# SpringMVC

尚硅谷Java研究院

版本：V1.0

## 第1章 简介

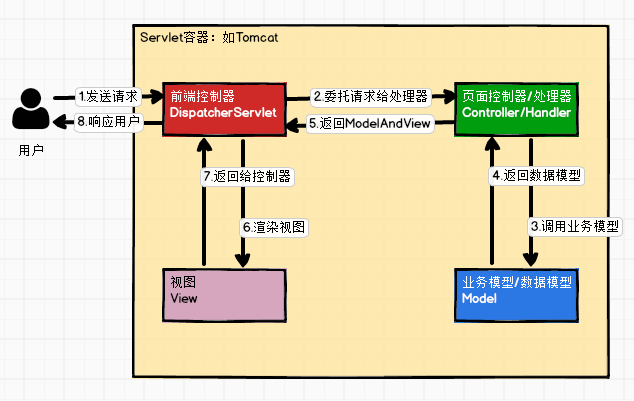
### 1.1 SpringMVC概述

Spring MVC属于SpringFrameWork的后续产品，是Spring 为展现层提供的基于 MVC 设计理念的优秀的 Web 框架，是目前最主流的MVC 框架。

Spring MVC 通过一套 MVC 注解，让 POJO 成为处理请求的控制器，而无须实现任何接口。

天生与Spring框架集成（如IOC容器、AOP等），而且支持REST 风格的 URL 请求。

### 1.2 SpringMVC处理请求的简单流程图



基本步骤：

1. 客户端请求提交到DispatcherServlet
2. 由DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller
3. DispatcherServlet将请求提交到Controller（也称为Handler）
4. Controller调用业务逻辑处理后，返回ModelAndView
5. DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图
6. 视图负责将结果显示到客户端

### 1.3 SpringMVC中的主要组件

1. DispatcherServlet：前端控制器
2. Controller：页面控制器/处理器，做的是MVC中的C的事情，但控制逻辑转移到前端控制器了，用于对请求进行处理。
3. HandlerMapping ：请求映射到处理器，找谁来处理，如果映射成功返回一个HandlerExecutionChain对象（包含一个Handler处理器(页面控制器)对象、多个HandlerInterceptor拦截器对象）
4. HandlerAdaptor：处理器适配器
5. View Resolver : 视图解析器，找谁来处理返回的页面。把逻辑视图解析为具体的View，进行这种策略模式，很容易更换其他视图技术：

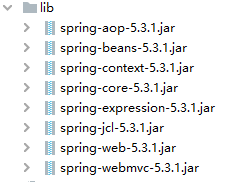
* 如InternalResourceViewResolver将逻辑视图名映射为JSP视图

1. MultipartResolver：文件上传解析器
2. HandlerExceptionResolver：异常处理器

### 1.4 HelloWorld

创建HelloWorld的步骤如下：

1. 创建Web工程，并导入以下jar包



1. 在web.xml文件中配置前端控制器DispatcheServlet

* 需要在servlet表情中配置一个初始化参数，指定SpringMVC的配置文件的路径和文件名。
* 如果初始化参数没有配置，SpringMVC就会按照默认的路径和文件名寻找。
  + 默认的路径和文件名为: /WEB-INF/<servlet-name>标签里配置的值-servlet.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0"**>  *<!--配置前端控制器-->* <**servlet**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置DispatcherServlet的初始化參數：设置SpringMVC文件的路径和文件名称 -->* <**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**>dispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

1. 在src目录下创建SpringMVC的配置配置文件，并命名为springmvc.xml
   1. 配置自动扫描的包
   2. 配置视图解析器

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.springmvc"**></**context:component-scan**>   *<!--配置视图解析器-->* <**bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  *<!--配置前缀-->* <**property name="prefix" value="/WEB-INF/views/"**></**property**>  *<!--配置后缀-->* <**property name="suffix" value=".jsp"**></**property**>  </**bean**> </**beans**> |

1. 在首页index.jsp页面中添加一个超链接

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**>  <**head**>  <**title**>首页</**title**>  </**head**>  <**body**>  <**a href="${**pageContext.request.contextPath**}/helloworld"**>Hello SpringMVC</**a**>  </**body**> </**html**> |

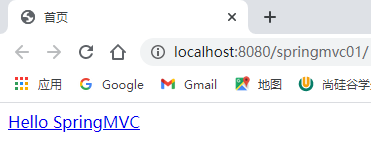
1. 创建控制器/处理器
   1. 在类上添加@Controller注解标识当前类是一个处理器
   2. 在方法上添加@RequestMapping注解设置该方法处理的请求路径

|  |
| --- |
| **package** com.atguigu.springmvc;  **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller **public class** HelloWorld {   @RequestMapping(value = **"/helloworld"**)  **public** String helloWorld(){  System.***out***.println(**"Hello SpringMVC!"**);  **return "success"**;  } } |

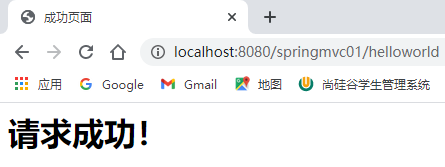
1. 在WEB-INF/views目录下创建视图页面success.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>成功页面</**title**> </**head**> <**body**>  <**h1**>请求成功！</**h1**> </**body**> </**html**> |

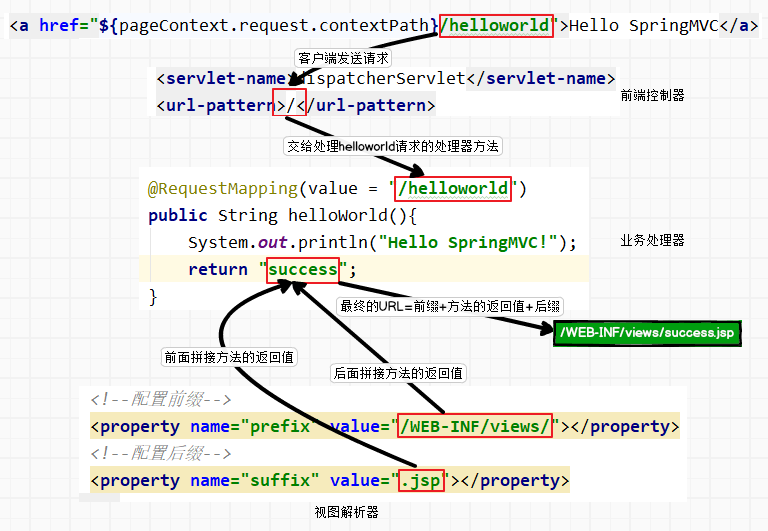
1. 启动Tomcat点击index.jsp页面中的超链接测试



1. 在浏览器中看到success.jsp页面中的内容



### 1.5 HelloWorld请求流程图解



## 第2章 @RequestMapping注解

### 2.1 通过@RequestMapping 注解映射请求

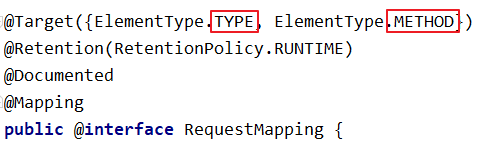
#### 2.1.1 概述

1. SpringMVC使用@RequestMapping注解为控制器指定可以处理哪些 URL 请求。
2. DispatcherServlet 截获请求后，就通过控制器上 @RequestMapping 提供的映射信息确定请求所对应的处理方法。

#### 2.1.2 @RequestMapping注解可以标注的位置

@RequestMapping注解可以添加到类上，也可以添加到方法上

1. 标记在类上：提供初步的请求映射信息。相对于WEB 应用的根目录
2. 标记在方法上：提供进一步的细分映射信息。相对于标记在类上的 URL。
3. @RequestMapping注解源码：



#### 2.1.3 @RequestMapping注解常用的属性

1. value属性
   1. 用来设置要映射的请求地址
   2. 值的类型是String类型的数组，所以可以同时设置多个映射的地址

|  |
| --- |
| @RequestMapping(value = {**"/testValue1"**,**"/testValue2"**}) **public** String testValue(){  System.***out***.println(**"测试@RequestMapping注解的value属性"**);  **return "success"**; } |

1. method属性
   1. 用来设置要映射的请求方式
   2. 值的类型是RequestMethod枚举类型的数组
   3. 请求方式有Get、Post、Put、Delete等

|  |
| --- |
| @RequestMapping(value = **"/testMethod"**,method = RequestMethod.***GET***) **public** String testMethod(){  System.***out***.println(**"测试@RequestMapping注解的method属性"**);  **return "success"**; } |

1. 可以使用 params 和 headers 来更加精确的映射请求，而且支持简单的表达式
   1. params属性（了解）
      1. 用来设置请求地址中必须包含的请求参数
      2. 值的类型是String类型的数组
      3. 各种表达式说明：
         1. param1: 表示请求必须包含名为 param1 的请求参数
         2. !param1: 表示请求不能包含名为 param1 的请求参数
         3. param1 != value1: 表示请求包含名为 param1 的请求参数，但其值不能为 value1
         4. {"param1=value1", "param2"}: 请求必须包含名为 param1 和param2 的两个请求参数，且 param1 参数的值必须为 value1

|  |
| --- |
| @RequestMapping(value = **"/testParams"**,method = RequestMethod.***GET***,params = {**"name=admin"**,**"age=18"**}) **public** String testParams(){  System.***out***.println(**"测试@RequestMapping注解的params属性"**);  **return "success"**; } |

* 1. headers属性（了解）
     1. 用来设置请求地址中必须包含的请求头
     2. 值的类型是String类型的数组

|  |
| --- |
| @RequestMapping(value = **"/testHeaders"**,method = RequestMethod.***GET***,  headers = **"Accept-Language=zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7"**) **public** String testHeaders(){  System.***out***.println(**"测试@RequestMapping注解的headers属性"**);  **return "success"**; } |

#### 2.1.4 @RequestMapping支持Ant 风格的路径（了解）

1. Ant 风格资源地址支持 3 种匹配符
   1. ?：匹配一个字符
   2. \*：匹配任意字符
   3. \*\*：匹配多层路径
2. 页面链接

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/testAnt/a/ant"**>Test Ant\_?</**a**><**br**> <**a href="${**pageContext.request.contextPath**}/testAnt/abc/ant"**>Test Ant\_\*</**a**><**br**> <**a href="${**pageContext.request.contextPath**}/testAnt/a/b/c/ant"**>Test Ant\_\*\*</**a**><**br**> |

1. 处理器方法

|  |
| --- |
| *// @RequestMapping("/testAnt/?/ant") // @RequestMapping("/testAnt/\*/ant")* @RequestMapping(**"/testAnt/\*\*/ant"**)  **public** String testAnt(){  System.***out***.println(**"测试Ant风格的URL"**);  **return "success"**;  } |

#### 2.1.5 @RequestMapping映射带占位符的URL

1. 带占位符的URL 是 Spring3.0 新增的功能，该功能在 SpringMVC 向 REST 目标挺进发展过程中具有里程碑的意义。
2. 通过 @PathVariable 可以将 URL 中占位符参数绑定到控制器处理方法的入参中，

URL 中的 {xxx} 占位符可以通过 @PathVariable("xxx") 绑定到处理方法的入参中。

1. @PathVariable注解主要用来处理REST风格的URL的中的请求参数。该注解中有以下三个属性：
   1. value属性：用来指定要映射的URL中的占位符的名字。
   2. name属性：与value功能一样，是value的一个别名。
   3. required属性：设置指定的占位符的名字是否是必须的，默认是true，如果不存在则抛出异常。
2. 页面连接

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/testPathVariable/1"**>Test PathVariable</**a**><**br**> |

1. 处理器方法

|  |
| --- |
| @RequestMapping(**"/testPathVariable/{id}"**) **public** String testPathVariable(@PathVariable(value = **"id"**,required = **false**) Integer id){  System.***out***.println(**"传入的id的值是："**+id);  **return "success"**; } |

### 2.2 REST

#### 2.2.1 概述

REST：即 Representational State Transfer，表现层状态转换。是Roy Thomas Fielding博士于2000年在他的博士论文中提出来的一种万维网软件架构风格，目的是便于不同软件/程序在网络（例如互联网）中互相传递信息。表现层状态转换是根基于超文本传输协议（HTTP）之上而确定的一组约束和属性，是一种设计提供万维网络服务的软件构建风格。

符合或兼容于这种架构风格（简称为 REST 或 RESTful）的网络服务，允许客户端发出以统一资源标识符访问和操作网络资源的请求。简单来说就是将服务器中的资源通过URI的方式来定位。

REST风格的URL将HTTP协议中的四种请求方式GET、POST、PUT、DELETE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。

#### 2.2.2 REST风格URL与传统URL的对比

|  |  |  |
| --- | --- | --- |
| 操作 | REST风格URL及请求方式 | 传统方式URL及请求方式 |
| 获取 | /getBookById/1 GET请求 | /getBookById?id=1 GET请求 |
| 添加 | /addBook PSOT请求 | /addBook POST请求 |
| 更新 | /updateBook PUT请求 | /updateBook POST请求 |
| 删除 | /deleteBookById/1 DELETE请求 | /deleteBookById/?id=1 GET请求 |

#### 2.2.3 如何发送PUT和DELETE请求

浏览器 form 表单只支持 GET 与 POST 请求，而PUT、DELETE 等请求方式并不

支持，Spring3.0 添加了一个过滤器HiddenHttpMethodFilter，可以将POST请求转换为PUT或DELETE请求。

转换PUT或DELETE请求的步骤：

1. 在web.xml配文件中配置过滤器 HiddenHttpMethodFilter。

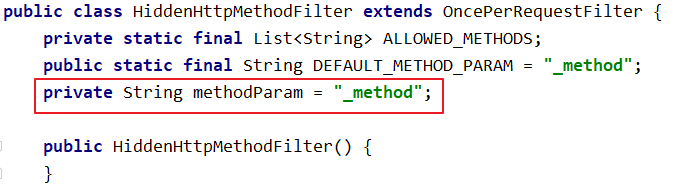
|  |
| --- |
| *<!--配置HiddenHttpMethodFilte过滤器，目的是为了将POST请求转换为PUT或DELETE请求-->* <**filter**>  <**filter-name**>hiddenHttpMethodFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.HiddenHttpMethodFilter</**filter-class**> </**filter**> <**filter-mapping**>  <**filter-name**>hiddenHttpMethodFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**> </**filter-mapping**> |

1. 在POST请求的form表单中添加一个\_method的请求参数

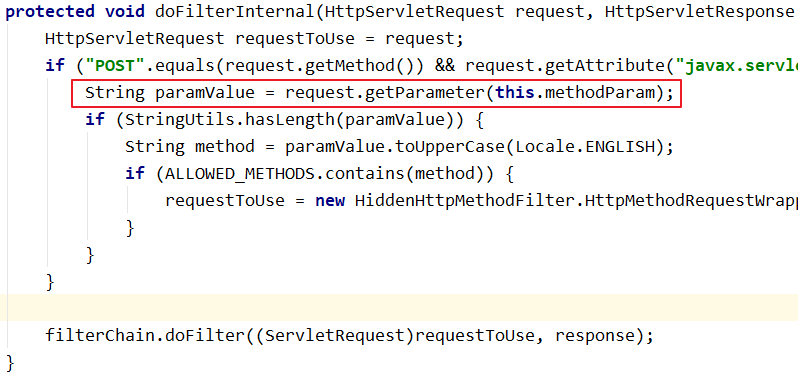
* \_method的值为put（忽略大小写）则转换为PUT请求
* \_method的值为delete（忽略大小写）则转换为DELETE请求

|  |
| --- |
| <**form action="${**pageContext.request.contextPath**}/testPut" method="post"**>  <**input type="hidden" name="\_method" value="put"**>  <**input type="submit" value="转换PUT请求"**> </**form**> <**form action="${**pageContext.request.contextPath**}/testDelete" method="post"**>  <**input type="hidden" name="\_method" value="delete"**>  <**input type="submit" value="转换DELETE请求"**> </**form**> |

1. \_method参数的由来，参照HiddenHttpMethodFilter底层源码
   1. 核心属性



* 1. 核心方法



#### 2.2.4 REST风格的CRUD操作

1. 页面内容

|  |
| --- |
| *<%--获取资源--%>* <**a href="${**pageContext.request.contextPath**}/testRest/1"**>Test REST\_get</**a**><**br**> *<%--添加资源--%>* <**form action="${**pageContext.request.contextPath**}/testRest" method="post"**>  <**input type="submit" value="REST\_post"**> </**form**> *<%--更新资源--%>* <**form action="${**pageContext.request.contextPath**}/testRest" method="post"**>  <**input type="hidden" name="\_method" value="put"**>  <**input type="hidden" name="id" value="1"**>  <**input type="submit" value="REST\_put"**> </**form**> *<%--删除资源--%>* <**form action="${**pageContext.request.contextPath**}/testRest/1" method="post"**>  <**input type="hidden" name="\_method" value="delete"**>  <**input type="submit" value="REST\_delete"**> </**form**> |

1. 处理器方法

|  |
| --- |
| *//测试REST风格获取操作* @RequestMapping(value = **"/testRest/{id}"**,method = RequestMethod.***GET***) **public** String testREST\_get(@PathVariable(**"id"**) Integer id){  System.***out***.println(**"传入的id的值是："**+id);  **return "success"**; }  *//测试REST风格添加操作* @RequestMapping(value = **"/testRest"**,method = RequestMethod.***POST***) **public** String testREST\_post(){  System.***out***.println(**"添加操作"**);  **return "success"**; }  *//测试REST风格更新操作* @RequestMapping(value = **"/testRest"**,method = RequestMethod.***PUT***) **public** String testREST\_put(){  System.***out***.println(**"更新操作"**);  **return "success"**; }  *//测试REST风格删除操作* @RequestMapping(value = **"/testRest/{id}"**,method = RequestMethod.***DELETE***) **public** String testREST\_delete(@PathVariable(**"id"**) Integer id){  System.***out***.println(**"传入的id的值是："**+id);  **return "success"**; } |

## 第3章 处理请求数据

Spring MVC 通过分析处理方法的签名(方法名+ 参数列表)，可以将HTTP请求信息绑定到处理方法的相应入参中，并根据方法的返回值类型做出相应的后续处理。

必要时可以对方法入参标注相应的注解（ @PathVariable 、@RequestParam、@RequestHeader、@CookieValue 等）。

### 3.1 @RequestParam注解

1. 在处理方法入参处使用 @RequestParam 可以把请求参数传递给请求方法。如果请求参数的参数名与处理方法的形参名一致，可以不加该注解。
2. @RequestParam中的属性：
   1. value属性：用来设置请求参数的参数名。
   2. required属性：用来设置该请求参数是否是必须的，默认是true。
   3. defaultValue属性：用来设置一个默认值，如果没有传入该请求参数将使用此值。

|  |
| --- |
| @RequestMapping(**"/testRequestParam"**) **public** String testRequestParam(@RequestParam(**"username"**) String username ,  @RequestParam(value = **"age"**,required = **false**,defaultValue = **"0"**) Integer age){  System.***out***.println(**"传入的用户名是："**+username);  System.***out***.println(**"传入的年龄是："**+age);  **return "success"**; } |

### 3.2 @RequestHeader注解

1. 每次发送请求时请求头包含了若干个属性，服务器可通过@RequestHeader注解获取客户端的信息，通过 @RequestHeader 即可将请求头中的属性值绑定到处理方法的入参中。
2. @RequestHeader中的属性
   1. value属性：用来设置请求头中的属性名。
   2. name属性：与value功能一样，是value的一个别名。
   3. required属性：用来设置该请求参数是否是必须的，默认是true。
   4. defaultValue属性：用来设置一个默认值，如果没有传入该请求参数将使用此值。

|  |
| --- |
| @RequestMapping(**"/testRequestHeader"**) **public** String testRequestHeader(@RequestHeader(**"Accept-Encoding"**) String encoding ,  @RequestHeader(**"Accept-Language"**) String language){  System.***out***.println(**"请求头中接受的压缩格式是："**+encoding);  System.***out***.println(**"请求头中接受的语言是："**+language);  **return "success"**; } |

### 3.3 @CookieValue注解

1. 通过 @CookieValue 即可将请求头中的Cookie对象的值绑定到处理方法的入参中。
2. @CookieValue中的属性
   1. value属性：用来设置请求头Cookie对象的名字。
   2. name属性：与value功能一样，是value的一个别名。
   3. required属性：用来设置该请求参数是否是必须的，默认是true。
   4. defaultValue属性：用来设置一个默认值，如果没有传入该请求参数将使用此值。

|  |
| --- |
| @RequestMapping(**"/testCookieValue"**) **public** String testCookieValue(@CookieValue(**"JSESSIONID"**) String jsessionId){  System.***out***.println(**"Cookie对象的value值是："**+jsessionId);  **return "success"**; } |

### 3.4 使用POJO作为入参

1. Spring MVC 会按请求参数名和 POJO 属性名进行自动匹配，自动为该对象填充属性值。而且支持级联属性赋值。
2. 必须要保证请求参数名与POJO的属性名保持一致。
3. 操作流程：
   1. 创建POJO类
      1. Employee类

|  |
| --- |
| **package** com.atguigu.springmvc.entities;  **public class** Employee {   **private** Integer **id**;  **private** String **lastName**;   **private** String **email**;    **private** Department **department**;    **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** Department getDepartment() {  **return department**;  }   **public void** setDepartment(Department department) {  **this**.**department** = department;  }   @Override  **public** String toString() {  **return "Employee [id="** + **id** + **", lastName="** + **lastName** + **", email="** + **email** + **", department="** + **department** + **"]"**;  }     } |

* + 1. Department类

|  |
| --- |
| **package** com.atguigu.springmvc.entities;  **public class** Department {   **private** Integer **id**;  **private** String **departmentName**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getDepartmentName() {  **return departmentName**;  }   **public void** setDepartmentName(String departmentName) {  **this**.**departmentName** = departmentName;  }   @Override  **public** String toString() {  **return "Department [id="** + **id** + **", departmentName="** + **departmentName** + **"]"**;  }   } |

* 1. 创建表单

|  |
| --- |
| <**h1**>处理入参为POJO</**h1**> <**form action="${**pageContext.request.contextPath**}/testPOJO" method="post"**>  员工编号：<**input type="text" name="id"**><**br**>  员工姓名：<**input type="text" name="lastName"**><**br**>  员工邮箱：<**input type="text" name="email"**><**br**>  员工部门编号：<**input type="text" name="department.id"**><**br**>  员工部门名称：<**input type="text" name="department.departmentName"**><**br**>  <**input type="submit" value="Test\_pojo"**> </**form**> |

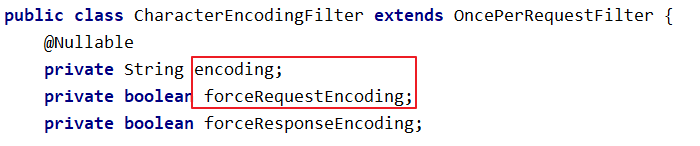
* 1. 创建处理方法

|  |
| --- |
| *//测试入参为POJO* @RequestMapping(**"/testPOJO"**) **public** String testPOJO(Employee employee){  System.***out***.println(**"员工信息是："**+employee);  **return "success"**; } |

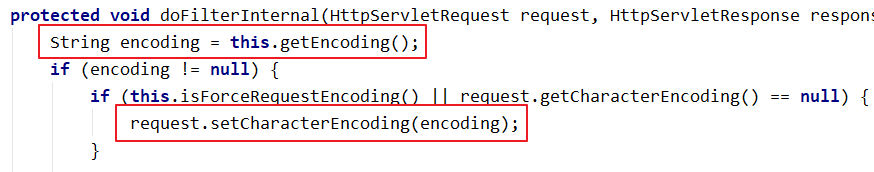
* 1. 如果要添加中文需要在web.xml中配置过滤器CharacterEncodingFilter
     1. 设置过滤器的encoding属性值为utf-8
     2. 设置过滤器的forceRequestEncoding属性值为true

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0"**>  *<!--配置CharacterEncodingFilter过滤器，解决POST请求中文乱码问题 -->* <**filter**>  <**filter-name**>encodingFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.CharacterEncodingFilter</**filter-class**>  *<!--设置属性encoding的值-->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>utf-8</**param-value**>  </**init-param**>  *<!--设置属性forceRequestEncoding的值-->* <**init-param**>  <**param-name**>forceRequestEncoding</**param-name**>  <**param-value**>true</**param-value**>  </**init-param**>  </**filter**>  <**filter-mapping**>  <**filter-name**>encodingFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**>  *<!--配置前端控制器-->* <**servlet**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置DispatcherServlet的初始化參數：设置SpringMVC文件的路径和文件名称 -->* <**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**>dispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**>    *<!--配置HiddenHttpMethodFilte过滤器，目的是为了将POST请求转换为PUT或DELETE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**> </**web-app**> |

* 1. CharacterEncodingFilter底层源码
     1. 核心属性



* + 1. 核心方法

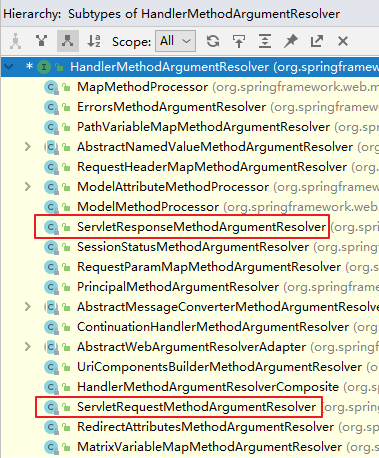


### 3.5 使用原生Servlet-API作为入参

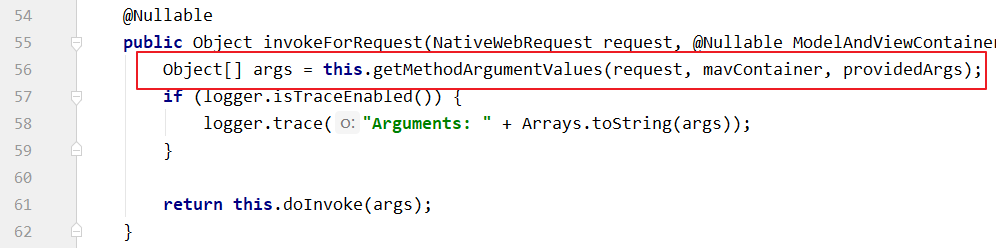
1. 不使用@RequestParam注解和入参为POJO同样可以获取请求参数，可以使用原生的Servlet API作为入参进行操作。
2. SpringMVC处理器方法中接受的原生Servlet API有以下9个：
   * HttpServletRequest
   * HttpServletResponse
   * HttpSession
   * java.security.Principal
   * Locale
   * InputStream
   * OutputStream
   * Reader
   * Writer
3. 处理方法

|  |
| --- |
| *//测试入参为Servlet API* @RequestMapping(**"/testServletAPI"**) **public** String testServletAPI(HttpServletRequest request){  *//获取员工编号* String id = request.getParameter(**"id"**);  *//获取员工姓名* String lastName = request.getParameter(**"lastName"**);  System.***out***.println(**"员工的编号为："**+id);  System.***out***.println(**"员工的姓名为："**+lastName);  **return "success"**; } |

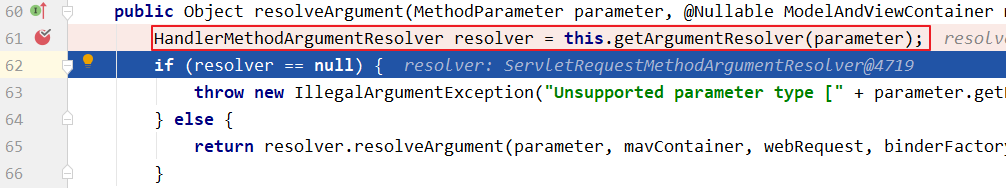
1. 底层源码
   * 解析器实现类



* InvocableHandlerMethod类的第56行



* HandlerMethodArgumentResolverComposite类的第61行



## 第4章 处理响应数据

SpringMVC提供了以下两种种途径输出模型数据：

1. ModelAndView: 处理方法返回值类型为 ModelAndView 时, 方法体即可通过该对象添加模型数据，最终会添加到request域中。
2. Map 或 Model或ModelMap: 入参为java.uti.Map、 org.springframework.ui.Model或org.springframework.ui.ModelMap 时，处理方法返回时，Map 中的数据会自动添加到模型中，并最终添加到request域中。

### 4.1 处理响应数据之ModelAndView

#### 4.1.1 ModelAndView简介

处理器方法的返回值如果为 ModelAndView，则其既包含视图信息，也包含模型数据信息。

1. 核心属性
   * private Object view; 视图信息
   * private ModelMap model; 模型数据
2. 添加模型数据的方法
   * MoelAndView addObject(String attributeName, Object attributeValue)
   * ModelAndView addAllObject(Map<String, ?> modelMap)
3. 获取模型数据的方法
   * protected Map<String, Object> getModelInternal()
   * public ModelMap getModelMap()
   * public Map<String, Object> getModel()
4. 设置视图的方法
   * void setView(View view) 设置视图对象
   * void setViewName(String viewName) 设置视图名字

#### 4.1.2 处理器方法返回值设置为ModelAndView

1. 处理器方法

|  |
| --- |
| *//处理响应数据之ModelAndView* @RequestMapping(**"/testModelAndView"**) **public** ModelAndView testModelAndView(){  *//创建ModelAndView对象* ModelAndView mv = **new** ModelAndView();  *//添加响应数据* mv.addObject(**"user"**,**"admin"**);  *//设置视图名* mv.setViewName(**"success"**);  **return** mv; } |

1. 成功页面内容

|  |
| --- |
| <**body**>  <**h1**>请求成功！</**h1**>  <**h1**>Request域中的属性值是：**${**requestScope.user**}**</**h1**> </**body**> |

1. 成功页面展示

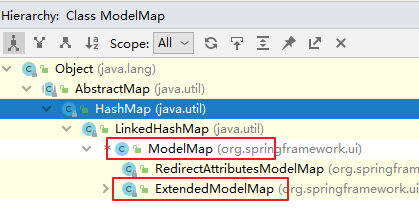


### 4.2 处理响应数据之Map、Model或ModelMap

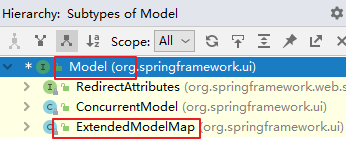
#### 4.2.1 简介

Spring MVC 在内部使用了一个 org.springframework.ui.Model 接口存储模型数据，具体使用步骤：

1. Spring MVC 在调用方法前会创建一个隐含的模型对象作为模型数据的存储容器。
2. 如果方法的入参为 Map 或 Model 类型，Spring MVC 会将隐含模型的引用传递给这些入参。
3. 在方法体内，开发者可以通过这个入参对象访问到模型中的所有数据，也可以向模型中添加新的属性数据。
4. Map、Model、ModelMap三者的关系：
   * ModelMap继承了LinkedHashMap



* + ExtendedModelMap继承了ModelMap实现了Model接口

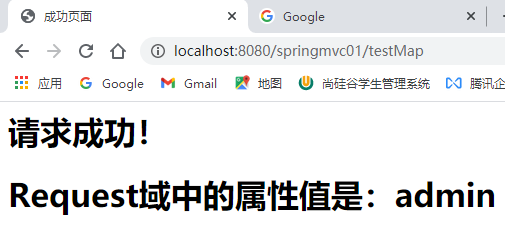


#### 4.2.2 处理器方法入参为Map

1. 处理器方法

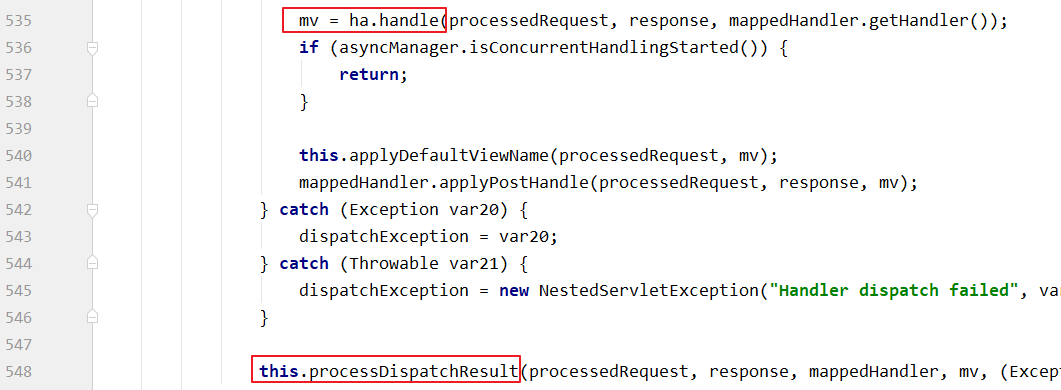
|  |
| --- |
| @RequestMapping(**"/testMap"**) **public** String testMap(Map<String, String> map) {  *//将模型数据放到map中，最终会放到request域中* map.put(**"user"**,**"admin"**);  **return "success"**; } |

1. 最终数据也是放到了request域中

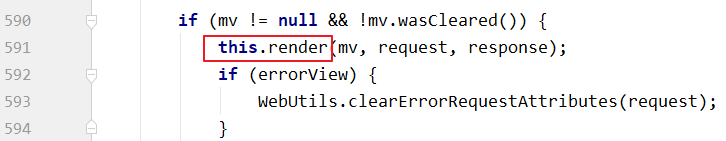


### 4.3 处理响应数据底层源码

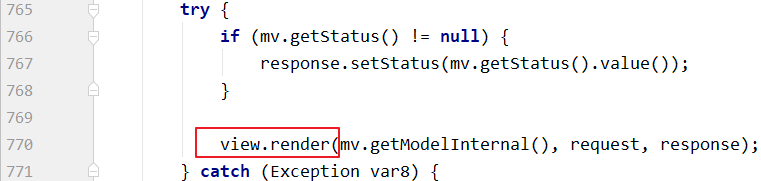
1. 在**DispacherServlet类**的doDispatch方法中：
   1. 在535行调用目标方法（处理器中我们创建的方法）得到ModelAndView对象。
   2. 在548行调用processDispatchResult方法处理结果。



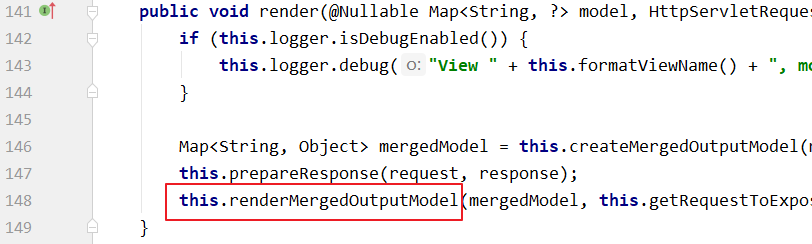
* 1. 在591行调用render方法开始渲染视图。



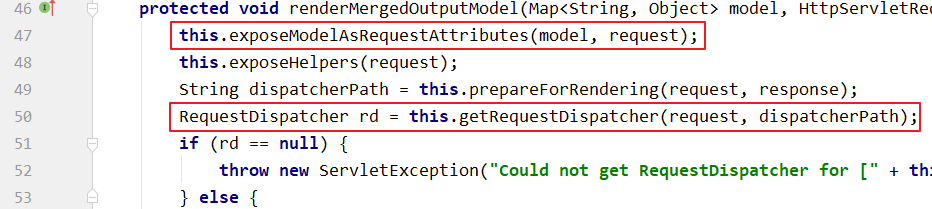
* 1. 在render方法里通过视图解析器解析视图名得到视图对象，然后在770行真正渲染视图。



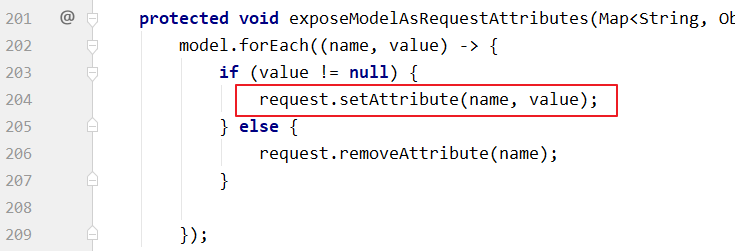
1. 在**InternalResourceView类**的render方法（继承的**AbstractView类**的render方法）的148行处理模型数据。



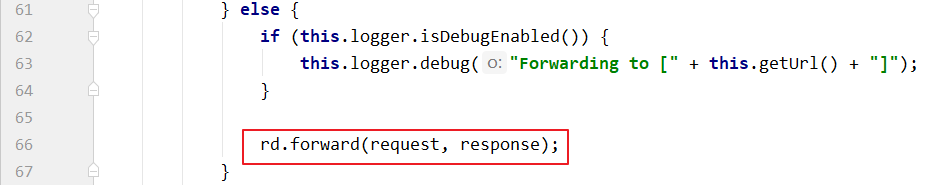
1. 在InternalResourceView类的renderMergedOutputModel方法中：
   1. 在47行将模型数据放到了request域中；在50行获取了转发器。



* 1. 在exposeModelAsRequestAttributes方法（继承的AbstractView类的方法）的204行将模型数据放到了request域中。



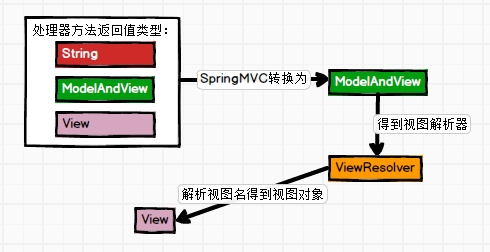
* 1. 在66行进行请求的转发



## 第5章 视图解析

### 5.1 SpringMVC如何解析视图

1. 不论处理器方法返回一个String，ModelAndView还是View，Spring MVC 都会在内部将它们转换成一个 ModelAndView 对象，由视图解析器解析视图，然后，进行页面的跳转。
2. Spring MVC 借助视图解析器（ViewResolver）得到最终的视图对象（View），最终的视图可以是 JSP ，也可能是 Excel、JFreeChart等各种表现形式的视图。
3. 对于最终究竟采取何种视图对象对模型数据进行渲染，处理器并不关心，处理器工作重点聚焦在生产模型数据的工作上，从而实现 MVC 的充分解耦。



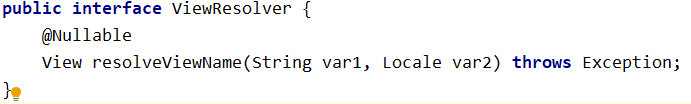
### 5.2 视图解析器

#### 5.2.1 简介

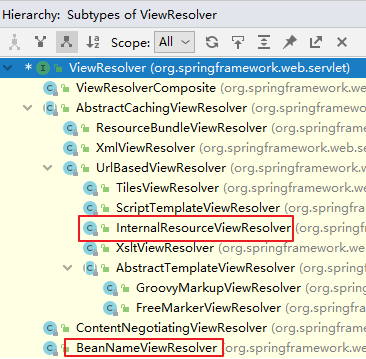
1. 视图解析器的作用比较单一：将逻辑视图解析为一个具体的视图对象。
2. 所有的视图解析器都必须实现 ViewResolver 接口
3. 可以在 SpringMVC 上下文中配置一种或多种解析器，每个视图解析器都实现了 Ordered 接口并开放出一个 order 属性，可以通过 order 属性指定解析器的优先顺序，order 越小优先级越高。
4. SpringMVC 会按视图解析器顺序的优先顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则将抛出 ServletException 异常。
5. JSP 是最常见的视图技术，可以使用 InternalResourceViewResolve作为视图解析器。

#### 5.2.2 接口和实现类

1. 视图解析器接口



1. 视图解析器实现类



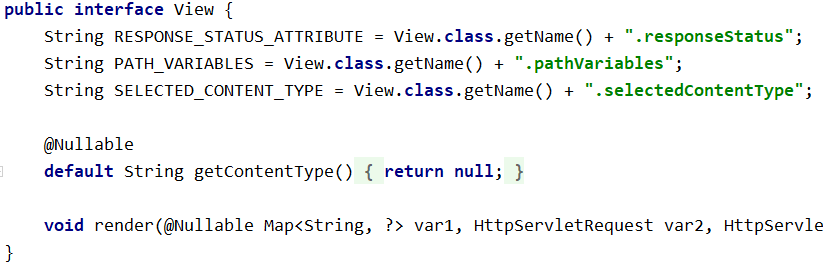
### 5.3 视图

#### 5.3.1 简介

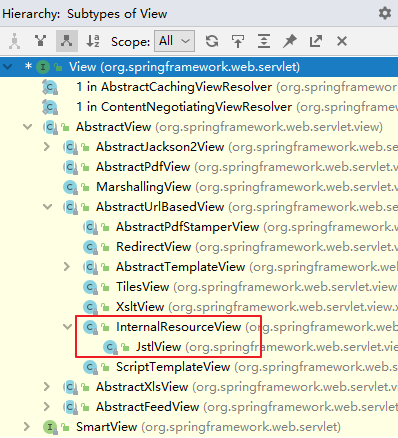
1. 视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。
2. 视图对象由视图解析器负责实例化。由于视图是无状态的，所以他们不会有线程安全的问题。
3. 为了实现视图模型和具体实现技术的解耦，Spring 在org.springframework.web.servlet 包中定义了一个高度抽象的 View 接口。

#### 5.3.2 接口和实现类

1. 视图接口



1. 视图实现类



1. 若要使用JstlView，只需要在Web工程中导入jstl的jar包即可。

### 5.4 mvc:view-controller标签

如果希望不通过处理器的方法直接响应SpringMVC渲染的页面，则可以使用 mvc:view-controller 标签实现。

步骤如下：

1. 在SpringMVC的配置文件中配置mvc:view-controller标签

|  |
| --- |
| *<!--**配置不经过处理器方法直接响应的页面-->* <**mvc:view-controller path="/testViewController" view-name="success"**></**mvc:view-controller**> |

1. 配置了以上标签之后会导致其他请求路径都失效，还需要配置以下标签（后面的内容再讲述具体原因），而且在实际的开发中我们也都要配置以下标签。

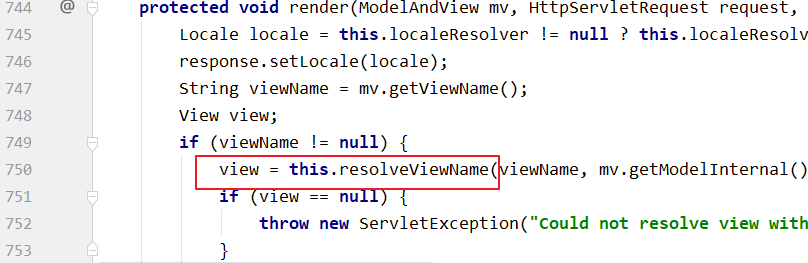
|  |
| --- |
| <**mvc:annotation-driven**></**mvc:annotation-driven**> |

### 5.5 重定向

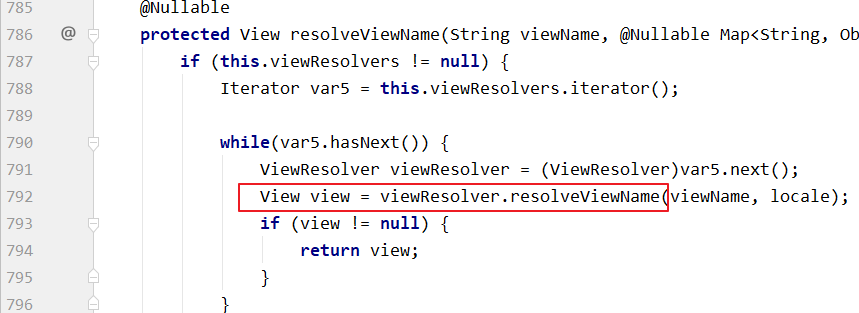
1. 一般情况下，处理器方法返回字符串类型的值会被当成逻辑视图名处理。如果返回的字符串中带 forward: 或 redirect: 前缀时，SpringMVC 会对他们进行特殊处理：将 forward: 和 redirect: 当成指示符，其后的字符串作为 URL 来处理。
   * redirect:/prefix\_test.jsp：会完成一个到 prefix\_test.jsp 的重定向的操作。
   * forward:/prefix\_test.jsp：会完成一个到 prefix\_test.jsp 的转发操作。
2. 处理器方法

|  |
| --- |
| @RequestMapping(**"/testRedirect"**)  **public** String testRedirect(){  System.***out***.println(**"测试重定向"**); *// return "forward:/prefix\_test.jsp"; //转发到prefix\_test.jsp页面* **return "forward:/prefix\_test.jsp"**; *//重定向到prefix\_test.jsp页面* } |

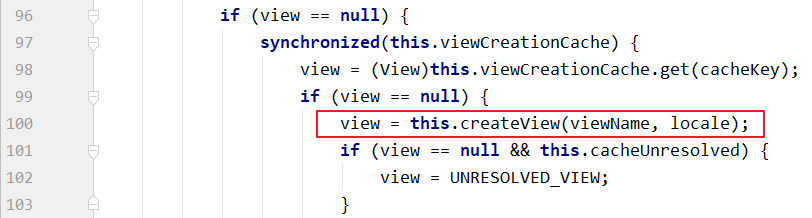
1. 前缀判断底层源码
   * 在DispatcherServlet类：
     + 在render方法的750行



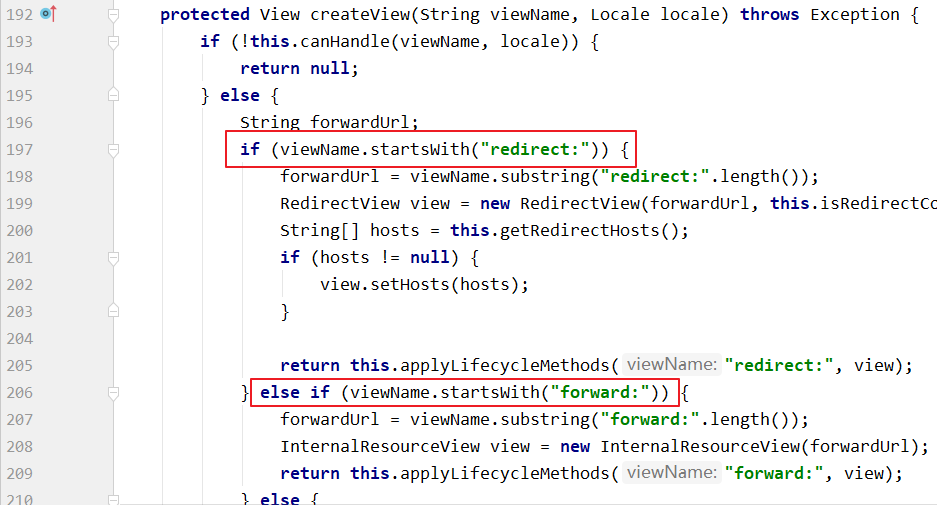
* + - 在resolveViewName方法的792行



* + 在抽象类AbstractCachingViewResolver的resoveViewName方法的100行



* + 在UrlBasedViewResolver类的createView方法的197行和206行

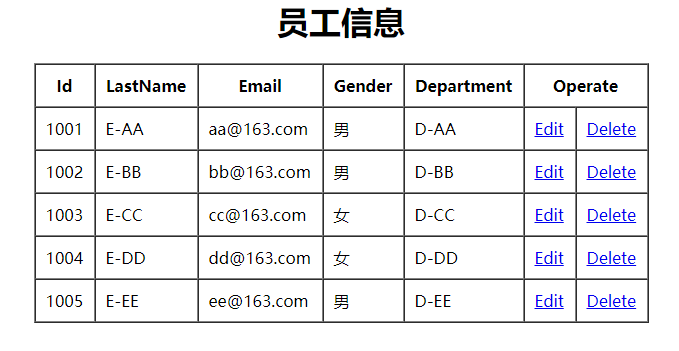


## 第6章 综合案例：RESTful\_CRUD

### 6.1 需求分析

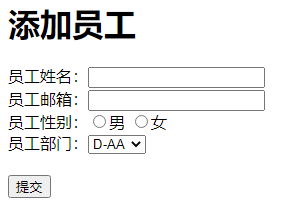
#### 6.1.1 获取所有员工

1. URL：getEmployees
2. 请求方式：GET
3. 显示效果



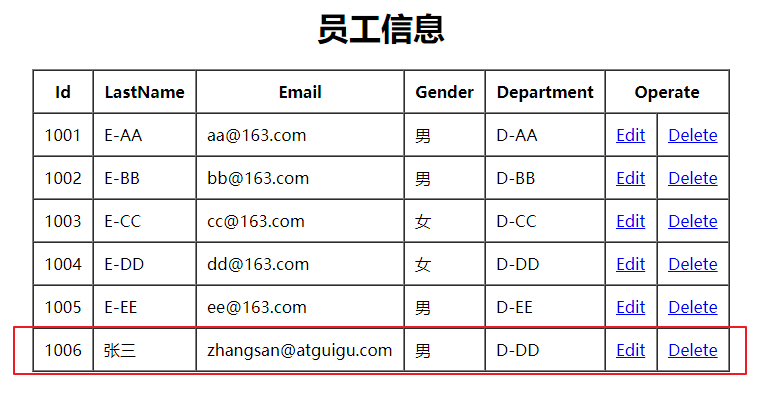
#### 6.1.2 添加操作-去添加员工的页面

1. URL：addEmployee
2. 请求方式：GET
3. 使用SpringMVC的form标签
4. 显示效果



#### 6.1.3 添加操作-添加员工

1. URL：addEmployee
2. 请求方式：POST
3. 添加成功之后重定向到查询所有员工的URL
4. 显示效果



#### 6.1.4 删除员工

1. URL：deleteEmployeeById/{id}
2. 请求方式：DELETE
3. 点击删除按钮弹出提示框
4. 删除成功之后同样重定向到查询所有员工的URL
5. 显示效果：对应的员工信息从数据库中删除

#### 6.1.5 修改操作-去修改员工的页面

1. URL：updateEmployee/{id}
2. 请求方式：GET
3. 根据员工的id从数据库中查询员工信息
4. 员工id不能修改，设置为只读
5. 显示效果



#### 6.1.6 修改操作-更新员工

1. URL：updateEmployee
2. 请求方式：PUT
3. 更新成功之后同样重定向到查询所有员工的URL
4. 显示效果：对应的员工信息被更新

#### 6.1.7 相关类

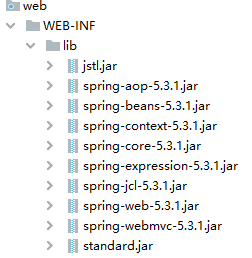
1. 实体类：Employee、Department
2. 处理器：EmployeeHandler
3. Dao：EmployeeDao、DepartmentDao
4. 为了教学方便，省略了Service层，也省略了连接数据库，操作的数据都设置在了Dao里。

#### 6.1.8 相关页面

1. 显示所有员工的页面：list.jsp
2. ~~添加员工的页面：add.jsp~~
3. ~~修改员工的页面：update.jsp~~
4. 添加或修改员工的页面：input.jsp

### 6.2 环境搭建

#### 6.2.1 创建web工程，导入以下jar包



#### 6.2.2 配置web.xml文件

1. 配置过滤器CharacterEncodingFilter
2. 配置前端控制器DispacherServlet
3. 配置过滤器HiddenHttpMethodFilter

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0"**>  *<!--配置CharacterEncodingFilter过滤器，解决POST请求中文乱码问题  注意：该过滤器一定要配置到最上面  -->* <**filter**>  <**filter-name**>encodingFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.CharacterEncodingFilter</**filter-class**>  *<!--设置属性encoding的值-->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>utf-8</**param-value**>  </**init-param**>  *<!--设置属性forceRequestEncoding的值-->* <**init-param**>  <**param-name**>forceRequestEncoding</**param-name**>  <**param-value**>true</**param-value**>  </**init-param**>  </**filter**>  <**filter-mapping**>  <**filter-name**>encodingFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**>  *<!--配置前端控制器-->* <**servlet**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置DispatcherServlet的初始化參數：设置SpringMVC文件的路径和文件名称 -->* <**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**>dispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**>   *<!--配置HiddenHttpMethodFilte过滤器，目的是为了将POST请求转换为PUT或DELETE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**> </**web-app**> |

#### 6.2.3 创建SpringMVC的配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.springmvc.crud"**></**context:component-scan**>   *<!--配置视图解析器-->* <**bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  *<!--配置前缀-->* <**property name="prefix" value="/WEB-INF/views/"**></**property**>  *<!--配置后缀-->* <**property name="suffix" value=".jsp"**></**property**>  </**bean**>  </**beans**> |

#### 6.2.4 创建实体类和Dao

1. 实体类
   1. Employee

|  |
| --- |
| **package** com.atguigu.springmvc.crud.entities;  **public class** Employee {   **private** Integer **id**;  **private** String **lastName**;   **private** String **email**;  *//1 male, 0 female* **private** Integer **gender**;    **private** Department **department**;    **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** Department getDepartment() {  **return department**;  }   **public void** setDepartment(Department department) {  **this**.**department** = department;  }   **public** Employee(Integer id, String lastName, String email, Integer gender,  Department department) {  **super**();  **this**.**id** = id;  **this**.**lastName** = lastName;  **this**.**email** = email;  **this**.**gender** = gender;  **this**.**department** = department;  }   **public** Employee() {  }   @Override  **public** String toString() {  **return "Employee [id="** + **id** + **", lastName="** + **lastName** + **", email="** + **email** + **", gender="** + **gender** + **", department="** + **department** + **"]"**;  }     } |

* 1. Department

|  |
| --- |
| **package** com.atguigu.springmvc.crud.entities;  **public class** Department {   **private** Integer **id**;  **private** String **departmentName**;   **public** Department() {  }    **public** Department(**int** i, String string) {  **this**.**id** = i;  **this**.**departmentName** = string;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getDepartmentName() {  **return departmentName**;  }   **public void** setDepartmentName(String departmentName) {  **this**.**departmentName** = departmentName;  }   @Override  **public** String toString() {  **return "Department [id="** + **id** + **", departmentName="** + **departmentName** + **"]"**;  }   } |

1. Dao
   1. EmployeeDao

|  |
| --- |
| **package** com.atguigu.springmvc.crud.dao;  **import** com.atguigu.springmvc.crud.entities.Department; **import** com.atguigu.springmvc.crud.entities.Employee; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Repository;  **import** java.util.Collection; **import** java.util.HashMap; **import** java.util.Map;  @Repository **public class** EmployeeDao {   **private static** Map<Integer, Employee> *employees* = **null**;    @Autowired  **private** DepartmentDao **departmentDao**;    **static**{  *employees* = **new** HashMap<Integer, Employee>();   *employees*.put(1001, **new** Employee(1001, **"E-AA"**, **"aa@163.com"**, 1, **new** Department(101, **"D-AA"**)));  *employees*.put(1002, **new** Employee(1002, **"E-BB"**, **"bb@163.com"**, 1, **new** Department(102, **"D-BB"**)));  *employees*.put(1003, **new** Employee(1003, **"E-CC"**, **"cc@163.com"**, 0, **new** Department(103, **"D-CC"**)));  *employees*.put(1004, **new** Employee(1004, **"E-DD"**, **"dd@163.com"**, 0, **new** Department(104, **"D-DD"**)));  *employees*.put(1005, **new** Employee(1005, **"E-EE"**, **"ee@163.com"**, 1, **new** Department(105, **"D-EE"**)));  }    **private static** Integer *initId* = 1006;    **public void** save(Employee employee){  **if**(employee.getId() == **null**){  employee.setId(*initId*++);  }    employee.setDepartment(**departmentDao**.getDepartment(employee.getDepartment().getId()));  *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);  } } |

* 1. DepartmentDao

|  |
| --- |
| **package** com.atguigu.springmvc.crud.dao;  **import** com.atguigu.springmvc.crud.entities.Department; **import** org.springframework.stereotype.Repository;  **import** java.util.Collection; **import** java.util.HashMap; **import** java.util.Map;  @Repository **public class** DepartmentDao {   **private static** Map<Integer, Department> *departments* = **null**;    **static**{  *departments* = **new** HashMap<Integer, Department>();    *departments*.put(101, **new** Department(101, **"D-AA"**));  *departments*.put(102, **new** Department(102, **"D-BB"**));  *departments*.put(103, **new** Department(103, **"D-CC"**));  *departments*.put(104, **new** Department(104, **"D-DD"**));  *departments*.put(105, **new** Department(105, **"D-EE"**));  }    **public** Collection<Department> getDepartments(){  **return** *departments*.values();  }    **public** Department getDepartment(Integer id){  **return** *departments*.get(id);  }   } |

1. 熟悉EmployeeDao和DepartmentDao中的方法

### 6.3 功能实现

#### 6.3.1 获取所有员工

1. 在首页中创建超链接

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/getEmployees"**>获取所有员工</**a**> |

1. 创建处理器类EmployeeHandler
   1. 注入EmployeeDao
   2. 创建处理方法

|  |
| --- |
| **package** com.atguigu.springmvc.crud.handler;  **import** com.atguigu.springmvc.crud.dao.EmployeeDao; **import** com.atguigu.springmvc.crud.entities.Employee; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.RequestMapping; **import** org.springframework.web.bind.annotation.RequestMethod;  **import** java.util.Collection; **import** java.util.Map;  @Controller **public class** EmployeeHandler {   @Autowired  **private** EmployeeDao **employeeDao**;   @RequestMapping(value = **"getEmployees"**,method = RequestMethod.***GET***)  **public** String getEmployees(Map<String , Object> map){  *//调用EmployeeDao中获取所有员工的方法* Collection<Employee> employees = **employeeDao**.getAll();  *//将employees放到map中* map.put(**"emps"**,employees);  **return "list"**;  } } |

1. 在WEB-INF/views目录下创建视图页面list.jsp
   1. 从request域中获取所有的员工
   2. 使用jstl的if标签判断是否为空
      1. 为空显示没有任何员工
      2. 非空使用forEach标签遍历所有员工
   3. 使用EL表达式显示员工信息

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <**html**> <**head**>  <**title**>查询所有员工</**title**> </**head**> <**body**>  <**center**>  <**c:if test="${empty** requestScope.emps**}"**>  <**h1**>没有任何员工</**h1**>  </**c:if**>  <**c:if test="${not empty** requestScope.emps**}"**>  <**h1**>员工信息</**h1**>  <**table border="1" cellpadding="10" cellspacing="0"**>  <**tr**>  <**th**>Id</**th**>  <**th**>LastName</**th**>  <**th**>Email</**th**>  <**th**>Gender</**th**>  <**th**>Department</**th**>  <**th colspan="2"**>Operate</**th**>  </**tr**>  <**c:forEach items="${**requestScope.emps**}" var="emp"**>  <**tr**>  <**td**>**${**emp.id**}**</**td**>  <**td**>**${**emp.lastName**}**</**td**>  <**td**>**${**emp.email**}**</**td**>  <**td**>  <**c:if test="${**emp.gender==1**}"**>男</**c:if**>  <**c:if test="${**emp.gender==0**}"**>女</**c:if**>  </**td**>  <**td**>**${**emp.department.departmentName**}**</**td**>  <**td**><**a href="#"**>Edit</**a**></**td**>  <**td**><**a href="#"**>Delete</**a**></**td**>  </**tr**>  </**c:forEach**>  </**table**>  </**c:if**>  </**center**> </**body**> </**html**> |

#### SpringMVC的form表单

通过 SpringMVC 的表单标签可以实现将模型数据中的属性和 HTML 表单元素相绑定，以实现表单数据更便捷编辑和表单值的回显。

使用SpringMVC的form标签需要在jsp页面中引入以下标签库

|  |
| --- |
| <%@ **taglib prefix**="**form**" **uri**="**http://www.springframework.org/tags/form**" %> |

1. form:form标签
   1. 一般情况下，通过 GET 请求获取表单页面，而通过 POST 请求提交表单页面，因此获取表单页面和提交表单页面的 URL 是相同的。
   2. 如果获取表单和提交表单的URL相同，<form:form> 标签就无需通过 action 属性指定表单提交的 URL，method属性默认页是post。
   3. SpringMVC默认会对表单中的所有表单项进行回显，默认会以command为key从request域中寻找模型数据，如果找不到则抛出异常；我们可以通过表单的modelAttribute属性指定查询模型数据的key。
2. 表单项form:input、form:password、form:radiobutton、form:select对应 HTML input表单项的 text、password、radio和select 标签
3. form:input标签中的属性
   1. path：表单字段，对应 html 元素的 name 属性，支持级联属性。
   2. htmlEscape：是否对表单值的 HTML 特殊字符进行转换，默认值为 true。
   3. cssClass：表单组件对应的 CSS 样式类名。
   4. cssErrorClass：表单组件的数据存在错误时，采取的 CSS 样式。
4. form:radiobutton：单选框组件标签，当表单POJO对象对应的属性值和 value 值相等时，单选框被选中。
5. form:radiobuttons：单选框组标签，用于构造多个单选框
   1. items：可以是一个 List、String[] 或 Map。
   2. itemValue：指定 radio 的 value 值。可以是集合中 bean 的一个属性值。
   3. itemLabel：指定 radio 的 label 值。
   4. delimiter：指定多个单选框的分隔符。
6. form:checkbox：复选框组件，用于构造单个复选框。
7. form:checkboxs：用于构造多个复选框。使用方式同 form:radiobuttons 标签。
8. form:select：用于构造下拉框组件。使用方式同 form:radiobuttons 标签。
9. form:option：下拉框选项组件标签。使用方式同 form:radiobuttons 标签。
10. form:errors：显示表单组件或数据校验所对应的错误
    1. <form:errors path= “\*”/> ：显示表单所有的错误。
    2. <form:errors path= “user\*” /> ：显示所有以 user 为前缀的属性对应的错误。
    3. <form:errors path= “username” /> ：显示特定表单对象属性的错误。

#### 6.3.3 添加员工

1. 在list.jsp页面中创建添加员工的连接

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/addEmployee"**>添加新员工</**a**> |

1. 创建去添加员工页面的处理方法
   1. 注入DepartmentDao获取所有部门信息
   2. 向request域中添加一个没有任何属性值的Employee对象作为模型数据

|  |
| --- |
| @RequestMapping(value = **"/addEmployee"**,method = RequestMethod.***GET***) **public** String toAddEmployeePage(Map<String , Object> map){  *//调用DepartmentDao中获取所有部门的方法* Collection<Department> departments = **departmentDao**.getDepartments();  *//将departments放到map中* map.put(**"depts"**,departments);  *//将一个没有任何属性值的Employee对象作为模型数据放到map中* map.put(**"emp"**,**new** Employee());  **return "add"**; } |

1. 创建添加员工的页面add.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <%@ **taglib prefix**="**form**" **uri**="**http://www.springframework.org/tags/form**" %> <**html**> <**head**>  <**title**>添加员工的页面</**title**> </**head**> <**body**>  <**h1**>添加员工</**h1**>  <**form:form modelAttribute="emp"**>  员工姓名：<**form:input path="lastName"**/><**br**>  员工邮箱：<**form:input path="email"**/><**br**>  员工性别：<**form:radiobutton path="gender" value="1" label="男"** />  <**form:radiobutton path="gender" value="0" label="女"** />  <**br**>  员工部门：<**form:select path="department.id" items="${**requestScope.depts**}"  itemValue="id" itemLabel="departmentName"**></**form:select**>  <**br**><**br**>  <**input type="submit"**>  </**form:form**> </**body**> </**html**> |

1. 创建添加员工的处理方法

|  |
| --- |
| @RequestMapping(value = **"/addEmployee"**,method = RequestMethod.***POST***) **public** String addEmployee(Employee employee){  *//调用EmployeeDao中添加员工的方法* **employeeDao**.save(employee);  *//重定向到查询所有员工的URL* **return "redirect:/getEmployees"**; } |

#### 6.3.4 处理静态资源

1. 向页面中导入静态资源jquery-1.9.1.min.js

|  |
| --- |
| **head**>  <**title**>查询所有员工</**title**>  <**script type="javascript" src="${**pageContext.request.contextPath**}/static/script/jquery-1.9.1.min.js"**></**script**> </**head**> |

1. 导入之后发现Idea的控制太出现警告



1. 原因是请求静态资源js文件的请求被DispacherServlet拦截，但是没有找到映射该请求的处理方法，但是请求静态资源本来就不需要创建处理请求的方法，所以我们需要在SpringMVC的配置文件中添加如下配置：

|  |
| --- |
| *<!--配置处理静态资源-->* <**mvc:default-servlet-handler**></**mvc:default-servlet-handler**>  *<!--配置了处理静态资源之后也需要配置以下标签-->* <**mvc:annotation-driven**></**mvc:annotation-driven**> |

1. mvc:default-servlet-handler说明
   1. 配置了该标签之后将在 SpringMVC 上下文中定义一个 DefaultServletHttpRequestHandler，它会对进入 DispatcherServlet 的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由 WEB 应用服务器默认的 Servlet 处理，如果不是静态资源的请求，才由 DispatcherServlet 继续处理。
   2. WEB应用服务器Tomcat中配置的默认的Servlet（在Tomcat的web.xml文件中查询）

|  |
| --- |
| <**servlet**>  <**servlet-name**>default</**servlet-name**>  <**servlet-class**>org.apache.catalina.servlets.DefaultServlet</**servlet-class**>  <**init-param**>  <**param-name**>debug</**param-name**>  <**param-value**>0</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>listings</**param-name**>  <**param-value**>false</**param-value**>  </**init-param**>  <**load-on-startup**>1</**load-on-startup**> </**servlet**>  <**servlet-mapping**>  <**servlet-name**>default</**servlet-name**>  <**url-pattern**>/</**url-pattern**> </**servlet-mapping**> |

* 1. 如果WEB服务器中配置的Servlet的名字不是default，那么mvc:default-servlet-handler标签中需要显示指定Servlet的名称；如果是default可以省略不指定。

|  |
| --- |
| <**mvc:default-servlet-handler default-servlet-name="WEB服务器中配置的Servlet的名称"**></**mvc:default-servlet-handler**> |

#### 6.3.5 删除员工

1. 在list.jsp页面的表单中设置删除员工的超链接

|  |
| --- |
| <**td**><**a class="delEmp" id="${**emp.lastName**}" href="${**pageContext.request.contextPath**}/deleteEmployeeById/${**emp.id**}"**>Delete</**a**></**td**> |

1. 由于删除员工要发送Delete请求，但是Delete请求需要通过Post请求转换，而超链接发送的是Get请求，所以我们需要先将其转换为通过form表单发送Post请求，可以通过js代码转换。

|  |
| --- |
| <**head**>  <**title**>查询所有员工</**title**>  <**script type="javascript" src="${**pageContext.request.contextPath**}/static/script/jquery-1.9.1.min.js"**></**script**>  <**script type="javascript"**>  **$**(**function** () {  *//给删除员工的超链接绑定单击事件* **$**(**".delEmp"**).click(**function** () {  *//获取要删除的员工的名字* **var** lastName = **$**(**this**).attr(**"id"**);  *//弹出提示框* **var** flag = *confirm*(**"确定要删除员工 "**+lastName+**" 吗？"**);  **if**(flag){  *//获取超链接的href属性值* **var** url = **$**(**this**).attr(**"href"**);  *//将url的值设置到form表单的action属性中* **$**(**"#deleteForm"**).attr(**"action"**,url);  *//提交表单* **$**(**"#deleteForm"**).submit();  }  *//取消默认行为，不让超链接发送请求* **return false**;  });  });  </**script**> </**head**> <**body**>  <**form action="" method="post" id="deleteForm"**>  <**input type="hidden" name="\_method" value="delete"**>  </**form**> |

1. 创建处理方法

|  |
| --- |
| @RequestMapping(value = **"/deleteEmployeeById/{id}"**,method = RequestMethod.***DELETE***) **public** String deleteEmployeeById(@PathVariable(**"id"**) Integer id){  *//调用EmployeeDao中删除员工的方法* **employeeDao**.delete(id);  *//重定向到查询所有员工的URL* **return "redirect:/getEmployees"**; } |

#### 6.3.6 修改员工

1. 在list.jsp页面的表单中设置编辑员工的超链接

|  |
| --- |
| <**td**><**a href="${**pageContext.request.contextPath**}/updateEmployee/${**emp.id**}"**>Edit</**a**></**td**> |

1. 创建去修改员工页面的处理方法

|  |
| --- |
| @RequestMapping(value = **"/updateEmployee/{id}"**,method = RequestMethod.***GET***) **public** String toUpdateEmployeePage(@PathVariable(**"id"**) Integer id , Map<String , Object> map){  *//调用EmployeeDao中获取员工信息的方法* Employee employee = **employeeDao**.get(id);  *//调用DepartmentDao中获取所有部门的方法* Collection<Department> departments = **departmentDao**.getDepartments();  *//将employee和departments放到map中* map.put(**"emp"**,employee);  map.put(**"depts"**,departments);  **return "update"**; } |

1. 创建修改员工的页面update.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <%@ **taglib prefix**="**form**" **uri**="**http://www.springframework.org/tags/form**" %> <**html**> <**head**>  <**title**>更新员工的页面</**title**> </**head**> <**body**>  <**h1**>更新员工</**h1**>  <**form:form modelAttribute="emp"**>  <**input type="hidden" name="\_method" value="put"**>  员工编号：<**form:input path="id" readonly="true"** /><**br**>  员工姓名：<**form:input path="lastName"**/><**br**>  员工邮箱：<**form:input path="email"**/><**br**>  员工性别：<**form:radiobutton path="gender" value="1" label="男"** />  <**form:radiobutton path="gender" value="0" label="女"** />  <**br**>  员工部门：<**form:select path="department.id" items="${**requestScope.depts**}"  itemValue="id" itemLabel="departmentName"**></**form:select**>  <**br**><**br**>  <**input type="submit"**>  </**form:form**> </**body**> </**html**> |

1. 创建修改员工的处理方法

|  |
| --- |
| @RequestMapping(value = **"/updateEmployee/{id}"**,method = RequestMethod.***PUT***) **public** String updateEmployee(Employee employee){  *//调用EmployeeDao中更新员工的方法* **employeeDao**.save(employee);  *//重定向到查询所有员工的URL* **return "redirect:/getEmployees"**; } |

#### 6.3.7 合并添加、更新员工的页面和处理器方法（学生完成）

1. input.jsp页面

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **taglib prefix**="**form**" **uri**="**http://www.springframework.org/tags/form**" %> <%@ **taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <**html**> <**head**>  <**title**>添加或修改员工的页面</**title**> </**head**> <**body**>  <**h1**>添加或修改员工</**h1**>  *<%--  SpringMVC默认会对表单中所有的表单项进行回显，默认会以command为key从request域中需要模型数据，  我们可以通过form标签的modelAttribute属性来指定该key  --%>* <**form:form action="${**pageContext.request.contextPath**}/addOrUpdateEmployee" modelAttribute="emp"**>   *<%--path属性就相当于input标签中的name属性，path的属性值要与POJO类的属性名保持一致--%>* <**c:if test="${not empty** requestScope.emp.id**}"**>  *<%--更新时才携带\_method参数--%>* <**input type="hidden" name="\_method" value="put"**>  员工编号：<**form:input path="id" readonly="true"**/><**br**>  </**c:if**>  员工姓名：<**form:input path="lastName"** /><**br**>  员工邮箱：<**form:input path="email"** /><**br**>  员工性别：<**form:radiobutton path="gender" value="1" label="男"**/>  <**form:radiobutton path="gender" value="0" label="女"**/> <**br**>  *<%--department.id：给级联属性赋值--%>* 员工部门：<**form:select path="department.id" items="${**requestScope.depts**}"  itemLabel="departmentName" itemValue="id"**/><**br**>  <**input type="submit"**>  </**form:form**> </**body**> </**html**> |

1. 处理器方法

|  |
| --- |
| *//添加或更新员工* @RequestMapping(value = **"/addOrUpdateEmployee"**,method = {RequestMethod.***POST***,RequestMethod.***PUT***}) **public** String addOrUpdateEmployee(Employee employee){  System.***out***.println(employee);  *//调用EmployeeDao中添加或更新员工的方法* **employeeDao**.save(employee);  *//重定向到查询所有员工的请求* **return "redirect:/getEmployees"**; } |

## 第7章 处理JSON

### 7.1 HttpMessageConverter<T>接口

#### 7.1.1 简介

1. 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。
2. HttpMessageConverter<T>接口定义的方法：
   1. boolean canRead(Class<?> clazz, @Nullable 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,applaiction/json等)。

* 1. boolean canWrite(Class<?> clazz, @Nullable 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。

* 1. List<MediaType> getSupportedMediaTypes();

获取该转换器支持的媒体类型。

* 1. T read(Class<? extends T> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException;

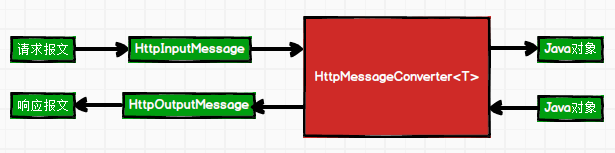
将请求信息流转换为 T 类型的对象。

* 1. void write(T t, @Nullable MediaType contentType, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException;

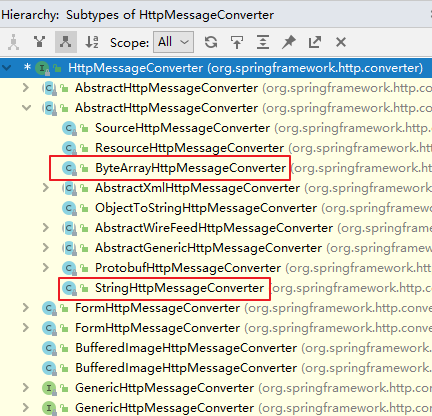
将T类型的对象写到响应流中，同时指定相应的媒体类型为 contentType。

#### 7.1.2 运行原理

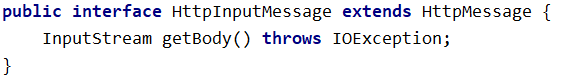
1. 原理图



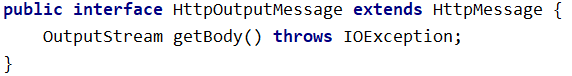
1. HttpMessageConverter<T>接口的实现类



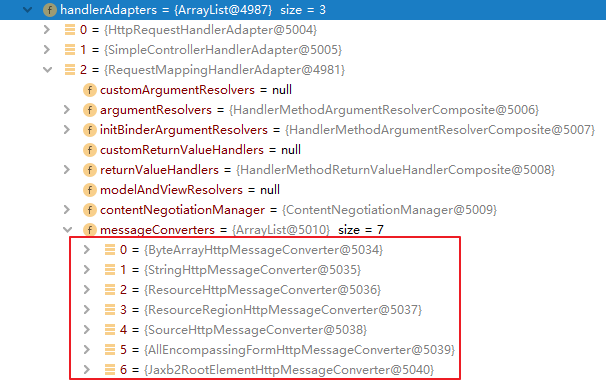
1. HttpInputMessage



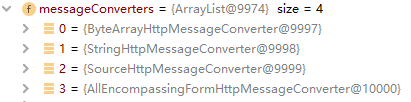
1. HttpOutputMessage



1. DispatcherServlet 默认装配了 RequestMappingHandlerAdapter ，而 RequestMappingHandlerAdapter 默认装配的 HttpMessageConverter如下：



1. 注意：SpringMVC的配置文件中一定要配置mvc:annotation-driven标签。如果不配置将只装配以下转换器：



#### 7.1.3 如何使用HttpMessageConverter

1. 使用 HttpMessageConverter<T> 将请求信息转化并绑定到处理方法的入参中或将响应结果转为对应类型的响应信息，Spring 提供了两种途径：
   1. 使用 @RequestBody / @ResponseBody 对处理方法进行标注。
   2. 使用 HttpEntity<T> / ResponseEntity<T> 作为处理方法的入参或返回值。
2. 当控制器处理方法使用到 @RequestBody/@ResponseBody 或HttpEntity<T>/ResponseEntity<T> 时，Spring 首先根据请求头或响应头的 Accept 属性选择匹配的 HttpMessageConverter, 进而根据参数类型或泛型类型的过滤得到匹配的 HttpMessageConverter, 若找不到可用的 HttpMessageConverter 将报错。
3. @RequestBody 和 @ResponseBody 不需要成对出现

#### 7.1.4 @RequestBody和HttpEntity<T>的使用

1. 需求：将请求体转换为字符串
2. 使用@RequestBody注解
   1. 表单页面

|  |
| --- |
| <**h1**>测试@RequestBody注解</**h1**> <**form action="${**pageContext.request.contextPath**}/testRequestBody" method="post"**>  用户名：<**input type="text" name="username"**><**br**>  密码：<**input type="password" name="password"**><**br**>  <**input type="submit"**> </**form**> |

* 1. 处理器方法

|  |
| --- |
| @RequestMapping(value = **"/testRequestBody"**,method = RequestMethod.***POST***) **public** String testRequestBody(@RequestBody String requestBody){  System.***out***.println(**"请求体中的内容有："**+requestBody);  **return "success"**; } |

* 1. 测试结果



1. 同样的表单使用HttpEntity<T>

|  |
| --- |
| @RequestMapping(value = **"/testHttpEntity"**,method = RequestMethod.***POST***) **public** String testHttpEntity(HttpEntity<String> request ){  System.***out***.println(**"请求头中的内容有："**+request.getHeaders());  System.***out***.println(**"请求体中的内容有："**+request.getBody());  **return "success"**; } |

#### 7.1.5 @ResponseBody和ResponseEntity<T>的使用

1. @ResponseBody注解
   1. 可以添加到类上也可以添加到方法上
   2. 添加了该注解的处理器方法方法的返回值将直接响应给浏览器
   3. 测试案例：将字符串直接响应给浏览器

|  |
| --- |
| @ResponseBody @RequestMapping(value = **"/testResponseBody"**) **public** String testResponseBody(){  System.***out***.println(**"测试@ResponseBody注解"**);  **return "success"**; } |

1. 使用ResponseEntity<T>实现文件下载
   1. 页面连接

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/testResponseEntity"**>文件下载</**a**> |

* 1. 处理器方法

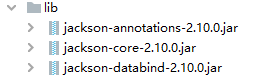
|  |
| --- |
| @RequestMapping(value = **"/testResponseEntity"**) **public** ResponseEntity<**byte**[]> testResponseEntity(HttpSession session) **throws** IOException {  *//获取ServletContext对象* ServletContext servletContext = session.getServletContext();  *//获取服务器中文件的真实路径* String realPath = servletContext.getRealPath(**"/download/atguigu.txt"**);  *//创建输入流* InputStream is = **new** FileInputStream(realPath);  *//创建字节数组* **byte**[] bytes = **new byte**[is.available()];  *//将流读到字节数组中* is.read(bytes);  *//创建HttpHeaders对象设置响应头信息* MultiValueMap<String, String> headers = **new** HttpHeaders();  *//设置要下载的文件的名字* headers.add(**"Content-Disposition"**, **"attachment;filename=sgg.txt"**);  *//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; } |

### 7.2 返回JSON格式数据

有了HttpMessageConverter<T>转换器之后，我们将对象转换为JSON格式的数据将变得非常方便。

将对象转换为JSON格式的步骤：

1. 导入以下jar包



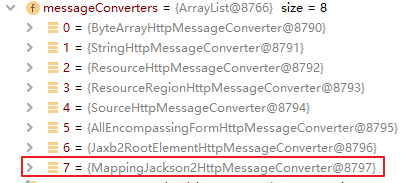
1. 处理器方法

|  |
| --- |
| @ResponseBody @RequestMapping(**"/testJSON"**) **public** Employee testJson(){  *//创建Employee对象* Employee employee = **new** Employee(1, **"zhangsan"**, **"zhangsan@atguigu.com"**, 1, **new** Department(1001, **"Teacher"**));  **return** employee; } |

1. 页面效果



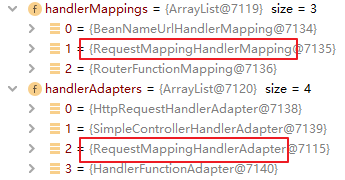
1. 添加了jackson的jar包之后多装配了一个转换器：



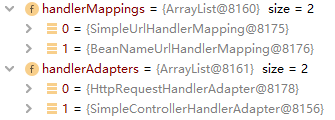
1. 必须配置mvc:annotation-driver标签才能装配Jackson的转换器

### 7.3 mvc:annotation-driven的作用

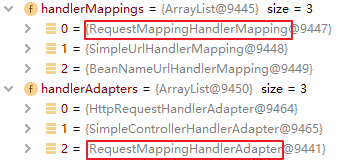
1. 何时配置了该标签？
   1. 配置不经过处理器方法直接响应的页面即配置了mvc:view-controller标签会导致其他请求路径失效，需要配置mvc:annotation-driven标签。
   2. 配置了处理静态资源即配置了mvc:default-servlet-handler标签会导致其他请求路径失效，需要配置mvc:annotation-driven标签。
2. 配置mvc:annotation-driven标签前后说明：
   1. 没有配置mvc:view-controller标签时，DispatcherServlet中装配的handlerAdapters有以下4个：



* 1. 配置了mvc:view-controller标签，没有配置mvc:annotation-driven标签时，DispatcherServlet中装配的handlerAdapters变成了2个，没有了RequestMappingHandlerAdapter这个适配器，导致所有的请求找到不映射。



* 1. 配置了mvc:view-controller标签，也配置mvc:annotation-driven标签时，DispatcherServlet中装配的handlerAdapters又有了RequestMappingHandlerAdapter，所以所有的映射又起作用了。



1. 关于mvc:default-servlet-handler标签配置之前其他路径失效的原因同mvc:view-controller类似。

## 第8章 文件上传

### 8.1 简介

1. Spring MVC 为文件上传提供了直接的支持，通过MultipartResolver 接口实现。
2. Spring MVC 上下文中默认没有装配 MultipartResovler，因此默认情况下不能处理文件的上传工作，如果想使用 SpringMVC 的文件上传功能，需要在上下文中配置 MultipartResolver。
3. Jakarta Commons FileUpload 技术实现了一个 MultipartResolver接口， 实现类是：CommonsMultipartResolver

### 8.2 文件上传步骤

1. 导入以下jar包



1. 在SpringMVC的配置文件中配置CommonsMultipartResolver解析器

|  |
| --- |
| <**bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  *<!--设置字符集-->* <**property name="defaultEncoding" value="utf-8"**></**property**>  *<!--设置总文件的大小-->* <**property name="maxUploadSize" value="102400"**></**property**> </**bean**> |

1. 表单页面
   1. enctype属性值设置为multipart/form-data
   2. 上传文件的表单项的type设置为file

|  |
| --- |
| <**h1**>文件上传</**h1**> <**form action="${**pageContext.request.contextPath**}/testFileUpload" method="post" enctype="****multipart/form-data"**>  描述:<**input type="text" name="desc"**><**br**>  文件：<**input type="file" name="filename"**><**br**>  <**input type="submit"**> </**form**> |

1. 处理器方法

|  |
| --- |
| @RequestMapping(**"/testFileUpload"**) **public** String testFileUpload(@RequestParam(**"desc"**) String desc ,  @RequestParam(**"filename"**)MultipartFile file,HttpSession session) **throws** IOException {  System.***out***.println(**"文件的描述信息是："**+desc);  *//获取文件名* String fileName = file.getOriginalFilename();  *//获取文件的类型* String contentType = file.getContentType();  *//获取文件的大小* **long** size = file.getSize();  System.***out***.println(**"文件名是："**+fileName);  System.***out***.println(**"文件的类型是："**+contentType);  System.***out***.println(**"文件的大小是："**+size+**" 个字节"**);  *//获取文件上传的真实路径* ServletContext servletContext = session.getServletContext();  String realPath = servletContext.getRealPath(**"/upload"**);  *//创建路径* File upload = **new** File(realPath);  **if**(!upload.exists()){  upload.mkdirs();  }  *//上传文件* file.transferTo(**new** File(realPath+**"/"**+fileName));  **return "success"**; } |

1. 思考：如何上传多个文件？

## 第9章 拦截器

### 9.1 自定义拦截器简介

1. Spring MVC也可以使用拦截器对请求进行拦截处理，用户可以自定义拦截器来实现特定的功能，自定义的拦截器可以实现HandlerInterceptor接口，也可以继承HandlerInterceptorAdapter 适配器类。
2. HandlerInterceptor接口方法说明：
   1. preHandle()：这个方法在业务处理器处理请求之前被调用，可以在此方法中做一些权限的校验。如果程序员决定该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去进行处理，则返回true；如果程序员决定不需要再调用其他的组件去处理请求，则返回false。
   2. postHandle()：这个方法在业务处理器处理请求之后，渲染视图之前调用。在此方法中可以对ModelAndView中的模型和视图进行处理。
   3. afterCompletion()：这个方法在 DispatcherServlet 完全处理完请求后被调用，可以在该方法中进行一些资源清理的操作。

### 9.2 配置单个拦截器

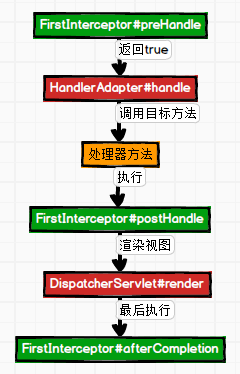
1. 创建一个类实现HandlerInterceptor接口

|  |
| --- |
| **package** com.atguigu.springmvc.interceptors;  **import** org.springframework.web.servlet.HandlerInterceptor; **import** org.springframework.web.servlet.ModelAndView;  **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse;  **public class** FirstInterceptor **implements** HandlerInterceptor {   @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  System.***out***.println(**"FirstInterceptor的preHandle方法执行"**);  **return true**;  }   @Override  **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  System.***out***.println(**"FirstInterceptor的postHandle方法执行"**);  }   @Override  **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.***out***.println(**"FirstInterceptor的afterCompletion方法执行"**);  }   } |

1. 在SpringMVC配置文件中配置拦截器

|  |
| --- |
| *<!--配置拦截器-->* <**mvc:interceptors**>  *<!--声明定义的拦截器  通过这种方式定义的拦截器会拦截所有请求  -->* <**bean id="firstInterceptor" class="com.atguigu.springmvc.interceptors.FirstInterceptor"**></**bean**> </**mvc:interceptors**> |

1. 单个拦截器的执行流程图



### 9.3 配置多个拦截器

1. 再创建一个拦截器类

|  |
| --- |
| **package** com.atguigu.springmvc.interceptors;  **import** org.springframework.web.servlet.HandlerInterceptor; **import** org.springframework.web.servlet.ModelAndView;  **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse;  **public class** SecondInterceptor **implements** HandlerInterceptor {   @Override  **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  System.***out***.println(**"SecondInterceptor的preHandle方法执行"**);  **return true**;  }   @Override  **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  System.***out***.println(**"SecondInterceptor的postHandle方法执行"**);  }   @Override  **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.***out***.println(**"SecondInterceptor的afterCompletion方法执行"**);  }   } |

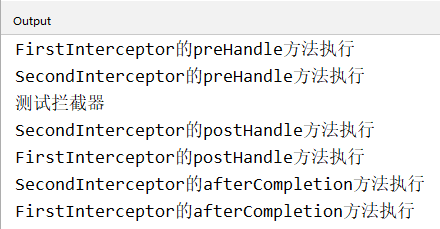
1. 通过另外一种方式在SpringMVC的配置文件中配置拦截器

|  |
| --- |
| *<!--配置拦截器-->* <**mvc:interceptors**>  *<!--声明定义的拦截器  通过这种方式定义的拦截器会拦截所有请求  -->* <**bean id="firstInterceptor" class="com.atguigu.springmvc.interceptors.FirstInterceptor"**></**bean**>  *<!--通过另外一种方式配置拦截器-->* <**mvc:interceptor**>  *<!--配置拦截的路径-->* <**mvc:mapping path="/testInterceptor"**/>  *<!--配置不拦截的路径--> <!-- <mvc:exclude-mapping path="/testInterceptor"/>-->* <**bean id="secondInterceptor" class="com.atguigu.springmvc.interceptors.SecondInterceptor"**></**bean**>  </**mvc:interceptor**> |

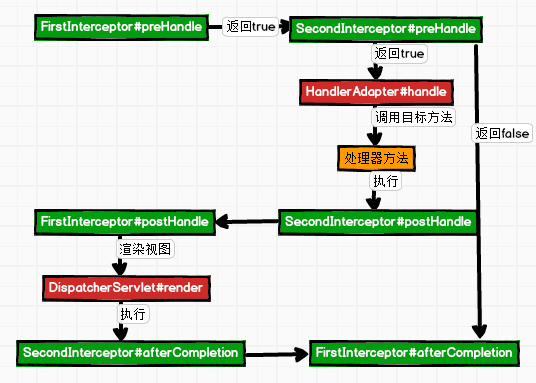
1. 处理器方法

|  |
| --- |
| @RequestMapping(**"/testInterceptor"**) **public** String testInterceptor(){  System.***out***.println(**"测试拦截器"**);  **return "success"**; } |

1. 测试结果

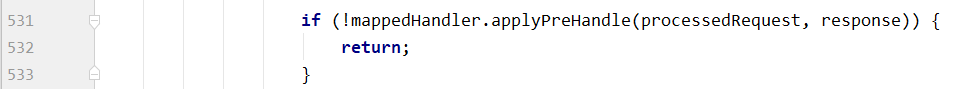


1. 多个拦截器的执行顺序由SpringMVC配置文件中配置的顺序决定。
2. 多个拦截器的执行流程图



### 9.4 拦截器底层源码

1. 在ispatcherServlet类中
   1. 531行执行拦截器的preHandle方法



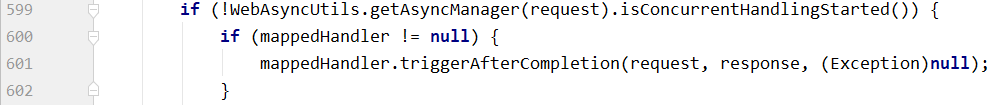
* 1. 535行执行目标方法



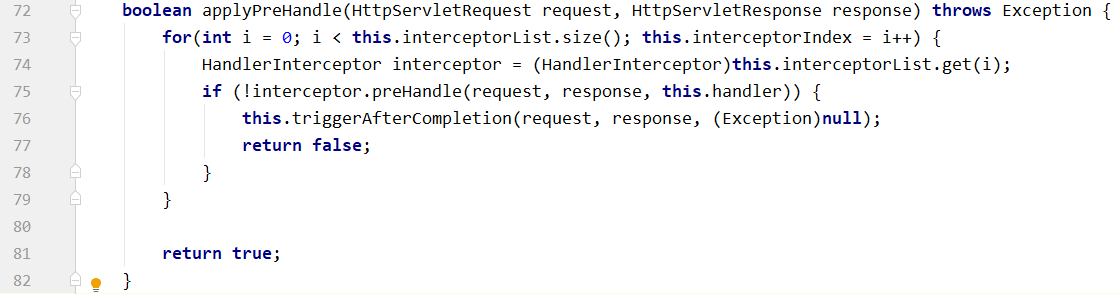
* 1. 541行执行拦截器的postHandle方法



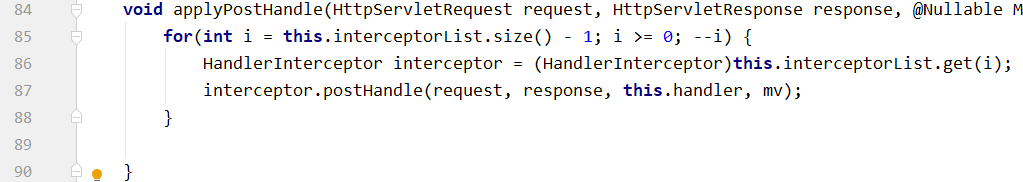
* 1. 601行执行拦截器的afterCompletion方法



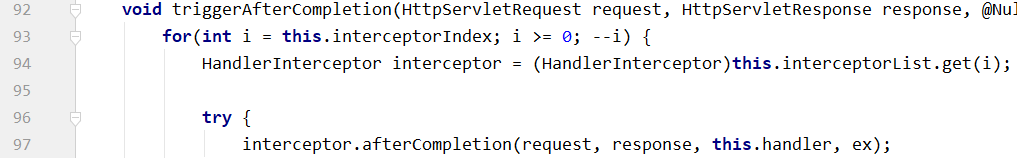
1. 在HandlerExecutionChain类中
   1. 72行的applyPreHandle方法中遍历所有的拦截器执行拦截器的preHandle方法



* 1. 84行的applyPostHandle方法中反向遍历所有的拦截器执行拦截器的postHandle方法



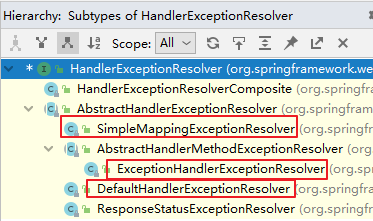
* 1. 92行的triggerAfterCompletion方法中反向遍历所有的拦截器执行拦截器的afterCompletion方法



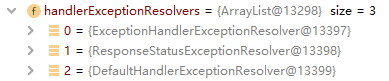
## 第10章 异常处理

### 10.1 简介

1. Spring MVC 通过 HandlerExceptionResolver 处理程序的异常，包括 Handler 映射、数据绑定以及目标方法执行时发生的异常。
2. SpringMVC 提供的 HandlerExceptionResolver 的实现类：



1. DispatcherServlet默认装配的异常处理器有：



### 10.2 DefaultHandlerExceptionResolver

通过底层源码查看该处理器能处理的异常：

|  |
| --- |
| @Nullable **protected** ModelAndView doResolveException(HttpServletRequest request, HttpServletResponse response, @Nullable Object handler, Exception ex) {  **try** {  **if** (ex **instanceof** HttpRequestMethodNotSupportedException) {  **return this**.handleHttpRequestMethodNotSupported((HttpRequestMethodNotSupportedException)ex, request, response, handler);  }   **if** (ex **instanceof** HttpMediaTypeNotSupportedException) {  **return this**.handleHttpMediaTypeNotSupported((HttpMediaTypeNotSupportedException)ex, request, response, handler);  }   **if** (ex **instanceof** HttpMediaTypeNotAcceptableException) {  **return this**.handleHttpMediaTypeNotAcceptable((HttpMediaTypeNotAcceptableException)ex, request, response, handler);  }   **if** (ex **instanceof** MissingPathVariableException) {  **return this**.handleMissingPathVariable((MissingPathVariableException)ex, request, response, handler);  }   **if** (ex **instanceof** MissingServletRequestParameterException) {  **return this**.handleMissingServletRequestParameter((MissingServletRequestParameterException)ex, request, response, handler);  }   **if** (ex **instanceof** ServletRequestBindingException) {  **return this**.handleServletRequestBindingException((ServletRequestBindingException)ex, request, response, handler);  }   **if** (ex **instanceof** ConversionNotSupportedException) {  **return this**.handleConversionNotSupported((ConversionNotSupportedException)ex, request, response, handler);  }   **if** (ex **instanceof** TypeMismatchException) {  **return this**.handleTypeMismatch((TypeMismatchException)ex, request, response, handler);  }   **if** (ex **instanceof** HttpMessageNotReadableException) {  **return this**.handleHttpMessageNotReadable((HttpMessageNotReadableException)ex, request, response, handler);  }   **if** (ex **instanceof** HttpMessageNotWritableException) {  **return this**.handleHttpMessageNotWritable((HttpMessageNotWritableException)ex, request, response, handler);  }   **if** (ex **instanceof** MethodArgumentNotValidException) {  **return this**.handleMethodArgumentNotValidException((MethodArgumentNotValidException)ex, request, response, handler);  }   **if** (ex **instanceof** MissingServletRequestPartException) {  **return this**.handleMissingServletRequestPartException((MissingServletRequestPartException)ex, request, response, handler);  }   **if** (ex **instanceof** BindException) {  **return this**.handleBindException((BindException)ex, request, response, handler);  }   **if** (ex **instanceof** NoHandlerFoundException) {  **return this**.handleNoHandlerFoundException((NoHandlerFoundException)ex, request, response, handler);  }   **if** (ex **instanceof** AsyncRequestTimeoutException) {  **return this**.handleAsyncRequestTimeoutException((AsyncRequestTimeoutException)ex, request, response, handler);  }  } **catch** (Exception var6) {  **if** (**this**.logger.isWarnEnabled()) {  **this**.logger.warn(**"Failure while trying to resolve exception ["** + ex.getClass().getName() + **"]"**, var6);  }  }   **return null**; } |

### SimpleMappingExceptionResolver

如果希望对所有异常进行统一处理，可以使用 SimpleMappingExceptionResolver，它将异常类名映射为视图名，即发生异常时使用对应的视图报告异常。

使用SimpleMappingExceptionResolver的步骤：

1. 在SpringMVC的配置文件中配置SimpleMappingExceptionResolver

|  |
| --- |
| <**bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver"**>  <**property name="exceptionMappings"**>  <**props**>  *<!--设置异常类型和视图页面-->* <**prop key="java.lang.ArithmeticException"**>error</**prop**>  </**props**>  </**property**>  *<!--设置向request域中存放异常信息的key，默认是exception-->* <**property name="exceptionAttribute" value="e"**></**property**> </**bean**> |

1. 创建处理器方法

|  |
| --- |
| @RequestMapping(**"/testSimple"**) **public** String testSimpleMappingExceptionResolver(@RequestParam(**"i"**) Integer i){  System.***out***.println(**"测试SimpleMappingExceptionResolver"**);  **int** result = 10 / i;  System.***out***.println(result);  **return "success"**; } |

1. 测试路径

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/testSimple?i=0"**>测试异常</**a**> |

1. 创建视图页面error.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>错误页面</**title**> </**head**> <**body**>  <**h1**>异常信息是：**${**requestScope.e**}**</**h1**> </**body**> </**html**> |

1. 页面效果



## 第11章 SpringMVC的运行流程

### 11.1 运行流程描述

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



* + - 1. 浏览器

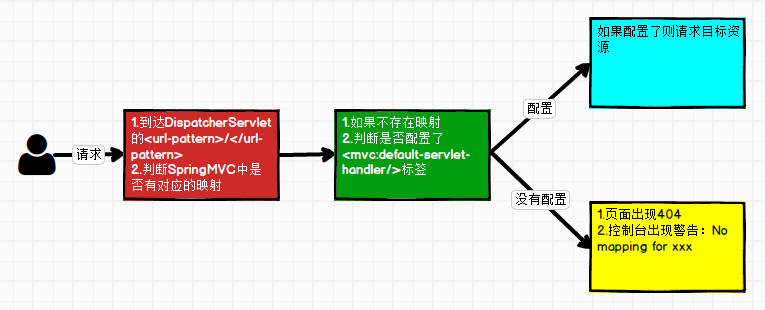


* + 1. 如果有配置，则执行目标资源（一般为静态资源，如：JS,CSS,HTML），找不到客户端也会展示404错误
  1. 存在
     1. 执行下面的流程

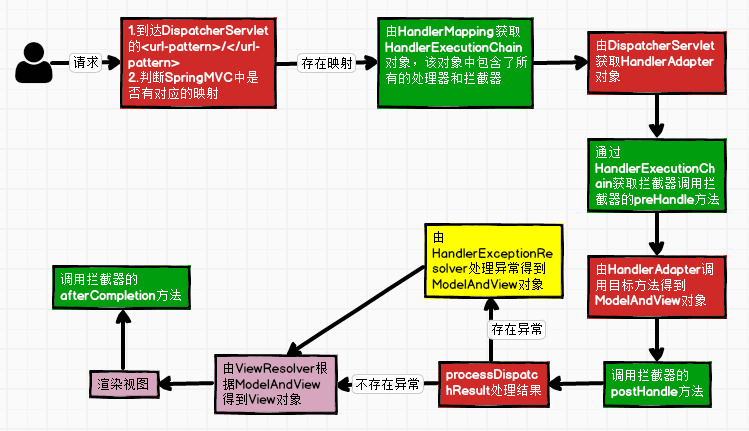
1. 根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回。
2. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。
3. 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(…)方法【正向】
4. 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)方法，处理请求。在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：
   1. HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息
   2. 数据转换：对请求消息进行数据转换。如String转换成Integer、Double等
   3. 数据格式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等
   4. 数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中
5. Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象。
6. 此时将开始执行拦截器的postHandle(...)方法【逆向】。
7. 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（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet，根据Model和View，来渲染视图。
8. 在返回给客户端时需要执行拦截器的afterCompletion(…)方法【逆向】。
9. 将渲染结果返回给客户端。

### 11.2 流程图

1. 不存在映射

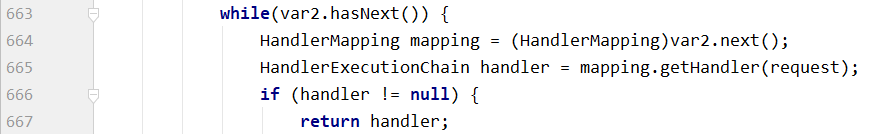


1. 存在映射

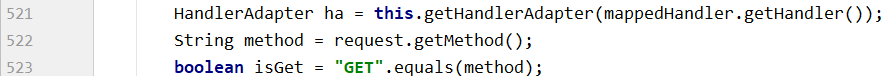


### 11.3 底层源码

1. 在DispatcherServlet类中
   1. 665行由HandlerMapping获取HandlerExecutionChain对象



* 1. 521行获取HandlerAdapter对象



* 1. 后续源码之前都曾看过

## 第12章 Spring整合SpringMVC

### 12.1 Spring与SpringMVC整合问题

1. 不整合
   1. 所有配置都放在 SpringMVC 的配置文件中
   2. 分多个 SpringMVC 的配置文件，然后使用 import 节点导入其他的配置文件
2. 整合
   1. 通常情况下，类似于数据源、事务、整合其他框架都是放在 Spring 的配置文件中
   2. Service和Dao也放在Spring的配置文件中
   3. Controller和视图解析器等放到SpringMVC的配置文件中
3. 整合带来的问题
   1. Spring的IOC容器何时初始化，即什么时候加载Spring的配置文件
      1. 解决方案：在web.xml配置文件中配置监听器ContextLoaderListener
   2. Bean被创建两次
      1. 解决方案：SpringMVC只扫描Controller，Spring不扫描Controller

### 12.2 Spring整合SpringMVC的步骤

1. 创建Web工程，在web.xml配置文件中配置DispatcherServlet和ContextLoaderListener

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0"**>  *<!--配置前端控制器-->* <**servlet**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置DispatcherServlet的初始化參數：设置SpringMVC文件的路径和文件名称 -->* <**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**>dispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**>   *<!--配置上下文的初始化参数，初始化Spring的配置文件-->* <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:beans.xml</**param-value**>  </**context-param**>  *<!--配置ContextLoaderListener监听器-->* <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  </**listener**> </**web-app**> |

1. 创建SpringMVC的配置文件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"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.springmvc" use-default-filters="false"**>  *<!--配置只扫描Controller-->* <**context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"**/>  </**context:component-scan**>   *<!--配置视图解析器-->* <**bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  *<!--配置前缀-->* <**property name="prefix" value="/WEB-INF/views/"**></**property**>  *<!--配置后缀-->* <**property name="suffix" value=".jsp"**></**property**>  </**bean**> </**beans**> |

1. 创建Spring的配置文件beans.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"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.springmvc"**>  *<!--配置不扫描Controller-->* <**context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"**/>  </**context:component-scan**> </**beans**> |

1. 创建EmployeeService接口和实现类
   1. 接口

|  |
| --- |
| **package** com.atguigu.spring.springmvc.service;  **public interface** EmployeeService {   **void** addEmployee();*//仅仅测试用，所以没传参数* } |

* 1. 实现类

|  |
| --- |
| **package** com.atguigu.spring.springmvc.service.impl;  **import** com.atguigu.spring.springmvc.service.EmployeeService; **import** org.springframework.stereotype.Service;  @Service(**"employeeService"**) **public class** EmployeeServiceImpl **implements** EmployeeService {   **public** EmployeeServiceImpl() {  System.***out***.println(**"EmployeeServiceImpl对象被创建"**);  }   @Override  **public void** addEmployee() {  System.***out***.println(**"EmployeeService中的添加方法"**);  } } |

1. 创建EmployeeController

|  |
| --- |
| **package** com.atguigu.spring.springmvc.controller;  **import** com.atguigu.spring.springmvc.service.EmployeeService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller **public class** EmployeeController {   @Autowired  **private** EmployeeService **employeeService**;   **public** EmployeeController() {  System.***out***.println(**"EmployeeController对象被创建"**);  }   @RequestMapping(**"/addEmployee"**)  **public** String addEmployee(){  **employeeService**.addEmployee();  **return "success"**;  } } |

1. 创建视图页面；首页中创建超链接测试EmployeeController中是否成功注入EmployeeServiceImpl

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/addEmployee"**>测试添加Employee</**a**> |

1. 同学们可以通过Spring整合SpringMVC并结合JdbcTemplate实现简单的增删改查操作。

### 12.3 Spring IOC容器与SpringMVC IOC容器的关系

1. SpringMVC 的 IOC 容器中的 bean 可以引用 Spring IOC 容器中的 bean
2. Spring的IOC 容器中的 bean 不能引用 SpringMVC IOC 容器中的 bean
3. SpringMVC的IOC容器为Spring的IOC容器的子容器