



# SpringMVC4 框架技术

北京动力节点教育科技有限公司

## 动力节点课程讲义

DONGLIJIEDIANKECHENGJIANGYI

[www.bjpowernode.com](http://www.bjpowernode.com)

# SpringMVC 5 框架技术讲义

## 第1章 SpringMVC 概述

### 1.1 SpringMVC 简介

SpringMVC 也叫 Spring web mvc。是 Spring 框架的一部分，是在 Spring3.0 后发布的。

### 1.2 SpringMVC 优点

#### 1.基于 MVC 架构

基于 MVC 架构，功能分工明确。解耦合，

#### 2.容易理解，上手快；使用简单。

就可以开发一个注解的 SpringMVC 项目，SpringMVC 也是轻量级的，jar 很小。不依赖的特定的接口和类。

3. 作为 Spring 框架一部分，能够使用 Spring 的 IoC 和 Aop。方便整合 Struts,MyBatis,Hibernate,JPA 等其他框架。

4.SpringMVC 强化注解的使用，在控制器，Service，Dao 都可以使用注解。方便灵活。

使用@Controller 创建处理器对象,@Service 创建业务对象，@Autowired 或者@Resource 在控制器类中注入 Service，Service 类中注入 Dao。

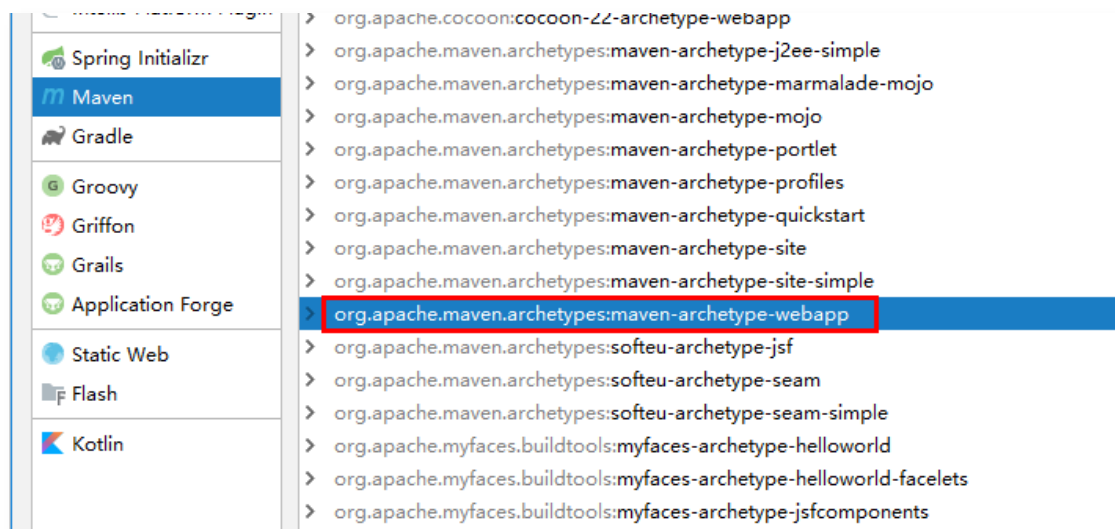
### 1.3 第一个注解的 SpringMVC 程序

所谓 SpringMVC 的注解式开发是指，在代码中通过对类与方法的注解，便可完成处理器在 springmvc 容器的注册。注解式开发是重点。

项目：primary-annotation

完成功能：用户提交一个请求，服务端处理器在接收到这个请求后，给出一条欢迎信息，在响应页面中显示该信息。

### 1.3.1 新建 maven web 项目



### 1.3.2 pom.xml

在创建好 web 项目后，加入 Servlet 依赖，SpringMVC 依赖

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
```

插件：

```
<build>
  <plugins>
    <!-- 编码和编译和JDK版本 -->
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```

    </plugin>
  </plugins>
</build>

```

### 1.3.3 注册中央调度器

```

<!-- 注册中央处理器 -->
<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>*.do</url-pattern>
</servlet-mapping>

```

#### (1) 全限定性类名

该中央调度器为一个 Servlet，名称为 DispatcherServlet。中央调度器的全限定性类名在导入的 Jar 文件 spring-webmvc-5.2.5.RELEASE.jar 的第一个包 org.springframework.web.servlet 下可找到。

#### (2) <load-on-startup/>

在<servlet/>中添加<load-on-startup/>的作用是，标记是否在 Web 服务器(这里是 Tomcat)启动时会创建这个 Servlet 实例，即是否在 Web 服务器启动时调用执行该 Servlet 的 init()方法，而不是在真正访问时才创建。

它的值必须是一个整数。

- 当值大于等于 0 时，表示容器在启动时就加载并初始化这个 servlet，数值越小，该 Servlet 的优先级就越高，其被创建的也就越早；
- 当值小于 0 或者没有指定时，则表示该 Servlet 在真正被使用时才会去创建。
- 当值相同时，容器会自己选择创建顺序。

#### (3) <url-pattern/>

对于<url-pattern/>，可以写为 /，建议写为 \*.do 的形式。

#### (4) 配置文件位置与名称

注册完毕后，可直接在服务器上发布运行。此时，访问浏览器页面，控制台均会抛出 `FileNotFoundException` 异常。即默认要从项目根下的 `WEB-INF` 目录下找名称为 `Servlet` 名称 `-servlet.xml` 的配置文件。这里的“`Servlet` 名称”指的是注册中央调度器 `<servlet-name/>` 标签中指定的 `Servlet` 的 `name` 值。本例配置文件名为 `springmvc-servlet.xml`。

```
Caused by: java.io.FileNotFoundException: Could not open ServletContext resource [/WEB-INF/springmvc-servlet.xml]
    at org.springframework.web.context.support.ServletContextResource.getInputStream(ServletContextResource
```

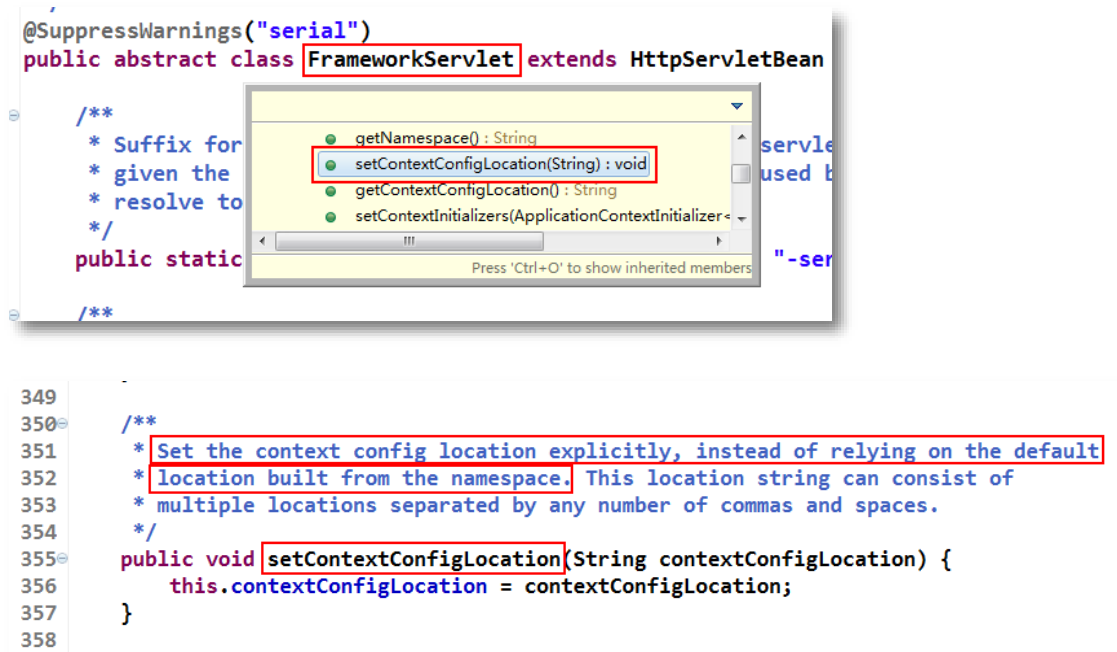
而一般情况下，配置文件是放在类路径下，即 `resources` 目录下。所以，在注册中央调度器时，还需要为中央调度器设置查找 `SpringMVC` 配置文件路径，及文件名。

```
<!-- 注册中央处理器 -->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

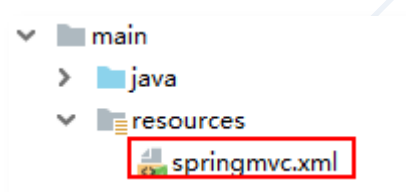
打开 `DispatcherServlet` 的源码，其继承自 `FrameworkServlet`，而该类中有一个属性 `contextConfigLocation`，用于设置 `SpringMVC` 配置文件的路径及文件名。该初始化参数的属性就来自于这里。

```
*/
@SuppressWarnings("serial")
public class DispatcherServlet extends FrameworkServlet {
    /** Well-known name for the MultipartResolver object is
```



### 1.3.4 创建 SpringMVC 配置文件

在工程的类路径即 `src` 目录下创建 SpringMVC 的配置文件 `springmvc.xml`。该文件名可以任意命名。



### 1.3.5 创建处理器

在类上与方法上添加相应注解即可。

**@Controller:** 表示当前类为处理器

**@RequestMapping:** 表示当前方法为处理器方法。该方法要对 `value` 属性所指定的 URI 进行处理与响应。被注解的方法的方法名可以随意。

```

@Controller
public class MyController {
    /**
     * @RequestMapping: 请求映射，把指定的请求交给方法处理
     */
    @RequestMapping(value = "/some.do")
    public ModelAndView doSome(){
        System.out.println("处理some.do请求");
        //调用service 处理请求，把处理结果放入到返回值ModelAndView
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", "使用注解的SpringMVC应用");
        mv.addObject("fun", "doSome");
        mv.setViewName("/show.jsp");
        return mv;
    }
}

```

若有多个请求路径均可匹配该处理器方法的执行，则@RequestMapping 的 value 属性中可以写上一个数组。

ModelAndView 类中的 addObject()方法用于向其 Model 中添加数据。Model 的底层为一个 HashMap。

Model 中的数据存储在 request 作用域中，SpringMVC 默认采用转发的方式跳转到视图，本次请求结束，模型中的数据被销毁。

### 1.3.6 声明组件扫描器

在 springmvc.xml 中注册组件扫描器

```

<!--声明组件扫描器-->
<context:component-scan base-package="com.bjpowernode.controller" />

```

### 1.3.7 定义目标页面

在 webapp 目录下新建一个子目录 jsp，在其中新建一个 jsp 页面 show.jsp。

```

<html>
<head>

    <title>title</title>
</head>
<body>
    /WEB-INF/view/show.jsp<br>
    <h3>msg数据:${msg}</h3>
    <h3>fun数据:${fun}</h3>
</body>
</html>

```

### 1.3.8 修改视图解析器的注册

SpringMVC 框架为了避免对于请求资源路径与扩展名上的冗余，在视图解析器 `InternalResourceViewResolver` 中引入了请求的前缀与后缀。而 `ModelAndView` 中只需给出要跳转页面的文件名即可，对于具体的文件路径与文件扩展名，视图解析器会自动完成拼接。

```

<!-- 注册视图解析器：帮助我们处理视图的路径和扩展名。生成视图对象 -->
<!-- 注册内部资源视图解析器 InternalResourceViewResolver -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- 前缀：表示视图所在的路径 -->
    <property name="prefix" value="/WEB-INF/jsp/" />
    <!-- 后缀：表示视图文件的扩展名 -->
    <property name="suffix" value=".jsp" />
</bean>

```

把 show.jsp 文件放到 /WEB-INF/jsp/路径中



### 1.3.9 修改处理器

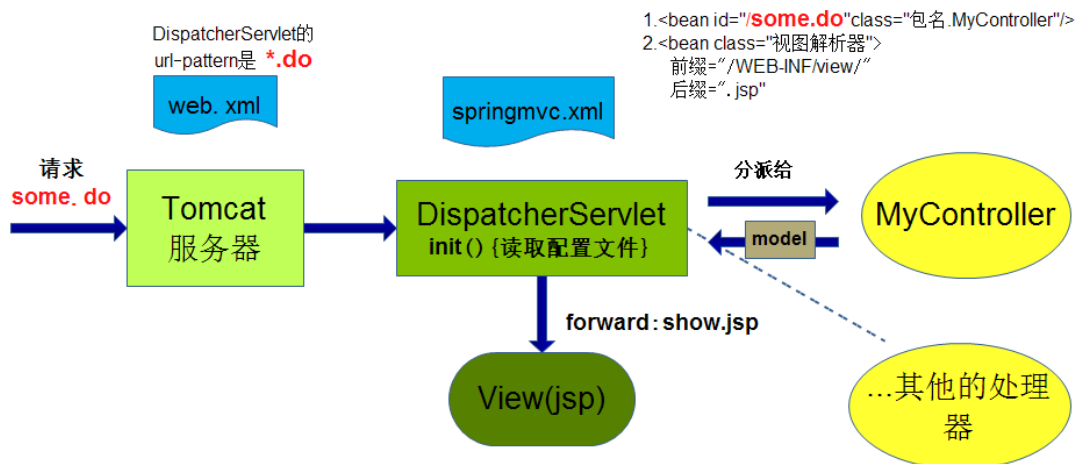
```

@Controller
public class MyController {
    /**
     * @RequestMapping: 请求映射, 把指定的请求交给方法处理
     */
    @RequestMapping(value = "/some.do")
    public ModelAndView doSome(){
        System.out.println("处理some.do请求");
        //调用service 处理请求, 把处理结果放入到返回值ModelAndView
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", "使用注解的SpringMVC应用");
        mv.addObject("fun", "doSome");
        mv.setViewName("show");
        return mv;
    }
}

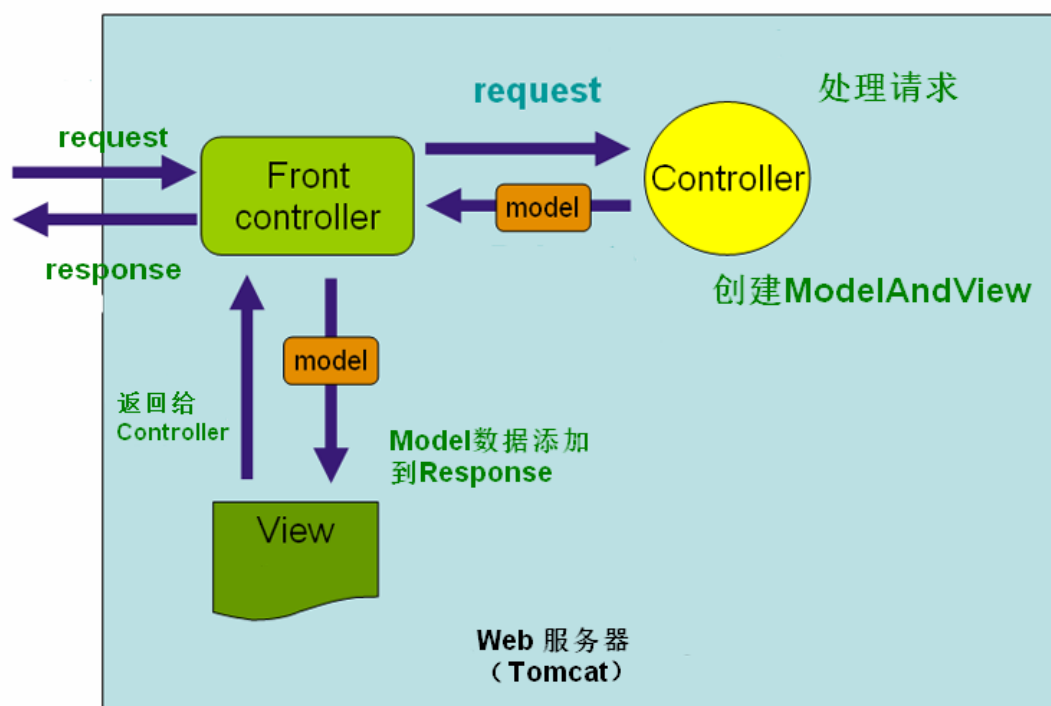
```

使用逻辑视图名称, show 是逻辑视图名称。

### 1.3.10 使用 SpringMVC 框架 web 请求处理顺序

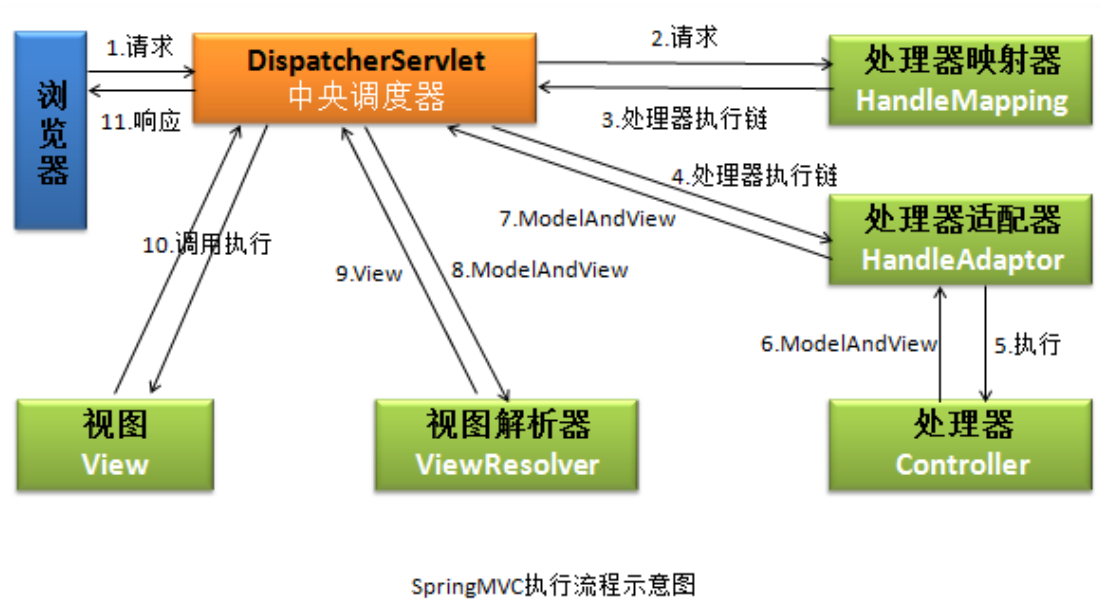


## 1.4 SpringMVC 的 MVC 组件



## 1.5 SpringMVC 执行流程（理解）

### 1.5.1 流程图



### 1.5.2 执行流程简单分析

- （1）浏览器提交请求到中央调度器
- （2）中央调度器直接将请求转给处理器映射器。
- （3）处理器映射器会根据请求，找到处理该请求的处理器，并将其封装为处理器执行链后返回给中央调度器。
- （4）中央调度器根据处理器执行链中的处理器，找到能够执行该处理器的处理器适配器。
- （5）处理器适配器调用执行处理器。
- （6）处理器将处理结果及要跳转的视图封装到一个对象 ModelAndView 中，并将其返回给处理器适配器。
- （7）处理器适配器直接将结果返回给中央调度器。
- （8）中央调度器调用视图解析器，将 ModelAndView 中的视图名称封装为视图对象。
- （9）视图解析器将封装了的视图对象返回给中央调度器
- （10）中央调度器调用视图对象，让其自己进行渲染，即进行数据填充，形成响应对象。
- （11）中央调度器响应浏览器。

## 第2章 SpringMVC 注解式开发

### 2.1 @RequestMapping 定义请求规则

#### 2.1.1 指定模块名称

通过 @RequestMapping 注解可以定义处理器对于请求的映射规则。该注解可以注解在方法上，也可以注解在类上，但意义是不同的。value 属性值常以 “/” 开始。

@RequestMapping 的 value 属性用于定义所匹配请求的 URI。但对于注解在方法上与类上，其 value 属性所指定的 URI，意义是不同的。

一个 @Controller 所注解的类中，可以定义多个处理器方法。当然，不同的处理器方法所匹配的 URI 是不同的。这些不同的 URI 被指定在注解于方法之上的 @RequestMapping 的 value 属性中。但若这些请求具有相同的 URI 部分，则这些相同的 URI，可以被抽取到注解在类之上的 @RequestMapping 的 value 属性中。此时的这个 URI 表示模块的名称。URI 的请求是相对于 Web 的根目录。

换个角度说，要访问处理器的指定方法，必须要在方法指定 URI 之前加上处理器类前定义的模块名称

项目：requestMapping-modelName。在 primary-annotation 基础上进行修改。

Step1: 修改处理器类 MyController。

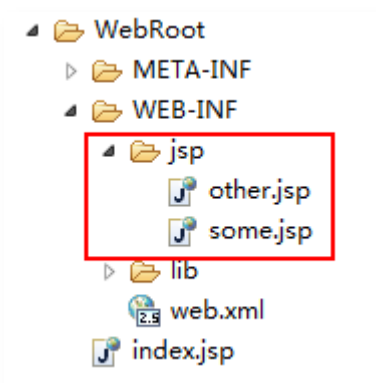
```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping("/some.do")
    public ModelAndView doSome(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        return new ModelAndView("some");
    }

    @RequestMapping("/other.do")
    public ModelAndView doOther(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        return new ModelAndView("other");
    }
}
```

Step2: 添加视图页面

在 /WEB-INF/jsp 目录下添加 some.jsp 与 other.jsp 页面，删除原 welcome.jsp 页面。



```
<body>
    some page
</body>
```

```
<body>
    other page
</body>
```

### 2.1.2 对请求提交方式的定义

对于 `@RequestMapping`，其有一个属性 `method`，用于对被注解方法所处理请求的提交方式进行限制，即只有满足该 `method` 属性指定的提交方式的请求，才会执行该被注解方法。

`Method` 属性的取值为 `RequestMethod` 枚举常量。常用的为 `RequestMethod.GET` 与 `RequestMethod.POST`，分别表示提交方式的匹配规则为 GET 与 POST 提交。

```
@RequestMapping(value="/register.do", method=RequestMethod.POST)
public ModelAndView register(HttpServletRequest request,
    HttpServletResponse response) throws Exception {
```

以上处理器方法只能处理 POST 方式提交的请求。客户端浏览器常用的请求方式，及其提交方式有以下几种：

序号	请求方式	提交方式
1	表单请求	默认 GET，可以指定 POST
2	AJAX 请求	默认 GET，可以指定 POST
3	地址栏请求	GET 请求
4	超链接请求	GET 请求
5	src 资源路径请求	GET 请求

也就是说，只要指定了处理器方法匹配的请求提交方式为 POST，则相当于指定了请求发送的方式：要么使用表单请求，要么使用 AJAX 请求。其它请求方式被禁用。

当然，若不指定 `method` 属性，则无论是 GET 还是 POST 提交方式，均可匹配。即对于请求的提交方式无要求。

项目：requestMapping-method。在 requestMapping-modelName 基础上进行修改。

Step1: 修改处理器类 MyController

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/some.do", method=RequestMethod.GET)
    public ModelAndView doSome(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        return new ModelAndView("some");
    }

    @RequestMapping(value="/other.do", method=RequestMethod.POST)
    public ModelAndView doOther(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        return new ModelAndView("other");
    }
}
```

Step2: 修改 index 页面

```
<a href="test/some.do">跳转到some页面</a><br>
<a href="test/other.do">跳转到other页面</a><br>
<br>
<form action="test/some.do" method="POST">
    <input type="submit" value="跳转到some页面"/>
</form>
<br>
<form action="test/other.do" method="POST">
    <input type="submit" value="跳转到other页面"/>
</form>
```

## 2.2 处理器方法的参数

处理器方法可以包含以下四类参数，这些参数会在系统调用时由系统自动赋值，即程序员可在方法内直接使用。

- HttpServletRequest
- HttpServletResponse
- HttpSession
- 请求中所携带的请求参数

### 2.2.1 逐个参数接收

只要保证请求参数名与该请求处理方法的参数名相同即可。

项目：receiveParameters-property。在 requestMapping-method 基础上修改。

Step1: 修改 index 页面

```
<form action="test/register.do" method="POST">
    姓名: <input type="text" name="name"/><br>
    年龄: <input type="text" name="age"/><br>
    <input type="submit" value="注册"/>
</form>
```

Step2: 修改处理器类 MyController

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/register.do")
    public ModelAndView register(String name, int age) {
        ModelAndView mv = new ModelAndView();
        // 相当于request.setAttribute("myname",name);
        mv.addObject("myname", name);
        mv.addObject("myage", age);
        mv.setViewName("show");
        return mv;
    }
}
```

Step3: 添加 show 页面

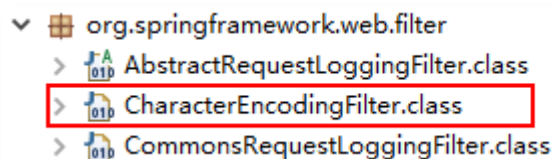
在/WEB-INF/jsp 下添加 show.jsp 页面。

```
<body>
    name = ${myname }<br>
    age = ${myage }<br>
</body>
```

### 2.2.2 请求参数中文乱码问题

对于前面所接收的请求参数，若含有中文，则会出现中文乱码问题。Spring 对于请求参

数中的中文乱码问题，给出了专门的字符集过滤器：spring-web-5.2.5.RELEASE.jar 的 org.springframework.web.filter 包下的 CharacterEncodingFilter 类。



## (1) 解决方案

在 web.xml 中注册字符集过滤器，即可解决 Spring 的请求参数的中文乱码问题。不过，最好将该过滤器注册在其它过滤器之前。因为过滤器的执行是按照其注册顺序进行的。

直接在项目 receiveParameters-property 上进行修改。

```
<!-- 注册字符集过滤器:解决post请求乱码的问题 -->
<filter>
  <filter-name>characterEncodingFilter</filter-name>
  <!-- spring-web.jar -->
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <!-- 指定字符集 -->
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
  <!-- 强制request使用字符集encoding -->
  <init-param>
    <param-name>forceRequestEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
  <!-- 强制response使用字符集encoding -->
  <init-param>
    <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>
```

## (2) 源码分析

字符集设置核心方法:



```

@Override
protected void doFilterInternal(
    HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {

    String encoding = getEncoding();
    if (encoding != null) {
        if (isForceRequestEncoding() || request.getCharacterEncoding() == null) {
            request.setCharacterEncoding(encoding);
        }
        if (isForceResponseEncoding()) {
            response.setCharacterEncoding(encoding);
        }
    }
    filterChain.doFilter(request, response);
}

```

强制request使用encoding的字符编码，忽略代码中设置的编码

强制response使用encoding的字符编码

### 2.2.3 校正请求参数名@RequestParam

所谓校正请求参数名，是指若请求 URL 所携带的参数名称与处理方法中指定的参数名不相同，则需在处理方法参数前，添加一个注解@RequestParam(“请求参数名”)，指定请求 URL 所携带参数的名称。该注解是对处理器方法参数进行修饰的。value 属性指定请求参数的名称。

项目：receiveParameters-params。在 receiveParameters-property 基础上修改。

Step1: 修改 index 页面

将表单中的参数名称修改的与原来不一样。

```

<form action="test/register.do" method="POST">
    姓名: <input type="text" name="rname"/><br>
    年龄: <input type="text" name="rage"/><br>
    <input type="submit" value="注册"/>
</form>

```

Step2: 修改处理器类 MyController

```

@Controller
public class MyController {
    /**
     * 处理器方法的形参前面加入@RequestParam(value="请求中参数名")
     */
    @RequestMapping(value="/some.do")
    public ModelAndView doSome(
        @RequestParam(value="rage") Integer age,
        @RequestParam(value="rname") String name){
        System.out.println("===doSome== name:"+name+"|age:"+age);
        //调用Service处理请求，把处理结果和视图放入到ModelAndView
        ModelAndView mv = new ModelAndView();
        mv.addObject("myname", name);
        mv.addObject("myage", age);
        mv.setViewName("show");
        return mv;
    }
}

```

required 属性:

```

* @RequestParam属性
* 1.value , 请求中参数名
* 2.required, boolean类型的, 默认是true
* true:表示请求中必须有参数
* false:可以没有此参数
*/
@RequestMapping(value="/some.do")
public ModelAndView doSome(@RequestParam(value="rage",required=false) Integer age,
    @RequestParam(value="rname",required=false) String name){

```

## 2.2.4 对象参数接收

将处理器方法的参数定义为一个对象，只要保证请求参数名与这个对象的属性同名即可。  
项目：receiveParameters-object。在 receiveParameters-property 基础上修改。

Step1: 定义类 Student

```

public class Student {
    private String name;
    private int age;

    // getter and setter
    // toString()
}

```

Step2: 修改处理器类 MyController

```

@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/register.do")
    public ModelAndView register(Student student) {
        ModelAndView mv = new ModelAndView();
        mv.addObject("myStudent", student);
        mv.setViewName("show");
        return mv;
    }
}

```

Step3: 修改 show 页面

```

<body>
    student = ${myStudent }<br>
</body>

```

## 2.3 处理器方法的返回值

使用@Controller 注解的处理器的方法，其返回值常用的有四种类型：

- 第一种：ModelAndView
- 第二种：String
- 第三种：无返回值 void
- 第四种：返回自定义类型对象

根据不同的情况，使用不同的返回值。

### 2.3.1 返回 ModelAndView

若处理器方法处理完后，需要跳转到其它资源，且又要在跳转的资源间传递数据，此时处理器方法返回 ModelAndView 比较好。当然，若要返回 ModelAndView，则处理器方法中需要定义 ModelAndView 对象。

在使用时，若该处理器方法只是进行跳转而不传递数据，或只是传递数据而并不向任何资源跳转（如对页面的 Ajax 异步响应），此时若返回 ModelAndView，则将总是有一部分多余：要么 Model 多余，要么 View 多余。即此时返回 ModelAndView 将不合适。

### 2.3.2 返回 String

处理器方法返回的字符串可以指定逻辑视图名，通过视图解析器解析可以将其转换为物

理视图地址

### 返回内部资源逻辑视图名

若要跳转的资源为内部资源，则视图解析器可以使用 `InternalResourceViewResolver` 内部资源视图解析器。此时处理器方法返回的字符串就是要跳转页面的文件名去掉文件扩展名后的部分。这个字符串与视图解析器中的 `prefix`、`suffix` 相结合，即可形成要访问的 URI。

```
<!-- 注册 视图解析器 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/admin/" />
    <property name="suffix" value=".jsp" />
</bean>
```

项目：returnString-viewName。在 `receiveParameters-object` 基础上修改。

直接修改处理器类 `MyController`

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/register.do")
    public String register(HttpServletRequest request, Student student) {
        request.setAttribute("myStudent", student);
        return "show";
    }
}
```

当然，也可以直接返回资源的物理视图名。不过，此时就不需要再在视图解析器中再配置前缀与后缀了。

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/register.do")
    public String register(HttpServletRequest request, Student student) {
        request.setAttribute("myStudent", student);
        return "/WEB-INF/jsp/welcome.jsp";
    }
}
```

### 2.3.3 返回 void（了解）

对于处理器方法返回 `void` 的应用场景，AJAX 响应。

若处理器对请求处理后，无需跳转到其它任何资源，此时可以让处理器方法返回 `void`。

例如，对于 AJAX 的异步请求的响应。

项目：returnVoid-ajax。在 primary-annotation 基础上进行修改。

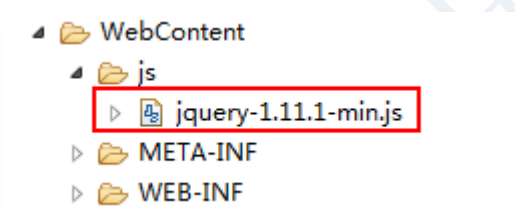
Step1: maven 加入 jackson 依赖

由于本项目中服务端向浏览器传回的是 JSON 数据，需要使用一个工具类将字符串包装为 JSON 格式，所以需要导入 JSON 的依赖。

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

Step2: 引入 jQuery 库

由于本项目要使用 jQuery 的 ajax() 方法提交 AJAX 请求，所以项目中需要引入 jQuery 的库。在 WebRoot 下新建一个 Folder（文件夹），命名为 js，并将 jquery-1.11.1.js 文件放入其中。



当然，该 jQuery 库文件，需要在使用 ajax() 方法的 index 页面中引入。

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript" src="js/jquery-1.11.1-min.js"></script>
```

Step3: 定义 index 页面

index 页面由两部分内容构成：一个是<button/>，用于提交 AJAX 请求；一个是<script/>，用于处理 AJAX 请求。

```

<script type="text/javascript" src="js/jquery-1.11.1-min.js"></script>
<script type="text/javascript">
    $(function(){
        $("button").click(function(){
            $.ajax({
                url:"myajax.do",
                data:{
                    name:"zs",
                    age:"24"
                },
                type:"post",
                dataType:"json",
                success:function(resp){
                    alert("resp:"+resp.name+" "+resp.age);
                }
            })
        })
    })

```

点击按钮发起请求

```

<body>
    index.jsp <br>
    <button>发起Ajax请求</button>
</body>

```

Step4: 定义对象 Student

```

public class Student {

    //属性名和请求参数名一样
    private String name;
    private Integer age;
    //set/get
}

```

Step5: 修改处理器类 MyController

处理器对于 AJAX 请求中所提交的参数，可以使用逐个接收的方式，也可以以对象的方式整体接收。只要保证 AJAX 请求参数与接收的对象类型属性同名。

以逐个方式接收参数：

```

/**
 * 处理器方法返回值void,不能表示数据,也没有视图。
 * 可以通过使用HttpServletResponse的输出对象,把数据输出到浏览器
 */
@RequestMapping(value="/myajax.do")
public void doSome(Integer age,String name,
    HttpServletResponse response) throws IOException {
    //调用Service处理请求,把处理结果转为一个对象存储
    Student student = new Student();
    student.setName(name);
    student.setAge(age);

    //使用jackson工具库,把Student转为json
    ObjectMapper mapper = new ObjectMapper();
    String json = mapper.writeValueAsString(student);
    System.out.println("json:"+json);

    //使用HttpServletResponse输出数据到浏览器
    PrintWriter pw = response.getWriter();
    pw.print(json);
    pw.flush();
    pw.close();
}

```

Step6: 删除视图页面

由于是服务端直接向浏览器发回数据,所以也就无需视图页面了,所以需要删除WEB-INF中的jsp目录及其中的show页面。

## 2.3.4 返回对象 Object

处理器方法也可以返回 Object 对象。这个 Object 可以是 Integer, String, 自定义对象, Map, List 等。但返回的对象不是作为逻辑视图出现的,而是作为直接在页面显示的数据出现的。

返回对象,需要使用@ResponseBody注解,将转换后的JSON数据放入到响应体中。

### (1) 环境搭建

#### A、maven pom.xml

由于返回 Object 数据,一般都是将数据转化为了JSON对象后传递给浏览器页面的。而这个由 Object 转换为JSON,是由Jackson工具完成的。所以需要导入Jackson的相关Jar包。

依赖:

```

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>

```

```

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

```

## B、 声明注解驱动

将 Object 数据转化为 JSON 数据，需要由消息转换器 `HttpMessageConverter` 完成。而转换器的开启，需要由 `<mvc:annotation-driven/>` 来完成。

SpringMVC 使用消息转换器实现请求数据和对象，处理器方法返回对象和响应输出之间的自动转换

当 Spring 容器进行初始化过程中，在 `<mvc:annotation-driven/>` 处创建注解驱动时，默认创建了七个 `HttpMessageConverter` 对象。也就是说，我们注册 `<mvc:annotation-driven/>`，就是为了让容器为我们创建 `HttpMessageConverter` 对象。

```

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

```

`HttpMessageConverter` 接口：`HttpMessageConverter<T>` 是 Spring3.0 新添加的一个接口，负责将请求信息转换为一个对象（类型为 T），将对象（类型为 T）输出为响应信息

`HttpMessageConverter<T>` 接口定义的方法：

`boolean canRead(Class<?> clazz, MediaType mediaType)`: 指定转换器可以读取的对象类型，即转换器是否可将请求信息转换为 `clazz` 类型的对象，同时指定支持 MIME 类型 (text/html,application/json 等)

`boolean canWrite(Class<?> clazz, MediaType mediaType)`: 指定转换器是否可将 `clazz` 类型的对象写到响应流中，响应流支持的媒体类型在 `MediaType` 中定义。

`List<MediaType> getSupportMediaTypes()`: 该转换器支持的媒体类型。

`T read(Class<? extends T> clazz, HttpInputMessage inputMessage)`: 将请求信息流转换为 T 类型的对象。

`void write(T t, MediaType contentType, HttpOutputMessage outputMessage)`: 将 T 类型的对象写到响应流中，同时指定相应的媒体类型为 `contentType`

加入注解驱动 `<mvc:annotation-driven/>` 后适配器类的 `messageConverters` 属性值



▼ messageConverters	ArrayList<E> (id=2024)
▼ ▲ elementData	Object[7] (id=2028)
> ▲ [0]	ByteArrayHttpMessageConverter (id=2029)
> ▲ [1]	StringHttpMessageConverter (id=2030)
> ▲ [2]	ResourceHttpMessageConverter (id=2031)
> ▲ [3]	SourceHttpMessageConverter<T> (id=2032)
> ▲ [4]	AllEncompassingFormHttpMessageConverter (id=2033)
> ▲ [5]	Jaxb2RootElementHttpMessageConverter (id=2034)
> ▲ [6]	MappingJackson2HttpMessageConverter (id=2035)

HttpMessageConverter 接口实现类	作用
ByteArrayHttpMessageConverter	负责读取二进制格式的数据和写出二进制格式的数据
StringHttpMessageConverter	负责读取字符串格式的数据和写出字符串格式的数据
ResourceHttpMessageConverter	负责读取资源文件和写出资源文件数据
SourceHttpMessageConverter	能够读 / 写来自 HTTP 的请求与响应的 javax.xml.transform.Source ,支持 DOMSource, SAXSource, 和 StreamSource 的 XML 格式
AllEncompassingFormHttpMessageConverter	负责处理表单(form)数据
Jaxb2RootElementHttpMessageConverter	使用 JAXB 负责读取和写入 xml 标签格式的数据
MappingJackson2HttpMessageConverter	负责读取和写入 json 格式的数据。利用 Jackson 的 ObjectMapper 读写 json 数据,操作 Object 类型数据,可读取 application/json, 响应媒体类型为 application/json

## (2) 返回自定义类型对象

返回自定义类型对象时,不能以对象的形式直接返回给客户端浏览器,而是将对象转换为 JSON 格式的数据发送给浏览器的。

由于转换器底层使用了 Jackson 转换方式将对象转换为 JSON 数据,所以需要导入 Jackson 的相关 Jar 包。

项目: returnObject-custom。在 returnVoid-ajax 基础上进行修改。

Step1: 定义数据类

```
public class Student {
    private String name;
    private int age;

    // getter and setter
    // toString()
}
```

Step2: 修改处理器 MyController

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value = "/myajax.do")
    @ResponseBody
    public Student doStudentJson() {
        //创建java对象, 转为json
        Student student = new Student();
        student.setName("张三同学");
        student.setAge(20);
        return student;
    }
}
```

Step3: 修改 index 页面

```
<script type="text/javascript">
    $(function(){
        $("button").click(function(){
            $.ajax({
                url:"test/myajax.do",
                success:function(data){
                    alert(data.name + " " + data.age);
                }
            });
        });
    });
</script>
<body>
    <button>提交Ajax请求</button>
</body>
```

### (3) 返回 List 集合

项目: returnObject-list。在 returnObject-custom 基础上进行修改。

Step1: 修改处理器 MyController

```

@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value = "/myajax.do" )
    @ResponseBody
    public List<Student> doStudentJsonArray() {
        //创建List对象, 转为jsonarray
        List<Student> students = new ArrayList<>();
        students.add(new Student("张三",22)); //张三
        students.add(new Student("李四",22)); //李四
        return students;
    }
}

```

Step2: 修改 index 页面

```

<script type="text/javascript">
    $(function(){
        $("button").click(function () {
            $.ajax({
                url:"test/myajax.do",
                success:function(data){
                    $.each(data,function(i,n){
                        alert(n.name+"==="+n.age)
                    })
                }
            })
        })
    })
</script>
</head>
<body>
    <button id="btnAjax">发起ajax请求</button>
</body>

```

#### (4) 返回字符串对象

若要返回非中文字符串, 将前面返回数值型数据的返回值直接修改为字符串即可。但若返回的字符串中带有中文字符, 则接收方页面将会出现乱码。此时需要使用 @RequestMapping 的 produces 属性指定字符集。

produces, 产品, 结果, 即该属性用于设置输出结果类型。

项目: returnObject-String。

直接修改处理器。

```

@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(
        value = "/myajax.do",
        produces = "text/plain;charset=utf-8"
    )
    @ResponseBody
    public String doText(){
        return "HelloSpringMVC使用注解开发";
    }
}

```

修改页面:

```

<script type="text/javascript">
    $(function(){
        $("button").click(function () {
            $.ajax({
                url:"test/myajax.do",
                success:function(data){
                    alert(data)
                }
            })
        })
    })
</script>
</head>
<body>
    <button id="btnAjax">发起ajax请求</button>
</body>

```

## 2.4 解读<url-pattern/>

### 2.4.1 配置详解

#### (1) \*.do

在没有特殊要求的情况下，SpringMVC 的中央调度器 DispatcherServlet 的<url-pattern/>常使用后缀匹配方式，如写为\*.do 或者 \*.action, \*.mvc 等。

#### (2) /

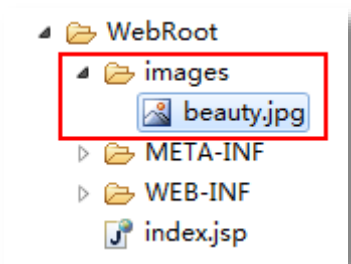
可以写为/, 因为 DispatcherServlet 会将向静态资源的获取请求，例如.css、.js、.jpg、.png 等资源的获取请求，当作是一个普通的 Controller 请求。中央调度器会调用处理器映射器为其查找相应的处理器。当然也是找不到的，所以在这种情况下，所有的静态资源获取请求也均会报 404 错误。

项目：url-pattern。在项目 primary-annotation 基础上进行修改。

需求：在 index.jsp 页面中存在一个访问图片的链接。该项目用于演示将<url-pattern/>写为\*.do 可以访问到该图片，而写为/，则无法访问。

### A、 在项目中添加图片

在项目的 WebRoot 下添加一个目录 images，并在其中添加一张图片资源。



### B、 修改 index 页面

```
<body>

</body>
```

### C、 修改<url-pattern/>的值

保持<url-pattern/>的值为 \*.do，扩展名方式，图片会正常显示。

将<url-pattern/>的值修改为 /，则图片将无法显示。

## 2.4.2 静态资源访问

<url-pattern/>的值并不是说写为/后，静态资源就无法访问了。经过一些配置后，该问题也是可以解决的。

### (1) 使用<mvc:default-servlet-handler/>

声明了 <mvc:default-servlet-handler /> 后，springmvc 框架会在容器中创建 DefaultServletHttpRequestHandler 处理器对象。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。一般的服务器都有默认的 Servlet。

在 Tomcat 中，有一个专门用于处理静态资源访问的 Servlet 名叫 DefaultServlet。其

<servlet-name/>为 default。可以处理各种静态资源访问请求。该 Servlet 注册在 Tomcat 服务器的 web.xml 中。在 Tomcat 安装目录/conf/web.xml。

```

101
102     <servlet>
103         <servlet-name>default</servlet-name>
104         <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
105         <init-param>
106             <param-name>debug</param-name>
107             <param-value>0</param-value>
108         </init-param>
109         <init-param>
110             <param-name>listings</param-name>
111             <param-value>>false</param-value>
112         </init-param>
113         <load-on-startup>1</load-on-startup>
114     </servlet>
115

```

项目：url-pattern-2。在项目 url-pattern 基础上修改。

只需要在 springmvc.xml 中添加<mvc:default-servlet-handler/>标签即可。

```

    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/sp

    <mvc:default-servlet-handler/>

</beans>

```

<mvc:default-servlet-handler/>表示使用 DefaultServletHttpRequestHandler 处理器对象。而该处理器调用了 Tomcat 的 DefaultServlet 来处理静态资源的访问请求。

当然了，要想使用<mvc: .../>标签，需要引入 mvc 约束

```

springmvc.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

```

该约束可从 Spring 帮助文档中搜索关键字 spring-mvc.xsd 即可获取：  
docs/spring-framework-reference/htmlsingle/index.html

## (2) 使用<mvc:resources/>（掌握）

项目：url-pattern-3。在项目 url-pattern 基础上修改。

在 Spring3.0 版本后，Spring 定义了专门用于处理静态资源访问请求的处理器 ResourceHttpRequestHandler。并且添加了<mvc:resources/>标签，专门用于解决静态资源无

法访问问题。需要在 springmvc 配置文件中添加如下形式的配置：

```
<mvc:resources location="/images/" mapping="/images/**"/>
```

location 表示静态资源所在目录。当然，**目录不要使用/WEB-INF/及其子目录。**

mapping 表示对该资源的请求（以 /images/ 开始的请求，如 /image/beauty.jpg ， /images/car.png 等）。注意，后面是两个星号\*\*。

### （3） 声明注解驱动

解决动态资源和静态资源冲突的问题，在 springmvc 配置文件加入：

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

## 第3章 SSM 整合开发

SSM 编程，即 SpringMVC + Spring + MyBatis 整合，是当前最为流行的 JavaEE 开发技术架构。其实 SSM 整合的实质，仅仅就是将 MyBatis 整合入 Spring。因为 SpringMVC 原本就是 Spring 的一部分，不用专门整合。

SSM 整合的实现方式可分为两种：基于 XML 配置方式，基于注解方式。

### 3.1 搭建 SSM 开发环境

#### 3.1.1 maven pom.xml

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

<!-- jsp依赖 -->
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.2.1-b03</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>
```



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>

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

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.3.1</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.1</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.9</version>
</dependency>

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.12</version>
</dependency>
```

插件:

```
<build>
  <resources>
```

```

<resource>
  <directory>src/main/java</directory><!--所在的目录-->
  <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
    <include>**/*.properties</include>
    <include>**/*.xml</include>
  </includes>
  <filtering>false</filtering>
</resource>
</resources>

<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
    </configuration>
  </plugin>
</plugins>
</build>

```

### 3.1.2 配置 web.xml

#### (1) 注册 ContextLoaderListener 监听器

```

<!--注册spring的监听器-->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:conf/applicationContext.xml</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

```

注册 ServletContext 监听器的实现类 ContextLoaderListener，用于创建 Spring 容器及将创建好的 Spring 容器对象放入到 ServletContext 的作用域中。

## (2) 注册字符集过滤器

```
<!--注册字符集过滤器-->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceRequestEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <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>
```

注册字符集过滤器，用于解决请求参数中携带中文时产生乱码问题。

## (3) 配置中央调度器

配置中央调度器时需要注意，SpringMVC 的配置文件名与其它 Spring 配置文件名不相同。这样做的目的是 Spring 容器创建管理 Spring 配置文件中的 bean，SpringMVC 容器中负责视图层 bean 的初始。

```
<!--注册springmvc的中央调度器-->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:conf/dispatcherServlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

## 3.2 SSM 整合注解开发

项目：ssm

需求：完成学生注册和信息浏览。

### 3.2.1 建表 Student

使用 Student 表

名	类型	长度	小数点	不是 null	
▶ id	int	11	0	<input checked="" type="checkbox"/>	🔑 1
name	varchar	255	0	<input type="checkbox"/>	
age	int	11	0	<input type="checkbox"/>	

### 3.2.2 新建 Web 工程

工程名称 ssm

### 3.2.3 maven 依赖

```

<!--servlet-->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<!-- jsp 依赖 -->
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.2.1-b03</version>
  <scope>provided</scope>
</dependency>
<!--springmvc-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
<!--事务的-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>

```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
<!--aspectj 依赖-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>5.2.5.RELEASE</version>
</dependency>
<!--jackson-->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
<!--mybatis 和 spring 整合的-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.3.1</version>
</dependency>
<!--mybatis-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.1</version>
</dependency>
<!--mysql 驱动-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.9</version>
</dependency>
<!--druid-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
```

```
    <version>1.1.12</version>
  </dependency>
</dependencies>
```

插件:

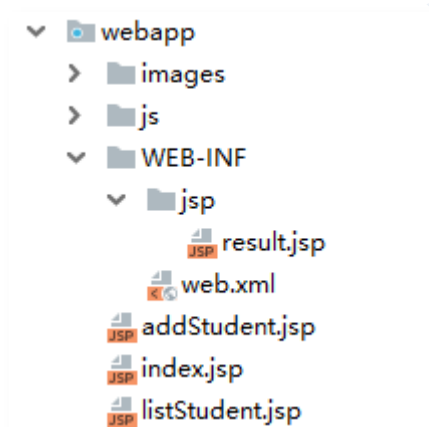
```
<build>
  <resources>
    <resource>
      <directory>src/main/java</directory><!--所在的目录-->
      <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>>false</filtering>
    </resource>
  </resources>

  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

### 3.2.4 定义包，组织程序的结构。



jsp 文件:



### 3.2.5 编写配置文件

Jdbc 属性配置文件 jdbc.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/springdb
jdbc.user=root
jdbc.pwd=root
```

Spring 配置文件 applicationContext.xml

```

<!-- 注册组件扫描器 -->
<context:component-scan base-package="com.bjpowernode.service" />

<!-- 引入属性配置文件 -->
<context:property-placeholder location="classpath:resource/jdbc.properties"/>
<!-- 配置阿里的Druid数据库连接池 -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
      init-method="init" destroy-method="close">
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.user}" />
    <property name="password" value="${jdbc.password}" />
</bean>
<!-- 注册SqlSessionFactoryBean -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" value="classpath:resource/mybatis.xml" />
</bean>
<!-- 动态代理对象 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
    <property name="basePackage" value="com.bjpowernode.dao" />
</bean>

```

Springmvc 配置文件: springmvc.xml

```

<!--注册组件扫描器
      base-package:指定@Controller注解所在的包
-->
<context:component-scan base-package="com.bjpowernode.controllers" />

<!-- 指定视图解析器 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- 前缀和后缀 -->
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
</bean>

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

```

mybatis.xml



```

<configuration>
    <!-- 配置别名 -->
    <typeAliases>
        <package name="com.bjpowernode.beans"/>
    </typeAliases>

    <!-- 指定sql映射文件 -->
    <mappers>
        <!--
            name:指定dao接口的包名
            <package> 使用的条件:
                1.sql映射文件名和Dao接口名要一样。
                2.sql映射文件和Dao接口在同一目录。
        -->
        <package name="com.bjpowernode.dao"/>
    </mappers>
</configuration>

```

### 3.2.6 定义 web.xml

- 1) 注册 ContextLoaderListener
- 2) 注册 DisatcherServlet
- 3) 注册字符集过滤器
- 4) 同时创建 Spring 的配置文件和 SpringMVC 的配置文件

### 3.2.7 实体类 Student

```

public class Student {

    private Integer id;
    private String name;
    private int age;
    // set , get 方法
}

```

### 3.2.8 Dao 接口和 sql 映射文件

```

public interface StudentDao {

    int insertStudent(Student student);
    List<Student> selectAllStudents();
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.bjpowernode.dao.StudentDao">
    <insert id="insertStudent">
        insert into student(name,age) values("#{name}",#{age})
    </insert>

    <select id="selectStudentByPage" resultType="Student">
        select id,name,age from student order by id desc
    </select>
</mapper>

```

### 3.2.9 Service 接口和实现类

Service 接口

```

public interface StudentService {

    int addStudent(Student student);
    List<Student> findStudents();
}

```

Service 实现类

```

//创建Service对象
@Service(value="studentService")
public class StudentServiceImpl implements StudentService {

    //引用类型 @Autowired , @Resource byType | byName
    //byType
    @Autowired
    private StudentDao stuDao;

    @Override
    public int addStudent(Student student) {
        return stuDao.insertStudent(student);
    }

    @Override
    public List<Student> findStudents() {
        return stuDao.selectAllStudents();
    }
}

```

### 3.2.10 处理器定义

StuentController.java

```
@Controller
@RequestMapping("/student")
public class StudentController {
    @Autowired
    private StudentService studentService;
    /** 注册学 */
    @RequestMapping("/addStudent.do")
    public ModelAndView addStudent(Student student){
        ModelAndView mv = new ModelAndView();
        //调用Service处理业务。将结果放入到ModelAndView
        int rows = studentService.addStudent(student);
        if( rows > 0 ){
            mv.addObject("msg", "注册成功!!!");
            mv.setViewName("success");
        } else {
            mv.addObject("msg", "注册失败!!!");
            mv.setViewName("fail");
        }
        return mv;
    }
    /** 响应ajax, 输出数据 */
    @RequestMapping("/queryStudent.do")
    @ResponseBody
    public List<Student> queryStudent(){
        List<Student> students = studentService.queryStudent();
        return students;
    }
}
```

### 3.2.11 定义视图-首页文件--- index.jsp

指定路径:

```
<%
    String basePath = request.getScheme() + "://" +
    request.getServerName() + ":" + request.getServerPort() +
    request.getContextPath() + "/";
%>
```

指定 base 标签

```
<head>
    <base href="<%=basePath%>">
    <title>title</title>
</head>
```

```
<body>  
  <div align="center">  
    <p>学生注册页面</p>  
    <form action="student/addStudent.do" method="post">  
      <table>  
        <tr>  
          <td>姓名: </td>  
          <td><input type="text" name="name"></td>  
        </tr>  
        <tr>  
          <td>年龄: </td>  
          <td><input type="text" name="age"></td>  
        </tr>  
        <tr>  
          <td>     </td>  
          <td><input type="submit" value="注册"></td>  
        </tr>  
      </table>  
    </form>  
  </div>  
</body>
```

页面表格

```
<div align="center">
  <p>查询学生数据</p>
  <table>
    <thead>
      <tr>
        <td>id</td>
        <td>姓名</td>
        <td>年龄</td>
      </tr>
    </thead>
    <tbody id="sttbody">
      <tr>
        <td>1</td>
        <td>张三</td>
        <td>18</td>
      </tr>
    </tbody>
  </table>
</div>
```

js 内容:

引入 JQuery

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%
String basePath = request.getScheme() + "://" +
    request.getServerName() + ":" +
    request.getServerPort() +
    request.getContextPath() + "/";
%>
<html>
<head>
  <base href="<%=basePath%>">
  <title>title</title>
  <script type="text/javascript" src="js/jquery-1.11.1-min.js" ></script>
```

js 发起 ajax

```
<script type="text/javascript">
    $(function () {
        stuinfo()
    })
    function stuinfo() {
        $.ajax({
            url:"student/queryStudent.do",
            type:"post",
            dataType:"json",
            success:function (resp) {
                // json array
                $("#sttbody").html("");
                $.each(resp,function (i,n) {
                    $("#sttbody").append("<tr>")
                        .append("<td>" + n.id+"</td>")
                        .append("<td>" + n.name+"</td>")
                        .append("<td>" + n.age+"</td>")
                        .append("</tr>")
                })
            }
        })
    }
}
</script>
```

### 3.2.14 注册成功页面--- success.jsp

```
<body>
    success.jsp : 注册成功!!!
</body>
```

### 3.2.15 注册失败页面--- fail.jsp

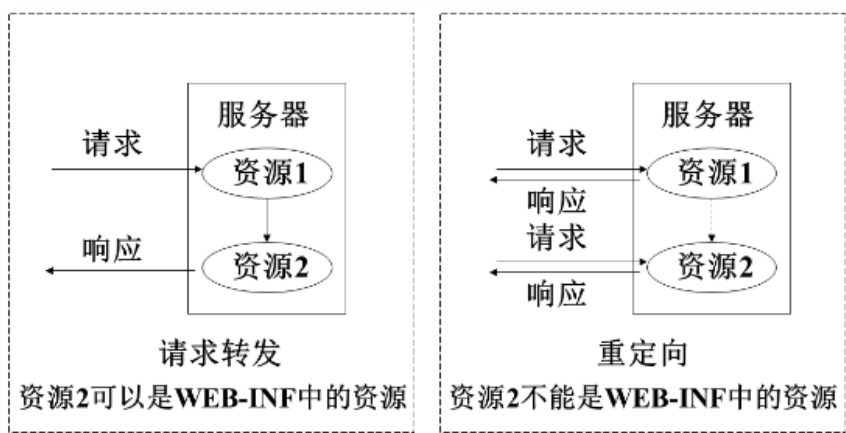
```
<body>
    fail.jsp : 注册失败!!!
</body>
```

## 第4章 SpringMVC 核心技术

### 4.1 请求重定向和转发

当处理器对请求处理完毕后,向其它资源进行跳转时,有两种跳转方式:请求转发与重定向。而根据所要跳转的资源类型,又可分为两类:跳转到页面与跳转到其它处理器。

注意,对于请求转发的页面,可以是 WEB-INF 中页面;而重定向的页面,是不能为 WEB-INF 中页的。因为重定向相当于用户再次发出一次请求,而用户是不能直接访问 WEB-INF 中资源的。



SpringMVC 框架把原来 Servlet 中的请求转发和重定向操作进行了封装。现在可以使用简单的方式实现转发和重定向。

forward:表示转发,实现 `request.getRequestDispatcher("xx.jsp").forward()`

redirect:表示重定向,实现 `response.sendRedirect("xxx.jsp")`

#### 4.1.1 请求转发

处理器方法返回 ModelAndView 时,需在 `setViewName()` 指定的视图前添加 **forward:**,且此时的视图不再与视图解析器一同工作,这样可以在配置了解析器时指定不同位置的视图。**视图页面**必须写出相对于项目根的路径。forward 操作不需要视图解析器。

处理器方法返回 String,在视图路径前面加入 **forward:** 视图完整路径。

```

/**
 * 处理器方法返回ModelAndView:实现转发到其他的视图页面
 * 语法: setViewName("forward:视图的完整路径")
 * forward关键字的特点: 不和视图解析器一同工作, 就当项目中没有视图解析器。
 */
@RequestMapping(value="/some.do")
public ModelAndView doSome(Integer age,String name){
    System.out.println("=====doSome===== name:"+name+"|age:"+age);
    ModelAndView mv = new ModelAndView();
    mv.addObject("myname", name);
    mv.addObject("myage",age);
    //使用forward,显示指定转发操作, forward:视图完整路径
    //mv.setViewName("forward:/WEB-INF/view/show.jsp");
    mv.setViewName("forward:/other.jsp");
    return mv;
}

```

#### 4.1.2 请求重定向

在处理器方法返回的视图字符串的前面添加 **redirect:**, 则可实现重定向跳转。

处理器方法定义:

```

/**
 * 处理器方法返回ModelAndView, 实现重定向到视图页面
 * 语法: mv.setViewName("redirect:视图完整路径"), 不能使用逻辑视图名称
 */
@RequestMapping(value = "/doredirect.do")
public ModelAndView doRedirect(String name, Integer age){
    System.out.println("执行redirect重定向" + name + " " + age);
    ModelAndView mv = new ModelAndView();
    //指定视图
    mv.setViewName("redirect:/other.jsp");
    //重定向不能访问受保护的WEB-INF下面的资源
    //mv.setViewName("redirect:/WEB-INF/view/show.jsp");
    return mv;
}

```



## 4.2 异常处理

SpringMVC 框架处理异常的常用方式：使用 `@ExceptionHandler` 注解处理异常。

### 4.2.1 `@ExceptionHandler` 注解

使用注解 `@ExceptionHandler` 可以将一个方法指定为异常处理方法。该注解只有一个可选属性 `value`，为一个 `Class<?>` 数组，用于指定该注解的方法所要处理的异常类，即所要匹配的异常。

而被注解的方法，其返回值可以是 `ModelAndView`、`String`，或 `void`，方法名随意，方法参数可以是 `Exception` 及其子类对象、`HttpServletRequest`、`HttpServletResponse` 等。系统会自动为这些方法参数赋值。

对于异常处理注解的用法，也可以直接将异常处理方法注解于 `Controller` 之中。

#### (1) 自定义异常类

定义三个异常类：`NameException`、`AgeException`、`MyUserException`。其中 `MyUserException` 是另外两个异常的父类。

```
public class MyUserException extends Exception {  
  
    public MyUserException() {  
        super();  
        // TODO Auto-generated constructor stub  
    }  
  
    public MyUserException(String message) {  
        super(message);  
        // TODO Auto-generated constructor stub  
    }  
  
}
```

```
public class NameException extends MyUserException {  
  
    public NameException() {  
        super();  
        // TODO Auto-generated constructor stub  
    }  
  
    public NameException(String message) {  
        super(message);  
        // TODO Auto-generated constructor stub  
    }  
  
}
```

```

public class AgeException extends MyUserException {

    public AgeException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public AgeException(String message) {
        super(message);
        // TODO Auto-generated constructor stub
    }

}

```

## (2) 修改 Controller 抛出异常

```

@Controller
public class MyController {
    @RequestMapping(value="/some.do")
    public ModelAndView doSome(Integer age,String name) throws MyUserException{
        ModelAndView mv = new ModelAndView();
        if( !"zs".equals(name)){
            throw new NameException("姓名不正确!!!");
        }
        mv.addObject("myname", name);
        mv.addObject("myage",age);
        mv.setViewName("show");
        return mv;
    }
    /** 定义方法处理异常，在处理器类中定义处理异常的方法
     * 方法的上面加入@ExceptionHandler*/
    @ExceptionHandler(value=NameException.class)
    public ModelAndView doNameException(Exception ex){
        ModelAndView mv = new ModelAndView();
        mv.addObject("tips", "处理NameException");
        mv.addObject("ex",ex);
        mv.setViewName("nameError");
        return mv;
    }
}

```

## (3) 定义异常响应页面

定义三个异常响应页面。

```
<body>
  nameErrors page<br>
  <hr>
  ${ex.message }<br>
</body>
```

```
<body>
  ageErrors page<br>
  <hr>
  ${ex.message }<br>
</body>
```

```
<body>
  defaultErrors page<br>
  <hr>
  ${ex.message }<br>
</body>
```

不过，一般不这样使用。而是将异常处理方法专门定义在一个类中，作为全局的异常处理类。

需要使用注解@ControllerAdvice，字面理解就是“控制器增强”，是给控制器对象增强功能的。使用@ControllerAdvice 修饰的类中可以使用@ExceptionHandler。

当使用@RequestMapping 注解修饰的方法抛出异常时，会执行@ControllerAdvice 修饰的类中的异常处理方法。

@ControllerAdvice 是使用@Component 注解修饰的，可以<context:component-scan>扫描到@ControllerAdvice 所在的类路径(包名)，创建对象。

#### (4) 定义全局异常处理类

```
/**
 * @ControllerAdvice: 控制器增强,
 * 处理器类发生异常可以到当前类中找ExceptionHandler
 * 位置: 类的上面
 */
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(value=NameException.class)
    public ModelAndView doNameException(Exception ex){
        ModelAndView mv = new ModelAndView();
        mv.addObject("tips", "@ControllerAdvice使用注解处理NameException");
        mv.addObject("ex",ex);
        mv.setViewName("nameError");
        return mv;
    }
    @ExceptionHandler(value=AgeException.class)
    public ModelAndView doAgeException(Exception ex){
        ModelAndView mv = new ModelAndView();
        mv.addObject("tips", "@ControllerAdvice使用注解处理AgeException");
        mv.addObject("ex",ex);
        mv.setViewName("ageError");
        return mv;
    }
    //处理其他异常, NameException ,AgeException以外的异常
    @ExceptionHandler
    public ModelAndView doOtherException(Exception ex){
        ModelAndView mv = new ModelAndView();
        mv.addObject("tips", "@ControllerAdvice使用注解处理OtherException");
        mv.addObject("ex",ex);
        mv.setViewName("defaultError");
        return mv;
    }
}
```

#### (5) 定义 Spring 配置文件

```
<!-- 注册组件扫描器
      base-package:指定Controller注解所在的包名
-->
<context:component-scan base-package="com.bjpowernode.controllers" />

<!-- 注册组件扫描器
      base-package:指定@ControllerAdvice注解所在的包名 -->
<context:component-scan base-package="com.bjpowernode.exceptions" />

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

### 4.3 拦截器

SpringMVC 中的 Interceptor 拦截器是重要和相当有用的，它的主要作用是拦截指定

的用户请求，并进行相应的预处理与后处理。其拦截的时间点在“处理器映射器根据用户提交的请求映射出了所要执行的处理器类，并且也找到了要执行该处理器类的处理器适配器，在处理器适配器执行处理器之前”。当然，在处理器映射器映射出所要执行的处理器类时，已经将拦截器与处理器组合为了一个处理器执行链，并返回给了中央调度器。

### 4.3.1 一个拦截器的执行

项目：interceptor。

自定义拦截器

```
public class MyInterceptor implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("执行MyInterceptor ----- preHandle()---");
        return true;
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("执行MyInterceptor ----- postHandle()---");
    }

    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("执行MyInterceptor ----- afterCompletion()---");
    }

}
```

自定义拦截器，需要实现 HandlerInterceptor 接口。而该接口中含有三个方法：

- **preHandle(request,response, Object handler):**  
该方法在处理器方法执行之前执行。其返回值为 boolean，若为 true，则紧接着会执行处理器方法，且会将 afterCompletion()方法放入到一个专门的方法栈中等待执行。
- **postHandle(request,response, Object handler,modelAndView):**  
该方法在处理器方法执行之后执行。处理器方法若最终未被执行，则该方法不会执行。由于该方法是在处理器方法执行完后执行，且该方法参数中包含 ModelAndView，所以该方法可以修改处理器方法的处理结果数据，且可以修改跳转方向。
- **afterCompletion(request,response, Object handler, Exception ex):**  
当 preHandle()方法返回 true 时，会将该方法放到专门的方法栈中，等到对请求进行响应的所有工作完成之后才执行该方法。即该方法是在中央调度器渲染（数据填充）了响应页面之后执行的，此时对 ModelAndView 再操作也对响应无济于事。

**afterCompletion** 最后执行的方法，清除资源，例如在 Controller 方法中加入数据

```

@RequestMapping(value = "/some.do" )
public ModelAndView doSome(String name, Integer age, HttpSession session) {

    System.out.println("====执行MyController处理器方法====");
    ModelAndView mv = new ModelAndView();

    mv.addObject("myname", name);
    mv.addObject("myage", age);
    mv.setViewName("show");

    session.setAttribute("attr", "session中数据");

    return mv;
}

```

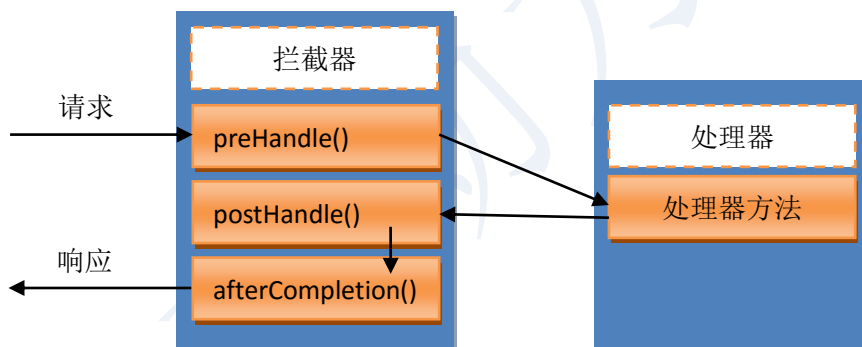
拦截器方法:

```

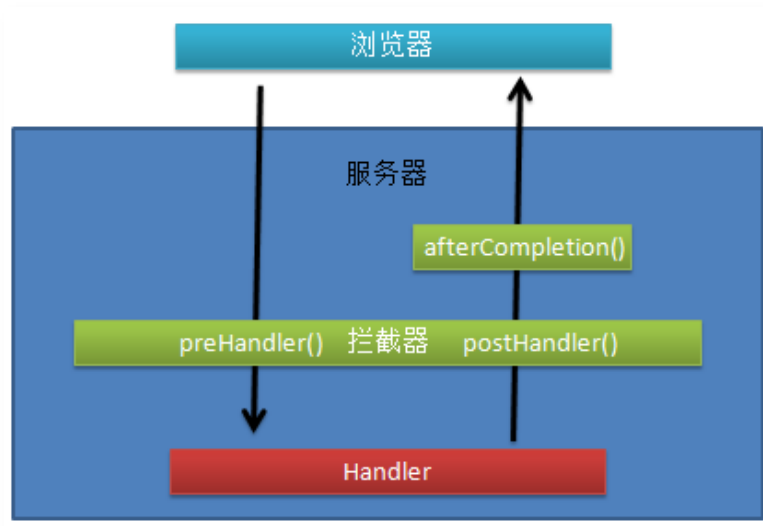
public void afterCompletion(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           Exception ex) throws Exception {
    System.out.println("执行了拦截器MyInterceptor的afterCompletion()方法");
    HttpSession session = request.getSession();
    Object attr = session.getAttribute("attr");
    System.out.println("attr删除之前===" + session.getAttribute("attr"));
    session.removeAttribute("attr");
    System.out.println("attr删除之后===" + session.getAttribute("attr"));
}

```

拦截器中方法与处理器方法的执行顺序如下图:



换一种表现方式，也可以这样理解：



### (1) 注册拦截器

```

<!-- 注册组件扫描器 -->
<context:component-scan base-package="com.bjpowernode.*"/>

<!-- 注册拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.bjpowernode.interceptors.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>

</beans>

```

`<mvc:mapping/>`用于指定当前所注册的拦截器可以拦截的请求路径，而`/**`表示拦截所有请求。

### (2) 修改 index 页面

```

<body>
    index page
</body>

```

### (3) 修改处理器

```
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value="/some.do")
    public ModelAndView doSome() {
        System.out.println("执行处理器方法");
        return new ModelAndView("/WEB-INF/jsp/show.jsp");
    }
}
```

### (4) 修改 show 页面

```
<body>
    show page
</body>
```

### (5) 控制台输出结果

```
执行MyInterceptor ----- preHandle()---
执行处理器方法
执行MyInterceptor ----- postHandle()---
执行MyInterceptor ----- afterCompletion()---
```

## 4.3.2 多个拦截器的执行

项目：interceptor2。在项目 interceptor 基础上修改。



## (1) 再定义一个拦截器

```

public class MyInterceptor2 implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("执行MyInterceptor222 ----- preHandle()---");
        return true;
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("执行MyInterceptor222 ----- postHandle()---");
    }

    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("执行MyInterceptor222 ----- afterCompletion()---");
    }

}

```

## (2) 多个拦截器的注册与执行

```

<!-- 注册拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/*" />
        <bean class="com.bjpowernode.interceptors.MyInterceptor" />
    </mvc:interceptor>

    <mvc:interceptor>
        <mvc:mapping path="/*" />
        <bean class="com.bjpowernode.interceptors.MyInterceptor2" />
    </mvc:interceptor>
</mvc:interceptors>

```

## (3) 控制台执行结果

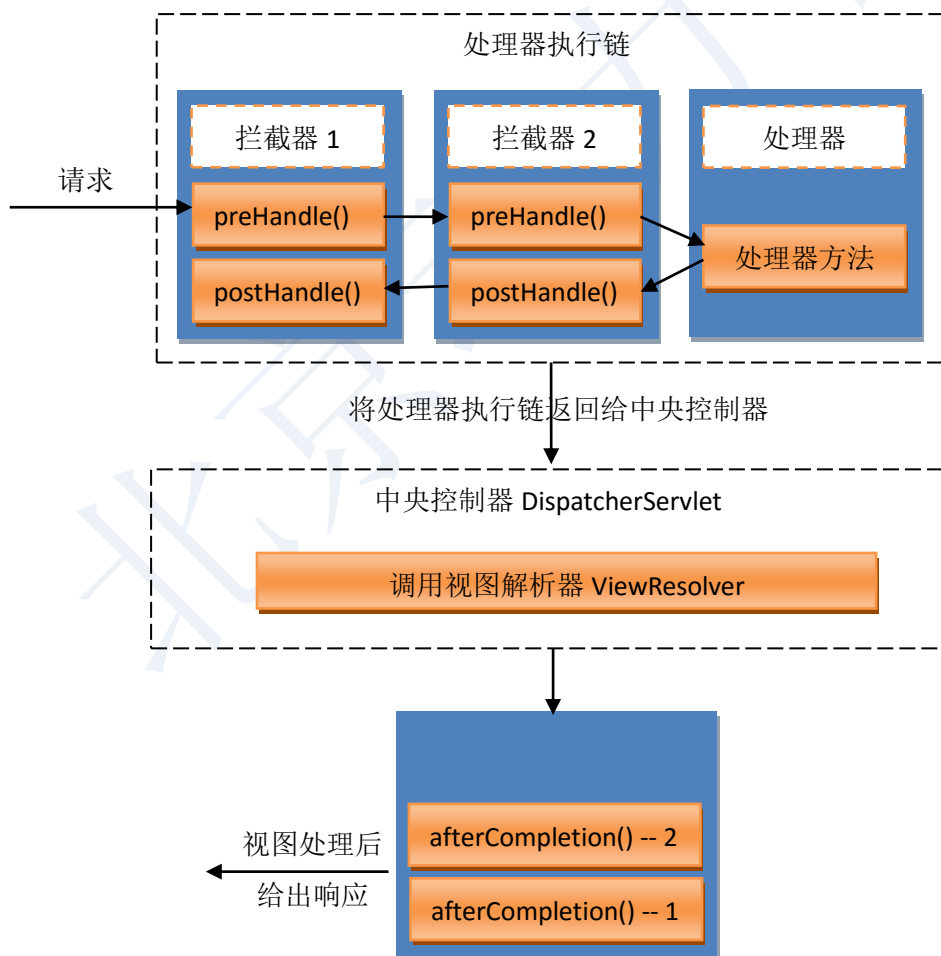
```

执行MyInterceptor ----- preHandle()---
执行MyInterceptor222 ----- preHandle()---
执行处理器方法
执行MyInterceptor222 ----- postHandle()---
执行MyInterceptor ----- postHandle()---
执行MyInterceptor222 ----- afterCompletion()---
执行MyInterceptor ----- afterCompletion()---

```

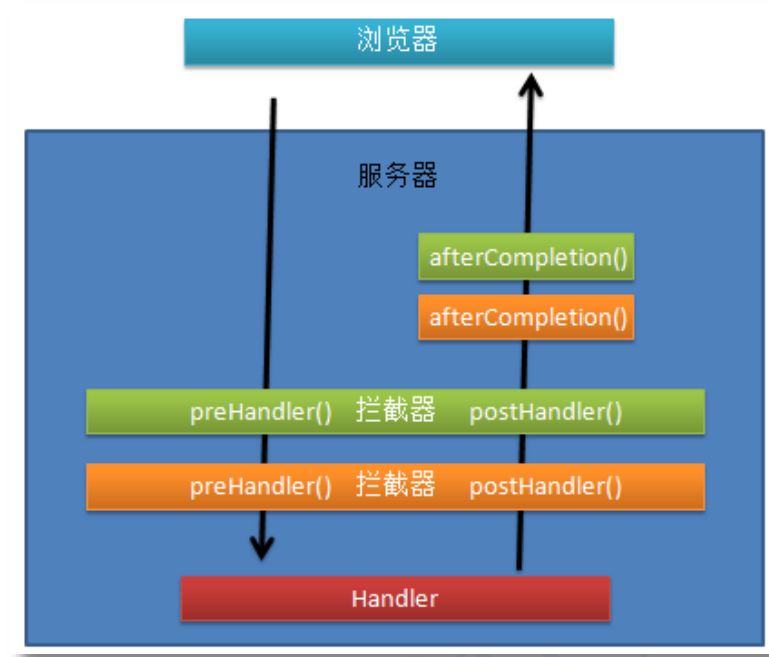
当有多个拦截器时，形成拦截器链。拦截器链的执行顺序，与其注册顺序一致。需要再次强调一点的是，当某一个拦截器的 `preHandle()` 方法返回 `true` 并被执行到时，会向一个专门的方法栈中放入该拦截器的 `afterCompletion()` 方法。

多个拦截器中方法与处理器方法的执行顺序如下图：



从图中可以看出，只要有一个 `preHandle()` 方法返回 `false`，则上部的执行链将被断开，其后续的处理方法方法与 `postHandle()` 方法将无法执行。但，无论执行链执行情况怎样，只要方法栈中有方法，即执行链中只要有 `preHandle()` 方法返回 `true`，就会执行方法栈中的 `afterCompletion()` 方法。最终都会给出响应。

换一种表现方式，也可以这样理解：



### 4.3.3 权限拦截器举例

只有经过登录的用户方可访问处理器，否则，将返回“无权访问”提示。

本例的登录，由一个 JSP 页面完成。即在该页面里将用户信息放入 `session` 中。也就是说，只要访问过该页面，就说明登录了。没访问过，则为未登录用户。

项目：interceptor\_permission。在项目 interceptor1 基础上修改。

#### (1) 修改 index 页面

```
<body>
  index page
</body>
```

## (2) 定义 Controller

```
@Controller
public class MyController {

    @RequestMapping(value="/system.do")
    public ModelAndView doSome() {
        System.out.println("欢迎进入系统");
        return new ModelAndView("/WEB-INF/jsp/welcome.jsp");
    }

}
```

## (3) 定义 welcome 页面

```
<body>
    欢迎进入系统!
</body>
```

## (4) 定义权限拦截器

当 preHandle()方法返回 false 时，需要使用 request 或 response 对请求进行响应。

```
public class PermissinoInterceptor implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("执行MyInterceptor ----- preHandle()---");
        String user = (String) request.getSession().getAttribute("user");
        if(!"beijing".equals(user)){
            request.getRequestDispatcher("/fail.jsp").forward(request, response);
            return false;
        }
        return true;
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("执行MyInterceptor ----- postHandle()---");
    }

    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("执行MyInterceptor ----- afterCompletion()---");
    }

}
```

## (5) 定义 fail 页面

```
<body>
    未登录，无权访问!
</body>
```

## (6) 注册权限拦截器

```
<!-- 注册 拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.bjpowernode.interceptors.PermissionInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

## (7) 定义 login 页面

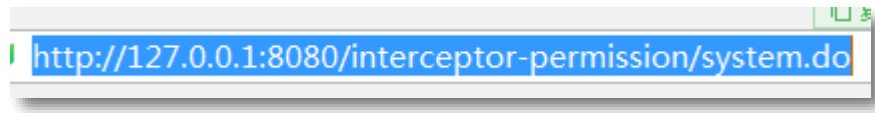
```
<body>
    <%                                用于模拟用户登录
        session.setAttribute("user", "bjpowernode");
    %>
    登录成功!
</body>
```

## (8) 定义 logout 页面

```
<body>
    <%                                模拟用户退出
        session.removeAttribute("user");
    %>
    已退出!
</body>
```

## (9) 项目测试

Step1: 在地址栏先直接提交 system.do 请求



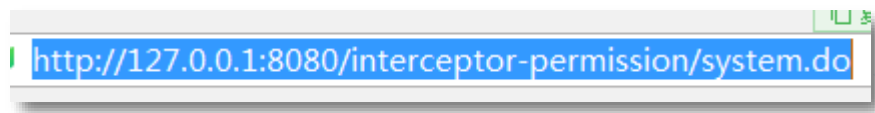
`http://127.0.0.1:8080/interceptor-permission/system.do`

Step2: 访问 login.jsp, 进行用户登录



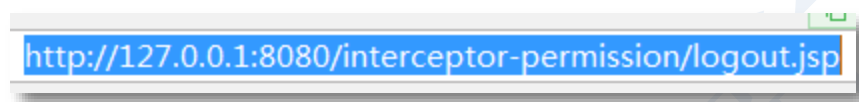
`http://127.0.0.1:8080/interceptor-permission/login.jsp`

Step3: 再次提交 system.do 请求




`http://127.0.0.1:8080/interceptor-permission/system.do`

Step4: 访问 logout.jsp, 进行用户退出



`http://127.0.0.1:8080/interceptor-permission/logout.jsp`

Step5: 三次提交 system.do 请求



`http://127.0.0.1:8080/interceptor-permission/system.do`