SpringMVC

1、SpringMVC简介

1.1、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。

1.2、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 Strust、WebWork、Strust2 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

1.3、SpringMVC的特点

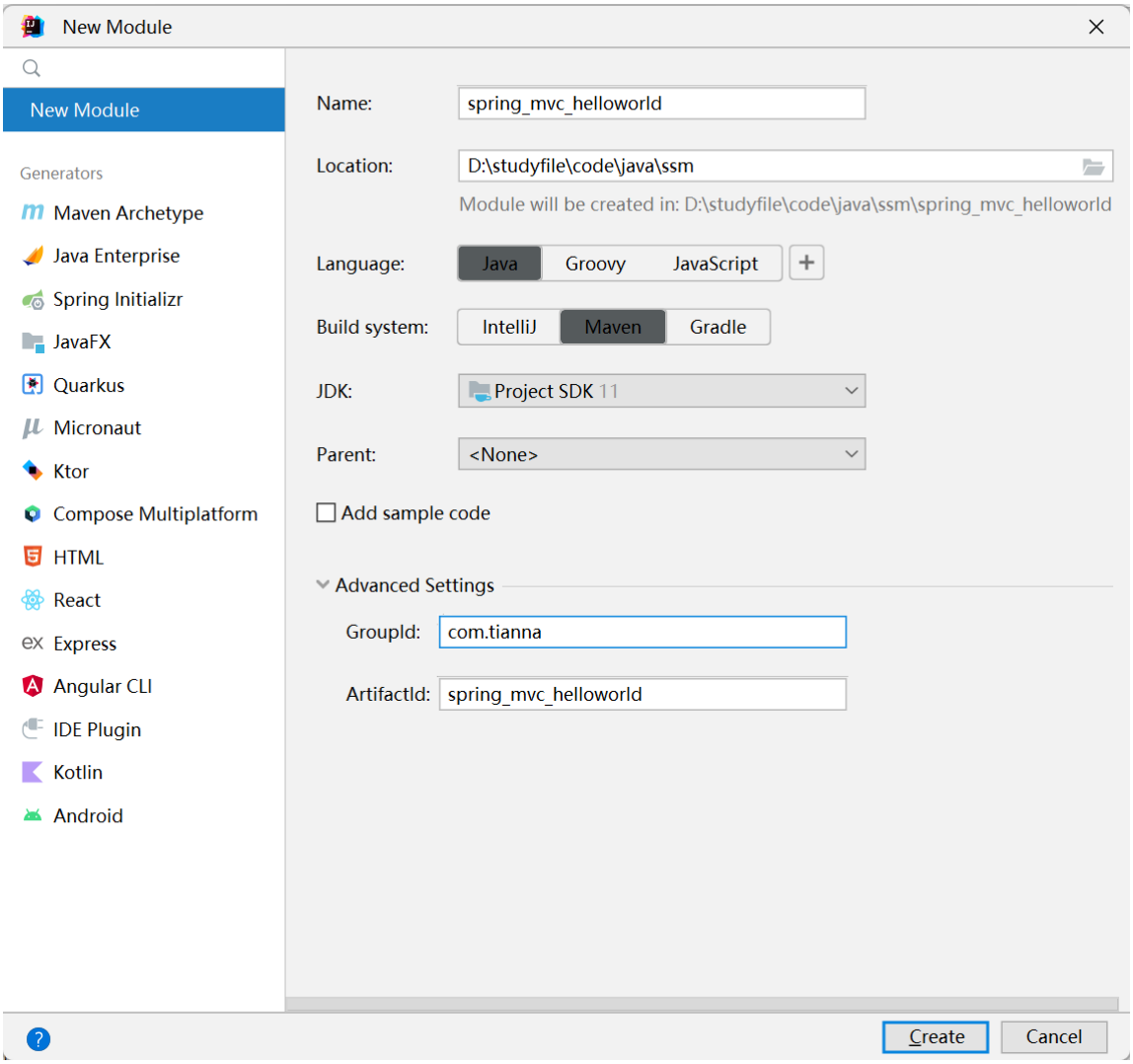
- **Spring 家族原生产品**，与 IOC 容器等基础设施无缝对接。
- **基于原生的Servlet**，通过了功能强大的前端控制器**DispatcherServlet**，对请求和响应进行统一处理。
- 表述层各细分领域需要解决的问题全**方位覆盖，提供全面解决方案。**
- 代码清新简洁，大幅度提升开发效率。
- 内部组件化程度高，可插拔式组件**即插即用**，想要什么功能配置相应组件即可。
- 性能卓著，尤其适合现代大型、超大型互联网项目要求。

2、入门案例

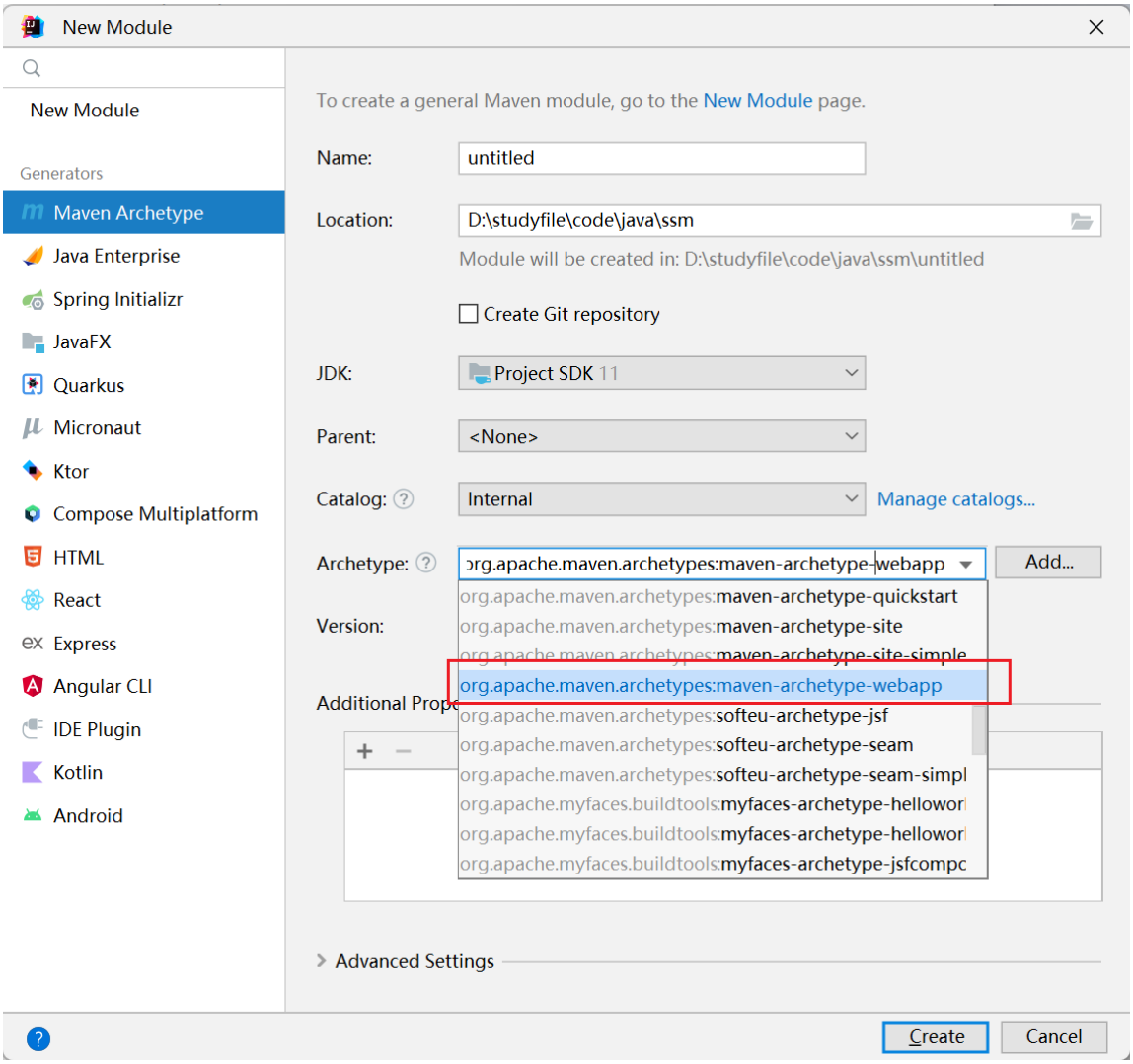
2.1、创建maven工程

1、创建一个maven工程

可以创建一个普通的maven工程，需要按照第二步手动添加web模块。



也可以直接创建一个带有web模块的maven工程，就不需要执行步骤二。



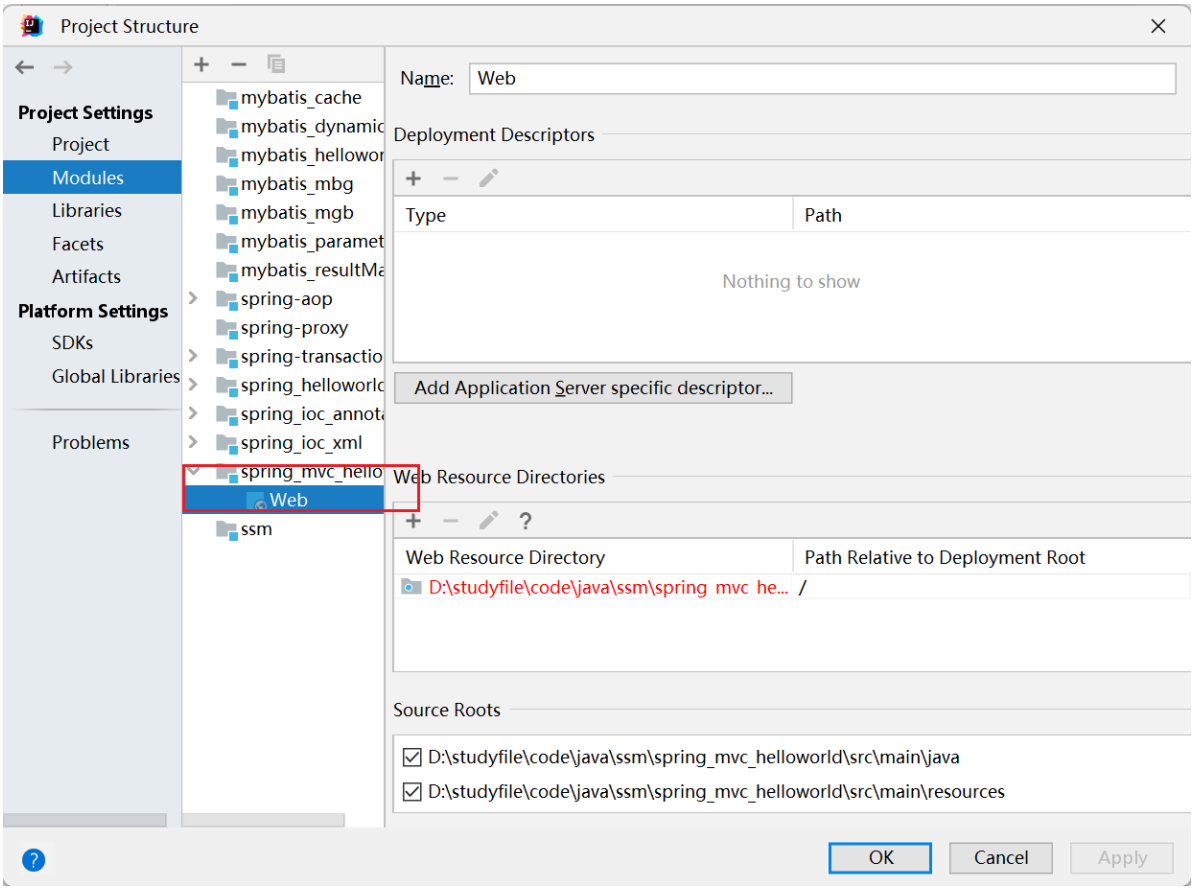
2、添加web模块

在普通的maven工程中没有web模块，因此需要自行添加。

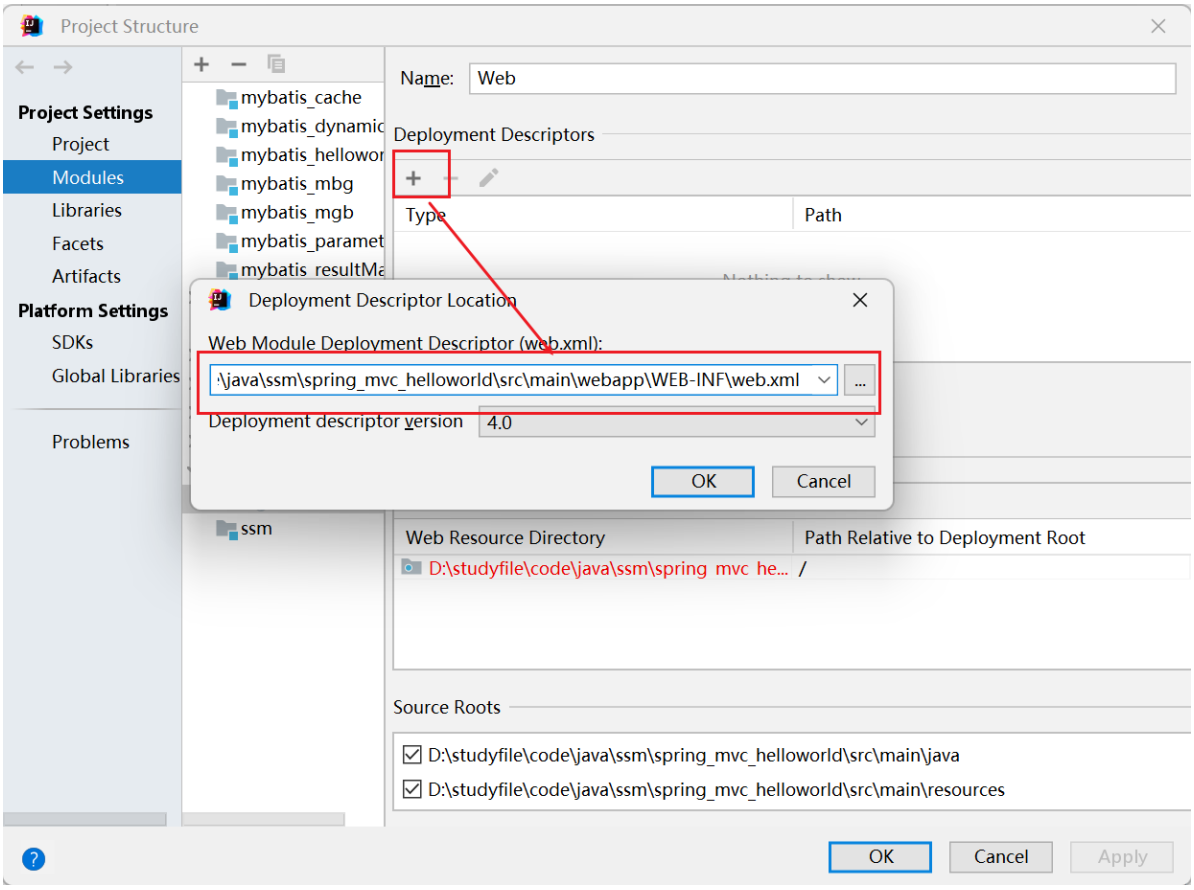
在pom.xml文件中将打包方式设为war包，然后导入更新：

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

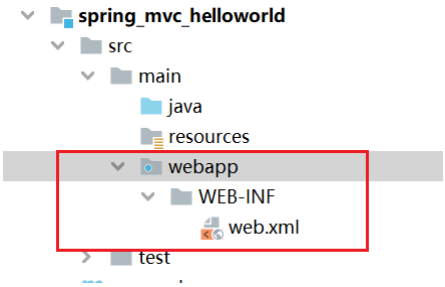
在项目结构中就可以看到web模块。



然后设置web.xml的路径，其给出的路径是不正确的，需要进行修改，如下图：



然后再src\main下会出现web资源文件以及web.xml文件

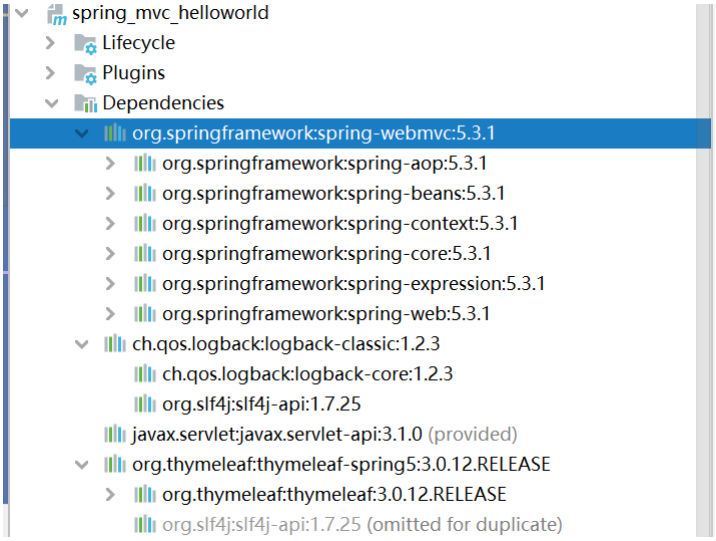


3、导入依赖

在pom.xml文件中导入如下的依赖环境：

```
<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>
```

> 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。



2.2、配置web.xml

注册SpringMVC的前端控制器DispatcherServlet

默认配置

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>属性值-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。

```
<?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/javaee/web-app_4_0.xsd"
         version="4.0">
    <!--
        配置Spring MVC的前端控制器DispatcherServlet

        url-pattern中/和/*的区别
        /：匹配浏览器向服务器发送的所有请求（不包括.jsp）
        /*：匹配浏览器向服务器发送的所有请求（包括.jsp）

    -->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

其中 <url-pattern>中/和/*的设置有如下区别：

- /：匹配浏览器向服务器发送的所有请求（不包括.jsp）即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。
- /*：匹配浏览器向服务器发送的所有请求（包括.jsp）。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

2.3、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 {
}
```

2.4、创建SpringMVC的配置文件

SpringMVC的配置文件默认位置和名称：

- 位置：WEB-INF下
- 名称：<servlet-name>的值-servlet.xml

> 注意：这里只是演示一下默认的情况，以后不会将配置文件放在WEB-INF下，一般放在resources下。

下面为配置文件最基本的配置：

- 扫描控制层组件
- 配置Thymeleaf视图解析器。其中的 视图前缀+逻辑视图+视图后缀 组成完整的物理视图。例如访问的路径为`/WEB-INF/templates/index.html`时，只需要设置逻辑视图`index`。视图解析器会自动将其解析成物理视图`/WEB-INF/templates/index.html`。

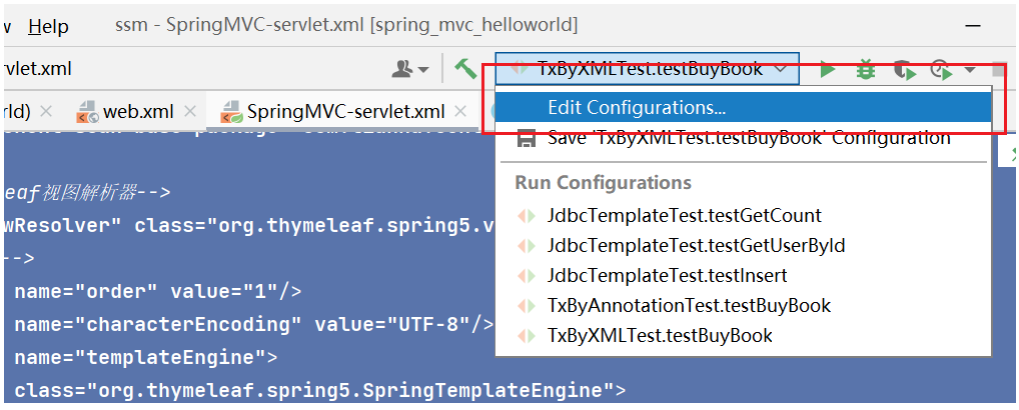
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">
    <!--扫描控制层组件-->
    <context:component-scan base-package="com.tianna.controller"/>

    <!--配置Thymeleaf视图解析器-->
```

```
    <bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
        <!--优先级-->
        <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.SpringResourceTemplateResolve
r">
                        <!-- 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" />
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

</beans>
```

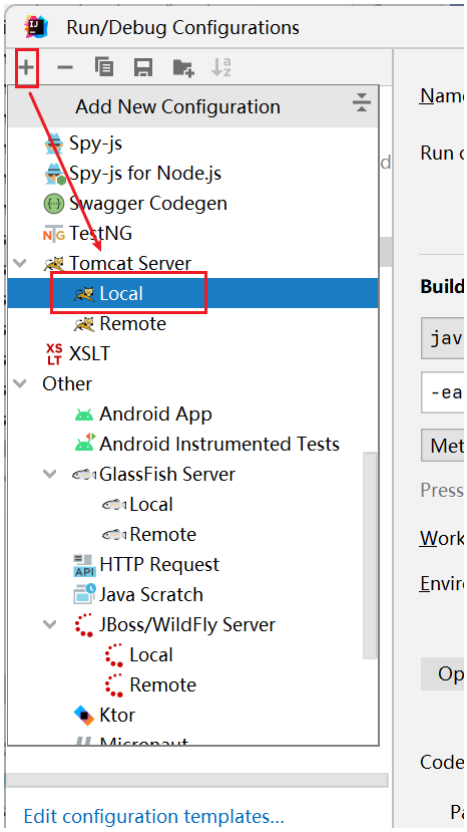
2.5、测试HelloWorld

2.5.1、配置tomcat

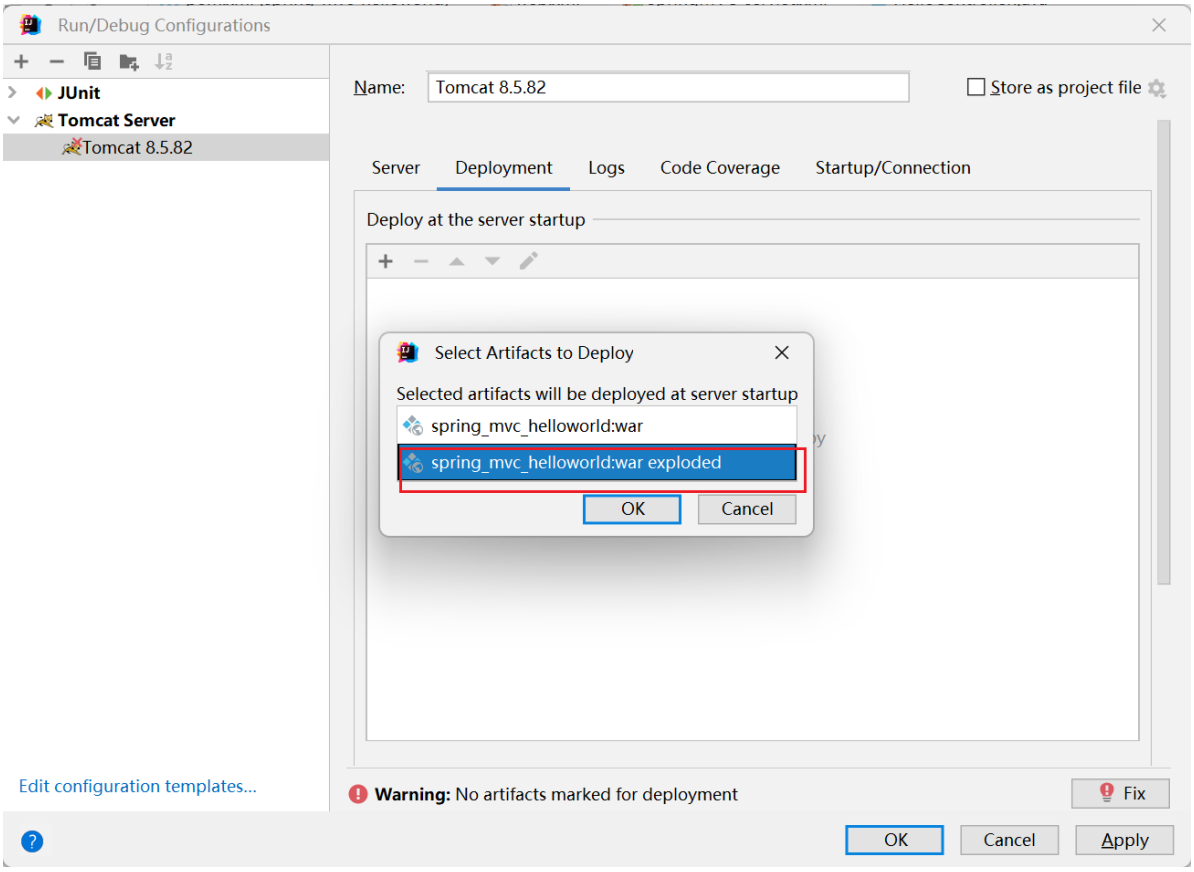
1、按照如下操作，打开Run/Debug Configuration界面。



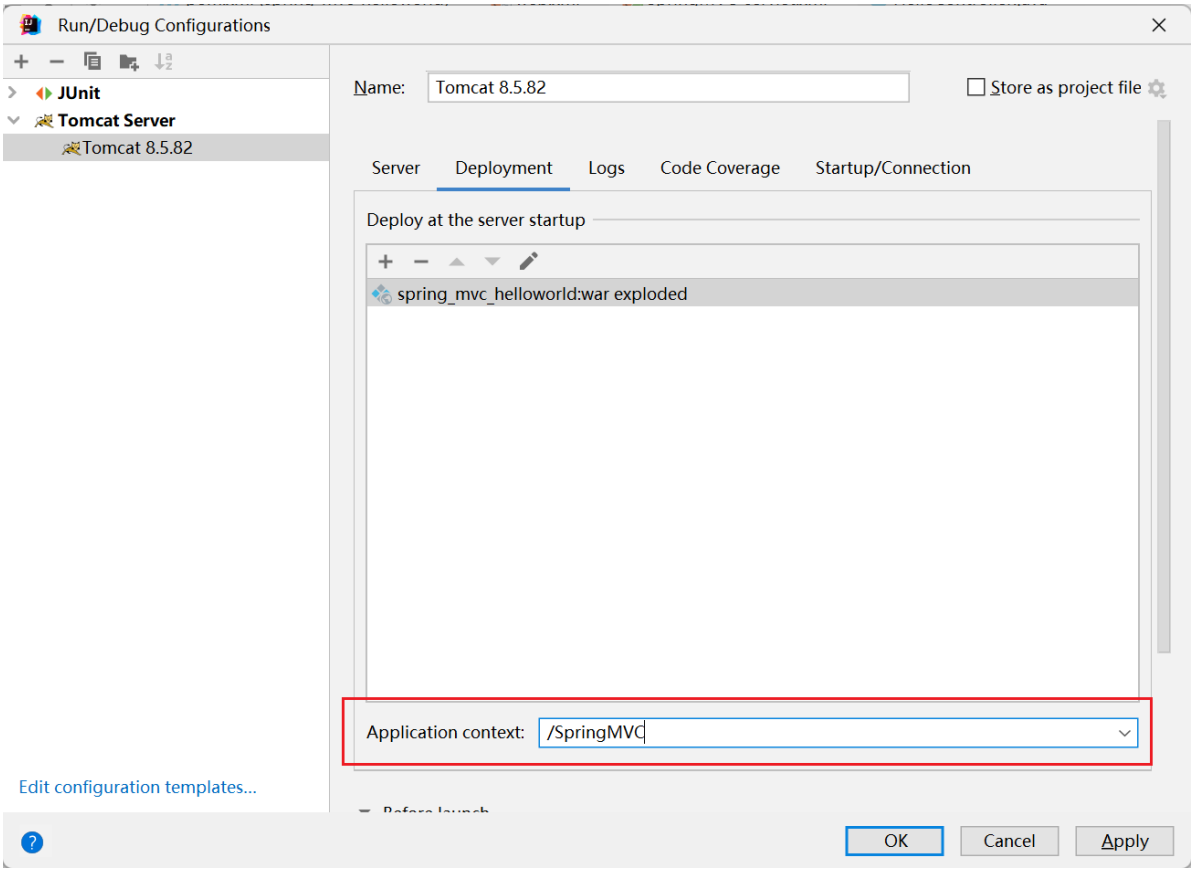
2、点击+号，选择tomcat Server下的Local。



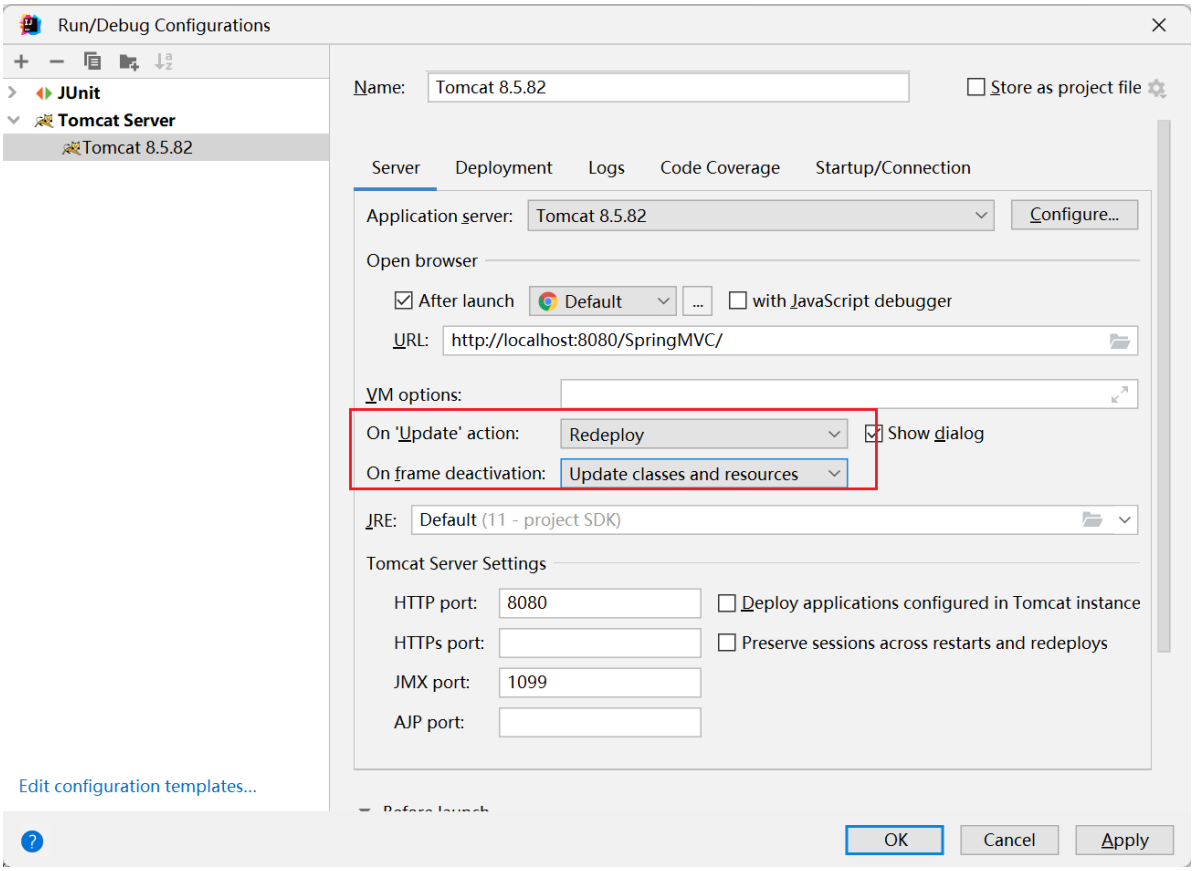
3、将项目部署到tomcat中，选择Deployment一栏，点击加号选择artifacts，然后选择以exploded结尾的选项。



4、上下文路径比较长，这里将其修改短一点，通过该路径就可以访问到服务器中的工程。



5、在Server一栏中，将On 'Update' action选择为Redeploy(重新部署)。将On frame deactivation选择为Update classes and resources（即当IDEA窗口失去焦点时，执行更新类和资源的操作）。



2.5.2、创建页面

在src/main/webapp/WEB-INF目录下创建templates文件夹，并在该文件夹下创建index.html页面。

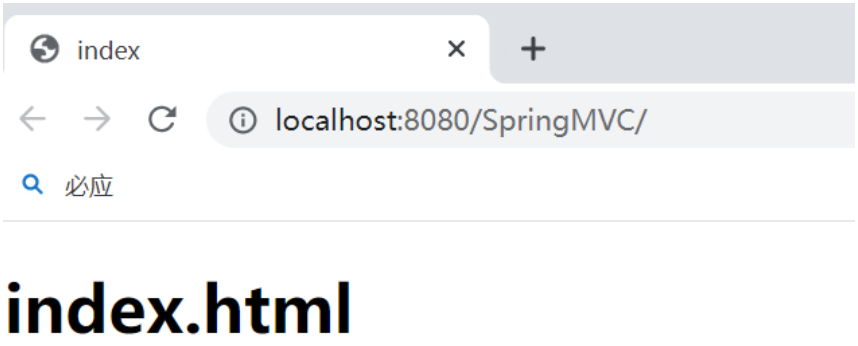
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>index</title>
</head>
<body>

<h1>index.html</h1>

</body>
</html>
```

2.5.3、实现对首页的访问

在请求控制器中创建处理请求的方法。让浏览器发送指定的请求，在处理请求的方法中会执行相应的操作，在这里我们返回一个逻辑视图。这个逻辑视图会被配置文件的视图解析器解析，然后把逻辑视图加上前缀和后缀成为物理视图，然后被Thymeleaf进行渲染得到页面。

```
@Controller
public class HelloController {
    // @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
    //即localhost:8080/springMVC/
    @RequestMapping("/")
    public String protal(){
        //将逻辑视图返回
        return "index";
    }
}
```

在浏览器中访问连接·`http://localhost:8080/SpringMVC/`可得到页面如下



2.5.4、通过超链接跳转到指定页面

因为需要使用thymeleaf的语法，因此要引入thymeleaf的命名空间：

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

在主页index.html中设置超链接，其内容如下：

其中hymeleaf的超链接会自动加上上下文路径`http://localhost:8080/SpringMVC/hello`（会被视图解析器进行解析），可成功访问；而普通的超链接不会加上上下文路径`http://localhost:8080/hello`，则会访问失败，要使可以成功访问，可以将内容修改为`/SpringMVC//hello`，即加上上下文路径。

```
<!DOCTYPE html>
<html lang="en" xmlns:th = "http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>index</title>
</head>
<body>
<h1>index.html</h1>
<!--会自动加上上下文路径 http://localhost:8080/SpringMVC/hello-->
<a th:href = "@{/hello}">测试SpringMVC</a>
<!--不加上下文路径 http://localhost:8080/hello  因此找不到路径-->
<a href = "/hello">测试绝对路径</a>

</body>
</html>
```

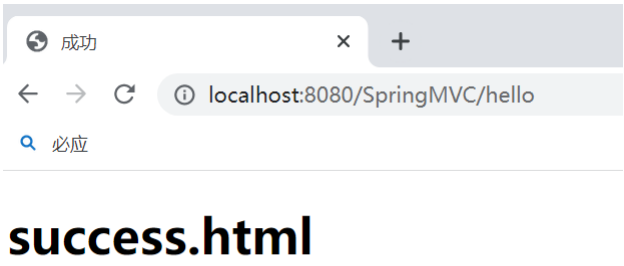
在请求控制器中创建处理请求的方法：

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

然后在templates下创建success.html页面

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>成功</title>
</head>
<body>
<h1>success.html</h1>

</body>
</html>
```

点击测试SpringMVC连接访问的地址为`http://localhost:8080/SpringMVC/hello`可成功访问，如下：



点击测试绝对路径连接访问的路径为`http://localhost:8080/SpringMVC/`，找不到该路径。

2.6、总结

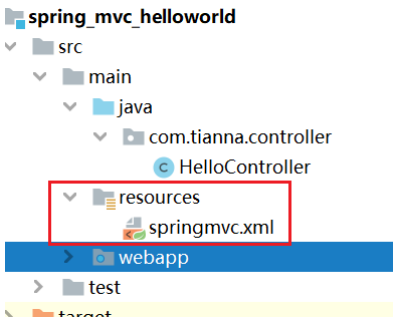
浏览器发送请求，若请求地址符合前端控制器的url-pattern，该请求就会被前端控制器DispatcherServlet处理。前端控制器会读取SpringMVC的核心配置文件，通过扫描组件找到控制器，将请求地址和控制器中@RequestMapping注解的value属性值进行匹配，若匹配成功，该注解所标识的控制器方法就是处理请求的方法。处理请求的方法需要返回一个字符串类型的视图名称，该视图名称会被视图解析器解析，加上前缀和后缀组成视图的路径，通过Thymeleaf对视图进行渲染，最终转发到视图所对应页面。

2.7、扩展

2.7.1、设置SpringMVC配置文件的位置和名称

一般将SpringMVC配置文件放到类路径下，即resources文件夹下，因此可以通过web.xml中servlet标签下的init-param标签进行修改SpringMVC配置文件的位置和名称。

然后既可以将SpringMVC配置文件移动到resources目录下，起名字可以根据init-param标签下param-value下的值修改，名字前需要加上`classpath:`，代表类路径。



2.7.2、修改前端控制器DispatcherServlet的初始化时间

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。可通过web.xml中servlet标签下load-on-startup标签设置SpringMVC前端控制器DispatcherServlet的初始化时间。

2.7.3、修改后的web.xml

```
<?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/javaee/web-app_4_0.xsd"
         version="4.0">
    <!--
        配置Spring MVC的前端控制器DispatcherServlet

        SpringMVC的配置文件默认位置和名称：
        位置：WEB-INF下
        名称：<servlet-name>的值-servlet.xml,当下配置下的配置文件名为
SpringMVC-servlet.xml

        url-pattern中/和/*的区别
        /：匹配浏览器向服务器发送的所有请求（不包括.jsp）
        /*：匹配浏览器向服务器发送的所有请求（包括.jsp）

    -->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--设置SpringMVC配置文件的位置和名称-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <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>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

3、@RequestMapping注解

3.1、@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。

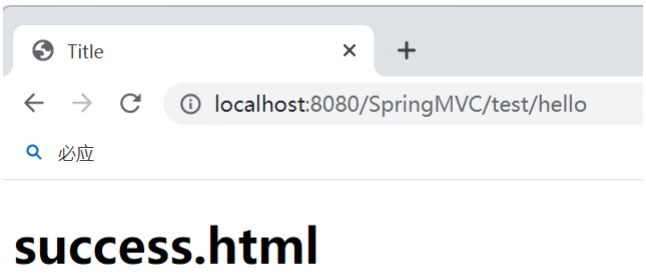
3.2、@RequestMapping注解的位置

@RequestMapping注解标识的位置：

- @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。

```
@Controller
@RequestMapping("/test")
public class TestRequestMappingController {
    //此时控制器方法所匹配的请求路径为/test/hello
    @RequestMapping("/hello")
    public String hello(){
        return "success";
    }
}
```

上面注解分别标识了类和方法，因此器请求路径为类和方法上@RequestMapping注解value值的组合，即/test/hello



3.3、@RequestMapping注解的value属性

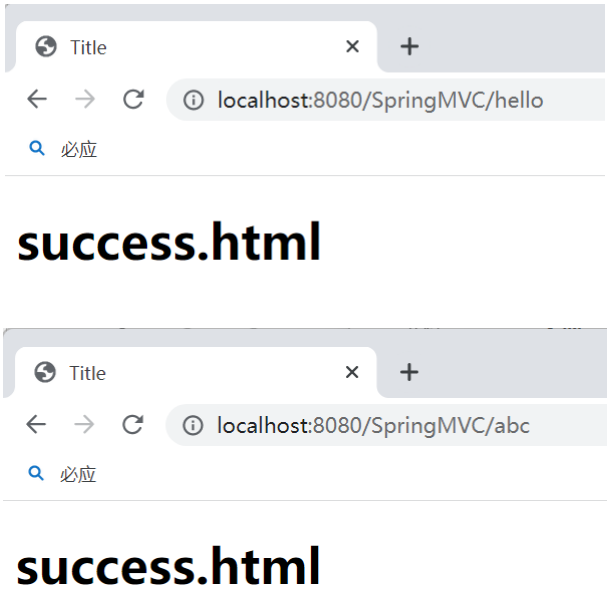
@RequestMapping注解的value属性通过请求的请求地址匹配请求映射。

value属性是数组类型，即当前浏览器所发送请求的请求路径匹配value属性中的任何一个值，则当前请求就会被注解所标识的方法进行处理。

@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。

```
@Controller
public class TestRequestMappingController {
    @RequestMapping(value = {"/hello","/abc"})
    public String hello(){
        return "success";
    }
}
```

上面代码中通过访问`/hello`和`/abc`都可以成功访问。



3.4、@RequestMapping注解的method属性

@RequestMapping注解method属性通过请求的请求方式匹配请求。

@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。

使用方式如下：

```
@Controller
public class TestRequestMappingController {
    @RequestMapping(
            value = {"/hello","/abc"},
            method = {RequestMethod.POST,RequestMethod.GET}
    )
    public String hello(){
        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。

3.5、@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
- "param!=value"：表示当前所匹配请求的请求参数中可以不携带param参数，若携带值一定不能是value。

在下面的代码中，请求参数必须携带username和age，其中age必须为20，请求参数必须不能有password，其中参数gener也以后也可以没有，如果有的话，其值必须不能为女。一下两个请求链接可以成功访问。

`http://localhost:8080/SpringMVC/hello?username=admin&age=20&gener=男` 或
`http://localhost:8080/SpringMVC/hello?username=admin&age=20`

```
@Controller
public class TestRequestMappingController {
    @RequestMapping(
            value = {"/hello","/abc"},
            method = {RequestMethod.POST,RequestMethod.GET},
            params = {"username","!password","age=20","gener!=女"}
    )
    public String hello(){
        return "success";
    }
}
```

3.6、@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可以不带header请求头信息，若要携带值一定不能为value。

使用方式如下，即所带的请求头信息需要有`referer`。

```
@Controller
public class TestRequestMappingController {
    @RequestMapping(
            value = {"/hello","/abc"},
            headers = {"referer"}
    )
    public String hello(){
        return "success";
    }

}
```

3.7、SpringMVC支持ant风格的路径

在@RequestMapping注解的value属性值中设置一些特殊字符：

- ？：表示任意的单个字符（不包括?）。
- *：表示任意的0个或多个字符（不包括?和/）。
- **：表示任意层数的任意目录。使用时，是能使用/**/xx的方式。

使用方式如下：

```
@Controller
public class TestRequestMappingController {

    @RequestMapping("/**/test/ant")
    public String testAnt(){
        return "success";
    }
}
```

3.8、SpringMVC支持路径中的占位符（重点）

原始方式：/deleteUser?id=1

rest方式：/user/delete/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。

请求路径为：

```
<a th:href="@{/test/rest/admin/1}">测试@RequestMapping注解的value属性中的占位符</a>
```

控制器方法如下：

```
@Controller
public class TestRequestMappingController {
    @RequestMapping("test/rest/{username}/{id}")
    public String testRest(@PathVariable("id") Integer
id,@PathVariable("username") String username){
        System.out.println("id: " + id + ", username: " + username);
        return "success";
    }
}
```

4、SpringMVC获取请求参数

4.1、通过ServletAPI获取

只需要在控制器方法的形参位置设置HttpServletRequest类型的形参，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。就可以在控制器方法中使用HttpServletRequest对象获取请求参数。

以获取请求中的username和password为例：

```
@Controller
public class TestParamController {
    @RequestMapping("/param/servletAPI")
    public String getParamByServletAPI(HttpServletRequest request){
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        System.out.println("username:" + username + ",password:" +
password);
        return "success";
    }
}
```

可以获取form表单中的参数：

```
<form th:action="@{/param/servletAPI}" method="post">
    用户名:<input type="text" name="username"><br>
    密码:<input type="password" name="password"><br>
    <input type="submit" value="登录">
</form>
```

或者直接获取在请求地址中的参数：
`http://localhost:8080/SpringMVC/param/servletAPI?username=admin&password=123456`。

4.2、通过控制器方法的形参获取请求参数

只需要在控制器方法的形参位置，设置一个形参，形参的名字和请求参数的名字一致即可。

> 注意：使用这种方式一定要保证形参的名字和请求参数的名字一致。

例如请求地址如下，请求的参数为username和password。

```
localhost:8080/SpringMVC/param?username=admin&password=123456
```

控制器方法如下：

```
@Controller
public class TestParamController {
    @RequestMapping("/param")
    public String getParam(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。

4.3、@RequestParam

上一小节介绍通过控制器方法的形参获取请求参数，但是这种方法必须要求形参的名字和请求参数的名字一致。当不一致时可以使用@RequestParam。

@RequestParam将请求参数和控制器方法的形参绑定。

@RequestParam注解的三个属性：value、required、defaultValue：

- value:设置和形参绑定的请求参数的名字。
- required:设置是否必须传输value所对应的请求参数，默认值为true，表示value所对应的请求参数必须传输，否则页面报错 400 - Required String parameter 'userName' is not present。若设置为false，则表示value所对应的请求参数不是必须传输，若不传输，则形参值为null。
- defaultValue:设置当没有传输value所对应的请求参数时，为形参设置的默认值，此时和required属性值无关。

当请求地址中的参数为userName和password时，其中userName与形参的username不匹配，就可以使用@RequestParam的value属性与之进行匹配。当required为true时，若参数userName不传值，则会报错；但这是设置了defaultValue的值为hello，因此此时不传值不会报错，该参数的值默认成为hello。

```
@Controller
public class TestParamController {
    @RequestMapping("/param")
    public String getParam(
            @RequestParam(value = "userName",required = true,defaultValue =
"hello") String username,
            String password){
        System.out.println("username:" + username + ",password:" +
password);
        return "success";
    }
}
```

4.4、@RequestHeader

@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。

下面演示获取请求头中的referer字段的信息。其中包含了用户在访问当前资源之前的位置。

```
@Controller
public class TestParamController {
    @RequestMapping("/param")
    public String getParam(
            //获取请求头中的referer的信息
            @RequestHeader("referer") String referer
    ){
        System.out.println("referer:" + referer);
        return "success";
    }
}
```

输入如下：

```
referer:http://localhost:8080/SpringMVC/
```

4.5、@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。

以获取cookie中的JSESSIONID为例。在创建会话时，即调用request.getSession()的时候会创建JSESSIONID。

```
@Controller
public class TestParamController {
    @RequestMapping("/param")
    public String getParam(
            @CookieValue("JSESSIONID") String jsessionId
    ){
        System.out.println("jsessionId:" + jsessionId);
        return "success";
    }
}
```

输出如下：

```
jsessionId:6D475398DBDE8A1006F3B98C6CB555D0
```

4.6、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：传输的请求参数的参数名一定要和实体类中的属性名一致(属性名不是变量名，是get、set方法去掉get、set后首字母小写)。

首先创建一个实体类User：

```java
public class User {
    private Integer id;
    private  String username;
    private String password;

    public User() {
    }

    public User(Integer id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}
```

控制器方法如下：

```
@Controller
public class TestParamController {
    @RequestMapping("/param/pojo")
    public String getParamByPojo(User user){
        System.out.println(user);
        return "success";
    }
}
```

当请求参数的参数名为username和password并其值分别为admin和123456时，输出如下：

```
User{id=null, username='admin', password='123456'}
```

4.7、解决获取请求参数的乱码问题

在tomcat8.5中GET请求不乱码，POST请求会乱码。这种乱码问题可以使用SpringMVC提供的编码过滤器CharacterEncodingFilter，但是必须在web.xml中进行注册。在web.xml中配置的代码如下：

```
<!--配置Spring的编码过滤器-->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

> 注意：
>
> 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。

5、域对象共享数据

Java四大域对象，作用范围由小到大为：

- page(jsp有效)—>page域指的是pageContext.（在这里不使用jsp，因此不再介绍）
- request(一次请求)—>request域request HttpServletContext（请求域）
- session(一次会话)—>session域session HttpSession（会话域）
- application(当前web应用)—>application域指的是application ServletContext*（应用域）

之所以他们是域对象，原因是他们都内置了map集合，都有setAttribute和getAttribute方法。

Request域:

1. 生命周期：

   在Service方法调用前由服务器创建，传入service方法。整个请求结束，request生命结束。

   用户发送一个请求，开始，服务器返回响应，请求结束，生命周期结束；

2. 作用范围：整个请求链（请求转发也存在）

3. 作用：在整个请求链中共享数据，经常用到：在servlet中处理好的数据交给JSP显示，此时参数就可以放在Request域中。

HttpSession 域:

1. 生命周期：

   在第一次调用request.getSession()方法时，服务器会检查是否已经有对应的session，如果没有就在内存中创建一个session并返回。

   1. 当一段时间内session没有被使用（默认为`30`分钟），则服务器会销毁该session。
   2. 如果服务器非正常关闭，没有到期的session也会跟着销毁。
   3. 如果调用session提供的invalidate()，可以立即销毁session。

2. 作用范围：一次会话。

   HttpSession 在服务器中，为浏览器创建独一无二的内存空间，在其中保存会话相关的信息。

   注意：服务器正常关闭，再启动，Session对象会进行钝化和活化操作。同时如果服务器钝化的时间在session 默认销毁时间之内， 则活化后session还是存在的。否则Session不存在。 如果JavaBean 数据在session钝化时，没有实现Serializable 则当Session活化时，会消失

ServletContext:

1. 生命周期：

   当WEB应用被加载进容器创建代表整个WEB应用的ServletContext对象；

   当服务器关闭或WEB应用被移除时，ServletContext对象跟着被销毁。

2. 作用范围：整个WEB应用。

5.1、使用ServletAPI向request域对象共享数据

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/servletapi}">测试通过使用ServletAPI向请求域共享数据</a>
```

在success.html中添加p标签，使用thymeleaf语法展示request域中的数据

```
<p th:text="${testRequestScope}"></p>
```

控制器中的方法如下：

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

5.2、使用ModelAndView向request域对象共享数据

使用ModelAndView时，可以使用其Model功能向请求域共享数据，使用View功能设置逻辑视图，但是控制器方法一定要将ModelAndView作为方法的返回值。

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/mav}">测试通过ModelAndView向请求域共享数据</a>
```

在success.html中添加p标签，使用thymeleaf语法展示request域中的数据

```
<p th:text="${testRequestScope}"></p>
```

控制器中的方法如下：

```
@RequestMapping("/test/mav")
public ModelAndView testMAV(){
    /**
     * ModelAndView包含Model和View的功能：
     * Model：向请求域中共享数据
     * View：设置逻辑视图实现页面跳转
     *
     */
    ModelAndView mav = new ModelAndView();
    //向请求域中共享数据
    mav.addObject("testRequestScope","hello,ModelAndView");
    //设置逻辑视图
    mav.setViewName("success");
    return mav;
}
```

5.3、使用Model向request域对象共享数据

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/model}">测试通过model向请求域共享数据</a>
```

在success.html中添加p标签，使用thymeleaf语法展示request域中的数据

```
<p th:text="${testRequestScope}"></p>
```

控制器中的方法如下：

```
@RequestMapping("/test/model")
public String testModel(Model model){
    model.addAttribute("testRequestScope","hello,Model");
    return "success";
}
```

5.4、使用map向request域对象共享数据

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/map}">测试通过map向请求域共享数据</a>
```

在success.html中添加p标签，使用thymeleaf语法展示request域中的数据

```
<p th:text="${testRequestScope}"></p>
```

控制器中的方法如下：

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

5.5、使用ModelMap向request域对象共享数据

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/modelmap}">测试通过modelMap向请求域共享数据</a>
```

在success.html中添加p标签，使用thymeleaf语法展示request域中的数据

```
<p th:text="${testRequestScope}"></p>
```

控制器中的方法如下：

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

5.6、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.

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

5.7、向session域和application域共享数据

向session域和application域共享数据一般使用ServletAPI，相较于SpringMVC提供的方式并没有使用ServletAPI简单。

在index.html页面中添加a标签，以便于发送请求。

```
<a th:href="@{/test/session}">测试会话域共享数据</a>
<a th:href="@{/test/application}">测试应用域共享数据</a>
```

控制器中的方法如下：

```
//session域共享数据的方法
@RequestMapping("/test/session")
public String testSession(HttpSession session){
    session.setAttribute("testSessionScope","hello,session");
    return "success";
}
//application域共享数据的方法
@RequestMapping("/test/application")
public String testApplication(HttpSession session){
    ServletContext servletContext = session.getServletContext();

 servletContext.setAttribute("testApplicationScope","hello,application");
    return "success";
}
```

在success.html中添加p标签，使用thymeleaf语法将域中的数据展示出来，在thymeleaf语法要获取session域中的数据需要在共享数据的属性名前加上`session.`；要获取application域中的数据需要在共享数据的属性名前加上`application.`。

```
<p th:text="${session.testSessionScope}"></p>
<p th:text="${application.testApplicationScope}"></p>
```

6、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。

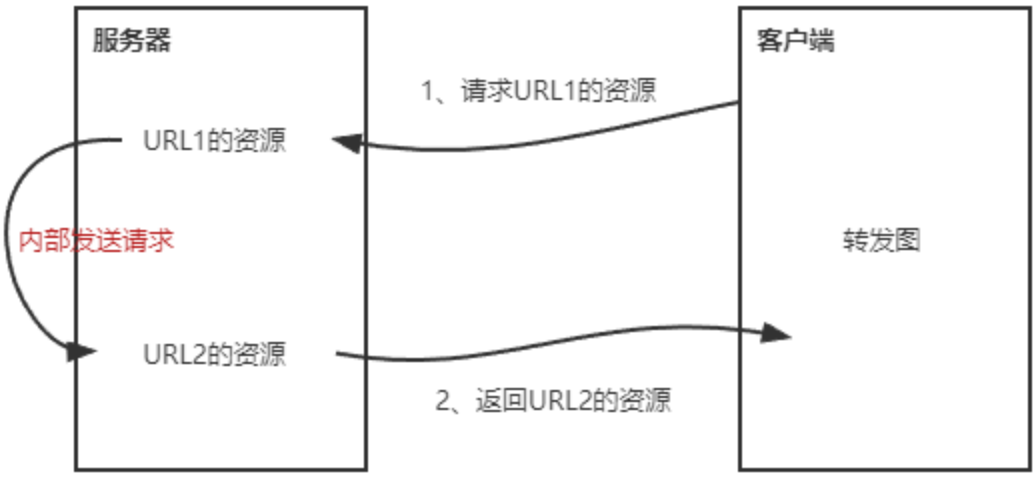
SpringMVC视图的种类很多，默认有转发视图和重定向视图。

当工程引入jstl的依赖，转发视图会自动转换为JstlView。

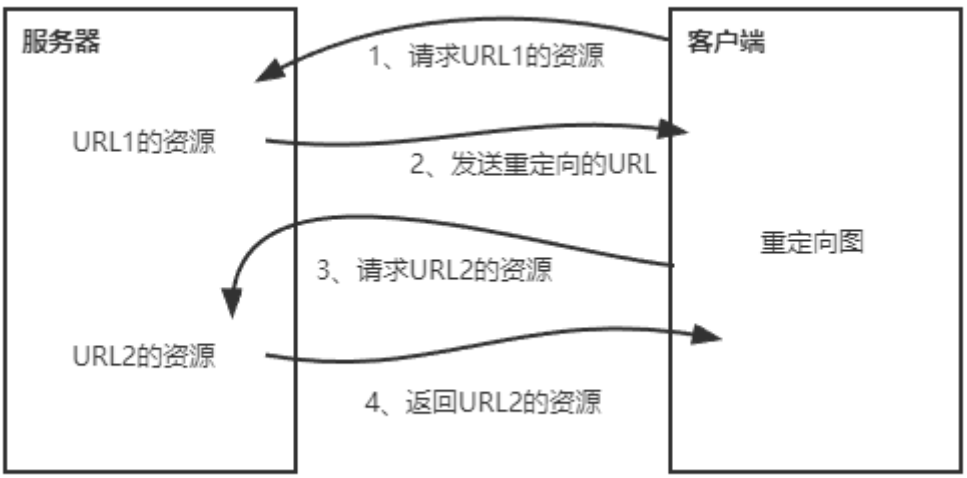
若使用的视图技术为Thymeleaf，在SpringMVC的配置文件中配置了Thymeleaf的视图解析器，由此视图解析器解析之后所得到的是ThymeleafView。

转发和重定向：

转发——>客户浏览器发送HTTP请求——>web服务器接受请求——>调用内部一个方法在容器内部完成请求处理和转发动作——>再将转发跳转到的那个网页资源返回给客户； **转发只能在同一个容器内完成 转发的时候浏览器地址是不会变的，在客户浏览器里只会显示第一次进入的那个网址或者路径，客户看不到这个过程，只是得到了想要的目标资源。**转发行为浏览器只做了一次请求。（转发只能跳转一次）



重定向——>客户浏览器发送HTTP请求——>web服务器接受请求后发送302状态码以及新的位置给客户浏览器——>客户浏览器发现是302响应，则自动再发送一个新的HTTP请求，请求指向新的地址（302：Found 临时移动，但资源只是临时被移动。即你访问网址A，但是网址A因为服务器端的拦截器或者其他后端代码处理的原因，会被重定向到网址B。）——>**服务器根据此请求寻找资源发个客户；再客户浏览器中显示的是重定向之后的路径，客户可以看到地址的变化。**重定向行为浏览器做了至少两次请求。（重定向可以跳转多次）



业务逻辑失败用转发，成功用重定向。

6.1、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。

在index.html中设置请求地址：

```
<a th:href="@{/test/view/thymeleaf}">测试SpringMVC的视图ThymeleafView</a>
<br>
```

控制器方法如下：

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

返回一个字符串success，该名称会被视图解析器解析，加上前缀和后缀得到最终的路径，然后实现跳转到success.html界面中。

6.2、转发视图

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。

在index.html中设置请求地址：

```
<a th:href="@{/test/view/forward}">测试SpringMVC的视图
InternalResourceView</a><br>
```

控制器方法如下：

```
@RequestMapping("/test/view/forward")
public String testInternalResourceView(){
    return "forward:/test/model";
}
```

当向`/test/view/forward`发送请求时，会通过转发的方式访问`/test/view/forward`地址，然后执行后续的操作。

6.3、重定向视图

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。

在index.html中设置请求地址：

```
<a th:href="@{/test/view/redirect}">测试SpringMVC的视图RedirectView</a>
```

控制器方法如下：

```
@RequestMapping("/test/view/redirect")
public String testRedirectView(){
    return "redirect:/test/model";
}
```

当向`/test/view/redirect`发送请求时，会通过重定向的方式访问`/test/view/redirect`地址，然后执行后续的操作。

6.4、视图控制器view-controller

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

如下演示演示通过"/"来访问index.html页面的配置：

```
<!--
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>
```

当SpringMVC中设置任何一个view-controller时，其他控制器中的请求映射（@RequestMapping）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：

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

7、RESTful

7.1、RESTful简介

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

1、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。

2、资源的表述

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。

3、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。

7.2、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

7.3、HiddenHttpMethodFilter（支持发送PUT和DELETE请求）

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：

- 当前请求的请求方式必须为post。
- 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**，要放在编码过滤器的后面：

```
<!--处理请求方式的过滤器-->
<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(编码过滤器)和HiddenHttpMethodFilter(请求方式的过滤器)。
>
> 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);`

下面用一个小demo演示一下在html和控制器方法中如何使用：

在index.html文件中创建查询、保存、更改、删除的请求如下：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
<h1>index.html</h1>
<a th:href="@{/user}">查询所有的用户信息</a><br>
<a th:href="@{/user/1}">查询id为1的用户信息</a><br>
<form th:action="@{/user}" method="post">
```

```
    <input type="submit" value="添加用户信息">
</form>
<form th:action="@{/user}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="submit" value="修改用户信息">
</form>
<form th:action="@{/user/5}" method="post">
    <input type="hidden" name="_method" value="delete">
    <input type="submit" value="删除用户信息">
</form>
</body>
</html>
```

查询、保存、更改、删除控制器方法如下，也可以使用派生注解：

```
@Controller
public class TestRestController {
    /*@GetMapping("/user")*/
    @RequestMapping(value = "/user",method = RequestMethod.GET)
    public String getAllUser(){
        System.out.println("查询所有的用户信息");
        return "success";
    }
    /*@GetMapping("/user/{id}")*/
    @RequestMapping(value = "/user/{id}",method = RequestMethod.GET)
    public String getUserById(@PathVariable("id") Integer id){
        System.out.println("根据id查询用户信息-->/user" + id + "-->get");
        return "success";
    }
    /*@PostMapping("/user")*/
    @RequestMapping(value = "/user",method = RequestMethod.POST)
    public String insertUser(){
        System.out.println("添加用户信息-->/user-->post");
        return "success";
    }
    /*@PutMapping("/user")*/
    @RequestMapping(value = "/user",method = RequestMethod.PUT)
    public String updateUser(){
        System.out.println("修改用户信息-->/user-->put");
        return "success";
    }
    /*@DeleteMapping("/user/{id}")*/
    @RequestMapping(value = "/user/{id}",method = RequestMethod.DELETE)
    public String deleteUser(@PathVariable("id") Integer id){
        System.out.println("删除用户信息-->/user/"+ id +"-->delete");
        return "success";
    }
}
```

8、RESTful案例

8.1、准备工作

和传统 CRUD 一样，实现对员工信息的增删改查。

实体类：

```
public class Employee {
    private Integer id;
    private String lastName;
    private String email;
    //1 male 0 female
    private Integer gender;

    public Employee() {
    }

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

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", email='" + email + '\'' +
                ", gender=" + gender +
                '}';
    }
}
```

准备dao层模拟数据：

```
@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);
    }

}
```

web.xml配置如下

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<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/javaee/web-app_4_0.xsd"
         version="4.0">
    <!--配置Spring的编码过滤器-->
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!--处理请求方式的过滤器-->
    <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>
    <!--设置SpringMVC的前端控制器-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

springmvc.xml配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!--扫描控制层组件-->
    <context:component-scan base-package="com.tianna"/>

    <!--配置Thymeleaf视图解析器-->
    <bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
        <!--优先级-->
        <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.SpringResourceTemplateResolve
r">
                        <!-- 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" />
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

    <!--
        配置默认的servlet处理静态资源，
        当前工程的web.xml配置的前端控制器DispatcherServlet的url-pattern是/
        tomcat的web.xml配置的DefaultServlet的url-pattern也是/
        此时，浏览器发送的请求会优先被DispatcherServlet进行处理，但是
DispatcherServlet无法处理静态资源
        如配置了<mvc:default-servlet-handler/>，此时浏览器发送的所有请求都会被
DefaultServlet处理
        若配置了<mvc:default-servlet-handler/>和<mvc:annotation-driven/>
```

```
        浏览器发送的请求会先被DispatcherServlet处理，无法处理在交给
DefaultServlet处理
    -->
    <mvc:default-servlet-handler/>

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

</beans>
```

在控制器中自动装配dao层的对象：

```
@Controller
public class EmployeeController {
    @Autowired
    private EmployeeDao employeeDao;
}
```

8.2、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/1	DELETE
跳转到添加数据页面	/to/add	GET
执行保存	/employee	POST
跳转到更新数据页面	/employee/1	GET
执行更新	/employee	PUT

8.3、具体功能：访问首页

1、在springmvc.xml中配置view-controller

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

2、创建页面

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

8.4、具体功能：查询所有员工信息

1、控制器方法：

```
@RequestMapping(value = "/employee",method = RequestMethod.GET)
public String getAllEmployee(Model model){
    //获取所有的员工信息
    Collection<Employee> allEmployee = employeeDao.getAll();
    //将所有的员工信息在请求域中共享
    model.addAttribute("allEmployee",allEmployee);
    //跳转到列表页面
    return "employee_list";
}
```

2、创建employee_list.html来展示数据

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>employee list</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}">
</head>
<body>
<div id="app">
    <table>
        <tr>
            <th colspan="5">employee list</th>
        </tr>
        <tr>
            <th>id</th>
            <th>lastName</th>
            <th>email</th>
            <th>gender</th>
            <th>options（<a th:href="@{/to/add}">add</a>）</th>
        </tr>
```

```
        <tr th:each="employee : ${allEmployee}">
            <td th:text = "${employee.id}"></td>
            <td th:text = "${employee.lastName}"></td>
            <td th:text = "${employee.email}"></td>
            <td th:text = "${employee.gender}"></td>
            <td>
                <a @click = "deleteEmployee" th:href="@{'/employee/' +
${employee.id}}">delete</a>
                <a th:href="@{'/employee/' + ${employee.id}}">update</a>
            </td>
        </tr>
    </table>
    <form method="post">
        <input type="hidden" name="_method" value = "delete">
    </form>
</div>

<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            deleteEmployee(){
                //获取form表单
                var form = document.getElementsByTagName("form")[0];
                //将超链接的href属性值赋值给form表单的action属性
                //event.target表示当前触发事件的标签
                form.action = event.target.href;
                //表单提交
                form.submit();
                //组织超链接的默认行为
                event.preventDefault();
            }
        }
    })

</script>
</body>
</html>
```

8.5、具体功能：删除

1、删除超链接

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

由于删除时需要发送delete请求，前端发送这种请求需要在form表单中完成，因此还需要创建一个form表单，超链接点击时提交form表单，已完成发送delete请求的功能。

2、创建处理delete请求方式的表单

```
<form method="post">
    <input type="hidden" name="_method" value = "delete">
</form>
```

3、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>
```

通过vue处理点击事件

```
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            deleteEmployee(){
                //获取form表单
                var form = document.getElementsByTagName("form")[0];
                //将超链接的href属性值赋值给form表单的action属性
                //event.target表示当前触发事件的标签
                form.action = event.target.href;
                //表单提交
                form.submit();
                //组织超链接的默认行为
                event.preventDefault();
            }
        }
    })
</script>
```

4、控制器方法

```
@RequestMapping(value = "/employee/{id}",method = RequestMethod.DELETE)
public String deleteEmployee(@PathVariable("id") Integer id){
    employeeDao.delete(id);
    //重定向到列表功能
    return "redirect:/employee";
}
```

8.6、添加数据

1、在springmvc.xml中配置view-controller跳转到添加数据页面

```
<mvc:view-controller path="/to/add" view-name="employee_add"></mvc:view-
controller>
```

2、创建添加数据页面employee_add.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>add employee</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}">
</head>
<body>
<form th:action="@{/employee}" method="post">
    <table>
        <tr>
            <th colspan="2">add employee</th>
        </tr>
        <tr>
            <td>lastName</td>
            <td>
                <input type="text" name="lastName">
            </td>
        </tr>
        <tr>
            <td>email</td>
            <td>
                <input type="text" name="email">
            </td>
        </tr>
        <tr>
            <td>gender</td>
            <td>
                <input type="radio" name="gender" value="1">male
                <input type="radio" name="gender" value="0">female
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="add">
            </td>
        </tr>
    </table>

</form>
</body>
</html>
```

3、执行保存功能，然后重定向到信息展示页面

```
@RequestMapping(value = "/employee",method = RequestMethod.POST)
public String addEmployee(Employee employee){
    //保存员工信息
    employeeDao.save(employee);
    //重定向到列表功能：/employee
    return "redirect:/employee";
}
```

8.7、更新数据

1、修改方法超链接

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

2、根据id查询需要修改的员工的全部内容，然后将这些内容传到修改页面中

控制器方法：

```
@RequestMapping(value = "/employee/{id}",method = RequestMethod.GET)
public String toUpdate(@PathVariable("id") Integer id,Model model){
    //根据id查询员工信息
    Employee employee = employeeDao.get(id);
    //将员工信息共享到请求域中
    model.addAttribute("employee",employee);
    //跳转到employee_update
    return "employee_update";
}
```

3、创建修改页面employee_update.html，用于修改信息

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>add employee</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}">
</head>
<body>
<form th:action="@{/employee}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="hidden" name="id" th:value="${employee.id}">
    <table>
        <tr>
            <th colspan="2">add employee</th>
        </tr>
        <tr>
            <td>lastName</td>
            <td>
                <input type="text" name="lastName"
th:value="${employee.lastName}">
```

```
            </td>
        </tr>
        <tr>
            <td>email</td>
            <td>
                <input type="text" name="email"
th:value="${employee.email}">
            </td>
        </tr>
        <tr>
            <td>gender</td>
            <td>
                <input type="radio" name="gender" value="1"
th:field="${employee.gender}">male
                <input type="radio" name="gender" value="0"
th:field="${employee.gender}">female
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="update">
            </td>
        </tr>
    </table>

</form>

</body>
</html>
```

3、执行更新的控制器方法

```
@RequestMapping(value = "employee" ,method = RequestMethod.PUT)
public String UpdateEmployee(Employee employee){
    //修改员工信息
    employeeDao.save(employee);
    //重定向到列表功能
    return "redirect:/employee";
}
```

8.8、处理静态资源

当前工程的web.xml配置的前端控制器DispatcherServlet的url-pattern是/

tomcat的web.xml配置的DefaultServlet的url-pattern也是/

此时，浏览器发送的请求会优先被DispatcherServlet进行处理，但是DispatcherServlet无法处理静态资源。

解决方法：

如配置了`<mvc:default-servlet-handler/>`，此时浏览器发送的所有请求都会被DefaultServlet处理。

若配置了`<mvc:default-servlet-handler/>`和`<mvc:annotation-driven/>`浏览器发送的请求会先被DispatcherServlet处理，无法处理在交给DefaultServlet处理。

因此需要配在springmvc.xml中配置`<mvc:default-servlet-handler/>`和`<mvc:annotation-driven/>`：

```
<!--配置默认的servlet处理静态资源-->
<mvc:default-servlet-handler/>

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

9、SpringMVC处理ajax请求

9.1、axios的使用

axios的使用方式：

```
axios({
    url:"",//请求路径
    method:"",//请求方式
    //以name=value&name=value的方式发送的请求参数，
    //请求参数会被拼接到请求地址后，get和post都可以。
    //此种方式的请求参数可以通过request.getParameter()获取
    params:{},
    //以json格式发送的请求参数，
    //请求参数会被保存到请求报文的请求体传输到服务器，设置为post，get无请求体。
    //此种方式的请求参数不可以通过request.getParameter()获取
     data:{}
     }).then(response=>{
         console.log(response.data)
     });
```

params中的参数和data的参数有以下区别：

params：

- 以name=value&name=value的方式发送的请求参数（和上面获取参数的方式一样，在请求地址后加问号那种）。
- 请求参数会被拼接到请求地址后，get和post都可以。
- 此种方式的请求参数可以通过request.getParameter()获取。

data：

- 以json格式发送的请求参数。
- 请求参数会被保存到请求报文的请求体传输到服务器，设置为post，get无请求体。
- 此种方式的请求参数不可以通过request.getParameter()获取。

下面简单演示一下如何接受ajax发送的数据并响应结果。

在index.html中发送ajax请求：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
<div id="app">
    <h1>index.html</h1>
    <input type="button" value="测试SpringMVC处理ajax" @click="testAjax()">
<br>
</div>
<script type="text/javascript" th:src="@{/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testAjax(){
                axios.post(
                    "/SpringMVC/test/ajax?id=1001",
                    {username:"admin",password:"123456"}
                ).then(respose=>{
                   console.log(respose.data);
                });
            }
        }
    })
</script>
</body>
</html>
```

其中id=1001为params中的参数，可以通过request.getParameter()获取，而{username:"admin",password:"123456"}为data中的json个数的数据，无法通过request.getParameter()获取，需要使用@RequestBody注解，**该注解可以将请求体中的内容和控制器方法的形参进行绑定。**

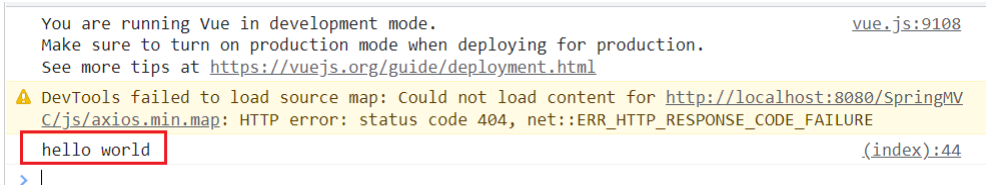
在控制器方法中处理ajax请求并响应数据：

```
@RequestMapping(value = "/test/ajax")
public void testAjax(Integer id, @RequestBody String
requestBody,HttpServletResponse response) throws IOException {
    System.out.println("requestBody:" + requestBody);
    System.out.println("id:" + id);
    response.getWriter().write("hello world");
}
```

输出结果如下：

```
requestBody:{"username":"admin","password":"123456"}
id:1001
```

响应结果如下：



9.2、@RequestBody

@RequestBody可以获取请求体信息（ajax请求中data中的信息），使用@RequestBody注解标识控制器方法的形参，当前请求的请求体就会为当前注解所标识的形参赋值。

```
<!--此时必须使用post请求方式，因为get请求没有请求体-->
<form th:action="@{/test/RequestBody}" method="post">
用户名：<input type="text" name="username"><br>
密码：<input type="password" name="password"><br>
<input type="submit">
</form>
```

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

输出结果：

```
requestBody:username=admin&password=123456
```

9.3、@RequestBody获取json格式的请求参数

在使用了axios发送ajax请求之后，浏览器发送到服务器的请求参数有两种格式：

1. name=value&name=value...，此时的请求参数可以通过request.getParameter()获取，对应SpringMVC中，可以直接通过控制器方法的形参获取此类请求参数。
2. {key:value,key:value,...}，此时无法通过request.getParameter()获取，之前我们使用操作json的相关jar包gson或jackson处理此类请求参数，可以将其转换为指定的实体类对象或map集合。在SpringMVC中，直接使用@RequestBody注解标识控制器方法的形参即可将此类请求参数转换为java对象。

下面介绍获取第二种方式的json数据并将其转为java对象。

使用@RequestBody注解将json格式的请求参数转换为java对象的步骤如下：

1、在pom.xml中导入jackson的依赖：

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

2、SpringMVC的配置文件中设置开启mvc的注解驱动

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

3、在控制器方法的形参位置，设置json格式的请求参数要转换成的java类型（实体类或map）的参数，并使用@RequestBody注解标识。

html文件的ajax请求：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
<div id="app">
    <h1>index.html</h1>
    <input type="button" value="使用@RequestBody注解处理json格式的请求参数"
@click="testRequestBody()">
<script type="text/javascript" th:src="@{/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testRequestBody(){
                axios.post(
                    "/SpringMVC/test/RequestBody/json",
                    {username:"admin",password:"123456",age:23,gender:"男"}
                ).then(response=>{
                    console.log(response.data);
                });
            }

        }
    })
</script>
</body>
</html>
```

控制器方法如下：

```
@RequestMapping(value = "/test/RequestBody/json")
public void testRequestBody(@RequestBody User user,HttpServletResponse
response) throws IOException {
    System.out.println(user);
    response.getWriter().write("hello,RequestBody");
}
```

User实体类定义如下：

```
package com.tianna.pojo;

/**
 * @author tiancn
 * @date 2022/8/21 17:06
 */
public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;

    public User() {
    }

    public User(Integer id, String username, String password, Integer age,
String gender) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.gender = gender;
    }

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
```

```
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                '}';
    }
}
```

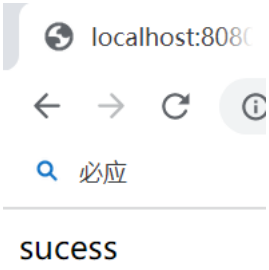
9.4、@ResponseBody

@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。

下面的控制器方法在加上@ResponseBody注解之后，会将success字符传返回到页面：

```
@RequestMapping(value = "/test/ResponseBody")
@ResponseBody
public String testResponseBody(){
    return "sucess";
}
```

页面结果：



9.5、@ResponseBody响应浏览器json数据

服务器处理ajax请求之后，大多数情况都需要向浏览器响应一个java对象，此时必须将java对象转换为json字符串才可以响应到浏览器，之前我们使用操作json数据的jar包gson或jackson将java对象转换为json字符串。在SpringMVC中，我们可以直接使用@ResponseBody注解实现此功能。

@ResponseBody响应浏览器json数据的条件：

1、导入jackson的依赖

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

2、在SpeingMVC的配置文件中设置`<mvc:annotation-driven/>`

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

3、使用@ResponseBody注解标识控制器方法，在方法中，将需要转换为json字符串并响应到浏览器的java对象作为控制器方法的返回值，此时SpringMVC就可以将此对象直接转换为json字符串并响应到浏览器。

常用的java对象转换为json的结果

- 实体类-->json对象
- map-->json对象
- list-->json数组

index.html发送ajax请求：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
<div id="app">
    <h1>index.html</h1>
```

```
    <input type="button" value="使用@ResponseBody注解响应json格式的数据"
@click="testResponseBody()">
</div>
<script type="text/javascript" th:src="@{/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testResponseBody(){
                axios.post(
                    "/SpringMVC/test/ResponseBody/json"
                ).then(response=>{
                    console.log((response.data))
                });
            }
        }
    })
</script>
</body>
</html>
```

控制器方法如下：

```
@RequestMapping("/test/ResponseBody/json")
@ResponseBody
public User testResponseBodyJson(){
    User user = new User(1001, "admin", "123456", 20, "男");
    return user;
}
```

9.6、@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

10、文件上传和下载

10.1、文件下载

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。

使用ResponseEntity实现下载文件的功能：

前端界面的请求为：

```
<a th:href="@{/test/down}">下载图片</a><br>
```

控制器方法如下：

```
@RequestMapping("/test/down")
public ResponseEntity<byte[]> testResponseEntity(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("/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
    MultiValueMap<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;
}
```

10.2、文件上传

文件上传的要求：

- form表单的请求方式必须为post。
- form表单必须设置属性enctype="multipart/form-data"。

其中enctype属性为浏览器向服务器传输请求参数的方式，默认为application/x-www-form-urlencoded；multipart/form-data:代表将表单中的数据以二进制的方式提交到服务器中；application/x-www-form-urlencoded:传到服务器只有数据没有文件。因此使用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。

上传步骤如下：

1、添加依赖

在pom.xml中添加如下依赖：

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

2、在SpringMVC的配置文件中添加配置

配置上传解析器，SpingMVC在获取该bean时不是通过类型获取的，而是通过id获取的，因此要设置id，并且id的值必须为`multipartResolver`。

```
<!--配置文件上传解析器-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
</bean>
```

3、在前端创建用于上传的form表单

```
<form th:action="@{/test/up}" method="post" enctype="multipart/form-data">
    头像：<input type="file" name="photo">
    <input type="submit" value="上传">
</form>
```

4、控制器方法

为了防止上传相同名称的文件导致前面的文件被覆盖的问题，因此使用uuid以防止文件名重复，上传路径可自行设置，上传的代码如下：

```
@RequestMapping("/test/up")
public String testUp(MultipartFile photo,HttpSession session) throws
IOException {
    //获取上传的文件的文件名
    String fileName = photo.getOriginalFilename();
    //获取上传文件的后缀名
    String hzName = fileName.substring(fileName.lastIndexOf("."));
    String uuid = UUID.randomUUID().toString();
    //拼接一个新的文件名
    fileName = uuid+hzName;
    //获取ServletContext对象
    ServletContext servletContext = session.getServletContext();
    //获取当前工程下photo目录下的真实路径 ,上传路径
    String photoPath = servletContext.getRealPath("photo");
    //创建photoPath所对应的File对象
    File file = new File(photoPath);
    //判断file所对应目录是否存在
    if(!file.exists()){
        file.mkdir();
    }
```

```
    String finalPath = photoPath + File.separator + fileName;
    //上传文件
    photo.transferTo(new File(finalPath));
    return "success";
}
```

11、拦截器

11.1、拦截器的配置

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，配置有两种方式：

第一种：这中配置方式都是对DispatcherServlet所处理的所有的请求进行拦截：

```
<bean id="firstInterceptor"
class="com.tianna.interceptor.FirstInterceptor"></bean>
<mvc:interceptors>
    <!--bean和ref标签所配置的拦截器默认对DispatcherServlet处理的所有的请求进行拦截-->
    <!--<bean class="com.tianna.interceptor.FirstInterceptor"></bean>-->
    <!--或者使用下面这种-->
    <ref bean="firstInterceptor"/>
</mvc:interceptors>
```

第二种：以下配置方式可以通过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。

```
<bean id="firstInterceptor"
class="com.tianna.interceptor.FirstInterceptor"></bean>
<mvc:interceptors>
    <mvc:interceptor>
        <!--配置需要拦截的请求的请求路径，/*值表示一层路径，例如/test，/**表示所有请求-->
        <mvc:mapping path="/**"/>
        <!--配置需要排除拦截的请求的请求路径-->
        <mvc:exclude-mapping path="/abc"/>
        <!--;配置拦截器-->
        <ref bean="firstInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

11.2、拦截器的使用（定义拦截器）

SpringMVC中的拦截器有三个抽象方法：

- preHandle():在控制器方法执行之前，其返回值表示对控制器方法的拦截(false)或放行(true)。

- postHandle():在控制器方法执行之后执行。
- afterCompletion():在控制器方法执行之后，且渲染视图完毕之后执行。

定义的拦截器需要实现`HandlerInterceptor`接口，一个简单的自定义的拦截器如下：

```
public class FirstInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("FirstInterceptor-->preHandle");
        return HandlerInterceptor.super.preHandle(request, response,
handler);
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("FirstInterceptor-->postHandle");
        HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("FirstInterceptor-->afterCompletion");
        HandlerInterceptor.super.afterCompletion(request, response,
handler, ex);
    }
}
```

11.3、多个拦截器的执行顺序

1、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()按照配置的顺序执行，而 postHandle()和afterCompletion()按照配置的反序执行。

2、若某个拦截器的preHandle()返回了false

preHandle()方法返回false的拦截器和它之前的拦截器的preHandle()方法都会执行。

所有的拦截器的postHandle()方法都不执行。

preHandle()方法返回false的拦截器之前的拦截器的afterCompletion()方法会执行,而preHandle()方法返回false的拦截器的afterCompletion()方法不会执行。

12、异常处理器

首先创建一个error.html，用于下面出现异常时跳转的页面，并接受请求域中共享的属性。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>错误</title>
</head>
<body>
<h1>error.html</h1>
<p th:text="${ex}"></p>
</body>
</html>
```

12.1、基于配置的异常处理器

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，基于配置的使用时有两个属性：

- exceptionMappings属性：设置指定异常要跳转的视图。
- 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。

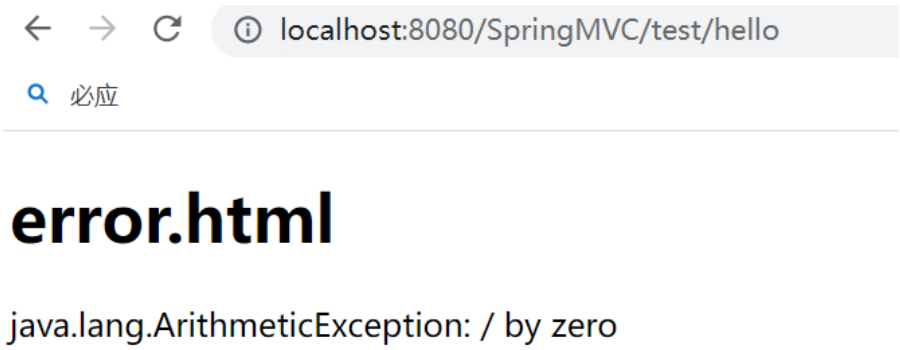
下面演示一个当出现数学运算异常时，跳转到error页面，并将异常信息在请求域中进行共享：

```
<!--配置异常处理-->
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <!--异常跳转的视图-->
    <property name="exceptionMappings">
        <props>
            <!--数学运算异常-->
            <!--key设置要处理的异常，value设置出现该异常时要跳转的页面所对应的逻辑视图-->
            <prop key="java.lang.ArithmeticException">error</prop>
        </props>
    </property>
    <!--设置共享在请求域中的异常信息的属性名-->
    <property name="exceptionAttribute" value="ex"></property>
</bean>
```

为模拟出现异常，可以在控制器方法中模拟数学运算异常：

```
@RequestMapping("/test/hello")
public String testHello(){
    System.out.println(1/0);
    return "success";
}
```

当出现异常时会跳转到error.html界面，并显示异常信息：



12.2、基于注解的异常处理

还可以通过注解的方式处理异常，创建一个处理异常的控制器，主要使用@ControllerAdvice和@ExceptionHandler注解，其作用如下：

- @ControllerAdvice：将当前类标识为异常处理的组件
- @ExceptionHandler：用于设置所标识方法处理的异常

具体代码如下：

```
//将当前类表示为异常处理的组件
@ControllerAdvice
public class ExceptionController {
    //设置要处理的异常信息
    @ExceptionHandler({ArithmeticException.class})
    public String handleException(Throwable ex, Model model){
        //ex表示控制器方法所出现的异常
        model.addAttribute("ex",ex);
        return "error";
    }
}
```

13、注解配置SpringMVC

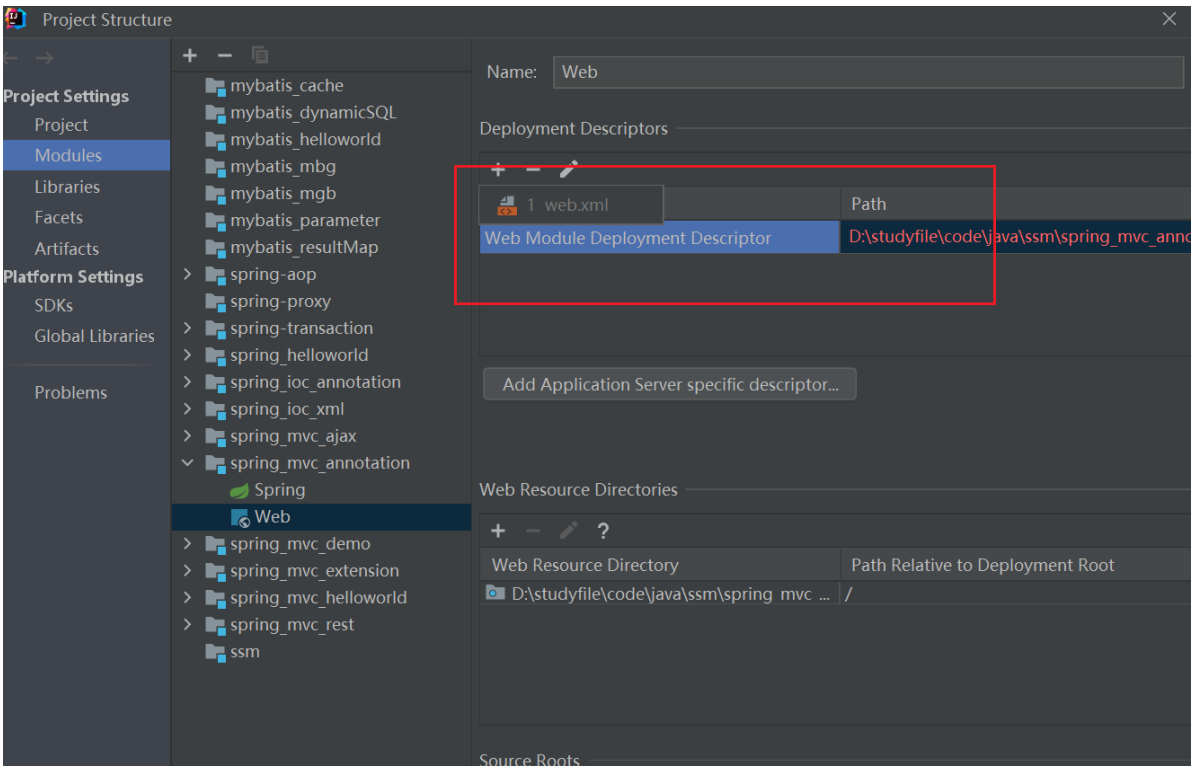
在Servlet3.0环境中，容器会在类路径中查找实现javax.servlet.ServletContainerInitializer接口的类，如果找到的话就用它来配置Servlet容器。 Spring提供了这个接口的实现，名为SpringServletContainerInitializer，这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成。Spring3.2引入了一个便利的WebApplicationInitializer基础实现，名为AbstractAnnotationConfigDispatcherServletInitializer，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。

13.1、准备工作

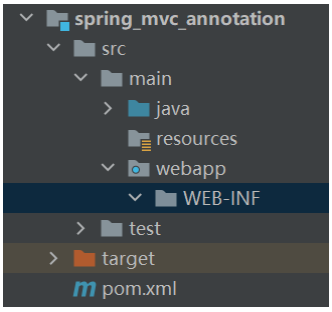
首先创建一个maven工程，然后引入以下依赖，并将打包方式设置为war包：

```
<packaging>war</packaging>

<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>
    <!--json数据有关-->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.1</version>
    </dependency>
    <!--文件上传有关-->
    <dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
    </dependency>
</dependencies>
```

添加web模块



由于使用注解方式配置SpringMVC，因此可以将web.xml删除，只保留webapp文件夹，删除后项目结构如下：



创建`com.tianna.config`包用于存放相关配置类，`com.tianna.controller`用于存放控制器，`com.tianna.interceptor`存放拦截器。

13.2、创建初始化类，代替web.xml

```
/**
 * @author tiancn
 * @date 2022/8/22 17:35
 * 用来代替web.xml
 */
public class WebInit extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    //设置一个配置类代替Spring的配置文件
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }

    @Override
    //设置一个配置类代替SpringMVC的配置文件
    protected Class<?>[] getServletConfigClasses() {
```

```
        return new Class[]{WebConfig.class};
    }

    @Override
    //设置SpringMVC的前端控制器DispatcherServlet的url-pattern(请求路径)
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    //设置当前的过滤器
    protected Filter[] getServletFilters() {
        //创建编码过滤器
        CharacterEncodingFilter characterEncodingFilter = new
CharacterEncodingFilter();
        characterEncodingFilter.setEncoding("UTF-8");
        characterEncodingFilter.setForceEncoding(true);
        //创建处理请求方式的过滤器
        HiddenHttpMethodFilter hiddenHttpMethodFilter = new
HiddenHttpMethodFilter();
        return new Filter[]
{characterEncodingFilter,hiddenHttpMethodFilter};
    }
}
```

13.3、创建SpringConfig配置类，代替spring的配置文件

```
/**
 * @author tiancn
 * @date 2022/8/22 17:40
 * 用来代替Spring的配置文件
 */
//将类标识为配置类
@Configuration
public class SpringConfig {
}
```

13.4、创建WebConfig配置类，代替SpringMVC的配置文件

```
/**
 * @author tiancn
 * @date 2022/8/22 17:40
 * 用来代替SpringMVC的配置文件
 * 扫描组件、视图解析器、默认的servlet、mvc注解驱动、视图控制器、文件上传解析
器、拦截器、异常解析器
 */
//将类标识为配置类
@Configuration
//扫描组件
@ComponentScan("com.tianna.controller")
```

```
//开启mvc的注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
    @Override
    //默认的servlet处理静态资源
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer)
{
        configurer.enable();
    }

    @Override
    //配置视图解析器
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }
    //@Bean注解可以将标识的方法的返回值作为bean进行管理，bean的id为方法的方法名
    @Bean
    public CommonsMultipartResolver multipartResolver(){
        return new CommonsMultipartResolver();
    }


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

    //配置异常解析器
    @Override
    public void
configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
        SimpleMappingExceptionResolver exceptionResolver = new
SimpleMappingExceptionResolver();
        Properties properties = new Properties();
        properties.setProperty("java.lang.ArithmeticException","error");
        exceptionResolver.setExceptionMappings(properties);
        exceptionResolver.setExceptionAttribute("ex");
        resolvers.add(exceptionResolver);
    }


    //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;
    }
    //生成视图解析器并未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;
    }
}
```

拦截器的代码如下：

```
/**
 * @author tiancn
 * @date 2022/8/22 19:23
 */
public class FirstInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        return HandlerInterceptor.super.preHandle(request, response,
handler);
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
    }

    @Override
```

```
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        HandlerInterceptor.super.afterCompletion(request, response,
handler, ex);
    }
}
```

13.5、功能测试

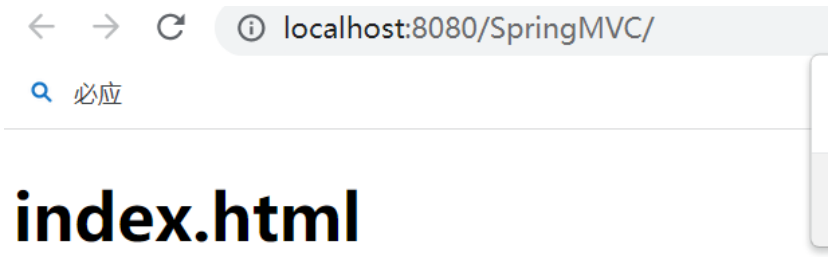
在webapp下的WEB-INF目录下创建templates文件夹用于存放视图文件，创建index.html界面：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>index.html</h1>
</body>
</html>
```

创建控制器方法，用于访问该界面（由于之前配置了视图解析器，可将其注释掉）：

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

最终可成功访问页面：



14、SpringMVC执行流程

14.1、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：处理器映射器，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：处理器，需要工程师开发（控制器controller）

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。

原理还看不懂，后面再写。。。。