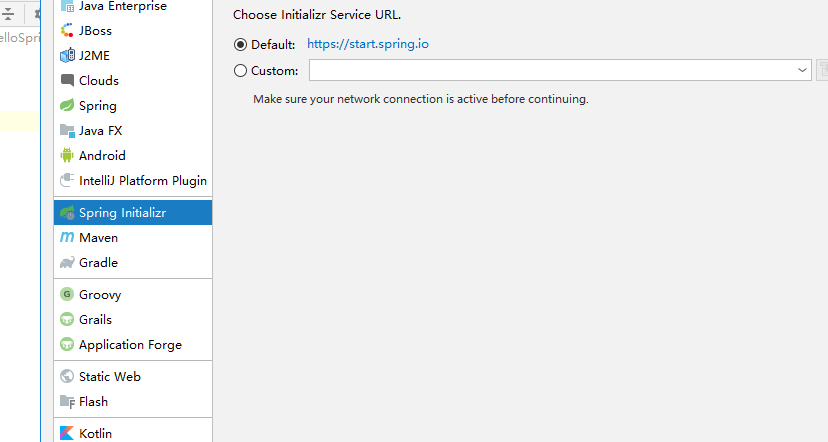
# 快速搭建spring应用

 使用 Spring Tool Suite 生成 Starter 代码；

第一个方案是使用“File | New | Spring Starter Project”导航菜单

 使用 IntelliJ IDEA 14.1，它对 Spring Boot 提供了良好的支持；

File-new-project-Spring Initializr



 借助 Spring 站点，从 http://start.Spring.io 上下载可配置的 ZIP 文件；

 使用到 http://start.Spring.io 站点的 curl 命令来达到相同的效果。

**$ curl http://start.Spring.io/starter.tgz \**

**-d name=masterSpringMvc \**

**-d dependencies=web \**

**-d language=java \**

**-d JavaVersion=1.8 \**

**-d type=gradle-project \**

**-d packageName=masterSpringMvc \**

**-d packaging=jar \**

**-d baseDir=app | tar -xzvf -**

1.5

GIT

如果你还不了解 Git 的话，我推荐下面的两个教程：  https://try.github.io，这是一个很好的交互式教程，可以引导你一的 Git 命令；  http://pcottle.github.io/learnGitBranching，这是一个很棒的教程，它将 树形的结构进行了可视化，它同时展现了 Git 的基本和高级功能。

1.5.1

Gradle

生成包装器的命令是 gradle wrapper：

**$ ./gradlew bootrun**

执行上面的命令将会运行一个嵌入式的 Tomcat，应用会位于它里面！

Spring Boot 插件将会产生一个胖（fat）JAR 文件，其中包含了项目的所有依赖。要构

建的话，只需输入：

**./gradlew build**

我们将会在“build/libs”目录下找到这个 JAR 文件。该目录下会包含两个文件，其中

一个为胖 JAR 包，名为 masterSpringMvc-0.0.1-SNAPSHOT.jar，另外一个是普通的 JAR 文

件，名为 masterSpringMvc-0.0.1-SNAPSHOT. jar.original，这个文件不包含任何的依赖。

1.5.2关注代码

需要注意的是@SpringBootApplication 注解，如果看一下这个注解的代码的话，就会发

现它实际上组合了 3 个其他的注解，也就是@Configuration、@EnableAutoConfiguration 和

@ComponentScan：

@ComponentScan 也是一个比较经典的注解，它会告诉 Spring 去哪里查找 Spring 组

件（服务、控制器等）。在默认情况下，这个注解将会扫描当前包以及该包下面的所有

子包。

在这里，比较新颖的是@EnableAutoConfiguration 注解，它会指导 Spring Boot 发挥其

魔力。如果你将其移除掉的话，就无法从 Spring Boot 的自动配置中收益了。

使用 Spring Boot 来编写 MVC 应用的第一步通常是在代码中添加控制器。将控制器放

到 controller 子包中，

HelloWorld例子

代码：

package masterSpringMvc.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

@Controller

public class HelloController {

@RequestMapping("/")

@ResponseBody

public String hello() {

return "Hello, world!";

}

}

1.6 幕后的 Spring Boot

如果你之前搭建过 Spring MVC 应用，那么可能已经习惯于编写相关的 XML 文件或 Java

注解配置类。

一般来讲，初始的步骤如下所示：

1．初始化 Spring MVC 的 DispatcherServlet；

2．搭建转码过滤器，保证客户端请求进行正确地转码；

3．搭建视图解析器（view resolver），告诉 Spring 去哪里查找视图，以及它们是使用哪

种方言编写的（JSP、Thymeleaf 模板等）；

4．配置静态资源的位置（CSS、JS）；

5．配置所支持的地域以及资源 bundle；

6．配置 multipart 解析器，保证文件上传能够正常工作；

7．将 Tomcat 或 Jetty 包含进来，从而能够在 Web 服务器上运行我们的应用；

8．建立错误页面（如 404）。

不过，Spring Boot 为我们处理了所有的事情。因为这些配置一般是与应用相关的，所

以你可以无限制地将它们进行组合。

1.6.1 分发器和 multipart 配置

接下来，让我们看一下在幕后到底发生了什么。

我们使用默认生成的Spring Boot 配置文件，并将其设置为debug 模式。在src/main/resources/

application.properties 中添加下面这一行：18 精通 Spring MVC 4

debug=true

现在，如果重新启动应用的话，就能看到 Spring Boot 的自动配置报告。它分为两部分：一

部分是匹配上的（positive matches），列出了应用中，所有的自动配置，另一部分是没有匹配

上的（negative matches），这部分是应用在启动的时候，需求没有满足的 Spring Boot 自动配置： 重新启动后日志。

现在，如果重新启动应用的话，就能看到 Spring Boot 的自动配置报告。它分为两部分：一

部分是匹配上的（positive matches），列出了应用中，所有的自动配置，另一部分是没有匹配

上的（negative matches），这部分是应用在启动的时候，需求没有满足的 Spring Boot 自动配置



DispatcherServletAutoConfiguration：

@Order(Ordered.HIGHEST\_PRECEDENCE)

@Configuration

@ConditionalOnWebApplication

@ConditionalOnClass(DispatcherServlet.class)

@AutoConfigureAfter(EmbeddedServletContainerAutoConfiguration.class)

这个文件中还包含了 Spring MVC 分发器 Servlet 和 multipart 解析器的典型配置。整个

Spring MVC 配置被拆分到了多个文件之中。

另外，值得一提的是，这些 bean 会遵循特定的规则，以此来检查是否处于激活状态。在

@Conditional(DefaultDispatcherServletCondition.class)条件满足的情况下，ServletRegistrationBean 函

数才会启用，这有些复杂，但是能够检查在你的配置中，是否已经注册了分发器 Servlet。

只有在满足@ConditionalOnMissingBean(name=DispatcherServlet.MULTIPART\_RESOLVER\_

BEAN\_NAME)条件的情况下，MultipartResolver 函数才会处于激活状态，例如，当我们自

己还没有注册的时候。

这意味着 Spring Boot 仅仅是基于常见的使用场景，帮助我们对应用进行配置。不过，

可以在任意的地方覆盖这些默认值，并声明自己的配置。

因此，通过查看 DispatcherServletAutoConfiguration，就了解了为什么我们已经拥有了

分发器 Servlet 和 multipart 解析器。

1.6.2 视图解析器、静态资源以及区域配置

另外一个密切相关的配置是 WebMvcAutoConfiguration，它声明了视图解析器、地域解

析器（localeresolver）以及静态资源的位置。

**视图解析器**的配置并没有什么特殊之处，这里真正有意思的是使用了配置属性，从而

允许用户对其进行自定义。

它的意思就是说“将会在用户的 application.properties 文件中查找两个变量，这两个变

量的名字是 spring.view.prefix 和 spring.view.suffix”。在配置中只需两行代码就能将视图解

析器搭建起来了，这是非常便利的。

资源位置的声明有点复杂，但是通过它，我们可以了解到以下两点：

 对带有“webjar”前缀的资源访问将会在类路径中解析。这样的话，我们就能使

用 Mavan 中央仓库中预先打包好的 JavaScript 依赖；

 我们的**静态资源**需要放在类路径中，并且要位于以下 4 个目录中的任意一个之中，24

“/META-INF/resources/”“/resources/”“/static/”或“/public/”。

在这个文件中，还专门有一部分用来声明地域管理：

@Bean

@ConditionalOnMissingBean(LocaleResolver.class)

@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")

public LocaleResolver localeResolver() {

return new FixedLocaleResolver(

StringUtils.parseLocaleString(this.mvcProperties.getLocale()));

}

默认的**地域解析器**只会处理一个地域，并且允许我们通过 spring.mvc.locale 配置属性来

进行定义。

1.7 错误与转码配置

还记得在没有添加控制器的时候，第一次启动应用吗？当时看到了一个有意思的

“Whitelabel Error Page”输出。

错误处理要比看上去更麻烦一些，尤其是在没有 web.xml 配置文件并且希望应用能够

跨 Web 服务器部署时更是如此。好消息是 Spring Boot 将会处理这些事情！让我们看一下

ErrorMvcAutoConfiguration：

这段配置都做了些什么呢？

 定义了一个 bean，即 DefaultErrorAttributes，它通过特定的属性暴露了有用的错误

信息，这些属性包括状态、错误码和相关的栈跟踪信息。

 定义了一个 BasicErrorController bean，这是一个 MVC 控制器，负责展现我们所看

到的错误页面。

 允许我们将 Spring Boot 的 whitelabel 错误页面设置为无效，这需要将配置文件

application.properties 中的 error.whitelable.enabled 设置为 false。

 我们还可以借助模板引擎提供自己的错误页面。例如，它的名字是 error.html，

ErrorTemplateMissingCondition 条件会对此进行检查。第 1 章 快速搭建 Spring Web 应用 27

在本书后面的内容中，我们将会看到如何恰当地处理错误。

至于转码的问题，非常简单的 HttpEncodingAutoConfiguration 将会负责处理相关的事宜，

这是通过提供 Spring 的 CharacterEncodingFilter 类来实现的。通过 spring.http.encoding.charset

配置，我们可以覆盖默认的编码（“UTF-8”），也可以通过 spring.http.encoding.enabled 禁用

这项配置。

1.8 嵌入式 Servlet 容器（Tomcat）的配置

默认情况下，Spring Boot 在打包和运行应用时，会使用 Tomcat 嵌入式 API（Tomcat

embedded API）。

我们来看一下 EmbeddedServletContainerAutoConfiguration：

可以将 Spring Boot 与 Tomcat、tc-server、Jetty 或者 Undertow 结合使用。服务器可以

很容易地进行替换，只需将 spring-boot-starter-tomcat JAR 依赖移除掉，并将其替换为 Jetty

或 Undertow 对应的依赖即可。如果你想这样做的话，请参考相关的文档。

对 Servlet 容器（Tomcat）的所有配置都会在 TomcatEmbeddedServletContainerFactory

中进行。尽管你应该读一下这个类，它为嵌入式 Tomcat 提供一个非常高级的配置（为其查

找文档会非常困难），但是在这里我们不会直接查看这个类。

1.8.1 HTTP 端口

通过在 application.properties 文件中定义 server.port 属性或者定义名为 SERVER\_PORT

的环境变量，我们可以修改默认的 HTTP 端口。

通过将该变量设置为−1，可以禁用 HTTP，或者将其配置为 0，这样的话，就会在随机

的端口上启动应用。对于测试，这是很便利的。

1.8.2 SSL 配置

配置 SSL 是一项很麻烦的事情，但是 Spring Boot 有一项很简单的解决方案。我们只需

一点属性就能保护服务器了：

**server.port = 8443**

**server.ssl.key-store = classpath:keystore.jks**

**server.ssl.key-store-password = secret**

**server.ssl.key-password = another-secret**

不过，为了使上面的例子运行起来，我们需要生成一个 keystore 文件。

我们将会在第 6 章中，深入介绍安全的可选方案。当然，我们还可以通过添加自己

EmbeddedServletContainerFactory 来进一步自定义 TomcatEmbeddedServletContainerFactory 的功

能。如果你希望添加多个连接器的话，这会是非常便利的，可以参考 http://docs.spring.io/

spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#howto-configure

ssl 来获取更多信息。

1.8.3 其他配置

在配置中，我们可以通过简单地声明@Bean 元素来添加典型的 Java Web 元素，如

Servlet、Filter 和 ServletContextListener。

除此之外，Spring Boot 还为我们内置了 3 项内容：

 在 JacksonAutoConfiguration 中，声明使用 Jackson 进行 JSON 序列化；

 在 HttpMessageConvertersAutoConfiguration 中，声明了默认的 HttpMessageConverter；

 在 JmxAutoConfiguration 中，声明了 JMX 功能。

我们将会在第 5 章中，更详细地了解 Jackson 的配置。关于 JMX 配置，我们可以在本30 精通 Spring MVC 4

地通过 jconsole 连接应用之后进行尝试，如图 1-9 所示。

图 1-9

通过将 org.springframework.boot:spring-boot-starter-actuator 添加到类路径下，我们可以

添加更多有意思的 MBean。我们甚至可以定义自己的 MBean，并通过 Jolokia 将其暴露为

HTTP。另一方面，我们也可以禁用这些端点，只需在配置中添加 spring.jmx.enabled=false

即可。

参考 http://docs.spring.io/spring-boot/docs/current/reference/

html/production-ready-jmx.html 了解更多细节。

1.9 小结

我们现在已经有了一个非常简陋的 Spring Web 应用，虽然我们没有对其进行什么配置，

但是它可以输出 RESTful JSON 的“Hello world”。我们已经看到过 Spring Boot 做了什么、第 1 章 快速搭建 Spring Web 应用 31

是如何实现的，也了解到如何重写默认的自动配置。

# 精通MVC 架构

在本章中，我们将会讨论 MVC 架构理念以及 Spring MVC 是如何实现这些理念的。

本章将会继续使用上一章的应用，并且会构建一些有意思的功能。目标是设计一个

简单的页面，在这个页面上用户可以根据特定的条件（criteria）查询 Tweet，并将其展

现出来。

为了实现该功能，我们将使用 Spring Social Twitter 项目，可以通过该地址 http://projects.

spring.io/spring-social-twitter/了解这个项目。

我们会让 Spring MVC 与一个现代化的模板引擎协作，这个引擎也就是 Thymeleaf，

并且还会试图理解这个框架的内部机制。引导用户在不同的视图间流转，最后，会借助

WebJars 和 Materialize（http://materializecss.com）让应用在外观上看起来更棒。

## 2.1 MVC 架构

大多数人对 MVC 这个缩写应该不会感到陌生。它代表的是模型（Model）、视图（View）

和控制器（Controller），它会将数据和展现层进行解耦，被视为构建用户界面的一种很流行

的方式。

自从在 Smalltalk 领域中提出这个理念，并在 Ruby on Rails 框架中采用之后，MVC 就

变得广受欢迎。

如图 2-1 所示，它的架构可以分为 3 层。

 模型：包含了应用中所需的各种展现数据。

 视图：由数据的多种表述所组成，它将会展现给用户。第 2 章 精通 MVC 架构 33

 控制器：将会处理用户的操作，它是连接模型和视图的桥梁。

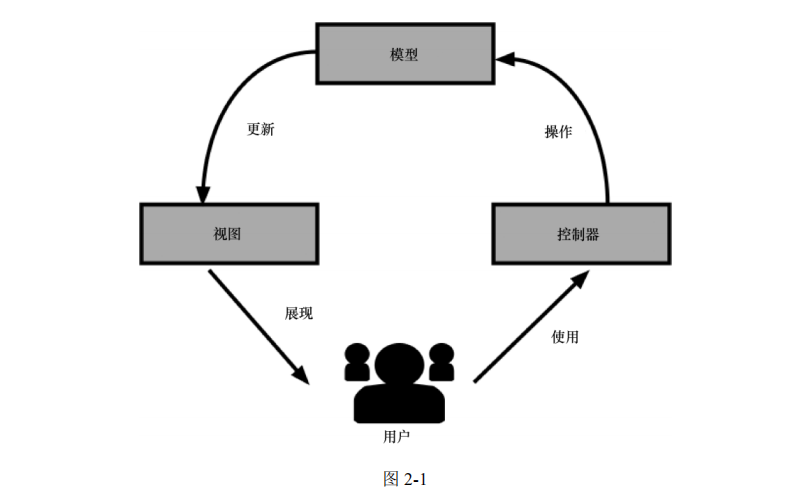


图 2-1

MVC 背后的理念是将视图与模型进行解耦，模型必须是自包含的并且与 UI 无关。这

样的话，基本上就可以实现相同的数据跨多个视图重用。其实，这些视图就是以不同的方

式来查看数据。通过钻取（Drill down）或使用不同的渲染器（HTML、PDF），可以很好地

阐述这一原则。

控制器会作为用户和数据的中间协调者，它的角色就是控制终端用户的可用行为，并

引导他们在应用的不同视图间跳转。

## 2.2 对 MVC 的质疑及其最佳实践

尽管 MVC 依然是当前设计 UI 的首选方案，但是随着它的流行，也有很多对它的批评。

实际上，大多数的批评都指向了该模式的错误用法。

2.2.1 贫血的领域模型

Eric Evans 编写过一本很有影响力的书，名为《领域驱动设计》（Domain Driven Design，

DDD）。在这本书中，定义了一组架构规则，能够指导我们更好地将业务领域集成到代34 精通 Spring MVC 4

码之中。

其中有一项核心的理念就是将面向对象的范式应用到领域对象之中。如果违背这一

原则的话，就会被称之为贫血的领域模型（Anemic Domain Model）。Martin Fowler 的博

客（http://www.martinfowler.com/bliki/AnemicDomainModel.html）对这一问题进行了很

好的定义。

贫血的领域模型通常来讲会具有如下的症状：

 模型是由简单老式的 Java 对象（plain old Java object，POJO）所构成的，只有

getter 和 setter 方法；

 所有业务逻辑都是在服务层处理的；

 对模型的校验会在本模型外部进行，例如在控制器中。

根据业务领域的复杂性不同，这可能是一种较差的实践方式。通常来讲，DDD 实践需

要付出额外的努力，将领域从应用逻辑中分离出来。

架构通常都是一种权衡，需要注意的是，设计 Spring 应用的典型方式往往会在这个过

程中导致系统在可维护性上变得较为复杂。

避免领域贫血的途径如下：

 服务层适合进行应用级别的抽象（如事务处理），而不是业务逻辑；

 领域对象应该始终处于合法的状态。通过校验器（validator）或 JSR-303 的校验注

解，让校验过程在表单对象中进行；

 将输入转换成有意义的领域对象；

 将数据层按照 Repository 的方式来实现，Repository 中会包含领域查询（例如参考

Spring Data 规范）；

 将领域逻辑与底层的持久化框架解耦；

 尽可能使用实际的对象，例如操作 FirstName 类而不是操作 String。

DDD 所涉及的内容远不止上述的规则：实体（Entity）、值类型（value type）、通用语

言（Ubiquitous Language）、限界上下文（Bounded Context）、洋葱架构（Onion Architecture）

以及防腐化层（anti corruption layer），我强烈建议你自行学习一下这些原则。就我们而言，

在构建 Web 应用的过程中，会努力遵循上述的指导原则。随着本书的推进，你会对这些关

-----------------------------------------启动sagan启动

Run the site locally from your IDE

Create a new run configuration with the class sagan.SiteApplication.

-Run the site locally from the command line

From the root directory of your sagan repository, run the following:

Tip: New to Gradle? Check out [Gradle tips](https://github.com/spring-io/sagan/wiki/Gradle-tips)!

./gradlew :sagan-site:bootRun

After a few seconds, you should see:

sagan.SiteApplication : Started SiteApplication

Now, go to [http://localhost:8080](http://localhost:8080/) and you should see homepage. Click around and explore a bit—you should find the site is identical in every way to what you see at [spring.io](http://spring.io/).

## 2.3 Spring MVC 1-0-1

在 Spring MVC 中，模型是由 Spring MVC 的 Model 或 ModelAndView 封装的简单 Map。 它可以来源于数据库、文件、外部服务等，这取决于你如何获取数据并将其放到模型中。与数 据层进行交互的推荐方式是使用 Spring Data 库：Spring Data JPA、Spring Data MongoDB 等。 有 10 多个与 Spring Data 相关的项目，推荐你查看一下 http://projects.spring.io/spring-data。

Spring MVC 的控制层是通过使用@Controller 注解来进行处理的。在 Web 应用中，控制器的角色是响应 HTTP 请求。带有@Controller 注解的类将会被 Spring 检索到，并且能够 有机会处理传入的请求。

通过使用@RequestMapping 注解，控制器能够声明它们会根据 HTTP 方法（如 GET 或

POST 方法）和 URL 来处理特定的请求。控制器就可以确定是在 Web 响应中直接写入内容， 还是将应用路由一个视图并将属性注入到该视图中。

纯粹的 RESTful 应用将会选择第一种方式，并且会在 HTTP 响应中直接暴露模型的

JSON 或 XML 表述，这需要用到@ResponseBody 注解。在 Web 应用中，这种类型的架构 通常会与前端 JavaScript 框架关联，如 Backbone.js、AngularJS 或 React。在这种场景中， Spring 应用只需处理 MVC 中的模型层。我们将会在第 4 章中学习这种架构。

在第二种方式中，模型会传递到视图中，视图会由模板引擎进行渲染，并写入到响应

之中。

视图通常会与某种模板方言关联，这种模板允许遍历模型中的内容，流行的模板方言

包括 JSP、FreeMarker 或 Thymeleaf。 混合式的方式则会在某些方面采用模板引擎与应用进行交互，并将视图层委托给前端

框架。

2.3 Spring MVC 1-0-1

在 Spring MVC 中，模型是由 Spring MVC 的 Model 或 ModelAndView 封装的简单 Map。它可以来源于数据库、文件、外部服务等，这取决于你如何获取数据并将其放到模型中。与数 据层进行交互的推荐方式是使用 Spring Data 库：Spring Data JPA、Spring Data MongoDB 等。

有 10 多个与 Spring Data 相关的项目，推荐你查看一下 http://projects.spring.io/spring-data。

Spring MVC 的控制层是通过使用@Controller 注解来进行处理的。在 Web 应用中，控制器的角色是响应 HTTP 请求。带有@Controller 注解的类将会被 Spring 检索到，并且能够 有机会处理传入的请求。

通过使用@RequestMapping 注解，控制器能够声明它们会根据 HTTP 方法（如 GET 或 POST 方法）和 URL 来处理特定的请求。控制器就可以确定是在 Web 响应中直接写入内容， 还是将应用路由一个视图并将属性注入到该视图中。

纯粹的 RESTful 应用将会选择第一种方式，并且会在 HTTP 响应中直接暴露模型的JSON 或 XML 表述，这需要用到@ResponseBody 注解。在 Web 应用中，这种类型的架构

通常会与前端 JavaScript 框架关联，如 Backbone.js、AngularJS 或 React。在这种场景中，

Spring 应用只需处理 MVC 中的模型层。我们将会在第 4 章中学习这种架构。

在第二种方式中，模型会传递到视图中，视图会由模板引擎进行渲染，并写入到响应

之中。

视图通常会与某种模板方言关联，这种模板允许遍历模型中的内容，流行的模板方言

包括 JSP、FreeMarker 或 Thymeleaf。

混合式的方式则会在某些方面采用模板引擎与应用进行交互，并将视图层委托给前端

框架。

2.5 Spring MVC 架构

让我们从这个新的“Hello World”页面后退一步，尝试去理解在这个 Web 应用中到底

发生了什么。为了做到这一点，需要跟踪浏览器所发送的 HTTP 请求的行程以及它是如何

从服务器端得到响应的。

2.5.1 DispatcherServlet

每个 Spring Web 应用的入口都是 DispatcherServlet。图 2-3 展现了 DispatcherServlet

的架构。

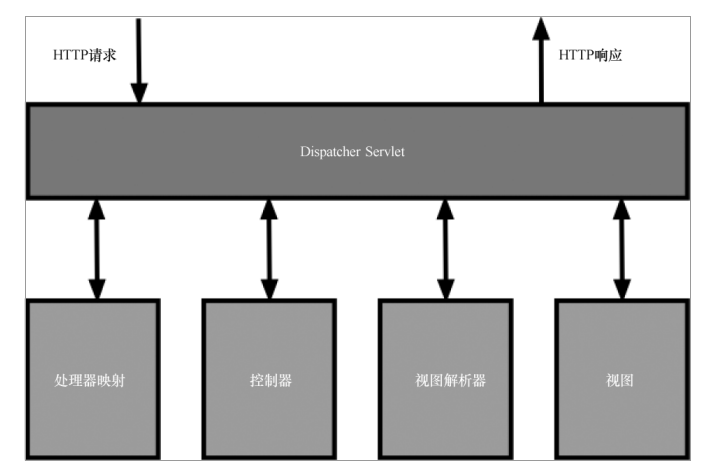


图 2-3

这个一个典型的 HttpServlet 类，它会将 HTTP 请求分发给 HandlerMapping。

HandlerMapping 会将资源（URL）与控制器关联起来。

控制器上对应的方法（也就是带有@RequestMapping 注解的方法）将会被调用。在这

个方法中，控制器会设置模型数据并将视图名称返回给分发器。

然后，DispatcherServlet 将会查询 ViewResolver 接口，从而得到对应视图的实现。

在样例中，ThymeleafAutoConfiguration 将会为我们搭建视图解析器。

通过查看 ThymeleafProperties 类，可以知道视图的默认前缀是“classpath:/templates/”，后缀

是“.html”。

这就意味着，假设视图名为 resultPage，那么视图解析器将会在类路径的 templates 目

录下查找名为 resultPage.html 的文件。

在我们的应用中，ViewResolver 接口是静态的，但是更为高级的实现能够根据请求的

头信息或用户的地域信息，返回不同的结果。

视图最终将会被渲染，其结果会写入到响应之中。

2.5.2 将数据传递给视图

第一个页面完全是静态的，其实并没有真正发挥出 Spring MVC 的威力。我们现在更进一

步，如果“Hello World”这个字符串不是硬编码的，而是来源于服务器，那该怎么实现呢？

你可能会问，还是显示这个无聊的“hello world”吗？是的，不过这种方式会开启更多

的可能性。现在，修改 resultPage.html 文件，让它展现来自模型中的信息：

Hello html

# 第 3 章 处理表单和复杂的 URL 映射

在本章中，将会构建一个基本信息（profile）页面。它的特性包括服务端和客户端的校

验，并且要上传一个文件作为基本信息的图片。我们会将这些信息存储到用户会话（session）

之中，为了保证应用有尽可能多的用户，还会将它翻译为多种语言。

3.1 基本信息页——表单

窗体顶端

<**form th:action="@{/profile}" method="post" class="col m8 s12  
offset-m2"**>  
 </**form**>

注意，“@{}”语法将会为资源构建完整的路径，它会将服务器上下文路径（在本例中，

也就是 localhost:8080）添加到它的参数上。

如果你希望学习 Java 8 date-time API 的更多信息，可以

参考 Oracle 提供的学习指南，地址是 https://docs.oracle.

com/javase/tutorial/datetime/TOC.html。

**Spring Boot** 中的日志

借助 Spring Boot，进行日志配置超级简单，只需将

logging.level.{package} = DEBUG 添加到 application.

properties 文件中，其中{package}是应用中某个类或包

的全限定名称。当然，你可以将 debug 换成自己任意想

要的日志级别。你也可以添加传统的日志配置，参考

http://docs.spring.io/spring-boot/docs/current/reference/

html/howto-logging.html 了解更多信息。

我们需要调试应用来了解发生了什么状况，那么在 application.properties 中添加如下这

行代码：

logging.level.org.springframework.web=DEBUG

这个类扩展了 WebMvcConfigurerAdapter，这是对 Spring MVC 进行自定义配置的一个

很便利的类。它提供了很多的扩展点，我们可以重写诸如 addFormatters()这样的方法来访

问这些扩展点。

在开发期，将信息源配置为每次都重新加载 bundle 是

非常便利的。添加如下的属性到 application.properties：

spring.messages.cache-seconds=0

0意味着每次都重新加载，而−1则代表着不进行重新加载。

在 Spring 中负责解析错误信息的类是 DefaultMessageCodesResolver。在进行输入域校

验的时候，这个类将会按照如下的顺序来尝试解析信息：

 编码+“.”+对象名+“.”+输入域；

 编码+“.”+输入域；

 编码+“.”+输入域类型；

 编码。

在前面的规则中，编码部分可能会是两种内容：第一种是注解类型，如 Size 或 Email；

第二种是异常码，如 typeMismatch。还记得我们因为数据格式不正确而得到的异常码吗？

它相关的异常码的确是 typeMismatch。

有了上面的这些消息，我们接下来让它更为具体。定义默认信息的最佳实践如下所示：

Size=the {0} field must be between {2} and {1} characters long

typeMismatch.java.util.Date = Invalid date format.

注意，这里的占位符，每个校验错误都有与之关联的一组参数。

声明错误信息的最后一种方式是直接在检验注解中定义错误信息，如下所示：

@Size(min = 2, message = "Please specify a valid twitter handle")

private String twitterHandle;

但是，这种方式的缺点在于它无法与国际化功能兼容。

### 3.3 国际化-修改地域

国际化，通常被称之为 i18n，指的是将应用程序设计为可以翻译成各种语言的

过程。第 3 章 处理表单和复杂的 URL 映射 75

这通常会涉及将翻译文本放到属性 bundle 中，并且要以目标地域作为后缀，例如

messages\_en.properties、messages\_en\_US.properties 和 messages\_fr.properties 文件。

属性 bundle 的解析过程是首先尝试最为具体的地域，如果无法找到的话，将会依次使

用备用的非具体地域。

例如，对于美国英语来说，如果想要从名为 x 的 bundle 中得到译文，那么应用程序首

先会从 x\_en\_US.properties 文件中进行查找，然后是 x\_en.properties 文件，最后会查找

x.properties 文件。

我们所要做的第一件事情就是将错误信息翻译为法语。将已有的 messages.properties 文件

重命名为 messages\_en.properties。

接下来，创建第二个 bundle，名为 messages\_fr.properties：

Spring 提供了这个接口的多个实现，如 FixedLocaleResolver。区域解析器非常简单，

我们可以通过一个属性来配置应用程序的地域，而且一旦定义之后，就不能进行修改了。

要配置我们应用的地域，只需添加如下的属性到 application.properties 文件中：

spring.mvc.locale=fr 76 精通 Spring MVC 4

这样，就会添加我们的法语检验信息。

如果我们看一下 Spring MVC 所提供的不同 LocaleResolver 接口实现，会发现它们如下

所示。

 FixedLocaleResolver：使用配置中固定的地域，一旦确定之后，不能发生变化。

 CookieLocaleResolver：允许在 Cookie 中检索和保存地域信息。

 AcceptHeaderLocaleResolver：根据用户浏览器所发送的 HTTP 头信息来查找地域。

 SessionLocaleResolver：在 HTTP 会话中查找和存储地域信息。

这些实现类能够涵盖很多的用户场景，但是在更为复杂的应用程序中，我们可以直接

实现一个 LocaleResolver，从而允许更复杂的逻辑，例如先从数据库中获取地域信息，如果

无法匹配的话，再使用浏览器中的地域。

国际化支持操作步骤：

1：支持Session中存放国际化数据

@Bean

public LocaleResolver localeResolver() {

return new SessionLocaleResolver();

}

2：创建国际化拦截器，并设置拦截器要拦截的保存存放语言的属性

@Bean  
**public** LocaleChangeInterceptor localeChangeInterceptor() {  
 LocaleChangeInterceptor localeChangeInterceptor = **new** LocaleChangeInterceptor();  
 localeChangeInterceptor.setParamName(**"lang"**);  
 **return** localeChangeInterceptor;  
}

3：添加注册拦截器

@Override  
**public void** addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(localeChangeInterceptor());  
}

4：页面添加链接请求

<**ul id="lang-dropdown"** >  
 <**li**><**a href="?lang=en\_US"**>English</**a**></**li**>  
 <**li**><**a href="?lang=fr"**>French</**a**></**li**>  
</**ul**>

### 3.4客户端校验

再补充一点，通过使用 HTML 5 的表单检验规范，如今实现客户端的校验已经非常容

易了。如果你的目标浏览器是 Internet Explorer 10 及以上的话，添加客户端校验只需要指定

正确的输入域类型，不要再将 type 属性设置为 text。

通过添加客户端校验，我们就可以预先校验表单，避免已知的不正确请求对服务器形成

过大的负载。关于客户端校验规范的更多信息可以参考 http://caniuse.com/#search=validation。

还存在其他类型的校验，请参考 http://www.the-art-of-web.com/html/html5-form-validation。

这种方式的缺点在于添加和移除口味的按钮也会触发校验。为了修正这个问题，我们

需要在默认的布局底部包含一个脚本，就放在 jQuery 声明的后面。

# 第 4 章 文件上传与错误处理

4.1 上传文件

文件上传操作步骤

1：新建文件操作的html页面文件（注意：**enctype="multipart/form-data"**）

2：后台操作新建参数为MultipartFile的文件上传方法，并配置参数RequestMapping,文件上传必须是post提交。

除了表单中的 enctype 属性以外，并没有太多值得关注的。文件将会通过 POST 方法发

送到 upload URL 上。我们将会创建一个新的控制器，它会位于 profile 包中，与 Profile

Controller 位于同级目录下：

在项目的根目录下创建 pictures 目录，并添加名为“.gitkeep”的空文件，从而保证能

够将其提交到 Git 上。

用户提交的文件将会以 MultipartFile 接口的形式注入到控制器中，这个接口提供了多

个方法，用来获取文件的名称、大小及其内容。

这个方法中，最令我们感兴趣的应该就是 getInputStream()，借助 IOUtils.copy

方法，将这个流复制到 fileOutputStream 上。将输入流复制到输出流的代码是很让人感觉乏

味的，所以将 Apache Utils 添加到类路径下，我们就可以非常便捷了（它是 tomcat-embedded

core.jar 文件的一部分）。

在这个类中，我们使用了 Spring Boot 的 ConfigurationProperties，这将会告诉 Spring Boot

以一种类型安全的方式，自动映射类路径下所发现的属性（默认情况下，位于

application.properties 文件中）。

注意，我们定义的 setter 方法接受“String”类型作为参数，这里最为有用的是我们

可以让 getter 返回任意的类型

窗体底端

窗体顶端

窗体底端