'https://rubygems.org'  
  
gem 'rails',                  '4.2.0.beta4'  
gem 'sass-rails',              '5.0.0.beta1'  
gem 'uglifier',                '2.5.3'  
gem 'coffee-rails',            '4.0.1'  
gem 'jquery-rails',            '4.0.0.beta2'  
gem 'turbolinks',              '2.3.0'  
gem 'jbuilder',                '2.2.3'
```

1. 如果使用云端 IDE，可以使用“Goto Anything”命令，输入部分文件名就能方便地在文件系统中找到所需的文件。现在三个应用都放在同一个工作空间中，只输入文件名效果可能不很理想。例如，如果查找名为“`Gemfile`”的文件，会出现六个结果，因为每个应用中都有能匹配查找条件的两个文件：`Gemfile` 和 `Gemfile.lock`。因此，你可以把前两个应用删除，方法是：进入 `workspace` 文件夹，执行 `rm -rf hello_app/ toy_app/` 命令（参见[表 1.1](#)）。只要你之前把这两个应用推送到 Bitbucket 中，以后再恢复都很容易。

```

gem 'rails-html-sanitizer', '1.0.1'
gem 'sdoc',                 '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',      '1.3.9'
  gem 'byebug',       '3.4.0'
  gem 'web-console',   '2.0.0.beta3'
  gem 'spring',        '1.1.3'
end

group :test do
  gem 'minitest-reporters', '1.0.5'
  gem 'mini_backtrace',     '0.1.3'
  gem 'guard-minitest',    '2.3.1'
end

group :production do
  gem 'pg',            '0.17.1'
  gem 'rails_12factor', '0.0.2'
end

```

和前两章一样，我们要执行 `bundle install` 命令安装并导入 `Gemfile` 中指定的 gem，而且指定 `--without production` 选项，² 不安装生产环境使用的 gem：

```
$ bundle install --without production
```

运行这个命令后不会在开发环境中安装 PostgreSQL 所需的 `pg` gem，在生产环境和测试环境中我们使用 SQLite。Heroku 极力不建议在开发环境和生产环境中使用不同的数据库，但是对这个演示应用来说这两种数据库没什么差别，而且在本地安装配置 SQLite 比 PostgreSQL 容易得多。³ 如果你之前安装了某个 gem（例如 Rails 本身）的其他版本，和 `Gemfile` 中指定的版本号不同，最好再执行 `bundle update` 命令，更新 gem，确保安装的版本和指定的一致：

```
$ bundle update
```

最后，我们还要初始化 Git 仓库：

```

$ git init
$ git add -A
$ git commit -m "Initialize repository"

```

和第一个应用一样，我建议你更新一下 `README` 文件（在应用的根目录中），更好的描述这个应用。我们先把这个文件的格式从 RDoc 改为 Markdown：

```
$ git mv README.rdoc README.md
```

然后写入[代码清单 3.3](#) 中的内容。

2. 注意，这个参数会被 Bundler 记住，下次只需运行 `bundle install` 即可。
 3. 现在时机还不成熟，但我建议你某一天一定要学会如何在开发环境中安装配置 PostgreSQL。届时，可以在谷歌中搜索“install configure postgresql <your system>”，以及“rails postgresql setup”。在云端 IDE 中，“<your system>”是 Ubuntu。

代码清单 3.3：修改演示应用的 README 文件

```
# Ruby on Rails Tutorial: sample application

This is the sample application for the
[*Ruby on Rails Tutorial:
Learn Web Development with Rails*](http://www.railstutorial.org/)
by [Michael Hartl](http://www.michaelhartl.com/).
```

最后，提交这次改动：

```
$ git commit -am "Improve the README"
```

你可能还记得，在 1.4.4 节，我们使用 `git commit -a -m "Message"` 命令，指定了“全部变化”的旗标 `-a` 和提交信息旗标 `-m`。如上面这个命令所示，我们可以把两个旗标合在一起，变成 `git commit -am "Message"`。

既然本书后续内容会一直使用这个演示应用，那么最好在 Bitbucket 中新建一个仓库，把这个应用推送上去：

```
$ git remote add origin git@bitbucket.org:<username>/sample_app.git
$ git push -u origin --all # 首次推送这个应用
```

为了避免以后遇到焦头烂额的问题，在这个早期阶段也可以把应用部署到 Heroku 中。参照第 1 章和第 2 章，我建议使用代码清单 1.8 和代码清单 1.9 中的代码，创建一个显示“hello, world!”的首页。然后提交改动，再推送到 Heroku 中：

```
$ git commit -am "Add hello"
$ heroku create
$ git push heroku master
```

（和 1.5 节一样，你可能会看到一些警告消息，现在暂且不管，7.5 节会解决。）除了 Heroku 为应用分配的地址之外，看到的页面应该和图 1.18 一样。

在阅读本书的过程中，我建议你定期推送和部署，这样不仅能在远程仓库中备份，而且还能尽早发现在生产环境中可能出现的问题。如果遇到和 Heroku 有关的问题，可以查看生产环境中的日志，试着找出问题所在：

```
$ heroku logs
```

注意，如果你决定把真实的应用放到 Heroku 中，一定要按照 7.5 节介绍的方法配置 Unicorn。

3.2. 静态页面

前一节的准备工作做好之后，我们可以开始开发这个演示应用了。本节，我们要向开发动态页面迈出第一步：创建一些 Rails 动作和视图，但只包含静态 HTML。Rails 动作放在控制器中（MVC 中的 C，参见 1.3.3 节），其中的动作是为了实现相关的功能。第 2 章已经简要介绍了控制器，全面熟悉 REST 架构之后（从第 6 章开始），你会更深入地理解控制器。回想一下 1.3 节介绍的 Rails 项目文件夹结构（图 1.4），会对我们有所帮助。这一节主要在 `app/controllers` 和 `app/views` 两个文件夹中工作。

在 1.4.4 节我们说过，使用 Git 时最好在单独的主题分支中完成工作。如果你使用 Git 做版本控制，现在应该执行下述命令，切换到一个主题分支中，然后再创建静态页面：

```
$ git checkout master
$ git checkout -b static-pages
```

(第一个命令的作用是确保我们现在处于主分支中，这样才能基于 `master` 分支创建 `static-pages` 分支。如果你当前就在主分支中，可以不执行这个命令。)

3.2.1. 生成静态页面

下面我们要使用第 2 章用来生成脚手架的 `generate` 命令生成一个控制器，既然这个控制器用来处理静态页面，那就把它命名为 `StaticPages` 吧。可以看出，控制器的名字使用[驼峰式命名法](#)。我们计划创建“首页”，“帮助”页面和“关于”页面，对应的动作名分别为 `home`、`help` 和 `about`。`generate` 命令可以接收一个可选的参数列表，指定要创建的动作。我们要在命令行中指定“首页”和“帮助”页面的动作，但故意不指定“关于”页面的动作，在 3.3 节再介绍怎么添加这个动作。生成静态页面控制器的命令如[代码清单 3.4](#) 所示。

代码清单 3.4：生成静态页面控制器

```
$ rails generate controller StaticPages home help
  create  app/controllers/static_pages_controller.rb
  route   get 'static_pages/help'
  route   get 'static_pages/home'
invoke  erb
  create  app/views/static_pages
  create  app/views/static_pages/home.html.erb
  create  app/views/static_pages/help.html.erb
invoke  test_unit
  create  test/controllers/static_pages_controller_test.rb
invoke  helper
  create  app/helpers/static_pages_helper.rb
invoke  test_unit
  create  test/helpers/static_pages_helper_test.rb
invoke  assets
invoke  coffee
  create  app/assets/javascripts/static_pages.js.coffee
invoke  scss
  create  app/assets/stylesheets/static_pages.css.scss
```

顺便说一下，`rails generate` 可以简写成 `rails g`。除此之外，Rails 还提供了几个命令的简写形式，参见[表 3.1](#)。为了表述明确，本书会一直使用命令的完整形式，但在实际使用中，大多数 Rails 开发者或多或少都会使用简写形式。

表 3.1：Rails 中一些命令的简写形式

完整形式	简写形式
<code>\$ rails server</code>	<code>\$ rails s</code>
<code>\$ rails console</code>	<code>\$ rails c</code>
<code>\$ rails generate</code>	<code>\$ rails g</code>
<code>\$ bundle install</code>	<code>\$ bundle</code>
<code>\$ rake test</code>	<code>\$ rake</code>

在继续之前，如果你使用 Git，最好把静态页面控制器对应的文件推送到远程仓库：

```
$ git status  
$ git add -A  
$ git commit -m "Add a Static Pages controller"  
$ git push -u origin static-pages
```

最后一个命令的意思是，把 `static-pages` 主题分支推送到 Bitbucket。以后再推送时，可以省略后面的参数，简写成：

```
$ git push
```

在现实的开发过程中，我一般都会先提交再推送，但是为了行文简洁，从这往后我们会省略提交这一步。

注意，在[代码清单 3.4](#) 中，我们传入的控制器名使用驼峰式，创建的控制器文件名使用[蛇底式](#)。所以，传入“`StaticPages`”得到的文件是 `static_pages_controller.rb`。这只是一种约定。其实在命令行中也可以使用蛇底式：

```
$ rails generate controller static_pages ...
```

这个命令也会生成名为 `static_pages_controller.rb` 的控制器文件。因为 Ruby 的类名使用驼峰式（[4.4 节](#)），所以提到控制器时我会使用驼峰式，不过这是我的个人选择。（因为 Ruby 文件名一般使用蛇底式，所以 Rails 生成器使用 `underscore` 方法把驼峰式转换成蛇底式。）

顺便说一下，如果在生成代码时出现了错误，知道如何撤销操作就很有用了。[旁注 3.1](#) 中介绍了一些如何在 Rails 中撤销操作的方法。

旁注 3.1：撤销操作

即使再小心，在开发 Rails 应用的过程中也可能会犯错。幸好 Rails 提供了一些工具能够帮助我们还原操作。

举例来说，一个常见的原因是，更改控制器的名字，这时你得删除生成的文件。生成控制器时，除了控制器文件本身之外，Rails 还会生成很多其他文件（参见[代码清单 3.4](#)）。撤销生成的文件不仅仅要删除控制器文件，还要删除一些辅助的文件。（在[2.2 节](#)和[2.3 节](#)我们看到，`rails generate` 命令还会自动修改 `routes.rb` 文件，因此我们也想自动撤销这些修改。）在 Rails 中，我们可以使用 `rails destroy` 命令完成撤销操作。一般来说，下面这两个命令是相互抵消的：

```
$ rails generate controller StaticPages home help  
$ rails destroy controller StaticPages home help
```

[第 6 章](#)会使用下面的命令生成模型：

```
$ rails generate model User name:string email:string
```

这个操作可以使用下面的命令撤销：

```
$ rails destroy model User
```

（在这个例子中，我们可以省略命令行中其余的参数。读到[第 6 章](#)时，看看你能否发现为什么可以这么做。）

对模型来说，还涉及到撤销迁移。[第 2 章](#)已经简要介绍了迁移，[第 6 章](#)开始会深入介绍。迁移通过下面的命令改变数据库的状态：

```
$ bundle exec rake db:migrate
```

我们可以使用下面的命令撤销前一个迁移操作：

```
$ bundle exec rake db:rollback
```

如果要回到最开始的状态，可以使用：

```
$ bundle exec rake db:migrate VERSION=0
```

你可能猜到了，把数字 0 换成其他的数字就会回到相应的版本，这些版本数字是按照迁移执行的顺序排列的。

知道这些技术，我们就可以得心应对开发过程中遇到的各种问题了。

代码清单 3.4 中生成静态页面控制器的命令会自动修改路由文件（`config/routes.rb`）。我们在 [1.3.4 节](#) 已经简略介绍过路由文件，它的作用是实现 URL 和网页之间的对应关系（[图 2.11](#)）。路由文件在 `config` 文件夹中。Rails 在这个文件夹中存放应用的配置文件（[图 3.1](#)）。

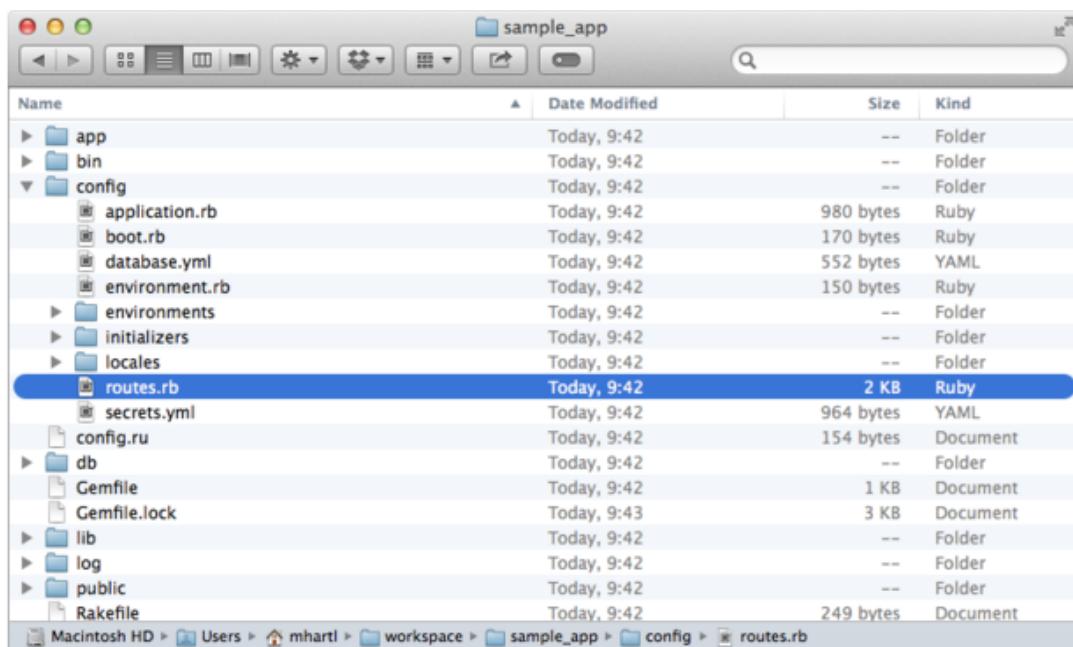


图 3.1：演示应用 config 文件夹中的内容

因为生成控制器时我们指定了 `home` 和 `help` 动作，所以在路由文件中已经添加了相应的规则，如[代码清单 3.5](#) 所示。

代码清单 3.5： 静态页面控制器中 `home` 和 `help` 动作的路由
`config/routes.rb`

```
Rails.application.routes.draw do
  get 'static_pages/home'
  get 'static_pages/help'
  .
```

```
end
```

如下的规则

```
get 'static_pages/home'
```

把发给 /static_pages/home 的请求映射到静态页面控制器中的 home 动作上。另外，`get` 表明这个路由响应 GET 请求。GET 是 HTTP（超文本传输协议，Hypertext Transfer Protocol）支持的基本请求方法之一（[旁注 3.2](#)）。在这个例子中，当我们在静态页面控制器中生成 home 动作时，就自动在 /static_pages/home 地址上获得了一个页面。若想查看这个页面，按照 [1.3.2 节](#) 中的方法，启动 Rails 开发服务器：

```
$ rails server -b $IP -p $PORT      # 如果在自己的电脑中，只需执行 `rails server`
```

然后访问 `/static_pages/home`，如图 3.2 所示。

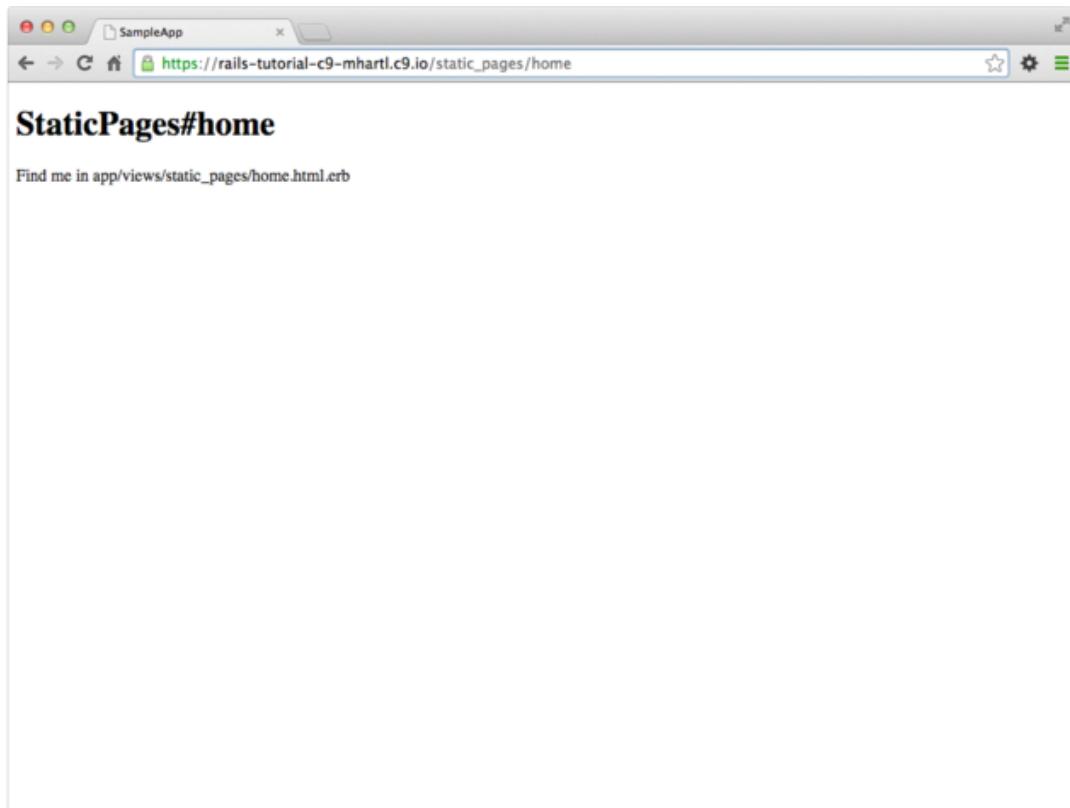


图 3.2：简陋的首页 (`/static_pages/home`)

旁注3.2：GET 等

超文本传输协议（HTTP）定义了几个基本操作，`GET`、`POST`、`PATCH` 和 `DELETE`。这四个动词表示客户端电脑（通常安装了一种浏览器，例如 Chrome、Firefox 或 Safari）和服务器（通常会运行一个 Web 服务器，例如 Apache 或 Nginx）之间的操作。（有一点很重要，你要知道，在本地电脑中开发 Rails 应用时，客户端和服务器在同一台物理设备中，但是二者是不同的概念。）受 REST 架构影响的 Web 框架

(包括 Rails) 都很重视对 HTTP 动词的实现，我们在[第 2 章](#)已经简要介绍了 REST，从[第 7 章](#)开始会做更详细的介绍。

`GET` 是最常用的 HTTP 操作，用来读取网络中的数据。它的意思是“读取一个网页”，当你访问 <http://www.google.com> 或 <http://www.wikipedia.org> 时，浏览器发送的就是 `GET` 请求。`POST` 是第二种最常用的操作，当你提交表单时浏览器发送的就是 `POST` 请求。在 Rails 应用中，`POST` 请求一般用来创建某个东西（不过 HTTP 也允许 `POST` 执行更新操作）。例如，提交注册表单时发送的 `POST` 请求会在网站中创建一个新用户。另外两个动词，`PATCH` 和 `DELETE`，分别用来更新和销毁服务器上的某个东西。这两个操作没 `GET` 和 `POST` 那么常用，因为浏览器没有内建对这两种请求的支持，不过有些 Web 框架（包括 Rails）通过一些聪明的处理方式，让它看起来就像是浏览器发出的一样。所以，这四种请求类型 Rails 都支持。

要想弄明白这个页面是怎么来的，我们先在文本编辑器中看一下静态页面控制器文件。你应该会看到类似[代码清单 3.6](#) 所示的内容。你可能注意到了，不像[第 2 章](#)中的用户和微博控制器，静态页面控制器没使用标准的 REST 动作。这对静态页面来说是很常见的，毕竟 REST 架构不能解决所有问题。

代码清单 3.6：代码清单 3.4 生成的静态页面控制器

`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

从上面代码中的 `class` 可以看出，`static_pages_controller.rb` 文件中定义了一个类，名为 `StaticPagesController`。类是一种组织函数（也叫方法）的有效方式，例如 `home` 和 `help` 动作就是方法，使用 `def` 关键字定义。[2.3.4 节](#)说过，尖括号 `<` 表示 `StaticPagesController` 继承自 `ApplicationController` 类，这就意味着我们定义的页面拥有了 Rails 提供的大量功能。（我们会在[4.4 节](#)更详细的介绍类和继承。）

在本例中，静态页面控制器中的两个方法默认都是空的：

```
def home
end

def help
end
```

如果是普通的 Ruby 代码，这两个方法什么也做不了。不过在 Rails 中就不一样了，`StaticPagesController` 是一个 Ruby 类，但是因为它继承自 `ApplicationController`，其中的方法对 Rails 来说就有了特殊意义：访问 `/static_pages/home` 时，Rails 会在静态页面控制器中寻找 `home` 动作，然后执行该动作，再渲染相应的视图（MVC 中的 V，参见[1.3.3 节](#)）。在本例中，`home` 动作是空的，所以访问 `/static_pages/home` 后只会渲染视图。那么，视图是什么样子，怎么才能找到它呢？

如果你再看一下[代码清单 3.4](#) 的输出，或许能猜到动作和视图之间的对应关系：`home` 动作对应的视图是 `home.html.erb`。[3.4 节](#)会告诉你 `.erb` 是什么意思。看到 `.html` 你或许就不奇怪了，这个文件基本上就是 HTML，如[代码清单 3.7](#) 所示。

代码清单 3.7：为“首页”生成的视图

app/views/static_pages/home.html.erb

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

`help` 动作的视图类似，如[代码清单 3.8](#) 所示。

代码清单 3.8：为“帮助”页面生成的视图

app/views/static_pages/help.html.erb

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

这两个视图都只是占位用的，它们的内容中都有一个一级标题（`h1` 标签）和一个显示视图文件完整路径的段落（`p` 标签）。

3.2.2. 修改静态页面中的内容

我们会在[3.4 节](#)添加一些简单的动态内容。现在，这些静态内容的存在是为了强调一件很重要的事：Rails 的视图可以只包含静态的 HTML。所以我们甚至无需了解 Rails 就可以修改“首页”和“帮助”页面的内容，如[代码清单 3.9](#) 和[3.10](#) 所示。

代码清单 3.9：修改“首页”的 HTML

app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

代码清单 3.10：修改“帮助”页面的 HTML

app/views/static_pages/help.html.erb

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.
  To get help on this sample app, see the
  <a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

修改之后，这两个页面显示的内容如[图 3.3](#) 和[3.4](#) 所示。

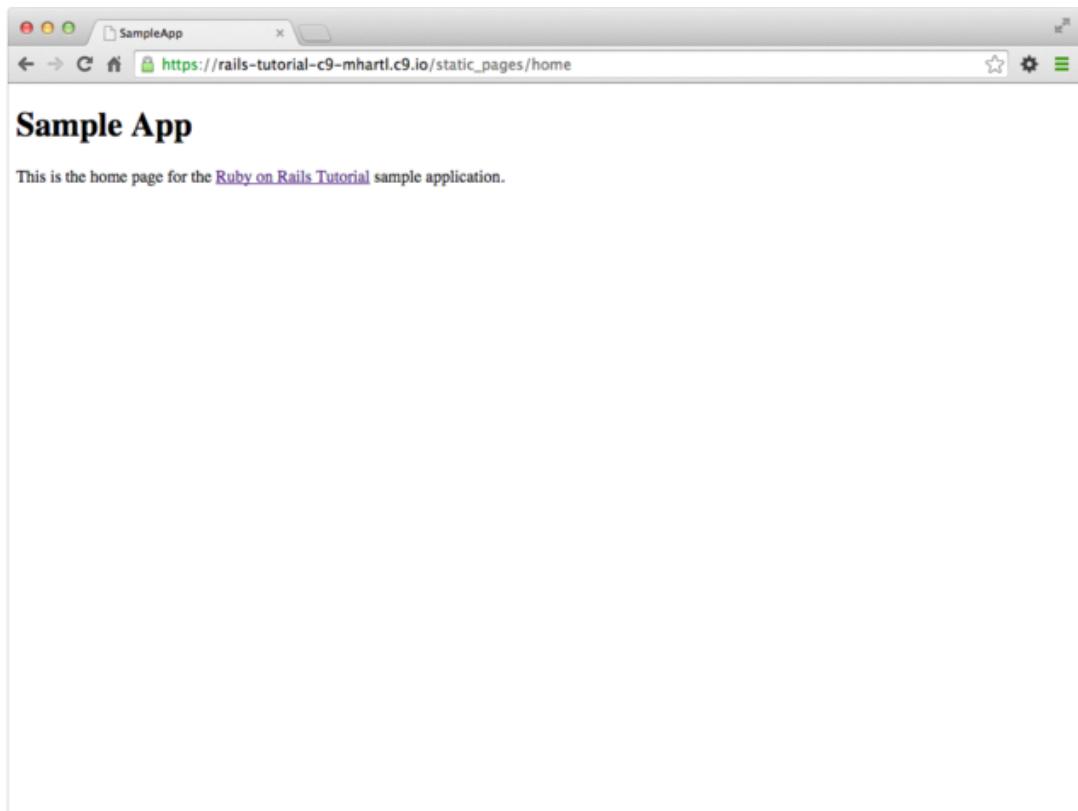


图 3.3：修改后的“首页”

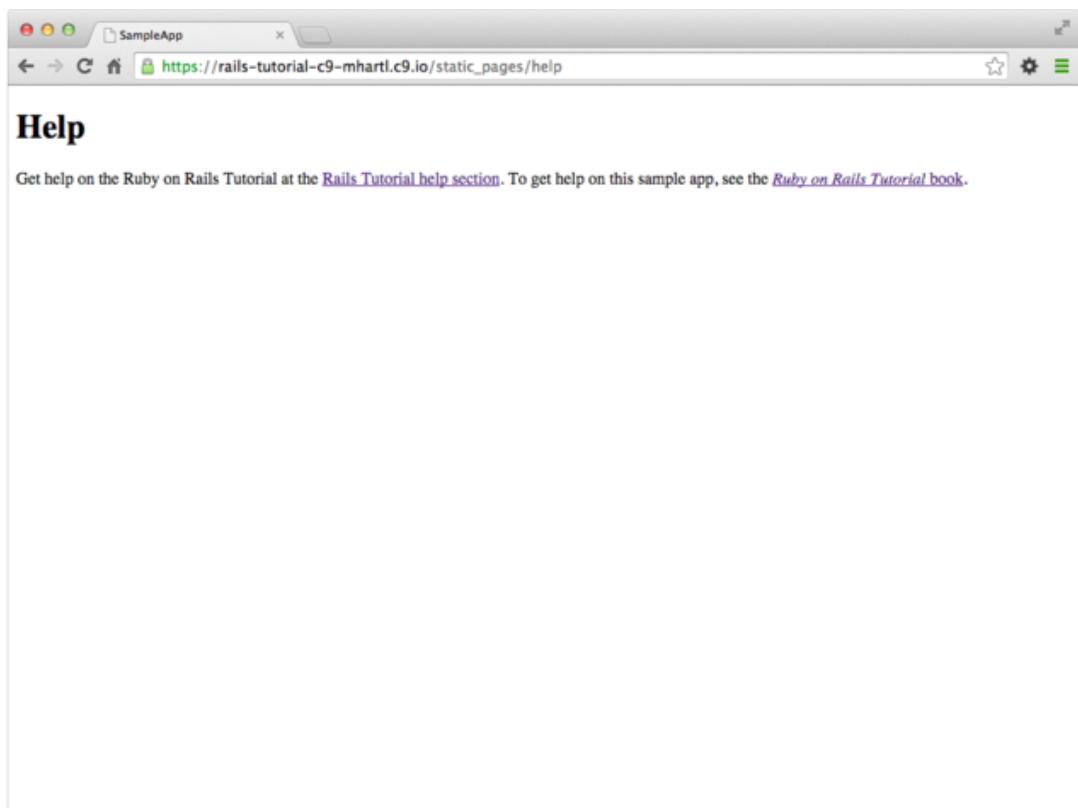


图 3.4：修改后的“帮助”页面

3.3. 开始测试

我们创建并修改了“首页”和“帮助”页面的内容，下面要添加“关于”页面。做这样的改动时，最好编写自动化测试确认实现的方法正确。对本书开发的应用来说，我们编写的测试组件有两个作用：其一，是一种安全防护措施；其二，作为源码的文档。虽然要编写额外的代码，但是如果方法得当，测试能协助我们快速开发，因为有了测试查找问题所用的时间会变少。不过，我们要善于编写测试才行，所以要尽早开始练习。

几乎每个 Rails 开发者都认同测试是好习惯，但具体的作法多种多样。最近有一场针对“测试驱动开发”（Test-Driven Development，简称 TDD）的辩论⁴，十分热闹。TDD 是一种测试技术，程序员要先编写失败的测试，然后再编写应用的代码，让测试通过。本书采用一种轻量级，符合直觉的测试方案，只在适当的时候才使用 TDD，而不严格遵守 TDD 理念（[旁注 3.3](#)）。

旁注 3.3：什么时候测试

判断何时以及如何测试之前，最好弄明白为什么要测试。在我看来，编写自动化测试主要有三个好处：

1. 测试能避免“回归”（regression），即由于某些原因之前能用的功能不能用了；
2. 有测试，重构（改变实现方式，但功能不变）时更有自信；
3. 测试是应用代码的客户，因此可以协助我们设计，以及决定如何与系统的其他组件交互。

以上三个好处都不要求先编写测试，但在很多情况下，TDD 仍有它的价值。何时以及如何测试，部分取决于你编写测试的熟练程度。很多开发者发现，熟练之后，他们更倾向于先编写测试。除此之外，还取决于测试较之应用代码有多难，你对想实现的功能有多深的认识，以及未来在什么情况下这个功能会遭到破坏。

现在，最好有一些指导方针，告诉我们什么时候应该先写测试（以及什么时候完全不用测试）。根据我自己的经验，给出一些建议：

- 和应用代码相比，如果测试代码特别简短，倾向于先编写测试；
- 如果对想实现的功能不是特别清楚，倾向于先编写应用代码，然后再编写测试，改进实现的方式；
- 安全是头等大事，保险起见，要为安全相关的功能先编写测试；
- 只要发现一个问题，就编写一个测试重现这种问题，以避免回归，然后再编写应用代码修正问题；
- 尽量不为以后可能修改的代码（例如 HTML 结构的细节）编写测试；
- 重构之前要编写测试，集中测试容易出错的代码。

在实际的开发中，根据上述方针，我们一般先编写控制器和模型测试，然后再编写集成测试（测试模型、视图和控制器结合在一起时的表现）。如果应用代码很容易出错，或者经常会变动（视图就是这样），我们就完全不测试。

4. 详情参见 Rails 创始人 David Heinemeier Hansson 写的一篇文章：[TDD is dead. Long live testing.](#)

我们主要编写的测试类型是控制器测试（本节开始编写），模型测试（第 6 章开始编写）和集成测试（第 7 章开始编写）。集成测试的作用特别大，它能模拟用户在浏览器中和应用交互的过程，最终会成为我们的主要关注对象，不过控制器测试更容易上手。

3.3.1. 第一个测试

现在我们要在这个应用中添加一个“关于”页面。我们会看到，这个测试很简短，所以按照[旁注 3.3](#)中的指导方针，我们要先编写测试。然后使用失败的测试驱动我们编写应用代码。

着手测试是件具有挑战的事情，要求对 Rails 和 Ruby 都有深入的了解。这么早就编写测试可能有点儿吓人。不过，Rails 已经为我们解决了最难的部分，因为执行 `rails generate controller` 命令时（[代码清单 3.4](#)）自动生成了一个测试文件，我们可以从这个文件入手：

```
$ ls test/controllers/  
static_pages_controller_test.rb
```

我们看一下这个文件的内容，如[代码清单 3.11](#) 所示。

代码清单 3.11：为静态页面控制器生成的测试 GREEN

`test/controllers/static_pages_controller_test.rb`

```
require 'test_helper'  
  
class StaticPagesControllerTest < ActionController::TestCase  
  
  test "should get home" do  
    get :home  
    assert_response :success  
  end  
  
  test "should get help" do  
    get :help  
    assert_response :success  
  end  
end
```

现在无需理解详细的句法，不过可以看出，其中有两个测试，对应我们在命令行中传入的两个动作（[代码清单 3.4](#)）。在每个测试中，先访问动作，然后确认（通过“断言”）得到正确的响应。其中，`get` 表示测试期望这两个页面是普通的网页，可以通过 GET 请求访问（[旁注 3.2](#)）；`:success` 响应是对 HTTP 响应码的抽象表示（在这里表示 200 OK）。也就是说，下面这个测试

```
test "should get home" do  
  get :home  
  assert_response :success  
end
```

它的意思是：我们要测试首页，那么就向 `home` 动作发起一个 GET 请求，确认得到的是表示成功的响应码。

下面我们要运行测试组件，确认测试现在可以通过。方法是，按照下面的方式运行 `rake` 任务（[旁注 2.1](#)）：⁵

5. [2.2 节](#)说过，在有些系统中不需要使用 `bundle exec`，其中就包括云端 IDE。不过为了表述完整，我加上了。一般情况下我都会省略 `bundle exec`，如果遇到错误，再加上试试。

代码清单 3.12: GREEN

```
$ bundle exec rake test  
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

按照需求，一开始测试组件可以通过（GREEN）。（如果没按照 3.7.1 节的说明添加 MiniTest 报告程序，不会看到绿色。）顺便说一下，测试要花点时间启动，因为（1）要启动 Spring 服务器预加载部分 Rails 环境，不过这一步只在首次启动时执行；（2）启动 Ruby 也要花点儿时间。（第二点可以使用 3.7.3 节推荐的 Guard 改善。）

3.3.2. 遇红

我们在旁注 3.3 中说过，TDD 流程是，先编写一个失败测试，然后编写应用代码让测试通过，最后再按需重构代码。因为很多测试工具都使用红色表示失败的测试，使用绿色表示通过的测试，所以这个流程有时也叫“遇红-变绿-重构”循环。这一节我们先完成这个循环的第一步，编写一个失败测试，“遇红”。然后在 3.3.3 节变绿，3.4.3 节重构。⁶

首先，我们要为“关于”页面编写一个失败测试。参照代码清单 3.11，你或许能猜到应该怎么写，如代码清单 3.13 所示。

代码清单 3.13: “关于”页面的测试 RED

`test/controllers/static_pages_controller_test.rb`

```
require 'test_helper'  
  
class StaticPagesControllerTest < ActionController::TestCase  
  
  test "should get home" do  
    get :home  
    assert_response :success  
  end  
  
  test "should get help" do  
    get :help  
    assert_response :success  
  end  
  
  test "should get about" do  
    get :about  
    assert_response :success  
  end  
end
```

如高亮显示的那几行所示，为“关于”页面编写的测试与首页和“帮助”页面的测试一样，只不过把“home”或“help”换成了“about”。

这个测试现在失败：

6. 默认情况下，执行 `rake test` 任务后，如果测试失败会显示红色，但测试通过不会显示绿色。若想显示绿色，参照 3.7.1 节的说明。

代码清单 3.14: RED

```
$ bundle exec rake test  
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
```

3.3.3. 变绿

现在有了一个失败测试（RED），我们要在这个失败测试的错误消息指示下，让测试通过（GREEN），也就是要实现一个可以访问的“关于”页面。

我们先看一下这个失败测试给出的错误消息：⁷

代码清单 3.15: RED

```
$ bundle exec rake test  
ActionController::UrlGenerationError:  
No route matches {:action=>"about", :controller=>"static_pages"}
```

这个错误消息说，没有找到需要的动作和控制器组合，其实就是提示我们要在路由文件中添加一个规则。参照[代码清单 3.5](#)，我们可以编写如[代码清单 3.16](#) 所示的路由。

代码清单 3.16: 添加 about 路由 RED

config/routes.rb

```
Rails.application.routes.draw do  
  get 'static_pages/home'  
  get 'static_pages/help'  
  get 'static_pages/about'  
  .  
  .  
  .  
end
```

这段代码中高亮显示的那行告诉 Rails，把发给 /static_pages/about 页面的 GET 请求交给静态页面控制器中的 `about` 动作处理。

然后再运行测试组件，仍然无法通过，不过错误消息变了：

代码清单 3.17: RED

```
$ bundle exec rake test  
AbstractController::ActionNotFound:  
The action 'about' could not be found for StaticPagesController
```

这个错误消息的意思是，静态页面控制器中缺少 `about` 动作。我们可以参照[代码清单 3.6](#) 编写这个动作，如[代码清单 3.18](#) 所示。

代码清单 3.18: 在静态页面控制器中添加 about 动作 RED

app/controllers/static_pages_controller.rb

7. 在有些系统中，可能要拖动滚动条跳过源码的错误路径——“调用跟踪”。[3.7.2 节](#)介绍了如何过滤调用跟踪，避免显示不需要的内容。

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

现在测试依旧失败，不过测试消息又变了：

```
$ bundle exec rake test
ActionView::MissingTemplate: Missing template static_pages/about
```

这表示没有模板。在 Rails 中，模板就是视图。[3.2.1 节](#)说过，`home` 动作对应的视图是 `home.html.erb`，保存在 `app/views/static_pages` 文件夹中。所以，我们要在这个文件夹中新建一个文件，并且要命名为 `about.html.erb`。

在不同的系统中新建文件有不同的方法，不过大多数情况下都可以在想要新建文件的文件夹中点击鼠标右键，然后在弹出的菜单中选择“新建文件”。或者，可以使用文本编辑器的“文件”菜单，新建文件后再选择保存的位置。除此之外，还可以使用我最喜欢的 [Unix touch 命令](#)，用法如下：

```
$ touch app/views/static_pages/about.html.erb
```

`touch` 的主要作用是更新文件或文件夹的修改时间戳，别无其他效果，但有个副作用，如果文件不存在，就会新建一个。（如果使用云端 IDE，或许要刷新文件树，参见 [1.3.1 节](#)。）

在正确的文件夹中创建 `about.html.erb` 文件之后，要在其中写入[代码清单 3.19](#) 中的内容。

代码清单 3.19：“关于”页面的内容 GREEN

`app/views/static_pages/about.html.erb`

```
<h1>About</h1>
<p>
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a> is a
  <a href="http://www.railstutorial.org/book">book</a> and
  <a href="http://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample application for the tutorial.
</p>
```

现在，运行 `rake test`，会看到测试通过了：

代码清单 3.20：GREEN

```
$ bundle exec rake test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

当然，我们还可以在浏览器中查看这个页面（[图 3.5](#)），以防测试欺骗我们。

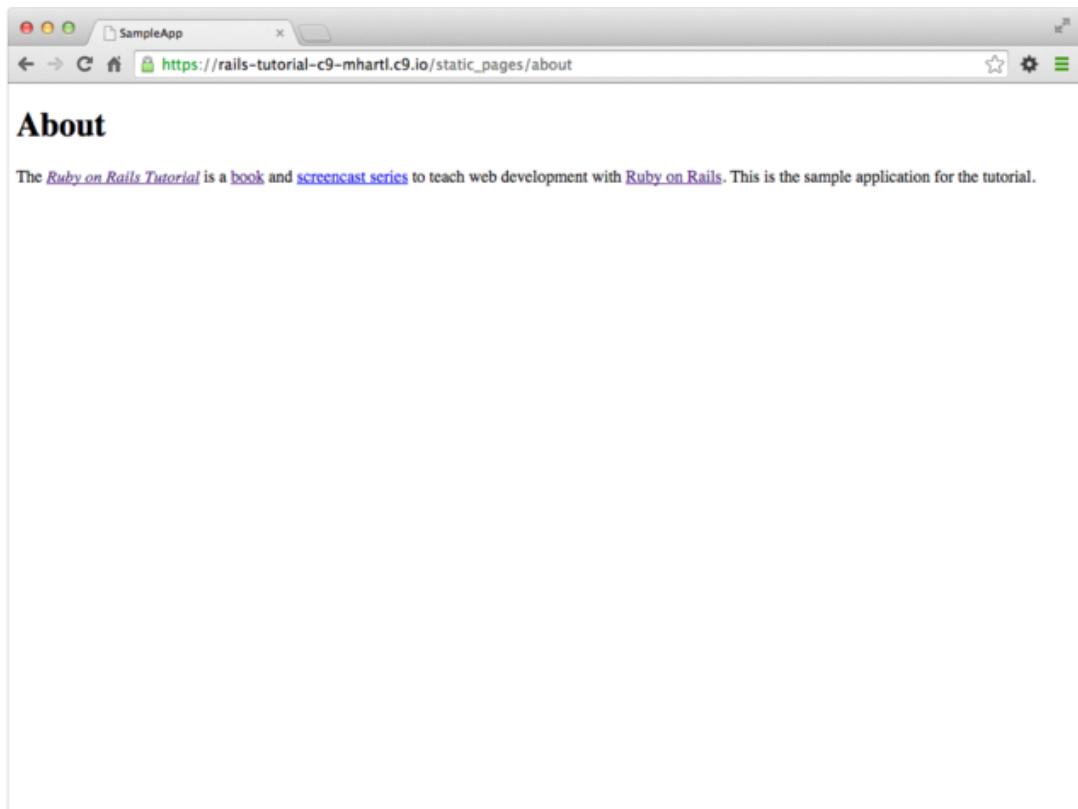


图 3.5：新添加的“关于”页面（/static_pages/about）

3.3.4. 重构

现在测试已经变绿了，我们可以自信地尽情重构了。开发应用时，代码经常会“变味”（意思是代码会变得丑陋、啰嗦，有大量的重复）。电脑不会在意，但是人类会，所以经常重构把代码变简洁一些是很重要的事情。我们的演示应用现在还很小，没什么可重构的，不过代码无时无刻不在变味，所以 3.4.3 节就要开始重构。

3.4. 有点动态内容的页面

我们已经为一些静态页面创建了动作和视图，现在要稍微添加一些动态内容，根据所在的页面不同而变化：我们要让标题根据页面的内容变化。改变标题到底算不算真正动态还有争议，这么做能为第 7 章实现的真正动态内容打下基础。

我们的计划是修改首页、“帮助”页面和“关于”页面，让每页显示的标题都不一样。为此，我们要在页面的视图中使用 `<title>` 标签。大多数浏览器都会在浏览器窗口的顶部显示标题中的内容，而且标题对“搜索引擎优化”（Search-Engine Optimization，简称 SEO）也有好处。我们要使用完整的“遇红-变绿-重构”循环：先为页面的标题编写一些简单的测试（遇红），然后分别在三个页面中添加标题（变绿），最后使用布局文件去除重复内容（重构）。本节结束时，三个静态页面的标题都会变成“<页面的名字> | Ruby on Rails Tutorial Sample App”这种形式（表 3.2）。

`rails new` 命令会创建一个布局文件，不过现在最好不用。我们重命名这个文件：

```
$ mv app/views/layouts/application.html.erb layout_file # 临时移动
```

在真实的应用中你不需要这么做，不过没有这个文件能让你更好地理解它的作用。

表 3.2：演示应用中基本上是静态内容的页面

页面	URL	基本标题	变动部分
首页	/static_pages/home	"Ruby on Rails Tutorial Sample App"	"Home"
帮助	/static_pages/help	"Ruby on Rails Tutorial Sample App"	"Help"
关于	/static_pages/about	"Ruby on Rails Tutorial Sample App"	"About"

3.4.1. 测试标题（遇红）

添加标题之前，我们要学习网页的一般结构，如[代码清单 3.21](#) 所示。

代码清单 3.21：网页一般的 HTML 结构

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

这段代码的最顶部是“文档类型声明”（document type declaration，简称 doctype），告诉浏览器使用哪个 HTML 版本（本例使用 [HTML5](#)⁸）。随后是 head 部分，包含一个 title 标签，其中的内容是“Greeting”。然后是 body 部分，包含一个 p 标签（段落），其中的内容是“Hello, world!”。（内容的缩进是可选的，HTML 不会特别对待空白，制表符和空格都会被忽略，但缩进可以让文档结构更清晰。）

我们要使用 `assert_select` 方法分别为[表 3.2](#) 中的每个标题编写简单的测试，合并到[代码清单 3.13](#) 的测试中。`assert_select` 方法的作用是检查有没有指定的 HTML 标签。这种方法有时也叫“选择符”，从方法名可以看出这一点。⁹

```
assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
```

这行代码的作用是检查有没有 `<title>` 标签，以及其中的内容是不是字符串“Home | Ruby on Rails Tutorial Sample App”。把这样的代码分别放到三个页面的测试中，得到的结果如[代码清单 3.22](#) 所示。

代码清单 3.22：加入标题测试后的静态页面控制器测试 RED

`test/controllers/static_pages_controller_test.rb`

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase
```

8. HTML 一直在变化，显式声明一个 doctype 可以确保未来浏览器还可以正确解析页面。`<!DOCTYPE html>` 这种极为简单的格式是最新的 HTML 标准 [HTML5](#) 的一个特色。

9. [Rails 指南中介绍测试的文章](#)列出了常用的 MiniTest 断言。

```

test "should get home" do
  get :home
  assert_response :success
  assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
end

test "should get help" do
  get :help
  assert_response :success
  assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
end

test "should get about" do
  get :about
  assert_response :success
  assert_select "title", "About | Ruby on Rails Tutorial Sample App"
end

```

(如果你觉得在标题中重复使用“Ruby on Rails Tutorial Sample App”不妥，可以看一下 3.6 节的练习。)

写好测试之后，应该确认一下现在测试组件是失败的（RED）：

代码清单 3.23: RED

```
$ bundle exec rake test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

3.4.2. 添加页面标题（变绿）

现在，我们要为每个页面添加标题，让前一节的测试通过。参照代码清单 3.21 中的 HTML 结构，把代码清单 3.9 中的首页内容换成代码清单 3.24 中的内容。

代码清单 3.24: 具有完整 HTML 结构的首页 RED

app/views/static_pages/home.html.erb

```

<!DOCTYPE html>
<html>
  <head>
    <title>Home | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

修改之后，首页如图 3.6 所示。¹⁰

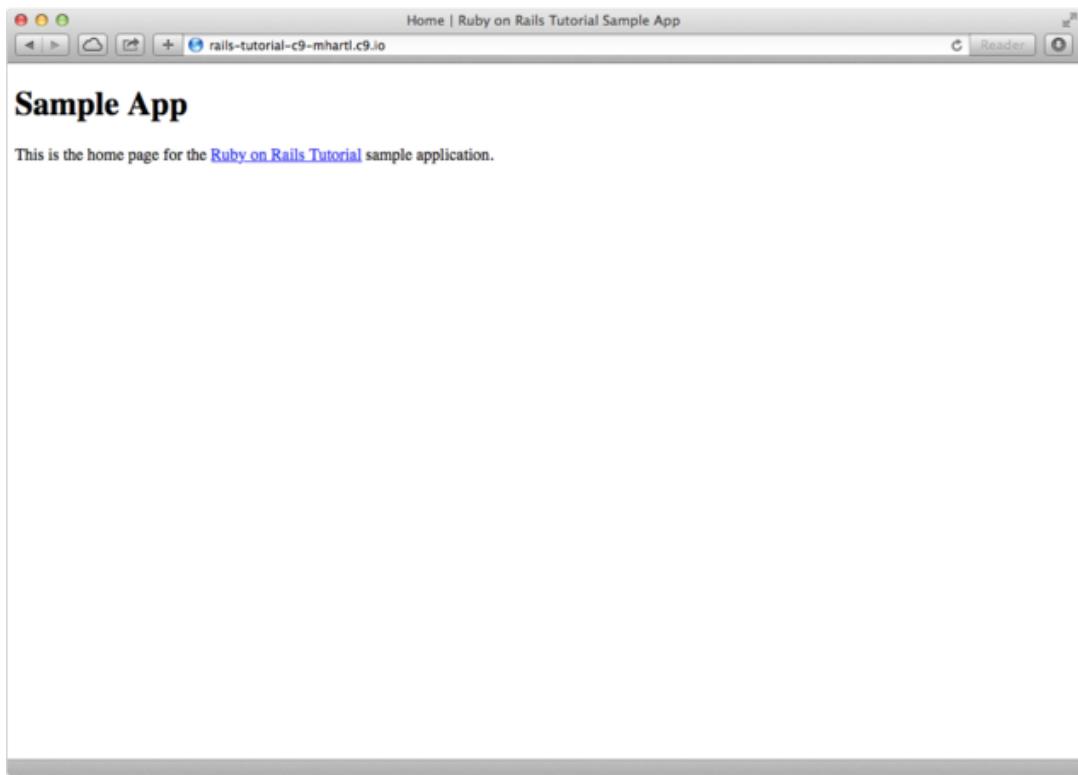


图 3.6：添加标题后的首页

然后使用类似的方式修改“帮助”页面和“关于”页面，得到的代码如代码清单 3.25 和代码清单 3.26 所示。

代码清单 3.25：具有完整 HTML 结构的“帮助”页面 RED

app/views/static_pages/help.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Help | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://www.railstutorial.org/#help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

10. 书中大多数截图都使用 Google Chrome，但是这张截图使用 Safari，因为在 Chrome 没显示完整的标题。

代码清单 3.26：具有完整 HTML 结构的“关于”页面 GREEN

app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>About | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>
```

现在，测试组件能通过了（GREEN）：

代码清单 3.27：GREEN

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

3.4.3. 布局和嵌入式 Ruby（重构）

到目前为止，本节已经做了很多事情，我们使用 Rails 控制器和动作生成了三个可用的页面，不过这些页面中的内容都是纯静态的 HTML，没有体现出 Rails 的强大之处。而且，代码中有着大量重复：

- 页面的标题几乎（但不完全）是一模一样的；
- 每个标题中都有“Ruby on Rails Tutorial Sample App”；
- 整个 HTML 结构在每个页面都重复地出现了。

重复的代码违反了很重要的“不要自我重复”（Don’t Repeat Yourself，简称 DRY）原则。本节要遵照 DRY 原则，去掉重复的代码。最后，我们要运行前一节编写的测试，确认显示的标题仍然正确。

不过，去除重复的第一步却是要增加一些代码，让页面的标题看起来是一样的。这样我们就能更容易地去掉重复的代码了。

在这个过程中，要在视图中使用嵌入式 Ruby（Embedded Ruby）。既然首页、“帮助”页面和“关于”页面的标题中有一个变动的部分，那我们就使用 Rails 提供的一个特别的函数 `provide`，在每个页面中设定不同的标题。通过把 `home.html.erb` 视图中标题的“Home”换成代码清单 3.28 所示的代码，我们可以看一下这个函数的作用。

代码清单 3.28：标题中使用了嵌入式 Ruby 代码的首页视图 GREEN

app/views/static_pages/home.html.erb

```

<% provide(:title, "Home") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>

```

在这段代码中我们第一次使用了嵌入式 Ruby，或者简称 ERb。（现在你应该知道为什么 HTML 视图文件的扩展名是 `.html.erb` 了。）ERb 是为网页添加动态内容主要使用的模板系统。¹¹ 下面的代码

```
<% provide(:title, 'Home') %>
```

通过 `<% ... %>` 调用 Rails 中的 `provide` 函数，把字符串 "Home" 赋给 `:title`。¹² 然后，在标题中，我们使用类似的符号 `<%= ... %>`，通过 Ruby 的 `yield` 函数把标题插入模板中：¹³

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

（这两种嵌入 Ruby 代码的方式区别在于，`<% ... %>` 只执行其中的代码；`<%= ... %>` 也会执行其中的代码，而且会把执行的结果插入模板中。）最终得到的页面和以前一样，不过，现在标题中变动的部分通过 ERb 动态生成。

我们可以运行前一节编写的测试确认一下——测试还能通过（GREEN）：

代码清单 3.29: GREEN

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

然后，按照相同的方式修改“帮助”（[代码清单 3.30](#)）和“关于”页面（[代码清单 3.31](#)）。

代码清单 3.30: 标题中使用了嵌入式 Ruby 代码的“帮助”页面视图 GREEN

`app/views/static_pages/help.html.erb`

```

<% provide(:title, "Help") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>

```

11. 还有一种受欢迎的模板系统是 `Haml`，我个人很喜欢用，不过在这样的初级教程中使用不太合适。

12. 经验丰富的 Rails 开发者可能觉得这里应该使用 `content_for`，可是它在 Asset Pipeline 中有点问题。`provide` 函数是替代方案。

13. 如果你学过 Ruby，可能会猜测 Rails 是把内容“拽入”区块中的，这么想也对。不过使用 Rails 开发应用不必知道这一点。

```

<body>
  <h1>Help</h1>
  <p>
    Get help on the Ruby on Rails Tutorial at the
    <a href="http://www.railstutorial.org/#help">Rails Tutorial help
    section</a>.
    To get help on this sample app, see the
    <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
    Tutorial</em> book</a>.
  </p>
</body>
</html>

```

代码清单 3.31：标题中使用了嵌入式 Ruby 代码的“关于”页面视图 GREEN
`app/views/static_pages/about.html.erb`

```

<% provide(:title, "About") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>

```

至此，我们把页面标题中的变动部分都换成了 ERb。现在，各个页面的内容类似下面这样：

```

<% provide(:title, "The Title") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    Contents
  </body>
</html>

```

也就是说，所有的页面结构都是一致的，包括 `title` 标签中的内容，只有 `body` 标签中的内容有些差别。

为了提取出共用的结构，Rails 提供了一个特别的布局文件，名为 `application.html.erb`。我们在 3.4 节重命名了这个文件，现在改回来：

```
$ mv layout_file app/views/layouts/application.html.erb
```

若想使用这个布局，我们要把默认的标题换成前面几段代码中使用的嵌入式 Ruby：

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

修改后得到的布局文件如[代码清单 3.32](#) 所示。

代码清单 3.32：这个演示应用的网站布局 GREEN

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

注意，其中有一行比较特殊：

```
<%= yield %>
```

这行代码的作用是，把每个页面的内容插入布局中。没必要了解它的具体实现过程，我们只需知道，在布局中使用这行代码后，访问 `/static_pages/home` 时会把 `home.html.erb` 中的内容转换成 HTML，然后插入 `<%= yield %>` 所在的位置。

还要注意，默认的 Rails 布局文件中还有下面这几行代码：

```
<%= stylesheet_link_tag ... %>
<%= javascript_include_tag "application", ... %>
<%= csrf_meta_tags %>
```

这几行代码的作用是，引入应用的样式表和 JavaScript 文件（Asset Pipeline 的一部分，[5.2.1 节](#)会介绍）；Rails 中的 `csrf_meta_tags` 方法，作用是避免“跨站请求伪造”（Cross-Site Request Forgery，简称 CSRF，一种恶意网络攻击）。

现在，[代码清单 3.28、3.30 和 3.31](#) 的内容还是和布局文件中类似的 HTML 结构，所以我们要把完整的结构删除，只保留需要的内容。清理后的视图如[代码清单 3.33、3.34 和 3.35](#) 所示。

代码清单 3.33：去除完整的 HTML 结构后的首页 GREEN

app/views/static_pages/home.html.erb

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
```

```
    sample application.  
  </p>
```

代码清单 3.34：去除完整的 HTML 结构后的“帮助”页面 GREEN

app/views/static_pages/help.html.erb

```
<% provide(:title, "Help") %>  
<h1>Help</h1>  
<p>  
  Get help on the Ruby on Rails Tutorial at the  
  <a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.   
  To get help on this sample app, see the  
  <a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>  
  book</a>.  
</p>
```

代码清单 3.35：去除完整的 HTML 结构后的“关于”页面 GREEN

app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>  
<h1>About</h1>  
<p>  
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails  
  Tutorial</em></a> is a  
  <a href="http://www.railstutorial.org/book">book</a> and  
  <a href="http://screencasts.railstutorial.org/">screencast series</a>  
  to teach web development with  
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.  
  This is the sample application for the tutorial.  
</p>
```

修改这几个视图后，首页、“帮助”页面和“关于”页面显示的内容还和之前一样，但是没有多少重复内容了。

经验告诉我们，即便是十分简单的重构，也容易出错，所以才要认真编写测试组件。有了测试，我们就无需手动检查每个页面，看有没有错误。初期阶段手动检查还不算难，但是当应用不断变大之后，情况就不同了。我们只需验证测试组件是否还能通过即可：

代码清单 3.36：GREEN

```
$ bundle exec rake test  
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

测试不能证明代码完全正确，但至少能提高正确的可能性，而且还提供了安全防护措施，避免以后出现问题。

3.4.4. 设置根路由

我们修改了网站中的页面，也顺利开始编写测试了，在继续之前，我们要设置应用的根路由。与 1.3.4 节和 2.2.2 节的做法一样，我们要修改 `routes.rb` 文件，把根路径 `/` 指向我们选择的页面。这里我们要指向前面创建的首页。（我还建议把 3.1 节添加的 `hello` 动作从应用的控制器中删除。）如 代码清单 3.37 所示，我们要把自动生成的 `get` 规则（代码清单 3.5）改成：

```
root 'static_pages#home'
```

我们把 `static_pages/home` 改成 `static_pages#home`，确保通过 GET 请求访问 / 时，会交给静态页面路由器中的 `home` 动作处理。修改路由后，首页如图 3.7 所示。（注意，修改路由之后，`/static_pages/home` 就无法访问了。）

代码清单 3.37：把根路由指向首页

`config/routes.rb`

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'static_pages/help'
  get 'static_pages/about'
end
```

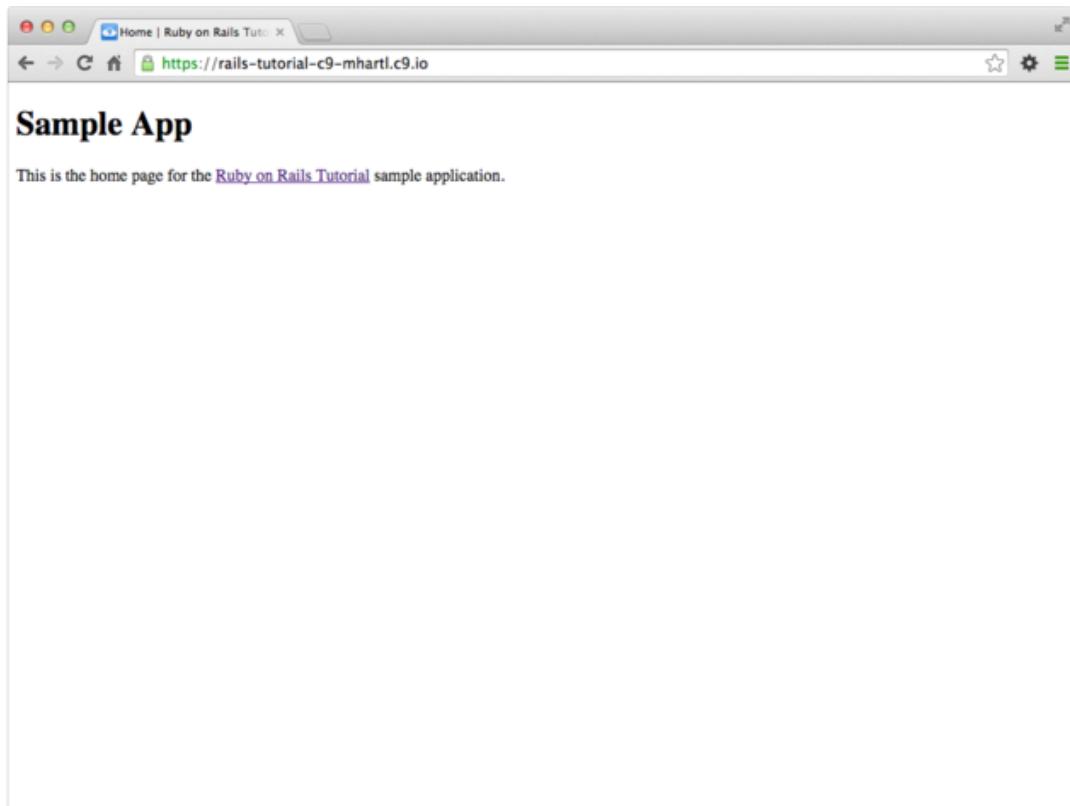


图 3.7：在根路由上显示的首页

3.5. 小结

总的来说，本章几乎没有做什么：我们从静态页面开始，最后得到的还几乎是静态内容的页面。不过从表面来看，我们使用了 Rails 中的控制器、动作和视图，现在我们已经可以向网站中添加任意的动态内容了。本书的后续内容会告诉你怎么添加。

在继续之前，我们花一点时间把改动提交到主题分支，然后将其合并到主分支中。在 3.2 节，我们为静态页面的开发工作创建了一个 Git 新分支，在开发的过程中如果你还没有提交，那么先来做一次提交吧，因为我们已经完成了一些工作：

```
$ git add -A
$ git commit -m "Finish static pages"
```

然后，使用 1.4.4 节介绍的方法，把改动合并到主分支中：

```
$ git checkout master  
$ git merge static-pages
```

每次完成一些工作后，最好把代码推送到远程仓库（如果你按照 1.4.3 节中的步骤做了，远程仓库在 Bitbucket 上）中：

```
$ git push
```

我还建议你把这个应用部署到 Heroku 中：

```
$ bundle exec rake test  
$ git push heroku
```

在部署之前，我们先运行测试组件——这是一个好习惯。

3.5.1. 读完本章学到了什么

- 我们第三次介绍从零开始创建一个新 Rails 应用的完整过程，包括安装所需的 gem，把应用推送到远程仓库，以及部署到生产环境中；
- 执行 `rails generate controller ControllerName <optional action names>` 命令会生成一个新控制器；
- 在 `config/routes.rb` 文件中定义了新路由；
- Rails 的视图中可以包含静态 HTML 及嵌入式 Ruby 代码（ERb）；
- 测试组件能驱动我们开发新功能，给我们重构的自信，以及捕获回归；
- 测试驱动开发使用“遇红-变绿-重构”循环；
- Rails 的布局定义页面共用的结构，可以去除重复。

3.6. 练习

从这以后，我建议在单独的主题分支中做练习：

```
$ git checkout static-pages  
$ git checkout -b static-pages-exercises
```

这么做就不会和本书正文产生冲突了。

练习做完后，可以把相应的分支推送到远程仓库中（如果有远程仓库的话）：

```
# 做完第一个练习后  
$ git commit -am "Eliminate repetition (solves exercise 3.1)"  
# 做完第二个练习后  
$ git add -A  
$ git commit -m "Add a Contact page (solves exercise 3.2)"  
$ git push -u origin static-pages-exercises  
$ git checkout master
```

（为了继续后面的开发，最后一步只切换到主分支，但不合并，以免和正文产生冲突。）在后面的章节中，使用的分支和提交消息有所不同，但基本思想是一致的。

- 你可能注意到了，静态页面控制的测试（[代码清单 3.22](#)）中有些重复，每个标题测试中都有“Ruby on Rails Tutorial Sample App”。我们要使用特殊的函数 `setup` 去除重复。这个函数在每个测试运行之前执行。请你确认[代码清单 3.38](#) 中的测试仍能通过。（[代码清单 3.38](#) 中使用了一个实例变量，[2.2.2 节](#) 简单介绍过，[4.4.5 节](#) 会进一步介绍。这段代码中还使用了字符串插值操作，[4.2.2 节](#) 会做介绍。）
- 为这个演示应用添加一个“联系”页面。¹⁴ 参照[代码清单 3.13](#)，先编写一个测试，检查页面的标题是否为“Contact | Ruby on Rails Tutorial Sample App”，从而确定 `/static_pages/contact` 对应的页面是否存在。把[代码清单 3.39](#) 中的内容写入“联系”页面的视图，让测试通过。注意，这个练习是独立的，所以[代码清单 3.39](#) 中的代码不会影响[代码清单 3.38](#) 中的测试。

代码清单 3.38：使用了通用标题的静态页面控制器测试 GREEN

test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  def setup
    @base_title = "Ruby on Rails Tutorial Sample App"
  end

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Home | #{@base_title}"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | #{@base_title}"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | #{@base_title}"
  end
end
```

代码清单 3.39：“联系”页面的内容

app/views/static_pages/contact.html.erb

```
<% provide(:title, "Contact") %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="http://www.railstutorial.org/#contact">contact page</a>.
</p>
```

14. 这个练习会在[5.3.1 节](#)完成。

3.7. 高级测试技术

这一节选读，介绍本书配套视频中使用的测试设置，包含三方面内容：增强版通过和失败报告程序（3.7.1 节）；过滤测试失败消息中调用跟踪的方法（3.7.2 节）；一个自动测试运行程序，检测到文件有变化后自动运行相应的测试（3.7.3 节）。这一节使用的代码相对高级，放在这里只是为了查阅方便，现在并不期望你能理解。

这一节应该在主分支中修改：

```
$ git checkout master
```

3.7.1. MiniTest 报告程序

为了让 Rails 中的测试适时显示红色和绿色，我建议你在测试辅助文件中加入[代码清单 3.40](#) 中的内容，¹⁵ 充分利用[代码清单 3.2](#) 中的 `minitest-reporters` gem。

代码清单 3.40：配置测试，显示红色和绿色

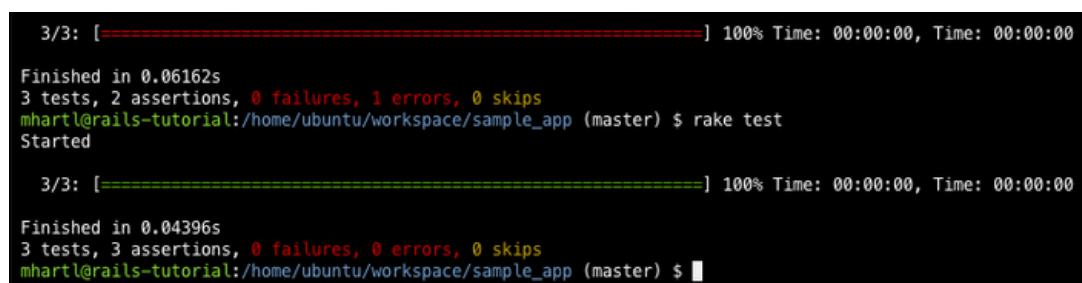
`test/test_helper.rb`

```
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../config/environment', __FILE__)
require 'rails/test_help'
require "minitest/reporters"
Minitest::Reporters.use!

class ActiveSupport::TestCase
  # Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical
  # order.
  fixtures :all

  # Add more helper methods to be used by all tests here...
end
```

修改后，在云端 IDE 中显示的效果如图 3.8 所示。



```
3/3: [=====] 100% Time: 00:00:00, Time: 00:00:00
Finished in 0.06162s
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
mhartl@rails-tutorial:/home/ubuntu/workspace/sample_app (master) $ rake test
Started

3/3: [=====] 100% Time: 00:00:00, Time: 00:00:00
Finished in 0.04396s
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
mhartl@rails-tutorial:/home/ubuntu/workspace/sample_app (master) $
```

图 3.8：在云端 IDE 中测试由红变绿

15. 这段代码既使用了单引号形式字符串，又使用了双引号形式字符串，因为 `rails new` 命令生成的文件使用单引号字符串，而 [MiniTest 报告程序的文档](#) 中使用双引号。在 Ruby 代码中混用两种形式的字符串很常见，详情参见 [4.2.2 节](#)。

3.7.2. 调用跟踪静默程序

如果有错误，或者测试失败，测试运行程序会显示调用跟踪，从失败的测试开始一直追溯到应用代码。调用跟踪对查找问题来说很有用，但在某些系统中（包括云端 IDE），会一直追溯到应用的代码以及各个 gem（包括 Rails）中，显示的内容往往很多。如果问题发生在应用代码中，而不是它的依赖件中，那么内容更多。

我们可以过滤调用追踪，不显示不需要的内容。为此，我们要使用[代码清单 3.2](#) 中的 `mini_backtrace` gem，然后再设置静默程序。在云端 IDE 中，大多数不需要的内容都包含字符串“rvm”（指的是 Ruby Version Manager），所以我建议使用[代码清单 3.41](#) 中的静默程序把这些内容过滤掉。

代码清单 3.41：添加调用跟踪静默程序，过滤 RVM 相关的内容

`config/initializers/backtrace_silencers.rb`

```
# Be sure to restart your server when you modify this file.

# You can add backtrace silencers for libraries that you're using but don't
# wish to see in your backtraces.
Rails.backtrace_cleaner.add_silencer { |line| line =~ /rvm/ }

# You can also remove all the silencers if you're trying to debug a problem
# that might stem from framework code.
# Rails.backtrace_cleaner.remove_silencers!
```

如这段代码中的注释所说，添加静默程序后要重启本地服务器。

3.7.3. 使用 Guard 自动测试

使用 `rake test` 命令有一点很烦人，总是要切换到命令行然后手动运行测试。为了避免这种不便，我们可以使用 `Guard` 自动运行测试。`Guard` 会监视文件系统的变动，假如你修改了 `static_pages_controller_test.rb`，那么 `Guard` 只会运行这个文件中的测试。而且，我们还可以配置 `Guard`，让它在 `home.html.erb` 文件被修改后，也自动运行 `static_pages_controller_test.rb`。

[代码清单 3.2](#) 中已经包含了 `guard` gem，所以我们只需初始化即可：

```
$ bundle exec guard init
Writing new Guardfile to /home/ubuntu/workspace/sample_app/Guardfile
00:51:32 - INFO - minitest guard added to Guardfile, feel free to edit it
```

然后再编辑生成的 `Guardfile` 文件，让 `Guard` 在集成测试和视图发生变化后运行正确的测试，如[代码清单 3.42](#) 所示。（这个文件的内容很长，而且需要高级知识，所以我建议直接复制粘贴。）

代码清单 3.42：修改 `Guardfile`

```
# Defines the matching rules for Guard.
guard :minitest, spring: true, all_on_start: false do
  watch(%r{^test/(.*)/?(.*)_test\.rb$})
  watch('test/test_helper.rb') { 'test' }
  watch('config/routes.rb') { integration_tests }
  watch(%r{^app/models/(.*?)\.rb$}) do |matches|
    "test/models/#{matches[1]}_test.rb"
  end
  watch(%r{^app/controllers/(.*?)_controller\.rb$}) do |matches|
```

```

    resource_tests(matches[1])
end
watch(%r{^app/views/([^\?]*?)\.*\.html\.erb$}) do |matches|
  ["test/controllers/#{matches[1]}_controller_test.rb"] +
  integration_tests(matches[1])
end
watch(%r{^app/helpers/(.*?)_helper\.rb$}) do |matches|
  integration_tests(matches[1])
end
watch('app/views/layouts/application.html.erb') do
  'test/integration/site_layout_test.rb'
end
watch('app/helpers/sessions_helper.rb') do
  integration_tests << 'test/helpers/sessions_helper_test.rb'
end
watch('app/controllers/sessions_controller.rb') do
  ['test/controllers/sessions_controller_test.rb',
   'test/integration/users_login_test.rb']
end
watch('app/models/micropost.rb') do
  ['test/models/micropost_test.rb', 'test/models/user_test.rb']
end
watch(%r{app/views/users/*}) do
  resource_tests('users') +
  ['test/integration/microposts_interface_test.rb']
end
end

# Returns the integration tests corresponding to the given resource.
def integration_tests(resource = :all)
  if resource == :all
    Dir["test/integration/*"]
  else
    Dir["test/integration/#{resource}/*.rb"]
  end
end

# Returns the controller tests corresponding to the given resource.
def controller_test(resource)
  "test/controllers/#{resource}_controller_test.rb"
end

# Returns all tests for the given resource.
def resource_tests(resource)
  integration_tests(resource) << controller_test(resource)
end

```

下面这行

```
guard :minitest, spring: true, all_on_start: false do
```

会让 Guard 使用 Rails 提供的 Spring 服务器减少加载时间，而且启动时不运行整个测试组件。

使用 Guard 时，为了避免 Spring 和 Git 发生冲突，应该把 `spring/` 文件夹加到 `.gitignore` 文件中，让 Git 忽略这个文件夹。在云端 IDE 中要这么做：

- 点击文件浏览器右上角的齿轮图标，如图 3.9 所示；
- 选择“Show hidden files”（显示隐藏文件），让 `.gitignore` 文件出现在应用的根目录中，如图 3.10 所示；
- 双击打开 `.gitignore` 文件（图 3.11），写入代码清单 3.43 中的内容。

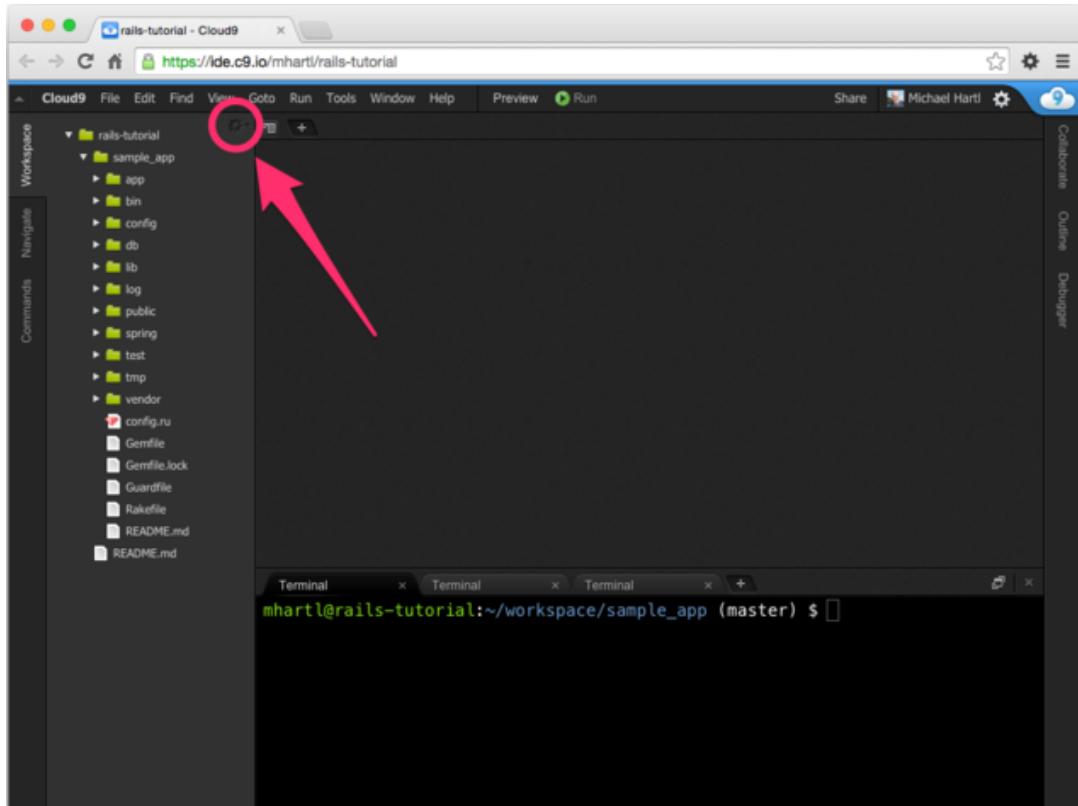


图 3.9：文件浏览器中的齿轮图标（不太好找）

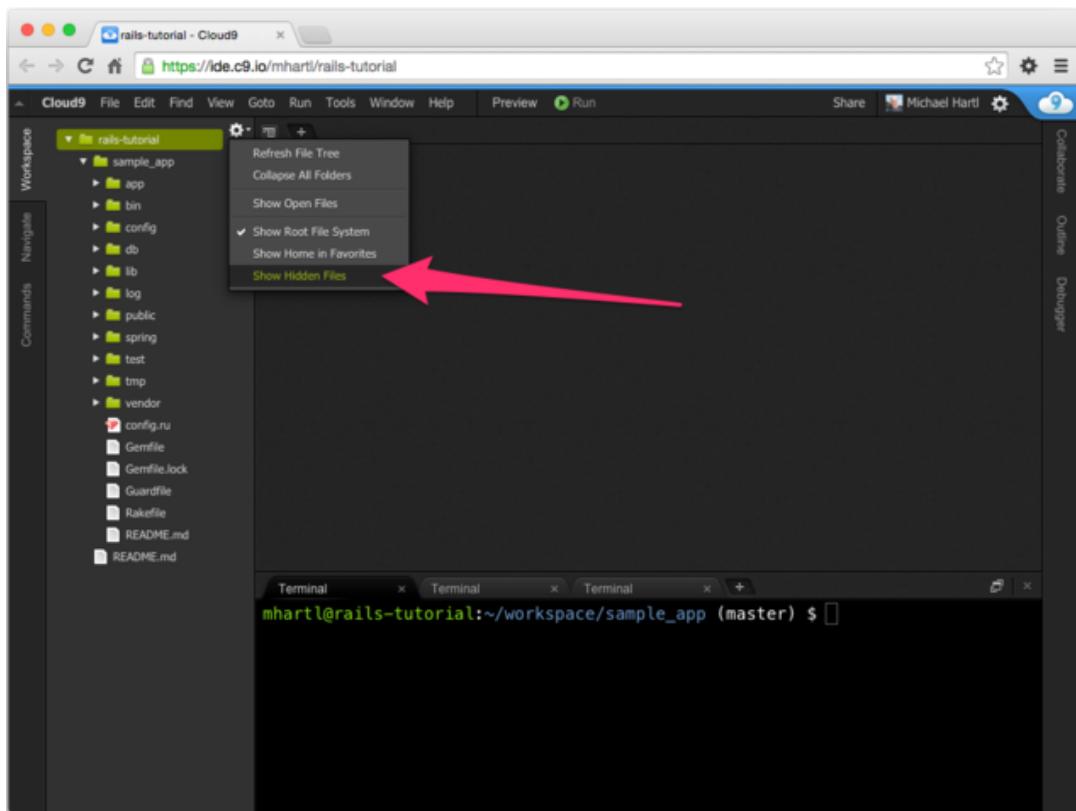


图 3.10：显示隐藏文件

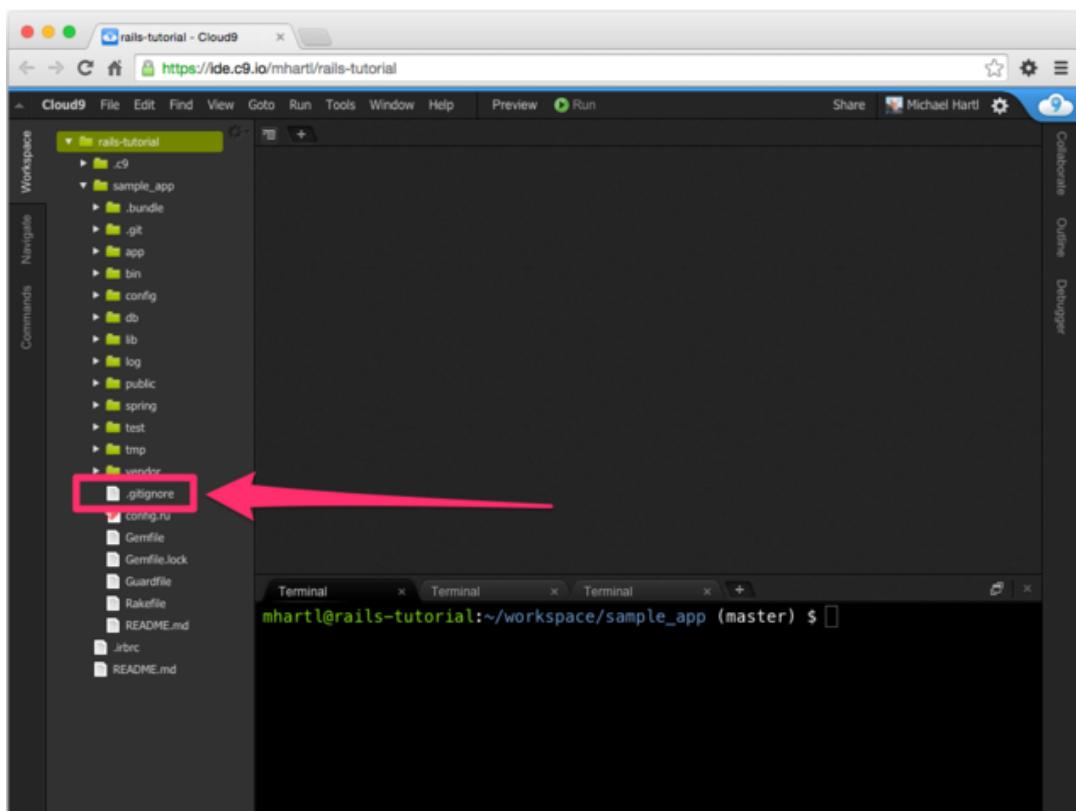


图 3.11：通常隐藏的 .gitignore 文件出现了

代码清单 3.43：把 Spring 添加到 .gitignore 文件中

```
# See https://help.github.com/articles/ignoring-files for more about ignoring
# files.
#
# If you find yourself ignoring temporary files generated by your text editor
# or operating system, you probably want to add a global ignore instead:
#   git config --global core.excludesfile '~/.gitignore_global'

# Ignore bundler config.
/.bundle

# Ignore the default SQLite database.
/db/*.sqlite3
/db/*.sqlite3-journal

# Ignore all logfiles and tempfiles.
/log/*.log
/tmp

# Ignore Spring files.
/spring/*.pid
```

写作本书时，Spring 服务器还有点儿怪异，有时 Spring 进程会不断拖慢测试的运行速度。如果你发现测试变得异常缓慢，最好查看系统进程（[旁注 3.4](#)），如果需要，关掉 Spring 进程。

旁注 3.4：Unix 进程

在 Unix 类系统中，例如 Linux 和 OS X，用户和系统执行的任务都在包装良好的容器中，这个容器叫“进程”（process）。若想查看系统中的所有进程，可以执行 `ps` 命令，并指定 `aux` 参数：

```
$ ps aux
```

若想过滤输出的进程，可以使用 Unix 管道操作（|）把 `ps` 命令的结果传给 `grep`，进行模式匹配：

```
$ ps aux | grep spring
ubuntu 12241 0.3 0.5 589960 178416 ? Ssl Sep20 1:46
spring app | sample_app | started 7 hours ago
```

显示的结果中有进程的部分详细信息，其中最重要的是第一个数字，即进程的 ID，简称 pid。若要终止不想要的进程，可以使用 `kill` 命令，向指定的 pid 发送 kill 信号（[恰巧是 9](#)）：

```
$ kill -9 12241
```

关闭单个进程，例如不再使用的 Rails 服务器进程（可执行 `ps aux | grep server` 命令找到 pid），我推荐使用这种方法。不过，有时最好能批量关闭进程名中包含特定文本的进程，例如关闭系统中所有的 `spring` 进程。针对 Spring，首先应该尝试使用 `spring` 命令关闭进程：

```
$ spring stop
```

不过，有时这么做没用，那么可以使用 `pkill` 命令关闭所有名为“spring”的进程：

```
$ pkill -9 -f spring
```

只要发现表现异常，或者进程静止了，最好执行 `ps aux` 命令看看怎么回事，然后再执行 `kill -9 <pid>` 或 `pkill -9 -f <name>` 命令关闭进程。

配置好 Guard 之后，应该打开一个新终端窗口（和 1.3.2 节启动 Rails 服务器的做法一样），在其中执行下述命令：

```
$ bundle exec guard
```

代码清单 3.42 中的规则针对本书做了优化，例如，修改控制器后会自动运行集成测试。如果想运行所有测试，在 `guard>` 终端中按回车键。（有时会看到一个错误，说连接 Spring 服务器失败。再次按回车键就能解决这个问题。）

若想退出 Guard，按 Ctrl-D 键。如果想为 Guard 添加其他的匹配器，参照代码清单 3.42，Guard 的说明文件和维基。

第 4 章 Rails 背后的 Ruby

有了[第 3 章](#)的例子做铺垫，本章要介绍一些对 Rails 来说很重要的 Ruby 知识。Ruby 语言的知识点很多，不过对 Rails 开发者而言需要掌握的很少。我们采用的方式有别于常规的 Ruby 学习过程。本章的目标是，不管你有没有 Ruby 编程经验，都得让你掌握编写 Rails 应用所需的 Ruby 知识。这一章的内容很多，第一次阅读不能完全掌握也没关系。后续的章节我会经常提到本章的内容。

4.1. 导言

从前一章得知，即使完全不懂 Ruby 语言，我们也可以创建 Rails 应用的骨架，以及编写测试。我们依赖于书中提供的测试代码，得到错误信息，然后让测试组件通过。但是我们不能总是这样，所以这一章暂时不讲网站开发，而要正视我们的短肋——Ruby 语言。

前一章末尾我们修改了几乎是静态内容的页面，让它们使用 Rails 布局，去除视图中的重复。我们使用的布局如[代码清单 4.1](#) 所示（和[代码清单 3.32](#) 一样）。

代码清单 4.1：演示应用的网站布局

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

我们把注意力集中在上述代码中的这一行：

```
<%= stylesheet_link_tag "application", media: "all",
                        "data-turbolinks-track" => true %>
```

这行代码使用 Rails 内置的方法 `stylesheet_link_tag`（详细信息参见 [Rails API 文档](#)），在所有媒介类型中引入 `application.css`。对有经验的 Rails 开发者来说，这行代码看起来很简单，但是其中至少有四个 Ruby 知识点可能会让你困惑：内置的 Rails 方法，调用方法时不用括号，符号和哈希。这几点本章都会介绍。

Rails 除了提供很多内置的方法供我们在视图中使用之外，还允许我们自己定义。自己定义的方法叫辅助方法（helper）。为了说明如何自己定义辅助方法，我们来看看[代码清单 4.1](#) 中标题那一行：

```
<%= yield(:title) %> | Ruby on Rails Tutorial Sample App
```

这行代码要求每个视图都要使用 `provide` 方法定义标题，例如：

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

那么，如果我们不提供标题会怎样呢？标题一般都包含一个公共部分，如果想更具体些，可以再加上变动的部分。我们在布局中用了个小技巧，基本上已经实现了这样的标题。如果在视图中不调用 `provide` 方法，也就是不提供变动的部分，那么得到的标题会变成：

```
| Ruby on Rails Tutorial Sample App
```

标题中有公共部分，但前面还显示了竖线。

为了解决这个问题，我们要自定义一个辅助方法，命名为 `full_title`。如果视图中没有定义页面的标题，`full_title` 返回标题的公共部分，即“Ruby on Rails Tutorial Sample App”；如果定义了，则在变动部分后面加上一个竖线，如 [代码清单 4.2](#) 所示。¹

代码清单 4.2：定义 `full_title` 辅助方法

`app/helpers/application_helper.rb`

```
module ApplicationHelper

  # 根据所在的页面返回完整的标题
  def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      "#{page_title} | #{base_title}"
    end
  end
end
```

现在，这个辅助方法定义好了，我们可以用它来简化布局。把下面这行：

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

改成：

```
<title><%= full_title(yield(:title)) %></title>
```

如 [代码清单 4.3](#) 所示。

代码清单 4.3：使用 `full_title` 辅助方法的网站布局 GREEN

`app/views/layouts/application.html.erb`

1. 如果辅助方法是针对某个特定控制器的，应该把它放进该控制器对应的辅助文件中。例如，为静态页面控制器创建的辅助方法一般放在 `app/helper/static_pages_helper.rb` 中。在这个例子中，我们想在所有页面中都使用 `full_title` 方法，所以要放在一个特殊的辅助文件中，即 `app/helper/application_helper.rb`。

```

<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>

```

为了让这个辅助方法起作用，我们要在首页的视图中把不必要的单词“Home”删掉，只保留标题的公共部分。首先，我们要修改测试代码，如[代码清单 4.4](#) 所示，确认标题中没有 “Home”。（注意，如果你做了前一章的练习，按照[代码清单 3.38](#) 修改了测试，现在要把定义 `@base_title` 变量的 `setup` 方法删掉。）

代码清单 4.4: 修改首页的标题测试 RED

test/controllers/static_pages_controller_test.rb

```

require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase
  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end

```

我们要运行测试组件，确认有一个测试失败：

代码清单 4.5: RED

```

$ bundle exec rake test
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips

```

为了让测试通过，我们要把首页视图中的 `provide` 那行删除，如[代码清单 4.6](#) 所示。

代码清单 4.6: 没定义页面标题的首页视图 GREEN

app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

现在测试应该可以通过了：

代码清单 4.7: GREEN

```
$ bundle exec rake test
```

(注意，之前运行 `rake test` 时都显示了通过和失败测试的数量，为了行文简洁，从这以后都会省略这些数据。)

和引入应用的样式表那行代码一样，[代码清单 4.2](#) 的内容对有经验的 Rails 开发者来说也很简单，但其中很多重要的 Ruby 知识：模块，方法定义，可选的方法参数，注释，本地变量赋值，布尔值，流程控制，字符串插值和返回值。本章会一一介绍这些知识。

4.2. 字符串和方法

我们学习 Ruby 主要使用的工具是 Rails 控制台，它是用来和 Rails 应用交互的命令行工具，在[2.3.3 节](#)介绍过。控制台基于 Ruby 的交互程序 (`irb`) 开发，因此能使用 Ruby 语言的全部功能。[\(4.4.4 节\)](#) 会介绍，控制台还可以访问 Rails 环境。)

如果使用云端 IDE，我建议使用一些 `irb` 配置参数。使用简单的 `nano` 文本编辑器，把[代码清单 4.8](#) 中的内容写入家目录里的 `.irbrc` 文件：²

```
$ nano ~/.irbrc
```

[代码清单 4.8](#) 中的内容作用是简化 `irb` 提示符，以及禁用一些烦人的自动缩进行为。

代码清单 4.8: 添加一些 irb 配置

```
~/.irbrc
```

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

不管加没加这些设置，控制器的启动方法都是在命令行中执行下面的命令：

```
$ rails console
Loading development environment
>>
```

默认情况下，控制台在开发环境中启动，这是 Rails 定义的三个独立环境之一（另外两个是测试环境和生产环境）。这三个环境的区别对本章不重要，[7.1.1 节](#) 会详细介绍。

控制台是学习的好工具，你可以尽情地探索它的用法。别担心，你（几乎）不会破坏任何东西。如果在控制台中遇到了问题，可以按 `Ctrl-C` 键结束当前执行的操作，或者按 `Ctrl-D` 键直接退出。在阅读本章后续内容的

2. 译者注：写完后，在命令行中按 `Ctrl-W` 键保存，然后按回车键确认保存在指定的文件中，最后按 `Ctrl-E` 键退出 `nano`。

过程中，你会发现查阅 Ruby API 很有帮助。API 中有很多信息（或许太多了），例如，如果想进一步了解 Ruby 字符串，可以查看 `String` 类的文档。

4.2.1. 注释

Ruby 中的注释以井号 `#`（也叫“哈希符号”，或者更诗意一点，叫“散列字元”）开头，一直到行尾结束。Ruby 会忽略注释，但是注释对人类读者（往往也包括代码的编写者）很有用。在下面的代码中

```
# 根据所在的页面返回完整的标题
def full_title(page_title = '')
  ...
end
```

第一行就是注释，说明其后方法的作用。

在控制台中一般不用写注释，不过为了说明代码的作用，我会按照下面的形式加上注释，例如：

```
$ rails console
>> 17 + 42    # 整数加法运算
=> 59
```

阅读的过程中，在控制台中输入或者复制粘贴命令时，如果愿意你可以不加注释，反正控制台会忽略注释。

4.2.2. 字符串

对 Web 应用来说，字符串或许是最重要的数据结构，因为网页的内容就是从服务器发送给浏览器的字符串。我们先在控制台中体验一下字符串：

```
$ rails console
>> ""          # 空字符串
=> ""
>> "foo"        # 非空字符串
=> "foo"
```

这些是字符串字面量，使用双引号（"）创建。控制台回显的是每一行的计算结果，本例中，字符串字面量的结果就是字符串本身。

我们还可以使用 `+` 号连接字符串：

```
>> "foo" + "bar"    # 字符串连接
=> "foobar"
```

"foo" 连接 "bar" 得到的结果是字符串 "foobar"。³

另一种创建字符串的方式是通过特殊的句法 `#{}</code>` 进行插值操作：⁴

3. 关于“foo”和“bar”，以及不太相关的“foobar”和“FUBAR”的起源，请查看 [Jargon File 中介绍“foo”的文章](#)。

4. 熟悉 Perl 或 PHP 的编程人员，可以把这个功能与自动插值美元符号开头的变量相对应，例如 `"foo $bar"`。

```
>> first_name = "Michael"      # 变量赋值
=> "Michael"
>> "#{first_name} Hartl"      # 字符串插值
=> "Michael Hartl"
```

我们先把 "Michael" 赋值给变量 `first_name`, 然后将其插入字符串 "`#{first_name} Hartl`" 中。我们也可以把两个字符串都赋值给变量:

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name    # 字符串连接, 中间加了空格
=> "Michael Hartl"
>> "#{first_name} #{last_name}"    # 作用相同的插值
=> "Michael Hartl"
```

注意, 最后两个表达式的作用相同, 不过我倾向于使用插值的方式。在两个字符串中间加入一个空格 (" ") 显得很别扭。

打印字符串

打印字符串最常用的 Ruby 方法是 `puts` (读作“put ess”, 意思是“打印字符串”):

```
>> puts "foo"      # 打印字符串
foo
=> nil
```

`puts` 方法还有一个副作用: `puts "foo"` 先把字符串打印到屏幕上, 然后返回空值字面量——`nil` 在 Ruby 中是个特殊值, 表示“什么都没有”。(为了行文简洁, 后续内容会省略 $\Rightarrow \text{nil}$ 。)

从前面的例子可以看出, `puts` 方法会自动在输出的字符串后面加入换行符 `\n`。功能类似的 `print` 方法则不会:

```
>> print "foo"      # 打印字符串 (和 puts 作用一样, 但没添加换行符)
foo=> nil
>> print "foo\n"    # 和 puts "foo" 一样
foo
=> nil
```

单引号字符串

目前介绍的例子都使用双引号创建字符串, 不过 Ruby 也支持用单引号创建字符串。大多数情况下这两种字符串的效果是一样的:

```
>> 'foo'          # 单引号创建的字符串
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

不过, 两种方式之间有个重要的区别: Ruby 不会对单引号字符串进行插值操作:

```
>> '#{foo} bar'    # 单引号字符串不能进行插值操作
=> "\#{foo} bar"
```

注意，控制台返回的是双引号字符串，因此要使用反斜线转义特殊字符，例如`#`。

如果双引号字符串可以做单引号能做的所有事，而且还能进行插值，那么单引号字符串存在的意义是什么呢？单引号字符串的用处在于它们真的就是字面值，只包含你输入的字符。例如，反斜线在很多系统中都很特殊，例如在换行符（`\n`）中。如果有一个变量需要包含一个反斜线，使用单引号就很简单：

```
>> '\n'      # 反斜线和 n 字面值
=> "\n"
```

和前面的`#`字符一样，Ruby 要使用一个额外的反斜线来转义反斜线——在双引号字符串中，要表达一个反斜线就要使用两个反斜线。对简单的例子来说，这省不了多少事，但是如果有很多需要转义的字符就显得出它的作用了：

```
>> 'Newlines (\n) and tabs (\t) both use the backslash character \.'
=> "Newlines (\n) and tabs (\t) both use the backslash character \\."
```

最后，有一点要注意，单双引号基本上可以互换使用，源码中经常混用，没有章法可循，对此我们只能默默接受——“欢迎进入 Ruby 世界”！

4.2.3. 对象和消息传送

在 Ruby 中，一切皆对象，包括字符串和`nil`都是。我们会在 4.4.2 节介绍对象技术层面上的意义，不过一般很难通过阅读一本书就理解对象，你要多看一些例子才能建立对对象的感性认识。

对象的作用说起来很简单：响应消息。例如，一个字符串对象可以响应`length`这个消息，返回字符串中包含的字符数量：

```
>> "foobar".length      # 把 length 消息传给字符串
=> 6
```

一般来说，传给对象的消息是“方法”，是在这个对象上定义的函数。⁵字符串还可以响应`empty?`方法：

```
>> "foobar".empty?
=> false
>> "".empty?
=> true
```

注意，`empty?`方法末尾有个问号，这是 Ruby 的约定，说明方法的返回值是布尔值，即`true`或`false`。布尔值在流程控制中特别有用：

```
>> s = "foobar"
>> if s.empty?
>>   "The string is empty"
>> else
>>   "The string is nonempty"
>> end
=> "The string is nonempty"
```

如果分支很多，可以使用`elsif`（`else + if`）：

5. 很抱歉，本章在“函数”和“方法”两个称呼之间随意变换。在 Ruby 中这二者是同一个概念：所有方法都是函数，所有函数也都是方法，因为一切皆对象。

```

>> if s.nil?
>>   "The variable is nil"
>> elsif s.empty?
>>   "The string is empty"
>> elsif s.include?("foo")
>>   "The string includes 'foo'"
>> end
=> "The string includes 'foo'"

```

布尔值还可以使用 `&&` (和)、`||` (或) 和 `!` (非) 操作符结合在一起使用：

```

>> x = "foo"
=> "foo"
>> y = ""
=> ""
>> puts "Both strings are empty" if x.empty? && y.empty?
=> nil
>> puts "One of the strings is empty" if x.empty? || y.empty?
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil

```

因为在 Ruby 中一切都是对象，那么 `nil` 也是对象，所以它也可以响应方法。举个例子，`to_s` 方法基本上可以把任何对象转换成字符串：

```

>> nil.to_s
=> ""

```

结果显然是个空字符串，我们可以通过下面的方法串联（chain）验证这一点：

```

>> nil.empty?
NoMethodError: undefined method `empty?' for nil:NilClass
>> nil.to_s.empty?      # 消息串联
=> true

```

我们看到，`nil` 对象本身无法响应 `empty?` 方法，但是 `nil.to_s` 可以。

有一个特殊的方法可以测试对象是否为空，你或许能猜到这个方法：

```

>> "foo".nil?
=> false
>> "".nil?
=> false
>> nil.nil?
=> true

```

下面的代码

```
puts "x is not empty" if !x.empty?
```

演示了 `if` 关键字的另一种用法：你可以编写一个当且只当 `if` 后面的表达式为真值时才执行的语句。还有个对应的 `unless` 关键字也可以这么用：

```
>> string = "foobar"
>> puts "The string '#{string}' is nonempty." unless string.empty?
The string 'foobar' is nonempty.
=> nil
```

我们需要注意一下 `nil` 对象的特殊性，除了 `false` 本身之外，所有 Ruby 对象中它是唯一一个布尔值为“假”的。我们可以使用 `!!`（读作“bang bang”）对对象做两次取反操作，把对象转换成布尔值：

```
>> !!nil
=> false
```

除此之外，其他所有 Ruby 对象都是“真”值，数字 0 也是：

```
>> !!0
=> true
```

4.2.4. 定义方法

在控制台中，我们可以像定义 `home` 动作（[代码清单 3.6](#)）和 `full_title` 辅助方法（[代码清单 4.2](#)）一样定义方法。（在控制台中定义方法有点麻烦，我们一般会在文件中定义，这里只是为了演示。）例如，我们要定义一个名为 `string_message` 的方法，接受一个参数，返回值取决于参数是否为空：

```
>> def string_message(str = '')
>>   if str.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> :string_message
>> puts string_message("foobar")
The string is nonempty.
>> puts string_message("")
It's an empty string!
>> puts string_message
It's an empty string!
```

如最后一个命令所示，可以完全不指定参数（这种情况可以省略括号）。因为 `def string_message(str = '')` 中提供了参数的默认值，即空字符串。所以，`str` 参数是可选的，如果不指定，就使用默认值。

注意，Ruby 方法不用显式指定返回值，方法的返回值是最后一个语句的计算结果。上面这个函数的返回值是两个字符串中的一个，具体是哪一个取决于 `str` 参数是否为空。在 Ruby 方法中也可以显式指定返回值，下面这个方法和前面的等价：

```
>> def string_message(str = '')
>>   return "It's an empty string!" if str.empty?
>>   return "The string is nonempty."
>> end
```

（细心的读者可能会发现，其实没必要使用第二个 `return`，这一行是方法的最后一个表达式，不管有没有 `return`，字符串 `"The string is nonempty."` 都会作为返回值返回。不过两处都加上 `return` 看起来更好。）

还有一点很重要，方法并不关心参数的名字是什么。在前面定义的第一个方法中，可以把 `str` 换成任意有效的变量名，例如 `the_function_argument`，但是方法的作用不变：

```
>> def string_message(the_function_argument = '')
>>   if the_function_argument.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>> end
>> end
=> nil
>> puts string_message("")
It's an empty string!
>> puts string_message("foobar")
The string is nonempty.
```

4.2.5. 回顾标题的辅助方法

下面我们来理解一下[代码清单 4.2](#) 中的 `full_title` 辅助方法，⁶ 在其中加上注解之后如[代码清单 4.9](#) 所示：

代码清单 4.9：注解 `full_title` 方法

`app/helpers/application_helper.rb`

```
module ApplicationHelper

  # 根据所在的页面返回完整的标题          # 在文档中显示的注释
  def full_title(page_title = '')           # 定义方法，参数可选
    base_title = "Ruby on Rails Tutorial Sample App" # 变量赋值
    if page_title.empty?                      # 布尔测试
      base_title                             # 隐式返回值
    else
      "#{page_title} | #{base_title}"         # 字符串插值
    end
  end
end
```

方法定义、变量赋值、布尔测试、流程控制和字符串插值——组合在一起定义了一个可以在网站布局中使用的辅助方法。这里还有一个知识点——`module ApplicationHelper`：模块为我们提供了一种把相关方法组织在一起的方式，我们可以使用 `include` 把模块插入其他的类中。编写普通的 Ruby 程序时，你要自己定义一个模块，然后再显式将其引入类中，但是辅助方法所在的模块会由 Rails 为我们引入，结果是，`full_title` 方法[自动](#)可在所有视图中使用。

4.3. 其他数据类型

虽然 Web 应用最终都是处理字符串，但也需要其他的数据类型来生成字符串。本节介绍一些对开发 Rails 应用很重要的其他 Ruby 数据类型。

6. 其实还有一个地方我们不理解，那就是 Rails 是怎么把这些联系在一起的：把 URL 映射到动作上，让 `full_title` 辅助方法可以在视图中使用，等等。这是个很有意思的话题，我建议你以后好好了解一下。不过使用 Rails 并不需要完全了解 Rails 的运作机理。（若想深入了解 Rails，我推荐阅读 Obie Fernandez 写的《The Rails 4 Way》。）

4.3.1. 数组和值域

数组是一组具有特定顺序的元素。前面还没用过数组，不过理解数组对理解哈希有很大帮助（4.3.3 节），也有助于理解 Rails 中的数据模型（例如 2.3.3 节用到的 `has_many` 关联，11.1.3 节会做详细介绍）。

目前，我们已经花了很多时间理解字符串，从字符串过渡到数组可以从 `split` 方法开始：

```
>> "foo bar     baz".split      # 把字符串拆分成有三个元素的数组
=> ["foo", "bar", "baz"]
```

上述操作得到的结果是一个有三个字符串的数组。默认情况下，`split` 在空格处把字符串拆分成数组，不过也可以在几乎任何地方拆分：

```
>> "fooxbarxbazx".split('x')
=> ["foo", "bar", "baz"]
```

和大多数编程语言的习惯一样，Ruby 数组的索引也从零开始，因此数组中第一个元素的索引是 0，第二个元素的索引是 1，依此类推：

```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]          # Ruby 使用方括号获取数组元素
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]         # 索引还可以是负数
=> 17
```

我们看到，Ruby 使用方括号获取数组中的元素。除了方括号之外，Ruby 还为一些经常需要获取的元素提供了别名：⁷

```
>> a           # 只是为了看一下 a 的值是什么
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
=> 17
>> a.last == a[-1]    # 用 == 符号对比
=> true
```

最后一行用到了相等比较操作符 `==`，Ruby 和其他语言一样还提供了 `!=`（不等）等其他操作符：

```
>> x = a.length      # 和字符串一样，数组也可以响应 length 方法
=> 3
>> x == 3
=> true
```

7. 这段代码中使用的 `second` 方法不是 Ruby 定义的，而是 Rails 添加的。在这里可以使用这个方法是因为，Rails 控制台会自动加载 Rails 对 Ruby 的扩展。

```
>> x == 1
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
```

除了 `length` (上述代码的第一行) 之外, 数组还可以响应一系列其他方法:

```
>> a
=> [42, 8, 17]
>> a.empty?
=> false
>> a.include?(42)
=> true
>> a.sort
=> [8, 17, 42]
>> a.reverse
=> [17, 8, 42]
>> a.shuffle
=> [17, 42, 8]
>> a
=> [42, 8, 17]
```

注意, 上面的方法都没有修改 `a` 的值。如果想修改数组的值, 要使用相应的“炸弹”(bang) 方法 (之所以这么叫是因为, 这里的感叹号经常都读作“bang”) :

```
>> a
=> [42, 8, 17]
>> a.sort!
=> [8, 17, 42]
>> a
=> [8, 17, 42]
```

还可以使用 `push` 方法向数组中添加元素, 或者使用等价的 `<<` 操作符:

```
>> a.push(6)          # 把 6 加到数组结尾
=> [42, 8, 17, 6]
>> a << 7           # 把 7 加到数组结尾
=> [42, 8, 17, 6, 7]
>> a << "foo" << "bar"      # 串联操作
=> [42, 8, 17, 6, 7, "foo", "bar"]
```

最后一个命令说明, 可以把添加操作串在一起使用; 也说明, Ruby 不像很多其他语言, 数组中可以包含不同类型的数据 (本例中包含整数和字符串)。

前面用 `split` 把字符串拆分成字符串, 我们还可以使用 `join` 方法进行相反的操作:

```
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join                 # 没有连接符
=> "428177foobar"
```

```
>> a.join(',')          # 连接符是一个逗号和空格
=> "42, 8, 17, 7, foo, bar"
```

和数组有点类似的是值域 (range) , 使用 `to_a` 方法把它转换成数组或许更好理解:

```
>> 0..9
=> 0..9
>> 0..9.to_a           # 错了, to_a 在 9 上调用了
NoMethodError: undefined method `to_a' for 9:Fixnum
>> (0..9).to_a         # 调用 to_a 要用括号包住值域
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

虽然 `0..9` 是有效的值域, 不过上面第二个表达式告诉我们, 调用方法时要加上括号。

值域经常用来获取数组中的一组元素:

```
>> a = %w[foo bar baz quux]      # %w 创建一个元素为字符串的数组
=> ["foo", "bar", "baz", "quux"]
>> a[0..2]
=> ["foo", "bar", "baz"]
```

有个特别有用的技巧: 值域的结束值使用 -1 时, 不用知道数组的长度就能从起始值开始一直获取到最后一个元素:

```
>> a = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..(a.length-1)]           # 显式使用数组的长度
=> [2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..-1]                     # 小技巧, 索引使用 -1
=> [2, 3, 4, 5, 6, 7, 8, 9]
```

值域也可以使用字母:

```
>> ('a'..'e').to_a
=> ["a", "b", "c", "d", "e"]
```

4.3.2. 块

数组和值域可以响应的方法中有很多都可以跟着一个块 (block) , 这是 Ruby 最强大也是最难理解的功能:

```
>> (1..5).each { |i| puts 2 * i }
2
4
6
8
10
=> 1..5
```

这段代码在值域 `(1..5)` 上调用 `each` 方法, 然后又把 `{ |i| puts 2 * i }` 这个块传给 `each` 方法。`|i|` 两边的竖线在 Ruby 中用来定义块变量。只有这个方法才知道如何处理后面跟着的块。本例中, 值域的 `each` 方法会处理后面的块, 块中有一个本地变量 `i`, `each` 会把值域中的各个值传进块中, 然后执行其中的代码。

花括号是表示块的一种方式, 除此之外还有另一种方式:

```

>> (1..5).each do |i|
?>   puts 2 * i
>> end
2
4
6
8
10
=> 1..5

```

块中的内容可以多于一行，而且经常多于一行。本书遵照一个常用的约定，当块只有一行简单的代码时使用花括号形式；当块是一行很长的代码，或者有多行时使用 `do..end` 形式：

```

>> (1..5).each do |number|
?>   puts 2 * number
>>   puts '-'
>> end
2
-
4
-
6
-
8
-
10
-
=> 1..5

```

上面的代码用 `number` 代替了 `i`，我想告诉你的是，变量名可以使用任何值。

除非你已经有了一些编程知识，否则对块的理解是没有捷径的。你要做的是多看，看多了就会习惯这种用法。⁸ 幸好人类擅长从实例中归纳出一般性。下面是一些例子，其中几个用到了 `map` 方法：

```

>> 3.times { puts "Betelgeuse!" }    # 3.times 后跟的块没有变量
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
=> 3
>> (1..5).map { |i| i**2 }           # ** 表示幂运算
=> [1, 4, 9, 16, 25]
>> %w[a b c]                      # 再说一下，%w 用来创建元素为字符串的数组
=> ["a", "b", "c"]
>> %w[a b c].map { |char| char.upcase }
=> ["A", "B", "C"]
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]

```

可以看出，`map` 方法返回的是在数组或值域中每个元素上执行块中代码后得到的结果。在最后两个命令中，`map` 后面的块在块变量上调用一个方法，这种操作经常使用简写形式：

⁸. 块是闭包（closure），知道这一点对资深编程人员可能会有些帮助。闭包是一种匿名函数，其中附带了一些数据。

```

>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
>> %w[A B C].map(&:downcase)
=> ["a", "b", "c"]

```

(简写形式看起来有点儿奇怪，其中用到了符号，[4.3.3 节](#)会介绍。) 这种写法比较有趣，一开始是由 Rails 扩展实现的，但人们太喜欢了，现在已经集成到 Ruby 核心代码中。

最后再看一个使用块的例子。我们看一下[代码清单 4.4](#)中的一个测试用例：

```

test "should get home" do
  get :home
  assert_response :success
  assert_select "title", "Ruby on Rails Tutorial Sample App"
end

```

现在不需要理解细节（其实我也不懂），从 `do` 关键字可以看出，测试的主体其实就是一个块。`test` 方法的参数是一个字符串（测试的描述）和一个块，运行测试组件时会执行块中的内容。

现在我们来分析一下我在[1.5.4 节](#)生成随机二级域名时使用的那行 Ruby 代码：

```
('a'..'z').to_a.shuffle[0..7].join
```

我们一步步分解：

```

>> ('a'..'z').to_a                                # 由全部英文字母组成的数组
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
    "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
>> ('a'..'z').to_a.shuffle                         # 打乱数组
=> ["c", "g", "l", "k", "h", "z", "s", "i", "n", "d", "y", "u", "t", "j", "q",
    "b", "r", "o", "f", "e", "w", "v", "m", "a", "x", "p"]
>> ('a'..'z').to_a.shuffle[0..7]                  # 取出前 8 个元素
=> ["f", "w", "i", "a", "h", "p", "c", "x"]
>> ('a'..'z').to_a.shuffle[0..7].join            # 把取出的元素合并成字符串
=> "mznpybjuj"

```

4.3.3. 哈希和符号

哈希（Hash）本质上就是数组，只不过它的索引不局限于只能使用数字。（实际上在一些语言中，特别是 Perl，因为这个原因把哈希叫做“关联数组”。）哈希的索引（或者叫“键”）几乎可以使用任何对象。例如，可以使用字符串当键：

```

>> user = {}                                     # {} 是一个空哈希
=> {}
>> user["first_name"] = "Michael"              # 键为 "first_name"，值为 "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"                  # 键为 "last_name"，值为 "Hartl"
=> "Hartl"
>> user["first_name"]                           # 获取元素的方式和数组类似
=> "Michael"
>> user                                         # 哈希的字面量形式
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}

```

哈希通过一对花括号中包含一些键值对的形式表示，如果只有一对花括号而没有键值对（{}）就是一个空哈希。注意，哈希中的花括号和块中的花括号不是一个概念。（是的，这可能会让你困惑。）哈希虽然和数组类似，但二者却有一个很重要的区别：哈希中的元素没有特定的顺序。⁹ 如果顺序很重要的话就要使用数组。

通过方括号的形式每次定义一个元素的方式不太敏捷，使用 \Rightarrow 分隔的键值对这种字面量形式定义哈希要简洁得多：

```
>> user = { "first_name" => "Michael", "last_name" => "Hartl" }
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

在上面的代码中我用到了一个 Ruby 句法约定，在左花括号后面和右花括号前面加入了一个空格，不过控制台会忽略这些空格。（不要问我为什么这些空格是约定俗成的，或许是某个 Ruby 编程大牛喜欢这种形式，然后约定就产生了。）

目前为止哈希的键都使用字符串，在 Rails 中用“符号”（Symbol）当键很常见。符号看起来有点儿像字符串，只不过没有包含在一对引号中，而是在前面加一个冒号。例如，`:name` 就是一个符号。你可以把符号看成没有约束的字符串：¹⁰

```
>> "name".split('')
=> ["n", "a", "m", "e"]
>> :name.split('')
NoMethodError: undefined method `split' for :name:Symbol
>> "foobar".reverse
=> "raboof"
>> :foobar.reverse
NoMethodError: undefined method `reverse' for :foobar:Symbol
```

符号是 Ruby 特有的数据类型，其他语言很少用到。初看起来感觉很奇怪，不过 Rails 经常用到，所以你很快就会习惯。符号和字符串不同，并不是所有字符都能在符号中使用：

```
>> :foo-bar
NameError: undefined local variable or method `bar' for main:Object
>> :2foo
SyntaxError
```

只要以字母开头，其后都使用单词中常用的字符就没事。

用符号当键，我们可以按照如下方式定义一个 `user` 哈希：

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> { :name=>"Michael Hartl", :email=>"michael@example.com" }
>> user[:name]           # 获取 :name 对应的值
=> "Michael Hartl"
>> user[:password]       # 获取未定义的键对应的值
=> nil
```

从上面的例子可以看出，哈希中没有定义的键对应的值是 `nil`。

9. 在 Ruby 1.9 及以后的版本中，其实会按照元素输入时的顺序保存哈希，不过依赖特定的顺序显然是不明智的。

10. 没有约束的好处是，符号很容易进行比较，字符串要按照字母一个一个的比较，而符号只需比较一次。这就使得符号成为哈希键的最佳选择。

因为符号当键的情况太普遍了，Ruby 1.9 干脆就为这种用法定义了一种新句法：

```
>> h1 = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h2 = { name: "Michael Hartl", email: "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h1 == h2
=> true
```

第二中句法把“`符号 =>`”变成了“`键的名字:`”形式：

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

这种形式更好地沿袭了其他语言（例如 JavaScript）中哈希的表示方式，在 Rails 社区中也越来越受欢迎。这两种方式现在都在使用，所以你要能识别它们。可是，新句法有点让人困惑，因为 `:name` 本身是一种数据类型（符号），但 `name:` 却没有意义。不过在哈希字面量中，`:name =>` 和 `name:` 作用一样。因此，`{ :name => "Michael Hartl" }` 和 `{ name: "Michael Hartl" }` 是等效的。如果要表示符号，只能使用 `:name`（冒号在前面）。

哈希中元素的值可以是任何对象，甚至是另一个哈希，如[代码清单 4.10](#) 所示。

代码清单 4.10：嵌套哈希

```
>> params = {}          # 定义一个名为 params (parameters 的简称) 的哈希
=> {}
>> params[:user] = { name: "Michael Hartl", email: "mhartl@example.com" }
=> {:name=>"Michael Hartl", :email=>"mhartl@example.com"}
>> params
=> {:user=>{:name=>"Michael Hartl", :email=>"mhartl@example.com"}}
>> params[:user][:email]
=> "mhartl@example.com"
```

Rails 大量使用这种哈希中有哈希的形式（或称为“嵌套哈希”），我们从[7.3 节](#)起会接触到。

与数组和值域一样，哈希也能响应 `each` 方法。例如，一个名为 `flash` 的哈希，它的键是两个判断条件，`:success` 和 `:danger`：

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

注意，数组的 `each` 方法后面的块只有一个变量，而哈希的 `each` 方法后面的块接受两个变量，分别表示键和对应的值。所以哈希的 `each` 方法每次遍历都会以一个键值对为单位进行。

这段代码用到了很有用的 `inspect` 方法，返回被调用对象的字符串字面量表现形式：

```
>> puts (1..5).to_a           # 把值域转换成数组
1
2
3
4
```

```

5
>> puts (1..5).to_a.inspect    # 输出数组的字面量形式
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"

```

顺便说一下，因为使用 `inspect` 打印对象的方式经常使用，为此还有一个专门的快捷方式，`p` 方法：¹¹

```

>> p :name          # 等价于 'puts :name.inspect'
:name

```

4.3.4. 重温引入 CSS 的代码

现在我们要重新认识一下[代码清单 4.1](#) 中在布局中引入层叠样式表的代码：

```

<%= stylesheet_link_tag 'application', media: 'all',
                        'data-turbolinks-track' => true %>

```

我们现在基本上可以理解这行代码了。在[4.1 节](#)简单提到过，Rails 定义了一个特殊的函数用来引入样式表，下面的代码

```

stylesheet_link_tag 'application', media: 'all',
                     'data-turbolinks-track' => true

```

就是对这个函数的调用。不过还有几个奇怪的地方。第一，括号哪去了？在 Ruby 中，括号是可以省略的，所以下面两种写法是等价的：

```

# 调用函数时可以省略括号
stylesheet_link_tag('application', media: 'all',
                     'data-turbolinks-track' => true)
stylesheet_link_tag 'application', media: 'all',
                     'data-turbolinks-track' => true

```

第二，`media` 部分显然是一个哈希，但是怎么没用花括号？调用函数时，如果哈希是最后一个参数，可以省略花括号。所以下面两种写法是等价的：

```

# 如果最后一个参数是哈希，可以省略花括号
stylesheet_link_tag 'application', { media: 'all',
                                      'data-turbolinks-track' => true }
stylesheet_link_tag 'application', media: 'all',
                      'data-turbolinks-track' => true

```

还有，为什么 `data-turbolinks-track` 这个键值对使用旧句法？如果使用新句法，写成

```
data-turbolinks-track: true
```

是无效的，因为其中有连字符。（[4.3.3 节](#)说过，符号中不能使用连字符。）所以只能使用旧句法，写成

11. 其实有些细微差别，`p` 返回打印的对象，而 `puts` 始终返回 `nil`。感谢读者 Katarzyna Siwek 指出这一点。

```
'data-turbolinks-track' => true
```

最后，为什么换了一行 Ruby 还能正确解析？

```
stylesheet_link_tag 'application', media: 'all',
'data-turbolinks-track' => true
```

因为在这种情况下，Ruby 不关心有没有换行。¹² 我之所以把代码写成两行，是要保证每行代码不超过 80 个字符。¹³

所以，下面这段代码

```
stylesheet_link_tag 'application', media: 'all',
'data-turbolinks-track' => true
```

调用了 `stylesheet_link_tag` 函数，并且传入两个参数：一个是字符串，指明样式表的路径；另一个是哈希，包含两个元素，第一个指明媒介类型，第二个启用 Rails 4.0 中添加的 `Turbolink` 功能。因为使用的是 `<%=`，函数的执行结果会通过 ERb 插入模板中。如果在浏览器中查看网页的源码，会看到引入样式表所用的 HTML，如 [代码清单 4.11](#) 所示。（你可能会在 CSS 的文件名后看到额外的字符，例如 `?body=1`。这是 Rails 加入的，确保修改 CSS 后浏览器会重新加载。）

代码清单 4.11：引入 CSS 的代码生成的 HTML

```
<link data-turbolinks-track="true" href="/assets/application.css" media="all"
rel="stylesheet" />
```

如果在浏览器中打开 <http://localhost:3000/assets/application.css> 查看 CSS 的话，会发现是这个文件是空的（但有一些注释）。[第 5 章](#)会介绍如何添加样式。

4.4. Ruby 类

我们之前说过，Ruby 中的一切都是对象。本节我们要自己定义一些对象。Ruby 和其他面向对象的语言一样，使用类来组织方法，然后实例化类，创建对象。如果你刚接触“面向对象编程”（Object-Oriented Programming，简称 OOP），这些听起来都似天书一般，那我们来看一些实例吧。

4.4.1. 构造方法

我们看过很多使用类初始化对象的例子，不过还没自己动手做过。例如，我们使用双引号初始化一个字符串，双引号是字符串的字面构造方法：

```
>> s = "foobar"      # 使用双引号字面构造方法
=> "foobar"
>> s.class
=> String
```

我们看到，字符串可以响应 `class` 方法，返回值是字符串所属的类。

12. 换行符在一行的结尾处，开始新的一行。在代码中，换行符用 `\n` 表示。

13. 数列数会让你发疯的，所以很多文本编辑器都提供了一个视觉标识。例如，如果再看一下[图 1.5](#)的话，会发现右边有一条细线，它可以帮助你把一行代码控制在 80 个字符以内。云端 IDE 默认会显示这条竖线。如果使用 TextMate，可以在如下菜单中找到这个功能：`View > Wrap Column > 78`。在 Sublime Text 中是：`View > Ruler > 78`，或：`View > Ruler > 80`。

除了使用字面构造方法之外，我们还可以使用等价的“具名构造方法”（named constructor），即在类名上调用 `new` 方法：¹⁴

```
>> s = String.new("foobar")    # 字符串的具名构造方法
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

这段代码中使用的具名构造方法和字面构造方法是等价的，只是更能表现我们的意图。

数组和字符串类似：

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

不过哈希就有点不同了。数组的构造方法 `Array.new` 可接受一个可选的参数指明数组的初始值，`Hash.new` 可接受一个参数指明元素的默认值，就是当键不存在时返回的值：

```
>> h = Hash.new
=> {}
>> h[:foo]          # 试图获取不存在的键 :foo 对应的值
=> nil
>> h = Hash.new(0)   # 让不存在的键返回 0 而不是 nil
=> {}
>> h[:foo]
=> 0
```

在类上调用的方法，如本例的 `new`，叫“类方法”（class method）。在类上调用 `new` 方法，得到的结果是这个类的一个对象，也叫做这个类的“实例”（instance）。在实例上调用的方法，例如 `length`，叫“实例方法”（instance method）。

4.4.2. 类的继承

学习类时，理清类的继承关系会很有用，我们可以使用 `superclass` 方法：

```
>> s = String.new("foobar")
=> "foobar"
>> s.class           # 查找 s 所属的类
=> String
>> s.class.superclass      # 查找 String 的父类
=> Object
>> s.class.superclass.superclass  # Ruby 1.9 使用 BasicObject 作为基类
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

这个继承关系如图 4.1 所示。可以看到，`String` 的父类是 `Object`，`Object` 的父类是 `BasicObject`，但是 `BasicObject` 就没有父类了。这样的关系对每个 Ruby 对象都适用：只要在类的继承关系上往上多走几层，就会发

14. 返回值可能由于 Ruby 版本的不同而有所不同。这个例子假设你使用的是 Ruby 1.9.3 或以上版本。

现 Ruby 中的每个类最终都继承自 `BasicObject`, 而它本身没有父类。这就是“Ruby 中一切皆对象”技术层面上的意义。

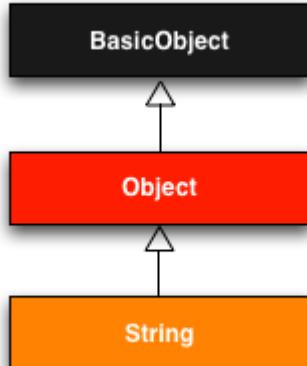


图 4.1: `String` 类的继承关系

要想更深入地理解类，最好的方法是自己动手编写一个类。我们来定义一个名为 `Word` 的类，其中有一个名为 `palindrome?` 方法，如果单词顺读和反读都一样就返回 `true`:

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

我们可以按照下面的方式使用这个类:

```
>> w = Word.new          # 创建一个 Word 对象
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

如果你觉得这个例子有点大题小做，很好，我的目的达到了。定义一个新类，可是只创建一个接受一个字符串作为参数的方法，这么做很古怪。既然单词是字符串，让 `Word` 继承 `String` 不就行了，如[代码清单 4.12](#) 所示。（你要退出控制台，然后再在控制台中输入这写代码，这样才能把之前的 `Word` 定义清除掉。）

代码清单 4.12：在控制台中定义 `Word` 类

```
>> class Word < String          # Word 继承自 String
>>   # 如果字符串和反转后相等就返回 true
>>   def palindrome?
>>     self == self.reverse      # self 代表这个字符串本身
>>   end
>> end
=> nil
```

其中，`Word < String` 在 Ruby 中表示继承（3.2 节简介过），这样除了定义 `palindrome?` 方法之外，`Word` 还拥所有字符串拥有的方法：

```
>> s = Word.new("level")      # 创建一个 Word 实例，初始值为 "level"
=> "level"
>> s.palindrome?            # Word 实例可以响应 palindrome? 方法
=> true
>> s.length                 # Word 实例还继承了普通字符串的所有方法
=> 5
```

`Word` 继承自 `String`，我们可以在控制台中查看类的继承关系：

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

这个继承关系如图 4.2 所示。

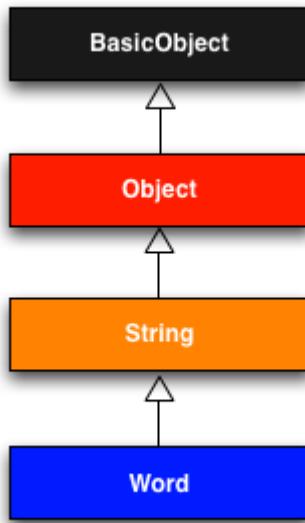


图 4.2：代码清单 4.12 中定义的 `Word` 类（非内置类）的继承关系

注意，在代码清单 4.12 中检查单词和单词的反转是否相同时，要在 `Word` 类中引用这个单词。在 Ruby 中使用 `self` 关键字¹⁵ 引用：在 `Word` 类中，`self` 代表的就是对象本身。所以我们可以使用

```
self == self.reverse
```

来检查单词是否为“回文”。其实，在类中调用方法或访问属性时可以不用 `self.`（赋值时不行），所以也可以写成 `self == reverse`。

15. 关于 Ruby 类和 `self` 关键字，请阅读 RailsTips 中的《Class and Instance Variables in Ruby》一文。

4.4.3. 修改内置的类

虽然继承是个很强大的功能，不过在判断回文这个例子中，如果能把 `palindrome?` 加入 `String` 类就更好了，这样（除了其他方法外）我们可以在字符串字面上调用 `palindrome?` 方法。现在我们还不能直接调用：

```
>> "level".palindrome?  
NoMethodError: undefined method `palindrome?' for "level":String
```

有点令人惊讶的是，Ruby 允许你这么做，Ruby 中的类可以被打开进行修改，允许像我们这样的普通人添加一些方法：

```
>> class String  
>>   # 如果字符串和反转后相等就返回 true  
>>   def palindrome?  
>>     self == self.reverse  
>>   end  
>> end  
=> nil  
>> "deified".palindrome?  
=> true
```

（我不知道哪一个更牛：Ruby 允许向内置的类中添加方法，或 "`deified`" 是个回文。）

修改内置的类是个很强大的功能，不过功能强大意味着责任也大，如果没有很好的理由，向内置的类中添加方法是不好的习惯。Rails 自然有很好的理由。例如，在 Web 应用中我们经常要避免变量的值是空白

（`blank`）的，像用户名之类的就不应该是空格或空白，所以 Rails 为 Ruby 添加了一个 `blank?` 方法。Rails 控制台会自动加载 Rails 添加的功能，下面看几个例子（在 `irb` 中不可以）：

```
>> "".blank?  
=> true  
>> " ".empty?  
=> false  
>> " ".blank?  
=> true  
>> nil.blank?  
=> true
```

可以看出，一个包含空格的字符串不是空的（`empty`），却是空白的（`blank`）。还要注意，`nil` 也是空白的。因为 `nil` 不是字符串，所以上面的代码说明了 Rails 其实是把 `blank?` 添加到 `String` 的基类 `Object` 中的。[8.4 节](#)会再介绍一些 Rails 扩展 Ruby 类的例子。）

4.4.4. 控制器类

讨论类和继承时你可能觉得似曾相识，不错，我们之前见过，在静态页面控制器中（[代码清单 3.18](#)）：

```
class StaticPagesController < ApplicationController  
  
  def home  
  end  
  
  def help  
  end
```

```
def about
end
end
```

你现在可以理解，至少有点能理解，这些代码的意思了：`StaticPagesController` 是一个类，继承自 `ApplicationController`，其中有三个方法，分别是 `home`、`help` 和 `about`。因为 Rails 控制台会加载本地的 Rails 环境，所以我们在控制台中创建一个控制器，查看一下它的继承关系：¹⁶

```
>> controller = StaticPagesController.new
=> #<StaticPagesController:0x22855d0>
>> controller.class
=> StaticPagesController
>> controller.class.superclass
=> ApplicationController
>> controller.class.superclass.superclass
=> ActionController::Base
>> controller.class.superclass.superclass.superclass
=> ActionController::Metal
>> controller.class.superclass.superclass.superclass.superclass
=> AbstractController::Base
>> controller.class.superclass.superclass.superclass.superclass.superclass
=> Object
```

这个继承关系如图 4.3 所示。

我们还可以在控制台中调用控制器的动作，动作其实就是方法：

```
>> controller.home
=> nil
```

`home` 动作的返回值为 `nil`，因为它是空的。

注意，动作没有返回值，或至少没返回真正需要的值。如我们在第 3 章看到的，`home` 动作的目的是渲染网页，而不是返回一个值。但是，我记得没在任何地方调用过 `StaticPagesController.new`，到底怎么回事呢？

原因在于，Rails 是用 Ruby 编写的，但 Rails 不是 Ruby。有些 Rails 类就像普通的 Ruby 类一样，不过也有些则得益于 Rails 的强大功能。Rails 是单独的一门学问，应该和 Ruby 分开学习和理解。

16. 你没必要知道继承关系中每个类的作用。我也不知道它们都是干什么的，而我从 2005 年就开始使用 Ruby on Rails 了。这可能意味着以下两个问题中的一个：第一，我是个废柴；第二，不需要知道所有内部知识也能成为熟练的 Rails 开发者。我们当然都希望是第二点。

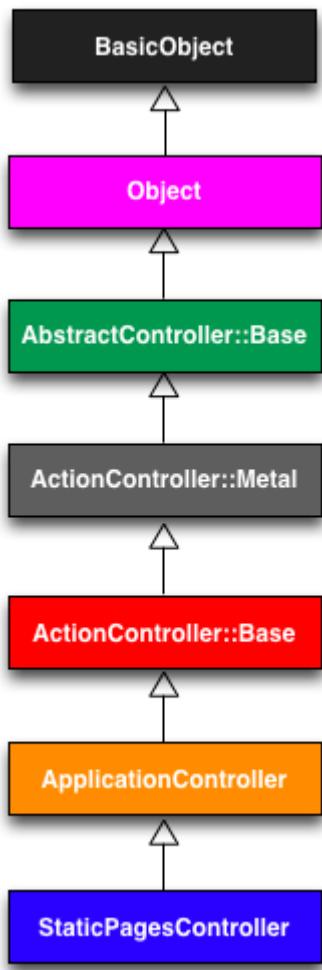


图 4.3：静态页面控制器的类继承关系

4.4.5. 用户类

我们要自己定义一个类，结束对 Ruby 的介绍。这个类名为 `User`，目的是实现 [第 6 章](#)用到的用户模型。

到目前为止，我们都在控制台中定义类，这样很快捷，但也有点不爽。现在我们要在应用的根目录中创建一个名为 `example_user.rb` 的文件，然后写入 [代码清单 4.13](#) 中的内容。

代码清单 4.13：定义 User 类

`example_user.rb`

```

class User
  attr_accessor :name, :email

  def initialize(attributes = {})
    @name  = attributes[:name]
    @email = attributes[:email]
  end

  def formatted_email

```

```
"#{@name} <#{@email}>"  
end  
end
```

这段代码有很多地方要说明，我们一步步来。先看下面这行：

```
attr_accessor :name, :email
```

这行代码为用户的名字和电子邮件地址创建“属性访问器”（attribute accessors），也就是定义了“获取方法”（getter）和“设定方法”（setter），用来取回和赋值 `@name` 和 `@email` 实例变量（[2.2.2 节](#) 和 [3.6 节](#) 简介过）。在 Rails 中，实例变量的意义在于，它们自动在视图中可用。而通常实例变量的作用是在 Ruby 类中不同的方法之间传递值。（稍后会更详细地介绍这一点。）实例变量总是以 `@` 符号开头，如果未定义，值为 `nil`。

第一个方法，`initialize`，在 Ruby 中有特殊的意义：执行 `User.new` 时会调用这个方法。这个 `initialize` 方法接受一个参数，`attributes`：

```
def initialize(attributes = {})  
  @name = attributes[:name]  
  @email = attributes[:email]  
end
```

`attributes` 参数的默认值是一个空哈希，所以我们可以定义一个没有名字或没有电子邮件地址的用户。（回想一下 [4.3.3 节](#) 的内容，如果键不存在就返回 `nil`，所以如果没定义 `:name` 键，`attributes[:name]` 会返回 `nil`，`attributes[:email]` 也是一样。）

最后，类中定义了一个名为 `formatted_email` 的方法，使用被赋了值的 `@name` 和 `@email` 变量进行插值，组成一个格式良好的用户电子邮件地址：

```
def formatted_email  
  "#{@name} <#{@email}>"  
end
```

因为 `@name` 和 `@email` 都是实例变量（如 `@` 符号所示），所以在 `formatted_email` 方法中自动可用。

我们打开控制台，加载（`require`）这个文件，实际使用一下这个类：

```
>> require './example_user'      # 加载 example_user 文件中代码的方式  
=> true  
>> example = User.new  
=> #<User:0x224ceec @email=nil, @name=nil>  
>> example.name                # 返回 nil，因为 attributes[:name] 是 nil  
=> nil  
>> example.name = "Example User"        # 赋值一个非 nil 的名字  
=> "Example User"  
>> example.email = "user@example.com"    # 赋值一个非 nil 的电子邮件地址  
=> "user@example.com"  
>> example.formatted_email  
=> "Example User <user@example.com>"
```

这段代码中的点号 `.`，在 Unix 中指“当前目录”，`'./example_user'` 告诉 Ruby 在当前目录中寻找这个文件。接下来的代码创建了一个空用户，然后通过直接赋值给相应的属性来提供他的名字和电子邮件地址（因为有 `attr_accessor` 所以才能赋值）。我们输入 `example.name = "Example User"` 时，Ruby 会把 `@name` 变量的值设为 `"Example User"` (`email` 属性类似)，然后就可以在 `formatted_email` 中使用。

4.3.4 节介绍过，如果最后一个参数是哈希，可以省略花括号。我们可以把一个预先定义好的哈希传给 `initialize` 方法，再创建一个用户：

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User:0x225167c @email="mhartl@example.com", @name="Michael Hartl">
>> user.formatted_email
=> "Michael Hartl <mhartl@example.com>"
```

从第 7 章开始，我们会使用哈希初始化对象，这种技术叫做“批量赋值”（mass assignment），在 Rails 中很常见。

4.5. 小结

现在对 Ruby 语言的介绍结束了。第 5 章会好好利用这些知识来开发演示应用。

我们不会使用 4.4.5 节创建的 `example_user.rb` 文件，所以我建议把它删除：

```
$ rm example_user.rb
```

然后把其他的改动提交到代码仓库中，再推送到 Bitbucket，然后部署到 Heroku：

```
$ git status
$ git commit -am "Add a full_title helper"
$ git push
$ bundle exec rake test
$ git push heroku
```

4.5.1. 读完本章学到了什么

- Ruby 提供了很多处理字符串的方法；
- 在 Ruby 中一切皆对象；
- 在 Ruby 中定义方法使用 `def` 关键字；
- 在 Ruby 中定义类使用 `class` 关键字；
- Ruby 内建支持的数据类型有数组、值域和哈希；
- Ruby 块是一种灵活的语言接口，可以遍历可枚举的数据类型；
- 符号是一种标记，和字符串类似，但没有额外的束缚；
- Ruby 支持对象继承；
- 可以打开并修改 Ruby 内建的类；
- “deified”是回文；

4.6. 练习

1. 把[代码清单 4.14](#) 中的问号换成合适的方法，结合 `split`、`shuffle` 和 `join` 实现一个函数，把字符串中的字符顺序打乱。
2. 参照[代码清单 4.15](#)，把 `shuffle` 方法添加到 `String` 类中。

3. 创建三个哈希，分别命名为 `person1`、`person2` 和 `person3`，把名和姓赋值给 `:first` 和 `:last` 键。然后创建一个名为 `params` 的哈希，让 `params[:father]` 对应 `person1`，`params[:mother]` 对应 `person2`，`params[:child]` 对应 `person3`。验证一下 `params[:father][:first]` 的值是否正确。
4. 找一个在线版 Ruby API 文档，了解哈希的 `merge` 方法的用法。下面这个表达式的计算结果是什么？

```
{ "a" => 100, "b" => 200 }.merge({ "b" => 300 })
```

代码清单 4.14：打乱字符串函数的骨架

```
>> def string_shuffle(s)
>>   s.?('').?..
>> end
>> string_shuffle("foobar")
=> "oobfra"
```

代码清单 4.15：添加到 `String` 类中的 `shuffle` 方法骨架

```
>> class String
>>   def shuffle
>>     self.?('').?..
>>   end
>> end
>> "foobar".shuffle
=> "boraf'o"
```