Rails 5 / RSpec 3

原书第 3 版

使用 RSpec 测试 Rails 应用

[美] Aaron Sumner 著
安道 译

原书第 3 版

使用 RSpec 测试 Rails 应用

[美] Aaron Sumner 著

安道 译

rev 0.0.1, 2018-02-04T12:00:00+08:00

目录

本书是《Everyday Rails Testing with RSpec》的简体中文版，由 Aaron Sumner 授权翻译。

前言

感谢阅读《使用 RSpec 测试 Rails 应用（原书第 3 版）》。这一版耗时较久，变化很大。希望全新的内容不枉您的等候。

为什么耗时这么久呢？就像我说的，因为变化很大，变化的不仅是本书的内容，还有 Rails 的整体测试格局。先说后者。Rails 5.0 发布时，Rails 团队宣布不再支持直接测试控制器。这对我来说真是一个噩耗，要知道，本书前一版可是有整整三章专门讲解控制器测试的啊；不过，这还只是时间和精力上的损失，像我这样频繁在控制器层测试的人今后该怎么办啊？

一年后，Rails 5.1 发布了，我终于等到了定位同一层级的系统测试（system testing）。读过本书前几版的读者知道，我一直喜欢在这一层测试，而现在，相关的工具就内置在 Rails 框架中。虽然 Rails 用的不是 RSpec，而且与 RSpec 区别很大，但我依然感到欣慰，喜欢使用 Rails 自带测试工具的开发者终于可以在多个层级测试应用了。

与此同时，RSpec 也在演进，很多新特性能让测试更具表现力。很多人依旧喜欢 RSpec，不惜为此多走几步将其添加到应用中。

这些变化不在我的控制范围之内。下面说说我对本书内容所做的改动，这是我争取超越前几版所做的努力。很多地方是我一直想做的改动，与 Rails 和 RSpec 团队对框架本身所做的抉择无关。

本书的前身是我在 Everyday Rails 网站上发布的一系列博客文章。五年前，我开始记录自己学习测试 Rails 应用的经验，结果大受欢迎，因此我决定再添加一些内容和完整的示例代码，集结成书。本书出版后，销量完全超出了我的预想，而且帮助了众多与我当时面临相同困境的人们。

软件是个有趣的东西，关键就在这个“软”字上，它不是一成不变的。随着认识的深入，我们会调整解决问题的方法。我现在编写测试的知识依然源自原先那些博客文章和本书的第一版，但是随着时间的变迁，我掌握了一些新技术，摒弃了一些旧技术，还不断打磨可以沿用的技术。

过去的一年，我面临一个棘手的问题：如何把我掌握的技术体现到此次改版中，让新开发者能从中受益，根据自身的情况增补自己的知识，打造出适用于自己的一套测试方案？幸运的是，本书一直采用的学习策略仍然适用：从一个简单的应用入手，适当地在浏览器中测试（也可以使用 Postman 等测试 API）；从小处着手测试，不断积累，构建起复杂的测试组件。我们要习惯反向思维，先测试，再编写实现代码。随着时间的推移，建立自己的有效测试策略。

此外，我对本书前几版使用的示例应用不太满意。起初我的想法是尽量保持简单，让读者能记住来龙去脉，这样方便介绍测试概念；但正是因为太过简单了，有时反而会造成困扰：代码太少，覆盖度不够。所以此次改版，我下决心一章一章修改代码，然而即使我只应用最简单的改动，也会导致版本控制系统出现冲突。这一版的示例应用比以往要复杂一些（但还没复杂到难以理解），用起来更得心应手。这是我近些年来第一次没有先编写测试而开发出来的应用。

艰辛的改版过程就说到这里，我希望你能从本书中获益，不管你是刚接触测试，还是想知道我对测试驱动开发的观点，抑或是想了解自第一版以来测试方式的变迁。

我多次仔细检查过本书内容和代码，但你可能还会发现不太正确或与你所用不一致的方法。如果发现错误，或者有什么建议，欢迎到针对这一版的 GitHub issues 中反馈，我会尽快回应。

感谢所有读者，希望你们喜欢这一版，我也很希望在 GitHub、Twitter 和邮件中得到你们的反馈。

致谢

首先，感谢 *why the lucky stiff*，不管他现在何处。我是从他诡异而富有娱乐性的项目和书籍才接触 Ruby 的。还要感谢 Ryan Bates，我从他的网站 Railscasts 学到了太多的 Rails 知识。没有他们，Ruby 社区不会像现在这样活跃。

感谢 Ruby 社区中所有其他的牛人，虽然我没有见过你们，但你们的贡献让我变成一个更好的开发者，尽管我并不总会在代码中用到你们的创见。

感谢 Everyday Rails 博客的读者，你们给我那个 RSpec 系列文章提供了很多反馈，使我意识到可以把这些文章编成一本书。感谢所有购买了本书早期版本的人，我收到了很多反馈，对我帮助极大。

感谢 David Gnojek 审阅了我为本书设计的多个封面，并帮助我选择了一个较好的。你可以在 DESIGNOJEK 中查看 Dave 的绘作和设计。

感谢安道把这本书翻译成中文；感谢 Junichi Ito、Toshiharu Akimoto 和 Shinkou Gyo 把这本书翻译成日文。我很高兴能借由他们的努力接触更多的开发者。

感谢家人和朋友对本书的祝愿，他们甚至不知道我所讲述的是什么。

最后，感谢我的妻子，她忍受了我对新事物的痴魔，容忍我熬夜或整夜不睡。也感谢写作时常伴我左右的猫咪们。

关于封面

中文版封面所用图片由 Freepik 设计，特此感谢。

第 1 章 导言

Ruby on Rails 和自动化测试的结合十分紧密。Rails 内建了一个测试框架，生成器会自动创建测试样板文件，供你编写测试。是的，测试在 Rails 开发中真的很重要，不过有些人根本不测试，或者只为不痛不痒的功能编写一点象征性的测试。

在我看来，出现这种情况的原因很多。或许使用 Ruby 或 Web 框架已经算是很异类了，再多添加一层测试，就更脱离人民群众了。抑或是因为时间有限，编写测试会占用一定的时间，会延期实现客户或者老大要求的功能。又或者我们对测试的想法根深蒂固，认为只需在浏览器中点击一下链接就可以了。

这几种情况我都经历过。我不算是传统意义上的工程师，但和工程师一样，也有要解决的问题。这些问题往往会在开发软件的实践中才能找到解决办法。我从 1995 年开始开发 Web 应用，很长一段时间，我只是公务部门项目的独立开发者。我儿时接触过 BASIC，大学时学过一点 C++，毕业后开发第二个成熟的项目时浪费了一周学习 Java，除此之外，从未真正意义上学过软件开发。其实，直到 2005 年，受够了 PHP 那浆糊一般丑陋的代码时，我才找到一种开发 Web 应用更好的方式。

我以前接触过 Ruby，但是在 Rails 受到关注之前没怎么用过。要学的知识很多，一门新的编程语言，一个真正可用的架构，一个更加面向对象的解决方案（不管你怎么看待 Rails 对面向对象的处理，它确实比我没有使用框架之前所编写的代码都更加符合面向对象思想）。尽管有诸多困难，但与没有使用框架之前相比，开发复杂应用所用的时间还是减少了不少。我深深地被 Rails 吸引了。

早期的 Rails 书籍和教程太过追求开发速度（例如在 15 分钟内开发一个博客），而没有关注良好的开发习惯，比如测试。如果涉及到测试的话，往往也是在编完主要功能之后用一章单独介绍。最近的书籍和教程已经意识到了这个不足，会在开发应用的过程中贯穿讲解测试，甚至还有很多书是专门介绍测试的。但是如果没有完整的测试方案，很多开发者，特别是和曾经的我一样的人，会觉得没有系统地掌握测试策略。即便写了测试，测试也可能不可靠、无意义，完全无法让开发者建立自信。

写作这本书，我的首要目的是向你介绍我使用的系统性测试策略。你可以沿用这个策略，让测试变得系统化。我的成功经验能让你增强自信，即便以后修改代码也不怕，因为你知道有测试在背后支持，万一功能被破坏了，测试能让你知道。

1.1 为什么用 RSpec

首先要澄清的是，我对其他 Ruby 测试框架没有任何不满。如果开发一个独立的 Ruby 库，我通常会选择 MiniTest。不过，开发和测试 Rails 应用时，我一直坚持使用 RSpec。

或许是因为之前做过文案，也开发过软件，我觉得我看中的是 RSpec 不复杂而且阅读起来很顺畅的语法。本书后面的内容会更详细地介绍这一点。我发现即使是非技术人员，稍做培训，也能阅读使用 RSpec 编写的测试，而且能理解在测试什么。RSpec 的表达能力特别强，我发现用它描述期望软件所具有的行为已经成了我的第二天性，自然而然，句法就从我的指尖流出，以后即便修改软件，读起来也毫不费力。

本书的第二个目的是让你熟悉 RSpec 的常用功能和句法。RSpec 是个复杂的框架，但是与很多复杂的系统一样，80% 的情况下只会用到 20% 的功能。由此可见，这不是一本全面介绍 RSpec 及其辅助库（如 Capybara）的指南，本书的重点是介绍过去这些年我测试 Rails 应用的方法。本书还会介绍一些常用的

模式，万一以后你遇到书中没有涵盖的问题，便能充分发挥创造力，找到解决方案。

1.2 哪些人应该阅读本书

如果 Rails 是你使用的第一个 Web 应用框架，而且之前在编程中没有接触过测试，本书或许能带领你进入测试领域。如果你对 Rails 完全不熟悉，阅读本书之前最好过一遍 Michael Hartl 写的《Ruby on Rails Tutorial》[1]、Daniel Kehoe 写的《Learn Ruby on Rails》或 Sam Ruby 写的《Agile Web Development with Rails 5》[2]，因为本书的内容需要一些基本的 Rails 知识打底。也就是说，本书不会教你如何使用 Rails，也不会系统地介绍 Rails 内建的测试工具，我们只会安装 RSpec 和一些额外的工具，让测试的过程变得易于理解和管理。因此，如果你刚接触 Rails，先阅读前面给出的某本书，然后再看本书。

提示

附录 A 中给出了一些书籍、网站和测试教程的链接。

如果你已经用 Rails 一段时间了，但是对测试还很陌生，那么这门书就是为你而写的！我曾经也处在这一阶段，本书中分享的技术增强了我的测试技能，也让我更多地以测试驱动开发理念去思考问题。

具体来说，你应该具备以下知识：

- Rails 中用到的 MVC 架构
- 管理依赖的 Bundler
- 如何使用 Rails 命令行
- 在 Git 仓库中切换分支的方法

如果你能熟练使用 Test::Unit、MiniTest 或 RSpec，而且已经总结了一套适合自己的测试流程，那么只需做细微的调整就能测试应用了。

本书不涉及一般性的测试理论，也不深入探讨性能问题。如果你对这些话题感兴趣，可以翻阅附录 A 给出的书籍、网站和测试教程链接。

1.3 我的测试哲学

哪种测试最好，单元测试还是整合测试？我应该采用测试驱动开发（Test-driven Development，TDD）或行为驱动开发（Behavior-driven Development，BDD）理念吗（这二者之间又有何差异呢）？应该先写测试还是先写代码？测试真的有必要吗？

探讨何为测试 Rails 应用**正确**的方法可能会导致一场旷日之争，这样的争辩虽不像“Mac vs. PC”或“Vim vs. Emacs”那么激烈，但当你和其他 Ruby 用户谈起时，氛围也不会好到哪儿去。此外，David Heinemeier-Hansen 在 Railsconf 2014 上所做的主题演讲声称 TDD“已死”，这又引起了新一轮纷争。

其实，测试真的有正确的方法，如果让我说，**正确**的测试可以分为好几个层次。测试时我关注的是以下这几个基本点：

---

1. 《Ruby on Rails Tutorial》已由本书译者翻译成中文，详情请访问 https://railstutorial-china.org。
2. 《Agile Web Development with Rails 5》已由本书译者翻译成中文，详情请访问 https://about.ac/books/agile-rails5/。

- 测试要可靠；
- 测试要易于编写；
- 测试要易于理解——当下和以后都是如此。

总之，测试应该给软件开发者自信。如果考虑这三个因素，在为应用编写较为全面的测试组件时会遇到很多困难，要成为真正的测试驱动开发践行者更是难上加难。

当然了，一般来说我们可以采取一些折中方案：

- 不关注效率（不过，稍后我们还是会讨论这个话题）
- 不过度考虑代码重复的问题，测试中的重复不一定是坏事（我们也会讨论如何消除代码重复）

最终，我们得到的是一些可靠且容易理解的测试，虽然还有很多地方可以优化，但这却是一个好的开始。以前，我会编写很多应用代码，然后在浏览器中到处点击链接，祈祷一切都能正常运行。采用上述测试方法后，我充分利用了自动化测试的长处，使用测试驱动开发的理念，捕获了很多潜在问题和边缘情况。

本书的目的就是讲解我所使用的测试方法。

1.4 内容结构

在本书中我会向你展示如何使用 RSpec 为一个完全没有测试的 Rails 应用编写合适的测试组件。本书涵盖 Rails 5.1 和 RSpec 3.6，这是撰写这一版时二者的最新版。

本书由以下几章组成：

- 你正在阅读的是第 1 章，导言；
- 第 2 章设置全新的或者现存的 Rails 应用，让应用使用 RSpec；
- 第 3 章为模型编写简单可靠的单元测试（unit test）；
- 第 4 章介绍生成测试数据的方法；
- 第 5 章直接测试应用的控制器；
- 第 6 章介绍通过功能测试实现的整合测试，测试应用中不同部分之间的交互；
- 第 7 章介绍如何绕过传统的 Web 用户界面，直接测试应用的可编程接口；
- 第 8 章说明何时以及如何去除测试中的重复，而何时可以保留重复；
- 第 9 章介绍一些提升测试编写速度的技术，以此提升测试的反馈速度；
- 第 10 章介绍如何测试前几章没有涉及的部分，例如电子邮件、文件上传，以及外部服务；
- 我会在第 11 章中分步演示测试驱动开发的流程；
- 最后，在第 12 章中总结全书。

每一章都详细分析了我在测试时使用的方法。很多章都有思考部分，让你深入探索测试的方法和缘由；随后还有一些练习，让你把所介绍的技术运用到自己的应用中。再次强调，我强烈建议你完成这些练习，在自己的应用中练练手，因为本书不会开发一个完整的应用，只是用到一些代码片段和技术。把书中介绍的技术运用到自己的项目中，才能让应用变得更好。

1.5 下载示例代码

我在 GitHub 中提供了一个全面测试过的应用。

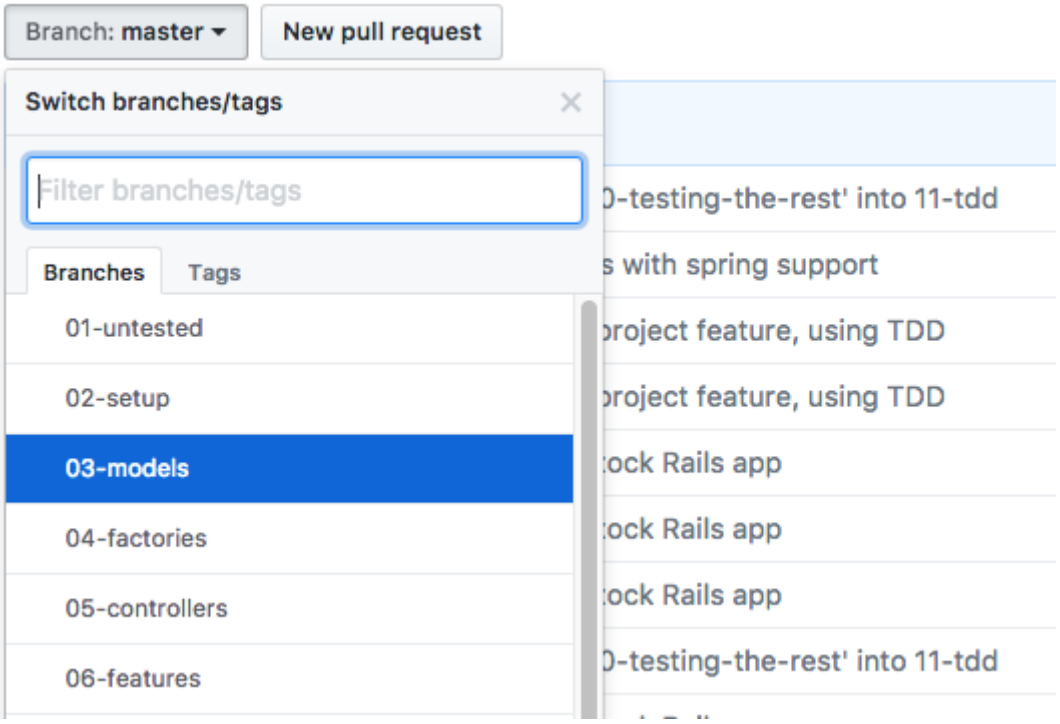
> **获取源码**
>
> https://github.com/everydayrails/everydayrails-rspec-2017

如果你熟悉 Git 操作（作为 Rails 开发者，应该熟悉），可以把源码克隆到本地电脑。每一章所用的代码都放在单独的分支中。如果想看完成后的代码，获取对应章节的分支就可以；如果想跟着内容一步步走，请获取前一章的分支。分支的名称按章序编号，每一章开头我都会告诉你应该切换到哪个分支。

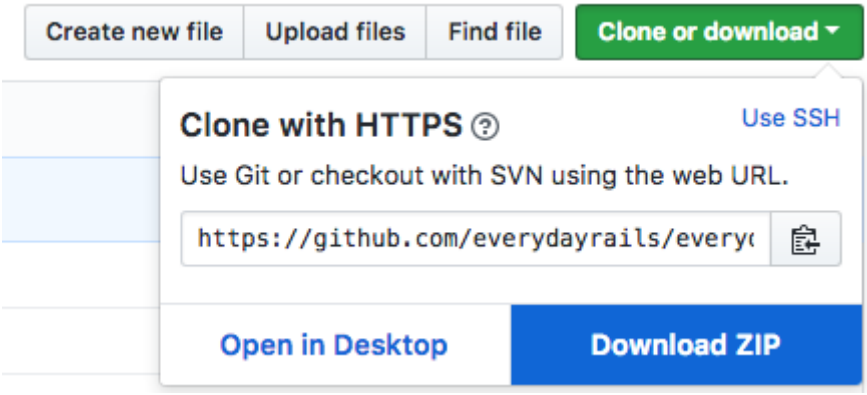
本书在内容编排上是一章连着一章的，后一章以前一章为基础。因此，开始阅读一章时，可以使用前一章的分支。例如，如果我想跟着第 5 章的内容一点点编写代码，可以在第 4 章的基础上开始：

```
$ git checkout -b my-05-controllers origin/04-factories
```

如果不熟悉 Git 操作也可以获取各章所用的示例代码。打开本书在 GitHub 上的页面，找到分支选择菜单，选择相应章节的分支：



然后点击“Download ZIP”按钮把源码下载到电脑中：



提示

Git Immersion 很不错，通过自己动手在命令行中操作，学习 Git 基本知识。Try Git 也不错。

1.6 代码约定

本书中的应用使用下面这些程序：

- **Rails 5.1**：本书重点关注的是 Rails 的最新版本，但据我所知，书中介绍的技术可以在 Rails 3.0 之后的任一版本中使用。如果你没有使用和我一样的版本，可能要适当修改书中用到的示例代码。
- **Ruby 2.4**：Rails 5.0 要求至少使用 Ruby 2.2。如果你的应用是使用旧版 Rails 开发的，使用 Ruby 2.0、2.1 或 2.2 应该都没问题。
- **RSpec 3.6**：2014 年春天，RSpec 3.0 发布了。RSpec 3.6 与 Rails 5.1 同期发布，基本上和 3.0 没有太大的区别。新版使用的句法和 RSpec 2.14 很像，不过也有些区别。

如果某些内容只针对以上所列的版本，我会尽量指出。如果你使用的是较早前的版本，可以参照本书针对 Rails 3.2 的源码。两个版本的源码之间功能虽不是一一对应的，但你或许能够发现一些明显的不同。

再次强调一下，**本书不是常规的教程**！书中的代码不是教你如何开发应用的，而是帮助你学习并理解测试模式和好习惯的，以便于在你自己编写的应用中使用。你可以直接复制粘贴，但这样就什么也学不到了。你可能很熟悉 Zed Shaw 在“笨方法学习系列”中采用的学习方式，本书没有照搬这种模式，但我很认同 Zed 的观点，自己敲出来的代码比从网上或电子书中复制粘贴更利于掌握知识。

1.7 讨论和勘误

我花了大量时间和精力，尽量确保本书没有错误，但你可能还是会发现我没有注意到的问题。如果你发现了问题，可以到本书源码所在的 GitHub 页面的 issues 区报告，或者请求我进一步说明某部分内容，地址是：https://github.com/everydayrails/everydayrails-rspec-2017/issues。

如果在阅读中文版的过程中遇到问题，或者有建议和勘误，可以直接通过微信与译者联系。

1.8 关于 gem 版本

本书及示例应用中使用的 gem 版本是 2017 年春天写作这一版时的最新版。当然，有些 gem 更新频繁，你可以留意 Rubygems.org、GitHub 和你喜欢的 Ruby 新闻订阅源，关注更新情况。

1.9 关于示例应用

本书中用到的示例应用是一个项目管理应用。虽然不像 Trello 或 Basecamp 那么完善，但是足以用于介绍测试技术。

这个应用支持下述功能：

- 用户可以添加项目，而且只对自己可见；
- 用户可以在项目中添加任务、记录和附件；
- 用户可以把任务标记为“已完成”；
- 用户的账户有个头像，由 Gravatar 服务提供；
- 开发者可以利用公开的 API 开发客户端应用。

在开发示例应用时我偷了懒，所有的代码都是用 Rails 的生成器生成的（参见 01-untested 分支中的示例代码），所以应用中有一个名为 *test* 的文件夹，里面的测试文件和固件我都没动过。此时执行 `bin/rails test` 命令，可能其中某些测试是可以通过的。既然这本书是介绍 RSpec 的，最好把这个文件夹删除，让 Rails 使用 RSpec，然后编写一个可靠的测试组件——本书会告诉你怎么做。

现下首先要做的事是配置应用，让它使用 RSpec。现在就开始吧。

第 2 章 安装 RSpec

我在第 1 章说过，本书附带的项目管理应用是可以使用的，至少我们**认为**如此，因为我们到处点击了应用中的链接，创建了一些示例账户和项目，还通过 Web 浏览器添加并编辑了数据。当然，这个过程随着功能的增加将变得异常繁琐。为应用添加新功能之前，我们要改变这个局面，使用 RSpec 和一些其他程序实现**自动化测试**。在接下来的几章中，我们将为应用添加足量的测试，一开始只使用 RSpec，需要时再补充其他测试库。

首先，我们要安装 RSpec，并配置应用，让应用使用它测试。以前，如果想在 Rails 中使用 RSpec，要做些复杂的设置才行，现在情况不同了，但还是要安装一些程序，并且要做一些配置。

本章将完成以下任务：

- 先使用 Bundler 安装 RSpec；
- 检测是否要安装“测试数据库”；
- 接下来我们要配置 RSpec，这样才能测试我们想测试的功能；
- 最后，我们要配置 Rails，当添加新功能时自动生成测试相关的文件。

提示

本章所有的代码改动可以在 GitHub 上通过一次差异查看。

如果你想跟着我一起做，参照第 1 章的说明，克隆仓库，然后检出前一章的分支：

```
$ git checkout -b my-02-setup origin/01-untested
```

2.1 Gemfile

既然 Rails 应用默认不提供 RSpec，那我们就花点时间自己安装吧。我们将使用 Bundler 安装这些依赖库。请打开应用的 *Gemfile* 文件，添加 RSpec：

Gemfile

```
group :development, :test do
  gem 'rspec-rails', '~> 3.6.0'
  # 省略了 Rails 提供的其他 gem
end
```

注意

如果你是在现有项目中添加 RSpec，看到的代码可能跟这里不同。不过只需记住，`rspec-rails` 依赖必须能加载到 Rails 开发环境和测试环境中。也就是说，我们无需在服务器中运行测试。

严格来说，我们安装的是 `rspec-rails` 库，它包含 `rspec-core` 和一些其他 gem。如果想使用 RSpec 测试

Sinatra 应用或 Rails 之外的其他 Ruby 应用，可能要单独安装各个 gem。`rspec-rails` 把相关的 gem 打包在一起，便于安装，还提供了一些专门针对 Rails 的便利措施（后文说明）。

在命令行中执行 `bundle` 命令，把 `rspec-rails` 及其依赖安装到你的系统中。至此，我们向可靠的测试组件迈出了必要的第一步。接下来要创建测试数据库。

2.2 测试数据库

如果是为现有的 Rails 应用添加测试，你的电脑中很可能已经有测试数据库了。如果没有，请参照下面的方法搭建。

打开 *config/database.yml* 文件，看看你的应用使用的是哪种数据库。如果你没有修改过这个文件，应该会看到下面针对 SQLite 的设置：

config/database.yml

```
test:
      <<: *default
      database: db/test.sqlite3
```

如果使用 MySQL 或 PostgreSQL 的话，看到的内容如下：

config/database.yml

```
test:
      <<: *default
      database: projects_test
```

如果没看到类似上面的内容，请把相应的代码加进去，并把 `projects_test` 换成你自己的应用所用的测试数据库的名称。

提示

如果你的数据库配置与这里不同，请阅读 Rails 指南中的“配置 Rails 应用”一章了解详情。

最后，为了确保能与数据库通信，执行下述命令：

```
$ bin/rails db:create:all
```

注意

在 Rails 5.0 之前，上述命令要写成 `bin/rake db:create:all`。如果使用的是 Rails 4.1 之前的版本，要把 `bin/rake` 换成 `bundle exec rake`。本书假定你使用的是 Rails 5.0 或以上版本，如若不然，执行创建数据库或迁移数据库模式等任务时只需把 `bin/rails` 替换成 `bin/rake` 或 `bundle exec rake`。使用生成器创建新文件时，使用 `bin/rails` 或 `bundle exec rails` 命令。

如果之前没有测试数据库，执行这个任务后就搭建好了；如果已经搭建了，这个任务会友好地提示数据库已经存在，所以你不必担心会把之前的数据库删掉。接下来配置 RSpec。

2.3 配置 RSpec

现在，我们可以在应用中添加一个 *spec* 文件夹了，同时还要对 RSpec 做些基本的配置。我们要使用下面的命令安装 RSpec：

```
$ bin/rails generate rspec:install
```

执行这个命令后，会输出如下内容：

```
Running via Spring preloader in process 28211
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

我们得到了 RSpec 配置文件（*.rspec*）、用于存放测试文件的文件夹（*spec*），以及两个保存辅助方法的文件（*spec/spec_helper.rb* 和 *spec/rails_helper.rb*）。我们可以在这两个辅助方法文件中定制如何与应用的代码交互。现在没必要通读这两个文件，不过，既然 RSpec 是 Rails 工具集中的一部分，我强烈建议你找个时间读一下，试验一下不同的设置，这样才能理解各项设置的作用。

下面这一步设置是可选的。我喜欢把 RSpec 默认的输出格式换成更便于阅读的**文档**格式。使用文档格式可以清晰地看出哪些测试通过了、哪些失败了，而且还会生成一个条理清晰的测试用例大纲，可作为文档使用。打开 *.rspec* 文件，把内容改成下面这样：

.rspec

```
--require spec_helper
--format documentation
```

此外，还可以在这个文件中添加 `--warnings` 旗标；此时，RSpec 会输出应用及其所用 gem 抛出的全部提醒。在真实的应用中启用这个选项很有用，因为我们要关注测试给出的功能弃用提醒，但现在我们的目的是学习测试，关闭提醒可以减少测试结果的输出量。如果以后需要，随时可以启用。

2.3.1 使用 `rspec binstub` 提升测试组件的启动速度

接下来，为 RSpec 测试运行程序安装 binstub，利用 Spring 提升应用的启动速度。这样，如果应用已经在开发服务器中启动，或者通过一个 Rake 任务启动过了，测试组件启动的速度更快，因为应用已在运行中。如果出于什么原因，你不想使用 Spring，跳过这一节就行，不过要把本书使用的 `bin/rspec` 命令换成 `bundle exec rspec`。

在 *Gemfile* 文件中的 `:development` 分组中添加下述依赖：

Gemfile

```
group :development do
  # 这一分组中的其他 gem ...
  gem 'spring-commands-rspec'
end
```

然后运行 `bundle` 安装这个新依赖，再生成所需的 binstub：

```
$ bundle exec spring binstub rspec
```

这个命令会在应用的 *bin* 目录里生成可执行文件 *rspec*。

2.4 试一试

虽然现在还没有测试，不过依然可以检查 RSpec 安装地是否正确。使用刚刚创建的 binstub 运行 RSpec：

```
$ bin/rspec
```

如果一切正常，应该看到类似下面的输出：

```
Running via Spring preloader in process 28279
No examples found.

Finished in 0.00074 seconds (files took 0.14443 seconds to load)
0 examples, 0 failures
```

如果你没看到这样的输出，回头看看前面的内容，确保你按照每一步操作了。别忘了在 *Gemfile* 中添加依赖，也别忘了执行 `bundle` 命令。

2.5 生成器

这一步告诉 Rails，使用内置的生成器生成代码时，生成 RSpec 所需的测试文件。

安装 RSpec 后，Rails 的生成器不会在 *test* 文件夹中生成 Test::Unit 测试文件了，而是在 *spec* 文件夹中生成 RSpec 测试文件（生成的预构件存储在 *spec/factories* 文件夹中）。不过，如果需要，还可以对生成器做些设置。例如，如果使用 `scaffold` 生成器生成应用代码的话，可能就想自定义一下设置，因为默认的生成器会生成很多测试文件，而本书不会做介绍，因此要修改默认设置，不生成多余的文件。

打开 *config/application.rb* 文件，在 `Application` 类中加入下面的代码：

config/application.rb

```
require_relative 'boot'
require 'rails/all'

Bundler.require(*Rails.groups)

module Projects
  class Application < Rails::Application
    config.load_defaults 5.1

    # Rails 生成的多行注释...

    config.generators do |g|
      g.test_framework :rspec,
        fixtures: false,
        view_specs: false,
        helper_specs: false,
        routing_specs: false
    end
```

```
    end
  end
```

你能猜到这些代码的作用吗？下面就来分析一下：

- `fixtures: false` 禁止生成便于在测试数据库中创建对象的文件。第 4 章开始使用预构件时再把这个选项改为 `true`。
- `view_specs: false` 的意思是不生成视图测试。本书不会介绍视图测试，测试界面元素我使用的是功能测试。
- `helper_specs: false` 的意思是生成控制器时不生成对应的辅助方法测试文件。如果你觉得有必要，可以把这个选项设为 `true`，对辅助方法进行测试。
- `routing_specs: false` 的意思是不生成针对 `config/routes.rb` 的测试文件。如果应用很简单，比如本书使用的示例应用，可以放心跳过路由测试。不过，如果是大型应用，路由很复杂，最好还是测试一下。

默认会生成模型和控制器的测试文件，如果不想自动生成，可以在这个配置块中指明。例如，若想跳过控制器测试，加上 `controller_specs: false`。

记住，RSpec 不生成某些文件，但是你可以手动创建，也可以删除不用的文件。假如需要一个辅助方法测试文件，可以在 *spec/helpers* 文件夹中新建，文件名要按照测试文件的命名约定。所以，如果要测试 *app/helpers/projects_helper.rb* 文件，建立的测试文件就是 *spec/helpers/projects_helper_spec.rb*；如果要测试 *lib/my_library.rb*，建立的测试文件就是 *spec/lib/my_library_spec.rb*；依此类推。

2.6 小结

本章，我们把 RSpec 作为一个依赖添加到应用的开发和测试环境中，还配置了一个专门用于测试的数据库。我们还创建了 RSpec 配置文件，并配置了 Rails 生成器要生成和不要生成的文件。

现在可以开始编写测试了！下一章从模型层开始，测试示例应用的功能。

2.7 问答

我可以删除 *test* 文件夹吗？

如果你使用的是新生成的应用，当然可以。如果你已经开发一段时间了，请先执行 `bin/rails test`，看一下这个文件夹中的测试有没有必要转到 RSpec 的 *spec* 文件夹中。

你为什么不测试视图呢？

编写可靠的视图测试是很困难的事，维护就更不用说了。我在设置生成器时说过，UI 相关的测试交给集成测试。这是 Rails 开发者普遍使用的方式。

2.8 练习

如果你使用的是现有的应用：

- 把 `rspec-rails` 添加到 *Gemfile* 文件中，使用 `bundle` 安装。虽然本书针对的是 Rails 5.1 和 RSpec 3.6，但是多数测试代码和技术应该也适用于旧版。

- 确保你的应用配置得当，可以与测试数据库通信。如果需要，请搭建测试数据库。
- 设置一下 Rails 的生成器，以后再为应用添加功能时使用 RSpec 生成测试相关的文件。当然，你也可以使用 `rspec-rails` 提供的默认设置；不过这样会生成额外的样板代码，你可以自己动手删除，也可以放着不管。（我建议删除用不到的代码。）
- 做一个清单，列出你的应用现在需要测试的功能有哪些，包括核心功能、以前遇到过要修正的问题、破坏现有功能的新功能，以及边缘情况。后面的章节会一一介绍这些情况。

如果你使用的是新建的应用：

- 按照前面介绍的方法，使用 Bundler 安装 RSpec。
- *database.yml* 文件中应该已经设置好了测试数据库，如果你不想使用 SQLite，可能要自己创建一个，请执行 `bin/rails db:create:all`。
- （选做）配置 Rails 的生成器，当添加新的模型和控制器时使用 RSpec 生成所需的测试文件。

附加题

如果你要经常新建 Rails 应用，可以创建一个 Rails 应用模板，自动把 RSpec 和相关的配置添加到应用的 *Gemfile* 和配置文件中，而且还可以自动创建测试数据库。Daniel Kehoe 开发的 Rails Composer 是个不错的工具。

第 3 章 模型测试

我们已经安装了编写稳固可靠测试所需的工具，下面可以使用这些工具来编写测试了。我们先从应用的核心部分——模型——入手。

本章将完成以下任务：

- 首先为现有模型编写测试；
- 然后为模型的数据验证、类方法和实例方法编写测试，并在这个过程中介绍如何合理地组织测试代码。

我们将自己动手创建针对现有模型的测试文件。以后添加新模型时，因为在第 2 章中做了设置，RSpec 生成器会自动生成所需的文件。

提示

本章所有的代码改动可以在 GitHub 上通过一次差异查看。

如果你想跟着我一起做，参照第 1 章的说明，克隆仓库，然后检出前一章的分支：

```
$ git checkout -b my-03-models origin/02-setup
```

3.1 模型测试剖析

我觉得在模型层中学习测试是最简单的，因为在模型中你可以检验应用的核心功能是否运作良好。模型层中精心测试的代码是应用开发的关键，打好这层基础，上层的代码才能可靠。

模型测试应该包含：

- 使用有效属性实例化的模型应该是有效的；
- 无法通过数据验证的数据，测试应该失败；
- 类方法和实例方法应按预期可正常使用。

现在最好来看一下 RSpec 模型测试的基本结构。我发现，可以把测试看成是一个大纲，这样会给你带来不少帮助。例如，下面是对 `User` 模型的要求：

```
describe User do
  it "is valid with a first name, last name, email, and password"
  it "is invalid without a first name"
  it "is invalid without a last name"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a user's full name as a string"
end
```

稍后我们会实现这里列出的测试用例，这个大纲对初学者来说可能信息量有点大。这是一个针对简单模型所编写的简单测试，但却体现了三个最佳实践：

- 描述了一系列期望的表现，也就是 `User` 模型能做哪些事情；
- 一个测试用例（以 `it` 开头的行）只检测一件事。注意，我分别单独测试了 `first_name`、`last_name` 和 `email` 能否通过验证。这么做如果测试失败了，我们就能知道到底是哪个属性没有通过验证，而不用排查 RSpec 的输出，至少不用很深入。
- 每个测试用例的目的都很明确。`it` 后面的描述在 RSpec 中其实是可以省略的，不过省略的话测试代码读起来就不太顺口了。
- 每个测试用例的描述都以动词开头，而没用“should（应该）”。请大声读出这些期望的表现：“User is invalid without a first name”“User is invalid without a last name”“User returns a user’s full name as a string”。我们关注的是可读性。这也是 RSpec 的一大特色。

请牢记这些最佳实践，下面我们来编写 `User` 模型的测试。

3.2 编写模型测试

在第 2 章我们配置过 RSpec，让它在添加模型和控制器时自动生成样板测试文件。不过，生成器随时都可以调用。现在，我们就使用一个生成器为我们的第一个模型测试生成样板文件。

在命令行中调用 `rspec:model` 生成器：

```
$ bin/rails g rspec:model user
```

RSpec 报告，新建了一个文件：

```
Running via Spring preloader in process 32008
      create  spec/models/user_spec.rb
```

打开这个文件，看看里面的内容：

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

借由这个文件，我们第一次见识到了 RSpec 的句法和约定。首先，导入 *rails_helper* 文件；测试组件中的每个文件几乎都要这么做。这么做的目的是告诉 RSpec，为了运行文件中的测试，要加载 Rails 应用。然后，通过 `describe` 方法列出我们对 `User` 模型行为的期待。第 11 章开始实践测试驱动开发时再深入说明 `pending`。现在，如果使用 `bin/rspec` 命令运行这个文件中的测试会得到什么结果呢？

```
Running via Spring preloader in process 41653

User
  add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
  (PENDING: Not yet implemented)
```

```
Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
     # Not yet implemented
     # ./spec/models/user_spec.rb:4


Finished in 0.00107 seconds (files took 0.43352 seconds to load)
1 example, 0 failures, 1 pending
```

注意

RSpec 测试文件不一定非得使用生成器创建，但是这么做能避免由于错别字而导致的简单错误。

保留外层的 `describe`，把里面的内容替换成前面给出的大纲：

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with a first name, last name, email, and password"
  it "is invalid without a first name"
  it "is invalid without a last name"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a user's full name as a string"
end
```

稍后我们会编写具体的测试代码，如果现在在命令行中运行测试的话（执行 `bin/rspec` 命令），应该会得到类似下面的输出：

```
Running via Spring preloader in process 32556

User
  is valid with a first name, last name, email, and password (PENDING:
  Not yet implemented)
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User is valid with a first name, last name, email, and password
     # Not yet implemented
     # ./spec/models/user_spec.rb:4
```

```
  2) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:5

  3) User is invalid without a last name
     # Not yet implemented
     # ./spec/models/user_spec.rb:6

  4) User is invalid without an email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:7

  5) User is invalid with a duplicate email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:8

  6) User returns a user's full name as a string
     # Not yet implemented
     # ./spec/models/user_spec.rb:9


Finished in 0.00176 seconds (files took 2.18 seconds to load)
6 examples, 0 failures, 6 pending
```

很好，有 6 个待实现（pending）的测试。RSpec 之所以把它们标记为“待实现”，是因为我们尚未编写具体的测试代码。下面就开始着手编写第一个测试用例。

注意

以前，要通过一个 Rake 任务把开发数据库的结构复制到测试数据库中。现在，多数情况下，运行迁移时 Rails 会自动复制。如果在测试环境中遇到报错，说缺少迁移，可以运行 `bin/rails db:migrate RAILS_ENV=test` 命令更新迁移。

3.3 RSpec 的句法

2012 年，就在我发布本书第一个完整版后的几天，RSpec 开发团队宣布，2.11 版使用新句法代替了传统的 `should` 式句法。新句法可能让人不太适应。

新句法削弱了 `should` 引起的技术问题。现在我们不说某物应该（`should`）或不应该（`should_not`）符合期望的输出，而是说期望某物是（`to`）或者不是（`not_to`）什么。

举个例子，请看下面这个测试用例。我们知道 2 + 1 肯定等于 3，对吧？用 RSpec 来表达，使用旧句法可以这么写：

```
it "adds 2 and 1 to make 3" do
  (2 + 1).should eq 3
end
```

新句法把要测试的值传递给 `expect()` 方法，然后与匹配器比较：

```
  it "adds 2 and 1 to make 3" do
    expect(2 + 1).to eq 3
  end
```

在 Google 或 Stack Overflow 中搜索 RSpec 相关的问题，或者维护旧的 Rails 应用时依然能见到 `should` 式句法。这种旧句法在 RSpec 的当前版本中依旧可用，不过会看到弃用警告。你可以做些配置，让 RSpec 关闭警告，但是早晚都要转变，何不早点学习新的 `expect()` 句法呢。

那么这种新句法在真实的测试中应该怎么使用呢？让我们来编写 `User` 模型测试中的第一个测试用例：

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with a first name, last name, email, and password" do
    user = User.new(
      first_name: "Aaron",
      last_name:  "Sumner",
      email:      "tester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )
    expect(user).to be_valid
  end

  it "is invalid without a first name"
  it "is invalid without a last name"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a user's full name as a string"
end
```

这个简单的测试用例使用了 RSpec 中的 `be_valid` 匹配器，检测模型是否能够检测数据的有效性。我们创建了一个对象（`User` 模型的实例 `user`，但没有存入数据库），传给 `expect` 方法，与匹配器比较。

现在如果在命令行中执行 `bin/rspec` 命令，会发现有一个测试用例通过了：

```
Running via Spring preloader in process 32678

User
  is valid with a first name, last name and email, and password
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:14
```

```
  2) User is invalid without a last name
     # Not yet implemented
     # ./spec/models/user_spec.rb:15

  3) User is invalid without an email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:16

  4) User is invalid with a duplicate email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:17

  5) User returns a user's full name as a string
     # Not yet implemented
     # ./spec/models/user_spec.rb:18


Finished in 0.02839 seconds (files took 0.28886 seconds to load)
6 examples, 0 failures, 5 pending
```

很好，第一个测试编写完毕！下面接着编写测试，一个个消灭待实现的测试。

3.4 测试数据验证

针对数据验证的测试是自动化测试很好的演示。数据验证的测试代码一般也就一到两行。我们来看一下 `first_name` 数据验证的测试：

spec/models/user_spec.rb

```
it "is invalid without a first name" do
  user = User.new(first_name: nil)
  user.valid?
  expect(user.errors[:first_name]).to include("can't be blank")
end
```

这一次我们期望新建的用户（把 `first_name` 设为了 `nil`）是无效的，会得到一个关于 `first_name` 属性的错误消息。我们使用 RSpec 提供的 `include` 匹配器检查一个可枚举的值中是否包含指定的值。再次运行 RSpec，应该看到有两个测试通过了。

目前我们采用的方法有个小问题。我们设法让几个测试通过了，但是从未见过它们失败。这可能就是一个信号，尤其是对刚接触测试的你而言。我们要确保测试代码的确做了该做的事，也就是让代码在测试的监管之下。

有几种方法能保证测试没有误检，例如把 `to` 换成 `to_not`：

spec/models/user_spec.rb

```
it "is invalid without a first name" do
  user = User.new(first_name: nil)
  user.valid?
  expect(user.errors[:first_name]).to_not include("can't be blank")
```

```
  end
```

此时，RSpec 肯定会报告有测试失败：

```
Failures:

  1) User is invalid without a first name
     Failure/Error: expect(user.errors[:first_name]).to_not
     include("can't be blank")
       expected ["can't be blank"] not to include "can't be blank"
     # ./spec/models/user_spec.rb:17:in `block (2 levels) in <top \
      (required)>'
     # /Users/asumner/.rvm/gems/ruby-2.4.1/gems/spring-commands-rspec-\
      1.0.4/lib/spring/commands/rspec.rb:18:in `call'
     # -e:1:in `<main>'

Finished in 0.06211 seconds (files took 0.28541 seconds to load)
6 examples, 1 failure, 4 pending

Failed examples:

rspec ./spec/models/user_spec.rb:14 # User is invalid without a first name
```

提示

RSpec 为这种否定预期提供了 `to_not` 和 `not_to` 两个匹配器，二者是可以互换的。本书使用的是 `to_not`，因为 RSpec 文档中经常用它。

此外，还可以修改应用代码，看看对测试有什么影响。把刚才对测试的改动改回去（把 `to_not` 改成 `to`），然后打开 `User` 模型，把 `first_name` 的验证注释掉：

app/models/user.rb

```
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # validates :first_name, presence: true
  validates :last_name, presence: true

  # 省略其他内容 ...
```

再次运行测试，应该还会看到一个失败。我们告诉 RSpec，没有名字的用户是无效的，而应用的代码却没有这样规定。

这两种方法能轻易确认测试能按预期工作，尤其是从简到繁的过程中，或者测试现有代码时。如果测试的输出没有变化，很有可能是测试没有与代码对接上，或者代码的行为与预期的不同。

下面用类似的方式编写针对 `:last_name` 数据验证的测试。

spec/models/user_spec.rb

```
  it "is invalid without a last name" do
    user = User.new(last_name: nil)
    user.valid?
    expect(user.errors[:last_name]).to include("can't be blank")
  end
```

你可能觉得这些测试没什么意义，“我怎么会轻易忘记在模型中加入数据验证呢？”其实，这种可能性比你想的要大。而且，编写测试时设想模型中应该加入哪些数据验证（理想情况下，最终你会养成“测试驱动开发”思维），你就更有可能记得把这些验证加入模型中。

下面在此基础上编写一个稍微复杂的测试。这一次，我们要检查 `email` 属性的唯一性验证：

spec/models/user_spec.rb

```
  it "is invalid with a duplicate email address" do
    User.create(
      first_name:  "Joe",
      last_name:  "Tester",
      email:      "tester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )
    user = User.new(
      first_name:  "Jane",
      last_name:  "Tester",
      email:      "tester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )
    user.valid?
    expect(user.errors[:email]).to include("has already been taken")
  end
```

注意，这段代码中有一处和之前不一样：我们保存了一个联系人（调用了 `create` 方法，而没用 `new` 方法），作为要比较的对象，然后又新建了另一个联系人，作为测试的对象。当然，保存的联系人一定要是有效的（有姓有名，有电子邮件和密码），而且必须使用相同的电子邮件地址。第 4 章会介绍简化这个过程的方法。现在，执行 `bin/rspec` 命令看一下测试的输出。

下面我们来测试一个稍微复杂的数据验证。我们将暂时离开 `User` 模型，转向 `Project` 模型。假设我们要确保同一用户的两个项目不能具有相同的名称，即同一用户名下的项目名称要是唯一的。比如说，我不能建立两个名为“粉刷屋子”的项目，而你和我各自却可以都有名为“粉刷屋子”的项目。这样的测试应该怎么写呢？

先为 `Project` 模型新建一个测试文件：

```
$ bin/rails g rspec:model project
```

然后在新建的文件中添加两个用例。我们将测试一个用户不能有两个同名的项目，但是不同的用户却可以有同名的项目。

```
require 'rails_helper'

RSpec.describe Project, type: :model do
  it "does not allow duplicate project names per user" do
    user = User.create(
```

```
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    user.projects.create(
      name: "Test Project",
    )

    new_project = user.projects.build(
      name: "Test Project",
    )

    new_project.valid?
    expect(new_project.errors[:name]).to include("has already been taken")
  end

  it "allows two users to share a project name" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    user.projects.create(
      name: "Test Project",
    )

    other_user = User.create(
      first_name: "Jane",
      last_name:  "Tester",
      email:      "janetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    other_project = other_user.projects.build(
      name: "Test Project",
    )

    expect(other_project).to be_valid
  end
end
```

这里，因为 `User` 和 `Project` 模型是通过 Active Record 关联的，所以要提供一些额外的信息。在第一个测试用例中，我们把两个项目赋予同一个用户。在第二个测试用例中，两个项目使用同一个名称，但是项目分属不同的用户。注意，在两个测试用例中，我们都新建了用户，并将其保存到数据库中，这样才能把项目赋予他们。

因为 `Project` 模型有下面这个数据验证：

app/models/project.rb

```
validates :name, presence: true, uniqueness: { scope: :user_id }
```

所以这个测试应该是可以通过的。别忘了检查一下测试编写地是否正确，把验证代码临时注释掉，或者把测试的预期改成其他值试试，改动之后测试会失败吗？

当然，带作用域的验证还算是简单的。很多测试可能会用到正则表达式或自定义的验证方法。你要养成测试数据验证的习惯，不仅要测试能通过验证的情况，还要测试不能通过验证的情况。比如，在针对本书使用的示例应用所做的测试中，我们就检测了当使用 `nil` 初始化对象时会发生什么事情。如果有个验证是确保属性的值必须是数字，提供一个字符串试试。如果有个验证要求字符串的长度在 4~8 个字符之间，提供 3 个或 9 个字符的字符串试试。

3.5 测试实例方法

下面接着测试 `User` 模型。在这个应用中，每次都要手动连接姓和名字生成用户的姓名有点麻烦，如果能直接调用 `@user.name` 这样的方法就方便多了，所以我们在 `User` 类中定义了这个方法：

app/models/user.rb

```
def name
  [firstname, lastname].join(' ')
end
```

我们可以参照上面用到的技术，编写针对这个方法的测试：

spec/models/user_spec.rb

```
it "returns a user's full name as a string" do
  user = User.new(
    first_name: "John",
    last_name:  "Doe",
    email:      "johndoe@example.com",
  )
  expect(user.name).to eq "John Doe"
end
```

注意

测试相等时，RSpec 推荐使用 `eq` 或 `eql`，而不是 `==`。

我们先创建测试数据，然后告诉 RSpec 期望得到什么结果。很简单吧，继续往下看。

3.6 测试类方法和作用域

我们实现了基本的搜索功能。为了演示方便，目前的实现方式是在 `Note` 模型中定义的一个作用域：

app/models/note.rb

```
scope :search, ->(term) {
  where("LOWER(message) LIKE ?", "%#{term.downcase}%")
}
```

使用 rspec:model 生成器在不断壮大的测试组件中为 Note 模型添加一个测试文件，然后编写下述测试：

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  it "returns notes that match the search term" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    project = user.projects.create(
      name: "Test Project",
    )

    note1 = project.notes.create(
      message: "This is the first note.",
      user: user,
    )
    note2 = project.notes.create(
      message: "This is the second note.",
      user: user,
    )
    note3 = project.notes.create(
      message: "First, preheat the oven.",
      user: user,
    )

    expect(Note.search("first")).to include(note1, note3)
    expect(Note.search("first")).to_not include(note2)
  end
end
```

search 作用域应该返回一系列匹配搜索词条的记录，而且应该只包含匹配词条的记录。

我们还可以利用这段测试做些其他实验？例如，把 to 换成 to_not 会怎样？或者再添加一些包含搜索词条的记录呢？

3.7 测试失败情况

我们测试了正常情况，即用户搜索的词条有返回结果，但如果凑巧搜索的词条没有结果呢？我们最好也测试一下这种情况。下面的测试应该可以涵盖这种情况：

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
```

```
    # 能搜索到结果的测试 ...

    it "returns an empty collection when no results are found" do
      user = User.create(
        first_name: "Joe",
        last_name:  "Tester",
        email:      "joetester@example.com",
        password:   "dottle-nouveau-pavilion-tights-furze",
      )

      project = user.projects.create(
        name: "Test Project",
      )

      note1 = project.notes.create(
        message: "This is the first note.",
        user: user,
      )
      note2 = project.notes.create(
        message: "This is the second note.",
        user: user,
      )
      note3 = project.notes.create(
        message: "First, preheat the oven.",
        user: user,
      )

      expect(Note.search("message")).to be_empty
    end
  end
```

这段测试使用 RSpec 的 `be_empty` 匹配器检查 `Note.search("message")` 的返回值中是不是没有内容。因为返回结果为空，这个测试能通过。我们不仅测试了正常情况，即用户搜索的词条有结果，而且也测试了没有搜索结果的情况。

3.8 匹配器

我们已经实际使用了四个匹配器：`be_valid`，`eq`，`include` 和 `be_empty`。`be_valid` 由 `rspec-rails` gem 提供，我们用它测试 Rails 模型的有效性。`eq` 和 `include` 由随 `rspec-rails` 一起安装的 `rspec-expectations` gem提供。

RSpec 默认提供的全部匹配器可以到 rspec-expectations 项目的 *README* 文件中查看。第 8 章将说明如何自定义匹配器。

3.9 使用 `describe`、`context`、`before` 和 `after` 去除重复

目前，我们编写的测试代码有些重复：每个测试用例中都创建了四个相同的对象。跟应用的代码一样，DRY 原则同样适用于测试（不过也有特例，详见后文）。下面利用一些 RSpec 技巧来整理一下测试代码。

以刚刚为 Note 模型编写的测试为例。首先，在 describe Note 块中再添加一个 describe 块，专门用于测试搜索功能。测试的大纲如下：

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  # 数据验证测试

  describe "search message for a term" do
    # 搜索功能测试用例 ...
  end
end
```

然后再添加两个 context 块，分别对应能搜到结果和不能搜到结果两种情况：

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  # 其他测试

  describe "search message for a term" do

    context "when a match is found" do
      # 能搜到结果的测试用例 ...
    end

    context "when no match is found" do
      # 不能搜到结果的测试用例 ...
    end
  end
end
```

注意

其实，严格来说，describe 和 context 是可以互换的，但我倾向于这么用：describe 用来表示需要实现的功能，而 context 针对该功能不同的情况。在这里我就用 context 描述了两种情况，即能搜到结果和不能搜到结果两种情况。

你可能已经看出来了，我们编写的这个测试大纲可以帮助我们把类似的测试用例组织在一起，这样测试读起来更通顺。现在我们要用 before 块来清理一下重新组织的测试。before 块放在 describe 或 context 中，其中的代码在各个测试之前运行。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  before do
    # 设置这个文件中所有测试都能用得到的数据
```

```
  end

  # 数据验证测试

  describe "search message for a term" do

    before do
      # 设置与搜索有关的测试数据
    end

    context "when a match is found" do
      # 能搜到结果的测试用例 ...
    end

    context "when no match is found" do
      # 不能搜到结果的测试用例 ...
    end
  end
end
```

RSpec 的 before 块是消除测试重复的利器。去除重复的方法很多，不过 before 块或许是最常用的。before 块可以在每个测试用例之前运行，可以在一组测试用例之前运行，也可以在整个测试组件之前运行：

- before(:each) 在 describe 或 context 块中的每个测试用例之前运行。如果愿意，可以使用别名 before(:example)，或者像前述示例中那样，直接使用 before。如果一个测试文件中有四个测试用例，那么 before 块将运行四次。
- before(:all) 在 describe 或 context 块中的所有测试用例之前运行一次。这是 before(:context) 的别名。此时，先运行一次 before 块，然后运行四个测试用例。
- before(:suite) 在整个测试组件之前运行。

before(:all) 和 before(:suite) 可以把耗时的设置独立出来，不过也可能为测试带来干扰。只要可能就应该使用 before(:each)。

注意

如果采用这里的方式定义 before 块，块中的代码将在每个测试用例之前运行。如果想明确表明意图，可以使用 before :each。二者随意使用，只要符合自己或团队的规范就行。

假如测试在用例执行后需要做些清理工作，例如中断与外部服务的连接，可以使用 after 块来善后。与 before 块一样，after 块也有 each、all 和 suite 三个选项。因为 RSpec 会自行清理数据库，所以我很少使用 after。

我们来看一下重新整理后的完整测试：

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  before do
```

```
  @user = User.create(
    first_name: "Joe",
    last_name:  "Tester",
    email:      "joetester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )

  @project = @user.projects.create(
    name: "Test Project",
  )
end

it "is valid with a user, project, and message" do
  note = Note.new(
    message: "This is a sample note.",
    user: @user,
    project: @project,
  )
  expect(note).to be_valid
end

it "is invalid without a message" do
  note = Note.new(message: nil)
  note.valid?
  expect(note.errors[:message]).to include("can't be blank")
end

describe "search message for a term" do
  before do
    @note1 = @project.notes.create(
      message: "This is the first note.",
      user: @user,
    )
    @note2 = @project.notes.create(
      message: "This is the second note.",
      user: @user,
    )
    @note3 = @project.notes.create(
      message: "First, preheat the oven.",
      user: @user,
    )
  end

  context "when a match is found" do
    it "returns notes that match the search term" do
      expect(Note.search("first")).to include(@note1, @note3)
    end
  end

  context "when no match is found" do
    it "returns an empty collection" do
      expect(Note.search("message")).to be_empty
```

```
      end
    end
  end
end
```

你可能注意到设置测试数据的方法跟之前有细微的差别。把分布在各测试用例中的设置移到 `before` 块中后，用户要赋予实例变量。否则，在测试用例中无法访问。

执行这些测试后，会看到一个十分清晰的大纲（因为在第 2 章把 RSpec 的输出格式设成了“文档”格式）：

```
Note
  is valid with a user, project, and message
  is invalid without a message
  search message for a term
    when a match is found
      returns notes that match the search term
    when no match is found
      returns an empty collection

Project
  does not allow duplicate project names per user
  allows two users to share a project name

User
  is valid with a first name, last name and email, and password
  is invalid without a first name
  is invalid without a last name
  is invalid with a duplicate email address
  returns a user's full name as a string

Finished in 0.22564 seconds (files took 0.32225 seconds to load)
11 examples, 0 failures
```

警告

有些开发者习惯使用方法名作为内部 `describe` 块的描述文本，例如把 `search for first name, last name, or email` 写成 `#search`。我个人并不喜欢这种方式，我坚信这个文本应该表述代码的作用而不仅仅是列出方法名。其实，我也不十分反对使用方法名。

3.9.1 避免过分消除重复

本章我们用了很多篇幅介绍如何把测试变得条理清晰。这是一个很容易被滥用的功能。

准备测试数据时，我认为 DRY 原则要适当地让位于可读性。如果你打开一个大型测试文件（或者加载了很多外部支持文件），来回拉滚动条，想弄明白到底在测试什么，这时就要考虑把测试数据放入更细化的 `describe` 块中，或者干脆直接在测试用例中创建所需的数据。

有时命名良好的变量可以节省很多时间。在上面的测试中我们定义的记录分别名为 `@note1` 和 `@note2`，但有些时候，命名为 `@matching_note` 或 `@note_with_numbers_only` 更好。当然，具体如何命名要视情况而定。不过一般而言，最好为变量和方法起个能表明其意的名称。

第 8 章将深入讨论这个话题。

3.10 小结

本章关注的是模型测试，不过也涉及了很多其他技术，在其他类型的测试中也可以使用：

- 明确指定希望得到的结果：使用动词说明希望得到的结果。每个测试用例只测试一种情况。
- 测试希望看到的结果和不希望看到的结果：测试时要分别思考两种情况，然后分别测试。
- 测试极端情况：如果有一个数据验证要求密码的长度在 4~10 个字符范围内，不要只测试 8 个字符的密码就觉得可以了。一个合格的测试应该分别检测 4 个字符和 10 个字符的情况，还要检测 3 个字符和 11 个字符的情况。（当然，借由这个机会你也可以思考一下为什么我要把密码限制的这么短，再长一些会不会更好？测试的过程中往往会有很多灵感，可以优化应用的需求和代码。）
- 测试要良好地组织，保证可读性：使用 `describe` 和 `context` 组织测试，形成一个清晰的大纲；使用 `before` 和 `after` 块消除代码重复。不过，当可读性更重要时，例如避免总是来来回回查看测试文件，就可以适当地有点重复。

为应用编写完整的模型测试后，就可以更确信代码是可以正常使用的。

3.11 问答

我应该怎么决定何时用 `describe`，何时用 `context` 呢？

对 RSpec 来说，如果你喜欢，可以只用 `describe`。和 RSpec 很多其他的功能一样，之所以提供 `context` 只是为了提升代码的可读性。你可以参考本章的用法，把 `context` 用在同一功能的不同条件中，或者根据应用的需求选择使用。

3.12 练习

- 为示例应用添加更多测试。我只为模型的部分功能编写了测试，例如 `Project` 模型的数据验证没有测试。自己编写试试。如果你已经配置了自己的应用使用 RSpec，也为你自己的模型添加一些测试。