



The
Pragmatic
Programmers

华章专业开发者丛书

本书第1版曾荣获Jolt大奖
Rails之父代表作

Web开发敏捷之道

应用Rails进行敏捷Web开发

原书第4版

Agile Web Development with Rails Fourth Edition

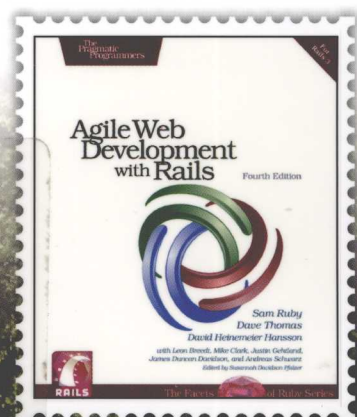
Sam Ruby

(美) Dave Thomas

著

David Heinemeier Hansson

慕尼黑Isar工作组 骆古道 等译



机械工业出版社
China Machine Press

华章专业开发者丛书

Web开发敏捷之道

应用Rails进行敏捷Web开发

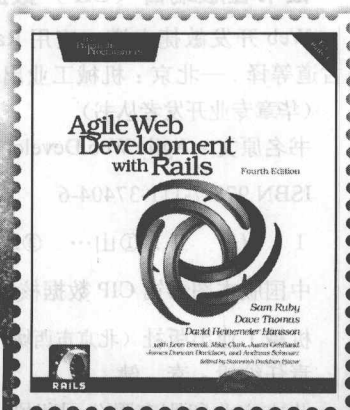
原书第4版

Agile Web Development with Rails Fourth Edition

Sam Ruby
(美) Dave Thomas

David Heinemeier Hansson

慕尼黑Isar工作组 骆古道 等译



机械工业出版社
China Machine Press

本书第1版曾荣获 Jolt 大奖“最佳技术图书”奖。在前3版的内容架构基础上,第4版增加了关于 Rails 中新特性和最佳实践的内容。本书从逐步创建一个真正的应用程序开始,然后介绍 Rails 的内置功能。全书分为3部分,第一部分介绍 Rails 的安装、应用程序验证、Rails 框架的体系结构,以及 Ruby 语言的知识;第二部分用迭代方式创建应用程序,然后依据敏捷开发模式搭建测试案例,最终用 Capistrano 完成部署;第三部分有条不紊地补充缺少的知识并涵盖足以应付日常的实际工作。本书既有直观的示例,又有深入的分析,同时涵盖了 Web 应用开发中各方面的相关知识,堪称一部内容全面而又深入浅出的佳作。

本书适合 Ruby 和 Rails 的初级、中级读者阅读,并可作为开发人员的参考手册。

Sam Ruby, Dave Thomas, David Heinemeier Hansson. *Agile Web Development with Rails, Fourth Edition* (ISBN 978-1-934356-54-8).

Copyright © 2011 The Pragmatic Programmers, LLC. All rights reserved.

Simplified Chinese Translation Copyright © 2012 by China Machine Press.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system, without permission, in writing, from the publisher.

All rights reserved.

本书中文简体字版由 The Pragmatic Programmers, LLC 授权机械工业出版社在全球独家出版发行。未经出版者书面许可,不得以任何方式抄袭、复制或节录本书中的任何部分。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2011-4810

图书在版编目(CIP)数据

Web 开发敏捷之道:应用 Rails 进行敏捷 Web 开发(原书第4版)/(美)山姆(Sam, R.)等著;骆古道等译. —北京:机械工业出版社,2012.3

(华章专业开发者丛书)

书名原文:Agile Web Development with Rails, Fourth Edition

ISBN 978-7-111-37404-6

I. W… II. ①山… ②骆… III. 互联网络—程序设计 IV. TP393.4

中国版本图书馆 CIP 数据核字(2012)第 020158 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:秦 健

北京京北印刷有限公司印刷

2012 年 3 月第 1 版第 1 次印刷

186mm×240mm·23 印张

标准书号:ISBN 978-7-111-37404-6

定价:59.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线:(010) 88378991; 88361066

购书热线:(010) 68326294; 88379649; 68995259

投稿热线:(010) 88379604

读者信箱:hzsj@hzbook.com

对本书的赞誉

当我开始学习 Ruby on Rails 时，我使用的是本书的第 1 版。其中关于 Rails 框架和社区的综述，为每一位 Rails 新手提供了成就辉煌事业的机会。在我读了最新的版本之后，我可以很欣慰地说，它秉承了第 1 版的这种趋势和风格，正因为这样，我把它推荐给每一位 Rails 新手。

——Mikel Lindsaar

Rails 核心委员会成员、Ruby Mail 库创始人、RubyX 负责人

这本书可以让读者用一种愉快且难忘的方式访问 Rails 环境，它在这方面做得非常出色。另外，我还是第一次见到一本书把 MVC 模式讲得如此合情合理、条理清楚，并通过实例消除了它的一切神秘感。

——Ken Coar

作家、开源软件推行者和 Apache 开发者

这本书成功地满足了不同读者的需求，既是 Rails（和 Ruby）的趣味读物，又以平实的语言讲解了框架的高级属性，完全有别于在线文档。

——Glen Daniels

独立技术专家和顾问

我从来没有读过像这本书一样成功的程序设计书籍。Sam 把 Ruby on Rails 讲得通俗易懂，全面而有趣。

——Keith Ballinger

WS-I 的第一 Basic Profile 工作组主席、作家、.NET 和 Visual Studio .NET 框架的主创人员

……

彭方强

罗永平

译者序

每一种成熟的软件框架都要经历一段漫长而又艰难的过程。当你打开本书时，Ruby on Rails（简称 Rails 或 RoR）框架已走过了七个年头了。Rails 框架从一开始，就带动了一门语言受到全球业内人士的高度关注。Rails 框架从诸强林立中脱颖而出，成为其他语言框架竞相模仿的楷模。Rails 框架从实践中来，已经成为人们广泛采用的互联网应用程序框架，并也在大型商业网站中得到了应用。

技术伴随着人类社会经济的发展而不断进步，Rails 框架也不例外。今天的 Rails 框架已经从框架底层结构上进行了全面彻底革新；在框架开发结构上更进一步全方位软件模块化。可以说，Rails 框架的每一次版本更新都包含了人们所期望的功能，让人感到振奋和喜悦，同时引领着软件框架发展的潮流。

放眼互联网世界，每一天都会呈现出无数的 gem 软件包。这些 gem 软件包提供了 Rails 框架的第三方资源。在本书中，作者推荐了一些十分优秀的 gem 软件包。这些资源不仅极大地丰富了 Rails 框架开发的需求，而且还避免了软件工程的重复劳动，提高了软件开发的效率和速度。

行动是成功的基础。普遍认为本书是 Rails 框架实践的经典之作，其作者就是该软件框架的创始人。在本书中，作者阐述了该框架最本质和最重要的内容。只有真正理解和认识 Rails 框架的本质，才能掌握和驾驭该框架的功能。当然，Rails 软件框架同时也是软件理论研究的成果，源自于成千上万人的实践，是人类智慧的结晶和实践的成果。这种成果是发展的和动态的。因此，学习和认识 Rails 框架是无止境的。

每一次翻译工作都是一次新的挑战。如何理解不断出现的新概念、新理念和新名词？如何克服陈旧概念、思维定式对我们的影响？如何面对看似可以理解却难以用汉语表达的句子？面对这一切，一次思考不行，再来一次；一个人的能力不行，再补充一个；一遍修改不行，再来一遍。即便在本书正式出版以后，要是你有不同的想法和理解，我们欢迎与你重新斟酌每一条语句。

合作是我们大家完成此书共同愿望。因此，在这里我首先要感谢为本书翻译和内部审稿付出辛苦劳动的所有慕尼黑 Isar 工作组同仁（按姓名笔画排序）：王大力、王德志、陈宇翔、姜伟、姜凯、胡捷、黄力和程帆。其次我要感谢负责本书出版和尽心劳动的机械工业出版社陈冀康编辑及其同仁。最后我也要感谢我的妻子以及同仁的家人，是他们给予了我们每一个人最大的鼓励和支持，使得此次翻译工作得以圆满完成。

一天一年，我们好好阅读此书……

骆古道
于慕尼黑

译者简介

(按姓名笔画排序)

王大力

Ruby 爱好者, 自 2005 年底开始自学 Rails, 并加入中文社区, 喜欢与国内爱好者交流学习心得, 介绍国外 Ruby 的动态。收集整理过 2006 ~ 2007 年由国内外开发者开发的中文 Rails 应用, 发起过 R4R 读书会, 在国内爱好者老徐提供的共享空间上搭建 Typo 博客, 录制过一些介绍 Rails 的视频, 网名自诩铁道播客, 在社区网站 Railsen 和 Chinaonrails 上使用的是个人业余无线电电台的呼号 BD7LX。

王德志

1988 年从西安交通大学计算机系研究生毕业后, 从事 VAX VMS 系统管理员工作 7 年、IT 项目经理和 IT 部门经理工作 15 年, 在中国和加拿大成功实施过多个 IT 相关项目, 拥有 8 种业界认可的专业技术证书, 参与了包括中国科学院 CAD 开放实验室 / 国家智能中心在内的多种研究机构的科研课题, 发表论文 30 余篇, 独立或参与编译 6 本 IT 技术书籍, 1997 年获高级电子工程师资格。

陈宇翔

毕业于慕尼黑工业大学计算机系, 现攻读博士, 主攻图像处理、机器学习、模式识别等研究方向。

骆古道

骆古道 (Gudao Luo), 网名 Cnruby, 20 世纪 80 年代初毕业于西北工业大学数理力学系, 1988 年公派留学德国, 从事组合最优化理论研究, 从 90 年代初期起一直致力于计算机领域软件开发、设计和管理等方面工作, 个人博客为“道喜技术日记”和“天天红玉世界”。

姜伟

上海应用技术学院副教授、系副主任, 主要从事计算机应用、企业信息化管理、系统工程、电子商务、管理信息系统等科研与教学工作。曾获部级科技进步奖 (三等奖) 一项、部级管理成果奖 (二等奖) 一项、省级自然科学奖 (论文二等奖) 两项、省部级优秀论文奖多项。

姜凯

2006 年留学德国, 毕业于慕尼黑工业大学电子系, 研究方向为微处理系统及数据和图像处理。

胡捷

毕业于慕尼黑工业大学计算机专业，在全球著名 IT 咨询公司担任技术顾问和架构师。在 CRM 领域的系统架构、实施计划、系统实施、系统集成、系统选型等方面具有多年工作经验。

黄力

现于慕尼黑工业大学土木学院攻读博士学位。专业方向是建筑信息模型，涉及针对初期设计模型的可持续发展分析以及钢 - 混凝土组合结构建筑模型自动生成的研究。

程帆

2007 年留学德国，从慕尼黑工业大学计算科学与工程专业毕业后，担任软件工程师，主要从事芯片分析软件的开发工作。此外，对人工智能以及自动控制有浓厚兴趣。业余时间则主要专注于中文教学。

王鹏志

1988 年毕业于西安交通大学计算机专业，毕业后在西安交通大学从事教学和科研工作。1992 年调入清华大学，从事清华大学计算机系的教学和科研工作。1997 年调入清华大学计算机系，从事清华大学计算机系的教学和科研工作。1997 年调入清华大学计算机系，从事清华大学计算机系的教学和科研工作。

蔡宇明

毕业于慕尼黑工业大学工程力学专业，从事工程力学方面的研究工作。在工程力学、材料力学、流体力学等方面有较深的造诣。主要从事工程力学方面的研究工作。

郭古超

毕业于清华大学计算机系，从事计算机方面的研究工作。在计算机组成原理、计算机体系结构、计算机操作系统等方面有较深的造诣。主要从事计算机方面的研究工作。

姜伟

毕业于清华大学计算机系，从事计算机方面的研究工作。在计算机组成原理、计算机体系结构、计算机操作系统等方面有较深的造诣。主要从事计算机方面的研究工作。

董皓

毕业于清华大学计算机系，从事计算机方面的研究工作。在计算机组成原理、计算机体系结构、计算机操作系统等方面有较深的造诣。主要从事计算机方面的研究工作。

第4版序言

当 Dave 希望我作为本书第 3 版的合著者时，我深感激动。因为我就是基于这本书的第 1 版学习 Rails 的。Dave 和我有许多共同之处，虽然他喜欢使用 Emacs 和 Mac OS X，而我喜欢使用 Vim 和 Ubuntu，但我们都爱好命令行和写代码——先从具体的代码入手，再学习深层理论。

自从第 3 版（实际上，包括第 1 版、第 2 版和第 3 版）出版以来，Rails 改变了很多。Rails 正处于重要的重构过程中，大部分是在其内部。因此，这一版从根本上抛弃了以前在代码中使用的一些特性，同时又加入了新的特性，还有我们从经验中得到的内容：使用 Rails 的最佳实践是什么。Rails 现在可以运行在 Ruby 1.9 上，本书中的每一个例子都已在 Ruby 1.8.7 和 Ruby 1.9.2 中测试通过。

另外，Rails 已经从一个流行的框架变为一个活跃的、充满生机的生态系统，而且许多流行插件和深度整合的第三方工具让它变得更完美。现在 Rails 逐渐变为一种主流工具，吸引了广大的开发人员到这个框架中来。

这使得我们需要重新组织这本书。根据读者反馈，许多 Rails 新手对开篇没有介绍 Ruby 而感到不适应，所以我们将这部分从附录里提到第一部分的第 4 章里。跟随第一部分内容一步步操作，从而构建一个真正的应用程序。为了把重点放在当前的最佳实践上，这一部分内容已经经过更新和简化。

本书最大的变动是最后一部分：因为它已经不适用于覆盖整个 Rails 系统，也跟不上其变化，现在这部分内容将专注于提供一个综合的视角，使读者能够了解可以找什么，当框架本身不能满足普通需求时，在哪里可以找到插件和相关工具来解决这个问题。

简而言之，这本书必须再一次修订。

Sam Ruby

2011 年 3 月

前言

Ruby on Rails 是一个能简化开发、部署和维护 Web 应用程序的框架。在 Rails 最初版本发布后的几个月内，Rails 就由开始的不为人知而发展为风靡全球，而更重要的是，它已经成为对于开发 Web 2.0 应用程序的重要选择。

为什么会这样？

Rails 就是让人感觉舒服

首先，大量的开发人员对其使用的 Web 应用程序开发技术感到失望，不论他们使用的是 Java、PHP 或是 .NET——它们使开发人员的工作难度不断增加。正在这时，Rails 出现了，它简单易学。

但是仅仅简单是不够的。我们谈论的是编写实实在在网站应用的专业开发人员。他们总是通过使用最新的、最专业的技术来让自己的应用程序能够经得起时间的考验。这些开发人员在深入 Rails 之后，就会发现它远不仅限于是开发网站的工具。

举例来说，所有的 Rails 应用程序都使用了模型 - 视图 - 控制器 (Model-View-Controller, MVC) 的结构。Java 程序员通常使用的 Tapestry 和 Struts 框架都是基于 MVC 的。但是 Rails 在应用 MVC 方面更进了一步：当你刚开始使用 Rails 开发时，你就已经处于一个工作状态的应用程序中了，每段代码都置于相应的位置，而且所有的应用程序段都以某种标准的方式交互。

专业的程序员总是编写相应的测试程序。Rails 也同样提供了这方面的支持。所有 Rails 应用程序内置了对测试的支持。当在代码中加入新的功能时，Rails 就会自动创建针对这项新功能的测试存根 (stub)。Rails 框架让应用程序的测试变得更容易，因此，Rails 应用程序趋向于已经通过测试的应用程序。

Rails 应用程序是用 Ruby 编写的，Ruby 是一种现代的面向对象脚本语言。Ruby 简练而容易理解，你可以通过 Ruby 自然而清晰地表述自己的想法，这使得使用 Ruby 编写程序很容易，而且即使几个月后再回顾代码时也很容易（这是很重要的）。

Rails 通过给 Ruby 加上了一些限制和一些新颖的扩展，从而减轻了程序员的工作，而且程序变得更短小、更易读，并且 Rails 允许我们将以前需要在外部配置文件中才能完成的任务，放到代码库中来完成。这样一来，我们可以更容易地弄明白正在发生的是什么。下面的代码定义了一个项目中的模型类。现在你不必关心其中的细节，只需要注意在这短短几行代码中描述了多少信息即可。

```
class Project < ActiveRecord::Base
  belongs_to :portfolio
  has_one    :project_manager
```

```

has_many :milestones
has_many :deliverables, :through => :milestones

validates :name, :description, :presence => true
validates :non_disclosure_agreement, :acceptance => true
validates :short_name, :uniqueness => true
end

```

在 Rails 中，有两个哲学观点使得 Rails 的代码短小易读：不要自我重复（Don't Repeat Yourself, DRY）和惯例重于配置（convention over configuration）。不要自我重复意味着：系统中的每个知识点只应该在一个地方描述一次。Rails 借助了强大的 Ruby 来实现这一目标。在 Rails 应用程序中，你几乎看不到重复的代码，把需要表达的内容只放在一个地方——通常是符合 MVC 架构惯例的地方。使用其他 Web 框架的程序员大多有这样的经历：只要对程序结构做一点点修改，就必须同时修改好几处代码，这是一种灾难。

惯例重于配置也同样重要。Rails 精准地定义了如何将应用程序拼接起来。只要遵循这些惯例，编写一个 Rails 应用程序所需的代码量远比使用 XML 配置的典型 Java Web 应用程序少得多。如果你不想应用这些惯例，在 Rails 中也可以很简单地实现。

除此之外，开发者还会在 Rails 中发现：它并没有给还很新的实际 Web 标准添乱，相反，它正在帮助定义这些标准。还有，Rails 能让开发者更轻松地将 Ajax 和 REST 风格的接口之类的新技术整合到自己的应用中——它内置了对这些技术的支持（如果你还不熟悉 Ajax 和 REST 接口，别担心，我们会在本书后面介绍它们）。

开发者还需要经常担心应用程序的部署问题。而使用 Rails，只需要输入一条命令，就可以将最新版本的应用程序部署到任意多台服务器上（如果发现上传的版本有问题，撤销也同样容易）。

Rails 来源于实际的商用程序。创造一个框架最好的办法是：首先概括出特定应用的核心功能，然后从中抽取出通用的代码作为基础。在开发 Rails 应用程序时，你就会发现，一个出色的应用程序的一半已经有了。

当然，Rails 还有其他优点——有些甚至很难言传。总之，Rails 就是让人感觉很舒服。当然了，这要等到你自己动手写一些 Rails 的应用程序时才能体会到（这大概是下一个 45 分钟的任务……）。这也就是我们本书的主旨。

Rails 是敏捷的

当你看到本书的书名的时候，可能会感到奇怪，为什么书里没有关于“在 Rails 中编写某某敏捷实践”这样的章节。

原因很简单：敏捷是 Rails 开发基础的一部分。

我们先来看看下面 4 条“敏捷开发宣言”所描述的价值观：^①

- 人和人与人之间的交互重于工序和工具。
- 可工作的软件重于全面的文档。
- 与客户交流重于合同谈判。

① <http://agilemanifesto.org/>。Dave Thomas 是这篇文档的 7 位作者之一。

- 快速响应变化重于墨守成规。

Rails 完全是为了增进人与人之间的交互。这里没有繁重的工具，没有复杂的配置，没有冗长的过程。这里只有开发者组成的小集体，他们最爱的编辑器，以及一堆 Ruby 代码。这使得开发具有高透明度，开发者的工作能够立即让客户看到。这是一个本质上的交互过程。

Rails 并不废弃文档。通过 Rails 可以轻易地为所有代码库生成 HTML 格式的文档。但 Rails 的开发过程并不由文档驱动。在一个 Rails 项目的核心里，你不会找到一份 500 页的说明书，只会看见一组用户和开发者共同探讨着他们的需求和寻找实现需求的办法。你会发现，通过这样的方式，随着开发者和用户逐渐加深对问题的了解，解决方案也会不断变化。你会发现，这个团队在开发周期的初期就开始交付可以工作的软件。这个软件的细节可能很粗糙，但它让用户可以马上初步体验到将来的产品。

通过这种方式，Rails 鼓励开发人员与客户之间进行合作。当客户看到 Rails 项目能够这么快响应他们的需求变化，他们开始相信这个开发团队可以提交令他们满意的产品，而并不只是仅仅符合基本要求。客户和开发人员的争执也变成互相之间的探讨。

归根到底，这些都是为了能够使 Rails 及时地响应变化。其中最能体现 Rails 符合 DRY 准则的一点是，对于 Rails 应用程序的调整，其影响的代码量要远远小于其他框架。又由于 Rails 是用 Ruby 开发的，因此开发理念可以通过 Ruby 简练而精准地表述，对于代码的修改也只限于局部化且易于编写。另外，Rails 特别重视对软件单元和功能的测试，以及对测试套件和测试存根的支持，这些都给开发者再调整程序时提供了相对安全的保障。有了这样一组完善的测试作为保障，对程序的调整已不再是一件伤脑筋的事情了。

相对不断提及 Rails 的敏捷性，我们认为最好的方式还是让框架自己来说话。在你阅读前 4 章的内容时，请设想你正在进行网站开发：你正在与客户进行讨论，确定任务的优先级和解决方案。然后，在你读到第三部分的高级部分时，再考虑如何通过你已了解的 Rails 框架来更快地实现客户需求。

关于敏捷开发和 Rails 的最后一点是：虽然可能听起来不是那么专业，但是你会体会到写代码时的无穷乐趣。

本书读者对象

本书适合正在寻找工具用于开发和部署基于 Web 的应用程序的程序员。这既包括 Rails 的入门者（甚至也许是 Ruby 的入门者），也包括那些已经具备 Rails 基本知识而希望更进一步学习的程序员。

我们假设读者已经熟悉了一些关于 HTML、CSS（Cascading Style Sheet，级联样式表）和 JavaScript 的基本知识，也就是说，可以读懂网页的源代码。你不需要是这方面的专家，你只要能从本书中复制和粘贴代码就行，这些代码也可以直接从网站上下载。

怎样阅读本书

本书第一部分是学好 Rails 的一切前提。通过学习这一部分的内容，你可以简单地了解

Ruby (编程语言), 关于 Rails 的概述, 以及关于 Ruby 和 Rails 的安装, 并且通过一个简单的例子来检验安装过程。

第二部分将通过一个扩展过的示例来介绍 Rails 背后的一些概念。在这个示例里, 我们创建了一个简单的网上商店。我们并不准备在这个示例里详细介绍 Rails 的每一个组件 (比如, 这一章是关于 MVC 的模型, 这里是关于 MVC 的视图等内容)。这些组件在设计时就是为了互相协调工作, 该部分的每一章只涉及了一组相关任务中的一部分, 而这组任务是通过很多组件互相协调来实现的。

大多数读者似乎乐于边读书边亲手尝试构建应用程序。但如果你懒得输入, 也可以直接下载源代码 (提供压缩的 tar 文件包和 zip 压缩包两种格式)。^①

本书第三部分涵盖了整个 Rails 生态系统。首先会介绍 Rails 功能和设备。然后介绍一些主要附件, 正是它们充分发挥了 Rails 框架的功能。最后还介绍了一些流行的 Rails 插件, 正是它们成就了 Rails 的开放生态系统, 而不仅仅是一个框架。

通读全书, 你会看到我们所采用的不同的约定。

本书中展示的代码片段大多来自可以下载的完整应用程序。为了帮助读者理解, 如果一段代码能够在下载的应用中找到, 在代码的上边就会有一条路径指明它所在的文件, 就像这样:

```
work/demol/app/controllers/say_controller.rb
```

```
class SayController < ApplicationController
▶   def hello
▶   end

    def goodbye
    end

end
```

有时候并不能马上找到书中例子到底修改了源文件的哪些行, 代码左边的小三角形可以帮助你清楚地找到这些修改的行。前面代码中的两行就有这样的指示。

Ruby 提示

虽然你需要懂 Ruby 才能编写 Rails 应用程序。不过我们意识到很多人在读本书的时候其实已在同时学习 Ruby 和 Rails 了。第 4 章将简单介绍 Ruby 语言。

David 说

你会经常看到“David 说”这样的内容, 其中的内容是 David Heinemeier Hansson 想要告诉你他对于 Rails 的独特见解——原理、技巧、推荐等。David 是 Rails 的创始人之一, 所以, 如果你想成为 Rails 专家的话, 这些内容是不容错过的。

Joe 问

Joe 是一个虚构的开发者形象, 他常常会针对本书讲解的内容提些问题, 而我们则会试着回答这些问题。

① http://pragprog.com/titles/rails4/source_code 是下载链接。

这不是一本 Rails 参考手册。经验告诉我们，大部分人都不是通过查阅参考手册来学习的，我们将展示大部分模块和方法，可能通过示例，也可能通过文字介绍来指明如何使用这些组件，以及如何把它们组合在一起。

但我们不会列出上百页的 API 列表。这么做的原因是，只要安装 Rails，就已经得到了最新的 API 文档，而且肯定比本书的内容还要新。如果通过 RubyGems 安装了 Rails（这也是推荐的安装方式），只要启动 gem 文档服务器（使用 `gem_server` 命令），再用浏览器访问 <http://localhost:8808>，你就可以访问所有的 Rails API 文档。你会在第 12 章中找到怎样建立更多的文档。

另外，你会发现，Rails 通过应答，可以清晰地指明错误所在，通过错误跟踪，Rails 不仅会告诉你错误在哪里，还包括错误的原因。你可以在 10.2 节的图 10.3 中找到这样的例子。如果你想知道关于这方面更多的信息，可以先翻到 10.2 节，看看怎样插入日志语句。

如果你真的在读到这本书的某个地方遇到了问题，不用担心，我们有大量的在线资源可以帮助你。另外，除了我们提及的代码清单，还有一个论坛[⊖]，在那里可以提问题和分享经验；还有一个勘误页[⊗]，在那里可以进行错误报告；最后还有一个 wiki[⊕]，在那里可以讨论书中的练习。

这些都是共享资料。请自由地提交你的疑问和问题到论坛和 wiki 上，还有你的建议，以及对你所了解的问题的答案。

那现在就让我们开始吧！第一步是安装 Ruby 和 Rails，然后我们通过一个简单的样例来检验你是否成功地安装了它们。

关于 Rails 版本的重要信息

本书是针对 Rails 3 写的。

因为 Rails 的核心团队在不断地改进 Rails，所以随着时间推移，新的 Rails 版本可能与之前的版本不兼容，这种问题可能在运行本书的样例代码时会遇到。

为避免出现这种情况，在运行本书的样例代码时，请按照在本书 1.1 节所述的那样，安装正确的 Rails 版本。

可以通过输入 `rails -v` 命令来确定 Rails 的版本。

可以在下面的链接中找到关于本书的勘误。网址：<http://www.pragprog.com/wikis/wiki/ChangesToRails>。

⊖ <http://forums.pragprog.com/forums/148>

⊗ <http://www.pragprog.com/titles/rails4/errata>

⊕ <http://www.pragprog.com/wikis/wiki/RailsPlayTime>

致 谢

也许你认为修订一本图书是一件容易的事。毕竟文字什么的都已经有了。只须修改部分代码和对文字进行一下调整，仅此而已。也会想……

很难表达清楚这有多么困难，至少我们的感觉是每一版图书花费的精力都与第 1 版一样多。Rails 正在不断地进化中，正如本书一样。部分 Depot 应用程序经过多次修改，所有的叙述部分也已经更新过。REST 的重要性以及对弃用特性的回避，导致了本书的结构不断改变，因为以前的热点现在已经过时了。

如果没有 Ruby 和 Rails 社区的大力帮助，就不会有本书。首先，我们要衷心感谢对本书进行审校的人员：

Jeremy Anderson、Ken Coar、Jeff Cohen、Joel Clermont、Geoff Drake、Pavan Gorakavi、Michael Jurewitz、Mikel Lindsaar、Paul Rayner、Martijn Reuvers、Doug Rhoten、Gary Sherman、Davanum Srinivas、Stefan Turalaki 和 José Valim

另外，本书在正式出版之前曾有一个 PDF 的 beta 版本，人们可以在线添加评论。网友反应热烈，仅仅这一版就收到了超过 800 条建议和错误报告。通过基于如此多的自愿者的贡献，本书的实用价值已远远超过其最初。在此感谢所有支持本书 beta 版的网友，以及那些反馈了诸多好的意见的人，许多贡献者都做了大量的额外工作：

Manuel E. Vidaurre Arenas、Seth Arnold、Will Bowlin、Andy Brice、Jason Catena、Victor Marius Costan、David Hadley、Jason Holloway、David Kapp、Trung LE、Kristian Riiber Mandrup、mltsy、Steve Nicholson、Jim Puls、Johnathan Ritzi、Leonel S、Kim Shrier、Don Smith、Joe Straitiff 和 Martin Zoller

最后，Rails 核心团队提供了极大的帮助，他们帮忙回答问题、检验代码，并且修补错误。衷心感谢以下各位：

Scott Barron (htonl)、Jamis Buck (minam)、Thomas Fuchs (madrobby)、Jeremy Kemper (bitsweat)、Yehuda Katz (wycats)、Michael Koziarski (nzkoz)、Marcel Molina Jr (noradio)、Rick Olson (technoweenie)、Nicholas Seckar (Ulysses)、Sam Stephenson (sam)、Tobias Lütke (xal)、José Valim (josevalim) 和 Florian Weber (csssh)

——Sam Ruby

2011 年 3 月

rubys@intertwingly.net

目 录

对本书的赞誉

译者序

译者简介

第4版序言

前言

致谢

第一部分 起步

第1章 安装 Rails.....1

1.1 Windows 上的安装.....1

1.2 Mac OS X 上的安装.....3

1.3 Linux 上的安装.....4

1.4 选择一个 Rails 版本.....5

1.5 设置开发环境.....6

1.5.1 命令行.....6

1.5.2 版本控制.....6

1.5.3 编辑器.....7

1.5.4 桌面.....8

1.6 Rails 和数据库.....9

1.7 本章小结.....10

第2章 即时满足.....11

2.1 新建一个应用程序.....11

2.2 Hello, Rails !.....13

2.2.1 Rails 和 URL 请求.....14

2.2.2 第一个动作.....14

2.2.3 创建动态网页.....15

2.2.4 动态内容.....15

2.2.5 把时间加上.....16

2.2.6 故事讲到现在.....17

2.3 把页面连起来.....18

2.4 本章小结.....20

2.4.1 练习时间.....20

2.4.2 清理现场.....21

第3章 Rails 应用程序框架.....22

3.1 模型、视图以及控制器.....22

3.2 Rails 的模型支持.....24

3.2.1 对象 - 关系映射.....24

3.2.2 Active Record.....25

3.3 Action Pack : 视图与控制器.....26

3.3.1 视图支持.....26

3.3.2 还有控制器.....26

第4章 Ruby 简介.....28

4.1 Ruby 是一门面向对象的语言.....28

4.1.1 Ruby 命名规则.....29

4.1.2 方法.....29

4.2 数据类型.....30

4.2.1 字符串.....30

4.2.2 数组和散列.....30

4.2.3 正则表达式.....32

4.3 逻辑方法.....32

4.3.1 控制结构.....32

4.3.2 代码块和迭代器.....33

4.3.3 异常.....34

4.4 组织结构.....34

4.4.1 类.....34

4.4.2 模块.....36

4.4.3	YAML	36
4.5	封送对象	37
4.6	综合分析	37
4.7	Ruby 语言习语	38

第二部分 构建应用程序

第 5 章	Depot 应用程序	41
5.1	增量式开发	41
5.2	Depot 是做什么的	42
5.2.1	用例	42
5.2.2	页面流程	42
5.2.3	数据	44
5.3	让我们来编码吧	45
第 6 章	任务 A：创建应用程序	46
6.1	迭代 A1：创建商品维护的 应用程序	46
6.1.1	创建 Rails 应用程序	46
6.1.2	创建数据库	46
6.1.3	生成脚手架	47
6.1.4	应用迁移	48
6.1.5	查看商品清单	49
6.2	迭代 A2：美化商品清单	51
6.3	本章小结	54
	练习时间	55
第 7 章	任务 B：验证和单元 测试	57
7.1	迭代 B1：验证	57
7.2	迭代 B2：模型的单元测试	60
7.2.1	真正单元测试	61
7.2.2	静态测试	63
7.2.3	使用静态测试数据	66
7.3	本章小结	67
	练习时间	67

第 8 章	任务 C：商品目录显示	68
8.1	迭代 C1：创建商品目录清单	68
8.2	迭代 C2：增加页面布局	71
8.3	迭代 C3：用帮助函数来调整 价格格式	74
8.4	迭代 C4：控制器功能测试	74
8.5	本章小结	77
	练习时间	77
第 9 章	任务 D：创建购物车	78
9.1	迭代 D1：寻找购物车	78
9.2	迭代 D2：将产品放到购物车中	79
9.3	迭代 D3：添加一个按钮	81
9.4	本章小结	85
	练习时间	85
第 10 章	任务 E：更智能的购物车	86
10.1	迭代 E1：创建更智能的购物车	86
10.2	迭代 E2：错误处理	90
10.3	迭代 E3：对购物车的最后加工	92
10.4	本章小结	96
	练习时间	96
第 11 章	任务 F：Ajax 初体验	97
11.1	迭代 F1：转移购物车	97
11.1.1	局部模板	97
11.1.2	改变流程	101
11.2	迭代 F2：建立一个基于 Ajax 的 购物车	102
11.2.1	排疑解难	103
11.2.2	客户永远不会满足	104
11.3	迭代 F3：高亮变化	104
11.4	迭代 F4：隐藏一个空的购物车	106
11.5	测试 Ajax 改变	110
11.6	本章小结	111
	练习时间	112

第 12 章 任务 G：付款	113	15.5 本章小结	175
12.1 迭代 G1：获取订单	113	练习时间	176
12.1.1 创建获取订单的表单	114	第 16 章 任务 K：部署和产品环境	177
12.1.2 获取订单细节	120	16.1 迭代 K1：用 Phusion Passenger 和 MySQL 部署	178
12.1.3 最后一个 Ajax 更改	124	16.1.1 安装 Passenger	178
12.2 循环 G2：Atom 推送	125	16.1.2 在本地部署应用程序	179
12.3 迭代 G3：分页	128	16.1.3 使用 MySQL 数据库	180
12.4 本章小结	131	16.1.4 加载数据库	182
练习时间	131	16.2 迭代 K2：用 Capistrano 远程部署	183
第 13 章 任务 H：发送电子邮件	132	16.2.1 准备好部署服务器	183
13.1 迭代 H1：发送确认邮件	132	16.2.2 把应用程序放到版本管理下	184
13.1.1 配置邮件	132	16.2.3 远程部署应用程序	185
13.1.2 发送邮件	133	16.2.4 冲洗，洗净，重复	187
13.1.3 邮件模板	134	16.3 迭代 K3：检查部署的应用程序	188
13.1.4 生成邮件	135	16.3.1 查看日志文件	188
13.1.5 发送多内容类型	136	16.3.2 使用命令行界面来查看实时的应用程序	188
13.1.6 邮件功能测试	137	16.3.3 处理日志文件	189
13.2 迭代 H2：应用程序的集成测试	138	16.3.4 开始发行，超越自我	189
13.3 本章小结	142	16.4 本章小结	190
练习时间	142	练习时间	190
第 14 章 任务 I：登录	143	第 17 章 Depot 回顾	191
14.1 迭代 I1：添加用户	143	17.1 Rails 的概念	191
14.2 迭代 I2：认证用户	150	17.1.1 模型	191
14.3 迭代 I3：限制访问	155	17.1.2 视图	192
14.4 迭代 I4：增加侧边栏，更多管理	157	17.1.3 控制器	192
14.5 本章小结	160	17.1.4 配置	192
练习时间	160	17.1.5 测试	193
第 15 章 任务 J：国际化	161	17.1.6 部署	193
15.1 迭代 J1：选择语言环境	161	17.2 文档化所做的事情	193
15.2 迭代 J2：翻译在线商店页面	164		
15.3 迭代 J3：翻译结账页面	169		
15.4 迭代 J4：添加语言环境的切换器	174		

第三部分 深入 Rails

第 18 章 自己去发现 Rails (工作)

方法.....195

18.1 东西都去哪里了.....195

18.1.1 应用程序的位置.....197

18.1.2 测试的位置.....197

18.1.3 文档的位置.....197

18.1.4 支持库的位置.....198

18.1.5 Rake 任务的位置.....199

18.1.6 日志的位置.....200

18.1.7 静态网页的位置.....200

18.1.8 脚本的位置.....200

18.1.9 临时文件的位置.....201

18.1.10 第三方代码的位置.....201

18.1.11 配置的位置.....201

18.2 命名约定.....202

18.2.1 混合大小写、下划线和复数.....202

18.2.2 把控制器分组到模块中.....203

18.3 本章小结.....205

第 19 章 Active Record 模块.....206

19.1 定义数据结构.....206

19.1.1 使用表和字段的规则.....206

19.1.2 Active Record 所提供的

附加字段.....209

19.2 查找和遍历记录.....210

19.2.1 识别单个行.....210

19.2.2 模型关联性说明.....211

19.2.3 一对一关联.....211

19.2.4 一对多关联.....212

19.2.5 多对多关联.....212

19.3 创建、读取、更新和删除操作.....213

19.3.1 创建新的行记录.....213

19.3.2 读取已有行记录.....215

19.3.3 动态查询器.....216

19.3.4 SQL 语言与 Active Record 模块.....217

19.3.5 使用 like 查询子句.....218

19.3.6 构造返回记录的子集.....219

19.3.7 获取字段统计.....221

19.3.8 范围函数.....221

19.3.9 编写自己 SQL 语句.....222

19.3.10 重新加载数据.....224

19.3.11 更新现有行记录.....224

19.3.12 方法 save、save!、create 和

create!.....225

19.3.13 删除行记录.....226

19.4 干预跟踪进程.....227

19.4.1 成组相关回调.....228

19.4.2 观察器.....231

19.4.3 观察器实例化.....232

19.5 数据库事务.....232

19.6 本章小结.....235

第 20 章 行为调度和行为控制.....237

20.1 分派请求到控制器.....237

20.1.1 REST: 表述性状态转移.....238

20.1.2 添加附加行为.....243

20.1.3 嵌套资源.....243

20.1.4 浅路由嵌套.....243

20.1.5 选择数据表述.....244

20.1.6 测试路由.....245

20.2 处理请求.....246

20.2.1 行为方法.....246

20.2.2 控制器环境.....246

20.2.3 用户响应.....248

20.2.4 呈现模板.....248

20.2.5 发送文件和其他数据.....251

XVIII

20.2.6 重定向	253	21.6.8 局部模板和控制器	283
20.3 持续请求的对象和操作	255	21.7 本章小结	283
20.3.1 Rails 会话	255	第 22 章 缓存	285
20.3.2 会话存储	257	22.1 页面缓存	285
20.3.3 比较会话存储选项	258	22.2 让页面失效	287
20.3.4 会话逾期与清除	259	22.2.1 显式地让页面失效	287
20.3.5 闪存：行为间通信	259	22.2.2 挑选缓存存储策略	288
20.3.6 过滤器	260	22.2.3 隐式地让页面失效	289
20.3.7 前置和后置过滤器	260	22.2.4 让基于时间的缓存页面失效	290
20.3.8 过滤器继承	261	22.2.5 正确处理客户端缓存	291
20.4 本章小结	261	22.2.6 过期头	291
第 21 章 Action View 模块	263	22.2.7 最后的修改和 ETag 支持	291
21.1 使用模板	263	22.3 片段缓存	292
21.1.1 模板存放的位置	263	22.4 本章小结	296
21.1.2 模板运行的环境	264	第 23 章 数据迁移	297
21.1.3 模板包含的内容	264	23.1 创建和运行迁移	297
21.2 生成表单	265	23.2 剖析迁移	299
21.3 处理表单	267	23.2.1 字段的类型	300
21.4 上传文件到 Rails 应用程序	268	23.2.2 重命名字段	301
21.5 使用帮助程序	271	23.2.3 修改字段	302
21.5.1 自定义的帮助程序	272	23.3 表的管理	302
21.5.2 格式和链接帮助程序	272	23.3.1 表的创建选项	303
21.5.3 格式帮助程序	272	23.3.2 表的重命名	304
21.5.4 链接到其他页面和资源	274	23.3.3 rename_table 方法的问题	304
21.6 用页面布局和局部模板减轻维护工作	277	23.3.4 定义索引	305
21.6.1 布局	277	23.3.5 主键	306
21.6.2 放置布局文件	278	23.3.6 没有主键的表	306
21.6.3 传递数据到布局	279	23.4 高级迁移	306
21.6.4 局部页面模板	281	23.4.1 使用原生 SQL	307
21.6.5 局部模板和集合	282	23.4.2 扩展迁移	307
21.6.6 共享模板	283	23.4.3 自定义消息和基准测试程序	309
21.6.7 局部模板与布局	283	23.5 当迁移变糟时	309
		23.6 迁移外的模式管理	310

23.7 本章小结.....	311	25.3 用 Bundler 管理包依赖关系.....	325
第 24 章 非浏览器应用.....	312	25.4 用 Rack 实现与 Web 服务器的交互.....	327
24.1 用 Active Record 开发独立应用程序.....	312	25.5 自动执行任务工具 Rake.....	330
24.2 使用 Active Support 库功能.....	313	25.6 Rails 包依赖关系揭秘.....	331
24.2.1 核心扩展.....	313	25.7 本章小结.....	333
24.2.2 附加的 Active Support 类.....	315	第 26 章 Rails 插件.....	334
24.2.3 使用 Action View 帮助程序.....	317	26.1 信用卡业务处理插件 Active Merchant.....	334
24.3 使用 Active Resource 开发远程应用程序.....	317	26.2 节约带宽的插件 Asset Packager.....	335
24.3.1 访问和更新简单属性.....	317	26.3 用 Haml 美化标记语言.....	337
24.3.2 关系和集合.....	318	26.4 用 JQuery 少写多做.....	339
24.3.3 汇总整理.....	320	26.5 在 RailsPlugins.org 上找出更多.....	342
24.4 本章小结.....	321	26.6 本章小结.....	343
第 25 章 Rails 包依赖关系.....	322	第 27 章 整装进发.....	344
25.1 用构建器生成 XML.....	322	参考文献.....	345
25.2 用 ERb 生成 HTML.....	323		

第一部分 起 步

第 ① 章

安装 Rails

本书第一部分将介绍 Ruby 语言和 Rails 框架。但是在继续下面的内容之前，首先要安装并确保它们能够正确运行。

为了让 Rails 运行起来，需要以下工具软件：

- 一个 Ruby 解释器。Rails 是用 Ruby 写的，而且将来你也需要用 Ruby 来编写应用程序。目前 Rails 需要 Ruby 1.8.7 版或 1.9.2 版，而与 Ruby 1.8.6 版和 1.9.1 版相冲突。
- 一个称作 RubyGems 的打包系统。本书使用的是 RubyGems 1.3.7 版。
- Ruby On Rails。本书 beta 版采用的是 Rails 3 版（准确地说是 Rails 3.0.5 版）。
- 一些必要的库，这取决于操作系统。
- 数据库。在本书中使用 SQLite 3 数据库。

如果只是开发本地应用程序，这些就足够了（当然，还需要一个编辑器，稍后单独讨论编辑器的问题）。不过，如果要部署应用程序，就（至少）还需要安装一个生产环境的 Web 服务器（最低限度的），以及一些能够保证 Rails 运行效率的支持代码。第 18 章会用整整一章来讨论这个话题，现在就先不讨论它了。

那么，怎么安装这些工具呢？这要取决于操作系统，下面逐一介绍。

1.1 Windows 上的安装

在 Windows 上安装 Ruby，最简单的方法是通过 RubyInstaller 安装包^①。请确认下载的

① <http://rubyinstaller.org/>

版本号是 1.8.7 或更新版本的 Ruby。我们使用 Ruby 1.8.7 版来测试本书中的示例（2010-01-10 patchlevel 249）。

基本安装是很简单的。下载之后，选择 Run → Next 按钮，单击“I accept the License”复选框（当然是在仔细阅读之后），单击 Next 按钮，并且选中“Add Ruby executables to your PATH”复选框。然后选择 Install → Finish 按钮。

现在通过选择单击 Windows “开始→运行”，输入“cmd”命令，然后单击“确定”来打开一个命令行运行窗口。

RubyInstaller 已包括了 RubyGems，重要的是，你需要检验版本是否是比 1.3.6 版更新。可以通过以下命令来查看 RubyGems 的版本：

```
gem -v
```

如果输出显示是 1.3.5 版或者更早的版本。可以通过以下的命令来升级 RubyGems：

```
gem update --system
gem uninstall rubygems-update
```

下面安装 SQLite 3[⊖]，在下载页面中可以找到已预编译的 Windows 二进制安装包。

请下载并解压缩以下文件：

- 基于命令行的 shell，通过它可以访问并修改 SQLite 数据库。
- SQLite 库的 DLL。

从压缩文件中复制这些文件到 C:\Ruby\bin 目录下。结果如下：

Directory of C:\Ruby\bin

```
02/11/2011  06:30 PM          3,744 sqlite3.def
02/11/2011  06:30 PM     511,383 sqlite3.dll
02/11/2011  06:31 PM     530,762 sqlite3.exe
              3 File(s)      1,045,889 bytes
```

现在接着安装 Ruby 的绑定 SQLite 3 和 Ruby 本身：

```
gem install sqlite3
gem install rails
```

在这一步完成后，Rails 就可以运行了。但是，你还需要了解的一点是：本书的样例是基于 Mac 机的，虽然 Ruby 和 Rails 的命令在不同操作系统上都是一样的，但是 UNIX 操作系统本身含有的命令与 Windows 不同。本书仅包含两条 UNIX 命令：第一条是 ls-a，对应的 Windows 命令是 dir/w；第二条是 rm，对应的 Windows 命令是 erase。

也将选择性在 12.2 节中使用 curl 命令，可以从 curl.haxx.se[⊖]下载。

对于 Windows 用户的安装已经完成了，可以直接跳到 1.4 节。

⊖ <http://www.sqlite.org/download.html>

⊖ <http://curl.haxx.se/download.html>

1.2 Mac OS X 上的安装

虽然在 OS X 10.4.6 (Tiger) 上已经默认安装了 Ruby, 甚至在 OS X 10.5 (Leopard) 上面还安装了 Rails, 但是一般包含的版本较旧。需要做一些升级工作。

Tiger 用户还需要升级 SQLite 3。这可以通过编译源码的方式来完成(没有听起来那么复杂)。具体方法可以在 <http://www.sqlite.org/download.html> 找到。

另一种方法是通过 MacPorts[⊖] 包来安装 SQLite 3。虽然它的安装说明看起来挺难, 但实际上每一步都很简单: 执行一个安装程序, 运行另一个安装程序, 加两行代码到文件中, 再运行另一个安装程序, 最后执行一条命令。看起来使用 MacPorts 来安装并不比从源代码编译更容易, 但许多人觉得这是值得的, 因为以后安装软件包简单得和执行一条命令一样。如果已经安装了 MacPorts, 就先升级 SQLite 3:

```
sudo port upgrade sqlite3
```

接下来, 比 Snow Leopard 操作系统更早的 OS X 的用户需要先把他们的 Ruby 版本升级到 1.8.7 版。同样, 这可以通过 MacPorts 来完成:

```
sudo port install ruby
sudo port install rb-rubygems
```

所有的 OS X 用户都可以使用下面的命令来完成升级过程。如果你刚安装了 MacPorts, 请通过打开一个新的 shell, 执行 env 命令来验证路径和变量的修改都已经生效。如果没有安装 MacPorts, 请先安装 Apple 的 XCode 开发人员工具 (Leopard 上要求比 3.2 更高的版本, Tiger 上则要求比 2.4.1 更高的版本), 可以在 Apple Developer Connection 网站或 Mac OS X 安装光盘中找到它。

```
sudo gem update --system
sudo gem uninstall rubygems-update
sudo gem install rails
sudo gem install sqlite3
```

这样安装之后, 操作系统上可能会有两个版本的 Ruby, 这里很重要的一点是: 需要检验运行相关工具的版本的正确性。可以通过使用以下的命令来进行检验:

```
which ruby irb gem rake
```

你所需要做的只是检验每个工具软件是否已安装在同一目录, 通常在 /opt/local/bin 目录下。如果你发现不同的工具软件具有不同路径, 请检查 PATH 环境变量的设置是否正确, 并且 / 或者重新安装与所需 Ruby 版本不匹配的工具软件。

可以使用 `sudo port install rb-rubygems`, 或者按照 RubyGems 网站[⊕] 上的说明 (网页最下面的步骤 1~3; 很可能还需要通过 `sudo` 预先设定 `ruby setup.rb` 命令的前缀) 来重新安装 RubyGems。

可以通过以下命令重新安装 Rake:

```
sudo gem install rake
```

⊖ <http://www.macports.org/install.php>

⊕ <http://rubygems.org/pages/download>

下面的步骤很少用到，只有在确实遇到麻烦时才需要。可以通过独立的 Ruby 程序来运行下面的代码，以验证 sqlite3-gem 接口绑定到了哪一个版本的 sqlite 3 上：

```
require 'rubygems'
require 'sqlite3'
tempname = "test.sqlite#{3+rand}"
db = SQLite3::Database.new(tempname)
puts db.execute('select sqlite_version()')
db.close
File.unlink(tempname)
```

对于 OS X 用户的安装已经完成，你可以直接跳到 1.4 节。

1.3 Linux 上的安装

打开 Linux 平台自带的包管理系统，例如：apt-get、dpkg、portage、rpm、rug、synaptic、up2date 或 yum。

第一步是安装必需的依赖库。下面的指令适合于 Ubuntu 10.04 和 Lucid Lynx 上的安装，你可以根据自己的需要进行修改，以适合你的 Linux 操作系统：

```
sudo apt-get install build-essential libopenssl-ruby libfcgi-dev
sudo apt-get install ruby irbrubygems ruby1.8-dev
sudo apt-get install sqlite3 libsqlite3-dev
```

在进行下一步之前，要先确认 RubyGems 版本不低于 1.3.6，可以使用 `gem -v` 来查看版本信息。

在 Linux 上升级 RubyGems

升级 RubyGems 可以有不同方法。遗憾的是，这都取决于所安装的 RubyGems 版本和 Linux 操作系统的发行版本，并不是所有的方法都能升级成功。请尝试下面所述的方法，直到找到一个行得通的解决方案。

- 使用 gem 升级系统：

```
sudo gem update --system
```

- 使用 gem 专门升级有问题的系统：

```
sudo gem install rubygems-update
sudo update_rubygems
```

- 使用 rubygems-update 提供的 setup.rb：

```
sudo gem install rubygems-update
cd /var/lib/gems/1.8/gems/rubygems-update-*
sudo ruby setup.rb
```

- 最后，尝试从源代码进行安装：

```
wget http://rubyforge.org/frs/download.php/69365/rubygems-1.3.6.tgz
tar xzf rubygems-1.3.6.tgz
cd rubygems-1.3.6
sudo ruby setup.rb
```


下面将安装 Rails 框架和 SQLite 3 数据库:

```
sudo gem install rails
sudo gem install sqlite3
```

在执行最后一条命令时, 需要选择在平台里安装哪一个 gem, 只须简单地选择最新(最上面)的在圆括号中包含 ruby 字样的 gem 就行。

安装完成后, 请尝试运行 `rails -v` 命令, 如果系统找不到 rails 命令, 还需要将 `/var/lib/gems/1.8/bin` 加入 PATH 环境变量中。可以在 `.bashrc` 文件里加入:

```
export PATH=/var/lib/gems/1.8/bin:$PATH
```

这样, 我们就完成了对 Windows、Mac OS X 和 Linux 安装的介绍。之后的指令对于这 3 种操作系统都是通用的。

1.4 选择一个 Rails 版本

前面的指令介绍了如何安装最新版本的 Rails。但有时, 你可能并不想使用最新版本。例如, 可能在本书出版的时候已经有了新版本的 Rails, 而你想确信书中的例子和你的运行结果完全吻合; 还有可能你在一台机器上进行开发, 但要把应用程序部署到另一台已经有 Rails 的机器上, 而你无权去更改它的版本。

如果你正好处于这两种情况之一, 那么首先需要了解几件事。第一, 可以使用 `gem` 命令查看所有已经安装的 Rails 版本:

```
gem list --local rails
```

可以使用 `rails --version` 来查看哪一个版本的 Rails 是默认运行。它应该是 3.0.5 或更高的版本。

也可以使用 `gem` 命令来安装不同版本的 Rails。根据操作系统的不同, 可能需要在 `gem` 命令前面加上 `sudo`。

```
gem install rails --version 3.0.5
```

多个版本的 Rails 并不会带来什么好处——除非有什么办法能从中选出一个。而我们恰恰有这么一种方法。可以在 `rails` 命令和第一个参数之间, 通过添加一个用下划线包围着的完整的版本号来选择需要运行的 Rails 版本:

```
rails _3.0.5_ --version
```

这种方法在创建新的 Rails 应用程序时特别方便, 因为一旦用一个特定 Rails 版本创建应用程序后, 它就会一直使用该版本的 Rails (即使在系统里又安装了更新版本的 Rails), 直到你决定进行升级。升级时, 只须修改在应用程序的根目录里 `Gemfile` 的版本号, 再运行 `bundle install` 即可。25.3 节将介绍这条命令。

1.5 设置开发环境

日常编写 Rails 程序的工作相当简单。但是每个人都有不同的工作方式，下面是我们的工作方式。

1.5.1 命令行

我们的很多工作都用命令行来完成。虽然有越来越多的 GUI 工具可以帮助我们生成和管理 Rails 应用程序，但我们感觉命令行仍然是最好用的工具。值得花一点时间去熟悉操作系统上的命令行：学会如何在其中编辑命令，如何搜索和编辑前面输入过的命令，如何快速补全输入的文件名和命令。

我的 IDE 在哪儿

如果你是从 C# 或者 Java 语言转向 Ruby 和 Rails 的，也许你会惊讶地问 Rails 的 IDE 在哪儿？我们都知道，要是没有一个至少 100MB 的 IDE 支持着每一次的输入，我们是无法开发一个现代应用程序的。对于你们这些现代人，我们的建议是：请先坐下来，给自己找上一大堆关于框架的参考文档，再加上 1000 页的“轻松学 ×××”的书——如果这样会让你感到舒服的话。

也许你惊讶于大多数的 Rails 开发者并不使用 Ruby 或 Rails 的完备 IDE（虽然已有些类似于 IDE 的开发环境）。但大多数 Rails 开发者都使用普通的旧式编辑器，这一点儿也不会带来什么问题。当使用那些缺乏表达力的编程语言时，程序员需要依赖 IDE 来进行大量的重复劳动：代码生成、辅助导航、增量编译（以便尽早发现程序中的错误）等。

当使用 Ruby 时，这些支持大多变成不必要的了。TextMate 和 BBEdit 之类的编辑器能够提供 90% 所需的功能，但它比起那些 IDE 可要轻量多了。在我看来，相比于 IDE，这些编辑器唯一欠缺的就是对于重构的支持。

Tab 补全是 UNIX shell 具有的标准功能，例如 Bash 和 zsh。只需要输入文件名的前几个字符，然后单击 Tab 键，shell 会自动查找和补全相匹配的文件名。

1.5.2 版本控制

我们把所有工作都保存在版本控制系统里（现在我们用的是 Git）。使用 Git 创建一个新的 Rails 版本控制项目；一旦通过测试，当创建新的 Rails 项目和提交更改时，可以检查 Git 中的对应项目。一般而言，我们在每小时会多次提交给存储库。

如果项目由多人一起开发，可以考虑搭建一个持续集成（Continuous Integration, CI）系统：每当有人验证文件是否有修改过时，CI 系统就会检查应用程序的一个新副本，然后运行所有测试。通过这种简单的方式，如果有意外的损坏发生，可以马上注意到。此外，CI 系统也可以让客户随时试用应用程序的最新版本。通过这种使用 CI 增加项目透明度的方式，可以确保项目不出现大的问题。

1.5.3 编辑器

我们使用程序员常用的编辑器来编写 Ruby 程序。这些年来我们发现，不同的编辑器适用于不同的语言和环境。譬如，Dave 最初用 Emacs 来书写这一章的内容，因为他感觉 Emacs 的 Filladappt 模式非常方便：当输入文字时，它会巧妙地格式化 XML。Sam 使用 Vim 来修改。也有很多人觉得 Emacs 和 Vim 都不够理想，他们更喜欢用 TextMate。虽说选择编辑器是一件私事，但对于合适的 Rails 编辑器，我们还是可以为你提供一些建议：

- 支持 Ruby 和 HTML 的语法高亮显示，最好也支持 .erb 文件的高亮显示（这是 Rails 使用的一种文件格式，在 HTML 中嵌入 Ruby 代码片段）。
- 支持 Ruby 源代码的自动缩进和重新缩进。这不仅仅为了美观：如果编辑器在你输入代码的同时还能够进行自动程序缩进，就可以很容易找出代码中的错误嵌套；能够支持重新缩进在对代码进行重构或是移动时也是很重要的。（TextMate 可以在从剪贴板粘贴代码时进行格式重排，这是一个很便利的功能。）
- 支持插入常用 Ruby 和 Rails 语法结构。在开发的过程中你会编写很多短小的方法，最好是只按一两次键就让 IDE 帮你创建方法骨架，这样就可以专注于编写里面真正的功能代码。
- 支持良好的文件浏览。正如你将看到的，Rails 应用程序分布在很多个文件上。在书写任何代码之前，一个新建的 Rail 应用程序就已经包含了分散在 34 个目录中的 46 个文件。这是编写代码之前的状况。你的开发环境应该能够帮助你快速地在这些文件之间切换——你可能会在控制器中添加一行代码，从而加载一个值；然后切换到视图，添加一行代码来显示这个值；然后又切换到测试，添加一个测试方法来验证一切是否工作正常。像 Notepad 这样的编辑器只允许遍历“打开文件”对话框来选择要编辑哪个文件，这样的编辑器是无法满足要求的。我们希望编辑器同时具备两种功能：在旁边有一个树状的文件视图，通过几个快捷键帮助我们根据名字找到文件；在编辑器内部具有一定的智能功能，知道如何，譬如说，从控制器转到与之相对应的视图。
- 支持名称补全。Rails 中采用的名称都比较长，一个好的编辑器允许只输入前几个字符，然后通过一个快捷键提示可能的补全方案。

我们很遗憾地不能列举所有的编辑器，这是因为我们也只使用过其中的一些，无疑，我们可能漏掉一些人偏爱的编辑器。不过，为了在初学时选择编辑器，除了 Notepad 之外，下面这些建议可以作为参考。

- 在 Mac OS X 上面开发 Ruby/Rails 的首选是 TextMate (<http://macromates.com/>)。
- Mac OS X 上的 XCode 3.0 有一个 Organizer，它能提供你需要的大多数功能。你能够在 <http://developer.apple.com/tools/developonrailsleopard.html> 上找到如何在 Leopard 上使用 Rails 的教程。
- 若想在 Windows 平台使用 TextMate，E-TextEditor (<http://e-texteditor.com/>) 提供了“The Power of TextMate on Windows”。
- Aptana RadRails (<http://www.aptana.com/products/radrails>) 是一个集成的 Rails 开发环境，它运行在 Aptana Studio 和 Eclipse 平台中，可以运行于 Windows、Mac OS X 和 Linux 等操

作系统上，该工具于 2006 年被评为基于 Eclipse 的最佳开源开发者工具。2007 年，该项目落户在 Aptana。

- NetBeans IDE 6.5 (<http://www.netbeans.org/features/ruby/index.html>) 可以运行于 Windows、Mac OS X、Solaris 和 Linux 系统。它提供两种方式，你可以下载捆绑了 Ruby 的 NetBeans IDE，也可以先下载 NetBeans IDE，然后再下载 Ruby 包。除了支持 Rails 2.0、Rake 目标和数据库迁移外，它还支持 Rails 代码生成器的图形化向导，以及 Rails 行为到相应视图的快速导航。
- jEdit (<http://www.jedit.org/>) 是一个功能完备的编辑器，提供了对 Ruby 的支持，并且支持插件扩展。
- Komodo (<http://www.activestate.com/Products/Komodo/>) 由 ActiveState 开发的 IDE，支持包括 Ruby 在内的多种动态语言。
- RubyMine (<http://www.jetbrains.com/ruby/features/index.html>) 是 Ruby 的一个商业 IDE，它还有一个用于教育的免费版本和一个开源项目。它可以运行于 Windows、Mac OS X、Solaris 和 Linux 系统上。

最后还有几点建议：找一位与你使用同样操作系统的、有经验的开发者，问问他用的是什么编辑器；在最终选定一个编辑器之前，用一周左右时间先试试其他的编辑器。

1.5.4 桌面

我们打算告诉你在使用 Rails 进行开发的时候应该如何组织你的桌面，这里只介绍我们自己的做法。

大部分时间里，我们都是在编写代码、运行测试，以及在浏览器里查看应用程序的运行结果。所以，在主开发桌面上总是有一直打开的一个编辑器窗口和一个浏览器窗口。此外，我们时常要查看应用程序生成的日志，所以终端窗口也总是开着，在跟踪日志文件更新时，使用 `tail -f` 命令来滚动显示其内容。我们通常把这个窗口的字体设置得很小，这样它就不必占用太多地方；如果看到什么有趣的东西一闪而过，我们会把字体放大来细细查看。

创建你自己的 Rails API 文档

你可以自己创建一份完整的 Rails API 文档，只要在命令行输入下列命令即可：

```
rails_apps> rails: new dummy_app
rails_apps> cd dummy_app
dummy_app> rake doc:rails
```

最后一个步骤需要花一点时间。当所有命令运行结束之后，在 `/doc/api` 目录树下就有了一份完整的 Rails API 文档。我们建议你把这个文件夹移到桌面上，然后删除 `dummy_app` 目录和其子目录。

要查看文档，只需用浏览器打开 `doc/api/index.html` 即可。

我们还需要经常通过浏览器来查看 Rails API 文档。在前面的介绍中已经提到过，用 `gem`

`server` 命令可以在本地运行一个其中包含 Rails 文档的 Web 服务器。这用起来很方便，唯一令人遗憾的是：它把 Rails 文档分割成了互不相关的几个文档树。如果你能够上网，可以到 <http://api.rubyonrails.org> 去查阅 Rails 文档——所有的文档都放在一个地方。

1.6 Rails 和数据库

本书中所用的例子都是用 SQLite 3 开发的（3.6.16 版本或与之相应的版本）。如果直接使用我们提供的代码，你最好也安装一个 SQLite 3；不过在 Rails 中使用其他的数据库也不是什么大问题：你可能需要对代码中的显式 SQL 语句稍作调整，但 Rails 已经很好地消除了应用程序中大部分某些数据库特有的 SQL 语句。

如果你打算连接 SQLite 3 之外的数据库。Rails 可以与 DB2、MySQL、Oracle、Postgres、Firebird 以及 SQL Server 数据库一起工作。使用除 SQLite 3 之外的这些数据库，首先必须安装数据库驱动程序——这是一个 Ruby 库，Rails 可以通过它连接并访问数据库引擎。本节就要介绍如何安装数据库驱动程序。

数据库驱动程序都是用 C 语言编写的，并且多以源码形式发布。如果你不想通过编译源码来构建数据库驱动程序，就请认真查看驱动程序的网站，很多时候作者也会同时发布二进制版本。

如果实在找不到二进制版本，或者你更愿意自己编译源码，就需要机器上的开发环境以构建对应库。在 Windows 平台上，这通常意味着需要安装 Visual C++；在 Linux 上，需要 GCC 和其他相关软件（不过这些东西很可能都已经安装好了）。

在 OS X 平台上，需要安装开发者工具（操作系统的安装包里有这套工具，不过默认情况下是不安装的）。另外别忘了，要把数据库驱动程序安装到正确的 Ruby 版本上。如果你已经安装了自己的 Ruby 版本，并且绕过了内置的 Ruby，请千万记得当编译和安装数据库驱动程序时，把你安装的 Ruby 版本放在系统路径的最前面。可以用 `which ruby` 命令来确保当前运行的 Ruby 不是位于 `/usr/bin` 的版本。

下面列出了 Rails 支持的各种数据库适配器，并给出了各个驱动程序的首页地址。

DB2	http://raa.ruby-lang.org/project/ruby-db2 或 http://rubyforge.org/projects/rubyibm
Firebird	http://rubyforge.org/projects/fireruby/
MySQL	http://www.tmtm.org/en/mysql/ruby
Oracle	http://rubyforge.org/projects/ruby-oci8
Postgres	http://rubyforge.org/projects/ruby-pg
SQL Server	http://github.com/rails-sqlserver
SQLite	http://rubyforge.org/projects/sqlite-ruby

Postgres 适配器的纯 Ruby 语言版本是可用的。可以从 Ruby-DBI 页面（<http://rubyforge.org/projects/postgres-pr>）下载 `postgres-pr`。

MySQL 和 SQLite 适配器也可以下载后使用，都已包含在 RubyGems 包里（分别为 `mysql` 和 `sqlite3`）。

1.7 本章小结

- 安装（或升级）了 Ruby 语言。
- 安装（或升级）了 Rails 框架。
- 安装（或升级）了 SQLite 3 数据库。
- 选择了一个编辑器。

至此，我们已经安装了 Rails，并且可以使用它了。现在就进入下一章，在那里我们将创建第一个 Rails 应用程序。

第 2 章

即时满足

在本章中，我们将学习：

- 如何创建一个新的应用程序
- 启动服务器
- 通过浏览器来访问服务器
- 创建动态的网页内容
- 添加超链接
- 把数据从控制器传送到视图

现在编写一个简单的 Web 应用程序，以验证 Rails 已经成功地安装在机器上了。在此过程中，我们还会带领读者一睹 Rails 应用程序的工作方式。

2.1 新建一个应用程序

安装了 Rails 框架之后，你同时也得到了一个新的命令行工具：**rails**。这个工具可以用于创建你编写的每个新 Rails 应用程序。

为什么我们需要这个工具，而不直接使用编辑器从头开始编写应用程序呢？我们当然可以这样做，因为不管怎么说，Rails 应用程序只是由 Ruby 源代码组成的。然而通过一个 Rails 命令行，Rails 可以在幕后做许多事情，让我们只需要做最少量的配置即可运行一个应用程序。为了让这些幕后工作生效，Rails 必须能够找到应用程序中所需的各种组件。正如我们稍后将会看到的（在 18.1 节中），为了实现这些功能，在创建 Rails 应用程序时，还需要创建某种固定的目录结构，以此将代码放在合适的目录下。**rails** 命令可以帮助我们创建这一目录结构，并且生成一些标准的 Rails 代码。

现在创建第一个 Rails 应用程序：打开 shell 窗口，进入文件系统中保存应用程序目录结构的地方（此处由你来指定）。在这个例子中，将把项目创建在一个名为 **work** 的目录之下。然后，在这个目录中用 **rails** 命令创建一个名为 **demo** 的应用程序。在这里请注意：如果已经存在一个名叫 **demo** 的目录，Rails 会询问你是否要覆盖已有的文件。（请注意：如果你想应用指定的 Rails 版本来创建应用程序，请按 1.4 节所述那样来做。）

```
rubys> cd work
work> rails new demo
create
create README
create .gitignore
create Rakefile
:      :
```

```
create tmp/pids
create vendor/plugins
create vendor/plugins/.gitkeep
work>
```

上述命令创建了一个名为 `demo` 的目录。进入这个目录，列出它的全部内容（在 UNIX 中使用 `ls` 命令，在 Windows 中使用 `dir` 命令），你将看到如下的一堆文件和子目录：

```
work> cd demo
demo> ls -p
app/          config.ru  doc/       lib/       public/    README    test/      vendor/
config/       db/       Gemfile    log/       Rakefile   script/    tmp/
```

突然面对那么多目录（还有它们包含的文件）也许会让你感到有点儿不知所措，不过现阶段我们可以忽略它们中的大多数。在本章中，我们直接使用到其中的一个目录，即 `app` 目录，我们会在这个目录之下编写应用程序。

只须包含这些文件，就可以在本地服务器上运行新建的 Rails 应用程序。所以无需多言，就让我们马上动手运行这个 `demo` 程序：

```
demo> rails server
=> Booting WEBrick
=> Rails 3.0.5 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2010-11-14 10:53:35] INFO  WEBrick 1.3.1
[2010-11-14 10:53:35] INFO  ruby 1.8.7 (2010-08-16) [i686-darwin9.8.0]
[2010-11-14 10:53:40] INFO  WEBrick::HTTPServer#start: pid=6044 port=3000
```

这里所运行的本地 Web 服务器是由你安装的。WEBrick 服务器是一个用纯 Ruby 编写的 Web 服务器，它可以运行在比 Ruby 1.8.1 版本更高的 Ruby 版本上。如果系统中安装了其他 Web 服务器（而且 Rails 能够正确地找到它），那么 `rails server` 命令可能启动该 Web 服务器而不是 WEBrick。对此，可以通过给 `rails` 命令添加以下选项，指定 Rails 使用 WEBrick：

```
demo> rails server webrick
```

从程序启动后跟踪输出信息的最后一行就可以看到，我们在 3000 端口上启动了一个 Web 服务器。地址的 0.0.0.0 这一部分是指 WEBrick 将从所有接口接收连接请求。在 Dave 的 OS X 系统上，这意味该服务器可以接收来自本地软件协议接口（127.0.0.1 和 ::1）和它的 LAN 的连接请求。打开浏览器，访问 `http://localhost:3000`，就会看到这个应用程序，其如图 2.1 所示。

在已启动的那台服务器的窗口里会输出一些跟踪信息，这表明你已启动了该应用程序。我们先让该服务器一直在命令行窗口中运行。当编写应用代码并在浏览器中运行这些代码时，可以通过控制台窗口来跟踪关于连接请求的相关信息。如果要关闭应用程序，可以在命令行窗口中按 `Ctrl+C` 快捷键来终止 WEBrick（现在别这样操作——我们还需要用到这个应用程序）。

现在，我们已经让这个新的应用程序运行起来了，但还没有编写自己的代码。接下来改变这种情况。

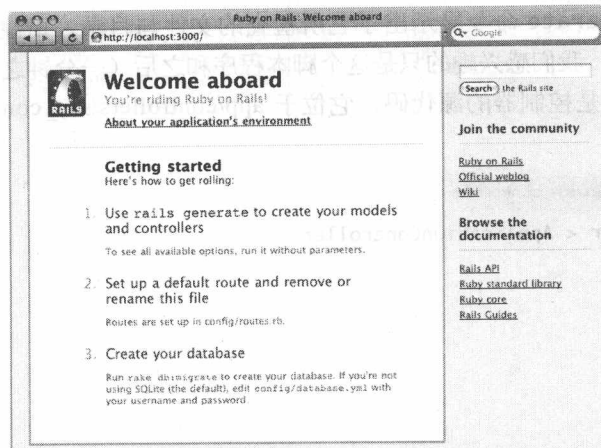


图 2.1 新建的 Rails 应用程序

2.2 Hello, Rails !

没有办法——按惯例，每次要试用一个新系统时，都得写一个“Hello, World!”程序。现在就让我们创建一个简单的 Rails 应用程序，以把我们诚挚的问候发送到浏览器上。之后我们会加入显示当前时间和添加链接的功能。

正如第 3 章所介绍的，Rails 是一个 MVC（模型 - 视图 - 控制器）框架。Rails 接收来自浏览器的请求，对请求进行解读以找到合适的控制器，再调用控制器中合适的方法。然后，控制器又调用一个特定的视图，将结果显示给用户。好消息是，Rails 已经帮助我们完成了把这些动作链接在一起的大部分任务。现在，为了实现这个简单的“Hello, World!”应用程序，需要编写一个控制器和一个视图，还需要设置一条路径来连接它们两个。不需要编写模型，因为不需要处理任何数据。现在就从控制器开始吧。

就像用 rails 命令新建一个 Rails 应用程序一样，也可以借助一条脚本命令来为该项目新建一个控制器。这条脚本命令是 rails generate。所以，要创建一个名为 say 的控制器，只需要在 demo 目录中运行这条脚本命令，将我们想要创建的控制器名称和我们限定这个控制器支持的动作名称传递过去即可：

```
demo> rails generate controller Say hello goodbye
create app/controllers/say_controller.rb
route get "say/goodbye"
route get "say/hello"
invoke erb
create app/views/say
create app/views/say/hello.html.erb
create app/views/say/goodbye.html.erb
invoke test_unit
create test/functional/say_controller_test.rb
invoke helper
create app/helpers/say_helper.rb
invoke test_unit
create test/unit/helpers/say_helper_test.rb
```

这条 `rails generate` 命令显示出了它所检查的文件与目录，请注意它是何时添加 Ruby 源代码和目录的。现在，我们感兴趣的只是这个脚本程序和之后（一分钟之后）的 `.html.erb` 文件。

我们首先要关注的是控制器的源代码，它位于 `app/controllers/say_controller.rb` 文件中，让我们来看看这个文件：

```
work/demo1/app/controllers/say_controller.rb
```

```
class SayController < ApplicationController
  def hello
  end

  def goodbye
  end
end
```

非常迷你，不是吗？`SayController` 是从 `ApplicationController` 继承而来的一个类，因此它自动继承所有默认的控制器行为。那么这段代码是做什么的呢？现在，它其实什么也没做——我们只是新建了一个空的动作方法，即 `hello`。为理解这个方法以这种方式命名的原因，我们需要先来看看 Rails 处理请求的方式。

2.2.1 Rails 和 URL 请求

和其他的 Web 应用一样，一个 Rails 应用程序在用户看来要与一个 URL 相关联。当你把浏览器指向这个 URL 时，你就像开始与这个应用程序代码进行对话一样，应用程序负责生成应答信息。

现在让我们试试这个程序。在浏览器的地址里输入：`http://localhost:3000/say/hello`（请注意：在测试环境下，路径中没有包含应用程序名——我们直接导向到控制器）。你会看到类似于图 2.2 那样的结果。



图 2.2 用于样例的模板

2.2.2 第一个动作

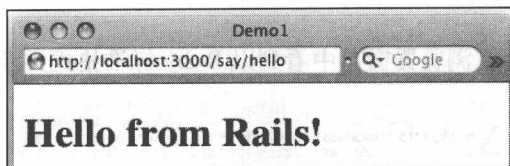
从这里可以看出，Rails 应用程序不仅通过 URL 连接到控制器，同时还指出 Rails 下一步需要做些什么，即告诉 Rails 需要显示什么内容。这里就需要用到视图。还记得我们运行脚本命令 `rails generate` 来新建控制器时的情境吗？那个脚本为该应用程序共生成了 6 个文件和 1 个新目录，其中目录用来存放控制器视图的模板文件。在这里，创建了一个名为 `say` 的控制器，因此视图就应该位于 `app/views/say` 目录中。

默认情况下, Rails 会寻找与当前动作(action)同名的文件中的模板。在我们的例子中,这就意味着要替换一个名为 hello.html.erb 文件,其位于 app/views/say/ 目录(为什么是 .html.erb? 我们稍后就会解释)。现在,只需在其中填入基本的 HTML 代码。

```
work/demo1/app/views/say/hello.html.erb
```

```
<h1>Hello from Rails!</h1>
```

保存 hello.html.erb 文件,刷新浏览器窗口,你就应该看到一句友好的问候。



到目前为止,我们已经了解了在 Rails 应用程序树状目录里的两个文件:我们看到了什么是控制器,然后修改了一个模板以便在浏览器上显示页面。这些文件都位于 Rails 层次结构中的标准位置:控制器在 app/controllers 目录下,视图在 app/views 中各自的子目录下,如图 2.3 所示。

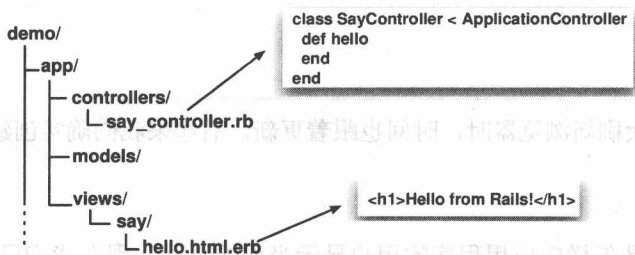


图 2.3 控制器和视图的标准位置

2.2.3 创建动态网页

到目前为止,该 Rails 应用程序还相当单调——它只能显示一个静态页面。为了给它增加一点动感,我们让它在每次显示页面时也显示当前的时间。

为了实现这一效果,要对视图的模板文件稍加修改——现在它需要以字符串的形式把“时间”这个信息包含进来。这时就出现了两个问题。首先,如何在模板中添加动态内容?其次,要显示的“时间”信息从哪里来?

2.2.4 动态内容

在 Rails 中,有多种方式可以创建动态模板。最常用的方法是将 Ruby 代码嵌入模板中,这也是在这里将会用到的。这也就是要把模板文件命名为 hello.html.erb 的原因: .html.erb 后缀告诉 Rails,需要借助 ERb 系统对文件的内容进行扩展。

ERb 是一个过滤器,会包含在 Rails 的默认安装里:输入的是 .erb 文件,过滤器输出的是经过转换的内容。在 Rails 中,输出文件通常是 HTML 格式,但它也可以是其他任何东西。普通的

内容经过 ERb 过滤器后没有任何变化。但 `<%=` 和 `>` 符号之间的内容则会被看做 Ruby 代码并执行。执行的结果将转换为字符串，并在文件中用 `<%=...%>` 序列替代该值。譬如，在 `hello.html.erb` 中加入下列内容来显示当前时间：

```
work/demo2/app/views/say/hello.html.erb
```

```
<h1>Hello from Rails!</h1>
▶ <p>
▶   It is now <%= Time.now %>
▶ </p>
```

当刷新浏览器时，你会在浏览器窗口中看到以 Ruby 标准格式显示的时间^①：



你可以注意到每次刷新浏览器时，时间也跟着更新。看起来我们确实创建了动态的网页内容。

2.2.5 把时间加上

我们最初的问题是怎样向应用程序的用户显示当前的时间。现在我们已经知道如何在应用程序中显示动态数据，第二个需要解决的问题就是：从哪里获取时间？

让开发更简单

你应该已经注意到，我们在开发过程中都做了些什么。当在应用程序中添加代码后，并不需要重新启动这个应用程序就可以运行了。我们所做的改动在每次通过浏览器运行这个应用程序时都能生效，这到底是怎么回事？

这是因为 Rails 调度器相当智能。在开发模式下（相对于测试模式和生产模式），它会在新的请求到来时自动重新加载应用的源文件。在这种方式下，当编辑应用程序时，调度器能够保证运行的是最新的版本。这非常有利于开发。

然而，这种灵活性也是有代价的：它会在“用户输入 URL”与“应用程序做出响应”之间造成一个短暂的间隔——这段间隔就是调度器重新加载文件的时间。在开发阶段这个代价是值得的，但对于最后的产品则是无法忍受的。正是由于这个原因，这项特性在产品环境中是禁用的（参见第 16 章）。

① 从 Ruby 1.9 起，时间的标准格式已经改变了，现在它看起来像：2011-02-11 14:33:14-0500。

我们已经演示了如何在 `hello.html.erb` 模板中调用 Ruby 的 `Time.now` 方法。每次只要访问这个页面，用户就会看到当前的时间显示在当前页面中。对于这个简单的例子，这种办法就可以了。然而，通常情况下会选择另一种稍有不同的办法，将“获得时间”的方法放在控制器中，视图则只承担显示信息的职责。因此，要对控制器中的动作方法稍加修改，将时间值放在名为 `@time` 的实例变量（见 4.4 节）中：

```
work/demo3/app/controllers/say_controller.rb
```

```
class SayController < ApplicationController
  def hello
    ▶ @time = Time.now
    end

    def goodbye
    end

end
```

在 `.html.erb` 模板中，使用这个时间实例变量替代输出中的时间：

```
work/demo3/app/views/say/hello.html.erb
```

```
<h1>Hello from Rails!</h1>
<p>
  ▶ It is now <%= @time %>
</p>
```

当刷新浏览器窗口时，我们将再一次看到当前时间，并且看到控制器与视图成功地完成交互。

为什么要在控制器中设置待显示的时间，然后在视图中使用它？这不是自找麻烦吗？问得好。在这个应用程序中，我们看不到什么区别，但是把逻辑功能放入控制器中会给我们带来方便。譬如，也许我们将来会对应用程序进行扩展，使其可以在不同国家中使用，这样我们就需要对时间的显示加以本地化，即与用户所在时区的时间相适应。这将是大量应用层代码，并不适合嵌在视图层中。通过在控制器中设置要显示的时间信息，可以使该应用程序更加灵活：可以在控制器中修改显示格式和时区设置，而不必对使用这个时间对象的视图做任何修改。时间信息是一份数据，它应该由控制器提供给视图。等到介绍模型时，我们会看到很多这样的例子。

2.2.6 故事讲到现在

我们来简单回顾一下，这个应用程序是如何工作的。

- 1) 用户通过浏览器进入该应用程序。在这里，我们使用一个本地 URL，例如 `http://localhost:3000/say/hello`。

- 2) 然后，Rails 把 URL 分解为两部分，并对其进行分析，以进行路由模式匹配。

`say` 这一部分被视为控制器的名称，因此 Rails 为 `SayController` 这个 Ruby 类（位于 `app/controllers/say_controller.rb` 文件）新建一个实例。

- 3) URL 路径中的下一部分 `hello` 被视为行为的名称。Rails 调用控制器中名称为 `hello` 的方法，该方法新建一个 `Time` 对象，其保存当前的时间，并将其放进 `@time` 实例变量中。

4) Rails 寻找一个用于显示结果的模板，它会在 `app/views` 目录中寻找与控制器名称相同的子目录（say），然后在该子目录中寻找与行为名称相符的文件（`hello.html.erb`）。

5) Rails 借助 ERb 模板系统对这个文件进行处理，执行其中嵌套的 Ruby 代码，以及将控制器提供的值替换进去。

6) Rails 处理的结果发送给浏览器，并且结束对这一请求的处理。

这并不是整个故事——Rails 给了我们很多机会，可以对基本的工作流程进行调整（我们将很快体会到这些优点）。到此，我们已经演示了 Rails 的基本思想之一：惯例重于配置原则。通过使用方便的默认值和应用按惯例来构建 URL、指定控制器定义的文件的位置和指定所使用的类名和方法，Rails 应用程序通常只需要很少或者完全不需要外部配置，由此应用程序的各个部分以一种自然的方式组合在一起。

2.3 把页面连起来

很少有哪个 Web 应用只有一个页面。现在，我们就来试试给“Hello, World!”应用再加上一点 Web 设计的美妙效果。

一般来说，应用程序中的每个页面都会对应一个视图。在这里，要新加一个行为方法来处理页面（这并非是非必需的，就如我们将会在本书后面的内容中看到的）。我们将在控制器中使用同样的行为，也放在 SayController 控制器中——再次声明，这不是必需的，不过目前并没有什么特别的理由来新建一个控制器。

我们已经知道创建了一个名为 `goodbye` 的动作，所以剩下只需在 `app/views/say` 目录下创建一个新模板就行了。这一次它的文件名为 `goodbye.html.erb`，这是因为在默认的情况下，模板的名字应该与其相关联动作的名字匹配。

```
work/demo4/app/views/say/goodbye.html.erb
```

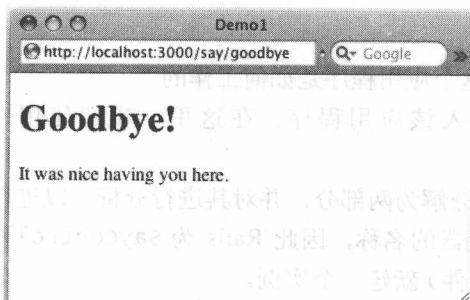
```
<h1>Goodbye! </h1>
```

```
<p>
```

```
  It was nice having you here.
```

```
</p>
```

打开我们信赖的浏览器，这次输入的 URL 是 `http://localhost:3000/say/goodbye`。你应该会看到如下图所示的效果。



现在需要将这两个页面连接起来。要在“hello”页面上放一个链接，让它把用户带到“goodbye”页面；反过来，后者也应该有一个指向前者的链接。在一个真实的应用程序中，我们可能希望通过好看的按钮来提供这样的功能，不过现在只要使用超链接就够了。

我们已经知道，Rails 将按照命名惯例把 URL 解析为目的控制器和该控制器中的行为名称两部分。因此，这些链接也应该遵循这一 URL 惯例。

hello.html.erb 文件将包含下列内容：

```
...
<p>
  Say <a href="/say/goodbye">Goodbye</a>!
</p>
...
```

而在 goodbye.html.erb 文件会使用其他方式：

```
...
<p>
  Say <a href="/say/hello">Hello</a>!
</p>
...
```

这种办法当然管用，但多少有点儿效果不佳：如果把应用程序移动到 Web 服务器的另一个目录下，这些 URL 就会失效。而且，这实际上是把 Rails 对 URL 格式的解读直接写进了代码中，而 Rails 将来的版本完全有可能改变目前的解读方式。

幸运的是，这里并没有什么风险需要承担。Rails 提供了一大堆可以在视图模板中使用的辅助方法。在这里，可以使用 `link_to()` 辅助方法，这个方法可以创建指向一个动作的超链接（这个链接事实上还含有额外的功能，这里先把它放在一边）。使用 `link_to()` 方法之后，hello.html.erb 就变成了：

```
work/demo5/app/views/say/hello.html.erb
<h1>Hello from Rails!</h1>
<p>
  It is now <%= @time %>
</p>
▶ <p>
▶   Time to say
▶   <%= link_to "Goodbye", say_goodbye_path %>!
▶ </p>
```

在最后一个 `<%=...%>` Erb 序列中嵌入了对 `link_to()` 的调用，它创建一个到 URL 的链接，这个 URL 会调用 goodbye 动作。在调用 `link_to()` 时，第一个参数是超链接中显示的文字，第二个参数则告诉 Rails：超链接应该指向 goodbye 这个动作。

我们不妨多花些时间来思考 `link_to()` 方法的第二个参数。我们刚才写了如下代码：

```
link_to "Goodbye!", say_goodbye_path
```

首先，`link_to()` 是一个方法（在 Rails 中把这样的方法叫做辅助方法（helper），因为它给编写视图模板带来了便利）。如果你使用过 Java 之类的编程语言，也许会略感惊讶地发现：Ruby

并不强求把方法参数放进圆括号里。当然，如果你愿意的话，也可以加上圆括号。

这个 `say_goodbye_path` 参数其实就是 Rails 预计算过的一个值，这样使得应用程序视图可以识别它。这个参数等于 `/say/goodbye` 路径。以后你会看到，Rails 提供对在应用程序中用到的所有路径命名的能力。

那么，回到该应用程序。现在，如果我们用浏览器查看 `hello` 页面，其中就会出现一个指向 `goodbye` 页面的链接，就像在下图所示的那样。



可以在 `goodbye.html.erb` 中做相应的修改，链接回初始的 `hello` 页面。

```
work/demo5/app/views/say/goodbye.html.erb
```

```
<h1>Goodbye!</h1>
<p>
  It was nice having you here.
</p>
<p>
  Say <%= link_to "Hello", say_hello_path %> again.
</p>
```

到这里我们已经完成了这个用于演示的 Rails 应用程序，并且通过它检验了安装的 Rails 是否正确运作。在复习这一章的内容后，我们将创建真实的应用程序。

2.4 本章小结

在本章中，我们创建了一个演示应用。通过这一过程，我们学会了：

- 如何新建一个 Rails 应用程序，以及如何在其中新建控制器；
- 如何在控制器中创建动态内容，以及如何在视图模板中显示动态内容；
- 如何把各个页面链接起来。

这是一个很好的开始，却没有花费我们太多的时间或精力。在第 3 章中，我们会继续构建一个更真实、更复杂的应用程序。

2.4.1 练习时间

可以自己尝试以下任务。

- 试试下面的表达式：

- 加法: `<%= 1+2 %>`
- 字符串连接: `<%= "cow" + "boy" %>`
- 一小时内时间的表达: `<%= 1.hour.from_now %>`
- 调用下列方法会返回一个列表, 其中包含当前目录中所有的文件:

```
@files = Dir.glob('*')
```

用它来设置控制器方法中的一个实例变量, 然后编写对应的模板来在浏览器中显示所有的文件名。

提示: 我们已经介绍过如何在 ERb 模板中循环 n 次, 用下列代码就可以遍历一个集合:

```
<% for file in @files %>
  file name is: <%= file %>
<% end %>
```

还可以用 `` 来显示这个列表。

(更多的提示请看 <http://www.pragprog.com/wikis/wiki/RailsPlayTime>。)

2.4.2 清理现场

如果你一直跟着我们的步伐编写本章提到的示例代码, 现在应用程序大概还在你的电脑上运行。等你继续跟着我们编写下一个应用程序并且运行新的应用程序时, 你就会遇到冲突, 因为现在的应用程序已经占用了电脑的 3000 端口。所以, 请在启动 WEBrick 的那个窗口里按 Ctrl+C 快捷键, 停止运行当前的应用程序。

现在, 让我们继续下面的内容, 它是关于 Rails 的概述。

第③章

Rails 应用程序框架

在本章中，我们将学习：

- 模型 (model)
- 视图 (view)
- 控制器 (controller)

Rails 有趣的特点之一是对“如何组织 Web 应用的结构”有着相当严格的约束。但令人惊讶的是，这些约束反倒使创建应用程序变得更加简单——甚至简单得多。现在，就让我们来看看这是为什么。

3.1 模型、视图以及控制器

1979 年，Trygve Reenskaug 提出了一种开发交互式应用的全新架构。在这个设计方案中，把应用程序分为三类组件：模型、视图以及控制器。

模型 (model) 负责维持应用程序的状态。有时候这种状态是短暂的，只对于和用户的几次交互有效；有时候这种状态则是持久的，需要将其保存在应用程序之外，通常保存在数据库中。

模型是数据，但又不只是数据，它还负责把业务规则附加在这些数据之上。譬如，“对于 20 美元以下的订单不予打折”这一约束就要由模型来执行。这种做法是很有意义的。通过将业务规则的实现放进模型中，就可以确保应用程序的其他部分不会使数据无效。这样使得模型不仅是数据的容器，还是数据的监护者。

视图 (view) 负责生成用户界面——通常会根据模型中的数据来生成。譬如，在线商店可能需要将一系列商品显示在屏幕上。通过模型可以访问这个商品列表，但还需要一个视图，它通过模型访问商品列表，并将其格式化为最终用户能够理解的形式。虽然视图可能允许用户以多种方式输入数据，但输入的数据一定不由视图本身来处理，视图的唯一工作就是显示数据。但是出于不同的目的，可能会有多个视图访问同一个模型数据。在在线商店的例子中，就有一个视图显示分类页面上的商品信息，还有管理员使用的视图集用于专门添加和编辑商品信息。

控制器 (controller) 负责协调整个应用程序的运转。控制器接收来自外界的事件（通常是用户输入），与模型进行交互，并将合适的视图展示给用户。

这个三位一体的组合——模型、视图和控制器——构成了一种架构，那就是著名的 MVC。图 3.1 大致描述了 MVC 架构的概念。

MVC 的设计初衷是用于传统的 GUI 应用程序。开发者发现：通过把程序进行这种概念上的分离，可以大大降低系统的耦合度，使代码易于编写、易于维护。每个概念、每个动作都只在一个规定的地方进行描述。使用 MVC 开发应用程序，就好像在搭好的桁架上盖大楼一样——只要结构已经到位，搭建其他部分就容易多了。

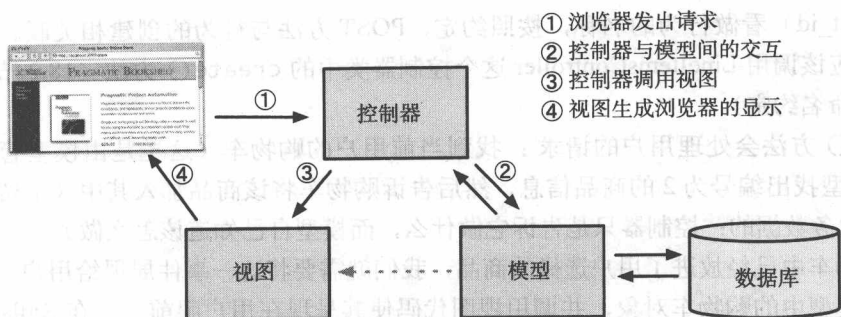


图 3.1 模型 - 视图 - 控制器架构

Ruby on Rails 也是一个 MVC 框架。Rails 强迫你将应用程序按照模型、视图和控制器进行划分，并遵循这一结构分别开发各部分的功能。当程序运行时，Rails 会把各个部分组装在一起。Rails 的有趣之处在于：通过使用智能化的默认设置把程序组合连接在一起，因此，一般情况下不需要编写任何外部元数据的配置信息。这正是 Rails 提倡的“约定重于配置”观念的体现。

在一个 Rails 应用程序中，进入的请求首先会发送给一个路由器，该路由器判断应该将请求发送到应用程序的什么部分以及如何解析这一请求。这一阶段将找出控制器代码中的某个特定方法，要求它来处理请求（用 Rails 语言表达，这个方法叫做行为（action））。行为可以查阅请求中携带的数据，可以与模型交互，也可以调用别的行为。最后，行为会为视图准备充分的信息，视图则将所需的信息展示给用户。

图 3.2 展示了 Rails 处理一个请求的全过程。在这个例子中，假设应用程序已经向用户展现了一个“产品分类列表”页面，用户则单击了某个产品旁边的 Add To Cart（放进购物车）按钮。这个按钮链接到这个应用程序，URL 是 `http://localhost:3000/line_items?product_id=2`，其中 `line_items` 是应用程序中的一个资源，“2”是所选商品在系统内部的 id 号。

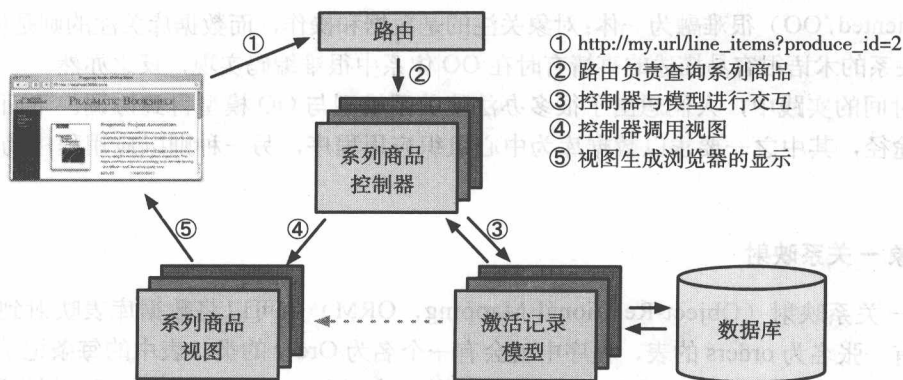


图 3.2 Rails 与 MVC

路由组件收到来自外部的请求之后，立即将其拆成小块。这个请求包含一条路径（`/line_items?product_id=2`）和一个方法（这里这个按钮完成 POST 操作，其他通常方法是 GET、PUT 和 DELETE）。在这个例子中，Rails 把路径的第一部分（`line_items`）看做控制器的名称，第二

部分 (product_id) 看做行为的名称, 按照约定, POST 方法与行为的创建相关联。这样路由组件就知道: 应该调用 LineItemsController 这个控制器类中的 create() 方法 (18.2 节将详细介绍 Rails 采用的命名约定)。

create() 方法会处理用户的请求: 找到当前用户的购物车 (这也是由模型管理的一个对象), 请求模型找出编号为 2 的商品信息, 然后告诉购物车将该商品加入其中 (请留意模型是如何跟踪所有业务数据的: 控制器只是告诉它做什么, 而模型自己知道该怎么做)。

既然购物车中已经放进了用户选择的商品, 我们就需要将这一事件展现给用户。控制器会安排视图访问模型中的购物车对象, 并调用视图代码使其呈现在用户眼前——在 Rails 中, 这一调用是隐式的。对于特定的行为, Rails 将根据命名约定自动为其查找一个特定的视图。

这就是关于一个 MVC Web 应用的全部内容了。只要遵循一定的命名约定, 并且合理划分功能, 你会发现编写代码变得轻松愉快, 应用程序会更具可扩展性、可维护性。看起来确实挺划算的。

如果 MVC 仅仅是“以某种方式划分代码”的话, 你可能会想, 那还要 Ruby on Rails 这样的框架做什么? 答案很简单: Rails 帮助你处理了所有底层的基础代码——所有那些需要耗费大把时间去处理的繁琐细节。从而让程序员只需要专注于应用程序的核心功能。现在, 让我们来看看它是如何做到这一点的。

3.2 Rails 的模型支持

一般来说, 人们都希望将 Web 应用中的信息保存到关系数据库中。一个订单系统会在数据库表中存储订单、订单项、客户信息等数据。即便是通常使用非结构化文本的应用 (例如, 博客和新闻网站), 也常常会用数据库作为后端数据存储介质。

虽然从通常使用的 SQL[⊖] 中不大看得出来, 但关系数据库确实是根据数学中的集合理论设计出来的。虽然从概念的角度来说这是件好事, 但它也使得关系数据库与面向对象编程语言 (Object-Oriented, OO) 很难融为一体: 对象关注的是数据和操作, 而数据库关注的则是值的集合。所以, 用关系的术语很容易描述的事情有时在 OO 体系中很难编码实现, 反之亦然。

在长时间的实践中, 人们想出了很多办法使关系模型与 OO 模型得到协调。下面将演示两种不同的途径, 其中之一要求以数据库为中心组织应用程序, 另一种则以应用程序为中心组织数据库。

3.2.1 对象 - 关系映射

对象 - 关系映射 (Object-Relational Mapping, ORM) 库可以将数据库表映射到类。如果数据库中有一张名为 orders 的表, 程序中就会有一个名为 Order 的类。表中的每条记录对应于该类的一个对象: 一条订单记录将表示为 Order 类的一个对象。在这个对象内部, 属性则用于读 / 写各个字段: Order 对象将拥有一些方法, 用于读 / 写数量 (amount)、销售税 (sales_tax) 等字段的值。

⊖ SQL: 结构化查询语言 (Structured Query Language), 是一种用于读、写关系数据库的语言。

此外, Rails 类还对数据库表进行了包装, 提供了一组类级别的方法, 用于执行表级别的操作。譬如, 我们可能需要找出具有某一特定 id 的订单, 这一功能就实现为一个类方法, 该方法返回对应的 Order 对象。在 Ruby 代码中, 这一操作大概会是这样:

```
order = Order.find(1)
puts "Customer #{order.customer_id}, amount=#{order.amount}"
```

有时, 这些类级别的方法会返回一组对象的集合:

```
Order.where(:name => 'dave').each do |order|
  puts order.amount
end
```

最后, 对应表中每行的对象会有一些方法, 用于操作自己所对应的那条记录。其中用得最多的方法大概就是 `save()`, 这个方法用于将记录保存到数据库中:

```
Order.where(:name => 'dave').each do |order|
  order.pay_type = "Purchase order"
  order.save
end
```

综上所述, ORM 层将数据库表映射到类, 将记录映射到对象, 将字段映射到对象属性。类方法用于执行表级别的操作, 实例方法则用于执行针对单条记录上的操作。

在一个典型的 ORM 库中, 你可以提供配置数据来指定数据库端与应用程序端之间的映射关系。使用这些 ORM 工具的程序员常常会发现: 他们不得不忙于创建和维护一大堆的 XML 配置文件。

3.2.2 Active Record

Active Record 是 Rails 所采用的 ORM 层。它完全遵循标准的 ORM 模型: 表映射到类, 记录映射到对象, 字段映射到对象属性。与其他大部分 ORM 库的不同之处在于它的配置方式: 它根据人们常用的命名约定提供了很有意义的默认配置, 因此, Active Record 将开发者执行的配置量降到了最低。

为了证明这一点, 下面就是一段用 Active Record 写的用于包装 orders 表的程序:

```
require 'active_record'

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.pay_type = "Purchase order"
order.save
```

这段代码使用新建的 Order 类来获取 id 为 1 的订单, 并修改它的 `pay_type` (在这里省略了创建数据库连接所需的代码)。Active Record 减轻了对于处理底层数据库的工作量, 让我们能够更专注于业务逻辑。

然而, Active Record 的威力还不只是这些。从第 5 章开始, 我们将开发一个“购物车”应

用程序，在那里你将看到 Active Record 是如何天衣无缝地与 Rails 框架的其他部分融为一体的。如果一个 Web 表单发送与业务对象相关的应用程序数据，那么 Active Record 可以将其抽取到模型中。Active Record 还支持复杂的模型数据验证，如果表单数据不能通过验证，只须编写一行代码，就可以在 Rails 视图中获取并格式化错误。

在 Rails 的 MVC 架构中，Active Record 是“模型”这一部分坚实的基础。

3.3 Action Pack: 视图与控制器

在 MVC 架构中，视图与控制器是密不可分的：控制器为视图提供数据，然后又接收来自页面的事件——这一页面正是由视图生成的。正因为有如此密切的交互，在 Rails 中对视图和控制器的支持捆绑在同一个组件中，那就是 Action Pack。

不过，不要因为 Action Pack 是一个组件，就认为 Rails 应用程序中的视图代码与控制器代码会搅在一起。恰恰相反，Rails 提供了编写 Web 应用时必需的隔离，清晰地将控制逻辑与表现逻辑区分开来。

3.3.1 视图支持

在 Rails 中，视图负责创建将要在浏览器上显示的页面——可能是整个页面，也可能是其中的一部分，由应用程序处理或作为 E-mail 发送。在最简单的情况下，视图就是一堆 HTML 代码，用于显示固定的文本。不过，一般来说，都需要在视图中加入一些动态内容——这些动态内容通常是控制器中的行为方法创建的。

在 Rails 中，动态内容是由模板生成的。模板的形式有 3 种，其中最常用的是 ERb：用一个名为 ERb (Embedded Ruby, 嵌入式 Ruby) 的工具，将 Ruby 代码片段嵌入视图文档中，在很多方式中类似于在其他 Web 框架（如 PHP 或 JSP）中完成它的方式。虽然这种方式非常灵活，但一些纯粹主义者认为它违背了 MVC 的精神：由于视图中嵌入了代码，人们就有可能将原本应该放在模型或控制器中的逻辑放进视图中。不过在我看来，这种批评是毫无根据的，即便在传统的 MVC 架构中，视图也会包含程序代码。保持关注点的清晰分离原本就是开发者的职责，不应该强求工具来确保这种分离（25.2 节将详细介绍 HTML 模板）。

XML Builder 也可以用 Ruby 代码来构造 XML 文档——生成的 XML 结构自动遵从代码的结构。我们将到时介绍 xml.builder 模板。

此外，Rails 还提供了 RJS 视图，用于在服务器端创建 JavaScript 片段，并将其传递到浏览器上执行。当创建 Ajax 界面时，这种视图会非常有用。11.2 节将介绍 RJS 视图。

3.3.2 还有控制器

Rails 控制器是应用程序的逻辑中心，它负责协调用户、视图与模型之间的交互。不过，Rails 已经在幕后处理了大部分的交互，你需要编写的代码都集中于应用层面的功能上。正基于此，Rails 的控制器非常容易开发和维护。

控制器还提供以下几种重要的辅助服务：

第④章

Ruby 简介

在本章中，我们将学习：

- 对象——命名规则和方法
- 数据类型——字符串、数组、散列和正则表达式
- 控制语句——if、while、代码块、迭代器和异常
- 构造代码块：类和模块
- YAML 和封送
- 本书所使用的常用习语

许多人可能是第一次接触 Ruby 和 Rails，不过，如果熟悉诸如 Java、JavaScript、PHP、Perl 或 Python 等语言中的一种，就会发现 Ruby 其实很容易上手。

本章既没有完整地介绍 Ruby 语言入门，也没有涵盖 Ruby 所有主题，比如优先级规则问题（在 Ruby 中 $1+2*3=7$ 的结果与大多数其他语言的结果是一样）。掌握了本章介绍的 Ruby 语言知识，就足以理解本书中的例子。

本章大量引用了参考书《Programming Ruby》第 2 章内容 [TFH08]。如果想要进一步了解 Ruby 语言背景知识，推荐阅读参考书《Programming Ruby》[TFH08]（也称为 PickAxe 图书）。它是学习 Ruby 的最佳途径，同时也是 Ruby 类、模块和库的最佳参考书。欢迎来到 Ruby 社区！

4.1 Ruby 是一门面向对象的语言

Ruby 语言操控的任何内容都是对象，操控的结果同样也是对象。

当编写面向对象代码时，通常要从现实世界中了解模型概念。通常在建模过程中，将发现需要对表达的事物进行归类。在在线商店中，单项商品的概念就是这样的类别。Ruby 语言通过关键词 `class` 定义类来表达每个不同的类别。然后，作为一种工厂使用这个类生成对象（object）——类的实例。对象是由状态和使用这些状态的方法组成的，前者如产品数量和编号，后者如计算单项商品总额的方法。4.4 节会介绍如何创建类。

对象是通过调用构造（constructor）方法来生成的，后者是与类相关联的特殊方法。标准的构造函数称为 `new`。

给出类 `LineItem`，可以像下面代码那样，创建单项商品对象：

```
line_item_one = LineItem.new
line_item_one.quantity = 1
line_item_one.sku      = "AUTO_B_00"
```

可以通过向对象发送消息来调用方法。该消息包含方法名及其可能需要的参数。当对象收到

消息时，它会在类本身查找相应的方法。试考虑下面几个方法调用的例子：

```
"dave".length  
line_item_one.quantity()  
cart.add_line_item(next_purchase)  
submit_tag "Add to Cart"
```

通常情况下，在方法调用时，圆括号是可选的。在 Rails 应用程序中，对于复杂的表达式中的大部分方法调用，都会使用圆括号，因为如果不使用的话，那么方法调用看上去就像命令或声明了。

像 Ruby 的其他结构一样，方法有自己的名称，Ruby 命名有着特殊的规则。如果刚从别的语言转到 Ruby 的话，以前可能从未见过这些规则。

4.1.1 Ruby 命名规则

局部变量名、方法参数名和方法名都必须以小写字母或下划线作为开头，例如 `order`、`line_item` 和 `xr2000` 都是符合 Ruby 命名规则的。实例变量名（将在 4.4 节深入介绍）要用 `@`（读“at”）作为前缀，如 `@quantity` 和 `@product_id`。对于那些包含多个单词的方法名和变量名，Ruby 通常习惯用下划线来分隔一个多词方法或变量名中的单词（所以 Ruby 语言通常用 `line_item` 而不用 `lineItem`）。

类名、模块名和常量名必须以大写字母作为首字母。对于那些包含多个单词的名称，这里通常使用的是以大写而不是下划线来分辨每个单词的开始。例如，类名称 `Object`、`PurchaseOrder` 和 `LineItem`。

Rails 用符号（symbol）数据类型来标识事物，尤其是，在命名方法参数和散列中查找内容时，可把符号作为关键字，如下所示：

```
redirect_to :action => "edit", :id => params[:id]
```

如上所示，符号看上去就像变量名一样，不过它有个冒号作为前缀，例如 `:action`、`:line_items` 和 `:id`。可以把符号看做字符串文字值（literal），使其如魔法般地转换成常量。或者可以把冒号看成“所命名的事物”，所以 `:id` 就是“该事物命名为 id”。

上面已经用过一些方法，接下来介绍如何定义方法。

4.1.2 方法

下面编写能返回个性化问候的方法并调用它几次：

```
def say_goodnight(name)  
  result = 'Good night, ' + name  
  return result  
end  
  
# Time for bed...  
puts say_goodnight('Mary-Ellen')  
puts say_goodnight('John-Boy')
```

现在已经定义了这个方法并调用它两次。在这两次调用中，把返回结果传给了方法 `puts`。在控制台上该方法输出其参数并换行（也就是把光标移到输出的下一行）。

只要把每条语句都放在单独一行中，就不需要在每条语句末尾加分号。Ruby 注释以 `#` 开始到行尾结束。缩进不是那么讲究的（事实上 Ruby 标准是两个字符缩进）。

Ruby 不是使用花括号来划定一组定义和复合语句（比如方法和类）的界限，而是使用关键字 `end`。关键字 `return` 是可选的，如果在方法中没有这个关键字的话，那就返回最后一个表达式的计算结果。

4.2 数据类型

由于 Ruby 语言中的任何事物都是对象，因此 Ruby 语言中的有些数据类型用于支持某些特别的语法，尤其是在定义文字值的时候。在上述例子中，已经用到了简单字符串和字符串连接。

4.2.1 字符串

前面例子已经演示了几个 Ruby 字符串对象。有两种生成字符串对象的方式，字符串文字值（string literal）加单引号或者加双引号。这两种方式的区别在于 Ruby 构造文字量时，对字符串的处理次数是不一样的。在单引号的情形下，Ruby 对字符串文字值基本不做任何处理，在单引号内所输入的文本就是该字符串的值。

在双引号的情况下，Ruby 会多做一些处理。首先，在文本中查找要替换的字符，也就是那些带反斜杠的字符，然后用二进制值替换。最常见的就是 `\n`，将其替换为换行符。当向控制台写入包含换行符的字符串时，这个 `\n` 可以强制换行。

其次，Ruby 会在双引号括起来的字符串中插入表达式。在字符串中，将字符串内的 `#{ 表达式 }` 替换成表达式的值。可以用这个功能重写前面的那个方法：

```
def say_goodnight(name)
  "Good night, #{name.capitalize}"
end
puts say_goodnight('pa')
```

当 Ruby 构造这个字符串对象时，它查看 `name` 的当前值，并将该值替换到字符串中。可以在 `#{...}` 这个结构中使用任意复杂的表达式。这里调用了适用于任何字符串的方法 `capitalize`，该方法将其参数首字母变成大写并输出。

字符串是原始数据类型，是有序的字节或字符集合。Ruby 也提供了可以包含任意对象的集合类型，即数组和散列。

4.2.2 数组和散列

Ruby 数组（array）和散列（hash）都是带索引的集合。这两个类型都是用来存储对象集合的，它们的对象可以通过键来访问。数组的键是整数类型，而散列的键可以是任何对象。随着动态变化，数组和散列都会增加其内存空间来存放新元素。相比较而言，数组访问效率更高，但散列更具灵活性。数组和散列都能保存任何类型的对象，例如，可以用数组来保存整数、字符串和

浮点数。

使用数组文字值 (array literal) 创建和初始化新数组对象——在方括号内的一组元素。给定一个数组对象，通过提供方括号内的索引，可以访问单独元素，就像下面的例子所演示的那样。Ruby 数组索引是从 0 开始的。

```
a = [ 1, 'cat', 3.14 ] # array with three elements
a[0]                  # access the first element (1)
a[2] = nil            # set the third element
                      # array now [ 1, 'cat', nil ]
```

你可能已经注意到了，在这个例子中使用了特殊值 `nil`。在许多编程语言中，`nil` (或 `null`) 概念意味着“没有对象”。Ruby 语言不是这样规定的；`nil` 是对象，就像其他的对象一样，只是它代表没有任何事物。

通常在数组中使用方法 `<<`，它把值追加到其接收器上。

```
ages = []
for person in @people
  ages << person.age
end
```

在创建以单词为元素的数组时，Ruby 有个捷径，如下所示：

```
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
# this is the same:
a = %w{ ant bee cat dog elk }
```

Ruby 散列和数组很相似，但散列文字值 (hash literal) 使用花括号而不是方括号。对于每个元素，该文字值必须提供两个对象：键和值。例如，可能要建立乐器与管弦乐器组的对应关系。

```
inst_section = {
  :cello    => 'string',
  :clarinet => 'woodwind',
  :drum     => 'percussion',
  :oboe     => 'woodwind',
  :trumpet  => 'brass',
  :violin   => 'string'
}
```

上面代码 `=>` 左边的内容是键，而右边的是对应值。键必须是唯一的，不能有两个元素的键都是 `:drum`。散列中的键和值可以是任意对象，散列中的值可以是数组或者其他散列等。在 Rails 中，散列通常使用符号作为键。Rails 已隐约地修改了许多散列，以至于当进行插入和查找值时，可交换地使用字符串或符号作为键。

像数组一样，通过方括号标记法，散列也可以索引散列元素项：

```
inst_section[:oboe]    #=> 'woodwind'
inst_section[:cello]   #=> 'string'
inst_section[:bassoon] #=> nil
```

如上所示，当要寻找的键不在散列中时，会返回 `nil`。在通常情况下，这还是很方便的，这是因为在条件表达式中，`nil` 表示 `false`。

在方法调用上可以把散列作为参数。Ruby 允许省略花括号，但前提是散列是方法调用的最

后一个参数。在 Rails 中该功能得到广泛应用。下面代码片段演示了如何将具有两个元素的散列传递给方法 `redirect_to`。从实际效果来看，可以认为它不是散列，并且假装 Ruby 具有关键字参数。

```
redirect_to :action => 'show', :id => product.id
```

特别值得一提的数据类型——正则表达式。

4.2.3 正则表达式

正则表达式可以指定字符的模式 (pattern)，使其与字符串相匹配。在 Ruby 中，通常通过使用代码 `/pattern/` 或 `%R{pattern}` 来生成正则表达式。

例如，对于所有包含文本 Perl 或 Python 的字符串，可以编写与之相匹配的正则表达式 `/Perl|Python/`。

正斜杠限定了这个模式的界限，该模式包含要匹配字符串的两个条件，它们是由竖线 (|) 分隔的。这条竖线表示“只要左边或右边有一个匹配即可”，在这种情况下，或者是 Perl，或者是 Python。在模式中使用圆括号，就像算术表达式一样，所以也可以将该模式写成 `/P(erl|ython)/`。程序通常使用 `==` 匹配运算符，以判断字符串是否匹配正则表达式：

```
if line == /P(erl|ython)/  
  puts "There seems to be another scripting language here"  
end
```

在模式中也可以指定会重复出现的字符。`/ab+c/` 匹配的是以单个字母 a 开始，紧跟着单个或多个字母 b，最后以单个字母 c 结尾的字符串。要是将模式中的加号 (+) 换成星号 (*)，即 `/ab*c/`，那么匹配的字符串就包含单个字母 a、零个或多个字母 b 以及单个字母 c。

以反斜杠开始的是有关联的一组特殊序列；最值得注意的是以下几个特殊序列，`\d` 表示匹配任何数字，`\s` 表示匹配任何空白字符，`\w` 表示匹配任何包括字母与数字（“单词”）的字符。

Ruby 正则表达式是深刻而又复杂的话题，这里只是非常简单地进行了介绍。详细介绍请见 PickAxe 图书。

本书只使用非常简单的正则表达式。

在简单介绍了这些数据类型后，下面探讨逻辑方法。

4.3 逻辑方法

函数调用就是语句。Ruby 提供了大量的判断方法，使其影响函数调用的重复性和顺序。

4.3.1 控制结构

Ruby 具有所有常见的控制结构，如 `if` 语句和 `while` 循环语句。Java、C 和 Perl 程序员可能会因语句段缺少花括号而困扰。相反，Ruby 使用关键字 `end` 表示一段语句的结束：

```
if count > 10  
  puts "Try again"  
elsif tries == 3
```



```
puts "You lose"
else
  puts "Enter a number"
end
```

同样，while 语句也是以 end 结束的：

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

Ruby 还包含这些语句的变种：unless 语句类似于 if，区别在于它判断条件为“不是 true”。同样，until 与 while 相类似，程序会继续循环直到判断条件为真为止。

当 if、unless、while 或 until 这些语句的主体只包含简单表达式时，Ruby 语句 (statement) 修饰符就是一种有用的捷径。可以非常简单地编写该表达式，其后紧接着修饰符关键字和条件即可：

```
puts "Danger, Will Robinson" if radiation > 3000
distance = distance * 1.2 while distance < 100
```

在 Ruby 应用程序中，虽然 if 语句应用相当普遍，但 Ruby 语言初学者常常会惊讶地发现，很少采用循环结构。取而代之，常常会使用代码块 (block) 和迭代器 (iterator)。

4.3.2 代码块和迭代器

代码块是由花括号或者 do...end 及其内部代码所组成的。通常花括号用于只有单行语句的代码块，而 do...end 用于多行语句的代码块：

```
{ puts "Hello" }      # this is a block

do                    ###
  club.enroll(person) # and so is this
  person.socialize    #
end                    ###
```

为了将代码块传递给方法，只要将代码块放在方法的参数（如果有的话）后。换句话说，把代码块开头部分放到方法调用的语句结尾处。例如，在下面代码中，包含语句 puts "Hi" 的代码块是与调用方法 greet 相关联的：

```
greet { puts "Hi" }
```

如果方法调用是带参数的，可以把参数放在代码块前：

```
verbose_greet("Dave", "loyal customer") { puts "Hi" }
```

通过使用 Ruby 语句 yield 方法可以一次或多次地调用与其相关联的代码块。可以把 yield 看成是像方法调用的操作，使其调用到与包含方法 yield 相关联的代码块中。通过把参数传给方法 yield，可以使值传递到代码块。在代码块中列出了这些参数名，以便接收在竖杠 (|) 内的参数。

Ruby 应用程序中到处出现代码块。它们往往与迭代器一起使用，从某种集合（如数组）中返回连续元素的方法：

```
animals = %w( ant bee cat dog elk )    # create an array
animals.each {|animal| puts animal }    # iterate over the contents
```

每个整数 N 实现 `times` 方法，该方法调用 N 次相关代码块：

```
3.times { print "Ho! " }    #=> Ho! Ho! Ho!
```

这个 `&` 前缀操作符允许方法作为命名参数抓取所传递的代码块，该代码块可看做命名参数。

```
def wrap &b
  print "Santa says: "
  3.times(&b)
  print "\n"
end
wrap { print "Ho! " }
```

在代码块或方法内，除非存在异常，否则控制语句是按次序进行的。

4.3.3 异常

异常（exception）都是对象（该对象属于类 `Exception` 或者其子类）。`raise` 方法通常用于异常抛出。这个异常打断了正常的程序流。Ruby 反向搜索调用栈中能处理异常的代码。

方法和代码块都封包在 `begin` 和 `end` 关键字之间，且用 `rescue` 子句来拦截某种异常类。

```
begin
  content = load_blog_data(file_name)
rescue BlogDataNotFound
  STDERR.puts "File #{file_name} not found"
rescue BlogDataFormatError
  STDERR.puts "Invalid blog data in #{file_name}"
rescue Exception => exc
  STDERR.puts "General error loading #{file_name}: #{exc.message}"
end
```

`rescue` 子句可以直接放在方法定义的最外层，并不需要包含 `begin/end` 块中的内容。

以上简单介绍了控制流，并且也了解到构建基本的代码块，以此可以搭建更大的结构。

4.4 组织结构

在 Ruby 中存在使方法系统化的两个基本概念，即类和模块。下面将依次介绍它们。

4.4.1 类

下面是 Ruby 类（class）的定义：

```
Line 1  class Order < ActiveRecord::Base
-
-      has_many :line_items
-
5      def self.find_all_unpaid
-      self.where('paid = 0')
-
-      end
```

```

-   def total
10   sum = 0
-   line_items.each {|li| sum += li.total}
-   end
- end

```

类定义以关键字 `class` 开头，紧接着是类名（类名必须以大写字母开头）^①。上述代码定义了 ActiveRecord 模块中 Base 类的子类 Order。

Rails 大量使用了类层面声明。这里 Active Record 模块定义了方法 `has_many`。而作为正在定义的类 Order 调用它。通常这些方法对这个类做出断言，因此在本书中，它们称为声明 (declaration)。

在类主体内，可以定义类方法和实例方法。在方法名前面加上 `self` 前缀（如第 5 行代码所示）就是类方法，可以在该类上广泛地调用它。在这种情况下，在 Depot 应用程序的任何地方可以使用如下调用：

```
to_collect = Order.find_all_unpaid
```

在实例变量 (instance variable) 中，类的对象保存其状态，这些变量名都以 `@` 开始，可用于该类的所有实例方法。每个对象都有自己的一组实例变量。

从类的外部无法直接访问实例变量。为了访问它们，就要实现返回其值的方法。

```

class Greeter
  def initialize(name)
    @name = name
  end

  def name
    @name
  end

  def name=(new_name)
    @name = new_name
  end
end

g = Greeter.new("Barney")
puts g.name #=> Barney
g.name = "Betty"
puts g.name #=> Betty

```

Ruby 提供了便利的方法来编写这些存取方法（对于所有那些忙于编写方法 `getter` 和 `setter` 的人们来说，这可是个好消息）：

```

class Greeter
  attr_accessor :name      # create reader and writer methods
  attr_reader  :greeting  # create reader only
  attr_writer  :age        # create writer only
end

```

① 除了这一行和最后一行关键词 `end` 外，其余代码称为类主体 (class body)。——译者注

在默认情况下, 类的实例方法都是公有的, 谁都可以调用它们。你可能想要重写这些方法, 以达到只有另外的类的实例方法调用它们:

```
class MyClass
  def m1      # this method is public
  end

  protected

  def m2      # this method is protected
  end

  private

  def m3      # this method is private
  end
end
```

`private` (私有) 指令是最严格的, 只能在同一个实例中调用私有 (`private`) 方法。可以在同一个实例中和由同一类及其子类的实例方法调用受保护的 (`protected`) 方法。

类不是 Ruby 语言的唯一组织结构, 另一个组织结构就是模块。

4.4.2 模块

模块 (module) 与类是相类似的, 它们都是一个由方法、常量、其他模块和类定义所构成的集合。与类不同的是, 模块不能创建基于其本身的对象。

模块有两个作用。首先, 它们作为命名空间 (namespace) 定义方法, 其名称不与其他地方定义的方法有冲突。其次, 它们允许在不同的类之间共享功能。如果类混合在模块中, 那么该模块的实例方法就是可用的, 好像这些方法已经在类中定义过了。多个类可以混合在同一个模块中, 它们不需要继承就可以共享模块的功能。还可以把多个模块放到单个类中。

辅助方法就是 Rails 使用模块的例子。Rails 自动地把这些帮助模块放到适当的视图模板中。例如, 如果想编写辅助方法, 该辅助方法从商店控制器采用的视图图中是可调用的, 那么就可以定义以下这个模块, 它是在目录 `app/helpers` 中文件 `store_helper.rb` 中的模块:

```
module StoreHelper
  def capitalize_words(string)
    string.split(' ').map {|word| word.capitalize}.join(' ')
  end
end
```

YAML 是 Ruby 标准库之一, 特别值得一提的是它在 Rails 中的用法。

4.4.3 YAML

YAML^① 是递归首字缩写 (recursive acronym), 其全称是 YAML Ain't Markup Language。在 Rails 的上下文中, YAML 通常用来定义配置内容, 如数据库、测试数据和翻译。如下例所示:

① <http://www.yaml.org/>

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

在 YAML 中，缩进是很重要的，这里定义 `development` 有 4 个键/值对，用冒号分隔。

当 YAML 是表示数据的一种方式时，特别是与人交互时，Ruby 为应用程序提供了一种更通用的数据表达方式。

4.5 封送对象

Ruby 能接收对象，并且将该对象转换成字节流，进而保存于应用程序之外，该过程叫封送 (marshaling)。以后该应用程序（或由完全独立的应用程序）的另外实例可以读取所保存的这一对象，并且可以重新构建最初保存对象的副本。

当使用封送时，存在两个潜在的问题。第一个问题是，有些对象不能转储，如果要转储的对象是绑定 (binding)、过程 (procedure) 或方法对象 (method object)、IO 类的实例或者单例 (singleton) 对象，或者如果要转储匿名类或模块，那么将抛出 `TypeError` 异常。

第二个问题是，当加载所封送的对象时，Ruby 需要知道该对象的类（及其包含所有的对象）定义。

Rails 使用封送方法来存储会话数据。如果想用 Rails 动态地加载类，那么在 Rails 重组会话数据时，或许还没有定义特定的类，这是有可能的。出于这个原因，在控制器中将使用 `model` 声明，以便列出所有封送的模型。这里就预先加载必要的类，使封送工作进行。

现在已经了解到 Ruby 语言基础知识，接下来通过比较大且带评注的示例，尝试把所学到的知识与该实例及其概念联系起来。接下来将快速地了解一些特殊的功能，它们对使用 Rails 编码会有所帮助。

4.6 综合分析

通过下面的示例，可以了解到 Rails 如何利用大量的 Ruby 特性来构造需要的代码，并使代码更有陈述性。在 6.1 节中将再次看到这个示例。现将侧重阐述这个示例的 Ruby 语言概况。

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price, :precision => 8, :scale => 2

      t.timestamps
    end
  end
end
```

```
def self.down
  drop_table :products
end
end
```

即使对 Ruby 一点儿都不了解，可能也会看懂上面那个例子，`up` 创建表 `products`，`down` 删除这个表。创建此表时定义了字段 `title`、`description`、`image_url` 和 `price` 以及时间戳（将在 4.7 节中介绍这些内容）。

现在从 Ruby 语言角度看一下这个例子。先定义类 `CreateProducts`，它继承自模块 `ActiveRecord` 中的类 `Migration`。然后定义了两个类方法，方法 `up` 和 `down`。每次调用单个类方法（这是在 `ActiveRecord::Migration` 中定义的），同时以符号的形式把表名传递给这个方法。

在创建该表之前，也把要计算的代码块传递给调用方法 `up`。当调用这个代码块时，将对象 `t` 传递给它。这个对象是用来存放字段列表的。在这个对象中，Rails 定义了一系列的方法——方法名都源自常用的数据类型。当调用这些方法时，它们仅把字段定义添加到一组不断扩充的字段名集合中。

定义的小数还接收许多可选的参数，该参数以散列方式表示。

对于 Ruby 语言新手而言，为了解决如此简单的问题，使用这种机制似乎过于复杂。对于熟悉 Ruby 语言的人，并不认为这种机制特别复杂。在任何情况下，Rails 都充分利用 Ruby 提供的工具，使得定义操作尽可能简单和可陈述（例如，定义迁移任务时）。甚至这种语言的很小特性，如可选的圆括号和花括号，也有助于提升整体的可读性和编码的简易性。

最后，还有一些小特性，或者说是一些较常见的习惯组合的特性，这些内容对于 Ruby 新手来说往往不太了解。本章最后将介绍它们。

4.7 Ruby 语言习语

许多 Ruby 语言的个别功能可以通过有趣的方式结合起来，而对于初学者，这种习惯用法的含义有点儿难以理解。在本书中会用到 Ruby 的一些习语：

方法 `empty!` 和 `empty?`

命名 Ruby 方法可以使用感叹号 (!)（一种感叹号 (bang) 类型方法）或问号 (?)（一种问号 (predicate) 类型方法）作为结尾。感叹号类型方法通常是用来销毁接收器的。问号类型方法取决于返回某个条件 `true` 或 `false`。

表达式 `a||b`

表达式 `a||b` 先计算 `a`。若 `a` 不是 `false` 或 `nil`，则停止计算，并返回 `a`。否则，该语句返回 `b`。如果还没有设置第一个值，那么返回默认值是通常的做法。

赋值语句 `a||=b`

这个赋值语句支持一组快捷方式：`a op=b` 就是 `a=a op b`[⊖]。对于大多数操作符这是适用的。

⊖ `op` 是假定某个操作符。——译者注


```
count += 1      # same as count = count + 1
price *= discount #      price = price * discount
count ||= 0     #      count = count || 0
```

所以，在还没有为变量 `count` 赋值的情况下，语句 `count ||= 0` 把变量 `count` 赋值为 0。

赋值语句 `obj = self.new`

有时候类方法需要生成这个类的实例。

```
class Person < ActiveRecord::Base
  def self.for_dave
    Person.new(:name => 'Dave')
  end
end
```

这段代码返回类 `Person` 的新对象。但是之后可能要继承该类：

```
class Employee < Person
  # ..
end

dave = Employee.for_dave # returns a Person
```

方法 `for_dave` 强制性地返回类 `Person` 的对象，这也就是由方法 `Employee.for_dave` 所返回的。相反，使用方法 `self.new` 返回接收器类的新对象 `Employee`。

操作符 `lambda`

`lambda` 操作符把块转换成 `Proc` 类的对象，将在 19.3 节中看到它的用法。

语句 `require File.dirname(__FILE__) + '/../test_helper'`

Ruby 语言 `require` 方法是把外部原代码文件加载到应用程序中。其用法是引入应用程序需要依赖的库和类。在通常情况下，借助于搜索目录 (`LOAD_PATH`) 清单，Ruby 语言找到这些文件。

有时候需要指定包括哪个文件。通过给语句 `require` 传递完整文件系统路径，可以做到。但问题是，并不知道路径是什么，用户可能在任何位置安装代码。

无论应用程序最终安装在哪里，对于目标文件和需要加载的文件而言，它们之间的相对路径都是不变的。知道了这一点，通过采用需要加载文件的绝对路径（可以通过特殊变量 `_FILE_` 来获得），可以得到目标文件的绝对路径，同时除目录名之外剔除所有内容，然后把这条相对路径附加到目标文件上。

此外，关于 Ruby 语言习语和 Ruby 语言陷阱，在网上还有大量很好的资料。以下是其中的几个网址：

- <http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/>
- http://en.wikipedia.org/wiki/Ruby_programming_language
- <http://www.zenspider.com/Languages/Ruby/QuickRef.html>

到这里，我们建立了坚实的基础：安装了 Rails、检验与简单应用程序一起工作的事情、了解了 Rails 以及重温了 Ruby 语言基础知识（当然，对于有些人而言，这是第一次接触）。现在是时候利用这些知识构建更大的应用程序了。

5.4.5 Ruby

```
count = 0
while count < 10
  puts "count is: #{count}"
  count = count + 1
end
```

所以，我们可以在变量 count 赋值 0 后，使用 count 范围 0-9 来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

```
class Person < ActiveRecord::Base
  def save!
    Person.new(name: 'David')
  end
end
```

这看起来跟 Ruby 的 Person 对象很像，但是这里可能跟 Ruby 的 Person 对象不太一样。

class Person < ActiveRecord::Base

end

David = Person.new(name: 'David')

David.save! 会返回一个 Person 对象，这跟 Ruby 的 Person 对象很像，但是这里可能跟 Ruby 的 Person 对象不太一样。

返回的 David 对象，使用 self.new 来创建新的 Person 对象，这跟 Ruby 的 Person 对象很像，但是这里可能跟 Ruby 的 Person 对象不太一样。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

```
David = Person.new(name: 'David')
David.save!
puts David.name
```

Ruby 的 Person 对象，使用 self.new 来创建新的 Person 对象，这跟 Ruby 的 Person 对象很像，但是这里可能跟 Ruby 的 Person 对象不太一样。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

在 Ruby 中，我们还可以通过以下方式来遍历 0-9 的每个数字。

第二部分

构建应用程序

第 5 章

Depot 应用程序

在本章中，我们将学习：

- 增量式开发方法
- 用例、页面流程和数据
- 优先级

我们可能一整天都深陷于简单测试程序的排错与分析中，但是这并不会有人因此而为我们买单，因此，不如学一些更实际的东西。下面创建称为 Depot 的网上购物车程序。

世界上已经有很多不同类型的购物车应用程序，还需要其他的吗？不需要，但还是有许多开发人员在开发它们。为什么我们的应用程序与其他的购物车程序就是不一样呢？

值得一提的是，开发购物车应用程序可以阐明 Rails 开发的许多特性。下面将看到如何创建简单维护网页、链接数据库表、处理会话以及创建表单。接下来的 12 章内容将介绍一些相关主题，如单元测试、安全性和页面布局。

5.1 增量式开发

接下来将增量地开发这个应用程序。在开始编写代码之前，我们不会尝试去弄清楚所有细节，而是只要有足够我们可以开始开发的需求规格说明书即可，然后立即着手实现某些功能。在开发时我们会尝试各种不同的想法，同时收集反馈信息，然后再开始另一个小设计（mini-design）和开发周期。

这种开发方式并不总是合适的。它需要与应用程序的用户密切合作，因为在整个开发过程中

都要收集反馈信息。我们可能会犯错误，或者客户可能会发现他们表达的事情和他们想真正得到的东西是不同的。这并不要紧，不管是什么原因，越早发现错误，那修复错误的代价就越少。总而言之，用这种开发方式，在开发中会有很多的变化。

因此需要一个工具集，使我们不会因为思路的改变而要修改许多东西。如果决定要添加新字段到数据库表中，或更改页面之间的导航方式，不应该为了这些变更，再去写一大堆代码或者改变许多的配置。正如即将看到的，Ruby on Rails 在处理变更时的优势——它是个理想的敏捷编程环境。

根据这个思路，我们将创建和维护一整套测试文档。这些测试将确保应用程序总能达到预期的目标。Rails 不仅能创建这样的测试，而且每当定义一个新的控制器时，它实际上还提供了—个初步的测试程序集合。

5.2 Depot 是做什么的

首先快速地写出 Depot 应用程序需求规格说明书的大轮廓。先将着眼于高层次的用例，并勾画出网页间的流程。同时尝试确定应用程序需要哪些数据（老实说，初步构想很可能是错误的）。

5.2.1 用例

用例（use case）仅是关于某些实体如何使用系统的简单陈述。咨询顾问总是在发明这些术语来标记其实我们都知道的一些东西——这是一种扭曲的商业模式，因为花哨的术语总比浅显易懂的语言更好赚钱，即使后者更有价值。

Depot 的用例是相当简单的（有些人甚至会说这么简陋）。首先确定两个不同的角色或演员：买家和卖家。

买家通过 Depot 浏览待售的商品，选择一些要购买的商品，并提供创建订单所需的信息。

卖家通过 Depot 来维护待售商品的清单，并确定正在等待发货的订单，以及标记那些已发货的订单。（卖家也依靠 Depot 赚了许多钱，然后退休去一个热带岛屿旅游，但那是另一本书的主题了。）

这就是我们所需的全部细节了。当然可以进一步细化，比如，维护商品信息究竟有什么具体含义，待发货物的订单由什么构成等，但何必呢？如果有的细节不是很清楚的话，我们会足够快地发现这些问题，因为我们会把工作成果不断地展示给客户。

谈及用户反馈，别忘记，立即去向客户收集反馈——要确保初始的用例（当然是粗略的）是符合客户需求的。假设用例过关，那么我们就可以从网站用户的不同角度来确定应用程序将如何运作。

5.2.2 页面流程

开发人员都喜欢在应用程序中有主页，并粗略地理解用户如何进行页面导航。在开发初期，这些页面流程可能是不完整的，但他们仍然可以帮助开发人员专注于所需要做的一些事情，以及各种行为的顺序关系。

有些人喜欢用 Photoshop、Word 或（令人有点儿害怕的）HTML 来模拟 Web 应用程序的页面流程。我们倾向于使用铅笔和纸。它的速度更快，客户也可以一起参与，随便抓支铅笔，修改涂画，非常方便。

买家的页面流程的第一张草图如图 5.1 所示。这个流程是一个非常传统的网站购物流程。买家看到分类目录页，然后每次选择一个商品。每个选中的商品都会添加到购物车中，每次选择后会显示购物车页面。买家可以去目录页继续购物，或付款购买已在购物车中的商品。在结账时，我们会获得买家的联系方式和相关的付款细节，然后显示在一个收据页面。目前还不知道要如何处理付款，所以在页面流程中的这些细节都是相当模糊的。

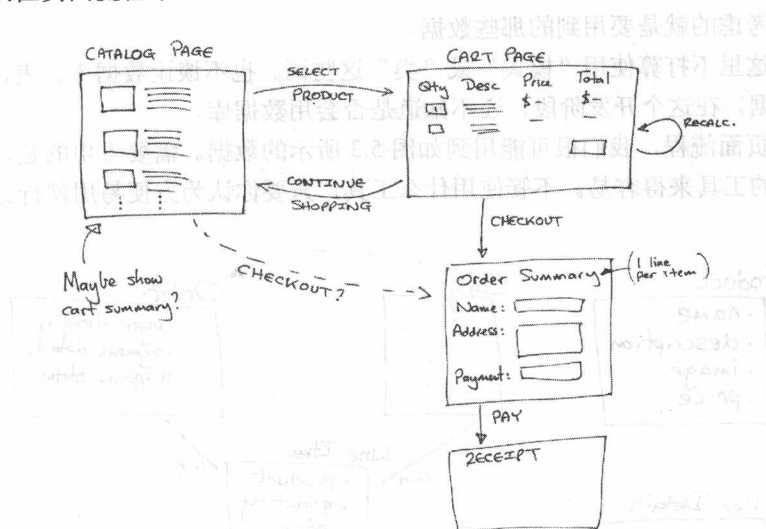


图 5.1 买家的页面流程

卖家的页面流程，如图 5.2 所示，也是相当简单。登录后，卖家看到一个菜单，让他创建或查看商品或配发已有的订单。当查看商品时，卖家可以选择编辑商品信息，或完全删除商品。

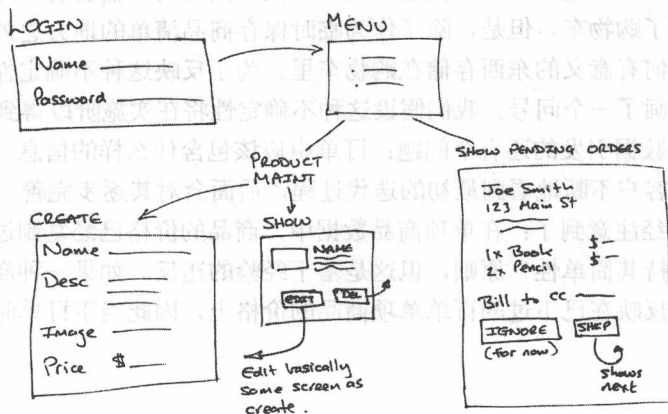


图 5.2 卖家的页面流程

发货选项非常简单。它显示每个尚未发货的订单，每个页面一个订单。卖家可以选择跳到一个订单或者用页面上的信息来配送当前的订单。

在现实环境下这个发货功能太简陋了，肯定是要改善的，其实发货这一块要比想象的复杂得多，如果现在就想详细地制定这个功能的完整的需求规范说明书，很有可能会做错。先就让它这样了，要有信心，当用户使用程序并给出反馈的时候，我们肯定可以再完善这个功能。

5.2.3 数据

最后，需要考虑的就是要用到的那些数据。

请注意，在这里不打算使用“模式”或“类”这些词。也不谈论数据库、表、关键字等。只是简单地谈论数据。在这个开发阶段，还不知道是否会用数据库。

基于用例和页面流程，我们很可能用到如图 5.3 所示的数据。需要重申的是，用铅笔和纸似乎比用一些花哨的工具来得容易。不管使用什么工具，只要你认为方便易用就行。

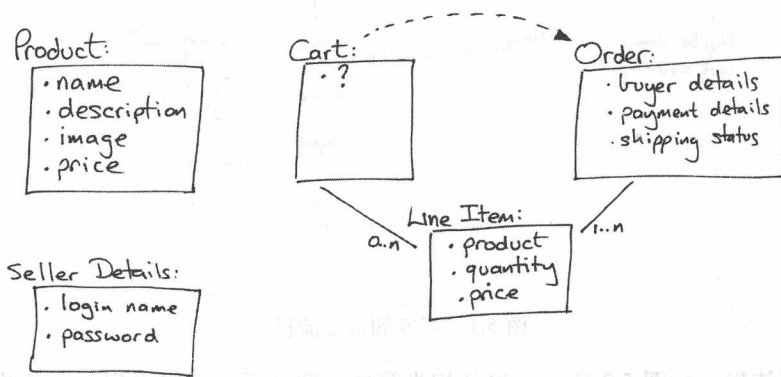


图 5.3 应用程序数据的初步假想

在处理数据流程图时，碰到了几个问题。当用户购买商品时，需要有个地方保存他们购买的商品清单，所以增加了购物车。但是，除了作为临时保存商品清单的地方之外，它似乎还有点奇怪——无法找到任何有意义的东西存储在购物车里。为了反映这种不确定性，于是在流程图中对应购物车的方框里画了一个问号。我们假设这种不确定性将在实施阶段得到解决。

随着这些高级的数据引发的还有个问题：订单中应该包含什么样的信息。同样，现在先不解决这个问题——随着客户不断地看到最初的迭代过程，后面会对其逐步完善。

最后，你应该已经注意到了：在单项商品数据中，商品的价格已经复制过了。这里有点儿打破了“从一开始就保持其简单性”原则，但这是基于经验的违反。如果一种商品的价格变化，这种价格变动应该不会反映在已下过的订单单项商品的价格上，因此当下订单时，每个单项商品才反映该商品的价格。

常用的复原建议

本书里的每段代码都是经过测试的。如果精确地跟随这里的开发场景，在 Linux、Mac OS X 或者 Windows 下使用推荐的 Rails 和 SQLite 3 版本，那么所有的一切都应该跟描述的一样，是能正常工作的。不过，偏离这场景的情况还是有可能发生。录入错误也时有发生，进行边际探索不仅是完全可能的，而且对此还应积极地鼓励。请注意，这可能会引导到陌生的领域去。不要害怕：对于经常出现的问题，相关章节会介绍相关的恢复办法，这里介绍一些常用的建议。

正如本书中指出的，只有少许的几个地方需要重新启动服务器。但如果真的被难倒了，重新启动服务器，可能也值得一试。

一条值得了解的“神奇”的命令会在第三部分详细解释，即 `rake db:migrate:redo`。这将撤销和重新应用最后一次迁移。

如果服务器不接收某些表单输入的话，可以尝试刷新浏览器的表单，并重新提交它。

接下来，需要再次与客户一起仔细核对，检查是否仍在正确轨道上。（当绘制这 3 个流程图时，客户很有可能还与我们坐在一起讨论。）

5.3 让我们来编码吧

所以，在坐下来与客户做了一些初步分析后，我们已经准备好开始用计算机进行开发了！工作将从那 3 个流程设计图开始，随着收集到的反馈信息，这些图也会迅速地过时，但这也正是扔掉这些原始设计方案的最佳时机。有趣的是，这也是为什么不用花很多时间在流程图上的缘故，如果没有花很长时间在某个东西上，那么扔掉就更容易了。

在接下来的章节中，将基于目前的理解来开始开发应用程序。不过，在翻过这一页之前，必须再回答一个问题：第一步应该做什么？

我们愿意和客户一起共同商定优先级。在这种情况下，应该告诉客户，在系统定义某些基本商品之前，开发其他的东西是困难的。所以我们建议先花几个小时的时间来制定商品的维护功能的最初版本并让它能运行起来。当然，客户会同意这种做法。

第 6 章

任务 A：创建应用程序

在本章中，我们将学习：

- 创建一个新的应用程序
- 配置数据库
- 创建模型和控制器
- 更新页面布局和视图

我们的第一个开发任务是创建 Web 界面，用来维护商品信息——创建新商品、编辑现有的商品、删除不再需要的商品等。下面将以小迭代的形式来开发这个应用程序，“小”的意思就是“可以用分钟来计量的”。让我们开始吧。

通常情况下，迭代过程会涉及多个步骤，比如在迭代 C 中有步骤 C1、C2、C3 等。本章迭代将有两个步骤。

6.1 迭代 A1：创建商品维护的应用程序

Depot 应用程序的核心是数据库。等到数据库安装、配置并测试完毕，再继续开发其他组件，这样可以避免许多烦恼。在安装配置数据库时，如果不能确定想要什么，那么就选择默认值，这样会容易许多。如果已经知道想要什么，在 Rails 中可以很简单地描述配置内容。

6.1.1 创建 Rails 应用程序

2.1 节已经介绍了如何创建新的 Rails 应用程序。在这里将做同样的事情，转到命令提示符下，输入 `rails new`，紧接着是项目名称。这里的项目名称为 Depot，先确保当前目录下不存在该应用程序，然后输入如下：

```
work> rails new depot
```

可以看到了一堆滚动的输出。当滚动输出结束后，会发现已创建的新目录 depot。这就是我们的工作目录。

```
work> cd depot
depot> ls -p
app/      config.ru  doc/      lib/      public/   README   test/    vendor/
config/   db/       Gemfile   log/      Rakefile  script/   tmp/
```

6.1.2 创建数据库

对于这个应用程序，将使用开源的 SQLite 数据库（如果想用接下来的代码，那么就要用这个数据库）。这里用 SQLite 3。

在开发 Rails 应用程序时, SQLite 3 是默认数据库。第 1 章也提到了这个数据库会随着 Rails 一起安装。在 SQLite 3 中不需要特定步骤来创建数据库, 也没有特殊的用户账户或密码来创建数据库。所以, 现在你就会体会到按照流程执行的好处了(或者就像 Rails 高手反复强调的那样: 约定优于配置)。

如果必须使用其他数据库服务器, 那用来创建数据库和授权的命令会有所不同。可以在“Getting Started Rails Guide”^①中找到一些有用的帮助。

6.1.3 生成脚手架

5.2 节的图 5.3 列举出了 products 表的基本内容。现在, 在数据库中实现它。需要创建数据库表和 Rails 模型, 应用程序通过这个模型使用该表、一系列创建用户界面的视图以及协调应用程序的控制器。

现在为 products 表创建模型、视图、控制器和迁移。在 Rails 中, 可以用一条命令完成所有工作, 对于给定的模型, Rails 可生成所谓的脚手架(scaffold)。请注意, 在命令行^②中使用的是单数形式, Product。在 Rails 中, 如果表名是该模型类的复数形式, 那模型会自动映射到数据库表。在这个例子中, 模型叫 Product, 那 Rails 就将这个模型关联到名为 products 表上。(模型将如何找到那个表? Rails 会在文件 config/database.yml 中的 development 项中找到该表的位置。对于 SQLite 3 用户而言, 在 db 目录中将有个文件。)

```
depot> rails generate scaffold Product \
      title:string description:text image_url:string price:decimal
  invoke  active_record
  create  db/migrate/20110211000001_create_products.rb
  create  app/models/product.rb
  invoke  test_unit
  create  test/unit/product_test.rb
  create  test/fixtures/products.yml
  route   resources :products
  invoke  scaffold_controller
  create  app/controllers/products_controller.rb
  invoke  erb
  create  app/views/products
  create  app/views/products/index.html.erb
  create  app/views/products/edit.html.erb
  create  app/views/products/show.html.erb
  create  app/views/products/new.html.erb
  create  app/views/products/_form.html.erb
  invoke  test_unit
  create  test/functional/products_controller_test.rb
  invoke  helper
  create  app/helpers/products_helper.rb
  invoke  test_unit
```

① http://guides.rubyonrails.org/getting_started.html#configuring-a-database

② 这条命令太长了, 无法在一行内完整输入。如果要输入多行命令, 只需要在每一行的末尾加反斜杠, 最后一行不需要添加。系统会提示你输入下一行。Windows 用户则需用补号符(^)来代替反斜杠。

```

create      test/unit/helpers/products_helper_test.rb
invoke      stylesheets
create      public/stylesheets/scaffold.css

```

生成器创建了一批文件。这里首先感兴趣的是迁移文件，即 `20110211000001_create_products.rb`。

一次迁移代表了对数据做出的一次修改，迁移放在一个独立于数据库的源文件中。这些修改既可以更新数据库模式，也可以更新数据库表中的数据。可以应用这些迁移来更新数据库，也可以撤销这些迁移来回滚对数据库的修改。本书中有一整章是关于迁移的（见第 23 章）。现在只是使用它们，并不多加解释。

迁移有一个基于 UTC 的时间戳前缀（`20110211000001`）、文件名（`create_products`）和文件扩展名（`.rb`，因为它是 Ruby 程序）。

实际看到的时间戳前缀会有所不同。事实上，在本书中所使用的时间戳显然是虚构的。时间戳通常不会是连续的，它们反映的是迁移的创建时间。

6.1.4 应用迁移

虽然已经告诉 Rails 每个属性的基本数据类型，但还需要继续完善价格的定义，价格属性共有 8 位有效数字，其中小数点后有两位。

```
depot_a/db/migrate/20110211000001_create_products.rb
```

```

class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price, :precision => 8, :scale => 2

      t.timestamps
    end
  end

  def self.down
    drop_table :products
  end
end

```

现在，迁移程序的修改已经完成了，可以用 **Rake** 命令让 Rails 在开发数据库中应用此迁移。**Rake** 就像个永远可靠的助手：告诉它做某个任务，那个任务就会完成。在这个例子中，会告诉 **Rake** 将所有尚未执行的迁移应用到数据库上：

```

depot> rake db:migrate
(in /Users/rubys/work/depot)
== CreateProducts: migrating =====
-- create_table(:products)
--> 0.0027s
== CreateProducts: migrated (0.0023s) =====

```

就是这样。**Rake** 搜索所有尚未应用到数据库的迁移，然后将它们应用到数据库上。在这个例子中，表 `products` 添加到了数据库中，`database.yml` 文件的 `development` 部分定义了这个数据库。

所有的基础工作已经完成。创建了一个 Rails 项目，即 Depot 应用程序；构造了开发用的数据库并配置了应用程序，使其可以连接到数据库；创建了商品的控制器和 Product 模型，并通过迁移生成了相应的 products 表；此外还生成了大量视图。现在是时候来看看所有这些内容了。

6.1.5 查看商品清单

上面的 3 条命令创建了应用程序和数据库（如果选择了其他的数据库而不是 SQLite 3，那么可能是现有的数据库中的一张表）。在关心后台具体发生了些什么之前，先尝试下这个网站。

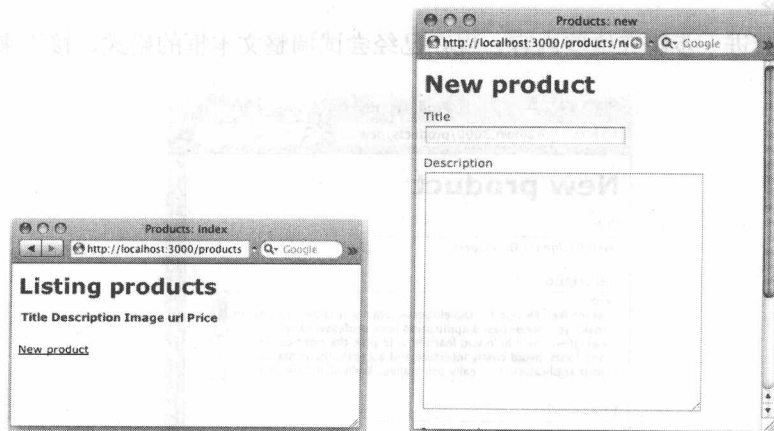
首先，启动 Rails 提供的本地服务器：

```
depot> rails server
=> Booting WEBrick
=> Rails 3.0.5 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2010-05-05 17:45:38] INFO WEBrick 1.3.1
[2010-05-05 17:45:38] INFO ruby 1.8.7 (2010-01-10) [x86_64-linux]
[2010-05-05 17:45:43] INFO WEBrick::HTTPServer#start: pid=24649 port=3000
```

正如第 2 章中演示的应用程序一样，这条命令启动本地主机上的 Web 服务器，端口 3000。如果在运行服务器时收到 Address already in use 错误信息，这就表示已经有一台 Rails 服务器在机器上运行了。如果一直按照本书中的例子在操作，这可能是第 4 章中的应用程序“Hello, World!”还在运行。可以在控制台上使用 Ctrl+C 快捷键来关闭服务器。如果是在 Windows 上运行，可能会看到提示信息：“Terminate batch job (Y/N)？”如果是这样的话，那么就输入 y。

接下来连接到服务器上。请记住，在浏览器中给出的 URL 要包含端口号（3000）和小写的控制器名（products）。

这个网页看起来很单调，只是一个空的商品清单。让我们再添加几个新商品，单击新商品的链接，就会弹出一个表单。



这几个表单都是简单的 HTML 模板，就像在 2.2 节中创建的那个表单一样。实际上，这些表单是可以修改的，现在来修改一下（商品）描述字段中的行数：

depot_a/app/views/products/_form.html.erb

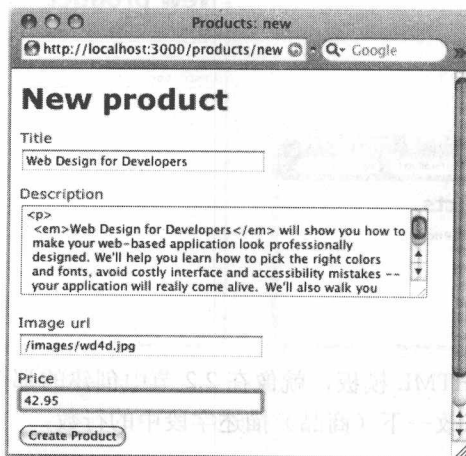
```

<%= form_for(@product) do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@product.errors.count, "error") %>
        prohibited this product from being saved:</h2>
      <ul>
        <% @product.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description, :rows => 6 %>
  </div>
  <div class="field">
    <%= f.label :image_url %><br />
    <%= f.text_field :image_url %>
  </div>
  <div class="field">
    <%= f.label :price %><br />
    <%= f.text_field :price %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

```

在第 8 章中会进一步研究这些内容。现在已经尝试调整文本框的格式，接下来输入内容：



Products: new

http://localhost:3000/products/new

New product

Title

Web Design for Developers

Description

<p>
Web Design for Developers will show you how to make your web-based application look professionally designed. We'll help you learn how to pick the right colors and fonts, avoid costly interface and accessibility mistakes -- your application will really come alive. We'll also walk you

Image url

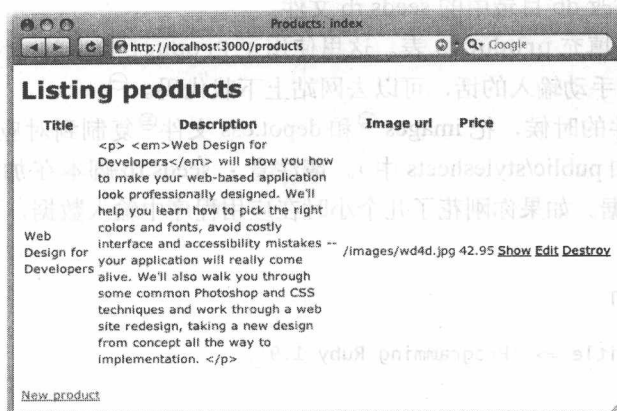
/images/wd4d.jpg

Price

42.95

Create Product

单击 create product（生成）按钮，应该可以看到生成的新商品。如果现在单击 Back（向后）的链接，会看到刚刚生成的新商品出现在清单中。



这个界面可能不是最漂亮的，但它能运行，可以演示给客户看，也可以展示别的功能（例如，查看商品详细信息、修改商品信息等）。我们会和客户解释，这只是第一步——我们也很清楚，这个用户界面是很粗糙的，但通过它可以更早地得到客户的反馈。（这4条命令可能算是在所有书中最早出现的。）

现在仅仅通过这4条命令就已经完成了许多工作了，在继续操作之前，先看一个其他的命令：

rake test

运行这条命令后，应该会看到输出中包含这两行，也就是 0 failures 和 0 errors。这是 Rails 根据这个脚手架生成的单元、功能和集成测试。目前这些测试是非常简单的，但至少知道了，我们已经有了这些可以运行的测试，而且应用程序也通过了测试，这也给了你信心。在第二部分的章节中应该多使用这条命令，因为它能帮助你检查和跟踪程序错误。7.2 节进一步介绍了这个命令。

如果用的数据库不是 SQLite 3，这一步可能会有错误。请根据 24.1 节中的注解，检查 database.yml 文件。

6.2 迭代 A2：美化商品清单

客户有个新的需求（客户似乎总是有新的需求，不是吗）：所有商品的清单看上去不太美观，能不能做得漂亮点儿？此外，可不可以通过图像 URL 来展示商品的照片呢？

现在我们面临一个两难的困境。开发人员对这类请求会有点儿头疼，下意识地摇摇头，喃喃地问：“你想要什么？”同时，开发出漂亮的界面也可以炫耀我们的技术。最后，事实上可以通过 Rails 非常方便地实现这些修改，然后打开信赖的编辑器进行工作。

在开始下一步之前，先来设想一下，如果能有一组一致的测试数据，那么就好了。我们就可以从浏览器中使用脚手架生成的接口来输入数据。然而，如果我们这样做，当之后的开发人员在我们的代码库的基础工作时，也必须做同样的事，而且，如果作为这个项目团队的一部分时，团

队的每个成员将必须输入自己的数据。我们希望能将数据以更可控的方式加载到数据库表中。事实上这是可行的，可以通过 Rails 来导入种子数据。

开始时，先简单修改 db 目录中的 seeds.rb 文件。

然后，添加代码来填充 products 表。这里使用了 Product 模型的 create 方法。以下是该文件的摘录。如果不想手动输入的话，可以去网站下载代码。^①

现在到了复制文件的时候，把 images^②和 depot.css 文件^③复制到对应的地方（即在应用程序目录 public/images 和 public/stylesheets 中）。请注意：seeds.rb 脚本在加载新数据时会删除 products 中的现有数据。如果你刚花了几个小时在应用程序中输入数据，可能不希望运行它！

```
depot_b/db/seeds.rb
```

```
Product.delete_all
# ...
Product.create(:title => 'Programming Ruby 1.9',
               :description =>
               %{<p>
                 Ruby is the fastest growing and most exciting dynamic language
                 out there. If you need to get working programs delivered fast,
                 you should add Ruby to your toolbox.
               </p>},
               :image_url => '/images/ruby.jpg',
               :price => 49.50)
# ...
```

（请注意，上面的代码中使用了 %{...}，这是双引号字符串文本的一种替代语法，这会在用长字符串时比较方便。还要注意的，因为它使用 Rails 的 create 方法，所以如果由于验证错误而不能插入记录，这个方法就会失败但不出现任何报错信息。）

要用测试数据填充 products 表，只需运行以下命令：

```
depot>rake db:seed
```

现在整理一下商品清单。为此有两项工作要做。最终还要写一些 HTML 代码，其中会运用 CSS 统一控制展示方式。但是为了达到这个目的，需要让浏览器去提取样式表。

需要个地方来存放 CSS 样式定义。所有脚手架生成的应用程序都使用目录 public/stylesheets 中的样式表 scaffold.css。我们不会修改这个文件，而是创建一个新的应用程序的样式表，depot.css，并将其放在同一目录下^④。

最后，要将样式表链接到 HTML 页面。如果查看当前创建的文件 .html.erb，不会找到任何样式表的引用。甚至不会发现 HTML 的 <head> 部分，通常会在这里使用样式表。相反，Rails 会使用一个单独的文件来为整个应用程序创建标准的页面环境。这个称为 application.html.erb 的

① http://media.pragprog.com/titles/rails4/code/depot_b/db/seeds.rb

② http://media.pragprog.com/titles/rails4/code/depot_b/public/images/

③ http://media.pragprog.com/titles/rails4/code/depot_b/public/stylesheets/depot.css

④ 假如你还没有这么做，可以从 http://media.pragprog.com/titles/rails4/code/depot_b/public/stylesheets/depot.css 下载这个样式表。

文件是 Rails 的页面布局, 在 layouts 目录中:

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

```
<!DOCTYPE html>
<html>
<head>
  <title>Depot</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body>

  <%= yield %>

</body>
</html>
```

stylesheet_link_tag 创建一个 HTML <link> 标签, 这个标签会从 public/stylesheets 目录中加载样式表。可以列出要链接的样式表名称, 但是因为它会加载所有的样式表, 所以我们先把它放在一边, 在 8.2 节中再进一步介绍它。

现在, 样式表已经到位, 可以使用基于表格的简单模板, 通过编辑 app/views/products 中的文件 index.html.erb 来替换由脚手架生成的视图:

```
depot_b/app/views/products/index.html.erb
```

```
<div id="product_list">
  <h1>Listing products</h1>

  <table>
    <% @products.each do |product| %>
      <tr class="<%= cycle('list_line_odd', 'list_line_even') %>">
        <td>
          <%= image_tag(product.image_url, :class => 'list_image') %>
        </td>

        <td class="list_description">
          <dl>
            <dt><%= product.title %></dt>
            <dd><%= truncate(strip_tags(product.description),
                          :length => 80) %></dd>
          </dl>
        </td>

        <td class="list_actions">
          <%= link_to 'Show', product %><br/>
          <%= link_to 'Edit', edit_product_path(product) %><br/>
          <%= link_to 'Destroy', product,
                :confirm => 'Are you sure?',
                :method => :delete %>

        </td>
      </tr>
```

```

<% end %>
</table>
</div>

<br />

<%= link_to 'New product', new_product_path %>

```

即使这只是个简单的模板，也用到了不少 Rails 内置的特性：

- 清单的行背景颜色是交替变化的。为了完成这件事，Rails 的帮助函数 `cycle` 通过将每行的 CSS 类设置为 `list_line_even` 或 `list_line_odd`，并在两个连续行之间自动切换样式名称。
- `truncate` 这个帮助函数用来显示描述字段的前 80 个字符。但在调用 `truncate` 之前，需要调用函数 `strip_tags` 去除描述字段中的 HTML 标签。
- 注意 `link_to 'Destroy'` 这行。看看它如何使用参数 `:confirm=>'Are you sure?'`。如果单击这个链接时，Rails 会让浏览器弹出一个对话框，要求确认删除该商品。（另外，可以在当前页面侧边栏里面看到这个行为的一些内幕消息。）

:method=>:delete 是做什么的？

你可能已经注意到，脚手架生成的销毁链接包括参数 `:method=>:delete`。这决定了在 `ProductsController` 类中调用哪个方法，同时也影响应该使用哪个 HTTP 方法。

浏览器使用 HTTP 与服务器交互。HTTP 定义了浏览器可以使用的一组动词以及何时可以使用哪个动词。例如，一个普通的超链接使用 HTTP 的 GET 来获得查询结果，它不应该有任何副作用。以这种方式使用该参数表明该超链接应使用 HTTP 的 DELETE 方法。Rails 使用此信息来确定这个请求会传到控制器中的哪个行为上。

需要注意的是，在浏览器中使用 PUT 和 DELETE 方法时，Rails 会用 POST 方法来替换。并在处理过程中附加一个参数，从而使路由器可以确定最初的目的，是 PUT 还是 DELETE。无论哪种方式，将不会缓存请求或触发网络爬虫。

我们将测试数据加载到数据库中，改写了显示商品清单的文件 `index.html.erb`，添加了 `depot.css` 样式表，并通过布局文件 `application.html.erb` 把该样式表加载到页面。现在，打开浏览器，输入网址 `http://localhost:3000/products`。最终的商品清单页面可能看起来就像下面这样：

我们自豪地给客户展示了新的商品清单网页，客户还是很满意的。现在是创建店面的时候了。

6.3 本章小结

在本章中，为 Depot 应用程序奠定了基础：

- 创建了一个开发数据库。
- 使用迁移创建并修改了开发数据库的模式。
- 创建了 `products` 表，并使用脚手架生成器编写了应用程序来维护这张表。
- 更新一个应用程序范围内的布局，以及和控制器相关的视图，以显示商品清单。



我们所做的并不需要花费很多工夫，并可以快速启动和运行。数据库对这个应用程序是至关重要的，但不必害怕——事实上，在许多情况下，可以推迟数据库的选择或干脆开始使用 Rails 提供的默认数据库。

在这个阶段，更重要的是确保模型的正确性。我们很快就会看到，即使在这个小的应用程序中，数据类型的简单选择并不总是完全满足所有模型属性的要求，这就是下一步将解决的问题。

练习时间

可以自己尝试以下任务：

- 如果精力充沛的话，可以尝试回滚迁移。只需输入以下内容：

```
depot> rake db:rollback
```

模式会变回以前的样子，products 表也会消失。再次调用 `rake db:migrate` 将重新创建它。也许还想要重新加载种子数据。在第 23 章中可以找到更多的信息。

- 1.5 节中曾经提到过版本控制，现在这非常重要，它会帮助保存所做的工作。如果碰巧选择了 Git（强烈推荐），这个软件前期需要做的配置非常少，基本上需要做的所有事情就是提供姓名和电子邮箱地址。在这之后，通常会有一个文件名为 `.gitconfig` 的文件保存在根目录下。这个文件看起来像这样：

```
[user]
name = Sam Ruby
email = rubys@intertwingly.net
```

可以用下面的命令验证配置：

```
depot> git repo-config --get-regexp user.*
```

Rails 也提供了一个名为 `.gitignore` 的文件，它告诉 Git 哪些文件不需要加入版本控制。

```
depot_b/.gitignore
.bundle
db/*.sqlite3
log/*.log
tmp/
```

请注意，因为这个文件名是以点开始的，基于 UNIX 的操作系统在默认情况下将不会把它显示在目录列表中。使用 `ls -a` 可以看到这个文件。

现在，已经完全配置好了 Git。唯一的任务是初始化一个储存库，添加所有文件，然后提交它们，在提交时可以把相应文件的说明信息作为参数一起提交：

```
depot> git init
depot> git add .
depot> git commit -m "Depot Scaffold"
```

这看上去似乎并不能令人非常振奋，但它确实意味着你可以更自由地实验了。如果不小心覆盖或删除一个文件，可以通过下面这个命令回到原来的状态：

```
depot> git checkout.
```

```
[user]
name = Sam Ruby
email = ruby@interweb.org
```

```
depot> git repo-cding --get-repoq user,
```

```
.pubdir
db:rd:173
log:log
tmp
```


第 7 章

任务 B：验证和单元测试

在本章中，我们将学习：

- 执行验证和报告错误
- 单元测试

现在，我们建立了商品的初始模型，并且 Rails 脚手架给这些数据提供了完整的维护应用程序。本章将专注于如何使模型更加坚固——以确保数据中发生的错误永远都不会提交给数据库——在后续的章节会处理 Depot 应用程序的其他方面。

7.1 迭代 B1：验证

在测试迭代 A1 的结果时，客户发现了一些错误。如果输入一个无效的价格，或忘记输入商品描述，应用程序还是会接收所输入的内容并添加到数据库中。当然没有商品描述是令人尴尬的，价格为 0 也会让客户赔钱，所以她要求在应用程序中添加验证。不允许那些无效的商品存入数据库中，如标题或描述字段为空、图像的 URL 或价格。

那么，在哪儿可以放置验证代码呢？模型层扮演了代码和数据库之间的看门人角色。那些应用程序用到的数据，无论是从数据库中读出的，还是存储到数据库中的，都会先通过模型层。因此，模型层将是放置验证代码的理想位置，它不会计较数据是来自表单还是来自该应用程序的操作处理。如果在写入数据库之前模型对它进行检查，那么数据库将不会因坏数据而损坏。

再来看下模型类的源代码（在 `app/models/product.rb` 文件中）：

```
class Product < ActiveRecord::Base
end
```

添加验证这活儿应该是相当干净利落的。在将数据写入数据库之前，先验证非空的文本字段。添加以下代码到现有的模型中：

```
validates :title, :description, :image_url, :presence => true
```

这个 `validates` 方法是个标准的 Rails 验证器。它会根据一个或多个条件来验证一个或多个模型字段。

`:presence=>true` 让验证器核实每个已命名的字段都存在，并且其内容不为空。如果尝试提交一个所有字段都为空的新商品，会发生什么？结果如图 7.1 所示。实验结果令人印象深刻：所有出错的字段都用红色高亮标出，并且在表单顶端还会看到所有错误的清单，每行描述一个错误。只用一行代码就实现这么多东西，还是挺不错的。你可能还注意到，在编辑并保存 `product.rb` 文件后，并不需要重新启动应用程序来测试修改。只要重新加载网页，就可以让 Rails 注意到对模式的修改，也就是说它会永远使用最新版本的代码。

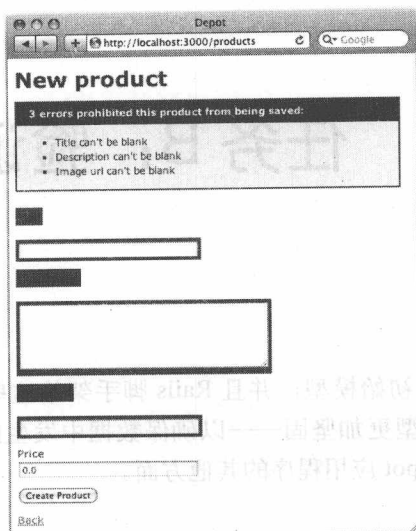


图 7.1 验证字段是否为空

我们也想验证价格是一个有效的正数。为此将用 `numericality` 这个选项，它可以判断输入的价格是否是一个有效的数字。此外还给选项 `:greater_than_or_equal_to` 传递一个为 0.01 的值（这个选项的名字真是相当冗长）：

```
validates :price, :numericality => {:greater_than_or_equal_to => 0.01}
```

现在，如果添加一个带无效价格商品的话，就会看到相应的报错消息，如图 7.2 所示。

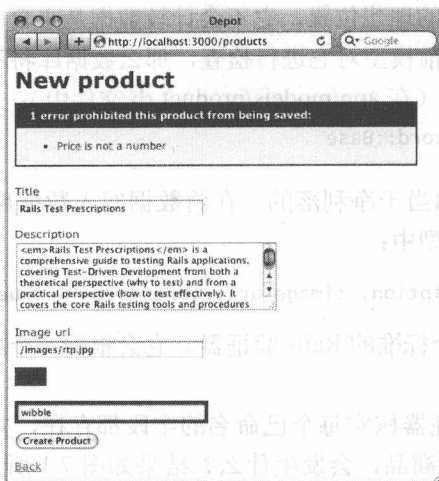


图 7.2 无效价格的验证

为什么用一分钱而不是 0 来测试？因为在这个字段中输入的数字可能是 0.001。如果和 0 相比，那么这个输入的价格会通过验证，但由于数据库只存储小数点后的两位数字，结果数据库中保存的商品价格还是为 0。所以检查输入的价格至少要有 1 美分，就可以确保只有正确的值才会

存储到数据库中。

还有两个字段要验证。首先，我们要确保每个商品都有唯一的标题。通过在 `Product` 的模型中再加一行来实现这个验证。唯一性验证将执行一个简单的检查，以确保 `products` 表中的没有其他商品和想要添加的新商品有相同的标题：

```
validates :title, :uniqueness => true
```

最后要验证所输入图像的 URL 是否有效。可用 `format` 选项来实现这个要求。这个选项可以判断字段是否和正则表达式相匹配。现在只检查 URL 是否是以 `.gif`、`.jpg` 或 `.png` 结尾。（正则表达式见 4.2 节。）

```
validates :image_url, :format => {
  :with => %r{\.(\.gif|jpg|png)$}i,
  :message => 'must be a URL for GIF, JPG or PNG image.'
}
```

以后可能要改变表单，让用户从一个可用图像的列表中选择，但这种情况还是要进行验证，以杜绝直接提交坏数据的可能性。

现在，在一两分钟内，已经添加了验证，用于检查以下几项内容：

- 字段的标题、描述以及图像的 URL 不是空的。
- 价格是一个有效的数字，且不得少于 \$0.01。
- 标题在所有商品中是唯一的。
- 图像的 URL 看起来是有效的。

修改后的 `Product` 模型看起来应该是这样：

```
depot_b/app/models/product.rb

class Product < ActiveRecord::Base
  validates :title, :description, :image_url, :presence => true
  validates :price, :numericality => { :greater_than_or_equal_to => 0.01 }
  validates :title, :uniqueness => true
  validates :image_url, :format => {
    :with => %r{\.(\.gif|jpg|png)$}i,
    :message => 'must be a URL for GIF, JPG or PNG image.'
  }
end
```

这个阶段快结束了，我们请客户测试这个应用程序，现在他开心多了。虽然只用了几分钟，但添加验证的简单动作使维护商品的页面看上去更加坚实了。

在继续之前，再测试一下：

```
rake test
```

糟糕。这一次有两个地方报错，一个在 `test_should_create_product` 中，另一个在 `test_should_update_product` 中。显然，先前添加验证的操作导致了创建和更新商品的失败。这并不令人惊奇。你想想看，这不就是验证的目的吗？

解决的办法是，在 `test/functional/products_controller_test.rb` 文件中给出有效的测试数据：

```
depot_c/test/functional/products_controller_test.rb
```

```
require 'test_helper'

class ProductsControllerTest < ActionController::TestCase
  setup do
    @product = products(:one)
    @update = {
      :title       => 'Lorem Ipsum',
      :description => 'Wibbles are fun!',
      :image_url   => 'lorem.jpg',
      :price       => 19.95
    }
  end

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:products)
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create product" do
    assert_difference('Product.count') do
      post :create, :product => @update
    end

    assert_redirected_to product_path(assigns(:product))
  end

  # ...

  test "should update product" do
    put :update, :id => @product.to_param, :product => @update
    assert_redirected_to product_path(assigns(:product))
  end

  # ...
end
```

做出修改后，重新运行测试，结果一切正常。但这只是说明我们并没有破坏什么。但要做的还不只是这些。需要确保的是，刚刚添加的验证代码不仅仅现在能工作，而且在进一步修改后，也能正常运行。8.4 节会更详细地介绍功能测试。至于现在，是时候来写一些单元测试了。

7.2 迭代 B2：模型的单元测试

Rails 框架的真正乐趣之一是，从每个项目开始时 Rails 就支持生成相应的测试。正如所看到的那样，从使用 rails 命令创建一个新的应用程序那一刻起，Rails 就开始生成一个基本的测试环境。

下面来看看 unit 的子目录中已经有了什么东西:

```
depot> ls test/unit
helpers product_test.rb
```

Rails 创建 `product_test.rb` 这个文件用来保存之前用脚本 `generate` 生成的模型单元测试。这是个良好的开端, 但 Rails 只能帮助我们这么多了。

来看看在生成模型时, Rails 在 `test/unit/product_test.rb` 文件里面为测试生成了什么:

```
depot_b/test/unit/product_test.rb

require 'test_helper'

class ProductTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

这个生成的 `ProductTest` 是 `ActiveSupport::TestCase` 的子类。而 `ActiveSupport::TestCase` 又是 `Test::Unit::TestCase` 的子类。这个事实说明, Rails 生成的测试基于 `Test::Unit` 框架。而这个框架是来自 Ruby 的。这是个好消息, 因为这意味着是否已经开始用 `Test::Unit` 测试来测试 Ruby 程序 (为什么不呢), 所以可以在这个基础上来测试 Rails 应用程序。如果对 `Test::Unit` 不了解, 不用担心。我们会慢慢来的。

在这个测试案例中, Rails 生成了一个单一的测试, 称为 “the truth”。刚开始的时候 `test...do` 的语法令人感觉奇怪, 这里的 Active Support 是由一个类方法、可选括号和一个代码块组成的。这个代码块会使定义测试方法稍微简单点。有时一些小事会使整件事情完全不同。

方法中 `assert` 这行是一个实际的测试。其实它什么都没做——它只是测试 `true` 为真。显然, 这是一个占位符, 但它的作用是非常重要的, 因为它可以让我们看到, 所需的测试环境已经到位。

7.2.1 真正单元测试

现在来看一下真实的测试验证。首先, 如果创建一个没有属性集的商品, 我们期待它是无效的, 而且会显示与每个字段关联的错误信息。那么可以使用该模型的 `errors` 和 `invalid?` 方法来验证它, 另外, 可以使用错误清单的 `any?` 方法来查看是否有和特定的属性相关联的错误。

现在知道要测试什么了, 接下来需要知道如何告诉测试框架该代码是否通过了测试。可以通过 `assertions` (断言) 来做到这一点。断言是一个简单的方法调用, 它告诉框架什么才是所期望的真。最简单的断言是方法 `assert`, 这个方法预期其参数为真。如果参数为真, 那么就不会发生什么特别的事情。但是, 如果 `assert` 的参数是假, 则断言失败。该框架将输出一条消息, 并且停止执行包含错误的测试方法。在本例中, 预计一个空的 `Product` 模型无法通过验证, 所以可以通过断言这种情况无效来表达这个期望。

```
assert product.invalid?
```

用下面的代码来代替 `the truth` 测试:

```
depot_c/test/unit/product_test.rb
```

```
test "product attributes must not be empty" do
  product = Product.new
  assert product.invalid?
  assert product.errors[:title].any?
  assert product.errors[:description].any?
  assert product.errors[:price].any?
  assert product.errors[:image_url].any?
end
```

可以通过命令 `rake test:units` 来再次运行单元测试。运行以下命令，就会看到测试成功运行了：

```
depot> rake test:units
Loaded suite lib/rake/rake_test_loader
Started
..
Finished in 0.092314 seconds.
1 tests, 5 assertions, 0 failures, 0 errors
```

果然，验证通过了，同时所有的断言也通过。

显然，这时候可以再深入一些，练习特殊的验证。这里只尝试众多测试中的3种。

首先，将检查价格验证是否如所期待的方式工作：

```
depot_c/test/unit/product_test.rb
```

```
test "product price must be positive" do
  product = Product.new(:title => "My Book Title",
                        :description => "yyy",
                        :image_url => "zzz.jpg")

  product.price = -1
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 0
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 1
  assert product.valid?
end
```

在这段代码中，创建了一个新商品，然后尝试设置其价格为-1、0和+1，每次都验证输入的数据。如果模型工作正常的话，那么前两个输入应该是无效的，并且证实了与 `price` 属性相关的错误消息，其内容正如所期望的那样。由于错误消息的列表是一个数组，因此可以使用非常方便的 `join`^① 方法来连接每个消息，我们表达断言，是基于“只有这样一个消息”的假设。

① <http://ruby-doc.org/core/classes/Array.html#M002182>

最后的价格是有效的，所以我们断言，现在该模型是有效的。（有些人会把这3种测试分为3个独立的测试方法——这是完全合理的。）

其次，我们将测试验证图像 URL 是否以 .gif、.jpg 或 .png 结尾。

```
depot_c/test/unit/product_test.rb
```

```
def new_product(image_url)
  Product.new(:title      => "My Book Title",
              :description => "yyy",
              :price      => 1,
              :image_url  => image_url)
end

test "image url" do
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
           http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }
  ok.each do |name|
    assert new_product(name).valid?, "#{name} shouldn't be invalid"
  end
  bad.each do |name|
    assert new_product(name).invalid?, "#{name} shouldn't be valid"
  end
end
```

在这里，已经将测试都混在一起了。而不是写9个独立的测试，上面的代码中使用了两个循环——第一个是检查期望通过验证的情况；第二个则是检查失败的情况。同时，提取出两个循环之间的共同的代码。

你会发现，代码中还添加了一个额外的参数给 `assert` 方法调用。所有的测试断言接收一个可选尾部参数，即一个字符串。如果断言失败，该字符串将与出错信息一起输出，这有助于诊断是什么出错了。

最后，该模型中还包含一个验证，这个验证会检查所有产品在数据库中的标题是否唯一。为了测试这个验证，将需要把商品数据存储在数据库中。

一种方法是，创造一个商品，然后保存它，然后再创建另一个商品，让它和那个已生成的商品有相同的标题，也保存到数据库中。这显然是可行的。但还有一个更简单的方法，可以使用 Rails 的 fixtures。

7.2.2 静态测试

静态测试 (test fixture) 在测试领域中是一个可以在其中运行测试的环境。例如，如果正在测试一个电路板，可以将它装载在一个静态测试中，这个静态测试提供测试功能所需要的电源和驱动功能测试所需要的输入。

在 Rails 的世界里，静态测试只是为待测试的一个或多个模型而准备的初始内容的规范。举例来说，如果要确保 `products` 表在每一个单元测试都从已知的数据开始，可以在一次静态测试中指定下面这些内容，Rails 会处理剩下的事情。

可以在 `test/fixtures` 目录内的文件中详细说明静态测试数据。这些文件包含以逗号分隔值 (CSV) 或 YAML 格式的测试数据。在测试中, 将首选 YAML 格式。每个静态测试文件包含单个模型的测试数据。静态测试文件的名称是很重要的; 文件的基本名称必须与数据库表的名称相匹配。因为需要给 `Product` 模型一些数据, 而这些数据存在 `products` 表中, 因此要把它添加到名为 `products.yml` 的文件中 (YAML 见 4.4 节)。

David 说

挑选好的静态测试名称

就像通常的变量名称一样, 静态测试名称也应尽可能不言自明, 通过名称就可以大致了解静态测试的用途。当用断言 `product(:valid_order_for_fred)` 来确认 Fred 的有效订单时, 静态测试的名称就增加了测试的可读性。这也让大家更容易地记住哪个是应该进行测试的静态测试, 而无需查找 `p1` 或 `order4`。静态测试越多, 那么好的静态测试名称就越重要。因此, 越早开始这么做就会使以后的工作越方便。

如果不能轻易地得到像 `valid_order_for_fred` 这样不解自明的静态测试名称, 怎么办呢? 使用一个可以容易联想到角色的自然名称。如用 `christmas_order` 而不是 `order1`, 用 `fred` 代替 `customer1`。一旦养成了用自然名称这个习惯, 那么就能编排出一个漂亮的小故事, 是关于 `fred` 如何先用他的无效信用卡 (`invalid_credit_card`) 来支付他的圣诞节订单 (`christmas_order`), 然后再用他的有效信用卡 (`valid_credit_card`) 来支付, 最后选择发货到 `aunt_mary` 那儿。

基于故事联想的方法是帮助记忆这个庞大的静态测试世界的关键。

当第一次生成这个模型时, Rails 已经生成了这个静态测试文件:

```
depot_b/test/fixtures/products.yml
```

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
```

```
one:
```

```
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

```
two:
```

```
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

这个静态测试文件每行都包含了一个要插入数据库的条目, 并且每行都给出了字段名。在 Rails 生成的静态测试中, 那些名为 `one` 和 `two` 的行对数据库是没有意义的——这些行不会插入该行数据中。但我们很快会看到, 这些名称提供了一个便捷的在测试代码中引用测试数据的途径。它们也是那些在生成的集成测试中所用的名称, 所以现在, 先把这些放在一边。

在每个条目中会看到一个名称/值组合的缩进列表。就像在 `config/database.yml` 文件中，每个数据行的开始，必须使用空格，而不是 Tab 键，并且数据库中与同一记录相关的所有行都必须有相同的缩进。在修改程序时请小心，因为必须确保每个条目中的列名是正确的，与数据库中的列不匹配名称可能导致难以跟踪的异常。

可以在静态测试文件中添加更多有用的数据来测试 `Product` 模型：

```
depot_c/test/fixtures/products.yml
```

```
ruby:
  title:      Programming Ruby 1.9
  description:
    Ruby is the fastest growing and most exciting dynamic
    language out there. If you need to get working programs
    delivered fast, you should add Ruby to your toolbox.
  price:      49.50
  image_url:  ruby.png
```

现在已经有了静态测试文件，在运行单元测试时，我们想让 Rails 把这些测试数据加载到 `products` 表中。实际上 Rails 已经在做这个了（这正是约定胜于配置原则的好处），但也可以通过在文件 `test/unit/product_test.rb` 中指定以下这行来控制加载哪些静态测试：

```
class ProductTest < ActiveSupport::TestCase
  ▶ fixtures :products
  #...
end
```

在测试案例中的每个测试方法运行前，该静态测试将给定名称的模型名对应的静态测试数据加载到对应的数据库表中。静态测试文件名决定要加载的表，因此用 `:products` 就意味着会使用 `products.yml` 静态测试文件。

换句话说。在 `ProductTest` 类中输入 `fixtures` 指令意味着：在运行每个测试方法前，`products` 表将清空，然后用静态测试中定义的那 3 行来填充表。

请注意，大部分 Rails 生成的脚手架并不包含调用 `fixtures` 方法。这是因为在运行测试之前默认加载所有静态测试。因为默认一般都是你想要的，通常不需要去改变它。再说一次，约定就是用来消除不必要的配置需要的。

`products` 方法通过加载静态测试为生成的表创建索引。需要修改这个索引来匹配在静态测试中所给出的名称。

目前为止一直是在开发数据库中工作。但现在我们是在进行测试，Rails 需要使用一个测试数据库。如果看看 `config` 目录中的 `database.yml` 文件，会发现实际上 Rails 为 3 个独立的数据库创建了配置：

- `db/development.sqlite3` 是开发数据库。所有的编程工作将在这里完成。
- `db/test.sqlite3` 是一个测试数据库。
- `db/production.sqlite3` 是实际产品数据库。应用程序正式上线时就使用这个数据库。

在测试数据库中每个测试方法都有一张刚刚初始化的表，加载了所提供的静态测试数据。这是由命令 `rake test` 自动完成的，但也可单独运行 `rake db:test:prepare` 来初始化数据库表。

7.2.3 使用静态测试数据

我们已经知道如何将静态测试数据放到数据库中，现在要找到在测试中使用这些数据的方法。

显然，一种办法是使用模型中的 `finder` 方法来读取数据。但是 Rails 有比这更容易的解决方案。对于每一个加载到测试中的静态测试，Rails 定义具有和静态测试相同名称的方法。可以使用此方法来访问已经预装了的、包含了静态测试数据的模型对象：简单地传递 YAML 静态测试文件中定义的行名，它会返回包含该行数据的模型对象。在商品数据这个例子中，调用 `products(:ruby)` 返回一个 `Product` 模型，其中包含了在静态测试中定义的数据。使用这个模型来验证商品名称的唯一性：

```
depot_c/test/unit/product_test.rb
```

```
test "product is not valid without a unique title" do
  product = Product.new(:title => products(:ruby).title,
                        :description => "yyy",
                        :price => 1,
                        :image_url => "fred.gif")

  assert !product.save
  assert_equal "has already been taken", product.errors[:title].join('; ')
end
```

这个测试假设数据库已经包含了 Ruby 这本书的数据行。它通过下面这个语句获得此行的标题：

```
products(:ruby).title
```

然后，它创建一个新的 `Product` 模型，并设置它的标题为数据库中已存在的标题。它断言，试图保存这个模型会失败，并且输出和 `title` 属性相关的错误信息。

如果想避免在 Active Record 错误中使用硬编码的字符串，可以将返回的消息和其内置的错误信息表进行比较来解决这个问题：

```
depot_c/test/unit/product_test.rb
```

```
test "product is not valid without a unique title - i18n" do
  product = Product.new(:title => products(:ruby).title,
                        :description => "yyy",
                        :price => 1,
                        :image_url => "fred.gif")

  assert !product.save
  assert_equal I18n.translate('activerecord.errors.messages.taken'),
               product.errors[:title].join('; ')
end
```

在第 15 章中将会介绍 `I18n` 函数。

现在，我们可以自信地说，验证代码不仅工作正确，而且还会继续工作。该商品现在有一个模型、一组视图、一个控制器和一组单元测试。它将为构建应用程序的其余部分打下良好基础。

7.3 本章小结

只是写了十几行代码，就为所生成的代码添加了验证：

- 确保所需的字段不为空。
- 保证价格字段值为数字并且至少为一美分。
- 保证标题是唯一的。
- 确保图像匹配给定的格式。
- 修改了 Rails 提供的单元测试，让这些测试既符合给模型加上的限制，又能验证新添加的代码。

我们给客户展示这些成果，虽然他同意这些功能可以给管理员使用，但可以肯定的是，这绝不是他想要给网站顾客提供的服务。显然，在未来的迭代中，将不得不把重点放在用户界面上。

练习时间

可以自己尝试以下任务：

- 如果用的是 Git，现在可能是提交工作的恰当时机。可以先通过使用 `git status` 命令来看看修改了哪些文件：

```
depot> git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   app/models/product.rb
# modified:   test/fixtures/products.yml
# modified:   test/functional/products_controller_test.rb
# modified:   test/unit/product_test.rb
# no changes added to commit (use "git add" and/or "git commit -a")
```

由于只修改了一些现有的文件并没有增加任何新的文件，可以结合 `git add` 和 `git commit` 命令，然后只要运行带 `-a` 参数的 `git commit` 命令：

```
depot> git commit -a -m 'Validation!'
```

完成上述命令之后，可以随意修改代码或删除文件，现在很安全，因为我们知道可以在任何时候使用一个简单的 `git checkout` 命令返回当前状态。

- 验证参数 `:length` 检查模型属性的长度。添加这个验证到 `Product` 模型可以检查标题是否满足至少 10 个字符大小的限制。
- 更改和验证相关的错误显示消息。

(你可以在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 找到相关的帮助信息。)

第 8 章

任务 C：商品目录显示

在本章中，我们将会学习：

- 编写用户视图
- 使用页面布局来装点页面
- 集成 CSS
- 使用帮助程序
- 编写功能测试程序

总的来说，前面已经进行了一组成功的迭代。从客户处收集了初始的用户需求，记录了基本流程，通过了即将需要的第一次数据测试，也让 Depot 应用程序的商品有了维护网页。尽管还没有编写什么代码，但是已有了短小而在不断扩充的测试套件。

有了这些基础工作，就可以开始下一个任务了。与客户讨论了任务的优先级，他说想先看看，从买家的角度，这个应用程序是什么样子。接下来的任务就是，创建简单的商品目录显示网页。

在我们看来，这样的要求也是很合理的。一旦把商品安全地导入数据库，显示它们就应该非常简单。接下来，开发购物车部分的代码也是以此为基础的。

这样应该可以模仿前面做过的商品维护任务，把它应用到当前的工作中来——这个商品目录显示其实只是个修饰过的商品清单。

最后，借用该控制器的功能测试，还将要完善模型单元测试。

8.1 迭代 C1：创建商品目录清单

前面已经创建了商品控制器，卖家可以用它来管理 Depot 应用程序。现在是时候创建第二个控制器了，它将用来与消费者进行互动，称为 Store（商店）。

```
depot> rails generate controller store index
create app/controllers/store_controller.rb
route get "store/index"
invoke erb
create app/views/store
create app/views/store/index.html.erb
invoke test_unit
create test/functional/store_controller_test.rb
invoke helper
create app/helpers/store_helper.rb
invoke test_unit
create test/unit/helpers/store_helper_test.rb
```


就像前面章节那样, 为了管理这些商品, 曾经使用了 `generate` 功能, 以便创建控制器和与之相关的支架代码, 这里已经调用它, 并且创建了控制器 (在文件 `store_controller.rb` 中 `StoreController` 类), 其中包含了单一的行为方法 `index`。

对于通过网址 `http://localhost:3000/store/index` 访问该行为, 尽管已经一切就绪 (放手去试试), 但是还可以做得更好。为了简化用户操作, 可以使其成为根网址。通过编辑文件 `config/routes.rb` 可以达到这样的效果:

```
depot_d/config/routes.rb
```

```
Depot::Application.routes.draw do
  get "store/index"
```

```
  resources :products
```

```
  # ...
```

```
  # You can have the root of your site routed with "root"
```

```
  # just remember to delete public/index.html.
```

```
  # root :to => "welcome#index"
```

```
  ▶ root :to => 'store#index', :as => 'store'
```

```
  # ...
```

```
end
```

可以看到, 在文件的最上方增加了几行代码来支持商店和商品的控制器。现将保留这几行代码。紧接着会看到一行已经注释了的代码, 这是用来定义根网址的。既可以取消对该行的注释, 也可以紧跟其后增加一行新代码。在这行代码中要修改的是控制器的名字 (把 `welcome` 改为 `store`), 并且还加上 `:as=>store`。后者告诉 Rails 去创建 `store_path` 变量, 就像在 2.3 节中看到的变量 `say_goodbye_path` 一样。

注意, 注释内容提示了, 要删除文件 `public/index.html`。现在就来做这件事情: [⊖]

```
depot> rm public/index.html
```

试一下, 在浏览器地址栏里输入 `http://localhost:3000/`, 将弹出下面这个网页。



⊖ Windows 用户可以执行命令 `erase public\index.html`; 如果用 Git 管理源代码的话, 那么使用命令 `git rm public/index.html`。

尽管页面看起来不是很充实，但是至少知道了，所有事情都正确地连接在一起了。这个页面甚至还说明了，在哪里可以找到形成这个页面的模板文件。

从显示简单清单开始。该清单包含了数据库中的所有商品。当然，最终的结果肯定会更复杂，要把它们分门别类地排列，但目前，这个清单足以让工作继续进行。

需要得到商品清单而非源自数据库，且把它提供给显示清单的视图代码。这意味着，得改变 `store_controller.rb` 中的 `index` 方法。要在合适的抽象层面上进行编程，这样就先假定可以从模型中得到可销售的商品清单：

```
depot_d/app/controllers/store_controller.rb

class StoreController < ApplicationController
  def index
    ▶ @products = Product.all
    end

  end
end
```

先征询客户对商品的显示顺序是否有偏好，然后共同确定了以字母顺序来显示商品清单，来看看效果如何。为了做到这一点，可以打开模型 `Product` 添加方法 `default_scope`。默认范围函数 (scopes) 会作用于该模型的所有查询。

```
depot_d/app/models/product.rb

class Product < ActiveRecord::Base
  ▶ default_scope :order => 'title'

  # validation stuff...
end
```

现在需要编写视图模板。通过编辑路径 `app/views/store` 下的 `index.html.erb` 文件，可以实现这一点。（记住，视图路径名称来自控制器名 `[store]` 及其行为方法名 `[index]`。文件扩展名 `.html.erb` 部分表示，使用 ERb 模板来产生 HTML 结果。）

```
depot_d/app/views/store/index.html.erb

<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">
      <span class="price"><%= product.price %></span>
    </div>
  </div>
<% end %>
```

要注意关于商品属性 description（描述）所使用的方法 `sanitize`。它允许安全地添加 HTML 风格代码，使客户对这一商品属性描述内容更有感兴趣。（注意，这个决定存在潜在的安全漏洞，但是因为商品描述都是由公司员工所创建的，所以可以认为该风险很小。详细情况请看 25.2 节讨论。）

前面已经使用了帮助函数 `image_tag`。这里使用其参数作为图像源，就生成 HTML 标签 ``。

单击刷新将显示如图 8.1 所示内容。之所以看起来并不是很漂亮，是因为还没有加入 CSS 样式。当正在琢磨这件事情时，客户恰巧经过，他指出，他想要在公开页面上看到比较漂亮的标题和侧边栏。

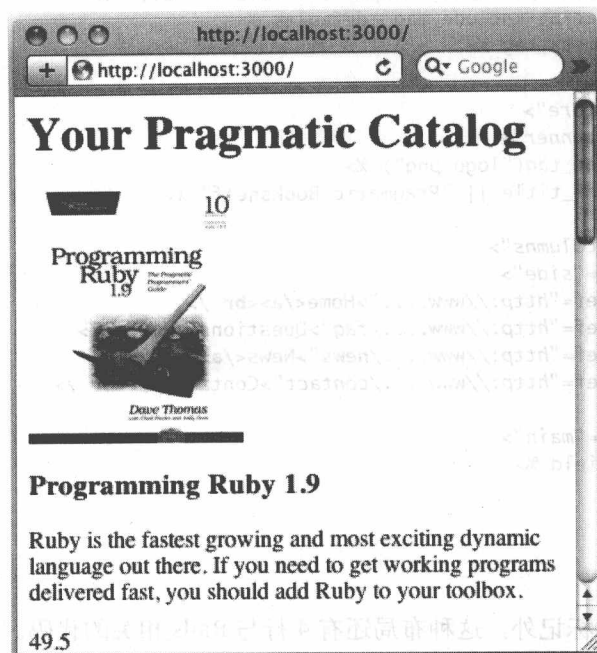


图 8.1 第一个（不美的）目录网页

在现实世界里碰到这种事情，恐怕只能求助于那些专业网页设计人员了——毕竟我们看到了太多由程序员自己做出来的网页，并不是那么赏心悦目。但是务实的网页设计师正在海滩的某个角落寻找灵感，并且年底之前都回不来，因此，现在先暂时搁置这一话题。该是做另一次迭代的时候了。

8.2 迭代 C2: 增加页面布局

通常网站页面会共享相似的页面布局——设计人员将已生成的标准模板用来填写内容。下面的工作是将这个网页装饰添加到商店的每个网页上。

将这个页面布局命名为 `application.html.erb`，在没有其他页面布局的情况下，所有控制器的

视图都将使用这个布局。由于只使用单个布局，因此只需要编辑该文件就可以修改整个网页的界面外观。现在先设置一个占位（placeholder）页面布局；这样可以等设计人员从岛上旅游回来后再进行更新，这是一个好办法。

将这个布局文件放在 `app/views/layouts` 目录中：

```
depot_e/app/views/layouts/application.html.erb
Line 1  <!DOCTYPE html>
      <html>
      <head>
        <title>Pragprog Books Online Store</title>
        <%= stylesheet_link_tag "scaffold" %>
        <%= stylesheet_link_tag "depot", :media => "all" %>
        <%= javascript_include_tag :defaults %>
        <%= csrf_meta_tag %>
      </head>
10     <body id="store">
      <div id="banner">
        <%= image_tag("logo.png") %>
        <%= @page_title || "Pragmatic Bookshelf" %>
      </div>
15     <div id="columns">
      <div id="side">
        <a href="http://www....">Home</a><br />
        <a href="http://www..../faq">Questions</a><br />
        <a href="http://www..../news">News</a><br />
20     <a href="http://www..../contact">Contact</a><br />
      </div>
      <div id="main">
        <%= yield %>
      </div>
25   </div>
      </body>
      </html>
```

除了通常的 HTML 标记外，这种布局还有 4 行与 Rails 相关的代码。第 6 行使用了 Rails 帮助函数，以便生成 `depot.css` 样式的 `<link>` 标签。第 8 行设置了一切幕后所需要的数据，用来防止跨站点请求的外部攻击，在第 12 章中添加表单时，这将是重要的。第 13 行把页面标题设置为实例变量 `@page_title` 值。真正的魔术出现在第 23 行，当调用运行方法 `yield`（参见 4.3 节）时，Rails 会自动替换页面相关的内容——由该请求调用的视图所生成的内容。在这里将是 `index.html.erb` 所生成的目录页。

要运行这段代码，还要在 `depot.css` 样式内加入以下代码：

```
depot_e/public/stylesheets/depot.css
/* Styles for main page */

#banner {
  background: #9c9;
  padding-top: 10px;
  padding-bottom: 10px;
```

```

border-bottom: 2px solid;
font: small-caps 40px/40px "Times New Roman", serif;
color: #282;
text-align: center;
}

#banner img {
  float: left;
}

#columns {
  background: #141;
}

#main {
  margin-left: 17em;
  padding-top: 4ex;
  padding-left: 2em;
  background: white;
}

#side {
  float: left;
  padding-top: 1em;
  padding-left: 1em;
  padding-bottom: 1em;
  width: 16em;
  background: #141;
}

#side a {
  color: #bfb;
  font-size: small;
}

```

单击刷新后, 浏览器窗口看起来如图 8.2 所示。它不会赢得任何设计奖项, 但它会告诉客户最终网页看上去大概是什么样子。

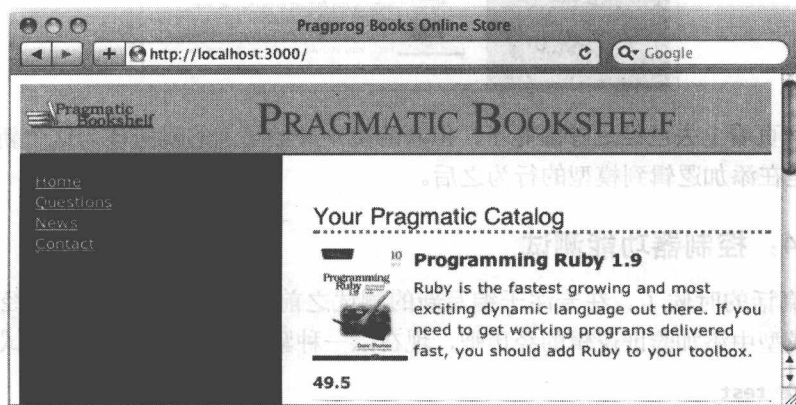


图 8.2 添加布局后的分类目录

仔细观察这个页面，发现显示的商品价格有个小问题。在数据库中价格是以数字形式存储的，但希望它能以美元和美分形式显示。价格 12.34 应显示为 \$12.34，13 则应显示为 \$13.00。下一节将解决这个问题。

8.3 迭代 C3：用帮助函数来调整价格格式

Ruby 提供了 `sprintf` 函数，可以用来对价格进行格式化。可以直接在视图中调用这个函数。例如，以下面这种方式进行调用：

```
<span class="price"><%= sprintf("%0.02f", product.price) %></span>
```

这样做当然没有错，但将货币格式写入视图里。如果以后想国际化这个应用程序，那么这就会有维护问题。

相反，如果能有个帮助函数把价格转换成货币格式，那就好了。Rails 恰好有这样的内置函数，即函数 `number_to_currency`。

在视图中可以非常方便地调用这个函数；在这个索引模板中，将下面这行

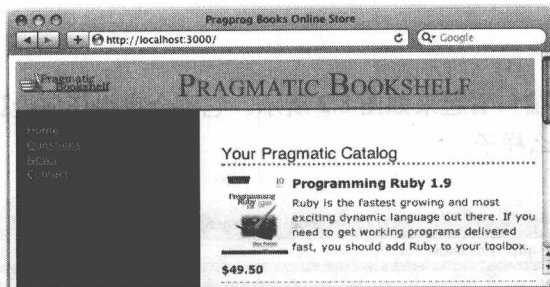
```
<span class="price"><%= product.price %></span>
```

改成：

```
depot_e/app/views/store/index.html.erb
```

```
<span class="price"><%= number_to_currency(product.price) %></span>
```

果然，单击刷新后，就看到了漂亮格式的价格：



虽然这个网页看上去已经足够漂亮了，但新的挑剔开始了，的确应该为这个新功能运行和编写测试，特别是在添加逻辑到模型的行为之后。

8.4 迭代 C4：控制器功能测试

现在该说真话的时候了。在专注于编写新的测试之前，需要确定，是否已经破坏了什么东西。还记得在模型中添加验证逻辑的经历吧，现在以一种紧张心情，再次运行测试：

```
depot> rake test
```

这次，一切都好。尽管加了很多东西，但没有破坏任何东西。这只是松了口气，但工作还没

做完；还要测试刚刚加上的功能。

之前所做的模型单元测试看上去是相当清楚的。调用方法，且比较其返回值和期待值。但当前在浏览器中所涉及的不仅是服务器处理请求，而且用户也注视着响应。我们需要的是功能测试，它能验证模型、视图和控制器是否在一起正常工作。别害怕，Rails 可以很容易地实现这一点。

首先，看一下 Rails 生成了什么：

```
depot_d/test/functional/store_controller_test.rb

require 'test_helper'

class StoreControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
  end
end
```

这个 should get index 测试得到了索引并断言期待成功响应。这个看上去是够直截了当的。那是个合理的开始，但还想验证响应是否包含了布局、商品信息和数字格式。看一看如下代码：

```
depot_e/test/functional/store_controller_test.rb

require 'test_helper'

class StoreControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    ▶ assert_select '#columns #side a', :minimum => 4
    ▶ assert_select '#main .entry', 3
    ▶ assert_select 'h3', 'Programming Ruby 1.9!'
    ▶ assert_select '.price', /\$[,\.d]+\.\d\d/
  end
end
```

上面所添加的这4行代码，通过CSS选择器表示法，查看所返回的HTML代码。先复习一下CSS，以井号(#)开始的选择器是与id属性相匹配的，以点开始的(.)的选择器是与class属性相匹配的，没前缀的选择器是与该元素名相匹配的。

所以，第一个选择测试寻找的是名为a的元素，在id值为side的元素中包含了该元素，而在id值为columns的元素中又包含这个id值为side的元素。这个测试是验证所有满足以上条件的这样的元素中最小值为4。方法assert_select是相当厉害的家伙，不是吗？

接下来的3行代码验证显示所有商品。第一行验证在页面的主要部分有3个类名为entry元素。下一行验证的是，存在具有Ruby书名的h3元素，之前已经输入了该书名。第3行验证的是：价格格式是否正确。这些断言都是基于静态测试中的测试数据：

```
depot_e/test/fixtures/products.yml
```

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
```

```
one:
```

```
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

```
two:
```

```
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

```
ruby:
```

```
  title:      Programming Ruby 1.9
  description:
    Ruby is the fastest growing and most exciting dynamic
    language out there. If you need to get working programs
    delivered fast, you should add Ruby to your toolbox.
  price:      49.50
  image_url:  ruby.png
```

注意到，方法 `assert_select` 执行的测试类型随着第 2 个参数类型而发生变化。如果这个参数是数字，那就当做数量对待它。如果是字符串，那就当做一个期待的结果处理它。另一种有用的测试类型是正则表达式，最后的就用到了那个断言。验证存在值的价格，该值包含美元符号，该符号紧接着任何多个数字（但至少有一个）、逗号或数字；然后是小数点；小数点后有两位数字（正则表达式见 4.2 节）。

在继续开发前还有最后一点：验证和功能测试都仅测试控制器的行为；它们不会影响任何对象，因为该对象已经在数据库或静态测试中。在上例中，两个商品有相同的标题。这样的数据将不会导致任何问题，而且将不会察觉到修改和保存这些记录，不会对现有的对象产生任何影响。

这里仅仅介绍了方法 `assert_select` 的一小部分功能。更多关于方法 `assert_select` 的信息请参见在线文档。[⊖]

只用几行代码就做了许多验证。可以看出，只要重新运行功能测试，这些验证就开始工作了（毕竟这是所做的全部修改）：

```
depot> rake test:functionals
```

现在不仅有了能够认可的事情如网站店面，而且还有相应测试代码以确保所有组件（模型、视图和控制器）都在一起正常地工作，并生成所需的结果。虽然这听起来好像很多，但是用 Rails 是很容易实现的。事实上，这里主要用到的是 HTML 和 CSS，并没有太多的代码或测试。

⊖ <http://api.rubyonrails.org/classes/ActionDispatch/Assertions/SelectorAssertions.html>

8.5 本章小结

上面我们将商店显示目录的那些基本部件组装在了一起。具体步骤如下:

- 创建了新的控制器来处理以用户为中心的交互。
- 实现了默认行为 `index`。
- 为模型 `Product` 添加了方法 `default_scope` 来指定网页上商品项目清单的顺序。
- 实现了视图(一个扩展名为 `.html.erb` 文件)及其页面布局(另一个也是扩展名为 `.html.erb` 文件)。
- 用帮助函数对价格进行想要格式转换的方式。
- 使用 CSS 样式表。
- 编写控制器功能测试。

现在是上传这些代码的时候,然后开始下一个任务,即创建购物车!

练习时间

可以自己尝试以下任务:

- 在网页侧边栏添加时间和日期。这是不需要自动更新的;只要打开网页时,显示它们就可以了。
- 尝试为 `number_to_currency` 帮助函数设置不同参数,并且看一下商品目录清单的不同效果。
- 使用函数 `assert_select` 编写商品维护应用程序的功能测试。这些测试要放在 `test/functional/products_controller_test.rb` 文件中。
- 正好提醒一下——在迭代结束时是用 Git 保存工作的好时候。如果一直在跟随本书一起工作,那么此时需要做个版本号。16.2 节将复习这些内容,并探索更多 Git 功能。(在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 可以找到相关提示。)

第 9 章

任务 D：创建购物车

在本章中，我们将学习：

- 会话和会话管理
- 添加模型间的关系
- 创建一个按钮，可添加产品到购物车中

现在已经可以显示目录了，目录包含了所有的精彩产品。如果能够卖了它们那就更好了。客户当然也认同，因此客户和我们共同决定下一步来实现购物车功能。这将涉及一些新的概念，包括会话、模型间关系以及在视图中添加一个按钮。让我们开始吧。

9.1 迭代 D1：寻找购物车

当用户浏览在线目录时，他们将（当然也是我们的希望）选择要购买的产品。通常的习惯是每个所选的产品会添加到在线商店的虚拟购物车中。买家选完商品后，就会去在线商店的收银台给购物车中的东西付款。

这意味着，应用程序需要跟踪所有由买家添加到购物车中的商品。要做到这一点，需要把购物车放在数据库中，并在会话中存储该购物车的唯一标识符，`cart.id`。每当请求出现时，可以从会话中找到该购物车的标识，并用该标识在数据库中查找购物车。

让我们继续创建一个购物车。

```
depot> rails generate scaffold cart
...
depot> rake db:migrate
== CreateCarts: migrating =====
-- create_table(:carts)
   -> 0.0012s
== CreateCarts: migrated (0.0014s) =====
```

Rails 的会话看上去像控制器的一个散列，所以将这个购物车 id 放在会话中，其可以使用符号 `:cart_id` 建立索引。

```
depot_f/app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  protect_from_forgery

  private

  def current_cart
    Cart.find(session[:cart_id])
  rescue ActiveRecord::RecordNotFound
```

```

▶      cart = Cart.create
▶      session[:cart_id] = cart.id
▶      cart
▶    end
end

```

方法 `current_cart` 先从 `session` 对象中得到 `:cart_id`，然后试图找到和这个 `id` 相对应的购物车。如果没找到（如果 `id` 是 `nil` 或由于某个原因而无效的时候，就会找不到，参见 4.3.3 节），该方法会创建一个新的 `Cart`，这个新购物车的 `id` 会保存在会话中，然后该方法返回新创建的购物车。

注意，我们把方法 `current_cart` 放在 `ApplicationController` 中，并定义该方法为私有的（参见 4.4.1 节）。这样的话，只有控制器才能调用该方法，同时也避免了 Rails 让该方法像控制器中的行为那样被调用。

9.2 迭代 D2: 将产品放到购物车中

我们现在正盯着会话看，因为我们需要有个地方来存放购物车。在 20.3 节中会更深入地研究这个会话。现在让我们先来实现购物车功能。

让事情变得简单点。购物车包含了一些选购的货品。基于 5.2 节中的图 5.3，同时我们又和客户简短地进行了沟通，现在可以创建 Rails 模型并应用迁移来创建相应的数据库表：

```

depot> rails generate scaffold line_item product_id:integer cart_id:integer
...
depot> rake db:migrate
== CreateLineItems: migrating =====
-- create_table(:line_items)
-> 0.0013s
== CreateLineItems: migrated (0.0014s) =====

```

表建完了，现在在数据库中有地方来存放在线商品、购物车和产品之间的关系了。然而，Rails 应用程序并不是把这些关系放在数据库中。我们需要在模型文件中添加一些声明来说明它们之间的关系。

在目录 `app/models` 下打开新建的文件 `cart.rb`，然后添加一个对 `has_many` 的调用：

```

class Cart < ActiveRecord::Base
▶   has_many :line_items, :dependent => :destroy
end

```

指令中 `has_many :line_items` 部分应该是自明的：一个购物车（潜在的）有许多相关联的在线商品。因为每个在线商品都包含一个对该购物车 `id` 的引用，所以这些商品都被关联到该购物车上。`:dependent => :destroy` 部分表示在线商品的存在依赖于购物车是否存在。如果我们销毁购物车，将其从数据库中删除，则会删除所有和该购物车相关联的商品。

接下来，我们将从相反的方向来定义连接，从在线商品到 `carts` 和 `products` 表。为了实现这个，要在文件 `line_item.rb` 中使用两次 `belongs_to` 声明：

```
depot_f/app/models/line_item.rb
```

```
class LineItem < ActiveRecord::Base
  ▶ belongs_to :product
  ▶ belongs_to :cart
  end
```

`belongs_to` 告诉 Rails 数据库，表 `line_items` 中的数据行是依赖于表 `carts` 和表 `products` 中的数据行的。在线商品是不可以单独存在的，除非数据库中有相对应的购物车和产品。有个简单的方法可以记住在哪里存放 `belongs_to` 声明：如果一个数据库表有外键，那么在相应的模型中每个外键都要有个 `belongs_to` 声明。

刚才这些不同的声明是干什么的？它们的基本目的是给模型对象添加导航的能力。因为我们给 `LineItem` 添加了 `belongs_to` 声明，那么现在我们能够获得 `LineItem` 的 `Product` 并展示其书名：

```
li = LineItem.find(...)
puts "This line item is for #{li.product.title}"
```

因为 `Cart` 定义为可以有多个在线商品的，所以我们可以从购物车对象中引用它们（作为集合）：

```
cart = Cart.find(...)
puts "This cart has #{cart.line_items.count} line items"
```

现在，出于完整性的考虑，应该在模型 `Product` 中也添加一个 `has_many` 指令。毕竟，如果我们有多个购物车的话，每个产品也可以有多个在线商品引用它。这次，我们用验证代码来避免删除那些正在被在线商品引用的产品。

```
depot_f/app/models/product.rb
```

```
class Product < ActiveRecord::Base
  default_scope :order => 'title'
  ▶ has_many :line_items
  ▶ before_destroy :ensure_not_referenced_by_any_line_item

  #...

  ▶ private

  ▶ # ensure that there are no line items referencing this product
  ▶ def ensure_not_referenced_by_any_line_item
  ▶   if line_items.empty?
  ▶     return true
  ▶   else
  ▶     errors.add(:base, 'Line Items present')
  ▶     return false
  ▶   end
  ▶ end
  end
```

这里我们声明了一个产品有多个在线商品，并定义了一个 hook（钩子）方法叫 `ensure_not_referenced_by_any_line_item`。hook 方法就是在对象的生命周期中某个给定的地方，

Rails 会自动调用的方法。在该例中, Rails 在尝试删除数据库中的一个数据行之前, 会先调用该方法。如果钩子方法返回 `false`, 那就不会删除该行。

注意, 在这里我们直接访问了 `errors` 对象。 `validates` 也会把错误信息放在这个对象里。错误信息可以关联到单个属性上, 但在本例中将错误信息直接与基本类关联。

我们会在 19.2 节中进一步讨论内部模型间的关系。

9.3 迭代 D3: 添加一个按钮

现在模型间的关系处理完毕, 该给每个产品添加一个 Add to Cart 按钮了。

我们不必创建一个新的控制器, 甚至也不必创建一个新的行为。来看一下由脚手架生成器提供的那些行为: `index`、`show`、`new`、`edit`、`create`、`update` 和 `destroy`, 我们要添加的这个按钮的操作正好可以用行为 `create`。(行为 `new` 可能听上去也差不多, 但这个行为通常是要获得一个表单, 可在其中输入某些信息, 然后再由行为 `create` 继续下去。)

一旦做出决定, 那就继续做下去。我们正在创建什么呢? 肯定不是 `Cart`, 更不是 `Product`。我们正在创建的是 `LineItem`。看看 `app/controllers/line_items_controller.rb` 中的关于 `create` 方法的注释, 你可以发现这个选择也决定了要用的 URL (`/line_items`) 和 HTTP 方法 (`POST`)。

这个决定甚至建议了要用的合适的用户界面控制。在添加链接前, 我们使用了 `link_to`, 但链接默认使用 HTTP GET 方法。而我们想要用 POST, 所以这次我们要添加一个按钮。这意味着我们将用 `button_to` 方法。

可以通过指定 URL 来将按钮和在线商品联系起来, 但还是让 Rails 来处理这件事, Rails 可以简单地将 `_path` 追加到控制器的名称后, 即可具备这个能力。在本例中, 我们将用 `line_items_path`。

然而, 这样做会有一个问题: `line_items_path` 方法如何知道究竟哪个产品要添加到购物车中呢? 我们要将和这个按钮相对应的产品 id 也传给这个方法。这很容易实现, 所需做的就是调用 `line_items_path` 时添加一个 `:product_id` 参数。我们甚至可以在 `product` 实例中传递实例本身——在某些情况下 Rails 知道从记录中提取 id, 就像这次。

总之, 我们需要添加到 `index.html.erb` 中的那一行看起来就像:

```
depot_f/app/views/store/index.html.erb
```

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

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">
      <span class="price"><%= number_to_currency(product.price) %></span>
      <%= button_to 'Add to Cart', line_items_path(:product_id => product) %>
```

```

    </div>
  </div>
<% end %>

```

在上面这段代码中，又出现了一个格式问题。`button_to` 创建了一个 HTML `<form>`，且这个表单包含了一个 HTML `<div>`。这两者通常是块元素，并将出现在下一行中。我们想将这些内容放在价格之后，所以需要一点点 CSS 技巧使它们内联：

```
depot_f/public/stylesheets/depot.css
```

```

#store .entry form, #store .entry form div {
  display: inline;
}

```

现在我们的索引网页如图 9.1 所示。但在单击这个按钮之前，还需修改在线商品控制器的 `create` 方法，从而使产品 `id` 作为表单的参数。这里我们开始发现模型中的 `id` 字段是多么重要了。Rails 通过字段 `id` 来标识模型对象（以及对应的数据库记录）。如果将 `id` 传递给 `create`，那么就能唯一地标识要添加的产品了。

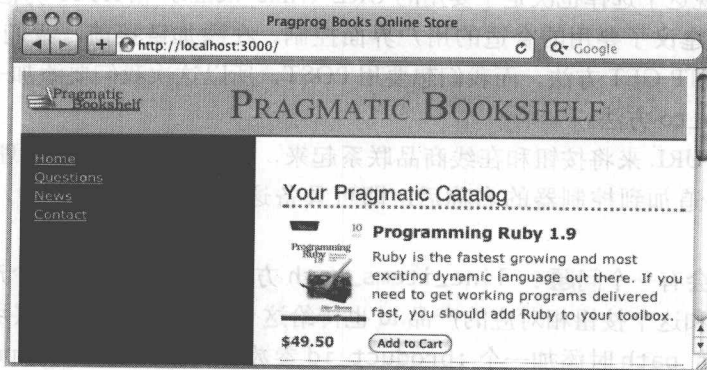


图 9.1 现在有了 Add to Cart 按钮

为什么是这个 `create` 方法呢？链接的默认 HTTP 方法是 `get`，按钮的默认 HTTP 方法是 `post`，Rails 用这些约定来确定要调用哪个方法。其他约定的介绍可参阅 `app/controllers/line_items_controller.rb` 文件中的注释。在 Depot 应用程序中会广泛应用到这些约定。

现在，让我们来修改 `LineItemsController` 以找到当前会话中的购物车（如果还没有，就创建一个），添加所选产品到购物车中，并显示购物车中的内容。我们需要做的只是在 `app/controllers/line_items_controller.rb` 的 `create` 方法中修改几行代码：[⊖]

```
depot_f/app/controllers/line_items_controller.rb
```

```

def create
  ▶ @cart = current_cart

```

⊖ 为了匹配这个页面，有几行代码已经强制换行了。

```

▶ product = Product.find(params[:product_id])
▶ @line_item = @cart.line_items.build(:product => product)

respond_to do |format|
  if @line_item.save
    ▶ format.html { redirect_to(@line_item.cart,
      :notice => 'Line item was successfully created.' ) }
    format.xml { render :xml => @line_item,
      :status => :created, :location => @line_item }
  else
    format.html { render :action => "new" }
    format.xml { render :xml => @line_item.errors,
      :status => :unprocessable_entity }
  end
end
end
end

```

我们使用了 9.1 节中实现的 `current_cart` 方法查找（或创建）会话中的购物车。接下来，我们用 `params` 对象从请求中得到 `:product_id` 参数。在 Rails 的应用程序中 `params` 对象是非常重要的。它保存了所有在浏览器请求中传递的参数。因为视图不需要访问这个结果，所以我们将该结果保存在一个本地变量里。

然后将找到的产品传递给 `@cart.line_items.build`。这会构造一个新的在线商品关系，即 `@cart` 对象和 `product` 之间的关系。你可以从任何一端来构造这个关系，Rails 会帮助建立双方的联系。

我们将这个在线商品的结果保存到一个名为 `@line_item` 的实例变量中。

这种方法的其余部分是用来处理 XML 请求的，这部分我们会在 25.1 节更详细地介绍，关于如何处理错误，则会在 10.2 节介绍。现在，我们只是想再修改一件事情：一旦创建了在线商品，我们希望将用户重定向到购物车，而不是回到在线商品网页。由于在线商品对象知道如何找到购物车对象，所有我们需要做的就是方法调用中添加 `.cart`。

当修改控制器功能时，我们知道需要更新相应的功能测试。在调用 `create` 方法时需要传递产品 id 给该方法，并将重定向的网址修改为我们想要的。通过修改 `test/functional/line_items_controller_test.rb` 可以实现这个目标。

```

depot_g/test/functional/line_items_controller_test.rb

test "should create line_item" do
  assert_difference('LineItem.count') do
    ▶ post :create, :product_id => products(:ruby).id
  end

  ▶ assert_redirected_to cart_path(assigns(:line_item).cart)
end

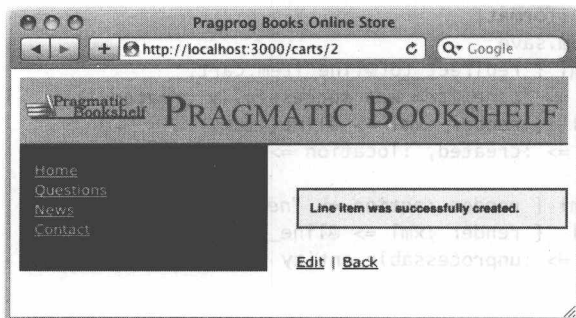
```

到目前为止，我们还没有谈到 `assigns` 方法，这是因为它已经在生成的脚手架里了。通过这个方法我们可以访问那些已经（或可能）由视图指派的变量。

现在，我们重新运行以下功能测试：

```
depot> rake test:functionals
```

我们相信，代码会如期待的那样工作，现在让我们尝试单击浏览器中的 **Add to Cart** 按钮。这就是所看到的：

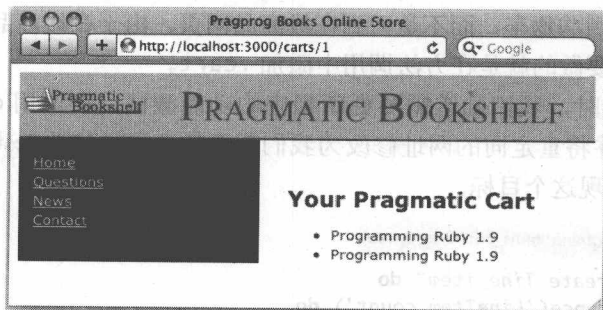


这个网页不会给人留下深刻印象。虽然给购物车准备了脚手架，但当创建购物车时，我们没有提供任何属性，所以这个视图中什么都没有显示。现在，让我们写个简单的模板（我们会在几分钟内搞定它）：

```
depot_1/app/views/carts/show.html.erb
```

```
<h2>Your Pragmatic Cart</h2>
<ul>
  <% @cart.line_items.each do |item| %>
    <li><%= item.product.title %></li>
  <% end %>
</ul>
```

好的，现在所有的东西都连接在一起了。让我们单击浏览器中的“刷新”按钮来看一下所显示的简单视图：



返回“主目录”页：<http://localhost:3000/>，并添加不同的产品到购物车。你会在购物车中看到原来的两个产品以及新选的产品。看起来会话工作正常，是时候给客户看我们的成果了，所以我们请他过来，骄傲地展示给他看这个漂亮的、新的购物车。可是他发出啧啧的声音，很明显没有给出他想要的东西，这可有点儿令我们失望。

他解释说，真实的购物车是不会将两个同样的产品显示在不同的行中的。相反，该产品行应该只显示一次，产品数量为2。看起来我们该做下一次迭代了。

9.4 本章小结

到目前为止，这是忙碌、富有成效的一天。我们给在线商店添加了购物车，在开发的过程中我们已经涉及了 Rails 中的某些功能：

- 我们在一个请求中创建了一个 `Cart` 对象，并能够在后续的请求中通过会话对象成功地找到该购物车。
- 我们在所有控制器的基类中添加了一个私有方法，从而使得所有的控制器都可以使用该方法。
- 我们创建了购物车和在线商品之间的关系以及在线商品和产品之间的关系，并且我们可以通过这些关系进行浏览。
- 我们添加了一个按钮，使得能添加产品到购物车中，同时还创建了一个新的在线商品。

练习时间

可以自己尝试以下任务：

- 修改应用程序，以便单击书的图像时也会激活 `create` 行为。提示：`link_to` 的第一个参数放置在生成的 `<a>` 标签中，Rails 的帮助函数 `image_tag` 构造了一个 HTML `` 标签。这里会调用一次这个帮助函数，该函数的参数即为 `link_to` 方法的第一个参数。在你调用 `link_to` 时，它的参数 `html_options` 里一定要包括：`method=>:post`。
- 在会话里添加一个新的变量，用来记录用户访问了商店控制器的 `index` 行为的次数。请注意，第一次页面被访问时，你的计数器还不在会话中。你可以用下面的代码来测试一下：

```
if session[:counter].nil?
```

```
...
```

如果会话变量不存在，你需要对其进行初始化。然后，你就可以对该变量进行累加计算了。

- 将这个计数器传给你的模板，并在目录页面的顶部显示该计数器。提示：在构造你要显示的消息时，帮助函数 `pluralize`（在 21.5 节中会介绍）可能会有用。
 - 每当用户添加东西到购物车时，将计数器重置为 0。
 - 更改模板，只有当计数器大于 5 时，才显示该计数器。
- （在 <http://pragprog.com/wikis/wiki/RailsPlayTime> 上你会找到相关的提示。）

第 10 章

任务 E：更智能的购物车

在本章中，我们将学习：

- 修改数据库模式 (schema) 与现有数据
- 诊断和处理错误
- 闪存
- 日志

虽然已经实现了购物车的基本功能，但还是有很多事情要做。首先，要识别何时用户将同一商品项目多次添加到购物车中。一旦做完之后，还必须确保购物车本身能够处理错误事件，并在问题发生时能以一种合适的方式与消费者或系统管理员沟通。

10.1 迭代 E1：创建更智能的购物车

由于购物车中的每个产品都有一个关联的计数器，这就要求修改 `line_items` 表。之前已经在 6.1 节中使用过迁移来修改数据库的模式了。虽然这只是建立模型的初始脚手架的一部分，但基本过程是相同的。

```
depot> rails generate migration add_quantity_to_line_items quantity:integer
```

Rails 可以从迁移名称 (如 `add_quantity_to_line_items`。——译者注) 中判断出正在加一个或多个数据字段到 `line_items` 表中，还能从最后一个参数 (如 `quantity:integer`。——译者注) 中获取每个字段名称和类型。Rails 有两种匹配模式，即 `add_XXX_to_TABLE` 和 `remove_XXX_from_TABLE`，这里 XXX 的值被忽略；重要的是出现在迁移名之后的字段名及其类型的清单。

Rails 无法判断的唯一事情就是，什么是该字段的合理默认值。在许多情况下，默认值会是 `null`，但让我们把已有的购物车的默认值设置为 1。为此，在应用迁移前，先做如下修改：

```
depot_g/db/migrate/20110211000004_add_quantity_to_line_items.rb
```

```
class AddQuantityToLineItems < ActiveRecord::Migration
  def self.up
    ▶ add_column :line_items, :quantity, :integer, :default => 1
  end
  def self.down
    remove_column :line_items, :quantity
  end
end
```

一旦修改完成，即可将该迁移应用到数据库中：

```
depot> rake db:migrate
```


现在 Cart 中需要一个聪明的 `add_product` 方法, 该方法用来判断商品清单中是否已包含了想要添加的产品; 如果是的话, 那就增加数量, 如果不是的话, 就生成个新的 `LineItem`:

```
depot_g/app/models/cart.rb
```

```
def add_product(product_id)
  current_item = line_items.find_by_product_id(product_id)
  if current_item
    current_item.quantity += 1
  else
    current_item = line_items.build(:product_id => product_id)
  end
  current_item
end
```

此代码使用了一个 Active Record 模块的小技巧。可以看到, 该方法第一行调用了 `find_by_product_id`, 但这个方法没有定义过。然而, Active Record 模块注意到调用未定义的方法, 并且发现其名称是以字符串 `find_by` 开始和字段名结束, 于是 Active Record 模块动态地构造了查询器方法 (finder method), 并将其添加到类中。在 19.3 节会详细介绍动态查询器。

为了使用这个方法, 还要修改商品项目控制器。

```
depot_g/app/controllers/line_items_controller.rb
```

```
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  ▶ @line_item = @cart.add_product(product.id)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to(@line_item.cart,
        :notice => 'Line item was successfully created.' ) }
      format.xml { render :xml => @line_item,
        :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @line_item.errors,
        :status => :unprocessable_entity }
    end
  end
end
end
```

为了使用新信息, 最后还需要快速地修改 show 视图:

```
depot_g/app/views/carts/show.html.erb
```

```
<h2>Your Pragmatic Cart</h2>
<ul>
  <% @cart.line_items.each do |item| %>
  ▶ <li><%= item.quantity %> &times; <%= item.product.title %></li>
  <% end %>
</ul>
```

现在所有东西都已到位, 可以回到商店网页, 单击已在购物车中的产品的 Add to Cart 按钮。

可能会看到的是:数量为 1 和数量为 2 的产品分开列出。这是因为我们增加了数量 1 到现有字段,来取代有可能铺开多行的显示方式。下一步要做的是数据迁移。

先创建一个迁移:

```
depot> rails generate migration combine_items_in_cart
```

这次, Rails 可推断不出想做什么了,所以,这次完全由我们来填写 `self.up` 方法:

```
depot_g/db/migrate/20110211000005_combine_items_in_cart.rb
```

```
def self.up
  # replace multiple items for a single product in a cart with a single item
  Cart.all.each do |cart|
    # count the number of each product in the cart
    sums = cart.line_items.group(:product_id).sum(:quantity)

    sums.each do |product_id, quantity|
      if quantity > 1
        # remove individual items
        cart.line_items.where(:product_id=>product_id).delete_all

        # replace with a single item
        cart.line_items.create(:product_id=>product_id, :quantity=>quantity)
      end
    end
  end
end
```

显然,这是到目前为止所看到的扩充代码最多的地方。接下来仔细地分析:

- 先从迭代每个购物车开始。(迭代见 4.3 节。)
- 对于每个购物车及其每个相关联的商品项目,按照字段 `product_id` 进行编组,得出各字段数量之和。计算结果将是字段 `product_ids` 和数量对的有序列表。

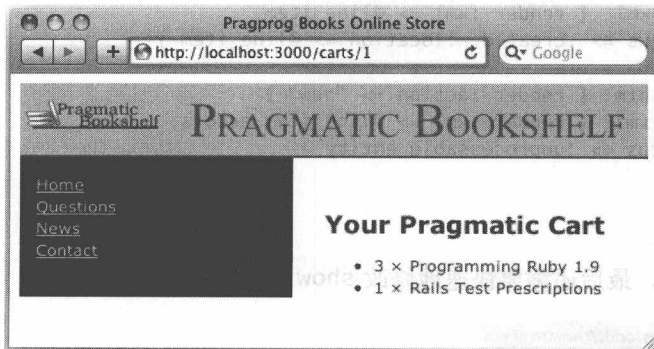


图 10.1 添加了商品数量的购物车

- 然后,迭代每一组之和,从每一个组中提取 `product` 和 `quantity`。
- 对于数量大于 1 的组,将删除与该购物车和该产品相关联的所有单个的商品项目,然后用正确数量的单行商品项目来替代它们。

注意, Rails 使我们可以如此简单和优雅地来表示这个算法。

代码到位了, 现在就像应用别的迁移一样来处理这个迁移:

```
depot> rake db:migrate
```

马上就可以通过查看购物车得到结果, 如图 10.1 所示。虽然我们已经有了可以高兴的理由, 但还没有做完。迁移的一个重要原则是, 每一步都要是可逆的, 所以, 还要实现了一个 `self.down` 方法。这种方法用于查找数量大于 1 的商品项目: 为该购物车和产品添加一个新的商品项目, 1 个数量增加 1 行, 最后删除该商品项目多余的行。该操作的代码如下:

```
depot_g/db/migrate/20110211000005_combine_items_in_cart.rb

def self.down
  # split items with quantity>1 into multiple items
  LineItem.where("quantity>1").each do |line_item|
    # add individual items
    line_item.quantity.times do
      LineItem.create :cart_id=>line_item.cart_id,
                     :product_id=>line_item.product_id, :quantity=>1
    end

    # remove original item
    line_item.destroy
  end
end
```

现在, 可以很容易地用以下单条命令来回滚迁移:

```
depot> rake db:rollback
```

还是可以立即通过查看购物车来验证结果, 如图 10.2 所示。一旦重新应用迁移 (用命令 `rake db:migrate`), 现在就有了可以保存产品计数器的购物车了, 此外还有个视图用来显示该计数器。

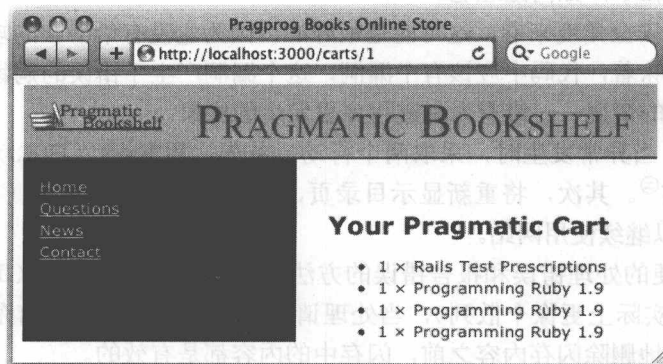


图 10.2 回滚迁移后的购物车

令人高兴的是, 有东西可以展示了, 让客户过来并告诉他今天上午的工作成果。他很高兴看到网站开始组合到一起了。然而, 还是有别的困扰, 他刚在贸易报刊文章中看到: 这种电子商

务网站每天都被攻击和损害。有种攻击是通过传递带错误参数的请求到 Web 应用程序。客户希望网站能暴露出缺陷和安全漏洞。他注意到,购物车的链接看起来像 `carts/nnn`, 其中 `nnn` 是内部的购物车 id。感觉这个不是很好,他直接在浏览器上输入了这个请求,并给这个购物车 id 传递 `wibble` 这个名字。当应用程序显示出了如图 10.3 所示的页面时,他不是很高兴,因为这个网页暴露了太多应用程序的信息。这看上去相当不专业。因此,下一次迭代将使应用程序具有更强的韧性。

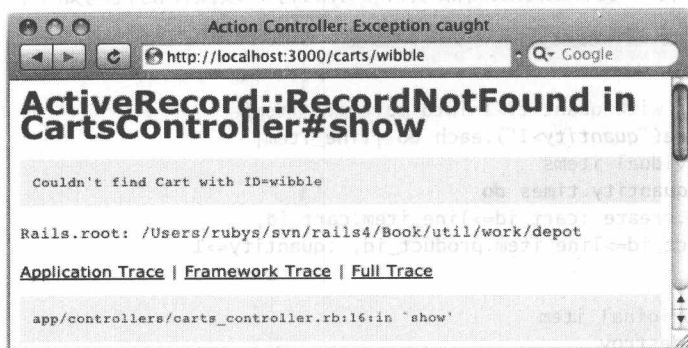


图 10.3 应用程序泄漏的内容

10.2 迭代 E2: 错误处理

看着如图 10.3 所示的这个网页,很明显应用程序在购物车控制器的第 16 行抛出了异常^①。也就是下面这一行:

```
@cart = Cart.find(params[:id])
```

如果无法找到购物车,Active Record 模块会抛出一个 `RecordNotFound` 的异常,显然需要处理这个异常。问题在于,如何处理它?

可以忽略它。从安全角度来看,这可能是最好的,因为它没有给潜在的攻击者提供所需要的信息。不过,这也意味着,代码中应该有个漏洞,这个漏洞产生了错误的购物车 id,应用程序会出现对外界没有响应的情况——没有人知道程序里发生的错误。

另一个方法是,当异常发生时,采取两个行动。首先,用 Rails “日志功能”在内部日志文件中记录发生的事实^②。其次,将重新显示目录页,并发个短消息给用户(比如“无效购物车”之类的),使他们可以继续使用网站。

Rails 提供了方便的处理错误和报告错误的方法。它定义了称为闪存(flash)的结构。闪存是一个桶(bucket,实际上更像个散列),当处理请求时,可以在其中存储东西。对于同一会话的下次请求,在自动地删除闪存内容之前,闪存中的内容都是有效的。

通常情况下闪存是用来收集错误信息的。例如,当 `show` 方法发现传递给它的参数是个无效

① 代码行数可能会不一样。在源代码中有些和本书相关的格式工具。

② http://guides.rubyonrails.org/debugging_rails_applications.html#the-logger

的购物车 id 时, 它可以将这个错误信息存储在闪存中, 并重定向到 index 行为, 以重新显示商品目录。index 行为的视图可以提取错误, 并将其显示在目录页的顶部。在视图中的 flash (闪存) 存取器方法 (accessor method) 来访问闪存的信息。

为什么不能将错误存放在过时的实例变量中呢? 记得应用程序发送重定向到浏览器后, 浏览器返回一个新的请求给应用程序。在收到请求的这段时间内, 应用程序已经继续向前运行了——以前请求中的所有实例变量都早已不复存在。闪存数据存储与会话中, 以使其能在请求与请求的中间被访问。

有了闪存数据的相关背景知识以后, 现在可以修改 show 方法来拦截那些无效的产品 id 并报告问题:

```
depot_h/app/controllers/carts_controller.rb

# GET /carts/1
# GET /carts/1.xml
def show
  ▶ @cart = Cart.find(params[:id])
  ▶ rescue ActiveRecord::RecordNotFound
  ▶   logger.error "Attempt to access invalid cart #{params[:id]}"
  ▶   redirect_to store_url, :notice => 'Invalid cart'
  ▶ else
  ▶   respond_to do |format|
  ▶     format.html # show.html.erb
  ▶     format.xml { render :xml => @cart }
  ▶   end
  ▶ end
end
```

Rescue 子句拦截了 Cart.find 抛出的异常。在该句柄中, 做到以下几点:

- 使用 Rails 日志器记录错误。每个控制器有一个 logger 属性。在这里, 用它来记录 error (错误) 日志级别的消息。
- 用 redirect_to 方法来重定向到目录网页。:notice 参数指定存储在闪存中的通知消息。在这里为什么要重定向, 而不是仅仅显示目录呢? 如果是重定向, 用户的浏览器将终止显示商店的网址, 不再显示 <http://.../cart/wibble>。这样可以更少地暴露应用程序的信息。还可防止用户通过单击刷新按钮再次引发这个错误。

有了这段代码后, 可以重新运行那个会引起问题的客户查询。输入以下网址:

<http://localhost:3000/carts/wibble>

在浏览器中没看到一堆错误信息。相反, 看到的是目录网页。如果看下日志文件结尾的部分 (目录 log 中的 development.log), 会看到这样的消息:

```
Processing CartsController#show to */*
sql (0.0ms)  SELECT * FROM "products" WHERE (("products"."id" = 2))
Completed in 38ms (View: 28, DB: 0) | 200 [unknown]
Parameters: {"id"=>"1"}
sql (0.0ms)  SELECT * FROM "carts" WHERE (("carts"."id" = 0))
```

```
ActiveRecord::RecordNotFound (Couldn't find Cart with ID=wibble):
```

```
app/controllers/carts_controller.rb:22:in 'show'
```

```
: :
```

► Attempt to access invalid cart wibble

```
sql (0.0ms) SELECT * FROM "carts" WHERE (("carts"."id" = 0))
```

```
Redirected to http://127.0.0.1:3000/store
```

```
: :
```

```
Rendering app/views/store/index.html.erb
```

```
Rendering template within app/views/layouts/application.html.erb
```

这是更加用户友好的结果，如图 10.4 所示。

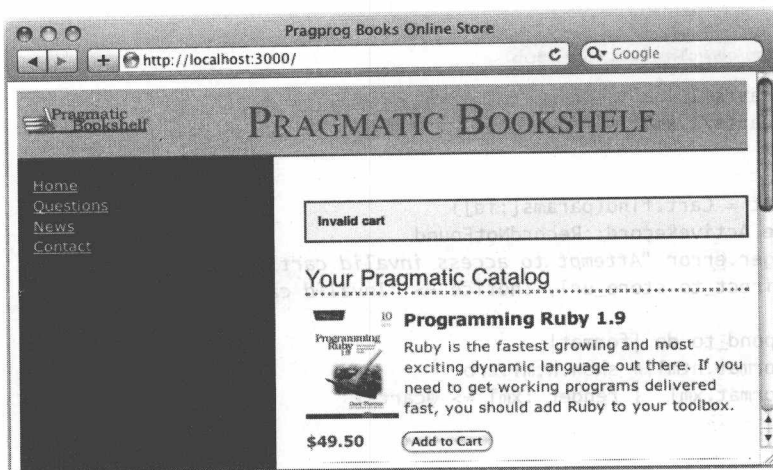


图 10.4 更面向用户的错误信息

在 UNIX 机器上，可能用命令 `tail` 或 `less` 来查看这个文件。在 Windows 上，可以用你喜欢的编辑器。让窗口保持打开状态，以便显示那些刚加到这个文件中的新文本行，通常这是个好注意。UNIX 中则可以使用命令 `tail -f` 达到此效果。可以从 <http://gnuwin32.sourceforge.net/packages/coreutils.htm> 下载给 Windows 用的 `tail` 命令或者从 <http://tailforwin32.sourceforge.net/> 得到一个带 GUI 的工具。最后，一些 OS X 的用户可以用 `Console.app` 来跟踪日志文件。只要在命令行输入 `open name.log`。

感到该结束这次迭代的时候，招呼客户过来看一下，现在已经妥善地处理了错误。他很高兴并继续浏览该应用程序。他发现新购物车显示页面有个小问题——有没有办法清空购物车。这种微小的修改将是下一次迭代的任务。要在回家之前搞定它。

10.3 迭代 E3：对购物车的最后加工

现在知道，为了实现“清空购物车”功能，必须要在购物车中添加个链接和修改购物车控制器中的 `destroy` 方法来清理会话。先从模板开始，并再次用 `button_to` 方法给页面添加个按钮：


```
depot_h/app/views/carts/show.html.erb
```

```
<h2>Your Pragmatic Cart</h2>
```

```
<ul>
```

```
<% @cart.line_items.each do |item| %>
```

```
<li><%= item.quantity %> &times; <%= item.product.title %></li>
```

```
<% end %>
```

```
</ul>
```

```
▶ <%= button_to 'Empty cart', @cart, :method => :delete,
```

```
▶ :confirm => 'Are you sure?' %>
```

在控制器中修改 `destroy` 方法, 以确保用户只是删除他自己的购物车 (想想吧), 并在重定向到索引页面之前 (带有通知消息), 从会话中删除该购物车:

```
depot_h/app/controllers/carts_controller.rb
```

```
def destroy
```

```
▶ @cart = current_cart
```

```
@cart.destroy
```

```
▶ session[:cart_id] = nil
```

```
respond_to do |format|
```

```
▶ format.html { redirect_to(store_url,
```

```
▶ :notice => 'Your cart is currently empty') }
```

```
format.xml { head :ok }
```

```
end
```

```
end
```

然后更新 `test/functional/carts_controller_test.rb` 文件中对应的测试。

```
depot_h/test/functional/carts_controller_test.rb
```

```
test "should destroy cart" do
```

```
assert_difference('Cart.count', -1) do
```

```
▶ session[:cart_id] = @cart.id
```

```
delete :destroy, :id => @cart.to_param
```

```
end
```

```
▶ assert_redirected_to store_path
```

```
end
```

David 说

路由的争斗: `product_path` 与 `product_url`

当链接或重定向到一个给定的路由时, 在一开始似乎很难知道何时用 `product_path`, 何时用 `product_url`。实际上, 它真的很简单。

当用 `product_url` 时, 会得到包括协议和域名的完整信息, 如 `http://example.com/products/1`。这就是在用 `redirect_to` 时所要的, 因为在做 302 重定向时, HTTP 规范要求完全合格的 URL。如果从一个域到另一个域的重定向, 那么也需要完整的 URL, 就像这样的格式:

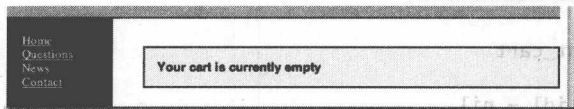
```
product_url(:domain => "example2.com", :product => product)
```

在其他的情况下,使用 `product_path` 无妨。它仅生成 `/products/1` 部分,当想要链接或转向表单时,这部分就是全部所需的,就像这样:

```
link_to "My lovely product", product_path(product)
```

令人困惑的地方是:一些要求不严谨的浏览器常常互换这两个。可以构造一个使用 `product_path` 的 `redirect_to`,并且它也能工作,但根据规范它是无效的。可以用 `link_to product_url`,那么 HTML 上就会充斥着不必要的字符,这可是个坏主意。

现在看看购物车并单击 Empty Cart 按钮,网页自动回到了目录页面,同时还有个漂亮的小消息如下:



当添加新的商品项目时,也可以删除那个自动生成的闪存消息:

```
depot_1/app/controllers/line_items_controller.rb
```

```
def create
```

```
  @cart = current_cart
```

```
  product = Product.find(params[:product_id])
```

```
  @line_item = @cart.add_product(product.id)
```

```
  respond_to do |format|
```

```
    if @line_item.save
```

```
      format.html { redirect_to(@line_item.cart) }
```

```
      format.xml { render :xml => @line_item,
```

```
        :status => :created, :location => @line_item }
```

```
    else
```

```
      format.html { render :action => "new" }
```

```
      format.xml { render :xml => @line_item.errors,
```

```
        :status => :unprocessable_entity }
```

```
    end
```

```
  end
```

```
end
```

最后,抽出点时间来整理购物车的显示页面。用表格来整理该页面,而不是给每个商品都加上 `` 元素。同样,用 CSS 制作样式:

```
depot_1/app/views/carts/show.html.erb
```

```
<div class="cart_title">Your Cart</div>
```

```
<table>
```

```
<% @cart.line_items.each do |item| %>
```

```
<tr>
```

```
<td>%= item.quantity %>&times;</td>
```

```
<td>%= item.product.title %></td>
```

```

      <td class="item_price"><%= number_to_currency(item.total_price) %></td>
    </tr>
  <% end %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to 'Empty cart', @cart, :method => :delete,
      :confirm => 'Are you sure?' %>

```

要让这个代码运行起来，要分别在 `LineItem` 和 `Cart` 模型中添加个方法，`LineItem` 模型中的新方法返回每种商品项目的总价，`Cart` 模型的新方法返回整个购物车中所有商品的总价。首先是 `LineItem` 中的方法，它只是用到了简单的乘法：

```

depot_i/app/models/line_item.rb

def total_price
  product.price * quantity
end

```

其次，`Cart` 模型中的方法则用到了 Rails 的漂亮的 `Array::sum` 方法来计算这个集合中每个元素的价格总和：

```

depot_i/app/models/cart.rb

def total_price
  line_items.to_a.sum { |item| item.total_price }
end

```

然后在 `depot.css` 样式表中再加点东西：

```

depot_i/public/stylesheets/depot.css

/* Styles for the cart in the main page */
#store .cart_title {
  font: 120% bold;
}

#store .item_price, #store .total_line {
  text-align: right;
}

#store .total_line .total_cell {
  font-weight: bold;
  border-top: 1px solid #595;
}

```

现在购物车更漂亮了，如图 10.5 所示。

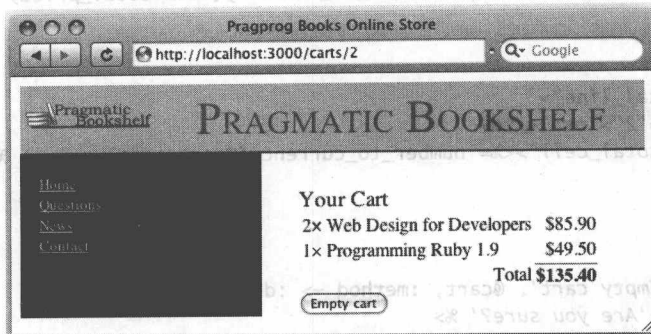


图 10.5 显示购物车中的总价

10.4 本章小结

现在客户非常满意购物车。在本章中介绍了以下内容：

- 在现有的数据库表中添加带有默认值的字段。
- 现有的数据迁移到新的数据库表格式。
- 为检测到的错误提供闪存通知。
- 使用日志器来记录事件。
- 删除记录。
- 使用 CSS 调整呈现页面表格的方式。

但正当考虑要把这个功能打包时，客户阅览《Information Technology and Golf Weekly》杂志。显然她看到了一篇关于浏览器界面新风格的文章，即动态地更新浏览器内容。“Ajax”，她说，“这个不错。明天再来看看这个吧。”

练习时间

可以自己尝试以下任务：

- 创建一个迁移，该迁移将产品的价格复制给商品项目，并修改 Cart 模型中 `add_product` 方法，每当创建一个新的商品项目时，该方法都可获得其价格。
- 添加单元测试：当增加唯一产品和重复产品时。请注意，你将需要修改静态测试以达到通过名称来引用产品和购物车，例如 `product: ruby`。
- 检查其他地方的商品和商品项目，在这些地方，用户友好的错误消息是按顺序的。
- 添加这一功能：从购物车中删除单个商品项目。这就要求每一行上都要添加一个按钮，这些按钮将需要连接 `LineItemsController` 的 `destroy` 行为。（可以在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 找到相关提示。）

第 11 章

任务 F: Ajax 初体验

在本章中，我们将学习：

- 使用局部模板
- 呈现页面布局
- 利用 Ajax 和 RJS 动态更新页面
- 利用 Script.aculo.us 高亮变化
- 隐藏和显示 DOM 元素
- 测试 Ajax 更新

客户想要我们为商店添加 Ajax 支持，但什么是 Ajax？

在以前（大约 2005 年以前），浏览器被视为很笨拙的设备。当你在写一个基于浏览器的应用程序时，你可能会把东西发给浏览器，然后不再使用那个会话。某些时刻，当用户填写某些表格项，或是单击了一个超链接时，应用程序会被收到的请求唤醒。浏览器会呈现整个页面给用户，然后这整个无趣的过程又重新开始。目前为止这恰恰是我们的 Depot 应用程序所做的。

但结果是浏览器还没有那么蠢（谁知道呢），它还可以运行代码。几乎所有的浏览器都可以运行 JavaScript。结果是，浏览器中的 JavaScript 能在幕后与服务器上的应用程序交互，并更新用户的所见。Jesse James Garrett 把这种交互的方式称为 Ajax（即异步 JavaScript 和 XML，或直白地说就是让浏览器没那么蠢）。

那么，让我们来为购物车添加 Ajax 吧。让我们把当前整个购物车的显示放到商品目录的侧边栏中，而不是一个独立的页面。然后，我们会利用 Ajax 来更新这个侧边栏，而不用重新显示整个页面。无论你何时使用 Ajax，以下的做法都是值得提倡的，即从一个非 Ajax 的应用程序版本开始，然后循序渐进地添加 Ajax 特性。这正是我们打算做的。第一步，让我们把购物车从独立的页面移到侧边栏中。

11.1 迭代 F1：转移购物车

当前，我们通过 CartController 中的 show 动作和相应的 .html.erb 模板来呈现购物车。我们想要做的是把这个呈现移到侧边栏中去。这意味着要把购物车呈现在一个显示所有商品目录的布局中，而不是在一个独立的页面里。利用局部模板，这很容易做到。

11.1.1 局部模板

程序语言让你可以定义方法。一个方法是一段命名的代码：用该名字调用这个方法会让这段代码得以运行。当然，你可以给方法传递参数，这可以让方法在不同的场合使用。

你可以把 Rails 的局部模板 (Rails partial template, 英文简称 partial) 当成是一种视图的方法。简单来说, 局部模板就是一个在单独文件中的视图。你可以利用模板或控制器来调用 (或者说呈现) 局部模板, 它会显示自己并返回显示的结果。正如方法一样, 你可以传递参数给局部模板, 这样使用相同的局部模板可以呈现出不同的结果。

在这个迭代周期中, 我们会两次用到局部模板。首先, 让我们来看看购物车的显示:

```
depot_j/app/views/carts/show.html.erb
```

```
<div class="cart_title">Your Cart</div>
<table>
  <% @cart.line_items.each do |item| %>
    <tr>
      <td><%= item.quantity %>&times;</td>
      <td><%= item.product.title %></td>
      <td class="item_price"><%= number_to_currency(item.total_price) %></td>
    </tr>
  <% end %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to 'Empty cart', @cart, :method => :delete,
  :confirm => 'Are you sure?' %>
```

这段代码生成了一列表格行, 其中每个都对应着购物车中的一个项目。每当你看到像这样的循环的时候, 都应该停下来问自己, 在模板中这样做是否太逻辑化了? 答案是, 我们可以利用局部模板来抽象化这些循环 (我们也会看到, 这样做还为接下来施展 Ajax 魔法做好了铺垫)。要这样做, 我们得利用到一点, 即你可以向显示局部模板的方法传递一个集合, 这个方法会自动为集合中的每一项调用一次局部模板。利用这个特点, 让我们重写一下购物车的显示:

```
depot_j/app/views/carts/show.html.erb
```

```
<div class="cart_title">Your Cart</div>
<table>
  ▶ <%= render(@cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to 'Empty cart', @cart, :method => :delete,
  :confirm => 'Are you sure?' %>
```

这样一来简单多了。render 方法会循环遍历任何一个传递给它的集合。局部模板本身不过

是另一个模板文件（该文件在默认情况下与当前呈现的对象在同一文件夹下，且与表格的名字相同）。但是，为了与普通的模板区别开，Rails 会自动在局部模板的名字前面加一个下划线。也就是说，得把局部模板命名为 `_line_item.html.erb`，且放在 `app/views/line_items` 文件夹下。

```
depot_j/app/views/line_items/_line_item.html.erb
```

```
<tr>
  <td><%= line_item.quantity %><times></td>
  <td><%= line_item.product.title %></td>
  <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
</tr>
```

这里需要注意一些细节。在局部模板中，当前对象的变量名称需与模板的名称相一致。在这里，局部模板的名字是 `line_item`，所以在其中也得有个变量称为 `line_item`。

现在就整理好了购物车的显示，但还没有移到侧边栏中去。要做到这一点，我们要重新审视一下布局。如果有一个局部模板能显示购物车，我们可以简单地在侧边栏中嵌入如下调用：

```
render("cart")
```

但局部模板怎么知道去哪里找购物车对象呢？一种方法是做一个假设。在布局中，我们可以访问控制器设置好的 `@cart` 实例变量。这样做的结果是，对于被布局调用的局部模板，该实例变量也是可用的。但是，这样做相当于调用一个方法并传给它某个全局变量。这样做虽然行得通，但却是代码丑陋，且增加了耦合（让程序不健壮且难以维护）。

现在对于一个商品项目有了一个局部模板，让我们对购物车做相同的事情。首先，建立 `_cart.html.erb` 模板。这基本上就是 `carts/show.html.erb` 模板，只不过使用 `cart` 而不是 `@cart`。（请注意，一个局部模板可以调用其他的局部模板。）

```
depot_j/app/views/carts/_cart.html.erb
```

```
<div class="cart_title">Your Cart</div>
<table>
  ▶ <%= render(cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    ▶ <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>
</table>

▶ <%= button_to 'Empty cart', cart, :method => :delete,
  :confirm => 'Are you sure?' %>
```

Rails 的宗旨是不要重复你自己（Don't Repeat Yourself），我们刚刚却违反了这一条规则。当前两个文件是同步的，所以不会有什么问题，但同时有一套逻辑控制 Ajax 调用，和另一套逻辑对付 JavaScript 被禁用的情况，会引入一些问题。为了避免这些问题，可以把原模板替换成如下呈现局部模板的代码：

```
depot_k/app/views/carts/show.html.erb
```

```
▶ <%= render @cart %>
```

现在改变一下应用程序的布局,把这个新的局部模板加入侧边栏中:

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

```
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "scaffold" %>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
      ▶ <div id="cart">
      ▶ <%= render @cart %>
      ▶ </div>
      <a href="http://www....">Home</a><br />
      <a href="http://www..../faq">Questions</a><br />
      <a href="http://www..../news">News</a><br />
      <a href="http://www..../contact">Contact</a><br />
    </div>
    <div id="main">
      <%= yield %>
    </div>
  </div>
</body>
</html>
```

现在我们将稍加修改一下商店的控制器。因为我们在调用布局的同时查看商店的索引动作,而当前这个动作并没有设置 @cart。这点非常容易修改:

```
depot_k/app/controllers/store_controller.rb
```

```
def index
  @products = Product.all
  ▶ @cart = current_cart
end
```

再加一点 CSS:

```
depot_/public/stylesheets/depot.css
```

```
/* Styles for the cart in the sidebar */
```

```
#cart, #cart table {
  font-size: smaller;
  color: white;
}
```

```
#cart table {
  border-top: 1px dotted #595;
  border-bottom: 1px dotted #595;
  margin-bottom: 10px;
}
```

如果你向购物车添加了一些东西后显示目录，应该看到如图 11.1 所示页面。让我们等待威比奖^①的提名吧！

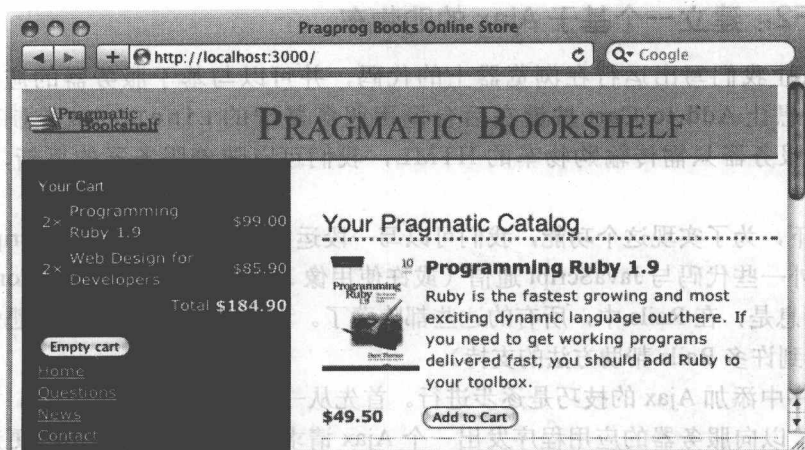


图 11.1 侧边栏中的购物车

11.1.2 改变流程

现在购物车已经显示在侧边栏中了，我们接着改变 Add to Cart 按键的工作方式。按下该键只需要刷新主索引页面，而不是显示一个单独的购物车页面。

这个改变很简单：在 create 动作的最后把浏览器重定向到索引：

```
depot_k/app/controllers/line_items_controller.rb
```

```
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.add_product(product.id)
```

① 威比奖 (Webby Awards) 是由国际数字艺术与科学学院主办的评选全球最佳网站的奖项。——译者注

```

respond_to do |format|
  if @line_item.save
    format.html { redirect_to(store_url) }
    format.xml { render :xml => @line_item,
      :status => :created, :location => @line_item }
  else
    format.html { render :action => "new" }
    format.xml { render :xml => @line_item.errors,
      :status => :unprocessable_entity }
  end
end
end
end

```

现在，我们商店的侧边栏中显示了购物车。当我们单击向购物车中增加项目的时候，页面会显示为更新后的购物车。但是，如果我们的商品目录很大的话，这个刷新可能要等待一段时间。这个过程会消耗带宽和服务器的资源。幸运的是，我们可以用 Ajax 来改进它。

11.2 迭代 F2：建立一个基于 Ajax 的购物车

Ajax 可以让我们写出运行在浏览器上的代码，并可以与基于服务器的应用程序互动。在这里，我们想让 Add to Cart 按键在后台调用服务器端的 LineItems 控制器的 create 动作。这样，服务器只需传输购物车的 HTML，我们可以随着服务器的更新，替换侧边栏中的购物车。

通常情况下，为了实现这个功能，我们可以写一段运行在浏览器中的 JavaScript 代码，并在服务器端写另外一些代码与 JavaScript 通信（或许使用像 JavaScript Object Notation [JSON] 一样的技术）。好消息是，在 Rails 中，所有的这些都隐藏了。利用 Ruby 我们可以做想做的任何事情（并且还可以得到许多 Rails 帮助方法的支持）。

向应用程序中添加 Ajax 的技巧是逐步进行。首先从一些最基础的步骤开始。让我们更改一下目录的页面，以向服务器的应用程序发出一个 Ajax 请求，并让程序回复一段更新过的购物车的 HTML 代码片段。

在索引页，我们正使用 button_to 来建立 create 动作的链接。我们想把这个改成发送 Ajax 请求。为了做到这一点，我们只需为调用增加一个 :remote => true 参数。

```

depot/_app/views/store/index.html.erb
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">

```

```

    <span class="price"><%= number_to_currency(product.price) %></span>
    <%= button_to 'Add to Cart', line_items_path(:product_id => product),
    :remote => true %>
  </div>
<% end %>

```

到现在为止，我们已经让浏览器可以向应用程序发送 Ajax 请求。下一步是让应用程序回复该请求。我们的计划是创建一个更新过的 HTML 片段用以表示购物车，并且让浏览器将此 HTML 插入到其显示文档的结构和内容的内部表示中去，即文档对象模型 (Document Object Model, DOM)。通过操作 DOM，我们可以改变浏览器的显示。

第一个要更改的是，如果请求的是 JavaScript，让 create 动作不要重定向到索引的显示上。这点可以通过以下方法实现：向 respond_to 增加一个调用，来告诉它我们希望得到一个 .js 格式的回复。

这个语法也许初看之下有点儿让人惊讶，但只不过是一个传递可选块参数的方法调用。块的介绍在 4.3 节。我们还会在 20.1 节中详细介绍 respond_to 方法。

```

depot_1/app/controllers/line_items_controller.rb
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.add_product(product.id)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to(store_url) }
      format.js
      format.xml { render :xml => @line_item,
        :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @line_item.errors,
        :status => :unprocessable_entity }
    end
  end
end
end
end

```

因为这个改变，当 create 完成了 Ajax 请求的处理，Rails 会寻找一个 create 模板来显示。

Rails 支持 RJS 模板——JS 代表 JavaScript。一个 .js.rjs 模板是让 JavaScript 在浏览器上做你想要做的事的一种方法，这全部是由服务器端的 Ruby 代码来实现。接下来编写我们的第一个：create.js.rjs。它放在 app/views/line_items 文件夹下，就像其他的商品项目的视图一样：

```

depot_1/app/views/line_items/create.js.rjs
page.replace_html('cart', render(@cart))

```

让我们来分析一下这个模板。变量 page 是一个 JavaScript 生成器的实例，JavaScript 生成器是一个 Rails 类，它知道如何在服务器上创建 JavaScript，并在浏览器上执行。这里，我们告诉

它将当前页面上的 id 为 cart 的元素内容，替换成给定购物车的局部模板。然后，这个简单的 RJS 模板就告诉浏览器用那段 HTML 替换 id="cart" 的元素的内容。

这样做行得通吗？虽然很难在书中显示，但答案是行得通。确保你重新加载了索引页面，以便将远程版本的表单和 JavaScript 库载入浏览器中。然后，单击 Add to Cart 按钮中的一个。你应该可以在侧边栏中看见更新的购物车。同时不应该看见浏览器有任何重载页面的征兆。这样，你就创建了一个 Ajax 应用程序。

11.2.1 排疑解难

尽管 Rails 让 Ajax 变得十分简单，但是却无法确保万无一失。另外，因为在这里你面对的是一些松散整合在一起的技术，若是 Ajax 不能正确工作，调试可能会比较困难。这也是应该总是一步一步来添加 Ajax 功能的原因之一。

如果 Ajax 的魔法无法在你的 Depot 应用程序中起作用，以下是线索：

- 浏览器是否有任何特别的功能强制刷新页面？有时浏览器会在本地缓存页面，这会把测试弄得混乱。那样的话就应该全部重载页面了。
- 是否有错误报告？查看一下 logs 目录下的 development.log 文件。也要查看一下 Rails 服务器的窗口，因为有些错误会显示在那里。
- 继续查看日志文件的内容，你能看见进来的动作 create 的请求吗？如果不能，意味着你的浏览器没有发出 Ajax 请求。如果 JavaScript 库已经加载了（利用浏览器上的“显示→源代码”，可以查看 HTML），那么或许就是你的浏览器把执行 JavaScript 给禁用了。
- 有些读者报告，他们得关掉并重启应用程序才能让基于 Ajax 的购物车运行。
- 如果你用的是 Internet Explorer 的话，原因可能就是浏览器运行在微软所谓的怪异模式（quirks mode），也就是为了向后兼容旧的 Internet Explorer 版本，但同样是坏的。如果下载下来的页面的第一行是一个正确的 DOCTYPE 头，那么 Internet Explorer 就会转到标准模式下，在该模式下，Ajax 可以得到更好的支持。为了实现这一点，我们在布局里使用如下代码：

```
<!DOCTYPE html>
```

11.2.2 客户永远不会满足

我们对自己感到非常满意。我们更改了许多行代码，让原本乏味的 Web 1.0 应用程序升级到了 Web 2.0 Ajax。我们把客户叫了过来，屏息以待。事先并没有告诉他任何事情，当我们骄傲地按下了 Add to Cart 按钮，并期待着对方的褒奖的时候，他却看起来非常惊讶，“你叫我来就是向我展示一个 bug？”她问道，“点了那个按钮，却什么也没发生。”

我们不得不耐心解释，其实，发生了很多事情。看看侧边栏吧，看到了吗？当我们加入一件东西的时候，数量从 4 变成了 5。

“哦，”她说道，“我没注意到这一点。”如果他没有注意到页面更新了，那么很有可能我们的顾客也不会注意到。是时候修改一下用户交互了。

11.3 迭代 F3: 高亮变化

Rails 中包含了一些 JavaScript 库。其中, `effects.js` 让你可以用一些视觉上有趣的效果来装点页面[⊖]。这其中的一个就是(现在)声名狼藉的 Yellow Fade 技术。该技术可以高亮显示浏览器中的某个元素: 默认情况下它让背景变成黄色, 然后逐渐褪成白色。将该技术应用到我们的购物车上, 可以看到首页变成如图 11.2 所示的样子; 在最后面的图片展示的是原始的购物车。用户单击 Add to Cart 按钮, 计数器更新到 2, 与此同时该行闪烁变亮。然后经过一段时间逐渐褪还成背景颜色。

现在将这种高亮的效果加入到购物车中。无论何时购物车中的项目更新了(当项目增加了, 或是当我们改变了数量), 让我们使背景变化一下。这个效果可以提醒我们的用户有些东西改变了, 尽管整个页面并没有更新。

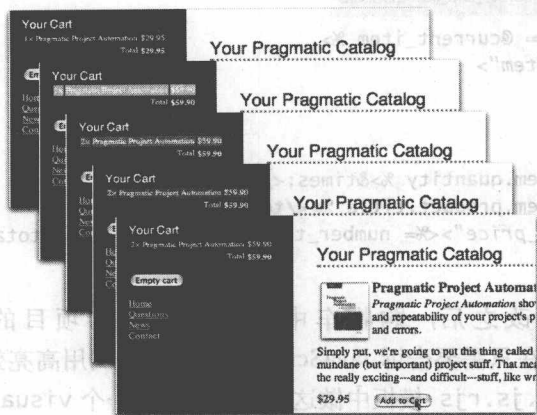


图 11.2 YELLOW FADE 技术下的购物车

第一个问题是如何在购物车中找到最近更新的项目。当前, 每个项目不过是一个 `<tr>` 元素。我们得找到一种办法来标记最近改变过的项目。让我们从 `LineItemsController` 开始着手。将当前的商品项目以赋值给一个实例变量的形式传给模板。

```
depot_m/app/controllers/line_items_controller.rb
```

```
def create
```

```
  @cart = current_cart
```

```
  product = Product.find(params[:product_id])
```

```
  @line_item = @cart.add_product(product.id)
```

```
  respond_to do |format|
```

```
    if @line_item.save
```

⊖ `effect.js` 是 `Script.aculo.us` 库的一部分。你可以通过查看 <https://github.com/madrobby/scriptaculous/wikis> 上的视觉效果来获悉都可以用它们来做什么。

```

format.html { redirect_to(store_url) }
▶ format.js { @current_item = @line_item }
format.xml { render :xml => @line_item,
:status => :created, :location => @line_item }
else
format.html { render :action => "new" }
format.xml { render :xml => @line_item.errors,
:status => :unprocessable_entity }
end
end
end

```

然后在 `_line_item.html.erb` 局部模板中，我们检查当前正在呈现的项目是否刚刚更改过。如果是，用一个 `current_item` 的 id 来标记它：

```

depot_m/app/views/line_items/_line_item.html.erb
▶ <% if line_item == @current_item %>
▶ <tr id="current_item">
▶ <% else %>
▶ <tr>
▶ <% end %>
    <td><%= line_item.quantity %>&times;</td>
    <td><%= line_item.product.title %></td>
    <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
  </tr>

```

经过这两个小的更改之后，购物车中最近更改过的项目的 `<tr>` 元素会被标记上 `id="current_item"`。现在只需要告诉 JavaScript 在该项目上调用高亮效果。

我们在已有的 `create.js.rjs` 模板中做这件事，即增加一个 `visual_effect` 方法的调用：

```

depot_m/app/views/line_items/create.js.rjs
▶ page.replace_html('cart', render(@cart))
▶ page[:current_item].visual_effect :highlight,
▶ :startcolor => "#88ff88",
▶ :endcolor => "#114411"

```

看到了如何通过向页面传递 `:current_item` 来找到想要施加效果的浏览器元素了吗？然后再利用 `highlight` 视觉效果，将默认的黄 / 白过渡替换成符合我们设计的颜色。单击向购物车中增加一个项目，你会看到改变的项目变成淡绿色，然后渐渐地褪变成背景色。

11.4 迭代 F4：隐藏一个空的购物车

顾客还有最后的一个要求。现在，即使是空的购物车仍然会显示在侧边栏中，是否可以仅当购物车中有内容的时候才显示它？当然可以！

事实上，有好几个选择。最简单的可能就是仅当购物车中有东西的时候才包含购物车的 HTML。我们可以完全在 `_cart` 局部模板中实现：

```

▶ <% unless cart.line_items.empty? %>
<div class="cart_title">Your Cart</div>
<table>
  <%= render(cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>
</table>

<%= button_to 'Empty cart', cart, :method => :delete,
      :confirm => 'Are you sure?' %>
▶ <% end %>

```

尽管这样做行得通，但是对于用户接口来说，这却有点儿野蛮：在购物车中商品从无到有的转变过程中，整个侧边栏都会重新显示。因此，最好不要使用这个代码。让我们来把这个过程平滑过渡一下。

效果库 Script.aculo.us 中包含了很多很好的过渡显示元素的效果。我们选择使用 **blind_down**，它可以将侧边栏中剩下的东西往下移动以腾出空间，来平滑地显示购物车。

毫不奇怪，我们将会用已有的 **.js.rjs** 模板来调用这个效果。因为当且仅当我们向购物车中添加了东西的时候，**create** 模板才会被调用，并且我们知道，无论何时购物车中有且只有一个项目的时候，我们就要在侧边栏中显示这个购物车（因为这意味着之前购物车是空的，因而被隐藏了）。并且，由于购物车必须在我们启动高亮效果之前才是可见的，所以得把显示购物车的代码加在触发高亮之前。

现在模板如下所示：

```

depot_n/app/views/line_items/create.js.rjs
page.replace_html('cart', render(@cart))

▶ page[:cart].visual_effect :blind_down if @cart.total_items == 1
▶ page[:current_item].visual_effect :highlight,
  :startcolor => "#88ff88",
  :endcolor => "#114411"

```

这还不能正确工作，因为在购物车模型中还没有 **total_items** 方法：

```

depot_n/app/models/cart.rb

def total_items
  line_items.sum(:quantity)
end

```

我们得让购物车是空的时候隐藏它。有两种基本的方法可以做到这一点。一种方法已经在本节开始的代码中演示过了，即不生成任何 HTML 代码。遗憾的是，如果这样做，当我们向购物车中添加东西并且突然创建购物车的 HTML 代码时，会在浏览器中看到一闪，因为先是显示购物车，然后隐藏并且缓慢地通过 **blind_down** 效果显示出来。

一个更好的方法是创建购物车的 HTML 代码，如果购物车为空，则将 CSS 风格设为 `display: none`。为了实现这一点，需要改变 `app/views/layouts` 中 `application.html.erb` 的布局。我们的第一次尝试如下所示：

```
<div id="cart"
  <% if @cart.line_items.empty? %>
    style="display: none"
  <% end %>
>
<%= render(@cart) %>
</div>
```

当且仅当购物车为空时，这段代码才将 CSS `style=` 属性加入 `<div>` 标签中。这样做虽然行得通，却非常丑陋。悬空的符号 `>` 像是放错了地方（虽然并没有），并且逻辑以这种方式插入到标签中就像是给模板语言起一个糟糕的名字。不要让这种丑陋现象污染我们的代码，取而代之的是创建一个抽象层将其隐藏起来——利用帮助方法。

帮助方法

不管我们何时想将某个过程从一个视图中（任何一种视图）抽象出来，都应该写一个帮助方法。如果你查看 `app` 文件夹的话，会发现 5 个子目录：

```
depot> ls -p app
controllers/ helpers/ mailers/ models/ views/
```

毫不奇怪，我们的帮助方法应该放在 `helpers` 文件夹下。如果你查看那个文件夹的内容的话，会发现里面已经有一些文件：

```
depot> ls -p app/helpers
application_helper.rb line_items_helper.rb store_helper.rb
carts_helper.rb       products_helper.rb
```

Rails 生成器自动为每个控制器（产品和商店）生成了一个帮助方法的文件。Rails 命令本身（那个创建初始应用程序的命令）创建了文件 `application_helper.rb`。如果你喜欢，可以将方法组织到控制器特定的帮助方法中，但是因为这个方法会在应用程序的布局中用到，所以把它放到应用程序的帮助方法中。

现在写一个叫 `hidden_div_if` 的帮助方法。它接收一个条件、一个可选的属性集合和一个块。它将块的输出整合在 `<div>` 标签中，如果条件为真，就在该标签中增加 `display:none` 风格。在在线商店布局中使用它，如下所示：

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

```
<%= hidden_div_if(@cart.line_items.empty?, :id => "cart") do %>
  <%= render @cart %>
<% end %>
```

为了让帮助方法对商店控制器可见，我们得将它加入 `app/helpers` 文件夹下的 `application_helper.rb` 中：

```
depot_n/app/helpers/application_helper.rb
```

```
module ApplicationHelper
  def hidden_div_if(condition, attributes = {}, &block)
    if condition
      attributes["style"] = "display: none"
    end
    content_tag("div", attributes, &block)
  end
end
```

这段代码使用了 Rails 的标准帮助方法，`content_tag`，它可以整合标签中块的输出。通过使用 `&block` 标记（见 4.3.2 节），我们让 Ruby 将传递到 `hidden_div` 的块进一步传递至 `content_tag`。

最后，我们需要停止在闪存中设置信息，这个信息原本是当用户清空购物车时显示的。它不再需要了，因为当目录索引页重新显示的时候，购物车清楚地从侧边栏中消失了。并且，这里还有另外一个原因要移除它。现在我们使用 Ajax 来向购物车中添加商品，在人们购物的时候，主页面不会再在请求之间刷新了。这意味着，购物车为空的闪存消息会继续显示，即使当我们在侧边栏中显示购物车。

```
depot_n/app/controllers/carts_controller.rb
```

```
def destroy
  @cart = current_cart
  @cart.destroy
  session[:cart_id] = nil

  respond_to do |format|
    format.html { redirect_to(store_url) }
    format.xml { head :ok }
  end
end
```

到现在为止，我们已添加了所有这些 Ajax 的优良特性，试一试清空你的购物车并添加一个项目吧。

虽然这样看起来好像做了许多工作，但实际上只有两个关键步骤。第一，让购物车中项目的数目来决定 CSS 的显示风格，可让购物车隐藏或显示。第二，当购物车从空的状态变成有了一个项目，我们使用了一个 RJS 模板调用 `blind_down` 效果。

现在，好像我们并没有进行测试，但由于我们没有在功能上做太多的改变，所以应该没有问题。为了保险起见，再次运行测试：

```
depot> rake test
Loaded suite
Started
.....
Finished in 0.172988 seconds.
8 tests, 29 assertions, 0 failures, 0 errors

Loaded suite
```

```

Started
...E...F.EEEE...EEEE..
Finished in 0.640226 seconds.
23 tests, 29 assertions, 1 failures, 9 errors

```

哦！失败和错误。这可不是一件好事。显然，我们得重新考虑测试的方法了，我们马上就要这么做。

11.5 测试 Ajax 改变

通过检查这些测试失败，我们看到了一些像下面这样的错误信息：

```

test_should_get_index(ProductsControllerTest):
ActionView::Template::Error: undefined method 'line_items' for nil:NilClass

```

因为这个错误代表了所报告的大多数问题，所以先来解决它。根据该测试，如果我们得到了商品索引，就会有问题。果然，当向浏览器中输入地址 `http://localhost:3000/products/` 时，结果如图 11.3 所示。

这个信息非常有帮助。该消息标定了当错误发生时，正在处理的模板文件（`app/views/layouts/application.html.erb`）、错误发生的行数和该错误附近的模板文件的摘录。通过这一点，我们可以知道错误发生的时候正在计算的是表达式 `@cart.line_items`，并且产生的错误信息是 `undefined method 'line_items' for nil`。

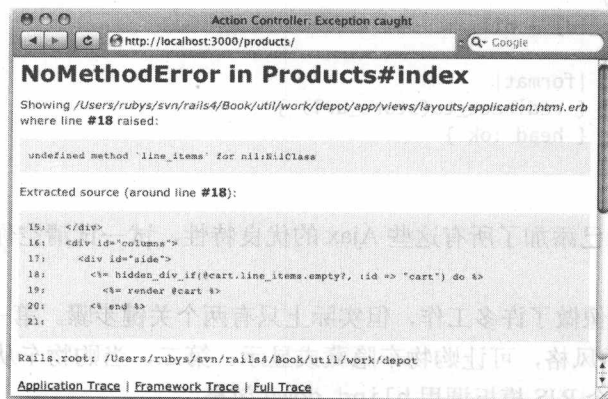


图 11.3 布局中的一个错误可以影响整个应用程序

因此，当我们显示商品的索引的时候，`@cart` 显然为 `nil`。这说得通，因为它是在商店控制器中被设置的。要修复这个错误非常简单，只需避免显示购物车，除非这个值被设置了：

```

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

▶ <% if @cart %>
  <%= hidden_div_if(@cart.line_items.empty?, :id => "cart") do %>
    <%= render @cart %>
  <% end %>
▶ <% end %>

```


修复这个问题之后，我们重新运行测试并发现错误减少了一个。重定向的值不是我们所期望的。该错误发生在创建一个行项目的时候。果然，我们是在 11.1 节中做的这个修改。与上个完全无意的修改不同的是，这个改变是有意的，因此我们来更新相应的函数测试案例：

```
depot_o/test/functional/line_items_controller_test.rb
```

```
test "should create line_item" do
  assert_difference('LineItem.count') do
    post :create, :product_id => products(:ruby).id
  end
  ▶ assert_redirected_to store_path
end
```

这个修改过后，我们的测试又一次通过了。想象会发生什么。为了支持一个新的需求，我们更改了应用程序的一部分，但这个修改却破坏了我们之前在应用程序另外一部分所写的函数。如果你不小心，这就会发生在像 Depot 一样的小的应用程序中。即使你小心了，这也会在一个大的应用程序中出现。

这还没结束。我们还没有测试所有与 Ajax 相关的其他东西，比如当我们按下 Add to Cart 按钮的时候会发生什么。使用 Rails 实现这个十分简单。

对于 should create line item，我们已经有有了一个测试，现在再添加另一个称为 should create line item via ajax 的测试：

```
depot_o/test/functional/line_items_controller_test.rb
```

```
test "should create line_item via ajax" do
  assert_difference('LineItem.count') do
    xhr :post, :create, :product_id => products(:ruby).id
  end
  assert_response :success
  assert_select_rjs :replace_html, 'cart' do
    assert_select 'tr#current_item td', /Programming Ruby 1.9/
  end
end
```

与其名字不同的是，该测试引用创建商品项目的测试方式（xhr: post 与简单的 post 比较，这里 xhr 的全称是 XMLHttpRequest）和预期的结果。我们预期获得一个成功的回复，而不是重定向。该回复包含了一个调用来代替购物车的 HTML，并且在该 HTML 中，我们应该可以找到一行 id 为 current_item，其值与 Programming Ruby 1.9 吻合。为了做到这一点，我们可以用 assert_select_rjs 来提取相关的 HTML，并且用你想要的其他任何断言来处理该 HTML。

维护应用程序的一个非常重要的部分是令测试集总是最新的。Rails 令这一点变得简单。测试是敏捷程序员的开发过程中不可缺少的一部分。许多程序员甚至先于第一行代码之前写下测试集。

11.6 本章小结

在本次迭代中，我们向购物车添加了对 Ajax 的支持：

- 我们将购物车移动到了侧边栏中。并安排 create 动作重新显示目录页面。
- 利用 Ajax, 我们使用了 `:remote => true` 来调用 `LineItemsController.create` 行为。
- 然后我们使用一个 RJS 模板, 用购物车的 HTML 来更新页面。
- 为了帮助用户更好地感知购物车的变化, 我们增加了一个高亮的效果, 这里再次利用了 RJS 模板。
- 我们写了一个帮助方法, 在购物车为空时来隐藏购物车, 当有项目添加进去后, 用 RJS 模板来显示该购物车。
- 我们写了一个测试, 它不仅验证了商品项目的创建, 还验证了该请求回复的内容。

这里的重点是 Ajax 的增量式开发。从一个普通的应用程序开始, 一步一步地增加 Ajax 特性。Ajax 很难调试: 通过慢慢地向应用程序中添加它, 如果你的应用程序不工作了, 要追溯起来什么东西被改变了也会变得简单。而且, 就像我们看见的一样, 从一个普通的应用程序开始可以让其在同一个代码库中更好地支持 Ajax 和非 Ajax 的行为。

最后给出我们的几点建议。第一, 如果你打算做很多 Ajax 开发, 或许你应该熟悉浏览器的 JavaScript 调试工具及其 DOM 检查器, 比如 Firefox 的 Firebug、Internet Explorer 8 的开发工具、Google Chrome 的开发工具、Safari 的 Web Inspector 或 Opera 的 Dragonfly。第二, Firefox 的 NoScript 扩展可以一键检查 JavaScript/ 无 JavaScript。其他人在开发时, 认为在两个不同的浏览器中 (一个开启了 JavaScript, 另一个没有) 运行十分有用。当加入新特性时, 可以在两个浏览器中查看, 以确保应用程序无论 JavaScript 是什么状态都能工作。

练习时间

可以自己尝试以下任务:

- 在 9.4 节的“练习时间”中, 其中一点是通过单击图片来将项目添加至购物车。在这里将这一过程修改成利用 `:remote => true` 和 `image_submit_tag` 来实现这一点。
 - 当前, 如果用户重新刷新整个购物目录可将购物车清空并隐藏。你可以用 `Script.aculo.us` 的 `blind_up` 效果来实现这一点吗?
 - 若是 JavaScript 被禁用, 这些所做的修改还可以工作吗?
 - 在新的购物车项目上尝试其他的视觉效果。例如, 你能将它们的初始状态设成隐藏, 然后让它们在空间中扩展吗? 这样做会给在 Ajax 代码和初始页面显示之间共享购物车项目的局部模板带来任何问题吗?
 - 在购物车的项目旁边添加一个链接。单击该链接时, 触发一个动作使得某个项目的数量减少, 并当数量为零时从购物车中将该项目删去。先不用 Ajax 实现, 然后再添加 Ajax 特性。
 - 为空的购物车写一个测试。验证其已经从侧边栏中删除。
- (可以在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 上面找到提示。)

第 12 章

任务 G：付款

在本章中，我们将学习：

- 链接表和外键
- 使用 `belongs_to`、`has_many` 和 `:through`
- 创建基于模型的表单 (`form_for`)
- 链接表单模型和视图
- 安装和使用插件
- 利用模型对象上的 `atom_` 帮助方法来生成推送

我们现在来评估应用程序。到目前为止，我们有了一个基本的商品管理系统，实现了商品目录，并且还有一个非常炫的购物车。那么现在我们得让顾客确实实购买购物车中的东西了。接下来实现付款的功能。

我们不会在这里介绍每个细节。当前要做的是得到顾客的联系信息和付款方式。利用这些信息，我们会在数据库中创建一个订单。在整个过程中，我们会着重介绍模型、验证和表单处理。

12.1 迭代 G1：获取订单

订单是商品项目及其购买交易的细节的集合。我们的购物车已经包含了 `line_items`，那么现在要做的只是在 `line_items` 表中添加一列 `order_id`，并且根据图 5.3 所示创建一个 `orders` 表格，该表格整合了与顾客的简短聊天记录。

首先，创建订单模型并更新 `line_items` 表格：

```
depot> rails generate scaffold order name:string address:text \
  email:string pay_type:string
...
depot> rails generate migration add_order_id_to_line_item \
  order_id:integer
```

创建好迁移之后，应用它们：

```
depot> rake db:migrate
== CreateOrders: migrating =====
-- create_table(:orders)
-> 0.0014s
== CreateOrders: migrated (0.0015s) =====

== AddOrderIdToLineItem: migrating =====
-- add_column(:line_items, :order_id, :integer)
-> 0.0008s
== AddOrderIdToLineItem: migrated (0.0009s) =====
```

因为数据库的 `schema_migrations` 表格中还没有这两个新东西的项目，所以 `db:migrate` 任务会将它们两个都迁移到数据库中。当然，我们也可以通过在创建每一个迁移之后运行迁移任务来单独地迁移它们。

Joe 问

信用卡处理在哪里？

在现实世界中，我们或许希望应用程序可以处理商业付款问题，甚至希望可以整合信用卡的处理。但是，与后台付款处理系统的整合需要许多文件和环节，这会分散对 Rails 的注意，所以这里会暂时搁置这个细节。

在 26.1 节中会再提到这一点，届时我们会使用一个插件来完成这个功能。

12.1.1 创建获取订单的表单

现在我们有了所需的表格和模型，可以开始付款的过程了。首先，在购物车中添加一个 Checkout 按钮。因为这样会创建一个新的订单，所以我们要将它链接回订单控制器的 `new` 动作：

```
depot_p/app/views/carts/_cart.html.erb
```

```
<div class="cart_title">Your Cart</div>
<table>
  <%= render(cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>
</table>
<%= button_to "Checkout", new_order_path, :method => :get %>
<%= button_to 'Empty cart', cart, :method => :delete,
  :confirm => 'Are you sure?' %>
```

我们要做的第一件事是检查购物车中是否有东西。如果没有，那么就将用户重定向回商店的首页，向用户提供有关我们刚刚做了什么的通知，并立即返回。这可以避免人们直接访问付款选项并创建空订单。返回语句在这里非常重要；没有它，就会发生 `double render error`^①，因为控制器会试图同时重定向和显示输出。

```
depot_p/app/controllers/orders_controller.rb
```

```
def new
  ▶ @cart = current_cart
  ▶ if @cart.line_items.empty?
  ▶   redirect_to store_url, :notice => "Your cart is empty"
  ▶   return
  ▶ end
```

① 双重显示错误。——译者注

```

▶ @order = Order.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @order }
  end
end

```

然后，为 `requires item in cart` 增加了一个测试，并修改 `should get new` 的测试以保证购物车中有商品：

```

depot_p/test/functional/orders_controller_test.rb
▶ test "requires item in cart" do
▶   get :new
▶   assert_redirected_to store_path
▶   assert_equal flash[:notice], 'Your cart is empty'
▶ end

test "should get new" do
▶   cart = Cart.create
▶   session[:cart_id] = cart.id
▶   LineItem.create(:cart => cart, :product => products(:ruby))
▶
  get :new
  assert_response :success
end

```

现在我们想要让 `new` 行为向用户展示一个表单，该表单让他们可以在订单表格中填写相关信息：他们的名字、地址、电子邮件地址以及付款方式。这意味着我们需要显示一个包含了表单的 Rails 模板。该表单的输入项需要连接到一个 Rails 模型对象的相应属性，因此我们得在 `new` 行为中创建一个空的模型对象以便让这些项可以在上面工作。

就像 HTML 表单一样，这里的技巧是向表单项中填入一些初始值，然后在用户单击提交按钮之后，取出这些值放回到应用程序中。

在这个控制器中，`@order` 实例变量被设成一个新的 `Order` 模型对象的引用。之所以这么做是因为视图从这个对象的数据中生成了表单。按照实际情况来说，这不是特别有意义。因为它是个新的模型对象，所有的项都是空的。但是，考虑一般情况。或许我们想要编辑一个已存在的订单。抑或用户试图输入一个订单，但数据验证却失败了。在这些情况下，我们希望显示表单的时候，任何在模型中存在的数据都会显示给用户。在此阶段传递一个空模型对象让所有的这些情况保持了一致——视图可以假定总是有一个模型对象可用。

然后，当用户单击了提交（submit）按钮，我们要提取表单中的新数据放到控制器的模型对象中。

幸运的是，Rails 让这个过程变得相对简单。它提供了很多 `form` 帮助方法。这些帮助方法与控制器和模型交互，以实现对表单处理的集成解决方案。在开始我们最后的表单之前，来看一个简单的例子：

```
<%= form_for @order do |f| %>
  <p>
    <%= f.label :name, "Name:" %>
    <%= f.text_field :name, :size => 40 %>
  </p>
<% end %>
```

这段代码有两点值得注意。第一，第 1 行的 `form_for` 帮助方法设置了一个标准的 HTML 表单。但它还做了更多的事。第 1 个参数 `@order` 告诉方法，当命名 fields 和安排传递回控制器的 field 值时，应该用什么样的实例变量。

你会看到 `form_for` 设置好了一个 Ruby 代码块环境（该代码块于第 6 行结束）。在这个代码块中，可以用普通的模板（例如 `<p>` 标签）。但是也可以用这个代码块的参数（在这个例子中是 `f`）去引用一个表单内容。我们在第 4 行用这个内容给表单增加了一个 text 项，因为该 text 项是在 `form_for` 的内容中创建的，所以它自动与 `@order` 对象中的数据相关联。

所有的这些关系有点让人迷惑。最重要的是要记住，Rails 需要同时知道与模型相关联的 names 和 values。`form_for` 和各种 field-level helpers（例如 `text_field`）的组合给出了这个信息。图 12.1 中显示了这个过程。

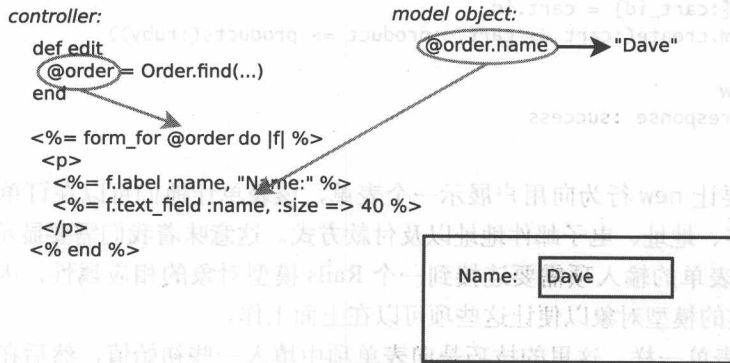


图 12.1 将 `form_for` 中的名字映射到对象和属性上

现在我们可以为获取顾客付款细节的表单更新模板了。它由订单控制器中的 `new` 动作调用，所以这个模板被命名为 `new.html.erb`，你可以在文件夹 `app/views/orders` 中找到它：

```
depot_p/app/views/orders/new.html.erb

<div class="depot_form">
  <fieldset>
    <legend>Please Enter Your Details</legend>
    <%= render 'form' %>
  </fieldset>
</div>
```

这个模板利用了一个名叫 `_form` 的局部模板：


```

depot_p/app/views/orders/_form.html.erb
<%= form_for(@order) do |f| %>
  <% if @order.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@order.errors.count, "error") %>
        prohibited this order from being saved:</h2>

      <ul>
        <% @order.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name, :size => 40 %>
  </div>
  <div class="field">
    <%= f.label :address %><br />
    <%= f.text_area :address, :rows => 3, :cols => 40 %>
  </div>
  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, :size => 40 %>
  </div>
  <div class="field">
    <%= f.label :pay_type %><br />
    <%= f.select :pay_type, Order::PAYMENT_TYPES,
      :prompt => 'Select a payment method' %>
  </div>
  <div class="actions">
    <%= f.submit 'Place Order' %>
  </div>
<% end %>

```

Rails 有许多表单帮助方法, 用来为所有不同层次的 HTML 表单元素服务。在之前的代码中, 我们使用 `text_field`、`email_field` 和 `text_area` 帮助方法去获取顾客的姓名、邮箱和地址。在 21.2 节中会详细介绍表单帮助方法。

这里唯一有点麻烦的是与选项列表相关的代码。我们已经假设可用的付款方式选项的列表是 `Order` 模型的一个属性。最好在我们忘记之前在模型 `order.rb` 中定义选项数组:

```

depot_p/app/models/order.rb
class Order < ActiveRecord::Base
  PAYMENT_TYPES = [ "Check", "Credit card", "Purchase order" ]
end

```

在这个模板中, 我们向 `select` 帮助方法传递了付款方式选项的数组。也传递了 `:prompt` 参数, 它添加了一个含有提示文字的伪选择。

让我们添加一点 CSS “魔法”:

```
depot_p/public/stylesheets/depot.css

/* Styles for order form */

.depot_form fieldset {
  background: #efe;
}

.depot_form legend {
  color: #dfd;
  background: #141;
  font-family: sans-serif;
  padding: 0.2em 1em;
}

.depot_form label {
  width: 5em;
  float: left;
  text-align: right;
  padding-top: 0.2em;
  margin-right: 0.1em;
  display: block;
}

.depot_form select, .depot_form textarea, .depot_form input {
  margin-left: 0.5em;
}

.depot_form .submit {
  margin-left: 4em;
}

.depot_form div {
  margin: 0.5em 0;
}
```

可以试试我们的表单了。向购物车中添加一点东西，然后单击 Checkout 按钮。应该可以看到如图 12.2 所示的页面。

看起来不错！在继续之前，让我们通过增加一些验证来完成 new 动作。我们要改变 Order 模型以检验顾客在所有项中输入的数据（包括在下拉菜单中的付款方式）。

我们还要验证付款方式是可接收值中的一个。

有人可能会想，既然付款方式的值是从仅含有效值的下拉菜单中选择的，为什么还要关心验证它。这是因为一个应用程序不能假设它所接收的值是从它所创建的表单而来的。什么也无法阻止恶意用户绕过我们的表单，直接向应用程序提交表单数据。如果恶意用户设置了一个未知的付款方式，他们或许可以免费得到我们的产品。

```
depot_p/app/models/order.rb

class Order < ActiveRecord::Base
  # ...
```

- ▶ validates :name, :address, :email, :pay_type, :presence => true
- ▶ validates :pay_type, :inclusion => PAYMENT_TYPES
- end

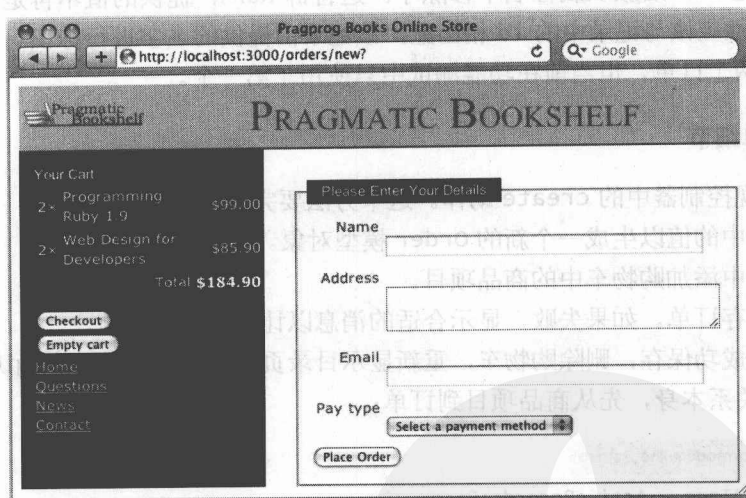


图 12.2 付款界面

注意，我们已经在页面的上方循环遍历了 `@order.errors`，这会报告验证失败。既然我们已经修改了验证规则，需要修改静态测试以匹配：

`depot_p/test/fixtures/orders.yml`

Read about fixtures at <http://ar.rubyonrails.org/classes/Fixtures.html>

one:

- ▶ name: Dave Thomas
- ▶ address: MyText
- ▶ email: dave@example.org
- ▶ pay_type: Check

two:

```
name: MyString
address: MyText
email: MyString
pay_type: MyString
```

并且，为了创建一个订单，购物车中需要有一个商品项目，所以我们也得修改商品项目的静态测试：

`depot_o/test/fixtures/line_items.yml`

Read about fixtures at <http://ar.rubyonrails.org/classes/Fixtures.html>

one:

- ▶ product: ruby
- ▶ cart: one

```
two:
  product: ruby
  cart: one
```

注意, 我们把 `_id` 后缀从属性名中移除了。这告诉 Rails, 提供的值不再是一个真正的数值 `id`; 相反, `id` 属性应该与记录中的 `id` 相匹配, 该记录由提供的名字进行标识。请随意修改两个订单, 但当前在功能测试中只使用了第一个。

12.1.2 获取订单细节

让我们来实现控制器中的 `create` 动作。这个方法要完成以下几件事:

- 1) 获取表单中的值以生成一个新的 `Order` 模型对象。
 - 2) 向该订单中添加购物车中的商品项目。
 - 3) 验证并保存订单。如果失败, 显示合适的消息以让用户纠正问题。
 - 4) 一旦订单成功保存, 删除购物车, 重新显示目录页, 并显示一则消息确认已下好订单。
- 首先, 定义关系本身, 先从商品项目到订单:

```
depot_p/app/models/line_item.rb
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product
  belongs_to :cart

  def total_price
    product.price * quantity
  end
end
```

然后, 从订单到商品项目, 再一次指明无论何时订单被销毁, 所有该订单的商品项目都要被销毁:

```
depot_p/app/models/order.rb
```

```
class Order < ActiveRecord::Base
  has_many :line_items, :dependent => :destroy
  # ...
end
```

最后, 方法本身如下所示:

```
depot_p/app/controllers/orders_controller.rb
```

```
def create
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(current_cart)

  respond_to do |format|
    if @order.save
      Cart.destroy(session[:cart_id])
      session[:cart_id] = nil
    end
  end
end
```

```

▶ format.html { redirect_to(store_url, :notice =>
▶   'Thank you for your order.') }
  format.xml { render :xml => @order, :status => :created,
    :location => @order }
  else
    format.html { render :action => "new" }
    format.xml { render :xml => @order.errors,
      :status => :unprocessable_entity }
  end
end
end
end

```

Joe 问

你没有创建重复订单吧？

Joe 很关心地来看我们的控制器，该控制器用两个动作 `new` 和 `create` 创建了订单模型对象。他想知道为什么这样不会导致数据库中的重复订单。

答案很简单：`checkout` 动作在内存中创建了一个 `Order` 对象来给模板代码工作。一旦将响应发给浏览器，就会抛弃那个特定的对象，并最终由 Ruby 的垃圾收集器清理。它从来无法接近数据库。

`create` 动作也创建了一个 `Order` 对象，它由表单中的项所生成。该对象不会保存在数据库中。所以模型对象有两个作用：它们将数据映射进或出数据库，但它们也只是保持商业数据的普通对象。当且仅当让它们影响数据库的时候它们才会这么做，通常情况下是通过调用 `save` 来实现。

首先让我们来创建一个新的 `Order` 对象，并用表单数据初始化它。在这个情况下，我们想要得到所有与订单对象相关的表单数据，所以从参数中选择 `:order` 散列（这是传递给 `form_for` 的第一个参数）。下一行将购物车中已存储的项目添加到该订单中——我们马上就会实现这个方法。

接下来让订单对象保存它本身（和它的孩子商品项目）到数据库中。在该过程中，订单对象会执行验证（我们后面再来讨论这一点）。如果保存成功，将会完成两件事情。第一，通过从会话中删除购物车，为顾客的下一个订单做好准备。第二，通过使用 `redirect_to` 方法重新显示目录并显示一条祝贺信息。相反，如果失败，则重新显示付款表单。

在 `create` 动作中，我们假定订单对象包含有 `add_line_items_from_cart` 方法，现在让我们来实现这个方法：

```

depot_p/app/models/order.rb
class Order < ActiveRecord::Base
  # ...
  ▶ def add_line_items_from_cart(cart)
  ▶   cart.line_items.each do |item|
  ▶     item.cart_id = nil
  ▶     line_items << item
  ▶   end
  ▶ end
end
end

```

对于每个从购物车转移到订单的项目，需要做两件事情。首先当销毁购物车的时候，将 `cart_id` 设为 `nil` 以防止项目消失。

接下来将项目本身添加到订单的 `line_items` 集合中。注意，我们不需要对各种不同的外键做任何特别的事情，比如在商品项目的行中设置 `order_id` 列以索引新创建的订单列，Rails 利用我们在 `Order` 和 `LineItem` 模型中添加的 `has_many` 和 `belongs_to` 声明来为我们实现这个功能。向 `line_items` 集合中附加的每个新商品项目都会将键管理这个责任交给 Rails。

我们还需要修改测试来反映这个新的重定向：

```
depot_p/test/functional/orders_controller_test.rb
```

```
require 'test_helper'
```

```
class OrdersControllerTest < ActionController::TestCase
```

```
  # ...
```

```
  test "should create order" do
```

```
    assert_difference('Order.count') do
```

```
      post :create, :order => @order.attributes
```

```
    end
```

```
  ▶ assert_redirected_to store_path
```

```
  end
```

```
  # ...
```

```
end
```

然后修改静态测试以便让一件商品位于第一位（在第一个购物车中留下一件商品是为了在购物车测试中使用）：

```
depot_p/test/fixtures/line_items.yml
```

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
```

```
one:
```

```
  product: ruby
```

```
▶ order: one
```

```
two:
```

```
  product: ruby
```

```
  cart: one
```

因此，作为所有这些的第一个测试，按下付款页面上的 `Place Order` 按钮且不填写任何表单项目。你应该可以看见付款页面上显示了一些有关未填写项的错误信息，如图 12.3 所示。（如果你尝试这些时得到的信息是 `No action responded to create`，这可能是因为在控制器中的 `private` 声明之后。私有方法不能称为动作。）

如果我们填写了一些数据（如图 12.4 的上部所示），然后单击 `Place Order` 按钮，我们应该被带回到目录，如图 12.4 的下部分所示。但究竟成功了吗？让我们看一下数据库。

```
depot> sqlite3 -line db/development.sqlite3
```

```
SQLite version 3.6.16
```

```
Enter ".help" for instructions
```



```
sqlite> select * from orders;
      id = 1
      name = Dave Thomas
      address = 123 Main St
      email = customer@example.com
      pay_type = Check
      created_at = 2010-06-09 13:40:40
      updated_at = 2010-06-09 13:40:40

sqlite> select * from line_items;
      id = 10
      product_id = 3
      cart_id =
      created_at = 2010-06-09 13:40:40
      updated_at = 2010-06-09 13:40:40
      quantity = 1
      price = 49.5
      order_id = 1

sqlite> .quit
```

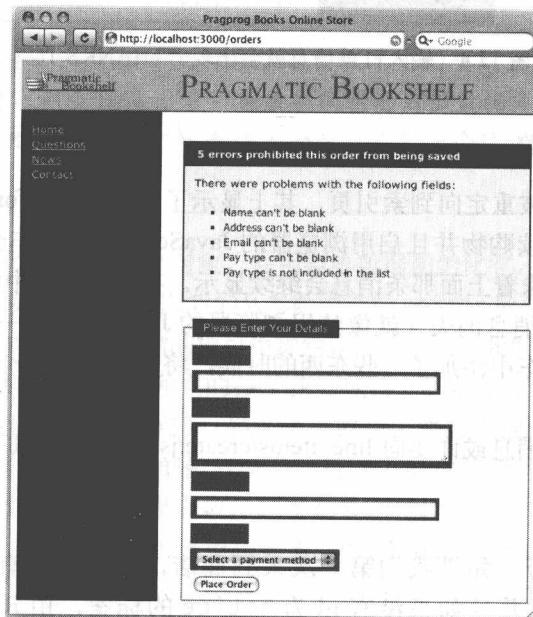


图 12.3 满堂红！所有的项都没有通过验证

虽然你所看见的会因为一些细节而不同，比如版本号和日期（仅当你完成了 10.3 节中的练习，价格才会显示），你应该可以看见一条订单和一件或多件你所选择的商品项目。

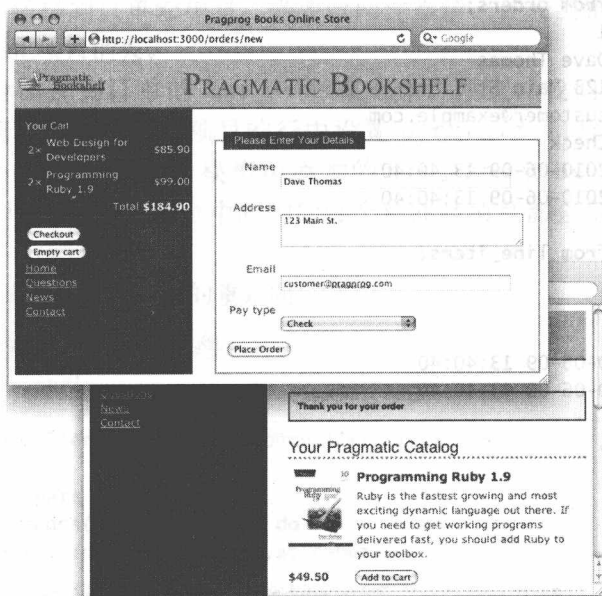


图 12.4 输入订单信息就会出现“THANKS！”

12.1.3 最后一个 Ajax 更改

接收订单之后，我们被重定向到索引页，其上显示了 Thank you for your order（感谢您的订单）的消息。如果用户继续购物并且启用浏览器的 JavaScript，购物车会被填充到侧边栏中，但却不会刷新主页面。这意味着上面那条消息会继续显示。在向购物车中添加了第一个商品项目的时候，我们更愿意让这条消息消失（就像禁用浏览器的 JavaScript 时一样）。幸运的是，这个修改很简单：当我们向购物车中添加了一些东西的时候，将包含消息的 `<div>` 隐藏起来。没有比这更简单的方法了。

第一次尝试隐藏这条消息或许要向 `line_items/create.js.rjs` 中添加以下代码：

```
page[:notice].hide
# rest as before...
```

但是，这样做却不行。如果我们第一次来到商店，闪存中没有任何消息，所以不会显示 id 为 `notice` 的段落。并且如果没有 id 为 `notice` 的标签，由 RJS 生成的试图隐藏它的 JavaScript 会显示出来，余下的模板就再也不会运行了。结果是，侧边栏中的购物车永远都不会更新。

解决方法用了一点特殊的技巧。仅当通知是可见的时候才运行 `.hide`，但 RJS 并不提供生成可以测试个体 id 的 JavaScript 的功能。但它却可以让我们循环遍历页面上符合特定 CSS selector 模式的标签。那么让我们来循环遍历所有 id 为 `notice` 的标签。该循环或者可以找到这样的标签，那我们就可以将它隐藏，或者找不到这样的标签，这样的话 `hide` 就不会被调用。

```
depot_p/app/views/line_items/create.js.rjs
```

```
▶ page.select("#notice").each { |notice| notice.hide }

page.replace_html('cart', render(@cart))
page[:cart].visual_effect :blind_down if @cart.total_items == 1
page[:current_item].visual_effect :highlight,
                                :startcolor => "#88ff88",
                                :endcolor => "#114411"
```

现在我们就已经可以获得订单，是时候通知接收订货的部门了。我们会用推送来实现，具体方式是，订单的 Atom 格式的推送。

12.2 循环 G2: Atom 推送

使用标准的推送格式，例如 Atom，意味着你可以立即利用许多已经存在的客户端。因为 Rails 已经知道了 id、日期和链接，它可以让你不用再担心那些烦人的细节，并让你专注于生成可读的汇总信息。第一步是向资源中添加一个新的动作，并在回复的列表格式中包含 Atom：

```
depot_o/app/controllers/products_controller.rb
```

```
def who_bought
  @product = Product.find(params[:id])
  respond_to do |format|
    format.atom
    format.xml { render :xml => @product }
  end
end
```

通过添加 `format.atom`，我们让 Rails 寻找一个名为 `who_bought.atom.builder` 的模板。该模板可以利用 Builders 提供的通用 XML 功能，以及 `atom_feed` 的帮助方法提供的有关 Atom 推送格式的信息：

```
depot_o/app/views/products/who_bought.atom.builder
```

```
atom_feed do |feed|
  feed.title "Who bought #{@product.title}"

  latest_order = @product.orders.sort_by(&:updated_at).last
  feed.updated( latest_order && latest_order.updated_at )

  @product.orders.each do |order|
    feed.entry(order) do |entry|
      entry.title "Order #{order.id}"
      entry.summary :type => 'xhtml' do |xhtml|
        xhtml.p "Shipped to #{order.address}"

        xhtml.table do
          xhtml.tr do
            xhtml.th 'Product'
            xhtml.th 'Quantity'
            xhtml.th 'Total Price'
          end
        end
      end
    end
  end
end
```

```

order.line_items.each do |item|
  xhtml.tr do
    xhtml.td item.product.title
    xhtml.td item.quantity
    xhtml.td number_to_currency item.total_price
  end
end
xhtml.th 'total', :colspan => 2
xhtml.th number_to_currency \
  order.line_items.map(&:total_price).sum
end
end

xhtml.p "Paid by #{order.pay_type}"
end
entry.author do |author|
  entry.name order.name
  entry.email order.email
end
end
end
end

```

Joe 问

为什么使用 Atom?

有很多不同的推送格式，最重要的格式为 RSS 1.0、RSS 2.0 和 Atom，分别于 2000、2002、2005 年标准化，这三种格式得到了最广泛的支持。为了支持事务，许多站点都为相同的页面提供多种推送，但是这种做法已经不是必需的，这只会增加用户的困惑，一般不予推荐。

Ruby 语言提供了一个底层的库，它可以生成这些格式中的任意一种，它也支持其他一些不那么常用的 RSS 版本。为了达到最好的效果，建议还是使用上述三种最主要的版本。

Rails 框架中总是使用合理的默认，它选择了 Atom 作为默认的推送格式。这种格式是由 IETF 指定的一种因特网社区的 Internet standards track protocol，Rails 提供了一个名为 `atom_feed` 的高层帮助方法，它依据 Rails 的命名习惯（例如 id 和日期），来帮你处理许多细节。

更多关于 Builder 的信息参见 25.1 节。

在推送的总体层，我们只需要提供两条信息：标题和最近更新的日期。如果没有订单，`updated_at` 的值为空，Rails 会为此提供当前的时间。

然后循环遍历与该产品有关的订单。注意，这两种模型之间没有直接的关系。事实上是间接的。商品有很多 `line_items` 而 `line_items` 属于一个订单。通过简单地声明 `line_items` 相关的产品与订单的关系，我们得以迭代和遍历，简化的代码如下：

```
depot_o/app/models/product.rb
```

```
class Product < ActiveRecord::Base
  default_scope :order => 'title'
  has_many :line_items
  ▶ has_many :orders, :through => :line_items
  # ...
end
```

对于每一个订单都提供标题、总结和作者。总结可以是完整的 XHTML，利用它可以生成商品名称，数量和总价表格。该表之后是一个包含 `pay_type` 的段落。

要让整个过程得以工作，我们还要定义一条路径。该动作会响应 HTTP GET 请求，并操作于集合的一个成员上（换句话说，在一个单一的产品上），而不是整个集合本身（也就是说全部的产品）：

```
depot_o/config/routes.rb
```

```
Depot::Application.routes.draw do
  resources :orders
```

```
resources :line_items
```

```
resources :carts
```

```
get "store/index"
```

```
▶ resources :products do
  ▶ get :who_bought, :on => :member
  ▶ end
```

```
# ...
```

```
# You can have the root of your site routed with "root"
# just remember to delete public/index.html.
# root :to => "welcome#index"
root :to => 'store#index', :as => 'store'
```

```
# ...
```

```
end
```

让我们来试一下：

```
depot> curl --silent http://localhost:3000/products/3/who_bought.atom
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <id>tag:localhost,2005:/products/3/who_bought</id>
  <link type="text/html" href="http://localhost:3000" rel="alternate"/>
  <link type="application/atom+xml"
    href="http://localhost:3000/info/who_bought/3.atom" rel="self"/>
  <title>Who bought Programming Ruby 1.9</title>
  <updated>2010-04-25T03:14:05Z</updated>
  <entry>
    <id>tag:localhost,2005:Order/1</id>
```

```
<published>2010-04-25T03:14:05Z</published>
<updated>2010-04-25T03:14:05Z</updated>
<link rel="alternate" type="text/html"
  href="http://localhost:3000/orders/1"/>
<title>Order 1</title>
<summary type="xhtml">
  <div xmlns="http://www.w3.org/1999/xhtml">
    <p>Shipped to 123 Main St</p>
    <table>
      ...
    </table>
    <p>Paid by check</p>
  </div>
</summary>
<author>
  <name>Dave Thomas</name>
  <email>customer@pragprog.com</email>
</author>
</entry>
</feed>
```

看起来不错。现在就可以将它订阅到我们最喜欢的推送阅读器上了。

12.3 迭代 G3: 分页

现在我们只有少量的商品和购物车，每个购物车或订单中只有少量的商品，但是本质上可以有任意多的订单，而且我们当然希望有足够多的订单，自然而然地我们会发现把所有这些都显示在一个订单页面上十分笨拙。我们可以借助 `will_paginate` 插件。这个插件扩展了 Rails 并提供了一个十分有用的函数。

为什么使用插件？在 Rails 1.0 的时候，这个功能是 Rails 本身的一部分。但是当时有许多不同的关于如何实现和改进它的想法，最后这个函数被剥离出去以便日后的创新。

第一步要做的是告诉 Rails 使用插件的意图。这个可以通过修改 `Gemfile` 文件来实现。我们要指明 Rails 的版本需大于或等于 3.0.pre，之前的版本都无法与 Rails 3.0 一起工作。

depot_q/Gemfile

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

```
gem 'rails', '3.0.5'
```

```
# Bundle edge Rails instead:
```

```
# gem 'rails', :git => 'git://github.com/rails/rails.git'
```

```
gem 'sqlite3'
```

```
# Use unicorn as the web server
```

```
# gem 'unicorn'
```

```
# Deploy with Capistrano
```

```
# gem 'capistrano'
```



```
# To use debugger (ruby-debug for Ruby 1.8.7+, ruby-debug19 for Ruby 1.9.2+)
# gem 'ruby-debug'
# gem 'ruby-debug19', :require => 'ruby-debug'

# Bundle the extra gems:
# gem 'bj'
# gem 'nokogiri'
# gem 'sqlite3-ruby', :require => 'sqlite3'
# gem 'aws-s3', :require => 'aws/s3'

▶
▶ gem 'will_paginate', '>= 3.0.pre'

# Bundle gems for the local environment. Make sure to
# put test-only gems in this group so their generators
# and rake tasks are available in development mode:
# group :development, :test do
#   gem 'webrat'
# end
```

完成了这个之后，使用 **bundle** 命令来安装依赖：

```
depot> bundle install
```

根据所使用的系统和设置，该命令可能需要 root 权限来运行。

bundle 命令实际上可做的事情远不止这些。它会反复验证 **gem** 依赖，找到一个可行的配置，并下载和安装所需的组件。但现在我们并不需要关心这些；我们只需添加一个组件，并随后确认该安装的组件包含在 **gem** 中。

在更新或安装完一个新的 **gem** 之后还需要做一件事：重启服务器。虽然 Rails 擅长检测并更新应用程序的最新修改，但是它却不可能预测当添加或替换整个 **gem** 时需要做什么。

现在让我们来生成一些测试数据。我们可以重复单击已有按钮，但计算机更擅长做这件事。这不完全是生成数据，只不过是需要做一次便可丢弃。让我们在 **script** 目录下创建一个文件。

```
depot_q/script/load_orders.rb

Order.transaction do
  (1..100).each do |i|
    Order.create(:name => "Customer #{i}", :address => "#{i} Main Street",
      :email => "customer-#{i}@example.com", :pay_type => "Check")
  end
end
```

它会创建 100 个无商品的订单。如果你想创建一些商品，尽管修改这个脚本。注意，该代码在一个事务中完成了所有这些工作。这虽不是该活动百分之百所需，但却可以加速整个过程。

注意，这里没有任何 **require** 语句或初始化来打开和关闭数据库。让 Rails 来处理这个：

```
rails runner script/load_orders.rb
```

现在设置完毕，可以开始对应用程序做出必要的修改了。首先，修改控制器以调用 **paginate**，并将我们想要显示结果的页面和订单传递给此函数：

```
depot_p/app/controllers/orders_controller.rb
```

```
def index
```

```
▶ @orders = Order.paginate :page=>params[:page], :order=>'created_at desc',
▶ :per_page => 10
```

```
respond_to do |format|
```

```
format.html # index.html.erb
```

```
format.xml { render :xml => @orders }
```

```
end
```

```
end
```

然后，在索引视图的底部添加链接：

```
depot_q/app/views/orders/index.html.erb
```

```
<h1>Listing orders</h1>
```

```
<table>
```

```
<tr>
```

```
<th>Name</th>
```

```
<th>Address</th>
```

```
<th>Email</th>
```

```
<th>Pay type</th>
```

```
<th></th>
```

```
<th></th>
```

```
</tr>
```

```
<% @orders.each do |order| %>
```

```
<tr>
```

```
<td><%= order.name %></td>
```

```
<td><%= order.address %></td>
```

```
<td><%= order.email %></td>
```

```
<td><%= order.pay_type %></td>
```

```
<td><%= link_to 'Show', order %></td>
```

```
<td><%= link_to 'Edit', edit_order_path(order) %></td>
```

```
<td><%= link_to 'Destroy', order, :confirm => 'Are you sure?',
:method => :delete %></td>
```

```
</tr>
```

```
<% end %>
```

```
</table>
```

```
<br />
```

```
<%= link_to 'New Order', new_order_path %>
```

```
▶ <p><%= will_paginate @orders %></p>
```

这就完成了！默认是每页显示 30 项，并且仅当有多于一页的订单时链接才会出现。

利用 :per_page 选项，控制器指定了每页显示的订单数量，如图 12.5 所示。

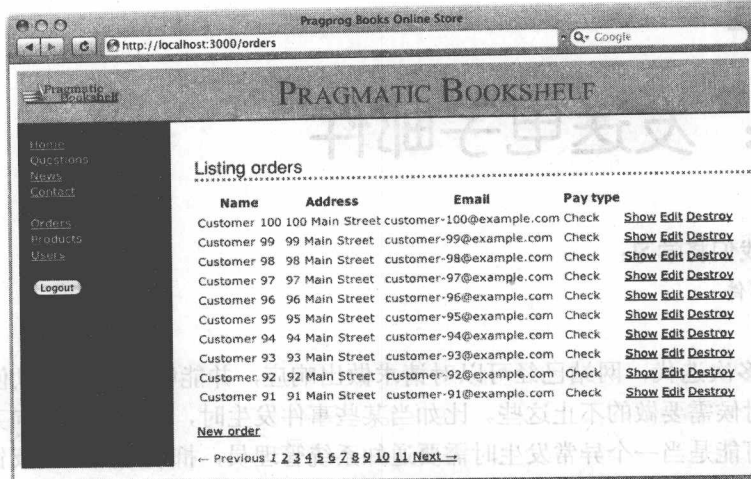


图 12.5 从超过 100 个订单中显示其中 10 个

顾客很满意。我们实现了产品的维护，基本的目录和购物车，现在又有了一个简单的订单系统。显然我们还得写一个类似履行的应用，但这要等到一个新的迭代了。（在此书中忽略了此迭代，因为其中没有介绍任何新的 Rails 东西。）

12.4 本章小结

在一个相对短的时间内，我们完成了以下事项：

- 创建了一个表单用以捕获订单细节，并将它连接到了一个新的订单模型。
- 添加了验证并使用了帮助方法来向用户显示错误。
- 安装并使用了一个插件以对订单列表进行分页。
- 提供了推送服务以便让管理员监视订单。

练习时间

可以自己尝试以下任务：

- 让 HTML、XML 和 JSON 格式的视图可以为 who_bought 请求工作。通过呈现 @product.to_xml(:include => :orders)，试验一下在 XML 视图中包含订单信息。用 JSON 来做同样的事。
- 当已经显示付款界面时，如果你单击侧边栏中的 Checkout 按钮会发生什么？你能在这种情形下找到一种方法来禁用此按钮吗？（提示：控制器中的变量集合在布局、局部模板和直接呈现的模板中都是可用的。）
- 当前可能的付款方式的列表是以一个常量的形式存储在 Order 类中。你可以把这个列表移到一个数据库的表中吗？你还能验证这项功能吗？

（可以在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 上面找到提示。）

第 13 章

任务 H: 发送电子邮件

在本章中，我们将学习：

- 发送电子邮件
- 集成测试

经过前面的多次迭代，网站已经可以对请求做出响应，并能够提供推送以便定期检查各个销售项目。有些时候需要做的不止这些。比如当某些事件发生时，可以主动地向某个特定的人发送一条消息，也可能是当一个异常发生时需要通知系统管理员，抑或是用户的反馈表单。在本章中，我们选择简单地向下订单的用户发送确认邮件。一旦完成了这个功能，不仅要为刚添加的邮件功能创建测试，还要为迄今为止创建的整个用户场景创建测试。

13.1 迭代 H1: 发送确认邮件

在 Rails 中发送邮件有三个基本的步骤：配置如何发送邮件，确定何时发送，以及指定要写的内容。下面按顺序介绍这三个步骤。

13.1.1 配置邮件

配置邮件是 Rails 应用程序环境的一部分，它要用到 `Depot::Application.configure` 代码块。如果想要在开发、测试和生产中使用相同的配置，可以在 `config` 目录下的 `environment.rb` 中添加此配置；否则，在 `config/environments` 目录下相应的文件中添加不同的配置。

在代码块内部需要一个或多个语句。首先要决定如何递送邮件：

```
config.action_mailer.delivery_method = :smtp | :sendmail | :test
```

当想要用 Action Mailer 来尝试发送邮件时，使用选项 `:smtp` 和 `:sendmail`。在产品环境上，你肯定需要用到其中的方法。

`:test` 设置对单元和功能测试非常有用，我们会在本节后面介绍。使用该选项时邮件实际并不会被发送，而是添加到一个数组中（该数组可通过属性 `ActionMailer::Base.deliveries` 访问）。在测试环境中，这是默认的发送函数。有意思的是，在开发模式下默认的是 `:smtp`。如果想要 Rails 在应用程序开发过程中发送邮件，这样做没什么问题。如果你更愿意在开发模式下禁用邮件发送，编辑目录 `config/environments` 下的 `development.rb` 文件，添加如下几行：

```
Depot::Application.configure do
  config.action_mailer.delivery_method = :test
end
```

`:sendmail` 设置将邮件发送委托给本地系统下的 `sendmail` 程序，假定其在目录 `/usr/sbin` 中。

这种发送机制可移植性差，因为在不同的操作系统下 sendmail 不总是在该目录下。另外，它还取决于本地的 sendmail 对命令行选项 `-i` 和 `-t` 的支持。

通过让该选项保持它默认的值 `:smtp`，可以达到更好的可移植性。如果这样做了，还需要指定另外一些配置，以告诉 Action Mailer 在哪里能找到 SMTP 服务器，该服务器用来处理待发送的电子邮件。它可以是运行 Web 应用的机器，也可以是一个单独的机器（如果 Rails 在托管环境下运行的话，它或许在互联网服务提供商那儿）。系统管理员可以告诉你如何设置这些参数。或许也可以通过自己的邮件客户端的配置来确定这些参数。

下面是 Gmail 的典型设置。请根据需要做出相应修改。

```
Depot::Application.configure do
  config.action_mailer.delivery_method = :smtp

  config.action_mailer.smtp_settings = {
    :address      => "smtp.gmail.com",
    :port         => 587,
    :domain       => "domain.of.sender.net",
    :authentication => "plain",
    :user_name    => "dave",
    :password     => "secret",
    :enable_starttls_auto => true
  }
end
```

当所有的配置更改完成之后，如果对任何环境文件做了修改，需要重启应用程序。

13.1.2 发送邮件

所有配置已经做好了，现在来实现发送邮件。

到现在为止，你不应该对 Rails 中由生成脚本来创建 mailer 感到惊讶，应该惊讶在哪里创建它们。在 Rails 中，mailer 是存放在目录 `app/mailers` 下的一个类，它包含一个或多个方法，每个方法对应一个邮件模板。这些方法依次使用视图来创建邮件体（与控制器行为使用视图来创建 HTML 和 XML 的方式相同）。那么让我们开始为商店应用创建一个 mailer。它用来发送两种不同的邮件：一种是当订单下达的时候，另一种是当订单发货的时候。命令 `rails generate mailer` 以 mailer 类的名字和邮件行为方法的名字作为参数：

```
depot> rails generate mailer Notifier order_received order_shipped
create    app/mailers/notifier.rb
invoke    erb
create    app/views/notifier
create    app/views/notifier/order_received.text.erb
create    app/views/notifier/order_shipped.text.erb
invoke    test_unit
create    test/functional/notifier_test.rb
```

注意，在 `app/mailers` 下创建了一个 `Notifier` 类和 `app/views/notifier` 中的两个模板文件，分别对应两种不同的邮件类型（还有一个测试文件，将在本节后面内容中详细介绍）。

在 mailer 类中的每个方法都负责为发送特定邮件设置环境。在深入介绍细节之前，先来看

一个例子。下面是一些为 `Notifier` 类所生成的代码，更改了一个默认项：

```
depot_p/app/mailers/notifier.rb

class Notifier < ActionMailer::Base
  ▶ default :from => 'Sam Ruby <depot@example.com>'

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.notifier.order_received.subject
  #
  def order_received
    @greeting = "Hi"

    mail :to => "to@example.org"
  end
  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.notifier.order_shipped.subject
  #
  def order_shipped
    @greeting = "Hi"

    mail :to => "to@example.org"
  end
end
```

你可能认为这很像一个控制器，其实它就是一个控制器。每个行为一个方法。但与调用呈现不同的是，这里是调用 `mail`。`mail` 接收一系列参数，包括 `:to`（如上所示）、`:cc`、`:from` 和 `:subject`，每个参数的作用都如其字面上的意思。在 `mailer` 中所有 `mail` 调用的共同的值可以通过调用 `default` 来设置，就像此类上的 `:from` 所做的那样。尽管按照需要修改这些设置。

此类中的注释也指明主题行（subject line）已经启用了翻译，有关该主题会在第 15 章中讲到。现在，我们简单地使用 `:subject` 参数。

就像控制器一样，模板包含了待发送的文本，并且控制器和 `mailer` 可以通过实例变量来提供在那些模板中插入的值。

13.1.3 邮件模板

生成脚本在 `app/views/notifier` 中创建了两个邮件模板，每个都对应 `Notifier` 类中的动作，它们是普通的 `.erb` 文件。我们将使用它们来创建纯文本格式的邮件（之后会介绍如何创建 HTML 格式的邮件），就像那些用来创建应用程序页面的模板一样，这些文件包括了静态文字和动态内容的组合。在 `order_received.text.erb` 中可以定制该模板。下面是用来确认订单的邮件：

```
depot_r/app/views/notifier/order_received.text.erb

Dear <%= @order.name %>
```


Thank you for your recent order from The Pragmatic Store.

You ordered the following items:

```
<%= render @order.line_items %>
```

We'll send you a separate e-mail when your order ships.

呈现商品的局部模板用商品的数量和名字格式化了一行。由于处在模板之内，所有的普通帮助方法，如 `truncate`，都是可用的：

```
depot_r/app/views/line_items/_line_item.text.erb
```

```
<%= sprintf("%2d x %s",
  line_item.quantity,
  truncate(line_item.product.title, :length => 50)) %>
```

现在必须回到 `Notifier` 类中填入 `order_received` 方法所需要的数据：

```
depot_r/app/mailers/notifier.rb
```

```
def order_received(order)
  @order = order
```

```
  mail :to => order.email, :subject => 'Pragmatic Store Order Confirmation'
end
```

在这里，向接收方法的调用（method-received call）添加了 `order` 参数，增加了拷贝传给实例变量的参数的代码，并更新 `mail` 调用，该调用指定发送邮件的目标和使用的邮件主题。

13.1.4 生成邮件

设置好模板并定义了 `mailer` 方法之后，就可以在普通的控制器中使用它们来创建或发送邮件了。

```
depot_r/app/controllers/orders_controller.rb
```

```
def create
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(current_cart)
```

```
  respond_to do |format|
```

```
    if @order.save
```

```
      Cart.destroy(session[:cart_id])
```

```
      session[:cart_id] = nil
```

```
      ▶ Notifier.order_received(@order).deliver
```

```
      format.html { redirect_to(store_url, :notice =>
```

```
        'Thank you for your order.') }
```

```
      format.xml { render :xml => @order, :status => :created,
        :location => @order }
```

```
    else
```

```
      format.html { render :action => "new" }
```

```
      format.xml { render :xml => @order.errors,
        :status => :unprocessable_entity }
```

```
    end
```

```
  end
```

```
end
```

我们需要更新 `order_shipped`，就像对 `order_received` 所做的那样：

```
depot_r/app/mailers/notifier.rb
```

```
def order_shipped(order)
  @order = order
```

```
  mail :to => order.email, :subject => 'Pragmatic Store Order Shipped'
end
```

至此，所有基本的设置已就绪，若没有禁用开发模式下发送邮件的功能，你可以下一个订单，一封简单的邮件就会发送给你自己。现在来给邮件添加一些格式。

13.1.5 发送多内容类型

一些人喜欢接收纯文本格式的邮件，而另一些人更偏好看起来像 HTML 的邮件。Rails 使得发送可选内容格式的邮件变得简单，它让用户（或是他们的邮件客户端）来决定自己喜欢的视图。

在本节前面的内容中，我们创建了纯文本格式的邮件。`order_received` 行为的视图文件叫做 `order_received.text.erb`，这是标准的 Rails 命名习惯。按照同样的方式可以创建 HTML 格式的邮件。

让我们用订单发货提醒来试试。不需要修改任何代码，只需简单地创建一个新的模板：

```
depot_r/app/views/notifier/order_shipped.html.erb
```

```
<h3>Pragmatic Order Shipped</h3>
```

```
<p>
```

```
  This is just to let you know that we've shipped your recent order:
```

```
</p>
```

```
<table>
```

```
  <tr><th colspan="2">Qty</th><th>Description</th></tr>
```

```
  <%= render @order.line_items %>
```

```
</table>
```

我们甚至不需要修改局部模板，因为已有的那个就足够了：

```
depot_r/app/views/line_items/_line_item.html.erb
```

```
<% if line_item == @current_item %>
```

```
<tr id="current_item">
```

```
<% else %>
```

```
<tr>
```

```
<% end %>
```

```
  <td><%= line_item.quantity %><times></td>
```

```
  <td><%= line_item.product.title %></td>
```

```
  <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
```

```
</tr>
```

但是，对于邮件模板，有个小的命名技巧。如果你用相同的名字创建了多个模板，但是在名字中嵌入了不同的内容类型，Rails 会在一封邮件中将它们全部发送，在此邮件中内容会以邮件客户端可以将它们分辨的方式来安排。

这意味着，要么更新，要么删除由 Rails 提供给 `order_shipped` 的用于提醒的纯文本模板。

Joe 问

我也可以接收电子邮件吗？

Action Mailer 让实现处理接收邮件的 Rails 程序变得简单。遗憾的是，你需要找到一种方法来从服务器环境中得到相应的邮件，并将它们注入应用程序中，这需要做一点工作。

简单的部分是在应用程序内部处理邮件。在 Action Mailer 类中，实现一个叫做 `receive` 的实例方法，让它接收单一参数。该参数是一个对应来件（收到的邮件）的 `Mail::Message` 对象。你可以提取出相应的项、正文和附件，并在应用程序中使用它们。

大凡截取接收邮件的技术都归结为运行一条命令，该命令以传入邮件的内容作为标准输入。如果让 Rails 的 `runner` 脚本作为每次邮件到达后调用的命令，便可将该邮件传入到应用程序的邮件处理代码中。例如，利用基于 `procmail` 的拦截，可以写一条如下所示的规则。借助 `procmail` 的神秘语法，这条规则拷贝所有主题行包含“Bug Report”的来件到 `runner` 脚本：

```
RUBY=/opt/local/bin/ruby
TICKET_APP_DIR=/Users/dave/Work/depot
HANDLER='IncomingTicketHandler.receive(STDIN.read)'
```

```
:0 c
* ^Subject:.*Bug Report.*
| cd $TICKET_APP_DIR && $RUBY runner $HANDLER
```

`receive` 类方法对所有 Action Mailer 类都是可用的。它接收邮件文本，将它解析成一个 `Tmail` 对象，创建一个 `receiver` 类的实例，并将 `Mail` 对象传入 `receiver` 类的 `receive` 实例方法中。

13.1.6 邮件功能测试

当我们使用生成脚本来创建订单 `mailer` 的时候，该脚本自动在应用程序的 `test/functional` 文件夹中构建了一个相应的 `notifier.rb` 文件。该文件直截了当；它调用每个动作并且验证生成邮件的选定部分。因为已经调整了邮件，为了匹配，让我们更新这个测试用例：

```
depot_r/test/functional/notifier_test.rb
```

```
require 'test_helper'
```

```
class NotifierTest < ActionMailer::TestCase
```

```
  test "order_received" do
```

```
    ▶ mail = Notifier.order_received(orders(:one))
```

```
    ▶ assert_equal "Pragmatic Store Order Confirmation", mail.subject
```

```
    ▶ assert_equal ["dave@example.org"], mail.to
```

```
    ▶ assert_equal ["depot@example.com"], mail.from
```

```
    ▶ assert_match /1 x Programming Ruby 1.9/, mail.body.encoded
```

```
  end
```

```

test "order_shipped" do
  ▶ mail = Notifier.order_shipped(orders(:one))
  ▶ assert_equal "Pragmatic Store Order Shipped", mail.subject
  ▶ assert_equal ["dave@example.org"], mail.to
  ▶ assert_equal ["depot@example.com"], mail.from
  ▶ assert_match /<td>1&times;</td>\s*<td>Programming Ruby 1.9</td>/,
  ▶ mail.body.encoded
end
end

```

该测试方法指导 mail 类创建（但不发送）一封邮件，并且使用了断言来验证该动态内容是我们所期望的。注意使用 `assert_match` 来验证部分邮件正文内容的用法。实际的结果可能因 `Notifier` 中 `default:from` 的裁剪方式的不同而不同。

至此，已经验证了消息的格式是正确的，但还没有验证当顾客完成了整个订单过程之后，它会被发送。对此，我们要部署集成测试。

13.2 迭代 H2：应用程序的集成测试

Rails 将测试分成单元测试、功能测试和集成测试。在解释集成测试之前，简单地来回顾一下到目前为止我们所讲的内容：

模型的单元测试

模型类包含了业务的逻辑。例如，当向购物车中添加了一个产品，购物车模型类检查该产品是否已经存在于购物车中。如果是，那么就增加该项的数量；如果否，那么就为该产品新增一项。

控制器的功能测试

控制器导演整个过程。它们接收进入的网络请求（通常是用户输入），与模型交互以获取应用程序状态，并且以向用户显示相应的视图来响应请求。所以当测试控制器时，必须确保一个给定的请求得到适当响应。我们仍然需要模型，但它们已经有了单元测试。

测试的下一个层次是检查应用程序的流程。很多时候，这就像是测试刚开始编写应用程序时顾客告诉我们的场景一样。例如，我们被告知了如下情形：用户浏览商店的索引页面。他选择了一个产品，添加到购物车中。然后付款，在付款表单中填写细节。当他提交之后，在数据库中就创建了一个订单，其中包含他的信息和购物车中产品的项目。一旦订单被接收，发送邮件确认购买。

这就是一个集成测试的完美材料。集成测试模拟了一个或多个虚拟用户和应用程序之间的连续会话。你可以使用它们来发送请求、监视响应、跟踪重定向等。

当创建了一个模型或控制器时，Rails 相应地创建了单元或功能测试。虽然集成测试不会被自动生成，但可以用一个生成器来创建它。

```

depot> rails generate integration_test user_stories
invoke test_unit
create test/integration/user_stories_test.rb

```

注意, Rails 自动添加 `_test` 到测试名字中。

让我们看一下生成的文件:

```
require 'test_helper'
```

```
class UserStoriesTest < ActionController::IntegrationTest
  fixtures :all

  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

让我们开始实现情景测试。因为只需测试购买产品, 所以只需产品的静态测试。

只用载入下面这个静态测试, 而不是所有的静态测试:

```
fixtures :products
```

现在让我们写一个称为 `buying a product` 的测试。在测试的末尾, 我们想要在 `orders` 表中添加一个订单, 并且在 `line_items` 表中加入一个产品项目, 因此在开始之前先清空它们。并且由于要经常使用 Ruby book 静态测试数据, 让我们将其载入到一个本地变量中:

```
depot_r/test/integration/user_stories_test.rb

LineItem.delete_all
Order.delete_all
ruby_book = products(:ruby)
```

首先实现情景中的第一句话: 用户浏览商店的索引页面。

```
depot_r/test/integration/user_stories_test.rb

get "/"
assert_response :success
assert_template "index"
```

这看起来非常像一个功能测试。最主要的差别是 `get` 方法。在功能测试中, 我们只检查一个控制器, 因此, 当调用 `get` 的时候只需指定一个动作。但是在集成测试中, 我们可以在整个应用程序中活动, 所以需要为控制器和被调用的动作传递一个完整(相对)的 URL。

情景中的下一句是: 他选择了一个产品, 添加到购物车中。我们知道应用程序使用一个 Ajax 请求来将物品添加到购物车中, 因此让我们使用 `xml_http_request` 方法来调用这个动作。当它返回时, 要检查购物车中包含有被请求的产品:

```
depot_r/test/integration/user_stories_test.rb

xml_http_request :post, '/line_items', :product_id => ruby_book.id
assert_response :success

cart = Cart.find(session[:cart_id])
assert_equal 1, cart.line_items.size
assert_equal ruby_book, cart.line_items[0].product
```

继续下一个步骤：然后付款……在测试中这非常简单：

```
depot_r/test/integration/user_stories_test.rb
```

```
get "/orders/new"
assert_response :success
assert_template "new"
```

当前，用户必须在付款表单中填写他们的细节。一旦完成并发送了数据，应用程序便会创建该订单并重定向到索引页面。让我们从 HTTP 端的发送表单数据到 `save_order` 动作和验证被重定向到索引页开始。还需检查此时购物车是否为空。测试帮助方法 `post_via_redirect` 生成 `post` 请求，然后跟随所有返回的重定向，直到没有返回非重定向响应（nonredirect response）。

```
depot_r/test/integration/user_stories_test.rb
```

```
post_via_redirect "/orders",
                  :order => { :name    => "Dave Thomas",
                              :address  => "123 The Street",
                              :email    => "dave@example.com",
                              :pay_type => "Check" }
assert_response :success
assert_template "index"
cart = Cart.find(session[:cart_id])
assert_equal 0, cart.line_items.size
```

接着进入到数据库中，保证订单和相应的商品项目已创建，并且它们的细节都是正确的。因为 `orders` 表在测试开始的时候已经清空，此时只需验证它只包含这个新的订单：

```
depot_r/test/integration/user_stories_test.rb
```

```
orders = Order.all
assert_equal 1, orders.size
order = orders[0]

assert_equal "Dave Thomas",    order.name
assert_equal "123 The Street",  order.address
assert_equal "dave@example.com", order.email
assert_equal "Check",          order.pay_type

assert_equal 1, order.line_items.size
line_item = order.line_items[0]
assert_equal ruby_book, line_item.product
```

最后，验证邮件本身的地址和主题行是正确的：

```
depot_r/test/integration/user_stories_test.rb
```

```
mail = ActionMailer::Base.deliveries.last
assert_equal ["dave@example.com"], mail.to
assert_equal 'Sam Ruby <depot@example.com>', mail[:from].value
assert_equal "Pragmatic Store Order Confirmation", mail.subject
```

这就是全部了。以下是集成测试的完整源代码：


```
depot_r/test/integration/user_stories_test.rb
```

```
require 'test_helper'
```

```
class UserStoriesTest < ActionDispatch::IntegrationTest
```

```
  fixtures :products
```

```
  # A user goes to the index page. They select a product, adding it to their
  # cart, and check out, filling in their details on the checkout form. When
  # they submit, an order is created containing their information, along with a
  # single line item corresponding to the product they added to their cart.
```

```
  test "buying a product" do
```

```
    LineItem.delete_all
```

```
    Order.delete_all
```

```
    ruby_book = products(:ruby)
```

```
    get "/"
```

```
    assert_response :success
```

```
    assert_template "index"
```

```
    xml_http_request :post, '/line_items', :product_id => ruby_book.id
```

```
    assert_response :success
```

```
    cart = Cart.find(session[:cart_id])
```

```
    assert_equal 1, cart.line_items.size
```

```
    assert_equal ruby_book, cart.line_items[0].product
```

```
    get "/orders/new"
```

```
    assert_response :success
```

```
    assert_template "new"
```

```
    post_via_redirect "/orders",
```

```
      :order => { :name      => "Dave Thomas",
                  :address   => "123 The Street",
                  :email     => "dave@example.com",
                  :pay_type  => "Check" }
```

```
    assert_response :success
```

```
    assert_template "index"
```

```
    cart = Cart.find(session[:cart_id])
```

```
    assert_equal 0, cart.line_items.size
```

```
    orders = Order.all
```

```
    assert_equal 1, orders.size
```

```
    order = orders[0]
```

```
    assert_equal "Dave Thomas", order.name
```

```
    assert_equal "123 The Street", order.address
```

```
    assert_equal "dave@example.com", order.email
```

```
    assert_equal "Check", order.pay_type
```

```
    assert_equal 1, order.line_items.size
```

```
    line_item = order.line_items[0]
```

```
    assert_equal ruby_book, line_item.product
```

```

mail = ActionMailer::Base.deliveries.last
assert_equal ["dave@example.com"], mail.to
assert_equal 'Sam Ruby <depot@example.com>', mail[:from].value
assert_equal "Pragmatic Store Order Confirmation", mail.subject
end
end

```

综上所述，单元测试、功能测试和集成测试在测试应用程序的各个方面显示出了足够的灵活性，可以独立使用，也可以与其他方法结合在一起进行测试。26.5 节会介绍在哪里可以找到插件(add-ons)，它们可以让测试工作到达一个新的层次，且允许书写纯文本格式的行为描述，这既可以让顾客读懂，又可以由系统自动验证。

说到顾客，是时候对这个迭代进行总结并看看商店中 Depot 的新功能。

13.3 本章小结

通过不多的代码和一些模板，我们做到了以下几点：

- 为了让 Rails 应用程序可以向外发送电子邮件，需要配置开发、测试和生产环境。
- 创建并修改了一个 mailer，它可以向预订我们产品的用户发送纯文本格式和 HTML 格式的确认邮件。
- 为生成的电子邮件创建功能测试，并且为整个下订单过程覆盖集成测试。

练习时间

可以自己尝试以下任务：

- 在订单表中添加一个 ship_date 列，当 OrdersController 更新该值时，将发送一个提醒。
- 当应用程序出现如 10.2 节中的错误时，让应用程序向管理员，也就是你自己，发送一封邮件。
- 为前面两项添加集成测试。

第 14 章

任务 I: 登录

在本章中，我们将学习：

- 向模型中添加虚拟属性
- 使用更多验证
- 不基于模型的表单编写
- 为会话添加认证
- 使用 rails 控制台
- 使用数据库事务
- 编写一个 Active Record 钩子

客户很高兴——在非常短的时间内做好了——一个基本的购物车，他可以展示给他的用户了。但他还想要改进一点。当前，所有人都可以访问管理功能。她想要增加一个基本的用户管理系统，以使得强制登录才可获得网站的管理员权限。

我们非常乐意这么做，因为它让我们有机会看到虚拟属性以及过滤器，并且可以让整个应用程序变得更加整洁。

在与客户交流之后，应用程序看起来并不需要是一个非常复杂的安全系统。所要做的只是根据用户名和密码进行识别。一旦识别成功，他就可以使用所有的管理员功能。

14.1 迭代 I1: 添加用户

首先创建一个存放管理员的用户名及其密码的模型和数据库表。密码不会以明文的格式存放，而是存放一个密码的 256 位的 SHA2 散列值和另外一个在密码学中叫做 salt 的值。

这么做出于以下几点考虑：即使数据库被破解了，从散列值也无法得出原始密码，这样就不能登录成功，也不能在其他地方使用，即使通常情况下用户总是在多个地方使用相同的密码。

```
depot> rails generate scaffold user \
  name:string hashed_password:string salt:string
```

像往常一样运行迁移：

```
depot> rake db:migrate
```

接下来充实一下用户模型。这有点复杂，因为从应用程序角度看，该模型要与明文的密码打交道，但同时还要维护数据库中的 salt 值和散列密码。让我们分步来看一下这个模型。首先是验证：

```
depot_r/app/models/user.rb
```

```
class User < ActiveRecord::Base
  validates :name, :presence => true, :uniqueness => true
```

```

validates :password, :confirmation => true
attr_accessor :password_confirmation
attr_reader :password

validate :password_must_be_present

private

def password_must_be_present
  errors.add(:password, "Missing password") unless hashed_password.present?
end
end

```

对于一个简单的模型来讲，这算一个比较复杂的验证。首先检查名字是否存在且唯一（即在数据库中不能有两个同名的用户）。然后出现了这个神秘的 `:confirmation => true` 参数。

你是否见过这种表单，它可以让你输入一个密码，然后在下面让你再输入一次？这样做可以验证你所输入的内容正确无误。Rails 可以自动验证两个密码是否一样。我们在后面会介绍这是怎么工作的。现在只要知道需要两个密码项，一个为实际的密码，另外一个为了确认。

最后，检查密码设置好了。但我们并不检查密码属性本身。为什么？因为它实际并不存在——至少在数据库中。相反，我们检查它的代理的存在，即散列密码。要理解这个原理，需要看看密码存储是怎么处理的。

现在创建一个散列密码，需要分三步。首先，创建一个独一无二的 salt 值（稍候我们会谈到 salt）。其次，将此 salt 值与明文密码相结合以生成一个字符串。最后，在结果上运行 SHA2 digest，这会返回一个 40 个字符的十六进制数字。将上述步骤写成一个公共类方法。（要记住在文件中引用 digest/sha2 库。在后面内容中介绍此库应该放在哪里。）

```

depot_r/app/models/user.rb

def User.encrypt_password(password, salt)
  Digest::SHA2.hexdigest(password + "wibble" + salt)
end

```

在密码学中，salt 字符串^①是随机的，它可让密码难以破解。在这里我们通过将一个随机数和用户对象的 id 连接而得到 salt 字符串。事实上 salt 是什么并不重要，只要它难以预测（例如使用时间作为 salt 的熵比使用随机字符要低）。将此 salt 值存放到模型对象的 salt 属性中。因为它应该是私有方法，在源代码中将它放置到 private 关键字之后：

```

depot_r/app/models/user.rb

def generate_salt
  self.salt = self.object_id.to_s + rand.to_s
end

```

在这段代码中有点儿微妙的东西我们从来没见过。注意 `self.salt = ...`，它强迫赋值使用 salt= 存取器方法——这相当于说“在当前对象中调用方法 salt=”。如果没有 self.，Ruby

① [http://en.wikipedia.org/wiki/Salt_\(cryptography\)](http://en.wikipedia.org/wiki/Salt_(cryptography))

会以为是对本地变量赋值，这样的话代码就没有效果了。

虽然直接使用实例变量也会得到正确的结果，但这样会将你束缚在一个不总是一致的表示上。属性 `salt/salt=` 是“官方”的对底层模型属性的接口，所以最好使用它们而不是实例变量。

从另一个角度来看是因为这些属性构成了类公共接口的一部分，那么该类就应该自我试用并也使用那些接口。如果在内部使用 `@xxx` 而在外部使用 `.xxx`，那么就会为某些错误匹配大开方便之门。

现在开始实现一段代码，无论何时将一个新的纯文本格式的密码存入用户对象，都会自动生成一个散列密码（它会自动保存在数据库中）。通过将纯文本格式的密码作为模型的一个虚拟属性来实现——它看起来像是应用程序的一个属性，但却不会在数据库中持续存在。

如果不需要创建散列密码，可以简单地通过使用 Ruby 的 `attr_accessor` 声明来实现这一点：

```
attr_accessor :password
```

`attr_accessor` 在幕后生成了两个存取器方法：一个叫做 `password` 的读方法和一个叫做 `password=` 的写方法。写方法名字以等号结尾意味着它可以被赋值。所以，使用标准的存取器可以简单地实现一些公共方法，并让写方法创建一个新的 `salt` 和设置散列密码：

```
depot_r/app/models/user.rb
```

```
def password=(password)
  @password = password

  if password.present?
    generate_salt
    self.hash_password = self.class.encrypt_password(password, salt)
  end
end
```

再次利用 `self` 来访问 `hashed_password`，但不使用 `self` 来访问 `password` 赋值函数，因为这样做只会重复递归调用。

还有最后一个修改。让我们实现一个公共类方法，如果调用者提供正确的用户名和密码，此方法返回该用户的对象。因为输入的密码是明文的，必须使用名字作为键值来访问用户记录，然后再次使用该记录中的 `salt` 值来生成散列密码。如果散列密码匹配，就返回该用户对象。使用这种方法就可以验证一个用户了。

```
depot_r/app/models/user.rb
```

```
def User.authenticate(name, password)
  if user = find_by_name(name)
    if user.hash_password == encrypt_password(password, user.salt)
      user
    end
  end
end
```

用户模型包含了许多代码，但它显示了模型如何承载大量的商业逻辑。在开始处理控制器之前让我们审视一下整个模型：

```
depot_r/app/models/user.rb
```

```
require 'digest/sha2'

class User < ActiveRecord::Base
  validates :name, :presence => true, :uniqueness => true
  validates :password, :confirmation => true
  attr_accessor :password_confirmation
  attr_reader :password
  validate :password_must_be_present

  def User.authenticate(name, password)
    if user = find_by_name(name)
      if user.hash_password == encrypt_password(password, user.salt)
        user
      end
    end
  end

  def User.encrypt_password(password, salt)
    Digest::SHA2.hexdigest(password + "wibble" + salt)
  end

  # 'password' is a virtual attribute
  def password=(password)
    @password = password

    if password.present?
      generate_salt
      self.hash_password = self.class.encrypt_password(password, salt)
    end
  end

  private

  def password_must_be_present
    errors.add(:password, "Missing password") unless hash_password.present?
  end

  def generate_salt
    self.salt = self.object_id.to_s + rand.to_s
  end
end
```

当这些代码完成之后，就可以在表单中显示和确认密码了，并且给定一个用户名和密码，可以认证该用户。

管理用户

除了设置好的模型和表之外，我们也已经生成了一些管理模型的脚手架。但是要使用到刚刚定义好的项，这些脚手架还需要一些调整。

首先从控制器开始。它定义了标准的方法：`index`、`show`、`new`、`edit`、`update` 和 `delete`。但是从用户的角度来讲，除了用户名和一个难以理解的散列（见 4.2.2 节）密码之外，并没有什么东西需要显示。所以在创建操作之后，让我们避免重定向到显示用户。而是重定向到用户索引并在闪存通知中加入用户名。

```
depot_r/app/controllers/users_controller.rb
```

```
def create
  @user = User.new(params[:user])

  respond_to do |format|
    if @user.save
      format.html { redirect_to(users_url,
        :notice => "User #{@user.name} was successfully created.") }
      format.xml { render :xml => @user,
        :status => :created, :location => @user }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @user.errors,
        :status => :unprocessable_entity }
    end
  end
end
```

对升级操作做相同的事：

```
depot_r/app/controllers/users_controller.rb
```

```
def update
  @user = User.find(params[:id])

  respond_to do |format|
    if @user.update_attributes(params[:user])
      format.html { redirect_to(users_url,
        :notice => "User #{@user.name} was successfully updated.") }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @user.errors,
        :status => :unprocessable_entity }
    end
  end
end
```

同时，对索引中返回的用户按 `name` 排序：

```
depot_r/app/controllers/users_controller.rb
```

```
def index
  @users = User.order(:name)

  respond_to do |format|
    format.html # index.html.erb
    format.xml { render :xml => @users }
  end
end
```

这样控制器上的修改就完成了，继续修改视图。当前视图并不显示通知信息，并且表格中包含了太多的信息。尤其是，表中显示了散列密码和 salt。让我们添加通知并删除这些项的 th 和 td 行：

```
depot_r/app/views/users/index.html.erb
<h1>Listing users</h1>
▶ <% if notice %>
▶ <p id="notice"><%= notice %></p>
▶ <% end %>

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

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

<br />

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

最后，需要更新表单，用来创建一个新用户和更新一个已有用户。首先，将散列密码和 salt 文本项替换成密码和密码确认项。接着添加 legend 和 fieldset 标签。最后用之前在样式表中定义的类型将输出包裹在 <div> 标签中。

```
depot_r/app/views/users/_form.html.erb
<div class="depot_form">

  <%= form_for @user do |f| %>
    <% if @user.errors.any? %>
      <div id="error_explanation">
        <h2><%= pluralize(@user.errors.count, "error") %>
          prohibited this user from being saved:</h2>
        <ul>
          <% @user.errors.full_messages.each do |msg| %>
            <li><%= msg %></li>
          <% end %>
        </ul>
      </div>
    <% end %>

    <div>
```

```

<fieldset>
<legend>Enter User Details</legend>

<div>
  <%= f.label :name %>:
  <%= f.text_field :name, :size => 40 %>
</div>

<div>
  <%= f.label :password, 'Password' %>:
  <%= f.password_field :password, :size => 40 %>
</div>

<div>
  <%= f.label :password_confirmation, 'Confirm' %>:
  <%= f.password_field :password_confirmation, :size => 40 %>
</div>

<div>
  <%= f.submit %>
</div>

</fieldset>
<% end %>

</div>

```

让我们来试试。浏览 <http://localhost:3000/users/new>。效果如图 14.1 所示。

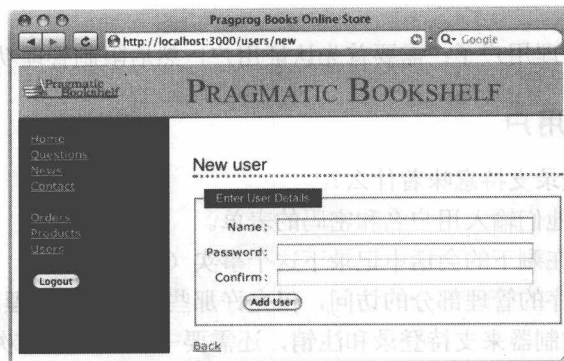


图 14.1 输入用户细节

在单击 Create User 之后，索引与一个祝贺的闪存通知一起显示。如果此时观察数据库，会发现已经存储了用户细节。（当然，行中的值会不同，因为 salt 值是随机的。）

```

depot> sqlite3 -line db/development.sqlite3 "select * from users"
      id = 1
    name = dave
hashed_password = a12b1dbb97d3843ee27626b2bb96447941887ded
      salt = 203333500.653238054564258
  created_at = 2011-02-11 21:40:19
  updated_at = 2011-02-11 21:40:19

```

就像我们之前做的一样，需要更新测试集来反映对验证和重定向所做的更改：

```
depot_s/test/functional/users_controller_test.rb

require 'test_helper'

class UsersControllerTest < ActionController::TestCase
  setup do
    ▶ @input_attributes = {
    ▶   :name           => "sam",
    ▶   :password       => "private",
    ▶   :password_confirmation => "private"
    ▶ }

    @user = users(:one)
  end
  #...
  test "should create user" do
    assert_difference('User.count') do
    ▶   post :create, :user => @input_attributes
    end

    ▶   assert_redirected_to users_path
    end
    #...
    test "should update user" do
    ▶   put :update, :id => @user.to_param, :user => @input_attributes
    ▶   assert_redirected_to users_path
    end
  end
end
```

至此，我们就可以管理用户了；需要首先认证用户，然后限制管理功能只能由管理员访问。

14.2 迭代 I2：认证用户

为商店管理员添加登录支持意味着什么？

- 需要提供一个可供他们输入用户名和密码的表单。
- 一旦登录了，需要在剩下的会话中记录下这个事实（直到注销）。
- 需要限制对应用程序的管理部分的访问，仅允许那些成功登录管理商店的人访问。

我们需要一个会话控制器来支持登录和注销，还需要一个控制器来欢迎管理员。

```
depot> rails generate controller sessions new create destroy
depot> rails generate controller admin index
```

SessionsController#create 行为需要在会话中记录管理员已经登录。让我们用键 :user_id 存储 User 对象的 id。登录代码如下所示：

```
depot_r/app/controllers/sessions_controller.rb

def create
  ▶ if user = User.authenticate(params[:name], params[:password])
  ▶   session[:user_id] = user.id
  ▶   redirect_to admin_url
  ▶ else
```

```

▶ redirect_to login_url, :alert => "Invalid user/password combination"
▶ end
end

```

还要做一件新的事情：使用一个并不直接与模型对象相关联的表单。要知道怎么做，看一下 sessions#new 动作的模板：

depot_r/app/views/sessions/new.html.erb

```

<div class="depot_form">
  <% if flash[:alert] %>
    <p id="notice"><%= flash[:alert] %></p>
  <% end %>

  <%= form_tag do %>
    <fieldset>
      <legend>Please Log In</legend>

      <div>
        <label for="name">Name:</label>
        <%= text_field_tag :name, params[:name] %>
      </div>
      <div>
        <label for="password">Password:</label>
        <%= password_field_tag :password, params[:password] %>
      </div>

      <div>
        <%= submit_tag "Login" %>
      </div>
    </fieldset>
  <% end %>
</div>

```

这个表单与我们之前看过的有所不同。它使用 form_tag，而不是 form_for，这样做就建立了一个普通的 HTML <form>。在那个表单之中，使用了 text_field_tag 和 password_field_tag 这两个帮助方法来创建 HTML <input> 标签。每个帮助方法接收两个参数。第一个是给此项的名字，第二个是此项的赋值。这种类型的表单让我们可以将 params 结构中的值直接与表单项关联起来——不需要模型对象。在这里，我们选择直接在表单里使用 params 对象。另外一种方法是让控制器设置实例变量。

见图 14.2。注意表单项中的值是如何利用 params 散列与控制器和视图通信的：视图从 params[:name] 中得到显示的值，当用户提交表单，该项新的值以同样的方式对控制器可用。

如果用户成功登录，在会话数据中存储下用户记录的 id。在会话期间我们会使用该值来作为管理员用户已经登录的标志。

如你所愿，注销的控制器行为非常简单：

depot_r/app/controllers/sessions_controller.rb

```

def destroy
  session[:user_id] = nil
  redirect_to store_url, :notice => "Logged out"
end

```

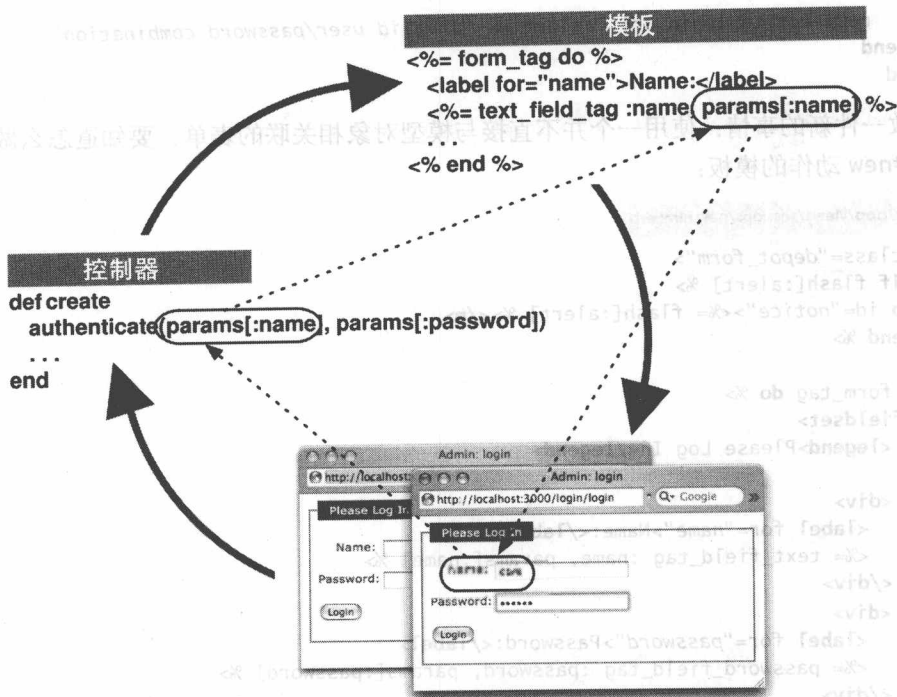


图 14.2 控制器、模板和浏览器之间的参数传递流程

最后，是时候增加索引页了，即管理员登录后的第一个页面。让我们使用它来显示在线商店中总的订单数。在目录 `app/views/admin` 中的 `index.html.erb` 文件中创建模板。（该模板使用 `pluralize` 帮助方法，它可以依据第一个参数的数目来生成 `order` 或 `orders` 字符串。）

```
depot_r/app/views/admin/index.html.erb
```

```
<h1>Welcome</h1>
```

```
It's <%= Time.now %>
```

```
We have <%= pluralize(@total_orders, "order") %>.
```

用 `index` 动作设置账户：

```
depot_r/app/controllers/admin_controller.rb
```

```
class AdminController < ApplicationController
```

```
def index
```

```
▶ @total_orders = Order.count
```

```
end
```

```
end
```

在使用它之前还有一件事情要做。尽管之前我们依靠脚手架生成器来创建模型和路由，然而这次我们直接生成了一个控制器，因为这个控制器没有后端数据库的模型。遗憾的是，没有了脚手架的指导，Rails 不知道用什么样的行为来响应 GET 请求，此 GET 请求又是用来响应 POST

请求的等。对于此控制器，需要通过编辑 config/routes.rb 文件来提供此信息：

```
depot_r/config/routes.rb

Depot::Application.routes.draw do
  ▶ get 'admin' => 'admin#index'

  ▶ controller :sessions do
  ▶   get 'login' => :new
  ▶   post 'login' => :create
  ▶   delete 'logout' => :destroy
  ▶ end

  resources :users

  resources :orders

  resources :line_items

  resources :carts

  get "store/index"
  resources :products do
    get :who_bought, :on => :member
  end

  # ...

  # You can have the root of your site routed with "root"
  # just remember to delete public/index.html.
  # root :to => "welcome#index"
  root :to => 'store#index', :as => 'store'

  # ...
end
```

之前已经涉及了这一点，在 8.1 节中我们增加了一条 root 语句。generate 命令向这个文件中添加的只不过是每个指定行为的通用 get 语句。可以（也应该）删除提供给 sessions/new、sessions/create 和 sessions/destroy 的路由。

对于 admin，我们要缩短用户要输入的 URL（通过移除 /index 部分）并把它映射到一个完整的行为上。对于会话行为，要完完全全地改变 URL（将 session/create 替换成 login）并且修剪要匹配的 HTTP 行为。注意到 login 被同时映射到 new 和 create 动作上，差别只在请求是 HTTP GET 还是 HTTP POST。

我们还会利用到一条捷径：将 session route 声明放在一个块中并传递给 controller 类方法。这样做的好处是不用输入那么多代码了，同时路由也变得更加便于阅读。20.1 节会完整地介绍此文件的用法。

这些路由就绪之后，就可以以管理员的身份来登录了，如图 14.3 所示。

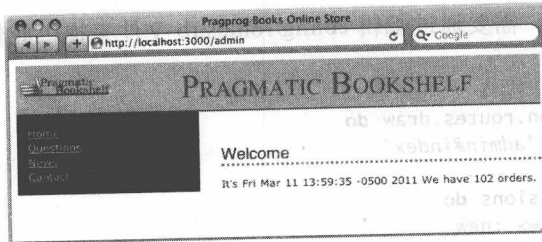


图 14.3 管理界面

还需要在会话控制器中对这些刚刚实现的东西进行功能测试:

```
depot_s/test/functional/sessions_controller_test.rb
require 'test_helper'

class SessionsControllerTest < ActionController::TestCase
  test "should get new" do
    get :new
    assert_response :success
  end

  ▶ test "should login" do
  ▶   dave = users(:one)
  ▶   post :create, :name => dave.name, :password => 'secret'
  ▶   assert_redirected_to admin_url
  ▶   assert_equal dave.id, session[:user_id]
  ▶ end
  ▶ test "should fail login" do
  ▶   dave = users(:one)
  ▶   post :create, :name => dave.name, :password => 'wrong'
  ▶   assert_redirected_to login_url
  ▶ end

  ▶ test "should logout" do
  ▶   delete :destroy
  ▶   assert_redirected_to store_url
  ▶ end
end
```

还要更新静态测试集以匹配:

```
depot_s/test/fixtures/users.yml
<% SALT = "NaCl" unless defined?(SALT) %>

one:
  name: dave
  hashed_password: <%= User.encrypt_password('secret', SALT) %>
  salt: <%= SALT %>

two:
  name: MyString
  hashed_password: MyString
  salt: MyString
```

注意，在静态测试中使用动态计算的值，特别是 `hashed_password` 的值。虽然 `salt` 值需要是随机的，但是为了测试，这里仅使用一个简单的值。

我们将进度给客户看，但他指出还没有控制管理员页面的访问（毕竟，这是这个练习的重点）。

14.3 迭代 I3: 限制访问

我们想要防止那些不具备管理员权限的人访问网站的管理员页面。利用 Rails 过滤器的帮助，这一点非常容易实现。

Rails 过滤器允许你拦截行为方法的调用，在它们被调用或返回后添加你自己的处理。在这里，使用 `before filter` 来拦截对管理控制器中行为的调用。拦截者能检查 `session[:user_id]`。如果该项被设置了并且对应了数据库中的一个用户，应用程序就知道一个管理员登录了，调用可以继续。如果没有设置，拦截者可以重定向，在这里可以重定向到登录页面。

应该将这个方法放在哪里呢？它可以直接放在 `admin` 控制器中，但是基于某些马上就会提到的原因，把它放到 `ApplicationController` 中，也就是所有控制器的父类。它在 `app/controllers` 文件夹下的 `application_controller.rb` 中。同时注意，还要限制访问此方法。这可以避免它作为一个行为暴露给终端用户。

```
depot_r/app/controllers/application_controller.rb
```

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

```
  before_filter :authorize
```

```
  protect_from_forgery
```

```
  private
```

```
    def current_cart
```

```
      Cart.find(session[:cart_id])
```

```
    rescue ActiveRecord::RecordNotFound
```

```
      cart = Cart.create
```

```
      session[:cart_id] = cart.id
```

```
      cart
```

```
    end
```

```
    # ...
```

```
  protected
```

```
    def authorize
```

```
      unless User.find_by_id(session[:user_id])
```

```
        redirect_to login_url, :notice => "Please log in"
```

```
      end
```

```
    end
```

```
  end
```

`before_filter` 那一行导致 `authorization` 于应用程序中每个行为之前被调用。

然而这么做有些过犹不及。我们刚刚将商店的访问只给了管理员。这并不好。

可以回到之前，仅修改那些特别需要验证的方法。这种方式叫做黑名单，但是很容易因为疏忽而出错。一个更好的方式是“白名单”，即仅列出那些不需要验证的方法或控制器。通过在

StoreController 中插入一个 skip_before_filter 调用，可以做到这一点：

```
depot_r/app/controllers/store_controller.rb
```

```
class StoreController < ApplicationController
  ▶ skip_before_filter :authorize
```

同时对 SessionsController 类：

```
depot_r/app/controllers/sessions_controller.rb
```

```
class SessionsController < ApplicationController
  ▶ skip_before_filter :authorize
```

还没有完成；现在要允许人们创建、更新和删除购物车：

```
depot_r/app/controllers/carts_controller.rb
```

```
class CartsController < ApplicationController
  ▶ skip_before_filter :authorize, :only => [:create, :update, :destroy]
```

创建商品项目：

```
depot_r/app/controllers/line_items_controller.rb
```

```
class LineItemsController < ApplicationController
  ▶ skip_before_filter :authorize, :only => :create
  ▶
```

并且创建订单（包括对 new 表单的访问）：

```
depot_r/app/controllers/orders_controller.rb
```

```
class OrdersController < ApplicationController
  ▶ skip_before_filter :authorize, :only => [:new, :create]
  ▶
```

验证逻辑就绪后，就可以访问 <http://localhost:3000/products> 了。过滤器方法在链接到产品列表的过程中会进行解析，并且取而代之，向用户显示登录页面。

遗憾的是，该更改会让大多数功能测试失效，因为许多操作现在都被重定向到登录页面而不是完成相应的功能了。幸好可以通过在 test_helper 中创建一个 setup 方法来全局性地解决这个问题。我们还会定义一些帮助方法来 login_as 和 login_in 一个用户。

```
depot_s/test/test_helper.rb
```

```
class ActiveSupport::TestCase
  # ...
```

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

```
  def login_as(user)
    session[:user_id] = users(user).id
  end
```

```
  def logout
    session.delete :user_id
  end
```

```
def setup
  login_as :one if defined? session
end
end
```

注意, 仅当定义了 `session`, `setup` 方法才会调用 `login_as`。这样做可以阻止在那些不调用控制器的测试中执行 `login`。

我们把这些给客户看, 他回报一个大大的微笑和请求: 可以增加一个侧边栏, 并把指向用户和商品管理的链接放到里面吗? 还有, 可以增加列出和删除管理员用户的功能吗? 当然可以!

14.4 迭代 I4: 增加侧边栏, 更多管理

首先在侧边栏中增加各式管理功能的链接, 并且仅当 `session` 中有 `:user_id` 时才显示它们:

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

```
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "scaffold" %>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
      <% if @cart %>
        <%= hidden_div_if(@cart.line_items.empty?, :id => "cart") do %>
          <%= render @cart %>
        <% end %>
      <% end %>

      <a href="http://www....">Home</a><br />
      <a href="http://www....faq">Questions</a><br />
      <a href="http://www....news">News</a><br />
      <a href="http://www....contact">Contact</a><br />

      <% if session[:user_id] %>
        <br />
        <%= link_to 'Orders', orders_path %><br />
        <%= link_to 'Products', products_path %><br />
        <%= link_to 'Users', users_path %><br />
      <% end %>
    </div>
  </div>
</body>
</html>
```

```

▶      <%= button_to 'Logout', logout_path, :method => :delete %>
▶      <% end %>
      </div>
      <div id="main">
        <%= yield %>
      </div>
    </div>
  </body>
</html>

```

所有的东西都渐渐成型。我们可以登录，并且通过单击侧边栏中的链接看见用户列表。让我们检查一下还有什么东西需要改进。

若是删除最后一个管理员

用户列表界面看起来如图 14.4 所示。通过单击 dave 旁边的 Destroy（销毁）链接来删除该用户。毫无疑问，用户被移除了。但意外的是，登录界面随后显示出来。我们刚刚删除了系统中唯一的用户。当收到下一个请求时，验证失败了，应用程序拒绝我们进入。必须重新登录才能使用管理功能。

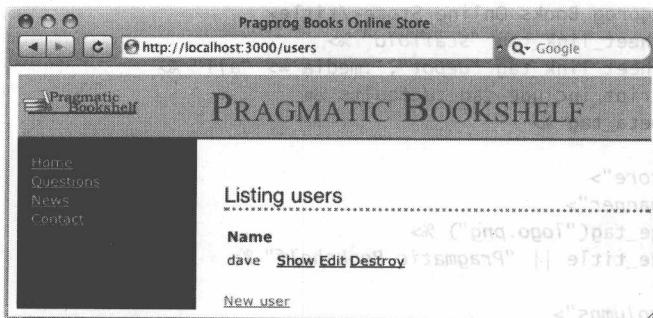


图 14.4 列出用户

但现在的问题有点尴尬：因为数据库中没有任何管理员用户，所以无法登录。

幸运的是，可以快速地通过命令行来向数据库中增加一个用户。如果调用命令 `rails console`，Rails 会调用 Ruby 的 `irb` 功能，但仅在此 Rails 应用程序的范围内。这意味着可以通过输入 Ruby 语句来与应用程序代码交互，并以此来查看它返回的值。

通过这种方法可以直接调用用户模型，让它向数据库中增加一个用户：

```

depot> rails console
Loading development environment.
>> User.create(:name => 'dave', :password => 'secret',
               :password_confirmation => 'secret')
=> #<User:0x2933060 @attributes={...} ...>
>> User.count
=> 1

```

>> 是提示符。在第一个提示符后，调用 `User` 类来创建新用户，第二个提示符后，再次调

用来显示数据库中确实存在一个用户。在每个输入命令之后，`rails console` 会显示代码的返回值（在第一个提示符后显示的是模型对象，第二个后面是用户数量）。

这样恐慌就结束了。现在可以重新登录应用程序。但如何才能防止这样的情况再次发生呢？有几种方法。比如可以实现防止删除自己的用户账号的代码。理论上说这种做法行不通，A 用户可以在 B 用户删除 A 的同时删除 B。因此，试一下另外一种方法。在数据库事务内部删除用户。如果在删除用户之后数据库中没有任何用户了，就将该事务回滚，恢复刚刚删除的用户。

使用 Active Record 钩子方法可以做到这一点。之前已经介绍过其中的一个方法：Active Record 调用 `validate` 钩子来验证对象的状态。事实上 Active Record 定义了大约 20 多个钩子方法，每个都在对象生命周期的特定点调用。这里使用 `after_destroy` 钩子，它在 SQL `delete` 执行之后调用。如果方法名是公共可见的，它会很方便地在与 `delete` 相同的事务中调用，因此，如果它引起了一个异常，事务就会被回滚。钩子方法如下所示：

```
depot_s/app/models/user.rb

after_destroy :ensure_an_admin_remains
```

```
def ensure_an_admin_remains
  if User.count.zero?
    raise "Can't delete last user"
  end
end
```

这里的重点是使用异常来指明删除用户时的错误。这个异常有两个意图。第一，因为它于事务中抛出，所以会导致一个自动回滚。如果 `users` 表在删除后为空，通过抛出这个异常可以撤销该删除并恢复用户。

第二，异常信号将错误通知控制器，在控制器中可以使用 `begin/end` 块来处理它，并将该错误在闪存中报告给用户。如果只想中止这个事务而不想将此异常报告，那么就使用 `ActiveRecord::Rollback` 异常，因为它是唯一一个不会通过 `ActiveRecord::Base.transaction` 的异常。

```
depot_s/app/controllers/users_controller.rb

def destroy
  @user = User.find(params[:id])
  begin
    @user.destroy
    flash[:notice] = "User #{@user.name} deleted"
  rescue Exception => e
    flash[:notice] = e.message
  end

  respond_to do |format|
    format.html { redirect_to(users_url) }
    format.xml { head :ok }
  end
end
```

上述代码还有一个潜在的时机问题——如果时机正确，两个管理员将最后的两个用户删除的

可能性依然存在。要消除这个问题需要更高深的数据库技术，它超出了本书讨论的范围。

事实上，在本章中介绍的登录系统还比较基础。当今大多数应用程序都使用插件来实现这个功能。许多插件都提供现成的解决方案，不仅比这里展示的验证逻辑更全面，而且通常使用起来不需要花费很多代码和精力。参见 26.5 节中的几个例子。

14.5 本章小结

在本次迭代工作中，我们完成了以下几点：

- 创建了代表纯文本格式密码的虚拟属性，并且无论明文密码何时更新，都会创建散列密码。
- 通过使用 before 过滤器调用 authorize 方法来控制管理员功能的访问。
- 学习了如何使用 rails console 直接与模型相交交互（并解决了删除最后一个用户之后的窘境）。
- 学习了如何使用事务防止删除最后一个用户。

练习时间

可以自由尝试以下任务：

- 如果系统是在一台新的机器上刚刚安装的，因为数据库中没有任何管理员，所以没有管理员能登录。但如果没有管理员可以登录，就不能创建管理员用户。请修改代码，当数据库中没有任何管理员时，任何用户名都可登录（即允许快速创建管理员）。
- 练习使用 rails console。试一试创建产品、订单和商品项目。当保存模型对象时看看返回的值——当验证失败时会看到返回 false。通过查看以下错误来找到原因：

```
>> prd = Product.new
=> #<Product id: nil, title: nil, description: nil, image_url:
nil, created_at: nil, updated_at: nil, price:
#<BigDecimal:246aa1c,'0.0',4(8)>>
>> prd.save
=> false
>> prd.errors.full_messages
=> ["Image url must be a URL for a GIF, JPG, or PNG image",
"Image url can't be blank", "Price should be at least 0.01",
"Title can't be blank", "Description can't be blank"]
```

（你可以在 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 上面找到提示。）

第 15 章

任务 J: 国际化

在本章中，我们将学习：

- 本地化模板
- 关于 I18n 数据库设计的考虑

现在已经拥有了一个基本运作的购物车，由于客户公司对新兴市场的大举拓展，仅仅有英语语言的网站已无法满足要求，他们开始询问在网站中使用其他语言的可能性。除非顾客能用自己熟悉的语言看得懂网站内容，否则他们不会轻易在该网站上购物，这样就会让客户损失利益。不能让这样的事发生。

首要问题是：我们都不是专业的翻译人员。客户一再保证，无需担忧此事，因为这部分工作将外包出去，而所要关心的只是实现翻译功能。另外，眼下也不用考虑管理页面，因为所有的管理员都说英语，只要专注于商店就行了。

负担是减轻了一些，但这仍然是一项艰巨的任务。需要定义一种方式，让用户可以选择语言，还必须提供翻译，并且得基于这些翻译来调整视图。尽管这样，我们还是可以胜任这项任务的，带着一点点高中时学习西班牙语的残留印象，开始工作。

Joe 问

如果只使用一种语言，还需要阅读这一章吗？

简单来说：不需要。实际上，许多 Rails 的应用程序都用于小型或者由相同地域的人组成的团队，根本不需要翻译。话虽如此，几乎每个确实觉得自己需要翻译功能的人，都会同意翻译最好早点进行。因此，除非十分肯定自己永远不会用到翻译，否则我们还是建议，至少懂得哪些是需要的，这样才能做出明智的决定。

15.1 迭代 J1: 选择语言环境

先从创建一个新的配置文件开始，这个文件概括了以下信息：哪些语言环境是可用的，默认的语言环境是什么。

```
depot_s/config/initializers/i18n.rb
```

```
#encoding: utf-8
```

```
I18n.default_locale = :en
```

```
LANGUAGES = [
```

```
  ['English', 'en'],
```

```
  ['Español', 'es'],
```

```
]
```

这段代码做了两件事。

第一件事，用 `i18n` 模型设置默认的语言环境。虽然 `i18n` 是个奇怪的名字，但它一定比每次都要打 `internationalization` 方便得多，究其根源，这么命名是因为 `internationalization` 由 `i` 开始，以 `n` 结尾，中间有 18 个字母。

第二件事，这段代码定义了显示名称和语言环境名称间的关联列表。可惜现阶段只有美式键盘可用，而 `español` 里有一个字符是不能用这个键盘直接输入的。对此，不同的操作系统有不同的解决办法，通常最简单的办法是从网页上复制正确的文本，然后粘贴。如果真是这么做的，一定要确保已将编辑器配置为 `UTF-8`。同时，这里选择使用 `HTML` 中与西班牙文“带波浪号的 `n`”相对应的字符。如果不做其他事情，这个标记会自动显示。但是当调用 `html_safe` 的时候，会通知 `Rails`，告诉它“把这个字符串解释为包含 `HTML`”是安全的。

要让这个配置在 `Rails` 中生效，需要重启服务器。

由于每个翻译过的页面都将会会有一个英语版本 (`en`) 和一个西班牙语版本 (`es`) (只现在是这样，以后会加入更多东西)，因此把 `i18n` 包含到 `URL` 里是合理的。语言环境计划放在最前面，设为可选，并且将默认配置设为当前语言环境，按照次序，它将默认为英语。

要实现这个巧妙的计划，可以从修改 `config/routes.rb` 开始：

`depot_s/config/routes.rb`

```
Depot::Application.routes.draw do
  get 'admin' => 'admin#index'
  controller :sessions do
    get 'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end
  ► scope '(:locale)' do
    resources :users
    resources :orders
    resources :line_items
    resources :carts
    resources :products do
      get :who_bought, :on => :member
    end
    root :to => 'store#index', :as => 'store'
  ► end
end
```

上面这段代码所做的事情是，在对 `:local` 的范围 (`scope`) 声明中嵌套资源和根地址声明。另外，这里 `:local` 在括号里，表明它是可选的。注意，这里没有选择把管理和会话功能放在这个范围中，因为不打算在这个时候就翻译它们。

这表明，两个 `http://localhost:3000/` 都将使用默认的语言环境，也就是“英语”，因此这两个页面都会被引导到与 `http://localhost:3000/en` 完全一样的路径。`http://localhost:3000/es` 将路由到同一个控制器和操作，但它应该产生不同的语言环境设置。

要完成这项工作，需要创建一个 `before_filter`，并且设置 `default_url_options`。在所有控制器的公共基类 (也就是 `ApplicationController`) 内修改是比较合理的，在这里可以一

箭双雕:

```

depot_s/app/controllers/application_controller.rb

class ApplicationController < ActionController::Base
  before_filter :set_i18n_locale_from_params
  # ...
  protected
  def set_i18n_locale_from_params
    if params[:locale]
      if I18n.available_locales.include?(params[:locale].to_sym)
        I18n.locale = params[:locale]
      else
        flash.now[:notice] =
          "#{params[:locale]} translation not available"
        logger.error flash.now[:notice]
      end
    end
  end

  def default_url_options
    { :locale => I18n.locale }
  end
end

```

`set_i18n_locale_from_params` 做的事情和它的名字描述大概一致: 它从参数中取值来设置对话框的语言环境——当然, 前提是在参数中有语言环境的值; 否则, 不会去修改它。在碰到错误时, 会同时给用户和管理员提供错误信息。

`default_url_options` 的工作也和名字描述相似, 在其中它提供了一个 URL 选项的散列, 无论是否设置这些选项, 都认为其是存在的。此处给 `:locale` 参数设定了一个值, 当一个未设定语言环境的页面视图尝试建立一个链接, 并指向已设定语言环境的页面时, 这是有必要的。将在使用中很快看到这种情况。

做好这些后, 更新后页面可以看到如图 15.1 所示的结果。

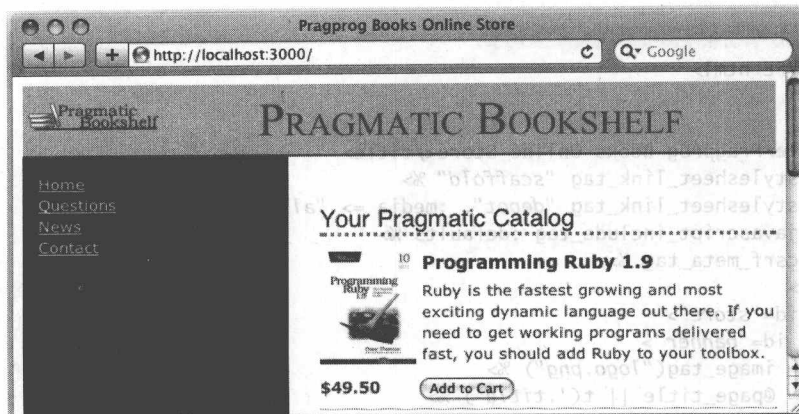


图 15.1 英语版的首页

现在, 网站根地址和以 /en 开头的页面都可以访问英语版的页面了。另外, 显示屏上有一条消息, 说明目前西班牙语的翻译还不可用 (见图 15.2), 与此同时, 日志中也会留下信息, 写明文件未找到。也许还并不完美, 但已经有进步了。

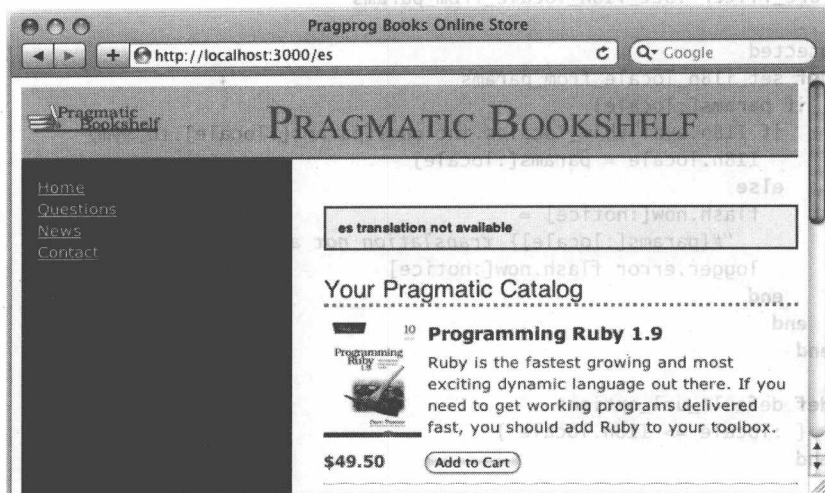


图 15.2 翻译不可用

15.2 迭代 J2: 翻译在线商店页面

至此, 是时候开始提供翻译文本了。现在将从版面设计入手, 因为它非常直观。将调用 `I18n.translate` 来替代所需翻译的文本。这个方法不仅有简单的别名 `I18n.t`, 而且还有一个称为 `t` 的帮助程序。

翻译函数的参数名称是唯一的, 且以点前缀方式命名。可以选择任何喜欢的名字, 但如果使用提供的 `t` 帮助函数, 以点开始的名称会首先按模板名展开。既然如此, 可以这样来实现:

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

```
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "scaffold" %>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || t('title') %>
  </div>
  <div id="columns">
```

```

<div id="side">
  <% if @cart %>
    <%= hidden_div_if(@cart.line_items.empty?, :id => "cart") do %>
      <%= render @cart %>
    <% end %>
  <% end %>
  <a href="http://www...."><%= t('.home') %></a><br />
  <a href="http://www..../faq"><%= t('.questions') %></a><br />
  <a href="http://www..../news"><%= t('.news') %></a><br />
  <a href="http://www..../contact"><%= t('.contact') %></a><br />

  <% if session[:user_id] %>
    <br />
    <%= link_to 'Orders', orders_path %><br />
    <%= link_to 'Products', products_path %><br />
    <%= link_to 'Users', users_path %><br />
    <br />
    <%= button_to 'Logout', logout_path, :method => :delete %>
  <% end %>
</div>
<div id="main">
  <%= yield %>
</div>
</div>
</body>
</html>

```

页面 layouts/application.html.erb 的英语映射将扩展为 en.layouts.application。下面是相关的语言环境文件:

```
depot_s/config/locales/en.yml
```

en:

```

layouts:
  application:
    title: "Pragmatic Bookshelf"
    home: "Home"
    questions: "Questions"
    news: "News"
    contact: "Contact"

```

然后是西班牙语的:

```
depot_s/config/locales/es.yml
```

es:

```

layouts:
  application:
    title: "Publicaciones de Pragmatic"
    home: "Inicio"
    questions: "Preguntas"
    news: "Noticias"
    contact: "Contacto"

```

其文件格式与配置数据库时用的也一样，也是 YAML。YAML 仅仅由缩进式名称和值构成，这里的缩进与命名时创建的结构匹配。

需要重启网络服务器，Rails 才能识别出新的 YAML 文件。

至此，可以在浏览器中看到实际翻译好的文本了，如图 15.3 所示。

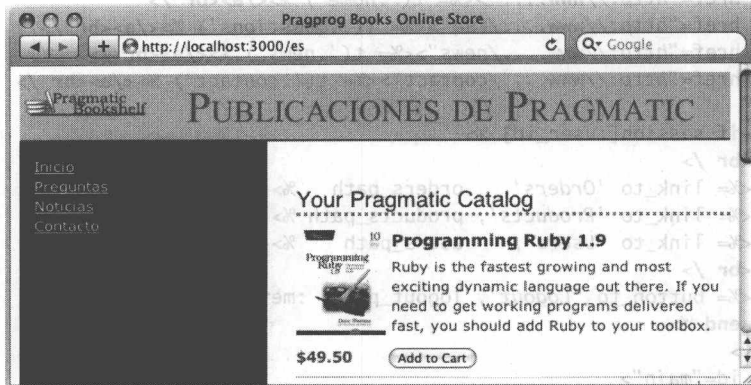


图 15.3 最初的一步：翻译好的标题和侧边栏

下一步需要更新的是主标题和 Add to Cart 按钮。在商店的索引模板中可以找到它们。

depot_s/app/views/store/index.html.erb

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

▶ <h1><%= t('.title_html') %></h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">
      <span class="price"><%= number_to_currency(product.price) %></span>
      ▶ <%= button_to t('.add_html'), line_items_path(:product_id => product),
        :remote => true %>
    </div>
  </div>
<% end %>
```

下面是语言环境文件的相关更新，首先是英语的：

depot_s/config/locales/en.yml

```
en:

  store:
    index:
      title_html: "Your Pragmatic Catalog"
      add_html: "Add to Cart"
```

接着是西班牙语的:

```
depot_s/config/locales/es.yml
```

```
es:
```

```
store:
```

```
index:
```

```
  title_html: "Su Catálogo de Pragmatic"
```

```
  add_html: "Añadir al Carrito"
```

注意, 由于 `title_html` 和 `add_html` 都以字符 `_html` 结尾, 因而可以自由地把没有出现在键盘上的字符作为 HTML 的实体名来对待。如果不这样命名翻译键, 最终在网页上看到的是标记本身。这又是 Rails 的另一个约定, 它让编程生活更轻松。Rails 也可以处理那些包含 html 的名称 (换句话说, 含字符串 `.html`), 把它们作为 HTML 的键名。

通过刷新浏览器窗口的页面, 可以看到结果, 如图 15.4 所示。

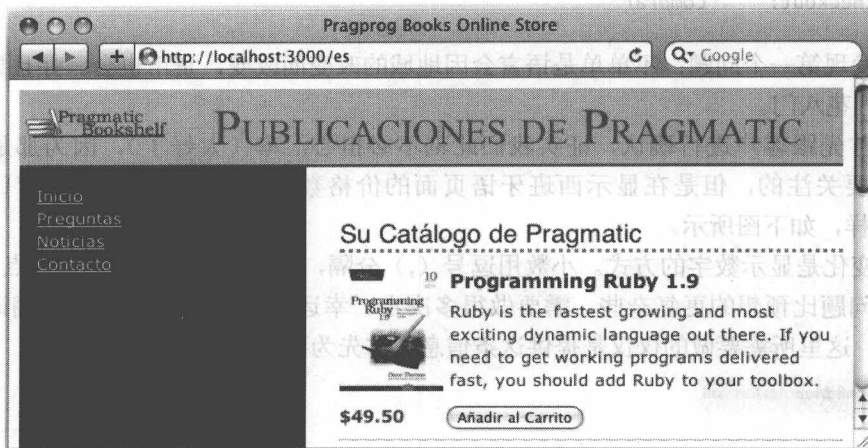


图 15.4 翻译好的标题和按钮

我们信心倍增, 接着进入购物车的部分:

```
depot_s/app/views/carts/_cart.html.erb
```

```
▶ <div class="cart_title"><%= t('title') %></div>
<table>
  <%= render(cart.line_items) %>

  <tr class="total_line">
    <td colspan="2">Total</td>
    <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>
</table>

▶ <%= button_to t('checkout'), new_order_path, :method => :get %>
▶ <%= button_to t('empty'), cart, :method => :delete,
  :confirm => 'Are you sure?' %>
```


和前面一样，语言环境的翻译：

```
depot_s/config/locales/en.yml
```

```
en:
```

```
  carts:
```

```
    cart:
```

```
      title: "Your Cart"
```

```
      empty: "Empty cart"
```

```
      checkout: "Checkout"
```

```
depot_s/config/locales/es.yml
```

```
es:
```

```
  carts:
```

```
    cart:
```

```
      title: "Carrito de la Compra"
```

```
      empty: "Vaciar Carrito"
```

```
      checkout: "Comprar"
```

此时，发现第一个问题：不单单是语言会因地域的改变而改变，货币也会。并且显示数字的习惯方法也五花八门。

因此，首先跟客户进行确认，证实我们此刻不必担心汇率（太好了），因为那是信用卡或电汇公司需要关注的，但是在显示西班牙语页面的价格数值时，得在其后标注“USD”或者“\$US”的字样，如下图所示。

另一个变化是显示数字的方式。小数用逗号（,）分隔，而千位的分隔符是一个点（.）。

货币的问题比预想的更复杂些，需要做很多决定。幸运的是，Rails 知道要在翻译文件中寻找这条信息，这里所需要做的仅仅是提供这条信息。首先为英语版本 en：

```
depot_s/config/locales/en.yml
```

```
en:
```

```
  number:
```

```
    currency:
```

```
      format:
```

```
        unit: "$"
```

```
        precision: 2
```

```
        separator: "."
```

```
        delimiter: ","
```

```
        format: "%u%n"
```

接下来是西班牙语的版本 es：

```
depot_s/config/locales/es.yml
```

```
es:
```

```
  number:
```

```
    currency:
```

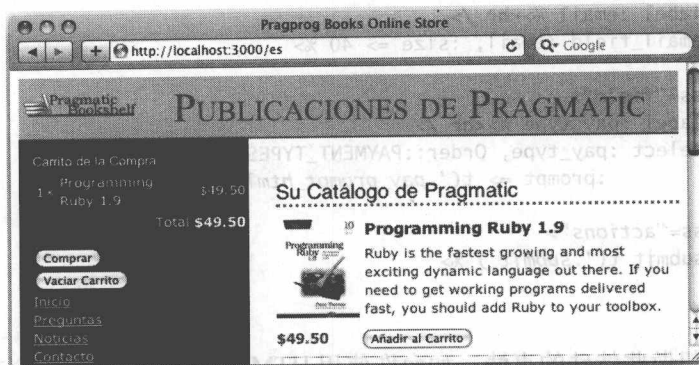
```
      format:
```

```
        unit: "$US"
```

```
        precision: 2
```

```
separator: ",",
delimiter: "."
format:      "%n    %;u"
```

这里已经为 `number.currency.format` 明确规定了单位、精度、分隔符和小数点。一切都不言自明。格式则有一点复杂：`%` 是数字的占位符；` ` 代表一个不间断空格的字符，可以防止这个值在多行中分开，`%u` 是单位的占位符，如下图所示。



15.3 迭代 J3: 翻译结账页面

现在我们感觉已经进入最后阶段了。新的订单页面如下：

```
depot_s/app/views/orders/new.html.erb
```

```
<div class="depot_form">
  <fieldset>
    <legend><%= t('.legend') %></legend>
    <%= render 'form' %>
  </fieldset>
</div>
```

这个页面使用的表单是：

```
depot_s/app/views/orders/_form.html.erb
```

```
<%= form_for(@order) do |f| %>
  <%= if @order.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@order.errors.count, "error") %>
        prohibited this order from being saved:</h2>

      <ul>
        <%= @order.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <%= end %>
      </ul>
    </div>
  <%= end %>
```

```

<div class="field">
  <%= f.label :name %><br />
  <%= f.text_field :name, :size => 40 %>
</div>
<div class="field">
  <%= f.label :address, t('.address_html') %><br />
  <%= f.text_area :address, :rows => 3, :cols => 40 %>
</div>
<div class="field">
  <%= f.label :email %><br />
  <%= f.email_field :email, :size => 40 %>
</div>
<div class="field">
  <%= f.label :pay_type %><br />
  <%= f.select :pay_type, Order::PAYMENT_TYPES,
    :prompt => t('.pay_prompt_html') %>
</div>
<div class="actions">
  <%= f.submit t('.submit') %>
</div>
<% end %>

```

请注意,除非想做些特殊的事情,如允许使用 HTML 的实体,通常不会在标签中显性地调用 I18n 函数。以下是相关的语言环境定义:

```
depot_s/config/locales/en.yml
```

en:

```

orders:
  new:
    legend:      "Please Enter Your Details"
  form:
    name:        "Name"
    address_html: "Address"
    email:       "E-mail"
    pay_type:    "Pay with"
    pay_prompt_html: "Select a payment method"
    submit:      "Place Order"

```

```
depot_s/config/locales/es.yml
```

es:

```

orders:
  new:
    legend:      "Por favor, introduzca sus datos"
  form:
    name:        "Nombre"
    address_html: "Dirección"
    email:       "E-mail"
    pay_type:    "Pagar con"
    pay_prompt_html: "Seleccione un método de pago"
    submit:      "Realizar Pedido"

```

请看图 15.5 中完成的表单。

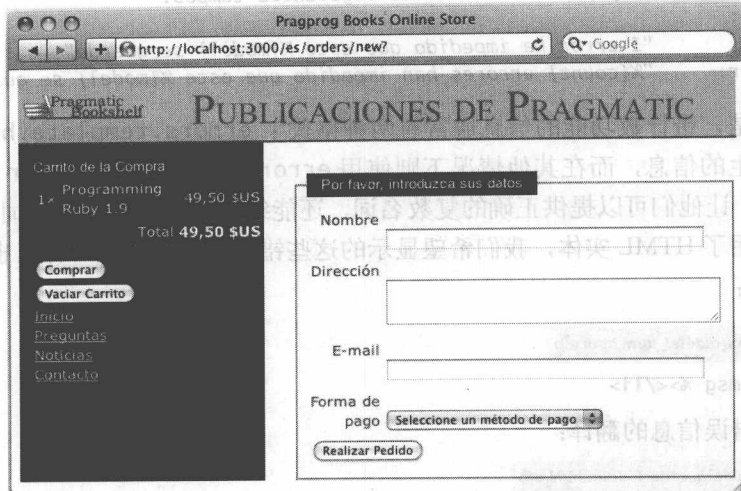


图 15.5 准备付账（西班牙语版）

一切都看起来很不错，直到我们贸然地单击了 Realizar Pedido 按钮，然后看到下一页（如图 15.6 所示）。Active Record 模块产生的这些错误信息也是可以翻译的，所需要做的是提供翻译：

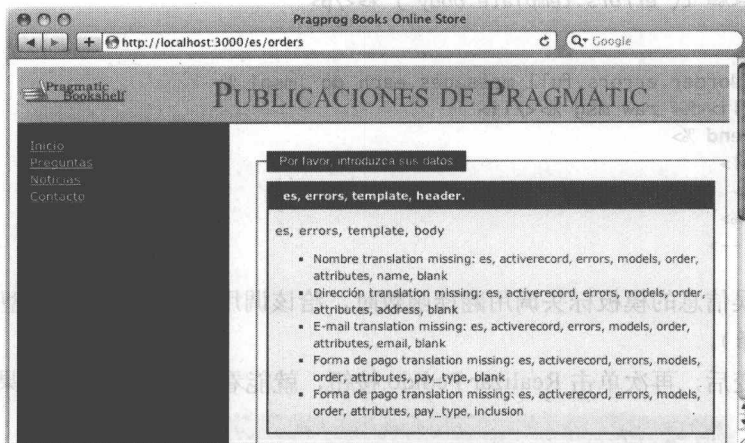


图 15.6 翻译缺失

```
depot_s/config/locales/es.yml
```

es:

```
activerecord:
  errors:
    messages:
      inclusion: "no est&acute; incluido en la lista"
      blank: "no puede quedar en blanco"
  errors:
```

```

template:
  body:      "Hay problemas con los siguientes campos:"
  header:
    one:     "1 error ha impedido que este %{model} se guarde"
    other:   "%{count} errores han impedido que este %{model} se guarde"

```

值得一提的是,带计数功能的信息通常有两种格式: `errors.template.header.one` 是在有一个错误时产生的信息,而在其他情况下则使用 `errors.template.header.other`。这给翻译人员带来机会,让他们可以提供正确的复数名词,还能给名词匹配正确的动词。

由于再次使用了 HTML 实体,我们希望显示的这些错误信息和现在一样(也就是 Rails 中说的原始值 (raw)):

```
depot_t/app/views/orders/_form.html.erb
```

```
<li><%= raw msg %></li>
```

还需要调整错误信息的翻译:

```
depot_t/app/views/orders/_form.html.erb
```

```

<%= form_for(@order) do |f| %>
  <% if @order.errors.any? %>
    <div id="error_explanation">
      <h2><%= t('errors.template.header', :count=>@order.errors.size,
        :model=>t('activerecord.models.order')) %>.</h2>
      <p><%= t('errors.template.body') %></p>

      <ul>
        <% @order.errors.full_messages.each do |msg| %>
          <li><%= raw msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
<!-- ... -->

```

注意,为错误信息的模板标头调用翻译函数时,给该调用传递了总数和模型名称(这个模型本身已实现了翻译)。

做了这些改变后,再次单击 Realizar Pedido 按钮,就能看到图 15.7 中的成果了。

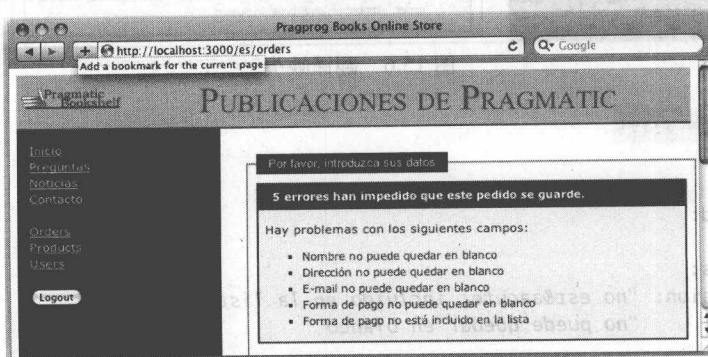


图 15.7 在西班牙语句中的英语名词

看起来虽然好一些了，但模型的名称和属性还暴露在界面上。这在英语界面中没有问题，因为所选的名字在英语界面中也能用。这里需要给每个语言版本都提供翻译。

同样，在 YAML 文件中修改：

```
depot_t/config/locales/es.yml
```

es:

```
activerecord:
  models:
    order: "pedido"
  attributes:
    order:
      address: "Direcci&ocute;n"
      name: "Nombre"
      email: "E-mail"
      pay_type: "Forma de pago"
```

在这里要特别指出，给英语界面提供翻译是没有必要的，因为那些信息在 Rails 中是内置的。我们很高兴地看到模型和属性的名称已经翻译好了（见图 15.8 所示）。填好表格，提交订单，然后收到“Thank you for your order”（感谢惠顾）的信息。

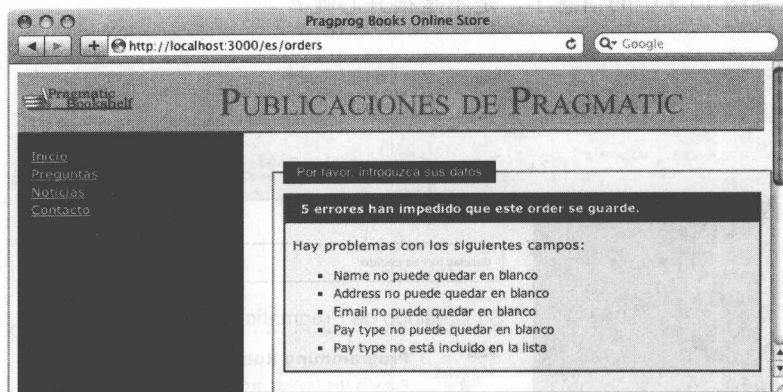


图 15.8 现在模型名称也翻译好了

接下来更新闪存消息：

```
depot_t/app/controllers/orders_controller.rb
```

```
def create
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(current_cart)

  respond_to do |format|
    if @order.save
      Cart.destroy(session[:cart_id])
      session[:cart_id] = nil
      Notifier.order_received(@order).deliver
      format.html { redirect_to(store_url, :notice =>
        I18n.t('.thanks')) }
    end
  end
end
```

```

format.xml { render :xml => @order, :status => :created,
              :location => @order }
else
  format.html { render :action => "new" }
  format.xml { render :xml => @order.errors,
                      :status => :unprocessable_entity }
end
end
end
end

```

最后，提供翻译：

```
depot_1/config/locales/en.yml
```

en:

```
  thanks:      "Thank you for your order"
```

```
depot_1/config/locales/es.yml
```

es:

```
  thanks:      "Gracias por su pedido"
```

现在就能看到图 15.9 中的信息了，是不是很开心呢？

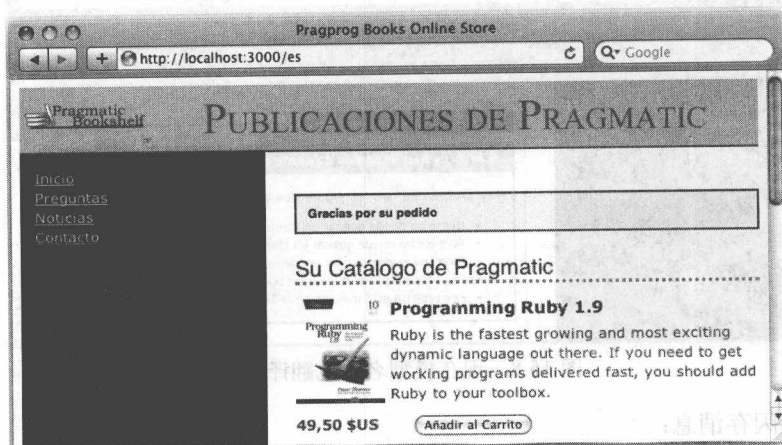


图 15.9 西班牙语的“感谢惠顾”

15.4 迭代 J4：添加语言环境的切换器

虽然任务已经完成了，但还需要更多地宣传其可用性。发现在版面的右上角还有一些空白的区域，于是马上在 `image_tag` 前添加一个表单：

```
depot_1/app/views/layouts/application.html.erb
```

```

▶ <%= form_tag store_path, :class => 'locale' do %>
  <%= select_tag 'set_locale',
    options_for_select(LANGUAGES, I18n.locale.to_s),

```



```

      :onchange => 'this.form.submit()' %>
    <%= submit_tag 'submit' %>
    <%= javascript_tag "$$(''.locale input').each(Element.hide)" %>
  ▶ <% end %>

```

在提交表单时, `form_tag` 指定商店的路径作为重新显示的页面。而 `class` 属性允许表单和某些 CSS 联系起来。

`select_tag` 用来为这个表单定义输入栏, 即语言环境。它是一个可选列表, 可供选择的内容基于配置文件中设置的 `LANGUAGES` 数组, 默认的选项是当前的语言环境 (也可以由 `I18n` 模型提供)。这里还设置了一个 `onchange` 事件句柄, 用来在值有所改变的时候随时提交这个表单。虽然只有当启用 JavaScript 时 `onchange` 事件句柄才运行, 但它是有用处的。

接下来加入 `submit_tag` 来应对浏览器没有启用 JavaScript 的情况。为了处理这样的情况: JavaScript 是可用的而提交按钮是多余的, 这里增加了一点 JavaScript 来隐藏语言环境表单中的每一个输入标签, 即使已知只有一个标签。

下一步是修改商店控制器, 如果使用了 `:set_locale` 表单, 这个控制器将对给定的语言环境重新定位商店的路径:

```

depot_t/app/controllers/store_controller.rb

def index
  ▶ if params[:set_locale]
  ▶   redirect_to store_path(:locale => params[:set_locale])
  ▶ else
  ▶   @products = Product.all
  ▶   @cart = current_cart
  ▶ end
end

```

最后, 加一点 CSS:

```

depot_t/public/stylesheets/depot.css

.locale {
  float: right;
  margin: -0.25em 0.1em;
}

```

真实的选择器效果, 请看图 15.10。现在通过单击鼠标就可以切换语言了。

至此, 可以用两种语言下订单了, 于是我们决定下一步转向实际部署。但因为已经忙碌了一整天, 是时候放下手中的工作放松放松了, 明天早上再开始部署部分的工作。

15.5 本章小结

在这个迭代里, 做了以下事情:

- 为应用程序设置了默认的语言环境, 并且为用户提供了一种方式, 让他可以选择候补的语言环境。
- 为文本输入框、货币数量、错误信息和模型名称创建了翻译文件。

- 为了翻译界面的文本部分，修改了版面和视图，让它们可以使用 `t` 帮助程序来调用 `i18n` 模块。

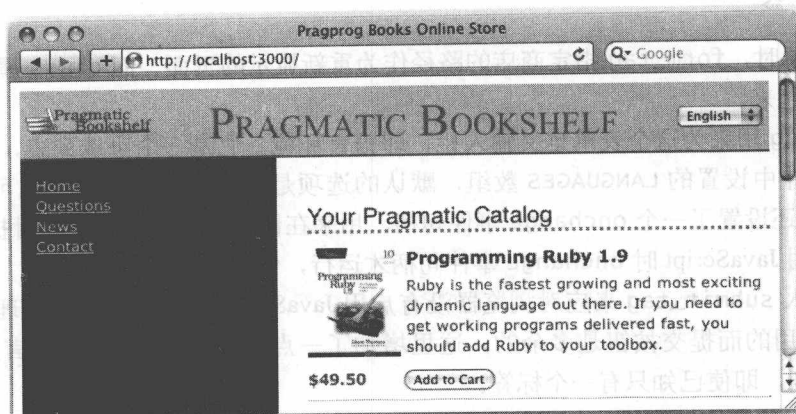


图 15.10 右上角的语言环境选择器

练习时间

可以自己尝试以下任务：

- 在产品数据库中增加一个语言环境字段，调整索引视图，让它只选择与语言环境匹配的产品。改变产品视图，以便可以查看、输入和修改这个新字段。在每个语言环境里输入一些产品，然后测试生成的应用程序。
- 确定当前美元和欧元之间的汇率，将显示本地化，当选中 `ES_es` 时，显示欧元及其对应的金额。
- 翻译下拉框中显示的 `Order::PAYMENT_TYPES`。得让选择值保持一致（它将被发送到服务器上）。只改变所显示出的类型。

（可以在这里 <http://www.pragprog.com/wikis/wiki/RailsPlayTime> 找到一些提示。）

第 16 章

任务 K：部署和产品环境

在本章中，我们将学习：

- 在产品 Web 服务器上运行应用程序
- 为 MySQL 配置数据库
- 用 Bundler 和 Git 进行版本管理
- 用 Capistrano 部署应用程序

部署是应用程序制作周期中令人高兴的标志性时刻。就是在这个时刻，我们把精心制作的代码上传到服务器上，供他人使用。就是在这个时刻，大家开始了庆祝。此后不久，《Wired》杂志将会报道该应用程序，然后我们在计算机奇才界将一夜成名。

然而现实并非如此，要完成流畅且可重复的应用程序部署，往往需要相当长时间的前期规划。

在本章结束的时候，设置将如图 16.1 所示。

Joe 问

可以将应用程序部署到微软的 Windows 系统中吗？

虽然可以把应用程序部署到 Windows 的环境中，但是大量的 Rails 工具和共享知识都假设其使用的环境是基于 UNIX 操作系统，如 Linux 或苹果操作系统（Mac OS X）。其中名为 Phusion Passenger 的工具就是这样，它得到了 Ruby on Rails 开发团队的强烈推荐，其内容会在本章中有所涵盖。

本章中描述的技术可以用于在 Windows 上进行开发，然后在 Linux 或苹果操作系统（Mac OS X）上部署的应用程序。

如果基础设施要求必须在 Windows 上部署，那么可以在《Deploying Rails Applications: A Step-by-Step Guide[ZT08]》一书的第 6 章中找到很好的建议。

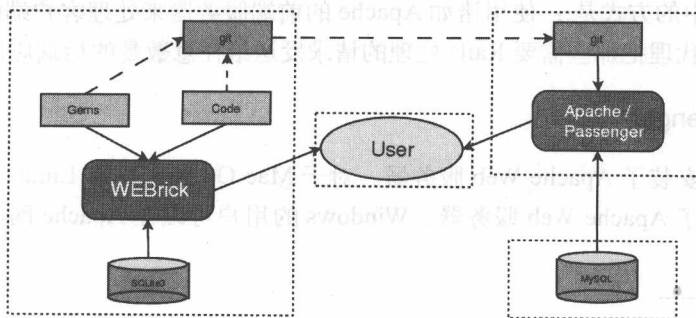


图 16.1 应用程序部署的路线

目前,尽管用户和 Web 服务器间的交互可以在另一台单独的机器上完成,但所有的工作仍然一直在同一台机器上进行。在图 16.1 中,用户的机器位于中心,WEBRick 网络服务器在左边。这个服务器会使用 SQLite 3、安装的各种 gem 以及应用程序的代码。在这个时候,代码可能已经放在了 Git 中,也可能没有;有两种方式可以把代码放到 Git 中,一种方式会在本章的最后介绍,而另一种方式则是利用正在使用的 gem。

Git 储存库在产品服务器上会被复制,这个产品服务器可以是另一台机器,但这不是必需的。这个服务器将会在装有 Apache httpd 和 Phusion Passenger 的机器上运行。而这部分代码会访问 MySQL 数据库,该数据库可能在第四台机器上。

Capistrano 是用来更新部署服务器的工具,它可以远程地、安全地、重复地在开发机器的舒适环境中完成更新。

这涉及许多可移动部件。

我们并不打算一步到位,而是把工作分为三个迭代。迭代 K1 让 Depot 应用程序运行起来,并且和 Apache、MySQL 以及 Passenger 一起运行——这是一个真正的高质量产品 Web 服务器环境。

Git、Bundler 和 Capistrano 会放在第二个迭代中。这些工具能够把开发活动从部署环境中分离出来。这意味着在完成时,要部署两次。但这仅仅是在第一次部署时需要做的,其目的只是想确保每个部件都在独立工作。开发活动和部署环境的独立性也允许可以在任何时候把重点放在一个较小的变量集合上,对那些可能会遇到的问题,这将简化解决问题的过程。

在第三个迭代中,将介绍各种管理和清除任务。这就开始吧。

16.1 迭代 K1: 用 Phusion Passenger 和 MySQL 部署

到目前为止,已经在本地机器上开发了一个 Rails 应用程序,在运行服务器时,也许已经使用了 WEBrick 或 Mongrel。大多数情况下,这没有问题。`rails server` 命令将挑选出最适当的方式来让该应用程序在开发模式上运行,端口为 3000。但是,部署后的 Rails 应用程序的工作原理有点不同。不能只开启一个单一的 Rails 服务器进程,然后让它完成所有工作。当然,也可以这么做,但结果会很不理想。因为 Rails 是单线程的,它一次只能处理一个请求。

然而,Web 是一个极度并发的环境。产品 Web 服务器,如 Apache、Lighttpd 和 Zeus 都可以同时处理多个请求(有时甚至几十或者几百个)。一个基于 Ruby 的单进程、单线程 Web 服务器可能跟不上这个速度。幸运的是,它不需要和产品网络服务器齐头并进。相反,把 Rails 应用程序部署到产品环境中的方式是:使用诸如 Apache 的前端服务器来处理客户端的请求。然后,用 Passenger 的 HTTP 代理把那些需要 Rails 处理的请求发送给任意数量的后端应用程序进程。

16.1.1 安装 Passenger

第一步是确保安装了 Apache Web 服务器。对于 Mac OS X 和许多 Linux 的用户来说,操作系统已经自带安装了 Apache Web 服务器。Windows 的用户可以在 Apache 网站上找到安装这个产品环境的说明^①。

① <http://httpd.apache.org/docs/2.2/platform/windows.html#inst>

下一步是安装 Passenger:

```
$ sudo gem install passenger
$ sudo passenger-install-apache2-module
```

如果没有满足必要的包依赖关系, 第二条命令会说明需要做什么。例如, 在 Ubuntu 10.04 (Lucid Lynx) Linux 上, 需要安装 build-essential、apache2-prefork-dev、libapr1-dev 和 libaprutil1-dev。如果发生这种情况, 按照提供的说明操作, 然后再试一次 Passenger 的安装命令。

一旦满足了包依赖关系, 这条命令会编译许多源代码, 并且更新配置文件。在这个过程中, 它会让用户更新 Apache 的配置。第一步是启用新建的模块, 然后在 Apache 的配置中添加新的行, 添加的内容如下(注意: Passenger 会给出要复制粘贴到配置中的确切内容, 因此请使用 Passenger 告知的那些内容, 而不是书中这些。另外, 为了符合页面大小, 书中将 LoadModule 一行对折了。而在输入时, 所有内容都要写成一行):

```
LoadModule passenger_module /opt/local/lib/ruby/gems/1.8/gems/passenger-2.2.11↵
/ext/apache2/mod_passenger.so
PassengerRoot /opt/local/lib/ruby/gems/1.8/gems/passenger-2.2.11
PassengerRuby /opt/local/bin/ruby
```

可以通过以下命令来确定 Apache 的配置文件的位置:

```
$ apachectl -V | grep SERVER_CONFIG_FILE
```

在一些系统中, 命令的名字是 apache2ctl, 而在另一些系统中是 httpd。不断尝试直到找到正确的命令。

如果想用相同的 Apache Web 服务器为多个应用程序服务, 需要先确认下面这行内容已经在配置文件里了:

```
NameVirtualHost *:80
```

如果配置文件中没有上述内容, 那么在包含 Listen 80 文本的前一行加上它。

16.1.2 在本地部署应用程序

接下来要部署应用程序。前面的步骤每个服务器只需要做一次, 而这一步却的确需要每个应用程序都做一次。用你的主机名称替换下面代码中 ServerName 后的内容:

```
<VirtualHost *:80>
  ServerName depot.yourhost.com
  DocumentRoot /home/rubys/work/depot/public/
</VirtualHost>
```

注意, 这儿的 DocumentRoot 设置为 Rails 应用程序的 public 目录。

修改好这些后, 为每个应用程序重复一遍这个 VirtualHost 代码块, 调整每块的 ServerName 和 DocumentRoot。还需要把公共目录标记为可读的。最终的版本应该和下面内容相似:

```
<VirtualHost *:80>
  ServerName depot.yourhost.com
  DocumentRoot /home/rubys/work/depot/public/

  <Directory /home/rubys/work/depot/public>
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

最后一步是重启 Apache Web 服务器。

```
$ sudo apachectl restart
```

现在需要配置客户端,以便它能把所选择的主机名映射到正确的机器上。可以在 `/etc/hosts` 文件中完成这件事。在安装 Windows 操作系统的机器上,可以在 `C:\windows\system32\drivers\etc\` 路径下找到这个文件。以管理员的身份打开该文件,才能编辑它。

典型的 `/etc/hosts` 行如下:

```
127.0.0.1 depot.yourhost.com
```

这就是所有工作了。现在可以用指定的主机(或者虚拟主机)访问该应用程序了。除非使用了另一个端口号,而不是 80,否则不必在 URL 中指定端口号。

下面是一些注意事项:

- 如果重启服务器时看到这样的信息“The address or port is invalid”(地址或端口无效),这表明 `NameVirtualHost` 行已经存在,也许在相同目录的另一个配置文件中。如果是这样,删除增加的 `NameVirtualHost` 行,因为这条指令只需要出现一次。
- 如果想在其他环境,而不是在产品环境中运行,可以在 Apache 配置的每个 `VirtualHost` 中包含一条 `RailsEnv` 指令:

```
RailsEnv development
```

- 通过在应用程序的 `tmp` 目录下创建名为 `restart.txt` 的文件,可以随时重启应用程序,而不用重启 Apache:

```
$ touch tmp/restart.txt
```

一旦服务器重启,将会删除这个文件。

- `passenger-install-apache2-module` 命令的输出会告知在哪里可以找到更多的文档。

16.1.3 使用 MySQL 数据库

SQLite 网站[⊖]的坦诚着实让人记忆犹新,它说明了这个数据库擅长什么,又不擅长什么。尤其在大容量、高并发且需要处理大型数据集合的网站上,不推荐使用 SQLite。然而,我们想开发的正是这样的网站。

⊖ <http://www.sqlite.org/whentouse.html>

有很多可以替代 SQLite 的选择，免费的和商业的都有。选择 MySQL 是因为它已经包含在操作系统（OS X）中了，并且通过 Linux 自带的包管理工具就可以使用，MySQL 网站[⊖]上提供了微软 Windows 的安装程序。

除了安装 MySQL 数据库，还需要在 Gemfile 中添加 mysql 的 gem:

```
depot_1/Gemfile
group :production do
  gem 'mysql'
end
```

把这个 gem 放到 production 组中，在开发或测试中运行时，就不会载入它。如果喜欢的话，可以把 sqlite3 gem（分别）放到 development 和 test 组中。

使用 bundle install 来安装 gem。可能需要为操作系统先定位并安装 MySQL 数据库开发文件。例如，在 Ubuntu 上，必须安装 libmysqlclient-dev。

可以使用 mysql 命令行客户端来创建数据库，或者倘若对诸如 phpmyadmin 或 CocoaMySQL 这样的工具更得心应手，尽管拿来使用：

```
depot> mysql -u root
mysql> CREATE DATABASE depot_production;
mysql> GRANT ALL PRIVILEGES ON depot_production.*
-> TO 'username'@'localhost' IDENTIFIED BY 'password';
mysql> EXIT;
```

如果选择了不同的数据库名称，记住它，因为将需要调整配置文件来匹配所选择的名称。现在来看看配置文件。

database.yml 包含有关数据库连接的信息，其中有三个部分，分别对应于开发、测试和产品环境数据库。当前的产品环境部分包括以下内容：

```
production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

用和下面相似的内容替换这个部分：

```
production:
  adapter: mysql
  encoding: utf8
  reconnect: false
  database: depot_production
  pool: 5
  username: username
  password: password
  host: localhost
```

必要时改变账号、密码和数据库字段。

⊖ <http://www.mysql.com/downloads/mysql/#downloads>

16.1.4 加载数据库

接下来实施迁移:

```
depot> rake db:setup RAILS_ENV="production"
```

有两种情况可能发生。如果一切都设置正确, 会看到如下输出:

```
-- create_table("carts", {:force=>true})
-> 0.1722s
-- create_table("line_items", {:force=>true})
-> 0.1255s
-- create_table("orders", {:force=>true})
-> 0.1171s
-- create_table("products", {:force=>true})
-> 0.1172s
-- create_table("users", {:force=>true})
-> 0.1255s
-- initialize_schema_migrations_table()
-> 0.0006s
-- assume_migrated_upto_version(20110211000008, "db/migrate")
-> 0.0008s
```

如果设置不正确, 就会看到某种错误, 不要惊慌! 有可能是很简单的配置问题。可以试试下面这些方法:

- 在文件 database.yml 中的 production: 部分, 核对给数据库定义的名称。它应该和所创建的数据库名称一致 (使用 mysqladmin 或其他数据库管理工具)。
- 检查文件 database.yml 中的用户名和密码, 它们要与上一节中创建的内容匹配。
- 检查数据库服务器是否运行正常。
- 检查能否用命令行连接数据库服务器。如果使用 MySQL, 可以用以下命令来测试:

```
depot> mysql depot_production
mysql>
```

- 如果可以从命令行连接, 试试看能否创建一个虚拟表 (dummy table)。(这可以测试该数据库用户是否有足够的权限来存取数据库。)

```
mysql> create table dummy(i int);
mysql> drop table dummy;
```

- 如果可以从命令行成功创建数据库表, 但是 rake db:migrate 失败, 那么再确认一次 database.yml 文件。如果该文件中存在 socket: 指令, 试着在每行开头处用 # 号把它们注释了。
- 如果看到 “No such file or directory...” (没有这样的文件或目录) 的错误, 并且该错误中的文件名为 mysql.sock, 那表明 Ruby MySQL 库找不到 MySQL 数据库。在安装数据库之前安装了 Ruby MySQL 库, 或者安装该库所使用的二进制版本对 MySQL 套接字文件的位置做出了错误的假设, 都有可能引发这个错误。要修复这个错误, 最好的办法是重装 Ruby MySQL 库。如果不想重装, 那么再次检查 database.yml 文件中的 socket: 一行, 确保它包含了到该系统 MySQL 套接字的正确路径。

- 如果遇到 “Mysql not loaded” (MySQL 没有加载) 的错误, 说明所用的 Ruby MySQL 库版本太旧。Rails 至少需要 2.5 版本。
 - 有些读者也反馈他们碰到了这样的错误信息 “Client does not support authentication protocol requested by server; consider upgrading MySQL client”。要解决 MySQL 安装版本和存取库之间不兼容的问题, 请按照这里的说明操作: <http://dev.mysql.com/doc/mysql/en/old-client.htm>, 并且发出一条 MySQL 命令, 如 `set password for 'some_user'@'some_host' = OLD_PASSWORD('newpwd');`。
 - 倘若是在 Windows 的 Cygwin 上使用 MySQL, 也许会在指定 localhost 主机时遇到问题。这时可以试试看 127.0.0.1。
 - 最后一点, 有可能会碰到 database.yml 文件格式的问题。YAML 库对 Tab 键出奇地敏感。如果文件中包含 Tab 键, 就会有问题。(这时回想起当初选择 Ruby 而不是 Python, 是因为不喜欢 Python 别有含义的空格键, 对吧?)
- 不断地运行 `rake db:setup` 直到所有的配置都正确为止。
- 如果这一切听起来有些吓人, 别担心。在现实中, 数据库连接在大多数时候都很有效。一旦 Rails 可以和数据库沟通, 就无需再担心这部分的问题了。

现在一切都设置好并且运行起来了。和运行单一用户时没有什么不同。只有当大量的并发用户或大型数据库时, 差异才会变得明显。

下一步是把开发从产品环境机器中分离出来。

16.2 迭代 K2: 用 Capistrano 远程部署

假设有一家大型商店, 它拥有一群可管理的专用服务器, 以便可以确保这些服务器和预想的一样, 运行着相同版本的必要软件。对于不太重要的需求, 共享服务器就可以做到这一点, 但是需要格外留心的是: 安装的软件版本可能不会总是和开发机器上安装的版本匹配。

别担心, 我们会帮你度过这一关。

16.2.1 准备好部署服务器

在开发过程中把软件放在版本管理之下真的是一个非常棒的主意, 但在部署时不这么做却是十足的蛮干——因此, 选择用来管理部署的软件 (也就是 Capistrano) 要求必须要有版本管理。

有很多软件配置管理 (Software Configuration Management, SCM) 系统可供选择。例如, Subversion 就是一个特别好的 SCM。但是如果还没有选择任何一个, 可以试试 Git, 它容易设置, 并且不要求单独的服务器进程。下面的例子都是基于 Git 的, 但如果选择了其他的 SCM 系统, 也不用担心。Capistrano 不太介意你选了哪个, 只要有了就好。

第一步是在部署服务器可以访问到的机器上创建一个空的储存库。实际上, 如果只有一个部署服务器, 完全有理由把它同时作为 Git 服务器和部署服务器。现在, 登录到那个服务器, 输入如下命令:

```
$ mkdir -p ~/git/depot.git
$ cd ~/git/depot.git
$ git --bare init
```

要明白的下一件事情是，即使 SCM 服务器和 Web 服务器是在同一台机器上，Capistrano 也会把 SCM 软件当做远程的来访问。可以通过生成公共密钥来让这个过程更顺畅（如果没有密钥的话），然后当访问自己的服务器时，用它来给出权限：

```
$ test -e ~/.ssh/id_dsa.pub || ssh-keygen -t dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys2
```

采用 SSH 连接到自己服务器的方法，可以测试公共密钥是否正常工作。除此之外，这么做还将更新 known_hosts 文件。

到这里，应该注意另外两件事情。第一件事情非常琐碎：Capistrano 将在应用程序目录名和 Rails 子目录之间插入名为 current 的目录，其中包括 public 子目录。这意味着，如果自行管理自己的服务器，那么必须调整在 httpd.conf 文件中 DocumentRoot 的设置，或者在共享主机的控制面板中修改。

```
DocumentRoot /home/rubys/work/depot/current/public/
```

至此服务器的预备工作已经结束。从这里开始，将在开发机器上完成所有工作。

16.2.2 把应用程序放到版本管理下

第一步是更新 Gemfile，以表明正在使用 Capistrano。

```
depot_1/Gemfile
source 'http://rubygems.org'

gem 'rails', '3.0.5'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'
group :production do
  gem 'mysql'
end

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

# Deploy with Capistrano
▶ gem 'capistrano'

# To use debugger (ruby-debug for Ruby 1.8.7+, ruby-debug19 for Ruby 1.9.2+)
# gem 'ruby-debug'
# gem 'ruby-debug19', :require => 'ruby-debug'

# Bundle the extra gems:
```

```
# gem 'bj'
# gem 'nokogiri'
# gem 'sqlite3-ruby', :require => 'sqlite3'
# gem 'aws-s3', :require => 'aws/s3'

gem 'will_paginate', '>= 3.0.pre'

# Bundle gems for the local environment. Make sure to
# put test-only gems in this group so their generators
# and rake tasks are available in development mode:
# group :development, :test do
#   gem 'webrat'
# end
```

现在使用 `bundle install` 来安装 Capistrano。在 12.3 节中, 曾使用这个指令来安装 `will_paginate` 的 gem。

如果还没有把应用程序放在配置管理之下, 现在来做这件事情:

```
$ cd your_application_directory
$ git init
$ git add .
$ git commit -m "initial commit"
```

下一步是可选的, 但倘若没有完全控制部署服务器, 抑或要管理许多部署服务器, 这一步可能会很有帮助。这里将使用 Bundler 的另一个功能, 即 `pack` 命令。这条命令所做的是把所依赖的软件版本放到储存库中。

```
$ bundle pack
$ git add Gemfile.lock vendor/cache
$ git commit -m "bundle gems"
```

在 25.3 节中会介绍更多 Bundler 的功能。

从这里开始, 把所有代码放到服务器上将是一件再简单不过的事情了。

```
$ git remote add origin ssh://user@host/~/.git/depot.git
$ git push origin master
```

经过这几步操作, 就已经获得了正在部署内容的控制权。你控制着那些提交到本地储存库中的内容, 也控制着什么时候把它们清理出服务器。下一步, 控制把代码放到产品环境中。

16.2.3 远程部署应用程序

之前在服务器上本地部署了应用程序。现在要进行另一个部署, 这一次将远程操作。

预备工作已经完成。现在代码在 SCM 服务器上, 并且可以用应用服务器访问。和前面一样, 这两个服务器是否相同并不重要, 重要的是它们扮演什么角色。

要让 Capistrano 发挥神奇功效, 需要给这个项目添加一些必要的文件, 执行下面的命令:

```
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

从输出可以看出, Capistrano 设置了两个文件。第一个文件名为 Capfile, 它是 Capistrano 对 Rakefile 的模拟。今后并不需要修改它。第二个文件是 config/deploy.rb, 它包含部署应用程序所需要的方法。Capistrano 会提供该文件的简单版本, 而下面这个是有的一些复杂的版本, 可以下载它, 然后把它当做起始点:

```
depot_t/config/deploy.rb
```

```
# be sure to change these
set :user, 'rubys'
set :domain, 'depot.pragprog.com'
set :application, 'depot'

# file paths
set :repository, "#{user}@#{domain}:git/#{application}.git"
set :deploy_to, "/home/#{user}/#{domain}"

# distribute your applications across servers (the instructions below put them
# all on the same server, defined above as 'domain', adjust as necessary)
role :app, domain
role :web, domain
role :db, domain, :primary => true

# you might need to set this if you aren't seeing password prompts
# default_run_options[:pty] = true

# As Capistrano executes in a non-interactive mode and therefore doesn't cause
# any of your shell profile scripts to be run, the following might be needed
# if (for example) you have locally installed gems or applications. Note:
# this needs to contain the full values for the variables set, not simply
# the deltas.
# default_environment['PATH']='<your paths>:/usr/local/bin:/usr/bin:/bin'
# default_environment['GEM_PATH']='<your paths>:/usr/lib/ruby/gems/1.8'

# miscellaneous options
set :deploy_via, :remote_cache
set :scm, 'git'
set :branch, 'master'
set :scm_verbose, true
set :use_sudo, false

namespace :deploy do
  desc "cause Passenger to initiate a restart"
  task :restart do
    run "touch #{current_path}/tmp/restart.txt"
  end

  desc "reload the database with seed data"
  task :seed do
    run "cd #{current_path}; rake db:seed RAILS_ENV=production"
  end
end

after "deploy:update_code", :bundle_install
desc "install the necessary prerequisites"
```

```
task :bundle_install, :roles => :app do
  run "cd #{release_path} && bundle install"
end
```

必须改变几个属性来匹配该应用程序。肯定需要修改的是：`:user`、`:domain` 和 `:application`。`:repository` 要改为先前放置 Git 文件的地方，`:deploy_to` 可能需要转而与这样的路径匹配，在这个路径下，Apache 可以找到该应用程序的 `config/public` 目录。

`default_run_options` 和 `default_environment` 只有在有特殊问题的时候使用。所提供的“miscellaneous options”是基于 Git 的。之后定义了两个任务。一个任务告诉 Capistrano 如何重启 Passenger。另一个任务利用先前放在 Git 储存库中的副本，安装 gem。只要认为合适，可以随意调整这些任务。

第一次部署应用程序时，还必须执行一个额外的步骤，用来为服务器上的部署设立基本的目录结构。

```
$ cap deploy:setup
```

在执行这条命令时，Capistrano 会提示服务器密码。如果提示失败或者登录不成功，也许需要取消 `deploy.rb` 文件中对 `default_run_options` 的注释，然后再试一次。一旦连接成功，它会生成必要的目录。做好这一步后，可以测试配置，看看还有没有其他问题。

```
$ cap deploy:check
```

和前面一样，可能需要在 `deploy.rb` 中取消 `default_environment` 的注释，并且修改它。重复这条命令，直到不再有错误，并且解决所有可能找到的问题。

现在已经准备好开始部署了。由于已经完成了所有必要的准备工作，并且检查了结果，部署应该很顺利：

```
$ cap deploy:migrations
```

至此，正式的工作才刚刚开始。

16.2.4 冲洗，洗净，重复

一旦走到这一步，服务器就随时准备好迎接要部署的应用程序的新版本了。所需要做的是检查储存库中的变化，然后重新部署。目前，有两个 Capistrano 文件尚未添加到版本管理中。尽管应用服务器并不需要这两个文件，但还是可以用它们来测试部署过程：

```
$ git add .
$ git commit -m "add cap files"
$ git push
$ cap deploy
```

前三条命令将更新 SCM 服务器。随着对 Git 的使用越来越熟悉，你可能想要管理得更细致，如什么时候添加了哪些文件，也许想在部署前提交多个改变，也许还有更多。只有最后一条命令会更新应用、Web 和数据库服务器。

如果出于某种原因，需要及时回滚到上一个应用程序版本，可以使用：

```
$ cap deploy:rollback
```

现在有一个完全部署好的应用程序，而且可以根据需要部署，用以更新在服务器上运行的代码。每次部署该应用程序，都会在服务器上生成一个新版本，另外，会更新一些符号链接，并且重启 Passenger 进程。

16.3 迭代 K3：检查部署的应用程序

一旦发布了应用程序，毫无疑问，必须不断地检查该程序是否运行正常。主要有两种方式检查：第一种方式是监视各种日志文件，这些日志是前端的 Web 服务器和 Apache 服务器运行该应用程序的输出文件。第二种方式是用 rails console 连接该应用程序。

16.3.1 查看日志文件

对应用程序发出请求时，要快速查看该应用程序正在发生的事情，可以使用 tail 命令来检查日志文件。最令人关注的信息当属这个应用程序自己生成的日志文件。即使 Apache 正在运行多个应用程序，每个应用程序的日志输出都会放在那个应用程序的 production.log 文件里。

假设要把这个应用程序部署到上一节提到的路径下，可以用如下方法查看日志文件：

```
# On your server
$ cd /home/rubys/work/depot/
$ tail -f log/production.log
```

有时需要低一级别的信息，如该应用程序中的数据正发生着什么？在这种情况下，就该打开最有用的实时服务器调试工具。

16.3.2 使用命令行界面来查看实时的应用程序

至此已经在应用程序的模型类中创建了大量的功能。当然，创建这些功能是为了让该应用程序的控制器使用。但是，也可以直接与这些功能进行交互。通往这个世界的门户是 rails console 脚本语言。可以在服务器上运行下列代码：

```
# On your server
$ cd /home/rubys/work/depot/
$ rails console production
Loading production environment.
irb(main):001:0> p = Product.find_by_title("Pragmatic Version Control")
=> #<Product:0x24797b4 @attributes={. . .}
irb(main):002:0> p.price = 32.95
=> 32.95
irb(main):003:0> p.save
=> true
```

一旦开启了命令行界面对话，就可以在模型上仔细检查所有种类的方法。可以创建、检查和删除记录。在某种程度上，就像该应用程序有个根命令行界面。

如果把应用程序放到产品环境中，需要关心一些琐事来保证这个应用程序平稳地运行。虽然系统不会自动照看这些琐事，但幸运的是，可以让它们自动化。

16.3.3 处理日志文件

当应用程序运行时，会不断地在自己的日志文件中添加数据。最后，这些日志文件可以变得相当巨大。为了克服这个问题，大部分日志的解决办法是通过创建随时间增长的渐进式日志文件集合，用来滚动增加日志文件。这将把日志文件拆分为可管理的组块，可以存档这些组块，甚至可以过一段时间后删除它们。

Logger 类用于支持滚动增加日志。需要指定想要多少（或者想要更频繁地得到）日志文件和每个日志文件的大小，只要在 `config/environments/production.rb` 文件中加入如下的一行内容：

```
config.logger = Logger.new(config.paths.log.first, 'daily')
```

加入的内容也有可能像这样：

```
require 'active_support/core_ext/numeric/bytes'
config.logger = Logger.new(config.paths.log.first, 10, 10.megabytes)
```

注意，在后一种情况中，需要有一个显性的 `active_support` 需求。这是因为我们在该应用程序中包含 Active Support 库之前的早期初始化过程中使用了这一声明。事实上，Rails 提供的其中一个配置选项根本不包括 Active Support 库：

```
config.active_support.bare = true
```

除此之外，可以把日志文件指向本机系统的日志文件：

```
config.logger = SyslogLogger.new
```

在 <http://rubyonrails.org/deploy> 上可以找到更多的方法。

16.3.4 开始发行，超越自我

一旦设置完初始的部署，就准备好完成该应用程序的开发，并且把它发行到产品环境中了。这里将很有可能要设置额外的部署服务器。在第一次部署中学习的东西涉及许多信息，这些信息是关于应该如何安排以后的部署的。例如，将有可能发现 Rails 是系统中比较慢的配件之一（更多的请求时间将花在 Rails 而不是数据库或文件系统上的等待）。这表明扩大规模的方式是增加机器来分担 Rails 的负载。

但是，也有可能发现一个请求把大部分时间花在数据库中。如果是这种情况，则应该想想如何优化数据库活动。也许可以改变存取数据的方法，或者需要自定义地精心制作一些 SQL，用以代替默认的 Active Record 行为。

有一点是可以确定的：每个应用程序在它的整个生命周期中将需要一系列不同的微调。最重要的事情是随着时间的推移留心倾听该应用程序，发现什么是要做的。所做的工作在发行这个应用程序时并没有完成，事实上，一切才刚刚开始。

当把该应用程序第一次部署到产品环境中，也就是任务开始之初时，已经完成了 Depot 应用程序的勘察。在重述本章要点之后，回顾一下在极少的几行代码后完成了什么任务。

16.4 本章小结

在本章中涉及了许多领域。我们把单一用户本地运行的代码，从开发机器放到了另一台机器上，在那儿，代码在一个不同的 Web 服务器运行，存取不同的数据库，甚至可能运行在不同的操作系统中。

为了完成这个任务，使用了许多不同的产品环境：

- 安装并配置了 Phusion Passenger 和 Apache httpd，一个产品质量的 Web 服务器。
- 安装并配置了 MySQL，一个产品质量的数据库服务器。
- 使用 Bundler 和 Git 对该应用程序的包依赖关系进行版本管理。
- 安装并配置了 Capistrano，用以自信地、重复地部署该应用程序。

练习时间

可以自己尝试以下任务：

- 如果有多个开发者合作开发，也许把数据库配置的详细信息（可能包含密码）放在配置管理系统中会令人有些不安。要解决这个问题，可以复制完整的 database.yml 文件，把它放到 shared 目录下，然后写一个任务，指示 Capistrano 在每次部署时将这个文件复制到 current 目录下。
- 尝试一种模式变化，为商品项目增加价格。要确保迁移填补了这个值。在 Cart 模型中修改 add_product 函数来从产品环境中捕获这条信息。修改 LineItem 模型中的 total_price 函数来计算使用这个值的总数。修改 _line_item 部分来显示这个值。
- 倘若改变后程序能正常工作了，部署它。

第 17 章

Depot 回顾

在本章中，我们将学习：

- 回顾 Rails 的概念：模型、视图、控制器、配置、测试以及部署
- 文档化所做过的事情

恭喜你！进行到这一步，你已经透彻地理解了每个 Rails 应用程序的基本知识。在第三部分将学习更多的知识。而现在，大家只要放松心情，回顾在第二部分都做了什么？

17.1 Rails 的概念

第 3 章介绍了模型、视图和控制器。现在来看看如何把这些概念应用到 Depot 应用程序中。然后探讨如何在配置、测试和部署中使用它们。

17.1.1 模型

模型是管理所有持久性数据的地方，这些数据由应用程序保留。在 Depot 应用程序的开发过程中，共创建了 5 个模型：Cart、LineItem、Order、Product 和 User。

默认情况下，所有的模型都具有 id、created_at 和 updated_at 属性。此外，在本书的模型中，新添加了其他类型的属性：string（如：title、name）、integer(quantity)、text(description, address)、decimal(price) 以及关系键（product_id、cart_id）。我们甚至还创建了一个虚拟的属性，但它永远不会在数据库中储存，那就是密码（password）。

创建的 has_many 和 belongs_to 关系，用于在模型对象之间导航，例如，从 carts 到 LineItems，再到 Products。

利用迁移来更新数据库，在迁移过程中不仅介绍了新的架构信息，还修改了已存在的数据。并且证明可以用一种完全可逆的方式应用它们。

所创建的模型并不单单是数据的被动容器。对启动装置来说，它们积极地对数据进行验证，防止错误传播。我们为存在性、包容性、数值、范围、独特性、格式、确认（如果完成了练习，那么还有更多）创建了验证。还创建了自定义校验来确保没有任何商品项目在使用已删除的产品。同时，用 Active Record 钩子来保证管理员总是存在，并且确保一个未完成的回滚事物处理会导致更新失败。

另外，还创建了行为规则，规定了如何在购物车中添加产品，如何把所有的商品项目从一个购物车添加到一个订单中，如何加密和验证密码，以及如何计算各种总数。

最后，由于显示需要，创建了一个默认的给产品排序的功能。

17.1.2 视图

视图控制着应用程序向外界展现自我的方式。默认情况下, Rails 的脚手架提供 `edit`、`index`、`new` 和 `show`, 此外还有一个叫 `form` 的部分, 由 `edit` 和 `new` 共享。我们修改了其中的许多内容, 还为购物车和商品项目创建了新的部分。

除了模型支持的资源视图外, 也为 `admin`、`sessions` 和 `store` 完整地创建了新视图。

更新总体版面设计来建立一个共同的外观, 并感受整个网站。为此连接了一个样式表。并使用了 RJS 模板来产生 JavaScript, JavaScript 可以利用 Web 2.0 技术来让网站变得更容易交互。

使用帮助程序来决定什么时候该在主视图中隐藏购物车。

本地化顾客的视图, 让网页可以同时用英语和西班牙语显示。

虽然主要注意力集中在 HTML 视图, 但还是创建了纯文本视图和 Atom 视图。另外, 不是所有的视图都是为浏览器设计的, 与此同时也为电子邮件创建了视图, 这些视图可以共享显示商品项目的部分。

17.1.3 控制器

到现在为止, 共创建了 8 个控制器: 每个建立的模型 (共 5 个) 都有一个相应的控制器, 剩下的另外 3 个控制器用于支持 `admin`、`sessions` 和 `store` 自身的视图。

这些控制器采用多种方式与模型进行交互: 例如寻找并获得数据, 然后把数据放到实例变量中; 或者更新模型, 通过表单来保存输入的数据。控制器完成一项任务后, 我们要么重定向到了另一个行为, 要么转入到一个视图。用户可以转入 HTML、JSON 和 Atom 的视图。

创建的过滤器用于授权请求, 这些过滤器在选择的行为之前运行。我们把许多控制器共有的行为规则放到所有控制器的公有基类 (即 `ApplicationController`) 中。

我们管理会话, 了解登录的用户 (对管理员来说) 和购物车 (对顾客来说) 的动态, 跟踪当前的语言环境, 该语言环境用于输出的国际化。此外, 我们捕获错误, 记录它们, 并通过提醒告知用户。通过使用 `will_paginate` 插件来给订单分页。

收到订单后, 还会发送确认邮件。

17.1.4 配置

虽然约定为 Rails 的应用程序保持着最低所需的配置, 但还是做了一些自定义。

修改了数据库的配置, 以便在产品中能够使用 MySQL。

给资源、管理、会话的控制器、网站的根目录 (`root`, 也就是在线商店首页) 定义了路由路径。为了访问 Atom 中包含买家信息的订阅页面, 还定义了一个名为 `who_bought` 的 `products` 资源成员。

为 `i18n` 创建了初始化程序, 并且给英语页面 (`en`) 和西班牙语页面 (`es`) 都更新了语言环境信息。

为数据库创建了订阅页面的数据。

为部署创建了 Capistrano 脚本, 其中包括一些自定义任务。

17.1.5 测试

我们在整个过程中都保持并加强了测试部分。

为校验方法做单元测试，还测试了增加一个给定商品项目的数量。

Rails 为所有的脚手架控制器提供基础测试，当进行改变时，需要维持这些测试。在这些测试的基础上，我们添加其他的测试（如 Ajax）来保证在创建订单前购物车里存在商品。

使用固件测试来提供测试数据，这些数据可以增强测试。

最后，创建了一个集成测试来测试一个端对端的情景，该情景包括一个用户添加产品到购物车中，输入订单，以及收到确认邮件。

17.1.6 部署

使用产品质量数据库服务器（MySQL），把应用程序部署到了产品质量 Web 服务器（Apache httpd）上。在这个过程中，安装并配置了三项内容：Phusion Passenger，用以运行该应用程序；Bundler，用于跟踪包依赖关系；Git，用于配置管理代码。此外，Capistrano 用来精心安排从开发机器到产品部署 Web 服务器的更新。

使用测试和产品环境来避免在开发过程中的实验影响到产品。开发环境使用的是轻量级的 SQLite 数据库服务器和 Web 服务器，如最常用的 WEBrick。而测试则在可控制的环境中运行，该环境的测试数据由固件测试提供。

17.2 文档化所做的事情

若要完成这次回顾，需要从两个新的角度来看一看代码。

在应用程序的所有源文件上运行 Ruby's RDoc[⊖]实用程序，以此来创建一个漂亮的程序员文档，这在 Rails 中很容易做到。但是在生成文档之前，可能需要创建一个很好的介绍页面，以便下一任开发者知道该应用程序是做什么用的。

要完成这项工作，需要编辑 doc/README_FOR_APP 文件，输入任何可能有用的信息。该文件将用 RDoc 处理，因此其格式具有非常大的灵活性。

可以用 rake 命令生成 HTML 格式的文档：

```
depot> rake doc:app
```

上述命令会在 doc/app 目录下生成文档，详情如图 17.1 所示。

最后，我们也许会对编写了多少代码感兴趣。这时可以使用 Rake 任务。

```
depot> rake stats
(in /Users/dave/Work/depot)
```

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	636	409	9	45	5	7

[⊖] <http://rdoc.sourceforge.net/>

Helpers	24	24	0	1	0	22
Models	192	101	5	12	2	6
Libraries	0	0	0	0	0	0
Integration tests	201	138	2	9	4	13
Functional tests	424	285	9	0	0	0
Unit tests	163	123	13	2	0	59
<hr/>						
Total	1640	1080	38	69	1	13
<hr/>						
Code LOC: 534 Test LOC: 546 Code to Test Ratio: 1:1.0						

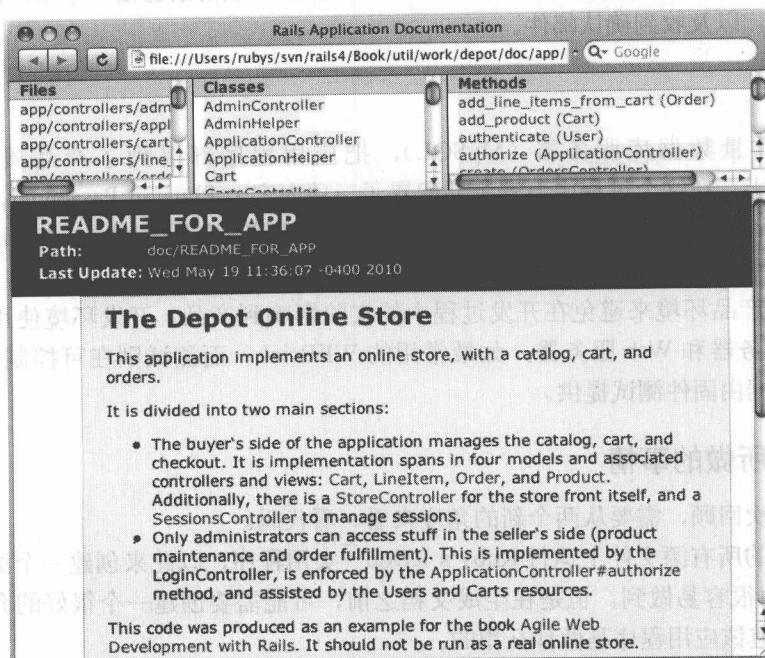


图 17.1 应用程序的内部文件

想想看，你完成了很多工作，但没有编写太多代码。而且，其中许多代码都是自动生成的。这就是 Rails 的神奇魔力。

第三部分

深入 Rails

第 18 章

自己去发现 Rails（工作）方法

在本章中，我们将学习：

- Rails 应用程序的目录结构
- 命名约定
- 为 Rails 自身生成文档
- 添加 Rake 任务
- 配置

在经历了整个 Depot 项目的开发后，你已经具备挖掘更深层 Rails 的条件。本书的剩余部分将按主题介绍 Rails（也就是说按照模块进行）。先前已经出现过许多行为中的模块了。这里将不仅仅讲述每个模块的功能，而且也会介绍如何扩展甚至替换模块，以及可能要这么做的原因。

第三部分的章节涵盖了 Rails 的所有主要子系统：Active Record 模块、Active Resource 模块、Action Pack 框架（同时包括 Action Controller 模块和 Action View 模块）以及 Active Support 模块。在这之后，将了解迁移的详细内容。

接下来会深入 Rails 的内部，展示如何把组件放到一起，如何启动组件，以及如何替换组件。在介绍如何将 Rails 的部件组合在一起后，会以一份调查结束本书，该调查针对大量的流行替换部件展开，其中大部分部件都可以在 Rails 外使用。

但首先得设置场景。本章涵盖了所有高层次的内容：目录结构、配置和环境，必须弄懂这些才能更好地理解其他内容。

18.1 东西都去哪里了

Rails 会假设某种运行时的目录布局，并且提供应用程序和脚手架生成器，以此来帮助创建

这个布局。例如，如果使用命令 `rails new my_app` 来生成 `my_app`，新应用程序的最高级别目录如图 18.1 所示。可以从该应用程序的顶级目录入手：

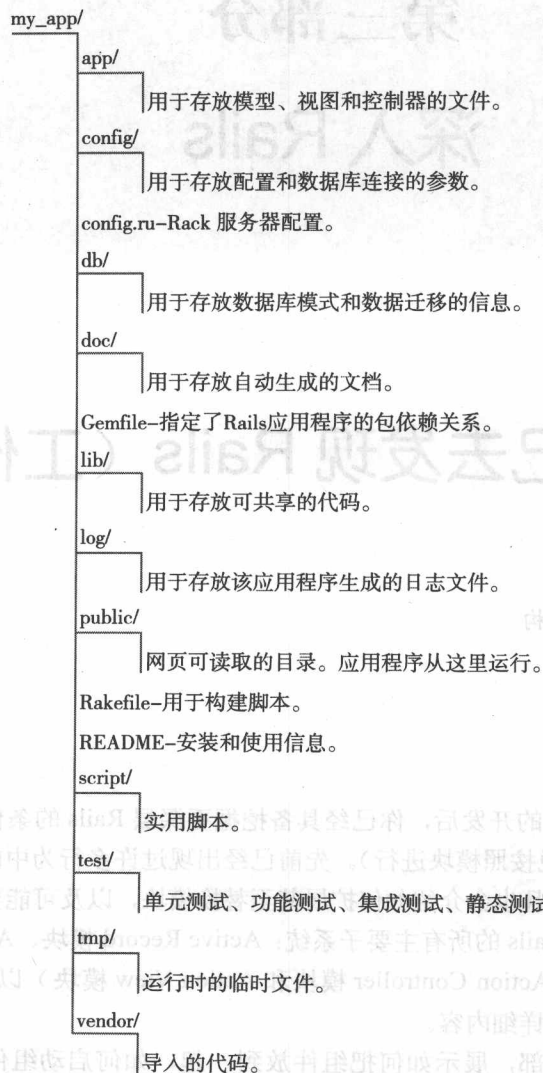


图 18.1 所有的 Rails 应用程序都存在这样一个顶级目录目录结构

- `config.ru` 会配置 Rack Web 服务器界面 (Rack Webserver Interface)，让它要么创建 Rails Metal 应用程序，要么使用 Rails 应用程序中的 Rack Middlewares。更详细的讨论请看 Rails 指南。[⊖]

⊖ http://guides.rubyonrails.org/rails_on_rack.html

Joe 问

那么, Rails 在哪儿?

如何组件化是 Rails 框架的一个有趣话题。从开发者的角度看, 你把所有的时间都花在处理高级别的模块上, 如 Active Record 模块和 Action View 模块。但深藏在其他组件之下, 有一个称为 Rails 的组件, 它默默地协调着模块间的活动, 让它们能亲密无间地共同工作。没有这个 Rails 组件, 很多工作都不能完成。但同时, 这个下层的基础设施只有一小部分功能与开发者的日常工作有关。相关部分的内容将会在本章的剩余部分讨论。

- Gemfile 指定了 Rails 应用程序的包依赖关系。在 will_paginate 插件添加到 Depot 应用程序时, 已经见识过该如何使用它了。应用程序的包依赖关系也包括数据库、Web 服务器甚至用于部署的脚本。

从技术上来说, 使用这个文件的并不是 Rails 本身, 而是该应用程序。可以在 config/boot.rb 文件和 config/application.rb 文件中找到对 Bundler[⊖]的调用。

- Rakefile 定义了这样一些任务: 运行测试、创建文档、提取这个架构的当前结构, 以及其他功能。在命令行输入 rake --tasks 来得到一个完整的列表。而输入 rake --describe task 可以查看特定任务的完整描述。
- README 包含有关 Rails 框架本身的普通信息。

现在来看一看每个目录都有些什么内容 (尽管没有必要按顺序来)。

18.1.1 应用程序的位置

大部分工作都在 app 目录下进行。应用程序的主要代码在 app 目录下, 如图 18.2 所示。在本书后面的章节中介绍各种 Rails 模块 (如 Active Record 模块、Action Controller 模块和 Action View 模块) 时, 再来讨论更多有关 app 目录结构的内容。

18.1.2 测试的位置

正如在 7.2 节、8.4 节以及 13.2 节中看到的那样, Rails 拥有丰富的措施来测试应用程序, 而 test 目录是所有测试相关活动的“大本营”, 包括定义测试所用数据的固定装置。

18.1.3 文档的位置

就像在 17.2 节里所做的那样, Rails 提供 doc:app Rake 任务来产生文档, 该文档在 doc 目录下。除了这个命令外, Rails 也提供其他任务, 用于生成文档: doc:rails 会提供正在运行的 Rails 版本, doc:guides 会提供使用指南。在建立指南之前, 得在 Gemfile 里添加 gem RedCloth (注意: 要区分大小写), 然后运行 bundle install。

⊖ <https://github.com/carlhuda/bundler>

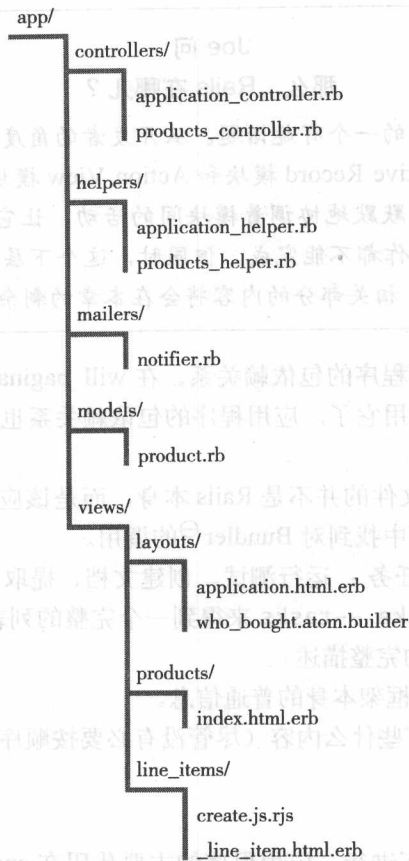


图 18.2 应用程序的主要代码保存在 app 目录下

Rails 还提供其他与文档相关的任务。输入命令 `rake -T doc` 可以显示所有任务。

18.1.4 支持库的位置

`lib` 目录存储着不能很好地适应模型、视图或控制器的应用程序代码。例如，已编写一个库，它可以为商店的顾客创建可以下载的 PDF 版发票^①。这些发票直接从控制器发送到浏览器（使用 `send_data` 函数）。顺理成章地，创建这些 PDF 版发票的代码将放在 `lib` 目录下。

`lib` 目录也是一个存放代码的好地方，这些代码在模型、视图、控制器之间共享。它可以是一个用来验证信用卡卡号校验位的库，或者一个执行某些财务核算的库，抑或是一个找出复活节日期的库。任何不直接定义为模型、视图或控制器的东西都应该放在 `lib` 中。

把诸多文件直接放在 `lib` 目录下并不是必需的。可以自由地创建子目录，以此来将 `lib` 下的相关功能分组。例如，在 Pragmatic Programmer 网站上，用来生成发票、顾客的发货文件，以及其他 PDF 格式文档的代码都保存在 `lib/pdf_stuff` 目录下。

① 就像在 Pragmatic Programmer 商店里做的一样。

在先前的 Rails 版本中, lib 目录下的文件会自动包含在搜索里。而现在这个功能是可选的, 必须显性地开启。成功开启的办法是, 在 config/application.rb 文件中添加下面内容:

```
config.autoload_paths += %W(#{Rails.root}/lib)
```

一旦把文件放在 lib 目录里, 并且把 lib 目录添加到自动加载的路径中, 就可以在该应用程序的其他部分使用它们了。如果文件包含类或模块, 并且文件以该类或模块的小写字母命名, 那么 Rails 会自动加载这个文件。例如, 也许在 lib/pdf_stuff 目录下的 receipt.rb 文件中有一个 PDF 版发票的打印程序。只要这个类以 PdfStuff::Receipt 命名, Rails 就可以找到并自动加载。

如果一个库不能满足加载条件, 可以使用 Ruby 的 require 机制。如果这个文件在 lib 目录下, 可以直接通过名称获取它 (参见 4.7 节)。例如, 倘若复活节日期的计算库在 lib/easter.rb 文件里, 可以在任意模型、试图或控制器中包含, 方法如下:

```
require "easter"
```

如果这个库在 lib 的子目录下, 别忘了在 require 语句中包含那个目录的名称。例如, 要包含一个航空邮件运输计算器, 可以添加下面内容:

```
require "shipping/airmail"
```

18.1.5 Rake 任务的位置

在 lib 目录下也可以找到一个空的 tasks 目录。在这里可以编写自己的 Rake 任务, 这些任务让该项目自动化。可惜本书不是关于 Rake 的书, 因此在这儿不打算深入讨论, 但下面是一个简单应用的例子。Rails 提供 Rake 任务, 告诉用户已执行过的最新迁移。

但是, 能看到所有执行过的迁移也许更有帮助。可以编写一个 Rake 任务, 用来输出 schema_migration 表中列出内容的版本。这些任务都是用 Ruby 写的, 但是它们必须以 .rake 作为文件后缀。调用写好的 db_schema_migrations.rake:

```
depot_t/lib/tasks/db_schema_migrations.rake
```

```
namespace :db do
  desc "Prints the migrated versions"
  task :schema_migrations => :environment do
    puts ActiveRecord::Base.connection.select_values(
      'select version from schema_migrations order by version' )
  end
end
```

就像任何其他 Rake 任务一样, 可以从命令行运行它:

```
depot> rake db:schema_migrations
(in /Users/rubys/Work/...)
20110211000001
20110211000002
20110211000003
20110211000004
20110211000005
20110211000006
20110211000007
```

可以在 <http://rubyrake.org/> 上查阅 **Rake** 的文档, 以获得更多有关编写 **Rake** 任务的信息。

18.1.6 日志的位置

Rails 在运行的同时会生成许多有用的日志信息。这些信息(在默认情况下)会放在 **log** 目录下。在这个目录下可以发现三个主要的日志文件, 分别是 **development.log**、**test.log** 和 **production.log**。这些日志文件不单单包含简单的跟踪线, 还包含时间统计、缓存信息, 以及执行过的数据库语句扩展。

使用哪个日志文件取决于该应用程序运行的环境(在 18.1 节中讨论 **config** 目录时, 将介绍更多有关环境的内容)。

18.1.7 静态网页的位置

public 目录是应用程序的外部接口。**Web** 服务器把这个目录作为应用程序的基础。在这里放置了静态(换句话说, 不改变)的文件, 如样式表、**JavaScript** 甚至一些网页。

18.1.8 脚本的位置

如果认为编写脚本代码很有帮助, 可以在 **scripts** 目录中存放这些脚本。它们可以在命令行中运行, 然后为应用程序执行各种各样的维护任务。

这个目录也保存 **Rails** 的脚本。在命令行运行 **rails** 命令时, 就会运行这个 **Rails** 脚本。第一个传入该脚本的参数决定了 **Rails** 将执行的函数:

benchmarker

为一个或多个应用程序的方法生成性能数据。

console

允许与 **Rails** 应用程序的方法交互。

dbconsole

允许直接通过命令行与数据库交互。

destroy

删除由 **generate** 自动生成的文件。

generate

代码生成器。在服务器外, 它将生成控制器、邮件程序、模型、脚手架, 以及 **Web** 服务器。也可以从 **Rails** 的网站^①上下载其他的生成器模型。不带任何参数运行 **generate** 可以得到特定生成器的用法信息, 如 **rails generate migration**。

new

生成 **Rails** 应用程序的代码。

plugin

它会帮忙安装和管理插件——这些功能部件扩展了 **Rails** 的能力。

① <http://wiki.rubyonrails.org/rails/pages/AvailableGenerators>

profiler

创建运行时的用户配置文件总结, 这个总结与该应用程序处理 URI 请求有关。

runner

在 Web 环境外的应用程序中执行一种方法。这是一个 rails console 的非交互等价。可以用它来从 cron 任务中调用缓存届满 (expiry) 方法, 或者处理收到的邮件。

server

使用 Mongrel (如果它在服务器上可用) 或 WEBrick, 让 Rails 应用程序在本身包含的 Web 服务器上运行。在开发 Depot 应用程序的过程中, 已经使用过这个函数。

18.1.9 临时文件的位置

Rails 把临时文件放在 tmp 目录下也许并不令人惊讶。在这儿可以找到缓存内容、会话和接口的子目录。通常情况下, Rails 会自动清除这些文件, 但是如果偶然间出了错, 可能需要到这里看一看, 然后删除旧文件。

18.1.10 第三方代码的位置

vendor 目录用来存放第三方代码。如今, 这一部分的代码一般有两种来源。

第一, Rails 把插件安装到 vendor/plugins 目录下。无论是在开发过程中还是在运行过程中, 插件都是扩展 Rails 功能的手段。

第二, 可以将 Rails 及其所有的包依赖关系安装到 vendor 目录下, 如 16.2 节中所描述的那样。

如果想再次使用系统范围的 gem 版本, 可以删除 vendor/cache 目录。

18.1.11 配置的位置

config 目录包含了配置 Rails 的文件。在开发 Depot 的过程中, 配置了一些新路由、配置了数据库、创建了初始化程序、修改了一些语言环境并且定义了部署。剩下的配置由 Rails 的约定完成。

在运行应用程序前, Rails 加载并执行 config/environment.rb 文件和 config/application.rb 文件。这些文件设置的标准环境会自动将以下目录包含在该应用程序的加载路径中 (与这个应用程序的基础目录有关):

- app/controllers 目录及其子目录
- app/models 目录及其以下划线或小写字母开始命名的子目录
- vendor 目录以及每个 plugin 子目录下包含的 lib 目录
- app 目录、app/helpers 目录、app/mailers 目录、app/services 目录、config 目录和 lib 目录

以上的每个目录只有在加载路径存在的情况下才会添加到其中。

此外, Rails 会加载每个环境的配置文件。这些文件在 environments 目录下, 并且记录了存放配置选项的地点, 这些配置选项花样百出, 随环境而定。

这么做的原因是 Rails 懂得，在编写代码、测试代码，以及在产品中运行代码时，开发人员的需求有所不同。在编写代码时，想要许多日志文件，想要方便地重载改变了的源文件，想要直截了当的错误通知等。在测试中，则想要一个单独存在的系统，以便可以重复结果。在生产环境中，系统应该转向提高性能，而用户则应远离错误。

对该应用程序来说，决定运行时环境的切换是外部的。这意味着从开发到测试，再到生产，无需改变该应用程序的代码。在第 16 章中，`rake` 命令里使用了 `RAILS_ENV` 参数来指定环境，而 Phusion Passenger 则在 Apache 的配置文件中使用 `RailsEnv` 行。当用 `rails server` 命令启动 WEBrick 时，又使用 `-e` 选项：

```
depot> rails server -e development
depot> rails server -e test
depot> rails server -e production
```

倘若有特殊需求，例如，想要一个临时环境，也可以创建自己的环境。需要在数据库的配置文件中添加一个新的部分，并且在 `config/environments` 目录下增加一个新文件。

在这些配置里面放什么东西完全取决于个人喜好。在先前用 `rake doc:guides` 命令生成的“Configuring Rails Applications guide”（配置 Rails 应用程序指南）中，能找到可以设置的配置参数列表。这些信息在网上也能找到^①。

18.2 命名约定

刚接触 Rails 的新手有时会对 Rails 自动处理命名的方式感到迷惑。大家会惊奇地发现，调用 `Person` 模型类时，Rails 自然而然地就知道要去找名为 `people` 的数据库表。在这一节中，将学习这种隐式命名的工作原理。

这里提及的规则是 Rails 使用的默认约定。可以使用配置选项来重写所有约定。

18.2.1 混合大小写、下划线和复数

通常使用短语来命名变量和类。在 Ruby 中，约定变量名全部小写，词与词之间用下划线分开。类和模块的命名则不同：不使用下划线，并且短语中的每个词（包括第一个词）开头都要大写。（相当明显，这种形式称为混合大小写（mixed case）。）这些约定让变量名形如 `order_status`，而类的名称则为 `LineItem`。

Rails 继承了这些约定，并且在两个方面进行了扩展。第一，Rails 假设数据库表名（如变量名）由小写字母组成，词之间用下划线连接。另外，Rails 还假设表名总是复数的。于是，表名看起来如 `orders` 或 `third_parties`。

第二方面，Rails 假设文件名由小写字母和下划线组成。

Rails 使用命名约定的知识来自动转换名称。例如，应用程序可能包含处理商品项目的模型类。可以用 Ruby 的命名约定来定义类的名称，如 `LineItem` 类。从这个名称开始，Rails 会自动推导出以下内容：

① <http://guides.rubyonrails.org/configuring.html>

- 相关联的数据库表名为 `line_items`。这是类名转化为小写，然后在词之间加入下划线，最后再复数化的结果。
 - Rails 还知道去 `line_item.rb` 文件中 (`app/models` 目录下) 寻找类的定义。
- Rails 控制器也有额外的命名约定。倘若应用程序中有商店控制器，那么将发生下面的事情：
- Rails 假设类名为 `StoreController`，并且这个类在 `app/controllers` 目录下的 `store_controller.rb` 文件中。
 - Rails 也会假设在 `app/helpers` 目录下的 `store_helper.rb` 文件中有一个名为 `StoreHelper` 帮助程序模块。
 - Rails 会在 `app/views/store` 目录下为这个控制器寻找视图模板。
 - 在默认情况下，Rails 将取得所有视图的输出，将它们包装到布局模板中。这个模板在 `app/views/layouts` 目录下的 `store.html.erb` 或 `store.xml.erb` 文件中定义。
- 所有这些约定如图 18.3 所示。

Model Naming	
Table	<code>line_items</code>
File	<code>app/models/line_item.rb</code>
Class	<code>LineItem</code>

Controller Naming	
URL	<code>http://../store/list</code>
File	<code>app/controllers/store_controller.rb</code>
Class	<code>StoreController</code>
Method	<code>list</code>
Layout	<code>app/views/layouts/store.html.erb</code>

View Naming	
URL	<code>http://../store/list</code>
File	<code>app/views/store/list.html.erb (or .builder or .rjs)</code>
Helper	<code>module StoreHelper</code>
File	<code>app/helpers/store_helper.rb</code>

图 18.3 命名约定在 Rails 应用程序中如何工作

除了 Rails 控制器新增的命名约定外，还有一个附加的内容。在普通的 Ruby 代码中，要引用 Ruby 源文件内定义的那些类和模块前，必须使用 `require` 关键字来包含这些文件。因为 Rails 知道文件名和类名之间的关系，`require` 在 Rails 应用程序中通常是没有必要的。相反，第一次引用未知的类或模块时，Rails 使用命名约定来把类名转换为文件名，并且尝试加载文件背后的场景。这么做所产生的连锁效应是，只要引用模型类的名称，然后这个模型就会自动下载到应用程序中。

18.2.2 把控制器分组到模块中

至此，所有的控制器都放置在 `app/controllers` 目录下。有时约定也会添加更多结构到这个目录中。例如，在线商店可能最终有许多控制器，这些控制器配合执行，但管理功能却不相交。可以选择将这些控制器分组到一个单独的 `admin` 命名空间下，而不是去干扰高级别的命名空间。

Rails 用简单的命名约定来完成这件事。如果传入的请求包含名为 `admin/book` 的控制器，

Rails 将在 `app/controllers/admin` 目录下寻找称为 `book_controller` 的控制器。也就是说，控制器名称的最后部分总是指向文件 `name_controller.rb`，并且所有指向路径的信息都会用于为子目录定位，而所有的定位都始于 `app/controllers` 目录。

试想，程序有两个控制器组（如 `admin/xxx` 和 `content/xxx`），并且两个组都各定义了 `book` 控制器。这样，在 `app/controllers` 目录的 `admin` 和 `content` 子目录下都将会有 `book_controller.rb` 文件。这两个控制器文件都可以定义 `BookController` 类。如果 Rails 不作任何分析，这两个类就会发生冲突。

要解决这个问题，Rails 假设 `app/controllers` 子目录下的控制器都在 Ruby 模块下，并且以子目录命名。这样，在 `admin` 子目录里的 `book` 控制器将以如下形式声明：

```
class Admin::BookController < ActionController::Base
  # ...
end
```

而 `content` 子目录下的书本控制器则在 `Content` 模块中：

```
class Content::BookController < ActionController::Base
  # ...
end
```

因此，这两个控制器就在该应用程序中分隔开了。

这些控制器的模板在 `app/views` 的子目录中。所以，视图模板的相关请求如下：

```
http://my.app/admin/book/edit/1234
```

该请求保存在下面的文件中：

```
app/views/admin/book/edit.html.erb
```

你将会很高兴地知道，控制器生成器懂得控制器在模块中的概念，可以用如下命令创建它们：

```
myapp> rails generate controller Admin::Book action1 action2 ...
```

David 说

为什么对表名使用复数？

因为复数的表名更符合使用英语国家的日常对话的语法。确实是这样，如 “Select a Product from products.” 和 “Order has many line items.”。

这么做的目的是，通过创建一种能在编程和语言之间共享的领域语言，使它们能够沟通。拥有这样一种语言意味着减少人为翻译，也因此减少了当讨论真正落实为商品主体，以及与客户讨论产品描述时会产生歧义。这些沟通的缺陷必会导致错误。

如果遵循标准约定，Rails 将免费提供大部分的配置让用户尝尝甜头。因此，开发者会因为做对事情而得到奖励。Rails 不是要你放弃“自己的方式”，而是提倡免费得到生产力。

18.3 本章小结

Rails 中的每样东西都有位置，而我们系统地探讨了那些角落和缝隙。在每个地方，Rails 包含的文件和数据都遵循命名约定，这些已经讨论过了。同时，还填补了一些缺失的部分：

- 为 Rails 本身生成 API 和用户指南文件。
- 添加 Rake 任务来输入迁移的版本。
- 展示了如何单独地配置每个 Rails 执行文件的环境。

接下来要讨论的是 Rails 的主要子系统，从最大的 Active Record 模块开始。

第 19 章

Active Record 模块

在本章中，我们将学习：

- 方法 `establish_connection()`
- 表、类、字段和属性
- 属性 `id` 及其相互关系
- 创建、读取、更新和删除操作
- 回调 (callback) 和事务 (transaction)

Rails 提供了对象关系映射 (Object-Relational Mapping, ORM) 层 Active Record。它是 Rails 应用程序模型的工具。

在这一章中，我们将数据映射至表的行和列（如在前面程序 Depot 中所做的）。然后，借助于 Active Record 管理表的关联，在此过程中包括了创建 (create)、读取 (read)、更新 (update) 和删除 (delete) 操作（即通常业内所说的 CRUD 方法）。最后，还将进一步探讨 Active Record 对象生命周期（包括回调和事务）。

19.1 定义数据结构

在 Depot 应用程序中，定义了一组模型，其中包括了 Order 模型。这一特殊模型具有许多属性，比如数据类型是字符串的电子邮件地址等。除了定义属性之外，Rails 还提供了名为 `id` 的属性，用来标识数据库记录的主键。Rails 还提供了其他几个属性，包括跟踪何时更新过数据库记录。最后，Rails 完全支持模型（如 Order 与 LineItem 模型）之间的相互关联。

Rails 对模型提供了大量支持。就此作如下探讨。

19.1.1 使用表和字段的规则

类 `ActiveRecord::Base` 的每个子类，如类 Order，封装了各自的数据库表。在默认情况下，Active Record 假定，与给定模型类关联的表名是该类名的复数形式。如果类名包含多个大写的单词，则应在这些单词之间使用下划线划分表名。

模型类名	数据库表名	模型类名	数据库表名
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

上述原则反映了 DHH 的哲学理念，模型类名应该是单数形式，而其对应的数据库表名则应该是复数形式。

虽然 Rails 能够正确处理大多数的非规则英文单词复数情况，但有时也会偶然发现不能正确处理的情况。如果遇到这类情况，可以通过修改词形变化 (inflection) 文件，补充英语语言的特性和不一致性，使得 Rails 可以理解它们。

```
depot_t/config/initializers/inflections.rb
```

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

```
# Add new inflection rules using the following format
```

```
# (all these examples are active by default):
```

```
# ActiveSupport::Inflector.inflections do |inflect|
```

```
#   inflect.plural /^(ox)$/i, '\1len'
```

```
#   inflect.singular /^(ox)en/i, '\1'
```

```
#   inflect.irregular 'person', 'people'
```

```
#   inflect.uncountable %w( fish sheep )
```

```
# end
```

```
ActiveSupport::Inflector.inflections 'do |inflect|
```

```
  inflect.irregular 'tax', 'taxes'
```

```
end
```

如果有必要处理应用程序的遗留表或者不喜欢这种关系方式，那么，通过对给定类设置 `table_name` 的值，就可以管理与已知模型类关联的表：

```
class Sheep < ActiveRecord::Base
```

```
  self.table_name = "sheep"
```

```
end
```

Active Record 模型类的对象对应于数据库表的行，而对象的属性则对应于数据库表的字段。你可能已经注意到，这里并没有在 `Order` 类定义中涉及 `orders` 表的任何字段。这是因为 Active Record 可以在运行时动态地确定它们。Active Record 映射数据库内部模式，以设置封装表的类。

在 Depot 应用程序中，下面的迁移代码定义了 `orders` 表：

```
depot_r/db/migrate/20110211000007_create_orders.rb
```

```
def self.up
```

```
  create_table :orders do |t|
```

```
    t.string :name
```

```
    t.text :address
```

```
    t.string :email
```

```
    t.string :pay_type, :limit => 10
```

```
    t.timestamps
```

```
  end
```

```
end
```

不管你脑海里有什么想法，都可以在 Rails console (控制台) 里先试一下这个模型。首先查询字段名清单：

```
depot> rails console
Loading development environment (Rails 3.0.5)
>> Order.column_names
=> ["id", "name", "address", "email", "pay_type", "created_at", "updated_at"]
```

然后查询字段 `pay_type` 的详细信息:

```
>> Order.columns_hash["pay_type"]
=> #<ActiveRecord::ConnectionAdapters::SQLiteColumn:0x7fe673f7da80
  @name="pay_type", @null=true, @default=nil, @sql_type="varchar(10)",
  @type=:string, @scale=nil, @precision=nil, @primary=false,
  @limit=10>
```

请注意, Active Record 已收集了有关 `pay_type` 字段的大量信息。大家知道, 这个字段最多可以有 10 个字符的字符串, 没有默认值, 且不是主键, 还有可能包含空值。在前面首次尝试使用 `Order` 类时, Rails 通过底层数据库 (underlying database) 得到了这一信息。

Active Record 对象的属性通常对应于数据库表相应行的数据记录。例如, 表 `orders` 应该包含以下数据:

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders limit 1"
id = 1
name = Dave Thomas
address = 123 Main St
email = customer@example.com
pay_type = Check
created_at = 2010-06-18 00:36:57.355069
updated_at = 2010-06-18 00:36:57.355069
```

如果把查询到的行记录存入到 Active Record 对象中, 那么该对象将有 7 个属性。属性 `id` 应该是 1 (该值是类 `Fixnum` 的对象), 而属性 `name` 应该是字符串 “Dave Thomas” 等, 其他属性就不再一一罗列。

David 说 属性在哪里?

由于区别对待程序员和数据库管理员 (DataBase Administrator, DBA) 的做法, 以致一些开发人员看到了在程序代码与数据结构描述之间的严格界限。而 Active Record 模糊了这个界限, 在模型中已不需要显式地定义属性。

但也不用担心。实践已经证明, 无论是在看待数据库结构描述、单独 XML 映射文件, 还是模型的内联属性, 这种差异都在变小。复合视图 (composite view) 是类似于这种分离的概念, 这种概念已经在 “模型 - 视图 - 控制器” 模式中出现了——只是在更小的范围内。

处理表结构描述并不是一件令人愉快的事情, 一旦把它作为模型定义的一部分, 这种不快也就不存在了, 这时你就会开始体会到坚持 DRY 的好处。当需要添加属性到模型中时, 就只需先创建新的迁移, 然后把它重新加载到该应用程序中。

以 “构建” 形式走出传统软件开发模式的这种演变, 使其像使用其他代码一样敏捷。这样就更容易从少量的结构描述开始, 当需要时再不断进行扩充和改变。

通过 accessor 方法可以访问这些属性。当 Rails 映射到结构描述时, Rails 自动地构造属性的读与写方法。

```
o = Order.find(1)
puts o.name          #=> "Dave Thomas"
o.name = "Fred Smith" # set the name
```

设置属性值不会改变数据库的任何东西——为了固化这种更改, 必须保存所修改的对象。

Active Record 把读取属性返回值尽可能地转换为适当的 Ruby 类型 (例如, 如果数据库字段是时间戳, 那么将会返回时间对象)。如果想获得属性的原始值, 就在其名称后追加 `_before_type_cast`, 如下代码所示:

```
product.price_before_type_cast #=> "29.95", a string
product.updated_at_before_type_cast #=> "2008-05-13 10:13:14"
```

在模型的代码中, 可以使用 `read_attribute` 和 `write_attribute` 私有方法。它们把属性名称用作字符串参数。

在图 19.1 中, 可以看到在 SQL 语言类型和 Ruby 语言表达形式之间的映射。Decimal 和 Boolean 字段稍微棘手一点儿。

SQL 类型	Ruby 类型	SQL 类型	Ruby 类型
int, integer	Fixnum	float, double	Float
decimal, numeric	BigDecimal ^①	char, varchar, string	String
interval, date	Date	datetime, time	Time
clob, blob, text	String	boolean	see text

①当 Decimal 和 numeric 字段的范围是 0 时映射为 integer。

图 19.1 SQL 类型与 Ruby 类型的映射对照

Rails 把无小数的 Decimal 字段映射为 Fixnum 对象; 在其他情况下, 将它映射到 BigDecimal 对象, 可以确保其精度不会丢失。

在 Boolean 的情况下, 事实上, 并非所有的数据库都有原生布尔类型。如在 MySQL 中, 0 代表 false, 而 1 代表 true。遗憾的是, Ruby 把所有不是 false 或者 nil 的对象都视为 true, 所以直接使用这些值是有问题的。可以给字段名附加一个问号来替代对这些值的直接使用:

```
user = User.find_by_name("Dave")
if user.superuser?
  grant_privileges
end
```

除了上面定义的属性外, 还有大量具有特殊含义的属性, Rails 也自动地提供了属性。

19.1.2 Active Record 所提供的附加字段

大量字段名对 Active Record 有着特殊意义。这里摘要如下:

- created_at、created_on、updated_at、updated_on

行记录的创建或最后更新的时间戳是自动更新的。这样可确保基础数据库字段有接收日期、日期时间或字符串的能力。通常情况下, Rails 应用程序对于日期字段使用 `_on` 后缀, 而时间字段采用 `_at` 后缀。

- `lock_version`

如果表存在字段 `lock_version`, 那么 Rails 将会跟踪并且乐观锁定 (optimistic locking) 行记录版本号。

- `type`

Active Record 可以生成子类。要是生成了子类, 所有一切子类的属性都会保存在相同表中。作为字段的属性 `type` 的作用是查询每一行记录属于哪个子类。

- `id`

这是表的主键字段的默认名称。

- `xxx_id`

这是外键引用表的默认名称, 而 `xxx` 是该表名称的复数形式。

- `xxx_count`

这是维护子表 `xxx` 的计数器缓存。

附加的插件, 如 `act_as_list`[⊖], 可以用来定义补充的字段。

在数据库操作中, 主键和外键起到了至关重要的作用, 并值得进一步讨论。

19.2 查找和遍历记录

在 Depot 应用程序中, 类 `LineItems` 直接关联到三个其他模型: `Cart`、`Order` 和 `Product`。此外, 借助于资源对象, 也可实现模型间的间接关联。借助于模型 `LineItems`, 实现在模型 `Order` 和模型 `Product` 之间的关联就是这样的示例。

所有这一切都离不开主键 `id`。

19.2.1 识别单个行

Active Record 类对应于数据库的表。而类的实例对应于数据库表的单个行记录。例如, 调用 `Order.find(1)`, 返回 `Order` 类的实例, 该实例包含主键 `id` 为 1 的行记录数据。

如果正在创建新的 Rails 应用程序结构描述, 那么就可能会顺其自然, 让 `id` 主键字段添加到所有表中。但是, 如果需要使用已有的结构描述, 那么 Active Record 提供了覆盖表主键默认名的简单方法。例如, 也许会利用现有遗留结构描述的工作, 使用 ISBN 作为 `books` 表的主键。

通过下面的 Active Record 模型代码, 可以详细说明这一点。

```
class LegacyBook < ActiveRecord::Base
  self.primary_key = "isbn"
end
```

[⊖] https://github.com/rails/acts_as_list

在通常情况下，为了已创建的并添加到数据库的行记录，Active Record 开始建立了新的递增整数（但有些值可能是空的）的主键值。然而，如果要覆盖主键字段名，那么，在保存新记录前，就必须把主键设置为唯一值。这也许令人不可思议，但是通过设置属性名为 `id`，这件事情就可以做到了。只要涉及 Active Record，主键属性就始终设置称为 `id` 的属性。`primary_key=` 声明只是设定了用于表中的字段名。即使数据库的主键是 `isbn`，也可称为 `id` 属性，如下所示：

```
book = LegacyBook.new
book.id = "0-12345-6789"
book.title = "My Great American Novel"
book.save

# ...

book = LegacyBook.find("0-12345-6789")
puts book.title # => "My Great American Novel"
p book.attributes #=> {"isbn" => "0-12345-6789",
                      "title" => "My Great American Novel"}
```

但更令人难以理解的是，具有 `isbn` 和 `title` 字段的模型对象属性 `id` 并未出现。只需要记住，在需要设置主键时使用 `id`。而在其他情况下，就用实际的字段名。

为了访问模型的主键，模型对象也重新定义了 Ruby 中的 `id` 和 `hash` 法。这意味着，取得有效 `id` 的模型对象就可以作为 `hash` 键使用。这也意味着，未保存的模型对象无法可靠地用作 `hash` 键（因为它们还没有有效的 `id`）。

最后一点：如果两个模型对象是同一个类的实例，并有相同的主键，那么，Rails 就认为它们是相等的（使用方法 `==`）。这就是说，即使未保存的模型对象具有不同的属性数据，它们也应该返回“是相等”的比较结果。如果发现自己正在比较未保存的模型对象（这不是特别常用的一种操作），那么可能需要重写 `==` 方法。

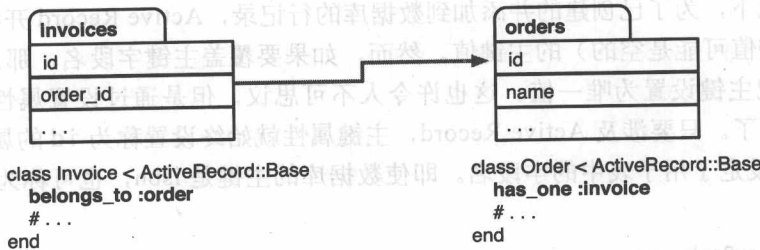
正如下面将要看到的，主键属性 `id` 在关联中扮演着重要角色。

19.2.2 模型关联性说明

Active Record 支持三种表之间的关系：一对一、一对多和多对多。通过添加模型声明，可以表示这些关联，如 `has_one`、`has_many`、`belongs_to`，或者更奇妙的函数名 `has_and_belongs_to_many`。

19.2.3 一对一关联

一对一关联（或者更准确地说，一对零或一对一的关联）是指通过使用一个表中任何一行的外键引用另一个表中最多一行的记录来实现的。在表 `orders` 和表 `invoices` 之间应该存在一对一关联：每个订单最多有一张发票。

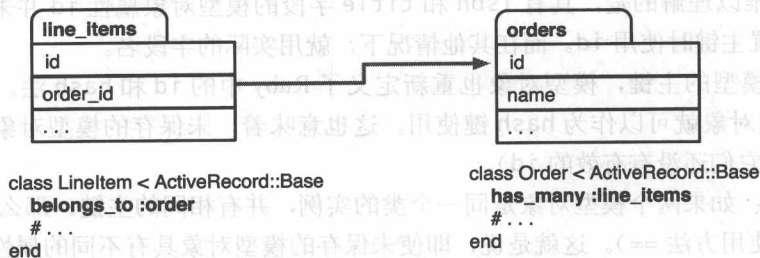


如上图所示，在 Rails 程序中，把 `has_one` 声明加入到模型 `Order`，而把 `belongs_to` 声明加入到模型 `Invoice`。

上图还隐含了一个重要规则：包含外键表的模型总是具有声明 `belongs_to`。

19.2.4 一对多关联

一对多的关联可以代表对象集合。例如，一个订单应该关联很多商品项目。在数据库中，对于特定订单而言，所有的商品项目记录都包含引用订单的外键字段。

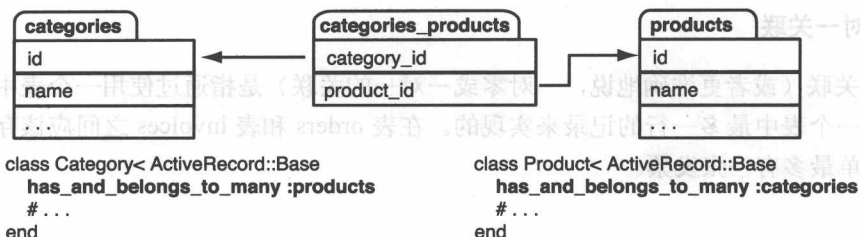


在 Active Record 中，父对象（逻辑上包含子对象集合的对象）使用 `has_many` 声明其与子对象的关联，而子对象使用 `belongs_to` 来指向其父对象。在上面例子中，类 `LineItem` 使用代码 `belongs_to :order`，而类 `Order` 使用代码 `has_many :line_items`。

值得注意的是，因为表 `line_items` 包含外键，所以类 `LineItem` 要声明 `belongs_to`。

19.2.5 多对多关联

最后该把商品分门别类。一个商品可以属于多个类别，而每个类别可能包含多种商品。这是多对多关联的例子。似乎看起来，关联的任何一方都包含了对方的项目集合。



在 Rails 程序中，可以把 `has_and_belongs_to_many`（书面上可简写为“`has_and_belongs_to_many`”）声明添

加到两个模型中。

多对多关联是对称的——用 `hasbtm` 声明相关的两个表实现了相互关联。

Rails 使用中间连接表来实现多对多关联。这个中间连接表包含连接两个目标表的外键对。Active Record 规定，连接表的名称是由两个目标表名按字母顺序连接组成，连接符采用下划线。在上面例子中，把表 `categories` 与表 `products` 连接起来，这样，Active Record 会查找名称为 `categories_products` 的连接表。

也可以直接定义连接表。在 Depot 应用程序中定义了连接表 `line_items`（原文是“LineItems”，这是错误的。——译者注），它要么把 `products` 与表 `carts` 关联起来，要么把表 `products` 与表 `orders` 关联起来。定义中间连接表的办法也提供了存储附加属性（如本例的属性 `quantity`）的地方。

现在解释完了数据定义的内容，接下来要做的是访问数据库内所包含的数据，让我们一起来做吧。

19.3 创建、读取、更新和删除操作

正如数据库 SQLite 和 MySQL 名称所强调的那样，访问数据库都是借助于结构化查询语言（Structured Query Language, SQL）。在大多数情况下，Rails 将负责这一工作，不过是否使用 SQL 语句完全取决于我们。像后面将要看到的那样，不仅可以提供 SQL 语言子句，而且还支持数据库可执行的完整 SQL 语句。

如果你已经熟悉 SQL 语言，在阅读本节时，就会注意到 Rails 是如何提供众所周知的 SQL 语言子句，如 `select`、`from`、`where` 和 `group by` 等。如果你还不太熟悉 SQL 语言，那么 Rails 优势之一就是可以把这方面的知识暂时搁置，直到真正需要在数据库层面上才会这样做。

针对 Depot 应用程序的模型 `Order`，在本节中将继续推进这一实例工作。下面将使用 `ActiveRecord` 方法，以便应用这四个基本数据库操作：创建、读取、更新和删除。

19.3.1 创建新的行记录

我们知道，Rails 视表为类且视对象为行记录，其过程是：先创建类的新对象，然后开始建立相应表的行记录。例如，先调用 `order.new`，创建 `Order` 类的新对象，该对象对应于表 `orders` 的行记录。然后输入属性值（对应到数据库中的字段值）。最后，调用该订单对象的 `save` 方法，将其存储到数据库中。如果不调用 `save` 方法，那么订单对象 `an_order` 只存在于本地计算机内存中。

```
e1/ar/new_examples.rb
```

```
an_order = Order.new
an_order.name      = "Dave Thomas"
an_order.email     = "dave@example.com"
an_order.address   = "123 Main St"
an_order.pay_type  = "check"
an_order.save
```

Active Record 的构造函数接收一个可选的代码块。如果存在的话，通过调用作为参数的这段代码块，可创建新对象。如果想要创建并保存该对象，那么就无需创建新的局部变量，这种方法应该是有用的。

```
e1/ar/new_examples.rb
```

```
Order.new do |o|
  o.name = "Dave Thomas"
  # ...
  o.save
end
```

最后，在下面的代码中，Active Record 的构造函数包含了一个可选的参数，它是 hash 类型的属性值。每个元素都对应于所要设置的属性名及其值。把 HTML 表单数据存储到数据库的行记录，这种做法也是可取的。

```
e1/ar/new_examples.rb
```

```
an_order = Order.new(
  :name => "Dave Thomas",
  :email => "dave@example.com",
  :address => "123 Main St",
  :pay_type => "check")
an_order.save
```

请注意，在所有这些例子中，并没有设置新行记录的 id 属性。因为前面曾经使用 Active Record 默认整数的字段作为主键，所以 Active Record 会自动地创建唯一值，且当保存该行记录时设置 id 属性。这样就可以通过查询该属性找到这个值。

```
e1/ar/new_examples.rb
```

```
an_order = Order.new
an_order.name = "Dave Thomas"
# ...
an_order.save
puts "The ID of this order is #{an_order.id}"
```

这一构造函数 new 在内存中创建新 Order 对象；一定要记住在某个未来时刻把它保存到数据库中。Active Record 有个简便的 create 方法，在创建实例模型对象的同时，也把它存储到数据库中。

```
e1/ar/new_examples.rb
```

```
an_order = Order.create(
  :name => "Dave Thomas",
  :email => "dave@example.com",
  :address => "123 Main St",
  :pay_type => "check")
```

对于 create 方法，可以传递对象属性为散列的数组；它将会在数据库中创建多个行记录，且返回相应模型对象的数组：

```
e1/ar/new_examples.rb
```

```
orders = Order.create(
  [ { :name    => "Dave Thomas",
      :email    => "dave@example.com",
      :address  => "123 Main St",
      :pay_type => "check"
    },
    { :name    => "Andy Hunt",
      :email    => "andy@example.com",
      :address  => "456 Gentle Drive",
      :pay_type => "po"
    }
  ] )
```

`new` 和 `create` 函数取数组元素为散列类型值的真正原因是，可以直接从表单参数中构造模型对象：

```
@order = Order.new(params[:order])
```

如果觉得这一行看起来很眼熟，那是因为已经在此前看到过了。它出现在 Depot 应用程序 `orders_controller.rb` 文件中。

David 说

抛出还是不抛出异常？

当使用主键作为参数的查询器时，你正在寻找特定的记录。你期望它存在。调用 `Person.find(5)` 方法是基于对 `people` 表的了解。其本意是想要得到 `id` 等于 5 的行记录。如果调用没有成功（比如已经删除了 `id` 为 5 的行记录），那么这意味着处于异常情况。结果 Rails 调用类 `RecordNotFound`，即抛出 `RecordNotFound`。

另外，使用查询器的搜索原则是为了寻找到相匹配的内容。因此，`Person.where(:name => 'Dave').first` 相当于告诉数据库（作为一个黑盒子）：“给我第一个且其名为 Dave 的 `Person` 对象。”这表现出一种明显不同的检索方法，事先并不能肯定会得到结果。搜索结果为空也是完全有可能的。因此，当查询器搜索单行记录时，返回 `nil`；而当搜索多行记录时，返回空数组。在这些情况下，没有异常响应是很自然的。

19.3.2 读取已有行记录

要想从数据库中读取行记录，首先要详细列举你具体感兴趣行的数据——给出 Active Record 的某些搜索条件，Active Record 将会返回包含行记录数据的且符合条件的对象。

要想在表中找出行记录，最简单的方法是列出主键。每个模型类都提供了 `find` 方法，它接收一个或几个主键值。如果只给定一个主键，那么它就会返回包含相应行记录数据的一个对象（或抛出类 `RecordNotFound` 异常）。如果是给定几个主键值，那么函数 `find()` 返回相应对象的数组。值得注意的是，如果无法找到其中任何一个 `id`，那么就会抛出类 `RecordNotFound` 异常（所以，如果调用方法返回而没有抛出错误，那么数组的长度将等于作为参数传递的 `id` 个数）：


```
an_order = Order.find(27) # find the order with id == 27

# Get a list of product ids from a form, then
# sum the total price
product_list = params[:product_ids]
total = Product.find(product_list).sum(&:price)
```

不过，在通常情况下，读取行记录，需要的是搜索条件，而不是它们的主键值。Active Record 提供了这些查询的大量选项。下面将先了解另外一种用简单 where 子句表达的方式，然后进一步优化 Rails 所生成的查询方式。

19.3.3 动态查询器

最常见的数据库搜索方法是，给定与字段相匹配的一个值，其结果是返回一行或多行记录。可能有这样的查询，如要求“返回所有‘Dave’的订单”或者“得到所有主题为‘Rails Rocks’的博客文章”。在许多其他语言和框架中，需要构造 SQL 查询来进行这些搜索。利用 Ruby 语言的动态功能，Active Record 模块也可以做到这一点。

例如，Order 模型有姓名、电子邮件地址和地址等属性。给定与字段相匹配的某些值，就可以使用带有这些属性名的查询器方法，达到返回行记录的目的：

```
e1/ar/find_examples.rb
```

```
order = Order.find_by_name("Dave Thomas")
orders = Order.find_all_by_name("Dave Thomas")
orders = Order.find_all_by_email(params['email'])
```

如果调用模型类方法，如以 find_by_、find_last_by_ 或 find_all_by_ 方法开始的名称，那么，可以借助方法名称的后面部分来确定所选中的字段，Active Record 模块将其转换为另一种查询。实例如下：

```
order = Order.find_by_name("Dave Thomas")
```

使用 Active Record，实际上可以把上面代码转换为如下代码：

```
order = Order.where(:name => "Dave Thomas").first
```

同样，调用 find_all_by_xxx 和 find_last_by_xxx 分别由调用 all 和 last 所取代，其中函数 first 是隐式调用的。

在函数 find_by_call 之后添加感叹号(!)表示：当没有查询到匹配结果时，不返回 nil，而将抛出 ActiveRecord::RecordNotFound 类的异常。

```
order = Order.find_by_name!("Dave Thomas")
```

神奇的功能远不止这些。Active Record 还会创建搜索多字段的查询器。例如，编写这样的代码：

```
user = User.find_by_name_and_password(name, pw)
```


等价于如下的代码：

```
user = User.where(:name => name, :password => pw).first
```

以 `find_by_` 或 `find_all_by_` 开始的函数名的后面包含字符串 “_and_”，对于该字符串前后的名称，为了确定查询的字段名，Active Record 模块简单地将它们拆分为字段名。在大多数情况下，这种表达方法是可行的，但是如果字符名为 `tax_and_shipping`，那么就不行了。在这种情况下，就必须使用其他方法来构造 `where` 子句。

有时候，要始终确保数据库表与模型对象进行交互。如果在数据库中没有这样的表，那么就要创建它。存在以 `find_or_initialize_by_` 或者 `find_or_create_by_` 名称开始的函数，在正常情况下，调用它们就是调用相应的模型函数 `new` 和 `create` (`find_or_initialize_by_` 和 `find_or_create_by_` 都是函数 `new` 和 `create` 的一种变体函数。——译者注)，否则将返回 `nil`。一旦将新的模型对象初始化，就可以定义其属性相应的查询器条件，其属性值也可传递到该查询方法，此外，要是使用了 `create` 的变体函数，其数据就已经存到数据库中。

```
cart = Cart.find_or_initialize_by_user_id(user.id)
cart.items << new_item
cart.save
```

而注意到，以 `find_by_` 形式开始的函数名称在字符名之间不是使用 “_or_”，而是使用 “_and_”。

19.3.4 SQL 语言与 Active Record 模块

为了说明如何使用 Active Record 模块与 SQL 语言进行交互，下面来比较一下，用简单字符串作为参数调用 `where` 方法与调用 SQL 语言 `where` 子句之间的对应关系。例如，要返回名字为 “Dave” 和支付类型为 “po” 的所有订单列表，可以这样做：

```
pos = Order.where("name = 'Dave' and pay_type = 'po'")
```

其结果将是包含所有相匹配行记录的 `ActiveRecord::Relation` 对象，每一个行记录包含了一个 `Order` 对象。

虽然搜索条件是预先定义好的，但是如何处理设置来自外部（也许是从 Web 表单中）的消费者名字？一种方式是用变量值替换条件字符串。

```
# get the name from the form
name = params[:name]
# DON'T DO THIS!!!
pos = Order.where("name = '#{name}' and pay_type = 'po'")
```

就这个建议而言，算不上是一个好主意。因为这样做造成了数据库完全敞开，使得所谓的 SQL 语句侵入攻击成为可能，在 18.1 节所生成的 Rails 指南中，有更详细的描述。可以试想，把来自外部的字符串代替到 SQL 语句中，实际上就是把整个数据库向网络世界公开。

与此相反，生成动态 SQL 语句的一种安全方式是，让 Active Record 处理它。要实现这一点，就要让 Active Record 生成合适转义的 SQL 语句，这是预防 SQL 语句侵入攻击的措施。下面来看看这是如何工作的。

若将多个参数传递到调用函数 `where`。那么 Rails 使用第一个参数作为生成 SQL 语句的模板。在该 SQL 语句中，可以嵌入占位符，在实时运行时这些占位符将由第二个参数（数组）来代替。

指定占位符的方法之一是在 SQL 语句中插入单个或几个单个问号标记^①。第一个问号标记所取代的是数组的第二个元素，接下来的问号是数组的第三元素，依此类推。^②

```
name = params[:name]
pos = Order.where(["name = ? and pay_type = 'po'", name])
```

还可以使用使用符号（Symbol 类）形式命名的占位符。第一个参数作为生成 SQL 语句的模板，把形式为“`:name`”作为 SQL 语句的占位符，而第二个参数是 Hash 类型的对象，其键是占位符名，其值是与位符名相对应的。

```
name = params[:name]
pay_type = params[:pay_type]
pos = Order.where("name = :name and pay_type = :pay_type",
  {:pay_type => pay_type, :name => name})
```

进一步，由于 `params` 事实上是 Hash 变量，可以简单地把所有条件传递给它。如果有一个可用于输入搜索条件的表单，那么就可以直接使用从表单中返回的这个 Hash 变量值。

```
pos = Order.where("name = :name and pay_type = :pay_type",
  params[:order])
```

紧接着可作更深入地探讨。如果给 Hash 变量传递这个条件，那么 Rails 生成 `where` 子句，使用字符名作为 Hash 变量键，而相应的传递值作为 Hash 变量值。因此，可以把前面的代码写得更简洁。

```
pos = Order.where(params[:order])
```

需要小心对待后面所说的表单条件：在 Rails 内部构造这个条件时，将会使用 Hash 变量的所有键/值对。另一种方法是明确指定使用该参数：

```
pos = Order.where(:name => params[:name],
  :pay_type => params[:pay_type])
```

无论使用哪一种形式的占位符，Active Record 都会非常谨慎地对待构造 SQL 语句值的引号和转义。使用这些动态 SQL 语句和 Active Record，能够防御入侵攻击。

19.3.5 使用 like 查询子句

下面尝试了解这样的代码，即在查询条件中使用参数化的 like 子句。

```
# Doesn't work
User.where("name like '%?'", params[:name])
```

① 原文：one or more question marks（一个或者几个问号标记）。每一个问号标记必须都是单一符号。——译者注

② 传递给 `where` 函数的第一个参数有三种可能性：字符串（String 类）、数组（Array 类）和散列（Hash 类）。请看源代码：<http://goo.gl/PUHGp>。——译者注

Rails 既不解析 SQL 语句里的条件,也不允许在占位符前后增加除此之外的内容。为了解决这个问题,可以在参数 `name` 的值前后添加额外的引号内容。正确的做法是,构建完整 `like` 子句的参数,并将其传递到该搜索条件中:

```
# Works
User.where("name like ?", params[:name]+"%")
```

当然,如果要做到这一点,需要考虑到一些特殊字符,如百分号(`%`)。如果这些符号出现在参数 `name` 的值中,那么就应该使用通配符(`#`)[⊖]。

19.3.6 构造返回记录的子集

现在已经知道如何指定查询条件,下面把注意力集中到 `ActiveRecord::Relation` 支持的各种方法,让我们开始吧。

你可能已经猜到了,函数 `first` 是返回关系集的第一行记录。如果关系集是空的,那么就返回 `nil`。同样,函数 `all` 是返回作为数组类型的所有行记录。`ActiveRecord::Relation` 支持许多类 `Array` 对象的方法,如函数 `each` 和 `map` 等。它们都是先隐式地调用函数 `all`。

在使用这些方法的任何一个之前,知道查询没有赋值是重要的。以不同的方法,即在开始调用它之前,借助于调用附加方法,修改查询是可能的。下面看看这些方法。

1. order 方法

要想以任何特定方式进行排序,返回 SQL 语句行记录,除非明确地把 `order by` 子句添加到查询中。`order` 方法允许指定查询条件,其参数是由 `order by` 子句的一些关键词组成。例如,下面查询将返回所有“Dave”的订单,首先按照付款类型升序排列,然后由发货日期降序排序:

```
orders = Order.where(:name => 'Dave').
  order("pay_type, shipped_at DESC")
```

2. limit 方法

通过调用 `limit` 方法,可以限定返回行记录的个数。一般来说,当使用 `limit` 方法,很可能还需要指定排列顺序,以确保得到一致结果。例如,下面返回前十个相匹配的订单:

```
orders = Order.where(:name => 'Dave').
  order("pay_type, shipped_at DESC").
  limit(10)
```

3. offset 方法

`offset` 方法进一步补充了 `limit` 方法。它允许返回在结果集中我们指定的第一行的偏移量。

```
# The view wants to display orders grouped into pages,
# where each page shows page_size orders at a time.
# This method returns the orders on page page_num (starting
# at zero).
def Order.find_on_page(page_num, page_size)
  order(:id).limit(page_size).offset(page_num*page_size)
end
```

⊖ 如代码: `"#{params[:name]}%"`。——译者注

可以把函数 `offset` 与函数 `limit` 一起使用，每一次查询得到下 `n` 行记录结果。

4. select 方法

在默认情况下，`ActiveRecord::Relation` 从底层数据库中获取所有字段名——它分配 `select * from...` 语句到数据库中。使用 `select` 方法重写 `select` 语句，该方法需要在 `SELECT` 语句的 “*” 位置上出现一个字符串。

这种方法能够限制返回值，例如只需要表中的子集数据。如在播客表中可能包含标题、主播人和日期信息等，且也可能还包含较大 MP3 的 BLOB 类型字段。如果只是开始建立节目列表，那么把每行声音数据也加载上来，这将是低效的方法。`select` 方法可以有机会选择加载那些字段。

```
list = Talk.select("title, speaker, recorded_on")
```

5. joins 方法

`joins` 方法允许指定一组相关表加入到默认表。把其参数直接插入到 SQL 语句中，它是处在模型表名之后并且在第一个参数所指定的任何条件之前[⊖]。`join` 语法是数据库相关的。下列代码返回所有书名为 “Programming Ruby” 的商品项目清单。

```
LineItem.select('li.quantity').
  where("pr.title = 'Programming Ruby 1.9'").
  joins("as li inner join products as pr on li.product_id = pr.id")
```

6. readonly 方法

`readonly` 方法的作用是，不能把 `ActiveRecord::Resource` 返回的 `Active Record` 对象存储到数据库中。

如果使用 `joins` 或 `select` 方法，那么将对象自动标记为 `readonly`。

7. group 方法

`group` 方法把 `group` 子句添加到由查询器所生成的 SQL 语句中：

```
summary = LineItem.select("sku, sum(amount) as amount").
  group("sku")
```

8. lock 方法

`lock` 方法接收一个可选的字符串作为参数。如果把字符串传递给它，那么它应该是数据库语法的 SQL 代码段，且同时指定了一种锁。例如，使用数据库 MySQL，共享模式锁提供了行记录的最新数据，并且当持有该锁时，保证没有人可以改变该行。例如，只有在账户上有足够的资金时，才可扣除账户款，可以编写如下代码：

```
Account.transaction do
  ac = Account.where(:id=>id).lock("LOCK IN SHARE MODE").first
  ac.balance -= amount if ac.balance > amount
  ac.save
end
```

⊖ 这里是在讨论将要生成的 SQL 语句。——译者注

如果没有指定字符串值或给定锁 `true` 值, 则获得数据库的默认独占锁 (通常这将是 “更新”)。经常可以使用事务来消除需要这种锁定 (将从 19.5 节开始讨论)。

比起简单地查询和可靠地检索数据, 数据库可以做更多的事情, 比如一些数据归纳分析。Rails 也提供这些访问的方法。

19.3.7 获取字段统计

Rails 有能力完成字段值的统计工作。例如, 对于给定的订单表而言, 就可以计算出以下内容:

```
average = Order.average(:amount) # average amount of orders
max      = Order.maximum(:amount)
min      = Order.minimum(:amount)
total    = Order.sum(:amount)
number   = Order.count
```

所有这些方法都对应于基础数据库的聚合函数, 但是它们以一种独立于数据库的方式工作。

可与之前的方法相结合:

```
Order.where("amount > 20").minimum(:amount)
```

这些函数都是合计值。在默认情况下, 它们返回单一结果, 例如, 为满足某些订单条件产生最小订单金额。不过, 如果包括 `group` 方法, 与此相反会产生一组结果, 其结果存在每一分组记录, 而每一分组表达式具有相似的值。例如, 下面计算每个州 (字段名为 `state`) 的最大销售金额:

```
result = Order.maximum(:amount).group(:state)
puts result #=> [{"TX", 12345}, {"NC", 3456}, ...]
```

此代码返回一个有序 “散列”^①。使用分组元素 (在例子中是 “TX”、“NC” 等) 作为索引。还可以使用 `each` 函数, 按顺序遍历每一个分组。每个分组包含了聚合函数值。

当使用 `group` 方法时, 也可以引入 `order` 和 `limit` 方法一起使用。

例如, 下面返回三个州最高金额的订单, 并且按照订单金额大小进行排序:

```
result = Order.group(:state).
  order("max(amount) desc").
  limit(3)
```

此代码是数据库相关的——由于排序需要聚合的字段名, 因此不得不使用 SQLite 语法的聚合函数 (在这种情况下是 `max` 函数)。

19.3.8 范围函数

由于这些方法调用链^②变得更长了, 所以使得链本身的重用性成为关注的问题。Rails 又一次提供解决这种问题的可能性。可以把 `Proc` 类与 Active Record 模块的 `scope` 函数结合使用, 由此该函数可能有如下代码的参数 (其中 `lambda` 参见 4.7 节):

① 具有散列类型的特点, 但它们的数据类型是 `Array`。——译者注

② 使用大量点操作把函数链接起来。——译者注

```
class Order < ActiveRecord::Base
  scope :last_n_days, lambda { |days| where('updated < ?', days) }
end
```

命名这样的范围（scope）函数可以十分轻松地找到上周的订单值：

```
orders = Order.last_n_days(7)
```

更为简单的范围函数可以方便地调用一组方法：

```
class Order < ActiveRecord::Base
  scope :checks, where(:pay_type => :check)
end
```

也可以把不同的范围函数组合在一起使用。找到上周且支付类型为“check”的订单值，这是十分容易的。

```
orders = Order.checks.last_n_days(7)
```

除此之外，也使得应用程序代码更易于编写和阅读，scope 函数可以使代码更高效。例如，前面的语句是作为单一的 SQL 查询语句执行。

ActiveRecord::Relation 对象是相当于匿名 scope 函数：

```
in_house = Order.where('email LIKE "%@pragprog.com"')
```

当然，这些对象也可以与范围函数组合在一起使用：

```
in_house.checks.last_n_days(7)
```

范围函数没有限制 where 条件；通过调用 limit、order 和 join 函数等，几乎可以任何事情。必须指出，Rails 无法处理多个 order 或者 limit 子句，因此必须确保每一次调用链只使用它们一次。

在大部分情况的查询中，这些方法已经足够用了。但是，Rails 并不能够完全满足处理所有查询，在这样的情况下，需要自己写查询语句，对此存在这样的 API。

19.3.9 编写自己 SQL 语句

前面介绍的每一种方法都有助于构建完整的 SQL 查询字符串。方法 find_by_sql 完全能够控制应用程序。它接收包含 SQL 语言 select 语句（或一个包含 SQL 语句及其占位符值的数组，像函数 find 一样）的单个参数，并返回来自结果集合中模型对象（可能为空）的数组。将查询返回字段值设置为这些模型的属性。在通常情况下，可以使用“select *”形式的语句返回表的所有列，但这是不必需的。

```
el/ar/find_examples.rb
```

```
orders = LineItem.find_by_sql("select line_items.* from line_items, orders " +
                             " where order_id = orders.id " +
                             " and orders.name = 'Dave Thomas' ")
```

仅仅由查询所返回的那些属性将应用于结果模型对象。使用函数 attributes、attribute_names 和 attribute_present?，可以判断模型对象的这些属性是否有效。第 1 行返回了属性

名/值对的 Hash 对象, 第 2 行返回由 name 属性值组成的数组, 并且, 如果在这个模型对象中 name 属性是有效的, 则第 3 行返回 true。

```
e1/ar/find_examples.rb
```

```
orders = Order.find_by_sql("select name, pay_type from orders")
first = orders[0]
p first.attributes
p first.attribute_names
p first.attribute_present?("address")
```

David 说

但 SQL 语句并不是灰色的 (drity) 吗?

自从开发者第一次用面向对象层封装关系数据库起, 人们就已经开始辩论持久层抽象要达到怎样程度的封装问题。有些对象关系映射器的查询可以完全不使用 SQL 语言, 这迫使所有的查询通过面向对象层完成, 寄希望于面向对象的纯洁性。

Active Record 模块并没有这样做。当时它是以这一概念建立起来的, 即 SQL 语言既不是灰色的也不是有缺陷的, 只是在很简单的查询情况下太累赘。其重点是不再需要处理那些查询的累赘 (对于任何程序员而言, 通过手工编写含十个属性的 insert 语句, 这样的工作是一件令人厌烦的事情), 但同时继续保持硬查询[⊖]的可表达性——充分利用曾经创建的传统 SQL 语句。

因此, 当使用函数 find_by_sql 处理性能瓶颈或硬查询时, 你不应该感到内疚。使用面向对象接口的出发点, 是为了提高生产力和让人身心皆愉悦, 并且随着问题更加深入和复杂, 你就需要这样做。

此代码将产生以下内容:

```
{"name"=>"Dave Thomas", "pay_type"=>"check"}
{"name", "pay_type"}
false
```

函数 find_by_sql 也可用于创建包含派生字段数据的模型对象。如果使用 “as xxx” 的 SQL 语法来作为派生字段结果集合的名称, 那么将把这个名称用作属性名。

```
e1/ar/find_examples.rb
```

```
items = LineItem.find_by_sql("select *,
                             \" products.price as unit_price, \" +
                             \" quantity*products.price as total_price, \" +
                             \" products.title as title \" +
                             \" from line_items, products \" +
                             \" where line_items.product_id = products.id ")
li = items[0]
puts "#{li.title}: #{li.quantity}x#{li.unit_price} => #{li.total_price}"
```

⊖ 即使用传统的 SQL 语言查询。——译者注

至于查询条件，也可以把数组传递到函数 `find_by_sql`，其中第一个元素是包含占位符的字符串。数组的其余部分可以是一个散列或一组要替换的值。

```
Order.find_by_sql(["select * from orders where amount > ?",
                  params[:amount]])
```

在 Rails 的早期版本中，经常使用 `find_by_sql` 进行重新排序。从那时起，所有添加到 `find` 方法的选项，将意味着能够避免重新排序这种低级方法。

19.3.10 重新加载数据

多个进程（或多个应用程序）可能同时访问包含数据库的应用程序，这样得到的模型对象已不再是最新的，这种可能性总是存在的——因为有人可能已经把更新的数据写入数据库中。

从某种程度上说，事务支持（在 19.5 节描述）解决了这类问题。不过，需要手动刷新模型对象，这种事情还是会时而发生。Active Record 模块使之变得简单，简单地调用其重载方法，且将从数据库中刷新对象属性：

```
stock = Market.find_by_ticker("RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end
```

在实践中，除了单元测试外很少使用重载。

19.3.11 更新现有行记录

在这样长时间的讨论查询器（finder）方法之后，你会很高兴了解到，要探讨很多关于 Active Record 模块的更新记录方法。

如果存在 Active Record 对象（也许表示订单表行记录），那么通过调用其 `save` 方法，就可以把它写入数据库中。如果先前从数据库中读取该对象，那么此时保存对象将更新现有的行记录，否则，这种保存将是插入新行记录。

如果更新现有行记录，那么 Active Record 模块将利用其与内存对象相匹配的主键字段。在 Active Record 模块对象中的属性决定这些将要更新的字段——只有改变了字段值，才能在数据库中更新该字段。在下面的例子中，对于订单 id 为 123，在数据库表中可以更新该行记录的所有属性值：

```
order = Order.find(123)
order.name = "Fred"
order.save
```

不过，在下面的例子中，Active Record 模块对象仅仅包含了这些属性 `id`、`name` 和 `paytype`——仅当保存该对象时，才可以更新这些字段。（请注意，如果打算保存由方法 `find_by_sql` 获取的行记录，那么必须包括 `id` 字段。）

```
orders = Order.find_by_sql("select id, name, pay_type from orders where id=123")
first = orders[0]
first.name = "Wilma"
first.save
```

除了 save 方法之外, 利用 Active Record 模块, 也可以改变属性值, 且以单一调用函数 update_attribute 方式保存模型对象:

```
order = Order.find(123)
order.update_attribute(:name, "Barney")
order = Order.find(321)
order.update_attributes(:name => "Barney",
                       :email => "barney@bedrock.com")
```

在控制器操作中, 最常用的方法就是 update_attributes, 它把表单数据合并到现有数据库行记录:

```
def save_after_edit
  order = Order.find(params[:id])
  if order.update_attributes(params[:order])
    redirect_to :action => :index
  else
    render :action => :edit
  end
end
```

使用类的方法 update 和 update_all, 可以把读取行记录和更新它的函数合并起来。这种更新方法需要 id 参数和一组属性。它先获取相应的行记录, 然后更新给定的属性, 再把结果保存到数据库中, 最后返回该模型对象。

```
order = Order.update(12, :name => "Barney", :email => "barney@bedrock.com")
```

首先可以给函数 update 传递 id 数组和属性值散列的数组, 然后更新数据库中所有相应的行记录, 最后返回一个模型对象数组。

最后, 利用类方法 update_all, 可以列出在 SQL 语言 update 语句中的 set 和 where 子句条件。例如, 下面的代码把所有在标题中含 Java 商品的价格提高 10%:

```
result = Product.update_all("price = 1.1*price", "title like '%Java%'")
```

方法 update_all 的返回值取决于数据库适配器, 大多数数据库系统 (但不包括 Oracle) 返回在数据库中已更改的行记录数。

19.3.12 方法 save、save!、create 和 create!

现在把注意力转移到 save 和 create 方法的两种版本上。它们报告错误的方式是不同的:

- 如方法 save 成功地保存了行记录, 则返回 true; 否则, 返回 nil。
- 如方法 save! 成功地保存了行记录, 则返回 true; 否则, 返回异常。
- 不管方法 create 是否成功地创建了 Active Record 模块对象, 都返回该对象。如果想要确定是否写入数据, 那么就需要检查该对象的错误有效性。

- 如方法 `create!` 成功地创建了 Active Record 模块对象，则返回该对象；否则，返回异常。

下面更详细地讨论一下。

如果模型对象是有效的，且是可保存的，那么方法 `save` 返回 `true`：

```
if order.save
  # all OK
else
  # validation failed
end
```

不仅要检查每次调用方法 `save` 的结果，而且还要查看其期望结果是否正确，这是理所当然的。之所以 Active Record 模块不是那样苛刻，是因为在控制器的相关行为方法中调用方法 `save`，并且在视图代码中将任何错误反馈给最终用户。对于许多应用来说，都应该这样做。

不过，在需要保存模型对象，要求以编程方式确定处理所有错误的情况下，就应该使用方法 `save!`。如果这种方法无法保存该对象，那么该方法会出现异常类 `RecordInvalid`：

```
begin
  order.save!
rescue RecordInvalid => error
  # validation failed
end
```

19.3.13 删除行记录

Active Record 模块支持两种风格的行删除。第一种形式是两个类层面的方法 `delete` 和 `delete_all`，其操作层面是在数据库上的。方法 `delete` 接收单个 id 或多个 id 的数组，且删除基础表中相应行。方法 `delete_all` 删除给定条件的行记录（或如果没有指定条件则就是所有行记录）。调用两个函数的返回值取决于数据库适配器，但通常情况下是所涉及的行记录数。如果调用之前该行就不存在，则不会抛出异常。

```
Order.delete(123)
User.delete([2,3,4,5])
Product.delete_all(["price > ?", @expensive_price])
```

与上面不同的方法 `destroy` 是 Active Record 模块提供行记录删除的第二种形式。所有这些方法是 Active Record 模型对象层面的操作。

首先，`destroy` 实例方法删除数据库中对应特定模式对象的行记录。然后，凝固该对象的内容，防止以后改变其属性。

```
order = Order.find_by_name("Dave")
order.destroy
# ... order is now frozen
```

存在两个类层面的销毁方法：`destroy` 方法（这需要 id 或多个 id 的数组）和 `destroy_all`（采用一个条件）。它们两者把数据库表中相应的行记录读取到模型对象，并调用这些对象的实例层面销毁方法。两个方法都返回无意义的事情。

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

为什么既需要 `delete` 又需要 `destroy` 类的方法呢？`delete` 方法绕过了 Active Record 模块的各种回调和验证函数，而 `destroy` 方法确保调用它们的所有一切过程。在一般情况下，如果要确保数据库是一致的，并且根据模型类中所定义的商业规则，那么最好使用 `destroy` 这类方法。

在第 7 章中已经阐述过验证性。接下来将探讨回调技术。

19.4 干预跟踪进程

Active Record 模块控制了模型对象的生命周期——创建它们；在修改时跟踪、储存和更新它们；在删除时观察它们。使用回调，Active Record 模块允许用户代码加入干预这一跟踪进程中。在对象生命周期中可以编写获取调用任何有效事件的代码。有了这些回调，可以完成复杂有效性验证；在字段值穿梭于数据库时映射它们；甚至还可以阻止完成某些操作。

Active Record 模块定义了 20 个回调方法。其中 18 个方法是以 `before` 或者 `after` 为前缀形式，且把 Active Record 对象的一些操作视为同一类。例如，在调用 `destroy` 方法之前，将调用 `before_destroy` 回调方法，而之后将调用 `after_destroy` 方法。另外两个方法是 `after_find` 和 `after_initialize`，它们没有相应的 `before_xxx` 回调方法。（将在后面探讨这两个回调方法在其他方式上的不同之处。）

在图 19.2 中，可以看到，Rails 围绕着针对模型对象的创建、更新和销毁等基本操作，打造出了这 18 个回调方法。也许令人惊讶的是，并没有严格地嵌套之前和之后的有效性验证调用。

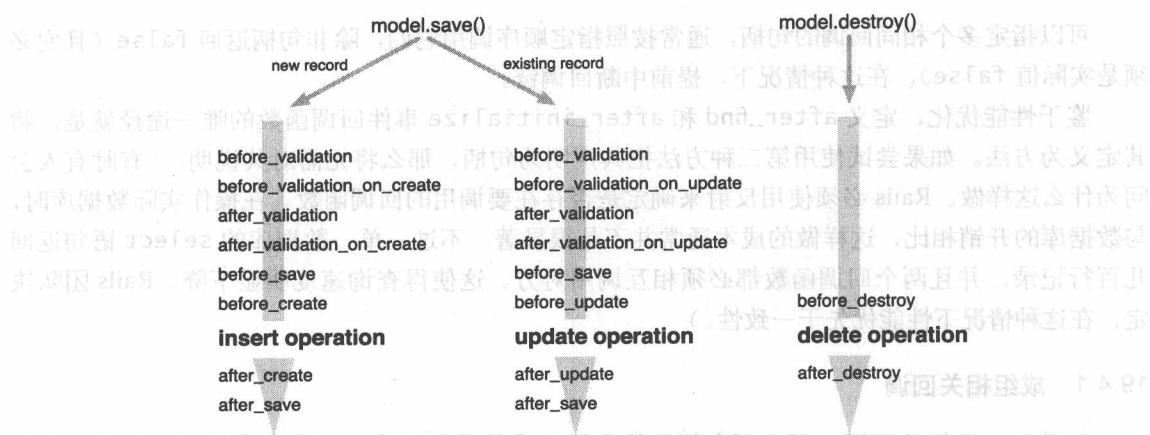


图 19.2 Active Record 回调顺序

除了这 18 个调用函数之外，在任何 `find` 操作函数之后，可调用 `after_find` 回调函数，且在创建 Active Record 模型对象之后，可调用 `after_initialize` 回调函数。

为了在回调期间执行用户代码，需要写一个句柄，并与相应回调函数关联在一起。

实施回调有两种基本方式。

第一种基本的方法：可直接定义回调实例方法。如果想处理“保存之前”的事件，例如，可这样写：

```
class Order < ActiveRecord::Base
  # ..
  def before_save
    self.payment_due ||= Time.now + 30.days
  end
end
```

定义回调函数的第二种基本方式是：可声明句柄。句柄可以是方法或代码块。使用事件后命名的类方法，可把句柄与特定事件相关联。要关联方法，声明其为 `private` 或 `protected`，并把其作为符号的名字指定为句柄声明。要指定代码块，只需将其添加到声明之后。此代码块接收模型对象作为参数。

```
class Order < ActiveRecord::Base
  before_validation :normalize_credit_card_number
  after_create do |order|
    logger.info "Order #{order.id} created"
  end
  protected

  def normalize_credit_card_number
    self.cc_number.gsub!(/[^\s]/, ' ')
  end
end
```

可以指定多个相同回调的句柄。通常按照指定顺序调用它们，除非句柄返回 `false`（且它必须是实际值 `false`），在这种情况下，提前中断回调链。

鉴于性能优化，定义 `after_find` 和 `after_initialize` 事件回调函数的唯一途径就是，将其定义为方法。如果尝试使用第二种方法把其声明为句柄，那么将无需对其说明。（有时有人会问为什么这样做。Rails 必须使用反射来确定是否存在要调用的回调函数。在操作实际数据库时，与数据库的开销相比，这样做的成本通常并不是很显著。不过，单一数据库的 `select` 语句返回几百行记录，并且两个回调函数都必须相互调用对方。这使得查询速度明显下降。Rails 团队决定，在这种情况下性能优先于一致性。）

19.4.1 成组相关回调

如果有一组相关回调，那么可方便地使之组合成单独句柄类，可在多个模型之间共享这些句柄。句柄类是定义回调方法（如 `before_save`、`after_create` 等）的简单类。在 `app/models` 目录下创建这些句柄类的源代码文件。

在使用句柄的模型对象中，可以创建此句柄类的实例，并把该实例传递到各种回调声明。下面几个例子能说得 clearer 一些。

如果在应用程序中多处使用信用卡，可能要在多个模型中共享 `normalize_credit_card_number` 方法。要做到这一点，将该方法移到其自己类中，且在需要它去处理事件之后，给出其名称。这种方法将接收单一参数，即生成回调的模型对象。

```
class CreditCardCallbacks
```

```
  # Normalize the credit card number
  def before_validation(model)
    model.cc_number.gsub!(/[-\s]/, '')
  end
end
```

现在，在模型类中可以为调用这种共享回调做准备：

```
class Order < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
```

```
class Subscription < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
```

在这个例子中，句柄类假设信用卡号码存放在模型属性名为 `cc_number` 下；`Order` 类和 `Subscription` 类将都有该名称的属性。但可以推广一下这个想法，句柄类很少取决于使用它的类实现细则。

例如，可以创建通用的加密和解密句柄。其用途可以是，在把已命名的字段存储到数据库之前，对其进行加密，而当读取行记录时，对其进行解密。在任何需要这种功能的模型中，可以将其作为回调句柄引用。

在模型数据写入数据库之前，句柄需要加密该模型的一组给定属性。因为应用程序需要处理这些属性的纯文本版本，所以它在完成保存后为其再次进行解密做准备。对此，当从数据库读取一行记录到模型对象时，它也需要解密数据。这些需求意味着，必须处理 `before_save`、`after_save` 和 `after_find` 事件。因为需要解密数据库的行记录，既在保存后又在找到新行记录时，所以通过给 `after_find` 方法别名为 `after_save` 方法可节省代码——一个方法有两个名字。

```
el/ar/encrypt.rb
```

```
class Encrypter
```

```
  # We're passed a list of attributes that should
  # be stored encrypted in the database
  def initialize(attrs_to_manage)
    @attrs_to_manage = attrs_to_manage
  end
  # Before saving or updating, encrypt the fields using the NSA and
  # DHS approved Shift Cipher
  def before_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("a-z", "b-za")
    end
  end
end
```

```
  # After saving, decrypt them back
  def after_save(model)
```



```
@attrs_to_manage.each do |field|
  model[field].tr!("b-za", "a-z")
end
end

# Do the same after finding an existing record
alias_method :after_find, :after_save
end
```

这个例子应用了简单加密方法——在真正使用这个类之前，应该要对其进行强化。现在可为从订单模型内调用 `Encrypter` 类做准备：

```
require "encrypter"

class Order < ActiveRecord::Base
  encrypter = Encrypter.new([:name, :email])

  before_save encrypter
  after_save encrypter
  after_find encrypter

  protected
  def after_find
  end
end
```

该程序创建了新的 `Encrypter` 对象，并把它挂到事件 `before_save`、`after_save` 和 `after_find`。这样，在保存订单对象之前，将调用针对对象 `encrypter` 的方法 `before_save`，其他也类似。

那么，为什么要定义空的 `after_find` 方法呢？还记得前面曾经说过，出于性能原因，特殊处理了函数 `after_find` 和 `after_initialize`。这种特殊处理的后果之一是，`Active Record` 并不知道调用 `after_find` 句柄，除非它发现模型类中实际存在的 `after_find` 方法。必须定义一个空占位符以便触发函数 `after_find` 过程。

这样做固然很好，但每一个模型类要使用加密句柄，都需要引用这样 8 行代码，就像 `Order` 类曾经所使用的那样。我们可以做得更好一些。定义完成所有这些工作的一个帮助方法，且将其提供给所有 `Active Record` 模型。要做到这一点，可以将其添加到 `ActiveRecord::Base` 类：

```
el/ar/encrypt.rb

class ActiveRecord::Base
  def self.encrypt(*attr_names)
    encrypter = Encrypter.new(attr_names)

    before_save encrypter
    after_save encrypter
    after_find encrypter

    define_method(:after_find) { }
  end
end
```


鉴于这一代码，现在可使用单个调用，把加密方法添加到任何模型类的属性中。

```
e1/ar/encrypt.rb

class Order < ActiveRecord::Base
  encrypt(:name, :email)
end
```

用上面代码做个简单的驱动程序实例：

```
e1/ar/encrypt.rb

o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email = "dave@example.com"
o.save
puts o.name

o = Order.find(o.id)
puts o.name
```

在终端控制台上，看到模型对象的消费者姓名（纯文本）：

```
ar> ruby encrypt.rb
Dave Thomas
Dave Thomas
```

不过，在数据库中，姓名和电子邮件地址都是通过工业强度加密遮蔽的：

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders"
id = 1
user_id =
name = Dbwf Tipnbt
address = 123 The Street
email = ebwf@qsbhqspsh.dpn
```

回调是一门优秀的技术，但它们有时可能产生模型类的结果，其结果并不是模型本身所固有的。如在 19.4 节中所示，当创建订单对象时，创建了产生日志消息的回调。这个功能并不真是基本 Order 类的一部分——之所以把它放在那里，是因为那是回调执行的地方。

Active Record 观察器克服了这个限制。

19.4.2 观察器

Active Record 的观察器（observer）是显式地链接到模型类本身的对象，其本身注册为回调，就好像它是模型的一部分，而不需要在模型本身产生任何改变。下面是使用观察器所编写的以前的日志实例：

```
e1/ar/observer.rb

class OrderObserver < ActiveRecord::Observer
  def after_save(an_order)
    an_order.logger.info("Order #{an_order.id} created")
  end
end
```

当 `ActiveRecord::Observer` 生成子类时，可以看到新的类名称，其结尾带字 `Observer`，并假定最后剩下的是待观察的模型类名。在这个例子中，调用了观察器类 `OrderObserver`，这样它本身会自动挂到模型 `Order` 上。

有时这种约定行不通。要做到这一点，可以使用 `observe` 方法，观察器类可以明确地列出想要观察的模型：

```
e1/ar/observer.rb
```

```
class AuditObserver < ActiveRecord::Observer

  observe Order, Payment, Refund

  def after_save(model)
    model.logger.info("[Audit] #{model.class.name} #{model.id} created")
  end
end
```

按照约定，观察器源代码文件是存放在 `app/models` 目录下。

19.4.3 观察器实例化

到目前为止已经定义了观察器。不过，还需要对它们进行实例化——如果不这样做，那么根本不会触发它们。实例化观察器取决于是否正在 Rails 应用程序的相关内部或外部中使用。

如果在 Rails 应用程序内使用观察器，那么需要在应用程序 `application.rb` 文件中列出它们（在 `config` 目录中）：

```
config.active_record.observers = :order_observer, :audit_observer
```

相反，如果在独立应用程序中正在使用 `Active Record` 对象（也就是说，没有在 Rails 应用程序内运行 `Active Record`），那么使用观察器对象就需要手动创建：

```
OrderObserver.instance
AuditObserver.instance
```

在某种程度上，观察器给 Rails 带来了第一代面向方面编程如 Java 语言的许多好处。允许把它们注入模型中的类，而没有改变这些类本身的任何代码。

19.5 数据库事务

数据库事务（`transaction`）把一系列数据库记录修改组合在一起，其处理方式，要么数据库接收所有改变，要么什么也不接收。需要事务的（在 `Active Record` 自己的文档中所使用的）典型例子是两个银行账户之间的资金转移。基本逻辑很简单：

```
account1.deposit(100)
account2.withdraw(100)
```

但必须小心。当存款成功而出于某种原因取款失败（也许客户透支了）时，会发生什么？这样将不得不在 `account1` 增加 100 美元达到平衡，而没有从 `account2` 上扣减相应款额。实际上凭空已经增加了 100 美元。

事务是一种援救工作。事务有点类似三剑客(Three Musketeers)座右铭“我为人人, 人人为我”(All for one and one for all)。在一个事务范围内, 或者每一个 SQL 语句都成功, 或者所有都没有效果。换言之, 如果任何一个语句失败, 那么整个事务对数据库就没有影响。

在 Active Record 中, 使用事务方法可以方便在数据库特定事务范围内执行一个代码块。在代码块结束时, 就提交事务, 进行数据库更新, 除非代码块抛出异常情况, 在这种情况下数据库可以回滚所有改变。由于事务存在于一个数据库连接范围内, 所以必须借助于作为接收器的 Active Record 类调用它们。

因此可以这样写:

```
Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

下面用事务做个试验。首先, 创建一个新的数据库表。(请确保数据库支持事务功能, 否则此代码将不会工作。)

```
e1/ar/transactions.rb
```

```
create_table :accounts, :force => true do |t|
  t.string :number
  t.decimal :balance, :precision => 10, :scale => 2, :default => 0
end
```

其次, 定义简单的银行账户类。这个类定义了实例方法, 以便在账户中存款而后提款。它还提供了一些基本的有效性验证——对于这个特定类型的账户, 余额永远不能是负数。

```
e1/ar/transactions.rb
```

```
class Account < ActiveRecord::Base
  validate :price_must_be_at_least_a_cent

  def withdraw(amount)
    adjust_balance_and_save(-amount)
  end

  def deposit(amount)
    adjust_balance_and_save(amount)
  end

  private

  def adjust_balance_and_save(amount)
    self.balance += amount
    save!
  end

  def price_must_be_at_least_a_cent
    errors.add(:balance, "is negative") if balance < 0
  end
end
```

看看帮助方法 `adjust_balance_and_save`。第一行只是更新余额字段。然后该方法调用 `save!` 方法，这样可保存模型数据。（请记住，如果不能保存对象，那么方法 `save!` 抛出异常——使用异常事务报告，可以了解出错的问题。）

现在可以编写在两个账户之间转账的代码。这是非常简单的：

```
e1/ar/transactions.rb
```

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")
```

```
Account.transaction do
  paul.deposit(10)
  peter.withdraw(10)
end
```

通过检查数据库，果然完成了转账：

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
```

```
id = 1
number = 12345
balance = 90
```

```
id = 2
number = 54321
balance = 210
```

现在进行更彻底的代码测试。再从头开始一次，但这一次尝试转账 350 美元，这次有效性验证规则不允许客户 Peter 进行转账。试试看：

```
e1/ar/transactions.rb
```

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")
```

```
e1/ar/transactions.rb
```

```
Account.transaction do
  paul.deposit(350)
  peter.withdraw(350)
end
```

当运行该代码时，在控制台上得到异常报告：

```
.../validations.rb:736:in 'save!': Validation failed: Balance is negative
from transactions.rb:46:in 'adjust_balance_and_save'
:
:
from transactions.rb:80
```

看看数据库，可得知，该数据保持不变：

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
```

```
id = 1
number = 12345
balance = 100
```

```
id = 2
number = 54321
balance = 200
```

不过，在这里有个容易犯错的地方。虽然事务防止了数据库不一致性，但是模型对象如何处理呢？要看到它们发生了什么事，就必须为拦截异常允许程序继续运行做准备：

```
el/ar/transactions.rb
```

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")
```

```
el/ar/transactions.rb
```

```
begin
  Account.transaction do
    paul.deposit(350)
    peter.withdraw(350)
  end
rescue
  puts "Transfer aborted"
end

puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
```

下面看到的代码令人感到有点惊讶：

```
Transfer aborted
Paul has 550.0
Peter has -250.0
```

虽然数据库丝毫没有改变，但是已经更新了模型对象。这是因为对于不同对象的前后状态，Active Record 没有保持跟踪——事实上这是不可能的，因为没有简便方法知道究竟在事务中调用了哪些模型。

内建事务

在 19.2.2 节中曾经讨论过父表和子表，当储存父表记录行时，Active Record 保存所有关联的子表行记录。这需要执行几个 SQL 语句（父表的一行记录和子表的每一个已改变的或新的行记录）。显然，这种改变应该是基元的，但到现在为止，当保存这些关联的对象时，并没有使用事务。是 Rails 有疏忽？

幸运的是，Rails 并没有疏忽。Active Record 是足够精明的，以致在事务中把所有更新和插入打包到特殊函数 `save`（就删除而言，是函数 `destroy`）；要么全部成功永久地写入到数据库中，要么没有数据写入。仅当自己管理几个 SQL 语句时，就需要有明确的事务。

虽然我们已经介绍了基础知识，但是事务其实很微妙。它们表现出所谓的 ACID 属性：它们是原子的（atomic），它们确保一致性（consistency），它们的工作孤立的（isolation），它们的效果是持久的（durable）（当永久致力于事务时）。如果打算使得数据库应用程序更具活力，找一本好的数据库图书并且阅读事务，这是值得的。

19.6 本章小结

首先，在本章中学习了相关的数据结构和命名表、类、字段、属性、主键属性 `id` 和模型关

第 20 章

行为调度和行为控制

在本章中，我们将学习：

- 表述性状态转移 (Representational State Transfer, REST)
- 定义如何使得请求路由到控制器
- 选择数据表述性
- 测试路由
- 控制器环境
- 呈现和转向
- 会话、闪存和过滤器

Action Pack 框架是 Rails 应用程序的核心部分。它由三个 Ruby 模块组成：ActionDispatch、ActionController 和 ActionView。Action Dispatch 模块使请求路由到控制器。Action Controller 模块使请求转换成响应。Action Controller 模块使用 Action View 模块使响应格式化。

在 Depot 应用程序的例子中，就曾经把根网站 (/) 路由到 StoreController 类的 index 方法。在执行完该方法后，再呈现到模板文件 app/views/store/index.html.erb。上述每一项活动都是由 ActionPack 模块组精心策划的。

当上述三个子模块一起工作时，它们提供处理输入请求和生成输出响应的支持。在本章中，将先介绍 Action Dispatch 模块和 Action Controller 模块。下一章将再介绍 Action View 模块。

在前面介绍 Active Record 时，曾将它视为是完全的独立库；当然，也可以将它用作非 Web 的 Ruby 应用程序的一部分。但 Action Pack 框架却不同。虽然它可以直接用作框架，但一般不会这种做。相反，可以利用 Rails 提供的紧密一体化功能。处理请求的过程使用一系列组件如 Action Controller 模块、Action View 模块和 Active Record 模块等，而 Rails 环境把它们编织成一个连贯的（且易于使用的）整体。出于这样的原因，这里将描述在 Rails 范围内的 Action Controller。首先，看看 Rails 应用程序是如何处理请求的。然后，将会深入路由和 URL 处理的细节。其次，将看看如何编写控制器代码。最后，将涵盖会话、闪存和过滤器三个方面话题。

20.1 分派请求到控制器

简单地讲，Web 应用程序从浏览器接收输入请求，对其进行处理，并发送生成的响应。

回顾一下，第一个问题是，应用程序如何知道输入请求做什么用呢？购物车应用程序将接收请求以显示目录、将商品项目添加到购物车和创建订单等。该应用程序又如何将这些请求路由到相应的代码呢？

原来，Rails 提供了定义路由请求的两种方式：一种综合方式是当需要时使用的；另一种便捷方式是在一般情况下经常使用的。

综合方式可定义基于模式匹配、需求和条件的行动的 URL 直接映射。便捷方式可定义基于资源的路由，如用户所定义的模型。因为便捷方式是以综合方式为基础的，所以可自由地搭配使用这两种方法。

在上述两种情况下，Rails 编码 URL 请求的信息，并使用 Action Dispatch 子系统，以确定这一请求应该做什么用。实际过程是非常灵活的，但在这一过程结束时，Rails 已经确定了处理这个特殊请求的控制器名，以及该请求其他任何的参数列表。在这一过程中，无论是这些额外的参数，还是 HTTP 方法本身，都是识别调用目标控制器的行为。

Rails 路由支持 URL 和行动之间的映射，它们是基于 URL 内容和调用请求所使用的 HTTP 方法。我们已经看到了如何以 URL 网址为基础使用匿名或命名路由进行操作。Rails 还支持更高层次方式，创建一组相关的路由。为了搞清楚这样做的目的，需要暂时先观察表述性状态转移 (Representational State Transfer, REST) 的世界。

20.1.1 REST：表述性状态转移

2000 年由 Roy Fielding 先生在其博士论文^①的第 5 章中正式提出 REST 的想法。在 REST 方式中，服务器端与客户端之间的通信使用无状态的连接。将两者之间所有互动状态的编码信息存放放到请求和响应之中。在服务器端把长期状态作为一组可识别的资源保存。使用一组定义良好 (well-defined) (并受到严格制约) 的资源标识符 (这里所说的 URL)，客户端访问这些资源。REST 从该内容的表象区分资源内容。REST 是这样设计的，以支持高度可扩展计算的同时从本质上去耦合制约的应用程序架构。

上面这种解释存在着大量的抽象内容。在实践中 REST 意味着什么？

首先，RESTful 方法的表达形式是指网络设计师知道何时何地可以缓存请求的响应。这能够负载网络的压力，并且在降低延迟的同时提高性能和弹性。

其次，经 REST 外加荷载的约束使应用程序变得更容易编写 (和维护)。RESTful 应用程序不必担心实施远程访问服务。相反，针对一组资源，它们提供了有规律的 (和简单的) 接口。应用程序实现列表、创建、编辑和删除每个资源的方法，并且由客户端完成剩余的部分。

下面对此作更具体的解释。在 REST 中，使用一组简单的动词，以一组丰富的名词形成动宾结构。如果使用 HTTP，则这些动词对应 HTTP 方法 (通常情况下是 GET、PUT、POST 和 DELETE)。这些名词都是应用程序中的资源。我们把这些资源作为 URL。

Depot 应用程序已经有了一组产品。这里隐式存在两种资源。一种是单个产品，每个构成了一种资源。第二种资源是：产品的集合。

为了获取所有产品的列表，可以对这个集合发出 HTTP GET 请求，其路径为 “/products”。为了得到单个资源的内容，必须明确地指出它。Rails 的方法将给予其主键的值 (也就是说，它的 id)。对此可发出 GET 请求，这个时候的 URL 为 “/products/1”。

要创建新产品到集合中，使用 HTTP POST 请求，这种请求是以 “/products” 路径定向，且包含要添加产品的事后数据。是的，获得产品清单，也使用相同的路径。如把 GET 发给它，则

① http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

它回应列表，而如对其进行 POST，则增加新产品到集合中。

借此作更进一步说明。上面已经看到了可以检索产品内容——只是发出 GET 请求，其路径为 “/products/1”。为了更新该产品，可发出相同 URL 的 HTTP PUT 请求。而要删除它，可发出 HTTP DELETE 请求，再次使用相同的 URL。

也许系统要对用户进行跟踪。又要处理一组资源。REST 告诉我们，对一组看上去类似的 URL（如 “/users” 和 “/users/1” 等），使用同样的一组动词（GET、POST、PUT 和 DELETE）。

上面已经看到经 REST 外加荷载的约束能力。已经熟悉了 Rails 的方法，以限制某种方式结构应用程序。REST 哲学告诉我们，用这种接口构造应用程序。突然之间世界变得简单了许多。

Rails 提供了直接支持这种类型的接口，它增加了一种宏路由工具，称为 `resources`。看看 6.1 节的文件 “config/routes.rb” 中的内容是什么样子的。

```
Depot::Application.routes.draw do |map|
  resources :products
end
```

`resources` 这一行代码产生了添加到应用程序的七个新路由。假定该应用程序称为 `ProductsController` 控制器，并且含有给定名称的七个行为。

这里可以看到所生成的路由。通过使用十分方便的 “`rake routes`” 命令可以得到：

```
products GET    /products(:format)
          POST    /products(:format)
          {action=>"create", :controller=>"products"}
new_product GET   /products/new(:format)
          {action=>"new", :controller=>"products"}
edit_product GET  /products/:id/edit(:format)
          {action=>"edit", :controller=>"products"}
product GET    /products/:id(:format)
          {action=>"show", :controller=>"products"}
          PUT    /products/:id(:format)
          {action=>"update", :controller=>"products"}
          DELETE /products/:id(:format)
          {action=>"destroy", :controller=>"products"}
```

所有已定义的路由是以列格式显示的。一般情况下，将把在屏幕上的这些行化装一番；事实上，在这个屏幕上每个路由已经折成两行。这些列都是（可选的）路由名称、HTTP 方法、路由路径以及（在此屏幕上单独行的）路由需求。

括号中的字段是可选的路径部分。使用 “`:format`” 参数让控制器可以根据不同的格式请求做出不同的处理。

现在看看与这些路由相关联的七个控制器行为。虽然在应用程序中已经创建管理产品的路由，但是下面进一步说明这一点，并且讨论一下资源——毕竟，对于所有以资源为基础的路由，都将需要这七个相同的方法：

`index`

返回资源列表。

create

从 POST 请求的数据中创建新资源，再将其添加到集合。

new

构造新资源，并把其传递给客户端。在服务器上不会保存这种资源。你可考虑新行动，为客户端创造可填写的空表格。

show

返回由 `params[:id]` 所确定的资源内容。

update

更新由 `params[:id]` 所确定的和与请求相关数据的资源内容。

edit

返回由 `params[:id]` 所确定的资源内容，为客户端创造可编辑的表格。

destroy

删除由 `params[:id]` 所确定的资源内容。

可以看到，这七项行动包含四个基本的 CRUD 操作（创建、读取、更新和删除）。它们还包含列出资源的行为，并且两个返回新的和现有的资源辅助行动，为客户端创造可编辑的表格。

如果由于某种原因，不需要或不希望所有七个行动，那么可以使用资源选项 `:only` 或者 `:except` 来限制这些行动：

```
resources :comments, :except => [:update, :destroy]
```

有几个路由都是使用帮助函数命名路由，如 `products_url` 和 `edit_product_url(:id=>1)`。

请注意，用可选的格式说明符 `:format`，每条路由都可定义的。下面将在 20.1 节中更详细地说明这种格式。

下面看看控制器代码：

```
depot_a/app/controllers/products_controller.rb
```

```
class ProductsController < ApplicationController
```

```
  # GET /products
```

```
  # GET /products.xml
```

```
  def index
```

```
    @products = Product.all
```

```
    respond_to do |format|
```

```
      format.html # index.html.erb
```

```
      format.xml { render :xml => @products }
```

```
    end
```

```
  end
```

```
  # GET /products/1
```

```
  # GET /products/1.xml
```

```
  def show
```

```
    @product = Product.find(params[:id])
```

```
    respond_to do |format|
```

```
      format.html # show.html.erb
```

```
      format.xml { render :xml => @product }
```

```

end
end

# GET /products/new
# GET /products/new.xml
def new
  @product = Product.new

  respond_to do |format|
    format.html { render :new.html.erb }
    format.xml { render :xml => @product }
  end
end

# GET /products/1/edit
def edit
  @product = Product.find(params[:id])
end

# POST /products
# POST /products.xml
def create
  @product = Product.new(params[:product])

  respond_to do |format|
    if @product.save
      format.html { redirect_to(@product,
        :notice => 'Product was successfully created.' ) }
      format.xml { render :xml => @product, :status => :created,
        :location => @product }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @product.errors,
        :status => :unprocessable_entity }
    end
  end
end

# PUT /products/1
# PUT /products/1.xml
def update
  @product = Product.find(params[:id])

  respond_to do |format|
    if @product.update_attributes(params[:product])
      format.html { redirect_to(@product,
        :notice => 'Product was successfully updated.' ) }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @product.errors,
        :status => :unprocessable_entity }
    end
  end
end
end

```

```

# DELETE /products/1
# DELETE /products/1.xml
def destroy
  @product = Product.find(params[:id])
  @product.destroy

  respond_to do |format|
    format.html { redirect_to(products_url) }
    format.xml { head :ok }
  end
end
end

```

注意，对于 REST 风格的每个行动如何才能有一个行为。在每一行前的注释显示了调用 URL 的格式。

还要注意到这些行动中的大多数都包含了 `respond_to` 代码块。正如在第 11 章开始内容中所看到的那样，Rails 使用它来确定响应发送的内容类型。支架生成器自动创建代码，它们针对 HTML 或 XML 内容请求做出适当响应。这里将做一些小小的发挥。

由生成器所产生的视图代码是相当简单的。唯一棘手的事情是需要使用正确的 HTTP 方法，将请求发送到服务器。例如，`index` 行为的视图代码如下所示：

```

depot_a/app/views/products/index.html.erb

<h1>Listing products</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Description</th>
    <th>Image url</th>
    <th>Price</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <% @products.each do |product| %>
    <tr>
      <td><%= product.title %></td>
      <td><%= product.description %></td>
      <td><%= product.image_url %></td>
      <td><%= product.price %></td>
      <td><%= link_to 'Show', product %></td>
      <td><%= link_to 'Edit', edit_product_path(product) %></td>
      <td><%= link_to 'Destroy', product, :confirm => 'Are you sure?',
        :method => :delete %></td>
    </tr>
  <% end %>
</table>

<br />

<%= link_to 'New Product', new_product_path %>

```

应该使用普通的 GET 方法，编辑产品且添加新产品的行动链接，这样标准函数 `link_to` 就能够很好地工作。然而，删除产品的请求必须发出 HTTP DELETE，因此调用包括函数 `link_to` 的 `:method=>:delete` 选项。

20.1.2 添加附加行为

虽然 Rails 资源提供了一组基础的行动，但是可进一步扩展自己的行为。在 12.2 节中增加了接口，让人们获取购买任何已知产品的清单。要利用 Rails 做到这一点，可以使用 `resources` 调用的扩展：

```
Depot::Application.routes.draw do
  resources :products do
    get :who_bought, :on => :member
  end
end
```

这种语法很简单。要添加新的行动，将其命名为 `who_bought`，通过 HTTP GET 调用。它适用于每个产品集合的成员。

如果不指定 `:member`，而指定 `:collection`，那么路由将适用于作为一个整体的集合。这经常应用于范围区域，例如，可能要允许有些产品集合，而另一些则通不过。

20.1.3 嵌套资源

通常，资源本身包含附加的资源集合。例如，可能希望消费者评论我们的产品。每次评论都将是一个资源，而评论集合将与每个产品的资源是相关的。

对于这种类型的情况，Rails 提供了方便和直观的声明路由的方式：

```
resources :products do
  resources :reviews
end
```

这定义了一组顶级 (top-level) 产品路由，而另外还创建对于评论的一组子路由。由于评论资源出现于产品代码块内，因此评论资源必须附属于产品资源。这意味着评论的路径必须始终以特定产品的路径为前缀。为了获取对于产品 id 为 99 的带 id 为 4 的评论，应该使用 `/products/99/reviews/4` 的路径。

对于 `/products/:product_id/reviews/:id` 所命名的路由是 `product_review`，而不是简单地 `review`。这种命名只是反映了这些资源的嵌套。

在通常情况下，可以看到使用 `rake routes` 命令所生成的一组完整路由。

20.1.4 浅路由嵌套

有时，嵌套资源可以产生繁杂的 URL。解决的办法是使用浅路由嵌套：

```
resources :products, :shallow => true do
  resources :reviews
end
```


这样如下的路由也就认可了:

```
/products/1 => product_path(1)
/products/1/reviews => product_reviews_index_path(1)
/reviews/2 => reviews_path(2)
```

尝试使用命令 `rake routes`, 查看完整的映射。

20.1.5 选择数据表述

REST 架构的目标之一就是数据表述解耦数据。如果使用 URL 路径 `/products` 来获取一些产品, 那么就应该看到很好的格式化的 HTML。如果应用程序询问同一个 URL, 那么可选择代码友好的格式 (如 YAML、JSON 或 XML 等) 结果。

前面已经看到了, Rails 可以使用在控制器的 `respond_to` 代码块的 HTTP 接收标头 (HTTP Accept header)。但是, 设置 HTTP 接收标头, 并不总是很容易的 (有时它并不可能是简单的)。为了解决这个问题, Rails 允许你传递响应的格式, 这种格式可以作为 URL 的一部分。要做到这一点, 把在路由中的参数 `:format` 设置为所要返回的 MIME 类型的文件扩展名。正如所看到的那样, Rails 是这样实现的, 包含在路由定义中称为 `:format` 的项:

```
GET /products(.:format)
{:action=>"index", :controller=>"products"}
```

因为在路由定义中句号 (一段) 作为分隔符字符, 所以这样把 `:format` 视为另外一项。要是给其 `nil` 默认值, 它就是可选项。

要是这样做, 就可以把在控制器中的 `respond_to` 代码块应用于选择响应类型, 这种类型依赖于请求格式:

```
def show
  respond_to do |format|
    format.html
    format.xml { render :xml => @product.to_xml }
    format.yaml { render :text => @product.to_yaml }
  end
end
```

鉴于此, 对于 `/store/show/1/` 或者 `/store/show/1.html` 的请求将返回 HTML 内容; `/store/show/1.xml` 将返回 XML; `/store/show/1.yaml` 将返回 YAML。也可以 HTTP 请求参数进行格式传递:

```
GET HTTP://pragprog.com/store/show/123?format=xml
```

在默认情况下, 资源定义的路由可启用此功能。

虽然单一控制器响应不同的内容类型, 这种想法似乎具有吸引力, 但是在现实情况中这是非常棘手的。尤其要指出的是, 错误处理可能是困难的。尽管对于向用户重定表单的错误是可接收的, 同时对其显示了不错的闪存消息, 但是当为 XML 提供服务时, 必须采取不同的策略。在决定把所有过程捆绑成单一控制器之前, 要仔细考虑应用程序体系结构。

Rails 使开发应用程序变得简单, 这是建立在资源基础上的路由。许多人声称, 这样大大地简化了开发应用程序。然而, 这种说明并不总是适当的。即使不能找到一种进行工作的方式, 也不要以为必须使用它。总是可以有相混合和相匹配的。有些控制器的基础可以是资源, 而另外的

可以基于行动。甚至还有一些控制器可以带一些额外的行为作为基础的资源。

20.1.6 测试路由

到目前为止，一直在通过使用命令 `rake routes` 查看路由方式，探讨它们。但是，当推出应用程序时，可能还要多一些形式，并且包括验证路由如期运作的单元测试。Rails 包括了一系列测试帮助程序，使其变得更容易：

```
assert_generates(path, options, defaults={}, extras={}, message=nil)
```

验证一组给定的选项，这些选项生成指定的路径。

```
depot_t/test/unit/routing_test.rb
```

```
def test_generates
  assert_generates("/", :controller => "store", :action => "index")
  assert_generates("/products",
    { :controller => "products", :action => "index" })
  assert_generates("/line_items",
    { :controller => "line_items", :action => "create",
      :product_id => "1",
      :method => :post }, { :product_id => "1" })
end
```

参数 `extras` 是用来传递附加请求参数的名称和值的请求（在前面代码的第三个断言中，这将是 `product_id=1`）。这些测试框没有把作为字符串请求添加所生成的 URL，而是测试这些已添加的值，这些值出现在散列类型参数 `extras` 中。

参数 `defaults` 用来指明 HTTP 方法。

```
assert_recognizes(options, path, extras={}, message=nil)
```

验证路由返回一组给定路径的特定选项。

```
depot_t/test/unit/routing_test.rb
```

```
def test_recognizes
  # Check the default index action gets generated
  assert_recognizes({ "controller" => "store", "action" => "index" }, "/")

  # Check routing to an action
  assert_recognizes({ "controller" => "products", "action" => "index" },
    "/products")

  # And routing with a parameter
  assert_recognizes({ "controller" => "line_items",
    "action" => "create",
    "product_id" => "1" },
    { :path => "/line_items", :method => :post },
    { "product_id" => "1" })
end
```

参数 `path` 可指定路由，该路由是以请求 HTTP 动词为条件的。作为方法 `assert_`

`recognizes` 的第二个参数，可以传递散列对其进行测试，而不是字符串。散列应该包含两个元素：`:path` 将包含输入的请求路径，而 `:method` 将包含要使用的 HTTP 动词。

参数 `extras` 也包含了附加的 URL 参数。在前面代码的第三个断言中，使用参数 `extras`，用来验证以 `?product_id= 1` 结尾的 URL，产生的散列 `params` 将包含适当的值。

```
assert_routing(path, options, defaults={}, extras={}, message=nil)
```

结合前面的两个断言，先验证路径生成的选项，接着核实选项生成的路径。

```
depot_t/test/unit/routing_test.rb
```

```
def test_routing
  assert_routing("/", :controller => "store", :action => "index")
  assert_routing("/products", :controller => "products", :action => "index")
  assert_routing({:path => "/line_items", :method => :post,
    { :controller => "line_items", :action => "create",
      :product_id => "1"},
    {, { :product_id => "1"}}
  end
```

重要的是，在选项散列中，要使用符号作为键，而用字符串作为值。如果不是这样做的话，那么可以断言，如果比较这些从路由返回的选项，将会失败。

20.2 处理请求

上一节探讨了行为调度是如何把外部请求路由到应用程序相应代码中。下面看看代码里面发生了什么变化。

20.2.1 行为方法

当控制器对象处理请求时，该对象会寻找同名的公共实例方法作为外部行动。如果该实例方法存在的话，那么就调用该方法。如果没有找到，且控制器实现 `method_missing`，那么调用这种方法 `method_missing`，传递行为名作为第一个参数，并且空参数列表作为第二个。如果没有一种方法可以调用，那么该控制器找寻在当前控制器行动之后所命名的模板。如果找到了，就直接呈现至这个模板。如果没有出现这些情况，则产生未知行为（unknown action）错误。

20.2.2 控制器环境

控制器设置了行动及其调用的视图（借助于扩展库）的环境。这些方法中的大多数提供了直接访问 URL 或请求中所包含的信息。

action_name

当前正要处理的行为名称。

cookies

请求相关联的 cookie。当发送响应时，设置这个对象的值存储在浏览器上的 cookie。Rails 支持会话是基于 cookie 的。在 20.3 节中将讨论会话。

headers

具有 HTTP 标头的散列将用于响应中。在默认情况下, 设置 Cache-Control 为 no-cache。对于特殊用途的应用程序, 要设置 Content-Type 标头。请注意, 不应该直接在标头中设置 cookie 的值——要做到这一点, 可以使用 Cookie API。

params

包含请求参数的类似于散列的对象 (与路由过程所生成伪参数一起)。之所以类似于散列, 是因为可以使用符号或字符串对条目进行索引——`params[:id]` 和 `params['id']` 返回相同值。在惯用上, Rails 应用程序使用符号形式。

request

外部请求对象。它包括以下属性:

- `request_method` 返回请求方法之一: `:delete`、`:get`、`:head`、`:post` 或 `:put`。
- `method` 与 `request_method` 返回相同值, 但不包括 `:head`, 之所以返回与 `:get` 是一样的, 是因为从应用的角度来看它们两个在功能上是相当的。
- `delete?`、`get?`、`head?`、`post?` 和 `put?` 在请求方法基础上返回 `true` 或 `false`。
- 如果任何一个 Ajax 帮助程序发出这个请求, 那么 `xml_http_request?` 和 `xhr?` 返回 `true`。请注意, 这个参数是独立于方法参数的。
- `url`, 它返回请求所使用的完整 URL。
- `protocol`、`host`、`port`、`path` 和 `query_string` 返回用于请求的 URL 组成部分, 是基于以下模式: `protocol://host:port/path?query_string`。
- `domain`, 它返回请求域名中的最后两个组成部分。
- `host_with_port`, 它是对于请求的 `host:port` 字符串。
- `port_string`, 如果该端口不是默认端口 (对于 HTTP 协议是 80, 对于 HTTPS 协议是 443), 那么它是对于请求的 `:port` 字符串。
- `ssl?`, 如果这是 SSL 请求, 换句话说, 请求是使用 HTTPS 协议, 那么它就是 `true`。
- `remote_ip`, 它返回作为字符串的远程 IP 地址。如果客户端处于代理之后的话, 那么该字符串可能有多于一个以上的地址。
- `path_without_extension`、`path_without_format_and_extension`、`format_and_extension` 和 `relative_path` 返回完整路径的一部分。
- `env`, 请求的环境。通过浏览器所设定的访问值, 可以使用它, 如:

```
request.env['HTTP_ACCEPT_LANGUAGE']
```
- `accepts`, 它是对于请求的 `accepts` MIME 类型的值。
- `format`, 这是对于请求的 `content-type` 值。如果没有格式可用的话, 那么就使用接收类型的第一个值。
- `mime_type`, 这是与扩展相关联的 MIME 类型。
- `content_type`, 这是对于请求的 MIME 类型。对于 `put` 和 `post` 请求而言, 这是有用的。
- `headers`, 这是一组完整的 HTTP 标头。
- `body`, 这是 I/O 流的请求主体。

- `content_length`，这说明了主体内的字节数。

Rails 充分利用名为 Rack 的软件包，为此提供了大量的功能。请参阅文件 `Rack::Request` 的全部细节。

response

响应对象，在请求处理过程中填补空缺。在通常情况下，这个对象是由 Rails 进行管理的。正如 20.3.6 节将看到的，有时会访问对于特殊过程的内部。

session

类似于散列的对象，它表示当前会话数据。这将在 20.3 节中进行阐述。

此外，日志器 (logger) 可用于整个 Action Pack。

20.2.3 用户响应

控制器工作的一部分是为了响应用户。有以下四种基本方式：

- 最常见的方式是呈现模板。在 MVC 概念中，模板就是视图，由控制器所提供的信息，并用它来生成浏览器的响应。
- 该控制器可以直接向浏览器返回字符串，而无需调用视图。这是相当罕见的，但可以用来发送错误通知。
- 控制器可以不向浏览器返回任何代码。在响应 Ajax 请求时，这种情况有时才会出现。然而，在所有情况下，之所以控制器返回一组 HTTP 标头，是因为期望得到了某种响应。
- 控制器可以把其他数据 (HTML 之外的数据) 发送到客户端中。通常这是某种形式的下载 (可能是 PDF 文档或文件的内容)。

控制器总是响应用户每个请求仅一次。这意味着，在处理任何请求时，只有一次调用 `render`、`redirect_to` 或 `send_xxx` 方法的机会。(在第二次呈现时，抛出 `DoubleRenderError` 异常。)

因为控制器必须只回应一次，所以在处理请求完成之前，它会检查是否已生成了回应。如果没有，那么控制器寻找在控制器的行为后所命名的模板，并且自动呈现 (`render`) 到该模板。这是最常见的呈现发生的方式。你可能已经注意到，在购物车教程的大多数行动中，从来没有明确地呈现到哪里。相反，行动方法建立起视图和返回的前后关系。控制器提示还没有任何呈现发生且自动调用相应的模板。

可以使用相同名但不同扩展名 (例如 `.html.erb`、`.xml.builder` 和 `.js.rjs`) 的多个模板。如果不指定呈现请求的扩展名，那么 Rails 默认为 `html.erb`。

20.2.4 呈现模板

模板是定义为应用程序响应内容的文件。在默认情况下，Rails 支持三种模板格式：`erb`，这是嵌入 Ruby 代码 (通常与 HTML 一起使用)；`builder`，构建 XML 内容的更程序化方式；`RJS`，它生成 JavaScript 代码。将从第 21 章开始谈论这些文件的内容。

按照惯例，行动 `action` 和控制器 `controller` 的模板将在文件 `app/views/controller/action.type.xxx` 中 (其中类型是文件类型，如 `html`、`atom` 或 `js`；而 `xxx` 是 `erb`、`builder` 或 `rjs` 之一)。

app/views 名称的一部分是默认的。对于整个应用程序,通过下面设置可以重新定义:

```
 ActionController.prepend_view_path dir_pathe
```

在 Rails 中,render 方法是所有呈现的核心。它需要散列选项,该选项告诉它该呈现什么和如何呈现。

在控制器中编写如下所示的代码,这是吸引人的:

```
# DO NOT DO THIS
def update
  @user = User.find(params[:id])
  if @user.update_attributes(params[:user])
    render :action => :show
  end
  render :template => "fix_user_errors"
end
```

调用 render 和 redirect_to 的行动应该以某种方式终止行动过程,这似乎是非常自然的。情况并非如此。在 update_attributes 成功情况下,上面的代码会产生错误(因为调用了两次 render)。

下面浏览一下在控制器中所使用的 render 选项(查看第 21 章中所涉及视图的呈现):

```
render( )
```

没有重新定义任何参数,render 方法将呈现到当前控制器的行动的默认模板。下面代码将呈现模板 app/views/blog/index.html.erb:

```
class BlogController < ApplicationController
  def index
    render
  end
end
```

也有下面的(如果没有行为存在的话,那么控制器的默认行为是调用 render):

```
class BlogController < ApplicationController
  def index
    end
end
```

且将有这种代码(因为如果没有定义任何行动方法,那么控制器将直接调用模板):

```
class BlogController < ApplicationController
  end
  render(:text=> string)
```

将给定的字符串发送到客户端。没有执行模板解析或 HTML 转义。

```
class HappyController < ApplicationController
  def index
    render(:text => "Hello there!")
  end
end
```

```
render(:inline=>string,[:type=>"erb"|"builder"|"rjs"],[:locals=>hash])
```

将作为代码的字符串解析成给定类型的模板，并把结果呈现到客户端。可以使用 `:locals` 散列来设置模板的局部变量值。

如果在开发模式下运行应用程序，那么下面的代码将 `method_missing` 添加到控制器。如果用无效行动调用控制器，那么这将呈现到内联模板，以显示该行为名称和请求参数的格式化版本。

```
class SomeController < ApplicationController
```

```
  if RAILS_ENV == "development"
```

```
    def method_missing(name, *args)
```

```
      render(:inline => %{
```

```
        <h2>Unknown action: #{name}</h2>
```

```
        Here are the request parameters:<br/>
```

```
        <%= debug(params) %> }%)
```

```
    end
```

```
  end
```

```
end
```

```
render(:action => action_name)
```

呈现到在此控制器中给定行为的模板。有时当应用重定向时，使用 `render` 的 `:action` 形式。请参阅 20.2.5 节中的讨论。

```
def display_cart
```

```
  if @cart.empty?
```

```
    render(:action => :index)
```

```
  else
```

```
    # ...
```

```
  end
```

```
end
```

请注意，调用 `render(:action...)` 并不调用行为方法，它只是显示模板。如果模板需要实例变量，那么这种调用 `render` 的方法必须设置它们。

之所以再重复强调这一点，是因为这是初学者经常发生的错误：调用 `render(:action...)` 不调用该行为方法。它只是呈现到该行动的默认模板。

```
render(:file => path, [ :use_full_path => true|false ], [:locals => hash])
```

呈现到给定路径（它必须包括文件扩展名）的模板。在默认情况下，这应该是模板的绝对路径，但如果 `:use_full_path` 选项是 `true`，那么视图将预先准备的模板基本路径值传递给该路径。模板基本路径是在应用程序的配置文件中设置的。如果已指定的话，那么在 `:locals` 散列值用于设置模板中的局部变量。

```
render(:template => name, [:locals => hash])
```

呈现模板，并安排要送回客户端所产生的文本。`:template` 值必须包含控制器和新名称行动的部分，可以通过正斜杠分隔。下面的代码将呈现到模板 `app/views/blog/short_list`：


```
class BlogController < ApplicationController
  def index
    render(:template => "blog/short_list")
  end
end
```

```
render(:partial => name, ...)
```

呈现局部模板。将在第 21 章中深入讨论局部模板。

```
render(:nothing => true)
```

返回无任何内容——将空体发送到浏览器中。

```
render(:xml => stuff)
```

呈现作为文本形式的内容，使内容类型为 application/xml。

```
render(:json => stuff, [:callback => hash])
```

呈现作为 JSON 形式的内容，使内容类型为 application/json。指定 :callback 将导致以调用名为 :callback 函数包封的结果。

```
render(:update) do |page| ... end
```

呈现作为 RJS 模板的代码块，传递到页面对象。

```
render(:update) do |page|
```

```
  page[:cart].replace_html :partial => 'cart', :object => @cart
```

```
  page[:cart].visual_effect :blind_down if @cart.total_items == 1
```

```
end
```

所有 render 的形式都采用可选 :status、:layout 和 :content_type 参数。:status 参数提供了在 HTTP 响应的状态标头中所使用的值。它默认为“200 OK”。不要使用带重定向的 3xx 状态的 render；Rails 提供了用于此目的的重定向方法。

参数 :layout 确定是否布局包封呈现的结果（在第 8 章中涉及布局，在第 21 章中将深入观察它）。如果参数是 false，那么将没有应用布局。如果设置为 nil 或 true，那么将应用当且仅当存在与当前行动相关的布局。如果 :layout 参数有字符串的值，则将它视为在呈现时使用的布局名称。当 :nothing 选项生效时，就不再应用布局。

参数 :content_type 指定一个值，以 Content-Type HTTP 标头方式将其传递到浏览器中。

能够获取的并不是以字符串形式发送到浏览器的内容，有时这是有用的。render_to_string 方法与 render 采用了相同参数，但返回结果是作为字符串呈现——这一呈现并不存储于响应对象，因此不会发送到用户，除非采取一些附加的步骤。调用 render_to_string 不能算作真正的呈现。可以调用真正的 render 方法，其后没有得到 DoubleRender 错误。

20.2.5 发送文件和其他数据

在前面内容中已经看到在控制器中呈现模板和发送字符串。第三种响应类型是向客户端发送数据（通常这些数据是文件内容，但不是必需的）。

1. send_data

发送包含客户端二进制数据的字符串。


```
send_data(data, options...)
```

将数据流发送到客户端。在通常情况下，浏览器将使用内容类型（content-type）和内容处置（content-disposition）的组合，它们在选项中设置，以确定这个数据的用途。

```
def sales_graph
  png_data = Sales.plot_for(Date.today.month)
  send_data(png_data, :type => "image/png", :disposition => "inline")
end
```

选项:

:disposition	string	建议浏览器应内嵌（选项 inline）或下载且保存（选项 attachment，这是默认值）显示文件。
:filename	string	建议当浏览器保存此数据时，应使用的默认文件名。
:status	string	状态代码（默认值是：“200 OK”）。
:type	string	内容类型，默认值是：application/octet-stream。
:url_based_filename	boolean	如果设置为 true，且 :filename 没有设置，那么阻止 Rails 提供在 Content-Disposition 标头中的文件基本名。指定这一点是必要的，以便使一些浏览器正确处理 il8n 文件名。

2. send_file

向客户端发送文件内容。

```
send_file(path, options...)
```

给客户端发送给定文件。该方法设置 Content-Length（内容长度）、Content-Type（内容类型）、Content-Disposition（内容处置）以及 Content-Disposition（内容传输编码）标头。

选项:

:buffer_size	number	以每次写方式发送到浏览器量，如果启用了流（:stream 为 true）。
:disposition	string	建议浏览器应内嵌（选项 inline）或下载且保存（选项 attachment，这是默认值）显示文件。
:filename	string	建议当浏览器保存此数据时，应使用的默认文件名。
:status	string	状态代码（默认值是：“200 OK”）。
:stream	true or false	如果设置为 false，那么把整个文件读入服务器内存，且发送给客户端。否则，以 :buffer_size 段数把文件读取和写入到客户端。
:type	string	内容类型，默认值是：application/octet-stream。

对于在控制器中使用标头属性的任何一个 send_ 方法，可设置附加标头。

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-Description"] = "Top secret"
end
```

从 21.4 节开始将展示如何上传文件。

20.2.6 重定向

从服务器发送 HTTP 重定向到响应请求的客户端。实际上，可以说：“我不能处理这个请求，但这里有一些可处理的网址。”重定向响应包含 URL，客户端应该尝试寻找随后附加的一些状态信息，明确重定向是永久性的（状态代码 301）还是临时（状态代码 307）。有时用重定向来重组网页；客户端访问旧位置的页面，将会引入到网站首页。更常见的是，Rails 应用程序使用重定向，其目的是将请求过程传递到其他的一些行动中。

在网络浏览器的后台处理重定向。在通常情况下，要知道已重定向的唯一方法就是轻微延迟，而事实上正在浏览页面的 URL 已经从请求的那个发生了变化。最后这一点是重要的——对浏览器而言，服务器重定向其作用等同于最终用户手动输入新的目标 URL。

当编写表现良好（well-behaved）的 Web 应用程序时，重定向结果是重要的。看看简单的博客应用程序，它支持评论张贴。在用户发表了评论之后，该应用程序应该重新显示文章，可能在其后带有新的评论。

按照这个想法编写如下代码：

```
class BlogController
  def display
    @article = Article.find(params[:id])
  end

  def add_comment
    @article = Article.find(params[:id])
    comment = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
      flash[:note] = "Thank you for your valuable comment"
    else
      flash[:note] = "We threw your worthless comment away"
    end
    # DON'T DO THIS
    render(:action => 'display')
  end
end
```

上面代码的目的是，在已经发布评论之后明确地显示该文章。要做到这一点，程序开发者调用 `render(:action=>'display')` 终止 `add_comment` 方法。这呈现到显示视图，向最终用户显示更新文章。但从浏览器的观点考虑这一点。它发出以 `blog/add_comment` 结尾的网址，并且反过来得到索引列表。就浏览器而言，当前 URL 仍然是以 `blog/add_comment` 结尾的。这意味着，如果用户单击刷新或重新加载（也许是为了看看其他人是否已经发表了评论），那么将 `add_comment` 的网址再次发送给应用程序。虽然用户打算刷新显示，但是应用程序收到添加另一评论的请求。在博客应用程序中，这种无意的两次切入不是时候。在在线商店中，它可以变得代价昂贵。

在这种处境中，显示索引列表添加评论的正确方法是，将浏览器重定向到 `display` 行动。

具体的做法是使用 Rails 的 `redirect_to` 方法。如果用户随后单击“刷新”，那么它会简单地重新调用 `display` 行动，并没有添加另一评论：

```
def add_comment
  @article = Article.find(params[:id])
  comment = Comment.new(params[:comment])
  @article.comments << comment
  if @article.save
    flash[:note] = "Thank you for your valuable comment"
  else
    flash[:note] = "We threw your worthless comment away"
  end
  redirect_to(:action => 'display')
end
```

Rails 具有简单而强大的重定向机制。它可以重定向到给定控制器（传递参数）的行动、URL（打开或关闭当前服务器）或前一页。下面看看这三种形式。

1. `redirect_to`

重定向到行动。

```
redirect_to(:action => ..., options...)
```

将临时重定向发送到基于选项散列值的浏览器。使用 `url_for` 方法生成目标 URL，所以这种 `redirect_to` 形式具有一切 Rails 路由代码背后的智慧。

2. `redirect_to`

重定向到 URL。

```
redirect_to(path)
```

重定向到给定路径。如果路径开始没有协议（如 `http://`），那么将添加当前请求的协议和端口。此方法没有执行任何 URL 重写，因此它也不应该用来创建以链接到应用程序的行动（除非使用 `url_for` 方法或给路由命名的 URL 生成器产生该路径）为目标的路径。

```
def save
  order = Order.new(params[:order])
  if order.save
    redirect_to :action => "display"
  else
    session[:error_count] ||= 0
    session[:error_count] += 1
    if session[:error_count] < 4
      self.notice = "Please try again"
    else
      # Give up -- user is clearly struggling
      redirect_to("/help/order_entry.html")
    end
  end
end
```

3. `redirect_to`

重定向引用。

```
redirect_to(:back)
```

重定向到通过 HTTP_REFERER 标头的当前请求的 URL 中。

```
def save_details
```

```
  unless params[:are_you_sure] == 'Y'
```

```
    redirect_to(:back)
```

```
  else
```

```
    ...
```

```
  end
```

```
end
```

在默认情况下，对所有重定向进行临时标记（它们只会影响当前请求）。当重定向到 URL 时，可能想使重定向永久，这是可能的。在这种情况下，相应地在响应标头中设置该状态：

```
headers["Status"] = "301 Moved Permanently"
```

```
redirect_to("http://my.new.home")
```

由于重定向方法将响应发送给浏览器，所以同样的规则适用于呈现方法——每个请求只能发出一次。

到目前为止，我们一直在寻找采用隔离方式的请求和响应。Rails 也提供了许多持续请求的机制。

20.3 持续请求的对象和操作

虽然大部分存留持续请求的状态属于数据库，且是通过 Active Record 访问的，但是其他一些位状态有不同的寿命持续和不同的需要进行管理。在 Depot 应用程序中，购物车本身是存储在数据库中，用于了解哪个购物车是由会话管理的当前购物车。闪存警告曾经用于将简单消息（如“无法删除最后一个用户”）传送到重定向后的下一个请求。而过滤器曾经用来提取 URL 本身的本地数据。

在本节中将依次探讨这些机制的每一个。

20.3.1 Rails 会话

Rails 会话是类似于散列的存留持续请求的结构。不同于原始 cookie，会话可以容纳任何对象（只要可以封送（见 4.5 节）这些对象），这使容纳 Web 应用程序的状态信息成为理想之选。例如，在 Web 应用程序中，曾经使用容纳在请求间购物车对象的会话。在该应用程序中，就像任何其他对象一样，可以使用 Cart 对象。但 Rails 曾经这样做的，在处理每个请求结束以后保存购物车，更重要的是，当 Rails 开始处理请求时，恢复对于输入该请求的正确购物车。使用会话，可以假设应用程序继续保留这些请求。

于是就产生了一个有趣的问题：究竟在何处这一数据继续保留这些请求呢？一种选择是从服务器将其作为 cookie 发送到客户端中。这是 Rails 3.0 的默认选择。它限制数据大小，且增加带宽，但也意味着有更少的服务器来管理和清理。请注意加密地签名该内容，但（默认）是不加密，这意味着用户可以看到但不能干预此内容。

David 说 基于 cookie 的会话奇迹

默认 Rails 会话存储，一开始听起来像是一个疯狂的想法。实际上会存储它到客户端上吗？如果想要在会话中存储核弹发射代码，而又不能让客户端真实地知道这些，行吗？

可以的，默认存储不适合存储需要保存到客户端的秘密。但实际上这是一种有价值的约束，这将避免保存复杂对象的风险，这些对象在会话中可能会过时。而核弹发射代码只是夸大其词和虚无缥缈的事情。

没有任何大小限制。cookie 允许大约只有 4KB 大小，所以不可能容纳所有东西。那仅是适合存储引用的最佳做法，像 `cart_id` 那样的对象，而不是实际的购物车对象本身。

应该担心的关键安全问题是，在实际客户端中能否改变会话，以确保所表达值的完整性。如果客户端可以对 `cart_id` 从 5 改变为 8，而得到别人的购物车，这将是一件不好的事情。值得庆幸的是，通过签名会话，Rails 完全防范了这种风险，如果它不匹配，那么抛出干预数据警告的异常。

得到的回报好处是，在获取和保存每次请求的会话过程中，没有给数据库添加负载，并且也没有清理工作。如果把会话保存在文件系统或数据库中，那么必须处理如何清理过时的会话，这是一件真正的麻烦事情。没有人喜欢清理工作。基于 cookie 的会话知道如何清理自己。有什么理由不喜欢这种方法呢？

另一种选择是在服务器上存储数据。对此存在两个部分。第一部分，Rails 必须保持跟踪会话。其方法是创建（在默认情况下）三十二进制的字符键（这意味着存在着 16^{32} 种组合）。这个键就是所谓的会话 id，并且事实上它是随机的。Rails 负责把 cookie（带着键 `_session_id`）的这个会话 id 存储到用户浏览器上。由于后续请求从该浏览器达到应用程序，所以 Rails 可以恢复会话 id。

第二部分，Rails 会保持持久性存储的会话数据，通过会话 id 进行索引。当请求到来时，Rails 使用会话 id，且查找数据存储。在那里找到的这一数据是序列化的 Ruby 对象。对其进行解压序列化，并且在控制器的会话属性中存储该结果，其中的数据可应用于应用程序代码。应用程序可以把此数据添加和修改到其核心内容。当完成每次请求的处理时，Rails 重新把会话数据写入到数据存储。这些数据在那里存储着，一直到该浏览器的下次请求到来。

应该在会话中存储什么？你可能想要存储任何东西，但受到了一些限制和有需要注意的事项：

- 对于在会话中存储对象存在一些限制。细节取决于所选择的存储机制（接下来就会看到）。在一般情况下，在会话中对象必须是可序列化的（使用 Ruby Marshal 函数）。这意味着，例如，不能在会话中存储 I/O 对象。
- 可能不希望在会话中存储大量对象——把它们放到数据库中，而从会话中引用它们。基于 cookie 的会话，这是尤其正确的事情，其中总限额为 4KB。
- 可能不希望在会话数据中存储不稳定的对象。例如，在博客应用程序中，可能想在会话中

存储保持文章的总数量，出于提供性能原因，存储于会话中。但是，如果这样做，那么当其他用户添加文章时，该数量不会得到更新。

存储在会话数据中表述当前登录用户的对象，这是十分有趣的事情。如果应用程序需要处理无效用户，那么这可能不是明智之举。即使在数据库中没有激活用户，该会话数据仍将反映出激活的状态。

在数据库中存储易变数据，并在会话中使用引用（reference）代替它。

- 可能不希望在会话数据中仅仅存储关键信息。例如，如果应用程序在请求中生成确认订单数量，并且把它存储到会话数据中，这样当处理下一个请求时，可以把它保存到数据库中，如果用户从浏览器中删除该 cookie，那么可能会失去这个数字。关键信息需要保存在数据库中。

特别需要留心。如果在会话数据中存储对象，那么下次回到浏览器时，应用程序将最终索回该对象。可是，如果在此期间已经更新了应用程序，那么在应用程序中，会话数据的对象可能不会与该对象类的定义相一致了，这样应用程序将无法处理该请求。这里存在三个选项。其一是使用传统模型在数据库中存储对象，只是在会话中保持行记录 id。与 Ruby 封送库相比，模型对象对于数据库模式改变更为宽松一点。其二是，一旦改变了在会话数据中所存储的类定义，就手动删除所有在服务器上存储的该会话数据。

其三稍微复杂一些。如果把版本号添加到会话密钥，并且一旦更新存储的数据，就更改这个数字，那么将永远只能装载与应用程序当前版本所对应的数据。可能更改在会话中存储对象的类，并根据与每个请求关联的会话键，使用适当的类。针对最后这种想法，可能需要做大量工作，所以需要决定它是否值得努力去做。

因为会话存储类似于散列一样，所以可以让它保存多个对象，而每个都有自己的键。

对于特殊行为，也没有必要禁用会话。因为会延迟加载会话，所以根本不会在任何不需要会话的行为中引用会话。

20.3.2 会话存储

当涉及存储会话数据时，Rails 提供了多个选项。每一项都有优点和缺点。先将选项列出，然后到后面比较它们。

类 ActionController::Base 的属性 session_store 决定了会话存储机制——将此属性设置为实现存储策略的类。这个类必须定义在 CGI::Session 模块中。把符号名应用于命名会话存储策略；把符号转换成按骆驼命名法（CamelCase）命名的类名。

```
session_store = :cookie_store
```

从 2.0 版本起，这是 Rails 使用的默认会话存储机制。此格式代表封送形式的对象，它允许在会话中保存任何可序列化的数据，但限于 4KB 总量。这是在 Depot 应用程序中所使用的选项。

```
session_store = :p_store
```

在 PStore 格式的平面文件（flat file）中存储每个会话的数据。这种格式保存封送形式的对象，它允许在会话中保存任何可序列化的数据。这种机制支持附加的配置选项 :prefix 和 :tmpdir。在 config 目录的文件 environment.rb 中有如下使用 PStore 会话配置的代码。

```
Rails::Initializer.run do |config|
  config.action_controller.session_store = CGI::Session::PStore
  config.action_controller.session_options[:tmpdir] = "/Users/dave/tmp"
  config.action_controller.session_options[:prefix] = "myapp_session_"
  # ...
```

```
session_store = :active_record_store
```

可以在应用程序数据库中使用 ActiveRecordStore 存储会话数据。可以生成使用 Rake 命令创建名为 sessions 表的迁移：

```
depot> rake db:sessions:create
```

运行命令 rake db:migrate 来创建实际的表。

如果观察一下迁移文件，那么将看到，Rails 创建字段为 session_id 的索引，这是因为它用来寻找会话数据。Rails 还定义了名为 updated_at 的字段，这样 Active Record 会自动地在会话表中使得行记录留下时间戳——后面将看到为什么这是一个好主意。

```
session_store = :drb_store
```

DRb 是允许通过网络连接共享对象的 Ruby 进程协议。使用 DRbStore 数据库管理，在 DRb 服务器（这可在 Web 应用程序之外进行管理）中，Rails 存储会话数据。应用程序的多个实例可能运行在分布式服务器上，可以访问相同的 DRb 存储。DRb 使用封送来对实例进行序列化。

```
session_store = :mem_cache_store
```

memcached 是可自由使用和分布式的对象缓存系统，该系统是由 Dormando[⊖]进行维护。Rails MemCacheStore 使用针对于 memcached 的 Michael Granger 的 Ruby 接口[⊖]来存储会话。与其他选择相比，使用 memcached 更复杂一些，只有在你的网站上由于其他原因使用它时才能引起你的注意。

```
session_store = :memory_store
```

此选项在本地存储会话数据到应用程序的内存中。因为没有序列化，任何对象都可以存储在 inmemory 会话中。正如在一分钟内将看到，这通常不是 Rails 应用程序的一个好主意。

```
session_store = :file_store
```

在平面文件中存储会话数据。对于 Rails 应用程序，这几乎是毫无用处的，因为其内容必须是字符串。这种机制支持附加的配置选项 :prefix、:suffix 和 :tmpdir。

20.3.3 比较会话存储选项

针对所有这些可供选择的会话选项，你应该在应用程序中使用哪些呢？与往常一样，答案是“视情况而定”。

当涉及性能时是没有绝对的，而每个人的背景也是不同的。硬件、网络延迟、数据库选择甚至天气都可能影响所有会话存储的组件是如何进行交互的。最好的建议是先从最简单可行的解决方案开始，然后监视它。如果它开始慢下来，那么找出为什么再次出现困难。

⊖ <http://memcached.org/>

⊖ Available from <http://deveiate.org/projects/RMemCache>

如果是高流量网站，那么尽可能少地保存会话数据，同时使用方法 `cookie_store`，这是一种可行的方法。

如果认为内存存储过于简单化、文件存储限制过于严格而 `memcached` 过于强大，那么服务器端可以选择 `PStore`、`Active Record` 存储和 `DRb` 存储器。如果需要在会话中存储比用 `cookie` 更多的内容，推荐从 `Active Record` 解决方案开始考虑。如果应用程序不断增长，你会发现这将成为瓶颈，那么可以迁移到以 `DRb` 为基础的解决方案。

20.3.4 会话逾期与清除

所有服务器端会话存储解决方案的问题在于，每个新会话把一些东西添加到会话存储中。这意味着最终需要做一些清理工作，否则将耗尽服务器资源。

清理会话还有另外一个原因。许多应用程序不希望会话永远持续下去。一旦用户从特定的浏览器登录了，那么应用程序就可能强制执行一种规则：只有用户是活跃的，才保留登录状态；当注销或在最后使用应用程序后的一段固定时间，其会话应终止。

通过使得保持会话 `id` 的 `cookie` 逾期，有时可以达到这种效果。对于最终用户违规操作，这也是开放的。更糟的是，相对于服务器上整齐的会话数据，难以同步浏览器逾期的 `cookie`。

因此建议，只通过删除其服务器端的会话数据，使得会话逾期。如果浏览器随后到达的请求中包含会话 `id`，但已删除了它的数据，那么应用程序将不会接收会话数据；事实上会话将不存在那里了。

实现这个逾期取决于正在使用的存储机制。

对于基于 `PStore` 的会话，最简单的方法是定期运行清理任务（例如在类似于 `UNIX` 系统下使用 `cron(1)`）。这个任务检查在会话数据目录中文件的最后修改时间，删去那些超过给定时间的会话数据。

对于基于 `Active Record` 的会话存储，使用 `sessions` 表的字段 `updated_at`。可以删除在最后一小时中所有没有修改的会话（忽略对夏令时的更改），通过发出 `SQL` 进行清理任务，代码如下：

```
delete from sessions
where now() - updated_at > 3600;
```

对于基于 `DRb` 的解决方案，逾期是在 `DRb` 服务器进程内进行的。或许想要与会话数据散列的所有项放在一起记录时间戳，可以运行单独线程（甚至是独立进程），定期删除该散列的所有项。

在所有情况下，当不再需要会话时（如用户退出登录），应用程序可以帮助这个进程，通过调用方法 `reset_session` 对会话进行删除。

20.3.5 闪存：行为间通信

当使用 `redirect_to` 将控制转移到另外的行为时，浏览器会生成单独请求来调用该行为。在应用程序中，借助于控制器对象 — 实例变量的新生实例（`fresh instance`），将处理该请求，这些变量已经在原有的行动中得到了设置，但还没有应用于处理重定向代码。但有时需要在这两个实例之间进行通信。使用名为 `flash`（闪存）的方法可以做到这一点。

闪存是临时的数值暂存器。这是以散列方式进行组织的，且以会话数据形式进行存储的，因此，可以存储与键关联的值并且以后检索它们。它具有特殊属性。在默认情况下，在处理请求期间存入的闪存值将用于处理紧随其后的请求。一旦处理了第二个请求，就从闪存中删除这些值。

最常见的使用闪存方法可能是，把错误和信息字符串从一个行为传递到下一个。这里的意图是，第一个行为通告了一些条件、创建一条描述该条件的消息并且重定向到单独行动。通过存储闪存中的消息，第二个行为是可以访问该消息文本，并在视图中使用它。

有时可以很简单地使用闪存，其方法是把消息传递到当前行动的模板。例如，方法 `display` 想要输出活泼的横幅广告，如果没有其他的话，那么就会吸引更多的注意。它不需要把消息传递到下一个行为——这仅仅为了在当前请求中使用。要做到这一点，可以使用方法 `flash.now`，它更新该闪存，但不添加到会话数据中。

方法 `flash.now` 创建了短暂的闪存项，但是方法 `flash.keep` 则正好相反，使得当前正处于闪存的闪存项停留用于另外的请求周期。如果没有传递参数到方法 `flash.keep`，那么将保留所有闪存内容。

闪存可以存储比文本消息更多的信息——可以使用它们在各种行动之间传递各种不同的信息。显然，对于较长期保存的信息，要使用会话（可能要与数据库相连接）来存储数据，但想要从一个请求把参数传递到下一个，闪存是很重要的。

由于闪存数据存储在会话中，所以适用于所有一般规则。特别是，每一个对象必须是可序列化的。这里强烈建议，在闪存中只传递一些简单对象。

20.3.6 过滤器

过滤器可以使你编写控制器代码，这些代码封装由行为所完成的处理过程——可以一次编写一段代码，且在控制器（或控制器的子类）中任何行为之前或之后调用它们。这应是一个强大工具。使用过滤器，可以实现身份验证方案、日志记录、响应压缩甚至响应定制。

Rails 支持三种类型的过滤器：前置、后置和前后置。在执行行动之前、之后以及任何时候，调用过滤器。这取决于如何定义它们，它们要么作为控制器内方法运行，要么当运行它们时传递给控制器对象。无论哪种方式，它们都获得请求和响应对象的细节，以及其他控制器的属性。

20.3.7 前置和后置过滤器

正如从标题所看到的，在行动之前或之后调用前置和后置过滤器。Rails 为每个控制器维护两种过滤器链。当控制器即将完成行为时，它执行在链上的所有前置过滤器。在运行后置过滤器链之前，执行该行动。

过滤器可以被动触发，由控制器进行监视活动。这样它们可以主动参与请求处理中。如果前置过滤器返回 `false`，那么这个过滤器链处理将终止，而该行动也是无法运行的。过滤器也可能呈现输出或重定向请求，在这种情况下，永远不会调用原有的行动。

在 14.3 节中关于商店示例的管理授权部分曾经使用了过滤器。在那里定义了授权方法，如果当前会话没有已登录用户，那么该方法重定向到登录页面。接着编写了前置过滤器的方法，以达到对管理控制器所有行动进行监视。

过滤器声明也接收代码块和类名。如果指定了代码块,那么用当前控制器作为参数调用它。如果是类,那么将用控制器作为参数,调用其过滤器类方法。

在默认情况下,过滤器适用于控制器中的所有行为(以及该控制器的任何子类)。可以使用选项 `:only` 修改它,这需要一个或几个要过滤的行动,而使用选项 `:except`,则列出了从过滤中所排除的行动清单。

`before_filter` 和 `after_filter` 声明追加到控制器的过滤器链。使用变种方法 `prepend_before_filter` 和 `prepend_after_filter` 可以把过滤器放到该链过滤器最前端。

可用后置过滤器来修改发出的响应,如需要时可更改标头和内容。有些应用程序使用此技术,以控制器模板所生成的内容完成全局替代(例如,在响应体中,用字符串 `<customer/>` 代替客户名)。另一种用途是,如果用户浏览器支持这种响应,那么可能就要压缩它。

前后置过滤器封包了行动的执行。可以采用两种不同的风格来编写前后置过滤器。第一种风格,过滤器是单独一段代码。在执行行为之前,调用该代码。如果过滤器代码调用方法 `yield`,那么就执行这个行为。当这个行为完成后,过滤器代码继续执行。

因此,在该方法 `yield` 之前代码像是前置过滤器一样,而在方法 `yield` 之后代码就是后置过滤器。如果过滤器代码从未调用方法 `yield`,那么这个行动就不会运行——这与前置过滤器返回 `false` 是一样的。

前后置过滤器的好处是,可以保留跨行动整体调用。不但可以给方法 `around_filter` 传递方法名,还可以给它传递代码块或过滤器类。

如果使用代码块作为过滤器,那么它会传递两个参数:控制器对象和行动代理。对于第二个参数,使用方法 `call` 来调用原有的行动。

第二个风格是把对象作为过滤器进行传递。这个对象应实现称为 `filter` 的方法。将这种方法传递到控制器对象。这等价于调用行为。

就像前置和后置过滤器一样,前后置过滤器也接收参数 `:only` 和 `:except`。在默认情况下,前后置过滤器以不同的方式添加到过滤器链:先添加的前后置过滤器先执行。在现有前后置过滤器内,将嵌套随后添加的前后置过滤器。

20.3.8 过滤器继承

如果包含过滤器类控制器具有子类的话,那么父类的过滤器将不仅在子对象上,而且也在父类中运行。不过,子类定义的过滤器将不会在父类中运行。

如果不希望特定的过滤器运行在子类的控制器中,那么可以用方法 `skip_before_filter` 和 `skip_after_filter` 声明覆盖默认处理过程。它们接收参数 `:only` 和 `:except`。

可以使用方法 `skip_filter` 跳过任何前置、后置和前后置过滤器。不过,只有作为(符号的)方法名所指定的过滤器,才能实现这一点,详细参见 14.3 节。

20.4 本章小结

本章介绍了行动调度与行动控制器是如何使服务器响应请求进行合作的。这一点的重要性没能得到足够的强调。在几乎所有的应用程序中,这是应用程序创造力表现的主要地方。路由和控

制器是行动存在的地方，而 Active Record 和 Action View 几乎都是主动的。

本章开始时涵盖了 REST 概念，对于这种方式的灵感，Rails 采纳了路由请求。从前面的内容中可以看出，如何提供 7 种基本行为，以及如何添加更多的行动。同时还介绍了如何选择数据表述（例如，JSON 或 XML）。还讨论了如何进行路由测试。

接下来，所涉及过的环境是，行为控制器（Action Controller）提供了行为，或者说方法，它提供了呈现和重定向。最后，还介绍了会话。闪存和过滤器中的每一个都可用于应用程序控制器中。

最后，说明了曾经如何在 Depot 应用程序中使用这些概念。现在已经看到了在应用中的每一个，并且已经揭示了其背后的理论，如何组合和使用这些概念只能由自己的创造力来完成。

在下一章中将涵盖其余的 Action Pack 组件部分，即 Action View，它处理呈现结果。

第 21 章

Action View 模块

在本章中，我们将学习：

- 模板
- 表单：字段和文件上传
- 帮助程序
- 布局和局部页面

前面已经解释了路由组件是如何确定要使用的控制器，以及控制器是如何选择行为。还知道了该控制器及其行为是如何为用户决定呈现什么的。但是，呈现过程发生在行为结束部分，通常都会调用模板。这是本章所要讨论的全部内容。ActionView（行为视图）模块封装了所有要呈现模板所需要的功能，这些功能主要是生成 HTML、XML 或 JavaScript，并返回给客户端。同它的字面意思一样，ActionView 模块是 MVC 模式中的视图部分。

本章将首先介绍模板，而 Rails 为其提供了大量的选项；然后阐述一系列不同的输入类型：表单、文件上传和链接，用户可以用它们来输入数据；最后将解释一些减少维护成本的方法，如帮助程序、布局以及局部模板。

21.1 使用模板

当编写视图时，其实就是编写模板：所编写内容扩充后才成为最终看到的样子。为了理解模板是怎么工作的，需要学习下面这些内容：

- 模板存放的位置
- 模板运行的环境
- 模板包含的内容

21.1.1 模板存放的位置

函数 `render` 期望在当前应用程序目录 `app/views` 下找到模板。在这个目录下，默认是对每个控制器所有视图具有单独的子目录。例如，Depot 应用程序包含产品（products）和商店（store）两个控制器，于是，模板便会存放在两个目录 `app/views/products` 和 `app/views/store` 下。通常用相应控制器的行为名称来命名每个目录的模板。

也可以不以行为名称来命名模板。对于这样的模板，在控制器中呈现模板如下所示：

```
render(:action => 'fake_action_name')
render(:template => 'controller/name')
render(:file => 'dir/template')
```


最后的一个调用表明：可以把模板存放在文件系统的任何位置。如果想在应用程序间共享模板，这样做就挺有用的。

21.1.2 模板运行的环境

模板是由固定文本和代码混合而成的。模板代码将动态内容添加到响应中。运行代码的环境使得代码可访问控制器所设置的信息。

- 在模板中可以使用控制器的所有实例变量，这是行为向模板传递数据的方式。
- 在视图中可以把控制器对象的方法 `flash`、`headers`、`logger`、`params`、`request`、`response` 和 `session` 用作 accessor 方法。不过，除了方法 `flash` 之外，视图代码通常不应该直接使用这些方法，这是因为怎么处理它们取决于控制器。然而，在调试时，使用这些方法还是很方便的。例如，下面的 `html.erb` 模板使用方法 `debug`，以显示会话内容、参数详细以及

当前响应：

```
<h4>Session</h4> <%= debug(session) %>
<h4>Params</h4> <%= debug(params) %>
<h4>Response</h4> <%= debug(response) %>
```

- 使用属性名 `controller` 可访问当前控制器对象。这样使得模板可以调用该控制器的所有公共方法（包括 `ActionController` 方法）。
- 在属性 `base_path` 中存放模板的根目录路径。

21.1.3 模板包含的内容

在默认情况下，Rails 支持三种模板类型：

- **Builder（构建器）** 模板使用构建器库去构造 XML 响应。在 25.1 节中将讨论更多关于构建器的内容。
- **ERb 模板** 是一种内容和内嵌 Ruby 语句的混合物，通常用来生成 HTML 页面。在 25.2 节将讨论更多关于 ERb 的内容。
- **RJS 模板** 用来生成 JavaScript 语言代码，而将在浏览器中执行该语言代码，并且通常用来与 Ajax 的网页交互。在 11.2 节中曾经了解并使用过了 RJS 模板。

今天在实际中将使用到的模板估计多数还是 ERb 模板。事实上，在开发 Depot 应用程序中，曾经大规模地使用了 ERb 模板。

本章讲到现在，重点讲的都是构建输出内容。而在第 20 章中着重点是对输入的处理。对于设计优秀的应用程序，输入与输出不是毫无关联的：所创建的输出内容包含了表单、链接和按钮，同时它们也引导了用户下一步进行的输入操作。正如所期待的那样，在这方面 Rails 也提供了相当多的帮助。

21.2 生成表单

HTML 提供了许多元素、属性及其值，它们控制了怎样采集输入。虽然可以在模板里强行

编写表单代码，但是这个实在没有必要。

在本节中将介绍一些 Rails 所提供的 helpers 程序，用来辅助表单生成过程。21.5 节将展示怎样创建自己的帮助程序。

HTML 提供了不少从表单采集数据的方法。图 21.1 展示了几个比较常用的方法。注意，这个表单本身不代表任何一种常用方式，通常情况下只会用到其中一部分方法来进行数据采集。

图 21.1 表单中数据输入的几种常用方式

下面来看看生成这个表单的模板文件：

```
views/app/views/form/input.html.erb
Line 1 <%= form_for(:model) do |form| %>
      <p>
        <%= form.label :input %>
        <%= form.text_field :input, :placeholder => 'Enter text here...' %>
      </p>
      <p>
        <%= form.label :address, :style => 'float: left' %>
        <%= form.text_area :address, :rows => 3, :cols => 40 %>
      </p>
      <p>
        <%= form.label :color %>:
        <%= form.radio_button :color, 'red' %>
        <%= form.label :red %>
        <%= form.radio_button :color, 'yellow' %>
        <%= form.label :yellow %>
        <%= form.radio_button :color, 'green' %>
        <%= form.label :green %>
      </p>
      <p>
        <%= form.label 'condiment' %>:
        <%= form.check_box :ketchup %>
        <%= form.label :ketchup %>
        <%= form.check_box :mustard %>
        <%= form.label :mustard %>
      </p>
```

```

-      <%= form.check_box :mayonnaise %>
-      <%= form.label :mayonnaise %>
30    </p>
-
-    <p>
-      <%= form.label :priority %>:
-      <%= form.select :priority, (1..10) %>
35    </p>
-
-    <p>
-      <%= form.label :start %>:
-      <%= form.date_select :start %>
40    </p>
-
-    <p>
-      <%= form.label :alarm %>:
-      <%= form.time_select :alarm %>
45    </p>
-    <%= end %>

```

在这个模板中可以看到使用了若干标签 label，例如在第 3 行中，使用标签 label 将文本内容和特定属性的输入框关联起来。如不加设定，将标签 label 的文本内容默认设置为对应的属性名。

帮助方法 text 和 text_area（分别见第 4 行和第 9 行）用来采集单行或多行输入框。在默认情况下，可以指定方法 placeholder（占位符），当用户输入数据之前，在输入框中显示它。不是所有的浏览器都支持这一特性，那些不支持的将只显示一个空框。由于在输入框中能不能显示占位符对用户的影响微乎其微，因此没有必要太纠结于这个问题，放心使用这个特性就好了，因为那些能看见它的人会立刻受益。

占位符是 HTML 5 所提供的诸多小“贴身和成品”的特性之一，还是那句话，即便是用户安装的浏览器还不具备，Rails 也已经为支持新特性做好准备。使用帮助方法 search_field、telephone_field、url_field、email_field、number_field 以及 range_field，可对特殊类型的输入进行提示。浏览器如何利用这些信息是千变万化的。为了更清楚地识别其功能，有些浏览器可能会产生略微不同的显示效果。例如，对于搜索输入框，Mac 浏览器 Safari 将显示成圆角效果，并且插入一个小 x，让用户可以在输入时清空输入框。而另外一些浏览器会提供附加的验证功能。例如，对于 URL 输入框，浏览器 Opera 将先验证再提交。对于邮件地址输入框，平板电脑 iPad 甚至会自动改变虚拟键盘的布局，其中将列出像 @ 之类的字符。

对于这些功能的支持，虽然不同的浏览器不尽相同，但是那些不提供额外支持的浏览器并不会受到什么影响，而会显示未加修饰的普通输入框。古人言，守株待兔。如果有一个输入框，并计划用来接收邮件地址，那么就不要再单纯使用 text_field（文本框），向前跨一步，开始使用 email_field（邮件框）。

第 14、24 和 34 行代码举例说明了三种不同的形式，用来提供一组限制选项。虽然显示效果可能因浏览器不同而不同，但是所有浏览器都能很好地支持全部这些功能。函数 select 是非常灵活的，可以表达简单的 Enumeration（比如例子中给出的这些）、“名称/数值”成对数组或者

散列。利用各种不同的数据源（包括数据库），大量的表单选项帮助程序^①生成这样的列表。

最后，第 39 和 44 行分别列举了日期和时间的提示。正如你现在所期待的那样，Rails 也提供了大量的选项^②。

在这个例子中没有列举的方法是 `hidden_field`（隐藏框）和 `password_field`（密码框）。虽然完全不会显示隐藏输入框，但将其值传送到服务器。这可以作为在会话中存储瞬态数据的一种方式，使请求数据传送到下一个。虽然将密码输入框本身显示出来，但是会把具体内容屏蔽起来。

对于大多数初学者而言，这些内容已经可以满足绝大部分需求。如果还要寻找更多的需求，那么有希望找到可直接使用的帮助程序或者插件。可以从 Rails 指南开始，那是不错的选择^③。

下面看看怎么处理数据表单的提交。

21.3 处理表单

在图 21.2 中可以看到模型的各种属性，它们从控制器传到视图中，再传到 HTML 页面，并再一次返回模型。模型对象拥有若干属性，如 `name`、`country` 和 `password`。模板使用帮助函数去创建 HTML 表单，这样用户可以编辑模型的数据。注意表单框是如何命名的。如属性 `country` 对应于以 `user[country]` 命名的 HTML 输入框。

当用户提交表单时，将原始的 POST 数据返回给应用程序。Rails 将提取表单框，并构建散列 `params`。将简单值（比如，由路由从表单行为中提取 `id` 框）直接存进散列里。但是，如果参数名含有方括号的话，那么 Rails 认为它是更复杂数据的一部分，并将构建散列来储存它。在这个散列内，方括号里的字符串将作为关键字。当参数名本身具有多组方括号时，这一过程可以重复进行下去。

来自参数

```
params
id=123      { :id => "123" }
user[name]=Dave    { :user => { :name => "Dave" } }
user[address][city]=wien  { :user => { :address => { :city => "wien" } } }
```

在整个集成过程的最后部分中，模型对象会接收散列中新的属性值，可以使用下面的语句：

```
user.update_attributes(params[:user])
```

Rails 集成要比这个过程更深入。看看图 21.2 的文件 `.html.erb`，可以发现模板使用一系列帮助方法（如 `form_for` 和 `text_field`），用来创建表单的 HTML 内容。

在继续学习之前，需要提醒一下：变量 `params` 不仅可以作为简单文本来使用，而且还可以上传整个文件。下面将对此进行介绍。

① <http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html>

② <http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html>

③ http://edgeguides.rubyonrails.org/form_helpers.html

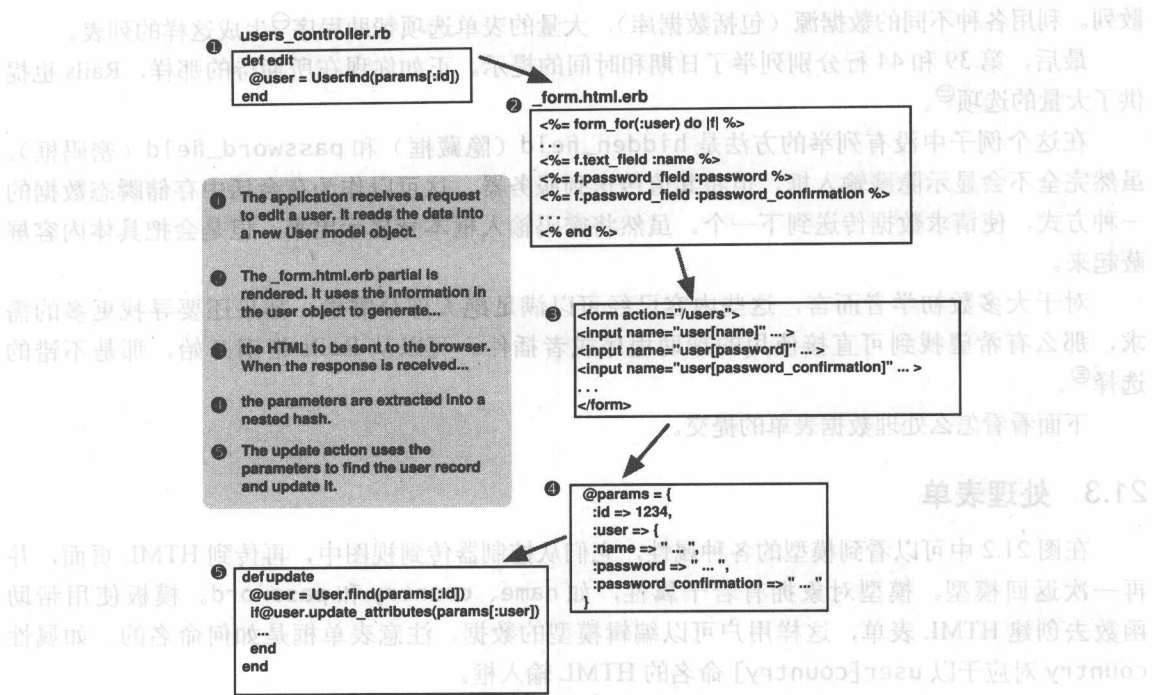


图 21.2 模型、控制器和视图协同工作

21.4 上传文件到 Rails 应用程序

在应用程序系统中可能会允许用户上传文件。例如，程序缺陷报告系统将附加日志文件以及代码示例到报告中，又如博客应用程序可以上传小幅图片给文章配上。

在 HTTP 中，把文件作为 POST 消息 multipart 或 form-data（表单数据）进行上传。如同其字面意思一样，使用表单生成这些消息。在表单中，可以使用一个或者几个具有属性 `type="file"` 的 `<input>` 标签。当浏览器呈现它们时，这些标签是由名称选择文件。当随后提交该表单时，就将一个或多个文件随同表单其他数据一起传回服务器。

为了阐明文件上传过程，这里将展示一些上传图片及其显示评论的代码。为了实现这个效果，首先需要有一个表 `pictures` 来储存相关数据：

```
e1/views/db/migrate/20110211000004_create_pictures.rb
```

```
class CreatePictures < ActiveRecord::Migration
```

```
def self.up
```

```
create_table :pictures do |t|
```

```
  t.string :comment
```

```
  t.string :name
```

```
  t.string :content_type
```

```
  # If using MySQL, blobs default to 64k, so we have to give
```

```
  # an explicit size to extend them
```

```
  t.binary :data, :limit => 1.megabyte
```

```
end
end
```

```
def self.down
  drop_table :pictures
end
end
```

下面将创建假想的上传控制器，而该控制器仅用来演示这个过程。这里的行为 `get` 与通常用法一样，它简单地创建新的图片对象并且呈现表单。

```
e1/views/app/controllers/upload_controller.rb
```

```
class UploadController < ApplicationController
  def get
    @picture = Picture.new
  end
  # . . .
end
```

这个模板 `get` 包含用来上传图片的表单及其评论。注意这里是如何覆盖编码类型，这样允许数据同响应一起返回。

```
e1/views/app/views/upload/get.html.erb
```

```
<%= error_messages_for("picture") %>

<% form_for(:picture,
  :url => {:action => 'save'},
  :html => { :multipart => true }) do |form| %>

  Comment:      <%= form.text_field("comment") %><br/>
  Upload your picture: <%= form.file_field("uploaded_picture") %><br/>

  <%= submit_tag("Upload file") %>
<% end %>
```

这个表单另有一个精妙之处，图片上传给属性 `uploaded_picture`，不过，数据库表并没有包含该属性的字段。这意味着该模型背后肯定有些奥妙。

```
e1/views/app/models/picture.rb
```

```
class Picture < ActiveRecord::Base
  validates_format_of :content_type,
    :with => /\Aimage/,
    :message => "--- you can only upload pictures"

  def uploaded_picture=(picture_field)
    self.name = base_part_of(picture_field.original_filename)
    self.content_type = picture_field.content_type.chomp
    self.data = picture_field.read
  end

  def base_part_of(file_name)
```

```
File.basename(file_name).gsub(/[\w._-]/, '')
end
end
```

这里定义了存取 (accessor) 方法 `uploaded_picture=`, 用来接收该表单上传的文件。这个表单所返回的对象是很有趣的混合体 (hybrid)。一方面, 其形式类似于文件, 可以使用方法 `read` 来读取其内容; 另一方面, 可将获取图片数据并入到字段 `data`。对此它具有两个属性: `content_type` 和 `original_filename`, 可以在已上传文件的元数据中获得它们。存取方法把所有的属性分离出来, 生成单一对象且存在数据库中。

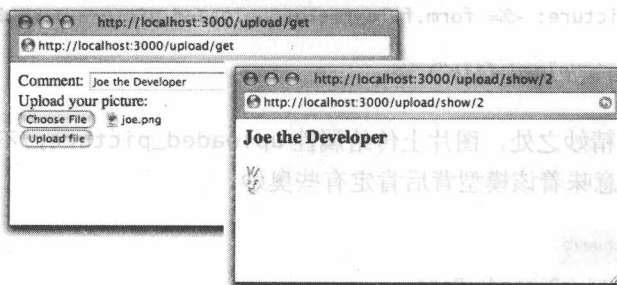
值得注意的是, 这里还添加了简单的验证, 用来检查内容类型是否具有 `image/xxx` 形式。请不要上传 JavaScript。

控制器行为 `save` 是完全依照惯例的:

```
e1/views/app/controllers/upload_controller.rb

def save
  @picture = Picture.new(params[:picture])
  if @picture.save
    redirect_to(:action => 'show', :id => @picture.id)
  else
    render(:action => :get)
  end
end
```

现在数据库中已经存储了一张图片, 该怎么显示它呢? 一种方法是使用其自己的 URL, 并简单地在标签 `image` 中链接该 URL。例如, 使用类似于 `upload/picture/123` 的 URL, 可以返回图片 123 的图像。这里需要使用 `send_data` 将图像返回到浏览器。



值得注意的是怎么设置内容类型和文件名——这让浏览器解析这些数据, 且提供默认名称让用户保存图像:

```
e1/views/app/controllers/upload_controller.rb

def picture
  @picture = Picture.find(params[:id])
  send_data(@picture.data,
            :filename => @picture.name,
            :type => @picture.content_type,
            :disposition => "inline")
end
```


最后实现显示评论和图片的行为 show。这个行为简单地加载图片模型对象：

```
e1/views/app/controllers/upload_controller.rb
def show
  @picture = Picture.find(params[:id])
end
```

在该模板中，标签 image 链接到返回图片内容的那个行为。在图 21.3 中，可以看到行为 get 和 set 页面显示：

```
e1/views/app/views/upload/show.html.erb
<h3><%= @picture.comment %></h3>


```

通过缓存行为 picture，可以优化这一过程的性能（第 22 章将开始讨论缓存）。

如果想要用更简单的方法处理上传和存储图像，那么可以看看由 thoughtbot 所开发的插件 Paperclip[Ⓐ]或者由 Rick Olson 所开发的插件 attachment_fu[Ⓑ]。先创建包含一组给定字段的数据库表（参见 Rick 博客文章），然后该插件将自动管理存储已上传的数据和上传的元数据。与之前方法不同的是，它既可以在文件系统中，也可以在数据库表中存储上传内容。

表单和上传只是 Rails 所提供的帮助程序的两个例子。下一节将阐述如何提供自己的帮助程序，并且介绍一些其他创建帮助程序的 Rails 代码。

21.5 使用帮助程序

前面说过可以在模板中放入代码。现在要更改这个观点了。在模板里加入“一些”代码完全是可接受的——这让模板更加动态，不过，在模板里添加太多代码是一种不良的编程风格。

对此存在 3 个理由。首先，越多代码在应用程序视图里，越容易乱了规矩，而且开始把应用程序层的功能添加到模板代码里。这绝对是一个效率很低的形式，应用程序层的内容应该放在控制器和模型层，这样它们才变得通用。当添加显示应用程序的新方法时，这样做是值得的。

其次，文件 html.erb 本质上就是 HTML。在编辑它时，就是在编辑 HTML 文件。如果由专业设计人员对页面进行布局，那么他们希望同 HTML 打交道。对于他们而言，放上一大堆 Ruby 代码，只会变得难以工作。

最后，在视图中嵌入代码难以进行测试，若把代码拆分放到帮助模块里，那么这些代码是分离的，并且作为独立单元进行测试。

Rails 提供了一个很妙的折中方案，这就是帮助程序形式。简单地说，helper（帮助程序）就是包含了若干辅助视图方法的模块。帮助方法都是以输出为中心的。其存在目的就是生成 HTML、XML 或 JavaScript——帮助程序就是扩展模板功能。

Ⓐ <https://github.com/thoughtbot/paperclip#readme>

Ⓑ https://github.com/technoweenie/attachment_fu

21.5.1 自定义的帮助程序

在默认情况下，每个控制器都有自己的帮助程序模块，另外还有应用程序层面的帮助程序 `application_helper.rb`。这一点不值得惊讶，Rails 做了一些默认设置：把帮助程序连接到控制器及其视图。当所有的控制器可以使用所有的视图帮助程序时，通常这是一个比较实际的管理帮助程序的方法。对于关联到控制器 `ProductController` 的视图而言，帮助程序是唯一的，该程序往往放在 `app/helpers` 目录下，其文件名称是 `product_helper.rb`，该文件拥有帮助程序模块 `ProductHelper`。不用记住所有的细节——`rails generate controller` 脚本会自动生成桩（stub）帮助程序模块。

在 11.4 节中，曾经创建过这样自定义的帮助函数 `hidden_div_if`，其作用是在特定条件下隐藏图表。这里可以使用同样的办法，对应用程序的页面布局加以整理。如下所示：

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
```

下面进行代码转移工作，把页面标题放到帮助函数里。由于代码处在 `store` 控制器中，所以需要编辑在目录 `app/helpers` 下的文件 `store_helper.rb`。

```
module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

现在视图代码可简单地调用帮助方法：

```
<h3><%= page_title %></h3>
```

（把呈现整个标题移到单独的局部模板中，并且共享所有控制器的视图，这会进一步减少重复，但是在 21.6 节中将更深入地讨论这一点。）

21.5.2 格式和链接帮助程序

Rails 自带了一套所有视图都可以使用的帮助函数。这一节只介绍其中比较重要的一部分，但是可以在 Action View 的 RDoc 文档中找到详细内容——那里面介绍了很多的功能。

21.5.3 格式帮助程序

一组处理日期、数字和文本的帮助程序：

```
<%= distance_of_time_in_words(Time.now, Time.local(2010, 12, 25)) %>
5 months
```

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, false) %>
1 minute
```

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, true) %>
Half a minute
```

```
<%= time_ago_in_words(Time.local(2010, 12, 25)) %>
```

```

3 months
<%= number_to_currency(123.45) %>
$123.45
<%= number_to_currency(234.56, :unit => "CAN$", :precision => 0) %>
CAN$235
<%= number_to_human_size(123_456) %>
120.6 KB
<%= number_to_percentage(66.66666) %>
66.667%
<%= number_to_percentage(66.66666, :precision => 1) %>
66.7%
<%= number_to_phone(2125551212) %>
212-555-1212
<%= number_to_phone(2125551212, :area_code => true, :delimiter => "|") %>
(212) 555 1212
<%= number_with_delimiter(12345678) %>
12,345,678
<%= number_with_delimiter(12345678, :delimiter => "_") %>
12_345_678
<%= number_with_precision(50.0/3, :precision => 2) %>
16.67

```

使 YAML 格式, 函数 debug 将输出其参数并没有显示结果, 这样就可以在 HTML 页面中显示它。查看模型对象的和请求参数的数值, 这是很有用的。

```

<%= debug(params) %>
--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test

```

另外, 还有一组处理文本的帮助程序。存在截短字符串和高亮字符串单词的方法。

```
<%= simple_format(@trees) %>
```

格式字符串、履行线和分段。下面提供 Joyce Kilmer 的诗《树》(Trees) 的文本, 且将 HTML 添入对其进行格式化:

```

<p>I think that I shall never see
<br />A poem lovely as a tree.</p>
<p>A tree whose hungry mouth is prest

```

```
<br />Against the sweet earth's flowing breast;
```

```
</p>
```

```
<%= excerpt(@trees, "lovely", 8) %>
```

```
...A poem lovely as a tre...
```

```
<%= highlight(@trees, "tree") %>
```

```
I think that I shall never see
```

```
A poem lovely as a <strong class="highlight">tree</strong>. A <strong
```

```
class="highlight">tree</strong>whose hungry mouth is prest
```

```
Against the sweet earth's flowing breast;
```

```
<%= truncate(@trees, :length => 20) %>
```

```
I think that I sh...
```

还有将名词变为复数的方法。

```
<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
```

```
1 person but 2 people
```

如果想要完成那些漂亮网站所做的事情，比如自动为 URL 和邮件地址加上超链接，那么就可以找到相应的帮助程序。另外，还有帮助程序可以从文本提取超链接。

回顾 6.2 节，在那里曾经看到过，可以使用帮助方法 `cycle`，从一个序列中每次调用它，以返回连续值，在需要时，重复该序列。对于处理表或列表，产生行记录的交替风格，这是经常使用的。还有函数 `current_cycle` 和 `reset_cycle` 也是可用的。

最后，如果编写博客类似网站的应用程序，或者想要用户对商店添加评论，那么可以让他们使用 Markdown (BlueCloth)[Ⓔ] 或者 Textile (RedCloth)[Ⓕ] 格式的文本。这是两个很简单的文本格式工具，以非常简单和人性化的标记处理文本且将其转化为 HTML。

21.5.4 链接到其他页面和资源

模块 `ActionView::Helpers::AssetTagHelper` 和 `ActionView::Helpers::UrlHelper` 包含了大量的函数，它们引用资源外部到当前模板中。其中最常用的是方法 `link_to`，该方法创建指向应用程序其他行为的超链接。

```
<%= link_to "Add Comment", new_comments_path %>
```

方法 `link_to` 的第一个参数是显示针对该链接的内容，第二个是指向链接目标的字符串或者散列。

第三个可选参数是所生成链接的 HTML 属性：

```
<%= link_to "Delete", product_path(@product),
  { :class => "dangerous", :method => 'delete' }
%>
```

Ⓔ <https://github.com/rtomayko/rdiscount>

Ⓕ <http://redcloth.org/>

这里的第三个参数还支持两个附加选项，它们可以修改该链接的反应形式。每一次请求都要得到浏览器运行 JavaScript 的允许。

选项 `:method` 是一种黑客手段——使该链接窥视应用程序，就像方法 `POST`、`PUT` 或者 `DELETE` 创建该请求那样，而不是通常的 `GET` 方法。这是通过创建一系列 JavaScript 代码来实现的，当单击链接时，提交该请求——如果浏览器禁用 JavaScript 的话，那么将会生成方法 `GET`。

选项 `:confirm` 提供简单消息。要是有的话，那么将生成 JavaScript 来显示该消息，并在执行链接之前，得到用户确认。

```
<%= link_to "Delete", product_path(@product),
      { :class => "dangerous",
        :confirm => "Are you sure?",
        :method => :delete} %>
```

方法 `button_to` 的工作原理与方法 `link_to` 是一样的，而不同的是：生成了放在自己表单里的按钮，而不是简单的超链接。这是具有边际效应（side effect）的行动链接的首选方法。不过，这些按钮都有自己的表单，这便产生了一系列限制：它们不能出现内联（inline），也不能出现在其他表单内。

Rails 还有条件链接函数，就是当某些条件满足时，才生成相应超链接，否则只显示其链接名称。方法 `link_to_if` 和 `link_to_unless` 多了一个条件参数，其他的与方法 `link_to` 一样。如果条件是 `true`（对于 `link_to_if`）或 `false`（对于 `link_to_unless`），那么使用剩余参数生成通常的链接，否则将显示普通链接名称（无超链接）。

帮助方法 `link_to_unless_current` 创建侧边栏菜单，其中将当前页面名称仅显示为链接名称，而其他的是超链接及其名称。

```
<ul>
  <% %w{ create list edit save logout }.each do |action| %>
    <li>
      <%= link_to_unless_current(action.capitalize, :action => action) %>
    </li>
  <% end %>
</ul>
```

如果当前行为（current action）就是给定行为，同时有效地提供该链接的一个替代，那么可以将方法 `link_to_unless_current` 传递到可计算的代码块。还有帮助方法 `current_page` 简单地可测试当前请求 URI 是否由给定选项所生成。

像方法 `url_for` 一样，方法 `link_to` 及其类似方法也支持绝对路径 URL：

```
<%= link_to("Help", "http://my.site/help/index.html") %>
```

帮助方法 `image_tag` 用来创建 `` 标签。可选参数 `:size`（格式是宽 × 高）用来定义图片的大小，也可以使用参数 `:width` 和 `:height`，分别给出其宽度和高度：

```
<%= image_tag("/images/dave.png", :class => "bevel", :size => "80x120") %>
<%= image_tag("/images/andy.png", :class => "bevel",
      :width => "80", :height => "120") %>
```

如果没有提供选项 `:alt`，那么 Rails 将使用图片文件名合而为一。如果图片路径不以 “/” 符号开头，则 Rails 将假定它存在于目录 `/images` 下。

通过组合方法 `link_to` 和 `image_tag`，就可以使图片变成超链接。

```
<%= link_to(image_tag("delete.png", :size => "50x22"),
  product_path(@product),
  { :confirm => "Are you sure?",
    :method => :delete}) %>
```

帮助方法 `mail_to` 用来创建超链接 `mailto:`，具体地说，在单击后会加载客户端的邮件应用程序。它需要邮件地址、链接名称以及一些 HTML 选项，其中包括：`:bcc`、`:cc`、`:body` 以及 `:subject` 选项，用来初始化相关邮件输入框。最后，奇妙的选项 `:encode=>"javascript"`，使用客户端的 JavaScript 来隐藏所生成的链接，这样爬虫 spider[⊖] 就不会探测到网站上的邮件地址。不过，遗憾的是，这也意味着，如果浏览器禁用了 JavaScript，那么用户便看不到邮件链接了。

```
<%= mail_to("support@pragprog.com", "Contact Support",
  :subject => "Support question from #{@user.name}",
  :encode => "javascript") %>
```

也可以使用弱隐藏手段，选项 `:replace_at` 和 `:replace_dot` 可以用其他字符串来替代所要显示邮件名称的 “@” 和 “.” 符号。但是这样做不能骗过爬虫。

模块 `AssetTagHelper` 也包括了一些帮助程序。它们可以很容易地实现：从页面链接到样式文件和 JavaScript 文件，以及自动创建 Atom Feed 链接。在 Depot 应用程序页面布局中曾经创建过样式链接，在 HTML 头中使用了帮助方法 `stylesheet_link_tag`：

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

```
<%= stylesheet_link_tag "scaffold" %>
<%= stylesheet_link_tag "depot", :media => "all" %>
```

函数 `stylesheet_link_tag` 还可以接收参数 `:all`，它指明了包含样式目录的所有样式文件。添加 `:recursive =>true` 也将让 Rails 包含所有子目录中的样式文件。

函数 `javascript_include_tag` 接收一个 JavaScript 文件名（假定存在路径 `public/javascripts` 下）列表，并且创建 HTML 使其加载到页面中。除了参数 `:all` 之外，方法 `javascript_include_tag` 还接收值 `:defaults` 作为参数，它像快捷方式一样，让 Rails 去加载文件 `prototype.js`、`effects.js`、`dragdrop.js` 和 `controls.js`，此外还有文件 `application.js`，如果存在的话。最后一个文件用来为应用程序添加用户 JavaScript 代码。

RSS 或者 Atom 链接是一种 HTML 标头项，它指向应用程序的 URL。当 URL 是可访问的时，应用程序会返回合适的 RSS 或者 Atom XML：

⊖ 邮件地址搜集软件。——译者注


```
<html>
<head>
  <%= auto_discovery_link_tag(:atom, products_url(:format => 'atom')) %>
</head>
...

```

最后，模块 `JavaScriptHelper` 也定义了与 JavaScript 一起工作的帮助程序。它们会创建可在浏览器中运行的 JavaScript 片段，用来生成特定的效果，让页面同应用程序动态互动。

在默认情况下，假定图像和样式资源存在于应用程序目录 `public` 的目录 `images` 和 `stylesheets` 下。如果给定资源 (asset) 标签方法的路径含有正斜杠 (/)，则该路径采用了绝对路径，且不使用前缀。有时把这些静态内容移到单独的服务器中或者当前服务器的不同路径下，这是有必要的。通过设置配置变量 `asset_host` 来实现这一点：

```
config.action_controller.asset_host = "http://media.my.url/assets"
```

虽然这些帮助程序看起来比较全面，但是 Rails 还提供更多的选择，每一新发行版本会引入新的帮助程序，并挑选少量退出或将其移到插件里，而这样它们可以有别于 Rails 本身的版本演变进程。现在是时候回顾一下在 18.1 节中所生成的在线手册，去看看 Rails 所提供的其他东西。

21.6 用页面布局和局部模板减轻维护工作

在本章前面中已经介绍过已分离的代码和 HTML 片段的模板。但是，Rails 背后的驱动理念之一是承诺 DRY 原则 (Don't Repeat Yourself, 不要重复你自己) 和排除需要重复的东西。然而，普通的网站里面却有大量的重复。

- 许多页面共享同样的页首、页尾和侧边栏。
- 多个页面包含同样的要呈现的 HTML 片段 (例如，博客网站可能会在多个地方显示同一篇文章)。
- 在多个地方出现同样的功能。很多网站都有标准搜索组件或者投票组件，它们会出现在大多数网站的侧边栏里。

Rails 提供了页面布局和局部模板，它们用来减少上面三种情况下需要重复的东西。

21.6.1 布局

Rails 呈现嵌入其他所要呈现页面中的页面。通常情况下，使用该特性可把来自行为的内容放入整个网站的标准页面帧内 (包括标题、页脚和侧边栏) 中。事实上，如果已经使用过 `generate` 脚本来创建基于脚手架的应用程序，那么就一直在使用这样的布局。

当 Rails 接收呈现来自控制器内模板的请求时，实际上呈现了两个模板。显然它呈现了所要求的模板 (或者当没有明确要求呈现任何东西时，就是呈现以行为名称所命名的默认模板)。此外，Rails 还会尝试去寻找并呈现布局模板 (紧接着将介绍如何寻找该布局模板)。如果找到了该布局，那么就把与行为相关的输出插入由布局模板所生成的 HTML 中。

我们来看一看布局模板：

```

<html>
  <head>
    <title>Form: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <%= yield :layout %>
  </body>
</html>

```

该布局输出了具有 head 和 body 的标准 HTML 页面。它使用当前的行为名作为页面标题，并包含了一个 CSS 文件。在 body 中，它调用了函数 yield，这就是神奇的地方。在呈现行为模板时，Rails 将其内容储存起来并标为 :layout。在布局模板中，调用 yield 取回这些内容。实际上，:layout 是当呈现时所返回的默认内容，这样就写代码 yield，而不要写代码 yield :layout，就可以了。从个人习惯来说，我们倾向于使用更加明确的版本。

如果模板 my_action.html.erb 包含这样的内容：

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

则浏览器会看到如下内容：

```

<html>
  <head>
    <title>Form: my_action</title>
    <link href="/stylesheets/scaffold.css" media="screen"
      rel="Stylesheet" type="text/css" />
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>

```

21.6.2 放置布局文件

你可能会想，Rails 做好了提供默认布局文件位置的工作，若想要不同的方式，就可以重写默认设置。

布局是与控制器相关的。如果 store 控制器正在处理请求，则在默认情况下 Rails 将在目录 app/views/layouts 下寻找布局 store（其文件扩展名通常为 .html.erb 或者 .xml.builder）。如果在目录 layouts 下创建名称为 application 的布局，那么它将作用于所有没有定义布局的控制器，否则的话，将要为它们定义布局。

在控制器中使用方法 layout 声明，可以重写布局文件名称。在最简单的情况下，声明采用字符串类型的布局名称作为参数。下面的声明将使得模板文件 standard.html.erb 或者 standard.xml.builder 成为 store 控制器所有行为的布局。Rails 将在目录 app/views/layouts 下寻找。

```
class StoreController < ApplicationController
```

```
  layout "standard"
```

```
  # ...
end
```

使用限定词 `:only` 和 `:except`，可以限定哪些行为具有服务于它们的该布局：

```
class StoreController < ApplicationController
```

```
  layout "standard", :except => [ :rss, :atom ]
```

```
  # ...
end
```

定义为 `nil` 的布局将关闭控制器布局。

有时需要实时改变一组页面的外观，例如，博客网站可以在用户登录后提供不同的侧边栏菜单，或者在线商店可在其停机维护时显示不同的页面。Rails 提供了动态布局来支持这一需求。如果 `layout` 声明参数是符号，则它将引向控制器实例函数，该函数返回要使用的布局名称。

```
class StoreController < ApplicationController
```

```
  layout :determine_layout
```

```
  # ...
```

```
  private
```

```
  def determine_layout
```

```
    if Store.is_closed?
```

```
      "store_down"
```

```
    else
```

```
      "standard"
```

```
    end
```

```
  end
```

```
end
```

控制器的子类使用其父类的布局，除非它们使用了 `layout` 指令。最后，通过向方法 `render` 传递 `:layout` 的选项，不同的行为可以使用特定的布局（或完全没有使用任何布局）来呈现：

```
def rss
```

```
  render(:layout => false) # never use a layout
```

```
end
```

```
def checkout
```

```
  render(:layout => "layouts/simple")
```

```
end
```

21.6.3 传递数据到布局

布局可以访问所有相同的数据，对于传统模板而言，这些数据也都是可用的。此外，在普通

模板中所设定的任何实例变量是可以在布局中使用的（因为在调用 `layout` 之前，将呈现常规模板）。这应该用来对布局的页头和菜单进行参数化处理。例如，布局可能包含以下内容：

```
<html>
  <head>
    <title><%= @title %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <h1><%= @title %></h1>
    <%= yield :layout %>
  </body>
</html>
```

特殊的模板可以通过分配变量 `@title` 来设置标题：

```
<% @title = "My Wonderful Life" %>
<p>
  Dear Diary:
</p>
<p>
  Yesterday I had pizza for dinner. It was nice.
</p>
```

实际上可以更深入一步，之前使用 `yield :layout` 将呈现模板嵌入布局中，这样的机制也可以用来生成模板的任意内容，而将该内容嵌入任意其他模板中。

例如，不同的模板有可能需要向标准页面侧边栏添加与自己模板相关的菜单项。在这些模板中采用方法 `content_for` 机制来定义内容，并用方法 `yield` 在布局中将该内容嵌入侧边栏。

在每一个常规模板中，可以使用 `content_for` 来给代码块所呈现的内容一个名称。这些内容将会在 Rails 中存储起来，而不会对该模板所生成的输出起作用。

```
<h1>Regular Template</h1>

<% content_for(:sidebar) do %>
  <ul>
    <li>this text will be rendered</li>
    <li>and saved for later</li>
    <li>it may contain <%= "dynamic" %> stuff</li>
  </ul>
<% end %>

<p>
  Here's the regular stuff that will appear on
  the page rendered by this template.
</p>
```

然后，在布局中，可以使用 `yield :sidebar` 来把该代码块包括到页面侧边栏中：

```
<!DOCTYPE .... >
<html>
  <body>
    <div class="sidebar">
      <p>
```

```

Regular sidebar stuff
</p>
<div class="page-specific-sidebar">
  <%= yield :sidebar %>
</div>
</div>
</body>
</html>

```

使用同样的技术，可以把与页面相关的 JavaScript 函数添加到布局 <head> 代码部分，创建特殊菜单之类的功能。

21.6.4 局部页面模板

在一般情况下，Web 应用程序显示相同应用程序对象或不同页面的多个对象的信息。购物车应该在购物车页面上和订单汇总页面上显示订单商品项目。博客应用程序应该在索引主页面上和寻求评论的页面上方显示文章内容。典型的情况是，这会引起不同模板页面之间进行代码片段的复制。

然而，Rails 使用局部页面模板（partial-page template，通常称为局部模板（partial））避免这一重复。可以把局部模板看做子程序，在其他模板中一次或多次地调用它，同时有可能将以参数的形式呈现传递到对象。当局部模板完成呈现后，它返回调用模板的控制。

从内部来看，局部模板看起来像任意其他模板一样。从外部来看，存在着稍微的不同。包含模板代码的文件名必须以下划线字符开始，同时显示出局部模板源代码与同类型完整模板之间的区别。

例如，在默认视图路径 app/views/blog 的文件 article.html.erb 中存放呈现博客文章的局部模板：

```

<div class="article">
  <div class="articleheader">
    <h3><%= article.title %></h3>
  </div>
  <div class="articlebody">
    <%= article.body %>
  </div>
</div>

```

其他模板使用代码 `render(:partial=>)` 来调用它：

```

<%= render(:partial => "article", :object => @an_article) %>
<h3>Add Comment</h3>

```

方法 `render` 的参数 `:partial` 就是呈现模板的名称（但没有开头下划线字符）。该名称必须是有效文件名和有效 Ruby 标识符（如 `a-b` 和 `20042501` 都不是局部模板的有效名称）。参数 `:object` 等同于将对象传递给该局部模板。通过与该模板采用相同名称的本地变量，这一对象在该模板内是可以使用的。在本例中，将对象 `@an_article` 传递到该模板中，且使用本地变量 `article`，该模板可以访问它。这就是为什么在局部模板中可以编写类似 `article.title` 一样的代码。

熟悉 Rails 的开发者使用模板名称作为变量名（在这个实例中是 `article`）。事实上，通常可以更深入一步，如果要传递给局部模板的对象在控制器实例变量中，且该实例变量与局部模板同名，则可以忽略参数 `:object`。例如在前面的例子中，控制器已经为该文章安排了实例变量 `@article`，视图可以如下所示呈现该局部模板：

```
<%= render(:partial => "article") %>
<h3>Add Comment</h3>
```

通过方法 `render` 传递参数 `:locals`，可以在模板中设置更多的局部变量。这里采用了散列，其中每一项代表要设置本地变量的名称及其值。

```
render(:partial => 'article',
      :object => @an_article,
      :locals => { :authorized_by => session[:user_name],
                  :from_ip      => request.remote_ip })
```

21.6.5 局部模板和集合

通常情况下，应用程序需要显示格式化项的集合。博客网站应该显示一系列文章，而每一篇文章都有文本、作者和日期等。在线商店应该按目录显示商品，而每一种商品都有图片、描述和价格等。

方法 `render` 的参数 `:collection` 是与参数 `:partial` 协同工作的。参数 `:partial` 可以使用局部模板来定义单独项的格式，而参数 `:collection` 将集合每一项应用于该模板。为了显示一组文章模型对象，可以使用之前定义的模板 `_article.html.erb`，可以这么写：

```
<%= render(:partial => "article", :collection => @article_list) %>
```

在这个局部模板中，将本地变量 `article` 赋值为集合的当前文章——该变量是用模板名称来命名的。此外，将变量 `article_counter` 设置为该集合当前文章的索引值。

可选参数 `:spacer_template` 可以指定该集合每一项之间所要呈现的模板。例如视图应该包含以下内容：

```
e1/views/app/views/partial/list.html.erb
```

```
<%= render(:partial      => "animal",
          :collection    => %w{ ant bee cat dog elk },
          :spacer_template => "spacer")
%>
```

这里使用 `_animal.html.erb` 来呈现给定列表中的每一个动物，同时彼此之间呈现局部模板 `_spacer.html.erb`。如果模板 `_animal.html.erb` 包含如下内容：

```
e1/views/app/views/partial/_animal.html.erb
```

```
<p>The animal is <%= animal %></p>
```

且模板 `_spacer.html.erb` 包含：


```
e1/views/app/views/partial/_spacer.html.erb
```

```
<hr />
```

则用户将看到动物名称的列表，它们之间是用线条分开的。

21.6.6 共享模板

如果调用方法 `render` 的首选选项或参数 `:partial` 是简单名称，则 Rails 认为目标模板就在当前控制器的视图目录下。但是，如果名称包含一个或多个字符“/”，则 Rails 认为直到最后一个斜线部分都是目录名，剩余的是模板名称。并假设该目录是在路径 `app/views` 下。这样可以很容易地在控制器之间共享局部模板及其子模板。

Rails 应用程序的约定是将共享局部模板存放在路径 `app/views` 下的目录 `shared` 中。使用这样的语句来呈现共享局部模板：

```
<%= render("shared/header", :title => @article.title) %>
<%= render(:partial => "shared/post", :object => @article) %>
...
```

在前面的例子中，将对象 `@article` 分配到该模板的局部变量 `post` 中。

21.6.7 局部模板与布局

可以与布局一起呈现局部模板，且可以将布局应用于任何模板内的代码块。

```
<%= render :partial => "user", :layout => "administrator" %>

<%= render :layout => "administrator" do %>
  # ...
<% end %>
```

可以在与控制器相关的路径 `app/views` 下直接找到局部模板布局，其名称通常使用下划线作为前缀，如：`app/views/users/_administrator.html.erb`。

21.6.8 局部模板和控制器

不仅视图模板使用局部模板，而且控制器也参与其中。局部模板让控制器能够生成页面的片段代码，像视图本身一样使用相同的局部模板。特别重要的是，当使用 Ajax 支持来更新在控制器中的部分页面时——使用局部模板，就可以清楚地知道，更新表格的商品项目和最初用来生成同类内容的商品项目，其格式是相互兼容的。对于正在更新的表行记录或商品项目进行格式化，将与初始化时曾经生成的是格式兼容的。

综上所述，局部模板和布局提供了一种有效的方式，以确保应用程序的用户界面部分是可维护的。但可维护只是软件工作的一部分；这么操作的同时，保证运行效率也是至关重要的。

21.7 本章小结

视图是 Rails 应用程序的对外窗口，我们也已经了解到：Rails 提供了广泛的支持，其中包括

用来建立健全和可维护的用户界面与应用程序界面。

Rails 支持三种默认模板类型：erb、builder 和 rjs。模板很容易地对于任何请求提供 HTML、XML 和 JavaScript 响应。在 26.3 节中将讨论添加另一个选项。

接着深入讨论了表单，这是用户与应用程序交互的主要手段。还顺便介绍了上传文件。

然后继续讨论了帮助程序，它将应用程序复杂的逻辑分解的重点放到表达外观的视图上。我们考察了 Rails 所提供的大量帮助程序——从简单的格式化定义到超链接，这些都是用户与 HTML 页面互动的决定性方式。

最后通过把大量内容分解为可重用的两种相关方式，完成了 Action View 之旅。使用布局来分离视图的最外层，并提供一致的外观和感觉。使用局部模板来分离通用的内部组件，例如单一表单或表格。

所有这些包括了用户怎么使用浏览器来访问 Rails 应用程序。下一步：涉及如何使用缓存技术，当数据库的基础数据在请求之间没有改变时，以减少甚至杜绝呈现视图的计算开销。

第 22 章 缓存

在本章中，我们将学习：

- 静态缓存完整界面
- 缓存行为结果
- 让缓存页面失效
- 缓存部分页面

许多应用程序似乎花了很多的时间来执行相同的任务。博客应用程序会为每一个访问者呈现当前的文章列表，而商店应用程序则会把同样的产品信息页面呈现给每一个访问它的人。

所有这些重复既消耗服务器的资源又消耗其时间。呈现网站的主页可能需要好几个数据库查询，最终可能会调用若干 Ruby 方法和 Rails 模板。对于每一个单独的访问来说，这并不是什么大问题，但是若给它加个系数，乘上每小时几千次的单击率，你就会突然意识到，这将导致服务器负载变高，用户对此感觉到的则是响应速度变慢。

遇到这样的情况，使用缓存可以在很大程度上减轻服务器的负载并提高应用程序的响应性能。相比之下，我们不用再一遍又一遍地重新生成旧的内容，而是生成一次并记录其结果，当下一次遇到相同页面的请求时，直接从缓存里提取并提供相应的内容，而不是再次创建它。Rails 提供三种缓存途径：页面缓存（page caching）、行为缓存（action caching）以及片段缓存（fragment caching）。

22.1 页面缓存

页面缓存（page caching）是 Rails 各种缓存形式中最简单且最有效的一种。用户第一次请求某个特定 URL 时，系统会调用应用程序并生成一个 HTML 页面，该页面的内容存储在缓存里，下一次接收到包含同样 URL 的请求时，缓存会直接提供该页面的 HTML，该请求将根本不会到达应用程序。事实上，Rails 根本不会参与，网站服务器会自己处理整个请求过程，这样就使得页面缓存非常高效。应用程序提供这些页面的速度同服务器提供其他任何静态内容的速度一样。

然而，有时应用程序仍然需要至少一定程度上参与这些请求的处理过程中。例如，在线商店可能将某些特定产品的详细信息只提供给一部分用户（譬如付费客户能提前接触新的产品）。在这样的情况下，显示的页面将有同样的内容，但其可见性并不是对所有人都一样——我们需要过滤对缓存内容的访问。Rails 为此提供了行为缓存（action caching）。使用行为缓存，我们将仍然需要调用应用程序，并且将会在过滤器运行之前调用应用程序，然而，如果已经有缓存页面存在，则并不会调用行为本身。

下面来看看一个包含公开内容和会员专有付费内容的网站，它拥有两个控制器：一个是管理

员控制器，用来验证某用户是否为会员；另一个是内容控制器，包含若干行为，用来显示公开内容以及付费内容。公开内容由单一页面构成，比如包含指向付费文章的链接。如果某人请求付费内容，但他不是会员，那么会转向管理员控制器的一个行为，让他进行注册。

假如暂时不考虑缓存，我们可以这样来实现该应用程序的内容，使用前置过滤器来验证用户状态，并使用若干行为方法来处理前面提到的两类内容。

```
el/cookies/app/controllers/content_controller.rb
```

```
class ContentController < ApplicationController
  before_filter :verify_premium_user, :except => :public_content

  def public_content
    @articles = Article.list_public
  end

  def premium_content
    @articles = Article.list_premium
  end

  private

  def verify_premium_user
    user = session[:user_id]
    user = User.find(user) if user
    unless user && user.active?
      redirect_to :controller => "login", :action => "signup_new"
    end
  end
end
```

因为内容页面都是固定的，所以可以缓存它们。可以将公开内容作为页面层面的缓存，但是，对于缓存的付费内容，还必须使其访问权限仅限会员，后者需要使用行为层面的缓存。要启用缓存，只需给类添加两个声明。

```
el/cookies/app/controllers/content_controller.rb
```

```
class ContentController < ApplicationController
  before_filter :verify_premium_user, :except => :public_content

  caches_page :public_content
  caches_action :premium_content
```

`caches_page` 指令让 Rails 缓存 `public_content` 第一次生成的输出内容。此后，这一页将直接由网站服务器提供。

第二条指令 `caches_action` 告诉 Rails 去缓存 `premium_content` 的执行结果，且仍然执行过滤器。这就是说，虽然我们还是会验证访问该页面的用户是否有权限，但实际上不会多次执行该行为。

行为缓存是前后置过滤器（见 20.3 节）的一个很好的例子。过滤器的前置部分检查缓存项目是否存在。如果存在，将它呈现给用户，以避免执行实际行为。过滤器的后置部分用来将行为运行的结果保存在缓存中。

`cache_action` 可以接收若干配置选项。`:cache_path` 选项用来修改行为缓存的路径。对于有些行为, 它们需要应对不同条件下不同的缓存需求, 这时 `:cache_path` 会很有帮助。`:if` 和 `:unless` 用来传递一个 Proc, 以控制一个行为应该什么时候通过。最后是 `:layout` 选项, 如果是 `false`, 那么 Rails 只会缓存行为的内容。当布局包含动态信息时 `:layout` 将很有用。

默认情况下, 缓存只在产品环境中启用。可以通过如下设定手动开启或关闭它:

```
ActionController::Base.perform_caching = true | false
```

可以在应用程序环境文件 (`config/environments` 下) 中改变, 虽然此时首选语法会略有不同:

```
config.action_controller.perform_caching = true
```

注意, Rails 中不论是行为缓存还是页面缓存都严格依据 URL。某一页面会根据第一次生产它的 URL 缓存起来, 随后我们对于指向相同 URL 的请求将返回已保存的内容。

这就意味着依赖 URL 以外的信息的动态页面将不适合缓存。包括下面这些:

- 页面的内容基于时间 (见 22.2 节)。
- 页面的内容依赖于会话信息。例如, 如果对于每一个用户都采用个性化的页面, 将不太可能缓存它们 (不过或许可以利用片段缓存, 见 22.3 节)。
- 页面生成于不受控的数据。例如, 某个页面显示的信息来自数据库, 它估计是不可缓存的, 如果某个非 Rails 的应用程序也能更新该数据库。页面可能会在应用程序不知情的情況下变为过时。

然而, 缓存还是“可以”应付由受控的不稳定内容生成的页面。这将在下一节中讲到, 它就是一个简单的关于移除过期缓存页面的问题。

22.2 让页面失效

创建缓存页面只是等式的一半。如果最初生成该页面的内容改变了, 已缓存的版本将变为过期的, 需要某种方法来让它们失效。

诀窍在于给应用程序添加代码, 让它注意到用来生成动态页面的数据改变了, 然后移除已缓存的版本。对于该 URL 的下一次请求, 将基于新的内容再次生成缓存的页面。

22.2.1 显式地让页面失效

移除缓存的底层方法是使用 `expire_page` 和 `expire_action`。它们将与 `url_for` 使用一样的参数, 并会使所生成 URL 相对应的页面失效。

例如, 内容控制器也许包含某个行为, 用来创建文章, 还包含另外某个行为, 用来更新已有的文章。当创建文章时, 公开页面的文章列表将变成过期的内容。于是调用 `expire_page`, 并传递显示公开页面的行为名。当更新已有文章时, 公开索引页面将保存不变 (至少在该应用程序中), 但是该文章的任何一个缓存版本都应当删除。因为该缓存是使用 `cache_action` 创建的, 所以需要使用 `expire_action` 来让该页面失效, 并传递其行为名和文章 id。

```

e1/cookies/app/controllers/content_controller.rb

def create_article
  article = Article.new(params[:article])
  if article.save
    expire_page :action => "public_content"
  else
    # ...
  end
end

def update_article
  article = Article.find(params[:id])
  if article.update_attributes(params[:article])
    expire_action :action => "premium_content", :id => article
  else
    # ...
  end
end

```

文章删除方法做了些附加的工作——它必须废止公开索引页面并移除特定的文章页面：

```

e1/cookies/app/controllers/content_controller.rb

def delete_article
  Article.destroy(params[:id])
  expire_page :action => "public_content"
  expire_action :action => "premium_content", :id => params[:id]
end

```

22.2.2 挑选缓存存储策略

像会话一样，我们也需要给缓存配置若干存储选项。片段缓存可以保留在文件、数据库、DRb 服务器或者 memcached 服务器中。但是会话通常包含少量的数据，对于每个用户只需要提供一个存储行，片段缓存可以很容易地创建大量的数据，并且每个用户都可以拥有多个。这让数据库难以成为合适的选择。

对于大量的配置，把缓存文件保存在文件系统里最简单。但是，不能把这些缓存文件保存在每一个服务器本地上，这是因为让某一个服务器上的某个缓存失效时，并不能让其他的缓存也失效。因此需要建立一个网络驱动器，这样所有的服务器可以共享它们的缓存。

如同会话的配置一样，可以把基于文件的缓存存储策略作为全局配置，放在 `environment.rb` 或某个特殊的环境文件中。

```
ActionController::Base.cache_store = :file_store, "#{RAILS_ROOT}/cache"
```

该配置假定在应用程序的根目录下存在一个可访问的以目录命名的缓存，并且网站服务器拥有完全的读写权限。我们可以把该目录很容易地创建为服务器路径下的一个符号链接，并指向网络驱动器。

不管选用何种存储方式进行片段缓存，都需要注意，网络瓶颈会很容易变成一个问题。如果一个网站高度依赖于片段缓存，那么它的每一个请求都会需要大量的数据传输。这些数据会先从

网络驱动器传向指定服务器，再发送给用户。倘若要在一个吞吐量很高的网站上使用片段缓存，则着实需要给服务器之间配置高速带宽，否则响应速度会明显下降。

缓存存储系统仅可以用来缓存行为和片段。完整页面缓存需要存储在文件系统的 public 目录中。在这种情况下，如果想要在多个网络服务器间使用页面缓存，就需要使用网络驱动器路由。你可以对整个 public 目录做符号链接（但是这样将导致图片、样式表以及 JavaScript 在网络间互相传递，有可能会引起问题），也可以只对页面缓存所需要的各个单一目录做符号链接。例如，在后一种情况中，可以将 public/products 符号链接到网络驱动器，用来存放产品控制器的页面缓存。

22.2.3 隐式地让页面失效

expire_xxx 方法工作得很好，但是它们将缓存函数同控制器的代码耦合在了一起。每次数据库修改了什么，都必须找出哪些缓存的页面可能会受到影响。虽然，对于较小的应用程序来说这很容易，但是随着应用程序的增长，这将变得越来越困难。某个控制器里的修改可能影响另一个缓存的页面。帮助方法中的业务逻辑本来不用负责 HTML 页面，现在却需要去关心缓存页面的失效。

庆幸的是，Rails 的 sweeper（清扫器）可以将一些这样的耦合简化。清扫器是一类特殊的模型对象观察器。当模型中发生重要改变时，清扫器会让依赖于该模型的缓存页面失效。

应用程序中的清扫器需要多少就可以设置多少。通常会创建一个单独的清扫器去管理每一个与控制器相关的缓存。把清扫器代码放在 app/sweepers 中：

```
e1/cookies/app/sweepers/article_sweeper.rb
```

```
class ArticleSweeper < ActionController::Caching::Sweeper
```

```
  observe Article
```

```
  # If we create a new article, the public list of articles must be regenerated
```

```
  def after_create(article)
```

```
    expire_public_page
```

```
  end
```

```
  # If we update an existing article, the cached version of that article is stale
```

```
  def after_update(article)
```

```
    expire_article_page(article.id)
```

```
  end
```

```
  # Deleting a page means we update the public list and blow away the cached article
```

```
  def after_destroy(article)
```

```
    expire_public_page
```

```
    expire_article_page(article.id)
```

```
  end
```

```
  private
```

```
  def expire_public_page
```

```
    expire_page(:controller => "content", :action => 'public_content')
```

```
  end
```

```
def expire_article_page(article_id)
  expire_action(:controller => "content",
               :action    => "premium_content",
               :id        => article_id)
end
```

清扫器的流程多少有些让人费解:

- 首先在一个或多个 Active Record 类中把清扫器声明为一个观察器。在上面的例子中，它在观察 Article 模型（第一次讨论观察器要回到 19.4.2 节）。清扫器使用钩子方法（例如 after_updated）在适当的时候让缓存页面失效。
- 清扫器同时也在控制器中用指令声明 cache_sweeper 为活跃的:

```
class ContentController < ApplicationController
  before_filter :verify_premium_user, :except => :public_content
  caches_page  :public_content
  caches_action :premium_content

  cache_sweeper :article_sweeper,
               :only => [ :create_article,
                          :update_article,
                          :delete_article ]

  # ...
end
```

- 如果某个请求进来并牵连某个清扫器过滤着的行为，则会激活清扫器。任何一个 Active Record 观察器方法的触发，都会发生对页面和行为失效方法的调用。如果调用了 Active Record 观察器，但是并没有选择把当前行为作为某个缓存清扫器，则会忽略清扫器中对失效的调用。反之则执行该失效操作。

22.2.4 让基于时间的缓存页面失效

考虑某个网站：显示有相对易变的信息，例如股票报价或头条新闻，如果缓存形式还是像前面一样——当页面内容改变时便将其失效；则我们将一直不断地让页面失效，这样基本不会用到缓存，进而损失了它带来的好处。

在这种情况下，也许要考虑使用基于时间的缓存——创建缓存同前面完全一样，然而在内容过期时并不将其终止。

此时将运行一个后台进程，定期地进入缓存目录并删除缓存文件。可以选择如何进行这些删除操作——可以简单地删除所有文件，或者删除超过 x 分钟之前创建的文件，又或者删除名称匹配某一模式的文件。这部分由应用程序决定。

下一次请求这些页面中的某一个时，它将不由缓存支持，应用程序将处理它。该进程将自动重新填充缓存中的某个特定页面，减轻对该页面随后的获取。

在哪里寻找要删除的文件？毫无疑问，这是可配置的。页面缓存文件默认存储在应用程序的 public 目录下。缓存时它们会以 URL 命名，并以 .html 结尾。例如 content/show/1 的页面将保存在这里：

```
app/public/content/show/1.html
```

这种命名模式不是巧合的；它让网站服务器可以自动找到缓存文件，然而也可以这样来覆盖此默认设定：

```
config.action_controller.page_cache_directory = "dir/name"
config.action_controller.page_cache_extension = ".html"
```

默认情况下，行为缓存文件并不存储在常规文件系统的目录架构中，并且不能使用该技巧来让其失效。

22.2.5 正确处理客户端缓存

我们之前说过创建缓存页面只是等式的一半，却忘记提到这里有一半。

客户端（通常指浏览器，但并不总是）通常也有缓存。服务器和客户端之间的中间层也会提供缓存服务。可以提供 HTTP 头来优化这些缓存（并因此减轻服务器上的载荷）。这么做完全是可选的，而且并不总是会减少带宽，但是另一方面，有时这会节省很多。

22.2.6 过期头

最高效的请求就是不请求。许多页面（特别是图像、脚本和样式）很少改变，但是会多次引用到。对于这些页面，可以提供 Expires 头来保证：在进入服务器之前就已经优化了重复检索。

虽然一个 Expires 头可以提供若干不同的选项，但最常用的选项是指定适合的响应时间，并据此建议客户端在此期间不必重新检索该数据。在控制器中调用 expires_in 方法可以达到这个目的：

```
expires_in 20.minutes
expires_in 1.year
```

服务器不应该在 Expires 头中提供超过 1 年的数据。

对于 Expires 头，另一个选择是指明完全不缓存响应。可以用 expires_now 完成，它不需要任何参数，这是可以理解的。

在 HTTP 规范中可以找到更多关于失效选项的内容[⊖]。

注意，如果使用页面层面的缓存，请求将缓存在服务器上，而不会进入应用程序，因而该机制也需要在服务器层面实现才能生效。这里是 Apache 网站服务器的一个例子：

```
ExpiresActive On
<FilesMatch "\.(ico|gif|jpe?g|png|js|css)$">
  ExpiresDefault "access plus 1 year"
</FilesMatch>
```

22.2.7 最后的修改和 ETag 支持

下一个完全消除请求的方法是即刻回馈一个 HTTP 304 Not Modified 响应。这将引导用户端

⊖ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.3>

和中间层去使用它们已有的缓存。至少，该响应可以减少带宽。通常这可以消除提供一个更完整响应给服务器带来的负载。

如果已经使用了 Apache 作为页面层面的缓存，那么让网站服务器自己处理会好一些，并且将基于与硬盘缓存关联的时间戳。

对于所有其他的缓存，如果使用 `stale?` 或 `fresh_when` 方法中的一个，Rails 会提供所需的 HTTP 头。

这两种方法都接收 `:last_modified` 时间戳（UTC 格式）和 `:etag`。后者要么是一个响应所依赖的对象，要么是该类对象的一个列表。这样的对象需要响应 `cache_key` 或 `to_param` 中的一个。ActiveRecord 将处理好这个。

`stale?` 的典型用法是在使用 `if` 语句涉及自定义呈现时：

```
def show
  @article = Article.find(params[:id])

  if stale?([:etag=>@article, :last_modified=>@article.created_at.utc]
    # ...
  end
end
```

当使用默认呈现时，`fresh_when` 更常用，这是因为它会产生一个 304 Not Modified 响应。

```
def show
  fresh_when(:etag=>@article, :last_modified=>@article.created_at.utc)
end
```

此外，除了可以缓存整页，Rails 还对缓存“部分”页面提供支持。接下来将介绍为什么需要该功能，以及怎样将其集成到应用程序中。

22.3 片段缓存

对于动态网站，我们会发现缓存部分页面特别有用。博客应用程序可能会为每一个单独的用户自定义欢迎内容以及边目录。这时，不能使用页面缓存，因为整个页面将因人而异。但是由于文章列表并没有因用户不同而改变，所以可以使用片段缓存——生成这样的 HTML，它只显示一次文章，并将这些文章包含到根据单独用户分别定制的页面中。

只是为了演示片段缓存，下面建立一个想象的博客应用程序。下面是控制器，它设定 `@dynamic_content`，代表那些每次访问页面时都应该改变的内容。对于这个假想的博客，使用当前时间作为此内容。

```
e1/views/app/controllers/blog_controller.rb

class BlogController < ApplicationController
  def list
    @dynamic_content = Time.now.to_s
  end
end
```

接下来定义一个模拟 `Article` 类，它模拟一个模型类，正常情况下它将从数据库获取文章。文章列表从第一个开始显示它创建的时间。

```
e1/views/app/models/article.rb

class Article
  attr_reader :body

  def initialize(body)
    @body = body
  end

  def self.find_recent
    [ new("It is now #{Time.now.to_s}"),
      new("Today I had pizza"),
      new("Yesterday I watched Spongebob"),
      new("Did nothing on Saturday") ]
  end
end
```

我们现在建立一个模板，以使用已呈现的文章的缓存版本，并仍然更新动态数据，你将会发现它原来很简单。

```
e1/views/app/views/blog/list.html.erb

<%= @dynamic_content %> <!-- Here's dynamic content. -->

<% cache do %> <!-- Here's the content we cache -->
  <ul>
    <% for article in Article.find_recent %>
      <li><p><%= h(article.body) %></p></li>
    <% end %>
  </ul>
<% end %> <!-- End of cached content -->

<%= @dynamic_content %> <!-- More dynamic content. -->
```

在这里，巧妙之处在于 `cache` 方法，与该方法关联块所生成的所有输出都将会缓存起来。下一次访问该页面的时候，将仍然会呈现动态内容，但是前面提到的块中的内容将直接从缓存获得——不会再次生成它们。如果运行上面的骨架版应用程序，等待几秒钟，然后单击更新，便会看到前文说到的效果，如图 22.1 所示。页面顶部和底部的时间（数据中的动态部分）在更新后改变了，但是中间部分的时间保持不变，这是因为它来自缓存。（如果你自己尝试，并看见所有三处时间字符串全部变了，很大可能是因为应用程序运行在开发模式下。默认设置下，缓存只在产品模式下生效。如果试验时用 `WEBrick`，`-e production` 选项将达到此目的。）

这里的关键概念是：缓存的内容是视图中生成的片段。如果已经在控制器中生成了文章列表，然后将该列表传递给视图，则此后对该页面的访问将不必再次呈现该列表，但是每次请求时将仍然访问数据库。把数据库请求移到视图中意味着：一旦输出已经缓存起来，便不会再调用此请求去访问数据库。

有人也许会说，但是这样将打破之前关于在视图模板中放置应用程序层面代码的规定。那么，能否在某种程度上避免这一点？可以，但是这意味着把缓存变得没有原先那么透明了。技巧是在行为中检测缓存片段是否存在。如果存在，则让行为知道我们将要使用片段，因而可以绕过高消耗的数据库操作。

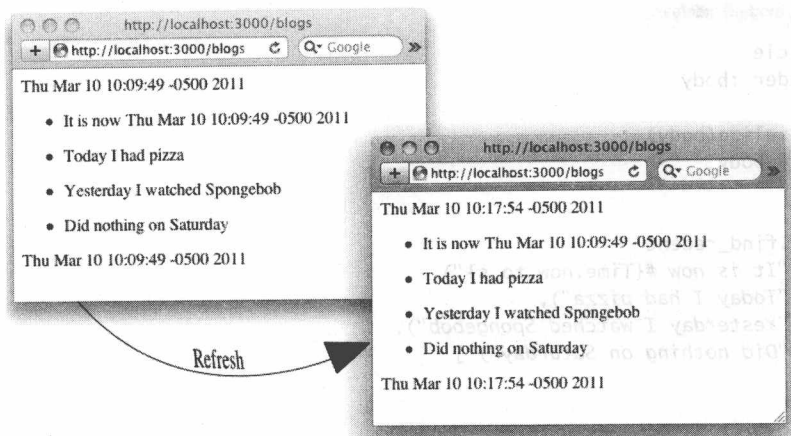


图 22.1 刷新页面后，缓存和非缓存数据

```
e1/views/app/controllers/blog1_controller.rb
```

```
class Blog1Controller < ApplicationController
```

```
  def list
```

```
    @dynamic_content = Time.now.to_s
```

```
    unless read_fragment(:action => 'list')
```

```
      logger.info("Creating fragment")
```

```
      @articles = Article.find_recent
```

```
    end
```

```
  end
```

```
end
```

行为使用 `read_fragment` 方法来查看某个片段是否存在。如果不存在，则从（伪造的）数据库中加载文章列表。

```
e1/views/app/views/blog1/list.html.erb
```

```
<%= @dynamic_content %> <!-- Here's dynamic content. -->
```

```
<% cache do %> <!-- Here's the content we cache -->
```

```
  <ul>
```

```
    <% for article in @articles %>
```

```
      <li><p><%= h(article.body) %></p></li>
```

```
    <% end %>
```

```
  </ul>
```

```
<% end %> <!-- End of the cached content -->
```

```
<%= @dynamic_content %> <!-- More dynamic content. -->
```

让缓存片段失效

现在我们已经拥有了缓存版的文章列表，不论何时引用该页面，该 Rails 应用程序都可以提供它。但是，如果文章更新了，缓存的版本将变成过期的，需要让它失效。`expire_fragment`

方法可以实现这一功能。默认情况下，片段缓存时使用控制器名称，并且呈现该页面的行为名称（如第一个例子中的 `blog` 和 `list`）命名。控制器可以使用下面的语句来让片段失效（例如文章列表改变了）：

```
e1/views/app/controllers/blog_controller.rb
expire_fragment(:controller => 'blog', :action => 'list')
```

显然该命名格式只适用于一个页面只有一个片段的情况。庆幸的是，如果需要更多，可以通过给 `cache` 方法添加参数（常用 `url_for`）来覆盖与片段相关联的名称。

```
e1/views/app/views/blog2/list.html.erb
<% cache(:action => 'list', :part => 'articles') do %>
  <ul>
    <% for article in @articles %>
      <li><p><%= h(article.body) %></p></li>
    <% end %>
  </ul>
<% end %>

<% cache(:action => 'list', :part => 'counts') do %>
  <p>
    There are a total of <%= @article_count %> articles.
  </p>
<% end %>
```

在本例中缓存了两个片段，第一个缓存使用了一个额外的 `:part` 参数，并定义为 `articles`，第二个缓存也有该参数，但定义为 `counts`。

在控制器中，可以给 `expire_fragment` 传递相同的参数来删除特定的片段。例如，当修改某篇文章时，必须让文章列表失效，但是总数仍然有效。而当删除一篇文章时，需要让这两个片段都失效。控制器看起来将像下面这样（并没有任何代码真正地对其中的文章做些什么——仅仅是关注了缓存）：

```
e1/views/app/controllers/blog2_controller.rb
class Blog2Controller < ApplicationController

  def list
    @dynamic_content = Time.now.to_s
    @articles = Article.find_recent
    @article_count = @articles.size
  end

  def edit
    # do the article editing
    expire_fragment(:action => 'list', :part => 'articles')
    redirect_to(:action => 'list')
  end

  def delete
    # do the deleting
```

```

expire_fragment(:action => 'list', :part => 'articles')
expire_fragment(:action => 'list', :part => 'counts')
redirect_to(:action => 'list')
end
end

```

`expire_fragment` 方法还可以使用单个正则表达式作为参数，用来让所有与名称匹配的片段失效：

```
expire_fragment(%r{/blog2/list.*})
```

22.4 本章小结

本章探索了三种技术，用来让网站的响应速度更快。

在本章中学习了怎样缓存整个页面，从而在处理由缓存供给的请求时，避免任何 Ruby、Rails 或数据库的开销。对于一些高流量、需要访问数据库来产生但不怎么改变的页面，此方法很有用。Depot 应用程序的商品目录列表就是这类页面的最好例子。

本章中还学习了怎样缓存控制器行为的结果，避免呈现和数据库的开销。当需要继续使用过滤器（通常用来认证）时，该方法很有用。整个过程涵盖了强行让缓存页面失效的显式及隐式机制。

本章最后学习了如何缓存页面片段，这让我们可以完全掌控负载平衡的优化及动态内容的生成。

所有这些都围绕着如何维持缓存，用以对视图提供支持。下一步：维持数据库模式，用来为模型提供支持。

第 23 章

数据迁移

在本章中，我们将学习：

- 命名迁移文件
- 重命名和字段
- 创建和重命名表
- 定义索引和键
- 使用原生 SQL

Rails 鼓励灵活、迭代的开发方式。我们并不指望第一次就能把每件事都做好。与之相反，在开发过程中编写测试，并且与客户交互，以此来增进理解。

要完成这项工作，需要一系列的支撑练习。可以编写测试来帮忙设计接口，当做出修改时，这些测试就像一张安全网。还可以使用版本控制来保存应用程序的源文件，以便撤销错误的修改，并且监控每天的变化。

但是，应用程序中仍然有一个地带，其变化是无法用版本控制直接管理的。在整个开发过程中，Rails 应用程序的数据库模式在不断演变：这里添加一个表，那里重命名一个字段等。数据库与应用程序的代码一起改变。

在 Rails 中，可以通过使用迁移来监控每个步骤。开发 Depot 应用程序的过程中就使用了迁移，从 6.1 节创建第一个 `products` 表开始，到 10.1 节在 `line_items` 表里添加一个数量字段，都涉及迁移的使用。现在是时候学习有关迁移更深层的工作原理，看看它们还能做什么。

23.1 创建和运行迁移

迁移只是一个 Ruby 源文件，这个文件在该应用程序的 `db/migrate` 目录下。每个迁移文件的名称都以大量数字开头（通常情况下是 14 位），随后是一个下划线。这些数字是迁移的密钥，因为它们定义了迁移应用的顺序——它们是单独的迁移版本号。

这个版本号是迁移创建时以世界标准时间（Coordinated Universal Time, UTC）定义的时间戳。这些数字包含四位数的年份，之后各是两位数的月份、日期、小时、分钟和秒，所有这些都基于格林尼治标准时间（the mean solar time at the Royal Observatory in Greenwich, GMT）。由于迁移创建的频率相对较低，并且记录精确到秒，因此任意两个人得到相同时间戳的几率微乎其微。使用时间戳的好处在于可以确定地排列这些文件，这个优点的收益远远超过了所承担的那点风险。

下面是 Depot 应用程序 `db/migrate` 目录的例子：

```
depot> ls db/migrate
20110211000001_create_products.rb
20110211000002_create_carts.rb
20110211000003_create_line_items.rb
20110211000004_add_quantity_to_line_items.rb
20110211000005_combine_items_in_cart.rb
20110211000006_create_orders.rb
20110211000007_add_order_id_to_line_item.rb
20110211000008_create_users.rb
```

虽然可以手动创建这些迁移文件，但使用生成器会更为轻松（且不容易出错）。就像创建 Depot 应用程序那样，实际中有两个生成器用来创建迁移文件：

- **model** 生成器将创建一个迁移，用于依次创建与这个模型相关的表（除非执行中定义了 `--skip-migration` 选项，否则将自动创建这个迁移）。正如下面这个例子所示，在创建 **discount** 模型的同时，也创建了一个称为 `yyyyMMddhhmmss_create_discounts.rb` 的迁移文件：

```
depot> rails generate model discount
```

```
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/discount.rb
create test/unit/discount_test.rb
create test/fixtures/discounts.yml
exists db/migrate
▶ create db/migrate/20110211000010_create_discounts.rb
```

- 也可以自己生成一个迁移：

```
depot> rails generate migration add_price_column
```

```
exists db/migrate
▶ create db/migrate/20110211000011_add_price_column.rb
```

稍后，在 23.2 节中可以看到迁移文件写入的内容。但现在，先跳过流程中的这一步，来看看如何运行迁移。

运行迁移

用 Rake 任务 `db:migrate` 可以运行迁移：

```
depot> rake db:migrate
```

要想看到下一步会发生什么，需要深入 Rails 内部。

迁移代码在每个 Rails 数据库中都包含一个称为 `schema_migrations` 的表。这个表只有一个名为 `version` 的字段，并且每成功应用一个迁移，该表就会为其添加一行新内容。

运行 `rake db:migrate` 时，该任务会先寻找 `schema_migrations` 表。如果该表不存在，就先创建它。

紧接着迁移代码会查看 `db/migrate` 目录下的所有迁移文件，倘若版本号（文件名中的数字）已经在数据库中，则跳过这个文件。然后把筛选后剩余的迁移——处理，并为每个迁移在 `schema_migrations` 表中创建一条新记录。

如果在这个时候再次运行迁移,则不会发生什么。由于每个迁移文件的版本号都将和数据库中的一条记录匹配,因此不会应用迁移。

但是,如果随后创建一个新的迁移文件,那么即使这时该版本号的时间出现在一个或多个已经应用过的迁移之前,这个文件的版本号也不会出现在数据库内。在多个用户使用版本控制系统来存储迁移文件时,可能会发生这种情况。如果这时运行迁移,将只会执行这个新的迁移文件。但是这可能导致迁移运行秩序混乱,因此得小心一些,以确保这些迁移相互独立。或者可以回到上一个版本的数据库,然后再按照顺序应用迁移。

可以为 rake db:migrate 命令提供 VERSION= 参数来迫使数据库执行特殊版本:

```
depot> rake db:migrate VERSION=20110211000009
```

如果给出的版本比任何已应用过的迁移版本都高,那么将应用这些迁移。

但是,倘若命令行中的版本号比 schema_migrations 表中列出的一个或多个版本号大,那么将会有不同的事情发生。在这些情况下,Rails 会寻找那个和数据库中版本号相符的迁移文件,然后撤销它。这个步骤将持续循环,直到 schema_migrations 表中列出的版本号都不超过命令行中给定的版本号。也就是说,迁移以相反的顺序应用,让模式回到指定的版本。

当然,也可以重做一个或多个迁移:

```
depot> rake db:migrate:redo STEP=3
```

默认情况下,redo 将回滚一个迁移,并重新运行它。要回滚多个迁移,可以修改 STEP= 参数。

23.2 剖析迁移

迁移都是 Rails 中类 ActiveRecord::Migration 的子类。创建一个类时至少应该包含两个类的方法: up 和 down:

```
class SomeMeaningfulName < ActiveRecord::Migration
  def self.up
    # ...
  end

  def self.down
    # ...
  end
end
```

类名必须与版本号后的那一部分文件名匹配,类名中所有大写字母都变为小写字母,并且每个词之前以下划线连接。例如,上面这个类可以在先前的文件 20110211000017_some_meaningful_name.rb 中找到。任意两个迁移不能包含相同的类名。

up 方法负责应用这个迁移的模式改变,而 down 方法则负责撤销这些改变。下面这个例子具体说明了这两个方法的功能,在这个例子中,一个迁移在 oeders 表中添加了 e_mail 字段:

```
class AddEmailToOrders < ActiveRecord::Migration
  def self.up
    add_column :orders, :e_mail, :string
  end
end
```

```
def self.down
  remove_column :orders, :e_mail
end
```

看到 down 方法如何撤销 up 方法的改变了吗?

23.2.1 字段的类型

add_column 的第三个参数指定了这个数据库字段的类型。在先前的例子中, 指定了 e_mail 字段的类型为 :string。但是, 这是什么意思? 因为通常情况下, 数据库没有 :string 这种类型的字段。

还记得 Rails 尝试让应用程序与底层数据库相互独立的事情吗? 例如, 可以使用 SQLite 3 开发, 然后根据个人喜好部署到 Postgres 上。但是不同的数据库使用不同的名称作为字段的类型。如果在迁移中使用 SQLite 3 的字段类型, 那么这个迁移在应用到 Postgres 数据库中后可能无法运行。因此, Rails 通过使用逻辑类型来把用户和底层数据库类型系统隔离开。如果迁移到 SQLite 3 数据库, 那么 :string 类型将创建一个类型为 varchar(255) 的字段。而在 Postgres 上, 相同的迁移会添加一个类型为 charvarying(255) 的字段。

迁移支持的类型有 :binary、:boolean、:date、:datetime、:decimal、:float、:integer、:string、:text、:time 和 :timestamp。图 23.1 给出了这些类型在 Rails 数据库适配器中默认的映射关系。查看这个图, 会发现在迁移中声明为 :integer 的字段将在 SQLite 3 中变成 integer 类型, 而在 Oracle 中则为 number(38)。

在迁移中定义大多数字段时, 可以指定至多三个选项。十进制数字段可以有另外两个选项。每个选项以 key => value 的形式给出。常用的选项如下:

:null => true 或 false

如果定义为 false, 这个基础字段添加一个 not null 约束 (如果数据库支持 not null 约束)。

:limit => size

为字段设置一个大小限制。这基本上用于追加字符串 (尺寸) 到数据库的字段类型定义中。

:default => value

为该字段设置默认值。注意, 在迁移运行时, 这个默认值将进行一次计算, 因此下面这行代码将把默认字段的值设置成迁移运行时的日期和时间:

```
add_column :orders, :placed_at, :datetime, :default => Time.now
```

此外, 十进制数字段还有两个选项 :precision 和 :scale。:precision 选项用来指定要保存的有效位数, 而 :scale 选项决定在这些数字中小数点所在的位置 (可以把它看做是小数点后的有效数字)。一个精确度为 5, 比例为 0 的十进制数可以存储从 -99,999 到 +99,999 的数。一个精确度为 5, 比例为 2 的十进制数则可以存储从 -999.99 到 +999.99 的数。

:precision 和 :scale 是十进制数字段的选项, 但是, 由于不同数据库间的不兼容性, 这里强力推荐每个十进制数字段都包含这两个选项。

	db2	mysql	openbase	oracle
:binary	blob(32768)	blob	object	blob
:boolean	decimal(1)	tinyint(1)	boolean	number(1)
:date	date	date	date	date
:datetime	timestamp	datetime	datetime	date
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float	number
:integer	int	int(11)	integer	number(38)
:string	varchar(255)	varchar(255)	char(4096)	varchar2(255)
:text	clob(32768)	text	text	clob
:time	time	time	time	date
:timestamp	timestamp	datetime	timestamp	date

	postgresql	sqlite	sqlserver	sybase
:binary	bytea	blob	image	image
:boolean	boolean	boolean	bit	bit
:date	date	date	date	datetime
:datetime	timestamp	datetime	datetime	datetime
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float(8)	float(8)
:integer	integer	integer	int	int
:string	(note 1)	varchar(255)	varchar(255)	varchar(255)
:text	text	text	text	text
:time	time	datetime	time	time
:timestamp	timestamp	datetime	datetime	timestamp

Note 1: character varying(256)

图 23.1 迁移和数据库的字段类型

下面是使用迁移类型和选项定义的一些字段：

```
add_column :orders, :attn, :string, :limit => 100
add_column :orders, :order_type, :integer
add_column :orders, :ship_class, :string, :null => false, :default => 'priority'
add_column :orders, :amount, :decimal, :precision => 8, :scale => 2
```

23.2.2 重命名字段

在重构代码时，经常会改变变量名，以让它们更有意义。Rails 的迁移也允许修改数据库字段名。例如：在添加这个字段一星期后，可能会觉得 e_mail 不是这个新字段最好的名字。可以创建一个迁移，使用 `rename_column` 方法来重命名它：

```
class RenameEmailColumn < ActiveRecord::Migration
  def self.up
    rename_column :orders, :e_mail, :customer_email
  end

  def self.down
    rename_column :orders, :customer_email, :e_mail
  end
end
```

注意，重命名没有损坏任何与这个字段相关联的现存数据。另外要知道，不是所有的适配器都支持重命名。

23.2.3 修改字段

可以使用 `change_column` 方法来修改字段的类型，或者改变与字段相关的选项。与 `add_column` 使用方法相同，但是 `change_column` 方法要指定一个现存的字段名。例如，订单字段的类型现在是整数，而我们要改为字符串。因为想要保持现有数据，因此订单类型 123 将变为字符串 123。稍后将使用非整数值，如 `new` 和 `existing`。

从整数字段到字符串字段的转换很简单：

```
def self.up
  change_column :orders, :order_type, :string
end
```

但是，从字符串字段到整数字段的转换却存在问题。可能需要尝试编写显性的 `down` 迁移：

```
def self.down
  change_column :orders, :order_type, :integer
end
```

倘若应用程序已经在这个字段中存储诸如 `new` 这样的数据，那么 `down` 函数将删除它——因为 `new` 不能转换为一个数字。如果这种做法是可以接受的，那么迁移也会接受。但如果想创建一个单方向的迁移，即一个无法逆转的迁移，那么可能会停止应用 `down` 迁移。在这种情况下，Rails 提供一个特殊的异常供用户抛出：

```
class ChangeOrderTypeToString < ActiveRecord::Migration
  def self.up
    change_column :orders, :order_type, :string, :null => false
  end

  def self.down
    raise ActiveRecord::IrreversibleMigration
  end
end
```

23.3 表的管理

至此，已经使用了迁移来管理现存表中的字段。那么，现在来看一看创建和删除表：

```

class CreateOrderHistories < ActiveRecord::Migration
  def self.up
    create_table :order_histories do |t|
      t.integer :order_id, :null => false
      t.text :notes

      t.timestamps
    end
  end

  def self.down
    drop_table :order_histories
  end
end

```

`create_table` 将取得表名（记住，表名都是复数的）和块。（它也会取得一些选项参数，这些参数会马上讨论到。）这个块是传入表定义的对象，用于定义表中的字段。

各种各样表定义的方法调用看起来很熟悉——它们和先前用过的 `add_column` 方法类似，只是这些方法不把表名作为第一个参数，方法本身的名称是所需的数据类型。这样可以减少重复。

要注意，新表里没有定义 `id` 字段。除非明确指定，否则都不需要定义。因为 Rails 迁移会自动在所有创建的表格中添加称为 `id` 的主键。更进一步的讨论请看 23.3 节。

`timestamps` 方法创建了 `created_at` 和 `updated_at` 字段，两个字段的 datatype 均为正确的 `timestamp`。尽管没有必要往任何特殊的表中添加这些字段，但这可以看做 Rails 的另一个例子，在其中 Rails 很方便并一致地实施了一个共同的公约。

23.3.1 表的创建选项

可以将一个散列选项作为 `create_table` 的第二个参数传递给它。

如果定义 `:force => true`，迁移会在创建这个新表前删除相同名字的现存表。倘若想创建一个迁移，并且这个迁移会迫使数据库回到已知状态，这将是一个有用的选项，但是这显然也存在数据丢失的可能。

`:temporary => true` 选项将创建一个临时表——这个表将在应用程序与数据库断开时删除。显然，这在迁移中毫无意义，但稍后可以看出，它在某些其他方面还是有用的。

`:options => "xxxx"` 参数用于为底层数据库指定选项。这些选项添加在 `CREATE TABLE` 语句的最后，也就是紧接在该语句结束的右括号后。尽管这在 SQLite 3 中没有必要，但有时这个选项对其他数据库还是有用的。例如，MySQL 的某些版本允许为自动递增的 `id` 字段指定初始值，可以像这样通过迁移传入下列内容：

```

create_table :tickets, :options => "auto_increment = 10000" do |t|
  t.text :description
  t.timestamps
end

```

在这种情况下，为 MySQL 配置时，从这个表的描述中，迁移会生成下面的 DDL：

```
CREATE TABLE "tickets" (
  "id" int(11) default null auto_increment primary key,
  "description" text,
  "created_at" datetime,
  "updated_at" datetime
) auto_increment = 10000;
```

在 MySQL 中要小心使用 `:options` 参数。Rails MySQL 数据库适配器设置的默认选项为 `ENGINE=InnoDB`。这将重写所有本地存在的默认值，并且迫使迁移使用新表的 InnoDB 存储引擎。但是，如果重写 `:options`，这些设置都将不复存在。创建新表时将使用该电脑上配置的默认数据库引擎。在这种情况下，可能需要显性地添加 `ENGINE=InnoDB` 选项来强制定义标准行为。如果使用 MySQL，也许还想继续使用 InnoDB，因为这个引擎支持事务。也许在应用程序中也需要事务支持。另外，如果使用默认的事务测试固定装置，在测试时肯定需要事务。

23.3.2 表的重命名

如果重构允许重命名变量和字段，那么当偶然发现也可以重命名表时，也就不会那么惊讶了。迁移支持 `rename_table` 方法：

```
class RenameOrderHistories < ActiveRecord::Migration
  def self.up
    rename_table :order_histories, :order_notes
  end

  def self.down
    rename_table :order_notes, :order_histories
  end
end
```

要注意，通过将其重命名为原表名，`down` 方法是如何撤销改变的。

23.3.3 rename_table 方法的问题

在迁移中重命名表时有一个微小的问题。

例如，假设迁移 4 创建了 `order_histories` 表，并且用一些数据来填充它。

```
def self.up
  create_table :order_histories do |t|
    t.integer :order_id, :null => false
    t.text :notes

    t.timestamps
  end

  order = Order.find :first
  OrderHistory.create(:order_id => order, :notes => "test")
end
```

随后，在迁移 7 中，将表由 `order_histories` 重命名为 `order_notes`。这时，也已经将模型 `OrderHistory` 改名为 `OrderNote` 了。

现在决定要删除开发数据库，然后重新应用所有迁移。这么做时，迁移在迁移4中抛出一个异常：该应用程序不再包含 OrderHistory 类，因此迁移失败。

Tim Lucas 提出的一种解决方案是，创建在迁移过程中需要的本地虚拟版本模型类。例如，即使该应用程序不再包含 OrderHistory 类，下面的第四个迁移的版本仍然奏效：

```
class CreateOrderHistories < ActiveRecord::Migration

  ▶ class Order < ActiveRecord::Base; end
  ▶ class OrderHistory < ActiveRecord::Base; end

  def self.up
    create_table :order_histories do |t|
      t.integer :order_id, :null => false
      t.text :notes

      t.timestamps
    end

    order = Order.find :first
    OrderHistory.create(:order => order_id, :notes => "test")
  end

  def self.down
    drop_table :order_histories
  end
end
```

只要该模型类中不包含任何额外的功能，并且这些功能都不在迁移中使用，那么上述内容都将正常工作。这里创建的所有内容都是最基本的版本。

23.3.4 定义索引

迁移可以（或者可能说“应该”）为表定义索引。例如，我们可能注意到，一旦应用程序在数据库中有大量的订单，基于顾客名称的搜寻时间将超过预期。这时就应该使用这个适合的 `add_index` 方法来添加一个索引：

```
class AddCustomerNameIndexToOrders < ActiveRecord::Migration
  def self.up
    add_index :orders, :name
  end

  def self.down
    remove_index :orders, :name
  end
end
```

如果给 `add_index` 传入选项参数 `:unique => true`，那么将会创建独一无二的索引，迫使索引字段中的所有值变成唯一的。

默认情况下，索引的名字为 `index_table_on_column`。可以使用 `:name => "somenamename"` 选项重写名称。如果在添加索引时使用 `:name` 选项，也需要定义何时删除这个索引。

通过在 `add_index` 中传入一个包含字段名的数组，可以创建一个复合索引 (composite index)，即在多个字段中的索引。在这种情况下，命名该索引时只使用第一个字段名。

23.3.5 主键

Rails 假设每个表都有一个数字主键 (通常称为 `id`)，并且确保每个新加入到这个表中的行在这个字段的值都是唯一的。

下面将措辞重新整理，以便理解。

除非每个表都有一个数字主键，否则 Rails 真的不能很好地工作。Rails 对字段名并不挑剔。因此，对一般的 Rails 应用程序来说，极力推荐这样一种方式：跟随流程，让 Rails 拥有自己的 `id` 字段。

如果想要冒险试试，可以从使用不同的主键字段名开始 (但是，先保证这个字段是递增的数字)。通过在 `create_table` 调用中指定 `:primary_key` 选项可以做到：

```
create_table :tickets, :primary_key => :number do |t|
  t.text :description

  t.timestamps
end
```

这段代码把 `number` 字段添加到表中，并且设为主键：

```
$ sqlite3 db/development.sqlite3 ".schema tickets"
CREATE TABLE tickets ("number" INTEGER PRIMARY KEY AUTOINCREMENT
NOT NULL, "description" text DEFAULT NULL, "created_at" datetime
DEFAULT NULL, "updated_at" datetime DEFAULT NULL);
```

下一步尝试创建一个不是整数的主键。Rails 的开发者们并不认为这是个很好的想法，大概的原因是：迁移不允许这么做 (至少不允许直接这么做)。

23.3.6 没有主键的表

有时可能需要定义一个没有主键的表。在 Rails 中最常见的情况是连接表 (join table)——这些表只有两个字段，每个字段是另一个表的外键。要使用迁移创建一个连接表，必须告诉 Rails 不要自动添加 `id` 字段：

```
create_table :authors_books, :id => false do |t|
  t.integer :author_id, :null => false
  t.integer :book_id, :null => false
end
```

在这种情况下，可能会想在表中创建一个或多个索引，用以提高 `books` 表和 `authors` 表之间的导航能力。

23.4 高级迁移

大多数 Rails 开发者使用基础的迁移设施来创建和维护数据库模式。但是，无论是现在还是将来，进一步走入迁移都是很有用的。这一节将介绍一些更高级的迁移用法。

23.4.1 使用原生 SQL

迁移让数据库与应用程序的模式间相互独立。但是，如果迁移没有包含所需的方法，就得深入数据库的特定代码中。为此，Rails 提供两种方式。第一种方式是在诸如 `add_column` 方法中加入 `options` 参数。第二种方式为 `execute` 方法。

迁移中的一个常见的例子是将外键约束添加到子表中。

使用 `options` 或 `execute` 时，由于所提供的这两个位置的所有 SQL 都使用数据库的原生语法，因此可以尝试把迁移与一个特定的数据库引擎紧密地结合起来。

`execute` 方法的第二个参数选项指定了一个字符串，这个字符串将伪装成执行 SQL 过程中产生的日志信息。

23.4.2 扩展迁移

如果回顾前面章节中提到的商品项目迁移，可能会对两个 `option` 参数间的重复产生疑问。抽象出 `helper` 方法创建外键约束的过程，会是一个很好的解释。

要完成这件事，可以把如下方法添加到迁移的源文件中：

```
def self.foreign_key(from_table, from_column, to_table)
  constraint_name = "fk_#{from_table}_#{to_table}"

  execute %{
    CREATE TRIGGER #{constraint_name}_insert
    BEFORE INSERT ON #{from_table}
    FOR EACH ROW BEGIN
      SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
      WHERE
        (SELECT id FROM #{to_table} WHERE
         id = NEW.#{from_column}) IS NULL;
    END;
  }

  execute %{
    CREATE TRIGGER #{constraint_name}_update
    BEFORE UPDATE ON #{from_table}
    FOR EACH ROW BEGIN
      SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
      WHERE
        (SELECT id FROM #{to_table} WHERE
         id = NEW.#{from_column}) IS NULL;
    END;
  }

  execute %{
    CREATE TRIGGER #{constraint_name}_delete
    BEFORE DELETE ON #{to_table}
    FOR EACH ROW BEGIN
      SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
      WHERE
```

```

        (SELECT id FROM #{from_table} WHERE
          #{from_column} = OLD.id) IS NOT NULL;
      END;
    }
  end

```

(self. 是必需的, 因为迁移运行的原理与类的方法一样, 因此需要在文本中调用 foreign_key.)

在 up 迁移中, 可以使用下面的方法调用新方法:

```

def self.up
  create_table ... do
    end
    foreign_key(:line_items, :product_id, :products)
    foreign_key(:line_items, :order_id, :orders)
  end
end

```

然而, 我们也许想更进一步, 让所有迁移都可以使用 foreign_key 方法。要完成这件事, 可以在该应用程序的 lib 目录下创建一个模块, 然后在其中添加 foreign_key 方法。这时, foreign_key 方法就变成一个普通的实例方法, 而不是一个类的方法了:

```

module MigrationHelpers

  def foreign_key(from_table, from_column, to_table)
    constraint_name = "fk_#{from_table}_#{to_table}"

    execute %{
      CREATE TRIGGER #{constraint_name}_insert
      BEFORE INSERT ON #{from_table}
      FOR EACH ROW BEGIN
        SELECT
          RAISE(ABORT, "constraint violation: #{constraint_name}")
        WHERE
          (SELECT id FROM #{to_table} WHERE id = NEW.#{from_column}) IS NULL;
      END;
    }

    execute %{
      CREATE TRIGGER #{constraint_name}_update
      BEFORE UPDATE ON #{from_table}
      FOR EACH ROW BEGIN
        SELECT
          RAISE(ABORT, "constraint violation: #{constraint_name}")
        WHERE
          (SELECT id FROM #{to_table} WHERE id = NEW.#{from_column}) IS NULL;
      END;
    }

    execute %{
      CREATE TRIGGER #{constraint_name}_delete
      BEFORE DELETE ON #{to_table}
      FOR EACH ROW BEGIN
        SELECT
          RAISE(ABORT, "constraint violation: #{constraint_name}")
    }
  end
end

```

```

WHERE
  (SELECT id FROM #{from_table} WHERE #{from_column} = OLD.id) IS NOT NULL;
END;
}
end
end

```

现在, 在该迁移文件的最上方添加如下内容, 就可以把它添加到任何迁移中了:

```

► require "migration_helpers"

class CreateLineItems < ActiveRecord::Migration
  ► extend MigrationHelpers

```

包含 `require` 的一行将模块定义添加到迁移代码中, 而 `extend` 那一行则将 `MigrationHelpers` 模块中的方法作为类的方法添加到迁移中。用这个技术就可以开发并且共享任何数量的迁移帮助程序了。

(另外, 有人写了一个插件[⊖], 该插件可以自动地处理添加外键约定的事情。如果想让生活变得更简单, 可参考它。)

23.4.3 自定义消息和基准测试程序

虽然这不全是一个高级迁移该有的内容, 但有时在高级迁移中输出自己的信息和基准测试程序是很有用的。可以在 `say_with_time` 方法中做到这件事情:

```

def self.up
  say_with_time "Updating prices..." do
    Person.all.each do |p|
      p.update_attribute :price, p.lookup_master_price
    end
  end
end
end

```

`say_with_time` 方法将输出块代码执行前传入的字符串, 并且在该块完成后输出基准测试程序。

23.5 当迁移变糟时

迁移有一个严重的问题。用于更新数据库模式的基础 DDL 语句不是事务性的。当然, 这并不是 Rails 的缺陷——大多数的数据库都不支持 `create table`、`alter table`, 以及其他一些 DDL 语句的回滚。

来看看迁移尝试在数据库中添加两个表的情况:

```

class ExampleMigration < ActiveRecord::Migration
  def self.up
    create_table :one do ...

```

⊖ <http://wiki.rubyonrails.org/rails/pages/AvailableGenerators>

```

end
create_table :two do ...
end
end

def self.down
  drop_table :two
  drop_table :one
end
end

```

在事件的正常过程中，up 方法添加表 one 和表 two，而 down 方法则用于删除它们。

倘若在创建第二个表的时候出现了问题，将会发生什么？我们会最终得到一个包含表 one 但不包含表 two 的数据库。当然，迁移中的任何问题都可以修复，但这个问题除外——如果想试试看，也会失败，因为表 one 已经存在。

可以尝试回滚迁移，但也不会奏效。因为初始的迁移失败，数据库的模式版本并没有更新，因此 Rails 不会试着回滚。

这时，也许情况已经一团糟了，然后你手动修改模式信息，删除表 one。但是，这么做不太值得。此时，推荐的做法是单单删除整个数据库，再重新创建这个数据库，接着应用迁移来让它回滚到先前的版本。这么做将毫无损失，并且可以得到一个一致的模式。

在这个讨论中涉及的所有内容都表明，在生产数据库中使用迁移是很危险的。该不该运行它们？还真的说不准。如果在组织中有数据库管理员，请他们帮忙会是个很好的办法。如果必须自己解决问题，那么就衡量风险了。但是，如果决定这么做，请切记一定要先备份数据库。然后才可以到那台承载生产服务器的机器上，在该应用程序目录下执行迁移，只要输入下面的命令：

```
depot> RAILS_ENV=production rake db:migrate
```

这是本书开始时免责声明中所述内容的一种情况。如果该操作删除了你的数据，我们不承担任何责任。

23.6 迁移外的模式管理

到目前为止，本章内讨论过的所有迁移都可以作为 Active Record 模块连接对象上的方法使用，因此也支持 Rails 应用程序模型、视图和控制器内的访问。

例如，如果 orders 表在 city 字段中有一个索引，你可能会发现一个特定的长期运行报告速度提高了很多。但是，在日常的应用程序运行中并不需要那个索引，并且还有测试表明，对该索引的维护明显减慢了该应用程序的速度。

可以编写一个方法，这个方法将创建这个索引，然后运行一个代码块，最后删除这个索引。它可以是一个模型内的私有方法，也可以是库里执行的方法：

```

def run_with_index(column)
  connection.add_index(:orders, column)
  begin
    yield
  ensure
    connection.remove_index(:orders, column)
  end
end
end

```

模型中统计数据收集的方法，可以使用如下代码：

```
def get_city_statistics
  run_with_index(:city) do
    # .. calculate stats
  end
end
```

23.7 本章小结

在 Depot 应用程序的整个开发过程中，甚至在部署过程中，已经非正式地使用了迁移，而在这一章中展示了迁移是如何作为有原则、有纪律的基础方法，去配置管理数据库模式。

在本章中讨论了如何创建、重命名，以及删除字段和表；学习了管理索引和键；学习了应用和撤销整组改变；甚至学会了去混合自定义的 SQL，所有操作步骤都是完全可重现的。

到这里，已经介绍了 Rails 的外部。接下来的几章会钻研得更深。这些章节会展示如何将 Rails 拆散，以及如何将其重组到一起。沿途的第一站会讨论在 Web 服务器环境外，如何使用选择 Rails 类和方法。

24.1 用 Active Record 开发独立应用程序

第一件想要做的事情是不受限制地访问数据，可以在独立应用程序中以命令行的方式用 Active Record 的特性轻松完成这件事情。尤其看如何实现“懒”（lazy）的编程方式，之所以能够做到这一点是因为它并不真是懒的。——毕竟这里讨论的是 Rails，它会将使用简单方法去进行编程。而将编写独立应用程序，使用 Active Record 来封装 SQL 数据库查询，以非独立应用程序在数据库特定 ID 的订单后，修改购买者姓名，且在数据库中保存其结果。同时，更简单地进行。

```
require "rubygems"
require "active_record"

ActiveRecord::Base.establish_connection(adapter: "sqlite3",
  :database => "development.sqlite3")

class Order < ActiveRecord::Base
end
```

② 懒（lazy）这里不是懒惰，而是指在需要时再加载，而不是在加载时就把所有数据都加载进来。

第 24 章

非浏览器应用

在本章中，我们将学习：

- 调用 Rails 的方法
- 访问 Rails 应用程序的数据
- 操作远程数据库

前面的章节主要关注服务器和使用者之间的通信，而这种通信主要是通过 HTML 协议进行。然而并非所有基于 Web 的通信都需要用户直接参与。本章将集中讨论如何从一个独立的脚程序里访问你的 Rails 应用程序及其数据。

不想使用浏览器访问 Rails 应用程序的原因是多种多样的。例如，如果只想装载数据库或者周期性地同步数据库，使用 cron 这样的后台工具就可以了。又如，现成的程序甚至是 Rails 应用程序，想要直接访问（另一个）Rails 应用程序的数据，说不定它们还不在同一台机器上。还有，想要使用命令行界面，这不是必需的理由，而是充分的。

无论你的理由是什么，Rails 始终都在那里。正如将要看到的那样，下面的工作或多或少会与 Rails 有关。

在开始之前，先假设应用程序与安装 Rails 以及储存数据是同一台机器，即这三者同在一台机器里面。然后阐述怎么在远程机器上做相同的工作。

24.1 用 Active Record 开发独立应用程序

第一件想要做的事情是不受制约地访问数据。可以在独立应用程序内充分利用 Active Record 的特性轻松完成这件事情。先看看如何实现“硬”（hard）的编程方式（之所以对硬字添加引号，是因为它并不真是硬的[⊖]——毕竟这里讨论的是 Rails）。之后将使用简单方法进行编程。

下面将编写独立应用程序，使用 Active Record 来封装 SQLite 3 数据库订单表。该独立应用程序在找到特定 id 的订单后，修改购买者姓名，且在数据库中保存其结果，同时更新原始行记录。

```
require "rubygems"
require "active_record"

ActiveRecord::Base.establish_connection(:adapter => "sqlite3",
                                       :database => "db/development.sqlite3")

class Order < ActiveRecord::Base
end
```

⊖ 硬（hard）这里是双关语，即难与硬，硬编程是指把程序写得难以扩充。——译者注


```
order = Order.find(1)
order.name = "Dave Thomas"
order.save
```

全部代码都在这里——这种情况不需要什么配置信息（除了数据库的连接部分之外）。Active Record 从数据库模式本身已获得了所需要的信息，且已考虑到了所有必要的细节。

现在已经看到了“硬”编程方式，接着来看看简单的编程方式，Rails 将处理数据库的连接且载入所有模型。

```
require "config/environment.rb"
order = Order.find(1)
order.name = "Dave Thomas"
order.save
```

为了装载应用程序，Ruby 需要查找到 config/environment.rb 文件，它包含了此应用程序的有关设置。可用 require 语句将此文件的完整路径列出来，或在 RUBYLIB 环境变量中包含该文件的路径。需要注意的另一个环境变量是 RAILS_ENV，它用来区分开发、测试和生产环境。

一旦导入了这个文件，就可以大体上访问应用程序的上述组件，这种做法正与 14.4 节中介绍的是一样的。

使用单个 require 语句就已完成所有任务。再也没有比这更容易的事情了。不过，不论信还是不信，有时只想访问 Rails 所提供的部分功能，而不超越 Rails 应用程序的关联。接下来将讨论这个问题。

24.2 使用 Active Support 库功能

Active Support 是所有 Rails 组件的共享库。其中有些库的目的是在 Rails 内部使用，而所有共享库都可用于非 Rails 应用程序。

认识这一点很重要。譬如，你开发 Rails 应用程序。在开发过程中，产生了一组类，或者只是产生了一组方法，但是却想在非 Rails 应用程序中使用它们。你开始复制并粘贴这段代码到单独的文件中，然后发现它并不能运行。这不是因为代码在逻辑上与原先的应用程序有任何的依赖关系，而是因为它使用了 Rails 所提供的其他方法和类。

下面将开始简短说明这个库的一些最重要的内容，且将展示这些内容是如何为应用程序服务。

24.2.1 核心扩展

Active Support 对一些 Ruby 内置类进行了扩展，这种扩展方式是有趣的、有用的且有时候是异想天开的。在本节里列出最常用的核心扩展（core extension）。^①

- Array（数组）：second^②、third、fourth、fifth 以及 forty_two（第四十二个数组元素）。这些补充了 Ruby 提供的 first（第一个数组元素）和 last（最后一个数组元素）方法。
- CGI：escape_skipping_slashes。顾名思义，与函数 escape 的区别是不能转义斜杠。

① 下面括号里分别是中文翻译的类名和方法名。——译者注

② 第二个数组元素。后同。——译者注

- **Class (类)**: 类属性的存取器、代理存取器、可继承读写器, 以及后代方法 (descendant) 也叫子类方法 (subclass)。这些方法太多, 无法一一列举, 详见有关文档。
- **Date (日期)**: yesterday (昨天)、future? (是否是未来的判断) 和 next_month (下个月) 等。
- **Enumerable (枚举)**: group_by (按组)、sum (合计)、each_with_object (带对象的每一个)、index_by (按...索引)、many? (是否是许多的判断) 和 exclude? (是否是不包含的判断)。
- **File (文件)**: atomic_write (以基元方式写入) 和 path (文件路径)。
- **Float (浮点精度)**: 添加了可选参数 precision, 来定义四舍五入的精度。
- **Hash (散列)**: diff (差别比较)、deep_merge (深度合并)、except (除外)、stringify_keys (字符串化键)、symbolize_keys (符号化键)、reverse_merge (反向合并) 以及 slice (切片)。这些方法中的大部分都是有感叹号结尾的变体方法。
- **Integer (整数)**: ordinalize (得到整数的序数, 如 3rd)、multiple_of? (倍数判断)、months (月)、years (年), 参见 Numeric。
- **Kernel (内核)**: debugger (调试器)、breakpoint (断点)、silence_warnings (静默警告)、enable_warnings (启用警告)。
- **Module (模块)**: 模块属性的存取器、别名支持、代理、过时 (deprecation)、内部读写器、同步以及亲缘 (parentage)。
- **Numeric (精度, 标度值)**: bytes (字节)、kilobytes (千字节) 和 megabytes (兆字节) 等; seconds (秒)、minutes (分钟) 和 hours (小时) 等。
- **Object (对象)**: blank? (对象是否为空的判断)、present? (对象是否存在的判断)、duplicable? (对象是否复制的判断)、instance_values (实例变量值)、instance_variable_names (实例变量名)、returning[Ⓔ] 以及 try (尝试)。

David 说

扩展基类不会导致灾变

第一次看到代码 `5.months+30.minutes`, 通常是由一种恐慌的状态取代惊讶的心情。如果每个人都可以改变整数类的行为, 那么会不会导致一种完全不可维护的糟糕局面? 是的, 如果每个人都在这样做, 那么就会发生。但他们并没有那样做, 所以也就不会发生了。

不能把 Active Support 看成随机收集的一堆 Ruby 语言扩展库, 如不可以把人们喜欢的功能强加到字符串类。而要把它当成是在 Rails 程序员中广泛传播的一种 Ruby 语言的方言。因为 Active Support 是 Rails 必不可少的一部分, 在任何 Rails 应用程序中都可以使用代码 `5.months`。那就否定存在大量个人 Ruby 方言的问题。

Active Support 扩展了 Ruby 语言, 并且提供了 Rails 和 Ruby 两者的最好特性, 这实际上是对语言的语境的标准化。

Ⓔ 最新 Rails 3.0.3 以后版本源代码已经没有 `returning` 方法, 最好用 `tap` 方法。——译者注

- **String (字符串)**: `exclude?` (不包含判断)、`pluralize` (获取单词的复数)、`singularize` (获取单词的单数)、`camelize` (转换骆驼拼法)、`titleize` (转换标题各单词首字母大写)、`underscore` (下划线分隔)、`dasherize` (短横线分隔)、`demodulize` (去掉模块名双冒号之前)、`parameterize` (参数化, 替换特殊符号可组成 URL)、`tableize` (转成下划线分隔)、`classify` (去下划线)、`humanize` (去下划线变空格分隔, 也可以做与 `foreign_key` 相反的转换)、`foreign_key` (加外键 id)、`constantize` (常数)、`squish` (挤压多行, 或去掉 `whitespace` 保留成单个空格)、`mb_chars` (多字节字符)、`at` (字符位置)[⊖]、`from` (从)、`to` (到)、`first` (第一个字符串字母)、`last` (最后一个字符串字母)、`to_time` (转成时间)、`to_date` (转成日期) 以及 `try` (尝试)。
- **Time (时间)**: `yesterday` (昨天)、`future?` (对未来判断) 和 `advance` (在之前) 等。

上面列出了很多类的方法, 但这些方法都是非常简洁的, 其中很多方法的源代码短小到了就一行代码的程度。虽然使用的机会很少, 但是它们在程序中都已经准备就绪。

是不是太多了呢? 大多数方法不一定直接用得上。但是很快你就会发现这些附加方法中的一小部分使用起来就像用 Ruby 语言自身提供的方法一样。尽管可以在联机文档[⊖]里找到所有的方法, 但最好的学习方式仍是直接在控制台命令行 (rails console) 里多加练习, 这里几个例子供参考:

- `2.years.ago` (2 年前)
- `[1,2,3,4].sum` (数组里元素求和)
- `5.gigabytes` (5GB 字节)
- `"man".pluralize` (男人复数形式)
- `String.methods.sort` (列出字符串类所有排序过的方法)

因为没有一种最好的方式来确定库的哪些子集是有用的, 所以只能让你知道, 这些方法存在, 并且当发现自己有需要时, 就检查这些文档, 因为 Rails 开发人员可能已经添加了所需要的方法。

24.2.2 附加的 Active Support 类

除了扩展 Ruby 语言本身提供的基本对象外, Active Support 还提供了大量附加功能。与扩展核心类不同的是, 这些类倾向于为其他的 Rails 组件提供特定的支持, 但也可直接利用这些功能。

- **BasicObject (基本对象)**: Object 的基类, 即 Ruby 1.9 根类向后兼容到 Ruby 1.8 的相同类名。
- **Benchmarkable (可评比的)**: 在模板中测量代码块执行时间, 把结果记录到日志文件。
- **Cache::Store (缓存存放)**: 多种缓存实现, 基于文件或内存; 带上要同步的或要压缩的可选项。

⊖ 其实 Rails 中并没有英文原版书中的 `at?` 方法。——译者注

⊖ <http://as.rubyonrails.org/>

- **Callbacks (回调)**: 在对象生命周期里提供钩子 (hook) 方法。
- **Concern (关注) 和 Dependencies (依赖)**: 以模块化方式辅助依赖管理。
- **Configurable (配置)**: 提供 Hash 类变量的配置方式。
- **Deprecation (折旧)**: 为过期的方法提供警示和替代方法告知。
- **Duration (时段)**: 附加的方法, 例如 ago (在 ... 前) 和 since (自从 ...) 等。
- **Gzip (压缩)**: 为了方便压缩和解压缩字符串的方法集。
- **HashwithIndifferentAccess (不同方式访问散列)**: 同时可以用参数 `params[:key]` 和 `params['key']` 获得散列值。
- **I18n**: 国际化支持。
- **Inflections (词形变化)**: 处理英语里复数不一致规则。
- **JSON**: JavaScript Object Notation (JavaScript 对象标记) 数据交换语言的编码和解码方法。
- **LazyLoadHooks (延迟加载钩子)**: 支持模块的延迟初始化。
- **Memoizable**: 缓存方法调用的结果。
- **MessageEncryptor (消息加密器)**: 将存放于不可信的位置数据值加密。
- **MessageVerifier (消息验证)**: 生成并验证前面的消息 (避免擅自改动)。
- **MultiByte (多字节)**: 对编码支持 (主要针对 Ruby 1.8.7)。
- **Notifications (通知)**: 仪表测量 API。
- **OptionMerger**: 深度合并 lambda 表达式。
- **OrderedHash (有序的散列) 和 OrderedOptions (有序化参数)**: 提供有序散列支持 (主要针对 Ruby 1.8.7)。
- **Railtie (枕木)**: 定义框架其余部分可依赖的核心对象。
- **Rescueable (可拯救的)**: 简化异常处理。
- **SecureRandom (安全随机码)**: 生成随机码, 适用在生成 HTTP cookie 会话键。
- **StringInquirer (字符串询问器)**: 提供一种相当好的等同测试方式。
- **TestCase (测试案例)**: 测试框架独立界面来测试案例。
- **Time (时间) 和 TimewithZone (带时区的时间)**: 支持更多的时间计算和转换。

尽管本书不会深度涉及所有 (现有) 的类和方法, 并且也不会逐个去分析某个方法, 如 `TimewithZone`, 但是前面的列表足以在 Rails 指南和 API 文档中找到所需要的函数。下面本书也将展示如何在独立应用程序里使用这些方法:

```
require "rubygems"
require "active_support/time"
Time.zone = 'Eastern Time (US & Canada)'
puts Time.zone.now
```

因此, 如果认为自己对一个或多个扩展感兴趣, 那么就可以简单地加载你所需要的扩展 (如代码, `require "active_support/basic_object"` 或 `require "active_support/core_ext"`) 或用代码 `require "active_support/all"` 调用整个库。

24.2.3 使用 Action View 帮助程序

好了，虽然这并不完全属于 Active Support 的范围，但它比较接近。凡适用于 Active Support，也同样适用于 Rails 的其他部分，不过，大多数路由、控制器和 Action View 的方法往往都只与处理活动的 Web 请求有关。

值得注意的一个例外是 Action View 帮助程序。下面的例子说明如何从独立应用程序访问 Action View 帮助程序：

```
require "rubygems"
require "action_view"
require "action_view/helpers"
include ActionView::Helpers::DateHelper
puts distance_of_time_in_words_to_now(Time.parse("December 25"))
```

总之，虽然这里比通常获得 Active Support 方法需要稍多做一点工作，但它仍然是切实可行的。

到此已经讨论了在同一台电脑（作为服务器）上从脚本程序来访问部分或全部 Rails 功能；接下来，开始把目光移到远程访问 Rails 应用程序。

24.3 使用 Active Resource 开发远程应用程序

在编写应用程序时，你可能会发现，并非所有的数据都很整洁且分门别类地存在于数据库中。它可能并不在数据库中。它甚至可能还没有在你的机器上。那就是为什么需要网络服务（web services）。而 Active Resource 则承担了为 Rails 提供网络服务的工作。请注意，这里的网络服务（web services），使用了小写 w 和小写 s，而不是 Web 服务（Web Services）如 SOAP、WSDL 和 UDDI。

24.3.1 访问和更新简单属性

下面将演示 Active Resource 的实例，并继续开发 Depot 应用程序。关键的区别在于，这一次将通过客户端应用程序远程访问 Depot 应用程序。对于客户端，将使用命令 `rails console`。先检查并确保 Depot 服务器正在运行。然后开始建立客户端：

```
work> rails new depot_client
work> cd depot_client
```

为 Product 模型编写测试桩（stub）：

```
depot_client/app/models/product.rb
```

```
class Product < ActiveRecord::Base
  self.site = 'http://dave:secret@localhost:3000/'
end
```

的确没有很多代码。类 Product 是从 ActiveRecord::Base 类继承的。其中存在单独一行语句，标识了用户名、密码、主机名和端口号。在实际的应用程序中，应该各自独立地获得用户名和密码，而不是将其硬编程到模型中[⊖]，但在这里只是理解概念。下面开始使用测试桩：

⊖ 在模型中直接写出用户名和密码。——译者注

```

depot_client> rails console
Loading development environment (Rails 3.0.5)
>> Product.find(3).title
=> "Programming Ruby 1.9"

```

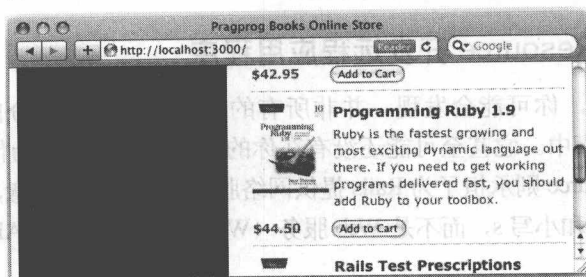
成功了！再大胆一点。如何把这本书价格降价 5 美元销售？

```

depot_client> rails console
Loading development environment (Rails 3.0.5)
>> p = Product.find(3)
=> #<Product:0x282e7ac @prefix_options={}, ... >
>> puts p.price
49.5
=> nil
>> p.price-=5
=> #<BigDecimal:7f88f8ebfef0,'0.445E2',18(36)>
>> p.save
=> true

```

这似乎好得难以置信。但为了验证这一点，可以只要简单地通过浏览器访问这个商店：



不知道你的感觉是什么，对于我们来说，Active Resource 太高科技了，简直像变魔术一样。

24.3.2 关系和集合

上面十分成功地处理了类 Product，下面继续看看类 Order。先编写测试桩：

```

depot_client/app/models/order.rb

class Order < ActiveRecord::Base
  self.site = 'http://dave:secret@localhost:3000/'
end

```

看起来很不错，就试试吧：

```

depot_client> rails console
>> Order.find(1).name
=> "Dave Thomas"
>> Order.find(1).line_items
NoMethodError: undefined method 'line_items' for #<Order:0x2818970>

```

现在我们需要理解幕后的工作原理。让我们回到理论上，但不用担心，它的内容并没有多少。

之所以神奇是因为它利用代码支架规定的所有 REST 和 XML 接口。为了得到产品清单，它就会访问 `http://localhost:3000/products.xml`。要得到第 2 号产品，就需要对 `http://localhost:3000/`

products/2.xml 使用 GET 方法。要对第 2 号产品修改进行保存,就需要对 http://localhost:3000/products/2.xml 使用 PUT 方法来更新产品信息。

因此,这便是神奇所在——产生 URL,类似于第 20 章讨论的那样,而产生(和使用)XML,如同 20.1 节讨论的那样。下面看看具体操作。先给 line_items 做个订单的嵌套资源。为此,在服务器应用程序中需要编辑 config.routes 文件:

```
resources :orders do
  resources :line_items
end
```

现在修改 LineItems 控制器代码,来寻找在变量 params 中的 :order_id,并把它作为 LineItems 控制器代码的一部分。问题在于,在 9.3 节中,已经对该方法进行了修改(以满足那时的需要)。因此,必须再次修改此方法的代码以满足处理两种输入类型的要求:

```
depot_u/app/controllers/line_items_controller.rb

def create
  @cart = current_cart
  if params[:line_item]
    # ActiveResource
    params[:line_item][:order_id] = params[:order_id]
    @line_item = LineItem.new(params[:line_item])
  else
    # HTML forms
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product.id)
  end
  respond_to do |format|
    if @line_item.save
      format.html { redirect_to(store_url) }
      format.js { @current_item = @line_item }
      format.xml { render :xml => @line_item,
        :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @line_item.errors,
        :status => :unprocessable_entity }
    end
  end
end
```

为了看看它究竟是什么样子,先获取该数据:

```
<?xml version="1.0" encoding="UTF-8"?>
<line-items type="array">
  <line-item>
    <price type="decimal">49.5</price>
    <created-at type="datetime">2010-02-11T03:11:44Z</created-at>
    <product-id type="integer">3</product-id>
    <quantity type="integer">1</quantity>
```

```

<order-id type="integer">1</order-id>
<updated-at type="datetime">2010-02-11T03:12:11Z</updated-at>
<id type="integer">10</id>
<cart-id type="integer" nil="true"></cart-id>
</line-item>
<line-item>
  <price type="decimal">42.95</price>
  <created-at type="datetime">2010-02-11T03:14:18Z</created-at>
  <product-id type="integer">2</product-id>
  <quantity type="integer">2</quantity>
  <order-id type="integer">102</order-id>
  <updated-at type="datetime">2010-02-11T03:14:25Z</updated-at>
  <id type="integer">11</id>
  <cart-id type="integer" nil="true"></cart-id>
</line-item>
</line-items>

```

现在给 Dave 的第一次购买优惠 20%:

```

>> li = LineItem.find(:all, :params => {:order_id=>1}).first
=> #<LineItem:0x2823334 @prefix_options={:order_id=>1}, ... >
>> puts li.price
39.6
=> nil
>> li.price*=0.8
=> 31.68
>> li.save
=> true

```

这样,一切都如所希望的那样了。需要注意的是使用 `:params`。其处理方法正如期望生成的 URL 那样,将替换与 `ActiveResource` 类提供的 `site` 模板匹配的参数。而将其他的参数用作查询参数。服务器使用该 URL 进行路由寻找,并且以 `params` 数组方式使用这些参数。

最后,将 `LineItems` 对象添加到订单中:

```

>> li2 = LineItem.new(:order_id=>1, :product_id=>2, :quantity=>1,
>> :price=>0.0)
=> #<LineItem:0x7fd986812b00 @prefix_options={},
@attributes={"price"=>0, "product_id"=>2, "quantity"=>1, "order_id"=>1}>
>> li2.save
=> true

```

24.3.3 汇总整理

虽然刚开始 `ActiveResource` 似乎有点神奇,但是它确实在很大程度上依赖于在本书前面描述的概念。这里有几个要点如下:

- 认证使用基本认证机制,这种机制不仅网站本身已经支持,而且也不违背 Web 现有可提供的协议方式。无论如何,Active Resource 只能做已解决的事情。

请注意,如果使用基本认证,那么就要使用传输层安全 (Transport Layer Security, TLS),也称为安全套接字层 (Secure Sockets Layer, SSL) 或 HTTPS,以确保不能让第三方嗅到密码。

- 虽然 Active Resource 不能有效地利用会话 (session) 或 cookie,但是这并不意味着服务器没有继续生成它们。

要么关闭 Active Resource 所使用接口的会话，要么确保使用基于 cookie 的会话。否则，服务器将设法终止不再需要的大量会话。更多细节见 20.3 节。

- 20.1 节已讨论了集合和成员。ActiveResource 定义了处理集合的 4 个类方法和处理成员的 4 个实例方法。这些方法名是 get、post、put 和 delete。方法名确定了基本 HTTP 所使用的方法。

对于这些方法的每一个，其第一个参数都是集合或成员名称。此信息只是用来构造 URL。可以指定附加的变量 :params，这个变量既可在 self.site 中匹配值，也可补充为查询参数。

与你的想象相比，你将可能最终变得更多地使用它。可能并不想要获取所有的订单，而只想提供如何获取近期或逾期订单的一个接口。对于所有这些方法，可以做任何事情，能限制你的只有你的想象力。

- Active Resource 把 HTTP 状态码映射成异常，对应关系如下：

```
301,302 ActiveResource::Redirection (重定向)
400 ActiveResource::BadRequest (请求错误)
401 ActiveResource::UnauthorizedAccess (访问未得到授权)
403 ActiveResource::ForbiddenAccess (禁止访问)
404 ActiveResource::ResourceNotFound (未找到资源)
405 ActiveResource::MethodNotAllowed (方法未得到使用权)
409 ActiveResource::ResourceConflict (冲突资源)
422 ActiveResource::ResourceInvalid (无效资源)
401..499 ActiveResource::ClientError (客户端错误)
```

- 通过覆盖 ActiveResource 基类的校验方法，还可以提供用户端校验。这种做法的效果跟在 ActiveRecord 里校验相同。服务器校验失败时结果得到返回码 422，可以用同样的方式访问失败状态码，详见 7.1 节。
- 虽然默认格式是 XML，但是也可使用 JSON 作为一种替代。在客户端的类中，只要简单地设置 self.format 为 :json。注意：客户端的日期格式将变成 ISO8601/RFC3339 格式字符串，且将小数当成类 Float 的实例。
- 除了 self.site 之外，还可以单独设置 self.user 和 self.password。
- self.timeout 作用是定义网络服务请求应等待多久，其单位为秒，在超时之后，放弃服务并且抛出错误 Timeout::Error。

24.4 本章小结

我们打破了从浏览器访问 Active Support、Action View 和 Active Record 方法的方式，而直接从独立脚本完成这种访问。这可把在命令行运行的脚本集成到现有的应用程序中，也可使用工具如 cron 定期地和自动地运行脚本。

最后，使用 Active Resource 可突破只能与在同一台机器上运行应用程序脚本的限制。虽然这需要稍微多点配置工作，但是它提供了一种可行的和安全的访问 Rails 应用程序数据的途径。

接下来将探讨其他单独可安装的组件，这些组件是与 Rails 捆绑在一起的。

第 25 章

Rails 包依赖关系

在本章中，我们将学习：

- XML 和 HTML 模板
- 管理应用程序的包依赖关系
- 编写任务脚本
- 与 Web 服务器交互

到目前为止，虽然我们已经讨论了 Rails 本身的基础，但仍然有很多东西没有介绍。Rails 之所以如此强大，是因为 Rails 提供了构建组件的功能。

这些组件我们一点都不陌生，因为在前面章节已经逐个用到过。至此，应该对 atom 模板，HTML 模板，命令 `rake db:migrate`、`bundle install` 以及 `rails server` 都非常熟悉了。

虽然本章内容不仅超出了日常开发所需的知识，而且还将展示如何单独使用每个组件，但这不代表要对所有组件做详尽的描述。每个组件都需要一本小书的篇幅，只有这样才能充分论述其全面的功能。而这一章的目的是介绍一些关键组件，以便读者自己能展开探索这方面的知识。

下面从介绍若干这样的包依赖关系开始，再进一步进入驱动视图的基础模板引擎。然后学习 Bundler（捆绑器）组件，这个组件用来管理包依赖关系。最后将展示如何用 Rack 和 Rake 把这些代码片段组装到一起。

25.1 用构建器生成 XML

构建器（Builder）是个独立的库，它可以在代码中表示结构化的文本（如 XML）。构建器模板（其文件扩展名为 `.xml.builder`）包含 Ruby 的代码，利用 Builder 库，这些代码可以生成 XML。

下面是个简单的构建器模板，它将货品名称和价格的列表以 XML 格式输出：

```
depot_1/app/views/products/index.xml.builder
xml.div(:class => "productlist") do

  xml.timestamp(Time.now)

  @products.each do |product|
    xml.product do
      xml.productname(product.title)
      xml.price(product.price, :currency => "USD")
    end
  end
end
```

如果这段代码让人想起 12.2 节所创建的模板，这是与使用 Atom 帮助程序相关的，那是因

为 Atom 帮助程序就是在 Builder 功能上搭建的。

有了适当数量的货品集合(通过从控制器传入),模板可能会生成如下内容:

```
<div class="productlist">
  <timestamp>Wed Jul 21 07:01:41 -0400 2010</timestamp>
  <product>
    <productname>Debug It!</productname>
    <price currency="USD">34.95</price>
  </product>
  <product>
    <productname>Programming Ruby 1.9</productname>
    <price currency="USD">49.5</price>
  </product>
  <product>
    <productname>Web Design for Developers</productname>
    <price currency="USD">42.95</price>
  </product>
</div>
```

注意 Builder 怎么取得方法名并把它们转成 XML 的标签。当一提到 `xml.price` 时,就创建了 `<price>` 标签,其内容是第一个参数,其属性由后续的散列来设定。如果要用的标签名与现存的方法名存在冲突,就需要使用 `tag!` 方法来生成标签:

```
xml.tag!("id", product.id)
```

Builder 可以生成几乎任何所需要的 XML。它支持命名空间(namespace)、实体(entity)、处理指令(processing instruction)甚至 XML 注释(XML comment)。详细内容请查阅 Builder 文档。

尽管 HTML 表面上看起来很像 XML,但用来生成 HTML 的模板引擎不同,带来的差别十分显著。接下来就讨论这个话题。

25.2 用 ERb 生成 HTML

简单地说,ERb 模板只是普通的 HTML 文件。如果模板没有包含动态内容,它只会原封不动地发送到用户浏览器。下面就是个完全有效的 `html.erb` 模板:

```
<h1>Hello, Dave!</h1>
<p>
  How are you, today?
</p>
```

可是,只呈现静态模板的应用程序,使用起来往往有点沉闷。可以使用动态内容来给它们增添些趣味:

```
<h1>Hello, Dave!</h1>
<p>
  It's <%= Time.now %>
</p>
```

JSP 程序员会认出这是内联表达式。ERb 对 `<%=` 和 `%>` 之间的所有代码进行评估求值,将结

果用 `to_s` 转成字符串，最后把字符串替换到结果页面里。标签内的表达式可以是任意的代码：

```
<h1>Hello, Dave!</h1>
<p>
  It's <%= require 'date'
    DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
                  Thursday Friday Saturday }
    today = Date.today
    DAY_NAMES[today.wday]
  %>
</p>
```

普遍认为，把大量的商业逻辑放到模板中，是一件非常糟糕的事情，这样会导致代码混乱的局面，要是真有“代码警察”的话，就肯定要抓你的。在 21.5 节中曾讨论过使用帮助程序来处理这个问题，那个办法更好一些。

有时候需要在这样的模板里存放代码：该模板不会直接生成任何输出。如果把起始标签中的等号去掉，里面内容仍执行，但不会有结果插入到模板里。所以可以把前面的例子这样写：

```
<% require 'date'
  DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
                  Thursday Friday Saturday }
  today = Date.today
%>
<h1>Hello, Dave!</h1>
<p>
  It's <%= DAY_NAMES[today.wday] %>.
  Tomorrow is <%= DAY_NAMES[(today + 1).wday] %>.
</p>
```

在 JSP 圈子里管这个叫脚本小程序 (scriptlet)。再说一遍，很多人要是发现你把代码往模板里放，一定会严厉地责骂你。别管他们了——他们都是一些教条主义者。把代码放到模板里没有错。只是不要放太多这样的代码（尤其不要把业务逻辑搅到模板里）。本书已经介绍过，可以成功地用帮助程序 (helper) 方法去抵制上面这种做法的诱导。

可以想象一下，在代码段之间的 HTML 文本，每一行代码仿佛都是由 Ruby 程序写出来的。代码段 `<%...%>` 加到了同一个程序里。编写的显式代码与 HTML 文本交织到一起。因此，在 `<%` 和 `%>` 之间的代码，可以影响到模板其他部分的 HTML 输出结果。

例如在这个模板里：

```
<% 3.times do %>
Ho!<br/>
<% end %>
```

出于安全的考虑，^①当用 `<%=...%>` 插入值时，其结果是转义的 HTML，但将其直接放到输出流之前。这通常符合人们的想法。

但是，如果要替换的文本里包含 HTML，并打算解析这些 HTML，就会造成 HTML 标签发

① http://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

生转义处理——假如创建了包含 `hello` 的字段，然后把它替换到模板里，用户看到的是 `hello` 而不是 `hello` [⊖]。Rails 框架里有两个帮助程序 (helper) 来处理这种情况。

`raw` (原始) 方法使字符串正确地传递到输出而没有转义。该方法提供最大的灵活性，当然安全性也最低。

`sanitize` (消毒) 方法提供一些保护功能。它取得包含 HTML 的字符串，并清除危险的元素：转义处理 `<form>` 和 `<script>` 标签，并把 `on=` 属性和以 `javascript:` 开头的链接移除。

Depot 应用程序中货品描述以 HTML 的方式呈现 (也就是说，曾经用 `raw` 方法将它们打上了安全标记)，这可以在其中嵌入格式信息。如果允许组织机构之外的人员输入这些描述，就要谨慎地用 `sanitize` 方法，以便成功地降低了黑客攻击网站的风险。

这两个模板引擎只是 Rails 框架依赖的众多 `gem` 中的两个。说到这里，下面就顺势讨论如何管理这些包依赖关系。

25.3 用 Bundler 管理包依赖关系

包依赖关系的管理看似是个难题。在开发过程中，可以选择安装所依赖 `gem` 的更新版本。一旦这么做了，可能会发现不能再现那些在产品中存在的问题了，因为每次运行都选择了不同的 `gem` 版本，这些 `gem` 是该应用程序所依赖的。

事实证明，对包依赖关系的管理，与对待程序源代码或数据库模式的管理同等重要。如果身处开发团队，会希望每个成员都使用相同版本的包依赖关系。在部署时，要确保测试所用的包依赖关系版本安装到目标机器上，并且它们实际上就是在产品环境中使用那些包依赖关系。

根据在应用程序目录下名为 `Gemfile` 的文件，`Bundler` [⊖] 控制着包依赖关系。在这个文件里，列出应用程序所需要的库。下面详细查看在 Depot 应用程序中 `Gemfile` 的内容：

```
depot_u/Gemfile
source 'http://rubygems.org'

gem 'rails', '3.0.5'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'
group :production do
  gem 'mysql'
end

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

# Deploy with Capistrano
gem 'capistrano'
```

⊖ `em` 是强调标签，结果是斜体。——译者注

⊖ <http://gembundler.com/>

```
# To use debugger (ruby-debug for Ruby 1.8.7+, ruby-debug19 for Ruby 1.9.2+)
# gem 'ruby-debug'
# gem 'ruby-debug19', :require => 'ruby-debug'
# Bundle the extra gems:
# gem 'bj'
# gem 'nokogiri'
# gem 'sqlite3-ruby', :require => 'sqlite3'
# gem 'aws-s3', :require => 'aws/s3'

gem 'will_paginate', '>= 3.0.pre'

# Bundle gems for the local environment. Make sure to
# put test-only gems in this group so their generators
# and rake tasks are available in development mode:
# group :development, :test do
#   gem 'webrat'
# end
```

第 1 行说明在哪里可以找到新的 gem 包, 以及现存 gem 的新版本。可以任意重复这一行, 以列出私有 gem 储存库。

下面一行说的是载入 Rails 的版本。需要注意的是, 它指定了特定的版本号。后面两行是注释, 可以使用这两行内容来运行 Rails 的最新版本。

剩下的几行列出了一些正在使用的 gem 和考虑使用的 gem。有些 gem 放在分组里, 如 :development、:test 或 :production, 并且只在相应的环境中有效。另一些 gem 包含可选择的参数 :require, 用来在 require 语句的装载名与 gem 名有区别时, 指定 require 语句要使用的名称。

在包含 will_paginate 的一行代码中, 有个以比较符号开始的版本定义符。虽然 Gemfile 文件支持许多这样的操作符, 但常用的只有两种。>= 使用的条件实在是十分少见, 这是因为在那里可以使 Gemfile 开发者维护的严格向后兼容性得到可靠保证, 所以需要定义的是最小版本号。

广泛推荐使用 ~>。除了最后一部分外, 基本上所有的版本部分都必须完全匹配, 而最后一个部分指定了一个最小值。因此, ~>3.1.4 与所有以 3.1 开头的版本匹配, 但不能小于 3.1.4。同样, ~>3.0 表明所有以 3.0 开头的版本字符串。

与 Gemfile 相伴的文件叫 Gemfile.lock。后者通常由两个命令来更新: bundle install 和 bundle update。两者之间的区别相当微妙。

在动手操作前, 先看一看 Gemfile.lock 是很有帮助的, 下面是一小部分节选:

```
GEM
  specs:
    abstract (1.0.0)
    builder (2.1.2)
    capistrano (2.5.19)
      highline
      net-scp (>= 1.0.0)
      net-sftp (>= 2.0.0)
      net-ssh (>= 2.0.14)
      net-ssh-gateway (>= 1.0.0)
    erubis (2.6.6)
      abstract (>= 1.0.0)
```

`bundle install` 使用 `Gemfile.lock` 作为出发点，只安装在文件里定义的各种 gem 版本。因此，把这个文件放入版本控制系统中检查十分重要，因为它将保证你的同事和部署目标都会使用完全相同的配置。

`bundle update` 会（毋庸置疑地）变更一个或多个已命名的 gem 并且将变更相应的 `Gemfile.lock` 文件。如果想使用某个特定版本的 gem，请按照如下流程进行：修改 `Gemfile` 文件来表达约束条件，然后运行 `bundle update`，列出想要更新的 gem。如果不指定 gem 列表，Bundler 将试图去更新所有的 gem——通常不推荐使用这种做法，尤其是临近部署的时候。

Bundler 还有 `runtime`（运行时）组件，用来保证应用程序严格地只加载了在 `Gemfile.lock` 里列出的 gem 版本。下面将进一步探索服务器是如何运行的。

25.4 用 Rack 实现与 Web 服务器的交互

在 Web 服务器环境里，Rails 框架运行应用程序。到目前为止，曾使用过两个独立的 Web 服务器：WEBrick 来自 Ruby 语言内置的服务器，而 Phusion Passenger 整合了 Apache HTTP 的 Web 服务器。还有许多其他可用的选择，包括 Mongrel、Lighttpd 和 Unicorn。

由此可能会得出这样的结论：Rails 具有可接入这些 Web 服务器的代码。在 Rails 的早期版本里，这个结论是正确的。但到了 Rails 2.3，这种整合托付给了名为 Rack^①的 gem 软件包。

也就是说，Rails 与 Rack 整合、Rack 与（例如）Passenger 整合、Passenger 与 Apache httpd 服务器整合。

虽然通常情况下这种整合是看不见的，但当启动命令 `rails server` 发出之后，就在起作用了，`config.ru` 文件还可以直接在 Rack 下启动程序：

```
depot_u/config.ru
```

```
# This file is used by Rack-based servers to start the application.
```

```
require ::File.expand_path('../config/environment', __FILE__)
run Depot::Application
```

用以下命令和上面的配置文件启动 Rails 服务器：

```
rackup
```

以这种方式启动完全相当于运行命令 `rails server`。为了演示单独应用 Rack 时的强大功能，可以从单纯的 Rack 应用重新开始：

```
depot_u/app/store.rb
```

```
require 'builder'
require 'active_record'
```

```
ActiveRecord::Base.establish_connection(
  :adapter => 'sqlite3',
  :database => 'db/development.sqlite3')
```

① rack 是机柜和机架的意思。——译者注

```

class Product < ActiveRecord::Base
end

class StoreApp
  def call(env)
    x = Builder::XmlMarkup.new :indent=>2

    x.declare! :DOCTYPE, :html
    x.html do
      x.head do
        x.title 'Pragmatic Bookshelf'
      end
      x.body do
        x.h1 'Pragmatic Bookshelf'

        Product.all.each do |product|
          x.h2 product.title
          x << "      #{product.description}\n"
          x.p product.price
        end
      end
    end

    response = Rack::Response.new(x.target!)
    response['Content-Type'] = 'text/html'
    response.finish
  end
end

```

这个应用利用了许多学过的知识。第一步，直接用 `require` 语句载入 `active_record` 和 `builder`。第二步，建立数据库连接，并定义类 `Product`。倘若把这个应用程序和 Rails 应用程序结合为一体，不必做任何事情，但在这里，使用的是单纯的 Rack 应用程序。

然后看应用程序本身。这是个简单的类，它定义了称为 `call` 的方法。这个方法接收单一参数 `env`，该参数包含请求的信息，且该参数并不为此应用程序所用。

这个应用程序使用 `Builder` 创建了简单的 HTML，用来呈现产品列表，然后构建响应、设置内容类型以及调用方法 `finish`。

再创建新的 `rackup` 文件，就能运行这个单独的应用了：

```

depot_u@store.ru

require 'rubygems'
require 'bundler/setup'

require './app/store'

use Rack::ShowExceptions

map '/store' do
  run StoreApp.new
end

```

这段脚本做的第一件事是初始化 Bundler，这将令所需的所有 gem 有正确的版本可用。然后，用 `requires` 语句加载商店应用程序。

接下来，它触发了 Rack 所提供的这些标准中间件（middleware）类之一；在出错时，该类负责对堆追踪（stack traceback）进行格式化。Rack 中间件就如同 Rails 过滤器——二者都能检查请求并调整生成的响应。

可以用命令 `rake middleware` 查看 Rack 为 Rails 应用程序所提供的中间件列表。

最后，将 `store` 的 URI（统一资源标识符）映射到这个应用程序上。

在命令行输入 `rackup` 命令，启动应用：

```
rackup store.ru
```

在默认情况下，`rackup` 在端口 9292 启动服务器，而不是端口 3000，可以选 `-p` 参数指定端口。通过浏览器访问页面，呈现出简洁的货品列表，如图 25.1 所示。

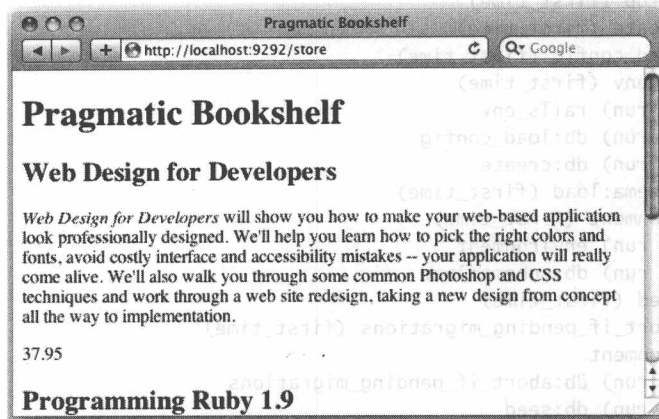


图 25.1 很简单但切实可用的货品列表

与 Rails 应用程序相比，原生 Rack 应用程序的缺点在于为其本身考虑得太少了。而主要的优点是这样有可能避免一些 Rails 开销，因此在单位时间里可以处理更多的请求。

大多数情况下，并不打算创建完整的独立应用程序，而想绕过 Rails 的控制器处理网站的一部分。这可以通过定义路由实现：

```
depot_u/config/routes.rb
```

```
► require './app/store'
Depot::Application.routes.draw do
► match 'store' => StoreApp.new
  get 'admin' => 'admin#index'
  controller :sessions do
    get 'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end
  scope '(:locale)' do
```

```
resources :users
resources :orders do
  resources :line_items
end
```

服务器不是 Rails 组件唯一发挥作用的地方。下面将阐述用来协调任务执行的工具。

25.5 自动执行任务工具 Rake

无可置疑, Rake 是一款经常使用的程序。它用来自动执行任务, 特别是那些有许多包依赖关系的任务。在应用程序根目录里, 可以找到定义这些任务的 Rakefile 文件。

db:setup 就是这样一个任务命令的示例。如果要看其涉及的子任务命令, 运行命令 Rake, 同时加上 -trace 和 -dry-run 参数:

```
$ rake --trace --dry-run db:setup
(in /home/rubys/work/depot)
** Invoke db:setup (first_time)
** Invoke db:create (first_time)
** Invoke db:load_config (first_time)
** Invoke rails_env (first_time)
** Execute (dry run) rails_env
** Execute (dry run) db:load_config
** Execute (dry run) db:create
** Invoke db:schema:load (first_time)
** Invoke environment (first_time)
** Execute (dry run) environment
** Execute (dry run) db:schema:load
** Invoke db:seed (first_time)
** Invoke db:abort_if_pending_migrations (first_time)
** Invoke environment
** Execute (dry run) db:abort_if_pending_migrations
** Execute (dry run) db:seed
** Execute (dry run) db:setup
```

对于可重复的部署, 按照正确的顺序执行正确的步骤至关重要, 这就是为什么在 16.1 节中曾使用这个特殊任务。

运行命令 rake -tasks 后, 可以看到一个有效的任务清单。Rails 提供的这个任务仅仅只是一个启动设置, 你可以自由创建更多的任务。只要在 lib/tasks 路径下, 创建新的 Ruby 代码文件即可。

下面是备份生产环境数据库的例子:

```
depot_u/lib/tasks/db_backup.rake
```

```
namespace :db do
```

```
  desc "Backup the production database"
```

```
  task :backup => :environment do
```

```
    backup_dir = ENV['DIR'] || File.join(Rails.root, 'db', 'backup')
```

```
    source = File.join(Rails.root, 'db', 'production.db')
```

```
    dest = File.join(backup_dir, "production.backup")
```



```
makedirs backup_dir
sh "sqlite3 #{source} .dump > #{dest}"
end
```

```
end
```

第 1 行代码包含命名空间 (namespace)。把备份任务放到 db 命名空间里。

第 2 行代码是一种描述。当命令行执行列出任务的时候, 该描述会显示出来。如果再次运行命令 `rails --tasks`, 会看到刚写的任务包含在 Rails 所提供的任务当中。

下一行代码包含着任务以及该任务可能的包依赖关系。依赖的 environment 与加载 rails console 所提供的大致上是相同的。

传递给任务的代码块是标准的 Ruby 代码。在这个例子里, 先确定源代码目录和目标目录 (目标默认设定为 db/backup, 但是可以在命令行里用参数 DIR 覆盖它), 然后处理要备份的目录 (假如有需要的话), 最后执行命令 `sqlite3 dump`。

25.6 Rails 包依赖关系揭秘

一个很好的出发点是回顾 Gemfile.lock 文件。你会发现有些名字很奇怪, 有的就很易懂。为了进一步深入学习, 下面列出常用名称的简单描述。

当然, 随着 Rails 的不断发展, 这份清单也将不可避免地发生变化。但是知晓了这些组件的名称, 就为进一步理解建立了一个良好的开端。找到更多名称的好办法是访问 RubyGems.org 网站[Ⓔ], 输入想查找的 gem 名字, 然后要么单击文档 (documentation) 链接, 要么单击主页 (homepage) 链接。

arel

关系代数, 由 Active Record 模块使用。

actionmailer

Rails 组成部分, 参见第 13 章。

actionpack

Rails 组成部分, 参见第 20 章。

activemodel

对 Active Record 模块和 Active Resource 模块提供模型化的支持。

activerecord

Rails 组成部分, 参见第 19 章。

activeresource

Rails 组成部分, 参见 24.3 节。

activesupport

Rails 组成部分, 参见 24.2 节。

Ⓔ <http://rubygems.org>

rails

整个框架的容器。

railties

Rails 组成部分, 参见 26.5 节的后段, 更多细节参考 RailsPlugins.org 网站, 本书 26.5 节中还有更多的关于这个主题的网络链接。

abstract

可用 Ruby 语言自定义抽象方法的库。

builder

创建 XML markup 的简洁方式, 参见 25.1 节。

capistrano

易于部署的工具, 参考 16.2 节。

Erubis

在 Rails 里用于实现 ERb 功能, 参考 25.2 节。

highline

命令行界面的输入输出 I/O 库。

i18n

提供国际化支持, 参考第 15 章。

mail

提供邮件功能, 参考第 13 章。

mime-types

依据扩展名确定文件类型, 用于邮件方面。

mysql

由 Active Record 模块所支持的产品数据库, 参考 16.1 节。

net-scp

文件安全拷贝。

net-sftp

文件安全传输。

net-ssh

安全连接远程服务器。

net-ssh-gateway

通过 SSH 隧道连接。

polyglot

自定义语言装载器。

rack

Rails 和网络服务器之间的界面, 参考 25.4 节。

rack-cache

为 rack 提供 HTTP 缓存。

rack-cache-purge

清除 Rack-cache 的缓存内容。

rack-mount

Rack 的路由功能, 参考 20.1 节。

rack-test

测试路由的 API 接口, 参考 20.1 节。

rake

自动化任务工具, 参考 25.5 节。

sqlite3

Active Record 模块支持的用于开发环境的数据库。

thor

脚本框架, 用于 rails 命令。

treetop

文字解析库, 可以用于处理邮件。

tzinfo

提供时区支持。

will_paginate

查询结果分页, 参考 12.3 节。

25.7 本章小结

我们探索了少量的 Rails 包依赖关系, 并展示了如何管理、实现与网络服务器集成以及从命令行界面进行整合这些包依赖关系。顺便找出了在应用程序根目录下的 Rakefile、Gemfile 和 Gemfile.lock 文件所起的作用。

到目前为止, 探寻之旅已经深入到了 Rails 框架内部, 下一站将开阔眼界, 学习在安装 Rails 的时候就开始起作用的外部插件, 依靠外部插件来扩展 Rails 框架。

```
gem 'will_paginate', '~> 3.0.0'
gem 'activerecord', '~> 3.0.0'
gem 'haml', '~> 3.0.0'
gem 'jquery-rails', '~> 2.0.0'
```

第 26 章

Rails 插件

在本章中，我们将学习：

- 添加新类到应用程序
- 添加 Rake 任务和视图帮助程序
- 添加新的模板语言
- 添加生成器且替代 JavaScript 库

本书一开始就不断地强调 Rails 的约定优于配置原则。在该原则下，Rails 为几乎所有事情准备了切合实际的默认约定。在前面的内容里，还深入讨论了基础软件包 gem（安装 Rails 时已经默认安装了）。现在我们把上面这两个部分整合起来，以揭示 Rails 提供的一组初始 gem 本身就是一组切合实际的默认插件——它们既是可添加的也是可改变的。

伴随 Rails 3.0 版本发布，gem 成了添加新功能的主要渠道。因此为了避免抽象的泛泛描述，这里选择了几个插件并实际使用它们，来展示如何安装它们以及它们能够做什么。事实上，在日常工作中这些插件的大多数都能产生立竿见影的效果。这简直就是意外收获。

下面从介绍简单的但的确可以赚钱的插件开始。

26.1 信用卡业务处理插件 Active Merchant

回顾 12.1 节临时搁置了的“信用卡处理”程序。能够真正向顾客收款显然是订单的重要组成部分。尽管这不是 Rails 的内置核心功能，但是 gem 库中存在这种功能的插件。

上面已经看到了应用程序是如何控制载入哪些 gem 的，其做法是编辑文件 Gemfile。既然本章涉及若干个这种 gem，索性把所有马上就要用到的插件一次性添加进来。实际上可以把这些 gem 添加到任何你愿意的地方。接下来立即在下面的代码里添加分页插件 will_paginate 及其依赖插件。

depot_v/Gemfile

```
gem 'will_paginate', '>= 3.0.pre'
▶ gem 'activemerchant', '~> 1.10.0'
▶ gem 'haml', '~> 3.0.18'
▶ gem 'jquery-rails', '~> 0.2.2'
```

你可能已经注意到了，这里遵循了最佳实践方法：指定了最小版本号，及其更高的版本号，这样挑选版本号的目的是不想在范例中产生不兼容的情况。

至于所添加的 gem，后面每小节将分别详细介绍。这一节将主要关注插件 Active Merchant（活跃的商家）^①。

^① <http://www.activemerchant.org/>

一旦修改了文件 Gemfile，通过命令 `bundle install` 就可以安装这些 gem 插件及其所有依赖的 gem 插件。另外，如果 Rails 服务器在运行中，就需要重新启动，让 Rails 识别出这些变化。在这一节中还没有但后面马上就会用到服务器。还要确保该服务器正在运行 Depot 应用程序。

为了演示这一功能，将创建一个短小的脚本程序，把它放在目录 script 下：

```
depot_v/script/creditcard.rb

credit_card = ActiveMerchant::Billing::CreditCard.new(
  :number      => '4111111111111111',
  :month       => '8',
  :year        => '2009',
  :first_name  => 'Tobias',
  :last_name   => 'Luetke',
  :verification_value => '123'
)

puts "Is #{credit_card.number} valid? #{credit_card.valid?}"
```

这段程序代码不多：先创建 `ActiveMerchant::Billing::CreditCard` 类的实例，然后调用该方法 `valid?`（判断信用卡是否有效）。下面运行一下这个程序。

```
$ rails runner script/creditcard.rb
Is 4111111111111111 valid? false
```

代码不多，但已经可以工作了。注意，没有必要使用 `require` 语句；只要在文件 Gemfile 中列出想要的 gem，就可以在应用程序里应用其功能了。

现在，你应该已经看见了在 Depot 应用程序里如何使用这个功能。你也明白，如何通过迁移技术添加字段到表 Orders 中；如何添加该字段到视图；如何添加验证逻辑（它调用刚才所用到的 `valid?` 方法）到模型。如果浏览该插件网站，甚至可能发现如何授权（方法 `authorize`）和付款（方法 `capture`）。当然，这需要有支付网关的登录用户名和密码。一旦设置完成，就知道如何从控制器调用这一逻辑了。

只在 Gemfile 文件里添加一行代码，一切就皆有可能。

正如本章开头所讲，添加 gem 到文件 Gemfile 里，是 Rails 3.0 扩展的首选方法。这种做法的好处很多：所有的依赖关系都可以用 Bundler 追踪管理；所有预装载的插件都可以直接为应用程序所使用；而且为了方便部署还可以打包。

到本书截稿为止，关于 ActiveMerchant 网站上安装指导尚未更新：针对不同的 Rails 版本，该网站还介绍了三种其他的安装此功能的方法。无论什么时候，都应该尽可能使用这里所介绍的方法。

下一节将介绍一种扩展，它还没有打包成 gem 软件包。

26.2 节约带宽的插件 Asset Packager

在 Depot 应用程序内部存在许多 CSS 文件。这些文件并不是一种紧凑的格式，而是以利于开发和维护的方式编写。但是，如果网站流量很大，合理的做法是：让更少的请求提供相同内容，并让那些请求包含更少字节。

插件 Asset Packager (资源打包器) plugin[Ⓐ]可以提供这个功能。与插件 Active Merchant 不同的是, 插件 Asset Packager 到目前为止还没有打包成 gem 软件包。[Ⓐ]非 gem 形式的软件包插件与 gem 形式的软件包插件相比, 其主要缺点体现在无法跟踪依赖关系; 无法确保插件与之前安装的其他插件版本兼容; 无法简单地检查其版本变化。

因为除了与 Rails 本身之外, 此功能没有其他依赖关系, 所以不需要依赖关系跟踪。也像插件 Active Merchant 那样, Asset Packager 并不是独立于 Rails 的; 实际上它扩展了 Rails 的功能, 这是现在介绍它的原因。

为了安装插件 Asset Packager, 可以使用命令 `rails plugin install`。

```
$ rails plugin install git://github.com/sbecker/asset_packager.git
```

这样就可以把文件安装到目录 `vendor/plugins` 中。把这些文件提交到 Git 代码库, 并使用 Capistrano 部署。

插件 Asset Packager 需要做的事情就是添加可用的新 Rake 任务。前面已经了解到如何列出 Rake 任务, 就是使用选项 `-T`:

```
$ rake -T asset
(in /home/rubys/work/depot)
rake asset:packager:build_all # Merge and compress assets
rake asset:packager:create_yaml # Generate asset_packages.yml from existing assets
rake asset:packager:delete_all # Delete all asset builds
```

现在要做的第一件事情是创建 YAML 文件。虽然可以自己建立它, 但还是可以利用现有的任务来完成它:

```
$ rake asset:packager:create_yaml
(in /home/rubys/work/depot)
config/asset_packages.yml example file created!
Please reorder files under 'base' so dependencies are loaded in correct order.
```

观察所生成的该文件:

```
depot_v/config/asset_packages.yml
---
javascripts:
- base:
  - jquery
  - jquery-ui
  - application
  - rails
stylesheets:
- base:
  - scaffold
  - depot
```

Ⓐ 这里不过是为了介绍插件的不同安装方法, 其实 github 里的 Ruby 开源项目 gem 软件包基本上都可以在网站 rubygems.org 上找到, 比如 http://rubygems.org/gems/rails_asset_packager, 需要注意的是, 项目名与 gem 名可能有差别, 见 http://rubygems.org/gems/asset_packager。是否适合 Rails 3, 需要参考 26.6 节所提到的网站 <http://railsplugins.org/>。——译者注

Ⓐ http://synthesis.sbecker.net/pages/asset_packager

可以看见这里的基本思路很简单：列出所有要打包的样式表和脚本，并按次序放入合并文件里。在许多情况下，次序并不是问题，且该插件提供的初始次序已经能很好地工作。这样就可进行下一个任务：打包。

```
$ rake asset:packager:build_all
(in /home/rubys/work/depot)
Created /home/rubys/work/depot/public/javascripts/base_packaged.js
Created /home/rubys/work/depot/public/stylesheets/base_packaged.css
```

现在检查一下打包过的样式表，发现合并后的整个 `base_packaged.css` 文件比单个 `depot.css` 文件还要小。可把所生成的样式表（和脚本）提交到 Git，或者在先前提及的 `asset_packager` 网站上找到关于如何在部署阶段创建 Capistrano 部署脚本的指南。

现在已经有了更有效的 CSS 文件，但页面布局还在继续使用原来的文件。当然，这很容易改过来：

```
depot_v/app/views/layouts/application.html.erb
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  ▶ <%= raw stylesheet_link_merged :base %>
  ▶ <%= raw javascript_include_merged :base %>
  <%= csrf_meta_tag %>
</head>
<!-- ... -->
```

这里替换了两行代码，一行是把调用方法 `stylesheet_link_merged` 代替调用方法 `stylesheet_link_tag`，另一行把调用方法 `javascript_include_merged` 替换调用方法 `javascript_include_tag`。

因为本章讨论的是插件在通常情况下能做什么事情，而不是关注于某个特定的插件偶然的行为了，所以需要特别指出的是，方法 `stylesheet_link_merged` 和 `javascript_link_merged` 是两个视图帮助程序，它们是由该插件所安装的。

到目前为止已经讨论了一些简单的添加方式：一个新的类、少量新的任务以及几个视图帮助方法。下面就做更重要的事情：提供一个代替 Rails 需要依赖的 gem 插件。

26.3 用 Haml 美化标记语言

下面再看一下在 Depot 里所使用的简单视图，这里以展示店面为例：

```
depot_u/app/views/store/index.html.erb
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>
<h1><%= t('.title_html') %></h1>

<% @products.each do |product| %>
  <div class="entry">
```

```
<%= image_tag(product.image_url) %>
<h3><%= product.title %></h3>
<%= sanitize(product.description) %>
<div class="price_line">
  <span class="price"><%= number_to_currency(product.price) %></span>
  <%= button_to t('.add_html'), line_items_path(:product_id => product),
    :remote => true %>
</div>
</div>
<% end %>
```

这段代码可以正常显示商店的页面。它包含了基本的 HTML 语句，在 `<%` 和 `>%` 标记中间散布了 Ruby 代码片段。在标记里面，等号用来标示要转成 HTML 并显示出来的表达式的值。

这个办法不仅能解决手边的问题，同时对于大多数 Rails 应用程序来说，已经完全可以满足需要了。另外，很多书都假定读者具备一定的 HTML 知识——本书亦如此。很多读者对 Rails 甚至 Ruby 都只是刚刚接触到。如果是这种情况，那他还是应该学习新的语言。

现在假设你过了那段学习曲线，就让我们向前再探寻一门新的语言——用 Ruby 的代码更紧凑地集成生成标记语言：HTML 抽象标记语言（HTML Abstraction Markup Language, **Haml**）。

让我们先删除刚才的那个文件：

```
$ rm app/views/store/index.html.erb
```

在删除文件的位置创建一个新文件：

```
depot_v/app/views/store/index.html.haml
```

```
- if notice
  %p#notice= notice

%h1= t('.title_html')

- @products.each do |product|
  .entry
    = image_tag(product.image_url)
    %h3= product.title
    = sanitize(product.description)
    .price_line
      %span.price= number_to_currency(product.price)
      = button_to t('.add_html'), line_items_path(:product_id => product),
        :remote => true
```

注意新后缀 `.html.haml`。它表示是 **Haml** 模板而不是 **ERb** 模板。

首先注意到文件变得相当小了。下面就快速概览代码所起到的作用，分析每行的第一个字符：

- 破折号（-）表示不产生任何结果的 Ruby 语句。
- 百分号（%）表示 HTML 元素。
- 等号（=）表示 Ruby 表达式，会生成要显示的结果。这既可以是数行 Ruby 语句，也可以紧接着多个 HTML 元素。
- 点（.）和井号（#）字符可以分别用于定义 HTML 语言的 `class` 和 `id` 属性。可以与百分号结合或单独使用。当单独使用时，代表 `div` 元素。

- 逗号 (,) 出现在包含表达式的行结尾, 意味着一种延续。在上面的代码示例里, `button_to` 调用延续跨两行。

Haml 缩进至关重要。返回到相同层级的缩进, 就关闭当前处于开放状态的条件语句、循环或标签。在上面示例里, 在 `h1` 之前关闭段落, 而在第一个 `div` 之前关闭 `h1`, 但 `div` 元素是嵌套结构, 其中包含的第一部分是 `h3` 元素, 第二部分是 `span` 和 `button_to`。

正如上面已看到的那样, 所有熟悉的帮助程序都是可以使用的, 如: 方法 `t`、方法 `image_tag` 和方法 `button_to`。跟 ERb 一样, 可以用各种有意义的方式把 Haml 集成到应用程序中。可采用混合和搭配方式: 一部分采用 ERb 模板, 而另一部分用 Haml。

由于已经安装了 `gem` 软件包 Haml, 所以已经不需要再做什么了。要看到效果, 只要访问在线商店。所看到的内容应该如图 26.1 所示。

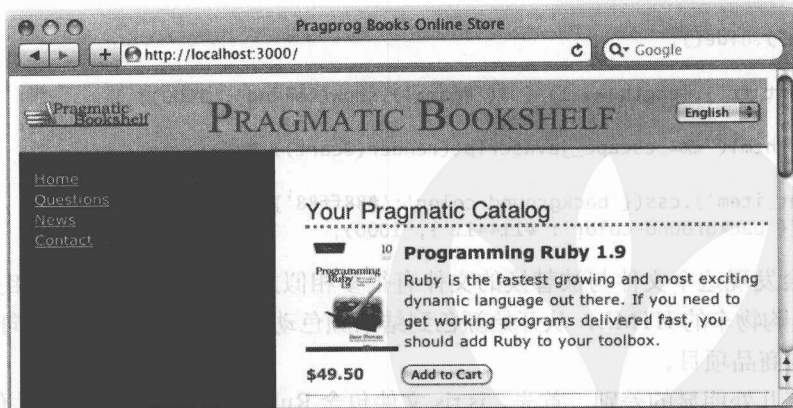


图 26.1 用 Haml 显示的商店页面

似乎看起来没有什么特别之处, 那是因为设计本应该与以前“一模一样”。如果你仔细思考一下前面对程序的修改就能明白, 应用程序布局继续使用 ERb 模板, 而索引页面本身用的还是 Haml 模板。这已经达到了不寻常的境界。尽管如此, 所有一切都无缝隙地且毫不费力地集成在一起了。

虽然这种集成明显比简单地添加任务或帮助程序更深一个层次, 但是实质上仍然是一种补充措施。接下来将探索真正的替代方案。

26.4 用 JQuery 少写多做

早期 Rails 版本使用早期的 JavaScript 库。当时很少有人对此有这样或那样的偏好, 因此 Rails 简单地选择了一个不错而有效的库。近来, 越来越多的人有了自己的偏爱, 其中一种流行趋势就是用 JQuery 替代 Prototype (原型)。

现在来看看在 11.2 节中创建的 RJS 文件:

```
depot_u/app/views/line_items/create.js.rjs
```

```
page.select("#notice").each { |notice| notice.hide }
```

```
page.replace_html('cart', render(@cart))
```

```
page[:cart].visual_effect :blind_down if @cart.total_items == 1
```

```
page[:current_item].visual_effect :highlight,
```

```
:startcolor => "#88ff88",
```

```
:endcolor => "#114411"
```

就像之前把 ERb 文件替换成 Haml 那样，现在再大胆地把 RJS 文件也搬走：

```
$ rm app/views/line_items/create.js.rjs
```

并且用下面的文件代替：

```
depot_v/app/views/line_items/create.js.erb
```

```
$("#notice").hide();
```

```
if ($('#cart tr').length == 1) { $('#cart').show('blind', 1000); }
```

```
$('#cart').html("<%= escape_javascript(render(@cart)) %>");
```

```
$('#current_item').css({'background-color': '#88ff88'}).
```

```
animate({'background-color': '#114411'}, 1000);
```

仔细观察会发现这个文件与被替换的文件有很多相似之处。四个语句都有相互对应：隐藏 #notice，替换购物车的 HTML，从开始颜色到结束颜色动态地更换背景颜色，向下展开（blind down）当前的商品项目。

当然还是有几处明显的差别。首先，js.rjs 文件包含 Ruby 语句，而 js.erb 文件使用 JavaScript 语言编写。更重要的是，RJS 文件在服务器端执行，访问数据库模型，而 JQuery 文件中的一切（含在 <% 和 %> 标记内的异常处理代码）都在客户端执行，访问浏览器的文档对象模型（Document Object Model, DOM）。这种差别就是因为语句顺序出现了小小的变化：在呈现之前，完成了购物车长度的检查。注意：之所以使用数量等于 1，是因为在显示页面里存在 HTML 表头的行。这对用 DOM 模型解决问题的人来讲非常重要。

还有一处需要修改，这是因为布局中还有一行代码使用了其他 Prototype 方法：

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

```
<!-- ... -->
```

```
<div id="banner">
```

```
<%= form_tag store_path, :class => 'locale' do %>
```

```
<%= select_tag 'set_locale',
```

```
options_for_select(LANGUAGES, I18n.locale.to_s),
```

```
:onchange => 'this.form.submit()' %>
```

```
<%= submit_tag 'submit' %>
```

```
▶ <%= javascript_tag "$(' .locale input').hide()" %>
```

```
<% end %>
```

```

<%= image_tag("logo.png") %>
<%= @page_title || t('title') %>
</div>
<!-- ... -->

```

与 Haml 那种单纯的附加不同，这里还需要完成一个额外的步骤，那就是下载 JQuery 库并移除现有不需要的 prototype（原型）库。为实现这个目标，jquery-rails 安装了 Rails 生成器，该生成器负责处理所有必需的步骤。要做的也就是调用如下的命令：

```

rails generate jquery:install --ui --force
remove public/javascripts/controls.js
remove public/javascripts/dragdrop.js
remove public/javascripts/effects.js
remove public/javascripts/prototype.js
fetching jQuery (1.4.3)
create public/javascripts/jquery.js
create public/javascripts/jquery.min.js
fetching jQuery UI (latest 1.x release)
create public/javascripts/jquery-ui.js
create public/javascripts/jquery-ui.min.js
fetching jQuery UJS adapter (github HEAD)
force public/javascripts/rails.js

```

参数 `-ui` 用于获得 jquery-ui 库，选项 `-force` 用于替代 rails.js 文件。

完成了以上工作，一切就绪。运行应用程序，并没有看出有什么差别。假如觉得上面的做法平淡无奇，可以看一下 tablesorter 库（表排序）[⊖]。到这个网站下载一个脚本程序然后加上几行 JavaScript 代码，只要单击一下表头，就可以实现表栏项目排序功能。

实际上还有最后一点差别，当运行功能测试时就会发现：

```

Started
.....F.....
Finished in 1.49875 seconds.
1) Failure:
test_should_create_line_item_via_ajax(LineItemsControllerTest)
[/test/functional/line_items_controller_test.rb:65]:
No RJS statement that replaces or inserts HTML content.
46 tests, 76 assertions, 1 failures, 0 errors

```

显然，“No RJS statements”（没有 RJS 语句）的消息是真实的，因为没有在视图里使用任何 RJS。解决的方法也很简单。把方法 `assert_select_rjs` 换成调用方法 `assert_select_jquery`，适当地调整调用的参数：

```

depot_v/test/functional/line_items_controller_test.rb

test "should create line item via ajax" do
  assert_difference('LineItem.count') do
    xhr :post, :create, :product_id => products(:ruby).id
  end
  assert_response :success

```

⊖ <http://tablesorter.com/docs/>

```

▶ assert_select_jquery :html, '#cart' do
  assert_select 'tr#current_item td', /Programming Ruby 1.9/
end
end

```

对于 JQuery 感兴趣的读者，可参考其他资料，但是本章的目的是介绍用插件能做什么事情，实际上插件几乎可以做任何事情。在这里，JQuery 插件完全替代了 Rails 默认安装的组件。

26.5 在 RailsPlugins.org 上找出更多

到此为止，已经探讨了四个插件。从 RailsPlugins.org^①网站上可以找到数百个插件，目前该网站收集了超过 500 多个 gem，其中大多数（不是所有）都与 Rails 3 兼容。

下面是按类别列出的一些插件，可以深入了解：

- 插件实现了一些之前在 Rails 核心中自带但现已去掉的功能。例如，过去分页就是 Rails 的核心功能，后续版本把它移到插件里，称为 `classic_pagination`^②。从那以后几乎所有分页功能都由 12.3 节介绍的 `will_paginate` 插件所替换。有一些功能迅速发展成了插件，如 `acts_as_tree`^③。还有就是，为了有助于迁移，一些插件，像 `rails_xss`^④，只保留最基本的功能，以便与未来的 Rails 版本兼容。
- 实际上，有些插件实现了常见的应用程序逻辑甚至用户界面的重要功能。插件 `devise`^⑤ 和 `authlogic`^⑥ 实现了用户认证和会话管理功能。虽然在 Depot 里自己实现了这些功能，但通常不推荐这么做。如果别人已经写好了你需要的功能插件，你不用那么费力直接使用就好了，这能节约更多的时间来开发自己的应用程序。
- 有些插件替代了 Rails 大部件。如，用 `datamapper`^⑦ 替代 `ActiveRecord`。可以单独或组合使用 `cucumber`^⑧、`rspec`^⑨ 和 `webrat`^⑩ 插件，以便替代测试脚本（与纯文本测试故事、规格说明书和浏览器模拟一起）。
- 插件 `hoptoad_notifier`^⑪ 和 `exception_notification`^⑫ 监控已部署服务器的异常行为。

当然，这仅是可用插件的一小部分。而这一列表在不断增加：毫无疑问，当你读到这里的时候，又有更多可用的插件出现了。

最后，很显然你也可以创建自己的插件。尽管本书并没有介绍怎么做，但是可以在与 Rails

① <http://www.railsplugins.org/plugins>

② https://github.com/masterkain/classic_pagination#readme

③ https://github.com/rails/acts_as_tree#readme

④ https://github.com/rails/rails_xss

⑤ <https://github.com/plataformatec/devise#readme>

⑥ <https://github.com/binarylogic/authlogic#readme>

⑦ <http://datamapper.org/>

⑧ <http://cukes.info/>

⑨ <http://rspec.info/>

⑩ <https://github.com/brynary/webrat#readme>

⑪ <http://www.hoptoadapp.com/pages/home>

⑫ https://github.com/rails/exception_notification#readme

Guides^①和 Rails 相关的文档^②里面找更多的资料。

26.6 本章小结

严格说来,本章仅仅介绍了一条新的 Rails 命令: rails plugin install。在许多情况下, bundler 和 Gemfile 已经包括了该命令的功能。虽然本章的确包含了一些插件,但目的并非要涵盖任何某个插件的内部细节,只是介绍了一些插件所提供的功能。

如果把前面章节介绍的 gem 也一并纳入进来理解的话,插件就是:可以让你很简单地添加新功能(Active Merchant 和 Capistrano)、添加 Rake 任务(Asset Packager)、添加视图帮助程序(Asset Packager 也起这样的作用)、给模型对象添加方法(will_paginate)、添加新的语言模板(Haml)、提供新的数据库接口(mysql),甚至替换整个客户端脚本编程框架(jquery-rails)。

如果仔细想一想,那么插件真的没有什么做不到的。

① <http://guides.rubyonrails.org/plugins.html>

② <http://api.rubyonrails.org/classes/Rails/Railtie.html>

第 27 章

整装进发

在本章中，我们将学习：

- 回顾 Rails 概念：模型、视图、控制器、配置、测试和部署

- 进一步学习链接

恭喜大家已经完成了基础知识的学习。

第一部分首先安装了 Rails，然后通过简单的应用程序验证了安装过程，介绍了 Rails 框架的体系结构，认识了（或者说重新认识了）Ruby 语言的知识。

第二部分开始用迭代方式建立了应用程序，然后依据敏捷开发模式搭建了测试案例，最终用 Capistrano 完成部署。这个应用程序的设计目的就是让开发者接触到 Rails 应知应会的各个方面。

鉴于第一部分和第二部分单独的写作目的，本书第三部分就担负有双重的目的。

对于一些读者而言，第三部分有条不紊地补充缺少的知识并涵盖足以应付真实环境的日常工作。对于其他的读者而言，这正是漫漫长路的第一步。

对于大部分读者而言，真实的价值体现在上面的两个方面。为了让读者有能力去进一步学习和探索，坚实的基础是必需的。这正是我们用了整个一章的内容来叙述的原因：不仅包括了 Rails 的惯例与配置，而且还包含了文档生成。

接着对模型、视图和控制器各用一章的篇幅进行介绍，它们是 Rails 架构的基石。之后涉及从数据库关联关系到 REST 架构，再到表单 HTML 和帮助程序（helper）等话题。

再者，讨论了在部署应用程序时需要关注的、与基础维护和调优有关的迁移和缓存相关内容。

最后，掰开 Rails，从不同的角度考察了 gem 概念：从分开使用 Rails 单独组件到完整利用 Rails 的基础功能（Rails 本身就搭建在其上，并最终构建和扩展成适应实际需要的框架）。

因此，拥有贯通上下文环境的基础和背景，就会在面对任何复杂的需求或者棘手问题的时候，有能力在更深层面上找出解决实际应用的恰当途径。推荐访问 [rubyonrails 网站](http://rubyonrails.org)^①浏览网页上方的每个链接。其中会经常更新本书所提到的材料，还可以找到大量有关报告问题、深入学习并保持与时俱进的信息。

另外，请继续参与本书所提到的 wiki 和论坛网站。

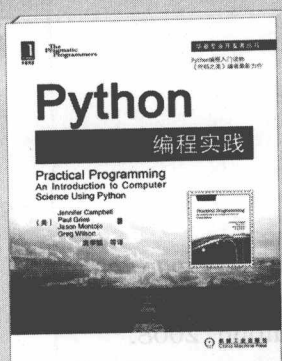
希望你愉快地学习 Ruby on Rails，正如我们以快乐心情写完这本书。

① <http://rubyonrails.org/>

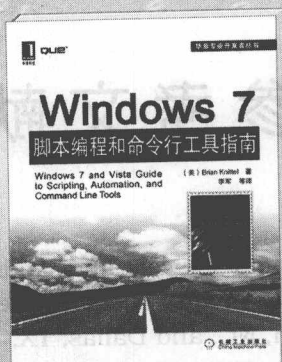
参考文献

- [TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.
- [ZT08] Ezra Zygmuntowicz and Bruce Tate. *Deploying Rails Applications: A Step-by-Step Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2008.

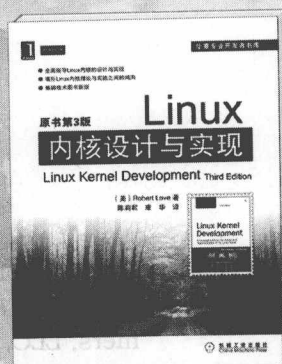
华章专业开发者书库



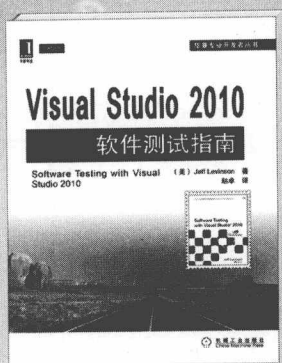
Python编程实践
作者: Jennifer Campbell 等
译者: 唐学韬 等
ISBN: 978-7-111-36478-8
定价: 49.00元



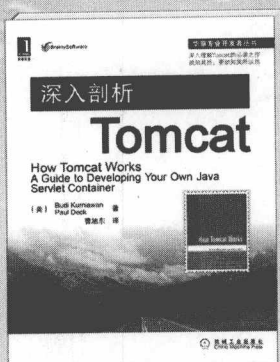
Windows 7脚本编程和命令行工具指南
作者: Brian Knittel
译者: 李军 等
ISBN: 978-7-111-35677-6
定价: 79.00元



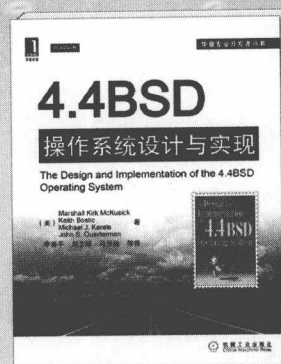
Linux内核设计与实现 (原书第3版)
作者: Robert Love
译者: 陈莉君 康华
ISBN: 978-7-111-33829-1
定价: 69.00元



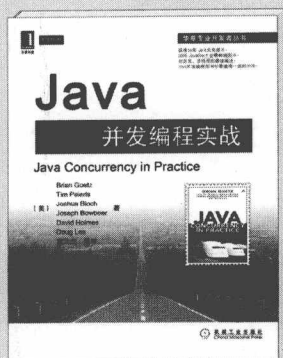
Visual Studio 2010软件测试指南
作者: Jeff Levinson
译者: 赵卓
ISBN: 978-7-111-35931-9
定价: 49.00元



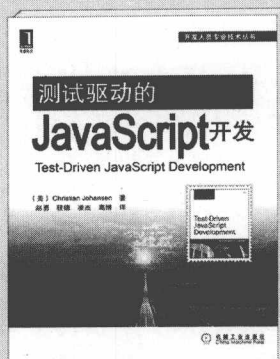
深入剖析 Tomcat
作者: Budi Kurniawan 等
译者: 曹旭东
ISBN: 978-7-111-36997-4
定价: 59.00元



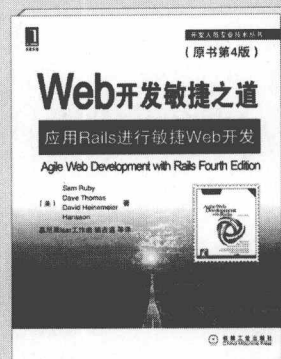
4.4BSD操作系统设计与实现
作者: Marshall Kirk McKusick 等
译者: 李善平 等
ISBN: 978-7-111-36647-8
定价: 79.00元



Java并发编程实战
作者: Brian Goetz 等
译者: 童云兰 等
即将出版



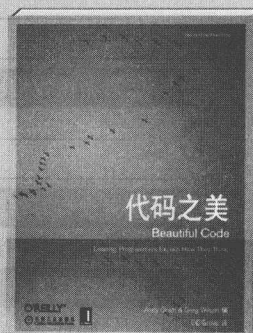
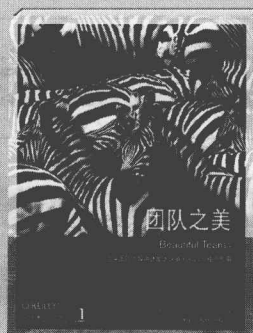
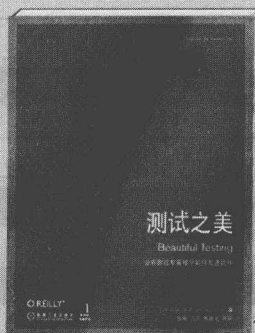
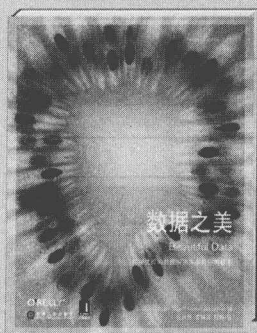
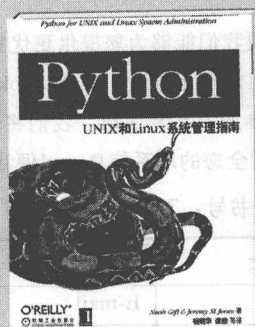
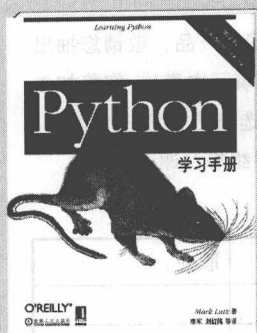
测试驱动的JavaScript开发
作者: Christian Johansen
译者: 赵勇 等
即将出版



Web开发敏捷之道
作者: Sam Ruby 等
译者: 骆古道 等
即将出版

为每一个团队提供最优价值的阅读服务 每一本书都值得您和您的团队一起阅读

分享阅读 分享成功





专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会
获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名： 书号： 7-111-()

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐ 朋友推荐 ☐ 书店 ☐ 图书目录 ☐ 杂志、报纸、网络等 ☐ 其他

2. 您从哪里购买本书：

☐ 新华书店 ☐ 计算机专业书店 ☐ 网上书店 ☐ 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

☐ 是，我的计划是_____ ☐ 否

7. 您希望获取图书信息的形式：

☐ 邮件 ☐ 信函 ☐ 短信 ☐ 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收
邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com

Web开发敏捷之道 | Agile Web Development 应用Rails进行敏捷Web开发(原书第4版) | with Rails Fourth Edition

在Ruby on Rails的帮助下,你可以快速创建出美观且高质量的网站,而且只需要专注于创建应用程序本身,Rails会完成所有细节部分的实现。

成千上万的开发者通过这本屡获殊荣的书来学习Rails。它由Rails核心团队极力推荐,是一本广泛、深远的教程和参考书。如果你是一位Rails初学者,本书会提供入门级指导;如果你是一位经验丰富的开发者,本书同样会提供全面、丰富的Rails信息。

作者简介



Sam Ruby 是一位卓越的软件开发者,他是W3C HTML工作组的负责人之一,并在Apache软件基金会的许多开源软件项目中作出了积极的贡献。他还是IBM新兴技术集团的一位高级技术人员。



Dave Thomas 是“敏捷宣言”的作者之一,所以他了解敏捷性;因为他是《Programming Ruby》一书的作者,所以他又了解Ruby;又因为他是一位活跃的Rails开发者,所以他了解Rails。



David Heinemeier Hansson 是Rails框架的创建者。

客服热线:(010) 88378991, 88361066
购书热线:(010) 68326294, 88379649, 68995259
投稿热线:(010) 88379604
读者信箱:hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计 范华明



上架指导: 计算机/程序设计

ISBN 978-7-111-37404-6



9 787111 374046

定价: 59.00元

[G e n e r a l I n f o r m a t i o n]

书名 = 书 S 1 3 2 6

作者 =

页数 = 5 0 0

S S 号 = 1 3 2 6

出版日期 =