1. 启动和运行（Up and Running）

首个教程的目的是让Phoenix应用程序尽可能快地启动和运行。

在我们开始之前，请花一分钟阅读Installation Guide。通过预先安装任何必要的依赖项，我们才能顺利地启动和运行应用程序。

此时，我们应该已经安装了Elixir、Erlang、Hex和Phoenix archive。我们还应该安装PostgreSQL和node.js来构建一个默认的应用程序。

好，我们已经整装待发了！

我们可以从任何目录运行 `mix phx.new` 来启动我们的Phoenix应用程序。Phoenix将接受一个绝对路径或相对路径来作为我们新项目的目录。假设我们应用程序的名称是 `hello` ，让我们运行以下命令:

```
$ mix phx.new hello
```

> 在我们开始之前，关于webpack的说明：Phoenix将默认使用webpack进行资源管理。Webpack的依赖是通过node包管理器安装的，而不是mix。Phoenix将在 `mix phx.new` 任务的最后提示我们安装它们。如果在那时我们选择了不安装，并且后面也没通过npm install安装那些依赖项，我们的应用程序将在我们试图启动它时引发错误，资源可能无法正确加载。如果我们完全不想使用webpack，我们可以简单地传递 `--no-webpack` 给 `mix phx.new` 。
>
> 关于Ecto的说明：Ecto允许我们的Phoenix应用程序与数据存储进行通信，比如PostgreSQL、MySQL等。如果我们的应用程序不需要这个组件，我们可以通过传递 `--no-ecto` 标志给 `mix phx.new` 来跳过这个依赖项。此标志还可以与 `--no-webpack` 组合来创建一个骨架应用程序。
>
> 学习有关 `mix phx.new` 的更多内容，你可以阅读Mix Tasks指南。

```
mix phx.new hello
* creating hello/config/config.exs
* creating hello/config/dev.exs
```

```
* creating hello/config/prod.exs
...
* creating hello/assets/static/images/phoenix.png
* creating hello/assets/static/favicon.ico

Fetch and install dependencies? [Yn]
```

Phoenix生成应用程序所需的目录结构和所有文件。完成后，它会询问我们是否需要为我们安装依赖项。我们选择是。

```
Fetch and install dependencies? [Yn] Y
* running mix deps.get
* running mix deps.compile
* running cd assets && npm install && node node_modules/webpack/bin/webpack.js --mode development

We are almost there! The following steps are missing:

    $ cd hello

Then configure your database in config/dev.exs and run:

    $ mix ecto.create

Start your Phoenix app with:

    $ mix phx.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phx.server
```

一旦依赖项安装完毕，任务将提示我们切换到项目目录并启动应用程序。

Phoenix假设我们的PostgreSQL数据库将有一个postgres用户帐户，该帐户具有正确的权限和一个“postgres”的密码。如果情况不是这样，请参阅Mix Tasks指南，以学习更多关于 `mix ecto.create` 任务。

好吧，让我们试一试。首先，我们 `cd` 进入刚才创建的 `hello/` 目录中:

```
$ cd hello
```

现在我们将创建我们的数据库:

```
$ mix ecto.create
Compiling 13 files (.ex)
Generated hello app
The database for Hello.Repo has been created
```

> 注意：如果这是您第一次运行此命令，Phoenix可能还会要求安装Rebar。继续安装，因为Rebar用于构建Erlang包。

最后，我们将启动Phoenix服务器:

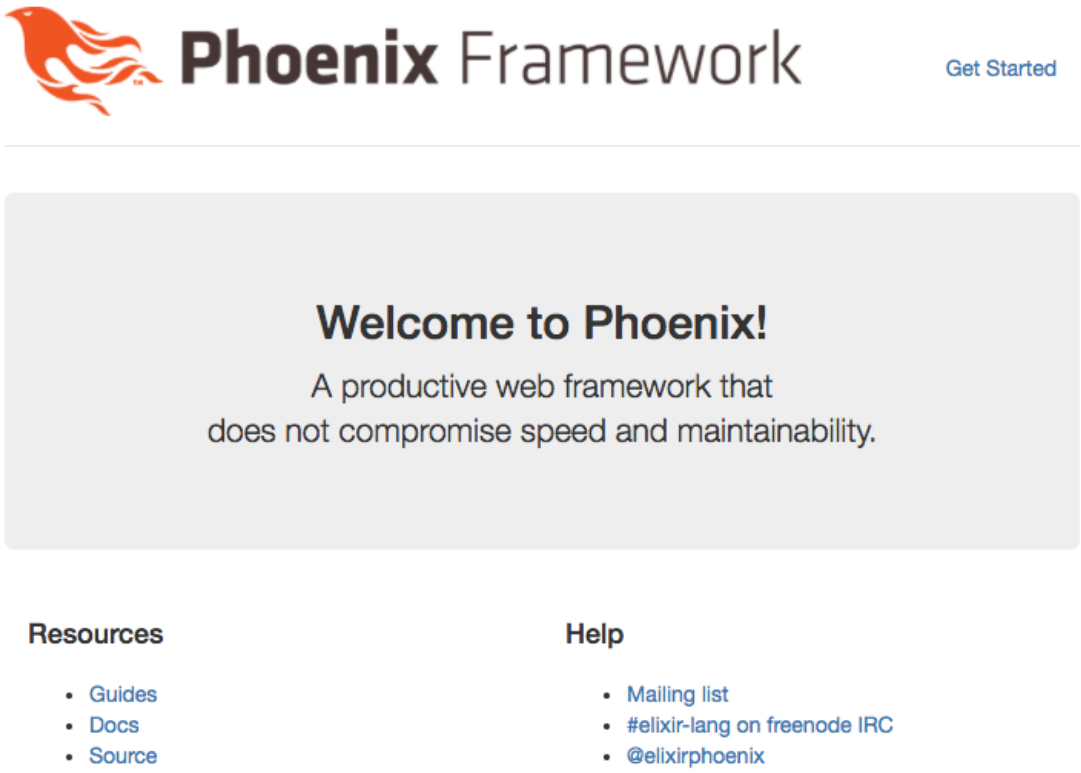
```
$ mix phx.server
[info] Running HelloWeb.Endpoint with cowboy 2.5.0 at http://localhost:4000

Webpack is watching the files…
...
```

如果我们在生成新应用程序时选择不让Phoenix安装我们的依赖项， `mix phx.new` 任务将提示我们当想要安装他们时采取的必要步骤。

```
Fetch and install dependencies? [Yn] n

We are almost there! The following steps are missing:

    $ cd hello
    $ mix deps.get
    $ cd assets && npm install && node node_modules/webpack/bin/webpack.js --mode development

Then configure your database in config/dev.exs and run:

    $ mix ecto.create

Start your Phoenix app with:
```

```
    $ mix phx.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phx.server
```

默认情况下，Phoenix接受4000端口的请求。如果我们将最喜欢的web浏览器指向http://localhost:4000，就会看到Phoenix Framework的欢迎页面。



如果你的屏幕看起来像上面的图片，恭喜你！你现在有了一个工作中的Phoenix应用程序。如果你看不到上面的页面，请尝试通过http://127.0.0.1:4000访问它，然后确保你的操作系统将“localhost”定义为“127.0.0.1”。

在本地，我们的应用程序在 `iex` 会话中运行。为了停止它，我们按下 `ctrl-c` 两次，就像我们通常停止 `iex` 一样。

下一步是稍微定制一下我们的应用程序，让我们了解一下Phoenix应用程序是如何组合在一起

的。

2. 添加页面

本指南的任务是为我们的Phoenix项目添加两个新页面。一个是纯静态的页面，另一个将取URL路径的一部分作为输入，并将其传递到模板以供显示。在此过程中，我们将熟悉Phoenix项目的基本组件：路由、控制器、视图和模板。

当Phoenix为我们生成一个新的应用程序时，它会构建一个像这样的顶级目录结构：

```
├── _build
├── assets
├── config
├── deps
├── lib
│   └── hello
│   └── hello_web
│   └── hello.ex
│   └── hello_web.ex
├── priv
├── test
```

本指南中的大部分工作都在 `lib/hello_web` 目录中，该目录保存应用程序中与web相关的部分。展开后如下：

```
├── channels
│   └── user_socket.ex
├── controllers
│   └── page_controller.ex
├── templates
│   ├── layout
│   │   └── app.html.eex
│   └── page
│       └── index.html.eex
└── views
    ├── error_helpers.ex
    ├── error_view.ex
    ├── layout_view.ex
    └── page_view.ex
├── endpoint.ex
```

```
├── gettext.ex
├── router.ex
```

当前在控制器、模板和视图目录中的所有文件都用于创建我们在上一篇指南中看到的“Welcome to Phoenix!”页面。稍后我们将看到如何重用其中一些代码。在开发环境中运行时，代码更改将在新的web请求时自动重新编译。

我们应用程序的所有静态资源，如js、css和图像文件都存在于 `assets` 文件夹中，这些资源由webpack或其他前端构建工具构建到 `priv/static` 中。我们暂时不会在这里做任何修改，但是将来参考的时候知道在哪里查找是很好的。

```
├── assets
│   ├── css
│   │   └── app.css
│   ├── js
│   │   └── app.js
│   └── static
│   └── node_modules
│   └── vendor
```

还有一些与web无关的文件我们应该知道。我们的应用程序文件(启动Elixir应用程序及其管理树)位于 `lib/hello/application.ex` 。在 `lib/hello/repo` 中也有Ecto Repo用于与数据库交互。你可以在Ecto的指南中了解更多。

```
lib
├── hello
|   ├── application.ex
|   └── repo.ex
├── hello_web
|   ├── channels
|   ├── controllers
|   ├── templates
|   ├── views
|   ├── endpoint.ex
|   ├── gettext.ex
|   └── router.ex
```

我们的 `lib/hello_web` 目录包含与web相关的文件——路由、控制器、模板、通道等。我们的Elixir应用程序其余部分位于 `lib/hello` 中，这里的代码结构与其他Elixir应用程序一样。

准备足够了，让我们开始第一个Phoenix新页面吧！

新的路由（A New Route）

路由是将唯一的HTTP verb/path对映射到处理它们的controller/action对。Phoenix在新应用程序中为我们生成一个路由文件 `lib/hello_web/router.ex` 。该文件是我们将在这个小节工作的地方。

我们上一节 "Welcome to Phoenix!"页面的路由是这样的：

```
get "/", PageController, :index
```

让我们来消化一下这个路由要告诉我们的东西。访问http://localhost:4000/向根路径发出一个HTTP GET请求。所有像这样的请求都由 `HelloWeb.PageController` 模块的 `index` 函数处理，该模块被定义在 `lib/hello_web/controllers/page_controller.ex` 文件中。

当我们将浏览器指向http://localhost:4000/hello时，我们将要构建的页面将简单地显示“Hello World, from Phoenix!”

创建该页面首先要做的是为它定义一个路由。在文本编辑器打开 `lib/hello_web/router.ex` 。它目前应该是这样的：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
```

```
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :index
  end

  # Other scopes may use custom stacks.
  # scope "/api", HelloWeb do
  #   pipe_through :api
  # end
end
```

现在，我们将忽略pipelines和 `scope` 的作用，只关注添加路由。（如果你感兴趣，我们将在路由指南中介绍这些主题）

让我们向router中添加一条新路由，它将 `/hello` 的 `GET` 请求映射到即将创建的 `HelloWeb.HelloController` 的 `index` 动作。

```
get "/hello", HelloController, :index
```

`router.ex` 文件中的 `scope "/"` 块现在应该是这样的：

```
scope "/", HelloWeb do
  pipe_through :browser

  get "/", PageController, :index
  get "/hello", HelloController, :index
end
```

新的控制器（A New Controller）

Controllers是Elixir的modules，action是其定义的Elixir functions。actions的目的是收集任何数据并执行渲染所需的任何任务。我们的路由表明我们需要一个带有 `index/2` 动作的 `HelloWeb.HelloController` 模块。

为此，我们创建一个新的 `lib/hello_web/controllers/hello_controller.ex` 文件，并使它看起来像下面这样：

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

  def index(conn, _params) do
    render(conn, "index.html")
  end
end
```

我们将在Controllers Guide保留 `use HelloWeb, :controller` 的讨论。现在，让我们专注于 `index/2` 动作。

所有控制器动作都带有两个参数。第一个是 `conn` ，该结构保存有关请求的大量数据。第二个是 `params` ，这是请求参数。在这里，我们没有使用 `params` ，而是通过添加 `_` 来避免编译器警告。

该动作的核心是 `render(conn, "index.html")` 。这告诉Phoenix去查找一个名为 `index.html.eex` 的模版并渲染它。Phoenix将在我们控制器目录后的那个目录中查找模板，也就是 `lib/hello_web/templates/hello` 。

> 注意：在这里也可以使用原子作为模板名称， `render(conn, :index)` ，但是将根据Accept headers来选择模板，例如 `”index.html”` 或 `"index.json"` 。

负责渲染的模块是视图，接下来我们将新建一个。

新的视图（A New View）

Phoenix视图有几个重要的工作。它们渲染模板。它们还充当来自控制器的原始数据的表示层，为在模板中使用做好准备。执行此转换的函数应该放在视图中。

例如，假设我们有一个数据结构，该数据结构用一个 `first_name` 字段和一个 `last_name` 字段表示用户，并且在模板中，我们要显示用户的全名。我们可以在模板中编写代码以将这些字段合并为一个全名，但是更好的方法是在视图中编写一个函数为我们完成，然后在模板中调用该函

数。这样做的结果是得到一个更干净，更清晰的模板。

为了给我们的 `HelloController` 渲染任何模板，我们需要一个 `HelloView` 。名称在这里很重要—视图和控制器名称的第一部分必须匹配。现在，让我们创建一个空的视图，并留下更详细的视图描述以供以后使用。创建 `lib/hello_web/views/hello_view.ex` 并使其如下所示：

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

新的模版（A New Template）

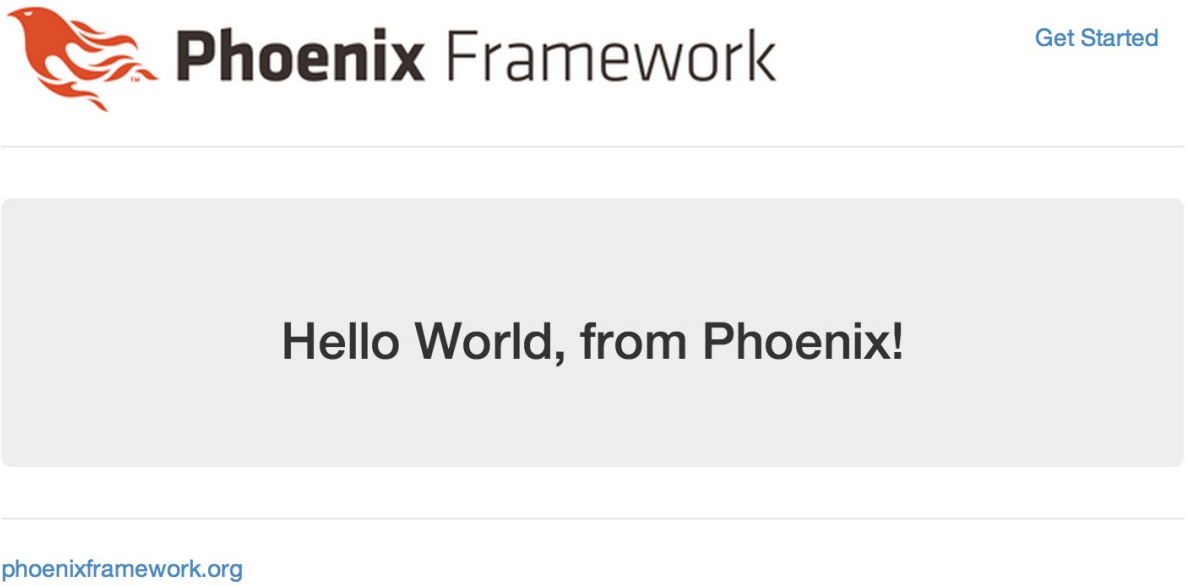
Phoenix模板就是可以把数据渲染其中的模版。Phoenix使用的标准模板引擎为 `EEx` （Embedded Elixir）。Phoenix增强了EEx，使其包括对值自动转义。这可以保护您免受跨站点脚本之类的安全漏洞的困扰，而无需您进行额外的工作。我们所有的模板文件都将具有 `.eex` 文件扩展名。

模板的作用域是视图，而视图的作用域是控制器。Phoenix创建了一个 `lib/hello_web/templates` 目录，我们可以在其中放置所有这些内容。最好为组织创建命名空间，因此对于我们的 `hello` 页面，这意味着我们需要在 `lib/hello_web/templates` 下创建 `hello` 目录，然后在里面创建 `index.html.eex` 文件。

现在开始吧。创建 `lib/hello_web/templates/hello/index.html.eex` 并使其如下所示：

```
<div class="phx-hero">
  <h2>Hello World, from Phoenix!</h2>
</div>
```

现在我们有了路由，控制器，视图和模板，我们应该能够将浏览器指向http://localhost:4000/hello并看到来自Phoenix的问候！（如果您在中途停止了服务器，则重新启动服务器的任务是 `mix phx.server` 。）



关于我们刚刚所做的事情，有几件有趣的事情要注意。进行这些更改后，我们无需停止并重新启动服务器。是的，Phoenix有代码热更新功能！同样，即使我们的 `index.html.eex` 文件仅包含一个div标签，但我们获得的页面都是完整的HTML文档。我们的index模板将被渲染到应用程序布局— `lib/hello_web/templates/layout/app.html.eex` 中。如果打开它，您将看到如下所示的一行：

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

这就是在HTML发送给浏览器之前将我们的模板渲染到布局中。

关于代码热更新的说明，某些编辑器的自动linters可能会阻止代码热更新工作。如果对您不起作用，请参阅这里的讨论。

另一个新页面（Another New Page）

让我们为应用程序增加一点复杂性。我们将添加一个新页面，该页面将识别一部分URL，将其标记为“messenger”，并将其通过控制器传递到模板中，以便我们的messenger可以打个招呼。

正如我们上次所做的那样，我们要做的第一件事就是创建一个新的路由。

新的路由（A New Route）

在本练习中，我们将重复使用刚创建的 `HelloController` 并添加一个新的 `show` 动作。我们将在最后一个路由下方添加一行，如下所示：

```
scope "/", HelloWeb do
  pipe_through :browser

  get "/", PageController, :index
  get "/hello", HelloController, :index
  get "/hello/:messenger", HelloController, :show
end
```

注意，我们将原子 `:messenger` 放置在路径中。Phoenix将采用URL中该位置出现的任何值，并将带有指向该值的键 `messenger` 以 `Map` 形式传递给控制器。

例如，如果将浏览器指向：http://localhost:4000/hello/Frank，则“:messenger”的值将为“Frank”。

新的动作（A New Action）

`HelloWeb.HelloController` `show` 动作将处理对我们新路由的请求。我们已经拥有控制器在 `lib/hello_web/controllers/hello_controller.ex` 中，因此我们所需要做的就是编辑该文件并向其中添加show动作。这次，我们需要在传递给动作的参数map中保留其中一项，以便我们可以将其（the messenger）传递给模板。为此，我们将以下show函数添加到控制器中：

```
def show(conn, %{"messenger" => messenger}) do
  render(conn, "show.html", messenger: messenger)
end
```

这里有几件事要注意。我们对传递给show函数的参数进行模式匹配，以便将 `messenger` 变量绑定到我们 `:messenger` 在URL中放置的值。例如，如果我们的URL是http://localhost:4000/hello/Frank，则messenger变量将绑定到 `Frank` 。

在该 `show` 动作的主体内，我们还将第三个参数传递给render函数，一个键/值对，其中 `:messenger` 是键， `messenger` 变量作为值传递。

> 注意：如果操作主体除了需要绑定的messenger变量之外，还需要访问绑定到params变量的完整参数映射，则可以这样定义 `show/2` ：

```
def show(conn, %{"messenger" => messenger} = params) do
  ...
end
```

最好记住，params映射的键将始终是字符串，并且等号不表示赋值，而是模式匹配断言。

新的模版（A New Template）

对于这个难题的最后一部分，我们需要一个新的模板。由于它是针对 `HelloController` 的 `show` 动作的，因此它将被放进 `lib/hello_web/templates/hello` 目录并被命名为 `show.html.eex` 。它看起来令人惊讶地类似于我们的 `index.html.eex` 模板，只是我们需要显示messenger的名称。

为此，我们将使用特殊的EEx标记执行Elixir表达式— `<%= %>` 。请注意，初始标签的等号如下： `<%=` 。这意味着将执行这些标签之间的所有Elixir代码，并且结果值将替换该标签。如果缺少等号，则仍将执行代码，但该值不会出现在页面上。

这是模板应该看起来的样子：

```
<div class="phx-hero">
  <h2>Hello World, from <%= @messenger %>!</h2>
</div>
```

我们的messenger显示为 `@messenger` 。在这种情况下，这不是模块属性。它是一种特殊的元编程语法，代表 `assign.messenger` 。这样的结果是在视觉上好得多，并且在模板中更容易使用。

大功告成。如果将浏览器指向此处：http:// localhost:4000/hello/Frank，您应该会看到一个如下所示的页面：

随便玩一下。无论您在 `/hello/` 后面放了什么它都会作为您的messenger出现在页面上。

3. 路由（Routing）

Routers是Phoenix应用程序的主要集线器。它们将HTTP请求匹配到控制器动作，连接实时通道处理程序，并定义了一系列管道转换以将中间件的作用域限定为路由集。

Phoenix生成的router文件 `lib/hello_web/router.ex` 看起来像这样：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :index
  end

  # Other scopes may use custom stacks.
  # scope "/api", HelloWeb do
  #   pipe_through :api
  # end
end
```

router和controller模块的名称都将以您为应用程序提供的名称开头，而不是 `HelloWeb` 。

该模块的第一行 `use HelloWeb, :router` 只是使Phoenix router函数在我们的特定router中可用。

Scopes在本指南中有自己的部分，因此我们不会花费时间在 `scope "/", HelloWeb do` 这

里。 `pipe_through :browser` 这行将在本指南的Pipeline部分中得到全面处理。现在，您只需要知道pipelines允许将一组中间件转换应用于不同的路由集。

但是，在scope块内，我们有第一个实际的路由：

```
get "/", PageController, :index
```

`get` 是Phoenix宏，它扩展为定义 `match/5` 函数的一个子句。它对应于HTTP动词GET。其他HTTP动词也存在类似的宏，包括
POST，PUT，PATCH，DELETE，OPTIONS，CONNECT，TRACE和HEAD。

这些宏的第一个参数是path。在这里，它是应用程序的根， `/` 。接下来的两个参数是我们要处理此请求的控制器和动作。这些宏还可以采用其他选项，我们将在本指南的其余部分中看到这些选项。

如果这是我们router模块中的唯一路由，则 `match/5` 函数的子句在宏扩展后如下所示：

```
def match(:get, "/", PageController, :index, [])
```

`match/5` 函数体建立连接并调用匹配的控制器动作。

当我们添加更多路由时，match函数的更多子句将添加到我们的router模块中。它们的行为将与Elixir中的任何其他多子句函数一样。它们将从顶部开始按顺序尝试，并且将执行匹配给定参数（动词和路径）的第一个子句。找到匹配项后，搜索将停止，并且不会尝试其他子句。

这意味着可以根据HTTP动词和路径创建一个永远不匹配的路由，而与控制器和动作无关。

如果确实创建了模凌两可的路由，router仍会编译，但会收到警告。让我们看看实际情况。

在该router `scope "/", HelloWeb do` 语句的底部定义此路由。

```
get "/", RootController, :index
```

然后在项目的根目录下运行 `mix compile` 。

检查路由（Examining Routes）

Phoenix提供了一个很好的工具来研究应用程序中的路由，即mix 任务 `phx.routes` 。

让我们看看它是如何工作的。前往新生成的Phoenix应用程序的根目录并运行 `mix phx.routes` 。（如果尚未这样做，则需要在运行 `routes` 任务前先运行 `mix do deps.get, compile` ）您应该看到类似以下内容，这些内容是根据我们目前拥有的唯一路由生成的：

```
$ mix phx.routes
page_path  GET  /  HelloWeb.PageController :index
```

输出的内容告诉我们，对于应用程序的根节点，任何HTTP GET请求将被 `HelloWeb.PageController` 的 `index` 动作处理。

`page_path` 是Phoenix称为路径助手的一个例子，我们将很快讨论它们。

资源（Resources）

除了HTTP动词 `get` ， `post` 和 `put` 之外，路由器还支持其他宏。其中最重要的是 `resources` ，它扩展为 `match/5` 函数的八个子句。

让我们在 `lib/hello_web/router.ex` 中添加一个像这样的资源：

```
scope "/", HelloWeb do
  pipe_through :browser

  get "/", PageController, :index
  resources "/users", UserController
end
```

出于这个目的，我们实际上没有 `HelloWeb.UserController` 并不重要。

然后前往项目的根目录并运行 `mix phx.routes`

您应该看到类似以下的内容：

```
user_path  GET     /users          HelloWeb.UserController :index
user_path  GET     /users/:id/edit  HelloWeb.UserController :edit
user_path  GET     /users/new       HelloWeb.UserController :new
user_path  GET     /users/:id       HelloWeb.UserController :show
user_path  POST    /users           HelloWeb.UserController :create
user_path  PATCH   /users/:id       HelloWeb.UserController :update
           PUT     /users/:id       HelloWeb.UserController :update
user_path  DELETE  /users/:id       HelloWeb.UserController :delete
```

当然，您的项目名称将替换 `HelloWeb` 。

这是HTTP动词，路径和控制器动作的标准矩阵。让我们以稍微不同的顺序分别来看它们。

- 对 `/users` 的GET请求将调用 `index` 动作来显示所有用户。
- 对 `/users/:id` 的GET请求将使用一个id调用 `show` 动作，以显示由该ID标识的单个用户。
- 对 `/users/new` 的GET请求将调用 `new` 动作以呈现用于创建新用户的表单。
- 对 `/users` 的POST请求将调用 `create` 动作以将新用户保存到数据存储中。
- 对 `/users/:id/edit` 的GET请求将使用一个id调用 `edit` 动作，以从数据存储中检索单个用户，并以表格形式显示信息以进行编辑。
- 对 `/users/:id` 的PATCH请求将使用一个id调用 `update` 动作，以将更新后的用户保存到数据存储中。
- 对 `/users/:id` 的PUT请求同样将使用一个id调用 `update` 动作，以将更新后的用户保存到数据存储中。
- 对 `/users/:id` 的DELETE请求使用一个id调用 `delete` 动作，以从数据存储中删除单个用户。

如果我们觉得不需要所有这些路由，可以使用 `:only` 和 `:except` 选项进行选择性的操作。

假设我们有一个只读的帖子资源。我们可以这样定义它：

```
resources "/posts", PostController, only: [:index, :show]
```

运行 `mix phx.routes` 显示我们只有定义了index和show动作的路由。

```
post_path  GET     /posts     HelloWeb.PostController :index
post_path  GET     /posts/:id  HelloWeb.PostController :show
```

同样，如果我们拥有一个评论资源，并且不想提供删除的路由，则可以定义这样的路由。

```
resources "/comments", CommentController, except: [:delete]
```

现在运行 `mix phx.routes` 显示我们有除了删除动作的DELETE请求外的所有路由。

```
comment_path  GET    /comments          HelloWeb.CommentController :index
comment_path  GET    /comments/:id/edit  HelloWeb.CommentController :edit
comment_path  GET    /comments/new       HelloWeb.CommentController :new
comment_path  GET    /comments/:id       HelloWeb.CommentController :show
comment_path  POST   /comments          HelloWeb.CommentController :create
comment_path  PATCH  /comments/:id       HelloWeb.CommentController :update
              PUT    /comments/:id       HelloWeb.CommentController :update
```

`Phoenix.Router.resources/4` 宏描述了定制资源路由的其他选项。

转发（Forward）

`Phoenix.Router.forward/4` 宏可以用于将以特定路径开始的所有请求发送到特定的plug。假设我们有一部分系统负责在后台运行jobs（甚至可以是一个单独的应用程序或库），它可以具有自己的web界面来检查jobs的状态。我们可以使用以下命令转发到该admin界面：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  ...

  scope "/", HelloWeb do
    ...
  end

  forward "/jobs", BackgroundJob.Plug
end
```

这意味着所有以 `/jobs` 开头的路由都将被发送到 `HelloWeb.BackgroundJob.Plug` 模块。

我们甚至可以在pipeline中使用 `forward/4` 宏。如果我们想确保用户通过身份验证并拥有管理员身份才能查看jobs页面，则可以在router中使用以下内容。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  ...

  scope "/" do
    pipe_through [:authenticate_user, :ensure_admin]
    forward "/jobs", BackgroundJob.Plug
  end
end
```

这意味着 `authenticate_user` 和 `ensure_admin` pipelines中的plug将在 `BackgroundJob.Plug` 允许它们发送适当的响应并调用 `halt()` 之前被调用。

传递给Plug `init/1` 回调的 `opts` 可以作为第三个参数传递。例如，也许后台job页面可让您设置要在页面上显示的应用程序名称。这可以通过下面来实现：

```
forward "/jobs", BackgroundJob.Plug, name: "Hello Phoenix"
```

可以传递第四个参数 `router_opts` 。这些选项在 `Phoenix.Router.scope/2` 文档中有概述。

尽管可以转发到任何模块plug，但不建议转发到另一个endpoint。这是因为您的应用程序定义的plug和转发的endpoint将被调用两次，这可能会导致错误。

编写一个实际的后台job worker不在本指南的讨论范围内。但是，为了方便起见并允许您测试上面的代码，以下是 `BackgroundJob.Plug` 的实现，您可以将其复制到应用程序中的 `lib/plugs/background_job_plug.ex` 里：

```
defmodule HelloWeb.BackgroundJob.Plug do
  def init(opts), do: opts
  def call(conn, opts) do
    conn
    |> Plug.Conn.assign(:name, Keyword.get(opts, :name, "Background Job"))
    |> HelloWeb.BackgroundJob.Router.call(opts)
```

```
  end
end

defmodule HelloWeb.BackgroundJob.Router do
  use Plug.Router

  plug :match
  plug :dispatch

  get "/", do: send_resp(conn, 200, "Welcome to #{conn.assigns.name}")
  get "/active", do: send_resp(conn, 200, "5 Active Jobs")
  get "/pending", do: send_resp(conn, 200, "3 Pending Jobs")
  match _, do: send_resp(conn, 404, "Not found")
end
```

路径助手（Path Helpers）

Path helpers是为单个应用程序动态定义于 `Router.Helpers` 模块的功能。对我们来说，它是 `HelloWeb.Router.Helpers` 。它们的名称是从路由定义所使用的控制器的名字中衍生出来的。我们的控制器是 `HelloWeb.PageController` ，而 `page_path` 是一个会返回应用程序根目录路径的函数。

百闻不如一见。在项目的根目录下运行 `iex -S mix` 。当我们在router helpers上调用 `page_path` 函数，以 `Endpoint` 或connection和action作为参数，它会返回路径。

```
iex> HelloWeb.Router.Helpers.page_path(HelloWeb.Endpoint, :index)
"/"
```

这很有用，因为我们可以在模板中使用 `page_path` 函数链接到应用程序的根目录。然后，我们可以在模板中使用该助手：

```
<a href="<%= Routes.page_path(@conn, :index) %>">To the Welcome Page!</a>
```

之所以可以使用 `Routes.page_path` 而不是全名 `HelloWeb.Router.Helpers.page_path` ，是因为在 `view/0` 中(`lib/hello_web.ex`) 默认定义 `Routes` 作为 `HelloWeb.Router.Helpers` 的别名，并通

过 `use HelloWeb, :view` 可以将其提供给我们的模板。当然，我们可以改用 `HelloWeb.Router.Helpers.page_path(@conn, :index)` ，但是为了简洁起见，约定使用别名版本（请注意，别名仅自动设置以用于视图，控制器和模板中，在这些之外您需要使用全名或在模块定义中自己指定： `alias HelloWeb.Router.Helpers, as: Routes` ）。请参阅View Guide以获取更多信息。

如果我们需要在router中改变路由的路径，这将会带来巨大的回报。由于path helper是基于路由动态构建的，所以在模板中任何对 `page_path` 的调用都能正常工作。

有关路径助手的更多信息（More on Path Helpers）

当我们为user resource运行 `phx.routes` 任务时，它列出了 `user_path` 作为每一行输出的path helper函数。这是每个动作的含义：

```
iex> alias HelloWeb.Router.Helpers, as: Routes
iex> alias HelloWeb.Endpoint
iex> Routes.user_path(Endpoint, :index)
"/users"

iex> Routes.user_path(Endpoint, :show, 17)
"/users/17"

iex> Routes.user_path(Endpoint, :new)
"/users/new"

iex> Routes.user_path(Endpoint, :create)
"/users"

iex> Routes.user_path(Endpoint, :edit, 37)
"/users/37/edit"

iex> Routes.user_path(Endpoint, :update, 37)
"/users/37"

iex> Routes.user_path(Endpoint, :delete, 17)
"/users/17"
```

如果path中有查询字符串呢？通过添加键值对作为第四个可选参数，path helpers将会以查询字符串返回这些键值对。

```
iex> Routes.user_path(Endpoint, :show, 17, admin: true, active: false)
"/users/17?admin=true&active=false"
```

如果我们需要一个完整的url而不是path呢？只需替换 `_path` 为 `_url` ：

```
iex(3)> Routes.user_url(Endpoint, :index)
"http://localhost:4000/users"
```

`_url` 函数将从每个环境设置的配置参数中获取构建完整URL所需的host，port，proxy port和SSL信息。我们将在它自己的指南中更详细地讨论配置。现在，您可以在自己的项目中查看 `config/dev.exs` 文件以查看这些值。

尽可能地传递 `conn` 来代替 `Endpoint` 。

嵌套资源（Nested Resources）

在Phoenix router中也可以嵌套resources。假设我们还有一个 `posts` 资源，它与 `users` 有着多对一的关系。也就是说，一个用户可以创建许多帖子，而单个帖子仅属于一个用户。我们可以在 `lib/hello_web/router.ex` 添加一个这样的嵌套路由来表示：

```
resources "/users", UserController do
  resources "/posts", PostController
end
```

当我们运行 `mix phx.routes` 时，除了上面看到的 `users` 路由外，我们还获得了以下一组路由：

```
...
user_post_path  GET     /users/:user_id/posts          HelloWeb.PostController :index
user_post_path  GET     /users/:user_id/posts/:id/edit HelloWeb.PostController :edit
user_post_path  GET     /users/:user_id/posts/new      HelloWeb.PostController :new
user_post_path  GET     /users/:user_id/posts/:id      HelloWeb.PostController :show
user_post_path  POST    /users/:user_id/posts          HelloWeb.PostController :create
user_post_path  PATCH   /users/:user_id/posts/:id      HelloWeb.PostController :update
                PUT     /users/:user_id/posts/:id      HelloWeb.PostController :update
user_post_path  DELETE  /users/:user_id/posts/:id      HelloWeb.PostController :delete
```

我们看到这些路由中的每一个都将posts限制到了一个用户ID。我们调用 `PostController` `index` 动作，但还会传递一个user_id。这意味着我们只显示某个用户的所有帖子。相同的限制应用于所有这些路由。

当我们为嵌套路由调用path helper函数时，我们需要按路由定义中的顺序传递这些ID。对于下面的 `show` 路由， `42` 是 `user_id` ， `17` 是 `post_id` 。在开始之前记得给我们的 `HelloWeb.Endpoint` 加上别名。

```
iex> alias HelloWeb.Endpoint
iex> HelloWeb.Router.Helpers.user_post_path(Endpoint, :show, 42, 17)
"/users/42/posts/17"
```

同样，如果我们在函数调用的末尾添加一个键/值对，则会将其添加到查询字符串中。

```
iex> HelloWeb.Router.Helpers.user_post_path(Endpoint, :index, 42, active: true)
"/users/42/posts?active=true"
```

如果我们像以前一样给 `Helpers` 模块加了别名（它仅自动为视图，模板和控制器设置别名，因为我们在 `iex` 里面，我们需要自己设置），我们可以改为：

```
iex> alias HelloWeb.Router.Helpers, as: Routes
iex> alias HelloWeb.Endpoint
iex> Routes.user_post_path(Endpoint, :index, 42, active: true)
"/users/42/posts?active=true"
```

作用域路由（Scoped Routes）

Scopes可以将路由分组到有着相同path前缀和一组作用域内的plug中间件里。我们可能想对admin功能，APIs，尤其是对版本化的APIs进行此操作。假设我们有一个用户在站点上生成reviews，并且这些reviews首先需要得到管理员的批准。这些资源的语义完全不同，它们可能不能共享同一个控制器。Scopes使我们能够分隔这些路由。

面向用户的reviews path看起来像是标准资源。

```
/reviews
/reviews/1234
/reviews/1234/edit
...
```

管理员的review path可以以 `/admin` 开头。

```
/admin/reviews
/admin/reviews/1234
/admin/reviews/1234/edit
...
```

我们使用作用域路由来完成此操作，该路由将path选项设置为 `/admin` 这样。现在，我们不要将此scope嵌套在任何其他scopes内（例如，在新应用程序中为我们提供的 `scope "/", HelloWeb do` ）。

```
scope "/admin" do
  pipe_through :browser

  resources "/reviews", HelloWeb.Admin.ReviewController
end
```

还要注意，当前定义此scope的方式，我们需要完全修饰控制器名称为 `HelloWeb.Admin.ReviewController` 。我们将在稍后修复这个问题。

再次运行 `mix phx.routes` ，除了前面的一组路由外，我们还得到以下内容：

```
...
review_path  GET     /admin/reviews          HelloWeb.Admin.ReviewController :index
review_path  GET     /admin/reviews/:id/edit HelloWeb.Admin.ReviewController :edit
review_path  GET     /admin/reviews/new      HelloWeb.Admin.ReviewController :new
review_path  GET     /admin/reviews/:id      HelloWeb.Admin.ReviewController :show
review_path  POST    /admin/reviews          HelloWeb.Admin.ReviewController :create
review_path  PATCH   /admin/reviews/:id      HelloWeb.Admin.ReviewController :update
             PUT     /admin/reviews/:id      HelloWeb.Admin.ReviewController :update
review_path  DELETE  /admin/reviews/:id      HelloWeb.Admin.ReviewController :delete
```

看起来不错，但是这里有问题。记住，我们既要用户面对的reviews路由，也要管理员的 `/admin/reviews` 。如果我们现在将用户面对的reviews包括在router中，如下所示：

```
scope "/", HelloWeb do
  pipe_through :browser
  ...
  resources "/reviews", ReviewController
  ...
end

scope "/admin" do
  resources "/reviews", HelloWeb.Admin.ReviewController
end
```

然后运行 `mix phx.routes` ，我们得到以下输出：

```
...
review_path  GET     /reviews               HelloWeb.ReviewController :index
review_path  GET     /reviews/:id/edit      HelloWeb.ReviewController :edit
review_path  GET     /reviews/new           HelloWeb.ReviewController :new
review_path  GET     /reviews/:id           HelloWeb.ReviewController :show
review_path  POST    /reviews               HelloWeb.ReviewController :create
review_path  PATCH   /reviews/:id           HelloWeb.ReviewController :update
             PUT     /reviews/:id           HelloWeb.ReviewController :update
review_path  DELETE  /reviews/:id           HelloWeb.ReviewController :delete
...
review_path  GET     /admin/reviews         HelloWeb.Admin.ReviewController :index
review_path  GET     /admin/reviews/:id/edit  HelloWeb.Admin.ReviewController :edit
review_path  GET     /admin/reviews/new     HelloWeb.Admin.ReviewController :new
review_path  GET     /admin/reviews/:id     HelloWeb.Admin.ReviewController :show
review_path  POST    /admin/reviews         HelloWeb.Admin.ReviewController :create
review_path  PATCH   /admin/reviews/:id     HelloWeb.Admin.ReviewController :update
             PUT     /admin/reviews/:id     HelloWeb.Admin.ReviewController :update
review_path  DELETE  /admin/reviews/:id     HelloWeb.Admin.ReviewController :delete
```

除了每行开头的 `review_path` path helper，我们实际获得的所有路由都看起来正确。我们为用户面对的review路由和管理员的使用了同一个helper，这是不对的。我们可以通过添加一个 `as: :admin` 选项到admin scope来解决这个问题。

```
scope "/", HelloWeb do
  pipe_through :browser
  ...
  resources "/reviews", ReviewController
  ...
end

scope "/admin", as: :admin do
  resources "/reviews", HelloWeb.Admin.ReviewController
end
```

`mix phx.routes` 现在向我们展示了我们所需要的结果。

```
...
      review_path  GET     /reviews                 HelloWeb.ReviewController :index
      review_path  GET     /reviews/:id/edit        HelloWeb.ReviewController :edit
      review_path  GET     /reviews/new             HelloWeb.ReviewController :new
      review_path  GET     /reviews/:id             HelloWeb.ReviewController :show
      review_path  POST    /reviews                 HelloWeb.ReviewController :create
      review_path  PATCH   /reviews/:id             HelloWeb.ReviewController :update
                   PUT     /reviews/:id             HelloWeb.ReviewController :update
      review_path  DELETE  /reviews/:id             HelloWeb.ReviewController :delete
...
admin_review_path  GET     /admin/reviews           HelloWeb.Admin.ReviewController :index
admin_review_path  GET     /admin/reviews/:id/edit  HelloWeb.Admin.ReviewController :edit
admin_review_path  GET     /admin/reviews/new       HelloWeb.Admin.ReviewController :new
admin_review_path  GET     /admin/reviews/:id       HelloWeb.Admin.ReviewController :show
admin_review_path  POST    /admin/reviews           HelloWeb.Admin.ReviewController :create
admin_review_path  PATCH   /admin/reviews/:id       HelloWeb.Admin.ReviewController :update
                   PUT     /admin/reviews/:id       HelloWeb.Admin.ReviewController :update
admin_review_path  DELETE  /admin/reviews/:id       HelloWeb.Admin.ReviewController :delete
```

path helper现在也会像我们预期的那样运作。运行 `iex -S mix` 并尝试一下。

```
iex(1)> HelloWeb.Router.Helpers.review_path(HelloWeb.Endpoint, :index)
"/reviews"

iex(2)> HelloWeb.Router.Helpers.admin_review_path(HelloWeb.Endpoint, :show, 1234)
"/admin/reviews/1234"
```

如果我们有一些资源都需要管理员处理呢？我们可以将它们全部放在同一个scope内，如下所示：

```
scope "/admin", as: :admin do
  pipe_through :browser

  resources "/images",  HelloWeb.Admin.ImageController
  resources "/reviews", HelloWeb.Admin.ReviewController
  resources "/users",   HelloWeb.Admin.UserController
end
```

这里就是 `mix phx.routes` 将会告诉我们的：

```
...
 admin_image_path  GET     /admin/images          HelloWeb.Admin.ImageController :index
 admin_image_path  GET     /admin/images/:id/edit HelloWeb.Admin.ImageController :edit
 admin_image_path  GET     /admin/images/new      HelloWeb.Admin.ImageController :new
 admin_image_path  GET     /admin/images/:id      HelloWeb.Admin.ImageController :show
 admin_image_path  POST    /admin/images          HelloWeb.Admin.ImageController :create
 admin_image_path  PATCH   /admin/images/:id      HelloWeb.Admin.ImageController :update
                   PUT     /admin/images/:id      HelloWeb.Admin.ImageController :update
 admin_image_path  DELETE  /admin/images/:id      HelloWeb.Admin.ImageController :delete
admin_review_path  GET     /admin/reviews         HelloWeb.Admin.ReviewController :index
admin_review_path  GET     /admin/reviews/:id/edit HelloWeb.Admin.ReviewController :edit
admin_review_path  GET     /admin/reviews/new     HelloWeb.Admin.ReviewController :new
admin_review_path  GET     /admin/reviews/:id     HelloWeb.Admin.ReviewController :show
admin_review_path  POST    /admin/reviews         HelloWeb.Admin.ReviewController :create
admin_review_path  PATCH   /admin/reviews/:id     HelloWeb.Admin.ReviewController :update
                   PUT     /admin/reviews/:id     HelloWeb.Admin.ReviewController :update
admin_review_path  DELETE  /admin/reviews/:id     HelloWeb.Admin.ReviewController :delete
 admin_user_path  GET     /admin/users           HelloWeb.Admin.UserController :index
 admin_user_path  GET     /admin/users/:id/edit  HelloWeb.Admin.UserController :edit
 admin_user_path  GET     /admin/users/new       HelloWeb.Admin.UserController :new
 admin_user_path  GET     /admin/users/:id       HelloWeb.Admin.UserController :show
 admin_user_path  POST    /admin/users           HelloWeb.Admin.UserController :create
 admin_user_path  PATCH   /admin/users/:id       HelloWeb.Admin.UserController :update
                  PUT     /admin/users/:id       HelloWeb.Admin.UserController :update
 admin_user_path  DELETE  /admin/users/:id       HelloWeb.Admin.UserController :delete
```

这很好，正是我们想要的，但是我们可以做得更好。请注意，每个资源的控制器前都要加

上 `HelloWeb.Admin` ，这很乏味且容易出错。假设每个控制器的名称都以 `HelloWeb.Admin` 开头，那么我们可以添加一个 `HelloWeb.Admin` 选项到我们的scope声明中，就在scope path后面。这样我们所有的路由都将具有正确完整的控制器名。

```
scope "/admin", HelloWeb.Admin, as: :admin do
  pipe_through :browser

  resources "/images",  ImageController
  resources "/reviews", ReviewController
  resources "/users",   UserController
end
```

现在再次运行 `mix phx.routes` ，您会看到与上面分别修饰每个控制器名称时相同的结果。

这不仅适用于嵌套路由，我们甚至可以将应用程序的所有路由嵌套在一个作用域中，该作用域的别名就是我们Phoenix应用的名字，这样可以消除控制器名称中重复的应用名。

Phoenix已经在为新应用程序生成的router中为我们做到了（请参阅本节的开头）。注意此处 `HelloWeb` 在 `scope` 声明中的使用：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  scope "/", HelloWeb do
    pipe_through :browser

    get "/images", ImageController, :index
    resources "/reviews", ReviewController
    resources "/users",   UserController
  end
end
```

再次运行 `mix phx.routes` ，结果表明我们所有的控制器现在都具有正确的，完全修饰的名称。

```
image_path  GET     /images           HelloWeb.ImageController :index
review_path  GET     /reviews          HelloWeb.ReviewController :index
review_path  GET     /reviews/:id/edit  HelloWeb.ReviewController :edit
review_path  GET     /reviews/new      HelloWeb.ReviewController :new
```

```
review_path  GET     /reviews/:id       HelloWeb.ReviewController :show
review_path  POST    /reviews           HelloWeb.ReviewController :create
review_path  PATCH   /reviews/:id       HelloWeb.ReviewController :update
             PUT     /reviews/:id       HelloWeb.ReviewController :update
review_path  DELETE  /reviews/:id       HelloWeb.ReviewController :delete
 user_path  GET     /users             HelloWeb.UserController :index
 user_path  GET     /users/:id/edit    HelloWeb.UserController :edit
 user_path  GET     /users/new         HelloWeb.UserController :new
 user_path  GET     /users/:id         HelloWeb.UserController :show
 user_path  POST    /users             HelloWeb.UserController :create
 user_path  PATCH   /users/:id         HelloWeb.UserController :update
            PUT     /users/:id         HelloWeb.UserController :update
 user_path  DELETE  /users/:id         HelloWeb.UserController :delete
```

尽管从技术上讲，作用域也可以嵌套（就像资源一样），但通常不建议使用嵌套作用域，因为它有时会使我们的代码混乱不清。话虽如此，假设我们有一个版本化的API，包含为images，reviews和用户定义的resources。从技术上讲，我们可以为版本化的API这样设置路由：

```
scope "/api", HelloWeb.Api, as: :api do
  pipe_through :api

  scope "/v1", V1, as: :v1 do
    resources "/images",  ImageController
    resources "/reviews", ReviewController
    resources "/users",   UserController
  end
end
```

`mix phx.routes` 表面我们得到了我们想要的路由。

```
 api_v1_image_path  GET     /api/v1/images           HelloWeb.Api.V1.ImageController :index
 api_v1_image_path  GET     /api/v1/images/:id/edit  HelloWeb.Api.V1.ImageController :edit
 api_v1_image_path  GET     /api/v1/images/new       HelloWeb.Api.V1.ImageController :new
 api_v1_image_path  GET     /api/v1/images/:id       HelloWeb.Api.V1.ImageController :show
 api_v1_image_path  POST    /api/v1/images           HelloWeb.Api.V1.ImageController :create
 api_v1_image_path  PATCH   /api/v1/images/:id       HelloWeb.Api.V1.ImageController :update
                    PUT     /api/v1/images/:id       HelloWeb.Api.V1.ImageController :update
 api_v1_image_path  DELETE  /api/v1/images/:id       HelloWeb.Api.V1.ImageController :delete
```

```
api_v1_review_path  GET     /api/v1/reviews          HelloWeb.Api.V1.ReviewController :index
api_v1_review_path  GET     /api/v1/reviews/:id/edit HelloWeb.Api.V1.ReviewController :edit
api_v1_review_path  GET     /api/v1/reviews/new      HelloWeb.Api.V1.ReviewController :new
api_v1_review_path  GET     /api/v1/reviews/:id      HelloWeb.Api.V1.ReviewController :show
api_v1_review_path  POST    /api/v1/reviews          HelloWeb.Api.V1.ReviewController :create
api_v1_review_path  PATCH   /api/v1/reviews/:id      HelloWeb.Api.V1.ReviewController :update
                    PUT     /api/v1/reviews/:id      HelloWeb.Api.V1.ReviewController :update
api_v1_review_path  DELETE  /api/v1/reviews/:id      HelloWeb.Api.V1.ReviewController :delete
 api_v1_user_path  GET     /api/v1/users            HelloWeb.Api.V1.UserController :index
 api_v1_user_path  GET     /api/v1/users/:id/edit   HelloWeb.Api.V1.UserController :edit
 api_v1_user_path  GET     /api/v1/users/new        HelloWeb.Api.V1.UserController :new
 api_v1_user_path  GET     /api/v1/users/:id        HelloWeb.Api.V1.UserController :show
 api_v1_user_path  POST    /api/v1/users            HelloWeb.Api.V1.UserController :create
 api_v1_user_path  PATCH   /api/v1/users/:id        HelloWeb.Api.V1.UserController :update
                   PUT     /api/v1/users/:id        HelloWeb.Api.V1.UserController :update
 api_v1_user_path  DELETE  /api/v1/users/:id        HelloWeb.Api.V1.UserController :delete
```

有趣的是，我们可以让多个scopes具有相同的path，只要我们注意不要重复路由。如果我们重复了路由，将会得到类似的警告。

```
warning: this clause cannot match because a previous clause at line 16 always matches
```

这个路由为同一个path定义了两个scopes，依旧完全正常。

```
defmodule HelloWeb.Router do
  use Phoenix.Router
  ...
  scope "/", HelloWeb do
    pipe_through :browser

    resources "/users", UserController
  end

  scope "/", AnotherAppWeb do
    pipe_through :browser

    resources "/posts", PostController
  end
  ...
end
```

当我们运行 `mix phx.routes` 时，我们看到以下输出：

```
user_path  GET     /users          HelloWeb.UserController :index
user_path  GET     /users/:id/edit  HelloWeb.UserController :edit
user_path  GET     /users/new       HelloWeb.UserController :new
user_path  GET     /users/:id       HelloWeb.UserController :show
user_path  POST    /users           HelloWeb.UserController :create
user_path  PATCH   /users/:id       HelloWeb.UserController :update
           PUT     /users/:id       HelloWeb.UserController :update
user_path  DELETE  /users/:id       HelloWeb.UserController :delete
post_path  GET     /posts           AnotherAppWeb.PostController :index
post_path  GET     /posts/:id/edit  AnotherAppWeb.PostController :edit
post_path  GET     /posts/new       AnotherAppWeb.PostController :new
post_path  GET     /posts/:id       AnotherAppWeb.PostController :show
post_path  POST    /posts           AnotherAppWeb.PostController :create
post_path  PATCH   /posts/:id       AnotherAppWeb.PostController :update
           PUT     /posts/:id       AnotherAppWeb.PostController :update
post_path  DELETE  /posts/:id       AnotherAppWeb.PostController :delete
```

管道（Pipelines）

距离在router中看到的第一行— `pipe_through :browser` 已经过了很久，我们还没谈论到。现在让我们处理它。

还记得Overview Guide中，我们曾把plug描述为按预先确定的顺序堆叠并执行吗？就像管道一样。我们将进一步研究这些plug堆栈是如何在router中工作的。

Pipelines就是plug按特定顺序堆叠在一起，然后起个名字。它们允许我们自定义与请求处理相关的行为和转换。Phoenix为我们提供了一些用于常见任务的默认pipelines。反过来，我们可以定制它们，并创建新的pipeline来满足我们的需求。

新生成的Phoenix应用程序定义了两个pipeline，叫做 `:browser` 和 `:api` 。我们将会介绍这些内容，但是首先我们需要讨论一下Endpoint plug中的plug堆栈。

端点Plugs（The Endpoint Plugs）

Endpoints组织了所有请求都要用到的plugs，并在将它们与底层的 `:browser` 、 `:api` 和自定义管道分配到router之前应用它们。默认的Endpoint plug做了非常多的工作。下面按顺序介绍它们。

- Plug.Static—服务静态资源。由于这个plug位于logger之前，所以静态资源服务是没有登记的。
- Phoenix.CodeReloader—为web目录中的所有入口开启了代码重载。它直接在Phoenix应用程序中配置。
- Plug.RequestId—为每个请求生成唯一的请求ID。
- Plug.Logger—记录传入的请求。
- Plug.Parsers—在已知解析器可用时解析请求的正文。默认情况下，解析器解析urlencoded，multipart和json（包括 `jason` ）。当不能解析请求内容类型时，请求主体将保持不变。
- Plug.MethodOverride—使用有效 `_method` 参数将POST请求的方法转换为PUT，PATCH或DELETE。
- Plug.Head—将HEAD请求转换为GET请求并剥离response body。
- Plug.Session—设置会话管理。请注意， `fetch_session/2` 在使用session之前，仍必须显式被调用，因为这个plug仅设置如何获取会话。
- Plug.Router-将router放进请求周期。

`:browser` 和 `:api` 管道

Phoenix定义了另外两个默认的管道， `:browser` 和 `:api` 。假定我们已经在一个封闭的scope中使用它们调用了 `pipe_through/1` ，则router会在匹配到一个路由之后调用它们。

顾名思义， `:browser` 管道是为准备向浏览器渲染请求的路由提供的。 `:api` 管道是为准备给API产生数据的路由提供的。

`:browser` 管道有五个plug： `plug :accepts, ["html"]` 定义了请求的格式或将要接受的格式， `:fetch_session` 会自然地获取session数据，并让其在连接中可用， `:fetch_flash` 会检索任何设置好的flash信息，以及 `:protect_from_forgery` 与 `:put_secure_browser_headers` ，可保护表单提交免受跨站点伪造。

当前， `:api` 管道仅定义了 `plug :accepts, ["json"]` 。

router在scope内定义的路由上调用pipeline。如果scope未定义，router将会在其所有路由上调用pipeline。尽管不鼓励嵌套scope（请参见上文），但是如果我们在一个嵌套scope内调用 `pipe_through` ，router将从父作用域调用所有的 `pipe_through` ，然后是嵌套作用域。

多说无益，直接上代码。

下面是新生成的Phoenix应用程序的另一个router。这次取消了api scope的注释，并添加了一条路由。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :index
  end

  # Other scopes may use custom stacks.
  scope "/api", HelloWeb do
```

```
    pipe_through :api

    resources "/reviews", ReviewController
  end
end
```

当服务器接受一个请求时，该请求总是会先经过Endpoint上的中的plugs，然后它会试图匹配path和HTTP verb。

假设该请求匹配到了我们的第一个路由：对 `/` 的GET。router会先将请求送至 `:browser` 管道—它会获取session数据，获取flash并执行防伪造保护。然后再将请求分派给 `PageController` `index` 动作。

相反，如果请求匹配到了 `resources/2` 宏定义的任何路由，router会将其送至 `:api` 管道—当前它什么都不会做—在将请求分派到 `HelloWeb.ReviewController` 中正确的动作之前。

如果我们知道我们的应用仅为浏览器渲染视图，则可以通过删除 `api` 以及scopes来简化router。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipe_through :browser

  get "/", HelloWeb.PageController, :index

  resources "/reviews", HelloWeb.ReviewController
end
```

删除了所有scope意味着router强制对其所有路由调用 `:browser` 管道。

让我们进一步思考一下。如果我们需要同时通过 `:browser` 和一个或多个自定义管道输送请求，该怎么办？我们简单地通过 `pipe_through` 管道列表，Phoenix将按顺序调用它们。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end
  ...

  scope "/reviews" do
    pipe_through [:browser, :review_checks, :other_great_stuff]

    resources "/", HelloWeb.ReviewController
  end
end
```

这是另一个示例，其中两个scopes有着不同的管道：

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end
  ...

  scope "/", HelloWeb do
    pipe_through :browser

    resources "/posts", PostController
```

```
  end

  scope "/reviews", HelloWeb do
    pipe_through [:browser, :review_checks]

    resources "/", ReviewController
  end
end
```

通常，管道的scoping规则和你预期一样。在此示例中，所有路由都将会经过 `:browser` 管道。但是，只有 `reviews` 资源路由会经过 `:review_checks` 管道。由于我们是以 `pipe_through [:browser, :review_checks]` 来声明管道的，因此Phoenix会按顺序调用列表中的每个管道。

创建新管道

Phoenix允许我们在router中的任何位置创建自己的自定义管道。为此，我们使用以下参数调用 `pipeline/2` 宏：一个代表管道名的原子，一个包含所有所需plug的块。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :review_checks do
    plug :ensure_authenticated_user
    plug :ensure_user_owns_review
  end

  scope "/reviews", HelloWeb do
    pipe_through :review_checks

    resources "/", ReviewController
```

```
  end
end
```

频道路由（Channel Routes）

频道是Phoenix框架中非常令人兴奋的实时组件。频道会处理给定主题的socket广播的传入和传出消息。因此，频道路由需要按socket和主题匹配请求，以便分配到正确的频道。（关于频道及其行为的详细说明，请参阅Channel Guide）

我们在 `lib/hello_web/endpoint.ex` 中的endpoint挂载了socket handlers。Socket handlers会处理认证回调和频道路由。

```
defmodule HelloWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :hello

  socket "/socket", HelloWeb.UserSocket,
    websocket: true,
    longpoll: false
  ...
end
```

默认情况下，在endpoint中调用 `Phoenix.Endpoint.socket/3` 时，它同时支持websocket和longpoll。这里我们指定可以通过WebSocket连接建立传入的socket连接。

接下来，我们需要打开 `lib/hello_web/channels/user_socket.ex` 文件并使用 `channel/3` 宏来定义我们的通道路由。路由会将一个主题模式与一个通道匹配以处理事件。如果我们有一个名为 `RoomChannel` 的通道模块和一个名为 `"rooms:*"` 的主题，那么执行此操作的代码可以这样写。

```
defmodule HelloWeb.UserSocket do
  use Phoenix.Socket

  channel "rooms:*", HelloWeb.RoomChannel
  ...
end
```

主题只是字符串标识符。我们在这里使用的形式是一种约定，它使我们可以在同一字符串中定义主题和子主题—`“topic：subtopic”`。`*`是一个通配符，它允许我们匹配任何子话题，所以`"rooms:lobby"`和`"rooms:kitchen"`会都匹配该路由。

每个socket可以为多个频道处理请求。

```
channel "rooms:*", HelloWeb.RoomChannel
channel "foods:*", HelloWeb.FoodChannel
```

我们可以把多个socket handlers挂载到endpoint中：

```
socket "/socket", HelloWeb.UserSocket
socket "/admin-socket", HelloWeb.AdminSocket
```

总结

路由是一个很大的话题，在这里我们花了许多笔墨。本指南的重点是：

- 以HTTP verb名为开头的路由会展开成match函数的一个子句。
- 以“resources”为开头的路由会展开为match函数的8个子句。
- Resources可以通过使用`only:`或`except:`选项来限制match函数子句的数量。
- 任何路由都可以嵌套。
- 任何路由都可以被scoped到一个给定的path。
- 在scope中使用`as:`选项可以减少重复。
- 对scoped路由使用helper选项可消除无法获取的paths。

4. Plug

Plug位于Phoenix HTTP层的核心，Phoenix将Plug放在重要的位置。我们在connection生命周期的每一步都与plugs进行交互，Phoenix的核心组件（如Endpoints，Routers和Controllers）本质上都是Plugs。让我们来看看是什么让Plug如此特别。

Plug是web应用中可组合模块的规范。它也是不同web服务的connection适配器的抽象层。Plug的基本思想就是统一我们操作的“connection”概念。这与其他HTTP中间件层不同，比如Rack，请求和响应在中间件堆栈中是分开的。

Plug规范可以简单地分为*function plugs*和*module plugs*。

Function Plugs

为了充当一个plug，一个函数只需要接受一个connection结构（ `%Plug.Conn{}` ）和选项。它的返回值也应该是connection结构。任何符合这些标准的函数都是plug。请看例子。

```
def put_headers(conn, key_values) do
  Enum.reduce key_values, conn, fn {k, v}, conn ->
    Plug.Conn.put_resp_header(conn, to_string(k), v)
  end
end
```

很简单，对吧？

这就是我们如何使用它们在Phoenix的connection上来构成一系列的转换：

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

  plug :put_headers, %{content_encoding: "gzip", cache_control: "max-age=3600"}
  plug :put_layout, "bare.html"

  ...
end
```

通过遵守plug协议， `put_headers/2` ， `put_layout/2` ，还有 `action/2` 把一个应用程序请求转换成一系列明确的变换。它不止于此。为了真正了解Plug的设计效果如何，让我们想象一个场景，我们需要检查一系列条件，然后在条件失败时重定向或停止。没有plug，我们可能需要这样：

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

  def show(conn, params) do
    case authenticate(conn) do
      {:ok, user} ->
        case find_message(params["id"]) do
          nil ->
            conn |> put_flash(:info, "That message wasn't found") |> redirect(to: "/")
          message ->
            case authorize_message(conn, params["id"]) do
              :ok ->
                render(conn, :show, page: find_message(params["id"]))
              :error ->
                conn |> put_flash(:info, "You can't access that page") |> redirect(to: "/")
            end
        end
      :error ->
        conn |> put_flash(:info, "You must be logged in") |> redirect(to: "/")
    end
  end
end
```

注意到了仅几步身份验证和授权就需要这么复杂的嵌套和重复吗？让我们通过几个plug来改进它。

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

  plug :authenticate
  plug :fetch_message
  plug :authorize_message

  def show(conn, params) do
    render(conn, :show, page: find_message(params["id"]))
```

```
  end

  defp authenticate(conn, _) do
    case Authenticator.find_user(conn) do
      {:ok, user} ->
        assign(conn, :user, user)
      :error ->
        conn |> put_flash(:info, "You must be logged in") |> redirect(to: "/") |> halt()
    end
  end

  defp fetch_message(conn, _) do
    case find_message(conn.params["id"]) do
      nil ->
        conn |> put_flash(:info, "That message wasn't found") |> redirect(to: "/") |> halt()
      message ->
        assign(conn, :message, message)
    end
  end

  defp authorize_message(conn, _) do
    if Authorizer.can_access?(conn.assigns[:user], conn.assigns[:message]) do
      conn
    else
      conn |> put_flash(:info, "You can't access that page") |> redirect(to: "/") |> halt()
    end
  end
end
```

通过使用一系列扁平的plug转换替换嵌套的代码块，我们可以以更加可组合的，清晰的和可复用的方式来实现同样的功能。

现在让我们来看看另一种plugs，即module plugs。

Module Plugs

Module plugs是另一种类型的Plug，可让我们在模块中定义connection转换。该模块仅需实现两个函数：

- `init/1` 初始化任何要传递给 `call/2` 的参数或选项

- `call/2` 执行connection转换。 `call/2` 就是我们之前看到的function plug

为了了解这一点，让我们编写一个module plug，将 `:locale` 键和值放入connection assign中，以供下游的其它plug，控制器动作和视图使用。

```
defmodule HelloWeb.Plugs.Locale do
  import Plug.Conn

  @locales ["en", "fr", "de"]

  def init(default), do: default

  def call(%Plug.Conn{params: %{"locale" => loc}} = conn, _default) when loc in @locales do
    assign(conn, :locale, loc)
  end
  def call(conn, default), do: assign(conn, :locale, default)
end

defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
    plug HelloWeb.Plugs.Locale, "en"
  end
  ...
```

我们可以通过 `plug HelloWeb.Plugs.Locale, "en"` 来将该module plug添加到我们的browser管道中。在 `init/1` 回调中，我们传入一个默认的locale，如果参数中不存在locale，就会使用它。我们还使用模式匹配来定义多个 `call/2` 函数头，来验证locale是否在params中，如果没有匹配到，就会返回“ en”。

这就是Plug的全部内容。Phoenix在整个stack中，都贯彻着plug这种可组合变换的设计。这只是开始。如果我们问自己：“我可以把这个放进plug中吗？” 答案通常是：“可以！”

5. 端点（Endpoint）

Phoenix应用程序启动 `HelloWeb.Endpoint` 作为管理进程。默认情况下，Endpoint被添加到 `lib/hello/application.ex` 管理树作为管理进程。每个请求都在应用程序的Endpoint开始和结束它的生命周期。Endpoint在调用Router之前启动web服务并通过一些定义好的plug转换请求。

```
defmodule Hello.Application do
  use Application
  def start(_type, _args) do
    ...

    children = [
      HelloWeb.Endpoint
    ]

    opts = [strategy: :one_for_one, name: Hello.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Endpoint Contents

Endpoint聚合了通用功能，并充当对应用程序所有HTTP请求的入口和出口。Endpoint持有对进入应用程序的所有请求都通用的plug。

让我们看一下Up and Running这篇教程里为 `Hello` 应用生成的Endpoint。

```
defmodule HelloWeb.Endpoint do
  ...
end
```

Endpoint模块内部的第一个调用是带有 `otp_app` 的 `use Phoenix.Endpoint` 宏。 `otp_app` 用于配置。这在 `HelloWeb.Endpoint` 模块上定义了几个函数，包括了在监视树中被调用的 `start_link` 函数。

```
use Phoenix.Endpoint, otp_app: :hello
```

接下来，Endpoint在"/socket" URI上声明一个socket。"/socket"请求将由应用程序其他地方声明的 `HelloWeb.UserSocket` 模块处理。在这里，我们只是声明这种连接将存在。

```
socket "/socket", HelloWeb.UserSocket,
  websocket: true,
  longpoll: false
```

接下来是一系列与我们应用程序中的所有请求相关的plug。我们可以自定义某些功能，例如，开启 `gzip: true` 在部署到生产环境时使用gzip压缩静态文件。

静态文件在我们请求的任何部分把它发送到router之前，都是从 `priv/static` 提供的。

```
plug Plug.Static,
  at: "/",
  from: :hello,
  gzip: false,
  only: ~w(css fonts images js favicon.ico robots.txt)
```

如果启用了代码重新加载，那么当服务器上的代码更改时，socket将用于通知浏览器需要重新加载页面。在开发环境中，默认情况下启用此功能。使用 `config :hello, HelloWeb.Endpoint, code_reloader: true` 进行配置。

```
if code_reloading? do
  socket "/phoenix/live_reload/socket", Phoenix.LiveReloader.Socket
  plug Phoenix.LiveReloader
  plug Phoenix.CodeReloader
end
```

Plug.RequestId为每个请求生成一个唯一的ID，Plug.Telemetry添加了检测点，因此Phoenix默认可以记录请求路径，状态码和请求时间。

```
plug Plug.RequestId
plug Plug.Telemetry, event_prefix: [:phoenix, :endpoint]
```

Plug.Session处理session cookies和session存储。

```
plug Plug.Session, @session_options
```

默认情况下，Endpoint中的最后一个plug是router。router将路径匹配到特定的控制器动作或plug。router在Routing Guide中有介绍。

```
plug HelloWeb.Router
```

可以自定义Endpoint以添加额外的plug，以允许HTTP basic authentication，CORS，subdomain路由等。

管理树不同部分（例如Ecto Repo）的故障不会立即影响主应用程序。因此，管理员可以在发生意外故障后分别重新启动这些进程。一个应用程序也可能有多个Endpoint，每个Endpoint都有自己的管理树。

Endpoint模块中定义了许多函数，用于路径助手、频道订阅和广播、检测和endpoint配置。这些都涵盖在 `Phoenix.Endpoint` 的 Endpoint API docs 文档中。

Using SSL

为了使应用程序能通过SSL来处理请求，我们需要添加一些配置和两个环境变量。为了使SSL真正起作用，我们需要来自证书颁发机构的密钥文件和证书文件。我们需要的环境变量正是这两个文件的路径。

该配置为我们的endpoint包含了一个新的 `https:` key，该key的值是由密钥文件的路径以及cert（pem）文件的路径组成的关键字列表。如果添加 `otp_app:` key，它的值是我们应用程序名称，Plug将开始在我们应用程序的根目录中查找它们。然后，我们可以将这些文件放在 `priv` 目录中，并将路径设置为 `priv/our_keyfile.key` 和 `priv/our_cert.crt` 。

这是一个 `config/prod.exs` 的配置例子。

```
use Mix.Config
```

```
config :hello, HelloWeb.Endpoint,
 http: [port: {:system, "PORT"}],
 url: [host: "example.com"],
 cache_static_manifest: "priv/static/cache_manifest.json",
 https: [
  port: 443,
  cipher_suite: :strong,
  otp_app: :hello,
  keyfile: System.get_env("SOME_APP_SSL_KEY_PATH"),
  certfile: System.get_env("SOME_APP_SSL_CERT_PATH"),
  # OPTIONAL Key for intermediate certificates:
  cacertfile: System.get_env("INTERMEDIATE_CERTFILE_PATH")
 ]
```

没有 `otp_app:` key，我们需要为这些文件提供它们在文件系统上的绝对路径，以便Plug可以找到它们。

```
Path.expand("../../../some/path/to/ssl/key.pem", __DIR__)
```

`https:` key下的选项将被传递到Plug适配器（通常是 `Plug.Cowboy` ），依次使用 `Plug.SSL` 来选择TLS socket选项。请参阅Plug.SSL.configure/1文档，以获取有关可用选项及其默认值的更多信息。Plug HTTPS Guide和Erlang/OTP ssl 文档同样也提供有价值的信息。

SSL in Development

如果您想在开发环境中使用HTTPS，则可以通过运行 `mix phx.gen.cert` 来生成自签名证书。这需要Erlang/OTP 20或更高版本。

使用您的自签名证书，您在 `config/dev.exs` 中的开发环境配置可以更新以运行HTTPS endpoint：

```
config :my_app, MyApp.Endpoint,
 ...
 https: [
  port: 4001,
  cipher_suite: :strong,
```

```
    keyfile: "priv/cert/selfsigned_key.pem",
    certfile: "priv/cert/selfsigned.pem"
  ]
```

这可以替换您的 `http` 配置，您也可以在不同的端口上运行HTTP和HTTPS服务器。

Force SSL

在许多情况下，您需要通过将HTTP重定向到HTTPS来强制所有传入请求使用SSL。这可以通过在您的endpoint配置中设置 `:force_ssl` 选项来完成。它需要一个被转发到 `Plug.SSL` 的选项列表。默认情况下，它在HTTPS请求中设置"strict-transport-security"请求头，从而强制浏览器始终使用HTTPS。如果发送了不安全的（HTTP）请求，它将使用 `:url` 配置中指定的 `:host` 重定向到HTTPS版本。例如：

```
config :my_app, MyApp.Endpoint,
  force_ssl: [rewrite_on: [:x_forwarded_proto]]
```

要动态重定向当前请求到 `host` ，请将 `:force_ssl` 配置中的 `:host` 设置为 `nil` 。

```
config :my_app, MyApp.Endpoint,
  force_ssl: [rewrite_on: [:x_forwarded_proto], host: nil]
```

在这些示例中， `rewrite_on:` key指定应用程序前的反向代理或负载均衡使用的HTTP header，以指示是通过HTTP还是HTTPS接收到的请求。有关将TLS卸载到外部元素（尤其是与安全的cookie有关）的影响的更多信息，请参阅Plug HTTPS Guide。请记住，在Phoenix应用中，那篇文档中传递给 `Plug.SSL` 的选项应该使用 `force_ssl:` endpoint选项来设置。

HSTS

HSTS或"strict-transport-security"是一种允许网站将其声明为只能通过安全链接（HTTPS）访问的机制。引入它是为了防止中间人剥夺SSL/TLS的攻击。除非链接使用SSL/TLS，否则它将导致Web浏览器从HTTP重定向到HTTPS并拒绝连接。

使用 `force_ssl: :hsts` 来设置 `Strict-Transport-Security` 头，它设置了一个最大的使用期限来定义该策略的有效时长。对于标准情况，现代Web浏览器将通过从HTTP重定向到HTTPS来对此作出响应，但它确实有其他后果。定义了HSTS的RFC6797还指定**浏览器应跟踪主机的策略并应用它直到它过期**。它还指定根据该策略**假定除80以外的任何端口上的通信都是加密的**。

如果您在本地主机上访问应用程序，例如https://localhost:4000，这可能导致意外的行为。因为从那时开始，除了将被重定向443端口的80端口，来自本地主机的通信和转发将被加密。这可能会中断您计算机上正在运行的任何其他本地服务或代理的通信。除非对通信进行加密，否则本地主机上的其他应用程序或代理将拒绝工作。

如果您无意中为本地主机打开了HSTS，则可能需要在浏览器上重置缓存，然后才能接受来自本地主机的任何HTTP通信。对于Chrome，您需要执行 `Empty Cache and Hard Reload` ，从开发者工具面板中单击并按住重新加载图标时出现的重新加载菜单就会出现。对于Safari，您需要清除缓存，从 `~/Library/Cookies/HSTS.plist` 删除条目（或完全删除该文件），然后重新启动Safari。或者您可以设置 `force_ssl` 中应该的 `:expires` 选项为 `0` ，该选项将使关闭HSTS的条目过期。有关HSTS选项的更多信息，请访问Plug.SSL。

6. 控制器（Controllers）

Phoenix控制器充当中介模块。它们的函数——称为动作——被router调用来响应HTTP请求。在调用view层去渲染模板或者返回JSON之前，这些动作将收集所有必要的数据并执行所有必要的步骤。

Phoenix控制器也建立在Plug package上，它们本身就是plug。控制器提供了函数去做我们在action中需要的任何事情。如果我们要寻找一些Phoenix控制器没有提供的东西，那么我们可能正在寻找Plug。请查看Plug Guide或Plug Documentation。

新生成的Phoenix应用中有一个 `PageController` 控制器，它可以在 `lib/hello_web/controllers/page_controller.ex` 中找到，如下所示。

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

  def index(conn, _params) do
    render(conn, "index.html")
  end
end
```

模块定义后的第一行，调用了 `HelloWeb` 模块的 `__using__/1` 宏，它导入了一些有用的模块。

`PageController` 为我们提供了 `index` 动作，显示与Phoenix在路由器中定义的默认路由关联的Phoenix欢迎页面。

Actions

控制器动作就是函数。只要它们遵循Elixir的命名规则，我们就可以任意命名它们。我们必须满足的唯一要求是，动作名称必须与路由器中定义的路由相匹配。

例如，在 `lib/hello_web/router.ex` 中我们可以更改Phoenix新应用提供的默认路由的动作名称：

```
get "/", PageController, :index
```

改为test：

```
get "/", PageController, :test
```

只要我们在 `PageController` 也把动作的名称改为 `test` ，欢迎页面跟之前一样加载。

```
defmodule HelloWeb.PageController do
  ...

  def test(conn, _params) do
    render(conn, "index.html")
  end
end
```

尽管我们可以随意命名动作，但是动作名称有一些约定，我们应该尽可能遵循这些约定。我们在Routing Guide中介绍了这些内容，但在这里再快速过一遍。

- index - 渲染一个给定资源类型的所有item的列表
- show - 根据id渲染单个item
- new - 渲染用于创建新item的表单
- create - 接收一个新item的参数并将其保存在数据存储中
- edit - 按id检索单个item并将其显示在表单中以进行编辑
- update - 接收一个已被编辑的item的参数并将其保存到数据存储中
- delete - 接收要删除的item的id，并将其从数据存储中删除

每个动作都带有两个参数，这些参数由Phoenix在幕后提供。

第一个参数始终为 `conn` ，一个包含了请求相关信息的结构，例如host，path elements，port，query string等。 `conn` 是通过Elixir的Plug中间件框架来实现的。有关更详细的信息，请参考plug's documentation。

第二个参数是params。毫不奇怪地说，它是一个包含HTTP请求中传递的任何参数的map。在函数参项中对参数进行模式匹配是一种很好的做法，以便我们可以通过轻量的数据包传递给渲染所需的数据。当我们在 `lib/hello_web/controllers/hello_controller.ex` 中的 `show` 路由中添加一个messenger参数，我们可以在Adding Pages guide中看到。

```
defmodule HelloWeb.HelloController do
  ...

  def show(conn, %{"messenger" => messenger}) do
    render(conn, "show.html", messenger: messenger)
  end
end
```

某些情况下——通常是在 `index` 动作中——我们不在乎参数，因为我们的行为不依赖于它。在这种情况下，我们不使用传入的参数，在变量名称前加上下划线 `_params` 。这将使编译器不发出“有未使用变量”的警告，同时也保持正确的参数个数。

Gathering Data

虽然Phoenix没有附带自己的数据访问层，但Elixir项目Ecto为使用Postgres关系数据库的用户提供了一个非常好的解决方案。我们在 Ecto Guide中介绍了如何在Phoenix项目中使用Ecto。Usage section of the Ecto README部分介绍了Ecto支持的数据库。

当然，还有许多其他数据访问选项。Ets和Dets是内置在OTP中的键值数据存储。OTP还提供了一种名为[mnesia]的关系数据库，它有自己的查询语言QLC。Elixir和Erlang都有许多库，可用于处理各种流行的数据存储。

数据是您梦想得以实现的世界，但我们不会在这些指南中介绍这些选项。

Flash Messages

有时候，我们需要在操作过程中与用户交互。也许更新schema时出错。也许我们只是想欢迎他们回到应用程序中。为此，我们有flash消息。

`Phoenix.Controller` 模块提供了 `put_flash/3` 和 `get_flash/2` 功能，以帮助我们设置和检索flash消息，以键值对的形式。让我们在 `HelloWeb.PageController` 中设置两个flash消息来尝试一下。

为此，我们将index操作修改如下：

```
defmodule HelloWeb.PageController do
  ...
  def index(conn, _params) do
    conn
    |> put_flash(:info, "Welcome to Phoenix, from flash info!")
    |> put_flash(:error, "Let's pretend we have an error.")
    |> render("index.html")
  end
end
```

`Phoenix.Controller` 模块并不限制我们使用的键。只要内部一致，什么键都可以。 `:info` 和 `:error` 只是常用的罢了。

为了看到我们的flash消息，我们需要检索它们，并在template/layout中显示它们。完成检索的一种方式就是使用 `get_flash/2` ，它的参数是 `conn` 和我们关心的key。然后，它就会返回该key的值。

幸运的是，我们的应用程序布局 `lib/hello_web/templates/layout/app.html.eex` 已经具有用于显示flash消息的标记。

```
<p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
<p class="alert alert-danger" role="alert"><%= get_flash(@conn, :error) %></p>
```

重新加载Welcome Page，我们的消息应该出现在"Welcome to Phoenix!"上方。

除了 `put_flash/3` 和 `get_flash/2` 之外， `Phoenix.Controller` 模块还有另一个值得了解的有用函数。 `clear_flash/1` 函数，仅使用 `conn` 参数删除可能存储在会话中的所有flash消息。

Rendering

控制器有几种渲染内容的方式。最简单的就是使用Phoenix提供的 `text/2` 函数来渲染一些纯文本。

假设我们有一个 `show` 动作，它从参数map接收一个id，然后我们要做的就是返回这个id以及一些文本。我们可以这样做。

```
def show(conn, %{"id" => id}) do
  text(conn, "Showing id #{id}")
end
```

假设我们有一个路由 `get "/our_path/:id"` ，映射到 `show` 这个动作，在浏览器中访问 `/our_path/15` 将会显示 `Showing id 15` 这段不带任何HTML格式的文本。

除此以外的步骤是使用 `json/2` 函数呈现纯JSON 。我们需要传递一些Jason library可以解码为JSON的内容，例如map。（Jason是Phoenix的依赖项之一。）

```
def show(conn, %{"id" => id}) do
  json(conn, %{id: id})
end
```

如果我们再次访问 `our_path/15` ，我们会看到一个JSON块，其键 `id` 映射到数字15。

```
{"id": "15"}
```

Phoenix控制器也可以不使用模板来呈现HTML。没错，就是 `html/2` 函数。这次，我们像这样实现 `show` 动作。

```
def show(conn, %{"id" => id}) do
  html(conn, """
    <html>
      <head>
        <title>Passing an Id</title>
      </head>
      <body>
        <p>You sent in id #{id}</p>
      </body>
    </html>
  """)
end
```

访问 `/our_path/15` ，现在渲染的是在 `show` 中定义的HTML字符串，其中插值了 `15` 。注意我们没有用 `eex` 模板。这是一个多行字符串，所以我们的插值用的是 `#{id}` 而不是 `<%= id %>` 。

值得一提的是 `text/2` ， `json/2` 和 `html/2` 函数，既不需要Phoenix view，也不需要template来渲染。

`json/2` 函数显然对编写API很有用，另外两个可能会派上用场，但是使用我们传入的值将template渲染到layout中是一个很常见的情况。

为此，Phoenix提供了 `render/3` 函数。

有趣的是 `render/3` 是在 `Phoenix.View` 模块中定义的，而不是 `Phoenix.Controller` ，但为方便起见 `Phoenix.Controller` 中有别名。

在 Adding Pages Guide中我们已经看到了render函数。 `lib/hello_web/controllers/hello_controller.ex` 中的 `show` 动作看起来是这样的。

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

  def show(conn, %{"messenger" => messenger}) do
    render(conn, "show.html", messenger: messenger)
  end
end
```

为了使 `render/3` 函数正常工作，控制器必须具有与视图相同的根名称。视图还必须具有与 `show.html.eex` 模板所在的目录相同的根名称。换句话说， `HelloController` 需要 `HelloView` ， `HelloView` 要求存在 `lib/hello_web/templates/hello` 目录，该目录必须包含 `show.html.eex` 模板。

`render/3` 还会将 `show` 动作从params hash中接收到的值传递到模板中进行插值。

如果在使用 `render` 时需要将值传递到模板中，那很容易。我们可以传递我们曾经见过的 `messenger: messenger` 字典，或者我们可以使用 `Plug.Conn.assign/3` 很方便的返回 `conn` 。

```
def index(conn, _params) do
  conn
  |> assign(:message, "Welcome Back!")
  |> render("index.html")
end
```

注意： `Phoenix.Controller` 模块导入了 `Plug.Conn` ，因此使用 `assign/3` 短语就可以了。

我们可以在 `index.html.eex` 模板或布局中访问此消息，通过 `<%= @message %>` 。

将多个值传递到我们的模板中，就像在管道中将 `assign/3` 函数连接在一起一样简单。

```
def index(conn, _params) do
  conn
  |> assign(:message, "Welcome Back!")
  |> assign(:name, "Dweezil")
  |> render("index.html")
end
```

这样， `@message` 和 `@name` 都可以在 `index.html.eex` 模板中可用。

如果我们想要一个默认的欢迎消息，某些操作可以覆盖该怎么办？这很容易，我们只需要在 `conn` 通往控制器动作的途中，使用 `plug` 来转换它即可。

```
plug :assign_welcome_message, "Welcome Back"

def index(conn, _params) do
  conn
  |> assign(:message, "Welcome Forward")
  |> render("index.html")
end

defp assign_welcome_message(conn, msg) do
  assign(conn, :message, msg)
end
```

如果我们只想 `assign_welcome_message` plug适用于一部分动作呢？Phoenix提供了一个解决方案，通过指定一个plug应该适用于哪些动作来完成。

如果我们只想 `plug :assign_welcome_message` 适用于 `index` 和 `show` 操作，则可以这样做。

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

```
  plug :assign_welcome_message, "Hi!" when action in [:index, :show]
...
```

Sending responses directly

如果上面的渲染选项都不能完全满足我们的需求，我们可以使用Plug提供的一些函数来组成自己的渲染。假设我们要发送状态为"201"，无论有没有任何内容的响应。我们可以使用 `send_resp/3` 函数轻松地做到这一点。

```
def index(conn, _params) do
  conn
  |> send_resp(201, "")
end
```

重新加载http://localhost:4000应该会显示一个完全空白的页面。浏览器开发者工具的network一栏中应显示响应状态"201"。

如果我们真的想指定内容的类型，可以用 `put_resp_content_type/2` 与 `send_resp/3` 组合。

```
def index(conn, _params) do
  conn
  |> put_resp_content_type("text/plain")
  |> send_resp(201, "")
end
```

通过这种方式使用Plug函数，我们可以打造自己需要的响应。

但是，渲染并不以模板结尾。默认情况下，模板渲染的结果将插入到布局中，该布局也将被渲染。

Templates and layouts都有自己的指南，所以这里就不多讲了。我们关注的是如何在控制器动作内，指定不同的布局，或者根本不用布局。

Assigning Layouts

布局只是模板的特殊子集。他们在 `lib/hello_web/templates/layout` 里。Phoenix在我们生成应用程序时为我们创建了一个。它称为 `app.html.eex` ，默认情况下所有模板都会在它之中渲染。

由于布局实际上只是模板，因此需要一个视图来呈现它们。那就是定义在 `lib/hello_web/views/layout_view.ex` 里的 `LayoutView` 模块。由于Phoenix为我们生成了此视图，因此我们只要将要渲染的布局放在 `lib/hello_web/templates/layout` 目录中，我们就不必创建一个新视图。

在创建新的布局之前，让我们做一个最简单的事情，渲染一个完全没有布局的模板。

`Phoenix.Controller` 模块的 `put_layout/2` 函数，为我们提供了切换布局的功能。它以 `conn` 作为第一个参数，并以一个字符串表示我们要呈现的布局的基本名称。该函数的另一个子句会为第二个参数匹配将在布尔值 `false` ，这就是我们如何在没有布局的情况下呈现Phoenix欢迎页面的方式。

在新生成的Phoenix应用程序中，编辑 `PageController` 模块 `lib/hello_web/controllers/page_controller.ex` 的 `index` 动作，如下所示。

```
def index(conn, _params) do
  conn
  |> put_layout(false)
  |> render("index.html")
end
```

重新加载http://localhost:4000/之后，我们应该看到一个非常不同的页面，该页面完全没有标题，logo图片或CSS样式。

有一点很重要！对于这种在管道中间被调用的函数，需要用括号包裹参数，因为管道操作绑定得非常紧密。否则很可能导致解析错误和非常奇怪的结果。

如果您得到了如下所示的stack trace，

```
**(FunctionClauseError) no function clause matching in Plug.Conn.get_resp_header/2

Stacktrace
```

```
    (plug) lib/plug/conn.ex:353: Plug.Conn.get_resp_header(false, "content-type")
```

您的参数取代了 `conn` 作为第一个参数，首先要检查的是在正确的位置是否有括号。

这是正确的。

```
def index(conn, _params) do
  conn
  |> put_layout(false)
  |> render("index.html")
end
```

而这是行不通的。

```
def index(conn, _params) do
  conn
  |> put_layout false
  |> render "index.html"
end
```

现在，让我们创建另一个布局并将index模板渲染到其中。作为示例，假设我们的应用程序的admin部分具有不同的布局，其中没有logo图片。为此，让我们把已存在的 `app.html.eex` 内容复制到同一个目录 `lib/hello_web/templates/layout` 中的 `admin.html.eex` 新文件中。然后，我们删除 `admin.html.eex` 显示的logo行。

```
<span class="logo"></span> <!-- remove this line -->
```

然后，将布局的基本名称传递到 `lib/hello_web/controllers/page_controller.ex` 中 `index` 动作的 `put_layout/2`

```
def index(conn, _params) do
conn
|> put_layout("admin.html")
|> render("index.html")
end
```

当我们加载页面时，我们会渲染不带logo的admin布局。

Overriding Rendering Formats

通过模板渲染HTML很好，但是如果我们需要动态更改渲染格式怎么办？假设有时需要HTML，有时需要纯文本，有时需要JSON呢？

Phoenix允许我们使用 `_format` 查询字符串参数动态更改格式。为此，Phoenix需要在正确的目录中使用一个合适命名的视图和一个合适命名的模板。

例如，让我们看看刚生成的应用 `PageController` 中的 `index` 动作。开箱即用，它具有正确的视图 `PageView` ，正确的模板目录 `lib/hello_web/templates/page` 和用于呈现HTML的正确模板 `index.html.eex` 。

```
def index(conn, _params) do
  render(conn, "index.html")
end
```

缺少的是用于呈现文本的替代模板。让我们添加一个 `lib/hello_web/templates/page/index.text.eex` 。这是我们 `index.text.eex` 模板的示例。

```
OMG, this is actually some text.
```

为了完成这项工作，我们还需要做几件事。我们需要告诉我们的router它应该接受 `text` 格式。为此，我们将添加 `text` 到 `:browser` 管道可接受的格式列表中。让我们打开 `lib/hello_web/router.ex` 并修改 `plug :accepts` ，让其包含 `text` 还有 `html` ，像下面这样。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router

  pipeline :browser do
    plug :accepts, ["html", "text"]
    plug :fetch_session
    plug :protect_from_forgery
    plug :put_secure_browser_headers
```

```
  end
...
```

我们还需要告知控制器，在渲染模板时使用和 `Phoenix.Controller.get_format/1` 的返回值相同的模板。为此，我们将"index.html"模板的名称替换为原子版本 `:index` 。

```
def index(conn, _params) do
  render(conn, :index)
end
```

如果我们前往http://localhost:4000/?_format=text，我们将看到 `OMG, this is actually some text.`

当然，我们也可以将数据传递到模板中。让我们更改动作以接收一个message参数，通过删除函数定义中 `params` 前面的 `_` 。这次，我们将使用不太灵活的字符串版本的text模板，只是为了看到它也能正常工作。

```
def index(conn, params) do
  render(conn, "index.text", message: params["message"])
end
```

让我们在文本模板中添加一些内容。

```
OMG, this is actually some text. <%= @message %>
```

现在，如果前往http://localhost:4000/?_format=text&message=CrazyTown，我们将看到"OMG, this is actually some text. CrazyTown"。

Setting the Content Type

类似于 `_format` 查询字符串参数，我们可以通过修改HTTP Content-Type Header并提供适当的模板来渲染所需的任何格式。

如果我们想渲染 `index` 动作的xml版本，则可以在 `lib/hello_web/page_controller.ex` 中实现这样的动作。

```
def index(conn, _params) do
  conn
  |> put_resp_content_type("text/xml")
  |> render("index.xml", content: some_xml_content)
end
```

然后，我们需要提供一个 `index.xml.eex` 创建有效的xml模板，然后就完成了。

有关合法的mime-types列表，请参阅mime type库的mime.types文档。

Setting the HTTP Status

我们还可以像设置内容类型一样设置响应的HTTP状态码。 `Plug.Conn` 模块，已经被导入到所有控制器，它有一个 `put_status/2` 函数执行此操作。

`Plug.Conn.put_status/2` 将 `conn` 作为第一个参数，第二个参数为一个整数或一个我们想要设置成状态码的"friendly name"原子。状态码原子表示的列表可以在 `Plug.Conn.Status.code/1` 文档中找到。

让我们在 `PageController` `index` 动作中更改status。

```
def index(conn, _params) do
  conn
  |> put_status(202)
  |> render("index.html")
end
```

我们提供的状态码必须合法——Cowboy，运行Phoenix的Web服务器，将在无效状态码上引发错误。如果查看开发日志（即iex会话），或者使用浏览器的网络检查工具，我们将会在重新加载页面时看到设置好的状态码。

如果动作发送响应——渲染或重定向——更改代码将不会改变响应的行为。例如，如果将status设置为404或500，然后 `render("index.html")` ，则不会显示错误页面。同样，没有一个300状态码会真的重定向。（即使代码确实影响了行为，它也不知道重定向到哪里。）

`HelloWeb.PageController` `index` 动作的下列实现将不会渲染默认 `not_found` 行为。

```
def index(conn, _params) do
  conn
  |> put_status(:not_found)
  |> render("index.html")
end
```

`HelloWeb.PageController` 中呈现404页面的正确方法是：

```
def index(conn, _params) do
  conn
  |> put_status(:not_found)
  |> put_view(HelloWeb.ErrorView)
  |> render("404.html")
end
```

Redirection

通常，我们需要在请求中间重定向到一个新的url。比如，一个成功的 `create` 动作，通常会为我们刚创建的schema重定向到 `show` 动作。或者，它可以重定向到 `index` 动作以显示同一类型的所有事物。在很多其他情况下，重定向也是有用的。

无论哪种情况，Phoenix控制器都提供了方便的 `redirect/2` 函数，使重定向变得容易。Phoenix区别对待应用内的重定向，和重定向到url——应用内的或外部的。

为了尝试 `redirect/2` ，让我们在 `lib/hello_web/router.ex` 中创建一个新路由。

```
defmodule HelloWeb.Router do
  use HelloWeb, :router
  ...

  scope "/", HelloWeb do
    ...
    get "/", PageController, :index
  end
```

```
  # New route for redirects
  scope "/", HelloWeb do
    get "/redirect_test", PageController, :redirect_test, as: :redirect_test
  end
  ...
end
```

然后，我们将修改 `index` 动作，除了重定向到我们的路由外什么也不做。

```
def index(conn, _params) do
  redirect(conn, to: "/redirect_test")
end
```

最后，让我们在同一文件中定义重定向到的动作，该动作只是呈现文本 `Redirect!` 。

```
def redirect_test(conn, _params) do
  text(conn, "Redirect!")
end
```

重新加载Welcome Page时，我们看到我们已被重定向到 `/redirect_test` ，它渲染了文本呈现文本 `Redirect!` 。成功了！

如果我们关心的话，我们可以打开开发人员工具，单击network选项卡，然后再次访问我们的根路径。我们看到此页面的两个主要请求——一个 `/` 的get请求，状态为的 `302` ，一个 `/redirect_test` 的请求，状态为 `200` 。

请注意，重定向功能还需要 `conn` 和一个代表我们应用程序中的相对路径的字符串。也可以使用 `conn` 和代表完整url的字符串。

```
def index(conn, _params) do
  redirect(conn, external: "https://elixir-lang.org/")
end
```

我们也可以使用在Routing Guide中学到的路径助手。

```
defmodule HelloWeb.PageController do
```

```
  use HelloWeb, :controller

  def index(conn, _params) do
    redirect(conn, to: Routes.redirect_test_path(conn, :redirect_test))
  end
end
```

请注意，我们不能在此处使用url helper，因为 `redirect/2` 使用原子 `:to` ，期望是一个路径。例如，以下操作将失败。

```
def index(conn, _params) do
  redirect(conn, to: Routes.redirect_test_url(conn, :redirect_test))
end
```

如果要使用url helper将完整的url传递到 `redirect/2` ，则必须使用原子 `:external` 。注意使用 `:external` ，url不一定要是外部的。

```
def index(conn, _params) do
  redirect(conn, external: Routes.redirect_test_url(conn, :redirect_test))
end
```

Action Fallback

Action Fallback使我们能够在plugs中集中化错误处理代码，当控制器动作出错返回 `Plug.Conn.t` 时调用。这些plug接收最初传递给控制器动作的conn以及动作的返回值。

假设我们有一个 `show` 动作，该动作使用 `with` 用于获取博客帖子，然后授权当前用户查看该博客帖子。在此示例中我们期望如果找不到该帖子 `Blog.fetch_post/1` 会返回 `{:error, :not_found}` ，如果用户未经授权， `Authorizer.authorize/3` 会返回 `{:error, :unauthorized}` 。我们可以直接为这些非快乐路径呈现错误视图。

```
defmodule HelloWeb.MyController do
  use Phoenix.Controller
  alias Hello.{Authorizer, Blog}
  alias HelloWeb.ErrorView
```

```
  def show(conn, %{"id" => id}, current_user) do
    with {:ok, post} <- Blog.fetch_post(id),
         :ok <- Authorizer.authorize(current_user, :view, post) do

      render(conn, "show.json", post: post)
    else
      {:error, :not_found} ->
        conn
        |> put_status(:not_found)
        |> put_view(ErrorView)
        |> render(:"404")
      {:error, :unauthorized} ->
        conn
        |> put_status(403)
        |> put_view(ErrorView)
        |> render(:"403")
    end
  end
end
```

很多时候——尤其是在为API实现控制器时——像这样在控制器中进行错误处理会导致大量重复。相对而言，我们可以定义一个知道如何处理这些错误情况的plug。

```
defmodule HelloWeb.MyFallbackController do
  use Phoenix.Controller
  alias HelloWeb.ErrorView

  def call(conn, {:error, :not_found}) do
    conn
    |> put_status(:not_found)
    |> put_view(ErrorView)
    |> render(:"404")
  end

  def call(conn, {:error, :unauthorized}) do
    conn
    |> put_status(403)
    |> put_view(ErrorView)
    |> render(:"403")
  end
```

```
  end
```

然后，我们可以使用action_fallback引用该plug，从而很轻松删除 `with` 中的 `else` 代码块。我们的plug将接收原始的conn以及操作结果，并做出适当的响应。

```
defmodule HelloWeb.MyController do
  use Phoenix.Controller
  alias Hello.{Authorizer, Blog}

  action_fallback HelloWeb.MyFallbackController

  def show(conn, %{"id" => id}, current_user) do
    with {:ok, post} <- Blog.fetch_post(id),
         :ok <- Authorizer.authorize(current_user, :view, post) do

      render(conn, "show.json", post: post)
    end
  end
end
```

Halting the Plug Pipeline

正如我们提到的那样——控制器就是plug……特别是在plug pipeline末端被调用的plug。在pipeline的任何步骤中，我们都可能导致停止进程——通常是因为我们已重定向或渲染了响应。 `Plug.Conn.t` 有一个 `:halted` 键——将其设置为true将导致跳过下游plug。我们可以使用 `Plug.Conn.halt/1` 轻松做到这一点。

考虑一个 `HelloWeb.PostFinder` plug。在一个访问中，如果找到与给定id相关的帖子，则将其添加到 `conn.assigns` 中；如果找不到该帖子，我们将显示404页面。

```
defmodule HelloWeb.PostFinder do
  use Plug
  import Plug.Conn

  alias Hello.Blog

  def init(opts), do: opts
```

```
  def call(conn, _) do
    case Blog.get_post(conn.params["id"]) do
      {:ok, post} ->
        assign(conn, :post, post)
      {:error, :notfound} ->
        conn
        |> send_resp(404, "Not found")
    end
  end
end
```

如果我们调用此plug作为plug pipeline的一部分，则任何下游plug仍将被处理。如果我们想阻止在404响应的情况下，下游的plug被处理，我们可以简单地调用 `Plug.Conn.halt/1` 。

```
    ...
    case Blog.get_post(conn.params["id"]) do
      {:ok, post} ->
        assign(conn, :post, post)
      {:error, :notfound} ->
        conn
        |> send_resp(404, "Not found")
        |> halt()
    end
```

请务必注意， `halt/1` 只是将 `Plug.Conn.t` 上的 `:halted` 键设置为true。这足以防止下游插件被调用，但不会停止本地代码的执行.因此

```
conn
|> send_resp(404, "Not found")
|> halt()
```

...在功能上等同于...

```
conn
|> halt()
|> send_resp(404, "Not found")
```

同样需要要注意，halting只会停止plug pipeline的继续。除非实现检查其:halted值，否则function plugs仍将执行。

```
def post_authorization_plug(%{halted: true} = conn, _), do: conn
def post_authorization_plug(conn, _) do
  ...
end
```

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
```

8. 模板（Templates）

模板听起来像是：我们向其中传递数据以形成完整HTTP响应的文件。对于Web应用程序，这些响应通常是完整的HTML文档。对于API，它们通常是JSON或XML。模板文件中的大多数代码通常是标记，但是Phoenix还将有一部分Elixir代码可以编译和求值。Phoenix模板是预编译的，这使得它们非常快。

EEx是Phoenix中的默认模板系统，它与Ruby中的ERB非常相似。它实际上是Elixir本身的一部分，Phoenix使用EEx模板在生成新应用程序时创建了router和主应用程序视图之类的文件。

正如我们在View Guide中所了解的那样，默认情况下，模板位于 `lib/hello_web/templates` 目录中，组织成以试图命名的目录。每个目录都有其自己的视图模块来渲染其中的模板。

例子（Examples）

我们已经看到了使用模板的几种方法，特别是在Adding Pages Guide和Views Guide中。我们可能在这里涵盖了相同的内容，但是我们肯定会添加一些新信息。

hello_web.ex

Phoenix生成一个 `lib/hello_web.ex` 文件，该文件可用于对常见的imports和aliases进行分组。所有在 `view` 块中的声明都会应用与您所有的模板。

让我们对应用程序进行一些补充，以便我们可以做一些试验。

首先，让我们在 `lib/hello_web/router.ex` 中定义一个新路由。

```
defmodule HelloWeb.Router do
  ...

  scope "/", HelloWeb do
    pipe_through :browser

    get "/", PageController, :index
    get "/test", PageController, :test
```

```
  end

  # Other scopes may use custom stacks.
  # scope "/api", Hello do
  #   pipe_through :api
  # end
end
```

现在，让我们定义在路由中指定的控制器动作。我们将在 `lib/hello_web/controllers/page_controller.ex` 文件中添加一个 `test/2` 动作。

```
defmodule HelloWeb.PageController do
  ...

  def test(conn, _params) do
    render(conn, "test.html")
  end
end
```

我们将创建一个函数，该函数告诉我们哪个控制器和动作正在处理我们的请求。

要做到这一点，我们需要在 `lib/hello_web.ex` 中从 `Phoenix.Controller` 导入 `action_name/1` 和 `controller_module/1` 功能。

```
  def view do
    quote do
      use Phoenix.View, root: "lib/hello_web/templates",
                        namespace: HelloWeb

      # Import convenience functions from controllers
      import Phoenix.Controller, only: [get_flash: 1, get_flash: 2, view_module: 1,
                                        action_name: 1, controller_module: 1]

      ...
    end
  end
```

接下来，让我们在 `lib/hello_web/views/page_view.ex` 底部定义一个 `handler_info/1` 函数，该函数使用

我们刚导入的 `controller_module/1` 和 `action_name/1` 函数。我们还将定义一个 `connection_keys/1` 函数稍后使用。

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

  def handler_info(conn) do
    "Request Handled By: #{controller_module(conn)}.#{action_name(conn)}"
  end

  def connection_keys(conn) do
    conn
    |> Map.from_struct()
    |> Map.keys()
  end
end
```

我们有一个路由。我们创建了一个新的控制器动作。我们已经对应用程序主视图进行了修改。现在，我们需要的是一个新模板，以显示从 `handler_info/1` 中获取的字符串。让我们创建一个新的 `lib/hello_web/templates/page/test.html.eex` 。

```
<div class="phx-hero">
  <p><%= handler_info(@conn) %></p>
</div>
```

请注意， `@conn` 通过 `assigns` 映射可自由地在模板中使用。

如果我们访问localhost:4000/test，我们将看到页面是由 `Elixir.HelloWeb.PageController.test` 发出的。

我们可以在 `lib/hello_web/views` 中的任意一个视图中定义函数。在单个视图中定义的函数仅可用于该视图渲染的模板。例如，上述的 `handler_info` 函数仅适用于 `lib/hello_web/templates/page` 模板。

显示列表（Displaying Lists）

到目前为止，我们仅在模板中显示了单一种类值——此处为字符串，其他指南中为整数。我们

将如何显示列表的所有元素？

答案是我们可以使用Elixir的列表推导方法。

现在我们有了一个对模板可见的函数，该函数在 `conn` 结构中返回一个keys列表，我们要做的就是稍微修改 `lib/hello_web/templates/page/test.html.eex` 模板以显示它们。

我们可以像这样添加一个header和一个列表推导。

```
<div class="phx-hero">
  <p><%= handler_info(@conn) %></p>

  <h3>Keys for the conn Struct</h3>

  <%= for key <- connection_keys(@conn) do %>
    <p><%= key %></p>
  <% end %>
</div>
```

我们使用 `connection_keys` 函数返回的keys列表作为迭代的源列表。请注意，在列表推导的第一行和用于显示key的哪一行，我们都需要 `<%=` 中的 `=` 。没有它们，实际上什么也不会示。

当我们再次访问localhost:4000/test时，我们会看到所有的key都显示出来了。

在模板中渲染模板（Render templates within templates）

在上面的列表推导示例中，实际显示值的部分非常简单。

```
<p><%= key %></p>
```

我们可以将其保留在原处。但是，这种显示代码通常会更复杂一些，并且将其置于列表推导的中间位置会使我们的模板难以阅读。

简单的解决方案是使用另一个模板！模板只是函数调用，因此，就像常规代码一样，由小的、目的明确的函数组成更大的模板可以使设计更清晰。这就是我们已经看到的渲染链的简单延续。布局是将常规模板渲染到其中的模板。常规模板可能还具有其他模板。

让我们将此显示代码片段转换为自己的模板。让我们在 `lib/hello_web/templates/page/key.html.eex` 上创建一个新的模板文件，像这样。

```
<p><%= @key %></p>
```

我们需要更改此处的 `key` 为 `@key` ，因为这是一个新模板，而不是列表推导的一部分。我们将数据传递到模板的方式是通过 `assigns` 映射，而我们从 `assigns` 映射中取出值的方式是在keys前面加上 `@` 。 `@` 实际上是一个转换 `@key` 为 `Map.get(assigns, :key)` 的宏。

现在我们有了一个模板，我们只需在 `test.html.eex` 模板的列表推导中渲染它即可。

```
<div class="phx-hero">
  <p><%= handler_info(@conn) %></p>

  <h3>Keys for the conn Struct</h3>

  <%= for key <- connection_keys(@conn) do %>
    <%= render("key.html", key: key) %>
  <% end %>
</div>
```

让我们再次看看localhost:4000/test。该页面应看起来像以前一样。

跨视图共享模板（Shared Templates Across Views）

通常，我们发现小块数据需要在应用程序的不同部分中以相同的方式渲染。最好将这些模板移到它们自己的共享目录中，以指示它们应该在应用程序中的任何位置都可用。

让我们将模板移到共享视图中。

`key.html.eex` 当前由 `HelloWeb.PageView` 模块渲染，但是我们使用render调用，它假定当前schema就是我们要渲染的对象。我们可以使它明确，然后像这样重写它：

```
<div class="phx-hero">
  ...

  <%= for key <- connection_keys(@conn) do %>
```

```
    <%= render(HelloWeb.PageView, "key.html", key: key) %>
  <% end %>
</div>
```

由于我们希望它位于一个新 `lib/hello_web/templates/shared` 目录中，因此我们需要一个新的单独视图以在该目录中渲染 `lib/hello_web/views/shared_view.ex` 模板。

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

现在我们可以移动 `key.html.eex` 从 `lib/hello_web/templates/page` 目录到 `lib/hello_web/templates/shared` 目录。一旦发生这种情况，我们可以将 `lib/hello_web/templates/page/test.html.eex` 中的 `render` 调用更改为使用新的 `HelloWeb.SharedView` 。

```
<%= for key <- connection_keys(@conn) do %>
  <%= render(HelloWeb.SharedView, "key.html", key: key) %>
<% end %>
```

再次回到localhost:4000/test。该页面应看起来跟以前一样。