

一、SpringMVC简介

- 1、什么是MVC
- 2、什么是SpringMVC
- 3、SpringMVC的特点

二、HelloWorld

- 1、开发环境
- 2、创建maven工程
 - a>添加web模块
 - c>引入依赖
- 3、配置web.xml
 - a>默认配置方式
 - b>扩展配置方式 (大部分使用的是这个)
- 4、创建请求控制器
- 5、创建springMVC的配置文件
- 6、测试HelloWorld
 - a>实现对首页的访问
 - b>通过超链接跳转到指定页面
- 7、总结

三、@RequestMapping注解

- 1、@RequestMapping注解的功能
- 2、@RequestMapping注解的位置
- 3、@RequestMapping注解的value属性
- 4、@RequestMapping注解的method属性
- 5、@RequestMapping注解的params属性 (了解)
- 6、@RequestMapping注解的headers属性 (了解)
- 7、SpringMVC支持ant风格的路径
- 8、SpringMVC支持路径中的占位符 (重点)

四、SpringMVC获取请求参数

- 1、通过ServletAPI获取
- 2、通过控制器方法的形参获取请求参数
- 3、@RequestParam
- 4、@RequestHeader
- 5、@CookieValue
- 6、通过POJO获取请求参数
 - 集合参数
- 7、解决获取请求参数的乱码问题

五、域对象共享数据

- 1、使用ServletAPI向request域对象共享数据
- 2、使用 ModelAndView 向 request 域对象共享数据
- 3、使用 Model 向 request 域对象共享数据
- 4、使用 map 向 request 域对象共享数据
- 5、使用 ModelMap 向 request 域对象共享数据
- 6、Model、ModelMap、Map 的关系
- 7、向 session 域共享数据 (建议使用)
- 8、向 application 域共享数据

六、SpringMVC的视图

- 1、Thymeleaf视图 ThymeleafView
- 2、转发视图 InternalResourceView
- 3、重定向视图 RedirectView
- 4、视图控制器 view-controller

七、RESTful

- 1、RESTful简介
 - a>资源

- b>资源的表述（状态）
 - c>状态转移
 - 2、RESTful的实现
 - 3、HiddenHttpMethodFilter
- 八、RESTful案例**
- 简化开发型
 - 1、准备工作
 - 2、功能清单
 - 3、具体功能：访问首页
 - a>配置view-controller
 - b>创建页面
 - 4、具体功能：查询所有员工数据
 - a>控制器方法
 - b>创建employee_list.html
 - 5、具体功能：删除
 - a>创建处理delete请求方式的表单
 - b>删除超链接绑定点击事件
 - c>控制器方法
 - 6、具体功能：跳转到添加数据页面
 - a>配置view-controller
 - b>创建employee_add.html
 - 7、具体功能：执行保存
 - a>控制器方法
 - 8、具体功能：跳转到更新数据页面
 - a>修改超链接
 - b>控制器方法
 - c>创建employee_update.html
 - 9、具体功能：执行更新
 - a>控制器方法

八、HttpMessageConverter

- 1、@RequestBody
- 2、RequestEntity
- 3、@ResponseBody
- 4、SpringMVC处理json
- 5、SpringMVC处理ajax
- 6、@RestController注解
- 7、ResponseEntity

九、文件上传和下载

- 1、文件下载
- 2、文件上传

十、拦截器

- 1、拦截器的配置
- 2、拦截器的三个抽象方法
- 3、多个拦截器的执行顺序

十一、异常处理器

- 1、基于配置的异常处理
- 2、基于注解的异常处理

十二、注解配置SpringMVC

- 1、创建初始化类，代替web.xml
- 2、创建SpringConfig配置类，代替spring的配置文件
- 3、创建WebConfig配置类，代替SpringMVC的配置文件
- 4、测试功能

十三、SpringMVC执行流程

- 1、SpringMVC常用组件
- 2、DispatcherServlet初始化过程

a>初始化WebApplicationContext
b>创建WebApplicationContext
c>DispatcherServlet初始化策略
3、DispatcherServlet调用组件处理请求
a>processRequest()
b>doService()
c>doDispatch()
d>processDispatchResult()

4、SpringMVC的执行流程

注解实现springmvc

启动服务器初始化过程

单次请求过程

简化开发版本

接收请求中的json数据

日期类型参数传递

响应POJO对象

ResponseBody

SSM整合

拦截器

步骤

一、SpringMVC简介

1、什么是MVC

MVC是一种软件架构的思想，将软件按照模型、视图、控制器来划分

M: Model，模型层，指工程中的JavaBean，作用是处理数据

JavaBean分为两类：

- 一类称为实体类Bean：专门存储业务数据的，如 Student、User 等
- 一类称为业务处理 Bean：指 Service 或 Dao 对象，专门用于处理业务逻辑和数据访问。

V: View，视图层，指工程中的html或jsp等页面，作用是与用户进行交互，展示数据

C: Controller，控制层，指工程中的servlet，作用是接收 请求和响应浏览器

MVC的工作流程：

用户通过视图层发送请求到服务器，在服务器中请求被Controller接收，Controller调用相应的Model层处理请求，处理完毕将结果返回到Controller，Controller再根据请求处理的结果找到相应的View视图，渲染数据后最终响应给浏览器

2、什么是SpringMVC

SpringMVC是Spring的一个后续产品，是Spring的一个子项目

SpringMVC 是 Spring 为表述层开发提供的一整套完备的解决方案。在表述层框架历经 Strut、WebWork、Strut2 等诸多产品的历代更迭之后，目前业界普遍选择了 SpringMVC 作为 Java EE 项目表述层开发的首选方案。

注：三层架构分为表述层（或表示层）、业务逻辑层、数据访问层，表述层表示前台页面和后台servlet

3、SpringMVC的特点

- Spring 家族原生产品，与 IOC 容器等基础设施无缝对接
- 基于原生的Servlet，通过了功能强大的前端控制器DispatcherServlet，对请求和响应进行统一处理
- 表示层各细分领域需要解决的问题全方位覆盖，提供全面解决方案
- 代码清新简洁，大幅度提升开发效率
- 内部组件化程度高，可插拔式组件即插即用，想要什么功能配置相应组件即可
- 性能卓著，尤其适合现代大型、超大型互联网项目要求

二、HelloWorld

1、开发环境

IDE: idea 2019.2

构建工具: maven3.5.4

服务器: tomcat7

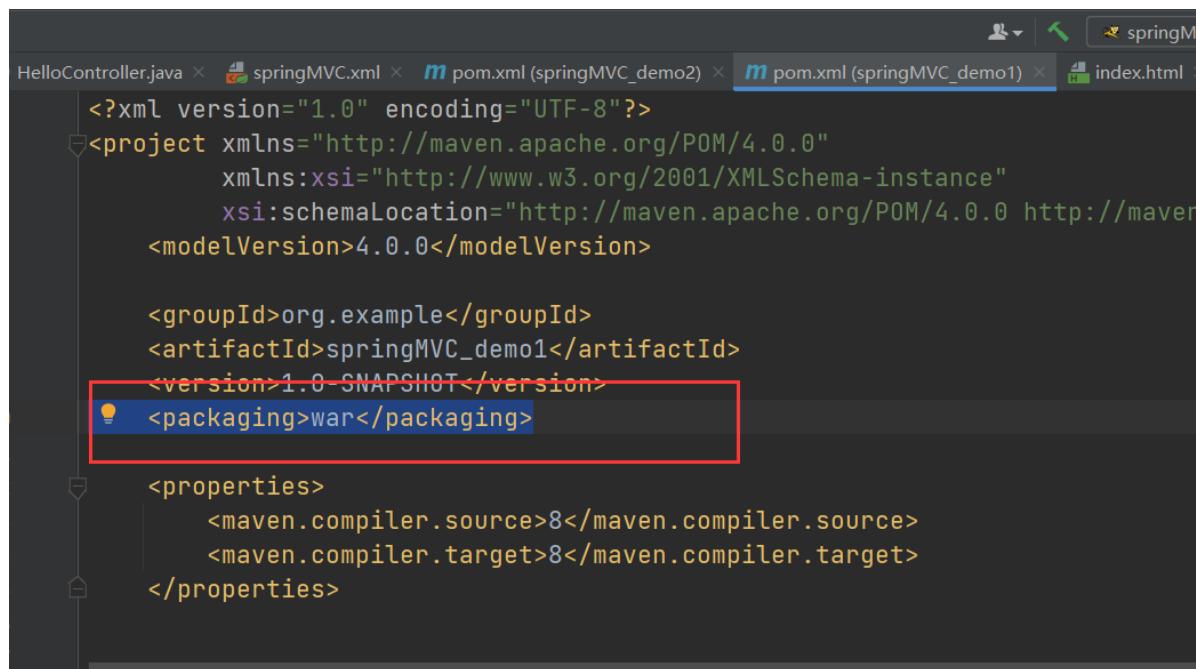
Spring版本: 5.3.1

2、创建maven工程

a>添加web模块

0.先给maven的打包方式设置为

```
<packaging>war</packaging>
```



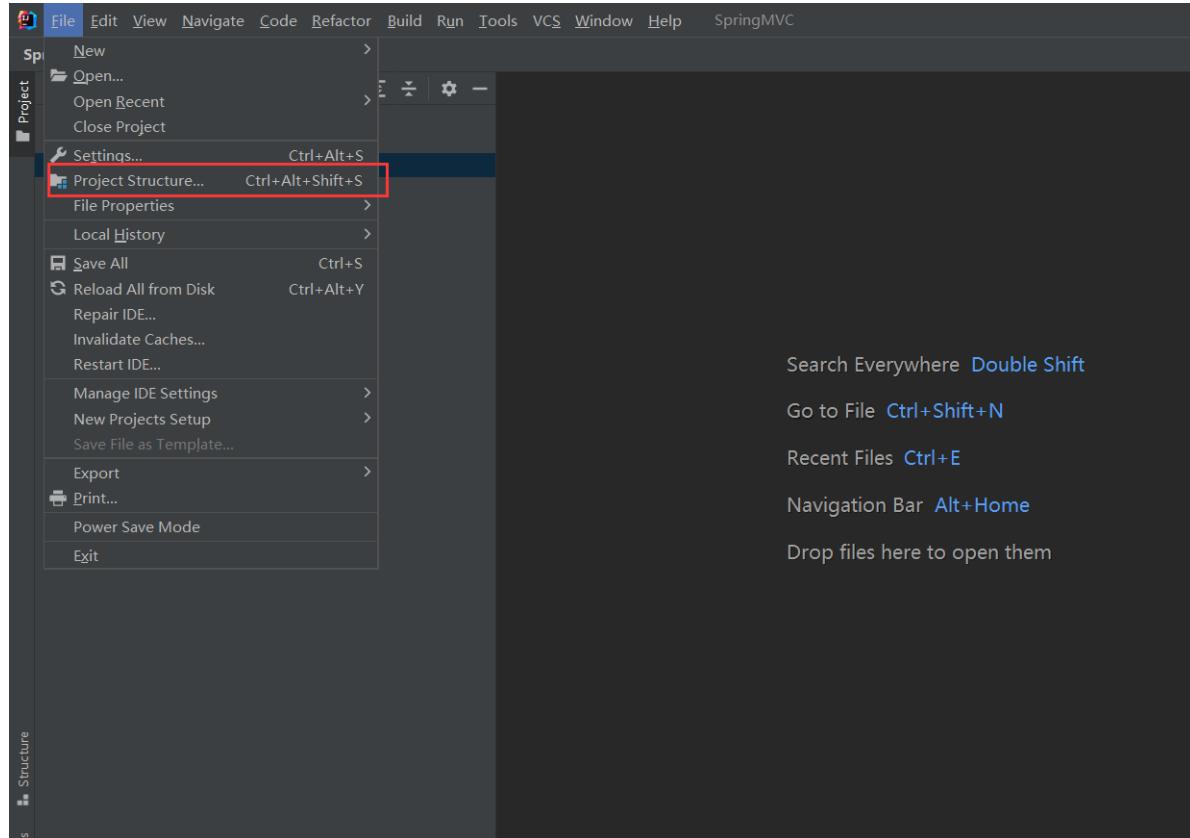
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>springMVC_demo1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

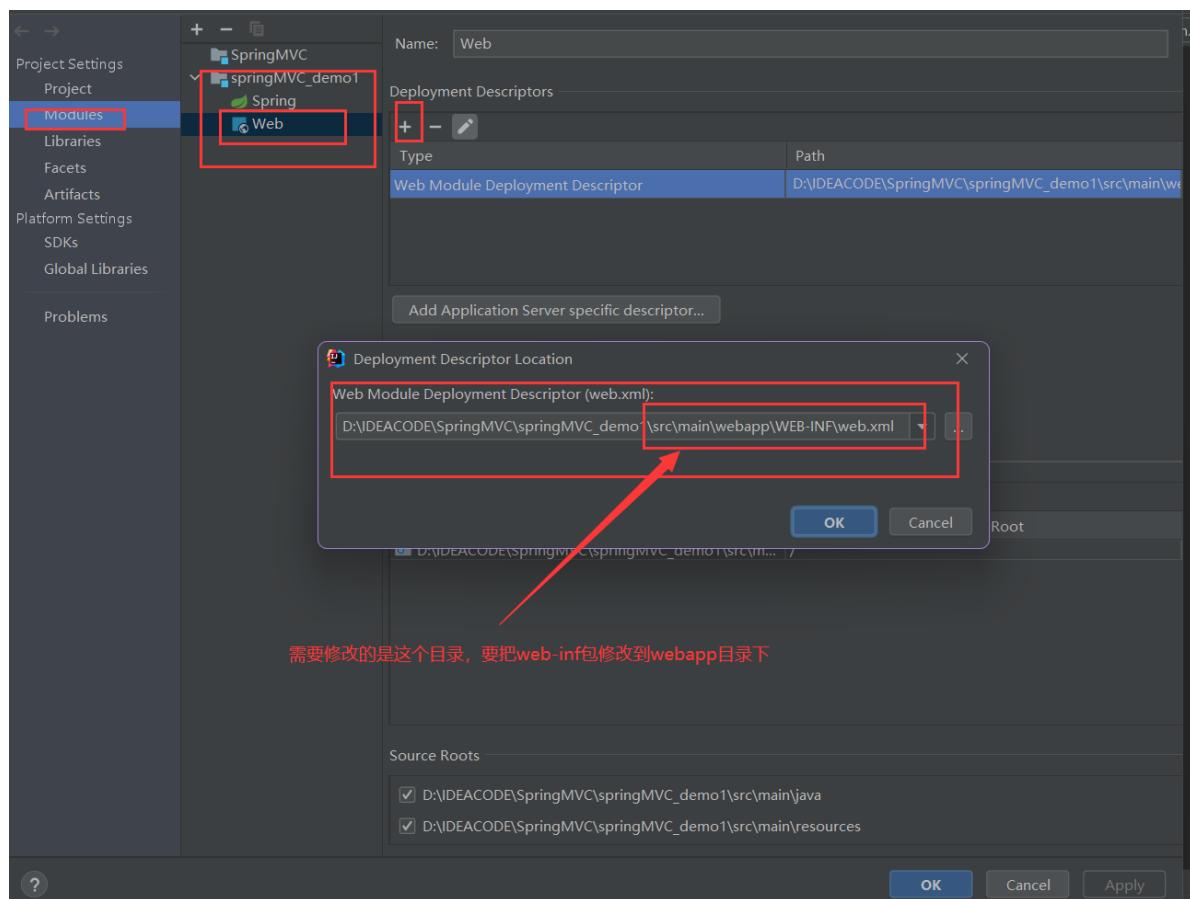
```

1.先在main包下面创建webapp包（如果webapp上面没有小蓝点，需要检查是不是没有打war包，或者是ide故障了）

2.



3.



c>引入依赖

```
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.1</version>
    </dependency>

    <!-- 日志 -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>

    <!-- ServletAPI -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>

    <!-- Spring5和Thymeleaf整合包 -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.0.12.RELEASE</version>
    </dependency>
</dependencies>
```

注：由于 Maven 的传递性，我们不必将所有需要的包全部配置依赖，而是配置最顶端的依赖，其他靠传递性导入。

3、配置web.xml

注册SpringMVC的前端控制器DispatcherServlet

a>默认配置方式

此配置作用下，SpringMVC的配置文件默认位于WEB-INF下，默认名称为<servlet-name>-servlet.xml，例如，以下配置所对应SpringMVC的配置文件位于WEB-INF下，文件名为springMVC-servlet.xml

```
<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
</servlet>
<servlet-mapping>
```

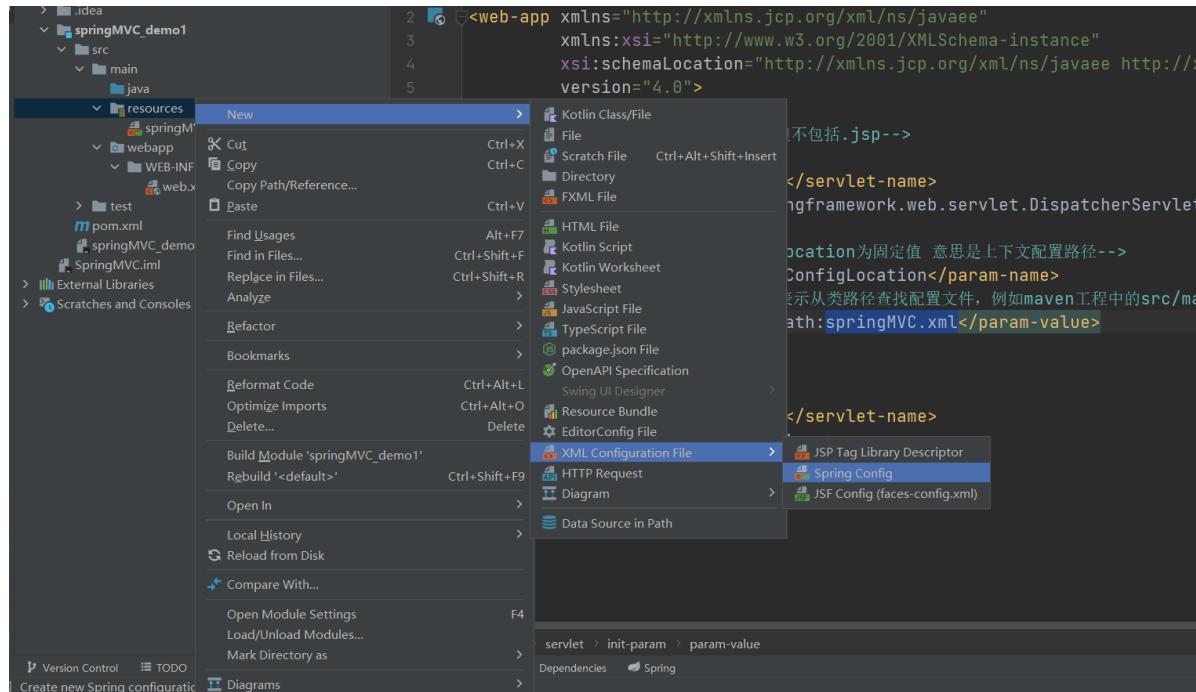
```

<servlet-name>springMVC</servlet-name>
<!--
    设置springMVC的核心控制器所能处理的请求的请求路径
    /所匹配的请求可以是/login或.html或.js或.css方式的请求路径
    但是/不能匹配.jsp请求路径的请求
-->
<url-pattern>/</url-pattern>
</servlet-mapping>

```

b>扩展配置方式 (大部分使用的是这个)

可通过init-param标签设置SpringMVC配置文件的位置和名称，通过load-on-startup标签设置SpringMVC前端控制器DispatcherServlet的初始化事件



```

<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 通过初始化参数指定SpringMVC配置文件的位置和名称 -->

```

```

<!--配置配置文件位置！！！-->
<init-param>
    <!-- contextConfigLocation为固定值 意思是上下文配置路径-->
    <param-name>contextConfigLocation</param-name>
    <!-- 使用classpath:表示从类路径查找配置文件，例如maven工程中的src/main/resources-->
-->
    <param-value>classpath:springMVC.xml</param-value>
</init-param>

```

<!--
作为框架的核心组件，在启动过程中有大量的初始化操作要做

而这些操作放在第一次请求时才执行会严重影响访问速度

因此需要通过此标签

将启动控制DispatcherServlet的初始化时间提前到服务器启动时

```
-->
<load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>springMVC</servlet-name>
  <!--
    设置springMVC的核心控制器所能处理的请求的请求路径
    /所匹配的请求可以是/login或.html或.js或.css方式的请求路径
    但是/不能匹配.jsp请求路径的请求
  -->
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

注：

<url-pattern>标签中使用/和/*的区别：

/所匹配的请求可以是/login或.html或.js或.css方式的请求路径，但是/不能匹配.jsp请求路径的请求

因此就可以避免在访问jsp页面时，该请求被DispatcherServlet处理，从而找不到相应的页面

/*则能够匹配所有请求，例如在使用过滤器时，若需要对所有请求进行过滤，就需要使用/*的写法

4、创建请求控制器

由于前端控制器对浏览器发送的请求进行了统一的处理，但是具体的请求有不同的处理过程，因此需要创建处理具体请求的类，即请求控制器

请求控制器中每一个处理请求的方法成为控制器方法

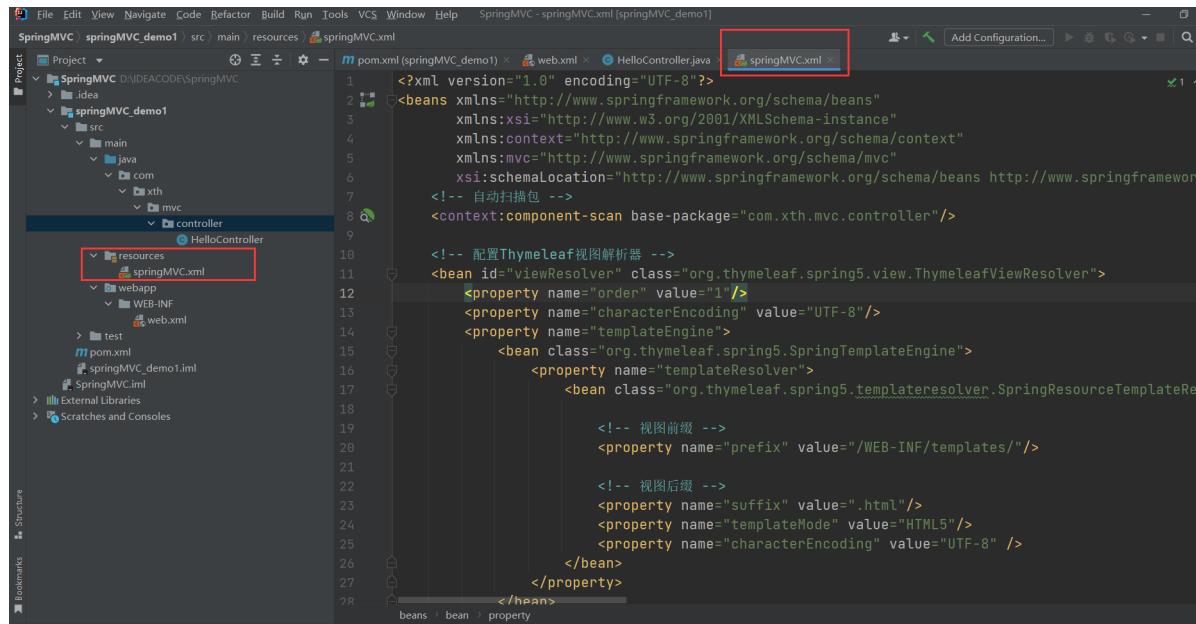
因为SpringMVC的控制器由一个POJO（普通的Java类）担任，因此需要通过@Controller注解将其标识为一个控制层组件，交给Spring的IoC容器管理，此时SpringMVC才能够识别控制器的存在



```
@Controller  
public class HelloController {  
}
```

5、创建springMVC的配置文件

通过扫描组件的方式配置springMVC.xml配置文件



```
<!-- 自动扫描包 -->  
<context:component-scan base-package="com.atguigu.mvc.controller"/>  
  
<!-- 配置Thymeleaf视图解析器 (这个是固定的哦) -->  
<bean id="viewResolver"  
      class="org.thymeleaf.spring5.view.ThymeleafViewResolver">  
    <property name="order" value="1"/>  
    <property name="characterEncoding" value="UTF-8"/>  
    <property name="templateEngine">  
      <bean class="org.thymeleaf.spring5.SpringTemplateEngine">  
        <property name="templateResolver">  
          <bean  
            class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResol  
              <!-- 视图前缀 -->  
              <property name="prefix" value="/WEB-INF/templates/" />
```

```

        <!-- 视图后缀 -->
        <property name="suffix" value=".html"/>
        <property name="templateMode" value="HTML5"/>
        <property name="characterEncoding" value="UTF-8" />
    
```

(现在还不需要这两个标签)

```

<!--
    处理静态资源，例如html、js、css、jpg
    若只设置该标签，则只能访问静态资源，其他请求则无法访问
    此时必须设置<mvc:annotation-driven/>解决问题
-->
<mvc:default-servlet-handler/>

```

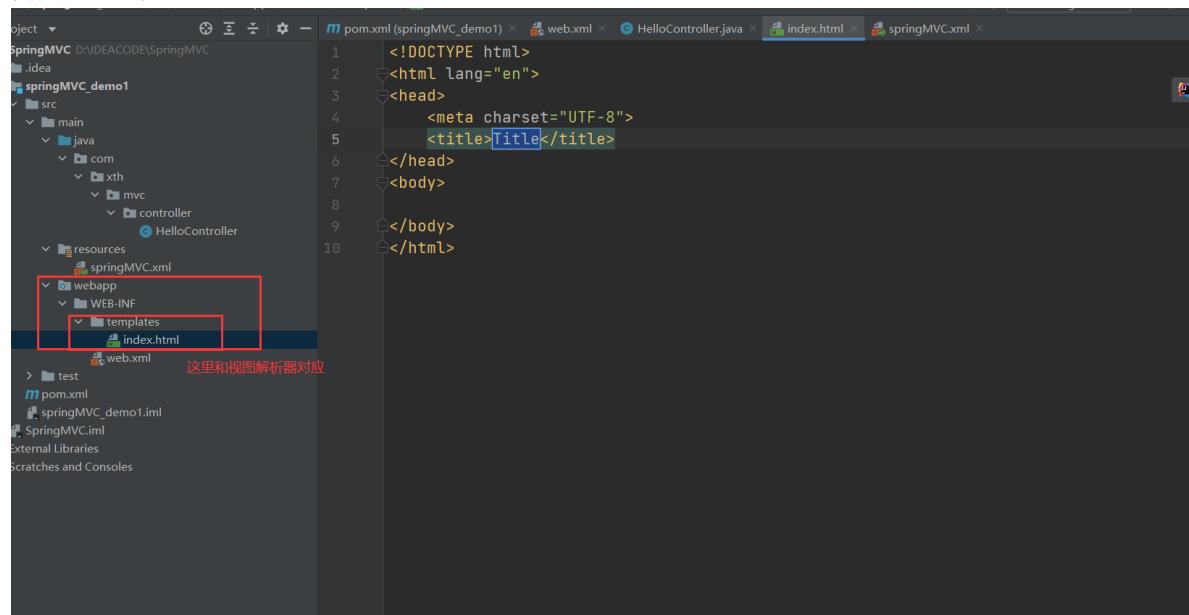
```

<!-- 开启mvc注解驱动 -->
<mvc:annotation-driven>

    <mvc:message-converters>
        <!-- 处理响应中文内容乱码 -->
        <bean
            class="org.springframework.http.converter.StringHttpMessageConverter">
            <property name="defaultCharset" value="UTF-8" />
            <property name="supportedMediaTypes">
                <list>
                    <value>text/html</value>
                    <value>application/json</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

```

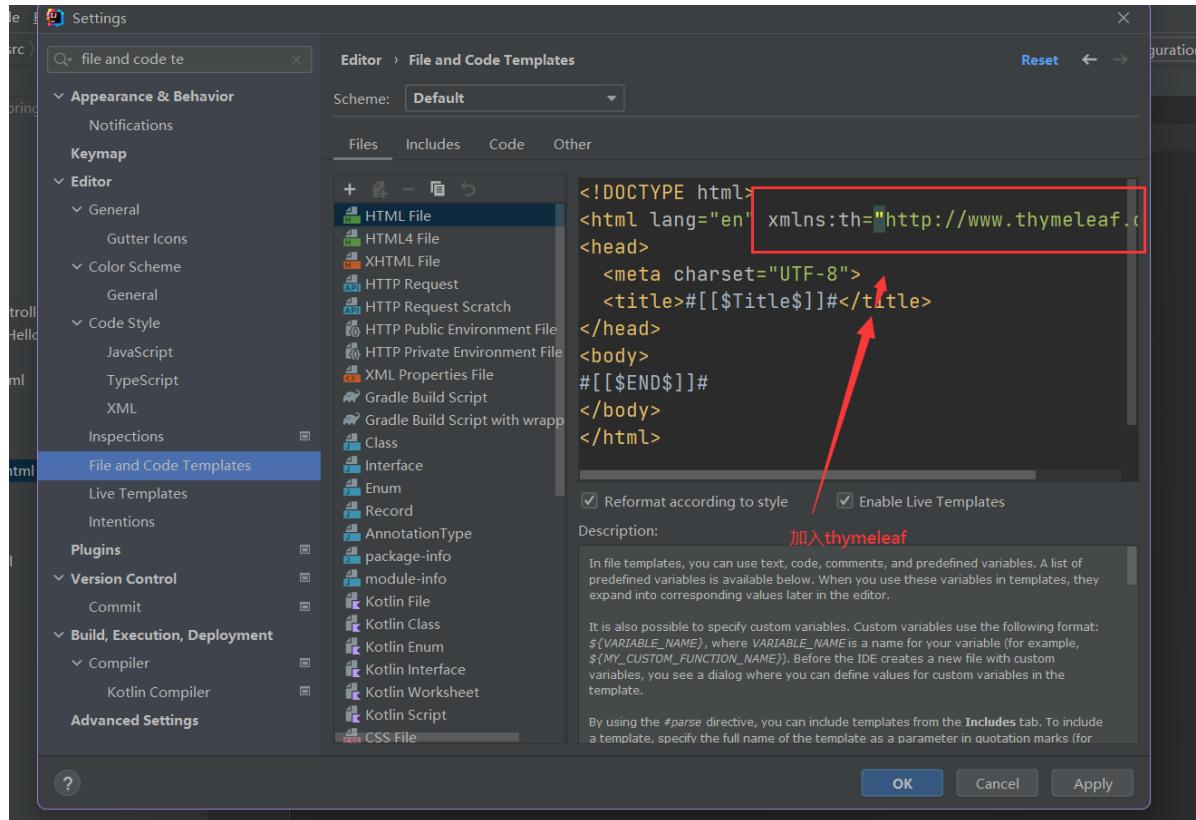
需要注意的是



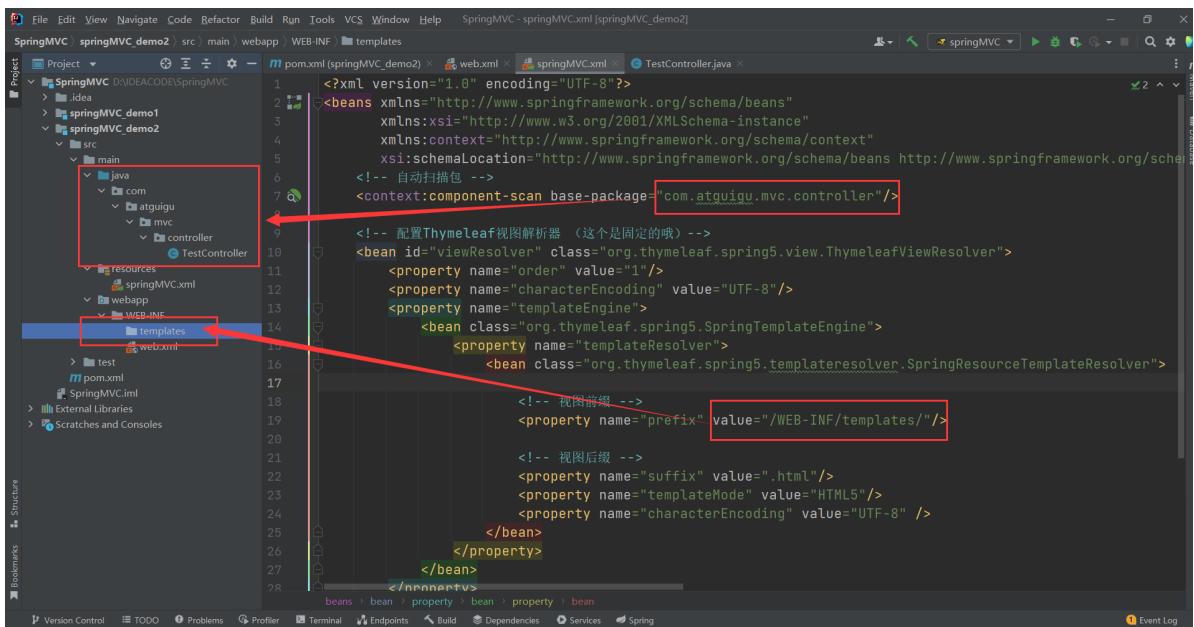
在html模板中加入thymeleaf

```
xmlns:th="http://www.thymeleaf.org"
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
    <h1>首页</h1>
    <!-- 使用thymeleaf解析，如果是绝对路径，会自动加上上下文参数-->
    <a th:href="@{/target}">HelloWorld</a><br/>
</body>
</html>
```



注意创建包的位置和扫描的位置



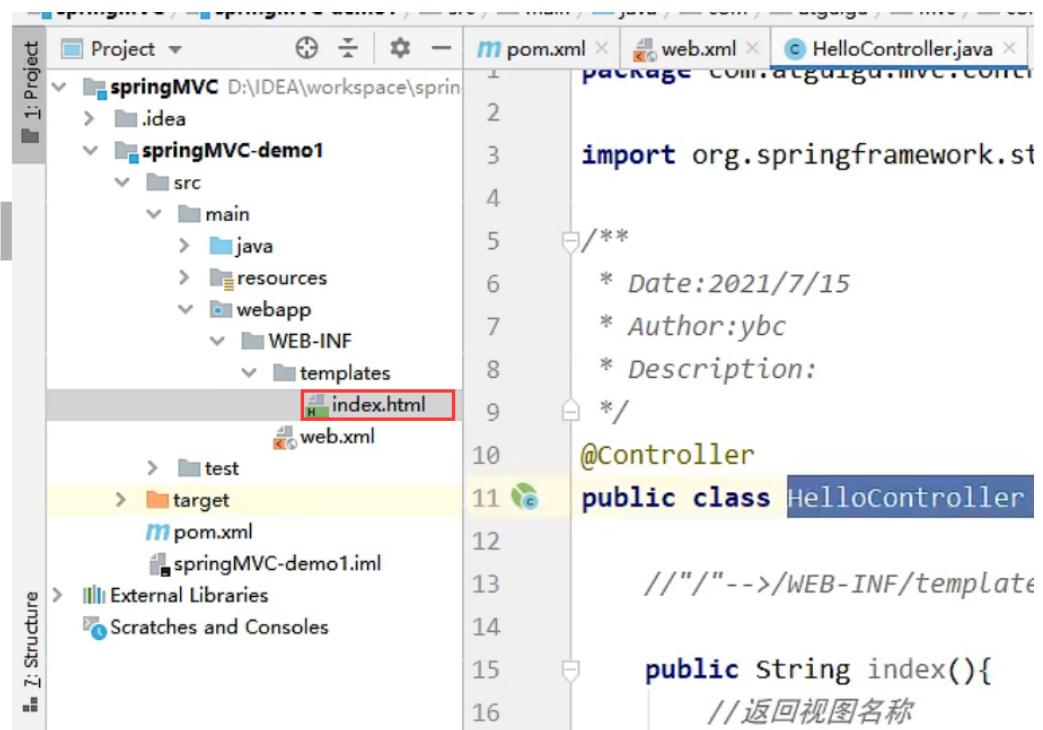
6、测试HelloWorld

工程路径就是上下文路径！！！就是在tomcat服务器配置的路径

a>实现对首页的访问

在请求控制器中创建处理请求的方法

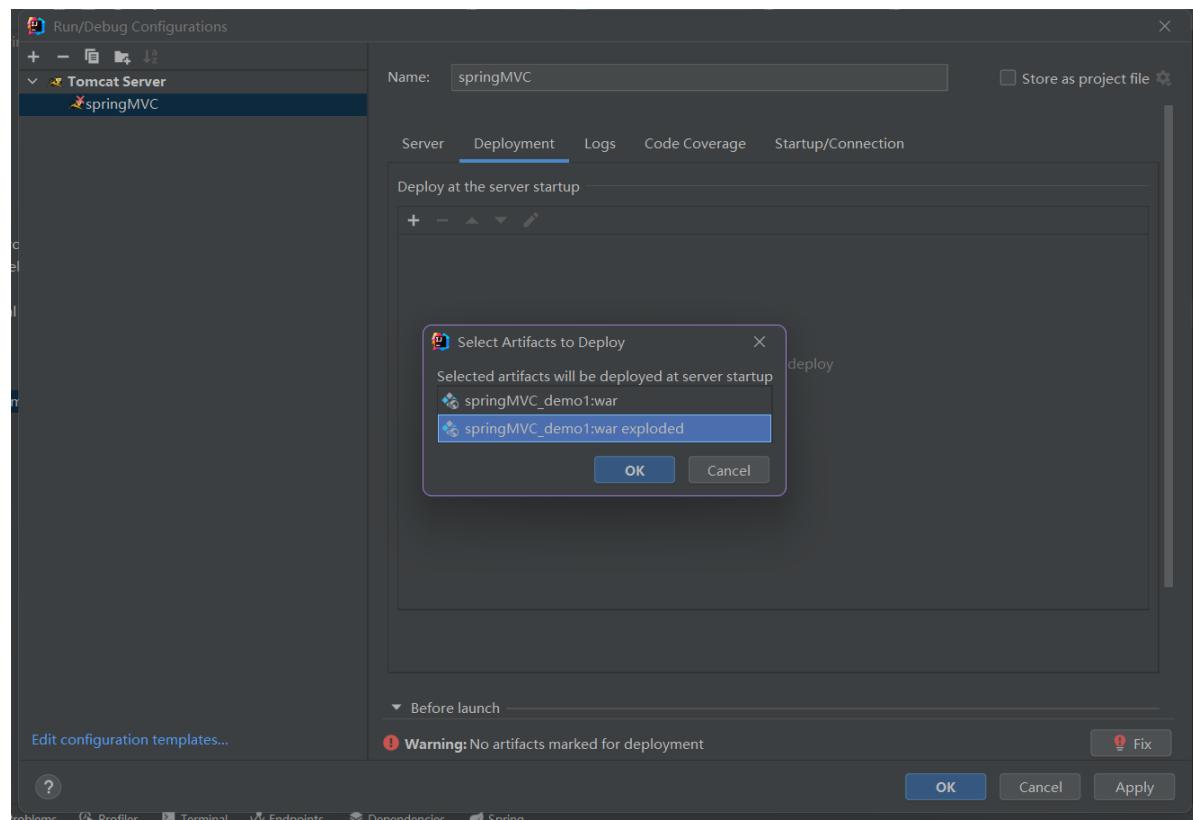
```
// @RequestMapping注解：处理请求和控制器方法之间的映射关系
// @RequestMapping注解的value属性可以通过请求地址匹配请求，/表示的当前工程的上下文路径
// localhost:8080/springMVC/
@RequestMapping(value = "/") //可以写成@RequestMapping("/")这两者是等价的
public String index() {
    //返回设置视图名称
    return "index";
}
//经过thymeleaf和上面的组装得到一个完整的地址http://localhost:8080/webapp/WEB-INF/templates/index.html
```

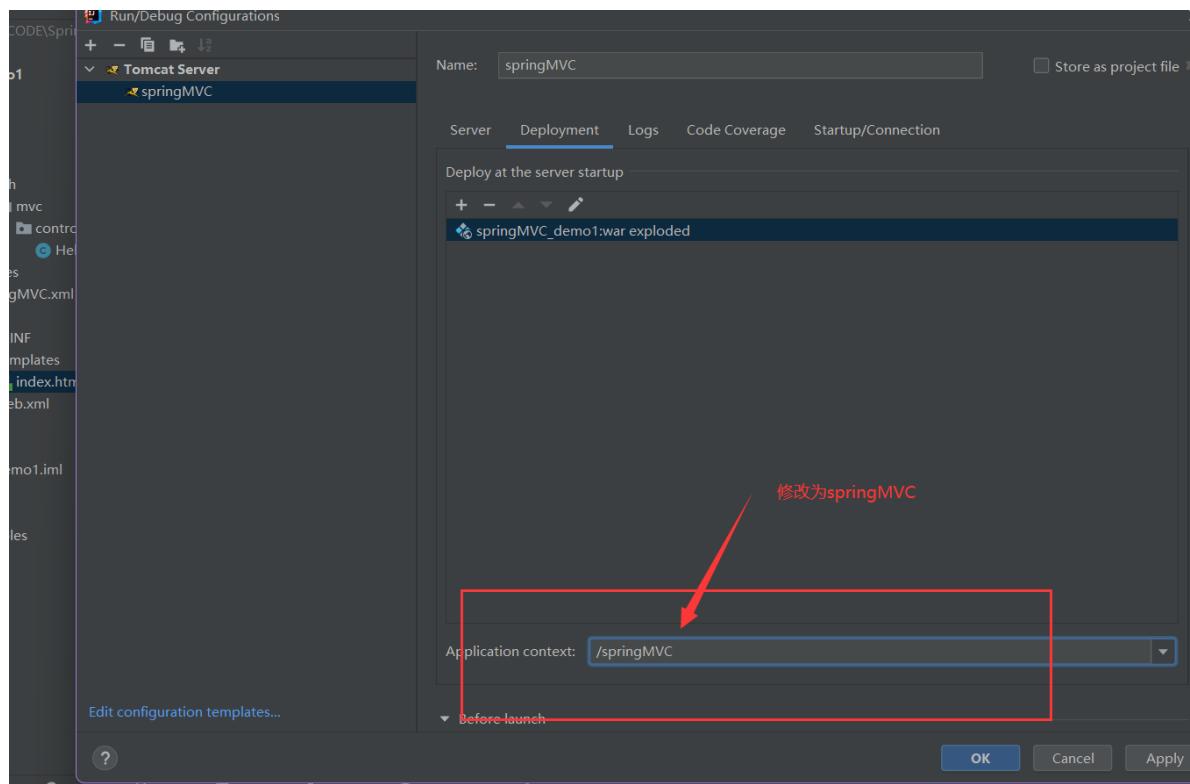


```
// localhost:8080/springMVC/  
@RequestMapping(value="/")//可以写成@RequestMapping("/")这两者是等价的  
public String index() {
```

斜线的意思是上下文路径，也就是工程路径，通过访问这个路径就会访问上面的html页面了

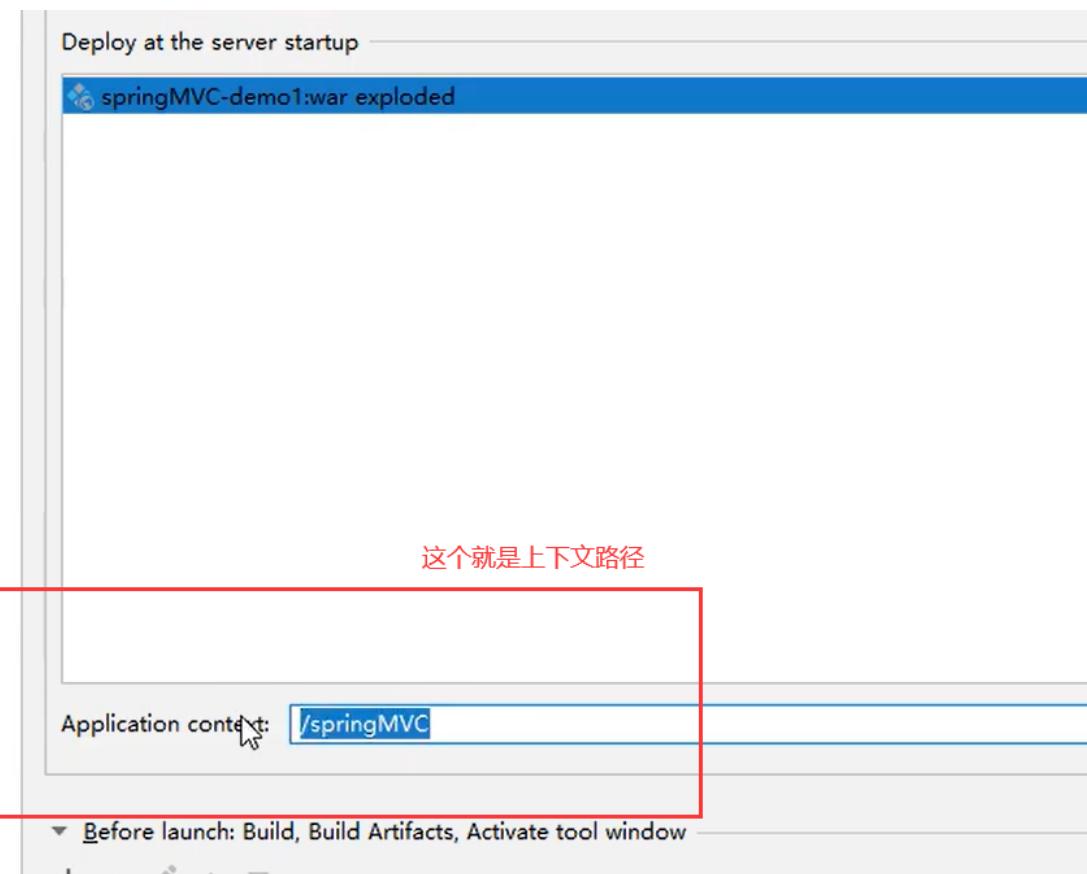
之后需要配置tomcat服务器





b>通过超链接跳转到指定页面

/ 斜杠 如果被浏览器解析，得到的地址是：<http://ip:port/>，如果我们要访问需要加上上下文路径



在主页index.html中设置超链接

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
    <h1>首页</h1>
    <!-- 使用thymeleaf解析，如果是绝对路径，会自动加上上下文参数 -->
    <a th:href="@{/target}">HelloWorld</a><br/>
</body>
</html>

```

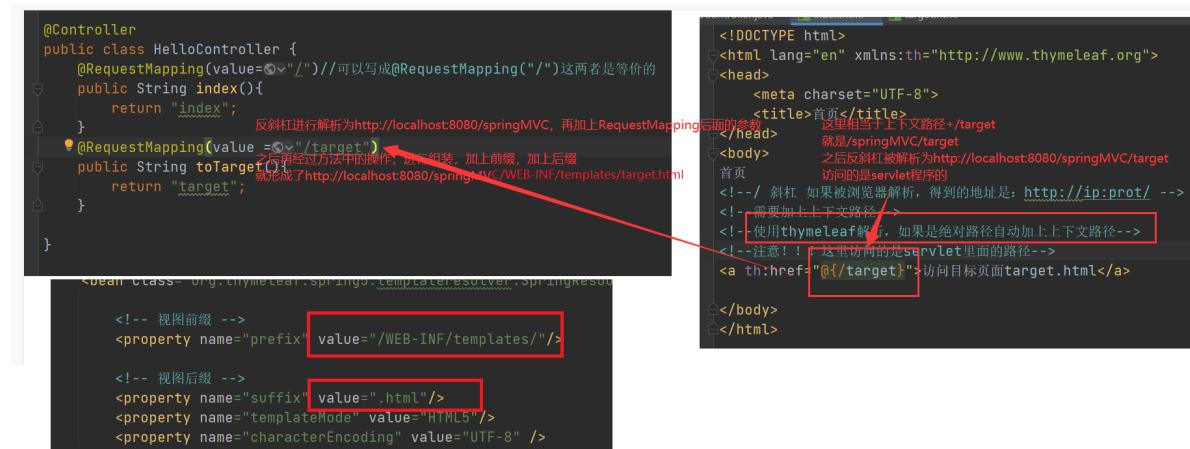
创建target.html

在请求控制器中创建处理请求的方法

```

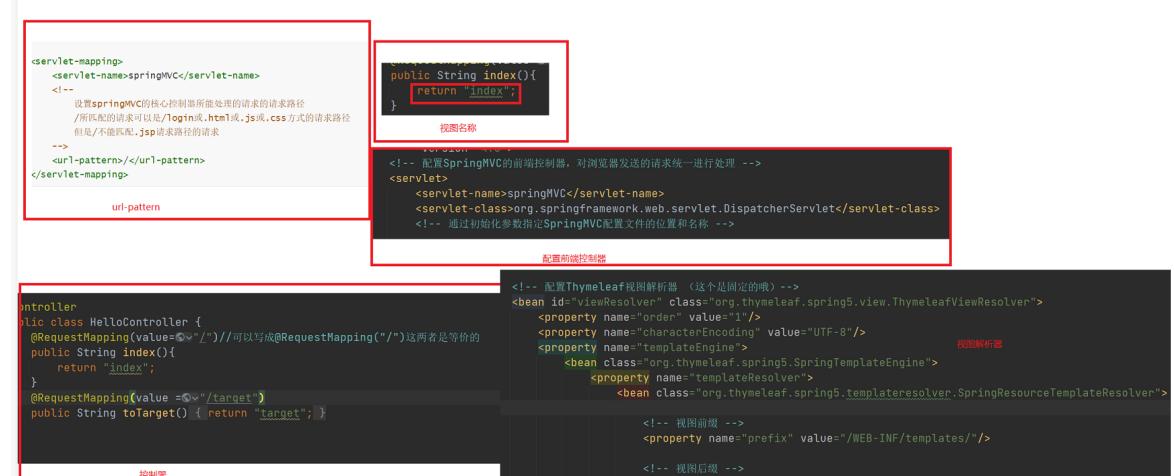
@RequestMapping("/target")
public String HelloWorld() {
    return "target";
}

```



7、总结

url-pattern



浏览器发送请求，若请求地址符合前端控制器的url-pattern，该请求就会被**前端控制器** DispatcherServlet处理。前端控制器会读取 SpringMVC的核心配置文件，通过扫描组件找到控制器，将请求地址和控制器中@RequestMapping注解的value属性值进行匹配，若匹配成功，该注解所标识的控制器方法就是处理请求的方法。处理请求的方法需要返回一个字符串类型的**视图名称**，该**视图名称**会被**视图解析器**解析，加上前缀和后缀组成视图的路径，通过Thymeleaf对视图进行渲染，最终转发到视图所对应页面

三、@RequestMapping注解

1、@RequestMapping注解的功能

从注解名称上我们可以看到，@RequestMapping注解的作用就是将请求和处理请求的控制器方法关联起来，建立映射关系。

SpringMVC 接收到指定的请求，就会来找到在映射关系中对应的控制器方法来处理这个请求。

```
@RequestMapping("/")
public String index(){
    return "index";
}
如果有好几个方法都是这个路径，会报错
```

2、@RequestMapping注解的位置

@RequestMapping标识一个类：设置映射请求的请求路径的初始信息

@RequestMapping标识一个方法：设置映射请求请求路径的具体信息

//此时请求映射所映射的请求的请求路径为：/hello/testRequestMapping

```
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
6   </head>
7   <body>
8     <h1>首页</h1>
9     <a th:href="@{/hello/testRequestMapping}">测试RequestMapping注解的位置</a><br>
10    </body>
11  </html>
```

```

@Controller
@RequestMapping("/hello")
public class RequestMappingController {

    //此时请求映射所映射的请求的请求路径为: /hello/testRequestMapping
    @RequestMapping("/testRequestMapping")
    public String testRequestMapping(){
        return "success";
    }

}

```

3、@RequestMapping注解的value属性

@RequestMapping注解的value属性通过请求的请求地址匹配请求映射

@RequestMapping注解的value属性是一个字符串类型的数组，表示该请求映射能够匹配多个请求地址所对应的请求

@RequestMapping注解的value属性必须设置，至少通过请求地址匹配请求映射

```

<a th:href="@{/testRequestMapping}">测试@RequestMapping的value属性-->/testRequestMapping</a><br>
<a th:href="@{/test}">测试@RequestMapping的value属性-->/test</a><br>

```

```

@RequestMapping(value = {"/testRequestMapping", "/test"})//请求地址满足任何一个就k
public String testRequestMapping(){
    return "success";
}

```

4、@RequestMapping注解的method属性

@RequestMapping注解的method属性通过请求的请求方式（get或post）匹配请求映射

@RequestMapping注解的method属性是一个RequestMethod类型的数组，表示该请求映射能够匹配多种请求方式的请求

若当前请求的请求地址满足请求映射的value属性，但是请求方式不满足method属性，则浏览器报错
405: Request method 'POST' not supported

```

<a th:href="@{/test}">测试@RequestMapping的value属性-->/test</a><br>
<form th:action="@{/test}" method="post">
    <input type="submit">
</form>

```

```

@RequestMapping(
    value = {"/testRequestMapping", "/test"},
    method = {RequestMethod.GET, RequestMethod.POST}
)
public String testRequestMapping(){
    return "success";
}

```

注：

1、对于处理指定请求方式的控制器方法，SpringMVC中提供了@RequestMapping的派生注解
处理get请求的映射-->@GetMapping

处理post请求的映射-->@PostMapping

处理put请求的映射-->@PutMapping

处理delete请求的映射-->@DeleteMapping

2、常用的请求方式有get, post, put, delete

但是目前浏览器只支持get和post，若在form表单提交时，为method设置了其他请求方式的字符串（put或delete），则按照默认的请求方式get处理

若要发送put和delete请求，则需要通过spring提供的过滤器HiddenHttpMethodFilter，在RESTful部分会讲到

5、@RequestMapping注解的params属性（了解）

@RequestMapping注解的params属性通过请求的请求参数匹配请求映射

@RequestMapping注解的params属性是一个字符串类型的数组，可以通过四种表达式设置请求参数和请求映射的匹配关系

"param": 要求请求映射所匹配的请求必须携带param请求参数

"!param": 要求请求映射所匹配的请求必须不能携带param请求参数

"param=value": 要求请求映射所匹配的请求必须携带param请求参数且param=value（注意是单等号）

"param!=value": 要求请求映射所匹配的请求必须携带param请求参数但是param!=value

```
<a th:href="@{/test(username='admin',password=123456)}>测试@RequestMapping的params属性-->/test</a><br>
后面的参数也可以用？连接，不过会有错，不过没事，上面是不报错的方法
```

```
@RequestMapping(
    value = {"/testRequestMapping", "/test"}
    ,method = {RequestMethod.GET, RequestMethod.POST}
    ,params = {"username", "password=123456"}
)
public String testRequestMapping(){
    return "success";
}
```

注：

若当前请求满足@RequestMapping注解的value和method属性，但是不满足params属性，此时页面回报错400： Parameter conditions "username, password!=123456" not met for actual request parameters: username={admin}, password={123456}

6、@RequestMapping注解的headers属性（了解）

@RequestMapping注解的headers属性通过请求的**请求头**信息匹配请求映射

@RequestMapping注解的headers属性是一个字符串类型的数组，可以通过四种表达式设置请求头信息和请求映射的匹配关系

"header": 要求请求映射所匹配的请求必须携带header请求头信息

"!header": 要求请求映射所匹配的请求必须不能携带header请求头信息

"header=value": 要求请求映射所匹配的请求必须携带header请求头信息且header=value

"header!=value": 要求请求映射所匹配的请求必须携带header请求头信息且header!=value

若当前请求满足@RequestMapping注解的value和method属性，但是不满足headers属性，此时页面显示404错误，即资源未找到

7、SpringMVC支持ant风格的路径

支持模糊匹配

可以在地址上加上模糊匹配占位符

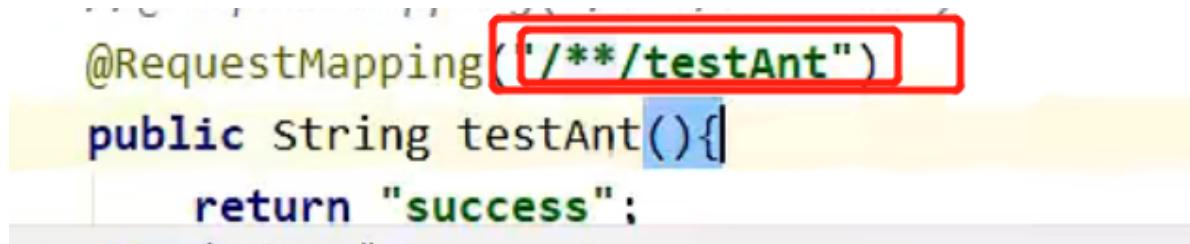


? : 表示任意的单个字符

* : 表示任意的0个或多个字符

** : 表示任意的一层或多层目录

注意：在使用**时，只能使用/**/xxx的方式



8、SpringMVC支持路径中的占位符（重点）

原始方式: /deleteUser?id=1

rest方式: /deleteUser/1

SpringMVC路径中的占位符常用于RESTful风格中，当请求路径中将某些数据通过路径的方式传输到服务器中，就可以在相应的@RequestMapping注解的value属性中通过占位符{xxx}表示传输的数据，在通过@PathVariable注解，将占位符所表示的数据赋值给控制器方法的形参

```
<a th:href="@{/testRest/1/admin}">测试路径中的占位符-->/testRest</a><br>
```

```
@RequestMapping("/testRest/{id}/{username})//加这些说明必定是有参数的，如果不传参数，是匹配不上的
public String testRest(@PathVariable("id") String id, @PathVariable("username")
String username){
    System.out.println("id:"+id+",username:"+username);
    return "success";
}
//最终输出的内容为-->id:1,username:admin
```

四、SpringMVC获取请求参数

1、通过ServletAPI获取

将HttpServletRequest作为控制器方法的形参，此时HttpServletRequest类型的参数表示封装了当前请求的请求报文的对象

```
<html>
<body>
    <h1>测试请求参数</h1>
    <a th:href="@{/testServletAPI(username='admin',password=123456)}">测试使用servletAPI获取请求
</body>
</html>
```

```
@RequestMapping("/testParam")
public String testParam(HttpServletRequest request){//可以直接调用这个方法，是由
dispatcher赋值的，表示当前请求
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    System.out.println("username:"+username+",password:"+password);
    return "success";
}
```

2、通过控制器方法的形参获取请求参数

在控制器方法的**形参位置**，设置和请求参数**同名**的形参，当浏览器发送请求，匹配到请求映射时，在DispatcherServlet中就会将请求参数赋值给相应的形参

```
<a th:href="@{/testParam(username='admin',password=123456)}">测试获取请求参数-->/testParam</a><br>
```

```

@RequestMapping("/testParam")
public String testParam(String username, String password){
    System.out.println("username:"+username+",password:"+password);
    return "success";
}

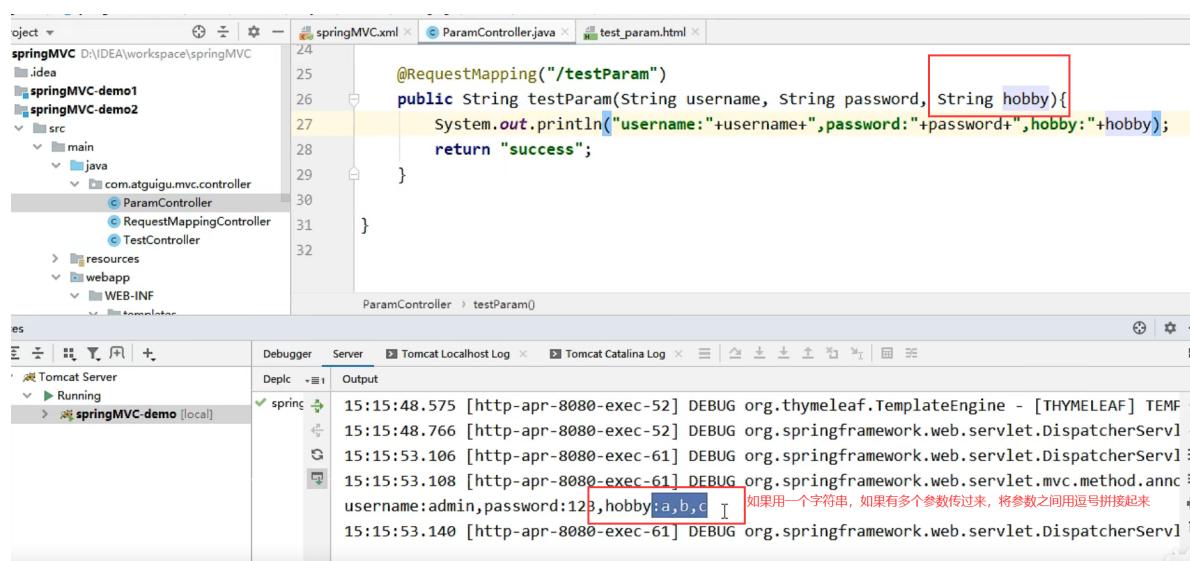
```

注：

若请求所传输的请求参数中有多个同名的请求参数，此时可以在控制器方法的形参中设置字符串数组或者字符串类型的形参接收此请求参数

若使用字符串数组类型的形参，此参数的数组中包含了每一个数据

若使用字符串类型的形参，此参数的值为每个数据中间使用逗号拼接的结果



也可以用这种方式

The screenshot shows a Java code editor and a terminal window. The code in the editor is:

```
    pping("/testParam")
    ing testParam(String username, String password, String[] hobby){
        .out.println("username:"+username+",password:"+password+",hobby:"+ Arrays.toString(hobby));
        "success";
    }
}
```

The line `.out.println("username:"+username+",password:"+password+",hobby:"+ Arrays.toString(hobby));` is highlighted with a red box. The word `Arrays.toString(hobby)` is also highlighted with a red box.

The terminal window below shows the output of the application:

```
Server Tomcat Localhost Log × Tomcat Catalina Log ×
Output
[THYMELEAF] * "StandardVariableExpressionEvaluator": SpringEL
[THYMELEAF] * "StandardConversionService": org.thymeleaf.spring5.expression.Spring
[THYMELEAF] TEMPLATE ENGINE CONFIGURED OK
15:17:40.679 [http-apr-8080-exec-67] DEBUG org.thymeleaf.TemplateEngine - [THYMELEAF] TEMP
15:17:40.876 [http-apr-8080-exec-67] DEBUG org.springframework.web.servlet.DispatcherServl
15:17:44.061 [http-apr-8080-exec-56] DEBUG org.springframework.web.servlet.DispatcherServl
15:17:44.062 [http-apr-8080-exec-56] DEBUG org.springframework.web.mvc.method.annotatio
username:admin,password:123,hobby:[a, b, c]
15:17:44.093 [http-apr-8080-exec-56] DEBUG org.springframework.web.servlet.DispatcherServl
```

The line `username:admin,password:123,hobby:[a, b, c]` is highlighted with a red box.

3、@RequestParam

跟这个很像

8、SpringMVC支持路径中的占位符（重点）

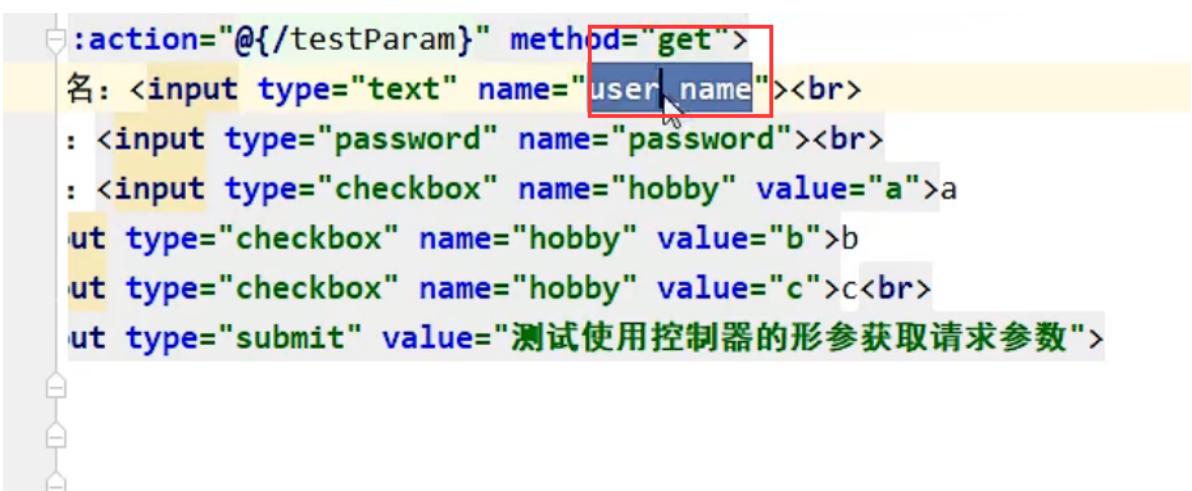
原始方式: /deleteUser?id=1

rest方式: /deleteUser/1

SpringMVC路径中的占位符常用于RESTful风格中，当请求路径中将某些数据通过路径的方式传输到服务器中，就可以在相应的@RequestMapping注解的value属性中通过占位符{xxx}表示传输的数据，在通过@PathVariable注解，将占位符所表示的数据赋值给控制器方法的形参

测试路径中的占位符-->/testRest


```
@RequestMapping("/testRest/{id}/{username})//加这些说明必定是有参数的,如果不传参数,是匹配不上的
public String testRest(@PathVariable("id") String id, @PathVariable("username") String username){
    System.out.println("id:"+id+",username:"+username);
    return "success";
}
//最终输出的内容为-->id:1.username:admin
```



通过映射关系

```
@RequestMapping("/testParam")
public String testParam(
    @RequestParam("user_name") String username,
    String password,
    String[] hobby){
    //若请求参数中出现多个同名的请求参数，可以在控制器方法的形参位置设置字符串类型或字符串数组接收此参数
    //若使用字符串类型的形参，最终结果为请求参数的每一个值之间使用逗号进行拼接
    System.out.println("username:"+username+",password:"+password+",hobby:"+ Arrays.toString(hobby));
    return "success";
}
```

@RequestParam是将请求参数和控制器方法的形参创建映射关系

@RequestParam注解一共有三个属性：

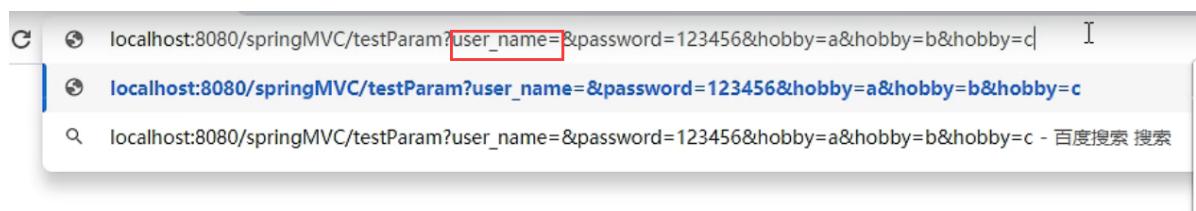
value：指定为形参赋值的请求参数的参数名

required：设置是否必须传输此请求参数，默认值为true（意思是@（RequestParam（“这里面的参数如果如果不传输，就报错”）））

不传输的意思是，如果浏览器给服务器发送的参数没有传这个参数或者参数不对，就会报错

若设置为true时，则当前请求必须传输value所指定的请求参数，**若没有传输该请求参数，且没有设置defaultValue属性**，则页面报错400：Required String parameter 'xxx' is not present；若设置为false，则当前请求不是必须传输value所指定的请求参数，若没有传输，则注解所标识的形参的值为null

defaultValue：不管required属性值为true或false，当value所指定的请求参数**没有传输或传输的值为""时**，则使用默认值为形参赋值



这样相当于传输了一个空字符串

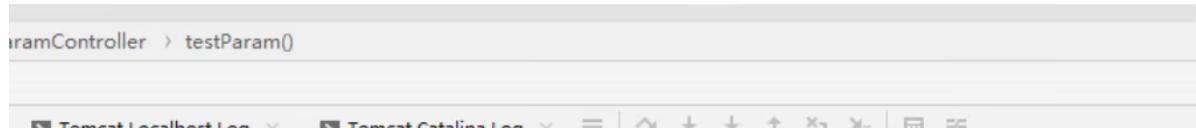
4、@RequestHeader

请求头信息没有对应的映射，必须手动引用

```

@RequestMapping("/testParam")
public String testParam(
    @RequestParam(value = "user_name", required = false, defaultValue = "匿名")
    String password,
    String[] hobby,   请求头
    @RequestHeader("Host") String host){
    //若请求参数中出现多个同名的请求参数，可以在控制器方法的形参位置设置字符串类型或字符串数组接收此请求
    //若使用字符串类型的形参，最终结果为请求参数的每一个值之间使用逗号进行拼接
    System.out.println("username:"+username+",password:"+password+",host:"+host);
    System.out.println("host:"+host);
    return "success";
}

```



@RequestHeader是将请求头信息和控制器方法的形参创建映射关系

@RequestHeader注解一共有三个属性：value、required、defaultValue，用法同@RequestParam

5、@CookieValue

@CookieValue是将cookie数据和控制器方法的形参创建映射关系

@CookieValue注解一共有三个属性：value、required、defaultValue，用法同@RequestParam

```

        String password,
        String[] hobby,
        @RequestHeader(value = "sayHaha", required = true, defaultValue = "haha") String host
        @CookieValue("JSESSIONID") String JSESSIONID{
    //若请求参数中出现多个同名的请求参数，可以在控制器方法的形参位置设置字符串类型或字符串数组接收此请求
    //若使用字符串类型的形参，最终结果为请求参数的每一个值之间使用逗号进行拼接
    System.out.println("username:"+username+",password:"+password+",hobby:"+ Arrays.toString(hobby));
    System.out.println("host:"+host);
    System.out.println("JSESSIONID:"+JSESSIONID);
    return "success";
}

```

6、通过POJO获取请求参数

可以在控制器方法的形参位置设置一个实体类类型的形参，此时若浏览器传输的请求参数的参数名和实体类中的属性名一致，那么请求参数就会为此属性赋值

注意pojo类需要有get和set方法和toString方法

```

<form th:action="@{/testpojo}" method="post">
    用户名: <input type="text" name="username"><br>
    密码: <input type="password" name="password"><br>
    性别: <input type="radio" name="sex" value="男">男<input type="radio"
    name="sex" value="女">女<br>
    年龄: <input type="text" name="age"><br>
    邮箱: <input type="text" name="email"><br>
    <input type="submit">
</form>

```

```

@RequestMapping("/testpojo")
public String testPOJO(User user){
    System.out.println(user);
    return "success";
}
//最终结果-->User{id=null, username='张三', password='123', age=23, sex='男',
email='123@qq.com'}

```

如果pojo类中有引用类型的属性，可以这样赋值

比如有一个Address address,address里面有city和province属性，可以这样赋值

address.city	beijing
address.province	beijing

集合参数

```

//集合参数
@RequestMapping("/listParam")
@ResponseBody
public String listParam(@RequestParam List<String> likes){
    System.out.println("集合参数传递 likes ==> "+ likes);
    return "{ 'module':'list param'}";
}

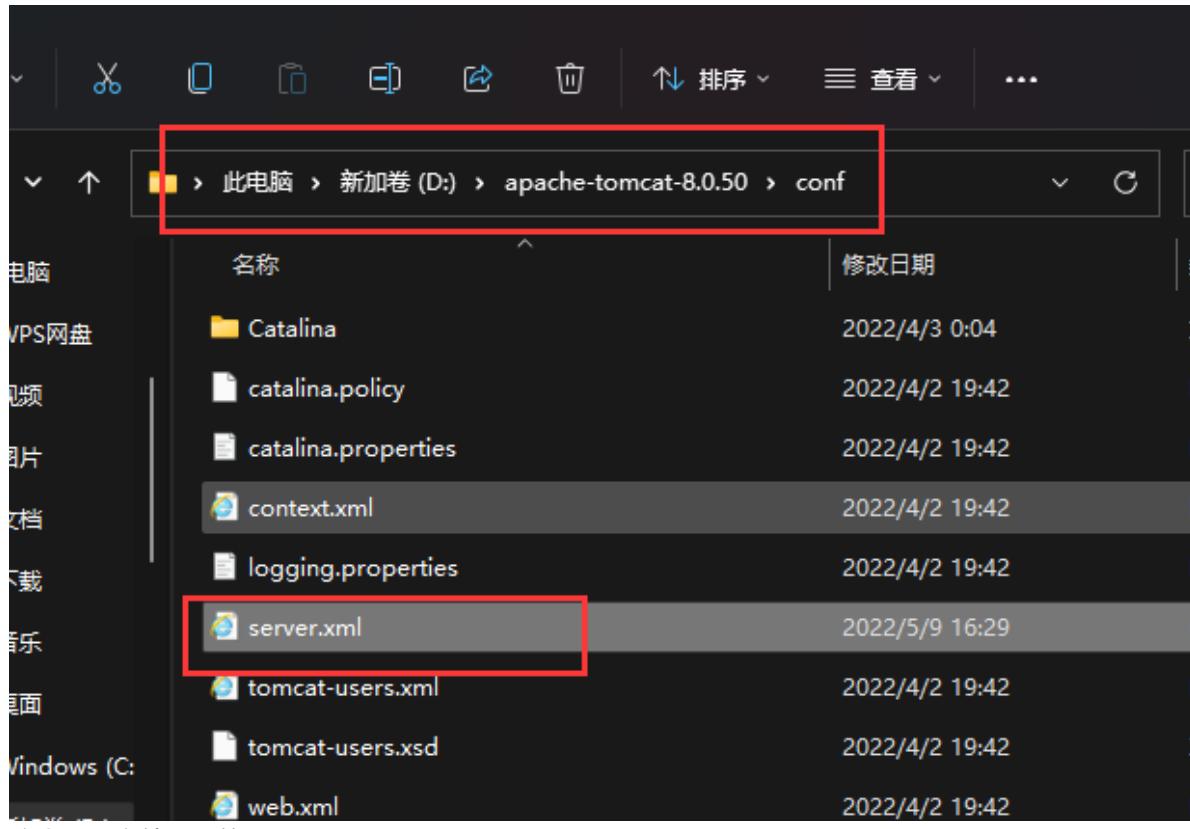
```

	KEY	VALUE	DESCRIPTION	...	Bulk
<input checked="" type="checkbox"/>	likes	game			
<input checked="" type="checkbox"/>	likes	music			
<input checked="" type="checkbox"/>	likes	travel			
	Key	Value	Description		

7、解决获取请求参数的乱码问题

只有post才会出现乱码

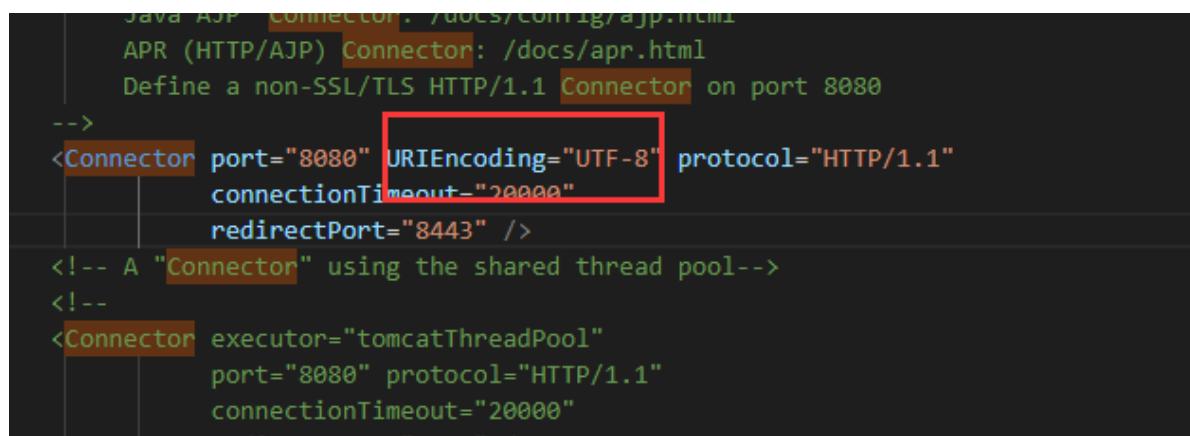
get请求不会出现乱码，get请求的乱码是由tomcat造成的，有了这个tomcat就不会出现乱码了，这个是



A screenshot of a Windows File Explorer window. The address bar shows the path: This PC > New Volume (D:) > apache-tomcat-8.0.50 > conf. A red box highlights this path. The main area displays a list of files and folders:

名称	修改日期
Catalina	2022/4/3 0:04
catalina.policy	2022/4/2 19:42
catalina.properties	2022/4/2 19:42
context.xml	2022/4/2 19:42
logging.properties	2022/4/2 19:42
server.xml	2022/5/9 16:29
tomcat-users.xml	2022/4/2 19:42
tomcat-users.xsd	2022/4/2 19:42
web.xml	2022/4/2 19:42

这个配置文件里面的



```
Java APR Connector: /docs/config/ajp.html
APR (HTTP/AJP) Connector: /docs/apr.html
Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
-->
<Connector port="8080" URIEncoding="UTF-8" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
<!-- A "Connector" using the shared thread pool--&gt;
&lt;!--
&lt;Connector executor="tomcatThreadPool"
           port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           maxThreads="150" /&gt;</pre>
```

解决获取请求参数的乱码问题，可以使用SpringMVC提供的编码过滤器CharacterEncodingFilter，但是必须在web.xml中进行注册

```
<!--配置springMVC的编码过滤器-->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <!--设置编码 -->
        <!--这里默认设置请求的编码也是UTF-8编码 -->
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <!--把响应的编码格式设置为UTF-8编码 -->
        <init-param>
            <param-name>forceResponseEncoding</param-name>
            <param-value>true</param-value>
```

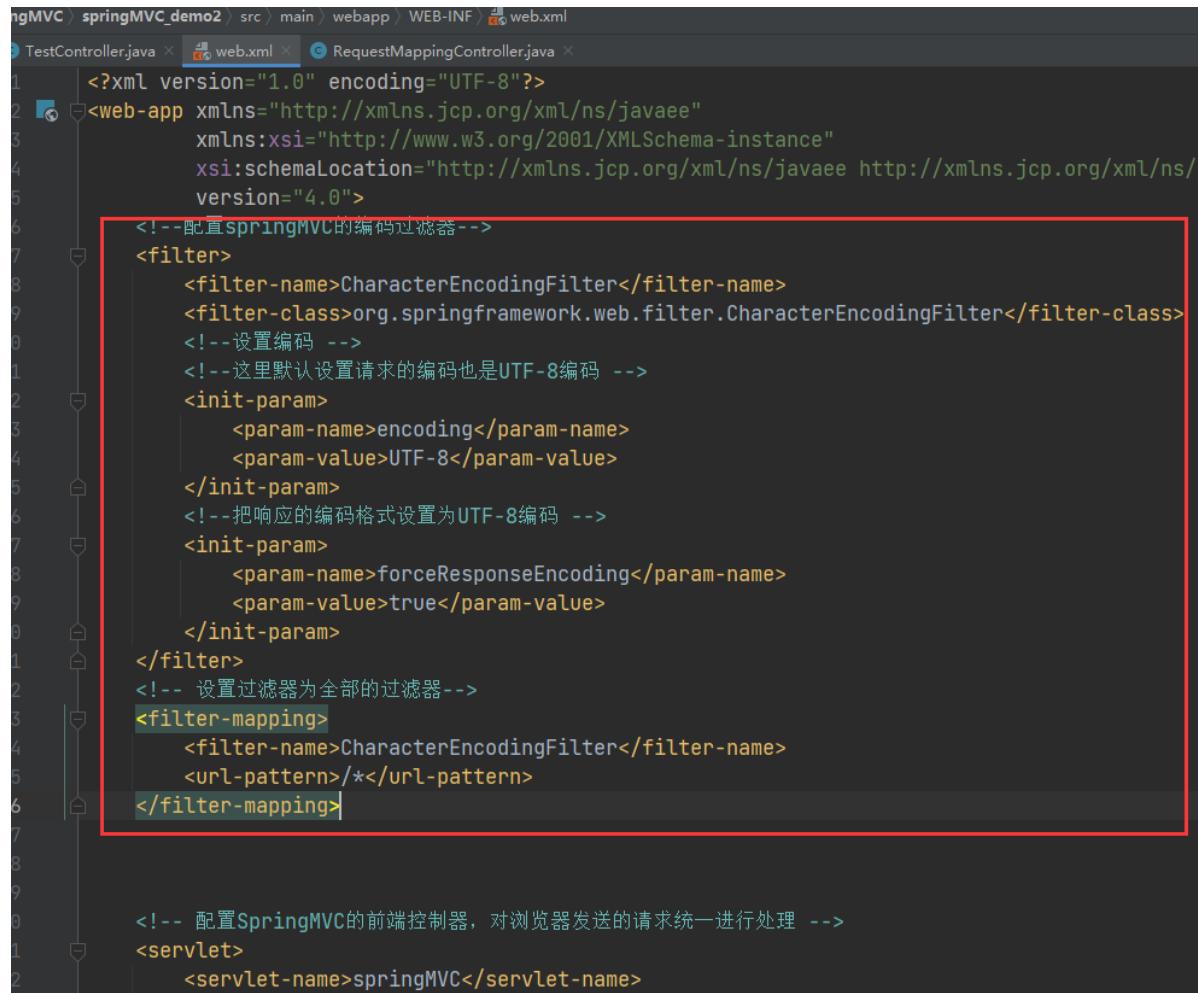
```

        </init-param>
    </filter>
    <!-- 设置过滤器为全部的过滤器-->
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

```

注：

SpringMVC中处理编码的过滤器一定要配置到其他过滤器之前，否则无效



```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/
    version="4.0">
    <!--配置springMVC的编码过滤器-->
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <!--设置编码 -->
        <!--这里默认设置请求的编码也是UTF-8编码 -->
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <!--把响应的编码格式设置为UTF-8编码 -->
        <init-param>
            <param-name>forceResponseEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <!-- 设置过滤器为全部的过滤器-->
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
    <servlet>
        <servlet-name>springMVC</servlet-name>

```

五、域对象共享数据

1、使用ServletAPI向request域对象共享数据

```

@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request){
    request.setAttribute("testRequestScope", "hello,servletAPI");
    return "success";
}

```

取出就这样取出



2、使用 ModelAndView 向 request 域对象共享数据

```
@RequestMapping("/testModelAndView")
public ModelAndView testModelAndView(){
    /**
     * ModelAndView有Model和view的功能
     * Model主要用于向请求域共享数据
     * view主要用于设置视图，实现页面跳转
     */
    ModelAndView mav = new ModelAndView();
    //向请求域共享数据
    mav.addObject("testScope", "hello, ModelAndView");
    //设置视图，实现页面跳转
    mav.setViewName("success");
    return mav;
}
```

3、使用 Model 向 request 域对象共享数据

```
@RequestMapping("/testModel")
public String testModel(Model model){ //model指的就是modelAndView中的model
    model.addAttribute("testScope", "hello, Model");
    return "success";
}
```

4、使用 map 向 request 域对象共享数据

```
@RequestMapping("/testMap")
public String testMap(Map<String, Object> map){
    map.put("testScope", "hello, Map");
    return "success";
}
```

5、使用ModelMap向request域对象共享数据

```
@RequestMapping("/testModelMap")
public String testModelMap(ModelMap modelMap){
    modelMap.addAttribute("testScope", "hello,ModelMap");
    return "success";
}
```

6、Model、ModelMap、Map的关系

Model、ModelMap、Map类型的参数其实本质上都是 BindingAwareModelMap 类型的，也就是说 BindingAwareModelMap 是他们的实现类

```
public interface Model{}
public class ModelMap extends LinkedHashMap<String, Object> {}
public class ExtendedModelMap extends ModelMap implements Model {}
public class BindingAwareModelMap extends ExtendedModelMap {}
```

其实最后还是和view视图信息封装到modelandview中

7、向session域共享数据（建议使用）

```
@RequestMapping("/testSession")
public String testSession(HttpSession session){
    session.setAttribute("testSessionScope", "hello,session");
    return "success";
}
```

得到session中的对象要这样



8、向application域共享数据

```
@RequestMapping("/testApplication")
public String testApplication(HttpServletRequest session){
    ServletContext application = session.getServletContext();
    application.setAttribute("testApplicationscope", "hello,application");
    return "success";
}
```

六、SpringMVC的视图

SpringMVC中的视图是View接口，视图的作用渲染数据，将模型Model中的数据展示给用户

SpringMVC视图的种类很多， 默认有转发视图和重定向视图

当工程引入jstl的依赖， 转发视图会自动转换为JstlView

若使用的视图技术为Thymeleaf，在SpringMVC的配置文件中配置了Thymeleaf的视图解析器， 由此视图解析器解析之后所得到的是ThymeleafView

1、Thymeleaf视图ThymeleafView

当控制器方法中所设置的视图名称没有任何前缀时， 此时的视图名称会被**SpringMVC配置文件中所配置的视图解析器**解析， 视图名称拼接视图前缀和视图后缀所得到的最终路径， 会通过**转发**的方式实现跳转

```
@RequestMapping("/testHello")
public String testHello(){
    return "hello";
}
```

2、转发视图InternalResourceView

SpringMVC中默认的转发视图是InternalResourceView

SpringMVC中创建转发视图的情况：

当控制器方法中所设置的视图名称以"forward:"为前缀时， 创建InternalResourceView视图， 此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析， 而是会将前缀"forward:"去掉， 剩余部分作为最终路径通过转发的方式实现跳转

例如"forward:/", "forward:/employee"

```
@RequestMapping("/testForward")
public String testForward(){
    return "forward:/testHello";
}
```

```
@RequestMapping("/testThymeleafView")
public String testThymeleafView(){
    return "success";
}
```

```
@RequestMapping("/testForward")
public String testForward(){
    return "forward:/testThymeleafView";
}
```

如果想跳过thymeleaf解析器， 之间转发到这个servlet， 可以这样子写

3、重定向视图RedirectView

重定向不能访问WEB-INF下面的页面，因为是浏览器发送的第二次请求，只有转发可以访问WEB-INF下面的页面，但是我们可以重定向到一个转发，然后转发到WEB-INF下的页面

SpringMVC中默认的重定向视图是RedirectView

当控制器方法中所设置的视图名称以"redirect:"为前缀时，创建RedirectView视图，此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀"redirect:"去掉，剩余部分作为最终路径通过**重定向**的方式实现跳转

例如"redirect://"重定向到首页

"redirect:/employee"重定向后的地址为/employee

```
@RequestMapping("/testRedirect")
public String testRedirect(){
    return "redirect:/testHello";
}
```

注：

重定向视图在解析时，会先将redirect:前缀去掉，然后会判断剩余部分是否以/开头，若是则会自动拼接上下文路径

4、视图控制器view-controller

当控制器方法中，仅仅用来实现页面跳转，即只需要设置视图名称时，可以将处理器方法使用view-controller标签进行表示

```
<!--
    path: 设置处理的请求地址
    view-name: 设置请求地址所对应的视图名称
-->
<mvc:view-controller path="/" view-name="index"></mvc:view-controller>
```

注：

当SpringMVC中设置任何一个view-controller时，其他控制器中的请求映射将全部失效，此时需要在SpringMVC的核心配置文件中设置开启mvc注解驱动的标签：

```
<mvc:annotation-driven />
```

The screenshot shows the configuration of a Spring MVC XML configuration file. The code is as follows:

```
30<mvc:view-controller path="/" view-name="index"></mvc:view-controller>
31<!-- 开启MVC的注解驱动-->
32<mvc:annotation-driven />
33
34
35
36</beans>
```

The line `<mvc:annotation-driven />` is highlighted with a red box. The entire configuration section is enclosed in a yellow box.

`<mvc:annotation-driven />`的功能是使其他控制器中的请求映射恢复有效果

The screenshot shows the configuration of a Spring MVC XML configuration file. The code is as follows:

```
25</bean>
26</property>
27</bean>
28</property>
29</bean>
30
31<mvc:view-controller path="/" view-name="index"></mvc:view-controller>
32
33</beans>
```

The line `<mvc:view-controller path="/" view-name="index"></mvc:view-controller>` is highlighted with a red box. A note "可以替代下面的那个" (can replace the one below) is displayed at the bottom right.

```
public class TestController {  
    @RequestMapping("/")  
    public String index(){  
        return "index";  
    }  
}
```

仅仅用来实现页面跳转

七、RESTful

1、RESTful简介

REST: **R**epresentational **S**tate **T**ransfer, 表现层资源状态转移。

a>资源

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是**名词**。一个资源可以由一个或多个URI来标识。URI既是资源的名称，也是资源在Web上的地址。对某个资源感兴趣的客户端应用，可以通过资源的URI与其进行交互。

b>资源的表述（状态）

资源的表述是一段对于资源在某个特定时刻的状态的描述。可以在客户端-服务器端之间转移（交换）。资源的表述可以有多种格式，例如HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

c>状态转移

状态转移说的是：在客户端和服务器端之间转移（transfer）代表资源状态的表述。通过转移和操作资源的表述，来间接实现操作资源的目的。

2、RESTful的实现

具体说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。

它们分别对应四种基本操作：GET用来**获取资源**，POST用来**新建资源**，PUT用来**更新资源**，DELETE用来**删除资源**。

REST风格提倡URL地址使用统一的风格设计，从前到后各个单词使用**斜杠**分开，不使用问号键值对方式携带请求参数，而是将要发送给服务器的数据作为URL地址的一部分，以**保证整体风格的一致性**。

操作	传统方式	REST风格
查询操作	getUserById?id=1	user/1-->get请求方式
保存操作	saveUser	user-->post请求方式
删除操作	deleteUser?id=1	user/1-->delete请求方式
更新操作	updateUser	user-->put请求方式

上面的查询操作和删除操作的意义是不一样的

同理保存和更新操作也都是不一样的

```

    /**
     * 使用RESTful 模拟用户资源的增删改查
     * /user      GET      查询所有用户信息
     * /user/{id}  GET      根据用户id查询用户信息
     * /user      POST     添加用户信息
     * /user/{id} DELETE   删除用户信息
     * /user      PUT      修改用户信息
     */

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public String getAllUser(){
        System.out.println("查询所有用户信息");
        return "success";
    }

    @RequestMapping(value = "/user/{id}", method = RequestMethod.GET)
    public String getUserById(){
        System.out.println("根据id查询用户信息");
        return "success";
    }

    @RequestMapping(value = "/user", method = RequestMethod.POST)
    public String insertUser(String username, String password){
        System.out.println("添加用户信息: "+username+","+password);
        return "success";
    }

    @RequestMapping(value = "/user", method = RequestMethod.PUT)
    public String updateUser(String username, String password){
        System.out.println("修改用户信息: "+username+","+password);
        return "success";
    }
}

```

```
</form>
<form th:action="@{/user}" method="post">
    <input type="hidden" name="_method" value="PUT">
    用户名: <input type="text" name="username"><br>
    密码: <input type="password" name="password"><br>
    <input type="submit" value="修改"><br>
</form>
```

3、HiddenHttpMethodFilter

由于浏览器只支持发送get和post方式的请求，那么该如何发送put和delete请求呢？

SpringMVC 提供了 **HiddenHttpMethodFilter** 帮助我们将 POST 请求转换为 DELETE 或 PUT 请求

HiddenHttpMethodFilter 处理put和delete请求的条件：

a>当前请求的请求方式必须为post

b>当前请求必须传输请求参数_method

满足以上条件，**HiddenHttpMethodFilter** 过滤器就会将当前请求的请求方式转换为请求参数 _method的值，因此请求参数_method的值才是最终的请求方式

在web.xml中注册**HiddenHttpMethodFilter**

```
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

注：

目前为止，SpringMVC中提供了两个过滤器：CharacterEncodingFilter和 HiddenHttpMethodFilter

在web.xml中注册时，必须先注册CharacterEncodingFilter，再注册HiddenHttpMethodFilter

原因：

- 在 CharacterEncodingFilter 中通过 request.setCharacterEncoding(encoding) 方法设置字符集的
- request.setCharacterEncoding(encoding) 方法要求前面不能有任何获取请求参数的操作
- 而 HiddenHttpMethodFilter 恰恰有一个获取请求方式的操作：
- ```
String paramValue = request.getParameter(this.methodParam);
```

设置编码之前不能获取任何参数，所以必须先设置编码（注意过滤器的执行顺序是根据fileter-mapping 定义的顺序决定的）

## 八、RESTful案例

### REST简介

- **REST** (Representational State Transfer) , 表现形式状态转换

- 传统风格资源描述形式

`http://localhost/user/getById?id=1`

`http://localhost/user/saveUser`

- REST风格描述形式

`http://localhost/user/1`

`http://localhost/user`



- 优点：

- 隐藏资源的访问行为，无法通过地址得知对资源是何种操作
  - 书写简化

- 按照REST风格访问资源时使用**行为动作**区分对资源进行了何种操作

|                                         |          |              |
|-----------------------------------------|----------|--------------|
| ■ <code>http://localhost/users</code>   | 查询全部用户信息 | GET (查询)     |
| ■ <code>http://localhost/users/1</code> | 查询指定用户信息 | GET (查询)     |
| ■ <code>http://localhost/users</code>   | 添加用户信息   | POST (新增/保存) |
| ■ <code>http://localhost/users</code>   | 修改用户信息   | PUT (修改/更新)  |
| ■ <code>http://localhost/users/1</code> | 删除用户信息   | DELETE (删除)  |



- 根据REST风格对资源进行访问称为**RESTful**

#### 注意事项

上述行为是约定方式，约定不是规范，可以打破，所以称REST风格，而不是REST规范

描述模块的名称通常使用复数，也就是加s的格式描述，表示此类资源，而非单个资源，例如：users、books、accounts.....

请求步骤：

## ①：设定http请求动作（动词）

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
@ResponseBody
public String save(@RequestBody User user){
 System.out.println("user save..." + user);
 return "{\"module\":\"user save\"}";
}

@RequestMapping(value = "/users" ,method = RequestMethod.PUT)
@ResponseBody
public String update(@RequestBody User user){
 System.out.println("user update..." + user);
 return "{\"module\":\"user update\"}";
}
```

## ②：设定请求参数（路径变量）

```
@RequestMapping(value = "/users/{id}" ,method = RequestMethod.DELETE)
@ResponseBody
public String delete(@PathVariable Integer id){
 System.out.println("user delete..." + id);
 return "{\"module\":\"user delete\"}";
}
```

- 名称: @RequestMapping
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法请求访问路径
- 范例:

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
@ResponseBody
public String save(@RequestBody User user){
 System.out.println("user save..." + user);
 return "{\"module\":\"user save\"}";
}
```

- 属性
  - value (默认) : 请求访问路径
  - method: http请求动作, 标准动作 (GET/POST/PUT/DELETE)
- 名称: @PathVariable
- 类型: **形参注解**
- 位置: SpringMVC控制器方法形参定义前面
- 作用: 绑定路径参数与处理器方法形参间的关系, 要求路径参数名与形参名一一对应
- 范例:

```
@RequestMapping(value = "/users/{id}" ,method = RequestMethod.DELETE)
@ResponseBody
public String delete(@PathVariable Integer id){
 System.out.println("user delete..." + id);
 return "{\"module\":\"user delete\"}";
}
```

区分这些@RequestBody和@RequestParam和@PathVariable

@RequestBody @RequestParam @PathVariable

- 区别

- @RequestParam用于接收url地址传参或表单传参
- @RequestBody用于接收json数据
- @PathVariable用于接收路径参数，使用{参数名称}描述路径参数

- 应用

- 后期开发中，发送请求参数超过1个时，以json格式为主，@RequestBody应用较广
- 如果发送非json格式数据，选用@RequestParam接收请求参数
- 采用RESTful进行开发，当参数数量较少时，例如1个，可以采用@PathVariable接收请求路径变量，通常用于传递id值

## 简化开发型

并且requestbody可以写到类上面，，表示此类的所有的方法都有这个

The screenshot shows a Java code editor with the following code:

```
[@Controller
[@ResponseBody
@RestController
@RequestMapping("/books")
public class BookController {

 @RequestMapping(method = RequestMethod.POST)
 public String save(@RequestBody Book book){
 System.out.println("book save..." + book);
 return "{ 'module':'book save'}";
 }

 // @RequestMapping(value = "/{id}" ,method = RequestMethod.DELETE)
 @DeleteMapping("/{id}")]
```

Annotations highlighted with red boxes:

- [@Controller]
- [@ResponseBody]
- [@RestController]
- [@RequestMapping("/books")]
- [@RequestMapping( method = RequestMethod.POST)]
- [@DeleteMapping("/{id}")]

A red annotation is also present above the code: **这个和上面那两个组合一起等价**.

## 1、准备工作

和传统CRUD一样，实现对员工信息的增删改查。

- 搭建环境
- 准备实体类

```
package com.atguigu.mvc.bean;

public class Employee {
```

```
private Integer id;
private String lastName;

private String email;
//1 male, 0 female
private Integer gender;

public Integer getId() {
 return id;
}

public void setId(Integer id) {
 this.id = id;
}

public String getLastName() {
 return lastName;
}

public void setLastName(String lastName) {
 this.lastName = lastName;
}

public String getEmail() {
 return email;
}

public void setEmail(String email) {
 this.email = email;
}

public Integer getGender() {
 return gender;
}

public void setGender(Integer gender) {
 this.gender = gender;
}

public Employee(Integer id, String lastName, String email, Integer gender) {
 super();
 this.id = id;
 this.lastName = lastName;
 this.email = email;
 this.gender = gender;
}

public Employee() {
}

}
```

- 准备dao模拟数据

```
package com.atguigu.mvc.dao;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import com.atguigu.mvc.bean.Employee;
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDao {

 private static Map<Integer, Employee> employees = null;

 static{
 employees = new HashMap<Integer, Employee>();

 employees.put(1001, new Employee(1001, "E-AA", "aa@163.com", 1));
 employees.put(1002, new Employee(1002, "E-BB", "bb@163.com", 1));
 employees.put(1003, new Employee(1003, "E-CC", "cc@163.com", 0));
 employees.put(1004, new Employee(1004, "E-DD", "dd@163.com", 0));
 employees.put(1005, new Employee(1005, "E-EE", "ee@163.com", 1));
 }

 private static Integer initId = 1006;

 public void save(Employee employee){
 if(employee.getId() == null){
 employee.setId(initId++);
 }
 employees.put(employee.getId(), employee);
 }

 public Collection<Employee> getAll(){
 return employees.values();
 }

 public Employee get(Integer id){
 return employees.get(id);
 }

 public void delete(Integer id){
 employees.remove(id);
 }
}
```

## 2、功能清单

| 功能         | URL 地址      | 请求方式   |
|------------|-------------|--------|
| 访问首页√      | /           | GET    |
| 查询全部数据√    | /employee   | GET    |
| 删除√        | /employee/2 | DELETE |
| 跳转到添加数据页面√ | /toAdd      | GET    |
| 执行保存√      | /employee   | POST   |
| 跳转到更新数据页面√ | /employee/2 | GET    |
| 执行更新√      | /employee   | PUT    |

### 3、具体功能：访问首页

a>配置view-controller

```
<mvc:view-controller path="/" view-name="index"/>
```

b>创建页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8" >
 <title>Title</title>
</head>
<body>
 <h1>首页</h1>
 <a th:href="@{/employee}">访问员工信息
</body>
</html>
```

### 4、具体功能：查询所有员工数据

a>控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.GET)
public String getEmployeeList(Model model){
 Collection<Employee> employeeList = employeeDao.getAll();
 model.addAttribute("employeeList", employeeList);
 return "employee_list";
}
```

b>创建employee\_list.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Employee Info</title>
```

```

<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
</head>
<body>

 <table border="1" cellpadding="0" cellspacing="0" style="text-align: center;" id="dataTable">
 <tr>
 <th colspan="5">Employee Info</th>
 </tr>
 <tr>
 <th>id</th>
 <th>lastName</th>
 <th>email</th>
 <th>gender</th>
 <th>options(<a th:href="@{/toAdd}">add)</th>
 </tr>
 <tr th:each="employee : ${employeeList}">
 <td th:text="${employee.id}"></td>
 <td th:text="${employee.lastName}"></td>
 <td th:text="${employee.email}"></td>
 <td th:text="${employee.gender}"></td>
 <td>
 <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/' + ${employee.id}}">delete
 <a th:href="@{'/employee/' + ${employee.id}}">update
 </td>
 </tr>
 </table>
</body>
</html>

```

## 5、具体功能：删除

a>创建处理delete请求方式的表单

```

<!-- 作用：通过超链接控制表单的提交，将post请求转换为delete请求 -->
<form id="delete_form" method="post">
 <!-- HiddenHttpMethodFilter要求：必须传输_method请求参数，并且值为最终的请求方式 -->
 <input type="hidden" name="_method" value="delete"/>
</form>

```

b>删除超链接绑定点击事件

引入vue.js

```

<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>

```

删除超链接

```

<a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/' + ${employee.id}}">delete

```

通过vue处理点击事件

```

<script type="text/javascript">
 var vue = new Vue({
 el:"#dataTable",
 methods:{
 //event表示当前事件
 deleteEmployee:function (event) {
 //通过id获取表单标签
 var delete_form = document.getElementById("delete_form");
 //将触发事件的超链接的href属性为表单的action属性赋值
 delete_form.action = event.target.href;
 //提交表单
 delete_form.submit();
 //阻止超链接的默认跳转行为
 event.preventDefault();
 }
 }
 });
</script>

```

## c>控制器方法

```

@RequestMapping(value = "/employee/{id}", method = RequestMethod.DELETE)
public String deleteEmployee(@PathVariable("id") Integer id){
 employeeDao.delete(id);
 return "redirect:/employee";
}

```

## 6、具体功能：跳转到添加数据页面

### a>配置view-controller

```

<mvc:view-controller path="/toAdd" view-name="employee_add"></mvc:view-
controller>

```

### b>创建employee\_add.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Add Employee</title>
</head>
<body>

<form th:action="@{/employee}" method="post">
 lastName:<input type="text" name="lastName">

 email:<input type="text" name="email">

 gender:<input type="radio" name="gender" value="1">male
 <input type="radio" name="gender" value="0">female

 <input type="submit" value="add">

</form>

</body>
</html>

```

## 7、具体功能：执行保存

a>控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.POST)
public String addEmployee(Employee employee){
 employeeDao.save(employee);
 return "redirect:/employee";
}
```

## 8、具体功能：跳转到更新数据页面

a>修改超链接

```
<a th:href="@{'/employee/' + ${employee.id}}>update
```

b>控制器方法

```
@RequestMapping(value = "/employee/{id}", method = RequestMethod.GET)
public String getEmployeeById(@PathVariable("id") Integer id, Model model){
 Employee employee = employeeDao.get(id);
 model.addAttribute("employee", employee);
 return "employee_update";
}
```

c>创建employee\_update.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Update Employee</title>
</head>
<body>

<form th:action="@{/employee}" method="post">
 <input type="hidden" name="_method" value="put">
 <input type="hidden" name="id" th:value="${employee.id}">
 lastName:<input type="text" name="lastName" th:value="${employee.lastName}">

 email:<input type="text" name="email" th:value="${employee.email}">

 <!--
 th:field="${employee.gender}" 可用于单选框或复选框的回显
 若单选框的value和employee.gender的值一致，则添加checked="checked"属性
 -->
 gender:<input type="radio" name="gender" value="1"
 th:field="${employee.gender}">male
 <input type="radio" name="gender" value="0"
 th:field="${employee.gender}">female

 <input type="submit" value="update">

</form>

</body>
</html>
```

## 9、具体功能：执行更新

a>控制器方法

```
@RequestMapping(value = "/employee", method = RequestMethod.PUT)
public String updateEmployee(Employee employee){
 employeeDao.save(employee);
 return "redirect:/employee";
}
```

# 八、HttpMessageConverter

HttpMessageConverter，报文信息转换器，将请求报文转换为Java对象，或将Java对象转换为响应报文

HttpMessageConverter提供了两个注解和两个类型：

@RequestBody, @ResponseBody,

RequestEntity, ResponseEntity

### 1、@RequestBody

@RequestBody可以获取请求体，需要在控制器方法设置一个形参，使用@RequestBody进行标识，当前请求的请求体就会为当前注解所标识的形参赋值

```
<form th:action="@{/testRequestBody}" method="post">
 用户名: <input type="text" name="username">

 密码: <input type="password" name="password">

 <input type="submit">
</form>
```

```
@RequestMapping("/testRequestBody")
public String testRequestBody(@RequestBody String requestBody){
 System.out.println("requestBody:"+requestBody);
 return "success";
}
```

输出结果：

requestBody:username=admin&password=123456

### 2、RequestEntity

RequestEntity封装**请求报文**的一种类型，需要在控制器方法的**形参**中设置该类型的形参，当前请求的请求报文就会赋值给该形参，

可以通过getHeaders()获取请求头信息，

通过getBody()获取请求体信息

表单：

```
<form th:action="@{/testRequestEntity}" method="post">
 <input type="text" name="username">
 <input type="text" name="password">
 <input type="submit" value="测试@RequestBody">
</form>
```

```
@RequestMapping("/testRequestEntity")
public String testRequestEntity(RequestEntity<String> requestEntity){
 System.out.println("requestHeader:"+requestEntity.getHeaders());
 System.out.println("requestBody:"+requestEntity.getBody());
 return "success";
}
```

输出结果：

requestHeader:

```
[host:"localhost:8080", connection:"keep-alive", content-length:"27", cache-control:"max-age=0",
sec-ch-ua:"" Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90""", sec-ch-ua-
mobile:"?0", upgrade-insecure-requests:"1", origin:"http://localhost:8080", user-agent:"Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93
Safari/537.36"]
```

requestBody:

username=admin&password=123

### 3、@ResponseBody

```
@RequestMapping("/testResponseBody")
public String testResponseBody(){
 return "success";
}
```

不加@ResponseBody  
是被thymeleaf解析，  
成为一个地址

@ResponseBody用于标识一个控制器方法，可以将该方法的返回值直接作为响应报文的响应体响应到浏览器

```
@RequestMapping("/testResponseBody")
@ResponseBody
public String testResponseBody(){//也可以返回对象，但是需要引入json依赖
 return "success";//不再是视图名称，而是响应体
}
```

这个和下面是同样的效果

如果是利用原生api是这样操作的

```
@RequestMapping("/testResponse")
public void testResponse(HttpServletRequest response) throws IOException {
 response.getWriter().print("hello,response");
}
```



## 4、SpringMVC处理json

json有对象和数组，可以通过最外层来区别

@ResponseBody处理json的步骤：

a>导入jackson的依赖

```
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.12.1</version>
</dependency>
```

b>在SpringMVC的核心配置文件中开启mvc的注解驱动，此时在HandlerAdaptor中会自动装配一个消息转换器： MappingJackson2HttpMessageConverter，可以将响应到浏览器的Java对象转换为Json格式的字符串

```
<mvc:annotation-driven />
```

c>在处理器方法上使用@RequestBody注解进行标识

d>将Java对象直接作为控制器方法的返回值返回，就会自动转换为**Json格式的字符串**

```
@RequestMapping("/testResponseUser")
@ResponseBody
public User testResponseUser() {//自动转换为Json格式的字符串
 return new User(1001,"admin","123456",23,"男");
}
```

浏览器的页面中展示的结果：

```
{"id":1001,"username":"admin","password":"123456","age":23,"sex":"男"}
```

## 5、SpringMVC处理ajax

a>请求超链接：

```
<div id="app">
 <a th:href="@{/testAjax}" @click="testAjax">testAjax

</div>
```

b>通过vue和axios处理点击事件：

```
<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/static/js/axios.min.js}"></script>
<script type="text/javascript">
 var vue = new Vue({
 el:"#app",
 methods:{
 testAjax:function (event) {
 axios({
 method:"post",
 url:event.target.href,
 params:{
 username:"admin",
 password:"123456"
 }
 }).then(function (response) {
 alert(response.data);
 });
 <!--取消超链接的默认行为-->
 event.preventDefault();
 }
 }
);
</script>
```

c>控制器方法：

```
@RequestMapping("/testAjax")
@ResponseBody
public String testAjax(String username, String password){
 System.out.println("username:"+username+",password:"+password);
 return "hello,ajax";
}
```

## 6、@RestController注解

@RestController注解是springMVC提供的一个复合注解，标识在控制器的类上，就相当于为类添加了@Controller注解，并且为其中的每个方法添加了@ResponseBody注解

## 7、 ResponseEntity

ResponseEntity用于控制器方法的返回值类型，该控制器方法的返回值就是响应到浏览器的响应报文

# 九、文件上传和下载

## 1、文件下载

配置好资源的位置，记得要先把maven包再打一下

使用ResponseEntity实现下载文件的功能

其中访问file.html已经配置好了再springMVC.xml中



The screenshot shows an IDE interface with three tabs: 'FileUpAndDownController.java', 'file.html', and 'springMVC.xml'. The 'file.html' tab is active, displaying the following Thymeleaf code:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>测试文件上传和下载</title>
</head>
<body>
 <a th:href="@{/testDown}">
</body>
</html>
```

```
@RequestMapping("/testDown")
public ResponseEntity<byte[]> test ResponseEntity(HttpSession session) throws
IOException {
 //获取ServletContext对象
 ServletContext servletContext = session.getServletContext();
 //获取服务器中文件的真实路径
 String realPath = servletContext.getRealPath("/static/img/1.jpg");
 //创建输入流
 InputStream is = new FileInputStream(realPath);
 //创建字节数组
 byte[] bytes = new byte[is.available()];
 //将流读到字节数组中
 is.read(bytes);
 //创建HttpHeaders对象设置响应头信息
 Multimap<String, String> headers = new HttpHeaders();
 //设置要下载方式以及下载文件的名字
 headers.add("Content-Disposition", "attachment;filename=1.jpg");
 //设置响应状态码
 HttpStatus statusCode = HttpStatus.OK;
```

```

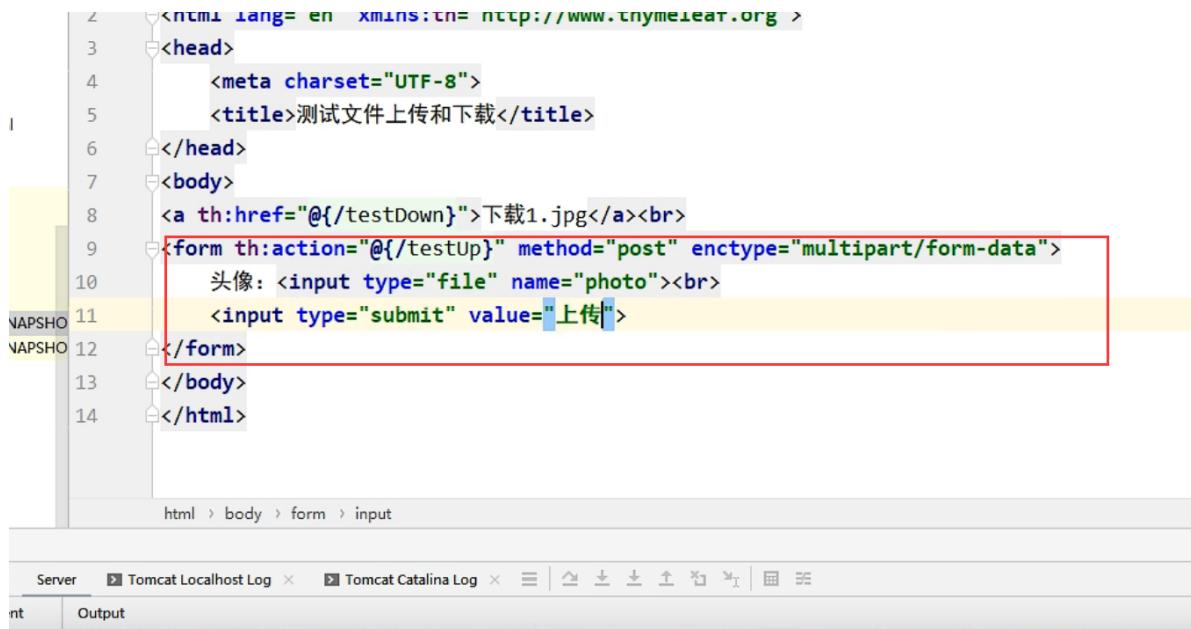
//创建 ResponseEntity对象
ResponseEntity<byte[]> responseEntity = new ResponseEntity<>(bytes, headers,
statusCode);
//关闭输入流
is.close();
return responseEntity;
}

```

## 2、文件上传

文件上传要求form表单的请求方式必须为post，并且添加属性enctype="multipart/form-data"

SpringMVC中将上传的文件封装到MultipartFile对象中，通过此对象可以获取文件相关信息



上传步骤：

a>添加依赖：

```

<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->
<dependency>
 <groupId>commons-fileupload</groupId>
 <artifactId>commons-fileupload</artifactId>
 <version>1.3.1</version>
</dependency>

```

b>在SpringMVC的配置文件中添加配置：

文件上传解析器

```

<!--必须通过文件解析器的解析才能将文件转换为MultipartFile对象-->
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
</bean>

```

这是一个固定值

```

<!--必须通过文件解析器的解析才能将文件转换为MultipartFile对象-->
<bean id="multipartResolver"
 class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
</bean>

```

c>控制器方法：

```
@RequestMapping("/setUp")
public String setUp(MultipartFile photo, HttpSession session) throws
IOException {
 //获取上传的文件的文件名
 String fileName = photo.getOriginalFilename();
 //处理文件重名问题
 String hzName = fileName.substring(fileName.lastIndexOf("."));
 fileName = UUID.randomUUID().toString() + hzName;
 //获取服务器中photo目录的路径
 ServletContext servletContext = session.getServletContext();
 String photoPath = servletContext.getRealPath("photo");
 File file = new File(photoPath);
 if(!file.exists()){
 file.mkdir();
 }
 String finalPath = photoPath + File.separator + fileName;
 //实现上传功能
 photo.transferTo(new File(finalPath));
 return "success";
}
```

上传到了这里



## 十、拦截器

### 1、拦截器的配置



The screenshot shows the configuration of a FirstInterceptor in the springMVC.xml file. A red arrow points from the text "引用他里面的组件" (Reference its internal components) to the <ref bean="firstInterceptor"></ref> tag.

```
55 <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsM
56
57 <!-- 配置拦截器-->
58 <mvc:interceptors>
59 <!--<bean class="com.atguigu.mvc.interceptors.FirstInterceptor"></bean>-->
60 <!--<ref bean="firstInterceptor"></ref>-->
61 </mvc:interceptors>
62
63 </beans></pre>
```

SpringMVC中的拦截器用于拦截控制器方法的执行

SpringMVC中的拦截器需要实现HandlerInterceptor

SpringMVC的拦截器必须在SpringMVC的配置文件中进行配置：

```
<bean class="com.atguigu.interceptor.FirstInterceptor"></bean>
<ref bean="firstInterceptor"></ref>
<!-- 以上两种配置方式都是对DispatcherServlet所处理的所有请求进行拦截 -->
<mvc:interceptor>
 <!--一个斜线星只能表示下面的一层目录-->
 <mvc:mapping path="/**"/>
 <mvc:exclude-mapping path="/testRequestEntity"/>
 <ref bean="firstInterceptor"></ref>
</mvc:interceptor>
<!--
 以上配置方式可以通过ref或bean标签设置拦截器，通过mvc:mapping设置需要拦截的请求，通过
 mvc:exclude-mapping设置需要排除的请求，即不需要拦截的请求
-->
```

## 2、拦截器的三个抽象方法

SpringMVC中的拦截器有三个抽象方法：

preHandle：控制器方法执行之前执行preHandle()，其boolean类型的返回值表示是否拦截或放行，返回true为放行，即调用控制器方法；返回false表示拦截，即不调用控制器方法

postHandle：控制器方法执行之后执行postHandle()

afterCompletion：处理完视图和模型数据，渲染视图完毕之后执行afterCompletion()

## 3、多个拦截器的执行顺序

```

56
57 <!-- 配置拦截器-->
58 <mvc:interceptors>
59 <ref bean="firstInterceptor"></ref>
60 <ref bean="secondInterceptor"></ref>
61 <!--<bean class="com.atguigu.mvc.interceptors.FirstInterceptor"></bean>-->
62 <!--<ref bean="firstInterceptor"></ref>-->

```

只跟这个顺序有关

a>若每个拦截器的preHandle()都返回true

此时多个拦截器的执行顺序和拦截器在SpringMVC的配置文件的配置顺序有关：

preHandle()会按照配置的顺序执行，而postHandle()和afterComplation()会按照配置的反序执行

b>若某个拦截器的preHandle()返回了false

preHandle()返回false和它之前的拦截器的preHandle()都会执行，postHandle()都不执行，返回false的拦截器之前的拦截器的afterComplation()会执行

## 十一、异常处理器

### 1、基于配置的异常处理

SpringMVC提供了一个处理控制器方法执行过程中所出现的异常的接口：HandlerExceptionResolver

HandlerExceptionResolver接口的实现类有：DefaultHandlerExceptionResolver和SimpleMappingExceptionResolver

SpringMVC提供了自定义的异常处理器SimpleMappingExceptionResolver，使用方式：

```

<bean
 class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
 <property name="exceptionMappings">
 <props>
 <prop key="ArithmeticException">error.html</prop>
 </props>
 </property>
</bean>

```

```

<bean
 class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
 <property name="exceptionMappings">
 <props>
 <prop key="ArithmeticException">error.html</prop>
 </props>
 </property>
</bean>

```

```

<!--
properties的键表示处理器方法执行过程中出现的异常
properties的值表示若出现指定异常时，设置一个新的视图名称，跳转到指定页面
-->
<prop key="java.lang.ArithemeticException">error</prop>
</props>
</property>
<!--
exceptionAttribute属性设置一个属性名，将出现的异常信息在请求域中进行共享
-->
<property name="exceptionAttribute" value="ex"></property>
</bean>

```

## 2、基于注解的异常处理



```

//@ControllerAdvice将当前类标识为异常处理的组件
@ControllerAdvice
public class ExceptionController {

 // @ExceptionHandler用于设置所标识方法处理的异常
 @ExceptionHandler(ArithemticException.class)
 //ex表示当前请求处理中出现的异常对象
 public String handleArithemticException(Exception ex, Model model){
 model.addAttribute("ex", ex);
 return "error";
 }

}

```

## 十二、注解配置SpringMVC

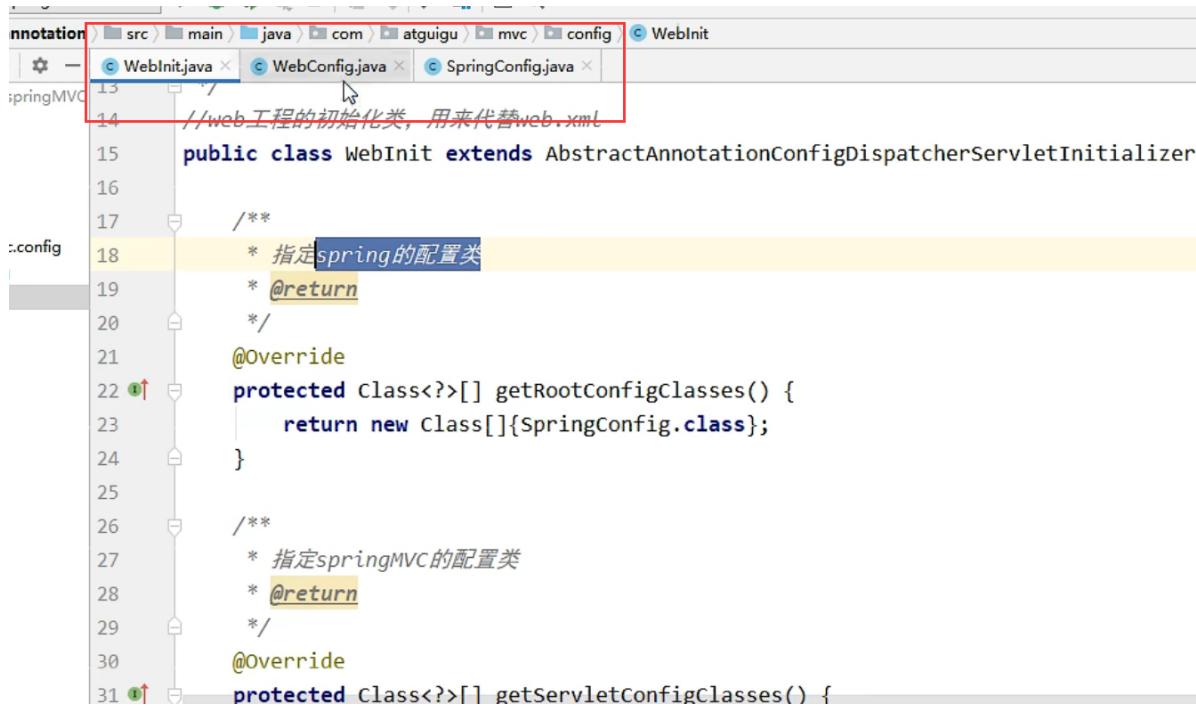
使用配置类和注解代替web.xml和SpringMVC配置文件的功能

### 1、创建初始化类，代替web.xml

在Servlet3.0环境中，容器会在类路径中查找实现javax.servlet.ServletContainerInitializer接口的类，如果找到的话就用它来配置Servlet容器。

Spring提供了这个接口的实现，名为SpringServletContainerInitializer，这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成。Spring3.2引入了一个便利的WebApplicationInitializer基础实现，名为AbstractAnnotationConfigDispatcherServletInitializer，当

我们的类扩展了AbstractAnnotationConfigDispatcherServletInitializer并将其部署到Servlet3.0容器的时候，容器会自动发现它，并用它来配置Servlet上下文。



```
14 //web工程的初始化类,用来代替web.xml
15 public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer
16
17 /**
18 * 指定spring的配置类
19 * @return
20 */
21 @Override
22 protected Class<?>[] getRootConfigClasses() {
23 return new Class[]{SpringConfig.class};
24 }
25
26 /**
27 * 指定springMVC的配置类
28 * @return
29 */
30 @Override
31 protected Class<?>[] getServletConfigClasses() {
```

```
public class WebInit extends
AbstractAnnotationConfigDispatcherServletInitializer {

 /**
 * 指定spring的配置类
 * @return
 */
 @Override
 protected Class<?>[] getRootConfigClasses() {
 return new Class[]{SpringConfig.class};
 }

 /**
 * 指定SpringMVC的配置类
 * @return
 */
 @Override
 protected Class<?>[] getServletConfigClasses() {
 return new Class[]{WebConfig.class};
 }

 /**
 * 指定DispatcherServlet的映射规则,即url-pattern
 * @return
 */
 @Override
 protected String[] getServletMappings() {
 return new String[]{"/"};
 }

 /**
 * 添加过滤器
 * @return
 */
```

```

 */
 @Override
 protected Filter[] getServletFilters() {
 CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();
 encodingFilter.setEncoding("UTF-8");
 encodingFilter.setForceRequestEncoding(true);
 HiddenHttpMethodFilter hiddenHttpMethodFilter = new
 HiddenHttpMethodFilter();
 return new Filter[]{encodingFilter, hiddenHttpMethodFilter};
 }
}

```

## 2、创建SpringConfig配置类，代替spring的配置文件

```

@Configuration
public class SpringConfig {
 //ssm整合之后，spring的配置信息写在此类中
}

```

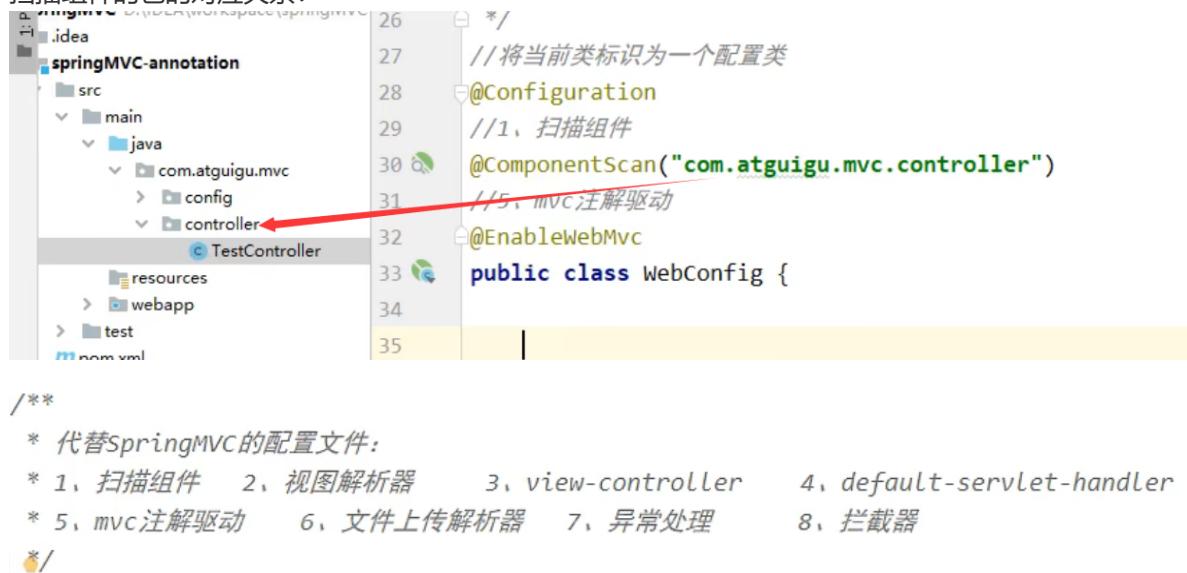
## 3、创建WebConfig配置类，代替SpringMVC的配置文件

@Bean注解的意思是

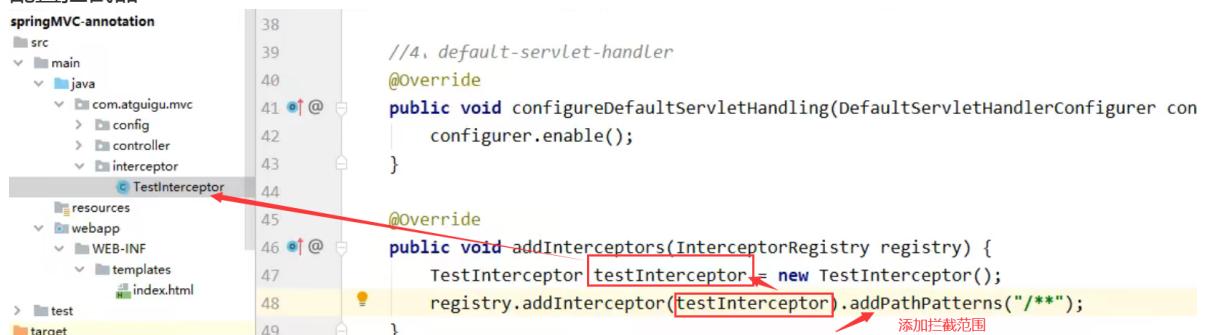
返回值可以作为IOC容器的一个bean

需要注意的是下面的参数的装配是springmvc对之的自动装配

扫描组件的包的对应关系：



配置拦截器



配置view-controller

```

<mvc:view-controller path="/" view-name="index"></mvc:view-controller>
@Override
public void addViewControllers(ViewControllerRegistry registry) {
 registry.addViewController(urlPathOrPattern: "/hello").setViewName("hello");
}

```

## 配置异常处理器

```

<!-- 配置异常处理-->
<!--<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
<property name="exceptionMappings">
<props>
 <prop key="java.lang.ArithemticException">error</prop>
</props>
</property>
<!-- 设置将异常信息共享在请求域中的键-->
<property name="exceptionAttribute" value="ex"></property>
</bean>-->
//7. 异常处理
@Override
public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
 SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
 Properties prop = new Properties();
 prop.setProperty("java.lang.ArithemticException", "error");
 exceptionResolver.setExceptionMappings(prop);
 exceptionResolver.setExceptionAttribute("exception");
 resolvers.add(exceptionResolver);
}

```

```

//将当前类标记为配置类
@Configuration
//1扫描组件
@ComponentScan("com.atguigu.mvc.controller")
//5开启MVC注解驱动
@EnableWebMvc
public class WebConfig implements
 WebMvcConfigurer {

 //4. 使用默认的servlet处理静态资源
 @Override
 public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
 configurer.enable();
 }

 //6配置文件上传解析器
 @Bean
 public CommonsMultipartResolver multipartResolver(){
 return new CommonsMultipartResolver();
 }

 //8配置拦截器
 //intercept 拦截; 拦阻; 截住
}

```

```

@Override
public void addInterceptors(InterceptorRegistry registry) {
 FirstInterceptor firstInterceptor = new FirstInterceptor();
 registry.addInterceptor(firstInterceptor).addPathPatterns("/**");
}

//3配置视图控制

/*@Override
public void addViewControllers(ViewControllerRegistry registry) {
 registry.addViewController("/").setViewName("index");
}*/

//7配置异常映射
/*@Override
public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
 SimpleMappingExceptionResolver exceptionResolver = new
SimpleMappingExceptionResolver();
 Properties prop = new Properties();
 prop.setProperty("java.lang.ArithmeticException", "error");
 //设置异常映射
 exceptionResolver.setExceptionMappings(prop);
 //设置共享异常信息的键
 exceptionResolver.setExceptionAttribute("ex");
 resolvers.add(exceptionResolver);
}*/

//2下面来配置Thymeleaf

//配置生成模板解析器
@Bean
public ITemplateResolver templateResolver() {
 webApplicationContext webApplicationContext =
ContextLoader.getCurrentWebApplicationContext();
 // ServletContextTemplateResolver需要一个ServletContext作为构造参数，可通过
 webApplicationContext 的方法获得
 ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver(
 webApplicationContext.getServletContext());
 templateResolver.setPrefix("/WEB-INF/templates/");
 templateResolver.setSuffix(".html");
 templateResolver.setCharacterEncoding("UTF-8");
 templateResolver.setTemplateMode(TemplateMode.HTML);
 return templateResolver;
}

//生成模板引擎并为模板引擎注入模板解析器
@Bean
public SpringTemplateEngine templateEngine(ITemplateResolver
templateResolver) {
 SpringTemplateEngine templateEngine = new SpringTemplateEngine();

```

```

 templateEngine.setTemplateResolver(templateResolver);
 return templateEngine;
 }

 //生成视图解析器并为解析器注入模板引擎
 @Bean
 public ViewResolver viewResolver(SpringTemplateEngine templateEngine) {
 ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
 viewResolver.setCharacterEncoding("UTF-8");
 viewResolver.setTemplateEngine(templateEngine);
 return viewResolver;
 }

}

```

## 4、测试功能

```

@RequestMapping("/")
public String index(){
 return "index";
}

```

# 十三、SpringMVC执行流程

## 1、SpringMVC常用组件

- DispatcherServlet：前端控制器，不需要工程师开发，由框架提供

作用：统一处理请求和响应，整个流程控制的中心，由它调用其它组件处理用户的请求

- HandlerMapping (ControllerMapping)：处理器映射器（控制器），不需要工程师开发，由框架提供

作用：根据请求的url、method等信息查找Handler，即控制器方法

- Handler (Controller)：处理器，需要工程师开发

作用：在DispatcherServlet的控制下Handler对具体的用户请求进行处理

- HandlerAdapter：处理器适配器，不需要工程师开发，由框架提供

作用：通过HandlerAdapter对处理器（控制器方法）进行执行

- ViewResolver：视图解析器，不需要工程师开发，由框架提供

作用：进行视图解析，得到相应的视图，例如：ThymeleafView、InternalResourceView、RedirectView

- View：视图

作用：将模型数据通过页面展示给用户

## 2、DispatcherServlet初始化过程

DispatcherServlet 本质上是一个 Servlet，所以天然的遵循 Servlet 的生命周期。所以宏观上是 Servlet 生命周期来进行调度。



### a>初始化WebApplicationContext

所在类: org.springframework.web.servlet.FrameworkServlet

```
protected WebApplicationContext initWebApplicationContext() {
 WebApplicationContext rootContext =
 webApplicationContextUtils.getWebApplicationContext(getServletContext());
 WebApplicationContext wac = null;

 if (this.webApplicationContext != null) {
 // A context instance was injected at construction time -> use it
 wac = this.webApplicationContext;
 if (wac instanceof ConfigurableWebApplicationContext) {
 ConfigurableWebApplicationContext cwac =
 (ConfigurableWebApplicationContext) wac;
 if (!cwac.isActive()) {
 // The context has not yet been refreshed -> provide services
 // such as
 // setting the parent context, setting the application context
 // id, etc
 if (cwac.getParent() == null) {
 // The context instance was injected without an explicit
 parent -> set
 // the root application context (if any; may be null) as the
 parent
 cwac.setParent(rootContext);
 }
 configureAndRefreshWebApplicationContext(cwac);
 }
 }
 }
 if (wac == null) {
 // No context instance was injected at construction time -> see if one
 // has been registered in the servlet context. If one exists, it is
 assumed
 // that the parent context (if any) has already been set and that the
 // user has performed any initialization such as setting the context id
 wac = findWebApplicationContext();
 }
 if (wac == null) {
 // No context instance is defined for this servlet -> create a local one
 // 创建WebApplicationContext
 wac = createWebApplicationContext(rootContext);
 }

 if (!this.refreshEventReceived) {
 // Either the context is not a ConfigurableApplicationContext with
 refresh
```

```

 // support or the context injected at construction time had already been
 // refreshed -> trigger initial onRefresh manually here.
 synchronized (this.onRefreshMonitor) {
 // 刷新WebApplicationContext
 onRefresh(wac);
 }
 }

 if (this.publishContext) {
 // Publish the context as a servlet context attribute.
 // 将IOC容器在应用域共享
 String attrName = getServletContextAttributeName();
 getServletContext().setAttribute(attrName, wac);
 }

 return wac;
}

```

## b>创建WebApplicationContext

所在类: org.springframework.web.servlet.FrameworkServlet

```

protected WebApplicationContext createWebApplicationContext(@Nullable
ApplicationContext parent) {
 Class<?> contextClass = getContextClass();
 if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass))
 {
 throw new ApplicationContextException(
 "Fatal initialization error in servlet with name '" +
 getServletName() +
 "' : custom WebApplicationContext class [" + contextClass.getName() +
 "] is not of type ConfigurableWebApplicationContext");
 }
 // 通过反射创建 IOC 容器对象
 ConfigurableWebApplicationContext wac =
 (ConfigurableWebApplicationContext)
 BeanUtils.instantiateClass(contextClass);

 wac.setEnvironment(getEnvironment());
 // 设置父容器
 wac.setParent(parent);
 String configLocation = getContextConfigLocation();
 if (configLocation != null) {
 wac.setConfigLocation(configLocation);
 }
 configureAndRefreshWebApplicationContext(wac);

 return wac;
}

```

## c>DispatcherServlet初始化策略

FrameworkServlet创建WebApplicationContext后，刷新容器，调用onRefresh(wac)，此方法在DispatcherServlet中进行了重写，调用了initStrategies(context)方法，初始化策略，即初始化DispatcherServlet的各个组件

所在类：org.springframework.web.servlet.DispatcherServlet

```
protected void initStrategies(ApplicationContext context) {
 initMultipartResolver(context);
 initLocaleResolver(context);
 initThemeResolver(context);
 initHandlerMappings(context);
 initHandlerAdapters(context);
 initHandlerExceptionResolvers(context);
 initRequestToViewNameTranslator(context);
 initViewResolvers(context);
 initFlashMapManager(context);
}
```

## 3、DispatcherServlet调用组件处理请求

### a>processRequest()

FrameworkServlet重写HttpServlet中的service()和doXxx()，这些方法中调用了processRequest(request, response)

所在类：org.springframework.web.servlet.FrameworkServlet

```
protected final void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {

 long startTime = System.currentTimeMillis();
 Throwable failureCause = null;

 LocaleContext previousLocaleContext =
 LocaleContextHolder.getLocaleContext();
 LocaleContext localeContext = buildLocaleContext(request);

 RequestAttributes previousAttributes =
 RequestContextHolder.getRequestAttributes();
 ServletRequestAttributes requestAttributes = buildRequestAttributes(request,
 response, previousAttributes);

 WebAsyncManager asyncManager = WebUtils.getAsyncManager(request);
 asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(),
 new RequestBindingInterceptor());

 initContextHolders(request, localeContext, requestAttributes);

 try {
 // 执行服务，doService()是一个抽象方法，在DispatcherServlet中进行了重写
 doService(request, response);
 }
```

```

 catch (ServletException | IOException ex) {
 failureCause = ex;
 throw ex;
 }
 catch (Throwable ex) {
 failureCause = ex;
 throw new NestedServletException("Request processing failed", ex);
 }

 finally {
 resetContextHolders(request, previousLocaleContext, previousAttributes);
 if (requestAttributes != null) {
 requestAttributes.requestCompleted();
 }
 logResult(request, response, failureCause, asyncManager);
 publishRequestHandledEvent(request, response, startTime, failureCause);
 }
 }
}

```

## b>doService()

所在类: org.springframework.web.servlet.DispatcherServlet

```

@Override
protected void doservice(HttpServletRequest request, HttpServletResponse
response) throws Exception {
 logRequest(request);

 // Keep a snapshot of the request attributes in case of an include,
 // to be able to restore the original attributes after the include.
 Map<String, Object> attributesSnapshot = null;
 if (webUtils.isIncludeRequest(request)) {
 attributesSnapshot = new HashMap<>();
 Enumeration<?> attrNames = request.getAttributeNames();
 while (attrNames.hasMoreElements()) {
 String attrName = (String) attrNames.nextElement();
 if (this.cleanupAfterInclude ||
 attrName.startsWith(DEFAULT_STRATEGIES_PREFIX)) {
 attributesSnapshot.put(attrName,
request.getAttribute(attrName));
 }
 }
 }

 // Make framework objects available to handlers and view objects.
 request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
getWebApplicationContext());
 request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
 request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
 request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

 if (this.flashMapManager != null) {
 FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request,
response);
 if (inputFlashMap != null) {

```

```

 request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE,
Collections.unmodifiableMap(inputFlashMap));
 }

 request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
 request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE, this.flashMapManager);
}

RequestPath requestPath = null;
if (this.parseRequestPath &&
!ServletRequestPathUtils.hasParsedRequestPath(request)) {
 requestPath = ServletRequestPathUtils.parseAndCache(request);
}

try {
 // 处理请求和响应
 doDispatch(request, response);
}
finally {
 if
(!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
 // Restore the original attribute snapshot, in case of an include.
 if (attributesSnapshot != null) {
 restoreAttributesAfterInclude(request, attributesSnapshot);
 }
 }
 if (requestPath != null) {
 ServletRequestPathUtils.clearParsedRequestPath(request);
 }
}
}
}

```

### c>doDispatch()

所在类: org.springframework.web.servlet.DispatcherServlet

```

protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
 HttpServletRequest processedRequest = request;
 HandlerExecutionChain mappedHandler = null;
 boolean multipartRequestParsed = false;

 WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

 try {
 ModelAndView mv = null;
 Exception dispatchException = null;

 try {
 processedRequest = checkMultipart(request);
 multipartRequestParsed = (processedRequest != request);

 // Determine handler for the current request.
 /*
 * mappedHandler: 调用链
 * 包含handler、interceptorList、interceptorIndex
 */
 }
 }
}

```

```
 handler: 浏览器发送的请求所匹配的控制器方法
 interceptorList: 处理控制器方法的所有拦截器集合
 interceptorIndex: 拦截器索引，控制拦截器afterCompletion()的执行
 */
 mappedHandler = getHandler(processedRequest);
 if (mappedHandler == null) {
 noHandlerFound(processedRequest, response);
 return;
 }

 // Determine handler adapter for the current request.
 // 通过控制器方法创建相应的处理器适配器，调用所对应的控制器方法
 HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

 // Process last-modified header, if supported by the handler.
 String method = request.getMethod();
 boolean isGet = "GET".equals(method);
 if (isGet || "HEAD".equals(method)) {
 long lastModified = ha.getLastModified(request,
 mappedHandler.getHandler());
 if (new ServletWebRequest(request,
 response).checkNotModified(lastModified) && isGet) {
 return;
 }
 }

 // 调用拦截器的preHandle()
 if (!mappedHandler.applyPreHandle(processedRequest, response)) {
 return;
 }

 // Actually invoke the handler.
 // 由处理器适配器调用具体的控制器方法，最终获得 ModelAndView 对象
 mv = ha.handle(processedRequest, response,
 mappedHandler.getHandler());

 if (asyncManager.isConcurrentHandlingStarted()) {
 return;
 }

 applyDefaultViewName(processedRequest, mv);
 // 调用拦截器的postHandle()
 mappedHandler.applyPostHandle(processedRequest, response, mv);
}

catch (Exception ex) {
 dispatchException = ex;
}
catch (Throwable err) {
 // As of 4.3, we're processing Errors thrown from handler methods as
 // well,
 // making them available for @ExceptionHandler methods and other
 // scenarios.
 dispatchException = new NestedServletException("Handler dispatch
 failed", err);
}
```

```

 // 后续处理：处理模型数据和渲染视图
 processDispatchResult(processedRequest, response, mappedHandler, mv,
 dispatchException);
 }
 catch (Exception ex) {
 triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
 }
 catch (Throwable err) {
 triggerAfterCompletion(processedRequest, response, mappedHandler,
 new NestedServletException("Handler processing
failed", err));
 }
 finally {
 if (asyncManager.isConcurrentHandlingStarted()) {
 // Instead of postHandle and afterCompletion
 if (mappedHandler != null) {

 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
 }
 }
 else {
 // Clean up any resources used by a multipart request.
 if (multipartRequestParsed) {
 cleanupMultipart(processedRequest);
 }
 }
 }
}

```

## d>processDispatchResult()

```

private void processDispatchResult(HttpServletRequest request,
HttpServletResponse response,
@Nullable HandlerExecutionChain
mappedHandler, @Nullable ModelAndView mv,
@Nullable Exception exception) throws
Exception {

 boolean errorView = false;

 if (exception != null) {
 if (exception instanceof ModelAndViewDefiningException) {
 logger.debug("ModelAndViewDefiningException encountered",
exception);
 mv = ((ModelAndViewDefiningException) exception).getModelAndView();
 }
 else {
 Object handler = (mappedHandler != null ? mappedHandler.getHandler()
: null);
 mv = processHandlerException(request, response, handler, exception);
 errorView = (mv != null);
 }
 }

 // Did the handler return a view to render?

```

```

if (mv != null && !mv.wasCleared()) {
 // 处理模型数据和渲染视图
 render(mv, request, response);
 if (errorView) {
 webUtils.clearErrorRequestAttributes(request);
 }
}
else {
 if (logger.isTraceEnabled()) {
 logger.trace("No view rendering, null ModelAndView returned.");
 }
}

if (WebAsyncutils.getAsyncManager(request).isConcurrentHandlingStarted()) {
 // Concurrent handling started during a forward
 return;
}

if (mappedHandler != null) {
 // Exception (if any) is already handled..
 // 调用拦截器的afterCompletion()
 mappedHandler.triggerAfterCompletion(request, response, null);
}
}

```

## 4、SpringMVC的执行流程

1. 用户向服务器发送请求，请求被SpringMVC 前端控制器 DispatcherServlet捕获。
2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI），判断请求URI对应的映射：

a) 不存在

- i. 再判断是否配置了mvc:default-servlet-handler
- ii. 如果没配置，则控制台报映射查找不到，客户端展示404错误



- iii. 如果有配置，则访问目标资源（一般为静态资源，如：JS,CSS,HTML），找不到客户端也会展示404错误



b) 存在则执行下面的流程

3. 根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain执行链对象的形式返回。
4. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。
5. 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(...)方法【正向】
6. 提取Request中的模型数据，填充Handler入参，开始执行Handler (Controller)方法，处理请求。  
在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

a) HttpMessageConveter: 将请求消息（如json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

b) 数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

c) 数据格式化：对请求消息进行数据格式化。如将字符串转换成格式化数字或格式化日期等

d) 数据验证：验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

7. Handler执行完成后，向DispatcherServlet 返回一个 ModelAndView对象。

8. 此时将开始执行拦截器的postHandle(...)方法【逆向】。

9. 根据返回的 ModelAndView（此时会判断是否存在异常：如果存在异常，则执行

HandlerExceptionResolver进行异常处理）选择一个适合的ViewResolver进行视图解析，根据 Model和View，来渲染视图。

10. 渲染视图完毕执行拦截器的afterCompletion(...)方法【逆向】。

11. 将渲染结果返回给客户端。

## 注解实现springmvc

### ①：导入SpringMVC坐标与Servlet坐标

```
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>javax.servlet-api</artifactId>
 <version>3.1.0</version>
 <scope>provided</scope>
</dependency>
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 <version>5.2.10.RELEASE</version>
</dependency>
```

#### 注意事项

课程版本基于Spring主版本5.2.10.RELEASE制作

导入spring-webmvc坐标自动依赖spring相关坐标

### ②：初始化SpringMVC环境（同Spring环境）

```
@Configuration
public class SpringMvcConfig {
}
```

③：创建SpringMVC控制器类（等同于Servlet功能）

```
@Controller
public class UserController {
 @RequestMapping("/save")
 public void save(){
 System.out.println("user save ...");
 }
}
```

④：设定SpringMVC加载对应的bean

```
@Configuration
@ComponentScan("com.itheima.controller")
public class SpringMvcConfig {
}
```

⑤：初始化Servlet容器，加载SpringMVC环境，并设置SpringMVC请求拦截的路径

```
public class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {
 protected WebApplicationContext createServletApplicationContext() {
 AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
 ctx.register(SpringMvcConfig.class);
 return ctx;
 }
 protected String[] getServletMappings() {
 return new String[]{"/"};
 }
 protected WebApplicationContext createRootApplicationContext() {
 return null;
 }
}
```

其中

- `AbstractDispatcherServletInitializer`类是SpringMVC提供的快速初始化Web3.0容器的抽象类
- `AbstractDispatcherServletInitializer`提供三个接口方法供用户实现
  - `createServletApplicationContext()`方法，创建Servlet容器时，加载SpringMVC对应的bean并放入`WebApplicationContext`对象范围中，而`WebApplicationContext`的作用范围为`ServletContext`范围，即整个web容器范围

```
protected WebApplicationContext createServletApplicationContext() {
 AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
 ctx.register(SpringMvcConfig.class);
 return ctx;
}
```

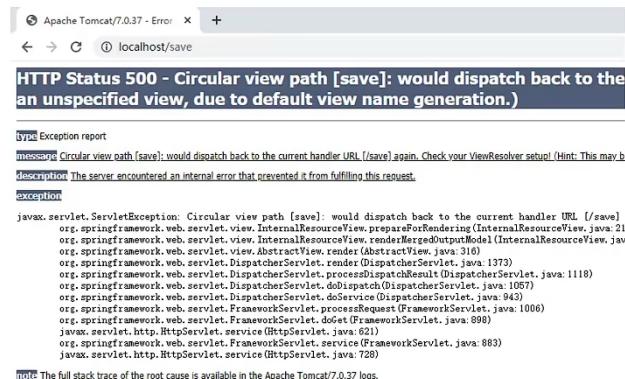
- `getServletMappings()`方法，设定SpringMVC对应的请求映射路径，设置为/表示拦截所有请求，任意请求都将转入到SpringMVC进行处理

```
protected String[] getServletMappings() {
 return new String[]{"/"};
}
```

- `createRootApplicationContext()`方法，如果创建Servlet容器时需要加载非SpringMVC对应的bean，使用当前方法进行，使用方式同`createServletApplicationContext()`

```
protected WebApplicationContext createRootApplicationContext() {
 return null;
}
```

## ⑥：使用浏览器测试请求



### 注意事项

由于未设定请求执行后响应的页面信息，页面显示空白，后台输出代码正常运行

## ⑦：设定请求方法的返回值为字符串类型，并返回自定义json数据

```
@Controller
public class UserController {
 @RequestMapping("/save")
 @ResponseBody
 public String save(){
 System.out.println("user save ...");
 return "{\"info\":\"springmvc\"}";
 }
}
```

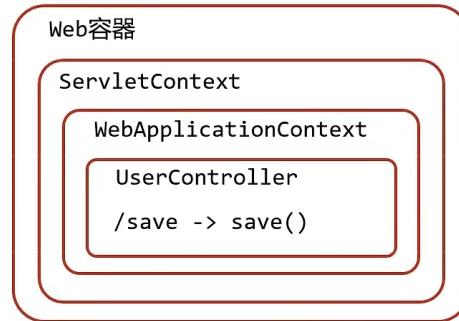
100%

- 名称：`@ResponseBody`
- 类型：**方法注解**
- 位置：SpringMVC控制器方法定义上方
- 作用：设置当前控制器方法响应内容为当前返回值，无需解析
- 范例：

```
@RequestMapping("/save")
@ResponseBody
public String save(){
 System.out.println("user save ...");
 return "{\"info\":\"springmvc\"}";
}
```

# 启动服务器初始化过程

- 启动服务器初始化过程
  - 服务器启动，执行ServletContainersInitConfig类，初始化Web容器
  - 执行createServletApplicationContext方法，创建了WebApplicationContext对象
  - 加载SpringMvcConfig
  - 执行@ComponentScan加载对应的bean
  - 加载UserController，每个@RequestMapping的名称对应一个具体的方法
  - 执行getServletMappings方法，定义所有的请求都通过SpringMVC



## 单次请求过程

- 启动服务器后，执行ServletContainersInitConfig类，初始化Web容器
  - 执行 UserController 类
  - 加载 UserController
  - 执行 @RequestMapping("/save")
  - 执行 @ResponseBody
  - 执行 public String save(){}
  - 加载 System.out.println("user save ...");
  - 返回 {"info": "springmvc"};
  - 执行 }
- 单次请求过程
    - 发送请求 localhost/save
    - web容器发现所有请求都经过SpringMVC，将请求交给SpringMVC处理
    - 解析请求路径 /save
    - 由 /save 匹配执行对应的方法 save()
    - 执行 save()
    - 检测到有@ResponseBody 直接将 save() 方法的返回值作为响应体返回给请求方

## 简化开发版本

下面的是一样的

## 十二、注解配置SpringMVC

使用配置类和注解代替web.xml和SpringMVC配置文件的功能

### 1、创建初始化类，代替web.xml

在Servlet3.0环境中，容器会在类路径中查找实现javax.servlet.ServletContainerInitializer接口的类，如果找到的话就用它来配置Servlet容器。Spring提供了这个接口的实现，名为SpringServletContainerInitializer，这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成。Spring3.2引入了一个便利的WebApplicationInitializer基础实现，名为AbstractAnnotationConfigDispatcherServletInitializer，当我们的类扩展了AbstractAnnotationConfigDispatcherServletInitializer并将其部署到Servlet3.0容器的时候，容器会自动发现它，并用它来配置Servlet上下文。



```
14 //web工程的初始化类，用来代替web.xml
15 public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer
16 {
17 /**
18 * 指定spring的配置类
19 * @return
20 */
21 @Override
22 protected Class<?>[] getRootConfigClasses() {
23 return new Class[]{SpringConfig.class};
24 }
25 }
```

#### ● 简化开发

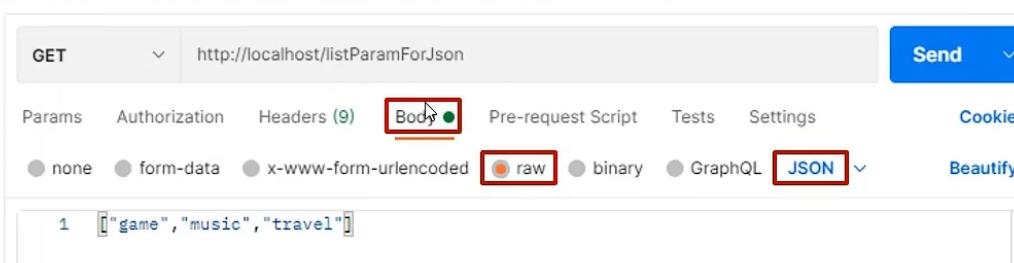
```
public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer{
 protected Class<?>[] getServletConfigClasses() {
 return new Class[]{SpringMvcConfig.class};
 }
 protected String[] getServletMappings() {
 return new String[]{"/"};
 }
 protected Class<?>[] getRootConfigClasses() {
 return new Class[]{SpringConfig.class};
 }
}
```

## 接收请求中的json数据

### ①：添加json数据转换相关坐标

```
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.9.0</version>
</dependency>
```

### ②：设置发送json数据（请求body中添加json数据）



### ③：开启自动转换json数据的支持

```
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig {
}
```

#### 注意事项

@EnableWebMvc注解功能强大，该注解整合了多个功能，此处仅使用其中一部分功能，即json数据进行自动类型转换

### ④：设置接收json数据

```
@RequestMapping("/listParamForJson")
@ResponseBody
public String listParamForJson(@RequestBody List<String> likes){
 System.out.println("list common(json)参数传递 list ==> "+likes);
 return "{ 'module': 'list common for json param'}";
}
```

### 请求参数（传递json数据）

- 名称：@EnableWebMvc
- 类型：**配置类注解**
- 位置：SpringMVC配置类定义上方
- 作用：开启SpringMVC多项辅助功能
- 范例：

```
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig {
}
```

- 名称：@RequestBody
- 类型：**形参注解**
- 位置：SpringMVC控制器方法形参定义前面
- 作用：将请求中请求体所包含的数据传递给请求参数，此注解一个处理器方法只能使用一次
- 范例：

```
@RequestMapping("/listParamForJson")
@ResponseBody
public String listParamForJson(@RequestBody List<String> likes){
 System.out.println("list common(json)参数传递 list ==> "+likes);
 return "{ 'module': 'list common for json param'}";
}
```

## @RequestBody与@RequestParam区别

- 区别

- @RequestParam用于接收url地址传参，表单传参【application/x-www-form-urlencoded】
- @RequestBody用于接收json数据【application/json】

- 应用

- 后期开发中，发送json格式数据为主，@RequestBody应用较广
- 如果发送非json格式数据，选用@RequestParam接收请求参数

## 日期类型参数传递

KEY	VALUE
<input checked="" type="checkbox"/> date	2088/08/08
<input checked="" type="checkbox"/> date1	2088-08-18
<input type="checkbox"/> date2	2088/08/28 8:08:08

```
92
93
94 //日期参数
95 @RequestMapping("/dataParam")
96 @ResponseBody
97 public String dataParam(Date date,
98 @DateTimeFormat(pattern="yyyy-MM-dd") Date date1,
99 @DateTimeFormat(pattern="yyyy/MM/dd HH:mm:ss") Date dat
100
101 System.out.println("参数传递 date ==> "+date);
102 System.out.println("参数传递 date1(yyyy-MM-dd) ==> "+date1);
103 System.out.println("参数传递 date2(yyyy/MM/dd HH:mm:ss) ==> "+date2);
104 return "{\"module\":\"data param\"}";
105 }
```

## 响应POJO对象

```
//响应POJO对象
@RequestMapping("/toJsonPOJO")
@ResponseBody
public User toJsonPOJO(){
 System.out.println("返回json对象数据");
 User user = new User();
 user.setName("itcast");
 user.setAge(15);
 return user;
}
```

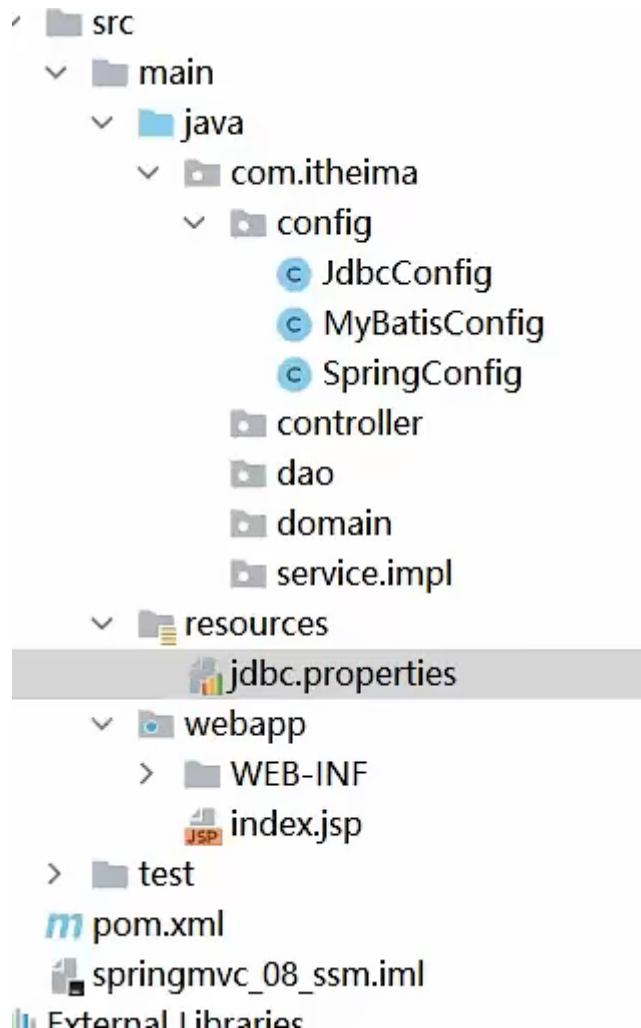
http://localhost/toJsonPOJO

## ResponseBody

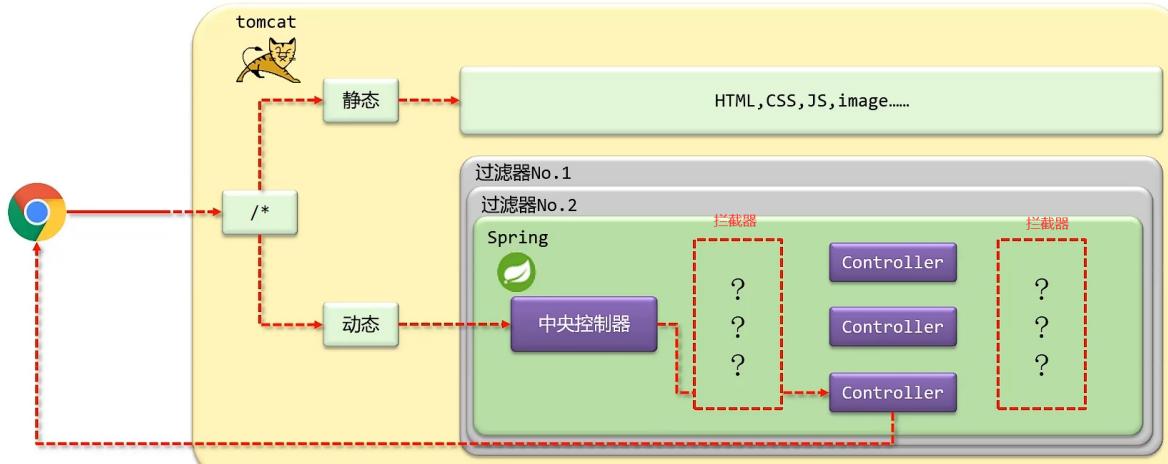
- 名称: @ResponseBody
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法响应内容为当前返回值, 无需解析
- 范例:

```
@RequestMapping("/save")
@ResponseBody
public String save(){
 System.out.println("save...");
 return "{\"info\":\"springmvc\"}";
}
```

## SSM整合



## 拦截器



## 拦截器与过滤器区别

- 归属不同：Filter属于Servlet技术，Interceptor属于SpringMVC技术
- 拦截内容不同：Filter对所有访问进行增强，Interceptor仅针对SpringMVC的访问进行增强



## 步骤

①：声明拦截器的bean，并实现HandlerInterceptor接口（注意：扫描加载bean）

```
@Component
public class ProjectInterceptor implements HandlerInterceptor {
 public boolean preHandle(..) throws Exception {
 System.out.println("preHandle...");
 return true;
 }
 public void postHandle(..) throws Exception {
 System.out.println("postHandle...");
 }
 public void afterCompletion(..) throws Exception {
 System.out.println("afterCompletion...");
 }
}
```

②：定义配置类，继承WebMvcConfigurationSupport，实现addInterceptors方法（注意：扫描加载配置）

```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 ...
 }
}
```

③：添加拦截器并设定拦截的访问路径，路径可以通过可变参数设置多个

```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
 @Autowired
 private ProjectInterceptor projectInterceptor;

 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 registry.addInterceptor(projectInterceptor).addPathPatterns("/books");
 }
}
```