SpringMVC 框架

尚硅谷 java 研究院

版本：V 1.0

第 1 章 SpringMVC 概述

1.1 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。

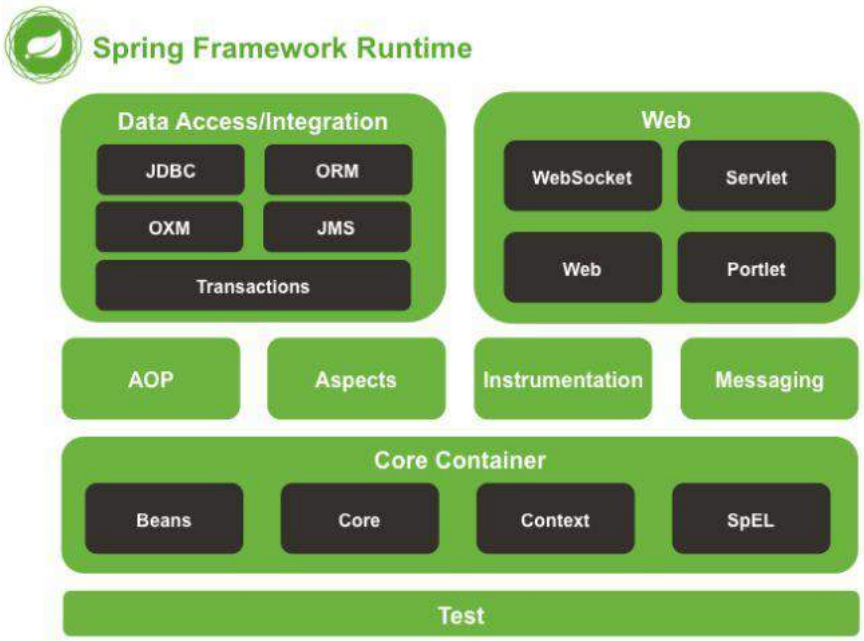
4）支持 REST 风格的 URL 请求。

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。

1.2 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
```

2） 在 web.xml 中配置 DispatcherServlet

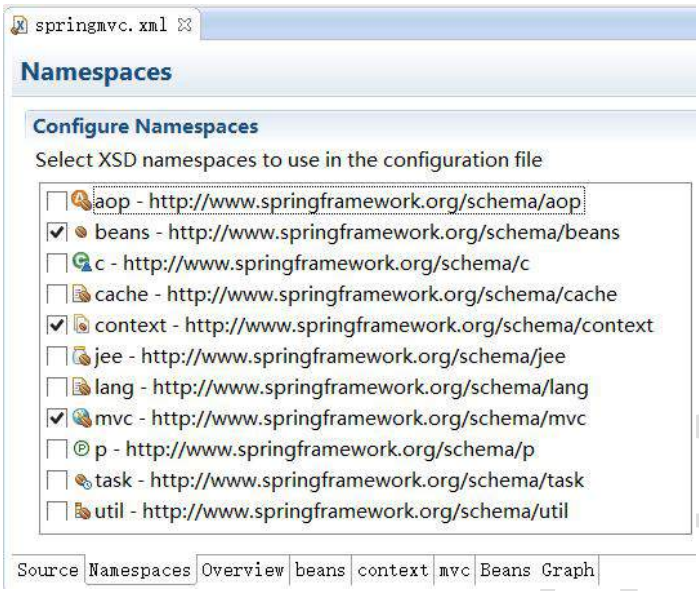
```
<!-- 配置 SpringMVC 核心控制器： -->
<servlet>
    <servlet-name>springDispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 配置 DispatcherServlet 的初始化参数：设置文件的路径和文件名称 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <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**

3） 加入 Spring MVC 的配置文件：springmvc.xml

① 增加名称空间



② 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>
```

4） 需要创建一个入口页面，index.jsp

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

5） 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
  }
}
```

6） 编写视图

/WEB-INF/views/success.jsp

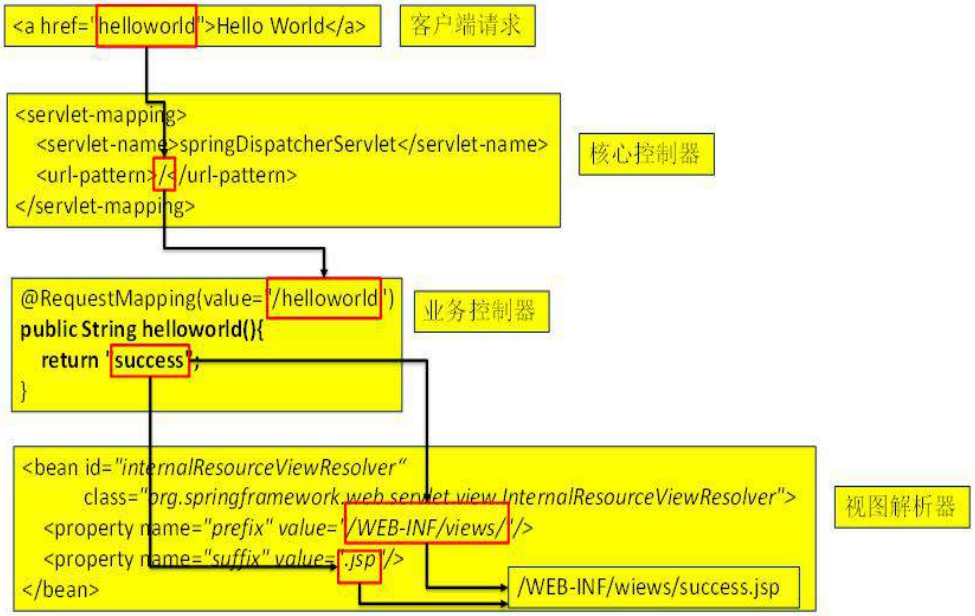
```
<h4>Sucess Page</h4>
```

7） 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：



2） 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";
}
```

3） 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>
```

4） 处理请求方式有哪几种

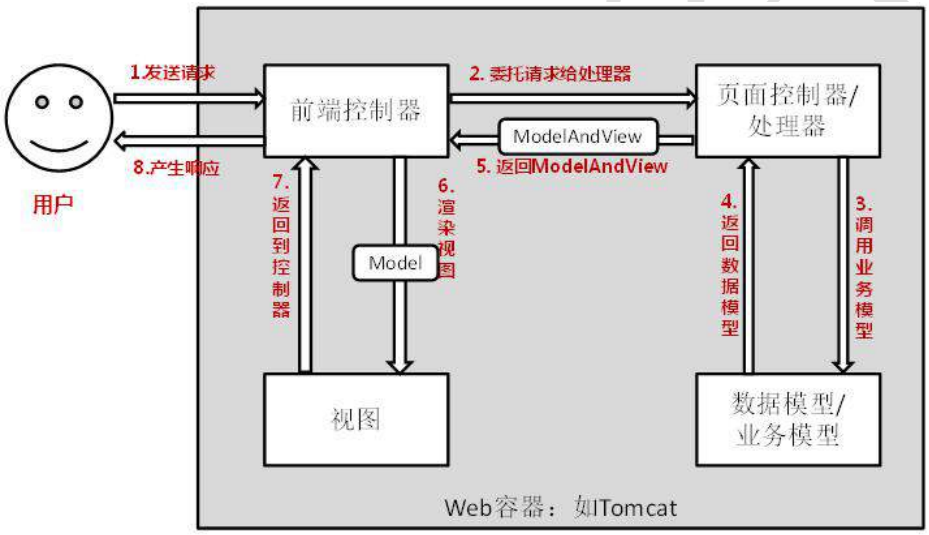
```
public enum RequestMethod {
```

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

5） @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";
    }
}
```

2） 以 get 方式请求

```
<a href="springmvc/testMethord">testMethord</a>
```

发生请求错误

3） 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>
```

2） 测试:使用火狐或 Chrom 浏览器 debug 测试

① 测试有参数情况(不正确)：

- testParamsAndHeaders

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

- testParamsAndHeaders

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']]

- testParamsAndHeaders

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

② 测试有参数情况(正确):

- testParamsAndHeaders

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

2） @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/createUseraa、/user/createUserbb 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";
}
```

2） 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
@RequestMapping(value="/testPathVariable/{id}")
public String testPathVariable(@PathVariable("id") Integer id){
    System.out.println("testPathVariable...id="+id);
    return "success";
}
```

2） 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

3.2 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

① **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。

② 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。

③ 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”。

④ 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

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

/order HTTP **PUT**：更新 order

/order HTTP **POST**：新增 order

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”

```
public class HiddenHttpMethodFilter extends OncePerRequestFilter {

    /** Default method parameter: {@code _method} */
    public static final String DEFAULT_METHOD_PARAM = "_method";
```

2） hiddenHttpMethodFilter 的处理过程

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

    String paramValue = request.getParameter(this.methodParam);
    if ("POST".equals(request.getMethod()) && StringUtils.hasLength(paramValue)) {
        String method = paramValue.toUpperCase(Locale.ENGLISH);
        HttpServletRequest wrapper = new HttpMethodRequestWrapper(request, method);
        filterChain.doFilter(wrapper, response);
    }
    else {
        filterChain.doFilter(request, response);
    }
}


private static class HttpMethodRequestWrapper extends HttpServletRequestWrapper {

    private final String method;

    public HttpMethodRequestWrapper(HttpServletRequest request, String method) {
        super(request);
        this.method = method;
    }

    @Override
    public String getMethod() {
        return this.method;
    }
}
```

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>
```

2） 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";
}
```

3） 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

4.1 请求处理方法签名

1） Spring MVC 通过分析处理方法的签名，HTTP 请求信息绑定到处理方法的相应人参中。
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";
}
```

2） 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**

```
@RequestMapping("/handle7")
public String handle7(@RequestHeader("Accept-Encoding") String encoding,
      @RequestHeader("Keep-Alive") long keppAlieve){
   return "success";
}
```

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

```
@RequestMapping("/handle6")
public String handle6(@CookieValue(value="sessionId", required=false) String sessionId,
        @RequestParam("age") int age){
    return "success";
}
```

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";
}
```

2） 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>
```

username: zhangsan
password: ●●●
email: zhangsan@atguigu.com
age: 22
city: 松原
province: 吉林
Submit

2） 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
}
```

3） 执行结果：

```
Console  Servers
Tomcat v6.0 Server at localhost [Apache Tomcat] C:\Java\jdk1.7.0_75\bin\javaw.exe (2016年4月12日 上午11:49:34)
testPojo: User [id=null, username=zhangsan, password=, email=zhangsan@atguigu.com, age=22]
```

4） 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>
```

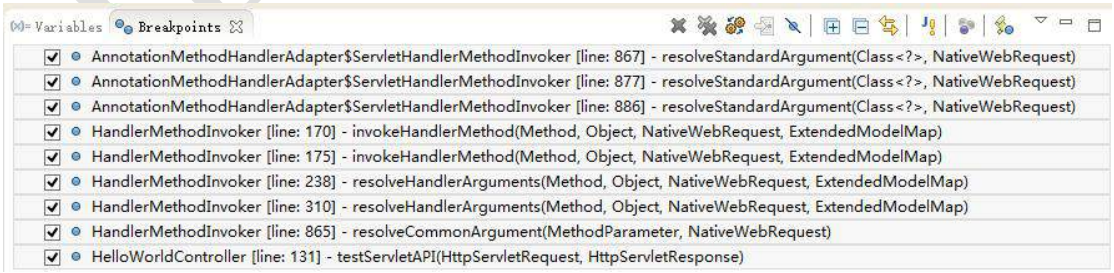
```
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <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

1) HttpServletRequest
2) HttpServletResponse
3) HttpSession
4) **java.security.Principal**
5) **Locale**
6) **InputStream**
7) **OutputStream**
8) **Reader**
9) **Writer**

2） 源码参考：AnnotationMethodHandlerAdapter L866



3）

```
@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

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 及 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。

3） **@SessionAttributes**: 将模型中的某个属性暂存到 HttpSession 中，以便多个请求之间可以共享这个属性

4） **@ModelAttribute**: 方法入参标注该注解后，入参的对象就会放到数据模型中

5.2 处理模型数据之 ModelAndView

5.2.1 ModelAndView 介绍

1） 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。

2）添加模型数据:

MoelAndView addObject(String attributeName, Object attributeValue)

ModelAndView addAllObject(Map<String, ?> modelMap)

3）设置视图:

void setView(View view)

void setViewName(String viewName)

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;
}
```

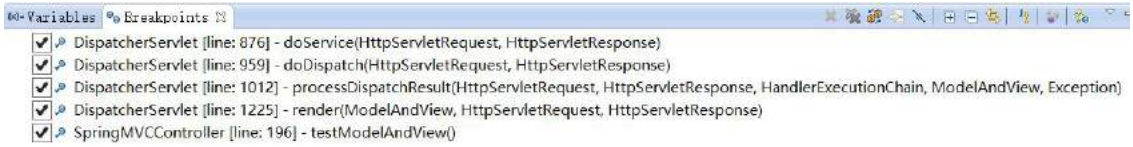
2） 增加页面链接

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

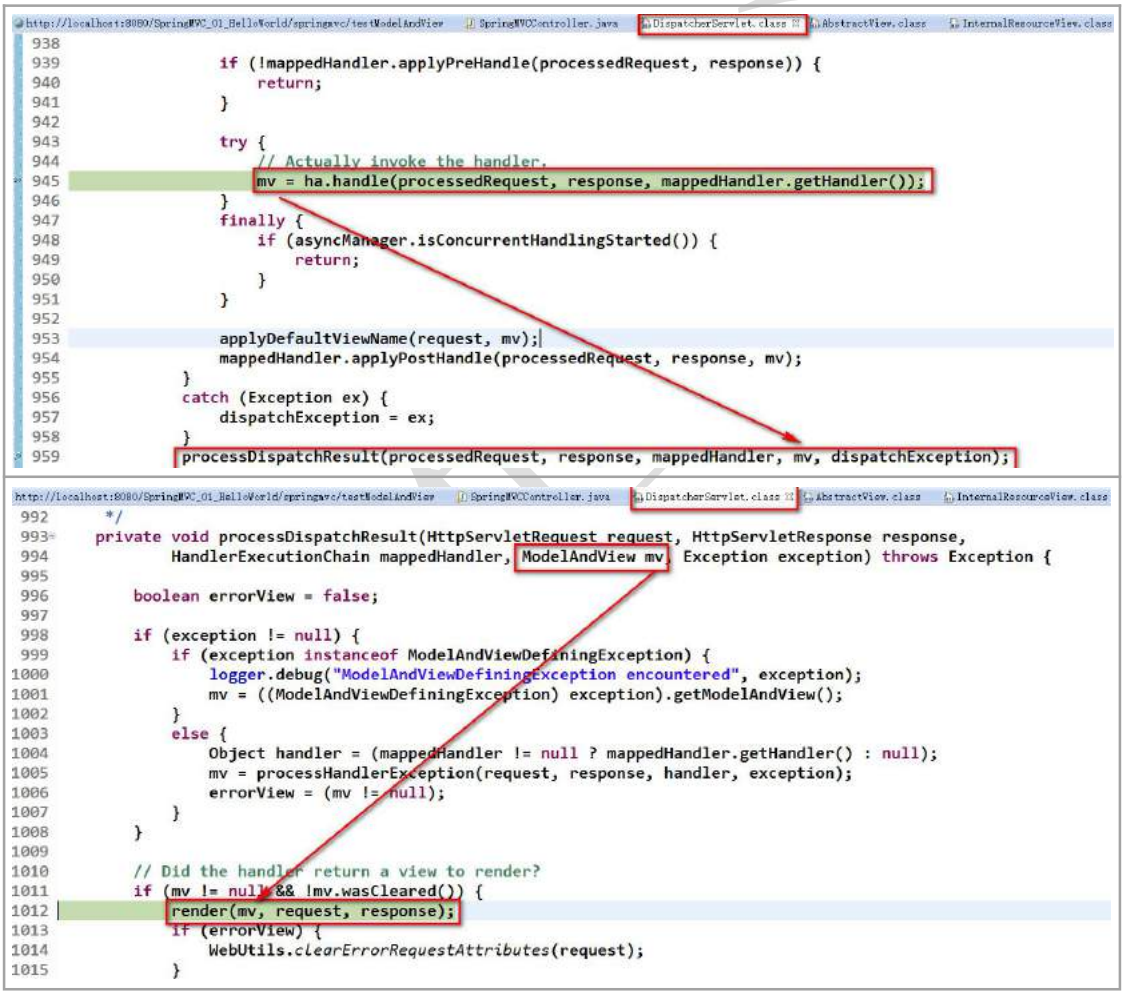
3） 增加成功页面，显示数据

```
time: ${requestScope.time }
```

4） 断点调试



5.2.2 源码解析



```
938
939                 if (!mappedHandler.applyPreHandle(processedRequest, response)) {
940                     return;
941                 }
942
943                 try {
944                     // Actually invoke the handler.
945                     mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
946                 }
947                 finally {
948                     if (asyncManager.isConcurrentHandlingStarted()) {
949                         return;
950                     }
951                 }
952
953                 applyDefaultViewName(request, mv);
954                 mappedHandler.applyPostHandle(processedRequest, response, mv);
955             }
956             catch (Exception ex) {
957                 dispatchException = ex;
958             }
959             processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
```

```
992      */
993     private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
994             HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws Exception {
995
996         boolean errorView = false;
997
998         if (exception != null) {
999             if (exception instanceof ModelAndViewDefiningException) {
1000                 logger.debug("ModelAndViewDefiningException encountered", exception);
1001                 mv = ((ModelAndViewDefiningException) exception).getModelAndView();
1002             }
1003             else {
1004                 Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
1005                 mv = processHandlerException(request, response, handler, exception);
1006                 errorView = (mv != null);
1007             }
1008         }
1009
1010         // Did the handler return a view to render?
1011         if (mv != null && !mv.wasCleared()) {
1012             render(mv, request, response);
1013             if (errorView) {
1014                 WebUtils.clearErrorRequestAttributes(request);
1015             }
```

```
1212            // No need to lookup: the ModelAndView object contains the actual View object.
1213            view = mv.getView();
1214            if (view == null) {
1215                throw new ServletException("ModelAndView [" + mv + "] neither contains a view name nor a " +
1216                        "View object in servlet with name '" + getServletName() + "'");
1217            }
1218        }
1219
1220        // Delegate to the View object for rendering.
1221        if (logger.isDebugEnabled()) {
1222            logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" + getServletName() + "'");
1223        }
1224        try {
1225            view.render(mv.getModelInternal(), request, response);
1226        }
1227        catch (Exception ex) {
1228            if (logger.isDebugEnabled()) {
1229                logger.debug("Error rendering view [" + view + "] in DispatcherServlet with name '"
1230                        + getServletName() + "'", ex);
1231            }
1232            throw ex;
1233        }
1234    }
```

```
256     @Override
257     public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
258         if (logger.isTraceEnabled()) {
259             logger.trace("Rendering view with name '" + this.beanName + "' with model " + model +
260                 " and static attributes " + this.staticAttributes);
261         }
262
263         Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);
264
265         prepareResponse(request, response);
266         renderMergedOutputModel(mergedModel, request, response);
267     }
```

```
170      * This includes setting the model as request attributes.
171      */
172     @Override
173     protected void renderMergedOutputModel(
174             Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
175
176         // Determine which request handle to expose to the RequestDispatcher.
177         HttpServletRequest requestToExpose = getRequestToExpose(request);
178
179         // Expose the model object as request attributes.
180         exposeModelAsRequestAttributes(model, requestToExpose);
181
182         // Expose helpers as request attributes, if any.
183         exposeHelpers(requestToExpose);
184
185         // Determine the path for the request dispatcher.
186         String dispatcherPath = prepareForRendering(requestToExpose, response);
187
188         // Obtain a RequestDispatcher for the target resource (typically a JSP).
189         RequestDispatcher rd = getRequestDispatcher(requestToExpose, dispatcherPath);
190         if (rd == null) {
191             throw new ServletException("Could not get RequestDispatcher for [" + getUrl() +
192                     "]: Check that the corresponding file exists within your web application archive!");
193         }
```

```
368     */
369     protected void exposeModelAsRequestAttributes(Map<String, Object> model, HttpServletRequest request) throws Exception {
370         for (Map.Entry<String, Object> entry : model.entrySet()) {
371             String modelName = entry.getKey();
372             Object modelValue = entry.getValue();
373             if (modelValue != null) {
374                 request.setAttribute(modelName, modelValue);
375                 if (logger.isDebugEnabled()) {
376                     logger.debug("Added model object '" + modelName + "' of type [" + modelValue.getClass().getName() +
377                             "] to request in view with name '" + getBeanName() + "'");
378                 }
379             }
380             else {
381                 request.removeAttribute(modelName);
382                 if (logger.isDebugEnabled()) {
383                     logger.debug("Removed model object '" + modelName +
384                             "' from request in view with name '" + getBeanName() + "'");
385                 }
386             }
387         }
388     }
```

5.3 处理模型数据之 Map

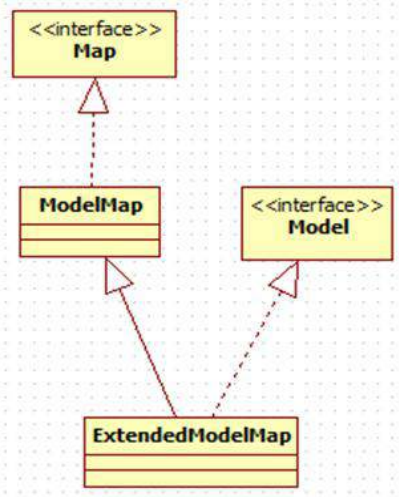
5.3.1 Map 介绍

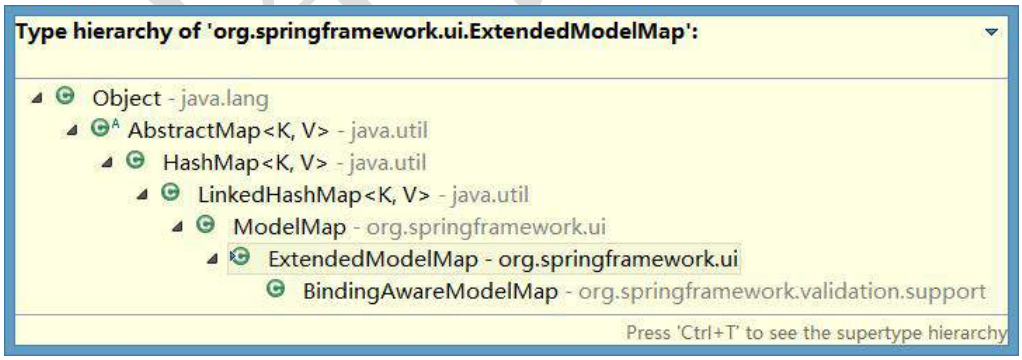
1）Spring MVC 在内部使用了一个 org.springframework.ui.Model 接口存储模型数据具体使用步骤

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";
}
```

2） 增加页面链接

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

3） 增加成功页面，显示结果

```
names: ${requestScope.names }
```

4） 显示结果截图

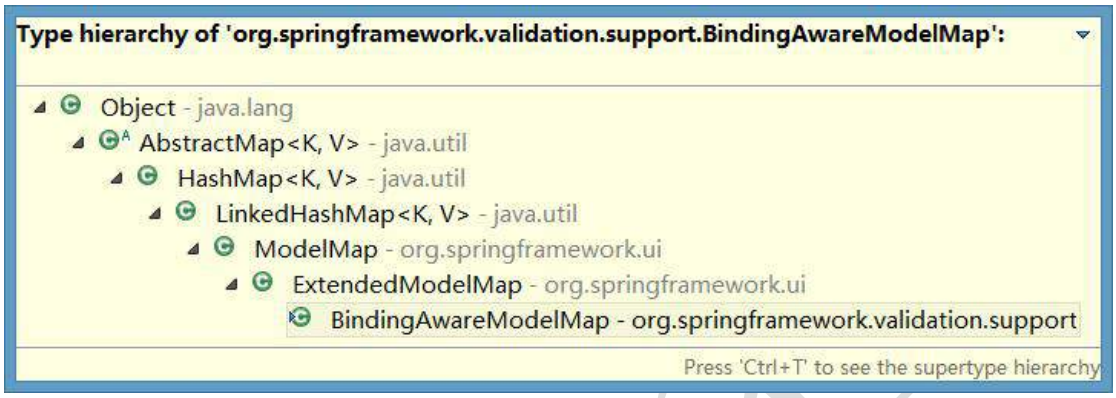
5） 注意问题：Map 集合的泛型，key 为 String,Value 为 Object,而不是 String

6） 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";
}
```

7） 类层次结构



8） 推荐：Map， 便于框架移植。

9） 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);
        }
    }
  }
}
}
```

5.4 处理模型数据之 SessionAttributes 注解【了解】

1） 若希望在多个请求之间共用某个模型属性数据，则可以在控制器类上标注一个 @SessionAttributes, Spring MVC 将在模型中对应的属性暂存到 HttpSession 中。

2） @SessionAttributes 除了可以通过属性名指定需要放到会话中的属性外，还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中

例如：

① @SessionAttributes(types=User.class) 会将隐含模型中所有类型为 User.class 的属性添加到会话中。

② @SessionAttributes(value={“user1”, “user2”})

③ @SessionAttributes(types={User.class, Dept.class})

④ @SessionAttributes(value={“user1”, “user2”}, types={Dept.class})

5.4.1 @SessionAttributes 源码

```
package org.springframework.web.bind.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.TYPE}) //说明这个注解只能应用在类型上面
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface SessionAttributes {
    String[] value() default {};     //推荐使用
    Class<?>[] types() default {};   //范围太广
}
```

5.4.2 实验代码

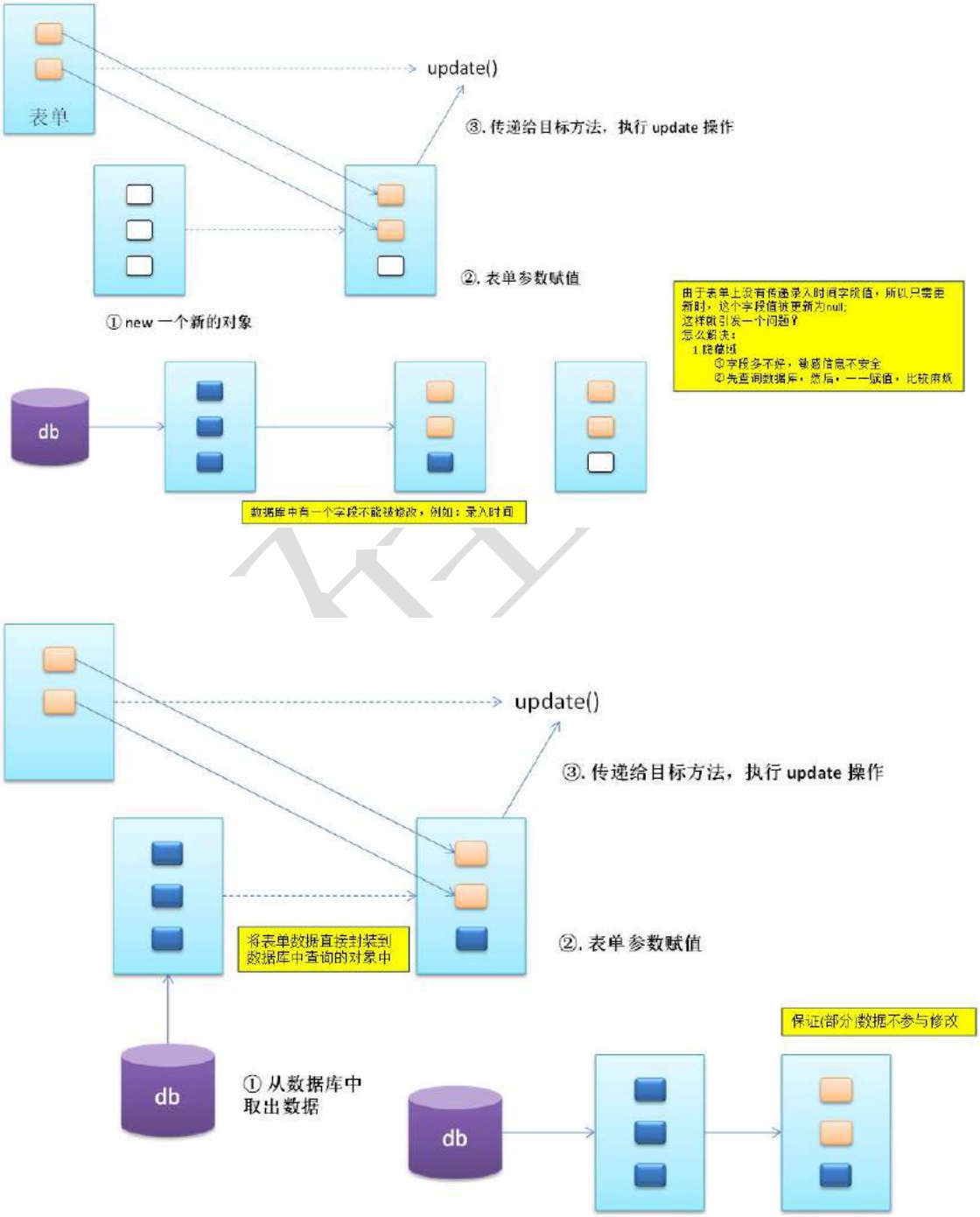
```
@Controller
//@SessionAttributes("user")
@SessionAttributes(value={"user"},types={String.class})
public class SpringMVCController {
/**
 * @SessionAttributes
 *  除了可以通过属性名指定需要放到会话中的属性外(实际上是通过 value 指定 key 值),
 *  还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中(实际上是通过 types 指定类型)
 * 注意：只能放在类的上面，不能修饰方法
 */
  @RequestMapping("/testSessionAttributes")
  public String testSessionAttributes(Map<String,Object> map){
      User user = new User("Tom","123","tom@atguigu.com",22);
      map.put("user", user);
      map.put("school", "atguigu");
      //默认是被存放到 request 域，如果设置了@SessionAttribute 注解，就同时存放到 session 域中
      return "success";
  }
}
```

```
<!--测试 @SessionAttribute 将数据存放到 session 域中 -->
<a href="testSessionAttributes">testSessionAttributes</a>
```

```
request user : ${requestScope.user } <br><br>
session user : ${sessionScope.user } <br><br>
request school : ${requestScope.school } <br><br>
session school : ${sessionScope.school } <br><br>
```

5.5 @ModelAttribute 注解

5.5.1 @ModelAttribute 注解之使用场景【了解】



5.5.2@ModelAttribute 注解之示例代码

1） 在方法定义上使用 @ModelAttribute 注解：Spring MVC 在调用目标处理方法前，会先逐个调用在方法级上标注了 @ModelAttribute 的方法。

2） 在方法的入参前使用 @ModelAttribute 注解：可以从隐含对象中获取隐含的模型数据中获取对象，再将请求参数绑定到对象中，再传入入参

3） 将方法入参对象添加到模型中

4） 示例代码

① 页面表单

```
<!--测试 @ModelAttribute 类似 Struts2 框架的模型驱动 -->
<!--
模拟修改操作：
1.原始数据为：1,Tom,123456,tom@atguigu.com,12
2.密码不需要修改
3.表单回显，模拟操作直接在表单 value 属性上赋值
 -->
<form action="springmvc/testModelAttribute" method="POST">
	<input type="hidden" name="id" value="1"><br>
	username: <input type="text" name="username" value="Tom"/><br>
	email: <input type="text" name="email" value="tom@atguigu.com"/><br>
	age: <input type="text" name="age" value="12"/><br>
	<input type="submit" value="Submit"/>
</form>
```

② 增加@ModelAttribute 注解

```
//1. 由 @ModelAttribute 标记的方法，会在每个目标方法执行之前被 SpringMVC 调用!
@RequestMapping("/testModelAttribute")
public String testModelAttribute(User user){
	System.out.println("user="+user);
	return "success";
}

@ModelAttribute
public void getUser(@RequestParam(value="id",required=false) Integer id,Map<String,Object>
map){
	if(id!=null){
		//模拟从数据库中获取到的 user 对象
		User user = new User(1,"Tom","123456","tom@atguigu.com",12);
		System.out.println("从数据库中查询的对象：user="+user );
		map.put("user", user);
```

```
    }
}
```

③ 测试

④ 异常

//org.springframework.web.HttpSessionRequiredException: Session attribute 'user' required - not found in session

//出现这个异常，是@SessionAttributes(value={"user"},types={String.class})导致的，去掉类上的这个注解

5.5.3 @ModelAttribute 源码参考

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ModelAttribute {
/**
  * The name of the model attribute to bind to.
  * <p>The default model attribute name is inferred from the declared
  * attribute type (i.e. the method parameter type or method return type),
  * based on the non-qualified class name:
  * e.g. "orderAddress" for class "mypackage.OrderAddress",
  * or "orderAddressList" for "List&lt;mypackage.OrderAddress&gt;".
  */
String value() default "";
}
```

5.5.4 @ModelAttribute 注解之运行原理

① 执行@ModelAttribute 注解所修饰的方法，将从数据库中获取的对象存放到 Map 集合中，key 为 user
② SpringMVC 从 Map 集合中获取 user 对象，将表单数据封装到与参数名称对应的 user 对象属性上
③ SpringMVC 将 user 对象作为参数，传递给目标方法。
④ 注意：@ModelAttribute 注解修饰的方法中，放入到 Map 集合中的 key 值，应该和目标方法参数类型的类名称首字母小写一致。

5.5.5 @ModelAttribute 注解之源码分析

1） 调用 @ModelAttribute 注解修饰的方法. 实际上把 @ModelAttribute 方法中 Map 中的数据放在了 implicitModel 中.
2） 解析请求处理器的目标参数, 实际上该目标参数来自于 WebDataBinder 对象的 target 属性
   ① 创建 WebDataBinder 对象
   ② 确定 objectName 属性: 若传入的 attrName 属性值为 "", 则 objectName 为类名第一个字母小写.
   注意: attrName. 若目标方法的 POJO 属性使用了 @ModelAttribute 来修饰, 则 attrName 值即为 @ModelAttribute 的 value 属性值
   ③ 确定 target 属性:
      - 在 implicitModel 中查找 attrName 对应的属性值.
        - 若存在, ok
        - 若不存在: 则验证当前 Handler 是否使用了@SessionAttributes 进行修饰,
      - 若使用了, 则尝试从 Session 中获取 attrName 所对应的属性值.
        - 若 session 中没有对应的属性值, 则抛出了异常.
        - 若 Handler 没有使用 @SessionAttributes 进行修饰, 或 @SessionAttributes 中没有使用 value 值指定的 key 和 attrName 相匹配, 则通过反射创建了 POJO 对象
3） SpringMVC 把表单的请求参数赋给了 WebDataBinder 的 target 对应的属性.
4） SpringMVC 会把 WebDataBinder 的 attrName 和 target 给到 implicitModel. 进而传到 request 域对象中.
5） 把 WebDataBinder 的 target 作为参数传递给目标方法的入参.

5.5.6 SpringMVC 确定目标方法 POJO 类型入参的过程

1） 确定一个 key:
   ① 若目标方法的 POJO 类型的参数木有使用 @ModelAttribute 作为修饰,

则 key 为 POJO 类名第一个字母的小写

② 若使用了@ModelAttribute 来修饰，则 key 为 @ModelAttribute 注解的 value 属性值.

2） 在 implicitModel 中查找 key 对应的对象，若存在，则作为入参传入

若在 @ModelAttribute 标记的方法中在 Map 中保存过，且 key 和 1）确定的 key 一致，则会获取到.

3） 若 implicitModel 中不存在 key 对应的对象，则检查当前的 Handler 是否使用 @SessionAttributes 注解修饰，

4） 若使用了该注解，且 @SessionAttributes 注解的 value 属性值中包含了 key，则会从 HttpSession 中来获取 key 所对应的 value 值，若存在则直接传入到目标方法的入参中. 若不存在则将抛出异常.

5） 若 Handler 没有标识 @SessionAttributes 注解或 @SessionAttributes 注解的 value 值中不包含 key，则会通过反射来创建 POJO 类型的参数，传入为目标方法的参数

6） SpringMVC 会把 key 和 POJO 类型的对象保存到 implicitModel 中，进而会保存到 request 中.

5.5.7@ModelAttribute 注解修饰 POJO 类型的入参

```
@RequestMapping("/testModelAttribute")
//public String testModelAttribute(User user){

public String testModelAttribute(@ModelAttribute("abc") User user){

   System.out.println("修改  user="+user);
   return "success";
}
/**
 * @ModelAttribute 注解也可以来修饰目标方法 POJO 类型的入参, 其 value 属性值有如下的作用:
1). SpringMVC 会使用 value 属性值在 implicitModel 中查找对应的对象, 若存在则会直接传入到目标方法的入参中.
2). SpringMVC 会以 value 为 key, POJO 类型的对象为 value, 存入到 request 中.
 */
@ModelAttribute
public void getUser(@RequestParam(value="id",required=false) Integer id,Map<String,Object> map){
      if(id!=null){
            //模拟从数据库中获取到的 user 对象
            User user = new User(1,"Tom","123456","tom@atguigu.com",12);
            System.out.println("从数据库中查询的对象：user="+user );
            //map.put("user", user); //BindingAwareModelMap

            //map.put("abc", user); //BindingAwareModelMap
      }
}
```

```
user user: ${requestScope.user }
<br><br>
abc user: ${requestScope.abc }
<br><br>
```

5.5.8@sessionAttributes 注解引发的异常

1） 由@SessionAttributes 引发的异常

```
org.springframework.web.HttpSessionRequiredException:
Session attribute 'user' required - not found in session
```

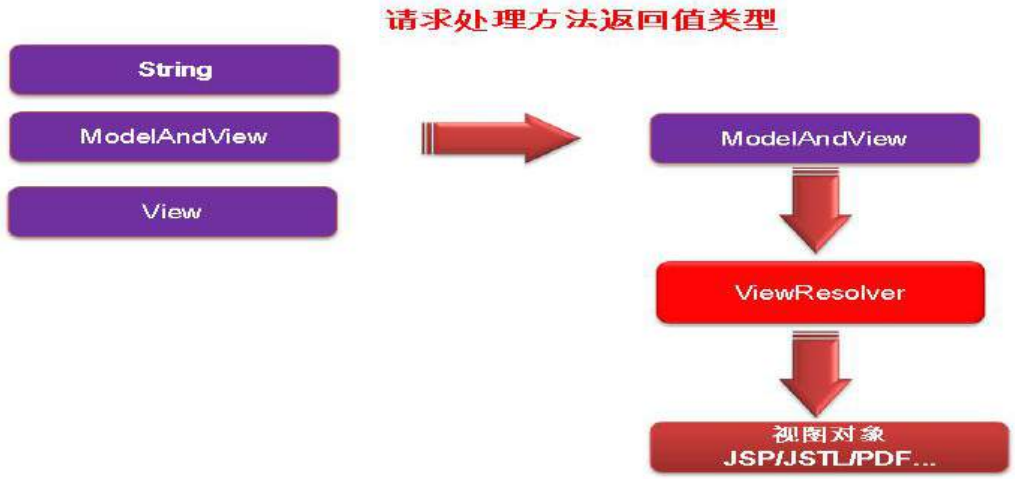
2） 如果在处理类定义处标注了@SessionAttributes(“xxx”)，则尝试从会话中获取该属性，并将其赋给该入参，然后再用请求消息填充该入参对象。如果在会话中找不到对应的属性，则抛出 HttpSessionRequiredException 异常

```
bindObject = this.sessionAttributeStore.retrieveAttribute(webRequest, name);
if (bindObject == null) {
    raiseSessionRequiredException("Session attribute '" + name + "' required - not found
}
```

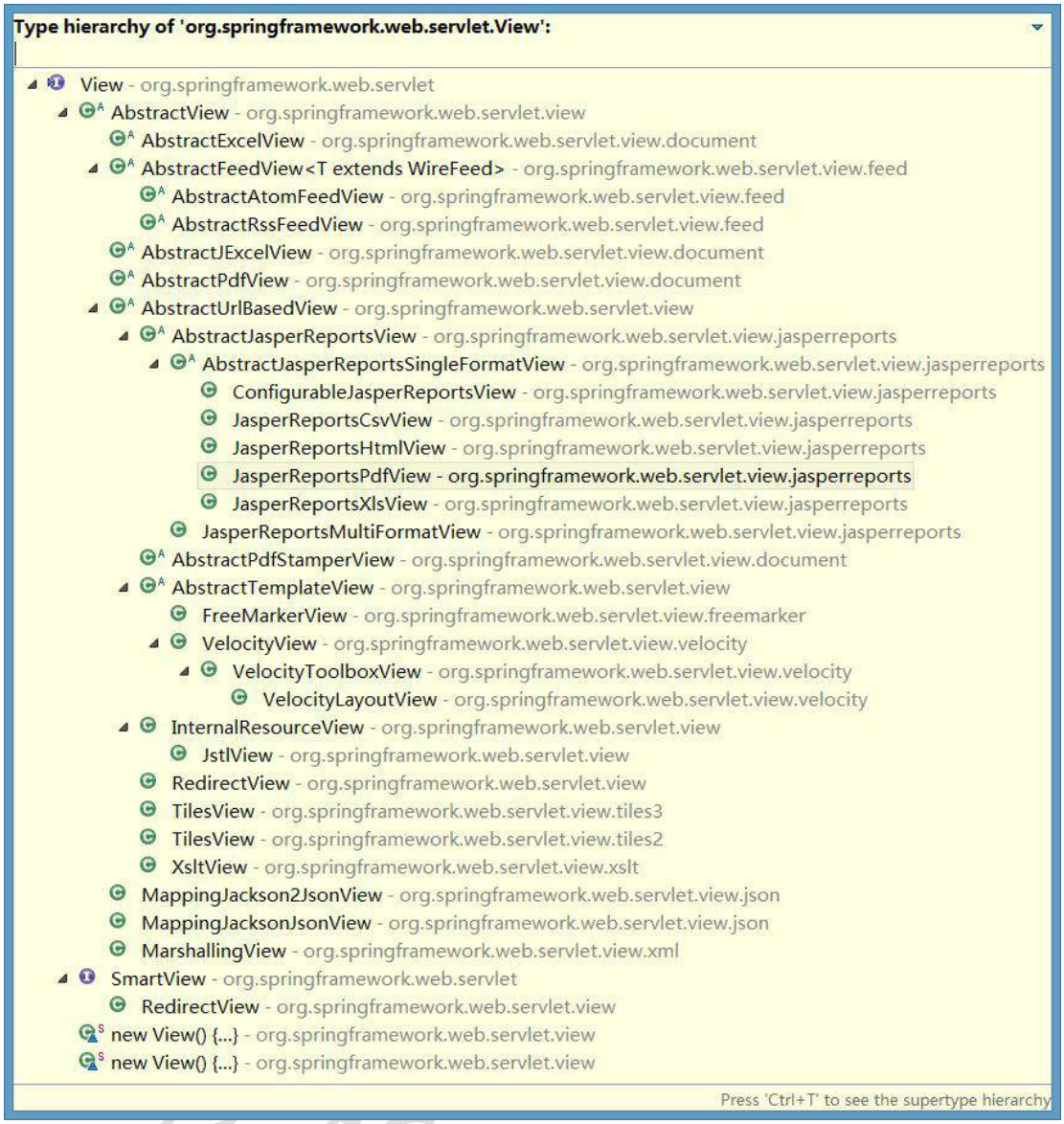
第 6 章 视图解析

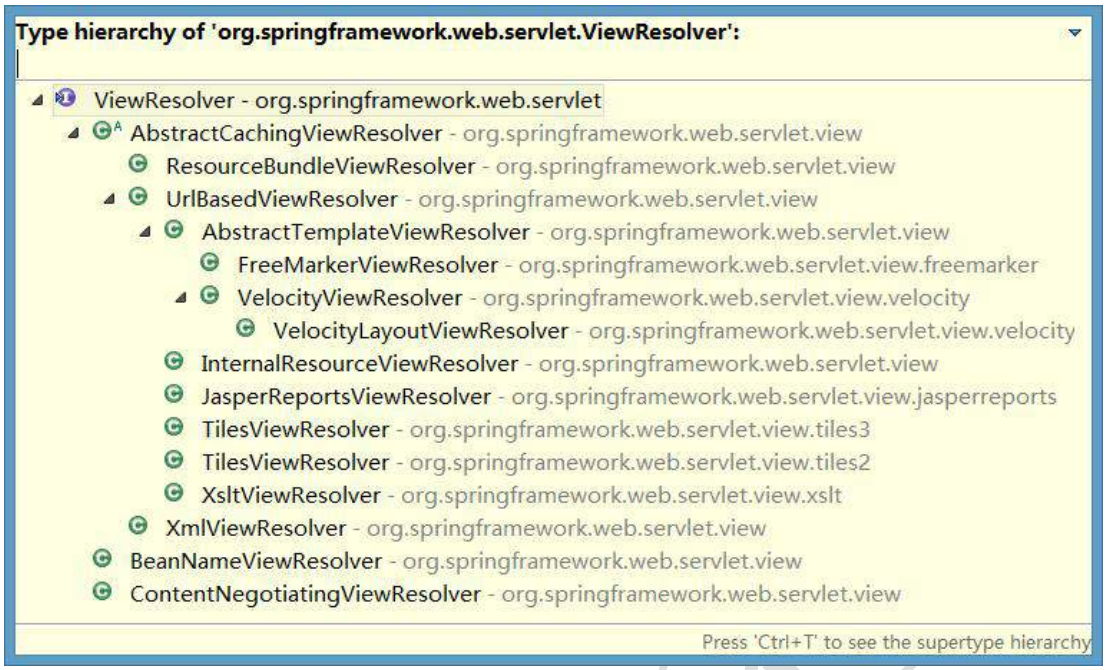
6.1 SpringMVC 如何解析视图概述

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。



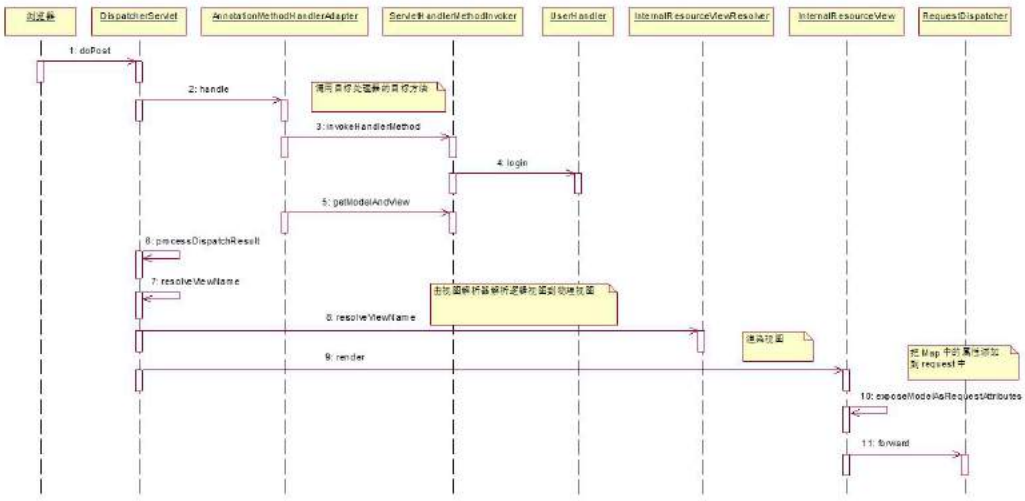
2） 视图解析源码分析：重要的两个接口





3） 断点调试源码

4） 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：

3） **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

大类	视图类型	说明
URL 视资源图	InternalResourceView	将 JSP 或其它资源封装成一个视图，是 InternalResourceViewResolver 默认使用的视图实现类
	JstlView	如果 JSP 文件中使用了 JSTL 国际化标签的功能，则需要使用该视图类
文档视图	AbstractExcelView	Excel 文档视图的抽象类。该视图类基于 POI 构造 Excel 文档
	AbstractPdfView	PDF 文档视图的抽象类，该视图类基于 iText 够着 PDF 文档
报表视图	ConfigurableJsperReportsView	几个使用 JasperReports 报表技术的视图
	JasperReportsCsvView	
	JasperReportsMultiFormatView	
	JasperReportsHtmlView	
	JasperReportsPdfView	
	JasperReportsXlsView	
JSON 视图	MappingJacksonJsonView	将模型数据通过 Jackson 开源框架的 ObjectMapper 以 JSON 方式输出。

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**

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n"></property>
</bean>
```

3） 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

```
<mvc:view-controller path="springmvc/testJstlView" view-name="success"/>
```

6.5.1 实验代码

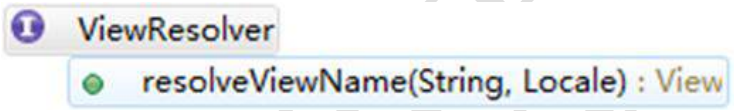
1）增加 jstl 标签 jar 包（**断点调试，这时的 View 对象就是 JstlView**）

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 Spring WEB 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

大类	视图类型	说明
解析为 Bean 的名字	BeanNameViewResolver	将逻辑视图名解析为一个 Bean，Bean 的 id 等于逻辑视图名
解析为 URL 文件	InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为一个保存在 WEB-INF 目录下的程序文件（如 JSP）
	JasperReportsViewResolver	JasperReports 是一个基于 Java 的开源报表工具，该解析器将视图名解析为报表文件对应的 URL
模版文件视图	FreeMarkerViewResoler	解析为基于 FreeMarker 模版技术的模版文件
	VelocityViewResolver	解析为基于 Velocity 模版技术的模版文件
	VelocityLayoutViewResolver	

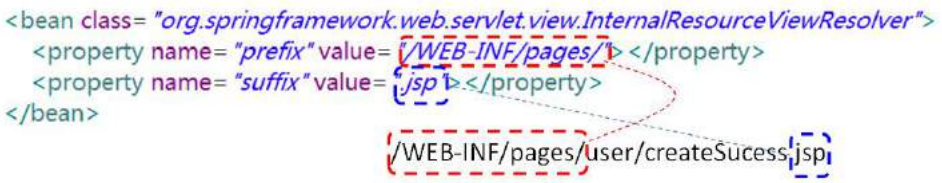
1） 程序员可以选择一种视图解析器或混用多种视图解析器

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：



②

6.8 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：

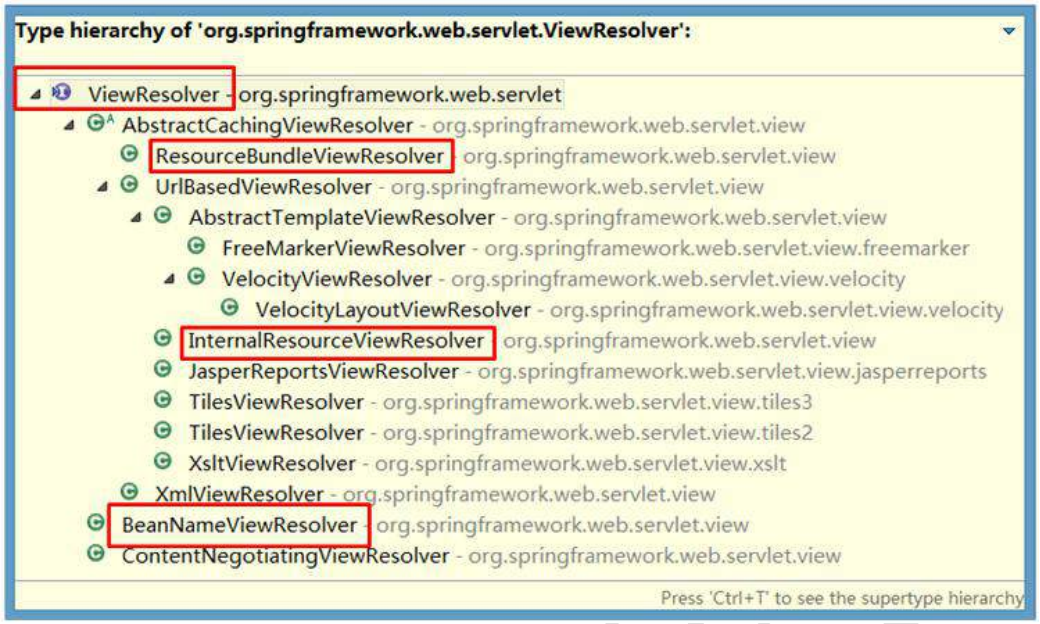
```
http://localhost:8080/SpringMVC_02_View/success
```

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 自定义视图【了解】

1） 自定义视图（需要加入 SpringMVC，那么，一定需要实现框架的接口）
2） 若希望使用 Excel 展示数据列表，仅需要扩展 SpringMVC 提供的 **AbstractExcelView** 或 AbstractJExcelView 即可。
3） 实现 buildExcelDocument() 方法，在方法中使用模型数据对象构建 Excel 文档就可以了。
4） **AbstractExcelView 基于 POI API**，而 AbstractJExcelView 是基于 JExcelAPI 的。
5） **视图对象需要配置 IOC 容器中的一个 Bean，使用 BeanNameViewResolver 作为视图解析器即可**
6） 若希望直接在浏览器中直接下载 Excel 文档，则可以设置响应头 **Content-Disposition** 的值为 **attachment;filename=xxx.xls**



6.9.1 实验代码

1） 页面链接

```
<a href="springmvc/testView">testView</a>
```

2） 控制器方法

```
@RequestMapping("/testView")
public String testView(){
    System.out.println("testView...");
    return "helloView"; //与视图 Bean 对象的 id 一致
}
```

3） 自定义视图

```
package com.atguigu.springmvc.view;

import java.util.Date;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.View;

@Component
public class HelloView implements View {
```

```
    @Override
    public String getContentType() {
        return "text/html";
    }

    @Override
    public void render(Map<String, ?> model, HttpServletRequest request,
                       HttpServletResponse response) throws Exception {
        response.getWriter().println("HelloView - time = " + new Date());
    }
}
```

4） 声明视图解析器

```
<!-- 配置视图解析器：按照 bean 的名称查找视图 -->
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver">
    <property name="order" value="100"></property>
</bean>
```

5）InternalResourceViewResolver 默认的优先级：private int order = Integer.MAX_VALUE;

6.9.2 源码参考

```
public class BeanNameViewResolver extends WebApplicationObjectSupport
implements ViewResolver, Ordered {

    private int order = Integer.MAX_VALUE;    // default: same as non-Ordered

    public void setOrder(int order) {
        this.order = order;
    }

    @Override
    public int getOrder() {
        return order;
    }

    @Override
    public View resolveViewName(String viewName, Locale locale) throws BeansException {
        ApplicationContext context = getApplicationContext();
        if (!context.containsBean(viewName)) {
            //说明视图组件必须增加到 Spring 的 IOC 容器中，所以需要@Component
```

```
                // Allow for ViewResolver chaining.
                return null;
            }
            return context.getBean(viewName, View.class);
        }
}
```

6.10 重定向

1） 关于重定向

① 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

② 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

③ redirect:success.jsp：会完成一个到 success.jsp 的重定向的操作

④ forward:success.jsp：会完成一个到 success.jsp 的转发操作

2） 定义页面链接

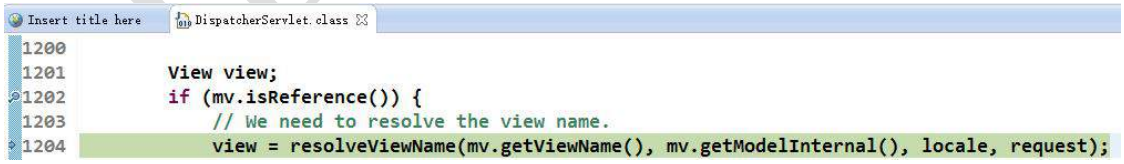
```
<a href="springmvc/testRedirect">testRedirect</a>
```

3） 定义控制器方法

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

4） 源码分析：重定向原理

1. 源码分析：重定向原理



```java
protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,
        HttpServletRequest request) throws Exception {

    for (ViewResolver viewResolver : this.viewResolvers) {
        View view = viewResolver.resolveViewName(viewName, locale);
        if (view != null) {
            return view;
        }
    }
    return null;
}
```

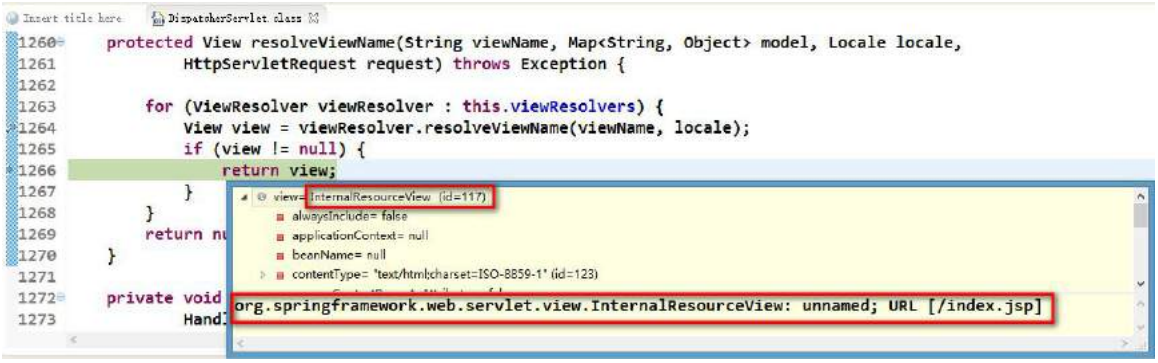
```java
@Override
public View resolveViewName(String viewName, Locale locale) throws Exception {
    if (!isCache()) {
        return createView(viewName, locale);
    }
    else {
        Object cacheKey = getCacheKey(viewName, locale);
        View view = this.viewAccessCache.get(cacheKey);
        if (view == null) {
```

```java
protected View createView(String viewName, Locale locale) throws Exception {
    // If this resolver is not supposed to handle the given view,
    // return null to pass on to the next resolver in the chain.
    if (!canHandle(viewName, locale)) {
        return null;
    }
    // Check for special "redirect:" prefix.
    if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
        String redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length());
        RedirectView view = new RedirectView(redirectUrl, isRedirectContextRelative(), isRedirectHttp10Compatible()
        return applyLifecycleMethods(viewName, view);
    }
    // Check for special "forward:" prefix.
    if (viewName.startsWith(FORWARD_URL_PREFIX)) {
        String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
        return new InternalResourceView(forwardUrl);
    }
    // Else fall back to superclass implementation: calling loadView.
    return super.createView(viewName, locale);
}
```

```java
// Delegate to the View object for rendering.
if (logger.isDebugEnabled()) {
    logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" + getServletName() + "'");
}
try {
    view.render(mv.getModelInternal(), request, response);
}
```

```
view= RedirectView (id=66)
  applicationContext= XmlWebApplicationContext (id=81)
  beanName= "redirect:/index.jsp" (id=89)
  contentType= "text/html;charset=ISO-8859-1" (id=98)
  contextRelative= true
org.springframework.web.servlet.view.RedirectView: name 'redirect:/index.jsp'; URL [/index.jsp]
```

- 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

ID	LastName	Email	Gender	Department	Edit	Delete
1001	AA	aa@163.com	Male	AA	Edit	Delete
1003	CC	cc@163.com	Female	CC	Edit	Delete
1002	BB	bb@163.com	Male	BB	Edit	Delete
1005	EE	ee@163.com	Male	EE	Edit	Delete
1004	DD	dd@163.com	Female	DD	Edit	Delete

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。

ID	LastName	Email	Gender	Department	Edit	Delete
1001	AA	aa@163.com	Male	AA	Edit	Delete
1003	CC	cc@163.com	Female	CC	Edit	Delete
1002	BB	bb@163.com	Male	BB	Edit	Delete
1005	EE	ee@163.com	Male	EE	Edit	Delete
1004	DD	dd@163.com	Female	DD	Edit	Delete
1006	FF	ff@163.com	Male	CC	Edit	Delete

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

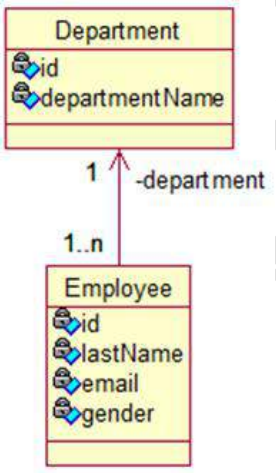
1） URI：emp/{id}
2） 请求方式：GET
3） 显示效果：回显表单。

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

2） 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>
```

3） 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>
```

4） 将 POST 请求转换为 PUT 或 DELETE 请求

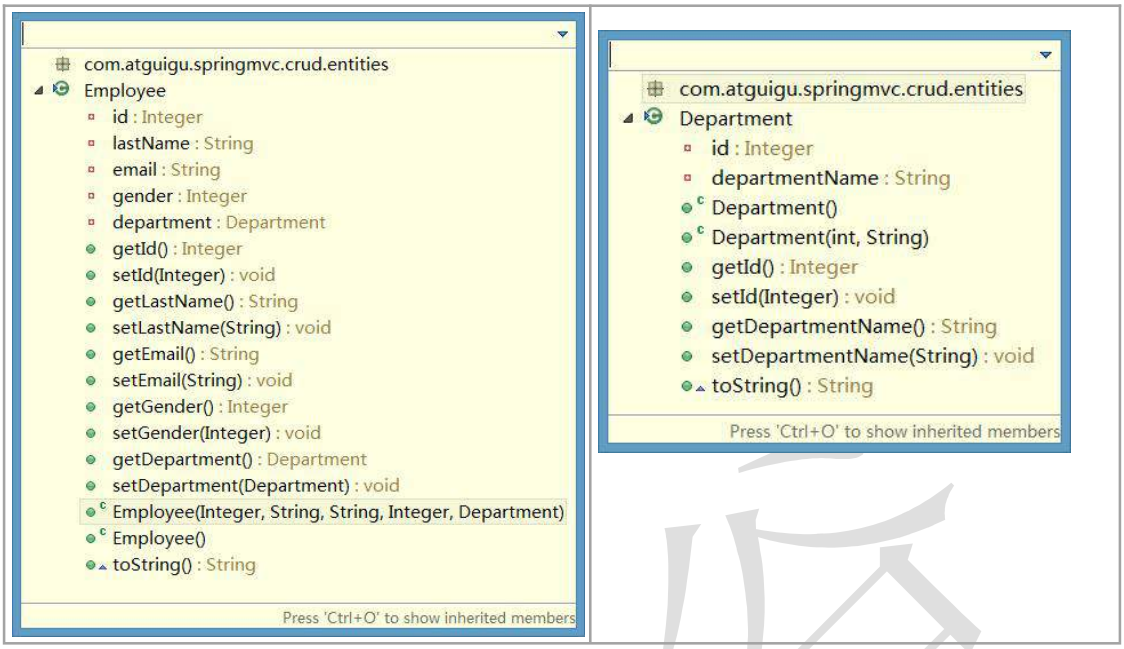
```
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

5） 创建相关页面

/WEB-INF/views/list.jsp

index.jsp

6） 增加实体类



7） 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")));
```

```
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"));
}
```

```
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);
}
}
```

```
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>
```

2） 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";
	}
}
```

3） 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>
```

2） 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";
```

```
}
```

3） 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>
```

4） 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>
```

2） 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>
```

2） 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

2） 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'
```

3） 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/>

4） 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"/>
```

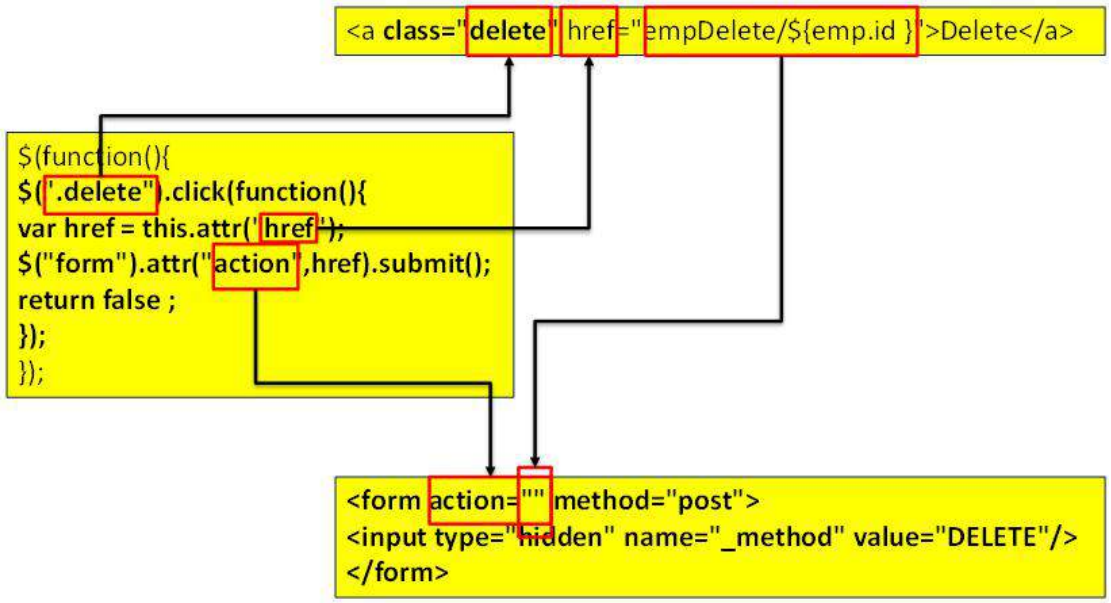
7.7.5 通过 jQuery 转换为 DELETE 请求

```
<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>
```

2） 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";
}
```

3） 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 数据绑定流程分析【了解】

8.1 提出问题

日期字符串格式的表单参数，提交后转换为 Date 类型

```
 <!--    解决问题：
 1.数据类型转换
 2.数据格式
 3.数据校验
 -->
BirthDay :<form:input path="birthDay"/>
```

```
Employee 类中增加日期类型属性：
//关于类型转换
private Date birthDay ;
```

8.2 数据绑定流程原理

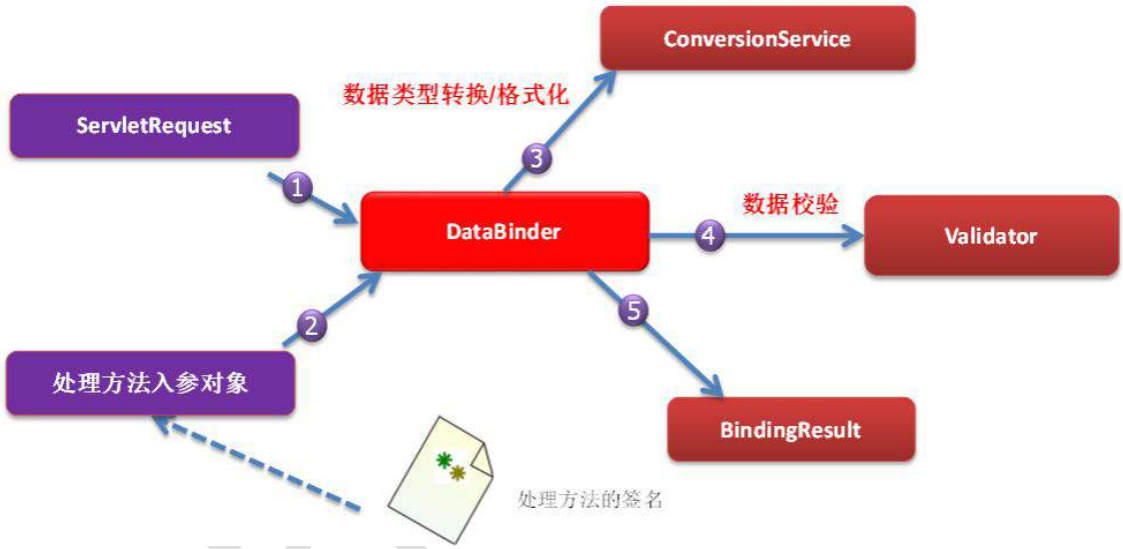
1） Spring MVC 主框架将 ServletRequest 对象及目标方法的入参实例传递给 WebDataBinderFactory 实例，以创建 **DataBinder** 实例对象

2） DataBinder 调用装配在 Spring MVC 上下文中的 **ConversionService** 组件进行**数据类型转换、数据格式化**工作。将 Servlet 中的请求信息填充到入参对象中

3） 调用 **Validator** 组件对已经绑定了请求消息的入参对象进行数据合法性校验，并最终生成数据绑定结果 **BindingData** 对象

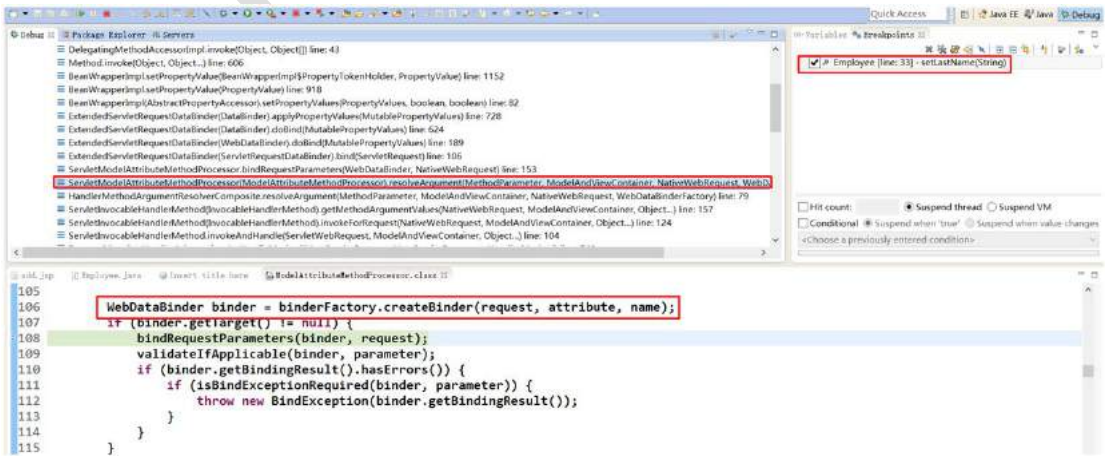
4） Spring MVC 抽取 **BindingResult** 中的入参对象和校验错误对象，将它们赋给处理方法的响应入参

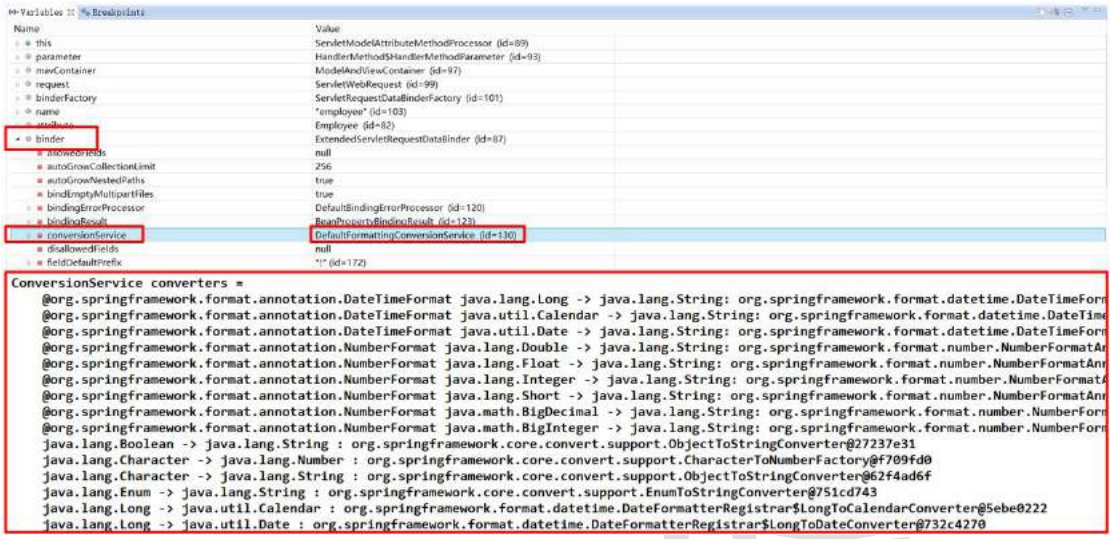
5） Spring MVC 通过反射机制对目标处理方法进行解析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 **DataBinder**，运行机制如下：

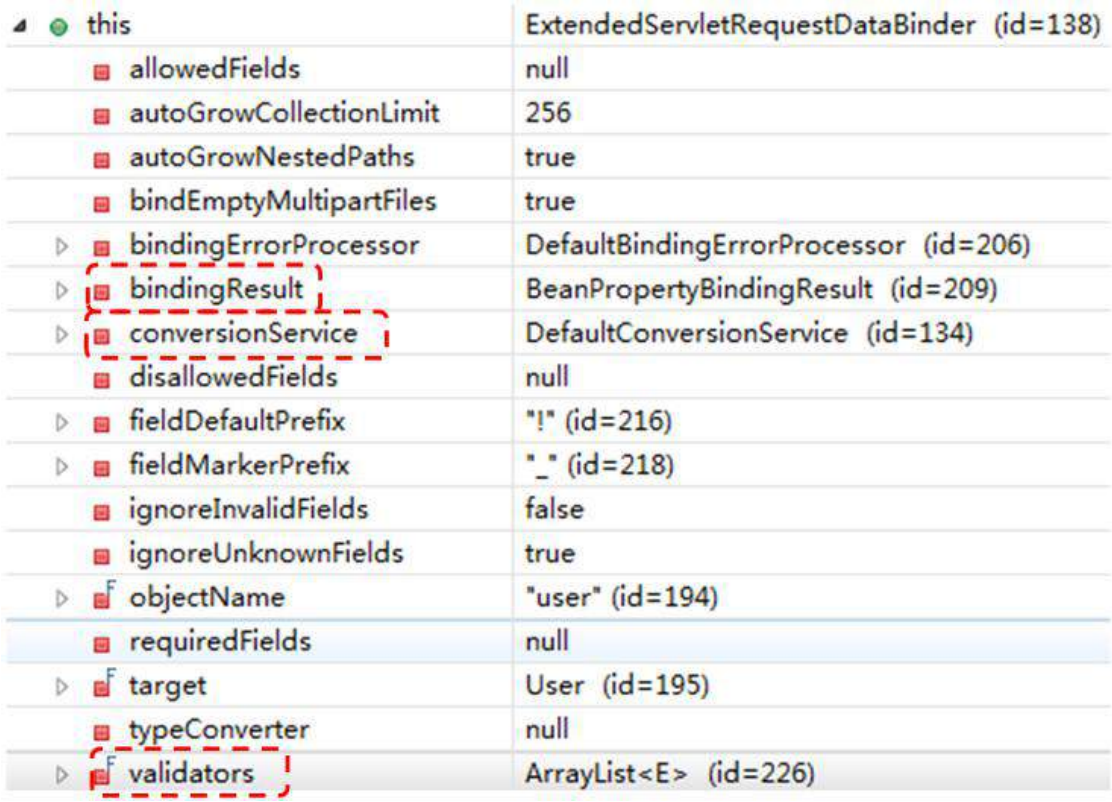


8.3 通过 debug 调试流程

1） 查看数据绑定流程，在 Employee 类的 set 方法上设置断点调试







2） Spring MVC 上下文中内建了很多转换器，可完成大多数 Java 类型的转换工作。

```
ConversionService converters =
java.lang.Boolean->java.lang.String:
org.springframework.core.convert.support.ObjectToStringConverter@f874ca
java.lang.Character -> java.lang.Number : CharacterToNumberFactory@f004c9
java.lang.Character -> java.lang.String : ObjectToStringConverter@68a961
java.lang.Enum -> java.lang.String : EnumToStringConverter@12f060a
java.lang.Number -> java.lang.Character : NumberToCharacterConverter@1482ac5
java.lang.Number -> java.lang.Number : NumberToNumberConverterFactory@126c6f
```

```
java.lang.Number -> java.lang.String : ObjectToStringConverter@14888e8
java.lang.String -> java.lang.Boolean : StringToBooleanConverter@1ca6626
java.lang.String -> java.lang.Character : StringToCharacterConverter@1143800
java.lang.String -> java.lang.Enum : StringToEnumConverterFactory@1bba86e
java.lang.String -> java.lang.Number : StringToNumberConverterFactory@18d2c12
java.lang.String -> java.util.Locale : StringToLocaleConverter@3598e1
java.lang.String -> java.util.Properties : StringToPropertiesConverter@c90828
java.lang.String -> java.util.UUID : StringToUUIDConverter@a42f23
java.util.Locale -> java.lang.String : ObjectToStringConverter@c7e20a
java.util.Properties -> java.lang.String : PropertiesToStringConverter@367a7f
java.util.UUID -> java.lang.String : ObjectToStringConverter@112b07f  ……
```

3） 查看：StringToNumberConverterFactory 源码，在 getConverter()方法中设置断点，在执行 set 方法(性别字段)前会调用该方法。

```
package org.springframework.core.convert.support;

import org.springframework.core.convert.converter.Converter;
import org.springframework.core.convert.converter.ConverterFactory;
import org.springframework.util.NumberUtils;

final class StringToNumberConverterFactory implements ConverterFactory<String, Number> {

    @Override
    public <T extends Number> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToNumber<T>(targetType);
    }

    private static final class StringToNumber<T extends Number> implements Converter<String, T> {

        private final Class<T> targetType;

        public StringToNumber(Class<T> targetType) {
            this.targetType = targetType;
        }

        @Override
        public T convert(String source) {
            if (source.length() == 0) {
                return null;
            }
            return NumberUtils.parseNumber(source, this.targetType);
        }
```

```
    }
}
```

8.4 自定义类型转换器

8.4.1 类型转换器概述

① ConversionService 是 Spring 类型转换体系的核心接口。

② 可以利用 ConversionServiceFactoryBean 在 Spring 的 IOC 容器中定义一个 ConversionService. Spring 将自动识别出 IOC 容器中的 ConversionService，并在 Bean 属性配置及 Spring MVC 处理方法入参绑定等场合使用它进行数据的转换

③ 可通过 ConversionServiceFactoryBean 的 converters 属性注册自定义的类型转换器

④ 例如：

```
<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="com.atguigu.springmvc.UserConverter"></bean>
    </list>
  </property>
</bean>
```

8.4.2 Spring 支持的类型转换器

① Spring 定义了 3 种类型的转换器接口，实现任意一个转换器接口都可以作为自定义转换器注册到 ConversionServiceFactoryBean 中：

② Converter<S,T>：将 S 类型对象转为 T 类型对象

③ ConverterFactory：将相同系列多个 “同质” Converter 封装在一起。如果希望将一种类型的对象转换为另一种类型及其子类的对象（例如将 String 转换为 Number 及 Number 子类（Integer、Long、Double 等）对象）可使用该转换器工厂类

④ GenericConverter：会根据源类对象及目标类对象所在的宿主类中的上下文信息进行类型转换

8.4.3 自定义类型转换器示例

1） 需求：字符串转换为对象。

2） 步骤：

① 定义页面

```
<form action="empAdd" method="POST">
    <!-- 解决问题：
```

```
            1.数据类型转换
            2.数据格式
            3.数据校验
            自定义类型转换器：
                将字符串转换为 Employee 对象,完成添加功能
BirthDay :<input type="text" name="birthDay"/><br><br>
-->
<!-- 字符串格式：lastName-email-gender-department.id
     例如：GG-gg@atguigu.com-0-105
 -->
Employee : <input type="text" name="employee"/>
            <input type="submit" value="Submit"><br><br>
</form>
```

② 控制器方法

```
package com.atguigu.springmvc.crud.handlers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.atguigu.springmvc.crud.dao.EmployeeDao;
import com.atguigu.springmvc.crud.entities.Employee;

@Controller
public class TypeConversionHandler {

    @Autowired
    private EmployeeDao employeeDao ;

    // String -> Employee 需要类型转换器帮忙
    @RequestMapping("/empAdd")
    public String empAdd(@RequestParam(value="employee") Employee employee){
        System.out.println("TypeConversionHandler - " + employee);
        employeeDao.save(employee);
        return "redirect:/empList";
    }
}
```

③ 自定义类型转换器

```
package com.atguigu.springmvc.converter;
```

```
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import com.atguigu.springmvc.crud.entities.Department;
import com.atguigu.springmvc.crud.entities.Employee;

/**
 * 将字符串转换为 Employee 对象类型
 */
@Component
public class StringToEmployeeConverter implements Converter<String, Employee> {

	@Override
	public Employee convert(String source) {
		if(source!=null){
			String[] strs = source.split("-");
			if(strs!=null && strs.length == 4){
				String lastName = strs[0];
				String email = strs[1];
				Integer gender = Integer.parseInt(strs[2]);
				Integer deptId = Integer.parseInt(strs[3]);
				Department dept = new Department();
				dept.setId(deptId);
				Employee employee = new Employee(null,lastName,email,gender,dept);
				System.out.println(source+"--converter--"+employee);
				return employee ;
			}
		}
		return null;
	}
}
```

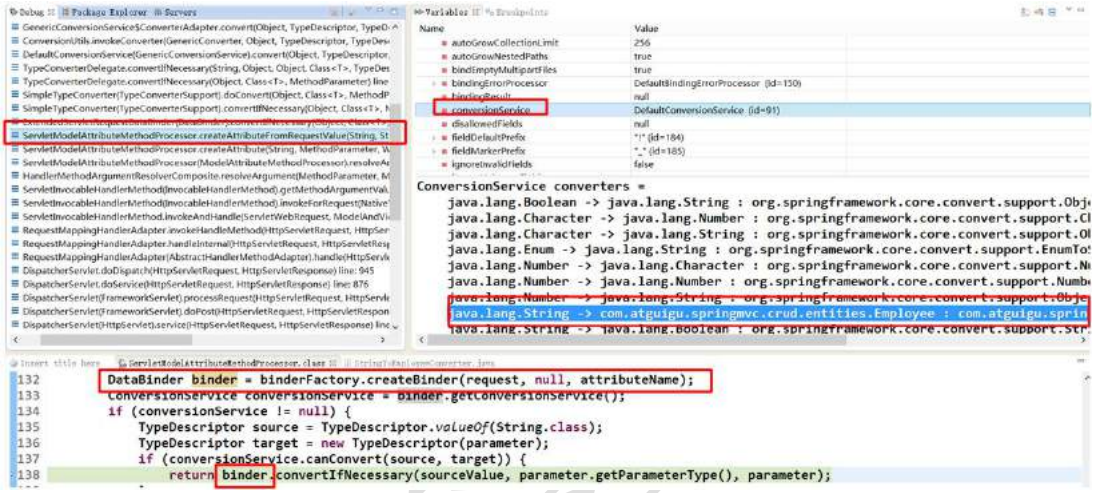
④ 声明类型转换器服务

```
<bean id="conversionService"
	class="org.springframework.context.support.ConversionServiceFactoryBean">
	<property name="converters">
		<set>
			<!-- 引用类型转换器 -->
			<ref bean="stringToEmployeeConverter"/>
		</set>
	</property>
</bean>
```

⑤ <mvc:annotation-driven conversion-service="conversionService"/> 会将自定义的 ConversionService 注册到 Spring MVC 的上下文中

8.4.4 Debug 调试

1） 增加新的转换器，之前的转换器是否还好使呢？好使。
查看，框架出厂设置，与目前将我们的自定义类型转换器加入出厂设置中。



2） 当配置了<mvc:annotation-driven/>后，会自动加载 org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping 和 org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

8.5 <mvc:annotation-driven/>配置在什么时候必须配置？

1） 直接配置响应的页面：无需经过控制器来执行结果 ；但会导致其他请求路径失效，需要配置 mvc:annotation-driven 标签

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

2） RESTful-CRUD 操作，删除时，通过 jQuery 执行 delete 请求时，找不到静态资源，需要配置 mvc:annotation-driven 标签

<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。

3） 配置类型转换器服务时，需要指定转换器服务引用
<mvc:annotation-driven conversion-service=“conversionService” /> 会将自定义的 ConversionService 注册到 Spring MVC 的上下文中

4） 后面完成 JSR 303 数据验证，也需要配置

8.5.1 关于 <mvc:annotation-driven /> 作用

1） <mvc:annotation-driven /> 会自动注册：
RequestMappingHandlerMapping 、
RequestMappingHandlerAdapter 与
ExceptionHandlerExceptionResolver 三个 bean。

2） 还将提供以下支持：
① 支持使用 **ConversionService** 实例对表单参数进行类型转换
② 支持使用 **@NumberFormat、@DateTimeFormat** 注解完成数据类型的格式化
③ 支持使用 **@Valid** 注解对 JavaBean 实例进行 JSR 303 验证
④ 支持使用 **@RequestBody** 和 **@ResponseBody** 注解

3） 结合**源码分析（在 bean 对象的 set 方法上设置断点进行调试）**
① 既没有配置 <mvc:default-servlet-handler/>
也没有配置 <mvc:annotation-driven/>

handlerAdapters	ArrayList<E> (id=199)
elementData	Object[3] (id=215)
[0]	HttpRequestHandlerAdapter (id=222)
[1]	SimpleControllerHandlerAdapter (id=224)
[2]	AnnotationMethodHandlerAdapter (id=79)

都没有配置情况下，AnnotationMethodHandlerAdapter 是默认出厂设置，干活的(过期)。
另外：conversionService 是 null(类型转换器是不起作用的)

```
四月 30, 2016 3:52:21 下午 org.springframework.web.servlet.PageNotFound noHandlerFound
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'
```

② 配置了 <mvc:default-servlet-handler/> 但没有配置 <mvc:annotation-driven/>

handlerAdapters	ArrayList<E> (id=121)
elementData	Object[2] (id=133)
[0]	HttpRequestHandlerAdapter (id=69)
[1]	SimpleControllerHandlerAdapter (id=140)

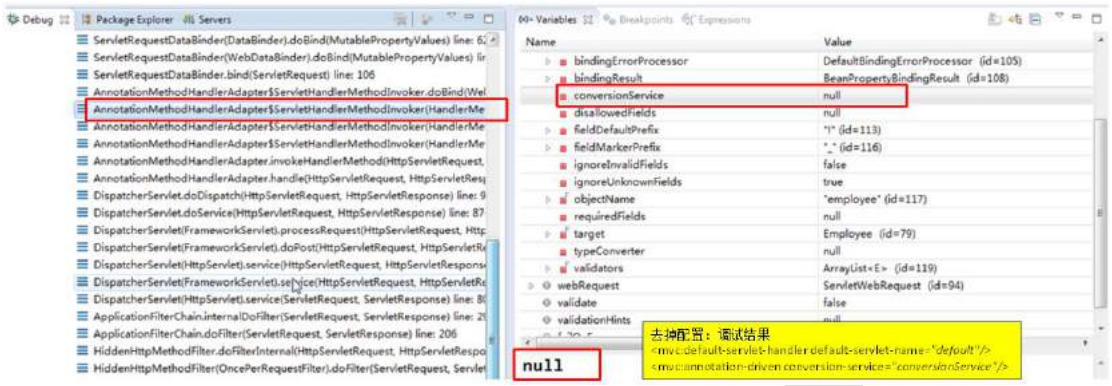
AnnotationMethodHandlerAdapter 被取消，解决了静态资源查找，但是 @RequestMapping 不好使了。

③ 既配置了 <mvc:default-servlet-handler/> 又配置 <mvc:annotation-driven/> 【重要】

handlerAdapters	ArrayList<E> (id=121)
elementData	Object[3] (id=135)
[0]	HttpRequestHandlerAdapter (id=142)
[1]	SimpleControllerHandlerAdapter (id=144)
[2]	RequestMappingHandlerAdapter (id=70)

AnnotationMethodHandlerAdapter 被替换成 RequestMappingHandlerAdapter 来干活了。

如果没有配置**<mvc:annotation-driven/>标签时，conversionService 为 null.**



AnnotationMethodHandlerAdapter 已经过时，Spring3.2 推荐 RequestMappingHandlerAdapter 来替代。所以说，默认情况下，没有配置这两个配置时，HelloWorld 程序可以正常运行，但是，涉及到静态资源查找的时候，就必须配置这个**<mvc:annotation-driven/>配置了**

8.6 InitBinder 注解【了解】

1） @InitBinder

① 由 **@InitBinder** 标识的方法，**可以对 WebDataBinder 对象进行初始化**。WebDataBinder 是 DataBinder 的子类，用于完成由表单字段到 JavaBean 属性的绑定

② **@InitBinder 方法不能有返回值**，它必须声明为 void。

③ **@InitBinder 方法的参数通常是 WebDataBinder**

```
/**
 * 不自动绑定对象中的 roleSet 属性，另行处理。
 */
@InitBinder
public void initBinder(WebDataBinder dataBinder){
    dataBinder.setDisallowedFields("roleSet");
}
```

2） 实验代码

```
@InitBinder
public void initBinder(WebDataBinder dataBinder){
    dataBinder.setDisallowedFields("lastName");
}
```

8.7 数据的格式化

8.7.1 数据格式化概述

1） 对属性对象的输入/输出进行格式化，从其本质上讲依然属于 “类型转换” 的范畴。

2） Spring 在格式化模块中定义了一个实现 ConversionService 接口的 **FormattingConversionService** 实现类，该实现类扩展了 GenericConversionService，因此它**既具有类型转换的功能，又具有格式化的功能**

3） FormattingConversionService 拥有一个 **FormattingConversionServiceFactoryBean** 工厂类，后者用于在 Spring 上下文中构造前者，FormattingConversionServiceFactoryBean 内部已经注册了：

① NumberFormatAnnotationFormatterFactory：支持对数字类型的属性使用 **@NumberFormat** 注解

② JodaDateTimeFormatAnnotationFormatterFactory：支持对日期类型的属性使用 @DateTimeFormat 注解

4） 装配了 FormattingConversionServiceFactroyBean 后，就可以在 Spring MVC 入参绑定及模型数据输出时使用注解驱动了。

① **<mvc:annotation-driven/> 默认创建的 ConversionService 实例即为** DefaultFormattingConversionService

8.7.2 日期格式化概述

1） @DateTimeFormat 注解可对 java.util.Date、java.util.Calendar、java.long.Long 时间类型进行标注：

① pattern 属性：类型为字符串。指定解析/格式化字段数据的模式，如：”

yyyy-MM-dd hh:mm:ss”

② iso 属性：类型为 DateTimeFormat.ISO。指定解析/格式化字段数据的 ISO 模式，包括四种：ISO.NONE（不使用） -- 默认、ISO.DATE(yyyy-MM-dd) 、ISO.TIME(hh:mm:ss.SSSZ)、 ISO.DATE_TIME(yyyy-MM-dd hh:mm:ss.SSSZ)

③ style 属性：字符串类型。通过样式指定日期时间的格式，由两位字符组成，第一位表示日期的格式，第二位表示时间的格式：S：短日期/时间格式、M：中日期/时间格式、L：长日期/时间格式、F：完整日期/时间格式、-：忽略日期或时间格式

8.7.3 数值格式化概述

1） @NumberFormat 可对类似数字类型的属性进行标注，它拥有两个互斥的属性：

① style：类型为 NumberFormat.Style。用于指定样式类型，包括三种：Style.NUMBER（正常数字类型）、Style.CURRENCY（货币类型）、Style.PERCENT（百分数类型）

② pattern：类型为 String，自定义样式，如 pattern="#,###";

```
<!-- 声明类型转换器服务 -->
<!-- <bean id="conversionService"
 class="org.springframework.context.support.ConversionServiceFactoryBean"> -->
<bean id="conversionService"
 class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <!-- 引用类型转换器 -->
            <ref bean="stringToEmployeeConverter"/>
        </set>
    </property>
</bean>
```

8.7.4 实验代码(格式化日期)

1） 页面表单

```
<!-- 解决问题：
        1.数据类型转换
        2.数据格式
        3.数据校验
        自定义类型转换器：
            将字符串转换为 Employee 对象,完成添加功能
-->
BirthDay :<input type="text" name="birthDay"/><br><br>
```

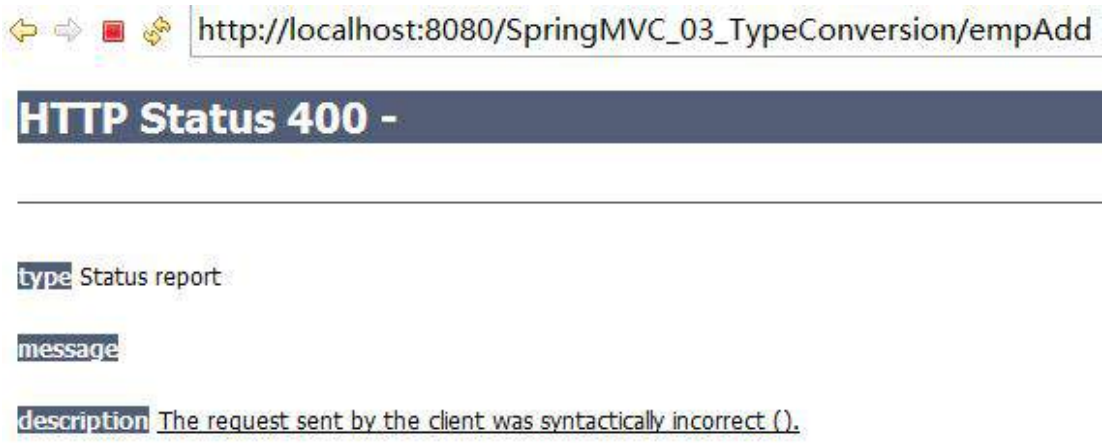
2） Employee 类增加日期对象属性

```
//关于类型转换
private Date birthDay ;
```

3） 关于格式错误（框架默认支持的格式为斜线方式。1990/09/09）

在页面上设置格式为：1990-09-09

报错：



4） 解决 400 错误：

在 Employee 类的日期属性上增加

```
@DateTimeFormat(pattern="yyyy-MM-dd")
private Date birthDay ;
```

5） 配置，配置时不能指定 conversion-service 属性，否则，依然报错 400。

用 **FormattingConversionServiceFactoryBean 替换 ConversionServiceFactoryBean 后再进行引用。**

```
<mvc:annotation-driven conversion-service="conversionService"/>
<mvc:annotation-driven />
```

8.7.5 实验代码(格式化数字)

```
Salary : <form:input path="salary"/>
```

```
@NumberFormat(pattern="#,###,###.#")
private double salary ;
```

8.7.6 获取错误消息

（后台获取错误消息，并打印）

```
//添加员工
@RequestMapping(value="/empAdd",method=RequestMethod.POST)
```

```
public String empAdd(Employee employee,BindingResult bindingResult){
    System.out.println("empAdd - employee="+employee);

    if(bindingResult.getErrorCount() > 0 ){
        System.out.println("类型转换处错误了");
        List<FieldError> fieldErrors = bindingResult.getFieldErrors();
        for(FieldError fieldError : fieldErrors){
            System.out.println(fieldError.getField() + " - " + fieldError.getDefaultMessage());
        }
    }

    employeeDao.save(employee);
    return "redirect:/empList";
}
```

```
类型转换出错误了
birthDay - Failed to convert property value of type 'java.lang.String' to required type 'java.util.Date' for property 'birthDay'; nested exception is org.springframework.core.convert.ConversionFailedException: Failed to convert from type java.lang.String to type @org.springframework.format.annotation.DateTimeFormat java.util.Date for value 's'; nested exception is java.lang.IllegalArgumentException: Unable to parse 's'
salary - Failed to convert property value of type 'java.lang.String' to required type 'double' for property 'salary'; nested exception is java.lang.NumberFormatException: For input string: "ss"
```

8.8 JSR303 数据校验

8.8.1 如何校验

1） 使用 JSR 303 验证标准
2） 加入 hibernate validator 验证框架
3） 在 SpringMVC 配置文件中增加<mvc:annotation-driven/>
4） 需要在 bean 的属性上增加对应验证的注解
5） 在目标方法 bean 类型的前面增加@Valid 注解

8.8.2 JSR 303

是 Java 为 Bean 数据合法性校验提供的标准框架，它已经包含在 JavaEE 6.0 中 .

JSR 303 (Java Specification Requests 意思是 Java 规范提案)通过在 Bean 属性上标注

类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证

注解	功能说明
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期
@Pattern(value)	被注释的元素必须符合指定的正则表达式

8.8.3 Hibernate Validator 扩展注解

Hibernate Validator 是 JSR 303 的一个参考实现，除支持所有标准的校验注解外，它还支持以下的扩展注解

注解	功能说明
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内

8.8.4 Spring MVC 数据校验

1） Spring 4.0 拥有自己独立的数据校验框架，同时支持 JSR 303 标准的校验框架。

2） Spring 在进行数据绑定时，可同时调用校验框架完成数据校验工作。**在 Spring MVC 中，可直接通过注解驱动的方式进行数据校验**

3） Spring 的 LocalValidatorFactroyBean 既实现了 Spring 的 Validator 接口，也实现了 JSR 303 的 Validator 接口。只要在 Spring 容器中定义了一个 **LocalValidatorFactoryBean**，即可将其注入到需要数据校验的 Bean 中。

4） **Spring 本身并没有提供 JSR303 的实现，所以必须将 JSR303 的实现者的 jar 包放到类路径下。**

5） **<mvc:annotation-driven/> 会默认装配好一个 LocalValidatorFactoryBean**，通过在处理方法的入参上标注 **@Valid** 注解即可让 Spring MVC 在完成数据绑定后执行数据校验的工作

6） 在已经标注了 JSR303 注解的表单/命令对象前标注一个 @Valid，Spring MVC 框架在将请求参数绑定到该入参对象后，就会调用校验框架根据注解声明的校验规则实施校验

8.8.5 实验代码

1） 添加 jar 包：

hibernate-validator-5.0.0.CR2\dist

- hibernate-validator-5.0.0.CR2.jar
- hibernate-validator-annotation-processor-5.0.0.CR2.jar

hibernate-validator-5.0.0.CR2\dist\lib\required （EL 就不需要加了）

- classmate-0.8.0.jar
- jboss-logging-3.1.1.GA.jar
- validation-api-1.1.0.CR1.jar

2） 在验证属性上增加验证注解

```
public class Employee {

private Integer id;

@NotEmpty
private String lastName;

@Email
private String email;
//1 male, 0 female
private Integer gender;

private Department department;

//关于类型转换
@Past //被标注的日期必须是一个过去的日期
@DateTimeFormat(pattern="yyyy-MM-dd")
private Date birthDay ;

@NumberFormat(pattern="#,###,###.#")
private double salary ;
}
```

3） 增加

```
//添加员工
/** 增加@Valid 注解，验证失败会报错。
 * 严重: Servlet.service() for servlet springDispatcherServlet threw exception
java.lang.NoSuchMethodError:
javax.el.ExpressionFactory.newInstance()Ljavax/el/ExpressionFactory;
 */
@RequestMapping(value="/empAdd",method=RequestMethod.POST)
public String empAdd(@Valid Employee employee,BindingResult bindingResult){
```

```
        System.out.println("empAdd - employee="+employee);

        if(bindingResult.getErrorCount() > 0 ){
            System.out.println("类型转换出错误了");
            List<FieldError> fieldErrors = bindingResult.getFieldErrors();
            for(FieldError fieldError : fieldErrors){
                System.out.println(fieldError.getField() + " - " + fieldError.getDefaultMessage());
            }
        }

        employeeDao.save(employee);
        return "redirect:/empList";
    }
```

4） 后台打印错误消息

```
v1.1

birth   typeMismatch    Failed to convert property value of type 'java.lang.String' to required type 'java.util
salary  typeMismatch    Failed to convert property value of type 'java.lang.String' to required type 'double' f

v1.2
lastName        NotEmpty        不能为空
email           Email   不是一个合法的电子邮件地址
```

5） 前台打印错误消息

```
v1.3
<form:errors path="lastName"></form:errors><br>  前台显示
```

6） 测试验证，解决 EL 表达式错误

拷贝 hibernate-validator-5.0.0.CR2\dist\lib\required 目录下的

el-api-2.2.jar、javax.el-2.2.4.jar、javax.el-api-2.2.4.jar

三个包到 Tomcat/lib 目录下，将原来的 el-api.jar 删除。重启 tomcat6

7） 如果希望验证失败，回到添加页面

```
@RequestMapping(value="/empAdd",method=RequestMethod.POST)
public String empAdd(@Valid Employee employee,BindingResult bindingResult){
    System.out.println("empAdd - employee="+employee);

    if(bindingResult.getErrorCount() > 0 ){
        System.out.println("类型转换出错误了");
        List<FieldError> fieldErrors = bindingResult.getFieldErrors();
        for(FieldError fieldError : fieldErrors){
            System.out.println(fieldError.getField() + " - " + fieldError.getDefaultMessage());
```

```
            }
            map.put("deptList",departmentDao.getDepartments());
            return "add"; //      /WEB-INF/views/add.jsp
        }
        employeeDao.save(employee);
        return "redirect:/empList";
    }
```

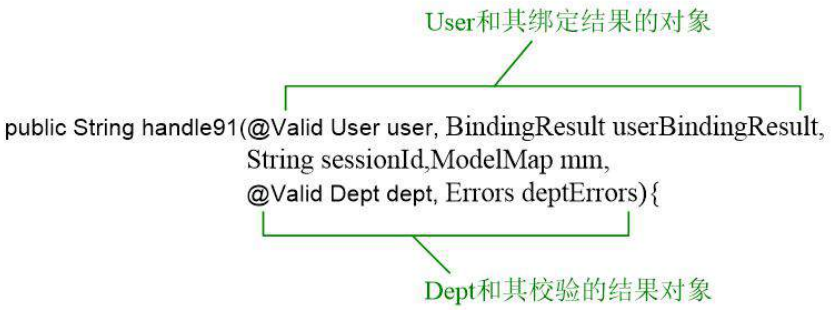
8） public interface BindingResult extends Errors

① Spring MVC 是通过对处理方法签名的规约来保存校验结果的：**前一个表单/命令对象的校验结果保存到随后的入参中，这个保存校验结果的入参必须是 BindingResult 或 Errors 类型**，这两个类都位于 org.springframework.validation 包中

② **需校验的 Bean 对象和其绑定结果对象或错误对象是成对出现的，它们之间不允许声明其他的入参**

③ Errors 接口提供了获取错误信息的方法，如 getErrorCount() 或 getFieldErrors(String field)

④ BindingResult 扩展了 Errors 接口



8.9 错误消息的显示及国际化

8.9.1 在页面上显示错误

1） Spring MVC 除了会将表单/命令对象的校验结果保存到对应的 BindingResult 或 Errors 对象中外，**还会将所有校验结果保存到 “隐含模型”**

2） 即使处理方法的签名中没有对应于表单/命令对象的结果入参，校验结果也会保存在 “隐含对象” 中。

3） 隐含模型中的所有数据最终将通过 HttpServletRequest 的属性列表暴露给 JSP 视图对象，因此在 JSP 中可以获取错误信息

4） 在 JSP 页面上可通过 **<form:errors path=“userName”>** 显示错误消息

8.9.2 示例：

1） 在表单上页面上显示所有的错误消息

```
<!-- 显示所有的错误消息 -->
<form:errors path="*"/>
```

2） 显示某一个表单域的错误消息

```
<form:errors path="lastName"/>
```

3） 有错，回到 add.jsp

```
<%@ 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">
      <!-- 显示所有的错误消息 --><form:errors path="*"/><br><br>

      LastName : <form:input path="lastName" /> <form:errors path="lastName"/> <br><br>

      Email : <form:input path="email" /><form:errors 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 : <form:radiobuttons path="gender" items="${genders}" delimiter="<br>"/>
    DeptName :
        <form:select path="department.id"
                        items="${deptList }"
                        itemLabel="departmentName"
                        itemValue="id"></form:select><br><br>


      BirthDay :<%-- <input type="text" name="birthDay"/> --%>
      <form:input path="birthDay"/><form:errors path="birthDay"/><br><br>
      Salary : <form:input path="salary"/><br><br>
       <input type="submit" value="Submit"><br><br>
 </form:form>
</body>
</html>
```

8.10 提示消息的国际化

1） 每个属性在数据绑定和数据校验发生错误时，都会生成一个对应的 FieldError 对象。

2） **当一个属性校验失败后，校验框架会为该属性生成 4 个消息代码**，这些代码以**校验注解类名为前缀**，结合 **modleAttribute、属性名及属性类型名**生成多个对应的消息代码：例如 User 类中的 password 属性标注了一个 @Pattern 注解，当该属性值不满足 @Pattern 所定义的规则时，就会产生以下 4 个错误代码：

   ① Pattern.user.password
   ② Pattern.password
   ③ Pattern.java.lang.String
   ④ Pattern

3） 当使用 Spring MVC 标签显示错误消息时， Spring MVC 会查看 WEB 上下文是否装配了对应的国际化消息，如果没有，则显示默认的错误消息，否则使用国际化消息。

4） 若数据类型转换或数据格式转换时发生错误，或该有的参数不存在，或调用处理方法时发生错误，都会在隐含模型中创建错误消息。其错误代码前缀说明如下：

   ① **required**：必要的参数不存在。如 @RequiredParam(“param1”) 标注了一个入参，但是该参数不存在
   ② **typeMismatch**：在数据绑定时，发生数据类型不匹配的问题
   ③ **methodInvocation**：Spring MVC 在调用处理方法时发生了错误

5） 注册国际化资源文件

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n"></property>
</bean>
```

8.10.1 提示消息的国际化实验

1） 定义国际化资源文件：i18n.properties

```
NotEmpty.employee.lastName=\u7528\u6237\u540D\u4E0D\u80FD\u4E3A\u7A7A
Email.employee.email=\u7535\u5B50\u90AE\u4EF6\u5730\u5740\u4E0D\u5408\u6CD5
Past.employee.birthDay=\u65E5\u671F\u5FC5\u987B\u662F\u4E00\u4E2A\u8FC7\u53BB\u7684\u65F6\u95F4

typeMismatch.employee.birthDay=\u4E0D\u662F\u4E00\u4E2A\u65E5\u671F\u6709\u6548\u683C\u5F0F
```

2） 声明国际化资源配置

```
<!-- 声明国际化资源文件 -->
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n"></property>
</bean>
```

第 9 章 处理 JSON

9.1 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

2） 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();
}
```

3） 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>
```

4） 测试

9.2 HttpMessageConverter 原理

9.2.1 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：**

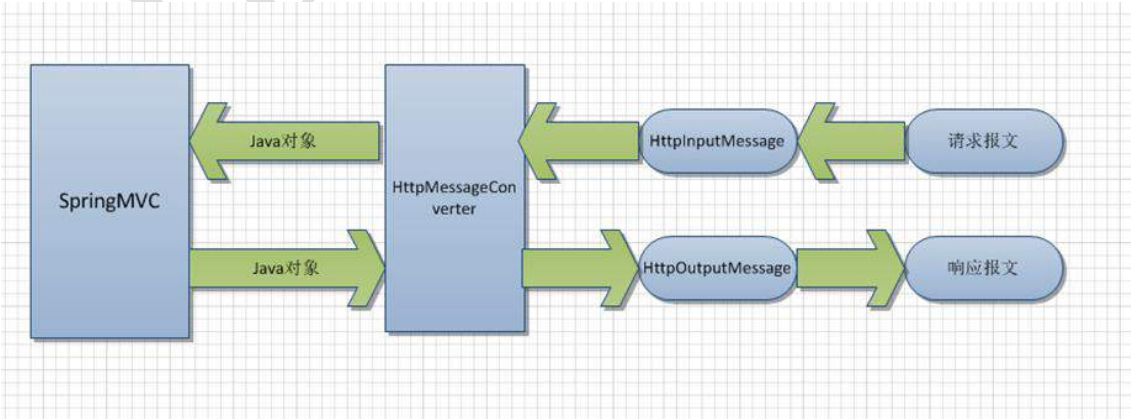
① Boolean canRead(Class<?> clazz,MediaType mediaType): 指定转换器可以读取的对象类型，即转换器是否可将请求信息转换为 clazz 类型的对象，同时指定支持 MIME 类型(text/html,applaiction/json 等)

② 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。

③ List<MediaType> getSupportMediaTypes()：该转换器支持的媒体类型。

④ T read(Class<? extends T> clazz,**HttpInputMessage** inputMessage)：将请求信息流转换为 T 类型的对象。

⑤ 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;	package org.springframework.http;

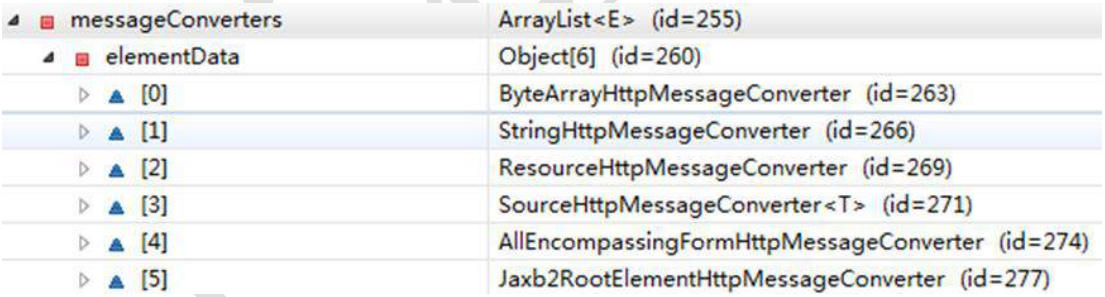
```
import java.io.IOException;
import java.io.InputStream;

public interface HttpInputMessage extends HttpMessage {

InputStream getBody() throws IOException;

}
```

```
import java.io.IOException;
import java.io.OutputStream;

public interface HttpOutputMessage extends HttpMessage {

OutputStream getBody() throws IOException;

}
```

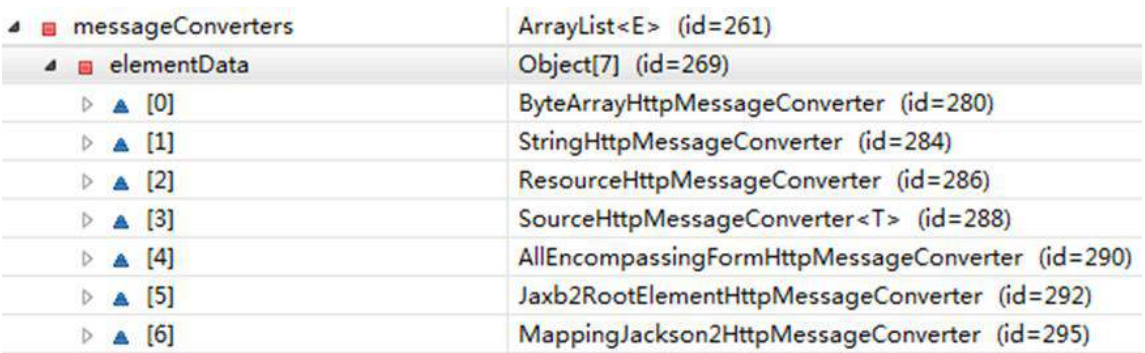
实现类	功能说明
StringHttpMessageConverter	将请求信息转换为字符串
FormHttpMessageConverter	将表单数据读取到 MultiValueMap 中
XmlAwareFormHttpMessageConverter	扩展于 FormHttpMessageConverter, 如果部分表单属性是 XML 数据，可用该转换器进行读取
ResourceHttpMessageConverter	读写 org.springframework.core.io.Resource 对象
BufferedImageHttpMessageConverter	读写 BufferedImage 对象
ByteArrayHttpMessageConverter	读写二进制数据
SourceHttpMessageConverter	读写 javax.xml.transform.Source 类型的数据
MarshallingHttpMessageConverter	通过 Spring 的 org.springframework.xml.Marshaller 和 Unmarshaller 读写 XML 消息
Jaxb2RootElemengHttpMessageConverter	通过 JAXB2 读写 XML 消息, 将请求消息转换到标注 XmlRootElement 和 XxmlType 直接的类中
MappingJacksonHttpMessageConverter	利用 Jackson 开源包的 ObjectMapper 读写 JSON 数据
RssChannelHttpMessageConverter	能够读写 RSS 种子消息
AtomFeedHttpMessageConverter	和 RssChannelHttpMessageConverter 能够读写 RSS 种子消息

3） DispatcherServlet 默认装配 RequestMappingHandlerAdapter ，

而 RequestMappingHandlerAdapter 默认装配如下 HttpMessageConverter：

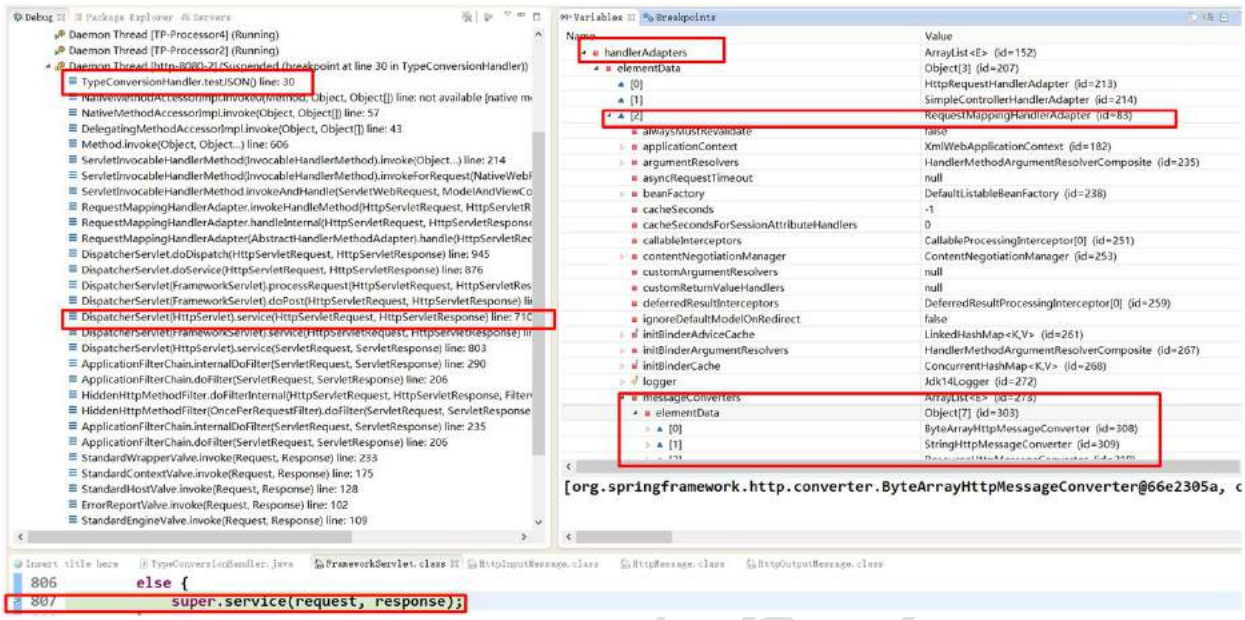


4） 加入 jackson jar 包后， RequestMappingHandlerAdapter 装配的 HttpMessageConverter 如下：



默认情况下数组长度是 6 个：增加了 jackson 的包，后多个一个

MappingJackson2HttpMessageConverter



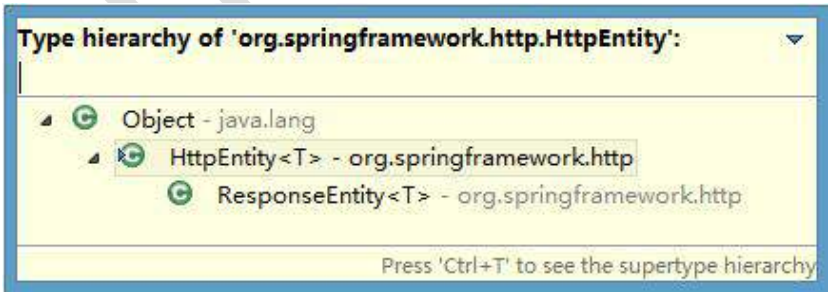
9.3 使用 HttpMessageConverter

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

3） @RequestBody 和 @ResponseBody 不需要成对出现

4） Content-Disposition：attachment; filename=abc.pdf



5） @RequestBody @ResponseBody

@RequestBody、@ResponseBody 示例

```
@ResponseBody
@RequestMapping("/handle15")
public byte[] handle15() throws IOException{
    Resource resource = new ClassPathResource("/Lighthouse.jpg");
    byte [] fileData = FileCopyUtils.copyToByteArray(resource.getInputStream());
    return fileData;
}

@RequestMapping(value="/handle14", method=RequestMethod.POST)
public String handle14(@RequestBody String requestBody){
    System.out.println(requestBody);
    return "success";
}
```

messageConverters	ArrayList<E> (id=261)
elementData	Object[7] (id=269)
[0]	ByteArrayHttpMessageConverter (id=280)
[1]	StringHttpMessageConverter (id=284)
[2]	ResourceHttpMessageConverter (id=286)
[3]	SourceHttpMessageConverter<T> (id=288)
[4]	AllEncompassingFormHttpMessageConverter (id=290)
[5]	Jaxb2RootElementHttpMessageConverter (id=292)
[6]	MappingJackson2HttpMessageConverter (id=295)

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
}
```

6） HttpEntity ResponseEntity

HttpEntity、ResponseEntity 示例

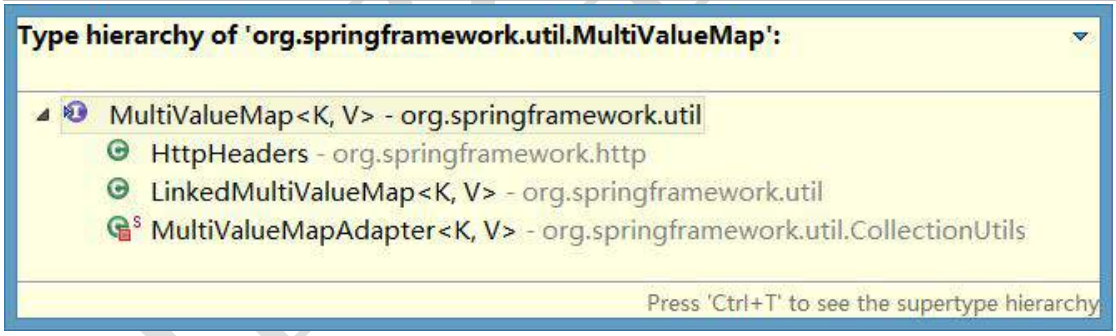
```
@RequestMapping("/handle16")
public String handle16(HttpEntity<String> entity){
    System.out.println(entity.getHeaders().getContentLength());
    return "success";
}

@RequestMapping("/handle17")
public ResponseEntity<byte[]> handle17() throws IOException{
    Resource resource = new ClassPathResource("/Lighthouse.jpg");
    byte [] fileData = FileCopyUtils.copyToByteArray(resource.getInputStream());
    ResponseEntity<byte[]> responseEntity =
            new ResponseEntity<byte[]>(fileData, HttpStatus.OK);
    return 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 ;
}
```



② 源码参考

- HttpHeaders
- HttpStatus

第 10 章 国际化【了解】

10.1 国际化_页面中获取国际化资源信息

1） 在页面上能够根据浏 览器语言设置的情况对文本，时间，数值进行本地化处理

2） 可以在 bean 中获取国际化资源文件 Locale 对应的消息
3） 可以通过超链接切换 Locale，而不再依赖于浏览器的语言设置情况

10.1.1 实现

1） 使用 JSTL 的 fmt 标签
2） 在 bean 中注入 ResourceBundleMessageSource 的实例，使用其对应的 getMessage 方法即可
3） 配置 LocalResolver 和 LocaleChangeInterceptor
4） 实验代码

① 定义国际化资源文件

i18n.properties	i18n_zh_CN.properties	i18n_en_US.properties
i18n.user=User i18n.password=Password		

② 声明国际化资源文件

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n"></property>
</bean>
```

③ 配置视图控制器

```
<!-- 测试国际化 -->
<mvc:view-controller path="/testI18N" view-name="i18n"/>
<mvc:view-controller path="/testI18N2" view-name="i18n2"/>
```

④ 主页面链接

index.jsp

```
<a href="testI18N">test I18N</a>
```

⑤ 页面国际化

/views/i18n.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<!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>
```

/views/i18n2.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<!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> <fmt:message key="i18n.user"/> test I18N2 </body> </html>	<body> <fmt:message key="i18n.password"/> test I18N </body> </html>

10.2 国际化_控制器中获取国际化资源信息

1） 屏蔽视图控制器，执行控制器中方法处理请求

```
<!-- 测试国际化
<mvc:view-controller path="/testI18N" view-name="i18n"/>
 -->
<mvc:view-controller path="/testI18N2" view-name="i18n2"/>
```

2） 控制器方法

```
package com.atguigu.springmvc.crud.handlers;

import java.util.Locale;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class I18NHandler {
    @Autowired
    private ResourceBundleMessageSource messageSource; //从 IOC 容器中获取 bean 对象，进行依赖注入

    @RequestMapping("/testI18N")
    public String testI18N(Locale locale){
        System.out.println(locale);
        String user = messageSource.getMessage("i18n.user", null, locale);
        System.out.println("i18n.user="+user);
        return "i18n";
    }
}
```

10.3 国际化_通过超链接切换 Locale

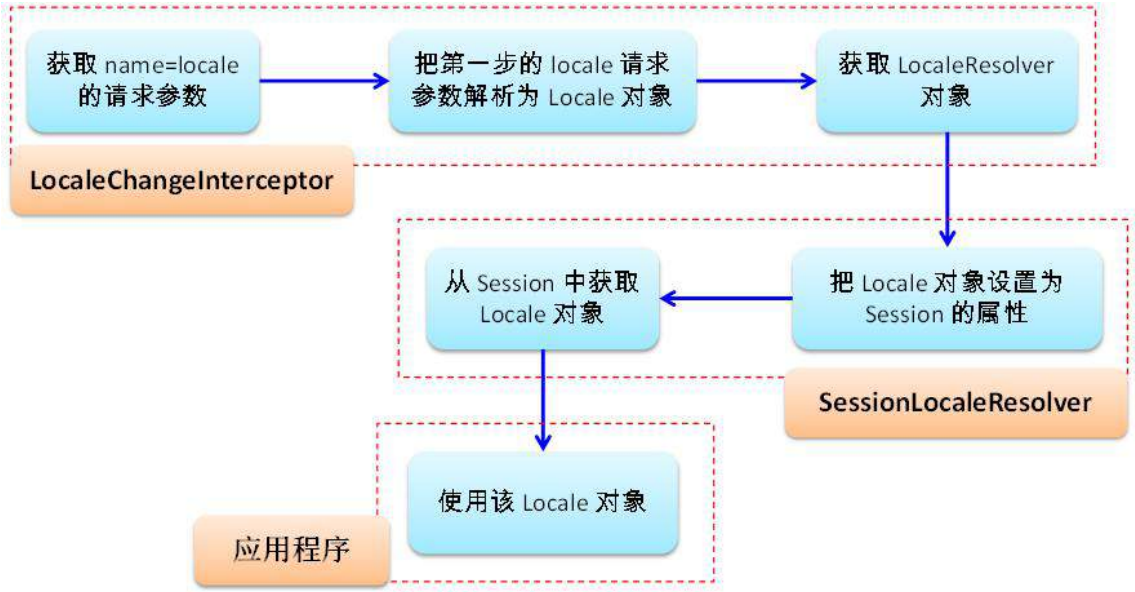
1） 国际化概述

① 默认情况下，SpringMVC 根据 **Accept-Language** 参数判断客户端的本地化类型。

② 当接受到请求时，SpringMVC 会在上下文中查找一个本地化解析器（**LocalResolver**），找到后使用它获取请求所对应的本地化类型信息。

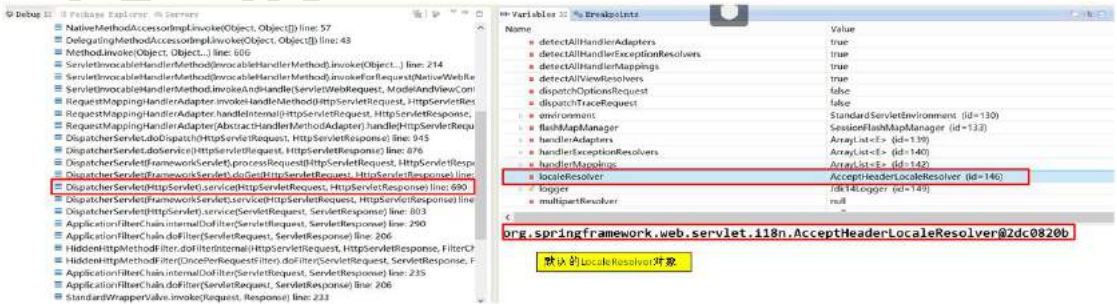
③ SpringMVC 还允许装配一个**动态更改本地化类型的拦截器**，这样通过指定一个请求参数就可以控制单个请求的本地化类型。

④ SessionLocaleResolver & LocaleChangeInterceptor 工作原理



2） 本地化解析器和本地化拦截器

① **AcceptHeaderLocaleResolver**：**根据 HTTP 请求头的 Accept-Language 参数确定本地化类型**，如果没有显式定义本地化解析器，SpringMVC 使用该解析器。




```
public class AcceptHeaderLocaleResolver implements LocaleResolver {
@Override
public Locale resolveLocale(HttpServletRequest request) {
return request.getLocale();
}
@Override
```

```
public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {
throw new UnsupportedOperationException(
"Cannot change HTTP accept header - use a different locale resolution strategy");
}
}
```

② **CookieLocaleResolver**：根据指定的 Cookie 值确定本地化类型

③ **SessionLocaleResolver**：根据 Session 中特定的属性确定本地化类型

④ **LocaleChangeInterceptor**：从请求参数中获取本次请求对应的本地化类型。

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResolver"></bean>
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"></bean>
</mvc:interceptors>
```

10.3.1 实验代码：实现中英文语言切换

1） 配置 SessionLocaleResolver 替换 **AcceptHeaderLocaleResolver 完成中英文切换**

```
<!-- 配置 SessionLocaleResolver 对象
id 必须是"localeResolver",否则，会报错误：
<bean id="sessionLocaleResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"></bean>
 -->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"></bean>
```

```
HTTP Status 500 -

type Exception report
message
description The server encountered an internal error () that prevented it from fulfilling this request.
exception
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is java.lang.UnsupportedOperationException: Cannot change HTTP accept header - use a different locale resolution strategy
        org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:943)
        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.UnsupportedOperationException: Cannot change HTTP accept header - use a different locale resolution strategy
    .....
```

2） 配置

```
<!-- 配置 LocaleChangeInterceptor 拦截器 -->
<mvc:interceptors>
    <bean id="localeChangeInterceptor"
          class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"></bean>
</mvc:interceptors>
```

3） 页面链接

/views/i18n.jsp 增加页面链接

```
<!-- 切换语言 -->
<a href="testI18N?locale=zh_CN">中文</a> | <a href="testI18N?locale=en_US">英语</a>
```

10.3.2 切换语言的原理

切换语言的参数名称必须：locale

```
public class LocaleChangeInterceptor extends HandlerInterceptorAdapter {

    /**
     * Default name of the locale specification parameter: "locale".
     */
    public static final String DEFAULT_PARAM_NAME = "locale";

    private String paramName = DEFAULT_PARAM_NAME;
```

第 11 章 文件上传

11.1 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

```
Insert title here   MultipartResolver.class
87  * @see org.springframework.web.multipart.support.StringMultipartFileEditor
88  * @see org.springframework.web.servlet.DispatcherServlet
89  */
90 public interface MultipartResolver {
91
92     /**
93      * Determine if the g
94      * <p>Will typically                                              ctually
95      * accepted requests                                              entation.
96      * @param request the
97      * @return whether th
98      */
99     boolean isMultipart(HttpServletRequest request);
100
101    /**
102     * Parse the given HTTP request into multipart files and parameters,
103     * and wrap the request inside a
104     * {@link org.springframework.web.multipart.MultipartHttpServletRequest} object
105     * that provides access to file descriptors and makes contained
106     * parameters accessible via the standard ServletRequest methods.
107     * @param request the servlet request to wrap (must be of a multipart content type)
108     * @return the wrapped servlet request
109     * @throws MultipartException if the servlet request is not multipart, or if
110     * implementation-specific problems are encountered (such as exceeding file size limits)
111     * @see MultipartHttpServletRequest#getFile
112     * @see MultipartHttpServletRequest#getFileNames
113     * @see MultipartHttpServletRequest#getFileMap
114     * @see javax.servlet.http.HttpServletRequest#getParameter
115     * @see javax.servlet.http.HttpServletRequest#getParameterNames
116     * @see javax.servlet.http.HttpServletRequest#getParameterMap
117     */
118    MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws MultipartExcepti
```

Type hierarchy of 'org.springframework.web.multipart.MultipartResolver':
MultipartResolver - org.springframework.web.multipart
CommonsMultipartResolver - org.springframework.web.multipart.commons
StandardServletMultipartResolver - org.springframework.web.multipart.support
Press 'Ctrl+T' to see the supertype hierarchy

4） 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。

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
   <property name="defaultEncoding" value="UTF-8"></property>
   <property name="maxUploadSize" value="5242880"></property>
</bean>
```

11.2 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

2） 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>
```

3） 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>
```

4） 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
    }
}
```

11.3 思考多个文件上传？

```
@RequestMapping(value="/testUpload",method=RequestMethod.POST)
public String testUpload(@RequestParam("file") MultipartFile[] file) throws IllegalStateExcepti
{
    System.out.println("-----testUpload: ");
    for (MultipartFile multipartFile : file)
    {
        if(!multipartFile.isEmpty())
        {
            multipartFile.transferTo(new File("D:\\44\\"+multipartFile.getOriginalFilename()));
        }
    }
    return "ok";
}
```

第 12 章 拦截器

12.1 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，**自定义的拦截器必须实现 HandlerInterceptor 接口**

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。

12.2 实验代码(单个拦截器)

1） 自定义拦截器类

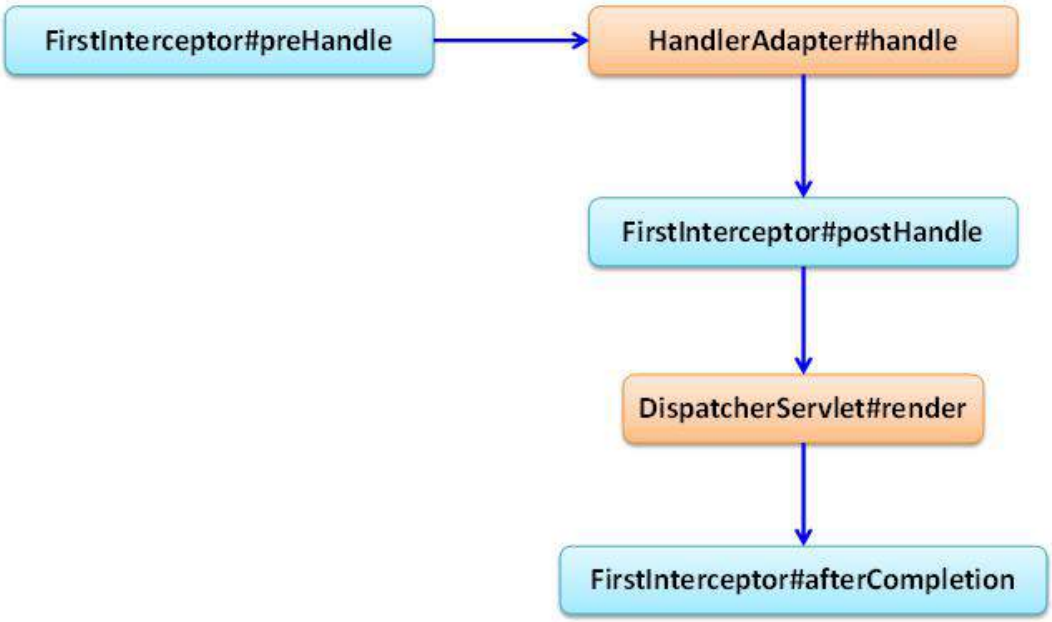
```
package com.atguigu.springmvc.interceptors;

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;
	}
}
```

2） 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>
```

3） 断点调试拦截器执行流程

4） 拦截器方法执行顺序（小总结）



12.3 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)
```

```
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;
}

}
```

```
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;
}

}
```

2） 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

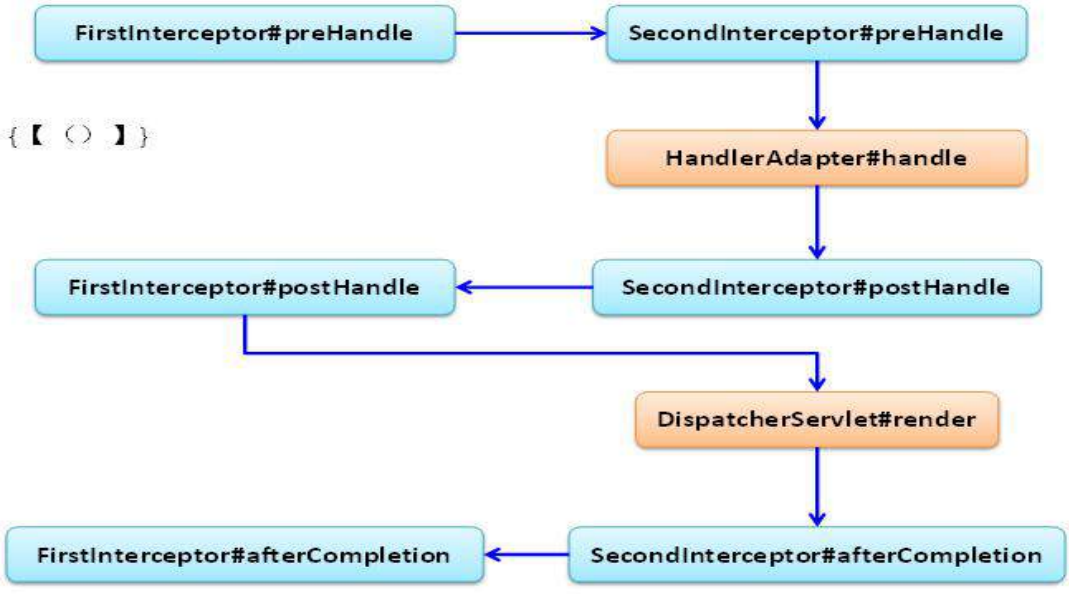
true,false com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle com.atguigu.springmvc.interceptors.SecondHandlerInterceptor - preHandle com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - afterCompletion
false,true
com.atguigu.springmvc.interceptors.**FirstHandlerInterceptor** - preHandle

12.4 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
```

2） 执行顺序图解



3） 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];
```

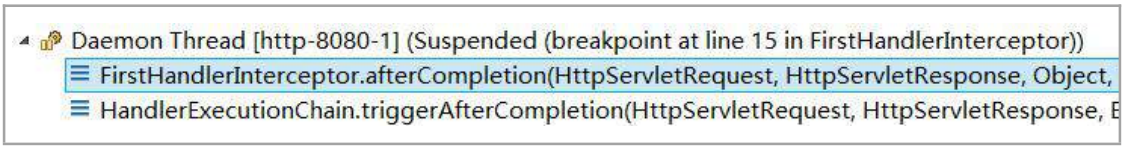
```
                if (!interceptor.preHandle(request, response, this.handler)) {
                    triggerAfterCompletion(request, response, null);
                    return false;
                }
                this.interceptorIndex = i;
            }
        }
        return true;
}
```

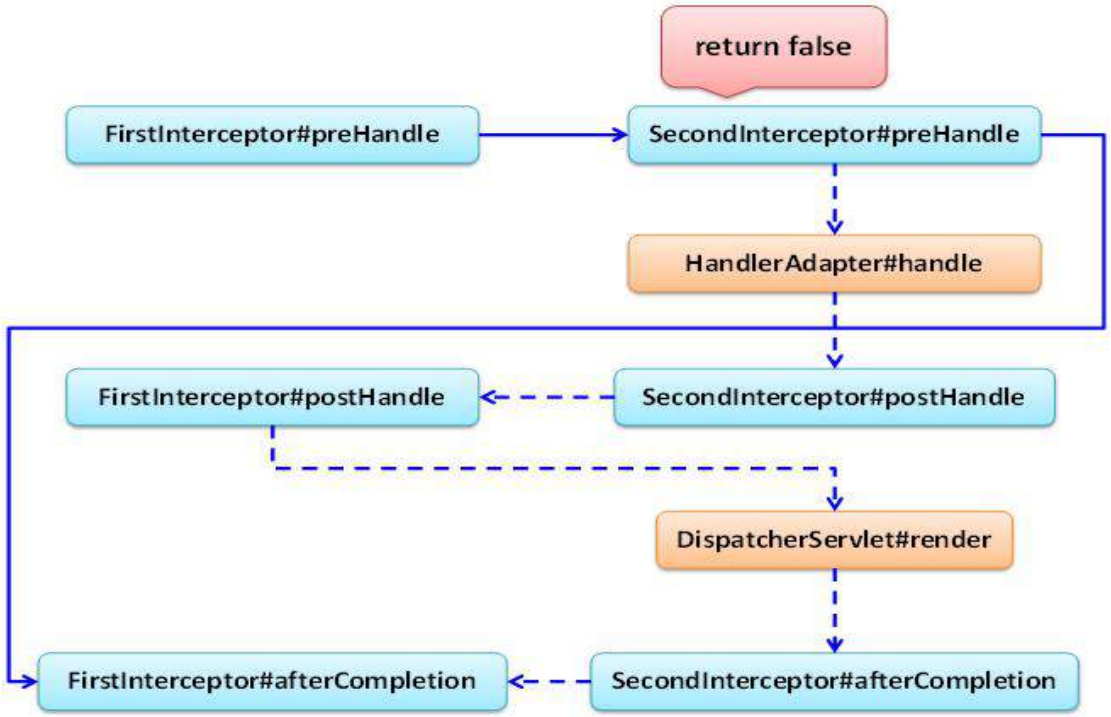
Daemon Thread [http-8080-1] (Suspended (breakpoint at line 27 in FirstHandlerIntercep
FirstHandlerInterceptor.preHandle(HttpServletRequest, HttpServletResponse, Objec
HandlerExecutionChain.applyPreHandle(HttpServletRequest, HttpServletResponse) li

```
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);
        }
}
```

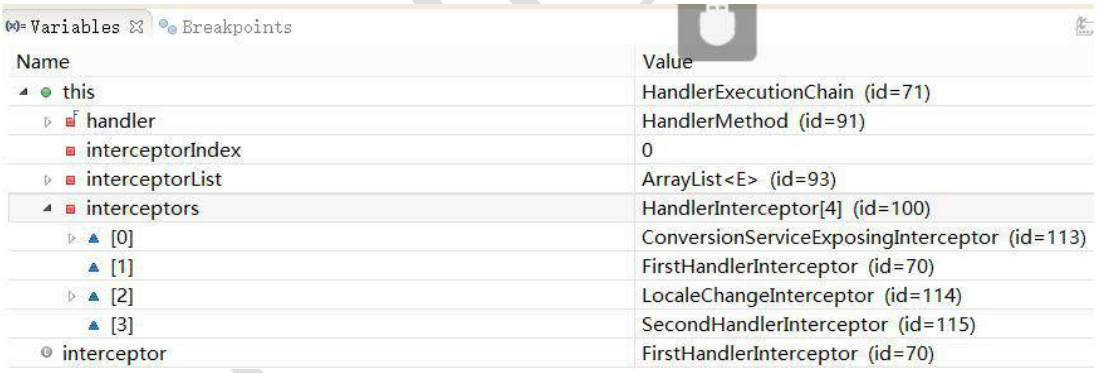
Daemon Thread [http-8080-1] (Suspended (breakpoint at line 21 in FirstHandlerInterceptor))
FirstHandlerInterceptor.postHandle(HttpServletRequest, HttpServletResponse, Object, ModelAnd
HandlerExecutionChain.applyPostHandle(HttpServletRequest, HttpServletResponse, ModelAndVi

```
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);
            }
        }
}
```





4） 源码分析：分析 interceptorIndex 的值情况



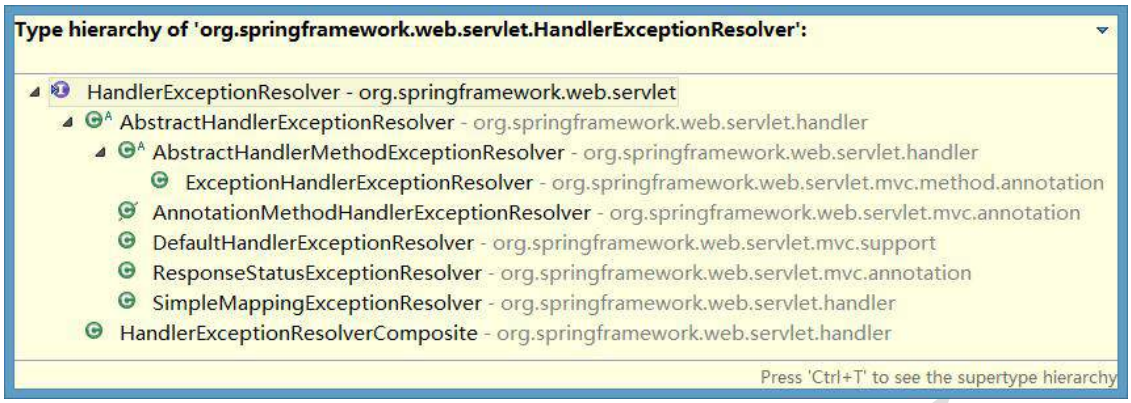
第 13 章 异常处理

13.1 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。

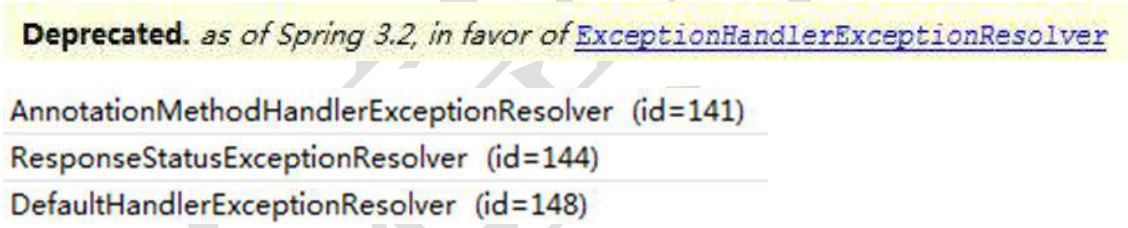
2） SpringMVC 提供的 HandlerExceptionResolver 的实现类



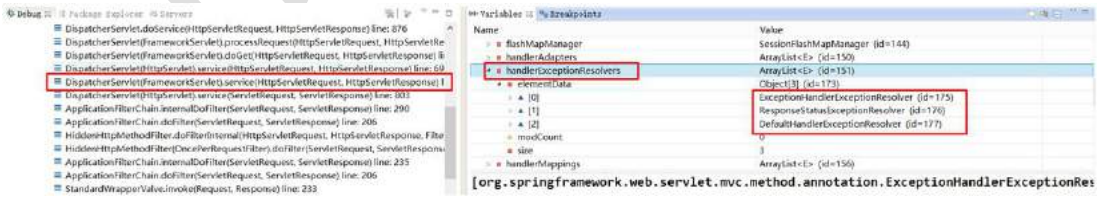
13.2 HandlerExceptionResolver

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：



13.3 ExceptionHandlerExceptionResolver

1） 主要处理 Handler 中用 **@ExceptionHandler** 注解定义的方法。

2） @ExceptionHandler 注解定义的方法**优先级问题**：例如发生的是 NullPointerException，但是声明的异常有 RuntimeException 和 Exception，此候会根据异常的最近继承关系找到继承深度最浅的那个 @ExceptionHandler 注解方法，即标记了 RuntimeException 的

方法

3） ExceptionHandlerMethodResolver 内部若找不到@ExceptionHandler 注解的话，会找 @ControllerAdvice 中的@ExceptionHandler 注解方法

13.3.1 实验代码

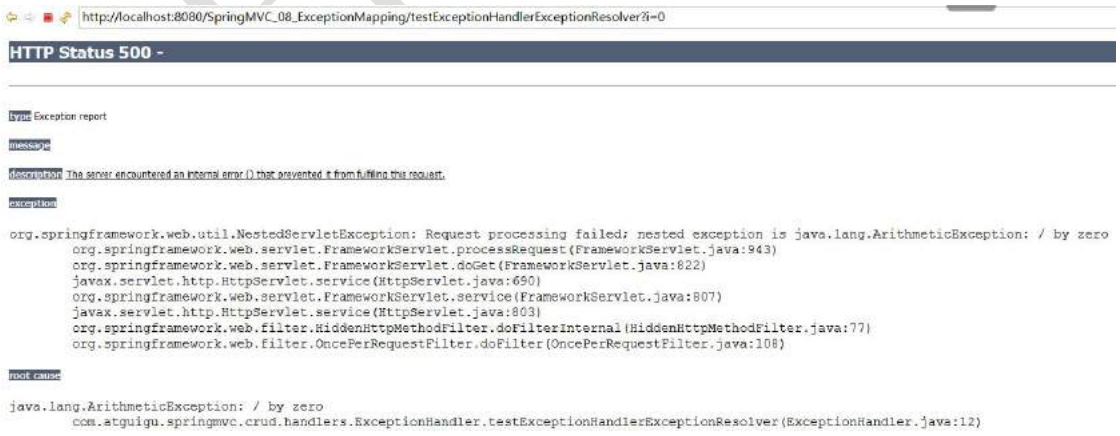
1） 页面链接

```
<a href="testExceptionHandlerExceptionResolver?i=1">testExceptionHandlerExceptionResolver</a>
```

2） 控制器方法

```
package com.atguigu.springmvc.crud.handlers;

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

@Controller
public class ExceptionHandler {
    @RequestMapping("/testExceptionHandlerExceptionResolver")
    public String testExceptionHandlerExceptionResolver(@RequestParam("i") int i){
        System.out.println("10/"+i+"="+(10/i));
        return "success";
    }
}
```

3） 测试，出现异常情况



处理异常，跳转到 error.jsp

4） 在控制器中增加处理异常的方法

```
@ExceptionHandler(value={java.lang.ArithmeticException.class})
public String handleException(Exception ex){
    System.out.println("出现异常啦☹"+ex);
```

```
    return "error";
}
```

5） 增加 error.jsp

```
<h3>Error Page</h3>
```

13.3.2 如何将异常对象从控制器携带给页面，做异常信息的获取

① 异常对象不能通过 Map 集合方式传递给成功页面。（*Map 不可以*）

```
// @ExceptionHandler(value={java.lang.ArithmeticException.class})
public String handleException(Exception ex,Map<String,Object> map){
    System.out.println("出现异常啦:"+ex);
    map.put("exception",ex);
    return "error";
}
```

② 可以通过 ModelAndView 将异常对象传递给成功页面上

```
@ExceptionHandler(value={java.lang.ArithmeticException.class})
public ModelAndView handleException(Exception ex){
    System.out.println("出现异常啦:"+ex);
    ModelAndView mv = new ModelAndView("error");
    mv.addObject("exception", ex);
    return mv;
}
```

13.3.3 匹配异常类型，执行顺序问题

```
@ExceptionHandler(value={java.lang.ArithmeticException.class})
public ModelAndView handleException(Exception ex){
    System.out.println("出现异常啦:"+ex);
    ModelAndView mv = new ModelAndView("error");
    mv.addObject("exception", ex);
```

```
    return mv;
}

@ExceptionHandler(value={java.lang.RuntimeException.class})
public ModelAndView handleException2(Exception ex){
    System.out.println("RuntimeException-出现异常啦:"+ex);
    ModelAndView mv = new ModelAndView("error");
    mv.addObject("exception", ex);
    return mv;
}
```

13.3.4 公共的处理异常的类@ControllerAdvice

1） 定义公共的处理异常的类

```
package com.atguigu.springmvc.exceptionAdvice;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.servlet.ModelAndView;

@ControllerAdvice
public class ExceptionAdviceHandler {

    /*
    @ExceptionHandler(value={java.lang.ArithmeticException.class})
    public ModelAndView handleException(Exception ex){

        System.out.println("出现异常啦:"+ex);
        ModelAndView mv = new ModelAndView("error");
        mv.addObject("exception", ex);

        return mv;
    }*/

    @ExceptionHandler(value={java.lang.RuntimeException.class})
    public ModelAndView handleException2(Exception ex){

        System.out.println("RuntimeException-出现异常啦:"+ex);
        ModelAndView mv = new ModelAndView("error");
        mv.addObject("exception", ex); //error.jsp      -> ${requestScope.exception}
```

```
        return mv;
    }
}
```

13.4 异常处理_ResponseStatusExceptionResolver

1） 在异常及异常父类中找到 @ResponseStatus 注解，然后使用这个注解的属性进行处理。
2） 定义一个 @ResponseStatus 注解修饰的异常类
3） 若在处理器方法中抛出了上述异常：若 ExceptionHandlerExceptionResolver 不解析上述异常。由于触发的异常 UnauthorizedException 带有@ResponseStatus 注解。因此会被 ResponseStatusExceptionResolver 解析到。最后响应 HttpStatus.UNAUTHORIZED 代码给客户端。HttpStatus.UNAUTHORIZED 代表响应码 401，无权限。 关于其他的响应码请参考 HttpStatus 枚举类型源码。

org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver

Implementation of the HandlerExceptionResolver interface that uses the @ResponseStatus annotation to map exceptions to HTTP status codes.
This exception resolver is enabled by default in the org.springframework.web.servlet.DispatcherServlet.

13.4.1 实验代码

1） 页面链接

```
<a href="testResponseStatusExceptionResolver?i=10">testResponseStatusExceptionResolver</a>
```

2） 自定义异常类

```
package com.atguigu.springmvc.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

/**
 * 自定义异常类
HttpStatus.FORBIDDEN 不允许的，禁用的
 */
@ResponseStatus(value=HttpStatus.FORBIDDEN,reason="用户名称和密码不匹配")
public class UsernameNotMatchPasswordException extends RuntimeException{}
```

3） 控制器方法

```
@RequestMapping(value="/testResponseStatusExceptionResolver")
public String testResponseStatusExceptionResolver(@RequestParam("i") int i){
    if(i==13){
        throw new UsernameNotMatchPasswordException();
    }
    System.out.println("testResponseStatusExceptionResolver...");
    return "success";
}
```

4） 出现的错误消息

1） 没使用注解时：**@ResponseStatus(value=HttpStatus.FORBIDDEN,reason="用户名称和密码不匹配")**

2） 使用注解时：**@ResponseStatus(value=HttpStatus.FORBIDDEN,reason="用户名称和密码不匹配")**

3） 测试在方法上使用注解

```
@ResponseStatus(value=HttpStatus.NOT_FOUND,reason="测试方法上设置响应状态码")
@RequestMapping(value="/testResponseStatusExceptionResolver")
public String testResponseStatusExceptionResolver(@RequestParam("i") int i){
    if(i==13){
        throw new UsernameNotMatchPasswordException();
    }
    System.out.println("testResponseStatusExceptionResolver...");
    return "success";
}
```

4） ResponseStatus

```
package org.springframework.web.bind.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.http.HttpStatus;

/**
 * Marks a method or exception class with the status code and reason that should be returned. The status code
is applied
 * to the HTTP response when the handler method is invoked, or whenever said exception is thrown.
 *
 * @author Arjen Poutsma
 * @see org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver
 * @since 3.0
 */
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ResponseStatus {

	HttpStatus value();

	String reason() default "";

}
```

5） HttpStatus

13.5 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.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

Default implementation of the 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.

13.5.1 实验代码

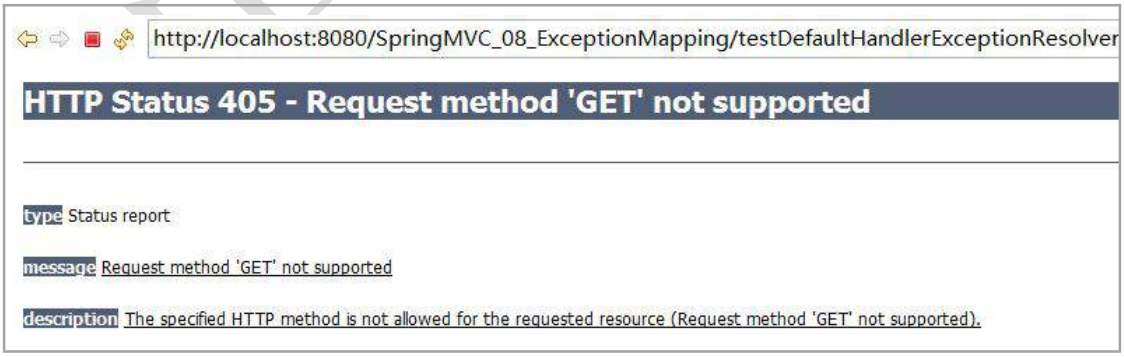
1） 增加页面链接：GET 请求

```
<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";
}
```

2） 出现异常错误



3） 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;
}
```

13.6 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

```
<bean id="simpleMappingExceptionResolver"
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="java.lang.ArithmeticException">error</prop>
    </props>
  </property>
</bean>
```

13.6.1 实验代码

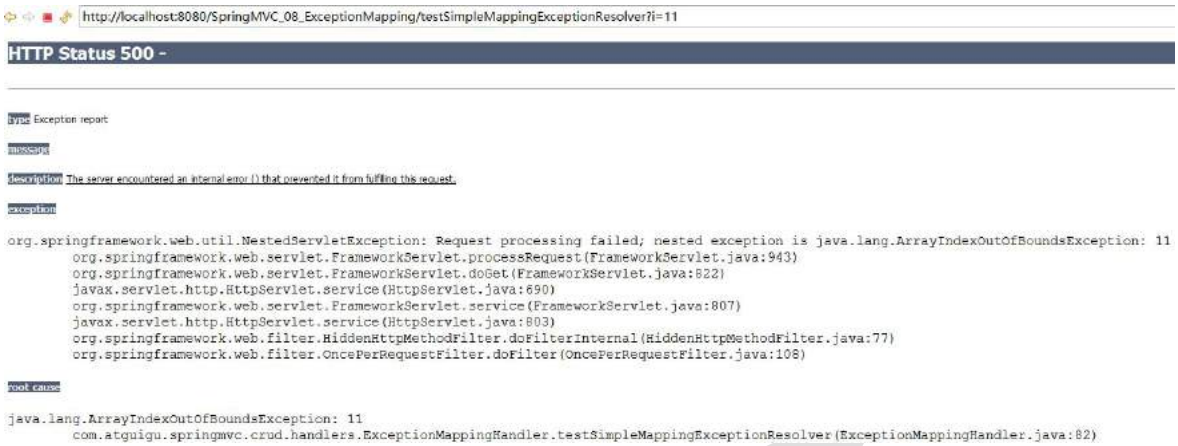
1） 增加页面链接

```
<a href="testSimpleMappingExceptionResolver?i=1">testSimpleMappingExceptionResolver</a>
```

2） 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";
}
```

3） 出现异常情况：参数 i 的值大于 10



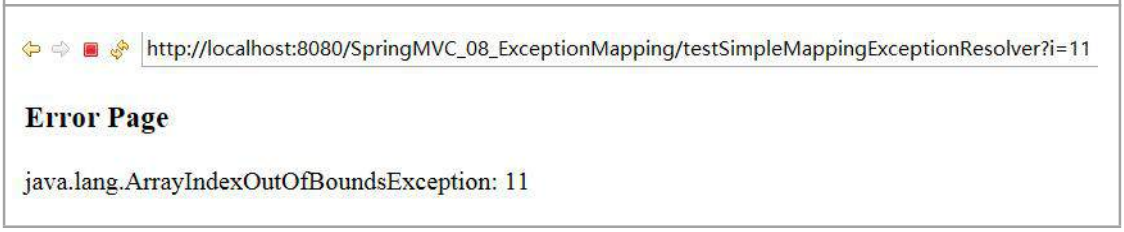
4） 配置异常解析器:自动将异常对象信息，存放到 request 范围内

```
<!-- 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>
```



5） 源码分析

SimpleMappingExceptionResolver L187 L339

```
public class SimpleMappingExceptionResolver extends AbstractHandlerExceptionResolver {

    /** The default name of the exception attribute: "exception". */
    public static final String DEFAULT_EXCEPTION_ATTRIBUTE = "exception";

    private Properties exceptionMappings;

    private Class<?>[] excludedExceptions;

    private String defaultErrorView;

    private Integer defaultStatusCode;

    private Map<String, Integer> statusCodes = new HashMap<String, Integer>();

    private String exceptionAttribute = DEFAULT_EXCEPTION_ATTRIBUTE;
```

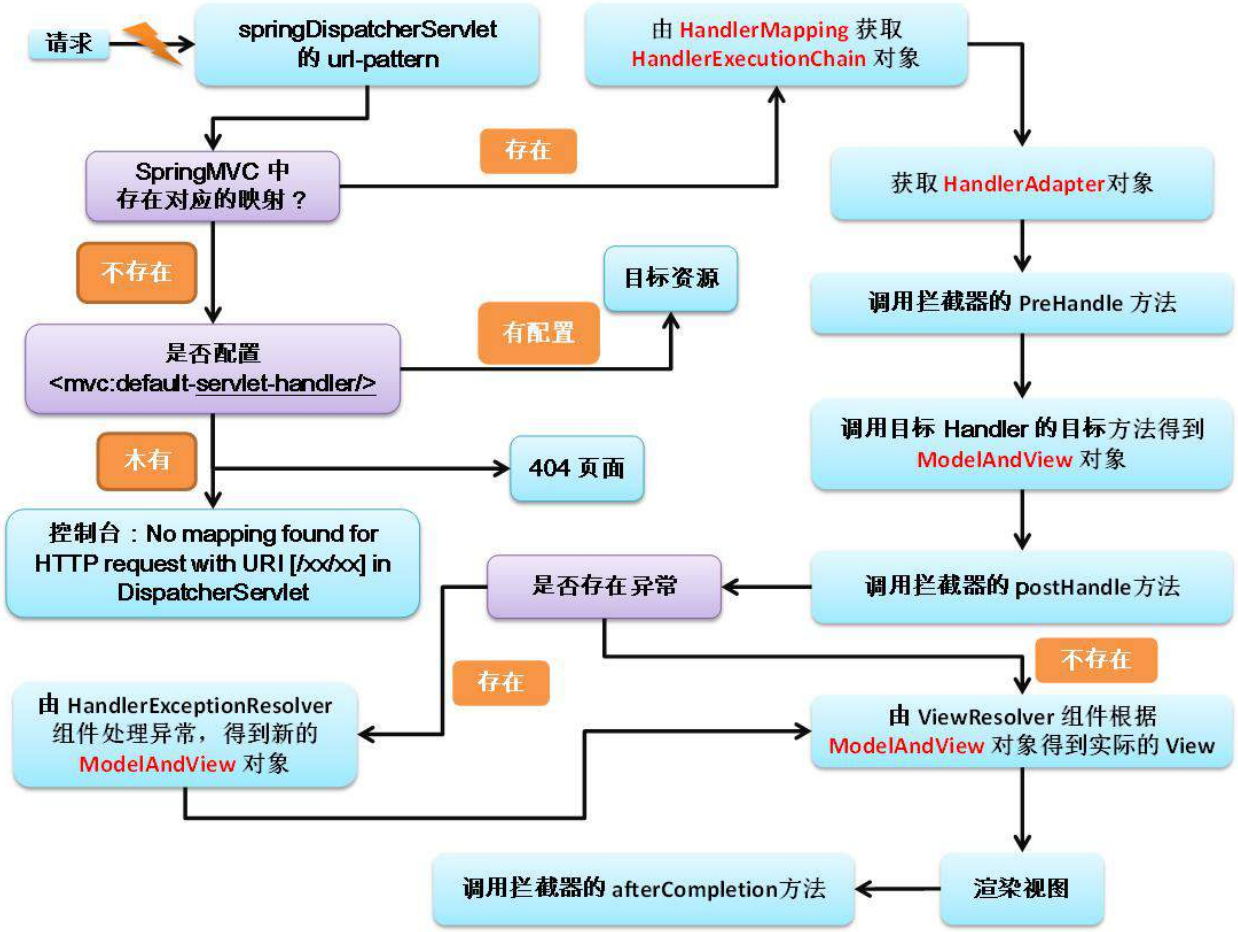
```
@Override
protected ModelAndView doResolveException(HttpServletRequest request,
  HttpServletResponse response,Object handler, Exception ex) {

      // 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 14 章 运行流程图解

14.1 流程图



14.2 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

3） 根据该 URI，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回；
4） DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter。
5） 如果成功获得 HandlerAdapter 后，此时将开始执行拦截器的 preHandler(...)方法【正向】
6） 提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（Controller)方法，处理请求。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：
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
7） Handler 执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象；
8） 此时将开始执行拦截器的 postHandle(...)方法【逆向】
9） 根据返回的 ModelAndView（此时会判断是否存在异常：如果存在异常，则执行 HandlerExceptionResolver 进行异常处理）选择一个适合的 ViewResolver（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
10） 在返回给客户端时需要执行拦截器的 AfterCompletion 方法【逆向】
11） 将渲染结果返回给客户端

14.3 源码解析

14.3.1 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
```

2） 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>
```

3） 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>
```

14.3.2 完成 HelloWorld

1） 页面链接

```
<a href="helloworld">Hello World</a>
```

2） 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";
    }
}
```

3） 成功页面：/views/success.jsp

```
<h3>Success Page</h3>
```

14.3.3 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

① http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2

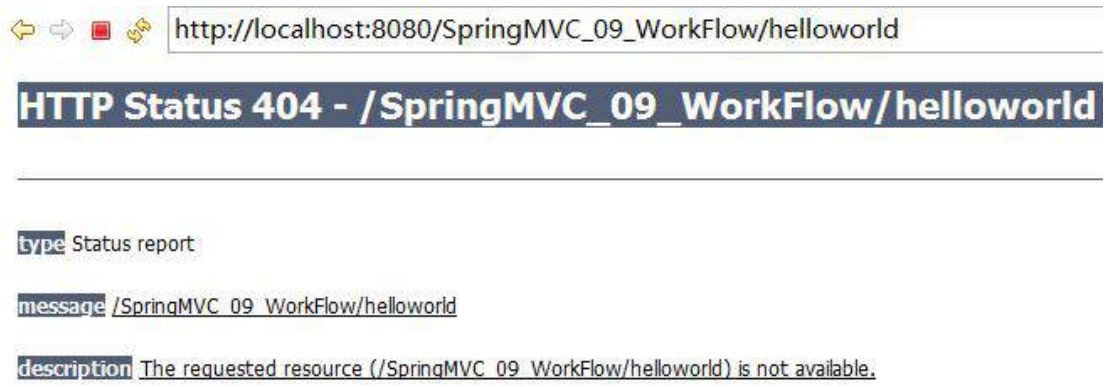
```
四月 20, 2016 11:53:19 上午 org.springframework.web.servlet.PageNotFound noHandlerFound
警 告 : No mapping found for HTTP request with URI [/SpringMVC_09_WorkFlow/helloworld2] in DispatcherServlet with name 'springDispatcherServlet'
```

② 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'
```

3） 配置<mvc:default-servlet-handler/>，测试，会去查找目标资源

4） 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。

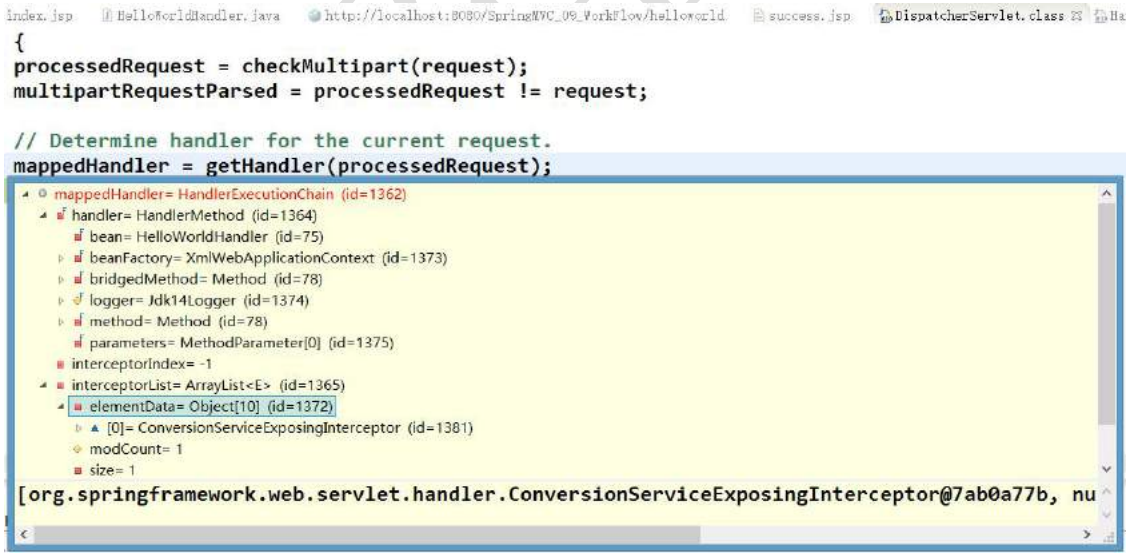


14.3.4 Debug 流程分析

1） HandlerExecutionChain mappedHandler；包含了拦截器和处理器方法；
DispatcherServlet L902 916

org.springframework.web.servlet.HandlerExecutionChain

Handler execution chain, consisting of handler object and any handler interceptors. Returned by HandlerMapping's HandlerMapping.getHandler method.



2） HandlerMapping

org.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 and org.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.

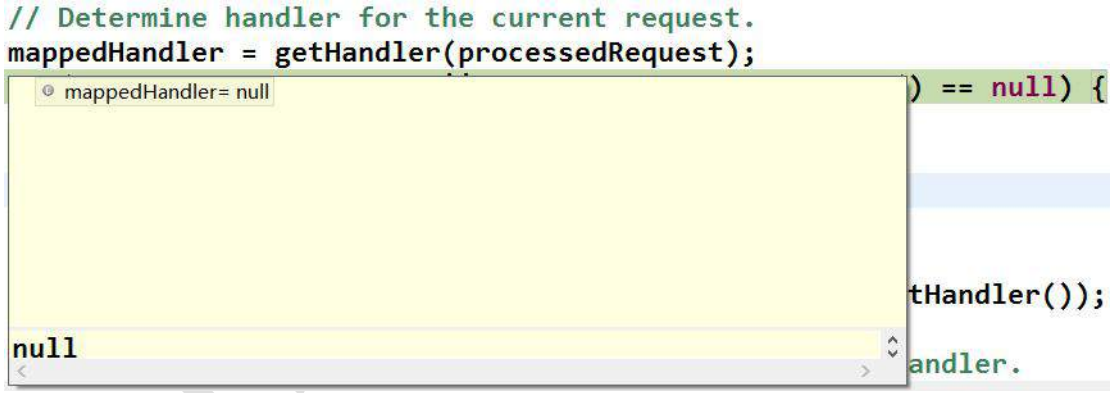
HandlerMapping implementations can support mapped interceptors but do not have to. A handler will always be wrapped in a HandlerExecutionChain instance, optionally accompanied by some HandlerInterceptor instances. The DispatcherServlet will first call each HandlerInterceptor's preHandle method in the 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 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.

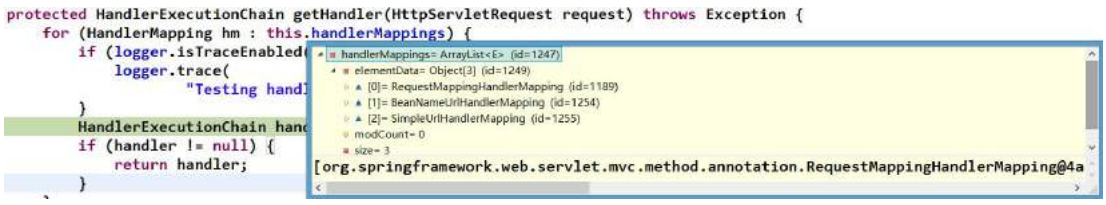
3） 没有配置<mvc:default-servlet-handler/>，<mvc:annotation-driven/>,发送一个不存在资源的请求路径，mappedHandler 为 null

- http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2

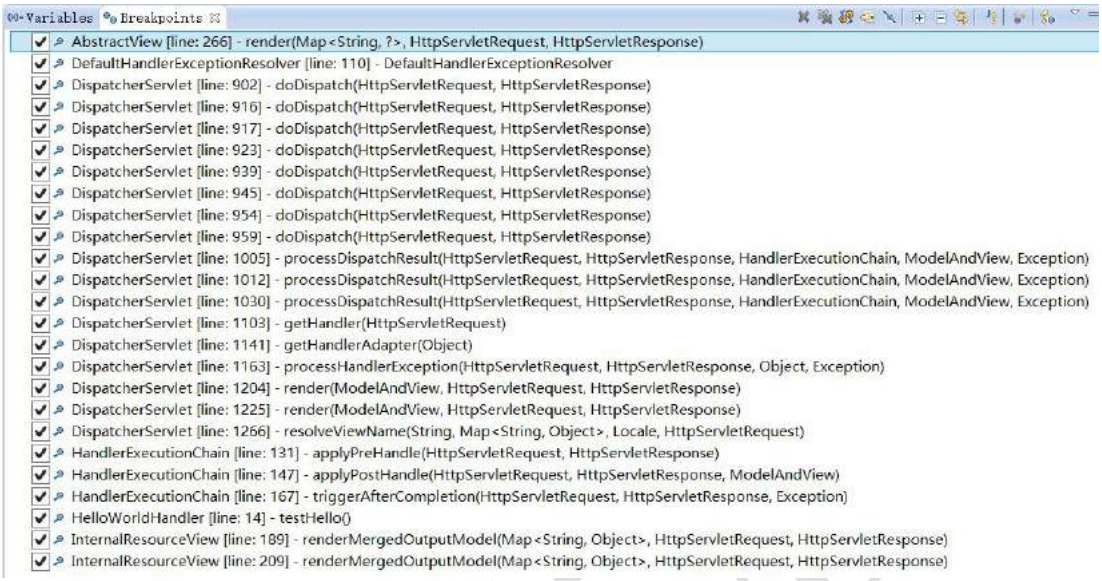


4） 配置<mvc:default-servlet-handler/>，<mvc:annotation-driven/>,发送一个不存在资源的请求路径

- http://localhost:8080/SpringMVC_09_WorkFlow/helloworld2
- mappedHandler 不为 null,原因是当循环 simpleUrlHandlerMapping 时，当做静态资源处理



14.3.5 断点



第 15 章 Spring 整合 SpringMVC

15.1 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

15.2 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>
```

2） 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>
```

3） 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>
```

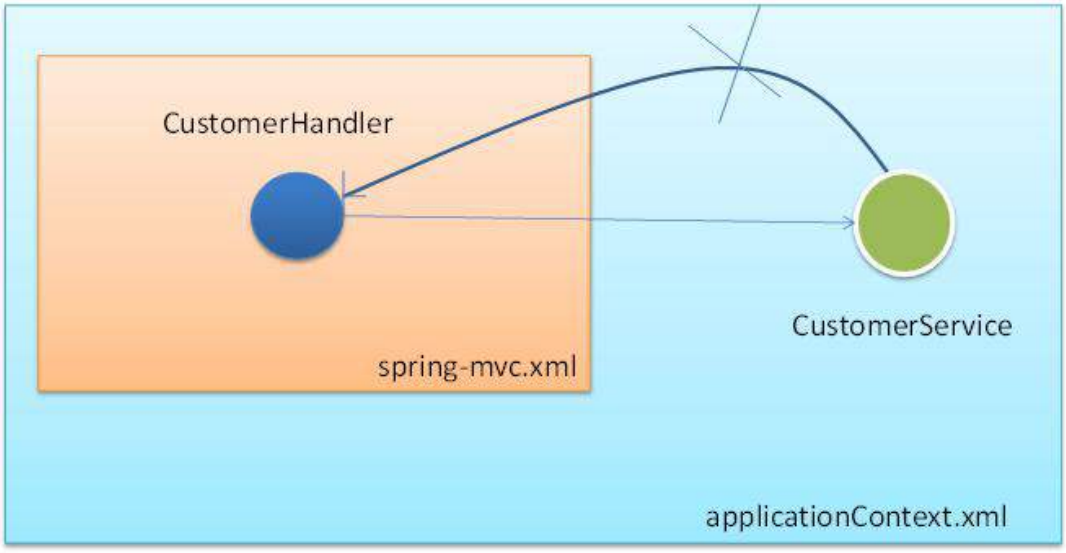
beans.xml

```
<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. -->
```

15.3 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



15.4 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.
4） 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
5） Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些.