原书第 6 版 · 涵盖 Rails 6

Ruby on Rails 教程

Ruby on Rails Tutorial 中文版

[美] Michael Hartl 著 • 安道 译

笨鸟码书

原书第 6 版

Ruby on Rails 教程

通过 Rails 学习 Web 开发

Michael Hartl 著

安道 译

rev 6.1.3.p1, 2021-03-19T20:00:00+08:00

目录

致中国读者

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

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

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

衷心的祝福你们，

Michael Hartl
《Ruby on Rails Tutorial》的作者

附原文：

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

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

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

Best wishes and good luck,

Michael Hartl
Author
The Ruby on Rails Tutorial

序

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

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

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

享受这本书吧！

Derek Sivers (sivers.org)
CD Baby 创始人

致谢

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

我要感谢很多 Ruby 高手，在过去这些年，他们教我知识，也给我启迪。他们是：David Heinemeier Hansson，Yehuda Katz，Carl Lerche，Jeremy Kemper，Xavier Noria，Ryan Bates，Geoffrey Grosenbach，Peter Cooper，Matt Aimonetti，Mark Bates，Gregg Pollack，Wayne E. Seguin，Amy Hoy，Dave Chelimsky，Pat Maddox，Tom Preston-Werner，Chris Wanstrath，Chad Fowler，Josh Susser，Obie Fernandez，Ian McFarland，Steph Bristol，Pratik Naik，Sarah Mei，Sarah Allen，Wolfram Arnold，Alex Chaffee，Giles Bowkett，Evan Dorn，Long Nguyen，James Lindenbaum，Adam Wiggins，Tikhon Bernstam，Ron Evans，Wyatt Greene，Miles Forrest，Tikhon Bernstam，Ron Evans，Wyatt Greene，Miles Forrest，Sandi Metz，Ryan Davis，Aaron Patterson，Aja Hammerly，Richard “Schneems” Schneeman，Pivotal Labs 公司的好心人们，Heroku 团队，thoughtbot 公司的小伙伴，以及 GitHub 的全体员工。

最后，还有很多很多读者（太多了，无法一一列举）在本书写作过程中反馈了众多问题，还给了我很多建议，我由衷地感谢这些人的帮助，让这本书变得更好。

作者简介

本书作者 Michael Hartl（https://michaelhartl.com）是把 Ruby on Rails Web 开发介绍给世人的先行者之一，他还是 LearnEnough.com 的创始人和校长。Michael 曾任加州理工学院的物理学导师，因成绩卓越，被授予“突出教学终身成就奖”。他毕业于哈佛学院，在加州理工学院获得了物理学博士学位。他还是 Y Combinator 创业者项目的毕业生。

版权声明和代码许可证

第 1 章 从零到部署

欢迎您打开本书。这是一本教程，旨在教您开发 Web 应用的方法。读完本书后，您将掌握足够的知识，可以胜任 Web 开发者的工作、可以开启自由就业的道路，也可以选择创业，创办自己的公司。如果您已经知道如何开发 Web 应用，本书将带领您快速掌握 Ruby on Rails。

本书的重点是教您通用的技能，不管您最终选择什么技术，都能用得上。充分理解 Web 应用的相关理论之后，无论学习什么框架，都将事半功倍。不过，本书着眼于众多框架中的一个——Ruby on Rails（https://rubyonrails.org），这是学习 Web 开发的不二选择（旁注 1.1）。

旁注 1.1：Rails 的众多优势

Ruby on Rails（简称 Rails）是一个 Web 开发框架，使用 Ruby（https://www.ruby-lang.org）编程语言开发，免费、开源。问世之后，Rails 迅速成为动态 Web 应用开发领域最受欢迎的框架之一。使用 Rails 的公司有很多，例如 Airbnb、SoundCloud、Disney、Hulu、GitHub 和 Shopify。此外，还有很多自由职业者、独立开发工作室和初创公司在用。

Web 开发的选择太多了，Rails 脱引而出归功于它优雅、强大、集成式的设计。即便是新手，在不脱离 Rails 框架的前提下也能构建出一个全栈 Web 应用——这对初学 Web 开发的人来说可谓一大亮点。另外，Rails 足够灵活，如果你想向前迈一步，它不会阻止你。假如你的兴趣在于构建单页应用或移动应用，Rails 也能做好后端工作。

Rails 的一大优势是，不像某些开发社区（尤其是 JavaScript/Node.js），没有新技术层出不穷的问题。Rails 的创始人 David Heinemeier Hansson（https://dhh.dk）说过：

> 过去，没有认清局面的商人选择了 J2EE，让人怨声载道；声未绝耳，这边又冒出个 JavaScript……Rails 的核心理念自出现的那一刻起就饱受争议，如今依然如此。Rails 把约定摆在第一位，摒除了无价值的选择，为创建一个完整的应用提供了众人皆需的默认设置，极大地提高了生产力。

就是因为坚守这样的理念，Rails 才能保持初心，一往无前。本书从 2014 年出版的第 3 版开始，也受益于这份坚持，才能让书中的内容不至快速过时。

此外，Rails 也一直在引入新鲜血液。例如，Rails 6 包含了一些重要的新特性，包括电子邮件路由、文本格式化、并行测试和多数据库支持。Rails 6 的一大变化是“默认可弹性伸缩”（https://youtu.be/8evXWvM4oXM），不管应用的规模有多大，Rails 都能招架得住。新特性的引入并没有打乱原有局面，其实，深受欢迎的开发者平台 GitHub、大获成功的在线商店搭建平台 Shopify，以及协作工具 Basecamp（第一个 Rails 应用）都使用过 Rails 的预发布版本。Rails 新版本有世界上最大型、最成功的 Web 应用测试，您还有什么不放心的呢！

2004 年，Rails 只是一位来自丹麦的自由 Web 开发者捣鼓出来的业余项目。可谁曾想，经过这些年的发展，Rails 取得了傲人的成绩，功能不断丰富、社区逐渐壮大。如今，Rails 框架已成为构建现代化 Web 应用的绝佳选择。

阅读本书不要求你事先掌握特定的知识。本书不仅介绍 Rails，还涉及底层的 Ruby 语言、Unix 命令行、HTML、CSS、少量的 JavaScript，以及一点 SQL。我们要学习的知识很多，笔者建议先阅读 Learn Enough 系列教程（https://www.learnenough.com），尤其是 Learn Enough Command Line to Be Dangerous 和 Learn Enough Ruby to Be Dangerous。不过，有很多什么都不懂的新手曾说，他们在阅读本书时没感到有什么障碍。如果你对 Web 应用开发感兴趣，不要被危言耸听的传闻吓退。

本书的一大特色是在实践中学习，我们将由简至深构建几个应用，首先是一个极简的 `hello_app`（图 1.1，1.2 节），然后增加一点功能，构建 `toy_app`（图 1.2，第 2 章），最后开发一个完整的应用 `sample_app`（图 1.3，从第 3 章到第 14 章）。从名称可以看出，这些应用关注的是一般性原则，可在各种 Web 应用中运用。最后那个完整的示例应用涵盖专业级 Web 应用所需的全部重要特性，包括用户注册、登录和账户管理。经过一番开发之后，这个应用在第 14 章达到最终状态，在功能上与 Twitter 相似（巧的是，Twitter 最初也是使用 Rails 开发的）。

下面开始我们的旅程！

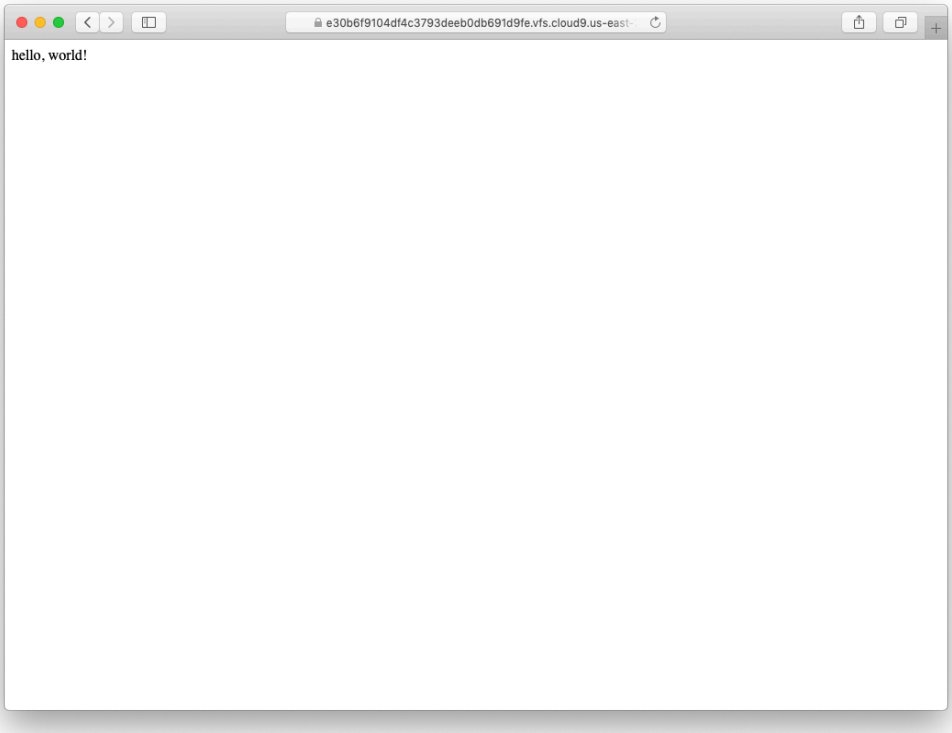


图 1.1：第一个应用 `hello_app`

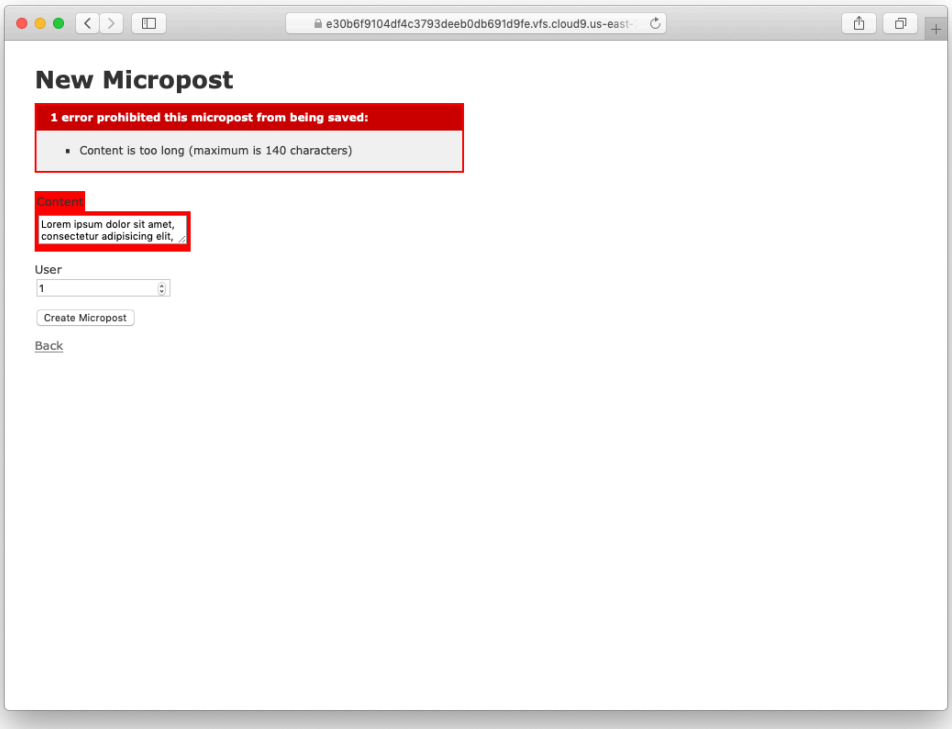


图 1.2：第二个应用 `toy_app`

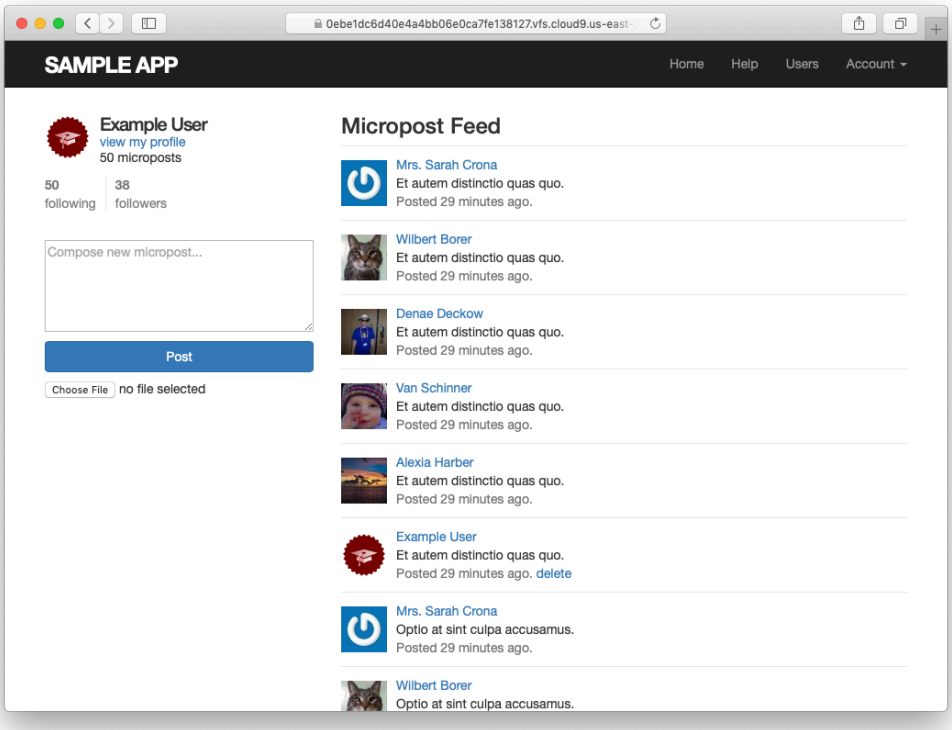


图 1.3：最后一个应用 `sample_app`

1.1 搭建环境

本书的一个特色是能让你快速上手。我们与 AWS Cloud9（https://aws.amazon.com/cloud9/）建立了长期伙伴关系，这是一个在浏览器中运行的开发环境，提供了本书所需的全部开发软件。

这一点十分重要，就算对有经验的开发者来说，安装 Ruby、Rails，以及相关的所有软件，也要几经波折。这些问题是由环境的多样性导致的。不同的操作系统、版本号、文本编辑器等，都会导致环境有所不同。

因此，推荐的方案是（尤其是对初学者来说），使用云集成开发环境（Integrated Development Environment，IDE，见 1.1.1 节），规避安装和配置问题。本书使用的云 IDE 运行在普通的 Web 浏览器中，因此在不同的平台中表现一致。此外，这个云 IDE 还能保存当前的工作状态，我们可以休息一会，然后再从离开的地方继续学习。

另外一种选择是自己动手在原生系统（Windows、macOS 或 Linux）中为 Rails 开发搭建环境。最终，这将是不二选择，但就目前来说，有一定的挑战，毕竟技术是复杂的（旁注 1.2）。

旁注 1.2：技术是复杂的

本书是 Learn Enough 系列教程的一部分。这一系列教程力图让你认清“技术是复杂的”，不要以为软硬技能兼备就能解决任何技术问题（见 xkcd 网站中的“Tech Support Cheat Sheet”，https://m.xkcd.com/627/）。

Web 开发，以及一般的计算机编程，最能体现这一点。除了技术本身，还要知道如何点击菜单项，学会使用某个应用；要知道如何使用 Google，弄清错误消息的意思；以及何时该放弃，直接重启设备。

Web 应用涉及的知识特别多，因此从中更容易认清“技术是复杂的”。就使用 Rails 做 Web 开发而言，体现这一点的包括：确保使用的 Ruby gem 版本正确，执行 `bundle install` 或 `bundle update` 命令，以及遇到问题时重启本地 Web 服务器。（如果你完全不知道我在说什么，不用担心，本书会涵盖这里提到的所有话题。）

在阅读本书的过程中，你可能偶尔会卡住，得不到预期的结果。书中的文字已经着重标出了容易出错的步骤，但是难免百密一疏。笔者建议你积极去解决这些拦路虎，借此提高自己的技能。说不定，就像极客们所说的：这不是缺陷，而是特性!

1.1.1 开发环境

不同的人有不同的喜好，每个 Rails 程序员都有自己的一套开发环境。为了避免问题复杂化，本书使用一个标准的云开发环境——Cloud9（隶属 Amazon Web Services，即 AWS）。这个开发环境预装了 Rails 开发所需的大多数软件，包括 Ruby、RubyGems 和 Git。（其实，唯有 Rails 要单独安装，而且这么做是有目的的，详见 1.1.2 节。）

这个云 IDE 包含 Web 应用开发所需的三个基本组件：命令行终端、文件系统浏览器和文本编辑器（图 1.4）。这个云 IDE 中的文本编辑器功能很多，其中一项是“在文件中查找”的全局搜索功能，笔者觉得这个功能对大型 Rails 项目来说是必备的。即便最终在实际开发中你不使用云 IDE（笔者始终建议不断学习其他工具），通过它也能了解文本编辑器和其他开发工具的基本功能。

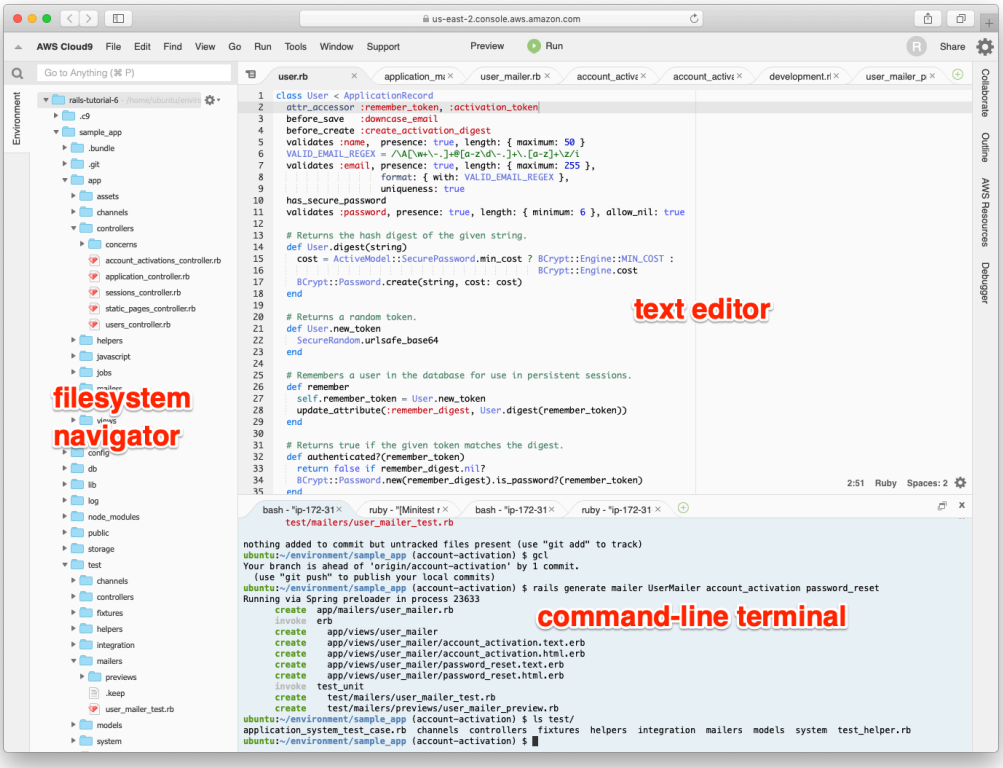


图 1.4：云端 IDE 的界面布局

这个云开发环境的使用步骤如下：[1]

1. Cloud9 隶属 Amazon Web Services（AWS），如果你已有 AWS 账户，可以直接登录。为了新建 Cloud9 工作区环境，访问 AWS 控制台（https://console.aws.amazon.com/），在搜索框中输入“Cloud9”。
2. 如果你没有 AWS 账户，可以到 AWS Cloud9 网站中注册一个免费账户（https://portal.aws.amazon.com/billing/signup/）。未免被滥用，AWS 要求注册时提供有效的信用卡，不过工作区是 100% 免费的（写作本书时是免费一年），不会从信用卡上扣款。可能要等 24 小时待账户激活，不过笔者就等了十多分钟。
3. 进入 Cloud9 管理页面后（图 1.5）后，点击“Create environment”（创建环境），填写图 1.6 中的信息：在“Name”（名称）栏中填入“rails-tutorial”，“Description”（说明）栏可参照图中给出的内容填写。在下一页中选择“**Ubuntu Server**”（别选“Amazon Linux”，图 1.7），然后点“Next step”（下一步）。点击确认按钮，接受默认设置，然后等待 AWS 配置 IDE（图 1.9）。

你可能会遇到一个提醒消息，说需要“root”用户，目前可以放心忽略。[我们将在 13.4.4 节讨论推荐但较为复杂的做法，叫做 Identity and Access Management（IAM）用户。]

因为使用两个空格缩进几乎是 Ruby 圈通用的约定，所以笔者建议修改编辑器的配置，把默认的四个空格改为两个。配置方法是，点击右上角的齿轮图标，然后按“Soft Tabs”旁的减号，直到变成 2，如图 1.10 所示。（注意，设置修改后立即生效，无需点击“Save”按钮。）

1. AWS 发展很快，网站经常变化，细节跟这里所讲的可能不同。如有差异，请设法解决。

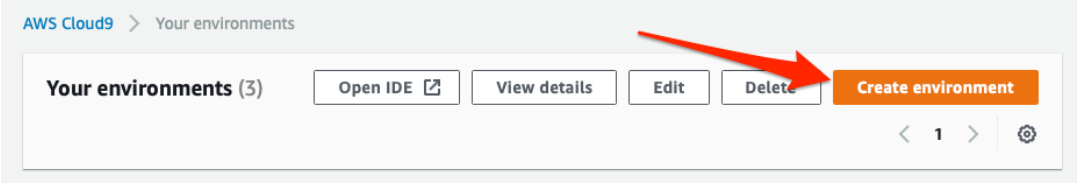


图 1.5：在 AWS Cloud9 中创建一个环境

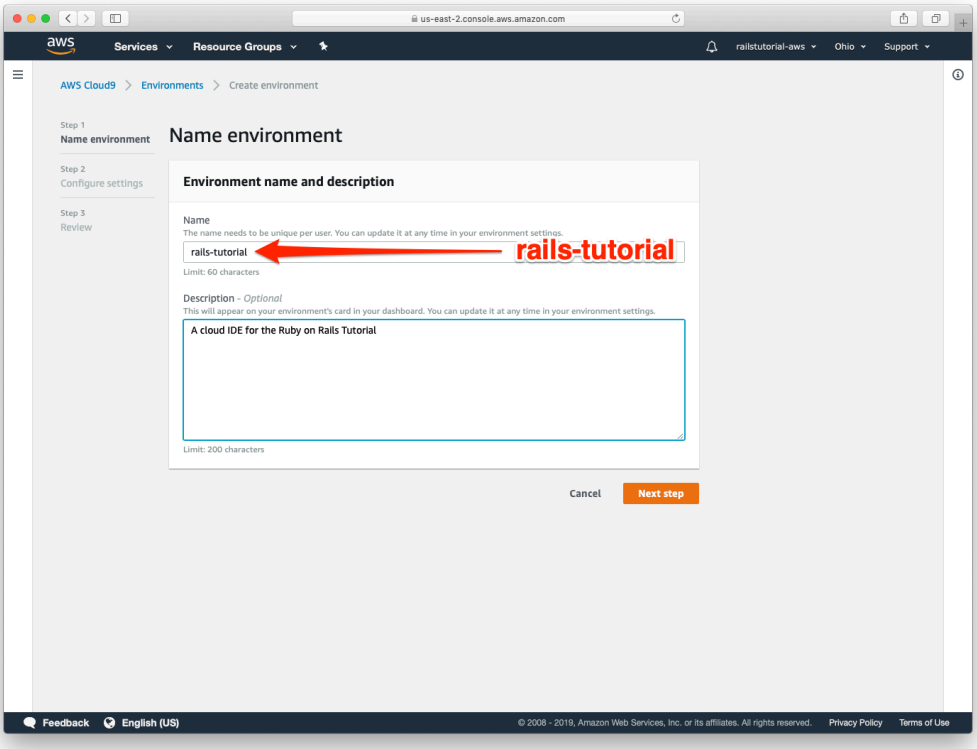


图 1.6：在 AWS Cloud9 中为新的工作环境命名

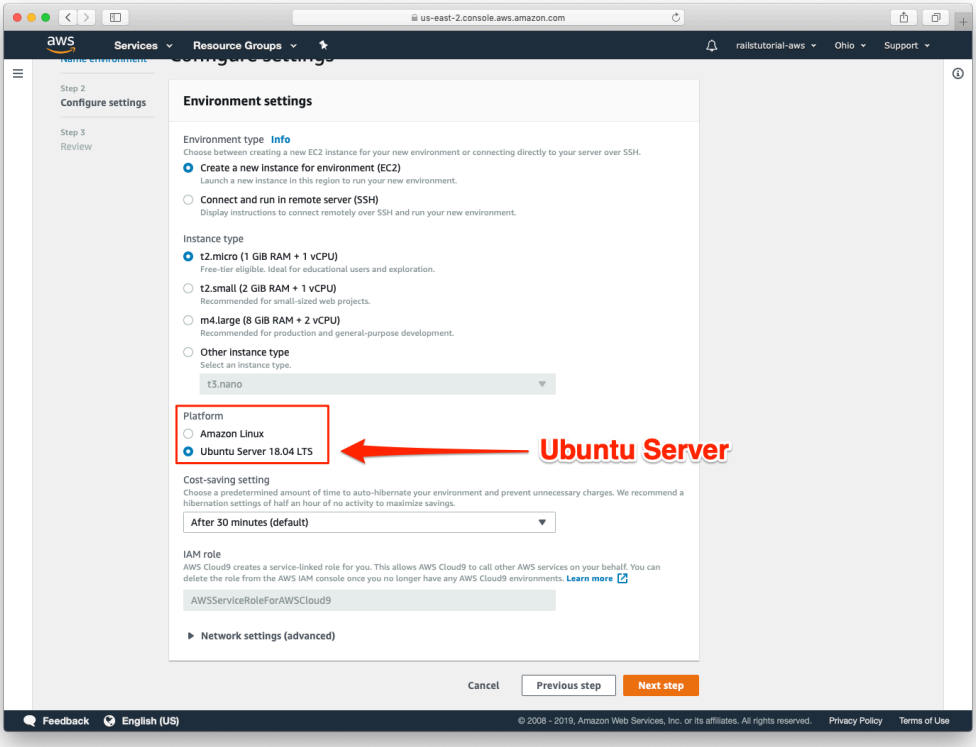


图 1.7：选择“Ubuntu Server”

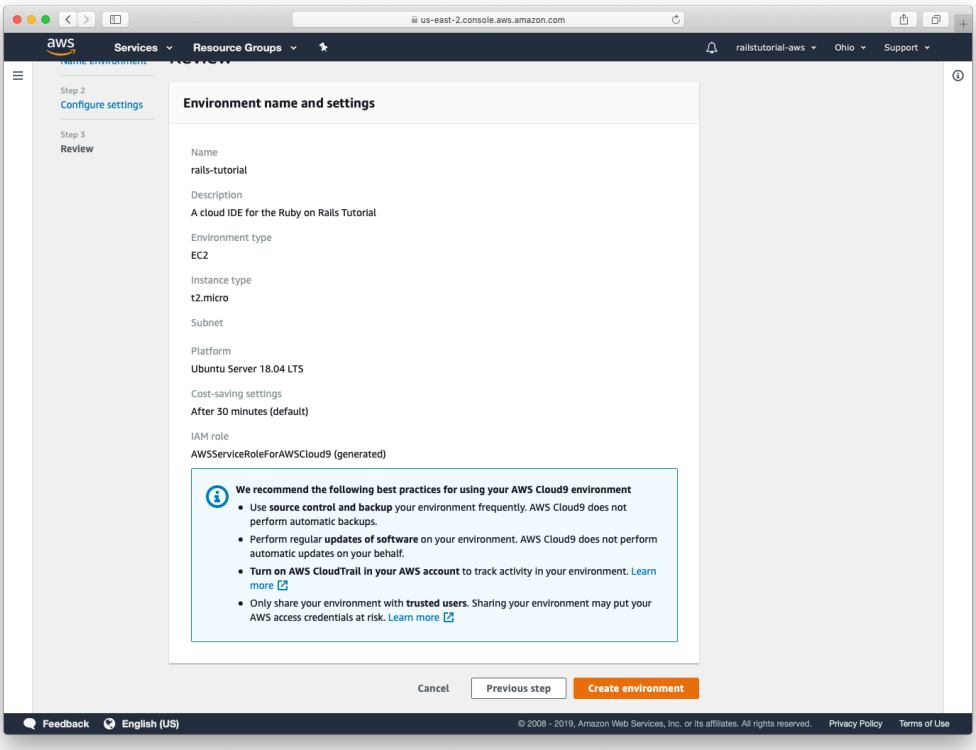


图 1.8：开始配置 IDE 前的最后一步

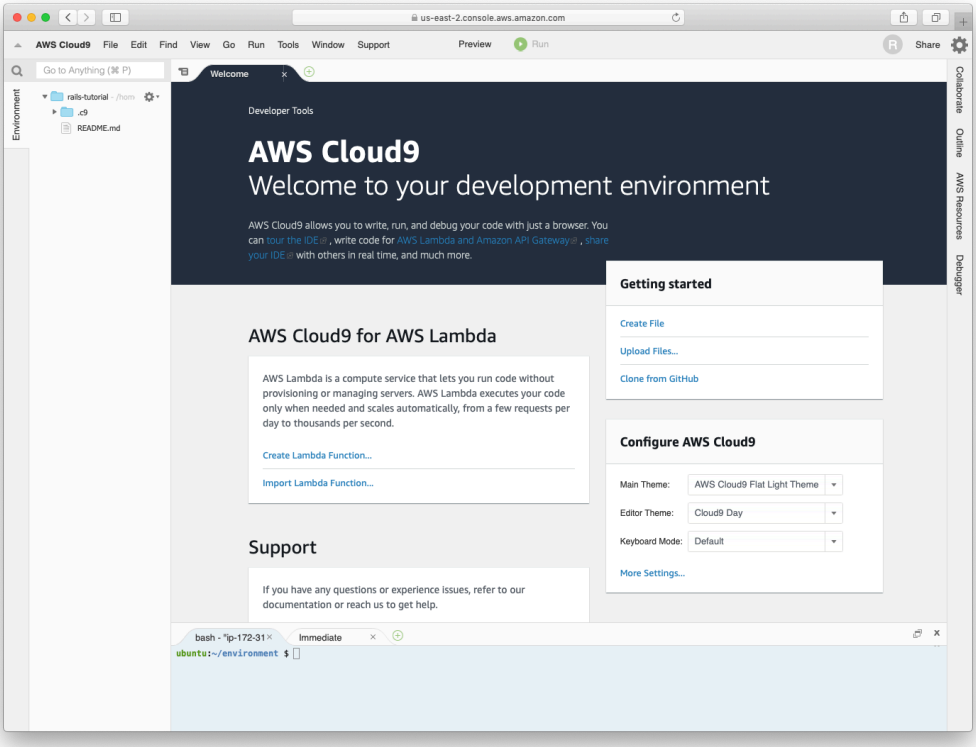


图 1.9：Cloud9 云 IDE 的默认界面

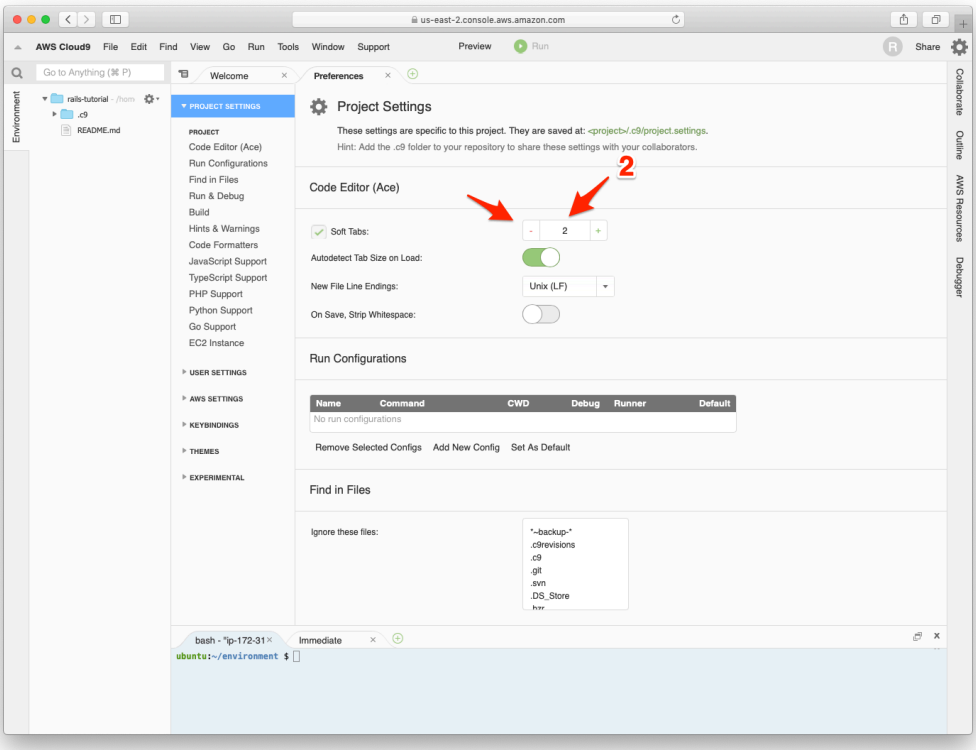


图 1.10：设置 Cloud9，使用两个空格缩进

1.1.2 安装 Rails

前一节创建的开发环境包含所有软件，但没有 Rails。这是有意为之的，因为你必须安装与书中一样的 Rails 版本才能得到预期的结果。

首先要做些准备工作，以免浪费时间在本地安装 Ruby 文档，如代码清单 1.1 所示。[2] 这项配置每个系统只需做一次。（关于命令行和其他排版约定，见 1.6 节。）

代码清单 1.1：配置 .gemrc 文件，禁止安装 Ruby 文档

```
$ echo "gem: --no-document" >> ~/.gemrc
```

为了安装 Rails，我们要使用包管理器 RubyGems 提供的 `gem` 命令，在命令行终端里输入代码清单 1.2 所示的命令。（如果在本地系统中开发，在终端窗口中输入这个命令；如果使用云 IDE，在图 1.4 中的命令行区域输入这个命令。）

代码清单 1.2：安装指定版本的 Rails

```
$ gem install rails -v 6.1.3
```

`-v` 标志的作用是指定安装哪个 Rails 版本。然后，把 `-v` 标志传给 `rails` 命令，确认安装是否成功：

```
$ rails -v
Rails 6.1.3
```

这个命令输出的版本号应该与代码清单 1.2 中安装的版本一致。

经验表明，使用标准版 `bundler` gem 也是个好主意，安装方法如下：

代码清单 1.3：安装 Bundler 的指定版本

```
$ gem install bundler -v 2.2.13
```

这个 gem 很重要，1.2.1 节详细说明。

此外，还要做一项配置，即安装软件依赖管理程序 Yarn（https://yarnpkg.com）。如果你使用自己的操作系统，请按照 Yarn 文档中的说明安装（https://yarnpkg.com/lang/en/docs/install/）。如果使用云 IDE，请运行下述命令，从 Learn Enough CDN 中下载并执行所需的命令：

代码清单 1.4：在云 IDE 中安装依赖管理程序 Yarn

```
$ source <(curl -sL https://cdn.learnenough.com/yarn_install)
```

如果得到类似下面的错误消息：

```
E: Could not get lock /var/lib/dpkg/lock-frontend - open
(11: Resource temporarily unavailable)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend),
is another process using it?
```

等上几秒钟，然后再执行代码清单 1.4 中的命令。

在开发 Rails 应用的过程中，你可能时不时看到如下的警告消息：

```
========================================
```

2. 这里用到了 `echo` 和 `>>` 命令。`>>` 命令足够智能，如果目标文件不存在，将新建一个。

```
  Your Yarn packages are out of date!
  Please run `yarn install --check-files` to update.
========================================
```

看到这样的警告，只需执行建议的 yarn 命令：

```
$ yarn install --check-files
```

至此，我们便搭建好了 Ruby on Rails Web 开发所需的完整环境。

1.2 第一个应用

按照计算机编程领域长期沿用的传统（http://www.catb.org/jargon/html/H/hello-world.html），第一个应用的目的是编写一个“hello, world”程序。具体来讲，我们将创建一个简单的应用，在网页中显示字符串“hello, world!”，在开发环境（1.2.4 节）和线上网站（1.4 节）中都是如此。

Rails 应用一般都从 `rails new` 命令开始，这个命令在你指定的目录中创建 Rails 应用的骨架。如果你没使用 1.1.1 节推荐的 Cloud9 IDE，首先要新建一个目录，命名为 `environment`，然后进入该目录，如代码清单 1.5 所示。

代码清单 1.5：为 Rails 项目新建一个目录，命名为 `environment`

```
# 在云 IDE 中不用做这一步
$ cd                  # 进入家目录
$ mkdir environment   # 新建 environment 目录
$ cd environment/     # 进入 environment 目录
```

代码清单 1.5 用到了 Unix 命令 `cd` 和 `mkdir`，如果你对这两个命令不熟悉，请阅读旁注 1.3。

旁注 1.3：Unix 命令行速成课

使用 Windows 和 macOS 的用户可能对 Unix 命令行不熟悉。如果使用推荐的云环境，你很幸运，这个环境提供了 Unix（Linux）命令行——在标准的 shell 命令行界面中运行的 Bash。

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

这是一门很深的学问，但在本书中只会用到一些最常用的 Unix 命令行命令，见表 1.1。若想更深入地学习 Unix 命令行，请阅读 Learn Enough 系列教程的第一本，Learn Enough Command Line to Be Dangerous（https://www.learnenough.com/command-line）。

表 1.1：一些常用的 Unix 命令

作用	命令	示例
列出内容	`ls`	`$ ls -l`
新建目录	`mkdir <dirname>`	`$ mkdir environment`
变换目录	`cd <dirname>`	`$ cd environment/`

作用	命令	示例
进入上层目录		`$ cd ..`
进入家目录		`$ cd ~` 或 `$ cd`
进入家目录中的文件夹		`$ cd ~/environment/`
移动文件（重命名）	`mv <source> <target>`	`$ mv foo bar`
复制文件	`cp <source> <target>`	`$ cp foo bar`
删除文件	`rm <file>`	`$ rm foo`
删除空目录	`rmdir <directory>`	`$ rmdir environment/`
删除非空目录	`rm -rf <directory>`	`$ rm -rf tmp/`
拼接并显示文件的内容	`cat <file>`	`$ cat ~/.ssh/id_rsa.pub`

不管在本地还是在云 IDE 中，下一步都是使用代码清单 1.6 中的命令创建第一个应用。注意，在这个代码清单中，我们明确指定了 Rails 版本。这么做的目的是确保使用代码清单 1.2 中安装的 Rails 版本来创建应用的文件结构。

代码清单 1.6：执行 `rails new` 命令（明确指定版本号）

```
$ cd ~/environment
$ rails _6.1.3_ new hello_app
      create
      create  README.md
      create  Rakefile
      create  .ruby-version
      create  config.ru
      create  .gitignore
      create  Gemfile
         run  git init from "."
Initialized empty Git repository in /home/ubuntu/environment/hello_app/.git/
      create  package.json
      create  app
      create  app/assets/config/manifest.js
      create  app/assets/stylesheets/application.css
      create  app/channels/application_cable/channel.rb
      create  app/channels/application_cable/connection.rb
      create  app/controllers/application_controller.rb
      create  app/helpers/application_helper.rb
      .
      .
      .
```

留意一下 `rails new` 命令创建的文件和目录。这个标准的文件结构（图 1.11）是 Rails 的众多优势之一：让你从零开始快速创建一个可运行的功能最简的应用。而且，所有 Rails 应用都使用这种文件结构，阅读他人的代码时很快就能理清头绪。

这些文件的作用见表 1.2，本书后面的内容将介绍其中大多数文件和目录。从 5.2.1 节开始，我们将介绍 `app/assets` 目录，这是 Asset Pipeline 的一部分。Asset Pipeline 简化了层叠样式表和图像等静态资源文件的组织和部署方式。

表 1.2：Rails 目录结构概要

文件/文件夹	作用
app/	应用的核心文件，包含模型、视图、控制器和辅助方法
app/assets	应用的静态资源文件，例如层叠样式表（CSS）和图像
bin/	可执行的二进制文件
config/	应用的配置
db/	数据库文件
doc/	应用的文档
lib/	代码库模块文件
log/	应用的日志文件
public/	公共（如通过浏览器）可访问的文件，例如错误页面
bin/rails	生成代码、打开终端会话或启动本地服务器的程序
test/	应用的测试
tmp/	临时文件
README.md	应用简介
Gemfile	应用所需的 gem
Gemfile.lock	gem 列表，确保这个应用的副本使用相同版本的 gem
config.ru	Rack 中间件的配置文件
.gitignore	Git 忽略文件的模式

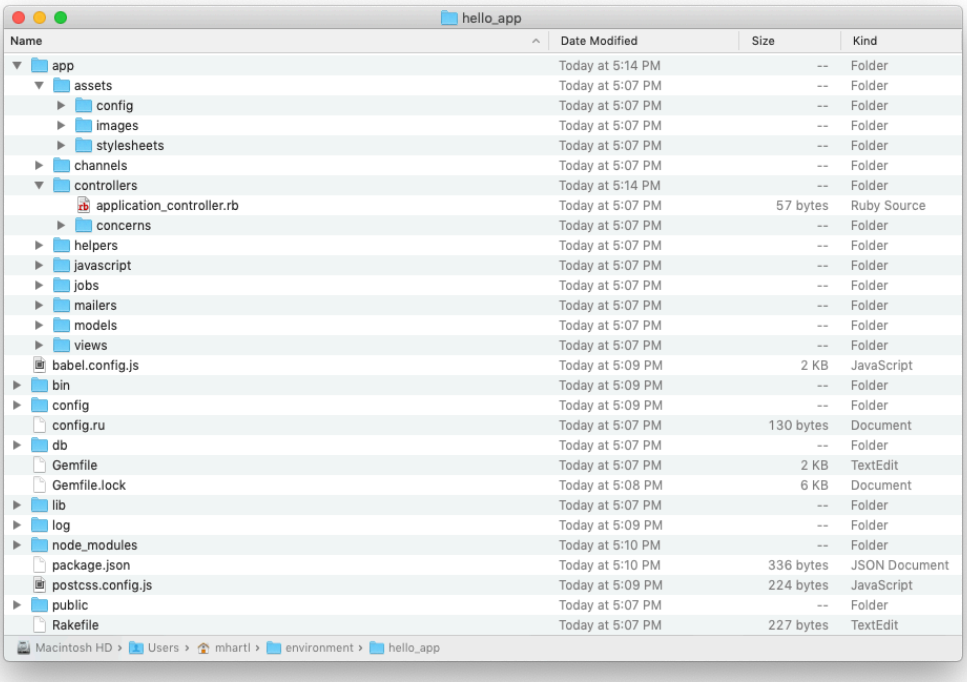


图 1.11：新建 Rails 应用时生成的目录结构

1.2.1 Bundler

创建完一个新的 Rails 应用后，下一步是使用 Bundler 安装和引入应用所需的 gem。执行代码清单 1.6 中的 `rails` 命令时会自动运行 Bundler（`bundle install`）。这一节，我们将修改应用默认使用的 gem，然后再次运行 Bundler。首先，在文本编辑器中打开 `Gemfile` 文件。（在云 IDE 中，点击文件系统浏览器中的应用目录，然后双击 `Gemfile` 文件。）虽然具体的版本号和内容有些许不同，但大概与代码清单 1.7 和图 1.12 差不多。（这个文件中的内容是 Ruby 代码，现在先不关心句法，第 4 章将详细介绍。）

如果你没看到如图 1.12 所示的文件和目录，点击文件浏览器中的齿轮图标，然后选择“Refresh File Tree”（刷新文件树）。（通常，如果某个文件或目录没出现，就可以刷新文件树。）[3]

3. 这里便体现了“技术是复杂的”（旁注 1.2）。

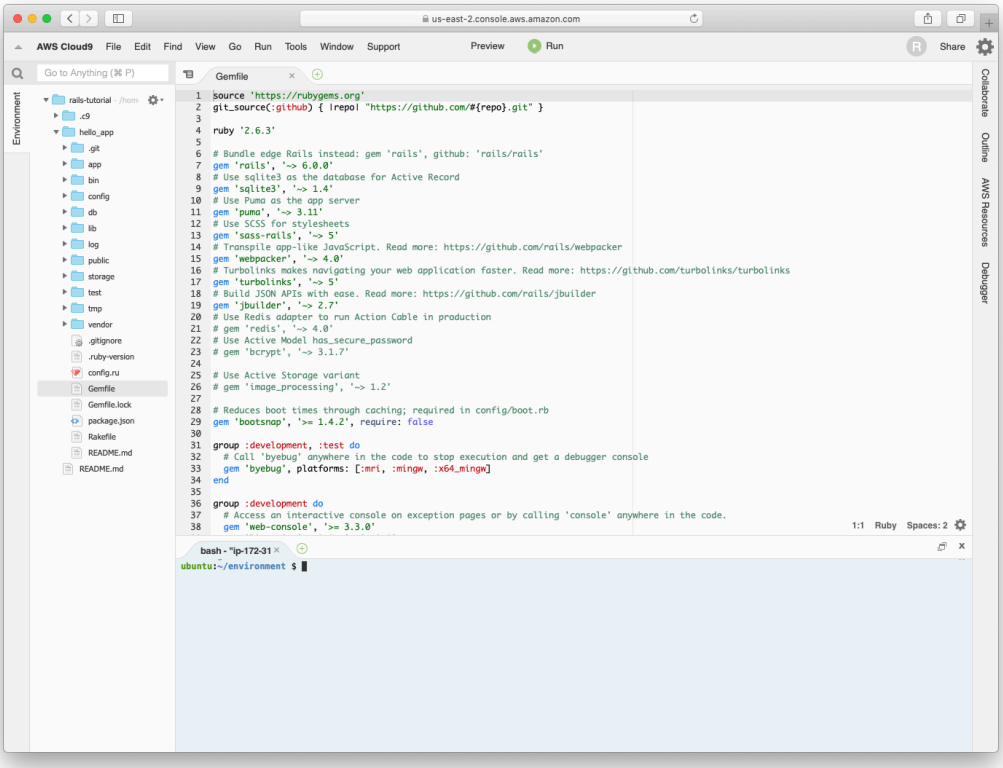


图 1.12：在文本编辑器中打开默认生成的 Gemfile 文件

代码清单 1.7：hello_app 目录中默认生成的 Gemfile 文件

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.6.3'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails', branch: 'main'
gem 'rails', '~> 6.1.3'
# Use sqlite3 as the database for Active Record
gem 'sqlite3', '~> 1.4'
# Use Puma as the app server
gem 'puma', '~> 5.0'
# Use SCSS for stylesheets
gem 'sass-rails', '>= 6'
# Transpile app-like JavaScript. Read more: https://github.com/rails/webpacker
gem 'webpacker', '~> 5.0'
# Turbolinks makes navigating your web application faster.
# Read more: https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.7'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 4.0'
# Use Active Model has_secure_password
# gem 'bcrypt', '~> 3.1.7'
```

```
# Use Active Storage variant
# gem 'image_processing', '~> 1.2'

# Reduces boot times through caching; required in config/boot.rb
gem 'bootsnap', '>= 1.4.4', require: false

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  # Access an interactive console on exception pages or by calling 'console'
  # anywhere in the code.
  gem 'web-console', '>= 4.1.0'
  # Display performance information such as SQL time and flame graphs for each
  # request in your browser.
  # Can be configured to work on production as well see:
  # https://github.com/MiniProfiler/rack-mini-profiler/blob/master/README.md
  gem 'rack-mini-profiler', '~> 2.0'
  gem 'listen', '~> 3.3'
  # Spring speeds up development by keeping your application running in the
  # background. Read more: https://github.com/rails/spring
  gem 'spring'
end

group :test do
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '>= 3.26'
  gem 'selenium-webdriver'
  # Easy installation and use of web drivers to run system tests with browsers
  gem 'webdrivers'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

其中很多行代码都用 # 符号注释掉了（4.2 节），这些代码是为了告诉你一些常用的 gem，也是为了展示 Bundler 的句法。现在，除了这些默认的 gem 之外，我们还不需要其他 gem。

如果没在 gem 指令中指定版本号，Bundler 自动安装最新版。下面就是一例：

```
gem 'spring'
```

此外，还有两种常用的方法，用于指定 gem 版本的范围，在一定程度上控制 Rails 使用的版本。首先看下面这行代码：

```
gem 'capybara', '>= 3.26'
```

这行代码的意思是，安装版本号大于或等于 3.26 的 capybara gem（在测试中使用），就算是 7.2 版也安装。

第二种方法如下所示：

```
gem 'sqlite3', '~> 1.4'
```

这行代码的意思是，安装版本号大于或等于 1.4，但不大于 2 的 sqlite3。也就是说，>= 的意思是始终安装最新版；~> 1.4 的意思是安装 1.5 版（如果有这一版的话），而不安装 2.0 版。

不过，经验告诉我们，即使是最小版本的升级也可能导致 Rails 应用无法运行，所以在本教程中我们基本上会为所有的 gem 都指定精确的版本号。你可以使用任何 gem 的最新版本，还可以在 Gemfile 文件中使用 ~>（一般推荐有经验的用户使用），但事先提醒你，这可能会导致本教程开发的应用表现异常。

修改代码清单 1.7 中的 Gemfile 文件，换用精确的版本号，得到的结果如代码清单 1.8 所示。[4] 注意，借此机会我们还变动了 sqlite3 gem 的位置，只在开发环境和测试环境（7.1.1 节）中安装，从而避免与 Heroku 所用的数据库冲突（1.4 节）。另外，我们把最后一行注释掉了，因为那个 gem 只用于 Microsoft Windows 系统，在 Windows 之外的系统中可能会出现提醒消息，让人不明所以。如果你使用的是 Windows 系统，请把那一行的注释去掉。最后，我们还把代码清单 1.7 中指定 Ruby 版本号的那行删掉了。其实，建议在重要的应用中保留这一行（7.5.4 节），不过对本书这样的教程来说，留着反而容易出错，会让问题复杂化。（当然，如果去掉那一行之后应用不能正常运行，那就应该加上。）

代码清单 1.8：在 Gemfile 文件中为所有 gem 指定精确的版本号

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.1.3'
gem 'puma',       '5.2.2'
gem 'sass-rails', '6.0.0'
gem 'webpacker',  '5.2.1'
gem 'turbolinks', '5.2.1'
gem 'jbuilder',   '2.10.0'
gem 'bootsnap',   '1.7.2', require: false

group :development, :test do
  gem 'sqlite3', '1.4.2'
  gem 'byebug',  '11.1.3', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',        '4.1.0'
  gem 'rack-mini-profiler', '2.3.1'
  gem 'listen',             '3.4.1'
  gem 'spring',             '2.1.1'
end

group :test do
  gem 'capybara',           '3.35.3'
  gem 'selenium-webdriver', '3.142.7'
  gem 'webdrivers',         '4.6.0'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
# Uncomment the following line if you're running Rails
```

4. 如果想查看各 gem 的具体版本号，在命令行中执行 gem list <gem name> 命令。不过代码清单 1.8 已经给出了。

```
# on a native Windows system:
# gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

把代码清单 1.8 中的内容写入应用的 Gemfile 文件之后，执行 bundle install 命令[5] 安装这些 gem：

代码清单 1.9：安装代码清单 1.8 中的 gem

```
$ cd hello_app/
$ bundle _2.2.13_ install
Fetching source index for https://rubygems.org/
.
.
.
```

与代码清单 1.6 中的 rails 命令一样，这里我们也指定了具体的版本号，确保使用正确的 bundler 版本（即代码清单 1.3 中安装的版本）。

bundle install 命令可能要执行一会儿，不过结束后我们的应用就能运行了。（在 Windows 系统中一定要去掉代码清单 1.8 中最后一行的注释。）

顺便说一下，执行 bundle install 命令时可能会提醒你先执行 bundle update 命令。此时，应该按照提醒，先执行 bundle update。（保险起见，最好像代码清单 1.9 那样，执行 bundle _2.2.13_ update 命令。）在事情不能按计划进行时要学着不要惊慌，这是应对“技术是复杂的”这一问题的关键。冷静下来你便会惊奇地发现，“错误”消息中往往包含修正问题的具体说明。

> **旁注 1.4：RubyGems 国内镜像**
>
> 有些情况下，使用 Bundler 安装 gem 的速度可能不理想，这时可以使用国内的镜像提速：
>
> ```
> $ bundle config mirror.https://rubygems.org https://gems.ruby-china.com
> ```
>
> 详见 https://gems.ruby-china.com。

1.2.2 rails server

运行完 1.2 节中的 rails new 命令和 1.2.1 节中的 bundle install 命令之后，我们的应用就可以运行了。但是怎样运行呢？Rails 自带了一个命令行程序（或叫脚本），可以运行一个本地服务器，协助我们的开发工作。这个命令是 rails server。

运行 rails server 命令之前，在某些系统（包括我们使用的云 IDE）中要设置一下，允许连接本地 Web 服务器。为此，打开 config/environments/development.rb 文件，加上代码清单 1.10 中高亮显示的那两行（图 1.13）。

代码清单 1.10：允许连接本地 Web 服务器
config/environments/development.rb

```
Rails.application.configure do
  .
  .
```

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

```
  # 允许连接本地服务器
  config.hosts.clear
end
```

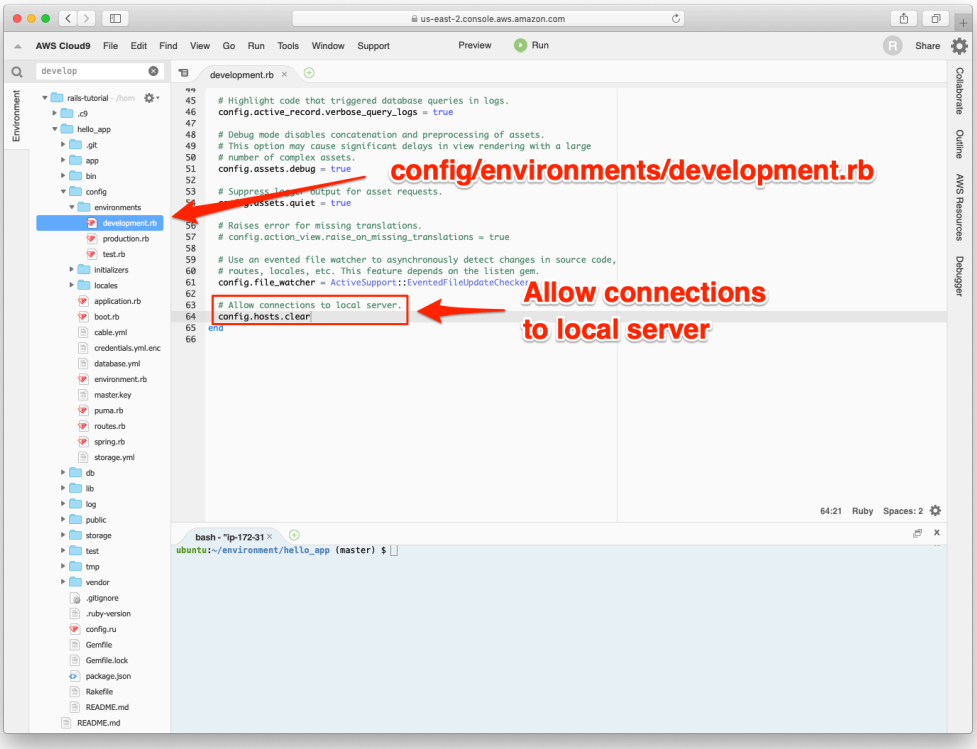


图 1.13：允许 Cloud9 连接 Rails 服务器

代码清单 1.11 中的 `rails server` 命令建议在单独的终端标签页中执行，这样便可以在原来的标签页中执行其他命令了，如图 1.14 和图 1.15 所示。注意代码清单 1.11 中的提示，服务器可以使用 Ctrl-C 组合键关闭。[6]

代码清单 1.11：运行 Rails 服务器

```
$ cd ~/environment/hello_app/
$ rails server
=> Booting Puma
=> Ctrl-C to shutdown server
```

在本地环境中，把 http://localhost:3000 粘贴到浏览器的地址栏中；在云 IDE 中，打开“Preview”（预览）菜单，点击“Preview Running Application”（预览运行中的应用）（图 1.16），在新窗口或新标签页中打开（图 1.17）。在这两种环境中，显示的页面应该都与图 1.18 类似。

6. 这里的“C”指代键盘上的字母，而不是大写的字母，所以不用按下 Shift 键输入大写的“C”。

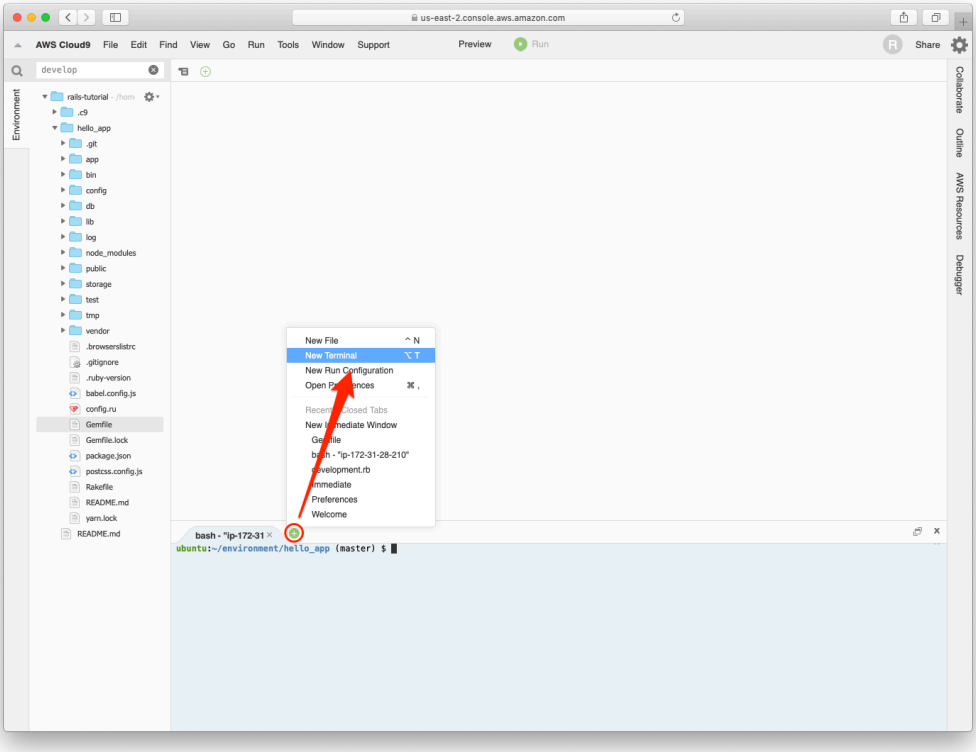


图 1.14：再打开一个终端标签页

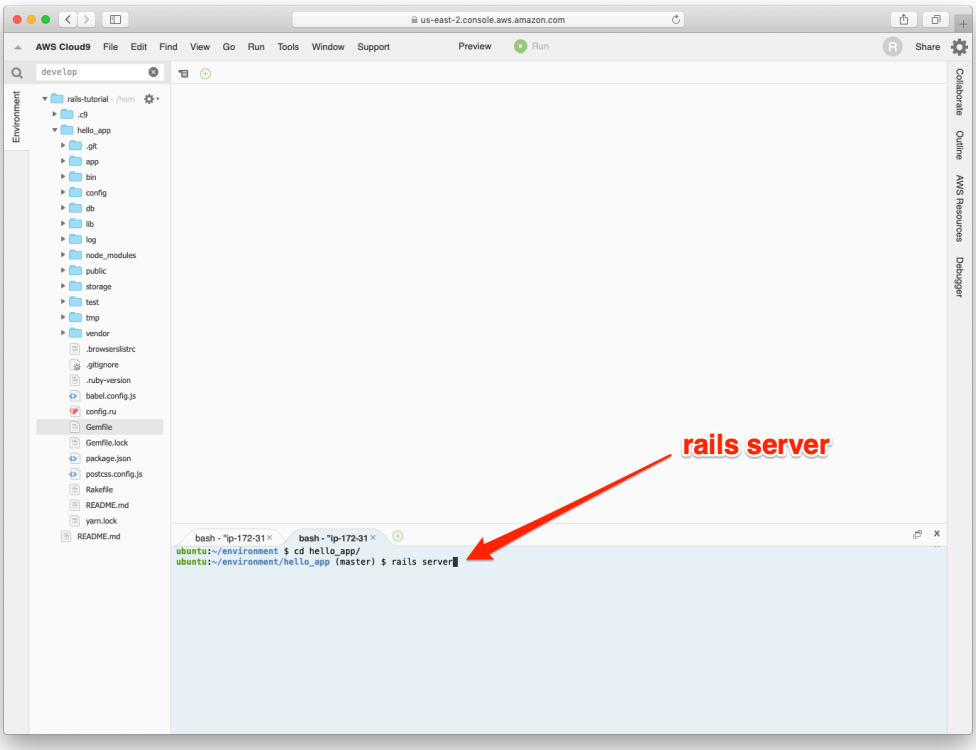


图 1.15：在另一个标签页中运行 Rails 服务器

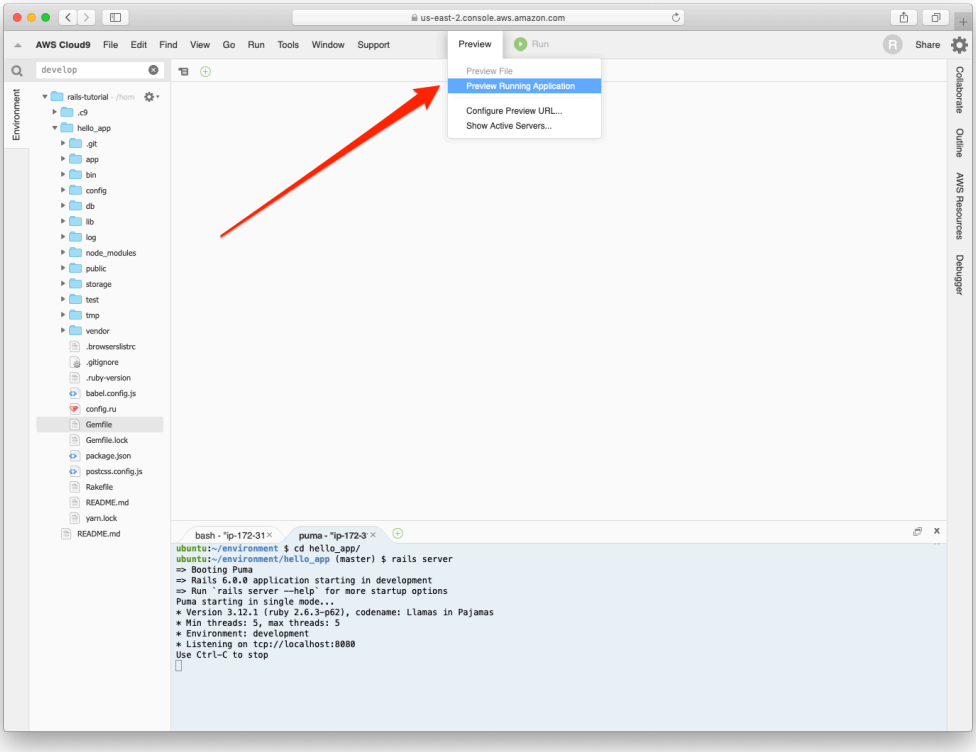


图 1.16：分享运行在云工作区中的本地服务器

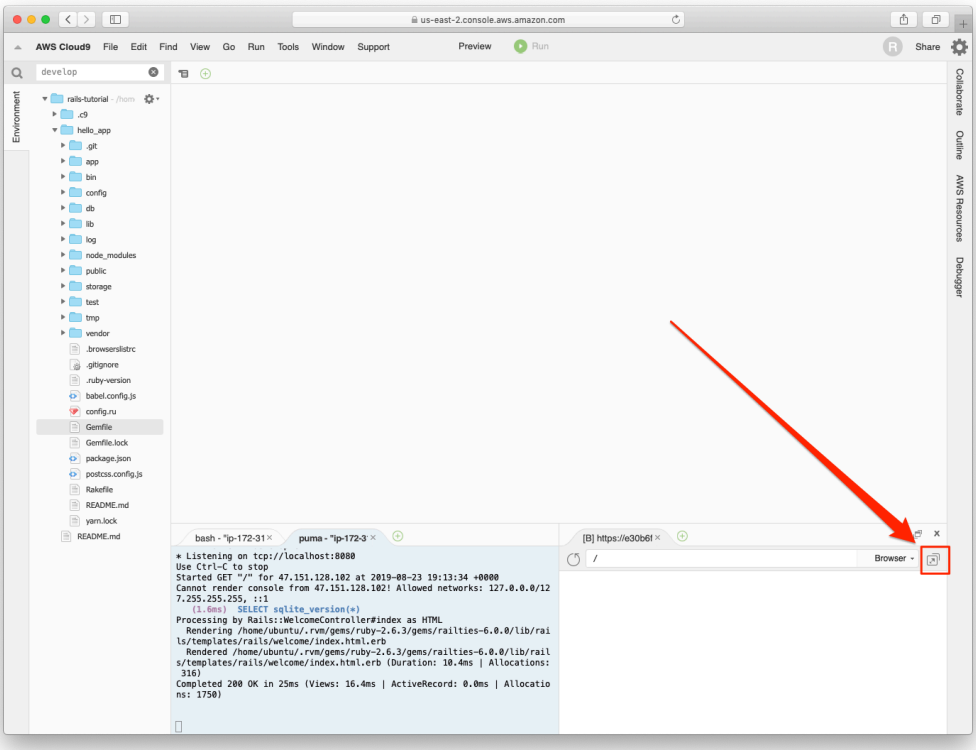


图 1.17：在新窗口或新标签页中打开运行中的应用

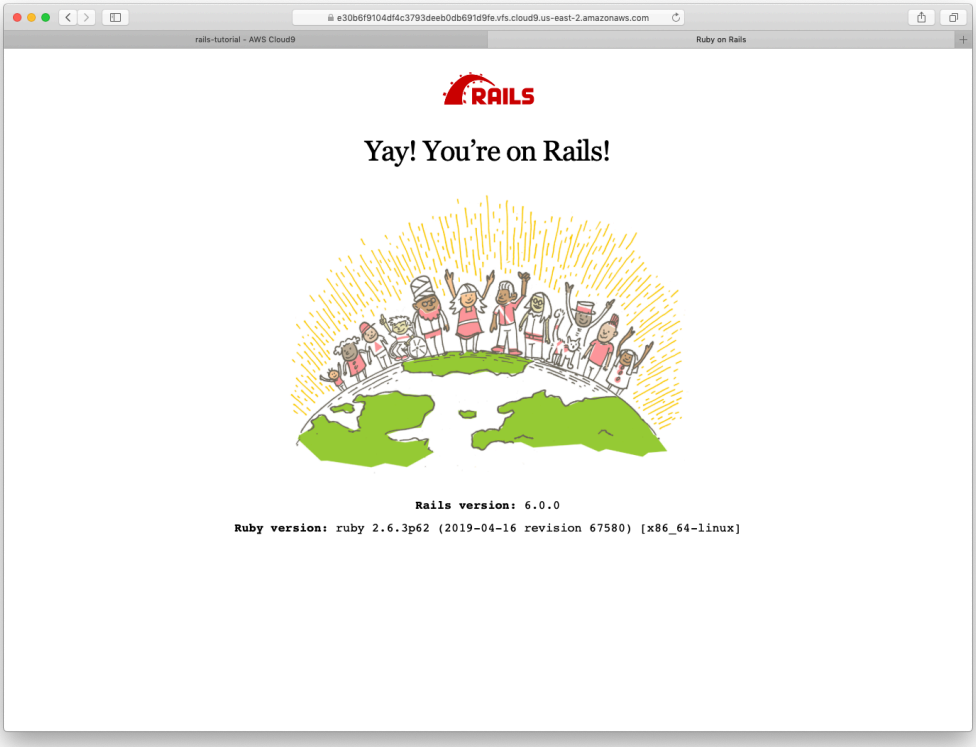


图 1.18：执行 `rails server` 命令后看到的 Rails 默认页面

练习

本书有大量练习，强烈建议你在阅读的过程中一一解答。

为了避免练习妨碍主线，解答通常不会影响后续的代码清单。不过，有极少数的练习解答在后面要用到，此时正文中会给出解答。这意味着，有时你的代码可能与书中给出的代码有出入，这时，请你自己设法解决。

很多练习具有一定难度，我们先来做几道简单的题，热热身：

1. 根据 Rails 默认页面中的信息，你的系统使用的 Ruby 是哪个版本？在命令行中执行 `ruby -v` 命令确认一下。
2. Rails 是哪个版本？确认是否与代码清单 1.2 中指定的版本一样。

1.2.3 模型-视图-控制器

在初期阶段，概览一下 Rails 应用的工作方式（图 1.19）多少会有些帮助。你可能注意到了，在 Rails 应用的标准文件结构中有一个名为 `app/` 的目录（图 1.11），其中有三个子目录：`models`、`views` 和 `controllers`。这表明 Rails 采用了“模型-视图-控制器”（MVC）架构模式。这种模式把应用中的数据（例如用户信息）与显示数据的代码分开，这是图形用户界面（GUI）常用的架构方式。

与 Rails 应用交互时，浏览器发出一个请求（request），Web 服务器收到请求之后将其传给 Rails 应用的控制器（controller），决定下一步做什么。某些情况下，控制器会立即渲染视图（view），使用模板生成 HTML，发回浏览器。在动态网站中，更常见的是控制器与模型（model）交互。一个模型是一个 Ruby 对象，表示网站中的一个元素（例如一个用户），并且负责与数据库通信。与模型交互后，控制器再渲染视图，把生成的 HTML 返回给浏览器。

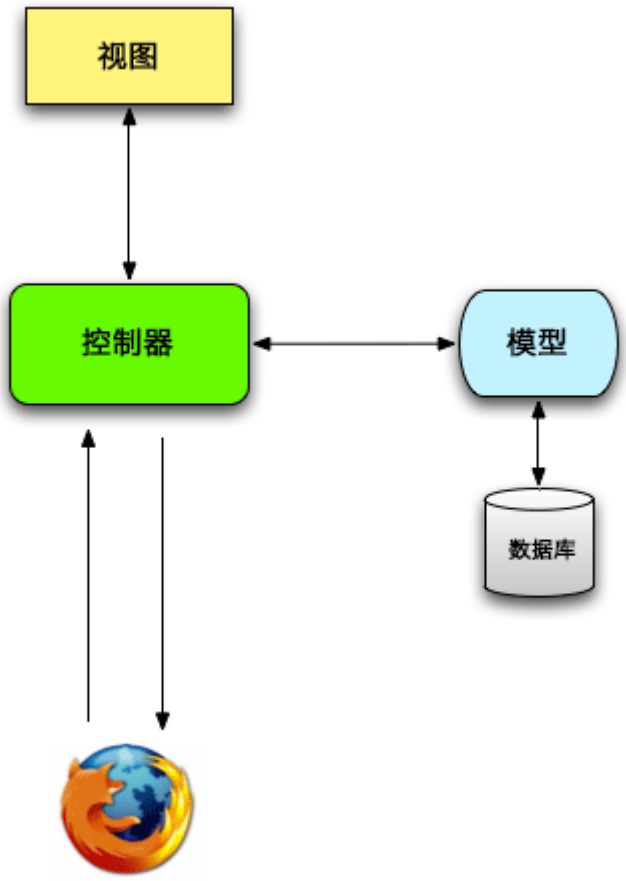


图 1.19：MVC 架构图解

如果你觉得这些内容有点抽象，不用担心，后文会进一步讨论这些概念。在 1.2.4 节，我们将首次使用 MVC 架构编写应用；2.2.2 节会以一个应用为例较为深入地讨论 MVC；在最后那个演示应用中会使用完整的 MVC 架构。从 3.2 节开始，介绍控制器和视图；从 6.1 节开始，介绍模型；7.1.2 节则把这三部分结合在一起使用。

1.2.4 Hello, world!

接下来我们要对这个使用 MVC 框架开发的第一个应用做些小改动：添加一个控制器动作（action），渲染字符串“hello, world!”，以此替代图 1.18 中的 Rails 默认页面。（从 2.2.2 节开始，我们将深入学习控制器动作。）

从“控制器动作”这个名称可以看出，动作在控制器中定义。我们要在 `Application` 控制器中定义这个动作，将其命名为 `hello`。其实，现在我们的应用只有这一个控制器。执行下述命令可以验证这一点：

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

`hello` 动作的定义如代码清单 1.12 所示，它调用 `render` 方法返回 HTML 文本“hello, world!”。（现在先不管 Ruby 句法，第 4 章将详细介绍。）

代码清单 1.12：在 `Application` 控制器中添加 `hello` 动作
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
```

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

定义好返回所需字符串的动作之后，我们要告诉 Rails 使用这个动作代替图 1.18 中的默认首页。为此，我们要修改 Rails 路由器（router）。路由器在控制器之前（图 1.19），决定浏览器发给应用的请求由哪个动作处理。（简单起见，图 1.19 中省略了路由器，从 2.2.2 节开始将详细介绍路由器。）具体而言，我们要修改默认的首页，也就是根路由（root route）。这个路由决定根 URL 显示哪个页面。根 URL 是 http://www.example.com/ 这种形式（最后一个斜线后面没有任何内容），所以经常简化使用 /（斜线）表示。

如代码清单 1.13 所示，Rails 路由文件（`config/routes.rb`）中有一行注释，让我们阅读 Rails 指南中讲解路由的文章（https://rails.guide/book/routing.html）。那篇文章说明了如何定义根路由，句法如下：

```
root 'controller_name#action_name'
```

这里，控制器的名称是 `application`，动作的名称是 `hello`，因此根路由要像代码清单 1.14 那样定义。

代码清单 1.13：默认的路由文件（经过重新编排）
config/routes.rb

```
Rails.application.routes.draw do
  # For details on the DSL available within this file,
  # see https://guides.rubyonrails.org/routing.html
end
```

代码清单 1.14：设置根路由
config/routes.rb

```
Rails.application.routes.draw do
  root 'application#hello'
end
```

有了代码清单 1.12 和代码清单 1.14 中的代码，根路由就会按照我们的要求显示“hello, world!”了，如图 1.20 所示。[7]

练习

1. 把 `hello` 动作（代码清单 1.12）中的“hello, world!”改成“hola, mundo!”。
2. 使用倒置的感叹号（如“¡Hola, mundo!”中的第一个字符），证明 Rails 支持非 ASCII 字符。[8] 结果如图 1.21 所示。在 Mac 中输入 ¡ 字符的方法是按 Option-1 键；此外，也可以直接把这个字符复制粘贴到编辑器中。
3. 按照编写 `hello` 动作的方式（代码清单 1.12），再添加一个动作，命名为 `goodbye`，渲染文本“goodbye, world!”。然后修改路由文件（代码清单 1.14），把根路由改成 `goodbye`。结果如图 1.22 所示。

7. 本书的 Cloud9 分享 URL 的基 URL 由 rails-tutorial-c9-mhartl.c9.io 变成了 Amazon Web Services 的一个地址，但是很多情况下截图没变，因此某些插图中的浏览器地址栏里显示的还是以前的 URL（例如图 1.20）。这是一种细微差异，虽然体现了技术是复杂的（旁注 1.2），但是你应该能区分。

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

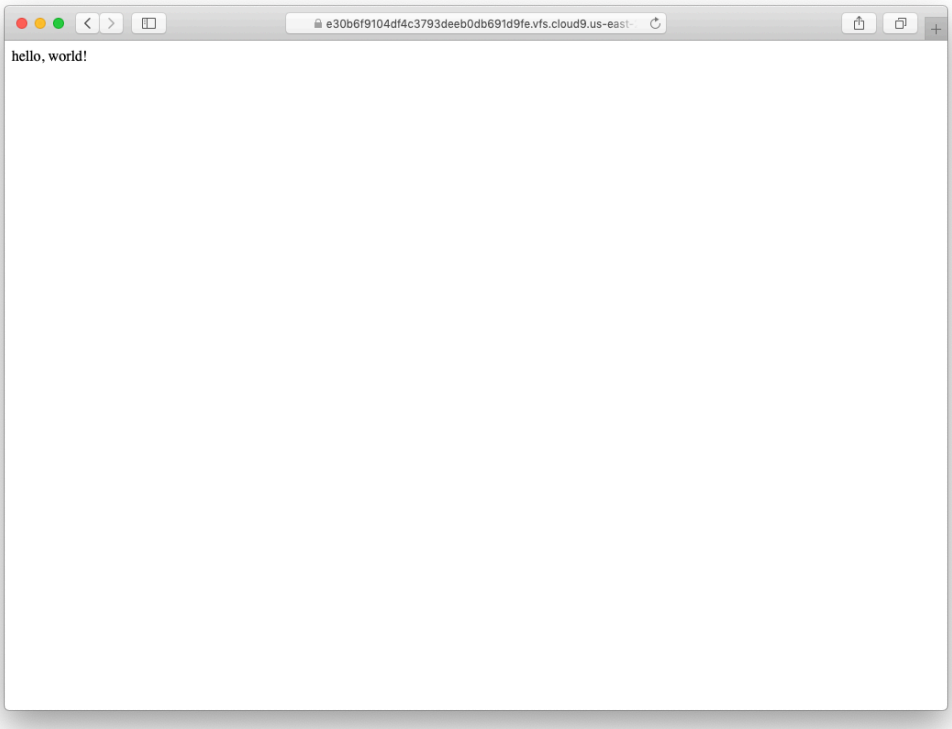


图 1.20：在浏览器中查看显示“hello, world!”的页面

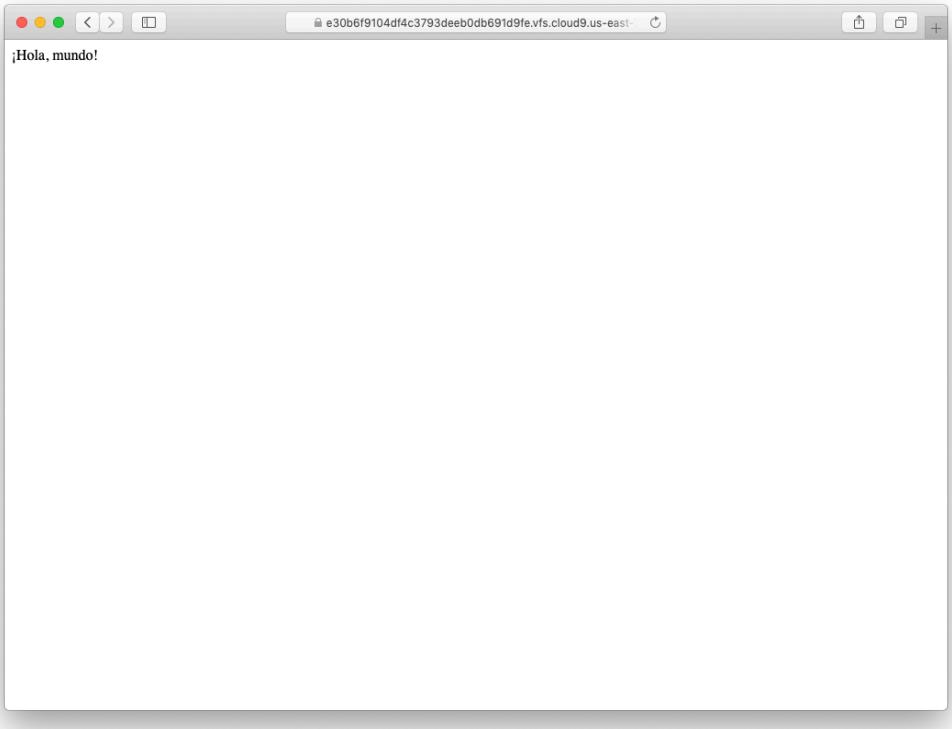


图 1.21：修改根路由，返回“¡Hola, mundo!”

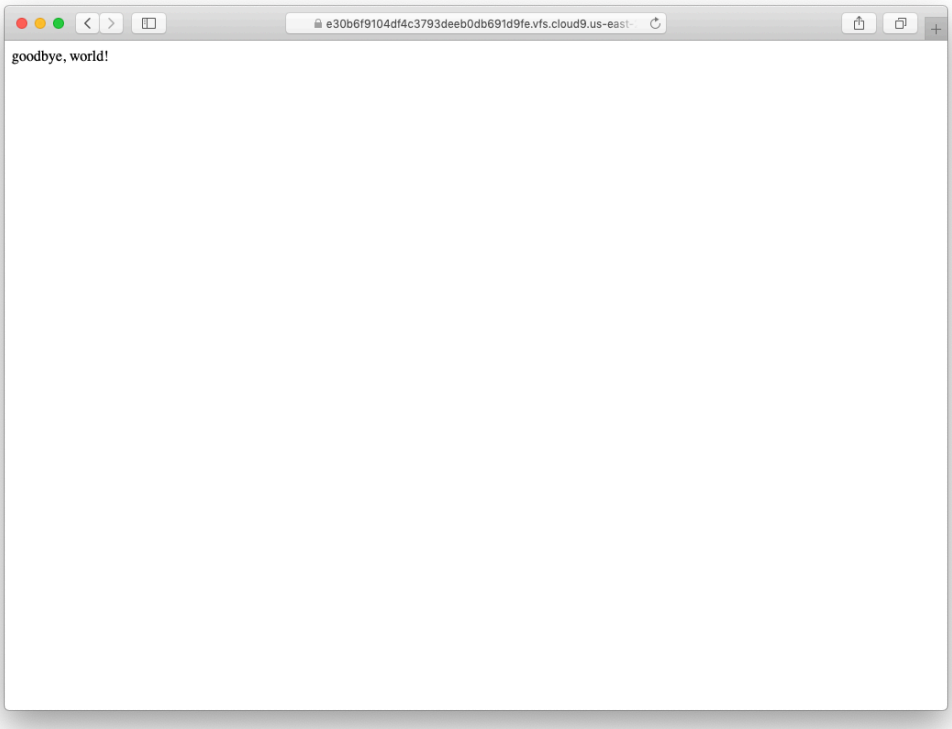


图 1.22：修改根路由，返回“goodbye, world!”

1.3 使用 Git 做版本控制

我们创建了一个“hello, world”应用，接下来要花点时间做一件事。虽然这件事不是必须的，但是经验丰富的软件开发人员都认为这是最基本的事情，即把应用的源代码纳入版本控制系统。版本控制系统可以跟踪项目中代码的变化，便于与他人协作；如果出现问题（例如不小心删除了文件），还可以回滚到以前的版本。每个专业的软件开发者都应该学习使用版本控制系统。

版本控制系统种类很多，Rails 社区基本都使用 Git（https://git-scm.com）。Git 是一个分布式版本控制系统，由 Linus Torvalds 开发，最初的目的是存储 Linux 内核代码。Git 相关的知识很多，本书只涉及一些皮毛。如果想深入了解，请阅读 Learn Enough Git to Be Dangerous（https://www.learnenough.com/git）。

之所以强烈推荐使用 Git 做版本控制，不仅因为 Rails 社区都在用，还因为使用 Git 更易于备份和分享代码（1.3.3 节），而且也便于部署应用（1.4 节）。

1.3.1 设置 Git

1.1.1 节推荐使用的云 IDE 默认自带 Git，不用再安装。如果你没使用云 IDE，可以参照 Learn Enough Git to Be Dangerous 中的说明，在自己的系统中安装 Git。

第一次运行前要做的系统设置

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

第一步（必须的），配置自己的姓名和电子邮件地址，如代码清单 1.15 所示。

代码清单 1.15：配置供 Git 使用的姓名和电子邮件地址

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

注意，在 Git 配置中设定的姓名和电子邮件地址会在所有公开的仓库中显示。

如果使用云 IDE，接下来还要配置默认的编辑器，在 Git 需要时使用（例如编辑或修补项目的改动时）。我们将使用 nano 编辑器，它对新手较为友好，而且是云 IDE 中默认的编辑器。写作本书时，默认编辑器在退出云 IDE 后将被重置，而且路径也是错误的。为了解决这两个问题，执行代码清单 1.16 中的命令，创建一个指向 nano 可执行文件正确地址的符号链接（symbolic link，或 symlink）。[9]（代码清单 1.16 中的命令有点复杂，不理解也没关系。）

代码清单 1.16：在云 IDE 中配置默认的编辑器

```
$ sudo ln -sf `which nano` /usr/bin
```

接下来，做一个便利的设置（可选），为常用的 checkout 命令创建别名，如代码清单 1.17 所示。

代码清单 1.17：把 co 设置为 checkout 的别名

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

为了尽量保证兼容性，本书始终使用完整的 git checkout 命令，不过笔者平时大都使用简短的 git co。

最后一步，禁止 Git 在每次执行 push 或 pull 命令（1.3.4 节）时都要求我们输入密码。具体方法因操作系统而异，如果你使用的不是 Linux（包括云 IDE），可以参照“Caching your GitHub password in Git”一文（https://git.io/Je8XW）；如果使用 Linux（当然也包括云 IDE），设置一个缓存超时时间即可，如代码清单 1.18 所示。

代码清单 1.18：让 Git 记住密码一段时间

```
$ git config --global credential.helper "cache --timeout=86400"
```

像代码清单 1.18 那样配置之后，Git 能记住你在 86,400 秒（一天）内用过的密码。[10] 如果你十分在意安全，可以把时间设得短一些，例如默认的 900 秒（15 分钟）。

第一次使用仓库前要做的设置

下面的步骤每次新建仓库（repository，有时简称 repo）时都要执行。第一步，进入 hello_app 的根目录，初始化一个新仓库：

```
$ cd ~/environment/hello_app    # 以防在其他目录中
$ git init
Reinitialized existing Git repository in
/home/ubuntu/environment/hello_app/.git/
```

注意，Git 输出的消息表示仓库被重新初始化了。这是因为，从 Rails 6 起，执行 rails new 命令（代码清单 1.6）会自动初始化一个 Git 仓库（足以体现 Git 在技术圈的使用有多普遍）。所以，严格来说，这里不用再执行 git init 命令。不过，一般情况并不尽然，因此始终执行 git init 命令是个好习惯。

9. 笔者更倾向于使用 Vim 作为 Git 的编辑器。如果你想使用 Vim，把代码清单 1.16 中的 which nano 替换为 which vim。

10. 理论上，超时时间还可以设为更长的时间，可是在我们使用的云 IDE 中，计时器好像每天都会重置，因此把时长设为大于 86,400 秒的值似乎没有意义。

接下来，执行 `git add -A` 命令，把项目中的所有文件都放到仓库中：[11]

```
$ git add -A
```

这个命令把当前目录中的所有文件都放到仓库中，但是匹配特殊文件 `.gitignore` 中模式的文件除外。`rails new` 命令会自动生成一个适用于 Rails 项目的 `.gitignore` 文件，此外你还可以添加其他模式。[12]

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

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .browserslistrc
        new file:   .gitignore
        new file:   .ruby-version
        new file:   Gemfile
        new file:   Gemfile.lock
        .
        .
        .
```

如果想让 Git 保存这些改动，使用 `commit` 命令：

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

`-m` 标志的作用是为这次提交添加一个说明。如果没指定 `-m` 标志，Git 会打开系统的默认编辑器，让你在其中输入说明。（本书所有的示例都会使用 `-m` 标志。）

有一点要特别注意：Git 提交只发生在本地，也就是说只在执行提交操作的设备中存储内容。1.3.4 节会介绍如何把改动推送到远程仓库（使用 `git push` 命令）。

顺便说一下，可以使用 `log` 命令查看提交历史：

```
$ git log
commit b981e5714e4d4a4f518aeca90270843c178b714e (HEAD -> master)
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Sun Aug 18 17:57:06 2019 +0000

    Initialize repository
```

11. 很多开发者使用基本上等效的 `git add .` 命令，其中 `.`（点号）表示当前目录。不过二者之间还是有细微差别的。通常我们需要的是 `git add -A`，Git 官方文档中用的是这个命令，本书也将这么做。

12. 本书基本不会修改这个文件，不过 3.6 节的高级测试配置中演示了如何修改。

如果仓库的提交历史很多，可能要输入 q 退出。（Learn Enough Git to Be Dangerous 中有说明，`git log` 用到了 Learn Enough Command Line to Be Dangerous 中介绍的 `less` 接口。）

1.3.2 使用 Git 有什么好处

如果你以前从未用过版本控制系统，现在可能不完全理解版本控制的好处。举个例子说明一下吧。假如你不小心做了件傻事，例如把重要的 `app/controllers/` 目录删除了：

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

这里，我们用 Unix `ls` 命令列出 `app/controllers/` 目录里的内容，然后用 `rm` 命令删除这个目录（表 1.1）。如 Learn Enough Command Line to Be Dangerous 所述，`-rf` 标志的意思是“强制递归”，无需明确征求同意就递归删除所有文件、目录和子目录等。

检查状态，看看发生了什么：

```
$ git status
On branch master
Changes not staged for commit:
  (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
        deleted:    app/controllers/concerns/.keep

no changes added to commit (use "git add" and/or "git commit -a")
```

可以看到，我们删除了一个文件。但是这个改动只发生在工作区（working tree）中，还未提交到仓库。这意味着，我们可以使用 `checkout` 命令，并指定 `-f` 标志，强制撤销这次改动：

```
$ git checkout -f
$ git status
On branch master
nothing to commit, working tree clean
$ ls app/controllers/
application_controller.rb  concerns/
```

被删除的目录和文件又回来了，这下放心了！

1.3.3 GitHub

我们已经把项目纳入 Git 版本控制系统了，接下来要把代码推送到 GitHub 中，[13] 这是一个专门用来托管和分享 Git 仓库的网站。在 GitHub 中放一份 Git 仓库的副本有两个目的：其一，对代码做个完整备份（包括所有提交历史）；其二，便于以后协作。

GitHub 的使用方法很简单，如果没有账户，要先注册一个（https://github.com/join），如图 1.23 所示。

13. Bitbucket 和 GitLab 也是不错的选择。与 GitHub 一样，GitLab 也是使用 Rails 编写的。

注册或登录后，点击下拉菜单“+”，选择“New repository”（新建仓库），如图 1.24 所示。

在新建仓库页面，填写仓库名称（`hello_app`）和可选的描述，另外别忘了选择“Private”（私有）选项，如图 1.25 所示。虽然 Rails 应用基本安全，可放心公开，但是很多因素能导致问题出现（例如意外暴露密码或私钥），因此最好存为私有仓库。[14]

旁注 1.5：GitHub 默认分支名称

以前，GitHub 的默认分支名称是 `master`；2020 年 10 月，GitHub 把新建仓库的默认分支名称改成了一些开发者喜欢的 `main`。你可以采用 GitHub 的新名称，不过很多 Git 文档依然使用 `master`。如果想使用以前的默认分支名称，打开 https://github.com/settings/repositories，在“Repository default branch”下面把 `main` 改成 `master`。

点击“Create repository”（创建仓库）按钮之后，你会看到类似图 1.26 中的页面，指明如何把现有的仓库添加到 GitHub 中（见旁注 1.5）。点击“HTTPS”按钮，[15] 然后复制针对现有项目那部分中的命令。建议点击界面右边那个小图标，自动把代码清单 1.19 中的命令复制到剪切板中，然后粘贴到命令行终端里。

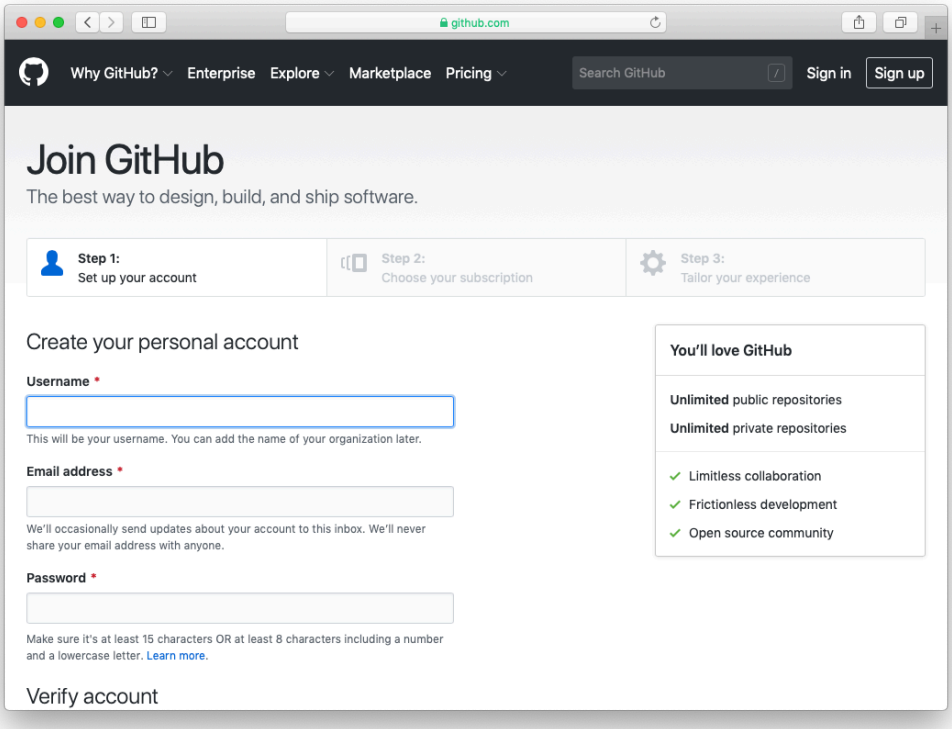


图 1.23：注册 GitHub 账户

14. GitHub 提供不限量的公开和私有仓库。

15. 图 1.26 中展示的 SSH 说明适合高级用户使用，如果你能熟练生成和配置 SSH 密钥，也可以使用 SSH 方式（https://git.io/Je8X8）。SSH 方式有一些优势，例如系统会自动缓存密码，这样就不用像代码清单 1.18 那样设置超时时间了。

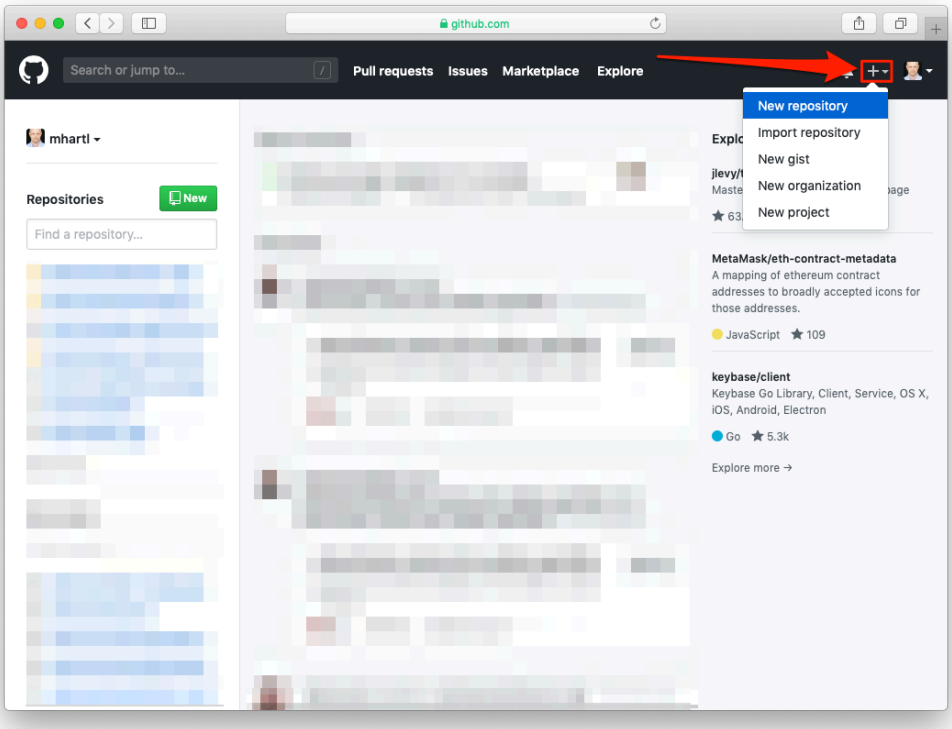


图 1.24：选择“New repository”（新建仓库）菜单

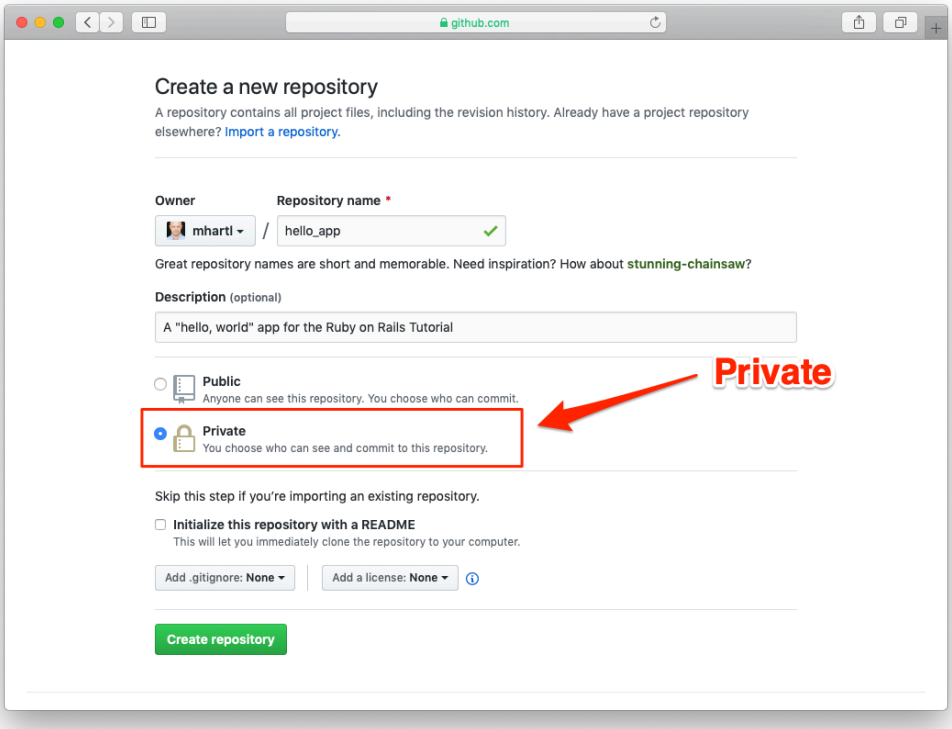


图 1.25：在 GitHub 中创建一个私有仓库

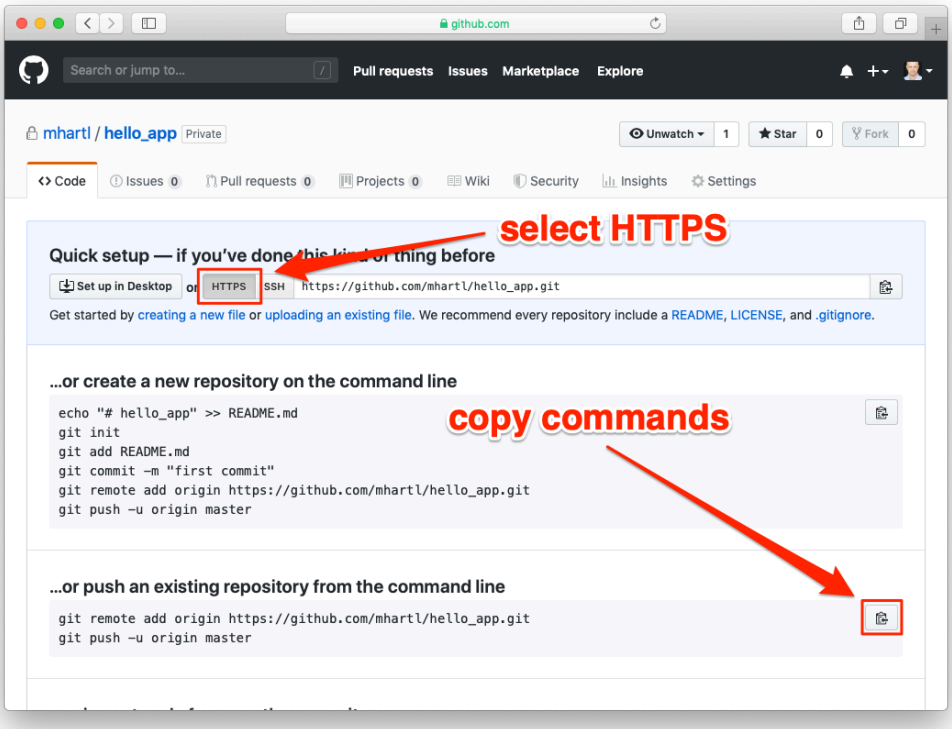


图 1.26：添加现有仓库的命令

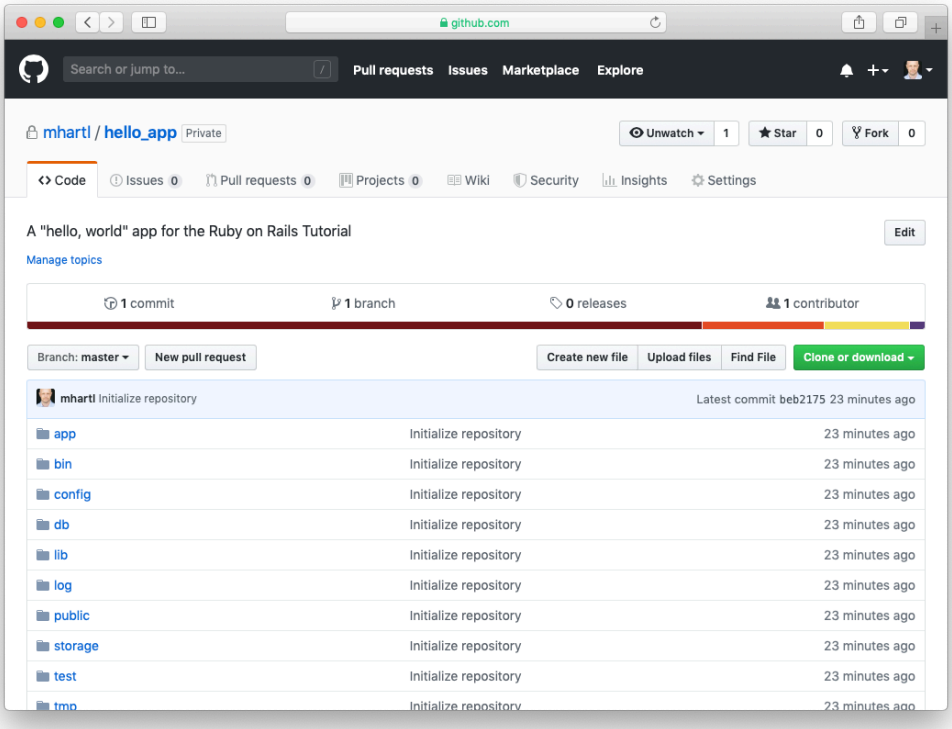


图 1.27：一个 GitHub 仓库的页面

最后，执行代码清单 1.19 中的命令。该命令会要求你输入 GitHub 密码，不过下一次（只要还在缓存期内）就不用了，因为我们做了代码清单 1.18 中的设置。

代码清单 1.19：把 GitHub 添加为远程源，然后推送仓库

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

代码清单 1.19 的意思是，先告诉 Git，你想添加 GitHub 为这个仓库的源（origin），然后把仓库推送到这个远程源。（别管 `-u` 标志的作用；如果你好奇，可以搜索“git set upstream”。）当然了，你要把 `<username>` 换成自己的用户名。例如，笔者执行的命令是：

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

推送完毕后，在 GitHub 中会显示 `hello_app` 仓库的页面。在这个页面中可以浏览文件、查看完整的提交历史，除此之外还有很多其他功能（图 1.27）。

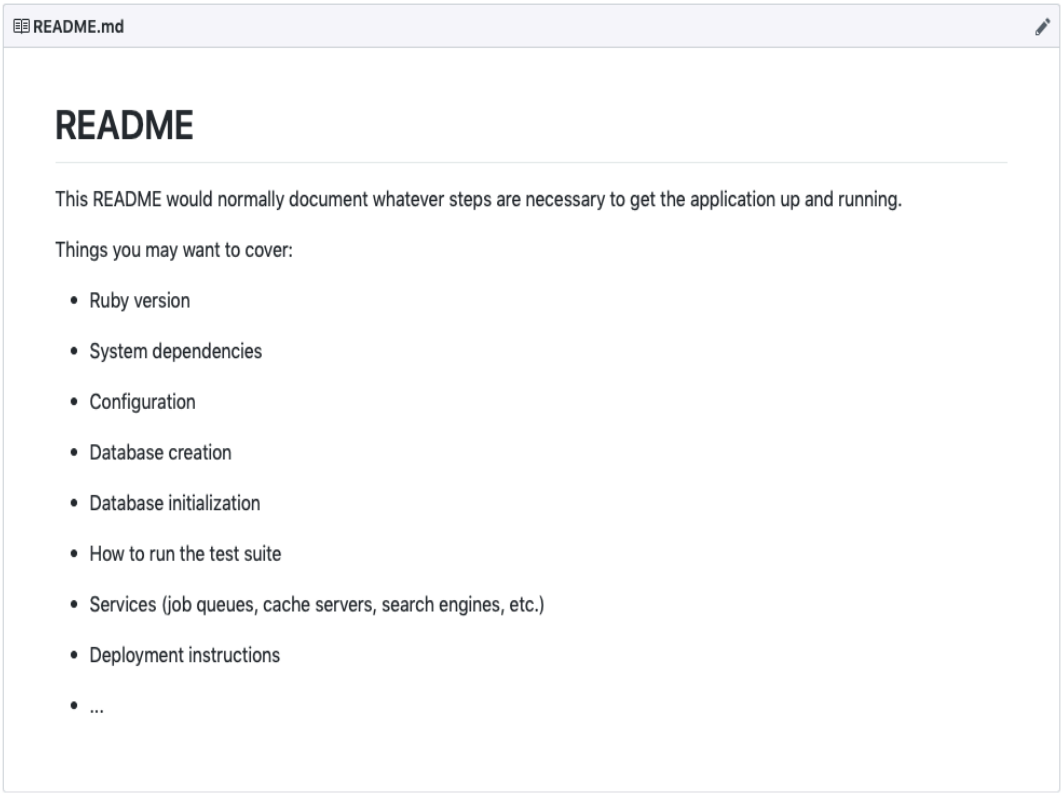


图 1.28：GitHub 渲染的 Rails 默认生成的 README 文件

1.3.4 分支、编辑、提交、合并

如果你跟着 1.3.3 节中的步骤做，可能注意到了，GitHub 自动渲染了仓库中的 README 文件，如图 1.28 所示。这个 `README.md` 文件由代码清单 1.6 中的命令自动生成。从文件的扩展名（`.md`）可以看出，这个文件使用 Markdown 编写。Markdown 是一门人类可读的标记语言，易于转换成 HTML——GitHub 就这么做了。

自动渲染 README 文件很贴心，不过我们最好修改里面的内容，描述手上的项目。这一节，我们将修改 README 文件，添加一些针对本书的内容。在修改的过程中，我们将首次演示笔者推荐在 Git 中使用的工作流程，即“分支、编辑、提交、合并”。[16]

分支

Git 分支（branch）的功能很强大。分支是对仓库的高效复制，在分支中所做的改动（或许是实验性质的）不会影响父级文件。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` 标志，新建一个主题分支（topic branch）：

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

其中，第二个命令 `git branch` 的作用是列出所有本地分支。星号（`*`）表示当前所在的分支。注意，`git checkout -b modify-README` 命令先创建一个新分支，然后切换到这个新分支——`modify-README` 分支前面的星号证明了这一点。

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

顺便说一下，像这种小改动，笔者一般不会新建分支（而是直接在主分支中修改）。现在这么做是为了让你养成好习惯。

编辑

创建好主题分支之后，我们要在 README 文件中添加一些内容，如代码清单 1.20 所示（图 1.29）。

代码清单 1.20：新的 README 文件

README.md

```
# Ruby on Rails Tutorial

## "hello, world!"

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

16. 如果想在图形界面中操作 Git 仓库，可以使用 Atlassian 推出的 SourceTree 应用（https://www.sourcetreeapp.com）。

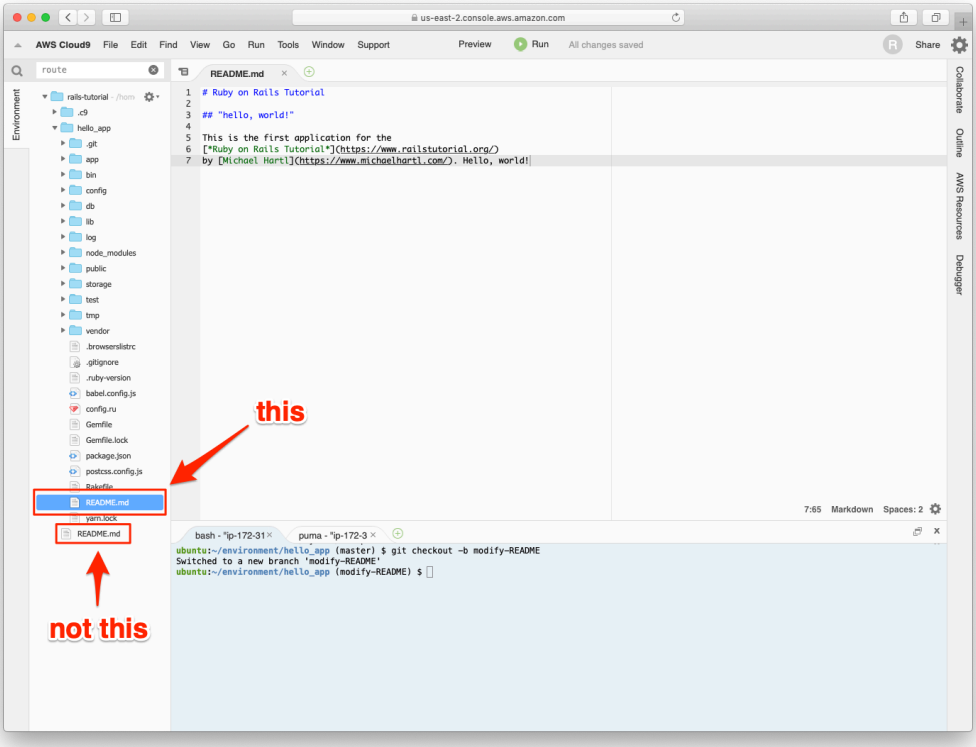


图 1.29：编辑 README 文件

提交

修改之后，查看一下该分支的状态：

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

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

这里，我们本可以使用“第一次使用仓库前要做的设置”一节中用过的 `git add -A` 命令，但是 `git commit` 命令提供了 `-a` 标志，可以直接提交现有文件中的全部改动：

```
$ git commit -a -m "Improve the README file"
[modify-README 34bb6a5] Improve the README file
 1 file changed, 5 insertions(+), 22 deletions(-)
```

使用 `-a` 标志一定要小心，千万别用错了。如果上次提交之后项目中添加了新文件，应该使用 `git add -A`，先告诉 Git 新增了文件。

注意，提交消息使用的是现在时（严格来说是祈使语气）。Git 把提交当做一系列补丁，在这种情况下，说明现在做了什么比说明过去做了什么要更合理。而且这种做法与 Git 命令生成的提交说明相配。

合并

我们已经改完了，现在可以把结果合并（merge）到主分支了：

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating b981e57..015008c
Fast-forward
 README.md | 27 +++++----------------------
 1 file changed, 5 insertions(+), 22 deletions(-)
```

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

合并之后，我们可以清理一下分支——如果主题分支不用了，可以使用 `git branch -d` 命令将其删除：

```
$ git branch -d modify-README
Deleted branch modify-README (was 015008c).
```

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

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

```
# 仅作演示之用，如果没搞砸，千万别这么做
$ git checkout -b topic-branch
$ <really mess up the branch>
$ git add -A
$ git commit -a -m "Make major mistake"
$ git checkout master
$ git branch -D topic-branch
```

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

推送

我们已经更新了 `README` 文件，现在可以把改动推送到 GitHub，看看改动的效果。之前我们已经推送过一次（1.3.3 节），在多数系统中都可以省略 `origin master`，直接执行 `git push` 命令：

```
$ git push
```

同样，GitHub 会把更新后的 Markdown 转换成 HTML（图 1.30）。

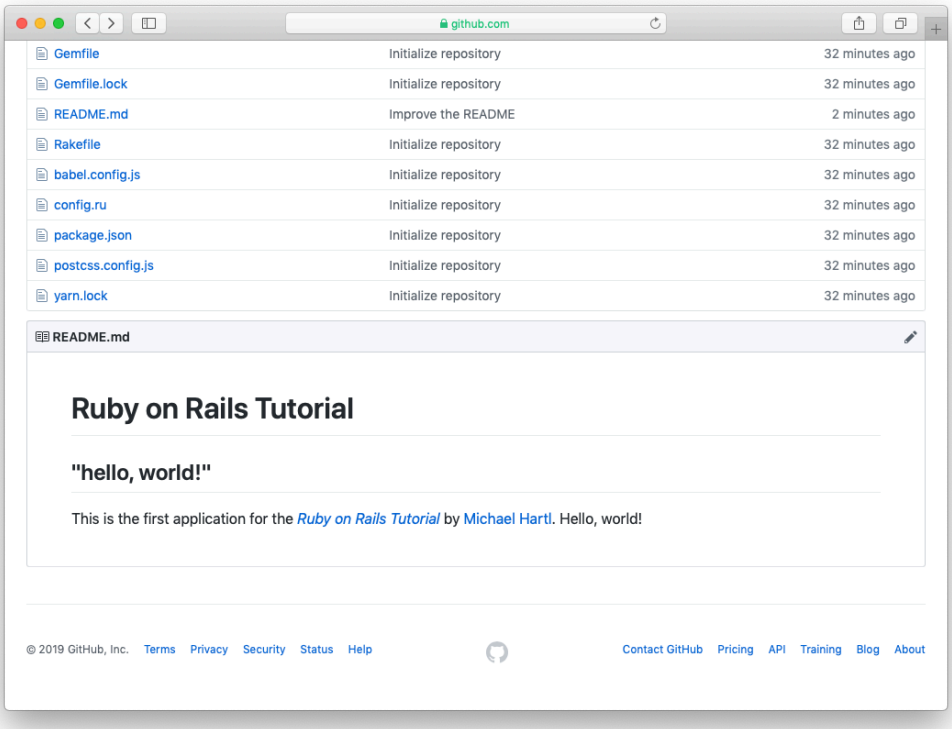


图 1.30：GitHub 中显示更新后的 README 文件

1.4 部署

虽然现在只是第一章，我们也要把 Rails 应用部署到生产环境。与 1.3 节中的版本控制一样，这一步可做可不做，不过在开发过程中尽早且频繁地部署，可以尽早发现开发中的问题。在开发环境中花费大量精力之后再部署，往往会在发布时遇到严重的集成问题。[17]

以前，部署 Rails 应用是件痛苦的事。但最近几年，Rails 开发生态系统不断成熟，已经出现很多好的解决方案，例如使用 Phusion Passenger（Apache 和 Nginx [18] Web 服务器的一个模块）的共享主机和虚拟私有服务器，Engine Yard 和 Rails Machine 这种提供全方位部署服务的公司，以及 Engine Yard Cloud 和 Heroku 这种云部署服务。

笔者最喜欢使用 Heroku 部署 Rails 应用。（你可能猜到了，Heroku 也是使用 Rails 编写的。）Heroku 专门用于部署 Rails 和其他 Web 应用，部署 Rails 应用的过程异常简单，只要源码纳入 Git 版本控制系统就好。这也是为什么要按照 1.3 节介绍的步骤设置 Git 的原因之一。如果你还没有照着做，现在赶紧做吧。很多情况下（包括这个教程），使用 Heroku 的免费套餐就可以了。

本节余下的内容专门介绍如何把我们的第一个应用部署到 Heroku 中。其中一些操作相对高级，如果没有完全理解也不要紧。本节的重点是把应用部署到线上环境中。

17. 如果你担心不小心公开了应用，可以参照 1.4.2 节的做法。不过对本书开发的示例应用来说，这份担心是多余的。
18. 读作“Engine X”。

1.4.1 搭建 Heroku 部署环境

Heroku 使用 PostgreSQL（读作 post-gres-cue-ell，通常简称 Postgres）数据库，因此我们要在生产环境安装 `pg` gem，这样 Rails 才能与 PostgreSQL 通信：

```
group :production do
  gem 'pg', '1.2.3'
end
```

另外，要加入代码清单 1.8 所做的改动，避免在生产环境安装 `sqlite3` gem，这是因为 Heroku 不支持 SQLite 数据库。[19]

```
  gem 'sqlite3', '1.4.2'
```

最终得到的 `Gemfile` 文件如代码清单 1.21 所示。

代码清单 1.21：增加并重新编排 gem 后的 `Gemfile` 文件

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.1.3'
gem 'puma',       '5.2.2'
gem 'sass-rails', '6.0.0'
gem 'webpacker',  '5.2.1'
gem 'turbolinks', '5.2.1'
gem 'jbuilder',   '2.10.0'
gem 'bootsnap',   '1.7.2', require: false

group :development, :test do
  gem 'sqlite3', '1.4.2'
  gem 'byebug',  '11.1.3', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.1.0'
  gem 'listen',                '3.2.1'
  gem 'spring',                '2.1.1'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',           '3.35.3'
  gem 'selenium-webdriver', '3.142.7'
  gem 'webdrivers',         '4.6.0'
end

group :production do
  gem 'pg', '1.2.3'
end
```

19. SQLite 通常作为嵌入式数据库使用，例如在手机中就十分普遍。Rails 在本地使用 SQLite 为默认数据库，因为 SQLite 易于设置，但是不适合作为 Web 应用的数据库，所以我们将在生产环境中使用 PostgreSQL。详情参见 3.1 节。

```
# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
# Uncomment the following line if you're running Rails
# on a native Windows system:
# gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

为了准备好部署环境，首先要执行 `bundle config` 命令，禁止在本地安装生产环境使用的 gem（这里的 `pg` gem），如代码清单 1.22 所示：

代码清单 1.22：不让 Bundler 安装生产环境的 gem

```
$ bundle _2.2.13_ config set --local without 'production'
$ bundle _2.2.13_ install
```

因为我们在代码清单 1.21 中只添加了用于生产环境的 gem，所以现在执行这个命令其实不会在本地安装任何新的 gem，但是又必须执行这个命令，因为我们要把 `pg` gem 添加到 `Gemfile.lock` 文件中。然后，提交这次改动：

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

接下来，我们要注册并配置一个 Heroku 新账户。首先，注册 Heroku 账户（https://signup.heroku.com/）。然后，检查系统中是否已经安装 Heroku 命令行客户端：

```
$ heroku --version
```

如果有 `heroku` 命令行接口（command-line interface，CLI），会看到它的当前版本号。不过，在多数系统中，我们要自己动手安装 Heroku CLI（https://toolbelt.heroku.com）。使用云 IDE 的读者可以使用代码清单 1.23 中的命令安装。

代码清单 1.23：在云 IDE 中安装 Heroku CLI 的命令

```
$ source <(curl -sL https://cdn.learnenough.com/heroku_install)
```

上述命令执行完毕后，查看 Heroku CLI 的当前版本号，确认安装是否成功（得到的结果可能与这里不同）：

```
$ heroku --version
heroku/7.50.0 linux-x64 node-v12.16.2
```

确认 Heroku 命令行接口安装好之后，使用 `heroku` 命令登录，输入注册时填写的电子邮件地址和密码（`--interactive` 选项的作用是禁止 `heroku` 启动浏览器）：[20]

```
$ heroku login --interactive
```

最后，执行 `heroku create` 命令，在 Heroku 的服务器中创建一个位置，存放我们的应用，如代码清单 1.24 所示。

代码清单 1.24：在 Heroku 中创建一个新应用

```
$ heroku create
Creating app... done, blooming-bayou-75897
https://blooming-bayou-75897.herokuapp.com/ |
https://git.heroku.com/blooming-bayou-75897.git
```

20. 可惜，写作本书时 `heroku --interactive` 还不支持多因素身份验证，因此在云 IDE 中使用，要禁用 Heroku 账户的多因素身份验证功能。

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

Heroku 部署第一步

第一步，使用 Git 把主分支推送到 Heroku 中：

```
$ git push heroku master
```

(你可能会看到一些警告消息，现在先不管，7.5 节会解决。)

Heroku 部署第二步

其实没有第二步了。我们已经完成部署了。现在可以通过 `heroku create` 命令给出的地址（代码清单 1.24）查看刚刚部署的应用。[21] 以后如果想知道 Heroku 应用的 URL，执行 `heroku apps:info` 命令，如获取 Heroku 应用的信息，其中包括 URL 所示。

获取 Heroku 应用的信息，其中包括 URL

$ heroku apps:info === mysterious-atoll-47182 . . . Web URL: https://mysterious-atoll-47182.herokuapp.com/

访问 Heroku 应用的 URL，看到的结果如图 1.31 所示。这个页面与图 1.20 一样，但是现在这个应用运行在生产环境中。

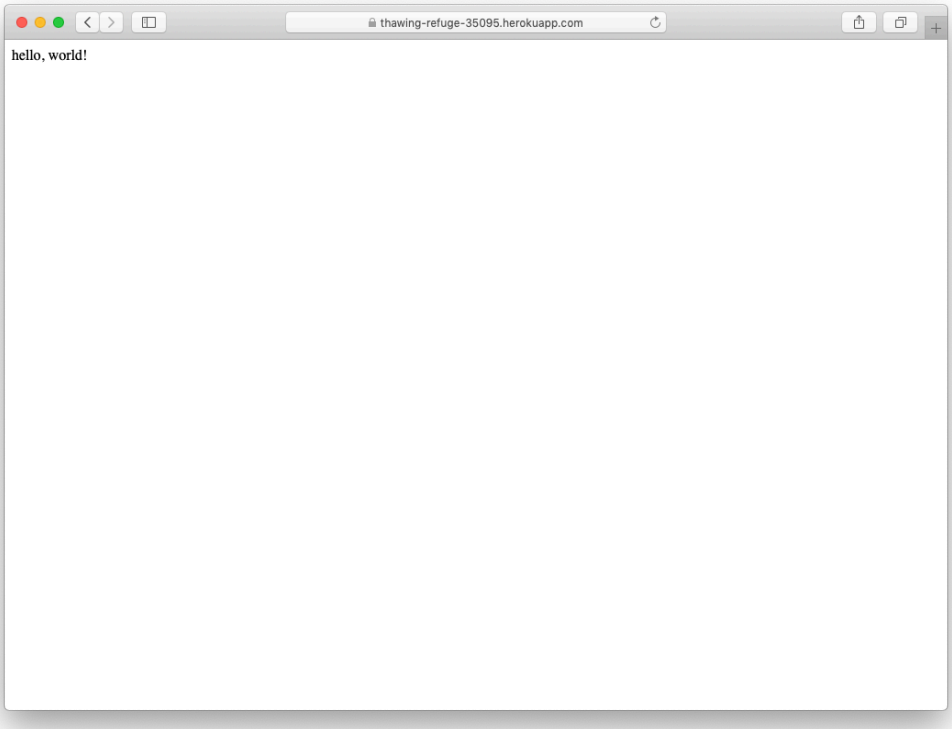


图 1.31：我们的第一个应用运行在 Heroku 中

21. 如果没用云 IDE，在本地可以执行 `heroku open` 命令自动在浏览器中打开网站。

练习

1. 跟前面的练习一样，想办法让生产环境中的应用显示“hola, mundo!”。
2. 跟前面的练习一样，想办法修改根路由，显示 `goodbye` 动作渲染的结果。部署时，看看能不能省略 `git push` 命令中的 `master`，只使用 `git push heroku`。

1.4.2 Heroku 命令

Heroku 命令行工具提供了很多命令，本节只简单介绍其中几个。下面花几分钟再介绍一个命令，其作用是重命名应用：

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

你别再使用这个名称了，笔者已经占用了。或许，现在你无需做这一步，使用 Heroku 提供的默认地址就行。不过，如果你真想重命名应用，出于安全考虑，可以使用一些随机或难以猜测的二级域名，例如：

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

使用这样随机的二级域名，只有你将地址告诉别人他们才能访问你的网站。为了让你一窥 Ruby 的强大，下面是笔者用来生成随机二级域名的代码，很精妙吧。[22]

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

除了二级域名，Heroku 还支持自定义域名。（其实，本书的网站就放在 Heroku 中。）在 Heroku 文档（https://devcenter.heroku.com）中可以查看更多关于自定义域名的信息以及 Heroku 相关的其他话题。

练习

1. 执行 `heroku help` 命令，查看 Heroku 命令列表。（如果 `heroku help` 命令的输出太多，超出了终端窗口，可以拖动滚动条，或者执行 `heroku help | less`，通过 Unix 管道把输出传给 `less` 命令。）找到显示应用日志的命令。
2. 使用前一题找到的命令查看应用的活动情况。应用刚刚发生了什么？（调试线上应用经常用到这个命令。）

1.5 小结

这一章做了很多事：安装、搭建开发环境，版本控制以及部署。下一章将在本章的基础上开发一个使用数据库的应用，让你看看 Rails 真正的本事。

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

> 我正在阅读《Ruby on Rails 教程》学习 Ruby on Rails！@flapybooks
> https://railstutorial-china.org/

22. 其实这行代码还可以使用 Ruby 的一个内置方法进一步精简。这个方法是 `sample`：`('a'..'z').to_a.sample(8).join`。感谢读者 Stefan Pochmann 指出这一点；在他告诉笔者之前，笔者甚至不知道有 `sample` 方法。

1.5.1 本章所学

- Ruby on Rails 是一个使用 Ruby 编程语言编写的 Web 开发框架。
- 在预先配置好的云环境中安装 Rails、新建应用，以及编辑文件都很简单。
- Rails 提供了命令行命令 `rails`，可用于新建应用（`rails new`）和启动本地服务器（`rails server`）。
- 添加了一个控制器动作，并且修改了根路由，最终开发出一个显示“hello, world!”的应用。
- 为了避免丢失数据，也为了协作，我们把应用的源码纳入 Git 版本控制系统，而且还把最终得到的代码推送到 GitHub 中的一个私有仓库里。
- 使用 Heroku 把应用部署到生产环境中。

1.6 排版约定

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

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

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

Rails 提供了很多可以在命令行中运行的命令。例如，在 1.2.2 节，我们使用 `rails server` 命令启动本地 Web 开发服务器：

```
$ rails server
```

与命令行提示符一样，本书也使用 Unix 惯用的目录分隔符（即一条斜线 `/`）。例如，演示应用的配置文件 `production.rb`，它的路径是：

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

上述文件路径相对于应用的根目录而言。在不同的系统中，根目录会有差别。在我们使用的云 IDE 中（1.1.1 节），根目录像下面这样：

```
/home/ubuntu/environment/sample_app/
```

所以，`production.rb` 文件的完整路径是：

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

通常，笔者会省略应用的路径，简写成 `config/environments/production.rb`。

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

本书涵盖 Rails 应用测试，所以最好知道某段代码会让测试组件失败（以红色表示）还是通过（以绿色表示）。为了方便，导致测试失败的代码使用“RED”标记，能让测试通过的代码使用“GREEN”标记。

最后，为了方便，本书使用两种排版方式，以便让代码清单更易于理解。第一，有些代码清单中包含一到多行高亮显示的代码，如下所示：

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

高亮显示的代码行一般用于标出该段代码清单中最重要的新代码，偶尔也用来表示当前代码清单和之前的代码清单之间的差异。第二，为了行文简洁，书中很多代码清单中都有竖排的点号，如下所示：

```
class User < ApplicationRecord
  .
  .
  .
  has_secure_password
end
```

这些点号表示有代码省略了，请不要直接复制。

第 2 章 玩具应用

本章我们要开发一个简单的玩具应用，展示 Rails 强大的功能。我们会使用脚手架快速生成应用，这样就能站在一定高度上概览 Ruby on Rails 编程的过程（也能大致了解 Web 开发）。正如旁注 2.1 所说，本书将采用与众不同的方法，循序渐进开发一个完整的演示应用，遇到新的概念都会详细说明。不过为了快速概览（也为了寻找成就感），无需对脚手架避而不谈。我们将通过 URL 与最终开发出来的玩具应用交互，了解 Rails 应用的结构，也第一次演示 Rails 使用的 REST 架构。

与后面的演示应用类似，这个玩具应用中有用户（users）和微博（microposts），因此算是一个简化的 Twitter 类应用。应用的功能还需要后续开发，而且开发过程中的很多步骤看起来很神秘，不过暂时不用担心：从第 3 章起将从零开始再开发一个类似的完整应用，笔者还会提供大量的资料供你后续阅读。你要有耐心，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，而要更深入地了解 Rails。

旁注 2.1：脚手架：更快、更简单、更诱人

Rails 出现伊始就吸引了众多目光，特别是 Rails 创始人 David Heinemeier Hansson 录制的著名的“15分钟开发一个博客”视频（https://youtu.be/Gzj723LkRJY）。这个视频及其衍生版本是窥探 Rails 强大功能一种很好的方式，推荐你看一下这些视频。不过事先提醒一下，这些视频中的演示能控制在 15 分钟以内，得益于一种叫做脚手架（scaffold）的功能，通过 Rails 提供的 `generate scaffold` 命令生成大量的代码。

写作本书时，笔者也想过使用脚手架，因为它更快、更简单、更诱人。不过脚手架生成的代码量多且复杂，会让初学者不明所以。这样虽然能学会脚手架的用法，但并不明白到底发生了什么。使用脚手架，你只是一个脚本生成器的使用者，无法提升对 Rails 的认识。

本书将采用一种不同的方式，虽然本章会用脚手架开发一个小型的玩具应用，但本书的核心是从第 3 章起开发的演示应用。在开发那个演示应用的每个阶段，我们只会编写少量的代码，易于理解但又具有一定的挑战性。通过这样一个过程，最终你会对 Rails 有较为深刻的理解，而且能灵活运用，开发出几乎任何类型的 Web 应用。

2.1 规划应用

这一节，我们要规划一下这个玩具应用。与 1.2 节一样，我们先使用 `rails new` 命令（指定 Rails 的版本号）生成应用的骨架：

```
$ cd ~/environment
$ rails _6.1.3_ new toy_app
$ cd toy_app/
```

如果使用1.1.1 节推荐的云 IDE，这个应用可以在第一个应用所在的环境中创建，没必要再新建一个环境。如果没看到文件，可以点击文件浏览器中的齿轮图标，然后选择“Refresh File Tree”（刷新文件树）。

然后，在文本编辑器中修改 `Gemfile` 文件，写入代码清单 2.1 中的内容。

代码清单 2.1：这个玩具应用的 `Gemfile` 文件

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

```
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.1.3'
gem 'puma',       '5.2.2'
gem 'sass-rails', '6.0.0'
gem 'webpacker',  '5.2.1'
gem 'turbolinks', '5.2.1'
gem 'jbuilder',   '2.10.0'
gem 'bootsnap',   '1.7.2', require: false

group :development, :test do
  gem 'sqlite3', '1.4.2'
  gem 'byebug',  '11.1.3', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.1.0'
  gem 'listen',                '3.2.1'
  gem 'spring',                '2.1.1'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',           '3.35.3'
  gem 'selenium-webdriver', '3.142.7'
  gem 'webdrivers',         '4.6.0'
end

group :production do
  gem 'pg', '1.2.3'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
# Uncomment the following line if you're running Rails
# on a native Windows system:
# gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

注意，代码清单 2.1 和代码清单 1.21 的内容完全一样。

与 1.4.1 节一样，安装 gem 时要设置 `bundle config set --local without 'production'`，禁止安装生产环境使用的 gem：

```
$ bundle _2.2.13_ config set --local without 'production'
$ bundle _2.2.13_ install
```

如 1.2.1 节所述，可能还要运行 `bundle update`（旁注 1.2）。此外，你可能还会看到一个提醒消息，以“The dependency tzinfo-data”开头，置之不理即可。

最后，把这个玩具应用纳入 Git 版本控制系统：

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

你还可以按照 1.3.3 节中的步骤，在 GitHub 中创建一个新仓库（图 2.1，别忘了设为私有仓库），然后把代码推送到这个远程仓库中：

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

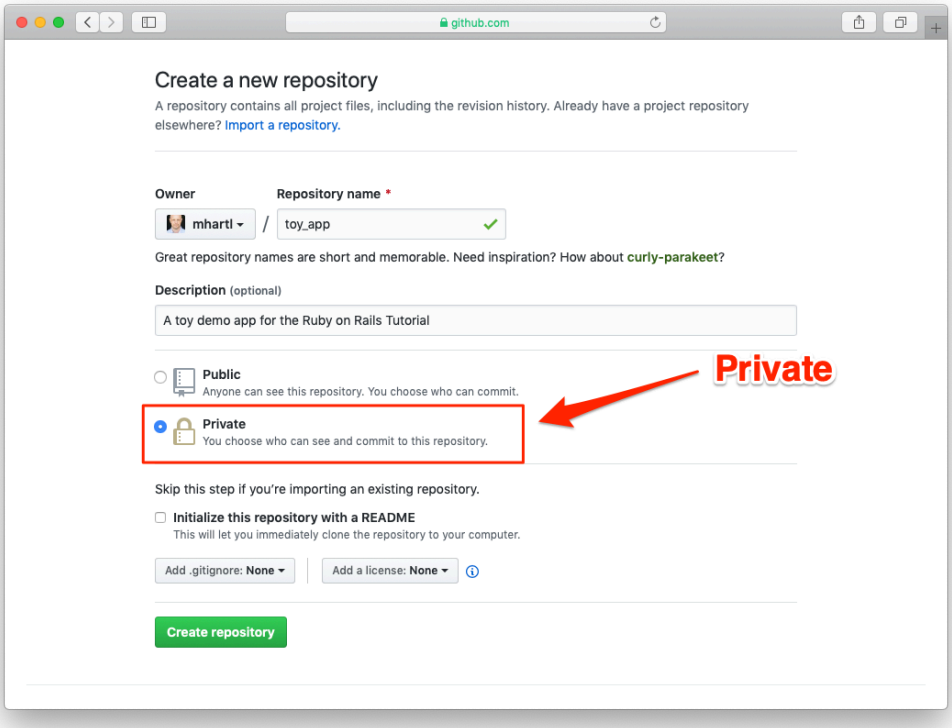


图 2.1：在 GitHub 中为这个玩具应用创建一个仓库

越早部署应用越好。建议你按照 1.2.4 节所述的步骤做，修改代码清单 2.2 和代码清单 2.3。

代码清单 2.2：在 Application 控制器中添加 hello 动作
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base

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

代码清单 2.3：设置根路由
config/routes.rb

```
Rails.application.routes.draw do
  root 'application#hello'
end
```

然后，提交改动，再推送到 Heroku 和 GitHub 中——最好让两份副本保持同步：

```
$ git commit -am "Add hello"
```

```
$ heroku create
$ git push && git push heroku master
```

这里，我们使用 `&&`（读作“and”）运算符把推送到 GitHub 和 Heroku 两个操作合并在一起，只有前一个命令成功执行，后一个命令才会执行。

与 1.4 节一样，你可能会看到一些警告消息，现在先不去管它。7.5 节会解决。除了 Heroku 为应用提供的地址之外，输出的内容应该与图 1.31 一样。

2.1.1 User 模型

下面要开发这个应用了。一般来说，开发 Web 应用的第一步是创建数据模型（data model，https://en.wikipedia.org/wiki/Data_model）。模型表示应用所需的结构，包括模型之间的关系。这个玩具应用是个 Twitter 类微博，只有用户和简短的文章（微博）。本节先为这个应用添加 `User` 模型，2.1.2 节再添加 `Micropost` 模型。

网络中有多少不同的注册表单，就有多少定义用户数据模型的方式。简单起见，我们将使用一种最简可用的方式。这个玩具应用的用户有一个唯一的标识 `id`（`integer` 类型）、一个公开的名字 `name`（`string` 类型）和一个电子邮件地址 `email`（也是 `string` 类型）。电子邮件地址将作为唯一的用户名使用。（注意，现在这个模型没有 `password` 属性，毕竟这只是一个“玩具”应用；第 6 章再添加密码。）`User` 模型的结构如图 2.2 所示。

users	
id	integer
name	string
email	string

图 2.2：`User` 数据模型

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

2.1.2 Micropost 模型

前面说过，微博是简短的文章，Twitter 称其为“推文”。`Micropost` 数据模型比 `User` 模型还要简单：微博只要一个 `id` 和表示微博内容的 `content`（`text` 类型）字段即可。[1] 此外还有一个比较复杂的字段要实现，这个字段把微博和用户关联起来。我们使用 `user_id` 存储微博的属主。最终得到的 `Micropost` 数据模型如图 2.3 所示。

2.3.3 节会介绍怎样使用 `user_id` 字段简单实现一个用户拥有多篇微博的功能。（第 13 章有更完整的说明。）

1. 微博的内容很短，`string` 类型可能就够了，但使用 `text` 类型更能表明我们的意图，而且也便于以后放宽微博长度限制。其实，Twitter 已经把 140 字限制放宽到了 280 字，这也是我们选择使用 `text` 类型的原因：`string` 类型最多可存储 255 个字符（2^8-1），对字数限制为 140 的推文是够用了，但是放不下 280 个字符的推文；而使用 `text` 类型则没有这个问题。

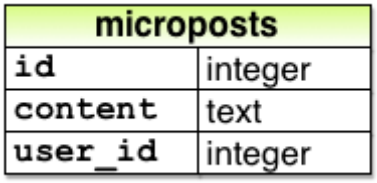


图 2.3：Micropost 数据模型

2.2 Users 资源

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

把 scaffold 传给 rails generate 命令就可以使用 Rails 的脚手架了。传给 scaffold 的参数是资源名的单数形式（这里是 User）[2]，后面可以再跟一些可选参数，指定数据模型中的字段：

```
$ rails generate scaffold User name:string email:string
      invoke  active_record
      create    db/migrate/<timestamp>_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      invoke  resource_route
       route    resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/controllers/users_controller_test.rb
      create      test/system/users_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      test_unit
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      create      app/views/users/_user.json.jbuilder
      invoke  assets
```

2. 脚手架中使用的名称与模型一样，是单数；而资源和控制器使用复数。因此，这里要使用 User，而不是 Users。

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

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

接下来我们要用 `rails db:migrate` 命令迁移（migrate）数据库，如代码清单 2.4 所示。

代码清单 2.4：迁移数据库

```
$ rails db:migrate
== CreateUsers: migrating ====================================
-- create_table(:users)
   -> 0.0027s
== CreateUsers: migrated (0.0036s) ===========================
```

上述命令的作用是使用新的 `User` 数据模型更新数据库。（从 6.1.1 节开始会深入学习数据库迁移。）

执行代码清单 2.4 中的迁移之后，可以新打开一个终端标签页（图 1.15），运行本地 Web 服务器。如果使用云 IDE，在此之前要先添加 1.2.2 节中的设置，允许连接本地服务器（代码清单 2.5）。

代码清单 2.5：允许连接本地 Web 服务器
config/environments/development.rb

```
Rails.application.configure do
  .
  .
  .
  # 允许 Cloud9 连接
  config.hosts.clear
end
```

然后，像 1.2.2 节那样运行 Rails 服务器：

```
$ rails server
```

现在，这个玩具应用应该可以在本地服务器中访问了，结果与 1.2.2 节一样。访问根 URL，我们会看到与图 1.20 一样的“hello, world!”页面。

2.2.1 浏览用户相关的页面

2.2 节使用脚手架生成 `Users` 资源时生成了很多用来处理用户的页面，包括列出所有用户的页面 /users、创建新用户的页面 /users/new。本节的目的是走马观花地浏览一下这些用户相关的页面。浏览时你会发现表 2.1 很有用，表中显示了页面和 URL 之间的对应关系。

表 2.1：`Users` 资源中页面和 URL 的对应关系

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

URL	动作	作用
/users/1/edit	`edit`	编辑 ID 为 1 的用户

先看显示应用中所有用户的页面，这个页面叫索引页，路径是 /users。若想打开这个页面，点击浏览器中的地址栏，在根 URL 后面添加“/users”（图 2.4），把整个 URL 变成 https://www.example.com/users 形式。

图 2.4：在根 URL 后面添加“/users”

和预期一样，目前还没有用户，如图 2.5 所示。

如果想创建新用户，可以点击图 2.5 中的“New User”（新建用户）链接，访问 /users/new 路径上的页面，如图 2.6 所示。第 7 章将把这个页面打造成用户注册页面。

我们可以在表单中填入名字和电子邮件地址，然后点击“Create User”（创建用户）按钮创建一个用户。此时，浏览器会转向这个用户的页面，即 /users/1，如图 2.7 所示。（页面中显示的绿色文字是闪现消息，7.4.2 节介绍。）注意，这个页面的 URL 是 /users/1。你可能猜到了，这里的 `1` 就是图 2.2 中的用户 `id`。7.1 节将把这个页面打造成用户的资料页。

如果想修改用户的信息，点击“Edit”（编辑）链接，访问编辑页面，即 /users/1/edit（图 2.8）。修改用户信息后点击“Update User”（更新用户）按钮就更改了这个玩具应用中该用户的信息（图 2.9）。（第 6 章将详细介绍，用户的信息存储在后端数据库中。）我们将在 10.1 节为演示应用添加编辑和更新用户信息的功能。

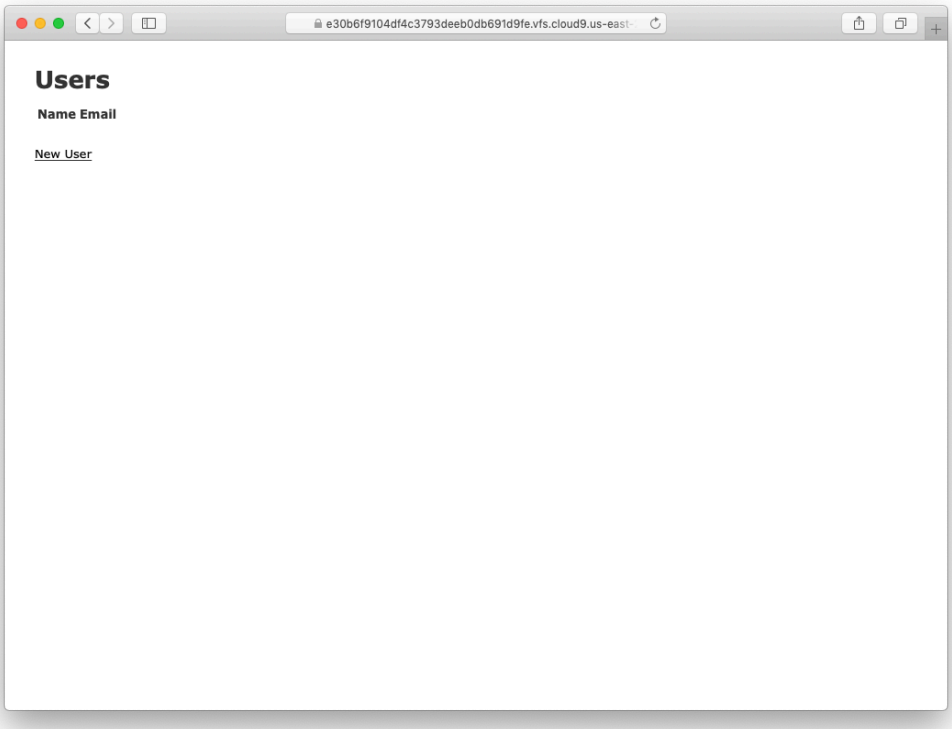


图 2.5：Users 资源的索引页（/users）

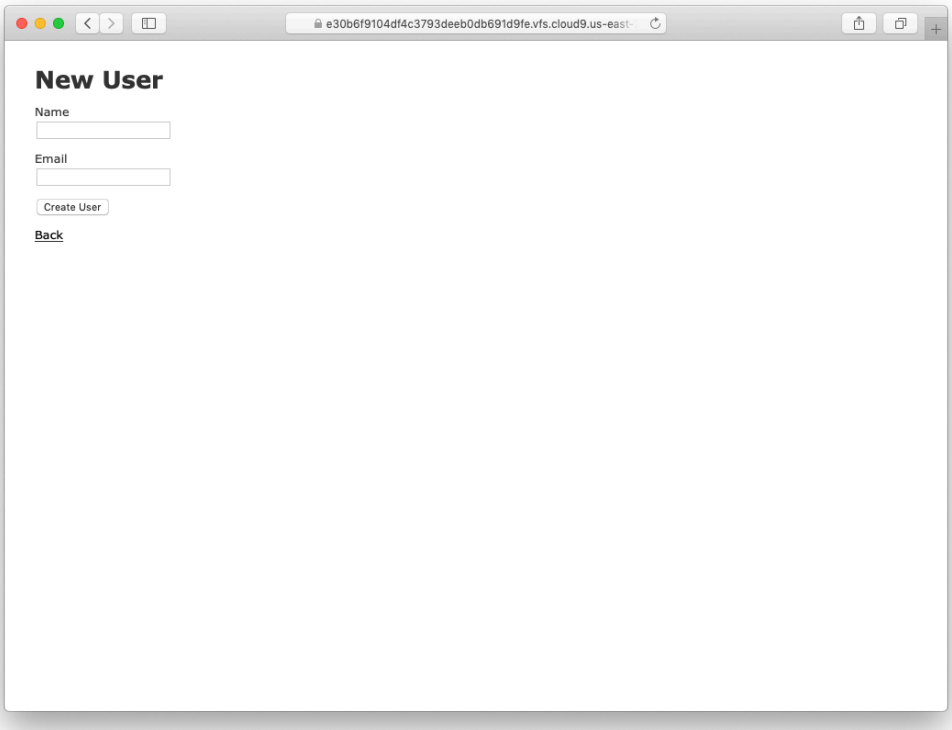


图 2.6：新建用户页面（/users/new）

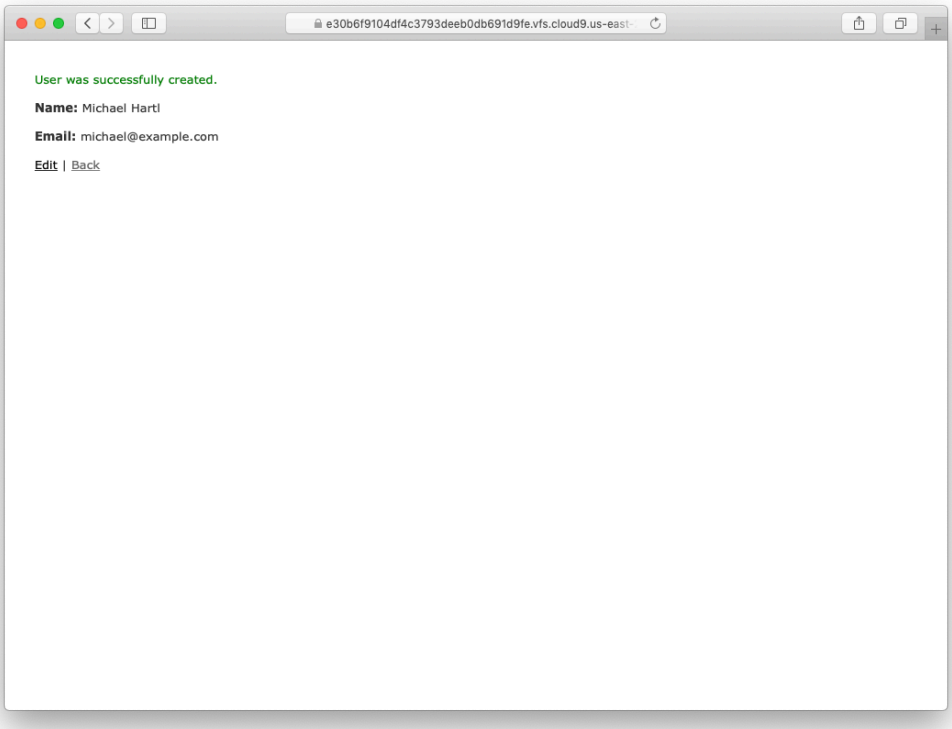


图 2.7：显示某个用户的页面（/users/1）

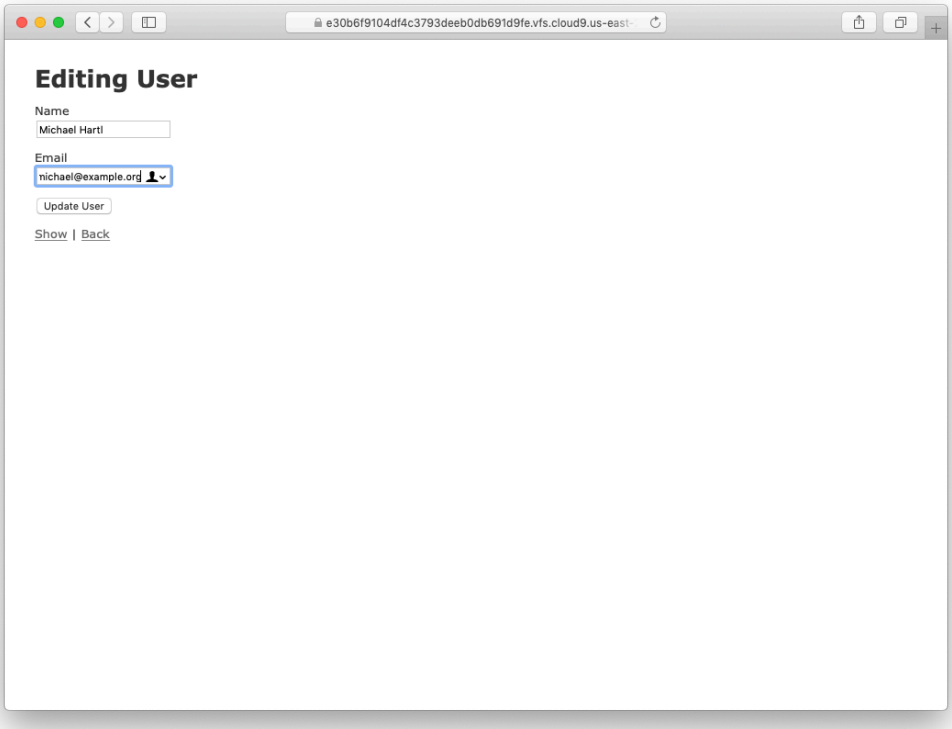


图 2.8：编辑用户信息的页面（/users/1/edit）

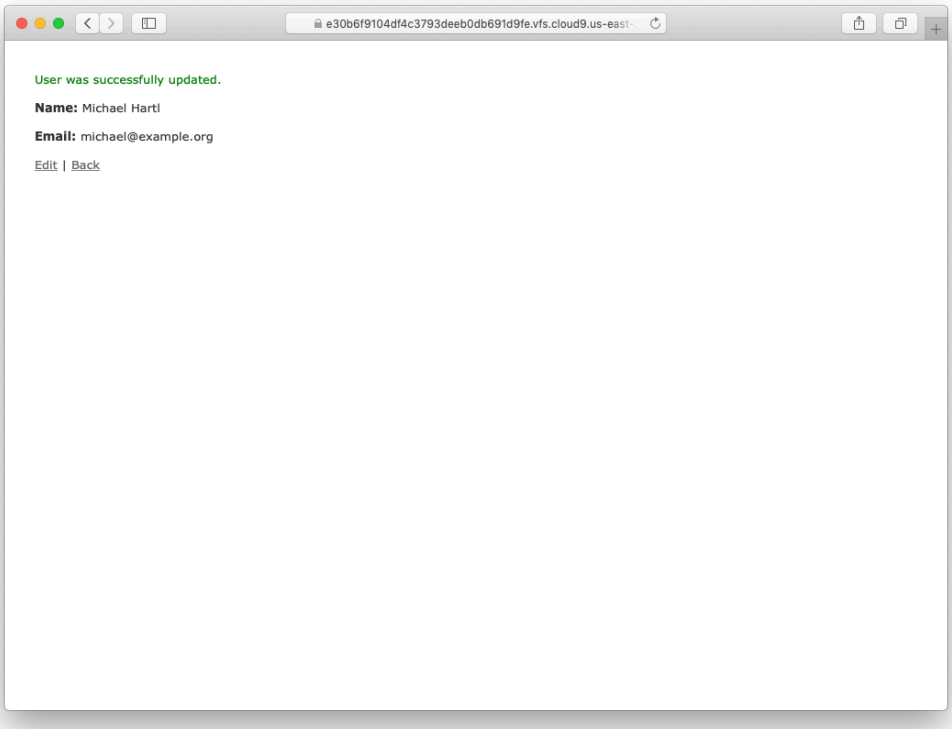


图 2.9：更新信息后的用户页面

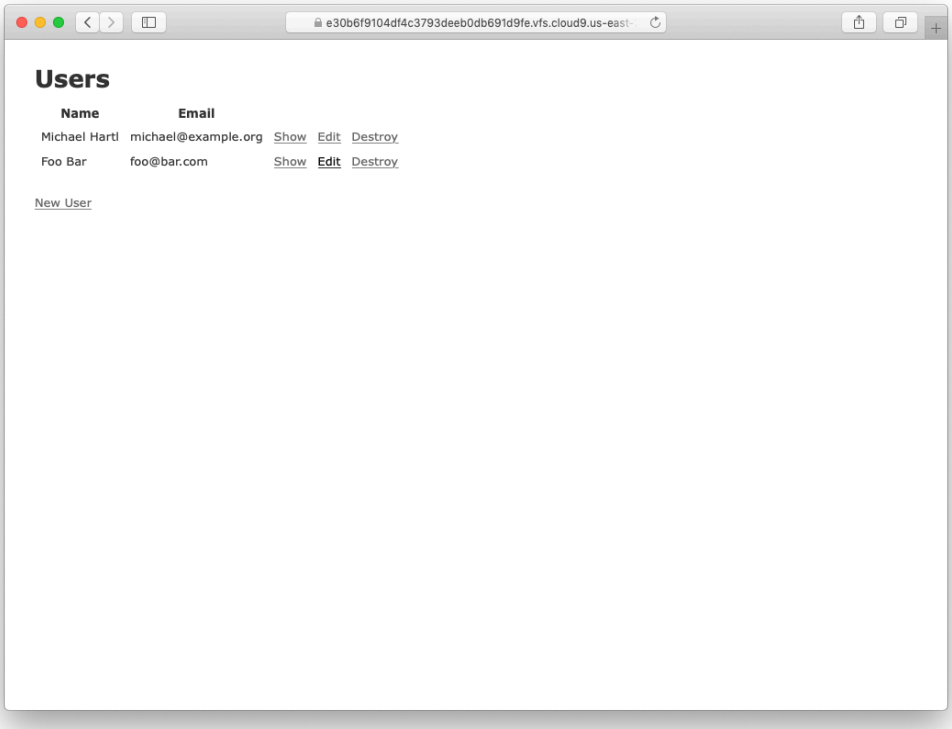


图 2.10：创建第二个用户后的用户索引页（/users）

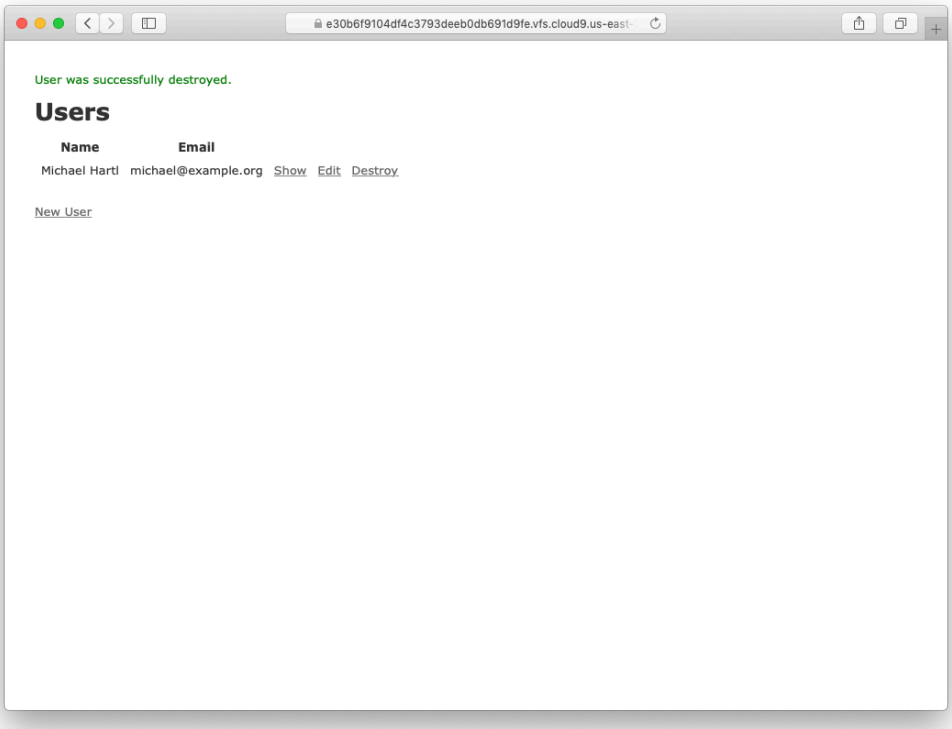


图 2.11：删除一个用户

现在回到 /users/new 页面，在表单中填写信息，再创建一个用户。然后访问用户索引页，结果如图 2.10 所示。7.1 节将美化这个显示所有用户的页面。

我们已经看了创建、显示和编辑用户的页面，最后要看删除用户的页面（图 2.11）。点击图 2.11 中所示的链接后，第二个用户将被删除，现在索引页面就只剩一个用户了。（如果这个操作不成功，确认浏览器是否启用了 JavaScript。Rails 通过 JavaScript 发送删除用户的请求。）10.4 节将为演示应用实现用户删除功能，而且仅限于管理员级别的用户才能执行这项操作。

练习

1. （如果你了解 CSS）创建一个新用户，然后使用浏览器中的 HTML 审查工具找出“User was successfully created.”文本的 CSS ID。刷新页面后会发生什么？
2. 如果创建用户时只填写名字，而没填写电子邮件地址，会发生什么？
3. 如果创建用户时填写的电子邮件地址无效，例如填写的是“@example.com”，会发生什么？
4. 删除前几题创建的用户。删除用户时，Rails 会显示消息吗？

2.2.2 MVC 实战

我们已经快速概览了 `Users` 资源，下面从 MVC（1.2.3 节）的视角出发，审视其中某些部分。我们将分析在浏览器中访问用户索引页（/users）的过程，了解一下 MVC（图 2.12）。

图中各步的说明如下：

1. 浏览器向 /users 发送请求；

2. Rails 路由器把 /users 交给 `Users` 控制器的 `index` 动作处理；
3. `index` 动作要求 `User` 模型检索所有用户（`User.all`）；
4. `User` 模型从数据库中读取所有用户；
5. `User` 模型把所有用户组成的列表返回给控制器；
6. 控制器把所有用户赋值给 `@users` 变量，然后传入 `index` 视图；
7. 视图使用嵌入式 Ruby 把页面渲染成 HTML；
8. 控制器把 HTML 送回浏览器。[3]

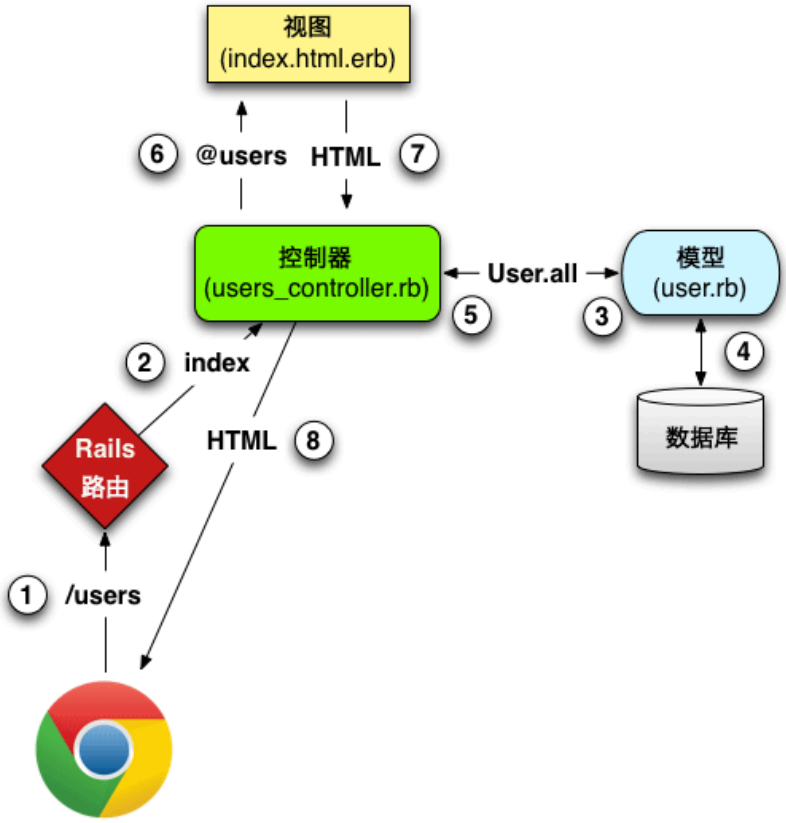


图 2.12：Rails 中的 MVC 架构详解

下面详细分析这个过程。首先，浏览器发送请求（第 1 步）。这一步可以直接在浏览器地址栏中输入地址，也可以点击网页中的链接。请求到达 Rails 路由器（第 2 步），路由器根据 URL（以及请求的类型，参见旁注 3.2）把请求分配给合适的控制器动作。把 `Users` 资源中相关的 URL 映射到控制器动作的代码如代码清单 2.6 所示。高亮显示的那行代码会按照表 2.1 中的对应关系做映射。`:users` 这个写法看着很奇怪，它是一个符号（symbol），4.3.3 节介绍。

代码清单 2.6：Rails 路由，为 `Users` 资源定义一条规则
config/routes.rb

```
Rails.application.routes.draw do
```

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

```
  resources :users
  root 'application#hello'
end
```

既然打开路由文件了，那就花点儿时间把根路由改为用户索引页吧。修改之后，访问根地址就会显示 /users 页面。我们在代码清单 2.3 中添加了根路由：

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

上述规则把根路由指向 `Application` 控制器中的 `hello` 动作。现在，我们想使用 `Users` 控制器中的 `index` 动作，因此要按照代码清单 2.7 所示的代码修改。

代码清单 2.7：把根路由指向 `Users` 控制器中的动作
config/routes.rb

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

一个控制器中有多个动作，2.2.1 节浏览的页面对应于 `Users` 控制器中的不同动作。脚手架生成的控制器代码摘要如代码清单 2.8 所示。注意 `class UsersController < ApplicationController` 这种写法，在 Ruby 中这表示类继承。（2.3.4 节会简要介绍继承，4.4 节再做详细说明。）

代码清单 2.8：`Users` 控制器代码摘要
app/controllers/users_controller.rb

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

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def edit
    .
    .
    .
  end
```

```
    def create
      .
      .
      .
    end

    def update
      .
      .
      .
    end

    def destroy
      .
      .
      .
    end
  end
```

你可能注意到了，动作的数量比我们看过的页面数量多，`index`、`show`、`new` 和 `edit` 对应于 2.2.1 节介绍的页面。此外还有一些其他动作，`create`、`update` 和 `destroy`。这些动作一般不直接渲染页面（不过有时也会），只是修改数据库中保存的用户数据。

表 2.2 列出了控制器的全部动作，这些动作就是 Rails 对 REST 架构（旁注 2.2）的实现。REST 架构由计算机科学家 Roy Fielding 提出，意思是“表现层状态转化”（Representational State Transfer）。[4] 注意表 2.2 中的内容，有些部分有重叠。例如 `show` 和 `update` 两个动作都映射到 /users/1 这个地址上，二者的区别是响应的 HTTP 请求方法不同。3.3 节将更详细地介绍 HTTP 请求方法。

表 2.2：代码清单 2.6 中 `Users` 资源生成的符合 REST 架构的路由

HTTP 请求	URL	动作	作用
`GET`	/users	`index`	列出所有用户
`GET`	/users/1	`show`	显示 ID 为 1 的用户
`GET`	/users/new	`new`	显示创建新用户的页面
`POST`	/users	`create`	创建新用户
`GET`	/users/1/edit	`edit`	显示 ID 为 1 的用户的编辑页面
`PATCH`	/users/1	`update`	更新 ID 为 1 的用户
`DELETE`	/users/1	`destroy`	删除 ID 为 1 的用户

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

旁注 2.2：表现层状态转化（REST）

如果阅读过一些 Ruby on Rails Web 开发相关的资料，你会发现很多地方都提到了“REST”，它是“表现层状态转化”（REpresentational State Transfer）的简称。REST 是一种架构风格，用于开发分布式、基于网络的系统和软件应用，例如万维网和 Web 应用。REST 理论很抽象，在 Rails 应用中，REST 意味着大多数组件（例如用户和微博）都被模型化为资源，可以创建、读取、更新和删除。这些操作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。3.3 节，特别是旁注 3.2 将更详细地介绍 HTTP 请求。

作为 Rails 应用开发者，REST 开发方式能帮助你决定编写哪些控制器和动作：你只需简单地把可以创建、读取、更新和删除的资源理清就可以了。对本章的“用户”和“微博”来说，这一过程非常明确，因为它们都是很自然的资源形式。在第 14 章将看到，使用 REST 架构可以通过一种自然而便捷的方式解决棘手问题（“关注用户”功能）。

为了探明 `Users` 控制器与 `User` 模型之间的关系，来看一下简化后的 `index` 动作，如代码清单 2.9 所示。（要阅读不完全能理解的代码也体现了“技术是复杂的”。）

代码清单 2.9：这个玩具应用中简化的 `index` 动作
app/controllers/users_controller.rb

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

`index` 动作中有一行代码，`@users = User.all`（图 2.12 中的第 3 步），让 `User` 模型从数据库中检索所有用户（第 4 步），然后把结果赋值给 `@users` 变量（读作“at-users”，第 5 步）。

`User` 模型的代码见代码清单 2.10。这段代码看似简单，但是通过继承具备了很多功能（见 2.3.4 节 和 4.4 节）。具体而言，使用 Rails 中名为 Active Record 的库后，`User.all` 就能返回数据库中的所有用户。

代码清单 2.10：玩具应用中的 `User` 模型
app/models/user.rb

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

定义 `@users` 变量后，控制器再调用视图（第 6 步）。视图的代码如代码清单 2.11 所示。以 `@` 开头的变量是实例变量（instance variable），在视图中自动可用。这里，`index.html.erb` 视图中的代码（代码清单 2.11）遍历 `@users`，为每个用户生成一行 HTML。（你现在可能读不懂这些代码，这里只是让你看一下视图是什么样子。）

代码清单 2.11：用户索引页的视图
app/views/users/index.html.erb

```
<p id="notice"><%= notice %></p>

<h1>Users</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @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 %>
  </tbody>
</table>

<br>

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

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

练习

1. 参照图 2.12，写出访问 /users/1/edit 页面的步骤。
2. 在脚手架生成的代码中找出前一题从数据库中检索用户的代码。提示：在名为 `set_user` 的特殊位置中。
3. 编辑用户页面的视图文件，其名称是什么？

2.2.3 这个 Users 资源的不足

脚手架生成的 `Users` 资源虽然能够让你大致了解 Rails，但也有一些不足：

- **没有验证数据**。`User` 模型会接受空名字和无效的电子邮件地址，而不报错。
- **没有验证身份**。没实现登录和退出功能，随意一个用户都可以进行任何操作。
- **没有测试**。也不是完全没有，脚手架会生成一些基本的测试，不过很粗糙也不灵便，没有针对数据验证和身份验证的测试，更别说针对其他功能的测试了。
- **没样式，没布局**。没有共用的样式和网站导航。

- **没真正理解**。如果你能读懂脚手架生成的代码，就不需要阅读这本书了。

2.3 Microposts 资源

我们已经生成并浏览了 Users 资源，现在要生成 Microposts 资源。阅读本节时，建议你和 2.2 节对比一下。你会发现这两个资源在很多方面都是一致的。通过这样重复生成资源，我们可以更好地理解 Rails 中的 REST 架构。在这样的早期阶段看一下 Users 资源和 Microposts 资源的相同之处，也是本章的主要目的之一。

2.3.1 概览 Microposts 资源

与 Users 资源一样，我们将使用 rails generate scaffold 命令生成 Microposts 资源的代码，不过这一次要实现图 2.3 中的数据模型：[5]

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

5. 与生成 Users 资源使用的脚手架命令一样，生成 Microposts 资源的脚手架也要使用单数形式，即 generate Micropost。

然后，跟 2.2 节一样，我们要执行迁移，更新数据库，使用新的数据模型：

```
$ rails 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。你可能猜到了，脚手架还会更新 Rails 的路由文件，为 `Microposts` 资源加入一条规则，如代码清单 2.12 所示。[6] 与 `Users` 资源类似，`resources :microposts` 把微博相关的 URL 映射到 `Microposts` 控制器上，如表 2.3 所示。

代码清单 2.12：Rails 的路由，有一条针对 `Microposts` 资源的新规则
config/routes.rb

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

表 2.3：代码清单 2.12 中 `Microposts` 资源生成的符合 REST 架构的路由

HTTP 请求	URL	动作	作用
`GET`	/microposts	`index`	列出所有微博
`GET`	/microposts/1	`show`	显示 ID 为 1 的微博
`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`	显示 ID 为 1 的微博的编辑页面
`PATCH`	/microposts/1	`update`	更新 ID 为 1 的微博
`DELETE`	/microposts/1	`destroy`	删除 ID 为 1 的微博

`Microposts` 控制器的代码简化后如代码清单 2.13 所示。注意，除了把 `UsersController` 换成 `MicropostsController` 之外，这段代码和代码清单 2.8 没什么区别。这说明了两个资源在 REST 架构中的共同之处。

代码清单 2.13：简化后的 `Microposts` 控制器
app/controllers/microposts_controller.rb

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

6. 与代码清单 2.12 相比，脚手架生成的代码可能会有额外的空行。无须担心，因为 Ruby 会忽略额外的空行。

```
    end

    def show
      .
      .
      .
    end

    def new
      .
      .
      .
    end

    def edit
      .
      .
      .
    end

    def create
      .
      .
      .
    end

    def update
      .
      .
      .
    end

    def destroy
      .
      .
      .
    end
  end
```

在微博索引页面（图 2.13）中点击“New Micropost”（新建微博）链接，打开发布微博的页面（/microposts/new），输入一些内容，发布一篇微博，如图 2.14 所示。

既然已经打开这个页面了，那就多发布几篇微博，并且确保至少把一篇微博的 `user_id` 设为 `1`，把微博赋予 2.2.1 节中创建的第一个用户。结果应该和图 2.15 类似。

练习

1. （如果你了解 CSS）发布一篇新微博，然后使用浏览器中的 HTML 审查工具找出“Micropost was successfully created.”文本的 CSS ID。刷新页面后会发生什么？
2. 发布微博时不输入内容也不指定用户 ID 试试。
3. 发布一篇内容超过 140 字（例如维基百科中介绍 Ruby 的第一段，https://en.wikipedia.org/wiki/Ru-

by_(programming_language)）的微博试试。

4. 删除前几题创建的微博。

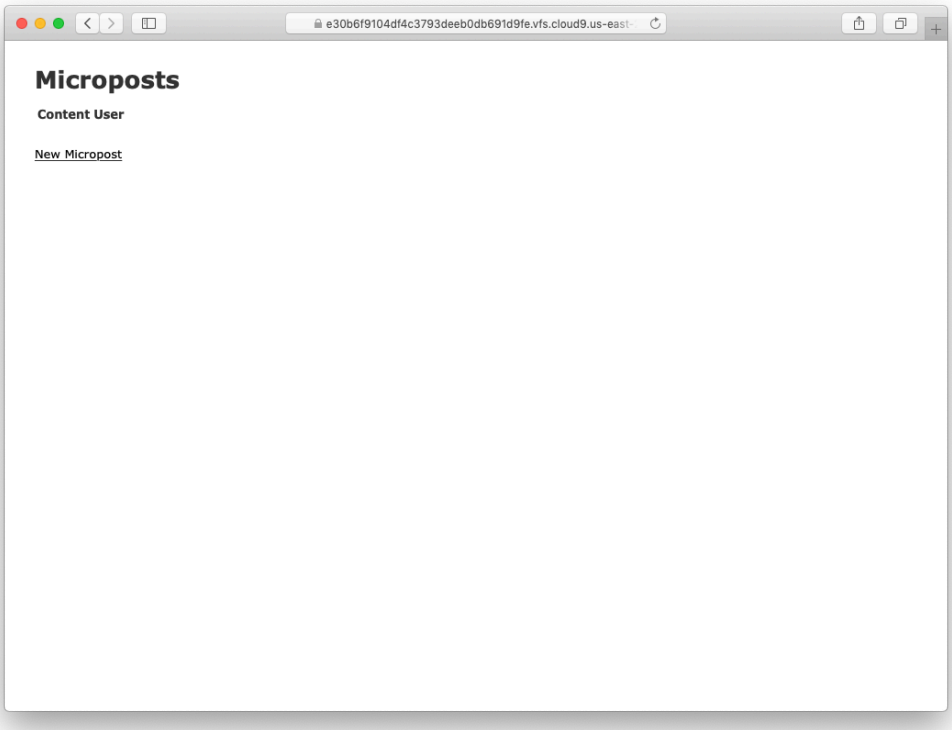


图 2.13：微博索引页面（/microposts）

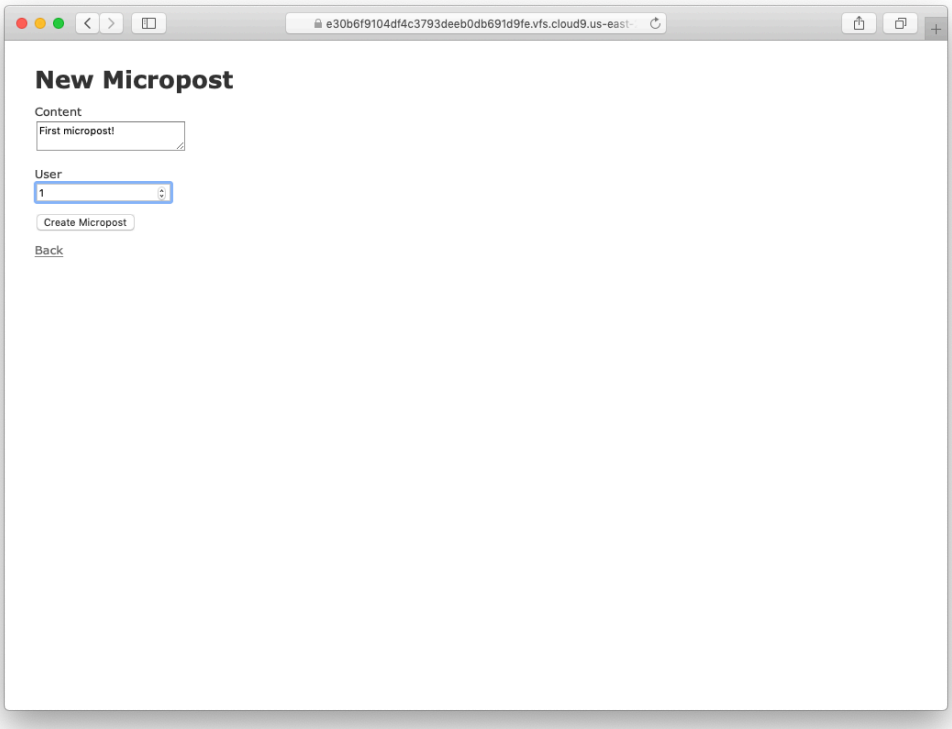


图 2.14：发布微博的页面（/microposts/new）

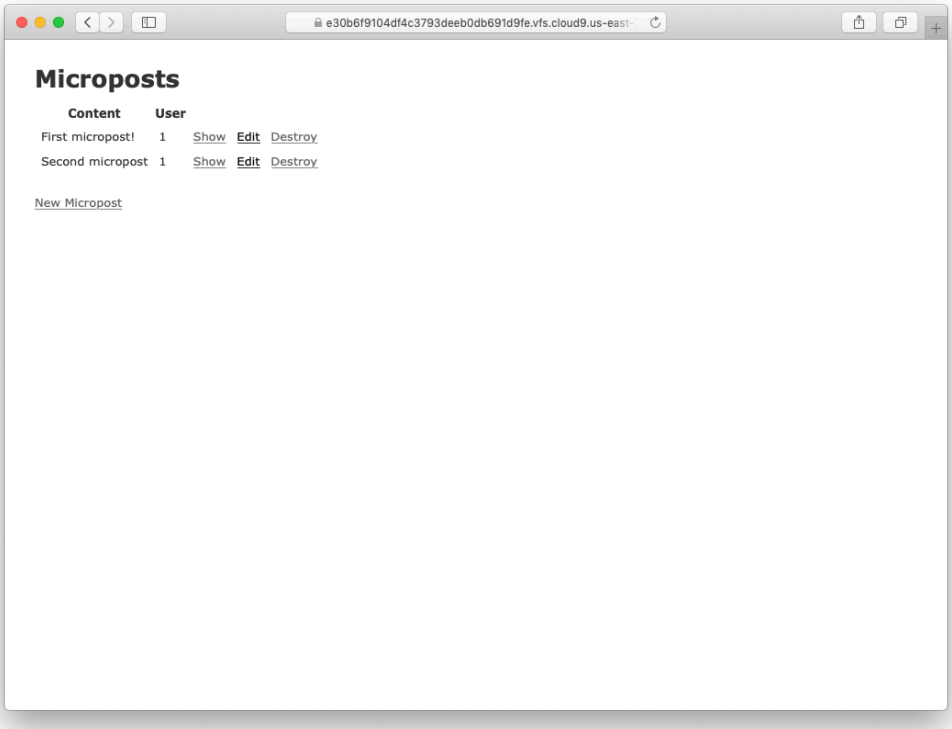


图 2.15：微博索引页中显示几篇微博

2.3.2 限制微博的长度

为了称得上“微博”这个名字，内容的长度要做限制。在 Rails 中实现这种限制很简单，使用验证（validation）功能即可。要限制微博的长度最多为 140 个字符（就像 Twitter 一开始那样），我们可以使用长度验证。在文本编辑器或 IDE 中打开 `app/models/micropost.rb` 文件，写入代码清单 2.14 中的代码。

代码清单 2.14：限制微博的长度最多为 140 个字符
app/models/micropost.rb

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

这段代码看起来可能很神秘，我们会在 6.2 节详细介绍验证。如果我们在发布微博的页面输入超过 140 个字符的内容，就能看到这个验证的作用了。如图 2.16 所示，Rails 会渲染错误消息，提示微博的内容太长了。（7.3.3 节将详细介绍错误消息。）

练习

1. 使用前一节练习中的那段文字发布微博，这一次有什么变化呢？
2. （如果你了解 CSS）使用浏览器中的 HTML 审查工具找到前一题那个错误消息的 CSS ID。

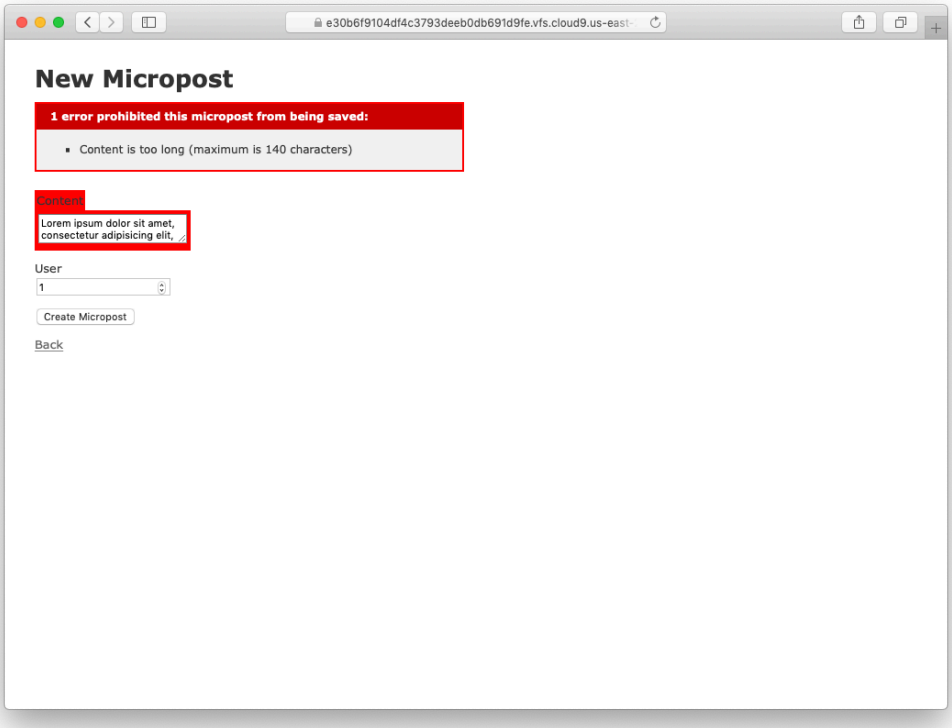


图 2.16：发布微博失败时显示的错误消息

2.3.3 一个用户拥有多篇微博

Rails 最强大的功能之一，是可以在不同的数据模型之间建立关联（association）。对这里的 `User` 模型而言，

每个用户可以拥有多篇微博。我们可以更新 `User` 模型（代码清单 2.15）和 `Micropost` 模型（代码清单 2.16）的代码实现这种关联。

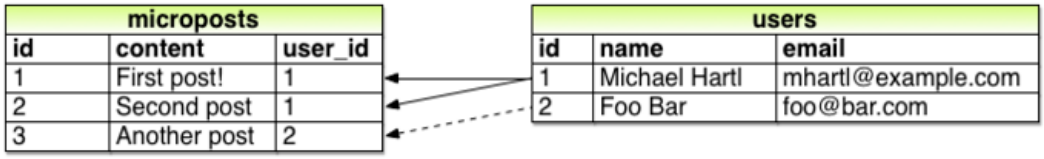
代码清单 2.15：一个用户拥有多篇微博
app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts
end
```

代码清单 2.16：一篇微博属于一个用户
app/models/micropost.rb

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

我们可以把这种关联用图 2.17 表示出来。因为 `microposts` 表中有 `user_id` 这一列，所以 Rails（通过 Active Record）能把微博和各个用户关联起来。



microposts

id	content	user_id
1	First post!	1
2	Second post	1
3	Another post	2

users

id	name	email
1	Michael Hartl	mhartl@example.com
2	Foo Bar	foo@bar.com

图 2.17：微博和用户之间的关联

在第 13 章和第 14 章，我们将利用微博和用户之间的关联显示用户的所有微博，还将生成一个和 Twitter 类似的动态流。现在，我们可以在控制台（console）中检查用户与微博之间的关联。控制台是与 Rails 应用交互常用的工具。在命令行中执行 `rails console` 命令，启动控制台。然后输入 `User.first`，从数据库中检索第一个用户，并把得到的数据赋值给 `first_user` 变量（代码清单 2.17）：[7]（笔者在这段代码的最后一行加上了 `exit`，说明如何退出控制台。在多数系统中也可以按 Ctrl-D 键退出控制台。）[8]

代码清单 2.17：使用 Rails 控制台查看应用的状态

```
$ rails console
>> first_user = User.first
   (0.5ms)  SELECT sqlite_version(*)
  User Load (0.2ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC
  LIMIT ?  [["LIMIT", 1]]
 => #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
 created_at: "2019-08-20 00:39:14", updated_at: "2019-08-20 00:41:24">
>> first_user.microposts
  Micropost Load (3.2ms)  SELECT "microposts".* FROM "microposts" WHERE
  "microposts"."user_id" = ? LIMIT ?  [["user_id", 1], ["LIMIT", 11]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Micropost id: 1, content:
 "First micropost!", user_id: 1, created_at: "2019-08-20 02:04:13", updated_at:
```

7. 你的控制台可能会显示类似 `2.6.3 :001 >` 的提示符，但示例中使用 `>>` 代替，因为不同的 Ruby 版本显示的提示符不同。
8. 与“Ctrl-C”一样，这里大写的“D”指代键盘上的按键，不是大写字母“D”，因此按 Ctrl 键的同时不用按住 Shift 键。

```
 "2019-08-20 02:04:13">, #<Micropost id: 2, content: "Second micropost",
 user_id: 1, created_at: "2019-08-20 02:04:30", updated_at: "2019-08-20
 02:04:30">]>
>> micropost = first_user.microposts.first
  Micropost Load (0.2ms)  SELECT "microposts".* FROM "microposts" WHERE
  "microposts"."user_id" = ? ORDER BY "microposts"."id" ASC LIMIT ?
  [["user_id", 1], ["LIMIT", 1]]
 => #<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
 "2019-08-20 02:04:13", updated_at: "2019-08-20 02:04:13">
>> micropost.user
 => #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
 created_at: "2019-08-20 00:39:14", updated_at: "2019-08-20 00:41:24"
>> exit
```

这段代码内容很多，从中梳理出相关的部分也是对技术能力的一种锻炼。上述输出中包含返回值，这些返回值是原始 Ruby 对象和相应的结构化查询语言（Structured Query Language，SQL）代码。

除了使用 `User.first` 检索第一个用户之外，代码清单 2.17 还演示了另外两个操作：（1）使用 `first_user.microposts` 获取 `user_id` 的值与 `first_user` 的 ID（1）相同的用户发布的所有微博；（2）使用 `micropost.user` 获取某篇微博所属的用户。第 4 章将详细说明这里涉及到的 Ruby 句法。在第 13 章和第 14 章中，我们会更深入地学习关联。

练习

1. 编辑显示用户的页面，显示用户发布的第一篇微博。（根据文件中的其他内容猜测所需的句法。）访问 /users/1，确认改动是正确的。
2. 代码清单 2.18 添加了一个存在性验证，确保微博的内容不能为空。确认这个验证的行为与图 2.18 中一样。
3. 把代码清单 2.19 中的 `FILL_IN` 替换成相应的代码，为 `User` 模型的 `name` 和 `email` 属性添加存在性验证。效果如图 2.19 所示。

代码清单 2.18：验证微博内容存在性的代码
app/models/micropost.rb

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

代码清单 2.19：为 User 模型添加存在性验证
app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts
  validates FILL_IN, presence: true # 把 FILL_IN 替换成正确的代码
  validates FILL_IN, presence: true # 把 FILL_IN 替换成正确的代码
end
```

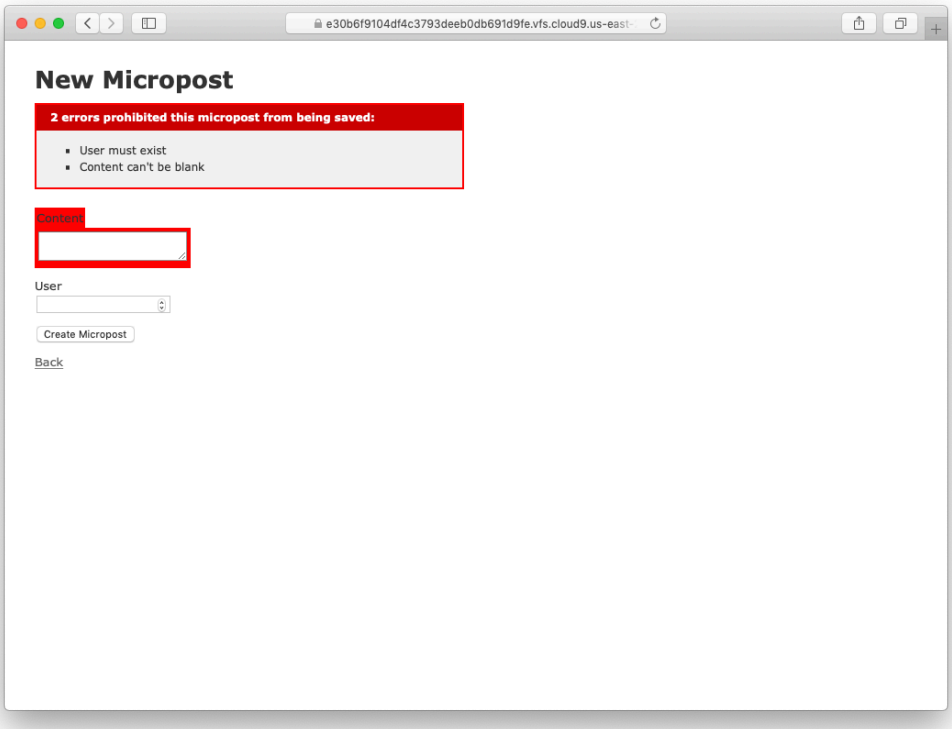


图 2.18：微博内容存在性验证的效果

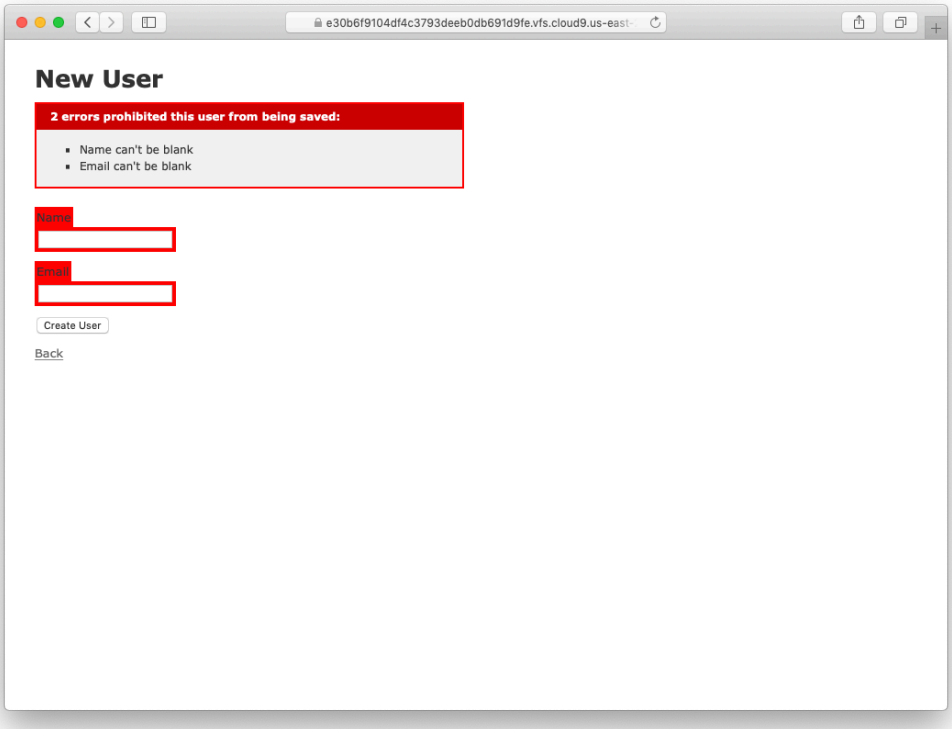


图 2.19：User 模型存在性验证的效果

2.3.4 继承体系

接下来简要介绍 Rails 中控制器和模型的类继承。如果你有面向对象编程（Object-oriented Programming，OOP）的经验，尤其是类（class），能更好地理解这些内容。如果暂时不理解，也没关系，4.4 节会详细说明这些概念。

先看模型的继承体系。对比一下代码清单 2.20 和代码清单 2.21，可以看出，`User` 和 `Micropost` 都（通过 `<` 符号）继承自 `ApplicationRecord` 类，而这个类继承自 `ActiveRecord::Base` 类，这是 Active Record 为模型提供的基类。图 2.20 列出了这种继承关系。继承 `ActiveRecord::Base` 类，模型对象才能与数据库通讯，才能把数据库中的列看做 Ruby 中的属性，等等。

代码清单 2.20：`User` 类的继承关系
app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
end
```

代码清单 2.21：`Mcropost` 类的继承关系
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  .
  .
  .
end
```

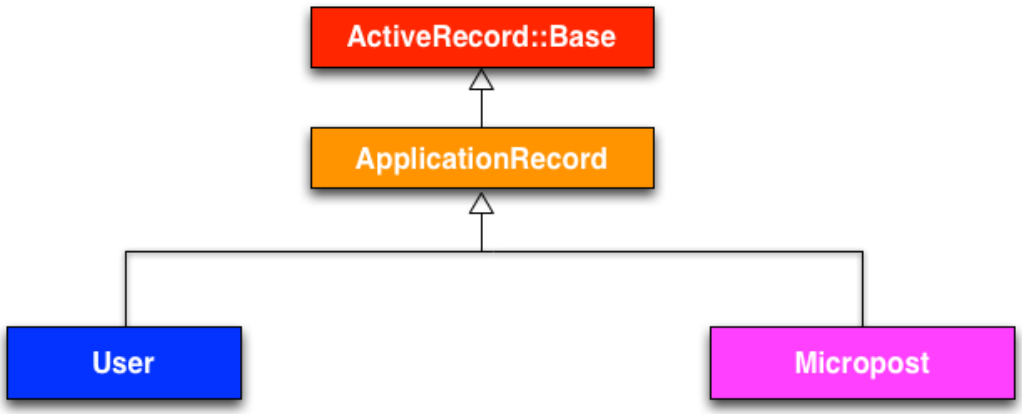


图 2.20：`User` 模型和 `Micropost` 模型的继承体系

控制器的继承体系与模型基本相同。对比代码清单 2.22 和代码清单 2.23，可以看出，`UsersController` 和 `MicropostsController` 都继承自 `ApplicationController`。如代码清单 2.24 所示，`ApplicationController` 继承自 `ActionController::Base`。`ActionController::Base` 是 Rails 中 Action Pack 库为控制器提供的基类。这些类之间的关系如图 2.21 所示。

代码清单 2.22：`UsersController` 类中的继承
app/controllers/users_controller.rb

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

代码清单 2.23：MicropostsController 类中的继承
app/controllers/microposts_controller.rb

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

代码清单 2.24：ApplicationController 类中的继承
app/controllers/application_controller.rb

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

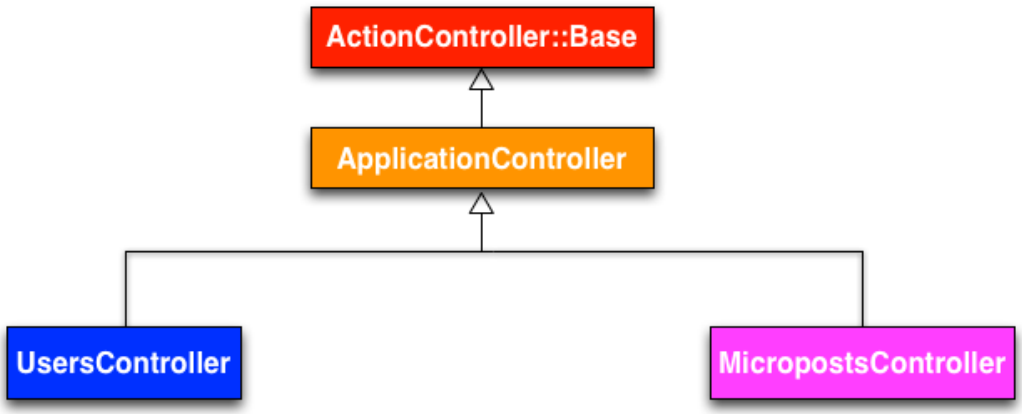


图 2.21：UsersController 和 MicropostsController 的继承体系

与模型的继承类似，通过继承 ActionController::Base，Users 控制器和 Microposts 控制器获得了很多功能。例如，处理模型对象、过滤入站 HTTP 请求，以及把视图渲染成 HTML 的能力。Rails 应用中的所有控制器都继承自 ApplicationController，所以其中定义的规则会自动运用于应用中的每个动作上。例如，9.1 节将介绍如何在 Application 控制器中引入辅助方法，为整个应用中的每个控制器都加上登录和退出功能。

练习

1. 查看 Application 控制器文件的内容，找出 ApplicationController 继承自 ActionController::Base 的代码。
2. ApplicationRecord 是不是也在类似的文件中继承 ActiveRecord::Base？提示：可能是 app/models 目录中名为 application_record.rb 的文件。

2.3.5 部署这个玩具应用

完成 Microposts 资源之后，是时候把仓库推送到 GitHub 中了：

```
$ git status # 添加内容之前查看状态是个好习惯
$ git add -A
$ git commit -m "Finish toy app"
$ git push
```

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

然后，还可以参照 1.4 节所述的步骤，把这个应用部署到 Heroku 中：

```
$ git push heroku
```

（执行这个命令之前要按照 2.1 节中的说明创建 Heroku 应用：先执行 `heroku create` 命令，然后再执行 `git push heroku master` 命令。）

此时访问 Heroku 中的应用会看到一个错误消息，如图 2.22 所示。

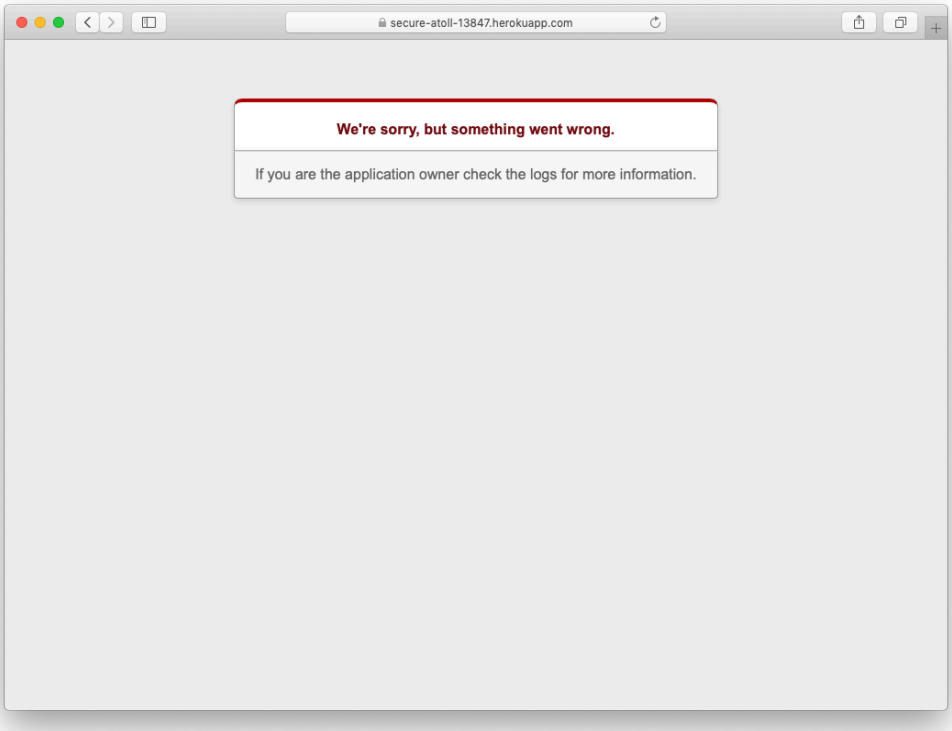


图 2.22：Heroku 显示错误页面

我们可以在 Heroku 日志中查找问题的根源：

```
$ heroku logs
```

在日志中应该能找到下面这行内容：

```
ActionView::Template::Error (PG::UndefinedTable: ERROR:  relation "users" does
not exist
```

这表明缺少 `users` 表。我们在代码清单 2.4 中学过如何解决这样的问题，即运行数据库迁移（同时还将创建 `microposts` 表）。

在 Heroku 中执行这样的命令，要在常规的 Rails 命令前面加上 `heroku run`，如下所示：

```
$ heroku run rails db:migrate
```

这个命令会按照 `User` 和 `Micropost` 数据模型更新 Heroku 中的数据库。迁移数据库之后，就可以在生产环境中使用这个应用了，如图 2.23 所示，而且这个应用使用 PostgreSQL 数据库。[9]

最后，如果你做了 2.3.3 节的练习，要把显示第一个用户微博的代码去掉，这样应用才能正确加载。你只需把那些代码删掉，做次提交，然后再推送到 Heroku。

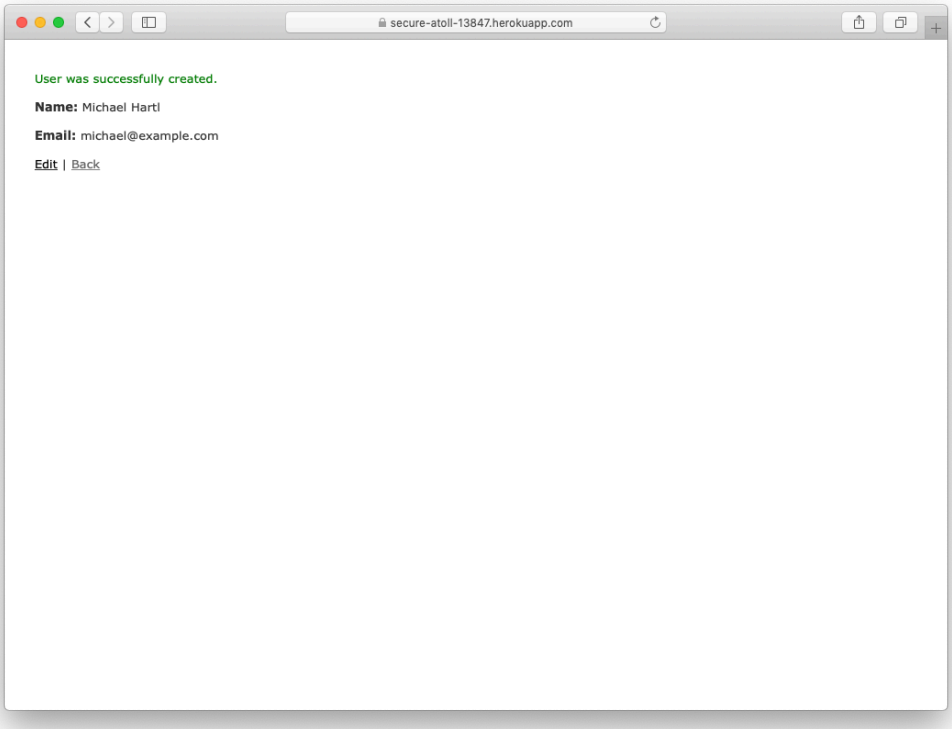


图 2.23：运行在生产环境中的玩具应用

练习

1. 在线上应用中创建几个用户。
2. 为第一个用户创建几篇微博。
3. 发布微博时填写超过 140 个字，确认代码清单 2.14 中的验证在生产环境中可用。

9. 生产数据库不做任何配置应该也能正常使用，不过根据 Heroku 官方文档的建议（https://devcenter.heroku.com/articles/getting-started-with-rails5），最好配置一下。详情参见 7.5.3 节。

2.4 小结

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

优点

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

缺点

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

本书后续的内容建立在这些优点之上，而且会改善缺点。

2.4.1 本章所学

- 使用脚手架自动生成模型的代码，然后通过 Web 界面与应用交互；
- 脚手架利于快速上手，但生成的代码不易理解；
- Rails 使用“模型-视图-控制器”（MVC）模式组织 Web 应用；
- 借由 Rails 我们得知，为了与数据模型交互，REST 架构制定了一套标准的 URL 和控制器动作；
- Rails 支持数据验证，用于约束数据模型的属性可以使用什么值；
- Rails 内建支持建立数据模型关联的功能；
- 可以使用 Rails 控制台在命令行中与 Rails 应用交互。

第 3 章 基本静态的页面

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

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

3.1 创建演示应用

与第 2 章一样，我们将先创建一个新 Rails 项目，名为 `sample_app`，如代码清单 3.1 所示。[1]

代码清单 3.1：创建一个新演示应用

```
$ cd ~/environment
$ rails _6.1.3_ new sample_app
$ cd sample_app/
```

（与 2.1 节一样，如果使用云 IDE，可以在同一个环境中创建这个应用，没必要再新建一个环境。）

注意：为了便于参考，本书实现的完整演示应用可以在 GitHub 中查看（https://github.com/mhartl/sample_app_6th_ed）。

与 2.1 节一样，接下来我们要用文本编辑器打开并编辑 `Gemfile` 文件，写入应用所需的 gem。代码清单 3.2 与代码清单 1.8 差不多，只是 `test` 组中的 gem 有所不同，这些是高级测试设置（3.6 节）和集成测试（5.3.4 节）所需的。注意，如果现在你想安装这个应用使用的全部 gem，请写入代码清单 13.76 中的内容。

代码清单 3.2：这个演示应用的 `Gemfile` 文件

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.1.3'
gem 'puma',       '5.2.2'
gem 'sass-rails', '6.0.0'
gem 'webpacker',  '5.2.1'
gem 'turbolinks', '5.2.1'
```

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

```
gem 'jbuilder',   '2.10.0'
gem 'bootsnap',   '1.7.2', require: false

group :development, :test do
  gem 'sqlite3', '1.4.2'
  gem 'byebug',  '11.1.3', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.1.0'
  gem 'listen',                '3.2.1'
  gem 'spring',                '2.1.1'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',                '3.35.3'
  gem 'selenium-webdriver',      '3.142.7'
  gem 'webdrivers',              '4.6.0'
  gem 'rails-controller-testing', '1.0.5'
  gem 'minitest',                '5.11.3'
  gem 'minitest-reporters',      '1.3.8'
  gem 'guard',                   '2.16.2'
  gem 'guard-minitest',          '2.4.6'
end

group :production do
  gem 'pg', '1.2.3'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
# Uncomment the following line if you're running Rails
# on a native Windows system:
# gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

与前两章一样，我们要执行 `bundle install` 命令安装并引入 `Gemfile` 文件中指定的 gem，而且设定 `bundle config set --local without 'production'`，[2] 不安装生产环境使用的 gem：

```
$ bundle _2.2.13_ config set --local without 'production'
$ bundle _2.2.13_ install
```

执行上述命令后不会在开发环境中安装 PostgreSQL 所需的 `pg` gem，在生产环境和测试环境中我们使用 SQLite。Heroku 不建议在开发环境和生产环境中使用不同的数据库，但是对这个演示应用来说，这两种数据库没什么差别，而且在本地安装、配置 SQLite 比 PostgreSQL 容易得多。[3] 1.2.1 节说过，如果你在 Windows 系统中运行 Rails，别忘了把代码清单 3.2 最后一行的注释去掉。

如果你之前安装了某个 gem（例如 Rails）的其他版本，与 `Gemfile` 中指定的版本号不同，最好再执行 `bundle`

2. 注意，这个设置会被 Bundler 记住，下次只需运行 `bundle install` 即可。

3. 一般来说，开发环境和生产环境要尽量一致，例如使用相同的数据库。但在本书中，本地一直使用 SQLite，生产环境则使用 PostgreSQL。现在时机还不成熟，笔者建议你以后一定要学会如何在开发环境中安装、配置 PostgreSQL。届时，可以在 Google 中搜索“install configure postgresql <your system>”，以及“rails postgresql setup”。在云 IDE 中，“<your system>”是 Linux。

update 命令更新 gem，确保安装的版本和指定的一致：

```
$ bundle _2.2.13_ update
```

最后，初始化 Git 仓库：

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

与第一个应用一样，建议你更新一下 README 文件，更好地描述这个应用。我们先把这个文件中的内容删掉，然后换成代码清单 3.3 中的 Markdown 内容。注意，README 文件中说明了如何安装这个应用。（直到第 6 章才会执行 rails db:migrate 命令，不过现在写上也无妨。）

代码清单 3.3：修改演示应用的 README 文件

README.md

````
# Ruby on Rails Tutorial sample application

This is the sample application for
[*Ruby on Rails Tutorial:
Learn Web Development with Rails*](https://www.railstutorial.org/)
(6th Edition)
by [Michael Hartl](https://www.michaelhartl.com/).

## License

All source code in the [Ruby on Rails Tutorial](https://www.railstutorial.org/)
is available jointly under the MIT License and the Beerware License. See
[LICENSE.md](LICENSE.md) for details.

## Getting started

To get started with the app, clone the repo and then install the needed gems:

```
$ gem install bundler -v 2.2.13
$ bundle _2.2.13_ config set --local without 'production'
$ bundle _2.2.13_ install
```

Next, migrate the database:

```
$ rails db:migrate
```

Finally, run the test suite to verify that everything is working correctly:

```
$ rails test
```

If the test suite passes, you'll be ready to run the app in a local server:
````

```
$ rails server
```

For more information, see the
[*Ruby on Rails Tutorial* book](https://www.railstutorial.org/book).

然后，提交改动：

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

你可能还记得，在 1.3.4 节，我们使用 `git commit -a -m "Message"` 命令，指定了表示“全部变化”的标志 `-a` 和提交信息的标志 `-m`。如上述命令所示，我们可以把二者合在一起，写成 `git commit -am "Message"`。

你还可以按照 1.3.3 节中的步骤，在 GitHub 中创建一个新仓库（图 3.1，别忘了设为私有仓库），然后把代码推送到这个远程仓库中：

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

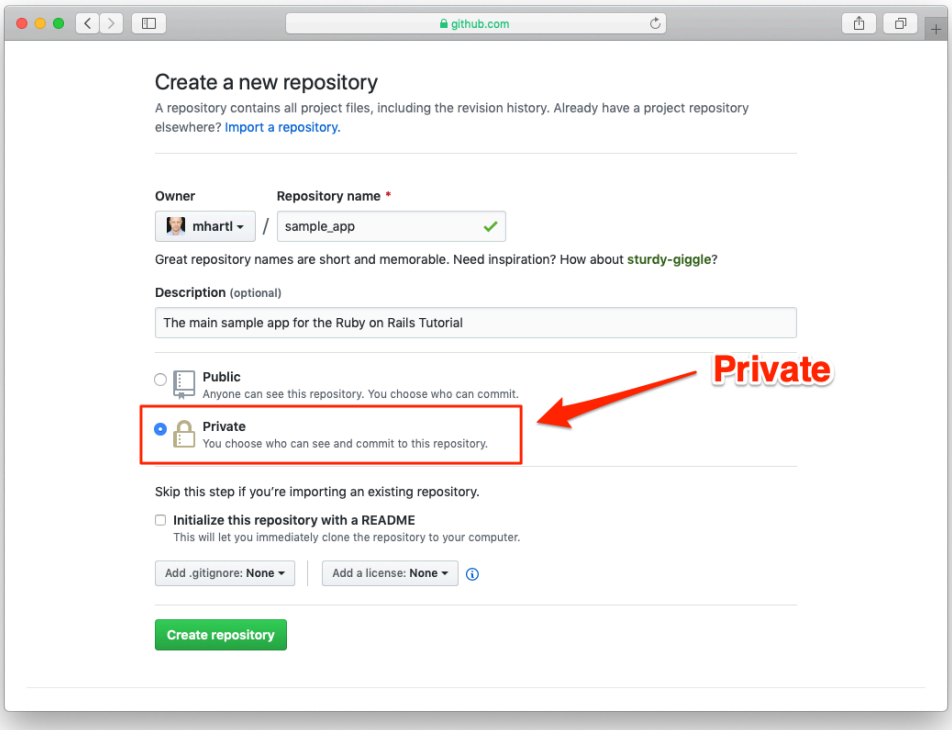


图 3.1：在 GitHub 中为这个演示应用创建一个仓库

如果你使用云 IDE，还要像前两章那样，编辑 `development.rb` 文件（代码清单 3.4），为连接本地服务器做好准备。

代码清单 3.4：允许连接本地 Web 服务器
config/environments/development.rb

```
Rails.application.configure do
  .
  .
  .
  # 允许连接本地服务器
  config.hosts.clear
end
```

为了避免以后遇到焦头烂额的问题，在这个早期阶段也可以把应用部署到 Heroku 中。参照第 1 章和第 2 章，笔者建议像代码清单 3.5 和代码清单 3.6 那样做，创建一个显示“hello, world!”的首页。（之所以这样做是因为 Rails 的默认页面往往无法在 Heroku 中显示，很难判断部署成功还是失败。）

代码清单 3.5：在 Application 控制器中添加 hello 动作
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base

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

代码清单 3.6：设置根路由
config/routes.rb

```
Rails.application.routes.draw do
  root 'application#hello'
end
```

然后提交改动，推送到 GitHub 和 Heroku 中：

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

与 1.4 节一样，你可能会看到一些警告消息，现在暂且不管，7.5 节会解决。除了 Heroku 为应用分配的地址之外，看到的页面应该与图 1.31 一样。

注意，有些读者反馈说遇到了与 `spring` gem 有关的错误。请在命令行中执行 `spring binstub` 命令，看能不能解决这个问题。

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

```
$ heroku logs          # 查看最近的事件
$ heroku logs --tail   # 实时查看事件，按 Ctrl-C 组合键退出
```

注意，如果你决定把真实的应用部署到 Heroku 中，一定要按照 7.5 节介绍的方法配置生产环境的 Web 服务器。

练习

1. 确认 GitHub 把代码清单 3.3 中的 Markdown 渲染成了 HTML（图 3.2）。
2. 访问生产环境中应用的根路由，确认成功部署到 Heroku 中了。

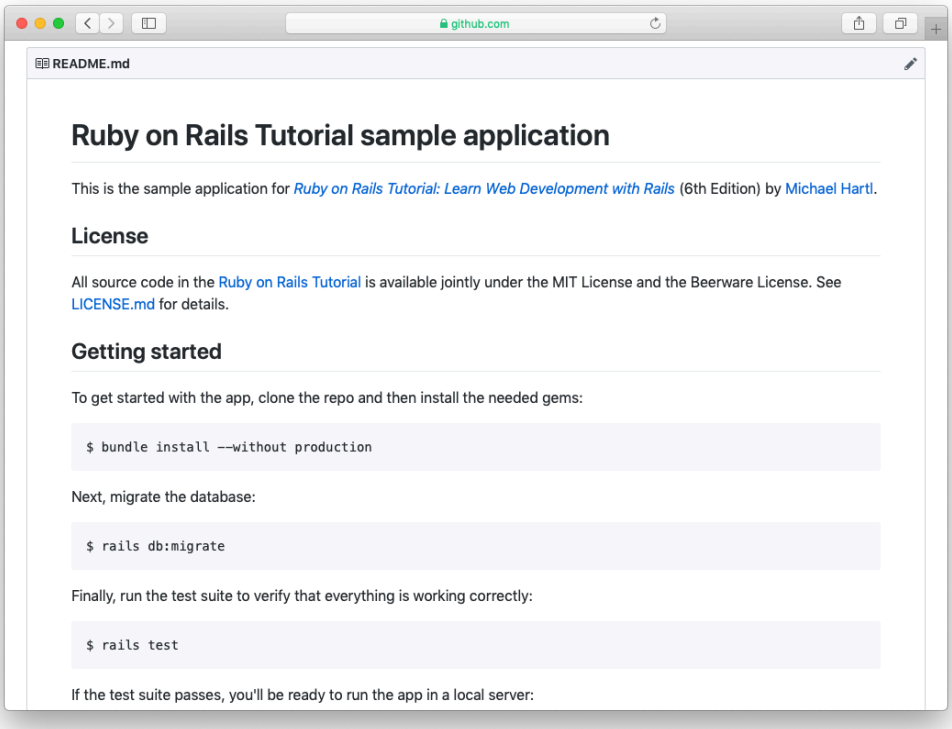


图 3.2：GitHub 中显示演示应用的 README 文件

3.2 静态页面

前一节的准备工作做好之后，我们可以开始开发这个演示应用了。本节，我们要向开发动态页面迈出第一步：创建一些 Rails 动作和视图，但只包含静态 HTML。[4] Rails 动作放在控制器中（MVC 中的 C，参见 1.2.3 节），用于组织相关的功能。第 2 章已经简要介绍了控制器，全面熟悉 REST 架构之后（从第 6 章开始），你会更深入地理解控制器。回想一下 1.2 节介绍的 Rails 项目目录结构（图 1.11），这对我们有所帮助。本节主要在 `app/controllers` 和 `app/views` 两个目录中工作。

1.3.4 节说过，使用 Git 时最好在单独的主题分支中完成工作，不要直接使用主分支。如果你使用 Git 做版本控制，现在应该执行下述命令，切换到一个主题分支，然后再创建静态页面：

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

3.2.1 生成静态页面

下面我们要使用第 2 章用来生成脚手架的 `generate` 命令生成一个控制器。既然这个控制器用来处理静态页面，那就把它命名为 `StaticPages` 吧。可以看出，控制器的名称使用驼峰式命名法（https://en.wikipedia.org/wiki/CamelCase）。我们计划创建“首页”、“帮助”页面和“关于”页面，对应的动作名分别为 `home`、`help` 和 `about`。`generate` 命令可以接收一个可选的参数列表，指定要创建的动作。我们将在命令行中指定 `home` 和 `help` 动作，故意不指定 `about` 动作，3.3 节再介绍怎么添加。生成 `StaticPages` 控制器的命令如代码清单 3.7

4. 这里讲的静态页面创建方法可能是最简单的，但不是唯一的，你应该根据需求使用合适的方法。如果要创建大量静态页面，使用静态页面控制器太麻烦，不过这个演示应用只需要几个静态页面。如果需要创建大量静态页面，可以使用 thoughtbot 开发的 `high_voltage` gem。

所示。

代码清单 3.7：生成 StaticPages 控制器

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

顺便说一下，`rails generate` 可以简写成 `rails g`。除此之外，Rails 还提供了几个命令的简写形式，见表 3.1。为了表述明确，本书会一直使用命令的完整形式，但在实际使用中，多数 Rails 开发者或多或少都会使用表 3.1 中的简写形式。[5]

表 3.1：一些 Rails 命令的简写形式

完整形式	简写形式
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ rails test`	`$ rails t`
`$ bundle install`	`$ bundle`

在继续之前，如果你使用 Git，最好把 `StaticPages` 控制器相关的文件推送到远程仓库：

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

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

```
$ git push
```

在现实的开发过程中，笔者一般都会先提交再推送，但是为了行文简洁，从这往后我们会省略提交这一步。（在阅读本书的过程中，建议每一节结束后提交一次。）

注意，在代码清单 3.7 中，我们传入的控制器名使用驼峰式（因为像骆驼的双峰一样），创建的控制器文件

5. 其实，很多 Rails 开发者还会为 `rails` 命令创建别名，简化成 `r`。这样，使用简洁的 `r s` 命令就能启动 Rails 服务器。

名则是蛇底式（https://en.wikipedia.org/wiki/Snake_case）。所以，传入“StaticPages”得到的文件是 `static_pages_controller.rb`。这只是一种约定，其实在命令行中也可以使用蛇底式：

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

这个命令也会生成名为 `static_pages_controller.rb` 的控制器文件。因为 Ruby 的类名使用驼峰式（4.4 节），所以提到控制器时笔者会使用驼峰式，不过这只是个人喜好。（Ruby 文件名一般使用蛇底式，Rails 生成器会使用 `underscore` 方法把驼峰式转换成蛇底式。）

顺便说一下，如果在生成代码时出错了，知道如何撤销操作就很有用了。旁注 3.1 中介绍了一些在 Rails 中撤销操作的方法。

旁注 3.1：撤销操作

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

举例来说，一个常见的情况是更改控制器的名称，这时你得删除生成的所有文件。生成控制器时，除了控制器文件本身之外，Rails 还会生成很多其他文件（见代码清单 3.7）。撤销生成的文件时不仅仅要删除控制器文件，还要删除不少辅助文件。（在 2.2 节和 2.3 节中我们看到，`rails generate` 命令还会自动修改 `routes.rb` 文件，因此我们也想自动撤销这些改动。）在 Rails 中，可以使用 `rails destroy` 命令完成撤销操作。一般来说，下面这两个命令是相互抵消的：

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

第 6 章将使用下面的命令生成模型：

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

这个操作可以使用下面的命令撤销：

```
$ rails destroy model User
```

（这里，我们可以省略命令行中其余的参数。读到第 6 章时，看看你能否发现为什么可以这样做。）

对模型来说，还涉及到撤销迁移。第 2 章已经简要介绍了迁移，第 6 章开始会深入说明。迁移通过下面的命令改变数据库的状态：

```
$ rails db:migrate
```

我们可以使用下面的命令撤销前一个迁移操作：

```
$ rails db:rollback
```

如果要回到最开始的状态，可以使用：

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

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

知道这些技术，我们就能得心应对开发过程中遇到的各种问题了。

代码清单 3.7 中生成 `StaticPages` 控制器的命令会自动修改路由文件（`config/routes.rb`）。我们在代码清单

1.14 和代码清单 3.6 中修改过根路由。路由文件的作用是实现 URL 和网页之间的对应关系（图 2.12）。路由文件在 `config` 目录中。Rails 在这个目录中存放应用的配置文件（图 3.3）。

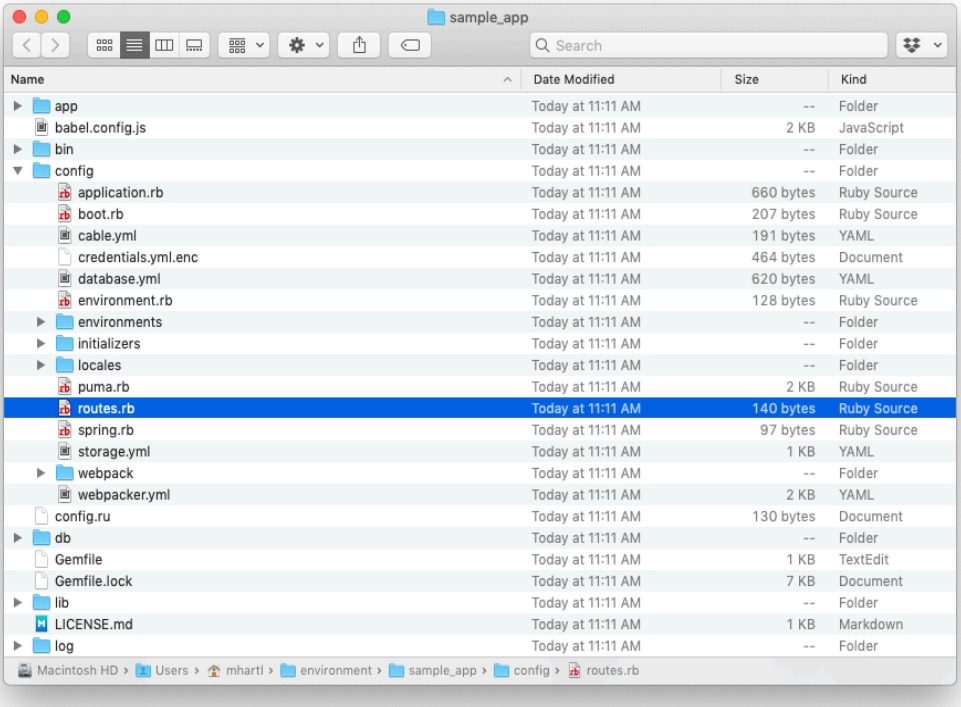


图 3.3：演示应用 `config` 目录中的内容

因为生成控制器时我们指定了 `home` 和 `help` 动作，所以路由文件中已经添加了相应的规则，如代码清单 3.8 所示。

代码清单 3.8：`StaticPages` 控制器中 `home` 和 `help` 动作的路由
`config/routes.rb`

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

如下的规则

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

把发给 /static_pages/home 的请求映射到 `StaticPages` 控制器的 `home` 动作上。另外，`get` 表明这个路由响应的是 GET 请求。GET 是 HTTP（Hypertext Transfer Protocol，超文本传输协议）支持的基本请求方法之一（旁注 3.2）。这里，当我们在 `StaticPages` 控制器中生成 `home` 动作时，就自动在 /static_pages/home 地址上获得了一个页面。若想查看这个页面，按照 1.2.2 节中的方法，启动 Rails 开发服务器：

```
$ rails server
```

然后访问 /static_pages/home，如图 3.4 所示。

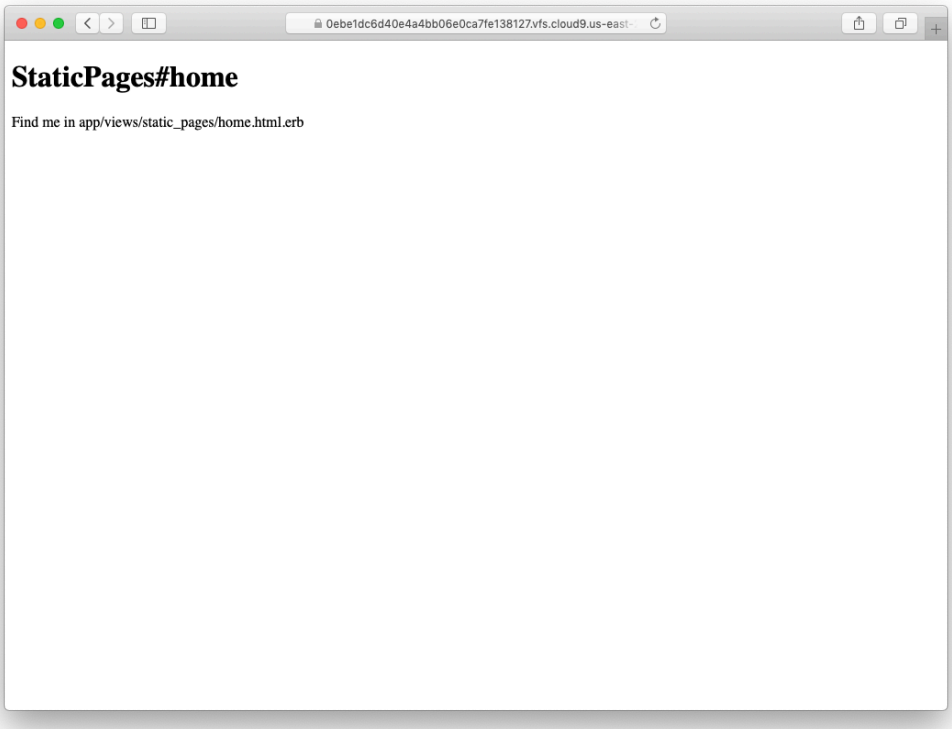


图 3.4：简陋的首页（/static_pages/home）

旁注 3.2：GET 等请求方法

超文本传输协议（HTTP）定义了几个基本操作，GET、POST、PATCH 和 DELETE。这四个动词表示客户端电脑（通常安装了一种浏览器，例如 Chrome、Firefox 或 Safari）与服务器（通常会运行一个 Web 服务器，例如 Apache 或 Nginx）之间的操作。（有一点很重要，你要知道：在本地电脑中开发 Rails 应用时，客户端和服务器在同一台物理设备中，但这二者是不同的概念。）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 章开始会更详细地说明。

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

要想弄明白这个页面是怎么来的，我们先在文本编辑器中看一下 StaticPages 控制器文件。你应该会看到类似代码清单 3.9 所示的内容。你可能注意到了，与第 2 章中的 Users 和 Microposts 控制器不同，StaticPages 控制器没使用标准的 REST 动作。这对静态页面来说是很常见的，毕竟 REST 架构不能解决所有问题。

代码清单 3.9：代码清单 3.7 生成的 StaticPages 控制器
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController
  def home
  end

  def help
  end
end
```

从上述代码中的 class 关键字可以看出，static_pages_controller.rb 文件中定义了一个类，名为 StaticPagesController。类是一种组织函数（也叫方法）的便利方式，例如 home 和 help 动作就是方法，使用 def 关键字定义。2.3.4 节说过，尖括号 < 表示 StaticPagesController 继承自 ApplicationController 类；稍后你将看到，这意味着我们定义的页面拥有了 Rails 提供的大量功能。（我们会在 4.4 节更详细地介绍类和继承。）

现在，StaticPages 控制器中的两个方法都是空的：

```
def home
end

def help
end
```

如果是普通的 Ruby 代码，这两个方法什么也做不了。不过在 Rails 中就不一样了。StaticPagesController 是一个 Ruby 类，但是由于它继承自 ApplicationController，其中的方法对 Rails 来说就有了特殊意义：访问 /static_pages/home 时，Rails 会在 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（MVC 中的 V，参见 1.2.3 节）。这里，home 动作是空的，所以访问 /static_pages/home 后只会渲染视图。那么，视图是什么样子，怎么才能找到它呢？

如果你再看一下代码清单 3.7 的输出，或许能猜到动作和视图之间的对应关系：home 动作对应的视图是 home.html.erb。3.4 节会告诉你 .erb 是什么意思。看到 .html 你或许就不奇怪了，这个文件基本上就是 HTML，如代码清单 3.10 所示。

代码清单 3.10：为“首页”生成的视图
app/views/static_pages/home.html.erb

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

help 动作的视图类似，如代码清单 3.11 所示。

代码清单 3.11：为“帮助”页面生成的视图
app/views/static_pages/help.html.erb

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

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

练习

1. 生成含有 bar 和 baz 两个动作的 Foo 控制器。
2. 使用旁注 3.1 中介绍的技术删除 Foo 控制器及相关的动作。

3.2.2 修改静态页面的内容

我们将在 3.4 节添加一些简单的动态内容。现在，这些静态内容的存在是为了强调一件很重要的事：Rails 的视图可以只包含静态 HTML。所以我们甚至无需了解 Rails 就可以修改“首页”和“帮助”页面的内容，如代码清单 3.12 和代码清单 3.13 所示。

代码清单 3.12：修改“首页”的 HTML
app/views/static_pages/home.html.erb

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

代码清单 3.13：修改“帮助”页面的 HTML
app/views/static_pages/help.html.erb

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

修改之后，这两个页面显示的内容如图 3.5 和图 3.6 所示。

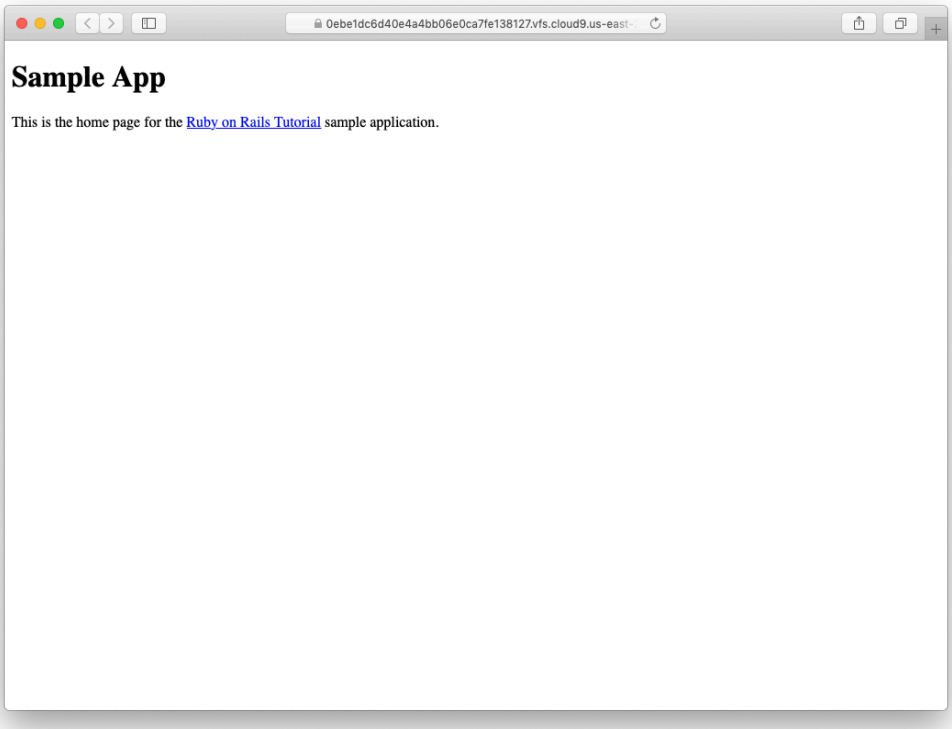


图 3.5：修改后的“首页”

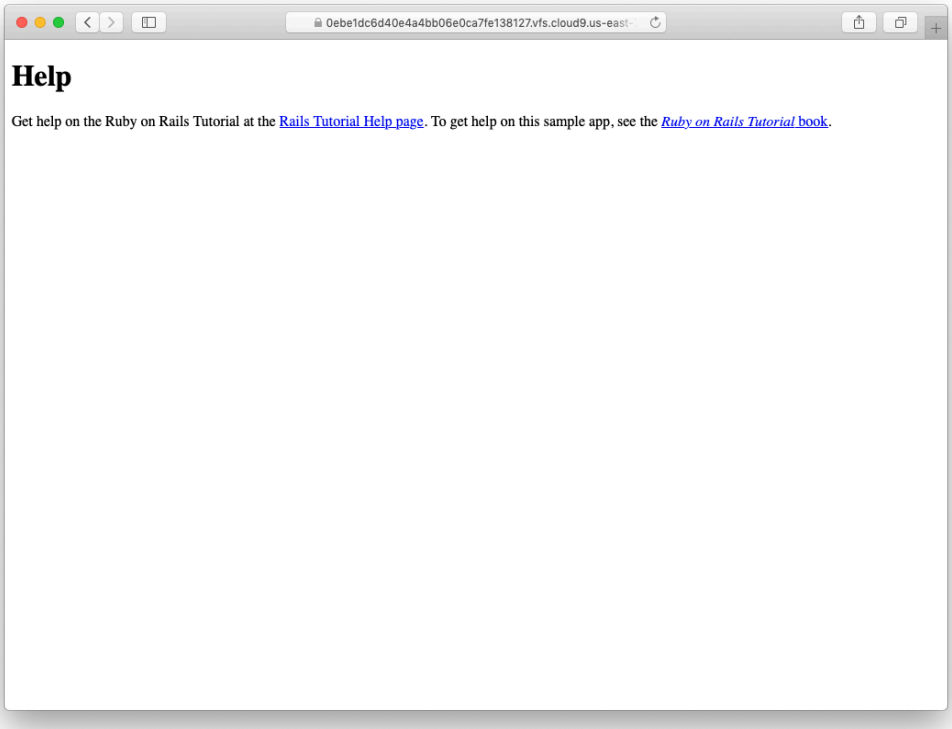


图 3.6：修改后的“帮助”页面

3.3 开始测试

我们创建并修改了“首页”和“帮助”页面的内容，下面要添加“关于”页面。做这样的改动时，最好编写自动化测试，确认实现的方法是否正确。对本书开发的应用来说，我们编写的测试组件有两个作用：其一，作为一种安全防护措施；其二，作为源码的文档。虽然要编写额外的代码，但是如果方法得当，测试能协助我们快速开发，因为有了测试之后，查找问题所用的时间会变少。不过，我们要善于编写测试才行，所以要尽早开始练习。

几乎每个 Rails 开发者都认同测试是好习惯，但具体的作法多种多样。最近有一场针对测试驱动开发（Test-Driven Development，TDD）的争论[6]，十分热闹。TDD 是一种测试技术，程序员要先编写失败的测试，然后再编写应用代码，让测试通过。本书采用一种轻量级、符合直觉的测试方案，只在适当的时候才使用 TDD，而不严格遵守 TDD 理念（旁注 3.3）。

旁注 3.3：何时测试

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

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

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

现在，最好有一些指导方针，告诉你什么时候应该先写测试（以及什么时候完全不用测试）。根据笔者的经验，下面给出一些建议：

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

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

我们主要编写的测试类型是控制器测试（本节开始编写）、模型测试（第 6 章开始编写）和集成测试（第 7 章开始编写）。集成测试的作用特别大，它能模拟用户在浏览器中与应用交互的过程，最终会成为我们的主

6. 详情参见 Rails 创始人 David Heinemeier Hansson 写的一篇文章：TDD is dead. Long live testing（https://dhh.dk/2014/tdd-is-dead-long-live-testing.html）。

要关注对象，不过控制器测试更容易上手。

3.3.1 第一个测试

现在我们要在这个应用中添加一个“关于”页面。我们将看到，这个测试很短，所以按照旁注 3.3 中的指导方针，我们先编写测试，然后使用失败的测试驱动我们编写应用代码。

着手测试是件具有挑战的事情，要求对 Rails 和 Ruby 都有深入的了解。这么早就编写测试可能让你有点儿害怕。不过，Rails 已经为我们解决了最难的部分，因为执行 `rails generate controller` 命令时（代码清单 3.7）自动生成了一个测试文件，我们可以从这个文件入手：

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

我们来看一下这个文件中的内容，如代码清单 3.14 所示。

代码清单 3.14：默认为 StaticPages 控制器生成的测试 GREEN
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

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

  test "should get help" do
    get static_pages_help_url
    assert_response :success
  end
end
```

现在无需理解详细的句法，不过可以看出，其中有两个测试，对应我们在命令行中传入的两个动作（代码清单 3.7）。各个测试先访问 URL，然后（通过断言）确认得到的是成功响应。其中，`get` 表示测试期望这两个页面是普通的网页，可以通过 `GET` 请求访问（旁注 3.2）；`:success` 响应（表示 200 OK）是对 HTTP 状态码的抽象表示。也就是说，下面这个测试的意思是：为了测试首页，向 `StaticPages` 控制器中 `home` 动作对应的 URL 发起 `GET` 请求，确认得到的是表示成功的状态码。

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

测试循环的第一步是运行测试组件，确认测试现在可以通过。我们要执行下述命令：

代码清单 3.15：GREEN

```
$ rails db:migrate     # 在某些系统中要先执行这个命令
$ rails test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

与预期一样，一开始测试组件可以通过（GREEN）。（在某些系统中默认不会显示绿色，除非按照 3.6.1 节的说明添加 MiniTest 报告程序。不过，即使看不到真正的绿色，我们也经常这样表述。）注意，测试的输出

往往很多，有时笔者会省略几行，只给出重要的部分。

顺便说一下，在某些系统中，db 目录中可能会出现类似下面这种生成的文件：

```
/db/test.sqlite3-0
```

为了避免把这样的文件添加到仓库中，建议在 .gitignore 文件中添加一条规则，忽略这类文件，如代码清单 3.16 所示。

代码清单 3.16：忽略生成的数据库文件
.gitignore

```
.
.
.
# Ignore db test files.
db/test.*
```

在某些系统中，测试要花相当长的时间才能启动，这是因为（1）要启动 Spring 服务器，预载部分 Rails 环境，不过只有首次启动时受此影响；（2）启动 Ruby 要花点儿时间。（第二点可以使用 3.6.2 节推荐的 Guard 改善。）

3.3.2 遇红

我们在旁注 3.3 中说过，测试驱动开发流程是先编写一个失败测试，然后编写应用代码让测试通过，最后再根据需要重构代码。因为很多测试工具都使用红色表示失败的测试，使用绿色表示通过的测试，所以这个流程有时也叫“遇红-变绿-重构”循环。这一节我们先完成这个循环的第一步，编写一个失败测试，即“遇红”。3.3.3 节“变绿”，3.4.3 节“重构”。[7]

首先，我们要为“关于”页面编写一个失败测试。参照代码清单 3.14，你能猜到该怎么写吗？答案在代码清单 3.17 中揭晓。

代码清单 3.17：“关于”页面的测试 RED
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

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

  test "should get help" do
    get static_pages_help_url
    assert_response :success
  end

  test "should get about" do
```

7. 在某些系统中，执行 rails test 命令后，如果测试失败会显示红色，但测试通过不显示绿色。若想得到由红变绿的过程，参照 3.6.1 节的说明。

```
    get static_pages_about_url
    assert_response :success
  end
end
```

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

与预期一样，这个测试现在失败：

代码清单 3.18：RED

```
$ rails test
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
```

3.3.3 变绿

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

先看一下这个失败测试给出的错误消息：

代码清单 3.19：RED

```
$ rails test
NameError: undefined local variable or method `static_pages_about_url'
```

这个错误消息指出，未定义获取“关于”页面地址的 Rails 代码，其实就是提示我们要在路由文件中添加一个规则。参照代码清单 3.8，我们可以编写如代码清单 3.20 所示的路由。

代码清单 3.20：添加 about 路由 RED
config/routes.rb

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

这段代码中高亮显示的那一行告诉 Rails，把发给 /static_pages/about 页面的 GET 请求交给 StaticPages 控制器的 about 动作处理。这条规则会自动创建一个辅助方法：

```
static_pages_about_url
```

再次运行测试组件，仍然无法通过，不过错误消息变了：

代码清单 3.21：RED

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

这个错误消息的意思是，StaticPages 控制器缺少 about 动作。我们可以参照代码清单 3.9 中的 home 和 help 编写这个动作，如代码清单 3.22 所示。

代码清单 3.22：在 StaticPages 控制器中添加 about 动作 RED
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

现在测试依旧失败，不过错误消息又变了：

```
$ rails test
ActionController::UnknownFormat: StaticPagesController#about is missing
a template for this request format and variant.
```

这表明缺少模板。在 Rails 中，模板其实就是视图。3.2.1 节说过，home 动作对应的视图是 home.html.erb，保存在 app/views/static_pages 目录中。所以，我们要在这个目录中新建一个文件，而且要命名为 about.html.erb。

在不同的系统中新建文件有不同的方法，不过多数情况下都可以在想要新建文件的目录中点击鼠标右键，然后在弹出的菜单中选择“新建文件”。我们也可以使用文本编辑器的“文件”菜单，新建文件后再选择保存的位置。除此之外，还可以使用笔者最喜欢的 Unix touch 命令，如下所示：

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

touch 命令的作用是更新文件或文件夹的修改时间戳，但有个副作用：如果文件不存在，它会新建一个空文件。（在云 IDE 中或许要刷新文件树，参见 1.2.1 节。这也体现了“技术是复杂的”。）

在正确的目录中创建 about.html.erb 文件之后，写入代码清单 3.23 中的内容。

代码清单 3.23：“关于”页面的内容 GREEN
app/views/static_pages/about.html.erb

```
<h1>About</h1>
<p>
  The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, part of the
  <a href="https://www.learnenough.com/">Learn Enough</a> family of
  tutorials, is a
  <a href="https://www.railstutorial.org/book">book</a> and
  <a href="https://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="https://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample app for the tutorial.
</p>
```

现在执行 rails test 命令，会看到测试通过了：

代码清单 3.24：GREEN

```
$ rails test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

当然，我们还可以在浏览器中查看这个页面（图 3.7），以防测试欺骗我们。

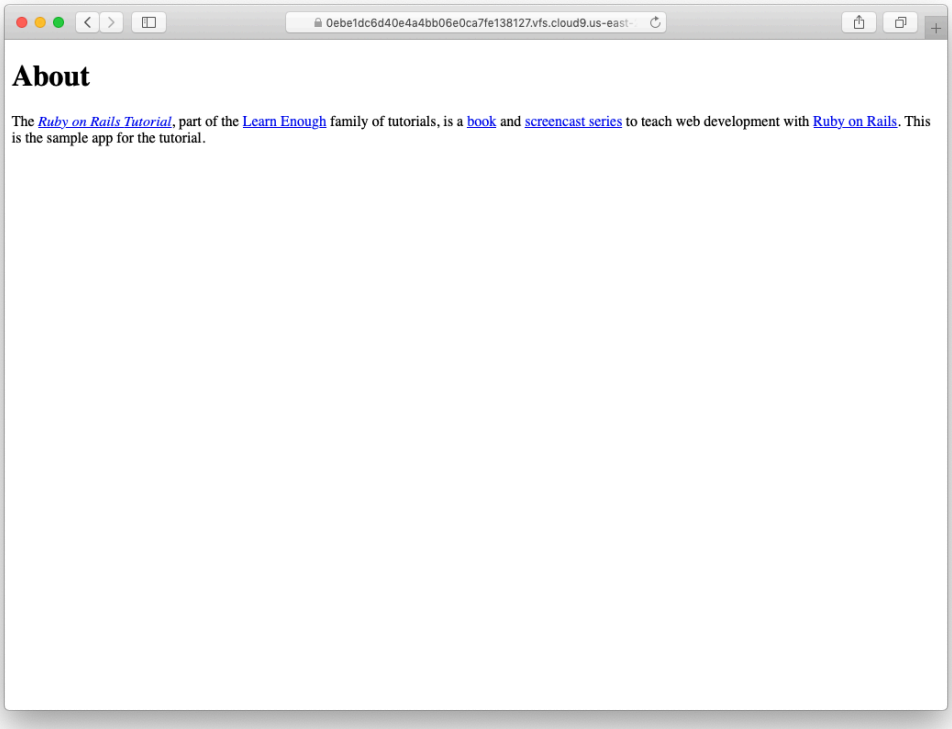


图 3.7：新添加的“关于”页面（/static_pages/about）

3.3.4 重构

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

3.4 有点动态内容的页面

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

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

`rails new` 命令创建了一个布局文件，不过现在最好不用。我们重命名这个文件：

```
$ mv app/views/layouts/application.html.erb layout_file   # 临时改动
```

在真实的应用中你不需要这么做，不过没有这个文件能让你更好地理解它的作用。

表 3.2：演示应用中基本上是静态内容的页面

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

3.4.1 测试标题（遇红）

添加标题之前，我们要知道网页的一般结构，如代码清单 3.25 所示。

代码清单 3.25：网页一般的 HTML 结构

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

这段代码的最顶部是文档类型声明（document type declaration，简称 doctype），作用是告诉浏览器使用哪个 HTML 版本（这里使用的是 HTML5）。[8] 随后是 `head` 部分，里面有一个 `title` 标签，其内容是“Greeting”。然后是 `body` 部分，里面有一个 `p` 标签（段落），其内容是“Hello, world!”。（缩进是可选的，HTML 不会特别对待空白，制表符和空格都会被忽略，但缩进可以让文档结构更清晰。）

我们要使用 `assert_select` 方法分别为表 3.2 中的每个标题编写简单的测试，然后合并到代码清单 3.17 中。`assert_select` 方法的作用是检查有没有指定的 HTML 标签。这种方法有时也叫选择符（selector），因此才为这个方法取这么一个名称。[9]

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

这行代码的作用是检查有没有 `<title>` 标签，以及其中的内容是不是字符串“Home | Ruby on Rails Tutorial Sample App”。把这样的代码分别放到三个页面的测试中，得到的结果如代码清单 3.26 所示。

代码清单 3.26：加入标题测试后的 `StaticPages` 控制器测试 RED

test/controllers/static_pages_controller_test.rb

```
require 'test_helper'
```

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

9. Rails 指南中说明测试的文章列出了常用的 MiniTest 断言（https://rails.guide/book/testing.html#available-assertions）。

```
class StaticPagesControllerTest < ActionDispatch::IntegrationTest

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

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

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

写好测试之后，应该确认一下现在测试组件是失败的（RED）：

代码清单 3.27：RED

```
$ rails test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

3.4.2 添加页面标题（变绿）

现在，我们要为每个页面添加标题，让前一节的测试通过。参照代码清单 3.25 中的 HTML 结构，把代码清单 3.12 中的首页内容换成代码清单 3.28 中的内容。

代码清单 3.28：具有完整 HTML 结构的首页 RED
app/views/static_pages/home.html.erb

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

修改之后的首页如图 3.8 所示。注意，截图中使用的浏览器（Safari）只在有多个标签页时才会显示页面的标题，因此图 3.8 中有两个标签页。

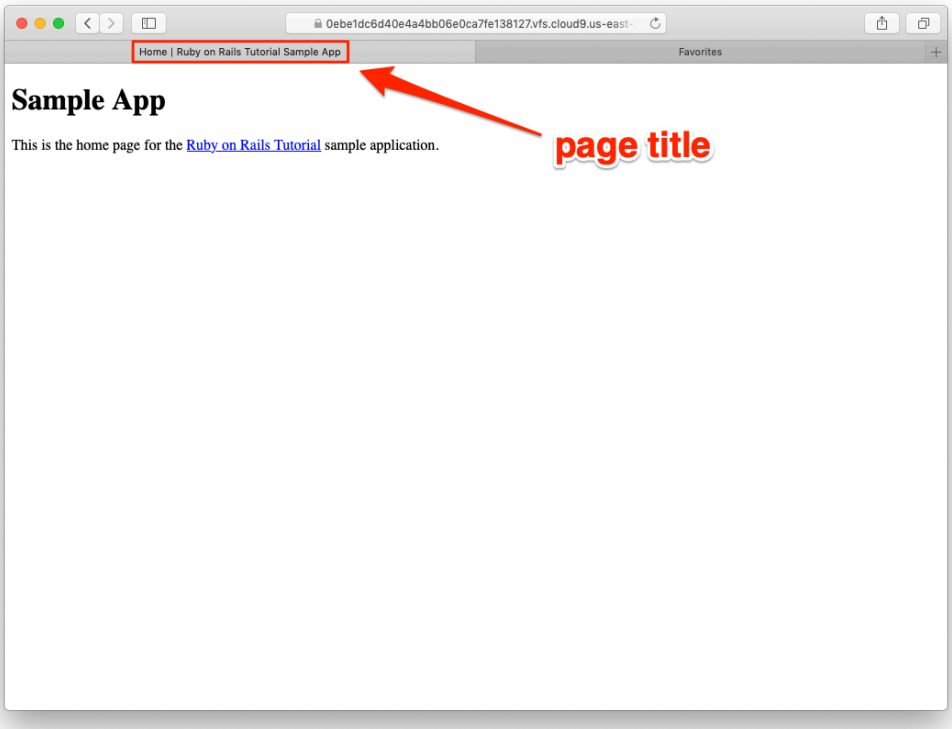


图 3.8：添加标题后的首页

使用类似的方式修改“帮助”页面和“关于”页面，得到的代码如代码清单 3.29 和代码清单 3.30 所示。

代码清单 3.29：具有完整 HTML 结构的“帮助”页面 RED
app/views/static_pages/help.html.erb

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

代码清单 3.30：具有完整 HTML 结构的“关于”页面 GREEN
app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <title>About | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a>, part of the
      <a href="https://www.learnenough.com/">Learn Enough</a> family of
      tutorials, is a
      <a href="https://www.railstutorial.org/book">book</a> and
      <a href="https://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="https://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample app for the tutorial.
    </p>
  </body>
</html>
```

现在，测试组件又能通过了：

代码清单 3.31：GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

练习

从本节开始，我们将修改应用代码，而且这些改动不会体现在以后的代码清单中。这么做是为了让没有做练习的读者能读懂正文，因为解答练习所需的代码与正文有差异。这也体现了“技术是复杂的”。

1. 你可能注意到了，`StaticPages` 控制器的测试（代码清单 3.26）中有些重复，每个标题测试中都有“Ruby on Rails Tutorial Sample App”。我们可以使用特殊的函数 `setup` 去除重复。这个函数在每个测试运行之前执行。请你确认代码清单 3.32 中的测试仍能通过。（代码清单 3.32 中使用了一个实例变量，2.2.2 节简单介绍过，4.4.5 节将进一步说明。这段代码还使用了字符串插值操作，4.2.1 节将做进一步说明。）

代码清单 3.32：使用一个基标题的 `StaticPages` 控制器测试 GREEN
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

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

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

```
  end

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

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

3.4.3 布局和嵌入式 Ruby（重构）

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

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

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

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

在这个过程中，我们要在视图中使用嵌入式 Ruby（Embedded Ruby）。既然首页、“帮助”页面和“关于”页面的标题中有一个变动的部分，那我们就使用 Rails 提供的一个特别的函数 `provide`，在每个页面中设定不同的标题。通过把 `home.html.erb` 视图中标题的“Home”换成代码清单 3.33 所示的代码，我们可以看一下这个函数的作用。

代码清单 3.33：标题中使用了嵌入式 Ruby 代码的首页视图 GREEN
app/views/static_pages/home.html.erb

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

```
</html>
```

在这段代码中我们第一次使用了嵌入式 Ruby（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。[10] 下面的代码

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

通过 `<% ... %>` 调用 Rails 提供的 `provide` 函数，把字符串 `"Home"` 赋给 `:title`。[11] 然后，在标题中，我们使用类似的符号 `<%= ... %>`，通过 Ruby 的 `yield` 函数把标题插入模板中：[12]

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

（这两种嵌入式 Ruby 代码的区别在于，`<% ... %>` 只**执行**其中的代码；`<%= ... %>` 除了执行其中的代码，还会把执行的结果**插入**模板中。）最终得到的页面跟以前一样，不过，现在标题中变动的部分是通过 ERb 动态生成的。

我们可以运行前一节编写的测试确认一下。现在，测试还能通过：

代码清单 3.34：GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

然后，按照相同的方式修改“帮助”页面（代码清单 3.35）和“关于”页面（代码清单 3.36）。

代码清单 3.35：标题中使用了嵌入式 Ruby 代码的“帮助”页面视图 GREEN
app/views/static_pages/help.html.erb

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

代码清单 3.36：标题中使用了嵌入式 Ruby 代码的“关于”页面视图 GREEN
app/views/static_pages/about.html.erb

10. 还有一种受欢迎的模板系统是 Haml（不是“HAML”，http://haml.info），笔者很喜欢用，不过在这样的初级教程中使用不太合适。

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

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

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

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

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

也就是说，所有页面的结构都是一样的，包括 `title` 标签中的内容，只有 `body` 标签中的内容有些差别。

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

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

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

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

修改后得到的布局文件如代码清单 3.37 所示。

代码清单 3.37：演示应用的网站布局 GREEN
app/views/layouts/application.html.erb

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

```
    <meta charset="utf-8">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

注意，其中有一行比较特殊：

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

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

另外，我们设置了字符集：指定使用 utf-8 编码显示 Unicode。

还要注意，默认的 Rails 布局文件中有下面几行代码：

```
<%= csrf_meta_tags %>
<%= csp_meta_tag %>
<%= stylesheet_link_tag ... %>
<%= javascript_pack_tag "application", ... %>
```

这几行代码的作用是，引入应用的样式表和 JavaScript 文件（Asset Pipeline 的一部分，见 5.2.1 节）；Rails 提供的 `csp_meta_tag` 方法实现内容安全策略（Content Security Policy，CSP），避免遭受跨站脚本（cross-site scripting，XSS）攻击；Rails 提供的 `csrf_meta_tags` 方法用于避免跨站请求伪造（Cross-Site Request Forgery，CSRF）攻击。（使用 Rails 这样成熟的框架有个好处，框架为我们分担了安全方面的担忧。）

虽然测试依然能通过，但是我们还有一件事要做：代码清单 3.33、代码清单 3.35 和代码清单 3.36 的内容还是和布局文件中的 HTML 结构类似，这样太冗余了（而且会导致 HTML 标记无效），我们要把完整的结构删除，只保留需要的内容。清理后的视图如代码清单 3.38、代码清单 3.39 和代码清单 3.40 所示。

代码清单 3.38：去除完整的 HTML 结构后的首页 GREEN
app/views/static_pages/home.html.erb

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

代码清单 3.39：去除完整的 HTML 结构后的“帮助”页面 GREEN
app/views/static_pages/help.html.erb

```
<% provide(:title, "Help") %>
```

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

代码清单 3.40：去除完整的 HTML 结构后的“关于”页面 GREEN
app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>
<h1>About</h1>
<p>
  The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, part of the
  <a href="https://www.learnenough.com/">Learn Enough</a> family of
  tutorials, is a
  <a href="https://www.railstutorial.org/book">book</a> and
  <a href="https://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="https://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample app for the tutorial.
</p>
```

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

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

代码清单 3.41：GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

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

练习

1. 为这个演示应用添加一个“联系”页面。参照代码清单 3.17，先编写一个测试，检查页面的标题是否为“Contact | Ruby on Rails Tutorial Sample App”，从而确定 /static_pages/contact 对应的页面是否存在。参照 3.3.3 节添加“关于”页面的步骤，把代码清单 3.42 中的内容写入“联系”页面的视图，让测试通过。（这道题会在 5.3.1 节完成。）

代码清单 3.42：“联系”页面的内容
app/views/static_pages/contact.html.erb

```
<% provide(:title, "Contact") %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
```

```
    <a href="https://www.railstutorial.org/contact">contact page</a>.
  </p>
```

3.4.4 设置根路由

我们修改了网站中的页面，也顺利开始编写测试了，在继续之前，我们要设置应用的根路由。与 1.2.4 节和 2.2.2 节的做法一样，我们要修改 `routes.rb` 文件，把根路径 `/` 指向我们选择的页面。这里我们要指向前面创建的首页。（还建议把 3.1 节添加的 `hello` 动作从 `Application` 控制器中删除。）如代码清单 3.43 所示，我们要把 `root` 规则由

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

改成

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

这样对 `/` 的请求就交给 `StaticPages` 控制器的 `home` 动作处理了。修改路由后，首页如图 3.9 所示。

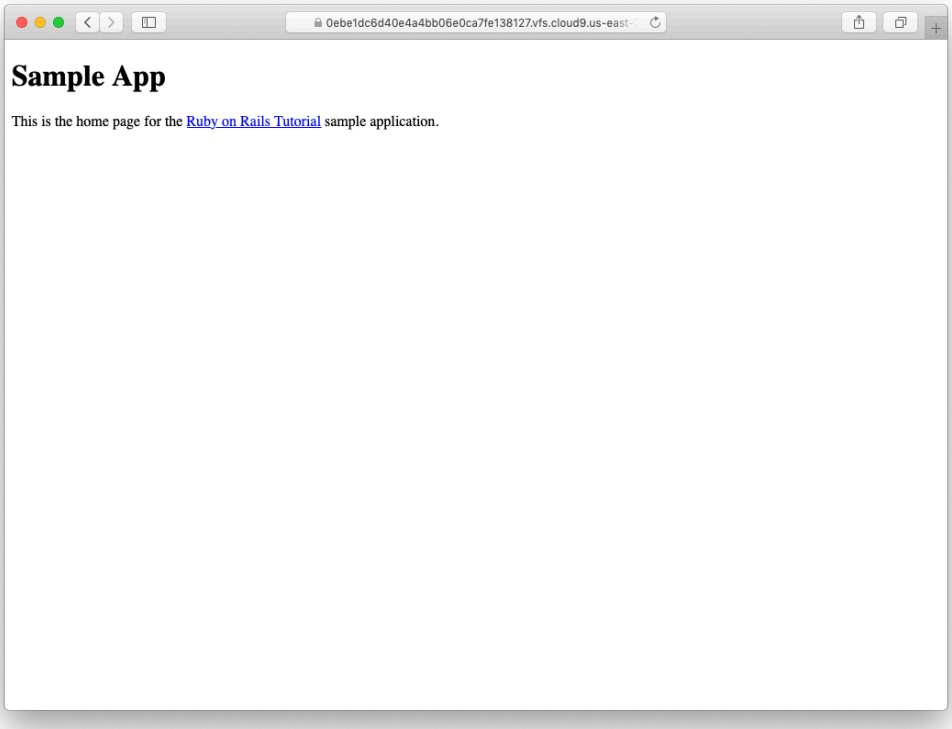


图 3.9：在根路由上显示的首页

代码清单 3.43：把根路由指向“首页”
config/routes.rb

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

练习

1. 添加代码清单 3.43 中的根路由后会得到一个名为 `root_url` 的辅助方法（与 `static_pages_home_url` 类似）。把代码清单 3.44 中的 `FILL_IN` 改成真正的代码，测试根路由。
2. 因为事先编写好了代码清单 3.43 中的那些代码，前一题的测试已经可以通过。但是，我们很难确信测试是正确的。修改代码清单 3.43 中的代码，把根路由注释掉（如代码清单 3.45 所示，4.2 节会进一步介绍注释），先“遇红”。然后，去掉注释（还原成代码清单 3.43 那样），确认测试可以通过。

代码清单 3.44：测试根路由 GREEN
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get root" do
    get FILL_IN
    assert_response FILL_IN
  end

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

  test "should get help" do
    get static_pages_help_url
    assert_response :success
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
  end
end
```

代码清单 3.45：注释掉根路由，让测试失败 RED
config/routes.rb

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

3.5 小结

总的来说，本章几乎没做什么：我们从静态页面开始，最后得到的几乎还是静态内容的页面。不过从表面上看，我们使用了 Rails 中的控制器、动作和视图，现在已经可以向网站中添加任何动态内容了。本书的后续章节会告诉你怎么添加。

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

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

然后，使用 1.3.4 节介绍的方法，把改动合并到主分支中：[13]

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

每次完成一些工作后，最好把代码推送到远程仓库中（如果你按照 1.3.3 节中的步骤做了，远程仓库在 GitHub 中）：

```
$ git push
```

笔者还建议你把这个应用部署到 Heroku 中：

```
$ rails test
$ git push heroku
```

在部署之前先运行测试组件是个好习惯。

3.5.1 本章所学

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

3.6 高级测试技术

这一节选读，介绍本书配套视频（https://screencasts.railstutorial.org）中使用的测试设置。额外的设置包含两方面内容：增强版通过和失败报告程序（3.6.1 节）；一个自动测试运行程序，检测到文件有变化后自动运行相应的测试（3.6.2 节）。这一节使用的代码相对高级，放在这里只是为了查阅方便，现在并不期望你能理解。

这一节应该在主分支中修改：

```
$ git checkout master
```

13. 如果报错说合并会覆盖 Spring 进程 ID（PID）文件，在命令行中执行 `rm -f *.pid` 命令，把那个文件删掉。

3.6.1 MiniTest 报告程序

尽管很多系统（包括我们使用的云 IDE）会在测试组件失败和通过时显示相应的颜色，但是添加 MiniTest 报告程序能进一步改善测试的输出，因此建议你在测试辅助文件中加入代码清单 3.46 中的内容，[14] 充分利用代码清单 3.2 中的 `minitest-reporters` gem。

代码清单 3.46：配置测试，显示红色和绿色
test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
require_relative '../config/environment'
require 'rails/test_help'
require "minitest/reporters"
Minitest::Reporters.use!

class ActiveSupport::TestCase
  # Run tests in parallel with specified workers
  parallelize(workers: :number_of_processors)

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

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

修改后，在云 IDE 中显示的效果如图 3.10 所示。

图 3.10：在云 IDE 中测试由红变绿

3.6.2 使用 Guard 自动测试

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

14. 代码清单 3.46 既使用了单引号形式字符串，也使用了双引号形式字符串，因为 `rails new` 命令生成的文件使用单引号字符串，而 MiniTest 报告程序的文档中使用双引号。在 Ruby 代码中混用两种形式的字符串很常见，详情参见 4.2.1 节。

代码清单 3.2 中的 Gemfile 已经包含了 guard gem，所以我们只需初始化即可：

```
$ bundle _2.2.13_ exec guard init
Writing new Guardfile to /home/ec2-user/environment/sample_app/Guardfile
00:51:32 - INFO - minitest guard added to Guardfile, feel free to edit it
```

然后，编辑生成的 Guardfile 文件，让 Guard 在集成测试和视图发生变化后运行正确的测试，如代码清单 3.47 所示。

代码清单 3.47：修改后的 Guardfile 文件

```
require 'active_support/core_ext/string'
# Defines the matching rules for Guard.
guard :minitest, spring: "bin/rails test", all_on_start: false do
  watch(%r{^test/(.*)/?(.*)_test\.rb$})
  watch('test/test_helper.rb') { 'test' }
  watch('config/routes.rb') { interface_tests }
  watch(%r{app/views/layouts/*}) { interface_tests }
  watch(%r{^app/models/(.*?)\.rb$}) do |matches|
    ["test/models/#{matches[1]}_test.rb",
     "test/integration/microposts_interface_test.rb"]
  end
  watch(%r{^test/fixtures/(.*?)\.yml$}) do |matches|
    "test/models/#{matches[1].singularize}_test.rb"
  end
  watch(%r{^app/mailers/(.*?)\.rb$}) do |matches|
    "test/mailers/#{matches[1]}_test.rb"
  end
  watch(%r{^app/views/(.*)_mailer/.*$}) do |matches|
    "test/mailers/#{matches[1]}_mailer_test.rb"
  end
  watch(%r{^app/controllers/(.*?)_controller\.rb$}) do |matches|
    resource_tests(matches[1])
  end
  watch(%r{^app/views/([^/]*?)/.*\.html\.erb$}) do |matches|
    ["test/controllers/#{matches[1]}_controller_test.rb"] +
    integration_tests(matches[1])
  end
  watch(%r{^app/helpers/(.*?)_helper\.rb$}) do |matches|
    integration_tests(matches[1])
  end
  watch('app/views/layouts/application.html.erb') do
    'test/integration/site_layout_test.rb'
  end
  watch('app/helpers/sessions_helper.rb') do
    integration_tests << 'test/helpers/sessions_helper_test.rb'
  end
  watch('app/controllers/sessions_controller.rb') do
    ['test/controllers/sessions_controller_test.rb',
     'test/integration/users_login_test.rb']
  end
  watch('app/controllers/account_activations_controller.rb') do
    'test/integration/users_signup_test.rb'
```

```
  end
  watch(%r{app/views/users/*}) do
    resource_tests('users') +
    ['test/integration/microposts_interface_test.rb']
  end
end

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

# Returns all tests that hit the interface.
def interface_tests
  integration_tests << "test/controllers"
end

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

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

在云 IDE 中还有一步要做：执行下述相当晦涩的命令，让 Guard 监视项目中的所有文件。

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf
$ sudo sysctl -p
```

配置好 Guard 之后，应该打开一个新终端窗口（与 1.2.2 节启动 Rails 服务器的做法一样），在其中执行下述命令（图 3.11）：

```
$ bundle _2.2.13_ exec guard
```

图 3.11：在云 IDE 中使用 Guard

代码清单 3.47 中的规则针对本书做了优化，例如，修改控制器后会自动运行集成测试。如果想运行所有测试，在 guard> 提示符中直接按回车键。

若想退出 Guard，按 Ctrl-D 组合键。如果想为 Guard 添加其他匹配器，参阅代码清单 3.47、Guard 的自述文件（https://github.com/guard/guard#readme）和维基（https://github.com/guard/guard/wiki）。

如果测试组件意外失败，退出 Guard，停止 Spring（Rails 通过它预加载信息，提升测试的速度），然后重启试试：

```
$ bin/spring stop    # 如果测试意外失败，执行这个命令试试
$ bundle _2.2.13_ exec guard
```

继续之前，应该添加改动，做次提交：

```
$ git add -A
$ git commit -m "Complete advanced testing setup"
```

第 4 章 Rails 背后的 Ruby

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

4.1 导言

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

与 3.2 节一样，我们将在单独的主题分支中修改：

```
$ git checkout -b rails-flavored-ruby
```

到 4.5 节，我们再合并到 `master` 分支。

4.1.1 内置的辅助方法

前一章末尾我们修改了几乎是静态内容的页面，让它们使用 Rails 布局，把视图中的重复去掉了。我们使用的布局如代码清单 4.1 所示（和代码清单 3.37 一样）。

代码清单 4.1：演示应用的网站布局
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <meta charset="utf-8">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

我们把注意力集中在这一行：

```
<%= stylesheet_link_tag 'application', media: 'all',
```

```
                                    'data-turbolinks-track': 'reload' %>
```

这行代码使用 Rails 内置的 `stylesheet_link_tag` 方法（详情参见 Rails API [1] 文档，https://api.rubyonrails.org/classes/ActionView/Helpers/AssetTagHelper.html#method-i-stylesheet_link_tag），在所有媒介类型（包括电脑屏幕和打印机）中引入 `application.css`。对有经验的 Rails 开发者来说，这行代码看起来很简单，但是其中至少有四个 Ruby 知识点可能会让你感到困惑：内置的 Rails 方法、调用方法时不用括号、符号（symbol）和散列（hash）。这几点本章都会介绍。

4.1.2 自定义辅助方法

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

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

这行代码要求每个视图都要使用 `provide` 方法定义标题，例如：

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

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

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

也就是说，标题中有公共部分，但前面还显示一条竖线。

为了解决这个问题，我们要自定义一个辅助方法，名为 `full_title`。如果视图中没有定义页面的标题，`full_title` 返回标题的公共部分，即“Ruby on Rails Tutorial Sample App”；如果定义了，则在变动部分后面加上一个竖线，如代码清单 4.2 所示。[2]

代码清单 4.2：定义 `full_title` 辅助方法
app/helpers/application_helper.rb

```
module ApplicationHelper

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

1. API 指应用程序编程接口（Application Programming Interface），即一些方法和约定，是与软件系统交互的一层抽象。作为开发者，我们无需了解程序的内部运作机制，只要熟悉公开的 API 即可。以这里的 `stylesheet_link_tag` 方法为例，我们不需要知道它是如何实现的，知道它的用法即可。

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

```
      else
        page_title + " | " + base_title
      end
    end
  end
```

现在，这个辅助方法定义好了，我们可以用它来简化布局。把下面这行：

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

改成：

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

如代码清单 4.3 所示。

代码清单 4.3：使用 full_title 辅助方法的网站布局 GREEN
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <meta charset="utf-8">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

为了让这个辅助方法起作用，我们要在首页的视图中把不必要的单词“Home”删掉，只保留标题的公共部分。首先，我们要修改测试代码，如代码清单 4.4 所示，确认标题中没有字符串 "Home"。

代码清单 4.4：修改首页的标题测试 RED
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

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

  test "should get help" do
    get static_pages_help_url
    assert_response :success
```

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

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

接着，运行测试组件，确认有一个测试失败：[3]

代码清单 4.5：RED

```
$ rails test
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
```

为了让测试通过，我们要把首页视图中的 `provide` 那行删除，如代码清单 4.6 所示。

代码清单 4.6：没定义页面标题的首页视图 GREEN
app/views/static_pages/home.html.erb

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

现在测试应该能通过了：

代码清单 4.7：GREEN

```
$ rails test
```

（注意，之前运行 `rails test` 时都显示了通过和失败测试的数量，为了行文简洁，自此以后都会省略这些信息。）

与 4.1.1 节引入应用的样式表那行代码一样，代码清单 4.2 的内容对有经验的 Rails 开发者来说也很简单，但其中有很多重要的 Ruby 知识：模块、方法定义、可选的方法参数、注释、局部变量赋值、布尔值、控制流、字符串拼接和返回值。本章会一一介绍这些知识。

4.2 字符串和方法

我们学习 Ruby 主要使用的工具是 Rails 控制台，一个与 Rails 应用交互的命令行工具，2.3.3 节介绍过。控制台基于 Ruby 的交互程序（`irb`）开发，因此能使用 Ruby 语言的全部功能。（4.4.4 节会介绍，控制台还可以访问 Rails 环境。）

如果使用云 IDE，建议添加几个 irb 配置参数。使用简单的 `nano` 文本编辑器打开家目录中的 `.irbrc` 文件：[4]

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

3. 这里运行测试组件是为了告诉你有这么一步，实际开发中笔者一般都使用 Guard 自动运行测试（3.6.2 节）。
4. nano 编辑器对新手来说更简单，不过笔者几乎都使用 Vim 做这种简单的编辑。

（这个文件可能不存在，保存后会自动创建。）写入代码清单 4.8 中的内容。这段代码的作用是简化 irb 提示符，以及禁用一些烦人的自动缩进行为。

代码清单 4.8：添加几个 irb 配置
~/.irbrc

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

编辑好之后，按 Ctrl-X 组合键退出 nano，然后输入 y 确认保存了 ~/.irbrc 文件。

现在，执行下述命令启动控制台：

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

默认情况下，控制台在开发环境中启动，这是 Rails 定义的三个独立环境之一（另外两个是测试环境和生产环境）。这三个环境的区别对本章不重要，但是对后文就重要了，我们将在 7.1.1 节详细说明。

控制台是学习的好工具，请尽情探索它的用法。别担心，你（几乎）不会破坏任何东西。如果在控制台中遇到问题，可以按 Ctrl-C 组合键结束当前执行的操作，或者按 Ctrl-D 组合键直接退出。与常规的 shell 终端一样，我们可以使用上箭头获取前一个命令，这能节省不少时间。

在阅读本章后续内容的过程中，你会发现查阅 Ruby API（https://ruby-doc.org）很有帮助。API 中有很多信息（或许太多了），例如，如果想进一步了解 Ruby 字符串，可以查看 String 类的文档。

在讲解的过程中，我们有时会用到注释。Ruby 中的注释以井号 #（也叫“散列符号”，或者更诗意一点，叫“散列字元”）开头，一直到行尾结束。Ruby 会忽略注释，但是注释对人类读者（往往也包括代码的编写者自己）很有用。在下面的代码中

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

第一行就是注释，说明其后方法的作用。

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

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

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

4.2.1 字符串

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

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

```
>> "foo"       # 非空字符串
=> "foo"
```

这些是字符串字面量，使用双引号（"）创建。控制台回显的是每一行的求值结果。这里，字符串字面量的结果就是字符串本身。

我们还可以使用 + 号拼接字符串：

```
>> "foo" + "bar"     # 字符串拼接
=> "foobar"
```

"foo" 与 "bar" 拼接得到的结果是字符串 "foobar"。[5]

另一种创建字符串的方式是使用特殊的句法 #{} 进行插值操作：[6]

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

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

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name    # 字符串拼接，中间加了空格
=> "Michael Hartl"
>> "#{first_name} #{last_name}"    # 等效的插值
=> "Michael Hartl"
```

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

打印字符串

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

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

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

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

```
>> print "foo"    # 打印字符串，不换行
foo=> nil
```

5. 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（http://www.catb.org/jargon/html/F/foo.html）。

6. 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"。

可以看出，输出的 foo 后面直接跟着提示符。

换行通常使用“\n”符号表示。我们可以在字符串中加上换行符，让 print 与 puts 的效果一样：

```
>> print "foo\n"  # 作用与 `puts "foo"` 一样
foo
=> nil
```

单引号字符串

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

```
>> 'foo'          # 单引号字符串
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

不过，二者之间有个重要的区别：Ruby 不会对单引号字符串进行插值操作。

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

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

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

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

与前面的 # 符号一样，Ruby 要使用一个额外的反斜线来转义反斜线——在双引号字符串中，要表达一个反斜线就要使用两个反斜线。对简单的例子来说，这省不了多少事，但是如果有很多需要转义的字符就显出它的作用了：

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

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

练习

1. 把你当前所在的省份和城市分别赋值给 province 和 city 变量。
2. 使用字符串插值打印一个字符串（使用 puts 方法），在省份和城市之间加上逗号，例如“安徽省，合肥市”。
3. 把前一题中的逗号换成制表符。
4. 如果把前一题中的双引号换成单引号，结果如何？

4.2.2 对象和消息传送

在 Ruby 中，一切皆对象，包括字符串和 nil 都是。我们会在 4.4.2 节介绍对象在技术层面上的意义，不过一

般很难通过阅读一本书就理解对象，你要多看一些例子才能建立对对象的感性认识。

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

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

一般来说，传给对象的消息是“方法”，即在对象上定义的函数。[7] 字符串还可以响应 empty? 方法：

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

注意，empty? 方法末尾有个问号，这是 Ruby 的约定，说明方法返回的是布尔值，即 true 或 false。布尔值在控制流中特别有用：

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

如果分支很多，可以使用 elsif（else + if）：

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

布尔值还可以使用 &&（与）、||（或）和 !（非）运算符组合在一起使用：

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

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

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

```
>> nil.to_s
=> ""
```

结果显然是个空字符串，我们可以通过下面的方法串联（chain）验证这一点：

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

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

有一个特殊的方法可以测试对象是否为空，你或许能猜到是哪个方法：

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

下面的代码

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

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

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

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

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

除此之外，其他所有 Ruby 对象都是“真”值，数字 0 也是：

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

练习

1. 字符串“racecar”有多长？
2. 使用 reverse 方法确认前一题中的字符串经过反转之后内容仍然一样。
3. 把字符串“racecar”赋值给变量 s。使用比较运算符 == 确认 s 与 s.reverse 相等。
4. 代码清单 4.9 的运行结果是什么？如果把 s 变量的值改成“onomatopoeia”呢？提示：使用上箭头获取前一个命令，然后编辑。

代码清单 4.9：简单的回文测试

```
>> puts "It's a palindrome!" if s == s.reverse
```

4.2.3 定义方法

在控制台中，可以像定义 home 动作（代码清单 3.9）和 full_title 辅助方法（代码清单 4.2）一样定义方法。（在控制台中定义方法有点麻烦，我们通常在文件中定义，这里只是为了演示。）例如，我们要定义一个名为 string_message 的方法，它有一个参数，返回值取决于参数是否为空：

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

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

注意，Ruby 方法不用显式指定返回值，方法的返回值是最后一个语句的计算结果。上述函数的返回值是两个字符串中的某一个，具体是哪一个取决于 str 参数是否为空。在 Ruby 方法中也可以显式指定返回值，下面这个方法和前面的等价：

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

（细心的读者可能会发现，其实没必要使用第二个 return，那一行是方法的最后一个表达式，不管有没有 return，字符串 "The string is nonempty." 都会作为返回值返回。不过两处都加上 return 看着更和谐。）

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

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

```
The string is nonempty.
```

练习

1. 把代码清单 4.10 中的 `FILL_IN` 换成正确的比较表达式，定义一个测试回文的方法。提示：使用代码清单 4.9 中的比较表达式。
2. 使用前一题定义的方法测试“racecar”和“onomatopoeia”，确认第一个词是回文，而第二个词不是。
3. 在 `palindrome_tester("racecar")` 上调用 `nil?` 方法，确认它的返回值是 `nil`（即在那个方法上调用 `nil?` 方法的结果是 `true`）。这是因为 `palindrome_tester` 方法是把结果打印出来的，而没有返回。

代码清单 4.10：测试回文的方法

```
>> def palindrome_tester(s)
>>   if FILL_IN
>>     puts "It's a palindrome!"
>>   else
>>     puts "It's not a palindrome."
>>   end
>> end
```

4.2.4 回顾标题的辅助方法

下面我们来理解一下代码清单 4.2 中的 `full_title` 辅助方法，[8] 在其中加上注解之后如代码清单 4.11 所示。

代码清单 4.11：注解 `full_title` 方法
app/helpers/application_helper.rb

```
module ApplicationHelper

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

我们把方法定义、变量赋值、布尔测试、流程控制和字符串拼接[9] 利用起来，定义了一个可以在网站布局中使用的辅助方法。这里还有一个知识点——`module ApplicationHelper`：模块为我们提供了一种把相关方法组织在一起的方式，我们可以使用 `include` 把模块插入其他类中。编写普通的 Ruby 程序时，你要自己定义模块，然后再显式将其引入类中，但是辅助方法所在的模块会由 Rails 为我们引入，结果是，`full_title` 方法

8. 其实还有一个地方我们不理解，那就是 Rails 是怎么把这些联系在一起的：把 URL 映射到动作上，让 `full_title` 辅助方法可以在视图中使用，等等。这是个很有意思的话题，建议你以后好好了解一下。不过使用 Rails 并不需要完全了解 Rails 的运作机制。

9. 这里你可能想使用字符串插值，其实本书前几版使用的都是插值，但 `provide` 方法会把字符串转换成 `SafeBuffer` 对象，而不是普通的字符串。插入视图模板的 HTML 会过度转义，把“Help’s on the way”转换成“Help's on the way”。（感谢读者 Jeremy Fleischman 指出这个小问题。）

自动在所有视图中可用。

4.3 其他数据类型

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

4.3.1 数组和值域

数组是一组具有特定顺序的元素。前文还没用过数组，不过理解数组对理解散列有很大帮助（4.3.3 节），也有助于理解 Rails 中的数据模型（例如 2.3.3 节用到的 `has_many` 关联，13.1.3 节将详细说明）。

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

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

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

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

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

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

我们看到，Ruby 使用方括号获取数组中的元素。除了方括号之外，Ruby 还为一些经常需要获取的元素提供了别名方法：[10]

```
>> 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 == 运算符比较
```

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

```
=> true
```

最后一行用到了等值比较运算符 ==，Ruby 和其他语言一样还提供了 !=（不等）等其他运算符：

```
>> x = a.length       # 与字符串一样，数组也可以响应 length 方法
=> 3
>> x == 3
=> true
>> x == 1
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
```

除了 length（上述代码的第一行）之外，数组还可以响应一系列其他方法：

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

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

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

还可以使用 push 方法向数组中添加元素，或者使用等价的 << 运算符：

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

最后一个命令说明，可以把添加操作串在一起使用；也说明，与其他语言不同，在 Ruby 中数组可以包含不

同类型的数据（本例中包含整数和字符串）。

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

```
>> a
=> [42, 8, 17, 6, 7, "foo", "bar"]
>> a.join                        # 没有连接符
=> "4281767foobar"
>> a.join(', ')                  # 连接符是一个逗号和空格
=> "42, 8, 17, 6, 7, foo, bar"
```

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

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

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

值域经常用于获取数组中的一组元素：

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

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

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

值域也可以使用字符定义：

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

练习

1. 在逗号和空格处分拆字符串“A man, a plan, a canal, Panama”，把结果赋值给变量 `a`。
2. 不指定连接符，把 `a` 连接起来，然后把结果赋值给变量 `s`。
3. 在空白处分拆 `s`，然后再连接起来。使用代码清单 4.10 中的方法确认得到的结果不是回文。使用 `downcase` 方法，确认 `s.downcase` 是回文。
4. 创建字母 `a` 到 `z` 的值域，第 7 个元素是什么？ 把值域反过来呢？提示：两次都要把值域转换成数组。

4.3.2 块

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

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

这段代码在值域 `(1..5)` 上调用 `each` 方法，然后又把 `{ |i| puts 2 * i }` 块传给 `each` 方法。`|i|` 两边的竖线在 Ruby 中用来定义块变量。只有方法本身才知道如何处理后面跟着的块。这里，值域的 `each` 方法会处理后面的块，块中有一个局部变量 `i`，`each` 会把值域中的各个值传进块中，然后执行块中的代码。

花括号是表示块的一种方式，除此之外还有另一种方式：

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

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

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

上面的代码用 `number` 代替了 `i`，这是为了告诉你，变量名可以使用任何值。

除非你已经有了一些编程知识，否则对块的理解是没有捷径的。你要做的是多看，看多了就会习惯这种用法。[11] 幸好，人类擅长从实例中归纳出一般性。下面举几个例子，其中几个用到了 `map` 方法：

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

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

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

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

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

最后再看一个使用块的例子。我们看一下代码清单 4.4 中的一个测试用例：

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

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

现在我们来分析一下 1.4.2 节生成随机二级域名时使用的那行 Ruby 代码：[12]

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

下面一步步分解：

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

12. 第 1 章说过，使用 ('a'..'z').to_a.sample(8).join 也能得到相同的结果，而且更简洁。

练习

1. 创建值域 `0..16`，把前 17 个元素的平方打印出来。
2. 定义一个名为 `yeller` 的方法，它的参数是一个由字符组成的数组，返回值是一个字符串，由数组中字符的大写形式组成。确认 `yeller(['o', 'l', 'd'])` 的返回值是 `OLD`。提示：要用到 `map`、`upcase` 和 `join` 方法。
3. 定义一个名为 `random_subdomain` 的方法，返回八个随机字母组成的字符串。
4. 把代码清单 4.12 中的问号换成正确的方法，结合 `split`、`shuffle` 和 `join` 方法，把指定字符串中的字符打乱。

代码清单 4.12：字符串打乱函数的骨架

```
>> def string_shuffle(s)
>>   s.?('').?.?
>> end
>> string_shuffle("foobar")
=> "oobfra"
```

4.3.3 散列和符号

散列（hash）本质上就是数组，只不过它的索引不局限于只能使用数字。（实际上在一些语言中，特别是 Perl，因为这个原因而把散列叫做“关联数组”。）散列的索引（或者叫“键”）几乎可以使用任何对象。例如，可以使用字符串做键：

```
>> user = {}                          # {} 是一个空散列
=> {}
>> 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"]                 # 获取元素的方式与数组类似
=> "Michael"
>> user                               # 散列的字面量形式
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

散列通过一对花括号中包含一些键值对的形式表示，如果只有一对花括号而没有键值对（`{}`）就是一个空散列。注意，散列中的花括号和块中的花括号不是一个概念。（是的，这可能会让你感到困惑。）散列虽然与数组类似，但二者却有一个很重要的区别：散列中的元素没有特定的顺序。[13] 如果看重顺序，就要使用数组。

通过方括号的形式每次定义一个元素的方式不太敏捷，使用 `=>` 分隔的键值对这种字面量形式定义散列要简洁得多：

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

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

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

目前为止散列的键都使用字符串，不过在 Rails 中用符号（symbol）做键更常见。符号看起来有点儿像字符串，只不过没有包含在一对引号中，而是在前面加一个冒号。例如，`:name` 就是一个符号。你可以把符号看成没有约束的字符串：[14]

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

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

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

只要以字母开头，其后都使用单词中常用的字符就没事。

用符号做键时，可以按照如下的方式定义 `user` 散列：

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

从上面的例子可以看出，散列中没有定义的键对应的值是 `nil`。

由于符号做键的情况太普遍了，Ruby 1.9 干脆为这种用法定义了一种新句法：

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

第二种句法把“符号 =>”变成了“键名:”形式：

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

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

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

面）。

散列中元素的值可以是任何对象，甚至是另一个散列，如代码清单 4.13 所示。

代码清单 4.13：嵌套散列

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

Rails 大量使用这种散列中有散列的形式（或称为“嵌套散列”），从 7.3 节起会接触到。

与数组和值域一样，散列也能响应 each 方法。例如，下面是一个名为 flash 的散列，它的键是两个判断条件，:success 和 :danger：

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

注意，数组的 each 方法后面的块只有一个变量，而散列的 each 方法后面的块接受两个变量，分别表示键和对应的值。所以散列的 each 方法每次遍历都会以一个键值对为单位进行。

这段代码用到了很有用的 inspect 方法，它的作用是返回被调用对象的字符串字面量表示形式：

```
>> puts (1..5).to_a            # 把数组以字符串的形式打印出来
1
2
3
4
5
>> puts (1..5).to_a.inspect    # 输出数组的字面量形式
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

顺便说一下，因为使用 inspect 打印对象的方式经常使用，为此还有一个专门的快捷方式，p 方法：[15]

```
>> p :name             # 等价于 'puts :name.inspect'
:name
```

15. 其实二者之间有些细微差别，p 返回打印的对象，而 puts 始终返回 nil。感谢读者 Katarzyna Siwek 指出这一点。

练习

1. 定义一个散列，把键设为 'one'、'two' 和 'three'，对应的值分别是 'uno'、'dos' 和 'tres'。迭代这个散列，把各个键值对以 "'#key' in Spanish is '#value'" 的形式打印出来。
2. 创建三个散列，分别命名为 person1、person2 和 person3，把名和姓赋值给 :first 和 :last 键。然后创建一个名为 params 的散列，让 params[:father] 对应 person1，params[:mother] 对应 person2，params[:child] 对应 person3。验证一下 params[:father][:first] 的值是否正确。
3. 定义一个散列，使用符号做键，分别表示名字、电子邮件地址和密码摘要，把键对应的值分别设为你的名字、电子邮件地址和一个由 16 个随机小写字母组成的字符串。
4. 找一个在线 Ruby API，查阅散列的 merge 方法。下述表达式的值是什么？

```
{ "a" => 100, "b" => 200 }.merge({ "b" => 300 })
```

4.3.4 重温引入 CSS 的代码

现在我们要重新认识一下代码清单 4.1 在布局中引入层叠样式表的代码：

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track': 'reload' %>
```

我们现在基本上可以理解这行代码了。4.1 节简单提到过，Rails 定义了一个特殊的函数用于引入样式表，下面的代码

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track': 'reload'
```

就是对这个函数的调用。不过还有几个奇怪的地方。第一，括号哪去了？在 Ruby 中，括号是可以省略的，所以下面两种写法是等价的：

```
# 调用函数时可以省略括号
stylesheet_link_tag('application', media: 'all',
                                   'data-turbolinks-track': 'reload')
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track': 'reload'
```

第二，media 部分显然是一个散列，但是怎么没用花括号？调用函数时，如果最后一个参数是散列，可以省略花括号。所以下面两种写法是等价的：

```
# 如果最后一个参数是散列，可以省略花括号
stylesheet_link_tag 'application', { media: 'all',
                                     'data-turbolinks-track': 'reload' }
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track': 'reload'
```

最后，为什么下述代码写成两行还能正确解析？

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track': 'reload'
```

因为在这种情况下，Ruby 不关心有没有换行。[16] 之所以把这段代码写成两行，是要保证每行代码不超过 80

16. 换行符在一行的结尾处，作用是开始新的一行。在代码中，换行符用 \n 表示。

个字符。[17]

所以，下面这段代码

```
stylesheet_link_tag 'application', media: 'all',
                                  'data-turbolinks-track': 'reload'
```

调用了 `stylesheet_link_tag` 函数，并且传入两个参数：一个是字符串，指明样式表的路径；另一个是散列，包含两个元素，第一个指明媒介类型，第二个启用 Rails 4.0 增加的 Turbolink 功能。因为使用的是 `<%= %>`，函数的执行结果会通过 ERb 插入模板中。如果在浏览器中查看网页的源码，会看到引入样式表所用的 HTML，如代码清单 4.14 所示。（你可能会在 CSS 的文件名后看到额外的字符，例如 `?body=1` 和一长串十六进制数。这是 Rails 加入的，用以确保修改 CSS 后浏览器会重新加载。按照设计，那串十六进制数是唯一的，因此你看到的结果与代码清单 4.14 中的不一样。）

代码清单 4.14：引入 CSS 的代码生成的 HTML

```
<link rel="stylesheet" media="all" href="/assets/application.self-
f0d704deea029cf000697e2c0181ec173a1b474645466ed843eb5ee7bb215794.css?body=1"
data-turbolinks-track="reload" />
```

4.4 Ruby 类

前面说过，Ruby 中的一切都是对象。本节我们要自己定义一些对象。Ruby 和其他面向对象的语言一样，使用类来组织方法，然后实例化类，创建对象。如果你刚接触面向对象编程（Object-Oriented Programming，OOP），这些听起来都似天书一般，那我们来看一些例子吧。

4.4.1 构造方法

我们看过很多使用类初始化对象的例子，不过还没自己动手做过。例如，我们使用双引号初始化一个字符串，双引号就是字符串的字面构造方法：

```
>> s = "foobar"        # 使用双引号字面构造方法
=> "foobar"
>> s.class
=> String
```

我们看到，字符串可以响应 `class` 方法，返回值是字符串所属的类。

除了使用字面构造方法之外，还可以使用等价的具名构造方法（named constructor），即在类名上调用 `new` 方法：[18]

```
>> s = String.new("foobar")   # 字符串的具名构造方法
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

17. 数列数会让你发疯的，所以很多文本编辑器都提供了一个视觉标识。例如，如果再看一下图 1.12 的话，你可能会发现右边有一条细线，它可以帮助你把一行代码控制在 80 个字符以内。云 IDE 默认会显示这条竖线。在 Sublime Text 中，可以在如下菜单中找到这个功能：View > Ruler > 78，或：View > Ruler > 80。

18. 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 或以上版本。

这段代码中使用的具名构造方法和字面构造方法是等价的，只是更能表明我们的意图。

数组与字符串类似，也有具名构造方法：

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

不过散列就有点不同了。数组的构造方法 `Array.new` 可接受一个可选的参数指明数组的初始值，`Hash.new` 可接受一个参数指明元素的默认值，即当键不存在时返回的值：

```
>> h = Hash.new
=> {}
>> h[:foo]            # 试图获取不存在的键 :foo 对应的值
=> nil
>> h = Hash.new(0)    # 让不存在的键返回 0 而不是 nil
=> {}
>> h[:foo]
=> 0
```

在类上调用的方法，如这里的 `new`，叫类方法（class method）。在类上调用 `new` 方法，得到的结果是这个类的对象，也叫这个类的实例（instance）。在实例上调用的方法，例如 `length`，叫实例方法（instance method）。

练习

1. 从 1 到 10 的值域，它的字面构造方法是什么？
2. 使用 `Range` 类和 `new` 方法怎么编写构造方法？提示：这里要为 `new` 方法提供两个参数。
3. 使用 `==` 运算符确认前两题使用字面构造方法和具名构造方法创建的值域相等。

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` 方法找出继承关系：

```
>> 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  # 从 Ruby 1.9 开始，BasicObject 是基类
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

这个继承关系如图 4.1 所示。可以看到，`String` 的父类是 `Object`，`Object` 的父类是 `BasicObject`，但是 `BasicObject` 就没有父类了。这样的关系对每个 Ruby 对象都适用：只要在类的继承关系上往上多走几层，就会发现 Ruby 中的每个类最终都继承自 `BasicObject`，而它本身没有父类。这就是“Ruby 中一切皆对象”在技术层面上的意义。

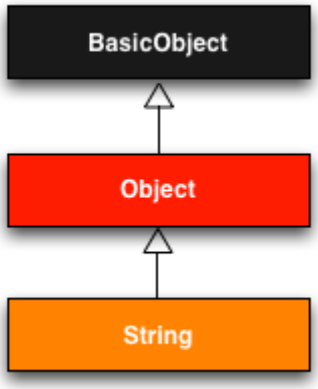


图 4.1：String 类的继承关系

要想更深入地理解类，最好的方法是自己动手编写一个。我们来定义一个名为 Word 的类，其中有一个名为 palindrome? 的方法，如果单词顺读和反读都一样就返回 true：

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

我们可以像下面这样使用该类：

```
>> w = Word.new              # 创建一个 Word 对象
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

如果你觉得这个例子有点小题大做，很好，我的目的达到了。定义一个新类，可是只创建一个接受一个字符串为参数的方法，这么做很古怪。既然单词是字符串，让 Word 继承 String 不就行了，如代码清单 4.15 所示。（请先退出控制台，然后在控制台中输入这些代码，这样才能把之前定义的 Word 类清除掉。）

代码清单 4.15：在控制台中定义 Word 类

```
>> class Word < String             # Word 继承自 String
>>   # 如果字符串和反转后相等就返回 true
>>   def palindrome?
>>     self == self.reverse        # self 代表这个字符串本身
>>   end
>> end
=> nil
```

其中，Word < String 在 Ruby 中表示继承（3.2 节简介过），这样除了定义 palindrome? 方法之外，Word 还拥有所有字符串拥有的方法：

```
>> s = Word.new("level")    # 创建一个 Word 实例，初始值为 "level"
=> "level"
```

```
>> s.palindrome?            # Word 实例可以响应 palindrome? 方法
=> true
>> s.length                 # Word 实例还继承了普通字符串的所有方法
=> 5
```

Word 继承自 String，我们可以在控制台中查看类的继承关系：

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

这个继承关系如图 4.2 所示。

注意，在代码清单 4.15 中检查单词和单词的反转是否相同时，要在 Word 类中访问单词。这在 Ruby 中使用 self 关键字引用：在 Word 类中，self 代表的是对象本身。所以我们可以使用

```
self == self.reverse
```

检查单词是否为回文。其实，在类中调用方法或访问属性时可以不用 self.（赋值例外），因此也可以写成

```
self == reverse
```

练习

1. 值域的类继承关系是怎样的？散列和符号呢？
2. 把代码清单 4.15 中的 self.reverse 换成 reverse，确认 palindrome? 方法依然可用。

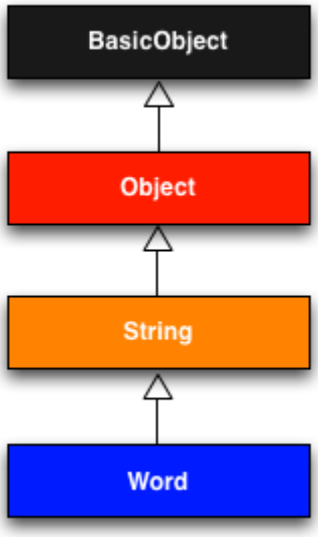


图 4.2：代码清单 4.15 中定义的 Word 类（非内置类）的继承关系

4.4.3 修改内置的类

虽然继承是个强大的功能，不过在判断回文这个例子中，如果能把 `palindrome?` 加入 `String` 类就更好了，这样（除了其他方法外）我们可以在字符串字面量上调用 `palindrome?` 方法。现在我们还不能直接调用：

```
>> "level".palindrome?
NoMethodError: undefined method `palindrome?' for "level":String
```

有点令人惊讶的是，Ruby 允许你这么做，Ruby 中的类可以被打开进行修改，允许像我们这样的普通人添加方法：

```
>> class String
>>   # 如果字符串和反转后相等就返回 true
>>   def palindrome?
>>     self == self.reverse
>>   end
>> end
=> nil
>> "deified".palindrome?
=> true
```

（笔者不知道哪一个更厉害：Ruby 允许向内置的类中添加方法，或 `"deified"` 是一个回文。）

修改内置的类是个很强大的功能，不过功能强大意味着责任也大，如果没有很好的理由，向内置的类中添加方法是不好的习惯。Rails 自然有很好的理由。例如，在 Web 应用中我们经常要避免变量的值是空白的（blank），像用户名之类的就不应该是空格或空白，所以 Rails 为 Ruby 添加了一个 `blank?` 方法。Rails 控制台会自动加载 Rails 添加的功能，下面看几个例子（在 `irb` 中不可以）：

```
>> "".blank?
=> true
>> "      ".empty?
=> false
>> "      ".blank?
=> true
>> nil.blank?
=> true
```

可以看出，一个包含空格的字符串不是空的（empty），却是空白的（blank）。还要注意，`nil` 也是空白的。因为 `nil` 不是字符串，所以上面的代码说明了 Rails 其实是把 `blank?` 添加到 `String` 的基类 `Object` 中的。9.1 节会再介绍一些 Rails 扩展 Ruby 类的例子。

练习

1. 验证“racecar”是回文，而“onomatopoeia”不是。印度南部方言“Malayalam”是回文吗？提示：先变成小写。
2. 以代码清单 4.16 为模板，为 `String` 类添加 `shuffle` 方法。提示：参照代码清单 4.12。
3. 删掉 `self.`，确认代码清单 4.16 依然可用。

代码清单 4.16：添加到 `String` 类中的 `shuffle` 方法的模板

```
>> class String
>>   def shuffle
>>     self.?('').?.?
```

```
>>   end
>> end
>> "foobar".shuffle
=> "borafo"
```

4.4.4 控制器类

讨论类和继承时你可能觉得似曾相识，不错，我们之前见过，在 StaticPages 控制器中（代码清单 3.22）：

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

你现在应该可以理解，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，分别是 home、help 和 about。因为 Rails 控制台会加载本地的 Rails 环境，所以我们可以在控制台中创建控制器，查看它的继承关系：[19]

```
>> 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。

19. 你没必要知道继承关系中每个类的作用。笔者也不知道它们都是干什么的，而笔者从 2005 年起就开始使用 Ruby on Rails 了。这可能意味着以下两个问题中的一个：第一，笔者是个废柴；第二，不需要知道所有内部知识也能成为熟练的 Rails 开发者。我们当然都希望是后一种情况。

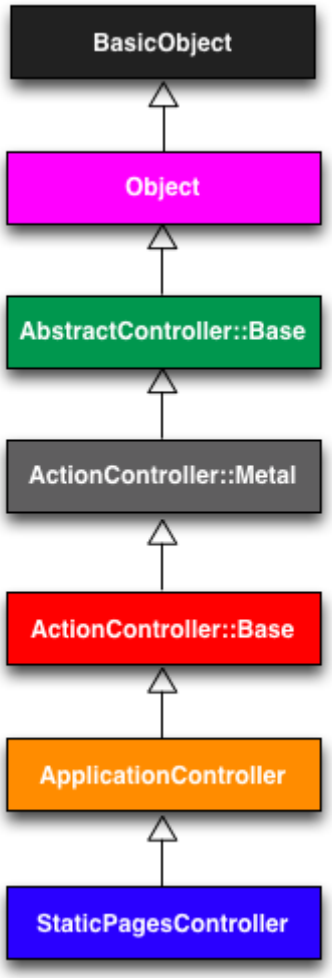


图 4.3：`StaticPagesController` 类的继承关系

我们还可以在控制台中调用控制器的动作，动作其实就是方法：

```
>> controller.home
=> nil
```

`home` 动作的返回值为 `nil`，因为它是空的。

注意，动作没有返回值，或至少没返回真正需要的值。如我们在第 3 章看到的，`home` 动作的目的是渲染网页，而不是返回一个值。但是，好像没在任何地方调用过 `StaticPagesController.new` 啊，到底怎么回事呢？

原因在于，Rails 是用 Ruby 编写的，但 Rails 不是 Ruby。有些 Rails 类就像普通的 Ruby 类那样，不过也有些则得益于 Rails 的强大功能。Rails 是单独的一门学问，应该跟 Ruby 分开学习和理解。

练习

1. 在第 2 章创建的玩具应用中运行 Rails 控制台，确认可以使用 `User.new` 创建用户对象。
2. 找出那个用户对象的类继承关系。

4.4.5 User 类

我们将自己定义一个类，以此结束对 Ruby 的介绍。这个类名为 User，目的是实现第 6 章用到的 User 模型。

目前为止，我们都在控制台中定义类，这样很快捷，但也有点不爽。现在我们要在应用的根目录中创建一个名为 example_user.rb 的文件，然后写入代码清单 4.17 中的内容。

代码清单 4.17：定义 User 类
example_user.rb

```
class User
  attr_accessor :name, :email

  def initialize(attributes = {})
    @name  = attributes[:name]
    @email = attributes[:email]
  end

  def formatted_email
    "#{@name} <#{@email}>"
  end
end
```

这段代码有很多地方要说明，我们一步步来。先看下面这行：

```
attr_accessor :name, :email
```

这行代码为用户的名字和电子邮件地址创建属性存取方法（attribute accessor），也就是定义读值方法（getter）和设值方法（setter），用于读取和设定 @name 和 @email 实例变量（2.2.2 节和 3.4.2 节的练习简介过）。在 Rails 中，实例变量的意义在于，它们自动在视图中可用。而通常实例变量的作用是在 Ruby 类中不同的方法之间传递值。（稍后会更详细地说明这一点。）实例变量都以 @ 符号开头，如未定义，值为 nil。

第一个方法，initialize，在 Ruby 中有特殊的意义：执行 User.new 时会调用它。这个 initialize 方法接受一个参数，attributes：

```
def initialize(attributes = {})
  @name  = attributes[:name]
  @email = attributes[:email]
end
```

attributes 参数的默认值是一个空散列，所以我们可以定义一个没有名字或没有电子邮件地址的用户。（回想一下 4.3.3 节的内容，如果键不存在会返回 nil，所以如果没定义 :name 键，attributes[:name] 返回 nil，attributes[:email] 也是一样。）

最后，类中定义了一个名为 formatted_email 的方法，使用被赋了值的 @name 和 @email 变量进行插值，组成一个有特定格式的电子邮件地址：

```
def formatted_email
  "#{@name} <#{@email}>"
end
```

@name 和 @email 都是实例变量（如 @ 符号所示），所以在 formatted_email 方法中自动可用。

下面打开控制台，加载（require）这个文件，实际使用一下这个类：

```
>> require './example_user'     # 加载 example_user 文件中代码的方式
=> true
>> example = User.new
=> #<User:0x224ceec @email=nil, @name=nil>
>> example.name                 # 返回 nil，因为 attributes[:name] 是 nil
=> nil
>> example.name = "Example User"           # 赋值一个非 nil 的名字
=> "Example User"
>> example.email = "user@example.com"      # 赋值一个非 nil 的电子邮件地址
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

这段代码中的点号 .，在 Unix 中指“当前目录”，'./example_user' 告诉 Ruby 在当前目录中寻找这个文件。接下来的代码创建一个空用户，然后通过直接赋值给相应的属性来提供名字和电子邮件地址（因为有 attr_accessor 所以才能赋值）。我们输入 example.name = "Example User" 时，Ruby 会把 @name 变量的值设为 "Example User"（email 属性类似），然后就可以在 formatted_email 方法中使用了。

4.3.4 节讲过，如果最后一个参数是散列，可以省略花括号。我们可以把一个预先定义好的散列传给 initialize 方法，再创建一个用户：

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User:0x225167c @email="mhartl@example.com", @name="Michael Hartl">
>> user.formatted_email
=> "Michael Hartl <mhartl@example.com>"
```

从第 7 章开始，我们将使用散列初始化对象，这种技术叫做批量赋值（mass assignment），在 Rails 应用中很常见。

练习

1. 在 User 类中定义一个名为 full_name 的方法，返回用户的名字和姓，中间以空格分开。
2. 添加一个名为 alphabetical_name 的方法，返回用户的姓和名字，中间以一个逗号和空格分开。
3. 确认 full_name.split 与 alphabetical_name.split(', ').reverse 得到的结果一样。

4.5 小结

至此，对 Ruby 语言的介绍结束了。第 5 章会好好利用这些知识来开发演示应用。

我们不会使用 4.4.5 节创建的 example_user.rb 文件，建议现在把它删除：

```
$ rm example_user.rb
```

然后把其他的改动提交到源码仓库中，合并到 master 分支之后，再推送到 GitHub，最好部署到 Heroku 中：

```
$ git commit -am "Add a full_title helper"
$ git checkout master
$ git merge rails-flavored-ruby
```

为了确保无误，最好在推送或部署之前运行测试组件：

```
$ rails test
```

确认无误后，再推送到 GitHub 中：

```
$ git push
```

最后，部署到 Heroku 中：

```
$ git push heroku
```

4.5.1 本章所学

- Ruby 提供了很多处理字符串的方法；
- 在 Ruby 中一切皆对象；
- 在 Ruby 中定义方法使用 `def` 关键字；
- 在 Ruby 中定义类使用 `class` 关键字；
- Ruby 内建支持的数据类型有数组、值域和散列；
- Ruby 块是一种灵活的语言接口，可以遍历可枚举的数据类型；
- 符号是一种标记，与字符串类似，但没有额外的束缚；
- Ruby 支持对象继承；
- 可以打开并修改 Ruby 内置的类；
- 单词“deified”是回文。