# 尚马教育 JAVA 高级课程

SpringBoot

文档编号：C16

创建日期： 2018-10-18

最后修改日期：2021-02-26

版 本 号：V1.0

电子版文件名：尚马教育-第三阶段-15.activiti工作流引擎.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2018-10-18 | 王绍成 | 初始版本 | V1.0 |
| 2019-09-12 | 王绍成 | 添加跨域配置 | V2.0 |
| 2021-02-26 | 冯勇涛 | 添加fastdfs-client | V3.0 |

目录

[SpringBoot 1](#_Toc22534)

[1.1 SpringBoot简介 2](#_Toc8857)

[1.1.1 什么是SpringBoot 2](#_Toc28961)

[1.1.2 SpringBoot开发者简介 2](#_Toc32117)

[1.1.3 SpringBoot 2.4环境要求 3](#_Toc10400)

[1.1.4 SpringBoot 2.4特性 3](#_Toc20183)

[1.2 第一个SpringBoot项目 3](#_Toc23678)

[1.2.1 使用IDEA Maven创建 3](#_Toc3159)

[1.2.2 使用IDEA中的Spring创建向导来创建 7](#_Toc9548)

[1.3 配置文件详解 10](#_Toc2078)

[1.3.1 第一个SpringBoot程序解析 10](#_Toc8456)

[1.4 SpringBoot开发Web 16](#_Toc29401)

[1.4.1 SpringBoot静态资源 16](#_Toc17834)

[1.4.1.1 增加当前项目的欢迎页 16](#_Toc23194)

[1.4.1.2 修改当前项目的浏览器图标 17](#_Toc268)

[1.4.2 Thymeleaf模板引擎 17](#_Toc30113)

[1.4.3 SpingBoot扩展 23](#_Toc22806)

[1.4.3.1 控制器映射 23](#_Toc16460)

[1.4.3.2 拦截器 23](#_Toc5723)

[1.4.3.3 格式转换器 24](#_Toc28621)

[1.4.4 自定义错误 25](#_Toc5255)

[1.5 日志 29](#_Toc1881)

[1.5.1.1 使用默认日志 29](#_Toc28493)

[1.5.1.2 Log4j2替换默认日志 30](#_Toc3044)

[1.6 Mybatis 31](#_Toc14263)

[1.6.1.1 集成环境配置 31](#_Toc29352)

[1.6.1.2 配置Druid连接池 32](#_Toc32006)

[1.6.1.2.1 配置简介 32](#_Toc2316)

[1.6.1.2.2 配置解析 33](#_Toc3461)

[1.6.1.3 测试Druid连接池对象是否被成功创建 34](#_Toc18603)

[1.6.1.4 Mybatis集成 34](#_Toc14072)

[1.6.1.4.1 目录结构文件 34](#_Toc24702)

[1.6.1.4.2 Mybatis核心文件配置及Mapper映射 35](#_Toc19351)

[1.6.1.4.3 将Dao层托管给Spring 36](#_Toc17533)

[1.6.1.4.4 解决java路径下xml不编译的问题 36](#_Toc6057)

[1.6.1.5 分页插件整合 37](#_Toc24376)

[1.6.1.5.1 加入插件依赖 37](#_Toc5443)

[1.6.1.5.2 使用PageHelper插件 37](#_Toc7470)

[1.6.1.6 事务整合 37](#_Toc22199)

[1.7 Redis整合 38](#_Toc2033)

[1.7.1 环境准备 38](#_Toc9648)

[1.7.1.1 引入依赖 38](#_Toc5979)

[1.7.1.2 修改application配置文件 39](#_Toc1049)

[1.7.1.3 基本使用示例 39](#_Toc20818)

[1.7.1.4 使对象自动序列化成JSON 39](#_Toc25912)

[1.7.2 注解方式使用 40](#_Toc21761)

[1.7.2.1 修改启动类 40](#_Toc26791)

[1.7.2.2 在需要使用缓存的位置使用注解 40](#_Toc5219)

[1.8 定时任务 41](#_Toc28177)

[1.8.1 环境准备 42](#_Toc22418)

[1.9 Solr整合 43](#_Toc20445)

[1.10 Idea开发时的小技巧 43](#_Toc16196)

[1.10.1 自动编译 43](#_Toc17453)

[1.11 兼容jsp 44](#_Toc26697)

[1.11.1 视图解析 44](#_Toc7900)

[1.11.2 引入依赖 44](#_Toc8221)

[1.11.3 修改启动类 45](#_Toc17502)

[1.11.4 发布到tomcat 45](#_Toc23227)

[1.11.5 一些错误 46](#_Toc23729)

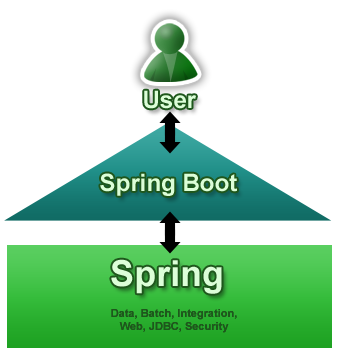
## SpringBoot简介

### 什么是SpringBoot

* JavaEE的一站式解决方案，一套真正的Spring全家桶应用
* Spring Boot是为了简化Spring应用的创建、运行、调试、部署等而出现的。
* 使用它可以做到专注于Spring应用的开发，无需过多关注XML的配置。
* 它提供了一堆依赖包，并已经按照使用习惯解决了依赖问题。使用默认方式实现快速开发。
* 提供大多数项目所需的非功能特性，诸如：嵌入式服务器(Jetty)、安全、心跳检查、外部配置等。
* Spring Boot 不生成代码，完全无需 XML配置，创建即用。

### SpringBoot开发者简介

SpringBoot默认集成Spring生态中的所有框架，只需要通过一些简短的配置即可使用。开发者只需要使用SpringBoot开发而不再需要关注Spring生态中的各种框架配置。



### SpringBoot 2.4环境要求

* SpringBoot2.4基于JDK8开发，JDK1.7无法运行
* Maven3以上
* 使用IntelliJ IDEA 或Spring官方的STS开发工具

### SpringBoot 2.4特性

* 创建独立的Spring应用程序
* 大大简化了安全自动配置
* 入门的“入门”依赖项，以简化构建配置
* 支持嵌入式 Netty
* Tomcat, Undertow 和 Jetty 均已支持 HTTP/2

## 第一个SpringBoot项目

### 使用IDEA Maven创建

1. 在pom中加入SpringBoot依赖

|  |
| --- |
| *<!-- Inherit defaults from Spring Boot -->* <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.4.0</version>  <relativePath/>  </parent>  *<!-- Add typical dependencies for a web application -->* <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**> </**dependencies**>  *<!-- Package as an executable jar -->* <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**> </**build**> |

1. 创建SpringBoot启动类（必须在所有类的最上层包中）

|  |
| --- |
| @SpringBootApplication **public class** SpringBootTestApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootTestApplication.**class**,args);  } } |

1. 创建一个SpringMVC的控制类

|  |
| --- |
| @Controller @RequestMapping(**"/hello"**) **public class** HelloController {    @RequestMapping(**"/test1"**)  @ResponseBody  **public** String test1() {  **return "index"**;  } } |

#### 在IDEA中直接执行

1. 运行SpringBoot启动类（直接右键Run即可）
2. 访问localhost:8080/hello/test1

#### 使用打包成Jar的方式执行

1. 检查pom文件中是否有当前maven打包插件支持

|  |
| --- |
|  |

1. 使用maven工具将当前项目打包

|  |
| --- |
|  |

1. 在当前项目中的target目录中找到生成好的Jar包

|  |
| --- |
|  |

1. 可以将Jar拷贝至其他地方使用cmd执行启动命令

|  |
| --- |
|  |

### 使用IDEA中的Spring创建向导来创建

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

* SpringBoot快速创建向导中创建的项目，默认会将SpringBoot项目的启动类、配置信息、测试用例等一并创建。
* SpringBoot默认不支持JSP，官方推荐使用thymeleaf页面模板引擎，这里是templates就是用来存放页面模板的。
* application.properties是SpringBoot的配置文件。关于SpringBoot内嵌tomcat端口号以及其他信息都是在这儿进行配置。同时SpringBoot也可以使用yml文件来描述配置。

## 配置文件详解

SpringBoot使用application.properties来声明当前框架的配置信息。同时在SpringBoot中也可以使用aplication.yml类描述配置。

* properties修改tomcat端口号实例

|  |
| --- |
|  |

* 配置文件的优先级（优先级由高到低，优先使用高级别的配置信息）
  + 项目路径下的config文件夹
  + 项目根目录下
  + resources目录下的config文件夹
  + resouces目录下

### 第一个SpringBoot程序解析

#### POM文件

|  |
| --- |
|  |

1. 父项目的依赖中包含一个新的父依赖spring-boot-dependencies,在这个父依赖中包含了当前版本的SpringBoot所有依赖Jar包的版本配置信息
2. SpringBoot将所有的功能都抽取出来，做成一个个starters，当我们需要使用对应的功能时，导入对应的starters就可以使用[【完整starters】](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "using-boot-starter)

|  |
| --- |
|  |

#### Application类解析

|  |
| --- |
| @SpringBootApplication **public class** SpringBootTestApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootTestApplication.**class**,args);  } } |

* 在SpringBoot主程序中只有简短的几行代码即可完成整个应用的启动，完成这些的关键点在于**@SpringBootApplication注解**

##### SpringBootApplication注解解析

* 被@SpringBootApplication标注的类就是当前SpringBoot应用的入口类。SpringBoot会在当前类中查找main方法并执行

|  |
| --- |
|  |

* 在SpringBootApplication类中，使用了两个注解分别标注当前类是配置类且开启自动配置

|  |
| --- |
|  |

* 在EnableAutoConfiguration中配置了自动配置包和需要导入的组件。
  + AutoConfigurationPackage实现了所有的包注册以及包扫描

|  |
| --- |
|  |

* + - 在Registrar静态内部类中，SpringBoot会查找到当前启动类下所在包下的所有子类，并将他们注册到Spring容器中
  + AutoConfigurationImportSelector 实现了给当前容器自动添加框架配置

|  |
| --- |
|  |

* + - 在selectImports方法中，SpringBoot已经给当前SpringBoot项目的Spring Framework的配置了默认配置。

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

* + - 且在selectImport中的loadMetadata方法中引入了一个Path常量。而该Path常量所指向的文件就是所有自动配置的属性配置文件

#### 什么是YML

YAML（YAML Ain’t Markup Language）是**“**YAML不是一种标记语言**”**的外语缩写；但为了强调这种语言以数据做为中心，而不是以置标语言为重点，。它是一种直观的能够被电脑识别的数据序列化格式，是一个可读性高并且容易被人类阅读，容易和脚本语言交互，用来表达资料序列的编程语言。

* 语法规则  
  1、大小写敏感   
  2、使用缩进表示层级关系   
  3、禁止使用tab缩进，只能使用空格键   
  4、缩进长度没有限制，只要元素对齐就表示这些元素属于一个层级。   
  5、使用#表示注释   
  6、字符串可以不用引号标注

#### 语法实例

YML可以方便直观的描述字符串、数字、时间、List和Map类型的数据。

* 案例

|  |
| --- |
| 配置application.yml |
| 创建实体类来测试yml数据描述 |
| 当创建完对应的实体类之后，IDEA会给出红色警告 |
| 按照官网提示，在pom文件中加入对应的可选依赖 |
| 在当前项目的测试用例中进行测试 |

#### SpringBoot配置属性一览

官方文档中给了properties所有可配置的选项。[【参考文档】](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "common-application-properties)

常见配置如下：

|  |
| --- |
| #内嵌服务器端口号  server.port=8081  #当前项目访问路径  server.servlet.context-path=/javasm |

## SpringBoot开发Web

### SpringBoot静态资源

SpringBoot中的静态资源存放在resources文件夹中，SpringBoot默认识别该文件夹下的public、static、resources子文件夹。当出现同名文件时优先级由低到高。

当需要自定义静态文件目录时,可以在属性配置文件中加入

|  |
| --- |
| spring.resources.static-locations=hello,res |

#### 增加当前项目的欢迎页

在SpringBoot中，只要在上述三个静态资源文件夹下创建一个index.html即可作为欢迎页来使用。

#### 修改当前项目的浏览器图标

在SpringBoot中，开发者可以在静态资源文件夹中添加favicon.ico来更改浏览器中显示的图标

### Thymeleaf模板引擎

在SpringBoot官方推荐的是使用Jar包启动，让每一个项目更加轻量，更加方便部署运行。由于使用了内嵌的轻量tomcat，所以SpringBoot默认不支持JSP页面引擎。

在SpringBoot中，官方更加推荐开发者使用Thymeleaf模板引擎来进行页面开发。Thymeleaf提供了更简单易读的标签，来替代我们在传统JSP中所使用的EL/JSTL等标签技术。

SpringBoot在WebMVC的视图解析中，默认查找的文件夹是在templates文件夹下查找。

#### 什么是Thymeleaf

Thymeleaf是一个Java服务器端的模板引擎，Thymeleaf的主要目标让开发者已更优雅的方式使用模板引擎，Thymeleaf已html的形式存在，它可以在服务器中运行也可以作为独立的静态页面运行，这样让前端开发和后端开发能够更加快速的协作。同时它可以与Spring框架进行集成，同时它还也完美兼容目前的HTML5特性。当然对于thymeleaf来说，它可以做的事情不仅仅如此。[【参考网站】](https://www.thymeleaf.org/index.html)

#### Thymeleaf环境准备

在前面我们做pom.xml解析时，我们提到了SpringBoot将所有的功能抽取出来做成一个个的starter启动器。而我们在SpringBoot中需要使用Thymeleaf也就只需要引入对于的starter就可以使用了。

#### pom添加Thymeleaf依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> |

#### Thymeleaf的使用

##### 基本使用案例

###### 创建一个controller

创建一个controller并往响应页面中传递一个参数

|  |
| --- |
| @Controller **public class** TestController {  @RequestMapping(**"/test"**)  **public** String test(Model model){  model.addAttribute(**"flag"**, **true**);  model.addAttribute(**"msg"**,**"你好,Thymeleaf"**);  **return "test"**;  } } |

###### 在html中添加thymeleaf环境

在templates文件夹下创建一个html文件,并在html文件头增加以下标签引用

|  |
| --- |
| <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> |

###### 页面使用

在body中使用thymeleaf属性来显示后端传递的值

|  |
| --- |
|  |

###### 开发小技巧

在实际的开发过程中,每次修改模板都需要能够快速的看到结果,也就是实时编译.而thymeleaf在集成中默认是开启了模板缓存的.可以通过下面的步骤来提高开发效率.

1. 在配置文件中关闭thymeleaf缓存.

|  |
| --- |
| spring.thymeleaf.cache=false |

1. 每次修改完thymeleaf后使用ctrl+F9快捷键, 重新进行编译.

#### 语法规则

##### th属性概览

在Thymeleaf官方文档中第10章**Attribute Precedence**中对所有的th语法介绍.th可以操作html中所有的属性.具体demo在文档第4章如下图.

|  |
| --- |
|  |

* 第一个应用示例
  + 在html中显示值文本是我们最常用的场景之一.

|  |
| --- |
| **#HTML标签文本方式**  <!--在任意标签中插入文本(不解析特殊字符)-->  <h4 th:text="${msg}"> </h4>  <!--在任意标签中插入文本(解析特殊字符)-->  <h4 th:utext="${msg}"></h4>  #HTML行内文本方式  <!--在html中直接显示值(不解析特殊字符)-->  [[${msg}]]  <!--在html中直接显示值(解析特殊字符)-->  [(${msg})] |

* + 当应用在标签中时,想要拼接字符串则可以使用以下两种方式

|  |
| --- |
| <!--使用字符串+号连接的拼接方式-->  <h4 th:text="'这是th:text标签 : ' + ${msg}"></h4>  <!--使用||包含字符串和表达式进行拼接-->  <h4 th:text="|这是th:text标签 ${msg}|"></h4> |

##### thymeleaf表达式

在thymeleaf中也跟el一样,会有丰富的表达式支持.在thymeleaf官方文档的第四章**Standard Expression Syntax**对所有的表达式进行了一个总结.常用表达式如下:

|  |  |  |  |
| --- | --- | --- | --- |
| **语法名称** | **语法规则** | **语法示例** | **语法介绍** |
| 属性表达式  (更多用法参照4.2节用法) | ${...} | //获取对象属性  ${userInfo.name}  ${userInfo[‘name’]}  //获取数组下标为0的name  ${userInfo[0].name}  //获取session中的属性.作用域还有param和application  ${session.username}  //当作用域前面加了#则表示使用对象中的方法,其他作用域还有#request和#servletContext  ${#session.getAttribute('username')}  //将当前请求中的变量进行日期转换  ${#calendars.format(date,'yyyy-MM-dd HH:mm:ss')} | 属性表达式的实质是OGNL表达式,使用方法类似于el表达式.开发者可以很方便的取出当前容器作用域中的值 |
| 选择表达式  (更多用法参考第4.3节用法) | \*{…} | <div th:object="${userInfo.user}">  //在th:object声明下的子集可以直接使用属性选择器获取属性值  <p>Name: <span th:text="\*{firstName}">Sebastian</span>.</p>  <p>Surname: <span th:text="\*{lastName}">Pepper</span>.</p>  <p>Nationality: <span th:text="\*{nationality}">Saturn</span>.</p>  </div> | 选择表达式一般搭配object和属性表达式一起使用.当HTML中发父级元素中声明了object则在子标签中可以使用选择表达式来直接获取object中的属性 |
| URL表达式  (更多用法参考第4.4节用法) | @{…} | //使用(参数名=参数值,参数名=参数值)来拼接参数,会自动追加?号  <!-- 'http://localhost:8080/gtvg/order/details?orderId=3' -->  <a  th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>  <!-- '/gtvg/order/details?orderId=3'-->  <a th:href="@{/order/details(orderId=${o.id})}">view</a>  <!-- '/gtvg/order/3/details?orderId=3'-->  <a th:href="@{‘/order/’+${orderId}+’/details’(orderId=${o.id})}">view</a> | URL表达式可以更加方便的声明超连接,可以在超链接中加载变量及拼接,并且会自动追加当前项目的访问路径 |
| 文档表达式  (更多用法参考第8节用法) | ~{…} | //使用~{文档文件名(可不写) :: 文档名};当文档名不写则表示引用当前html中的文档  //创建thymeleaf公共模板   |  | | --- | | **#tmp.html**  <div **th:fragment**="publicTemplate">  hello ,公共文档  </div> |   //使用文档表达式进行文档引用.   |  | | --- | | **#test.html**  <!—将文档中的代码插入到当前div中-->  <div **th:insert**="~{tmp :: publicTemplate }"></div>  <!—使用文档代码替换当前div-->  <div **th:replace**="~{tmp :: publicTemplate }"></div> |   //运行结果   |  | | --- | | <div><div> hello ,公共文档</div></div>  <div>hello ,公共文档</div> | | 也可以直接写html地址,下面路径会自动引用/templates/common/left.html  <div th:insert="common/left"></div>  <div th:replace="common/left"></div>  <div th:include="common/left"></div> | | 文档表达式相当于jsp中的include,可以任意引用其他公共的html代码,减少前端代码冗余 |

##### th常用标签介绍

th可以用来做数据显示、HTML属性操作、逻辑判断、集合迭代、文档引用等.可以理解为是JSP中的EL和JSTL的更强替代品.

Th:value th:text th:action

|  |  |  |  |
| --- | --- | --- | --- |
| **标签名称** | **标签规则** | **标签示例** | **标签介绍** |
| 条件判断  (更多用法参考第7节) | th:if | //判定值,满足条件则显示当前div  <div th:if="${counts>= 100}">  大于等于100  </div> | If标签的使用方法与c:if基本一致 |
| th:switch | //使用switch标签与属性表达式来判定  <div th:switch="${user.role}">  //case中可以直接进行值或表达式值判定  <p th:case="'admin'">系统管理员</p>  <p th:case="${roles.manager}">主管</p>  //case中的值为\*号时表示默认  <p th:case="\*">其他</p>  </div> | Switch-case标签组合的基本使用与Java一致.能够更好的判断不同的值给出不同的显示情况.也同样包含默认值. |
| 集合迭代  (更多用法参考第6节) | th:each | //th:each=”迭代元素名 : 迭代的数组”   |  | | --- | | **#基本迭代示例**  <table>  <tr>  <td>名称</td>  <td>性别</td>  </tr>  <tr th:each="user : ${users}">  <td th:text="${user.username}"></td>  <td th:text="${user.sex}"></td>  </tr>  </table> |   //th:each=”迭代元素名,迭代状态名 : 迭代的数组”   |  | | --- | | **#迭代状态示例**  <table>  <tr>  <td>名称</td>  <td>性别</td>  </tr>  <tr th:each="user,userstat : ${users}">  <td th:text="${user.username}"></td>  <td th:text="${user.sex}"></td>  <td th:text="${userStat.index}">状态变量:索引(0开始)</td>  <td th:text="${userStat.count}">状态变量：当前行</td>  <td th:text="${userStat.size}">状态变量：总长度</td>  <td th:text="${userStat.even}">状态变量：是偶数?</td>  <td th:text="${userStat.odd}">状态变量：是奇数?</td>  <td th:text="${userStat.first}">状态变量：是第一位?</td>  <td th:text="${userStat.last}">状态变量：是最后一位?</td>  </tr>  </table> | | th:each与c:forEach标签用法也差不多,在each中,想要获得当前数组的遍历标记,则需要在遍历对象名后增加一个新的参数名即可使用. |

### SpingBoot扩展

在SpringBoot2.0版本中,使用了一个核心接口WebMvcConfigurer 来定义用户自定义的扩展. 比如表单提交中的时间转换,默认访问路径映射,拦截器等.

#### 控制器映射

在SpringBoot的开发中,静态资源中的文件都是可以直接访问的,而我们的页面在某些情况下是不希望用户能直接通过在浏览器中填写URL直接访问,那么我们就会把页面放在用户不可直接访问的目录,由所有的controller来做一一映射跳转.

那么在这种情况下,我们可能就会需要定义多个跳转页面的Controller methed来做页面映射.而现在在SpringBoot中的WebMvcConfigurer中直接提供了添加多个控制器的映射的方法.

##### addViewControllers的使用

|  |
| --- |
|  |

#### 拦截器

SpringBoot中声明拦截器与SpringMVC没什么区别，拦截器本身使用的仍然还是SpringMVC那一套。在SpringBoot中只需要通过WebMvcConfigurer的addInterceptors注册一下拦截器即可。

##### 自定义登录拦截器

|  |
| --- |
| **public class** SessionIntercept **implements** HandlerInterceptor {   @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  Object userInfo = request.getSession().getAttribute(**"userInfo"**);  **if**(userInfo != **null**){  **return true**;  }**else**{  request.getRequestDispatcher(**"/login"**).forward(request,response );  **return false**;  }  } } |

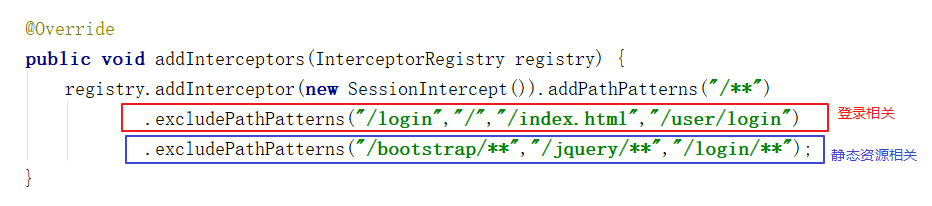
##### 注册拦截器

|  |
| --- |
|  |

在SpringBoot中拦截器的添加与视图控制映射的添加差不多。通过addPathPatterns可以添加当前拦截器需要拦截的请求路径。

##### 排除路径

在实际的开发中，拦截路径是需要排除登录页面以及登录请求等入口页面及请求的。在SpringBoot中也可以直接使用excludePathPatterns方法来排除请求路径。



#### 格式转换器

|  |
| --- |
|  |

当从前端表单中传递的参数中携带有Date字段,而我们又希望能够使用SpringMVC提供的自动接收参数时,我们就需要使用自定义一个参数格式转换器.

##### 自定义时间转换器

开发者可以自己定义一个实现org.springframework.core.convert.converter.Converter的子类.在实现时定义参数类型以及转换类型.在Convert中对需要转换的类型进行转换即可.

|  |
| --- |
| **public class** MyDateFormat **implements** Converter<String, Date> {  **private static final** String ***dateFormat*** = **"yyyy/MM/dd"**;  @Override  **public** Date convert(String source) {  **if** (StringUtils.*isEmpty*(source)) {  **return null**;  }  **try** {  SimpleDateFormat formatter = **new** SimpleDateFormat(***dateFormat***);  Date date = formatter.parse(source);  **return** date;  } **catch** (Exception e) {  **throw new** RuntimeException(String.*format*(**"parser %s to Date fail"**, source));  }  } } |

##### 配置时间转换器

自定义时间格式转换器需要添加到SpringMVC容器中才能生效。在SpringBoot在WebMvcConfigurer中提供了addFormatters来供用户提供添加自定义格式化转换器的方法。

|  |
| --- |
| @Override **public void** addFormatters(FormatterRegistry registry) {  registry.addConverter(**new** MyDateFormat()); } |

### 自定义错误

在实际的项目开发中我们遇到的最多的错误就是400错误和500错误。

在SpringBoot中,MVC错误由ErrorMvcAutoConfiguration来负责自动配置错误页面及错误信息返回，也就是我们看到的4XX、5XX这些错误页面信息的网页,且可以将默认的错误信息显示在自定义错误页面中。当我们通过其他接口访问测试工具，如postman这些访问时，所有的报错页面会变成错误json。

|  |
| --- |
|  |
|  |

#### 自定义错误页面

##### 创建错误页面

当我们需要自定义404错误时，我们只需要在templates文件夹下创建error文件夹，并在error中创建一个404.html即可。同时我们也可以创建一个4xx.html，当服务器返回的只要是400类型的错误就会自动跳转到4xx.html中。

###### 原理解析

在SpringBoot中，自动配置机制会加载ErrorMvcAutoConfiguration类，在该类中由basicErrorController主要负责发生错误后的页面跳转，而basicErrorController中的errorHtml()和error()方法就分别是返回html和json的两个控制器。

|  |
| --- |
| **#在ErrorMvcAutoConfiguration中的basicErrorController()方法会返回一个BasicErrorController** |
| **#在basicErrorController上我们可以看到这是一个控制器,且映射路径是可以手工配置的** |
| **#基本错误控制器中的两个方法的调用场景** |
| **#在错误网页方法中,依靠一个错误页面解析器方法来解析并返回视图模型** |
| **#模型视图名称组装规则** |

##### 自定义异常处理器

在自定义页面中,我们可以使用SpringBoot的自动错误处理机制去方便的写自己的个性化错误页面,但是字段我们却不能个性化添加。如果我们需要个性化添加错误字段以及提示文本，则我们需要自定义一个异常处理器来解决。根据错误页面自动配置相同的原理，错误字段属性也在ErrorMvcAutoConfiguration中的errorAttributes方法中自动注册。而在该方法中定义了一个@ConditionalOnMissingBean的注解，这个注解的意思是，当容器中没有找到自定义的类时，就使用默认的实现（DefaultErrorAttributes）来处理。

而在basicErrorController中对于页面错误处理都有相同的获取错误属性操作，且根据该方法的调用连最终就能找到DefaultErrorAttributes类的默认实现。

|  |
| --- |
|  |

基于以上原理，我们只需要给DefaultErrorAttributes创建一个子类即可实现自定义属性的功能。

###### 自定义异常属性方法

|  |
| --- |
|  |

### 跨域配置

|  |
| --- |
| @Configuration  public class MyConfiguration {  //springmvc中 没有FilterRegistrationBean这个类  @Bean  public FilterRegistrationBean这个类 corsFilter() {  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  CorsConfiguration config = new CorsConfiguration();  config.setAllowCredentials(true);  config.addAllowedOrigin("http://domain1.com");  config.addAllowedHeader("\*");  config.addAllowedMethod("\*");  source.registerCorsConfiguration("/\*\*", config);  FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));  bean.setOrder(0);  return bean;  }  } |

## 日志

SpringBoot默认集成有日志功能，使用的是Apache的commons-logging做日志的输出功能，且使用的logback的日志机制。除了默认的以外还提供一个Log4J2的starter供开发者选择，当我们引入相应的starter则默认日志应用就能被切换。[【官方文档】](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "boot-features-logging)

#### 使用默认日志

SpringBoot默认不配置的情况下可以直接使用commons-logging来进行日志打印,可以直接调用相应的方法进行日志打印功能。

|  |
| --- |
|  |

同时可以在application中对日志规则进行设定。

|  |
| --- |
| *#log配置信息* **logging:** *#配置日志目录以及级别,root表示根目录* **level:** {**root:** debug}  *#配置日志输出目录的盘符路径* **path:** D:/logs/  *#配置日志的名,相对路径(与path只能选其一)* **file:** logs/test.log |

使用默认的日志功能，只需要上述简单的配置即可完成对于的设置功能。

#### Log4j2替换默认日志

在SpringBoot中，官方默认为我们预先实现了log4j和logback，我们只需要引入相关的starter即可使用。[【官方文档】](https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/" \l "howto-configure-log4j-for-logging)

|  |
| --- |
|  |

因为默认使用了logging，则我们在使用时，需要排除掉logging的依赖，再加入log4j的starter

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  <**exclusions**>  <**exclusion**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  <**groupId**>org.springframework.boot</**groupId**>  </**exclusion**>  </**exclusions**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j2</**artifactId**>  </**dependency**> |

最后在项目的resources文件夹下加入log4j2.xml即可使用.

理论上来说，加入exclusion就已经排除了logging的jar包，但是实际使用中，logging偶尔会依然存在，这样会导致jar包冲突，配置文件使用失败。需要进入maven的视图，找到对应的jar包手动移除。

## Mybatis

#### 集成环境配置

在当前项目中加入Mybatis集成需要依赖的Jar。

|  |
| --- |
| *<!--jdbc依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> *<!-- mybatis依赖-->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.2</**version**> </**dependency**> *<!-- mysql驱动依赖-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> *<!--durid依赖-->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid-spring-boot-starter</**artifactId**>  <**version**>1.1.10</**version**> </**dependency**> |
| <!—build标签下加入下面代码 解决idea下不自动编译xml文件的问题 -->  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.xml</include>  </includes>  </resource>  <resource>  <directory>src/main/resources</directory>  <filtering>true</filtering>  </resource>  </resources> |

#### 配置Druid连接池

Druid是阿里巴巴开源的一款连接池工具，是目前Java语言中最好的数据库连接池工具。代码发布在github上。

##### 配置简介

在application配置文件中加入Druid的配置信息。

|  |
| --- |
| **spring:**  **datasource:  name:** mysql\_test  **type:** com.alibaba.druid.pool.DruidDataSource  *#druid相关配置* **druid:** *#监控统计拦截的filters* **filters:** stat  **driver-class-name:** com.mysql.jdbc.Driver  *#基本属性* **url:** jdbc:mysql://localhost:3306/javasm?useUnicode=true&characterEncoding=UTF-8&allowMultiQueries=true&serverTimezone=GMT%2B8&useSSL=false  **username:** root  **password:** root  *#配置初始化大小/最小/最大* **initial-size:** 1  **min-idle:** 1  **max-active:** 20  *#获取连接等待超时时间* **max-wait:** 60000  *#间隔多久进行一次检测，检测需要关闭的空闲连接* **time-between-eviction-runs-millis:** 60000  *#一个连接在池中最小生存的时间* **min-evictable-idle-time-millis:** 300000  **validation-query:** SELECT 'x'  *#空闲连接是否被回收* **test-while-idle: true** *#申请连接时是否检测有效性* **test-on-borrow: false** *#归还连接时是否检测有效性* **test-on-return: false** |

##### 配置解析

在SpringBoot中默认采用DataSourceAutoConfiguration来配置数据源（DataSource）。官方默认只需要在application配置文件中使用简单的配置就可以连接数据库。

当加入了Druid-starter之后，它的DruidDataSourceAutoConfigure默认初始化了一个创建DataSource的方法。而这个DataSource默认就是读取当前application配置文件中的spring.datasource.druid的值，由此创建了基于Druid的数据源。

|  |
| --- |
| **#SpringBoot自带的DataSourceAutoConfiguration** |
|  |

#### 测试Druid连接池对象是否被成功创建

|  |
| --- |
|  |

#### Mybatis集成

在前面的集成环境配置中，我们已经加入了mybatis相关的starter，也就是说，mybatis已经集成好了。我们仅仅只需要一段简短的配置即可完成整合并开发。

##### 目录结构文件

文件创建规则与SSM框架整合中使用一致。

|  |
| --- |
|  |

##### Mybatis核心文件配置及Mapper映射

在之前我们整合SSM时，我们会将Mapper映射文件以及Mybatis核心配置文件交给Spring来管理并加载。在Mybatis的starter中，按照SpringBoot的规范，同样也由一个MybatisAutoConfiguration来负责自动加载配置文件，并注册配置类。

|  |
| --- |
|  |

##### 将Dao层托管给Spring

###### 使用Mapper注解

|  |
| --- |
|  |

在dao层，给所有的dao方法加上Mapper注解，将当前的类标注为Mybatis的mapper接口托管给Spring

###### 使用MapperScan注解

|  |
| --- |
|  |

使用MapperScan注解来扫描接口包路径

##### 解决java路径下xml不编译的问题

|  |
| --- |
| *<resource>*  *<directory>src/main/java</directory>*  *<includes>*  *<include>\*\*/\*.xml</include>*  *</includes>*  *</resource>* |

解决样式字体不生效问题

|  |
| --- |
| *<resource>*  *<directory>src/main/resources</directory>*  *<filtering>true</filtering>*  *<excludes>*  *<exclude>\*\*/\*.woff</exclude>*  *<exclude>\*\*/\*.woff2</exclude>*  *<exclude>\*\*/\*.ttf</exclude>*  *</excludes>*  *</resource>*  *<resource>*  *<directory>src/main/resources</directory>*  *<filtering>false</filtering>*  *<includes>*  *<include>\*\*/\*.woff</include>*  *<include>\*\*/\*.woff2</include>*  *<include>\*\*/\*.ttf</include>*  *</includes>*  *</resource>* |

#### 分页插件整合

##### 加入插件依赖

在POM文件中加入分页插件的依赖。PageHelper的作者也同样为SpringBoot创建了对应的starter。加入starter就可以使用Mybatis的分页功能了。如果需要定制其他功能则可以按照github上的文档进行配置。[【文档】](https://github.com/pagehelper/pagehelper-spring-boot)

|  |
| --- |
| *<!-- 分页插件 -->* <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  <**version**>1.2.5</**version**> </**dependency**> |

##### 使用PageHelper插件

PageHelper在项目的使用中基本上对已有的代码没有任何干扰。只需要在查询方法前调用一个静态方法即可完成分页查询。

|  |
| --- |
|  |

#### 事务整合

在SpringBoot中使用事务也很简单,开发者只需要在启动类上加上@EnableTransactionManagement注解开启事务管理,并在需要做增删改的操作上加上@Transactional注解即可完成事务的配置及使用。

## Redis整合

在实际的开发中,我们会将用户经常访问请求的数据放到redis服务中,用来提升服务器响应速度及性能。SpringBoot作为Spring全家桶的代表性框架，也同样为我们默认集成了Redis，按照之前我们提到的，Spring也同样为Redis做了一个Starter。而这个starter基于Spring-Data框架。

SpringData框架是Spring出的一款数据访问框架，它包括许多子项目，基本上涵盖了目前企业开发所需要使用的主流数据存储应用，包括搜索引擎。这个项目同时也包含了Spring想一统JavaEE企业开发的野心。但是由于它提供的ORM框架-JPA对于复杂SQL以及后期维护等，并不是很友好，所以目前现阶段来讲，无法去替代MyBatis的地位。

### 环境准备

与其他框架整合相同，需要继承redis也仅仅只需要引入redis的starter，然后在application配置文件中简单配置下即可使用

|  |
| --- |
|  |

#### 引入依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  <exclusions>  <exclusion>  <groupId>io.lettuce</groupId>  <artifactId>lettuce-core</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency> |
| <!-- Fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.7</version>  </dependency> |

#### 修改application配置文件

如果当前redis实例连接的是本地且是默认端口6379，则无需配置。具体参数可以参考RedisAutoConfiguration类中引入的RedisProperties类

|  |
| --- |
|  |

#### 基本使用示例

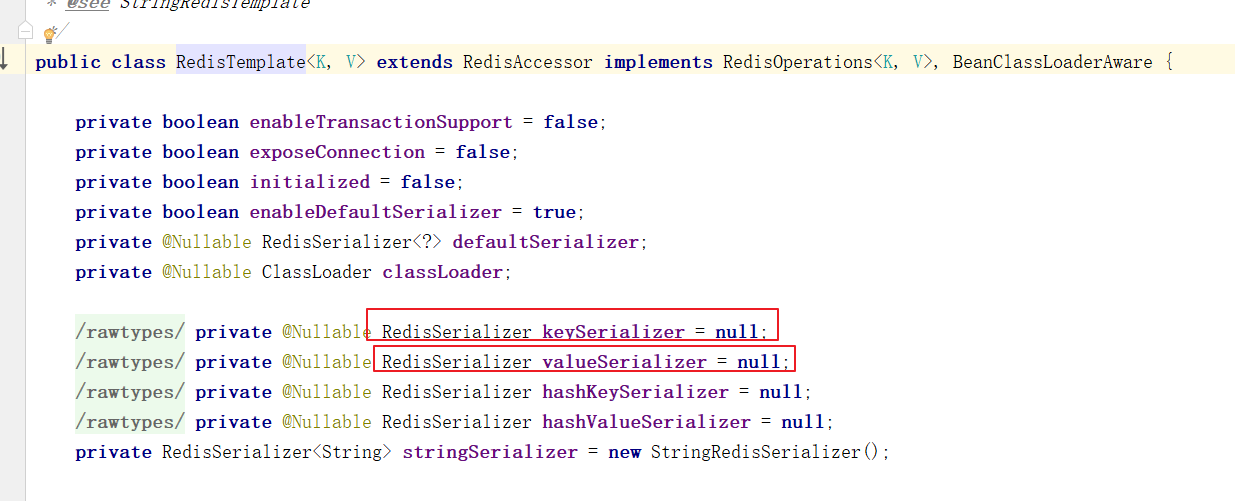
由于SpringBoot已自动帮我们创建好了Redis的连接以及操作模板，开发者只需要引入模板类即可对Redis进行操作。

|  |
| --- |
|  |

#### 使对象自动序列化成JSON

在实际的开发中，我们会经常讲用户信息或其他对象信息以json的方式保存至redis中。而每次我们都需要在代码中将对象给重新序列化成json，这样就会很麻烦。实际上在RedisAutoConfiguration中，我们可以看到以下两个方法，他们分别向容器中注册了redisTemplate和stringRedisTemplate两个方法，并向容器中注册了两个对象。而StringRedisTemplate就是我们上述使用的字符串操作对象。

RedisTemplate就是可以直接保存对象的模板。而在RedisTemplate类中我们可以看到如下信息：



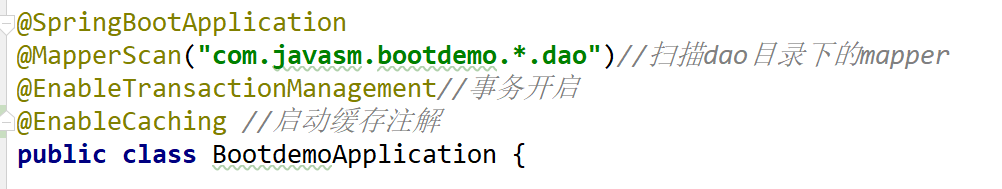
该类分别由两个成员对象，key序列化和值序列化对象。基于这样的情况，我们只需要在Spring初始化的时候，自己创建一个RedisTemplate对象并给它设置自定义的值序列化对象。

|  |
| --- |
|  |

当我们重新定义了RedisTemplate之后，我们就可以直接进行对象保存了。

### 注解方式使用

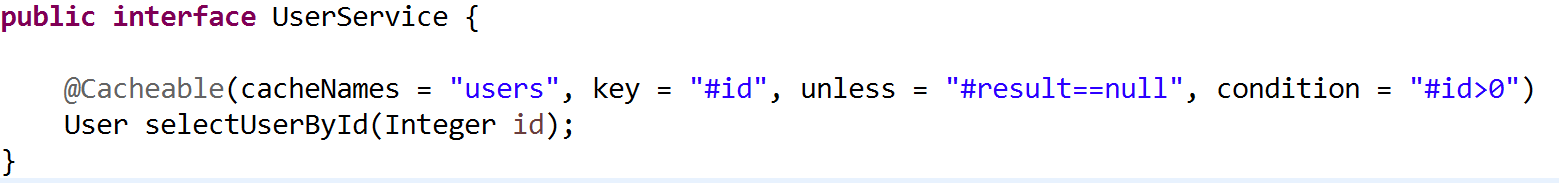
#### 修改启动类



加入@EnableCaching

#### 在需要使用缓存的位置使用注解

注解可以加在接口的方法上，也可以加在接口实现类的方法上，一个地方加就可以





|  |  |
| --- | --- |
| **注解** | **描述** |
| @Cacheable | 表明在Spring调用之前，首先应该在缓存中查找方法的返回值，如果这个值能够找到，就会返回缓存的值，否则这个方法会被调用，返回值会放到缓存中 |
| @CachePut | 表明Spring应该将该方法返回值放到缓存中，在方法调用前不会检查缓存，方法始终会被调用 |
| @CacheEvict | 表明Spring应该在缓存中清除一个或多个条目 |
| @Caching | 分组注解，能够同时应用多个其他的缓存注解 |
| @CacheConfig | 可以在类层级配置一些共有的缓存配置 |

## 定时任务

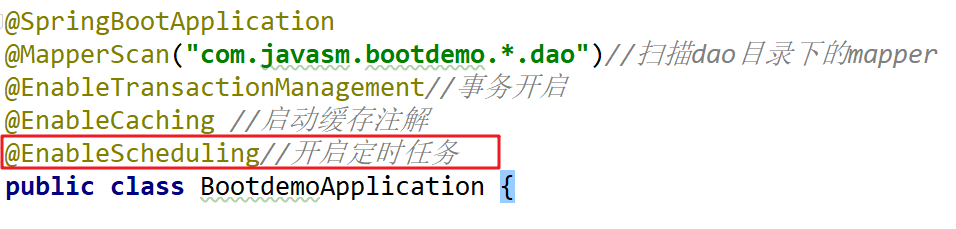
Spring Schedule是Spring自带的任务框架，是一个简化版本的Quartz

### 环境准备

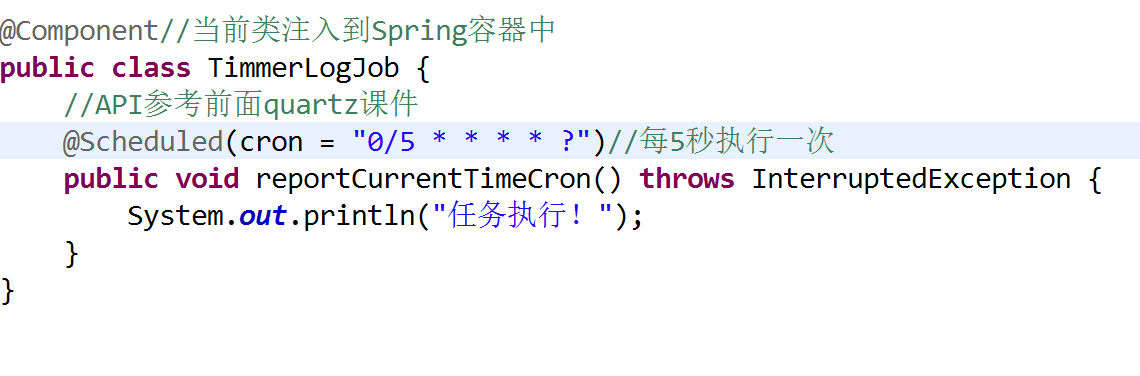
#### 引入依赖

|  |
| --- |
| <!-- Quartz定时任务调度 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-quartz</artifactId>  </dependency> |

#### 修改启动类



#### 创建任务类



## Solr整合

## Idea开发时的小技巧

### 自动编译

Idea默认是自动保存的，但是不会自动编译。在开发SpingBoot时，我们可以添加一个开发工具依赖简单配置一下即可完成修改文件实时生效。

1. 添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> </**dependency**> |

1. 配置idea

|  |
| --- |
|  |

1. 使用ctrl+alt+shitf+/ 换出idea的维护设置

|  |
| --- |
|  |
|  |

**注：改文件名，添加新的文件，仍然还是需要自己手动重启。**

**开发工具也有各种各样的bug存在,当出现各种奇幻的情况时,可以选择手动重启项目、开发工具、操作系统。**

## 兼容jsp

官方不推荐使用jsp作为前端页面，这里介绍一下如何兼容jsp

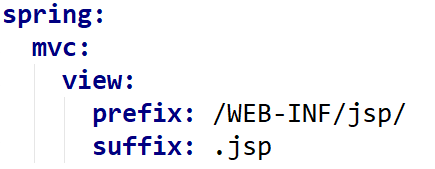
想要访问jsp需要外部tomcat的支持，tomcat版本必须是8以上

发布的jar包改成war

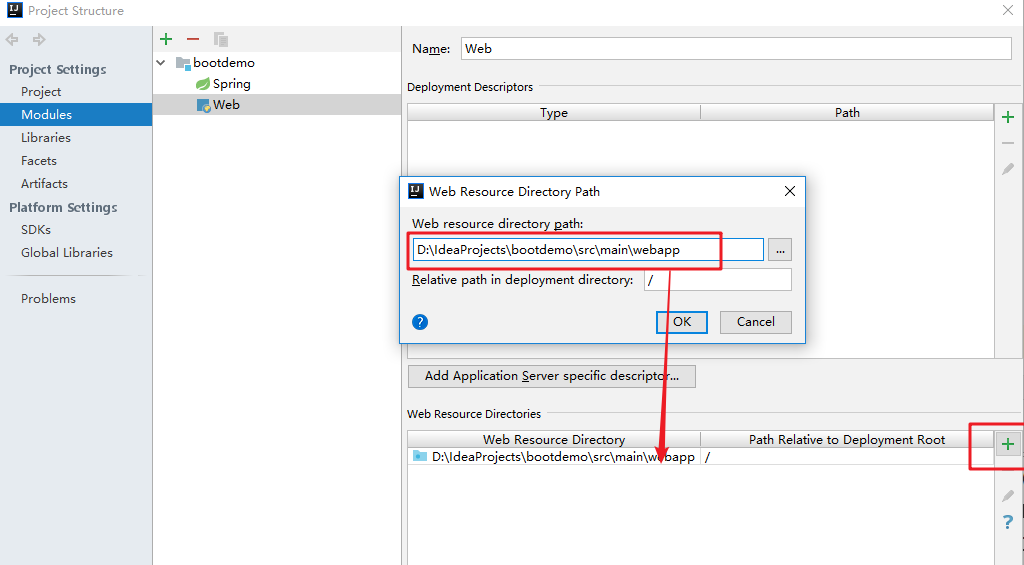
### 视图解析

不使用默认的视图解析路径，手动修改视图解析

修改yml配置文件，根据项目实际路径修改



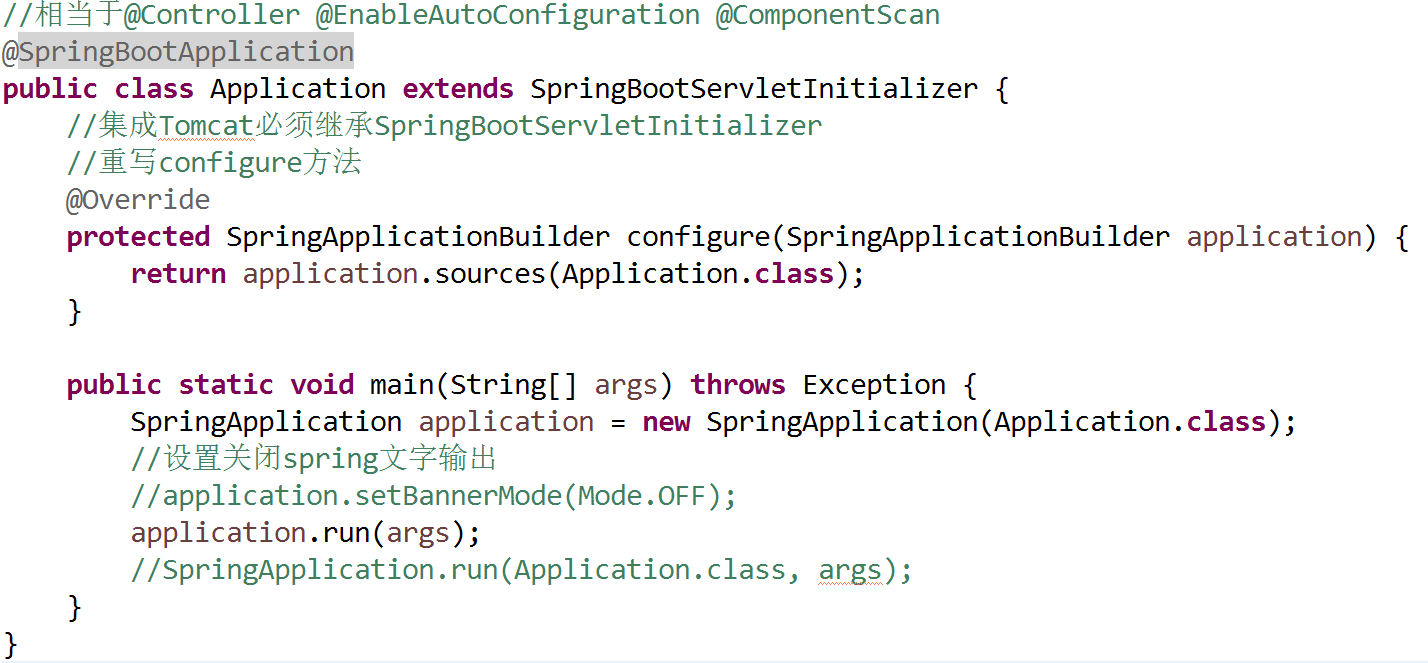
Idea创建的springboot项目默认不能创建jsp文件，项目设置中配置web’目录



### 引入依赖

|  |
| --- |
| <!--用于编译jsp-->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  <scope>provided</scope>  </dependency> |
| <!--jsp页面使用jstl标签 高版本Tomcat/Myeclipse 以下可以省略-->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  </dependency>  <!-- 避免外部Tomcat和内嵌Tomcat冲突-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  <scope>provided</scope>  </dependency> |
| <!--为了兼容jsp的使用 注释掉thymeleaf  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>--> |

### 修改启动类



### 发布到tomcat

启动tomcat

### 一些错误

