第 15 章

Web 开发：Ruby on Rails

Ruby on Rails 是无可非议的 Ruby 杀手锏应用程序。它提供很多保障将 Ruby 从其日文本土的隐晦中解脱出来。没有其他任何一门编程语言可以以这样一个简单的Web应用程序框架而自豪，该框架也吸取了大部分该语言开发者的思想（注 1）。本章论证重要的基础Rails 使用原理（类似 15.6 节中），给出普通 Web 应用程序模式的 Rails 实现（15.4 节和 15.8 节）并且讨论如何在 Rails 内部使用标准的 Ruby 工具（15.22 节和 15.23 节）。

不谈其质量和流行程度，Rails 并没有为 Web 开发带来其他新鲜的东西。其基础在于标准的编程模式，比如 ActiveRecord 以及 Model-View-Controller。它重用很多已经存在的Ruby 库（比如 Rake 和 ERb）。Rails 的功能在于集成这些标准的技术自动化完成任务，并且会断言合理的默认行为。

如果 Rails 有秘密的话，那就在于命名惯例。最为广泛使用的 Web 应用程序是 CRUD 应用程序：从数据库创建、阅读、更新并且删除信息。在这些类型的应用程序中，Rails 的功劳很是夺目。可能一开始只有一个数据库计划而没有任何代码，但是 Rails 可以将很多段代码通过命名惯例和快捷方式联系起来。这些使得可以十分快速地创建应用程序。

因为这么多的设定和命名可以显式地从其他信息片断里继承，Rails 比其他框架需要更少的“ 文书工作 ”。数据隐式包含在代码里，或者数据库方案并不需要在其他地方指定。本系统的必需部分是针对复数名词的 ActiveSupport 系统（15.7 节）。

在命名惯例不起作用的地方，Rails 使用 decorator 方法来声明对象间的关系，而不是在膨胀的 XML 配置文件里。结果是更小，更易于理解，并且更灵活的应用程序。

注 1： 比如 Python，有多个杰出的 Web 应用程序框架，但是这也正是问题所在。它有多个，但在某种特定情况下究竟选择哪一个。Ruby 除了 Rails 外没有其他主要的 Web 应用程序框架。所以说，Ruby 以前的模糊性使得 Rails 的统治成为可能。

正如上面所说，Rails 架构在普通 Ruby 库之上，这些库在本书的很多地方都有所涉及，它们包括 ActiveRecord（13 章的大部分，尤其在 13.11 节中）、ActionMailer（14.5 节）、ERb（1.3 节）、Rake（19 章）和 Test::Unit（17.7 节）。其中某些出现在 Rails 之前，某些是为 Rails 编写却在外部无法使用的。反过来也是正确的：因为 Rails 应用程序可以被用来实现很多目的，几乎本章的每一节在 Rails 程序中都很有用。

Rails 可用作 `rails gem`，它包含库以及 `rails` 的命令行程序。这就是运行来创建 Rails 应用程序的程序。当调用该程序（比如，用 `rails mywebapp`）时，Rails 为 Web 应用程序产生一个目录结构，通过 WEBrick 测试服务器以及单元测试框架完成。当使用 `script/generate` 脚本来跳过应用程序的创建时，Rails 会用更多文件组装该目录结构。这些脚本生成的代码很小，并且等同于启动项目时大部分 IDE 生成的代码。

Rails 的架构是流行的 Model-View-Controller 架构。这就将 Web 应用程序分割成 3 个可预先命名的部分。本章会详细讲述，但是先有一个引导性的介绍。

模型（*model*）是应用程序使用的数据形式的代表。通常是一组 Ruby 类，`ActiveRecord::Base` 的子类，每个都对应于应用程序数据库的一个表。本章第一个重要的模型将在 15.6 节重点介绍。要生成某个特定数据库表的模型，通过表名调用 `script/generate` 模型，比如：

```
$ script/generate model users
```

这会创建一个名为 *app/models/users.rb* 的文件，它定义了 `User` ActiveRecord 类以及单元测试该模块的基本结构。并不会创建实际的数据库表。

控制器（*controller*）是 Ruby 的一个类（`ActionController::Base` 的子类），它的方法在模块上定义了操作。每个操作当作 controller 的方法被定义。

要生成一个控制器，用控制器的名称以及要做的操作来调用 `script/generate controller`：

```
$ script/generate controller user add delete login logout
```

该命令创建文件 `app/controllers/user_controller.rb`，它定义了一个类 `UserController`。该类定义了 4 个 stub 方法：`add`、`delete`、`login` 以及 `logout`，每个对应于终端用户可以在重要的 User 模型上执行的操作。它也创建了功能单元测试控制器的模板。

控制器在本章的首节讨论（15.1 节）。

视图（*view*）是应用程序的用户界面。它包含在一组ERb模板中，保存在*.rhtml*文件里。更重要的是，对于每个控制器的操作通常都会有一个*.rhtml*文件：这是针对该特殊操作的Web界面。上述创建了`UserController`类的同一条命令也创建了*app/views/user/*下的4个文件：*add.rhtml*、*delete.rhtml*、*login.rhtml*以及*logout.rhtml*。这是由于`UserController`类，这些被当作stub文件启动。我们的工作是自定义它们来表示应用程序的接口。

就像控制器，视图在本章首节所展现：15.1节，而15.3节、15.5节以及15.14节讲述如何自定义属于自己的视图。

这样的分节不是随意的。如果限制改变数据库为模型的代码，那么就将容易进行单元测试该代码并且审查其安全问题。通过移动所有处理代码到控制器中，可以将用户界面的显示与其内部的工作分离开。这样做的最为明显的好处在于可以拥有一个UI设计器来更改视图模板，而无需占用很多Ruby代码。

要想学习Model-View-Controller如何工作的话，参考15.2节，该节讲述控制器和视图之间的关系，15.16节则将三者都集成起来。

如下有更多可以参考的资源来帮助我们快速地学习Rails：

- 本书的姐妹篇，*Rails Cookbook* by Rob Orsini (O'Reilly)，更详细地讨论了Rails问题，正如Chad Fowler（注重实效的编程者）在Rails节所述
- Dave Thomas、David Hansson、Leon Breedt、Mike Clark、Thomas Fuchs以及Andrea Schwarz（注重实效的编程者）所著*Agile Web Development with Rails*是Rails程序员的标准参考书
- Rails Web站点*http://www.rubyonrails.com/*上的Ruby，尤其是RDoc文档（*http://api.rubyonrails.org/*）以及wiki（*http://wiki.rubyonrails.com/*）

15.1 编写简单的Rails应用程序显示系统状态

问题

想通过创建很简单的应用程序来开始Rails的学习。

解决方案

该例显示了Unix系统上运行着的进程。如果开发在Windows上进行，可以用其他命令代替（比如`dir`的输出）或者只是使得应用程序输出静态消息。

首先，确保安装了`rails gem`。

要创建Rails应用程序，运行`rails`命令并且传递进入应用程序的名称。应用程序名为“status”。

```
$ rails status
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
...
```

Rails应用程序至少需要两部分：一个控制器和一个视图。控制器能够得到系统信息，视图则可以显示它。

能够产生控制器以及用generate脚本生成相应的视图。如下调用定义了控制器和视图，它们实现名为`index`的单一操作。这就是应用程序的主视图（也是唯一的）。

```
$ cd status
$ ./script/generate controller status index
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/status
      exists  test/functional/
      create  app/controllers/status_controller.rb
      create  test/functional/status_controller_test.rb
      create  app/helpers/status_helper.rb
      create  app/views/status/index.rhtml
```

生成的控制器在Ruby源文件`app/controllers/status_controller.rb`里。该文件定义了`StatusController`类，实现`index`操作，就像名为`index`的空白方法一样。填满`index`方法，从而可以显示想要在视图里使用的对象：

```
class StatusController < ApplicationController
  def index
    # This variable won't be accessible to the view, since it is local
    # to this method
    time = Time.now

    # These variables will be accessible in the view, since they are
    # instance variables of the StatusController.
    @time = time
    @ps = `ps aux`
  end
end
```

生成的视图在pp/views/status/index.rhtml里。它由一段静态HTML开始，改变它为ERb模板，使用StatusController#index设置的实例变量：

```
<h1>Processes running at <%= @time %></h1>
<pre><%= @ps %></pre>
```

现在应用程序完整了。要想运行它，用如下命令启动Rails服务器：

```
$ ./script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
...
```

可以通过访问*http://localhost:3000/status/*查看该应用程序。

当然，不需要将该应用程序发布到外部世界，因为这样可能会将系统信息提供给攻击者。

讨论

首先，需要注意的与Rails应用程序有关的事情是不要为每个URL创建单独的代码。Rails使用一种架构，在该架构中，控制器（Ruby源文件）以及视图（.rhtml文件里的ERb模板）组合起来服务于一系列操作。每个操作处理站点上的部分URL。

考虑类似*http://www.example.com/hello/world*的URL。为了在Rails应用程序里服务该URL，需要创建hello 控制器并且给它一个称为world的操作。

```
$ ./script/generate controller hello world
```

控制器类需要有一个world方法，*views/hello*目录则需要一个包含视图的*world.rhtml*文件。

```
class HelloController < ApplicationController
  def world
  end
end
```

访问*http://www.example.com/hello/world*会调用HelloController#world方法，翻译*world.rhtml*模板，得到一些HTML输出，并且将这些输出送至客户端。

控制器的默认操作是index，就好比静态Web服务器目录里的默认页面是*index.html*。因此访问*http://www.example.com/hello/*和访问*http://www.example.com/hello/index/*会得到同样的效果。

如上所述，视图文件仅仅是Rails服务的最终页面的主要部分。它不是完整的HTML页面，不可以在其中放入<html>或者<body>标签（参考15.3节）。因为视图文件是ERB模板，也就不可以在视图里调用puts或者print。ERB的介绍见1.3节，但是很值得在Rails应用程序上下文里进行扫描。

要想插入Ruby表达式的值到ERB模板里，使用<%= %>指示。如下是hello操作的可能存在的*world.rhtml*视图：

```
<p>Several increasingly silly ways of displaying "Hello world!":</p>

<p><%= "Hello world!" %></p>
<p><%= "Hello" + "world!" %></p>
<p><%= w = "world"
       "Hello #{w}!" %></p>
<p><%= 'H' + ?e.chr + ('l' * 2) %><%=('o word!').gsub('d', 'ld')%></p>
```

最后这个例子是额外的，但是它证明了一点。没有必要在视图模板里放入这么多的Ruby代码（可能必须进入控制器，或者需要以类似PHP的代码结束），但是如果需要这么做的话也是可以实现的。

在ERb指示里等号意味着将会打印出输出。如果想要执行某个没有输出的命令，忽略等号并且使用<% %>指示。

```
<% hello = "Hello" %>
<% world = "world!" %>
<%= hello %> <%= world %>
```

视图和控制器只是基于一些从Ruby代码里得到的数据（比如当前时间以及ps aux的输出）。但是大多数真实世界的视图和控制器是基于模型的：一组数据库表，包含被视图显示并且被控制器操作的数据。这就是著名的“Model-View-Controller”架构，并且这无疑是Rails独一无二的。

参考

- 1.3节“将变量代入现有的字符串”有更多ERB相关信息
- 15.3节“创建页眉和页脚的布局”

15.2 从控制器传递数据到视图

问题

想要在控制器和视图之间传递数据。

解决方案

视图是ERB模板，该模板在其控制器对象的上下文中得到解释。视图不会调用任何控制器的方法，但是它可以访问控制器的实例变量。传递数据到视图，设置控制器的一个实例变量。

这是一个`NovelController`类，被放至`app/controllers/novel_controller.rb`。可以通过运行`script/generate controller novel index`为其生成stub。

```
class NovelController < ApplicationController
  def index
    @title = 'Shattered View: A Novel on Rails'
    one_plus_one = 1 + 1
    increment_counter one_plus_one
  end

  def helper_method
    @help_message = "I see you've come to me for help."
  end

  private

  def increment_counter(by)
    @counter ||= 0
    @counter += by
  end
end
```

因为这是`Novel`控制器和`index`操作，相应的视图在`app/views/novel/index.rhtml`。

```
<h1><%= @title %></h1>

<p>I looked up, but saw only the number <%= @counter %>.</p>

<p>"What are you doing here?" I asked sharply. "Was it <%=
@counter.succ %> who sent you?"</p>
```

该视图在`NovelController#index`运行后被解释。下面说明视图可以和不可以访问的地方：

- 可以访问实例变量`@title`和`@counter`，因为它们在`NovelController#index`运行结束时被定义在`NovelController`对象上。
- 可以调用实例变量`@title`和`@counter`的实例方法。
- 不可以访问实例变量`@help_message`，因为改变量是由`helper_method`方法定义的，而该方法永远也不会被调用。
- 不可以访问变量`one_plus_one`，因为这不是一个实例变量：它对于`index`方法来说是本地的。

- 即使其运行在NovelController的上下文中，也不可以调用NovelController的任何方法，不管是helper_method或者是set_another_variable，都不可以。而且不可以再次调用 index。

讨论

控制器的操作方法负责创建和存储（在实例变量里）所有视图工作涉及的对象。这些变量可能很简单，如字符串，也可能是很复杂的helper类。每一种形式下，大部分的应用程序逻辑都必须在控制器里。这在类似数据结构迭代的视图里没有什么问题，但是大部分工作会发生在控制器，或者某个在实例变量间发布的对象里。

Rails为每个请求实例化新的NovelController对象。这意味着无法通过将其放入控制器实例变量来在请求间维持数据。不管重新加载多少次页面，@counter变量都不会超过2。每次调用 increment_counter 时，都是在一个全新的NovelController对象里实现的。

类似任何Ruby类，Rails控制器可以定义类变量和常数，但是它们对于视图而言不可用。考虑一个类似如下的NovelController：

```
class NovelController < ApplicationController
  @@numbers = [1, 2, 3]
  TITLE = 'Revenge of the Counting Numbers'
end
```

@@numbers和TITLE在这个控制器的任何视图里都不可以被访问。它们只能够被控制器方法调用。

然而，在控制器上下文之外定义的常数是可以被每个视图访问的。当想在某个容易改变的位置声明Web 站点名字时，这很是有用。config/environment.rb 文件是定义这些常数的好地方：

```
# config/environment.rb
AUTHOR = 'Lucas Carlson'
...
```

在基于对象的编程中使用全局变量总是一个坏主意。但是 Ruby 的确使用了它们，一个全局变量一旦定义后在所有的视图里都可用。它们是全局可用的，而不用管它们是否定义在某个操作、控制器的作用域里或是任何作用域外。

```
$one = 1
class NovelController < ApplicationController
  $two = 2
  def sequel
```

```
    $three = 3
  end
end
```

如下是视图，`sequel.rhtml`，它们使用 3 个全局变量：

```
Here they come, the counting numbers, <%= $one %>, <%= $two %>, <%= $three
%>.
```

15.3 创建页眉和页脚的布局

问题

想要为Web应用程序上的每一页创建页眉和页脚。某页必须有特定的页眉页脚，想要动态决定对于给定请求该采用哪种页眉页脚。

解决方案

多数Web应用程序要求定义页眉页脚文件，并且自动包含这些文件到每一页的顶端和底部。Rails转化了这种模式。被调用的单一文件包含页眉和页脚，并且每个特殊页的内容被插入进这些文件中。

在Web应用程序中应用每一页的布局，创建名为*app/views/layouts/application.rhtml*的文件。类似如下：

```
<html>
  <head>
    <title>My Website</title>
 </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

任何布局文件里的关键信息是指示`<%= content_for_layout %>`。这会被每个单独页面的内容所替代。

可以为每个控制器创建自定义的布局，通过在*app/views/layouts*文件夹里独立创建文件来实现。比如，*app/views/layouts/status.rhtml*是`status`控制器、`StatusController`的布局。`PriceController`的布局文件是*price.rhtml*。

重载 site-wide layout 来自定义布局，它们并没有加上这个。

讨论

正如主视图模板一样，布局模板已经访问了所有操作设定的实例变量。任何可以在视图里做的事情，都可以在布局模板里完成。这意味着可以完成这样的事情，比如在操作中动态设定页面标题，然后在布局里使用它们：

```
class StatusController < ActionController:Base
  def index
    @title = "System Status"
  end
end
```

现在`application.rhtml`文件可以访问`@title`，类似如下：

```
<html>
  <head>
    <title>My Website - <%= @title %></title>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

`application.rhtml`并不是仅仅针对Rails应用程序的控制器而发生在默认的布局模板里。它这样发生是因为每个控制器都是从`ApplicationController`继承而来。默认来说，布局的名字是从控制器类名得到的。因此`ApplicationController`转变为`application.rhtml`。如果有一个名为`MyFunkyController`的控制器，那么布局的默认文件名会是`app/views/layouts/my_funky.rhtml`。如果这个文件并不存在，Rails会查找对应于`MyFunkyController`父类的布局，并且会在`app/views/layouts/application.rhtml`里找到。

要想改变一个控制器的布局文件，调用其布局方法：

```
class FooController < ActionController:Base
  # Force the layout for /foo to be app/views/layouts/bar.rhtml,
  # not app/view/layouts/foo.rhtml.
  layout 'bar'
end
```

如果在某次操作中使用`render`方法（参考15.5节），可以传递`:layout`变量到`render`里，并且给此次操作一个不同于其他控制器的布局。在本例中，`FooController`的大多数操作使用`bar.rhtml`作为它们的布局，但是`count`操作使用的是`count.rhtml`：

```
class FooController < ActionController:Base
  layout 'bar'

  def count
```

```
    @data = [1,2,3]
    render :layout => 'count'
  end
end
```

甚至可以得到没有布局的操作。本段代码赋给 FooController 的所有操作一个 bar.html 的布局, 除去count操作, 它根本没有任何布局 :它负责自己所有的HTML。

```
class FooController < ActionController:Base
  layout 'bar', :except => 'count'
end
```

如果需要动态计算布局文件, 传递一个方法符号到布局方法中。这告诉布局在每次请求时都调用该方法。该方法的返回值定义了布局文件。方法可以调用action_name来决定当前请求的操作名字。

```
class FooController < ActionController:Base
  layout :figure_out_layout

  private

  def figure_out_layout
    if action_name =~ /pretty/
      'pretty'          # use pretty.rhtml for the layout
    else
      'standard'        # use standard.rhtml
    end
  end
end
```

最后，布局接受 lambda 函数作为一个参数。这使得可以用更少的代码动态决定布局：

```
class FooController < ActionController:Base
  layout lambda { |controller| controller.logged_in? ? 'user' : 'guest' }
end
```

对于程序员和设计者来说，使用布局文件取代单独的页眉和页脚，这都是很好的事情：更加容易查看这个图片。但是如果需要使用显式的页眉页脚，那么也可以。创建名为app/views/layouts/_header.rhtml和app/views/layouts/_footer.rhtml的文件。这里的下划线表明它们是 “ 局部变量 ”（参考 15.14 节）。要使用它们，设置操作使之不使用任何布局，并且在视图文件里编写下述代码：

```
<%= render :partial => 'layouts/header' %>
... your view's content goes here ...
<%= render :partial => 'layouts/footer' %>
```

参考

- 15.5 节“ 用 render 显示模板 ”
- 15.14 节“ 重构视图为视图的部分片断 ”

15.4 重新定位不同的位置

问题

想重新定位用户到应用程序的另一个操作中，或者到某个外部 URL。

解决方案

`ActionController::Base`类（`ApplicationController`的父类）定义了名为`redirect_to`的方法，它实现HTTP 重新定位。要重新定位到另一个站点，可以将其当作一个字符串传递到一个URL里。要重新定位到应用程序里的不同操作，传递给它一个指定的控制器、操作和 ID 的散列。

这是一个`BureaucracyController`，在不同的操作间来回搅乱进入的请求，最终发送客户端到一个外部站点：

```
class BureaucracyController < ApplicationController
  def index
    redirect_to :controller => 'bureaucracy', :action =>
'reservation_window'
  end

  def reservation_window
    redirect_to :action => 'claim_your_form', :id => 123
  end

  def claim_your_form
    redirect_to :action => 'fill_out_your_form', :id => params[:id]
  end

  def fill_out_your_form
    redirect_to :action => 'form_processing'
  end

  def form_processing
    redirect_to "http://www.dmv.org/"
  end
end
```

如果运行 Rails 服务器并且在浏览器上单击 *http://localhost:3000/bureaucracy/*，会最终定位到 *http://www.dmv.org/*。Rails 服务器记录会显示链接到那里的 HTTP 请求：

```
"GET /bureaucracy HTTP/1.1" 302
"GET /bureaucracy/reservation_window HTTP/1.1" 302
"GET /bureaucracy/claim_your_form/123 HTTP/1.1" 302
"GET /bureaucracy/fill_out_your_form/123 HTTP/1.1" 302
"GET /bureaucracy/form_processing HTTP/1.1" 302
```

不需要为每个操作创建视图模板，因为 HTTP 重新定位的主体不会被 Web 浏览器显示。

讨论

`redirect_to` 方法使用简练的默认方式。如果给它一个没有指定控制器的散列，它会假定用户想要在同一个控制器里移动至另一个操作。如果不考虑操作，会假定用户涉及的是 `index` 操作。

由“解决方案”里所示简单的重新定位可见，可能会认为 `redirect_to` 的确中止了操作方法，并且执行了一次立即的 HTTP 重新定位。这是不对的。操作方法会继续运行直到其结束，或者该调用返回。`redirect_to` 方法无法实现一次重新定位：它告诉 Rails 一旦操作方法结束运行的话，那么实现一次重新定位。

如下是该问题的解释。可能会认为如下 `redirect_to` 的调用会阻止 `do_something_dangerous` 方法的调用。

```
class DangerController < ApplicationController
  def index
    redirect_to (:action => 'safety') unless params[:i_like_danger]
    do_something_dangerous
  end

  # ...
end
```

实际上不会。唯一能够在运行中中止一次操作方法的方式是调用 `return`（注 2）。真正需要做的是：

```
class DangerController < ApplicationController
  def index
    redirect_to (:action => 'safety') and return unless
params[:i_like_danger]
    do_something_dangerous
  end
```

注 2： 可以抛出异常，但是当重新定位没有发生时，用户会看到一个异常屏幕。

```
  end
```

注意redirect_to结尾的and return。在告诉Rails重新定位用户到另一页面后，又要执行代码，这种情况通常很少见。为了避免出现问题，形成添加的习惯，并且在redirect_to或者render调用结束后返回。

参考

- ApplicationController::Base#redirect_to和ApplicationController::Base#url_for方法生成的RDoc

15.5 用render显示模板

问题

Rails的操作方法与视图模板之间的默认映射对于用户而言灵活性不够。想要自定义一个模板，该模板通过直接调用Rails的渲染代码来实现特定操作的渲染实现。

解决方案

渲染发生在ActionController::Base#render方法里。Rails的默认操作是在操作方法运行后调用render，将此方法映射到相应的视图模板。Foo操作映射到foo.rhtml模板。可以在操作方法内调用render来使得Rails渲染不同的模板。这个控制器定义了两种操作，都是用shopping_list.rhtml模板来实现渲染的：

```
class ListController < ApplicationController
  def index
    @list = ['papaya', 'polio vaccine']
    render :action => 'shopping_list'
  end

  def shopping_list
    @list = ['cotton balls', 'amino acids', 'pie']
  end
end
```

默认来说，render假定用户在与控制器通话，并且在操作render被调用时已经在运行。如果调用无参数的render，Rails会以正常方式处理。但是要指定'shopping_list'作为视图重载默认方式，使得index操作使用shopping_list.rhtml模板，就像shopping_list操作一样。

讨论

尽管使用的是相同的模板，访问index操作和访问shopping_list操作却并不相同。它们显示不同的列表，因为index定义的列表和shopping_list不同。

回顾15.4节，redirect方法没有实现立即的HTTP重新定位。它告诉Rails一旦当前操作方法运行结束，那么实现一次重新定位。相似地，render方法并不立即实现渲染。它只是告诉Rails在操作完成后，渲染哪一个模板。

考虑如下例子：

```
class ListController < ApplicationController
  def index
    render :action => 'shopping_list'
    @budget = 87.50
  end

  def shopping_list
    @list = ['lizard food', 'baking soda']
  end
end
```

可能会认为调用index设置@list而不是@budget。实际上，反过来才是正确的。调用index设置@budget而不是@list。

@budget变量被设置是因为render不会停止当前操作的执行。调用render更像是将某条信息装进信封，供Rails在今后某个时间点打开阅读。仍然可以自由设置实例变量，并且调用其他方法。一旦操作方法返回了，Rails会打开信封并且使用其中包含的渲染策略。

@list变量不被设置，因为render调用不会调用shopping_list操作。它只是实现现有的操作，index，使用shopping_list.rhtml模板而不是index.rhtml模板。这里甚至不需要shopping_list操作：只需要一个名为shopping_list.rhtml的模板。

如果的确想要从另一个操作中调用某个操作，可以显式调用该操作方法。代码会使得index设置@budget和@list：

```
class ListController < ApplicationController
  def index
    shopping_list and render :action => 'shopping_list'
    @budget = 87.50
  end
end
```

这样的“封装”行为的另一个结果是在一次单一的客户端请求里永远不可以调用render两次（这对于很类似render的redirect_to也是一样的，redirect_to也是在信封里封装了某条信息）。

如果编写如下代码，Rails会罢工。用户给了它两个封装了的信封，但它不知道该打开哪一个：

```
class ListController < ApplicationController
  def plain_and_fancy
    render :action => 'plain_list'
    render :action => 'fancy_list'
  end
end
```

但是下述代码是正确的，因为任何给定请求只会触发if/else语句的某一个分支。不管发生了什么，render在每次请求里只会被调用一次。

```
class ListController < ApplicationController
  def plain_or_fancy
    if params[:fancy]
      render :action => 'fancy_list'
    else
      render :action => 'plain_list'
    end
  end
end
```

使用redirect_to时，如果要强制操作方法停止运行，可以在调用render后，立刻加入一个return语句。本代码没有设置@budget变量，因为执行永远不会越过return语句：

```
class ListController < ApplicationController
  def index
    render :action => 'shopping_list' and return
    @budget = 87.50         # This line won't be run.
  end
end
```

参考

- 15.4节“重新定位不同的位置”
- 15.14节“重构视图为视图的部分片断”给出了在视图模板里调用render的示例

15.6 集成数据库到 Rails 应用程序中

问题

想要 Web 应用程序在一个相关数据库里保存持久数据。

解决方案

最困难的部分是设置所有东西：创建数据库并且使得 Rails 与之建立联系。一旦这些完成了，数据库的访问就像编写 Ruby 代码一样简单。

告诉Rails如何访问数据库, 打开应用程序的`config/database.yml`文件。假定Rails应用程序叫做 `mywebapp`，看上去会类似如下：

```
development:
adapter: mysql
database: mywebapp_development
host: localhost
username: root
password:

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

production:
adapter: mysql
database: mywebapp
host: localhost
username: root
password:
```

现在，只需确保`development`部分包含有效的用户名和密码，这是指针对用户数据库不同类型的正确适配器的名称（查看第 13 章的列表）。

现在创建一个数据库表。如果遵照如下惯例，Rails会自动完成很多数据库工作。如果有必要，可以重载这些惯例，但是现在更简单的方法是很好地使用它们。

表的名称必须是复数形式的名词：比如，“ people ”、“ tasks ”、“ items ”。

表必须包含一个称为 `id` 的自动递增的主键字段。

如在以下的例子里，使用数据库工具或者 `CREATE DATABASE` SQL 命令来创建 `mywebapp_development` 数据库（如果需要帮助，查看第 13 章的介绍章节）。然后在此数

据库里创建名为people的表。如下SQL在MySQL里创建people表，可以为数据库改编它。

```
use mywebapp_development;

DROP TABLE IF EXISTS 'people';
CREATE TABLE 'people' (
'id' INT(11) NOT NULL AUTO_INCREMENT,
'name' VARCHAR(255),
'email' VARCHAR(255),
PRIMARY KEY (id)
) ENGINE=InnoDB;
```

现在回到命令行，改变到Web应用程序的根目录，并且键入`./script/generate model Person`。这会生成一个Ruby类，它知道如何操作people表。

```
$ ./script/generate model Person
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/person.rb
create test/unit/person_test.rb
create test/fixtures/people.yml
```

注意模型被命名为Person，即使表被命名为people。如果遵守这样的惯例，Rails会自动处理这些复数形式的名词（查看15.7节）。

Web应用程序现在已经通过Person类访问了people表。再次通过命令行，运行如以下命令：

```
$ ./script/runner 'Person.create(:name => "John Doe", \
:email => "john@doe.com")'
```

这段代码在people表里创建一个新的记录项。（如果已经阅读过13.11节，会知道这是ActiveRecord代码。）

要想在应用程序里访问peoson，创建一个新的控制器和视图：

```
$ ./script/generate controller people list
exists app/controllers/
exists app/helpers/
create app/views/people
exists test/functional/
create app/controllers/people_controller.rb
create test/functional/people_controller_test.rb
create app/helpers/people_helper.rb
create app/views/people/list.rhtml
```

编辑*app/view/people/list.rhtml*，看上去如下：

```
<!-- list.rhtml -->
<ul>
<% Person.find(:all).each do |person| %>
<li>Name: <%= person.name %>, Email: <%= person.email %
></li>
<% end %>
</ul>
```

启动 Rails 服务器，访问 *http://localhost:3000/people/list/*，会看到排列好的 John Doe。

`Person` 模型类可以在 Rails 应用程序的所有部分对其访问：控制器、视图、helper 以及 mailer。

讨论

到现在为止，本节创建的应用程序只是用到控制器和视图（注 3）。`Person` 类，以及其重要的数据库表，给出了 Model-View-Controller 三角关系中的 Model 部分的首次示例。

相关数据库通常是存储真实世界模型的最佳选择，但是很难直接编程相关数据库。Rails 使用 ActiveRecord 库隐藏 `people` 表到 `Person` 类之后。类似 `Person.find` 的方法使得可以不用编写 SQL 就能实现 `person` 数据库表的搜索。结果会自动转变为 `Person` 对象。ActiveRecord 的基础在 13.11 节里有所介绍。

`Person.find` 方法有很多可选的参数。如果给其传递一个整数，会查询唯一 ID 是整数的 person 记录项，并且返回一个合适的 `Person` 对象。`:all` 和 `:first` 符号从表（`Person` 对象数组）里抓取所有的记录项，或者仅仅是匹配的第一个 person。可以通过指定 `:limit` 或者 `:order` 来限制或者排序数据库项，甚至可以通过 `:conditions` 设置原始 SQL 条件。

如下介绍如何在包含邮件地址的 `people` 表里找到前 5 个记录项。结果会是包含 5 个 `Person` 对象的列表，以它们的名字字段排序。

```
Person.find(:all,
     :limit => 5,
     :order => 'name',
     :conditions => 'email IS NOT NULL')
```

`config/database.yml` 的 3 个不同的部分指定 Rails 应用程序不同的时间里使用的 3 个不同的数据库：

注 3： 更确切地说，我们的模型嵌入在 controller 里，就像添加 hoc 数据结构比如很难编写的 shopping 列表。

开发数据库
: 运行应用程序时使用的数据库。通常填充的是测试数据。

测试数据库
: 应用程序运行测试时，单元测试框架使用的数据库。其数据是单元测试框架自动组装的。

产品数据库
: Web 站点运行动态数据时使用的数据库模式。

除非显式设置Rails运行在产品或者测试模式里，否则默认为开发模式。因此要想启动，仅仅需要确保`database.yml`的开发部分的设置正确。

参考

- 第 13 章
- 13.11 节“ 用 ActiveRecord 使用对象关系映射 ”
- 13.13 节“ 以编程方式构建查询 ”
- 13.14 节“ 用 ActiveRecord 确认数据 ”
- ActiveRecord 并不能完成 SQL 可以完成的任何事情。对于复杂的数据库操作，需要使用`DBI`或者绑定至特定类型数据库的一种 Ruby。这些话题在 13.15 节“ 阻止 SQL 注入攻击 ”里都已有所介绍，该节给出了关于`database.yml`文件格式的更多细节

15.7 理解复数规则

问题

想要理解和自定义 Rails 的自动复数化名词的规则。

解决方案

可以在应用程序的任何部分使用Rails的复数化功能，但是ActiveRecord是Rails自动复数化的唯一主要部分。ActiveRecord通常期望表名是复数名词形式，对应的模式类是同一个名词的单数形式。

因此当创建一个模型类时，通常需要使用单数名称。Rails 自动将其复数化：

- 模型的相应表名
- has_many 关系
- has_and_belongs_to_many 关系

比如，如果创建一个 LineItem 模型，表名自动变为 line_items。也要注意表名全变为小写，并且原始名词现在被下划线分割开。

如果接着创建一个 Order 模型，相应的表需要被命名为 orders。如果想要描述有很多行项目的排序，代码如下所示：

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

正如参考的表名，has_many 关系里使用的符号被复数化并被下划线分隔开。对于表间的其他关系也有类似的机制，比如 has_and_belongs_to_many 关系。

讨论

ActiveRecord 复数化这些名称，使得代码读起来更像是英文句子：has_many :line_items 可以被读成“ has many line items ”。如果复数化会造成困惑，可以通过设定 ActiveRecord::Base.pluralize_table_names 为 false 来禁止该功能。在 Rails 里，完成这个的最简单方法就是将如下代码加入到 config/environment.rb: 里：

```
Rails::Initializer.run do |config|
  config.active_record.pluralize_table_names = false
end
```

如果应用程序知道一些特定的单词是 ActiveRecord 不知道如何复数化的，则可以通过操作 Inflector 类定义属于自己的复数化规则。设定“ foo ”的复数形式是“ fooze ”，构建应用程序来管理 fooze。在 Rails 里，可以通过添加如下代码到 config/environment.rb 来指定这样的转换：

```
Inflector.inflections do |inflect|
  inflect.plural /^(foo)$/i, '\1ze'
  inflect.singular /^(foo)ze/i, '\1'
end
```

在这种情况下，更简单的方法是使用 irregular 方法：

```
Inflector.inflections do |inflect|
  inflect.irregular 'foo', 'fooze'
end
```

如果有些名词不能够被改变（通常因为它们是聚集名词，或者因为它们的复数形式和其单数形式相同），那么可以将其传递给 `uncountable` 方法：

```
Inflector.inflections do |inflect|
  inflect.uncountable ['status', 'furniture', 'fish', 'sheep']
end
```

`Inflector` 类是 `activesupport` gem 的一部分，可以在 ActiveRecord 或 Rails 的外部使用它，并将其作为复数化英文单词的通用方式。如下是标准的 Ruby 程序：

```
require 'rubygems'
require 'active_support/core_ext'

'blob'.pluralize                                    # => "blobs"
'child'.pluralize                                   # => "children"
'octopus'.pluralize                                 # => "octopi"
'octopi'.singularize                                # => "octopus"
'people'.singularize                                # => "person"

'goose'.pluralize                                   # => "gooses"
Inflector.inflections { |i| i.irregular 'goose', 'geese' }
'goose'.pluralize                                   # => "geese"

'moose'.pluralize                                   # => "mooses"
Inflector.inflections { |i| i.uncountable 'moose' }
'moose'.pluralize                                   # => "moose"
```

参考

- 13.11 节“用 ActiveRecord 使用对象关系映射”

15.8 创建登录系统

问题

想使得应用程序支持基于用户账户的登录系统。用户使用唯一的用户名和密码登录，就像大多数商业和社区 Web 站点使用的一样。

解决方案

创建一个包含非空用户名和密码字段的 `users` 表。创建该表的 SQL 看上去就像如下的 MySQL 示例：

```
use mywebapp_development;
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
```

```
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(255) NOT NULL,
  `password` VARCHAR(40) NOT NULL,
  PRIMARY KEY (`id`)
);
```

进入应用程序的主目录并且生成对应于该表的 User 模型：

```
$ ./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
```

打开生成的文件 *app/models/user.rb* 并且编辑它使之看上去类似：

```
class User < ActiveRecord::Base
  validates_uniqueness_of :username
  validates_confirmation_of :password, :on => :create
  validates_length_of :password, :within => 5..40

  # If a user matching the credentials is found, returns the User object.
  # If no matching user is found, returns nil.
  def self.authenticate(user_info)
    find_by_username_and_password(user_info[:username],
                                  user_info[:password])
  end
end
```

现在已经得到一个代表用户账户的 User 类，一种基于数据库存储项目来确认用户名和密码的方式。

讨论

“解决方案”里给出的简单 User 模型定义了一种完成用户名 / 密码确认的方法，以及一些强加限制到存储数据上的确认规则。这些确认规则告诉 User：

- 保证每个用户名是唯一的。两个不同的用户不能使用相同的用户名。
- 保证不管设置的是什么样的密码属性，password_confirmation 属性都具有相同的值。
- 保证密码属性值介于 5～40 字符长度之间

现在来为该模型创建一个控制器。它有一个显示登录页面的 login 操作，一个检查用户名和密码的 process_login 操作，以及一个鉴别登录会话的 logout 操作。因为用户账户会实际完成某些事情，也需要添加一个 my_account 操作：

```
$ ./script/generate controller user login process_login logout my_account
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/user
      exists  test/functional/
      create  app/controllers/user_controller.rb
      create  test/functional/user_controller_test.rb
      create  app/helpers/user_helper.rb
      create  app/views/user/login.rhtml
      create  app/views/user/process_login.rhtml
      create  app/views/user/logout.rhtml
```

编辑 *app/controllers/user_controller.rb* 来定义 3 个操作：

```
class UserController < ApplicationController
  def login
    @user = User.new
    @user.username = params[:username]
  end

  def process_login
    if user = User.authenticate(params[:user])
      session[:id] = user.id # Remember the user's id during this session
      redirect_to session[:return_to] || '/'
    else
      flash[:error] = 'Invalid login.'
      redirect_to :action => 'login', :username =>
params[:user][:username]
    end
  end

  def logout
    reset_session
    flash[:message] = 'Logged out.'
    redirect_to :action => 'login'
  end

  def my_account
  end
end
```

现在针对视图。`process_login`和`logout`操作只是重新定位到其他操作, 因此只需针对`login`和`my_account`的视图。如下是针对`login`的视图：

```
<!-- app/views/user/login.rhtml -->
<% if @flash[:message] %><div><%= @flash[:message] %></div><% end %>
<% if @flash[:error] %><div><%= @flash[:error] %></div><% end %>

<%= form_tag :action => 'process_login' %>
 Username: <%= text_field "user", "username" %>&#x00A;
  Password: <%= password_field "user", "password" %>&#x00A;
  <%= submit_tag %>
<%= end_form_tag %>
```

@flash实例变量是一个类散列的对象,用于为操作间的用户存储临时消息。当logout操作设置flash[:message]以及login的重新定位，或者process_login设置flash[:error]以及login的重新定位时，结果对于login操作的视图是可用的。然后它们被清除。如下是my_account的简单视图：

```
<!-- app/views/user/my_account.rhtml -->
<h1>Account Info</h1>

<p>Your username is <%= User.find(session[:id]).username %>
```

在users表里创建记录项，启动服务器，会发现可以从*http://localhost:3000/user/login*登录，从*http://localhost:3000/user/my_account*查看账户信息。

```
$ ./script/runner 'User.create(:username => "johndoe", \
                               :password => "changeme")'
```

这里还有一个遗漏的部分：即使没有登录也可以访问my_account操作。没有方法可以关闭未经鉴定的用户的操作。添加如下代码到*app/controllers/application.rb*文件：

```
class ApplicationController < ActionController::Base
  before_filter :set_user

protected
  def set_user
    @user = User.find(session[:id]) if @user.nil? && session[:id]
  end

  def login_required
    return true if @user
    access_denied
    return false
  end

  def access_denied
    session[:return_to] = request.request_uri
    flash[:error] = 'Oops. You need to login before you can view that
page.'
    redirect_to :controller => 'user', :action => 'login'
  end
end
```

这段代码定义了两个过滤器，set_user和login_required，可以应用它们到操作或者控制器上。set_user过滤器在每个操作上运行（因为我们传递它到ApplicationController的before_filter，而ApplicationController是所有controller的父类）。set_user方法在用户已经登录时设置实例变量@user。现在有关登录用户的信息在应用程序里就可用了。操作方法和视图可以正常使用这个实例变量。这对于不要求登录的操作尤其有用：比如，主布局 视图可能会在每页上显示登录用户的名字。

可以通过为login_required方法传递符号到before_filter，来禁止未经鉴别的用户使用特定的操作或者控制器。如下显示如何保护定义在 *app/controllers/user_controller.rb* 里的 my_account 操作：

```
class UserController < ApplicationController
  before_filter :login_required, :only => :my_account
end
```

现在如果在未登录时试图使用 my_account 操作，会被重新定位到登录页面。

参考

- 13.14 节“ 用 ActiveRecord 确认数据 ”
- 15.6 节“ 集成数据库到 Rails 应用程序中 ”
- 15.9 节“ 保存散列化的用户密码到数据库中 ”
- 15.11 节“ 设置并找回会话信息 ”
- 不用再由自己完成这部分工作，可以安装 login_generator gem，并且使用其login产生器：它会给应用程序一个User模型以及实现基于密码的验证系统的控制器。查看 *http://wiki.rubyonrails.com/rails/pages/LoginGenerator*，以及 *http://wiki.rubyonrails.com/rails/pages/AvailableGenerators* 有其他产生器的信息（包括久负盛名的 model_security_generator）

15.9 保存散列化的用户密码到数据库中

问题

15.8 节定义的数据库表以普通文本保存用户的密码。这不是个好方法：如果某人侵入数据库，就能够得到所有用户的密码。所以最好保存为密码的安全散列。这样的话，没有密码（因此没有人能够窃取它），但是可以确保用户知道自己的密码。

解决方案

重新创建 15.8 节的users 表，取代密码字段，它有一个hashed_password字段。如下是完成这些的 MySQL 代码：

```
use mywebapp_development;
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(255) NOT NULL,
```

```
  `hashed_password` VARCHAR(40) NOT NULL,
  PRIMARY KEY (id)
);
```

打开 15.8 节创建的文件 *app/models/user.rb*，并且类似如下编辑它：

```
require 'sha1'

class User < ActiveRecord::Base
  attr_accessor :password
  attr_protected :hashed_password
  validates_uniqueness_of :username
  validates_confirmation_of :password,
    :if => lambda { |user| user.new_record? or not user.password.blank? }
  validates_length_of :password, :within => 5..40,
    :if => lambda { |user| user.new_record? or not user.password.blank? }

  def self.hashed(str)
    SHA1.new(str).to_s
  end

  # If a user matching the credentials is found, returns the User object.
  # If no matching user is found, returns nil.
  def self.authenticate(user_info)
    user = find_by_username(user_info[:username])
    if user && user.hashed_password == hashed(user_info[:password])
      return user
    end
  end

  private
  before_save :update_password

  # Updates the hashed_password if a plain password was provided.
  def update_password
    if not password.blank?
      self.hashed_password = self.class.hashed(password)
    end
  end
end
```

一旦完成了这些，应用程序会像以前一样起作用（虽然需要转变任何预先存在的用户账户为新的密码格式）。不需要修改任何控制器或者视图代码，因为`User.authenticate`方法采取和以前一样的工作方式。这是分离商业逻辑和表示逻辑的优点之一。

讨论

现在我们的用户模型有 3 段。第一段是加强了的确认代码。用户模型如下：

- 提供获取器和设置器密码属性。
- 确保数据库的 hashed_password 字段无法从外部访问。
- 确保每个用户有唯一的用户名。

当新的用户被创建，或者密码被更改时，`User` 确保：

- `password_confirmation` 属性的值等于密码属性的值。
- 密码在 5～40 字符长度之间。

代码的第二部分像以前一样定义 `User` 类。添加一个新的类级别的方法，`hashed`，在纯文本密码上执行散列化功能。如果想要在以后改变散列化机制，仅仅需要改变这个方法即可（同时移动所有现有密码）。

模型中代码的第三部分是私有的实例方法，`update_password`，它实现数据库中纯文本密码属性与散列化版本之间的同步。`before_save` 的调用设置该方法在 `User` 对象保存至数据库之前被调用。这样可以通过设置密码为纯文本值来改变用户的密码，而不用自己来完成这样的散列。

参考

- 13.14 节“ 用 ActiveRecord 确认数据 ”
- 15.8 节“ 创建登录系统 ”

15.10 转义显示用的 HTML 和 JavaScript

问题

想要显示可能包含HTML或JavaScript的数据，并且不让浏览器将其当作HTML渲染或者翻译该段 JavaScript。这在显示用户输入的数据时尤其重要。

解决方案

传递数据字符串到 `h( )` 辅助函数来转义其 HTML 记录项。也就是说，代替如下：

```
<%= @data %>
```

为：

```
<%=h @data %>
```

h( )帮助器函数转变字符为 HTML 记录项等式：与符号（&）、双引号（”），左三角括号（<）以及右三角括号（>）。

讨论

在 Rails 源代码中找不到 h( ) 帮助器函数的定义，因为这是 ERb 的内置帮助器函数 html_escape( ) 的快捷方式。

JavaScript 在类似 <SCRIPT> 的 HTML 标签里被展开，因此转义 HTML 字符串会压制 HTML 里的所有 JavaScript。然而，有时仅仅需要转义 JavaScript 为字符串。Rails 添加了可以使用的名为 escape_javascript( ) 的帮助器函数。该函数的功能不多：只是转变行转换符为字符串“ \n ”，同时在单和双引号前添加反斜杠。这在想要在自己的 JavaScript 代码里使用任意数据时很有用：

```
<!-- index.rhtml -->
<script lang="javascript">
var text = "<%= escape_javascript @javascript_alert_text %>";
alert(text);
</script>
```

参考

- 第 11 章

15.11 设置并找回会话信息

问题

想要关联数据到每个使用应用程序的独特 Web 客户端。数据需要在 HTTP 请求间保持恒定。

解决方案

可以使用 cookie（参考 15.12 节），但是将数据放到用户的会话中通常更简单。Rails 站点的每个访问者会被自动给予一个会话 cookie。Rails 以 cookie 值作为键，在服务器上创建一个任意数据的散列。

遍历全部的 Rails 应用程序，控制器、视图、helper 以及 mailer，可以通过调用称为 session 的方法来访问这个散列。保存在该散列里的对象在同一个 Web 浏览器的请求间保持恒定。

控制器里的下述代码跟踪客户端首次访问Web站点的时间：

```
class IndexController < ApplicationController
  def index
    session[:first_time] ||= Time.now
  end
end
```

在视图里，可以编写如下代码显示时间（注4）：

```
<!-- index.rhtml -->
You first visited this site on <%= session[:first_time] %>.

That was <%= time_ago_in_words session[:first_time] %> ago.
```

讨论

Cookie和会话很是相似。它们都保存有关站点访问者的持久数据。也都在HTTP顶端实现正式的操作，该HTTP没有属于自己的状态。cookie和会话的主要区别在于cookie里，所有数据保存在访问者计算机的小型cookie文件中，而会话里，所有数据保存在Web服务器里。客户端只是保存一个小型的会话cookie，它包含联系到服务器上数据的唯一ID。任何私人数据都保存在访问者的计算机上。

有几个理由使得可能需要使用会话代替cookie：

- cookie只能保存4K比特数据。
- cookie只能保存字符串值。
- 如果在cookie里保存个人信息，它会被截取，除非所有客户端的请求都被SSL封装。即使这样，站点间的脚本攻击可能会阅读客户端cookie并且得到敏感信息。

另一方面，cookie在如下情况很有用：

- 信息既不敏感，也不是很大。
- 不需要在服务器上保存每个访问者的会话信息。
- 需要应用程序的速度，并不是每一页都需要访问会话数据。

通常来说，使用会话优于在cookie里保存数据。

注4： 帮助函数`tine_ago_in_words()`计算某个特定时间起至返回英文文本之间的时间，文本类似为“about a minute”或“5 hours”或“2 days”。这是一种简单直观的方式告诉用户日期的含义。

可以在会话里包含模型对象 :这会在每个请求里从数据库找回相同的对象时省去很多麻烦。然而，如果想要完成这些，可以在应用程序控制器里列出所有将要放到会话里的模型。这会在从会话存储里找回数据时降低 Rails 不能解串行对象的风险。

```
class ApplicationController < ActionController::Base
  model :user, :ticket, :item, :history
end
```

然后可以将 ActiveRecord 对象放到会话里：

```
class IndexController < ApplicationController
  def index
    session[:user] ||= User.find(params[:id])
  end
end
```

如果站点不需要保存任何会话信息，可以禁止使用这个特性，通过添加如下代码到 `app/controllers/application.rb` 文件实现：

```
class ApplicationController < ActionController::Base
  session :off
end
```

也许可以猜到，还可以使用 `session` 方法将会话转变为单一的控制器：

```
class MyController < ApplicationController
  session :off
end
```

甚至可以将其降为操作级别：

```
class MyController < ApplicationController
  session :off, :only => ['index']

  def index
    #This action will not have any sessions available to it
  end
end
```

会话界面提供给需要在很多操作间保持的数据，可能是通过用户记录项访问站点。如果只是需要传递一个对象（比如状态消息）到下一个操作，那么使用 15.8 节介绍的 flash 构造会更简单：

```
flash[:error] = 'Invalid login.'
```

默认来说，Rails 会话通过 `PStore` 机制保存到服务器上。该机制使用 `Marshal` 串行化会话数据到临时文件里。这种方法对于小型站点很有效，但是如果站点有大量的访问者或者需要在多台服务器上并发运行 Rails 应用程序，那么就需要研究另外的方法。

3种主要可以考虑的方法为ActiveRecordStore、DRbStore和MemCacheStore。ActiveRecordStore保存会话信息到数据库表中：可以通过在命令行运行rake create_sessions_table来设置表。DRbStore和MemCacheStore都创建一个可以在网络上访问的内存散列，但是它们使用不同的库。

Ruby有一个称为DRb的标准库，允许在网络上共享对象（包括散列）。Ruby也有到Memcached守护进程的绑定，它帮助衡量Web站点比如Slashdot和LiveJournal。Memcached工作起来类似直接存储到RAM，可以自动分布到不同计算机上而不用任何特殊的配置。

要改变会话存储机制，编辑*config/environment.rb*文件类似如下：

```
Rails::Initializer.run do |config|
  config.action_controller.session_store = :active_record_store
end
```

参考

- 15.8节“创建登录系统”，有一个使用flash的例子
- 15.12节“设置并找回cookie”
- 16.10节“在任意数目的计算机间共享散列”
- 16.16节“用MemCached在分布式RAM上保存数据”
- *http://wiki.rubyonrails.com/rails/pages/HowtoChangeSessionOptions*

15.12 设置并找回Cookie

问题

想要在Rails里设置cookie

解决方案

回顾15.11节，所有Rails的控制器、视图、helper以及mailer访问名为sessions的方法，返回当前客户端会话信息的散列。控制器、helper以及mailer（不是用户的视图）也访问名为cookie的方法，返回当前客户端HTTP cookie的散列。

要为用户设置一个cookie，只需简单地在该散列里设置键/值对。比如，要保持跟踪访问者查看过的页面数目，可以设置“visits”cookie：

```
class ApplicationController < ActionController::Base
  before_filter :count_visits

  private

  def count_visits
    value = (cookies[:visits] || '0').to_i
    cookies[:visits] = (value + 1).to_s
    @visits = cookies[:visits]
  end
end
```

调用`before_filter`告诉Rails在调用任何操作方法之前运行该方法。私有声明使得Rails不会将`count_visits`方法当作可以公开查看的操作方法。

因为cookie对于视图不是直接可用的，`count_visits`使得`:visits cookie`的值可以当作实例变量`@visits`使用。这个变量可以从视图里访问：

```
<!-- index.rhtml -->
You've visited this website's pages <%= @visits %> time(s).
```

HTTP cookie值只能是字符串。Rails可以自动转化某些值为字符串，但是只把值存储到`cookie`里，这是最安全的方式。如果需要保存无法简单与字符串来回转变的对象，很可能需要将它们保存到会话散列里。

讨论

在很多情况下需要更多的cookie上的控制权。比如，Rails cookie默认在用户关闭浏览器会话时失效。如果想要改变浏览器失效时间，可以赋给`cookie`一个包含`:expires`关键值的散列，以及一个cookie失效的时间。如下面的cookie会在一个小时之后失效（注5）：

```
cookies[:user_id] = { :value => '123', :expires => Time.now + 1.hour}
```

还有另外一些可以传递进cookie的cookie散列的选项。

Cookie请求的域名：

```
:domain
```

注5： Rails扩展Ruby的数值类，使之包含一些十分有用的方法（比如此处所示的`hour`方法）。这些方法转变给定的单元为秒。比如，`Time.nor+1.hour`和`Time.nor+3600`是一样的，因为`1.hour`返回一个小时的秒数。其他有用的方法还包括`minutes`、`hours`、`days`、`months`、`weeks`以及`years`。因为它们全都被转变为秒数，甚至可以把它们都加起来，比如`1.week+3.days`。

cookie请求的URL路径（默认来说，cookie请求到入口域：这意味着如果在相同域名处运行多个应用程序，它们的cookie可能会冲突）：

```
:path
```

不管该cookie是否安全（安全的cookie只在HTTPS连接里传输，默认值为false）：

```
:secure
```

最后，Rails提供一个简单快捷的删除cookie的方式：

```
cookies.delete :user_id
```

当然，每个Ruby散列实现一种`delete`方法，但是`cookie`散列有一点不同。它包含特殊的代码，因此调用`delete`不仅会从`cookie`散列里移除键/值对，也会从用户浏览器里移除相应的cookie。

参考

- 3.5节“计算日期”
- 15.11节“设置并找回会话信息”讨论了何时使用`cookie`，以及何时使用会话

15.13 提取代码到辅助函数中

问题

视图被Ruby代码混淆了。

解决方案

创建一个有着十分复杂视图的控制器，查看会发生些什么：

```
$ ./scripts/generate controller list index
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/list
      exists  test/functional/
      create  app/controllers/list_controller.rb
      create  test/functional/list_controller_test.rb
      create  app/helpers/list_helper.rb
      create  app/views/list/index.rhtml
```

编辑`app/controllers/list_controller.rb`类似如下：

```
class ListController < ApplicationController
  def index
    @list = [1, "string", :symbol, ['list']]
  end
end
```

编辑app/views/list/index.rhtml包含如下代码。它迭代@list里的每个元素，并且打印出索引及其对象 ID 的 SHA1 散列：

```
<!-- app/views/list/index.rhtml -->
<ul>
<% @list.each_with_index do |item, i| %>
  <li class="<%= i%2==0 ? 'even' : 'odd' %>"><%= i %>:
    <%= SHA1.new(item.id.to_s) %></li>
<% end %>
</ul>
```

这显得很是混乱，但是如果已经做过很多 Web 编程工作，它们看上去会很相似。

要清除这段代码，需要为控制器移动其中一些到帮助器里。该例中，控制器称为list，因此其帮助器依赖于app/helpers/list_helper.rb。

创建名为create_li的帮助器函数。给定一个对象及其在列表里的位置，该函数创建适合在index 视图里使用的<LI>标签：

```
module ListHelper
  def create_li(item, i)
    %{<li class="#{ i%2==0 ? 'even' : 'odd' }">#{i}:
      #{SHA1.new(item.id.to_s)}</li>}
  end
end
```

list控制器的视图访问了所有定义在ListHelper里的函数。可以类似如下来清除index视图：

```
<!-- app/views/list/index.rhtml -->
<ul>
<% @list.each_with_index do |item, i| %>
  <%= create_li(item, i) %>
<% end %>
</ul>
```

帮助器函数可以完成所有通常从视图里完成的事情，因此这是一种很好的提取重要部分的方法。

讨论

帮助器函数的目的是创建维护性更强的代码，并且在程序员和UI设计者之间执行更好的

工作分工。可维护代码使得程序员更容易维护，当它在帮助器函数里时，它不会妨碍设计人员，他们能够来回tweak HTML，而无需过滤代码。

何时使用帮助器的一个很好的规则是大声朗读代码。如果它对于熟悉HTML的人来说，听上去没有什么意义，或者它由多于一个的英文短句所组成，那么应该将它隐藏到帮助器后。

这样做的负面作用是必须最小化帮助器里生成的HTML的数量。UI设计者，或者其他熟悉HTML的人员采取的方式不会混乱代码，就能找出何处可以找到需要tweak的HTML数据。

虽然帮助器函数很有用并且经常会被用到，Rails也提供了*partial*，另一种提取代码为转化成更小部分的方法。

参考

- 15.14节“重构视图为视图的部分片断”中更详细地介绍了partial

15.14 重构视图为视图的部分片断

问题

视图不包含很多Ruby代码，但是它仍然变得越来越复杂。想要重构视图的逻辑到分散的、可重用的模板里。

解决方案

可以重构视图模板到称为partial的多个模板中。一个模板可以通过调用`render`方法来包含其他模板，`render`方法在15.5节首次出现。

从15.5节所示视图的更加复杂的版本开始讨论：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
 <li><%= item.name %> -
  <%= link_to 'Delete', {:action => 'delete', :id => item.id},
              :post => true %>
 </li>
<% end %>
```

```
</ul>

<h2>Add a new item</h2>

<%= form_tag :action => 'new' %>
 Item: <%= text_field "product", "name" %>&#x00A;
 <%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

如下是相应的控制器类，以及一个虚拟的作为模型的 `ListItem` 类：

```
# app/controllers/list_controller.rb
class ListController < ActionController::Base
  def shopping_list
    @list = [ListItem.new(4, 'aspirin'), ListItem.new(199, 'succotash')]
  end

  # Other actions go here: add, delete, etc.
  # ...
end

class ListItem
  def initialize(id, name)
    @id, @name = id, name
  end
end
```

该视图包括两部分：第一部分列出所有项，第二部分打印出表格添加一个新项。很明显第一步需要分离新项表格。

可以通过创建一个部分视图来打印出新项表格。要完成这个，需要在 *app/views/list/* 里创建新的名为*_new_item_form.rhtml*的文件。文件名前的下划线表明这是一个部分视图，不是 `new_item_form` 操作的完整的视图。如下是部分文件。

```
<!-- app/views/list/_new_item_form.rhtml -->

<%= form_tag :action => 'new' %>
Item: <%= text_field "item", "value" %>&#x00A;
<%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

要包含一个部分，从某个模板里调用 `render` 方法。如下是集成到主视图里的 `_new_item_form` 部分。视图看上去几乎一样，但是代码被更好地组织了。

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
 <li><%= item.name %> -
  <%= link_to 'Delete', {:action => 'delete', :id => item.id},
```

```
                :post => true %>
 </li>
<% end %>
</ul>
<%= render :partial => 'new_item_form' %>
```

即使文件名以下划线开始，当调用部分时，仍可能会忽略下划线。

讨论

部分视图继承了控制器提供的所有实例变量，因此可以像访问父视图一样访问相同的实例变量。这就是为什么没有必要为`_new_item_form`部分而改变所有表格的代码。

可以创建第二部分来构建为每个列表项目打印出<LI>标签的代码。如下是*_list_item.rhtml*：

```
<!-- app/views/list/_list_item.rhtml -->
<li><%= list_item.name %> -
<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},
            :post => true %>
</li>
```

如下是修改了的主视图：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
<% @list.each do |item| %>
  <%= render :partial => 'list_item', :locals => {:list_item => item} %>
<% end %>
</ul>

<%= render :partial => 'new_item_form' %>
```

部分视图没有从父视图继承本地变量，因此项目变量需要传递进部分，在特殊的称为`:locals`的散列里。它在部分里作为`list_item`可以被访问，因为这就是它在散列中被赋予的名称。

这种情况，在`Enumerable`上的迭代，为每个元素翻译一个部分，这在Web应用程序中十分常见，因此Rails为此提供了快捷方式。可以更加简化主视图，通过传递数组到`render`里（作为`:collection`参数）实现，使之完成这样的迭代：

```
<!-- app/views/list/shopping_list.rhtml -->
<h2>My shopping list</h2>

<ul>
 <%= render :collection => @list, :partial => 'list_item' %>
```

```
</ul>

<%= render :partial => 'new_item_form' %>
```

部分针对@list里的每个元素被翻译一次。每个列表元素作为本地变量list_item可用。万一还没有猜出来,这个名字来自于部分自身的名字:render自动给_foo.rhtml一个名为 foo 的本地变量。

list_item_counter 是另一个自动设置的变量（再一次，名字反映了模板的名字）。list_item_counter 是迭代集合的当前项的索引。在想改变列表项以不同的风格显示时，这个变量很是有用：

```
<!-- app/views/list/_list_item.rhtml -->
<li><%= list_item.name %> -
<% css_class = list_item_counter % 2 == 0 ? 'a' : 'b' %>
<%= link_to 'Delete', {:action => 'delete', :id => list_item.id},
            {'class' => css_class}, :post => true %>
</li>
```

在当前没有集合的地方，可以通过指定 render 的:object 参数，来传递一个单一对象到部分里。这比创建只需传递一个对象的:locals 所需的整个散列要简单得多。通过:collection，对象被当作本地变量使用，该本地对象的名称基于此部分的名称。

如下有一个例子：发送购物列表到new_item_form.rhtml部分，因此新项的表格可以打印出更详细的消息。这给 shopping_list.rhtml 发挥作用提供了机会：

```
<%= render :partial => 'new_item_form', :object => @list %>
```

如下是 _new_item_form.rhtml 的新版本：

```
<!-- app/views/list/_new_item_form.rhtml -->
<h2>Add a new item to the <%= new_item_form.size %> already in this
list</h2>

<%= form_tag :action => 'new' %>
 Item: <%= text_field "product", "name" %>
 <%= submit_tag "Add new item" %>
<%= end_form_tag %>
```

参考

- 15.5 节“ 用 render 显示模板 ”

15.15 用 script.aculo.us 添加 DHTML 效果

问题

想要添加奇特的效果，比如应用程序的淡出，而无需编写任何 JavaScript。

解决方案

每个 Rails 应用程序和一些 JavaScript 库绑定，允许创建 Ajax 和 DHTML 效果。甚至不需要在 Rails 的 Web 站点编写 JavaScript 使能 DHTML。

首先编写主布局模板（查看 15.3 节），在 `<HEAD>` 标签调用 `javascript_include_tag`：

```
<!-- app/views/layouts/application.rhtml -->

<html>
  <head>
    <title>My Web App</title>
    <%= javascript_include_tag "prototype", "effects" %>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

现在在视图里，可以调用 `visual_effect` 方法来实现 `script.aculo.us` 库里的 DHTML 窍门。

如下是“高亮”效果的例子：

```
<p id="important">Here is some important text, it will be highlighted
when the page loads.</p>

<script type="text/javascript">
<%= visual_effect(:highlight, "important", :duration => 1.5) %>
</script>
```

如下是“淡去”效果的例子：

```
<p id="deleted">Here is some old text, it will fade away when the page
loads.</p>

<script type="text/javascript">
<%= visual_effect(:fade, "deleted", :duration => 1.0) %>
</script>
```

讨论

如上示例代码段在页面加载时被触发，因为它们被封装在<SCRIPT>标签里。在实际应用程序中，可能需要响应用户操作显示文本效果：删除了的项目会淡去消失，或者选中某项时会高亮相关项目。如下是一幅图像，当点击其下方的链接时会发出咯吱的声音：

```
<img id="to-squish" src="bug.jpg">
<%=link_to_function("Squish the bug!", visual_effect(:squish, "to-
squish"))%>
```

visual_effect 生成的 JavaScript 代码看上去很想传递进方法的参数。比如，Rails 视图的一段代码如下：

```
<script type="text/javascript">
<%= visual_effect(:fade, 'deleted-text', :duration => 1.0) %>
</script>
```

生成如下 JavaScript：

```
<script type="text/javascript">
new Effect.Fade("deleted-text", {duration:1.0});
</script>
```

这样的相似性意味着 script.aculo.us 库的文档几乎可以直接应用到 visual_effect。这也意味着如果更愿意编写直接的 JavaScript，那么代码对于懂得 visual_effect 的人来说仍然很容易理解。

如下表格列出 Rails 1.0 里的很多可用的效果。

JavaScript 初始化	Rails 初始化
new Effect.Highlight	visual_effect(:highlight)
new Effect.Appear	visual_effect(:appear)
new Effect.Fade	visual_effect(:fade)
new Effect.Puff	visual_effect(:puff)
new Effect.BlindDown	visual_effect(:blind_down)
new Effect.BlindUp	visual_effect(:blind_up)
new Effect.SwitchOff	visual_effect(:switch_off)
new Effect.SlideDown	visual_effect(:slide_down)
new Effect.SlideUp	visual_effect(:slide_up)
new Effect.DropOut	visual_effect(:drop_out)
new Effect.Shake	visual_effect(:shake)

JavaScript 初始化	Rails 初始化
new Effect.Pulsate	visual_effect(:pulsate)
new Effect.Squish	visual_effect(:squish)
new Effect.Fold	visual_effect(:fold)
new Effect.Grow	visual_effect(:grow)
new Effect.Shrink	visual_effect(:shrink)
new Effect.ScrollTo	visual_effect(:scroll_to)

参考

- `script.aculo.us` 演示（*http://wiki.script.aculo.us/scriptaculous/show/CombinationEffectsDemo*）
- 15.3 节“创建页眉和页脚的布局”中有更多关于布局模板的信息
- 15.17 节“创建 Ajax 表格”

15.16 生成操作模型对象的表格

问题

想要定义操作，使得用户创建或者编辑保存在数据库里的对象。

解决方案

让我们创建一个简单的模型，然后为其构建表格。如下是一段 MySQL 代码，创建一个键 / 值对的表：

```
use mywebapp_development;
DROP TABLE IF EXISTS items;
CREATE TABLE `items` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(255) NOT NULL default '',
  `value` varchar(40) NOT NULL default '[empty]',
  PRIMARY KEY (`id`)
);
```

现在，从命令行，创建模型类，以及控制器和视图：

```
$ ./script/generate model Item
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
```

```
      create  app/models/item.rb
      create  test/unit/item_test.rb
      create  test/fixtures/items.yml
      create  db/migrate
      create  db/migrate/001_create_items.rb

$ ./script/generate controller items new create edit
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/items
      exists  test/functional/
      create  app/controllers/items_controller.rb
      create  test/functional/items_controller_test.rb
      create  app/helpers/items_helper.rb
      create  app/views/items/new.rhtml
      create  app/views/items/edit.rhtml
```

第一步是要自定义视图。从 *app/views/items/new.rhtml* 开始。编辑它类似如下：

```
<!-- app/views/items/new.rhtml -->

<%= form_tag :action => "create" %>
 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

所有这些方法调用生成HTML：`form_tag`，打开一个`<FORM>`标签，`submit_tag`生成一个提交按钮，等等。可以手工键入相同的HTML，Rails不会介意这些，但是构建方法调用会更容易，也会使得模板更加简洁。

`text_field`调用会涉及一些。它创建一个`<INPUT>`标签，作为文本记录项字段显示在HTML表格里。但是它同时绑定该字段的值到`@item`实例变量的成员之一。本段代码创建一个文本记录项字段，绑定到`@item`的`name`成员：

```
<%= text_field "item", "name" %>
```

但是`@item`实例变量是什么呢？是的，它还没有定义，因为我们使用的是已经生成的控制器。如果现在尝试访问页面`/items/new` page，可能会得到一个错误，提示预期外的`nil`值。`nil`值指的就是`@item`变量，它还没有定义就被使用了（在`text_field`调用里）。

让我们自定义`ItemsController`类，从而`new`操作会恰当设置`@item`实例变量。同时实现`create`操作，从而当用户单击生成的表格上的提交按钮时确实会发生预想的事情。

```
class ItemsController < ApplicationController
  def new
```

```
    @item = Item.new
  end

  def create
    @item = Item.create(params[:item])
    redirect_to :action => 'edit', :id => @item.id
  end
end
```

如果访问/items/new page，会看到预期的效果：一个有两个文本记录项字段的表格。“ Name ”字段是空的，“ Value ”字段包含“ [empty] ”的默认数据库值。

填写该表格并提交，会在items表里创建新行。会被重新定位到edit操作，这个操作并不存在。现在创建它。如下是控制器部分（注意上述ItemsController#edit和ItemsController#create间的相似性）：

```
class ItemsController < ApplicationController
  def edit
    @item = Item.find(params[:id])

    if request.post?
      @item.update_attributes(params[:item])
      redirect_to :action => 'edit', :id => @item.id
    end
  end
end
```

实际上，edit操作和create操作十分相似，它们的表格几乎都是一样的。唯一不同的地方在于form_tag的参数：

```
<!-- app/views/items/edit.rhtml -->

<%= form_tag :action => "edit", :id => @item.id %>
 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

讨论

这几乎是Web开发人员每天都会遇到的常见问题。这样的问题如此普遍，所以Rails有一个名为scaffold的工具，可以生成这样的代码。如果想要这样调用generate而不用上述给定的参数，Rails会为“ 解决方案 ”里给出的操作生成代码，并再加上一些：

```
$ ./script/generate scaffold Items
```

用scaffold开始并不意味着可以不必知道Rails表格如何生成，因为仍然需要明确自定义scaffold代码。

代码中有两处是神奇之处。第一个是视图中的text_field调用，它在“ 解决方案 ”里已涉及了。它绑定对象成员（比如@item.name）到一个 HTML 表格控制。如果查看 /items/new 页面的源代码，会看到表格字段类似如下：

```
Name: <input type="text" name="item[name]" value="" /><br />
Value <input type="text" name="item[value]" value="[empty]" /><br />
```

这些特殊的字段名称被第二个神奇之处使用，定位在Item.create（在new里）和Item#update_attributes的调用里。两种情况下，Item对象被填入新值的散列作为其成员。该散列被嵌入到params散列里，它包含 CGI 格式值。

HTML 表格字段的名称（item[name]和item[value]）翻译到params散列，类似如下：

```
{
  :item => {
    :name => "Name of the item",
    :value => "Value of the item"
  },
  :controller => "items",
  :action => "create"
}
```

因此这样一行代码：

```
Item.create(params[:item])
```

和下面这行代码一样高效：

```
Item.create(:name => "Name of the item", :value => "Value of the item")
```

在edit的操作里Item#update_attributes的调用几乎以相同的方式起作用。

如上所述，edit和new的视图很是相似，只是表格的目的地不同而已。用很少的重构操作，可以彻底移除视图文件之一。

没有任何参数的<%= form_tag %>调用设置表格目的地为当前的URL。适当地修改 *new.rhtml* 文件：

```
<!-- app/views/items/new.rhtml -->
<%= form_tag %>

Name: <%= text_field "item", "name" %>&#x00A;
Value: <%= text_field "item", "value" %>&#x00A;

<%= submit_tag %>
<%= end_form_tag %>
```

现在new.rhtml视图适合于new和edit的使用。只需改变new操作来调用create方法（因为表格不会到那里），并且改变edit操作来渲染new.rhtml而不是edit.rhtml（它可能会被移除）：

```
class ItemsController < ApplicationController
  def new
    @item = Item.new
    create if request.post?
  end

  def edit
    @item = Item.find(params[:id])

    if request.post?
      @item.update_attributes(params[:item])
      redirect_to :action => 'edit', :id => @item.id and return
    end
    render :action => 'new'
  end
end
```

回顾15.5节，render调用近几年指定会使用的模板文件。edit里的render调用不会实际调用new方法，因此不需要担心new方法会覆盖@item的值。

在现实生活里，add和edit表格的内容会有很大的不同来为每个操作分离视图。然而，这些表格之间又有足够的相似性使得它们能够被重构到两个视图共享着的单一的部分视图里（查看15.14节）。这是一个关于DRY（Don't Repeat Yourself）规则的很好的例子。如果add和edit视图都各自有一个单表格，它更简单也很少出错，那么倾向于在数据库方案改变时维护表格。

参考

- 15.5节“用render显示模板”
- 15.14节“重构视图为视图的部分片断”

15.17 创建Ajax表格

问题

想要构建一个能够做出响应并且易于使用的Web应用程序。不会让用户耗费大量时间等待浏览器重画屏幕。

解决方案

可以使用 JavaScript 来使得浏览器的 XMLHTTPRequest 对象发送数据给服务器，而不会令用户不得不忍受普遍的（但是缓慢的）页面更新。这样的技术叫做 Ajax（注6），Rails 使得可以很简单地使用 Ajax，而无需编写或熟悉 JavaScript。

在 Web 应用程序里实现 Ajax 之前，必须编辑应用程序的主布局模板，使得在 <HEAD> 标签里调用 javascript_include_tag 方法。这和 15.15 节做出的改变是一样的：

```
<!-- app/views/layouts/application.rhtml -->

<html>
  <head>
    <title>My Web App</title>
    <%= javascript_include_tag "prototype", "effects" %>
  </head>
  <body>
    <%= @content_for_layout %>
  </body>
</html>
```

从 15.16 节的基础上改变应用程序，这里 new 操作是 AJAX 使能的（如果依据那一节的所有方式，并使得 edit 操作使用 new.rhtml 而不是 edit.rhtml，那么不需要那些改变并使得 edit 使用其自己的视图模板）。

从视图模板开始。编辑 *app/views/items/new.rhtml* 类似如下：

```
<!-- app/views/items/new.rhtml -->
<div id="show_item"></div>

        <%= form_remote_tag :url => { :action => :create },
        :update => "show_item",
        :complete => visual_effect(:highlight, "show_item") %>

 Name: <%= text_field "item", "name" %><br />
 Value: <%= text_field "item", "value" %><br />
 <%= submit_tag %>
<%= end_form_tag %>
```

这些细小的改动使得标准的 HTML 格式变为 Ajax 格式。主要区别在于调用 form_remote_tag 而不是 form_tag。另一个区别在于传递进方法的参数。

第一个改变是将 :action 参数放进一个散列，该散列传递进 :url 选项。Ajax 格式比起正常的格式有着更多的相关选项，因此像 form_tag 一样简单描述其格式操作。

注 6： 这并不能完全代表 Asynchronous JavaScript 和 XML。单词 Ajax 的原型现在是计算机神话的一部分，并非只是首字母的缩写。

当用户单击提交按钮时，表格值被串行化并且发送到后台的目的操作里（本例中，为create）。create操作像之前一样处理表格的提交，并且返回一段HTML。

这段HTML是怎么回事呢？这是:update选项需要的。它告诉Rails得到表格提交的结果，并且将其放入HTML ID为“show_item”的元素里。这是为什么我们在模板顶部添加 标签的原因：这是服务器响应的位置。

最后new.rhtml 视图的改变是:complete选项。这是一个回调参数：它指定JavaScript代码的字符串，并会在Ajax请求结束时运行。我们在服务器请求出现时高亮它们。

这就是视图。也需要修改控制器里的create操作，使得当构建一次Ajax表格提交时，服务器返回一段HTML。这就是会插入到浏览器端“show_item”元素里的片断。如果对于一次标准（非Ajax）的表格提交，服务器的行为类似15.16节，发送一次HTTP重新定位（注7）。控制器类看上去需要类似如下：

```
class ItemsController < ApplicationController
  def new
    @item = Item.new
  end

  def create
    @item = Item.create(params[:item])
      if request.xml_http_request?
      render :action => 'show', :layout => false
    else
      redirect_to :action => 'edit', :id => @item.id
    end
  end

  def edit
    @item = Item.find(params[:id])

    if request.post?
       @item.update_attributes(params[:item])
       redirect_to :action => 'edit', :id => @item.id
    end
  end
end
```

这段代码参考一个新的视图，show。这是服务器返回的小型HTML片断，被Web浏览器插入到“show_element”标签里。需要定义它：

```
<!-- app/views/items/show.rhtml -->

Your most recently created item:<br />
```

注7：这会发生在有人在JavaScript关闭时使用应用程序的情况下。

```
Name: <%= @item.name %><br />
Value: <%= @item.value %><br />
   <hr>
```

现在当使用 *http://localhost:3000/items/new* 来添加新项到数据库时，不会被重新定位到 `edit` 操作。会仍然停留在新的页面上，表格提交的结果会显示在表格上方。这使得一次创建很多新项更简单。

讨论

15.16 节介绍了如何以传统方式提交数据给表格：用户单击“提交”按钮，浏览器发送请求到服务器，服务器返回响应页面，浏览器渲染相应页面。

最近，类似 Gmail 和 Google Maps 这样的站点普及一种无需页面刷新而发送并接受数据的技术。相同地，这些技术被称为 Ajax。Ajax 是很实用的工具可以改进应用程序的响应时间及可用性。

Ajax 请求是某个应用程序操作的真实 HTTP 请求，可以像对待任何其他请求一样处理它。然而，大部分时间里，并不希望返回的是一整页 HTML，而只是数据的片断。Web 浏览器会在全部 Web 页面（早些时候服务的）内容里发送 Ajax 请求，该页面知道如何处理请求片断。

可以在一次 Ajax 请求生命周期的好几处定义 JavaScript 回调函数。一个回调函数，`:complete`，被用来在插入某片断到页面后高亮它。下表列出其他回调函数。

回调函数名称	回调函数描述
:loading	当 Web 浏览器设计用来加载远程文档时被调用
:loaded	当浏览器完成远程文件加载时被调用
:interactive	当用户可以与远程文档交互，即使其还没有完成加载时被调用
:success	当 XMLHttpRequest 完成，HTTP 状态代码在 2XX 范围时被调用
:failure	当 XMLHttpRequest 完成，HTTP 状态代码不在 2XX 范围时被调用
:complete	当 XMLHttpRequest 完成时被调用。如果:success 和 / 或:failure 也正运行着，那么该函数在它们之后运行

15.18 在 Web 站点上发布 Web 服务

问题

想要在自己的 Web 应用程序里提供 SOAP 和 XML-RPC Web 服务。

解决方案

Rails有一个内置的Web服务生成器，它使得很容易实现将控制器的操作当作Web服务发布。不需要花时间编写WSDL文件或者设置，不需要知道SOAP和XML-RPC是如何工作的。

如下是一个简单的例子。首先，按照15.16节的指导，创建一个名为`items`的数据库表，并且为该表生成一个模型。不要生成控制器。

现在，在命令行运行该表格：

```
./script/generate web_service Item add edit fetch
      create  app/apis/
      exists  app/controllers/
      exists  test/functional/
      create  app/apis/item_api.rb
      create  app/controllers/item_controller.rb
      create  test/functional/item_api_test.rb
```

这会创建一个`item`控制器，它支持3种操作：`add`、`edit`和`fetch`。但是不使用有`.rhtml`视图的Web应用程序，这些是通过SOAP或XML-RPC访问的Web服务操作。

Ruby方法不关心对象接收为参数的数据类型，或者其返回值的数据类型。但是SOAP或XML-RPC Web服务方法会关心这些。要想通过SOAP或XML-RPC接口发布Ruby方法，需要为其签名定义类型信息。打开文件*app/apis/item_api.rb*并对它做类似如下的编辑：

```
class ItemApi < ActionWebService::API::Base
  api_method :add, :expects => [:string, :string], :returns => [:int]
  api_method :edit, :expects => [:int, :string, :string], :returns =>
[:bool]
  api_method :fetch, :expects => [:int], :returns => [Item]
end
```

现在需要实现实际的Web服务界面。打开*app/controllers/item_controller.rb*，并做类似如下的编辑：

```
class ItemController < ApplicationController
  wsdl_service_name 'Item'

  def add(name, value)
    Item.create(:name => name, :value => value).id
  end

  def edit(id, name, value)
    Item.find(id).update_attributes(:name => name, :value => value)
  end
```

```
  def fetch(id)
    Item.find(id)
  end
end
```

讨论

现在item控制器为items表实现了SOAP和XML-RPC Web服务。这个控制器和items的控制器同时存在，后者实现传统的Web 界面（注 8）。

XML-RPC API 的 URL 是 *http://www.yourserver.com/item/api*，SOAP API 的 URL 是 *http://www.yourserver.com/item/service.wsdl*。要测试这些服务，如下是一段简短的 Ruby 脚本，它通过 SQAP 客户端调用 Web 服务器方法：

```
require 'soap/wsdlDriver'

wsdl = "http://localhost:3000/item/service.wsdl"
item_server = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver

item_id = item_server.add('foo', 'bar')

if item_server.edit(item_id, 'John', 'Doe')
  puts 'Hey, it worked!'
else
  puts 'Back to the drawing board...'
end
# Hey, it worked!

item = item_server.fetch(item_id)
item.class                                          # =>
SOAP::Mapping::Object
item.name                                           # => "John"
item.value                                          # => "Doe"
```

如下是 XML-RPC 等效程序：

```
require 'xmlrpc/client'
item_server = XMLRPC::Client.new2('http://localhost:3000/item/api')

item_id = item_server.call('Add', 'foo', "bar")
if item_server.call('Edit', item_id, 'John', 'Doe')
  puts 'Hey, it worked!'
else
  puts 'Back to the drawing board...'
```

注 8： 甚至可以添加 Web 界面操作到 `ItemController` 类里。然后单一的 controller 可以实现传统Web界面以及 Web服务界面。但是无法将Web应用程序操作的名称定义为和Web服务操作相同的名字，因为 controller 类只能包含一种给定名称的方法。

```
end
# Hey, it worked!

item = item_server.call('Fetch', item_id)
# => {"name"=>"John", "id"=>2, "value"=>"Doe"}
```

参考

- Matt Biddulph的文章“REST on Rails”描述了如何在Rails顶端创建REST类型的Web服务（*http://www.xml.com/pub/a/2005/11/02/rest-on-rails.html*）
- 16.3节“编写XML-RPC客户端”，以及16.4节“编写SOAP客户端”
- 16.5节“编写SOAP服务器”提出了一种SOAP Web服务的非Rails的实现方式

15.19 用Rails发送邮件

问题

想要从Rails应用程序里发送邮件：可能是一次配置或者一个排序，或者通知某个操作已经被当作是用户行为。

解决方案

首先要做的是生成一些mailer基础构造。进入应用程序的基础目录，键入如下命令：

```
./script/generate mailer Notification welcome
      exists  app/models/
      create  app/views/notification
      exists  test/unit/
      create  test/fixtures/notification
      create  app/models/notification.rb
      create  test/unit/notification_test.rb
      create  app/views/notification/welcome.rhtml
      create  test/fixtures/notification/welcome
```

向应用程序的邮件中心提供名称“Notification”，这和Web界面里的控制器有某种程度的类似。设置mailer来生成单一的邮件，名为“welcome”：这和视图模板的一个操作很是类似。

现在打开`app/models/notification.rb`，并做类似如下的编辑：

```
class Notification < ActionMailer::Base
  def welcome(user, sent_at=Time.now)
    @subject =  'A Friendly Welcome'
    @recipients = user.email
    @from =  'admin@mysite.com'
```

```
    @sent_on =    sent_at
    @body = {
      :user => user,
      :sent_on => sent_at
    }

    attachment 'text/plain' do |a|
      a.body = File.read('rules.txt')
    end
  end
end
```

邮件的主题是“ A Friendly Welcome ”，它从地址 *admin@mysite.com* 处发送到用户的邮箱地址。它有一个附件，是磁盘文件 `rules.txt`（相对于Rails应用程序的根目录而言）。

虽然文件 `notification.rb` 在 `models/` 目录下，它的功能类似于控制器，其每个邮箱消息都有相关联的视图模板。Welcome邮件的视图在 `app/views/notification/welcome.rhtml` 里，它和正常控制器的视图功能相同。

最重要的区别在于 mailer 视图不需要访问 mailer 的实例变量。要为 mailer 设置实例变量，传递这些变量的散列到 `body` 方法。键称为实例变量的名称，值称为它们的值。在 `notification.rb` 里，为 welcome 视图构造两个可用的实例变量，`@user` 和 `@sent_on`。如下是视图代码本身：

```
<!-- app/views/notification/welcome.rhtml -->

Hello, <%= @user.name %>, and thanks for signing up at <%= @sent_on
%>. Please print out the attached set of rules and keep them in a
prominent place; they help keep our community running smoothly. Be
sure to pay special attention to sections II.4 ("Assignment of
Intellectual Property Rights") and XIV.21.a ("Dispute Resolution
Through Ritual Combat").
```

从 Rails 应用程序里发送 welcome 邮件，添加下述代码到控制器、模型或者观察器里均可：

```
Notification.deliver_welcome(user)
```

这里，`user` 变量可以是任何实现 `#name` 和 `#email` 的对象，这两个方法在 `welcome` 方法和模板里被调用。

讨论

永远不会直接调用 `Notification#welcome` 方法。实际上，`Notification#welcome` 是不可用的，因为它是一个实例方法，所以永远无法直接实例化 `Notification` 对象。

ActionMailer::Base类定义了method_missing的实现，它查看所有未定义类方法的调用。这就是为什么即使没有定义过deliver_welcome但仍然可以调用它的原因。

上述给定的welcome.rhtml模板生成普通文本的邮件。要发送HTML邮件，只需简单添加下述代码到Notification#welcome：

```
content_type 'text/html'
```

现在模板可以生成HTML，邮件客户端会识别出邮件的格式并适当地加以渲染。

有时会想得到有关传输过程的更多的控制权，比如，当单元测试ActionMailer类时。不用调用deliver_welcome来发送邮件，可以调用create_welcome得到作为Ruby对象的邮件。这些“create”方法返回TMail对象，有必要时可以检查和操作它们。

如果本地Web服务器无法发送邮件，可以修改environment.rb来联系远程的SMTP服务器：

```
Rails::Initializer.run do |config|
  config.action_mailer.server_settings = {
    :address => 'someserver.com',
    :user_name => 'uname',
    :password => 'passwd',
    :authentication => 'cram_md5'
  }
end
```

参考

- 10.8节“响应对未定义方法的调用”
- 14.5节“发送邮件”中介绍了有关ActionMailer和SMTP设置方面的更多信息

15.20 自动发送错误信息到邮箱

问题

想要在每次用户遇到应用程序错误时都收到一封描述性的邮件信息。

解决方案

应用程序运行中发生的错误会被发送到ActionController::Base#log_error方法。如果已经设置了mailer（如15.19节所示），那么可以重载该方法并且使其发送邮件。代码类似如下：

```
class ApplicationController < ActionController::Base

private
   def log_error(exception)
      super
      Notification.deliver_error_message(exception,
        clean_backtrace(exception),
        session.instance_variable_get("@data"),
        params,
        request.env
      )
   end
end
```

代码收集失败发生时 Rails 请求的全方位状态信息。它捕获异常对象、相应的踪迹、会话数据、CGI 请求参数以及所有环境变量的值。

重载了的 `log_error` 调用 `Notification.deliver_error_messsage`，它假定已经创建了名为“ Notification ”的 mailer，并且定义了 `Notification.error_message` 方法。如下是实现代码：

```
class Notification < ActionMailer::Base
  def error_message(exception, trace, session, params, env, sent_on =
Time.now)
    @recipients         = 'me@mydomain.com'
    @from               = 'error@mydomain.com'
    @subject            = "Error message: #{env['REQUEST_URI']}"
    @sent_on            = sent_on
    @body = {
      :exception => exception,
      :trace => trace,
      :session => session,
      :params => params,
      :env => env
    }
  end
end
```

该邮件的模板如下：

```
<!-- app/views/notification/error_message.rhtml -->

Time: <%= Time.now %>
Message: <%= @exception.message %>
Location: <%= @env['REQUEST_URI'] %>
Action: <%= @params.delete('action') %></td></tr>
Controller: <%= @params.delete('controller') %></td></tr>
Query: <%= @env['QUERY_STRING'] %></td></tr>
Method: <%= @env['REQUEST_METHOD'] %></td></tr>
SSL: <%= @env['SERVER_PORT'].to_i == 443 ? "true" : "false" %>
Agent: <%= @env['HTTP_USER_AGENT'] %>
```

```
Backtrace
<%= @trace.to_a.join("</p>\n<p>") %>

Params
<% @params.each do |key, val| -%>
* <%= key %>: <%= val.to_yaml %>
<% end -%>

Session
<% @session.each do |key, val| -%>
* <%= key %>: <%= val.to_yaml %>
<% end -%>

Environment
<% @env.each do |key, val| -%>
* <%= key %>: <%= val %>
<% end -%>
```

讨论

`ActionController::Base#log_error` 提供了根据自己喜好处理错误的灵活性。这在 Rails 应用程序工作在访问受限的主机上时尤其有效：可以要求错误发送出来，而不是写入到一个可能无法查阅的文件里。或者更愿意记录错误到数据库里，因此可以查找选用相应的模式。

`ApplicationController#log_error`方法被私有调用来避免二义性。如果它不是私有的，所有控制器会认为它们已经定义了`log_error`。用户能够访问`/<controller>/log_error`并使得 Rails 工作异常。

参考

- 15.19 节“ 用 Rails 发送邮件 ”

15.21 文档化 Web 站点

问题

想要文档化 Web 应用程序的控制器、模型和 helper，开发人员需要负责维护应用程序，使其理解该如何去工作。

解决方案

和其他任何 Ruby 程序一样，通过向代码添加特定格式的命令来文档化 Rails 应用程序。如下说明如何向`FooController` 类及其方法添加文档：

```
# The FooController controller contains miscellaneous functionality
# rejected from other controllers.
class FooController < ApplicationController
  # The set_random action sets the @random_number instance variable
  # to a random number.
  def set_random
    @random_number = rand*rand
  end
end
```

类和方法的文档在其声明之前出现，而不是在之后。

当已经完成应用程序文档命令的添加时，回到 Rails 应用程序根目录下并处理 `rake appdoc` 命令：

```
$ rake appdoc
```

这个 Rake 任务为 Rails 应用程序运行 RDoc，并且生成名为 `doc/app` 的目录。该目录包含一个 Web 站点，它是所有文档命令的集合，与源代码互为参考。在任何 Web 浏览器中打开 `doc/app/index.rhtml`，可以浏览生成的文档。

讨论

RDoc 文档可以包含 markup 和特殊的指示：可以在定义列表里描述参数，通过 `:nodoc:` 指令从文档里隐藏类或方法。这在 17.11 节里有具体的描述。

Rails 应用程序和其他 Ruby 程序唯一的不同之处在于 Rails 有一个定义了 appdoc 任务的 Rakefile。不需要自己查找或编写它。

可能已经在方法里加入了内联命令，在操作发生时描述它们。因为 RDoc 文档包含了源代码的格式化版本，这些命令对于查阅 RDoc 的人来说是可见的。然而，这些命令被格式化成 Ruby 源代码，而不是 RDoc markup。

参考

- 17.11 节“ 文档化应用程序 ”
- 第 19 章，尤其是 19.2 节“ 自动生成文档 ”
- RDoc 的 RDoc（*http://rdoc.sourceforge.net/doc/index.html*）

15.22 Web站点的单元测试

问题

想要创建自动化测试套件来测试Rails应用程序的功能。

解决方案

Rails无法编写比视图和控制器更多的测试代码，但是它的确使得组织和运行自动化测试变得简单。

当使用`./script/generate`命令创建控制器和模型时，不仅可以节省时间，而且得到了生成的针对单元和功能测试的框架。通过自己编写的功能测试来填写该框架，可以得到相当理想的测试覆盖率。

至今为止，本章的所有例子都是在Rails应用程序开发数据库上运行的，因此只需要确保正确设置了*config/database.yml*文件的开发部分。单元测试代码基于应用程序测试数据库运行，因此现在需要同时设置测试部分。`mywebapp_test`数据库无需包含任何表，但是它必须存在而且能够被Rails使用。

当用`generate`脚本生成模型时，Rails也会在test目录生成针对该模型的单元测试脚本。它同时创建一个*fixture*、YAML文件，包含可以加载进`mywebapp_test`数据库的测试数据。这就是单元测试基于运行的数据：

```
./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
      create  db/migrate
      create  db/migrate/001_create_users.rb
```

当用`generate`创建一个控制器时，Rails为控制器创建了功能测试脚本：

```
./script/generate users list
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/users
      exists  test/functional/
      create  app/controllers/users_controller.rb
      create test/functional/users_controller_test.rb
      create  app/helpers/users_helper.rb
      create  app/views/users/list.rhtml
```

因为在模型和控制器类里编写代码，需要在这些文件里编写相应的测试。

要运行单元和功能测试，在主目录下调用 `rake` 命令。默认的 Rake 任务会运行所有测试。如果在生成测试文件之后立即运行它，会产生如下效果：

```
$ rake
(in /home/lucas/mywebapp)
/usr/bin/ruby1.8 "test/unit/user_test.rb"
Started
.
Finished in 0.048702 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
/usr/bin/ruby1.8 "test/functional/users_controller_test.rb"
Started
.
Finished in 0.024615 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

讨论

所有用其他语言或在其他Ruby程序里编写的单元测试知识都可以应用到Rails里。Rails为我们完成一些计算，并且它定义了一些很是实用的断言（如下所示），但是仍然需要我们自己完成一些工作。回报也是相同的：可以自信地修改和重构代码，因为知道如果什么中断了，那么测试也会中断。可以立即知道问题所在并且可以快速修改。

一起看看 Rails 生成了什么。如下是生成的 *test/unit/user_test.rb*：

```
require File.dirname(__FILE_  _) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

不错的开始，但是 `test_truth` 有些同义反复。如下是更理性的测试：

```
class UserTest
  def test_first
    assert_kind_of User, users(:first)
  end
end
```

该代码从`users`表得到第一个元素，并且断言 ActiveRecord 将其转变为`User`对象。这

样测试User代码（没有编写任何代码）并没有测试Rails和ActiveRecord效果好，但是它显示了针对良好单元测试的断言。

但是users(:first)是如何返回什么的呢？测试套件在mywebapp_test数据库上运行，甚至没有在其中添加任何表，以及少许示例数据。

我们没有，但是Rails有。当运行测试套件时，Rails复制开发数据库方案到测试数据库中。不用针对存在于开发数据库的每一个数据都运行测试，Rails从名为fixtures的YAML文件里加载特殊的测试数据。fixture文件包含需要测试的所有数据库数据：被测试删除而存在的对象、不同表行间的特殊关系、或者其他任何需要的东西。

上述示例中，users表的fixture被fixtures :users这一行代码加载。如下是为User模型生成的fixture，在*test/fixtures/users.yml*里：

```
first:
  id: 1
another:
  id: 2
```

在运行单元测试之前，Rails阅读该文件，在users表里创建两行，并且为它们定义别名（:first和:another），因此可以在单元测试里引用它们。然后定义users方法（正如其他所示，该方法名称基于模型的名称）。在test_first里，users(:first)的调用找回User对象，它对应fxture里的:first：ID 1的那个对象。

如下是另一个单元测试：

```
class UserTest
  def test_another
    assert_kind_of User, users(:another)
    assert_equal 2, users(:another).id
    assert_not_equal users(:first), users(:another)
  end
end
```

Rails添加如下Rails特定的断言到Ruby的Test::Unit：

- assert_dom_equal
- assert_dom_not_equal
- assert_generates
- assert_no_tag
- assert_recognizes
- assert_redirected_to

- assert_response
- assert_routing
- assert_tag
- assert_template
- assert_valid

参考

- “测试Rails”是Rails单元和功能测试的指导（*http://manuals.rubyonrails.com/read/book/5*）
- Rails 1.1 通用支持集成测试，针对 controller 和操作之间交互的测试，查看 *http://rubyonrails.com/rails/classes/ActionController/IntegrationTest.html* 和 *http://jamis.jamisbuck.org/articles/2006/03/09/integration-testing-in-rails-1-1*
- ZenTest 库包含 Test::Rails，它使得可以为视图和 controller 编写单独的测试（*http://rubyforge.org/projects/zentest/*）
- 在 *http://ar.rubyonrails.org/classes/Fixtures.html* 阅读有关 fixture 的信息
- 在 *http://rails.rubyonrails.com/classes/Test/Unit/Assertions.html* 阅读有关 assertions that Rails adds to `Test::Unit` 的信息
- 15.6 节“集成数据库到 Rails 应用程序中”
- 17.7 节“编写单元测试”
- 第 19 章

15.23 在 Web 应用程序中使用断点

问题

Rails应用程序有无法通过记录消息找到的bug。需要一种耐受力强的调试工具，可以内部审查应用程序在任何给定点的全部状态。

解决方案

`Breakpoint` 库使得可以停止代码流程并且进入 `irb`，一个交互的 Ruby 会话。在 irb 里，可以内部审查当前作用域中的本地变量，修改这些变量，并且继续代码正常流程的

执行。如果曾经耗费很长时间试图通过在某处放置记录消息来跟踪bug，会发现breakpoint使得一切变得简单并且可以更直观地进行调试。

但是如何从Web应用程序里运行一个交互控制台程序呢？答案是必须事先运行该控制台程序，侦听Rails服务器的调用。

第一步是从命令行运行./script/breakpointer。这个命令启动服务器，它侦听网络上的Rails服务器的breakpoint调用。在终端窗口一直运行该程序：这就是irb会话开启之处：

```
$ ./script/breakpointer
No connection to breakpoint service at druby://localhost:42531
Tries to connect will be made every 2 seconds...
```

要触发一次irb会话，可以在Rails中的任何地方调用breakpoint方法，在模型、控制器或者帮助器方法里。当执行到这一点时，进入的客户端请求的处理会停止，irb会话会在终端启动。当退出会话时，请求处理会继续。

讨论

有一个例子。假定已经编写了如下控制器，在修改Item对象名称属性时遇到困难。

```
class ItemsController < ApplicationController
  def update
    @item = Item.find(params[:id])
    @item.value = '[default]'
    @item.name = params[:name]
    @item.save
    render :text => 'Saved'
  end
end
```

可以在Item类中放置一个breakpoint调用，如下：

```
class Item < ActiveRecord::Base
  attr_accessor :name, :value

  def name=(name)
    super
    breakpoint
  end
end
```

访问URL *http://localhost:3000/items/update/123?name=Foo*调用ItemController#update，它发现Item号码123并且随后调用其name=方法。name=的调用触发断点。不用翻译文本“Saved”，站点挂起并且不响应请求。

但是如果返回终端运行breakpointer服务器，可以看到已经启动了一个交互的Ruby会话。该会话允许在断点调用处处理所有本地变量和方法：

```
Executing break point "Item#name=" at item.rb:4 in `name='
  irb:001:0> local_variables
  => ["name", "value", "_", "_  _"]
  irb:002:0> [name, value]
  => ["Foo", "[default]"]
 irb:003:0> [@name, @value]
  => ["Foo", "[default]"]
  irb:004:0> self
  => #<Item:0x292fbe8 @name="Foo", @value="[default]">
  irb:005:0> self.value = "Bar"
  => "Bar"
  irb:006:0> save
 => true
  irb:006:0> exit

Server exited. Closing connection...
```

一旦结束了，键入exit中止交互的Ruby会话。Rails应用程序从刚才断开处继续运行，按照预期翻译“Saved”。

默认来说，断点是根据其所在的方法命名的。可以传递一个字符串进breakpoint，得到一个更具描述性的名称。这在某个方法中包含多个断点时很有用：

```
breakpoint "Trying to set Item#name, just called super"
```

不必直接调用breakpoint，也可以调用assert，该方法得到一个代码块。如果该块计算后得到false，Ruby调用breakpoint，否则继续正常的流程。使用assert使得可以设置这样的断点，它们只在运行有误时才被调用（在传统的调试器里，这是“条件断点”）：

```
1.upto 10 do |i|
  assert { Person.find(i) }
  p = Person.find(i)
  p.update_attribute(:name, 'Lucas')
end
```

如果所有要求的Person对象都被找出，breakpoint不会被调用，因为Person.find会一直返回true。如果有一个Person对象丢失了，那么Ruby调用breakpoint方法，会得到有待调查的irb会话。

断点是功能强大的工具，能够可观地简化调试过程。要理解其真正的功能很难，除非亲身体会，因此用自己的代码来验证这个解决方案吧。

参考

- 17.10节的“使用断点审查并改变应用程序的状态”中更详细地讲解了有关断点的更多知识
- *http://wiki.rubyonrails.com/rails/show/HowtoDebugWithBreakpoint*