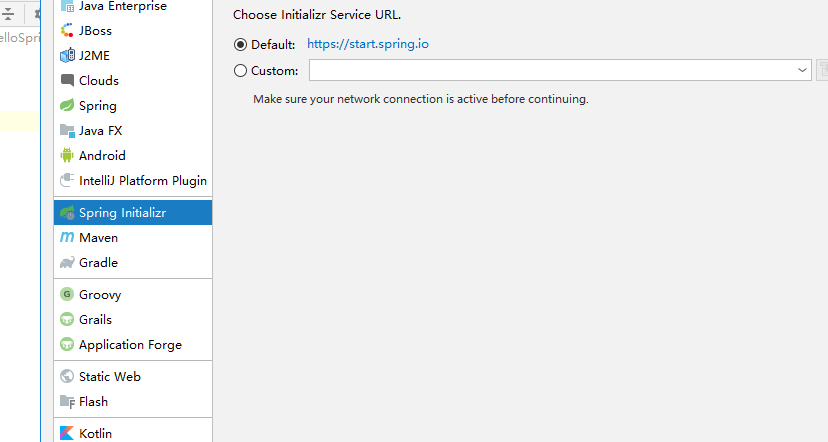
1. 快速搭建spring应用

 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。