目　录

致谢

当前文档 《Play 框架中文文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-18。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/play-for-scala-developers

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

Play 中文文档

Play 的 Scala API 在 play.api 包中。

> *play* 包中的 *API* 是为 *Java* 开发者准备的，比如 *play.mvc*。对于 *Scala* 开发者，请使用 *play.api.mvc*。

参与翻译

首先，你得有个 Github 账号，然后：

1. 打开这个项目，猛戳「Fork」按钮
2. 把 Fork 到你账号下的项目 Clone 到本地：git clone git@github.com:你的账号名/play-for-scala-developers.git
3. 在项目目录下，创建一个新分支来工作，比如新分支名叫 dev，则：git branch dev
4. 切换到新分支：git checkout dev
5. 运行 git remote add upstream https://github.com/Hawstein/play-for-scala-developers.git 把原始项目库添加为上游库，用于同步最新内容
6. 在 dev 分支下进行翻译工作，比如你在 ScalaRock.md 上做

修改（Play Doc: https://www.playframework.com/documentation/2.3.x/ScalaHome）

7. 提交你的工作：git add ScalaRock.md，然后 git commit -m ‘翻译 ScalaRock’。在这些过程中，你都可以用 git status 查看状态
8. 运行 git remote update 更新
9. 运行 git fetch upstream master 拉取上游库的更新到本地
0. 运行 git rebase upstream/master 将上游库的更新合并到你的 dev 分支
1. 运行 git push origin dev:master 将你的提交 push 到你的库中
2. 登录 Github，在你 Fork 的项目页有个 「Pull Request」按钮，点击它，填写一些说明，然后提交
3. 重复步骤 6～12

为了避免多人重复翻译同一章节，在翻译前先进行章节的认领。可以在帖子 http://scalachina.org/topic/5501c59784ddfe6644e8c8d4 回复里认领相应章节，也可以加 QQ 群，在群里说。群号：312213800。

认领及完成情况见：http://scalachina.org/topic/5501c59784ddfe6644e8c8d4

术语对照表

下面是翻译文档的术语对照表，如有疑义或更好的翻译选择，欢迎提交 Pull Requests。

英文	中文
controller	控制器
type class	类型类

HTTP programming

Actions, Controllers and Results

什么是 Action

Play 应用收到的大部分请求都是由 `Action` 来处理的。

`play.api.mvc.Action` 就是一个处理收到的请求然后产生结果发给客户端的函数（`play.api.mvc.Request => play.api.mvc.Result`）。

```
1. val echo = Action { request =>
2.   Ok("Got request [" + request + "]")
3. }
```

Action 返回一个类型为 `play.api.mvc.Result` 的值，代表了发送到 web 客户端的 HTTP 响应。在这个例子中，`OK` 构造了一个 **200 OK** 的响应，并包含一个 **text/plain** 类型的响应体。

构建一个 Action

`play.api.mvc.Action` 的伴生对象（companion object）提供了一些 helper 方法来构造一个 Action 值。

最简单的一个函数是 `OK`，它接受一个表达式块作为参数并返回一个 `Result`：

```
1. Action {
2.   Ok("Hello world")
3. }
```

这是构造 Action 的最简单方法，但在这种方法里，我们并没有使用传进来的请求。实际应用中，我们常常要使用调用这个 Action 的 HTTP 请求。

因此，还有另一种 Action 构造器，它接受一个函数 `Request => Result` 作为输入：

```
1. Action { request =>
2.   Ok("Got request [" + request + "]")
3. }
```

实践中常常把 `request` 标记为 `implicit`，这样一来，其它需要它的 API 能够隐式使用它：

```
1. Action { implicit request =>
2.   Ok("Got request [" + request + "]")
3. }
```

最后一种创建 Action 的方法是指定一个额外的 `BodyParser` 参数：

```
1. Action(parse.json) { implicit request =>
2.   Ok("Got request [" + request + "]")
3. }
```

这份手册后面会讲到 Body 解析器（Body Parser）。现在你只需要

知道，上面讲到的其它构造 Action 的方法使用的是一个默认的解析器：任意内容 **body** 解析器（Any content body parser）。

控制器（Controller）是 action 生成器

一个 `Controller` 就是一个产生 `Action` 值的单例对象。

定义一个 action 生成器的最简单方式就是定义一个无参方法，让它返回一个 `Action` 值：

```
1.  package controllers
2.
3.  import play.api.mvc._
4.
5.  object Application extends Controller {
6.
7.    def index = Action {
8.      Ok("It works!")
9.    }
10.
11. }
```

当然，生成 action 的方法也可以带参数，并且这些参数可以在 `Action` 闭包中访问到：

```
1.  def hello(name: String) = Action {
2.    Ok("Hello " + name)
3.  }
```

简单结果

到目前为止，我们就只对简单结果感兴趣：HTTP 结果。它包含了一个状态码，一组 HTTP 报头和发送给 web 客户端的 body。

这些结果由 `play.api.mvc.Result` 定义：

```
def index = Action {
  Result(
    header = ResponseHeader(200, Map(CONTENT_TYPE ->
"text/plain")),
    body = Enumerator("Hello world!".getBytes())
  )
}
```

当然，Play 提供了一些 helper 方法来构造常见结果，比如说 `OK`。下面的代码和上面的代码是等效的：

```
def index = Action {
  Ok("Hello world!")
}
```

上面两段代码产生的结果是一样的。

下面是生成不同结果的一些例子：

```
val ok = Ok("Hello world!")
val notFound = NotFound
val pageNotFound = NotFound(<h1>Page not found</h1>)
val badRequest = BadRequest(views.html.form(formWithErrors))
val oops = InternalServerError("Oops")
val anyStatus = Status(488)("Strange response type")
```

上面的 helper 方法都可以在 `play.api.mvc.Results` 特性（trait）和伴生对象（companion object）中找到。

重定向也是简单结果

重定向到一个新的 URL 是另一种简单结果。然而这些结果类型并不包

含一个响应体。

同样地，有一些 helper 方法可以来创建重定向结果：

```
1.  def index = Action {
2.    Redirect("/user/home")
3.  }
```

默认使用的响应类型是： `303 SEE_OTHER` ，当然，如果你有需要，可以自己设定状态码：

```
1.  def index = Action {
2.    Redirect("/user/home", MOVED_PERMANENTLY)
3.  }
```

「TODO」 dummy 页面

你可以使用一个定义为 `TODO` 的空的 `Action` 实现，它的结果是一个标准的 `Not implemented yet` 页面：

```
1.  def index(name:String) = TODO
```

HTTP routing

HTTP routing

内建 HTTP 路由

路由是一个负责将传入的 HTTP 请求转换为 Action 的组件。

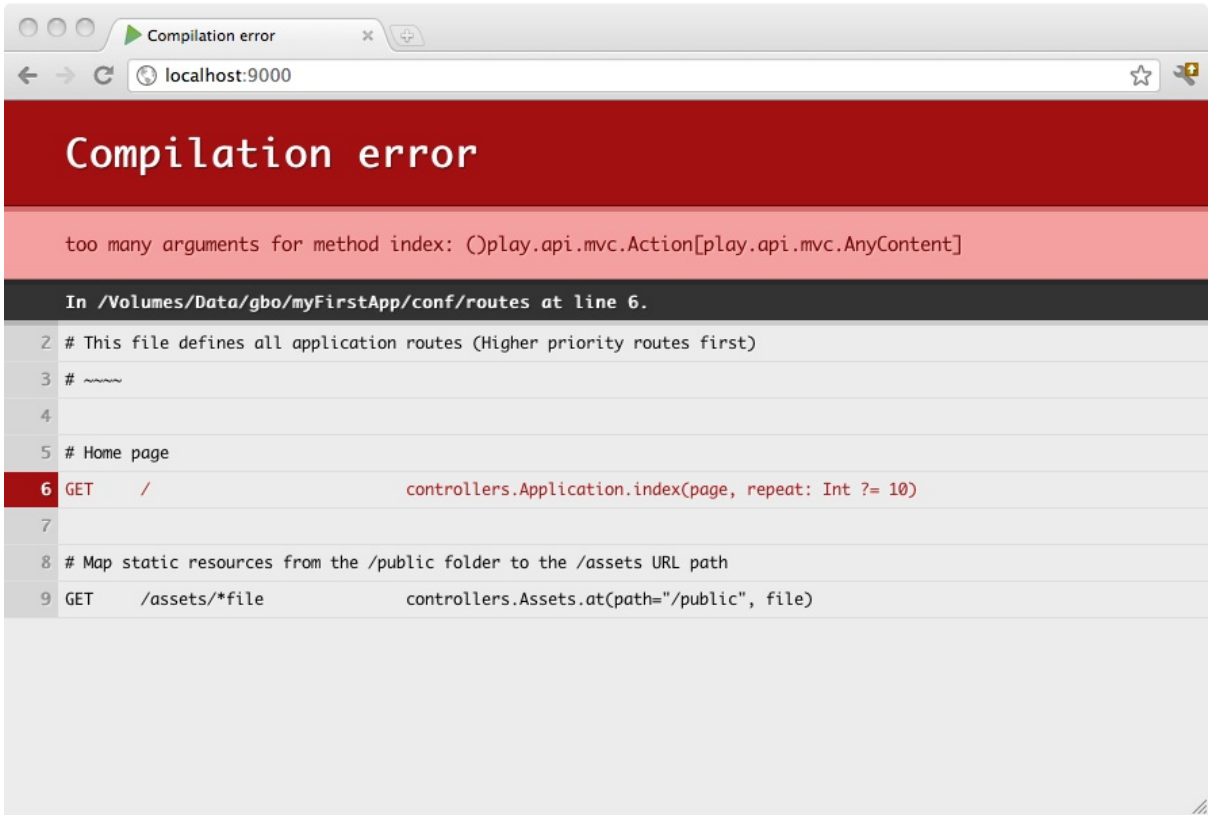
一个 HTTP请求通常被 MVC 框架视为一个事件（event）。这个事件包含了两个主要信息：

- 请求路径（例如：`/clients/1542`，`/photos/list`），其中包括了

查询串（如 ?page=1&max=3）

- HTTP 方法（例如：GET，POST 等）

路由定义在了 `conf/routes` 文件里，该文件会被编译。也就是说你可以直接在你的浏览器中看到这样的路由错误：



路由文件的语法

Play 的路由使用 `conf/routes` 作为配置文件。这个文件列出了该应用需要的所有路由。每个路由都包含了一个 HTTP 方法和一个 URI 模式，两者被关联到一个 Action 生成器。

让我们来看下路由的定义：

```
1.  GET   /clients/:id          controllers.Clients.show(id: Long)
```

每个路由都以 HTTP 方法开头，后面跟着一个 URI 模式。最后是一

个方法的调用。

你也可以使用 `#` 字符添加注释：

```
1. # 显示一个 client
2. GET   /clients/:id           controllers.Clients.show(id: Long)
```

HTTP 方法

HTTP 方法可以是任何 HTTP 支持的有效方法（`GET`，`POST`，`PUT`，`DELETE`，`HEAD`）。

URI 模式

URI 模式定义了路由的请求路径。其中部分的请求路径可以是动态的。

静态路径

例如，为了能够准确地匹配 `GET /clients/all` 请求，你可以这样定义路由：

```
1. GET   /clients/all           controllers.Clients.list()
```

动态匹配

如果你想要在定义的路由中根据 ID 获取一个 client，则需要添加一个动态部分：

```
1. GET   /clients/:id           controllers.Clients.show(id: Long)
```

> 注意：一个 *URI* 模式可以有多个动态的部分。

动态部分的默认匹配策略是通过正则表达式 `[^/]+` 定义的，这意味着任何被定义为 `:id` 的动态部分只会匹配一个 URI。

跨 / 字符的动态匹配

如果你想要一个动态部分能够匹配多个由 `/` 分隔开的 URI 路径，你可以用 `*id` 来定义，此时匹配的正则表达式则为 `.+`：

```
1.  GET   /file/*name           controllers.Application.download(name)
```

对于像 `GET /files/images/logo.png` 这样的请求，动态部分 `name` 匹配到的是 `images/logo.png`。

使用自定义正则表达式做动态匹配

你也可以为动态部分自定义正则表达式，语法为 `$id<regex>`：

```
1.  GET   /items/$id<[0-9]+>     controllers.Items.show(id: Long)
```

调用 Action 生成器方法

路由定义的最后一部分是个方法的调用。这里必须定义一个返回 `play.api.mvc.Action` 的合法方法，一般是一个控制器 action 方法。

如果该方法不接收任何参数，则只需给出完整的方法名：

```
1.  GET   /                     controllers.Application.homePage()
```

如果这个 action 方法定义了参数，Play 则会在请求 URI 中查找所有参数。从 URI 路径自身查找，或是在查询串里查找：

```
# 从路径中提取参数 page
GET   /:page                controllers.Application.show(page)
```

```
# 从查询串中提取参数 page
GET   /                     controllers.Application.show(page)
```

这是定义在 `controllers.Application` 里的 `show` 方法：

```
def show(page: String) = Action {
  loadContentFromDatabase(page).map { htmlContent =>
    Ok(htmlContent).as("text/html")
  }.getOrElse(NotFound)
}
```

参数类型

对于类型为 `String` 的参数来说，可以不写参数类型。如果你想要 Play 帮你把传入的参数转换为特定的 Scala 类型，则必须显式声明参数类型：

```
GET   /clients/:id          controllers.Clients.show(id: Long)
```

同样的，`controllers.Clients` 里的 `show` 方法也需要做相应更改：

```
def show(id: Long) = Action {
  Client.findById(id).map { client =>
    Ok(views.html.Clients.display(client))
  }.getOrElse(NotFound)
}
```

设定参数固定值

有时候，你会想给参数设置一个固定值：

```
# 从路径中提取参数 page，或是为 / 设置固定值
GET   /                     controllers.Application.show(page = "home")
GET   /:page                controllers.Application.show(page)
```

设置参数默认值

你也可以给参数提供一个默认值，当传入的请求中找不到任何相关值时，就使用默认参数：

```
# 分页链接，如 /clients?page=3
GET   /clients              controllers.Clients.list(page: Int ?= 1)
```

可选参数

同样的，你可以指定一个可选参数，可选参数可以不出现在请求中：

```
# 参数 version 是可选的。如 /api/list-all?version=3.0
GET   /api/list-all         controllers.Api.list(version: Option[String])
```

路由优先级

多个路由可能会匹配到同一个请求。如果出现了类似的冲突情况，第一个定义的路由（以定义顺序为准）会被启用。

反向路由

路由同样可以通过 Scala 的方法调用来生成 URL。这样做能够将所有的 URI 模式定义在一个配置文件中，让你在重构应用时更有把握。

Play 的路由会为路由配置文件里定义的每个控制器，在 `routes` 包

中生成一个「反向控制器」，其中包含了同样的 action 方法和签名，不过返回值类型为 `play.api.mvc.Call` 而非 `play.api.mvc.Action`。

`play.api.mvc.Call` 定义了一个 HTTP 调用，并且提供了 HTTP 请求方法和 URI。

例如，如果你创建了这样一个控制器：

```
1.  package controllers
2.
3.  import play.api._
4.  import play.api.mvc._
5.
6.  object Application extends Controller {
7.
8.    def hello(name: String) = Action {
9.      Ok("Hello " + name + "!")
10.   }
11. }
```

并且在 `conf/routes` 中设置了该方法：

```
1.  # Hello action
2.  GET   /hello/:name          controllers.Application.hello(name)
```

接着你就可以调用 `controllers.routes.Application` 的 `hello` action 方法，反向得到相应的 URL：

```
1.  // 重定向到 /hello/Bob
2.  def helloBob = Action {
3.    Redirect(routes.Application.hello("Bob"))
4.  }
```

Manipulating results

Manipulating results

改变默认的 Content-Type

Content-Type 能够从你所指定的响应体的 Scala 值自动推断出来。

例如：

```
1.  val textResult = Ok("Hello World!")
```

将会自动设置 `Content-Type` 报头为 `text/plain`， 而：

```
1.  val xmlResult = Ok(<message>Hello World!</message>)
```

会设置 Content-Type 报头为 `application/xml`。

> 提示：这是由 *`play.api.http.ContentTypeOf`* 类型类（*type class*）完成的。

这相当有用，但是有时候你想去改变它。只需要调用 Result 的 `as(newContentType)` 方法来创建一个新的、类似的、具有不同 `Content-Type` 报头的 Result：

```
1.  val htmlResult = Ok(<h1>Hello World!</h1>).as("text/html")
```

或者用下面这种更好的方式：

```
1.  val htmlResult2 = Ok(<h1>Hello World!</h1>).as(HTML)
```

> 注意：使用 `HTML` 代替 `"text/html"` 的好处是会为你自动处理字符集，这时实际的 *Content-Type* 报头会被设置为 `text/html; charset=utf-8` 。稍后我们就能看到。

处理 HTTP 报头

你还能添加（或更新）结果的任意 HTTP 报头：

```
1.  val result = Ok("Hello World!").withHeaders(
2.    CACHE_CONTROL -> "max-age=3600",
3.    ETAG -> "xx")
```

需要注意的是，如果一个 Result 已经有一个 HTTP 报头了，那么新设置的会覆盖前面的。

设置和丢弃 Cookie

Cookie 是一种特殊的 HTTP 报头，但是我们提供了一系列 helper 方法来简化操作。

你可以像下面那样很容易地添加 Cookie 到 HTTP 响应中：

```
1.  val result = Ok("Hello world").withCookies(
2.    Cookie("theme", "blue"))
```

或者，要丢弃先前存储在 Web 浏览器中的 Cookie：

```
1.  val result2 = result.discardingCookies(DiscardingCookie("theme"))
```

你也可以在同一个响应中同时添加和丢弃 Cookie：

```
1.  val result3 = result.withCookies(Cookie("theme",
    "blue")).discardingCookies(DiscardingCookie("skin"))
```

改变基于文本的 HTTP 响应的字符集

对于基于文本的 HTTP 响应，正确处理好字符集是很重要的。Play 处理它的方式是采用 `utf-8` 作为默认字符集。

字符集一方面将文本响应转换成相应的字节来通过网络 Socket 进行传输，另一方面用正确的 `;charset=xxx` 扩展来更新 `Content-Type` 报头。

字符集由 `play.api.mvc.Codec` 类型类自动处理。仅需要引入一个隐式的 `play.api.mvc.Codec` 实例到当前作用域中，从而改变各种操作所用到的字符集：

```
1.  object Application extends Controller {
2.
3.    implicit val myCustomCharset = Codec.javaSupported("iso-8859-1")
4.
5.    def index = Action {
6.      Ok(<h1>Hello World!</h1>).as(HTML)
7.    }
8.
9.  }
```

这里，因为作用域中存在一个隐式字符集的值，它会被应用到 `Ok(...)` 方法来将 XML 消息转化成 `ISO-8859-1` 编码的字节，同时也用于生成 `text/html;charset=iso-8859-1` Content-Type 报头。

现在，如果你想知道 `HTML` 方法是怎么工作的，以下是它的定义：

```
1.  def HTML(implicit codec: Codec) = {
```

```
2.   "text/html; charset=" + codec.charset
3. }
```

如果你的 API 中需要以通用的方式来处理字符集，你可以按照上述方法进行操作。

Session and Flash scopes

Session 和 Flash 域

它们在 Play 中有何不同

如果你必须跨多个 HTTP 请求来保存数据，你可以把数据存在 Session 或是 Flash 域中。存储在 Session 中的数据在整个会话期间可用，而存储在 Flash 域中数据只对下一次请求有效。

Session 和 Flash 数据不是由服务器来存储，而是以 Cookie 机制添加到每一次后续的 HTTP 请求。这也就意味着数据的大小是很受限的（最多 4KB），而且你只能存储字符串类型的值。在 Play 中默认的 Cookie 名称是 `PLAY_SESSION`。默认的名称可以在 application.conf 中通过配置 `session.cookieName` 的值来修改。

> 如果 *Cookie* 的名字改变了，可以使用上一节中「设置和丢弃 *Cookie*」提到的同样的方法来使之前的 *Cookie* 失效。

当然了，Cookie 值是由密钥签名了的，这使得客户端不能修改 Cookie 数据（否则它会失效）。

Play 的 Session 不能当成缓存来用。假如你需要缓存与某一会话

相关的数据，你可以使用 Play 内置的缓存机制并在用户会话中存储一个唯一的 ID 与之关联。

> 技术上来说，*Session* 并没有超时控制，它在用户关闭 *Web* 浏览器后就会过期。如果你的应用需要一种功能性的超时控制，那就在用户 *Session* 中存储一个时间戳，并在应用需要的时候用它（例如：最大的会话持续时间，最大的非活动时间等）。

存储数据到 Session 中

由于 Session 是个 Cookie 也是个 HTTP 头，所以你可用操作其他 Result 属性那样的方式操作 Session 数据：

```
1.  Ok("Welcome!").withSession(
2.    "connected" -> "user@gmail.com")
```

这会替换掉整个 Session。假如你需要添加元素到已有的 Session 中，方法是先添加元素到传入的 Session 中，接着把它作为新的 Session：

```
1.  Ok("Hello World!").withSession(
2.    request.session + ("saidHello" -> "yes"))
```

你可以用同样的方式从传入的 Session 中移除任何值：

```
1.  Ok("Theme reset!").withSession(
2.    request.session - "theme")
```

从 Session 中读取值

你可以从 HTTP 请求中取回传入的 Session：

```
1.  def index = Action { request =>
2.    request.session.get("connected").map { user =>
3.      Ok("Hello " + user)
```

```
  }.getOrElse {
    Unauthorized("Oops, you are not connected")
  }
}
```

丢弃整个 Session

有一个特殊的操作用来丢弃整个 Session:

```
Ok("Bye").withNewSession
```

Flash 域

Flash 域工作方式非常像 Session，但有两点不同:

- 数据仅为一次请求而保留。
- Flash 的 Cookie 未被签名，这留给了用户修改它的可能。

> *Flash* 域只应该用于简单的非 *Ajax* 应用中传送 *success/error* 消息。由于数据只保存给下一次请求，而且在复杂的 *Web* 应用中无法保证请求的顺序，所以在竞争条件（*race conditions*）下 *Flash* 可能不那么好用。

下面是一些使用 Flash 域的例子:

```
def index = Action { implicit request =>
  Ok {
    request.flash.get("success").getOrElse("Welcome!")
  }
}

def save = Action {
  Redirect("/home").flashing(
    "success" -> "The item has been created")
}
```

为了在你的视图中使用 Flash 域，需要加上 Flash 域的隐式转换：

```
@()(implicit flash: Flash) ...
@flash.get("success").getOrElse("Welcome!") ...
```

如果出现 `could not find implicit value for parameter flash: play.api.mvc.Flash` 的错误，像下面这样加上 `implicit request=>` 就解决了：

```
1. def index() = Action {
2.   implicit request =>
3.     Ok(views.html.Application.index())
4. }
```

Body parsers

请求体解析器（Body parsers）

什么是请求体解析器？

一个 HTTP 的 PUT 或 POST 请求包含一个请求体。请求体可以是请求头中 `Content-Type` 指定的任何格式。在 Play 中，一个请求体解析器将请求体转换为对应的 Scala 值。

然而，HTTP 请求体可能非常的大，请求体解析器不能等到所有的数据都加载进内存再去解析它们。`BodyParser[A]` 实际上是一个 `Iteratee[Array[Byte], A]`，也就是说它一块一块的接收数据（只要 Web 浏览器在上传数据）并计算出类型为 `A` 的值作为结果。

让我们来看几个例子。

- 一个 **`text`** 类型的请求体解析器能够把逐块的字节数据连接成一个字符串，并把计算得到的字符串作为结果（`Iteratee[Array[Byte], String]`）。
- 一个 **`file`** 类型的请求体解析器能够把逐块的字节数据存到一个本地文件，并以 `java.io.File` 引用作为结果

（`Iteratee[Array[Byte], File]`）。

- 一个 **s3** 类型的请求体解析器能够把逐块的字节数据推送给 Amazon S3 并以 S3 对象 ID 作为结果（`Iteratee[Array[Byte], S3ObjectId]`）。

此外，请求体解析器在开始解析数据之前已经访问了 HTTP 请求头，因此有机会可以做一些预先检查。例如，请求体解析器能检查一些 HTTP 头是否正确设置了，或者检查用户是否有权限上传一个大文件。

> 注意：这就是为什么请求体解析器不是一个真正的 *`Iteratee[Array[Byte], A]`*，确切的说是一个 *`Iteratee[Array[Byte], Either[Result, A]]`*，也就是说它在无法为请求体计算出正确的值时，可以直接发送 *HTTP* 结果（典型的像 *`400 BAD_REQUEST`*、*`412 PRECONDITION_FAILED`* 或者 *`413 REQUEST_ENTITY_TOO_LARGE`*）。

一旦请求体解析器完成了它的工作且返回了类型为 `A` 的值时，相应的 `Action` 函数就会被执行，此时计算出来的请求体的值也已被传入到请求中。

更多关于 Action 的内容

前面我们说 `Action` 是一个 `Request => Result` 函数，这不完全对。让我们更深入地看下 `Action` 这个特性（trait）：

```
1.  trait Action[A] extends (Request[A] => Result) {
2.    def parser: BodyParser[A]
3.  }
```

首先我们看到有一个泛型类 A，然后一个 Action 必须定义一个 `BodyParser[A]`。`Request[A]` 的定义如下：

```
1.  trait Request[+A] extends RequestHeader {
2.    def body: A
3.  }
```

`A` 是请求体的类型，我们可以使用任何 Scala 类型作为请求体，例如 `String`，`NodeSeq`，`Array[Byte]`，`JsonValue`，或是 `java.io.File`，只要我们有一个请求体解析器能够处理它。

总的来说，`Action[A]` 用返回类型为 `BodyParser[A]` 的方法去从 HTTP 请求中获取类型为 `A` 的值，并构建出 `Request[A]` 类型的对象传递给 Action 代码。

默认的请求体解析器：AnyContent

在我们前面的例子中还从未指定过请求体解析器，那它是怎么工作的呢？如果你不指定自己的请求体解析器，Play 就会使用默认的，它把请求体处理成一个 `play.api.mvc.AnyContent` 实例。

默认的请求体解析通过查看 `Content-Type` 报头来决定要处理的请求体类型：

- **text/plain:** `String`
- **application/json:** `JsValue`
- **application/xml, text/xml 或者 application/XXX+xml:** `NodeSeq`
- **application/form-url-encoded:** `Map[String, Seq[String]]`
- **multipart/form-data:** `MultipartFormData[TemporaryFile]`
- 任何其他的类型: `RawBuffer`

例如：

```
1.  def save = Action { request =>
2.    val body: AnyContent = request.body
3.    val textBody: Option[String] = body.asText
4.
5.    // Expecting text body
```

```
6.    textBody.map { text =>
7.      Ok("Got: " + text)
8.    }.getOrElse {
9.      BadRequest("Expecting text/plain request body")
10.   }
11. }
```

指定一个请求体解析器

Play 中可用的请求体解析器定义在 `play.api.mvc.BodyParsers.parse` 中。

例如，定义了一个处理 text 类型请求体的 Action（像前面示例中那样）：

```
1. def save = Action(parse.text) { request =>
2.   Ok("Got: " + request.body)
3. }
```

你知道代码是如何变简单的吗？这是因为如果发生了错误，`parse.text` 这个请求体解析器会发送一个 `400 BAD_REQUEST` 的响应。我们在 Action 代码中没有必要再去做检查。我们可以放心地认为 `request.body` 中是合法的 `String`。

或者，我们也可以这么用：

```
1. def save = Action(parse.tolerantText) { request =>
2.   Ok("Got: " + request.body)
3. }
```

这个方法不会检查 `Content-Type` 报头，而总是把请求体加载为字符串。

> 小贴士：在 *Play* 中所有的请求体解析都提供有 *`tolerant`* 样式的方法。

这是另一个例子，它将把请求体存为一个文件：

```
def save = Action(parse.file(to = new File("/tmp/upload"))) {
request =>
  Ok("Saved the request content to " + request.body)
}
```

组合请求体解析器

在前面的例子中，所有的请求体都会存到同一个文件中。这会产生难以预料的问题，不是吗？我们来写一个定制的请求体解析器，它会从请求会话中得到用户名，并为每个用户生成不同的文件：

```
val storeInUserFile = parse.using { request =>
  request.session.get("username").map { user =>
    file(to = new File("/tmp/" + user + ".upload"))
  }.getOrElse {
    sys.error("You don't have the right to upload here")
  }
}

def save = Action(storeInUserFile) { request =>
  Ok("Saved the request content to " + request.body)
}
```

> 注意：这里我们并没有真正的写自己的请求体解析器，只不过是组合了现有的。这样做通常足够了，能应付多数情况。从头写一个 `请求体解析器` 会在高级主题部分涉及到。

最大内容长度

基于文本的请求体解析器（如 `text` ， `json` ， `xml` 或 `formUrlEncoded` ）会有一个最大内容长度，因为它们要加载所有内容到内存中。

默认的最大内容长度是 100KB，但是你也可以在代码中指定它：

```
1. // Accept only 10KB of data.
2. def save = Action(parse.text(maxLength = 1024 * 10)) { request =>
3.   Ok("Got: " + text)
4. }
```

> 小贴士：默认的内容大小可在 `application.conf` 中定义：`parsers.text.maxLength=128K`

你可以在任何请求体解析器中使用 `maxLength`：

```
1. // Accept only 10KB of data.
2. def save = Action(parse.maxLength(1024 * 10, storeInUserFile)) {
   request =>
3.   Ok("Saved the request content to " + request.body)
4. }
```

Actions composition

Actions composition

这一章引入了几种定义通用action的方法

自定义action构造器

之前，我们已经介绍了几种声明一个action的方法 - 带有请求参数，无请求参数和带有body解析器（body parser）等。事实上还有其他一些方法，我们会在异步编程中介绍。

这些构造action的方法实际上都是有由一个命名为 `ActionBuilder` 的特性（trait）所定义的，而我们用来声明所有action的 `Action` 对象只不过是这个特性（trait）的一个实例。通过实现自己的 `ActionBuilder` ，你可以声明一些可重用的action栈，并以此来构建action。

让我们先来看一个简单的日志装饰器例子。在这个例子中，我们会记录每一次对该action的调用。

第一种方式是在 `invokeBlock` 方法中实现该功能，每个由 `ActionBuilder` 构建的action都会调用该方法：

```
1. import play.api.mvc._
2.
3. object LoggingAction extends ActionBuilder[Request] {
4.   def invokeBlock[A](request: Request[A], block: (Request[A]) =>
   Future[Result]) = {
5.     Logger.info("Calling action")
6.     block(request)
7.   }
8. }
```

现在我们就可以像使用 `Action` 一样来使用它了：

```
1. def index = LoggingAction {
2.   Ok("Hello World")
3. }
```

`ActionBuilder` 提供了其他几种构建action的方式，该方法同样适用于如声明一个自定义body解析器（body parser）等方法：

```
1. def submit = LoggingAction(parse.text) { request =>
2.   Ok("Got a bory " + request.body.length + " bytes long")
3. }
```

组合action

在大多数的应用中，我们会有多个action构造器，有些用来做各种类型的验证，有些则提供了多种通用功能等。这种情况下，我们不想为每个类型的action构造器都重写日志action，这时就需要定义一种可重用的方式。

可重用的action代码可以通过嵌套action来实现：

```
import play.api.mvc._

case class Logging[A](action: Action[A]) extends Action[A] {

  def apply(request: Request[A]): Future[Result] = {
    Logger.info("Calling action")
    action(request)
  }

  lazy val parser = action.parser
}
```

我们也可以使用 `Action` 的action构造器来构建，这样就不需要定义我们自己的action类了：

```
import play.api.mvc._

def logging[A](action: Action[A]) = Action.async(action.parser) {
request =>
  Logger.info("Calling action")
  action(request)
}
```

Action同样可以使用 `composeAction` 方法混入（mix in）到action构造器中：

```
object LoggingAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) =>
Future[Result]) = {
    block(request)
  }
  override def composeAction[A](action: Action[A]) = new
Logging(action)
}
```

现在构造器就能像之前那样使用了：

```
def index = LoggingAction {
  Ok("Hello World")
}
```

我们也可以不用action构造器来混入（mix in）嵌套action:

```
def index = Logging {
  Action {
    Ok("Hello World")
  }
}
```

更多复杂的action

到现在为止，我们所演示的action都不会影响传入的请求。我们当然也可以读取并修改传入的请求对象:

```
import play.api.mvc._

def xForwardedFor[A](action: Action[A]) =
Action.async(action.parser) { request =>
  val newRequest = request.headers.get("X-Forwarded-For").map { xff
=>
    new WrappedRequest[A](request) {
      override def remoteAddress = xff
    }
  } getOrElse request
  action(newRequest)
}
```

```
注意： Play已经内置了对X-Forwarded-For头的支持
```

我们可以阻塞一个请求:

```
import play.api.mvc._
```

```
2.
3.  def onlyHttps[A](action: Action[A]) = Action.async(action.parser) {
    request =>
4.    request.headers.get("X-Forwarded-Proto").collect {
5.      case "https" => action(request)
6.    } getOrElse {
7.      Future.successful(Forbidden("Only HTTPS requests allowed"))
8.    }
9.  }
```

最后，我们还可以修改返回的结果：

```
1.  import play.api.mvc._
2.  import play.api.libs.concurrent.Execution.Implicits._
3.
4.  def addUaHeader[A](action: Action[A]) = Action.async(action.parser)
    { request =>
5.    action(request).map(_.withHeaders("X-UA-Compatible" ->
    "Chrome=1"))
6.  }
```

不同的请求类型

当组合action允许在HTTP请求和响应的层面进行一些额外的操作时，你自然而然的就会想到构建数据转换的管道（pipeline），为请求本身增加上下文（context）或是执行一些验证。你可以把 `ActionFunction` 当做是一个应用在请求上的方法，该方法参数化了传入的请求类型和输出类型，并将其传至下一层。每个action方法可以是一个模块化的处理，如验证，数据库查询，权限检查，或是其他你想要在action中组合并重用的操作。

Play还有一些预定义的特性（trait），它们实现了 `ActionFunction` ，并且对不同类型的操作都非常有用：

- `ActionTransformer`可以更改请求，比如添加一些额外的信息。
- `ActionFilter`可选择性的拦截请求，比如在不改变请求的情况下处理错误。
- `ActionRefiner`是以上两种的通用情况
- `ActionBuilder`是一种特殊情况，它接受`Request`作为参数，所以可以用来构建action。

你可以通过实现 `invokeBlock` 方法来定义你自己的 `ActionFunction` 。通常为了方便，会定义输入和输出类型为 `Request` （使用 `WrappedRequest` ），但这并不是必须的。

验证

Action方法最常见的用例之一就是验证。我们可以简单的实现自己的验证action转换器（transformer），从原始请求中获取用户信息并添加到 `UserRequest` 中。需要注意的是这同样也是一个 `ActionBuilder` ，因为其输入是一个 `Request` ：

```
1.  import play.api.mvc._
2.
3.  class UserRequest[A](val username: Option[String], request:
    Request[A]) extends WrappedRequest[A](request)
4.
5.  object UserAction extends
6.      ActionBuilder[UserRequest] with ActionTransformer[Request,
    UserRequest] {
7.    def transform[A](request: Request[A]) = Future.successful {
8.      new UserRequest(request.session.get("username"), request)
9.    }
10. }
```

Play提供了内置的验证action构造器。更多信息请参考这里

1. 注意：内置的验证action构造器只是一个简便的helper，目的是为了用尽可能少的代码为一些简单的用例添加验证功能，其实现和上面的例子非常相似。
2.
3. 如果你有更复杂的需求，推荐实现你自己的验证action

为请求添加信息

现在让我们设想一个REST API，处理类型为 `Item` 的对象。在 `/item/:itemId` 的路径下可能有多个路由，并且每个都需要查询该 `item` 。这种情况下，将逻辑写在action方法中非常有用。

首先，我们需要创建一个请求对象，将 `Item` 添加到 `UserRequest` 中：

```
import play.api.mvc._

class ItemRequest[A](val item: Item, request: UserRequest[A])
extends WrappedRequest[A](request) {
  def username = request.username
}
```

现在，创建一个action修改器（refiner）查找该item并返回 `Either` 一个错误（ `Left` ）或是一个新的 `ItemRequest` （ `Right` ）。注意这里的action修改器（refiner）定义在了一个方法中，用来获取该item的id:

```
def ItemAction(itemId: String) = new ActionRefiner[UserRequest,
ItemRequest] {
  def refine[A](input: UserRequest[A]) = Future.successful {
    ItemDao.findById(itemId)
      .map(new ItemRequest(_, input))
      .toRight(NotFound)
  }
}
```

验证请求

最后，我们希望有个action方法能够验证是否继续处理该请求。例如，我们可能需要检查 `UserAction` 中获取的user是否有权限使用 `ItemAction` 中得到的item，如果不允许则返回一个错误：

```
1. object PermissionCheckAction extends ActionFilter[ItemRequest] {
2.   def filter[A](input: ItemRequest[A]) = Future.successful {
3.     if (!input.item.accessibleByUser(input.username))
4.       Some(Forbidden)
5.     else
6.       None
7.   }
8. }
```

合并起来

现在我们可以将所有这些action方法链起来（从 `ActionBuilder` 开始）， 使用 `andThen` 来创建一个action:

```
1. def tagItem(itemId: String, tag: String) =
2.   (UserAction andThen ItemAction(itemId) andThen
   PermissionCheckAction) { request =>
3.     request.item.addTag(tag)
4.     Ok("User " + request.username + " tagged " + request.item.id)
5.   }
```

Play同样支持全局过滤API，对于全局的过滤非常有用。

Content negotiation

Content negotiation

内容协商（Content negotiation）这种机制使得将相同的资源（URI）提供为不同的表示这件事成为可能。这一点非常有用，比如说，在写 Web 服务的时候，支持几种不同的输出格式（XML，Json 等）。服务器端驱动的协商是通过使用 `Accept*` 请求报头（header）来做的。你可以在这里找到更多关于内容协商的信息。

语言

你可以通过 `play.api.mvc.RequestHeader#acceptLanguages` 方法来获取针对一个请求的可接受语言列表，该方法从 `Accept-Language` 报头获取这些语言并根据它们的品质值来排序。Play 在 `play.api.mvc.Controller#lang` 方法中使用它，该方法为你的 action 提供了一个隐式的 `play.api.i18n.Lang` 值，因此它会自动选择最可能的语言（如果你的应用支持的话，否则会使用你应用的默认语言）。

内容

与上面相似，`play.api.mvc.RequestHeader#acceptedTypes` 方法给出针对一个请求的可接受结果的 MIME 类型列表。该方法从 `Accept` 请求报头获取这些 MIME 类型并根据它们的品质因子进行排序。

事实上，`Accept` 报头并不是真的包含 MIME 类型，而是媒体种类（比如一个请求如果接受的是所有文本结果，那媒体种类可设置为 `text/*`。而 `*/*` 表示所有类型的结果都是可接受的。）。控制器（Controller）提供了一个高级方法 `render` 来帮助你处理媒体种类。例如，考虑以下的 action 定义：

```
1.  val list = Action { implicit request =>
2.    val items = Item.findAll
3.    render {
4.      case Accepts.Html() => Ok(views.html.list(items))
5.      case Accepts.Json() => Ok(Json.toJson(items))
6.    }
7.  }
```

`Accepts.Html()` 和 `Accepts.Json()` 是两个提取器（extractor），用于测试提供的媒体种类是 `text/html` 还是 `application/json`。`render` 方法接受一个类型为 `play.api.http.MediaRange => play.api.mvc.Result` 的部分函数（partial function）作为参数，并按照优先顺序将它应用在 `Accept` 报头中的每个媒体种类，如果所有可接受的媒体种类都不被你的函数支持，那么会返回一个 `NotAcceptable` 结果。

例如，如果一个客户端发出的请求有如下的 `Accept` 报头：`*/*;q=0.5,application/json`，意味着它接受任意类型的结果，但更倾向于要 JSON 类型的，上面那段代码就会给它返回 JSON 类型的结果。如果另一个客户端发出的请求的 `Accept` 报头是 `application/xml`，这意味着它只接受 XML 的结果，上述代码会返回 `NotAcceptable`。

请求提取器（Request extractors）

参见 API 文档中 `play.api.mvc.AcceptExtractors.Accepts` 对象，了解 Play 在 `render` 方法中所支持的 MIME 类型。使用 `play.api.mvc.Accepting` case 类，你可以很容易地为给定的 MIME 类型创建一个你自己的提取器。例如，下面的代码创建了一个提取器，用于检查媒体类型是否配置 `audio/mp3` MIME 类型。

```
1. val AcceptsMp3 = Accepting("audio/mp3")
2. render {
3.   case AcceptsMp3() => ???
4. }
```

Asynchronous HTTP programming

Handling asynchronous results

构建异步controller

Play框架本身就是异步的。对每一个请求而言，Play都会使用异步，非阻塞的方式来处理。

默认配置下controller即是异步的了。换言之，应用应该避免在controller中做阻塞操作，例如，让controller等待某一个操作完成。类似的阻塞操作还有：调用JDBC，流处理（streaming）API，HTTP请求和长时间的计算任务。

虽然不可能通过增加默认执行环境（execution context）中的线程数来提升阻塞controller的并发请求处理能力，但使用异步controller可以使应用更易于扩展，在负载较大的情况下依然能够保持响应。

创建非阻塞action

由于Play的异步工作方式，action应该做到尽可能的快，例如非阻

塞。那么，在还没有得到返回结果的情况下，应该返回什么作为结果呢？答案是future结果！

一个 `Future[Result]` 最终会被替换成一个类型为 `Result` 的值。通过提供 `Future[Result]` 而非 `Result` ，我们能够在非阻塞的情况下快速生成结果。一旦从promise中得到了最终结果，Play就会应用该结果。

Web客户端在等待响应时依然是阻塞的，但在服务器端没有任何操作被阻塞，而且服务器资源依然可以服务其他客户端。

如何创建 `Future[Result]`

在创建一个 `Future[Result]` 之前，我们需要先创建另一个future：这个future会返回一个真实的值，以便我们依此生成结果：

```
1. import play.api.libs.concurrent.Execution.Implicits.defaultContext
2.
3. val futurePIValue: Future[Double] = computePIAsynchronously()
4. val futureResult: Future[Result] = futurePIValue.map { pi =>
5.   Ok("PI value computed: " + pi)
6. }
```

所有的Play异步API调用都会返回一个 `Future` 。不管你是在调用外部web服务，如 `play.api.libs.WS` API，或是用Akka调度异步任务，亦或是使用 `play.api.libs.Akka` 与actor进行通讯。

以下是一个异步调用并获得一个 `Future` 的简单例子：

```
1. import play.api.libs.concurrent.Execution.Implicits.defaultContext
2.
3. val futureInt: Future[Int] = scala.concurrent.Future {
4.   intensiveComputation()
5. }
```

注意：理解哪个线程运行了future非常重要，以上的两段代码都导入了Play的默认执行环境（execution context）。这是一个隐式（implicit）的参数，会被传入所有接受回调的future API方法中。执行环境（execution context）通常等价于线程池，但这并不是一定的。

简单的把同步IO封装入 `Future` 并不能将其转换为异步的。如果你不能通过改变应用架构来避免阻塞操作，那么该操作总会在某一时刻被执行的，而相应的线程则会被阻塞。所以，除了将操作封装于 `Future` 中，还必须让它运行在配置了足够线程来处理可预计并发的独立执行环境（execution context）中。更多信息请见理解Play线程池

这对于使用Actor来阻塞操作也非常有用。Actor提供了一种非常简洁的模型来处理超时和失败，设置阻塞执行环境（execution context），并且管理了该服务的一切状态。Actor还提供了像 `ScatterGatherFirstCompletedRouter` 这样的模式来处理同时缓存和数据库请求，并且能够远程执行于后端服务器集群中。但使用actor可能有些多余，主要还是取决你想要的是什么。

返回future

迄今为止，我们一直都在调用 `Action.apply` 方法来构建action，为了发出一个异步的结果，我们需要调用 `Action.async` ：

```
1. import play.api.libs.concurrent.Execution.Implicits.defaultContext
2.
3. def index = Action.async {
4.   val futureInt = scala.concurrent.Future { intensiveComputation()
   }
5.   futureInt.map(i => Ok("Got result: " + i))
6. }
```

Action默认即是异步的

Play的action默认即是异步的。比如说，在下面这个controller中，`{ Ok(...) }`这部分代码并不是controller的方法体。这其实是一个传入`Action`对象`apply`方法的匿名函数，用来创建一个`Action`对象。运行时，你写的这个匿名函数会被调用并返回一个`Future`结果。

```
1. val echo = Action { request =>
2.   Ok("Got request [" + request + "]")
3. }
```

注意：`Action.apply`和`Action.async`创建的`Action`对象在Play内部会以同样的方式处理。他们都是异步的`Action`，而不是一个同步一个异步。`.async`构造器只是用来简化创建基于API并返回`Future`的action，让非阻塞的代码更加容易写。

处理超时

能够合理的处理超时通常很有用，当出现问题时可以避免浏览器无谓的阻塞和等待。简单的通过调用promise超时来构造另一个promise便能够处理这些情况：

```
1. import play.api.libs.concurrent.Execution.Implicits.defaultContext
2. import scala.concurrent.duration._
3.
4. def index = Action.async {
5.   val futureInt = scala.concurrent.Future { intensiveComputation()
   }
6.   val timeoutFuture =
   play.api.libs.concurrent.Promise.timeout("Oops", 1.second)
7.   Future.firstCompletedOf(Seq(futureInt, timeoutFuture)).map {
8.     case i: Int => Ok("Got result: " + i)
```

```
 9.       case t: String => InternalServerError(t)
10.     }
11.   }
```

Streaming HTTP responses

Streaming HTTP responses

标准响应和 Content-Length 报头

从 HTTP 1.1 开始，服务器为了保持一个连接的连通并服务多个 HTTP 请求和响应，必须在响应中写入合适的 `Content-Length` HTTP 报头。

一般情况下，当你发送一个简单结果时并不会指定 `Content-Length` 报头，比如：

```
1. def index = Action {
2.   Ok("Hello World")
3. }
```

当然，由于你所发送的内容显而易见，Play能够计算出内容长度并生成适当的报头。

需要注意的是，基于文本的内容长度计算并没有如你所见的这般简单，因为 `Content-Length` 报头的计算取决于将字符转换为字节码的字符编码。

事实上，我们之前看到的响应体都是由

`play.api.libs.iteratee.Enumerator` 所指定的：

```
def index = Action {
  Result(
    header = ResponseHeader(200),
    body = Enumerator("Hello World")
  )
}
```

也就是说，为了能够正确的计算出 `Content-Length` 报头，Play必须读取整个枚举器（enumerator），并将其加载入内存中。

发送大量数据

如果对于简单的枚举器（enumerator）来说，将所有数据加载入内存中并不是个问题，但大数据集呢？假设我们想要返回给 web 客户端一个很大的文件。

让我们先来看看怎么创建一个 `Enumerator[Array[Byte]]` 来枚举整个文件的内容：

```
val file = new java.io.File("/tmp/fileToServe.pdf")
val fileContent: Enumerator[Array[Byte]] =
Enumerator.fromFile(file)
```

这看起来很简单对吧？让我们接着用这个枚举器（enumerator）来指定响应体：

```
def index = Action {

  val file = new java.io.File("/tmp/fileToServe.pdf")
  val fileContent: Enumerator[Array[Byte]] =
Enumerator.fromFile(file)

```

```
  Result(
    header = ResponseHeader(200),
    body = fileContent
  )
}
```

事实上，这里有一个问题。因为我们并没有指定 `Content-Length` 报头，Play 需要自己来计算，唯一的方法就是读取整个枚举器（enumerator）并加载入内存，然后再计算响应的长度。

问题在于我们并不想将整个大文本加载入内存中。为了避免这种情况，我们必须自己来指定 `Content-Length` 报头。

```
def index = Action {

  val file = new java.io.File("/tmp/fileToServe.pdf")
  val fileContent: Enumerator[Array[Byte]] =
Enumerator.fromFile(file)

  Result(
    header = ResponseHeader(200, Map(CONTENT_LENGTH ->
file.length.toString)),
    body = fileContent
  )
}
```

Play 会以一种惰性方式来读取这个枚举器（enumerator），在数据块可用时才将其复制到 HTTP 响应中。

处理文件

当然，Play 提供了简单易用的 helper 来处理本地文件：

```
def index = Action {
  Ok.sendFile(new java.io.File("/tmp/fileToServe.pdf"))
```

```
3. }
```

这个 helper 能够根据文件名计算出 `Content-Type` 报头，并且添加 `Content-Disposition` 报头告诉 web 浏览器该如何处理这个响应。默认会让 web 浏览器下载该文件并在响应中添加 `Content-Disposition: attachment; filename=fileToServe.pdf` 报头。

你也可以指定文件名：

```
1. def index = Action {
2.   Ok.sendFile(
3.     content = new java.io.File("/tmp/fileToServe.pdf"),
4.     fileName = _ => "termsOfService.pdf"
5.   )
6. }
```

如果你想以 `inline` 的方式处理该文件：

```
1. def index = Action {
2.   Ok.sendFile(
3.     content = new java.io.File("/tmp/fileToServe.pdf"),
4.     inline = true
5.   )
6. }
```

这样就不用指定文件名了，因为 web 浏览器根本不会尝试下载，而是将所有文本显示在 web 浏览器窗口中。这对于那些浏览器本身已经支持了的文本类型非常有用，如文本，html 和图片。

分块响应

到现在为止，流处理文件内容工作的非常好，主要是因为能够在流处理之前算出文本长度。但是如果是动态计算，还没有得出长度的内容呢？

对于这样的响应，我们需要使用 分块传输编码 （Chunked transfer encoding）。

分块传输编码（Chunked transfer encoding）是一种定义在 Hypertext Transfer Protocol（HTTP）1.1 版本中的数据传输机制，其中 web 服务器会将文本分块处理。它使用了 `Transfer-Encoding` HTTP 响应报头而非 `Content-Length` 报头，如果你没有使用 `Content-Length` 的话，则必须使用这个报头。由于没有 `Content-Length`，服务器无需在开始传输响应到客户端（通常是 web 客户端）之前就知道内容的长度。 Web 服务器在知道内容总长前就能够传输动态生成的内容响应。

在发送每个数据块前，都会先发送块的大小，客户端能够依此判断是否已完整接收该数据块。数据传输会在收到一个长度为零的数据块后终结。

http://en.wikipedia.org/wiki/Chunked_transfer_encoding

这样做的好处是能够实时处理数据，一旦数据可用我们便会发送。缺点是由于浏览器不知道内容长度，无法显示正确的下载进度。

假设我们有个服务，提供了一个动态 `InputStream` 来计算一些数据。首先我们需要为这个流创建一个 `Enumerator`：

```
1.  val data = getDataStream
2.  val dataContent: Enumerator[Array[Byte]] =
    Enumerator.fromStream(data)
```

这样就能使用 `ChunkedResult` 来流处理这些数据了：

```
1.  def index = Action {
2.
```

```
  val data = getDataStream
  val dataContent: Enumerator[Array[Byte]] =
Enumerator.fromStream(data)

  ChunkedResult(
    header = ResponseHeader(200),
    chunks = dataContent
  )
}
```

Play 同样提供了一些 helper ：

```
def index = Action {

  val data = getDataStream
  val dataContent: Enumerator[Array[Byte]] =
Enumerator.fromStream(data)

  Ok.chunked(dataContent)
}
```

当然，我们可以使用任一 `Enumerator` 来指定数据块：

```
def index = Action {
  Ok.chunked(
    Enumerator("kiki", "foo", "bar").andThen(Enumerator.eof)
  )
}
```

我们可以检查服务器发回的 HTTP 响应：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

4
kiki
```

```
3
foo
3
bar
0
```

我们得到了三个数据块，最后还跟了一个空数据块用来结束响应。

Comet sockets

Comet sockets

使用分块响应来创建 comet sockets

分块响应（chunked responses）的一个好处是可以用来创建 comet sockets。comet socket 是一块只包含 `<script>` 元素的 `text/html` 响应。在每一个块中，我们都写入一个 `<script>` 标签，这样一来，它会被浏览器立即执行。通过这种方式，我们可以实时地从服务器发送各种事件到浏览器：将每条信息包在一个 `<script>` 标签中（它会调用一个 JavaScript 回调函数），然后将它写到响应块里。

让我们通过以下例子来验证上面的概念：下面的枚举器（enumerator）产生了 3 个 `<script>` 标签，每个都会调用浏览器的 `console.log` JavaScript 函数：

```
1. def comet = Action {
2.   val events = Enumerator(
3.     """<script>console.log('kiki')</script>""",
4.     """<script>console.log('foo')</script>""",
5.     """<script>console.log('bar')</script>"""
6.   )
7.   Ok.chunked(events).as(HTML)
8. }
```

如果你在浏览器中运行这个 action，你会看到 3 个事件在浏览器控制台打印日志。

对于上述例子，我们可以用一种更好的方法来做，即使用 `play.api.libs.iteratee.Enumeratee` 这个适配器，它能将 `Enumerator[A]` 转成 `Enumerator[B]`。下面我们使用它将标准信息包到 `<script>` 标签里：

```
1.  import play.twirl.api.Html
2.
3.  // Transform a String message into an Html script tag
4.  val toCometMessage = Enumeratee.map[String] { data =>
5.    Html("""<script>console.log('""" + data + """')</script>""")
6.  }
7.
8.  def comet = Action {
9.    val events = Enumerator("kiki", "foo", "bar")
10.   Ok.chunked(events &> toCometMessage)
11. }
```

> 注意：*`events &> toCometMessage`* 只是 *`events.through(toCometMessage)`* 的另一种表述方式。

使用 helper：`play.api.libs.Comet`

Play 提供了相应的 helper 来处理 comet 分块流，效果与上面所写的方法基本一样。

> 注意：事实上，它做得更多。比如为了浏览器的兼容性，它提供了一个初始的空的缓冲数据。此外，它支持字符串和 *JSON* 格式的信息。你还可以通过类型类来扩展它，使它支持更多信息类型。

前面的例子可以重写如下：

```
1.  def comet = Action {
2.    val events = Enumerator("kiki", "foo", "bar")
```

```
3.   Ok.chunked(events &> Comet(callback = "console.log"))
4. }
```

iframe 流技术

写 comet socket 的标准方法是在 HTML `iframe` 中加载无限的分块 comet 响应，并指定一个回调函数来调用父级 frame：

```
1. def comet = Action {
2.   val events = Enumerator("kiki", "foo", "bar")
3.   Ok.chunked(events &> Comet(callback = "parent.cometMessage"))
4. }
```

HTML 页面如下：

```
1. <script type="text/javascript">
2.   var cometMessage = function(event) {
3.     console.log('Received event: ' + event)
4.   }
5. </script>
6.
7. <iframe src="/comet"></iframe>
```

> 注意：本章概念可参考：*http://en.wikipedia.org/wiki/Comet_(programming)*

WebSockets

WebSockets

WebSocket 是一种可以在浏览器内使用的 socket，它基于一种支持全双工通信的协议。只要服务器端和客户端之间存在一个活跃的 WebSocket 连接，它们之间就可以在任意时刻收发信息。

兼容 HTML5 的现代 web 浏览器通过 JavaScript WebSocket API 可以原生支持 WebSocket。然而，WebSocket 并不只局限于在浏览器中使用，有许多 WebSocket 客户端库可以用于服务器间通信，或者提供给原生的移动应用来使用。在这些情况下使用 WebSocket 有一个很大的优势，就是复用 Play 服务器已经使用的 TCP 端口。

处理 WebSocket

到目前为止，我们都是用 `Action` 实例来处理标准的 HTTP 请求，然后返回标准的 HTTP 响应。WebSocket 则完全不同，它无法通过标准的 `Action` 来处理。

Play 提供了两种不同的内建机制来处理 WebSocket。第一种使用 actor，第二种使用 iteratee。这两种机制都可以通过 Play 为 WebSocket 提供的构建器来使用。

用 actor 处理 WebSocket

想用 actor 来处理 WebSocket，我们需要一个 `akka.actor.Props` 对象来描述 actor，当 WebSocket 连接建立起来后，Play 应该创建这个 actor。Play 会提供一个 `akka.actor.ActorRef` 用于向它发送消息，因此我们可以用它（即下面的 out）来创建 `Props` 对象：

```
1. import play.api.mvc._
2. import play.api.Play.current
3.
4. def socket = WebSocket.acceptWithActor[String, String] { request =>
   out =>
5.   MyWebSocketActor.props(out)
6. }
```

在这个例子中，我们向它发送消息的 actor 定义如下：

```
1. import akka.actor._
2.
3. object MyWebSocketActor {
4.   def props(out: ActorRef) = Props(new MyWebSocketActor(out))
5. }
6.
7. class MyWebSocketActor(out: ActorRef) extends Actor {
8.   def receive = {
9.     case msg: String =>
10.      out ! ("I received your message: " + msg)
11.  }
12. }
```

任何从客户端收到的信息都会被发送到 actor（即 out），任何发送给 Play 提供的 actor 的信息都会被发送到客户端。上面的 actor 就简单地将它收到的信息在前面加上 `I reveived your message:`，然后返回给客户端。

检测 WebSocket 何时关闭

如果 WebSocket 已经关闭，Play 会自动停掉 actor。这意味着你可以通过实现 `postStop` 方法来清理 WebSocket 可能使用的资源。例如：

```
override def postStop() = {
  someResource.close()
}
```

关闭 WebSocket

当处理 WebSocket 的 actor 终止时，Play 会自动关闭 WebSocket。因此，你可以通过发送一个 `PoisonPill` 给你自己的 actor 来关闭 WebSocket：

```
import akka.actor.PoisonPill

self ! PoisonPill
```

拒绝 WebSocket 请求

有时候你可能想拒绝一个 WebSocket 请求，例如，必须是授权用户才能连接 WebSocket，或者 WebSocket 所关联到的资源不存在。Play 提供了 `tryAcceptWithActor` 来处理这种情况，允许你返回一个结果（如 forbidden 或 not found），或是返回一个处理

WebSocket 的 actor:

```
1.  import scala.concurrent.Future
2.  import play.api.mvc._
3.  import play.api.Play.current
4.
5.  def socket = WebSocket.tryAcceptWithActor[String, String] { request
    =>
6.    Future.successful(request.session.get("user") match {
7.      case None => Left(Forbidden)
8.      case Some(_) => Right(MyWebSocketActor.props)
9.    })
10. }
```

处理不同类型的信息

到目前为止，我们看到的都是在处理 `String` 信息。其实，Play 也提供了内建的方式来处理 `Array[Byte]` 和 `JsValue` 信息。你可以把这些作为类型参数传递给 WebSocket 构建方法，例如:

```
1.  import play.api.mvc._
2.  import play.api.libs.json._
3.  import play.api.Play.current
4.
5.  def socket = WebSocket.acceptWithActor[JsValue, JsValue] { request
    => out =>
6.    MyWebSocketActor.props(out)
7.  }
```

你可以已经注意到了，上面有两个类型参数（都是 JsValue），这允许我们将接收进来的信息与发送出去的信息定义为不同的类型。这通常对于低级类型来说是没什么用的，但当你想把信息转换成高级类型时，它就非常有用了。

例如，我们想接收 JSON 类型的信息，然后把它解析成 `InEvent` 类

型，再把返回的信息转成 `OutEvent` 类型。我们要做的第一件事是为 `InEvent` 和 `OutEvent` 创建 JSON 格式（用于隐式转换）：

```
1. import play.api.libs.json._
2.
3. implicit val inEventFormat = Json.format[InEvent]
4. implicit val outEventFormat = Json.format[OutEvent]
```

接着，我们为这两个类型创建 WebSocket `FrameFormatter`：

```
1. import play.api.mvc.WebSocket.FrameFormatter
2.
3. implicit val inEventFrameFormatter =
   FrameFormatter.jsonFrame[InEvent]
4. implicit val outEventFrameFormatter =
   FrameFormatter.jsonFrame[OutEvent]
```

最终，我们可以在 WebSocket 中使用这两个类型：

```
1. import play.api.mvc._
2. import play.api.Play.current
3.
4. def socket = WebSocket.acceptWithActor[InEvent, OutEvent] { request
   => out =>
5.   MyWebSocketActor.props(out)
6. }
```

现在，在我们的 actor 中，我们会接收到类型为 `InEvent` 的信息，而发送出去的信息类型为 `OutEvent`。

用 iteratee 处理 WebSocket

actor 是一种更好的抽象来处理离散信息，而 iteratee 是一种更好的抽象来处理流。

处理 WebSocket 请求，使用 `WebSocket` 而不是 `Action`：

```
import play.api.mvc._
import play.api.libs.iteratee._
import play.api.libs.concurrent.Execution.Implicits.defaultContext

def socket = WebSocket.using[String] { request =>

  // Log events to the console
  val in = Iteratee.foreach[String](println).map { _ =>
    println("Disconnected")
  }

  // Send a single 'Hello!' message
  val out = Enumerator("Hello!")

  (in, out)
}
```

`WebSocket` 可以访问初始化 WebSocket 连接的 HTTP 请求的报头，允许你获取标准报头和会话数据。然而，它无权访问请求体或是 HTTP 响应。

当你通过这种方式来构建 WebSocket 时，你必须同时返回 `in` 和 `out` 两个通道。

- `in` 通道的类型是 `Iteratee[A,Unit]`（其中 `A` 是消息类型，这里我们用的是 `String`），对于每条传来的消息，它都会被通知到。当客户端的 socket 关闭后，它会收到 `EOF`。
- `out` 通道的类型是 `Enumerator[A]`，它会产生发送给 Web 客户端的消息。它也可以通过发送 `EOF` 在服务器端关闭连接。

在这个例子中，我们创建了一个简单的 iteratee，它会在控制台打印每条消息。同时，我们创建了一个简单的枚举器（enumerator）来

发送一个 `Hello!` 消息。

> 小贴士：你可以在这里测试 *WebSocket*，只需要把 *location* 设置为 `ws://localhost:9000` 即可。

下面的例子直接忽略输入数据，在发送完 **Hello!** 消息后就关闭 socket：

```
1.  import play.api.mvc._
2.  import play.api.libs.iteratee._
3.
4.  def socket = WebSocket.using[String] { request =>
5.
6.    // Just ignore the input
7.    val in = Iteratee.ignore[String]
8.
9.    // Send a single 'Hello!' message and close
10.   val out = Enumerator("Hello!").andThen(Enumerator.eof)
11.
12.   (in, out)
13. }
```

下面是另外一个例子，其中输入数据被打印到标准输出，然后利用 `Concurrent.broadcast` 广播到客户端：

```
1.  import play.api.mvc._
2.  import play.api.libs.iteratee._
3.  import play.api.libs.concurrent.Execution.Implicits.defaultContext
4.
5.  def socket =  WebSocket.using[String] { request =>
6.
7.    // Concurrent.broadcast returns (Enumerator, Concurrent.Channel)
8.    val (out, channel) = Concurrent.broadcast[String]
9.
10.   // log the message to stdout and send response back to client
11.   val in = Iteratee.foreach[String] {
12.     msg => println(msg)
```

```
13.       // the Enumerator returned by Concurrent.broadcast subscribes
    to the channel and will
14.       // receive the pushed messages
15.       channel push("I received your message: " + msg)
16.     }
17.     (in,out)
18.   }
```

The template engine

Templates syntax

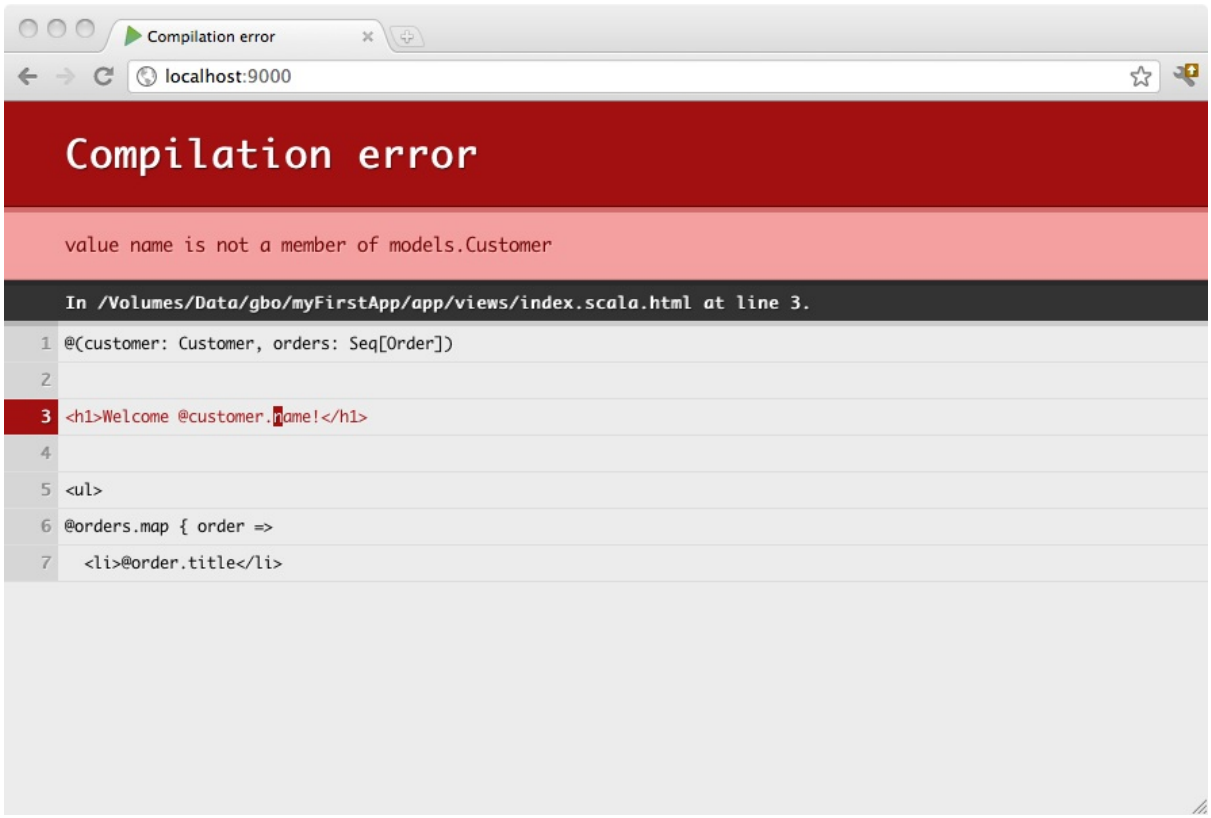
基于 Scala 的类型安全的模板引擎

Play 自带了一个基于 Scala 的模板引擎，叫 Twirl，它的设计是受到了 ASP.NET Razor 的启发。它：

- 简洁、富于表现力且灵活：它最小化了一个文件中的字符和按键数，使你的工作流快速、连续。与大部分模板语法不同，你无需在 HTML 内显式地标注服务端的代码块，解析器能从你的代码中自动推断出来。这使得模板语法简洁紧凑而富有表现力，写起来干净、快速而且好玩。
- 容易学习：只需要学习少量概念，即可使你高效多产。你只需要使用简单的 Scala 概念及你已有的 HTML 技巧即可。

- 不是一门新语言：我们有意识地选择不去创造一门新语言，而是想让 Scala 开发者能使用他们现有的 Scala 语言技巧，并提供一种使 HTML 构建过程更赞的模板标记语法。
- 可在任意文本编辑器中编辑：它不需要特殊的开发工具，你可以在任意的纯文本编辑器中高效使用它。

模板会被编译，因此如果有错误，你可以在浏览器中直接看到：



概述

Play 的 Scala 模板是一个包含小段 Scala 代码块的文本文件。模板可以生成任意基于文本的格式，如 HTML，XML 或 CSV。

模板系统的设计使 HTML 使用者用起来会很舒服，前端开发者很容易就可以上手。

模板会被编译成标准 Scala 函数（使用一种简单的命名约定）。如果

你创建一个 `views/Application/index.scala.html` 模板文件，它会生成一个 `views.html.Application.index` 类，这个类有一个 `apply()` 方法。

例如，下面是一个简单的模板：

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.title</li>
}
</ul>
```

你可以在任意 Scala 代码中调用上述代码，就像你调用一个类的方法一样：

```
val content = views.html.Application.index(c, o)
```

神奇的 ‘@’ 字符

Scala 模板只使用一个特殊字符：@。每次遇到这个字符，它就表明一条动态语句的开始。你无需显式地标注代码块的结束，动态语句在哪结束会被自动地推断出来：

```
Hello @customer.name!
       ^^^^^^^^^^^^^
       Dynamic code
```

模板引擎是通过分析你的代码来推断代码块在哪结束，因此它只能支持简单的语句。如果你要插入一条复杂语句，用括号来显式地标注它：

```
Hello @(customer.firstName + customer.lastName)!
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    Dynamic Code
```

如果你想插入一个包含多条语句的块，可以使用大括号：

```
Hello @{val name = customer.firstName + customer.lastName; name}!
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                           Dynamic Code
```

由于 `@` 是特殊字符，当你想使用 `@` 这个字符时，就需要转义它（即 `@@` ）：

```
My email is bob@@example.com
```

模板参数

模板就像函数，因此它也需要参数。参数必须被声明在模板文件的顶部：

```
@(customer: Customer, orders: List[Order])
```

你也可以为参数提供默认值：

```
@(title: String = "Home")
```

甚至可以提供几个参数组：

```
@(title: String)(body: Html)
```

遍历

你可以使用关键词 `for` 进行遍历：

```
<ul>
@for(p <- products) {
  <li>@p.name ($@p.price)</li>
}
</ul>
```

> 注意：确保 `{` 与 `for` 在同一行，用于指示表达式还没结束，会延续到下一行。

if 块

if 块的使用并没有什么特别的，和 Scala 中的 `if` 用法相似：

```
@if(items.isEmpty) {
  <h1>Nothing to display</h1>
} else {
  <h1>@items.size items!</h1>
}
```

声明可复用的代码块

你可以创建可复用的代码块：

```
@display(product: Product) = {
  @product.name ($@product.price)
}

<ul>
@for(product <- products) {
  @display(product)
}
</ul>
```

注意，你也可以声明可复用的纯代码块：

```
@title(text: String) = @{
```

```
2.   text.split(' ').map(_.capitalize).mkString(" ")
3. }
4.
5. <h1>@title("hello world")</h1>
```

> 注意：在模板里声明代码块有时候是有用的，但请记住，模板里不宜放复杂的逻辑。这部分代码应该放在外部的 *Scala* 类中（如果你愿意，你也可以把它放在 *views/* 包下）。

按照惯例，一个可复用的代码块命名如果以 **implicit** 开头，它就会被标记为 `implicit`：

```
1. @implicitFieldConstructor = @{ MyFieldConstructor() }
```

声明可复用的值

你可以使用 `defining` helper 方法来定义一个局部值：

```
1. @defining(user.firstName + " " + user.lastName) { fullName =>
2.   <div>Hello @fullName</div>
3. }
```

导入声明

你可以在模板（或子模板）的开始处导入任何你想要的东西：

```
1. @(customer: Customer, orders: List[Order])
2.
3. @import utils._
4.
5. ...
```

为确保绝对的解析路径，可以在 import 语句前加上 **root** 前缀：

```
1. @import _root_.company.product.core._
```

如果在所有的模板中，你都要导入一个相同的东西，那么你可以将它声明在 `build.sbt`：

```
TwirlKeys.templateImports += "org.abc.backend._"
```

注释

使用 `@* *@`，你就可以在模板中像服务端那样写块注释：

```
@*********************
* This is a comment *
*********************@
```

你可以在模板最开始处写上注释，这样可以把你的模板文档化到 Scala API doc 中：

```
@************************************
 * Home page.                       *
 *                                  *
 * @param msg The message to display *
 ************************************@
@(msg: String)

<h1>@msg</h1>
```

转义

默认情况下，动态内容会根据模板的类型（如 HTML 或 XML）规则进行转义。如果你想输出原始内容，则需要将它包在模板的内容类型里。

例如，要输出原始的 HTML 内容：

```
<p>
  @Html(article.content)
```

```
3. </p>
```

Common use cases

Common use cases

模板作为简单的函数，可以由任意的方式构成。下面是一些常见的使用情况。

页面布局（layout）

让我们来声明一个 `views/main.scala.html` 模板来作为主要的布局模板：

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
  </body>
</html>
```

正如你所见，这个模板接收两个参数：标题（`title`）和 HTML 内容块（`content`）。下面我们在另一个模板（ `views/Application/index.scala.html` ）中使用它：

```
@main(title = "Home") {

  <h1>Home page</h1>

}
```

> 注意：有时候我们使用命名参数，如 `@main(title = "Home")` ，而不是 `@main("Home")` 。这个视具体情况，选择一个表述清楚的即可。

有时候，你需要另一个页面内容来作为侧边栏（sidebar），这时你可以添加一个额外的参数来做到：

```
@(title: String)(sidebar: Html)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="sidebar">@sidebar</section>
    <section class="content">@content</section>
  </body>
</html>
```

在我们的 `index` 模板中使用：

```
@main("Home") {
  <h1>Sidebar</h1>

} {
  <h1>Home page</h1>

}
```

或者，我们可以单独声明侧边栏：

```
@sidebar = {
```

```
  <h1>Sidebar</h1>
}

@main("Home")(sidebar) {
  <h1>Home page</h1>

}
```

标签（它们也是函数）

下面我们来写一个简单的标签 `views/tags/notice.scala.html`，用于显示 HTML 通知：

```
@(level: String = "error")(body: (String) => Html)

@level match {

  case "success" => {
    <p class="success">
      @body("green")
    </p>
  }

  case "warning" => {
    <p class="warning">
      @body("orange")
    </p>
  }

  case "error" => {
    <p class="error">
      @body("red")
    </p>
  }

}
```

现在，我们在另一个模板中使用它：

```
@import tags._

@notice("error") { color =>
  Oops, something is <span style="color:@color">wrong</span>
}
```

Includes

同样没有任何特别之处，你可以调用任意其它模板（事实上可以调用任意地方的任意函数）：

```
<h1>Home</h1>

<div id="side">
  @common.sideBar()
</div>
```

moreScripts 与 moreStyles 等价物

想在 Scala 模板中定义 moreScripts 和 moreStyles 等价物（像在 Play! 1.x 那样），你只需像下面那样在主模板中定义一个变量：

```
@(title: String, scripts: Html = Html(""))(content: Html)

<!DOCTYPE html>

<html>
    <head>
        <title>@title</title>
        <link rel="stylesheet" media="screen"
href="@routes.Assets.at("stylesheets/main.css")">
```

```
        <link rel="shortcut icon" type="image/png"
    href="@routes.Assets.at("images/favicon.png")">
        <script src="@routes.Assets.at("javascripts/jquery-
    1.7.1.min.js")" type="text/javascript"></script>
        @scripts
    </head>
    <body>
        <div class="navbar navbar-fixed-top">
            <div class="navbar-inner">
                <div class="container">
                    <a class="brand" href="#">Movies</a>
                </div>
            </div>
        </div>
        <div class="container">
            @content
        </div>
    </body>
</html>
```

对于一个需要额外脚本的扩展模板，用法如下：

```
@scripts = {
    <script type="text/javascript">alert("hello !");</script>
}

@main("Title",scripts){

   Html content here ...

}
```

如果扩展模板不需要额外脚本，则：

```
@main("Title"){

   Html content here ...
```

```
4.
5. }
```

Custom format addition

Custom format addition

内建的模板引擎支持常见的模板格式（HTML，XML 等），但如果有需要，你可以非常容易地增加对自定义格式的支持。下面总结了添加自定义格式支持的步骤。

模板化概述

模板引擎通过将模板中的静态和动态内容组合到一起来构建最终结果。考虑如下模板：

```
1.  foo @bar baz
```

它由两个静态部分（`foo` 和 `baz`）和一个动态部分（`bar`）组成。模板引擎将这些拼接起来构建最终结果。事实上，为了避免跨站点脚本攻击，`bar` 的值在拼接到其它结果之前可以先进行转义。转义过程与具体的格式相关。例如，对于 HTML 格式，"<" 转义的结果是 "<"。

模板引擎是如何知道一个模板的格式的呢？它会检查这个文件的扩展名，例如该文件的扩展名是 `.scala.html`，模板引擎就会把该文件与 HTML 格式关联起来。

最后，你的模板文件需要作为 HTTP 响应的主体，因此你还需要定义如何用模板渲染的结果来构建 Play 返回给客户端的结果。

综上所述，如果想让 Play 支持你自己定义的模板格式，需要做以下几件事：

- 为该格式实现文本整合过程
- 将你定义的格式与某一文件扩展名关联起来
- 最后，你需要告诉 Play 如何把模板渲染的结果作为 HTTP 响应体发送出去

实现一个模板格式

实现 `play.twirl.api.Format[A]` trait，该 trait 包含两个方法：`raw(text: String): A` 和 `escape(text: String): A`，分别用于整合模板的静态和动态部分。

类型参数 `A` 定义了模板渲染后的结果类型，例如，对于 HTML 模板，该类型是 `HTML`。该类型必须是 `play.twirl.api.Appendable[A]` trait 的子类，该 trait 定义了如何把各部分拼接在一起。

方便起见，Play 提供了 `play.twirl.api.BufferedContent[A]` 抽象类，它实现了 `play.twirl.api.Appendable[A]` trait，使用 `StringBuilder` 来构建它的结果。它还实现了 `play.twirl.api.Content` trait，因此 Play 知道如何将它作为 HTTP 响应体进行序列化（详情参见本节最后一部分）。

简而言之，你需要写两个类：一个定义结果（实现 `play.twirl.api.Appendable[A]`），另一个定义文本整合过程（实现 `play.twirl.api.Format[A]`）。例如，下面是对 HTML 格式的定义：

```
1.  // The `Html` result type. We extend `BufferedContent[Html]` rather
```

```
    than just `Appendable[Html]` so
2.  // Play knows how to make an HTTP result from a `Html` value
3.  class Html(buffer: StringBuilder) extends BufferedContent[Html]
    (buffer) {
4.    val contentType = MimeTypes.HTML
5.  }
6.
7.  object HtmlFormat extends Format[Html] {
8.    def raw(text: String): Html = …
9.    def escape(text: String): Html = …
10. }
```

将该格式与某一文件扩展名关联起来

在编译整个项目源码之前，模板文件会在构建过程中被编译成 `.scala` 文件。`TwirlKeys.templateFormats` 是一个类型为 `Map[String, String]` 的 sbt 配置项，定义了文件扩展名与模板格式之间的映射。例如，如果 Play 不支持 HTML 格式，你就需要在构建文件中写入下面的配置来关联 `.scala.html` 文件和 `play.twirl.api.HtmlFormat` 格式：

```
1.  TwirlKeys.templateFormats += ("html" -> "my.HtmlFormat.instance")
```

注意，箭头右边包含了一个类型为 `play.twirl.api.Format[_]` 的值的全限定名。

用模板渲染结果构建 HTTP 结果

类型 `A` 的值如果可以隐式转换为类型 `play.api.http.Writeable[A]` 的值，则 Play 可以将 `A` 的任意值写入 HTTP 响应体中。因此你所需要做的就是为你模板的结果类型 `A` 定义一个隐式转换的值。例如，以下展示了如何为 HTTP 定义一个这样的值：

```
implicit def writableHttp(implicit codec: Codec): Writeable[Http] =
  Writeable[Http](result => codec.encode(result.body),
Some(ContentTypes.HTTP))
```

> 注意：如果你的模板结果类型扩展了 `play.twirl.api.BufferedContent`，你就只需要定义一个隐式的 `play.api.http.ContentTypeOf` 值：
> `implicit def contentTypeHttp(implicit codec: Codec): ContentTypeOf[Http] = ContentTypeOf[Http](Some(ContentTypes.HTTP))`

Form submission and validation

Handling form submission

概述

表单的处理和提交是每个 web 应用的重要部分。Play 自带了一些功能，简化了简易表单的处理，并使处理复杂表单成为可能。

Play 的表单处理方式基于数据绑定的概念。当数据来自于一个 POST 请求时，Play 会查找格式化的值并将其绑定到 `Form` 对象上。Play 可以根据绑定的表单及数据构建一个样本类（case class），调用自定义的验证器，等等。

通常，表单会在 `Controller` 实例中直接使用。不过 `Form` 的定义并不需要和样本类（case class）或是模型一致匹配：他们纯粹都是为了处理输入，不同的 POST 使用不同的 `Form` 非常合理。

导入

为了使用表单，必须在类中先导入以下几个包：

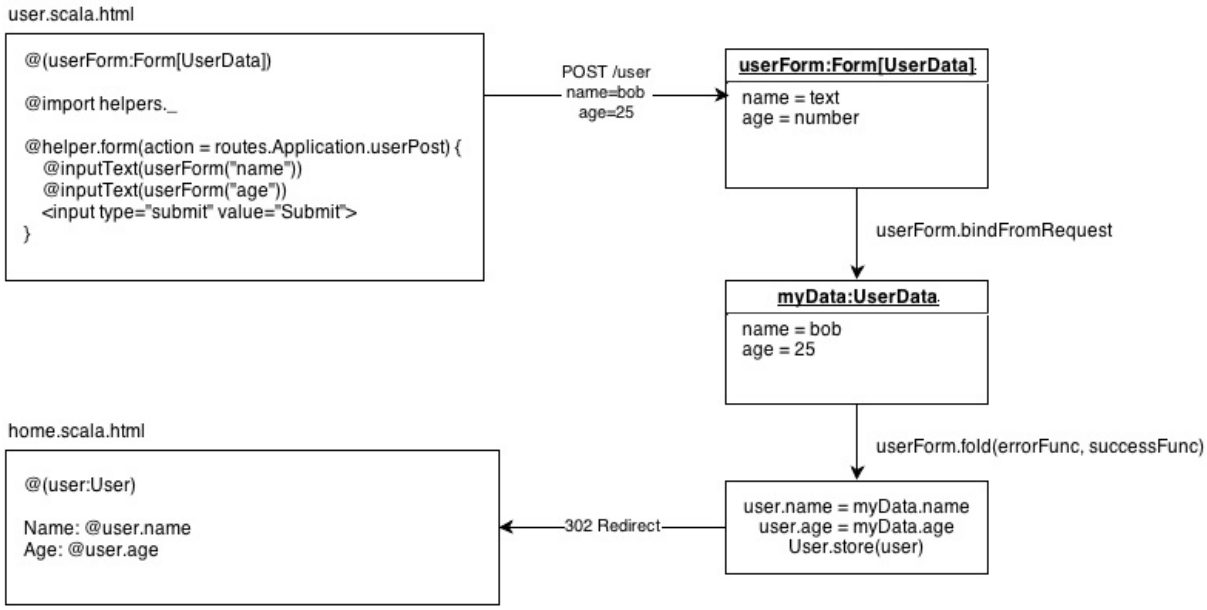
```
1.  import play.api.data._
2.  import play.api.data.Forms._
```

表单基础

让我们先来看一下基本的表单处理：

- 定义一个表单，
- 定义表单中的约束，
- 在一个 action 中验证该表单，
- 在视图模板中显示表单，
- 最后，在视图模板中处理表单的结果（或是错误）

最终的结果看起来是这样的：



定义一个表单

首先，定义一个包含了表单中所有元素的样本类（case class）。这里我们想要获取一个用户的名字和年龄，因此我们创建了一个 `UserData` 对象：

```
1.  case class UserData(name: String, age: Int)
```

现在我们有了一个样本类（case class），接着需要定义一个 `Form` 结构。定义表单的方法是将表单的数据绑定到样本类（case class）的实例中，定义如下：

```
1.  val userForm = Form(
2.    mapping(
3.      "name" -> text,
4.      "age" -> number
5.    )(UserData.apply)(UserData.unapply)
6.  )
```

表单对象定义了 `mapping` 方法。这个方法接收表单的名字和约束作为参数，并接收另外两个方法：一个 `apply` 方法和一个 `unapply`

方法。因为 `UserData` 是一个样本类（case class），我们可以将 `apply` 和 `unapply` 直接插入到 `mapping` 方法中。

注意：由于表单处理的实现问题，一个单一元组（tuple）或是 mapping 最多只能有18个元素。如果你的表单有超过18个元素，你需要将他们通过列表或是嵌套值分开来。

当你使用了 Map ，表单会创建一个带有绑定值的 UserData 实例：

```
1.  val anyData = Map("name" -> "bob", "age" -> "21")
2.  val userData = userForm.bind(anyData).get
```

但大多数的时候会在 Action 中使用表单，其中的数据由请求提供。`Form` 包含了 `bindFormRequest`，接收一个请求作为隐式（implicit）参数。如果你定义了一个隐式（implicit）请求，那么 `bindFormRequest` 就能够找到他。

```
1.  val userData = userForm.bindFromRequest.get
```

注意：这里使用 `get` 其实是有问题的。如果表单没能绑定数据，那么 `get` 会抛出异常。在接下来的几章中我们会介绍一种更安全的方法来处理输入。

Play 并没有限制说只能用样本类（case class） 来映射你的表单。只要正确设置了 `apply` 和 `unapply` 方法，就能传入你想要的任何东西，比如元组可以使用 `Forms.tuple` 来映射，或是模型样本类（model case class）。但是，为表单指定一个样本类（case class）有以下这些好处：

- 表单指定的样本类（**case class**）简单易用。样本类（case class）原本便被设计为存储数据的简易容器，即写即用，和表单功能天然匹配。

- 表单指定的样本类（**case class**）功能强大。元组易于使用，但元组并不允许你自定义 `apply` 或是 `unapply` 方法，且只能通过元数（arity，如 _1, _2 等）来引用其中的数据。
- 表单指定的样本类（**case class**）专为表单设计。重用模型样本类（model case class）确实很方便，但模型通常都含有一些额外的领域逻辑，甚至于一些数据持久化的细节，这些都会导致耦合过于紧密。另外，如果表单和模型并不是1：1严格映射的话，敏感数据必须被显示的忽略，以此来抵御篡改参数攻击。

定义表单的约束

`text` 约束认定空字符串依然有效。也就是说 `name` 可以为空且不会报错，这显然不是我们想要的。一个保证 `name` 有值的方法是使用 `nonEmptyText` 约束。

```
1.  val userFormConstraints2 = Form(
2.    mapping(
3.      "name" -> nonEmptyText,
4.      "age" -> number(min = 0, max = 100)
5.    )(UserData.apply)(UserData.unapply)
6.  )
```

如果表单的输入没有满足该表单的约束条件，则会报错：

```
1.  val boundForm = userFormConstraints2.bind(Map("bob" -> "", "age" ->
    "25"))
2.  boundForm.hasErrors must beTrue
```

预置的约束定义在了表单对象中：

- `text`：对应于 `scala.String`，可选参数为 `minLength` 和 `maxLength`。

- `nonEmptyText`：对应于 `scala.String`，可选参数为 `minLength` 和 `maxLength`。
- `number`：对应于 `scala.Int`，可选参数为 `min`，`max` 和 `strict`。
- `longNumber`：对应于 `scala.Long`，可选参数为 `min`，`max`，和 `strict`。
- `bigDecimal`：参数为 `precision` 和 `scale`。
- `date`，`sqlDate`，`jodaDate`：对应于 `java.util.Date`，`java.sql.Date` 和 `org.joda.time.DateTime`，可选参数为 `pattern` 和 `timeZone`。
- `jodaLocalDate`：对应于 `org.joda.time.LocalDate`，可选参数为 `pattern`。
- `email`：对应于 `scala.String`，使用邮件正则表达式.
- `boolean`：对应于 `scala.Boolean`。
- `checked`：对应于 `scala.Boolean`。
- `optional`：对应于 `scala.Option`。

定义特殊约束

你可以使用验证包在样本类（`case class`）中定义你自己的特殊约束。

```
1.  val userFormConstraints = Form(
2.    mapping(
3.      "name" -> text.verifying(nonEmpty),
4.      "age" -> number.verifying(min(0), max(100))
5.    )(UserData.apply)(UserData.unapply)
6.  )
```

你也可以直接在样本类（`case class`）中定义特殊约束:

```
def validate(name: String, age: Int) = {
  name match {
    case "bob" if age >= 18 =>
      Some(UserData(name, age))
    case "admin" =>
      Some(UserData(name, age))
    case _ =>
      None
  }
}

val userFormConstraintsAdHoc = Form(
  mapping(
    "name" -> text,
    "age" -> number
  )(UserData.apply)(UserData.unapply) verifying("Failed form constraints!", fields => fields match {
    case userData => validate(userData.name, userData.age).isDefined
  })
)
```

当然，你还能定义你自己的验证器。详情请见自定义验证器。

在 Action 中验证表单

现在已经有了约束，我们需要在 action 中验证表单，并处理表单错误。

我们使用 `fold` 方法来实现，它接收了两个函数：绑定失败时第一个会被调用，绑定成功则会调用第二个。

```
userForm.bindFromRequest.fold(
  formWithErrors => {
    // binding failure, you retrieve the form containing errors:
    BadRequest(views.html.user(formWithErrors))
```

```
5.    },
6.    userData => {
7.      /* binding success, you get the actual value. */
8.      val newUser = models.User(userData.name, userData.age)
9.      val id = models.User.create(newUser)
10.     Redirect(routes.Application.home(id))
11.   }
12. )
```

绑定失败时，我们用了 BadRequest 来渲染页面，并将错误作为参数传入该页。如果使用了视图 helper （在下面讨论），那么任何绑定于一个元素的错误都会被渲染在该元素边上。

绑定成功时，我们发出了一个 `Redirect` ，路由到 `routes.Application.home` ，而不是渲染一个视图模板。这种模式称为 POST 后重定向，这是一种很好的防止重复提交的方式。

注意：当使用了 `flashing` 或是在其他方法中用到了 flash 域，使用 “POST 后重定向” 是必需的，因为新的 cookie 只能在重定向 HTTP 请求后获取。

在视图模板中显示表单

有了表单之后，我们就能在模板引擎中调用它。做法是在视图模板中将表单作为参数引入。对于 `user.scala.html` 来说，该页的开头是这样的：

```
1. @(userForm: Form[UserData])
```

由于 `user.scala.html` 需要接收一个表单，在渲染 `user.scala.html` 时应先传入一个空的 `userForm` 。

```
1. def index = Action {
2.   Ok(views.html.user(userForm))
```

```
3. }
```

首先是要创建表单标签。这是一个简单的视图 `helper`， 创建了一个表单标签，并根据你所传入的逆向路由设置了 `action` 和 `method` 的标签参数。

```
1. @helper.form(action = routes.Application.userPost()) {
2.   @helper.inputText(userForm("name"))
3.   @helper.inputText(userForm("age"))
4. }
```

你可以在 `views.html.helper` 包中找到多个输入 helper。传入一个表单域，他们就能显示出相应的 HTML 输入，设置值和约束，并在表单绑定失败时报错。

注意：你可以在模板中使用 `@import heler._` 来避免在调用 helper 时前置 `@helper` 。

Play 有多个输入 helper，其中最有用的是：

- `form`： 渲染表单。
- `inputText`： 渲染文本输入。
- `inputPassword`： 渲染密码输入。
- `inputDate`： 渲染日期输入。
- `inputFile`： 渲染文件输入。
- `inputRadioGroup`： 渲染单选输入。
- `select`： 渲染下拉列表。
- `textarea`： 渲染文本域。
- `checkbox`： 渲染复选框。
- `input`： 渲染通用输入（需要显示声明参数）。

在 `form` helper 中，你可以通过指定额外的参数，将其添加到生

成的 HTML中：

```
@helper.inputText(userForm("name"), 'id -> "name", 'size -> 30)
```

上面这个通用的 `input` helper 能让你编写你想要的 HTML 代码：

```
@helper.input(userForm("name")) { (id, name, value, args) =>
    <input type="text" name="@name" id="@id" @toHtmlArgs(args)>
}
```

注意：所有额外的参数都会被添加到生成的 HTML 中，除非他们以 `_` 字符开头。以 `_` 开头的参数是域构造器参数。

对于复杂表单元素，你同样可以自定义视图 helper （在 `views` 页面中使用 scala 类）并自定义域构造器。

在视图模板中显示错误

表单中的错误类型为 `Map[String, FormError]`，其中 `FormError` 中有：

- `key`：应和表单域相同。
- `message`：一段错误提示信息或是对应于该信息的键。
- `args`：错误提示信息中用到的一组参数。

绑定的表单实例中可以获得如下错误：

- `errors`：返回所有错误，类型为 `Seq[FormError]`。
- `globalErrors`：返回所有错误，类型为没有键的 `Seq[FormError]`。
- `error("name")`：返回第一个绑定了该键的错误，类型为 `Option[FormError]`。

- `errors("name")`：返回所有绑定了该键的错误，类型为 `Option[FormError]`。

使用表单 helper 可以自动渲染绑定于某表单域的错误，如 `@helper.inputText` 的错误显示如下：

```
<dl class="error" id="age_field">
    <dt><label for="age">Age:</label></dt>
    <dd><input type="text" name="age" id="age" value=""></dd>
    <dd class="error">This field is required!</dd>
    <dd class="error">Another error</dd>
    <dd class="info">Required</dd>
    <dd class="info">Another constraint</dd>
</dl>
```

没有绑定于任一键的全局错误（Global errors）不会有相应的 helper，且必须显示的定义在页面中：

```
@if(userForm.hasGlobalErrors) {
  <ul>
  @for(error <- userForm.globalErrors) {
    <li>@error.message</li>
  }
  </ul>
}
```

使用元组做映射

你可以在表单域中使用元组，而非样本类（case class）：

```
val userFormTuple = Form(
  tuple(
    "name" -> text,
    "age" -> number
  ) // tuples come with built-in apply/unapply
```

```
6.  )
```

有时候使用元组会比定义样本类（case class）更方便，尤其在元组的元数（arity）很小时：

```
1.  val anyData = Map("name" -> "bob", "age" -> "25")
2.  val (name, age) = userFormTuple.bind(anyData).get
```

使用 single 做映射

元组只有在有多个值的时候才有用。如果只有一个表单域，可以用 `Forms.single` 来映射单一值，避免了使用样本类（case class）或是元组的额外开销：

```
1.  val singleForm = Form(
2.    single(
3.      "email" -> email
4.    )
5.  )
```

```
1.  val email = singleForm.bind(Map("email", "bob@example.com")).get
```

填值

有时候，你想要在表单中预置一些值，通常用于编辑数据：

```
1.  val filledForm = userForm.fill(UserData("Bob", 18))
```

当配合视图 helper 使用时，该元素会填上预置的值：

```
1.  @helper.inputText(filledForm("name")) @* will render value="Bob" *@
```

填值在 helper 需要一列值或是键值对时尤其有用，比如 `select`

和 `inputRadioGroup` helper 。使用 `options` 可以为 helper 填入列表，键值对和对值（pair）。

嵌套值

表单映射同样可以在已有的映射中使用 `Forms.mapping` 来实现嵌套值：

```
case class AddressData(street: String, city: String)

case class UserAddressData(name: String, address: AddressData)
```

```
val userFormNested: Form[UserAddressData] = Form(
  mapping(
    "name" -> text,
    "address" -> mapping(
      "street" -> text,
      "city" -> text
    )(AddressData.apply)(AddressData.unapply)
  )(UserAddressData.apply)(UserAddressData.unapply)
)
```

注意：当你使用这种方法来嵌套数据时，浏览器传来的表单值必须命名为 `address.street` ， `address.city` …

```
@helper.inputText(userFormNested("name"))
@helper.inputText(userFormNested("address.street"))
@helper.inputText(userFormNested("address.city"))
```

重复值

表单映射可以使用 `Forms.list` 或 `Forms.seq` 来定义重复值：

```
case class UserListData(name: String, emails: List[String])
```

```
val userFormRepeated = Form(
  mapping(
    "name" -> text,
    "emails" -> list(email)
  )(UserListData.apply)(UserListData.unapply)
)
```

当你这么使用重复数据时，浏览器传入的表单值必须被命名为 `emails[0]` ， `emails[1]` ， `emails[2]` …

使用 `repeat` helper 来生成和表单 `emails` 域相同数目的输入：

```
@helper.inputText(myForm("name"))
@helper.repeat(myForm("emails"), min = 1) { emailField =>
    @helper.inputText(emailField)
}
```

`min` 参数允许你在表单数据为空时，定义最少显示多少个表单域。

可选值

使用 `Forms.optional` 来定义可选值：

```
case class UserOptionalData(name: String, email: Option[String])
```

```
val userFormOptional = Form(
  mapping(
    "name" -> text,
    "email" -> optional(email)
  )(UserOptionalData.apply)(UserOptionalData.unapply)
)
```

这样做会输出一个 `Option[A]` ，在没有找到任何表单值的情况下则返

回 `None`。

默认值

你可以使用 `Form#fill` 来初始化值:

```
1.  val filledForm = userForm.fill(User("Bob", 18))
```

你也可以通过 `Forms.default` 来定义默认值:

```
1.  Form(
2.    mapping(
3.      "name" -> default(text, "Bob")
4.      "age" -> default(number, 18)
5.    )(User.apply)(User.unapply)
6.  )
```

忽略值

如果你想定义某个表单域为一个静态值，使用 `Forms.ignored`:

```
1.  val userFormStatic = Form(
2.    mapping(
3.      "id" -> ignored(23L),
4.      "name" -> text,
5.      "email" -> optional(email)
6.    )(UserStaticData.apply)(UserStaticData.unapply)
7.  )
```

合并起来

这个例子演示了使用模型和控制器来处理一个实体:

样本类 `Contact`:

```
case class Contact(firstname: String,
                   lastname: String,
                   company: Option[String],
                   informations: Seq[ContactInformation])

object Contact {
  def save(contact: Contact): Int = 99
}

case class ContactInformation(label: String,
                              email: Option[String],
                              phones: List[String])
```

需要注意的是，`Contact` 里有一个包含了 `ContactInformation` 元素的 `Seq` 和一个含有 `String` 的 `List`。在这个例子中，我们组合了嵌套映射和重复映射（分别由 `Forms.seq` 和 `Forms.list` 定义）。

```
val contactForm: Form[Contact] = Form(

  // Defines a mapping that will handle Contact values
  mapping(
    "firstname" -> nonEmptyText,
    "lastname" -> nonEmptyText,
    "company" -> optional(text),

    // Defines a repeated mapping
    "informations" -> seq(
      mapping(
        "label" -> nonEmptyText,
        "email" -> optional(email),
        "phones" -> list(
          text verifying pattern("""[0-9.+]+""".r, error="A valid phone number is required")
        )
      )(ContactInformation.apply)(ContactInformation.unapply)
    )
```

```
19.   )(Contact.apply)(Contact.unapply)
20. )
```

这段代码演示了一条已存在的 `Contact` 如何通过填入数据在表单中显示出来：

```
1.  def editContact = Action {
2.    val existingContact = Contact(
3.      "Fake", "Contact", Some("Fake company"), informations = List(
4.        ContactInformation(
5.          "Personal", Some("fakecontact@gmail.com"),
    List("01.23.45.67.89", "98.76.54.32.10")
6.        ),
7.        ContactInformation(
8.          "Professional", Some("fakecontact@company.com"),
    List("01.23.45.67.89")
9.        ),
10.       ContactInformation(
11.         "Previous", Some("fakecontact@oldcompany.com"), List()
12.       )
13.     )
14.   )
15.   Ok(views.html.contact.form(contactForm.fill(existingContact)))
16. }
```

最后是表单提交：

```
1.  def saveContact = Action { implicit request =>
2.    contactForm.bindFromRequest.fold(
3.      formWithErrors => {
4.        BadRequest(views.html.contact.form(formWithErrors))
5.      },
6.      contact => {
7.        val contactId = Contact.save(contact)
8.
    Redirect(routes.Application.showContact(contactId)).flashing("success
     -> "Contact saved!")
```

```
 9.      }
10.    )
11.  }
```

Protecting against CSRF

Protecting against CSRF

跨站请求伪造（CSRF）是一种安全漏洞。攻击者欺骗受害者浏览器，并使用受害者的会话发送请求。由于每个请求中都含有会话令牌（session token），如果攻击者能够强制使用受害者的浏览器发送请求，也就相当于受害者在以用户的名义发送请求。

最好能先熟悉一下 CSRF ，哪些攻击手段属于 CSRF ，哪些不是。建议参考 OWASP 的相关内容。

简单的说，攻击者能够强迫受害者的浏览器发送以下请求：

- 所有的 `GET` 请求
- 内容类型为 `application/x-www-form-urlencoded` ， `multipart/form-data` 和 `text/plain` 的 `POST` 请求

攻击者不能：

- 强迫浏览器使用 `PUT` 和 `DELETE` 请求
- 强迫浏览器发送其他内容类型的请求，如 `application/json`
- 强迫浏览器发送新的 cookie ，而不是服务器已经设置了的

cookie
- 强迫浏览器设置任意报头，而不是浏览器通常会在请求中添加的那些报头

由于 `GET` 请求是无法更改的，应用中使用该请求不会有任何危险。所以唯一需要防御 CSRF 攻击的是带有如上提到的内容类型的 `POST` 请求。

Play 的 CSRF 防御

Play 支持了多种方法来验证是否为 CSRF 请求。最主要的机制是 CSRF 令牌（token）。该令牌需设置在查询字符串或是每个提交的表单正文中，并且还要设置于用户会话中。Play 之后会验证这两个令牌是否存在并匹配。

为了能够简单的防御那些非浏览器请求，例如通过 AJAX 发送的请求，Play 同样支持以下几种：

- 如果包头中有 `X-Requested-With`，则 Play 会认为该请求安全。很多流行的 Javascript 库都会在请求中加入 `X-Requested-With`，比如 jQuery。
- 如果 `Csrf-Token` 报头的值为 `nocheck`，或是一个有效的 CSRF 令牌，则 Play 会认为该请求安全。

应用全局 CSRF 过滤

Play 提供了全局 CSRF 过滤，可以将其应用于所有请求。这是给应用添加 CSRF 防御最简单的方法。在你项目中的 `build.sbt` 内添加 Play 过滤 helper 依赖，就能启用全局过滤了：

```
1.  libraryDependencies += filters
```

现在需要将过滤器添加到 `Global` 对象中：

```
import play.api._
import play.api.mvc._
import play.filters.csrf._

object Global extends WithFilters(CSRFFilter()) with GlobalSettings {
  // ... onStart, onStop etc
}
```

获得当前令牌

当前 CSRF 令牌可通过调用 `getToken` 方法获取。该方法接收一个隐式的 `RequestHeader`，所以要确保在作用域中设置该参数。

```
import play.filters.csrf.CSRF

val token = CSRF.getToken(request)
```

Play 提供了一些模板 helper 来辅助添加 CSRF 令牌到表单内。第一个是添加到 action URL 的查询字符串中：

```
@import helper._

@form(CSRF(routes.ItemsController.save())) {
    ...
}
```

渲染后的表单如下：

```
<form method="POST" action="/items?csrfToken=1234567890abcdef">
   ...
</form>
```

如果你不想在查询字符串中设置令牌，Play 还提供了 helper，将 CSRF 令牌作为隐藏域添加到表单中：

```
@form(routes.ItemsController.save()) {
    @CSRF.formField
    ...
}
```

渲染后的表单如下：

```
<form method="POST" action="/items">
   <input type="hidden" name="csrfToken" value="1234567890abcdef"/>
   ...
</form>
```

所有表单 helper 方法都要求在作用域中设置隐式的令牌或是请求。通常，如果没有的话，需要在你的模板中设置隐式的 `RequestHeader` 参数。

在会话中添加 CSRF 令牌

为了保证能在表单中找到 CSRF 令牌并发回客户端，如果传入的请求中没有令牌，全局过滤器会为所有接收 HTML 的 `GET` 请求生成一个新的令牌。

基于单个 action 的 CSRF 过滤

有时候应用全局 CSRF 过滤并不合适，比如应用可能需要允许部分跨站表单的提交。有一些并非基于会话的标准，如 OpenID 2.0，需要使用跨站表单提交，或是在服务器到服务器的 RPC 通讯中使用表单提交。

在这样的情况下，Play 提供了两个 action，可供组合到应用的

action 中。

第一个是 `CSRFCheck` action，提供了验证操作。需添加到所有接收已认证会话 POST 表单提交的 action 中：

```
import play.api.mvc._
import play.filters.csrf._

def save = CSRFCheck {
  Action { req =>
    // handle body
    Ok
  }
}
```

第二是个 `CSRFAddToken` action，在传入请求没有令牌的情况下会生成一个 CSRF 令牌。需添加到所有渲染表单的 action 中：

```
import play.api.mvc._
import play.filters.csrf._

def form = CSRFAddToken {
  Action { implicit req =>
    Ok(views.html.itemsForm())
  }
}
```

更简便的方法是将这些 action 和 Play 的 `ActionBuilder` 一起组合使用：

```
import play.api.mvc._
import play.filters.csrf._

object PostAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) =>
Future[Result]) = {
```

```
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) =
CSRFCheck(action)
}

object GetAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) =>
Future[Result]) = {
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) =
CSRFAddToken(action)
}
```

这样可以最大限度的减少在编写 action 时所需的样板代码（boiler plate code）：

```
def save = PostAction {
  // handle body
  Ok
}

def form = GetAction { implicit req =>
  Ok(views.html.itemsForm())
}
```

CSRF 配置选项

以下选项可在 `application.conf` 中配置：

- `csrf.token.name` - 应用于会话和请求正文/查询字符串中的令牌名称。默认为 `csrfToken`。

- `csrf.cookie.name` - 如果配置了该选项，Play 会根据该名称将 CSRF 令牌存储到 cookie，而非会话中。
- `csrf.cookie.secure` - 如果设置了 `csrf.cookie.name`，则 CSRF cookie 是否需要设置安全标志位。默认该值与 `session.secure` 相同。
- `csrf.body.bufferSize` - 为了能在正文中读取令牌，Play 必须缓存正文，并在可能的情况下进行解析。该选项设置了缓存正文时最大缓存大小。默认为 100k。
- `csrf.sign.tokens` - Play 是否使用签名 CSRF 令牌。签名 CSRF 令牌保证了每个请求的令牌值是随机的，这样可以防御 BREACH 攻击。

Custom Validations

Custom Validations

验证包 允许你调用 `verifying` 方法来创建专门的约束。Play 还提供了用 `Constraint` 样本类的方法来自定义约束。

这里，我们会实现一个简单的密码强度约束，通过正则表达式来验证密码不是由全字母或是全数字组成。 `Constraint` 接受一个返回 `ValidationResult` 的函数，我们使用这个函数来返回密码验证的结果：

```
1.  val allNumbers = """\d*""".r
2.  val allLetters = """[A-Za-z]*""".r
3.
4.  val passwordCheckConstraint: Constraint[String] =
    Constraint("constraints.passwordcheck")({
5.    plainText =>
6.      val errors = plainText match {
7.        case allNumbers() => Seq(ValidationError("Password is all
    numbers"))
8.        case allLetters() => Seq(ValidationError("Password is all
    letters"))
9.        case _ => Nil
10.     }
11.     if (errors.isEmpty) {
12.       Valid
13.     } else {
14.       Invalid(errors)
15.     }
16. })
```

注意： 这个例子是为了演示自定义约束而故意设计的。关于正确的密码安全设计，请参考 OWASP 指南

我们还可以结合 `Constraints.min` 来给密码添加额外的一层验证、

```
1.  val passwordCheck: Mapping[String] = nonEmptyText(minLength = 10)
2.    .verifying(passwordCheckConstraint)
```

Custom Field Constructors

Custom Field Constructors

一个表单域的渲染并不仅仅需要包含 `<input>` 标签，还有 `<label>` 和其他一些标签可能被你的 CSS 框架用来装饰表单域。

所有的输入 helper 都会接受一个隐式（implicit）的 `FieldConstructor` 来处理这一部分。默认的构造器（在作用域内没有指定其他域构造器时会被调用），生成的 HTML 类似于：

```
<dl class="error" id="username_field">
    <dt><label for="username">Username:</label></dt>
    <dd><input type="text" name="username" id="username" value="">
</dd>
    <dd class="error">This field is required!</dd>
    <dd class="error">Another error</dd>
    <dd class="info">Required</dd>
    <dd class="info">Another constraint</dd>
</dl>
```

这个默认的域构造器支持传入额外的选项到输入 helper:

```
'_label -> "Custom label"
'_id -> "idForTheTopDlElement"
'_help -> "Custom help"
'_showConstraints -> false
'_error -> "Force an error"
'_showErrors -> false
```

自定义域构造器

通常，你需要自定义域构造器。首先要写一个模板：

```
@(elements: helper.FieldElements)

<div class="@if(elements.hasErrors) {error}">
    <label for="@elements.id">@elements.label(elements.lang)</label>
    <div class="input">
        @elements.input
        <span class="errors">@elements.errors(elements.lang).mkString(", ")</span>
        <span class="help">@elements.infos(elements.lang).mkString(", ")</span>
    </div>
</div>
```

注意： 这只是一个例子。你想要实现多复杂的功能都可以。你同样还能使用 `@elements.field` 来获取原始的表单域。

现在使用模板函数来创建 `FieldConstructor` ：

```
object MyHelpers {
  import views.html.helper.FieldConstructor
  implicit val myFields = FieldConstructor(html.myFieldConstructorTemplate.f)
}
```

还可以让表单 helper 来调用他，只需要将其导入到你的模板中：

```
@import MyHelpers._
```

```
@helper.inputText(myForm("username"))
```

模板会使用你的域构造器来渲染输入文本。

你也可以为 `FieldConstructor` 设置一个隐式（implicit）的值：

```
@implicitField = @{
helper.FieldConstructor(myFieldConstructorTemplate.f) }
```

```
@helper.inputText(myForm("username"))
```

Working with Json

JSON 基础

现代Web应用程序经常需要解析和生成JSON（JavaScrpit Object Notation）格式的数据。Play框架通过JSON库的支持可以完成上述任务。

JSON是一种轻量级的数据交换格式，请看下面一个例子：

```
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
```

如果想了解跟多关于JSON的知识，请访问json.org

Play框架的JSON库

`play.api.libs.json` 包中包含表示JSON数据的数据结构和用于将这些数据结构与其他数据格式互相转换的实用工具。

JsValue

这是一个特质（trait），可以表示任何JSON值。JSON库中通过一系列对JsVlaue进行扩展的case类来表示各种有效的JSON类型。

- JsString
- JsNumber
- JsBoolean
- JsObject
- JsArray

- JsNull

你可以利用这些多种多样的JsValue类型来构造任何JSON结构。

Json

Json对象提供一些工具，这些工具用于将数据格式转换为JsValue结构或者逆向转换。

JsPath

JsPath用于表示JsValue内部结构的路径，类似于XPath对XML的意义。可以利用JsPath析取JsValue结构，或者对隐式转换进行模式匹配。

构造JsValue实例

字符串解析

```
import play.api.libs.json._

val json: JsValue = Json.parse("""
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
```

```
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
```

类构造器

```
import play.api.libs.json._

val json: JsValue = JsObject(Seq(
  "name" -> JsString("Watership Down"),
  "location" -> JsObject(Seq("lat" -> JsNumber(51.235685), "long" -> JsNumber(-1.309197))),
  "residents" -> JsArray(Seq(
    JsObject(Seq(
      "name" -> JsString("Fiver"),
      "age" -> JsNumber(4),
      "role" -> JsNull
    )),
    JsObject(Seq(
      "name" -> JsString("Bigwig"),
      "age" -> JsNumber(6),
      "role" -> JsString("Owsla")
    ))
  ))
))
```

通过 `Json.obj` 和 `Json.arr` 构造可能更简单些。注意大部分值不需要显式得用JsValue类封装，工厂方法会执行隐式转换（接下来是一个例子）。

```
import play.api.libs.json.{JsNull,Json,JsString,JsValue}

val json: JsValue = Json.obj(
  "name" -> "Watership Down",
```

```
  "location" -> Json.obj("lat" -> 51.235685, "long" -> -1.309197),
  "residents" -> Json.arr(
    Json.obj(
      "name" -> "Fiver",
      "age" -> 4,
      "role" -> JsNull
    ),
    Json.obj(
      "name" -> "Bigwig",
      "age" -> 6,
      "role" -> "Owsla"
    )
  )
)
```

Writers转换器

Scala中通过工具方法 `Json.toJson[T](T)(implicit writes:Writes[T])` 。这个功能通过类型转换器 `Writes[T]` 将T类型的数据转换为JsValue。

Play框架的JSON库API接口提供了大部分基础类型的隐式 `Writes` ，例如 `Int` ， `Double` ， `String` 和 `Boolean` 。当然，该JSON库也有针对包含上述基本类型元素的集合的 `Writes` 转换器。

```
import play.api.libs.json._

// basic types
val jsonString = Json.toJson("Fiver")
val jsonNumber = Json.toJson(4)
val jsonBoolean = Json.toJson(false)

// collections of basic types
val jsonArrayOfInts = Json.toJson(Seq(1, 2, 3, 4))
val jsonArrayOfStrings = Json.toJson(List("Fiver", "Bigwig"))
```

如果想把自己定义的模型转换成JsValues，你需要定义隐式

的 `Writes` 转换器，并将它们引入执行环境。

```
import play.api.libs.json._

// basic types
val jsonString = Json.toJson("Fiver")
val jsonNumber = Json.toJson(4)
val jsonBoolean = Json.toJson(false)

// collections of basic types
val jsonArrayOfInts = Json.toJson(Seq(1, 2, 3, 4))
val jsonArrayOfStrings = Json.toJson(List("Fiver", "Bigwig"))
To convert your own models to JsValues, you must define implicit Writes converters and provide them in scope.

case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
import play.api.libs.json._

implicit val locationWrites = new Writes[Location] {
  def writes(location: Location) = Json.obj(
    "lat" -> location.lat,
    "long" -> location.long
  )
}

implicit val residentWrites = new Writes[Resident] {
  def writes(resident: Resident) = Json.obj(
    "name" -> resident.name,
    "age" -> resident.age,
    "role" -> resident.role
  )
}

implicit val placeWrites = new Writes[Place] {
  def writes(place: Place) = Json.obj(
```

```
    "name" -> place.name,
    "location" -> place.location,
    "residents" -> place.residents)
}

val place = Place(
  "Watership Down",
  Location(51.235685, -1.309197),
  Seq(
    Resident("Fiver", 4, None),
    Resident("Bigwig", 6, Some("Owsla"))
  )
)

val json = Json.toJson(place)
```

作为备选，你可以通过配合模式（combinator pattern）来定义自己的 `Writes` 转换器。

注意：关于配合模式（combinator pattern）在JSON Reads/Writes/Formats 选择器一节中有详细介绍。

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val residentWrites: Writes[Resident] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "age").write[Int] and
  (JsPath \ "role").writeNullable[String]
)(unlift(Resident.unapply))

implicit val placeWrites: Writes[Place] = (
```

```
16.    (JsPath \ "name").write[String] and
17.    (JsPath \ "location").write[Location] and
18.    (JsPath \ "residents").write[Seq[Resident]]
19.  )(unlift(Place.unapply))
```

析取JsValue结构

你可以析取JsValue结构并读取特定的值。语法和功能类似于Scala处理XML的方法。

```
1.  注意：下面的例子应用在前面创建的JsValue结构上
```

Simple path `\`

将 `\` 操作符应用于一个 `JsValue` 可以返回跟相关域对应的属性。下面假设有一个JsObject:

```
1.  val lat = json \ "location" \ "lat"
2.  // 返回JsNumber(51.235685)
```

Recursive path `\\`

使用 `\\` 操作符将会递归查找当前对象中以及所有依赖对象中的所有对应域。

```
1.  val names = json \\ "name"
2.  // returns Seq(JsString("Watership Down"), JsString("Fiver"),
    JsString("Bigwig"))
```

Index lookup(for JsArrays)

可以通过索引值从 `JsArray` 中获取值。

```
val bigwig = (json \ "residents")(1)
// returns {"name":"Bigwig","age":6,"role":"Owsla"}
```

解析JsValue结构

字符串工具

- 微型

```
val minifiedString: String = Json.stringify(json)
{"name":"Watership Down","location":
{"lat":51.235685,"long":-1.309197},"residents":
[{"name":"Fiver","age":4,"role":nul
```

- 可读的

```
val readableString: String = Json.prettyPrint(json)
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
```

JsValue.as或者Jsvalue.asOpt

将JsValue对象转换成其他类型的最简单的方法是使用 `JsValue.as[T](implicit fjs：Reads[T]):T` 需要自定义一个类型转换器 `Reads[T]` 来讲 `JsValue` 转换成 `T` 类型的数据(`和Writes[T]相反`)。 跟 `Writes` 一样，JSON库提供了 `Reads` 转换器需要的基本类型。

```
1. val name = (json \ "name").as[String]
2. // "Watership Down"
3.
4. val names = (json \\ "name").map(_.as[String])
5. // Seq("Watership Down", "Fiver", "Bigwig")
```

如果路径（path）不存在或者转换失败， `as` 方法会抛出 `JsResultException` 异常。更安全的方法是使用 `JsValue.asOpt[T](implicit fjs:Reads[T]):Option[T]`

```
1. val nameOption = (json \ "name").asOpt[String]
2. // Some("Watership Down")
3.
4. val bogusOption = (json \ "bogus").asOpt[String]
5. // None
```

尽管asOpt方法更安全，但是如果有错误也不能捕捉到。

使用有效性（validation）

推荐使用 `validate` 方法将 `JsValue` 转换为其他类型(这个方法含有一个 `Read` 类型的参数)。这个方法同时执行有效性验证和类型转换操作，返回的结果类型是 `JsResult` 。 `JsResult` 通过两个类实现：

- JsSuccess——表示验证/转换成功并封装结果。
- JsError——表示验证/转换不成功，并包含有错误列表。

可以使用多种模式来处理验证结果：

```
1.  val json = { ... }
2.
3.  val nameResult: JsResult[String] = (json \ "name").validate[String]
4.
5.  // Pattern matching
6.  nameResult match {
7.    case s: JsSuccess[String] => println("Name: " + s.get)
8.    case e: JsError => println("Errors: " +
    JsError.toFlatJson(e).toString())
9.  }
10.
11. // Fallback value
12. val nameOrFallback = nameResult.getOrElse("Undefined")
13.
14. // map
15. val nameUpperResult: JsResult[String] =
    nameResult.map(_.toUpperCase())
16.
17. // fold
18. val nameOption: Option[String] = nameResult.fold(
19.   invalid = {
20.     fieldErrors => fieldErrors.foreach(x => {
21.       println("field: " + x._1 + ", errors: " + x._2)
22.     })
23.     None
24.   },
25.   valid = {
26.     name => Some(name)
27.   }
28. )
```

由JsValue转换为模型（model）

如果要将JsValue转换为模型，你要定义隐式 `Reads[T]` 转换器，其中T是模型的类型。

注意：此处实现Reads所用的模式和自定义有效性的技术细节在JSON

Reads/Writes/Formats 选择器一节中有详细介绍。

```
1.  case class Location(lat: Double, long: Double)
2.  case class Resident(name: String, age: Int, role: Option[String])
3.  case class Place(name: String, location: Location, residents:
    Seq[Resident])
4.  import play.api.libs.json._
5.  import play.api.libs.functional.syntax._
6.
7.  implicit val locationReads: Reads[Location] = (
8.    (JsPath \ "lat").read[Double] and
9.    (JsPath \ "long").read[Double]
10. )(Location.apply _)
11.
12. implicit val residentReads: Reads[Resident] = (
13.   (JsPath \ "name").read[String] and
14.   (JsPath \ "age").read[Int] and
15.   (JsPath \ "role").readNullable[String]
16. )(Resident.apply _)
17.
18. implicit val placeReads: Reads[Place] = (
19.   (JsPath \ "name").read[String] and
20.   (JsPath \ "location").read[Location] and
21.   (JsPath \ "residents").read[Seq[Resident]]
22. )(Place.apply _)
23.
24.
25. val json = { ... }
26.
27. val placeResult: JsResult[Place] = json.validate[Place]
28. // JsSuccess(Place(...),)
29.
30. val residentResult: JsResult[Resident] = (json \ "residents")
    (1).validate[Resident]
31. // JsSuccess(Resident(Bigwig,6,Some(Owsla)),)
```

下一节： JSON with HTTP

JSON with HTTP

JSON with HTTP

通过 HTTP API 与 JSON 库的组合，Play 可以支持 Content-Type 为 JSON 的 HTTP 请求和响应。

> 关于控制器，*Action* 和路由，详情可见 *HTTP 编程*

我们通过设计一个简单的、Restful 的 web 服务来说明一些必要的概念，通过 GET 来得到实体列表，POST 来创建新的实体。对于所有数据，该 web 服务使用的 Content-Type 均为 JSON。

以下是用于我们服务的模型：

```
1.  case class Location(lat: Double, long: Double)
2.
3.  case class Place(name: String, location: Location)
4.
5.  object Place {
6.
7.    var list: List[Place] = {
8.      List(
9.        Place(
10.         "Sandleford",
11.         Location(51.377797, -1.318965)
12.       ),
13.       Place(
14.         "Watership Down",
15.         Location(51.235685, -1.309197)
```

```
16.       )
17.     )
18.   }
19.
20.   def save(place: Place) = {
21.     list = list ::: List(place)
22.   }
23. }
```

以 JSON 格式提供实体列表

首先，在控制器中导入必要的东西：

```
1. import play.api.mvc._
2. import play.api.libs.json._
3. import play.api.libs.functional.syntax._
4.
5. object Application extends Controller {
6.
7. }
```

在写 `Action` 之前，我们先要处理模型到 `JsValue` 转换的问题，通过定义一个隐式的 `Writes[Place]` 即可。

```
1. implicit val locationWrites: Writes[Location] = (
2.   (JsPath \ "lat").write[Double] and
3.   (JsPath \ "long").write[Double]
4. )(unlift(Location.unapply))
5.
6. implicit val placeWrites: Writes[Place] = (
7.   (JsPath \ "name").write[String] and
8.   (JsPath \ "location").write[Location]
9. )(unlift(Place.unapply))
```

接着就可以写 `Action` 了：

```
def listPlaces = Action {
  val json = Json.toJson(Place.list)
  Ok(json)
}
```

`Action` 拿到一个包含 `Place` 对象的列表，使用 `Json.toJson` 将它们转换为 `JsValue`（用的是隐式 `Writes[Place]`），然后将这个作为结果的 body 返回。Play 识别出该结果是 JSON 格式，然后为响应设置适当的 `Content-Type` 和 body。

最后一步是为我们的 `Action` 添加路由，写在 `conf/routes` 中：

```
GET   /places               controllers.Application.listPlaces
```

我们可以通过浏览器或 HTTP 工具来发送请求进行测试，下面我们通过 curl 进行测试：

```
curl --include http://localhost:9000/places
```

响应是：

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 141

[{"name":"Sandleford","location":
{"lat":51.377797,"long":-1.318965}},{"name":"Watership
Down","location":{"lat":51.235685,"long":-1.309197}}]
```

创建新实体

对于接下来的 `Action`，我们需要定义一个隐式的 `Reads[Place]` 来将 `JsValue` 转换成我们的模型。

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location]
)(Place.apply _)
```

然后，我们来定义这个 `Action`。

```
def savePlace = Action(BodyParsers.parse.json) { request =>
  val placeResult = request.body.validate[Place]
  placeResult.fold(
    errors => {
      BadRequest(Json.obj("status" ->"KO", "message" ->
JsError.toFlatJson(errors)))
    },
    place => {
      Place.save(place)
      Ok(Json.obj("status" ->"OK", "message" -> ("Place
'"+place.name+"' saved.") ))
    }
  )
}
```

这个 `Action` 比前面那个要复杂，需要注意以下几点：

- 该 `Action` 接收的请求的 `Content-Type` 需要是 `text/json` 或 `application/json`，body 包含的是要创建的实体的 JSON 表示。
- 它使用针对 JSON 的 `BodyParser` 来解析请求，并将 `request.body` 解析成 `JsValue`。
- 我们使用 `validate` 方法来做转换，它依赖于前面定义的隐式 `Reads[Place]`。

- 我们使用一个带有错误和成功处理的 `fold` 来处理 `validate` 的结果。这种模式也可以用于表单提交。
- 该 `Action` 发送的响应也是 JSON 格式的。

最后我们在 `conf/routes` 中加上路由绑定：

```
POST  /places               controllers.Application.savePlace
```

下面我们用有效及无效的请求来测试这个 action，以验证成功及错误处理的工作流。

使用有效数据测试：

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"name":"Nuthanger Farm","location":{"lat" :
51.244031,"long" : -1.263224}}'
  http://localhost:9000/places
```

响应：

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"status":"OK","message":"Place 'Nuthanger Farm' saved."}
```

使用无效数据测试（“name” 字段缺失）：

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"location":{"lat" : 51.244031,"long" : -1.263224}}'
  http://localhost:9000/places
```

响应：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 79

{"status":"KO","message":{"obj.name":
[{"msg":"error.path.missing","args":[]}]}}
```

使用无效数据测试（“lat” 数据类型错误）：

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"name":"Nuthanger Farm","location":{"lat" : "xxx","long"
: -1.263224}}'
  http://localhost:9000/places
```

响应：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 92

{"status":"KO","message":{"obj.location.lat":
[{"msg":"error.expected.jsnumber","args":[]}]}}
```

总结

Play 天生支持 REST 和 JSON，因此开发此类服务应该是相当简单直观的。大部分的工作就是在为你的模型写 `Reads` 和 `Writes`，下一节我们将来详细介绍。

JSON Reads/Writes/Format Combinators

JSON Reads/Writes/Format Combinators

在 JSON 基础一节中，我们介绍了 `Reads` 和 `Writes` 转换器。使用它们，我们可以在 `JsValue` 结构和其他数据类型之间做转换。这一节将更详细地介绍如何构建这些转换器，以及在转换的过程中如何进行验证。

这一节使用到的 `JsValue` 结构以及相应的模型：

```
1.  import play.api.libs.json._
2.
3.  val json: JsValue = Json.parse("""
4.  {
5.    "name" : "Watership Down",
6.    "location" : {
7.      "lat" : 51.235685,
```

```
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
```

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents:
Seq[Resident])
```

JsPath

`JsPath` 是构建 `Reads/Writes` 的核心。`JsPath` 指明了数据在 `JsValue` 中的位置。你可以使用 `JsPath` 对象（根路径）来定义一个 `JsPath` 子实例，语法与遍历 `JsValue` 相似：

```
import play.api.libs.json._

val json = { ... }

// Simple path
val latPath = JsPath \ "location" \ "lat"

// Recursive path
val namesPath = JsPath \\ "name"

// Indexed path
```

```
12.  val firstResidentPath = (JsPath \ "residents")(0)
```

`play.api.libs.json` 包中为 `JsPath` 定义了一个别名：`__`（两个下划线），如果你喜欢的话，你也可以使用它：

```
1.  val longPath = __ \ "location" \ "long"
```

Reads

`Reads` 转换器用于将 `JsValue` 转换成其他类型。你可以通过组合与嵌套 `Reads` 来构造更复杂的 `Reads`。

你需要导入以下内容来创建 `Reads`：

```
1.  import play.api.libs.json._ // JSON library
2.  import play.api.libs.json.Reads._ // Custom validation helpers
3.  import play.api.libs.functional.syntax._ // Combinator syntax
```

Path Reads

`JsPath` 含有以下两个方法可用来创建特殊的 `Reads`，它将应用另一个 `Reads` 到指定路径的 `JsValue`：

- `JsPath.read[T](implicit r: Reads[T]): Reads[T]` - 创建一个 `Reads[T]`，它将应用隐式参数 `r` 到该路径的 `JsValue`。
- `JsPath.readNullable[T](implicit r: Reads[T]): Reads[Option[T]]` - 在路径可能缺失，或包含一个空值时使用。

> 注意：*JSON* 库为基本类型（如 *String*，*Int*，*Double* 等）提供了隐式 *Reads*。

定义具体某一路径的 `Reads`：

```
1.  val nameReads: Reads[String] = (JsPath \ "name").read[String]
```

定义具体某一路径的 `Reads`，并且包含自定义样例类：

```
1.  case class DisplayName(name:String)
2.  val nameReads: Reads[DisplayName] = (JsPath \
    "name").read[String].map(DisplayName(_))
```

复合 Reads

你可以将多个单路径 `Reads` 组合成一个复合 `Reads`，这样就可以转换复杂模型了。

为了便于理解，我们先将一个组合结果分解成两条语句。首先通过 `and` 组合子来组合 `Reads` 对象：

```
1.  val locationReadsBuilder =
2.    (JsPath \ "lat").read[Double] and
3.    (JsPath \ "long").read[Double]
```

上面产生的结果类型为 `'FunctionalBuilder[Reads]#CanBuild2[Double, Double]'`，这只是一个中间结果，它将被用来创建一个复合 `Reads`。

第二步调用 `CanBuildX` 的 `apply` 方法来将单个的结果转换成你的模型，这将返回一个复合 `Reads`。如果你有一个带有构造器签名的样例类，你就可以直接使用它的 `apply` 方法：

```
1.  implicit val locationReads =
    locationReadsBuilder.apply(Location.apply _)
```

将上述代码合成一条语句：

```
1.  implicit val locationReads: Reads[Location] = (
```

```
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)
```

Reads 验证

`JsValue.validate` 方法已经在 JSON 基础 一节中介绍过，我们推荐它进行验证以及用它将 `JsValue` 转换成其它类型。以下是基本用法：

```
val json = { ... }

val nameReads: Reads[String] = (JsPath \ "name").read[String]

val nameResult: JsResult[String] = json.validate[String](nameReads)

nameResult match {
  case s: JsSuccess[String] => println("Name: " + s.get)
  case e: JsError => println("Errors: " +
JsError.toFlatJson(e).toString())
}
```

`Reads` 的默认验证比较简单，比如只检查类型转换错误。你可以通过使用 `Reads` 的验证 helper 来自定义验证规则。以下是一些常用的 helper：

- `Reads.email` - 验证字符串是邮箱地址格式。
- `Reads.minLength(nb)` - 验证一个字符串的最小长度。
- `Reads.min` - 验证最小数值。
- `Reads.max` - 验证最大数值。
- `Reads[A] keepAnd Reads[B] => Reads[A]` - 尝试 `Reads[A]` 及 `Reads[B]` 但最终只保留 `Reads[A]` 结果的运算符（如果你知道 Scala 解析组合子，即有 `keepAnd == <~`）。

- `Reads[A] andKeep Reads[B] => Reads[B]` - 尝试 `Reads[A]` 及 `Reads[B]` 但最终只保留 `Reads[B]` 结果的运算符（如果你知道 Scala 解析组合子，即有 `andKeep == ~>`）。
- `Reads[A] or Reads[B] => Reads` - 执行逻辑或并保留最后选中的 Reads 的运算符。

想添加验证，只需将相应 helper 作为 `JsPath.read` 的参数即可：

```
1.  val improvedNameReads =
2.    (JsPath \ "name").read[String](minLength[String](2))
```

将它们合起来

通过使用复合 `Reads` 和自定义验证，我们可以为模型定义一组有效的 `Reads` 并应用它们：

```
1.  import play.api.libs.json._
2.  import play.api.libs.json.Reads._
3.  import play.api.libs.functional.syntax._
4.
5.  implicit val locationReads: Reads[Location] = (
6.    (JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
7.    (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
8.  )(Location.apply _)
9.
10. implicit val residentReads: Reads[Resident] = (
11.   (JsPath \ "name").read[String](minLength[String](2)) and
12.   (JsPath \ "age").read[Int](min(0) keepAnd max(150)) and
13.   (JsPath \ "role").readNullable[String]
14. )(Resident.apply _)
15.
16. implicit val placeReads: Reads[Place] = (
17.   (JsPath \ "name").read[String](minLength[String](2)) and
18.   (JsPath \ "location").read[Location] and
19.   (JsPath \ "residents").read[Seq[Resident]]
```

```
)(Place.apply _)


val json = { ... }

json.validate[Place] match {
  case s: JsSuccess[Place] => {
    val place: Place = s.get
    // do something with place
  }
  case e: JsError => {
    // error handling flow
  }
}
```

注意，复合 `Reads` 可以嵌套使用。在上面的例子中，`placeReads` 使用了前面定义的隐式 `locationReads` 和 `residentReads`。

Writes

`Writes` 用于将其它类型转换成 `JsValue`。

为样例类构建简单的 Writes，只需在 `Writes` 的 apply 方法体中使用一个函数即可：

```
case class DisplayName(name:String)
implicit val displayNameWrite: Writes[DisplayName] = Writes {
  (displayName: DisplayName) => JsString(displayName.name)
}
```

你可以使用 `JsPath` 和组合子来构建复合 `Writes`，这一点与 `Reads` 非常相似。以下是我们模型的 `Writes`：

```
import play.api.libs.json._
import play.api.libs.functional.syntax._
```

```
3.
4.  implicit val locationWrites: Writes[Location] = (
5.    (JsPath \ "lat").write[Double] and
6.    (JsPath \ "long").write[Double]
7.  )(unlift(Location.unapply))
8.
9.  implicit val residentWrites: Writes[Resident] = (
10.   (JsPath \ "name").write[String] and
11.   (JsPath \ "age").write[Int] and
12.   (JsPath \ "role").writeNullable[String]
13. )(unlift(Resident.unapply))
14.
15. implicit val placeWrites: Writes[Place] = (
16.   (JsPath \ "name").write[String] and
17.   (JsPath \ "location").write[Location] and
18.   (JsPath \ "residents").write[Seq[Resident]]
19. )(unlift(Place.unapply))
20.
21.
22. val place = Place(
23.   "Watership Down",
24.   Location(51.235685, -1.309197),
25.   Seq(
26.     Resident("Fiver", 4, None),
27.     Resident("Bigwig", 6, Some("Owsla"))
28.   )
29. )
30.
31. val json = Json.toJson(place)
```

复合 `Writes` 与复合 `Reads` 有以下不同点：

- 单个路径的 `Writes` 通过 `JsPath.write` 方法来创建。
- 将模型转换成 `JsValue` 无需验证，因此也不需要验证 helper。
- 中间结果 `FunctionalBuilder#CanBuildX`（由 `and` 组合子产生）接收一个函数，该函数将复合类型 `T` 转换成一个元组，该元组与

单路径 `Writes` 匹配。尽管看起来和 `Reads` 非常对称，样例类的 `unapply` 方法返回的是属性元组的 `Option` 类型，因此需要使用 `unlift` 方法将元组提取出来。

递归类型

有一种特殊的情况是上面的例子没有讲到的，即如何处理递归类型的 `Reads` 和 `Writes`。`JsPath` 提供了 `lazyRead` 和 `lazyWrite` 方法来处理这种情况：

```
1.  case class User(name: String, friends: Seq[User])
2.
3.  implicit lazy val userReads: Reads[User] = (
4.    (__ \ "name").read[String] and
5.    (__ \ "friends").lazyRead(Reads.seq[User](userReads))
6.  )(User)
7.
8.  implicit lazy val userWrites: Writes[User] = (
9.    (__ \ "name").write[String] and
10.   (__ \ "friends").lazyWrite(Writes.seq[User](userWrites))
11. )(unlift(User.unapply))
```

Format

`Format[T]` 是 `Reads` 和 `Writes` 组合的特性（trait），它可以代替 `Reads` 和 `Writes` 来做隐式转换。

通过 Reads 和 Writes 创建 Format

你可以通过 `Reads` 和 `Writes` 来构建针对同一类型的 `Format`：

```
1.  val locationReads: Reads[Location] = (
```

```
  (JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply _)

val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val locationFormat: Format[Location] =
  Format(locationReads, locationWrites)
```

使用组合子创建 Format

对于 `Reads` 和 `Writes` 对称的情况（实际应用中不一定能满足对称的条件），你可以直接通过组合子来创建 Format:

```
implicit val locationFormat: Format[Location] = (
  (JsPath \ "lat").format[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").format[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply, unlift(Location.unapply))
```

JSON Transformers

JSON Transformers

> 注意，这一节的内容最早由 *Pascal Voitot* 发表在 *mandubian.com* 上。（文章太旧，请带着批判的眼光去读。）

现在已经知道如何验证 JSON，以及如何将 JSON 转成任意结构或将

任意结构转成 JSON。但当我开始用那些组合子来写 web 应用，我立即遇到了这样的情况：从网络中读取 JSON，验证它然后再将它转成 JSON。

JSON coast-to-coast 设计介绍

我们注定要将 JSON 转成 OO 吗？

近几年来，几乎在所有的 web 框架中（除了最近出现的用 JS 写服务端的情况，JSON 就是默认的数据结构），我们都要做的一件事就是：从网络中读取 JSON，然后将 JSON 转成 OO 结构，比如转成类（或 Scala 中的样例类），为什么？

- 一个好的理由：OO 结构是「语言原生」的，对于你的业务逻辑，在保证与 web 层隔离的情况下，可以以一种无缝的方式操作数据。
- 一个更值得怀疑的理由：ORM 框架只能使用 OO 结构与数据库进行对接，我们基本上确信在现有 ORM 的特性下（不管好的坏的），这是无法通过其他方式来完成的。

OO 转换真的是默认的使用案例吗？

在许多情况下，你并不需要真的对数据执行什么业务逻辑，更多的是在存储前或提取后，对数据进行验证和转换。

让我们来看一下「增/删/改/查」操作：

- 你通过网络请求获得数据，验证它们是没有问题的，然后将它们插入数据库，或用它们更新数据库。
- 另一种情况是，你从数据库中取得数据，然后将他们发送出去。

因此，一般情况下，对于「增/删/改/查」操作，你把 JSON 转换成 OO 结构仅仅是因为框架的限定。

> 我并不是因此就说你不应该把 *JSON* 转成 *OO* 结构，但大部分情况下你可以不必这么做。我们应该在只有真正的业务逻辑需要处理的时候，才将相关数据转成 *OO* 结构。

新技术玩家改变操作 JSON 的方式

除了上述事实，在数据库上我们多了一些新的选择，如 MongoDB 或 CouchDB，它们接收的是类似于 JSON 树的文档结构数据（BSON）。

对于这些数据库，我们还有一些好用的工具如 ReactiveMongo，它提供了一个响应式的环境以非常自然的方式流式地将数据写入与读出 MongoDB。

在写 Play2-ReactiveMongo 模块的同时，我与 Stephane Godbillon 也在将 ReactiveMongo 集成到 Play2.1 中。除了为 Play2.1 提供 MongoDB 的便捷操作，该模块还支持 JSON 与 BSON 之间的转换。

> 这意味着你可以操作 *JSON* 流直接读写数据库，而无需将它们转成 *OO* 结构。

JSON coast-to-coast 设计

考虑到这一点，我们可以简单地想象以下情形：

- 接收 JSON
- 验证 JSON
- 将 JSON 变换成对应数据库的文档结构
- 直接发送 JSON 给数据库

当从数据库中取数据对外服务时，也是相似的情况：

- 直接从数据库中取出 JSON 格式的数据

- 过滤/变换这个 JSON，选取那些允许展示给客户端的数据（例如，你应该不想把安全信息也发送出去）
- 直接发送 JSON 给客户端

在这种情况下，我们可以非常简单地想象在客户端与数据库之间操作 JSON 格式的数据流而无需做 JSON 以外的变换。自然地，当你将这种变换流融入 Play2.1 提供的响应式基础设施中，突然间就为你打开了新的视野。

> 这被我叫作 *JSON coast-to-coast* 设计：
>
> - 不要把 *JSON* 数据视为一块一块的，而是把它看成客户端与数据库之间的数据流
> - 把 *JSON* 流视为一个管道，你可以将它与其它管道相连，同时对它进行修改和变换
> - 以异步/非阻塞的方式看待数据流
>
> 这也是 *Play2.1* 成为响应式体系结构的一个原因。我相信把你的应用视为承载数据流的棱镜将极大改变你设计 *web* 应用的方式（如果看不懂，请略过，译者也看不懂）。它可能会开拓一个比传统架构更适应今天 *web* 应用需求的新领域。

因此，正如你自己推断出来的那样，想要直接基于验证和变换操作 JSON 流，我们需要一些新的工具。JSON 组合子是个不错的选择，但它们过于通用了。这就是为什么我们创造了一些更加专用的组合子和 API 来做这件事，我们把它们称为 JSON 变换器（JSON transformers）。

JSON 变换器

JSON 变换器其实就是 `f:JSON => JSON`。因此一个 JSON 变换器可以是一个简单的 `Writes[A <: JsValue]`。但一个 JSON 变换器并不仅仅是一个函数，正如我们之前所说，我们想在变换 JSON 的同时验证它。最终结果是，一个 JSON 变换器其实是一个 `Reads[A <: JsValue]`。

> 注意：`Reads[A <: JsValue]` 并不仅仅能读取/验证，它还可以进行变换。

使用 `JsValue.transform` 而不是 `JsValue.validate`

我们提供了一个 helper 方法，以此帮助人们将 `Reads[T]` 视为一个变换器（transformer），而不仅仅是一个验证器（validator）。

```
1. JsValue.transform[A <: JsValue](reads: Reads[A]): JsResult[A]
```

该函数签名与 `JsValue.validate(reads)` 是类似的。

细节

在接下来的示例代码中，我们将使用以下的 JSON 数据：

```
1. {
2.   "key1" : "value1",
3.   "key2" : {
4.     "key21" : 123,
5.     "key22" : true,
6.     "key23" : [ "alpha", "beta", "gamma"],
7.     "key24" : {
8.       "key241" : 234.123,
9.       "key242" : "value242"
10.    }
11.  },
12.  "key3" : 234
13. }
```

案例 1：在 JsPath 中取 JSON 值

取值（作为 JsValue）

```
1. import play.api.libs.json._
2.
```

```
3.  val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick
4.
5.  scala> json.transform(jsonTransformer)
6.  res9: play.api.libs.json.JsResult[play.api.libs.json.JsValue] =
7.      JsSuccess(
8.        ["alpha","beta","gamma"],
9.        /key2/key23
10.     )
```

`(__ \ 'key2 \ 'key23).json...`

- 所有的 JSON 变换器都在 `JsPath.json` 中

`(__ \ 'key2 \ 'key23).json.pick`

- `pick` 是一个 `Reads[JsValue]`，根据给定的 JsPath 取值。这里值是：`["alpha","beta","gamma"]`

`JsSuccess(["alpha","beta","gamma"],/key2/key23)`

- 这就是一个取值成功后的 `JsResult`
- `/key2/key23` 表示读取这些值的 JsPath，不过不用管它，把它们放这里只是为了组成一个 `JsResult`
- 出来 `["alpha","beta","gamma"]` 是由于我们重写了 `toString`

> 注意：*`jsPath.json.pick`* 只会取 *JsPath* 中的值。

取值（作为类型）

```
1.  import play.api.libs.json._
2.
3.  val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick[JsArray]
4.
5.  scala> json.transform(jsonTransformer)
6.  res10: play.api.libs.json.JsResult[play.api.libs.json.JsArray] =
7.      JsSuccess(
8.        ["alpha","beta","gamma"],
9.        /key2/key23
```

```
    )
```

```
(__ \ 'key2 \ 'key23).json.pick[JsArray]
```

`pick[T]` 是一个 `Reads[T <: JsValue]`，根据给定的 JsPath 取值（在我们的例子中，取出来是一个 `JsArray`）。

> 注意：*`jsPath.json.pick[T <: JsValue]`* 只提取 *JsPath* 中相应类型（*T*）的值。

案例 2：根据 JsPath 提取分支

提取分支（作为 JsValue）

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key24 \ 'key241).json.pickBranch

scala> json.transform(jsonTransformer)
res11: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
  JsSuccess(
    {
      "key2": {
        "key24":{
          "key241":234.123
        }
      }
    },
    /key2/key24/key241
  )
```

`(__ \ 'key2 \ 'key24 \ 'key241).json.pickBranch`

- `pickBranch` 是一个 `Reads[JsValue]`，根据给定的 JsPath 提取对应的 JSON 分支。

```
{"key2":{"key24":{"key241":234.123}}}
```

- 分支提取结果。

> 注意：`jsPath.json.pickBranch` 根据 *JsPath* 提取单条分支以及其中的值。

案例 3：从输入 JsPath 拷贝值到新的 JsPath

```
1.  import play.api.libs.json._
2.
3.  val jsonTransformer = (__ \ 'key25 \ 'key251).json.copyFrom( (__ \
    'key2 \ 'key21).json.pick )
4.
5.  scala> json.transform(jsonTransformer)
6.  res12: play.api.libs.json.JsResult[play.api.libs.json.JsObject]
7.    JsSuccess(
8.      {
9.        "key25":{
10.         "key251":123
11.       }
12.     },
13.     /key2/key21
14.   )
```

```
(__ \ 'key25 \ 'key251).json.copyFrom( reads: Reads[A <: JsValue] )
```

- `copyFrom` 是一个 `Reads[JsValue]`
- `copyFrom` 使用提供的 `Reads[A]` 从给定的 JSON 中读取 JsValue
- `copyFrom` 将提取出来的 JsValue 拷贝到新分支的叶子节点

```
{"key25":{"key251":123}}
```

- `copyFrom` 读出值 `123`
- `copyFrom` 将这个值拷贝进新分支：`(__ \ 'key25 \ 'key251)`

> 注意：`jsPath.json.copyFrom(Reads[A <: JsValue])` 从输入 *JSON* 中读值，然

> 后将它拷贝进新创建的分支中。

案例 4：拷贝整个输入 JSON & 更新一个分支

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key24).json.update(
    __.read[JsObject].map{ o => o ++ Json.obj( "field243" -> "coucou" ) }
)

scala> json.transform(jsonTransformer)
res13: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
  JsSuccess(
    {
      "key1":"value1",
      "key2":{
        "key21":123,
        "key22":true,
        "key23":["alpha","beta","gamma"],
        "key24":{
          "key241":234.123,
          "key242":"value242",
          "field243":"coucou"
        }
      },
      "key3":234
    },
  )
```

`(__ \ 'key2).json.update(reads: Reads[A < JsValue])`

- 这是一个 `Reads[JsObject]`

`(__ \ 'key2 \ 'key24).json.update(reads)` 做了以下 3 件事：

- 从输入 JSON 中提取路径 `(__ \ 'key2 \ 'key24)` 上的值

- 应用 `reads` 在这个值上，并重新创建一个 `(__ \ 'key2 \ 'key24)` 分支，将 `reads` 的结果加到这个分支上
- 用新的分支替代输入 JSON 中的原有分支（因此它只能用于 JsObject 而非其它类型的 JsValue）

`JsSuccess({…},)`

- 我们可以看到在返回的结果中，并没有 JsPath 作为第二个参数，因为 JSON 操作已经从根 JsPath 处完成

> `jsPath.json.update(Reads[A <: JsValue])` 只能用于 *JsObject*，拷贝整个 *JsObject* 然后用提供的 `Reads[A <: JsValue]` 更新 *jsPath*

案例 5：在新分支中放置一个给定值

```
1.  import play.api.libs.json._
2.
3.  val jsonTransformer = (__ \ 'key24 \
    'key241).json.put(JsNumber(456))
4.
5.  scala> json.transform(jsonTransformer)
6.  res14: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
7.    JsSuccess(
8.      {
9.        "key24":{
10.         "key241":456
11.       }
12.     },
13.   )
```

`(__ \ 'key24 \ 'key241).json.put( a: => JsValue )`

- 这是一个 `Reads[JsObject]`
- 创建一个新的分支：`(__ \ 'key24 \ 'key241)`
- 将 `a` 值放入这个分支中

`jsPath.json.put( a: => JsValue )`

- 接收一个 JsValue 参数（按名称传参），甚至可以允许传递一个闭包

`jsPath.json.put`

- 并不关心输入 JSON 是什么
- 简单地用给定值替换输入 JSON

> 注意：*`jsPath.json.put( a: => Jsvalue )`* 用给定的值创建一个新分支，无需考虑输入 *JSON* 是什么

案例 6：从输入 JSON 中剪掉一个分支

```
1.  import play.api.libs.json._
2.
3.  val jsonTransformer = (__ \ 'key2 \ 'key22).json.prune
4.
5.  scala> json.transform(jsonTransformer)
6.  res15: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
7.    JsSuccess(
8.      {
9.        "key1":"value1",
10.       "key3":234,
11.       "key2":{
12.         "key21":123,
13.         "key23":["alpha","beta","gamma"],
14.         "key24":{
15.           "key241":234.123,
16.           "key242":"value242"
17.         }
18.       }
19.     },
20.     /key2/key22/key22
21.   )
```

`(__ \ 'key2 \ 'key22).json.prune`

- 这是一个 `Reads[JsObject]`，只用于 JsObject
- 从输入 JSON 中移除给定的 JsPath（`key2` 下的 `key22` 已经被剪掉了）

我们可以注意到输出的 JsObject 的键（key）的顺序与输入 JsObject 是不一样的。这是由 JsObject 的实现及合并机制所导致，但键的顺序不一致并不重要，因为我们重写了 `JsObject.equals` 方法，并且把这种情况考虑在内了。

> 注意：`jsPath.json.prune` 只能用于 *JsObject*，它的作用是从输入 *JSON* 中移除给定的 *JsPath*
> 还需注意以下两点：
> `prune` 暂时无法用于递归的 *JsPath*
> 如果 `prune` 无法找到可以删除的分支，它并不会产生错误，而是将 *JSON* 原样返回

案例 7：选择一个分支并在两处更新它的内容

```
1.  import play.api.libs.json._
2.  import play.api.libs.json.Reads._
3.
4.  val jsonTransformer = (__ \ 'key2).json.pickBranch(
5.    (__ \ 'key21).json.update(
6.      of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }
7.    ) andThen
8.    (__ \ 'key23).json.update(
9.      of[JsArray].map{ case JsArray(arr) => JsArray(arr :+
    JsString("delta")) }
10.   )
11. )
12.
13. scala> json.transform(jsonTransformer)
14. res16: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
15.   JsSuccess(
16.     {
17.       "key2":{
18.         "key21":133,
```

```
        "key22":true,
        "key23":["alpha","beta","gamma","delta"],
        "key24":{
          "key241":234.123,
          "key242":"value242"
        }
      }
    },
    /key2
  )
```

`(__ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])`

- 从输入 JSON 中提取分支 `__ \ 'key2` 并对该分支下的叶子结点应用 `reads`（只针对内容）

`(__ \ 'key21).json.update(reads: Reads[A <: JsValue])`

- 更新 `(__ \ 'key21)` 分支

`of[JsNumber]`

- 这就是一个 `Reads[JsNumber]`
- 从路径 `(__ \ 'key21)` 下提取一个 JsNumber

`of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }`

- 读取一个 JsNumber（即路径 `__ \ 'key21` 下的值 123）
- 使用 `Reads[A].map` 将值增加 10（创建一个比原来大 10 的值并替换原来的）

`andThen`

- 用于组合两个 `Reads[A]`
- 应用第一个 `reads` 后将结果以管道的方式送给第二个 `reads` 处理

`of[JsArray].map{ case JsArray(arr) => JsArray(arr :+ JsString("delta")`

- 读取一个 JsArray（即路径 `__ \ 'key23` 下的值 [“alpha”,”beta”,”gamma”]）
- 使用 `Reads[A].map` 在上述值后追加一个 `JsString("delta")`

> 注意：得到的结果仅是 `__ \ 'key2` 分支，因为我们只选取了它

案例 8：选取一条分支并将其子分支剪掉

```
1.  import play.api.libs.json._
2.
3.  val jsonTransformer = (__ \ 'key2).json.pickBranch(
4.    (__ \ 'key23).json.prune
5.  )
6.
7.  scala> json.transform(jsonTransformer)
8.  res18: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
9.    JsSuccess(
10.     {
11.       "key2":{
12.         "key21":123,
13.         "key22":true,
14.         "key24":{
15.           "key241":234.123,
16.           "key242":"value242"
17.         }
18.       }
19.     },
20.     /key2/key23
21.   )
```

`(__ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])`

- 从输入 JSON 中提取分支 `__ \ 'key2` 并对该分支下的叶子结点应用 `reads`（只针对内容）

`(__ \ 'key23).json.prune`

- 从上述结果中移除 `__ \ 'key23` 分支

> 注意最终结果是一个没有 *`key23`* 的 *`__ \ 'key2`* 分支

那组合子（combinator）呢？

在话题变得枯燥无聊前，我及时打住了。

你只需要住记，你现在有一个非常强大的工具包来创建通用的 JSON 变换器。你可以组合，map，flatmap 这些变换器，因此几乎有无限种可能性。

最后，我们还要把这些新产生的 JSON 变换器与之前的 `Reads` 组合子组合起来用。

让我们通过下面的例子来展示（一个将 Gizmo 转为 Gremlin 的 JSON 变换器）。

下面是 Gizmo:

```
1.  val gizmo = Json.obj(
2.    "name" -> "gizmo",
3.    "description" -> Json.obj(
4.      "features" -> Json.arr( "hairy", "cute", "gentle"),
5.      "size" -> 10,
6.      "sex" -> "undefined",
7.      "life_expectancy" -> "very old",
8.      "danger" -> Json.obj(
9.        "wet" -> "multiplies",
10.       "feed after midnight" -> "becomes gremlin"
11.     )
12.   ),
13.   "loves" -> "all"
14. )
```

以下是 Gremlin:

```
val gremlin = Json.obj(
  "name" -> "gremlin",
  "description" -> Json.obj(
    "features" -> Json.arr("skinny", "ugly", "evil"),
    "size" -> 30,
    "sex" -> "undefined",
    "life_expectancy" -> "very old",
    "danger" -> "always"
  ),
  "hates" -> "all"
)
```

让我们来写一个 JSON 变换器来完成 Gizmo 到 Gremlin 的变换:

```
import play.api.libs.json._
import play.api.libs.json.Reads._
import play.api.libs.functional.syntax._

val gizmo2gremlin = (
    (__ \ 'name).json.put(JsString("gremlin")) and
    (__ \ 'description).json.pickBranch(
        (__ \ 'size).json.update( of[JsNumber].map{ case JsNumber(size) => JsNumber(size * 3) } ) and
        (__ \ 'features).json.put( Json.arr("skinny", "ugly", "evil") ) and
        (__ \ 'danger).json.put(JsString("always"))
        reduce
    ) and
    (__ \ 'hates).json.copyFrom( (__ \ 'loves).json.pick )
) reduce

scala> gizmo.transform(gizmo2gremlin)
res22: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
  JsSuccess(
    {
      "name":"gremlin",
      "description":{
```

```
22.         "features":["skinny","ugly","evil"],
23.         "size":30,
24.         "sex":"undefined",
25.         "life_expectancy":
26.         "very old","danger":"always"
27.       },
28.       "hates":"all"
29.     },
30.   )
```

搞定！我不打算解释上面的变换了，因为看完上面的内容后你们应该能理解了。需要注意一点：

`(__ \ 'features).json.put(…)` 放在 `(__ \ 'size).json.update` 之后，因此它可以覆盖原有的 `(__ \ 'features)`。

`(Reads[JsObject] and Reads[JsObject]) reduce`

- 它将两个 `Reads[JsObject]` 合并（JsObject ++ JsObject）
- 它将同一个 JSON 应用到两个 `Reads[JsObject]`，不像 `andThen`，`andThen` 是将第一个 reads 的处理结果注入给第二个进行处理

JSON Macro Inception

JSON Macro Inception

> 注意，这一节的内容最早由 *Pascal Voitot* 发表在 *mandubian.com* 上。（文章太旧，请带着批判的眼光去读。）
>
> 这个特性还处于实验中，因为 *Scala* 宏在 *Scala 2.10.0* 中仍是实验性的。如果你不想使用 *Scala* 中的实验性特性，请手写 *Reads/Writes/Format*，同样可以达到一样的效果。

写样例类（case class）的默认 Reads/Writes/Format 是非常无聊的

还记得你是如何为一个样例类写 `Reads[T]` 的吗：

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = (
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
)(Person)
```

你为这个样例类写了 5 行代码，你知道吗，许多人认为为他们的类写一个 `Reads[TheirClass]` 是非常不 cool 的，因为像 Java 的 JSON 框架，如 Jackson 或 Gson，会为你做这些事情，而你根本不需要写这些多余的代码。

我们会这么说 Play2.1 的 JSON 序列化与反序列化：

- 完全类型安全的
- 完全编译的
- 在运行时使用内省/反射机制，无需任务操作

但对于一些人来说，以上的好处并无法抹平额外代码带来的麻烦。

我们相信这是一个非常好的方法，因此坚持并提出：

- JSON 简化语法
- JSON 组合子
- JSON 变换器

虽然被加强了，但仍然没改变要写额外代码的事实。

做一个极简主义者

鉴于我们是完美主义者，我们提出了一种新的方法来达到同样的效果：

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = Json.reads[Person]
```

只需要一行！你马上可能会问：

> 它有使用运行时字节码增强吗？ -> 没有
> 它有使用运行时自省机制吗？ -> 没有
> 它会打破类型安全吗？ -> 不会

所以呢？

> 在创造了 *JSON coast-to-coast* 设计一词后，让我们把它叫做：*JSON Inception*。

JSON Inception

代码等价性

正如之前所解释的：

```
import play.api.libs.json._
// please note we don't import functional.syntax._ as it is managed by the macro itself

implicit val personReads = Json.reads[Person]

// IS STRICTLY EQUIVALENT TO writing

implicit val personReads = (
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
```

```
12.  )(Person)
```

Inception 等式

下面是描述 inception 概念的等式：

```
1.  (Case Class INSPECTION) + (Code INJECTION) + (COMPILE Time) =
    INCEPTION
```

样例类检查

也许你自己就可以推断出来，为了达到前面说的代码等价性，我们需要：

- 检查 `Person` 样例类
- 提取 `name`，`age` 和 `lovesChocolate` 3 个字段以及它们的类型
- 隐式解析类型类
- 找到 `Person.apply`

注入？

我要立马阻止你，并不是你想的那样

> 代码注入并不是依赖注入。不是 *Spring* 那套东西，没有 *IOC*，也没有 *DI*。

我是故意使用这个词的，因为我知道说到「注入」，大家马上会联想到 IOC 和 Spring。但我还是想用这个词的真实涵义还重新建立大家对它的认识。这里，代码注入指的就是在编译期，我们把代码注入到已编译的 Scala AST 中（Abstract Syntax Tree，抽象语法树）。

因此，`Json.reads[Person]` 会被编译并用下面内容替换到编译后的 AST 中：

```
(
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
)(Person)
```

不多也不少。

编译期

没错，一切都是在编译期执行的。并没有运行时字节码增强，也没有运行时自省。

> 由于一切都是在编译期解析的，因此如果没有导入各个字段类型所需的隐式转换，就会报编译错误。

Json inception 是 Scala 2.10 中的宏

我们需要启用 Scala 的一个特性来支持 Json inception:

- 编译期代码增强
- 编译期类/隐式检查
- 编译期代码注入

以上这些可以由 Scala 2.10 中的一个新的实验性特性来提供: Scala 宏。

Scala 宏是一个拥有具大潜力的新特性（仍是实验性的）。有了它，可以:

- 在编译期使用 Scala 提供的反射 API 进行代码自省
- 在当前的编译上下文中，访问所有的导入和隐式内容
- 创造新的代码表达式，产生编译错误（如果有的话）并将它们注入编译链

请注意：

- 我们使用 Scala 宏，因为它正好满足我们的需求
- 我们使用 Scala 宏作为推动者，而不是目的本身
- 宏是一个帮助你生成代码的 helper，这样你就不用自己写这部分代码
- 它并没有增加或隐藏代码
- 我们遵循的是 no-surprise 原则

你可能发现了，写宏并不是一个简单的过程，因为你写的宏是在编译期执行的。

```
So you write macro code
  that is compiled and executed
  in a runtime that manipulates your code
     to be compiled and executed
     in a future runtime…
```

这也是为什么我把它叫做 Inception。

因此想完全按照你想做的来，来需要一些练习。提供的 API 也相当复杂，文档也不齐全。因此，当你开始使用宏时，你需要有坚持不懈的精神。

Writes[T] & Format[T]

> 请注意，*JSON inception* 只能用于含有 *unapply/apply* 方法的结构。

自然地，你可将它用于 `Writes[T]` 和 `Format[T]` 。

Writes[T]

```
import play.api.libs.json._
```

```
2.
3.  implicit val personWrites = Json.writes[Person]
```

Format[T]

```
1.  import play.api.libs.json._
2.
3.  implicit val personWrites = Json.format[Person]
```

特殊模式

- 你可以在伴生对象（companion object）中定义你的 Reads/Writes

这样当你操作你的类的一个实例，隐式的 Reads/Writes 就会被自动推断出来。

```
1.  import play.api.libs.json._
2.
3.  case class Person(name: String, age: Int)
4.
5.  object Person{
6.    implicit val personFmt = Json.format[Person]
7.  }
```

- 你现在可以为单字段样例类定义 Reads/Writes

```
1.  import play.api.libs.json._
2.
3.  case class Person(names: List[String])
4.
5.  object Person{
6.    implicit val personFmt = Json.format[Person]
7.  }
```

已知限制

- 不要覆盖伴生对象中的 apply 函数，因为这样一来宏就有几个 apply 函数，而不知道选择哪个。
- 只有当 apply 和 unapply 方法有相应的输入输出类型时，才能用 Json 宏。这对样例类来说是很自然的，但如果你想要特性（trait）也能达到一样的效果，你就需要实现与样例类中相同的 apply/unapply 方法。
- Json 宏可用于以下类型: Option/Seq/List/Set/Map[String,_]，如果想用于其它类型，你需要进行测试，如果不行，请使用传统方式手写 Reads/Writes。

Working with XML

处理XML请求和提供XML响应

处理XML请求

XML请求是指HTTP请求将有效的XML格式的内容作为请求体，该HTTP请求必须在 `Content-Type` 中指定它的MIME类型为 `application/xml` 或者 `text/xml` 。

一般来说， `Action` 使用任意内容解析器，这使得程序可以接受XML格式的请求体（实际上是作为 `NodeSeq` 类处理）：

```
def sayHello = Action { request =>
  request.body.asXml.map { xml =>
    (xml \\ "name" headOption).map(_.text).map { name =>
      Ok("Hello " + name)
    }.getOrElse {
      BadRequest("Missing parameter [name]")
    }
  }.getOrElse {
    BadRequest("Expecting Xml data")
  }
}
```

更好（也更简单）的方法是明确给定我们要用的 `BodyParser` ，让Play框架直接将请求体内容当作XML格式解析：

```
def sayHello = Action(parse.xml) { request =>
```

```
  (request.body \\ "name" headOption).map(_.text).map { name =>
    Ok("Hello " + name)
  }.getOrElse {
    BadRequest("Missing parameter [name]")
  }
}
```

```
注意：在使用XML内容解析器时，request.body本身就是一个有效的NodeSeq实例
```

在命令行中可以使用**cURL**测试XML请求处理接口：

```
curl
  --header "Content-type: application/xml"
  --request POST
  --data '<name>Guillaume</name>'
  http://localhost:9000/sayHello
```

返回的内容是：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 15

Hello Guillaume
```

提供XML响应

在之前的例子中，我们展示了如何使用Play框架处理XML请求，但是响应的格式是 `text/plain` 。下面的代码展示了如何返回一个XML格式的HTTP响应。

```
def sayHello = Action(parse.xml) { request =>
  (request.body \\ "name" headOption).map(_.text).map { name =>
    Ok(<message status="OK">Hello {name}</message>)
  }.getOrElse {
```

```
5.      BadRequest(<message status="KO">Missing parameter [name]
    </message>)
6.    }
7.  }
```

然后还是使用**cURL**命令测试，返回的内容如下：

```
1.  HTTP/1.1 200 OK
2.  Content-Type: application/xml; charset=utf-8
3.  Content-Length: 46
4.
5.  <message status="OK">Hello Guillaume</message>
```

Handling file upload

Direct upload and multipart/form-data

在表单中使用multipart/form-data上传文件

在Web应用中上传文件的标准方法是使用以 `multipart/form-data` 编码的表单，这样能让附件数据与标准表单数据混合在一起。请注意：提交表单时只能用POST方法而不能用GET。

首先构建一个HTML表单：

```
@helper.form(action = routes.Application.upload, 'enctype -> "multipart/form-data") {
    <input type="file" name="picture">
    <p>
        <input type="submit">
    </p>
}
```

然后用利用 `multipartFormData` body parser来定义一个upload动作：

```
def upload = Action(parse.multipartFormData) { request =>
  request.body.file("picture").map { picture =>
    import java.io.File
    val filename = picture.filename
    val contentType = picture.contentType
```

```
6.      picture.ref.moveTo(new File(s"/tmp/picture/$filename"))
7.      Ok("File uploaded")
8.    }.getOrElse {
9.      Redirect(routes.Application.index).flashing(
10.       "error" -> "Missing file")
11.   }
12. }
```

`ref` 属性用于对 `TemporaryFile` 进行说明。这是 `mutipartFormData` 提取器处理文件上传的默认方式。

注意：你也可以使用 `anyContent` body parser，将其作为 `request.body.asMultipartFormData` 来检索。

最后，添加一个POST路由：

```
1. POST  /          controllers.Application.upload()
```

直接文件上传

另一种上传文件的方法是在表单中使用Ajax异步上传。在这种情况下请求内容不再是 `multipart/form-data` 编码了，而仅有包含文件内容本身。

这时我们可以使用一个body parser将请求内容存入文件。比如， 我们使用 `temporaryFile` body parser：

```
1. def upload = Action(parse.temporaryFile) { request =>
2.   request.body.moveTo(new File("/tmp/picture/uploaded"))
3.   Ok("File uploaded")
4. }
```

自定义body parser

如果你不想经过临时文件缓存而是直接处理上传的文件，你可以自己写一个 `BodyParser` 来决定怎么处理这些大块的数据。

如果你想使用 `multipart/form-data` 编码，你仍可以使用默认的 `mutipartFormData` 提取器，你要做的就是自己写一个 `PartHandler[FilePart[A]]` 来接收部分 headers，然后提供一个 `Iteratee[Array[Byte]` , `FilePart[A]]` 来生成正确的 `FilePart` 。

Using the Cache

The Play cache API

Play 缓存 API

使用EHCache为缓存API的默认实现.你同时可以通过一个插件来提供你自己的实现。

载入缓存API

Add cache into your dependencies list. For example, in build.sbt:
将cache加入到你的关联列表中。例如，在build.sbt中:

```
1.  libraryDependencies ++= Seq(
2.        cache,
3.        ...
4.     )
```

访问缓存API

play.api.cache.Cache对象提供了缓存API.其需要一个注册的缓

存插件。

```
1.  注意： 该API有意的最小化允许多种实现可被嵌入。如果你需要一个更具体的API，使用你的缓存插件提供的那一个。
```

使用这个简单的API你既可以存储数据于缓存中：

```
1.  Cache.set("item.key", connectedUser)
```

并在不久后获取它：

```
1.  val maybeUser: Option[User] = Cache.getAs[User]("item.key")
```

当其不存在时候这里还有一个简洁的帮助器去从缓存中获取或在缓存中设定值：

```
1.  val user: User = Cache.getOrElse[User]("item.key") {
2.        User.findById(connectedUser)
3.  }
```

To remove an item from the cache use the remove method:
使用remove方法去从缓存中移除一个条目：

```
1.  Cache.remove("item.key")
```

缓存HTTP回应

你可以使用标准的Action组合。

```
1.  注意： Play HTTP 结果实例对缓存是安全的并可以之后从用。
```

Play为默认的例子提供了一个默认的嵌入帮助器：

```
def index = Cached("homePage") {
      Action {
        Ok("Hello world")
          }
}
```

甚至于：

```
def userProfile = Authenticated {
      user =>
    Cached(req => "profile." + user) {
          Action {
            Ok(views.html.profile(User.find(user)))
                }
        }
}
```

缓存控制

你可以简单的控制什么是你想缓存的或者什么是你不想缓存的。

你可能仅需要结果为“缓存200 OK”

```
def get(index: Int) = Cached.status(_ => "/resource/"+ index, 200)
{
      Action {
    if (index > 0) {
          Ok(Json.obj("id" -> index))
        } else {
              NotFound
        }
      }
    }
```

或者“缓存404未找到”的信息仅存在几分钟

```
1.  def get(index: Int) = {
2.        val caching = Cached
3.          .status(_ => "/resource/"+ index, 200)
4.          .includeStatus(404, 600)
5.
6.        caching {
7.          Action {
8.            if (index % 2 == 1) {
9.              Ok(Json.obj("id" -> index))
10.           } else {
11.             NotFound
12.                   }
13.         }
14.       }
15. }
```

Calling WebServices

The Play WS API

有的时候我们需要在 Play 应用中调用其它 HTTP 服务。Play 通过

WS 库来支持这一需求，它提供了一种方法来做异步 HTTP 调用。

使用 WS API 包含两个重要部分：构造请求以及处理响应。我们会先讨论如何构造 GET 和 POST 的 HTTP 请求，然后展示如何处理返回的响应。最后，我们会讨论一些常见用例。

构造请求

想要使用 WS，首先需要在你的 `build.sbt` 文件中加上 `ws` 库：

```
1. libraryDependencies ++= Seq(
2.   ws
3. )
```

然后导入以下内容：

```
1. import play.api.Play.current
2. import play.api.libs.ws._
3. import play.api.libs.ws.ning.NingAsyncHttpClientConfigBuilder
4. import scala.concurrent.Future
```

想要构建一个 HTTP 请求，你首先要用 `WS.url()` 来指定 URL：

```
1. val holder: WSRequestHolder = WS.url(url)
```

上面的语句返回一个类型为 `WSRequestHolder` 的值，你可以通过它指定各种 HTTP 选项，例如设置各种报头。你可以使用链式调用来构造复杂的请求。

```
1. val complexHolder: WSRequestHolder =
2.   holder.withHeaders("Accept" -> "application/json")
3.     .withRequestTimeout(10000)
4.     .withQueryString("search" -> "play")
```

最后你通过调用一个与 HTTP 方法对应的方法来发出请求，它会使用你之前设置的所有选项。

```
1.  val futureResponse: Future[WSResponse] = complexHolder.get()
```

上述调用返回的类型是 `Future[WSResponse]`，返回的响应包含了服务器传来的数据。

带验证的请求

如果你需要使用 HTTP 验证，你可以要构造请求的过程中指定它，使用用户名、密码以及 AuthScheme。AuthScheme 的有效样例对象（case objects）有：`BASIC`，`DIGEST`，`KERBEROS`，`NONE`，`NTLM` 和 `SPNEGO`。

```
1.  WS.url(url).withAuth(user, password, WSAuthScheme.BASIC).get()
```

带重定向的请求

如果一个 HTTP 请求的结果是 302 或 301 重定向，你可以自动执行重定向而无需另一次的调用。

```
1.  WS.url(url).withFollowRedirects(true).get()
```

带查询参数的请求

参数可以被指定为一系列的 K/V 元组。

```
1.  WS.url(url).withQueryString("paramKey" -> "paramValue").get()
```

带额外报头的请求

报头也可以被指定为一系列的 K/V 元组。

```
1.  WS.url(url).withHeaders("headerKey" -> "headerValue").get()
```

如果你想以特殊格式发送纯文本，你需要显式地定义内容类型：

```
1.  WS.url(url).withHeaders("Content-Type" ->
    "application/xml").post(xmlString)
```

带虚拟主机的请求

你可以用一个字符串来指定一个虚拟主机：

```
1.  WS.url(url).withVirtualHost("192.168.1.1").get()
```

带超时时间的请求

如果你想指定请求的超时时间，你可以使用 `withRequestTimeout` 来设置一个单位为毫秒的值：

```
1.  WS.url(url).withRequestTimeout(5000).get()
```

提交表单数据

想要 post 一个 url-form-encoded 的数据，你需要给 `post` 方法传一个类型为 `Map[String, Seq[String]]` 的值。

```
1.  WS.url(url).post(Map("key" -> Seq("value")))
```

提交 JSON 数据

post Json 数据最简单的方法就是使用 JSON 库。

```
1. import play.api.libs.json._
2. val data = Json.obj(
3.   "key1" -> "value1",
4.   "key2" -> "value2"
5. )
6. val futureResponse: Future[WSResponse] = WS.url(url).post(data)
```

提交 XML 数据

post XML 数据最简单的方法是使用 XML 字面量。XML 字面量用起来很方便，但速度不快。追求效率的话，请考虑使用 XML 视图模板或 JAXB 库。

```
1. val data = <person>
2.   <name>Steve</name>
3.   <age>23</age>
4. </person>
5. val futureResponse: Future[WSResponse] = WS.url(url).post(data)
```

处理响应

处理 Response 可以通过在 Future 内做映射来简单完成。

下面给出的例子都有一些共同的依赖，在这里先简单地说明一下。

任何时候 `Future` 上的一个操作，都是需要一个隐式的执行上下文的，这一点声明了回调应该运行在哪个线程池。通常，Play 默认的执行上下文就足够了：

```
1. implicit val context =
   play.api.libs.concurrent.Execution.Implicits.defaultContext
```

下面的例子还使用了以下样例类来进行序列化与反序列化：

```
1.  case class Person(name: String, age: Int)
```

处理 JSON 响应

你可以通过调用 `response.json` 将响应当作 JSON 对象来处理。

```
1.  val futureResult: Future[String] = WS.url(url).get().map {
2.    response =>
3.      (response.json \ "person" \ "name").as[String]
4.  }
```

Play 的 JSON 库还有一个有用的特性，可以将一个隐式的 `Reads[T]` 直接映射成一个类：

```
1.  import play.api.libs.json._
2.
3.  implicit val personReads = Json.reads[Person]
4.
5.  val futureResult: Future[JsResult[Person]] = WS.url(url).get().map
    {
6.    response => (response.json \ "person").validate[Person]
7.  }
```

处理 XML 响应

你可以通过调用 `response.xml` 将响应当作 XML 字面量来处理。

```
1.  val futureResult: Future[scala.xml.NodeSeq] = WS.url(url).get().map
    {
2.    response =>
3.      response.xml \ "message"
4.  }
```

处理大块响应

调用 `get()` 或 `post()` 方法会导致一个问题，就是只有当响应体完全载入到内存中，响应才可用。当你在下载几个 G 的大文件时，这可能会导致另人不爽的 GC，甚至出现内存不足的错误。

`WS` 可以让你通过 `iteratee` 来增量地使用响应。`WSRequestHolder` 的 `stream()` 和 `getStream()` 方法返回一个 `Future[(WSResponseHeaders, Enumerator[Array[Byte]])]`。其中，枚举器包含了响应体。

下面是一个常见的例子，使用 `iteratee` 计算响应返回的字节数：

```
1.  import play.api.libs.iteratee._
2.
3.  // Make the request
4.  val futureResponse: Future[(WSResponseHeaders,
    Enumerator[Array[Byte]])] =
5.    WS.url(url).getStream()
6.
7.  val bytesReturned: Future[Long] = futureResponse.flatMap {
8.    case (headers, body) =>
9.      // Count the number of bytes returned
10.     body |>>> Iteratee.fold(0l) { (total, bytes) =>
11.       total + bytes.length
12.     }
13. }
```

当然，通常情况下你不会像上面那样只是计算数据的字节数，更常见的情况是把响应返回的数据写到另一个地方去，比如写入文件：

```
1.  import play.api.libs.iteratee._
2.
3.  // Make the request
4.  val futureResponse: Future[(WSResponseHeaders,
```

```
    Enumerator[Array[Byte]])] =
  WS.url(url).getStream()

val downloadedFile: Future[File] = futureResponse.flatMap {
  case (headers, body) =>
    val outputStream = new FileOutputStream(file)

    // The iteratee that writes to the output stream
    val iteratee = Iteratee.foreach[Array[Byte]] { bytes =>
      outputStream.write(bytes)
    }

    // Feed the body into the iteratee
    (body |>>> iteratee).andThen {
      case result =>
        // Close the output stream whether there was an error or
    not
        outputStream.close()
        // Get the result or rethrow the error
        result.get
    }.map(_ => file)
}
```

另一种情况是，当前服务器把拿到的响应体流式写入另一个响应中，返回给它所服务的对象：

```
def downloadFile = Action.async {

  // Make the request
  WS.url(url).getStream().map {
    case (response, body) =>

      // Check that the response was successful
      if (response.status == 200) {

        // Get the content type
        val contentType = response.headers.get("Content-
```

```
    Type").flatMap(_.headOption)
12.           .getOrElse("application/octet-stream")
13.
14.         // If there's a content length, send that, otherwise return
    the body chunked
15.         response.headers.get("Content-Length") match {
16.           case Some(Seq(length)) =>
17.             Ok.feed(body).as(contentType).withHeaders("Content-
    Length" -> length)
18.           case _ =>
19.             Ok.chunked(body).as(contentType)
20.         }
21.       } else {
22.         BadGateway
23.       }
24.   }
25. }
```

`POST` 和 `PUT` 的调用需要手动调用 `withMethod` 方法：

```
1.  val futureResponse: Future[(WSResponseHeaders,
    Enumerator[Array[Byte]])] =
2.    WS.url(url).withMethod("PUT").withBody("some body").stream()
```

常见模式和用例

链式 WS 调用

使用 for 解析是一种链接 WS 调用的好方式。for 解析应该和 Future.recover 配合使用，用于处理可能的失败。

```
1.  val futureResponse: Future[WSResponse] = for {
2.    responseOne <- WS.url(urlOne).get()
3.    responseTwo <- WS.url(responseOne.body).get()
4.    responseThree <- WS.url(responseTwo.body).get()
5.  } yield responseThree
```

```
6.
7.  futureResponse.recover {
8.    case e: Exception =>
9.      val exceptionData = Map("error" -> Seq(e.getMessage))
10.     WS.url(exceptionUrl).post(exceptionData)
11. }
```

在控制器中使用

当在控制器中构建请求时，你可以将响应映射为 `Future[Result]`。这可以与 Play 的 `Action.async` 结合使用，详见：Handling Asynchronous Results。

```
1.  def wsAction = Action.async {
2.    WS.url(url).get().map { response =>
3.      Ok(response.body)
4.    }
5.  }
6.  status(wsAction(FakeRequest())) must_== OK
```

使用 WSClient

WSClient 是底层 AsyncHttpClient 的 wrapper。有时候你需要定义多个客户端，建议使用不同的配置文件，或使用模拟的方式。

默认的客户端可以由 WS 单例调用：

```
1.  val client: WSClient = WS.client
```

你可以直接从代码中定义一个 WS 客户端，通过 `WS.clientUrl()` 隐式地使用。注意，当你配置你的客户端时，你应该使用 `NingAsyncHttpClientConfigBuilder` 来做 TLS 配置：

```
1.  val clientConfig = new DefaultWSClientConfig()
```

```
2.  val secureDefaults:com.ning.http.client.AsyncHttpClientConfig = new
    NingAsyncHttpClientConfigBuilder(clientConfig).build()
3.  // You can directly use the builder for specific options once you
    have secure TLS defaults...
4.  val builder = new
    com.ning.http.client.AsyncHttpClientConfig.Builder(secureDefaults)
5.  builder.setCompressionEnabled(true)
6.  val
    secureDefaultsWithSpecificOptions:com.ning.http.client.AsyncHttpClien
     = builder.build()
7.  implicit val implicitClient = new
    play.api.libs.ws.ning.NingWSClient(secureDefaultsWithSpecificOptions)
8.  val response = WS.clientUrl(url).get()
```

> 注意：如果你实例化一个 *NingWSClient* 对象，它并不会使用 *WS* 插件系统，因此它不会在 `Application.onStop` 里自动关闭。当处理结束时，你需要使用 `client.close()` 来手动关闭客户端。这会释放 *AsyncHttpClient* 使用的底层 *ThreadPoolExecutor*。客户端关闭失败可能会导致内存不足的异常（尤其在开发模式下，需要经常性重新加载应用的时候）。

也可以像下面直接使用：

```
1.  val response = client.url(url).get()
```

或使用磁铁模式（magnet pattern）来自动匹配合适的客户端：

```
1.  object PairMagnet {
2.    implicit def fromPair(pair: (WSClient, java.net.URL)) =
3.      new WSRequestHolderMagnet {
4.        def apply(): WSRequestHolder = {
5.          val (client, netUrl) = pair
6.          client.url(netUrl.toString)
7.        }
8.      }
9.  }
10.
11. import scala.language.implicitConversions
```

```
12. import PairMagnet._
13.
14. val client = WS.client
15. val exampleURL = new java.net.URL(url)
16. val response = WS.url(client -> exampleURL).get()
```

默认情况下，配置一般写在 `application.conf` 里，但你也可以直接在代码中设置配置：

```
1.  import com.typesafe.config.ConfigFactory
2.  import play.api.libs.ws._
3.  import play.api.libs.ws.ning._
4.
5.  val configuration =
    play.api.Configuration(ConfigFactory.parseString(
6.    """
7.      |ws.followRedirects = true
8.    """.stripMargin))
9.
10. val classLoader = app.classloader // Play.current.classloader or
    other
11. val parser = new DefaultWSConfigParser(configuration, classLoader)
12. val builder = new NingAsyncHttpClientConfigBuilder(parser.parse())
```

你也可以直接访问底层的 `async client`。

```
1. import com.ning.http.client.AsyncHttpClient
2.
3. val client: AsyncHttpClient = WS.client.underlying
```

可以直接访问底层类是非常重要的，因为 `WS` 在有的时候会有一些限制：

- `WS` 并不直接支持 `multi-part-form` 数据的上传。你可以使用底层客户端的 `RequestBuilder.addBodyPart` 来做。

- `WS` 不支持流式数据上传。在这种情况下，你应该使用 AsyncHttpClient 提供的 `FeedableBodyGenerator`。

配置 WS

在 `application.conf` 文件中使用如下属性来配置 WS 客户端：

- `ws.followRedirects`：配置客户端做 301、302 重定向（默认为 true）。
- `ws.useProxyProperties`：使用系统的 http 代理设置（http.proxyHost，http.proxyPort）（默认为 true）。
- `ws.useragent`：配置 User-Agent 报头。
- `ws.compressionEnabled`：如果使用 gzip/deflater 编码，则将它设置为 true（默认为 false）。

用 SSL 配置 WS

想要配置 WS 在 SSL/TLS（HTTPS）之上使用 HTTP，请移步：配置 WS SSL

配置超时时间

WS 中有 3 种不同的超时。超时会导致 WS 请求中断。

- `ws.timeout.connection`：连接远程主机的最大等待时间（默认是 120 秒）。
- `ws.timeout.idle`：请求保持空闲状态的最大时间（此时连接已经建立，在等待更多的数据）（默认是 120 秒）。
- `ws.timeout.request`：你能允许的请求使用的总时间（当达到这个时间时，请求就会中断，哪怕远程主机仍然在发送数据）（默认是 none，为的是可以处理流式数据）。

在一个具体的连接中，你可以用 `withRequestTimeout()` 来覆盖配置文件中的请求超时设置（详见「构造请求」一节）。

Connecting to OpenID services

Connecting to OpenID services

OpenID 是一种使用一个账号访问多个服务的协议。作为一个 web 开发者，你可以使用 OpenID 让用户用他们已有的账号（例如 Google 账号）来进行登录。在企业中，你可以使用 OpenID 来连接公司的 SSO 服务器。

OpenID 工作流程一览

- 用户给你他的 OpenID（一个 URL）。
- 你的服务器检查 URL 后面的内容，然后产生一个 URL，并将用户重定向到那。
- 用户在他的 OpenID 提供方那确认授权，然后重定向回你的服务器。
- 你的服务器收到重定向返回的信息，并与 OpenID 提供方检查信息是否正确。

如果你的用户使用的都是同一个 OpenID 提供方，那么可以忽略第一步（例如你决定全部都使用 Google 账号来登录）。

用法

使用 OpenID，首先需要把 `ws` 库添加到你的 `build.sbt` 文件中：

```
1. libraryDependencies ++= Seq(
2.   ws
3. )
```

在 Play 中使用 OpenID

OpenID API 有两个重要的函数：

- `OpenID.redirectURL` 计算用户需要重定向的 URL，这个过程包括异步地获取用户的 OpenID 页面，这就是为什么它返回的是 `Future[String]`。如果 OpenID 是无效的，返回的 `Future` 就会失败。
- `OpenID.verifiedId` 需要一个隐式的 `Request`，并且检查它来建立用户的信息，包含他验证过的 OpenID。它会异步地请求一下 OpenID 服务器以确认信息的真实性，这也是为什么它返回的是一个 `Future[UserInfo]`。如果信息不正确或者服务器的检查结果不正确（例如重定向 URL 是伪造的），返回的 `Future` 就会失败。

如果 `Future` 失败，你可以定义一个回退将用户重定向回登录页面或者返回 `BadRequest`。

下面请看一个用例：

```
1. def login = Action {
2.   Ok(views.html.login())
3. }
4.
5. def loginPost = Action.async { implicit request =>
6.   Form(single(
7.     "openid" -> nonEmptyText
```

```
  )).bindFromRequest.fold(
    { error =>
      Logger.info("bad request " + error.toString)
      Future.successful(BadRequest(error.toString))
    },
    { openId =>
      OpenID.redirectURL(openId,
routes.Application.openIDCallback.absoluteURL())
        .map(url => Redirect(url))
        .recover { case t: Throwable =>
Redirect(routes.Application.login) }
    }
  )
}

def openIDCallback = Action.async { implicit request =>
  OpenID.verifiedId.map(info => Ok(info.id + "\n" +
info.attributes))
    .recover {
      case t: Throwable =>
      // Here you should look at the error, and give feedback to
the user
      Redirect(routes.Application.login)
    }
}
```

扩展属性

用户的 OpenID 给你的是他的身份标识，该协议也支持获取一些扩展属性，如 Email 地址，名或者姓。

你可以从 OpenID 服务器请求一些可选属性或必需属性。要求提供某些必需属性意味着如果用户没有提供的话，他将无法登录你的服务。

你在重定向 URL 中请求扩展属性：

```
OpenID.redirectURL(
    openid,
    routes.Application.openIDCallback.absoluteURL(),
    Seq("email" -> "http://schema.openid.net/contact/email")
)
```

这样一来，OpenID 服务器提供的 `UserInfo` 中就会有这个属性了。

Accessing resources protected by OAuth

Accessing resources protected by OAuth

OAuth 是发布受保护数据并与之交互的一种简单方式。它是授与你访问权的更安全可靠的方式。例如，你可以通过它访问你在 Twitter 上的用户数据。

OAuth 有 2 个非常不同的版本：OAuth 1.0 和 OAuth 2.0。版本 2 非常简单，无需库和 helper 的支持。因此 Play 只提供 OAuth 1.0 的支持。

用法

想用 OAuth，首先需要把 `ws` 库添加到你的 `build.sbt` 文件中：

```
1.  libraryDependencies ++= Seq(
2.    ws
3.  )
```

需要的信息

OAuth 需要你将你的应用注册到服务提供方。确保检查了你提供的回

调 URL，因为如果回调 URL 不匹配的话，服务提供方可能会拒绝你的调用。本地使用时，你可以在 `/etc/hosts` 中为本机伪造一个域名。

服务提供方会给你：

- 应用 ID
- 密钥
- 请求令牌 URL（Request Token URL）
- 访问令牌 URL（Access Token URL）
- 授权 URL

验证流程

大部分事情都由 Play 的库完成。

- 从服务器获取请求令牌（服务器之间的调用）。
- 将用户重定向到服务提供方，在那里他会授权你的应用可以使用他的数据。
- 服务提供方会将用户重定向回去，并给你一个检验器。
- 有了检验器，你就可以将请求令牌换成访问令牌（服务器之间的调用）。

现在你可以用访问令牌去访问受保护的用户数据了。

例子

```
1. object Twitter extends Controller {
2.
3.   val KEY = ConsumerKey("xxxxx", "xxxxx")
4.
5.   val TWITTER = OAuth(ServiceInfo(
```

```
6.      "https://api.twitter.com/oauth/request_token",
7.      "https://api.twitter.com/oauth/access_token",
8.      "https://api.twitter.com/oauth/authorize", KEY),
9.      true)
10.
11.   def authenticate = Action { request =>
12.     request.getQueryString("oauth_verifier").map { verifier =>
13.       val tokenPair = sessionTokenPair(request).get
14.       // We got the verifier; now get the access token, store it
    and back to index
15.       TWITTER.retrieveAccessToken(tokenPair, verifier) match {
16.         case Right(t) => {
17.           // We received the authorized tokens in the OAuth object
    - store it before we proceed
18.           Redirect(routes.Application.index).withSession("token" ->
    t.token, "secret" -> t.secret)
19.         }
20.         case Left(e) => throw e
21.       }
22.     }.getOrElse(
23.       TWITTER.retrieveRequestToken("http://localhost:9000/auth")
    match {
24.         case Right(t) => {
25.           // We received the unauthorized tokens in the OAuth
    object - store it before we proceed
26.
    Redirect(TWITTER.redirectUrl(t.token)).withSession("token" ->
    t.token, "secret" -> t.secret)
27.         }
28.         case Left(e) => throw e
29.       })
30.   }
31.
32.   def sessionTokenPair(implicit request: RequestHeader):
    Option[RequestToken] = {
33.     for {
34.       token <- request.session.get("token")
35.       secret <- request.session.get("secret")
```

```
    } yield {
      RequestToken(token, secret)
    }
  }
}
```

```
object Application extends Controller {

  def timeline = Action.async { implicit request =>
    Twitter.sessionTokenPair match {
      case Some(credentials) => {

WS.url("https://api.twitter.com/1.1/statuses/home_timeline.json")
          .sign(OAuthCalculator(Twitter.KEY, credentials))
          .get
          .map(result => Ok(result.json))
      }
      case _ =>
Future.successful(Redirect(routes.Twitter.authenticate))
    }
  }
}
```

Integrating with Akka

Setting up Actors and scheduling asynchronous tasks

Akka 使用 Actor 模型来提高抽象级别，它提供了一个更好的平台来构建正确的、并发的和可扩展的应用。在错误容忍度（fault-tolerance）上，它采用的是 ‘Let it crash’ 模型，该模型被成功地应用于电信业，用于构建永不停歇的自愈系统。Actor 提供了对透明分布的抽象，以及构建真正可扩展和高错误容忍度应用的基础。

应用的 actor 系统

Akka 可以和一些叫做 `ActorSystem` 的容器一起工作。Actor 系统管理着它所配置的资源，这样它可以运行它所包含的 actor。

一个 Play 应用会定义一个特殊的 actor 系统来给该应用使用，这个 actor 系统贯穿了整个应用的生命周期，并且在应用重启时它会自动重启。

> 注意：在 *Play* 应用中，你也可以使用另外一个 *actor* 系统。默认提供的 *actor* 系统，只是在你想启动少量 *actor* 而又不想麻烦地去配置你自己的 *actor* 系统时，提供一些方便。

你可以通过 `play.api.libs.concurrent.Akka` 中的 helper 方法来使用默认的 actor 系统：

```
1.  val myActor = Akka.system.actorOf(Props[MyActor], name = "myactor")
```

配置

默认 actor 系统的配置是从 Play 应用的配置文件中读取出来的。例如，想为应用的 actor 系统配置默认的调度器，需要在配置文件 `conf/application.conf` 中加入以下两行：

```
1.  akka.default-dispatcher.fork-join-executor.pool-size-max =64
2.  akka.actor.debug.receive = on
```

> 注意：你可以在相同的文件中配置其它任意的 *actor* 系统，记得提供一个顶层的配置键即可。

调度异步任务

你可以定时发送消息给 actor 并执行相应的任务（函数或 `Runnable`），然后你会收到一个 `Cancellable`，你可以调用 `cancel` 来取消相应操作的执行。

例如，每 300 毫秒发送一条消息给 `testActor`：

```
1.  import play.api.libs.concurrent.Execution.Implicits._
2.  Akka.system.scheduler.schedule(0.microsecond, 300.microsecond,
    testActor, "tick")
```

> 注意：上面的例子使用了定义在 *scala.concurrent.duration* 中的隐式转换，来将数字转换成不同时间单位的 *Duration* 对象。

相似地，下面的例子展示的是 1 秒后执行一个代码块：

```
1.  import play.api.libs.concurrent.Execution.Implicits._
2.  Akka.system.scheduler.scheduleOnce(1000.microsecond) {
3.    file.delete()
```

```
4. }
```

Internationalization

Messages externalisation and i18n

指定应用支持语言

你可以按 **ISO 639-2** 标准语言代码或者 **ISO 3166-1 alpha-2** 标准国家代码来指定应用支持语言，譬如 `fr` 或者 `en-US`。

语言指定信息存放在应用的 `conf/application.conf` 文件中：

```
1.  application.langs="en,en-US,fr"
```

外部化信息（Externalizing messages）

你可以在 `conf/messages.xxx` 文件中外部化（externalize）信息。

`conf/messages` 文件默认适配所有语言。你可以添加你所指定语言所对应的文件，譬如 `conf/messages.fr` 或 `conf/messages.en-US`。

你可以利用 `play.api.i18n.Messages` 对象检出(retrieve)信息：

```
1.  val title = Messages("home.title")
```

所有国际化相关的API都隐式调用 `play.api.i18n.Lang` 参数，其中参数的值根据当前环境给出。你可以自己显式地指定：

```
1.  val title = Messages("home.title")(Lang("fr"))
```

注意：如果你在当前范围发出一个隐式请求，程序会根据头部的 `Accept-Language` 从应用支持语言匹配一个，然后隐式地返回给你一个 `Lang` 参数。你应当像这样在模板文件中添加 `Lang` 变量： `@()(implicit lang: Lang)` 。

消息格式

消息使用 `java.text.MessageFormat` 库来定义格式，譬如你的消息这样定义：

```
1.  files.summary=The disk {1} contains {0} file(s).
```

那么你可以这样指定参数：

```
1.  Messages("files.summary", d.files.length, d.name)
```

关于单引号

由于我们使用 `java.text.MessageFormat` 来定义消息格式，所以要注意的是单引号会被作为转意字符。

譬如你定义了如下一段消息：

```
1.  info.error=You aren''t logged in!
```

```
1.  example.formatting=When using MessageFormat, '''{0}''' is replaced
    with the first parameter.
```

你应期望如下结果：

```
1. Messages("info.error") == "You aren't logged in!"
```

```
1. Messages("example.formatting") == "When using MessageFormat, '{0}'
   is replaced with the first parameter."
```

从 HTTP 请求中提取支持语言

你可以（像这样）从给定的HTTP请求提取出支持语言：

```
1. def index = Action { request =>
2.   Ok("Languages: " +
   request.acceptLanguages.map(_.code).mkString(", "))
3. }
```

The application Global object

应用全局设置

Global 对象

你可以在项目中定义一个 `Global` 对象来控制应用的全局设置。这个对象必须在默认（空）包中定义并扩展自 `GlobalSettings`。

```
1. import play.api._
2.
3. object Global extends GlobalSettings {
4.
5. }
```

> 提示：你也可以在 *`application.global`* 设置字段中指定一个自定义 *`GlobalSettings`* 实现的类名。

关联应用启动和停止事件

你可以重写 `onStart` 和 `onStop` 方法来获取应用生命周期中的事件通知：

```
1. import play.api._
2.
3. object Global extends GlobalSettings {
```

```
4.
5.   override def onStart(app: Application) {
6.     Logger.info("Application has started")
7.   }
8.
9.   override def onStop(app: Application) {
10.    Logger.info("Application shutdown...")
11.  }
12.
13. }
```

提供应用错误讯息页

当你的应用出现一个异常，这将会引发 `onError` 操作。默认使用内部框架的错误讯息页：

```
1.  import play.api._
2.  import play.api.mvc._
3.  import play.api.mvc.Results._
4.  import scala.concurrent.Future
5.
6.  object Global extends GlobalSettings {
7.
8.    override def onError(request: RequestHeader, ex: Throwable) = {
9.      Future.successful(InternalServerError(
10.       views.html.errorPage(ex)
11.     ))
12.   }
13.
14. }
```

处理丢失的动作和绑定错误

如果框架无法为请求找到一个 `Action`，这将会引发 `onHandlerNotFound` 操作：

```
import play.api._
import play.api.mvc._
import play.api.mvc.Results._
import scala.concurrent.Future

object Global extends GlobalSettings {

  override def onHandlerNotFound(request: RequestHeader) = {
    Future.successful(NotFound(
      views.html.notFoundPage(request.path)
    ))
  }

}
```

如果获取的路由无法绑定请求的参数，这将会引发 `onBadRequest` 操作：

```
import play.api._
import play.api.mvc._
import play.api.mvc.Results._
import scala.concurrent.Future

object Global extends GlobalSettings {

  override def onBadRequest(request: RequestHeader, error: String)
= {
    Future.successful(BadRequest("Bad Request: " + error))
  }

}
```

Intercepting requests

侦听请求

使用过滤器

你可以使用过滤组件侦听来自应用的请求，转换请求和响应。过滤器能很好地实现你的应用中的横切关注点（Cross-cutting concern）。你可以通过扩展 `Filter` 特质来创建一个过滤器，然后将其加入 `Global` 对象。
以下示例创建了一个记录所有动作结果的访问记录过滤器：

```
1.  import play.api.Logger
2.  import play.api.mvc._
3.
4.  object AccessLoggingFilter extends Filter {
5.    def apply(next: (RequestHeader) => Future[Result])(request:
    RequestHeader): Future[Result] = {
6.      val result = next(request)
7.      Logger.info(request + "\n\t => " + result)
8.      result
9.    }
10. }
11.
12. object Global extends WithFilters(AccessLoggingFilter)
```

扩展了 `WithFilters` 类的 `Global` 对象让你可以通过传递多个过滤器来组成过滤链。

> 注意: `WithFilters` 现在扩展了 `GlobalSettings` 特质。

另一个体现过滤器用处的示例，在调用某个动作前检查授权：

```
object AuthorizedFilter {
  def apply(actionNames: String*) = new AuthorizedFilter(actionNames)
}

class AuthorizedFilter(actionNames: Seq[String]) extends Filter {

  def apply(next: (RequestHeader) => Future[Result])(request: RequestHeader): Future[Result] = {
    if(authorizationRequired(request)) {
      /* do the auth stuff here */
      println("auth required")
      next(request)
    }
    else next(request)
  }

  private def authorizationRequired(request: RequestHeader) = {
    val actionInvoked: String = request.tags.getOrElse(play.api.Routes.ROUTE_ACTION_METHOD, "")
    actionNames.contains(actionInvoked)
  }


}


object Global extends WithFilters(AuthorizedFilter("editProfile", "buy", "sell")) with GlobalSettings {}
```

> 提示: `RequestHeader.tags` 提供了大量关于调用动作的路由的有用信息。

重写 onRouteRequest

`Global` 对象的另一个重要方面是它提供了一个监听请求和在请求被分派给一个动作前执行业务逻辑的方法。

> 提示：这种关联也可以被用来劫持请求，让开发者插入他们自己的请求路由机制。

让我们来看看这在实际中是如何运作的：

```
1.  import play.api._
2.  import play.api.mvc._
3.
4.  // Note: this is in the default package.
5.  object Global extends GlobalSettings {
6.
7.    override def onRouteRequest(request: RequestHeader):
    Option[Handler] = {
8.      println("executed before every request:" + request.toString)
9.      super.onRouteRequest(request)
10.   }
11.
12. }
```

也可以使用 `Action 组合` 来监听某个特定 Action 方法。

Testing your application

使用specs2编写一个功能测试

Play提供了一批类和简洁的方法可以用来帮助进行功能测试。其中的大多数可以在
Play.api.test包或在Helpers对象中找到。
你可以通过引入如下模块来添加这些方法和类:

```
1.  import play.api.test._
2.  import play.api.test.Helpers._
```

FakeApplication

Play经常需要一个运行的Application作为上下文: 其通常由Play.api.Play.current提供。

为了针对测试提供一个环境。Play提供了一个FakeApplication雷其可以和一个不同的全局对象，其他配置甚至其他插件一起被配置。

```
1. val fakeApplicationWithGlobal = FakeApplication(withGlobal =
   Some(new GlobalSettings() {
2.       override def onStart(app: Application) { println("Hello
   world!") }
3. }))
```

WithApplication

为了传递一个应用到一个例子，请使用WithApplication.一个显式的
FakeApplication可以被传递进去，但是默认的FakeApplication为了方便使用而被提供。
因为WithAppication是一个嵌入的Around块，你可以重写它来提供你自己的数据全域：

```
1. abstract class WithDbData extends WithApplication {
2.       override def around[T: AsResult](t: => T): Result =
   super.around {
3.         setupData()
4.         t
5.       }
6.
7.       def setupData() {
8.         // 配置数据
9.       }
10. }
11.
12. "Computer model" should {
13.
14.       "be retrieved by id" in new WithDbData {
15.         // 你的测试代码
16.       }
17.       "be retrieved by email" in new WithDbData {
18.         // 你的测试代码
19.       }
```

```
20. }
```

WithServer

有时候你想从你的测试中测试真正的HTTP协议栈，在这种例子中你可以使用WithServer来启用一个测试服务器：

```
1. "test server logic" in new WithServer(app =
   fakeApplicationWithBrowser, port = testPort) {
2.     // 这个测试支付网关在其返回一个结果前需要针对该服务器的一个回调...
3.       val callbackURL = s"http://$myPublicAddress/callback"
4.
5.       // await is from play.api.test.FutureAwaits
6.       val response =
   await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" -
   > callbackURL).get())
7.
8.       response.status must equalTo(OK)
9.     }
```

port值包含了服务器上运行的这个端口号。其默认的是19001，甚至你可以通过传递这个端口到WinthServer中或设置系统属性testserver.port来改变它。这个在与持续集成服务器结合的时候非常有用，所以这个端口可以为每个构建而动态保留。
一个FakeApplication也可以被传递到测试服务器，当设定定制的路由和测试WS调用时会非常有用：

```
1. val appWithRoutes = FakeApplication(withRoutes = {
2.       case ("GET", "/") =>
3.     Action {
4.       Ok("ok")
5.     }
6. })
7.
```

```
8.  "test WS logic" in new WithServer(app = appWithRoutes, port = 3333)
    {
9.        await(WS.url("http://localhost:3333").get()).status must
    equalTo(OK)
10. }
```

WithBrowser

如果你想使用一个浏览器来测试你的应用，你可以使用Selenium WebDriver.Play将为你启用该WebDriver,并使用FluentLenium提供的简洁的API来包裹它。如同WithServer一样使用WithBrowser，你可以更改端口，FakeApplication；而且你可以通过选择网页浏览器而使用：

```
1.  val fakeApplicationWithBrowser = FakeApplication(withRoutes = {
2.        case ("GET", "/") =>
3.      Action {
4.        Ok(
5.          """
6.            |<html>
7.            |<body>
8.            |  <div id="title">Hello Guest</div>
9.            |  <a href="/login">click me</a>
10.           |</body>
11.           |</html>
12.         """.stripMargin) as "text/html"
13.     }
14.       case ("GET", "/login") =>
15.         Action {
16.               Ok(
17.                 """
18.                   |<html>
19.                   |<body>
20.                   |  <div id="title">Hello Coco</div>
21.                   |</body>
22.                   |</html>
```

```
                """.stripMargin) as "text/html"
            }
        })

    "run in a browser" in new WithBrowser(webDriver = WebDriverFactory(HTMLUNIT), app = fakeApplicationWithBrowser) {
        browser.goTo("/")

        // 检查页面
        browser.$("#title").getTexts().get(0) must equalTo("Hello Guest")

        browser.$("a").click()

        browser.url must equalTo("/login")
        browser.$("#title").getTexts().get(0) must equalTo("Hello Coco")
    }
```

PlaySpecification

PlaySpecification是specification的一个扩展它不包含被默认的specs2规范所提供而与Play 帮助器方法相冲突的一些混合对象。为了便利它也混合进Play测试帮助器和类型。

```
object ExamplePlaySpecificationSpec extends     PlaySpecification {
        "The specification" should {

          "have access to HeaderNames" in {
                USER_AGENT must be_===("User-Agent")
            }

            "have access to Status" in {
                  OK must be_===(200)
              }
            }
```

```
12.         }
```

测试一个view模板

因为一个模板是一个标准的Scala功能，你可以从你的测试中执行它，并检测结果：

```
1.  "render index template" in new WithApplication {
2.         val html = views.html.index("Coco")
3.
4.         contentAsString(html) must contain("Hello Coco")
5.  }
```

测试一个控制器

你可以调用任何被一个FakeRequest所提供的任何Action：

```
1.  "respond to the index Action" in {
2.         val result = controllers.Application.index()(FakeRequest())
3.
4.         status(result) must equalTo(OK)
5.         contentType(result) must beSome("text/plain")
6.         contentAsString(result) must contain("Hello Bob")
7.  }
```

技术上讲，你这里不需要WithApplication，尽管使用它并不会带来不利影响。

测试路由器

取代自己调用Action，你可以让Router来做：

```
1.  "respond to the index Action" in new
```

```
    WithApplication(fakeApplication) {
2.        val Some(result) = route(FakeRequest(GET, "/Bob"))
3.
4.        status(result) must equalTo(OK)
5.        contentType(result) must beSome("text/html")
6.        charset(result) must beSome("utf-8")
7.        contentAsString(result) must contain("Hello Bob")
8.  }
```

测试一个模块

如果你使用一个SQL数据库，你可以使用inMemoryDatabase来替换与一个数据库连接的H2数据库中的内存中实例。

```
val appWithMemoryDatabase =
FakeApplication(additionalConfiguration =
inMemoryDatabase(“test”))
“run an application” in new
WithApplication(appWithMemoryDatabase) {
```

```
1.        val Some(macintosh) = Computer.findById(21)
2.
3.        macintosh.name must equalTo("Macintosh")
4.        macintosh.introduced must beSome.which(_ must
    beEqualTo("1984-01-24"))
5.  }
```

Writing functional tests with ScalaTest

Writing functional tests with ScalaTest

用ScalaTest写功能测试

为你的应用编写测试可是一个复杂的过程。Play提供了帮助手册和应用存根。并且ScalaTest提供了一个整合库，ScalaTest + Play，使测试你的应用变得尽可能简单。

总览

测试文件位于“test”目录下。

你可以通过Play控制台运行测试。

1. 点击test运行所有测试。
2. 点击类名后跟随的test-only，比如，test-only my.namespace.MySpec 来运行一个测试类。
3. 点击test-quick运行只有已经失败的测试。

```
4. 输入一个前面带~的命令，比如 ~test-quick来持续性的运行测试。
5. 运行test:console来在控制台中查看测试帮助手册如 FakeApplication.
```

在Play中测试是基于SBT,testing SBT章节中提供了更加详细的信息。

使用 ScalaTest + Play

为了能够使用ScalaTest + Play, 你将需要将它添加到你的构建中，通过改变projects/Build.scala 例如:

```
1. val appDependencies = Seq(
2.       // 在此处添加你的项目依赖,
3.       "org.scalatestplus" %% "play" % "1.1.0" % "test"
4. )
```

你不需要显性的添加ScalaTest到你的构建中。适当版本的ScalaTest会作为ScalaTest + Play的一个过渡的依赖关系。你将需要选择一个和你的Play版本匹配的一个版本的ScalaTest + Play，你可以通过ScalaTest + Play的versions页面。

ScalaTest + Play中，你可以通过扩展PlaySpec的特性来定义测试类。

例如这个例子:

```
1. import collection.mutable.Stack
2. import org.scalatestplus.play._
3.
4. class StackSpec extends PlaySpec {
5.
6.       "A Stack" must {
7.         "pop values in last-in-first-out order" in {
8.           val stack = new Stack[Int]
```

```
 9.            stack.push(1)
10.            stack.push(2)
11.            stack.pop() mustBe 2
12.            stack.pop() mustBe 1
13.          }
14.          "throw NoSuchElementException if an empty stack is popped"
     in {
15.                val emptyStack = new Stack[Int]
16.                a [NoSuchElementException] must be thrownBy {
17.                  emptyStack.pop()
18.                    }
19.              }
20.            }
21.      }
```

你或者可以通过定义自己的基类来替换使用PlaySpec.

你可以与Play一起或在IntelliJ IDEA(使用Scala plugin)或Eclipse(使用Scala IDE和 ScalaTest Eclipse plugin)中运行你的测试。请通过IDE页面来获取更多详细信息。

适配器

PlaySpec混合了ScalaTest的MustMatchers,所以你可以通过使用ScalaTest的适配器DSL来写声明变量:

```
1.  “Hello world” must endWith ("world")
```

请参考MustMatchers的文档来获取更多信息。

Mockito

你可以通过使用mocks来隔离单元测试需要的外部依赖。例如，如果你的类依赖于一个外部类DataService,你可以为你的类采集适当的数据

而不需要实例化一个DataService对象。

ScalaTest通过MockitoSuger特性提供与Mockito集成。

为了使用Mockito，混合MockitoSUger到你的测试类中然后使用Mockito库来模拟依赖关系：

```
1.  case class Data(retrievalDate: java.util.Date)
2.
3.  trait DataService {
4.        def findData: Data
5.  }
6.
7.  import org.scalatest._
8.  import org.scalatest.mock.MockitoSugar
9.  import org.scalatestplus.play._
10.
11. import org.mockito.Mockito._
12.
13. class ExampleMockitoSpec extends PlaySpec with MockitoSugar {
14.
15.       "MyService#isDailyData" should {
16.         "return true if the data is from today" in {
17.               val mockDataService = mock[DataService]
18.               when(mockDataService.findData) thenReturn Data(new
    java.util.Date())
19.
20.               val myService = new MyService() {
21.                 override def dataService = mockDataService
22.               }
23.
24.               val actual = myService.isDailyData
25.               actual mustBe true
26.             }
27.           }
28.     }
```

模拟对于测试类的公共方法是非常有用的。模拟对象和私有方法是可能的，但是难以实现。

单元测试模型

Play不需要通过模式去使用一个特定的数据库数据访问层。而且，如果应用使用Anorm或Slick,该模式往往将拥有一个数据库访问的内部引用。

```
1.  import anorm._
2.  import anorm.SqlParser._
3.
4.  case class User(id: String, name: String, email: String) {
5.         def roles = DB.withConnection { implicit connection =>
6.       ...
7.      }
8.  }
```

对于单元测试，这种方法可以微妙的模拟roles方法。

一个通用的方法是保持模式尽可能的逻辑上从数据库分离，并抽象数据库的访问于一个仓库层的后面。

```
1.  case class Role(name:String)
2.
3.  case class User(id: String, name: String, email:String)
4.
5.  trait UserRepository {
6.        def roles(user:User) : Set[Role]
7.  }
8.
9.  class AnormUserRepository extends UserRepository {
10.       import anorm._
11.       import anorm.SqlParser._
12.
```

```
13.       def roles(user:User) : Set[Role] = {
14.         ...
15.       }
16. }
```

然后通过服务访问它们：

```
1.  class UserService(userRepository : UserRepository) {
2.
3.        def isAdmin(user:User) : Boolean = {
4.          userRepository.roles(user).contains(Role("ADMIN"))
5.        }
6.  }
```

通过这种方式， isAdmin方法可以通过模拟出UserRepository引用并传递其到该服务中来测试：

```
1.  class UserServiceSpec extends PlaySpec with MockitoSugar {
2.
3.        "UserService#isAdmin" should {
4.          "be true when the role is admin" in {
5.                val userRepository = mock[UserRepository]
6.                when(userRepository.roles(any[User])) thenReturn
    Set(Role("ADMIN"))
7.
8.                val userService = new UserService(userRepository)
9.
10.               val actual = userService.isAdmin(User("11", "Steve",
    "user@example.org"))
11.               actual mustBe true
12.             }
13.           }
14.     }
```

单元测试控制器

在Play中控制器被定义为对象，所以更难来进行单元测试。在Play中可以通过依赖注入使用getControllerInstance来缓解。另一种方式去巧妙处理包含一个控制器的单元测试是针对这个控制器使用一个隐式类型自引用的一种特质：

```
1.  trait ExampleController {
2.        this: Controller =>
3.
4.        def index() = Action {
5.          Ok("ok")
6.        }
7.  }
8.
9.  object ExampleController extends Controller with ExampleController
```

并接着测试这个特质：

```
1.  import scala.concurrent.Future
2.
3.  import org.scalatest._
4.  import org.scalatestplus.play._
5.
6.  import play.api.mvc._
7.  import play.api.test._
8.  import play.api.test.Helpers._
9.
10. class ExampleControllerSpec extends PlaySpec with Results {
11.
12.       class TestController() extends Controller with
    ExampleController
13.
14.       "Example Page#index" should {
15.         "should be valid" in {
16.           val controller = new TestController()
17.           val result: Future[SimpleResult] =
    controller.index().apply(FakeRequest())
```

```
18.           val bodyText: String = contentAsString(result)
19.           bodyText mustBe "ok"
20.         }
21.       }
22. }
```

单元测试基本动作

测试Action或Filter需要测试一个EssentialAction(关于什么是EssentialAction的详细信息)

对此，测试Helpers.call可以像这样使用:

```
1.  class ExampleEssentialActionSpec extends PlaySpec {
2.
3.       "An essential action" should {
4.         "can parse a JSON body" in {
5.               val action: EssentialAction = Action { request =>
6.             val value = (request.body.asJson.get \
    "field").as[String]
7.             Ok(value)
8.           }
9.
10.              val request = FakeRequest(POST,
    "/").withJsonBody(Json.parse("""{ "field": "value" }"""))
11.
12.              val result = call(action, request)
13.
14.              status(result) mustEqual OK
15.              contentAsString(result) mustEqual "value"
16.        }
17.      }
18. }
```

Testing with specs2

Testing with specs2

使用specs2来测试你的应用

为你的应用编写测试是一个参与的过程。Play为你提供了一个默认的测试框架，并提供了帮助器和应用存根使测试你的应用尽可能的简单。

概述

测试文件的位置在”test”文件夹中。这里有两个简单的测试文件其可以被用作为模板。

你可以从Play控制台运行测试。

1. 点击test来按钮运行所有测试。
2. 点击标注测试类名字比如：test-only my.namespace.MySpec的test-only按钮来运行一个测试类。
3. 点击test-quick按钮来运行会失败的测试类。
4. 运行一个前面带波浪线的的命令比如~test-quick来持续的运行测试。

```
5. 点击test:console按钮来在控制台中访问测试帮助器如FakeApplication.
```

Testing in Play is based on SBT, and a full description is available in the testing SBT chapter. 在Play中测试是基于SBT，完整的描述请参考testing SBT章节。

使用specs2

在specs2中，测试组织成规格，其包含了运行基于不同代码路径运行的测试的系统。

规格扩展了Specification特性big使用should/in格式：

```
1. import org.specs2.mutable._
2.
3. class HelloWorldSpec extends Specification {
4.
5.       "The 'Hello world' string" should {
6.         "contain 11 characters" in {
7.           "Hello world" must have size(11)
8.     }
9.     "start with 'Hello'" in {
10.          "Hello world" must startWith("Hello")
11.    }
12.    "end with 'world'" in {
13.          "Hello world" must endWith("world")
14.        }
15.      }
16. }
```

规格可以在IntelliJ IDEA(使用Scala插件)或Eclipse（使用Scala IDE）中运行，更多细节请参考IDE网页。

注意： 基于展示编译器的一个漏洞，在Eclipse中测试必须被定义为一个特定的格式：

```
1. 包名字必须与目录路径名完全一致。
2. 规格必须与@RunWith(classOf[JUnitRunner])一起声明。
```

这里一个Eclipse中可用的规格:

package models // 这个文件必须存在于一个名字为“models”的目录中

```
1. import org.specs2.mutable._
2. import org.specs2.runner._
3. import org.junit.runner._
4.
5. @RunWith(classOf[JUnitRunner])
6. class ApplicationSpec extends Specification {
7.       ...
8. }
```

适配器

当你使用一个示例，你必须返回一个示例结果，通常，你将看到一个包含mast字段的声明:

“Hello world” must endWith(“world”)

这个跟随must关键词的表达式被称为matchers.适配器返回一个示例结果，通常成功或失败。这个示例不会被编译如果其不会返回一个结果。

最有用的适配器是匹配结果。用来检测相等性，判断部分和两者其一的结果，甚至检测是否抛出异常。

这里同时有部分性适配器其允许在测试中使用XML和JSON匹配。

Mockito

Mocks 用来隔离单元测试和外部依赖。例如，如果你的类以来一个外

部的DataService类，你可以针对你的类输入适当的数据而不需要实例化一个DataService对象。

Mockito继承与specs2中作为默认的mocking库。
为了使用Mockito,添加以下的引用到你的程序中:

```
1.  import org.specs2.mock._
2.
3.  你可以模拟出引用类如：
4.  trait DataService {
5.        def findData: Data
6.  }
7.
8.  case class Data(retrievalDate: java.util.Date)
9.
10. import org.specs2.mock._
11. import org.specs2.mutable._
12.
13. import java.util._
14.
15. class ExampleMockitoSpec extends Specification with Mockito {
16.
17.       "MyService#isDailyData" should {
18.         "return true if the data is from today" in {
19.               val mockDataService = mock[DataService]
20.               mockDataService.findData returns Data(retrievalDate =
    new java.util.Date())
21.
22.               val myService = new MyService() {
23.             override def dataService = mockDataService
24.           }
25.
26.           val actual = myService.isDailyData
27.           actual must equalTo(true)
28.         }
29.       }
30.
```

```
31. }
```

Mocking 在测试类的公共方法时尤其有效。Mocking对象和私有方法也可以但是异常困难。

使用测试模块

Play不需要模块来使用特定的数据库数据访问层。设置如果应用使用Anorm或Slick,这个模块内部将拥有一个针对数据库访问的引用。

```
1. import anorm._
2. import anorm.SqlParser._
3.
4. case class User(id: String, name: String, email: String) {
5.        def roles = DB.withConnection { implicit connection =>
6.           ...
7.     }
8. }
```

针对单元测试，这种方式可以技巧的模拟出roles方法。
一个通用的方式是保持模块从数据库中分离出来并且尽可能的逻辑化，并抽象出基于一个库层的数据库访问。

```
1. case class Role(name:String)
2.
3. case class User(id: String, name: String, email:String)
4.
5. trait UserRepository {
6.       def roles(user:User) : Set[Role]
7. }
8.
9. class AnormUserRepository extends UserRepository {
10.       import anorm._
11.       import anorm.SqlParser._
12.
```

```
13.       def roles(user:User) : Set[Role] = {
14.         ...
15.       }
16. }
```

然后通过服务访问它们：

```
1.  class UserService(userRepository : UserRepository) {
2.
3.        def isAdmin(user:User) : Boolean = {
4.          userRepository.roles(user).contains(Role("ADMIN"))
5.        }
6.  }
```

以这种方式，isAdmin方法可以通过模拟出UserRepository引用并传递其到服务中来被测试：

```
1.  object UserServiceSpec extends Specification with Mockito {
2.
3.        "UserService#isAdmin" should {
4.          "be true when the role is admin" in {
5.                val userRepository = mock[UserRepository]
6.                userRepository.roles(any[User]) returns
    Set(Role("ADMIN"))
7.
8.                val userService = new UserService(userRepository)
9.                val actual = userService.isAdmin(User("11", "Steve",
    "user@example.org"))
10.               actual must beTrue
11.             }
12.           }
13.     }
```

单元测试控制器

Controllers are defined as objects in Play, and so

can be trickier to unit test. In Play this can be alleviated by dependency injection using getControllerInstance. Another way to finesse unit testing with a controller is to use a trait with an explicitly typed self reference to the controller:
在Play中控制器被定义为对象，因此更难被单元测试使用。在Play中这可以通过依赖注入使用getControllerInstance来缓解。另一种方式去处理有一个控制器的单元测试是针对这个控制器使用一个有显示的类型自我引用的特性。

```
1.  trait ExampleController {
2.        this: Controller =>
3.
4.        def index() = Action {
5.          Ok("ok")
6.        }
7.  }
8.
9.  object ExampleController extends Controller with ExampleController
10.
11. 然后测试该特性：
12. import play.api.mvc._
13. import play.api.test._
14. import scala.concurrent.Future
15.
16. object ExampleControllerSpec extends PlaySpecification with Results
    {
17.
18.       class TestController() extends Controller with
    ExampleController
19.
20.       "Example Page#index" should {
21.         "should be valid" in {
22.               val controller = new TestController()
23.               val result: Future[Result] =
```

```
    controller.index().apply(FakeRequest())
24.                 val bodyText: String = contentAsString(result)
25.                 bodyText must be equalTo "ok"
26.           }
27.         }
28. }
```

单元测试EssentialAction

测试Action或Filter需要测试一个EssentialAction(更详细的信息请参见什么是EssentialAction)

对此，这个测试Helpers.call可是这样使用:

```
1.  object ExampleEssentialActionSpec extends PlaySpecification {
2.
3.        "An essential action" should {
4.          "can parse a JSON body" in {
5.                val action: EssentialAction = Action { request =>
6.              val value = (request.body.asJson.get \
    "field").as[String]
7.              Ok(value)
8.                }
9.
10.       val request = FakeRequest(POST,
    "/").withJsonBody(Json.parse("""{ "field": "value" }"""))
11.
12.       val result = call(action, request)
13.
14.       status(result) mustEqual OK
15.       contentAsString(result) mustEqual "value"
16.         }
17.       }
18. }
```

Writing functional tests with specs2

Writing functional tests with specs2

Writing functional tests with specs2

通过ScalaTest编写功能性测试

Play提供了一系列的类和简洁的方法来帮助实现功能性测试。大多数可以在play.api.test包或Helpers对象中找到。ScalaTest + Play集成库位ScalaTest构建了这种测试支持。

你可以通过以下的模块导入来访问所有的Play的内建测试支持和ScalaTest + Play:

```
1. import org.scalatest._
2. import play.api.test._
```

```
3. import play.api.test.Helpers._
4. import org.scalatestplus.play._
```

FakeApplication

Play经常性需要一个运行的Application作为上下文：其通常通过play.api.Play.current提供。

为了为测试提供一个环境，Play提供了一个FakeApplication类其可以和一个不同的Global对象，附加配置或附加的插件一起被配置。

```
1. val fakeApplicationWithGlobal = FakeApplication(withGlobal =
   Some(new GlobalSettings() {
2.       override def onStart(app: Application) { println("Hello
   world!") }
3. }))
```

如果所有或大多数你的测试类需要一个FakeApplication,并且它们可以共享相同的FakeApplication,与OneAppPerSuite特性混合使用。你可以通过定制FakeApplication来如果这个例子展示的那样重写PP：

```
1. class ExampleSpec extends PlaySpec with OneAppPerSuite {
2.     //重写app如果你需要一个不仅有默认参数的FakeApplication。
3.       implicit override lazy val app: FakeApplication =
4.     FakeApplication(
5.           additionalConfiguration = Map("ehcacheplugin" ->
   "disabled")
6.     )
7.
8.       "The OneAppPerSuite trait" must {
9.         "provide a FakeApplication" in {
10.              app.configuration.getString("ehcacheplugin") mustBe
   Some("disabled")
11.            }
```

```
12.             "start the FakeApplication" in {
13.                   Play.maybeApplication mustBe Some(app)
14.                 }
15.               }
16.     }
```

如果你需要每个测试去获取它自己的FakeApplication而不是共享相同的，请使用OneAppPerTest来替代：

```
1.  class ExampleSpec extends PlaySpec with OneAppPerTest {
2.      //重写app如果你需要一个不仅有默认参数的FakeApplication。
3.        implicit override def newAppForTest(td: TestData):
    FakeApplication =
4.      FakeApplication(
5.            additionalConfiguration = Map("ehcacheplugin" ->
    "disabled")
6.          )
7.
8.        "The OneAppPerTest trait" must {
9.          "provide a new FakeApplication for each test" in {
10.               app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
11.         }
12.         "start the FakeApplication" in {
13.               Play.maybeApplication mustBe Some(app)
14.             }
15.           }
16.     },
```

ScalaTest + Play提供了OneAppPerSuite和OneAppPerTest的原因是来允许你去选择共享性策略来让你的测试最快的运行。如果你想让应用状态在成功的测试之间保持，你将需要使用OneAppPerSuite.如果每个测试需要一个清空的状态，你可以使用OneAppPerTest或使用OneAppPerSuite,但需要在每个测试结束来清理状态。甚至，如果你的测试集将尽可能快的运行多个测试类共享这个相同的应用，你可以

定义一个主集合起与OneAppPerSuite混用和内嵌的集合其和ConfiguredApp混用，正如在ConfiguredApp的文档中的例子展示的那样。你可以使用任意策略来实你的测试集最快的运行。

和一个服务器一起测试

有时候你想使用真的HTTP栈来进行测试。如果你的测试类中的所有测试能够重用这个相同的服务器实例，你可以通过与OneServerPerSuite一起混用（其将为这个集合提供一个新的FakeApplication）：

```
1.  class ExampleSpec extends PlaySpec with OneServerPerSuite {
2.      // 重写app如果你需要一个不仅有默认参数的FakeApplication。
3.        implicit override lazy val app: FakeApplication =
4.      FakeApplication(
5.            additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.            withRoutes = {
7.              case ("GET", "/") => Action { Ok("ok") }
8.            }
9.      )
10.
11.    "test server logic" in {
12.      val myPublicAddress =  s"localhost:$port"
13.      val testPaymentGatewayURL = s"http://$myPublicAddress"
14.      //这个测试支付网关在它返回一个值之前需要针对这个服务器的一个回调...
15.      val callbackURL = s"http://$myPublicAddress/callback"
16.      // 从play.api.test.FutureAwaits中等待调用
17.      val response =
    await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" -
    > callbackURL).get())
18.
19.      response.status mustBe (OK)
20.        }
21.  }
```

如果你的测试类中的所有测试需要不同的服务器实例，请使用OneServerPerTest来代替（其将同时为这个测试集提供一个新的FakeApplication）：

```
1.  class ExampleSpec extends PlaySpec with OneServerPerTest {
2.      //重写newAppForTest如果你需要一个不仅有默认参数的FakeApplication.
3.        override def newAppForTest(testData: TestData):
    FakeApplication =
4.              new FakeApplication(
5.                additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.                withRoutes = {
7.                  case ("GET", "/") => Action { Ok("ok") }
8.                }
9.          )
10.
11.       "The OneServerPerTest trait" must {
12.             "test server logic" in {
13.                   val myPublicAddress =  s"localhost:$port"
14.                   val testPaymentGatewayURL =
    s"http://$myPublicAddress"
15.                 //测试支付网关在返回一个结果之前需要针对这个服务器的一个回
    调...
16.                   val callbackURL =
    s"http://$myPublicAddress/callback"
17.                   // await is from play.api.test.FutureAwaits
18.                   val response =
    await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" -
    > callbackURL).get())
19.
20.                   response.status mustBe (OK)
21.                 }
22.               }
23.         }
```

OneServerPerSuite和OneServerPerTest特性为该服务器运行为

port域提供了port号。其默认是19001，甚至你可以通过重写端口或设定系统属性testserver.port来更改。其将非常有用当与持续集成服务器集成在一起，因此端口可以被动态的为每个构建所保留。

你同时可以如之前例子所展示的那样通过重写app来自定义FakeApplication.

最后，如果允许多个测试类来共享这个相同的服务器将会给你比OneServerPerSuite或OneServerPerTest方式更好的性能，正如ConfiguredServer文档中的例子所展示的那样，你可以定义一个主测试集其与OneServePerSuite混用和嵌入式测试集其与ConfiguredServer混用。

和一个网页浏览器一起测试

ScalaTest + Play库构基于Selenium DSL构建以使其简单的从网页浏览器中测试你的Play应用。
混合OneBrowserPerSuite到你的测试类中以便使用一个相同的浏览器来运行你的测试类中所有的测试。你将同时需要与一个BrowserFactory特性其将提供一个Selenium web驱动混用: ChromeFactory, FirefoxFactory, HtmlUnitFactory, InternetExplorerFactory, SafariFactory之一。

更进一步为了混合入一个BrowserFactory，你将需要混合进一个ServerProvider特性其提供了一个TestServer: ChromeFactory, FirefoxFactory, HtmlUnitFactory, InternetExplorerFactory, SafariFactory之一。

比如，以下的测试类混合进OneServerPerSuite和HtmUnitFactory:

```
1.  class ExampleSpec extends PlaySpec with OneServerPerSuite with
    OneBrowserPerSuite with HtmlUnitFactory {
2.      //重写app如果你需要一个不仅有默认参数的FakeApplication.
3.    implicit override lazy val app: FakeApplication =
4.      FakeApplication(
5.            additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.            withRoutes = {
7.              case ("GET", "/testing") =>
8.                Action(
9.                  Results.Ok(
10.                   "<html>" +
11.                 "<head><title>Test Page</title></head>" +
12.                 "<body>" +
13.                 "<input type='button' name='b' value='Click Me'
    onclick='document.title=\"scalatest\"' />" +
14.                 "</body>" +
15.                 "</html>"
16.                 ).as("text/html")
17.               )
18.           }
19.     )
20.
21.   "The OneBrowserPerTest trait" must {
22.     "provide a web driver" in {
23.           go to (s"http://localhost:$port/testing")
24.           pageTitle mustBe "Test Page"
25.           click on find(name("b")).value
26.           eventually { pageTitle mustBe "scalatest" }
27.         }
28.       }
29. }
```

如果你的每个测试需要一个新的浏览器实例，请使用OneBrowserPerTest来替代。比如OneBrowserPerSuite,你将需要同时与一个ServerProvider和BrowserFactory来混用:

```
1.  class ExampleSpec extends PlaySpec with OneServerPerTest with
    OneBrowserPerTest with HtmlUnitFactory {
2.      //重写newAppForTest如果你需要一个不仅有默认参数的FakeApplication.
3.        override def newAppForTest(testData: TestData):
    FakeApplication =
4.          new FakeApplication(
5.                additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.                withRoutes = {
7.                  case ("GET", "/testing") =>
8.                        Action(
9.                          Results.Ok(
10.                               "<html>" +
11.                           "<head><title>Test Page</title></head>"
    +
12.                           "<body>" +
13.                           "<input type='button' name='b'
    value='Click Me' onclick='document.title=\"scalatest\"' />" +
14.                           "</body>" +
15.                           "</html>"
16.                         ).as("text/html")
17.                       )
18.                   }
19.             )
20.
21.   "The OneBrowserPerTest trait" must {
22.     "provide a web driver" in {
23.           go to (s"http://localhost:$port/testing")
24.           pageTitle mustBe "Test Page"
25.           click on find(name("b")).value
26.           eventually { pageTitle mustBe "scalatest" }
27.         }
28.       }
29. }
```

如果你需要多个测试类来共享这个相同的浏览器实例，混合OneBrowserPerSuite进一个主测试集和混合ConfiguredBrowser

进多个内嵌的测试集。内嵌的测试集将一起共享相同的浏览器。请参考ConfiguredBrowser特性的文档中的例子。

在多浏览器中运行相同的测试

如果你想在多浏览器中运行测试，以保证你的应用可以在所有你支持的浏览器中正常的工作，你可以使用AllBrowserPerSuite或AllBrowsersPerTest特性。这些特性都声明了一个IndexedSeq[BrowserInfo]特性和一个抽象的sharedTests方法其拥有一个BrowerInfo.browsers域指定了那个浏览器你想让你的测试在其中运行。默认的是Chrome, Firefox, Internet Explorer, HtmlUnit和Safari. 你可以重写browsers如果默认的不符合你的需求.你在sharedTests方法中嵌入你想在多浏览器中运行的测试，将浏览器的名字置于每个测试名的后面（浏览器名字是可用的从BrowserInfo传入进sharedTests.）这里有一个使用AllBrowsersPerSuite的例子:

```
1.  class ExampleSpec extends PlaySpec with OneServerPerSuite with
    AllBrowsersPerSuite {
2.      //重写App如果你需要一个不仅有默认参数的FakeApplication.
3.        implicit override lazy val app: FakeApplication =
4.          FakeApplication(
5.                additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.                withRoutes = {
7.                  case ("GET", "/testing") =>
8.                    Action(
9.                      Results.Ok(
10.                           "<html>" +
11.                       "<head><title>Test Page</title></head>" +
12.                       "<body>" +
13.                       "<input type='button' name='b' value='Click
    Me' onclick='document.title=\"scalatest\"' />" +
```

```
                        "</body>" +
                        "</html>"
                    ).as("text/html")
                )
            }
        )

      def sharedTests(browser: BrowserInfo) = {
        "The AllBrowsersPerSuite trait" must {
            "provide a web driver " + browser.name in {
              go to (s"http://localhost:$port/testing")
              pageTitle mustBe "Test Page"
              click on find(name("b")).value
              eventually { pageTitle mustBe "scalatest" }
            }
        }
      }
}
```

所有被sharedTests所声明的测试将被在browsers域中提到的所有浏览器一起运行，只要它们在客户端系统中是可用的。为任何浏览器其不存在于客户端系统的测试将会自动被取消。注意你需要手动的为测试名称追加browser.name以保障这个集合内的每个测试都有一个独立的名字（ScalaTest需要它）.如果你保留其空白，当你运行你的测试的时候将得到一个重复测试名字的错误。

AllBrowsersPerSuite 将为每种类型的浏览器创建一个单独的实例并为在sharedTests中声明的所有测试中使用。如果你想要每个测试去拥有它自己的，全新品牌的浏览器实例，请使用AllBrowsersPerTest来替代：

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with
AllBrowsersPerTest {
    //重写app如果你需要一个不仅有默认参数的FakeApplication.
      implicit override lazy val app: FakeApplication =
```

```
4.          FakeApplication(
5.                additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
6.                withRoutes = {
7.                  case ("GET", "/testing") =>
8.                    Action(
9.                      Results.Ok(
10.                           "<html>" +
11.                         "<head><title>Test Page</title></head>" +
12.                         "<body>" +
13.                         "<input type='button' name='b' value='Click
    Me' onclick='document.title=\"scalatest\"' />" +
14.                         "</body>" +
15.                             "</html>"
16.                     ).as("text/html")
17.                   )
18.               }
19.         )
20.
21.       def sharedTests(browser: BrowserInfo) = {
22.             "The AllBrowsersPerTest trait" must {
23.               "provide a web driver"  + browser.name in {
24.                 go to (s"http://localhost:$port/testing")
25.                 pageTitle mustBe "Test Page"
26.                 click on find(name("b")).value
27.                 eventually { pageTitle mustBe     "scalatest" }
28.               }
29.             }
30.       }
31. }
```

尽管AllBrowserPerSuite和AllBrowsersPerTest都会针对不提供的浏览器类型而取消测试，这些测试
仍将在输出中显示为被取消。为了清理这些不必要的输出，你可以如下例子所示来通过重写browsers来排除那些永远不会被支持的网页浏览器：

```
1.  class ExampleOverrideBrowsersSpec extends PlaySpec with
    OneServerPerSuite with AllBrowsersPerSuite {
2.
3.        override lazy val browsers =
4.          Vector(
5.            FirefoxInfo(firefoxProfile),
6.            ChromeInfo
7.          )
8.       //重写app如果你需要一个不仅有默认参数的FakeApplication.
9.        implicit override lazy val app: FakeApplication =
10.         FakeApplication(
11.               additionalConfiguration = Map("ehcacheplugin" ->
    "disabled"),
12.               withRoutes = {
13.                 case ("GET", "/testing") =>
14.                   Action(
15.                     Results.Ok(
16.                           "<html>" +
17.                             "<head><title>Test Page</title></head>"
    +
18.                          "<body>" +
19.                         "<input type='button' name='b' value='Click
    Me' onclick='document.title=\"scalatest\"' />" +
20.                         "</body>" +
21.                         "</html>"
22.                         ).as("text/html")
23.                       )
24.                   }
25.             )
26.
27.       def sharedTests(browser: BrowserInfo) = {
28.         "The AllBrowsersPerSuite trait" must {
29.               "provide a web driver"  + browser.name in {
30.                         go to (s"http://localhost:$port/testing")
31.                     pageTitle mustBe "Test Page"
32.                     click on find(name("b")).value
33.                     eventually { pageTitle mustBe "scalatest" }
34.                   }
```

```
            }
          }
    }
```

之前的测试类仅将尝试和Firefox，chrome一起运行共享的测试（并自动化的取消测试如果一个浏览器是不可用的）。

PlaySpec

PlaySpec为Play测试提供了一个便捷的“超集”ScalaTest基础类，你可以通过扩展
PlaySpec来自动得到WordSpec, MustMatchers, OptionValues, 和 WsScalaTestClient:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with ScalaFutures with IntegrationPatience {
    //重写app如果你需要一个不仅有默认参数的FakeApplication.
      implicit override lazy val app: FakeApplication =
        FakeApplication(
              additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
              withRoutes = {
                case ("GET", "/testing") =>
                  Action(
                    Results.Ok(
                          "<html>" +
                        "<head><title>Test Page</title></head>" +
                        "<body>" +
                        "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
                        "</body>" +
                        "</html>"
                        ).as("text/html")
                    )
                }
            )
```

```
20.       "WsScalaTestClient's" must {
21.
22.         "wsUrl works correctly" in {
23.               val futureResult = wsUrl("/testing").get
24.               val body = futureResult.futureValue.body
25.               val expectedBody =
26.                 "<html>" +
27.                   "<head><title>Test Page</title></head>" +
28.                   "<body>" +
29.                   "<input type='button' name='b' value='Click Me'
    onclick='document.title=\"scalatest\"' />" +
30.                   "</body>" +
31.                   "</html>"
32.                   assert(body == expectedBody)
33.             }
34.
35.         "wsCall works correctly" in {
36.           val futureResult = wsCall(Call("get", "/testing")).get
37.           val body = futureResult.futureValue.body
38.           val expectedBody =
39.             "<html>" +
40.               "<head><title>Test Page</title></head>" +
41.               "<body>" +
42.               "<input type='button' name='b' value='Click Me'
    onclick='document.title=\"scalatest\"' />" +
43.               "</body>" +
44.               "</html>"
45.           assert(body == expectedBody)
46.         }
47.       }
48. }
```

你可以将任何之前提到的特性混合进PlaySpec.

在之前例子中所展示的那些测试类中，测试类中的所有或大多数测试需要相同的样式。这个很常见但不总是这样。如果相同的测试类中不同的测试需要不同的样式，与特性MixedFixtures混用。然后使用这些无

参功能: App, Server, Chrome, Firefox, HtmlUnit, InternetExplorer, 或 Safari之一来给予每个测试器需要的样式.
你不可以混合MixedFixtures进PlaySpec。因为MixedFixtures需要一个ScalaTest fixture.Suite但PlaySpec仅仅是一个不同的集合。如果你需要针对混合样式的一个简洁的基础类。通过扩展MixedPlaySpec来替代。这里是一个例子:

```
1.  // MixedPlaySpec 已经混合进MixedFixtures
2.  class ExampleSpec extends MixedPlaySpec {
3.  // 一些帮助器方法
4.        def fakeApp[A](elems: (String, String)*) =
    FakeApplication(additionalConfiguration = Map(elems:_*),
5.          withRoutes = {
6.                case ("GET", "/testing") =>
7.              Action(
8.                    Results.Ok(
9.                      "<html>" +
10.                       "<head><title>Test Page</title></head>" +
11.                       "<body>" +
12.                       "<input type='button' name='b' value='Click
    Me' onclick='document.title=\"scalatest\"' />" +
13.                       "</body>" +
14.                       "</html>"
15.                   ).as("text/html")
16.             )
17.         })
18.       def getConfig(key: String)(implicit app: Application) =
    app.configuration.getString(key)
19.  // 如果一个测试进需要一个FakeApplication, 使用"new APP":
20.       "The App function" must {
21.         "provide a FakeApplication" in new
    App(fakeApp("ehcacheplugin" -> "disabled")) {
22.           app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
23.         }
```

```
24.         "make the FakeApplication available implicitly" in new
    App(fakeApp("ehcacheplugin" -> "disabled")) {
25.               getConfig("ehcacheplugin") mustBe Some("disabled")
26.         }
27.         "start the FakeApplication" in new
    App(fakeApp("ehcacheplugin" -> "disabled")) {
28.               Play.maybeApplication mustBe Some(app)
29.         }
30.       }
31. // 如果一个测试需要一个FakeApplicaiton并运行TestServer,请使用"new
    Server":
32.   "The Server function" must {
33.     "provide a FakeApplication" in new
    Server(fakeApp("ehcacheplugin" -> "disabled"))     {
34.           app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
35.         }
36.     "make the FakeApplication available implicitly" in new
    Server(fakeApp("ehcacheplugin" -> "disabled")) {
37.           getConfig("ehcacheplugin") mustBe Some("disabled")
38.     }
39.     "start the FakeApplication" in new
    Server(fakeApp("ehcacheplugin" -> "disabled")) {
40.           Play.maybeApplication mustBe Some(app)
41.     }
42.     import Helpers._
43.     "send 404 on a bad request" in new Server {
44.           import java.net._
45.           val url = new URL("http://localhost:" + port + "/boom")
46.           val con =
    url.openConnection().asInstanceOf[HttpURLConnection]
47.           try con.getResponseCode mustBe 404
48.           finally con.disconnect()
49.     }
50.   }
51.
52. // 如果一个测试需要一个FakeApplication,运行TestServer, 和Selenium
    HtmlUnit驱动请使用"new HtmlUnit":
```

```
53.   "The HtmlUnit function" must {
54.     "provide a FakeApplication" in new
    HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
55.       app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
56.     }
57.     "make the FakeApplication available implicitly" in new
    HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
58.           getConfig("ehcacheplugin") mustBe Some("disabled")
59.     }
60.     "start the FakeApplication" in new
    HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
61.           Play.maybeApplication mustBe Some(app)
62.     }
63.     import Helpers._
64.     "send 404 on a bad request" in new HtmlUnit {
65.           import java.net._
66.           val url = new URL("http://localhost:" + port + "/boom")
67.           val con =
    url.openConnection().asInstanceOf[HttpURLConnection]
68.           try con.getResponseCode mustBe 404
69.           finally con.disconnect()
70.     }
71.     "provide a web driver" in new HtmlUnit(fakeApp()) {
72.          go to ("http://localhost:" + port + "/testing")
73.           pageTitle mustBe "Test Page"
74.           click on find(name("b")).value
75.           eventually { pageTitle mustBe "scalatest" }
76.     }
77.   }
78.
79. // 如果一个测试需要一个FakeApplication,运行TestServer,和Selenium
    Firefox驱动请使用"new Firefox"：
80.   "The Firefox function" must {
81.     "provide a FakeApplication" in new
    Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
82.       app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
```

```
83.     }
84.     "make the FakeApplication available implicitly" in new
    Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
85.           getConfig("ehcacheplugin") mustBe Some("disabled")
86.     }
87.     "start the FakeApplication" in new
    Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
88.           Play.maybeApplication mustBe Some(app)
89.     }
90.     import Helpers._
91.     "send 404 on a bad request" in new Firefox {
92.           import java.net._
93.           val url = new URL("http://localhost:" + port + "/boom")
94.           val con =
    url.openConnection().asInstanceOf[HttpURLConnection]
95.           try con.getResponseCode mustBe 404
96.           finally con.disconnect()
97.     }
98.     "provide a web driver" in new Firefox(fakeApp()) {
99.           go to ("http://localhost:" + port + "/testing")
100.          pageTitle mustBe "Test Page"
101.          click on find(name("b")).value
102.          eventually { pageTitle mustBe "scalatest" }
103.    }
104.  }
105.
106. // // 如果一个测试需要一个FakeApplication,运行TestServer,和Selenium
    Safari驱动请使用"new Safari"：
107.   "The Safari function" must {
108.     "provide a FakeApplication" in new
    Safari(fakeApp("ehcacheplugin" -> "disabled")) {
109.           app.configuration.getString("ehcacheplugin") mustBe
    Some("disabled")
110.     }
111.     "make the FakeApplication available implicitly" in new
    Safari(fakeApp("ehcacheplugin" -> "disabled")) {
112.           getConfig("ehcacheplugin") mustBe Some("disabled")
113.     }
```

```
114.      "start the FakeApplication" in new
      Safari(fakeApp("ehcacheplugin" -> "disabled")) {
115.            Play.maybeApplication mustBe Some(app)
116.      }
117.      import Helpers._
118.      "send 404 on a bad request" in new Safari {
119.            import java.net._
120.            val url = new URL("http://localhost:" + port + "/boom")
121.            val con =
      url.openConnection().asInstanceOf[HttpURLConnection]
122.            try con.getResponseCode mustBe 404
123.            finally con.disconnect()
124.      }
125.      "provide a web driver" in new Safari(fakeApp()) {
126.            go to ("http://localhost:" + port + "/testing")
127.            pageTitle mustBe "Test Page"
128.            click on find(name("b")).value
129.            eventually { pageTitle mustBe "scalatest" }
130.      }
131.    }
132.
133.  // 如果一个测试需要一个FakeApplication,运行TestServer,和Selenium Chrome
      驱动请使用"new Chrome"：
134.    "The Chrome function" must {
135.      "provide a FakeApplication" in new
      Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
136.            app.configuration.getString("ehcacheplugin") mustBe
      Some("disabled")
137.      }
138.      "make the FakeApplication available implicitly" in new
      Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
139.            getConfig("ehcacheplugin") mustBe Some("disabled")
140.      }
141.      "start the FakeApplication" in new
      Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
142.            Play.maybeApplication mustBe Some(app)
143.      }
144.      import Helpers._
```

```
145.      "send 404 on a bad request" in new Chrome {
146.            import java.net._
147.            val url = new URL("http://localhost:" + port + "/boom")
148.            val con =
      url.openConnection().asInstanceOf[HttpURLConnection]
149.            try con.getResponseCode mustBe 404
150.            finally con.disconnect()
151.      }
152.      "provide a web driver" in new Chrome(fakeApp()) {
153.            go to ("http://localhost:" + port + "/testing")
154.            pageTitle mustBe "Test Page"
155.            click on find(name("b")).value
156.            eventually { pageTitle mustBe "scalatest" }
157.      }
158.    }
159.
160.  // 如果一个测试需要一个FakeApplication,运行TestServer,和Selenium
      InternetExplorer驱动请使用"new InternetExplorer":
161.    "The InternetExplorer function" must {
162.      "provide a FakeApplication" in new
      InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
163.            app.configuration.getString("ehcacheplugin") mustBe
      Some("disabled")
164.      }
165.      "make the FakeApplication available implicitly" in new
      InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
166.            getConfig("ehcacheplugin") mustBe Some("disabled")
167.      }
168.      "start the FakeApplication" in new
      InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
169.        Play.maybeApplication mustBe Some(app)
170.      }
171.      import Helpers._
172.      "send 404 on a bad request" in new InternetExplorer {
173.            import java.net._
174.            val url = new URL("http://localhost:" + port + "/boom")
175.            val con =
      url.openConnection().asInstanceOf[HttpURLConnection]
```

```
176.            try con.getResponseCode mustBe 404
177.            finally con.disconnect()
178.      }
179.      "provide a web driver" in new InternetExplorer(fakeApp()) {
180.            go to ("http://localhost:" + port + "/testing")
181.            pageTitle mustBe "Test Page"
182.            click on find(name("b")).value
183.            eventually { pageTitle mustBe "scalatest" }
184.      }
185.    }
186.
187.  // 如果一个测试不需要任何的特殊样式，仅需要写成"in { () => ... "
188.    "Any old thing" must {
189.      "be doable without much boilerplate" in { () =>
190.         1 + 1 mustEqual 2
191.       }
192.    }
193.  }
```

测试一个模板

因为一个模板是一个标准的Scala功能，你可以从你的测试中执行它，并检测结果：

```
1.  "render index template" in new App {
2.        val html = views.html.index("Coco")
3.
4.        contentAsString(html) must include ("Hello Coco")
5.  }
```

测试一个控制器

You can call any Action code by providing a FakeRequest:
你可以通过提供一个FakRequest来调用任何Action代码：

```
import scala.concurrent.Future

import org.scalatest._
import org.scalatestplus.play._

import play.api.mvc._
import play.api.test._
import play.api.test.Helpers._

class ExampleControllerSpec extends PlaySpec with Results {

      class TestController() extends Controller with
ExampleController

      "Example Page#index" should {
        "should be valid" in {
              val controller = new TestController()
              val result: Future[SimpleResult] =
controller.index().apply(FakeRequest())
              val bodyText: String = contentAsString(result)
              bodyText mustBe "ok"
            }
          }
    }
```

技术上讲，在这里你不需要WithApption,尽管使用它并不会产生任何不利影响。

测试一个路由

你可以通过让Router来做而不是你自己来调用Action:

```
"respond to the index Action" in new App(fakeApplication) {
      val Some(result) = route(FakeRequest(GET, "/Bob"))

      status(result) mustEqual OK
      contentType(result) mustEqual Some("text/html")
```

```
6.         charset(result) mustEqual Some("utf-8")
7.         contentAsString(result) must include ("Hello Bob")
8.   }
```

测试一个模块

如果你使用一个SQL数据库，你可以通过使用inMemoryDatabase来用一个H2数据库的内存实例替换数据库连接。

```
val appWithMemoryDatabase = FakeApplication(additionalConfigurat
"run an application" in new App(appWithMemoryDatabase) {

      val Some(macintosh) = Computer.findById(21)

      macintosh.name mustEqual "Macintosh"
      macintosh.introduced.value mustEqual "1984-01-24"
}
```

测试WS调用

如果你正在调用一个网页服务，你可以使用WSTestClinet.这里两个调用可用，wsCall和wsUrl其将分别执行一个调用或字符串。注意他们期望在WithApplication的上下文里被调用。

```
wsCall(controllers.routes.Application.index()).get()

wsUrl("http://localhost:9000").get()
```

The Logging API

The Logging API

在你的应用中使用日志有助于监控、调试、错误跟踪以及商业智能分析。Play 提供了日志 API，可以通过 `Logger` 对象来使用，Play 使用 Logback 作为日志引擎。

日志体系结构

日志 API 提供了一组构件来帮助你实现高效的日志记录策略。

Logger

你的应用可以定义 `Logger` 实例来发送日志信息。每个 `Logger` 有一个名字，它会出现在日志信息中，该名字是用于配置层级关系的。

Logger 根据它们的名字会有一个层级继承结构。如果一个 logger 的名字后面加个点后是另外一个 logger 名字的前缀，那么我们就说

第一个 logger 是第二个 logger 的祖先。例如，名为 “com.foo” 的 logger 是名为 “com.foo.bar.Baz” 的 logger 的祖先。所有的 logger 都继承自根 logger。Logger 继承机制的一个好处就是，如果你想配置一组 logger，那你只需要配置它们的共同祖先即可。

Play 应用有一个默认的 logger，叫 “application”。当然，你也可以创建你自己的 logger。Play 库使用的 logger 名叫 “play”，还有一些第三方库会自己命名 logger。

日志级别

日志级别是用来区分日志消息的严重性的。当你写日志请求时，你需要指定严重程度，它会出现在产生的日志消息里。

以下是一组可用的日志级别，以严重性的降序列出。

- `OFF` - 关闭日志，不作为消息分类。
- `ERROR` - 运行时错误，或不可预料的情况。
- `WARN` - 用在一些不建议使用的 API 提示上，或是其它一些不可预料但又算不上错误的运行时情况。
- `INFO` - 用在你感兴趣的运行时事件上，比如应用启动和关闭时。
- `DEBUG` - 系统工作流程上的一些细节信息。
- `TRACE` - 大部分细节信息。

日志级别是用来配置 logger 和 appender 打印日志的阈值的。例如，如果你把 logger 的日志级别设置为 `INFO`，则会打印 `INFO` 及严重性更高的日志（ `INFO`， `WARN` 和 `ERROR` ），而忽略严重性比它低的日志（ `DEBUG`， `TRACE` ）。使用 `OFF` 会忽略所有的日志请求。

Appenders

日志 API 允许日志请求打印到一个或多个叫做 appender 的目标输出。appender 在配置中指定，可选的有：控制台，文件，数据库或其它输出。

appender 和 logger 组合可以帮助你路由及过滤日志信息。例如，你可以使用一个 appender 打印有用的信息用于分析，用另一个 appender 打印错误信息，用于运维团队的监控。

> 注意：想了解更多关于日志体系结构的信息，请移步：*Logback 文档*。

使用 Logger

首先导入 `Logger` 类及其伴生对象：

```
1. import play.api.Logger
```

默认 Logger

`Logger` 对象是默认的 logger，名为 “application”。你可以使用它来打印日志：

```
1. / Log some debug info
2. Logger.debug("Attempting risky calculation.")
3.
4. try {
5.   val result = riskyCalculation
6.
7.   // Log result if successful
8.   Logger.debug(s"Result=$result")
9. } catch {
10.   case t: Throwable => {
11.     // Log error with message and Throwable.
```

```
12.     Logger.error("Exception with riskyCalculation", t)
13.   }
14. }
```

使用 Play 默认的日志配置，上面的语句会产生类似下面的控制台输出：

```
1. [debug] application - Attempting risky calculation.
2. [error] application - Exception with riskyCalculation
3. java.lang.ArithmeticException: / by zero
4.     at
   controllers.Application$.controllers$Application$$riskyCalculation(Ap
    ~[classes/:na]
5.     at
   controllers.Application$$anonfun$test$1.apply(Application.scala:18)
   [classes/:na]
6.     at
   controllers.Application$$anonfun$test$1.apply(Application.scala:12)
   [classes/:na]
7.     at
   play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390)
    [play_2.10-2.3-M1.jar:2.3-M1]
8.     at
   play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390)
    [play_2.10-2.3-M1.jar:2.3-M1]
```

输出包含了日志级别（debug，error），logger 名称（application），消息，如果抛出异常的话，还会有堆栈跟踪信息。

创建你自己的 logger

也许你想在所有地方都用默认 logger，但这种使用方式并不好。用一个不同的名字创建你自己的 logger，这样配置起来会更灵活，可以过

滤你的日志输出，并且精确知道日志的来源。

你可以使用 `Logger.apply` 工厂方法来创建一个新 logger，它需要一个名字作为参数：

```
1. val accessLogger: Logger = Logger("access")
```

一个常用的策略是，为每个类配置一个不同的 logger，并且用类名来命名它。你可以使用另一个工厂方法，它接受一个类作为参数：

```
1. val logger: Logger = Logger(this.getClass())
```

日志模式

高效地使用 logger 可以帮助你达到许多目标：

```
1. import scala.concurrent.Future
2. import play.api.Logger
3. import play.api.mvc._
4.
5. trait AccessLogging {
6.
7.   val accessLogger = Logger("access")
8.
9.   object AccessLoggingAction extends ActionBuilder[Request] {
10.
11.     def invokeBlock[A](request: Request[A], block: (Request[A]) =>
    Future[Result]) = {
12.       accessLogger.info(s"method=${request.method}
    uri=${request.uri} remote-address=${request.remoteAddress}")
13.       block(request)
14.     }
15.   }
16. }
17.
18. object Application extends Controller with AccessLogging {
```

```
19.
20.   val logger = Logger(this.getClass())
21.
22.   def index = AccessLoggingAction {
23.     try {
24.       val result = riskyCalculation
25.       Ok(s"Result=$result")
26.     } catch {
27.       case t: Throwable => {
28.         logger.error("Exception with riskyCalculation", t)
29.         InternalServerError("Error in calculation: " +
    t.getMessage())
30.       }
31.     }
32.   }
33. }
```

这个例子使用了 action 组合 来定义 `AccessLoggingAction`，它会把日志打印到一个名为 “access” 的 logger。`Application` 控制器使用了这个 action，并且它使用了自己的 logger 来记录该应用里的事件。在配置中，你就可以将这些 logger 路由到不同的 appender，例如访问日志和应用日志。

如果你只想为指定 action 打印请求数据，那么上面的设计就足够了。想要打印所有的请求，则最好使用过滤器：

```
1.  import scala.concurrent.ExecutionContext.Implicits.global
2.  import scala.concurrent.Future
3.  import play.api.Logger
4.  import play.api.mvc._
5.  import play.api._
6.
7.  object AccessLoggingFilter extends Filter {
8.
9.    val accessLogger = Logger("access")
10.
```

```
11.   def apply(next: (RequestHeader) => Future[Result])(request:
    RequestHeader): Future[Result] = {
12.     val resultFuture = next(request)
13.
14.     resultFuture.foreach(result => {
15.       val msg = s"method=${request.method} uri=${request.uri}
    remote-address=${request.remoteAddress}" +
16.         s" status=${result.header.status}";
17.       accessLogger.info(msg)
18.     })
19.
20.     resultFuture
21.   }
22. }
23.
24. object Global extends WithFilters(AccessLoggingFilter) {
25.
26.   override def onStart(app: Application) {
27.     Logger.info("Application has started")
28.   }
29.
30.   override def onStop(app: Application) {
31.     Logger.info("Application has stopped")
32.   }
33. }
```

上面的代码中，我们在打印日志请求里加入了响应状态，当 `Future[Result]` 完成时会打印输出。注意，对于像应用开启和应用关闭这种事件，用全局对象来使用默认 `logger` 是一种明智的选择。

配置

详见日志配置。