**Springmvc第二天**

上海千锋 文威

* 响应数据和结果视图
* Springmvc实现文件上传
* Springmvc中的异常处理
* Springmvc中的拦截器
* 整合SSM

## 1.1 响应数据和结果视图

首先搭建今日工程的项目环境，这里省略。

### 1.1.1 返回值分类

#### 1.1.1.1 返回字符串

controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。这种方式我们在上次课里面讲到过，在这里我们简单的演示一下即可。

* 编写jsp页面

|  |
| --- |
| <**a href="user/testString"**>[返回值是字符串]</**a**> <**br**> |

* 编写controller控制器

|  |
| --- |
| @RequestMapping(**"testString"**) **public** String testString(){  **return "success"**; } |

* 测试省略

#### 1.1.1.2 返回值void

在昨天的学习中，我们知道Servlet原始 API 可以作为控制器中方法的参数，所以在 controller方法形参上可以定义request和response，使用request或response指定响应结果：

* 编写jsp页面

|  |
| --- |
| <**a href="user/testVoid"**>[返回值是Void]</**a**> <**br**> |

* 编写controller控制器

|  |
| --- |
| @RequestMapping(**"testVoid"**) **public void** testVoid(HttpServletRequest request, HttpServletResponse response) **throws** Exception{  request.getRequestDispatcher(**"/pages/success.jsp"**).forward(request,  response); } |

我们也可以通过response页面重定向：

|  |
| --- |
| @RequestMapping(**"testVoid"**) **public void** testVoid(HttpServletRequest request, HttpServletResponse response) **throws** Exception{  response.sendRedirect(**"/pages/success.jsp"**); } |

#### 1.1.1.3 返回值ModelAndView

ModelAndView是SpringMVC 为我们提供的一个对象，该对象也可以用作控制器方法的返回值。

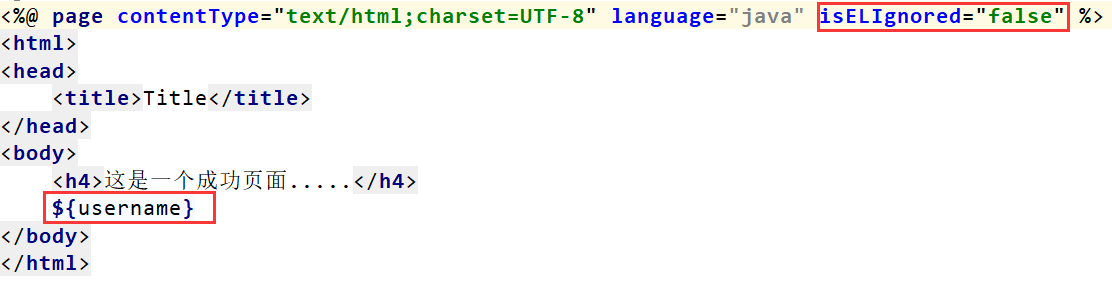
* 编写jsp页面

|  |
| --- |
| <**a href="user/testModelAndView"**>[返回值是ModelAndView]</**a**> <**br**> |

* 编写controller控制器

|  |
| --- |
| @RequestMapping(**"testModelAndView"**) **public** ModelAndView testModelAndView(){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"username"**,**"eric"**);  modelAndView.setViewName(**"success"**);  **return** modelAndView; } |

* 测试



### 1.1.2 springmvc作用域传值

#### 1.1.2.1 使用原生的Request域对象传值

|  |
| --- |
| */\*\*  \* 不带返回值类型的controller方法  \* 使用原生的Request对象值传递  \*/* @RequestMapping(**"testVoid"**) **public void** testVoid(HttpServletRequest request, HttpServletResponse response) **throws** Exception{  request.setAttribute(**"age"**,18);  *//使用原生的servlet进行转发* request.getRequestDispatcher(**"/pages/success.jsp"**).forward(request,response); } |

#### 1.1.2.2 使用Map集合传值

|  |
| --- |
| */\*\*  \* 带返回值的方法  \* 使用Map集合进行值传递 将map对象作为方法的形式参数  \** ***@return*** *\*/* @RequestMapping(**"hello"**) **public** String sayHello(Map<String,Object> map){  map.put(**"sname"**,**"铁蛋"**);  map.put(**"sage"**,19);  **return "success"**; } |

#### 1.1.2.3 使用Model对象传值

|  |
| --- |
| */\*\*  \* 使用Model对象传值  \** ***@param model*** *\** ***@return*** *\*/* @RequestMapping(**"model"**) **public** String testModel(Model model){  model.addAttribute(**"address"**,**"美国"**);  model.addAttribute(**"score"**,19);  **return "success"**; } |

#### 1.1.2.4 使用ModelAndView对象传值

|  |
| --- |
| */\*\*  \* 使用ModelAndView进行值传递  \** ***@return*** *\*/* @RequestMapping(**"testModelAndView"**) **public** ModelAndView testModelAndView(){  ModelAndView mv = **new** ModelAndView();  *//存储值* mv.addObject(**"username"**,**"kobe"**);  *//设置响应视图* mv.setViewName(**"success"**);  **return** mv; } |

### 1.1.3 转发和重定向

#### 1.1.3.1 forward转发

controller方法在提供了String类型的返回值之后，默认就是请求转发。我们也可以写成：

|  |
| --- |
| @RequestMapping(**"testString"**) **public** String testString(){  *//return "success";* **return "forward:/pages/success.jsp"**; } |

它相当于“request.getRequestDispatcher("url").forward(request,response)”。使用请求转发，既可以转发到jsp，也可以转发到其他的控制器方法。

转发到请他控制器方法:

|  |
| --- |
| @RequestMapping(**"testString"**) **public** String testString(){  **return "forward:testModelAndView"**; } |

#### 1.1.3.2 Redirect重定向

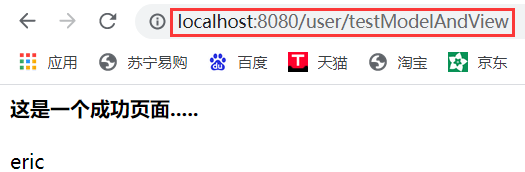
contrller方法提供了一个 String 类型返回值之后，它需要在返回值里使用:redirect:

|  |
| --- |
| @RequestMapping(**"testString"**) **public** String testString(){  **return "redirect:/pages/success.jsp"**; } |

那么redirect可不可以写逻辑视图呢?

|  |
| --- |
| @RequestMapping(**"testString"**) **public** String testString(){  **return "redirect:testModelAndView"**; } |

经过测试，是可以的



### 1.1.3 ResponseBody响应json数据

* 使用说明

**作用：**

该注解用于将Controller的方法返回的对象，通过HttpMessageConverter接口转换为指定格式的数据如:json,xml等，通过Response响应给客户端。

* 使用示例

需求：使用@ResponseBody注解实现将controller方法返回对象转换为json响应给客户端。

准备jsp代码:

|  |
| --- |
| *<!-- 测试异步请求 -->* <**input type="button" value="测试ajax请求json和响应json" id="testJson"**/> |

第一步:在线引入js库文件

|  |
| --- |
| <**script src="http://libs.baidu.com/jquery/1.9.0/jquery.js"**></**script**> |

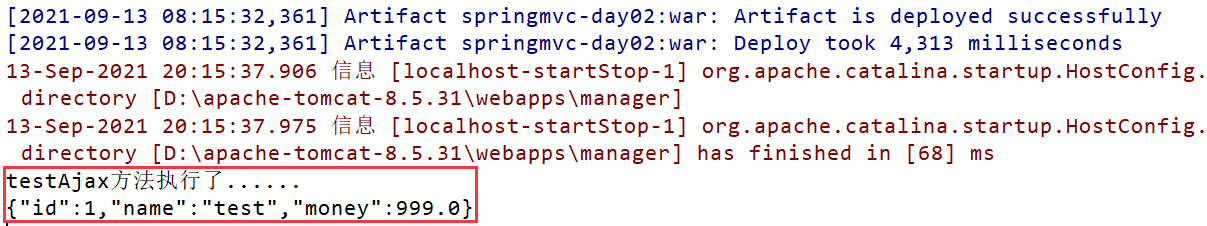
第二步：编写ajax代码

|  |
| --- |
| <**script type="text/javascript"**>  **$**(**function**(){  **$**(**"#testJson"**).click(**function**(){  **$**.**ajax**({  **type**:**"post"**,  **url**:**"${**pageContext.request.contextPath**}/user/testAjax"**,  **contentType**:**"application/json;charset=UTF-8"**,  **data**:**'{"id":1,"name":"test","money":999.0}'**,  **dataType**:**"json"**,  success:**function**(data){  *alert*(data);  }  })  });  }) </**script**> |

第三步:编写控制器代码

|  |
| --- |
| @RequestMapping(**"testAjax"**) **public void** testAjax(@RequestBody String body){  System.***out***.println(**"testAjax方法执行了......"**);  System.***out***.println(body); } |

点击页面上的按钮测试:



控制器方法正常执行，说明我们的ajax代码没有问题。

那么如何在后台获取前台传递的json数据呢?

如何将后台的数据(pojo)以json格式传递给前台呢?

首先我们需要导入json相关的依赖:

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

然后我们还需要改造控制器代码:

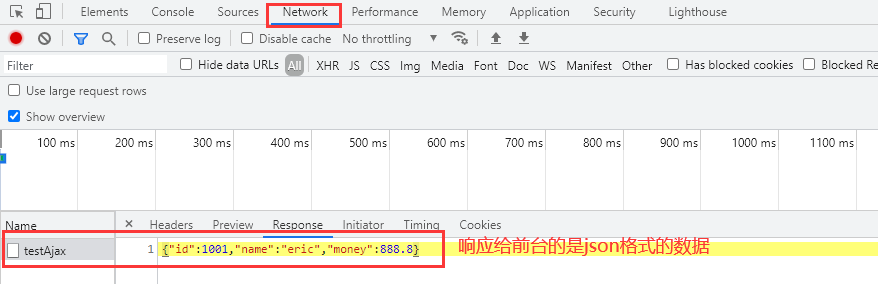
|  |
| --- |
| @RequestMapping(**"testAjax"**) **public** @ResponseBody User testAjax(@RequestBody User user){  System.***out***.println(**"testAjax方法执行了......"**);  System.***out***.println(user);  *//模拟从数据可能查询的数据* user.setId(1001);  user.setName(**"eric"**);  user.setMoney(888.8);  **return** user; } |

@RequestBody:会自动的将前台传递的json格式的数据转换成pojo对象。

@ResponseBody:会自动的将pojo对象以json格式的数据传递给前台。

最后我们测试结果:





## 1.2 SpringMVC 实现文件上传

### 1.2.1 使用传统方式进行文件上传

#### 1.2.1.1 导入文件上传的依赖

|  |
| --- |
| <**dependency**>  <**groupId**>commons-fileupload</**groupId**>  <**artifactId**>commons-fileupload</**artifactId**>  <**version**>1.3.1</**version**> </**dependency**> <**dependency**>  <**groupId**>commons-io</**groupId**>  <**artifactId**>commons-io</**artifactId**>  <**version**>2.4</**version**> </**dependency**> |

#### 1.2.1.2 编写文件上传jsp页面

|  |
| --- |
| <**body**>  <**form action="upload" method="post" enctype="multipart/form-data"**>  文件: <**input type="file" name="upload"**></**input**>  <**input type="submit" value="提交"**>  </**form**> </**body**> |

#### 1.2.1.3 编写文件上传的代码

|  |
| --- |
| @RequestMapping(**"upload"**) **public** String fileUpload(HttpServletRequest request) **throws** Exception{  System.***out***.println(**"----开始执行文件上传----"**);  *//获取要上传的文件目录* String path = request.getSession().getServletContext().getRealPath(**"/uploads"**);  System.***out***.println(**"path:"**+path);  *//根据文件上传的目录创建File对象，如果不存在则创建1个File对象* File file = **new** File(path);  **if**(!file.exists()){  *//创建一个file对象* file.mkdirs();  }  *//创建磁盘文件项工厂* DiskFileItemFactory factory = **new** DiskFileItemFactory();  *//根据磁盘文件项工厂创建文件上传对象* ServletFileUpload fileUpload = **new** ServletFileUpload(factory);  *//如果存在，就准备解析request作用域里面上传的数据* List<FileItem> fileItems = fileUpload.parseRequest(request);  *// 遍历* **for**(FileItem fileItem:fileItems){  **if**(fileItem.isFormField()){  *//普通表单项* }**else**{  *//获取文件上传的文件名称* String name = fileItem.getName();  *//上传文件* fileItem.write(**new** File(path,name));  *//删除临时文件* fileItem.delete();  }  }  **return "success"**; } |

### 1.2.2 使用springmvc的方式进行文件上传

* 引入文件上传的依赖(省略)
* 编写文件上传的jsp页面(省略)
* 编写文件上传的控制器代码

|  |
| --- |
| @RequestMapping(path=**"/upload"**) **public** String fileUpload(HttpServletRequest request, MultipartFile upload) **throws** IOException {  System.***out***.println(**"springmvc方式的文件上传"**);  *//获取要上传的文件目录* String path = request.getSession().getServletContext().getRealPath(**"/uploads"**);  System.***out***.println(**"path:"**+path);  *//根据文件上传的目录创建File对象，如果不存在则创建1个File对象* File file = **new** File(path);  **if**(!file.exists()){  *//创建一个file对象* file.mkdirs();  }  *//获取文件上传名称* String filename = upload.getOriginalFilename();  *//完成文件上传* upload.transferTo(**new** File(path,filename));  **return "success"**; } |

* 配置文件上传解析器

|  |
| --- |
| *<!--  配置文件上传解析器 -->* <**bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  <**property name="defaultEncoding" value="UTF-8"** />  <**property name="maxUploadSize" value="2097152"** /> </**bean**> |

## 1.3 SpringMVC 中的异常处理

### 1.3.1 springmvc默认处理异常的方式

Springmvc默认处理异常的方式和servlet处理异常的方式是一样的。都是为web.xml里面定义错误处理代码和错误处理页面。

* 在web.xml里面定义错误处理页面

|  |
| --- |
| *<!--配置错误处理页面-->* <**error-page**>  <**error-code**>400</**error-code**>  <**location**>/error/400.jsp</**location**> </**error-page**>  <**error-page**>  <**error-code**>404</**error-code**>  <**location**>/error/404.jsp</**location**> </**error-page**>  <**error-page**>  <**error-code**>500</**error-code**>  <**location**>/error/500.jsp</**location**> </**error-page**> |

* 编写错误处理页面

在error目录下面，编写error.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" **isELIgnored**="**false**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**>  <**h4**>这是一个错误处理页面</**h4**> </**body**> </**html**> |

### 1.3.2 @ExceptionHandler注解进行异常处理

* 编写控制器代码

|  |
| --- |
| @Controller @RequestMapping(**"exception"**) **public class** ExceptionController {   */\*\*  \* 模拟 NullPointerException  \** ***@return*** *\*/* @RequestMapping(**"/show1"**)  **public** String showInfo(){  String str = **null**;  str.length();  **return "success"**;  }   */\*\*  \* 模拟 ArithmeticException  \** ***@return*** *\*/* @RequestMapping(**"/show2"**)  **public** String showInfo2(){  **int** a = 10/0;  **return "success"**;  }   */\*\*  \* 模拟数组下标越界 ArrayIndexOutOfBoundsException  \** ***@return*** *\*/* @RequestMapping(**"/show3"**)  **public** String showInfo3(){  **int**[] arr = **new int**[3];  arr[100] = 99;  **return "success"**;  }  } |

* 编写错误处理的方法

在当前的controller里面进行异常处理。

|  |
| --- |
| */\*\*  \* 处理空指针异常  \* Exception ex 封装的是对应异常处理的信息  \** ***@return*** *\*/* @ExceptionHandler(value = {NullPointerException.**class**}) **public** ModelAndView nullPointerExceptionHandler(Exception e){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,e.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView; }  */\*\*  \* 处理算术异常  \** ***@param e*** *\** ***@return*** *\*/* @ExceptionHandler(value = ArithmeticException.**class**) **public** ModelAndView arithmeticExceptionHandler(Exception e){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,e.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView; }  */\*\*  \* 处理下标越界异常  \** ***@param ex*** *\** ***@return*** *\*/* @ExceptionHandler(value = ArrayIndexOutOfBoundsException.**class**) **public** ModelAndView arrayIndexOutOfBoundsExceptionHandler(Exception ex){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,ex.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView; } |

我们也可以一次性的处理所有异常

|  |
| --- |
| */\*\*  \* 处理了所有异常  \** ***@param ex*** *\** ***@return*** *\*/* @ExceptionHandler(value = **RuntimeException**.**class**)  **public** ModelAndView runtimeExceptionHandler(Exception ex){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,ex.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView; } |

这种异常处理的方式解决了问题。但是异常处理的代码和控制器代码在一个controller里面定义，违反了代码编写的职责单一性。

### 1.3.3 使用@controlleradvice注解

* 编写控制器类

省略

* 定义一个类，专门进行异常处理

|  |
| --- |
| @ControllerAdvice **public class** MyExceptionHandler {  */\*\*  \* 处理空指针异常  \* Exception ex 封装的是对应异常处理的信息  \** ***@return*** *\*/* @ExceptionHandler(value = NullPointerException.**class**)  **public** ModelAndView nullPointerExceptionHandler(Exception e){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,e.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView;  }   */\*\*  \* 处理算术异常  \** ***@param e*** *\** ***@return*** *\*/* @ExceptionHandler(value = ArithmeticException.**class**)  **public** ModelAndView arithmeticExceptionHandler(Exception e){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,e.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView;  }   */\*\*  \* 处理下标越界异常  \** ***@param ex*** *\** ***@return*** *\*/* @ExceptionHandler(value = ArrayIndexOutOfBoundsException.**class**)  **public** ModelAndView arrayIndexOutOfBoundsExceptionHandler(Exception ex){  ModelAndView modelAndView = **new** ModelAndView();  modelAndView.addObject(**"msg"**,ex.toString());  modelAndView.setViewName(**"error"**);  **return** modelAndView;  }  } |

我们也可以将异常统一处理(RuntimeException)。类似于上面。

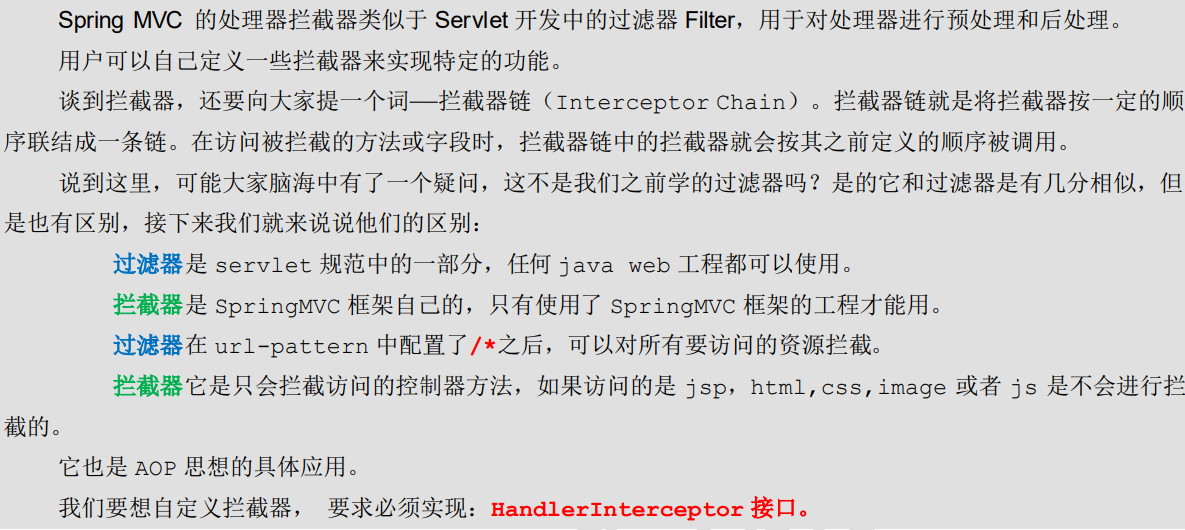
### 1.3.4 实现HandlerExceptionResolver接口进行异常处理

* 定义一个类，实现HandlerExceptionResolver接口

|  |
| --- |
| @Component **public class** MyExceptionResolver **implements** HandlerExceptionResolver {   **public** ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) {  ModelAndView mv = **new** ModelAndView();  mv.addObject(**"msg"**,e.toString());  mv.setViewName(**"error"**);  **return** mv;  } } |

## 1.4 springmvc中的拦截器

### 1.4.1 拦截器的作用



### 1.4.2 自定义拦截器步骤

#### 1.4.2.1 编写控制器类

|  |
| --- |
| @RequestMapping(**"testInterceptor"**) **public** String testInterceptor(){  System.***out***.println(**"testInterceptor方法执行了....."**);  **return "success"**; } |

在success.jsp页面上输出一句话

|  |
| --- |
| **<%** System.***out***.println(**"success页面执行了....."**); **%>** |

#### 1.4.2.2 编写拦截器类

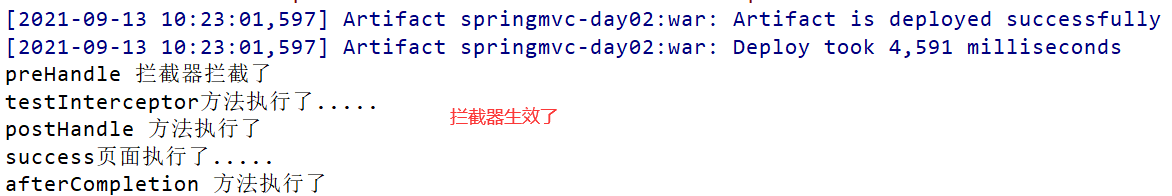
自定义一个类实现HandlerInterceptor接口。

|  |
| --- |
| */\*\*  \* 自定义拦截器  \*/* **public class** MyInterceptor1 **implements** HandlerInterceptor {   */\*\*  \* 预处理 controller方法执行前 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  System.***out***.println(**"preHandle 拦截器拦截了"**);  **return true**;  }   */\*\*  \* 后处理 controller方法执行后 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param modelAndView*** *\** ***@throws*** *Exception  \*/* **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  System.***out***.println(**"postHandle 方法执行了"**);  }   */\*\*  \* 页面执行完 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@throws*** *Exception  \*/* **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.***out***.println(**"afterCompletion 方法执行了"**);  } } |

#### 1.4.2.3 配置拦截器

|  |
| --- |
| *<!-- 配置拦截器 -->* <**mvc:interceptors**>  <**mvc:interceptor**>  *<!--你要拦截的具体方法-->* <**mvc:mapping path="/user/\*"**/>  *<!--你要拦截的方法-->  <!--<mvc:exclude-mapping path=""></mvc:exclude-mapping>-->* <**bean id="handlerInterceptorDemo1" class="com.qf.interceptor.MyInterceptor1"**></**bean**>  </**mvc:interceptor**> </**mvc:interceptors**> |

#### 1.4.2.4 测试

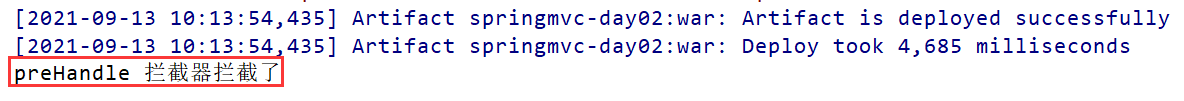


### 1.4.3 拦截器的细节

#### 1.4.3.1 拦截器的放行

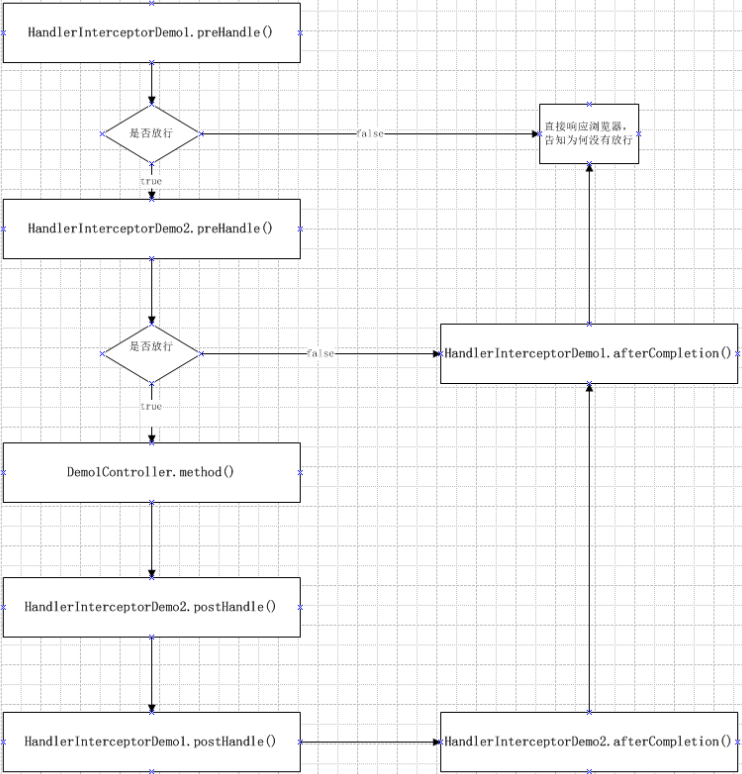
放行的含义是指，如果有下一个拦截器就执行下一个，如果该拦截器处于拦截器链的最后一个，则执行控制器中的方法。





#### 1.4.3.2 多个拦截器的执行顺序

多个拦截器是按照配置的顺序决定的。



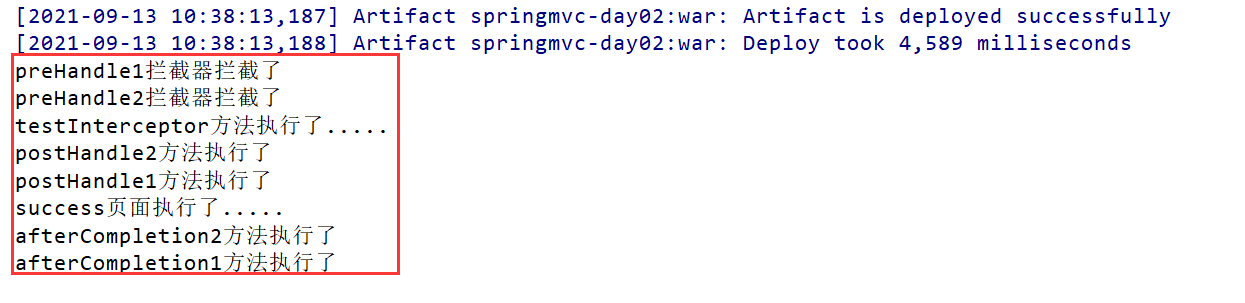
* 配置拦截器1(已经定义，就不再定义)
* 配置拦截器2(复制拦截器1的代码，改造即可)

|  |
| --- |
| **public class** MyInterceptor2 **implements** HandlerInterceptor {   */\*\*  \* 预处理 controller方法执行前 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  *//request.getRequestDispatcher("/error/error.jsp").forward(request,response);* System.***out***.println(**"preHandle2拦截器拦截了"**);  **return true**;  }   */\*\*  \* 后处理 controller方法执行后 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param modelAndView*** *\** ***@throws*** *Exception  \*/* **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {   System.***out***.println(**"postHandle2方法执行了"**);  }   */\*\*  \* 页面执行完 执行这个方法  \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@throws*** *Exception  \*/* **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  System.***out***.println(**"afterCompletion2方法执行了"**);  } } |

* 配置多个拦截器

|  |
| --- |
| *<!-- 配置拦截器 -->* <**mvc:interceptors**>  <**mvc:interceptor**>  *<!--你要拦截的具体方法-->* <**mvc:mapping path="/user/\*"**/>  *<!--你要拦截的方法-->  <!--<mvc:exclude-mapping path=""></mvc:exclude-mapping>-->* <**bean id="handlerInterceptorDemo1" class="com.qf.interceptor.MyInterceptor1"**></**bean**>  </**mvc:interceptor**>   <**mvc:interceptor**>  *<!--你要拦截的具体方法-->* <**mvc:mapping path="/user/\*"**/>  *<!--你要拦截的方法-->  <!--<mvc:exclude-mapping path=""></mvc:exclude-mapping>-->* <**bean id="handlerInterceptorDemo2" class="com.qf.interceptor.MyInterceptor2"**></**bean**>  </**mvc:interceptor**> </**mvc:interceptors**> |

* 测试



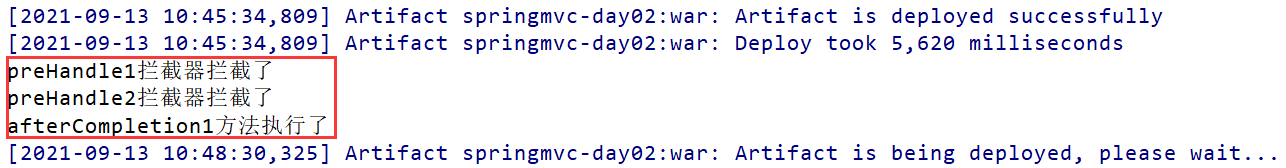
#### 1.4.3.3 中断流程测试

如果我们想中断一个拦截器的执行呢?

我们改造拦截器2里面的代码:



其他不用变，然后继续测试:



## 1.5 SSM整合

### 1.5.1创建数据库和表

|  |
| --- |
| CREATE DATABASE ssm;  USE ssm;  CREATE TABLE account(  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR(20),  money DOUBLE  ) |

### 1.5.2 引入依赖

|  |
| --- |
| <**properties**>  <**spring.version**>5.0.2.RELEASE</**spring.version**>  <**slf4j.version**>1.6.6</**slf4j.version**>  <**log4j.version**>1.2.12</**log4j.version**>  <**mysql.version**>5.1.6</**mysql.version**>  <**mybatis.version**>3.4.5</**mybatis.version**> </**properties**>  <**dependencies**>  *<!-- spring -->* <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjweaver</**artifactId**>  <**version**>1.6.8</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-aop</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-web</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-tx</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  <**scope**>compile</**scope**>  </**dependency**>   <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>${mysql.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>servlet-api</**artifactId**>  <**version**>2.5</**version**>  <**scope**>provided</**scope**>  </**dependency**>   <**dependency**>  <**groupId**>javax.servlet.jsp</**groupId**>  <**artifactId**>jsp-api</**artifactId**>  <**version**>2.0</**version**>  <**scope**>provided</**scope**>  </**dependency**>   <**dependency**>  <**groupId**>jstl</**groupId**>  <**artifactId**>jstl</**artifactId**>  <**version**>1.2</**version**>  </**dependency**>   *<!-- log start -->* <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>${log4j.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  <**version**>${slf4j.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>${slf4j.version}</**version**>  </**dependency**> *<!-- log end -->* <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>${mybatis.version}</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis-spring</**artifactId**>  <**version**>1.3.0</**version**>  </**dependency**>   <**dependency**>  <**groupId**>c3p0</**groupId**>  <**artifactId**>c3p0</**artifactId**>  <**version**>0.9.1.2</**version**>  <**type**>jar</**type**>  <**scope**>compile</**scope**>  </**dependency**> </**dependencies**> |

然后部署项目，只需要将项目部署到tomcat服务器即可。

### 1.5.3 编写实体类

|  |
| --- |
| **public class** Account {  **private** Integer **id**;  **private** String **name**;  **private** Double **money**;  **public** Integer getId() {  **return id**;  }  **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", name='"** + **name** + **'\''** +  **", money="** + **money** +  **'}'**;  } } |

### 1.5.5 编写dao

|  |
| --- |
| **public interface** AccountDao {  **public void** saveAccount(Account account);  **public** List<Account> findAll(); } |

### 1.5.6 编写service

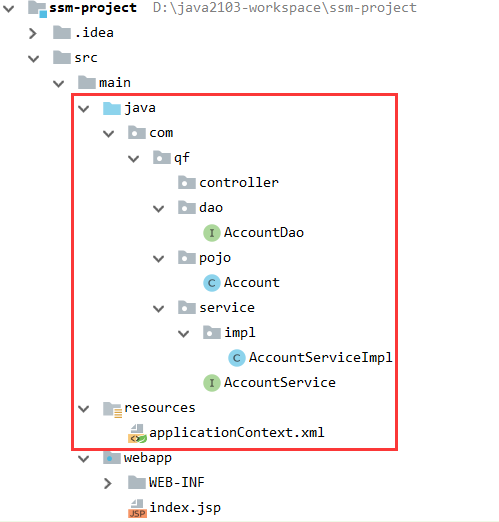
|  |
| --- |
| **public interface** AccountService {    **public void** saveAccount(Account account);    **public** List<Account> findAll(); } |

|  |
| --- |
| @Service(**"accountService"**) **public class** AccountServiceImpl **implements** AccountService { **public void** saveAccount(Account account) {  System.***out***.println(**"业务层查询所有账户信息"**);  }   **public** List<Account> findAll() {  System.***out***.println(**"业务层，保存账户"**);  **return null**;  } } |

### 1.5.7 编写spring配置文件

|  |
| --- |
| *<!--开启注解扫描--> <!-- 开启注解扫描，要扫描的是service和dao层的注解，要忽略web层注解，因为web层让SpringMVC框架 去管理 -->* <**context:component-scan base-package="com.qf"**>  *<!--配置哪些注解不扫描-->* <**context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"**></**context:exclude-filter**> </**context:component-scan**>  *<!-- 配置C3P0的连接池对象 -->* <**bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"** />  <**property name="url" value="jdbc:mysql://192.168.10.137:3306/ssm"** />  <**property name="username" value="root"** />  <**property name="password" value="Admin123!"** /> </**bean**>  *<!-- 配置SqlSession的工厂 -->* <**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  *<!--通过连接池构建Session工厂-->* <**property name="dataSource" ref="dataSource"** /> </**bean**> *<!-- 配置扫描dao的包 -->* <**bean id="mapperScanner" class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  <**property name="basePackage" value="com.qf.dao"**/> </**bean**>  *<!--配置平台事务管理器-->* <**bean id="dataSourceTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**bean**> <**tx:advice id="txAdvice" transaction-manager="dataSourceTransactionManager"**>  <**tx:attributes**>  *<!--只读事务-->* <**tx:method name="findAll" read-only="true"**/>  <**tx:method name="\*" isolation="DEFAULT"**></**tx:method**>  </**tx:attributes**> </**tx:advice**> *<!--配置aop增强-->* <**aop:config**>  <**aop:advisor advice-ref="txAdvice" pointcut="execution(\* com.qf.service.impl.AccountServiceImpl.saveAccount(..))"**></**aop:advisor**> </**aop:config**> |

最后目录结构如下:

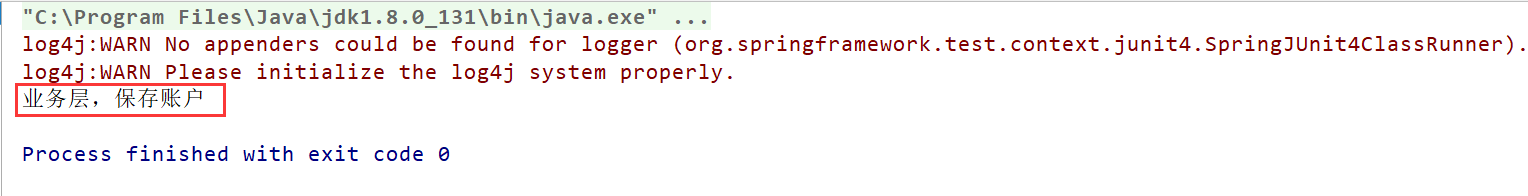


接下来，我们可以测试一下，我们的spring环境搭建的怎么样。

编写一个测试类:

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(locations = {**"classpath:applicationContext.xml"**}) **public class** TestAccount {   @Autowired  AccountService **accountService**;   @Test  **public void** test(){  **accountService**.findAll();  } } |

查看控制台效果:



我们测试，没有问题，接下来我们就可以整合springmvc了。

### 1.5.8 编写springmvc配置文件

编写springmvc.xml配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>  *<!-- 扫描controller的注解，别的不扫描 -->* <**context:component-scan base-package="com.qf"**>  <**context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"**/>  </**context:component-scan**>   *<!-- 配置视图解析器 -->* <**bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  *<!-- JSP文件所在的目录 -->* <**property name="prefix" value="/pages/"** />  *<!-- 文件的后缀名 -->* <**property name="suffix" value=".jsp"** />  </**bean**>  *<!-- 设置静态资源不过滤 -->* <**mvc:resources location="/css/" mapping="/css/\*\*"** />  <**mvc:resources location="/images/" mapping="/images/\*\*"** />  <**mvc:resources location="/js/" mapping="/js/\*\*"** />  *<!-- 开启对SpringMVC注解的支持 -->* <**mvc:annotation-driven** /> </**beans**> |

### 1.5.9 编写jsp页面/控制器

在index.jsp页面里面编写:

|  |
| --- |
| <**a href="account/findAll"**>查询所有</**a**> |

编写AccountController:

|  |
| --- |
| @Controller @RequestMapping(**"account"**) **public class** AccountController {   @Autowired  AccountService **accountService**;   @RequestMapping(**"findAll"**)  **public** String findAll(){  **accountService**.findAll();  **return "list"**;  } } |

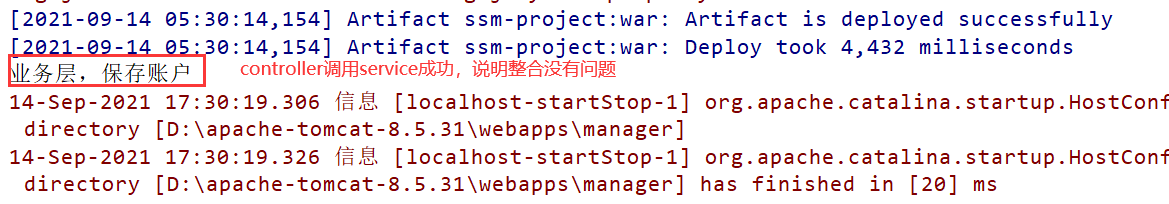
### 1.5.10 编写web.xml

web.xml里面需要配置加载spring配置文件的监听器，还有就是springmvc的前端控制器，以及解决中文乱码的编码过滤器。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1"**>  <**display-name**>Archetype Created Web Application</**display-name**>  *<!-- 配置Spring的监听器 -->* <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  </**listener**>  *<!-- 配置加载类路径的配置文件 -->* <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:applicationContext.xml</**param-value**>  </**context-param**>  *<!-- 配置前端控制器：服务器启动必须加载，需要加载springmvc.xml配置文件 -->* <**servlet**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置初始化参数，创建完DispatcherServlet对象，加载springmvc.xml配置文件 -->* <**init-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:springmvc.xml</**param-value**>  </**init-param**>  *<!-- 服务器启动的时候，让DispatcherServlet对象创建 -->* <**load-on-startup**>1</**load-on-startup**>  </**servlet**>  <**servlet-mapping**>  <**servlet-name**>dispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**>  <**filter**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.CharacterEncodingFilter</**filter-class**>  *<!-- 设置过滤器中的属性值 POST -->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>UTF-8</**param-value**>  </**init-param**>  *<!-- 启动过滤器 -->* <**init-param**>  <**param-name**>forceEncoding</**param-name**>  <**param-value**>true</**param-value**>  </**init-param**>  </**filter**>  *<!-- 过滤所有请求 -->* <**filter-mapping**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**> </**web-app**> |

### 1.5.11 测试springmvc是否整合成功

点击index.jsp页面的超链接标签[查询所有]:



### 1.5.12 整合mybatis

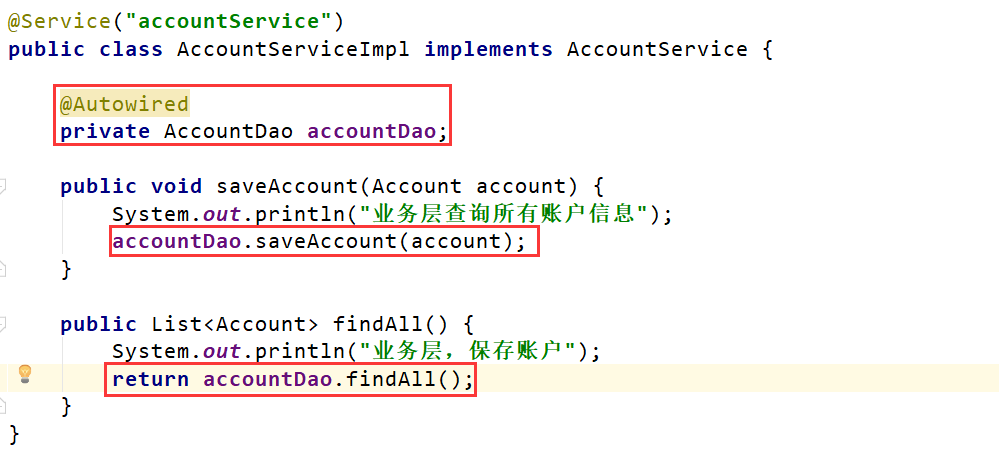
* 在spring的配置文件applicationContext.xml里面添加mybatis的配置信息

|  |
| --- |
| *<!-- 配置C3P0的连接池对象 -->* <**bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"** />  <**property name="url" value="jdbc:mysql://192.168.10.137:3306/ssm"** />  <**property name="username" value="root"** />  <**property name="password" value="Admin123!"** /> </**bean**>  *<!-- 配置SqlSession的工厂 -->* <**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  *<!--通过连接池构建Session工厂-->* <**property name="dataSource" ref="dataSource"** /> </**bean**> *<!-- 配置扫描dao的包 -->* <**bean id="mapperScanner" class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  <**property name="basePackage" value="com.qf.dao"**/> </**bean**> |

* 在接口上使用注解的方式实现接口

|  |
| --- |
| @Insert(**"insert into account(name,money) values(#{name},#{money})"**) **public void** saveAccount(Account account);  @Select(**"select \* from account"**) **public** List<Account> findAll(); |

* 改造AccountServiceImpl



* 改造controller

|  |
| --- |
| @Controller @RequestMapping(**"account"**) **public class** AccountController {   @Autowired  AccountService **accountService**;   */\*\*  \* 查询所有  \** ***@return*** *\*/* @RequestMapping(**"findAll"**)  **public** String findAll(Model model){  List<Account> list = **accountService**.findAll();  model.addAttribute(**"list"**,list);  **return "list"**;  }   */\*\*  \* 保存  \** ***@param account*** *\** ***@return*** *\*/* @RequestMapping(**"save"**)  **public** String save(Account account){  **accountService**.saveAccount(account);  **return "redirect:findAll"**;  } } |

* 改造jsp页面

index.jsp页面:

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**>  <**a href="account/findAll"**>查询所有</**a**>   <**h3**>测试保存的方法</**h3**>  <**form action="account/save" method="post"**>  姓名: <**input type="text" name="name"**> <**br**>  金额: <**input type="text" name="money"**> <**br**>  <**input type="submit" value="提交"**>  </**form**> </**body**> |

list.jsp页面:

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" **isELIgnored**="**false**" %> <%@**taglib prefix**="**c**" **uri**="**http://java.sun.com/jsp/jstl/core**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> <**h2**>查询所有账户</**h2**> <**c:forEach var="list" items="${**list**}"**>  **${**list.id**} ${**list.name**} ${**list.money**}** <**br**> </**c:forEach**> </**body**> </**html**> |

最后分别测试查询和新增的请求(省略)。

面试题补充:

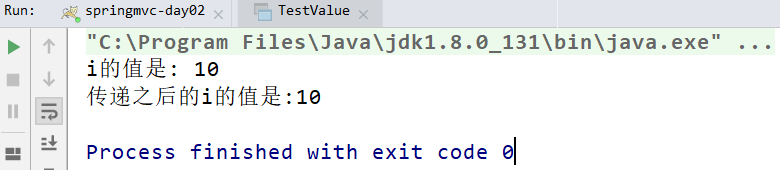
值传递和引用传递的区别:

值传递的概念:将实参赋予给形参。

* 基本类型数据的值传递

|  |
| --- |
| **public class** TestValue {  **public static int** *i* = 10;   **public static void** update(**int** value){  value = 10 \* value;  }  **public static void** main(String[] args) {  System.***out***.println(**"i的值是: "** + *i*);  *update*(*i*);*//值传递 将基本类型数据i传递给value这个形式参数* System.***out***.println(**"传递之后的i的值是:"** + *i*);  } } |

结果:



* 引用类型数据的值传递

|  |
| --- |
| **public class** TestAnimal {  **private static** Animal *animal* = **null**;   **public static void** updateUser(Animal animal) {  animal.setName(**"Lishen"**);  animal.setAge(18);  }   **public static void** main(String[] args) {  *animal* = **new** Animal(**"zhangsan"**, 26);  System.***out***.println(**"调用前user的值："** + *animal*.toString());  *updateUser*(*animal*);//将animal这个对象赋予给形式参数animal  System.***out***.println(**"调用后user的值："** + *animal*.toString());  } } |

