# 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！”