# 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
```