原书第2版 · 涵盖 Rails 4

Ruby on Rails 教程

Ruby on Rails Tutorial 中文版

Michael Hartl 著 • Andor Chen 译

http://railstutorial-china.org

目录

作者译者 ..... **iv**
致中国读者 ..... **v**
序 ..... **vi**
第 1 章 从零到部署 **1**
1.1 简介 2
1.2 搭建环境 6
1.3 用 git 做版本控制 20
1.4 部署 30
1.5 小结 35
第 2 章 演示程序 **37**
2.1 规划程序 37
2.2 Users 资源（users resource） 40
2.3 Microposts 资源 52
2.4 小结 61
第 3 章 基本静态的页面 **62**
3.1 静态页面 66
3.2 第一个测试 72
3.3 有点动态内容的页面 81
3.4 小结 90
3.5 练习 91
3.6 高级技术 93
第 4 章 Rails 背后的 Ruby **105**
4.1 导言 105
4.2 字符串和方法 109
4.3 其他的数据类型 115
4.4 Ruby 类 125
4.5 小结 133
4.6 练习 133
第 5 章 完善布局 **135**
5.1 添加一些结构 136
5.2 Sass 和 asset pipeline 151
5.3 布局中的链接 160
5.4 用户注册：第一步 173
5.5 小结 177
5.6 练习 177
第 6 章 用户模型 **181**
6.1 User 模型 182

6.2 用户数据验证 191
6.3 加上安全密码 205
6.4 小结 216
6.5 练习 216
第 7 章 用户注册 219
7.1 显示用户信息 219
7.2 注册表单 235
7.3 注册失败 247
7.4 注册成功 255
7.5 小结 262
7.6 练习 262
第 8 章 登录和退出 265
8.1 session 和登录失败 265
8.2 登录成功 280
8.3 Cucumber 简介（选读） 299
8.4 小结 306
8.5 练习 307
第 9 章 更新、显示和删除用户 309
9.1 更新用户 309
9.2 权限限制 319
9.3 列出所有用户 330
9.4 删除用户 345
9.5 小结 353
9.6 练习 355
第 10 章 用户的微博 359
10.1 Microposts 模型 359
10.2 显示微博 372
10.3 微博相关的操作 381
10.4 小结 403
10.5 练习 403
第 11 章 关注用户 407
11.1 关系模型 407
11.2 关注用户功能的网页界面 423
11.3 动态列表 447
11.4 小结 454
11.5 练习 457

作者译者

本书的英文版原作者是 Michael Hartl，把 Ruby on Rails Web 开发介绍给世人的先行者。他之前曾经写作并开发了 RailsSoace，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。

本书的中文版译者是 Andor Chen。他毕业于中南大学，现在一家国内领先的锂离子电池制造企业任职机械工程师。用过 WordPress 的读者或许会熟悉这个名字。现在他是个 Ruby 初学者，并在学习使用 Rails 做 Web 开发。本书就是他在学习 Rails 的过程中翻译的。

致中国读者

Ruby 是一门很美的计算机语言，其设计原则就是“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，也从未如此简单。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 应用程序开发都在迅猛发展，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更有效果。正因为这样，当看到 Andor Chen 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造访。希望我到中国时能见到本书的一些读者！

衷心的祝福你们，

《Ruby on Rails Tutorial》作者 Michael Hartl

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）是大张旗鼓的转用 Ruby on Rails 最早的企业之一，然后又更加惹眼的换回了 PHP（可以在 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，我又迁移到 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，感觉却是自然无比。本书也是唯一一本自始至终都是用测试驱动开发理念的 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被如此清楚的介绍过。书中的示例还用到了 Git、GitHub 和 Heroku，作者真的是让你体验了一把开发真正能用的程序是什么感觉，而且书中用到的代码并不是凭空捏造的。

线性叙述是很好的模式。我花了三整天的时间阅读了本书，做了书中所有的示例程序和练习。从头至尾，循序渐进，不要跳着读，这样才能这种从中收益。

享受 Rails 的盛宴吧！

Derek Sivers (sivers.org)
CD Baby 创始人

第 1 章 从零到部署

欢迎学习《Ruby on Rails 教程》。本书的目标是成为对“如果想学习使用 Ruby on Rails 进行 Web 开发，我应该从哪儿开始？”这一问题的最好答案。学习完本书的内容之后，你将具备使用 Rails 进行开发和部署 Web 程序的技能。同时你还能够通过一些进阶的书籍、博客和视频教程等活跃的 Rails 教学体系继续深造。本书基于 Rails 4，这里的知识代表着 Web 开发的发展方向。（《Ruby on Rails 教程》的最新版本可以从本书的网站上获取。）

（你现在看到的是针对 Rails 4 的版本，所有内容都和 Rails 4 兼容，但这并不是一个新版本，因为 Rails 4 的变化还没大到新出一版。作为一本入门书，本书和之前针对 Rails 3.2 的版本差别很小。小变化虽多（详见旁注 1.1），但主要的变化是一个称为“健壮参数”的新安全技术，会在 7.3.2 节中介绍。只要 Rails 的变化足大，我就会新出一版，介绍 Turbolinks 和俄罗斯套娃缓存等技术，还会介绍 RSpec 的新用法，例如“功能测试”。）

旁注 1.1：与第二版的差异

下面的列表列出了本书和第二版之间几乎所有的差异。（最重要的一个差异是，使用了“健壮参数”，其他都是小变动。）我列出这个列表是为了方便第二版的读者（或者熟悉 Rails 3.2 的读者）快速查看两版的不同之处。如果你没用过 Rails 3.2 请忽略这个列表。

列表的每一项后面都有一个链接，指向变动所在的章节或代码。

- 从 Rails 3.2 升级到 Rails 4.0（1.2.2 节）
- 显式引入 Capybara DSL（代码 3.10）
- 把 RSpec 的 `.should` 句法换成了 `expect().to`（3.2.1 节）
- 把 `have_selector('title', ...)` 换成了 `have_title(...)`（3.3.1 节）
- 更新操作所用的 HTTP 请求从 `PUT` 换成了 `PATCH`（旁注 3.3）
- 引入样式表和 JavaScript 脚本时指定了针对 Turbolinks 的参数（代码 3.26）
- 把 `root to: 'path'` 换成了 `root 'path'`（代码 5.24）
- 把 `find_by_thing(...)` 换成了 `find_by(thing: ...)`（6.1.4 节）
- 把 `rake db:test:prepare` 命令换成了 `rake test:prepare`（6.2.1 节）
- 撤掉 `attr_accessible`，换用“健壮参数”（7.3.2 节）
- 换用加密的记忆权标（8.2.1 节）
- 把 `before_filter` 换成了 `before_action`（代码 9.21）
- 使用 Capybara 的 `match: :first` 参数点击第一个匹配的链接（代码 9.42）
- 把 `default_scope` 的参数从 Hash 换成了 lambda（代码 10.11）
- 把 `dup` 换成了 `to_a` 方法（代码 10.12）
- 使用 XPath 测试按钮的状态变化（11.2.4 节）

注意，本书的目标并不仅仅是教你 Rails，而是教你怎样使用 Rails 进行 Web 开发，教会你为因特网开发软件的技能。除了讲到 Ruby on Rails 之外，涉及到的技术还有 HTML、CSS、数据库、版本控制、测试和部署。为了达成学

习目标，本书使用了一个完整的方案：通过实例学习使用 Rails 从零开始创建一个真正的程序。如 Derek Sivers 在前言中所说的，本书内容采用线性TDD结构，需要从头开始按顺序读到结尾。如果你经常跳着阅读技术类书籍，这种线性的组织方式需要你适应一下。你可以将本书设想为一个电子游戏，学习完每一章就会升一级。（而练习就是每一关的小怪兽。）

本章首先将介绍如何安装 Ruby on Rails 所需的软件，搭建开发所需的环境（1.2 节）。然后创建第一个 Rails 程序 `first_app`。本书会遵从一些优秀的软件开发习惯，所以在创建第一个程序后我们会立即将它放到版本控制系统 Git 中（1.3 节）。最后，我们还将把这个程序放到实际的生产环境中运行（1.4 节）。

第 2 章我们会创建第二个程序，演示一些 Rails 程序的基本操作。为了快速创建，我们会使用脚手架功能（旁注 1.1）来创建一个示例程序（名为 `demo_app`），因为生成的代码还很粗糙也很复杂，第 2 章将集中精力在通过浏览器的 URI （经常称为 URL）[1]来和程序交互这一点上。

本书剩下的章节将介绍从零开始开发一个大型示例程序（名为 `sample_app`）。在这个程序的开发过程中将使用“测试驱动开发”（Test-driven Development, TDD）理念，从第 3 章开始创建静态页面，然后增加一些动态的内容。第四章则会简要的介绍一下 Rails 背后的 Ruby 程序语言。第五章到第九章将逐步完善这个程序的基础框架：网站的布局，用户数据模型，完整的注册和验证系统。最后在第 10 章和第 11 章将添加微博和社交功能，最终开发出一个可以实际运行的示例网站。

最终的示例程序将在外表上和一个同样采用 Rails 开发的微博网站十分相似[2]。虽然我们将主要的精力集中在这个示例程序上了，但是本书的重点却在于提供一些通用的方法，这样你就会具有坚实的基本功，不论开发什么样的 Web 程序都能够派上用场。

1.1 简介

自 2004 年出现以来，Rails 迅速成为动态 Web 程序开发领域功能最强大、最受欢迎的框架之一。从初创的项目到很多的大公司都在使用 Rails：37signals，Github，Shopify，Scribd，Twitter，Disney，Hulu 和 Yellow Pages 等，这个列表还很长。有很多 Web 开发工作室也在使用 Rails，比如 ENTP，thoughtbot，Pivotal Labs 和 Hashrocket，以及无数的独立顾问，培训人员和项目承包商。

是什么使得 Rails 如此成功呢？首先，Ruby on Rails 是完全开源的，基于 MIT 协议发布，可以免费下载、使用。Rails 的成功有很大一部分是得益于它优雅而紧凑的设计。研习 Ruby 语言的高可扩展性后，Rails 开发了一套用于开发 Web 程序的“领域专属语言”（Domain-speific Language, DSL）。所以 Web 编程中像生成 HTML、创建数据模型、URL 路由等任务在 Rails 中都很容易实现，最终得到的程序代码很简洁而且可读性较高。

Rails 还会快速跟进 Web 领域最新的技术和框架架构技术。例如，Rails 是最早实现 REST 这个 Web 程序架构体系的框架之一（这一体系将贯穿本书）。当其他的框架开发出成功的新技术后，Rails 的创建者 DHH 及其核心开发团队会毫不犹豫的将其吸纳进来。或许最典型的例子就是 Rails 和 Merb （和 Rails 类似的 Ruby Web 框架）两个项目的合并，这样一来 Rails 就继承了 Merb 的模块化设计、稳定的 API，性能也得到了提升。

1. URI 是统一资源标识符（Uniform Resources 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。
2. 译者注：指 Twiiter

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，组织了很多会议，开发了大量的 gem，还有很多内容丰富的博客，一些讨论组和 IRC 频道。有如此多数量的 Rails 程序员也使得处理程序错误变得简单了："使用 Google 搜索错误信息"的方法几乎总能搜到一篇相关的博客文章或讨论组的话题。

1.1.1 给不同读者群的建议

本书的内容不仅只是讲解 Rails，还会涉及 Ruby 语言、RSpec 测试框架、HTML、CSS、少量的 JavaScript 和一些 SQL。所以不管你的 Web 开发技能处在什么层次，读完本书后你就能够继续学习一些较为高级的 Rails 资源了，同时也会对书中提到的其他技术有一个大体的认识。这么说也意味着本书要覆盖很多知识，如果你不是一个有些经验的程序员学起来会觉得有些吃力。下面就根据不同的开发背景给出使用本书的一些建议。

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 DHH （David Heinemeier Hansson）制作的著名的"15分钟博客程序"视频，该视频以及其衍生版本是窥探 Rails 强大功能一种很好的方式，我推荐你也看一下这些视频。剧透一下：这些视频中的演示能控制在15分钟得益于一个叫做"脚手架（scaffold）"的功能，它通过 Rails 命令 `generate` 生产大量的代码。

很多人制作 Rails 教程时选择使用脚手架功能，因为它更快、更简单、更诱人。不过脚手架会生成大量复杂的代码，会使初学者产生困惑，虽然会用了但却不明白到底发生了什么事。使用脚手架功能可能会把你变成一个脚本生成器的使用者但却不会增进你对 Rails 知识的掌握。

本书将采用一种不同的方式，虽然第二章会用脚手架开发一个小型的示例程序，但本书的核心是从第三章开始开发的较为大型的程序。在开发这个大型程序的每一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。这样的过程最终会让你对 Rails 知识有较为深刻地理解，能灵活运用，创建几乎任何类型的 Web 程序。

所有读者：学习 Rails 时一个常见的疑问是，是否要先学习 Ruby。这个问题的答案取决于你个人的学习方式以及你所具有的编程经验。如果你希望较为系统的彻底学习，或者你以前从未编程过，那么先学 Ruby 或许更适合你，我推荐你阅读 Peter Cooper 的《Ruby 入门》一书。很多 Rails 开发初学者很想立马就开始 Web 程序开发，而不是在此之前阅读一本 500 多页纯粹讲解 Ruby 的书。如果你是这类人群，我推荐你在 Try Ruby 上学习一些短小的交互式教程，然后还可以看一下 Rails for Zombies 这个免费的视频教程，看看 Rails 都能做些什么。

另外一个常见的疑问是，是否要在一开始就使用测试。就如前面的介绍所说的，本书会使用"测试驱动开发（也叫"先测试后开发"）"理念，我认为这是使用 Rails 进行 Web 开发最好的方式，但这样也会增加难度和复杂度。如果你觉得做测试有些困难，我建议你在第一遍阅读时直接跳过所有测试，或者（更好的是）只把它们当做验证代码正确性的工具，而不用管测试的机理。如果采用后一种方法，你要创建一些必要的测试文件（叫做 spec），然后将本书中提供的测试代码编写进去，然后运行这些测试用例（第五章会介绍）得到失败消息，然后编写代码再运行测试让其通过。

缺乏经验的程序员：本书的主要读者群不是刚入门的程序员，Web 程序及其相关的任意一个技术都是很复杂的。如果你完全是个 Web 编程菜鸟，发现本书的内容太难了，我建议你先学习基本的 HTML 和 CSS（很可惜这两种技术我没有推荐的书籍，但是《深入浅出 HTML》应该不错，一个读者推荐 David Sawyer McFarland 的《CSS 实战手

册》），然后再试着阅读本书。你也可以考虑先阅读 Peter Cooper 的《Ruby 入门》的前几章，这几章中的示例程序都比功能完善的 Web 程序小得多。不过也有一批初学者通过本书学会了 Web 开发，所以你不妨也试一下，而且我强烈推荐本书配套的教学视频，通过观看别人的操作来学习 Rails 开发。

经验丰富的程序员，但是刚接触 Web 开发：以前的经验说明你可能已经理解了类、方法、数据结构等概念，这是个好的开始。不过，如果你以前是 C/C++ 或 Java 程序员，你会觉得 Ruby 有点另类，需要花一段时间才能适应；慢慢的适应，你会习惯的。（如果你实在无法放弃使用行尾的分号，Ruby 允许你这么做）本书会为你介绍所有 Web 相关的概念，所以如果你现在并不知道 `POST` 和 `PATCH` 的区别也不要紧。

经验丰富的 Web 开发者，但是刚接触 Rails：你有很好的基础了，如果你曾经使用过 PHP 或 Python（更好）这些动态语言就更好了。我们要讲的基础都是一致的，但是你可能对 TDD 还有 Rails 采用的 REST 架构感到陌生。而且 Ruby 语言有自己的风格，这一点对你来说也是陌生的。

经验丰富的 Ruby 程序员：如今 Ruby 程序员不懂 Rails 的很少，如果你是这种情况，可以快速的过一遍本书，然后自己动手开发一个应用程序。

缺乏经验的 Rails 程序员：你或许阅读过其他的 Rails 教程，也开发过小型的 Rails 程序。根据一些读者的反馈，本书还是会给你带来帮助，别的不说，单就时效性而言，本书会比你当初学习 Rails 使用的教程要更新一些。

经验丰富的 Rails 程序员：你不需要阅读本书了，但是很多经验丰富的 Rails 开发者还是说他们从本书中学到了很多，或许通过本书你会换个角度来看 Rails。

读完本书后，我建议经验丰富的程序员继续阅读 David A. Black 的《The Well-Grounded Rubyist》，Russ Olsen 的《Eloquent Ruby》，或者 Hal Fulton 的《Ruby 之道》，这本也是进阶书籍，不过更为专注某些特定的话题。

不管你是从哪里开始的，学完本书你还应该继续学习一些中高级 Rails 资源。以下是我推荐的学习资源[3]：

- RailsCasts，Ryan Bates：优秀的免费（大多数）视频教程
- PeepCode：优秀的收费视频教程
- Code School：交互式的编程课程
- Rails 官方指南：按话题编写经常更新的 Rails 参考
- Ryan Bates 的 RailsCasts：我是不是已经说过 RailsCasts 了？真的，强烈推荐 RailsCasts。

1.1.2 Rails 的性能

在继续介绍之前，我想花点时间说明一下 Rails 框架发布初期一个备受指责的问题：Rails 的性能很不好，例如不能处理较大的访问量。这个问题之所以存在是因为有些人没搞清状况，性能是要在你的网站中优化，而不是在框架中，强大的 Rails 只是一个框架而已。所以上面的问题应该换个角度来看：使用 Rails 开发的网站可以做性能优化吗？这样的问题已经得到了肯定的回答，因为很多世界上访问量最大的网站就是用 Rails 开发的。实际上性能优化涉及到的不仅仅是 Rails，如果你的程序需要处理类似 Hulu 或 Yellow Pages 这种数量级的访问量，Rails 并不会拖你的后腿。

3. 译者注：Ruby 中文社区也有一些质量比较高的资源，如 RailsCasts China，Happycasts 等

1.1.3 本书排版约定

本书中使用的排版约定很多都是不言自明的，在本节我要说一下那些意义不是很清晰的部分。

本书的 HTML 版和 PDF 版都包含了大量的链接，有内部各章节之间的链接（例如 1.2节），也有链接到其他网站的链接（例如Ruby on Rails 下载页面）。[4]

本书中很多例子都用到了命令行命令，为了行文方便，所有的命令行示例都使用了 Unix 风格的命令行提示符（美元符号），例如：

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

Windows 用户要知道在 Windows 中命令行的提示符是 >：

```
C:\Sites> echo "hello, world"
hello, world
```

在 Unix 系统中，一些命令要使用 `sudo`（超级用户的工作，“substitute user do”）执行。[5]默认情况下，使用 `sudo` 执行的命令是以管理员的身份执行的，这样就能访问普通用户无法访问的文件和文件夹了。例如 1.2.2 节中的一个例子：

```
sudo ruby setup.rb
```

在多数的 Unix/Linux/OS X 系统中默认需要使用 `sudo`，但是如果使用下面介绍的 Ruby 版本管理工具就没必要使用了，直接使用以下命令即可：

```
ruby setup.rb
```

Rails 附带了很多可以在命令行中运行的命令。例如，在 1.2.5 节中将使用下面的命令在本地运行一个开发服务器：

```
$ rails server
```

和命令提示符一样，本书也使用了 Unix 风格文件夹分隔符（例如，一个斜线 /）。例如，我的示例程序存放在：

```
/Users/mhartl/rails_projects/sample_app
```

在 Windows 中等价的文件夹可能是：

---

4. 阅读本书时你会发现内部章节之间的链接很有用，你可以查看引用的内容然后快速的回到之前的位置。在浏览器中阅读时这种操作很简单，直接点击浏览器的后退按钮就可以了，不过 Adobe Reader 和 OS X 的预览程序也为 PDF 提供了这种功能。在 Reader 中，在文档中点击鼠标右键，然后选择“上一个视图”就可以返回了。在预览程序中要使用“浏览（Go）”菜单：浏览->返回（Back）。

5. 很多人都错误地认为 `sudo` 的全称是“superuser do”，因为它以超级用户（root）的身份执行命令。其实，`sudo` 结合了 `su` 命令和英语单词“do”，而 `su` 的全称是“substitute user”，不信你可以执行 `man su` 命令验证一下。

```
C:\Sites\sample_app
```

一个程序的根目录称为“Rails 根目录”，但是这个称呼很容易让一些人产生困惑，他们以为“Rails 根目录”是指 Rails 框架的根目录。为了避免歧义，本书将使用“程序根目录”替代“Rails 根目录”的称呼，程序中所有文件夹都是相对该目录的。例如，示例程序的 `config` 目录是：

```
/Users/mhartl/rails_projects/sample_app/config
```

这个程序的根目录就是 `config` 之前的部分：

```
/Users/mhartl/rails_projects/sample_app
```

为了方便，如果需要指向下面这个文件

```
/Users/mhartl/rails_projects/sample_app/config/routes.rb
```

我会省略前面的程序根目录，直接写成 `config/routes.rb`。

本书经常需要显示一些来自其他程序（命令行，版本控制系统，Ruby 程序等）的输出，因为系统之间存在差异，你所得到的输出结果可能和本书中的不同，但是无需担心。

你在使用某些命令时可能会导致一些错误的发生，我不会一一列举各个错误的解决方法，你可以自行通过 Google 搜索解决。如果你在学习本书的过程中遇到了问题，我建议你看一下本书帮助页面中列出的资源。

1.2 搭建环境

> 我认为第一章就像法学院的“淘汰阶段”一样，如果你能成功的搭建开发环境，后面就会很顺利。
>
> – 本书读者 Bob Cavezza

现在可以开始搭建 Ruby on Rails 开发环境并创建第一个程序了。本节的知识量比较大，特别是对于没有很多编程经验的人来说，所以如果在某个地方卡住了也不要灰心，不只你一个人如此，每个开发者都是从这一步走过来的，慢慢来，功夫不负有心人。

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，但基本上分为两类：文本编辑器+命令行的环境和“集成开发环境”（IDE=Integrated Development Environment）。先来说说后一种。

IDE

使用最多的 Rails IDE 是 RadRails 和 RubyMine。我听说 RubyMine 不错，一个读者（David Loeffler）还总结了一篇文章讲解如何结合本书使用 RubyMine。如果你习惯使用 IDE，我建议你把上面提到的几个全都试一下，找到最符合自己要求的一款。

图 **1.1**：文本编辑器和命令行的开发环境

文本编辑器和命令行

较之 IDE，我更喜欢使用文本编辑器编辑文本，使用命令行执行命令（如图 1.1）。如何组合取决于你的喜好和所用的平台。

- **文本编辑器**：我推荐使用 Sublime Text 2，这是一个跨平台支持的文本编辑器，简单易学且功能强大。[6] Sublime Text 深受 Textmate 的影响，它能兼容大多数 Textmate 的定制功能，例如代码片段和配色方案。（Textmate 只可在 OS X 中使用，如果你使用 Mac 的话，它仍然是一个很好的选择。）另外一个很好的选择是 Vim [7]，它有针对各种主要平台的版本。Sublime Text 需要付费，而 Vim 则是免费的。二者都是界内人士普遍使用的编辑器，但就我的经验而言，Sublime Text 对初学者更友好。
- **终端（命令行）**：OS X 系统中我推荐使用 iTerm 或是内置的终端程序。Linux 系统默认的终端就很好。在 Windows 中，很多用户选择在虚拟机中运行 Linux 来开发 Rails 程序，那么终端就使用默认的好了。如果你是在纯 Windows 系统中开发，我推荐使用 Rails Installer 中附带的终端。

如果你决定使用 Sublime Text，可以参照针对本书的安装说明来安装。（这些设置可能有些繁琐，且易于出错，所以我只推荐给高级用户使用。即便不做这些设置，Sublime Text 也是编写 Rails 应用程序的绝佳工具。）

浏览器

虽然浏览器有很多选择，但是大多数的 Rails 开发者使用 Firefox、Safari 或 Chrome 进行开发。这三个浏览器都内建了“查看元素功能”，在页面中右键点击（或加 Control 键）可以调出。

6. 写作本书时，Sublime Text 3 已经推出了 Beta 版，但只对 Sublime Text 2 的注册用户开放。我只建议尝鲜用户升级。
7. vi 编辑器是 Unix 中古老而强大的强力工具，Vim 是 vi 的增强版（vi improved）。

关于工具的一点说明

在搭建开发环境的过程中，你会发现花费了很多时间来熟悉各种工具。特别是学习编辑器和 IDE，需要花费特别长的时间。单单用在 Sublime Text 和 Vim 教程上的时间就可能是几个星期。如果你刚刚接触这一领域，我要告诉你，学习工具要花费时间是正常的。每个人都是这样过来的。有时你会抓狂，当你在脑中有了很好的程序构思时，你只想学习 Rails，但却要浪费一个星期去学习老旧的 Unix 编辑器，这时很容易失去耐心。但请记住，工欲善其事必先利其器。

1.2.2 安装 Ruby，RubyGems，Rails 和 Git

> 世界上几乎所有的软件不是无法使用就是很难使用。所以用户惧怕软件。用户们已经得到经验了，不论是安装软件还是填写一个在线表格，都不会成功。我也害怕安装东西，可笑的是我还是个计算机科学博士。
>
> – Paul Graham，《创业者》，Jessica Livingston 著

现在可以安装 Ruby on Rails 了。我会尽量讲得浅一点，但是系统之间存在差异，很多地方都可能出现问题，如果你遇到问题的话请通过 Google 搜索，或访问本书的帮助页面。

除非有特殊说明，你应该使用和本书中所有软件的相同的版本，包括 Rails，这样才能得到相同的结果。有时候不同的补丁版本会产生相同的结果，但不完全相同，特别是针对 Rails 的版本。不过 Ruby 是个例外，1.9.2、1.9.3 和 2.0.0 都可以用于本教程，所以三者随意选择。

Rails Installer（Windows）

以前在 Windows 中安装 Rails 是件很痛苦的事，但多亏了 Engine Yard 公司的大牛们，特别是 Nic Williams 博士和 Wayne E. Seguin，现在在 Windows 中安装 Rails 及相关的软件简单多了。如果你使用 Windows 的话，可以到 Rails Installer 的网站下载 Rails Installer 安装程序，顺便可以看一下安装视频。双击安装文件按照说明安装 Git、Ruby、RubyGems 和 Rails。安装完成后你就可以直接跳到 1.2.3 节去创建第一个应用程序了。

有一点需要说明，使用 Rails Installer 安装的 Rails 版本可能和下面介绍的方法得到的版本不一致，这可能会导致不兼容的问题。为了解决这个问题，我现在正在与 Nic 和 Wayne 一起工作，按照 Rails 版本号的顺序列出一个 Rails Installer 列表。

安装 Git

Rails 社区中的人多少都会使用一个叫 Git（1.3 节中会详细介绍）的版本控制系统，因为大家都这么做，所以你在一开始就要开始用 Git。如何在你使用的平台中安装 Git 可以参考《Pro Git》书中的“安装 Git”一节。

安装 Ruby

接下来要安装 Ruby 了。（安装的过程可能很痛苦也容易出错，其实我也很害怕安装 Ruby 新版本，不过这是学习的必经之路。）

很有可能你使用的系统已经自带了 Ruby，你可以执行下面的命令来看一下：

```
$ ruby -v
```

这个命令会显示 Ruby 的版本。Rails 4 需要使用 Ruby 1.9 或以上的版本，但最好是 Ruby 2.0。（基本上无法使用 Ruby 1.8.7。）本教程假设多数的读者使用的是 Ruby 1.9.3 或 2.0.0，不过 Ruby 1.9.2 应该也可以用。

如果你使用的是 OS X 或者 Linux，在安装 Ruby 时我强烈建议使用 Ruby 版本管理工具 RVM，它允许你在同一台电脑上安装并管理多个 Ruby 版本。（在 Windows 中可以使用 Pik）如果你希望在同一台电脑中运行不同版本的 Ruby 或 Rails 就需要它了。如果你在使用 RVM 时遇到什么问题的话，可以在 RVM 的 IRC 频道（freenode.net 上的 #rvm）中询问它的开发者 Wayne E. Seguin。[8]如果你使用的是 Linux，我推荐你阅读 Sudobits 博客中的《如何在 Ubuntu 中安装 Ruby on Rails》一文。

在此之前，OS X 用户可能要先安装 Xcode 开发者工具。相对于安装整个工具包我更推荐你安装较小的 Xcode 命令行工具包。

在安装 Ruby 之前，先要安装 RVM：

```
$ curl -L https://get.rvm.io | bash -s
```

（如果你已经安装了 RVM，应该运行下面的命令

```
$ rvm get stable
```

确保安装的是最新版本。）

然后运行下面的命令检查安装 Ruby 的需求条件：

```
$ rvm requirements
```

在我的系统中，要安装以下所示的库（使用 Homebrew，OS X 中的包管理系统）：

```
$ brew install libtool libxslt libksba openssl
```

在 Linux 中，可以使用 `apt-get` 或 `yum`。

我还得安装 YAML 库：

```
# For Mac with Homebrew
$ brew install libyaml

# For Debian-based Linux systems
$ apt-get install libyaml-dev
```

8. 如果你从未使用过 IRC，我建议你先搜索一下“irc client <你的平台>”。OS X 上的客户端有 Colloquy 和 LimeChat。当然网页客户端总是可以使用的 http://webchat.freenode.net/?channels=rvm。

```
# For Fedora/CentOS/RHEL Linux systems
$ yum install libyaml-devel
```

最后，我要告知 RVM，OpenSSL 的安装位置，才能安装 Ruby 2.0.0：

```
$ rvm install 2.0.0 --with-openssl-dir=$HOME/.rvm/usr
<wait a while>
```

这样，至少在我的系统中，Ruby 2.0.0 已经成功安装了。

不过，安装的过程很容易出错。我尽力说明了常用的方法，更通用的方式是依靠搜索，然后自己解决。

安装 Ruby 之后，要配置一下你的系统，这样其他程序才能运行 Rails。这个过程会涉及到 gem 的安装，gem 是 Ruby 代码的打包系统。因为不同版本的 gem 会有差异，我们经常要创建一个额外的 gem 集（gemset），包含一系列的 gem。针对本教程，我推荐你创建一个名为 `railstutorial_rails_4_0` 的 gemset：

```
$ rvm use 2.0.0@railstutorial_rails_4_0 --create --default
Using /Users/mhartl/.rvm/gems/ruby-2.0.0-p0 with gemset railstutorial_rails_4_0
```

上面的命令会使用 Ruby 2.0.0 创建（`--create`）一个名为 `railstutorial_rails_4_0` 的 gemset，然后立马就开始使用（`use`） 这个 gemset，并将其设为默认的（`--default`） gemset，这样每次打开新的终端就会自动使用 `2.0.0@railstutorial_rails_4_0` 这个 Ruby 和 gemset 的组合。RVM 提供了大量的命令用来处理 gemset，更多内容可以查看其文档（http://rvm.io/gemsets/）。如果你在使用 RVM 时遇到了问题，可以运行以下的命令显示帮助信息：

```
$ rvm help
$ rvm gemset help
```

安装 RubyGems

RubyGems 是 Ruby 项目的包管理程序，有很多有用的代码库（包括 Rails）都可以通过包（或叫做 gem）的形式获取。安装 Ruby 后再安装 RubyGems 就很简单了。如果你安装了 RVM 就已经安装 RubyGems 了，因为 RVM 已经自动将其安装了：

```
$ which gem
/Users/mhartl/.rvm/rubies/ruby-2.0.0-p0/bin/gem
```

如果你还没有安装 RubyGems，可以下载 RubyGems，解压文件，然后进入 `rubygems` 目录运行安装程序：

```
ruby setup.rb
```

（如果你遇到了权限错误的提示，参照 1.1.3 节所说的，要使用 `sudo`。）

安装 RubyGems 之后，要确保你使用的版本和本书一致：

```
gem update --system 2.1.9
```

将你的系统定格在这个版本可以避免以后因为 RubyGems 升级而产生的差异。

安装 gem 时，默认情况下 RubyGems 会生成两种不同的文档（ri 和 rdoc），但是很多 Ruby 和 Rails 开发者认为花时间生成这些文档没什么必要。（很多程序员更依靠在线文档，而不是内置的 ri 和 rdoc 文档。）为了禁止自动生成文档，我建议你执行代码 1.1 中的命令，在家目录（home directory）中创建一个名为 `.gemrc` 的 gem 配置文件，文件的内容参见代码 1.2。（波浪号"~"代表"家目录"，`.gemrc` 中的点号代表这是个隐藏文件，配置文件一般都是隐藏的。）

代码 1.1：创建 gem 配置文件

```
$ subl ~/.gemrc
```

这里的 `subl` 是 OS X 中启动 Sublime Text 的命令，你可以参照 Sublime Text 2 文档中的 "OS X 命令"一文进行设置。如果你使用的是其他系统，或者你使用的是其他的编辑器，只需换用其他相应的命令（例如，你可以直接双击来启动程序，或者使用其他的命令，如 `mate`、`vim`、`gvim` 或 `mvim`）。为了行文简洁，在本书后续的内容中当我说使用 `subl` 时，我的意思是"使用你喜好的文本编辑器打开"。

代码 1.2：在 `.gemrc` 中配置不生成 ri 和 rdoc 文档

```
install: --no-rdoc --no-ri
update: --no-rdoc --no-ri
```

安装 Rails

安装完 RubyGems 后安装 Rails 也就简单了。本教程使用 Rails 4.0，通过以下命令安装[9]：

```
$ gem install rails --version 4.0.2 --no-ri --no-rdoc
```

（如果你按照上一节所述，设置了 `.gemrc`，就不必加上最后两个选项了。但是当前的 Rails 4 候选版本必须要加上，所以我才写上的。）检查 Rails 是否安装成功，执行以下命令显示 Rails 的版本号：

```
$ rails -v
Rails 4.0.2
```

注意：如果你是通过前面介绍的 Rails Installer 安装的 Rails，所得到的版本号可能会有些不同。在写这本书的时候，版本的不同还不会带来大的问题，但是如果 Rails 升级到了更高的版本，问题可能就很严重了。我目前正在和 Engine Yard 一起工作来创建一个 Rails Installer 版本列表。

如果你使用的是 Linux，现在或许还需要安装一些其他的代码包：

```
$ sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev # 只针对 Linux
```

9. 译者注：鉴于国内的"社会主义特色"，你可能无法正常下载 gem，这时你可以使用 VPN 或者使用 RubyGems 在国内的镜像。

1.2.3 第一个程序

Rails 程序一般都是从 `rails new` 命令开始的，这个命令会在你指定的文件夹中创建一个 Rails 程序模板。首先为你的 Rails 程序新建一个文件夹，然后执行 `rails new` 命令创建第一个程序（参见代码 1.3）：

代码 1.3：运行 `rails new` 生成一个新程序

```
$ mkdir rails_projects
$ cd rails_projects
$ rails new first_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
      .
      .
      .
         run  bundle install
.
.
.
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled
gem is installed.
```

如代码 1.3 所示，运行 `rails new` 命令会在文件创建完之后自动执行 `bundle install`。如果这一步没有正确执行，先不要担心，按照 1.2.4 节中的步骤来做应该就可以了。

表格 1.1：简单介绍 Rails 默认文件结构

文件/文件夹	说明
`app/`	程序的核心文件，包含模型、视图、控制器和帮助方法
`app/assets`	程序的资源文件，如 CSS、JavaScript 和图片
`bin/`	可执行文件
`config/`	程序的设置
`db/`	数据库文件

文件/文件夹	说明
`doc/`	程序的文档
`lib/`	代码库文件
`lib/assets`	代码库包含的资源文件，如 CSS、JavaScript 和 图片
`log/`	程序的日志文件
`public/`	公共（例如浏览器）可访问的数据，如出错页面
`script/rails`	生成代码、打开终端会话或开启本地服务器的脚本
`test/`	程序的测试文件（在 3.1.2 节 中换用 `spec/`）
`tmp/`	临时文件
`vendor/`	第三方代码，如插件和 gem
`vendor/assets`	第三方代码包含的资源文件，如 CSS、JavaScript 和图片
`README.rdoc`	程序简介
`Rakefile`	`rake` 命令包含的任务
`Gemfile`	该程序所需的 gem
`Gemfile.lock`	一个 gem 的列表，确保本程序的复制版使用相同版本的 gem
`config.ru`	Rack 中间件的配置文件
`.gitignore`	git 忽略的文件类型

留意一下 `rails` 命令创建的文件和文件夹。这些标准的文件夹和文件结构（如图 1.2）是 Rails 的很多优势之一，能让你从零开始快速的创建一个可运行的简单的程序。而且因为这样的结构对 Rails 程序都是一致的，阅读其他人的代码时就显得很亲切。表格 1.1 是这些文件的简介，在本书的后续内容中将介绍其中的大多数。从 5.2.1 节 开始，首先将介绍 `app/assets` 文件夹，它是 asset pipeline（Rails 3.1 新增）的一部分，这个功能让组织和部署 CSS 和 JavaScript 等资源文件变得异常简单。

1.2.4 Bundler

创建完一个新的 Rails 程序后，你可以使用 Bundler 来安装和包含该程序所需的 gem。在 1.2.3 节 中提到过，`rails` 命令会自动执行 Bundler（通过 `bundle install`），不过本节将对程序默认包含的 gem 做些修改，然后再运行 Bundler。首先在你喜好的文本编辑器中打开 `Gemfile` 文件：

图 **1.2**：新创建的 Rails 程序的文件结构

```
$ cd first_app/
$ subl Gemfile
```

该文件内容如代码 1.4。这些代码就是常规的 Ruby 代码，现在无需关注句法，第四章将会详细的介绍 Ruby。

代码 1.4：`first_app` 默认的 `Gemfile` 文件

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

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# 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', '~> 1.0.1'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

# Use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

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

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

# Use debugger
# gem 'debugger', group: [:development, :test]
```

其中很多行代码都用 # 注释掉了，这些代码放在这是告诉你一些常用的 gem，也展示了 Bundler 的句法。现在，除了默认的 gem 我们还不需要其他的 gem。

如果不为 `gem` 命令指定一个版本号，Bundler 会自动安装 gem 的最新版本。有些 gem 的更新会带来细微但有时会破坏代码的差异，所以在本教程中我们特意加入了可以正常运行的 gem 版本号，如代码 1.5 所示（同时我们也将注释掉的代码去掉了）。

代码 1.5： 指定了 gem 版本号的 `Gemfile` 文件

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'

group :development do
  gem 'sqlite3', '1.3.8'
```

```
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'

group :doc do
  gem 'sdoc', '0.3.20', require: false
end
```

代码 1.5 加入了下面这行

```
ruby '2.0.0'
```

说明应用程序期望使用的 Ruby 版本，在部署时特别有用（参见 1.4 节）。（如果你使用的不是 Ruby 2.0.0，需要修改到实际的版本号。）

更新后的 `Gemfile` 还把 Rails 默认使用的 JavaScript 库 jQuery 的 gem 从

```
gem 'jquery-rails'
```

改为

```
gem 'jquery-rails', '2.2.1'
```

同时也将

```
gem 'sqlite3'
```

修改成

```
group :development do
  gem 'sqlite3', '1.3.8'
end
```

强制 Bundler 安装 `sqlite3` gem 的 `1.3.8` 版。注意，我们仅把 SQLite 放到了开发环境中（7.1.1 节），这样可以避免和 Heroku（1.4 节）的数据库冲突。

代码 1.5 也修改了其他几行，将

```
# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'
```

```
# 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'
```

改成了

```
gem 'sass-rails',   '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
```

如下的代码

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

会安装 `1.3.0` 或以上的最新版 `uglifier` gem（在 asset pipeline 中处理文件的压缩），当然也可以安装 `7.2` 版。而下面的代码

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

只会安装高于 `4.0.0` 版的 `coffee-rails`（也是 asset pipeline 用到的）。换句话说，`>=` 总会升级到最新版；`~> 4.0.0` 只会升级补丁版本的更新（例如从 `4.0.0` 到 `4.0.1`），而不会升级到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会产生错误，所以在本教程中我们基本上会为所有的 gem 指定明确的版本号。你可以使用任何 gem 的最新版本，还可以在 `Gemfile` 中使用 `~>`（推荐高级用户使用），但我事先提醒你，这可能会导致本教程开发的程序表现异常。

修改完 `Gemfile` 后，运行 `bundle update`[10] 和 `bundle install` 安装所需的 gem：

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

`bundle install` 命令会花费一点时间，一旦结束我们的程序就可以运行了。

10. 这个命令只有修改了 Rails gem 版本时才必须要执行，可能只有使用 Rails Installer 才会用到，不过在其他情况下执行也没有坏处。

1.2.5 rails server

运行完 1.2.3 节中介绍的 `rails new` 和 1.2.4 节 中介绍的 `bundle install` 后我们的程序就可以运行了，但怎么运行呢？Rails 自带了一个命令行程序可以在开发电脑上运行一个本地服务器：[11]

```
$ rails server
=> Booting WEBrick
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

（如果系统提示缺少 JavaScript 运行时，请浏览 execjs 位于 github 的页面查看一些可选的运行时，我建议安装 Node.js。）上述代码的提示信息告诉我们这个应用程序在 `0.0.0.0` 地址的 3000[12] 端口运行。这个地址告诉系统监听这台电脑上的每一个可用的 IP 地址。一般来说，我们可以通过一个特殊的地址 `127.0.0.1` 来查看应用程序，或者也可以使用 `localhost`。通过 http://localhost:3000 查看结果，如图 1.3 所示。

图 **1.3**：默认的 Rails 页面

点击“About your application’s environment” 可以查看应用程序的信息。结果如图 1.4 所示。（图 1.4 显示的是截图时我电脑上的环境信息，你的结果可能会与我的不同。）

11. 如 1.1.3 节中所说，在 Windows 中或许你要输入 `ruby rails server`。
12. 一般情况下网站使用的是 80 端口，但这需要特别的权限，所以 Rails 为开发服务器选择了一个没有受限制的较大值。

当然我们不是真的想使用默认的 Rails 页面，但这个页面告诉我们 Rails 可以正常运行了。我们会在 5.3.2 节中移除默认页面。

图 1.4：默认页面中的应用程序信息

1.2.6 模型-视图-控制器（MVC）

在初期阶段，概览一下 Rails 程序的工作方式（如图 1.5）多少是会有些帮助的。你可能已经注意到了，在 Rails 应用程序的文件结构（如图 1.2）中有一个文件夹叫 `app/`，其中有三个子文件夹：`models`、`views` 和 `controllers`。这暗示 Rails 采用了 MVC 架构模式，这种模式强制地将“域逻辑（domain logic）”（也叫“业务逻辑（business logic）”）和图形用户界面（GUI）的输入、表现逻辑分开。在 Web 应用程序中，“域逻辑”的典型代表是“用户（users）”、“文章（articles）”和“产品（products）”等数据模型，GUI 则是浏览器中的网页。

在 Rails 交互中，浏览器发送一个请求（request），网络服务器收到请求将其传送到 Rails 的控制器，然后决定下一步做什么。某些情况下，控制器会立即渲染视图（view）模板，生成 HTML 然后将结果发送回浏览器。对于动态网站来说，控制器会和模型（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。调用模型后，控制器再渲染视图并将生成的 HTML 代码返回给浏览器。

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 2.2.2 节 中会以演示程序为例较为深入的讨论 MVC；在后面的大型示例程序中会使用 MVC 的全部内容，3.1.2 节 将介绍控制器和视图，6.1 节将介绍模型，7.1.2 节中将把三个部分放在一起使用。

1.3 用 git 做版本控制

我们已经创建了一个可以运行的 Rails 应用程序，接下来要花点时间来做一件事，虽然不是必须做的，但是很多 Rails 开发者基本上都认为这是应该做的最基本的事情，这件事就是将应用程序的源代码放入版本控制系统中。版本控制系统可以跟踪项目代码的变化，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。每个软件开发者都应该学习使用版本控制系统。

版本控制工具很多，Rails 社区更多的会使用 git，它最初是由 Linus Torvalds 开发用来存储 Linux 内核代码的。git 的知识很多，这里我们只会介绍一些简单的内容，网络上有很多免费的资料可以阅读，我特别推荐 Scott Chacon 的《Pro Git》（Apress 2009 年出版。中文版）。之所以推荐你将代码放到 git 这个版本控制系统中是因为这几乎是 Rails 社区的普遍做法，还因为这样做更利于代码的分享（1.3.4 节），也便于程序的部署（1.4 节）。

图 **1.5**：MVC 架构的图解

1.3.1 安装与设置

如果你还没安装 git，请按照前面的介绍进行安装。（如前所述，安装的过程可以参考《Pro Git》书中的“安装 Git”一节。）

第一次运行时的设置

安装 git 后，你应该做一些只需做一次的事情：系统设置——这样的设置在每台电脑上只需做一次：

```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
```

我还想用 co 代替字数较多的 checkout 命令，那么要做如下设置：

```
$ git config --global alias.co checkout
```

本书中我基本上都会使用完整的 checkout 命令，防止你没有做以上的设置，但我自己都是使用 git co。

最后，你还可以设置编辑 git 提交信息时使用的编辑器。如果你使用的是图形界面的编辑器，例如 Sublime Text、TextMate、gVim 或 MacVim，要加上一个旗标确保编辑器会在终端中保持状态而不是立马结束命令：[13]

```
$ git config --global core.editor "subl -w"
```

如果使用其他编辑器，请使用以下代码替换 subl -w：TextMate 用 mate -w，gVim 用 gvim -f，MacVim 用 mvim -f。

设置第一个仓库

下面的步骤你每次新建一个仓库时都要执行。首先进入刚创建的应用程序的根目录，然后初始化一个新仓库：

```
$ git init
Initialized empty Git repository in /Users/mhartl/rails_projects/first_app/.git/
```

接下来要将项目的文件添加到仓库中。不过有一点要说明一下：git 默认会跟踪所有文件的变化，但是有些文件我们并不想跟踪。例如，Rails 会创建一些日志文件记录应用程序的动作，这些文件经常变化，我们并不需要版本控制系统跟踪这些文件。git 有忽略文件的机制：在应用程序的根目录创建一个名为 .gitignore 的文件，然后写入一些规则告诉 git 要忽略哪些文件即可。[14]

看一下前面的表格 1.1，rails 命令默认会在应用程序的根目录创建 .gitignore 文件，其内容如代码 1.6 所示。

代码 1.6： rails 命令默认创建的 .gitignore 文件

```
# See http://help.github.com/ignore-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
```

13. 这其实是一个特性，因为启动编辑器之后你可以继续使用命令行，不过 git 执行程序在脱离后会关闭文件，返回一个空信息，这就阻碍了提交的进行。我在这里提到这一点是因为如果不说你在使用 subl 和 gvim 时就会困惑为什么行不通。如果你发现这个注释很难懂那就完全忽略它吧。

14. 如果你没有看到 .gitignore 文件，或许你需要设置一下让文件查看器显示隐藏文件。

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

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

代码 1.6 中的代码会让 git 忽略日志文件，Rails 的临时文件（`tmp/`）和 SQLite 数据库。（为了忽略 `log/` 文件夹中的日志文件，我们用 `log/*.log` 来忽略所有以 `.log` 结尾的文件）大部分被忽略的文件都是变动频繁，而且是自动创建的，没必要把这些文件纳入版本控制。而且，当和他人协作时这些文件还可能会导致冲突。

代码 1.6 中的代码只是针对本教程的，但是系统中的一些文件也要忽略，代码 1.7 则更为全面。增强后的 `.gitignore` 文件会忽略 Rails 应用程序的文档、Vim 和 Emacs 的交换文件（swap file），以及 Mac Finder 程序生成的诡异的 `.DS_Store` 文件（针对 OS X 用户）。如果你想使用这个更全面的忽略文件，用你喜好的文本编辑器打开 `.gitignore` 文件，然后写入代码 1.7 中的代码。

代码 1.7：加强版 `.gitignore` 文件

```
# Ignore bundler config.
/.bundle

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

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

# Ignore other unneeded files.
database.yml
doc/
*.swp
*~
.project
.DS_Store
.idea
.secret
```

1.3.2 添加文件并提交

最后我们要把 Rails 项目中的文件添加到 git 中，然后提交结果。你可以使用下述命令添加所有的文件（除了 `.gitignore` 中忽略的文件）：

```
$ git add .
```

这里的点号（.）代表当前目录，git 会自动的将所有的文件，包括子目录中的文件添加到 git 中。这个命令会将项目的文件添加到暂存区域（staging area），这个区域包含未提交的改动。你可以使用 `status` 命令查看暂存区域有哪些文件：[15]

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

（显示的结果很长，所以我用点号代替了。）

用 `commit` 命令告诉 git 你想保存这些改动：

```
$ git commit -m "Initial commit"
[master (root-commit) df0a62f] Initial commit
42 files changed, 8461 insertions(+), 0 deletions(-)
create mode 100644 README.rdoc
create mode 100644 Rakefile
.
.
.
```

旗标 `-m` 允许你为这次提交添加一个信息，如果没有提供 `-m`，git 会打开你在 1.3.1 节中设置的编辑器，你需要在编辑器中填写信息。

有一点很重要，git 提交是针对本地的，数据只存在执行提交的电脑中。这一点和另一个很著名的开源版本控制系统 SVN 不同，SVN 提交时会更新远程仓库。git 将 SVN 中的提交分成了两部分：本地保存的更改（`git commit`）和将更改推送到远程仓库（`git push`）。在 1.3.5 节中会演示推送这一步。

顺便说一下，你可以使用 `log` 命令查看提交的历史信息：

```
$ git log
commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4
```

15. 如果以后你执行 `git status` 看到一些不需要的文件出现了，你就可以将其加入你的 `.gitignore` 文件中。

```
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Oct 15 11:36:21 2009 -0700

  Initial commit
```

如果要退出 `git log`，输入 `q`。

1.3.3 git 为我们带来了什么好处？

现在你可能还不是完全清楚将源码纳入版本控制系统有什么好处，那我就举个例子来说明一下吧。（后续章节中还有很多例子）假设你不小心做了一些改动，比如说删除了 `app/controllers/` 文件夹：

```
$ ls app/controllers/
application_controller.rb
$ 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/` 文件夹中的内容，用 `rm` 命令删除这个文件夹。旗标 `-rf` 的意思是“强制递归”，无需得到确认就递归的删除所有文件、文件夹、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")
```

可以看到一个文件被删除了，但是这个改动只发生在工作区，还没有提交。这样我们就可以使用 `checkout` 命令切换到前一个提交记录来撤销这次改动（其中旗标 `-f` 意思是覆盖当前的改动）：

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

删除的文件夹和文件又回来了，这下放心了！

1.3.4 GitHub

你已经将项目的代码纳入 git 版本控制系统了，现在可以将其推送到 GitHub 了。GitHub 是一个针对 Git 仓库的存储及分享社交平台。将代码的拷贝存一份在 GitHub 有两个目的：其一是对代码的完整备份（包括完整的提交历史），其二是方便以后的协作。这一步不是必须要做的，不过加入 GitHub 可以给你提供机会参与到更广为人知的开源项目中。

GitHub 有一些收费的计划，但是对开源项目的代码是免费的，如果你还没有 GitHub 的账户就赶快注册一个免费的账户吧。（或许你先要参考 GitHub 的“创建 SSH 密匙”一文）注册后，点击创建仓库的链接（New repository），然后填入所需的信息，如图 1.6 所示。（注意，不要选择使用 `README` 文件初始化仓库（Initialize this repository with a README），因为 `rails new` 已经自动创建了这个文件。）提交表单后，按照下面的方法将你第一个应用程序推送上去：

```
$ git remote add origin git@github.com:<username>/first_app.git
$ git push -u origin master
```

图 1.6： 在 GitHub 创建第一个仓库

上面的代码告诉 Git 你要添加 GitHub 上面的仓库地址为代码的原本，代表本地的主分支（`master`），然后将本地的仓库推送到 GitHub 上。（先不要关心旗标 `-u` 的作用，如果你实在好奇可以搜索“git set upstream”。）当然你要把 `<username>` 换成你真正的用户名。例如，我执行的命令是：

```
$ git remote add origin https://github.com/mhartl/first_app.git
```

然后在 GitHub 就有了这个应用程序仓库的页面，页面中有文件浏览功能，包含了完整的提交历史，还有一些其他好玩的功能（如图 1.7）。

GitHub 还提供了增强命令行界面的工具，如果你更喜欢使用 GUI 程序，可以到 GitHub for Windows 和 GitHub for Mac 页面下载。（GitHub 针对 Linux 的工具看样子就是 Git 本身了。）

图 **1.7**：GitHub 仓库页面

1.3.5 分支，编辑，提交，合并

如果你按照 1.3.4 节中所说的做了，或许已经注意到了，GitHub 在仓库的主页面会自动显示 `README` 文件的内容。在这里，因为项目是使用 `rails` 命令生成的，这个 `README` 是 Rails 自带的（如图 1.8）。根据 `.rdoc` 扩展名，GitHub 会按照一定的格式显示其内容。但是其内容本身并没有什么意义。在本节，我们将做第一次编辑，修改 `README` 文件的内容来描述我们的项目而不是 Rails 框架。在编辑的过程中你可以看到分支、编辑、提交、合并的工作流，我推荐你在 Git 中使用这样的工作流。

图 **1.8**：GitHub 中显示的项目初始生成的 `README` 文件

分支

Git 中的分支功能很强大，分支是对仓库的复制，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` 旗标创建一个新分支：

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

第二个命令，`git branch`，会将本地所有的分支列出来，分支名前面的星号（`*`）指明当前所在的分支。注意，`git checkout -b modify-README` 会创建一个新分支，然后切换到这个分支，`modify-README` 前面的星号证明了这一点。（如果你在 1.3 节中设置了别名 `co`，那么你就要使用 `git co -b modify-README` 了。）

分支的唯一价值是在多个开发人员协同开发一个项目时使开发的过程更明了，[16]不过对只有一个开发者的项目（比如本教程）也有用。一般而言，主分支是和从分支隔离开的，所以即便我们搞砸了也只需切换回到主分支并删除从分支来丢掉改动。在本节末尾我们会看到怎么做。

16. 更多内容请查看《Pro Git》书中的“Git 分支（中文版）”部分。

顺便说一下，对于较小的改动我一般不会动用新分支，这里是对好的习惯做一个演示。

编辑

创建了从分支后，我们要编辑文件让其更好的描述我们的项目。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`，GitHub 会自动为你排版。首先我们使用 Unix 命令 `mv`（移动，move）的 git 版本来修改文件名，然后写入代码 1.8 所示的内容：

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

代码 1.8：新的 `README` 文件，`README.md`

```
# Ruby on Rails Tutorial: first application

This is the first application for the
[*Ruby on Rails Tutorial*](http://railstutorial.org/)
by [Michael Hartl](http://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
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.md
#
```

这时，我们可以使用 1.3.2 节中用到的 `git add .`，但是 Git 提供了旗标 `-a`，它的意思是将现有文件的所有改动（包括使用 `git mv` 创建的文件，对 Git 来说这并不是新的文件）添加进来：

```
$ 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
```

千万别误用了 `-a` 旗标。如果在上次提交之后你向项目添加了新文件的话，首先你要使用 `git add` 告诉 Git 你添加的文件。

注意，提交的信息我们用的是现在时。Git 将提交作为一系列打补丁的活动，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 生成的提交信息更匹配。更多信息请查看 GitHub 的博文《Shiny new commit styles》。

合并

我们已经修改完了，现在可以将其合并到主分支了：

```
$ 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 经常会显示类似 `34f06b7` 的字符，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。

合并完后，我们可以清理一下分支了，使用 `git branch -d` 删除这个从分支：

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

这一步是可选的，事实上一般我们都会留着这个从分支，这样你就可以在主、从分支之间来回切换，在合适的时候将改动合并到主分支中。

如前面提到的，你可以使用 `git branch -D` 放弃对从分支所做的修改：

```
# For illustration only; don't do this unless you mess up a branch
$ git checkout -b topic-branch
$ <really screw up the branch>
$ git add .
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

和旗标 `-d` 不同，即使还未合并 `-D` 也会删除分支。

推送

我们已经更新了 `README` 文件，可以将改动推送到 GitHub 看看改动的结果。因为之前我们已经推送过一次了（1.3.4 节），在大多数系统中我们都可以省略 `origin master`，只要运行 `git push`：

```
$ git push
```

正像我们介绍的，GitHub 使用 Markdown 解析器对文件进行了排版（如图 1.9）。

图 1.9： 使用 Markdown 排版的改进版 `README` 文件

1.4 部署

即使现在还处在早期阶段，我们还是要将我们（没什么内容）的 Rails 应用程序部署到生产环境。这一步是可选的，不过在开发过程中尽早、频繁的部署可以尽早的发现开发中的问题。在开发环境中极力解决问题之后再部署,等到发布日期到来时经常会导致严重的问题。[17]

17. 虽然针对本书的示例程序你无需关心，不过如果你很担心太早的将你的应用程序公开，参考在 1.4.4 节中提供几个方法。

过去部署 Rails 应用程序是很痛苦的事，但最近几年 Rails 开发群体不断的成熟，现在有很多好的解决方案了。这些方案包括运行 Phusion Passenger（Apache 和 Nginx[18] 网络服务器的一个模块）的共享主机或私有虚拟服务器，Engine Yard 和 Rails Machine 这种提供全方位部署服务的公司，Engine Yard Cloud 和 Heroku 这种云部署服务。

我最喜欢的部署方案是 Heroku，这是一个特别针对 Rails 和其他 Ruby Web 应用程序[19]的托管平台。Heroku 让 Rails 应用程序的部署变得异常简单，只要你的源码纳入了 Git 版本控制系统就好。（这也是为什么你要按照 1.3 节 中介绍的步骤安装 Git 的原因，如果你还没有安装就赶快安装吧。）本节下面的内容就是介绍如何将我们的第一个应用程序部署到 Heroku。

1.4.1 搭建 Heroku 部署环境

Heroku 使用 PostgreSQL（发音 post-gres-cue-ell，经常简称 Postgres），所以我们要把 `pg` 加入生产组，Rails 才能和 PostgreSQL 通信：

```
group :production do
  gem 'pg', '0.15.1'
end
```

我们在 1.2.4 节说过，最好指明应用程序期望使用的 Ruby 版本：

```
ruby '2.0.0'
```

（如果你使用的是 Ruby 1.9.3，就要改成 `ruby 1.9.3`，对本教程来说，这两个版本不会有太大区别。）在代码 1.5 的基础上作上述修改，得到代码 1.9：

代码 1.9：添加了一些 gem，并指定了 Ruby 版本后的 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'
#ruby-gemset=railstutorial_rails_4_0

gem 'rails', '4.0.2'

group :development do
  gem 'sqlite3', '1.3.8'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '3.0.4'
gem 'turbolinks', '1.1.1'
```

18. 发音为“Engine X”。
19. Heroku 可以正常运行任意一个使用 Rack 中间件的 Ruby Web 程序，Rack 为 Web 框架和 Web 服务器之间提供了标准接口。Ruby 社区很严格的承袭了这一标准，很多框架都是这样做的，例如 Sinatra、Ramaze、Camping 和 Rails，这也就意味着 Heroku 基本上支持所有的 Ruby Web 应用程序。

```
gem 'jbuilder', '1.0.2'

group :doc do
  gem 'sdoc', '0.3.20', require: false
end

group :production do
  gem 'pg', '0.15.1'
  gem 'rails_12factor', '0.0.2'
end
```

安装这些 gem，要执行 `bundle install`，并指定旗标：

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

指定 `--without production` 后本地环境就不会安装生产环境所需的 gem 了，对本例来说有 `pg` 和 `rails_12factor` 两个 gem。（如果 Bundler 显示一个 `readline` 相关的错误，请把 `gem rb-readline, '~> 0.4.2'` 加入 `Gemfile`。）因为我们添加的 gem 只针对生产环境，所以现在执行这个命令不会在本地安装任何 gem，但这一步却是必须的，因为我们要把 `pg` 和 Ruby 版本更新到 `Gemfile.lock` 中，这样 Heroku 才知道应用程序要使用的 Ruby 版本。然后提交上述改动：

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

随后还要做一些设置，生成 Heroku 服务静态资源所需的文件，例如图片和 CSS：

```
$ rake assets:precompile
$ git add .
$ git commit -m "Add precompiled assets for Heroku"
```

（这里用到了 `rake` 命令，在 2.2 节 会详细介绍。）

然后要注册并设置一个新 Heroku 账户。第一步是注册，然后查看邮箱，完成注册，再使用 Heroku Toolbelt 安装所需的软件。接着打开命令行，通过 `heroku` 命令登录：

```
$ heroku login
```

最后，回到 Rails 应用程序所在目录，使用 `heroku` 命令在 Heroku 的服务器上创建一个区域放置你的应用程序（参照代码 1.10）。

代码 1.10：在 Heroku 上新建一个应用程序

```
$ cd ~/rails_projects/first_app
$ heroku create
Created http://stormy-cloud-5881.herokuapp.com/ |
git@heroku.com:stormy-cloud-5881.herokuapp.com
Git remote heroku added
```

heroku 命令会为你的应用程序新建一个子域名，立马就可以生效。当然，现在还看不到内容，让我们开始部署吧。

1.4.2 Heroku 部署第一步

要部署到 Heroku，第一步是通过 Git 将应用程序推送到 Heroku 中：

```
$ git push heroku master
```

图 1.10：本教程的第一个应用程序运行在 Heroku 上

1.4.3 Heroku 部署第二步

事实上没有第二步了。我们已经完成部署了（如图 1.10）。你可以通过 heroku create 命令给出的地址（参照代码 1.9，但那里的地址是我的应用程序的）查看你刚刚部署的应用程序了。你可以向 heroku 命令传递一个参数来让其自动启动浏览器并打开你的地址：

```
$ heroku open
```

糟糕，显示的是一个错误页面。对 Rails 4 来说，由于技术原因，Heroku 无法显示默认页面。幸好当我们在 5.3.2 节添加跟路由后错误就会消失了。

部署成功后，Heroku 会提供一个很精美的界面管理和设置你的应用程序（如图 1.11）。

1.4.4 Heroku 的其他命令

Heroku 提供了很多命令，本书只介绍了几个基本的。下面再介绍一个命令，用来重命名应用程序：

```
$ heroku rename railstutorial
```

图 **1.11**：Heroku 提供的精美界面

你不要再使用这个名字了，我已经使用了。事实上，现在你无需做这样的修改，使用 Heroku 提供的默认值就行。不过如果你真的想重命名你的应用程序，你可以基于安全的考虑使用一些随机或难猜测到的名字，例如：

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

使用这样随机的域名，只有你将地址告诉别人他们才能访问你的网站。（顺便说一下，让你一窥 Ruby 的强大，以下是我用来生成随机域名的代码，很精妙吧。）

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

除了支持子域名，Heroku 也支持自定义域名。（事实上本书的网站[20]就放在 Heroku 上。如果你阅读的是本书的在线版，你现在就正在浏览一个托管在 Heroku 上的网站。）在 Heroku 文档 中可以查看更多关于自定义域名的信息以及其他 Heroku 相关的话题。

1.5 小结

这一章中做的事情真不少：安装，搭建开发环境，版本控制以及部署。如果你现在想分享阅读本书的进度，你可以发一条推或者更新 Facebook 状态写上类似下面的内容：

我正在阅读 @railstutorial 学习 Ruby on Rails！ http://railstutorial.org/

20. 译者注：英文版原书的网站托管于 Heroku，本中文版在 Github（http://railstutorial-china.org）上

此页留白

第 2 章 演示程序

本章我们要开发一个简单的演示应用程序来展示一下 Rails 强大的功能。我们会使用脚手架（scaffold）功能快速的生成程序，这样就能以一定的高度概览一下 Ruby on Rails 编程的过程（也能大致的了解一下 Web 开发）。正如在第一章的旁注 1.1 中所说，本书将采用另一种方法，我们会循序渐进的开发程序，遇到新的概念都会详细说明，不过为了概览功能（也为了寻找成就感）也无需对脚手架避而不谈。我们可以通过 URL 和最终的演示程序进行交互，了解一下 Rails 应用程序的结构，也第一次演示 Rails 使用的 REST 架构。

和后面的大型示例程序类似，这个演示程序将包含用户（users）和微博（microposts）两个模型（因此实现了一个小型的 Twitter 类程序）。程序的功能还需要后续的开发，而且开发过程中的很多步骤看起来也很神秘，不过暂时不用担心：从第三章起将从零开始再开发一个类似的程序，我还会提供大量的资料供后续参考。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 规划程序

在这一节我们要规划一下这个演示程序。和 1.2.3 节类似，我们先使用 `rails` 命令生成程序的骨架。

```
$ cd ~/rails_projects
$ rails new demo_app
$ cd demo_app
```

然后我们用一个文本编辑器修改 `Gemfile`，写入代码 2.1 所示的代码。

代码 2.1：演示程序的 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'

group :development do
  gem 'sqlite3', '1.3.8'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'
```

```
group :doc do
  gem 'sdoc', '0.3.20', require: false
end

group :production do
  gem 'pg', '0.15.1'
end
```

注意，代码 2.1 和代码 1.9 是一样的。

和 1.4.1 节一样，在本地安装 gem 时指定 `--without production` 选项不安装生产环境所需的 gem：

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

（再次提醒，如果 Bundler 提示一个和 `readline` 有关的错误，请在 `Gemfile` 中加入 `gem rb-readline`。）

图 2.1： 为演示程序在 GitHub 新建一个仓库

最后我们还要把演示程序纳入版本控制。提醒一下，`rails` 命令会生成一个默认的 `.gitignore` 文件，不过对于你所使用的系统而言代码 1.7 中的代码似乎更有用。然后初始化一个 Git 仓库，做第一次提交：

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

你可以重新创建一个仓库然后将代码推送到 GitHub：

```
$ git remote add origin https://github.com/<username>/demo_app.git
$ git push -u origin master
```

（和第一章中的程序一样，注意不要使用 GitHub 自动生成的 `README` 文件初始化仓库。）

下面要开发这个程序了。开发 Web 应用程序一般来说第一步是创建数据模型（data model），模型代表应用程序所需的结构。对我们这个程序而言，它是个轻博客，有用户和微博。那么我们先为程序创建一个用户（users）模型（2.1.1 节），然后再添加微博（microposts）模型（2.1.2 节）。

2.1.1 用户模型

不同的注册表单代表了不同的用户数据模型，我们将选择一种简化的模型。这个演示程序的用户要有一个唯一的标识符 `id`（整数 `integer`），一个对外显示的名字 `name`（字符串 `string`），还有一个 Email 地址 `email`（字符串 `string`），它将同时兼任用户名。用户模型的结构如图 2.2。

users	
id	integer
name	string
email	string

图 **2.2**：用户数据模型

我们会在 6.1.1 节中介绍，图 2.2 中的标签 `users` 代表数据库中的一个表，`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的模型还要简单：微博要有一个 `id` 和一个内容 `content`（字符串 `string`）。[1] 不过还有一个比较复杂的数据要实现：将微博和用户关联起来，我们使用 `user_id` 来存储微博的拥有者。最终的数据模型如图 2.3。

microposts	
id	integer
content	string
user_id	integer

图 **2.3**：微博的数据模型

在 2.3.3 节中我们会看到怎样使用 `user_id` 字段简单的实现一个用户拥有多个微博的功能（第 10 章会做更详尽的介绍）。

1. 如果要实现内容更长的文章，例如一篇常规博客中的文章，应该将字符串类型（`string`）换成文本类型（`text`）；

2.2 Users 资源（users resource）

本节我们将要实现 2.1.1 节中设定的用户数据模型，还会为这个模型创建基于网页的界面。这二者结合起来就是一个“Users 资源”，“资源”的意思是将用户设想为对象，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，我们的 Users 资源会使用脚手架功能生成，Rails 内置了这样的功能。我强烈建议你先不要细看生成的代码，在这个时候看只会让你更困惑。

将 `scaffold` 传递给 `rails generate` 就可以使用 Rails 的脚手架功能了。传给 `scaffold` 的参数是资源名的单数形式（本例中就是 `User`），后面可以再跟着指定数据模型的字段：[2]

```
$ rails generate scaffold User name:string email:string
      invoke  active_record
      create    db/migrate/20130305221714_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  jbuilder_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
       exist      app/views/users
      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
```

2. 脚手架后面跟着的名字和模型一样，是单数形式，而资源和控制器是复数形式。因此是 `User` 而不是 `Users`；

```
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

上面代码中的命令加入了 `name:string` 和 `email:string`，这样我们就可以实现如图 2.2 所示的用户模型了。（注意没必要指定 `id`，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：

```
$ 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 User 数据模型更新数据库。（在 6.1.1 节中将详细介绍数据库迁移）注意，为了使用 `Gemfile` 中指定的 Rake 版本，我们通过 `bundle exec` 来执行 `rake`。（如果你使用 RVM，如前一章所说，可以不加 `bundle exec`，但为了命令完整，我会一直都加上。不加 `bundle exec` 还有其他办法，参见 3.6.1 节。）

然后我们可以使用 `rails s`（`rails server` 的缩略形式）来启动本地服务器：

```
$ rails s
```

现在演示程序应该已经可以通过 http://localhost:3000/ 查看了。

旁注 2.1：Rake

在 Unix 中，在将源码编译成可执行程序的过程中，make 组件起了很重要的作用。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
```

这行代码在 Unix 中（包括 Linux 和 Mac OS X）会对代码进行编译。

Rake 就是 Ruby 版的 make，用 Ruby 编写的类 make 程序。Rails 灵活的运用了 Rake 的功能，特别是提供了一些用来开发基于数据库的 Web 程序所需的任务。`rake db:migrate` 是最常用的了，还有很多其他的命令，你可以运行 `rake -T db` 来查看所有和数据库有关的任务：

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

如果要查看所有的 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，你不需要知道所有的（或大多数）命令。学完本教程后你会知道所有重要的命令。

图 **2.4**：Users 资源的初始索引页面（/users）

2.2.1 浏览用户相关的页面

访问根地址 http://localhost:3000/ 得到的还是如图 1.3 所示的 Rails 程序默认页面，不过使用脚手架生成 Users 资源的时候也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的浏览一下这些用户相关的页面。浏览的时候你会发现表格 2.1 很有用，表中显示了页面和 URL 地址之间的对应关系。

我们先来看一下显示所有用户的页面，叫做“index”，如你所想，目前还没有用户存在。（如图 2.4）

如果想创建新用户就要访问“new”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，因此在后面的内容中我会省略这一部分）在第七章中我们会将其改造成用户注册页面。

表格 2.1：Users 资源中页面和 URL 的对应关系

URL	动作（Action）	目的
/users	`index`	显示所有用户的页面
/users/1	`show`	显示 ID 为 1 的用户的页面
/users/new	`new`	创建新用户的页面
/users/1/edit	`edit`	编辑 ID 为 1 的用户的页面

图 **2.5**：创建新用户的页面（/users/new）

你可以在表格中填入名字和 Email 地址，然后点击创建用户（Create User）按钮来创建一个用户。然后就会显示这个用户的页面（`show`），如图 2.6 所示。（页面中的绿色文字是通过 Flash 消息实现的，会在 7.4.2 节中介绍）注意页面的地址是 /users/1，正如你猜想的，这里的 1 就是图 2.2 中的用户 `id`。在 7.1 节 中会将其打造成用户的资料页面。

如果要修改用户的信息就要访问编辑（edit）页面了（如图 2.7）。修改用户的信息后点击“更新用户（Update User）”按钮就更改了演示程序中该用户的信息（如图 2.8）。（在第 6 章我们会看到，用户的数据存储在后端的数据库中。）我们会在 9.1 节中添加编辑和更新用户的功能。

图 **2.6**：显示某个用户的页面（/users/1）

现在我们重新回到创建新用户页面，然后提交表格创建第二个用户。然后访问用户索引（index）页面（如图 2.9）。7.1 节将美化一下这个显示所有用户的页面。

我们已经演示了创建、展示、编辑用户的页面，下面要演示销毁用户页面了（如图 2.10）。点击图 2.10 中的链接会出现一个验证对话框，确认后就会删除第二个用户，索引页面就只会显示一个用户。（如果这个操作没有顺利完成，请确保浏览器启用了 JavaScript 支持。销毁用户时 Rails 是通过 JavaScript 发送请求的。）9.4 节会增强用户的删除功能，只有管理员级别的用户才能删除用户。

2.2.2 MVC 实践

我们已经大概的浏览了 Users 资源，下面我们要用 1.2.6 节中介绍的 MVC 的视角来仔细的看一下其中某些特定的部分。我们会分析在浏览器中做一次点击的内在过程，这里通过访问用户索引页面做演示，来了解一下 MVC。（如图 2.11）

1. 浏览器向 /users 发起一个请求；
2. Rails 的路由将 /user 分配到 Users 控制器的 `index` 动作；
3. `index` 动作向 User 模型获取所有的用户（`User.all`）；
4. User 模型从数据库中将所有的用户读取出来；
5. User 模型将所有的用户返回给控制器；
6. 控制器将获得的所有用户数据赋予 `@users` 变量，然后传递给 `index` 的视图；

7. 视图使用内嵌 Ruby 代码的模板渲染成 HTML；
8. 控制器将生成的 HTML 发送回浏览器。[3]

首先我们要从浏览器中发起一个请求，你可以直接在浏览器地址栏中敲入地址，也可以点击页面中的链接。（图 2.11 中的第 1 步）接着请求到达 Rails 路由（第 2 步），根据 URL 将其分发到适当的控制器动作（而且还会考量请求的类型，旁注 3.2 中会介绍）。将 Users 资源中相关的 URL 映射到控制器动作的代码如代码 2.2 所示。这些代码会按照表格 2.1 中的对应关系做映射。（`:users` 是一个 Symbol，4.3.3 节会介绍）

代码 2.2： Rails 的路由设置，包含一条 Users 资源的规则
`config/routes.rb`

```
DemoApp::Application.routes.draw do
  resources :users
  .
  .
  .
end
```

图 2.7： 编辑用户的页面（/users/1/edit）

3. 有些文章会说是视图直接将 HTML 返回给浏览器的（通过 Web 服务器，例如 Apache 和 Nginx）。不管实现的细节是怎样的，我更相信控制器是一个中枢，应用程序中所有的信息都会通过它；

2.2.1 节中浏览的页面就对应了 Users 控制器中不同的动作。脚手架生成的控制器代码大致如代码 2.3 所示。注意一下 `class UsersController < ApplicationController` 的用法，这是 Ruby 中类继承的写法。（2.3.4 节中将简要的介绍一下继承，4.4 节将详细介绍类和继承。）

代码 2.3： 用户控制器的代码概要
app/controllers/users_controller.rb

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

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def edit
    .
    .
    .
  end

  def update
    .
    .
    .
  end
```

```
  def destroy
    .
    .
    .
  end
end
```

图 2.8： 显示“信息已更新”提示的用户页面

或许你发现了动作的数量比我们看过的页面数量要多，`index`、`show`、`new` 和 `edit` 对应了 2.2.1 节中介绍的页面。不过还有一些其他的动作，`create`、`update` 和 `destroy` 等，这些动作一般不会直接渲染页面（不过有时也会），它们只会修改数据库中保存的用户数据。表格 2.2 列出的是控制器的全部动作，这些动作就是 Rails 对 REST 架构（参见旁注 2.2）的实现。REST 是由计算机科学家 Roy Fielding 提出的概念，意思是表现层状态转化（Representational State Transfer）。[4] 注意表格 2.2 中的内容，有些部分是有重叠的。例如 `show` 和 `update` 两个动作都映射到 /users/1 这个地址上。二者的区别是它们所用的 HTTP 请求方法不同。3.2.1 节将更详细的介绍 HTTP 请求方法。

4. 加利福尼亚大学欧文分校 2000 年 Roy Thomas Fielding 的博士论文《架构风格与基于网络的软件架构设计》（译者注：中文翻译）

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，你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、基于网络的系统和程序，例如 WWW 和 Web 应用程序。REST 理论是很抽象的，在 Rails 程序中，REST 意味着大多数的组件（例如用户和微博）会被模型化，变成资源（resource），可以被创建（create）、读取（read）、更新（update）和删除（delete），这些操作会与关系型数据库中的 CRUD 操作和 HTTP 请求方法（`POST`，`GET`，`PATCH` 和 `DELETE`）对应起来。（3.2.1 节，特别是旁注 3.3，将更详细的介绍 HTTP 请求）

作为 Rails 程序开发者，REST 开发方式会帮助你决定编写哪些控制器和动作：你只需简单的将可以创建、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 11 章中将看到 REST 架构允许我们将一个很棘手的问题（“关注用户”功能）通过一种自然而便捷的方式处理。

图 **2.9**：显示了第二个用户的用户索引页面（/users）

图 2.10：销毁用户

表格 2.2：代码 2.2 中 Users 资源生成的符合 REST 架构的路由

HTTP 请求	**URL**	动作	目的
`GET`	/users	`index`	显示所用用户的页面
`GET`	/users/1	`show`	显示 ID 为 1 的用户页面
`GET`	/users/new	`new`	创建新用户的页面
`POST`	/users	`create`	创建新用户
`GET`	/users/1/edit	`edit`	编辑 ID 为 1 的用户页面
`PATCH`	/users/1	`update`	更新 ID 为 1 的用户
`DELETE`	/users/1	`destroy`	删除 ID 为 1 的用户

图 **2.11**：Rails 中 MVC 的详细说明图解

为了解释 Users 控制器和 User 模型之间的关系，我们要看一下简化了的 `index` 动作的代码，如代码 2.4 所示。（脚手架生成的代码很粗糙，所以我们做了简化）

代码 2.4：演示程序中被简化了的用户 `index` 动作
`app/controllers/users_controller.rb`

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

`index` 动作有一行代码是 `@users = User.all`（图 2.11 中的第 3 步），它要求 User 模型从数据库中取出所有的用户（第 4 步），然后将结果赋值给 `@users` 变量（第 5 步）。User 模型的代码参见代码 2.5。代码看似简单，不过它通过继承具备了很多功能（参见 2.3.4 节 和 4.4 节）。简单来说就是通过调用 Rails 中叫做 Active Record 的库，代码 2.5 中的 `User.all` 就会返回所有的用户。

代码 2.5： 演示程序中的 User 模型
app/models/user.rb

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

一旦定义了 `@users` 变量，控制器就会调用视图代码（第 6 步），其代码如代码 2.6 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` 视图的代码会遍历 `@users`，为每个用户生成一行 HTML。（记住，你现在可能读不懂这些代码，这里只是让你看一下这些代码是什么样子。）

代码 2.6： 用户索引页面的视图代码
app/views/users/index.html.erb

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

<table>
  <tr>
    <th>Name</th>
    <th>Email</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.email %></td>
    <td><%= link_to 'Show', user %></td>
    <td><%= link_to 'Edit', edit_user_path(user) %></td>
    <td><%= link_to 'Destroy', user, method: :delete,
                                     data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

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

视图会将代码转换成 HTML（第 7 步），然后控制器将其返回浏览器显示出来（第 8 步）。

2.2.3 上述 Users 资源的缺陷

脚手架生成的 Users 资源相关代码虽然能够让你大致的了解一下 Rails，不过它也有一些缺陷：

- **没有对数据进行验证（validation）**。User 模型会接受空的名字和不合法的 Email 地址而不会报错。

- **没有用户身份验证机制（authentication）**。没有实现登录和退出功能，随意一个用户都可以进行任何的操作。
- **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，没有对数据进行验证，不包含验证机制的测试，以及其他的需求。
- **没有布局**。没有共用的样式和网站导航。
- **没有真正的被理解**。如果你能读懂脚手架生成的代码就不需要阅读本书了。

2.3 Microposts 资源

我们已经生成也浏览了 User 资源，现在要生成 Microposts 资源了。阅读本节时我推荐你和 2.2 节对比一下，你会看到两个资源在很多方面都是一致的。通过这样重复的生成资源我们可以更好的理解 Rails 中的 REST 架构。在这样的早期阶段看一下 Users 资源和 Microposts 资源的相同之处也是本章的主要目的之一。（后面我们会看到，开发一个比本章的演示程序复杂的程序要付出很多汗水，Microposts 资源在第 10 章才会用到，而我不想这么晚才介绍。）

2.3.1 概览 Microposts 资源

和 Users 资源一样，我们使用 `rails generate scaffold` 命令生成 Microposts 资源的代码，实现图 2.3 中所示的数据模型：[5]

```
$ rails generate scaffold Micropost content:string user_id:integer
      invoke  active_record
      create    db/migrate/20130307005528_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  jbuilder_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
```

5. 和生成 Users 资源的脚手架命令一样，生成 Microposts 资源的脚手架也使用了单数形式，因此我们使用 `generate Micropost`；

```
      invoke      test_unit
      create        test/helpers/microposts_helper_test.rb
      invoke    jbuilder
       exist      app/views/microposts
      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
```

然后要更新数据库，使用最新的数据模型，我们要执行类似 2.2 节中用到的迁移命令：

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

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。就像你猜测的，脚手架也会更新 Rails 的路由文件，为 Microposts 资源加入一条规则，如代码 2.7 所示。[6] 和 Users 资源一样，`resources :micropsts` 会将微博相关的 URL 地址映射到 Microposts 控制器，如表格 2.3 所示。

表格 2.3：代码 2.7 中 Microposts 资源生成的符合 REST 架构的路由

HTTP 请求	URL	动作	目的
GET	/microposts	index	显示所有微博的页面
GET	/microposts/1	show	显示 ID 为 1 的微博页面
GET	/microposts/new	new	显示创建新微博的页面
POST	/microposts	create	创建新微博
GET	/microposts/1/edit	edit	编辑 ID 为 1 的微博页面
PATCH	/microposts/1	update	更新 ID 为 1 的微博
DELETE	/microposts/1	destroy	删除 ID 为 1 的微博

代码 2.7： Rails 的路由配置，有一条针对 Microposts 资源的新规则
`config/routes.rb`

6. 和代码 2.7 相比，脚手架生成的代码可能会有额外的空行。你无须担心，因为 Ruby 会忽略额外的空行；

```
DemoApp::Application.routes.draw do
  resources :microposts
  resources :users
  .
  .
  .
end
```

Microposts 控制器的代码简化后如代码 2.8 所示。注意，除了将 `UsersController` 换成 `MicropostsController` 之外，这段代码和代码 2.3 没什么区别。这说明了这两个资源在 REST 架构中的共同之处。

代码 2.8： Microposts 控制器的代码简化形式
`app/controllers/microposts_controller.rb`

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

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def edit
    .
    .
```

```
    .
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

我们在创建微博页面（/microposts/new）输入一些内容来添加一个微博，如图 2.12 所示。

图 2.12：创建微博的页面（/microposts/new）

既然已经在这个页面了，那就多创建几个微博，确保至少有一个微博的 `user_id` 设为了 `1`，这样就对应到 2.2.1 节中创建的第一个用户了。结果应该和图 2.13 类似。

2.3.2 限制微博内容的长度

如果要称得上微博这样的名字就要限制其内容的长度。在 Rails 中实现这种限制很简单，使用数据验证（validation）功能。要限制微博的长度最大为 140 个字符（就像 Twitter 一样），我们可以使用长度限制数据验证。现在你可以用你的文本编辑器或 IDE 打开 `app/models/micropost.rb` 写入代码 2.9 所示的代码。（代码 2.9 中使用的 `validates` 方法只针对 Rails 3；如果你之前用过 Rails 2.3，就可以对比一下它和 `validates_length_of` 的区别。）

图 2.13：微博索引页面（/microposts）

代码 2.9：现在微博的长度最长为 140 个字符
`app/models/micropost.rb`

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

上面的代码看起来可能很神秘，我们会在 6.2 节中详细介绍数据验证。如果我们在创建微博页面输入超过 140 个字符的内容就会看到这个验证的样子了。如图 2.14 所示，Rails 会显示一个错误提示信息（error message）提示微博的内容太长了。（7.3.3 节将更详细的介绍错误信息）

2.3.3 一个用户有多篇微博

Rails 强大的功能之一是可以为不同的数据模型之间创建关联（association）。针对本例中的 User 模型，每个用户可以有多篇微博。我们可以通过更新 User 模型（参见代码 2.10）和 Micropost 模型（参见代码 2.11）的代码来实现这种关联。

图 **2.14**：创建微博失败后显示的错误信息

代码 **2.10**：一个用户有多篇微博
app/models/user.rb

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

代码 **2.11**：一篇微博只属于一个用户
app/models/micropost.rb

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

我们可以将这种关联用图 2.15 所示的图形表示出来。因为 `microposts` 表中有 `user_id` 这一列，所以 Rails（通过 Active Record）就可以将微博和每个用户关联起来。

图 2.15：微博和用户之间的关联

在第 10 章和第 11 章中，我们将使用用户和微博之间的关联来显示某一个用户的所有微博，并且生成一个和 Twitter 类似的动态列表。我们可以使用控制台（console）来检查一下用户与微博之间关联的实现，控制台是和 Rails 应用程序交互很有用的工具。在命令行中执行 `rails console` 来启动控制台，然后使用 `User.first` 从数据库中读取第一个用户（并将读取的数据赋值给 `first_user` 变量）：[7]

```
$ rails console
>> first_user = User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2013-03-06 02:01:31", updated_at: "2013-03-06 02:01:31">
>> first_user.microposts
=> [#<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2013-03-06 02:37:37", updated_at: "2013-03-06 02:37:37">, #<Micropost id: 2,
content: "Second micropost", user_id: 1, created_at: "2013-03-06 02:38:54",
updated_at: "2013-03-06 02:38:54">]
>> exit
```

（上面代码中我包含了最后一行用来演示如何退出控制台，在大多数系统中也可以使用 Ctrl-d 组合键。）然后使用 `first_user.microposts` 获取用户的微博：Active Record 会自动返回 `user_id` 和 `first_user` 的 id 相同的（1）所有微博。我们将在第 10 章和第 11 章更详细的学习 Active Record 中这种关联的实现。

2.3.4 继承关系

接下来我们暂时结束演示程序的讨论，来简单的介绍一下 Rails 中控制器和模型的类继承。如果你有一些面向对象编程（Object-oriented Programming，OOP）的经验将更好的理解这些内容，如果你未接触过 OOP的话可以选择跳过本小节。一般来说，如果你不熟悉类的概念（4.4 节中会介绍），我建议你稍晚些时候再回过头来看本小节。

我们先介绍模型的继承关系。对比一下代码 2.12 和代码 2.13 中的代码，User 模型和 Micropost 模型都继承自（通过 `<`）`ActiveRecord::Base`，它是 ActiveRecord 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` 我们的模型对象才能够和数据库通讯、将数据库中的列看做 Ruby 中的属性等。

代码 2.12：`User` 类，包括继承关系
`app/models/user.rb`

7. 你的控制台可能会显示类似 `ruby-2.0.0-head >` 的开头，示例中使用 `>>` 替代，因为不同的 Ruby 版本会有所不同。

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

代码 2.13： `Micropost` 类，包括继承关系
`app/models/micropost.rb`

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

图 2.16： User 模型和 Micropost 模型的继承关系

控制器的继承关系更复杂一些。对比一下代码 2.14 和代码 2.15，我们可以看到 Users 控制器和 Microposts 控制器都继承自应用程序的控制器（`ApplicationController`）。如代码 2.16 所示，`ApplicationController` 继承自 `ActionController::Base`，它是 Rails 中的 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。

图 2.17： Users 控制器和 Microposts 控制器的继承关系

代码 2.14： `UsersController` 类，包含继承关系
`app/controllers/users_controller.rb`

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

代码 2.15：MicropostsController 类，包含继承关系
app/controllers/microposts_controller.rb

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

代码 2.16：ApplicationController 类，包含继承关系
app/controllers/application_controller.rb

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

和模型的继承类似，通过继承 ActionController::Base，Users 控制器和 Microposts 控制器获得了很多的功能，例如处理模型对象的功能，过滤输入的 HTTP 请求，以及将视图渲染成 HTML 的功能。因为 Rails 中的控制器都继承自 ApplicationController，所以在应用程序控制器中定义的内容就会应用到程序中的所有动作。例如，在 8.2.1 节中将看到如何在应用程序控制器中添加一个登录、退出的帮助方法。

2.3.5 部署演示程序

完成 Microposts 资源之后，是时候将代码推送到 GitHub 的仓库中了：

```
$ git add .
$ git commit -m "Finish demo 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的提交也可以。

然后，你也可以按照 1.4 节中介绍的方法将演示程序部署到 Heroku：

```
$ heroku create
$ rake assets:precompile
$ git add .
$ git commit -m "Add precompiled assets for Heroku"
$ git push heroku master
```

最后，迁移生产环境中的数据库：

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

上面的代码会用 User 和 Micropost 数据模型更新 Heroku 上的数据库。

2.4 小结

现在我们已经结束了对一个 Rails 应用程序的分析，本章中开发的演示程序有一些好的地方也有一些有缺陷的地方。

好的地方

- 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 程序

有缺陷的地方

- 没有自定义布局和样式
- 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
- 没实现用户和微博的自动关联
- 没实现“关注”和“被关注”功能
- 没实现动态列表
- 没使用 TDD
- 没有真的理解所做的事情

本书后续的内容会建立在这些好的部分之上，然后改善有缺陷的部分。

第 3 章 基本静态的页面

从本章开始我们要开发一个大型的示例程序，本书后续内容都会基于这个示例程序。最终完成的程序会包含用户、微博功能，以及完整的登录和用户身份验证系统，不过我们会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，对这个初建的程序而言也是个很好的开端。

虽然 Rails 是被设计用来开发基于数据库的动态网站的，不过它也能胜任使用纯 HTML 创建的静态页面。其实，使用 Rails 创建动态页面还有一点好处：我们可以方便的添加一小部分动态内容。这一章就会教你怎么做。在这个过程中我们还会一窥自动化测试（automated testing）的面目，自动化测试可以让我们确信自己编写的代码是正确的。而且，编写一个好的测试用例还可以让我们信心十足的重构（refactor）代码，修改实现过程但不影响最终效果。

本章有很多的代码，特别是在 3.2 节和3.3 节，如果你是 Ruby 初学者先不用担心没有理解这些代码。就像在 1.1.1 节中说过的，你可以直接复制粘贴测试代码，用来验证程序中代码的正确性而不用担心其工作原理。第 4 章会更详细的介绍 Ruby，你有的是机会来理解这些代码。还有 RSpec 测试，它在本书中会被反复使用，如果你现在有点卡住了，我建议你硬着头皮往下看，几章过后你就会惊奇地发现，原本看起来很费解的代码已经变得很容易理解了。（你还可以考虑学习 Code School 的 RSpec 课程，有位读者说这个课程解惑了很多 RSpec 疑问。）

类似第二章，在开始之前我们要先创建一个新的 Rails 项目，这里我们叫它 `sample_app`：

```
$ cd ~/rails_projects
$ rails new sample_app --skip-test-unit
$ cd sample_app
```

上面代码中传递给 `rails` 命令的 `--skip-test-unit` 选项的意思是让 Rails 不生成默认使用的 `Test::Unit` 测试框架对应的 `test` 文件夹。这样做并不是说我们不用写测试，而是从 3.2 节开始我们会使用另一个测试框架 RSpec 来写整个的测试用例。

类似 2.1 节，接下来我们要用文本编辑器打开并编辑 `Gemfile`，写入程序所需的 gem。这个示例程序会用到之前没用过的两个 gem：RSpec 所需的 gem 和针对 Rails 的 RSPec 库 gem。代码 3.1 所示的代码会包含这些 gem。（注意：如果此时你想安装这个示例程序用到的所有 gem，你应该使用代码 9.47 中的代码。）

代码 3.1：示例程序的 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'

group :development, :test do
  gem 'sqlite3', '1.3.8'
  gem 'rspec-rails', '2.13.1'
end
```

```
group :test do
  gem 'selenium-webdriver', '2.35.1'
  gem 'capybara', '2.1.0'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'

group :doc do
  gem 'sdoc', '0.3.20', require: false
end

group :production do
  gem 'pg', '0.15.1'
end
```

上面的代码将 `rspec-rails` 放在了开发组中，这样我们就可以使用 RSpec 相关的生成器了，同样我们还把它放到了测试组中，这样才能在测试时使用。我们没必要单独的安装 RSpec，因为它是 rspec-rails 的依赖件（dependency），会被自动安装。我们还加入了 Capybara，这个 gem 允许我们使用类似英语中的句法编写模拟与应用程序交互的代码，还有 Capybara 的一个依赖库 Selenium。[1] 和第 2 章一样，我们还要把 PostgreSQL 所需的 gem 加入生产组，这样才能部署到 Heroku：

```
group :production do
  gem 'pg', '0.15.1'
end
```

Heroku 推荐在开发环境和生产环境使用相同的数据库，不过对我们的示例程序而言没什么影响，SQLite 比 PostgreSQL 更容易安装和配置。在你的电脑中安装和配置 PostgreSQL 会作为一个练习。（参见 3.5 节）

要安装和包含这些新加的 gem，请运行 `bundle update` 和 `bundle install`：

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

与 1.4.1 节 和第 2 章一样，我们使用 `-without production` 禁止安装生产环境所需的 gem。这个选项会被记住，所以后续调用 Bundler 就不用再指定这个选项，直接运行 `bundle install` 就可以自动不安装生产环境所需的 gem。[2]

1. Webrat 的继任者，Capybara 是以世界上最大的啮齿类动物命名的。

因为我们会公开分享这个示例程序的源码，所以一定要修改“安全权标”。Rails 使用安全权标来加密会话。我们要把硬编码的字符串改为动态生成的（如代码 3.2）。（代码 3.2 中的代码现在看来是很高级的，但这是一个很严重的安全隐患，所以我觉得有必要尽早加入这段代码。）请确保使用了代码 1.7 所示的 `.gitignore` 文件，不把 `.secret` 纳入仓库。

代码 3.2： 动态生成安全权标
`config/initializers/secret_token.rb`

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

# Your secret key is used for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!

# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
# You can use `rake secret` to generate a secure secret key.

# Make sure your secret_key_base is kept private
# if you're sharing your code publicly.
require 'securerandom'

def secure_token
  token_file = Rails.root.join('.secret')
  if File.exist?(token_file)
    # Use the existing token.
    File.read(token_file).chomp
  else
    # Generate a new token and store it in token_file.
    token = SecureRandom.hex(64)
    File.write(token_file, token)
    token
  end
end

SampleApp::Application.config.secret_key_base = secure_token
```

接着我们要设置一下让 Rails 使用 RSpec 而不用 `Test::Unit`。这个设置可以通过 `rails generate rspec:install` 命令实现：

```
$ rails generate rspec:install
```

如果系统提示缺少 JavaScript 运行时，你可以访问 execjs 在 GitHub 的页面查看可以使用的运行时。 我一般都建议安装 Node.js。

然后剩下的就是初始化 Git 仓库了：[3]

2. 事实上你可以省略 `install`。单独的 `bundle` 就是 `bundle install` 的别名。

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

和第一个程序一样，我建议你更新一下 `README` 文件，更好的描述这个程序，还可以提供一些帮助信息，可参照代码 3.3。

代码 3.3：示例程序改善后的 `README` 文件

```
# Ruby on Rails Tutorial: sample application

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

然后添加 `.md` 后缀将其更改为 Markdown 格式，再提交所做的修改：

```
$ git mv README.rdoc README.md
$ git commit -am "Improve the README"
```

你可能还记得，在 1.3.5 节中，我们使用了 `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`。如上面第二个命令所示，可以把这两个旗标和在一起，使用 `git commit -am "Message"` 命令。

这个程序在本书的后续章节会一直使用，所以建议你在 GitHub 新建一个仓库，然后将代码推送上去：

```
$ git remote add origin https://github.com/<username>/sample_app.git
$ git push -u origin master
```

我自己也做了这一步，你可以在 GitHub 上找到这个示例程序的代码。（我用了一个稍微不同的名字）[4]

当然我们也可以选择在这个早期阶段将程序部署到 Heroku：

```
$ heroku create
$ rake assets:precompile
$ git add .
$ git commit -m "Add precompiled assets for Heroku"
$ git push heroku master
$ heroku run rake db:migrate
```

在阅读本书的过程中，我建议你经常地推送并部署这个程序：

3. 和之前一样，使用代码 1.7 的内容对你的系统可能更有用。
4. https://github.com/railstutorial/sample_app_rails_4

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

这样你可在远端做个备份，也可以尽早的获知生成环境中出现的错误。如果你在 Heroku 遇到了问题，可以看一下生产环境的日志文件尝试解决：

```
$ heroku logs
```

确保你做了代码 3.3 所示的设置。

所有的准备工作都结束了，下面要开始开发这个示例程序了。

3.1 静态页面

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。[^1-5]Rails 动作放在控制器（MVC 中的 C，参见 1.2.6 节）中，控制器中包含的动作都是为了实现同一类功能的。第 2 章 已经简要介绍了控制器，当更完整的熟悉 REST 架构后（从第 6 章开始）会更深入的理解控制器，本质上，控制器就是一组网页（可能是动态的）的容器。

现在回想一下 1.2.3 节 中讲过的 Rails 目录结构（图 1.2）会对我们有点帮助。本节主要的工作都在 `app/controllers` 和 `app/views` 文件夹中。（3.2 节中我们还会新建一个文件夹。）我建议你现在在文本编辑器或 IDE 中打开示例程序（参见旁注 3.1。）。

旁注 3.1：打开项目

在文本编辑器或 IDE 中打开整个 Rails 目录会带来很多便利。不过怎么做却取决于你的系统，大多数情况下你可以在命令行中用你选择的浏览器命令打开当前应用程序所在的目录，在 Unix 中当前目录就是一个点号（`.`）：

```
$ cd ~/rails_projects/sample_app
$ <editor name> .
```

例如，用 Sublime Text 打开示例程序，你可以输入：

```
$ subl .
```

对于 Vim 来说，输入

```
$ vim .
```

（你可以根据你使用的 Vim 变种，把 `vim` 替换成 `gvim` 或 `mvim`。）

如 1.3.5 节 所说，在开始之前，如果使用 Git，最好别在主分支上开发，要新建一个从分支。如果你正使用 Git 做版本控制，应该执行下面的命令：

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

Rails 提供了一个脚本用来创建控制器，叫做 `generate`，只要提供控制器的名字就可以运行了。既然我们要创建的控制器是处理静态页面的，那就叫它 StaticPages 吧。我们计划要生成处理“首页”，“帮助”页面和“关于”页面的动作。`generate` 命令可以接受一个可选的参数，指定要生成的动作，现在我们只提供两个动作，如代码 3.4 所示。

代码 3.4： 生成 StaticPages 控制器

```
$ rails generate controller StaticPages home help --no-test-framework
      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  helper
      create    app/helpers/static_pages_helper.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/static_pages.js.coffee
      invoke    scss
      create      app/assets/stylesheets/static_pages.css.scss
```

我们使用了 `--no-test-framework` 选项禁止生成 RSpec 测试代码，因为我们不想自动生成，在 3.2 节会手动创建测试。同时我们还故意从命令行参数中省去了 `about` 动作，稍后我们会看到如何通过 TDD 添加它（3.2 节）。

注意，在代码 3.4 中，传入命令的控制器名使用的是驼峰式命名法，生成的控制器文件名使用的是蛇底式命名法，因此名为 StaticPages 的控制器对应的文件名为 `static_pages_controller.rb`。这只是一个约定，其实在命令行中使用蛇底式也可以，如下的命令

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

同样会生成 `static_pages_controller.rb` 控制器文件。因为 Ruby 开发者使用驼峰式命名类名（参见 4.4 节），所以我喜欢用驼峰式引用控制器，当然这是个人喜好的问题。（因为 Ruby 文件习惯使用蛇底式名称，所以 Rails 生成器会调用 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 3.2 中介绍了一些如何在 Rails 中撤销操作的方法。

旁注 3.2：撤销操作

即使再小心，在开发 Rails 应用程序过程中仍然可能犯错。幸运的是，Rails 提供了一些工具能够帮助你进行复原。

举例来说，一个常见的情况是，你想更改控制器的名字，这时你就要撤销生成的代码。生成控制器时，除了控制器文件本身之外，Rails 还会生成很多其他的文件（参见代码 3.4）。撤销生成的文件不仅仅要删除主要的文件，还要删除一些辅助的文件。（事实上，我们还要撤销对 `routes.rb` 文件自动做的一些改动。）在 Rails 中，我们可以通过 `rails destroy` 命令完成这些操作。一般来说，下面的两个命令是相互抵消的：

```
$ rails generate controller FooBars baz quux
$ rails destroy  controller FooBars baz quux
```

同样的，在第 6 章中会使用下面的命令生成模型：

```
$ rails generate model Foo bar:string baz:integer
```

生成的模型可通过下面的命令撤销：

```
$ rails destroy model Foo
```

（对模型来说我们可以省略命令行中其余的参数。当阅读到第六章时，看看你能否发现为什么可以这么做。）

对模型来说涉及到的另一个技术是撤销迁移。第 2 章已经简要的介绍了迁移，第 6 章开始会更深入的介绍。迁移通过下面的命令改变数据库的状态：

```
$ rake db:migrate
```

我们可以使用下面的命令撤销一个迁移操作：

```
$ rake db:rollback
```

如果要回到最开始的状态，可以使用：

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

你可能已经猜到了，将数字 0 换成其他的数字就会回到相应的版本状态，这些版本数字是按照迁移顺序排序的。

拥有这些技术，我们就可以得心的应对开发过程中遇到的各种混乱（snafu）了。

代码 3.4 中生成 StaticPages 控制器的命令会自动更新路由文件（route），叫做 `config/routes.rb`，Rails 会通过这个文件寻找 URL 和网页之间的对应关系。这是我们第一次讲到 `config` 目录，所以让我们看一下该目录的结构吧（如图 3.1）。`config` 目录如其名字所示，是存储 Rails 应用程序中的设置文件的。

图 3.1：示例程序的 `config` 文件夹

因为我们生成了 `home` 和 `help` 动作，路由文件中已经为它们生成了配置，如代码 3.5。

代码 3.5：StaticPages 控制器中 `home` 和 `help` 动作的路由配置
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  .
  .
  .
end
```

如下的规则

```
get "static_pages/home"
```

将来自 /static_pages/home 的请求映射到 StaticPages 控制器的 `home` 动作上。另外，当使用 `get` 时会将其对应到 GET 请求方法上，GET 是 HTTP（超文本传输协议，Hypertext Transfer Protocol）支持的基本方法之一（参见旁注

3.3）。在我们这个例子中，当我们在 StaticPages 控制器中生成 `home` 动作时，就自动的在 /static_pages/home 地址上获得了一个页面。访问 /static_pages/home 可以查看这个页面（如图 3.2）。

旁注 3.3：GET 等

超文本传输协议（HTTP）定义了几个基本操作，GET、POST、PATCH 和 DELETE。这四个动词表现了客户端电脑（通常会运行一个浏览器，例如 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。（有一点很重要需要你知道，当在本地电脑上开发 Rails 应用程序时，客户端和服务器是在同一个物理设备上的，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，用来从网络上读取数据，它的意思是“读取一个网页”，当你访问 google.com 或 wikipedia.org 时，你的浏览器发出的就是 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 应用程序中，POST 请求一般被用来创建某个东西（不过 HTTP 也允许 POST 进行更新操作）。例如，你提交注册表单时发送的 POST 请求就会在网站中创建一个新用户。剩下的两个动词，PATCH 和 DELETE 分别用来更新和销毁服务器上的某个东西。这两个操作比 GET 和 POST 少用一些，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，看起来就像是浏览器发出的一样。

顺便说一下，在 Rails 之前的版本中，没有使用 PATCH，用的是 PUT，Rails 4 仍然支持 PUT，但 PATCH 能更好的表明 HTTP 请求的意图，所以最好在新版中使用 PATCH。

图 3.2： 简陋的“首页”视图（/static_pages/home）

要想弄明白这个页面是怎么来的，让我们在浏览器中看一下 StaticPages 控制器文件吧，你应该会看到类似代码 3.6 的内容。你可能已经注意到了，不像第 2 章中的 Users 和 Microposts 控制器，StaticPages 控制器没有使用标准的 REST 动作。这对静态页面来说是很常见的，REST 架构并不能解决所有的问题。

代码 3.6： 代码 3.4 生成的 StaticPages 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` 可以看到 `static_pages_controller.rb` 文件定义了一个类（class），叫做 `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。尖括号 `<` 说明 `StaticPagesController` 是继承自 Rails 的 `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 节中更详细的介绍类和继承。）

在本例中，StaticPages 控制器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，因为它继承自 `ApplicationController`，它的方法对 Rails 来说就有特殊的意义了：访问 /static_pages/home 时，Rails 在 StaticPages 控制器中寻找 home 动作，然后执行该动作，再渲染相应的视图（1.2.6 节中介绍的 MVC 中的 V）。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 3.4 的输出，或许你能猜到动作和视图之间的对应关系：home 动作对应的视图叫做 `home.html.erb`。3.3 节将告诉你 `.erb` 是什么意思。看到 `.html` 你或许就不会奇怪了，它基本上就是 HTML（代码 3.7）。

代码 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` 动作的视图代码类似（参见代码 3.8）。

代码 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>
```

这两个视图只是占位用的，它们的内容都包含了一个一级标题（`h1` 标签）和一个显示视图文件完整的相对路径的段落（`p` 标签）。我们会在 3.3 节中添加一些简单的动态内容。这些静态内容的存在是为了强调一个很重要的事情：Rails 的视图可以只包含静态的 HTML。

在本章剩下的内容中，我们会为“首页”和“帮助”页面添加一些内容，然后补上 3.1 节中丢下的“关于”页面。然后会添加少量的动态内容，在每个页面显示不同的标题。

在继续下面的内容之前，如果你使用 Git 的话最好将 StaticPages 控制器相关的文件加入仓库：

```
$ git add .
$ git commit -m "Add a StaticPages controller"
```

3.2 第一个测试

本书采用了一种直观的测试应用程序表现的方法，而不关注具体的实现过程，这是 TDD 的一个变种，叫做 BDD（行为驱动开发，Behavior-driven Development）。我们使用的主要工具是集成测试（integration test）和单元测试(unit test)。集成测试在 RSpec 中叫做 request spec，它允许我们模拟用户在浏览器中和应用程序进行交互的操作。

和 Capybara 提供的自然语言句法（natural-language syntax）一起使用，集成测试提供了一种强大的方法来测试应用程序的功能，而不用在浏览器中手动检查每个页面。（BDD 另外一个受欢迎的选择是 Cucumber，在 8.3 节中会介绍。）

TDD 的好处在于测试优先，比编写应用程序的代码还早。刚接触的话要花一段时间才能适应这种方式，不过好处很明显。我们先写一个失败测试（failing test），然后编写代码使这个测试通过，这样我们就会相信测试真的是针对我们设想的功能。这种“失败-实现-通过”的开发循环包含了一个心流，可以提高编程的乐趣并提高效率。测试还扮演着应用程序代码客户的角色，会提高软件设计的优雅性。

关于 TDD 有一点很重要需要你知道，它不是万用良药，没必要固执的认为总是要先写测试、测试要囊括程序所有的功能、所有情况都要写测试。例如，当你不确定如何处理某些编程问题时，通常推荐你跳过测试先编写代码看一下解决方法能否解决问题。（在极限编程中，这个过程叫做“探针实验（spike）”。）一旦看到了解决问题的曙光，你就可以使用 TDD 实现一个更完美的版本。

本节我们会使用 RSpec 提供的 `rspec` 命令运行测试。初看起来这样做是理所当然的，不过却不完美，如果你是个高级用户我建议你按照 3.6 节的内容设置一下你的系统。

3.2.1 测试驱动开发

在测试驱动开发中，我们先写一个会失败的测试，在很多测试工具中会将其显示为红色。然后编写代码让测试通过，显示为绿色。最后，如果需要的话，我们还会重构代码，改变实现的方式（例如消除代码重复）但不改变功能。这样的开发过程叫做“遇红，变绿，重构（Red, Green, Refactor）”。

我们先来使用 TDD 为“首页”增加一些内容，一个内容为 `Sample App` 的顶级标题（`<h1>`）。第一步要做的是为这些静态页面生成集成测试（request spec）：

```
$ rails generate integration_test static_pages
      invoke  rspec
      create    spec/requests/static_pages_spec.rb
```

上面的代码会在 `spec/requests` 文件夹中生成 `static_pages_spec.rb` 文件。自动生成的代码不能满足我们的需求，用文本编辑器打开 `static_pages_spec.rb`，将其内容替换成代码 3.9 所示的代码。

代码 3.9： 测试“首页”内容的代码
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end
```

```
  end
end
```

代码 3.9 是纯粹的 Ruby，不过即使你以前学习过 Ruby 也看不太懂，这是因为 RSpec 利用了 Ruby 语言的延展性定义了一套“领域专属语言”（Domain-specific Language, DSL）用来写测试代码。重要的是，如果你想使用 RSpec 不是一定要知道 RSpec 的句法。初看起来是有些神奇，RSpec 和 Capybara 就是这样设计的，读起来很像英语，如果你多看本书中的示例，很快你就会熟练了。

代码 3.9 包含了一个 `describe` 块以及其中的一个测试用例（sample），以 `it "..." do` 开头的代码块就是一个用例：

```
describe "Home page" do

  it "should have the content 'Sample App'" do
    visit '/static_pages/home'
    expect(page).to have_content('Sample App')
  end
end
```

第一行代码指明我们描绘的是“首页”，内容就是一个字符串，如果需要你可以使用任何的字符串，RSpec 不做强制要求，不过你以及其他的人类读者或许会关心你用的字符串。然后测试说，如果你访问地址为 `/static_pages/home` 的“首页”时，其内容应该包含“Sample App”这两个词。和第一行一样，这个双引号中的内容 RSpec 没做要求，只要能为人类读者提供足够的信息就行了。下面这一行：

```
visit '/static_pages/home'
```

使用了 Capybara 中的 `visit` 函数来模拟在浏览器中访问 `/static_pages/home` 的操作。下面这一行：

```
expect(page).to have_content('Sample App')
```

使用了 `page` 变量（同样由 Capybara 提供）来测试页面中是否包含了正确的内容。

```
3. ~/rails_projects/sample_app (bash)
[sample_app (static-pages)]$ bundle exec rspec spec/
F

Failures:

  1) Static pages Home page should have the content 'Sample App'
     Failure/Error: expect(page).to have_content('Sample App')
       expected there to be text "Sample App" in "StaticPages#home Find me in ap
p/views/static_pages/home.html.erb"
     # ./spec/features/static_pages_spec.rb:9:in `block (3 levels) in <top (requ
ired)>'

Finished in 0.25114 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/features/static_pages_spec.rb:7 # Static pages Home page should hav
e the content 'Sample App'

Randomized with seed 35298

[sample_app (static-pages)]$
```

图 3.3：红色（失败）的测试

若要测试正确运行，我们要在 `spec_helper.rb` 中加入一行代码，如代码 3.10 所示。（在本书后续的草稿中，我计划不再使用这行代码，启用功能测试中的新技术。）

代码 3.10：把 Capybara DSL 加入 RSpec 帮助文件
`spec/spec_helper.rb`

```
# This file is copied to spec/ when you run 'rails generate rspec:install'
.
.
.
RSpec.configure do |config|
  .
  .
  .
  config.include Capybara::DSL
end
```

我们有很多种方式来运行测试代码，3.6 节中还提供了一些便利且高级的方法。现在，我们在命令行中执行 `rspec` 命令（前面会加上 `bundle exec` 来保证 RSpec 运行在 `Gemfile` 指定的环境中）：[5]

5. 每次都要输入 `bundle exec` 显然很麻烦，参考 3.6 节中介绍的方法来避免输入它。

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

上述命令会输出一个失败测试。失败测试的具体样子取决于你的系统，在我的系统中它是红色的，如图 3.3。[6]

要想让测试通过，我们要用代码 3.11 中的 HTML 替换掉默认的“首页”内容。

代码 3.11：让“首页”测试通过的代码
`app/views/static_pages/home.html.erb`

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

这段代码中一级标题（`<h1>`）的内容是 `Sample App` 了，会让测试通过。我们还加了一个锚记标签 `<a>`，链接到一个给定的地址（在锚记标签中地址由“href”（hypertext reference）指定）：

```
<a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
```

现在再运行测试看一下结果：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

在我的系统中，通过的测试显示如图 3.4 所示。

基于上面针对“首页”的例子，或许你已经猜到了“帮助”页面类似的测试和程序代码。我们先来测试一下相应的内容，现在字符串变成“`Help`”了（参见代码 3.12）。

代码 3.12：添加测试“帮助”页面内容的代码
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end
  end

  describe "Help page" do
```

6. 实际上我的终端和编辑器背景都是暗色调的，在截图中使用亮色效果更好。

```
    it "should have the content 'Help'" do
      visit '/static_pages/help'
      expect(page).to have_content('Help')
    end
  end
end
```

图 3.4：绿色（通过）的测试

然后运行测试：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

有一个测试会失败。（因为系统的不同，而且统计每个阶段的测试数量很难，从现在开始我就不会再截图 RSpec 的输出结果了。）

程序所需的代码和代码 3.11 类似，如代码 3.13 所示。

代码 3.13：让“帮助”页面的测试通过的代码
app/views/static_pages/help.html.erb

```
<h1>Help</h1>
<p>
```

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

现在测试应该可以通过了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

3.2.2 添加页面

看过了上面简单的 TDD 开发过程，下面我们要用这个技术完成一个稍微复杂一些的任务，添加一个新页面，就是 3.1 节中没有生成的“关于”页面。通过每一步中编写测试和运行 RSpec 的过程，我们会看到 TDD 是如何引导我们进行应用程序开发的。

遇红

先来到“遇红-变绿”过程中的“遇红”部分，为“关于”页面写一个失败测试。参照代码 3.12 的代码，或许你已经知道如何写这个测试了（参见代码 3.14）。

代码 3.14： 添加测试“关于”页面内容的代码
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end
  end

  describe "Help page" do

    it "should have the content 'Help'" do
      visit '/static_pages/help'
      expect(page).to have_content('Help')
    end
  end

  describe "About page" do
```

```
    it "should have the content 'About Us'" do
      visit '/static_pages/about'
      expect(page).to have_content('About Us')
    end
  end
end
```

变绿

回顾一下 3.1 节的内容，在 Rails 中我们可以通过创建一个动作并添加相应的视图文件来生成静态页面。所以首先我们要在 StaticPages 控制器中添加一个 about 动作。我们已经写过失败测试了，现在已经确信，如果能通过，就创建了一个可以运行的“关于”页面。

如果你运行 RSpec 测试：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

输出的结果会提示下面的错误：

```
No route matches [GET] "/static_pages/about"
```

这提醒我们要在路由文件中添加 static_pages/about，我们可以按照代码 3.5 所示的格式添加，结果如代码 3.15 所示。

代码 3.15：添加“关于”页面的路由
config/routes.rb

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  get "static_pages/about"
  .
  .
  .
end
```

现在运行测试

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

将提示如下错误

```
The action 'about' could not be found for StaticPagesController
```

为了解决这个问题，我们按照代码 3.6 中 home 和 help 的格式在 StaticPages 控制器中添加 about 动作的代码（如代码 3.16 所示）。

代码 3.16：添加了 about 动作的 StaticPages 控制器
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

再运行测试

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

会提示缺少模板（template，例如一个视图）：

```
Missing template static_pages/about
```

要解决这个问题，我们要添加 about 动作对应的视图。我们需要在 app/views/static_pages 目录下创建一个名为 about.html.erb 的新文件，写入代码 3.17 所示的内容。

代码 3.17：“关于”页面视图代码
app/views/static_pages/about.html.erb

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

再运行 RSpec 就应该“变绿”了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

当然，在浏览器中查看一下这个页面来确保测试没有失效也是个不错的主意。（如图 3.5）

重构

现在测试已经变绿了，我们可以很自信的尽情重构了。我们的代码经常会“变味”（意思是代码会变得丑陋、啰嗦、大量的重复），电脑不会在意，但是人类会，所以经常重构让代码变得简洁是很重要的。这时候一个好的测试就显出其价值了，因为它可以降低重构过程中引入 bug 的风险。

我们的示例程序现在还很小没什么可重构的，不过代码无时无刻不在变味，所以我们的重构也不会等很久：在 3.3.4 节中就要忙于重构了。

3.3 有点动态内容的页面

到目前为止，我们已经为一些静态页面创建了动作和视图，我们还改变了每一个页面显示的内容（标题）让它看起来是动态的。改变标题到底算不算真正动态还有争议，不过前面的内容却可以为第 7 章介绍的真正动态打下基础。

如果你跳过了 3.2 节中的 TDD 部分，在继续阅读之前请先按照代码 3.15、代码 3.16 和代码 3.17 创建“关于”页面。

图 **3.5**：新添加的“关于”页面（/static_pages/about）

3.3.1 测试标题的变化

我们计划修改“首页”、“帮助”页面和“关于”页面的标题，在每一页都有所变化。这个过程将使用视图中的 `<title>` 标签。大多数浏览器会在浏览器窗口的顶部显示标题的内容（Google Chrome 是个特例），标题对搜索引擎优化也是很重要的。我们会先写测试标题的代码，然后添加标题，最后使用布局（layout）文件进行重构，削除重复。

你可能已经注意到了，`rails new` 命令已经创建了布局文件。稍后我们会介绍这个文件的作用，现在在继续之前先将其重命名：

```
$ mv app/views/layouts/application.html.erb foobar    # 临时修改
```

（`mv` 是 Unix 命令，在 Windows 中你可以在文件浏览器中重命名或者使用 `rename` 命令。）在真正的应用程序中你不需要这么做，不过没有了这个文件之后你就能更容易理解它的作用。

本节结束后，三个静态页面的标题都会是“Ruby on Rails Tutorial Sample App | Home”这种形式，标题的后面一部分会根据所在的页面而有所不同（参见表格 3.1）。我们接着代码 3.14 中的测试，添加代码 3.18 所示测试标题的代码。

代码 3.18：标题测试

```
it "should have the right title" do
  visit '/static_pages/home'
  expect(page).to have_title("Ruby on Rails Tutorial Sample App | Home")
end
```

`have_title` 方法会测试一个 HTML 页面标题中是否含有指定的内容。换句话说，下面的代码：

```
expect(page).to have_title("Ruby on Rails Tutorial Sample App | Home")
```

会检查 `title` 标签的内容是否为

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

要注意一下，检查的内容不一定要完全匹配，任何的子字符串都可以，所以

```
expect(page).to have_title("Home")
```

也会匹配完整形式的标题。

表格 3.1：示例程序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"

我们按照代码 3.18 的格式为三个静态页面都加上测试代码，结果参照代码 3.19。注意，测试中有很多重复，我们会在 5.3.4 节重构。

代码 3.19： StaticPages 控制器的测试文件，包含标题测试
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end

    it "should have the title 'Home'" do
      visit '/static_pages/home'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | Home")
    end
  end

  describe "Help page" do

    it "should have the content 'Help'" do
      visit '/static_pages/help'
      expect(page).to have_content('Help')
    end

    it "should have the title 'Help'" do
      visit '/static_pages/help'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | Help")
    end
  end

  describe "About page" do

    it "should have the content 'About Us'" do
      visit '/static_pages/about'
      expect(page).to have_content('About Us')
    end

    it "should have the title 'About Us'" do
      visit '/static_pages/about'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | About Us")
```

```
    end
  end
end
```

现在已经有了如代码 3.19 所示的测试，你应该运行

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

确保得到的结果是红色的（失败的测试）。

3.3.2 让标题测试通过

现在我们要让标题测试通过，同时我们还要完善 HTML，让它通过验证。符合最新标准的页面结构是下面这种形式的：

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

这种结构的最顶部包含一个文档类型声明（document type declaration，简称 doctype），告知浏览器所用的 HTML 版本（本例使用 HTML5）[7]。随后是 `head` 部分，包含一个 `title` 标签，其内容为“Greeting”。然后是 `body` 部分，包含一个 `p` 标签（表示段落），其内容为“Hello, world!”。（文件内容的缩进是可选的，HTML 不会特别对待空白，Tab 和空格都会被忽略，但缩进可以让文档更易阅读。）

在“首页”中使用这种基本结构后得到的文档如代码 3.20 所示。

代码 3.20：“首页”的完整 HTML
`app/views/static_pages/home.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | Home</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
```

7. HTML 一直在变化，显式声明一个 doctype 可以确保浏览器在未来还可以正确的解析页面。`<!DOCTYPE html>` 这种极为简单的格式是最新的 HTML 标准 HTML5 的一个特色。

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

代码 3.20 使用了代码 3.19 中测试用到的标题：

```
<title>Ruby on Rails Tutorial Sample App | Home</title>
```

所以，“首页”的测试现在应该可以通过了。你还会看到红色的错误提示是因为“帮助”页面和“关于”页面的测试还是失败的，我们使用代码 3.21 和代码 3.22 中的代码让它们也通过测试。

代码 3.21：“帮助”页面的完整 HTML
app/views/static_pages/help.html.erb

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

代码 3.22：“关于”页面的完整 HTML
app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | About Us</title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      is a project to make a book and screencasts 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>
```

3.3.3 嵌入式 Ruby

本节到目前为止已经做了很多事情，我们通过 Rails 控制器和动作生成了三个可以通过句法验证的页面，不过这些页面都是纯静态的 HTML，没有体现出 Rails 的强大所在。而且，它们的代码充斥着重复：

- 页面的标签几乎（但不完全）是一模一样的
- 每个标题中都有“Ruby on Rails Tutorial Sample App”
- HTML 结构在每个页面都重复的出现了

代码重复的问题违反了很重要的“不要自我重复”（Don’t Repeat Yourself, DRY）原则，本小节和下一小节将按照 DRY 原则去掉重复的代码。

不过我们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。这样我们就可以更容易的去掉重复的代码了。

这个过程会在视图中使用嵌入式 Ruby（Embedded Ruby）。既然“首页”、“帮助”页面和“关于”页面的标题有一个变动的部分，那我们就利用一个 Rails 中特别的函数 `provide` 在每个页面设定不同的标题。通过将视图 `home.html.erb` 标题中的“Home”换成如代码 3.23 所示的代码，我们可以看一下实现的过程。

代码 3.23：标题中使用了嵌入式 Ruby 代码的“首页”视图
`app/views/static_pages/home.html.erb`

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

代码 3.23 中我们第一次使用了嵌入式 Ruby，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使用的主要模板系统。[8]下面的代码

8. 其实还有另外一个受欢迎的模板系统叫 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。

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

通过 `<% ... %>` 调用 Rails 中的 `provide` 函数，然后将字符串 `'Home'` 赋给 `:title`。[9]然后，在标题中，我们使用类似的符号 `<%= ... %>` 通过 Ruby 的 `yield` 函数将标题插入模板中：[10]

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

（这两种嵌入 Ruby 代码的方式区别在于，`<% ... %>` **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。）最终得到的结果和以前是一样的，只不过标题中变动的部分现在是通过 ERb 动态生成的。

我们可以运行 3.3.1 节中的测试来证实一下，测试还是会通过：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

然后我们要对“帮助”页面和“关于”页面做相应的修改了。（参见代码 3.24 和代码 3.25。）

代码 3.24： 标题中使用了嵌入式 Ruby 代码的“帮助”页面视图
`app/views/static_pages/help.html.erb`

```
<% provide(:title, 'Help') %>
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://railstutorial.org/help">Rails Tutorial help page</a>.
      To get help on this sample app, see the
      <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
    </p>
  </body>
</html>
```

代码 3.25： 标题中使用了嵌入式 Ruby 代码的“关于”页面视图
`app/views/static_pages/about.html.erb`

```
<% provide(:title, 'About Us') %>
<!DOCTYPE html>
<html>
```

9. 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 中不能很好的工作。`provide` 函数是替代方法。
10. 如果你学习过 Ruby，可能会猜测 Rails 是将内容拽入区块中的，这样想也是对的。不过使用 Rails 开发应用程序不必知道这一点。

```
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <h1>About Us</h1>
    <p>
      The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      is a project to make a book and screencasts 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>
```

3.3.4 使用布局文件来消除重复

我们已经使用 ERb 将页面标题中变动的部分替换掉了，每一个页面的代码很类似：

```
<% provide(:title, 'Foo') %>
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
  </head>
  <body>
    <!--内容-->
  </body>
</html>
```

换句话说，所有的页面结构都是一致的，包括 `title` 标签中的内容，只有 `body` 标签中的内容有细微的差别。

为了提取出相同的结构，Rails 提供了一个特别的布局文件，叫做 `application.html.erb`，我们在 3.3.1 节中将它重命名了，现在我们再改回来：

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

为了让布局正常的运行，我们要把默认的标题改为前几例代码中使用的嵌入式 Ruby 代码：

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

最终的布局文件如代码 3.26 所示。

代码 3.26：示例程序的网站布局
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
<head>
  <title>Ruby on Rails Tutorial Sample App | <%= 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>
```

注意一下比较特殊的一行

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

这行代码是用来将每一页的内容插入布局中的。没必要了解它的具体实现过程，我们只需要知道，在布局中使用后，当访问 /static_pages/home 时会将 `home.html.erb` 中的内容转换成 HTML 然后插入 `<%= yield %>` 所在的位置。

还要注意一下，默认的 Rails 布局文件包含几行特殊的代码：

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

这些代码会引入应用程序的样式表和 JavaScript 文件（asset pipeline 的一部分）；Rails 中的 `csrf_meta_tags` 方法是用来避免“跨站请求伪造”（cross-site request forgery，CSRF，一种网络攻击）的。

现在代码 3.23、代码 3.24 和代码 3.25 的内容还是和布局文件中类似的 HTML，所以我们要将内容删除，只保留需要的部分。清理后的视图如代码 3.27、代码 3.28 和代码 3.29 所示。

代码 3.27：去除完整的 HTML 结构后的“首页”
`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://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

代码 3.28：去除完整的 HTML 结构后的“帮助”页面
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://railstutorial.org/help">Rails Tutorial help page</a>.
  To get help on this sample app, see the
  <a href="http://railstutorial.org/book">Rails Tutorial book</a>.
</p>
```

代码 3.29：去除完整的 HTML 结构后的“关于”页面
app/views/static_pages/about.html.erb

```
<% provide(:title, 'About Us') %>
<h1>About Us</h1>
<p>
  The <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  is a project to make a book and screencasts 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，但是却没有重复的内容了。运行一下测试看是否还会通过，通过了才能证实重构是成功的：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

3.4 小结

总的来说，本章几乎没有做什么：我们从静态页面开始，最后完成的几乎还是静态的页面。不过从表面来看我们使用了 Rails 中的控制器、动作和视图进行开发工作，现在我们已经可以向我们的网站中添加任意的动态内容了。本教程的后续内容会告诉你怎么添加。

在继续之前，让我们花一点时间提交本章的改动，然后将其合并到主分支中。在 3.1 节中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做提交，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 .
$ git commit -m "Finish static pages"
```

然后利用 1.3.5 节中介绍的技术将变动合并到主分支中：

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

每次完成一些工作后，最好将代码推送到远端的仓库（如果你按照 1.3.4 节中的步骤做了，远端仓库就在 GitHub 上）中：

```
$ git push
```

如果你愿意，现在你还可以将改好的应用程序部署到 Heroku 上：

```
$ git push heroku
```

3.5 练习

1. 为示例程序制作一个“联系”页面。你可以参照代码 3.19，编写一个测试用例检测页面的标题是否为 “Ruby on Rails Tutorial Sample App | Contact”，确定 /static_pages/contact 对应的页面是否存在。将代码 3.30 的内容写入“联系”页面，让测试可以通过。（这个练习会在 5.3 节中解决。）

2. 你可能已经发现 StaticPages 测试文件（代码 3.19）中有重复的地方，“Ruby on Rails Tutorial Sample App”在每个标题测试中都出现了。使用 RSpec 的 `let` 函数，将值赋给变量，确保代码 3.31 中的测试仍然是通过的。代码 3.31 中使用了字符串插值（interpolation），会在 4.2.2 节中介绍。

3. （附加题）Heroku 网站中一篇介绍如何在开发环境中使用 sqlite3 的文章提到，最好在开发环境、测试环境和生产环境中使用相同类型的数据库。按照 Heroku 网站上介绍如何在本地环境中安装 PostgreSQL 的文章内容，在你的电脑上安装 PostgreSQL 数据库。修改 `Gemfile`，删掉 `sqlite3`，换上 `pg`，如代码 3.32 所示。你还要知道如何修改 `config/database.yml` 文件，以及如何在本地运行 PostgreSQL 数据库。（注意，基于安全考虑，应该把 `database.yml` 加入 `.gitignore`，如代码 1.7 所示。）这个练习的目标是使用 PostgreSQL 数据库创建并设置开发环境和测试环境中用到的数据库。我特别推荐使用 Induction 连接并查看本地 PostgreSQL 数据库。**注意：**你会发现本题还是有点难度的，我只推荐高级用户做这一题。如果你在某个地方卡住了，不要坚持不放。前面我已经说过了，本教程开发的示例程序完全兼容 SQLite 和 PostgreSQL。

代码 3.30：“联系”页面的内容
`app/views/static_pages/contact.html.erb`

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

代码 3.31：使用了一个通用标题的 StaticPages 测试文件
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do
```

```
let(:base_title) { "Ruby on Rails Tutorial Sample App" }

describe "Home page" do

  it "should have the content 'Sample App'" do
    visit '/static_pages/home'
    expect(page).to have_content('Sample App')
  end

  it "should have the title 'Home'" do
    visit '/static_pages/home'
    expect(page).to have_title("#{base_title} | Home")
  end
end

describe "Help page" do

  it "should have the content 'Help'" do
    visit '/static_pages/help'
    expect(page).to have_content('Help')
  end

  it "should have the title 'Help'" do
    visit '/static_pages/help'
    expect(page).to have_title("#{base_title} | Help")
  end
end

describe "About page" do

  it "should have the content 'About Us'" do
    visit '/static_pages/about'
    expect(page).to have_content('About Us')
  end

  it "should have the title 'About Us'" do
    visit '/static_pages/about'
    expect(page).to have_title("#{base_title} | About Us")
  end
end

describe "Contact page" do

  it "should have the content 'Contact'" do
    visit '/static_pages/contact'
```

```
      expect(page).to have_content('Contact')
    end

    it "should have the title 'Contact'" do
      visit '/static_pages/contact'
      expect(page).to have_title("#{base_title} | Contact")
    end
  end
end
```

代码 3.32： 删除 SQLite 使用 PostgreSQL 数据库所需的 `Gemfile` 文件

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'
gem 'pg', '0.15.1'

group :development, :test do
  gem 'rspec-rails', '2.13.1'
end

group :test do
  gem 'selenium-webdriver', '2.35.1'
  gem 'capybara', '2.1.0'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'

group :doc do
  gem 'sdoc', '0.3.20', require: false
end
```

3.6 高级技术

3.2 节中曾经提到过，直接使用 `rspec` 命令不是理想的选择。在本节，我们先会介绍一个方法避免输入 `bundle exec`，然后会介绍如何使用 Guard（3.6.2 节）以及可选的 Spork（3.6.3 节）实现自动运行测试的功能。最后我们会介绍一种如何直接在 Sublime Text 中运行测试的方法，这种技术和 Spork 一起使用时特别方便。

本节的内容只针对高级用户，跳过这一节不会影响后面的内容。相比本书其他的部分，这一节的内容可能很快就过时了，所以你不要期盼在你的系统中使用时得到的结果和这里的一样，你可以利用 Google 确保一切都能正常运行。

3.6.1 去掉 bundle exec

3.2.1 节中提到过，一般我们要在 `rake` 或 `rspec` 命令前加上 `bundle exec`，这样才能保证我们运行的程序就是 `Gemfile` 中指定的版本。（基于技术上的原因，`rails` 命令是个例外。）这种做法很啰嗦，本节会介绍两种方法避免这么做。

集成了 Bundler 的 RVM

第一个，也是推荐的方法是使用 RVM，从 V1.11 开始它就集成了 Bundler。你可以运行下面的命令确保自己使用的是 RVM 最新版：

```
$ rvm get stable
$ rvm -v

rvm 1.19.5 (stable)
```

只要版本是 1.11.x 或以上，安装的 gem 就会在特定的 Bundler 环境中执行，所以你就可以直接运行

```
$ rspec spec/
```

而不用加上前面的 `bundle exec`。如果你成功了，那么就可以跳过本小节剩下的内容了。

如果由于某种原因无法使用较新版的 RVM，你还可以通过使用集成 Bundler 所需的 gem配置 RVM 让它在本地环境中自动包含相应的可执行文件，这样也能去掉 `bundle exec`。如果你好奇的话，其实步骤很简单。首先，执行下面的两个命令：

```
$ rvm get head && rvm reload
$ chmod +x $rvm_path/hooks/after_cd_bundler
```

然后执行：

```
$ cd ~/rails_projects/sample_app
$ bundle install --without production --binstubs=./bundler_stubs
```

这些命令会通过某种神秘的力量将 RVM 和 Bundler 组合在一起，确保如 `rake` 和 `rspec` 等命令可以自动的在正确的环境中执行。因为这些文件是针对你本地环境的，你应该将 `bundler_stubs` 文件夹加入 `.gitignore` 文件（参见代码 3.33）。

代码 3.33：把 `bundler_stubs` 加入 `.gitignore` 文件

```
# Ignore bundler config.
/.bundle
```

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

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

# Ignore other unneeded files.
doc/
*.swp
*~
.project
.DS_Store
.idea
bundler_stubs/
```

如果你添加了其他的可执行文件（例如在 3.6.2 节中加入了 `guard`），你要重新运行 `bundle install` 命令：

```
$ bundle install --binstubs=./bundler_stubs
```

binstubs

如果你没使用 RVM 也可以避免输入 `bundle exec`。Bunlder 允许你通过下面的命令生成相关的可执行程序：

```
$ bundle --binstubs
$ bundle config --delete bin && rm -rf bin
$ rake rails:update:bin
```

（事实上，虽然这里用的是不同的目标目录，不过 RVM 也可以使用它。）这个命令会在应用程序中的 `bin/` 文件夹中生成所有必须的可执行文件，所以我们就可以通过下面的方式运行测试了：

```
$ bin/rspec spec/
```

对 `rake` 等来说是一样的：

```
$ bin/rake db:migrate
```

如果你添加了其他的可执行文件（例如 3.6.2 节中的 `guard`），需要重新执行 `bundle --binstubs` 命令。

鉴于某些读者会跳过这一节，本教程的后续内容还是会使用 `bundle exec`，避免出现错误。不过，如果你的系统已经做了正确的设置，你应该使用更简洁的形式。

3.6.2 使用 Guard 自动测试

使用 rspec 命令有一点很烦人，你总是要切换到命令行然后手动输入命令执行测试。（另一个很烦人的事情是，测试的启动时间很慢，3.6.3 节会说明）本节我们会介绍如何使用 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_spec.rb，那么只有这个文件中的测试会被运行。而且，我们可以适当的设置 Guard，当 home.html.erb 被修改后，也会自动运行 static_pages_spec.rb。

首先我们要把 guard-rspec 加入 Gemfile。（参见代码 3.34）

代码 3.34：示例程序的 Gemfile，包含 Guard

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'

group :development, :test do
  gem 'sqlite3', '1.3.8'
  gem 'rspec-rails', '2.13.1'
  gem 'guard-rspec', '2.5.0'
end

group :test do
  gem 'selenium-webdriver', '2.35.1'
  gem 'capybara', '2.1.0'

  # Uncomment this line on OS X.
  # gem 'growl', '1.0.3'

  # Uncomment these lines on Linux.
  # gem 'libnotify', '0.8.0'

  # Uncomment these lines on Windows.
  # gem 'rb-notifu', '0.0.4'
  # gem 'win32console', '1.3.2'
  # gem 'wdm', '0.1.0'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'

group :doc do
```

```
  gem 'sdoc', '0.3.20', require: false
end

group :production do
  gem 'pg', '0.15.1'
end
```

一定要根据自己所用的系统，取消上面测试组中相关 gem 的注释。

然后运行 `bundle install` 安装这些 gem：

```
$ bundle install
```

然后初始化 Guard，这样它才能和 RSpec 一起使用：

```
$ bundle exec guard init rspec
Writing new Guardfile to /Users/mhartl/rails_projects/sample_app/Guardfile
rspec guard added to Guardfile, feel free to edit it
```

然后再编辑 `Guardfile`，这样当集成测试和视图改变后 Guard 才能运行对应的测试。（参见代码 3.35）

代码 3.35：加入默认 `Guardfile` 的代码，注意顶部的 `require`

```
require 'active_support/inflector'

guard 'rspec', all_after_pass: false do
  .
  .
  .
  watch('config/routes.rb')
  # Custom Rails Tutorial specs
  watch(%r{^app/controllers/(.+)_(controller)\.rb$}) do |m|
    ["spec/routing/#{m[1]}_routing_spec.rb",
     "spec/#{m[2]}s/#{m[1]}_#{m[2]}_spec.rb",
     "spec/acceptance/#{m[1]}_spec.rb",
     (m[1][/_pages/] ? "spec/requests/#{m[1]}_spec.rb" :
                       "spec/requests/#{m[1].singularize}_pages_spec.rb")]
  end
  watch(%r{^app/views/(.+)/}) do |m|
    (m[1][/_pages/] ? "spec/requests/#{m[1]}_spec.rb" :
                      "spec/requests/#{m[1].singularize}_pages_spec.rb")
  end
  watch(%r{^app/controllers/sessions_controller\.rb$}) do |m|
    "spec/requests/authentication_pages_spec.rb"
  end
  .
```

```
  .
  .
end
```

下面这行

```
guard 'rspec', all_after_pass: false do
```

确保失败的测试通过后 Guard 不会运行所有的测试（为了加快“遇红，变绿，重构”过程）。

现在我们可以运行下面的命令启动 `guard` 了：

```
$ bundle exec guard
```

如果你不想输入命令前面的 `bundle exec`，需要按照 3.6.1 节中介绍的内容进行设置。

顺便说一下，如果 Guard 提示缺少 `spec/routing` 目录，你可以创建一个空的文件夹来修正这个错误：

```
$ mkdir spec/routing
```

3.6.3 使用 Spork 加速测试

运行 `bundle exec rspec` 时你或许已经察觉到了，在开始运行测试之前有好几秒的停顿时间，一旦测试开始就会很快完成。这是因为每次 RSpec 运行测试时都要重新加载整个 Rails 环境。Spork 测试服务器[11]可以解决这个问题。Spork 只加载一次环境，然后会为后续的测试维护一个进程池。Spork 结合 Guard（参见 3.6.2 节）使用就更强大了。

首先要把 `spork` 加入 `Gemfile`。（参见代码 3.36）

代码 3.36： 示例程序的 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'

group :development, :test do
  .
  .
  .
  gem 'spork-rails', '4.0.0'
  gem 'guard-spork', '1.5.0'
  gem 'childprocess', '0.3.6'
end
```

11. Spork 是 spoon-fork 的合成词。这个项目的名字之所以叫做 Spork 也是取 POSIX forks 的双关。

```
.
.
.
```

（上述代码使用了针对本教程修改的 `spork-rails`，目的是兼容 Rails 4。希望以后不再需要使用这个修改的版本，请随时关注该部分内容。）然后运行 `bundle install` 安装 Spork：

```
$ bundle install
```

接下来导入 Spork 的设置：

```
$ bundle exec spork --bootstrap
```

现在我们要修改名为 `spec/spec_helper.rb` 的 RSpec 设置文件，让所需的环境在一个预派生（prefork）代码块中加载，这样才能保证环境只被加载一次。（参见代码 3.37）

代码 3.37：将环境加载代码加入 `Spork.prefork` 代码块
`spec/spec_helper.rb`

```
require 'rubygems'
require 'spork'

Spork.prefork do
  ENV["RAILS_ENV"] ||= 'test'
  require File.expand_path("../../config/environment", __FILE__)
  require 'rspec/rails'
  require 'rspec/autorun'

  # Requires supporting ruby files with custom matchers and macros, etc,
  # in spec/support/ and its subdirectories.
  Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

  # Checks for pending migrations before tests are run.
  # If you are not using ActiveRecord, you can remove this line.
  ActiveRecord::Migration.check_pending! if defined?(ActiveRecord::Migration)

  RSpec.configure do |config|
    # ## Mock Framework
    #
    # If you prefer to use mocha, flexmock or RR, uncomment the appropriate line:
    #
    # config.mock_with :mocha
    # config.mock_with :flexmock
    # config.mock_with :rr

    # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
```

```
    config.fixture_path = "#{::Rails.root}/spec/fixtures"

    # If you're not using ActiveRecord, or you'd prefer not to run each of your
    # examples within a transaction, remove the following line or assign false
    # instead of true.
    config.use_transactional_fixtures = true

    # If true, the base class of anonymous controllers will be inferred
    # automatically. This will be the default behavior in future versions of
    # rspec-rails.
    config.infer_base_class_for_anonymous_controllers = false

    # Run specs in random order to surface order dependencies. If you find an
    # order dependency and want to debug it, you can fix the order by providing
    # the seed, which is printed after each run.
    #     --seed 1234
    config.order = "random"
    config.include Capybara::DSL
  end
end

Spork.each_run do
  # This code will be run each time you run your specs.

end
```

运行 Spork 之前，我们可以运行下面的计时命令为测试时间的提高找一个基准：

```
$ time bundle exec rspec spec/requests/static_pages_spec.rb
......

6 examples, 0 failures

real   0m8.633s
user   0m7.240s
sys    0m1.068s
```

我们看到测试组件用了超过 7 秒的时间，测试本身也用了超过 0.1 秒。为了加速这个过程，我们可以打开一个专门的命令行窗口，进入应用程序的根目录，然后启动 Spork 服务器[12]：

```
$ bundle exec spork
Using RSpec
```

12. 译者注：若Spork在Ubuntu下无法运行，请参照这个 gist 修改相应文件的权限。

```
Loading Spork.prefork block...
Spork is ready and listening on 8989!
```

（如果不想输入命令前面的 `bundle exec`，请参照 3.6.1 节中的内容。）在另一个命令行窗口中，运行测试组件，并指定 `--drb`（distributed Ruby，分布式 Ruby）选项，验证一下环境的加载时间是否明显的减少了：

```
$ time bundle exec rspec spec/requests/static_pages_spec.rb --drb
......

6 examples, 0 failures

real   0m2.649s
user   0m1.259s
sys 0m0.258s
```

每次运行 `rspec` 都要指定 `-drb` 选项有点麻烦，所以我建议将其加入应用程序根目录下的 `.rspec` 文件中，如代码 3.38 所示。

代码 3.38： 设置 RSpec 让其自动使用 Spork
`.rspec`

```
--colour
--drb
```

使用 Spork 时的一点说明：修改完预派生代码块中包含的文件后（例如 `routes.rb`），你要重启 Spork 服务器让它重新加载 Rails 环境。如果测试失败了，而你觉得它应该是通过的，可以使用 `Ctrl-C` 退出然后重启 Spork：

```
$ bundle exec spork
Using RSpec
Loading Spork.prefork block...
Spork is ready and listening on 8989!
^C
$ bundle exec spork
```

Guard 和 Spork 协作

Spork 和 Guard 一起使用时会很强大，我们可以使用如下的命令设置：

```
$ bundle exec guard init spork
```

然后我们要按照代码 3.39 所示的内容修改 `Guardfile`。

代码 3.39： 为使用 Spork 而修改的 `Guardfile`

```
require 'active_support/inflector'

guard 'spork', :cucumber_env => { 'RAILS_ENV' => 'test' },
```

```
                :rspec_env     => { 'RAILS_ENV' => 'test' } do
  watch('config/application.rb')
  watch('config/environment.rb')
  watch('config/environments/test.rb')
  watch(%r{^config/initializers/.+\.rb$})
  watch('Gemfile')
  watch('Gemfile.lock')
  watch('spec/spec_helper.rb') { :rspec }
  watch('test/test_helper.rb') { :test_unit }
  watch(%r{features/support/}) { :cucumber }
end

guard 'rspec', all_after_pass: false, cli: '--drb' do
  .
  .
  .
end
```

注意我们修改了 `guard` 的参数，包含了 `cli: '--drb'`，这可以确保 Guard 是在 Spork 服务器的命令行界面（Command-line Interface, cli）中运行的。我们还加入了监视 `features/support/` 目录的命令，这个目录会从第 5 章开始监视。

修改完之后，我们就可以通过 `guard` 命令同时启动 Guard 和 Spork 了：

```
$ bundle exec guard
```

Guard 会自动启动 Spork 服务器，大大减少了每次运行测试的时间。

配置了 Guard、Spork和（可选的）测试通知的测试环境会让测试驱动开发的过程变得有趣，让人沉溺其中。更多内容请观看本书的配套视频。

3.6.4 在 Sublime Text 中进行测试

如果你使用 Sublime Text 的话，它有一些强大的命令可以在编辑器中直接运行测试。这是我目前使用的方式，在测试很少时运作的相当好，即便是特大型的测试表现也不错。如果要使用这个功能，你要参考 Sublime Text 2 Ruby 测试中针对你所用系统的说明进行设置。在我的系统中（Mac OS X），我可以按照下面的方法安装所需的命令：

```
$ cd ~/Library/Application\ Support/Sublime\ Text\ 2/Packages
$ git clone https://github.com/maltize/sublime-text-2-ruby-tests.git RubyTest
```

这时你或许也想按照 Rails 教程 Sublime Text 的说明设置一下。

重启 Sublime Text，RubyTest 包提供了如下的命令：

- **Command-Shift-R**：运行单一的测试（如果在 `it` 块中运行），或者一组测试（如果在 `describe` 块中运行）
- **Command-Shift-E**：运行上一次运行的测试
- **Command-Shift-T**：运行当前文件中的所有测试

因为即使是一个小型项目的测试也可能会花费很多时间，所以可以单独的运行一个测试（或一小组测试）就可以节省很多的时间。即便是单一的测试也要预先加载 Rails 环境，所以这些命令最好是在 Spork 中运行：运行单一的测试减少了运行整个测试文件的时间，再运行 Spork 还可以减少启动测试环境的时间。下面是我推荐的操作顺序：

1. 在一个命令行窗口中启动 Spork；
2. 编写一个测试或一小组测试；
3. 执行 Command-Shift-R 命令，查看测试或测试组是否为红色；
4. 编写相应的程序代码；
5. 执行 Command-Shift-E 再次运行刚才的测试，查看测试是否变成绿色了；
6. 如果需要，重复第 2-5 步；
7. 完成一个任务后（在提交之前），在命令行中运行 `rspec spec/` 确保全部的测试仍是绿色的。

即使你可以在 Sublime Text 中运行测试，有时我还是会选择使用 Guard。以上就是我要介绍的我经常使用的 TDD 技术。

此页留白

第 4 章 Rails 背后的 Ruby

有了第 3 章中的例子做铺垫，本章将为你介绍一些对 Rails 来说很重要的 Ruby 知识。Ruby 语言的知识点很多，不过对一个 Rails 开发者而言需要掌握的很少。我们采用的是有别于常规的 Ruby 学习过程，我们的目标是开发动态的 Web 应用程序，所以我建议你先学习 Rails，在这个过程中学习一些 Ruby 知识。如果要成为一个 Rails 专家，你就要更深入的掌握 Ruby 了。本书会为你在成为专家的路途上奠定一个坚实的基础。如 1.1.1 节中说过的，读完本书后我建议你阅读一本专门针对 Ruby 的书，例如《Ruby 入门》、《The Well-Grounded Rubyist》或《Ruby 之道》。

本章介绍了很多内容，第一遍阅读没有掌握全部是可以理解的。在后续的章节我会经常提到本章的内容。

4.1 导言

从上一章我们可以看到，即使不懂任何背后用到的 Ruby 语言，我们也可以创建一个 Rails 应用程序骨架，也可以进行测试。不过我们依赖的是本教程中提供的测试代码，得到错误信息，然后让其通过。我们不能总是这样做，所以这一章我们要暂别网站开发学习，正视我们的 Ruby 短肋。

上次接触应用程序时，我们已经使用 Rails 布局去掉了几乎是静态的页面中的代码重复，参见代码 4.1，这段代码就是重新排版一下代码 3.26。

代码 4.1：示例程序的网站布局
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= 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>
```

让我们把注意力集中在代码 4.1 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`（更多内容请查看 Rails API 文档）为所有的媒介类型引入了 `application.css`。对于经验丰富的 Rails 开发者来说，这一行很简单，但是这里却至少包含了困惑着你的四个 Ruby 知识点：内置的 Rails 方法，不用括号的方法调用，Symbol 和 Hash。这几点本章都会介绍。

除了提供很多内置的方法供我们在视图中使用之外，Rails 还允许我们自行创建。自行创建的这些方法叫做帮助方法（helper）。要说明如何自行创建一个帮助方法，我们要来看看代码 4.1 中标题那一行：

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

这行代码依赖于每个视图中定义的页面标题（使用 `provide`），例如

```
<% provide(:title, 'Home') %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://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？我们的标题一般都包含一个公共部分，如果想更具体些就要加上一个变动的部分了。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。如果我们删除视图中的 `provide` 方法调用，输出的标题就没有了变动的那部分：

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

公共部分已经输出了，而且后面还有一个竖杠 |。

为了解决这个标题问题，我们会自定义一个帮助方法，叫做 `full_title`。如果视图中没有定义标题，`full_title` 会返回标题的公共部分，即“Ruby on Rails Tutorial Sample App”；如果定义了，则会在公共部分后面加上一个竖杠，然后再接上该页面的标题（如代码 4.2）。[1]

代码 4.2： 定义 `full_title` 帮助方法
`app/helpers/application_helper.rb`

```
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title)
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      "#{base_title} | #{page_title}"
```

1. 如果帮助函数是针对某个特定控制器的，你应该把它放进该控制器相应的帮助文件中。例如，为 StaticPages 控制器创建的帮助函数一般放在 `app/helper/static_pages_helper.rb` 中。在这个例子中，我们会把 `full_title` 这个帮助函数用在网站内所有的网页中，针对这种情况 Rails 提供了一个特别的文件：`app/helper/application_helper.rb`。

```
    end
  end
end
```

现在我们已经定义了一个帮助方法，我们可以用它来简化布局，将

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

替换成

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

如代码 4.3 所示。

代码 4.3： 示例程序的网站布局
app/views/layouts/application.html.erb

```
<!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>
```

为了让这个帮助方法起作用，我们要在“首页”视图中将不必要的“Home”这个词删掉，让标题只保留公共部分。首先我们要按照代码 4.4 的内容更新现有的测试，增加对没包含 `'Home'` 的标题测试。

代码 4.4： 更新“首页”标题的测试
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      expect(page).to have_content('Sample App')
    end
```

```
    it "should have the base title" do
      visit '/static_pages/home'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App")
    end

    it "should not have a custom page title" do
      visit '/static_pages/home'
      expect(page).not_to have_title('| Home')
    end
  end
  .
  .
  .
end
```

试试看你能否猜到为什么我们添加了一个新测试而不是直接修改之前的测试。（提示：答案在 3.3.1 节中。）

运行测试，查看是否有一个测试失败了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

为了让测试通过，我们要将“首页”视图中的 `provide` 那行删除，如代码 4.5 所示。

代码 4.5：删除标题定义后的“首页”
`app/views/static_pages/home.html.erb`

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

现在测试应该可以通过了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

和引入应用程序样式表那行代码一样，代码 4.2 的内容对经验丰富的 Rails 开发者来说看起来很简单，但是充满了很多会让人困惑的 Ruby 知识：module，注释，局部变量的赋值，布尔值，流程控制，字符串插值，还有返回值。这章也会介绍这些知识。

4.2 字符串和方法

学习 Ruby 我们主要使用的工具是 Rails 控制台，它是用来和 Rails 应用程序交互的命令行，在 2.3.3 节中介绍过。这个控制台是基于 Ruby 的交互程序（`irb`）开发的，因此也就能使用 Ruby 语言的全部功能。（在 4.4.4 节中会介绍，控制台还可以进入 Rails 环境。）使用下面的方法在命令行中启动控制台：

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

默认情况下，控制台是以开发环境启用的，这是 Rails 定义的三个独立的环境之一（其他两个是测试环境和生产环境）。三个环境的区别在本章还不需要知道，我们会在 7.1.1 节中更详细的介绍。

控制台是个很好的学习工具，你不用有所畏惧尽情的使用吧，没必要担心，你（几乎）不会破坏任何东西。如果你在控制台中遇到问题了可以使用 Ctrl-C 结束当前执行的命令，或者使用 Ctrl-D 直接退出控制台。在阅读本章后面的内容时，你会发现查阅 Ruby API 会很有用。API 包含很多信息，例如，如果你想查看关于 Ruby 字符串更多的内容，可以查看其中的 `String` 类页面。

4.2.1 注释

Ruby 中的注释以井号 #（也叫“Hask Mark”，或者更诗意的叫“散列字元”）开头，一直到行尾结束。Ruby 会忽略注释，但是注释对代码阅读者（包括代码的创作者）却很有用。在下面的代码中

```
# Returns the full title on a per-page basis.
def full_title(page_title)
  .
  .
  .
end
```

第一行就是注释，说明了后面方法的作用。

一般无需在控制台中写注释，不过为了说明代码，我会按照下面的形式加上注释，例如：

```
$ rails console
>> 17 + 42   # Integer addition
=> 59
```

在本节的阅读过程中，在控制台中输入或者复制粘贴命令时，如果愿意你可以不复制注释，反正控制台会忽略注释。

4.2.2 字符串

字符串算是 Web 应用程序中最有用的数据结构了，因为网页的内容就是从数据库发送到浏览器的字符串。我们先在控制台中体验一下字符串，这次我们使用 `rails c` 启动控制台，这是 `rails console` 的简写形式：

```
$ rails c
>> ""          # 空字符串
=> ""
>> "foo"       # 非空的字符串
=> "foo"
```

上面的字符串是字面量（字面量字符串，literal string），通过双引号（`"`）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"` 的运行结果是字符串 `"foobar"`。[2]

另外一种创建字符串的方式是通过一个特殊的句法（`#{}`）进行插值操作：[3]

```
>> 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`，然后将其插入到字符串 `"#{first_name} Hartl"` 中。我们可以将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，两个表达式的结果是相同的，不过我倾向使用插值的方式。在两个字符串中加入一个空格（`" "`）显得很别扭。

2. 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。
3. 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"`。

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（side-effect）：`puts "foo"` 首先会将字符串打印到屏幕上，然后再返回空值字面量：`nil` 是 Ruby 中的“什么都没有”。（后续内容中为了行文简洁我会省略 `=> nil`。）

`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 类似，但没有添加换行符）
foo=> nil
>> print "foo\n"   # 和 puts "foo" 一样
=> nil
```

单引号字符串

目前介绍的例子都是使用双引号创建的字符串，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"
```

不过两种方法还是有个很重要的区别：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"
```

注意控制台是如何使用双引号返回结果的，需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所做的所有事，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，就像换行符（`\n`）一样。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"
```

和前例的 `#` 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，不过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 \\."
```

4.2.3 对象及向其传递消息

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。

不过说出对象的作用就很简单：它可以响应消息。例如，一个字符串对象可以响应 `length` 这个消息，它返回字符串包含的字符数量：

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

这样传递给对象的消息叫做方法，它是在对象中定义的函数。[4]字符串还可以响应 `empty?` 方法：

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

注意 `empty?` 方法末尾的问号，这是 Ruby 的一个约定，说明方法的返回值是布尔值：`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"
```

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

```
>> 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?
```

4. 很抱歉本章在函数和方法之间随意的来回使用。在 Ruby 中这二者是同一个概念：所有的方法都是函数，所有的函数也都是方法，因为一切皆对象。

```
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil
```

因为 Ruby 中的一切都是对象，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: You have a nil object when you didn\'t expect it!
You might have expected an instance of Array.
The error occurred while evaluating nil.empty?
>> 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，你应该能猜到这个方法：

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

下面的代码

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

说明了关键词 if 的另一种用法：你可以编写一个当且只当 if 后面的表达式为真时才执行的语句。对应的，关键词 unless 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 的特殊性，除了 false 本身之外，所有的 Ruby 对象中它是唯一一个布尔值为“假”的：

```
>> if nil
>>   true
>> else
>>   false        # nil 是假值
>> end
=> false
```

基本上所有其他的 Ruby 对象都是“真”的，包括 0：

```
>> if 0
>>   true        # 0（除了 nil 和 false 之外的一切对象）是真值
>> else
>>   false
>> end
=> 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 3.6）和 `full_title` 帮助方法（代码 4.2）一样进行方法定义。（在控制台中定义方法有点麻烦，我们一般会在文件中定义，不过用来演示还行。）例如，我们要定义一个名为 `string_message` 的方法，可以接受一个参数，返回值取决于参数是否为空：

```
>> def string_message(string)
>>   if string.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.
```

注意 Ruby 方法会非显式的返回值：返回最后一个语句的值。在上面的这个例子中，返回的值会根据参数是否为空而返回两个字符串中的一个。Ruby 也支持显式的指定返回值，下面的代码和上面的效果一样：

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

细心的读者可能会发现其实这里第二个 `return` 不是必须的，作为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。不过两处都加上 `return` 看起来更好看。

4.2.5 回顾一下标题的帮助方法

下面我们来理解一下代码 4.2 中的 `full_title` 帮助方法：[5]

```
module ApplicationHelper

  # 根据所在页面返回完整的标题                                  # 在文档中显示的注释
  def full_title(page_title)                                    # 方法定义
    base_title = "Ruby on Rails Tutorial Sample App"            # 变量赋值
    if page_title.empty?                                        # 布尔测试
      base_title                                                # 非显式返回值
    else
      "#{base_title} | #{page_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帮助方法。还用到了 `module ApplicationHelper`：module 为我们提供了一种把相关方法组织在一起的方式，稍后我们可以使用 `include` 把它插入其他的类中。编写一般的 Ruby 程序时，你要自己定义一个 module 然后再显式的将其引入类中，但是对于帮助方法所在的 module 就交由 Rails 来处理引入了，最终的结果是 `full_title` 方法自动的就可以在所有的视图中使用了。

4.3 其他的数据类型

虽然 Web 程序一般都是处理字符串，但也需要其他的数据类型来生成字符串。本节我们就来介绍一些对开发 Rails 应用程序很重要的 Ruby 中的其他数据类型。

4.3.1 数组和 Range

数组就是一组顺序特定的元素。本书尚且没有用过数组，不过理解了数组就能很好的理解 Hash （4.3.3 节），也有助于理解 Rails 中的数据模型（例如 2.3.3 节中用到的 `has_many` 关联，10.1.3 节会做详细介绍）。

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

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

5. 其实这里还有一个地方我们还不能理解，那就是 Rails 是怎么把这些联系在一起的：把 URI 映射到动作上，`full_title` 帮助函数可以在视图中使用，等。这是个很有意思的话题，我建议你以后好好的了解一下，不过使用 Rails 并不需要完全了解 Rails 的运作机理。（若想更深入的了解 Rails，我推荐阅读 Obie Fernandez 的《Rails 3 之道》。）

上述代码的返回结果是一个有三个元素的数组。默认情况下，`split` 在空格处把字符串分割成数组，当然你几乎可以在任何地方进行分割：

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

和其他编程语言的习惯一样，Ruby 中数组的索引（index）也是从零开始的，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，在 Ruby 中是使用方括号来获取数组元素的。除了这种方法，Ruby 还为一些常用的元素获取操作提供了别名（synonym）：[6]

```
>> 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 == 进行对比
=> true
```

最后一行介绍了相等比较操作符 `==`，Ruby 和其他语言一样还提供了对应的 `!=`（不等）等其他的操作符：

```
>> x = a.length            # 和字符串一样，数组也可以响应 length 方法
=> 3
>> x == 3
=> true
>> x == 1
=> false
```

6. 下面代码中使用的 `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 的功能扩展。

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

除了 length（上述代码的第一行）之外，数组还可以响应一堆其他的方法：

```
>> a
=> [42, 8, 17]
>> 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。如果你想修改数组的值要使用对应的“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]
```

你还可以使用 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"]
```

最后一个例子说明你可以把添加操作串在一起操作；同时也说明，Ruby 不像很多其他的语言，数组可以包含不同类型的数据（本例中是数字和字符串混合）。

前面我们用 split 把字符串分割成字符串，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 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 Range
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

虽然 `0..9` 是一个合法的 Range，不过上面第二个表达式告诉我们调用方法时要加上括号。

Range 经常被用来获取一组数组元素：

```
>> 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"]
```

Range 也可使用字母：

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

4.3.2 块

数组和 Range 可以响应的方法中有很多都可以跟着一个块（block），这是 Ruby 中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
```

这个代码在 Range `(1..5)` 上调用了 `each` 方法，然后又把 `{ |i| puts 2*i }` 这个块传递给 `each` 方法。`|i|` 两边的竖杠在 Ruby 句法中是用来定义块变量的。只有这个方法才知道如何处理后面跟着的块。本例中，Range 的 `each` 方法会处理后面的块，块中有一个本地变量 `i`，`each` 会将 Range 中的各个值传进块中然后执行相应的操作。

花括号是一种定义块的方法，还有另一种方法可用：

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

块可以多于一行，也经常是多于一行的。本书中我们会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`，我想告诉你的是任何变量名都可以使用。

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，看得多了最后你就会习惯它的用法了。[7] 幸好人类擅长于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!"
```

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

```
=> 3
>> (1..5).map { |i| i**2 }          # ** 表示幂
=> [1, 4, 9, 16, 25]
>> %w[a b c]                        # 再说一下，%w 可以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"]
```

上面的代码说明，map 方法返回的是在数组或 Range 的每个元素上执行块中代码后的结果。

现在我们就可以来理解一下我在 1.4.4 节中用来生成随机二级域名的那行 Ruby 代码了：

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

我们一步一步分解一下：

```
>> ('a'..'z').to_a                      # 字母表数组
=> ["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]        # 取出前面的 8 个元素
=> ["f", "w", "i", "a", "h", "p", "c", "x"]
>> ('a'..'z').to_a.shuffle[0..7].join   # 将取出的元素合并成字符串
=> "mznpybuj"
```

4.3.3 Hash 和 Symbol

Hash 本质上就是数组，只不过它的索引不局限于使用数字。（实际上在一些语言中，特别是 Perl，因为这个原因就把 Hash 叫做关联数组（associative array）。）Hash 的索引（或者叫“键”）几乎可以是任何对象。例如，我们可以使用字符串当键：

```
>> user = {}                          # {} 是一个空 Hash
=> {}
>> 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"]                 # 获取元素的方式类似数组
=> "Michael"
```

```
>> user                                    # Hash 的字面量形式
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Hash 通过一对花括号中包含一些键值对的形式表示，只有一对花括号而没有键值对（`{}`）就是一个空 Hash。需要注意，Hash 中的花括号和块中的花括号不是一个概念。（是的，这可能会让你迷惑。）不过，Hash 虽然和数组类似，但却有一个很重要的区别：Hash 的元素没有特定的顺序。[8] 如果顺序很重要的话就要使用数组了。

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 `=>` 分隔的键值对这种字面量的形式定义 Hash 要简洁得多，我们称后一种方式为“hashrocket”：

```
>> 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，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。）

目前为止我们的键用的都是字符串，但在 Rails 中用 Symbol 当键却很常见。Symbol 看起来像字符串，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 Symbol。你可以把 Symbol 看成没有约束的字符串：[9]

```
>> "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
```

Symbol 是 Ruby 特有的一个数据类型，其他语言很少用到，初看起来感觉很奇怪，不过 Rails 经常用到它，所以你很快就会习惯的。

用 Symbol 当键，我们可以按照如下的方式定义一个 `user` Hash：

```
>> 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]            # 获取一个未定义的键对应的值
=> nil
```

8. 在 Ruby 1.9 中，其实会按照元素输入时的顺序保存 Hash，不过依赖顺序显然是不明智的。
9. 没有了约束的好处是，Symbol 很容易进行比较，字符串要按照字母一个一个的比较，而 Symbol 只需进行一次操作。这就使得 Symbol 成为 Hash 键的最佳选择。

从上面的例子我们可以看出，Hash 中没有定义的键对应的值是 `nil`。

因为 Symbol 当键的情况太普遍了，Ruby 1.9 干脆就为这种情况定义了一个新的句法：

```
>> 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
```

第二个命令把 hashrocket 形式的键值对变成了键后跟着一个冒号然后再跟着一个值的形式：

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

这种结构更好的沿袭了其他语言（例如 JavaScript）中 Hash 的表现方式，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。

Hash 元素的值可以是任何对象，甚至是一个 Hash，如代码 4.6 所示。

代码 4.6：Hash 嵌套

```
>> params = {}        # 定义一个名为 params（parameters 的简称）的 Hash
=> {}
>> 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"
```

这种 Hash 中有 Hash 的形式（或称为 Hash 嵌套）在 Rails 中大量的使用，我们从 7.3 节开始会接触到。

与数组和 Range 一样，Hash 也可以响应 `each` 方法。例如，一个名为 `flash` 的 Hash，它的键是两个条件判断，`:success` 和 `:error`：

```
>> flash = { success: "It worked!", error: "It failed." }
=> {:success=>"It worked!", :error=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :error has value "It failed."
```

注意，数组的 `each` 方法后面的块只有一个变量，而 Hash 的 `each` 后面的块接受两个变量，`key` 和 `value`。所以 Hash 的 `each` 方法每次遍历都会以一个键值对为基本单位进行。

上面的示例中用到了很有用的 `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              # 把数组作为字符串输出
1
2
3
4
5
>> puts (1..5).to_a.inspect      # 输出一个数组字面量形式
[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` 输出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：

```
>> 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 4.1 中在布局中引入 CSS 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 节中简单的提到过，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，括号哪儿去了？因为在 Ruby 中，括号是可以省略的，下面的两行代码是等价的：

```
# Parentheses on function calls are optional.
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 Hash，但是怎么没用花括号？因为在调用函数时，如果 Hash 是最后一个参数，它的花括号是可以省略的。下面的两行代码是等价的：

```
# Curly braces on final hash arguments are optional.
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？因为如果使用新句法，写成

```
data-turbolinks-track: true
```

就会尝试创建一个 Symbol :data-turbolinks-track，但是因为包含了连字符，所以这个 Symbol 是不合法的。所以就只能使用旧句法，写成

```
"data-turbolinks-track" => true
```

最后，为什么下面这两行代码

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

为什么可以这么写，中间有空格也可以？当然可以，Ruby 不关心有没有换行。[^1-10]我之所以把代码拆成两行，是要保持每行代码不超过 80 列。[^1-11]

所以我们就看到了这样一行代码

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

它调用了 stylesheet_link_tag 函数，传进两个参数：一个是字符串，指明样式表的路径；另一个是 Hash，包含两个元素，指明媒介类型，并启用 Turbolink 功能(Rails 4 的新功能，本书的后续草稿会深入介绍)。因为使用的是 <%= %>，函数的执行结果会通过 ERb 插入模板中，如果你在浏览器中查看网页的源代码就会看到引入样式表所用的 HTML（参见代码 4.7）。（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，可以确保 CSS 修改后浏览器会重新加载它。）

代码 4.7：引入 CSS 的代码生成的 HTML

```
<link data-turbolinks-track="true" href="/assets/application.css" media="all"
rel="stylesheet" />
```

如果你打开 http://localhost:3000/assets/application.css 查看 CSS 的话，会发现是空的（除了一些注释）。在第 5 章中我们会看介绍如何添加样式。

4.4 Ruby 类

我们之前已经说过 Ruby 中的一切都是对象，本节我们就会自己定义一些对象。Ruby 和其他面向对象（object-oriented）编程语言一样，使用类来组织方法。然后实例化（instantiate）类创建对象。如果你刚接触面向对象编程，这些都似天书一般，那么让我们来看一些实际的例子吧。

4.4.1 构造器

我们看过很多例子使用类初始化对象，不过还没有正式的初始化过。例如，我们使用一个双引号初始化一个字符串，双引号是字符串的字面构造器（literal constructor）：

```
>> s = "foobar"        # 使用双引号的字面构造器
=> "foobar"
>> s.class
=> String
```

我们看到字符串可以响应 `class` 方法，返回的结果是字符串所属的类。

除了使用字面构造器之外，我们还可以使用等价的具名构造器（named constructor），即在类名上调用 `new` 方法：[10]

```
>> s = String.new("foobar")    # 字符串的具名构造器
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

上面的代码和字面构造器是等价的，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 Hash 就有些不同了。数组的构造器 `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)     # 设置不存在的键返回 0 而不是 nil
```

10. 返回的结果可能由于 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。

```
=> {}
>> h[:foo]
=> 0
```

在类上调用的方法，如本例的 new，我们称之为类方法（class method）。在类上调用 new 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，叫做实例方法（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。我们可以看到，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 对象都是适用的：只要在类的继承关系上往上多走几层就会发现 Ruby 中的每个类最终都是继承自 BasicObject，而其本身没有父类。这就是“Ruby 中一切皆对象”技术层面的意义。

图 4.1： String 类的继承关系

要更深入的理解类，最好的方法就是自己动手编写。我们来创建一个名为 Word 的类，包含一个名为 palindrome? 方法，如果单词顺读和反读时都一样则返回 true：

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> nil
```

我们可以按照下面的方式使用：

```
>> 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，我们的目的达到了。定义一个新类，可是只创建一个可以接受一个字符串参数的方法，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 4.8 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 4.8：在控制台中定义 Word 类

```
>> class Word < String              # Word 继承自 String
>>   # 如果字符串和自己反转后相等则返回 true
>>   def palindrome?
>>     self == self.reverse         # self 代表这个字符串本身
>>   end
>> end
=> nil
```

上面代码中的 Word < String 在 Ruby 中表示继承（3.1 节中简单介绍过），这样除了刚定义的 palindrome? 方法之外，Word 还拥有所有字符串拥有的方法：

```
>> s = Word.new("level")     # 创建一个新的 Word，初始值为 level
=> "level"
>> s.palindrome?             # Word 实例可以响应 palindrome? 方法
=> true
>> s.length                  # Word 实例还继承了字符串所有的常规方法
=> 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
```

继承关系如图 4.2 所示。

图 4.2：代码 4.8 中定义的 `Word` 类（非内置类）的继承关系

在代码 4.8 中，注意，要检查单词和单词的反转是否相同，要在 `Word` 类中引用这个单词，在 Ruby 中我们使用 `self` 进行引用：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
```

来检查单词是否是一个回文。[11]

4.4.3 修改内置的类

虽然继承是个很强大的功能，不过在回文判断的例子中，如果能把 `palindrome?` 加入 `String` 类就更好了，这样（除了其他对象外）我们就可以在字符串字面量上调用 `palindrome?` 方法了。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，允许像我们自己这样的普通人添加一些方法：[12]

```
>> class String
>>   # 如果字符串和自己反转后相等则返回 true
```

11. 关于 Ruby 类和 `self` 关键字，请阅读 RailsTips 上的《Class and Instance Variables in Ruby》一文。
12. 了解 JavaScript 的人可能知道这个功能与使用内置的类原型对象扩充类的方式类似。（感谢读者 Erik Eldridge 指出这一点。）

```
>>   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（神化，奉为神明）”是个回文。）

可以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，如果没有一个很好的理由，向内置的类中添加方法被认为是不好的习惯。Rails 自然有很好的理由，例如，在 Web 应用程序中我们经常要避免变量是空白（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。因为 Rails 控制台会自动加载 Rails 添加的扩展功能，我们可以看一下示例（在 `irb` 就不可以）：

```
>> "".blank?
=> true
>> "      ".empty?
=> false
>> "      ".blank?
=> true
>> nil.blank?
=> true
```

我们可以看到，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` 上的。我们会在 8.2.1 节中介绍一些 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，我们之前介绍过，在使用 StaticPages 控制器时（代码 3.16）：

```
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`，`StaticPagesController` 类中有三个方法 `home`、`help` 和 `about`。因为 Rails 控制台会加载本地的 Rails 环境，所以我们可以在控制台中创建一个控制器来查看一下它的继承关系：[13]

```
>> 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。不过，我记得没有在任何地方调用过 `StaticPagesController.new`，这是怎么回事？

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，应该区别于 Ruby 进行学习和理解。这就是为什么，如果你的兴趣是开发 Web 应用程序，我建议你先学 Rails 再学 Ruby，然后再回到 Rails 上的原因。

4.4.5 用户类

我们通过创建一个完整的类来结束对 Ruby 的介绍，一个 `User` 类，提前实现第 6 章的 User 模型。

到目前为止，我们都是在控制台中定义类的，这样很快捷，但也有点不爽。现在我们要在应用程序的根目录创建一个名为 `example_user.rb` 的文件，写入代码 4.9 中的内容。

13. 你没必要知道继承关系中的每个类。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 进行开发了。这可能意味着了以下两个问题中的一个，第一，我是个废柴，第二，你不需要知道所有的内在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**：StaticPages 控制器的继承关系

代码 **4.9**：`User` 类的代码
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
```

上面的代码有很多要说明的，我们一步步来。先看下面这行：

```
  attr_accessor :name, :email
```

它为用户的名字和 Email 地址创建了属性访问器（attribute accessors）。也就是定义了“获取（getter）”和“设定（setter）”方法，用来取回和赋值 `@name` 和 `@email` 实例变量，我们在 2.2.2 节中介绍过实例变量。在 Rails 中，实

例变量的意义在于，它们自动的在视图中可用。而通常实例变量的作用是用来在 Ruby 类中不同的方法之间传递变量值。（稍后会更详细的介绍这点。）实例变量总是以 @ 符号开头，如果未定义则其值为 nil。

第一个方法，initialize，在 Ruby 中有特殊意义：当我们执行 User.new 时会调用该方法。这个 initialize 方法可以接受一个参数，attributes：

```
def initialize(attributes = {})
  @name  = attributes[:name]
  @email = attributes[:email]
end
```

attributes 变量的初始值是一个空的 Hash，所以我们可以定义一个没有名字或没有 Email 地址的用户（回想一下 4.3.3 节，如果键不存在则返回 nil，所以如果没定义 :name 键，则 attributes[:name] 会返回 nil，attributes[:email] 也是一样）。

最后，定义了一个名为 formatted_email 的方法，它使用被赋了值的 @name 和 @email 变量进行插值，组成一个格式良好的用户 Email 地址（4.2.2 节）：

```
def formatted_email
  "#{@name} <#{@email}>"
end
```

因为 @name 和 @email 都是实例变量（如 @ 符号指明），所以它们在 formatted_email 方法中自动可见。

让我们打开控制台，加载（require）这个文件，实际操作一下这个用户类：

```
>> require './example_user'      # 这就是加载文件的方式
=> ["User"]
>> 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 Email 地址
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

上面代码中的点号 . 在 Unix 中是指“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的用户，然后通过直接赋值给相应的属性来提供他的名字和 Email 地址（因为有了代码 4.9 中 attr_accessor 那行才能进行赋值操作）。我们输入

```
example.name = "Example User"
```

Ruby 会将 `@name` 变量的值设为 `"Example User"`（`email` 属性类似），然后可以在 `formatted_email` 中使用。

如 4.3.4 节中介绍的，如果最后一个参数是 Hash，我们就可以省略花括号，我们可以把一个预先定义好的 Hash 传递给 `initialize` 方法来创建另一用户：

```
>> 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 Hash 初始化对象，这种技术叫做“mass assignment”，在 Rails 中很常用。

4.5 小结

现在结束对 Ruby 语言的介绍。在第 5 章我们会好好的利用这些知识来开发示例程序。

我们不会使用 4.4.5 节中创建的 `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：

```
$ git add .
$ git commit -m "Add a full_title helper"
```

4.6 练习

1. 将下面代码 4.10 中的问号换成合适的方法，结合`split`、`shuffle` 和 `join` 实现一个函数，把一个给定字符串中的字符顺序打乱。
2. 以下面代码 4.11 为蓝本，将 `shuffle` 方法添加到 `String` 类中。
3. 创建三个 Hash，分别名为 `person1`、`person2` 和 `person3`，将名和姓对应到 `:first` 和 `:last` 键上。再创建一个名为 `params` 的 Hash，使 `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. 找一个在线的 Ruby API 文档，阅读 `Hash` 的 `merge` 方法的使用方法。
5. （选做）跟着 Ruby Koans 学习 Ruby 入门知识。

代码 4.10：打乱字符串函数的骨架

```
>> def string_shuffle(s)
>>   s.split('').?.?
>> end
=> nil
>> string_shuffle("foobar")
```

代码 **4.11**：添加到 `String` 类的 `shuffle` 方法骨架

```
>> class String
>>   def shuffle
>>     self.split('').?.?
>>   end
>> end
=> nil
>> "foobar".shuffle
```

第 5 章 完善布局

第 4 章对 Ruby 做了简单的介绍，我们讲解了如何在应用程序中引入样式表，不过，就像在 4.3.4 节中说过的，这个样式表现在还是空的。本章我们会做些修改，把 Bootstrap 框架引入应用程序中，然后再添加一些自定义的样式。[1] 我们还会把已经创建的页面（例如“首页”和“关于”页面）添加到布局中（5.1 节）。在这个过程中，我们会介绍局部视图（partial）、Rails 路由和 asset pipeline，还会介绍 Sass（5.2 节）。我们还会用最新的 RSpec 技术重构第 3 章中的测试。最后，我们还会向前迈出很重要的一步：允许用户在我们的网站中注册。

图 5.1：示例程序“首页”的构思图

1. 感谢读者 Colm Tuite 使用 Bootstrap 重写了本书原来的示例程序；

5.1 添加一些结构

本书是关于 Web 开发而不是 Web 设计的，不过在一个看起来很垃圾的应用程序中开发会让人提不起劲，所以本书我们要向布局中添加一些结构，再加入一些 CSS 构建基本的样式。除了使用自定义的 CSS 之外，我们还会使用 Bootstrap，由 Twitter 开发的开源 Web 设计框架。我们还要按照一定的方式组织代码，即使用局部视图来保持布局文件的结构清晰，避免大量的代码混杂在布局文件中。

开发 Web 应用程序时，尽早的对用户界面有个统筹安排往往会对你有所帮助。在本书后续内容中，我会经常插入网页的构思图（mockup）（在 Web 领域经常称之为“线框图（wireframe）”），这是对应用程序最终效果的草图设计。[2]本章大部分内容都是在开发 3.1 节中介绍的静态页面，页面中包含一个网站 LOGO、导航条头部和网站底部。这些网页中最重要的一个是“首页”，它的构思图如图 5.1 所示。图 5.7 是最终实现的效果。你会发现二者之间的某些细节有所不同，例如，在最终实现的页面中我们加入了一个 Rails LOGO——这没什么关系，因为构思图没必要画出每个细节。

和之前一样，如果你使用 Git 做版本控制的话，现在最好创建一个新分支：

```
$ git checkout -b filling-in-layout
```

5.1.1 网站导航

在示例程序中加入链接和样式的第一步，要修改布局文件 `application.html.erb`（上次使用是在代码 4.3 中），添加一些 HTML 结构。我们要添加一些区域，一些 CSS class，以及网站导航。布局文件的内容参见代码 5.1，对各部分代码的说明紧跟其后。如果你迫不及待的想看到结果，请查看图 5.2。（注意：结果（还）不是很让人满意。）

代码 5.1：添加一些结构后的网站布局文件
`app/views/layouts/application.html.erb`

```
<!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 %>
    <!--[if lt IE 9]>
    <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>
  <body>
    <header class="navbar navbar-fixed-top navbar-inverse">
      <div class="navbar-inner">
        <div class="container">
```

2. 本书中的所有构思图都是通过 Mockingbird这个在线应用制作的；

```
          <%= link_to "sample app", '#', id: "logo" %>
          <nav>
            <ul class="nav pull-right">
              <li><%= link_to "Home",    '#' %></li>
              <li><%= link_to "Help",    '#' %></li>
              <li><%= link_to "Sign in", '#' %></li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

我们从上往下看一下代码 5.1 中新添加的元素。3.3.2 节简单的介绍过，Rails 4 默认会使用 HTML5（如 `<!DOCTYPE html>` 所示），因为 HTML5 标准还很新，有些浏览器（特别是较旧版本的 IE 浏览器）还没有完全支持，所以我们加载了一些 JavaScript 代码（称作“HTML5 shim”）来解决这个问题：

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

如下有点古怪的句法

```
<!--[if lt IE 9]>
```

只有当 IE 浏览器的版本小于 9 时（`if lt IE 9`）才会加载其中的代码。这个奇怪的 `[if lt IE 9]` 句法不是 Rails 提供的，其实它是 IE 浏览器为了解决兼容性问题而特别支持的条件注释（conditional comment）。这就带来了一个好处，因为这说明我们只会在 IE9 以前的版本中加载 HTML5 shim，而 Firefox、Chrome 和 Safari 等其他浏览器则不会受到影响。

后面的区域是一个 `header`，包含网站的 LOGO（纯文本）、一些小区域（使用 `div` 标签）和一个导航列表元素：

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", '#', id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home",    '#' %></li>
          <li><%= link_to "Help",    '#' %></li>
          <li><%= link_to "Sign in", '#' %></li>
```

```
        </ul>
      </nav>
    </div>
  </div>
</header>
```

header 标签的意思是放在网页顶部的内容。我们为 header 标签指定了三个 CSS class[3]，navbar、navbar-fixed-top 和 navbar-inverse，用空格分开：

```
<header class="navbar navbar-fixed-top navbar-inverse">
```

所有的 HTML 元素都可以指定 class 和 id，它们不仅是个标注，在 CSS 样式中也有用（5.1.2 节）。class 和 id 之间主要的区别是，class 可以在同一个网页中多次使用，而 id 只能使用一次。这里的三个 class 在 Bootstrap 框架中有特殊的意义，我们会在 5.1.2 节中安装并使用 Bootstrap。header 标签内是一些 div 标签：

```
<div class="navbar-inner">
  <div class="container">
```

div 标签是常规的区域，除了把文档分成不同的部分之外，没有特殊的意义。在以前的 HTML 中，div 标签被用来划分网站中几乎所有的区域，但是 HTML5 增加了 header、nav 和 section 元素，用来划分大多数网站中都有用到的区域。本例中，每个 div 也都指定了一个 CSS class。和 header 标签的 class 一样，这些 class 在 Bootstrap 中也有特殊的意义。

在这些 div 之后，有一些 ERb 代码：

```
<%= link_to "sample app", '#', id: "logo" %>
<nav>
  <ul class="nav pull-right">
    <li><%= link_to "Home",    '#' %></li>
    <li><%= link_to "Help",    '#' %></li>
    <li><%= link_to "Sign in", '#' %></li>
  </ul>
</nav>
```

这里使用了 Rails 中的 link_to 帮助方法来创建链接（在 3.3.2 节中我们是直接创建 a 标签来实现的）。link_to 的第一个参数是链接文本，第二个参数是链接地址。在 5.3.3 节中我们会指定链接地址为设置好的路由，这里我们用的是 Web 设计中经常使用的占位符 #。第三个参数是可选的，为一个 Hash，本例使用这个参数为 LOGO 添加了一个 logo id。（其他三个链接没有使用这个 Hash 参数，没关系，因为这个参数是可选的。）Rails 帮助方法经常这样使用 Hash 参数，可以让我们仅使用 Rails 的帮助方法就能灵活的添加 HTML 属性。

第二个 div 中是个导航链接列表，使用无序列表标签 ul，以及列表项目标签 li：

3. 这些 class 和 Ruby 的类一点关系都没有；

```
<nav>
  <ul class="nav pull-right">
    <li><%= link_to "Home",    '#' %></li>
    <li><%= link_to "Help",    '#' %></li>
    <li><%= link_to "Sign in", '#' %></li>
  </ul>
</nav>
```

上面代码中的 `nav` 标签以前是不需要的，它的目的是显示导航链接。`ul` 标签指定的 `nav` 和 `pull-right` class 在 Bootstrap 中有特殊的意义。 Rails 处理这个布局文件并执行其中的 ERb 代码后，生成的列表如下面的代码所示：

```
<nav>
  <ul class="nav pull-right">
    <li><a href="#">Home</a></li>
    <li><a href="#">Help</a></li>
    <li><a href="#">Sign in</a></li>
  </ul>
</nav>
```

布局文件的最后一个 `div` 是主内容区域：

```
<div class="container">
  <%= yield %>
</div>
```

和之前一样，`container` class 在 Bootstrap 中有特殊的意义。3.3.4 节已经介绍过，`yield` 会把各页面中的内容插入网站的布局中。

除了网站的底部（在 5.1.3 节添加）之外，布局现在就完成了，访问一下“首页”就能看到结果了。为了利用后面添加的样式，我们要向 `home.html.erb` 视图中加入一些元素。（参见代码 5.2。）

代码 5.2：“首页”的代码，包含一个到注册页面的链接
`app/views/static_pages/home.html.erb`

```
<div class="center hero-unit">
  <h1>Welcome to the Sample App</h1>

  <h2>
    This is the home page for the
    <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </h2>

  <%= link_to "Sign up now!", '#', class: "btn btn-large btn-primary" %>
</div>
```

```
<%= link_to image_tag("rails.png", alt: "Rails"), 'http://rubyonrails.org/' %>
```

上面代码中第一个 `link_to` 创建了一个占位链接，指向第 7 章中创建的用户注册页面

```
<a href="#" class="btn btn-large btn-primary">Sign up now!</a>
```

`div` 标签中的 `hero-unit` class 在 Bootstrap 中有特殊的意义，注册按钮的 `btn`、`btn-large` 和 `btn-primary` 也是一样。

图 **5.2**：没有定义 CSS 的“首页”（/static_pages/home）

第二个 `link_to` 用到了 `image_tag` 帮助方法，第一个参数是图片的路径；第二个参数是可选的，一个 Hash，本例中这个 Hash 参数使用一个 Symbol 键设置了图片的 `alt` 属性。为了更好的理解，我们来看一下生成的 HTML：[4]

```
<img alt="Rails" src="/assets/rails.png" />
```

`alt` 属性的内容会在图片无法加载时显示，也会在针对视觉障碍人士的屏幕阅读器中显示。人们有时懒得加上 `alt` 属性，可是在 HTML 标准中却是必须的。幸运的是，Rails 默认会加上 `alt` 标签，如果你没有在调用 `image_tag` 时指定的话，Rails 就会使用图片的文件名（不包括扩展名）。本例中，我们自己设定了 `alt` 文本，显示一个首字母大写的“Rails”。

4. 你大概注意到了 `img` 标签的格式，不是 `<img>...</img>` 而是 `<img ... />`。这样的标签叫做自关闭标签;

在 Rails 之前的版本中，每个 Rails 应用程序中都有 `rails.png` 这个图片，但在最新版中这个图片不会在执行 `rails new` 命令时生成，所以要手动从 Ruby on Rails 官网下载，地址为 http://rubyonrails.org/images/rails.png，保存到 `app/assets/images/` 文件夹中。（或许还要新建这个文件夹，可以使用 `mkdir` 命令或者图形化文件管理工具。）因为在代码 5.2 中我们使用了 `image_tag` 帮助方法，所以 Rails 会通过 Asset Pipeline（参见 5.2 节）自动在这个文件夹中寻找所需的图片。

现在我们终于可以看到劳动的果实了（如图 5.2）。你可能会说，这并不很美观啊。或许吧。不过也可以小小的高兴一下，我们已经为 HTML 结构指定了合适的 class，可以用来添加 CSS。

5.1.2 Bootstrap 和自定义的 CSS

在 5.1.1 节我们为很多 HTML 元素指定了 CSS class，这样我们就可以使用 CSS 灵活的构建布局了。5.1.1 节中已经说过，很多 class 在 Bootstrap 中都有特殊的意义。Bootstrap 是 Twitter 开发的框架，可以方便的把精美的 Web 设计和用户界面元素添加到使用 HTML5 开发的应用程序中。本节，我们会结合 Bootstrap 和一些自定义的 CSS 为示例程序添加样式。

首先要安装 Bootstrap，在 Rails 程序中可以使用 bootstrap-sass 这个 gem，参见代码 5.3。Bootstrap 框架本身使用 LESS 来动态的生成样式表，而 Rails 的 asset pipeline 默认支持的是（非常类似的）Sass，bootstrap-sass 会将 LESS 转换成 Sass 格式，而且 Bootstrap 中必要的文件都可以在当前的应用程序中使用。[5]

代码 5.3： 把 bootstrap-sass 加入 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'
gem 'bootstrap-sass', '2.3.2.0'
.
.
.
```

像往常一样，运行 `bundle install` 安装 Bootstrap：

```
$ bundle install
```

然后重启 Web 服务器，改动才能在应用程序中生效。（在大多数系统中可以使用 Ctrl-C 结束服务器，然后再执行 `rails server` 命令。）最后，对 Rails 4 来说，我们还要在 `config/application.rb` 中添加一行代码，让 `bootstrap-sass` 这个 gem 和 Asset Pipeline 兼容，如代码 5.4 所示。

代码 5.4： 添加一行代码，兼容 Asset Pipeline
`config/application.rb`

```
require File.expand_path('../boot', __FILE__)
.
.
.
```

5. 在 asset pipeline 中当然也可以使用 LESS，详见 less-rails-bootstrap gem；

```
module SampleApp
  class Application < Rails::Application
    .
    .
    .
    config.assets.precompile += %w(*.png *.jpg *.jpeg *.gif)
  end
end
```

要向应用程序中添加自定义的 CSS，首先要创建一个 CSS 文件：

```
app/assets/stylesheets/custom.css.scss
```

文件存放的目录和文件名都很重要。其中目录

```
app/assets/stylesheets
```

图 5.3：使用 Bootstrap CSS 后的示例程序

是 asset pipeline 的一部分（5.2 节），这个目录中的所有样式表都会自动的包含在网站的 `application.css` 中。`custom.css.scss` 文件的第一个扩展名是 `.css`，说明这是个 CSS 文件；第二个扩展名是 `.scss`，说明这是个

“Sassy CSS”文件。asset pipeline 会使用 Sass 处理这个文件。（在 5.2.2 节中才会使用 Sass，有了它 bootstrap-sass 才能运作。）

在自定义 CSS 文件中，我们可以使用 `@import` 引入 Bootstrap，如代码 5.5 所示。

代码 5.5： 引入 Bootstrap
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";
```

这行代码会引入整个 Bootstrap CSS 框架，结果如图 5.3 所示。（或许你要通过 Ctrl-C 来重启服务器。还有一点要注意，截图中使用的是 Bootstrap 2.0，而本教程用的是 Bootstrap 2.3，所以外观可能有所不同，不过无需过分担心。）可以看到，文本的位置还不是很合适，LOGO 也没有任何样式，不过颜色搭配和注册按钮看起来还不错。

下面我们要加入一些整站都会用到的 CSS，用来样式化网站布局和各单独页面，如代码 5.6 所示。代码 5.6 中定义了很多样式规则。为了说明 CSS 规则的作用，我们经常会加入一些 CSS 注释，放在 `/*...*/` 之中。代码 5.6 的 CSS 加载后的效果如图 5.4 所示。

图 5.4： 添加一些空白和其他的全局性样式

代码 5.6： 添加全站使用的 CSS
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";

/* universal */
```

```
html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

注意代码 5.6 中的 CSS 格式是很统一的。一般来说，CSS 规则是通过 class、id、HTML 标签或者三者结合在一起来定义的，后面会跟着一些样式声明。例如：

```
body {
  padding-top: 60px;
}
```

把页面的上内边距设为 60 像素。我们在 `header` 标签上指定了 `navbar-fixed-top` class，Bootstrap 就把这个导航条固定在页面的顶部。所以页面的上内边距会把主内容区和导航条隔开一段距离。（导航条的颜色在 Bootstrap 2.0 和 2.1 之间是不一样的，所以我们要加入 `navbar-inverse` class，把亮色变暗。）下面的 CSS 规则：

```
.center {
  text-align: center;
}
```

把 `.center` class 的样式定义为 `text-align: center;`。`.center` 中的点号说明这个规则是样式化一个 class。（我们会在代码 5.8 中看到，`#` 是样式化一个 id。）这个规则的意思是，任何 class 为 `.center` 的标签（例如 `div`），其中包含的内容都会在页面中居中显示。（代码 5.2 中有用到这个 class。）

虽然 Bootstrap 中包含了很精美的文字排版样式，我们还是要为网站添加一些自定义的规则，如代码 5.7 所示。（并不是所有的样式都会应用于“首页”，但所有规则都会在网站中的某个地方用到。）代码 5.7 的效果如图 5.5 所示。

图 5.5：添加了一些文字排版样式

代码 5.7：添加一些精美的文字排版样式
app/assets/stylesheets/custom.css.scss

```
@import "bootstrap";
.
.
.

/* typography */

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
```

```
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: #999;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}
```

最后，我们还要为只包含文本“sample app”的网站 LOGO 添加一些样式。代码 5.8 中的 CSS 样式会把文字变成全大写字母，还修改了文字大小、颜色和位置。（我们使用的是 id，因为我们希望 LOGO 在页面中只出现一次，不过你也可以使用 class。）

图 **5.6**：样式化 LOGO 后的示例程序

代码 **5.8**：添加网站 LOGO 的样式

app/assets/stylesheets/custom.css.scss

```
@import "bootstrap";
.
.
.

/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  line-height: 1;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

其中 `color: #fff;` 会把 LOGO 文字的颜色变成白色。HTML 中的颜色代码是由 3 个 16 进制数组成的，分别代表了三原色中的红、绿、蓝。`#ffffff` 是 3 种颜色都为最大值的情况，代表了纯白色。`#fff` 是 `#ffffff` 的简写形式。CSS 标准中为很多常用的 HTML 颜色定义了别名，例如 `white` 代表的是 `#fff`。代码 5.8 中的样式效果如图 5.6 所示。

5.1.3 局部视图

虽然代码 5.1 中的布局达到了目的，但它的内容看起来有点混乱。HTML shim 就占用了三行，而且使用了只针对 IE 的奇怪句法，所以如果能把它打包放在一个单独的地方就好了。头部的 HTML 自成一个逻辑单元，所以也可以把这部分打包放在某个地方。在 Rails 中我们可以使用局部视图来实现这种想法。先来看一下定义了局部视图之后的布局文件（参见代码 5.9）。

代码 5.9：定义了 HTML shim 和头部局部视图之后的网站布局
app/views/layouts/application.html.erb

```
<!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 %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

代码 5.9 中，我们把加载 HTML shim 的那几行代码换成了对 Rails 帮助函数 `render` 的调用：

```
<%= render 'layouts/shim' %>
```

这行代码会寻找一个名为 `app/views/layouts/_shim.html.erb` 的文件，执行文件中的代码，然后把结果插入视图。[6]（回顾一下，执行 Ruby 表达式并将结果插入模板中要使用 `<%=...%>`。）注意文件名 `_shim.html.erb` 的开头是个下划线，这个下划线是局部视图的命名约定，可以在目录中快速定位所有的局部视图。

当然，若要局部视图起作用，我们要写入相应的内容。本例中的 HTML shim 局部视图只包含三行代码，如代码 5.10 所示。

代码 5.10：HTML shim 局部视图
`app/views/layouts/_shim.html.erb`

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

类似的，我们可以把头部的内容移入局部视图，如代码 5.11 所示，然后再次调用 `render` 把这个局部视图插入布局中。

代码 5.11：网站头部的局部视图
`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", '#', id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home",    '#' %></li>
```

6. 很多 Rails 程序员都会使用 `shared` 目录存放需要在不同视图中共用的局部视图。我倾向于在 `shared` 目录中存放辅助的局部视图，而把每个页面中都会用到的局部视图放在`layouts` 目录中。（我们会在第 7 章创建 `shared` 目录。）在我看来，这样用比较符合逻辑，不过，都放在 `shared` 目录里对运作没有影响；

```
          <li><%= link_to "Help",    '#' %></li>
          <li><%= link_to "Sign in", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

现在我们已经知道怎么创建局部视图了，让我们来加入和头部对应的网站底部吧。你或许已经猜到了，我们会把这个局部视图命名为 `_footer.html.erb`，放在布局目录中（参见代码 5.12）。[7]

代码 5.12： 网站底部的局部视图
`app/views/layouts/_footer.html.erb`

```
<footer class="footer">
  <small>
    <a href="http://railstutorial.org/">Rails Tutorial</a>
    by Michael Hartl
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   '#' %></li>
      <li><%= link_to "Contact", '#' %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

和头部类似，在底部我们使用 `link_to` 创建到“关于”页面和“联系”页面的链接，地址暂时使用占位符 `#`。（和 `header` 一样，`footer` 标签也是 HTML5 新增加的。）

按照 HTML shim 和头部局部视图采用的方式，我们也可以在布局视图中渲染底部局部视图。（参见代码 5.13。）

代码 5.13： 网站的布局，包含底部局部视图
`app/views/layouts/application.html.erb`

```
<!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 %>
    <%= render 'layouts/shim' %>
```

7. 你可能想知道为什么要使用 `footer` 标签和 `.footer` class。理由就是，这样的标签对于人类来说更容易理解，而那个 class 是因为 Bootstrap 里在用，所以不得不用。如果愿意的话，用 `div` 标签代替 `footer` 也没什么问题；

```
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
    </div>
  </body>
</html>
```

当然，如果没有样式的话，底部还是很丑的（样式参见代码 5.14）。添加样式后的效果如图 5.7 所示。

代码 5.14：添加底部所需的 CSS
app/assets/stylesheets/custom.css.scss

```
.
.
.

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #999;
}

footer a {
  color: #555;
}

footer a:hover {
  color: #222;
}

footer small {
  float: left;
}

footer ul {
  float: right;
  list-style: none;
}

footer ul li {
```

```
  float: left;
  margin-left: 10px;
}
```

5.2 Sass 和 asset pipeline

在较新版本的 Rails 中，最大的变化是增加了 asset pipeline，这个功能可以明显提高如 CSS、JavaScript和图片等静态资源文件（asset）的生成效率，降低管理成本。本节我们会概览一下 asset pipeline，然后再介绍如何使用 Sass 这个生成 CSS 很强大的工具，Sass 现在是 asset pipeline 默认的一部分。

5.2.1 Asset pipeline

Asset pipeline 对 Rails做了很多改动，但对 Rails 开发者来说只有三个特性需要了解：资源目录，清单文件（manifest file），还有预处理器引擎（preprocessor engine）。[8]我们会一个一个的介绍。

图 **5.7**：添加底部后的“首页”（/static_pages/home）

8. 本节架构的依据是 Michael Erasmus 的《The Rails 3 Asset Pipeline in (about) 5 Minutes》一文。更多内容，请阅读 Rails 指南中的《Asset Pipeline》；

资源目录

在 Rails 3.0 之前（包括 3.0），静态文件分别放在如下的 `public/` 目录中：

- `public/stylesheets`
- `public/javascripts`
- `public/images`

这些文件夹中的文件通过请求 http://example.com/stylesheets 等地址直接发送给浏览器。（Rails 3.0 之后的版本也可以这么做。）

从 Rails 3.1 开始到最新的 Rails 4，静态文件可以存放在三个标准的目录中，各有各的用途：

- `app/assets`：存放当前应用程序用到的资源文件
- `lib/assets`：存放开发团队自己开发的代码库用到的资源文件
- `vendor/assets`：存放第三方代码库用到的资源文件

你可能猜到了，上面的目录中都会有针对不同资源类型的子目录。例如：

```
$ ls app/assets/
images javascripts stylesheets
```

现在我们就可以知道 5.1.2 节中 `custom.css.scss` 存放位置的用意：因为 `custom.css.scss` 是应用程序本身用到的，所以把它存放在 `app/assets/stylesheets` 中。

清单文件

当你把资源文件存放在适当的目录后，要通过清单文件告诉 Rails怎么把它们合并成一个文件（使用 Sprockets gem。只适用于 CSS 和 JavaScript，而不会处理图片。）举个例子，让我们看一下应用程序默认的样式表清单文件（参见代码 5.15）。

代码 5.15：应用程序的样式表清单文件
`app/assets/stylesheets/application.css`

```
/*
 * This is a manifest file that'll automatically include all the stylesheets
 * available in this directory and any sub-directories. You're free to add
 * application-wide styles to this file and they'll appear at the top of the
 * compiled file, but it's generally better to create a new file per style
 * scope.
 *= require_self
 *= require_tree .
*/
```

这里的关键代码是几行 CSS 注释，Sprockets 会通过这些注释加载相应的文件：

```
/*
 * .
 * .
 * .
 *= require_self
 *= require_tree .
*/
```

上面代码中的

```
*= require_tree .
```

会把 `app/assets/stylesheets` 目录中的所有 CSS 文件都引入应用程序的样式表中。

下面这行：

```
*= require_self
```

会把 `application.css` 这个文件中的 CSS 也加载进来。

Rails 提供的默认清单文件可以满足我们的要求，所以本书不会对其做任何修改。Rails 指南中有一篇专门介绍 asset pipeline 的文章，该文有你需要知道的更为详细的内容。

预处理器引擎

准备好资源文件后，Rails 会使用一些预处理器引擎来处理它们，通过清单文件将其合并，然后发送给浏览器。我们通过扩展名告诉 Rails 要使用哪个预处理器。三个最常用的扩展名是：Sass 文件的 `.scss`，CoffeeScript 文件的 `.coffee`，ERb 文件的 `.erb`。我们在 3.3.3 节介绍过 ERb，5.2.2 节会介绍 Sass。本教程不需要使用 CoffeeScript，这是一个很小巧的语言，可以编译成 JavaScript。（RailsCast 中关于 CoffeeScript 的视频是个很好的入门教程。）

预处理器引擎可以连接在一起使用，因此

```
foobar.js.coffee
```

只会使用 CoffeeScript 处理器，而

```
foobar.js.erb.coffee
```

会使用 CoffeeScript 和 ERb 处理器（按照扩展名的顺序从右向左处理，所以 CoffeeScript 处理器会先执行）。

在生产环境中的效率问题

Asset pipeline 带来的好处之一是，它会自动优化资源文件，在生产环境中使用效果极佳。CSS 和 JavaScript 的传统组织方式是将不同功能的代码放在不同的文件中，而且代码的格式是对人类友好的（有很多缩进）。虽然这对编程人员很友好，但在生产环境中使用却效率低下，加载大量的文件会明显增加页面加载时间（这是影响用户体验最主

要的因素之一）。使用 asset pipeline，生产环境中应用程序所有的样式都会集中到一个 CSS 文件中（`application.css`），所有 JavaScript 代码都会集中到一个 JavaScript 文件中（`javascript.js`），而且还会压缩这些文件（包括 `lib/assets` 和 `vendor/assets` 中的相关文件），把不必要的空格删除，减小文件大小。这样我们就最好的平衡了两方面的需求：编程人员使用格式友好的多个文件，生产环境中使用优化后的单个文件。

5.2.2 句法强大的样式表

Sass 是一种编写 CSS 的语言，从多方面增强了 CSS 的功能。本节我们会介绍两个最主要的功能，嵌套和变量。（还有一个是 mixin，会在 7.1.1 节中介绍。）

如 5.1.2 节中的简单介绍，Sass 支持一种名为 SCSS 的格式（扩展名为 `.scss`），这是 CSS 句法的一个扩展集。SCSS 只是为 CSS 添加了一些功能，而没有定义全新的句法。[9]也就是说，所有合法的 CSS 文件都是合法的 SCSS 文件，这对已经定义了样式的项目来说是件好事。在我们的程序中，因为要使用 Bootstrap，从一开始就使用了 SCSS。Rails 的 asset pipeline 会自动使用 Sass 预处理器处理扩展名为 `.scss` 的文件，所以 `custom.css.scss` 文件会首先经由 Sass 预处理器处理，然后引入程序的样式表中，再发送给浏览器。

嵌套

样式表中经常会定义嵌套元素的样式，例如，在代码 5.6 中，定义了 `.center` 和 `.centr h1` 两个样式：

```
.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

使用 Sass 可将其改写成

```
.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}
```

上面代码中的 `h1` 会自动嵌入 `.center` 中。

嵌套还有另一种形式，句法稍有不同。在代码 5.8 中，有如下的代码

9. Sass 仍然支持较早的 `.sass` 格式，这个格式相对来说更简洁，花括号更少，但是对现存项目不太友好，已经熟悉 CSS 的人学习难度也相对更大。

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  line-height: 1;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

其中 LOGO 的 id `#logo` 出现了两次，一次是单独出现的，另一次是和 `hover` 伪类一起出现的（鼠标悬停其上时的样式）。如果要嵌套第二个样式，我们需要引用父级元素 `#logo`，在 SCSS 中，使用 `&` 符号实现：

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  line-height: 1;
  &:hover {
    color: #fff;
    text-decoration: none;
  }
}
```

把 SCSS 转换成 CSS 时，Sass 会把 `&:hover` 编译成 `#logo:hover`。

这两种嵌套方式都可以用于代码 5.14 中的底部样式上，转换后的样式如下：

```
footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #999;
```

```
  a {
    color: #555;
    &:hover {
      color: #222;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 10px;
    }
  }
}
```

自己动手转换一下代码 5.14 是个不错的练习，转换之后你应该验证一下 CSS 是否还能正常使用。

变量

Sass 允许我们自定义变量来避免重复，这样也可以写出更具表现力的代码。例如，代码 5.7 和代码 5.14 中都重复使用了同一个颜色代码：

```
h2 {
  .
  .
  .
  color: #999;
}
.
.
.
footer {
  .
  .
  .
  color: #999;
}
```

上面代码中的 `#999` 是淡灰色（ligh gray），我们可以为它定义一个变量：

```
$lightGray: #999;
```

然后我们就可以这样写 SCSS：

```
$lightGray: #999;
.
.
.
h2 {
  .
  .
  .
  color: $lightGray;
}
.
.
.
footer {
  .
  .
  .
  color: $lightGray;
}
```

因为像 `$lightGray` 这样的变量名比 `#999` 更具说明性，所以为没有重复使用的值定义变量往往也是很有用的。Bootstrap 框架定义了很多颜色变量，Bootstrap 页面中有这些变量的 LESS 形式。这个页面中的变量使用的是 LESS 句法，而不是 Sass 句法，不过 bootstrap-sass gem 为我们提供了对应的 Sass 形式。二者之间的对应关系也不难猜出，LESS 使用 `@` 符号定义变量，而 Sass 使用 `$` 符号。在 Bootstrap 的变量页面我们可以看到为淡灰色定义的变量：

```
@grayLight: #999;
```

也就是说，在 bootstrap-sass gem 中会有一个对应的 SCSS 变量 `$grayLight`。我们可以用它换掉自己定义的 `$lightGray` 变量：

```
h2 {
  .
  .
  .
  color: $grayLight;
}
.
.
.
footer {
```

```
  .
  .
  .
  color: $grayLight;
}
```

使用 Sass 提供的嵌套和变量功能后得到的完整 SCSS 文件如代码 5.16 所示。这段代码中使用了 Sass 形式的颜色变量（参照 Bootstrap 变量页面中定义的 LESS 形式的颜色变量）和内置的颜色名称（例如，`white` 代表 `#fff`）[10]。请特别注意一下 `footer` 标签样式明显的改进。

代码 5.16：使用嵌套和变量转换后的 SCSS 文件
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";

/* mixins, variables, etc. */

$grayMediumLight: #eaeaea;

/* universal */

html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}
```

10. 译者注：一般不建议在 CSS 中使用颜色名称，因为不同的浏览器和不同的系统对同一个颜色的渲染有所不同，没有使用十六进制的颜色代码准确。

```
/* typography */

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: $grayLight;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}


/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: white;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  line-height: 1;
  &:hover {
    color: white;
    text-decoration: none;
  }
}
```

```
/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid $grayMediumLight;
  color: $grayLight;
  a {
    color: $gray;
    &:hover {
      color: $grayDarker;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 10px;
    }
  }
}
```

Sass 提供了很多功能，可以用来简化样式表，不过代码 5.16 只用到了最主要的功能，这是个好的开端。更多功能请查看 Sass 网站。

5.3 布局中的链接

我们已经为网站的布局定义了看起来还不错的样式，下面要把链接中暂时使用的占位符 # 换成真正的链接地址。当然，我们可以像下面这样手动加入链接：

```
<a href="/static_pages/about">About</a>
```

不过这样不太符合 Rails 风格。一者，“关于”页面的地址如果是 /about 而不是 /static_pages/about 就好了；再者，Rails 习惯使用具名路由（named route）来指定链接地址，相应的代码如下：

```
<%= link_to "About", about_path %>
```

使用这种方式能更好的表达链接与 URI 和路由的对应关系，如表格 5.1 所示。本章完结之前除了最后一个链接之外，其他的链接都会设定好。（第 8 章会添加最后一个。）

表格 5.1：网站中链接的路由和 URL 地址的映射关系

页面	URL	对应的路由
“首页”	/	`root_path`
“关于”	/about	`about_path`
“帮助”	/help	`help_path`
“联系”	/contact	`contact_path`
“注册”	/signup	`signup_path`
“登录”	/signin	`signin_path`

继续之前，让我们先添加一个“联系”页面（第 3 章的一个练习题），测试如代码 5.17 所示，形式和代码 3.19 差不多。

代码 5.17：“联系”页面的测试
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  describe "Contact page" do

    it "should have the content 'Contact'" do
      visit '/static_pages/contact'
      expect(page).to have_content('Contact')
    end

    it "should have the title 'Contact'" do
      visit '/static_pages/contact'
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | Contact")
    end
  end
end
```

为保证代码 5.17 中测试失败，我们要把页面底部的“联系”页面链接注释掉，如代码 5.18 所示。

代码 5.18：注释掉页面底部的“联系”页面链接
app/views/layouts/_footer.html.erb

```
<footer class="footer">
  <small>
    <a href="http://railstutorial.org/">Rails Tutorial</a>
```

```
    by Michael Hartl
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   '#' %></li>
      <li><%#= link_to "Contact", '#' %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

从代码 5.18 中可以得知在 ERb 中注释代码的方法，即在百分号后面加上 Ruby 中的注释符号。

```
<%#= link_to "Contact", '#' %>
```

现在，代码 5.17 中的测试应该是失败的：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

这里要采用的步骤和 3.2.2 节中添加“关于”页面的步骤是一致的：先更新路由设置（参见代码 5.19），然后在 StaticPages 控制器中添加 `contact` 动作（参见代码 5.20），最后再编写“联系”页面的视图（参见代码 5.21）。

代码 5.19：添加“联系”页面的路由设置
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  get "static_pages/about"
  get "static_pages/contact"
  .
  .
  .
end
```

代码 5.20：添加“联系”页面对应的动作
`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController
  .
  .
  .
  def contact
  end
end
```

代码 5.21：“联系”页面的视图
`app/views/static_pages/contact.html.erb`

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

再看一下测试是否可以通过：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

通过这个“遇红-变绿”的过程，证明了我们测试的目标是正确地，既然如此，就可以把页面底部“联系”页面链接的注释去掉了（参见代码 5.22）。不过这样做还不完美，如果不小心删除了代码 5.21 中的 h1 标签，测试是无法检测到的。要解决这个问题，就要编写一个针对 h1 标签内容的测试，测试的编写留作练习（参见 5.6 节）。

代码 5.22： 去掉底部“联系”页面链接的注释
app/views/layouts/_footer.html.erb

```
<footer class="footer">
  <small>
    <a href="http://railstutorial.org/">Rails Tutorial</a>
    by Michael Hartl
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   '#' %></li>
      <li><%= link_to "Contact", '#' %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

5.3.1 路由测试

静态页面的集成测试编写完之后，再编写路由测试就简单了：只需把硬编码的地址换成表格 5.1中相应的具名路由就可以了。也就是说，要把

```
visit '/static_pages/about'
```

修改为

```
visit about_path
```

其他的页面也这样做，修改后的结果如代码 5.23 所示。

代码 5.23： 具名路由测试

spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the h1 'Sample App'" do
      visit root_path
      expect(page).to have_content('Sample App')
    end

    it "should have the base title" do
      visit root_path
      expect(page).to have_title("Ruby on Rails Tutorial Sample App")
    end

    it "should not have a custom page title" do
      visit root_path
      expect(page).not_to have_title('| Home')
    end
  end

  describe "Help page" do

    it "should have the h1 'Help'" do
      visit help_path
      expect(page).to have_content('Help')
    end

    it "should have the title 'Help'" do
      visit help_path
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | Help")
    end
  end

  describe "About page" do

    it "should have the h1 'About Us'" do
      visit about_path
      expect(page).to have_content('About Us')
    end

    it "should have the title 'About Us'" do
      visit about_path
```

```
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | About Us")
    end
  end

  describe "Contact page" do

    it "should have the content 'Contact'" do
      visit contact_path
      expect(page).to have_content('Contact')
    end

    it "should have the title 'Contact'" do
      visit contact_path
      expect(page).to have_title("Ruby on Rails Tutorial Sample App | Contact")
    end
  end
end
```

和往常一样，现在应该看一下测试是否是失败的（红色）：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

顺便说一下，很多人都会觉得代码 5.23 有很多重复，也很啰嗦，我们会在 5.3.4 节进行重构。

5.3.2 Rails 路由

我们已经编写了针对所有 URL 地址的测试，现在就要实现这些地址了。如 3.1 节所说，Rails 使用 `config/routes.rb` 文件设置 URL 地址的映射关系。如果你看一下默认的路由文件，会发现内容很杂乱，不过还是能提供些帮助的，因为有很多注释，说明了各路由的映射关系。我建议你找个时间通读一下路由文件，也建议你阅读一下 Rails 指南中《详解 Rails 路由》一文，更深入的了解一下路由。

定义具名路由，要把

```
get 'static_pages/help'
```

修改为

```
match '/help', to: 'static_pages#help', via: 'get'
```

这样在 `/help` 地址上就有了一个可访问的页面（响应 `GET` 请求），也定义了一个名为 `help_path` 的具名路由，该函数会返回相应页面的地址。（其实把 `match` 换成 `get` 效果是一样的，不过使用 `match` 更符合约定。）

其他页面也要做类似修改，结果如代码 5.24 所示。不过“首页”有点特殊，参见代码 5.26。

代码 5.24： 静态页面的路由
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  match '/help',    to: 'static_pages#help',    via: 'get'
  match '/about',   to: 'static_pages#about',   via: 'get'
  match '/contact', to: 'static_pages#contact', via: 'get'
  .
  .
  .
end
```

如果认真阅读代码 5.24，或许会发现它的作用。例如，你会发现

```
match '/about', to: 'static_pages#about',  via: 'get'
```

会匹配到 `/about` 地址的 `GET` 请求，并将其分发到 StaticPages 控制器的 `about` 动作上。之前的设置意图更明显，我们用

```
get 'static_pages/about'
```

也可以得到相同的页面，不过 /about 的地址形式更简洁。而且，如前面提到的，`match '/about'` 会自动创建具名路由函数，可以在控制器和视图中使用：

```
about_path -> '/about'
about_url  -> 'http://localhost:3000/about'
```

注意，`about_url` 返回的结果是完整的 URL 地址 http://localhost:3000/about（部署后，会用实际的域名替换 `localhost:3000`，例如 `example.com`）。如 5.3 节的用法，如果只想返回 /about，使用 `about_path` 就可以了。本书基本上都会使用惯用的 `path` 形式，不过在页面转向时会使用 `url` 形式，因为 HTTP 标准要求转向后的地址为完整的 URL，不过大多数浏览器都可以正常使用这两种形式。

设置了这些路由之后，“帮助”页面、“关于”页面和“联系”页面的测试应该就可以通过了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

不过“首页”的测试还是失败的。

要设置“首页”的路由，可以使用如下的代码：

```
match '/', to: 'static_pages#home', via: 'get'
```

不过没必要这么做。Rails 在路由设置文件的下部为根地址 `/`（斜线）提供了特别的设置方式（参见代码 5.25）。

代码 5.25：注释掉的根路由设置说明
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  .
```

```
  .
  .
  # You can have the root of your site routed with "root"
  # root :to => "welcome#index"
  .
  .
  .
end
```

按照上述说明，我们把根地址 / 映射到“首页”上（参见代码 5.26）。

代码 5.26：添加根地址的路由设置
config/routes.rb

```
SampleApp::Application.routes.draw do
  root to: 'static_pages#home'
  match '/help',    to: 'static_pages#help',    via: 'get'
  match '/about',   to: 'static_pages#about',   via: 'get'
  match '/contact', to: 'static_pages#contact', via: 'get'
  .
  .
  .
end
```

上面的代码会把根地址 / 映射到 /static_pages/home 页面上，也就是说 http://localhost:3000/ 中的内容和图 3.1 中的 Rails 默认页面不一样了，同时还生成了两个 URL 地址帮助方法，如下所示：

```
root_path -> '/'
root_url  -> 'http://localhost:3000/'
```

至此，所有静态页面的路由都设置好了，而且所有测试应该都可以通过了：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

下面，我们要在布局中插入这些链接。

5.3.3 具名路由

现在要在布局中使用上一小节设置的路由帮助方法，把 `link_to` 函数的第二个参数设为相应的具名路由。例如，要把

```
<%= link_to "About", '#' %>
```

改为

```
<%= link_to "About", about_path %>
```

其他链接以此类推。

先从头部局部视图 `_header.html.erb` 开始，这个视图中包含了到“首页”和“帮助”页面的链接。既然要对头部视图做修改，顺便就按照网页的惯例为 LOGO 添加一个到“首页”的链接吧（参见代码 5.27）。

代码 5.27：头部局部视图，包含一些链接
`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home",    root_path %></li>
          <li><%= link_to "Help",    help_path %></li>
          <li><%= link_to "Sign in", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

第 8 章才会为“注册”页面设置具名路由，所以现在还是用占位符 # 代替该页面的地址。

还有一个包含链接的文件是底部局部视图 `_footer.html.erb`，有到“关于”页面和“联系”页面的链接（参见代码 5.28）。

代码 5.28：底部局部视图，包含一些链接
`app/views/layouts/_footer.html.erb`

```
<footer class="footer">
  <small>
    <a href="http://railstutorial.org/">Rails Tutorial</a>
    by Michael Hartl
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   about_path %></li>
      <li><%= link_to "Contact", contact_path %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

如此一来，第 3 章创建的所有静态页面的链接都加入布局了，以“关于”页面为例，输入 /about，就会进入网站的“关于”页面（如图 5.8）。

顺便说一下，要注意，虽然我们没有编写测试检测布局中是否包含这些链接，不过如果没有设置路由的话，前面的测试也会失败，不信你可以把代码 5.24 中的路由注释掉再运行测试来验证一下。检查链接是否指向正确页面的测试代码参见 5.6 节。

图 **5.8**：“关于”页面 /about

5.3.4 简化 RSpec 测试代码

在 5.3.1 节中说过，静态页面的测试有点啰嗦，也有些重复（参见代码 5.23）。本节我们就会使用一些最新的 RSpec 特性，把测试变得简洁一些、优雅一些。

先看一下如何改进下面的代码：

```
describe "Home page" do

  it "should have the content 'Sample App'" do
    visit root_path
    expect(page).to have_content('Sample App')
  end
```

```
  it "should have the base title" do
    visit root_path
    expect(page).to have_title("Ruby on Rails Tutorial Sample App")
  end

  it "should not have a custom page title" do
    visit root_path
    expect(page).not_to have_title('| Home')
  end
end
```

我们注意到，三个测试用例都访问了根地址，使用 `before` 块可以消除这个重复：

```
describe "Home page" do
  before { visit root_path }

  it "should have the content 'Sample App'" do
    expect(page).to have_content('Sample App')
  end

  it "should have the base title" do
    expect(page).to have_title("Ruby on Rails Tutorial Sample App")
  end

  it "should not have a custom page title" do
    expect(page).not_to have_title('| Home')
  end
end
```

上面的代码使用

```
before { visit root_path }
```

在每个测试用例运行之前访问根地址。（`before` 方法还可以使用别名 `before(:each)` 调用。）

还有个代码在每个用例中都出现了，我们使用了

```
it "should have the content 'Sample App'" do
```

同时还使用了

```
expect(page).to have_content('Sample App')
```

二者虽然形式不同，要表达的意思却是相同的。而且两个用例都引用了 page 变量。我们可以告诉 RSpec，page 就是要测试的对象（subject），这样就可以避免多次使用 page：

```
subject { page }
```

然后再使用 it 方法的另一种形式，把测试代码和描述文本合二为一：

```
it { should have_content('Sample App') }
```

因为指明了 subject { page }，所以调用 should 时就会自动使用 Capybara 提供的 page 变量（参见 3.2.1 节）。

使用这些技巧可以把“首页”的测试变得简洁一些：

```
  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_content('Sample App') }
    it { should have_title("Ruby on Rails Tutorial Sample App") }
    it { should_not have_title('| Home') }
  end
```

这样代码看起来就舒服多了，不过标题的测试还有点长。其实，代码 5.23 中大多数标题都是这样的长标题：

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

3.5 节的练习题建议定义一个 base_title 变量，再使用字符串插值来消除这个重复（参见代码 3.31）。我们可以更进一步，定义一个和代码 4.2 中 full_title 类似的方法。

为此我们要新建 spec/support 文件夹，然后在其中新建 RSpec 通用函数文件 utilities.rb（参见代码 5.29）。

代码 5.29：RSpec 通用函数文件，包含 full_title 方法
spec/support/utilities.rb

```
def full_title(page_title)
  base_title = "Ruby on Rails Tutorial Sample App"
  if page_title.empty?
    base_title
  else
    "#{base_title} | #{page_title}"
  end
end
```

其实这就是代码 4.2 中那个帮助方法的复制，不过，定义两个独立的方法可以捕获标题公共部分中的错误，其实这样也不太靠得住，更好的（也更强大的）方法是直接测试原来那个 `full_title` 帮助方法，参见 5.6 节中的练习。

RSpec 会自动加载 `spec/support` 目录中的文件，所以我们就可以按照如下的方式编写“首页”的测试：

```
  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_content('Sample App') }
    it { should have_title(full_title('')) }
  end
```

下面我们要用类似“首页”的方法来简化“帮助”页面、“关于”页面和“联系”页面的测试，结果如代码 5.30 所示。

代码 5.30：简化后的静态页面测试
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_content('Sample App') }
    it { should have_title(full_title('')) }
    it { should_not have_title('| Home') }
  end

  describe "Help page" do
    before { visit help_path }

    it { should have_content('Help') }
    it { should have_title(full_title('Help')) }
  end

  describe "About page" do
    before { visit about_path }

    it { should have_content('About') }
    it { should have_title(full_title('About Us')) }
  end
```

```
  describe "Contact page" do
    before { visit contact_path }

    it { should have_content('Contact') }
    it { should have_title(full_title('Contact')) }
  end
end
```

现在应该验证一下测试代码是否还可以通过：

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

代码 5.30 中的 RSpec 测试比代码 5.23 简化多了，其实，还可以变得更简洁，详见 5.6 节。在示例程序接下来的开发过程中，只要可以，我们都会使用这种简洁的方式。

5.4 用户注册：第一步

为了完结本章的目标，本节我们要设置“注册”页面的路由，为此要创建第二个控制器。这是允许用户注册最重要的一步，用户模型会在第 6 章构建，整个注册功能则会在第 7 章完成。

5.4.1 Users 控制器

创建第一个控制器 StaticPages 是很久以前的事了，还是在 3.1 节中。现在我们要创建第二个了，Users 控制器。和之前一样，我们使用 `generate` 命令创建所需的控制器骨架，包含用户注册页面所需的动作。遵照 Rails 的 REST 架构约定，我们把这个动作命名为 `new`，将其传递给 `generate controller` 就可以自动创建这个动作了（参见代码 5.31）。

代码 5.31：生成 Users 控制器（包含 new 动作）

```
$ rails generate controller Users new --no-test-framework
      create  app/controllers/users_controller.rb
       route  get "users/new"
      invoke  erb
      create    app/views/users
      create    app/views/users/new.html.erb
      invoke  helper
      create    app/helpers/users_helper.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
```

这个命令会创建 Users 控制器，还有其中的 new 动作（参见代码 5.32）和一个占位用的视图文件（参见代码 5.33）。

代码 5.32：默认生成的 Users 控制器，包含 new 动作
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def new
  end

end
```

代码 5.33：默认生成的 new 动作视图
app/views/users/new.html.erb

```
<h1>Users#new</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

5.4.2 “注册”页面的 URL 地址

5.4.1 节中生成的代码会在 /users/new 地址上对应一个页面，不过如表格 5.1所示，我们希望“注册”页面的地址是 /signup。为此，和 5.3 节一样，首先要编写集成测试，可以通过下面的命令生成：

```
$ rails generate integration_test user_pages
```

然后，按照代码 5.30 中静态页面测试代码的形式，我们要编写测试检测“注册”页面中是否有 h1 和 title 标签，如代码 5.34 所示。

代码 5.34：Users 控制器的测试代码，包含“注册”页面的测试用例
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "signup page" do
    before { visit signup_path }

    it { should have_content('Sign up') }
    it { should have_title(full_title('Sign up')) }
  end
end
```

和之前一样，可以执行 rspec 命令运行测试：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb
```

不过有一点要注意，你还可以指定整个目录来运行所有的 request 测试：

```
$ bundle exec rspec spec/requests/
```

同理，你可能还想知道怎么运行全部测试：

```
$ bundle exec rspec spec/
```

为了测试全面，在本书后续内容中，我们一般都会使用这个命令运行所有的测试。顺便说一下，你要知道，你也可以使用 Rake 的 `spec` 任务运行测试（你可能见过其他人这样使用）：

```
$ bundle exec rake spec
```

（事实上，你可以只输入 `rake`，因为 `rake` 的默认任务就是运行测试。）现在唯一的问题是，运行测试会报错，因为还没有准备好“测试数据库”，我们会在 6.2.1 节 中详细介绍。

我们已经为 Users 控制器生成了 `new` 动作，如要测试通过，需要正确设置路由，还要有相应内容的视图文件。我们按照代码 5.24 的方式，为“注册”页面加入 `match '/signup'` 路由设置（参见代码 5.35）。

代码 5.35：“注册”页面的路由设置
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  get "users/new"

  root to: 'static_pages#home'
  match '/signup',  to: 'users#new',            via: 'get'
  match '/help',    to: 'static_pages#help',    via: 'get'
  match '/about',   to: 'static_pages#about',   via: 'get'
  match '/contact', to: 'static_pages#contact', via: 'get'
  .
  .
  .
end
```

注意，我们保留了 `get "users/new"` 设置，这是控制器生成命令（代码 5.31）自动添加的路由，如要路由可用，这个设置还不能删除，不过这不符合 REST 约定（参见表格 2.2），会在 7.1.2 节删除。

要让测试通过，视图中还要有相应的 `h1` 和 `title`（参见代码 5.36）。

代码 5.36：“注册”页面视图
`app/views/users/new.html.erb`

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

现在，代码 5.34 中“注册”页面的测试应该可以通过了。下面要做的就是为“首页”中的注册按钮加上链接。和其他的具名路由一样，`match '/signup'` 会生成 `signup_path` 方法，用来链接到“注册”页面（参见代码 5.37）。

代码 5.37：把按钮链接到“注册”页面
`app/views/static_pages/home.html.erb`

```
<div class="center hero-unit">
  <h1>Welcome to the Sample App</h1>

  <h2>
    This is the home page for the
    <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </h2>

  <%= link_to "Sign up now!", signup_path, class: "btn btn-large btn-primary" %>
</div>

<%= link_to image_tag("rails.png", alt: "Rails"), 'http://rubyonrails.org/' %>
```

至此，除了没有设置“登录/退出”路由之外（第 8 章会实现），我们已经完成了添加链接和设置路由的任务。注册用户的页面（/signup）如图 5.9 所示。

图 **5.9**：“注册”页面 /signup

现在测试应该可以通过了：

```
$ bundle exec rspec spec/
```

5.5 小结

本章，我们为应用程序定义了一些样式，也设置了一些路由。本书剩下的内容会不断的充实这个示例程序：先添加用户注册、登录和退出的功能，然后实现微博功能，最后是关注用户功能。

如果使用 Git 的话，现在你应该把本章所做的改动合并到主分支中：

```
$ git add .
$ git commit -m "Finish layout and routes"
$ git checkout master
$ git merge filling-in-layout
```

还可以把代码推送到 GitHub 上：

```
$ git push
```

最后，你可以把应用程序部署到 Heroku：

```
$ git push heroku
```

然后在“生产环境”中就得到了一个可以运行的示例程序：

```
$ heroku open
```

如果遇到问题，运行

```
$ heroku logs
```

试着使用 Heroku 的日志文件排错。同时确保你已经做了代码 2.17 所示的改动。

5.6 练习

1. 从 5.3 节我们得知，RSpec 的 `have_content` 匹配器有时匹配的范围太宽了，所以我们必须注释掉底部的“联系”页面链接才能避免误伤（参见代码 5.18）。为了解决这个问题，代码 5.38 使用了 Capybara 提供的 `have_selector` 方法，可以直接用来测试页面中是否存在指定的 HTML 标签。我们可以先把代码 5.21 中的 `h1` 标签删掉，然后再加进来，验证一下代码 5.38 中的测试即使不注释掉底部的链接测试也会按预期失败。

2. 测试静态页面的代码 5.30 已经简化了，但还有一些重复的地方。RSpec 提供了一种名为“共享用例（shared example）”的辅助功能，可以消除这些重复。按照代码 5.39 的形式，添加没有编写的“帮助”页面、“关于”页面和“联系”页面的测试。注意，代码 3.31 中用到的 `let` 方法只要需要就会用指定的值创建一个局部变量（例如，引用这个变量时），相比之下，实例变量只在赋值时才被创建。（代码 5.39 也用到了 代码 5.38 中的 `have_selector`。）

3. 或许你已经注意到了，对布局中链接的测试，只测试了路由设置，而没有测试链接是否指向了正确的页面。实现这个测试的方法之一是使用 RSpec 集成测试中的 `visit` 和 `click_link` 函数。加入代码 5.40 中的测试来验证一下链接的地址是否正确。

4. 不要使用代码 5.29 中的 `full_title` 方法，另外编写一个用例，测试原来那个帮助方法，可参照代码 5.41。（需要新建 `spec/helpers` 目录和 `application_helper_spec.rb` 文件。）然后用代码 5.42 中的代码将其引入（`require`）测试。运行测试验证一下新代码是否可以正常使用。注意：代码 5.41 用到了正则表达式（regular expression），6.2.4 节会做介绍。（感谢 Alex Chaffee 的建议，并提供了本题用到的代码。）

代码 5.38：针对“联系”页面更有针对性的测试
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  describe "Contact page" do
    before { visit contact_path }

    it { should have_selector('h1', text: 'Contact') }
    it { should have_title(full_title('Contact')) }
  end
end
```

代码 5.39：用 RSpec “共享用例”来消除重复
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  subject { page }

  shared_examples_for "all static pages" do
    it { should have_content(heading) }
    it { should have_title(full_title(page_title)) }
  end

  describe "Home page" do
```

```
    before { visit root_path }
    let(:heading)    { 'Sample App' }
    let(:page_title) { '' }

    it_should_behave_like "all static pages"
    it { should_not have_title('| Home') }
  end

  describe "Help page" do
    .
    .
    .
  end

  describe "About page" do
    .
    .
    .
  end

  describe "Contact page" do
    .
    .
    .
  end
end
```

代码 5.40： 测试布局中的链接
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  it "should have the right links on the layout" do
    visit root_path
    click_link "About"
    expect(page).to have_title(full_title('About Us'))
    click_link "Help"
    expect(page).to # fill in
    click_link "Contact"
    expect(page).to # fill in
    click_link "Home"
    click_link "Sign up now!"
```

```
    expect(page).to # fill in
    click_link "sample app"
    expect(page).to # fill in
  end
end
```

代码 **5.41**：对 full_title 帮助方法的测试
spec/helpers/application_helper_spec.rb

```
require 'spec_helper'

describe ApplicationHelper do

  describe "full_title" do
    it "should include the page title" do
      expect(full_title("foo")).to match(/foo/)
    end

    it "should include the base title" do
      expect(full_title("foo")).to match(/^Ruby on Rails Tutorial Sample App/)
    end

    it "should not include a bar for the home page" do
      expect(full_title("")).not_to match(/\|/)
    end
  end
end
```

代码 **5.42**：使用一个简单的引用代替测试中的 full_title 方法
spec/support/utilities.rb

```
include ApplicationHelper
```

第 6 章 用户模型

第 5 章的末尾我们创建了一个临时的用户注册页面（5.4 节），本教程接下来的四章会逐步丰富这个页面的功能。第一个关键的步骤是为网站的用户创建一个数据模型，以及存储数据的方式。第 7 章会实现用户注册功能，并创建用户资料页面。用户能注册后，我们就要实现登录和退出功能（第 8 章）。第 9 章（9.2.1 节）会介绍如何保护页面避免被无权限的人员访问。第 6 章到第 9 章的内容结合在一起，我们就开发出了一个功能完整的 Rails 登录和用户验证系统。或许你知道已经有很多开发好了的 Rails 用户验证方案，旁注 6.1解释了为什么，至少在初学阶段，自己开发一个用户验证系统或许是更好的方法。

这一章很长，内容很多，你也许会觉得有些挑战性，特别是对数据模型新手来说。不过学完本章后，我们会开发出一个可在实际应用程序中使用的系统，包括数据验证、存储和用户信息获取等功能。

旁注 6.1：为什么要自己开发用户验证系统

基本上所有的 Web 应用程序都会需要某种登录和用户验证系统。所以 Web 框架大都有很多验证系统的实现方案，Rails 当然也不例外。用户验证及授权系统有很多，包括 Clearance、Authlogic、Devise 和 CanCan（还有一些不是专门针对 Rails 的基于 OpenID 和 OAuth 开发的系统）。所以你肯定就会问，为什么还要重复制造轮子，为什么不直接用现有的解决方案，而要自己开发呢？

首先，实践已经证明，大多数网站的用户验证系统都要对第三方代码库做一些定制和修改，这往往比重新开发一个验证系统的工作量还大。再者，现有的方案就像一个“黑盒”，你无法了解其中到底有些什么功能，而自己开发的话就能更好的理解实现的过程。而且，Rails 最近的更新（参见6.3 节），使开发验证系统变得很简单。最后，如果后续开发要用第三方代码库的话，因为自己开发过，所以你可以更好的理解其实现过程，便于定制功能。

和之前一样，如果你一直使用 Git 做版本控制，现在最好为本章创建一个从分支：

```
$ git checkout master
$ git checkout -b modeling-users
```

（第一个命令是要确保处在主分支，这样创建的 `modeling-users` 从分支才会基于 `master` 分支的当前状态。如果你已经处在主分支的话，可以跳过第一个命令。）

6.1 User 模型

接下来的三章最终是要实现网站的“注册”页面（构思图如图 6.1 所示），在此之前我们要解决用户的存储问题，因为现在还没有地方存储用户信息。所以，实现用户注册功能的第一步是，创建一个数据结构，获取并存储用户的信息。

图 6.1：用户注册页面的构思图

在 Rails 中，数据模型的默认数据结构叫做模型（model，MVC 中的 M，参见1.2.6 节）。Rails 为解决数据持久化提供的默认解决方案是，使用数据库存储需要长期使用的数据。和数据库交互默认的代码库是 Active Record[1]。Active Record 提供了一系列的方法，可直接用于创建、保存、查询数据对象，而无需使用关系数据库所用的结构化查询语言（structured query language， SQL）[2]。Rails 还支持数据库迁移（migration）功能，允许我们使用纯 Ruby 代码定义数据结构，而不用学习数据定义语言（data definition language, DDL）。Active Record 把你和数据库层完全隔开了。本教程开发的应用程序在本地使用的是 SQLite 数据库，部署后使用 PostgreSQL 数据库（由 Heroku 提供，参见1.4 节）。这就引入了一个更深层次的话题，那就是在不同的环境中，即便使用的是不同类型的数据库，我们也无需关心 Rails 是如何存储数据的。

1. Active Record 这个名字来自“Active Record 模式”，出自 Martin Fowler 的《企业应用架构模式》一书。
2. SQL 读作 ess-cue-ell，不过也经常读作 sequel。

6.1.1 数据库迁移

回顾一下 4.4.5 节的内容，在我们自己创建的 User 类中已经使用了用户对象，有 name 和 email 两个属性。那是个很有用的例子，但没有实现持久性最关键的要求：在 Rails 控制台中创建的用户对象，退出控制台后就会消失。本节的目的就是创建一个用户模型，持久化存储数据。

和 4.4.5 节中的 User 类一样，我们先把用户模型设计为只有两个属性，分别是 name 和 email。Email 地址稍后会作为用户登录的用户名。[3]（6.3 节会添加用户密码相关的属性）代码 4.9 使用 Ruby 的 attr_accessor 方法创建了这两个属性：

```
class User
  attr_accessor :name, :email
  .
  .
  .
end
```

在 Rails 中不用这样定义属性。如前面提到的，Rails 默认使用关系型数据库存储数据，数据库中的表（table）是由记录行（row）组成的，每行中都有相应的数据属性列（column）。例如，要存储用户的名字和 Email 地址，我们要创建 users 表，表中包含 name 列和 email 列（每一行代表一个用户）。这样命名列，Active Record 就能找到用户对象的属性了。

让我们看一下 Active Record 是怎么找到的。（如果下面这些讨论对你来说太抽象了，请耐心的看下去。6.1.3 节在控制台中的操作示例，以及图 6.3 和图 6.6 中的数据库浏览器截图应该会让你更清楚的理解这些内容。）代码 5.29 使用下面的命令生成了 Users 控制器和 new 动作：

```
$ rails generate controller Users new --no-test-framework
```

创建模型可以使用类似的命令：generate model。代码 6.1 所示为生成 User 模型，以及 name 和 email 属性所用的命令。

代码 6.1：生成用户模型

```
$ rails generate model User name:string email:string
      invoke active record
      create db/migrate/[timestamp]_create_users.rb
      create app/models/user.rb
      invoke rspec
      create   spec/models/user_spec.rb
```

（注意，和生成控制器的命令习惯不同，模型的名字是单数：控制器是 Users，而模型是 User。）我们提供了可选的参数 name:string 和 email:string，告诉 Rails 我们需要的两个属性是什么，以及各自的类型（两个都是字符串）。你可以把这两个参数与代码 3.4 和代码 5.29 中的动作名称对比一下，看看有什么不同。

3. 把 Email 作为用户名，以后如果需要和用户联系就方便了。

代码 6.1 中的 `generate` 命令带来的结果之一是创建了一个数据库迁移文件。迁移是一种精确修改数据库结构的方式，可以根据需求修改数据模型。本例中 User 模型的迁移文件是直接通过 `generate model` 命令生成的。迁移文件会创建 `users` 表，包含两个列，即 `name` 和 `email`，如代码 6.2 所示。（我们会在 6.2.5 节以及 6.3 节介绍如何手动创建迁移文件。）

代码 6.2： User 模型的迁移文件（创建 `users` 表）
`db/migrate/[timestamp]_create_users.rb`

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :email
      t.timestamps
    end
  end
end
```

注意一下迁移文件名的前面有个时间戳（timestamp），这是该文件被创建时的时间。早期迁移文件名的前缀是个递增的数字，在协作团队中如果多个编程人员生成了相同序号的迁移文件就可能会发生冲突。除非两个迁移文件在同一秒钟生成这种小概率事件发生了，否则使用时间戳基本可以避免冲突的发生。迁移文件的代码中有一个名为 `change` 的方法，这个方法定义要对数据库做什么操作。代码 6.2 中，`change` 方法使用 Rails 提供的 `create_table` 方法在数据库中新建一个表，用来存储用户数据。`create_table` 后跟着一个块，指定了一个块参数 `t`（代表这个表对象）。在块中，`create_table` 方法通过 `t` 对象创建了 `name` 和 `email` 两个列，均为 `string` 类型。[4]这里使用的表名是复数形式（`users`），不过模型名是单数形式（User），这是 Rails 在用词上的一个约定，即模型表现的是单个用户的特性，而数据库表却存储了很多用户。块中最后一行 `t.timestamps` 是个特殊的方法，它会自动创建两个列，`created_at` 和 `updated_at`，这两个列分别记录创建用户的时间戳和更新用户数据的时间戳。（6.1.3 节会介绍使用这两个列的例子。）这个迁移文件表示的完整数据模型如图 6.2 所示。

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime

图 6.2： 代码 6.2 生成的用户数据模型

我们可以使用如下的 `rake` 命令（参见旁注 2.1）来执行这个迁移（也叫“向上迁移”）：

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

（你可能还记得，我们在 2.2 节中使用过这个命令。）第一次运行 `db:migrate` 命令时会新建 `db/development.sqlite3`，这是 SQLite[5]数据库文件。若要查看数据库结构，可以使用 SQLite 数据库浏览器打开

4. 不要管 `t` 对象是怎么实现的，作为抽象层的东西，我们无需过多关心它的具体实现。你只要相信它可以完成指定的工作就行了。
5. SQLite 读作 ess-cue-ell-ite，不过倒是经常使用错误的读音 sequel-ite。

db/development.sqlite3 文件（如图 6.3 所示）。和图 6.2 中的模型对比之后，你会发现 id 列在迁移文件中没有定义。在 2.2 节 中介绍过，Rails 把这个列作为行的唯一标识符。

图 6.3： 用 SQLite 数据库浏览器查看刚创建的 users 表

大多数迁移，包括本教程中的所有迁移，都是可逆的，也就是说可以通过一个简单的 Rake 命令“向下迁移”，撤销之前的迁移操作，这个命令是 db:rollback：

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

（另外还有一个撤销迁移的方法请查看旁注 3.2。）这个命令会调用 drop_table 方法把 users 表从数据库中删除。我们之所以可以这么做，是因为 change 方法知道 create_table 的反操作是 drop_table，所以回滚时就会直接调用 drop_table 方法。对于一些无法自动逆转的操作，例如删除列，就不能依赖 change 方法了，我们要分别定义 up 和 down 方法。关于迁移的更多内容请查看 Rails 指南。

如果你执行了上面的回滚操作，在继续阅读之前请再迁移回来：

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

6.1.2 模型文件

我们介绍了怎样使用代码 6.1 生成的迁移文件（参见代码 6.2）创建用户模型，在图 6.3 中看到了执行迁移操作后的结果，即修改了 `db/development.sqlite3`，新建 `users` 表，并创建了 `id`、`name`、`email`、`created_at` 和 `updated_at` 这几个列。代码 6.1 同时也生成了模型文件，本节的目的就是来理解一下这个模型文件。

我们先从 User 模型的代码说起。模型文件存放在 `app/models` 目录下，名为 `user.rb`。这个文件的内容很简单（参见代码 6.3）。

代码 6.3：刚创建的 User 模型文件
`app/models/user.rb`

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

在 4.4.2 节中介绍过，`class User < ActiveRecord::Base` 的意思是 `User` 类继承自 `ActiveRecord::Base` 类，所以 User 模型就自动获得了 `ActiveRecord::Base` 提供的功能。当然了，只知道这种继承关系没什么用，我们并不知道 `ActiveRecord::Base` 做了什么。稍后我们会介绍一下这个类。在继续阅读之前，还有两件事要做。

6.1.3 创建用户对象

和 第 4 章一样，探索数据模型我们使用的工具是 Rails 控制台。因为我们还不想改动数据库，所以我们要在沙盒模式（sandbox）中启动控制台：

```
$ rails console --sandbox
Loading development environment in sandbox
Any modifications you make will be rolled back on exit
>>
```

如提示信息所说，“Any modifications you make will be rolled back on exit”，在沙盒模式下使用控制台，退出当前会话后，对数据库做的所有改动都会回归到原来的状态。

4.4.5 节中的控制台会话，需要手动加载自己编写的 `User` 类才能使用 `User.new` 创建用户对象。创建模型后，情况就不一样了。4.4.4 节中介绍过，Rails 控制台会自动加载 Rails 环境，当然也会自动加载模型。也就是说，现在无需加载额外的代码就可以直接创建用户对象了：

```
>> User.new
=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>
```

上述的的控制台显示了一个用户对象的默认值。

如果不为 `User.new` 指定参数，对象的所有属性值都是 `nil`。在 4.4.5 节中，自己编写的 `User` 类可以接受一个 Hash 参数用来初始化对象的属性，这种方式是受 Active Record 启发的，在 Active Record 中也可以使用相同的方式指定初始值：

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User id: nil, name: "Michael Hartl", email: "mhartl@example.com",
created_at: nil, updated_at: nil>
```

我们看到 name 和 email 属性的值都已经设定了。

如果查看开发日志的话，会发现没有什么新内容。这是因为 User.new 方法并没有改动数据库，只是在内存中创建了一个用户对象。如果要把用户对象保存到数据库中，要在 user 对象上调用 save 方法：

```
>> user.save
=> true
```

如果保存成功，save 方法会返回 true；否则返回 false。（现在所有的保存操作都会成功[6]，6.2 节会看到保存失败的例子。）保存成功后，日志文件中就会出现一行 SQL 语句：INSERT INTO "users"。Active Record 提供了很多方法用来和数据库交互，所以你根本不需要直接使用SQL 语句，今后我也不会介绍 SQL 相关的知识。如果你想学习 SQL，可以查看日志文件。

你可能注意到了，刚创建时用户对象的 id、created_at 和 updated_at 属性值都是 nil，下面看一下保存之后有什么变化：

```
>> user
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">00:57:46",
updated_at: "2011-12-05 00:57:46">
```

如上述代码所示，id 的值变成 1 了，那两个自动创建的时间戳属性也变成了当前的时间。[7]现在这两个时间戳是一样的，6.1.5 节会看到二者不同的情况。

和 4.4.5 节中的 User 类一样，User 模型的实例也可以使用点号获取属性：[8]

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "mhartl@example.com"
>> user.updated_at
=> Mon, 11 Mar 2013 00:57:46 UTC +00:00
```

6. 译者注：因为还没加入数据验证
7. 你可能对 "2013-03-11 00:57:46" 这个时间感到好奇，为什么作者半夜还在写书呢？其实不然，这个时间戳是标准协时（Coordinated Universal Time, UTC），类似于格林尼治平时（Greenwich Mean Time, GMT）。以下摘自美国国家标准技术研究所时间和频率司的 FQA 网页。问：为什么标准协时的缩写是 UTC 而不是 CUT？答：标准协时是在 1970 年由国际电信联盟（ITU）的专家顾问团设计的，ITU 觉得应该使用一个通用的缩写形式避免混淆，因为各方无法达成共识，最终 ITU 没有采用英文缩写 CUT 或法文缩写 TUC，而是折中选择了 UTC。
8. 注意 user.updated_at 的值，上面说过了，这是 UTC 时间。

在第 7 章中会介绍，虽然一般习惯把创建和保存分成如上所示的两步分别操作，不过 Active Record 也允许我们使用 `User.create` 方法把这两步合成一步：

```
>> User.create(name: "A Nother", email: "another@example.org")
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2013-03-11 01:05:24", updated_at: "2013-03-11 01:05:24">
>> foo = User.create(name: "Foo", email: "foo@bar.com")
#<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-03-11
01:05:42", updated_at: "2013-03-11 01:05:42">
```

注意，`User.create` 的返回值不是 `true` 或 `false`，而是返回创建的用户对象，可直接赋值给变量（例如上面第二个命令中的 `foo` 变量）.

`create` 的反操作是 `destroy`：

```
>> foo.destroy
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-03-11
01:05:42", updated_at: "2013-03-11 01:05:42">
```

奇怪的是，`destroy` 和 `create` 一样，返回值是销毁的对象。我不觉得什么地方会用到 `destroy` 的返回值。更奇怪的事情是，销毁的对象还在内存中：

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-03-11
01:05:42", updated_at: "2013-03-11 01:05:42">
```

那么我们怎么知道对象是否真的被销毁了呢？对于已经保存而没有销毁的对象，怎样从数据库中获取呢？要回答这些问题，我们要先学习如何使用 Active Record 查找用户对象。

6.1.4 查找用户对象

Active Record 为查找对象提供了好几种方法。我们要使用这些方法来查找刚创建的第一个用户，同时也验证一下第三个用户（`foo`）是否被销毁了。先看一下还存在的用户：

```
>> User.find(1)
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
```

我们把用户的 id 传递给 `User.find` 方法，Active Record 会返回 id 为1 的用户对象。

下面来看一下 id 为 3 的用户是否还在数据库中：

```
>> User.find(3)
=> #ActiveRecord::RecordNotFound: Couldn't find User with ID=3
```

因为在 6.1.3 节中销毁了第三个用户，所以 Active Record 无法在数据库中找到，抛出了一个异常，说明在查找过程中出现了问题。因为 id 不存在，所以 `find` 方法才会抛出 `ActiveRecord::RecordNotFound` 异常。[9]

除了 `find` 方法之外，Active Record 还支持指定属性来查找用户：

```
>> User.find_by_email("mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
```

`find_by_email` 方法是 Active Record 根据 `users` 表中的 `email` 列自动定义的（你可能猜到了，Active Record 还定义了 `find_by_name` 方法）。从 Rails 4.0 开始，通过属性查找记录推荐的做法是使用 `find_by` 方法，属性值以 Hash 形式传入：

```
>> User.find_by(email: "mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
```

因为我们会把 Email 地址当成用户名使用，所以在实现用户登录功能时，这种查找用户的方式会很有用（参见第 7 章）。你也许会担心如果用户数量过多，使用 `find_by` 的效率不高。事实的确如此，我们会在 6.2.5 节介绍这个问题，以及解决方法，即使用数据库索引。

最后我们还要介绍几个常用的用户查找方法。首先是 `first` 方法：

```
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
```

很明显，`first` 会返回数据库中的第一个用户。还有 `all` 方法：

```
>> User.all
=> [#<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">,
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2013-03-11 01:05:24", updated_at: "2013-03-11 01:05:24">]
```

`all` 方法会返回一个数组，包含数据库中的所有用户。

6.1.5 更新用户对象

创建对象后，一般都会进行更新操作。更新有两种基本的方式，其一，我们可以分别为各属性赋值，在 4.4.5 节中就是这么做的：

9. 异常和异常处理是 Ruby 语言的高级功能，本书基本不会涉及这二者。不过异常是 Ruby 语言中很重要的一部分，建议你通过 1.1.1 节中推荐的书籍学习。

```
>> user            # Just a reminder about our user's attributes
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
>> user.email = "mhartl@example.net"
=> "mhartl@example.net"
>> user.save
=> true
```

注意，最后一个命令是必须的，我们要把改动写入数据库。我们可以执行 `reload` 命令来看一下没保存的话是什么情况。`reload` 命令会使用数据库中的数据重新加载对象：

```
>> user.email
=> "mhartl@example.net"
>> user.email = "foo@bar.com"
=> "foo@bar.com"
>> user.reload.email
=> "mhartl@example.net"
```

现在我们已经更新了用户数据，如在 6.1.3 节中所说，自动创建的那两个时间戳属性就不一样了：

```
>> user.created_at
=> "2013-03-11 00:57:46"
>> user.updated_at
=> "2013-03-11 01:37:32"
```

更新数据的第二种常用方式是使用 `update_attributes` 方法：

```
>> user.update_attributes(name: "The Dude", email: "dude@abides.org")
=> true
>> user.name
=> "The Dude"
>> user.email
=> "dude@abides.org"
```

`update_attributes` 方法可接受一个指定对象属性的 Hash 作为参数，如果操作成功，会执行更新和保存两个命令（保存成功时返回值为 `true`）。注意，如果任何一个数据验证失败了，例如存储记录时需要密码（会在 6.3 节实现），`update_attributes` 操作就会失败。如果要更新单个属性，请使用 `update_attribute`，这样就能跳过验证限制了：

```
>> user.update_attribute(:name, "The Dude")
=> true
>> user.name
=> "The Dude"
```

6.2 用户数据验证

6.1 节创建的 User 模型现在已经有了可以使用的 `name` 和 `email` 属性，不过功能还很简单：任何字符串（包括空字符串）都可以使用。名字和 Email 地址的格式显然要复杂一些。例如，`name` 不应该是空白的，`email` 应该符合 Email 地址的特定格式。而且，我们要把 Email 地址当成用户名用来登录，那么某个 Email 地址在数据库中就应该是唯一的。

总之，`name` 和 `email` 不是什么字符串都可以接受的，我们要对二者可接受的值做个限制。Active Record 是通过数据验证机制（validation）实现这种限制的。本节，我们会介绍几种常用的数据验证：存在性、长度、格式和唯一性。在 6.3.4 节还会介绍另一种常用的数据验证：二次确认。7.3 节会看到，如果提交了不合要求的数据，数据验证机制会显示一些很有用的错误提示信息。

6.2.1 用户模型测试

和开发示例程序的其他功能一样，我们会使用 TDD 方式添加 User 模型的数据验证功能。因为生成 User 模型时没有指定 `--no-test-framework` 旗标（和代码 5.29 不同），代码 6.1 中的命令会自动为 User 模型生成一个初始的测试文件，不过文件中没有什么内容（参见代码 6.4）。

代码 6.4： 几乎没什么内容的 User 模型测试文件
`spec/models/user_spec.rb`

```
require 'spec_helper'
describe User do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

上述代码使用了 `pending` 方法，提示我们应该编写一些真正有用的测试。我们可以准备一个空“测试数据库”然后运行 User 模型的测试看一下现在的情况：

```
$ bundle exec rake db:migrate
$ bundle exec rake test:prepare
$ bundle exec rspec spec/models/user_spec.rb
*


Finished in 0.01999 seconds
1 example, 0 failures, 1 pending

Pending:
  User add some examples to (or delete)
  /Users/mhartl/rails_projects/sample_app/spec/models/user_spec.rb
  (Not Yet Implemented)
```

在大多数系统中，有待实现的测试都会显示为黄色，介于通过（绿色）和失败（红色）之间。

这是我们第一次看到创建“测试数据库”的正确命令：

```
$ bundle exec rake test:prepare
```

这个命令会把“开发数据库” `db/development.sqlite3` 中的数据模型复制到“测试数据库” `db/test.sqlite3` 中。（执行迁移后很多人经常都会忘记执行这个 Rake 任务。而且，“测试数据库”经常会损坏，需要还原。如果测试出现了莫名其妙的问题，可以执行 `rake test:prepare` 看能否解决。）

我们会按照提示的建议，编写一些 RSpec 测试用例，如代码 6.5 所示。

代码 6.5：针对 `:name` 和 `:email` 属性的测试
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before { @user = User.new(name: "Example User", email: "user@example.com") }

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
end
```

`before` 块，在代码 5.28 中用过，会在各测试用例之前执行块中的代码，本例中这个块的作用是为 `User.new` 传入一个合法的初始 Hash 参数，创建 `@user` 实例变量。接下来的

```
subject { @user }
```

把 `@user` 设为这些测试用例默认的测试对象。在 5.3.4 节中设定的测试对象是 `page` 变量。

代码 6.5 中的两个测试用例对 `name` 和 `email` 属性的存在性进行了测试：

```
it { should respond_to(:name) }
it { should respond_to(:email) }
```

其实，这两个测试用例使用的是 Ruby 的 `respond_to?` 方法，这个方法可以接受一个 Symbol 参数，如果对象可以响应指定的方法或属性就返回 `true`，否则返回 `false`：

```
$ rails console --sandbox
>> user = User.new
>> user.respond_to?(:name)
=> true
>> user.respond_to?(:foobar)
=> false
```

（在 4.2.3 节中介绍过，Ruby 使用问号标明返回值是布尔值的方法。）这些测试使用了 RSpec 关于布尔值的约定，所以如下的代码

```
@user.respond_to?(:name)
```

在 RSpec 中可以写成

```
it "should respond to 'name'" do
  expect(@user).to respond_to(:name)
end
```

因为指定了 `subject { @user }`，我们还可以写成在 5.3.4 节中用过的单行形式：

```
it { should respond_to(:name) }
```

为 User 模型添加新方法或新属性时可以采用这种测试方式，而且使用这种方式还能清晰的列出 `User` 实例对象可以响应的所有方法。

因为我们前面已经执行 `rake test:prepare` 命令准备好了“测试数据库”，所以测试应该是通过的：

```
$ bundle exec rspec spec/
```

6.2.2 验证存在性

存在性验证算是最基本的验证了，它只是检查给定的属性是否存在。本节我们就会确保在用户存入数据库之前，名字和 Email 地址字段都是存在的。7.3.3 节会介绍如何把这个限制应用在创建用户的注册表单中。

我们先来编写检查 `name` 属性是否存在的测试。虽然 TDD 的第一步是写一个失败测试（参见 3.2.1 节），不过此时我们还没有完全理解数据验证是怎么实现的，无法编写合适的测试，所以我们暂时先把数据验证加进来，在控制台中实操一下，理解一下验证机制。然后我们会把验证代码注释掉，编写失败测试，再把注释去掉，看一下测试是否还是可以通过的。这么做对这样简单的测试来说可能有点小题大做，不过我见过很多“简单”的测试并没有检查真正要测试的内容。要保证测试检测真正需要检测的内容，最好谨慎一点。（这种加注释的方法在为没有测试的应用程序编写测试时也经常用到。）

验证 `name` 属性是否存在的方法是使用 `validates` 方法，传入 `presence: true` 参数，如代码 6.6 所示。参数 `presence: true` 是只有一个元素的可选 Hash 参数。我们在 4.3.4 节中介绍过，如果方法的最后一个参数是 Hash 的话，可以省略 Hash 的花括号。（如 5.1.1 节中提到的，这样的可选 Hash 在 Rails 中很普遍。）

代码 6.6：验证 `name` 属性的存在性
`app/models/user.rb`

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

代码 6.6 看起来很神奇，其实 `validates` 只不过是一个方法而已。加上括号后等效的代码如下：

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

下面在控制台中看一下，为 User 模型添加这个验证之后的效果：[10]

```
$ rails console --sandbox
>> user = User.new(name: "", email: "mhartl@example.com")
>> user.save
=> false
>> user.valid?
=> false
```

`user.save` 的返回值是 `false`，说明存储失败了。最后一个命令使用了 `valid?` 方法，如果对象没有通过任意一个验证就会返回 `false`，如果全部验证都通过了则返回 `true`。这个例子只有一个验证，所以我们知道是哪个验证失败了，不过看一下失败的提示信息还是会有点收获的：

```
>> user.errors.full_messages
=> ["Name can't be blank"]
```

（错误提示信息暗示了 Rails 中属性存在性验证使用的是 `blank?` 方法，在 4.4.3 节的末尾用过这个方法。）

现在来编写测试用例。为了保证测试是失败的，我们要把验证用到的代码注释掉，如代码 6.7 所示。

代码 6.7：注释掉验证代码，保证测试是失败的
`app/models/user.rb`

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

对存在性验证的测试用例如代码 6.8 所示。

代码 6.8：验证 `name` 属性的失败测试
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end

  subject { @user }
```

10. 如果控制台中命令的输出没什么实际意义，我就会省略掉。例如 `User.new` 的输出。

```
  it { should respond_to(:name) }
  it { should respond_to(:email) }

  it { should be_valid }

  describe "when name is not present" do
    before { @user.name = " " }
    it { should_not be_valid }
  end
end
```

如下这个新加入的测试是为了保证测试的全面，确保 `@user` 对象开始时是合法的：

```
it { should be_valid }
```

这是 6.2.1 节中介绍的 RSpec 对布尔值约定的又一个实例：只要对象可以响应返回值为布尔值的方法 `foo?`，就有一个对应的 `be_foo` 可以在测试中使用。所以在本例中，我们可以把如下的测试

```
@user.valid?
```

写成

```
it "should be valid" do
  expect(@user).to be_valid
end
```

和之前一样，因为指定了 `subject { @user }`，所以就可以使用单行形式了：

```
it { should be_valid }
```

第二个测试用例先把用户的名字设为不合法的值（空格），然后测试 `@user` 对象是否是不合法的：

```
describe "when name is not present" do
  before { @user.name = " " }
  it { should_not be_valid }
end
```

这段代码使用 `before` 块把用户的名字设为一个不合法的值（空格），然后检查用户对象是否为不合法的。

现在你可以看一下测试是否是失败的：

```
$ bundle exec rspec spec/models/user_spec.rb
...F
4 examples, 1 failure
```

现在去掉验证代码前的注释（把代码 6.7 变回代码 6.6），测试就可以通过了：

```
$ bundle exec rspec spec/models/user_spec.rb
...
4 examples, 0 failures
```

接着，我们还要验证 Email 地址的存在性，测试代码和对 name 属性的测试类似，参见代码 6.9。

代码 6.9：对 email 属性存在性的测试
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email is not present" do
    before { @user.email = " " }
    it { should_not be_valid }
  end
end
```

验证的实现几乎也一样，如代码 6.10 所示。

代码 6.10：验证 name 和 email 属性的存在性
app/models/user.rb

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

现在所有的测试应该都可以通过了，我们也就完成了存在性验证。

6.2.3 长度验证

我们已经对 User 模型可接受的数据做了一些限制，现在必须为用户提供一个名字，不过我们应该做的更进一步。用户的名字会在示例程序中显示，所以最好限制一下它的字符长度。有了 6.2.2 节的基础，这一步就简单了。

我们先来编写测试。最大长度并没有比较科学的方法来选定，我们就使用 50 作为长度的上限吧，那么我们就要确保 51 个字符超长了（参见代码 6.11）。

代码 6.11：对名字长度的测试
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when name is too long" do
    before { @user.name = "a" * 51 }
    it { should_not be_valid }
  end
```

为了方便，在代码 6.11 中我们使用字符串连乘生成了一个有 51 个字符的字符串。在控制台中可以看到连乘是什么：

```
>> "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> ("a" * 51).length
=> 51
```

代码 6.11 中的测试应该会失败。为了让测试通过，我们要知道怎么使用限制长度的参数 `:length`，以及限制上限的 `:maximum` 参数。验证相关的代码如代码 6.12 所示。

代码 6.12：为 `name` 属性添加长度验证
`app/models/user.rb`

```
class User < ActiveRecord::Base
  validates :name,  presence: true, length: { maximum: 50 }
  validates :email, presence: true
end
```

现在测试应该可以通过了。既然测试又通过了，我们要实现另一个更具挑战性的验证了，即对 Email 格式的验证。

6.2.4 格式验证

对 `name` 属性的验证只需做一些简单的限制就好，任何非空、长度小于 51 个字符的字符串就可以。不过 `email` 属性则需要更复杂的限制。目前我们只是拒绝空的 Email 地址，本节我们要限制 Email 地址符合常用的形式，类似 `user@example.com` 这种。

这里我们用到的测试和验证不是十全十美的，只是刚刚好可以接受大多数的合法 Email 地址，并拒绝大多数不合法的 Email 地址。我们会先对一些合法的 Email 集合和不合法的 Email 集合进行测试。我们使用 `%w[]` 来创建集合，集合中的元素都是字符串形式，如下面的控制台会话所示：

```
>> %w[foo bar baz]
=> ["foo", "bar", "baz"]
>> addresses = %w[user@foo.COM THE_US-ER@foo.bar.org first.last@foo.jp]
=> ["user@foo.COM", "THE_US-ER@foo.bar.org", "first.last@foo.jp"]
>> addresses.each do |address|
?> puts address
>> end
user@foo.COM
THE_US-ER@foo.bar.org
first.last@foo.jp
```

在上面这个控制台会话中，我们使用 each 方法（参见 4.3.2 节）遍历 address 数组中的元素。使用 each 方法可以编写一些简单的 Email 格式验证测试用例（参见代码 6.13）。

代码 6.13：对 Email 格式验证的测试
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email format is invalid" do
    it "should be invalid" do
      addresses = %w[user@foo,com user_at_foo.org example.user@foo.
                     foo@bar_baz.com foo@bar+baz.com]
      addresses.each do |invalid_address|
        @user.email = invalid_address
        expect(@user).not_to be_valid
      end
    end
  end

  describe "when email format is valid" do
    it "should be valid" do
      addresses = %w[user@foo.COM A_US-ER@f.b.org frst.lst@foo.jp a+b@baz.cn]
      addresses.each do |valid_address|
        @user.email = valid_address
        expect(@user).to be_valid
      end
    end
```

```
  end
end
```

如前所说，这些测试并不完美，不过我们测试了一些常用的合法的 Email 格式，例如 `user@foo.COM`、`THE_USER@foo.bar.org`（包含大写字母、下划线，以及子域名）和 `first.last@foo.jp`（用户名是合成的 `first.last`，顶级域名是两个字母形式的 `jp`（代表日本）），也测试了一些不合法的 Email 格式。

在应用程序中，我们会使用正则表达式定义要验证的 Email 格式，然后通过 `validates` 方法的 `:format` 参数指定合法的格式（参见代码 6.14）。

代码 6.14： 使用正则表达式验证 Email 格式
`app/models/user.rb`

```
class User < ActiveRecord::Base
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX }
end
```

我们把这个正则表达式定义为常量 `VALID_EMAIL_REGEX`，Ruby 中的常量都是以大写字母开头的。

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
validates :email, presence: true, format: { with: VALID_EMAIL_REGEX }
```

使用上面的代码可以确保只有匹配这个正则表达式的 Email 地址才是合法的。（因为 `VALID_EMAIL_REGEX` 以大写字母开头，是个常量，所以其值是不能改变的。）

那么，这个正则表达式是怎么编写出来的呢？正则表达式中的文本匹配模式是由简练的语言编写的（很多人会觉得很难读懂），学习如何编写正则表达式是一门艺术，为了便于理解，我会把 `VALID_EMAIL_REGEX` 拆分成几块来讲解（如表格 6.1所示）。[11]要想认真学习正则表达式，我推荐使用 Rubular 正则表达式编辑器（如图 6.4），这个工具在学习的过程中是必备的。[12]Rubular 网站的界面很友好，便于编写所需的正则表达式，网站中还有一个便捷的快速语法参考。我建议你使用 Rubular 来理解表格 6.1中的正则表达式片段。读的再多也不比不上在 Rubular 中实操几次。（注意：如果你在 Rubular 中输入代码 6.14 中用到的正则表达式，要把 `\A` 和 `\z` 去掉。）

表格 6.1：拆分代码 6.14 中匹配 Email 地址的正则表达式

表达式	含义
`/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i`	完整的正则表达式
`/`	正则表达式开始
`\A`	匹配字符串的开头

11. 注意，在表格 6.1 中，当我说到“字母”时，我的实际意思是指“小写字母”，因为后面的 `i` 已经指定了不区分大小写的模式，所以也没必要分的这么细了。
12. 如果你和我一样觉得 Rubular 很有用，我建议你向作者 Michael Lovitt 适当的捐献一些钱，以感谢他的辛勤劳动。

表达式	含义
`[\w+\-.]+`	一个或多个字母、加号、连字符、或点号
`@`	匹配 @ 符号
`[a-z\d\-.]+`	一个或多个小写字母、数字、连字符或点号
`\.`	匹配点号
`[a-z]+`	一个或多个小写字母
`\z`	匹配字符串结尾
`/`	结束正则表达式
`i`	不区分大小写

图 6.4： 强大的 Rubular 正则表达式编辑器

顺便说一下，在官方标准中确实有一个正则表达式可以匹配所有的合法 Email 地址，但没必要使用这么复杂的正则表达式，代码 6.14 中使用的正则表达式就很好，甚至可能比官方的更好。[13]不过，上面的正则表达式有一个缺陷，能匹配 `foo@bar..com` 这种有连续点号的地址。修正这个瑕疵的方法会留作练习，参见 6.5 节。

13. 你知道吗，根据 Email 标准，`"Michael Hartl"@example.com` 虽有引号和空格，但也是合法的 Email 地址，很不可思议吧。

现在，测试应该都可以通过了。（其实，对合法 Email 地址的测试一直都是可以通过的，因为正则表达式很容易出错，进行合法 Email 格式测试只是为了检测 `VALID_EMAIL_REGEX` 是否可用。）那么就只剩一个限制要实现了：确保 Email 地址的唯一性。

6.2.5 唯一性验证

确保 Email 地址的唯一性（这样才能作为用户名），要使用 `validates` 方法的 `unique` 参数。提前说明，实现的过程中存在一个很大的陷阱，所以不要轻易的跳过本小节，要认真的阅读。

和之前一样，先来编写测试。之前的模型测试，只是使用 `User.new` 在内存中创建一个对象，而做唯一性测试则要把数据存入数据库中。[14]对相同 Email 地址的（第一个）测试如代码 6.15 所示。

代码 6.15： 拒绝相同 Email 地址的测试
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email address is already taken" do
    before do
      user_with_same_email = @user.dup
      user_with_same_email.save
    end

    it { should_not be_valid }
  end
end
```

我们使用 `@user.dup` 方法创建一个和 `@user` Email 地址一样的用户对象，然后存储这个用户，因为数据库中的 `@user` 已经占用了这个 Email 地址，所以不能成功存入，无法得到一个合法的用户记录。

代码 6.16 中的代码可以让代码 6.15 中的测试通过。

代码 6.16： 验证 Email 地址的唯一性
`app/models/user.rb`

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

14. 如本节介绍中所说的，这里就要用到测试数据库 `db/test.sqlite3` 了。

```
  .
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
uniqueness: true
end
```

这还不行，一般来说 Email 地址是按照不区分大小写来处理的，也就说 `foo@bar.com` 和 `FOO@BAR.COM` 或 `FoO@BAr.coM` 是等效的，所以验证时也要考虑这种情况。[15]代码 6.17 是针对这种问题的测试代码。

代码 6.17：拒绝相同 Email 地址的测试，不区分大小写
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email address is already taken" do
    before do
      user_with_same_email = @user.dup
      user_with_same_email.email = @user.email.upcase
      user_with_same_email.save
    end

    it { should_not be_valid }
  end
end
```

上面的代码，在字符串上调用了 `upcase` 方法（参照 4.3.2 节）。这个测试和前面对相同 Email 地址的测试类似，只是把 Email 地址转换成全部大写字母的形式。如果觉得太抽象，那就在控制台中实操一下吧：

```
$ rails console --sandbox
>> user = User.create(name: "Example User", email: "user@example.com")
>> user.email.upcase
=> "USER@EXAMPLE.COM"
>> user_with_same_email = user.dup
>> user_with_same_email.email = user.email.upcase
```

15. 严格的说，Email 地址只有域名部分是不区分大小写的，`foo@bar.com` 和 `Foo@bar.com` 其实是不同的地址。但在实际使用中，千万别依赖这个规则，about.com 中相关的文章说道，“区分大小写的 Email 地址会带来很多麻烦，不易互换使用，也不利传播，所以要求输入正确地大小写是很愚蠢的。几乎没有 Email 服务提供商或 ISP 强制要求使用区分大小写的 Email 地址，也不会提示收件人的大小写错了（例如，要全部大写）。”感谢读者 Riley Moses 指正这个问题。

```
>> user_with_same_email.valid?
=> true
```

现在 `user_with_same_email.valid?` 的返回值是 `true`，因为唯一性验证还是区分大小写的。我们希望得到的结果是 `false`。幸好 `uniqueness` 可以指定 `:case_sensitive` 选项，正好可以解决这个问题，如代码 6.18 所示。

代码 6.18：验证 Email 地址的唯一性，不区分大小写
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  validates :email, presence: true,
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
end
```

注意，我们直接把 `true` 换成了 `case_sensitive: false`，Rails 会自动指定 `:uniqueness` 的值为 `true`。至此，应用程序虽还有不足，但基本可以保证 Email 地址的唯一性了，所有的测试都可以通过了。

唯一性验证的不足

现在还有一个小问题，在此衷心的提醒你：**唯一性验证无法真正保证唯一性。**

不会吧，哪里出了问题呢？下面我来解释一下。

1. Alice 用 alice@wonderland.com 注册；
2. Alice 不小心按了两次提交按钮，连续发送了两次请求；
3. 然后就会发生下面的事情：请求 1 在内存中新建了一个用户对象，通过验证；请求 2 也一样。请求 1 创建的用户存入了数据库，请求 2 创建的用户也存入了数据库。
4. 结果是，尽管有唯一性验证，数据库中还是有两条用户记录的 Email 地址是一样的。

相信我，上面这种难以置信的过程是可能会发生的，只要有一定的访问量，在任何 Rails 网站中都可能发生。幸好解决的办法很容易实现，只需在数据库层也加上唯一性限制。我们要做的是在数据库中为 `email` 列建立索引，然后为索引加上唯一性限制。

为 `email` 列建立索引就要改变数据库模型，在 Rails 中可以通过迁移实现（参见 6.1.1 节）。在 6.1.1 节 中我们看到，生成 User 模型时会自动创建一个迁移文件（参见代码 6.2），现在我们是要改变已经存在的模型结构，那么使用 `migration` 命令直接创建迁移文件就可以了：

```
$ rails generate migration add_index_to_users_email
```

和 User 模型的迁移不一样，实现 Email 唯一性的迁移操作没有事先定义的模板可用，所以我们要手动把代码 6.19 中的内容写入迁移文件。[16]

代码 6.19： 确保 Email 唯一性的迁移文件
db/migrate/[timestamp]_add_index_to_users_email.rb

```
class AddIndexToUsersEmail < ActiveRecord::Migration
  def change
    add_index :users, :email, unique: true
  end
end
```

上述代码调用了 Rails 中的 `add_index` 方法，为 `users` 表的 `email` 列建立索引。索引本身并不能保证唯一性，所以还要指定 `unique: true`。

然后执行数据库迁移操作：

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

（如果失败的话，就退出所有打开的控制台沙盒会话，这些会话可能会锁定数据库，拒绝进行迁移操作。）如果你想看一下操作执行后的效果，请打开 `db/schema.rb` 文件，会发现多了一行：

```
add_index "users", ["email"], name: "index_users_on_email", unique: true
```

为了保证 Email 地址的唯一性，还要做些修改：存入数据库之前把 Email 地址转换成全小写字母的形式，因为不是所有数据库适配器的索引都是区分大小写的。[17] 为此，我们要使用回调函数（callback），在 Active Record 对象生命周期的特定时刻调用（参阅 Rails API 中关于回调函数的文档）。本例中，我们要使用的回调函数是 `before_save`，在用户存入数据库之前强行把 Email 地址转换成全小写字母形式，如代码 6.20 所示。

代码 6.20： 把 Email 地址转换成全小写形式，确保唯一性
app/models/user.rb

```
class User < ActiveRecord::Base
  before_save { self.email = email.downcase }
  .
  .
  .
end
```

在代码 6.20 中，`before_save` 后跟有一个块，块中的代码调用了字符串的 `downcase` 方法，把用户的 Email 地址转换成小写字母形式。这些代码有些深度，此时你只需相信这些代码可以达到目的就行了。如果你有所怀疑，可以把代码 6.16 中的唯一性验证代码注释掉，创建几个 Email 地址一样的用户，看一下存储时得到的错误信息。（8.2.1 节 还会用到这个技巧，到时会使用推荐的“方法引用”约定。）编写针对代码 6.20 的测试会留作练习，参见 6.5 节。

至此，上面 Alice 遇到的问题就解决了，数据库会存储请求 1 创建的用户，不会存储请求 2 创建的用户，因为它违反了唯一性限制。（在 Rails 的日志中会显示一个错误，不过无大碍。其实我们可以捕获抛出的 `ActiveRecord::StatementInvalid` 异常，不过本教程不会涉及异常处理。）为 `email` 列建立索引同时也解决了 6.1.4 节中提到的 `find_by_email` 的效率问题（参阅旁注 6.2）。

16. 当然，我们可以直接编辑代码 6.2，不过，需要先回滚再迁移回来。这不是 Rails 的风格，正确的做法是每次修改数据库结构都要使用迁移。
17. 我在本地的 SQLite 以及 Heroku 的 PostgreSQL 中做过实验，证明这么做其实是必须的。

旁注 6.2：数据库索引

创建数据库列时，要考虑是否会用这个列进行查询。例如，代码 6.2 中的迁移，创建了 `email` 列，第 7 章中实现的用户登录功能，会通过提交的 Email 地址查询对应的用户记录。按照现有的数据模型，使用 Email 地址查找用户的唯一方式是遍历数据库中所有的用户记录，对比提交的 Email 地址和记录中的 `email` 列，看看是否一致。在数据库的术语中，这叫做“全表扫描（full-table scan）”，对一个有上千用户的网站而言，这可不是一件轻松的事。

为 `email` 列建立索引则可以解决这个问题。我们可以将数据库索引比拟成书籍的索引。如果要在一本书中找出某个字符串（例如 `"foobar"`）出现的所有位置，我们需要翻看书中的每一页。但是如果有索引的话，只需在索引中找到 `"foobar"` 条目，就能看到所有包含 `"foobar"` 的页码。数据库索引基本上也是这种原理。

6.3 加上安全密码

本节我们要加入用户所需的最后一个常规属性：安全密码，用来验证用户的身份。实现的方式是，用户记着自己的密码，而在数据库中存储着加密后的密码。稍后我们还会加入基于密码的用户身份验证机制，第 8 章会利用这个机制实现用户登录功能。

验证用户身份的方法是，获取用户提交的密码，进行加密，再和数据库中存储的加密密码对比，如果二者一致，用户提交的就是正确的密码，用户的身份也就验证了。我们要对比的是加密后的密码，而不是原始的密码文本，所以验证用户身份时不用在数据库中存储用户的密码，这样可以规避一个很大的安全隐患。

安全密码机制基本上是由 Rails 中一个单独的方法 `has_secure_password` 实现的（自 Rails 3.1 开始使用）。因为后续的内容都是基于这个方法的，所以很难循序渐进的讲解。从 6.3.2 节开始，我建议尽早的加入 `has_secure_password` 方法，然后每编写一个测试就注释掉这个方法，这样才能正确的使用 TDD。（视频往往能更好的表现一个循序渐进的开发过程，所以为了能更好的理解这一块，有兴趣的读者可以考虑观看本教程的配套视频。）

6.3.1 加密密码

我们先要对 User 数据结构做些改动，向 `users` 表中加入 `password_digest` 列（如图 6.5 所示）。digest 这个词是加密哈希函数中的一个术语。之所以要使用 `password_digest` 命名这个列，是因为 6.3.4 节实现的功能会用到这个列。适当的加密后，即便攻击者设法获取了数据库拷贝也无法黑掉网站。

users	
id	integer
name	string
email	string
password_digest	string
created_at	datetime
updated_at	datetime

图 **6.5**：加入 `password_digest` 属性后的 User 模型

我们要使用目前最先进的哈希函数 bcrypt 对密码进行不可逆的加密，得到密码的哈希值。如果要在程序中使用 bcrypt，需要把 `bcrypt-ruby` 这个 gem 加入 `Gemfile`，如代码 6.21 所示。

代码 6.21：把 bcrypt-ruby 加入 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'

gem 'rails', '4.0.2'
gem 'bootstrap-sass', '2.3.2.0'
gem 'bcrypt-ruby', '3.0.1'
.
.
.
```

然后运行 `bundle install` 安装：

```
$ bundle install
```

既然我们规划的 `users` 表中有一列是 `password_digest`，那么用户对象就应该可以响应 `password_digest` 方法，按照这个思路我们就可以编写出如代码 6.22 所示的测试。

代码 6.22：确保 `users` 表中有 `password_digest` 列
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  .
  .
  .
end
```

为了让测试通过，首先要生成一个合适的迁移文件，添加 `password_digest` 列：

```
$ rails generate migration add_password_digest_to_users password_digest:string
```

上述命令的第一个参数是迁移的名字，第二个参数指明要添加列的名字和数据类型。（可以和代码 6.1 中生成 users 表的代码对比一下。）迁移的名字可以随便起，但一般会以 _to_users 结尾，Rails 会自动生成一个向 users 表中增加列的迁移。我们还提供了第二个参数，Rails 就得到了足够的信息，会为我们生成整个迁移文件，如代码 6.23 所示。

代码 6.23： 向 users 表中添加 password_digest 列的迁移
db/migrate/[ts]_add_password_digest_to_users.rb

```
class AddPasswordDigestToUsers < ActiveRecord::Migration
  def change
    add_column :users, :password_digest, :string
  end
end
```

上述代码中调用 add_column 方法向 users 表添加 password_digest 列。

在开发数据库中执行迁移操作，再准备好测试数据库，代码 6.22 中的测试就可以通过了：

```
$ bundle exec rake db:migrate
$ bundle exec rake test:prepare
$ bundle exec rspec spec/
```

6.3.2 密码和密码确认

如图 6.1 中的构思图所示，我们希望用户进行密码确认，这在网络中是很普遍的做法，可以减小误输入带来的风险。虽然在控制器层可以实现这个想法，不过在模型层实现更好，Active Record 可以确保密码确认万无一失。为此，我们要把 password 和 password_confirmation 两列加入 User 模型，在记录存入数据库之前比较这两列的值是否一样。和之前见过的属性不一样，password 是虚拟的属性，只是临时存在于内存中，而不会存入数据库中。

我们先编写检查是否可以响应 password 和 password_confirmation 方法的测试，如代码 6.24 所示。

代码 6.24： 测试 password 和 password_confirmation 属性
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User",
                     email: "user@example.com",
                     password: "foobar",
                     password_confirmation: "foobar")
  end

  subject { @user }
```

```
  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }
  it { should be_valid }
  .
  .
  .
end
```

注意，我们在 `User.new` 的初始化参数 Hash 中加入了 `password` 和 `password_confirmation`：

```
before do
  @user = User.new(name: "Example User",
                   email: "user@example.com",
                   password: "foobar",
                   password_confirmation: "foobar")
end
```

我们当然不想让用户输入空白的密码，那么再编写一个验证密码存在性的测试：

```
describe "when password is not present" do
  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: " ", password_confirmation: " ")
  end
  it { should_not be_valid }
end
```

因为稍后会测试密码不一致的情形，所以存在性验证就把密码和密码确认两项都设为了空格字符串。

我们还要确保密码和密码确认的值是相同的，这种情况 `it { should be_valid }` 已经覆盖了。那么再编写针对二者不同时的测试就可以了：

```
describe "when password doesn't match confirmation" do
  before { @user.password_confirmation = "mismatch" }
  it { should_not be_valid }
end
```

把上述代码放在一起，就是代码 6.25 中的（失败）测试了。

代码 6.25： 对 `password` 和 `password_confirmation` 的测试
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }

  it { should be_valid }
  .
  .
  .
  describe "when password is not present" do
    before do
      @user = User.new(name: "Example User", email: "user@example.com",
                       password: " ", password_confirmation: " ")
    end
    it { should_not be_valid }
  end

  describe "when password doesn't match confirmation" do
    before { @user.password_confirmation = "mismatch" }
    it { should_not be_valid }
  end
end
```

我们只需添加一行代码就可以让代码 6.25 中的测试通过，如代码 6.26 所示。

代码 6.26：让密码测试通过
app/models/user.rb

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

has_secure_password 这一行代码是很强大的，可以让所有针对密码的测试都通过，而且作用还不知如此。但是，这行代码的作用太过强大，甚至后面的测试未遇红就变绿了，所以在继续之前，我建议把这行注释掉（如代码 6.27 所示）。

代码 6.27：为了践行 TDD，注释掉 has_secure_password
app/models/user.rb

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

6.3.3 用户身份验证

实现密码机制的最后一步，是找到一种方法，使用 Email 地址和密码取回用户对象。这个方法很自然的可以分成两步，首先，通过 Email 地址找到用户记录；然后再用密码进行身份验证。除了最后一个测试，本节所有测试都是通过 has_secure_password 实现的，所以为了让测试通过必须解除代码 6.27 中的注释行。

第一步很简单，如 6.1.4 节中介绍过的，我们可以调用 find_by 方法，通过 Email 地址查找用户记录：

```
user = User.find_by(email: email)
```

接下来的第二步，可以使用 authenticate 方法验证用户的密码。在第 8 章中会使用类似下面的代码获取当前登录的用户：

```
current_user = user.authenticate(password)
```

如果提交的密码和用户的密码一致，上述代码就会返回一个用户对象，否则返回 false。

和之前一样，我们可以使用 RSpec 测试必须要定义有 authenticate 方法。这个测试比前面见过的要难一些，我们分段来看。如果你刚接触 RSpec，可能要多读几遍本节的内容。首先，用户对象应该能够响应 authenticate 方法：

```
it { should respond_to(:authenticate) }
```

然后测试密码是否正确：

```
describe "return value of authenticate method" do
  before { @user.save }
  let(:found_user) { User.find_by(email: @user.email) }

  describe "with valid password" do
    it { should eq found_user.authenticate(@user.password) }
  end
```

```
  describe "with invalid password" do
    let(:user_for_invalid_password) { found_user.authenticate("invalid") }

    it { should_not eq user_for_invalid_password }
    specify { expect(user_for_invalid_password).to be_false }
  end
end
```

before 块中的代码先把用户存入数据库，然后在 let 块中调用 find_by 方法取出用户：

```
let(:found_user) { User.find_by(email: @user.email) }
```

在前几章的练习题中用过很多次 let 了，但这是第一次在正文中出现。旁注 6.3较为深入的介绍了 let 方法。接下来的两个 describe 块测试了 @user 和 found_user 是否为同一个用户。测试代码中使用 eq 测试对象是否相同（会调用 == 测试是否相同，参见 4.3.1 节）。注意，下面的测试中

```
describe "with invalid password" do
  let(:user_for_invalid_password) { found_user.authenticate("invalid") }

  it { should_not eq user_for_invalid_password }
  specify { expect(user_for_invalid_password).to be_false }
end
```

再次用到了 let 方法，还用到了 specify 方法。specify 是 it 方法的别名，如果你觉得某个地方用 it 读起来怪怪的，就可以换用 specify。本例中，“it should not equal wrong user”读起来很顺，不过“user: user with invalid password should be false”有点累赘，换用“specify: user with invalid password should be false”感觉就好些。

旁注 6.3：let 方法

我们可以使用 RSpec 提供的 let 方法便捷的在测试中定义局部变量。let 方法的句法看起来有点怪，不过和变量赋值语句的作用是一样的。let 方法的参数是一个 Symbol，后面可以跟着一个块，块中代码的返回值会赋给名为 Symbol 代表的局部变量。也就是说：

```
let(:found_user) { User.find_by(email: @user.email) }
```

定义了一个名为 found_user 的变量，其值等于 find_by 的返回值。在这个测试用例的任何一个 before 或 it 块中都可以使用这个变量。使用 let 方法定义变量的一个好处是，它可以记住（memoize）变量的值。（memoize 是个行业术语，不是“memorize”的误拼写。）对上面的代码而言，因为 let 的备忘功能，found_user 的值会被记住，因此不管调用多少次 User 模型测试，find_by 方法只会运行一次。

最后，安全起见，我们还要编写一个密码长度测试，大于 6 个字符才能通过：

```
describe "with a password that's too short" do
  before { @user.password = @user.password_confirmation = "a" * 5 }
  it { should be_invalid }
end
```

把上面的代码放在一起后，如代码 6.28 所示。

代码 6.28： 对 authenticate 方法的测试
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }
  .
  .
  .
  it { should respond_to(:authenticate) }
  .
  .
  .
  describe "with a password that's too short" do
    before { @user.password = @user.password_confirmation = "a" * 5 }
    it { should be_invalid }
  end

  describe "return value of authenticate method" do
    before { @user.save }
    let(:found_user) { User.find_by(email: @user.email) }

    describe "with valid password" do
      it { should eq found_user.authenticate(@user.password) }
    end

    describe "with invalid password" do
      let(:user_for_invalid_password) { found_user.authenticate("invalid") }

      it { should_not eq user_for_invalid_password }
      specify { expect(user_for_invalid_password).to be_false }
    end
```

```
  end
end
```

如旁注 6.3中介绍的，`let` 方法会记住变量的值，所以嵌套中的第一个 `describe` 块通过 `let` 方法把 `find_by` 的结果赋值给 `found_user` 之后，在后续的 `describe` 块中就无需再次查询数据库了。

6.3.4 用户的安全密码

在较旧版本的 Rails 中，添加一个安全的密码是很麻烦也很费时的事，本书的第一版（针对 Rails 3.0）中就从零起开发了一个用户身份验证系统。熟知用户身份验证系统的开发者完全没必要在此浪费时间，所以在 Rails 的最新版中已经集成了用户身份验证功能。因此，我们只需要几行代码就可以为用户添加一个安全的密码，还可以让前几小节的测试通过。

首先，我们要为密码添加一个长度验证，使用和代码 6.12 中用过的 `:maximum` 类似的 `:minimum`：

```
validates :password, length: { minimum: 6 }
```

（密码和密码确认的存在性验证都在 `has_secure_password` 方法中实现了。）

然后，我们要添加 `password` 和 `password_confirmation` 属性，二者都要填写一些内容（非空格），而且要相等；还要定义 `authenticate` 方法，对比加密后的密码和 `password_digest` 是否一致，验证用户的身份。这些步骤本来很麻烦，不过在最新版的 Rails 中已经集成好了，只需调用一个方法就可以了，这个方法是 `has_secure_password`：

```
has_secure_password
```

只要数据库中有 `password_digest` 列，在模型文件中加入 `has_secure_password` 方法后就能验证用户身份了。

（如果你觉得 `has_secure_password` 方法太过神奇，不妨阅读一下 `secure_password.rb` 文件中的代码，里面有很多注释，代码本身也不难理解。这段代码中有这么几行：

```
validates_confirmation_of :password,
                          if: lambda { |m| m.password.present? }
```

这行代码会自动创建 `password_confirmation` 属性（Rails API 中有说明）。这段代码还有针对 `password_digest` 属性的数据验证。）

把以上的代码放在一起就得到了如代码 6.29 所示的 User 模型，同时也实现了安全的密码机制。

代码 6.29：最终实现的安全的密码机制
app/models/user.rb

```
class User < ActiveRecord::Base
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
```

```
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence:   true,
                    format:     { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
  has_secure_password
  validates :password, length: { minimum: 6 }
end
```

现在你可以看一下测试是否可以通过了：

```
$ bundle exec rspec spec/
```

注意：如果看到如下的弃用提示：

```
[deprecated] I18n.enforce_available_locales will default to true in the future
```

请按照如下方式修改 `config/application.rb`：

```
require File.expand_path('../boot', __FILE__)
.
.
.
module SampleApp
  class Application < Rails::Application
    .
    .
    .
    I18n.enforce_available_locales = true
    .
    .
    .
  end
end
```

6.3.5 创建用户

至此，基本的 User 模型已经架构好了，接下来我们要在数据库中存入一个用户记录，为 7.1 节开发的用户资料显示页面做准备，同时也可以看一下前几节所做工作的实际效果。测试通过并不意味着工作做完了，如果开发数据库中有一条用户记录的话，或许能给我们带来一点成就感。

因为现在还不能在网页中注册（第 7 章会实现），我们要在控制台中手动创建新用户。和 6.1.3 节不一样，本节使用的不是沙盒模式下的控制台，因为我们真的要在数据库中保存一条记录：

```
$ rails console
>> User.create(name: "Michael Hartl", email: "mhartl@example.com",
?>              password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com", created_at:
"2013-03-11 20:45:19", updated_at: "2013-03-11 20:45:19", password_digest:
"$2a$10$kn4cQDJTzV76ZgDxOWk6Je9A0Ttn5sKNaGTEmT0jU7.n...">
```

在 SQLite 数据库浏览器中打开开发数据库（`db/development.sqlite3`），会发现上述命令执行后存入的记录（如图 6.6），各列对应了图 6.5 中数据模型相应的属性。

图 6.6： SQLite 数据库（`db/development.sqlite3`）中的一条记录

再回到控制台中，读取 `password_digest` 属性的值，看一下代码 6.29 中 `has_secure_password` 方法的作用：

```
>> user = User.find_by(email: "mhartl@example.com")
>> user.password_digest
=> "$2a$10$kn4cQDJTzV76ZgDxOWk6Je9A0Ttn5sKNaGTEmT0jU7.ncBJ/60gHq"
```

这是初始化用户时提供的原始密码（`"foobar"`）对应的加密形式。下面再来验证一下 `authenticate` 方法是否可以正常使用，先提供不正确的密码，再提供正确的密码，结果如下：

```
>> user.authenticate("invalid")
=> false
>> user.authenticate("foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com", created_at:
"2013-03-11 20:45:19", updated_at: "2013-03-11 20:45:19", password_digest:
"$2a$10$kn4cQDJTzV76ZgDxOWk6Je9A0Ttn5sKNaGTEmT0jU7.n...">
```

如代码 6.28 中的测试需求，如果密码不正确，`authenticate` 方法会返回 `false`；如果密码正确，则会返回对应的用户对象。

6.4 小结

本章从零开始，建立了一个运作良好的 User 模型，包含 `name`、`email` 以及几个密码相关的属性，通过数据验证对属性的取值做了限定。而且，已经可以使用密码对用户进行身份验证了。在较旧版本的 Rails 中，实现这些功能需要编写大量的代码，不过在最新版中，使用强大的 `validates` 和 `has_secure_password` 方法，只需几行代码就可以构建完整的 User 模型了。

在接下来的第 7 章中，我们会创建一个注册表单用来新建用户，还会创建一个页面用来显示用户的信息。第 8 章则会使用 6.3 节实现的身份验证机制让用户可以登录网站。

如果你使用 Git，而且一直都没做提交的话，现在最好提交一下本章所做的改动：

```
$ git add .
$ git commit -m "Make a basic User model (including secure passwords)"
```

然后合并到主分支中：

```
$ git checkout master
$ git merge modeling-users
```

6.5 练习

1. 为代码 6.20 中把 Email 地址转换成小写字母形式的功能编写一个测试，测试代码可以参考代码 6.30。这段测试使用 `reload` 方法从数据库中重新加载数据；使用 `eq` 方法测试相等。把代码 6.20 中的 `before_save` 一行注释掉，看一下代码 6.30 的测试是否可以通过。

2. 把代码 6.20 中 `before_save` 一行改成代码 6.31 的形式，运行测试看一下这样改写是否可行。

3. 我们在 6.2.4 节中说过，代码 6.14 中的 Email 正则表达式能匹配出现连续点号的不合法 Email 地址，例如 “foo@bar..com”。把这个地址添加到代码 6.13 中的不合法地址列表中让测试失败，然后使用代码 6.32 中较复杂的正则表达式让测试通过。

4. 通读 Rails API 中关于 `ActiveRecord::Base` 的内容，了解一下这个类的作用。

5. 细读 Rails API 中关于 `validates` 方法的内容，学习这个方法其他的用法和参数。

6. 花点时间熟悉一下 Rubular。

代码 6.30：对代码 6.20 中 Email 变小写的测试
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  describe "email address with mixed case" do
    let(:mixed_case_email) { "Foo@ExAMPle.CoM" }

    it "should be saved as all lower-case" do
      @user.email = mixed_case_email
      @user.save
      expect(@user.reload.email).to eq mixed_case_email.downcase
    end
  end
  .
  .
  .
end
```

代码 6.31：`before_save` 回调函数的另一种写法
`app/models/user.rb`

```
class User < ActiveRecord::Base
  has_secure_password
  before_save { email.downcase! }
  .
  .
  .
end
```

代码 6.32：不允许 Email 地址中出现连续点号的正则表达式
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-]+(\.[a-z]+)*\.[a-z]+\z/i
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
  .
```

```
  .
  .
end
```

第 7 章 用户注册

User 模型可以正常使用了，接下来要实现的功能大多数网站都离不开：用户注册。在 7.2 节我们会创建一个表单，提交用户注册时填写的信息，然后在 7.4 节中使用提交的数据创建新用户，把相应的属性值存入数据库。注册功能实现后，还要创建一个用户资料页面，显示用户的个人信息，这是实现 Users 资源 REST 架构（参见 2.2.2 节）的第一步。和之前一样，开发的过程中要编写测试，结合 RSpec 和 Capybara 写出简洁有效的集成测试。

创建资料页面之前，数据库中先要有用户记录。这有点类似“先有鸡还是先有蛋”的问题：网站还没实现注册功能，数据库中怎么会有用户记录呢？其实这个问题在 6.3.5 节中已经解决了，我们在控制台中向数据库中存储了一个用户记录。如果你跳过了那一节，现在赶快往回翻，完成相应的操作。

如果你一直坚持使用版本控制系统，现在要新建一个从分支了：

```
$ git checkout master
$ git checkout -b sign-up
```

7.1 显示用户信息

本节要实现的用户资料页面是完整页面的一小部分，只显示用户的名字和头像，构思图如图 7.1 所示。[1] 最终完成的用户资料页面会显示用户的头像、基本信息和一些微博，构思图如图 7.2 所示。[2]（在图 7.2 中，我们第一次用到了“lorem ipsum”占位文字，这些文字背后的故事很有意思，有空的话你可以了解一下。）整个资料页面会和整个示例程序一起在 第 11 章完成。

7.1.1 调试信息和 Rails 环境

本节要实现的用户资料页面是第一个真正意义上的动态页面。虽然视图的代码不会动态改变，不过每个用户资料页面显示的内容却是动态的从数据库中读取的。添加动态页面之前，最好做些准备工作，现在我们能做的就是在网站布局中加入一些调试信息（参见代码 7.1）。代码 7.1 使用 Rails 内置的 `debug` 方法和 `params` 变量（7.1.2 节会详细介绍），在每一页中都显示一些对开发有所帮助的信息。

代码 7.1：把调试信息加入网站的布局中
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  .
  .
```

1. Mockingbird 不支持插入额外的图片，图 7.1 中的图片是我使用 Adobe Fireworks 加上的。
2. 图中的河马原图在此 http://www.flickr.com/photos/43803060@N00/24308857/

```
  .
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
  </body>
</html>
```

图 7.1：本节要实现的用户资料页面构思图

我们要在第 5 章中创建的自定义样式表文件中加入一些样式规则（参见代码 7.2），美化一下这些调试信息。

代码 7.2：添加美化调试信息的样式，使用了一个 Sass mixin
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";
```

```
/* mixins, variables, etc. */

$grayMediumLight: #eaeaea;

@mixin box_sizing {
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
.
.
.

/* miscellaneous */

.debug_dump {
  clear: both;
  float: left;
  width: 100%;
  margin-top: 45px;
  @include box_sizing;
}
```

上面的代码用到了 Sass 的 mixin 功能，创建的这个 mixin 名为 `box-sizing`。mixin 可以打包一系列的样式规则，供多次使用。预处理器处理时，会把

```
.debug_dump {
  .
  .
  .
  @include box_sizing;
}
```

转换成

```
.debug_dump {
  .
  .
  .
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

在 7.2.2 节中还会再次用到这个 mixin。美化后的调试信息如图 7.3 所示。

图 7.2：最终实现的用户资料页面构思图

图 7.3 中显示的调试信息给出了当前页面的一些信息：

```
---
controller: static_pages
action: home
```

这是 `params` 变量的 YAML[3]形式，和 Hash 类似，显示了当前页面的控制器名和动作名。在 7.1.2 节中会介绍其他调试信息的意思。

我们不想让部署后的示例程序显示这个调试信息，所以代码 7.1 中用如下的代码做了限制，只在“开发环境”中显示：

```
if Rails.env.development?
```

3. Rails 的调试信息是 YAML（YAML Ain’t Markup Language）格式的，这种格式对机器和人类都很友好。

图 7.3：显示有调试信息的示例程序首页（/）

“开发环境”是 Rails 定义的三个环境之一（详细介绍参见旁注 7.1）[4]，只有在“开发环境”中 `Rails.env.development?` 才会返回 `true`，所以下面的 ERb 代码

```
<%= debug(params) if Rails.env.development? %>
```

不会在“生产环境”和“测试环境”中执行。（在“测试环境”中显示调试信息虽然没有坏处，但也没什么好处，所以最好只在“开发环境”中显示。）

旁注 7.1：Rails 的三个环境

Rails 定义了三个环境，分别是“生产环境”、“开发环境”和“测试环境”。Rails 控制台默认使用的是“开发环境”：

```
$ rails console
Loading development environment
>> Rails.env
=> "development"
>> Rails.env.development?
```

4. 你还可以自己定义其他的环境，详细方法可以参照 Railscasts 的《Adding an Environment》。

```
=> true
>> Rails.env.test?
=> false
```

如前所示，Rails 对象有一个 `env` 属性，属性上还可以调用各环境对应的布尔值方法，例如，`Rails.env.test?`，在“测试环境”中的返回值是 `true`，而在其他两个环境中的返回值则是 `false`。

如果需要在其他环境中使用控制台（例如，在“测试环境”中进行调试），只需把环境名称传递给 `console` 命令即可：

```
$ rails console test
Loading test environment
>> Rails.env
=> "test"
>> Rails.env.test?
=> true
```

Rails 本地服务器和控制台一样，默认使用“开发环境”，不过也可以在其他环境中运行：

```
$ rails server --environment production
```

如果要在“生产环境”中运行应用程序，先要提供生产环境数据库。在“生产环境”中执行 `rake db:migrate` 命令可以生成“生产环境”所需的数据库：

```
$ bundle exec rake db:migrate RAILS_ENV=production
```

（我发现在控制台、服务器和迁移命令中指定其他环境的方法不一样，这可能会产生混淆，所以我特意演示了三个命令的用法。）

顺便说一下，把应用程序部署到 Heroku 后，可以使用如下的命令进入远端的控制台：

```
$ heroku run console
Ruby console for yourapp.herokuapp.com
>> Rails.env
=> "production"
>> Rails.env.production?
=> true
```

Heroku 是用来部署网站的平台，自然会在“生产环境”中运行应用程序。

7.1.2 Users 资源

在第 6 章末尾，我们在数据库中存储了一个用户记录，在 6.3.5 节查看过，用户的 id 是 1，现在我们就来创建一个页面，显示这个用户的信息。我们会遵从 Rails 使用的 REST 架构，把数据视为资源（resource），可以创建、显示、更新和删除，这四个操作分别对应了 HTTP 标准中的 `POST`、`GET`、`PATCH` 和 `DELETE` 请求方法（参见旁注 3.3）。

按照 REST 约定，资源一般是由资源名加唯一标识符表示的。对 User 而言，我们把它看做一个资源，若要查看 id 为 1 的用户，就要向 /users/1 地址发送一个 `GET` 请求。REST 架构解析时，会自动把这个 `GET` 请求分发到 `show` 动作上，因此这里没必要指明用哪个动作。

在 2.2.1 节中曾经见过，id 为 1 的用户对应的地址是 /users/1，现在访问这个地址的话会显示错误提示信息（如图 7.4）。

图 **7.4**：/users/1 地址显示的错误

我们只需在路由文件 `config/routes.rb` 中添加如下的一行代码就可以正常访问 REST 架构对应的 URL 地址了：

```
resources :users
```

修改后的路由文件如代码 7.3 所示。

代码 7.3： 在路由文件中添加 Users 资源设置
config/routes.rb

```
SampleApp::Application.routes.draw do
  resources :users
  root to: 'static_pages#home'
  match '/signup',  to: 'users#new',            via: 'get'
  .
  .
  .
end
```

你可能发现了，我们把下面这行在代码 5.33 中出现的代码删掉了：

```
get "users/new"
```

这是因为 `resources :users` 不仅使 /users/1 地址可以访问了，而且还为示例程序的 Users 资源提供了符合 REST 架构的一系列动作[5]，以及用来获取相应 URL 地址的具名路由（named route，参见 5.3.3 节）。最终得到的 URI、动作和具名路由的对应关系如表格 7.1 所示（可以和表格 2.2 对比一下）。接下来的三章会介绍 `show` 之外的所有动作，并不断完善，把 Users 打造成完全符合 REST 架构的资源。

表格 7.1：Users 资源对应的路由

HTTP 请求	URL	动作	具名路由	作用
GET	/users	index	users_path	显示所有用户的页面
GET	/users/1	show	user_path(user)	显示某个用户的页面
GET	/users/new	new	new_user_path	创建（注册）新用户的页面
POST	/users	create	users_path	创建新用户
GET	/users/1/edit	edit	edit_user_path(user)	编辑 id 为 1 的用户页面
PATCH	/users/1	update	user_path(user)	更新用户信息
DELETE	/users/1	destroy	user_path(user)	删除用户

添加代码 7.3 之后，路由就生效了，但是页面还不存在（如图 7.5）。下面我们就来为页面添加一些内容，7.1.4 节还会添加更多的内容。

用户资料页面的视图存放在应用程序特定的目录中，即 `app/views/users/show.html.erb`。这个视图和自动生成的 `new.html.erb`（参见代码 5.29）不同，现在不存在，要手动创建。新建 `show` 视图后请写入代码 7.4 中的代码。

5. 这句话的意思是，路由已经设置好了，但相应的页面还无法正常访问。例如，/users/1/edit 已经映射到 Users 控制器的 `edit` 动作上了，但是 `edit` 动作还不存在，所以访问这个地址就会显示一个错误页面。

图 7.5：/users/1 地址生效了，但是页面不存在

代码 7.4： 用户资料页面的临时视图
app/views/users/show.html.erb

```
<%= @user.name %>, <%= @user.email %>
```

在上面的代码中，我们假设 `@user` 变量是存在的，使用 ERb 代码显示用户的名字和 Email 地址。这和最终实现的视图有点不一样，在最终的视图中不会公开显示用户的 Email 地址。

我们要在 Users 控制器的 `show` 动作中定义 `@user` 变量，用户资料页面才能正常渲染。你可能猜到了，我们要在 User 模型上调用 `find` 方法，从数据库中取出用户记录，如代码 7.5 所示。

代码 7.5： 含有 show 动作的 Users 控制器
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
  end
end
```

在上面的代码中，我们使用 `params` 获取用户的 id。当我们向 Users 控制器发送请求时，`params[:id]` 会返回用户的 id，即 1，所以这就和 6.1.4 节中直接调用 `User.find(1)` 的效果一样。（严格来说，`params[:id]` 返回的是字符串 `"1"`，`find` 方法会自动将其转换成整数形式。）

定义了视图和动作之后，/users/1 地址就可以正常显示了（如图 7.6）。留意一下调试信息，其内容证实了 `params[:id]` 的值和前面分析的一样：

```
---
action: show
controller: users
id: '1'
```

所以，代码 7.5 中的 `User.find(params[:id])` 才会取回 id 为 1 的用户记录。

图 **7.6**：设置 Users 资源后的用户资料页面 /users/1

7.1.3 使用预构件测试用户资料页面

至此，用户资料页面已经可以正常访问了，接下来我们要实现图 7.1 所示的构思了。与创建静态页面和 User 模型一样，开发的过程会践行 TDD 思想。

在 5.4.2 节中，我们使用集成测试对 Users 资源相关的页面进行了测试，以“注册”页面为例，测试先访问 `signup_path`，然后检测页面中 `h1` 和 `title` 标签的内容是否正确。代码 7.6 是对代码 5.32 的补充。（注意，我们删除了 5.3.4 节用到的 `full_title` 帮助方法，因为标题已经测试过了。）

代码 7.6： 补充用户相关页面的测试
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "signup page" do
    before { visit signup_path }

    it { should have_content('Sign up') }
    it { should have_title(full_title('Sign up')) }
  end
end
```

为了测试用户资料页面，先要有一个 User 模型对象，代码 7.5 中的 `show` 动作才能执行查询操作：

```
describe "profile page" do
  # Replace with code to make a user variable
  before { visit user_path(user) }

  it { should have_content(user.name) }
  it { should have_title(user.name) }
end
```

我们要把上面代码中的注释换成相关的代码才行。在注释后面，调用 `user_path` 具名路由（参见表格 7.1）访问用户资料页面的地址，然后检测页面内容和标题是否都包含用户的名字。

一般情况下，创建 User 模型需要调用 Active Record 提供的 `User.create` 方法，不过经验告诉我们，使用预构件（factory）创建用户对象更方便，存入数据库也更容易。

我们要使用 Factory Girl 来生成预构件，这个 gem 是由 thoughtbot 公司的达人开发的。和 RSpec 类似，Factory Girl 也定义了一套领域专属语言（Domain-specific Language, DSL），用来生成 Active Record 对象。Factory Girl 的句法很简单，使用块和方法定义对象的属性值。本章还没有显出预构件的优势，不过后续的内容会多次使用预构件的高级更能，到时你就可以看到它的强大之处了。例如，在 9.3.3 节中，需要生成一批 Email 地址各不相同的用户对象，用预构件就可以很轻松的完成这种操作。

和其他的 gem 一样，我们要在 Bundler 的 `Gemfile` 中加入如代码 7.7 所示的代码来安装 Factory Girl。（因为只有测试时才会用到 Factory Girl，所以把它归入测试组中。）

代码 7.7： 把 Factory Girl 加入 `Gemfile`

```
source 'https://rubygems.org'
  .
  .
  .
group :test do
  .
  .
  .
  gem 'factory_girl_rails', '4.2.1'
end
.
.
.
```

然后和往常一样，运行以下命令安装：

```
$ bundle install
```

Factory Girl 生成的预构件都保存在 `spec/factories.rb` 中，RSpec 会自动加载这个文件。用户相关的预构件如代码 7.8 所示。

代码 7.8：模拟 User 模型对象的预构件
`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    name     "Michael Hartl"
    email    "michael@example.com"
    password "foobar"
    password_confirmation "foobar"
  end
end
```

`factory` 方法的 `:user` 参数说明，块中的代码定义了一个 User 模型对象。

加入代码 7.8 之后，就可以在测试中使用 `let` 方法（参见旁注 6.3）和 Factory Girl 提供的 `FactoryGirl` 方法来生成 User 对象：

```
let(:user) { FactoryGirl.create(:user) }
```

修改后的测试如代码 7.9 所示。

代码 7.9：用户资料页面的测试
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
```

```
  subject { page }

  describe "profile page" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit user_path(user) }

    it { should have_content(user.name) }
    it { should have_title(user.name) }
  end

  describe "signup page" do
    before { visit signup_path }

    it { should have_content('Sign up') }
    it { should have_title(full_title('Sign up')) }
  end
end
```

现在你应该看一下测试是否是失败的（红色）：

```
$ bundle exec rspec spec/
```

加入代码 7.10 之后，测试就可以通过了（绿色）。

代码 7.10：在用户资料页面视图中加入标题和标头
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<h1><%= @user.name %></h1>
```

再次运行 RSpec，确认代码 7.9 中的测试是否可以通过：

```
$ bundle exec rspec spec/
```

使用 Factory Girl 后，明显可以察觉测试变慢了，这不是 Factory Girl 导致的，而是有意为之，并不是 bug。变慢的原因在于 6.3.1 节中用来加密密码的 BCrypt，其加密算法设计如此，因为慢速加密的密码很难破解。慢速加密的过程会延长测试的运行时间，不过我们可以做个简单的设置改变这种情况。`bcrypt-ruby` 使用耗时因子（cost factor）设定加密过程的耗时，耗时因子的默认值倾向于安全性而不是速度，在生产环境这种设置很好，但测试时的关注点却有所不同：测试追求的是速度，而不用在意测试数据库中用户的密码强度。我们可以在“测试环境”配置文件 `config/environments/test.rb` 中加入一行代码来解决速度慢的问题：把耗时因子的默认值修改为最小值，提升加密的速度，如代码 7.11 所示。即使测试量很少，修改设置之后速度的提升也是很明显的，所以我建议每个读者都在 `test.rb` 文件中加入代码 7.11 的内容。

代码 7.11：为测试环境重新设置 BCrypt 耗时因子
`config/environments/test.rb`

```
SampleApp::Application.configure do
  .
  .
  .
  # Speed up tests by lowering bcrypt's cost function.
  ActiveModel::SecurePassword.min_cost = true
end
```

7.1.4 添加 Gravatar 头像和侧边栏

上一小节创建了一个略显简陋的用户资料页面，这一小节要再添加一些内容：用户头像和侧边栏。构建视图时，我们关注的是显示的内容，而不是页面底层的结构，所以我们暂时不测试视图，等遇到容易出错的页面结构时，例如 9.3.3 节中的分页导航，再使用 TDD 理念。

首先，我们要在用户资料页面中添加一个“全球通用识别”的头像，这个头像也称作 Gravatar[6]，由 Tom Preston-Werner（GitHub 的联合创始人）开发，后被 Automattic（开发 WordPress 的公司）收购。Gravatar 是一个免费服务，用户只需上传图片并将其关联到 Email 地址上即可。使用 Gravatar 可以简单的在网站中加入用户头像，开发者不必再分心去处理图片上传、剪裁和存储，只要使用用户的 Email 地址构成头像的 URL 地址，关联的头像就可以显示出来了。[7]

我们计划定义一个名为 `gravatar_for` 的方法，返回指定用户的 Gravatar 头像，如代码 7.12 所示。

代码 7.12： 显示用户名字和 Gravatar 头像的用户资料页面视图
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<h1>
  <%= gravatar_for @user %>
  <%= @user.name %>
</h1>
```

现在看一下测试是不是失败的：

```
$ bundle exec rspec spec/
```

因为还没定义 `gravatar_for` 方法，所以用户资料页面会显示错误提示。（测试视图最大的作用大概就是可以捕获这种错误，所以一定要掌握视图测试的量。）

默认情况下，所有帮助方法文件中定义的方法都可以直接用在任意的视图中，不过为了便于管理，我们会把 `gravatar_for` 放在 Users 控制器对应的帮助文件中。Gravatar 的首页中有介绍说，头像的 URL 地址要使用 MD5 加密的 Email 地址。在 Ruby 中，MD5 加密算法由 `Digest` 库的 `hexdigest` 方法实现：

6. 在印度教中，avatar 是神的化身，可以是一个人，也可以是一种动物。由此引申到其他领域，特别是在虚拟世界中，avatar 就代表一个人。你可能看过《阿凡达》这部电影了，所以你可能也已经知道 avatar 的意思了。
7. 如果你的应用程序需要处理图片及其他文件的上传，我推荐你使用 Paperclip 或 CarrierWave。

```
>> email = "MHARTL@example.COM".
>> Digest::MD5::hexdigest(email.downcase)
=> "1fda4469bcbec3badf5418269ffc5968"
```

Email 地址不区分大小写，而 MD5 加密算法却区分，所以，我们要先调用 `downcase` 方法把 Email 地址转换成小写形式，然后再传递给 `hexdigest` 方法。我们定义的 `gravatar_for` 方法如代码 7.13 所示。

代码 7.13：定义 `gravatar_for` 帮助方法
`app/helpers/users_helper.rb`

```
module UsersHelper

  # Returns the Gravatar (http://gravatar.com/) for the given user.
  def gravatar_for(user)
    gravatar_id = Digest::MD5::hexdigest(user.email.downcase)
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

`gravatar_for` 方法的返回值是 Gravatar 头像的 `img` 元素，`img` 标签的 class 设为 `gravatar`，`alt` 属性值是用户的名字（对视觉障碍人士使用的屏幕阅读器很友好）。现在你可以验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

用户资料页面的效果如图 7.7 所示，页面中显示的头像是 Gravatar 的默认图片，因为 `user@example.com` 不是真的 Email 地址（example.com 这个域名是专门用来举例的）。

我们调用 `update_attributes` 方法（参见 6.1.5 节）更新一下数据库中的用户记录，然后就可以显示用户真正的头像了：

```
$ rails console
>> user = User.first
>> user.update_attributes(name: "Example User",
?>                        email: "example@railstutorial.org",
?>                        password: "foobar",
?>                        password_confirmation: "foobar")
=> true
```

上面的代码，把用户的 Email 地址设为 `example@railstutorial.org`，我已经把这个 Email 地址的头像设为了本书网站的 LOGO。修改后的结果如图 7.8 所示。

我们还要添加一个侧边栏，才能完整的实现图 7.1 中的构思。我们要使用 `aside` 标签定义侧边栏，`aside` 中的内容一般是对主体内容的补充，不过也可以自成一体。我们要把 `aside` 标签的 class 设为 `row span4`，这也是 Bootstrap 会用到的。在用户资料页面中添加侧边栏用到的代码如代码 7.14 所示。

图 7.7：显示默认 Gravatar 头像的用户资料页面 /users/1

代码 7.14： 为用户资料页面添加侧边栏
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="span4">
    <section>
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
  </aside>
</div>
```

添加了 HTML 结构和 CSS class 后，我们再用 SCSS 为资料页面定义样式，如代码 7.15 所示。（注意：因为 asset pipeline 使用了 Sass 预处理器，所以样式中才可以使用嵌套。）最终的效果如图 7.9 所示。

代码 7.15： 用户资料页面的样式，包括侧边栏的样式
`app/assets/stylesheets/custom.css.scss`

```
.
.
```

```
.

/* sidebar */

aside {
  section {
    padding: 10px 0;
    border-top: 1px solid $grayLighter;
    &:first-child {
      border: 0;
      padding-top: 0;
    }
    span {
      display: block;
      margin-bottom: 3px;
      line-height: 1;
    }
    h1 {
      font-size: 1.4em;
      text-align: left;
      letter-spacing: -1px;
      margin-bottom: 3px;
      margin-top: 0px;
    }
  }
}

.gravatar {
  float: left;
  margin-right: 10px;
}
```

7.2 注册表单

用户资料页面已经可以访问了，但内容还不完整。下面我们要为网站创建一个注册表单。如图 5.9 和图 7.10 所示，“注册”页面还没有什么内容，无法注册新用户。本节会实现如图 7.11 所示的注册表单，添加注册功能。

因为我们要实现通过网页创建用户的功能，现在就把 6.3.5 节在控制台中创建的用户删除吧。最简单的方法是使用 `db:reset` 命令：

```
$ bundle exec rake db:reset
```

还原数据库后，在有些系统中还要重新准备测试数据库：

```
$ bundle exec rake test:prepare
```

图 7.8：显示用户真实头像的用户资料页面 /users/1

在某些系统中还要重启 Web 服务器，还原数据库的操作才能生效。[8]

7.2.1 测试用户注册功能

在 Web 框架没有完全支持测试之前，测试是件很痛苦的事，也很容易出错。例如，手动测试“注册”页面时，我们要在浏览器中访问这个页面，然后分别提交不合法的和合法的数据，检查在这两种情况下应用程序的表现是否正常。而且，每次修改程序后，都要重复上述的操作。使用 RSpec 和 Capybara 之后，以前需要手动进行的测试，现在可以编写测试用例自动执行了。

前面的章节已经介绍过 Capybara 访问网页时使用的很直观的句法，其中用的最多的就是访问某个页面的 `visit` 方法。Capybara 的功能可不仅限于此，它还可以填写如图 7.11 所示的表单字段，然后点击提交按钮，句法如下：

```
visit signup_path
fill_in "Name", with: "Example User"
.
.
```

8. 有点搞不懂是吧，其实我也不知道原因。

```
.
click_button "Create my account"
```

现在我们要分别提交不合法的和合法的注册数据，验证注册功能是否可以正常使用。我们要用到的测试相对高级一些，所以会慢慢分析。如果你想查看最终的测试代码（以及测试文件的位置），可以直接跳到代码 7.16。先来测试没有正确填写信息的注册表单，我们访问“注册”页面，什么也不填，直接点击注册按钮（调用 `click_button` 方法），这个操作模拟的就是提交不合法数据的情况：

```
visit signup_path
click_button "Create my account"
```

图 7.9：添加侧边栏并定义了样式之后的用户资料页面 /users/1

上面的代码，等同于手动访问注册页面，然后提交空白的不合法注册信息。相对的，我们要调用 `fill_in` 方法填写合法信息，以此来模拟提交合法数据的情况：

```
visit signup_path
fill_in "Name",         with: "Example User"
fill_in "Email",        with: "user@example.com"
fill_in "Password",     with: "foobar"
fill_in "Confirmation", with: "foobar"
click_button "Create my account"
```

我们测试的最终目的，是要检测点击“Create my account”按钮之后，程序的表现是否正常，即当提交合法的数据时，创建新用户；当提交不合法的数据时，不创建新用户。检测是是否创建了新用户，我们要看用户的数量是否发生了变化，在测试中，我们使用每个 Active Record 对象都可以响应的 `count` 方法来获取对象的数量，以用户为例，即：

```
$ rails console
>> User.count
=> 0
```

图 7.10：“注册”页面现在的样子 /signup

现在 `User.count` 的返回值是 0，因为本节开头我们还原了数据库。提交不合法数据时，我们希望用户的数量是不变的；提交合法数据时，我们希望用户的数量增加 1 个。在 RSpec 中，上面的设想要结合 `expect` 和 `to`，或者和 `not_to` 方法来表述。我们先从不合法的数据开始，因为这种情况比较简单。我们先访问“注册”页面，然后点击提交按钮，希望用户的数量不变：

```
visit signup_path
expect { click_button "Create my account" }.not_to change(User, :count)
```

注意，通过花括号我们可以看出，`expect` 把 `click_button` 包含在一个块中（参见 4.3.2 节），这是为 `change` 方法做的特殊处理。`change` 方法可接受两个参数，第一个参数是对象名，第二个是 Symbol。`change` 方法会在

expect 块中的代码执行前后，分别计算在第一个参数上调用第二参数代表的方法返回的结果。也就是说，如下的代码

```
expect { click_button "Create my account" }.not_to change(User, :count)
```

会在执行

```
click_button "Create my account"
```

前后，两次计算

```
User.count
```

的结果。

图 7.11：“注册”页面的构思图

本例，我们用 not_to 方法表示不愿看到用户数量发生变化。把点击按钮的代码放入块中，相当于把

```
initial = User.count
click_button "Create my account"
final = User.count
expect(initial).to eq final
```

替换成简单的一行代码

```
expect { click_button "Create my account" }.not_to change(User, :count)
```

这样读起来更顺口，代码也更简洁。（注意，eq 是 RSpec 提供的方法，用来测试是否相等。）

提交合法数据的情况和上述不合法数据的情况类似，不过用户数量不是不变，而是增加了 1 个：

```
visit_signup path
fill_in "Name", with: "Example User"
fill_in "Email", with: "user@example.com"
fill_in "Password", with: "foobar"
fill_in "Confirmation", with: "foobar"
expect do
  click_button "Create my account"
end.to change(User, :count).by(1)
```

这里使用了 to 方法，我们希望点击提交按钮后，这些合法的数据可以用来创建一个新用户。我们把上面两种情况放入一个 describe 块中，再把共用的代码放入 before 块中，最终得到的注册功能测试代码如代码 7.16 所示。我们还做了一项重构，用 let 方法定义了 submit 变量，表示注册按钮的文本。

代码 7.16：测试用户注册功能的代码
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do

  subject { page }
  .
  .
  .
  describe "signup" do

    before { visit signup_path }

    let(:submit) { "Create my account" }

    describe "with invalid information" do
      it "should not create a user" do
        expect { click_button submit }.not_to change(User, :count)
```

```
      end
    end

    describe "with valid information" do
      before do
        fill_in "Name",         with: "Example User"
        fill_in "Email",        with: "user@example.com"
        fill_in "Password",     with: "foobar"
        fill_in "Confirmation", with: "foobar"
      end

      it "should create a user" do
        expect { click_button submit }.to change(User, :count).by(1)
      end
    end
  end
end
```

后续几节还会添加更多的测试，不过现在这个测试已经可以检测相当多的功能表现是否正常了。若要使这个测试通过，先得创建包含正确元素的注册页面，提交注册信息后页面要转向正确的地址，而且如果数据是合法的，还要创建一个新用户，并存入数据库中。

当然了，现在测试还是失败的：

```
$ bundle exec rspec spec/
```

7.2.2 使用 form_for

我们已经为用户注册功能编写了适当的测试代码，接下来要创建用户注册表单了。在 Rails 中，创建表单可以使用 `form_for` 帮助方法，指定其参数为 Active Record 对象，然后使用对象的属性构建表单的字段。注册表单的视图如代码 7.17 所示。（熟悉 Rails 2.x 的读者要注意一下，这里 `form_for` 使用的是 `<%=...%>` 形式，而 Rails 2.x 使用的是 `<%...%>` 形式。）

代码 7.17： 用户注册表单
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>

      <%= f.label :name %>
      <%= f.text_field :name %>
```

```
      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation %>

      <%= f.submit "Create my account", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

我们来分析一下这些代码。在上面的代码中，我们使用了关键词 `do`，说明 `form_for` 后面可以跟着块，而且可以传入一个块参数 `f`，代表这个表单：

```
<%= form_for(@user) do |f| %>
  .
  .
  .
<% end %>
```

我们一般无需了解 Rails 帮助方法的内部实现，但是对于 `form_for` 来说，我们要知道 `f` 对象的作用是什么：调用表单字段（例如，文本字段、单选按钮、密码字段）对应的方法时，生成的表单字段元素可以用来设定 `@user` 对象的属性。也就是说：

```
<%= f.label :name %>
<%= f.text_field :name %>
```

生成的 HTML 是一个有标号（label）的文本字段，可以用来设定 User 模型的 `name` 属性。（7.2.3 节会看到生成的 HTML）看过生成的 HTML 才能理解为什么字段可以设定属性。在此之前，还有个问题要解决，因为没有定义 `@user` 变量，页面无法显示。和其他未定义的实例变量一样，`@user` 的值现在是 `nil`。所以如果运行测试的话，会看到代码 7.6 中针对注册页面内容的测试是失败的：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "signup page"
```

（上述命令中的 `-e` 参数指定只运行描述文本包含“signup page”字符串的测试用例。如果改成“signup”，则会运行代码 7.16 中的所有测试。）

要使这个测试通过，同时也让页面可以正常显示，我们要在 `new.html.erb` 视图对应的 `new` 动作中定义 `@user` 变量。`form_for` 方法的参数需要一个 User 对象，而且我们要创建新用户，所以我们可以使用 `User.new` 方法，如代码 7.18 所示。

代码 7.18：在 new 动作中定义 @user 变量
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def new
    @user = User.new
  end
end
```

定义 @user 变量后，注册页面的测试就可以通过了：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "signup page"
```

再添加代码 7.19 中的样式，表单的效果如图 7.12 所示。注意，我们再次用到了代码 7.2 中的 box-sizing 这个 mixin。

代码 7.19：注册表单的样式
app/assets/stylesheets/custom.css.scss

```
.
.
.

/* forms */

input, textarea, select, .uneditable-input {
  border: 1px solid #bbb;
  width: 100%;
  margin-bottom: 15px;
  @include box_sizing;
}

input {
  height: auto !important;
}
```

7.2.3 表单的 HTML

如图 7.12 所示，“注册”页面现在可以正常显示了，说明代码 7.17 中的 form_for 方法生成了合法的 HTML。生成的表单 HTML（可以使用 Firebug 或浏览器的“查看源文件”菜单查看）如代码7.20 所示。某些细节现在无需关心，我们只解说最重要的结构。

代码 7.20：图 7.12 中表单的 HTML

```
<form accept-charset="UTF-8" action="/users" class="new_user"
      id="new_user" method="post">

  <label for="user_name">Name</label>
  <input id="user_name" name="user[name]" type="text" />

  <label for="user_email">Email</label>
  <input id="user_email" name="user[email]" type="text" />

  <label for="user_password">Password</label>
  <input id="user_password" name="user[password]"
         type="password" />

  <label for="user_password_confirmation">Confirmation</label>
  <input id="user_password_confirmation"
         name="user[password_confirmation]" type="password" />

  <input class="btn btn-large btn-primary" name="commit" type="submit"
         value="Create my account" />
</form>
```

图 **7.12**：注册用户的表单 /signup

（上面的代码省略了“鉴别权标（authenticity token）”相关的 HTML。Rails 使用鉴别权标来防止跨站请求伪造（cross-site request forgery, CSRF）攻击。如果你对鉴别权标感兴趣，可以阅读一下 Stack Overflow 网站中的《Understand Rails Authenticity Token!》一文，这篇文章介绍了鉴别权标的工作原理及重要意义。）

下面看一下表单的字段。比较代码 7.17 和代码 7.20 之后，我们可以看到，如下的 ERb 代码

```
<%= f.label :name %>
<%= f.text_field :name %>
```

生成的 HTML 是

```
<label for="user_name">Name</label>
<input id="user_name" name="user[name]" type="text" />
```

图 7.13：填写了文本字段和密码字段的表单

下面的 ERb 代码

```
<%= f.label :password %>
<%= f.password_field :password %>
```

生成的 HTML 是

```
<label for="user_password">Password</label><br />
<input id="user_password" name="user[password]" type="password" />
```

如图 7.13 所示，文本字段（`type="text"`）会直接显示填写的内容，而密码字段（`type="password"`）基于安全考虑会遮盖输入的内容。

在 7.4 节中我们会介绍，之所以可以创建用户，全赖于 `input` 元素的 `name` 属性：

```
<input id="user_name" name="user[name]" - - - />
.
.
.
<input id="user_password" name="user[password]" - - - />
```

图 7.14：注册失败后的页面构思图

Rails 会以 `name` 属性的值为键，用户输入的内容为值，构成一个名为 `params` 的 Hash，用来创建用户。另外一个重要的标签是 `form`。我们使用 `@user` 对象来创建 `form`元素，因为每个 Ruby 对象都知道它所属的类（参见 4.4.1

节），所以 Rails 知道 @user 所属的类是 User；而且，@user 代表的是新创建的用户，Rails 知道要使用 POST 请求方法，这正是创建新对象所需的 HTTP 请求（参见旁注 3.3）：

```
<form action="/users" class="new_user" id="new_user" method="post">
```

先不看 class 和 id 属性，我们现在关注的是 action="/users" 和 method="post"。设定这两个属性后，Rails 就会向 /users 地址发送一个 POST 请求。接下来的两节会介绍这个请求产生的效果。

7.3 注册失败

虽然上一节大概的介绍了图 7.12 中表单的 HTML 结构（参见代码 7.20），不过注册失败时才能更好的理解这个表单的作用。本节，我们会在注册表单中填写一些不合法的数据，提交表单后，页面不会转向其他页面，而是返回“注册”页面，显示一些错误提示信息，页面的构思图如图7.14 所示。

7.3.1 可正常使用的表单

回顾一下 7.1.2 节中的内容，在 routes.rb 中设置 resources :users 之后（参见代码 7.3），Rails 应用程序就可以响应表格 7.1中符合 REST 架构的 URI 地址了。一般来说，发送到 /users 地址的 POST 请求是由 create 动作处理的。在 create 动作中，我们可以调用 User.new 方法，使用提交的数据创建一个新用户对象，尝试存入数据库，失败后再重新渲染“注册”页面，允许访客重新填写注册信息。我们先来看一下生成的 form 元素：

```
<form action="/users" class="new_user" id="new_user" method="post">
```

在 7.2.3 节中介绍过，这个表单会向 /users 地址发送 POST 请求。

添加代码 7.21 之后，代码 7.16 中对不合法数据的测试就可以通过了。代码 7.21 中再次调用了 render 方法，第一使用时是为了插入局部视图（参见5.1.3 节），不过如你所见，在控制器的动作中也可以使用这个方法。同时，我们也借此代码介绍了 if-else 分支结构的用法：根据 @user.save 的返回值，分别处理用户存储成功和失败这两种情况（存储成功时返回值为 true，失败时返回值为 false）。

代码 7.21：处理存储失败的 create 动作（还不能处理存储成功的情况）
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(params[:user])    # Not the final implementation!
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
```

```
  end
end
```

留意代码 7.21 中的注释，这并不是最终的实现方式，但现在完全够用。最终版会在 7.3.2 节中实现。

我们要实际的操作一下，提交一些不合法的注册数据，这样才能更好的理解代码 7.21 的作用，结果如图 7.15 所示，底部完整的调试信息如图 7.16 所示。

图 7.15：注册失败

下面我们来分析一下调试信息中请求参数 Hash 的 `user` 部分（如图 7.16），以便对 Rails 处理表单的过程有个更清晰地认识：

```
"user" => { "name" => "Foo Bar",
            "email" => "foo@invalid",
            "password" => "[FILTERED]",
            "password_confirmation" => "[FILTERED]"
          }
```

在 7.1.2 节中就说过，`params` Hash 中包含了每次请求的信息，例如向 /users/1 发送的请求，`params[:id]` 的值就是用户的 id，即 1。提交表单发送 `POST` 请求时，`params` 则是一个嵌套的 Hash。嵌套 Hash 在 4.3.3 节中使用控制

台介绍 params 时用过。上面的调试信息说明，提交表单后，Rails 会构建一个名为 user 的 Hash，其键是 input 标签的 name 属性值（参见代码 7.17），键对应的值是用户填写的字段文本。例如，

```
<input id="user_email" name="user[email]" type="text" />
```

图 7.16： 注册失败时的调试信息

该字段 name 属性的值是 user[email]，它代表的就是 user Hash 的 email 元素。

虽然调试信息中的键是字符串形式，不过却以 Symbol 形式传递给 Users 控制器。params[:user] 这个嵌套的 Hash，实际上就是 User.new 方法创建用户所需的参数值。（我们在 4.4.5 节中介绍过 User.new 的用法，代码 7.21 再次用到了这个方法。）也就是说，如下的代码

```
@user = User.new(params[:user])
```

基本上等同于

```
@user = User.new(name: "Foo Bar", email: "foo@invalid",
                 password: "foo", password_confirmation: "bar")
```

在 Rails 之前的版本中，使用

```
@user = User.new(params[:user])
```

就行了，但默认情况下这种用法并不安全，需要采用谨慎的处理过程避免恶意用户篡改程序的数据库。在 Rails 4.0 中，这行代码会抛出异常（如图 7.15 和图 7.16 所示），这样就安全了。保险起见，我们可以看一下测试是否是失败的：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
-e "signup with invalid information"
```

7.3.2 健壮参数

我们在 4.4.5 节中提到过 mass assignment，使用一个 Hash 初始化 Ruby 变量，例如下面这行代码

```
@user = User.new(params[:user])    # Not the final implementation!
```

上面这行代码中的注释来自代码 7.21，说明这不是最终的实现方式。因为初始化整个 `params` Hash 是十分危险的，一旦初始化就会把用户提交的所有数据传递给 `User.new` 方法。假设除了前述的属性，User 模型还包含 `admin` 属性，用来标识网站的管理员。（我们会在 9.4.1 节加入这个属性。）要把这个属性设为 `true`，就要在 `params[:user]` 中包含 `admin='1'`，这个过程可以轻易的使用 `curl` 这种命令行 HTTP 客户端实现。结果是，把整个 `params` 传递给 `User.new` 网站的任一用户都可以在请求中包含 `admin='1'` 来获取管理员权限。

在 Rails 的之前版本中可以在模型层中使用 `attr_accessible` 方法解决这个问题，但在 Rails 4.0 中推荐的方法是在控制器层使用一种叫做“健壮参数”（strong parameter）的技术。这个技术可以指定需要哪些请求参数，允许传入哪些请求参数。而且，如果按照上面的方式传入整个 `params` Hash，程序会抛出异常，所以现在的 Rails 程序默认情况下已经堵住 mass assignment 漏洞了。

本例，我们需要 `params` Hash 包含 `:user` 元素，而且只允许传入 `name`、`email`、`password` 和 `password_confirmation` 属性。所需代码如下：

```
params.require(:user).permit(:name, :email, :password, :password_confirmation)
```

这行代码会返回一个 `params` Hash，只包含允许使用的属性，而且如果没有指定 `:user` 元素会抛出异常。

为了使用方便，可以定义一个名为 `user_params` 的辅助方法，返回所需的 Hash。这个方法只会在 Users 控制器内部使用，不需要暴露给用户，所以我们可以使用 Ruby 中的 `private` 方法把这个方法设为私有作用域，如代码 7.22 所示。（我们会在 8.2.1 节中更详细的介绍 `private`。）

代码 7.22： 在 `create` 动作中使用健壮参数
`app/controller/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
```

```
    @user = User.new(user_params)
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

添加了上述代码后，针对不合法数据的测试应该可以通过了：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
-e "signup with invalid information"
```

7.3.3 注册时的错误提示信息

处理注册失败的最后一步，我们要加入一个错误提示信息，说明注册失败的原因。默认情况下 Rails 会显示这样的信息，基于 User 模型的数据验证机制实现。举个例子，我们试着用不合法的 Email 地址和长度较短的密码创建用户看看会发生什么：

```
$ rails console
>> user = User.new(name: "Foo Bar", email: "foo@invalid",
?>                 password: "dude", password_confirmation: "dude")
>> user.save
=> false
>> user.errors.full_messages
=> ["Email is invalid", "Password is too short (minimum is 6 characters)"]
```

如上所示，`errors.full_message` 对象是一个错误信息组成的数组。和上面的控制台对话类似，代码 7.21 中的代码也无法保存用户，会生成一个附属在 `@user` 对象上的错误信息数组。如果要在注册页面显示这些错误，我们需要渲染一个错误信息局部视图，如代码 7.23 所示。（最好先为这些错误信息编写测试，我们会把这个任务留作练习，详情参见 7.6 节。）注意，这个局部视图只是暂时使用，我们会在 10.3.2 节中编写最终版本。

代码 7.23： 在注册表单前显示的错误提示信息
`app/views/users/new.html.erb`

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
```

```
<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>
      .
      .
      .
    <% end %>
  </div>
</div>
```

注意，在上面的代码中渲染的局部视图名为 shared/error_messages，这里用到了 Rails 的一个约定：如果局部视图要在多个控制器中使用，则把它存放在专门的 shared 目录下。（这个约定 9.1.1 节还会再介绍）我们除了要新建 _error_messages.html.erb 文件之外，还要新建 app/views/shared 文件夹。错误提示信息局部视图的内容如代码 7.24 所示。

代码 7.24： 显示表单错误提示信息的局部视图
app/views/shared/_error_messages.html.erb

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <div class="alert alert-error">
      The form contains <%= pluralize(@user.errors.count, "error") %>.
    </div>
    <ul>
    <% @user.errors.full_messages.each do |msg| %>
      <li>* <%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

这个局部视图的代码使用了几个之前没用过的 Rails/Ruby 结构，还有两个新方法。第一个新方法是 count，它的返回值是错误信息的数量：

```
>> user.errors.count
=> 2
```

第二个新方法是 any?，它和 empty? 的作用相反：

```
>> user.errors.empty?
=> false
>> user.errors.any?
=> true
```

第一次使用 `empty?` 方法是在 4.2.3 节，用在字符串上；从上面的代码可以看出，`empty?` 也可用在 Rails 错误信息对象上，如果对象为空就返回 `true`，否则返回 `false`。`any?` 方法就是取反 `empty?` 的返回值，如果对象中有内容就返回 `true`，没内容则返回 `false`。（顺便说一下，`count`、`empty?` 和 `any?` 都可以用在 Ruby 数组上，在 10.2 节中会好好地介绍这三个方法。）

图 7.17：注册失败时显示的错误提示信息

还有一个比较新的方法是 `pluralize`，在控制台中默认不可用，不过我们可以引入 `ActionView::Helpers::TextHelper` 模块加载这个方法：[9]

```
>> include ActionView::Helpers::TextHelper
>> pluralize(1, "error")
=> "1 error"
>> pluralize(5, "error")
=> "5 errors"
```

如上所示，`pluralize` 方法的第一个参数是整数，返回值是这个数字和第二个参数文本组合在一起正确的单复数形式。`pluralize` 方法是由功能强大的转置器（inflector）实现的，转置器知道怎么处理大多数单词的单复数变换，甚至是一些不规则的变换方式：

9. 我之所以知道要引入 `ActionView::Helpers::TextHelper` 模块，是因为我在 Rails API 中查了 `pluralize` 的文档。

```
>> pluralize(2, "woman")
=> "2 women"
>> pluralize(3, "erratum")
=> "3 errata"
```

所以，使用 `pluralize` 方法后，如下的代码

```
<%= pluralize(@user.errors.count, "error") %>
```

返回值就是 `"0 errors"`、`"1 error"` 或 `"2 errors"`等，单复数形式取决于错误的数量。这样就可以避免类似 `"1 errors"` 这种低级的错误了（这是网络中常见的错误之一）。注意，代码 7.24 中还为一个 `div` 标签指定了 `error_explanation` id，可用来样式化错误提示信息。（在 5.1.2 节中介绍过，CSS 中以 # 开头的规则是用来给 id 添加样式的。）出错时，Rails 还会自动把有错误的字段包含在一个 class 为 `field_with_errors` 的 `div` 元素中。我们可以利用这个 id 和 class 为错误提示信息添加样式，所需的 SCSS 如代码 7.24 所示。代码 7.25 中使用 Sass 的 `@extend` 函数引入了 `control-group` 和 `error` 两个样式规则集合。添加样式后，如果提交失败，错误信息和出错的字段就会显示为红色，如图 7.17 所示。错误信息的文本是基于模型数据验证自动生成的，所以如果你修改了验证规则（例如，Email 地址的格式，密码的最小长度），错误信息就会自动变化，符合修改后的规则。

代码 7.25：错误提示信息的样式
`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* forms */
.
.
.
#error_explanation {
  color: #f00;
  ul {
    list-style: none;
    margin: 0 0 18px 0;
  }
}

.field_with_errors {
  @extend .control-group;
  @extend .error;
 }
```

我们可以通过下面的方法模拟代码 7.16 中针对不合法数据的测试过程，来看一下本节编程的效果：在浏览器中，访问“注册”页面，什么也不填，直接点击“Create my account”按钮。结果如图 7.18 所示。既然“注册”页面可以正常使用了，相应的测试也应该可以通过了。

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
> -e "signup with invalid information"
```

图 7.18： 访问“注册”页面后直接点击“Create my account”按钮的效果

7.4 注册成功

上一节已经处理了提交不合法数据的情况，本节我们要完成注册表单的功能，如果提交的数据合法，就把用户存入数据库。我们先尝试保存用户，如果保存成功，用户的数据就会存入数据库中，然后网页会转向刚注册用户的资料页面，页面中会显示一个欢迎信息，构思图如图 7.19 所示。如果保存用户失败了，就交由上一节实现的功能处理。

7.4.1 完整的注册表单

要完成注册表单的功能，我们要把代码 7.21 中的注释部分换成相应的处理代码。现在，对提交合法数据的测试还是失败的：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
> -e "signup with valid information"
```

测试之所以会失败，是因为 Rails 处理动作的默认方式是渲染视图，可是 `create` 动作还没有（也不应该有）对应的视图。相反的，我们要转向其他的页面，最合理的转向页面是刚创建用户的资料页面。检测是否转向正确页面的测试会留作练习（参见 7.6 节），`create` 动作的代码如代码 7.26 所示。

图 7.19：注册成功后显示的页面构思图

代码 7.26：`create` 动作的代码，处理了保存和转向操作
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to @user
    else
      render 'new'
    end
```

```
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

注意，转向地址我们直接写了 `@user`，而没用 `user_path`，Rails 会自动转向到用户的资料页面。

加入代码 7.26 后，注册表单就可以正常使用了，你可以运行测试验证一下：

```
$ bundle exec rspec spec/
```

7.4.2 Flash 消息

验证合法数据是否能够正确提交之前，我们还要加入一个 Web 应用程序大都会实现的功能：在转向后的页面中显示一个消息（这里我们要显示的是一个欢迎新用户的消息），如果访问了其他页面或者刷新了页面，这个消息便会消失。在 Rails 中这种功能是通过 `flash` 变量实现的，`flash` 就像闪存一样，只是暂时存储的数据。`flash` 变量的值其实是一个 Hash，你可能还记得，4.3.3 节中我们在控制台中遍历了一个名为 `flash` 的 Hash：

```
$ rails console
>> flash = { success: "It worked!", error: "It failed." }
=> {:success=>"It worked!", error: "It failed."}
>> flash.each do |key, value|
?> puts "#{key}"
?> puts "#{value}"
>> end
success
It worked!
error
It failed.
```

我们可以把显示 Flash 消息的代码加入应用程序的布局，这样整个网站在需要的时候就会显示消息了，如代码 7.27 所示。（代码 7.27 混合了 HTML 和 ERb 代码，有点乱，7.6 节中的练习会进行重构。）

代码 7.27： 把 `flash` 消息相关的代码加入网站的布局中
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  .
  .
```

```
  .
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <% flash.each do |key, value| %>
        <div class="alert alert-<%= key %>"><%= value %></div>
      <% end %>
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
    .
    .
    .
  </body>
</html>
```

代码 7.27 会为每一个 Flash 消息插入一个 `div` 标签，并且把 CSS class 指定为消息的类型。例如，如果 `flash[:success] = "Welcome to the Sample App!"`，那么下面的代码

```
<% flash.each do |key, value| %>
  <div class="alert alert-<%= key %>"><%= value %></div>
<% end %>
```

生成的 HTML 如下

```
<div class="alert alert-success">Welcome to the Sample App!</div>
```

（注意，键 `:success` 是 Symbol，在插入模板之前，ERb 会自动将其转换成字符串 `"success"`。）我们遍历了所有可能出现的 Flash 消息，这样当消息存在时才能显示。在 8.1.5 节 中会使用 `flash[:error]` 显示登录失败消息。[10]

检测页面中是否显示了正确的 Flash 消息的测试留作练习（参见 7.6 节）。在 `create` 动作中给 `flash[:success]` 赋值一个欢迎信息后（如代码 7.28 所示），这个测试就可以通过了。

代码 7.28：注册成功后显示 Flash 信息
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
```

10. 其实我们真正想用的是 `flash.now`，`flash` 和 `flash.now` 之间还是有细微差别的，后续会介绍。

```
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

7.4.3 首次注册

现在我们可以注册一下看看到目前为止所实现的功能，用户的名字使用“Rails Tutorial”，Email 地址使用“example@railstutorial.org”。注册成功后，页面中显示了一个友好的欢迎信息，信息的样式是由 5.1.2 节中加入的 Bootstrap 框架提供的 `.success` class 实现的，如图 7.20 所示。（如果你无法注册，提示 Email 地址已经使用，请确保你运行了 7.2 节中的 `rake db:reset` 命令。）然后刷新页面，Flash 消息就会消失了，如图 7.21 所示。

我们还可以检查一下数据库，确保真的创建了新用户：

```
$ rails console
>> User.find_by(email: "example@railstutorial.org")
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.org",
created_at: "2013-03-12 05:51:34", updated_at: "2013-03-12 05:51:34",
password_digest: "$2a$10$A58/j7wwh3aAffGkMAO9Q.jjh3jshd.6akhDKtchAz/R...">
```

7.4.4 部署到生产环境，并开启 SSL

创建了 User 模型、实现了注册功能之后，现在可以把示例程序部署到生产环境中了。（如果你没有按照第 3 章章头介绍中的内容设置生产环境的话，现在最好回过头去设置一下。）部署的时候我们还会开启对安全套接层（Secure Sockets Layer, SSl）[11]协议的支持，确保注册过程的安全。其实我们会全站都开启 SSL，这样用户的登录（参见第 8 章）也会很安全了，而且还可以避免会话劫持（session hijacking）的发生（参见 8.2.1 节）。

在部署之前，你应该把本章实现的功能合并到 `master` 分支：

```
$ git add .
$ git commit -m "Finish user signup"
```

11. 严格来说，SSL 现在的称呼是 TLS（Transport Layer Security，安全传输层协议），不过人们已经习惯了 SSL 这个名字。

```
$ git checkout master
$ git merge sign-up
```

图 7.20：注册成功后显示的页面，页面中有一个 Flash 消息

然后我们要设置一下，强制在生产环境中使用 SSL，这里我们要编辑的文件是 `config/environments/production.rb`，添加的设置如代码 7.29 所示。

代码 7.29：设置程序在生产环境中开启 SSL
`config/environments/production.rb`

```
SampleApp::Application.configure do
  .
  .
  .
  # Force all access to the app over SSL, use Strict-Transport-Security,
  # and use secure cookies.
  config.force_ssl = true
  .
  .
  .
end
```

图 7.21：刷新页面后 Flash 消息就不见了

我们要把这次修改提交到 Git 仓库中，然后再推送到 Heroku，所做的设置才能生效：

```
$ git commit -a -m "Add SSL in production"
$ git push heroku
```

然后，我们还要在生产环境中运行数据库迁移，告知 Heroku 我们建立了 User 模型：

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

（执行这个命令后，你可能会得到一些功能废弃的警告提示，现在可以直接忽视这些警告。）

最后一步，我们要在远程服务器上架设 SSL。在生产环境中架设对 SSL 的支持很麻烦，也很容易出错，而且还要为域名购买 SSL 签名证书。幸好，使用 Heroku 提供的二级域名可以直接使用 Heroku 的 SSL 签名证书，这算是 Heroku 平台的一个特性。如果你要为自己的域名（例如 `example.com`）开启 SSL，就无法享受这个便利的服务了，还要自行克服一些设置上的麻烦，具体的步骤在 Heroku 关于 SSL 的文档中有说明。

以上所有工作得到的最终结果是，在生产服务器上可以正常使用注册表单了，如图 7.22 所示：

```
$ heroku open
```

注意，在图 7.22 中，通常显示为 `http://` 的地方现在显示的是 `https://`，就是这个额外的“s”，证明我们正在使用 SSL。

现在你可以打开注册页面注册一个新用户了。如果遇到问题，运行 `heroku logs`，尝试使用 Heroku 的日志文件排错。

图 7.22：线上正常运作的注册页面

7.5 小结

实现注册功能对示例程序来说算是取得了很大的进展。虽然现在还没实现真正有用的功能，不过我们却为后续功能的开发奠定了坚实的基础。第 8 章，我们会实现用户登录、退出功能，完成整个身份验证机制。第 8 章，我们会实现更新用户个人信息的功能。我们还会实现管理员删除用户的功能，这样才算是完整的实现了表格 7.1 中所列 Users 资源相关的 REST 动作。最后，我们还会在各动作中实现权限验证功能，提升网站的安全。

7.6 练习

1. 以代码 7.30 为蓝本，验证一下在 7.1.4 节中定义的 `gravatar_for` 帮助方法是否可以接受名为 `size` 的可选参数，允许在视图中使用类似 `gravatar_for user, size: 40` 这样的形式。

2. 编写测试检测代码 7.23 中实现的错误提示信息。可以参照代码 7.31。

3. 采取先编写测试的方式，或者等到程序出错后再补救，验证一下代码 7.32 中的测试可以确保 `create` 动作中保存用户之后顺利的转到了用户资料页面。代码 7.32 中用到了 Capybara 提供的 `have_selector` 方法，这个方法可以测试 HTML 标签（通过 CSS class 指定）是否出现在页面中。

4. 我们前面说过，代码 7.27 中 Flash 消息相关的代码有点乱，我们要换用代码 7.33 中的代码，运行测试看一下使用 `content_tag` 帮助方法之后效果是否一样。

代码 7.30：重新定义 `gravatar_for` 方法，允许接受可选的 `size` 参数
`app/helpers/users_helper.rb`

```
module UsersHelper

  # Returns the Gravatar (http://gravatar.com/) for the given user.
  def gravatar_for(user, options = { size: 50 })
    gravatar_id = Digest::MD5::hexdigest(user.email.downcase)
    size = options[:size]
    gravatar_url = "https://secure.gravatar.com/
avatar/#{gravatar_id}.png?s=#{size}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

代码 7.31：错误提示信息测试的参考
`spec/requests/user_pages_spec.rb`

```
  .
  .
  .
  describe "signup" do

    before { visit signup_path }
    .
    .
    .
    describe "with invalid information" do
      .
      .
      .
      describe "after submission" do
        before { click_button submit }

        it { should have_title('Sign up') }
        it { should have_content('error') }
      end
      .
      .
      .
```

代码 7.32：对 create 动作中保存用户操作的测试
spec/requests/user_pages_spec.rb

```
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "after saving the user" do
        before { click_button submit }
        let(:user) { User.find_by(email: 'user@example.com') }

        it { should have_title(user.name) }
        it { should have_selector('div.alert.alert-success', text: 'Welcome') }
      end
      .
      .
      .
```

代码 7.33：使用 content_for 编写的 Flash 消息视图代码
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
  <html>
  .
  .
  .
  <% flash.each do |key, value| %>
    <%= content_tag(:div, value, class: "alert alert-#{key}") %>
  <% end %>
  .
  .
  .
</html>
```

第 8 章 登录和退出

第 7 章已经实现了注册新用户的功能，本章我们要为已注册的用户提供登录和退出功能。实现登录功能之后，就可以根据登录状态和当前用户的身份定制网站的内容了。例如，本章我们会更新网站的头部，显示“登录”或“退出”链接，以及到个人资料页面的链接；在第 10 章中，会根据当前登录用户的 id 创建关联到这个用户的微博；在第 11 章，我们会实现当前登录用户关注其他用户的功能，实现之后，在首页就可以显示被关注用户发表的微博了。

实现登录功能之后，还可以实现一种安全机制，即根据用户的身份限制可以访问的页面，例如，在第 9 章中会介绍如何实现只有登入的用户才能访问编辑用户资料的页面。登录系统还可以赋予管理员级别的用户特别的权限，例如删除用户（也会在第 9 章中实现）等。

实现验证系统的核心功能之后，我们会简要的介绍一下 Cucumber 这个流行的行为驱动开发（Behavior-driven Development, BDD）系统，使用 Cucumber 重新实现之前的一些 RSpec 集成测试，看一下这两种方式有何不同。

和之前的章节一样，我们会在一个新的从分支中工作，本章结束后再将其合并到主分支中：

```
$ git checkout -b sign-in-out
```

8.1 session 和登录失败

session 是两台电脑（例如运行有网页浏览器的客户端电脑和运行 Rails 的服务器）之间的半永久性连接，我们就是利用它来实现“登录”这一功能的。网络中常见的 session 处理方式有好几种：可以在用户关闭浏览器后清除 session；也可以提供一个“记住我”单选框让用户选择永远保存，直到用户退出后 session 才会失效。[1] 在示例程序中我们选择使用第二种处理方式，即用户登录后，会永久的记住登录状态，直到用户点击“退出”链接之后才清除 session。（在 8.2.1 节中会介绍“永久”到底有多久。）

很显然，我们可以把 session 视作一个符合 REST 架构的资源，在登录页面中准备一个新的 session，登录后创建这个 session，退出则会销毁 session。不过 session 和 Users 资源有所不同，Users 资源使用数据库（通过 User 模型）持久的存储数据，而 Sessions 资源是利用 cookie 来存储数据的。cookie 是存储在浏览器中的简单文本。实现登录功能基本上就是在实现基于 cookie 的验证机制。在本节及接下来的一节中，我们会构建 Sessions 控制器，创建登录表单，还会实现控制器中相关的动作。在 8.2 节中会加入处理 cookie 所需的代码。

8.1.1 Sessions 控制器

登录和退出功能其实是由 Sessions 控制器中相应的动作处理的，登录表单在 `new` 动作中处理（本节的内容），登录的过程就是向 `create` 动作发送 `POST` 请求（8.1 节和 8.2 节），退出则是向 `destroy` 动作发送 `DELETE` 请求（8.2.6 节）。（HTTP 请求和 REST 动作之间的对应关系可以查看表格 7.1。）首先，我们要生成 Sessions 控制器，以及验证系统所需的集成测试：

1. 另外一个常见的 session 处理方式是，在一定时间之后失效。这种方式特别适合包含敏感信息的网站，例如银行和交易账户。

```
$ rails generate controller Sessions --no-test-framework
$ rails generate integration_test authentication_pages
```

参照 7.2 节中的“注册”页面，我们要创建一个登录表单，用来生成新的 session。注册表单的构思图如图 8.1 所示。

“登录”页面的地址由 `signin_path`（稍后定义）获取，和之前一样，我们要先编写相应的测试，如代码 8.1 所示。（可以和代码 7.6 中对“注册”页面的测试比较一下。）

图 8.1：注册表单的构思图

代码 8.1：对 `new` 动作和对应视图的测试
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do

  subject { page }

  describe "signin page" do
```

```
    before { visit signin_path }

    it { should have_content('Sign in') }
    it { should have_title('Sign in') }
  end
end
```

现在测试是失败的：

```
$ bundle exec rspec spec/
```

要让代码 8.1 中的测试通过，首先，我们要为 Sessions 资源设置路由，还要修改“登录”页面具名路由的名称，将其映射到 Sessions 控制器的 new 动作上。和 Users 资源一样，我们可以使用 `resources` 方法设置标准的 REST 动作：

```
resources :sessions, only: [:new, :create, :destroy]
```

因为我们没必要显示或编辑 session，所以我们对动作的种类做了限制，为 `resources` 方法指定了 `:only` 选项，只创建 `new`、`create` 和 `destroy` 动作。最终的结果，包括登录和退出具名路由的设置，如代码 8.2 所示。

代码 8.2： 设置 session 相关的路由
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users
  resources :sessions, only: [:new, :create, :destroy]
  root to: 'static_pages#home'
  match '/signup',  to: 'users#new',            via: 'get'
  match '/signin',  to: 'sessions#new',         via: 'get'
  match '/signout', to: 'sessions#destroy',     via: 'delete'
  .
  .
  .
end
```

注意，设置退出路由那行使用了 `via: 'delete'`，这个参数指明 `destroy` 动作要使用 `DELETE` 请求。

代码 8.2 中的路由设置会生成类似表格 7.1 所示的URI 地址和动作的对应关系，如表格 8.1 所示。注意，我们修改了登录和退出具名路由，而创建 session 的路由还是使用默认值。

为了让代码 8.1 中的测试通过，我们还要在 Sessions 控制器中加入 `new` 动作，相应的代码如代码 8.3 所示（同时也定义了 `create` 和 `destroy` 动作）。

代码 8.3： 没什么内容的 Sessions 控制器
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  def new
```

```
  end

  def create
  end

  def destroy
  end
end
```

表格 8.1：代码 8.2 中的设置生成的符合 REST 架构的路由关系

HTTP 请求	URL 地址	具名路由	动作	目的
GET	/signin	signin_path	new	创建新 session 的页面（登录）
POST	/sessions	sessions_path	create	创建 session
DELETE	/signout	signout_path	destroy	删除 session（退出）

接下来还要创建“登录”页面的视图，因为“登录”页面的目的是创建新 session，所以创建的视图位于 `app/views/sessions/new.html.erb`。在视图中我们要显示网页的标题和一个一级标头，如代码 8.4 所示。

代码 8.4：“登录”页面的视图
`app/views/sessions/new.html.erb`

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>
```

现在代码 8.1 中的测试应该可以通过了，接下来我们要编写登录表单。

```
$ bundle exec rspec spec/
```

8.1.2 测试登录功能

对比图 8.1 和图 7.11 之后，我们发现登录表单和注册表单外观上差不多，只是少了两个字段，只有 Email 地址和密码字段。和注册表单一样，我们可以使用 Capybara 填写表单，再点击按钮进行测试。

在测试的过程中，我们不得不向程序中加入相应的功能，这也正是 TDD 带来的好处之一。我们先来测试填写不合法数据的登录过程，构思图如图 8.2 所示。

从图 8.2 我们可以看出，如果提交的数据不正确，我们会重新渲染“注册”页面，还会显示一个错误提示消息。这个错误提示是 Flash 消息，我们可以通过下面的测试验证：

```
it { should have_selector('div.alert.alert-error', text: 'Invalid') }
```

这行代码使用了代码 7.32（在第 7 章的练习中）中用过的 `have_selector` 方法。`have_selector` 会检查页面中是否出现了指定的元素（例如，一个 HTML 标签，不过在 Capybara 2.0 中只会检查**可见**的元素） 本例我们要查找的元素是：

```
div.alert.alert-error
```

上面这行代码会寻找一个 `div` 标签。前面介绍过，这里的点号代表 CSS 中的 class（参见 5.1.2 节），你也许猜到了，这里我们要查找的是同时具有 `alert` 和 `alert-error` class 的 `div` 元素。而且我们还检测了错误提示消息中是否包含了 `"Invalid"` 这个词。所以，上述测试是检测页面中是否有下面这个元素的：

```
<div class="alert alert-error">Invalid...</div>
```

图 8.2：注册失败页面的构思图

代码 8.5 是针对标题和 Flash 消息的测试。我们可以看出，这些代码缺少了一个很重要的部分，会在 8.1.5 节中说明。

代码 8.5：登录失败时的测试
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_title('Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }
    end
  end
end
```

测试了登录失败的情况，下面我们要测试登录成功的情况了。我们要测试登录成功后是否转向了用户资料页面（从页面的标题判断，标题中应该包含用户的名字），还要测试网站的导航中是否有以下三个变化：

1. 出现了指向用户资料页面的链接
2. 出现了“退出”链接
3. “登录”链接消失了

（对“设置（Settings）”链接的测试会在 9.1 节中实现，对“所有用户（Users）”链接的测试会在 9.3 节中实现。）如上变化的构思图如图 8.3 所示。[2]注意，“退出”和“个人资料”链接位于“账户（Account）”下拉菜单中。在 8.2.4 节中会介绍如何通过 Bootstrap 实现这种下拉菜单。

对登录成功时的测试如代码 8.6 所示。

代码 8.6： 登录成功时的测试
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    before { visit signin_path }
    .
    .
    .
```

2. 图片来自 http://www.flickr.com/photos/hermanusbackpackers/3343254977/

```
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        fill_in "Email",    with: user.email.upcase
        fill_in "Password", with: user.password
        click_button "Sign in"
      end

      it { should have_title(user.name) }
      it { should have_link('Profile',     href: user_path(user)) }
      it { should have_link('Sign out',    href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
    end
  end
end
```

图 8.3： 登录成功后显示的用户资料页面构思图

在代码 8.6 中用到了 Capybara 提供的 `have_link` 方法，它的第一参数是链接文本，第二个参数是可选的 `:href`，指定链接的地址，因此如下的代码

```
it { should have_link('Profile', href: user_path(user)) }
```

确保了页面中有一个 `a` 元素，链接到指定的 URL 地址。这里我们要检测的是一个指向用户资料页面的链接。注意，我们调用 `upcase` 方法把用户的 Email 地址转换成了大写形式，确保在数据库中查询用户时不用担心大小写问题。

8.1.3 登录表单

写完测试之后，我们就可以创建登录表单了。在代码 7.17 中，注册表单使用了 `form_for` 帮助函数，并指定其参数为 `@user` 变量：

```
<%= form_for(@user) do |f| %>
.
.
.
<% end %>
```

注册表单和登录表单的区别在于，程序中没有 Session 模型，因此也就没有类似 `@user` 的变量。也就是说，在构建登录表单时，我们要给 `form_for` 提供更多的信息。一般来说，如下的代码

```
form_for(@user)
```

Rails 会自动向 /users 地址发送 `POST` 请求。对于登录表单，我们则要明确的指定资源的名称以及相应的 URL 地址：

```
form_for(:session, url: sessions_path)
```

（创建表单还有另一种方法，不用 `form_for`，而用 `form_tag`。`form_tag` 也是 Rails 程序常用的方法，不过换用 `form_tag` 之后就和注册表单有很多不同之处了，我现在是想使用相似的代码构建登录表单。使用 `form_tag` 构建登录表单会留作练习（参见 8.5 节）。）

使用上述这种 `form_for` 形式，参照代码 7.17 中的注册表单，很容易的就能编写一个符合图 8.1 的登录表单，如代码 8.7 所示。

代码 8.7：注册表单的代码
`app/views/sessions/new.html.erb`

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>

<div class="row">
  <div class="span6 offset3">
```

```
    <%= form_for(:session, url: sessions_path) do |f| %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.submit "Sign in", class: "btn btn-large btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

注意，为了访客的便利，我们还加入了到“注册”页面的链接。代码 8.7 中的登录表单效果如图 8.4 所示。

图 8.4：登录表单（/signup）

用的多了你就不会老是查看 Rails 生成的 HTML（你会完全信任所用的帮助函数可以正确的完成任务），不过现在还是来看一下登录表单的 HTML 吧（如代码 8.8 所示）。

代码 8.8：代码 8.7 中登录表单生成的 HTML

```
<form accept-charset="UTF-8" action="/sessions" method="post">
  <div>
    <label for="session_email">Email</label>
    <input id="session_email" name="session[email]" type="text" />
  </div>
  <div>
    <label for="session_password">Password</label>
    <input id="session_password" name="session[password]"
           type="password" />
  </div>
  <input class="btn btn-large btn-primary" name="commit" type="submit"
       value="Sign in" />
</form>
```

你可以对比一下代码 8.8 和代码 7.20。你可能已经猜到了，提交登录表单后会生成一个 `params` Hash，其中 `params[:session][:email]` 和 `params[:session][:password]` 分别对应了 Email 和密码字段。

8.1.4 分析表单提交

和创建用户类似，创建 session 时先要处理提交不合法数据的情况。我们已经编写了对提交不合法数据的测试（参见代码 8.5），也添加了有几处难理解但还算简单的代码让测试通过了。下面我们就来分析一下表单提交的过程，然后为登录失败添加失败提示信息（如图 8.2）。最后，以此为基础，验证提交的 Email 和密码，处理登录成功的情况（参见 8.2 节）。

首先，我们来编写 Sessions 控制器的 `create` 动作，如代码 8.9 所示，现在只是直接渲染登录页面。在浏览器中访问 /sessions/new，然后提交空表单，显示的页面如图 8.5 所示。

代码 8.9： Sessions 控制器中 `create` 动作的初始版本
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    render 'new'
  end
  .
  .
  .
end
```

图 **8.5**：代码 8.9 中的 `create` 动作显示的登录失败后的页面

仔细看一下图 8.5 中显示的调试信息，你会发现，如在 8.1.3 节末尾说过的，表单提交后会生成 `params` Hash，Email 和密码都在 `:session` 键中：

```
---
session:
  email: ''
  password: ''
commit: Sign in
action: create
controller: sessions
```

和注册表单类似，这些参数是一个嵌套的 Hash，在代码 4.6 中见过。`params` 包含了如下的嵌套 Hash：

```
{ session: { password: "", email: "" } }
```

也就是说

```
params[:session]
```

本身就是一个 Hash：

```
{ password: "", email: "" }
```

所以，

```
params[:session][:email]
```

就是提交的 Email 地址，而

```
params[:session][:password]
```

就是提交的密码。

也就是说，在 `create` 动作中，`params` 包含了使用 Email 和密码验证用户身份所需的全部数据。幸运的是，我们已经定义了身份验证过程中所需的两个方法，即由 Active Record 提供的 `User.find_by_email`（参见 6.1.4 节），以及由 `has_secure_password` 提供的 `authenticate` 方法（参见 6.3.3 节）。我们之前介绍过，如果提交的数据不合法，`authenticate` 方法会返回 `false`。基于以上的分析，我们计划按照如下的方式实现用户登录功能：

```
def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    # Sign the user in and redirect to the user's show page.
  else
    # Create an error message and re-render the signin form.
  end
end
```

`create` 动作的第一行，使用提交的 Email 地址从数据库中取出相应的用户。（我们在 6.2.5 节讲过，Email 地址都是以小写字母形式保存的，所以这里调用了 `downcase` 方法，确保提交合法 Email 地址后能查询到相应的记录。）第二行看起来很怪，但在 Ruby 中经常使用：

```
user && user.authenticate(params[:session][:password])
```

表格 8.2：`user && user.authenticate(...)` 可能出现的结果

用户	密码	**a && b**
不存在	任意值	`nil && [anything] == false`
存在	错误的密码	`true && false == false`
存在	正确的密码	`true && true == true`

我们使用 `&&`（逻辑与）检测获取的用户是否合法。因为除了 `nil` 和 `false` 之外的所有对象都被视作 `true`，上面这个语句可能出现的结果如表格 8.2所示。我们可以从表格 8.2 中看出，当且仅当数据库中存在提交的 Email 并提交了对应的密码时，这个语句才会返回 `true`。

8.1.5 显示 Flash 消息

在 7.3.3 节中，我们使用 User 模型的数据验证信息来显示注册失败时的提示信息。这些错误提示信息是关联在某个 Active Record 对象上的，不过这种方式不可以用在 session 上，因为 session 不是 Active Record 模型。我们要采取的方法是，在登录失败时，把错误提示信息赋值给 Flash 消息。代码 8.10 显示的是我们首次尝试实现这种方法所用的代码，其中有个小小的错误。

图 **8.6**：登录失败后显示的 Flash 消息

代码 8.10：尝试处理登录失败（有个小小的错误）
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
```

```
    if user && user.authenticate(params[:session][:password])
      # Sign the user in and redirect to the user's show page.
    else
      flash[:error] = 'Invalid email/password combination' # Not quite right!
      render 'new'
    end
  end

  def destroy
  end
end
```

布局中已经加入了显示 Flash 消息的局部视图（代码 7.27），所以无需其他修改，上述 Flash 错误提示消息就会显示出来，而且因为使用了 Bootstrap，这个错误消息的样式也很美观（如图 8.6）。

不过，就像代码 8.10 中的注释所说，这些代码还有问题。显示的页面看起来很正常啊，那么，问题出现在哪儿呢？问题的关键在于，Flash 消息在一个请求的生命周期内是持续存在的，而重新渲染页面（使用 `render` 方法）和代码 7.28 中的转向不同，它不算新的请求，你会发现这个 Flash 消息存在的时间比设想的要长很多。例如，我们提交了不合法的登录信息，Flash 消息生成了，然后在登录页面中显示出来（如图 8.6），这时如果我们点击链接转到其他页面（例如“首页”），这只算是表单提交后的第一次请求，所以页面中还是会显示 Flash 消息（如图 8.7）。

图 8.7：仍然显示有 Flash 消息的页面

Flash 消息没有按预期消失算是程序的一个 bug，在修正之前，我们最好编写一个测试来捕获这个错误。现在，登录失败时的测试是可以通过的：

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "signin with invalid information"
```

不过程序中有错误，测试应该是失败的，所以我们要编写一个能够捕获这种错误的测试。幸好，捕获这种错误正是集成测试的拿手好戏，所用的代码如下：

```
describe "after visiting another page" do
  before { click_link "Home" }
  it { should_not have_selector('div.alert.alert-error') }
end
```

提交不合法的登录信息之后，这个测试用例会点击网站中的“首页”链接，期望显示的页面中没有 Flash 错误消息。添加上述测试用例的测试文件如代码 8.11 所示。

代码 8.11：登录失败时的合理测试
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do

    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_title('Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }

      describe "after visiting another page" do
        before { click_link "Home" }
        it { should_not have_selector('div.alert.alert-error') }
      end
    end
    .
    .
    .
  end
end
```

新添加的测试和预期一致，是失败的：

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "signin with invalid information"
```

要让这个测试通过，我们要用 `flash.now` 替换 `flash`。`flash.now` 就是专门用来在重新渲染的页面中显示 Flash 消息的，在发送新的请求之后，Flash 消息便会消失。正确的 `create` 动作代码如代码 8.12 所示。

代码 8.12： 处理登录失败所需的正确代码
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Sign the user in and redirect to the user's show page.
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

现在登录失败时的所有测试应该都可以通过了：

```
$ bundle exec rspec spec/requests/authentication pages spec.rb \
> -e "with invalid information"
```

8.2 登录成功

上一节处理了登录失败的情况，这一节我们要处理登录成功的情况了。实现用户登录的过程是本书目前为止最考验 Ruby 编程能力的部分，你要坚持读完本节，做好心理准备，付出大量的脑力劳动。幸好，第一步还算是简单的，完成 Sessions 控制器的 `create` 动作没什么难的，不过还是需要一点小技巧。

我们需要把代码 8.12 中处理登录成功分支中的注释换成具体的代码，使用 `sign_in` 方法实现登录操作，然后转向用户的资料页面，如代码 8.13 所示。这就是我们使用的技巧，使用还没定义的方法 `sign_in`。本节后面的内容会定义这个方法。

代码 8.13： 完整的 `create` 动作代码（还不能正常使用）

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      sign_in user
      redirect_to user
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end
  .
  .
  .
end
```

8.2.1 “记住我”

现在我们要开始实现登录功能了，第一步是实现“记住我”这个功能，即用户登录的状态会被“永远”记住，直到用户点击“退出”链接为止。实现登录功能用到的函数已经超越了传统的 MVC 架构，其中一些函数要同时在控制器和视图中使用。在4.2.5 节中介绍过，Ruby 支持模块（module）功能，打包一系列函数，在不同的地方引入。我们会利用模块来打包用户身份验证相关的函数。我们当然可以创建一个新的模块，不过 Sessions 控制器已经提供了一个名为 `SessionsHelper` 的模块，而且这个模块中的帮助方法会自动引入 Rails 程序的视图中。所以，我们就直接使用这个现成的模块，然后在 Application 控制器中引入，如代码 8.14 所示。

代码 8.14： 在 Application 控制器中引入 Sessions 控制器的帮助方法模块
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper
end
```

默认情况下帮助函数只可以在视图中使用，不能在控制器中使用，而我们需要同时在控制器和视图中使用帮助函数，所以我们就手动引入帮助函数所在的模块。

因为 HTTP 是无状态的协议，所以如果应用程序需要实现登录功能的话，就要找到一种方法记住用户的状态。维持用户登录状态的方法之一，是使用常规的 Rails session（通过 `session` 函数），把用户的 id 保存在“记忆权标（remember token）”中：

```
session[:remember_token] = user.id
```

session 对象把用户 id 保存在浏览器的 cookie 中，这样在网站的所有页面就都可以使用了。浏览器关闭后，cookie 也随之失效。在网站中的任何页面，只需调用 User.find(session[:remember_token]) 就可以取回用户对象了。Rails 在处理 session 时，会确保安全性。倘若用户企图伪造用户 id，Rails 可以通过每个 session 的 session id 检测到。

根据示例程序的设计目标，我们计划要实现的是持久保存的 session，即使浏览器关闭了，登录状态依旧存在，所以，登入的用户要有一个持久保存的标识符才行。为此，我们要为每个用户生成一个唯一而安全的记忆权标，长期存储，不会随着浏览器的关闭而消失。

记忆权标要附属到特定的用户对象上，而且要保存起来以待后用，所以我们就可以把它设为 User 模型的属性（如图 8.8）。我们先来编写 User 模型的测试，如代码 8.15 所示。

users	
id	integer
name	string
email	string
password_digest	string
remember_token	string
created_at	datetime
updated_at	datetime

图 8.8： User 模型，添加了 remember_token 属性

代码 8.15： 记忆权标的第一个测试
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:password_confirmation) }
  it { should respond_to(:remember_token) }
  it { should respond_to(:authenticate) }
  .
  .
  .
end
```

要让这个测试通过，我们要生成记忆权标属性，执行如下命令：

```
$ rails generate migration add_remember_token_to_users
```

然后按照代码 8.16 修改生成的迁移文件。注意，因为我们要使用记忆权标取回用户，所以我们为 remember_token 列加了索引（参见 旁注 6.2）。

代码 8.16： users 表添加 remember_token 列的迁移
db/migrate/[timestamp]_add_remember_token_to_users.rb

```
class AddRememberTokenToUsers < ActiveRecord::Migration
  def change
    add_column :users, :remember_token, :string
    add_index :users, :remember_token
  end
end
```

然后，还要更新“开发数据库”和“测试数据库”：

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

现在，User 模型的测试应该可以通过了：

```
$ bundle exec rspec spec/models/user_spec.rb
```

接下来我们要考虑记忆权标要保存什么数据，这有很多种选择，其实任何足够长的随机字符串都是可以的，只要能保证唯一性。Ruby 标准库中 `SecureRandom` 模块提供的 `urlsafe_base64` 方法可以满足我们的要求。[3] `SecureRandom.urlsafe_base64` 创建的字符串长度为 16，由 A-Z、a-z、0-9、下划线（_）和连字符（-）组成（每一位字符都有 64 种可能的情况，所以叫做“base64”）.所以两个记忆权标相等的概率就是 $1/64^{16} = 2^{-96} \approx 10^{-29}$，完全可以忽略。

我们计划在浏览器中存储这个 base64 权标，在数据库中存储加密后的版本。如果要自动登入用户，就可以从 cookie 中取出记忆权标，加密后查询数据库。数据库之所以只保存加密后的权标是因为，即便整个数据库都泄露了，攻击者也无法使用记忆权标登入网站。为了让记忆权标更安全，我们计划每次会话都生成不一样的权标，这样即使会话被劫持了（攻击者偷取 cookie 伪装成某个用户登录），用户下次登录时前一个会话就会失效。（会话劫持可以通过 Firesheep 扩展检测，使用这个扩展可以看到很多著名的网站在连接到公共 WIFI 时，记忆权标都是可见的。为了避免这个问题可以全站使用 SSL，在 7.4.4 节有介绍。）

真实的应用程序都会自动登入刚注册的用户（这样做的一个副作用就是创建了一个新的记忆权标），但是我们不想这么做，我们要用一种更好的方式，确保从一开始用户就有可用的记忆权标。为此，我们要使用回调函数生成权标。在 6.2.5 节中确保 Email 唯一性时用过回调函数，调用的是 `before_save`。现在我们要调用 `before_create` 在创建用户时设置记忆权标。[4]

要测试这个过程，我们可以先保存测试所需的用户对象，然后检查 `remember_token` 是否为非空值。这样做，如果以后需要改变记忆权标的生成方式，也无需修改测试。测试代码如代码 8.17 所示。

代码 8.17：测试合法的（非空）记忆权标值
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
```

3. 我选择这么生成记忆权标是因为看了 Railscasts 第 274 集《Remember Me & Reset Password》。
4. Active Record 支持的其他回调函数在 Rails 指南中有介绍。

```
  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }
  .
  .
  .
  describe "remember token" do
    before { @user.save }
    its(:remember_token) { should_not be_blank }
  end
end
```

代码 8.17 中用到了 its 方法，它和 it 很像，不过测试对象是参数中指定的属性而不是整个测试的对象。也就是说，如下的代码：

```
its(:remember_token) { should_not be_blank }
```

等同于

```
it { expect(@user.remember_token).not_to be_blank }
```

User 模型的代码会涉及到一些新的知识。其一，我们添加了一个回调函数，在用户存入数据库之前生成记忆权标：

```
before_create :create_remember_token
```

上面这行代码叫做“方法引用”。当 Rails 执行到这行代码时，会寻找一个名为 create_remember_token 的方法，在创建用户之前执行。（在代码 6.20 中，我们直接给 before_save 指定了块参数，但更推荐使用这种“方法引用”方式。）而且，create_remember_token 只会在 User 模型内部使用，所以没必要把它开放给用户之外的对象。如 7.3.2 节中所讲，在 Ruby 中，我们可以使用 private 关键字[5]限制方法的可见性：

```
pivate

  def create_remember_token
    # Create the token.
  end
```

在类中，private 之后定义的方法都会被设为私有方法，所以，如果执行下面的操作

```
$ rails console
>> User.first.create_remember_token
```

5. 译者注：其实 private 是方法而不是关键字，请参阅《Ruby 编程语言》P233

就会抛出 NoMethodError 异常。

在 create_remember_token 方法中，要给用户的属性赋值，需要在 remember_token 前加上 self 关键字：

```
def User.new_remember_token
  SecureRandom.urlsafe_base64
end

def User.encrypt(token)
  Digest::SHA1.hexdigest(token.to_s)
end

private

  def create_remember_token
    self.remember_token = User.encrypt(User.new_remember_token)
  end
```

Active Record 是把模型的属性和数据库表中的列对应的，如果不指定 self 的话，我们就只是创建了一个名为 remember_token 的局部变量而已，这可不是我们期望得到的结果。加上 self 之后，赋值操作就会把值赋值给用户的 remember_token 属性，保存用户时，随着其他的属性一起存入数据库。（现在你知道了为什么代码 6.20 中的 before_save 为什么要用 self.name 而不直接用 email。）

注意，我们使用 SHA1 加密了记忆权标。这种哈希算法比 6.3.1 节中用来加密用户密码的 Bcrypt 速度快得多。之所以看中速度，是因为对登入的用户来说，每个页面都会加密记忆权标（参加8.2.2 节）。SHA1 的安全性不如 Bcrypt，但能满足现在的需求，加密后的记忆权标是 16 位随机字符，基本无法破解。（上述代码之所以调用 to_s，是为了处理输入为 nil 的情况。对然在浏览器中不会遇到，到测试时偶尔会出现。）

encrypt 和 new_remember_token 方法定义在 User 类中，因为二者无需用户实例就可使用[6]，而且还是公开方法（位于 private 上面）。在 8.2.3 节中会将这两个方法剥离出 User 模型。

把上述的分析结合起来，最终得到的 User 模型文件如代码 8.18 所示。

代码 8.18： 生成记忆权标的 before_create 回调函数
app/models/user.rb

```
class User < ActiveRecord::Base
  before_save { self.email = email.downcase }
  before_create :create_remember_token
  .
  .
  .
  def User.new_remember_token
    SecureRandom.urlsafe_base64
  end
```

6. 其实这两种方式是完全等效的，attr_accessor 会自动创建取值和设定方法。

```
  def User.encrypt(token)
    Digest::SHA1.hexdigest(token.to_s)
  end

  private

    def create_remember_token
      self.remember_token = User.encrypt(User.new_remember_token)
    end
end
```

顺便说一下，我们为 `create_remember_token` 方法增加了一层缩进，这样可以更好的突出这些方法是在 `private` 之后定义的。（经验表明，这样做是明智地。）[7]

因为加密后的 `SecureRandom.urlsafe_base64` 方法创建的字符串不可能为空值，所以对 User 模型的测试现在应该可以通过了：

```
$ bundle exec rspec spec/models/user_spec.rb
```

8.2.2 定义 sign_in 方法

本小节我们要开始实现登录功能了，首先来定义 `sign_in` 方法。上一小节已经说明了，我们计划实现的身份验证方式是，在用户的浏览器中存储记忆权标，在网站的页面与页面之间通过这个记忆权标获取数据库中的用户记录（会在 8.2.3 节实现）。实现这一设想所需的代码如代码 8.19 所示，这段代码使用了 `current_user` 方法，会在 8.2.3 节中定义。

代码 8.19： 完整但还不能正常使用的 `sign_in` 方法
`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    remember_token = User.new_remember_token
    cookies.permanent[:remember_token] = remember_token
    user.update_attribute(:remember_token, User.encrypt(remember_token))
    self.current_user = user
  end
end
```

上述代码组找了设定的步骤：首先，创建新权标；随后，把未加密的权标存入浏览器的 cookie；然后，把加密后的权标存入数据库；最后，把制定的用户设为当前登入的用户。我们会在 8.2.3 节中看到，现在并不一定要把 `user` 指定为当前用户，因为 `create` 动作完成后回立即转向（代码 8.13）。不过指定也好，防止无需转向的登录操作。

7. 译者注：如果按照 bbatsov 的《Ruby 编程风格指南》来编写 Ruby 代码的话，就没必要多加一层缩进。

注意，保存记忆权标使用的是 `update_attribute` 方法，这样可以跳过数据验证更新单个属性。我们必须用这个方法，因为我们无法提供用户的密码及密码确认。代码 8.19 中用到的 `cookies` 方法是由 Rails 提供的，我们可以把它看成 Hash，其中每个元素又都是一个 Hash，包含两个元素，`value` 指定 cookie 的文本，`expires` 指定 cookie 的失效日期。例如，我们可以使用下述代码实现登录功能，把 cookie 的值设为用户的记忆权标，失效日期设为 20 年之后：

```
cookies[:remember_token] = { value:   remember_token,
                             expires: 20.years.from_now.utc }
```

（这里使用了 Rails 提供的时间帮助方法，详情参见旁注 8.1。）

旁注 8.1：cookie 在 20.years.from_now 之后失效

在 4.4.2 节中介绍过，你可以向任何的 Ruby 类，甚至是内置的类中添加自定义的方法，我们就向 `String` 类添加了 `palindrome?` 方法（而且还发现了 `"deified"` 是回文）。我们还介绍过，Rails 为 `Object` 类添加了 `blank?` 方法（所以，`"".blank?`、`" ".blank?` 和 `nil.blank?` 的返回值都是 `true`）。代码 8.19 中处理 cookie 的代码又是一例，使用了 Rails 提供的时间帮助方法，这些方法是添加到 `Fixnum` 类（数字的基类）中的。

```
$ rails console
>> 1.year.from_now
=> Sun, 13 Mar 2011 03:38:55 UTC +00:00
>> 10.weeks.ago
=> Sat, 02 Jan 2010 03:39:14 UTC +00:00
```

Rails 还添加了其他的帮助函数，如：

```
>> 1.kilobyte
=> 1024
>> 5.megabytes
=> 5242880
```

这几个帮助函数可用于限制上传文件的大小，例如，图片最大不超过 `5.megabytes`。

这种为内置类添加方法的特性很灵便，可以扩展 Ruby 的功能，不过使用时要小心一些。其实 Rails 的很多优雅之处正式基于 Ruby 语言的这一特性。

因为开发者经常要把 cookie 的失效日期设为 20 年后，所以 Rails 特别提供了 `permanent` 方法，前面处理 cookie 的代码可以改写成：

```
cookies.permanent[:remember_token] = remember_token
```

Rails 的 `permanent` 方法会自动把 cookie 的失效日期设为 20 年后。

设定了 cookie 之后，在网页中我们就可以使用下面的代码取回用户：

```
User.find_by(remember_token: remember_token)
```

其实浏览器中保存的 cookie 并不是 Hash，赋值给 `cookies` 只是把值以文本的形式保存在浏览器中。这正体现了 Rails 的智能，我们无需关心具体的处理细节，专注地实现应用程序的功能。

8.2.3 获取当前用户

上一小节已经介绍了如何在 cookie 中存储记忆权标以待后用，这一小节我们要看一下如何取回用户。我们先回顾一下 `sign_in` 方法：

```
module SessionsHelper

  def sign_in(user)
    remember_token = User.new_remember_token
    cookies.permanent[:remember_token] = remember_token
    user.update_attribute(:remember_token, User.encrypt(remember_token))
    self.current_user = user
  end
end
```

现在还无法使用的代码是：

```
self.current_user = user
```

我们在代码 8.19 后面说过，这行代码并不会真的在这个程序中使用，因为登录后会直接转向。如果 `sign_in` 方法只考虑转向一种情况也是很危险的。

`current_user` 方法可以在控制器和视图中使用，所以你既可以这样用：

```
<%= current_user.name %>
```

也可以这样用：

```
redirect_to current_user
```

这行代码中的 `self` 也是必须的，原因在分析代码 8.18 时已经说过，如果没有 `self`，Ruby 只是定义了一个名为 `current_user` 的局部变量。

在开始编写 `current_user` 方法的代码之前，请仔细看这行代码：

```
self.current_user = user
```

这是一个赋值操作，我们必须先定义相应的方法才能这么用。Ruby 为这种赋值操作提供了一种特别的定义方式，如代码 8.20 所示。

代码 8.20：实现 current_user 方法对应的赋值操作
app/helpers/sessions_helper.rb

```
module SessionsHelper

  def sign_in(user)
    .
    .
    .
  end

  def current_user=(user)
    @current_user = user
  end
end
```

这段代码看起来很奇怪，因为大多数的编程语言并不允许在方法名中使用等号。其实这段代码定义的 current_user= 方法是用来处理 current_user 赋值操作的。也就是说，如下的代码

```
self.current_user = ...
```

会自动转换成下面这种形式

```
current_user=(...)
```

就是直接调用 current_user= 方法，接受的参数是赋值语句右侧的值，本例中是要登录的用户对象。current_user= 方法定义体内只有一行代码，即设定实例变量 @current_user 的值，以备后用。

在常见的 Ruby 代码中，我们还会定义 current_user 方法，用来读取 @current_user 的值，如代码 8.21 所示。

代码 8.21：尝试定义 current_user 方法，不过我们不会使用这种方式

```
module SessionsHelper

  def sign_in(user)
    .
    .
    .
  end

  def current_user=(user)
    @current_user = user
  end
```

```
  def current_user
    @current_user # Useless! Don't use this line.
  end
end
```

上面的做法其实就是实现了 `attr_accessor` 方法的功能（4.4.5 节介绍过）。[8]如果按照代码 8.21 来定义 `current_user` 方法，会出现一个问题：程序不会记住用户的登录状态。一旦用户转到其他的页面，session 就失效了，会自动退出。若要避免这个问题，我们要使用代码 8.19 中生成的记忆权标查找用户，如代码 8.22 所示。注意，因为数据库中保存的记忆权标是加密的，所以在用来查找用户之前要加密从 cookie 中读取的权标。我们可以使用代码 8.18 中定义的 `User.encrypt` 方法加密。

代码 8.22：通过记忆权标查找当前用户
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  def current_user=(user)
    @current_user = user
  end

  def current_user
    remember_token = User.encrypt(cookies[:remember_token])
    @current_user ||= User.find_by(remember_token: remember_token)
  end
end
```

代码 8.22 中使用了一个常见但不是很容易理解的 `||=`（“or equals”）操作符（旁注 8.2中有详细介绍）。使用这个操作符之后，当且仅当 `@current_user` 未定义时才会把通过记忆权标获取的用户赋值给实例变量 `@current_user`。[9]也就是说，如下的代码

```
@current_user ||= User.find_by(remember_token: remember_token)
```

只在第一次调用 `current_user` 方法时调用 `find_by` 方法，如果后续再调用的话就直接返回 `@current_user` 的值，而不必再查询数据库。[10]这种方式的优点只有当在一个请求中多次调用 `current_user` 方法时才能显现。不管怎样，只要用户访问了相应的页面，`find_by` 方法都至少会执行一次。

8. 如果方法不需要在实例上调用，就要定义为类方法。
9. 这也是一种备忘（memoization），详情参见旁注 6.3。
10. 一般来说，这句话的意思是把初始值为 `nil` 的变量附上了新值，不过 `||=` 也会把初始值为 `false` 的变量附上新值。

旁注 8.2：||= 操作符简介

||= 操作符非常能够体现 Ruby 的特性，如果你打算长期进行 Ruby 编程的话就要好好学习它的用法。初学时会觉得 ||= 很神秘，不过通过和其他操作符类比之后，你会发现也不是很难理解。

我们先来看一下改变已经定义的变量时经常使用的结构。在很多程序中都会把变量自增一，如下所示

```
x = x + 1
```

大多数语言都为这种操作提供了简化的操作符，在 Ruby 中，可以按照下面的方式重写（C、C++、Perl、Python、Java 等也如此）：

```
x += 1
```

其他操作符也有类似的简化形式：

```
$ rails console
>> x = 1
=> 1
>> x += 1
=> 2
>> x *= 3
=> 6
>> x -= 7
=> -1
```

上面的举例可以概括为，`x = x o y` 和 `x o=y` 是等效的，其中 `o` 表示操作符。

在 Ruby 中还经常会遇到这种情况，如果变量的值为 `nil` 则赋予其他的值，否则就不改变这个变量的值。4.2.3 节 中介绍过 || 或操作符，所以这种情况可以用如下的代码表示：

```
>> @user
=> nil
>> @user = @user || "the user"
=> "the user"
>> @user = @user || "another user"
=> "the user"
```

因为 `nil` 表示的布尔值是 `false`，所以第一个赋值操作等同于 `nil || "the user"`，这个语句的计算结果是 `"the user"`；类似的，第二个赋值操作等同于 `"the user" || "another user"`，这个语句的计算结果还是 `"the user"`，因为 `"the user"` 表示的布尔值是 `true`，这个或操作在执行了第一个表达式之后就终止了。（或操作的执行顺序是从左至右，只要出现真值就会终止语句的执行，这种方式称作“短路计算（short-circuit evaluation）”。）

和上面的控制台会话对比之后，我们可以发现 `@user = @user || value` 符合 `x = x o y` 的形式，只需把 `o` 换成 `||`，所以就得到了下面这种简写形式：

```
>> @user ||= "the user"
=> "the user"
```

不难理解吧！[11]

8.2.4 改变导航链接

本小节我们要完成的是实现登录、退出功能的最后一步，根据登录状态改变布局中的导航链接。如图 8.3 所示，我们要在登录和退出后显示不同的导航，要添加指向列出所有用户页面的链接、到用户设置页面的链接（第 9 章加入），还有到当前登录用户资料页面的链接。加入这些链接之后，代码 8.6 中的测试就可以通过了，这是本章目前为止测试首次变绿通过。

在网站的布局中改变导航链接需要用到 ERb 的 if-else 分支结构：

```
<% if signed_in? %>
# Links for signed-in users
<% else %>
# Links for non-signed-in-users
<% end %>
```

若要上述代码起作用，先要用 `signed_in?` 方法。我们现在就来定义。

如果 session 中存有当前用户的话，就可以说用户已经登录了。我们要判断 `current_user` 的值是不是 `nil`，这里需要用到取反操作符，用感叹号 ! 表示，一般读作“bang”。只要 `current_user` 的值不是 `nil`，就说明用户登录了，如代码 8.23 所示。

代码 8.23： 定义 `signed_in?` 帮助方法
`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    remember_token = User.new_remember_token
    cookies.permanent[:remember_token] = remember_token
    user.update_attribute(:remember_token, User.encrypt(remember_token))
    self.current_user = user
  end
```

11. 译者注：这里对 `||=` 的分析和 Peter Cooper 的分析有点差异，我推荐你看以下 Ruby Inside 中的《What Ruby's ||= (Double Pipe / Or Equals) Really Does》一文。

```
  def signed_in?
    !current_user.nil?
  end
  .
  .
  .
end
```

定义了 signed_in? 方法后就可以着手修改布局中的导航了。我们要添加四个新链接，其中两个链接的地址先不填（第 9 章再填）：

```
<%= link_to "Users", '#' %>
<%= link_to "Settings", '#' %>
```

退出链接的地址使用代码 8.2 中定义的 signout_path：

```
<%= link_to "Sign out", signout_path, method: "delete" %>
```

（注意，我们还为退出链接指定了类型为 Hash 的参数，指明点击链接后发送的是 HTTP DELETE 请求。[12]）最后，我们还要添加一个到资料页面的链接：

```
<%= link_to "Profile", current_user %>
```

这个链接我们本可以写成

```
<%= link_to "Profile", user_path(current_user) %>
```

不过我们可以直接把链接地址设为 current_user，Rails 会自动将其转换成 user_path(current_user)。

在添加导航链接的过程中，我们还要使用 Bootstrap 实现下拉菜单的效果，具体的实现方式可以参阅 Bootstrap 的文档。添加导航链接所需的代码如代码 8.24 所示。注意其中和 Bootstrap 下拉菜单有关的 CSS id 和 class。

代码 8.24： 根据登录状态改变导航链接
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", '#' %></li>
```

12. 浏览器其实并不能发送 DELETE 请求，Rails 是通过 JavaScript 模仿的。

```
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", '#' %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

实现下拉菜单还要用到 Bootstrap 中的 JavaScript 代码，我们可以编辑应用程序的 JavaScript 文件，通过 asset pipeline 引入所需的文件，如代码 8.25 所示。

图 8.9：登录后显示了新链接和下拉菜单

代码 8.25：把 Bootstrap 的 JavaScript 代码加入 `application.js`
`app/assets/javascripts/application.js`

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require turbolinks
//= require_tree .
```

引入文件的功能是由 Sprockets 实现的，而文件本身是由 5.1.2 节中添加的 `bootstrap-sass` gem 提供的。

添加了代码 8.24 之后，所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

现在登入的用户就可以看到代码 8.24 新添加的链接和下拉菜单了，如图 8.9 所示。

现在你可以验证一下是否可以登录，然后关闭浏览器，再打开看一下是否还是登入的状态。如果需要，你还可以直接查看浏览器的 cookies，如图 8.10 所示。

图 **8.10**：查看浏览器中的记忆权标 cookie

8.2.5 注册后直接登录

虽然现在基本完成了用户身份验证功能，但是新注册的用户可能还是会困惑，为什么注册后没有登录呢。在实现退出功能之前，我们还要实现注册后直接登录的功能。我们要先编写测试，在身份验证的测试中加入一行代码，如代码 8.26 所示。这段代码要用到第 7 章一个练习中的“after saving the user” `describe` 块（参见代码 7.32），如果之前你没有做这个练习的话，现在请添加相应的测试代码。

代码 8.26：测试刚注册的用户是否会自动登录
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "after saving the user" do
        before { click_button submit }
```

```
        let(:user) { User.find_by(email: 'user@example.com') }

        it { should have_link('Sign out') }
        it { should have_title(user.name) }
        it { should have_selector('div.alert.alert-success', text: 'Welcome') }
      end
    end
  end
end
```

我们检测页面中有没有退出链接，来验证用户注册后是否登录了。

有了 8.2 节中定义的 `sign_in` 方法，要让这个测试通过就很简单了：在用户保存到数据库中之后加上 `sign_in @user` 就可以了，如代码 8.27 所示。

代码 8.27： 用户注册后直接登录
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      sign_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end
  .
  .
  .
end
```

8.2.6 退出

在 8.1 节中介绍过，我们要实现的身份验证机制会记住用户的登录状态，直到用户点击退出链接为止。本小节，我们就要实现退出功能。

目前为止，Sessions 控制器的动作完全遵从了 REST 架构，`new` 动作用于登录页面，`create` 动作实现登录的过程。我们还要添加一个 `destroy` 动作，删除 session，实现退出功能。针对退出功能的测试，我们可以检测点击退出链接后，页面中是否有登录链接，如代码 8.28 所示。

代码 8.28： 测试用户退出

spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "followed by signout" do
        before { click_link "Sign out" }
        it { should have_link('Sign in') }
      end
    end
  end
end
```

登录功能是由 `sign_in` 方法实现的，对应的，我们会使用 `sign_out` 方法实现退出功能，如代码 8.29 所示。

代码 8.29：销毁 session，实现退出功能
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def destroy
    sign_out
    redirect_to root_path
  end
end
```

和其他身份验证相关的方法一样，我们会在 Sessions 控制器的帮助方法模块中定义 `sign_out` 方法。方法本身的实现很简单，我们先把当前用户设为 `nil`，然后在 cookies 上调用 `delete` 方法从 session 中删除记忆权标，如代码 8.30 所示。（其实这里没必要把当前用户设为 `nil`，因为在 `destroy` 动作中我们加入了转向操作。这里我们之所以这么做是为了兼容不转向的退出操作。）

代码 8.30：Sessions 帮助方法模块中定义的 `sign_out` 方法
app/helpers/sessions_helper.rb

```
module SessionsHelper
```

```
  def sign_in(user)
    remember_token = User.new_remember_token
    cookies.permanent[:remember_token] = remember_token
    user.update_attribute(:remember_token, User.encrypt(remember_token))
    self.current_user = user
  end
  .
  .
  .
  def sign_out
    self.current_user = nil
    cookies.delete(:remember_token)
  end
end
```

现在，注册、登录和退出三个功能都实现了，测试也应该可以通过了：

```
$ bundle exec rspec spec/
```

有一点需要注意，我们的测试覆盖了身份验证机制的大多数功能，但不是全部。例如，我们没有测试“记住我”到底记住了多久，也没测试是否设置了记忆权标。我们当然可以加入这些测试，不过经验告诉我们，直接测试 cookie 的值不可靠，而且要依赖具体的实现细节，而实现的方法在不同的 Rails 版本中可能会有所不同，即便应用程序可以使用，测试却会失败。所以我们只关注抽象的功能（验证用户是否可以登录，是否可以保持登录状态，以及是否可以退出），编写的测试没必要针对实现的细节。

8.3 Cucumber 简介（选读）

前面两节基本完成了示例程序的身份验证系统，这一节我们将介绍如何使用 Cucumber 编写登录测试。Cucumber 是一个流行的行为驱动开发（Behavior-driven Development, BDD）工具，在 Ruby 社区中占据着一定的地位。本节的内容是选读的，你可以直接跳过，不会影响后续内容。

Cucumber 使用纯文本的故事（story）描述应用程序的行为，很多 Rails 开发者发现使用 Cucumber 处理客户案例时十分方便，因为非技术人员也能读懂这些行为描述，Cucumber 测试可以用于和客户沟通，甚至经常是由客户来编写的。当然，使用不是纯 Ruby 代码组成的测试框架有它的局限性，而且我还发现纯文本的故事很啰嗦。不管怎样，Cucumber 在 Ruby 测试工具中还是有其存在意义的，我特别欣赏它对抽象行为的关注，而不是死盯底层的具体实现。

因为本书着重介绍的是 RSpec 和 Capybara，所以本节对 Cucumber 的介绍很浅显，也不完整，很多内容都没做详细说明，我只是想让你体验一下如何使用 Cucumber，如果你感觉不错，可以阅读专门介绍 Cucumber 的书籍深入学习。（一般我会推荐你阅读 David Chelimsky 的《The RSpec Book》，Ryan Bigg 和 Yehuda Katz 的《Rails 3 in Action》，以及 Matt Wynne 和 Aslak Hellesøy 的《The Cucumber Book》。）

8.3.1 安装和设置

若要安装 Cucumber，需要在 `Gemfile` 的 `:test` 组中加入 `cucumber-rails` 和 `database_cleaner` 这两个 gem，如代码 8.31 所示。

代码 8.31 在 `Gemfile` 中加入 `cucumber-rails`

```
.
.
.
group :test do
  .
  .
  .
  gem 'cucumber-rails', '1.4.0', :require => false
  gem 'database_cleaner', github: 'bmabey/database_cleaner'
end
.
.
.
```

然后和之前一样运行一下命令安装：

```
$ bundle install
```

如果要在程序中使用 Cucumber，我们先要生成一些所需的文件和文件夹：

```
$ rails generate cucumber:install
```

这个命令会在根目录中创建 `features` 文件夹，Cucumber 相关的文件都会存在这个文件夹中。

8.3.2 功能和步骤定义

Cucumber 中的“功能（feature）”就是希望应用程序实现的行为，使用一种名为 Gherkin 的纯文本语言编写。使用 Gherkin 编写的测试和写的很好的 RSpec 测试用例差不多，不过因为 Gherkin 是纯文本，所以特别适合那些不是很懂 Ruby 代码而可以理解英语的人使用。

下面我们要编写一些 Cucumber 功能，实现代码 8.5 和代码 8.6 中针对登录功能的部分测试用例。首先，我们在 `features` 文件夹中新建名为 `signing_in.feature` 的文件。

Cucumber 的功能由一个简短的描述文本开始，如下所示：

```
Feature: Signing in
```

然后再添加一定数量相对独立的场景（scenario）。例如，要测试登录失败的情况，我们可以按照如下的方式编写场景：

```
  Scenario: Unsuccessful signin
    Given a user visits the signin page
    When he submits invalid signin information
    Then he should see an error message
```

类似的，测试登录成功时，我们可以加入如下的场景：

```
  Scenario: Successful signin
    Given a user visits the signin page
    And the user has an account
    When the user submits valid signin information
    Then he should see his profile page
    And he should see a signout link
```

把上述的文本放在一起，就组成了代码 8.32 所示的 Cucumber 功能文件。

代码 8.31：测试用户登录功能
features/signing_in.feature

```
Feature: Signing in

  Scenario: Unsuccessful signin
    Given a user visits the signin page
    When he submits invalid signin information
    Then he should see an error message

  Scenario: Successful signin
    Given a user visits the signin page
    And the user has an account
    When the user submits valid signin information
    Then he should see his profile page
    And he should see a signout link
```

然后使用 `cucumber` 命令运行这个功能：

```
$ bundle exec cucumber features/
```

上述命令和执行 RSpec 测试的命令类似：

```
$ bundle exec rspec spec/
```

提示一下，Cucumber 和 RSpec 一样，可以通过 Rake 命令执行：

```
$ bundle exec rake cucumber
```

（鉴于某些原因，我经常使用的命令是 `rake cucumber:ok`。）

我们只是写了一些纯文本，所以毫不意外，Cucumber 场景现在不会通过。若要让测试通过，我们要新建一个步骤定义文件，把场景中的纯文本和 Ruby 代码对应起来。步骤定义文件存放在 `features/step_definition` 文件夹中，我们要将其命名为 `authentication_steps.rb`。

以 `Feature` 和 `Scenario` 开头的行基本上只被视作文档，其他的行则都要和 Ruby 代码对应。例如，功能文件中下面这行

```
Given a user visits the signin page
```

对应到步骤定义中的

```
Given /^a user visits the signin page$/ do
  visit signin_path
end
```

在功能文件中，`Given` 只是普通的字符串，而在步骤定义中 `Given` 则是一个方法，可以接受一个正则表达式作为参数，后面还可以跟着一个块。`Given` 方法的正则表达式参数是用来匹配功能文件中某个特定行的，块中的代码则是实现描述的行为所需的 Ruby 代码。本例中的“a user visits the signin page”是由下面这行代码实现的：

```
visit signin_path
```

你可能觉得这行代码很眼熟，不错，这就是前面用过的 Capybara 提供的方法，Cucumber 的步骤定义文件会自动引入 Capybara。接下来的两行代码实现也同样眼熟。如下的场景步骤：

```
When he submits invalid signin information
Then he should see an error message
```

对应到步骤定义文件中的

```
When /^he submits invalid signin information$/ do
  click_button "Sign in"
end

Then /^he should see an error message$/ do
  expect(page).to have_selector('div.alert.alert-error')
end
```

上面这段代码的第一步还是用了 Capybara，第二步则结合了 Capybara 的 `page` 和 RSpec。很明显，之前我们使用 RSpec 和 Capybara 编写的测试，在 Cucumber 中也是有用武之地的。

场景中接下来的步骤也可以做类似的处理。最终的步骤定义文件如代码 8.33 所示。你可以一次只添加一个步骤，然后执行下面的代码，直到测试都通过为止：

```
$ bundle exec cucumber features/
```

代码 8.32： 使登录功能通过的步骤定义
features/step_definitions/authentication_steps.rb

```
Given /^a user visits the signin page$/ do
  visit signin_path
end

When /^he submits invalid signin information$/ do
  click_button "Sign in"
end

Then /^he should see an error message$/ do
  expect(page).to have_selector('div.alert.alert-error')
end

Given /^the user has an account$/ do
  @user = User.create(name: "Example User", email: "user@example.com",
                      password: "foobar", password_confirmation: "foobar")
end

When /^the user submits valid signin information$/ do
  fill_in "Email",    with: @user.email
  fill_in "Password", with: @user.password
  click_button "Sign in"
end

Then /^he should see his profile page$/ do
  expect(page).to have_title(@user.name)
end

Then /^he should see a signout link$/ do
  expect(page).to have_link('Sign out', href: signout_path)
end
```

添加了代码 8.33，Cucumber 测试应该就可以通过了：

```
$ bundle exec cucumber features/
```

8.3.3 小技巧：自定义 RSpec 匹配器

编写了一些简单的 Cucumber 场景之后，我们来和相应的 RSpec 测试用例对比一下。先看一下代码 8.32 中的 Cucumber 功能和代码 8.33 中的步骤定义，然后再看一下如下的 RSpec 集成测试：

```
describe "Authentication" do

  subject { page }

  describe "signin" do
    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_title('Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }
    end

    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        fill_in "Email",    with: user.email.upcase
        fill_in "Password", with: user.password
        click_button "Sign in"
      end

      it { should have_title(user.name) }
      it { should have_link('Profile',     href: user_path(user)) }
      it { should have_link('Sign out',    href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
    end
  end
end
```

由此你大概就可以看出 Cucumber 和集成测试各自的优缺点了。Cucumber 功能可读性很好，但是却和测试代码分隔开了，同时削弱了功能和测试代码的作用。我觉得 Cucumber 测试读起来很顺口，但是写起来怪怪的；而集成测试读起来不太顺口，但是很容易编写。

Cucumber 把功能描述和步骤定义分开，可以很好的实现抽象层面的行为。例如，下面这个描述

```
Then he should see an error message
```

表达的意思是，期望看到一个错误提示信息。如下的步骤定义则检测了能否实现这个期望：

```
Then /^he should see an error message$/ do
  expect(page).to have_selector('div.alert.alert-error', text: 'Invalid')
end
```

Cucumber 这种分离方式特别便捷的地方在于，只有步骤定义是依赖具体实现的，所以假如我们修改了错误提示信息所用的 CSS class，功能描述文件是不需要修改的。

那么，如果你只是想检测页面中是否显示有错误提示信息，就不想在多个地方重复的编写下面的测试：

```
should have_selector('div.alert.alert-error', text: 'Invalid')
```

如果你真的这么做了，就把测试和具体的实现绑死了，一旦改变了实现方式，就要到处修改测试。在 RSpec 中，可以自定义匹配器来解决这个问题，我们可以直接这么写：

```
should have_error_message('Invalid')
```

我们可以在 5.3.4 节 中定义 `full_title` 测试帮助方法的文件中定义这个匹配器，代码如下：

```
RSpec::Matchers.define :have_error_message do |message|
  match do |page|
    expect(page).to have_selector('div.alert.alert-error', text: message)
  end
end
```

我们还可以为一些常用的操作定义帮助方法，例如：

```
def valid_signin(user)
  fill_in "Email",    with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
end
```

最终的文件如代码 8.34 所示（把 5.6 节中的代码 5.38 和代码 5.39 合并了）。我觉得这种方法比 Cucumber 的步骤定义还要灵活，特别是当匹配器和帮助方法可以接受一个参数时，例如 `valid_signin(user)`。我们也可以用步骤定义中的正则表达式匹配来实现这种功能，不过太过繁杂。

代码 8.33： 添加一个帮助函数和一个 RSpec 自定义匹配器
`spec/support/utilities.rb`

```
include ApplicationHelper

def valid_signin(user)
  fill_in "Email",    with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
end
```

```
RSpec::Matchers.define :have_error_message do |message|
  match do |page|
    expect(page).to have_selector('div.alert.alert-error', text: message)
  end
end
```

添加了代码 8.34 之后，我们就可以直接写

```
it { should have_error_message('Invalid') }
```

和

```
describe "with valid information" do
  let(:user) { FactoryGirl.create(:user) }
  before { valid_signin(user) }
  .
  .
  .
```

还有很多测试用例把测试和具体的实现绑缚在一起了，我们会在 8.5 节的练习中彻底的搜查现有的测试组件，使用自定义匹配器和帮助方法解耦测试和具体实现。

8.4 小结

本章我们介绍了很多基础知识，也为稍显简陋的应用程序实现了注册和登录功能。实现了用户身份验证功能后，我们就可以根据登录状态和用户的身份限制对特定页面的访问权限。在实现限制访问的过程中，我们会为用户添加编辑个人信息的功能，还会为管理员添加删除用户的功能。这些是第 9 章的主要内容。

在继续阅读之前，先把本章的改动合并到主分支吧：

```
$ git add .
$ git commit -m "Finish sign in"
$ git checkout master
$ git merge sign-in-out
```

然后再推送到 GitHub 和 Heroku “生产环境”服务器：

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

8.5 练习

1. 重构登录表单，把 `form_for` 换成 `form_tag`，确保测试还是可以通过的。提示：可以参照 Railscasts 第 270 集《Authentication in Rails 3.1》，特别留意一下 `params` Hash 结构的变化。

2. 参照 8.3.3 节中的示例，遍览用户和身份验证相关的集成测试，在 `spec/support/utilities.rb` 中定义帮助函数，解耦测试和具体实现。附加题：把这些帮助方法放到不同的文件和模块中，然后再引入相应的模块。

此页留白

第 9 章 更新、显示和删除用户

本章我们要完成表格 7.1所示的Users 资源，添加 `edit`、`update`、`index` 和 `destroy` 动作。首先我们要实现更新用户个人资料的功能，实现这样的功能自然要依靠安全验证系统（基于第 8 章中实现的权限限制）。然后要创建一个页面列出所有的用户（也需要权限限制），期间会介绍示例数据和分页功能。最后，我们还要实现删除用户的功能，从数据库中删除用户记录。我们不会为所有用户都提供这种强大的权限，而是会创建管理员，授权他们来删除用户。

在开始之前，我们要新建 `updating-users` 分支：

```
$ git checkout -b updating-users
```

9.1 更新用户

编辑用户信息的方法和创建新用户差不多（参见第 7 章），创建新用户的页面是在 `new` 动作中处理的，而编辑用户的页面则是在 `edit` 动作中；创建用户的过程是在 `create` 动作中处理了 `POST` 请求，而编辑用户要在 `update` 动作中处理 `PATCH` 请求（HTTP 请求参见旁注 3.3）。二者之间最大的区别是，任何人都可以注册，但只有当前用户才能更新他自己的信息。所以我们就要限制访问，只有授权的用户才能编辑更新资料，我们可以利用第 8 章实现的身份验证机制，使用”事前过滤器（before filter）“实现访问限制。

9.1.1 编辑表单

我们先来创建编辑表单，其构思图如图 9.1 所示。[1]和之前一样，我们要先编写测试。注意构思图中修改 Gravatar 头像的链接，如果你浏览过 Gravatar 的网站，可能就知道上传和编辑头像的地址是 http://gravatar.com/emails，我们就来测试编辑页面中有没有一个链接指向了这个地址。[2]

对编辑用户表单的测试和第七章练习中的代码 7.31 类似，同样也测试了提交不合法数据后是否会显示错误提示信息，如代码 9.1 所示。

代码 9.1：用户编辑页面的测试
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
```

1. 图片来自 http://www.flickr.com/photos/sashawolff/4598355045/
2. Gravatar 会把这个地址转向 http://en.gravatar.com/emails，我去掉了前面的 en，这样选择其他语言的用户就会自动转向相应的页面了。

```
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit edit_user_path(user) }

    describe "page" do
      it { should have_content("Update your profile") }
      it { should have_title("Edit user") }
      it { should have_link('change', href: 'http://gravatar.com/emails') }
    end

    describe "with invalid information" do
      before { click_button "Save changes" }

      it { should have_content('error') }
    end
  end
end
```

图 9.1：编辑用户页面的构思图

程序所需的代码要放在 `edit` 动作中，我们在表格 7.1中列出了，用户编辑页面的地址是 /users/1/edit（假设用户的 id 是 1）。我们介绍过用户的 id 是保存在 `params[:id]` 中的，所以我们可以按照代码 9.2 所示的方法查找用户。

代码 9.2：Users 控制器的 `edit` 方法
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def edit
    @user = User.find(params[:id])
  end
  .
  .
  .
end
```

要让测试通过，我们就要编写编辑用户页面的视图，如代码 9.3 所示。仔细观察一下视图代码，它和代码 7.17 中创建新用户页面的视图代码很相似，这就暗示我们要进行重构，把重复的代码移入局部视图。重构会留作练习，详情参见 9.6 节。

代码 9.3：编辑用户页面的视图
`app/views/users/edit.html.erb`

```
<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.label :password_confirmation, "Confirm Password" %>
      <%= f.password_field :password_confirmation %>

      <%= f.submit "Save changes", class: "btn btn-large btn-primary" %>
```

```
    <% end %>

    <%= gravatar_for @user %>
    <a href="http://gravatar.com/emails">change</a>
  </div>
</div>
```

在这段代码中我们再次使用了 7.3.3 节中创建的 `error_messages` 局部视图。

添加了视图代码，再加上代码 9.2 中定义的 `@user` 变量，代码 9.1 中的“编辑页面”测试应该就可以通过了：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "edit page"
```

编辑用户页面如图 9.2 所示，我们看到 Rails 会自动读取 `@user` 变量，预先填好了名字和 Email 地址字段。

图 9.2：编辑用户页面，名字和 Email 地址字段已经自动填好了

查看一下编辑用户页面的源码，我们可以发现的确生成了一个 `form` 元素，参见代码 9.4。

代码 9.4：编辑表单的 HTML

```
<form action="/users/1" class="edit_user" id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="patch" />
  .
```

```
  .
  .
</form>
```

留意一下其中的一个隐藏字段：

```
<input name="_method" type="hidden" value="patch" />
```

因为浏览器本身并不支持发送 PATCH 请求（表格 7.1中列出的 REST 动作要用），所以 Rails 就在 POST 请求中使用这个隐藏字段伪造了一个 PATCH 请求。[3]

还有一个细节需要注意一下，代码 9.3 和代码 7.17 都使用了相同的 form_for(@user) 来构建表单，那么 Rails 是怎么知道创建新用户要发送 POST 请求，而编辑用户时要发送 PATCH 请求的呢？这个问题的答案是，通过 Active Record 提供的 new_record? 方法可以检测用户是新创建的还是已经存在于数据库中的：

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

所以在使用 form_for(@user) 构建表单时，如果 @user.new_record? 返回 true 则发送 POST 请求，否则就发送 PATCH 请求。

最后，我们还要在导航中添加一个指向编辑用户页面的链接（“设置（Settings）”）。因为只有登录之后才会显示这个页面，所以对“设置”链接的测试要和其他的身份验证测试放在一起，如代码 9.5 所示。（如果能再测试一下没登录时不会显示“设置”链接就更完美了，这会留作练习，参见 9.6 节。）

代码 9.5：添加检测“设置”链接的测试
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
    .
    .
    .
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before { sign_in user }

      it { should have_title(user.name) }
      it { should have_link('Profile',     href: user_path(user)) }
      it { should have_link('Settings',    href: edit_user_path(user)) }
      it { should have_link('Sign out',    href: signout_path) }
```

3. 不要担心实现的细节。具体的实现方式是 Rails 框架的开发者需要关注的，作为 Rails 程序开发者则无需关心。

```
      it { should_not have_link('Sign in', href: signin_path) }
      .
      .
      .
    end
  end
end
```

为了简化，代码 9.5 中使用 `sign_in` 帮助方法，这个方法的作用是访问登录页面，提交合法的表单数据，如代码 9.6 所示。

代码 9.6： 用户登录帮助方法
`spec/support/utilities.rb`

```
.
.
.
def sign_in(user, options={})
  if options[:no_capybara]
    # Sign in when not using Capybara.
    remember_token = User.new_remember_token
    cookies[:remember_token] = remember_token
    user.update_attribute(:remember_token, User.encrypt(remember_token))
  else
    visit signin_path
    fill_in "Email",    with: user.email
    fill_in "Password", with: user.password
    click_button "Sign in"
  end
end
```

如上述代码中的注释所说，如果没有使用 Capybara 的话，填写表单的操作是无效的，所以我们提供了 `no_capybara: true` 选项，跳过默认的登录操作，直接处理 cookie。如果直接使用 HTTP 请求方法就必须要有上面这行代码，具体的用法在代码 9.46 中有介绍。（注意，测试中使用的 `cookies` 对象和真实的 cookies 对象是有点不一样的，代码 8.19 中使用的 `cookies.permanent` 方法不能在测试中使用。）你可能已经猜到了，`sing_in` 在后续的测试中还会用到，而且还可以用来去除重复代码（参见 9.6 节）。

在程序中添加“设置”链接很简单，我们就直接使用表格 7.1 中列出的 `edit_user_path` 具名路由，其参数设为代码 8.22 中定义的 `current_user` 帮助方法：

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

完整的代码如代码 9.7 所示。

代码 9.7： 添加“设置”链接
`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

9.1.2 编辑失败

本小节我们要处理编辑失败的情况，让代码 9.1 中对错误提示信息的测试通过。我们要在 Users 控制器的 `update` 动作中使用 `update_attributes` 方法，传入提交的 `params` Hash，更新用户记录，如代码 9.8 所示。如果提交了不合法的数据，更新操作会返回 `false`，交由 `else` 分支处理，重新渲染编辑用户页面。我们之前用过类似的处理方式，代码结构和第一个版本的 `create` 动作类似（参见代码 7.21）。

代码 9.8：还不完整的 `update` 动作
`app/controllers/users_controller.rb`

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

```
  .
  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      # Handle a successful update.
    else
      render 'edit'
    end
  end
  .
  .
  .
end
```

注意，在调用 `update_attributes` 方法时指定的 `user_params` 参数，这种用法是“健壮参数”（strong parameter），可以避免 mass assignment 漏洞（参见 7.3.2 节）。

图 9.3： 提交编辑表单后显示的错误提示信息

提交不合法信息后显示了错误提示信息（如图 9.3），测试就可以通过了，你可以运行测试组件验证一下：

```
$ bundle exec rspec spec/
```

9.1.3 编辑成功

现在我们要让编辑表单能够正常使用了。编辑头像的功能已经实现了，因为我们把上传头像的操作交由 Gravatar 处理了，如需更换头像，点击图 9.2 中的“change”链接就可以了，如图 9.4 所示。下面我们来实现编辑其他信息的功能。

图 9.4：Gravatar 的剪切图片界面，上传了一个帅哥的图片

对 `update` 动作的测试和对 `create` 的测试类似。代码 9.9 介绍了如何使用 Capybara 在表单中填写合法的数据，还介绍了怎么测试提交表单的操作是否正确。测试的代码很多，你可以参考第 7 章中的测试，试一下能不能完全理解。

代码 9.9：测试 Users 控制器的 `update` 动作
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
```

```
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before do
      sign_in user
      visit edit_user_path(user)
    end
    .
    .
    .
    describe "with valid information" do
      let(:new_name)  { "New Name" }
      let(:new_email) { "new@example.com" }
      before do
        fill_in "Name",             with: new_name
        fill_in "Email",            with: new_email
        fill_in "Password",         with: user.password
        fill_in "Confirm Password", with: user.password
        click_button "Save changes"
      end

      it { should have_title(new_name) }
      it { should have_selector('div.alert.alert-success') }
      it { should have_link('Sign out', href: signout_path) }
      specify { expect(user.reload.name).to  eq new_name }
      specify { expect(user.reload.email).to eq new_email }
    end
  end
end
```

上述代码中出现了一个新的方法 `reload`，出现在检测用户的属性是否已经更新的测试中：

```
specify { expect(user.reload.name).to  == new_name }
specify { expect(user.reload.email).to eq new_email }
```

这两行代码使用 `user.reload` 从测试数据库中重新加载 `user` 的数据，然后检测用户的名字和 Email 地址是否更新成了新的值。

要让代码 9.9 中的测试通过，我们可以参照最终版本的 `create` 动作（代码 8.27）来编写 `update` 动作，如代码 9.10 所示。我们在代码 9.8 的基础上加入了下面这三行。

```
flash[:success] = "Profile updated"
sign_in @user
redirect_to @user
```

代码 9.10： Users 控制器的 update 动作
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
end
```

注意，现在这种实现方式，每次更新数据都要提供密码（填写图 9.2 中那两个空的字段），虽然有点烦人，不过却保证了安全。

添加了本小节的代码之后，编辑用户页面应该可以正常使用了，你可以运行测试组件再确认一下，测试应该是可以通过的：

```
$ bundle exec rspec spec/
```

9.2 权限限制

第 8 章中实现的身份验证机制有一个很好的作用，可以实现权限限制。身份验证可以识别用户是否已经注册，而权限限制则可以限制用户可以进行的操作。

虽然 9.1 节中已经基本完成了 edit 和 update 动作，但是却有一个安全隐患：任何人（甚至是未登录的用户）都可以访问这两个动作，而且登录后的用户可以更新所有其他用户的资料。本节我们要实现一种安全机制，限制用户必须先登录才能更新自己的资料，而不能更新他人的资料。没有登录的用户如果试图访问这些受保护的页面，会转向登录页面，并显示一个提示信息，构思图如图 9.5 所示。

9.2.1 必须先登录

因为对 edit 和 update 动作所做的安全限制是一样的，所以我们就在同一个 RSpec describe 块中进行测试。我们从要求登录开始，测试代码要检测未登录的用户试图访问这两个动作时是否转向了登录页面，如代码 9.11 所示。

代码 9.11： 测试 edit 和 update 动作是否处于被保护状态
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "in the Users controller" do

        describe "visiting the edit page" do
          before { visit edit_user_path(user) }
          it { should have_title('Sign in') }
        end

        describe "submitting to the update action" do
          before { patch user_path(user) }
          specify { expect(response).to redirect_to(signin_path) }
        end
      end
    end
  end
end
```

代码 9.11 除了使用 Capybara 的 `visit` 方法之外，还第一次使用了另一种访问控制器动作的方法：如果需要直接发起某种 HTTP 请求，则直接使用 HTTP 动词对应的方法即可，例如本例中的 `patch` 发起的就是 `PATCH` 请求：

```
describe "submitting to the update action" do
  before { patch user_path(user) }
  specify { expect(response).to redirect_to(signin_path) }
end
```

上述代码会向 /users/1 地址发送 `PATCH` 请求，由 Users 控制器的 `update` 动作处理（参见表格 7.1）。我们必须这么做，因为浏览器无法直接访问 `update` 动作，必须先提交编辑表单，所以 Capybara 也做不到。访问编辑资料页面只能测试 `edit` 动作是否有权限继续操作，而不能测试 `update` 动作的授权情况。所以，如果要测试 `update` 动作是否有权限进行操作只能直接发送 `PATCH` 请求。（你可能已经猜到了，除了 `patch` 方法之外，Rails 中的测试还支持 `get`、`post` 和 `delete` 方法。）

直接发送某种 HTTP 请求时，我们需要处理更底层的 `response` 对象。和 Capybara 提供的 `page` 对象不同，我们可以使用 `response` 测试服务器的响应。本例我们检测了 `update` 动作的响应是否转向了登录页面：

```
specify { expect(response).to redirect_to(signin_path) }
```

图 9.5：访问受保护页面转向后的页面构思图

我们要使用 `before_action` 方法实现权限限制，这个方法会在指定的动作执行之前，先运行指定的方法。为了实现要求用户先登录的限制，我们要定义一个名为 `signed_in_user` 的方法，然后调用 `before_action :signed_in_user`，如代码 9.12 所示。

代码 9.12：添加 `signed_in_user` 事前过滤器
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:edit, :update]
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
```

```
                                    :password_confirmation)
    end

    # Before filters

    def signed_in_user
      redirect_to signin_url, notice: "Please sign in." unless signed_in?
    end
end
```

默认情况下，事前过滤器会应用于控制器中的所有动作，所以在上述代码中我们传入了 `:only` 参数指定只应用在 `edit` 和 `update` 动作上。

图 9.6：尝试访问受保护的页面后显示的登录表单

注意，在代码 9.12 中我们使用了设定 `flash[:notice]` 的简便方式，把 `redirect_to` 方法的第二个参数指定为一个 Hash。这段代码等同于：

```
unless signed_in?
  flash[:notice] = "Please sign in."
  redirect_to signin_url
end
```

（flash[:error] 也可以使用上述的简便方式，但 flash[:success] 却不可以。）

flash[:notice] 加上 flash[:success] 和 flash[:error] 就是我们要介绍的三种 Flash 消息，Bootstrap 为这三种消息都提供了样式。退出后再尝试访问 /users/1/edit，就会看到如图 9.6 所示的黄色提示框。

现在所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

9.2.2 用户只能编辑自己的资料

当然，要求用户必须先登录还是不够的，用户必须只能编辑自己的资料。我们的测试可以这么编写，用其他用户的身份登录，然后访问 edit 和 update 动作，如代码 9.13 所示。注意，这段测试我们不用 Capybara（no_capybara: true）,而使用 get 和 patch 方法直接访问 edit 和 update 动作。而且，用户不应该尝试编辑其他用户的资料，我们没有转向登录页面，而是转到了网站的首页。

代码 9.13：测试只有自己才能访问 edit 和 update 动作
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as wrong user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:wrong_user) { FactoryGirl.create(:user, email: "wrong@example.com") }
      before { sign_in user, no_capybara: true }

      describe "submitting a GET request to the Users#edit action" do
        before { get edit_user_path(wrong_user) }
        specify { expect(response.body).not_to match(full_title('Edit user')) }
        specify { expect(response).to redirect_to(root_url) }
      end

      describe "submitting a PATCH request to the Users#update action" do
        before { patch user_path(wrong_user) }
        specify { expect(response).to redirect_to(root_url) }
      end
    end
```

```
  end
end
```

注意，创建预构件的方法还可以接受第二个参数：

```
FactoryGirl.create(:user, email: "wrong@example.com")
```

上述代码会用指定的 Email 替换默认值，然后创建用户。我们的测试要确保其他的用户不能访问原来那个用户的 `edit` 和 `update` 动作。

我们在控制器中加入了第二个事前过滤器，调用 `correct_user` 方法，如代码 9.14 所示。

代码 9.14： 保护 `edit` 和 `update` 动作的 `correct_user` 事前过滤器
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update_attributes(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Before filters

    def signed_in_user
      redirect_to signin_url, notice: "Please sign in." unless signed_in?
```

```
    end

    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_path) unless current_user?(@user)
    end
end
```

上述代码中的 correct_user 方法使用了 current_user? 方法，我们要在 Sessions 帮助方法模块中定义一下，如代码 9.15。

代码 9.15：定义 current_user? 方法
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  def current_user
    remember_token = User.encrypt(cookies[:remember_token])
    @current_user ||= User.find_by(remember_token: remember_token)
  end

  def current_user?(user)
    user == current_user
  end
  .
  .
  .
end
```

代码 9.14 同时也更新了 edit 和 update 动作的代码。之前在代码 9.2 中，我们是这样写的：

```
def edit
  @user = User.find(params[:id])
end
```

update 代码类似。既然 correct_user 事前过滤器中已经定义了 @user，这两个动作中就不再需要再定义 @user 变量了。

在继续阅读之前，你应该验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

9.2.3 更友好的转向

程序的权限限制基本完成了，但是还有一点小小的不足：不管用户尝试访问的是哪个受保护的页面，登录后都会转向资料页面。也就是说，如果未登录的用户访问了编辑资料页面，会要求先登录，登录转到的页面是 /users/1，而不是 `/users/1/edit`。如果登录后能转到用户之前想访问的页面就更好了。

针对这种更友好的转向，我们可以这样编写测试，先访问编辑用户资料页面，转向登录页面后，填写正确的登录信息，点击“Sign in”按钮，然后显示的应该是编辑用户资料页面，而不是用户资料页面。相应的测试如代码 9.16 所示。

代码 9.16：测试更友好的转向
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email",    with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do

          it "should render the desired protected page" do
            expect(page).to have_title('Edit user')
          end
        end
      end
      .
      .
      .
    end
    .
    .
    .
```

```
  end
end
```

下面我们来实现这个设想。[4]要转向用户真正想访问的页面，我们要在某个地方存储这个页面的地址，登录后再转向这个页面。我们要通过两个方法来实现这个过程，store_location 和 redirect_back_or，都在 Sessions 帮助方法模块中定义，如代码 9.17。

代码 9.17：实现更友好的转向所需的代码
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  def redirect_back_or(default)
    redirect_to(session[:return_to] || default)
    session.delete(:return_to)
  end

  def store_location
    session[:return_to] = request.fullpath if request.get?
  end
end
```

地址的存储使用了 Rails 提供的 session，session 可以理解成和 8.2.1 节中介绍的 cookies 是类似的东西，会在浏览器关闭后自动失效。（在 8.5 节中介绍过，其实 session 的实现方法正是如此。）我们还使用了 request 对象的 fullpath 方法获取了所请求页面的完整地址。在 store_location 方法中，把完整的请求地址存储在 session[:return_to] 中。但这个方法只能在 GET 请求中使用（if request.get?）。这么做，当未登录的用户提交表单时，不会存储转向地址（这种情况虽然很罕见，但在提交表单前，如果用户手动删除了记忆权标，还是会发生的），那么本来期望接收 POST、PATCH 或 DELETE 请求的动作实际收到的却是 GET 请求，就会产生异常。[5]

要使用 store_location，我们要把它加入 signed_in_user 事前过滤器中，如代码 9.18 所示。

代码 9.18：把 store_location 加入 signed_in_user 事前过滤器
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end
```

4. 实现的代码来自 thoughtbot 的 Clearance gem。
5. 感谢读者 Yoel Adler 提出了这个问题，并和我讨论了解决办法。

```
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Before filters

    def signed_in_user
      unless signed_in?
        store_location
        redirect_to signin_url, notice: "Please sign in."
      end
    end

    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_path) unless current_user?(@user)
    end
end
```

实现转向操作，要在 Sessions 控制器的 `create` 动作中加入 `redirect_back_or` 方法，用户登录后转到适当的页面，如代码 9.19 所示。如果存储了之前请求的地址，`redirect_back_or` 方法就会转向这个地址，否则会转向参数中指定的地址。

代码 9.19： 加入友好转向后的 `create` 动作
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      sign_in user
      redirect_back_or user
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end
```

```
  .
  .
  .
end
```

（如果你做了第 8 章的练习 1，记得把代码 9.19 中的 `params` Hash 换掉。）

`redirect_back_or` 方法在下面这行代码中使用了“或”操作符 `||`：

```
session[:return_to] || default
```

如果 `session[:return_to]` 的值不是 `nil`，上面这行代码就会返回 `session[:return_to]` 的值，否则会返回 `default`。注意，在代码 9.17 中，成功转向后就会删除存储在 session 中的转向地址。如果不删除的话，在关闭浏览器之前，每次登录后都会转到存储的地址上。（对这一过程的测试留作练习，参见 9.6 节。）

加入上述代码之后，代码 9.16 中对友好转向的集成测试应该可以通过了。至此，我们也就完成了基本的用户身份验证和页面保护机制。和之前一样，在继续阅读之前，最好确认一下所有的测试是否都可以通过：

```
$ bundle exec rspec spec/
```

图 **9.7**：用户列表页面的构思图，包含了分页链接和“Users”导航链接

9.3 列出所有用户

本节我们要添加计划中的倒数第二个用户动作，`index`。`index` 动作不会显示某一个用户，而是显示所有的用户。在这个过程中，我们要学习如何在数据库中生成示例用户数据，以及如何分页显示用户列表，显示任意数量的用户。用户列表、分页链接和“所有用户（Users）”导航链接的构思图如图 9.7 所示。[6]在 9.4 节 中，我们还会在用户列表中添加删除链接，这样就可以删除有问题的用户了。

9.3.1 用户列表

单个用户的资料页面是对外开放的，不过用户列表页面只有注册用户才能访问。我们先来编写测试。在测试中我们要检测 `index` 动作是被保护的，如果访问 `users_path` 会转向登录页面。和其他的权限限制测试一样，我们也会把这个测试放在身份验证的集成测试中，如代码 9.20 所示。

代码 9.20：测试 `index` 动作是否是被保护的
`spec/requests/authentication_pages_spec.rb`

6. 婴儿的图片来自 http://www.flickr.com/photos/glasgows/338937124/

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      .
      .
      .
      describe "in the Users controller" do
        .
        .
        .
        describe "visiting the user index" do
          before { visit users_path }
          it { should have_title('Sign in') }
        end
      end
      .
      .
      .
    end
  end
end
```

若要这个测试通过，我们要把 index 动作加入 signed_in_user 事前过滤器，如代码 9.21 所示。

代码 9.21： 访问 index 动作必须先登录
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:index, :edit, :update]
  before_action :correct_user,   only: [:edit, :update]

  def index
  end

  def show
    @user = User.find(params[:id])
  end
  .
  .
```

```
  .
end
```

接下来，我们要测试用户登录后，用户列表页面要有特定的标题和内容，还要列出网站中所有的用户。为此，我们要创建三个用户预构件，以第一个用户的身份登录，然后检测用户列表页面中是否有一个列表，各用户的名字都包含在一个单独的 `li` 标签中。注意，我们要为每个用户分配不同的名字，这样列表中的用户才是不一样的，如代码 9.22 所示。

代码 9.22： 用户列表页面的测试
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do
    before do
      sign_in FactoryGirl.create(:user)
      FactoryGirl.create(:user, name: "Bob", email: "bob@example.com")
      FactoryGirl.create(:user, name: "Ben", email: "ben@example.com")
      visit users_path
    end

    it { should have_title('All users') }
    it { should have_content('All users') }

    it "should list each user" do
      User.all.each do |user|
        expect(page).to have_selector('li', text: user.name)
      end
    end
  end
  .
  .
  .
end
```

你可能还记得，我们在演示程序的相关代码中介绍过（参见代码 2.4），在程序中我们可以使用 `User.all` 从数据库中取回所有的用户，赋值给实例变量 `@users` 在视图中使用，如代码 9.23 所示。（你可能会觉得一次列出所有的用户不太好，你是对的，我们会在 9.3.3 节中改进。）

代码 9.23： Users 控制器的 `index` 动作
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

要显示用户列表页面，我们要创建一个视图，遍历所有的用户，把单个用户包含在 `li` 标签中。我们要使用 `each` 方法遍历所有用户，显示用户的 Gravatar 头像和名字，然后把所有的用户包含在无序列表 `ul` 标签中，如代码 9.24 所示。在代码 9.24 中，我们用到了 7.6 节练习中代码 7.30 的成果，允许向 Gravatar 帮助方法传入第二个参数，指定头像的大小。如果你之前没有做这个练习题，在继续阅读之前请参照代码 7.30 更新 Users 控制器的帮助方法文件。

代码 9.24： 用户列表页面的视图
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>
```

我们再添加一些 CSS（更确切的说是 SCSS）美化一下，如代码 9.25。

代码 9.25： 用户列表页面的 CSS
`app/assets/stylesheets/custom.css.scss`

```
.
.
.
/* Users index */

.users {
  list-style: none;
  margin: 0;
  li {
```

```
    overflow: auto;
    padding: 10px 0;
    border-top: 1px solid $grayLighter;
    &:last-child {
      border-bottom: 1px solid $grayLighter;
    }
  }
}
```

最后，我们还要在头部的导航中加入到用户列表页面的链接，链接的地址为 users_path，这是表格 7.1中还没介绍的最后一个具名路由了。相应的测试（代码 9.26）和程序所需的代码（代码 9.27）都很简单。

代码 9.26：检测“Users”链接的测试
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
    .
    .
    .
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before { sign_in user }

      it { should have_title(user.name) }
      it { should have_link('Users',       href: users_path) }
      it { should have_link('Profile',     href: user_path(user)) }
      it { should have_link('Settings',    href: edit_user_path(user)) }
      it { should have_link('Sign out',    href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
      .
      .
      .
    end
  end
end
```

代码 9.27：添加“Users”链接
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
```

```
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", users_path %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

至此，用户列表页面的功能就实现了，所有的测试也都可以通过了：

```
$ bundle exec rspec spec/
```

不过，如图 9.8 所示，页面中只显示了一个用户，有点孤单单。下面，让我们来改变一下这种悲惨状况。

9.3.2 示例用户

在本小节中，我们要为应用程序添加更多的用户。如果要让用户列表看上去像个列表，我们可以在浏览器中访问注册页面，然后一个一个地注册用户，不过还有更好的方法，让 Ruby 和 Rake 为我们创建用户。

首先，我们要在 `Gemfile` 中加入 `faker`（如代码 9.28 所示），使用这个 gem，我们可以使用半真实的名字和 Email 地址创建示例用户。

代码 9.28： 把 faker 加入 `Gemfile`

```
source 'https://rubygems.org'
ruby '2.0.0'
#ruby-gemset=railstutorial_rails_4_0
```

```
gem 'rails', '4.0.2'
gem 'bootstrap-sass', '2.3.2.0'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.1.2'
.
.
.
```

图 9.8：用户列表页面，只有一个用户

然后和之前一样，运行下面的命令安装：

```
$ bundle install
```

接下来我们要添加一个 Rake 任务创建示例用户。这个 Rake 任务保存在 `lib/tasks` 文件夹中，而且在 `:db` 命名空间中定义，如代码 9.29 所示。（代码中涉及到一些高级知识，现在不必深入了解。）

代码 9.29：在数据库中生成示例用户的 Rake 任务
`lib/tasks/sample_data.rake`

```
namespace :db do
  desc "Fill database with sample data"
```

```
  task populate: :environment do
    User.create!(name: "Example User",
                 email: "example@railstutorial.org",
                 password: "foobar",
                 password_confirmation: "foobar")
    99.times do |n|
      name  = Faker::Name.name
      email = "example-#{n+1}@railstutorial.org"
      password  = "password"
      User.create!(name: name,
                   email: email,
                   password: password,
                   password_confirmation: password)
    end
  end
end
```

图 **9.9**：显示了 100 个用户的用户列表页面（/users）

上述代码定义了一个名为 `db:populate` 的 Rake 任务，先创建一个用户替代之前存在的那个用户，然后还创建了 99 个用户。下面这行代码

```
task populate: :environment do
```

确保这个 Rake 任务可以获取 Rails 环境的信息，包括 User 模型，所以才能使用 `User.create!` 方法。`create!` 方法和 `create` 方法的作用一样，只不过如果提供的信息不合法不会返回 `false` 而是会抛出异常（参见 6.1.4 节），这样如果出错的话就很容易找到错误发生的地方。

这个任务是定义在 `:db` 命名空间中的，所以我们要按照如下的方式来执行：

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake test:prepare
```

执行这三个任务之后，我们的应用程序就有 100 个用户了，如图 9.9 所示。（我牺牲了一点个人时间为前几个用户上传了头像，这样就不都会显示默认的 Gravatar 头像了。）

9.3.3 分页

现在，当初的用户不再孤单单了，但是又出现了新的问题：用户太多，全在一个页面中显示。现在的用户数量是 100 个，算是少的了，在真实的网站中，这个数量可能是以千计的。为了避免在一页中显示过多的用户，我们可以使用分页功能，一页只显示 30 个用户。

在 Rails 中有很多实现分页的方法，我们要使用其中一个最简单也最完善的，叫做 will_paginate。我们要使用 `will_paginate` 和 `bootstrap-will_paginate` 这两个 gem，`bootstrap-will_paginate` 的作用是设置 will_paginate 使用 Bootstrap 中的分页样式。修改后的 `Gemfile` 如代码 9.30 所示。

代码 9.30：在 `Gemfile` 中加入 will_paginate

```
source 'https://rubygems.org'
ruby '2.0.0'
#ruby-gemset=railstutorial_rails_4_0

gem 'rails', '4.0.2'
gem 'bootstrap-sass', '2.3.2.0'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.1.2'
gem 'will_paginate', '3.0.4'
gem 'bootstrap-will_paginate', '0.0.9'
.
.
.
```

然后执行下面的命令安装：

```
$ bundle install
```

安装后你还要重启 Web 服务器，确保成功加载这两个新 gem。

因为 `will_paginate` 这个 gem 使用的范围很广，所以我们不必做大量的测试，只需简单的测试一下就可以了。首先我们要检测页面中是否包含一个 CSS class 为 `pagination` 的 `div` 元素，这个元素就是由 will_paginate 生成的。然后，我们要检测分页的第一页中是否显示有正确的用户列表。在测试中我们要用到 `paginate` 方法，稍后会做介绍。

和之前一样，我们要使用 Factory Girl 生成用户，但是我们立马就会遇到一个问题：因为用户的 Email 地址必须是唯一的，那么我们就要手动生成 30 个用户，这可是一件很费事的活儿。而且，在测试用户列表时，用户的名字最好也不一样。幸好 Factory Girl 料事如神，提供了 `sequence` 方法来解决这种问题。在代码 7.8 中，我们是直接输入名字和 Email 地址来创建预构件的：

```
FactoryGirl.define do
  factory :user do
    name     "Michael Hartl"
    email    "michael@example.com"
    password "foobar"
    password_confirmation "foobar"
  end
end
```

现在我们要使用 `sequence` 方法自动创建一系列的名字和 Email 地址：

```
factory :user do
  sequence(:name)  { |n| "Person #{n}" }
  sequence(:email) { |n| "person_#{n}@example.com"}
  .
  .
  .
```

`sequence` 方法可以接受一个 Symbol 类型的参数，对应到属性上（例如 `:name`），其后还可以跟着块，有一个块参数，我们将其命名为 `n`。`FactoryGirl.create(:user)` 方法执行成功后，块参数会自动增加 1。因此，创建的第一个用户名字为“Person 1”，Email 地址为“person_1@example.com”；第二个用户的名字为 `Person 2`，Email 地址为“person_2@example.com”；依此类推。完整的代码如代码 9.31 所示。

代码 9.31：定义 Factory Girl 序列
`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    sequence(:name)  { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com"}
    password "foobar"
    password_confirmation "foobar"
  end
end
```

创建了预构件序列后，在测试中就可以生成 30 个用户了，这 30 个用户就可以产生分页了：

```
before(:all) { 30.times { FactoryGirl.create(:user) } }
after(:all)  { User.delete_all }
```

注意，上述代码使用 `before(:all)` 确保在块中所有测试执行之前，一次性创建 30 个示例用户。这是对速度做的优化，因为在某些系统中每个测试都创建 30 个用户会很慢。对应的，我们调用 `after(:all)` 方法，在测试结束后一次性删除所有的用户。

代码 9.32 检测了页面中是否包含正确的 `div` 元素，以及是否显示了正确的用户。注意，我们把代码 9.22 中的 `User.all` 换成了 `User.paginate(page: 1)`，这样我们才能从数据库中取回第一页中要显示的用户。还要注意一下，代码 9.32 中使用的 `before(:each)` 方法是和 `before(:all)` 方法相反的操作。

代码 9.32：测试分页
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do
    let(:user) { FactoryGirl.create(:user) }
    before(:each) do
      sign_in user
      visit users_path
    end

    it { should have_title('All users') }
    it { should have_content('All users') }

    describe "pagination" do

      before(:all) { 30.times { FactoryGirl.create(:user) } }
      after(:all)  { User.delete_all }

      it { should have_selector('div.pagination') }

      it "should list each user" do
        User.paginate(page: 1).each do |user|
          expect(page).to have_selector('li', text: user.name)
        end
      end
    end
  end
  .
  .
```

```
  .
end
```

要实现分页，我们要在用户列表页面的视图中加入一些代码，告诉 Rails 要分页显示用户，而且要把 `index` 动作中的 `User.all` 换成知道如何分页的方法。我们先在视图中加入特殊的 `will_paginate` 方法，如代码 9.33 所示。稍后我们会看到为什么要在用户列表的前后都加入分页代码。

代码 9.33： 在用户列表视图中加入分页
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>

<%= will_paginate %>
```

`will_paginate` 方法有点小神奇，在 Users 控制器的视图中，它会自动寻找名为 `@users` 的对象，然后显示一个分页导航链接。代码 9.33 所示的视图现在还不能正确显示分页，因为现在 `@users` 的值是通过 `User.all` 方法获取的，而 `will_paginate` 的分页需要调用 `paginate` 方法：

```
$ rails console
>> User.paginate(page: 1)
  User Load (1.5ms)  SELECT "users".* FROM "users" LIMIT 30 OFFSET 0
   (1.7ms)  SELECT COUNT(*) FROM "users"
=> #<ActiveRecord::Relation [#<User id: 1,...
```

`paginate` 方法可以接受一个 Hash 类型的参数，键 `:page` 的值指定第几页。`User.paginate` 方法根据 `:page` 的值，一次取回一系列的用户（默认为 30 个）。所以，第一页显示的是第 1-30 个用户，第二页显示的是第 31-60 个，等。如果指定的页数不存在，`paginate` 会显示第一页。

我们可以把 `index` 动作中的 `all` 方法换成 `paginate`，这样页面中就可以显示分页导航了，如代码 9.34 所示。`paginate` 方法所需的 `:page` 参数值由 `params[:page]` 指定，这个 `params` 元素是由 `will_pagenate` 自动生成的。

代码 9.34： 在 `index` 动作中按分页取回用户
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.paginate(page: params[:page])
  end
  .
  .
  .
end
```

现在，用户列表页面应该可以显示分页了，如图 9.10 所示。（在某些系统中，可能需要重启 Rails 服务器。）因为我们在用户列表前后都加入了 `will_paginate` 方法，所以这两个地方都会显示分页链接。

图 9.10： 显示了分页链接的用户列表页面（/users）

如果点击链接“2”，或者“Next”，就会显示第二页，如图 9.11 所示。

图 **9.11**：用户列表的第二页（/users?page=2）

你还应该验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

9.3.4 视图重构

用户列表页面现在已经可以显示分页了，但是有个改进点我不得不介绍一下。Rails 提供了一些很巧妙的方法可以精简视图的结构，本小节我们就要利用这些方法重构一下用户列表页面。因为我们已经做了很好的测试，所以就可以放手去重构，不用担心会破坏网站的功能。

重构的第一步，要把代码 9.33 中的 `li` 换成对 `render` 方法的调用，如代码 9.35 所示。

代码 9.35：重构用户列表视图的第一步
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
```

```
  <% @users.each do |user| %>
    <%= render user %>
  <% end %>
</ul>

<%= will_paginate %>
```

在上述代码中，`render` 的参数不再是指定局部视图的字符串，而是代表 `User` 类的 `user` 变量。[7]Rails 会自定寻找一个名为 `_user.html.erb` 的局部视图，我们要手动创建这个视图，然后写入代码 9.36 中的内容。

代码 9.36： 显示单一用户的局部视图
`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
</li>
```

这个改进很不错，不过我们还可以做的更好。我们可以直接把 `@users` 变量传递给 `render` 方法，如代码 9.37 所示。

代码 9.37： 完全重构后的用户列表视图
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <%= render @users %>
</ul>

<%= will_paginate %>
```

Rails 会把 `@users` 当作一系列的 `User` 对象，遍历这些对象，然后使用 `_user.html.erb` 渲染每个对象。所以我们就得到了代码 9.37 这样简洁的代码。每次重构后，你都应该验证一下测试组件是否还是可以通过的：

```
$ bundle exec rspec spec/
```

7. 我们并不是一定要使用 `user`，遍历时如果用的是 `@users.each do |foobar|`，那么就要用 `render foobar`。这里的关键是要知道对象的类，也就是 `User`。

9.4 删除用户

至此，用户索引也完成了。符合 REST 架构的Users 资源就只剩下最后一个 `destroy` 动作了。本节，我们先添加删除用户的链接（构思图如图 9.12 所示），然后再编写适当的 `destroy` 动作代码完成删除操作。不过，首先我们要先创建管理员级别的用户，并授权这些用户进行删除操作。

图 9.12： 显示有删除链接的用户列表页面构思图

9.4.1 管理员

我们要通过 User 模型中一个名为 `admin` 的属性来判断用户是否具有管理员权限。`admin` 属性的类型为布尔值，Active Record 会自动生成一个 `admin?` 方法，返回布尔值，判断用户是否为管理员。针对 `admin` 属性的测试如代码 9.38 所示。

代码 9.38： 测试 `admin` 属性
`spec/models/user_spec.rb`

```
require 'spec_helper'
```

```
describe User do
  .
  .
  .
  it { should respond_to(:authenticate) }
  it { should respond_to(:admin) }

  it { should be_valid }
  it { should_not be_admin }

  describe "with admin attribute set to 'true'" do
    before do
      @user.save!
      @user.toggle!(:admin)
    end

    it { should be_admin }
  end
  .
  .
  .
end
```

在上述代码中我们使用 `toggle!` 方法把 `admin` 属性的值从 `false` 转变成 `true`。`it { should be_admin }` 这行代码说明用户对象应该可以响应 `admin?` 方法（这是 RSpec 对布尔值属性的一个约定）。

和之前一样，我们要使用迁移添加 `admin` 属性，在命令行中指定其类型为 `boolean`：

```
$ rails generate migration add_admin_to_users admin:boolean
```

这个命令生成的迁移文件（如代码 9.39 所示）会在 users 表中添加 `admin` 这一列，得到的数据模型如图 9.13 所示。

users	
id	integer
name	string
email	string
password_digest	string
remember_token	string
admin	boolean
created_at	datetime
updated_at	datetime

图 9.13：添加了 `admin` 属性后的 User 模型

代码 9.39：为 User 模型添加 `admin` 属性所用的迁移文件
`db/migrate/[timestamp]_add_admin_to_users.rb`

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
```

```
    add_column :users, :admin, :boolean, default: false
  end
end
```

注意，在代码 9.40 中，我们为 add_column 方法指定了 default: false 参数，添加这个参数后用户默认情况下就不是管理员。（如果没有指定 default: false，admin 的默认值是 nil，也是“假值”，所以严格来说，这个参数不是必须的。不过，指定这个参数，可以更明确地向 Rails 以及代码的阅读者表明这段代码的意图。）

然后，我们要在“开发数据库”中执行迁移操作，还要准备好“测试数据库”：

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

和预想的一样，Rails 可以自动识别 admin 属性的类型为布尔值，而且自动生成了 admin? 方法：

```
$ rails console --sandbox
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

执行迁移操作后，针对 admin 属性的测试应该可以通过了：

```
$ bundle exec rspec spec/models/user_spec.rb
```

最后，我们要修改一下生成示例用户的代码，把第一个用户设为管理员，如代码 9.40 所示。

代码 9.40：生成示例用户的代码，把第一个用户设为管理员
lib/tasks/sample_data.rake

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    admin = User.create!(name: "Example User",
                         email: "example@railstutorial.org",
                         password: "foobar",
                         password_confirmation: "foobar",
                         admin: true)
    .
    .
    .
  end
end
```

之后还要还原数据库，并且重新生成示例用户：

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake test:prepare
```

“健壮参数”再探

你可能注意到了，在代码 9.40 中，我们在初始化 Hash 参数中指定了 `admin: true` 把用户设为管理员。这么做的后果是用户对象暴露在网络中了，如果在请求中提供初始化参数，恶意用户就可以发送如下的 PATCH 请求[8]：

```
patch /users/17?admin=1
```

这个请求会把 17 号用户设为管理员，这怎么说也是个潜在的安全隐患。

鉴于此，必须只允许传入可安全编辑的属性。我们在 7.3.2 节中说过，可以使用“健壮参数”，在 params Hash 上调用 `require` 和 `permit` 方法：

```
    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
```

注意，`admin` 不并在允许使用的属性列表中。这样就可以避免用户取得网站的管理权。因为这一步很重要，最好再测试一下各属性是否是可访问的，对 `:admin` 属性的可访问性测试留作练习，参见 9.6 节。

9.4.2 destroy 动作

编写完整的 Users 资源还要再添加删除链接和 `destroy` 动作。我们先在用户列表页面每个用户后面都加入一个删除链接，而且限制只有管理员才能看到这些链接。

编写针对删除功能的测试，最好能有个创建管理员的工厂方法，为此，我们可以在预构件中加入一个名为 `:admin` 的块，如代码 9.41 所示。

代码 9.41：添加一个创建管理员的工厂方法
spec/factories.rb

```
FactoryGirl.define do
  factory :user do
    sequence(:name)  { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com"}
    password "foobar"
    password_confirmation "foobar"
```

8. 类似 curl 的命令行工具可以发送这种 PATCH 请求。

```
    factory :admin do
      admin true
    end
  end
end
```

添加了以上代码之后，我们就可以在测试中调用 `FactoryGirl.create(:admin)` 创建管理员用户了。

基于安全考虑，普通用户是看不到删除用户链接的，所以：

```
it { should_not have_link('delete') }
```

只有管理员才能看到删除用户链接，如果管理员点击了删除用户链接，该用户会被删除，用户的数量就会减少 1 个：

```
it { should have_link('delete', href: user_path(User.first)) }
it "should be able to delete another user" do
  expect do
    click_link('delete', match: :first)
  end.to change(User, :count).by(-1)
end
it { should_not have_link('delete', href: user_path(admin)) }
```

上述代码中用到了 `match: :first`，这个参数告知 Capybara，不管是哪个删除链接，只点击第一个看到的。注意，我们还添加了一个测试，确保管理员不会看到删除自己的链接。针对删除用户的完整测试如代码 9.42 所示。

代码 9.42：测试删除用户功能
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do

    let(:user) { FactoryGirl.create(:user) }

    before do
      sign_in user
      visit users_path
    end

    it { should have_title('All users') }
    it { should have_content('All users') }
```

```
    describe "pagination" do
      .
      .
      .
    end

    describe "delete links" do

      it { should_not have_link('delete') }

      describe "as an admin user" do
        let(:admin) { FactoryGirl.create(:admin) }
        before do
          sign_in admin
          visit users_path
        end

        it { should have_link('delete', href: user_path(User.first)) }
        it "should be able to delete another user" do
          expect do
            click_link('delete', match: :first)
          end.to change(User, :count).by(-1)
        end
        it { should_not have_link('delete', href: user_path(admin)) }
      end
    end
  end
  .
  .
  .
end
```

然后在视图中加入代码 9.43。注意链接中的 `method: delete` 参数，它指明点击链接后发送的是 `DELETE` 请求。我们还把各链接放在了 `if` 语句中，这样就只有管理员才能看到删除用户链接。管理员看到的页面如图 9.14 所示。

代码 9.43：删除用户的链接（只有管理员才能看到）
app/views/users/_user.html.erb

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
  <% if current_user.admin? && !current_user?(user) %>
    | <%= link_to "delete", user, method: :delete,
                                  data: { confirm: "You sure?" } %>
```

```
  <% end %>
</li>
```

图 **9.14**：显示有删除用户链接的用户列表页面（/users）

浏览器不能发送 DELETE 请求，Rails 通过 JavaScript 进行模拟的。也就是说，如果用户禁用了 JavaScript，那么删除用户的链接就不可用了。如果必须要支持没有启用 JavaScript 的浏览器，你可以使用一个发送 POST 请求的表单来模拟 DELETE 请求，这样即使浏览器的 JavaScript 被禁用了，删除用户的链接还是可用的，更多细节请观看 RailsCasts 第 77 集《Destroy Without JavaScript》。

若要删除用户的链接起作用，我们要定义 destroy 动作（参见表格 7.1）。在 destroy 动作中，先找到要删除的用户，使用 Active Record 提供的 destroy 方法删除这个用户，然后再转向用户列表页面，如代码 9.44 所示。

代码 9.44：加入 destroy 动作
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def destroy
```

```
    User.find(params[:id]).destroy
    flash[:success] = "User destroyed."
    redirect_to users_url
  end
  .
  .
  .
end
```

注意上述 destroy 动作中，把 find 方法和 destroy 方法链在一起使用了：

```
User.find(params[:id]).destroy
```

理论上，只有管理员才能看到删除用户的链接，所以只有管理员才能删除用户。但实际上，还是存在一个严重的安全隐患：只要攻击者有足够的经验，就可以在命令行中发送 DELETE 请求，删除网站中的用户。为了保证网站的安全，我们还要限制对 destroy 动作的访问，因此我们在测试中不仅要确保只有管理员才能删除用户，还要保证其他用户不能执行删除操作，如代码 9.45 所示。注意，和代码 9.11 中的 put 方法类似，在这段代码中我们使用 delete 方法向指定的地址（user_path，参见表格 7.1）发送了一个 DELETE 请求。

代码 9.45：测试访问受限的 destroy 动作
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as non-admin user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:non_admin) { FactoryGirl.create(:user) }

      before { sign_in non_admin, no_capybara: true }

      describe "submitting a DELETE request to the Users#destroy action" do
        before { delete user_path(user) }
        specify { expect(response).to redirect_to(root_path) }
      end
    end
  end
end
```

理论上来说，网站中还是有一个安全漏洞，管理员可以发送 DELETE 请求删除自己。有些人可能会想，这样的管理员是自作自受。不过作为开发人员，我们最好还是要避免这种情况的发生，具体的实现留作练习，参见 9.6 节。

你可能已经知道了，我们要使用一个事前过滤器限制对 destroy 动作的访问，如代码 9.46 所示。

代码 9.46： 限制只有管理员才能访问 destroy 动作的事前过滤器
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_action :correct_user,   only: [:edit, :update]
  before_action :admin_user,     only: :destroy
  .
  .
  .
  private
    .
    .
    .
    def admin_user
      redirect_to(root_path) unless current_user.admin?
    end
end
```

至此，所有的测试应该都可以通过了，而且 Users 相关的资源，包括控制器、模型和视图，都已经实现了。

```
$ bundle exec rspec spec/
```

9.5 小结

我们用了好几章来介绍如何实现 Users 资源，在 5.4 节用户还不能注册，而现在不仅可以注册，还可以登录、退出、查看个人资料、修改设置，还能浏览网站中所有的用户列表，某些用户甚至可以删除其他的用户。

本书剩下的内容会以这个 Users 资源为基础（以及相关的权限授权系统），在第 10 章中为示例程序加入类似 Twitter 的微博功能，在第 11 章中实现关注用户的状态列表。最后这两章会介绍几个 Rails 中最为强大的功能，其中就包括通过 has_many 和 has_many through 实现的数据模型关联。

在继续阅读之前，先把本章所做的改动合并到主分支：

```
$ git add .
$ git commit -m "Finish user edit, update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
```

你还可以将程序部署到“生产环境”，再生成示例用户（在此之前要使用 pg:reset 命令还原“生产数据库”）：

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:populate
```

若要改动显示出来，可能要在 Heroku 中强制重启应用程序：

```
$ heroku restart
```

还有一点需要注意，本章我们加入了程序所需的最后一个 gem，最终的 `Gemfile` 如代码 9.47 所示。

代码 9.47：示例程序所需 `Gemfile` 的最终版本

```
source 'https://rubygems.org'
ruby '2.0.0'
#ruby-gemset=railstutorial_rails_4_0

gem 'rails', '4.0.2'
gem 'bootstrap-sass', '2.3.2.0'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.1.2'
gem 'will_paginate', '3.0.4'
gem 'bootstrap-will_paginate', '0.0.9'

group :development, :test do
  gem 'sqlite3', '1.3.8'
  gem 'rspec-rails', '2.13.1'
  # The following optional lines are part of the advanced setup.
  # gem 'guard-rspec', '2.5.0'
  # gem 'spork-rails', '4.0.0'
  # gem 'guard-spork', '1.5.0'
  # gem 'childprocess', '0.3.6'
end

group :test do
  gem 'selenium-webdriver', '2.0.0'
  gem 'capybara', '2.1.0'
  gem 'factory_girl_rails', '4.2.0'
  gem 'cucumber-rails', '1.4.0', :require => false
  gem 'database_cleaner', github: 'bmabey/database_cleaner'

  # Uncomment this line on OS X.
  # gem 'growl', '1.0.3'

  # Uncomment these lines on Linux.
  # gem 'libnotify', '0.8.0'
```

```
  # Uncomment these lines on Windows.
  # gem 'rb-notifu', '0.0.4'
  # gem 'win32console', '1.3.2'
  # gem 'wdm', '0.1.0'
end

gem 'sass-rails', '4.0.1'
gem 'uglifier', '2.1.1'
gem 'coffee-rails', '4.0.1'
gem 'jquery-rails', '2.2.1'
gem 'turbolinks', '1.1.1'
gem 'jbuilder', '1.0.2'

group :doc do
  gem 'sdoc', '0.3.20', require: false
end

group :production do
  gem 'pg', '0.15.1'
  gem 'rails_12factor', '0.0.2'
end
```

9.6 练习

1. 参照代码 9.48，直接向 `update` 动作发起 `PATCH` 请求，验证一下无法修改 `admin` 属性。确保测试先是红色的，然后才会变绿。（提示：先要把 `admin` 加入 `user_params` 方法的允许访问属性列表中。）

2. 把代码 9.3 中修改 Gravatar 头像的链接（“change”），使链接在新窗口（或新标签）中打开。提示：请搜索，你会发现一个很常用的方法，涉及到 `_blank` 的用法。

3. 现在针对身份验证系统的测试会确保用户登录后能看到“Profile”和“Settings”等导航链接。增加一个测试，确保用户未登录时看不到这些导航链接。

4. 在测试中尽量多的使用代码 9.6 中的 `sign_in` 帮助方法。

5. 使用代码 9.49 中的代码重构 `new.html.erb` 和 `edit.html.erb` 中的表单。注意，你要明确的传入 `f` 这个表单变量，如代码 9.50 所示。你还要修改相应的测试，因为表单已经不完全一样了。仔细的查找修改前后表单的差异，据此修改测试。

6. 已经登录的用户就没必要再访问 Users 控制器的 `new` 和 `create` 动作了，修改程序，如果登录后的用户访问这些地址时，转向到网站首页。

7. 在网站的布局中插入一些 Rails API中介绍的方法，了解一下 `request` 对象。（如果遇到了困难，可以参考代码 7.1。）

8. 编写一个测试，确保友好转向只在第一次转向指定的地址，其后再登录的话就转向默认设定的地址（如资料页面）。代码 9.51 是个提示，其实也就是所需的代码。

9. 修改 destroy 动作，避免管理员删除自己。（先编写测试。）

代码 9.48：测试 admin 属性是否无法修改
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    .
    .
    .
    describe "forbidden attributes" do
      let(:params) do
        { user: { admin: true, password: user.password,
                  password_confirmation: user.password } }
      end
      before { patch user_path(user), params }
      specify { expect(user.reload).not_to be_admin }
    end
  end
end
```

代码 9.49：注册和编辑表单字段的局部视图
app/views/users/_fields.html.erb

```
<%= render 'shared/error_messages' %>

<%= f.label :name %>
<%= f.text_field :name %>

<%= f.label :email %>
<%= f.text_field :email %>

<%= f.label :password %>
<%= f.password_field :password %>

<%= f.label :password_confirmation, "Confirm Password" %>
<%= f.password_field :password_confirmation %>
```

代码 9.50：使用局部视图后的注册页面视图
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'fields', f: f %>
      <%= f.submit "Create my account", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

代码 **9.51**：测试友好的转向后，只能转向到默认的页面
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      .
      .
      .
      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email",    with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do

          it "should render the desired protected page" do
            expect(page).to have_title('Edit user')
          end

          describe "when signing in again" do
            before do
              delete signout_path
              visit signin_path
              fill_in "Email",    with: user.email
```

```
            fill_in "Password", with: user.password
            click_button "Sign in"
          end

          it "should render the default (profile) page" do
            expect(page).to have_title(user.name)
          end
        end
      end
    end
    .
    .
    .
  end
end
```

第 10 章 用户的微博

我们在第 9 章中已经实现了一个完整且符合 REST 架构的资源：用户，本章我们要再实现一个资源：用户微博（micropost）。[1]微博是由用户发布的一种简短消息，我们在第 2 章中实现了微博的雏形。 本章我们会在 2.3 节的基础上，实现一个功能完善的 Microposts 资源。首先，我们要创建微博所需的数据模型，通过 `has_many` 和 `belongs_to` 方法把微博和用户关联起来，再建立处理和显示微博所需的表单及局部视图。在 第 11 章，还要加入关注其他用户的功能，其时，我们这个山寨版 Twitter 才算完成。

如果你使用 Git 做版本控制的话，和之前一样，我建议你新建一个分支：

```
$ git checkout -b user-microposts
```

10.1 Microposts 模型

实现 Microposts 资源的第一步是创建微博所需的数据模型，在模型中设定微博的基本属性。和 2.3 节 创建的模型类似，我们要实现的 Micropost 模型要包含数据验证，以及和 User 模型的关联。除此之外，我们还会做充分的测试，指定默认的排序方式，自动删除已注销用户的微博。

10.1.1 基本模型

Micropost 模型只需要两个属性：一个是 `content`，用来保存微博的内容；[2]另一个是 `user_id`，把微博和用户关联起来。我们要使用 `generate model` 命令生成所需的模型，这一点和创建用户模型时是一样的（参见代码 6.1）：

```
$ rails generate model Micropost content:string user_id:integer
```

这个命令有可能会生成一个微博预构件，你应该把它删掉，因为后面我们会自己编写所需的预构件（参见10.1.4 节）：

```
$ rm -f spec/factories/microposts.rb
```

上述命令会生成一个迁移文件，在数据库中生成一个名为 `microposts` 的表（参见代码 10.1）。读者朋友可以和生成 `users` 表的迁移文件对照一下（参见代码 6.2）。

代码 10.1：创建微博模型的迁移文件（注意：为 `user_id` 和 `created_at` 列加入了索引）
`db/migrate/[timestamp]_create_microposts.rb`

1. 严格来说，我们在第 8 章中是把 session 当做资源来处理的，不过 session 不会像 Users 和 Microposts 资源那样被存入数据库。
2. `content` 属性是字符串（`string`）类型，不过我们曾在 2.1.2 节中简要介绍过，较长的文本应该使用 `text` 类型。

```
class CreateMicroposts < ActiveRecord::Migration
  def change
    create_table :microposts do |t|
      t.string :content
      t.integer :user_id

      t.timestamps
    end
    add_index :microposts, [:user_id, :created_at]
  end
end
```

注意，因为我们设想要按照发布时间的倒序查询某个用户所有的微博，所以在上述代码中为 `user_id` 和 `created_at` 列加入了索引：

```
add_index :microposts, [:user_id, :created_at]
```

我们把 `user_id` 和 `created_at` 放在一个数组中，告诉 Rails 我们要创建的是“多键索引（multiple key index）”，Active Record 便会同时使用这两个键。还要注意 `t.timestamps` 这行，我们在 6.1.1 节中介绍过，它会自动创建 `created_at` 和 `updated_at` 两个属性。在 10.1.4 节 和 10.2.1 节 中才会用到 `created_at`。

我们先参照 User 模型的测试（参照代码 6.5），为 Micropost 模型编写一些基本的测试。我们要测试微博对象是否可以响应 `content` 和 `user_id` 方法，如代码 10.2 所示。

代码 10.2： Micropost 模型测试（初始版）
`spec/models/micropost_spec.rb`

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before do
    # This code is not idiomatically correct.
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
  end

  subject { @micropost }

  it { should respond_to(:content) }
  it { should respond_to(:user_id) }
end
```

若要这个测试通过，我们先要执行数据库迁移，再准备好“测试数据库”：

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

执行上面两个命令之后，会生成 Micropost 模型，结构如图 10.1 所示。

microposts	
id	integer
content	string
user_id	integer
created_at	datetime
updated_at	datetime

图 **10.1**：Micropost 数据模型

然后确认测试是否可以通过：

```
$ bundle exec rspec spec/models/micropost_spec.rb
```

测试虽然可以通过，不过你可能注意到代码 10.2 中的这几行代码了：

```
let(:user) { FactoryGirl.create(:user) }
before do
  # This code is not idiomatically correct.
  @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
end
```

就像其中的注释所说，`before` 块中的代码并不完全正确。你可以想一下为什么，我们会在 10.1.3 节中告诉你答案。

10.1.2 第一个数据验证

Micropost 模型必须要有一个属性表明用户的 ID，这样才能知道是哪个用户发布的微博。要加入这个属性，最好的方法是使用 Active Record 关联，会在 10.1.3 节实现。实现的过程需要一点重构，本节我们就编写一个测试捕获可能出现的错误。

测试如代码 10.3 所示，我们先把用户 ID 设为 `nil`，然后检测对应的微博是否不合法。（可以和代码 6.8 中针对 User 模型的测试对比一下。）

代码 10.3：测试微博能否通过验证
`spec/models/micropost_spec.rb`

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before do
    # This code is not idiomatically correct.
```

```
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
  end

  subject { @micropost }

  it { should respond_to(:content) }
  it { should respond_to(:user_id) }

  it { should be_valid }

  describe "when user_id is not present" do
    before { @micropost.user_id = nil }
    it { should_not be_valid }
  end
end
```

这段代码测试了微博是否能够通过验证，以及是否指定了 `user_id` 的值。要想让上述测试通过，我们要按照代码 10.4 所示，加入一个简单的存在性验证。

代码 10.4： 对微博 `user_id` 属性的验证
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  validates :user_id, presence: true
end
```

10.1.3 用户和微博之间的关联

在为 Web 程序构建数据模型时，最基本的考虑要素是要能够在不同的模型之间建立关联。在我们这个程序中，每篇微博都关联着一个用户，而每个用户一般都会关联多篇微博。用户和微博之间的关系在 2.3.3 节中简单的介绍过，二者之间的关系如图 10.2 和图 10.3 所示。在实现这种关联的时候，我们会编写针对 Micropost 模型的测试，还会编写针对 User 模型的测试。

图 10.2： 微博和用户之间的“属于（`belongs_to`）”关系

图 10.3：用户和微博之间的“拥有多个（`has_many`）”关系

使用本小节介绍的 `belongs_to` 和 `has_many` 之后，Rails 会自动创建如表格 10.1 所示的方法。

表格 10.1：用户和微博关联后所得方法的简介

方法	作用
`micropost.user`	返回该微博对应的用户对象
`user.microposts`	返回该用户的所有微博数组
`user.microposts.create(arg)`	创建一篇微博（`user_id = user.id`）
`user.microposts.create!(arg)`	创建一篇微博（失败时抛出异常）
`user.microposts.build(arg)`	生成一个新的微博对象（user_id = user.id）

注意，从表格 10.1 可知，相较于以下的方法

```
Micropost.create
Micropost.create!
Micropost.new
```

我们得到了

```
user.microposts.create
user.microposts.create!
user.microposts.build
```

后者才是创建微博的正确方式，即通过相关联的用户对象创建。通过这种方式创建的微博，其 `user_id` 属性会自动设为正确的值。所以，我们可以把代码 10.3 中的下述代码

```
let(:user) { FactoryGirl.create(:user) }
before do
  # This code is not idiomatically correct.
  @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
end
```

修改为

```
let(:user) { FactoryGirl.create(:user) }
before { @micropost = user.microposts.build(content: "Lorem ipsum") }
```

只要正确定义了用户和微博之间的关联关系，`@micropost` 变量的 `user_id` 属性就会自动设为相对应用户的 id。

如表格 10.1 所示，用户和微博建立关联之后，还会生成 `micropost.user` 方法，返回该微博的用户对象。对此，我们可以使用 `it` 和 `its` 做个测试：

```
it { should respond_to(:user) }
its(:user) { should eq user }
```

以上对 Micropost 模型的测试结合在一起后如代码 10.5 所示。

代码 10.5：测试微博和用户之间的关联
spec/models/micropost_spec.rb

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before { @micropost = user.microposts.build(content: "Lorem ipsum") }

  subject { @micropost }

  it { should respond_to(:content) }
  it { should respond_to(:user_id) }
  it { should respond_to(:user) }
  its(:user) { should eq user }

  it { should be_valid }

  describe "when user_id is not present" do
    before { @micropost.user_id = nil }
    it { should_not be_valid }
  end
end
```

在用户和微博关联关系的 User 模型一边，我们会在 10.1.4 节做详细的测试，现在我们只是简单的测试下是否可以响应 microposts 方法，如代码 10.6 所示。

代码 10.6： 测试用户对象是否可以响应 microposts 方法
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }
  .
  .
  .
  it { should respond_to(:admin) }
  it { should respond_to(:microposts) }
  .
  .
  .
end
```

写好了上面的测试，实现用户和微博之间关联就简单了：只需分别加入下面这两行代码，代码 10.5 和代码 10.6 中的测试就可以通过了：belongs_to :user（如代码 10.7 所示）和 has_many :microposts（如代码 10.8 所示）。

代码 10.7： 微博“属于（belongs_to）”用户
app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :user_id, presence: true
end
```

代码 10.8： 用户“拥有多篇（has_many）”微博
app/models/user.rb

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

现在，你应该结合表格 10.1和代码 10.5、代码 10.6，确保你理解了关联的基本知识点。你还应该检查一下测试是否可以通过：

```
$ bundle exec rspec spec/models
```

10.1.4 改进 Micropost 模型

代码 10.6 中的代码并没有深入测试通过 `has_many` 实现的关联，仅仅检测了是否可以响应 `microposts` 方法。本小节，我们会为 Micropost 模型加入排序方法和依属关系，还会测试 `user.microposts` 方法的返回结果是否为数组。

我们需要在 User 模型的测试中生成一些微博，所以现在我们先要创建一个生成微博的预构件。在所创建的预构件中我们要找到一种方法把微博和用户关联起来，幸运的是，在 FactoryGirl 中实现关联是很容易的，如代码 10.9 所示。

代码 10.9：完整的预构件文件，包含了创建微博的新预构件
`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    sequence(:name)  { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com"}
    password "foobar"
    password_confirmation "foobar"

    factory :admin do
      admin true
    end
  end

  factory :micropost do
    content "Lorem ipsum"
    user
  end
end
```

在 FactoryGirl 中我们只需要在创建微博的预构件中包含一个用户对象就可以实现所需的关联了：

```
factory :micropost do
  content "Lorem ipsum"
  user
end
```

在下一节中会介绍，我们可以使用下面的方法生成一篇微博：

```
FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
```

默认作用域

默认情况下，使用 `user.microposts` 从数据库中读取用户的微博不能保证微博的次序，但是按照博客和 Twitter 的习惯，我们希望微博按照创建时间倒序排列，也就是最新创建的微博在最前面。要测试微博的次序，我们要先创建两篇微博：

```
FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
```

我们把第二篇微博的创建时间设的晚一些，即 `1.hour.ago`（利用了旁注 8.1中介绍的帮助函数），第一篇微博的创建时间要早一些，是 `1.day.ago`。请注意一下使用 FactoryGirl 创建微博是多么方便：我们不仅可以直接指定微博所属的用户（FactoryGirl 会逃过 `attr_accessible` 限制），还可以设定通常情况下不能自由设定的 `created_at` 属性（因为在 Active Record 做了限制）。（再次说明一下，`created_at` 和 `updated_at` 两个属性是“魔法”列，会被自动设为相应的创建时间戳和更新时间戳，即使手动指定了值也会被覆盖。）

大多数数据库适配器（包括 SQLite 的适配器）读取的微博都是按照 ID 来排序的，因此代码 10.10 中的测试肯定不会通过。在这段测试代码中没有使用 `let`，而用了 `let!`（读作“let bang”），因为 `let` 方法指定的变量是“惰性”的，只有当后续有引用时才会被创建。而我们希望这两个微博变量立即被创建，这样才能保证两篇微博时间戳的顺序是正确的，也保证了 `@user.microposts` 数组不是空的。所以我们才用了 `let!` 方法，强制相应的变量立即被创建。

代码 10.10： 测试用户微博的次序
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  .
  .
  .
  describe "micropost associations" do

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end

    it "should have the right microposts in the right order" do
      expect(@user.microposts.to_a).to eq [newer_micropost, older_micropost]
    end
  end
end
```

这个测试中最关键的一行是：

```
expect(@user.microposts.to_a).to eq [newer_micropost, older_micropost]
```

这行代码表明所创建的微博应该按照创建时间倒序排列，即最新创建的微博排在最前面。这个测试注定是无法通过的，因为微博默认是按照 ID 排序的，即 `[older_micropost, newer_micropost]`。这个测试同时也验证了 `has_many` 关联最基本的效果是否正确，即检测 `user.microposts` 返回的结果是否是数组。其中用到的 `to_a` 方法，在 4.3.1 节中介绍过，会把 `@user.microposts` 从初始状态（Active Record 的“集合代理（collection proxy）”）转换成一个数组，以便和我们手动编写的数组比较。

要让这个测试通过，我们要使用 Rails 中的 `default_scope` 方法，还要设定它的 `:order` 参数，如代码 10.11 所示。（这是我们第一次接触作用域的概念，在第 11 章中会介绍作用域更一般的用法。）

代码 10.11：通过 `default_scope` 设定微博的排序
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order('created_at DESC') }
  validates :user_id, presence: true
end
```

我们通过 `microposts.created_at DESC` 设定了所需的排序，其中 `DESC` 在 SQL 中是“倒序”的意思，即按照由新到旧这种顺序排序。

从 Rails 4.0 开始，所有的作用域都需要通过匿名函数的方式定义，函数的返回值为所需的条件。这么做主要的好处是，作用域的条件不用立即计算，只在需要时才执行（所以叫做“惰性计算”）。本例所用的匿名函数是

```
-> { order('created_at DESC') }
```

这种对象叫 Proc 或 lambda，以 `->` 开头，后面跟着一个代码块，调用 `call` 方法时才执行块中的代码。我们可以在控制台中看一下：

```
>> -> { puts "foo" }
=> #<Proc:0x007fab938d0108@(irb):1 (lambda)>
>> -> { puts "foo" }.call
foo
=> nil
```

（Proc 属于 Ruby 语言的高级功能，所以如果没有立刻理解也不用担心。）

依属关系：destroy

除了设定恰当的排序外，我们还要对微博模型做另一项改进。我们在 9.4 节中介绍过，管理员是有权限删除用户的。那么，在删除用户的同时，就有必要把该用户发布的微博也删除。对此我们可以编写一个测试，检测当用户被删除后，其发布的微博是否还在数据库中。

为了能够正确的测试微博是否被删除了，我们先要把用户的一篇微博赋值给一个局部变量，然后再删除这个用户。对此，一种比较直观的实现方式如下所示：

```
microposts = @user.microposts.to_a
@user.destroy
expect(microposts).not_to be_empty
microposts.each do |micropost|
  # Make sure the micropost doesn't appear in the database.
end
```

我们调用了 to_a 方法，高效地复制了一份微薄列表。我们还加入了下面这行

```
expect(microposts).not_to be_empty
```

这是一份双保险，避免不小心漏掉了 to_a。如果没有 to_a 的话，一旦删除了用户就会删掉 microposts 变量中的该用户的微博，那么

```
microposts.each do |micropost|
  # Make sure the micropost doesn't appear in the database.
end
```

就什么也测试不了，因为 microposts 是空的。

我们可以通过下面的代码确认微博不在数据库中：

```
microposts.each do |micropost|
  expect(Micropost.where(id: micropost.id)).to be_empty
end
```

这里我们用的是 Micropost.where，二没用 Micropost.find，因为如果没有查询到记录 where 会返回一个空值对象，而 find 会抛出异常。测试空值对象要更容易。如果你好奇想知道如何测试异常，可以使用下面的代码：

```
expect do
  Micropost.find(micropost)
end.to raise_error(ActiveRecord::RecordNotFound)
```

完整的测试代码如代码 10.12 所示。

代码 10.12： 测试用户删除后，所发布的微博是否也被删除了
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  .
  .
```

```
  .
  describe "micropost associations" do

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end
    .
    .
    .
    it "should destroy associated microposts" do
      microposts = @user.microposts.to_a
      @user.destroy
      expect(microposts).not_to be_empty
      microposts.each do |micropost|
        expect(Micropost.where(id: micropost.id)).to be_empty
      end
    end
  end
end
```

要让代码 10.12 中的测试通过，我们甚至不需要加入一行完整的代码，只需在 has_many 方法中设定一个参数即可，如代码 10.13 所示。

代码 10.13：保证用户的微博在删除用户的同时也会被删除
app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  .
  .
  .
end
```

上面代码中有这么一行：

```
has_many :microposts, dependent: :destroy
```

其中的 dependent: :destroy 设定程序在用户被删除的时候，其所属的微博也要被删除。这么一来，如果管理员删除了用户，数据库中就不会出现无主的微博了。

至此，用户和微博之间的关联就设置好了，所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

10.1.5 验证微博内容

在结束讨论 Micropost 模型之前，我们还要为微博的内容加上数据验证（参照 2.3.2 节）。和 `user_id` 一样，`content` 属性不能为空，而且还要限制内容的长度不能多于 140 个字符，这才是真正的“微”博。我们要编写的测试和 6.2 节中对用户模型的验证测试类似，如代码 10.14 所示。

代码 10.14： 测试 Micropost 模型的数据验证
`spec/models/micropost_spec.rb`

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before { @micropost = user.microposts.build(content: "Lorem ipsum") }
  .
  .
  .
  describe "when user_id is not present" do
    before { @micropost.user_id = nil }
    it { should_not be_valid }
  end

  describe "with blank content" do
    before { @micropost.content = " " }
    it { should_not be_valid }
  end

  describe "with content that is too long" do
    before { @micropost.content = "a" * 141 }
    it { should_not be_valid }
  end
end
```

和 6.2 节一样，在代码 10.14 中我们用到了字符串乘积来测试微博内容长度的验证：

```
$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

我们需要在程序中加入下面这行代码：

```
validates :content, presence: true, length: { maximum: 140 }
```

Micropost 模型的最终代码如代码 10.15 所示。

代码 10.15：Micropost 模型的数据验证
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order('created_at DESC') }
  validates :content, presence: true, length: { maximum: 140 }
  validates :user_id, presence: true
end
```

10.2 显示微博

尽管我们还没实现直接在网页中发布微博的功能（将在 10.3.2 节实现），不过我们还是有办法显示微博，并对显示的内容进行测试。我们将按照 Twitter 的方式，用户的微博不在 `index` 页面中显示，而在 `show` 页面中，构思图如图 10.4 所示。我们会先创建一些很简单的 ERb 代码，在用户的资料页面显示微博，然后要在 9.3.2 节中实现的数据生成器中加入生成微博的代码，这样我们才有内容可以显示。

和 8.2.1 节中对登录机制的介绍类似，10.2.1 节也会经常将一些元素推送到堆栈里，然后再一个一个的从栈尾取出来。如果理解起来有点困难，多点耐心，你的付出会在 10.2.2 节得到回报。

10.2.1 充实用户资料页面

我们先在用户的 request spec 中加入对显示微博的测试。我们采用的方法是，先通过预构件创建几篇微博，然后检查用户资料页面是否显示了这几篇微博，同时我们还要验证是否显示了如图 10.4 中所示的总的微博数量。

我们可以使用 `let` 方法创建微博，不过如代码 10.10 所示，我们希望用户和微博的关联立即生效，这样微博才能显示在用户的资料页面中。所以，我们要使用 `let!` 方法：

```
let(:user) { FactoryGirl.create(:user) }
let!(:m1) { FactoryGirl.create(:micropost, user: user, content: "Foo") }
let!(:m2) { FactoryGirl.create(:micropost, user: user, content: "Bar") }

before { visit user_path(user) }
```

图 10.4： 显示了微博的资料页面构思图

按照上面这种方式定义了微博后，我们就可以使用代码 10.16 中的代码测试用户资料页面中是否显示了这些微博。

代码 10.16： 检测用户资料页面是否显示了微博的测试
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "profile page" do
    let(:user) { FactoryGirl.create(:user) }
    let!(:m1) { FactoryGirl.create(:micropost, user: user, content: "Foo") }
    let!(:m2) { FactoryGirl.create(:micropost, user: user, content: "Bar") }

    before { visit user_path(user) }

    it { should have_content(user.name) }
```

```
    it { should have_title(user.name) }

    describe "microposts" do
      it { should have_content(m1.content) }
      it { should have_content(m2.content) }
      it { should have_content(user.microposts.count) }
    end
  end
  .
  .
  .
end
```

注意，我们可以在关联关系的方法上调用 `count` 方法：

```
user.microposts.count
```

这个 `count` 方法是很聪明的，可以直接在数据库层统计数量。也就是说，`count` 的计数过程不是把微博从数据库中取出来，然后再在所得的数组上调用 `length` 方法，如果这样做的话，微博数量一旦很多的话，效率就会很低。其实，`count` 方法会直接在数据库层中统计用户的微博数量。如果统计数量仍然是程序的性能瓶颈的话，你可以使用“计数缓存”进一步提速。

在加入代码 10.18 之前，代码 10.16 中的测试是无法通过的，不过现在我们可以先在用户的资料页面中加入一些微博，如代码 10.17 所示。

代码 10.17：在用户资料页面中加入微博
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
  .
  .
  .
  <aside>
    .
    .
    .
  </aside>
  <div class="span8">
    <% if @user.microposts.any? %>
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
```

```
  </div>
</div>
```

微博列表稍后分析，现在先看看其他部分。在这段代码中，`if @user.microposts.any?`（在代码 7.24 中见过类似的用法）的作用是，如果用户没有发布微博的话，就不会显示后面的列表。

还要注意一下，在代码 10.17 中我们也加入了微博的分页显示功能：

```
<%= will_paginate @microposts %>
```

如果和用户索引页面中对应的代码（参见代码 9.33）比较的话，会发现，之前所用的代码是：

```
<%= will_paginate %>
```

之前之所以可以直接调用，是因为在 Users 控制器中，`will_paginate` 默认程序中存在一个名为 `@users` 的变量（在 9.3.3 节中介绍过，该变量的值必须是 `AvtiveRecord::Relation` 的实例）。现在，虽然我们还在 Users 控制器中，但是我们要对微博分页，所以 `will_paginate` 方法要指定 `@microposts` 变量作为参数。当然了，我们还要在 `show` 动作中定义 `@microposts` 变量（参见代码 10.19）。

接着，还显示了当前微博的数量：

```
<h3>Microposts (<%= @user.microposts.count %>)</h3>
```

前面介绍过，`@user.microposts.count` 的作用和 `User.count` 类似，不过统计的微博数量却是建立在用户和微博的关联关系上的。

最后，我们来看一下显示微博的列表：

```
<ol class="microposts">
  <%= render @microposts %>
</ol>
```

这段代码使用了一个有序列表标签 `ol`，显示一个微博列表，不过关键的部分是通过一个微博局部视图实现的。在 9.3.4 节中介绍过，如下的代码

```
<%= render @users %>
```

会使用名为 `_user.html.erb` 的局部视图渲染 `@users` 变量中的每一个用户。因此，如下的代码

```
<%= render @microposts %>
```

会对微博做同样的渲染操作。所以，我们要创建名为 `_micropost.html.erb` 的局部视图（存放在微博对应的视图文件夹中），如代码 10.18 所示。

代码 10.18： 显示单篇微博的局部视图
`app/views/microposts/_micropost.html.erb`

```
<li>
  <span class="content">
    micropost.content
  </span>
  <span class="timestamp">
    Posted time_ago_in_words(micropost.created_at) ago.
  </span>
</li>
```

这段代码使用了 `time_ago_in_words` 帮助方法，会在 10.2.2 节中介绍。

图 **10.5**：添加了显示微博代码后的用户资料页面，不过还没有微博可显示

至此，尽管定义了所需的全部视图，但是代码 10.16 中的测试仍旧无法通过，提示未定义 `@microposts` 变量。加入代码 10.19 中的代码后，测试就可以通过了。

代码 10.19：在 Users 控制器的 `show` 动作中加入 `@microposts` 实例变量
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def show
    @user = User.find(params[:id])
    @microposts = @user.microposts.paginate(page: params[:page])
```

```
  end
  .
  .
  .
end
```

请注意一下 paginate 方法是多么的智能，它甚至可以通过关联关系，在 microposts 数据表中取出每个分页中要显示的微博。

现在，我们可以查看一下刚编好的用户资料页面了，如图 10.5 所示，可能会出乎你的意料，这也是理所当然的，因为我们还没有发布微博呢。下面我们就来发布微博。

10.2.2 示例微博

在 10.2.1 节中为了显示微博，创建了几个视图，但是结果有点不给力。为了改变这个悲剧，我们要在 9.3.2 节中用到的示例数据生成器中加入生成微博数据的代码。如果给所有的用户都生成一些微博的话要用很长的时间，所以我们暂且只给前六个用户[3]生成微博数据，这要用到 User.all 方法的 :limit 选项：[4]

```
users = User.all(limit: 6)
```

我们要为每个用户生成 50 篇微博（这个数量大于单页显示的 30 篇限制），使用 Faker gem 中简便的 Lorem.sentence 方法生成每篇微博的内容。（Faker::Lorem.sentence 生成的是 lorem ipsum 示例文字，我们在第 6 章中介绍过，lorem ipsum 背后有一段有趣的故事。）代码 10.20 中显示的是修改后的示例数据生成器。

代码 10.20：在示例数据生成器中加入生成微博的代码
lib/tasks/sample_data.rake

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    .
    .
    .
    users = User.all(limit: 6)
    50.times do
      content = Faker::Lorem.sentence(5)
      users.each { |user| user.microposts.create!(content: content) }
    end
  end
end
```

当然，如果要生成示例数据，我们要执行 db:populate 命令：

3. 例如自定义了头像的 5 个用户和 使用 Gravatar 默认头像的 1 个用户。
4. 如果你对这个方法生成的 SQL 感兴趣，可以查看 log/development.log 文件。

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake test:prepare
```

然后，我们就能看到 10.2.1 节中劳动的果实了，在用户资料页面显示了生成的微博。[5]初步结果如图 10.6 所示。

图 10.6：用户资料页面（/users/1）中显示的尚未样式化的微博列表

图 10.6 中显示的微博列表还没有加入样式，那我们就加入一些样式（参见代码 10.21）[6]，再看一下页面显示的效果。图 10.7 显示的是第一个用户（当前登录用户）的资料页面，图 10.8 显示的是另一个用户的资料页面，图 10.9 显示的是第一个用户资料页面的第 2 页，页面底部还显示了分页链接。注意观察这三幅图，我们可以看到微博后面显示了距离发布的时间（例如，“Posted 1 minute ago.”），这个效果是通过代码 10.18 中的 `time_ago_in_words` 方法实现的。过一会再刷新页面，你会发现这些文字依据当前时间自动更新了。

5. Faker 中的 lorem ipsum 文本是被设计为随机生成的，所以你的示例微博可能和我的不一样。
6. 为了行文方便，代码 10.21 实际上包含了本章用到的所有 CSS。

图 **10.7**：显示了微博的用户资料页面（/users/1）

代码 **10.21**：微博列表的样式（包含了本章用到的所有样式）
app/assets/stylesheets/custom.css.scss

```
.
.
.

/* microposts */

.microposts {
  list-style: none;
  margin: 10px 0 0 0;

  li {
    padding: 10px 0;
    border-top: 1px solid #e8e8e8;
  }
}
.content {
  display: block;
}
```

```
.timestamp {
  color: $grayLight;
}
.gravatar {
  float: left;
  margin-right: 10px;
}
aside {
  textarea {
    height: 100px;
    margin-bottom: 5px;
  }
}
```

图 **10.8**：另一个用户的资料页面，也显示了微博列表（/users/5）

图 **10.9**：微博分页链接（/users/1?page=2）

10.3 微博相关的操作

微博的数据模型构建好了，也编写了相关的视图文件，接下来我们的开发重点是，通过网页发布微博。在实现的过程中，我们会第三次用到表单来创建资源，这一次创建的是 Microposts 资源[7]。本节，我们还会初步实现动态列表（status feed），在第 11 章再完善。最后，和 Users 资源类似，我们还要实现在网页中删除微博的功能。

上述功能的实现和之前的惯例有一点是不一样的地方，需要特别注意，那就是，Microposts 资源相关的页面不是通过 Microposts 控制器实现的，而是依赖于 Users 和 StaticPages 控制器。这也就意味着，Microposts 资源的路由设置是很简单的，如代码 10.22 所示。代码 10.22 中的代码所代表的符合 REST 结构的路由如表格 10.2 所示，表中的路由只是表格 2.3 的一部分。不过，路由虽然简化了，但预示着实现的过程需要更高级的技术，而不会减小代码的复杂度。从第 2 章起我们就十分依赖脚手架，不过现在我们将舍弃脚手架的大部分功能。

代码 10.22：Microposts 资源的路由设置
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users
  resources :sessions,   only: [:new, :create, :destroy]
  resources :microposts, only: [:create, :destroy]
```

7. 另外两个资源分别是 7.2 节 中的 Users 资源和 8.1 节中的 Sessions 资源。

```
  .
  .
  .
end
```

表格 10.2：代码 10.25 设置的 Microposts 资源路由

HTTP 请求	URL	动作	作用
POST	/microposts	`create`	创建新微博
DELETE	/microposts/1	`destroy`	删除 id 为 1 的微博

10.3.1 访问限制

开发 Microposts 资源的第一步，我们要在 Microposts 控制器中实现访问限制。我们要实现的效果很简单：若要访问 `create` 和 `destroy` 动作就要先登录。针对访问限制的 RSpec 测试如代码 10.23 所示。（在 10.3.4 节中，我们还会测试并加入第三层保护措施，确保只有微博的发布者才能删除该微博。）

代码 10.23： 限制访问 Microposts 资源的测试
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      .
      .
      .
      describe "in the Microposts controller" do

        describe "submitting to the create action" do
          before { post microposts_path }
          specify { expect(response).to redirect_to(signin_path) }
        end

        describe "submitting to the destroy action" do
          before { delete micropost_path(FactoryGirl.create(:micropost)) }
          specify { expect(response).to redirect_to(signin_path) }
        end
```

```
      end
      .
      .
      .
    end
  end
end
```

上述代码没有使用即将实现的网页界面，而是直接在 Microposts 控制器层面操作：如果向 /microposts 发送 POST 请求（`POST microposts_path` 访问的是 `create` 动作），或者向 /microposts/1 发送 DELETE 请求（`delete micropost_path(micropost)` 访问的是 `destroy` 动作），则会转向登录页面——我们在代码 9.13 中就用过这种方法。

我们要先对程序的代码做点重构，然后再加入程序中，让代码 10.23 中的测试通过。在 9.2.1 节中，我们定义了一个名为 `signed_in_user` 的事前过滤器（参见代码 9.12），确保访问相关的动作之前用户要先登录。那时，我们只需要在 Users 控制器中使用这个事前过滤器，但是现在我们在 Microposts 控制器中也要用到，那么我们就把它移到 Sessions 的帮助方法中，如代码 10.24 所示。[8]

代码 10.24： 把 `signed_in_user` 方法移到 Sessions 帮助方法中
`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def current_user?(user)
    user == current_user
  end

  def signed_in_user
    unless signed_in?
      store_location
      redirect_to signin_url, notice: "Please sign in."
    end
  end
  .
  .
  .
end
```

为了避免代码重复，同时还要把 `signed_in_user` 从 Users 控制器中删掉。

8. 我们在 8.2.1 节中介绍过，默认情况下帮助方法只能在视图中使用，因此要在控制器中使用 Sessions 帮助方法就要在 Application 控制器中加入代码 `include SessionHelper`。

加入了代码 10.24 之后，我们就可以在 Microposts 控制器中使用 `signed_in_user` 方法了，因此我们就可以使用代码 10.25 中的事前过滤器来限制访问 `create` 和 `destroy` 动作了。（因为我们没有使用命令行生成 Microposts 控制器文件，因此需要手动创建。）

代码 10.25： 在 Microposts 控制器中加入访问限制功能
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  before_action :signed_in_user

  def create
  end

  def destroy
  end
end
```

注意，我们没有明确指定事前过滤器要限制的动作有哪几个，因为默认情况下仅有的两个动作都会被限制。如果我们要加入第三个动作，例如 `index` 动作，未登录的用户可以访问，那么我们就要明确的指定要限制的动作了：

```
class MicropostsController < ApplicationController
  before_action :signed_in_user, only: [:create, :destroy]

  def index
  end

  def create
  end

  def destroy
  end
end
```

现在，测试应该可以通过了：

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb
```

10.3.2 创建微博

在第 7 章中，我们实现了用户注册功能，方法是使用 HTML 表单向 Users 控制器的 `create` 动作发送 POST 请求。创建微博的功能实现起来是类似的，最大的不同点在于，表单不是放在单独的页面 /microposts/new 中，而是在网站的首页（模仿 Twitter，在根地址 /），构思图如图 10.10 所示。

上一次接触首页时，是图 5.6 那个样子，在页面中部有个“Sign up now!”链接按钮。因为创建微博的表单只对登录后的用户有用，所以本节的目标之一就是根据用户的登录状态显示不同的首页内容，如代码 10.28 所示，不过在此之前，我们可以先编写测试。和用户资源一样，我们要使用集成测试：

```
$ rails generate integration_test micropost_page
```

图 10.10：包含创建微博表单的首页构思图

对创建微博功能的测试和对创建用户功能的测试（参见代码 7.16）类似，如代码 10.26 所示：

代码 10.26：对创建微博功能的测试
spec/requests/micropost_pages_spec.rb

```
require 'spec_helper'

describe "Micropost pages" do

  subject { page }

  let(:user) { FactoryGirl.create(:user) }
  before { sign_in user }

  describe "micropost creation" do
    before { visit root_path }
```

```
    describe "with invalid information" do

      it "should not create a micropost" do
        expect { click_button "Post" }.not_to change(Micropost, :count)
      end

      describe "error messages" do
        before { click_button "Post" }
        it { should have_content('error') }
      end
    end

    describe "with valid information" do

      before { fill_in 'micropost_content', with: "Lorem ipsum" }
      it "should create a micropost" do
        expect { click_button "Post" }.to change(Micropost, :count).by(1)
      end
    end
  end
end
```

下面我们来编写 Microposts 控制器 `create` 动作的代码，和 Users 控制器的有点类似（参见代码 7.26），主要的区别是，创建微博时，要使用用户和微博的关联关系来构建微博对象，如代码 10.27 所示。

代码 10.27： Microposts 控制器的 `create` 动作
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  before_action :signed_in_user

  def create
    @micropost = current_user.microposts.build(micropost_params)
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_url
    else
      render 'static_pages/home'
    end
  end

  def destroy
  end

  private
```

```
    def micropost_params
      params.require(:micropost).permit(:content)
    end
end
```

我们使用代码 10.28 来构建创建微博所需的表单，这个视图会根据用户的登录状态显示不同的 HTML 内容。

代码 10.28： 在首页中加入创建微博所需的表单
app/views/static_pages/home.html.erb

```
<% if signed_in? %>
  <div class="row">
    <aside class="span4">
      <section>
        <%= render 'shared/user_info' %>
      </section>
      <section>
        <%= render 'shared/micropost_form' %>
      </section>
    </aside>
  </div>
<% else %>
  <div class="center hero-unit">
    <h1>Welcome to the Sample App</h1>

    <h2>
      This is the home page for the
      <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </h2>

    <%= link_to "Sign up now!", signup_path,
                                class: "btn btn-large btn-primary" %>
  </div>

  <%= link_to image_tag("rails.png", alt: "Rails"), 'http://rubyonrails.org/' %>
<% end %>
```

上述代码中 `if-else` 条件语句的各分支包含的代码太多，有点乱，在练习中（10.5 节）我们会进行优化，把分支中的代码放入局部视图中。不过代码 10.28 中用到的局部视图不会留作练习，现在就来创建：用户信息侧边栏局部视图如代码 10.29 所示，创建微博的表单局部视图如代码 10.30 所示。

代码 10.29： 用户信息侧边栏局部视图
app/views/shared/_user_info.html.erb

```
<a href="<%= user_path(current_user) %>">
  <%= gravatar_for current_user, size: 52 %>
</a>
<h1>
  <%= current_user.name %>
</h1>
<span>
  <%= link_to "view my profile", current_user %>
</span>
<span>
  <%= pluralize(current_user.microposts.count, "micropost") %>
</span>
```

和代码 9.24 一样，代码 10.29 使用了代码 7.30 中定义的 `gravatar_for` 帮助方法。

注意，和用户资料页面的侧边栏类似（参见代码 10.20），代码 10.29 中的用户信息也显示了用户发布的微博数量。不过显示上有细微的差别，在用户资料页面的侧边栏中，“Microposts” 是作为标签使用的，所以“Microposts (1)”这样的用法是合理的。而在本例中，如果说“1 microposts”的话就有点不合语法了，所以我们调用了 `pluralize` 方法，显示成“1 micropost”，“2 microposts”等。

下面我们来编写创建微博表单的局部视图，如代码 10.30 所示，和代码 7.17 中的注册表单是类似的。

代码 10.30：创建微博表单局部视图
`app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-large btn-primary" %>
<% end %>
```

如果要代码 10.30 中的表单可用，我们还要做两件事。第一，（和之前一样）我们要通过关联关系定义 `@micropost` 变量：

```
@micropost = current_user.microposts.build
```

写入控制器后如代码 10.31 所示。

代码 10.31：在 home 动作中定义 `@micropost` 实例变量
`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

  def home
    @micropost = current_user.microposts.build if signed_in?
  end
```

```
  .
  .
  .
end
```

代码 10.31 中的代码有一个好处，如果我们忘记加入要求用户登录的限制，相应的测试会失败。

我们要做的第二件事是，重写错误提示信息局部视图，让 `<%= render 'shared/error_messages', object: f.object %>` 这行代码能正确运行。你可能还记得，在代码 7.23 中，错误提示信息的局部视图代码直接引用了 `@user` 变量，但现在我们提供的变量却是 `@micropost`。因此，我们要编写一个新的错误提示信息局部视图，不管传入的是什么变量，都能正常使用。幸好，表单的块变量 `f` 可以通过 `f.object` 获取当前的对象，因此，在 `form_for(@user) do |f|` 中，`f.object` 的返回值是 `@user`；在 `form_for(@micropost) do |f|` 中，`f.object` 的返回值是 `@micropost`。

我们要通过一个 Hash 把对象传入局部视图，其值是该对象，键是局部视图中所需的变量名称，如下面的代码所示：

```
<%= render 'shared/error_messages', object: f.object %>
```

换句话说，`object: f.object` 会创建一个名为 `object` 的变量，供 `error_messages` 局部视图使用。利用这个对象，我们就可以编写不同的错误提示信息，如代码 10.32 所示。

代码 10.32：修改代码 7.23 中的错误提示信息局部视图，如果传入其他对象也可以使用
`app/views/shared/_error_messages.html.erb`

```
<% if object.errors.any? %>
  <div id="error_explanation">
    <div class="alert alert-error">
      The form contains <%= pluralize(object.errors.count, "error") %>.
    </div>
    <ul>
    <% object.errors.full_messages.each do |msg| %>
      <li>* <%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

现在，代码 10.26 中的测试应该可以通过了：

```
$ bundle exec rspec spec/requests/micropost_pages_spec.rb
```

不过，Users 资源相关视图的 request spec 测试却失败了，因为注册和编辑用户的表单使用的仍是旧的错误提示信息局部视图。要想修正这个错误，就要换用新的局部视图，如代码 10.33 和代码 10.34 所示。（注意：如果你加入了 9.6 节练习中的代码 9.49 和代码 9.50，那么要修改的代码有点不一样，需要做适当的修改。）

代码 10.33：修改用户注册表单的错误提示信息局部视图

app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>
      .
      .
      .
    <% end %>
  </div>
</div>
```

图 **10.11**：显示有创建微博表单的首页（/）

代码 **10.34**：修改编辑用户表单的错误提示信息局部视图
app/views/users/edit.html.erb

```
<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>
```

```
<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>
      .
      .
      .
    <% end %>

    <%= gravatar_for(@user) %>
    <a href="http://gravatar.com/emails">change</a>
  </div>
</div>
```

现在，所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

而且，本节添加的所有 HTML 代码也都能正确渲染：图 10.11 中显示了创建微博的表单，图 10.12 中显示了有一个错误提示信息的表单。现在你可以自己发布一篇微博试一下一切是否都可以正常运行，不过最好还是在 10.3.3 节之后再试吧。

图 10.12：显示有错误提示信息的首页

10.3.3 临时的动态列表

10.3.2 节遗留了一个功能没有实现：首页没有显示动态。如果你愿意的话，可以在图 10.11 所示的表单中发表一篇合法的微博，然后转到用户资料页面，验证一下这个表单是否可以正常使用。这样在页面之间来来回回是有点麻烦的，如果能在首页显示一个含有当前登入用户的微博列表就好了，构思图如图 10.13 所示。（在第 11 章中，我们会在这个微博列表中加入当前登入用户所关注用户的微博。）

图 10.13：显示有临时动态列表的首页构思图

因为每个用户都应该有一个动态列表，因此我们可以在 User 模型中定义一个名为 `feed` 的方法。等功能全部实现后，我们会测试微博列表中是否包含了所关注用户的微博，不过现在我们只需要测试 `feed` 方法的返回值是否只有当前登入用户的微博，而没有其他用户的微博。测试所需的代码如代码 10.35 所示。

代码 10.35：对临时动态列表的测试
`spec/models/user_spec.rb`

```
require 'spec_helper'
```

```
describe User do
  .
  .
  .
  it { should respond_to(:microposts) }
  it { should respond_to(:feed) }
  .
  .
  .
  describe "micropost associations" do

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end
    .
    .
    .
    describe "status" do
      let(:unfollowed_post) do
        FactoryGirl.create(:micropost, user: FactoryGirl.create(:user))
      end

      its(:feed) { should include(newer_micropost) }
      its(:feed) { should include(older_micropost) }
      its(:feed) { should_not include(unfollowed_post) }
    end
  end
end
```

上述代码，在数组上调用了 `include?` 方法（基于 RSpec 对布尔值方法的约定），该方法的作用是检查数组中是否包含指定的元素：[9]

```
$ rails console
>> a = [1, "foo", :bar]
>> a.include?("foo")
=> true
>> a.include?(:bar)
=> true
```

9. 我之所以在 1.1.1 节中推荐读者在读完本书后阅读一本纯介绍 Ruby 的书，就是为了学习使用类似 `include?` 这样的方法。

```
>> a.include?("baz")
=> false
```

这段代码说明了 RSpec 对布尔值方法的约定是多么的灵活，虽然 `include` 是 Ruby 语言的关键字（用来引入模块，在代码 8.14 中有用到），但在当前语境中，RSpec 依然能正确的识别我们的意图是要测试数组是否包含了指定的元素。

在编写 `feed` 方法时，我们需要从数据库中取出所有 `user_id` 等于当前用户 id 的微博，要在 `Micropost` 对象上调用 `where` 方法，如代码 10.36 所示。[10]

代码 10.36： 动态列表的初步实现
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def feed
    # This is preliminary. See "Following users" for the full implementation.
    Micropost.where("user_id = ?", id)
  end
  .
  .
  .
end
```

`Micropost.where("user_id = ?", id)` 中的问号可以确保 `id` 的值在传入底层的 SQL 查询语句之前做了适当的转义，避免“SQL 注入”这种严重的安全隐患。这里用到的 `id` 属性是个整数，没什么危险，不过在 SQL 语句中引入变量之前做转义是个好习惯。

细心的读者可能已经注意到了，现阶段代码 10.36 中的代码和下面的定义作用是一样的：

```
def feed
  microposts
end
```

我们之所以使用了代码 10.39 的版本，是因为它能更好的服务于第 11 章中实现的完整的动态列表。

要测试动态列表的显示，我们首先要创建一些微博，然后检测各微博在页面中是否均包含在列表元素 `li` 中，如代码 10.37 所示。

代码 10.37： 测试首页显示的微博列表
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do
```

10. 阅读 Rails 指南中的《Active Record Query Interface》一文，更深入的学习 `where` 等方法的用法。

```
  subject { page }

  describe "Home page" do
    .
    .
    .
    describe "for signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        FactoryGirl.create(:micropost, user: user, content: "Lorem ipsum")
        FactoryGirl.create(:micropost, user: user, content: "Dolor sit amet")
        sign_in user
        visit root_path
      end

      it "should render the user's feed" do
        user.feed.each do |item|
          expect(page).to have_selector("li##{item.id}", text: item.content)
        end
      end
    end
  end
  .
  .
  .
end
```

代码 10.37 假设所显示的每篇微博都有唯一的 CSS id，所以如下代码

```
expect(page).to have_selector("li##{item.id}", text: item.content)
```

会为每个微博生成一个匹配器。（注意，`li##{item.id}` 中的第一个 # 是 Capybara 中的对应 CSS id 的句法，而第二个 # 则代表 Ruby 字符串插值操作 `#{}` 的开始。）

要在示例程序中使用动态列表，我们可以在 home 动作中定义一个 `@feed_items` 实例变量，如代码 10.38 所示。然后再在首页（参见代码 10.41）中加入一个动态列表局部视图（参见代码 10.39）。（对分页的测试会留作练习，参见 10.5 节。）

代码 10.38： 在 home 动作中加入一个实例变量
`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

  def home
    if signed_in?
```

```
      @micropost  = current_user.microposts.build
      @feed_items = current_user.feed.paginate(page: params[:page])
    end
  end
  .
  .
  .
end
```

代码 10.39：动态列表局部视图
app/views/shared/_feed.html.erb

```
<% if @feed_items.any? %>
  <ol class="microposts">
    <%= render partial: 'shared/feed_item', collection: @feed_items %>
  </ol>
  <%= will_paginate @feed_items %>
<% end %>
```

动态列表局部视图使用如下代码，把列表项目的渲染支配给了 `feed_item` 局部视图：

```
<%= render partial: 'shared/feed_item', collection: @feed_items %>
```

我们传入了一个名为 `:collection` 的参数，其值为 `@feed_items`，这样 `render` 方法就会在指定的局部视图中（本例中的 `feed_item`）使用这个集合中的数据渲染各项目。（之前，在渲染局部视图时，我们都省略了 `:partial` 参数，例如 `render 'shared/micropost'`，不过如果指定了 `:collection` 参数，这种省略的用法就是错误的。）列表项目局部视图的代码如代码 10.40 所示。

代码 10.40：渲染单个动态列表项目的局部视图
app/views/shared/_feed_item.html.erb

```
<li id="<%= feed_item.id %>">
  <%= link_to gravatar_for(feed_item.user), feed_item.user %>
  <span class="user">
    <%= link_to feed_item.user.name, feed_item.user %>
  </span>
  <span class="content"><%= feed_item.content %></span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(feed_item.created_at) %> ago.
  </span>
</li>
```

代码 10.40 还使用如下的代码为每个动态项目指定了 CSS id：

```
<li id="<%= feed_item.id %>">
```

这正是代码 10.37 中的测试所要检测的。

和之前一样，我们可以把动态列表局部视图加入首页来显示动态，如代码 10.41 所示。加入后的效果就是在首页显示了动态列表，和预期一样，如图 10.14 所示。

代码 10.41： 在首页中加入动态列表
app/views/static_pages/home.html.erb

```
<% if signed_in? %>
  <div class="row">
    .
    .
    .
    <div class="span8">
      <h3>Micropost Feed</h3>
      <%= render 'shared/feed' %>
    </div>
  </div>
<% else %>
  .
  .
  .
<% end %>
```

图 10.14： 显示了临时动态列表的首页（/）

现在，发布新微博的功能可以按照预期正常工作了，如图 10.15 所示。不过还有个小小不足：如果发布微博失败，首页还会需要一个名为 `@feed_items` 的实例变量，所以提交失败时网站就无法正常运行了（你也可以运行测试来验证一下）。最简单的解决方法是，如果提交失败就把 `@feed_items` 设为空数组，如代码 10.42 所示。[11]

图 10.15：发布新微博后的首页

代码 10.42：在 `create` 动作加入一个空的 `@feed_items` 实例变量
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  .
  .
  .
  def create
    @micropost = current_user.microposts.build(micropost_params)
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_url
    else
      @feed_items = []
      render 'static_pages/home'
    end
```

11. 很不幸，这样的话分页就不可用了。你可以加入分页，点击分页链接看一下是为什么。

```
  end
  .
  .
  .
end
```

至此，临时的动态列表可以正常工作了，测试组件也可以通过了：

```
$ bundle exec rspec spec/
```

10.3.4 删除微博

我们要为 Microposts 资源实现的最后一个功能是删除微博。和删除用户类似（参见 9.4.2 节），删除微博也是通过“delete”链接实现的，构思图如图 10.16 所示。删除用户要限制只有管理员才能进行，而删除微博的链接只对微博的发布者可用。

图 10.16： 显示有删除链接的临时动态列表构思图

第一步，我们要在微博局部视图（代码 10.21）中加入删除链接，同时还要在动态列表项目的局部视图（代码 10.40）中加入类似的链接，结果如代码 10.43 和代码 10.44 所示。（这两个局部视图基本上是一样的，消除代码重复会留作练习，参见 10.5 节。）

代码 10.43： 在微博局部视图中加入删除链接
app/views/microposts/_micropost.html.erb

```
<li>
  <span class="content">><%= micropost.content %></span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
  </span>
  <% if current_user?(micropost.user) %>
    <%= link_to "delete", micropost, method: :delete,
                                     data: { confirm: "You sure?" },
                                     title: micropost.content %>
  <% end %>
</li>
```

代码 10.44： 在动态列表项目局部视图中加入删除链接
app/views/shared/_feed_item.html.erb

```
<li id="<%= feed_item.id %>">
  <%= link_to gravatar_for(feed_item.user), feed_item.user %>
    <span class="user">
      <%= link_to feed_item.user.name, feed_item.user %>
    </span>
    <span class="content">><%= feed_item.content %></span>
    <span class="timestamp">
      Posted <%= time_ago_in_words(feed_item.created_at) %> ago.
    </span>
  <% if current_user?(feed_item.user) %>
    <%= link_to "delete", feed_item, method: :delete,
                                     data: { confirm: "You sure?" },
                                     title: feed_item.content %>
  <% end %>
</li>
```

对删除微博的测试要用 Capybara 来点击“delete”链接，然后检测微博的数量是否减少了一个，如代码 10.45 所示。

代码 10.45： 针对 Microposts 控制器 `destroy` 动作的测试
spec/requests/micropost_pages_spec.rb

```
require 'spec_helper'

describe "Micropost pages" do
  .
  .
  .
```

```
  describe "micropost destruction" do
    before { FactoryGirl.create(:micropost, user: user) }

    describe "as correct user" do
      before { visit root_path }

      it "should delete a micropost" do
        expect { click_link "delete" }.to change(Micropost, :count).by(-1)
      end
    end
  end
end
```

程序所需的代码还是和代码 9.46 中删除用户时类似，两种情况的主要不同之处是，删除微博不使用 `admin_user` 事前过滤器了，而是用 `correct_user` 事前过滤器，检查当前用户是否发布了指定 id 的微博，如代码 10.46 所示。图 10.17 显示的是删除了倒数第二个微博后的页面。

代码 10.46： Microposts 控制器的 `destroy` 动作
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  before_action :signed_in_user, only: [:create, :destroy]
  before_action :correct_user,   only: :destroy
  .
  .
  .
  def destroy
    @micropost.destroy
    redirect_to root_url
  end

  private

    def micropost_params
      params.require(:micropost).permit(:content)
    end

    def correct_user
      @micropost = current_user.microposts.find_by(id: params[:id])
      redirect_to root_url if @micropost.nil?
    end
end
```

注意，在 `correct_user` 事前过滤器中，我们是通过关联关系查寻微博的：

```
current_user.microposts.find_by_id(params[:id])
```

这行代码可以自行确保只在当前用户发布的微博中查询。这里，我们使用的是 `find_by`，而没用 `find`，因为如果没有找到微博 `find` 会抛出异常，而不会返回 `nil`。顺便说一下，如果你习惯处理 Ruby 异常，也可以按照下面的方式定义 `correct_user` 过滤器：

```
def correct_user
  @micropost = current_user.microposts.find(params[:id])
rescue
  redirect_to root_url
end
```

你可能会说，我们也可以直接使用 `Micropost` 对象来实现 `correct_user` 过滤器，就像下面这样：

```
@micropost = Micropost.find_by(id: params[:id])
redirect_to root_url unless current_user?(@micropost.user)
```

这种实现方式基本上和代码 10.46 是一样的，不过，Wolfram Arnold 在《Access Control 101 in Rails and the Citibank Hack》一文中解释过，安全起见，最好还是通过关联来查找。

图 10.17：删除了倒数第二篇微博后的首页

加入了本节的代码后，我们的 Micropost 模型以及用户界面就都完成了，测试组件也可以通过了：

```
$ bundle exec rspec spec/
```

10.4 小结

加入了 Microposts 资源后，我们的示例程序基本上完成了。还没实现的部分是社交功能，让用户之间可以相互关注。在第 11 章中我们会学习如何实现这种关联关系，还要实现一个真正的动态列表。

如果你使用 Git 做版本控制的话，在继续之前，先提交改动，然后再合并到主分支：

```
$ git add .
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

现在你也可以把程序推送到 Heroku 上。因为加入 `microposts` 表改动了数据模型，因此你需要在“生产数据库”中执行迁移操作：

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:populate
```

10.5 练习

到目前为止，我们已经实现了很多功能，在现有功能的基础上我们还可以实现更多的功能，下面就是一些例子。

1. 添加测试，检测侧边栏中微博的数量是否正确显示了，要包含对单复数的检查。
2. 添加测试，检测微博的分页功能。
3. 重构首页的视图文件，把 `if-else` 语句的两个分支分别放到单独的局部视图中。
4. 编写一个测试，确保不是当前用户发布的微博下方不会显示删除链接。
5. 使用局部视图去掉代码 10.43 和代码 10.44 中的重复。
6. 现在，特别长的单词会撑破布局，如图 10.18 所示。使用代码 10.47 中定义的 `wrap` 帮助方法修正这个错误。注意，其中用到的 `raw` 方法是为了避免 Rails 转义 HTML 代码，`sanitize` 方法是为了防止跨站脚本攻击。这段代码还用到了看起来很奇怪但是很实用的三元操作符（ternary operator，参见旁注 10.1 的说明）。
7. （**附加题**）在首页中添加一段 JavaScript 代码，当在发布微博的表单中输入文字时，显示输入的内容与 140 个字符的差值。

旁注 10.1：三种人

世界上有三种人：一种喜欢用三元操作符，一种不喜欢，还有一种甚至不知道三元操作符是什么。（如果你不幸是第三种人的话，看过下面的说明就不再归属其中了。）

在编写很多代码之后，你会发现最常用到的流程控制是类似下面这种：

```
if boolean?
  do_one_thing
else
  do_something_else
end
```

Ruby 和其他很多语言一样（包括 C/C++，Perl，PHP 和 Java），提供了一种更为简单的表达式来替换这种流程控制结构——三元操作符（之所以起了这个名字，是因为三元操作符涉及三个部分）：

```
boolean? ? do_one_thing : do_something_else
```

三元操作符甚至可以用来替代赋值操作：

```
if boolean?
  var = foo
else
  var = bar
end
```

可以改写成

```
var = boolean? ? foo : bar
```

另外一个经常使用地方是在函数的返回值中：

```
def foo
  do_stuff
  boolean? ? "bar" : "baz"
end
```

因为 Ruby 函数的默认返回值是定义体中的最后一个表达式，所以 `foo` 方法的返回值会根据 `boolean?` 的结果而不同，不是 `"bar"` 就是 `"baz"`。代码 10.47 中就用到了这种结构。

代码 10.47：换行显示长单词的帮助方法
`app/helpers/microposts_helper.rb`

```
module MicropostsHelper

  def wrap(content)
    sanitize(raw(content.split.map{ |s| wrap_long_string(s) }.join(' ')))
  end

  private

    def wrap_long_string(text, max_width = 30)
      zero_width_space = "&#8203;"
      regex = /.{1,#{max_width}}/
      (text.length < max_width) ? text :
                                  text.scan(regex).join(zero_width_space)
    end
end
```

图 10.18：特别长的单词撑破了网站的布局

此页留白

第 11 章 关注用户

在这一章中，我们会在现有程序的基础上增加社交功能，允许用户关注（follow）或取消关注其他人，并在用户主页上显示其所关注用户的微博更新。我们还会创建两个页面用来显示关注的用户列表和粉丝列表。我们将会在 11.1 节学习如何构建用户之间的模型关系，随后在 11.2 节设计网页界面，同时还会介绍 Ajax。最后，我们会在 11.3 节实现一个完整的动态列表。

这是本书最后一章，其中会包含了一些本教程中最具有挑战性的内容，为了实现动态列表，我们会使用一些 Ruby/SQL 小技巧。 通过这些例子，你会了解到 Rails 是如何处理更加复杂的数据模型的，而这些知识会在你日后开发其他应用时发挥作用。 为了帮助你平稳地从教程学习过渡到独立开发，在 11.4 节我们推荐了几个可以在已有微博核心基础上开发的额外功能，以及一些进阶资料的链接。

和之前章节一样，Git 用户应该创建一个新的分支：

```
$ git checkout -b following-users
```

因为本章的内容比较有挑战性，在开始编写代码之前，我们先来思考一下网站的界面。 和之前的章节一样，在开发的早期阶段我们会通过构思图来呈现页面。[1]完整的页面流程是这样的：一名用户 (John Calvin) 从他的个人资料页面（如图 11.1 ）浏览到用户索引页面（如图 11.2），关注了一个用户。Calvin 打开另一个用户 Thomas Hobbes 的个人主页（如图 11.3），点击“Follow（关注）”按钮 关注该用户，然后，这个“Follow”按钮会变为“Unfollow（取消关注）”，而且 Hobbes 的关注人数会增加 1 个（如图 11.4）。接着，Calvin 回到自己的主页，看到关注人数增加了 1 个，在动态列表中也能看到 Hobbes 的状态更新（如图 11.5）。接下来的整章内容就是要实现这样的页面流程。

11.1 关系模型

为了实现关注用户这一功能，第一步我们要做的是创建一个看上去并不是那么直观的数据模型。一开始我们可能会认为一个 `has_many` 的数据关系能满足我们的要求：一个用户可以关注多个用户，同时一个用户还能被多个用户关注。但实际上这种关系是存在问题的，下面我们就来学习如何使用 `has_many through` 来解决这个问题。本节很多功能的实现初看起来都有点难以理解，你需要花一点时间思考，才能真正搞清楚这样做的原因。如果在某个地方卡住了，尝试着先往后读，然后再把本节读一遍，看一下刚才卡住的地方想明白了没。

11.1.1 数据模型带来的问题以及解决方式

构造数据模型的第一步，我们先来看一个典型的情况。假如一个用户关注了另外一个用户，比如 Calvin 关注了 Hobbes，也就是 Hobbes 被 Calvin 关注了，那么 Calvin 就是关注者（follower），而 Hobbes 则是被关注者(followed)。按照 Rails 默认的复数表示习惯， 我们称关注某一特定用户的用户集合为该用户的 followers，

1. 构思图中的头像来自 http://www.flickr.com/photos/john_lustig/2518452221 和 http://www.flickr.com/photos/30775272@N05/2884963755

`user.followers` 就是这些用户组成的数组。不过，当我们颠倒一下顺序，上述关系则不成立了：默认情况下，所有被关注的用户称为 followeds，这样说在英语语法上并不通顺恰当。我们可以称被关注者为 following，但这个词有些歧义：在英语里，”following” 指关注你的人，和我们想表达的恰恰相反。考虑到上述两种情况，尽管我们将使用“following”作为标签，如“50 following, 75 followers”, 但在数据模型中会使用“followed users”表示我们关注的用户集合，以及一个对应的 `user.followed_users` 数组。[2]

图 11.1：当前登入用户的个人资料页面

经过上述的讨论，我们会按照图 11.6 的方式构建被关注用户的模型，使用 `followed_users` 表实现一对多 (`has_many`)关联。由于 `user.followed_users` 应该是一个用户对象组成的数组，所以 `followed_users` 表中的每一行应该对应一个用户，并且指定 `followed_id` 列，和其他用户建立关联。[3]除此之外，由于每一行均对应一个用户，所以我们还要在表中加入用户的其他属性，包括名字，密码等。

图 11.6 中描述的数据模型仍存在一个问题，那就是存在非常多的冗余，每一行不仅包括了所关注用户的 id, 还包括了他们的其他信息，而这些信息在 `users` 表中都有。 更糟糕的是，为了建立关注用户的数据模型，我们还需要一

2. 在本书第一版中，使用了 `user.following`， 但我发现有时读起来感觉怪怪的。感谢读者 Cosmo Lee 说服我修改这个表述，并建议我如何使其理解起来更容易些。(不过我并没有完全采纳他的建议，所以如果你阅读时仍感到迷惑请不要怪他。)
3. 为了简单清晰，图 11.6 中没有显示 `followed_users` 表的 id 列

个单独的，同样冗余的 followers 表。这最终将导致数据模型极难维护，每当用户修改姓名时，我们不仅要修改用户在 users 表中的数据，还要修改 followed_users 和 followers 表中对应该用户的每一个记录。

图 11.2：寻找一个用户来关注

造成这个问题的主要原因是，我们缺少了一层抽象。 找到合适抽象的一个方法是，思考我们会如何在应用程序中实现关注用户的操作。在 7.1.2 节中我们介绍过，REST 架构涉及到创建资源和销毁资源两个过程。 由此引出两个问题： 当用户关注另一个用户时，创建了什么？ 当用户取消关注另一个用户是，销毁了什么？

按照 REST 架构的思路再次思考之后，我们会发现，在关注用户的过程中，被创建和被销毁的是两个用户之间的“关系”。在这种“关系”中，一个用户有多个“关系”（has_many :relationships），并有很多关注的用户（followed_users 或 followers)。其实，在图 11.6 中我们已经基本实现了这种“关系”： 由于每一个被关注的用户都是由 followed_id 独一无二的标识出来的，我们就可以将 followed_users 表转化成 relationships 表，删掉用户的详细资料，使用 followed_id 从 users 表中获得被关注用户的数据。 同样的，这种“关系”反过来，我们可以使用 follower_id 获取所有粉丝组成的数组。

为了得到一个由所有被关注用户组成的 followed_users 数组，我们可以先获取由 followed_id 属性组成的数组，再查找每个用户。不过，如你所想，Rails 为我们提供了一种更简单的方式，那就是 has_many through。我们将在 11.1.4 节介绍， Rails 允许我们使用下面这行清晰简洁的代码，通过 relationships 表来描述一个用户关注了很多其他用户：

```
has_many :followed_users, through: :relationships, source: "followed_id"
```

这行代码会自动获取被关注用户组成的数组，也就是 `user.followed_users`。 图11.7 描述了这个数据模型。

图 11.3：一个想要关注的用户资料页面，显示有关注按钮

下面让我们动手实现，首先我们通过下面的命令创建 Relationship 模型：

```
$ rails generate model Relationship follower_id:integer followed_id:integer
```

这个命令可能会生成 Relationship 预构件，你应该将其删除：

```
$ rm -f spec/factories/relationship.rb
```

由于我们需通过 `follower_id` 和 `followed_id` 来查找用户之间的关系，考虑到性能，要为这两列加上索引， 如代码 11.1 所示。

代码 11.1：在 `relationships` 表中设置索引
`db/migrate/[timestamp]_create_relationships.rb`

```
class CreateRelationships < ActiveRecord::Migration
  def change
    create_table :relationships do |t|
      t.integer :follower_id
      t.integer :followed_id

      t.timestamps
    end
    add_index :relationships, :follower_id
    add_index :relationships, :followed_id
    add_index :relationships, [:follower_id, :followed_id], unique: true
  end
end
```

图 11.4：关注按钮变为取消关注的同时，关注人数增加了 1 个

在代码 11.1 中，我们还设置了一个组合索引（composite index），其目的是确保 (`follower_id, followed_id`) 组合是唯一的，这样用户就无法多次关注同一个用户了 (可以和代码 6.19 中为保持 Email 地址唯一的 index 做比较一下)：

```
add_index :relationships, [:follower_id, :followed_id], unique: true
```

从 11.1.4 节开始，我们会发现，在用户界面中这样的事情是不会发生的，但是添加了组合索引后，如果用户试图二次关注时，程序会抛出异常（例如，使用像 `curl` 这样的命令行程序）。我们也可以在 Relationship 模型中添加唯一性数据验证，但因为每次尝试创建一个重复关系时都会触发错误，所以这个组合索引现在已经能满足我们的需求了。

图 **11.5**：个人主页出现了新关注用户的微博，关注人数增加了 1 个

图 **11.6**：一个简单的用户互相关注实现

图 **11.7**：通过 `relationships` 表建立的被关注用户数据模型

为了创建 `relationships` 表，和之前一样，我们要先执行数据库迁移，再准备好“测试数据库”：

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

得到的 Relationship 数据模型如图 11.8 所示。

relationships	
id	integer
follower_id	integer
followed_id	integer
created_at	datetime
updated_at	datetime

图 **11.8**：Relationship 数据模型

11.1.2 User 和 Relationship 模型之间的关联

在着手实现已关注用户和关注者之前，我们先要建立 User 和 Relationship 模型之间的关联关系。一个用户可以有多个“关系”（`has_many` relationships），因此一个“关系”涉及到两个用户，所以这个“关系”就同时属于（`belongs_to`）该用户和被关注者。

和 10.1.3 节创建微博一样，我们将通过 User 和 Relationship 模型之间的关联来创建这个“关系”，使用如下的代码实现：

```
user.relationships.build(followed_id: ...)
```

首先，我们来编写测试，如代码 11.2 所示。

代码 11.2：测试建立“关系”以及属性的可访问性

spec/models/relationship_spec.rb

```
require 'spec_helper'

describe Relationship do

  let(:follower) { FactoryGirl.create(:user) }
  let(:followed) { FactoryGirl.create(:user) }
  let(:relationship) { follower.relationships.build(followed_id: followed.id) }

  subject { relationship }

  it { should be_valid }
end
```

这里需要注意，与测试 User 和 Micropost 模型时使用 `@user` 和 `@micropost` 不同，代码 11.2 中使用 `let` 代替了实例变量。这两种方式之间几乎没有差别[4]，但我认为使用 `let` 相对于使用实例变量更易懂。测试 User 和 Micropost 时之所以使用实例变量，是希望读者早些接触这个重要的概念，而 `let` 则略显高深，所以我们放在这里才用。

同时，在 User 模型中我们还要测试用户对象是否可以响应 `relationships` 方法，如代码 11.3 所示。

代码 11.3：测试 `user.relationships`
spec/models/user_spec.rb

```
require 'spec_helper'
describe User do
  .
  .
  .
  it { should respond_to(:feed) }
  it { should respond_to(:relationships) }
  .
  .
  .
end
```

此时，你可能会想在程序中中加入类似于 10.1.3 节中用到的代码，我们要添加的代码确实很像，但二者之间有一处很不一样：在 Micropost 模型中， 我们使用

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

4. 请在 Stack Overflow 网站上阅读关于“何时应使用 let 方法”的讨论来了解更多内容

和

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

因为 microposts 表中存有 user_id 属性，可以标示用户（参见 10.1.1 节）。这种连接两个数据表的 id，我们称之为外键（foreign key），当指向 User 模型的外键为 user_id 时，Rails 就会自动的获知关联关系，因为默认情况下，Rails 会寻找 <class>_id 形式的外键，其中 <class> 是模型类名的小写形式。[5]现在，尽管我们处理的还是用户，但外键是 follower_id 了，所以我们要告诉 Rails 这一变化，如代码 11.4 所示。[6]

代码 11.4： 实现 User 和 Relationship 模型之间 has_many 的关联关系
app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  has_many :relationships, foreign_key: "follower_id", dependent: :destroy
  .
  .
  .
end
```

（由于删除用户后，也应该删除该用户的所有“关系”， 于是我们指定了 dependent: :destroy 参数；针对删除效果的测试会留作练习，参见 11.5 节。）

和 Micropost 模型一样，Relationship 模型和 User 模型之间也有一层 belongs_to 关系，此时，这种关系同时属于关注者和被关注者，针对这层“关系”的测试如代码 11.5 所示。

代码 11.5： 测试 User 和 Relationship 模型之间的 belongs_to 关系
spec/models/relationship_spec.rb

```
describe Relationship do
  .
  .
  .
  describe "follower methods" do
    it { should respond_to(:follower) }
    it { should respond_to(:followed) }
    its(:follower) { should eq follower }
    its(:followed) { should eq followed }
```

5. 严格来说，Rails 是使用 underscore 方法把类名转换为 id 的。例如，"Foobar".underscore 的结果是 "foo_bar"，所以 Foobar 对象的外键是 foo_bar_id。（顺便说一下，underscore 的逆操作是 camelize，这个方法会把 "camel_case" 转换成 "CamelCase"。）
6. 如果你注意到 followed_id 同样可以标识用户，并且担心这个解决方法会造成被关注者和粉丝之间存在非对称的关系，你已经想到我们的前面了，我们会在 11.1.5 节解决这个问题。

```
  end
end
```

下面我们开始写程序的代码，`belongs_to` 关系的建立和之前一样。Rails 会通过 Symbol 获知外键的名字（例如，`:follower` 对应的外键是 `follower_id`，`:followed` 对应的外键是 `followed_id`），但 Followed 或 Follower 模型是不存在的，因此这里就要使用 `User` 这个类名， 如代码 11.6 所示。注意，与默认生成的 Relationship 模型不同，这里只有 `followed_id` 是可以访问的。

代码 11.6：为 Relationship 模型添加 `belongs_to` 关系
`app/models/relationship.rb`

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end
```

尽管直到 11.1.5 节我们才会用到 `followed` 关联，但同时实现 follower 和 followed 关联会更容易理解。

此时，代码 11.2 和代码 11.3 中的测试应该可以通过了。

```
$ bundle exec rspec spec/
```

11.1.3 数据验证

在结束这部分之前，我们将添加一些针对 Relationship 模型的数据验证，确保代码的完整性。测试（代码 11.7）和程序代码（代码 11.8）都非常易懂。

代码 11.7：测试 Relationship 模型的数据验证
`spec/models/relationship_spec.rb`

```
describe Relationship do
  .
  .
  .
  describe "when followed id is not present" do
    before { relationship.followed_id = nil }
    it { should_not be_valid }
  end

  describe "when follower id is not present" do
    before { relationship.follower_id = nil }
    it { should_not be_valid }
  end
end
```

代码 11.8：添加 Relationship 模型数据验证
`app/models/relationship.rb`

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
  validates :follower_id, presence: true
  validates :followed_id, presence: true
end
```

11.1.4 被关注的用户

下面到了 Relationship 关联关系的核心部分了，获取 `followed_users` 和 `followers`。 我们首先从 `followed_users` 开始，测试如代码 11.9 所示。

代码 11.9：测试 `user.followed_users` 属性
`spec/models/user_spec.rb`

```
require 'spec helper'
describe User do
  .
  .
  .
  it { should respond_to(:relationships) }
  it { should respond_to(:followed_users) }
  .
  .
  .
end
```

实现的代码会第一次使用 `has_many through`：用户通过 `relationships` 表拥有多个关注关系，就像图 11.7 所示的那样。默认情况下，在 `has_many through` 关联中，Rails 会寻找关联名单数形式对应的外键，也就是说，像下面的代码

```
has_many :followeds, through: :relationships
```

会使用 `relationships` 表中的 `followed_id` 列生成一个数组。但是，正如在 11.1.1 节中说过的，`user.followeds` 这种说法比较蹩脚，若使用“followed users”作为 “followed”的复数形式会好得多，那么被关注的用户数组就要写成 `user.followed_users` 了。Rails 当然会允许我们重写默认的设置，针对本例，我们可以使用 `:source` 参数（参见代码 11.10），告知 Rails `followed_users` 数组的来源是 `followed` 所代表的 id 集合。

代码 11.10：在 User 模型中添加 `followed_users` 关联
`app/models/user.rb`

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  has_many :relationships, foreign_key: "follower_id", dependent: :destroy
  has_many :followed_users, through: :relationships, source: :followed
  .
```

```
  .
  .
end
```

为了创建关注关联关系，我们将定义一个名为 `follow!` 的方法，这样我们就能使用 `user.follow!(other_user)` 这样的代码创建关注了。（`follow!` 方法应该与 `create!` 和 `save!` 方法一样，失败时抛出异常，所以我们在后面加上了感叹号。）对应地，我们还会添加一个 `following?` 布尔值方法，检查一个用户是否关注了另一个用户。[7]代码 11.11 中的测试表明了我们希望如何使用这两个方法。

代码 11.11： 测试关注关系用到的方法
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:followed_users) }
  it { should respond_to(:following?) }
  it { should respond_to(:follow!) }
  .
  .
  .
  describe "following" do
    let(:other_user) { FactoryGirl.create(:user) }
    before do
      @user.save
      @user.follow!(other_user)
    end

    it { should be_following(other_user) }
    its(:followed_users) { should include(other_user) }
  end
end
```

在实现的代码中，`following` 方法接受一个用户对象作为参数，参数名为 `other_user`，检查这个被关注者的 id 在数据库中是否存在；`follow!` 方法直接调用 `create!` 方法，通过和 Relationship 模型的关联来创建关注关系，如代码 11.12 所示。

代码 11.12： 定义 `following?` 和 `follow!` 方法
`app/models/user.rb`

7. 当你拥有在某个领域大量建立模型的经验后，总能提前猜到这样的工具方法，如果没有猜到的话，也经常能发现自己动手写这样的方法可以使测试代码更加整洁。此时，如果你没有猜到它们的话也很正常。软件开发经常是一个循序渐进的过程，你先埋头编写代码，发现代码很乱时，再重构。 但为了行文简洁，本书采取的是直捣黄龙的方法。

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

  def following?(other_user)
    relationships.find_by(followed_id: other_user.id)
  end

  def follow!(other_user)
    relationships.create!(followed_id: other_user.id)
  end
  .
  .
  .
end
```

注意，在代码 11.12 中我们忽略了用户对象自身，直接写成

```
relationships.create!(…)
```

而不是等效的

```
self.relationships.create!(…)
```

是否使用 `self` 关键字只是个人偏好而已。

当然，用户应该既能关注也能取消关注，那么还应该有一个 `unfollow!` 方法，如代码 11.13 所示。[8]

代码 11.13：测试取消关注用户
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:follow!) }
```

8. 事实上 `unfollow!` 方法在失败时不会抛出异常，我甚至不知道 Rails 是如何表明删除操作失败的。不过为了和 `follow!` 保持一致，我们还是加上了感叹号。

```
  it { should respond_to(:unfollow!) }
  .
  .
  .
  describe "following" do
    .
    .
    .
    describe "and unfollowing" do
      before { @user.unfollow!(other_user) }

      it { should_not be_following(other_user) }
      its(:followed_users) { should_not include(other_user) }
    end
  end
end
```

unfollow! 方法的定义很容易理解，通过 followed_id 找到对应的“关系”删除就行了，如代码 11.14 所示。

代码 11.14：删除“关系”取消关注用户
app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  def following?(other_user)
    relationships.find_by(followed_id: other_user.id)
  end

  def follow!(other_user)
    relationships.create!(followed_id: other_user.id)
  end

  def unfollow!(other_user)
    relationships.find_by(followed_id: other_user.id).destroy
  end
  .
  .
  .
end
```

11.1.5 粉丝

关注关系的最后一部分是定义和 user.followed_users 相对应的 user.followers 方法。从图 11.7 你或许发现了，获取粉丝数组所需的数据都已经存入 relationships 表中了。这里我们用到的方法和实现被关注者时一

样，只要对调 `follower_id` 和 `followed_id` 的位置即可。这说明， 只要我们对调这两列的位置，组建成 `reverse_relationships` 表（如图 11.9 所示），`user.followers` 方法的定义就很容易了。

图 **11.9**：使用倒转后的 Relationship 模型获取粉丝

我们先来编写测试，相信神奇的 Rails 将再一次显现威力，如代码 11.15 所示。

代码 11.15：测试对调后的关注关系
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:relationships) }
  it { should respond_to(:followed_users) }
  it { should respond_to(:reverse_relationships) }
  it { should respond_to(:followers) }
  .
  .
  .

  describe "following" do
    .
    .
    .
    it { should be_following(other_user) }
    its(:followed_users) { should include(other_user) }

    describe "followed user" do
```

```
      subject { other_user }
      its(:followers) { should include(@user) }
    end
    .
    .
    .
  end
end
```

注意一下上述代码中，我们是如何使用 subject 来转变测试对象的，我们从 @user 转到了 other_user，然后，我们就能使用下面下面这种很自然的方式测试粉丝中是否包含 @user 了：

```
subject { other_user }
its(:followers) { should include(@user) }
```

你可能已经想到了，我们不会再建立一个完整的数据表来存放倒转后的关注关系。事实上，我们会通过被关注者和粉丝之间的对称关系来模拟一个 reverse_relationships 表，主键设为 followed_id。也就是说，relationships 表使用 follower_id 做外键：

```
has_many :relationships, foreign_key: "follower_id"
```

那么，reverse_relationships 虚拟表就用 followed_id 做外键：

```
has_many :reverse_relationships, foreign_key: "followed_id"
```

粉丝的关联就建立在这层反转的关系上，如代码 11.16 所示。

代码 11.16：通过反转的关系实现 user.followers
app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :reverse_relationships, foreign_key: "followed_id",
                                   class_name:  "Relationship",
                                   dependent:   :destroy
  has_many :followers, through: :reverse_relationships, source: :follower
  .
  .
  .
end
```

（和代码 11.4 一样，针对 dependent :destroy 的测试会留作练习，参见 11.5 节。）注意为了实现数据表之间的关联，我们要指定类名：

```
has_many :reverse_relationships, foreign_key: "followed_id",
                                 class_name: "Relationship"
```

如果没有指定类名，Rails 会尝试寻找 `ReverseRelationship` 类，而这个类并不存在。

还有一点值得注意一下，在里我们其实可以省略 `:source` 参数，使用下面的简单方式

```
has_many :followers, through: :reverse_relationships
```

对 `:followers` 属性而言，Rails 会把“followers”转成单数形式，自动寻找名为 `follower_id` 的外键。在此我保留了 `:source` 参数是为了保持与 `has_many :followed_users` 关系之间结构上的对称，你也可以选择去掉它。

加入代码 11.16 之后，关注者和粉丝之间的关联就完成了，所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

11.2 关注用户功能的网页界面

在本章的导言中，我们介绍了关注用户功能的操作流程。本节我们会实现这些构思的基本界面，以及关注和取消关注操作。同时，我们还会创建两个页面，分别列出关注的用户和粉丝。在 11.3 节中我们会加入用户的动态列表，其时，这个示例程序才算完成。

11.2.1 用户关注用到的示例数据

和之前的几章一样，我们会使用 Rake 任务生成示例数据，向数据库中存入临时的用户关注关联数据。有了这些示例数据，我们就可以先开发网页，而把后端功能的实现放在本节的最后。

我们在代码 10.20 中用到的示例数据生成器有点乱，所以现在我们要分别定义两个方法，用来生成用户和微博示例数据，然后再定义 `make_relationships` 方法，生成用户关注关联数据，如代码 11.17 所示。

代码 11.17： 加入用户关注关联示例数据
`lib/tasks/sample_data.rake`

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    make_users
    make_microposts
    make_relationships
  end
end

def make_users
  admin = User.create!(name:     "Example User",
```

```
                        email:    "example@railstutorial.org",
                        password: "foobar",
                        password_confirmation: "foobar",
                        admin: true)
  99.times do |n|
    name  = Faker::Name.name
    email = "example-#{n+1}@railstutorial.org"
    password  = "password"
    User.create!(name:     name,
                 email:    email,
                 password: password,
                 password_confirmation: password)
  end
end

def make_microposts
  users = User.all(limit: 6)
  50.times do
    content = Faker::Lorem.sentence(5)
    users.each { |user| user.microposts.create!(content: content) }
  end
end

def make_relationships
  users = User.all
  user  = users.first
  followed_users = users[2..50]
  followers      = users[3..40]
  followed_users.each { |followed| user.follow!(followed) }
  followers.each      { |follower| follower.follow!(user) }
end
```

用户关注关联的示例数据是由下面的代码生成的：

```
def make_relationships
  users = User.all
  user  = users.first
  followed_users = users[2..50]
  followers      = users[3..40]
  followed_users.each { |followed| user.follow!(followed) }
  followers.each      { |follower| follower.follow!(user) }
end
```

我们的安排是随机的，让第 1 个用户关注第 3 到第 51 个用户，再让第 4 到第 41 个用户关注第 1 个用户。形成了这样的用户关注网，就足够用来开发程序的界面了。

和之前一样，要想运行代码 11.17，就要执行下面的数据库命令：

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake test:prepare
```

11.2.2 数量统计和关注表单

现在用户已经有关注的人和粉丝了，我们要更新一下用户资料页面和首页，把这些变动显示出来。首先，我们要创建一个关注和取消关注的表单，然后再创建显示被关注用户列表和粉丝列表的页面。

我们在 11.1.1 节中说过，“following”这个词作为属性名是有点奇怪的（因为 `user.following` 既可以理解为被关注的用户，也可以理解为关注的用户），但可以作为标签使用，例如，可以说“50 following”。其实，Twitter 就使用了这种标签形式，图 11.1 中的构思图沿用了这种表述，这部分详细的构思如图 11.10 所示。

图 11.10：数量统计局部视图的构思图

图 11.10 中显示的数量统计包含了当前用户关注的用户和关注了该用户的粉丝数，二者又分别链接到了各自详细的用户列表页面。在第 5 章中，我们使用 # 占位符代替真实的网址，因为那时我们还没怎么接触路由。现在，虽然在 11.2.3 节中才会创建所需的页面，不过我们可以先设置路由，如代码 11.18 所示。这段代码在 `resources` 块中使用了 `:member` 方法，以前没用过，你可以猜测一下这个方法的作用是什么。（注意，代码 11.18 是用来替换原来的 `resources :users` 的。）

代码 11.18：把 `following` 和 `folloers` 动作加入 Users 控制器的路由中
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users do
    member do
      get :following, :followers
    end
  end
  .
  .
  .
end
```

你可能猜到了，设定上述路由后，得到的 URL 地址应该是类似 /users/1/following 和 /users/1/followers 这种形式，不错，代码 11.18 的作用确实如此。因为这两个页面都是用来显示数据的，所以我们使用了 `get` 方法，指定这两个地址响应的是 GET 请求。（符合 REST 架构对这种页面的要求）。路由设置中使用的 `member` 方法作用是，设置这两个动作对应的 URL 地址中应该包含用户的 id。类似地，我们还可以使用 `collection` 方法，但 URL 中就没有用户 id 了，所以，如下的代码

```
resources :users do
  collection do
    get :tigers
  end
end
```

设定路由后得到的 URL 是 /users/tigers（可以用来显示程序中所有的老虎）。关于路由的这种设置，更详细的说明可以阅读 Rails 指南中的《Rails Routing from the Outside In》一文。代码 11.18 所生成的路由如表格 11.1 所示。请留意一下被关注用户和粉丝页面的具名路由是什么，稍后我们会用到。为了避免用词混淆，在“following”路由中我们没有使用“followed users”这种说法，而沿用了 Twitter 的方式，采用“following”这个词，因为“followed users” 这种用法会生成 `followed_users_user_path` 这种奇怪的具名路由。如表格 11.1 所示，我们选择使用“following”这个词，因此得到的具名路由是 `following_user_path`。

表格 11.1：代码 11.18 中设置的路由生成的 REST 路由

HTTP 请求	URL	动作	具名路由
GET	/users/1/following	`following`	`following_user_path(1)`
GET	/users/1/followers	`followers`	`followers_user_path(1)`

设好了路由后，我们来编写对数量统计局部视图的测试。（原本我们可以先写测试的，但是如果没加入所需的路由设置，可能无从下手编写测试。）数量统计局部视图会出现在用户资料页面和首页中，代码 11.19 只对首页进行了测试。

代码 11.19：测试首页中显示的关注和粉丝数量统计
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  describe "Home page" do
    .
    .
    .
    describe "for signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        FactoryGirl.create(:micropost, user: user, content: "Lorem")
        FactoryGirl.create(:micropost, user: user, content: "Ipsum")
        sign_in user
        visit root_path
      end
```

```
      it "should render the user's feed" do
        user.feed.each do |item|
          expect(page).to have_selector("li##{item.id}", text: item.content)
        end
      end

      describe "follower/following counts" do
        let(:other_user) { FactoryGirl.create(:user) }
        before do
          other_user.follow!(user)
          visit root_path
        end

        it { should have_link("0 following", href: following_user_path(user)) }
        it { should have_link("1 followers", href: followers_user_path(user)) }
      end
    end
  end
  .
  .
  .
end
```

上述测试的核心是，检测页面中是否显示了关注和粉丝数，以及是否指向了正确的地址：

```
it { should have_link("0 following", href: following_user_path(user)) }
it { should have_link("1 followers", href: followers_user_path(user)) }
```

我们用到了表格 11.1中的具名路由来测试链接是否指向了正确的地址。还有一处要注意一下，这里的“followers”是作为标签使用的，所以我们会一直用复数形式，即使只有一个粉丝也是如此。

数量统计局部视图的代码很简单，在一个 `div` 元素中显示几个链接就行了，如代码 11.20 所示。

代码 11.20： 显示关注数量统计的局部视图
app/views/shared/_stats.html.erb

```
<% @user ||= current_user %>
<div class="stats">
  <a href="<%= following_user_path(@user) %>">
    <strong id="following" class="stat">
      <%= @user.followed_users.count %>
    </strong>
    following
  </a>
  <a href="<%= followers_user_path(@user) %>">
    <strong id="followers" class="stat">
```

```
      <%= @user.followers.count %>
    </strong>
    followers
  </a>
</div>
```

因为这个局部视图会同时在用户资料页面和首页中显示，所以在代码 11.20 的第一行中，我们要获取正确的用户对象：

```
<% @user ||= current_user %>
```

我们在旁注 8.2中介绍过这样的用法，如果 `@user` 不是 `nil`（在用户资料页面），这行代码就没什么效果，而如果是 `nil`（在首页）， 就会把当前用户对象赋值给 `@user`。

还有一处也要注意一下，关注数量和粉丝数量是通过关联获取的，分别使用 `@user.followed_users.count` 和 `@user.followers.count`。

我们可以和代码 10.17 中获取微博数量的代码对比一下，微博的数量是通过 `@user.microposts.count` 获取的。

最后还有一些细节需要注意下，那就是某些元素的 CSS id，例如

```
<strong id="following" class="stat">
...
</strong>
```

这些 id 是为 11.2.5 节中实现 Ajax 功能服务的，Ajax 会通过独一无二的 id 获取页面中的元素。

编好了局部视图，把它放入首页中就很简单了，如代码 11.21 所示。（加入局部视图后，代码 11.19 中的测试也就会通过了。）

代码 11.21：在首页中显示的关注和粉丝数量统计
`app/views/static_pages/home.html.erb`

```
<% if signed_in? %>
      .
      .
      .
      <section>
        <%= render 'shared/user_info' %>
      </section>
      <section>
        <%= render 'shared/stats' %>
      </section>
      <section>
        <%= render 'shared/micropost_form' %>
      </section>
      .
```

```
      .
      .
<% else %>
  .
  .
  .
<% end %>
```

我们会添加一些 SCSS 代码来美化一下数量统计部分，如代码 11.22 所示（这段代码包含了本章用到的所有样式）。添加样式后的页面如图 11.11 所示。

代码 11.22： 首页侧边栏的 SCSS 样式
`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* sidebar */
.
.
.
.stats {
  overflow: auto;
  a {
    float: left;
    padding: 0 10px;
    border-left: 1px solid $grayLighter;
    color: gray;
    &:first-child {
      padding-left: 0;
      border: 0;
    }
    &:hover {
      text-decoration: none;
      color: $blue;
    }
  }
  strong {
    display: block;
  }
}

.user_avatars {
  overflow: auto;
  margin-top: 10px;
```

```
  .gravatar {
    margin: 1px 1px;
  }
}
.
.
.
```

图 11.11：显示了关注数量统计的首页

稍后我们再把数量统计局部视图加入用户资料页面，现在先来编写关注和取消关注按钮的局部视图，如代码 11.23 所示。

代码 11.23：关注和取消关注表单
app/views/users/_follow_form.html.erb

```
<% unless current_user?(@user) %>
  <div id="follow_form">
  <% if current_user.following?(@user) %>
    <%= render 'unfollow' %>
  <% else %>
    <%= render 'follow' %>
  <% end %>
```

```
  </div>
<% end %>
```

这段代码其实也没做什么，只是把具体的工作分配给 follow 和 unfollow 局部视图了，这样我们就要再次设置路由，加入 Relationships 资源，参照 Microposts 资源的设置（参见代码 10.25），如代码 11.24 所示。

代码 11.24：添加 Relationships 资源的路由设置
config/routes.rb

```
SampleApp::Application.routes.draw do
  .
  .
  .
  resources :sessions,      only: [:new, :create, :destroy]
  resources :microposts,    only: [:create, :destroy]
  resources :relationships, only: [:create, :destroy]
  .
  .
  .
end
```

follow 和 unfollow 局部视图的代码分别如代码 11.25 和代码 11.26 所示。

代码 11.25：关注用户的表单
app/views/users/_follow.html.erb

```
<%= form_for(current_user.relationships.build(followed_id: @user.id)) do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <%= f.submit "Follow", class: "btn btn-large btn-primary" %>
<% end %>
```

代码 11.26：取消关注用户的表单
app/views/users/_unfollow.html.erb

```
<%= form_for(current_user.relationships.find_by(followed_id: @user),
             html: { method: :delete }) do |f| %>
  <%= f.submit "Unfollow", class: "btn btn-large" %>
<% end %>
```

这两个表单都使用了 form_for 来处理 Relationship 模型对象，二者之间主要的不同点是，代码 11.25 中的代码是构建一个新的“关系”，而代码 11.26 是查找现有的“关系”。很显然，第一个表单会向 Relationships 控制器发送 POST 请求，创建“关系”；而第二个表单发送的是 DELETE 请求，销毁“关系”。（两个表单用到的动作会在 11.2.4 节编写。）你可能还注意到了，这两个表单中除了按钮之外什么内容也没有，但是还要传送 followed_id，我们会调用 hidden_fields 方法将其加入，生成的 HTML 如下：

```
<input id="relationship_followed_id" name="relationship[followed_id]"
type="hidden" value="3" />
```

隐藏的 input 表单域会把所需的信息包含在表单中，但是在浏览器中不会显示出来。

现在我们可以在页面中加入关注表单和关注数量统计了，只需渲染相应的局部视图即可，如代码 11.27 所示。在用户资料页面中，根据实际的关注情况，会分别显示关注按钮或取消关注按钮，如图 11.12 和图 11.13 所示。

图 11.12：显示了关注按钮的用户资料页面（/users/2）

代码 11.27：在用户资料页面加入关注表单和关注数量统计
app/views/users/show.html.erb

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="span4">
    <section>
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
    <section>
      <%= render 'shared/stats' %>
    </section>
  </aside>
  <div class="span8">
    <%= render 'follow_form' if signed_in? %>
    .
```

```
    .
    .
  </div>
</div>
```

稍后我们会让这些按钮起作用，而且我们会使用两种实现方式，一种是常规方式（11.2.4 节），另一种是使用 Ajax 的方式（11.2.5 节）。不过在此之前，我们要完成用户界面的制作，创建显示关注的用户列表和粉丝列表的页面。

11.2.3 关注列表和粉丝列表页面

显示关注列表和粉丝列表的页面基本上是重组用户的资料页面和用户索引页面（参见 9.3.1 节），在侧边栏中显示用户的信息（包括关注数量统计），再列出用户列表。除此之外，还会在侧边栏中显示一个由用户的头像组成的栅格。构思图如图 11.14（关注的人）和图 11.15（粉丝们）所示。

首先，我们要让这两个页面的地址可访问，按照 Twitter 的方式，访问这两个页面都需要先登录，测试如代码 11.28 所示。用户登录后，页面中应该分别显示关注的用户列表和粉丝列表，测试如代码 11.29 所示。

代码 11.28： 测试关注列表和粉丝列表页面的访问权限设置
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "in the Users controller" do
        .
        .
        .
        describe "visiting the following page" do
          before { visit following_user_path(user) }
          it { should have_title('Sign in') }
        end

        describe "visiting the followers page" do
          before { visit followers_user_path(user) }
          it { should have_title('Sign in') }
        end
      end
      .
```

```
      .
      .
    end
    .
    .
    .
  end
  .
  .
  .
end
```

代码 11.29：测试关注列表和粉丝列表页面
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "following/followers" do
    let(:user) { FactoryGirl.create(:user) }
    let(:other_user) { FactoryGirl.create(:user) }
    before { user.follow!(other_user) }

    describe "followed users" do
      before do
        sign_in user
        visit following_user_path(user)
      end

      it { should have_title(full_title('Following')) }
      it { should have_selector('h3', text: 'Following') }
      it { should have_link(other_user.name, href: user_path(other_user)) }
    end

    describe "followers" do
      before do
        sign_in other_user
        visit followers_user_path(other_user)
      end

      it { should have_title(full_title('Followers')) }
      it { should have_selector('h3', text: 'Followers') }
      it { should have_link(user.name, href: user_path(user)) }
```

```
    end
  end
end
```

图 11.13：显示了取消关注按钮的用户资料页面（/users/6）

在这两个页面的实现过程中，唯一一处很难想到的地方是，要意识到我们需要在 Users 控制器中添加两个动作，按照代码 11.18 中路由的设置，这两个动作分别名为 `following` 和 `followers`。在这两个动作中，需要设置页面的标题，查询用户，分别获取 `@user.followed_users` 和 `@user.followers`（要分页显示），然后再渲染页面，如代码 11.30 所示。

代码 11.30：`following` 和 `followers` 动作
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :signed_in_user,
                only: [:index, :edit, :update, :destroy, :following, :followers]
  .
  .
  .
  def following
    @title = "Following"
    @user = User.find(params[:id])
```

```
    @users = @user.followed_users.paginate(page: params[:page])
    render 'show_follow'
  end

  def followers
    @title = "Followers"
    @user = User.find(params[:id])
    @users = @user.followers.paginate(page: params[:page])
    render 'show_follow'
  end

  private
  .
  .
  .
end
```

图 11.14：关注列表页面的构思图

图 11.15：粉丝列表页面的构思图

注意，在这两个动作中都明确的调用了 `render` 方法，渲染一个名为 `show_follow` 的视图，接下来我们就来编写这个视图。这两个动作之所以使用同一个视图，是因为两种情况用到的 ERb 代码差不多，如代码 11.31 所示。

代码 11.31：渲染关注列表和粉丝列表的 `show_follow` 视图
`app/views/users/show_follow.html.erb`

```
<% provide(:title, @title) %>
<div class="row">
  <aside class="span4">
    <section>
      <%= gravatar_for @user %>
      <h1><%= @user.name %></h1>
      <span><%= link_to "view my profile", @user %></span>
      <span><b>Microposts:</b> <%= @user.microposts.count %></span>
    </section>
    <section>
      <%= render 'shared/stats' %>
      <% if @users.any? %>
        <div class="user_avatars">
```

```
          <% @users.each do |user| %>
            <%= link_to gravatar_for(user, size: 30), user %>
          <% end %>
        </div>
      <% end %>
    </section>
  </aside>
  <div class="span8">
    <h3><%= @title %></h3>
    <% if @users.any? %>
      <ul class="users">
        <%= render @users %>
      </ul>
      <%= will_paginate %>
    <% end %>
  </div>
</div>
```

这时，测试应该可以通过了，页面也能正常显示了，如图 11.16（关注的人）和图 11.17（粉丝们）所示。

图 11.16： 显示当前用户关注的人

图 11.17：显示当前用户的粉丝

11.2.4 关注按钮的常规实现方式

创建好了视图后，我们就要让关注和取消关注按钮起作用了。针对这两个按钮的测试用到了本教程中介绍的很多测试技术，也是对代码阅读能力的考察。请认真的阅读代码 11.32，直到你理解了测试的内容以及为什么这么做，再阅读后面的内容。（这段代码中有一处很小的安全疏漏，看一下你是否能发现。稍后我们会说明。）注意，`have_xpath` 方法是高级而强大的技术，该方法使用 `[XPath](http://en.wikipedia.org/wiki/XPath)` 查找 XML 文档（也包括 HTML5）。你可以搜索“XPath syntax” 了解更多内容。

代码 11.32：测试关注和取消关注按钮
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "profile page" do
    let(:user) { FactoryGirl.create(:user) }
    .
    .
```

```
.
describe "follow/unfollow buttons" do
  let(:other_user) { FactoryGirl.create(:user) }
  before { sign_in user }

  describe "following a user" do
    before { visit user_path(other_user) }

    it "should increment the followed user count" do
      expect do
        click_button "Follow"
      end.to change(user.followed_users, :count).by(1)
    end

    it "should increment the other user's followers count" do
      expect do
        click_button "Follow"
      end.to change(other_user.followers, :count).by(1)
    end

    describe "toggling the button" do
      before { click_button "Follow" }
      it { should have_xpath("//input[@value='Unfollow']") }
    end
  end

  describe "unfollowing a user" do
    before do
      user.follow!(other_user)
      visit user_path(other_user)
    end

    it "should decrement the followed user count" do
      expect do
        click_button "Unfollow"
      end.to change(user.followed_users, :count).by(-1)
    end

    it "should decrement the other user's followers count" do
      expect do
        click_button "Unfollow"
      end.to change(other_user.followers, :count).by(-1)
    end

    describe "toggling the button" do
```

```
          before { click_button "Unfollow" }
          it { should have_xpath("//input[@value='Follow']") }
        end
      end
    end
  end
  .
  .
  .
end
```

上述针对关注和取消关注按钮的测试，先点击这两个按钮，然后检测是否做了适当的操作。实现这两个按钮的操作需要想得深入一点：关注和取消关注的过程涉及到创建“关系”和销毁“关系”，也就是说，要在 Relationships 控制器中定义 `create` 和 `destroy` 动作（这就是我们要做的）。虽然只有登录后的用户才能看到关注和取消关注按钮，增加了一层安全措施，但是代码 11.32 中的测试疏忽了一个较为底层的问题，那就是 `create` 和 `destroy` 动作本身只能被登录后的用户访问。（这就是前面提到的安全疏漏。）代码 11.33 中的测试，分别调用 `post` 和 `delete` 方法直接访问这两个动作，检测未登录的用户是否能访问相应的动作。

代码 11.33： 测试 Relationships 控制器的访问限制
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      .
      .
      .
      describe "in the Relationships controller" do
        describe "submitting to the create action" do
          before { post relationships_path }
          specify { expect(response).to redirect_to(signin_path) }
        end

        describe "submitting to the destroy action" do
          before { delete relationship_path(1) }
          specify { expect(response).to redirect_to(signin_path) }
        end
      end
      .
```

```
      .
      .
    end
  end
end
```

注意，我们不想多定义一个没用的 Relationship 对象，所以在针对 `delete` 请求的测试中，在具名路由中直接写入了 id 的值：

```
before { delete relationship_path(1) }
```

这样的代码之所以有效，是因为程序在尝试使用 id 获取“关系”之前就应该转向到登录页面了。

能够让上述测试通过的控制器代码极其简单，我们只需获得已经关注或想关注的用户对象，然后使用相应的工具方法关注或取消关注就可以了。具体的实现代码如代码 11.34 所示。

代码 11.34： Relationships 控制器
`app/controllers/relationships_controller.rb`

```
class RelationshipsController < ApplicationController
  before_action :signed_in_user

  def create
    @user = User.find(params[:relationship][:followed_id])
    current_user.follow!(@user)
    redirect_to @user
  end

  def destroy
    @user = Relationship.find(params[:id]).followed
    current_user.unfollow!(@user)
    redirect_to @user
  end
end
```

从代码 11.34 我们能够看出为什么前面提到的安全疏漏不是什么大问题，因为如果未登录的用户直接访问了任意一个动作（例如，使用命令行工具 curl），`current_user` 的值就是 `nil`，那么动作的第二行代码就会抛出异常，得到错误提示，而不会破坏程序或数据。不过，最好还是不要依靠这样的处理方式，因此我们在前面才加入了额外的安全措施。

至此，整个关注和取消关注的功能就都实现了，任何一个用户都可以关注或取消关注另一个用户了，你可以在程序中点击相应的链接检查一下，也可以运行测试验证一下：

```
$ bundle exec rspec spec/
```

11.2.5 关注按钮的 Ajax 实现方式

虽然在上一小节我们说过关注用户的功能已经完全实现了，但是在实现状态列表之前，还有可以增强的地方。你可能已经注意到了，在 11.2.4 节中，Relationships 控制器的 `create` 和 `destroy` 动作最后都返回了一开始访问的用户资料页面。也就是说，用户 A 先浏览了用户 B 的资料页面，点击关注按钮关注用户 B，然后页面会立马转回到用户 A 的资料页面。因此，对这样的过程我们就有了一个疑问：为什么要多一次页面转向呢？

Ajax 可以解决这个疑问，通过向服务器发送异步请求，在不刷新页面的情况下，Ajax 就可以更新页面里的内容。[9] 因为在表单中处理 Ajax 请求是很常用的技术，所以 Rails 把实现 Ajax 的过程变得很简单。其实，关注和取消关注表单局部视图不用做大的改动，只要把 `form_for` 改成 `form_for..., remote: true`，Rails 就会自动使用 Ajax 处理表单了。[10]更新后的局部视图如代码 11.35 和代码 11.36 所示。

代码 11.35：使用 Ajax 后的关注用户表单
`app/views/users/_follow.html.erb`

```
<%= form_for(current_user.relationships.build(followed_id: @user.id),
             remote: true) do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <%= f.submit "Follow", class: "btn btn-large btn-primary" %>
<% end %>
```

代码 11.36：使用 Ajax 后的取消关注用户表单
`app/views/users/_unfollow.html.erb`

```
<%= form_for(current_user.relationships.find_by(followed_id: @user),
             html: { method: :delete },
             remote: true) do |f| %>
  <%= f.submit "Unfollow", class: "btn btn-large" %>
<% end %>
```

上述 ERb 代码生成的 HTML 没什么好说的，如果你好奇的话，可以看一下下面的示例：

```
<form action="/relationships/117" class="edit_relationship" data-remote="true"
      id="edit_relationship_117" method="post">
  .
  .
  .
</form>
```

从这段代码中我们可以看到，`form` 元素中设置了 `data-remote="true"`，这个属性就是用来告知 Rails 该表单可以使用 JavaScript 处理的。Rails 3 遵从了“非侵入式 JavaScript”原则（unobtrusive JavaScript），没有在视图中写入整个 JavaScript 代码（在 Rails 之前的版本中却是这么做的），而是使用了一个简单的 HTML 属性。

更新表单后，我们要让 Relationships 控制器可以响应那个 Ajax 请求。针对 Ajax 的测试有点复杂，完全可以写本书了，不过我们可以先从代码 11.37 下手。这段测试中使用了 `xhr` 方法（表示“XmlHttpRequest”）发送 Ajax 请求，

9. 因为 Ajax 是 asynchronous JavaScript and XML 的缩写，所以经常被错误的拼写为“AJAX”，不过在最初介绍 Ajax 的文章中，通篇都拼写为“Ajax”。
10. 只有当浏览器启用 JavaScript 时才能正常使用，不过可以优雅降级，如果禁用了 JavaScript 就会按照 11.2.4 节中的方式工作。

xhr 方法和之前使用的 get、post、patch 和 delete 方法是类似的。然后再检查发送 Ajax 请求后，create 和 destroy 动作是否进行了正确的操作。（如果要为大量使用 Ajax 的程序编写完整的测试，请了解一下 Selenium 和 Watir。）

代码 11.37： 测试 Relationships 控制器对 Ajax 请求的响应
spec/controllers/relationships_controller_spec.rb

```
require 'spec_helper'

describe RelationshipsController do

  let(:user) { FactoryGirl.create(:user) }
  let(:other_user) { FactoryGirl.create(:user) }

  before { sign_in user, no_capybara: true }

  describe "creating a relationship with Ajax" do

    it "should increment the Relationship count" do
      expect do
        xhr :post, :create, relationship: { followed_id: other_user.id }
      end.to change(Relationship, :count).by(1)
    end

    it "should respond with success" do
      xhr :post, :create, relationship: { followed_id: other_user.id }
      expect(response).to be_success
    end
  end

  describe "destroying a relationship with Ajax" do

    before { user.follow!(other_user) }
    let(:relationship) { user.relationships.find_by(followed_id: other_user) }

    it "should decrement the Relationship count" do
      expect do
        xhr :delete, :destroy, id: relationship.id
      end.to change(Relationship, :count).by(-1)
    end

    it "should respond with success" do
      xhr :delete, :destroy, id: relationship.id
      expect(response).to be_success
    end
```

```
  end
end
```

代码 11.37 是我们第一次使用控制器测试（controller test），我以前经常使用控制器测试（如在本书的第一版中），但是现在我倾向于使用集成测试。这里之所以使用控制器测试是因为，xhr 方法在集成测试中不可用（有点让人不解）。虽然这是我们第一次使用 xhr 方法，但结合本书前面的内容，你应该可以理解下面这行代码的作用：

```
xhr :post, :create, relationship: { followed_id: other_user.id }
```

我们看到，xhr 方法的第一个参数是相应的 HTTP 方法，第二个参数是动作名，第三个参数是一个 Hash，其元素是控制器中的 params 变量的值。和以前的测试一样，我们把相关的操作放入 expect 块中，检查数量是不是增加或减少了。

这段测试说明：在程序中，我们要使用同等的 create 和 destroy 动作响应 Ajax 请求，这两个动作之前可响应的是普通的 POST 请求和 DELETE 请求。我们要做的是：当接到普通的 HTTP 请求时，进行页面转向（参见 11.2.4 节），而当接到 Ajax 请求时使用 JavaScript 进行处理。控制器的代码如代码 11.38 所示。（在 11.5 节的练习中，我们介绍了一种更简单的实现方式。）

代码 11.38：在 Relationships 控制器中响应 Ajax 请求
app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :signed_in_user

  def create
    @user = User.find(params[:relationship][:followed_id])
    current_user.follow!(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end

  def destroy
    @user = Relationship.find(params[:id]).followed
    current_user.unfollow!(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end
end
```

代码 11.38 中使用了 respond_to 方法，根据接到的请求类型进行不同的操作。（这里用到的 respond_to 和 RSpec 中的 respond_to 没任何联系。）repond_to 方法的写法可能有点让人迷糊，你要知道，如下的代码

```
respond_to do |format|
  format.html { redirect_to @user }
  format.js
end
```

只有一行会被执行（依据请求的类型而定）。

在处理 Ajax 请求时，Rails 会自动调用文件名和动作名一样的“含有 JavaScript 的 ERb（JavaScript Embedded Ruby）”文件（扩展名为 `.js.erb`），例如 `create.js.erb` 和 `destroy.js.erb`。你可能已经猜到了，这种文件是可以包含 JavaScript 和 Ruby 代码的，可以用来处理当前页面的内容。在关注用户和取消关注用户时，更新用户资料页面的内容就需要创建这种文件。

在 JS-ERb 文件中，Rails 自动提供了 jQuery 库的帮助函数，可以通过“文本对象模型（Document Object Model，DOM）”处理页面的内容。jQuery 库中有很多处理 DOM 的函数，但现在我们只会用到其中的两个。首先，我们要知道通过 id 获取 DOM 元素的美元符号，例如，要获取 `follow_form` 元素，我们可以使用如下的代码：

```
$("#follow_form")
```

（参见代码 11.23，这个元素是包含表单的 `div`，而不是表单本身。）上面的句法和 CSS 一样，`#` 符号表示 CSS 中的 id。由此你可能猜到了，jQuery 和 CSS 一样，点号 `.` 表示 CSS 中的 class。

我们会使用的第二个函数是 `html`，它会使用参数中指定的内容修改元素所包含的 HTML。例如，如果要把整个表单换成字符串 `"foobar"`，jQuery 代码可以这么写：

```
$("#follow_form").html("foobar")
```

和常规的 JavaScript 文件不同，JS-ERb 文件还可以使用嵌入式 Ruby 代码。在 `create.js.erb` 文件中我们会把关注用户表单换成取消关注用户表单（成功关注后），并更新关注者的数量，如代码 11.39 所示。这段代码中用到了 `escape_javascript` 方法，在 JavaScript 中写入 HTML 代码必须使用这个方法对 HTML 进行转义。

代码 11.39： 创建关注“关系”的 JS-ERb 代码
`app/views/relationships/create.js.erb`

```
$("#follow_form").html("<%= escape_javascript(render('users/unfollow')) %>")
$("#followers").html('<%= @user.followers.count %>')
```

`destroy.js.erb` 文件的内容类似，如代码 11.40 所示。

代码 11.40： 销毁关注“关系”的 JS-ERb 代码
`app/views/relationships/destroy.js.erb`

```
$("#follow_form").html("<%= escape_javascript(render('users/follow')) %>")
$("#followers").html('<%= @user.followers.count %>')
```

加入上述代码后，你应该访问用户资料页面，看一下关注或取消关注用户后页面是不是真的没有刷新，再验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

在 Rails 中使用 Ajax 可以讲的太多了，技术变化的也快，我们只是介绍了点皮毛（本书其他的内容也是如此），你可以据此为基础学习其他更高级的用法。

11.3 动态列表

接下来我们要实现示例程序最难的功能：动态列表。基本上本节的内容算是全书最高深的部分了。完整的动态列表是以 10.3.3 节的临时动态列表为基础实现的，列表中除了当前用户自己的微博之外，还包含了他所关注用户的微博。为了实现这样的功能，我们会用到一些很高级的 Rails、Ruby 和 SQL 技术。

因为我们要做的事情很多，在此之前最好先清楚我们要实现的是什么样的功能。图 11.5 显示了最终要实现的动态列表，图 11.8 还是同一幅图。

11.3.1 目的和策略

我们对动态列表的构思是很简单的。图 11.19 中显示了一个示例的 `microposts` 表和要显示的动态。动态列表就是要把当前用户所关注用户的微博（也包括当前用户自己的微博）从 `microposts` 表中取出来，如图中箭头所指。

图 11.18： 显示有动态列表的用户资料页面构思图

因为需要把指定用户所关注用户的全部微博取出来，我们计划定义一个名为 `from_users_followed_by` 方法，以下面的方式调用：

```
Micropost.from_users_followed_by(user)
```

虽然我们还不知道怎么定义这个方法，但在此之前却可以先编写测试检测它的功能是否按设想实现了。测试的关键是要覆盖三种情况：动态列表中要包含被关注用户和用户自己的微博，而且不能包含未被关注用户的微博。前面的测试已经检测了其中两种情况：代码 10.35 验证了动态列表中有用户自己的微博，而且没有未被关注用户的微博。既然我们已经实现了关注用户功能，那就加入对第三种情况的测试吧，检测动态列表中是否包含了被关注用户的微博，如代码 11.41 所示。

图 **11.19**：id 为 1 的用户关注了 id 为 2，7，8，10 的用户后得到的动态列表

代码 11.41：针对动态列表测试的最终版
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  .
  .
  .
  describe "micropost associations" do
    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end
    .
    .
    .
    describe "status" do
      let(:unfollowed_post) do
        FactoryGirl.create(:micropost, user: FactoryGirl.create(:user))
      end
      let(:followed_user) { FactoryGirl.create(:user) }
```

```
      before do
        @user.follow!(followed_user)
        3.times { followed_user.microposts.create!(content: "Lorem ipsum") }
      end

      its(:feed) { should include(newer_micropost) }
      its(:feed) { should include(older_micropost) }
      its(:feed) { should_not include(unfollowed_post) }
      its(:feed) do
        followed_user.microposts.each do |micropost|
          should include(micropost)
        end
      end
    end
  end
  .
  .
  .
end
```

我们要把动态列表的实现交给 `Micropost.from_users_followed_by` 方法，如代码 11.42 所示。

代码 11.42： 在 User 模型中加入实现动态列表的方法
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def feed
    Micropost.from_users_followed_by(self)
  end
  .
  .
  .
end
```

11.3.2 初步实现动态列表

现在我们要来定义 `Micropost.from_users_followed_by` 方法了，为了行文简洁，在后面的内容中我会使用“动态列表”指代这个方法。因为要实现的结果有点复杂，因此我们会一点一点的说明动态列表的实现过程。

首先，我们要知道需要使用怎样的查询语句。我们要做的是，从 `microposts` 表中取出被关注用户发布的微博（也要取出用户自己的微博）。对此，我们可以使用类似下面的查询语句：

```
SELECT * FROM microposts
WHERE user_id IN (<list of ids>) OR user_id = <user id>
```

写下这个查询语句时，我们假设 SQL 支持使用 `IN` 关键字检测集合是否包含指定的元素。（还好，SQL 支持。）

在 10.3.3 节实现临时动态列表时，是调用 Active Record 中的 `where` 方法完成上面这种查询的，如代码 10.36 所示。这段代码中用到的查询很简单，只是通过当前用户的 id 取出了他发布的微博：

```
Micropost.where("user_id = ?", id)
```

而现在，我们遇到的情况复杂的多，要使用类似下面的代码实现：

```
where("user_id in (?) OR user_id = ?", following_ids, user)
```

（在指定查询条件时，我们使用了 Rails 中的约定，用 `user` 代替 `user.id`，Rails 会自动获取用户的 `id`。我们还省略了方法的调用者 `Micropost`，因为我们只会在 Micropost 模型中使用这个方法。）

从上面的查询条件可以看出，我们需要生成一个数组，其元素为被关注用户的 id。生成这个数组的方法之一是，使用 Ruby 中的 `map` 方法，这个方法可以在任意的“可枚举（enumerable）”对象上调用，例如包含了元素的集合类对象（数组，Hash 等）。[11]我们在 4.3.2 节中举例介绍过这个方法，其用法如下：

```
$ rails console
>> [1, 2, 3, 4].map { |i| i.to_s }
=> ["1", "2", "3", "4"]
```

像上面这种在每个元素上调用同一个方法的情况是很常见的，所以 Ruby 为此定义了一种简写形式，在 `&` 符号后面跟上被调用方法的 Symbol 形式：[12]

```
>> [1, 2, 3, 4].map(&:to_s)
=> ["1", "2", "3", "4"]
```

然后再调用 `join` 方法（参见 4.3.1 节），就可以将数组中的元素合并起来组成字符串，各元素之间用逗号加一个空格分开：

```
>> [1, 2, 3, 4].map(&:to_s).join(', ')
=> "1, 2, 3, 4"
```

参照上面介绍的方法，我们可以在 `user.followed_users` 的每个元素上调用 `id` 方法，得到一个由被关注用户的 id 组成的数组。例如，对数据库中第 1 个用户而言，就可以用下面的代码实现：

```
>> User.first.followed_users.map(&:id)
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
```

11. 可枚举的对象最基本的要求是必须实现 `each` 方法，用来遍历集合。
12. 这个简写方式最初是 Rails 对 Ruby 的扩展，因为很有用，所以 Ruby 最后自己也实现了。很牛吧。

```
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

其实，因为这种用法太普遍了，所以 Active Record 默认已经提供了：

```
>> User.first.followed_user_ids
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

上述代码中的 `followed_user_ids` 方法是 Active Record 根据 `has_many :followed_users` 关联（参见代码 11.10）合成的，这样我们只需在关联名的后面加上 `_ids` 就可以获取 `user.followed_users` 集合中所有用户的 id 了。用户 id 组成的字符串如下：

```
>> User.first.followed_user_ids.join(', ')
=> "4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51"
```

不过，当插入 SQL 语句中时，你无须手动生成字符串，`?` 插值操作会为你代劳（同时也避免了一些数据库之间的兼容问题）。所以，我们实际要使用的只是 `user.followed_user_ids` 而已。

现在这个阶段，你可能已经想到了，要使用 `Micropost.from_users_followed_by(user)`，就要在 `Micropost` 类中定义一个类方法（4.4.1 节简单的介绍过）。这个类方法的初步定义如代码 11.43 所示，其中包含了上述分析得到的一些代码：

代码 11.43： `from_users_followed_by` 方法的初步定义
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
  def self.from_users_followed_by(user)
    followed_user_ids = user.followed_user_ids
    where("user_id IN (?) OR user_id = ?", followed_user_ids, user)
  end
end
```

虽然代码 11.43 之前的讨论都是以假设为基础进行的，但得到的代码却是可用的。你可以运行测试验证一下，应该是可以通过的：

```
$ bundle exec rspec spec/
```

在某些程序中，这样的初步实现已经能满足大部分需求了。不过这不是我们最终要使用的实现方式，在继续阅读下一小节之前，你可以想一下为什么。（提示：如果用户关注了 5000 个用户呢？）

11.3.3 子查询（subselect）

如上一小节末尾中所说的，11.3.2 节的实现方式，当动态列表中的微博数量很多时性能就会下降，这种情况可能会发生在用户关注了 5000 个用户后。本小节，我们会重新实现动态列表，在关注用户数量很多时，性能表现也会很好。

11.3.2 节中所用代码的问题在于`followed_user_ids = user.followed_user_ids`这行代码，它会把所有被关注用户的 id 取出存入内存，然后再创建一个元素数量和被关注用户数量相同的数组。既然代码 11.43 的目的只是为了检查集合是否包含了指定的元素，那么就一定存在一种更高效的方法，其实 SQL 真的提供了针对这种问题的优化措施：使用子查询把查询被关注用户 id 的操作放入数据库层进行。

针对动态列表的重构，先从代码 11.44 中的小改动开始。

代码 11.44：改进 `from_users_followed_by` 方法
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
  # Returns microposts from the users being followed by the given user.
  def self.from_users_followed_by(user)
    followed_user_ids = user.followed_user_ids
    where("user_id IN (:followed_user_ids) OR user_id = :user_id",
          followed_user_ids: followed_user_ids, user_id: user)
  end
end
```

为了给下一步做准备，我们把

```
where("user_id IN (?) OR user_id = ?", followed_user_ids, user)
```

换成了等效的

```
where("user_id IN (:followed_user_ids) OR user_id = :user_id",
      followed_user_ids: followed_user_ids, user_id: user))
```

使用问号做插值虽然可以，但当我们要在多处插入同一个值时，后一种写法就方便多了。

上面这段话表明，我们要在 SQL 查询语句中两次用到 `user_id`。简单来说就是，我们要把下面这行 Ruby 代码

```
followed_user_ids = user.followed_user_ids
```

换成包含 SQL 语句的代码

```
followed_user_ids = "SELECT followed_id FROM relationships
                     WHERE follower_id = :user_id"
```

上面这行代码使用了 SQL 的子查询语句，那么针对 id 为 1 的用户，整个查询语句就可以写成：

```
SELECT * FROM microposts
WHERE user_id IN (SELECT followed_id FROM relationships
                  WHERE follower_id = 1)
      OR user_id = 1
```

使用子查询后，所有的集合包含关系都会交由数据库处理，这样性能就得到提升了。[13]

有了这些基础，我们就可以着手实现更高效的动态列表了，如代码 11.45 所示。注意，因为现在要使用纯 SQL 语句，所以 `followed_user_ids` 是被插值进语句中的，而没有通过转义的方式。（其实两种方式都可以使用，只不过这种情况使用插值操作更合理。）

代码 11.45： `from_users_followed_by` 方法的最终版本
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order('created_at DESC') }
  validates :content, presence: true, length: { maximum: 140 }
  validates :user_id, presence: true

  # Returns microposts from the users being followed by the given user.
  def self.from_users_followed_by(user)
    followed_user_ids = "SELECT followed_id FROM relationships
                         WHERE follower_id = :user_id"
    where("user_id IN (#{followed_user_ids}) OR user_id = :user_id",
          user_id: user.id)
  end
end
```

这段代码结合了 Rails、Ruby 和 SQL 的优势，达到了目的，而且做的很好。（当然，子查询也不是万能的。对于更大型的网站而言，可能要使用“后台作业（background job）”异步生成动态列表。性能优化这个话题已经超出了本书的范围。）

11.3.4 新的动态列表

编完代码 11.45，动态列表功能就实现了。提醒一下，`home` 动作所需的代码如代码 11.46 所示，这段代码生成了一个可分页显示的动态列表变量，可在视图中使用，如图 11.20 所示。[14]注意，`paginate` 最终会调用代码 11.45 中定义的类方法，一次只从数据库中取出 30 篇微博。（如果你想验证一下，可以查看“开发服务器”的日志。）

代码 11.46： 定义了可分页动态列表的 `home` 动作
`app/controllers/static_pages_controller.rb`

13. 创建子查询语句更多高级的方式，请阅读《Hacking a subselect in ActiveRecord》一文。
14. 为了图 11.20 中显示的动态列表更好看，我自己动手在 Rails 控制台中加入了一些微博。

```
class StaticPagesController < ApplicationController

  def home
    if signed_in?
      @micropost  = current_user.microposts.build
      @feed_items = current_user.feed.paginate(page: params[:page])
    end
  end
  .
  .
  .
end
```

11.4 小结

实现了动态列表后，本书的核心示例程序就开发完了。这个程序演示了 Rails 全部的主要功能，包括模型，视图，控制器，模板，局部视图，过滤器，数据验证，回调函数，`has_many/belongs_to` 和 `has_many through` 关联，安全，测试，以及部署。除此之外，Rails 还有很多功能值得我们学习。你可以把本节作为后续学习的第一站，下面几小节介绍了可以对示例程序进行的功能扩展，也对后续学习提供一些参考资源。

图 11.20：显示了动态列表的首页

在介绍可对程序进行的功能扩展之前，最好先把本章的改动合并到主分支：

```
$ git add .
$ git commit -m "Add user following"
$ git checkout master
$ git merge following-users
```

和之前一样，你也可以把代码推送到 GitHub，还可以部署到 Heroku：

```
$ git push
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:populate
```

11.4.1 扩展示例程序的功能

本节建议的功能扩展灵感基本上都是来自常规 Web 程序的一般功能，如密码提醒和 Email 地址确认，或是来自同类型程序，例如搜索，回复和私信。自己实现几个功能扩展，可以让你从教程的跟学者变成自主程序开发者。

如果开始时觉得有点难也不用奇怪，从零开始实现一个功能确实有难度。为了帮助你，我可以提两点建设性的建议。第一点，在开始实现新功能之前，浏览一下 RailsCasts 的归档，看一下 Ryan 是否介绍过类似的功能。[15]如果他介绍过，先看一下相关的视频会节省很多时间。第二点，总是在 Google 中大范围的搜索你要实现的功能，寻找相关的博文和教程。Web 程序开发是有难度的，从有经验的开发者那里取经总是会对你有所帮助的。

下面列出的功能很多都是有一定挑战性的，在功能的介绍中我会给你一定的提示，告诉你实现过程中可能会用到的工具。虽然有提示，但是这些功能实现起来比章后的练习难多了，在没下真功夫之前千万别轻言放弃。因为时间有限，我无法一对一的辅导，不过如果你对这些功能感兴趣，将来我可能会发布一些独立的文章或视频介绍一下，请到本书的网站 http://railstutorial.org/ 订阅 Feed 获取最新的更新。

回复

Twitter 允许用户使用“@replies” 的格式进行回复，回复也是一篇微博，不过内容的开头是 `@` 符号加用户名。回复只会出现在被回复用户的动态列表和粉丝的动态列表中。请实现一个简化的回复功能，限制回复只可以出现在接收者和发送者的动态列表中。实现的过程可能要在 `microposts` 表中加入 `in_reply_to` 列，还要在 Micropost 模型中添加 `including_replies` 作用域。

因为我们的示例程序没有限制用户登录名要是唯一的，所以你可能要决定一下要采用什么方式表示用户的身份。一种方式是，结合 id 和名字，例如 `@1-michael-hartl`。另一种方式是，在注册表单中添加一个用户名字段，用户名将是唯一的，然后用来表示 @replies。

15. 注意，RailsCasts 经常会省略测试，可能是为了要控制视频的质量和长度，这可能会让你会错意，认为测试并不重要。观看过 RailsCasts 中相关的视频知道怎么开发后，我建议你按照“测试驱动开发”原则实现功能。（我建议你看一下 RailsCasts 中的《How I test》，你会看到 Ryan Bates 在实际的开发中经常会使用 TDD，而且他的测试风格和本书的很像。）

私信

Twitter 支持在微博的前面加上字母“d”发送私信。请在示例程序中加入这个功能。实现的过程中可能要创建 Message 模型，还要使用正则表达式匹配微博的内容。

被关注提醒

请实现当用户有新粉丝时向被关注用户发送提醒邮件的功能，并把这一功能设为可选的，这样如果用户不想接收提醒就可以不选择这个功能。实现这个功能需要学习如何在 Rails 中发送邮件，我建议观看 RailsCasts 中的《Action Mailer in Rails 3》一集来学习。

密码提醒

现在，如果程序的用户忘记密码了，就没办法获取了。因为我们在第 6 章使用了单向密码加密，程序没办法把密码通过 Email 发送给用户，但是我们可以发送一个重设密码表单的链接。按照 RailsCasts 中的《Remember Me & Reset Password》一集来修正这个问题。

注册确认

除了匹配 Email 地址的正则表达式之外，示例程序现在没有其他方法可以验证用户的 Email 地址是否合法。请在注册步骤中添加确认用户注册这一步。这个功能应该在注册时把用户设为未激活状态，发送一封包含激活链接的邮件，当链接被点击后再把用户设为已激活状态。你可能要先阅读 Rails state machine 相关的文章，学习如何在未激活和激活状态之间转换。

RSS Feed

请为每一个用户的微博更新创建一个 RSS，然后再为用户的状态列表实现一个 RSS，如果可以，你还可以使用身份验证机制限制对动态列表 RSS 的访问。RailsCasts 中的《Generating RSS Feeds》一集可以给你一些帮助。

REST API

很多网站都提供了“应用编程接口（Application Programmer Interface，API）”，允许第三方程序获取（get），创建（post），更新（put）和删除（delete）程序的资源。请为示例程序实现这种 REST API。实现的过程中可能要为程序的多数控制器动作添加 `respond_to` 代码块（参见 11.2.5 节），响应 XML 类型的请求。请注意安全问题，API 应该只对授权的用户开放。

搜索

现在，除了浏览用户索引页面，或者查看其他用户的动态列表之外，没有办法找到另外的用户。请实现搜索功能来弥补这个缺陷。然后再添加搜索微博的功能。RailsCasts 中的《Simple Search Form》一集可以给你一些帮助。如果你的程序部署在共享主机或专用服务器，我建议你使用 Thinking Sphinx（参考 RailsCasts 中的《Thinking Sphinx》一集）。如果你的程序部署在 Heroku 上，你应该参照《Full Text Search Options on Heroku》一文中的说明。

11.4.2 后续学习的资源

商店和网上都有很多 Rails 资源，而且多得会让你挑花眼。可喜的是，你阅读到这里时，已经可以学习几乎所有的其他知识了。下面是建议你后续学习的资源：

- 本书配套视频：我为本书录制了内容充足的配套视频，除了覆盖本书的内容之外，在视频中我还介绍了很多小技巧，当然视频还能弥补印刷书的不足，让你观看别人是如何开发的。你可以在本书的网站上购买这些视频。
- RailsCasts：如何强调 RailsCasts 的重要性都不为过。我建议你浏览一下 RailsCasts 的视频归档，观看你感兴趣的视频。
- Ruby 和 Rails 相关的书：学习 Ruby 我推荐 Peter Cooper 的《Ruby 入门》，David A. Black 的《The Well-Grounded Rubyist》，Russ Olsen 的《Eloquent Ruby》和 Hal Fulton 的《Ruby 之道》。继续学习 Rails 我推荐 Obie Fernandez 的《Rails 3 之道》和 Ryan Bigg、Yehuda Katz 合著的《Rails 3 实战》（请阅读第 2 版）。
- PeepCode 和 Code School：PeepCode 的视频教程和 Code School 的交互教程质量都很高，我真心地向你推荐。

11.5 练习

1. 添加针对销毁指定用户关注关联（通过代码 11.4 和代码 11.16 中的 `dependent :destroy` 实现）的测试。提示：可参照代码 10.12。

2. 代码 11.38 中用到的 `respond_to` 方法可以提取出来放入 Relationships 控制器中，而且 `respond_to` 代码块可以使用 Rails 中的 `respond_with` 方法代替。请运行测试组件确认上面两个操作实施后得到的代码（如代码 11.47 所示）仍是正确的。（`respond_with` 方法的详细用法，请在 Google 中搜索“rails respond_with”。）

3. 重构代码 11.31，把关注者列表页面、粉丝列表页面，首页和用户资料页面的通用部分提取出来，创建成局部视图。

4. 按照代码 11.19 中的方式，编写测试检测个人资料页面的关注数量统计。

代码 11.47：重构，把代码 11.38 变得更简洁

```
class RelationshipsController < ApplicationController
  before_action :signed_in_user

  respond_to :html, :js

  def create
    @user = User.find(params[:relationship][:followed_id])
    current_user.follow!(@user)
    respond_with @user
  end
```

```
  def destroy
    @user = Relationship.find(params[:id]).followed
    current_user.unfollow!(@user)
    respond_with @user
  end
end
```