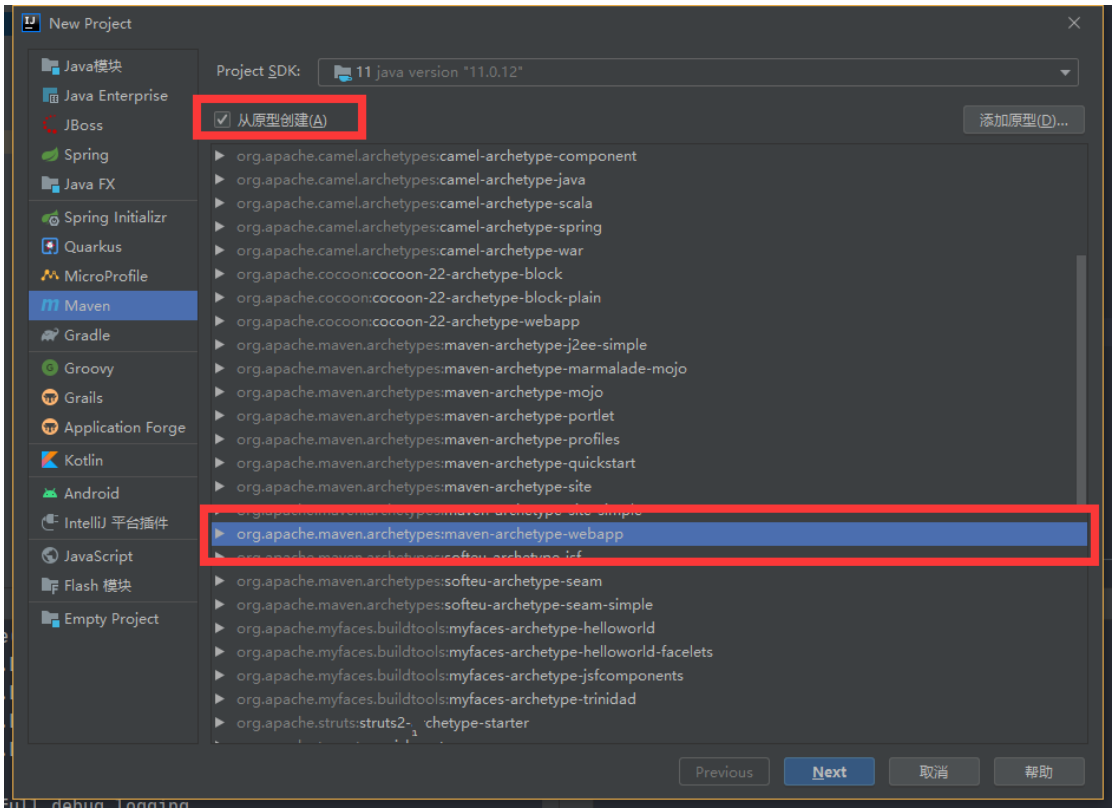
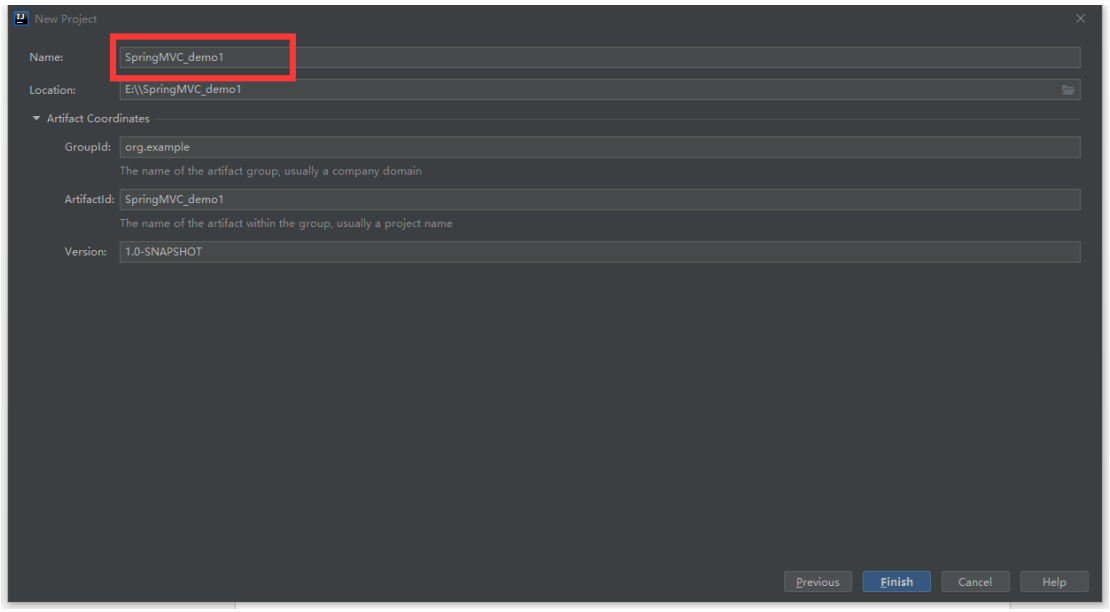
SpringMVC

创建 MavenWeb 工程

一、创建工程



二、设置工程名

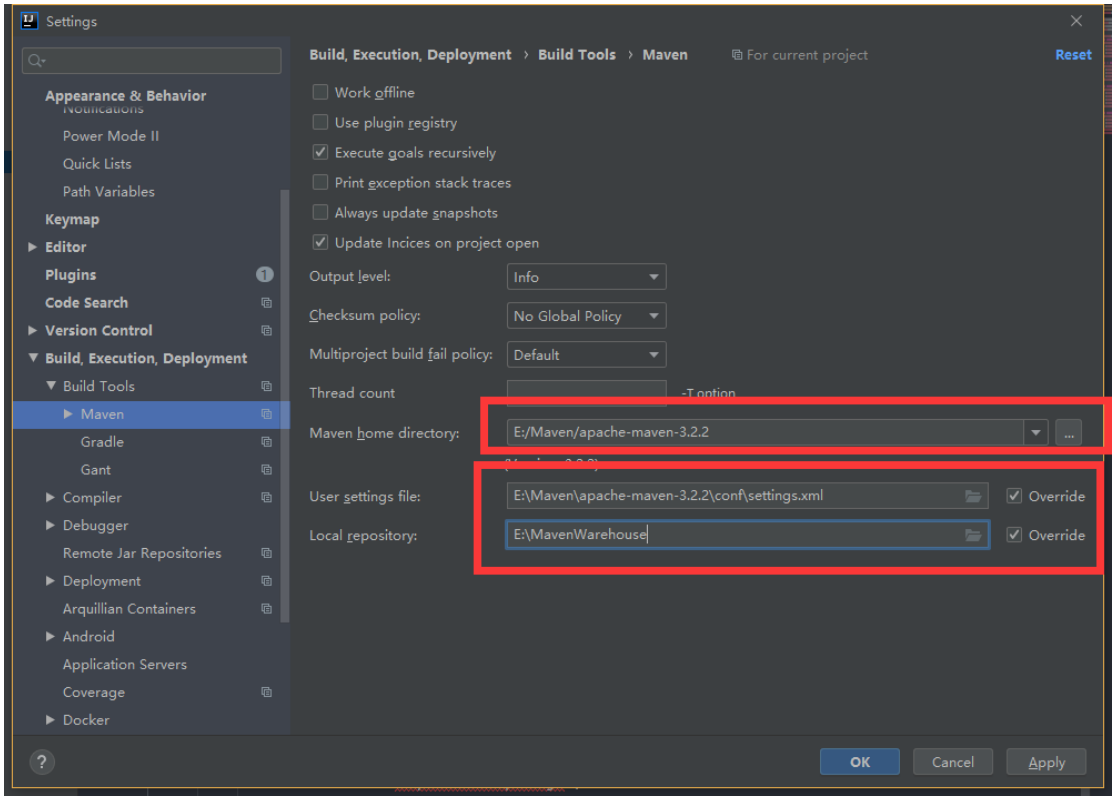


三、添加依赖

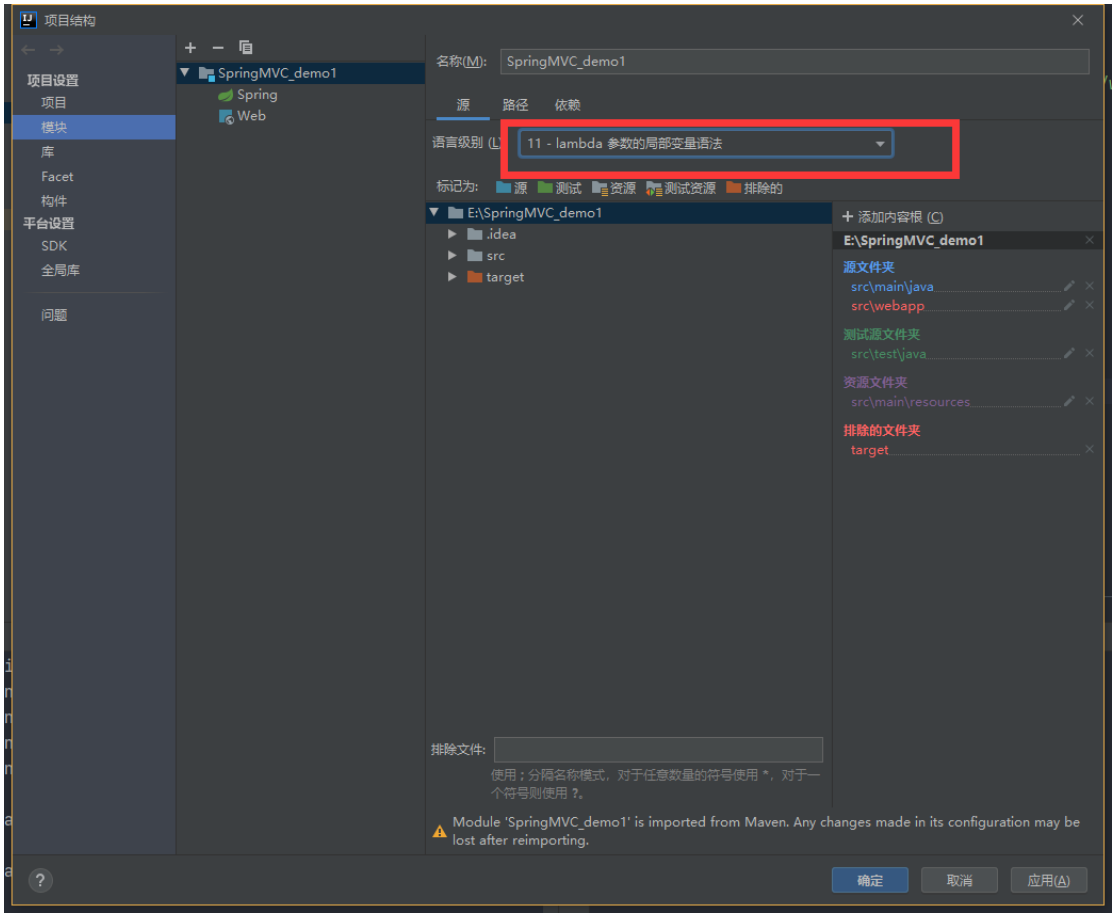
```
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.1</version>
    </dependency>

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

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

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

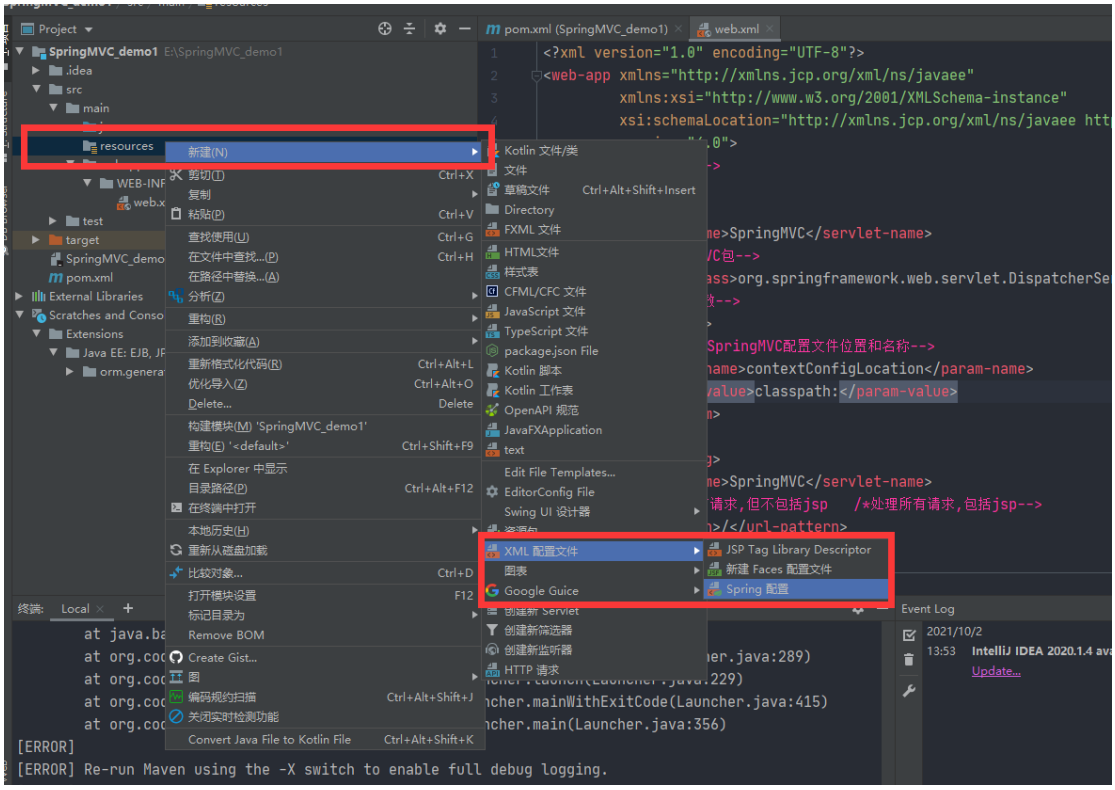
四、设置仓库

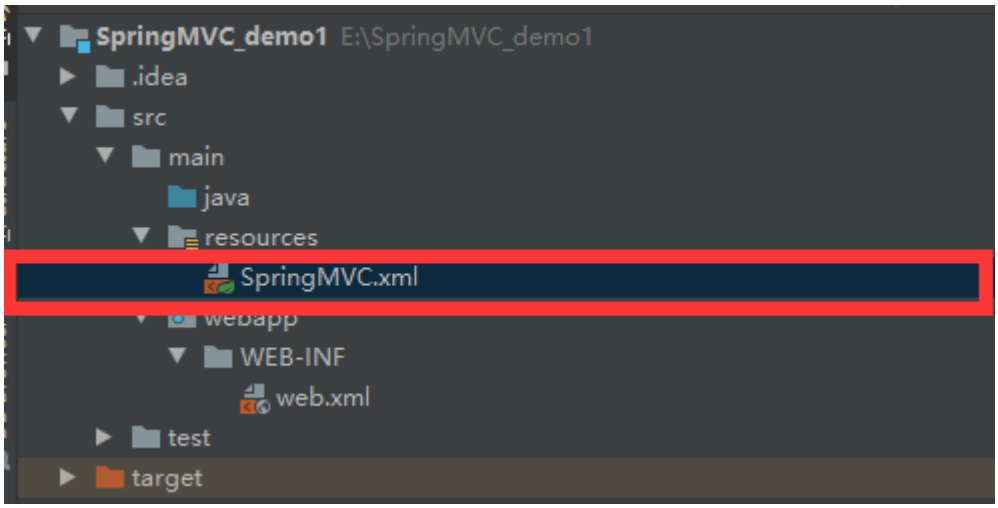


五、修改 Maven 版本


项目结构
项目设置
项目
模块
库
Facet
构件
平台设置
SDK
全局库
问题
SpringMVC_demo1
Spring
Web
名称(M): SpringMVC_demo1
源
路径
依赖
语言级别 (L)
11 - lambda 参数的局部变量语法
标记为: 源 测试 资源 测试资源 排除的
E:\SpringMVC_demo1
.idea
src
target
添加内容根 (C)
E:\SpringMVC_demo1
源文件夹
src\main\java
src\webapp
测试源文件夹
src\test\java
资源文件夹
src\main\resources
排除的文件夹
target
排除文件:
使用；分隔名称模式，对于任意数量的符号使用 *，对于一个符号则使用 ?。
Module 'SpringMVC_demo1' is imported from Maven. Any changes made in its configuration may be lost after reimporting.
确定
取消
应用(A)

六、创建 SpringMVC 配置文件





七、配置 web.xml

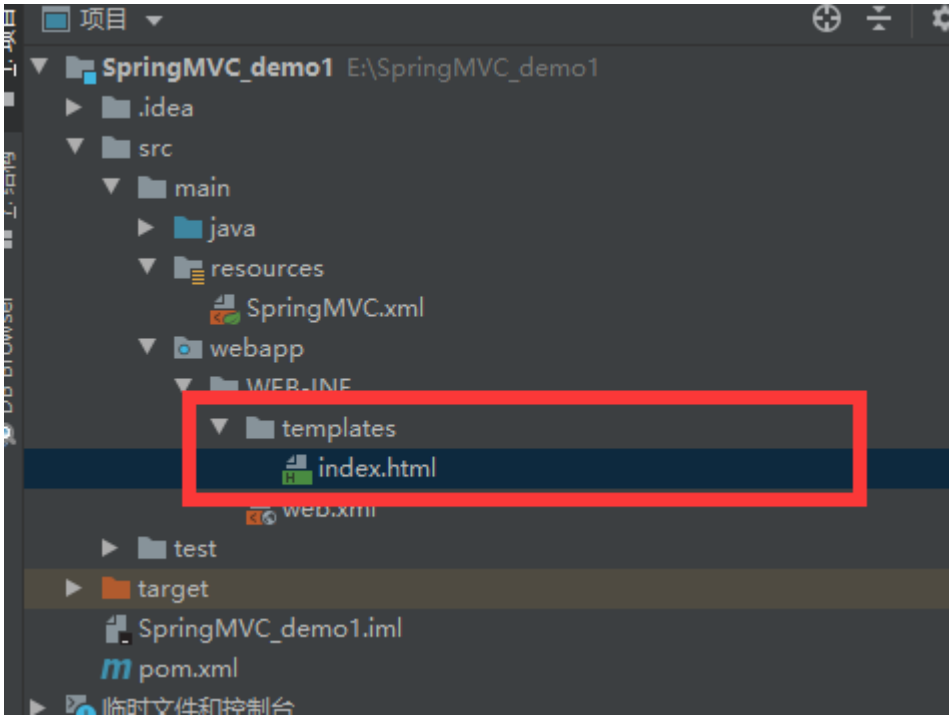
```
<!--处理全部请求-->
<servlet>
    <!--name-->
    <servlet-name>SpringMVC</servlet-name>
    <!--SpringMVC 包-->
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--初始化参数-->
```

```
        <init-param>
            <!--配置 SpringMVC 配置文件位置和名称-->
            <param-name>contextConfigLocation</param-name>
            <!--SpringMVC 的文件名-->
            <param-value>classpath:SpringMVC.xml</param-value>
        </init-param>
        <!--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>
        <!--/处理所有请求,但不包括 jsp     /*处理所有请求,包括 jsp-->
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

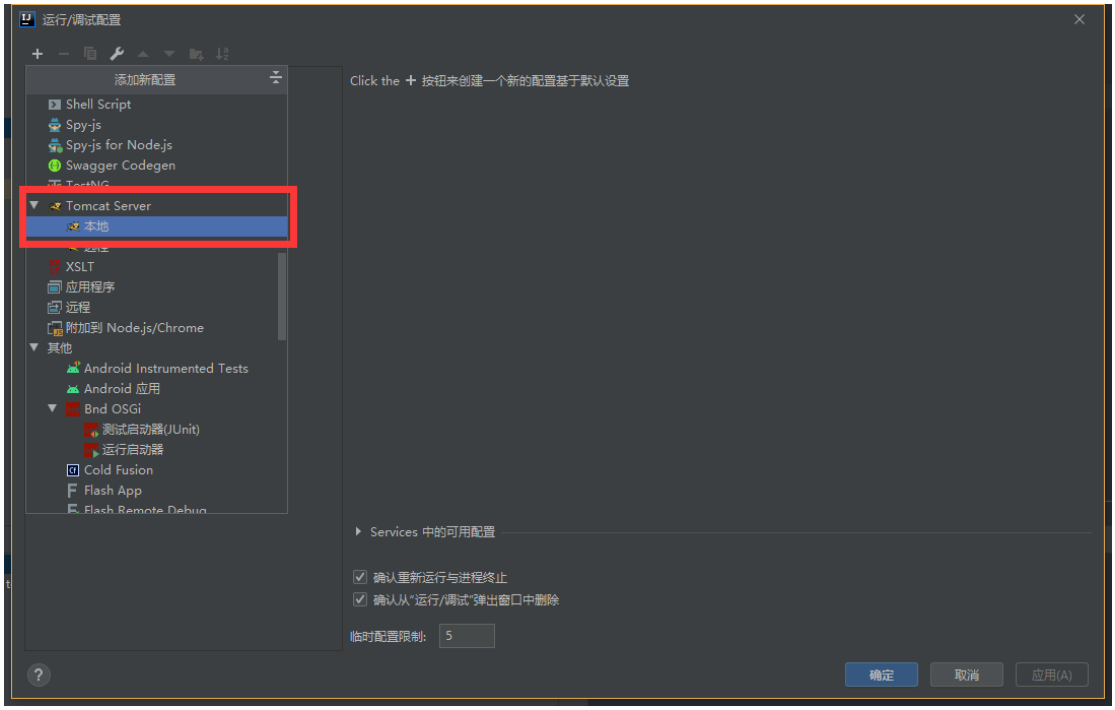
八、配置 SpringMVC.xml

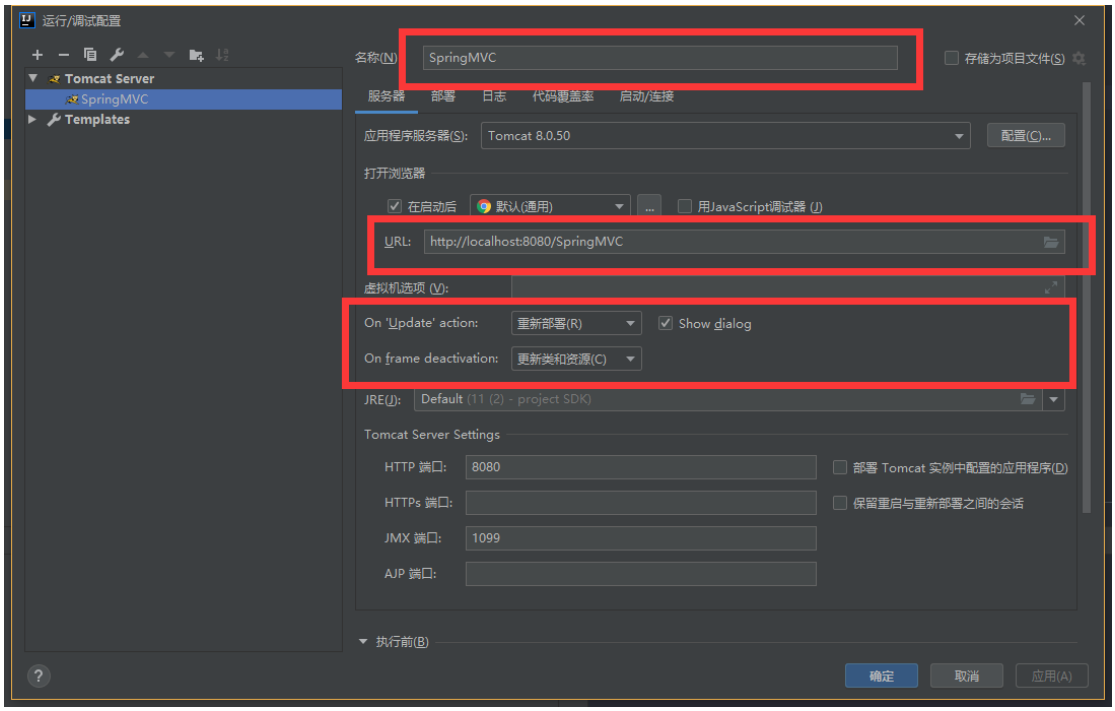
```
<!--开启扫描-->
<context:component-scan base-package="com.SpringMVC"/>
<!--  配置 Thymeleaf 视图解析器  -->
<bean id="viewResolver" class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
        <!--优先级-->
        <property name="order" value="1"/>
        <!--编码-->
        <property name="characterEncoding" value="UTF-8"/>
        <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 name="templateMode" value="HTML5"/>
                        <property name="characterEncoding" value="UTF-8" />
                    </bean>
                </property>
            </bean>
        </property>
</bean>
```

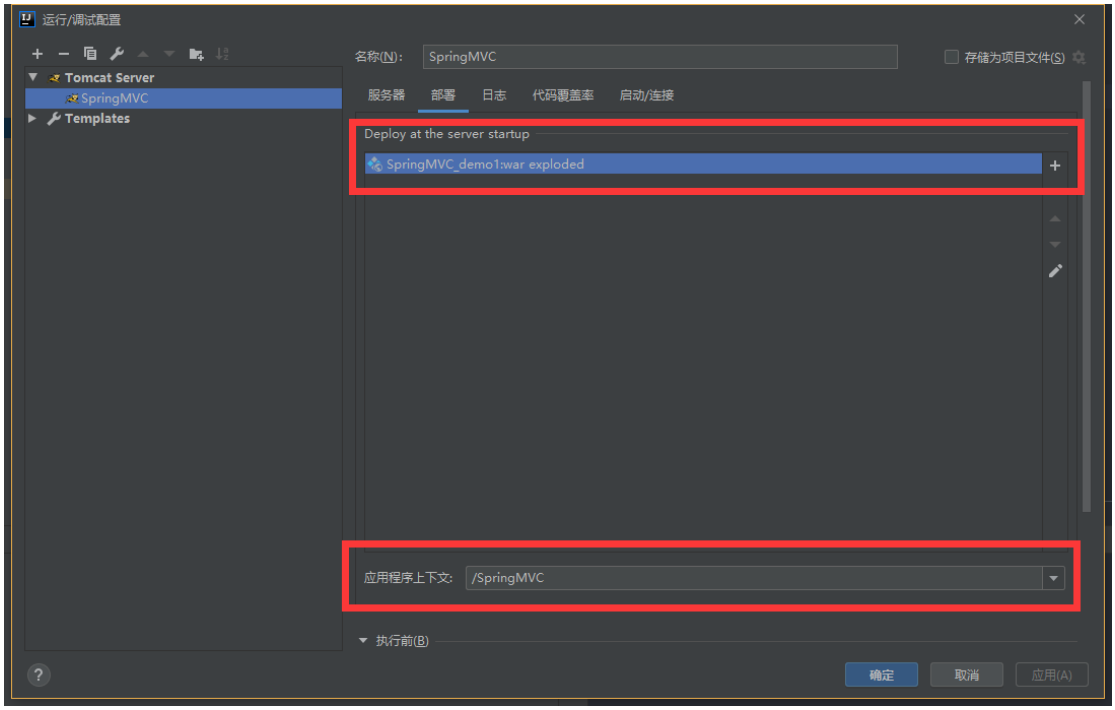
九、创建视图主页



十、配置 tomcat







十一、配置控制器重定向

```
Controller
public class HelloController {

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

```
    }
}
```

十二、访问指定页面

HTML 中添加 a 标签

```
<a th:href="@{/hello}">aaa</a>
```

控制前添加方法

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

十三、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。

注：

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

导入。

@RequestMapping 注解

一、@RequestMapping 功能

将请求处理的控制器方法关联起来,建立映射关系

SpringMVC 接受到指定请求,然后到映射关系中查找对应控制器方法处理请求

二、@RequestMapping 注解的位置

方法上

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

如果类上没有 @requestMapping 注解访问路径就是@{/hello}

```
<a th:href="@{/hello}">aaa</a>
```

类上

```
@Controller
@RequestMapping("/text")
public class HelloController {

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

此时访问的路径就需要加上类注解的 value

@{/text/hello}

```
<a th:href="@{/text/hello}">aaa</a>
```

三、@RequestMapping value 属性

(一)、value 属性是一个存放 Stirng 的数组,可以存放多个值,表示该映射可以匹配到多个请求地址对应的请求

(二)、value 值必须设置,至少通过请求地址匹配映射

```
@RequestMapping(
        value = {"/testRequestMapping", "/test"}
)
public String testRequestMapping(){
    return "success";
}
```

四、@RequestMapping method 属性

(一)、存储 RequestMethod[]数组,实现类装入了四个常量值

GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE

设置请求默认什么都支持,但如果设置了值比如说 RequestMethod.GET,RequestMethod.Post 设置了 2 个值后就支持 GET 和 POST 请求

(二)、如果出现错误为

405：Request method 'POST' not supported

就是请求方式不正确

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

(三)、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

```
@GetMapping("/testGetMapping")
public String testGetMapping(){
    return "success";
}
```

五、@RequestMapping params 属性（了解）

(一)、@RequestMapping 的 params 是匹配请求参数的,也是字符串数组类型,有四种形式,也可以组合使用

"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

(二)、html 访问的时候使用()连接参数

```
<a th:href="@{/test(username='admin',password=123456)">测试@RequestMapping的
params属性-->/test</a><br>
```

(三)、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}

六、@RequestMapping headers 属性（了解）

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

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

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

"!header"：要求请求映射所匹配的请求必须不能携带

header 请求头信息

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

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

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

七、@RequestMapping value 属性支持模糊匹配(正则)

?:表示任意的单个字符

@requestMapping(“/a?a/text”)　必须是 a(所有字符)a/text 好使

*:表示任意的 0 个或多个字符

@requestMapping(“/a*a/text”)　必须是 a(任意个字符)a/text 好使

**:表示任意的一层或多层目录

@requestMapping(“/**/text”)　可以是 /a/text 也可以是 /a/a/text

注意:在使用**的时候只能使用/**/xxx 的形式

八、@RequestMapping value 占位符

原始方式：/deleteUser?id=1

rest 方式：/deleteUser/1

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

示例:

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

@RequestMapping("/testRest/{id}/{username}")
public String text(@PathVariable("id") String id,
@PathVariable("username") String username)
{
    System.out.println("id:"+id+",username:"+username);
    return "success";
}
//最终输出的内容为-->id:1,username:admin
```

SpringMVC 获取请求参数

一、原生的 ServletAPI 获取请求

发送请求参数

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

接受请求参数

```
String username = request.getParameter("username");     //获取单个 name

String username = request.getParameterValues("username");     //获取多个 name
```

二、通过控制器的形参获取参数

请求 name 必须和形参 name 一样

发送请求参数

```
<form th:action="@{/testParam}" method="post">
    用户名：<input type="text" name="username"><br>
    密码：<input type="password" name="password"><br>
    爱好：<input type="checkbox" name="hobby" value="a">a
    <input type="checkbox" name="hobby" value="b">b
    <input type="checkbox" name="hobby" value="c">c<br>
    <input type="submit" value="">
</form>
```

接受请求参数

如果接受多个 name 值可以使用 String 或者 String[]

使用 String　参数之间用,隔开　　hobby:a,b,c

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

使用 String[] 参数用数组包括　hobby:[a,b,c]

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

三、使用注解形参的方式获取参数

使用@requestParam 注解形参

```
@RequestMapping("/testParam")
public String testParam(@RequestParam(value = "user_name",required = false,defaultValue = "1") String username
        , String password
        , String[] hobby){
    System.out.println("username:"+username+",password:"+password+"hobby:"+ Arrays.toString(hobby));
    return "success";
}
```

@requestParam 有 3 个参数

value = 表单提交的名字

此参数设置表单提交的 name 值赋值给添加注解的形参

required = false

此参数如果是 true 的话请求参数不能为空,为空抛异常。

此参数如果是 false 的话请求值可以为空,默认值为 null。

defaultValue = “默认值”

此参数为表单提交如果值为空的话使用设置的默认值

四、获取 HTTP 请求头信息

```
@RequestMapping("/testParam")
public String testParam(@RequestParam(value = "user_name",required = false,defaultValue = "1") String username
        , String password
        , String[] hobby
        , @RequestHeader(value = "Host",required = false,defaultValue = "1")String host){
    System.out.println("username:"+username+",password:"+password+"hobby:"+Arrays.toString(hobby));
    return "success";
}
```

获取请求头中 Host 信息并赋值给字符串,和@requestParam 一样的三个参数

五、获取 Cookie 信息

```
@RequestMapping("/testParam")
public String testParam(@RequestParam(value = "user_name",required = false,defaultValue = "1") String username
        , String password
        , String[] hobby
        , @CookieValue(value = "cookie",required = false,defaultValue = "1")String cookie){
    System.out.println("username:"+username+",password:"+password+"hobby:"+Arrays.toString(hobby));
    return "success";
}
```

六、使用 POJO 实体类获取参数

说明:

比如现在做一个注册功能,需要输入账户,密码,邮箱,手机号,这样需要和数据库进行交换,就需要创建一个 POJO 类存储用户数据,当浏览器发送的时候可以使用控制器获取参数直接存入 POJO 类中

实例:

发送请求:

用户名:

密　码:

性别： 男 女

年龄：

邮箱：

提交

接受请求:

在形参中添加需要接受的 POJO 类,name 名和属性名要一致,如果不一致的话值为 null

```
@RequestMapping("/testpojo")
public String testPOJO(User user){
    System.out.println(user);
    return "success";
}
// 最 终 结 果 -->User{id=null, username=' 张 三 ', password='123', age=23, sex='男', email='123@qq.com'}
```

六、解决参数乱码问题

控制器方法底层是调用 ServletAPI 此时参数已经获取到,再设置编码已经晚了,所以要在 ServletAPI 之前设置编码,此时就用到了过滤器

在 web.xml 中设置 MVC 包装过的过滤器

```
    <filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <!--设置请求编码为 UTF-8-->
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <!--设置响应编码为 UTF-8-->
        <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>
```

域对象共享数据

一、使用原生 ServletAPI 向 request 域对象共享数据

在控制器添加方法

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

二、使用 SpringMVC 中的 ModelAndView 类存储对象

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

三、使用形参 ModelAndView 形式共享数据

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

```
}
```

四、使用形参 Map 的形式共享数据

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

五、使用形参 ModelMap 形式共享数据

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

六、model、modelMap、Map 中间的关系

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 {}
public class ExtendedModelMap extends ModelMap
```

```
implements Model {}
public class BindingAwareModelMap extends
ExtendedModelMap {}
```

七、使用原生 ServletAPI 存放 Session 数据

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

八、使用 ServletContext 存放 application 数据

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

SpringMVC 的视图

一、 ThymeleafView

当控制器中方法返回的没有加前缀的都是 TheymeleafView

```
public String test (){
        return "success";
}
```

二、 转发视图 internalResourceView

当转发的时候转发的不是页面,而是另一个控制器 Servlet 的时候使用格式

return “forward:转发的控制器 Servlet”

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

三、 重定向视图 RedirectView

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

四、 视图控制器 view-controller

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

<!--

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"/>

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 />

五、 转发 JSP 页面

(一)、配置 SpringMVC.xml 为转发视图

```
<!--配置转发视图-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

(二)、配置 jsp 跳转控制器页面

```
<a href="${pageContext.request.contextPath}/textJSP"></a>
```

(三)、控制器转发

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

RestFul

一、 简介:

统一命名规范罢了

二、实现:

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

二、 实现发送 put 和 delete 请求

在控制器添加筛选器过滤 post 请求

```
<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>
```

然后表单为 post 请求,添加隐藏域为_method 为想要发送的请求

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

三、模拟登录

控制器

```
//根据 id 查询用户信息
@RequestMapping(value ="/textUser/{id}",method = RequestMethod.GET)
public String getUsers(@PathVariable("id")String id){
        System.out.println("根据 id 查询信息"+id);
        return "helloword";
}

//根据 id 修改用户信息
@RequestMapping(value = "/textUser/{id}",method = RequestMethod.PUT)
public String putUsers(@PathVariable("id")String id){
        System.out.println("根据 id 修改信息"+id);
        return "helloword";
}

//根据 id 删除用户信息
@RequestMapping(value = "/textUser/{id}",method = RequestMethod.DELETE)
public String delUsers(@PathVariable("id")String id){
        System.out.println("根据 id 删除信息"+id);
        return "helloword";
}

//查询所有用户信息
```

```java
@RequestMapping(value = "/textUser",method = RequestMethod.GET)
public String getUsers() {
        System.out.println("查询所有信息");
        return "helloword";
}

//添加用户
@RequestMapping(value = "/textUser",method = RequestMethod.POST)
public String postUsers(){
        System.out.println("添加用户信息");
        return "helloword";
}
```

HTML

根据用户 id 查询用户信息

```html
<a th:href="@{/textUser/1}">查询 id 为 1 的 id</a><br>
```

根据用户 id 修改用户信息

```html
<form th:action="@{textUser/1}" method="post">
        <input type="hidden" name="_method" value="PUT">
        用户名:<input type="text" name="username">
        密码:<input type="password" name="password">
        <input type="submit" value="修改">
</form>
```

根据用户 id 删除用户信息

```html
<form th:action="@{textUser/1}" method="post">
        <input type="hidden" name="_method" value="DELETE">
        用户名:<input type="text" name="username">
        密码:<input type="password" name="password">
        <input type="submit" value="删除">
</form>
```

查询所有用户信息

```html
<a th:href="@{/textUser}">查询所有信息</a><br>
```

添加用户信息

```html
<form th:action="@{textUser}" method="post">
        用户名:<input type="text" name="username">
        密码:<input type="password" name="password">
```

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

RestFul 案例

设置控制器访问主页

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

开启注解扫描

```
<!--开启扫描-->
<context:component-scan base-package="com"/>
```

创建员工信息类

```
private Integer id;
private String lastName;
private String email;
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() {
}

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

创建 DAO 类

```
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);
}
```

创建控制器

```
@Autowired
EmployeesDAO employeesDAO = new EmployeesDAO();

@RequestMapping(value = "/employee",method = RequestMethod.GET)
public String getAllEmployee(Model model){
        Collection<Employee> all = employeesDAO.getAll();
        model.addAttribute("employees",all);
        return "shoEmployees";
}

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

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

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

@RequestMapping(value = "/addEmployee",method = RequestMethod.PUT)
public String updateEmployee(Employee employee){
        employeesDAO.save(employee);
```

```
        return "redirect:/employee";
}
```

主页

```
<h1>首页</h1>
<a th:href="@{/employee}">查看所有员工信息</a>
```

员工信息展示

```
<table id="dataTable" border="1" cellspacing="0" cellpadding="0" style="text-align:
center;">
        <tr>
                <th colspan="5">employees</th>
        </tr>
        <tr>
                <th>id</th>
                <th>lastName</th>
                <th>email</th>
                <th>gender</th>
                <th>options<a th:href="employeeAdd">add</a> </th>
        </tr>
        <tr th:each="employee : ${employees}">
                <td th:text="${employee.id}"></td>
                <td th:text="${employee.lastName}"></td>
                <td th:text="${employee.email}"></td>
                <td th:text="${employee.gender}"></td>
                <td>
                        <a @click="deleteEmployee"
th:href="@{/delEmployee/}+${employee.id}" >删除</a>
                        <a th:href="@{/oneEmployee/}+${employee.id}" >修改</a>
                </td>
        </tr>
</table>

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

<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
<script type="text/javascript">
        var vue = new Vue({
                el:"#dataTable",
```

```
        methods:{
            deleteEmployee:function(event) {
                var delform = document.getElementById("delEmployee");
                //获取 a 标签的连接
                delform.action = event.target.href;
                //提交表单
                delform.submit();
                //取消超链接的默认行为
                event.preventDefault();
            }
        }
    })
</script>
```

添加用户信息

```
<form th:action="@{/addEmployee}" method="post">
    lastName:<input type="text" name="lastName" ><br>
    email:<input type="text" name="email"><br>
    gender:<input type="radio" name="gender" value="1">男
    <input type="radio" name="gender" value="0">女
    <input type="submit" value="添加">
</form>
```

修改用户信息

```
<form th:action="@{/addEmployee}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="hidden" name="id" th:value="${oneEmployee.id}">
    lastName:<input type="text" name="lastName"
th:value="${oneEmployee.lastName}"><br>
    email:<input type="text" name="email" th:value="${oneEmployee.email}"><br>
    gender:<input type="radio" name="gender" value="1"
th:field="${oneEmployee.gender}">男
    <input type="radio" name="gender" value="0" th:field="${oneEmployee.gender}">
女
    <input type="submit" value="修改">
</form>
```

HttpMessageConverter

一、简介:

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

HttpMessageConverter 提供了两个注解和两个类型：

@RequestBody	获取请求体
@ResponseBody	将响应转换为参数(常用)
RequestEntity	获取请求头或请求体
ResponseEntity	响应浏览器报文(常用)

二、@RequestBody

用户名： admin

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=admin&password=123456

三、 RequestEntity

用户名： admin

密　码： ******

提交

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

输出结果：requestHeader:[host:"localhost:8080", connection:"keep-alive", content-length:"27", cache-control:"max-age=0", sec-ch-ua:"" Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90"", sec-ch-ua-mobile:"?0", upgrade-insecure-requests:"1", origin:"http://localhost:8 080", 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

四、 原生的 response 响应请求

```
@RquestMapping(“/text/Response”)
public void testResponse(HttpservletResponse resp)
        resp.getWriter().pring(“hello,response”)
}
```

五、通过@responseBody 注解响应请求

```
@RquestMapping(“/text/Response”)
@RsponseBody
public void testResponse(HttpservletResponse resp)
        return “response”; //此时的 return 不是跳转页面,而是 servlet 响应浏览器的信息
}
```

五、 处理 json 数据

(一)、导入依赖

```
<!--json 的 jar 包-->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.1</version>
</dependency>
```

(二)、开启注解控制

```
<!--开启 xml 访问控制-->
<mvc:annotation-driven/>
```

(三)、创建控制器方法,返回的对象自动转换为 json 对象

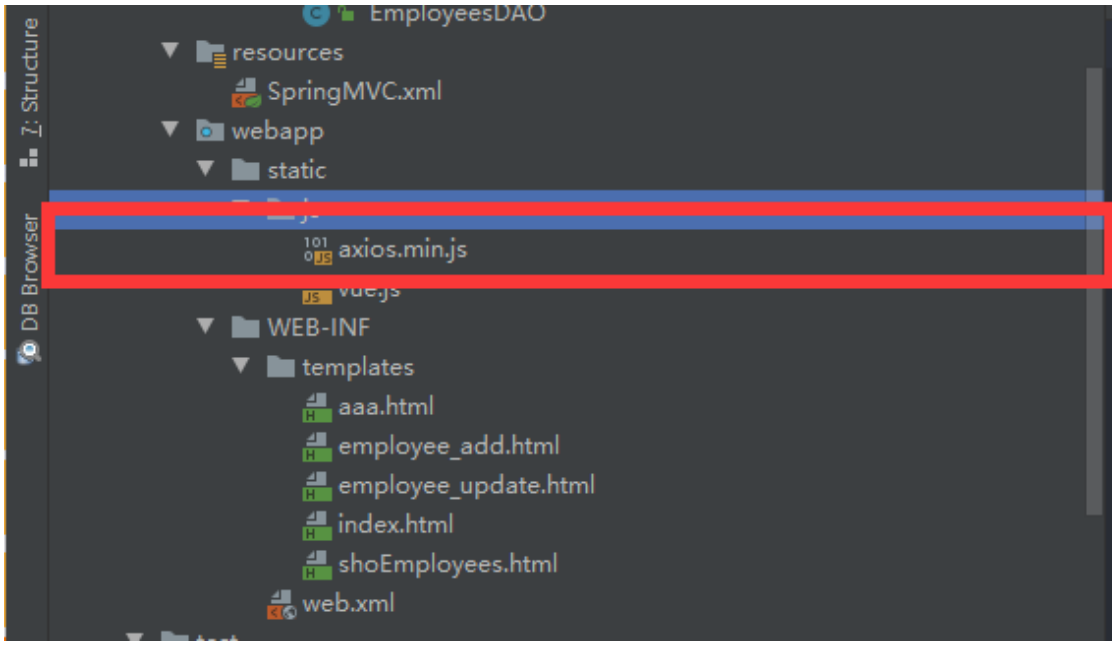
```
@RequestMapping("/testResponseUser")
@ResponseBody
public User testResponseUser(){
    return new User(1001,"admin","123456",23,"男");
}
```

(四)、输出结果

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

六、发送 AJAX 请求

添加 axios.min.js



网页设置点击超链接发送

```
<div id="app">
    <a th:href="@{/testAjax}" @click="testAjax">testAjax</a><br>
</div>
<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/static/js/axios.min.js}"></script>
<script type="text/javascript">
    var vue = new Vue({
        el:"#app",
        methods:{
            testAjax:function (event) {
                axios({
                    method:"post",
                    url:event.target.href,
```

```
                    params:{
                        username:"admin",
                        password:"123456"
                    }
                }).then(function (response) {
                    alert(response.data);
                });
                event.preventDefault();
            }
        }
    });
</script>
```

控制接接收 ajax 请求

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

七、@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

八、ResponseEntity(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,一般用于文件下载

文件上传和下载

一、 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    si.available 读取文件最大字节数并赋值
        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;
}
```

二、 文件上传

说明：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"

(一)、添加依赖

```
<dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
</dependency>
```

(二)、SpringMVC 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-->

id 必须为 multipartResolver

```
<bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.
CommonsMultipartResolver">
</bean>
```

(三)、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();
    //处理文件重名问题
    String hzName = fileName.substring(fileName.lastIndexOf("."));
```

```
        fileName = UUID.randomUUID().toString() + hzName;
        //获取服务器中 photo 目录的路径
        ServletContext servletContext = session.getServletContext();
        String photoPath = servletContext.getRealPath("photo");
        File file = new File(photoPath);
        if(!file.exists()){
            file.mkdir();
        }
        String finalPath = photoPath + File.separator + fileName;
        //实现上传功能
        photo.transferTo(new File(finalPath));
        return "success";
}
```

拦截器

filter 过滤器拦截浏览器到 Servlet 的请求,拦截器拦截 Servlet 到控制器的请求

一、 创建一个类实现 HandlerInterceptor 接口

```
public class FirstInterceptors implements HandlerInterceptor {

    //控制器方法执行之前 返回false为拦截,true为放行
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        return false;
    }

    //控制器方法执行之后
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)

    }

    //控制器方法视图加载完成之后
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws E

    }
}
```

二、 配置 SpringMV 拦截器

(一)、方式一:拦截所有请求

拦截器指向创建的实现类

```
<!--拦截器-->
<mvc:interceptors>
    <bean class="com.interceptors.FirstInterceptors"/>
</mvc:interceptors>
```

(二)、方式二:排除指定请求

```
<!--排除/目录-->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <mvc:exclude-mapping path="/"/>
        <bean class="com.interceptors.FirstInterceptors"/>
    </mvc:interceptor>
</mvc:interceptors>
```

(三)、多个拦截器执行顺序

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

2.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

一、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

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

```
<!--异常处理-->
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <!--当出现指定错误的时候跳转到 error 页面-->
            <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-->
    <property name="exceptionAttribute" value="ex"/>
</bean>
```

使用域共享

二、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{存储数组,可放多个异常错误}
    @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";
```

```
    }
}
```

SpringMVC 完全注解开发

一、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
     * @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};
    }
}
```

二、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();
            prop.setProperty("java.lang.ArithmeticException", "error");
            //设置异常映射
            exceptionResolver.setExceptionMappings(prop);
/           /设置共享异常信息的键
            exceptionResolver.setExceptionAttribute("ex");
            resolvers.add(exceptionResolver);
        }

        //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;
        }
}
```

SpringMVC 执行流程

一、 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

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

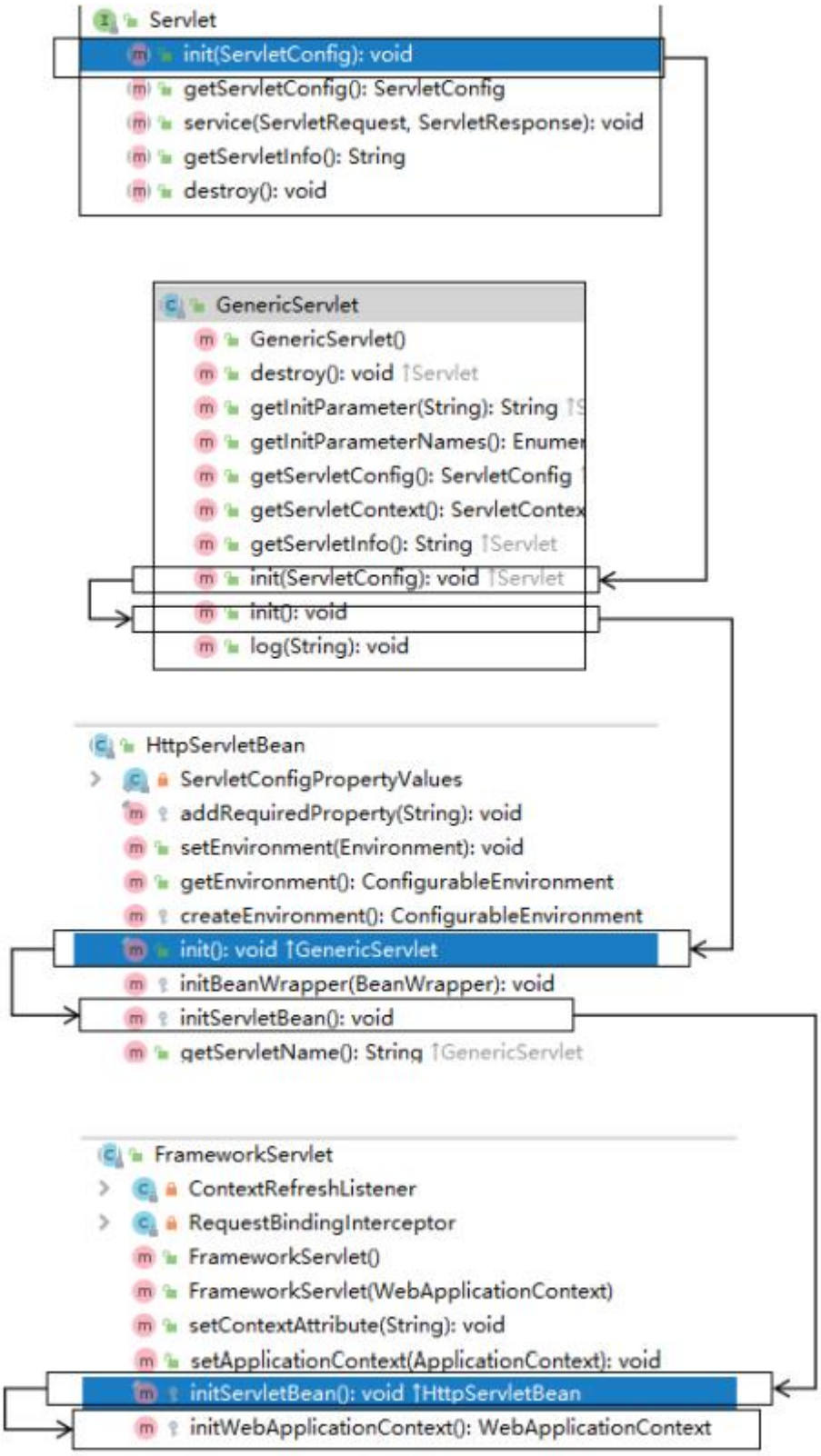
ViewResolver：视图解析器，不需要工程师开发，由框架提供

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

二、 DispatcherServlet 初始化过程



(一)、WebApplicationContext

```
// 创建WebApplicationContext
wac = createWebApplicationContext(rootContext);
```

```
// 刷新WebApplicationContext
onRefresh(wac);
```

(二)、创建 WebApplicationContext

```
    // 通过反射创建 IOC 容器对象
    ConfigurableWebApplicationContext wac =
        (ConfigurableWebApplicationContext)
BeanUtils.instantiateClass(contextClass);

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

    return wac;
```

创建完再调用初始化刷新类然后到 DsipatcherServlet 初始化策略

(三)、DispatcherServlet 初始化策略

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

(四)、调用组件

(五)、执行流程

1) 用户向服务器发送请求，请求被 SpringMVC 前端控制器 DispatcherServlet 捕获。

2) DispatcherServlet 对请求 URL 进行解析，得到请求资源标识符（URI），判断请求 URI 对应的映射

3) 根据该 URI，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 执行链对象的形式返回。

4) DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter。

5) 如果成功获得 HandlerAdapter，此时将开始执行拦截器的

preHandler(…)方法【正向】

6) 提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（Controller)方法，处理请求。 在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作

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

8) 此时将开始执行拦截器的 postHandle(...)方法【逆向】。

9) 根据返回的 ModelAndView（此时会判断是否存在异常：如果存在异常，则执行 HandlerExceptionResolver 进行异常处理）选择一个适合的 ViewResolver 进行视图解析，根据 Model 和 View，来渲染视图。

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

11) 将渲染结果返回给客户端。