SpringMVC

MVC是一种软件架构思想，将软件按照模型model，视图view，控制器controller

视图层：

指的是工程中的html，作用是与用户进 行交互，展示数据

控制层：

指的是工程中的servlet，作用是接收请求和响应浏览器

模型层：

指的是工程中的JavaBean，作用是处理数据

一种是实体类Bean专门储蓄业务数据的，如User

另一种是业务处理Bean专门用于处理业务逻辑和数据访问，如DAO

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通过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

特点

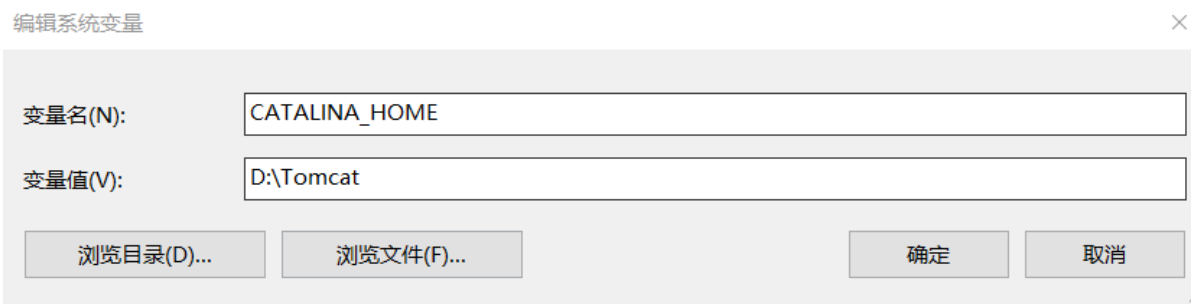
对servlet的封装有dispatcherservlet,对请求和响应统一管理

与spring无缝衔接

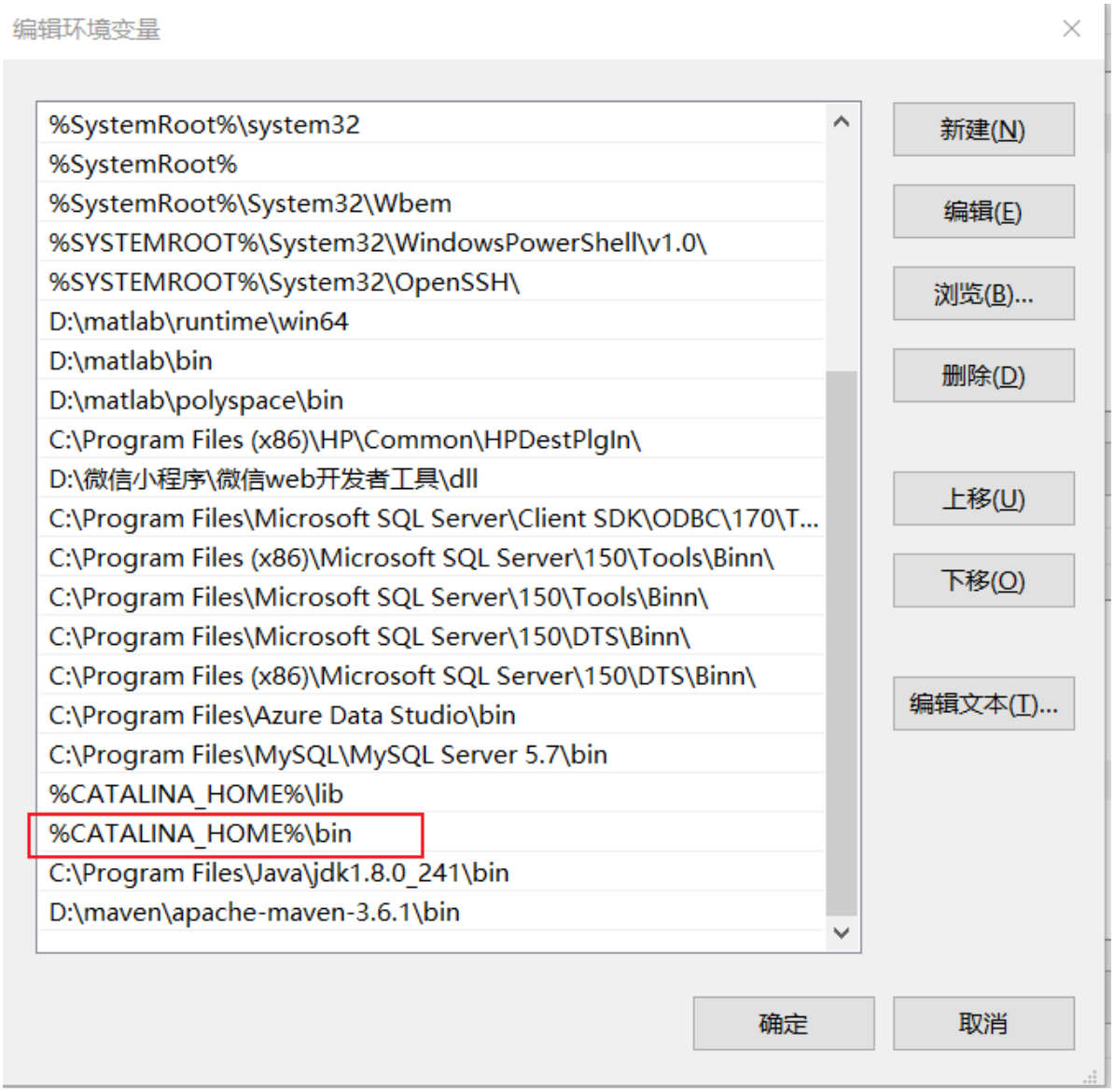
原先流程

先配置tomcat环境变量

CATALINA_HOME，以及tomcat位置

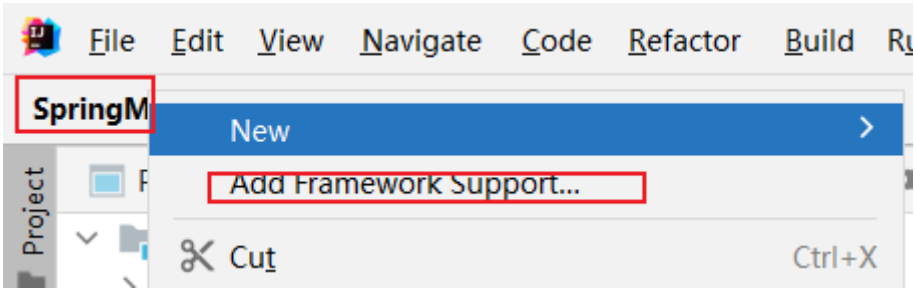


再在系统变量Path添加%CATALINA_HOME%\bin

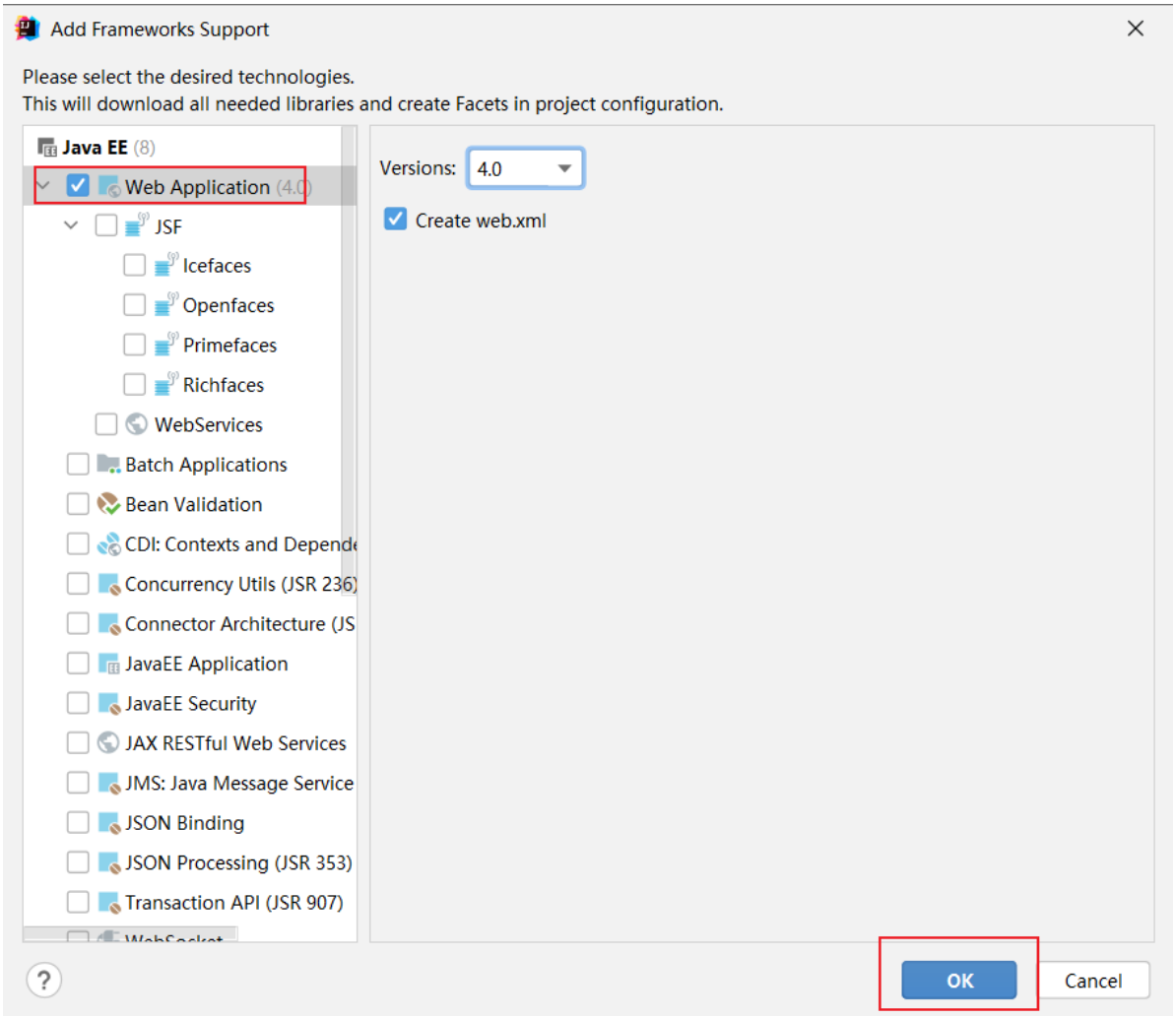


创建项目加入web框架

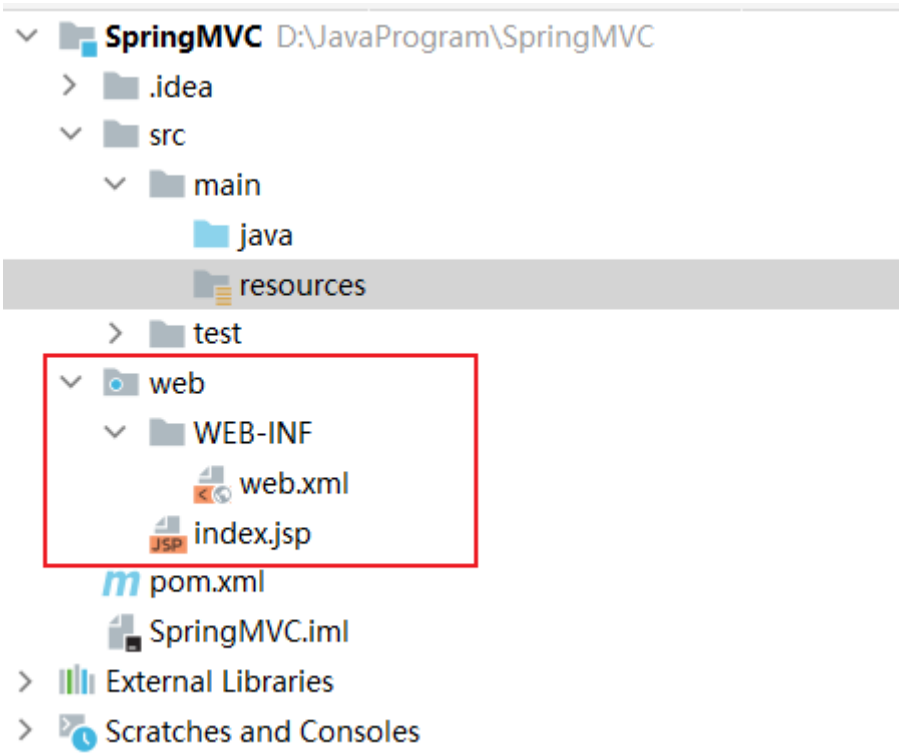
添加框架



添加web

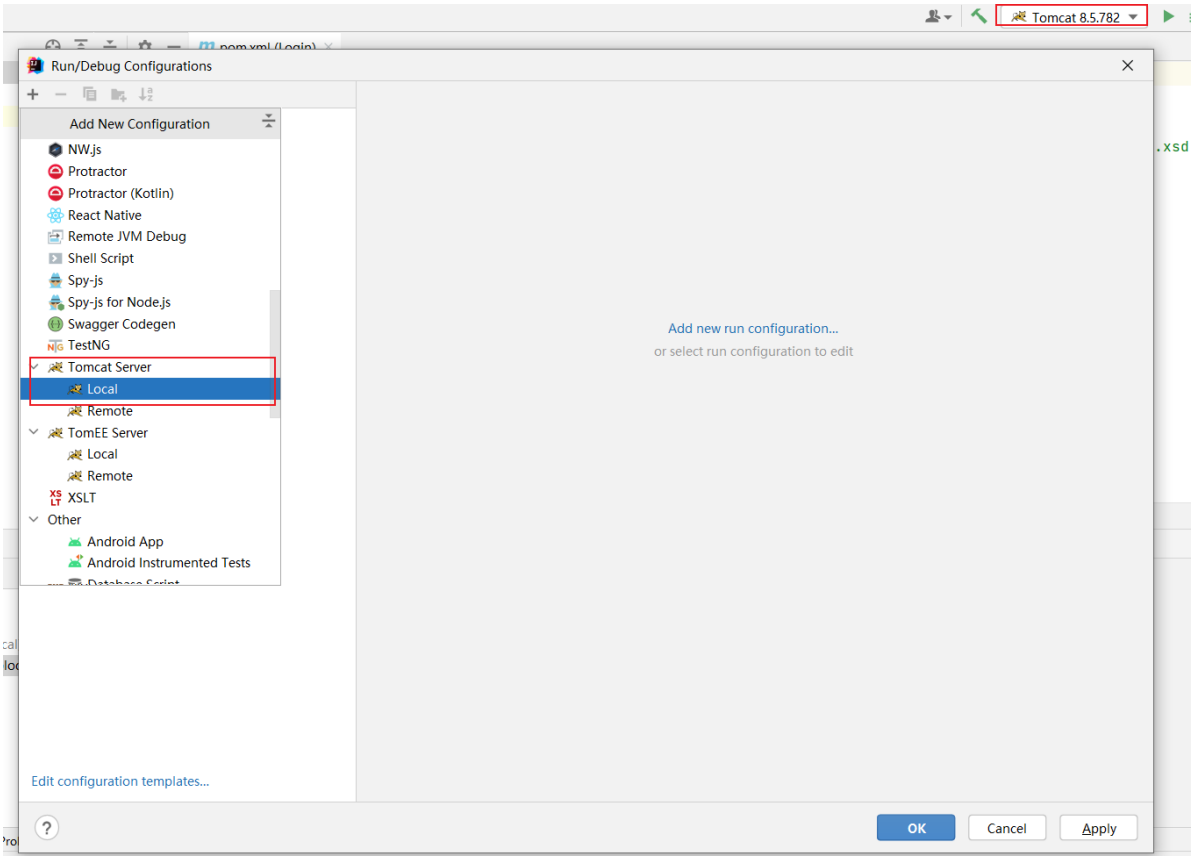


则有

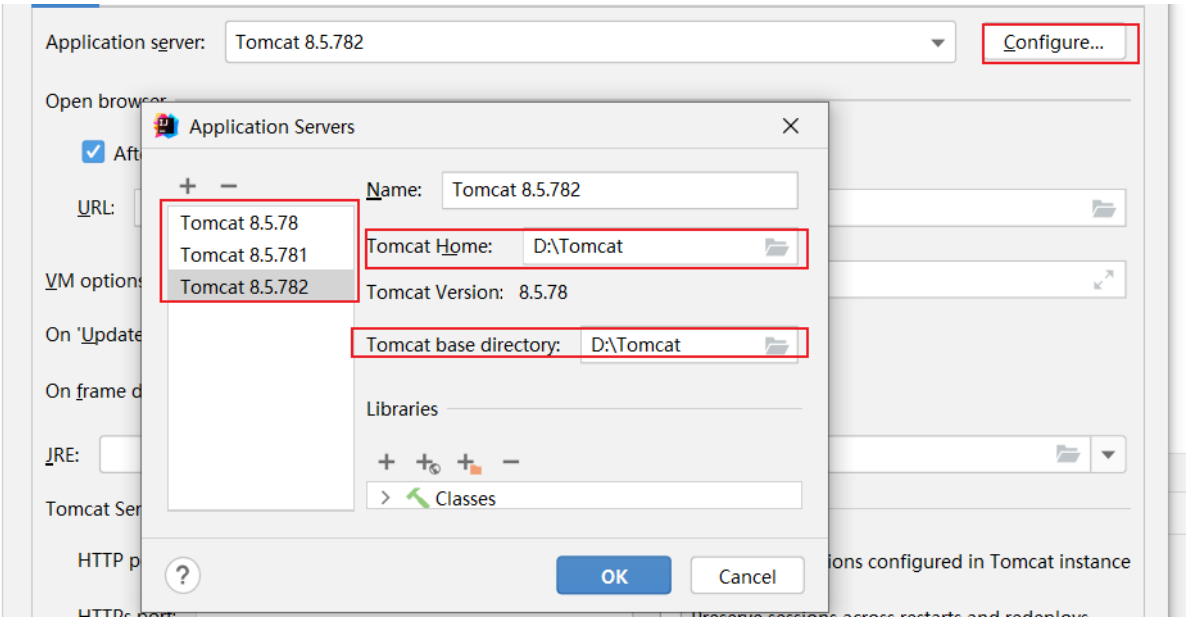


配置tomcat服务器

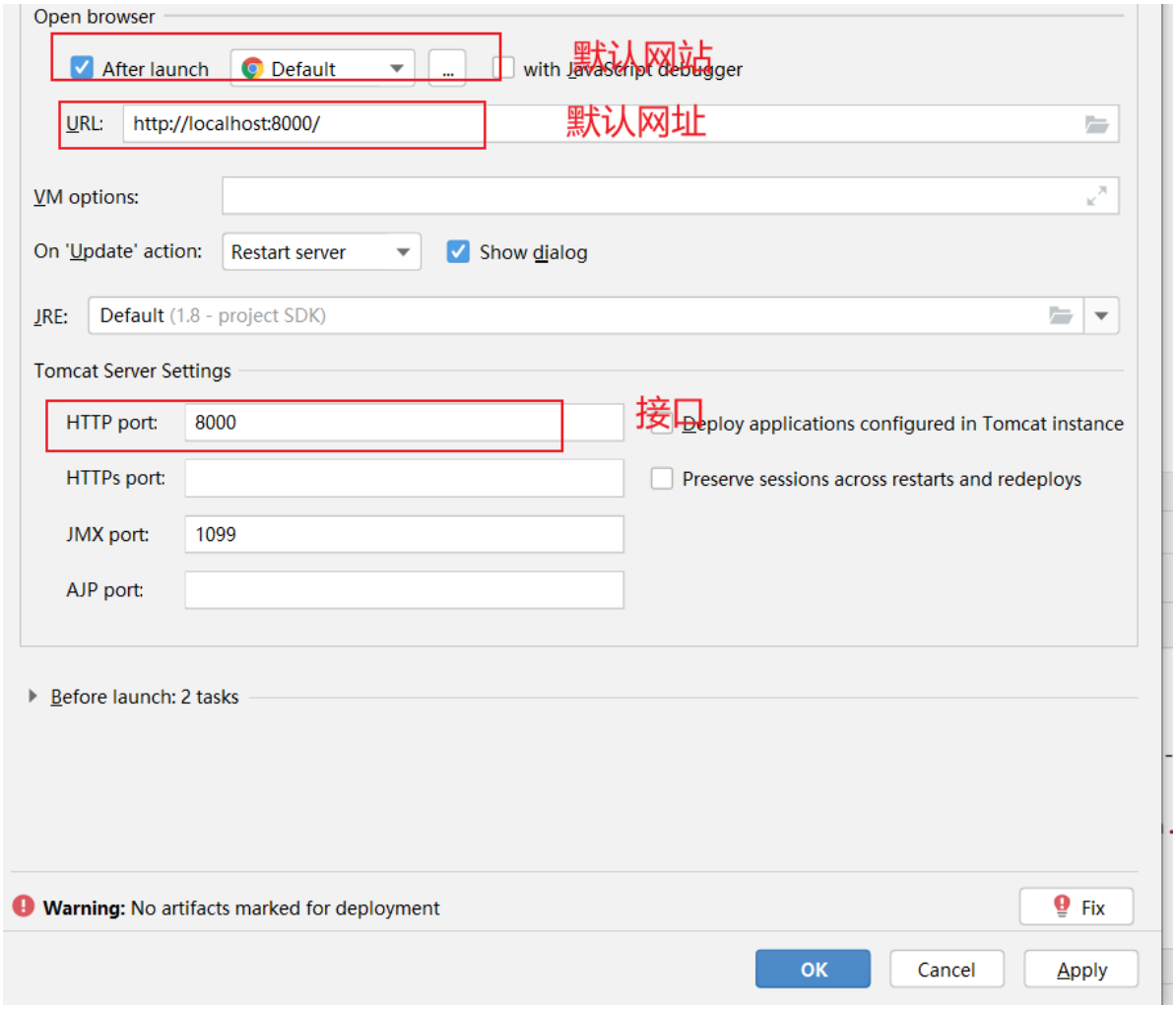
找到tomcat配置



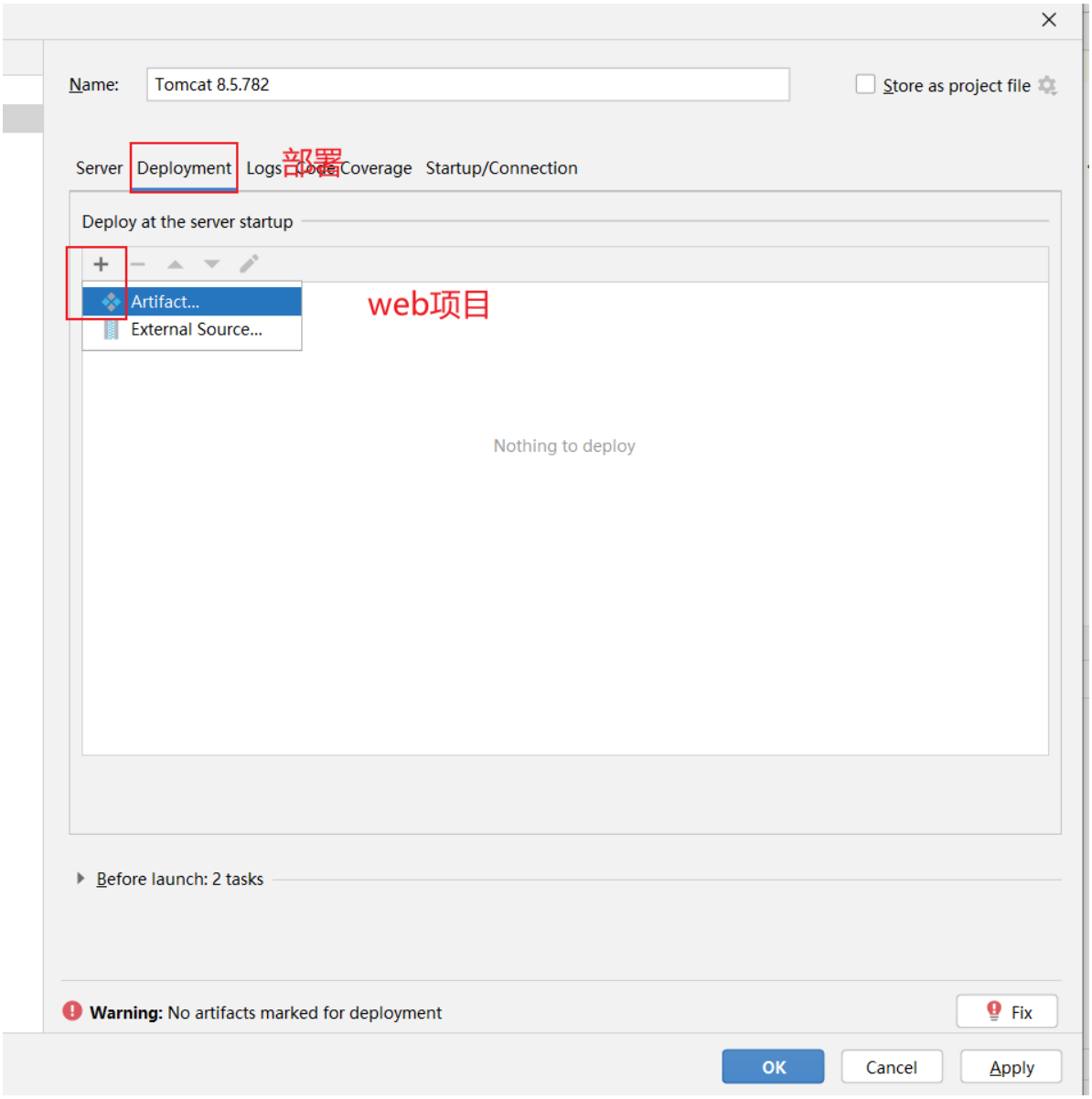
加入电脑的tomcat，要知道自己电脑tomcat的位置



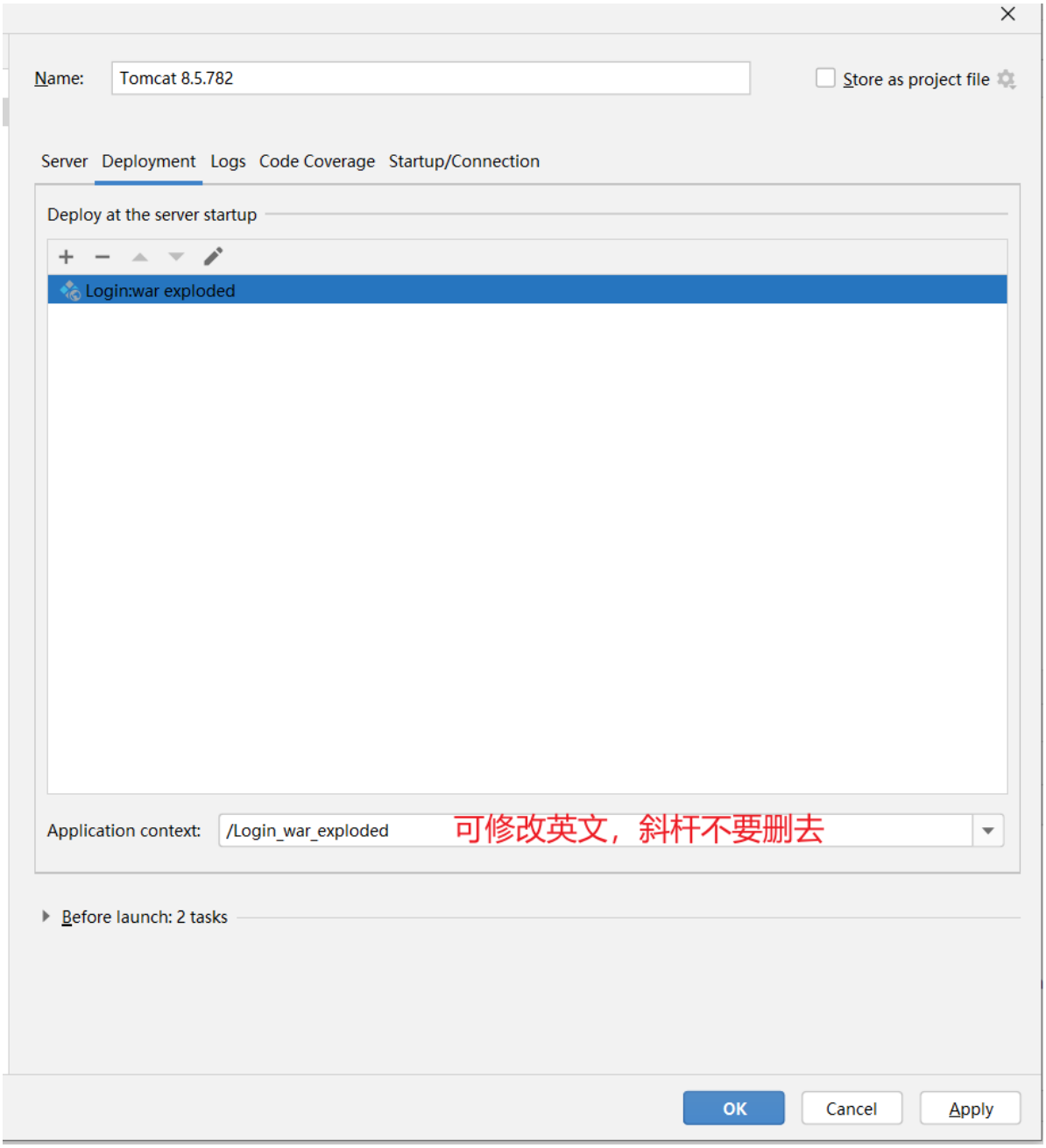
服务器属性，注意如果端口号用不了就随便换一个数字！


Open browser
After launch
Default
...
with JavaScript debugger
默认网站
URL:
http://localhost:8000/
默认网址
VM options:
On 'Update' action:
Restart server
Show dialog
JRE:
Default (1.8 - project SDK)
Tomcat Server Settings
HTTP port:
8000
接口
Deploy applications configured in Tomcat instance
HTTPs port:
Preserve sessions across restarts and redeploys
JMX port:
1099
AJP port:
Before launch: 2 tasks
Warning: No artifacts marked for deployment
Fix
OK
Cancel
Apply

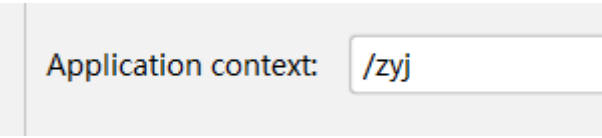
部署web项目


Name:
Tomcat 8.5.782
Store as project file
Server
Deployment
Logs
Code Coverage
Startup/Connection
部署
Deploy at the server startup
Artifact...
External Source...
web项目
Nothing to deploy
Before launch: 2 tasks
Warning: No artifacts marked for deployment
Fix
OK
Cancel
Apply

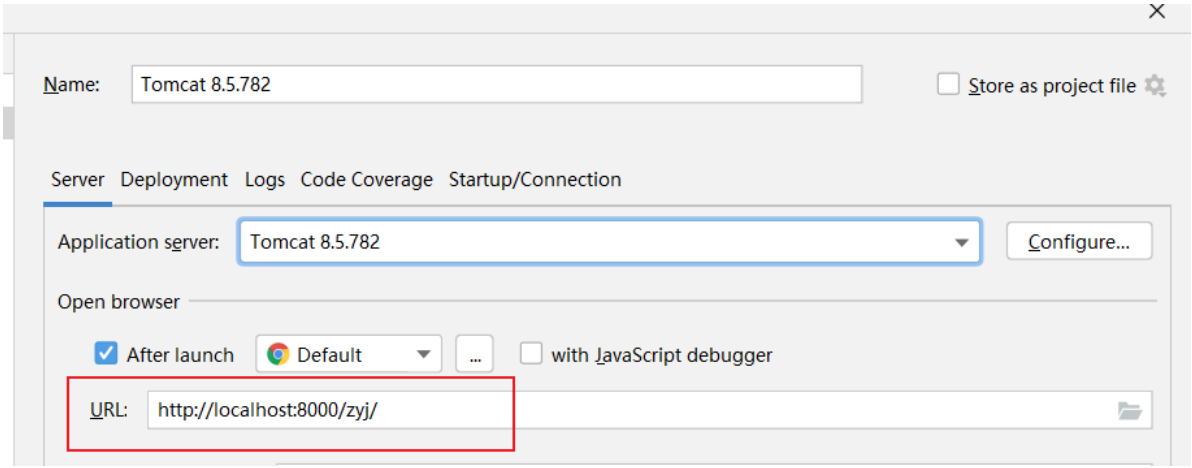
注意部署后的网址



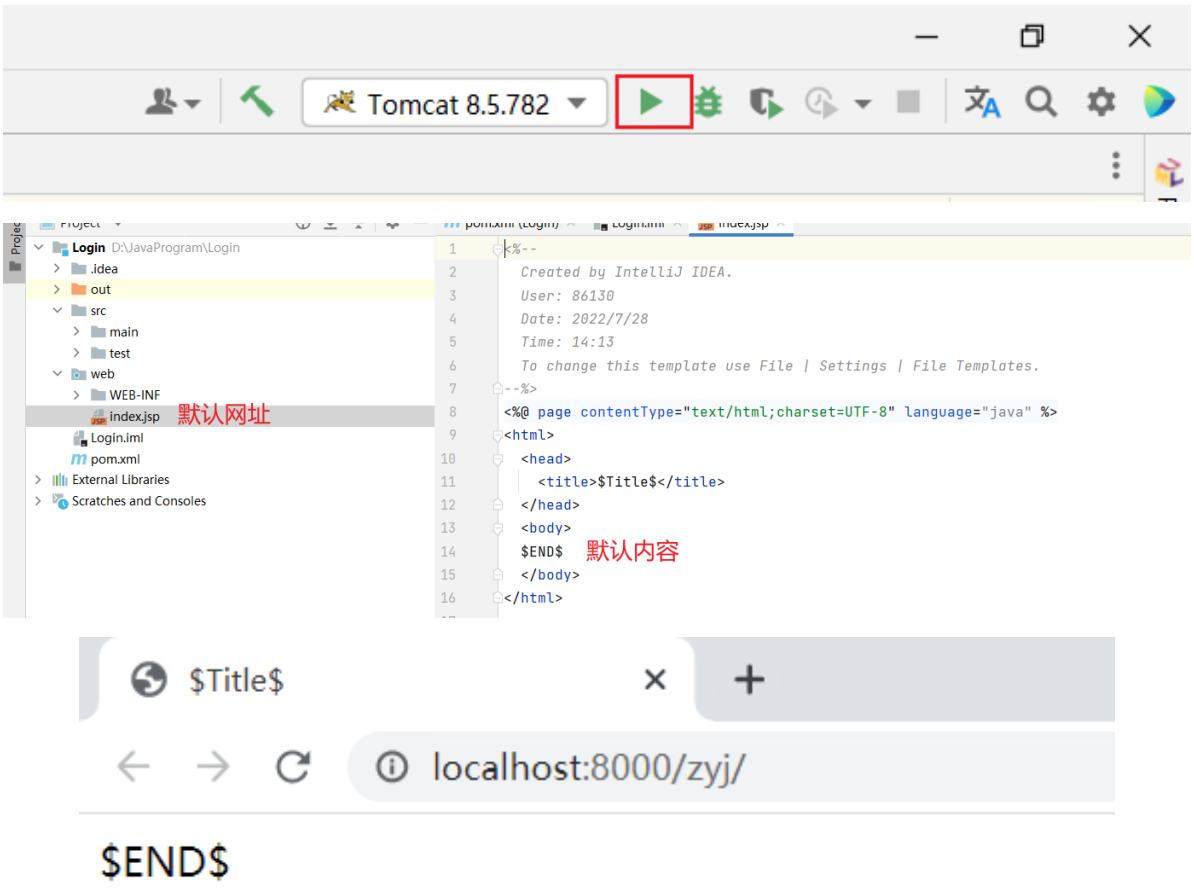
最终网址出现在这里



这里他自己会改变



运行后成功



现在我们写一下简易的登录

在index.jsp中写

```
<%--
  Created by IntelliJ IDEA.
  User: 86130
  Date: 2022/7/28
  Time: 14:13
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>$Title$</title>
  </head>
  <body>
  <form method="post" action="">
      <input type="text" name="id">
      <input type="text" name="pwd">
      <input type="submit" name="login">
  </form>
  </body>
</html>
```

引入servlet依赖

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
</dependencies>
```

创建servlet

```
import javax.servlet.http.HttpServlet;
@WebServlet("/login")
public class text extends HttpServlet {

}
```

联系servlet和网站

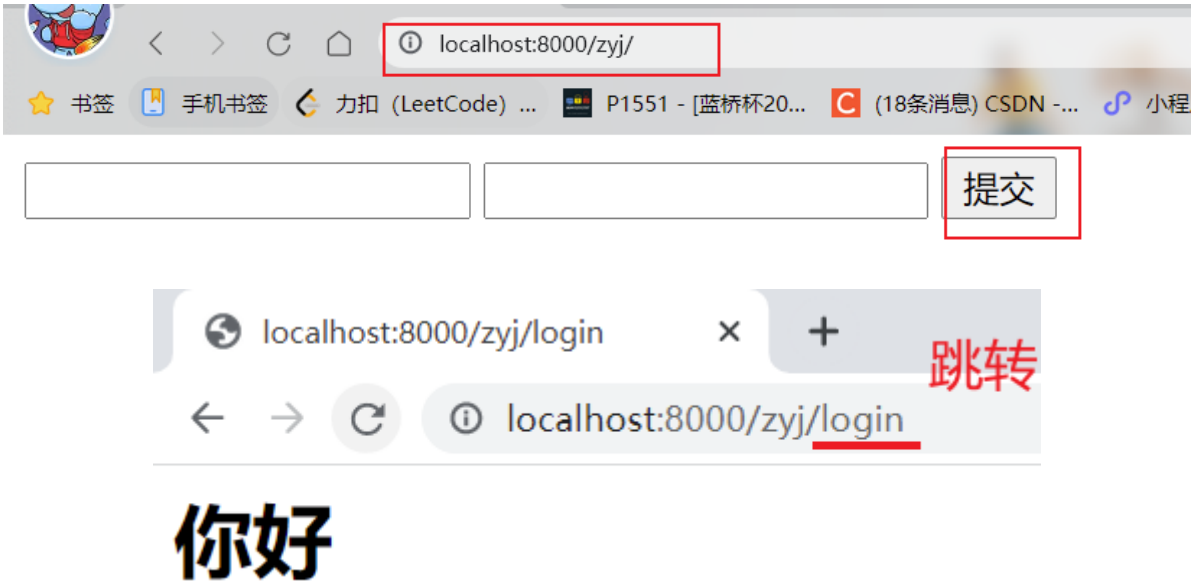
利用注解让servlet和网站联系起来处理网站信息，还有一种方式不是写注解是写xml文件！！！！

@WebServlet("/login")，注解一定要以横杆开头

```
<%--
  Created by IntelliJ IDEA.
  User: 86130
  Date: 2022/7/28
  Time: 14:13
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>$Title$</title>
  </head>
  <body>
  <form method="post" action="login">
      <input type="text" name="id">
      <input type="text" name="pwd">
      <input type="submit" name="login">
  </form>
  </body>
</html>
```

响应

```java
@WebServlet("/login")
public class text extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        PrintWriter writer = resp.getWriter();
        //设置服务端编码
        resp.setCharacterEncoding("UTF-8");
        //设置客户端编码格式和响应MIME类型
        resp.setHeader("content-type","text/html;charset=UTF-8");
        //响应数据打出个你好
        writer.write("<h1>你好<h1>");
    }
}
```



注意跳转了！！！

流程

打包

打包方式是war包，跟web有关是war包，要部署到服务器的

pom.xml (SpringMVC)

```xml
<?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 htt
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>SpringMVC</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
```

引入依赖

```xml
<?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</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <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>1.2.11</version>
        </dependency>
        <!--        ServletAPI-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
        </dependency>
        <!--            spring和thymeleaf整合包-->
        <dependency>
            <groupId>org.thymeleaf</groupId>
```

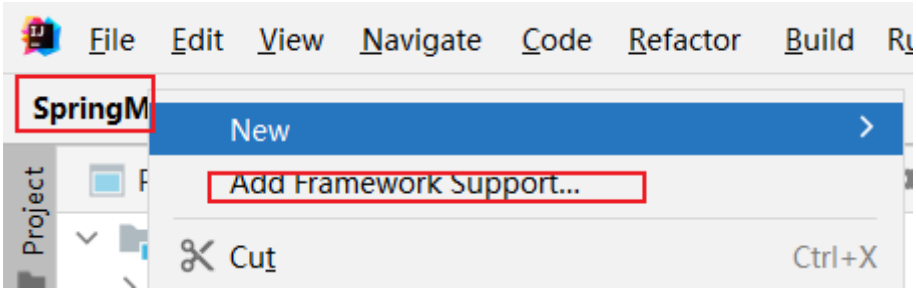
```
            <artifactId>thymeleaf-spring5</artifactId>
            <version>3.0.9.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

注意！！！不能加的区域

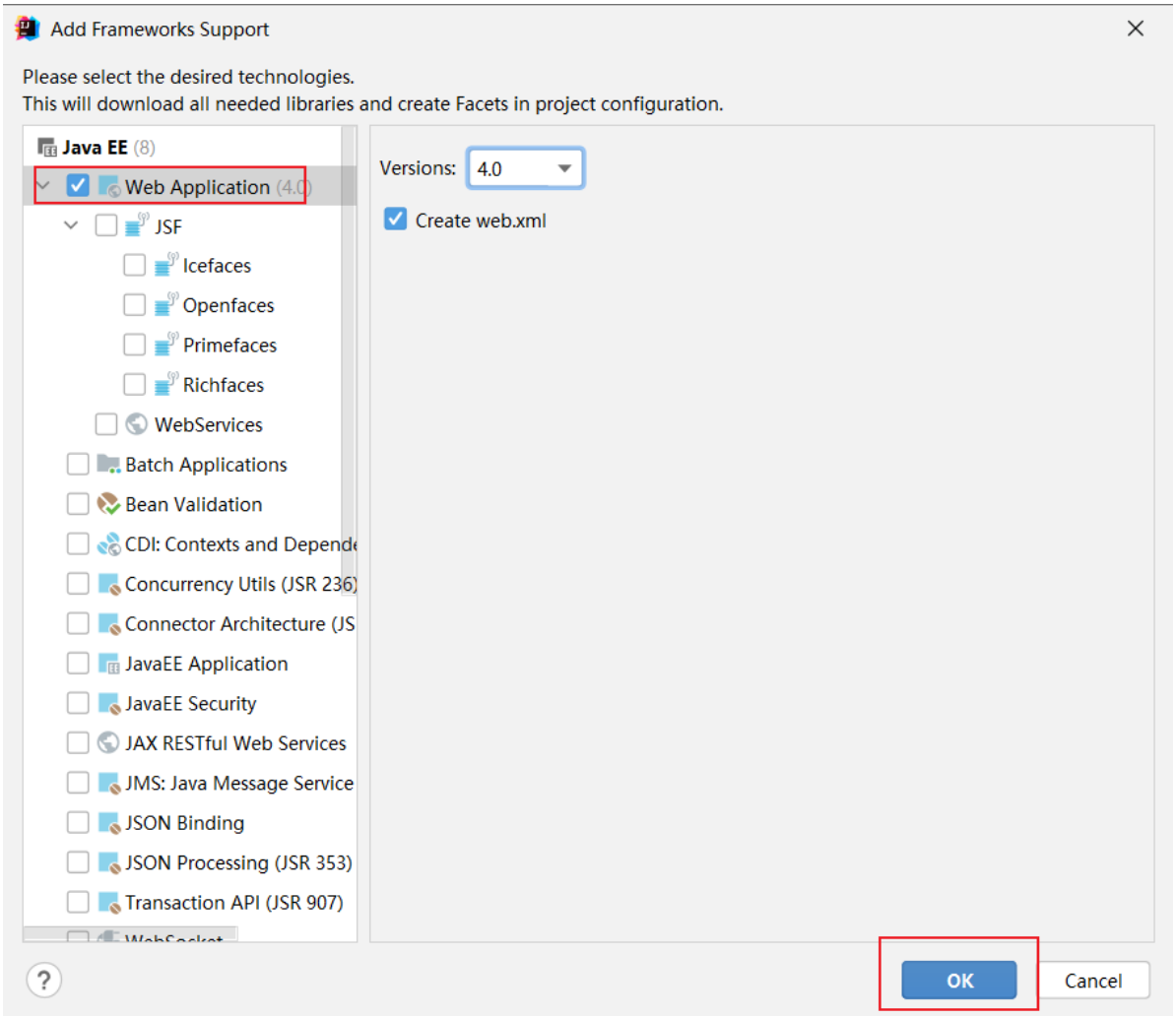
```
<!--        ServletAPI-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <!--            只在maven编译测试阶段生效，放到服务器时候服务器自己提供了-->
    <scope>provided</scope>
</dependency>
```

创建Web工程的选项

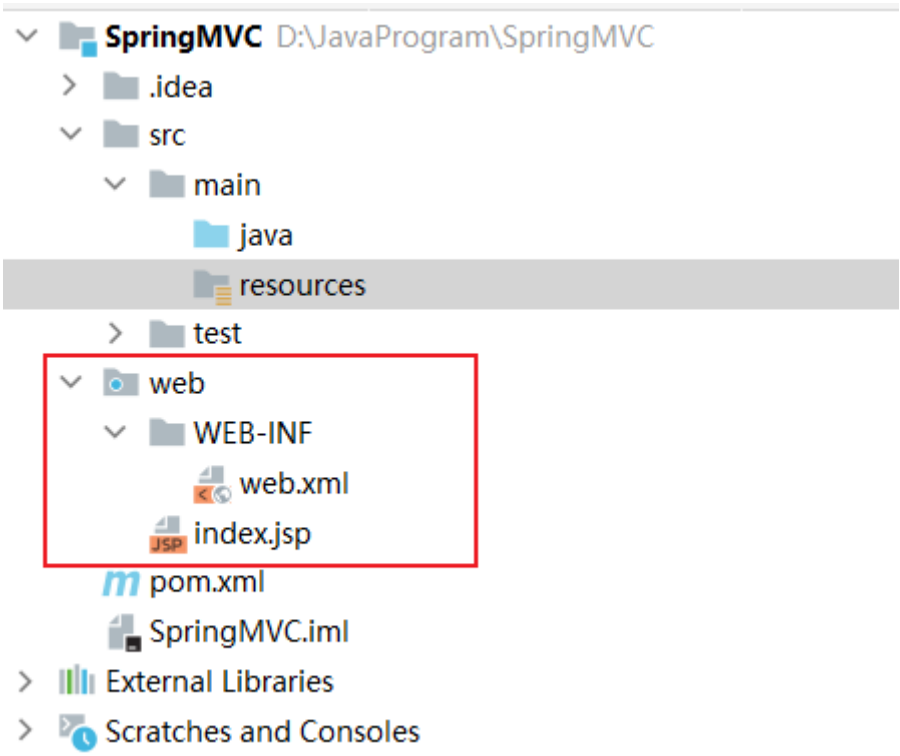
添加框架



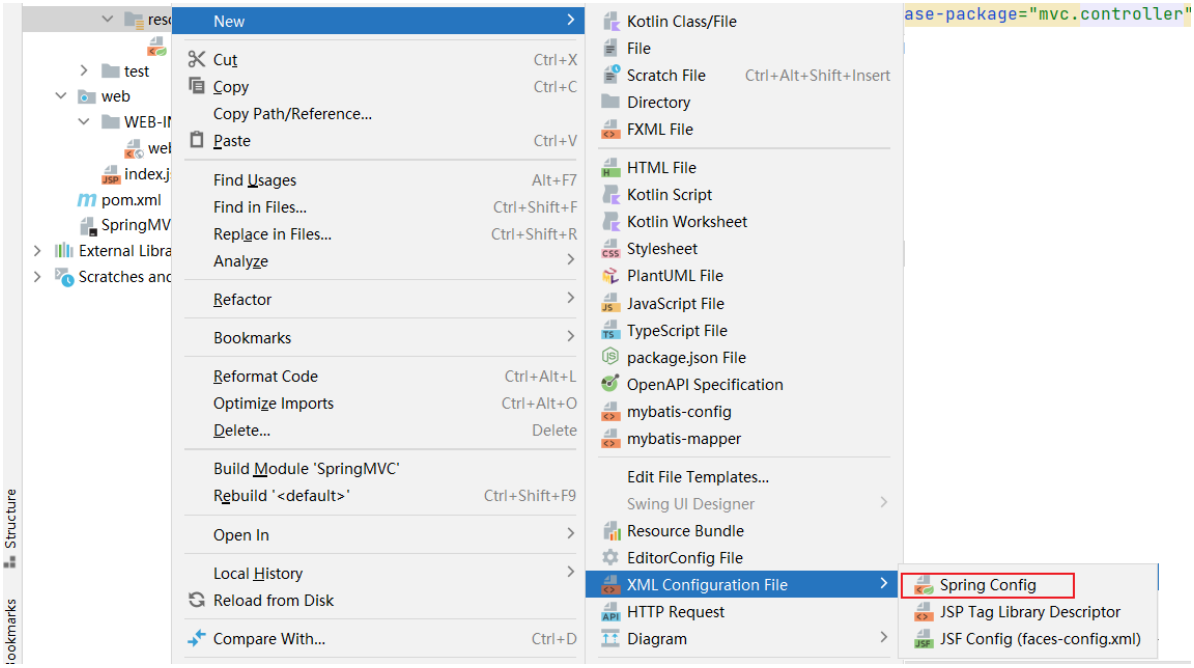
添加web

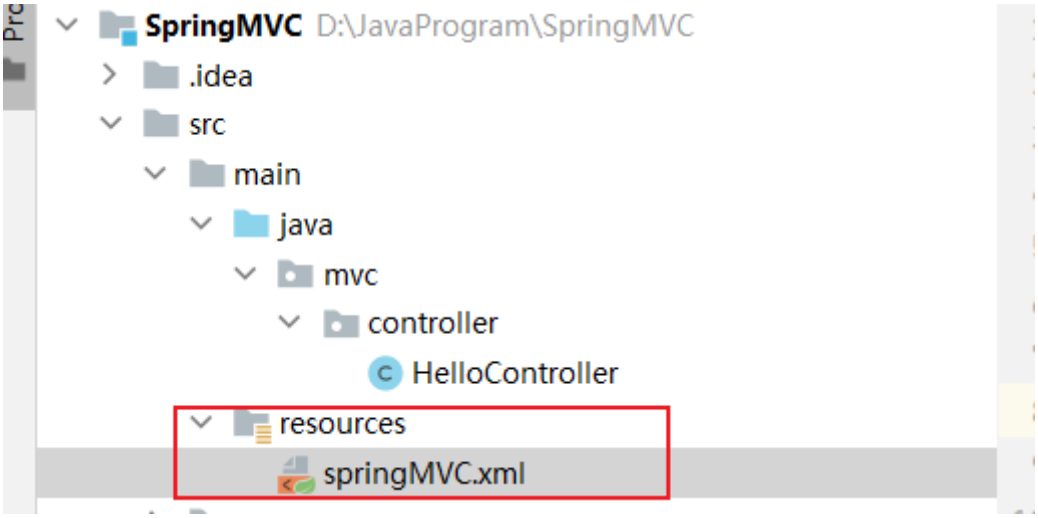


则有



配置springmvc.xml





开启组件扫描

```
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
http://www.springframework.org/schema/context/spring-context.xsd">
<!--     扫描组件-->
<context:component-scan base-package="mvc.controller"></context:component-scan>
```

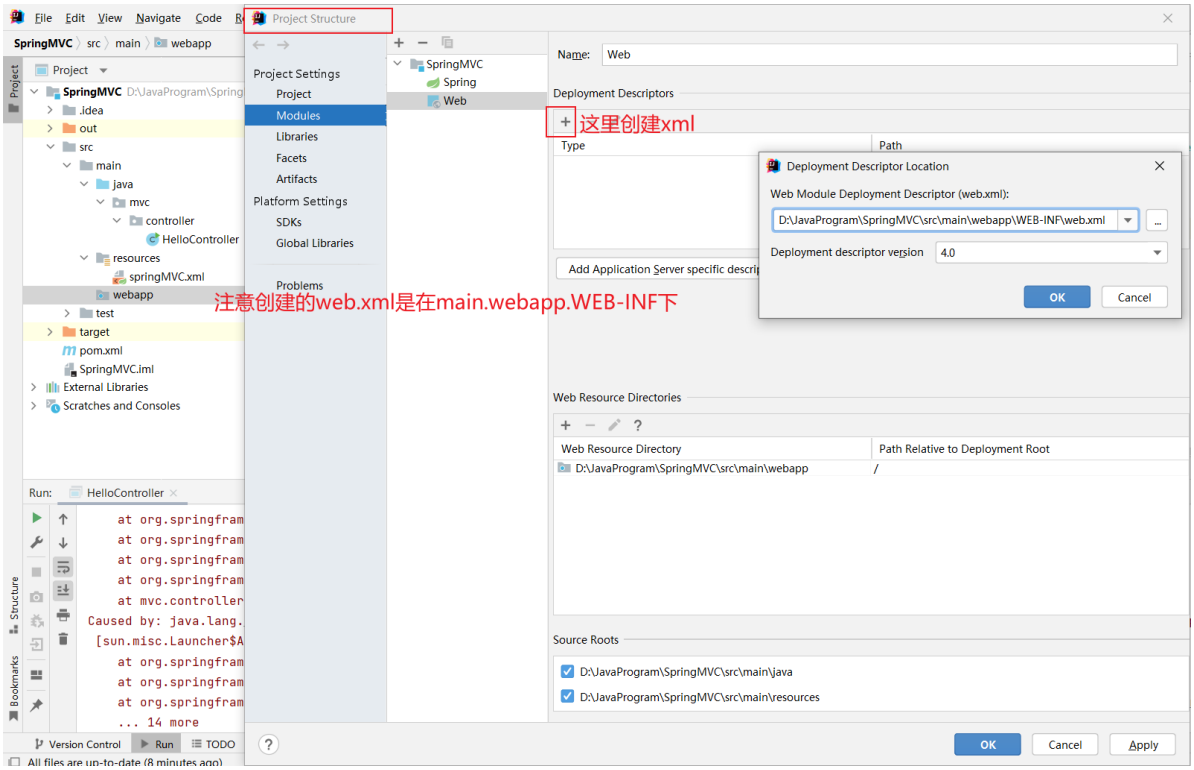
创建HelloController的bean

```
package mvc.controller;

import org.springframework.stereotype.Controller;

@Controller
public class HelloController {
}
```

配置Thymeleaf视图解析器bean

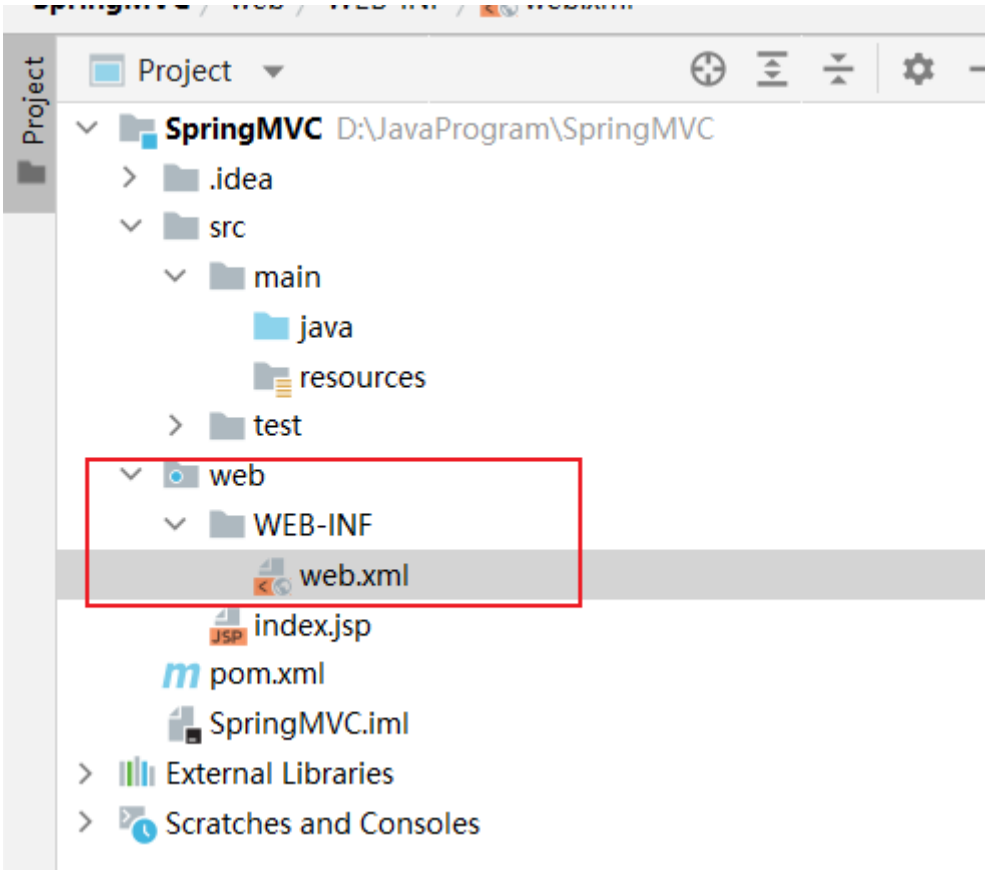
```
<!--    配置Thymeleaf视图解析器-->
    <bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
<!--        视图解析器优先级-->
        <property name="order" value="1"/>
<!--        编码-->
        <property name="characterEncoding" value="UTF-8"/>
<!--        模板-->
        <property name="templateEngine">
            <bean class="org.thymeleaf.spring5.SpringTemplateEngine">
                <property name="templateResolver">
                    <bean
class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">
<!--                        视图前缀-->
                        <property name="prefix" value="/WEB-INF/templates/"/>
<!--                        视图后缀-->
                        <property name="suffix" value=".html"></property>
<!--                        模板模型-->
                        <property name="templateMode" value="HTML5"></property>
<!--                        编码-->
                        <property name="characterEncoding" value="UTF-8"/>
                    </bean>
                </property>
            </bean>
        </property>
  </bean>
```

注意web.xml是这里创建，以下是错误的！



配置web.xml，这是个坑

默认是在这里配置，配置到这里是错的！！！




```
<?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">
```

```
<!--
等同于new test extends HttpServlet
         <servlet>注册一个Servlet
         <servlet-name>：用于指定一个Servlet名称
         <servlet-class>：用于指定Servlet程序所在的路径
        </servlet>
-->
<!--    配置SpringMVC的前端控制器，对浏览器发送的请求进行统一管理-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
<!--
等同于@WebServlet（"url-pattern的属性值"）
            <servlet-mapping> 用于映射一个已注册的Servlet的对外访问路径
          <servlet-name>用于指定访问路径的Servlet名称</servlet-name>
          <ul-pattern>用于指定Servlet的访问路径</ul-pattern>
        </servlet-mapping>
-->
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
<!--         该请求可以是/login或者.html或者.js或.css方式的请求路径！！！但不可以是.jsp的请
求路径，如-->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

为了让springmvc知道web.xml则要求如下配置，很重要！

首先在web.xml中标签servlet写到

```
<!--         配置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>
```

web.xml

```
4           xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-a
5           version="4.0">
6   <!--
7   等同于new test extends HttpServlet
8            <servlet>注册一个Servlet
9           <servlet-name>：用于指定一个Servlet名称
10          <servlet-class>：用于指定Servlet程序所在的路径
11         </servlet>
12  -->
13  <!--    配置SpringMVC的前端控制器，对浏览器发送的请求进行统一管理-->
14      <servlet>
15          <servlet-name>SpringMVC</servlet-name>
16          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
17  <!--        配置SpringMVC配置文件的位置和名称-->
18          <init-param>
19              <param-name>contextConfigLocation</param-name>
20              <param-value>classpath:springMVC.xml</param-value>
21          </init-param>
22      </servlet>
```

重点

1、如果 < init-param> 元素存在并且通过其子元素配置了Spring MVC配置文件的路径，则应用程序在启动时会加载配置路径下的配置文件。
2、如果没有通过 < init-param> 元素配置，则应用程序会默认去WEB-INF目录下寻找以 servletName-servlet.xml 方式命名的配置文件，这里的 servletName指下面的 springmvc。

初始化前端控制器时间

不要等请求时候才初始化，启动服务器就初始化

```
<!--        将前端控制器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-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<!--        配置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>
<!--        将前端控制器DispatcherServlet的初始化时间提前到服务器启动时-->
        <load-on-startup>1</load-on-startup>
    </servlet>
```

创建网站

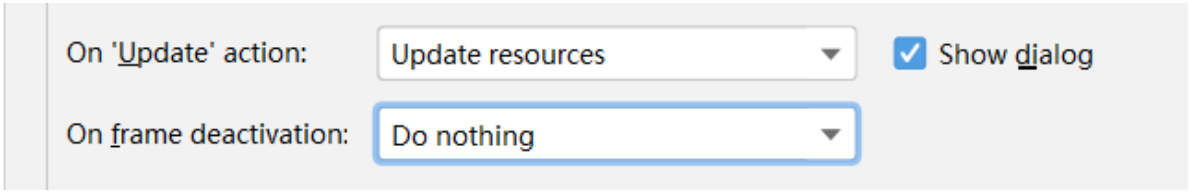
跟视图解析器联系到一起，此时还未起到作用

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

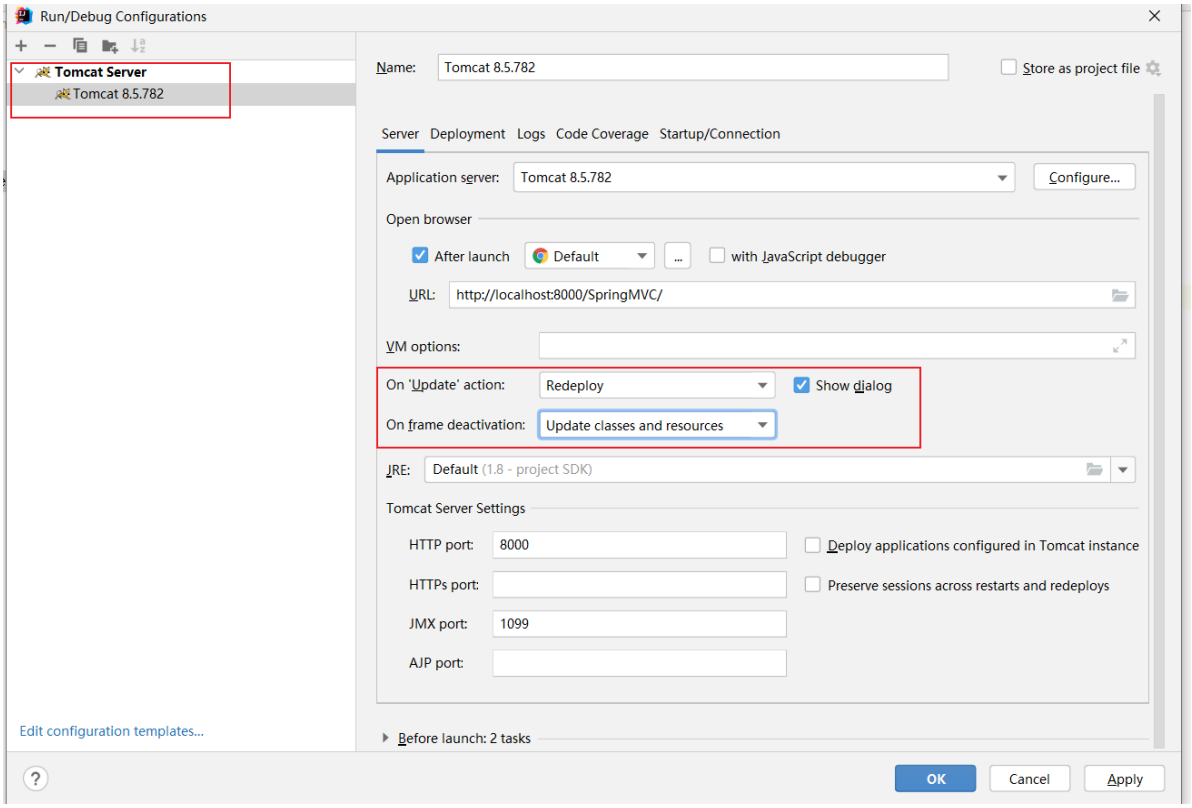
index.html

```
1   <!DOCTYPE html>
2   <html lang="en" xmlns:th="http://www.thymeleaf.org">
3   <head>
4       <meta charset="UTF-8">
5       <title>Title</title>
6   </head>
7   <body>
8
9   </body>
10  </html>
```

配置tomcat，运行时候更新资源



改为重新部署war包，更新war包的类和资源



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

```
package mvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloController {
    //让前端控制器调度
    @RequestMapping(value = "/")
    public String index()
    {
        //视图解析器会解析到这里获得
        //返回视图名称
        return "index";
    }
}
```

视图解析器解析网站的请求

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

一定是要th:href="@{/testRequestMapping}"

因为要让视图解析器解读为http://localhost：端口号/path/testRequestMapping

index.html

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

</body>
</html>
```

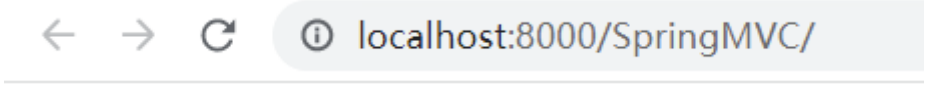
视图解析器解读请求映射

//视图解析器解析网站的请求后发送请求给前端控制器，前端控制器请求映射到该方法，最后视图解析器解析返回值，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。

```
@RequestMapping(value = "/testRequestMapping")
public String index()
{
  //视图解析器会解析到这里获得
  //返回视图名称
  return "index";
}
```

请求映射注解@RequestMapping

初始化打开服务器时候



```
http：//host[:port]/[path]
//在后端中 "/"的路径就是 http://localhost:8080/SpringMVC
    //  通过@RequestMapping注解，可以通过请求路径匹配要处理的具体的请求
    //  /表示的当前工程的上下文路径
    @RequestMapping("/")
    public String index(){
        return "index";
    }
    //在后端中 "/target"的路径就是 http://localhost:8080/SpringMVC/target
    @RequestMapping("/target")
    public String toTarget(){
        return "target";
    }
```

@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 RequestMappingController
{
    //此时请求映射所映射的请求的请求路径为：/test/testRequestMapping    才能访     //问到success    success页面是通过themleaf渲染过的
    @RequestMapping("/testRequestMapping")
    public String testRequestMapping()
    {
        return "success";
    }
}
```

value属性

通过请求的请求地址匹配请求映射，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

例如：@RequestMapping(　　value = {"/testRequestMapping", "/test"})

注意！！！@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，其它属性可不写，则不用再加一个条件来匹配请求映射

method属性

通过请求的请求方式（get或post）匹配请求映射,若请求方式不满足method属性，则浏览器报错405：Request method ‘POST’ not supported，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

可以通过枚举来得到RequestMethod里的枚举对应相对应的请求方式

@RequestMapping(　　value = {"/testRequestMapping", "/test"},　　method = {RequestMethod.GET, RequestMethod.POST})

params属性

“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”：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

```
@RequestMapping(        value = {"/testRequestMapping", "/test"}        ,method
= {RequestMethod.GET, RequestMethod.POST},params =
{"username","password!=123456"} )
```

则要求请求要带有参数username，并且参数password的值不能为123456

```
<a th:href="@{/testReqiestMapping?username=admit}"
或者
<a th:href="@{/testReqiestMapping(username='admit',password=123456)}"
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}
```

headers属性了解即可

@RequestMapping的派生注解CRUD

对于处理指定请求方式的控制器方法，SpringMVC中提供了

处理get请求的映射	@GetMapping	处理查
处理post请求的映射	@PostMapping	处理增
处理put请求的映射	@PutMapping	处理改
处理delete请求的映射	@DeleteMapping	处理删

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

SpringMVC支持ant风格的路径了解即可

@RequestMapping中的

？：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

针对的是@RequestMapping中的value可以让html的请求表示表示任意的单/0/多个字符。

**：表示任意的一层或多层目录

SpringMVC支持网址路径中的占位符（重点）

网址上

/deleteUser?id=1，用问号来隔开后属性=属性值&属性=属性值

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>
```

传过来的1和admit可以这样获取值

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

SpringMVC获取请求参数

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

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

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

通过控制器方法的形参获取请求参数

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

注意该方式要求http的属性与形参名必须相同！！！！
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

通过注解

```
@RequestMapping("/testParam")
public  String testParam(
@RequestParam
(value = "user_name",required = false defaultValue="default")   String name
,String password){
     System.out.println("username:"+username+",password:"+password);
    return  "success";
}
```

@RequestParam是将（请求参数)和(控制器方法的形参)创建映射关系

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

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@RequestHeader是将请求头信息和控制器方法的形参创建映射关系

通过实体类

通过反射调用的是无参构造，注意写无参！！！不然有参写了就没无参了。注意没有相关的值，对象默认为null，boolean值为false!!!

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

```
<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){ //接受到的实体对象会和本类的实体对象
    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'}
```

解决乱码

需要在DispatcherServlet获取参数前解决乱码，则服务器开启时就解决乱码。

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。已经提供但要注册！！！

web.xml中

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

@RequestHeader

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

@CookieValue

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

@RequestParam。客户端第一次请求的时候,服务器会叫客户端将sessionid给我,没有的话,服务器端会创建一个sessionid,响应回去的时候会将sessionid交给客户端,客户端以cookie的形式,将sessionid保存起来。

域对象

使用ServletAPI向request域对象共享数据

```
<a th:href="@{/RequestCope}">RequestCope</a>    index.html
```

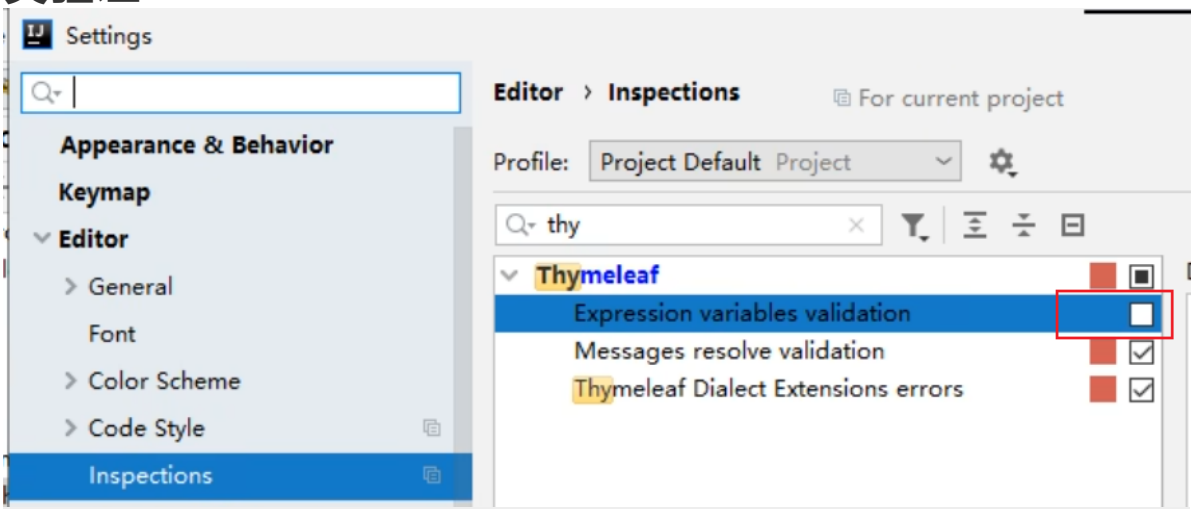
```
@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request){
    request.setAttribute("testScope", "hello,servletAPI");
    //转发！！！！
    return "success";
}
```

将testServletAPI请求映射对应的请求设置属性testScope属性值为hello,servletAPI后转发到succes网站这个servlet程序

则success网站标签段落会显示为hello,servletAPI

```
<p th:text="${testScope}"></p>    success.html
```

解决thymeleaf总是报错的问题，代码风格->检查->表达种类验证



使用ModelAndView向request域对象共享数据，推荐！

不管用什么方法最后都封装到ModelAndView

```
@RequestMapping("/testModelAndView")
public ModelAndView testModelAndView(){
    /**
     * ModelAndView有Model和View的功能default-servlet-handler
     * 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;
}
```

ModelAndView必须作为返回值

使用Model向request域对象共享数据

与上面的区别就是返回值不用返回对象了

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

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";
}
```

使用map向request域对象共享数据

键和值string和object

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

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
```

向session域共享数据，常用来保存用户登录状态

```
@RequestMapping("/testSession")
public String testSession(HttpSession session){
    session.setAttribute("testSessionScope", "hello,session");
    return "success";
}
网页获取值是得${session.testSessionScope},不是直接${testSessionScope}
```

向application域共享数据，不常用

```
@RequestMapping("/testApplication")
public String testApplication(HttpSession session){
    //会话获取ServletContext
    ServletContext application = session.getServletContext();
    application.setAttribute("testApplicationScope", "hello,application");
    return "success";
}
网页获取值是得${application.testSessionScope},不是直接${testSessionScope}
```

SpringMVC的视图

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"; }
```

只有ThymeleafView才能直接跳转到网页！地址栏不改变

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

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

注意即会跳转到另一个@RequestMapping所对应的方法！

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

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

注意即会跳转到另一个@RequestMapping所对应的方法！地址栏改变

转发和请求的区别

	请求转发	重定向
地址栏	不改变	改变
请求次数	一次	两次
对象	共享	不共享
行为	服务端行为	客户端行为
地址	当前站点的资源	任何地址

视图控制器view-controller 第四种方法

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

```
springmvc的配置文件中！！！！
*<!-- 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="/testView" view-name="success"></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 />
不开启这个注解驱动,所有的注解都会没有用,会默认被defaultservlet处理

```
<!--     将注解功能启动-->
    <mvc:annotation-driven/>
<!--当springmvc处理不了静态资源的时候才会启用   默认servlet-handler 来处理静态资源-->
<mvc:default-servlet-handler/>
```

RESTFul，写代码的格式

rest：representational state transfer 表现层资资源状态转移

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

HiddenHttpMethodFilter隐藏http方法过滤器

html中其实没有delete或者put方式，写错了或者没写默认是get方式！！！！！。则需要添加delete方式和put方式需要在springmvc配置文件配置过滤器

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); paramValue = request.getParameter(this.methodParam);

  通俗易懂编码肯定得先获取！！！！

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

```
<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>
```

流程：

DAO层----------

准备实体类

```java
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);
   }
}
```

control层----------

查询所有员工数据

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

如果当本次业务是对数据库，本地文件的增删改操作时，就需要使用请求重定向

为什么呢?
因为，这时如果我们使用请求转发（整个处理业务过程就只是一次请求），浏览就会记录我们这次请求（操作数据库），一旦客户端刷新页面，就会执行（操作数据库）的请求，这时很严重的bug，我们要避免，所有就要用到请求重定向;

因为我们每次操作完数据库，都会跳转到首页，或者数据展示的页面，所以这时我们使用请求重定向跳转到这个页面（实际是跳转到某个Servlet查询数据库中的记录，这个Servlet再跳转到数据展示的页面）；
此时浏览器记录的最后一次请求就是查询数据库中记录的请求了，这时我们刷新页面，也只是查询数据，不再是对数据库的增删改了;
总结：当我们修改数据库中的数据的之后，就需要使用请求重定向来避免上述的问题了;

当两个页面之间有数据传送用请求转发，没有就用请求重定向。

删除

```
@RequestMapping(value = "/employee/{id}", method = RequestMethod.DELETE)
public String deleteEmployee(@PathVariable("id") Integer id){
    employeeDao.delete(id);
    //操作成功与原来请求没有关系，要有新的请求
    return "redirect:/employee";
}
```

添加

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

更新

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

view层----------

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

查询所有员工数据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</a>)</th>
        </tr>
        //相当于for循环！！！！！！employee相当于i，${employeeList}是传过来的值
        <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="${employee.gender}"></td>
            <td>
                //不能"@{/employee/${value}}"
                <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
                <a th:href="@{'/employee/'+${employee.id}}">update</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

原有方式

"@{/testParam(username='admin',password=123456)"

"@{/testParam?username=admin&password=123456)"

restful风格有值则要

th:href="@{'/employee/'+${employee.id}}"

或者"@{/employpee/}+${value}"

删除

引入超链接表示删除但还需要请求方式

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>
```

连接表单和超链接

先导入资源vue.js，主要要重新打包到服务器否则没有用！！！

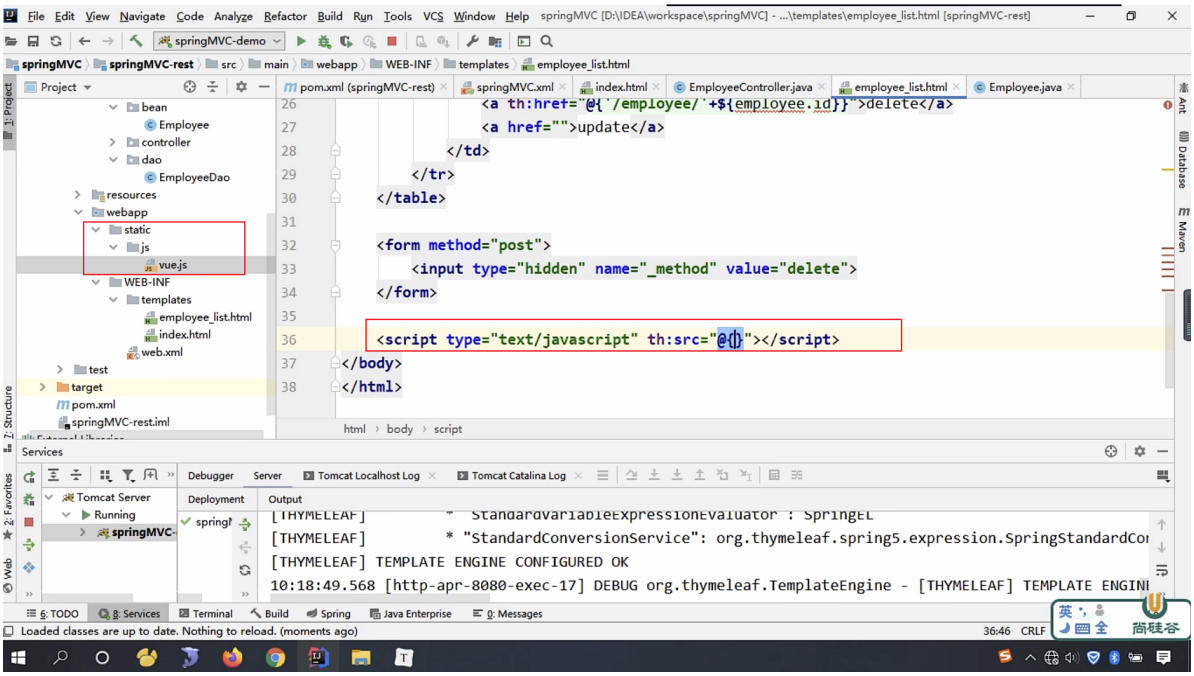
要在springmvc.xml配置文件开放对静态资源的访问，不然它找不到vue的事件处理

```
开放对静态资源的访问
<mvc:default-servlet-handler/>
SpringMVC中 <mvc: default-servlet-handler/>的作用

背景
优雅REST风格的资源URL不希望带 .html 或 .do 等后缀.由于早期的Spring MVC不能很好地处理静态资源，所以在web.xml中配置DispatcherServlet的请求映射，往往使用 *.do 、 *.xhtml等方式。这就决定了请求URL必须是一个带后缀的URL，而无法采用真正的REST风格的URL。

如果将DispatcherServlet请求映射配置为"/"，则Spring MVC将捕获Web容器所有的请求，包括静态资源的请求，Spring MVC会将它们当成一个普通请求处理，因此找不到对应处理器将导致错误。

设置web.xml中的DispatcherServlet的配置，使其可以捕获所有的请求

<servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
分析
如何让Spring框架能够捕获所有URL的请求，同时又将静态资源的请求转由Web容器处理，是可将DispatcherServlet的请求映射配置为"/"的前提。
```

方法：采用<mvc:default-servlet-handler />
<mvc:default-servlet-handler />

在springMVC-servlet.xml中配置<mvc:default-servlet-handler />后，会在Spring MVC上下文中定义一个org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler，它会像一个检查员，对进入DispatcherServlet的URL进行筛查，如果发现是静态资源的请求，就将该请求转由Web应用服务器默认的Servlet处理，如果不是静态资源的请求，才由DispatcherServlet继续处理。

理解：
如果把静态资源交给DispatcherServlet处理，那么就会经过HandlerMapping，HandlerAdapter的处理，但是由于静态资源文件的名字是带有.jsp或者.html后缀的，在解析的时候就会出问题。



接着html中引入vue.js

通过html中vue处理点击事件

```
    <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</a>)</th>
        </tr>
        //相当于for循环！！！！！！employee相当于i，${employeeList}是传过来的值
        <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="${employee.gender}"></td>
            <td>
                //不能"@{/employee/${value}}"
                <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
                <a th:href="@{'/employee/'+${employee.id}}">update</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

```
<!-- 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>
```

添加

查询所有员工数据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</a>)</th>
        </tr>
        //相当于for循环！！！！！！employee相当于i，${employeeList}是传过来的值
```

配置view-controller

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

创建employee_add.html

lastName:

email:

gender: male female

add

更新

```
    <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</a>)</th>
        </tr>
        //相当于for循环！！！！！！employee相当于i，${employeeList}是传过来的值
        <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="${employee.gender}"></td>
            <td>
                //不能"@{/employee/${value}}"
                <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
                <a th:href="@{'/employee/'+${employee.id}}">update</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

```
@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);
    //带着要修改的数据跳转到页面
    return "employee_update";
}
```

//更新！ lastName:

email:

gender: ○ male ○ female

update

th:value是为了先回显数据

@RequestBody请求体，不支持get

用户名:

密码:

提交

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

输出结果：

requestBody:username=userNameValue&password=passwordValue

@RequestEntity请求报文，用的不多

```
@RequestMapping("/testRequestEntity") public String
testRequestEntity(RequestEntity<String> requestEntity){
    //请求头
    System.out.println("requestHeader:"+requestEntity.getHeaders());
    //请求体
    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
```

@ResponseBody响应体，用的最多

@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(){
    return "success";
}
```

结果：浏览器页面显示success，注意返回的不是success.html！！！是响应一个网站写一个信息

@RestController注解，每个方法添加了@ResponseBody

@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！！！！！！！！

返回对象

@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@ResponseBody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(){
    return new User(1001,"admin","123456",23,"男");
}
```

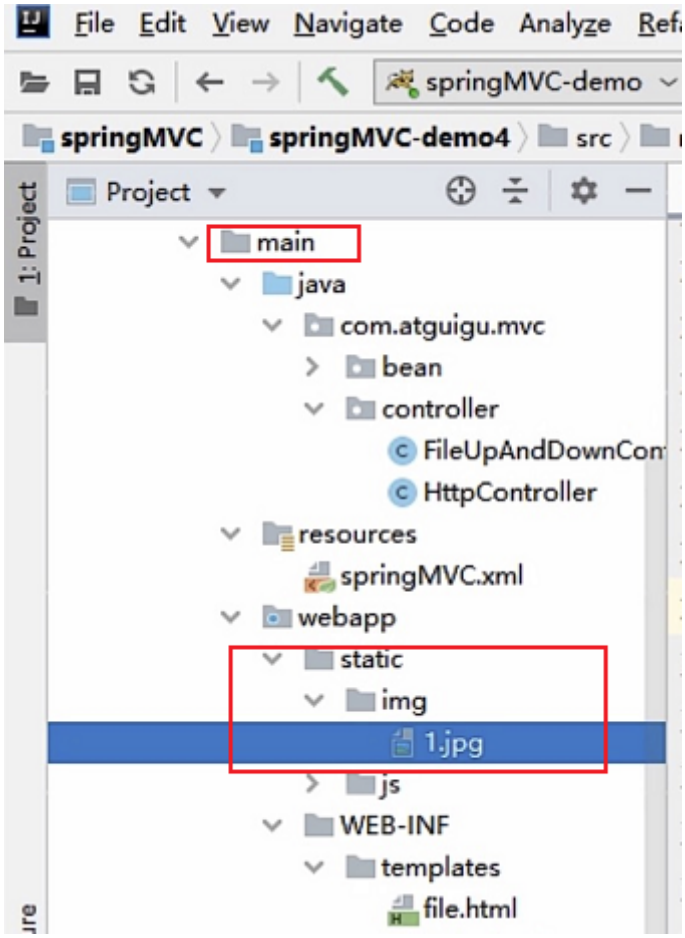
浏览器的页面中展示的结果：

{“id”:1001,“username”:“admin”,“password”:“123456”,“age”:23,“sex”:“男”}

axios处理点击事件不学

文件上传和下载

ResponseEntity响应报文



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

文件下载

```
@RequestMapping("/testDown")
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("/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
    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;
}
```

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

头像: 选择文件 未选择任何文件

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:

```
<!--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("/testUp")
public String testUp(MultipartFile photo, HttpSession session) throws
IOException {
    //获取上传的文件的文件名
    String fileName = photo.getOriginalFilename();
    //获取文件后缀
    String hzName = fileName.substring(fileName.lastIndexOf("."));
    //处理文件重名问题
    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);
    //判断路径是否存在
    if(!file.exists()){
        file.mkdir();
    }                                   //文件分隔符
    String finalPath = photoPath + File.separator + fileName;
    //实现上传功能
    photo.transferTo(new File(finalPath));
    return "success";
}
```

拦截器Intercepted

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，必须在SpringMVC的配置文件中进行配置。

```
//ctrl+o重写三个方法
public class Intercepter implements HandlerInterceptor {
    @Override
    //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！！！！
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        return HandlerInterceptor.super.preHandle(request, response, handler);
    }

    @Override
    //postHandle：控制器方法执行之后执行postHandle()
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler, modelAndView);
    }

    @Override
    //afterComplation：处理完视图和模型数据，渲染视图完毕之后执行afterComplation()
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        HandlerInterceptor.super.afterCompletion(request, response, handler, ex);
    }
}
```

```
<bean class="com.Intercepter"></bean>
 <ref bean="firstInterceptor"></ref>
<!-- 以上两种配置方式都是对DispatcherServlet所处理的所有的请求进行拦截 -->
    <mvc:interceptor>
        <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
    -->
```

多个拦截器的执行顺序
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

异常处理

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>
            <!--
                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
            -->
            //key是什么类型的异常，error是返回的网址
            <prop key="java.lang.ArithmeticException">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
    -->
    //属性ex获取值为异常信息，则在网站可获取该信息
    <property name="exceptionAttribute" value="ex"></property>
</bean>
```

即网站error.html可通过${ex}来获取值

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(ArithmeticException.class)
    //ex表示当前请求处理中出现的异常对象
    public String handleArithmeticException(Exception ex, Model model){
        model.addAttribute("ex", ex);
        return "error";
    }
}
```

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

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* }
```

创建初始化类，代替web.xml

```
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，针对乱码和多种请求方式的
     * @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};
    }
}
```

创建WebConfig配置类，代替SpringMVC的配置文件

```
@Configuration
//扫描组件
@ComponentScan("com.atguigu.mvc.controller")
//开启MVC注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    //使用默认的servlet处理静态资源
    @Override
```

```
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

    //配置文件上传解析器
    @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("/**");
    }

    //配置视图控制
    /*@Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }*/

    //配置异常映射
    /*@Override
    public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
        SimpleMappingExceptionResolver exceptionResolver = new
SimpleMappingExceptionResolver();
        Properties prop = new Properties();             //一个是异常，一个是网
址！！！
        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                  //属性ex的值为异常信息！！！！！
        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;
    }
}
```