5 天搞定 Ruby on Rails

企业内训纲要

第 5 天：Rails 布局和视图渲染

版本： V1

口号：快速迭代，不断完善

文档，实例地址：

https://github.com/nienwoo/ruby-learn

1. 第 5 天目标

- 掌握 Ruby Rails 布局和视图渲染。

2. 布局结构

Rails 渲染响应的视图时，会把视图和当前模板结合起来。查找当前模板的方法前文已经介绍过。在布局中可以使用三种工具把各部分合在一起组成完整的响应：

静态资源标签

yield 和 content_for

局部视图

2.1. 静态资源标签帮助方法

静态资源帮助方法用来生成链接到 Feed、JavaScript、样式表、图片、视频和音频的 HTML 代码。Rails 提供了六个静态资源标签帮助方法：

auto_discovery_link_tag

javascript_include_tag

stylesheet_link_tag

image_tag

video_tag

audio_tag

这六个帮助方法可以在布局或视图中使用，不过 auto_discovery_link_tag、

javascript_include_tag 和 stylesheet_link_tag 最常出现在布局的 <head> 中。

静态资源标签帮助方法不会检查指定位置是否存在静态资源，假定你知道自己在做什么，只负责生成对应的链接。

2.1.1. 使用 auto_discovery_link_tag 链接到 Feed

auto_discovery_link_tag 帮助方法生成的 HTML，大多数浏览器和 Feed 阅读器都能用来自动识别 RSS 或 Atom Feed。auto_discovery_link_tag 接受的参数包括链接的类型（:rss 或 :atom），传递给 url_for 的 Hash 选项，以及该标签使用的 Hash 选项：

```
<%= auto_discovery_link_tag(:rss, {action: "feed"},
  {title: "RSS Feed"}) %>
```

auto_discovery_link_tag 的标签选项有三个：

:rel：指定链接 rel 属性的值，默认值为 "alternate"；

:type：指定 MIME 类型，不过 Rails 会自动生成正确的 MIME 类型；

:title：指定链接的标题，默认值是 :type 参数值的全大写形式，例如 "ATOM" 或 "RSS"；

2.1.2. 使用 javascript_include_tag 链接 JavaScript 文件

javascript_include_tag 帮助方法为指定的每个资源生成 HTML script 标签。

如果启用了 Asset Pipeline，这个帮助方法生成的链接指向 /assets/javascripts/ 而不是 Rails 旧版中使用的 public/javascripts。链接的地址由 Asset Pipeline 伺服。

Rails 程序或引擎中的 JavaScript 文件可存放在三个位置：app/assets，lib/assets 或 vendor/assets。详细说明参见 Asset Pipeline 中的“静态资源的组织方式”一节。

文件的地址可使用相对文档根目录的完整路径，或者是 URL。例如，如果想链接到 app/assets、lib/assets 或 vendor/assets 文件夹中名为 javascripts 的子文件夹中的文件，可以这么做：

```
<%= javascript_include_tag "main" %>
```

Rails 生成的 script 标签如下：

```
<script src='/assets/main.js'></script>
```

对这个静态资源的请求由 Sprockets gem 伺服。

同时引入 app/assets/javascripts/main.js 和 app/assets/javascripts/columns.js 可以这么做：

```
<%= javascript_include_tag "main", "columns" %>
```

引入 app/assets/javascripts/main.js 和 app/assets/javascripts/photos/columns.js：

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

引入 http://example.com/main.js：

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

2.1.3. 使用 stylesheet_link_tag 链接 CSS 文件

stylesheet_link_tag 帮助方法为指定的每个资源生成 HTML <link> 标签。

如果启用了 Asset Pipeline，这个帮助方法生成的链接指向 /assets/stylesheets/，由 Sprockets gem 伺服。样式表文件可以存放在三个位置：app/assets，lib/assets 或 vendor/assets。

文件的地址可使用相对文档根目录的完整路径，或者是 URL。例如，如果想链接到 app/assets、lib/assets 或 vendor/assets 文件夹中名为 stylesheets 的子文件夹中的文件，可以这么做：

```
<%= stylesheet_link_tag "main" %>
```

引入 app/assets/stylesheets/main.css 和 app/assets/stylesheets/columns.css：

```
<%= stylesheet_link_tag "main", "columns" %>
```

引入 app/assets/stylesheets/main.css 和 app/assets/stylesheets/photos/columns.css：

```
<%= stylesheet_link_tag "main", "photos/columns" %>
```

引入 http://example.com/main.css：

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

默认情况下，stylesheet_link_tag 创建的链接属性为 media="screen" rel="stylesheet"。指定相应的选项（:media，:rel）可以重写默认值：

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

2.1.4. 使用 image_tag 链接图片

image_tag 帮助方法为指定的文件生成 HTML <img /> 标签。默认情况下，文件存放在 public/images 文件夹中。

注意，必须指定图片的扩展名。

```
<%= image_tag "header.png" %>
```

可以指定图片的路径：

```
<%= image_tag "icons/delete.gif" %>
```

可以使用 Hash 指定额外的 HTML 属性：

```
<%= image_tag "icons/delete.gif", {height: 45} %>
```

可以指定一个替代文本，在关闭图片的浏览器中显示。如果没指定替代文本，Rails 会使用图片的文件名，去掉扩展名，并把首字母变成大写。例如，下面两个标签会生成相同的代码：

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", alt: "Home" %>
```

还可指定图片的大小，格式为“{width}x{height}”：

```
<%= image_tag "home.gif", size: "50x20" %>
```

除了上述特殊的选项外，还可在最后一个参数中指定标准的 HTML 属性，例如 :class、:id 或 :name：

```
<%= image_tag "home.gif", alt: "Go Home",
                          id: "HomeImage",
                          class: "nav_bar" %>
```

2.1.5. 使用 video_tag 链接视频

video_tag 帮助方法为指定的文件生成 HTML5 <video> 标签。默认情况下，视频文件存放在 public/videos 文件夹中。

```
<%= video_tag "movie.ogg" %>
```

生成的代码如下：

```
<video src="/videos/movie.ogg" />
```

和 image_tag 类似，视频的地址可以使用绝对路径，或者相对 public/videos 文件夹的路径。而且也可以指定 size: "#{width}x#{height}" 选项。video_tag 还可指定其他 HTML 属性，例如 id、class 等。

video_tag 方法还可使用 HTML Hash 选项指定所有 <video> 标签的属性，包括：

poster: "image_name.png"：指定视频播放前在视频的位置显示的图片；

autoplay: true：页面加载后开始播放视频；

loop: true：视频播完后再次播放；

controls: true：为用户提供浏览器对视频的控制支持，用于和视频交互；

autobuffer: true：页面加载时预先加载视频文件；

把数组传递给 video_tag 方法可以指定多个视频：

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

生成的代码如下：

```
<video><source src="trailer.ogg" /><source src="movie.ogg" /></video>
```

2.2. yield

2.2.1. 理解 yield

在布局中，yield 标明一个区域，渲染的视图会插入这里。

最简单的情况是只有一个 yield，此时渲染的整个视图都会插入这个区域：

```
<html>
  <head>
  </head>
  <body>
  <%= yield %>
  </body>
```

```
</html>
```

布局中可以标明多个区域：

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

视图的主体会插入未命名的 yield 区域。

要想在具名 yield 区域插入内容，得使用 content_for 方法。

2.2.2. content_for 方法

content_for 方法在布局的具名 yield 区域插入内容。

例如，下面的视图会在前一节的布局中插入内容：

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

套入布局后生成的 HTML 如下：

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

如果布局不同的区域需要不同的内容，例如侧边栏和底部，就可以使用 content_for 方法。content_for 方法还可用来在通用布局中引入特定页面使用的 JavaScript 文件或 CSS 文件。

2.2.3. 使用局部视图

局部视图可以把渲染过程分为多个管理方便的片段，把响应的某个特殊部分移入单独的文件。

（1）具名局部视图

在视图中渲染局部视图可以使用 render 方法：

```
<%= render "menu" %>
```

渲染这个视图时，会渲染名为 _menu.html.erb 的文件。注意文件名开头的下划线：局部视图的文件名开头有个下划线，用于和普通视图区分开，不过引用时无需加入下划线。即便从其他文件夹中引入局部视图，规则也是一样：

```
<%= render "shared/menu" %>
```

这行代码会引入 app/views/shared/_menu.html.erb 这个局部视图。

（2）使用局部视图简化视图

局部视图的一种用法是作为“子程序”（subroutine），把细节提取出来，以便更好地理解整个视图的作用。例如，有如下的视图：

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>

...
```

```
<%= render "shared/footer" %>
```

这里，局部视图 _ad_banner.html.erb 和 _footer.html.erb 可以包含程序多个页面共用的内容。在编写某个页面的视图时，无需关心这些局部视图中的详细内容。

程序所有页面共用的内容，可以直接在布局中使用局部视图渲染。

（3）局部布局

和视图可以使用布局一样，局部视图也可使用自己的布局文件。例如，可以这样调用局部视图：

```
<%= render partial: "link_area", layout: "graybar" %>
```

这行代码会使用 _graybar.html.erb 布局渲染局部视图 _link_area.html.erb。

注意，局部布局的名字也以下划线开头，和局部视图保存在同个文件夹中（不在 layouts 文件夹中）。

还要注意，指定其他选项时，例如 :layout，必须明确地使用 :partial 选项。

（4）传递本地变量

地变量可以传入局部视图，这么做可以把局部视图变得更强大、更灵活。例如，可以使用这

种方法去除新建和编辑页面的重复代码，但仍然保有不同的内容：

```
<h1>New zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
<h1>Editing zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
<%= form_for(zone) do |f| %>
  <p>
    <b>Zone name</b><br>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

虽然两个视图使用同一个局部视图，但 Action View 的 submit 帮助方法为 new 动作生成的提交按钮名为“Create Zone”，为 edit 动作生成的提交按钮名为“Update Zone”。

每个局部视图中都有个和局部视图同名的本地变量（去掉前面的下划线）。通过 object 选项可以把对象传给这个变量：

```
<%= render partial: "customer", object: @new_customer %>
```

在 customer 局部视图中，变量 customer 的值为父级视图中的 @new_customer。

如果要在局部视图中渲染模型实例，可以使用简写句法：

```
<%= render @customer %>
```

假设实例变量 @customer 的值为 Customer 模型的实例，上述代码会渲染 _customer.html.erb，其中本地变量 customer 的值为父级视图中 @customer 实例变量的值。

（5）渲染集合

渲染集合时使用局部视图特别方便。通过 :collection 选项把集合传给局部视图时，会把集合中每个元素套入局部视图渲染：

```
<h1>Products</h1>
<%= render partial: "product", collection: @products %>
<p>Product Name: <%= product.name %></p>
```

传入复数形式的集合时，在局部视图中可以使用和局部视图同名的变量引用集合中的成员。在上面的代码中，局部视图是 _product，在其中可以使用 product 引用渲染的实例。

渲染集合还有个简写形式。假设 @products 是 product 实例集合，在 index.html.erb 中可以直接写成下面的形式，得到的结果是一样的：

```
<h1>Products</h1>

<%= render @products %>
```

Rails 根据集合中各元素的模型名决定使用哪个局部视图。其实，集合中的元素可以来自不同的模型，Rails 会选择正确的局部视图进行渲染。

```
<h1>Contacts</h1>

<%= render [customer1, employee1, customer2, employee2] %>

<p>Customer: <%= customer.name %></p>

<p>Employee: <%= employee.name %></p>
```

在上面几段代码中，Rails 会根据集合中各成员所属的模型选择正确的局部视图。

如果集合为空，render 方法会返回 nil，所以最好提供替代文本。

```
<h1>Products</h1>

<%= render(@products) || "There are no products available." %>
```

（6）本地变量

要在局部视图中自定义本地变量的名字，调用局部视图时可通过 :as 选项指定：

```
<%= render partial: "product", collection: @products, as: :item %>
```

这样修改之后，在局部视图中可以使用本地变量 item 访问 @products 集合中的实例。

使用 locals: {} 选项可以把任意本地变量传入局部视图：

```
<%= render partial: "product", collection: @products,
        as: :item, locals: {title: "Products Page"} %>
```

在局部视图中可以使用本地变量 title，其值为 "Products Page"。

在局部视图中还可使用计数器变量，变量名是在集合后加上 _counter。例如，渲染 @products 时，在局部视图中可以使用 product_counter 表示局部视图渲染了多少次。不过不能和 as: :value 一起使用。

在使用主局部视图渲染两个实例中间还可使用 :spacer_template 选项指定第二个局部视图。

（7）集合局部视图的布局

渲染集合时也可使用 :layout 选项。

```
<%= render partial: "product", collection: @products, layout: "special_layout" %>
```

使用局部视图渲染集合中的各元素时会套用指定的模板。和局部视图一样，当前渲染的对象以及 object_counter 变量也可在布局中使用。

2.2.4. 使用嵌套布局

在程序中有时需要使用不同于常规布局的布局渲染特定的控制器。此时无需复制主视图进行编辑，可以使用嵌套布局（有时也叫子模板）。下面举个例子。

假设 ApplicationController 布局如下：

```
<html>
<head>
  <title><%= @page_title or "Page Title" %></title>
  <%= stylesheet_link_tag "layout" %>
  <style><%= yield :stylesheets %></style>
</head>
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content"><%= content_for?(:content) ? yield(:content) : yield %></div>
</body>
</html>
```

在 NewsController 的页面中，想隐藏顶部目录，在右侧添加一个目录：

```
<% content_for :stylesheets do %>
```

```
    #top_menu {display: none}
    #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<% content_for :content do %>
    <div id="right_menu">Right menu items here</div>
    <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render template: "layouts/application" %>
```

就这么简单。News 控制器的视图会使用 news.html.erb 布局，隐藏了顶部目录，在 中添加一个右侧目录。

使用子模板方式实现这种效果有很多方法。注意，布局的嵌套层级没有限制。使用 render template: 'layouts/news' 可以指定使用一个新布局。如果确定，可以不为 News 控制器创建子模板，直接把 content_for?(:news_content) ? yield(:news_content) : yield 替换成 yield 即可。

3. 表单

3.1. 表单基础

3.1.1. 空表单

（8）form_tag

最基本的表单帮助方法是 form_tag。

```
<%= form_tag do %>
  Form contents
<% end %>
```

（9）<form> 标签

像上面这样不传入参数时，form_tag 会创建一个 <form> 标签，提交表单后，向当前页面发起 POST 请求。假设当前页面是 /home/index，生成的 HTML 如下（为了提升可读性，添加了一些换行）：

```
<form accept-charset="UTF-8" action="/home/index" method="post">
  <div style="margin:0;padding:0">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
```

```
    Form contents
</form>
```

你会发现 HTML 中多了一个 div 元素，其中有两个隐藏的 input 元素。这个 div 元素很重要，没有就无法提交表单。第一个 input 元素的 name 属性值为 utf8，其作用是强制浏览器使用指定的编码处理表单，不管是 GET 还是 POST。第二个 input 元素的 name 属性值为 authenticity_token，这是 Rails 的一项安全措施，称为“跨站请求伪造保护”。form_tag 帮助方法会为每个非 POST 表单生成这个元素（表明启用了这项安全保护措施）。详情参阅“Rails 安全指南”。

为了行文简洁，后续代码没有包含这个 div 元素。

3.1.2. 搜索表单

（10）方法

创建这样一个表单要分别使用帮助方法 form_tag、label_tag、text_field_tag 和 submit_tag，如下所示：

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

（11）HTML

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/search" method="get">
  <div style="margin:0;padding:0;display:inline"><input name="utf8" type="hidden" value="&#x2713;" /></div>
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

表单中的每个 input 元素都有 ID 属性，其值和 name 属性的值一样（上例中是 q）。ID 可用于 CSS 样式或使用 JavaScript 处理表单控件。

除了 text_field_tag 和 submit_tag 之外，每个 HTML 表单控件都有对应的帮助方法。

搜索表单的请求类型一定要用 GET，这样用户才能把某个搜索结果页面加入收藏夹，以便后续访问。一般来说，Rails 建议使用合适的请求方法处理表单。

3.1.3. 多个 Hash 参数

1.2 调用 form_tag 时使用多个 Hash 参数

form_tag 方法可接受两个参数：表单提交地址和一个 Hash 选项。Hash 选项指定提交表单使用的请求方法和 HTML 选项，例如

form 元素的 class 属性。

和 link_to 方法一样，提交地址不一定非得使用字符串，也可使用一个由 URL 参数组成的 Hash，这个 Hash 经 Rails 路由转换成 URL 地址。这种情况下，form_tag 方法的两个参数都是 Hash，同时指定两个参数时很容易产生问题。假设写成下面这样：

```
form_tag(controller: "people", action: "search", method: "get", class: "nifty_form")

# => '<form accept-charset="UTF-8" action="/people/search?method=get&class=nifty_form" method="post">'
```

在这段代码中，method 和 class 会作为生成 URL 的请求参数，虽然你想传入两个 Hash，但实际上只传入了一个。所以，你要把第一个 Hash（或两个 Hash）放在一对花括号中，告诉 Ruby 哪个是哪个，写成这样：

```
form_tag({controller: "people", action: "search"}, method: "get", class: "nifty_form")

# => '<form accept-charset="UTF-8" action="/people/search" method="get" class="nifty_form">'
```

3.2. 表单方法

3.2.1. 帮助方法

Rails 提供了很多用来生成表单中控件的帮助方法，例如复选框，文本输入框和单选框。这些基本的帮助方法都以 _tag 结尾，例如 text_field_tag 和 check_box_tag，生成单个 input 元素。这些帮助方法的第一个参数都是 input 元素的 name 属性值。提交表单后，name 属性的值会随表单中的数据一起传入控制器，在控制器中可通过 param 复选框 Rails 使用特定的规则生成 input 的 name 属性值，便于提交非标量值，例如数组和 Hash，这些值也可通过 params 获取。

各帮助方法的详细用法请查阅 API 文档。

（12）复选框

复选框是一种表单控件，给用户一些选项，可用于启用或禁用某项功能。

```
<%= check_box_tag(:pet_dog) %>

<%= label_tag(:pet_dog, "I own a dog") %>

<%= check_box_tag(:pet_cat) %>
```

```
<%= label_tag(:pet_cat, "I own a cat") %>
```

生成的 HTML 如下：

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

check_box_tag 方法的第一个参数是 name 属性的值。第二个参数是 value 属性的值。选中复选框后，value 属性的值会包含在提交的表单数据中，因此可以通过 params 获取。

（13）单选框

单选框有点类似复选框，但是各单选框之间是互斥的，只能选择一组中的一个：

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
```

```
<%= radio_button_tag(:age, "adult") %>

<%= label_tag(:age_adult, "I'm over 21") %>
```

生成的 HTML 如下：

```
<input id="age_child" name="age" type="radio" value="child" />

<label for="age_child">I am younger than 21</label>

<input id="age_adult" name="age" type="radio" value="adult" />

<label for="age_adult">I'm over 21</label>
```

和 check_box_tag 方法一样，radio_button_tag 方法的第二个参数也是 value 属性的值。因为两个单选框的 name 属性值一样（都是 age），所以用户只能选择其中一个单选框，params[:age] 的值不是 "child" 就是 "adult"。

复选框和单选框一定要指定 label 标签。label 标签可以为指定的选项框附加文字说明，还能增加选项框的点选范围，让用户更容易选中。

（14）其他帮助方法

其他值得说明的表单控件包括：多行文本输入框，密码输入框，

隐藏输入框，搜索关键字输入框，电话号码输入框，日期输入框，时间输入框，颜色输入框，日期时间输入框，本地日期时间输入框，月份输入框，星期输入框，URL 地址输入框，Email 地址输入框，数字输入框和范围输入框：

```
<%= text_area_tag(:message, "Hi, nice site", size: "24x6") %>

<%= password_field_tag(:password) %>

<%= hidden_field_tag(:parent_id, "5") %>

<%= search_field(:user, :name) %>

<%= telephone_field(:user, :phone) %>

<%= date_field(:user, :born_on) %>

<%= datetime_field(:user, :meeting_time) %>

<%= datetime_local_field(:user, :graduation_day) %>

<%= month_field(:user, :birthday_month) %>

<%= week_field(:user, :birthday_week) %>

<%= url_field(:user, :homepage) %>

<%= email_field(:user, :address) %>
```

```
<%= color_field(:user, :favorite_color) %>

<%= time_field(:task, :started_at) %>

<%= number_field(:product, :price, in: 1.0..20.0, step: 0.5) %>

<%= range_field(:product, :discount, in: 1..100) %>
```

生成的 HTML 如下：

```
<textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea>

<input id="password" name="password" type="password" />

<input id="parent_id" name="parent_id" type="hidden" value="5" />

<input id="user_name" name="user[name]" type="search" />

<input id="user_phone" name="user[phone]" type="tel" />

<input id="user_born_on" name="user[born_on]" type="date" />

<input id="user_meeting_time" name="user[meeting_time]" type="datetime" />

<input id="user_graduation_day" name="user[graduation_day]"
```

type="datetime-local" />

<input id="user_birthday_month" name="user[birthday_month]" type="month" />

<input id="user_birthday_week" name="user[birthday_week]" type="week" />

<input id="user_homepage" name="user[homepage]" type="url" />

<input id="user_address" name="user[address]" type="email" />

<input id="user_favorite_color" name="user[favorite_color]" type="color" value="#000000" />

<input id="task_started_at" name="task[started_at]" type="time" />

<input id="product_price" max="20.0" min="1.0" name="product[price]" step="0.5" type="number" />

<input id="product_discount" max="100" min="1" name="product[discount]" type="range" />

用户看不到隐藏输入框，但却和其他文本类输入框一样，能保存数据。隐藏输入框中的值可以通过 JavaScript 修改。

搜索关键字输入框，电话号码输入框，日期输入框，时间输入框，

颜色输入框，日期时间输入框，本地日期时间输入框，月份输入框，星期输入框，URL 地址输入框，Email 地址输入框，数字输入框和范围输入框是 HTML5 提供的控件。如果想在旧版本的浏览器中保持体验一致，需要使用 HTML5 polyfill（使用 CSS 或 JavaScript 编写）。polyfill 虽无不足之处，但现今比较流行的工具是 Modernizr 和 yepnope，根据检测到的 HTML5 特性添加相应的功能。

如果使用密码输入框，或许还不想把其中的值写入日志。具体做法参见“Rails 安全指南”。

3.2.2. 表单与对象绑定

（15）绑定属性与 Hash 对象

表单的一个特别常见的用途是编辑或创建模型对象。

这时可以使用 *_tag 帮助方法，但是太麻烦了，每个元素都要设置正确的参数名称和默认值。Rails 提供了很多帮助方法可以简化这一过程，这些帮助方法没有 _tag 后缀，例如 text_field 和 text_area。

这些帮助方法的第一个参数是实例变量的名字，第二个参数是在对象上调用的方法名（一般都是模型的属性）。

Rails 会把在对象上调用方法得到的值设为控件的 value 属性值,并且设置相应的 name 属性值。

如果在控制器中定义了 @person 实例变量，其名字为“Henry”，在表单中有以下代码：

```
<%= text_field(:person, :name) %>
```

生成的结果如下：

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

提交表单后，用户输入的值存储在 params[:person][:name] 中。

params[:person] 这个 Hash 可以传递给 Person.new 方法；

如果 @person 是 Person 的实例，还可传递给 @person.update。

一般来说，这些帮助方法的第二个参数是对象属性的名字，但 Rails 并不对此做强制要求，只要对象能响应 name 和 name= 方法即可。

传入的参数必须是实例变量的名字，例如 :person 或 "person"，而不是模型对象的实例本身。

Rails 还提供了用于显示模型对象数据验证错误的帮助方法，详情参阅“Active Record 数据验证”一文。

（16）把表单绑定到对象上

虽然上述用法很方便，但却不是最好的使用方式。

如果 Person 有很多要编辑的属性，我们就得不断重复编写要编辑对象的名字。

我们想要的是能把表单绑定到对象上的方法，form_for 帮助方法就是为此而生。

假设有个用来处理文章的控制器 app/controllers/articles_controller.rb：

```
def new
  @article = Article.new
end
```

在 new 动作对应的视图 app/views/articles/new.html.erb 中可以像下面这样使用 form_for 方法：

```
<%= form_for @article, url: {action: "create"}, html: {class: "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, size: "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

有几点要注意：

@article 是要编辑的对象；

form_for 方法的参数中只有一个 Hash。

路由选项传入嵌套 Hash :url 中，

HTML 选项传入嵌套 Hash :html 中。

还可指定 :namespace 选项为 form 元素生成一个唯一的 ID 属

性值。:namespace 选项的值会作为自动生成的 ID 的前缀。

form_for 方法会拽入一个表单构造器对象（f 变量）；

生成表单控件的帮助方法在表单构造器对象 f 上调用；

上述代码生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/articles/create" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

form_for 方法的第一个参数指明通过 params 的哪个键获取表单中的数据。

在上面的例子中，第一个参数名为 article，因此所有控件的 name 属性都是 article[attribute_name] 这种形式。所以，在 create 动作中，

params[:article] 这个 Hash 有两个键：:title 和 :body。name 属性的重要性参阅“理解参数命名约定”一节。

在表单构造器对象上调用帮助方法和在模型对象上调用的效果一样，唯有一点区别，无法指定编辑哪个模型对象，因为这由表单构造器负责。

使用 fields_for 帮助方法也可创建类似的绑定，但不会生成 <form> 标签。在同一表单中编辑多个模型对象时经常使用 fields_for 方法。

例如，有个 Person 模型，和 ContactDetail 模型关联，编写如下的表单可以同时创建两个模型的对象：

```
<%= form_for @person, url: {action: "create"} do |person_form| %>

  <%= person_form.text_field :name %>
```

```
<%= fields_for @person.contact_detail do |contact_details_form| %>
  <%= contact_details_form.text_field :phone_number %>
<% end %>
<% end %>
```

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/people/create" class="new_person" id="new_person" method="post">
<input id="person_name" name="person[name]" type="text" />
<input id="contact_detail_phone_number" name="contact_detail[phone_number]" type="text" />
</form>
```

fields_for 方法拽入的对象和 form_for 方法一样，都是表单构造器（其实在代码内部 form_for 会调用 fields_for 方法）。

（17）自动绑定

使用“记录辨别”技术可以简化 form_for 方法的调用。简单来说，你可以只把模型实例传给 form_for，让 Rails 查找模型名等其他信息：

```
## Creating a new article
# long-style:
form_for(@article, url: articles_path)
# same thing, short-style (record identification gets used):
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, url: article_path(@article), html: {method: "patch"})
# short-style:
form_for(@article)
```

用户可以直接处理程序中的 Article 模型，根据开发 Rails 的最佳实践，应该将其视为一个资源：

```
resources :articles
```

声明资源有很多附属作用。资源的创建与使用请阅读“Rails 路由全解”一文。

注意，不管记录是否存在，使用简短形式的 form_for 调用都很方便。记录辨别技术很智能，会调用 record.new_record? 方法检查是否为新记录；而且还能自动选择正确的提交地址，根据对象所属的类生成 name 属性的值。

Rails 还会自动设置 class 和 id 属性。在新建文章的表单中，id 和 class 属性的值都是 new_article。如果编辑 ID 为 23 的文章，表单的 class 为 edit_article，id 为 edit_article_23。为了行文简洁，后文会省略这些属性。

如果在模型中使用单表继承（single-table inheritance，简称 STI），且只有父类声明为资源，子类就不能依赖记录辨别技术，必须指定模型名，:url 和 :method 选项。

（18）处理命名空间

如果在路由中使用了命名空间，form_for 方法也有相应的简写形式。如果程序中有个 admin 命名空间，表单可以写成：

```
form_for [:admin, @article]
```

这个表单会提交到命名空间 admin 中的 ArticlesController（更新文章时提交到 admin_article_path(@article)）。如果命名空间有很多层，句法类似：

form_for [:admin, :management, @article]

关于 Rails 路由的详细信息以及相关的约定，请阅读“Rails 路由全解”一文。

（19）请求方法处理

Rails 框架建议使用 REST 架构设计程序，因此除了 GET 和 POST 请求之外，还要处理 PATCH 和 DELETE 请求。但是大多数浏览器不支持从表单中提交 GET 和 POST 之外的请求。

为了解决这个问题，Rails 使用 POST 请求进行模拟，并在表单中加入一个名为 _method 的隐藏字段，其值表示真正希望使用的请求方法：

form_tag(search_path, method: "patch")

生成的 HTML 为：

```
<form accept-charset="UTF-8" action="/search" method="post">
  <div style="margin:0;padding:0">
```

```
    <input name="_method" type="hidden" value="patch" />
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  ...
```

处理提交的数据时，Rails 以 _method 的值为准，发起相应类型的请求（在这个例子中是 PATCH 请求）。

3.2.3. 选择列表

HTML 中的选择列表往往需要编写很多标记语言（每个选项都要创建一个 option 元素），因此最适合自动生成。

选择列表的标记语言如下所示：

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

这个列表列出了一组城市名。在程序内部只需要处理各选项的 ID，因此把各选项的 value

属性设为 ID。下面来看一下 Rails 为我们提供了哪些帮助方法。

（20）select 和 option 标签

最常见的帮助方法是 select_tag，如其名所示，其作用是生成 select 标签，其中可以包含一个由选项组成的字符串：

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

这只是个开始，还无法动态生成 option 标签。option 标签可以使用帮助方法 options_for_select 生成：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...]) %>
```

生成的 HTML 为：

```
<option value="1">Lisbon</option>

<option value="2">Madrid</option>

...
```

options_for_select 方法的第一个参数是一个嵌套数组，每个元素都有两个子元素：选项的文本（城市名）和选项的 value 属性值（城市 ID）。选项的 value 属性值会提交到控制器中。ID 的值经常表示数据库对象，但这个例子除外。

知道上述用法后，就可以结合 select_tag 和 options_for_select 两个方法生成所需的完整 HTML 标记：

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

options_for_select 方法还可预先选中一个选项，通过第二个参数指定：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...], 2) %>
```

生成的 HTML 如下：

```
<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

当 Rails 发现生成的选项 value 属性值和指定的值一样时，就会在这个选项中加上 selected 属性。

options_for_select 方法的第二个参数必须完全和需要选中的选项 value 属性值相等。如果 value 的值是整数 2，就不能传入字符串 "2"，必须传入数字 2。注意，从 params 中获取的值都是字符串。

使用 Hash 可以为选项指定任意属性：

```
<%= options_for_select([['Lisbon', 1, {'data-size' => '2.8 million'}], ['Madrid', 2, {'data-size' => '3.2 million'}]], 2) %>
```

生成的 HTML 如下：

```
<option value="1" data-size="2.8 million">Lisbon</option>
<option value="2" selected="selected" data-size="3.2 million">Madrid</option>
...
```

（21）模型的选择列表

大多数情况下，表单的控件用于处理指定的数据库模型，正如你所期望的，Rails 为此提供了很多用于生成选择列表的帮助方法。和其他表单帮助方法一样，处理模型时要去掉 select_tag 中的 _tag：

```
# controller:

@person = Person.new(city_id: 2)

# view:

<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 2], ...]) %>
```

注意，第三个参数，选项数组，和传入 options_for_select 方法的参数一样。这种帮助方法的一个好处是，无需关心如何预先选中正确的城市，只要用户设置了所在城市，Rails 就会读取 @person.city_id 的值，为你代劳。

和其他帮助方法一样，如果要在绑定到 @person 对象上的表单构造器上使用 select 方法，相应的句法为：

```
# select on a form builder
```

```
<%= f.select(:city_id, ...) %>
```

select 帮助方法还可接受一个代码块：

```
<%= f.select(:city_id) do %>
  <% [['Lisbon', 1], ['Madrid', 2]].each do |c| -%>
    <%= content_tag(:option, c.first, value: c.last) %>
  <% end %>
<% end %>
```

如果使用 select 方法（或类似的帮助方法，例如 collection_select 和 select_tag）处理 belongs_to 关联，必须传入外键名（在上例中是 city_id），而不是关联名。如果传入的是 city 而不是 city_id，把 params 传给 Person.new 或 update 方法时，会抛出异常：ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750)。这个要求还可以这么理解，表单帮助方法只能编辑模型的属性。此外还要知道，允许用户直接编辑外键具有潜在地安全隐患。

（22）根据任意对象组成的集合创建 option 标签

使用 options_for_select 方法生成 option 标签必须使用数组指定各选项的文本和值。如果有个 City 模型，想根据模型实例组成的集合生成 option 标签应该怎么做呢？一种方法是遍历集合，创建一个嵌套数组：

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>

<%= options_for_select(cities_array) %>
```

这种方法完全可行，但 Rails 提供了一个更简洁的帮助方法：options_from_collection_for_select。这个方法接受一个由任意对象组成的集合，以及另外两个参数：获取选项文本和值使用的方法。

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

从这个帮助方法的名字中可以看出，它只生成 option 标签。如果想生成可使用的选择列表，和 options_for_select 方法一样要结合 select_tag 方法一起使用。select 方法集成了 select_tag 和 options_for_select 两个方法，类似地，处理集合时，可以使用 collection_select 方法，它集成了 select_tag 和 options_from_collection_for_select 两个方法。

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

options_from_collection_for_select 对 collection_select 来说，就像 options_for_select 与 select 的关系一样。

传入 options_for_select 方法的子数组第一个元素是选项文本，第二个元素是选项的值，但传入 options_from_collection_for_select 方法的第一个参数是获取选项值的方法，第二个才是获取选项文本的方法。

（23）时区和国家选择列表

要想在 Rails 程序中实现时区相关的功能，就得询问用户其所在的时区。设定时区时可以使用 collection_select 方法根据预先定义的时区对象生成一个选择列表，也可以直接使用

time_zone_select 帮助方法：

```
<%= time_zone_select(:person, :time_zone) %>
```

如果想定制时区列表，可使用 time_zone_options_for_select 帮助方法。这两个方法可接受的参数请查阅 API 文档。

以前 Rails 还内置了 country_select 帮助方法，用于创建国家选择列表，但现在已经被提取出来做成了 country_select gem。使用这个 gem 时要注意，是否包含某个国家还存在争议（正因为此，Rails 才不想内置）。

3.2.4. 日期和时间表单

你可以选择不使用生成 HTML5 日期和时间输入框的帮助方法，而使用生成日期和时间选择列表的帮助方法。生成日期和时间选择列表的帮助方法和其他表单帮助方法有两个重要的不同点：

日期和时间不在单个 input 元素中输入，而是每个时间单位都有各自的元素，因此在 params 中就没有单个值能表示完整的日期和时间；

其他帮助方法通过 _tag 后缀区分是独立的帮助方法还是操作模型对象的帮助方法。对日期和时间帮助方法来说，

select_date、select_time 和 select_datetime 是独立的帮助方法，date_select、time_select 和 datetime_select 是相应的操作模型对象的帮助方法。

这两类帮助方法都会为每个时间单位（年，月，日等）生成各自的选择列表。

（24）独立的帮助方法

select_* 这类帮助方法的第一个参数是 Date、Time 或 DateTime 类的实例，并选中指定的日期时间。

如果不指定，就使用当前日期时间。例如：

```
<%= select_date Date.today, prefix: :start_date %>
```

生成的 HTML 如下（为了行为简便，省略了各选项）：

```
<select id="start_date_year" name="start_date[year]"> ... </select>

<select id="start_date_month" name="start_date[month]"> ... </select>

<select id="start_date_day" name="start_date[day]"> ... </select>
```

上面各控件会组成 params[:start_date]，其中包含名为 :year、:month 和 :day 的键。如果想获取 Time 或 Date 对象，要读取各时间单位的值，然后传入适当的构造方法中，例如：

```
Date.civil(params[:start_date][:year].to_i,
params[:start_date][:month].to_i, params[:start_date][:day].to_i)
```

:prefix 选项的作用是指定从 params 中获取各时间组成部分的键名。在上例中，:prefix 选项的值是 start_date。如果不指定这个选项，就是用默认值 date。

（25）模型对象的帮助方法

select_date 方法在更新或创建 Active Record 对象的表单中有点力不从心，因为 Active Record 期望 params 中的每个元素都对应一个属性。

用于处理模型对象的日期和时间帮助方法会提交一个名字特殊的参数，Active Record 看到这个参数时就知道必须和其他参数结合起来传递给字段类型对应的构造方法。

例如：

<%= date_select :person, :birth_date %>

生成的 HTML 如下（为了行为简介，省略了各选项）：

<select id="person_birth_date_1i" name="person[birth_date(1i)]"> ... </select>

<select id="person_birth_date_2i" name="person[birth_date(2i)]"> ... </select>

<select id="person_birth_date_3i" name="person[birth_date(3i)]"> ... </select>

创建的 params Hash 如下：

{'person' => {'birth_date(1i)' => '2008', 'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}

传递给 Person.new（或 update）方法时，Active Record 知道这些参数应该结合在一起组成 birth_date 属性，使用括号中的信息决定传给 Date.civil 等方法的顺序。

（26）通用选项

这两种帮助方法都使用同一组核心函数生成各选择列表，因此使用的选项基本一样。

默认情况下，Rails 生成的年份列表包含本年前后五年。

如果这个范围不能满足需求，可以使用 :start_year 和 :end_year 选项指定。更详细的可用选项列表请参阅 API 文档。

基本原则是，使用 date_select 方法处理模型对象，其他情况都使用 select_date 方法，例如在搜索表单中根据日期过滤搜索结果。

很多时候内置的日期选择列表不太智能，不能协助用户处理日期和星期几之间的对应关系。

3.3. 参数命名约定

从前几节可以看出，表单提交的数据可以直接保存在 params Hash 中，或者嵌套在子 Hash 中。例如，在 Person 模型对应控制器的 create 动作中，params[:person] 一般是一个 Hash，保存创建 Person 实例的所有属性。params Hash 中也可以保存数组，或由 Hash

组成的数组，等等。

HTML 表单基本上不能处理任何结构化数据，提交的只是由普通的字符串组成的键值对。在程序中使用的数组参数和 Hash 参数是通过 Rails 的参数命名约定生成的。

如果想快速试验本节中的示例，可以在控制台中直接调用 Rack 的参数解析器。例如： T> ruby

TIP: Rack::Utils.parse_query "name=fred&phone=0123456789"

TIP: # => {"name"=>"fred", "phone"=>"0123456789"}

TIP:

3.3.1. 基本结构

数组和 Hash 是两种基本结构。

获取 Hash 中值的方法和 params 一样。如果表单中包含以下控件：

<input id="person_name" name="person[name]" type="text" value="Henry"/>

得到的 params 值为：

{'person' => {'name' => 'Henry'}}

在控制器中可以使用 params[:person][:name] 获取提交的值。

Hash 可以随意嵌套，不限制层级，例如：

<input id="person_address_city" name="person[address][city]" type="text" value="New York"/>

得到的 params 值为：

{'person' => {'address' => {'city' => 'New York'}}}

一般情况下 Rails 会忽略重复的参数名。如果参数名中包含空的方括号（[]），Rails 会将其组建成一个数组。如果想让用户输入多个电话号码，在表单中可以这么做：

<input name="person[phone_number][]" type="text"/>

```
<input name="person[phone_number][]" type="text"/>

<input name="person[phone_number][]" type="text"/>
```

得到的 params[:person][:phone_number] 就是一个数组。

3.3.2. 结合在一起使用

上述命名约定可以结合起来使用，让 params 的某个元素值为数组（如前例），或者由 Hash 组成的数组。例如，使用下面的表单控件可以填写多个地址：

```
<input name="addresses[][line1]" type="text"/>

<input name="addresses[][line2]" type="text"/>

<input name="addresses[][city]" type="text"/>
```

得到的 params[:addresses] 值是一个由 Hash 组成的数组，Hash 中的键包括 line1、line2 和 city。如果 Rails 发现输入框的 name 属性值已经存在于当前 Hash 中，就会新建一个 Hash。

不过有个限制，虽然 Hash 可以嵌套任意层级，但数组只能嵌套一层。如果需要嵌套多层数组，可以使用 Hash 实现。例如，如果想创建一个包含模型对象的数组，可以创建一个 Hash，以模型对象的

ID、数组索引或其他参数为键。

数组类型参数不能很好的在 check_box 帮助方法中使用。根据 HTML 规范，未选中的复选框不应该提交值。但是不管是否选中都提交值往往更便于处理。为此 check_box 方法额外创建了一个同名的隐藏 input 元素。如果没有选中复选框，只会提交隐藏 input 元素的值，如果选中则同时提交两个值，但复选框的值优先级更高。处理数组参数时重复提交相同的参数会让 Rails 迷惑，因为对 Rails 来说，见到重复的 input 值，就会创建一个新数组元素。所以更推荐使用 check_box_tag 方法，或者用 Hash 代替数组。

3.3.3. 表单帮助方法

前面几节并没有使用 Rails 提供的表单帮助方法。你可以自己创建 input 元素的 name 属性，然后直接将其传递给 text_field_tag 等帮助方法。但是 Rails 提供了更高级的支持。本节介绍 form_for 和 fields_for 方法的 name 参数以及 :index 选项。

你可能会想编写一个表单，其中有很多字段，用于编辑某人的所有地址。例如：

```
<%= form_for @person do |person_form| %>
```

```
<%= person_form.text_field :name %>

<% @person.addresses.each do |address| %>

  <%= person_form.fields_for address, index: address.id do |address_form|%>

    <%= address_form.text_field :city %>

  <% end %>

<% end %>

<% end %>
```

假设这个人有两个地址，ID 分别为 23 和 45。那么上述代码生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/people/1" class="edit_person" id="edit_person_1" method="post">

<input id="person_name" name="person[name]" type="text" />

<input id="person_address_23_city" name="person[address][23][city]" type="text" />

<input id="person_address_45_city"
```

name="person[address][45][city]" type="text" />

</form>

得到的 params Hash 如下：

{'person' => {'name' => 'Bob', 'address' => {'23' => {'city' => 'Paris'}, '45' => {'city' => 'London'}}}}

Rails 之所以知道这些输入框中的值是 person Hash 的一部分，是因为我们在第一个表单构造器上调用了 fields_for 方法。指定 :index 选项的目的是告诉 Rails，其中的输入框 name 属性值不是 person[address][city]，而要在 address 和 city 索引之间插入 :index 选项对应的值（放入方括号中）。这么做很有用，因为便于分辨要修改的 Address 记录是哪个。:index 选项的值可以是具有其他意义的数字、字符串，甚至是 nil（此时会新建一个数组参数）。

如果想创建更复杂的嵌套，可以指定 name 属性的第一部分（前例中的 person[address]）：

<%= fields_for 'person[address][primary]', address, index: address do |address_form| %>

```
    <%= address_form.text_field :city %>

<% end %>
```

生成的 HTML 如下：

```
<input id="person_address_primary_1_city" name="person[address][primary][1][city]" type="text" value="bologna" />
```

一般来说，最终得到的 name 属性值是 fields_for 或 form_for 方法的第一个参数加 :index 选项的值再加属性名。:index 选项也可直接传给 text_field 等帮助方法，但在表单构造器中指定可以避免代码重复。

为了简化句法，还可以不使用 :index 选项，直接在第一个参数后面加上 []。这么做和指定 index: address 选项的作用一样，因此下面这段代码

```
<%= fields_for 'person[address][primary][]', address do |address_form| %>

    <%= address_form.text_field :city %>
```

<% end %>

生成的 HTML 和前面一样。

4. 渲染视图

你可能已经听说过 Rails 的开发原则之一是“多约定，少配置”。默认渲染视图的处理就是这一原则的完美体现。默认情况下，Rails 中的控制器会渲染路由对应的视图。例如，有如下的 BooksController 代码：

```
class BooksController < ApplicationController
end
```

在路由文件中有如下定义：

```
resources :books
```

而且有个名为 app/views/books/index.html.erb 的视图文件：

```
<h1>Books are coming soon!</h1>
```

那么，访问 /books 时，Rails 会自动渲染视图 app/views/books/index.html.erb，网页中会看到显示有“Books are coming soon!”。

网页中显示这些文字没什么用，所以后续你可能会创建一个 Book 模型，然后在 BooksController 中添加 index 动作：

```
class BooksController < ApplicationController
```

```
    def index
      @books = Book.all
    end
  end
```

注意，基于“多约定，少配置”原则，在 index 动作末尾并没有指定要渲染视图，Rails 会自动在控制器的视图文件夹中寻找 action_name.html.erb 模板，然后渲染。在这个例子中，Rails 渲染的是 app/views/books/index.html.erb 文件。

如果要在视图中显示书籍的属性，可以使用 ERB 模板：

```
<h1>Listing Books</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Summary</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @books.each do |book| %>
```

```
    <tr>
      <td><%= book.title %></td>
      <td><%= book.content %></td>
      <td><%= link_to "Show", book %></td>
      <td><%= link_to "Edit", edit_book_path(book) %></td>
      <td><%= link_to "Remove", book, method: :delete, data: { confirm: "Are you sure?" } %></td>
    </tr>
  <% end %>
  </table>

  <br>

  <%= link_to "New book", new_book_path %>
```

真正处理渲染过程的是 ActionView::TemplateHandlers 的子类。本文不做深入说明，但要知道，文件的扩展名决定了要使用哪个模板处理程序。从 Rails 2 开始，ERB 模板（含有嵌入式 Ruby 代码的 HTML）的标准扩展名是 .erb，Builder 模板（XML 生成器）的标准扩展名是 .builder。

4.1. render

调用 render 方法，向浏览器发送一个完整的响应；

大多数情况下，ActionController::Base#render 方法都能满足需求，而且还有多种定制方式，可以渲染 Rails 模板的默认视图、指定的模板、文件、行间代码或者什么也不渲染。渲染的内容格式可以是文本，JSON 或 XML。而且还可以设置响应的内容类型和 HTTP 状态码。

如果不想使用浏览器直接查看调用 render 方法得到的结果，可以使用 render_to_string 方法。render_to_string 和 render 的用法完全一样，不过不会把响应发送给浏览器，而是直接返回字符串。

4.1.1. 什么都不渲染

或许 render 方法最简单的用法是什么也不渲染：

```
render nothing: true
```

如果使用 cURL 查看请求，会得到一些输出：

```
$ curl -i 127.0.0.1:3000/books
HTTP/1.1 200 OK
Connection: close
Date: Sun, 24 Jan 2010 09:25:18 GMT
Transfer-Encoding: chunked
Content-Type: */*; charset=utf-8
X-Runtime: 0.014297
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
```

```
Cache-Control: no-cache

$
```

可以看到，响应的主体是空的（Cache-Control 之后没有数据），但请求本身是成功的，因为 Rails 把响应码设为了“200 OK”。调用 render 方法时可以设置 :status 选项修改状态码。这种用法可在 Ajax 请求中使用，因为此时只需告知浏览器请求已经完成。

或许不应该使用 render :nothing，而要用后面介绍的 head 方法。head 方法用起来更灵活，而且只返回 HTTP 报头。

4.1.2. 渲染动作的视图

如果想渲染同个控制器中的其他模板，可以把视图的名字传递给 render 方法：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render "edit"
  end
end
```

如果更新失败，会渲染同个控制器中的 edit.html.erb 模板。

如果不想用字符串，还可使用 Symbol 指定要渲染的动作：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render :edit
  end
end
```

4.1.3. 渲染其他控制器中的动作模板

如果想渲染其他控制器中的模板该怎么做呢？

还是使用 render 方法，指定模板的完整路径即可。例如，如果控制器 AdminProductsController 在 app/controllers/admin 文件夹中，可使用下面的方式渲染 app/views/products 文件夹中的模板：

```
render "products/show"
```

因为参数中有个斜线，所以 Rails 知道这个视图属于另一个控制器。如果想让代码的意图更明显，可以使用 :template 选项（Rails 2.2 及先前版本必须这么做）：

```
render template: "products/show"
```

4.1.4. 小结

上述三种渲染方式的作用其实是一样的。在 BooksController 控制器的 update 动作中，如果更新失败后想渲染 views/books 文件夹中的 edit.html.erb 模板，下面这些用法都能达到这个目的：

```
render :edit
render action: :edit
render "edit"
render "edit.html.erb"
render action: "edit"
render action: "edit.html.erb"
render "books/edit"
render "books/edit.html.erb"
render template: "books/edit"
render template: "books/edit.html.erb"
render "/path/to/rails/app/views/books/edit"
render "/path/to/rails/app/views/books/edit.html.erb"
render file: "/path/to/rails/app/views/books/edit"
render file: "/path/to/rails/app/views/books/edit.html.erb"
```

你可以根据自己的喜好决定使用哪种方式，总的原则是，使用符合代码意图的最简单方式。

4.1.5. 渲染类型

1.1.1.1 渲染文本

调用 render 方法时指定 :plain 选项，可以把没有标记语言的纯文本发给浏览器：

```
render plain: "OK"
```

渲染纯文本主要用于 Ajax 或无需使用 HTML 的网络服务。

默认情况下，使用 :plain 选项渲染纯文本，不会套用程序的布局。如果想使用布局，可以指定 layout: true 选项。

1.1.1.2 渲染 HTML

调用 render 方法时指定 :html 选项，可以把 HTML 字符串发给浏览器：

```
render html: "<strong>Not Found</strong>".html_safe
```

这种方法可用来渲染 HTML 片段。如果标记很复杂，就要考虑使用模板文件了。

如果字符串对 HTML 不安全，会进行转义。

1.1.1.3 渲染 JSON

JSON 是一种 JavaScript 数据格式，很多 Ajax 库都用这种格式。Rails 内建支持把对象转换成 JSON，经渲染后再发送给浏览器。

```
render json: @product
```

在需要渲染的对象上无需调用 to_json 方法，如果使用了 :json 选项，render 方法会自动调用 to_json。

1.1.1.4 渲染任意文件

render 方法还可渲染程序之外的视图（或许多个程序共用一套视图）：

```
render "/u/apps/warehouse_app/current/app/views/products/show"
```

因为参数以斜线开头，所以 Rails 将其视为一个文件。如果想让代码的意图更明显，可以使用 :file 选项（Rails 2.2+ 必须这么做）

```
render file: "/u/apps/warehouse_app/current/app/views/products/show"
```

:file 选项的值是文件系统中的绝对路径。当然，你要对使用的文件拥有相应权限。

默认情况下，渲染文件时不会使用当前程序的布局。如果想让 Rails 把文件套入布局，要指定 layout: true 选项。

如果在 Windows 中运行 Rails，就必须使用 :file 选项指定文件的路径，因为 Windows 中的文件名和 Unix 格式不一样。

1.1.1.5 渲染 XML

Rails 也内建支持把对象转换成 XML，经渲染后再发回给调用者：

```
render xml: @product
```

在需要渲染的对象上无需调用 to_xml 方法，如果使用了 :xml 选项，render 方法会自动调用 to_xml。

1.1.1.6 渲染普通的 JavaScript

Rails 能渲染普通的 JavaScript：

```
render js: "alert('Hello Rails');"
```

这种方法会把 MIME 设为 text/javascript，再把指定的字符串发给浏览器。

1.1.1.7 渲染原始的主体

调用 render 方法时使用 :body 选项，可以不设置内容类型，把原始的内容发送给浏览器：

```
render body: "raw"
```

只有不在意内容类型时才可使用这个选项。大多数时候，使用 :plain 或 :html 选项更合适。

如果没有修改，这种方式返回的内容类型是 text/html，因为这是 Action Dispatch 响应默认使用的内容类型。

4. 1. 6. render 选项

1.1.1.8 :content_type

默认情况下，Rails 渲染得到的结果内容类型为 text/html；

如果使用 :json 选项，内容类型为 application/json；

如果使用 :xml 选项，内容类型为 application/xml。

如果需要修改内容类型，可使用 :content_type 选项

```
render file: filename, content_type: "application/rss"
```

1.1.1.9 :layout 选项

render 方法的大多数选项渲染得到的结果都会作为当前布局的一部分显示。后文会详细介绍布局。

:layout 选项告知 Rails，在当前动作中使用指定的文件作为布局：

render layout: "special_layout"

也可以告知 Rails 不使用布局：

render layout: false

1.1.1.10 :location 选项

:location 选项可以设置 HTTP Location 报头：

render xml: photo, location: photo_url(photo)

1.1.1.11 :status 选项

Rails 会自动为生成的响应附加正确的 HTTP 状态码（大多数情况下是 200 OK）。使用 :status 选项可以修改状态码：

render status: 500

render status: :forbidden

Rails 能理解数字状态码和对应的符号，如下所示：

响应类别 HTTP 状态码 符号

信息 100 :continue

101 :switching_protocols

102 :processing

成功 200 :ok

201 :created

202 :accepted

203 :non_authoritative_information

204 :no_content

205 :reset_content

206 :partial_content

207 :multi_status

208 :already_reported

226 :im_used

重定向 300 :multiple_choices

301 :moved_permanently

302 :found

303 :see_other

304 :not_modified

305 :use_proxy

306 :reserved

307 :temporary_redirect

308 :permanent_redirect

客户端错误 400 :bad_request

401 :unauthorized

402 :payment_required

403 :forbidden

404 :not_found

405 :method_not_allowed

406 :not_acceptable

407 :proxy_authentication_required

408 :request_timeout

409 :conflict

410 :gone

411 :length_required

412 :precondition_failed

413 :request_entity_too_large

414 :request_uri_too_long

415 :unsupported_media_type

416 :requested_range_not_satisfiable

417 :expectation_failed

422 :unprocessable_entity

423 :locked

424 :failed_dependency

426 :upgrade_required

428 :precondition_required

429 :too_many_requests

431 :request_header_fields_too_large

服务器错误 500 :internal_server_error

501 :not_implemented

502 :bad_gateway

503 :service_unavailable

504 :gateway_timeout

505 :http_version_not_supported

506 :variant_also_negotiates

507 :insufficient_storage

508 :loop_detected

510 :not_extended

511 :network_authentication_required

4.1.7. 查找布局

查找布局时，Rails 首先查看 app/views/layouts 文件夹中是否有和控制器同名的文件。

例如，渲染 PhotosController 控制器中的动作会使用 app/views/layouts/photos.html.erb（或 app/views/layouts/photos.builder）。

如果没找到针对控制器的布局，Rails 会使用 app/views/layouts/application.html.erb 或

app/views/layouts/application.builder。

如果没有 .erb 布局，Rails 会使用 .builder 布局（如果文件存在）。

Rails 还提供了多种方法用来指定单个控制器和动作使用的布局。

1.1.1.12 指定控制器所用布局

在控制器中使用 layout 方法，可以改写默认使用的布局约定。例如：

```
class ProductsController < ApplicationController

layout "inventory"

  #...

end
```

这么声明之后，ProductsController 渲染的所有视图都将使用 app/views/layouts/inventory.html.erb 文件作为布局。

要想指定整个程序使用的布局，可以在 ApplicationController 类中使用 layout 方法：

```
class ApplicationController < ActionController::Base

  layout "main"
```

```
  #...
end
```

这么声明之后，整个程序的视图都会使用 app/views/layouts/main.html.erb 文件作为布局。

1.1.1.13 运行时选择布局

可以使用一个 Symbol，在处理请求时选择布局：

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
    @product = Product.find(params[:id])
  end

  private
    def products_layout
      @current_user.special? ? "special" : "products"
    end

end
```

如果当前用户是特殊用户，会使用一个特殊布局渲染产品视图。

还可使用行间方法，例如 Proc，决定使用哪个布局。如果使用 Proc，其代码块可以访问 controller 实例，这样就能根据当前请求决定使用哪个布局：

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr? ? "popup" : "application" }
end
```

1.1.1.14 条件布局

在控制器中指定布局时可以使用 :only 和 :except 选项。这两个选项的值可以是一个方法名或者一个方法名数组，这些方法都是控制器中的动作：

```
class ProductsController < ApplicationController
  layout "product", except: [:index, :rss]
end
```

这么声明后，除了 rss 和 index 动作之外，其他动作都使用 product 布局渲染视图。

1.1.1.15 布局继承

布局声明按层级顺序向下顺延，专用布局比通用布局优先级高。例如：

application_controller.rb

```
class ApplicationController < ActionController::Base
```

```
    layout "main"
end
```

posts_controller.rb

```
class PostsController < ApplicationController
end
```

special_posts_controller.rb

```
class SpecialPostsController < PostsController
    layout "special"
end
```

old_posts_controller.rb

```
class OldPostsController < SpecialPostsController
    layout false

    def show
      @post = Post.find(params[:id])
    end

    def index
      @old_posts = Post.older
      render layout: "old"
    end
    # ...
end
```

在这个程序中：

一般情况下，视图使用 main 布局渲染；

PostsController#index 使用 main 布局；

SpecialPostsController#index 使用 special 布局；

OldPostsController#show 不用布局；

OldPostsController#index 使用 old 布局；

1.1.1.16 避免双重渲染错误

大多数 Rails 开发者迟早都会看到一个错误消息：Can only render or redirect once per action（动作只能渲染或重定向一次）。这个提示很烦人，也很容易修正。出现这个错误的原因是，没有理解 render 的工作原理。

例如，下面的代码会导致这个错误：

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
```

```
    render action: "regular_show"
  end
```

如果 @book.special? 的结果是 true ,Rails 开始渲染 ,把 @book 变量导入 special_show 视图中。但是，show 动作并不会就此停止运行，当 Rails 运行到动作的末尾时，会渲染 regular_show 视图，导致错误出现。解决的办法很简单，确保在一次代码运行路线中只调用一次 render 或 redirect_to 方法。有一个语句可以提供帮助，那就是 and return。下面的代码对上述代码做了修改：

```
  def show
    @book = Book.find(params[:id])
    if @book.special?
      render action: "special_show" and return
    end
    render action: "regular_show"
  end
```

千万别用 && return 代替 and return，因为 Ruby 语言操作符优先级的关系，&& return 根本不起作用。

注意 ,ActionController 能检测到是否显式调用了 render 方法 ,所以下面这段代码不会出错：

```
  def show
    @book = Book.find(params[:id])
```

```
    if @book.special?
      render action: "special_show"
    end
  end
```

如果 @book.special? 的结果是 true，会渲染 special_show 视图，否则就渲染默认的 show 模板。

4.1.8. redirect_to

调用 redirect_to 方法，向浏览器发送一个 HTTP 重定向状态码；

响应 HTTP 请求的另一种方法是使用 redirect_to。如前所述，render 告诉 Rails 构建响应时使用哪个视图（以及其他静态资源）。redirect_to 做的事情则完全不同：告诉浏览器向另一个地址发起新请求。例如，在程序中的任何地方使用下面的代码都可以重定向到 photos 控制器的 index 动作：

```
redirect_to photos_url
```

redirect_to 方法的参数与 link_to 和 url_for 一样。有个特殊的重定向，返回到前一个页面：

```
redirect_to :back
```

4.1.9. 设置不同的重定向状态码

调用 redirect_to 方法时，Rails 会把 HTTP 状态码设为 302，即临时重定向。如果想使用其他的状态码，例如 301（永久重定向），可以设置 :status 选项：

```
redirect_to photos_path, status: 301
```

和 render 方法的 :status 选项一样，redirect_to 方法的 :status 选项同样可使用数字状态码或符号。

4.1.10. render 和 redirect_to 的区别

有些经验不足的开发者会认为 redirect_to 方法是一种 goto 命令，把代码从一处转到别处。这么理解是不对的。执行到 redirect_to 方法时，代码会停止运行，等待浏览器发起新请求。你需要告诉浏览器下一个请求是什么，并返回 302 状态码。

下面通过实例说明。

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
```

```
    if @book.nil?
      render action: "index"
    end
  end
```

在这段代码中，如果 @book 变量的值为 nil 很可能会出问题。记住，render :action 不会执行目标动作中的任何代码，因此不会创建 index 视图所需的 @books 变量。修正方法之一是不渲染，使用重定向：

```
  def index
    @books = Book.all
  end

  def show
    @book = Book.find_by(id: params[:id])
    if @book.nil?
      redirect_to action: :index
    end
  end
```

这样修改之后，浏览器会向 index 动作发起新请求，执行 index 方法中的代码，一切都能正常运行。

这种方法有个缺点，增加了浏览器的工作量。浏览器通过 /books/1 向 show 动作发起请求，

控制器做了查询，但没有找到对应的图书，所以返回 302 重定向响应，告诉浏览器访问 /books/。浏览器收到指令后，向控制器的 index 动作发起新请求，控制器从数据库中取出所有图书，渲染 index 模板，将其返回浏览器，在屏幕上显示所有图书。

在小型程序中，额外增加的时间不是个问题。如果响应时间很重要，这个问题就值得关注了。下面举个虚拟的例子演示如何解决这个问题：

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    @books = Book.all
    flash.now[:alert] = "Your book was not found"
    render "index"
  end
end
```

在这段代码中，如果指定 ID 的图书不存在，会从模型中取出所有图书，赋值给 @books 实例变量，然后直接渲染 index.html.erb 模板，并显示一个 Flash 消息，告知用户出了什么问题。

4.2. head

调用 head 方法，向浏览器发送只含报头的响应；

4.2.1. 使用 head 构建只返回报头的响应

head 方法可以只把报头发送给浏览器。还可使用意图更明确的 render :nothing 达到同样的目的。head 方法的参数是 HTTP 状态码的符号形式（参见前文表格），选项是一个 Hash，指定报头名和对应的值。例如，可以只返回报错的报头：

```
head :bad_request
```

生成的报头如下：

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

或者使用其他 HTTP 报头提供其他信息：

head :created, location: photo_path(@photo)

生成的报头如下：

HTTP/1.1 201 Created

Connection: close

Date: Sun, 24 Jan 2010 12:16:44 GMT

Transfer-Encoding: chunked

Location: /photos/1

Content-Type: text/html; charset=utf-8

X-Runtime: 0.083496

Set-Cookie: _blog_session=...snip...; path=/; HttpOnly

Cache-Control: no-cache

5. respond_to

5.1. 请求头

"text/plain" => :text

"text/html" => :html

"application/xhtml+xml" => :html

"text/javascript" => :js

"application/javascript" => :js

"application/x-javascript" => :js

"text/calendar" => :ics

"text/csv" => :csv

"application/xml" => :xml

"text/xml" => :xml

"application/x-xml" => :xml

"text/yaml" => :yaml

"application/x-yaml" => :yaml

"application/rss+xml" => :rss

"application/atom+xml" => :atom

"application/json" => :json

"text/x-json" => :json

5.2. 设置

@request.accept = "text/javascript" #=> request JS

方法二：

```
// get an instance to a responder from the base class
var responder = get_responder()

// register html to render in the default way
// (by way of the views and conventions)
responder.register('html')

// register json as well. the argument to .json is the second
// argument to method_missing ('json' is the first), which contains
// optional ways to configure the response. In this case, serialize as json.
responder.register('json', renderOptions)
```

5.3. 例子

respond_to 可以讓你根據客戶端要求的格式進行不同的格式回應，以 RESTful 與 Rails 中完成的應用程式首頁為例，若想要客戶端在請求

http://localhost:3000/bookmarks.html、

http://localhost:3000/bookmarks.xml、

http://localhost:3000/bookmarks.json 時，

分別給 HTML、XML、JSON 格式回應，可以如下修改：

```
class BookmarksController < ApplicationController
    def index
        @pages = Page.all
        respond_to do |format|
            format.html
            format.xml { render :xml => @pages }
            format.json { render :json => @pages }
        end
    end
    ...
end
```