7. 视图（Views）

Phoenix views有两个主要工作。首先，它们渲染模板（包括布局）。渲染时调用的核心函数 `render/3` 在Phoenix本身的 `Phoenix.View` 模块中定义。视图还提供了获取raw数据并使模板更易于使用的功能。如果您熟悉decorators或者facade模式，这很类似。

渲染模板（Rendering Templates）

Phoenix从控制器到视图再到其渲染的模板都遵循严格的命名约定。 `PageController` 需要一个 `PageView` 渲染 `lib/hello_web/templates/page` 目录中的模板。如果我们愿意，可以更改Phoenix认定的模板根目录。Phoenix在 `lib/hello_web.ex` 中定义的 `HelloWeb` 模块提供了一个 `view/0` 函数。第一行 `view/0` 允许我们通过修改 `:root` 键的值来改变我们的根目录。

一个新创建的Phoenix应用有三个视图模块—— `ErrorView` 、 `LayoutView` 和 `PageView` ——它们都在 `lib/hello_web/views` 目录。

让我们快速看一下 `LayoutView` 。

```
defmodule HelloWeb.LayoutView do
  use HelloWeb, :view
end
```

这很简单。只有一行 `use HelloWeb, :view` 。这行调用了我们上面刚刚看到的 `view/0` 函数。除了允许我们更改模板根目录之外， `view/0` 还让 `Phoenix.View` 模块里的 `__using__` 宏发挥了作用。它还处理了我们应用中view模块需要的任何模块的imports或aliases。

在本教程的开头，我们提到views是用来存放在模板中要使用的函数的地方。让我们来试试看。

让我们打开应用的layout模板 `lib/hello_web/templates/layout/app.html.eex` ，然后更改此行，

```
<title>Hello · Phoenix Framework</title>
```

像这样调用一个 `title/0` 函数。

```
<title><%= title() %></title>
```

现在为我们的 `LayoutView` 添加一个 `title/0` 函数。

```
defmodule HelloWeb.LayoutView do
  use HelloWeb, :view

  def title do
    "Awesome New Title!"
  end
end
```

重新加载Welcome to Phoenix page页面时，应该会看到新标题。

`<%=` 和 `%>` 来自Elixir的EEx项目。它们将可执行的Elixir代码包含在模板中。 `=` 告诉EEx打印结果。如果 `=` 不存在，则EEx仍将执行代码，但是将没有输出。在我们的示例中，我们调用 `LayoutView` 里的 `title/0` 函数，并将输出打印到title标签中。

请注意，我们不需要使用 `HelloWeb.LayoutView` 来完整地调用 `title/0` 。因为 `LayoutView` 实际上可以进行渲染。实际上，Phoenix中的"templates"只是其视图模块上定义的函数。您可以通过临时删除 `lib/hello_web/templates/page/index.html.eex` 文件并将此函数子句添加到 `lib/hello_web/views/page_view.ex` 中的 `PageView` 模块来进行尝试。

```
defmodule HelloWeb.PageView do
  use HelloWeb, :view

  def render("index.html", assigns) do
    "rendering with assigns #{inspect Map.keys(assigns)}"
  end
end
```

现在，如果您使用 `mix phx.server` 和启动服务器，并访问http://localhost:4000，则应该在layout header下方看到以下文本，而不是main template页面：

```
rendering with assigns [:conn, :view_module, :view_template]
```

很整洁吧？在编译时，Phoenix会预编译所有 `*.html.eex` 模板，并将它们转换为各自视图模块上的 `render/2` 函数子句。在运行时，所有模板都已加载到内存中。无需进行磁盘读取，复杂的文件缓存或模板引擎计算。这也是为什么我们能够在 `LayoutView` 中定义诸如 `title/0` 之类的函数，并且它们立即在 `app.html.eex` 布局中可用的原因——对 `title/0` 的调用只是局部函数调用！

当我们使用 `use HelloWeb, :view` ，我们还将获得其他便利。由于 `view/0` 将 `HelloWeb.Router.Helpers` 设置别名为 `Routes` （在 `lib/hello_web.ex` 中可以看到），因此我们可以简单地使用 `Routes.*_path` 在模板中调用这些helpers。让我们通过更改"Welcome to Phoenix"页面的模板来了解其工作原理。

让我们打开 `lib/hello_web/templates/page/index.html.eex` 并找到该节。

```
<section class="phx-hero">
  <h1><%= gettext "Welcome to %{name}!", name: "Phoenix" %></h1>
  <p>A productive web framework that<br/>does not compromise speed or maintainability.</p>
</section>
```

然后，我们添加一行返回到同一页面的链接。（目的是查看路径助手在模板中的响应方式，而不是添加任何功能。）

```
<section class="phx-hero">
  <h1><%= gettext "Welcome to %{name}!", name: "Phoenix" %></h1>
  <p>A productive web framework that<br/>does not compromise speed or maintainability.</p>
  <p><a href="<%= Routes.page_path(@conn, :index) %>">Link back to this page</a></p>
</div>
```

现在，我们可以重新加载页面并查看源代码，来看我们得到什么。

```
<a href="/">Link back to this page</a>
```

很棒， `Routes.page_path/2` 被求值为 `/` ，正如我们期望的那样，我们只需要使用 `Phoenix.View` 中设置的别名即可。

如果您碰巧需要访问视图，控制器或模板之外的路径助手，则可以使用全名来调用它们，例如， `HelloWeb.Router.Helpers.page_path(@conn, :index)` ，也可以在调用模块中自己定义别名，通过在要使用的模块中定义 `alias HelloWeb.Router.Helpers, as: Routes` ，然后调用，例如 `Routes.page_path(@conn, :index)` 。

关于Views的更多（More About Views）

您可能想知道视图如何与模板紧密配合。

`Phoenix.View` 模块获得了访问模板行为的途径，是通过 `use Phoenix.Template` 这一行中的 `__using__/1` 宏。 `Phoenix.Template` 提供了许多对模板有用的的便捷方法——查找模板，提取模板的名称和路径等。

让我们用Phoenix生成并提供给我们的其中之一的视图， `lib/hello_web/views/page_view.ex` 来进行一些实验。我们将向其添加一个 `message/0` 函数，如下所示。

```
defmodule HelloWeb.PageView do
  use HelloWeb, :view

  def message do
    "Hello from the view!"
  end
end
```

现在，让我们创建一个新模板，与之使用， `lib/hello_web/templates/page/test.html.eex` 。

```
This is the message: <%= message() %>
```

这与控制器中的任何动作都不对应，但是我们将在 `iex` 会话中进行练习。在项目的根目录，我们可以运行 `iex -S mix` ，然后明确地渲染我们的模板。

```
iex(1)> Phoenix.View.render(HelloWeb.PageView, "test.html", %{})
  {:safe, [["" | "This is the message: "] | "Hello from the view!"]}
```

如你所见，我们正在使用负责测试模板的单个视图，测试模板的名称以及一个空映射来表示我们可能想要传递的任何数据来调用 `render/3` 。返回值是一个元组，以原子 `:safe` 开头，后面是插值过的模板的结果io列表。这里的"Safe"表示Phoenix已转义了渲染模板的内容。Phoenix定义了它自己的 `Phoenix.HTML.Safe` 协议，包含对用原子，位串，列表，整数，浮点数和元组的实现，以在模板渲染为字符串时为我们处理这种转义。

如果我们将某些键值对分配给 `render/3` 的第三个参数会怎样？为了找出答案，我们只需要稍微更改一下模板即可。

```
I came from assigns: <%= @message %>
This is the message: <%= message() %>
```

请注意第一行中的 `@` 。现在，如果我们更改函数调用，则在重新编译 `PageView` 模块后会看到不同的渲染。

```
iex(2)> r HelloWeb.PageView
warning: redefining module HelloWeb.PageView (current version loaded from _build/dev/lib/hello/ebin/Elixir.HelloWeb.PageView.beam)
  lib/hello_web/views/page_view.ex:1

{:reloaded, HelloWeb.PageView, [HelloWeb.PageView]}

iex(3)> Phoenix.View.render(HelloWeb.PageView, "test.html", message: "Assigns has an @.")
{:safe,
 [[[[""  | "I came from assigns: "] | "Assigns has an @."] |
  "\nThis is the message: "] | "Hello from the view!"]}
```

让我们测试一下HTML转义，只是为了好玩。

```
iex(4)> Phoenix.View.render(HelloWeb.PageView, "test.html", message: "<script>badThings();</script>")
{:safe,
 [[[[""  | "I came from assigns: "] |
    "&lt;script&gt;badThings();&lt;/script&gt;"] |
   "\nThis is the message: "] | "Hello from the view!"]}
```

如果只需要渲染的字符串，而不需要整个元组，则可以使用 `render_to_iodata/3` 。

```
iex(5)> Phoenix.View.render_to_iodata(HelloWeb.PageView, "test.html", message: "Assigns has an @.")
[[[["" | "I came from assigns: "] | "Assigns has an @."] |
  "\nThis is the message: "] | "Hello from the view!"]
```

关于布局的杂谈（A Word About Layouts）

布局只是模板。就像其他模板一样，它们具有视图。在新生成的应用中，它为 `lib/hello_web/views/layout_view.ex` 。您可能想知道由渲染视图产生的字符串如何最终在布局内。这是一个很好的问题！如果我们看一下 `lib/hello_web/templates/layout/app.html.eex` ，大约在 `<body>` 中间的地方，我们将会看到。

```
<%= render(@view_module, @view_template, assigns) %>
```

这是从控制器中来的视图模块及其模板被渲染为字符串并放置在布局中的位置。

错误视图（The ErrorView）

Phoenix有一个叫做 `ErrorView` 的视图，在 `lib/hello_web/views/error_view.ex` 。它的目通常是从一个集中的位置处理两种最常见的错误—— `404 not found` 和 `500 internal error` 。让我们看看它是什么样子。

```
defmodule HelloWeb.ErrorView do
  use HelloWeb, :view

  def render("404.html", _assigns) do
    "Page not found"
  end

  def render("500.html", _assigns) do
    "Server internal error"
  end
end
```

在我们深入探讨之前，让我们看看 `404 not found` 在浏览器中呈现的消息是什么样的。在开发环

境中，Phoenix默认情况下会调试错误，从而为我们提供了非常有用的调试页面。但是，我们在这里要查看的是应用程序将在生产环境中使用的页面。为此，我们需要设置 `debug_errors: false为config/dev.exs` 。

```
use Mix.Config

config :hello, HelloWeb.Endpoint,
  http: [port: 4000],
  debug_errors: false,
  code_reloader: true,
  ...
```

修改配置文件后，我们需要重启服务，以使此更改生效。重启服务后，让我们访问http://localhost:4000/such/a/wrong/path来查看正在运行的本地应用程序，然后看看会得到什么。

好吧，那不是很令人兴奋。我们得到的裸字符串"Page not found"，显示时没有任何标记或样式。

让我们看看是否可以使用我们已经了解的关于视图的知识来使该页面更有趣。

第一个问题是，错误字符串从何而来？答案是 `ErrorView` 。

```
def render("404.html", _assigns) do
  "Page not found"
end
```

太好了，我们有一个 `render/2` 函数，参数是一个模板和一个 `assigns` 映射，我们将其忽略。这个 `render/2` 函数在哪里被调用？答案是 `Phoenix.Endpoint.RenderErrors` 模块中定义的 `render/5` 函数。该模块唯一的目的是捕获错误，并使用一个视图（在我们的示例中为 `HelloWeb.ErrorView` ）来渲染它们。现在我们了解了如何实现，让我们制作一个更好的错误页面。Phoenix为我们生成了一个 `ErrorView` ，但是它没有给我们 `lib/hello_web/templates/error` 目录。让我们创建一个。在我们的新目录中，让我们添加一个模板 `404.html.eex` 并为其添加一些标签——混合了我们的应用布局和一个带有我们给用户的信息的新div。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <meta name="author" content="">

    <title>Welcome to Phoenix!</title>
    <link rel="stylesheet" href="/css/app.css">
  </head>

  <body>
    <div class="container">
      <div class="header">
        <ul class="nav nav-pills pull-right">
          <li><a href="https://hexdocs.pm/phoenix/overview.html">Get Started</a></li>
        </ul>
        <span class="logo"></span>
      </div>

      <div class="phx-hero">
        <p>Sorry, the page you are looking for does not exist.</p>
      </div>

      <div class="footer">
        <p><a href="http://phoenixframework.org">phoenixframework.org</a></p>
      </div>

    </div> <!-- /container -->
    <script src="/js/app.js"></script>
  </body>
</html>
```

现在，我们可以使用上面在 `iex` 会话中进行渲染实验时看到的 `render/2` 功能。由于我们知道Phoenix会将 `404.html.eex` 模板作为 `render("404.html", assigns)` 函数子句进行预编译，因此我们可以从 `ErrorView` 中删除该子句。

```
- def render("404.html", _assigns) do
-   "Page not found"
```

```
- end
```

当我们回到http://localhost:4000/such/a/wrong/path时，我们应该看到一个更好的错误页面。值得注意的是，尽管我们希望错误页面具有网站其余部分的外观，但我们并未通过应用程序布局呈现 `404.html.eex` 模板。主要原因是，在全局处理错误时很容易碰到极端情况。如果我们希望最大程度地减少应用程序布局和 `404.html.eex` 模板之间的重复，则可以为页眉和页脚实现共享模板。请参阅 `Template Guide` 以获取更多信息。当然，我们也可以使用 `ErrorView` 中的 `def render("500.html", _assigns) do` 子句执行相同的步骤。我们还可以使用传递到 `ErrorView` 中任何 `render/2` 子句中的 `assigns` 映射，而不是丢弃它，以便在模板中显示更多信息。

渲染JSON（Rendering JSON）

视图的工作不仅是渲染HTML模板。视图是关于数据表示的。给定一大包数据，视图的目的是以有意义的方式呈现给定的某种格式，例如HTML，JSON，CSV或其他格式。如今，许多Web应用程序都将JSON返回到远程客户端，并且Phoenix视图非常适合JSON渲染。Phoenix使用 `Jason` 将Maps编码为JSON，因此我们在视图中所需要做的就是将要响应的数据格式化为Map，其余的将由Phoenix来完成。可以直接从控制器以JSON响应并跳过视图。但是，如果我们认为控制器具有接收请求和获取要发送回的数据的职责，则数据操作和格式化不属于这些职责。视图为我们提供了一个负责格式化和处理数据的模块。让我们看一下 `PageController` ，当我们用一些静态页面映射（例如JSON）而不是HTML响应时，会是什么样。

```
defmodule HelloWeb.PageController do
  use HelloWeb, :controller

  def show(conn, _params) do
    page = %{title: "foo"}

    render(conn, "show.json", page: page)
  end

  def index(conn, _params) do
    pages = [%{title: "foo"}, %{title: "bar"}]

    render(conn, "index.json", pages: pages)
  end
```

```
  end
```

在这里，我们有 `show/2` 和 `index/2` 动作返回静态页面数据。我们通过传递 `"show.json"` 而不是 `"show.html"` 来作为模板的名称。这样，我们就可以通过不同文件类型上的模式匹配来拥有负责呈现HTML和JSON的视图。

```
defmodule HelloWeb.PageView do
  use HelloWeb, :view

  def render("index.json", %{pages: pages}) do
    %{data: render_many(pages, HelloWeb.PageView, "page.json")}
  end

  def render("show.json", %{page: page}) do
    %{data: render_one(page, HelloWeb.PageView, "page.json")}
  end

  def render("page.json", %{page: page}) do
    %{title: page.title}
  end
end
```

在视图中，我们看到我们的 `render/2` 函数模式匹配 `"index.json"` ， `"show.json"` 和 `"page.json"` 。在我们控制器 `show/2` 函数中， `render(conn, "show.json", page: page)` 将根据视图的 `render/2` 函数中匹配名称和扩展名进行模式匹配。换句话说， `render(conn, "index.json", pages: pages)` 将调用 `render("index.json", %{pages: pages})` 。 `render_many/3` 函数获取我们要响应的数据（页面），一个View和一个字符串，以在View上定义的 `render/2` 函数上进行模式匹配。它会映射页面中的每个item，并将它们传递给View中与文件字符串相匹配的 `render/3` 函数。 `render_one/3` 使用相同的签名，最终使用 `render/2` 匹配的 `page.json` 来指定每个页面的外观。匹配了 `"index.json"` 的 `render/2` 函数如您所愿，将使用JSON进行响应：

```
{
  "data": [
    {
      "title": "foo"
    },
    {
```

```
    "title": "bar"
   },
 ]
}
```

`render/2` 匹配 `"show.json"` ：

```
{
 "data": {
  "title": "foo"
 }
}
```

以这种方式构建views很有用，因为它们可以被组合。想象一下这样一种情况，我们的 `Page` 与 `Author` 是 `has_many` 的关系，依照请求，我们可能希望使用 `page` 来返回 `author` 数据。我们可以使用新的 `render/2` 轻松完成此操作：

```
defmodule HelloWeb.PageView do
 use HelloWeb, :view
 alias HelloWeb.AuthorView

 def render("page_with_authors.json", %{page: page}) do
  %{title: page.title,
    authors: render_many(page.authors, AuthorView, "author.json")}
 end

 def render("page.json", %{page: page}) do
  %{title: page.title}
 end
end
```

分配中使用的名称由视图确定。例如 `PageView` 将会使用 `%{page: page}` ， `AuthorView` 将会使用 `%{author: author}` 。可以使用 `as` 选项覆盖。假设作者视图使用 `%{writer: writer}` 而不是 `%{author: author}` ：

```
def render("page_with_authors.json", %{page: page}) do
 %{title: page.title,
```

```
    authors: render_many(page.authors, AuthorView, "author.json", as: :writer)}
end
```