Spring Boot

# 研究计划

1. **Spring boot介绍**
2. **Spring boot入门**
3. **Spring boot详解**
4. **Spring boot特性**
5. **与springmvc整合**
6. **与mybatis整合**
7. **Spring boot管理事务**
8. **文件上传**
9. **拦截器**
10. **异常处理**
11. **集成JPA**

# Springboot介绍

Spring Boot使开发独立的， 产品级别的基于Spring的应用变得非常简单， 你只需"just run"。 我们为Spring平台及第三方库提供开箱即用的设置， 这样你就可以有条不紊地开始。 多数Spring Boot应用需要很少的Spring配置。  
 你可以使用Spring Boot创建Java应用， 并使用 java -jar 启动它或采用传统的war部署方式。 我们也提供了一个运行"spring脚本"的命令行工具。

官网：<https://projects.spring.io/spring-boot/>

官方帮助文档：<https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/>

## 系统要求

Spring Boot 1.3.0.BUILD-SNAPSHOT 需要Java7和Spring框架4.1.3或以上。虽说java7也可以，但我们更建议使用java8。

## Servlet容器参考

下列内嵌容器支持开箱即用（out of the box） ：

|  |  |  |
| --- | --- | --- |
| 名称 | **Servlet**版本 | **Java**版本 |
| Tomcat 8 | 3.1 | Java 7+ |
| Tomcat 7 | 3.0 | Java 6+ |
| Jetty 9 | 3.1 | Java 7+ |
| Jetty 8 | 3.0 | Java 6+ |
| Undertow 1.1 | 3.1 | Java 7+ |

你也可以将Spring Boot应用部署到任何兼容Servlet 3.0+的容器。

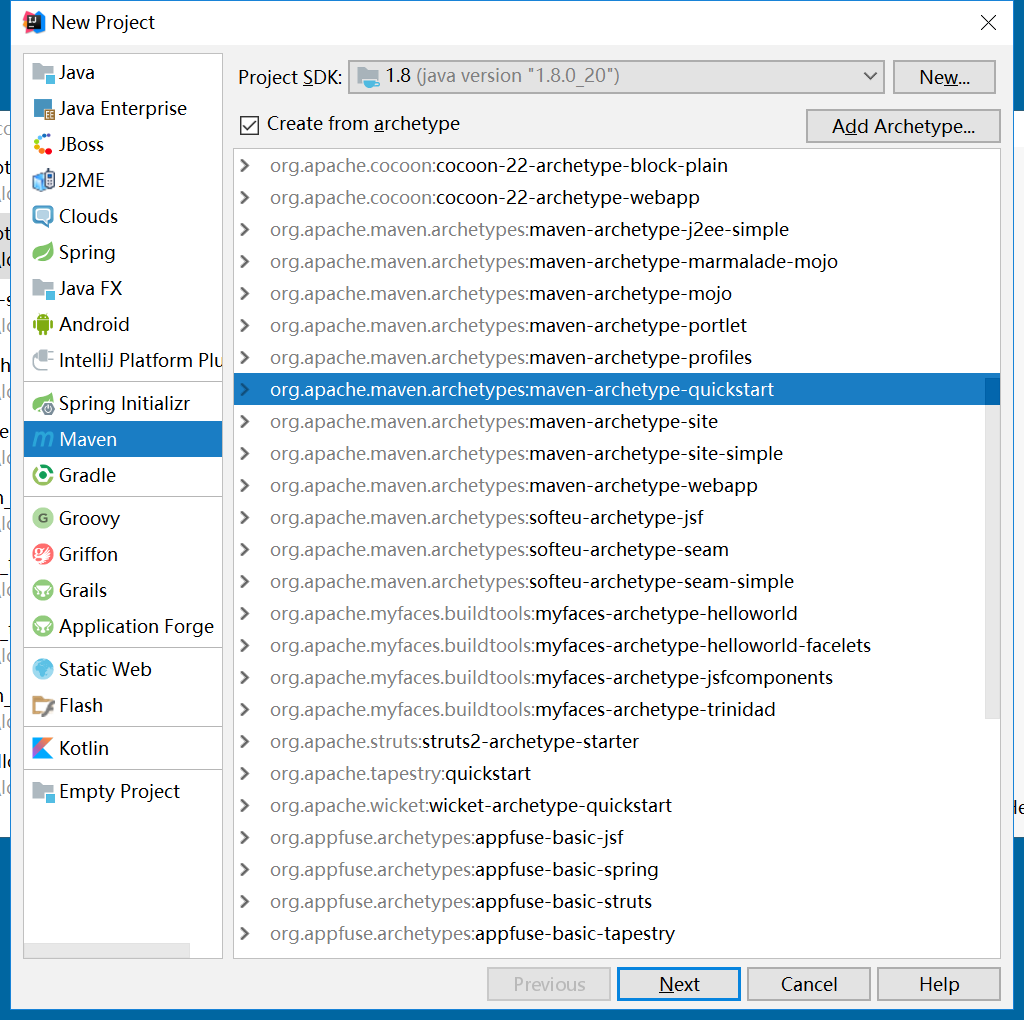
# 入门案例

Spring Boot兼容Apache Maven 3.2或更高版本，我们这里创建一个maven工程。

## 创建maven工程

创建步骤如下：

New Project>Maven>Project SDK(选择1.8)>勾选quickstart



填写项目坐标，如下图：

一直下一步到结束。

## 导入Springboot

Spring Boot依赖的groupId为 org.springframework.boot 。 通常你的Maven POM文件需要继承 spring-boot-starter-parent ，然后声明一个或多个“Starter POMs”依赖。

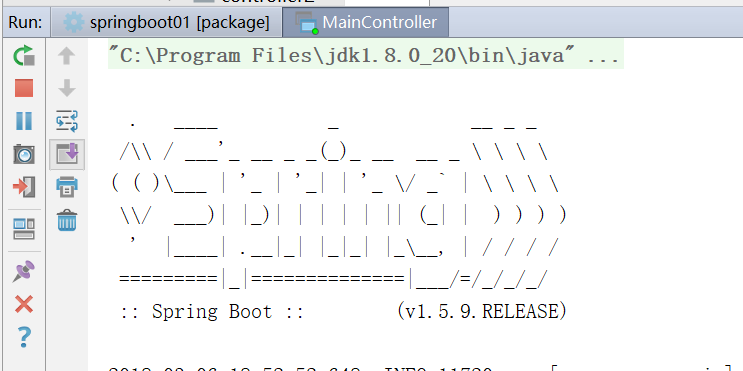
修改pom.xml文件：

|  |
| --- |
| <**project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>com.person</**groupId**>  <**artifactId**>springboot01</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  <**packaging**>jar</**packaging**>   *<!--继承springboot默认包-->* <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.9.RELEASE</**version**>  </**parent**>   <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  </**properties**>   <**dependencies**>  *<!--添加一个特殊的web应用依赖，多包关联依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  </**dependencies**>   *<!--建议配置编译器与资源加载插件-->* <**build**>  <**defaultGoal**>install</**defaultGoal**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.6.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-resources-plugin</**artifactId**>  <**version**>3.0.2</**version**>  <**configuration**>  <**encoding**>UTF-8</**encoding**>  </**configuration**>  </**plugin**>  </**plugins**>  </**build**>  </**project**> |

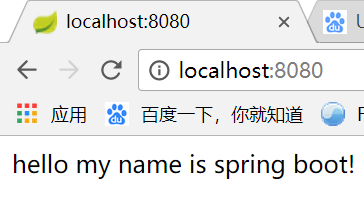
## 编写Controller代码

|  |
| --- |
| @EnableAutoConfiguration @RestController **public class** MainController {   @RequestMapping(**"/"**)  **public** String index(){  **return "hello my name is spring boot!"**;  }   **public static void** main(String[] args) {  SpringApplication.*run*(MainController.**class**);  }  } |

### 运行main方法，控制台输出：



### 输入http://localhost:8080执行请求测试



# SpringBoot使用讲解

## 神奇之处

* 你没有做任何的web.xml配置。
* 你没有做任何的sping mvc的配置,springboot为你做了。
* 你没有配置tomcat，springboot内嵌tomcat.

## 相关方法与注解说明

### main方法

我们的main方法通过调用run， 将业务委托给了Spring Boot的SpringApplication类。 SpringApplication将引导我们的应用， 启动Spring， 相应地启动被自动配置的Tomcat web服务器。 我们需要将 Example.class 作为参数传递给run方法来告诉SpringApplication谁是主要的Spring组件。

### @RestController注解

我们看到程序中出现了一个@RestController注解，这个注解有什么作用呢？我们看看它的源码

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented @Controller @ResponseBody **public** @**interface** RestController {  String value() **default ""**; } |

它上面加了@Controller和@ResponseBody注解，说明它同时拥有这2注解的功能，@Controller注解标识后，该类将会被spring管理，@ResponseBody注解标识后，响应数据可以是文本或者JSON数据类型。所以刚才我们return string的时候，可以直接输出text/html数据。当然@RestController注解属于springmvc的，并不属于springboot的。

### @RequestMapping注解的其它写法

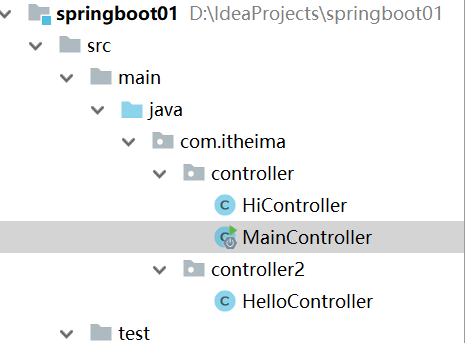
@GetMapping 等同于 @RequestMapping(method = RequestMethod.***GET***)  
@PostMapping 等同于 @RequestMapping(method = RequestMethod.***POST***)

### @EnableAutoConfiguration注解

这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于 spring-boot-starter-web 添加了Tomcat和Spring MVC， 所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。

#### 默认装配bean原则

我们看看如下案例，如图多创建HiController与HelloController



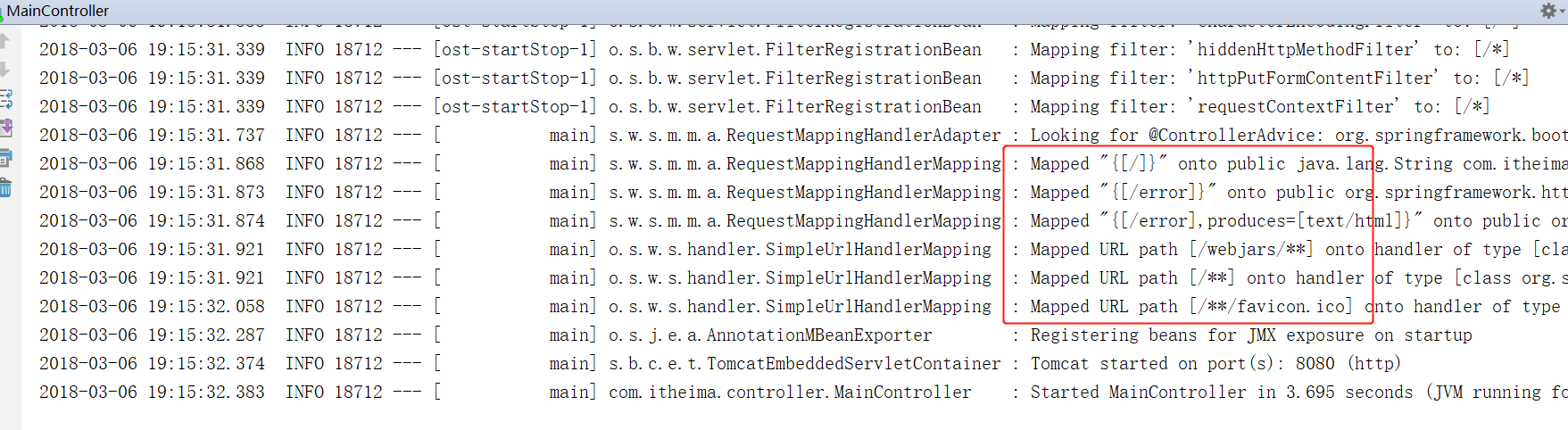
##### HiController代码：

@RestController  
**public class** HiController {  
  
 @GetMapping(**"hi"**)  
 **public** String hello(){  
 **return "hi,my name is Steven."**;  
 }  
}

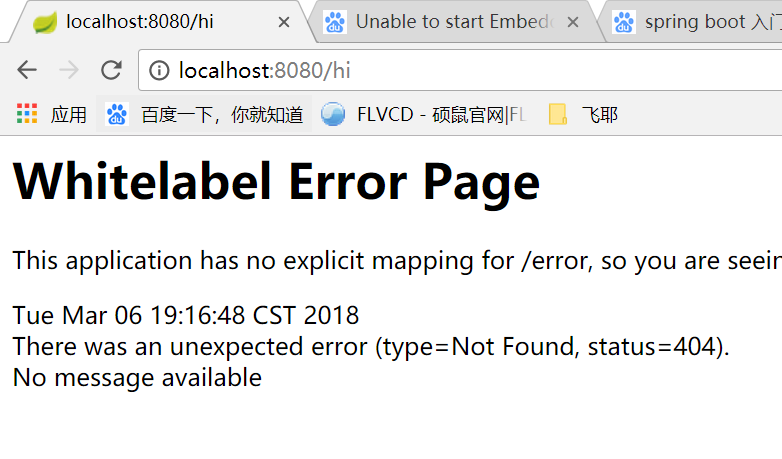
##### HelloController代码：

@RestController  
**public class** HelloController {  
  
 @GetMapping(**"hello"**)  
 **public** String hello(){  
 **return "hello,my name is Steven2."**;  
 }  
}

##### 创建完成，继续启动MainController的main方法结果：



发现hi与hello并没有被加载，如果浏览器访问会报404错误



##### 结论

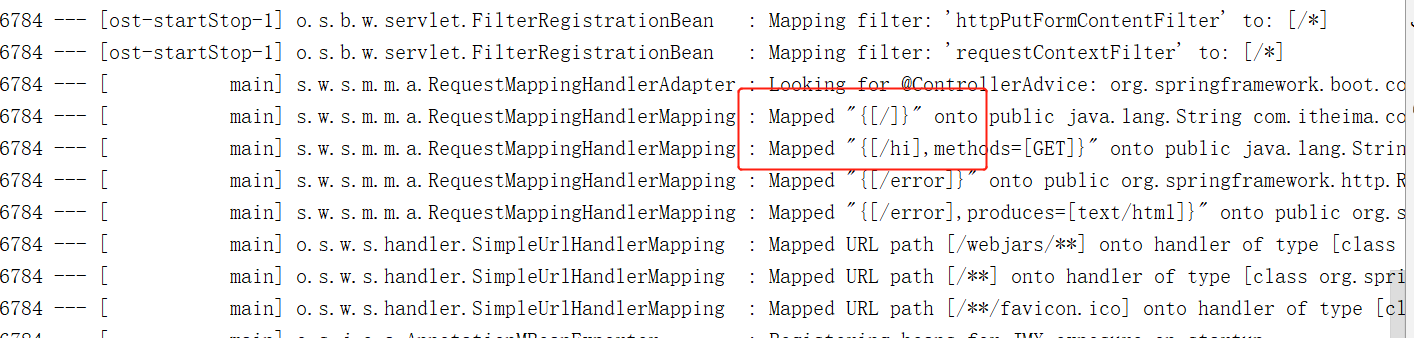
默认情况下@EnableAutoConfiguration只会加载main方法中SpringApplication.run载入的Example.class对象。

#### 解决同目录加载

在MainController中加入@ComponentScan注解，标识包扫描(只能扫描同级目录下的所有Controller)

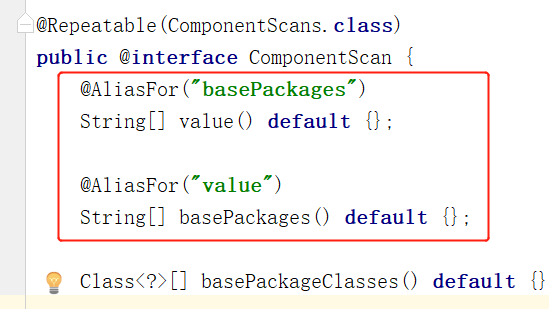


再次启动Main方法，结果HiController可以加载，HelloController不能加载



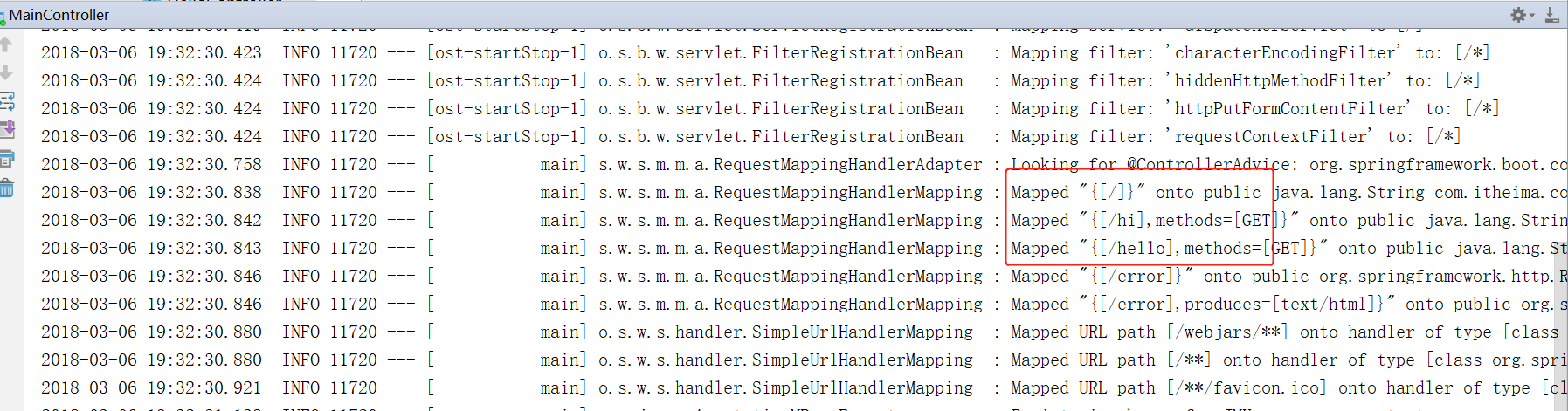
#### 解决不同目录加载

打开@ComponentScan原码发现有有个可配置的basepackage



修改@ComponentScan注解

再次启动，发现HiController与HelloController都能加载了



#### 定义main应用类

我们通常建议你将main应用类放在位于其他类上面的根包（root package） 中。 通常使用 @EnableAutoConfiguration 注解你的main类， 并且暗地里为某些项定义了一个基础“search package”。 例如， 如果你正在编写一个JPA应用，被 @EnableAutoConfiguration 注解的类所在包将被用来搜索 @Entity 项。

使用根包允许你使用 @ComponentScan 注解而不需要定义一个 basePackage 属性。 如果main类位于根包中， 你也可以使用 @SpringBootApplication 注解。

创建com.person.Application.java 文件与main 方法。这时，记得注释了MainController的@EnableAutoConfiguration与@ComponentScan。

|  |
| --- |
| */\*@EnableAutoConfiguration @ComponentScan\*/ //简写方式* @SpringBootApplication **public class** Application {   **public static void** main(String[] args) {  SpringApplication.*run*(Application.**class**, args);  } } |

##### @SpringBootApplication注解

很多Spring Boot开发者总是使用 @Configuration ， @EnableAutoConfiguration 和 @ComponentScan 注解他们的main类。 由于这些注解被如此频繁地一块使用（特别是你遵循以上最佳实践时） ， Spring Boot提供一个方便的 @SpringBootApplication 选择。该 @SpringBootApplication 注解等价于以默认属性使用 @Configuration ， @EnableAutoConfiguration 和 @ComponentScan 。

我们来看看@SpringBootApplication注解代码,如下代码，它分别继承了3个注解

1. @EnableAutoConfiguration
2. @ComponentScan
3. @SpringBootConfiguration

所以它同时具备了@ComponentScan、@EnableAutoConfiguration的功能。

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherited @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(  excludeFilters = {@Filter(  type = FilterType.CUSTOM,  classes = {TypeExcludeFilter.**class**} ), @Filter(  type = FilterType.CUSTOM,  classes = {AutoConfigurationExcludeFilter.**class**} )} ) **public** @**interface** SpringBootApplication {  //.....  } |

我们再来看看@SpringBootConfiguration继承了@Configuration注解功能

|  |
| --- |
| @Target({ElementType.TYPE}) @Retention(RetentionPolicy.RUNTIME) @Documented @Configuration **public** @**interface** SpringBootConfiguration { } |

## tomcat配置

当我们创建一个springboot项目的时候，系统默认会为我们在src/main/java/resources目录下创建一个application.properties。

关于Tomcat的所有属性都在org.springframework.boot.autoconfigure.web.ServerProperties配置类中做了定义，我们只需要在application.properties配置属性做配置即可。通用的Servlet容器配置都以”server”作为前缀，而Tomcat特有配置都以”server.tomcat“作为前缀。下面举些常用的例子：

*#-----begin配置配置Servlet容器-----  
#配置程序端口,默认为8080***server.port**=**8081***#用户会话session过期时间,以秒为单位***server.session-timeout**=**1800***#配置访问路径,默认为/***server.context-path**=**/***#-----end配置配置Servlet容器-----  
  
#-----begin配置配置tomcat-----  
#配置Tomcat编码，默认为UTF-8***server.tomcat.uri-encoding**=**UTF-8***#-----end配置配置tomcat-----*

更为详细的Servlet容器配置以及Tomcat配置，请查看官网文档。<https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/>

## Spring Beans和依赖注入

你可以自由地使用任何标准的Spring框架技术去定义beans和它们注入的依赖。 这里经常使用 @ComponentScan 注解搜索beans， 并结合 @Autowired 构造器注入。

如果使用上面建议的结构组织代码（将应用类放到根包下） ， 你可以添加 @ComponentScan 注解而不需要任何参数。 你的所有应用程序组件（@Component , @Service , @Repository , @Controller 等） 将被自动注册为Spring Beans。

下面做一个Controller注入Service注入dao实例的例子：

User代码

**package** com.person.pojo;  
**public class** User {  
 **private** Long **id**;  
 **private** String **name**;  
 **private** Integer **age**;  
 *//省略了get..与set..*

}

UserController代码

|  |
| --- |
| **package** com.person.controller;  @RestController @RequestMapping(value = **"user"**) **public class** UserController {  @Autowired  **private** UserService **userService**;   @RequestMapping(value = **"queryUser"**)  **public** User list(){  *//调用了userService，这里userService通过@Autowired注入进来* User user = **userService**.queryUser();  **return** user;  } } |

UserService接口

|  |
| --- |
| **package** com.person.service; **public interface** UserService {   User queryUser(); } |

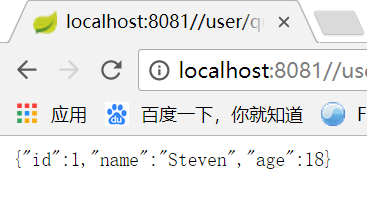
UserServiceImpl实现类

|  |
| --- |
| **package** com.person.service.impl; @Service **public class** UserServiceImpl **implements** UserService {   @Autowired  **private** UserDao **userDao**;   @Override  **public** User queryUser() {  **return userDao**.queryUser();  } } |

UserDao代码

**package** com.person.dao;  
@Repository  
**public class** UserDao {  
 **public** User queryUser(){  
 User user = **new** User();  
 user.setId(1L);  
 user.setName(**"Steven"**);  
 user.setAge(18);  
 **return** user;  
 }  
}

启动Application.main方法，浏览器输入：[http://localhost:8081//user/queryUser](http://localhost:8081/user/queryUser)



# springboot特性

## 配置文件读取

### Application配置文件

application.properties提供自定义属性的支持，这样我们就可以把一些常量配置在这里

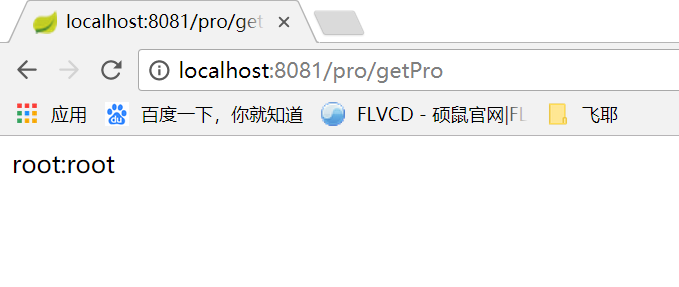
### 读取properties文件

在application.properties中添加如下数据

|  |
| --- |
| *#配置自定义属性* **jdbc.name**=**root jdbc.pass**=**root** |

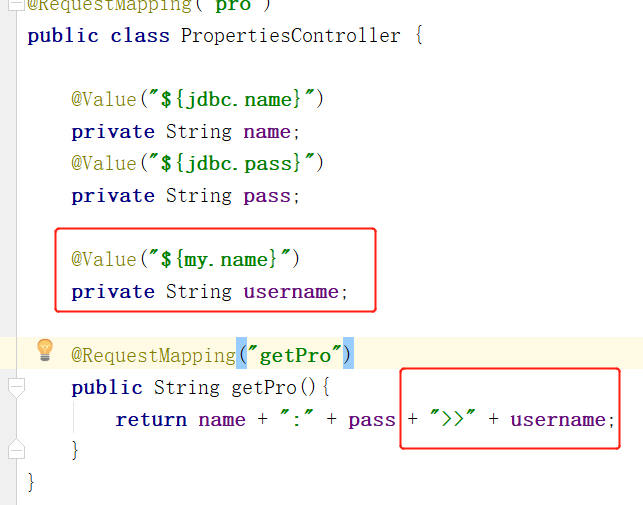
然后直接在要使用的地方通过注解@Value("${jdbc.name}")取出

@RestController  
@RequestMapping(**"pro"**)  
**public class** PropertiesController {  
  
 @Value(**"${jdbc.name}"**)  
 **private** String **name**;  
 @Value(**"${jdbc.pass}"**)  
 **private** String **pass**;  
  
 @RequestMapping(**"getPro"**)  
 **public** String getPro(){  
 **return name** + **":"** + **pass**;  
 }  
}



### 中文乱码问题

*#配置自定义属性***jdbc.name**=**root  
jdbc.pass**=**root  
  
my.name**=**赵云**





解决办法：

#### IDEA

设置 File Encodings的Transparent native-to-ascii conversion为true，具体步骤如下：

依次点击

File -> Settings -> Editor -> File Encodings

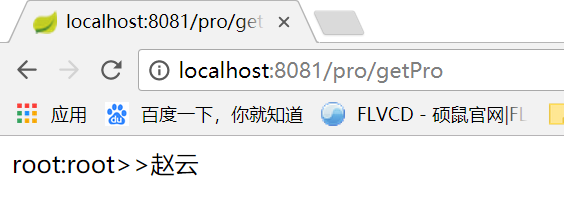
将Properties Files (\*.properties)下的Default encoding for properties files设置为UTF-8，将Transparent native-to-ascii conversion前的勾选上。

#### eclipse

[在eclipse中安装properties插件PropertiesEditor及设置(附图)，ASCII码转换成中文](http://www.cnblogs.com/zdz8207/p/java-eclipse-properties.html)

推荐使用里面的离线安装。安装完成PropertiesEditor 插件后，使用该编辑器重新编辑属性文件中的中文，然后重新运行程序，发现读取的中文配置文件不是乱码了。

修改配置后，结果如下：



### 将配置文件的属性赋给实体类

当我们有很多配置属性的时候，这时我们会把这些属性作为字段来创建一个javabean，并将属性值赋予给他们,比如：

**my.name**:**赵云  
my.age**:**21  
my.number**:**${random.int}  
my.uuid**:**${random.uuid}  
my.max**:**${random.int(10)}  
my.value**:**${random.value}  
my.greeting**:**hi,i'm ${my.name}**

其中配置文件中用到了${random} ，它可以用来生成各种不同类型的随机值。

怎么讲这些属性赋于给一个javabean 呢，首先创建一个javabean ：

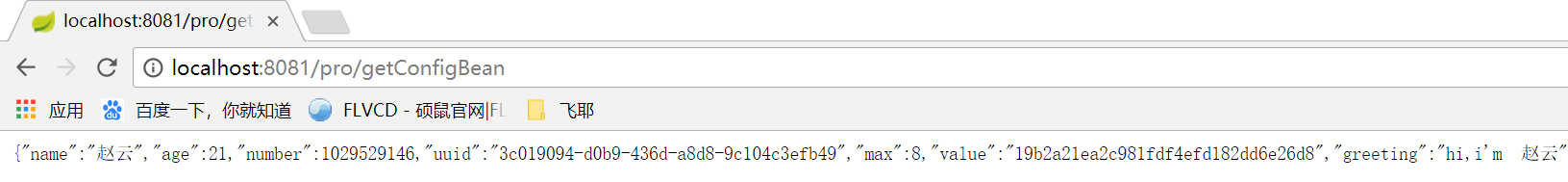
@ConfigurationProperties(prefix = **"my"**)  
@Component  
**public class** ConfigBean {  
  
 **private** String **name**;  
 **private int age**;  
 **private int number**;  
 **private** String **uuid**;  
 **private int max**;  
 **private** String **value**;  
 **private** String **greeting**;  
  
 *//省略了getter setter....*}

注意：需要加个注解@ConfigurationProperties并加上它的prrfix。同时加上@Component以标记为bean。

PropertiesController加入代码

@Autowired  
**private** ConfigBean **configBean**;  
  
@RequestMapping(**"getConfigBean"**)  
**public** ConfigBean getConfigBean(){  
 **return configBean**;  
}

启动springboot运行结果如下：



### 属性文件的另一种配置方式yml

在 spring boot 中，有两种配置文件，一种是application.properties,另一种是application.yml,两种都可以配置spring boot 项目中的一些变量的定义，参数的设置等。下面来说说两者的区别。

application.properties 配置文件在写的时候要写完整，如：

*#-----begin配置配置Servlet容器-----  
#配置程序端口,默认为8080***server.port**=**8081***#用户会话session过期时间,以秒为单位***server.session-timeout**=**1800***#配置访问路径,默认为/***server.context-path**=**/***#-----end配置配置Servlet容器-----  
  
#-----begin配置配置tomcat-----  
#配置Tomcat编码，默认为UTF-8***server.tomcat.uri-encoding**=**UTF-8***#-----end配置配置tomcat-----*

在yml文件中配置的话，写法如下：

*#-----begin配置配置Servlet容器-----***server:** *#配置程序端口,默认为8080* **port:** 8081  
 **session:** *#用户会话session过期时间,以秒为单位* **timeout:** 1800  
 *#配置访问路径,默认为/* **context-path:** /  
  
 *#-----begin配置配置tomcat-----  
 #配置Tomcat编码，默认为UTF-8* **tomcat:  
 uri-encoding:** utf-8  
 *#-----end配置配置tomcat-----*

yml文件在写的时候层次感强，而且少写了代码。所以现在很多人都使用yml配置文件。

### 自定义属性文件名的使用

有时我们不愿意把配置都写到application配置文件中，写在一起不便于维护，这时需要我们自定义配置文件，比如新建my.yml，我们把ConfigBean的属性分离出来。

步骤：把application.properties相关节点删除，新建my.yml，内容如下 ：

**my:  
 name:** 赵云  
 **age:** 21  
 **number:** ${random.int}  
 **uuid:** ${random.uuid}  
 **max:** ${random.int(10)}  
 **value:** ${random.value}  
 **greeting:** hi,i'm ${name}

注意属性与值之间一定要有空格

那么抽取yml后，怎么将这个配置文件信息赋予给一个javabean呢？

*//@ConfigurationProperties(prefix = "my")*@ConfigurationProperties  
@PropertySource(**"classpath:my.yml"**)  
@Component  
**public class** ConfigBean {

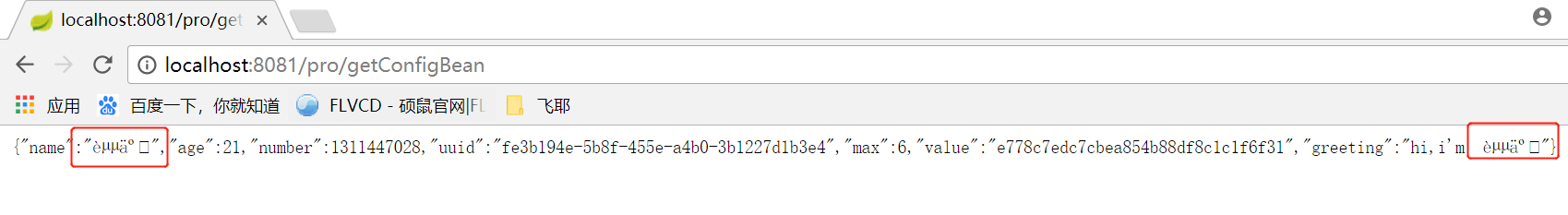
//省略其它内容

}

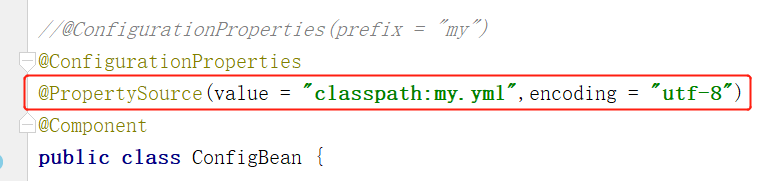
修改的点：

1. 把ConfigurationProperties的prefix属性去掉，或者设置为你pojo所在的包下
2. 加入@PropertySource(**"classpath:my.yml"**)，告诉springboot如果加载属性文件

启动springboot运行结果如下：

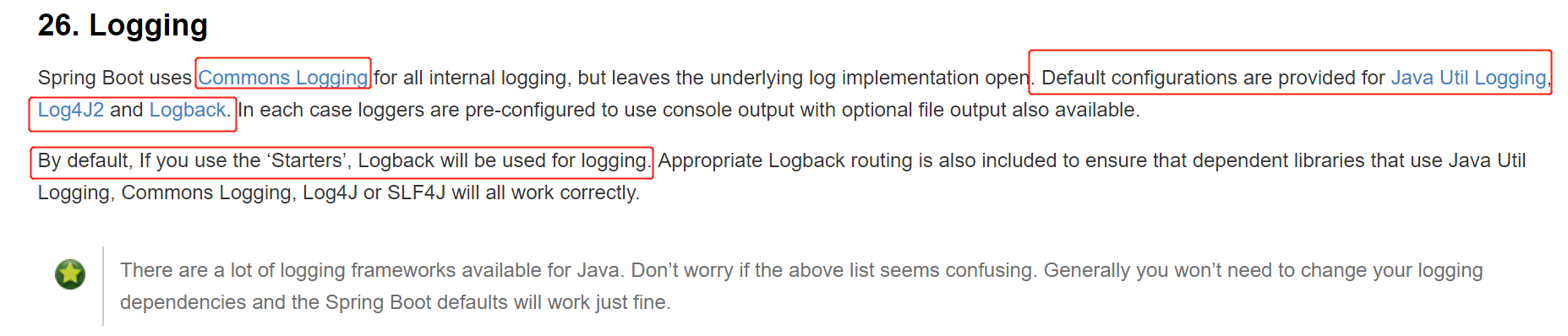


#### 新的乱码问题，解决方案(修改引入文件的编码方式)：



代码：@PropertySource(value = **"classpath:my.yml"**,encoding = **"utf-8"**)

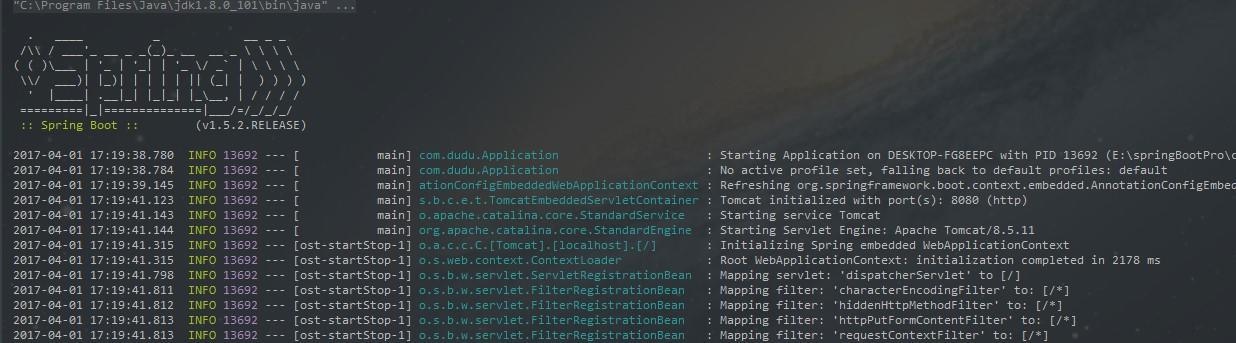
## 日志文件配置



大家请先简单看下这篇英文的官方文档，文中有说 SpringBoot 内部日志系统使用的是 Commons Logging 并且 SpringBoot 给 JDKLogging , Log4j2, Logback 都提供了默认配置，并且如果使用了 Starters ，那么默认使用 Logback 。

简单的说，只要你的 pom 文件中使用了 spring-boot-starter 就代表着你使用了 SpringBoot 的 Starters 。相信各位玩 SpringBoot 的朋友肯定是看到了自己的 pom 文件中有这个了，因为，Starters（启动器）是 SpringBoot 最核心的部件之一，没有了启动器， SpringBoot 就几乎废掉了。

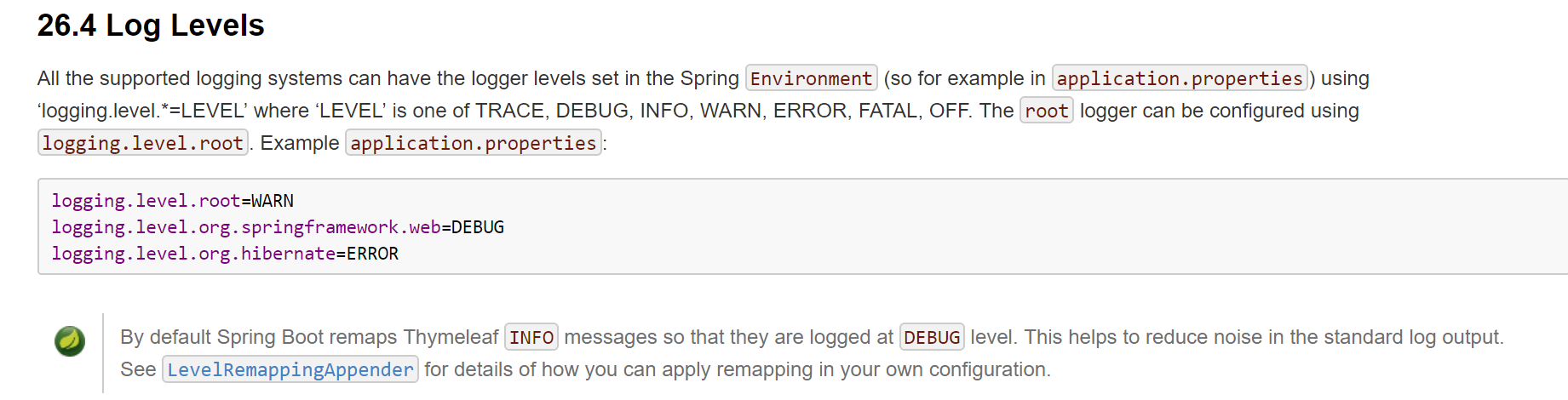
默认情况下，Spring Boot会用Logback来记录日志，并用INFO级别输出到控制台。在运行应用程序和其他例子时，你应该已经看到很多INFO级别的日志了。



从上图可以看到，日志输出内容元素具体如下：

* 时间日期：精确到毫秒
* 日志级别：ERROR, WARN, INFO, DEBUG or TRACE
* 进程ID
* 分隔符：— 标识实际日志的开始
* 线程名：方括号括起来（可能会截断控制台输出）
* Logger名：通常使用源代码的类名
* 日志内容

### 日志输出级别配置：



官方文档中有提到， SpringBoot 的 Logging 配置的级别有7个：

日志级别从低到高分为：

TRACE < DEBUG < INFO < WARN < ERROR < FATAL。

如果设置为 WARN ，则低于 WARN 的信息都不会输出

配置格式：

logging.level.\*=LEVEL

举例：

*#日志配置  
#root日志以WARN级别输出***logging.level.root**=**WARN***#springframework.web日志以DEBUG级别输出***logging.level.org.springframework.web**=**DEBUG***#hibernate日志以ERROR级别输出***logging.level.org.hibernate**=**ERROR**

在进行了这样的配置后，就可以在控制台打印Log信息了！

### 使用Log4j2的指定配置文件实现更高级的日志配置

从官方文档中我们看到springboot支持log4j2输出日志，我们习惯也是使用log4j，所以接下来我们使用下log4j来输出我们的日志。

注意：高版本的springboot(1.4以上)不再对log4j进行支持，所以新版本只能用log4j2。

#### 步骤-更改pom文件

在创建SpringBoot 工程时，我们引入了 spring-boot-starter，其中包含了 spring-boot-starter-logging ，该依赖内容就是 SpringBoot 默认的日志框架 Logback ，所以我们在引入 log4j 之前，需要先排除该包的依赖，再引入 log4j2 的依赖。

<**dependencies**>  
 *<!--添加一个特殊的web应用依赖，多包关联依赖-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 *<!--排除Logback -->* <**exclusions**>  
 <**exclusion**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-logging</**artifactId**>  
 </**exclusion**>  
 </**exclusions**>  
 </**dependency**>  
  
 *<!--加入log4j支持-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-log4j2</**artifactId**>  
 </**dependency**>  
</**dependencies**>

#### 创建log4j2.xml

在 classpath 的 resources下新建 log4j2.xml文件，这里简单示例一下：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**configuration**>  
 <**appenders**>  
 *<!--配置控制台输出-->* <**Console name="Console" target="SYSTEM\_OUT"**>  
 <**ThresholdFilter level="trace" onMatch="ACCEPT" onMismatch="DENY"** />  
 <**PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"** />  
 </**Console**>  
 *<!--配置文件输出-->* <**File name="log" fileName="D:/logs/test.log" append="false"**>  
 <**PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"** />  
 </**File**>  
 <**RollingFile name="RollingFile" fileName="D:/logs/spring.log" filePattern="log/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log"**>  
 <**PatternLayout pattern="%d{yyyy-MM-dd 'at' HH:mm:ss z} %-5level %class{36} %L %M - %msg%xEx%n"** />  
 <**SizeBasedTriggeringPolicy size="50MB"** />  
 </**RollingFile**>  
 </**appenders**>  
  
 <**loggers**>  
 *<!--配置日志输出级别-->* <**root level="debug"**>  
 <**appender-ref ref="RollingFile"** />  
 <**appender-ref ref="Console"** />  
 </**root**>  
 </**loggers**>  
</**configuration**>

# Springmvc应用

Spring Boot非常适合开发web应用程序。 你可以使用内嵌的Tomcat， Jetty或Undertow轻轻松松地创建一个HTTP服务器。 大多数的web应用都使用spring-boot-starter-web模块进行快速搭建和运行。

Spring Web MVC框架（通常简称为"Spring MVC"） 是一个富"模型， 视图， 控制器"的web框架。 Spring MVC允许你创建特定的@Controller或@RestController beans来处理传入的HTTP请求。 使用@RequestMapping注解可以将控制器中的方法映射到相应的HTTP请求

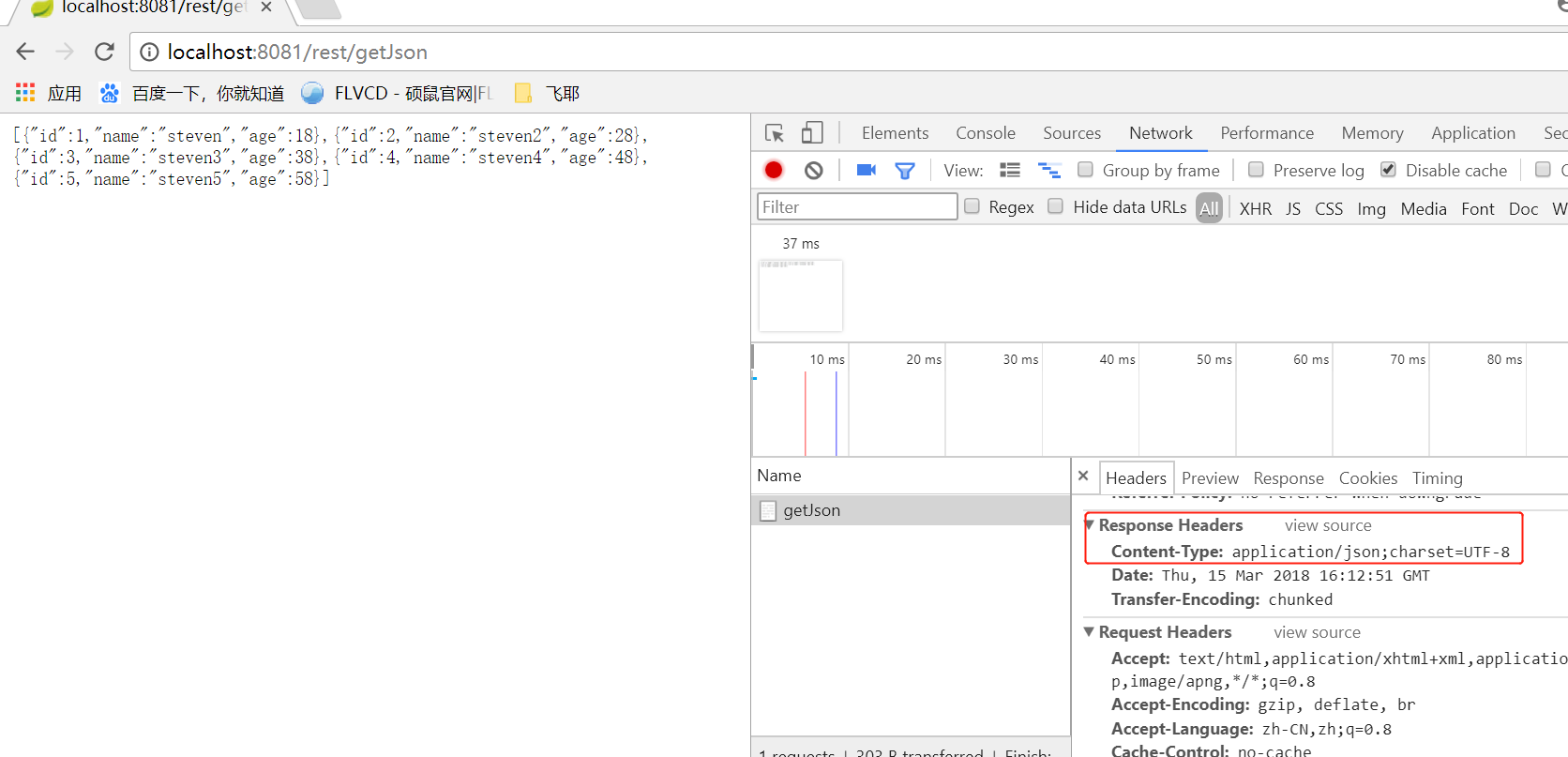
## JSON REST实现

在Spring Boot应用中， 任何Spring @RestController 默认应该渲染为JSON响应， 只要classpath下存在Jackson2。 例如：

|  |
| --- |
| @RestController @RequestMapping(**"rest"**) **public class** RestWebServiceController {  @RequestMapping(**"getJson"**)  **public** List<User> getUserList(){  List<User> users = **new** ArrayList<>();  users.add(**new** User(1L, **"steven"**, 18));  users.add(**new** User(2L, **"steven2"**, 28));  users.add(**new** User(3L, **"steven3"**, 38));  users.add(**new** User(4L, **"steven4"**, 48));  users.add(**new** User(5L, **"steven5"**, 58));  **return** users;  } } |

响应结果：

http://localhost:8081/rest/getJson



## XML REST实现

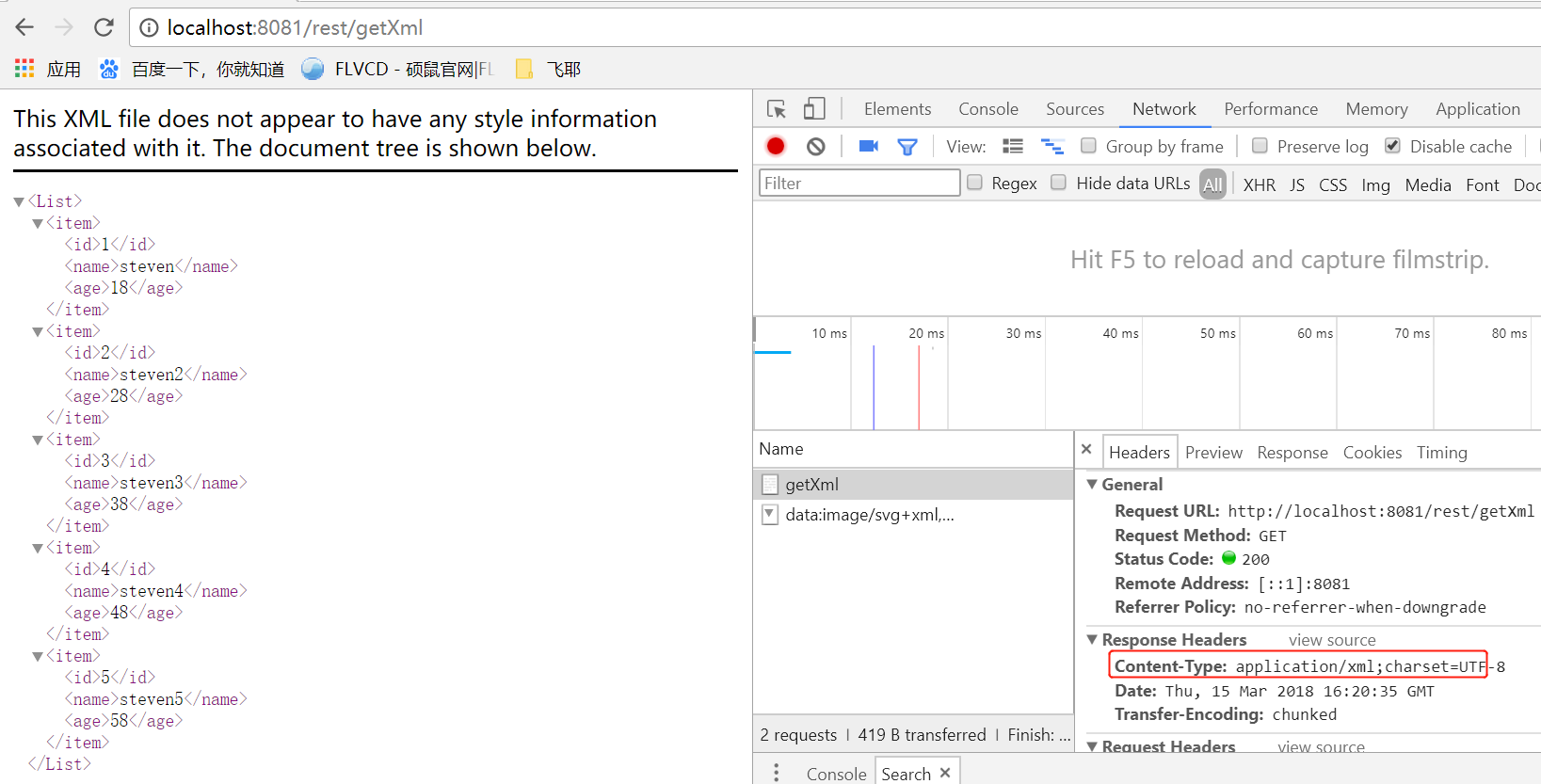
如果classpath下存在Jackson XML扩展（jackson-dataformat-xml） ， 它会被用来渲染XML响应， 示例和JSON的非常相似。想要使用它， 只需为你的项目添加以下的依赖。

|  |
| --- |
| *<!--Jackson XML扩展包-->* <**dependency**>  <**groupId**>com.fasterxml.jackson.dataformat</**groupId**>  <**artifactId**>jackson-dataformat-xml</**artifactId**> </**dependency**> |

在RestWebServiceController加入代码实现：

|  |
| --- |
| @RequestMapping(value = **"getXml"**,produces = MediaType.***APPLICATION\_XML\_VALUE***) **public** List<User> getXml(){  List<User> users = **new** ArrayList<>();  users.add(**new** User(1L, **"steven"**, 18));  users.add(**new** User(2L, **"steven2"**, 28));  users.add(**new** User(3L, **"steven3"**, 38));  users.add(**new** User(4L, **"steven4"**, 48));  users.add(**new** User(5L, **"steven5"**, 58));  **return** users; } |

测试结果：

http://localhost:8081/rest/getXml

## SpringMVC视图解析器配置-jsp

注意：jsp需要跑在web工程中，所以要测试这个功能需要新建webapp骨架工程测试。这里直接改项目属性无效，原因不详。

由于springboot已经不推荐使用jsp了，如果想使用jsp，需要手动引入相关依赖包。

|  |
| --- |
| *<!-- jsp -->* <**dependency**>  <**groupId**>org.apache.tomcat.embed</**groupId**>  <**artifactId**>tomcat-embed-jasper</**artifactId**>  *<!-- <scope>provided</scope> -->* </**dependency**>  *<!-- servlet 依赖 -->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**>  *<!-- <scope>provided</scope> -->* </**dependency**>  *<!-- 添加 JSTL 支持 -->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>jstl</**artifactId**> </**dependency**> |

视图解析器配置

|  |
| --- |
| *#视图解析器* **spring.mvc.view.prefix**=**/WEB-INF/views/ spring.mvc.view.suffix**=**.jsp** |

加上Controller代码测试：

@Controller  
@RequestMapping(**"mvc"**)  
**public class** MvcController {  
  
 @GetMapping(**"hello"**)  
 **public** String toJsp(Model model){  
 model.addAttribute(**"msg"**, **"steven is good."**);  
 System.***out***.println(**"hello,tojsp"**);  
 **return "hello"**;  
 }  
}

再添加一个jsp页面接收就可。

# 基于配置文件集成MyBatis

## 添加依赖

|  |
| --- |
| *<!--mybatis集成依赖-->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.0</**version**> </**dependency**> *<!--<MySQL依赖-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> |

## 配置数据源与mybatis

|  |
| --- |
| *#配置数据源* **spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.url**=**jdbc:mysql://localhost:3306/springboot spring.datasource.username**=**root spring.datasource.password**=**root** *#表示打印出sql语句* **logging.level.com.shyroke.mapper**=**debug** *#mybatis配置-映射文件扫描* **mybatis.mapper-locations**=**classpath:com/person/mapper/\*Mapper.xml** *#mybatis配置-别名包扫描* **mybatis.type-aliases-package**=**com.person.pojo** |

## 编写测试

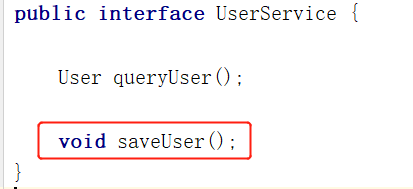
### 创建表：

CREATE TABLE `tb\_user` (  
  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  
  `name` varchar(12) DEFAULT NULL,  
  `age` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf32

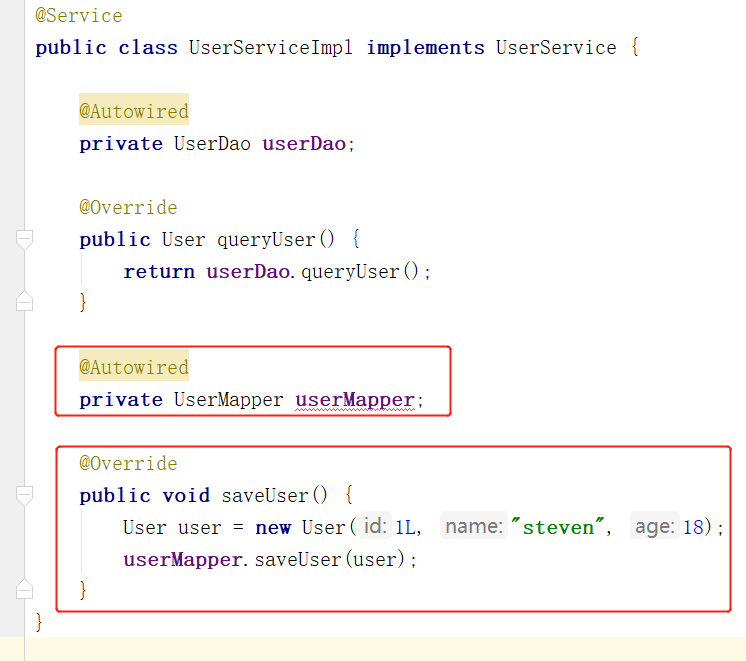
### UserController加入测试方法：

@RequestMapping(**"saveUser"**)  
**public** String saveUser(){  
 **userService**.saveUser();  
 **return "save user is ok"**;  
}

### UserService与impl



Impl



### UserMapper.java

|  |
| --- |
| @Mapper **public interface** UserMapper {  **void** saveUser(User user); } |

UserMapper.xml

*<?***xml version="1.0" encoding="UTF-8"***?>***<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>***<**mapper namespace="com.person.mapper.UserMapper"**>  
 <**select id="saveUser" parameterType="user"**>  
 INSERT INTO `tb\_user`  
 (`id`,  
 `name`,  
 `age`)  
 VALUES (#{id},  
 #{name},  
 #{age});  
 </**select**>  
</**mapper**>

### 注意的点：由于在idea默认中不会加加载java源码目录的xml，所以别忘了配置一下pom的resources节点在build中

<**resources**>  
 *<!-- 使用Maven部署的时候，xml和properties配置文件也一起部署到Tomcat -->* <**resource**>  
 <**directory**>src/main/java</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.properties</**include**>  
 <**include**>\*\*/\*.xml</**include**>  
 </**includes**>  
 <**filtering**>false</**filtering**>  
 </**resource**>  
</**resources**>

## druid连接池的使用

引入依赖

|  |
| --- |
| *<!--导入druid数据源-->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**> </**dependency**> |

指定连接池

|  |
| --- |
| *#配置数据源 #使用DruidDataSource* **spring.datasource.type**=**com.alibaba.druid.pool.DruidDataSource spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.url**=**jdbc:mysql://localhost:3306/springboot spring.datasource.username**=**root spring.datasource.password**=**root** *# 下面为连接池的补充设置，应用到上面所有数据源中 # 初始化大小，最小，最大* **spring.datasource.initialSize**=**5 spring.datasource.minIdle**=**5 spring.datasource.maxActive**=**20** *# 配置获取连接等待超时的时间* **spring.datasource.maxWait**=**60000** *# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒* **spring.datasource.timeBetweenEvictionRunsMillis**=**60000** *# 配置一个连接在池中最小生存的时间，单位是毫秒* **spring.datasource.minEvictableIdleTimeMillis**=**300000** |

# 声明式事务配置

## 基于注解

只需要在参与事务控制的方法上加上@Transactional注解即可。

|  |
| --- |
| @Transactional @Override **public void** addUser(User user) {  **int** acount = **userMapper**.addUser(user); } |

## 基于配置

引入aop依赖

|  |
| --- |
| *<!--aop依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-aop</**artifactId**> </**dependency**> |

制造异常

|  |
| --- |
| @Override **public void** addUser(User user) {  **int** acount = **userMapper**.addUser(user);  System.***out***.println(**"=======================================acount:"**+acount);  **int** i =0;  **int** q =10/i; } |

Xml事务传播特性配置

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd"**>   *<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean  below) 事务传播特性配置 -->* <**tx:advice id="txAdvice" transaction-manager="txManager"**>  *<!-- the transactional semantics... -->* <**tx:attributes**>  <**tx:method name="add\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="save\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="insert\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  *<!-- 查询方法 -->* <**tx:method name="query\*" read-only="true"** />  <**tx:method name="select\*" read-only="true"** />  <**tx:method name="find\*" read-only="true"** />  </**tx:attributes**>  </**tx:advice**>  *<!-- 配置事务管理器 -->* <**bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 声明式事务AOP配置 -->* <**aop:config**>  <**aop:pointcut expression="execution(\* cn.person.service.impl.\*.\*(..))" id="tranpointcut"** />  <**aop:advisor advice-ref="txAdvice" pointcut-ref="tranpointcut"** />  </**aop:config**> </**beans**> |

配置导入

|  |
| --- |
| @ImportResource(value = **"classpath:spring.xml"**) @SpringBootApplication **public class** SpringBootApp {   **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootApp.**class**,args);  } } |

# 文件上传

配置文件上传属性

|  |
| --- |
| *#默认支持文件上传.* **spring.http.multipart.enabled**=**true** *#支持文件写入磁盘.* **spring.http.multipart.file-size-threshold**=**0** *# 最大支持文件大小* **spring.http.multipart.max-file-size**=**1Mb** *# 最大支持请求大小* **spring.http.multipart.max-request-size**=**10Mb** |

文件上传代码实现

|  |
| --- |
| @RestController @RequestMapping(value = **"/upload"**) **public class** UploadController {   @PostMapping(value = **"/image"**)  **public** String image(@RequestParam(value = **"userimage"**)MultipartFile file, HttpSession session) **throws** Exception{  *//获取upload目录，不存在就创建* String realPath = session.getServletContext().getRealPath(**"upload"**);  File path = **new** File(realPath);  **if**(!path.exists()){  path.mkdirs();  }  *//上传* file.transferTo(**new** File(realPath+**"/"**+file.getOriginalFilename()));  **return "http://localhost:8080/upload/"**+file.getOriginalFilename();  } } |

# 注册拦截器

## 定义拦截器

|  |
| --- |
| **public class** UserInterceptor **implements** HandlerInterceptor {  @Override  **public boolean** preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o) **throws** Exception {  System.***out***.println(**"preHandle-------------------------------"**);  **return true**;  }   @Override  **public void** postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) **throws** Exception {  System.***out***.println(**"postHandle-------------------------------"**);  }   @Override  **public void** afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) **throws** Exception {  System.***out***.println(**"afterCompletion-------------------------------"**);  } } |

## 注册拦截器

|  |
| --- |
| @Configuration **public class** WebRegister **extends** WebMvcConfigurerAdapter {  @Override  **public void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(**new** UserInterceptor()).addPathPatterns(**"/add"**);  **super**.addInterceptors(registry);  } } |

# SpringBoot异常处理

@ControllerAdvice

该注解是spring2.3以后新增的一个注解，主要是用来Controller的一些公共的需求的低侵入性增强提供辅助，作用于@RequestMapping标注的方法上。

@ExceptionHandler

该注解是配合@ExceptionHandler一起使用的注解，自定义错误处理器，可自己组装json字符串，并返回到页面。

|  |
| --- |
| @ControllerAdvice **public class** ErrorHandlerController {   @ResponseBody  @ExceptionHandler(value = Exception.**class**)  **public** String handleGlobalException(Exception e){  System.***out***.println(**"0000000000000000000000------------------00000000000000000000000000"**+e.getMessage());  **return "error"**;  } } |

# 集成JPA实现增删改查

## 加入依赖

|  |
| --- |
| <**dependencies**>  *<!--添加一个特殊的web应用依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>   *<!-- 数据库 -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  </**dependency**>   *<!--MySQL-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**>  </**dependency**>   *<!--druid连接池-->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.20</**version**>  </**dependency**>   *<!--aop依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-aop</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**>  </**dependency**>   *<!-- jsp -->* <**dependency**>  <**groupId**>org.apache.tomcat.embed</**groupId**>  <**artifactId**>tomcat-embed-jasper</**artifactId**>  *<!-- <scope>provided</scope> -->* </**dependency**>   *<!-- servlet 依赖 -->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**>  *<!-- <scope>provided</scope> -->* </**dependency**>   *<!-- 添加 JSTL 支持 -->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>jstl</**artifactId**>  </**dependency**> </**dependencies**> |

## 编写测试

Controller代码

|  |
| --- |
| */\*\*  \* 输出JSON，get请求  \** ***@return*** *\*/* @GetMapping(value = **"/add"**) **public** String user(){  User user = **new** User();  user.setUsername(**"小红"**);  user.setUserage(21);  **userService**.addUser(user);  **return "list"**; } |

UserServiceImpl

|  |
| --- |
| @Autowired **private** UserMapper **userMapper**;  @Override **public void** addUser(User user) {  **userMapper**.save(user); } |

UserMapper

|  |
| --- |
| **public interface** UserMapper **extends** JpaRepository<User,Integer>{ } |