**SpringMVC框架**

尚硅谷java研究院

版本：V 1.0

# 第1章 SpringMVC 概述

## SpringMVC 概述

1. 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 框架之一

2）Spring3.0 后全面超越 Struts2，成为最优秀的 MVC 框架。

3）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。

4）支持 REST 风格的 URL 请求。 Restful

5）采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性。

## SpringMVC是什么

1）一种轻量级的、基于MVC的Web层应用框架。偏前端而不是基于业务逻辑层。Spring框架的一个后续产品。

2）Spring框架结构图(新版本)：



## 1.3 SpringMVC能干什么

1. 天生与Spring框架集成，如：(IOC,AOP)
2. 支持Restful风格
3. 进行更简洁的Web层开发
4. 支持灵活的URL到页面控制器的映射
5. 非常容易与其他视图技术集成，如:Velocity、FreeMarker等等
6. 因为模型数据不存放在特定的API里，而是放在一个Model里(Map数据结构实现，因此很容易被其他框架使用)
7. 非常灵活的数据验证、格式化和数据绑定机制、能使用任何对象进行数据绑定，不必实现特定框架的API
8. 更加简单、强大的异常处理
9. 对静态资源的支持
10. 支持灵活的本地化、主题等解析

## 1.4 SpringMVC怎么玩

1. 将Web层进行了职责解耦，基于请求-响应模型
2. 常用主要组件
   * 1. **DispatcherServlet**：前端控制器
     2. **Controller**：处理器/页面控制器，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. **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
     5. **LocalResolver**：本地化、国际化
     6. **MultipartResolver**：文件上传解析器
     7. **HandlerExceptionResolver**：异常处理器

## 1.5 永远的HelloWorld

1. 新建Web工程，加入 jar 包

|  |
| --- |
| spring-aop-4.0.0.RELEASE.jar  spring-beans-4.0.0.RELEASE.jar  spring-context-4.0.0.RELEASE.jar  spring-core-4.0.0.RELEASE.jar  spring-expression-4.0.0.RELEASE.jar  commons-logging-1.1.3.jar  **spring-web-4.0.0.RELEASE.jar**  **spring-webmvc-4.0.0.RELEASE.jar** |

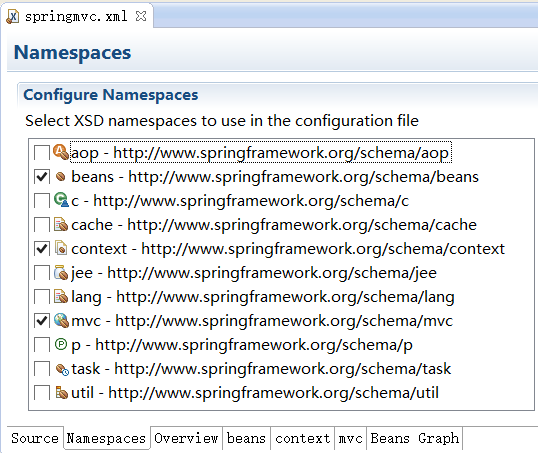
1. 在 web.xml 中配置 DispatcherServlet

|  |
| --- |
| <!-- 配置SpringMVC核心控制器： -->  <servlet>  <servlet-name>springDispatcherServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  **<!-- 配置DispatcherServlet的初始化參數：设置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>springDispatcherServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |

1. 解释配置文件的名称定义规则:

实际上也可以不通过 contextConfigLocation 来配置 SpringMVC 的配置文件, 而使用默认的.默认的配置文件为: **/WEB-INF/<servlet-name>-servlet.xml**

1. 加入 Spring MVC 的配置文件：springmvc.xml
2. 增加名称空间



1. 增加配置

|  |
| --- |
| <!-- 设置扫描组件的包： -->`  <context:component-scan base-package="**com.atguigu.springmvc**"/>    <!-- 配置映射解析器：如何将控制器返回的结果字符串，转换为一个物理的视图文件-->  <bean id="internalResourceViewResolver"  class="org.springframework.web.servlet.view.**InternalResourceViewResolver**">  **<property name="prefix" value="/WEB-INF/views/"/>**  **<property name="suffix" value=".jsp"/>**  </bean> |

1. 需要创建一个入口页面，index.jsp

|  |
| --- |
| <a href="**${pageContext.request.contextPath }**/helloworld">Hello World</a> |

1. 编写处理请求的处理器，并标识为处理器

|  |
| --- |
| package com.atguigu.springmvc.controller;    import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;    @Controller //声明Bean对象，为一个控制器组件  public class HelloWorldController {    /\*\*  \* 映射请求的名称：用于客户端请求；类似Struts2中action映射配置的action名称  \* 1. 使用 @RequestMapping 注解来映射请求的 URL  \* 2. 返回值会通过视图解析器解析为实际的物理视图, 对于 InternalResourceViewResolver 视图解析器,  \* 会做如下的解析:  \*                 通过 prefix + returnVal + suffix 这样的方式得到实际的物理视图, 然后做转发操作.  \*                 /WEB-INF/views/**success**.jsp  \*/  @RequestMapping(value="/helloworld",method=RequestMethod.GET)  public String helloworld(){    System.out.println("hello,world");    return "**success**"; //结果如何跳转呢？需要配置映射解析器  }  } |

1. 编写视图

/WEB-INF/views/success.jsp

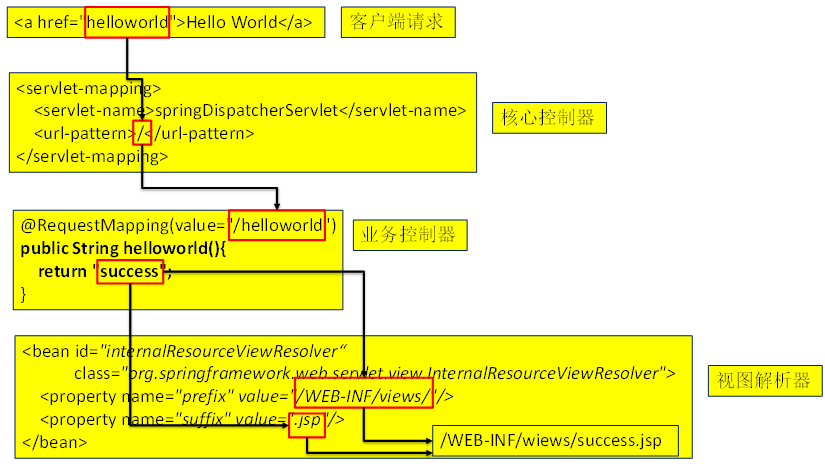
|  |
| --- |
| <h4>Sucess Page</h4> |

1. 部署测试：

<http://localhost:8080/SpringMVC_01_HelloWorld/index.jsp>

## 1.6 HelloWorld深度解析

1. HelloWorld请求流程图解：



1. 一般请求的映射路径名称和处理请求的方法名称最好一致（实质上方法名称任意）

|  |
| --- |
| @RequestMapping(value="/**helloworld**",method=RequestMethod.GET)  public String **helloworld**(){  //public String **abc123**(){  System.out.println("hello,world");  return "success";  } |

1. 演示一个错误

经常有同学会出现配置上错误，把“**/WEB-INF/views/**”**配置成了 "/WEB-INF/views"**

|  |
| --- |
| <bean id="internalResourceViewResolver"  class="org.springframework.web.servlet.view.**InternalResourceViewResolver**">  **<property name="prefix" value="/WEB-INF/views/"/>**  **<property name="suffix" value=".jsp"/>**  </bean> |

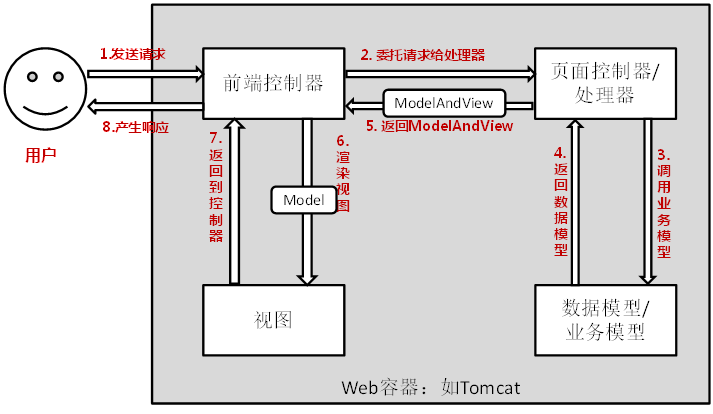
1. 处理请求方式有哪几种

|  |
| --- |
| public enum RequestMethod {  GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE  } |

1. @RequestMapping可以应用在什么地方

|  |
| --- |
| @Target({ElementType.**METHOD**, ElementType.**TYPE**})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Mapping  public @interface RequestMapping {…} |

6）流程分析:



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

# 第2 章 @RequestMapping注解

## 2.1 @RequestMapping 映射请求注解

### 2.1.1 @RequestMapping 概念

* 1. SpringMVC使用@RequestMapping注解为控制器指定可以处理哪些 URL 请求
  2. 在控制器的**类定义及方法定义处**都可标注 @RequestMapping
     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. 若类上未标注 @RequestMapping，则方法处标记的 URL 相对于 WEB 应用的根目录

4） 作用：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源码参考

|  |
| --- |
| package org.springframework.web.bind.annotation;  @Target({ElementType.METHOD, ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Mapping  public @interface RequestMapping {  String[] value() default {};  RequestMethod[] method() default {};  String[] params() default {};  String[] headers() default {};  String[] consumes() default {};  String[] produces() default {};  } |

## 2.2 RequestMapping 可标注的位置

### 2.2.1 实验代码

定义页面链接、控制器方法

|  |
| --- |
| <a href="springmvc/helloworld">test @RequestMapping</a> |

|  |
| --- |
| @Controller //声明Bean对象，为一个控制器组件  @RequestMapping("/springmvc")  public class HelloWorldController {  /\*\*  \* 映射请求的名称：用于客户端请求；类似Struts2中action映射配置的，action名称  \*1 使用@RequestMapping 注解来映射请求的 URL  \*2 返回值会通过视图解析器解析为实际的物理视图,  \* 对于 InternalResourceViewResolver 视图解析器,  \* 会做如下的解析:  \*  通过 prefix + returnVal + 后缀 这样的方式得到实际的物理视图, 然会做转发操作.  \* /WEB-INF/views/success.jsp  \*/  @RequestMapping(value="/helloworld")  public String helloworld(){  System.out.println("hello,world");  return "success"; //结果如何跳转呢？需要配置视图解析器  }  } |

## 2.3 RequestMapping映射请求方式

### 2.3.1 标准的 HTTP 请求报头



### 2.3.2 映射请求参数、请求方法或请求头

1）@RequestMapping 除了可以使用请求 URL 映射请求外，还可以使用请求方法、请求参数及请求头映射请求

2）@RequestMapping 的 value【重点】、method【重点】、params【了解】 及 heads【了解】 分别表示请求 URL、请求方法、请求参数及请求头的映射条件，他们之间是与的关系，联合使用多个条件可让请求映射更加精确化。

3）params 和 headers支持简单的表达式：

param1: 表示请求必须包含名为 param1 的请求参数

!param1: 表示请求不能包含名为 param1 的请求参数

param1 != value1: 表示请求包含名为 param1 的请求参数，但其值不能为 value1

{"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

### 2.3.3 实验代码

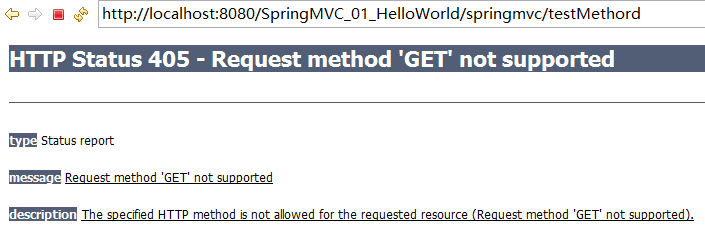
1. 定义控制器方法

|  |
| --- |
| @Controller  @RequestMapping("/springmvc")  public class SpringMVCController {  @RequestMapping(value="/testMethord",**method=RequestMethod.POST**)  public String testMethord(){  System.out.println("testMethord...");  return "success";  }  } |

1. 以get方式请求

|  |
| --- |
| <a href="springmvc/testMethord">testMethord</a> |

发生请求错误



1. 以POST方式请求

|  |
| --- |
| <form action="springmvc/testMethord" method="post">  <input type="submit" value="submit">  </form> |

## 2.4 RequestMapping映射请求参数&请求头

### 2.4.1 RequestMapping\_请求参数&请求头【了解】

|  |
| --- |
| //了解: 可以使用 params 和 headers 来更加精确的映射请求. params 和 headers 支持简单的表达式.  @RequestMapping(value="/testParamsAndHeaders",  **params= {"username","age!=10"}**, **headers = { "Accept-Language=en-US,zh;q=0.8" })**  public String testParamsAndHeaders(){  System.out.println("testParamsAndHeaders...");  return "success";  } |

### 2.4.2 实验代码

1. 请求URL

|  |
| --- |
| <!--设置请求参数和请求头信息 -->  <a href="springmvc/testParamsAndHeaders">testParamsAndHeaders</a> |

1. 测试:使用火狐或Chrom浏览器debug测试
2. 测试有参数情况(不正确)：

* <a href="springmvc/testParamsAndHeaders">testParamsAndHeaders</a>

|  |
| --- |
| 警告: No matching handler method found for servlet request: path '/springmvc/testParamsAndHeaders', method 'GET', parameters map[[empty]] |

* <a href="springmvc/testParamsAndHeaders?username=atguigu&age=10">testParamsAndHeaders</a>

|  |
| --- |
| 警告: No matching handler method found for servlet request: path '/springmvc/testParamsAndHeaders', method 'GET', parameters map['username' -> array<String>['atguigu'], 'age' -> array<String>['10']] |

* <a href="springmvc/testParamsAndHeaders?age=11">testParamsAndHeaders</a>

|  |
| --- |
| 警告: No matching handler method found for servlet request: path '/springmvc/testParamsAndHeaders', method 'GET', parameters map['age' -> array<String>['11']] |

1. 测试有参数情况(正确)：

* <a href="springmvc/testParamsAndHeaders?username=atguigu&age=15">testParamsAndHeaders</a>

## 2.5 RequestMapping支持Ant 路径风格

### 2.5.1 Ant

1. **Ant 风格资源地址支持 3 种匹配符**：【了解】

**?：匹配文件名中的一个字符**

**\*：匹配文件名中的任意字符**

**\*\*：\*\* 匹配多层路径**

1. @RequestMapping 还**支持 Ant 风格的 URL**：

|  |
| --- |
| /user/\*/createUser  匹配 /user/**aaa**/createUser、/user/**bbb**/createUser 等 URL  /user/\*\*/createUser  匹配 /user/createUser、/user/**aaa/bbb/**createUser 等 URL  /user/createUser??  匹配 /user/createUser**aa**、/user/createUser**bb** 等 URL |

### 2.5.2 实验代码

1. 定义控制器方法

|  |
| --- |
| //Ant 风格资源地址支持 3 种匹配符  //@RequestMapping(value="/testAntPath/\*/abc")  //@RequestMapping(value="/testAntPath/\*\*/abc")  @RequestMapping(value="/testAntPath/abc??")  public String testAntPath(){  System.out.println("testAntPath...");  return "success";  } |

1. 页面链接

|  |
| --- |
| <!-- Ant 风格资源地址支持 3 种匹配符 -->  <a href="springmvc/testAntPath/\*/abc">testAntPath</a>  <a href="springmvc/testAntPath/xxx/yyy/abc">testAntPath</a>  <a href="springmvc/testAntPath/abcxx">testAntPath</a> |

## 2.6 RequestMapping映射请求占位符PathVariable注解

### 2.6.1 @PathVariable

**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

**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**") 绑定到操作方法的入参中。

### 2.6.2 实验代码

1. 定义控制器方法

|  |
| --- |
| //@PathVariable 注解可以将请求URL路径中的请求参数，传递到处理请求方法的入参中  浏览器的请求: testPathVariable/1001  @RequestMapping(value="/testPathVariable/{id}")  public String testPathVariable(@PathVariable("id") Integer id){  System.out.println("testPathVariable...id="+id);  return "success";  } |

1. 请求链接

|  |
| --- |
| <!-- 测试 @PathVariable -->  <a href="springmvc/testPathVariable/1">testPathVariable</a> |

# 第3章 REST

## 3.1参考资料：

1）理解本真的REST架构风格: <http://kb.cnblogs.com/page/186516/>

2）REST: <http://www.infoq.com/cn/articles/rest-introduction>

## REST是什么？

1. REST：即 Representational State Transfer。**（资源）表现层状态转化。是目前最流行的一种互联网软件架构**。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用
2. **资源（Resources）**：网络上的一个实体，或者说是网络上的一个具体信息。

它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。

可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的 URI 。

获取这个资源，访问它的URI就可以，因此 URI 即为每一个资源的独一无二的识别符。

1. 表现层（Representation）：把资源具体呈现出来的形式，叫做它的表现层（Representation）。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。
2. 状态转化（State Transfer）：每发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）

而这种转化是建立在表现层之上的，所以就是 “表现层状态转化”。

1. 具体说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。

它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。

2）URL风格

示例：

/order/1   HTTP **GET** ：得到 id = 1 的 order    gerOrder?id=1

/order/1   HTTP **DELETE：**删除 id = 1的 order    deleteOrder?id=1

/order   HTTP **PUT**：更新order

/order      HTTP **POST**：新增 order

扩展: 开放平台 支付功能: 你应该发送什么URL ，需要传递什么参数, 需要有哪些验证 ，响应哪些数据.

3）HiddenHttpMethodFilter

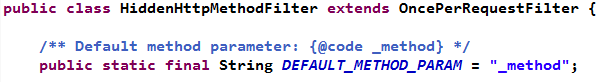
浏览器 form 表单只支持 GET 与 POST 请求，而DELETE、PUT 等 method 并不

支持，Spring3.0 添加了一个过滤器，可以将这些请求转换为标准的 http 方法，使

得支持 GET、POST、PUT 与 DELETE 请求。

## 3.3 HiddenHttpMethodFilter过滤器源码分析

1. 为什么请求隐含参数名称必须叫做”\_method”



1. hiddenHttpMethodFilter 的处理过程





## 3.4 实验代码

1. 配置**HiddenHttpMethodFilter过滤器**

|  |
| --- |
| <!-- 支持REST风格的过滤器：可以将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. 代码

|  |
| --- |
| /\*\*  \* 1.测试REST风格的 GET,POST,PUT,DELETE 操作  \* 以CRUD为例：  \* 新增: /order POST  \* 修改: /order/1 PUT         update?id=1  \* 获取: /order/1 GET                get?id=1  \* 删除: /order/1 DELETE        delete?id=1    \* 2.如何发送PUT请求或DELETE请求?  \* ①.配置HiddenHttpMethodFilter  \* ②.需要发送POST请求  \* ③.需要在发送POST请求时携带一个 name="\_method"的隐含域，值为PUT或DELETE    \* 3.在SpringMVC的目标方法中如何得到id值呢?  \* 使用@PathVariable注解  \*/  @RequestMapping(value="/testRESTGet/{id}",method=RequestMethod.GET)  public String testRESTGet(@PathVariable(value="id") Integer id){  System.out.println("testRESTGet id="+id);  return "success";  }    @RequestMapping(value="/testRESTPost",method=RequestMethod.POST)  public String testRESTPost(){  System.out.println("testRESTPost");  return "success";  }    @RequestMapping(value="/testRESTPut/{id}",method=RequestMethod.PUT)  public String testRESTPut(@PathVariable("id") Integer id){  System.out.println("testRESTPut id="+id);  return "success";  }    @RequestMapping(value="/testRESTDelete/{id}",method=RequestMethod.DELETE)  public String testRESTDelete(@PathVariable("id") Integer id){  System.out.println("testRESTDelete id="+id);  return "success";  } |

1. 请求链接

|  |
| --- |
| <!-- 实验1 测试 REST风格 GET 请求 -->  <a href="springmvc/testRESTGet/1">testREST GET</a><br/><br/>    <!-- 实验2 测试 REST风格 POST 请求 -->  <form action="springmvc/testRESTPost" method="POST">  <input type="submit" value="testRESTPost">  </form>    <!-- 实验3 测试 REST风格 PUT 请求 -->  <form action="springmvc/testRESTPut/1" method="POST">  <input type="hidden" name="\_method" value="PUT">  <input type="submit" value="testRESTPut">  </form>    <!-- 实验4 测试 REST风格 DELETE 请求 -->  <form action="springmvc/testRESTDelete/1" method="POST">  <input type="hidden" name="\_method" value="DELETE">  <input type="submit" value="testRESTDelete">  </form> |

# 第4章 处理请求数据

请求数据 : 请求参数 cookie信息 请求头信息…..

JavaWEB : HttpServletRequest

Request.getParameter(参数名); Request.getParameterMap();

Request.getCookies();

Request.getHeader();

## 4.1请求处理方法签名

1. Spring MVC 通过分析处理方法的签名(方法名+ 参数列表)，HTTP请 求信息绑定到处理方法的相应形参中。
2. Spring MVC 对控制器处理方法签名的限制是很宽松的，几乎可以按喜欢的任何方式对方法进行签名。
3. 必要时可以对方法及方法入参标注相应的注解（ @PathVariable 、@RequestParam、@RequestHeader 等）、
4. Spring MVC 框架会将 HTTP 请求的信息绑定到相应的方法入参中，并根据方法的返回值类型做出相应的后续处理。

## 4.2 @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

2）value：参数名

3）required：是否必须。默认为 true, 表示请求参数中必须包含对应的参数，若不存在，将抛出异常

4）defaultValue: 默认值，当没有传递参数时使用该值

### 4.2.1 实验代码

* 1. 增加控制器方法

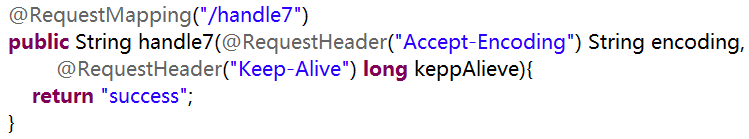
|  |
| --- |
| /\*\*  \* @RequestParam 注解用于映射请求参数  \*         value 用于映射请求参数名称  \*         required 用于设置请求参数是否必须的  \*         defaultValue 设置默认值，当没有传递参数时使用该值  \*/  @RequestMapping(value="/testRequestParam")  public String testRequestParam(  **@RequestParam(value="username")** String username,  **@RequestParam(value="age",required=false,defaultValue="0")** int age){  System.out.println("testRequestParam - username="+username +",age="+age);  return "success";  } |

* 1. 增加页面链接

|  |
| --- |
| <!--测试 请求参数 @RequestParam 注解使用 -->  <a href="springmvc/testRequestParam?username=atguigu&age=10">testRequestParam</a> |

## 4.3 @RequestHeader 注解

1. 使用 @RequestHeader 绑定请求报头的属性值
2. 请求头包含了若干个属性，服务器可据此获知客户端的信息，**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**

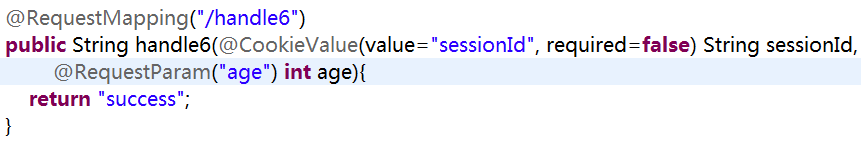


### 4.3.1 实验代码

|  |
| --- |
| //了解: 映射请求头信息 用法同 @RequestParam  @RequestMapping(value="/testRequestHeader")  public String testRequestHeader(**@RequestHeader(value="Accept-Language")** String al){  System.out.println("testRequestHeader - Accept-Language："+al);  return "success";  } |
| <!-- 测试 请求头@RequestHeader 注解使用 -->  <a href="springmvc/testRequestHeader">testRequestHeader</a> |

## 4.4 @CookieValue 注解

1. 使用 @CookieValue 绑定请求中的 Cookie 值
2. **@CookieValue** 可让处理方法入参绑定某个 Cookie 值



### 4.4.1实验代码

1. 增加控制器方法

|  |
| --- |
| //了解:@CookieValue: 映射一个 Cookie 值. 属性同 @RequestParam  @RequestMapping("/testCookieValue")  public String testCookieValue(**@CookieValue("JSESSIONID")** String sessionId) {  System.out.println("testCookieValue: sessionId: " + sessionId);  return "success";  } |

1. 增加页面链接

|  |
| --- |
| <!--测试 请求Cookie @CookieValue 注解使用 -->  <a href="springmvc/testCookieValue">testCookieValue</a> |

## 4.5 使用POJO作为参数

1. 使用 POJO 对象绑定请求参数值
2. 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**。**支持级联属性**。如：dept.deptId、dept.address.tel 等

### 4.5.1实验代码

1. 增加控制器方法、表单页面

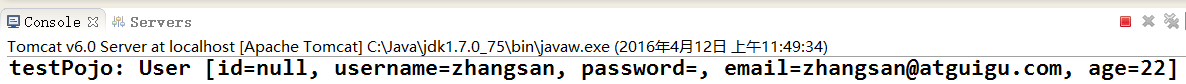
|  |
| --- |
| /\*\*  \* Spring MVC 会按请求参数名和 POJO 属性名进行自动匹配， 自动为该对象填充属性值。  \* 支持级联属性  \*                 如：dept.deptId、dept.address.tel 等  \*/  @RequestMapping("/testPOJO")  public String testPojo(**User user**) {  System.out.println("testPojo: " + user);  return "success";  } |

|  |  |
| --- | --- |
| <!-- 测试 POJO 对象传参，支持级联属性 -->  <form action=" testPOJO" method="POST">  username: <input type="text" name="username"/><br>  password: <input type="password" name="password"/><br>  email: <input type="text" name="email"/><br>  age: <input type="text" name="age"/><br>  city: <input type="text" name="**address.city**"/><br>  province: <input type="text" name="**address.province**"/>  <input type="submit" value="Submit"/>  </form> |  |

1. 增加实体类

|  |  |
| --- | --- |
| package com.atguigu.springmvc.entities;    public class Address {    private String province;  private String city;    //get/set    } | package com.atguigu.springmvc.entities;    public class User {  private Integer id ;  private String username;  private String password;    private String email;  private int age;    private Address address;    //get/set  } |

1. 执行结果：



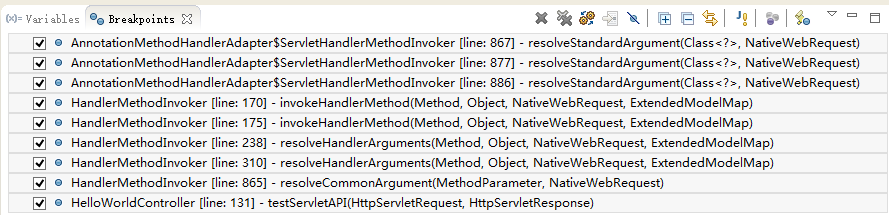
1. 如果中文有乱码，需要配置字符编码过滤器，且配置其他过滤器之前，

如（HiddenHttpMethodFilter），否则不起作用。（思考method=”get”请求的乱码问题怎么解决的）

|  |
| --- |
| **<!-- 配置字符集 -->**  <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

## 4.6 使用Servlet原生API作为参数

1. MVC 的 Handler 方法可以接受哪些 ServletAPI 类型的参数
2. HttpServletRequest
3. HttpServletResponse
4. HttpSession
5. **java.security.Principal**
6. **Locale**
7. **InputStream**
8. **OutputStream**
9. **Reader**
10. **Writer**
11. 源码参考：AnnotationMethodHandlerAdapter L866



|  |
| --- |
| @Override  protected Object resolveStandardArgument(Class<?> parameterType, NativeWebRequest webRequest) throws Exception {  HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);  HttpServletResponse response = webRequest.getNativeResponse(HttpServletResponse.class);    if (**ServletRequest**.class.isAssignableFrom(parameterType) ||  MultipartRequest.class.isAssignableFrom(parameterType)) {  Object nativeRequest = webRequest.getNativeRequest(parameterType);  if (nativeRequest == null) {  throw new IllegalStateException(  "Current request is not of type [" + parameterType.getName() + "]: " + request);  }  return nativeRequest;  }  else if (**ServletResponse**.class.isAssignableFrom(parameterType)) {  this.responseArgumentUsed = true;  Object nativeResponse = webRequest.getNativeResponse(parameterType);  if (nativeResponse == null) {  throw new IllegalStateException(  "Current response is not of type [" + parameterType.getName() + "]: " + response);  }  return nativeResponse;  }  else if (**HttpSession**.class.isAssignableFrom(parameterType)) {  return request.getSession();  }  else if (**Principal**.class.isAssignableFrom(parameterType)) {  return request.getUserPrincipal();  }  else if (**Locale**.class.equals(parameterType)) {  return RequestContextUtils.getLocale(request);  }  else if (**InputStream**.class.isAssignableFrom(parameterType)) {  return request.getInputStream();  }  else if (**Reader**.class.isAssignableFrom(parameterType)) {  return request.getReader();  }  else if (**OutputStream**.class.isAssignableFrom(parameterType)) {  this.responseArgumentUsed = true;  eturn response.getOutputStream();  }  else if (**Writer**.class.isAssignableFrom(parameterType)) {  this.responseArgumentUsed = true;  return response.getWriter();  }  return super.resolveStandardArgument(parameterType, webRequest);  } |

### 4.6.1 实验代码

|  |
| --- |
| /\*\*  \* 可以使用 Serlvet 原生的 API 作为目标方法的参数 具体支持以下类型  \*  \* HttpServletRequest  \* HttpServletResponse  \* HttpSession  \* java.security.Principal  \* Locale InputStream  \* OutputStream  \* Reader  \* Writer  \* @throws IOException  \*/  @RequestMapping("/testServletAPI")  public void testServletAPI(HttpServletRequest request,HttpServletResponse response, Writer out) throws IOException {  System.out.println("testServletAPI, " + request + ", " + response);  out.write("hello springmvc");  //return "success";  } |

|  |
| --- |
| <!-- 测试 Servlet API 作为处理请求参数 -->  <a href="springmvc/testServletAPI">testServletAPI</a> |

# 第5章 处理响应数据

javaWEB： request.setAttribute(xxx) request.getRequestDispatcher(“地址”).forward(req,resp);

## 5.1 SpringMVC 输出模型数据概述

### 5.1.1提供了以下几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
2. **Map 或 Model**: 入参为 org.springframework.ui.Model、

org.springframework.ui.ModelMap 或 java.uti.Map 时，处理方法返回时，Map 中的数据会自动添加到模型中。

## 5.2处理模型数据之 ModelAndView

### ModelAndView介绍

控制器处理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. 两个重要的成员变量:

private Object view; 视图信息

private ModelMap model; 模型数据

3）添加模型数据:

MoelAndView addObject(String attributeName, Object attributeValue) 设置模型数据

ModelAndView addAllObject(Map<String, ?> modelMap)

4）设置视图:

void setView(View view) 设置视图对象

void setViewName(String viewName) 设置视图名字

5）获取模型数据

protected Map<String, Object> getModelInternal() 获取模型数据

public ModelMap getModelMap()

public Map<String, Object> getModel()

### 5.2.2 实验代码

1. 增加控制器方法

|  |
| --- |
| /\*\*  \* 目标方法的返回类型可以是ModelAndView类型  \*                 其中包含视图信息和模型数据信息  \*/  @RequestMapping("/testModelAndView")  public ModelAndView testModelAndView(){  System.out.println("testModelAndView");  String viewName = "success";  **ModelAndView mv = new ModelAndView(viewName );**  **mv.addObject("time",new Date().toString()); //实质上存放到request域中**  return mv;  } |

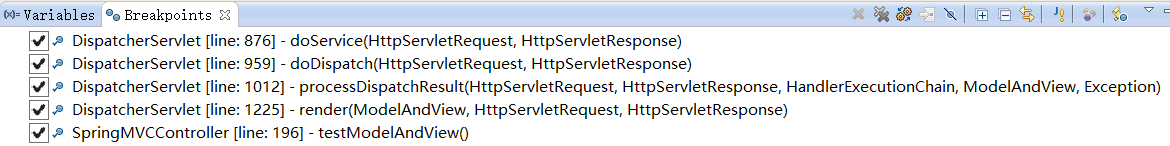
1. 增加页面链接

|  |
| --- |
| <!--测试 ModelAndView 作为处理返回结果 -->  <a href="springmvc/testModelAndView">testModelAndView</a> |

1. 增加成功页面，显示数据

|  |
| --- |
| time: ${requestScope.time } |

1. 断点调试



### 5.2.2 源码解析

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

## 5.3 处理模型数据之 Map Model

### Map介绍

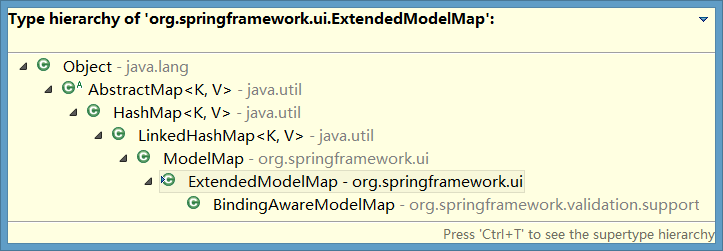
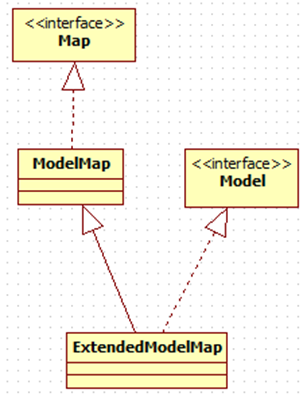
1）Spring MVC 在内部使用了一个 org.springframework.ui.Model 接口存储模型数据

具体使用步骤

**2）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**。

**3）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。

4）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**



### 5.3.2 实验代码

1. 增加控制器方法

|  |
| --- |
| //目标方法的返回类型也可以是一个Map类型参数（也可以是Model,或ModelMap类型）  @RequestMapping("/testMap")  public String testMap(**Map<String,Object> map**){ //**【重点】**  System.out.println(map.getClass().getName());  //**org.springframework.validation.support.BindingAwareModelMap**  map.put("names", Arrays.asList("Tom","Jerry","Kite"));  return "success";  } |

1. 增加页面链接

|  |
| --- |
| <!-- 测试 Map 作为处理返回结果 -->  <a href="springmvc/testMap">testMap</a> |

1. 增加成功页面，显示结果

|  |
| --- |
| names: ${requestScope.names } |

1. 显示结果截图

|  |
| --- |
|  |

1. 注意问题：Map集合的泛型，key为String,Value为Object,而不是String
2. 测试参数类型

|  |
| --- |
| //目标方法的返回类型也可以是一个Map类型参数（也可以是Model,或ModelMap类型）  @RequestMapping("/testMap2")  public String testMap2(**Map<String,Object>** map,**Model** model,**ModelMap** modelMap){  System.out.println(map.getClass().getName());  map.put("names", Arrays.asList("Tom","Jerry","Kite"));  model.addAttribute("model", "org.springframework.ui.Model");  modelMap.put("modelMap", "org.springframework.ui.ModelMap");    System.out.println(map == model);  System.out.println(map == modelMap);  System.out.println(model == modelMap);    System.out.println(map.getClass().getName());  System.out.println(model.getClass().getName());  System.out.println(modelMap.getClass().getName());    /\*  true  true  true  org.springframework.validation.support.**BindingAwareModelMap**  org.springframework.validation.support.**BindingAwareModelMap**  org.springframework.validation.support.**BindingAwareModelMap**  \*/  return "success";  } |

1. 类层次结构



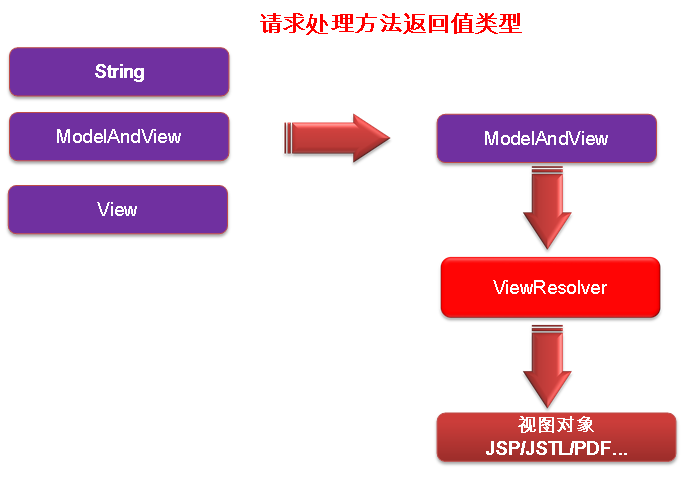
1. 推荐：Map, 便于框架移植。
2. 源码参考

|  |
| --- |
| public class **BindingAwareModelMap** extends ExtendedModelMap {    @Override  public Object **put**(String key, Object value) {  removeBindingResultIfNecessary(key, value);  return super.**put**(key, value);  }    @Override  public void **putAll**(Map<? extends String, ?> map) {  for (Map.Entry<? extends String, ?> entry : map.entrySet()) {  removeBindingResultIfNecessary(entry.getKey(), entry.getValue());  }  super.**putAll**(map);  }    private void removeBindingResultIfNecessary(Object key, Object value) {  if (key instanceof String) {  String attributeName = (String) key;  if (!attributeName.startsWith(BindingResult.MODEL\_KEY\_PREFIX)) {  String bindingResultKey = BindingResult.MODEL\_KEY\_PREFIX + attributeName;  BindingResult bindingResult = (BindingResult) get(bindingResultKey);  if (bindingResult != null && bindingResult.getTarget() != value) {  remove(bindingResultKey);  }  }  }  }  } |

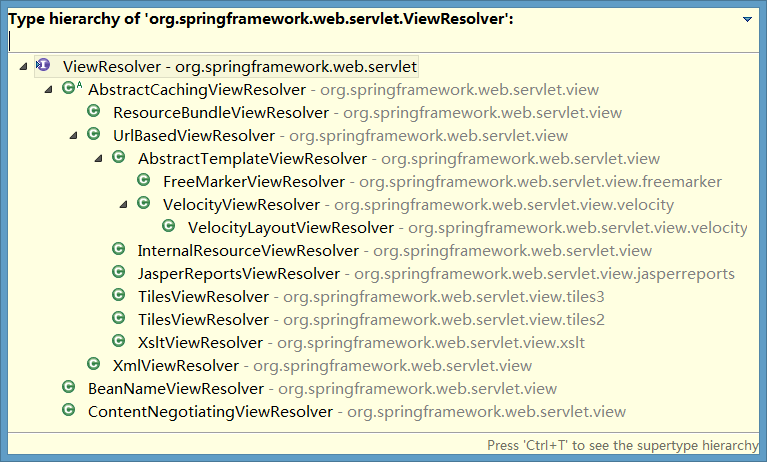
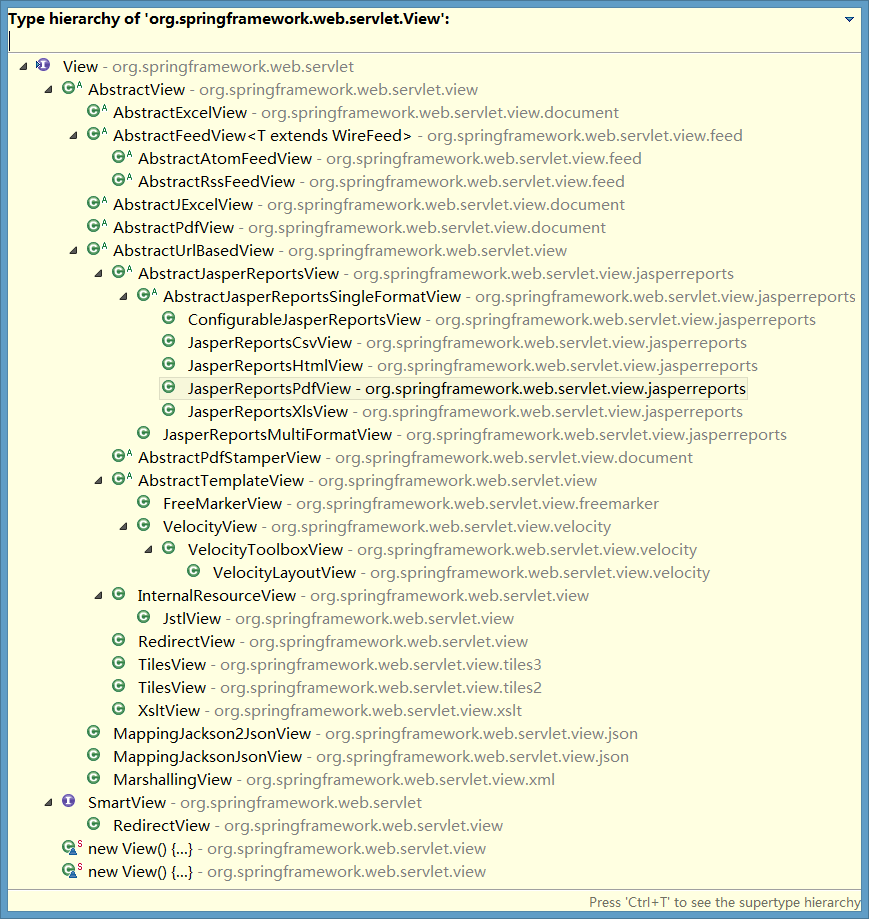
# 第6章 视图解析

## 6.1 SpringMVC如何解析视图概述

1. 不论控制器返回一个String,ModelAndView,View都会转换为ModelAndView对象，由视图解析器解析视图，然后，进行页面的跳转。



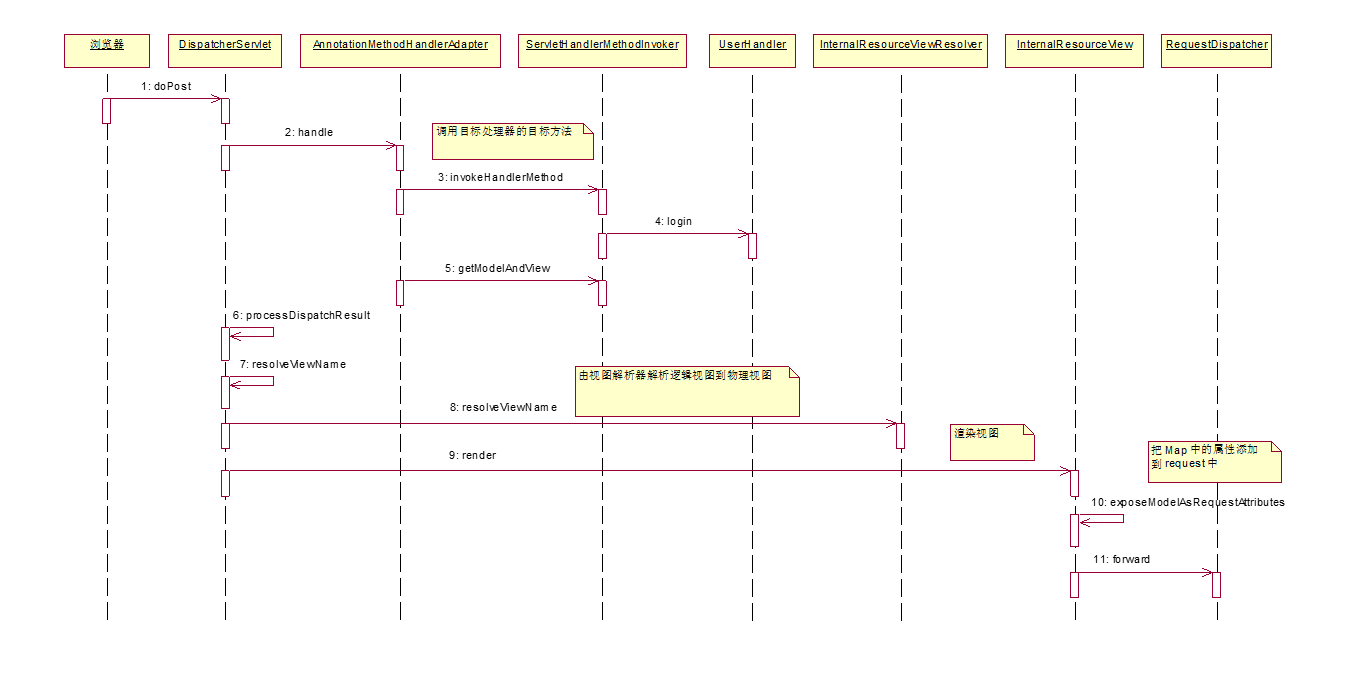
1. 视图解析源码分析：重要的两个接口



1. 断点调试源码



1. 流程图

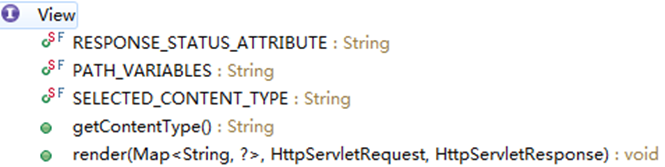


## 6.2 视图和视图解析器

1. 请求处理方法执行完成后，最终返回一个 ModelAndView 对象。对于那些返回 String，View 或 ModeMap 等类型的处理方法，**Spring MVC 也会在内部将它们装配成一个 ModelAndView 对象**，它包含了逻辑名和模型对象的视图
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

## 6.3 视图

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. 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：



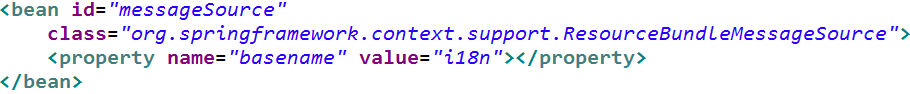
1. **视图对象由视图解析器负责实例化**。由于视图是**无状态**的，所以他们不会有**线程安全**的问题

## 6.4 常用的视图实现类



## 6.5 JstlView

1. 若项目中使用了JSTL，则SpringMVC 会自动把视图由InternalResourceView转为 **JstlView （断点调试，将JSTL的jar包增加到项目中，视图解析器会自动修改为JstlView）**
2. 若使用 JSTL 的 fmt 标签则需要在 SpringMVC 的配置文件中**配置国际化资源文件**

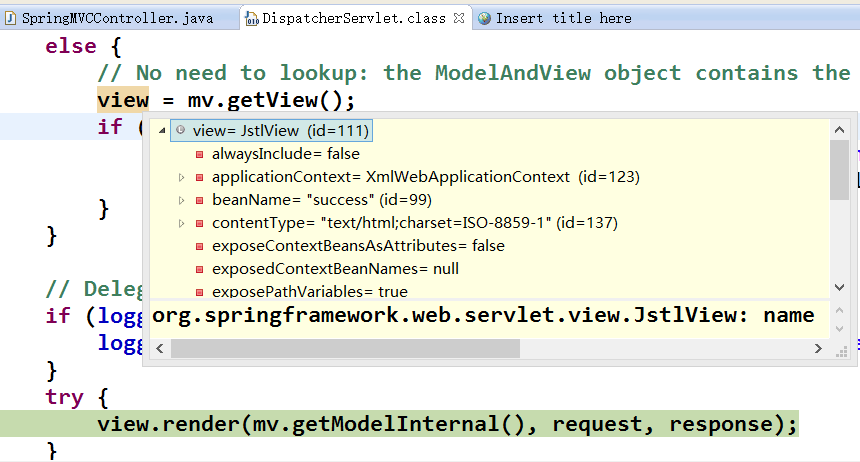


1. 若希望直接响应通过 SpringMVC 渲染的页面，可以使用 **mvc:view-controller** 标签实现



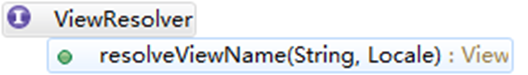
### 6.5.1 实验代码

1）增加jstl标签 jar包（**断点调试，这时的View对象就是JstlView**）

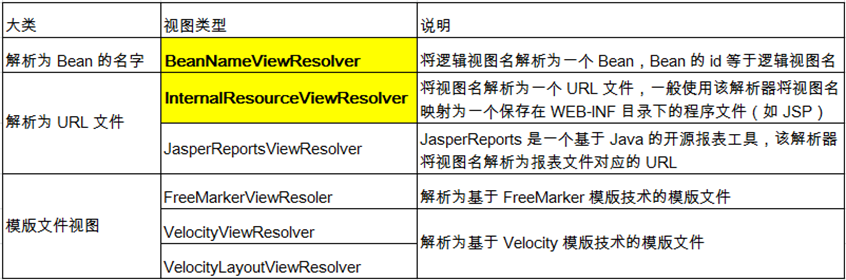


## 6.6 视图解析器

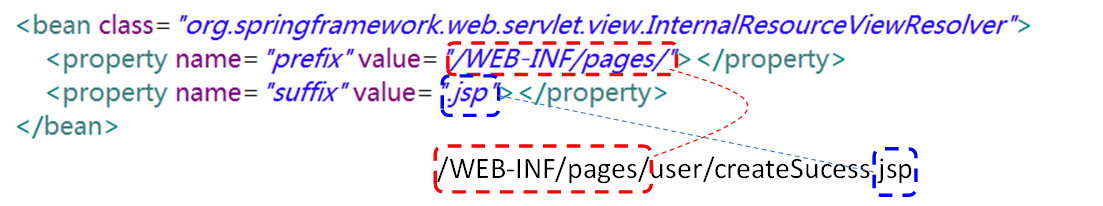
1. SpringMVC 为逻辑视图名的解析提供了不同的策略，可以在 SpringMVC 上下文中**配置一种或多种解析策略**，**并指定他们之间的先后顺序**。每一种映射策略对应一个具体的视图解析器实现类。
2. 视图解析器的作用比较单一：将逻辑视图解析为一个具体的视图对象。
3. 所有的视图解析器都必须实现 ViewResolver 接口：



## 6.7 常用的视图解析器实现类



1. 程序员可以选择一种视图解析器或混用多种视图解析器
2. 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**。
3. 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
4. InternalResourceViewResolver
   1. JSP 是最常见的视图技术，可以使用 InternalResourceViewResolve作为视图解析器：



## mvc:view-controller标签

1）若希望直接响应通过 SpringMVC 渲染的页面，可以使用 **mvc:view-controller** 标签实现

|  |
| --- |
| <!-- 直接配置响应的页面：无需经过控制器来执行结果 -->  <mvc:view-controller path="**/success**" view-name="success"/> |

2）请求的路径：

|  |
| --- |
| [http://localhost:8080/SpringMVC\_02\_View**/success**](http://localhost:8080/SpringMVC_02_View/success) |

3）配置<mvc:view-controller>会导致其他请求路径失效

解决办法：

|  |
| --- |
| <!-- 在实际开发过程中都需要配置mvc:annotation-driven标签，后面讲，这里先配置上 -->  <mvc:annotation-driven/> |

## 6.9重定向

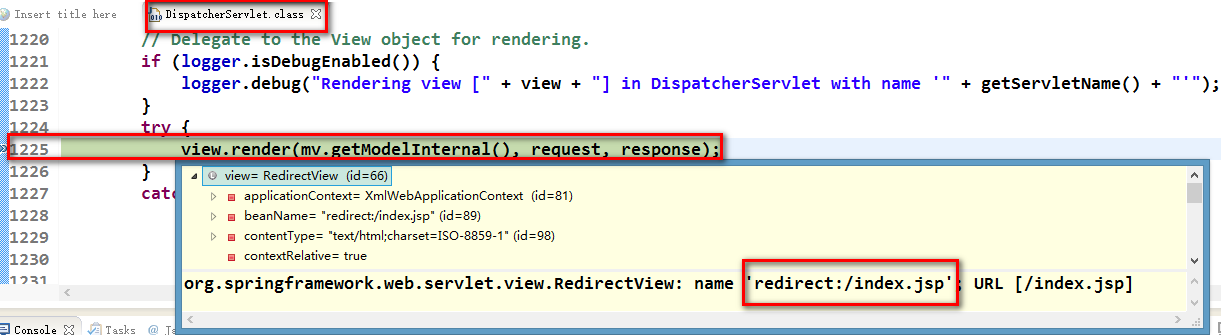
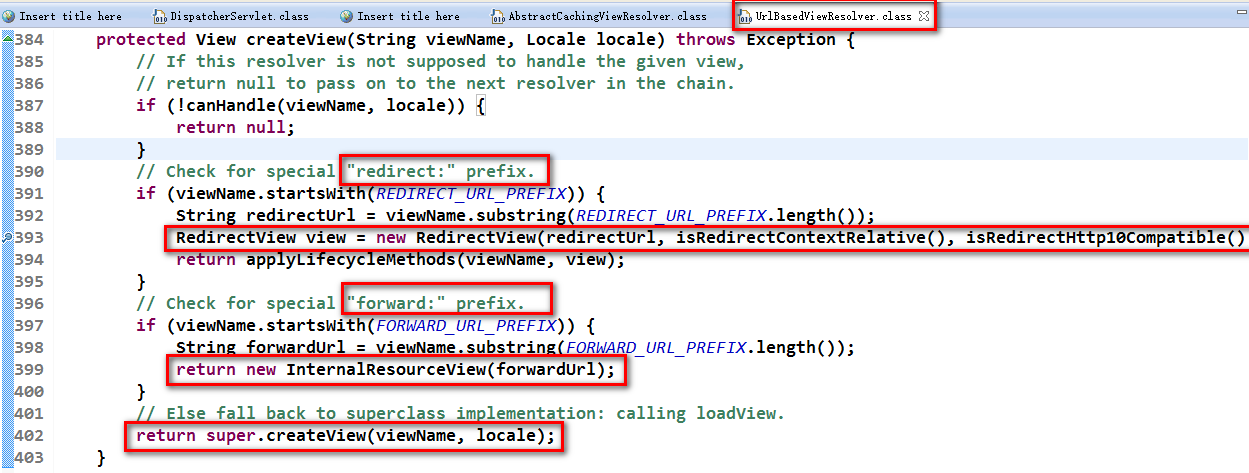
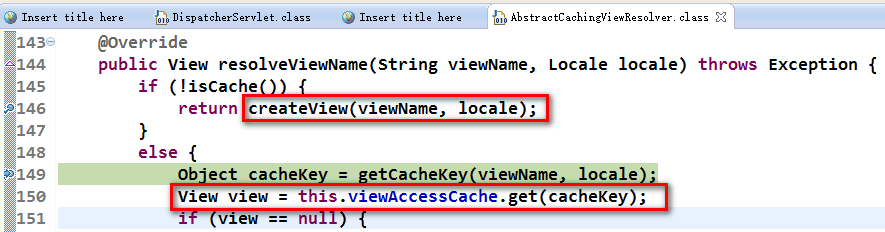
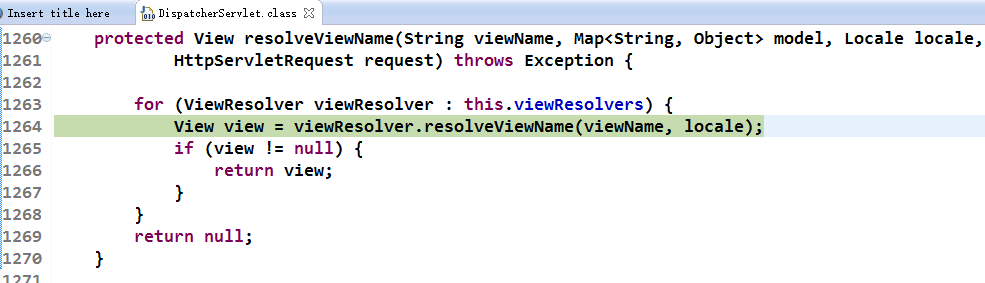
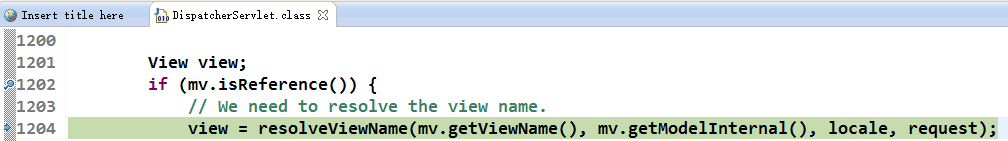
1. 关于重定向
2. 一般情况下，控制器方法返回字符串类型的值会被当成逻辑视图名处理
3. 如果返回的字符串中带 **forward: 或 redirect:** 前缀时，SpringMVC 会对他们进行特殊处理：将 forward: 和 redirect: 当成指示符，其后的字符串作为 URL 来处理
4. redirect:success.jsp：会完成一个到 success.jsp 的重定向的操作
5. forward:success.jsp：会完成一个到 success.jsp 的转发操作
6. 定义页面链接

|  |
| --- |
| <a href="springmvc/testRedirect">testRedirect</a> |

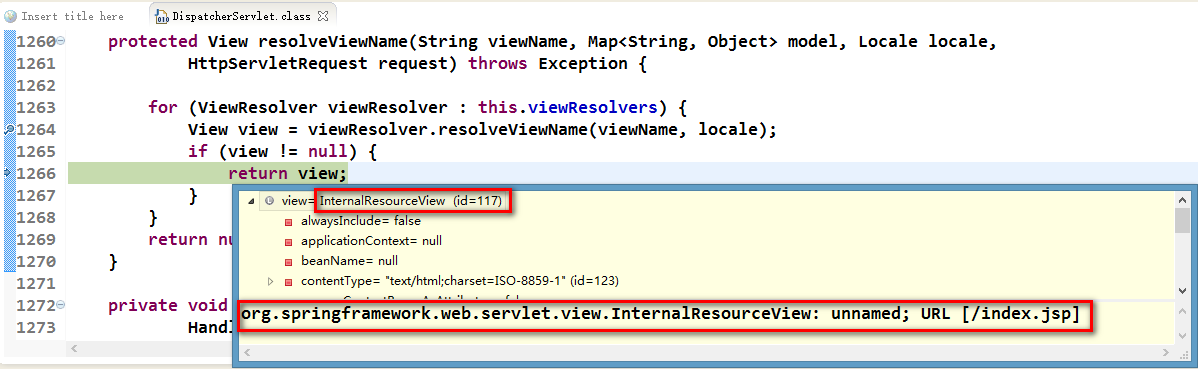
1. 定义控制器方法

|  |
| --- |
| @RequestMapping("/testRedirect")  public String testRedirect(){  System.out.println("testRedirect");  **return "redirect:/index.jsp";**  **//return "forward:/index.jsp";**  } |

1. 源码分析：重定向原理
2. 源码分析：重定向原理



* return "forward:/index.jsp"

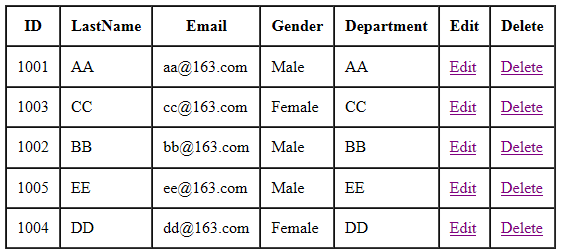


# 第7章 综合案例RESTRUL\_CRUD

## 7.1 RESTRUL\_CRUD\_需求

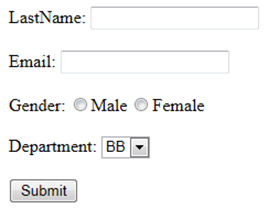
### 7.1.1 显示所有员工信息

1. URI：**emps**
2. 请求方式：**GET**
3. 显示效果



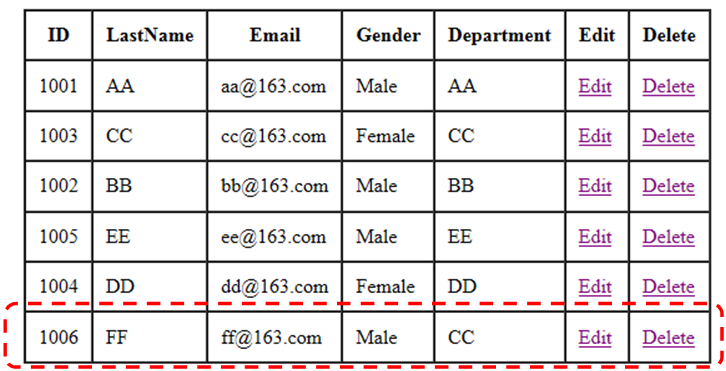
### 7.1.2 添加操作-去往添加页面

1. 显示添加页面：
2. URI：**emp**
3. 请求方式：**GET**
4. 显示效果



### 7.1.3 添加操作-添加员工

1. 添加员工信息：
2. URI：**emp**
3. 请求方式：**POST**
4. 显示效果：完成添加，**重定向**到 list 页面。



### 7.1.4 删除操作

1. URL：**emp/{id}**
2. 请求方式：**DELETE**
3. 删除后效果：对应记录从数据表中删除

### 7.1.5 修改操作-去往修改页面

1. URI：**emp/{id}**
2. 请求方式：**GET**
3. 显示效果：**回显表单**。

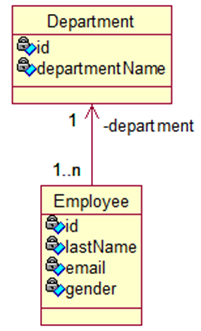
### 7.1.6 修改操作-修改员工

1. URI：**emp**
2. 请求方式：**PUT**
3. 显示效果：完成修改，**重定向**到 list 页面。

### 7.1.7 相关的类

省略了Service层，为了教学方便

1. 实体类：Employee、Department
2. Handler：**EmployeeHandler**
3. Dao：EmployeeDao、DepartmentDao



### 7.1.8 相关的页面

1. list.jsp
2. input.jsp
3. edit.jsp

## 7.2 搭建开发环境

* 1. 拷贝jar包

|  |
| --- |
| **com.springsource.net.sf.cglib-2.2.0.jar**  **com.springsource.org.aopalliance-1.0.0.jar**  **com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar**  **spring-aop-4.0.0.RELEASE.jar**  **spring-aspects-4.0.0.RELEASE.jar**  commons-logging-1.1.3.jar  spring-beans-4.0.0.RELEASE.jar  spring-context-4.0.0.RELEASE.jar  spring-core-4.0.0.RELEASE.jar  spring-expression-4.0.0.RELEASE.jar  **spring-jdbc-4.0.0.RELEASE.jar**  **spring-orm-4.0.0.RELEASE.jar**  **spring-tx-4.0.0.RELEASE.jar**  spring-web-4.0.0.RELEASE.jar  spring-webmvc-4.0.0.RELEASE.jar |

* 1. 创建配置文件:springmvc.xml 增加context,mvc,beans名称空间。

|  |
| --- |
| <?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/mvc <http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd>  <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-4.0.xsd>">    <!-- 配置扫描的包：com.atguigu.springmvc.crud -->  <context:component-scan base-package="com.atguigu.springmvc"/>    <!-- 配置视图解析器：默认采用转发 -->  <bean id="internalResourceViewResolver"  class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/views/"/>  <property name="suffix" value=".jsp"></property>  </bean>  </beans> |

* 1. 配置核心控制器：web.xml

|  |
| --- |
| <servlet>  <servlet-name>springDispatcherServlet</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>springDispatcherServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |

* 1. 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. 创建相关页面

/WEB-INF/views/list.jsp

index.jsp

* 1. 增加实体类

|  |  |
| --- | --- |
|  |  |

* 1. 增加DAO类

|  |  |
| --- | --- |
| package com.atguigu.springmvc.crud.dao;    import java.util.Collection;  import java.util.HashMap;  import java.util.Map;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Repository;    import com.atguigu.springmvc.crud.entities.Department;  import com.atguigu.springmvc.crud.entities.Employee;    @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);  }  } | package com.atguigu.springmvc.crud.dao;    import java.util.Collection;  import java.util.HashMap;  import java.util.Map;    import org.springframework.stereotype.Repository;    import com.atguigu.springmvc.crud.entities.Department;    @Repository  public class **DepartmentDao** {    private static Map<Integer, Department> departments = null;    static{  departments = new LinkedHashMap<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);  }    } |

## 7.3 RESTRUL\_CRUD\_显示所有员工信息

* 1. 增加页面链接

|  |
| --- |
| <a href="empList">To Employee List</a> |

* 1. 增加处理器

|  |
| --- |
| package com.atguigu.springmvc.crud.handlers;    import java.util.Map;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;    import com.atguigu.springmvc.crud.dao.EmployeeDao;    @Controller  public class EmployeeHandler {    @Autowired  private EmployeeDao employeeDao ;    @RequestMapping("/empList")  public String empList(**Map<String,Object> map**){  map.put("empList", employeeDao.getAll());        //默认存放到request域中  return "list";  }  } |

* 1. SpringMVC中没遍历的标签，需要使用jstl标签进行集合遍历增加jstl标签库jar包

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  **<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>**    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01  Transitional//EN" "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>  <c:if test="${empty requestScope.empList }">  对不起，没有找到任何员工！  </c:if>  <c:if test="${!empty requestScope.empList }">  <table border="1" cellpadding="10" cellspacing="0">  <tr>  <td>EmpId</td>  <td>LastName</td>  <td>Gender</td>  <td>Email</td>  <td>DepartmentName</td>  <td>Edit</td>  <td>Delete</td>  </tr>  **<c:forEach items="${requestScope.empList }" var="emp">**  <tr>  <td>${emp.id }</td>  <td>${emp.lastName }</td>  <td>${emp.gender==0?"Female":"Male" }</td>  <td>${emp.email }</td>  <td>${emp.department.departmentName }</td>  <td><a href="">Edit</a></td>  <td><a href="">Delete</a></td>  </tr>  **</c:forEach>**  </table>  </c:if>    </body>  </html> |

## 7.4 RESTRUL\_CRUD\_添加操作

1. 在list.jsp上增加连接

|  |
| --- |
| <a href="empInput">Add Employee</a> |

1. 增加处理器方法

|  |
| --- |
| @RequestMapping(value="/empInput",method=RequestMethod.GET)  public String empInput(Map<String,Object> map){  map.put("deptList", departmentDao.getDepartments());  //解决错误：java.lang.IllegalStateException: Neither BindingResult nor plain target object for bean name 'command' available as request attribute  Employee employee = new Employee();  //map.put("**command**", employee);  map.put("**employee**", employee);  return "add";  } |

1. 显示添加页面

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8" import="java.util.\*"%>  **<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>**  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01  Transitional//EN" "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>    <!--  1.为什么使用SpringMVC的form标签  ① 快速开发  ② 表单回显  2.可以通过modelAttribute指定绑定的模型属性，  若没有指定该属性，则默认从request域中查找command的表单的bean  如果该属性也不存在，那么，则会发生错误。  -->  <form:form action="empAdd" method="POST" **modelAttribute="employee"**>          LastName : <form:input path="lastName"/><br><br>          Email : <form:input path="email"/><br><br>          <%                  Map<String,String> map = new HashMap<String,String>();                  map.put("1", "Male");                  map.put("0","Female");                  request.setAttribute("genders", map);          %>          Gender : **<br>**<form:radiobuttons path="gender" items="${genders }" **delimiter="<br>"**/><br><br>          DeptName :                  <form:select path="**department.id**"                                                  items="${deptList }"                                                  itemLabel="departmentName"                                                  itemValue="id"></form:select><br><br>  **<input type="submit" value="Submit">**<br><br>  </form:form>  </body>  </html> |

1. 显示表单信息时，会报错：

|  |
| --- |
| **HTTP Status 500 -**    **type** Exception report  **message**  **description** The server encountered an internal error () that prevented it from fulfilling this request.  **exception**  org.apache.jasper.JasperException: An exception occurred processing JSP page /WEB-INF/views/add.jsp at line 18  15:                         ② 表单回显 16:          --> 17:          <form:form action="empAdd" method="POST"> 18:                  LastName : <form:input path="lastName"/> 19:                  Email : <form:input path="email"/> 20:                  <% 21:                          Map<String,String> map = new HashMap<String,String>();  Stacktrace:         org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:505)         org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:410)         org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:337)         org.apache.jasper.servlet.JspServlet.service(JspServlet.java:266)         javax.servlet.http.HttpServlet.service(HttpServlet.java:803)         org.springframework.web.servlet.view.InternalResourceView.renderMergedOutputModel(InternalResourceView.java:209)         org.springframework.web.servlet.view.AbstractView.render(AbstractView.java:266)         org.springframework.web.servlet.DispatcherServlet.render(DispatcherServlet.java:1225)         org.springframework.web.servlet.DispatcherServlet.processDispatchResult(DispatcherServlet.java:1012)         org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:959)         org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:876)         org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:931)         org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:822)         javax.servlet.http.HttpServlet.service(HttpServlet.java:690)         org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:807)         javax.servlet.http.HttpServlet.service(HttpServlet.java:803)         org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal(HiddenHttpMethodFilter.java:77)         org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:108)  **root cause**  java.lang.IllegalStateException: Neither BindingResult nor plain target object for bean name 'command' available as request attribute         org.springframework.web.servlet.support.BindStatus.<init>(BindStatus.java:141) |

## 7.5 使用Spring的表单标签

1. 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**
2. form 标签
   * 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**。
   * **只要满足该最佳条件的契约**，**<form:form> 标签就无需通过 action 属性指定表单提交的 URL**
   * 可以通过 **modelAttribute** 属性指定绑定的模型属性，若没有指定该属性，则默认从 request 域对象中读取 **command** 的表单 bean，如果该属性值也不存在，则会发生错误。
3. SpringMVC 提供了多个表单组件标签，如 <form:input/>、<form:select/> 等，用以绑定表单字段的属性值，它们的共有属性如下：
   * **path**：**表单字段，对应 html 元素的 name 属性，支持级联属性**
   * htmlEscape：是否对表单值的 HTML 特殊字符进行转换，默认值为 true
   * cssClass：表单组件对应的 CSS 样式类名
   * cssErrorClass：表单组件的数据存在错误时，采取的 CSS 样式
4. form:input、form:password、form:hidden、form:textarea：对应 HTML 表单的 text、password、hidden、textarea 标签
5. form:radiobutton：单选框组件标签，当表单 bean 对应的属性值和 value 值相等时，单选框被选中
6. **form:radiobuttons**：单选框组标签，用于构造多个单选框
   * **items**：可以是一个 List、String[] 或 Map
   * **itemValue**：指定 radio 的 value 值。可以是集合中 bean 的一个属性值
   * **itemLabel**：指定 radio 的 label 值
   * **delimiter**：多个单选框可以通过 delimiter 指定分隔符
7. form:checkbox：复选框组件。用于构造单个复选框
8. form:checkboxs：用于构造多个复选框。使用方式同 form:radiobuttons 标签
9. form:select：用于构造下拉框组件。使用方式同 form:radiobuttons 标签
10. form:option：下拉框选项组件标签。使用方式同 form:radiobuttons 标签
11. **form:errors**：显示表单组件或数据校验所对应的错误
    * <form:errors path= “\*” /> ：显示表单所有的错误
    * <form:errors path= “user\*” /> ：显示所有以 user 为前缀的属性对应的错误
    * <form:errors **path= “username”** /> ：显示特定表单对象属性的错误

## 7.6 添加员工实验代码

1. 表单

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8" import="java.util.\*"%>  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>    <!--  1.为什么使用SpringMVC的form标签  ① 快速开发  ② 表单回显  **2.可以通过modelAttribute指定绑定的模型属性，**  **若没有指定该属性，则默认从request域中查找command的表单的bean**  **如果该属性也不存在，那么，则会发生错误。**  -->  <form:form action="empAdd" method="POST" modelAttribute="employee">          LastName : <form:input path="lastName" /><br><br>          Email : <form:input path="email" /><br><br>          <%                  Map<String,String> map = new HashMap<String,String>();                  map.put("1", "Male");                  map.put("0","Female");                  request.setAttribute("genders", map);          %>          Gender : <br><form:radiobuttons path="gender" items="${genders }" **delimiter="<br>"**/><br><br>          DeptName :                  <form:select path="department.id"                                                  items="${deptList }"                                                  itemLabel="departmentName"                                                  itemValue="id"></form:select><br><br>                  <input type="submit" value="Submit"><br><br>  </form:form>  </body>  </html> |

1. 控制器方法

|  |
| --- |
| @Controller  public class EmployeeHandler {  @RequestMapping(value="/empAdd",method=RequestMethod.**POST**)  public String empAdd(Employee employee){  employeeDao.save(employee);  return "redirect:/empList";  }  } |

## 7.7 RESTRUL\_CRUD\_删除操作&处理静态资源

### 7.7.1 删除实验代码

1. 页面链接

|  |
| --- |
| <td><a href="/empDelete/${emp.id }">Delete</a></td> |

1. 控制器方法

|  |
| --- |
| @RequestMapping(value="/empDelete/{id}" ,**method=RequestMethod.DELETE**)  public String empDelete(@PathVariable("id") Integer id){  employeeDao.delete(id);  return "redirect:/empList";  } |

### 7.7.2 HiddenHttpMethodFilter过滤器

发起请求，无法执行，因为delete请求必须通过post请求转换为delete请求，借助：HiddenHttpMethodFilter过滤器

### 7.7.3 需要使用jQuery来转换请求方式

1. 加入jQuery库文件

/scripts/jquery-1.9.1.min.js

1. jQuery库文件不起作用

|  |
| --- |
| 警告: No mapping found for HTTP request with URI [/SpringMVC\_03\_RESTFul\_CRUD/scripts/jquery-1.9.1.min.js] in DispatcherServlet with name 'springDispatcherServlet' |
|  |

1. 解决办法，SpringMVC 处理静态资源
   * 1. 为什么会有这样的问题:

优雅的 REST 风格的资源URL 不希望带 .html 或 .do 等后缀，若将 DispatcherServlet 请求映射配置为 /, 则 Spring MVC 将捕获 WEB 容器的所有请求, 包括静态资源的请求, SpringMVC 会将他们当成一个普通请求处理, 因找不到对应处理器将导致错误。

②解决: 在 SpringMVC 的配置文件中配置 <mvc:default-servlet-handler/>

1. 配置后，原来的请求又不好使了

需要配置<mvc:annotation-driven />

### 7.7.4 关于<mvc:default-servlet-handler/>作用

|  |
| --- |
| <!--  <mvc:default-servlet-handler/> 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  一般 WEB 应用服务器默认的 Servlet 的名称都是 default。  若所使用的 WEB 服务器的默认 Servlet 名称不是 default，则需要通过 default-servlet-name 属性显式指定  参考：**CATALINA\_HOME/config/web.xml**  <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>  该标签属性default-servlet-name默认值是"default",可以省略。  <mvc:default-servlet-handler/>  -->  <mvc:default-servlet-handler **default-servlet-name="default"**/> |

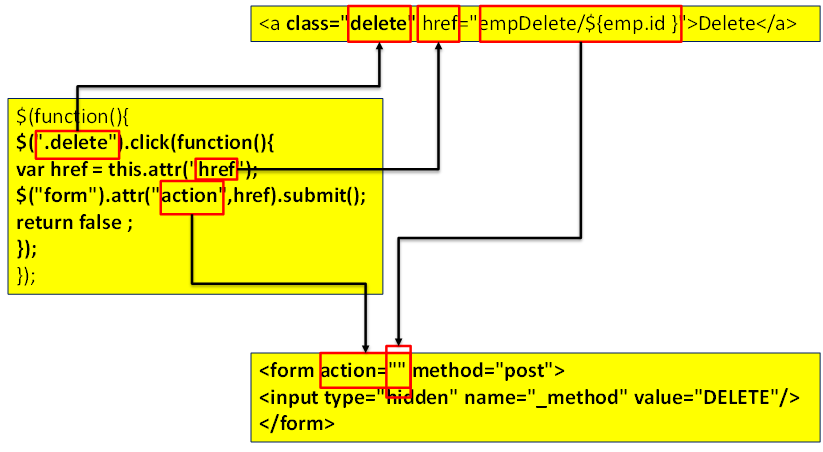
### 7.7.5 通过jQuery转换为DELETE请求

|  |
| --- |
| <td><a **class="delete"** href="empDelete/${emp.id }">Delete</a></td> |

|  |
| --- |
| **<form action="" method="post">**  **<input type="hidden" name="\_method" value="DELETE"/>**  **</form>** |

|  |
| --- |
| <script type="text/javascript" src="scripts/jquery-1.9.1.min.js"></script>  <script type="text/javascript">  $(function(){  **$(".delete").click(function(){**  **var href = $(this).attr("href");**  **$("form").attr("action",href).submit();**  **return false ;**  **});**  });  </script> |

### 7.7.6 删除操作流程图解



## 7.8 RESTRUL\_CRUD\_修改操作

### 7.8.1 根据id查询员工对象，表单回显

1. 页面链接

|  |
| --- |
| <td><a href="empEdit/${emp.id }">Edit</a></td> |

1. 控制器方法

|  |
| --- |
| //修改员工 - 表单回显  @RequestMapping(value="/empEdit/{id}",method=RequestMethod.GET)  public String empEdit(@PathVariable("id") Integer id,Map<String,Object> map){  map.put("**employee**", employeeDao.get(id));  map.put("**deptList**",departmentDao.getDepartments());  return "edit";  } |

1. 修改页面

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8" import="java.util.\*"%>  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>    <!--  1.为什么使用SpringMVC的form标签  ① 快速开发  ② 表单回显  2.可以通过modelAttribute指定绑定的模型属性，  若没有指定该属性，则默认从request域中查找command的表单的bean  如果该属性也不存在，那么，则会发生错误。  **修改功能需要增加绝对路径，相对路径会报错，路径不对**  -->  <**form:form** action="**${pageContext.request.contextPath }**/empUpdate"  method="POST" **modelAttribute="employee"**>  <%--          LastName : <form:input path="lastName" /><br><br>           --%>  **<form:hidden path="id"/>**  **<input type="hidden" name="\_method" value="PUT">**                  <%-- 这里不要使用form:hidden标签，否则会报错。  <form:hidden path="\_method" **value="PUT"**/>                          Spring的隐含标签，**没有value属性**，同时,path指定的值，在**模型对象中也没有这个属性**(\_method)，所以回显时会报错。                          org.springframework.beans.NotReadablePropertyException:                          Invalid property '\_method' of bean class [com.atguigu.springmvc.crud.entities.Employee]:                          Bean property '\_method' is not readable or has an invalid getter method:                          Does the return type of the getter match the parameter type of the setter?                   --%>            Email : <form:input path="email" /><br><br>          <%                  Map<String,String> map = new HashMap<String,String>();                  map.put("1", "Male");                  map.put("0","Female");                  request.setAttribute("genders", map);          %>          Gender : <br><form:radiobuttons path="gender" items="${genders }" delimiter="<br>"/><br><br>          DeptName :                  <form:select path="department.id"                                                  items="${deptList }"                                                  itemLabel="departmentName"                                                  itemValue="id"></form:select><br><br>                  <input type="submit" value="Submit"><br><br>  </**form:form**>  </body>  </html> |

### 7.8.2 提交表单，修改数据

1. 控制器方法

|  |
| --- |
| @RequestMapping(value="/empUpdate",method=RequestMethod.PUT)  public String empUpdate(Employee employee){  employeeDao.save(employee);  return "**redirect**:/empList";  } |

# 第8章 处理JSON

javaWEB: Gson、fastJson、JsonLib、jackson…

## 8.1 返回JSON

1. 加入 jar 包：

<http://wiki.fasterxml.com/JacksonDownload/> 下载地址

jackson-annotations-2.1.5.jar

jackson-core-2.1.5.jar

jackson-databind-2.1.5.jar

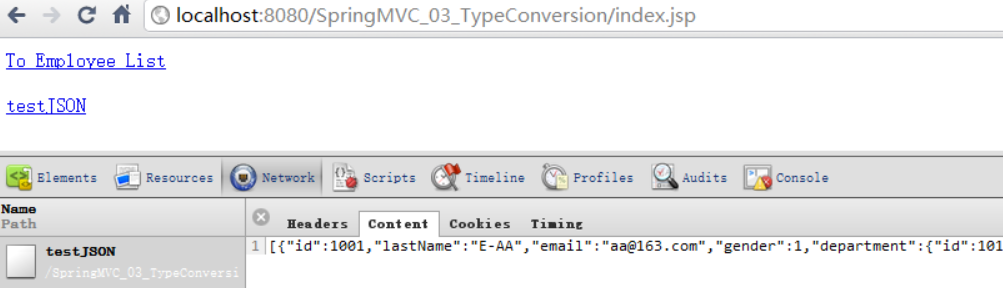
1. 编写目标方法，使其返回 JSON 对应的对象或集合

|  |
| --- |
| **@ResponseBody //SpringMVC对JSON的支持**  @RequestMapping("/testJSON")  public **Collection<Employee>** testJSON(){  return employeeDao.getAll();  } |

1. 增加页面代码：index.jsp

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  <script type="text/javascript" src="scripts/jquery-1.9.1.min.js"></script>  <script type="text/javascript">  **$(function(){**  **$("#testJSON").click(function(){**    **var url = this.href ;**  **var args = {};**  **$.post(url,args,function(data){**  **for(var i=0; i<data.length; i++){**  **var id = data[i].id;**  **var lastName = data[i].lastName ;**  **alert(id+" - " + lastName);**  **}**  **});**    **return false ;**  **});**  **});**  </script>    </head>  <body>    <a href="empList">To Employee List</a>  <br><br>    **<a id="testJSON" href="testJSON">testJSON</a>**    </body>  </html> |

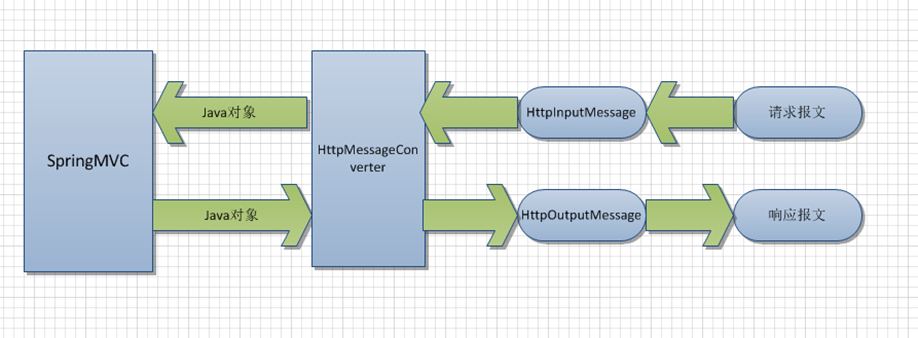
1. 测试



## 8.2 HttpMessageConverter原理

### 8.2.1 HttpMessageConverter<T>

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：**
3. 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,applaiction/json等)
4. 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。
5. List<MediaType> getSupportMediaTypes()：该转换器支持的媒体类型。
6. T read(Class<? extends T> clazz,**HttpInputMessage** inputMessage)：将请求信息流转换为 T 类型的对象。
7. void write(T t,MediaType contnetType,**HttpOutputMessgae** 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。

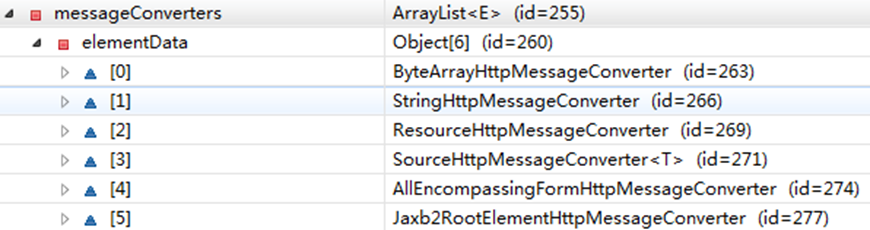


|  |  |
| --- | --- |
| package org.springframework.http;    import java.io.IOException;  import java.io.InputStream;    public interface **HttpInputMessage** extends **HttpMessage** {    InputStream getBody() throws IOException;    } | package org.springframework.http;    import java.io.IOException;  import java.io.OutputStream;    public interface **HttpOutputMessage** extends **HttpMessage** {    OutputStream getBody() throws IOException;    } |



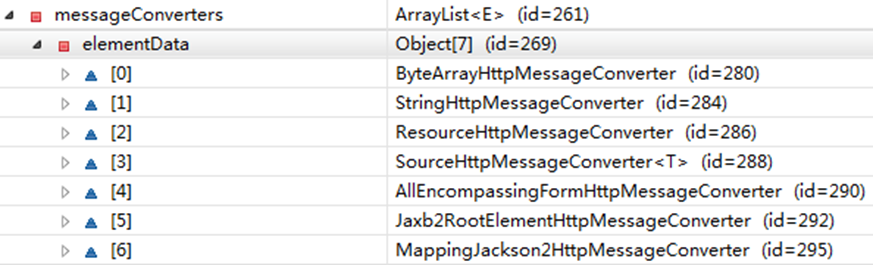
1. DispatcherServlet 默认装配 RequestMappingHandlerAdapter ，

而 RequestMappingHandlerAdapter 默认装配如下 HttpMessageConverter：



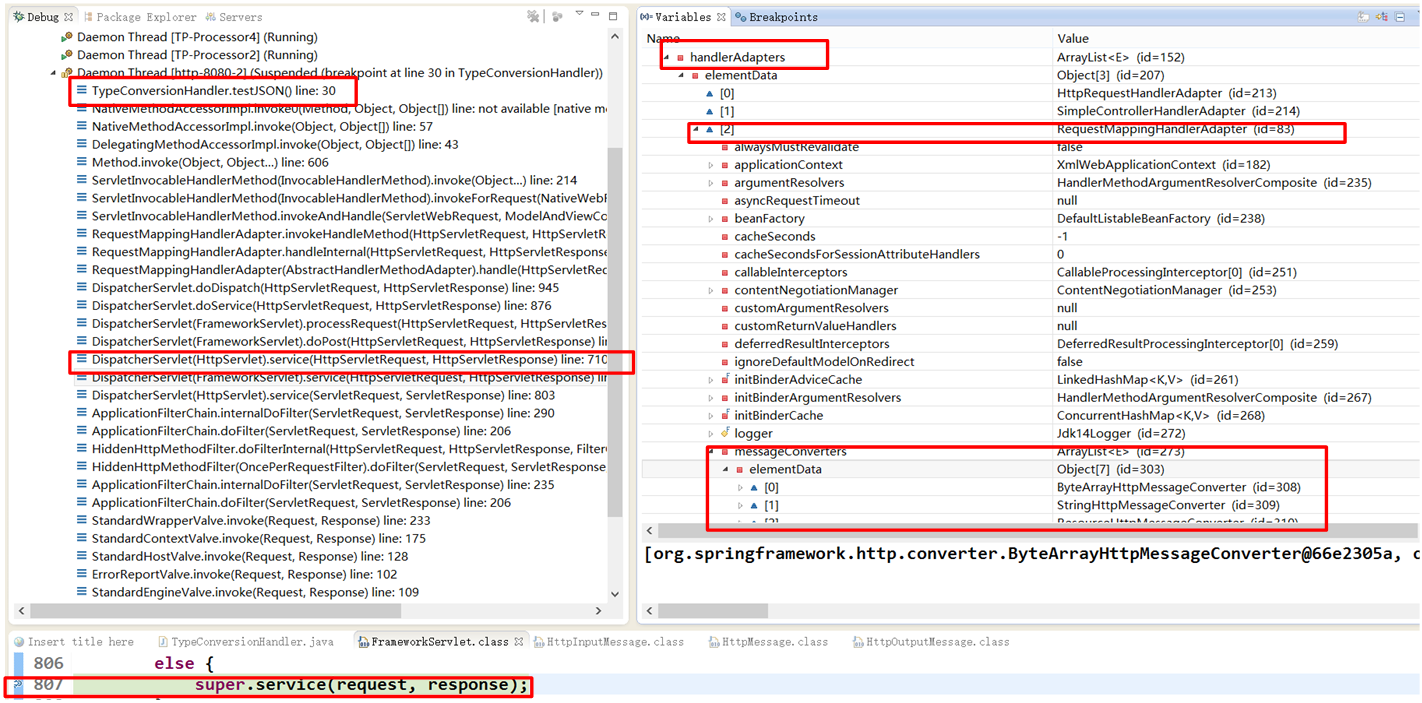
1. 加入 jackson jar 包后， RequestMappingHandlerAdapter

装配的 HttpMessageConverter 如下：



默认情况下数组长度是6个；增加了jackson的包后多了一个

MappingJackson2HttpMessageConverter

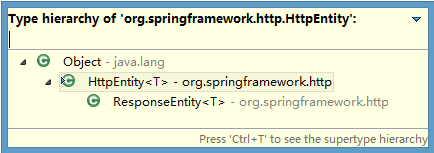


## 8.3 使用HttpMessageConverter

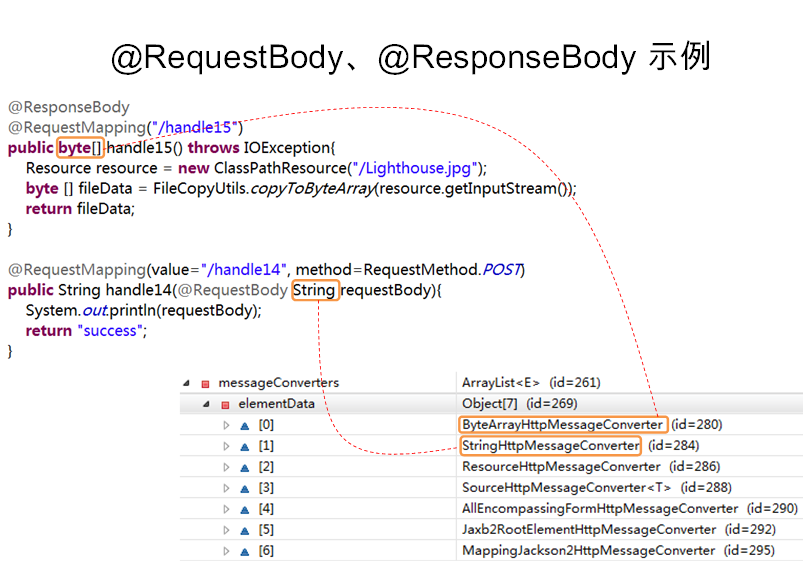
1. **使用 HttpMessageConverter<T> 将请求信息转化并绑定到处理方法的入参中或将响应结果转为对应类型的响应信息，Spring 提供了两种途径**：
   * 使用 **@RequestBody / @ResponseBody** 对处理方法进行标注
   * 使用 **HttpEntity<T> / ResponseEntity<T>** 作为处理方法的入参或返回值
2. 当控制器处理方法使用到 @RequestBody/@ResponseBody 或

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

1. **@RequestBody 和 @ResponseBody 不需要成对出现**
2. Content-Disposition：attachment; filename=abc.pdf



1. @RequestBody @ResponseBody

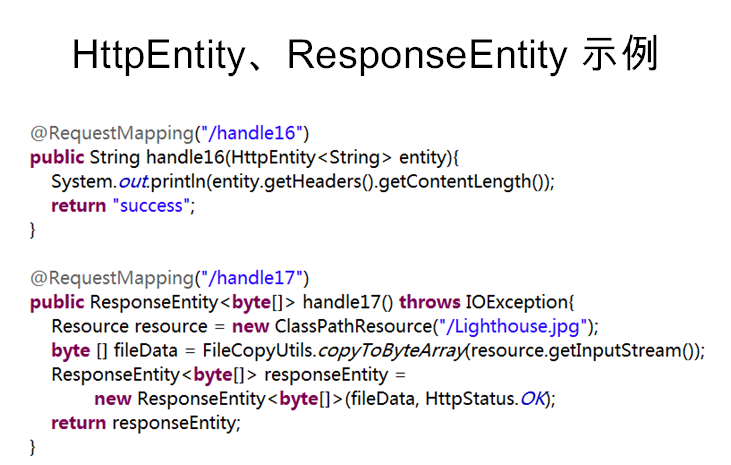


1. 实验代码

|  |
| --- |
| <form action="testHttpMessageConverter" method="post" enctype="multipart/form-data">  文件: <input type="file" name="file"/><br><br>  描述: <input type="text" name="desc"/><br><br>  <input type="submit" value="提交"/>  </form> |

|  |
| --- |
| **@ResponseBody //**@ResponseBody:是将内容或对象作为Http响应正文返回  @RequestMapping("/testHttpMessageConverter")  **//**@RequestBody:是将Http请求正文插入方法中，修饰目标方法的入参  public String testHttpMessageConverter(**@RequestBody** String body){  System.out.println("body="+body);  return "Hello," + new Date(); //不再查找跳转的页面  } |

1. HttpEntity ResponseEntity

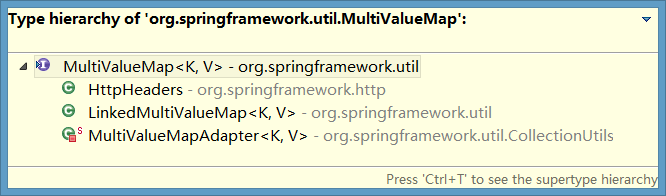


1. 实验代码

|  |
| --- |
| /files/abc.txt 准备一个下载的文件 |

|  |
| --- |
| <a href="testResponseEntity">abc.txt</a> |

|  |
| --- |
| @RequestMapping("testResponseEntity")  public ResponseEntity<byte[]> testResponseEntity(HttpSession session) throws IOException{    ServletContext servletContext = session.getServletContext();  InputStream resourceAsStream = servletContext.getResourceAsStream("/files/abc.txt");  byte[] body = new byte[resourceAsStream.available()] ;  resourceAsStream.read(body);    MultiValueMap<String, String> headers = new HttpHeaders();  headers.add("Content-Disposition", "attachment;filename=abc.txt");    HttpStatus statusCode = HttpStatus.OK;    ResponseEntity<byte[]> responseEntity = new ResponseEntity<byte[]>(body, headers, statusCode);    return responseEntity ;  } |



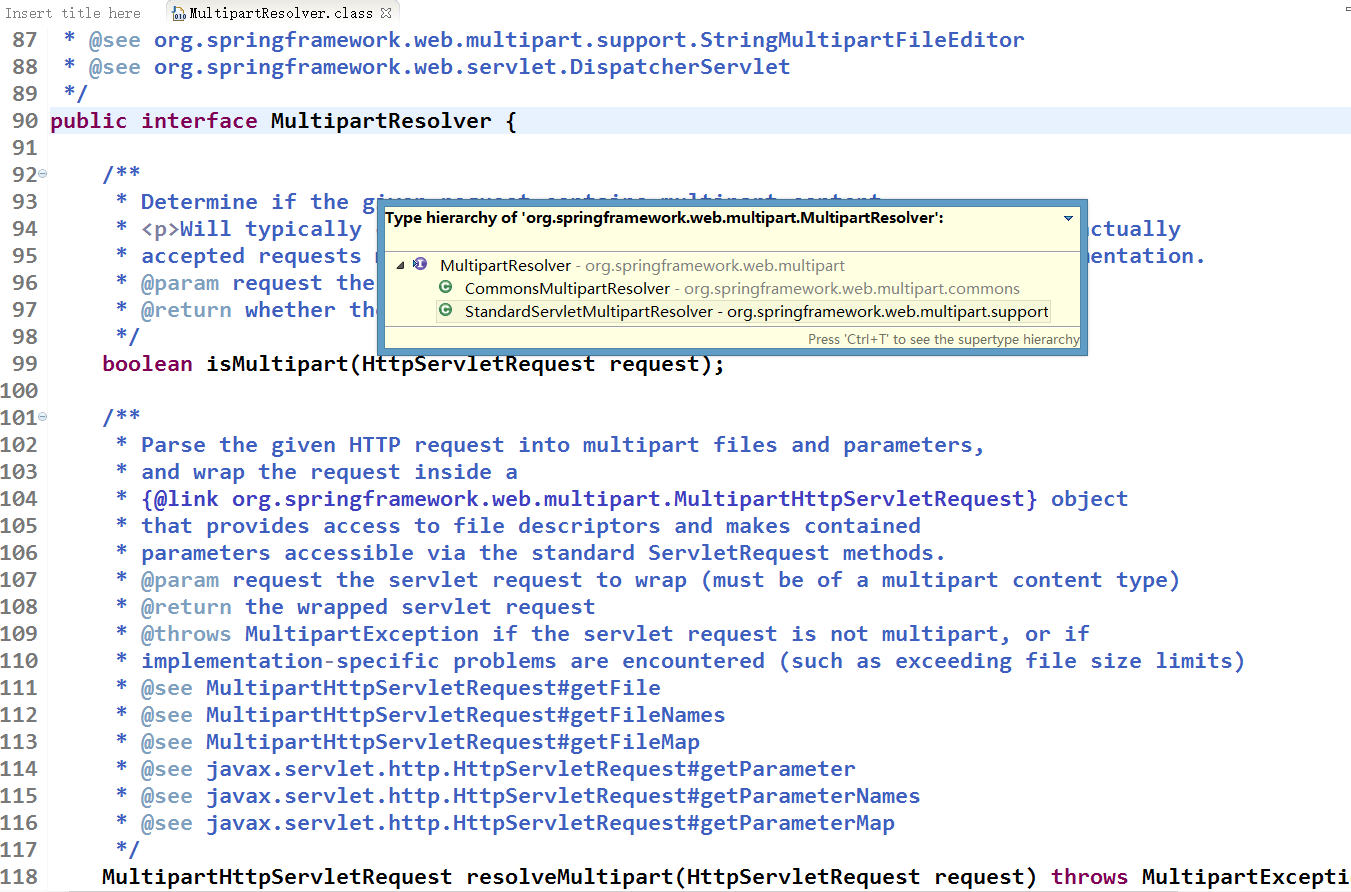
1. 源码参考

* HttpHeaders
* HttpStatus

# 第9章 文件上传

## 9.1 文件上传

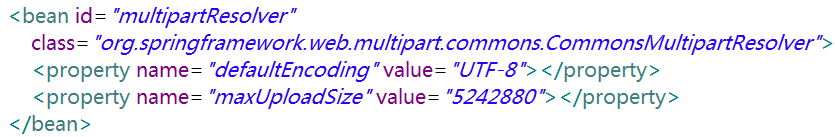
1. Spring MVC 为文件上传提供了直接的支持，这种支持是通过即插即用的 **MultipartResolver** 实现的。
2. Spring 用 **Jakarta Commons FileUpload** 技术实现了一个 MultipartResolver 实现类：**CommonsMultipartResolver**
3. Spring MVC 上下文中默认没有装配 MultipartResovler，因此默认情况下不能处理文件的上传工作，如果想使用 Spring 的文件上传功能，需现在上下文中配置 MultipartResolver



1. 配置 MultipartResolver

defaultEncoding: 必须和用户 JSP 的 pageEncoding 属性一致，以便正确解析表单的内

容,为了让 **CommonsMultipartResolver** 正确工作，必须先将 Jakarta Commons FileUpload 及 Jakarta Commons io 的类包添加到类路径下。



## 9.2 文件上传示例

1. 拷贝jar包

|  |
| --- |
| commons-fileupload-1.2.1.jar  commons-io-2.0.jar |
| 严重: Servlet /SpringMVC\_06\_FileUpload threw load() exception  java.lang.ClassNotFoundException: org.apache.commons.fileupload.FileItemFactory |

1. 配置文件上传解析器

|  |
| --- |
| <!-- 配置文件上传解析器  id必须是"multipartResolver",否则，会报错误：  java.lang.IllegalArgumentException: Expected MultipartHttpServletRequest: is a MultipartResolver configured?  -->  <bean id="multipartResolver"  class="org.springframework.web.multipart.commons.**CommonsMultipartResolver**">  <property name="defaultEncoding" value="UTF-8"></property>  <property name="maxUploadSize" value="1024000"></property>  </bean> |

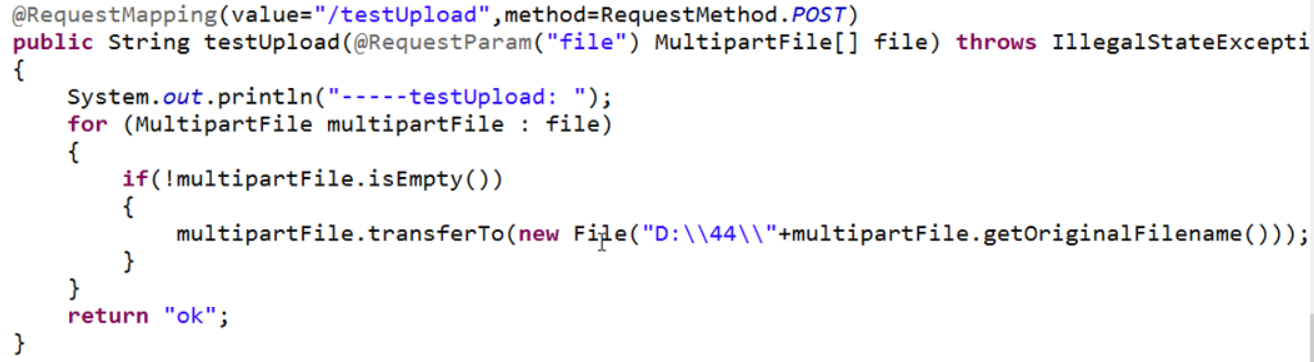
1. 上传页面

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>  **<form action="testUpload" method="post" enctype="multipart/form-data">**  **文件: <input type="file" name="file"/><br><br>**  **描述: <input type="text" name="desc"/><br><br>**  **<input type="submit" value="提交"/>**  **</form>**  </body>  </html> |

1. 控制器方法

|  |
| --- |
| package com.atguigu.springmvc.crud.handlers;    import java.io.IOException;  import java.io.InputStream;    import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RequestParam;  import org.springframework.web.multipart.MultipartFile;    @Controller  public class UploadHandler {    @RequestMapping(value="/testUpload",method=RequestMethod.**POST**)  public String testUpload(@RequestParam(value="desc",required=false) String desc, @RequestParam("file") **MultipartFile** multipartFile) throws IOException{  System.out.println("desc : "+desc);  System.out.println("OriginalFilename : "+multipartFile.**getOriginalFilename**());  InputStream inputStream = multipartFile.**getInputStream**();  System.out.println("inputStream.available() : "+inputStream.available());  System.out.println("inputStream : "+inputStream);  return "success"; //增加成功页面: /views/success.jsp  }  } |

## 9.3 思考多个文件上传？



# 第10章 拦截器

## 10.1 自定义拦截器概述

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** 
   1. **preHandle**()：这个方法在业务处理器处理请求之前被调用，在该方法中对用户请求 request 进行处理。**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完请求后，但是DispatcherServlet 向客户端返回响应前被调用**，在该方法中对用户请求request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。

## 10.2 实验代码(单个拦截器)

1. 自定义拦截器类

|  |
| --- |
| package com.atguigu.springmvc.interceptors;    import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    import org.springframework.web.servlet.HandlerInterceptor;  import org.springframework.web.servlet.ModelAndView;    public class **FirstHandlerInterceptor** implements **HandlerInterceptor** {    @Override  public void **afterCompletion**(HttpServletRequest arg0,  HttpServletResponse arg1, Object arg2, Exception arg3) throws Exception {  System.out.println(this.getClass().getName() + " - afterCompletion");  }    @Override  public void **postHandle**(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2, ModelAndView arg3) throws Exception {  System.out.println(this.getClass().getName() + " - postHandle");  }    @Override  public boolean **preHandle**(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2) throws Exception {  System.out.println(this.getClass().getName() + " - preHandle");  return true;  }  } |

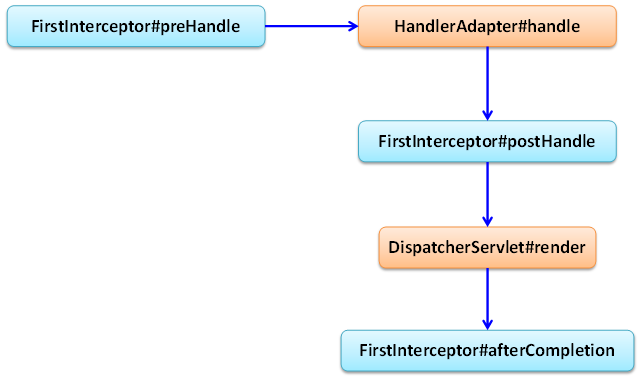
1. 配置拦截器

|  |
| --- |
| **<mvc:interceptors>**  **<!-- 声明自定义拦截器 -->**  **<bean id="firstHandlerInterceptor"**  **class="com.atguigu.springmvc.interceptors.FirstHandlerInterceptor"></bean>**  **</mvc:interceptors>** |

1. 断点调试拦截器执行流程

|  |
| --- |
|  |

1. 拦截器方法执行顺序（小总结）



## 10.3 实验代码(多个拦截器)

1. 自定义拦截器类(两个)

|  |  |
| --- | --- |
| package com.atguigu.springmvc.interceptors;    import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    import org.springframework.web.servlet.HandlerInterceptor;  import org.springframework.web.servlet.ModelAndView;    public class **FirstHandlerInterceptor** implements **HandlerInterceptor** {    @Override  public void afterCompletion(HttpServletRequest arg0,  HttpServletResponse arg1, Object arg2, Exception arg3)  throws Exception {  System.out.println(this.getClass().getName() + " - afterCompletion");  }    @Override  public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2, ModelAndView arg3) throws Exception {  System.out.println(this.getClass().getName() + " - postHandle");  }    @Override  public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2) throws Exception {  System.out.println(this.getClass().getName() + " - preHandle");  return **true**;  }    } | package com.atguigu.springmvc.interceptors;    import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    import org.springframework.web.servlet.HandlerInterceptor;  import org.springframework.web.servlet.ModelAndView;    public class **SecondHandlerInterceptor** implements **HandlerInterceptor** {    @Override  public void afterCompletion(HttpServletRequest arg0,  HttpServletResponse arg1, Object arg2, Exception arg3)  throws Exception {  System.out.println(this.getClass().getName() + " - afterCompletion");  }    @Override  public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2, ModelAndView arg3) throws Exception {  System.out.println(this.getClass().getName() + " - postHandle");  }    @Override  public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1,  Object arg2) throws Exception {  System.out.println(this.getClass().getName() + " - preHandle");  return **true**;  }    } |

1. 配置自定义拦截器

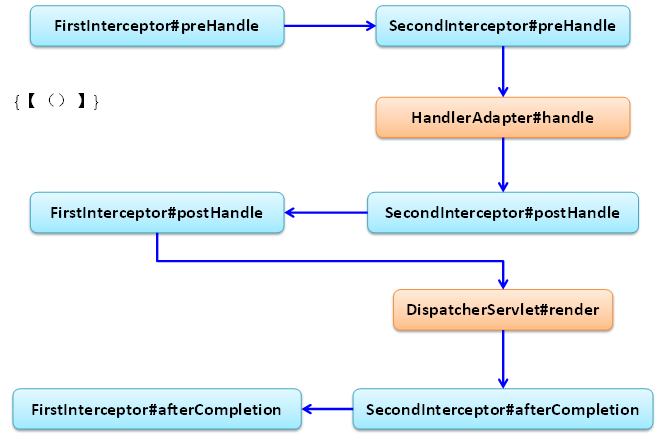
|  |
| --- |
| **<mvc:interceptors>**  **<!-- 声明自定义拦截器 -->**  **<bean id="firstHandlerInterceptor"**  **class="com.atguigu.springmvc.interceptors.FirstHandlerInterceptor"></bean>**  **<!-- 配置拦截器引用 -->**  **<mvc:interceptor>**  **<mvc:mapping path="/empList"/>**  **<!-- <mvc:exclude-mapping path="/empList"/> -->**  **<bean id="secondHandlerInterceptor"**  **class="com.atguigu.springmvc.interceptors.SecondHandlerInterceptor"></bean>**  **</mvc:interceptor>**  **</mvc:interceptors>** |
| 两个都是返回true :  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - preHandle  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*biz method\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - postHandle  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - postHandle  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - afterCompletion  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - afterCompletion |
| 两个都是返回false:  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle |
| true,false  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - preHandle  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - afterCompletion |
| false,true |
| com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle |

## 10.4 多个拦截方法的执行顺序

1. 关于执行顺序

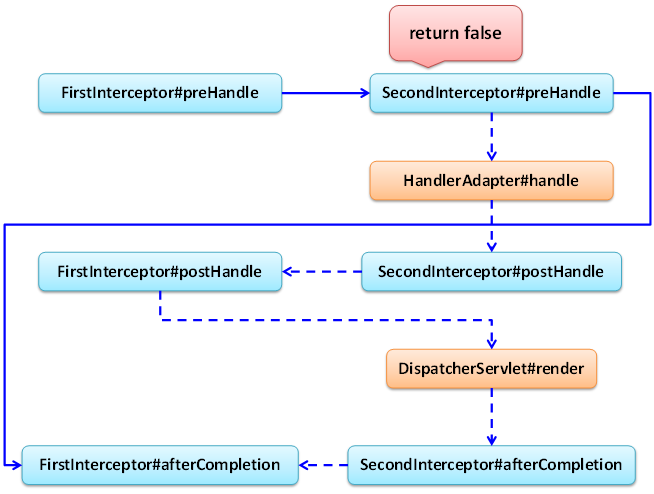
|  |
| --- |
| com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor – preHandle  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*biz method\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - postHandle  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - postHandle  com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - afterCompletion  com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - afterCompletion |

1. 执行顺序图解

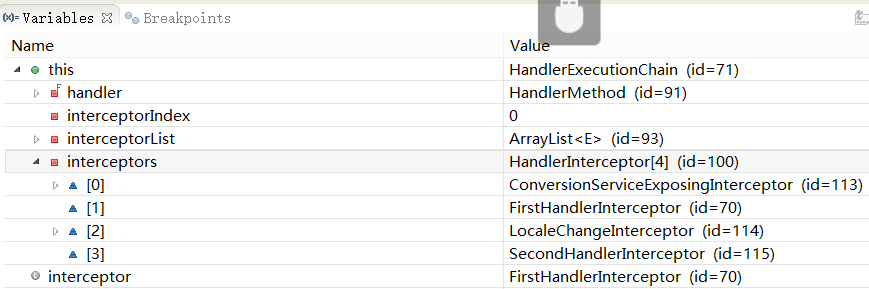


1. 从源代码的执行角度分析流程：

|  |
| --- |
| boolean **applyPreHandle**(HttpServletRequest request, HttpServletResponse response) throws Exception {  if (getInterceptors() != null) {  for (int i = 0; i < getInterceptors().length; i++) {  HandlerInterceptor interceptor = getInterceptors()[i];  if (!interceptor.preHandle(request, response, this.handler)) {  triggerAfterCompletion(request, response, null);  return false;  }  this.interceptorIndex = i;  }  }  return true;  } |
| void **applyPostHandle**(HttpServletRequest request, HttpServletResponse response, ModelAndView mv) throws Exception {  if (getInterceptors() == null) {  return;  }  for (int i = getInterceptors().length - 1; i >= 0; i--) {  HandlerInterceptor interceptor = getInterceptors()[i];  interceptor.postHandle(request, response, this.handler, mv);  }  } |
| void **triggerAfterCompletion**(HttpServletRequest request, HttpServletResponse response, Exception ex)  throws Exception {    if (getInterceptors() == null) {  return;  }  for (int i = this.interceptorIndex; i >= 0; i--) {  HandlerInterceptor interceptor = getInterceptors()[i];  try {  interceptor.afterCompletion(request, response, this.handler, ex);  }  catch (Throwable ex2) {  logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);  }  }  } |



1. 源码分析：分析interceptorIndex的值情况

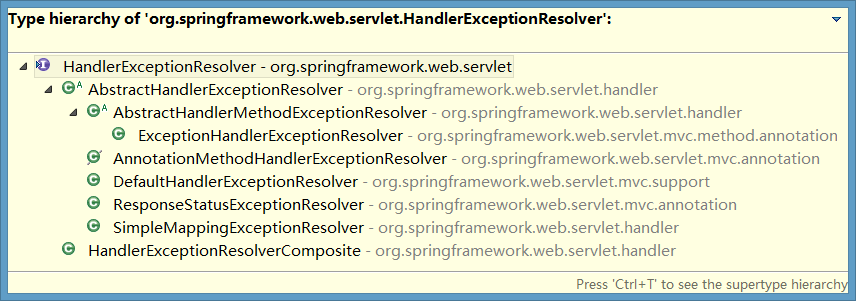


# 第11章 异常处理

java: throws throw try ..catch..finally

## 11.1异常处理概述

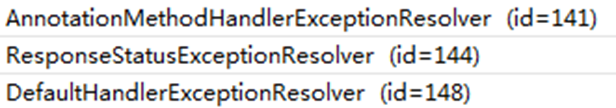
1. Spring MVC 通过 HandlerExceptionResolver 处理程序的异常，包括 Handler 映射、数据绑定以及目标方法执行时发生的异常。
2. SpringMVC 提供的 HandlerExceptionResolver 的实现类



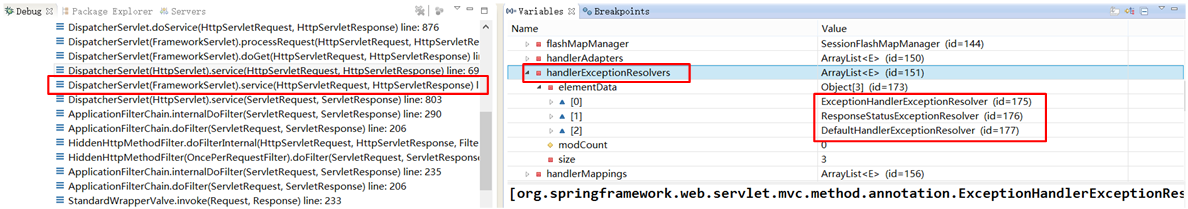
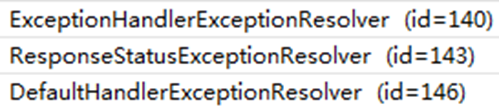
## 11.2 HandlerExceptionResolver

1）DispatcherServlet 默认装配的 HandlerExceptionResolver ：

2）没有使用 <mvc:annotation-driven/> 配置：



3）使用了 <mvc:annotation-driven/> 配置：



## 11.3 异常处理\_DefaultHandlerExceptionResolver

1）对一些特殊的异常进行处理，比如：

* + NoSuchRequestHandlingMethodException、
  + **HttpRequestMethodNotSupportedException**、
  + HttpMediaTypeNotSupportedException、
  + HttpMediaTypeNotAcceptableException等。

2）javadoc

|  |
| --- |
| [**org**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg)**.**[**springframework**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework)**.**[**web**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web)**.**[**servlet**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet)**.**[**mvc**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet.mvc)**.**[**support**](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet.mvc.support)**.DefaultHandlerExceptionResolver**  Default implementation of the [HandlerExceptionResolver](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet.mvc.support(DefaultHandlerExceptionResolver.class%E2%98%83DefaultHandlerExceptionResolver%E2%98%82org.springframework.web.servlet.HandlerExceptionResolver) interface that resolves standard Spring exceptions and translates them to corresponding HTTP status codes.  This exception resolver is enabled by default in the [org.springframework.web.servlet.DispatcherServlet](eclipse-javadoc:%E2%98%82=SpringMVC_08_ExceptionMapping/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_08_ExceptionMapping%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet.mvc.support(DefaultHandlerExceptionResolver.class%E2%98%83DefaultHandlerExceptionResolver%E2%98%82org.springframework.web.servlet.DispatcherServlet). |

### 11.3.1 实验代码

1. 增加页面链接：GET请求

|  |
| --- |
| <a href="testDefaultHandlerExceptionResolver">testDefaultHandlerExceptionResolver</a> |

增加处理器方法

|  |
| --- |
| //@RequestMapping(value="/testDefaultHandlerExceptionResolver")  @RequestMapping(value="/testDefaultHandlerExceptionResolver",method=**RequestMethod.POST**) //不支持GET请求  public String testDefaultHandlerExceptionResolver(){  System.out.println("testDefaultHandlerExceptionResolver...");  return "success";  } |

1. 出现异常错误

|  |
| --- |
|  |

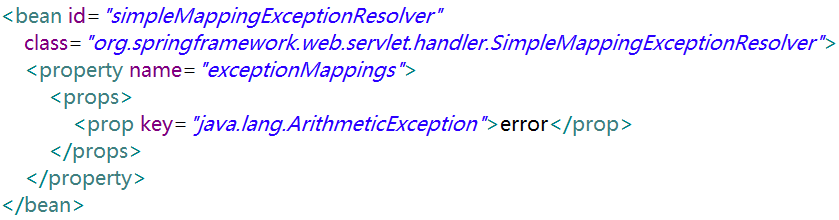
1. 出现异常交给DefaultHandlerExceptionResolver处理

|  |
| --- |
| @Override  protected ModelAndView doResolveException(  HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {    try {  if (ex instanceof NoSuchRequestHandlingMethodException) {  return handleNoSuchRequestHandlingMethod((NoSuchRequestHandlingMethodException) ex, request, response,  handler);  }  **else if (ex instanceof HttpRequestMethodNotSupportedException) {**  **return handleHttpRequestMethodNotSupported((HttpRequestMethodNotSupportedException) ex, request,**  **response, handler);**  **}**  else if (ex instanceof HttpMediaTypeNotSupportedException) {  return handleHttpMediaTypeNotSupported((HttpMediaTypeNotSupportedException) ex, request, response,  handler);  }  else if (ex instanceof HttpMediaTypeNotAcceptableException) {  return handleHttpMediaTypeNotAcceptable((HttpMediaTypeNotAcceptableException) ex, request, response,  handler);  }  else if (ex instanceof MissingServletRequestParameterException) {  return handleMissingServletRequestParameter((MissingServletRequestParameterException) ex, request,  response, handler);  }  else if (ex instanceof ServletRequestBindingException) {  return handleServletRequestBindingException((ServletRequestBindingException) ex, request, response,  handler);  }  else if (ex instanceof ConversionNotSupportedException) {  return handleConversionNotSupported((ConversionNotSupportedException) ex, request, response, handler);  }  else if (ex instanceof TypeMismatchException) {  return handleTypeMismatch((TypeMismatchException) ex, request, response, handler);  }  else if (ex instanceof HttpMessageNotReadableException) {  return handleHttpMessageNotReadable((HttpMessageNotReadableException) ex, request, response, handler);  }  else if (ex instanceof HttpMessageNotWritableException) {  return handleHttpMessageNotWritable((HttpMessageNotWritableException) ex, request, response, handler);  }  else if (ex instanceof MethodArgumentNotValidException) {  return handleMethodArgumentNotValidException((MethodArgumentNotValidException) ex, request, response, handler);  }  else if (ex instanceof MissingServletRequestPartException) {  return handleMissingServletRequestPartException((MissingServletRequestPartException) ex, request, response, handler);  }  else if (ex instanceof BindException) {  return handleBindException((BindException) ex, request, response, handler);  }  else if (ex instanceof NoHandlerFoundException) {  return handleNoHandlerFoundException((NoHandlerFoundException) ex, request, response, handler);  }  }  catch (Exception handlerException) {  logger.warn("Handling of [" + ex.getClass().getName() + "] resulted in Exception", handlerException);  }  return null;  } |

## 11.4 异常处理\_SimpleMappingExceptionResolver

1）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



### 11.4.1实验代码

1. 增加页面链接

|  |
| --- |
| <a href="testSimpleMappingExceptionResolver?i=1">testSimpleMappingExceptionResolver</a> |

1. 增加控制器方法

|  |
| --- |
| @RequestMapping("/testSimpleMappingExceptionResolver")  public String testSimpleMappingExceptionResolver(@RequestParam("i") int i){  System.out.println("testSimpleMappingExceptionResolver...");  String[] s = new String[10];  System.out.println(s[i]);  return "success";  } |

1. 出现异常情况：参数i的值大于10



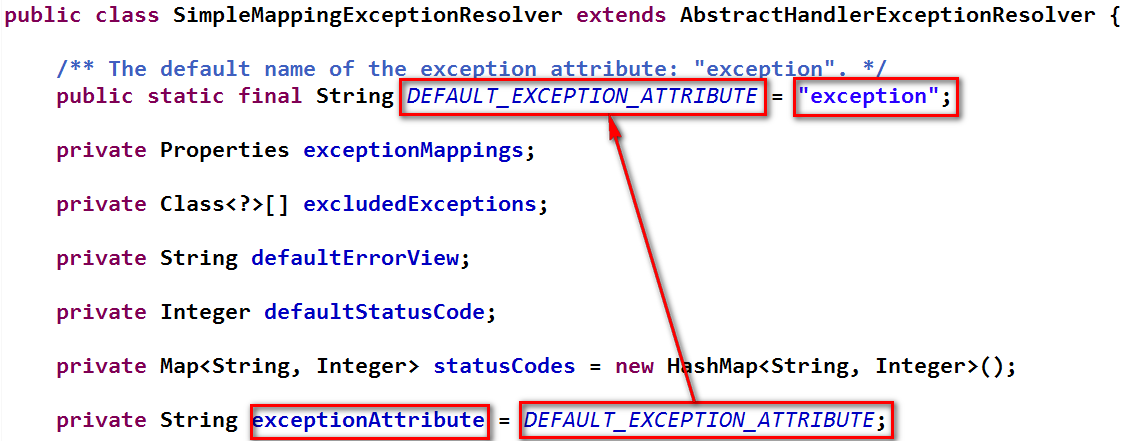
1. 配置异常解析器:自动将异常对象信息，存放到request范围内

|  |
| --- |
| <!-- 配置SimpleMappingExceptionResolver异常解析器 -->  <bean id="simpleMappingExceptionResolver"  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <!-- exceptionAttribute默认值(通过ModelAndView传递给页面)：  exception -> ${requestScope.exception}  public static final String DEFAULT\_EXCEPTION\_ATTRIBUTE = "exception";  -->  <property name="**exceptionAttribute**" value="exception"></property>  <property name="**exceptionMappings**">  <props>  <prop key="java.lang.ArrayIndexOutOfBoundsException">**error**</prop>  </props>  </property>  </bean> |

|  |
| --- |
| **error**.jsp |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  "<http://www.w3.org/TR/html4/loose.dtd>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Insert title here</title>  </head>  <body>  <h3>Error Page</h3>  ${exception }  **${requestScope.exception }**  </body>  </html> |
|  |

1. 源码分析

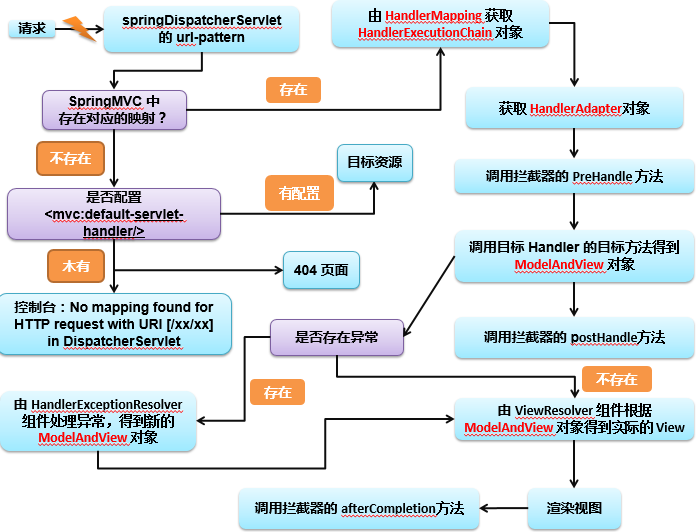
|  |
| --- |
| SimpleMappingExceptionResolver L187 L339 |



|  |
| --- |
| @Override  protected ModelAndView doResolveException(HttpServletRequest request,  HttpServletResponse response,Object handler, Exception ex) {    // Expose ModelAndView for chosen error view.  String viewName = determineViewName(ex, request);  if (viewName != null) {  // Apply HTTP status code for error views, if specified.  // Only apply it if we're processing a top-level request.  Integer statusCode = determineStatusCode(request, viewName);  if (statusCode != null) {  applyStatusCodeIfPossible(request, response, statusCode);  }  return **getModelAndView(viewName, ex, request)**;  }else {  return null;  }  } |
| /\*\*  \* **Return a ModelAndView for the given view name and exception.**  \* <p>The default implementation adds the specified exception attribute.  \* Can be overridden in subclasses.  \* @param viewName the name of the error view  \* @param ex the exception that got thrown during handler execution  \* @return the ModelAndView instance  \* @see #setExceptionAttribute  \*/  protected ModelAndView getModelAndView(String viewName, Exception ex) {  ModelAndView mv = new ModelAndView(viewName);  if (this.exceptionAttribute != null) {  if (logger.isDebugEnabled()) {  logger.debug("Exposing Exception as model attribute '" + this.exceptionAttribute + "'");  }  **mv.addObject(this.exceptionAttribute, ex);**  }  return mv;  } |

# 第12章 运行流程图解

## 12.1 流程图



## 12.2 Spring工作流程描述

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：
     + 再判断是否配置了mvc:default-servlet-handler：
     + 如果没配置，则控制台报映射查找不到，客户端展示404错误
     + 如果有配置，则执行目标资源（一般为静态资源，如：JS,CSS,HTML）
  2. 存在：
     + 执行下面流程

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后，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. 数据根式化：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

## 12.3 源码解析

### 12.3.1 搭建环境

1. 拷贝jar包

|  |
| --- |
| spring-aop-4.0.0.RELEASE.jar  spring-beans-4.0.0.RELEASE.jar  spring-context-4.0.0.RELEASE.jar  spring-core-4.0.0.RELEASE.jar  spring-expression-4.0.0.RELEASE.jar  commons-logging-1.1.3.jar  **spring-web-4.0.0.RELEASE.jar**  **spring-webmvc-4.0.0.RELEASE.jar** |

1. 配置文件web.xml

|  |
| --- |
| <servlet>  <servlet-name>springDispatcherServlet</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>springDispatcherServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |

1. 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"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/mvc <http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd>  <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-4.0.xsd>">    <!-- 设置扫描组件的包 -->  <context:component-scan base-package="com.atguigu.springmvc"/>    <!-- 配置视图解析器 -->  <bean id="internalResourceViewResolver"  class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/views/"/>  <property name="suffix" value=".jsp"/>  </bean>    </beans> |

### 12.3.2 完成HelloWorld

1. 页面链接

|  |
| --- |
| <a href="helloworld">Hello World</a> |

1. 控制器方法

|  |
| --- |
| package com.atguigu.springmvc.handler;    import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;    @Controller  public class HelloWorldHandler {  @RequestMapping("/helloworld")  public String testHello(){  System.out.println("Hello,SpringMVC...");  return "success";  }  } |

1. 成功页面：/views/success.jsp

|  |
| --- |
| <h3>Success Page</h3> |

### 12.3.3 Debug实验

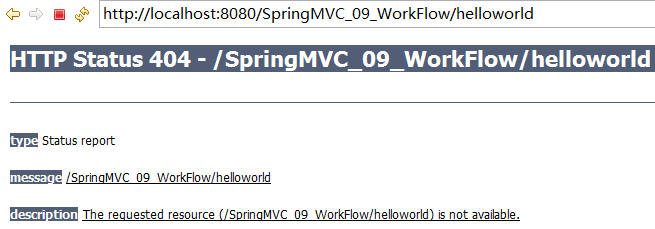
1. 正常流程，运行出结果
2. 没有配置<mvc:default-servlet-handler/>，测试，直接报404
3. <http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2>

|  |
| --- |
| 四月 20, 2016 11:53:19 上午 org.springframework.web.servlet.PageNotFound noHandlerFound  警告: No mapping found for HTTP request with URI [/SpringMVC\_09\_WorkFlow/helloworld2] in DispatcherServlet with name 'springDispatcherServlet' |

1. <http://localhost:8080/SpringMVC_09_WorkFlow/test.html>

|  |
| --- |
| 四月 20, 2016 11:54:16 上午 org.springframework.web.servlet.PageNotFound noHandlerFound  警告: No mapping found for HTTP request with URI [/SpringMVC\_09\_WorkFlow/test.html] in DispatcherServlet with name 'springDispatcherServlet' |

1. 配置<mvc:default-servlet-handler/>，测试，会去查找目标资源
2. 测试，依然发生错误，这时，需要配置：<mvc:annotation-driven/>，否则，映射解析不好使。

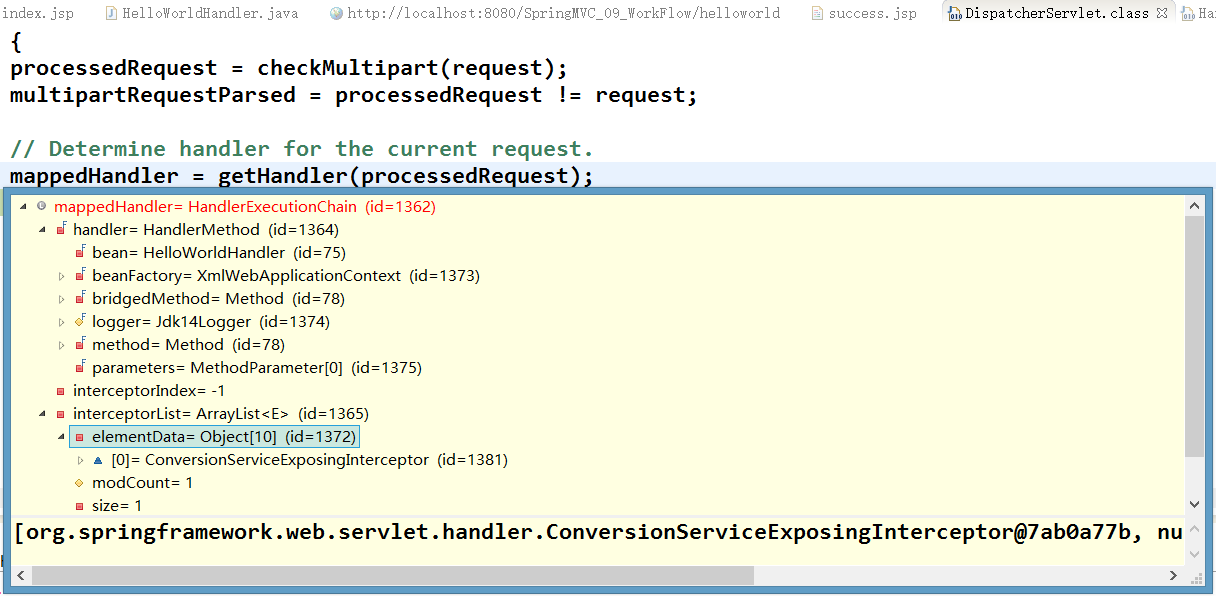


### 12.3.4 Debug流程分析

1. HandlerExecutionChain mappedHandler；包含了拦截器和处理器方法；

DispatcherServlet L902 916

|  |
| --- |
| [**org**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg)**.**[**springframework**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework)**.**[**web**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web)**.**[**servlet**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet)**.HandlerExecutionChain**  Handler execution chain, consisting of handler object and any handler interceptors. Returned by  HandlerMapping's [HandlerMapping.getHandler](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerExecutionChain.class%E2%98%83HandlerExecutionChain%E2%98%82HandlerMapping%E2%98%82getHandler) method. |

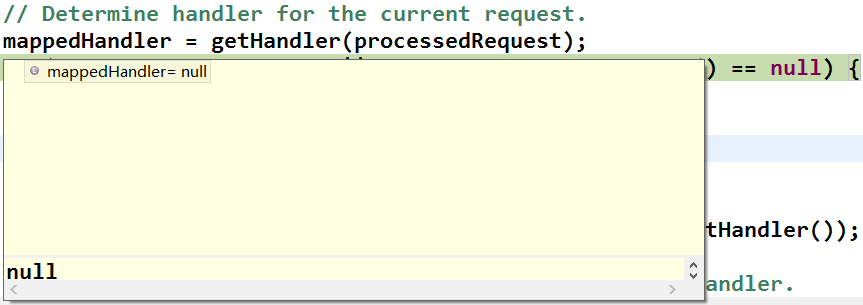


1. HandlerMapping

|  |
| --- |
| [**org**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg)**.**[**springframework**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework)**.**[**web**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web)**.**[**servlet**](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet)**.HandlerMapping**  **Interface to be implemented by objects that define a mapping between requests and handler objects.**  This class can be implemented by application developers, although this is not necessary, as [org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerMapping.class%E2%98%83HandlerMapping%E2%98%82org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping) and [org.springframework.web.servlet.handler.SimpleUrlHandlerMapping](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerMapping.class%E2%98%83HandlerMapping%E2%98%82org.springframework.web.servlet.handler.SimpleUrlHandlerMapping) are included in the framework. The former is the default if no HandlerMapping bean is registered in the application context.  HandlerMapping implementations can support mapped interceptors but do not have to. A handler will always be wrapped in a [HandlerExecutionChain](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerMapping.class%E2%98%83HandlerMapping%E2%98%82HandlerExecutionChain) instance, optionally accompanied by some [HandlerInterceptor](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerMapping.class%E2%98%83HandlerMapping%E2%98%82HandlerInterceptor) instances. The DispatcherServlet will first call each HandlerInterceptor's preHandle method in the given order, finally invoking the handler itself if all preHandle methods have returned true.  The ability to parameterize this mapping is a powerful and unusual capability of this MVC framework. For example, it is possible to write a custom mapping based on session state, cookie state or many other variables. No other MVC framework seems to be equally flexible.  Note: Implementations can implement the [org.springframework.core.Ordered](eclipse-javadoc:%E2%98%82=SpringMVC_09_WorkFlow/E:%5C/atguigu%5C/workspacessm%5C/SpringMVC_09_WorkFlow%5C/WebContent%5C/WEB-INF%5C/lib%5C/spring-webmvc-4.0.0.RELEASE.jar%3Corg.springframework.web.servlet(HandlerMapping.class%E2%98%83HandlerMapping%E2%98%82org.springframework.core.Ordered) interface to be able to specify a sorting order and thus a priority for getting applied by DispatcherServlet. Non-Ordered instances get treated as lowest priority. |

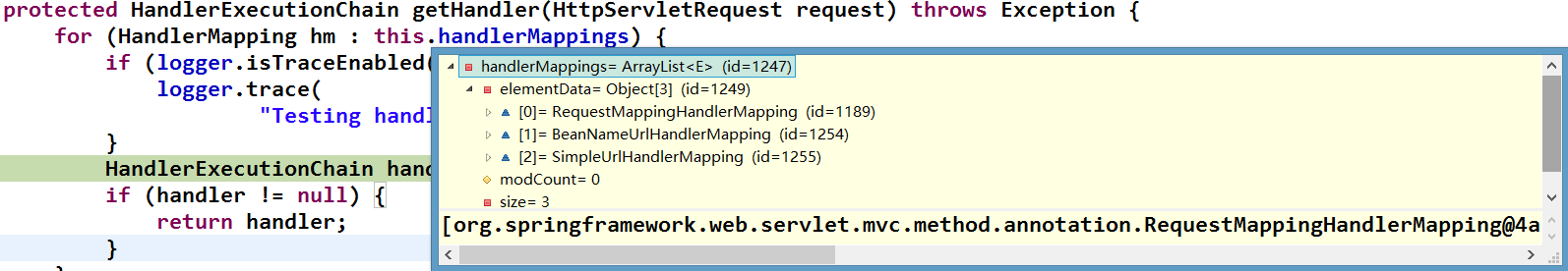
1. 没有配置<mvc:default-servlet-handler/>，<mvc:annotation-driven/>,发送一个不存在资源的请求路径，mappedHandler为null

* [http://localhost:8080/SpringMVC\_09\_WorkFlow/helloworld**2**](http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2)

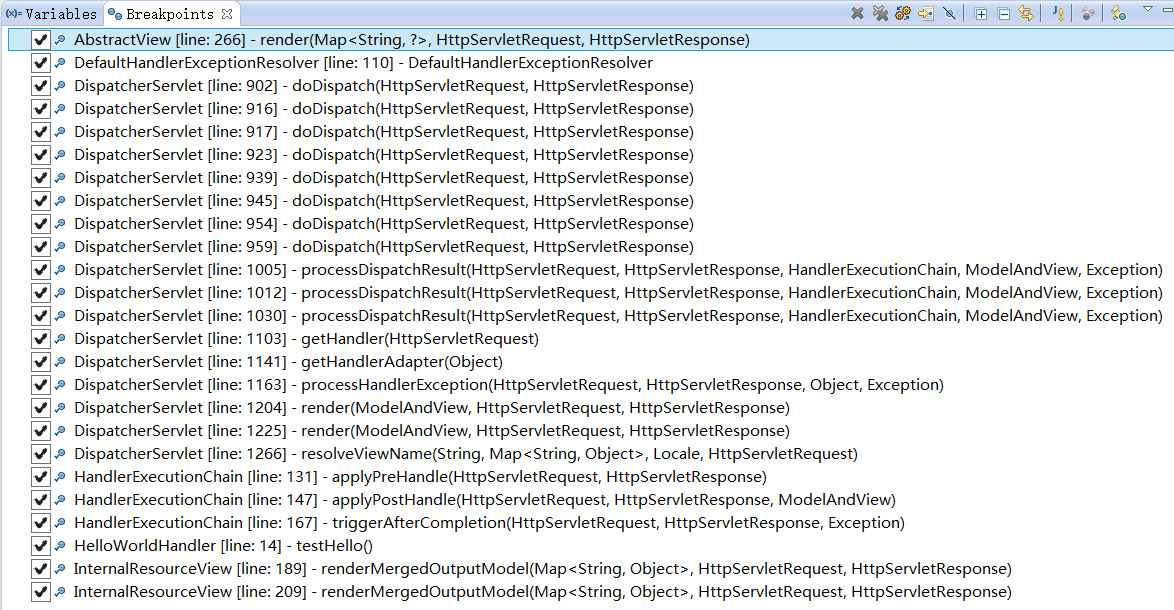


1. 配置<mvc:default-servlet-handler/>，<mvc:annotation-driven/>,发送一个不存在资源的请求路径

* [http://localhost:8080/SpringMVC\_09\_WorkFlow/helloworld**2**](http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2)
* mappedHandler不为null,原因是当循环simpleUrlHandlerMapping时，当做静态资源处理



### 12.3.5 断点



# 第13章 Spring整合SpringMVC

## 13.1 Spring 与SpringMVC的整合问题：

1. 需要进行 Spring 整合 SpringMVC 吗 ?
2. 还是否需要再加入 Spring 的 IOC 容器 ?
3. 是否需要在web.xml 文件中配置启动 Spring IOC 容器的 ContextLoaderListener ?

需要: 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(而不是放在 SpringMVC 的配置文件中).

实际上放入 Spring 配置文件对应的 IOC 容器中的还有 Service 和 Dao.

不需要: 都放在 SpringMVC 的配置文件中. 也可以分多个 Spring 的配置文件, 然后使

用 import 节点导入其他的配置文件

## 13.2 Spring整合SpringMVC\_解决方案配置监听器

1. 监听器配置

|  |
| --- |
| <!-- 配置启动 Spring IOC 容器的 Listener -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:beans.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

1. 创建Spring的bean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> <http://www.springframework.org/schema/context/spring-context-4.0.xsd>">    **<!-- 设置扫描组件的包 -->**  **<context:component-scan base-package="com.atguigu.springmvc"/>**    <!-- 配置数据源, 整合其他框架, 事务等. -->    </beans> |

1. springmvc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"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/mvc <http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd>  <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-4.0.xsd>">    **<!-- 设置扫描组件的包 -->**  **<context:component-scan base-package="com.atguigu.springmvc"/>**  <!-- 配置视图解析器 -->  <bean id="internalResourceViewResolver"  class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/views/"/>  <property name="suffix" value=".jsp"/>  </bean>    <mvc:default-servlet-handler/>  <mvc:annotation-driven/>  </beans> |

在HelloWorldHandler、UserService类中增加构造方法，启动服务器，查看构造器执行情况。

问题: 若 Spring 的 IOC 容器和 SpringMVC 的 IOC 容器扫描的包有重合的部分, 就会导致有的 bean 会被创建 2 次.

**解决:**

使 Spring 的 IOC 容器扫描的包和 SpringMVC 的 IOC 容器扫描的包没有重合的部分.

使用 exclude-filter 和 include-filter 子节点来规定只能扫描的注解

|  |
| --- |
| **springmvc.xml** |
| <context:component-scan base-package="com.atguigu.springmvc" use-default-filters="false">  <**context:include-filter** type="annotation"  expression="org.springframework.stereotype.Controller"/>  <**context:include-filter** type="annotation"  expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan> |

|  |
| --- |
| **beans.xml** |
| <context:component-scan base-package="com.atguigu.springmvc">  <**context:exclude-filter** type="annotation"  expression="org.springframework.stereotype.Controller"/>  <**context:exclude-filter** type="annotation"  expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan>  <!-- 配置数据源, 整合其他框架, 事务等. --> |

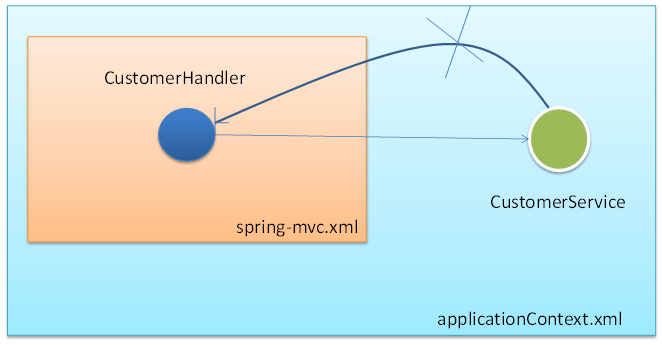
## 13.3 SpringIOC 容器和 SpringMVC IOC 容器的关系

SpringMVC 的 IOC 容器中的 bean 可以来引用 Spring IOC 容器中的 bean.

返回来呢 ? 反之则不行. Spring IOC 容器中的 bean 却不能来引用 SpringMVC IOC 容器中的 bean

1. 在 Spring MVC 配置文件中引用业务层的 Bean
2. 多个 Spring IOC 容器之间可以设置为父子关系，以实现良好的解耦。
3. Spring MVC WEB 层容器可作为 “业务层” Spring 容器的子容器：

即 WEB 层容器可以引用业务层容器的 Bean，而业务层容器却访问不到 WEB 层容器的 Bean



## 13.4 SpringMVC对比Struts2

1） Spring MVC 的入口是 Servlet, 而 Struts2 是 FilterSpring MVC 会稍微比 Struts2 快些.

2） Spring MVC 是基于方法设计, 而 Sturts2 是基于类,

每次发一次请求都会实例一个 Action.

1. Spring MVC 使用更加简洁, 开发效率Spring MVC确实比 struts2 高: 支持 JSR303, 处 理ajax 的请求更方便
2. Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些.