springmvc第二天 注解开发高级知识

# 复习

springmvc框架：

用户请求url到DispatcherServlet前端控制器，相当于中央调度器，降低系统各组件之间耦合度

DispatcherServlet前端控制器通过HandlerMapping根据url找到Handler。

DispatcherServlet前端控制器通过HandlerAdapter处理器适配器执行Handler。

DispatcherServlet前端控制器拿着Handler返回的ModelAndView通过视图解析器ViewResolver去进行视图解析。

视图解析：将程序中写的逻辑视图名，转成真正的视图（springmvc通过view表示各各不同类型的视图）。

DispatcherServlet前端控制器调用View的渲染方法进行视图渲染（将ModelAndView中的Model放到request域）。

**要掌握springmvc的注解开发，企业中常用springmvc注解开发。**

使用专门注解处理器映射器（RequestMappingHandlerMapping）和处理器适配器（RequestMappingHandlerAdapter）。

<mvc:annotation-driven/>可以代替上边的处理器映射器和适配器的配置。

在Handler(Controller)中定义很多的方法，一个方法通过RequestMapping和url进行映射。

方法返回值：ModelAndView、string(jsp的逻辑视图名)、void（通过response将数据输出成json）

方法输入参数（形参）：springmvc需要将请求的key/value(串，id=001&type=t002)、解析、绑定到Handler(Controller)中方法的形参上。

springmvc默认支持多类型的参数绑定。

默认支持哪些类型：

HttpServletRequest、response、session、Model(用于将数据填充到request域)

@requestParam注解：用于绑定单个请求参数，常用于简单类型参数（Integer、String 、Float。。。）绑定。

不用 @requestParam要求请求参数的名称和方法形参名一致方可绑定。

对于简单类型参数中的日期型，建议使用自定义参数绑定，对日期型数据个化定义日期的格式。

自定义参数绑定：建议使用Convertor进行参数绑定。

还可以绑定pojo、包装的pojo。

# 课程安排

注解开发：

数据回显：表单提交出现错误，重新回到表单，用户重新填写数据，刚才提交的参数在页面上回显。

集合类型（String[]、List<>、map(自学)）的参数绑定。

springmvc上传图片（重点）

json数据交互（提交json数据、响应json数据）（重点）

**Validation(springmvc使用校验方式，使用Hibernate Validator（和Hibernate的ORM没有任何关系）)**

**异常处理器（可以用于系统的统一异常处理，架构的内容）**

springmvc提供RESTful支持

拦截器（用于权限控制）

学生练习

# 数据回显

## 需求

表单提交出现错误，重新回到表单，用户重新填写数据，刚才提交的参数在页面上回显。

## 对简单类型的数据回显

对商品修改数据回显：

注意在进入修改页面的controller方法中和提交修改商品信息方法model.addAttribute方法设置的key一致。

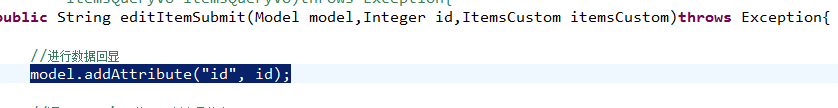
修改商品显示方法：

@RequestMapping(value = **"/editProduct"**)  
**public** String editProduct(Model model, Integer id) **throws** Exception {  
 ModelAndView modelAndView = **new** ModelAndView();  
 *// 将id传到页面* model.addAttribute(**"id"**, id);

修改商品页面：

<**input type="hidden" name="id" value="${**id**}"**/>

修改商品提交方法：



## pojo类型数据回显

### 方法1：

使用Model.addtribute方法进行数据回显：

@RequestMapping(**"/editProductSubmit"**)  
**public** String editProductSubmit(Model model, Integer id, ProductCustom productCustom) **throws** Exception {  
 *// 数据回显* model.addAttribute(**"id"**, id);  
 model.addAttribute(**"product"**, productCustom);

### 方法2：

使用@ModelAttribute，作用于将请求pojo数据放到Model中回显到页面

@RequestMapping(**"/editProductSubmit"**)  
**public** String editProductSubmit(Model model, Integer id, @ModelAttribute(**"product"**) ProductCustom productCustom) **throws** Exception {

在ModelAttribute方法指定的名称就是要填充Model中的key，在页面中就要通过key取数据。

### @ModelAttribute将方法返回值传到页面

需求：商品类别信息在商品信息页面显示。

*/\*\*  
 \* 单独将商品类型的方法提出来，将方法返回值填充到request，在页面显示  
 \*  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/*@ModelAttribute(**"productType"**)  
**public** Map<String, String> getProductType() **throws** Exception {  
 Map<String, String> productType = **new** HashMap<>();  
 productType.put(**"001"**, **"数码"**);  
 productType.put(**"002"**, **"服装"**);  
 **return** productType;  
}

页面：

<**td**>  
 商品类别：  
 <**select**>  
 <**c:forEach items="${**productType**}" var="product"**>  
 <**option value="${**product.key**}"**> **${**product.value**}** </**option**>  
 </**c:forEach**>  
 </**select**>  
</**td**>

使用@ModelAttribute将公用的取数据的方法返回值传到页面，不用在每一个controller方法通过Model将数据传到页面。

# 参数绑定集合类型

## 绑定数组

需求：在商品查询列表页面，用户选择要删除的商品，批量删除商品。

在controller方法中如何将批量提交的数据绑定成数组类型。

### 页面定义

<**c:forEach items="${**productList**}" var="product"**>  
 <**tr**>  
 <**td**><**input type="checkbox" name="delete\_id" value="${**item.id**}"** /></**td**>

### controller方法定义

*/\*\*  
 \* 删除商品  
 \*  
 \** ***@param delete\_id*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/*@RequestMapping(**"deleteProduct"**)  
**public** String deleteProduct(Integer[] delete\_id) **throws** Exception {  
 System.***out***.println(Arrays.*toString*(delete\_id));  
  
 *// 调用service删除  
 // .....* **return "success"**;  
}

## 绑定List<Object>

需求：批量修改商品信息提交。

先进入批量修改商品页面，填写信息，点击提交。

### 页面定义

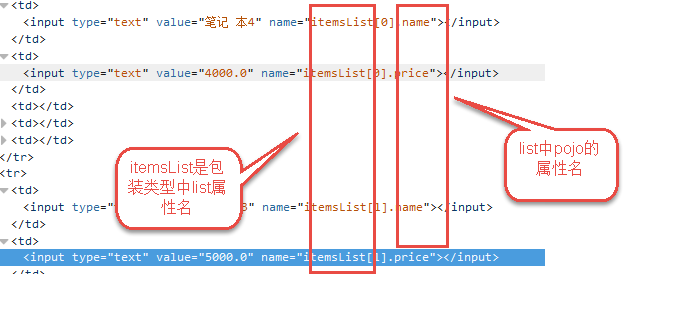
<**c:forEach items="${**productList**}" var="product" varStatus="s"**>  
 <**tr**>  
 <**td**><**input type="text" name="productList[${**s.index **}].name" value="${**product.name **}"**/></**td**>  
 <**td**><**input type="text" name="productList[${**s.index **}].price" value="${**product.price **}"**/></**td**>

注释：

productList：controller方法形参包装类型中list的属性名。

productList [0]或productList [1]。。，[]中是序号，从0开始。

productList [].name：name就是controller方法形参包装类型中list中pojo的属性名

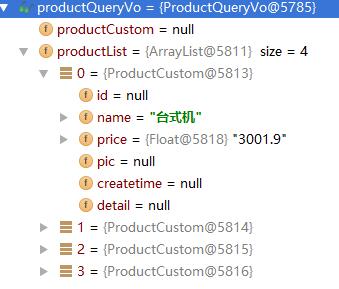


### controller方法定义

使用包装类型接收页面批量提交的数据，绑定成List。

*/\*\*  
 \* 商品查询类  
 \* create:2016-10-30 20:08  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** ProductQueryVo {  
  
 **private** ProductCustom **productCustom**;  
  
 *//定义一个list* **private** List<ProductCustom> **productList**;

*/\*\*  
 \* 批量修改商品提交  
 \*  
 \** ***@param productQueryVo*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/*@RequestMapping(**"/editItemsListSubmit"**)  
**public** String editItemsListSubmit(ProductQueryVo productQueryVo) **throws** Exception {  
  
 **return "success"**;  
}



# springmvc和struts的区别

springmvc是通过方法的形参接收参数，在使用时可以以单例方式使用，建议使用单例。

struts是通过成员变量接收参数，在使用时必须以多例方式使用。

springmvc是基于方法开发，struts基于类开发。

springmvc将一个请求的Method和Handler进行关联绑定，一个method对应一个Handler。

springmvc开发以方法为单位进行开发，方法更帖进service(业务方法)。

**经过实际测试，发现struts标签解析速度比较慢，建议在实际开发时使用jstl。**

# 商品图片上传

## 需求

在商品修改页面，增加图片上传的功能。

操作流程：

用户进入商品修改页面

上传图片

点击提交（提交的是图片和商品信息）

再次进入修改页面，图片在商品修改页面展示

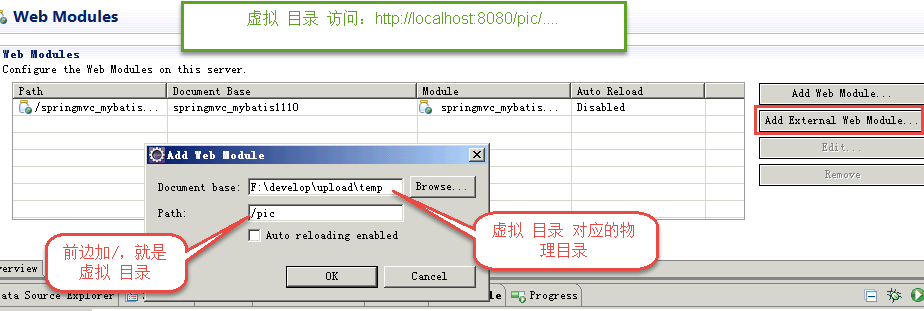
## 图片存储问题

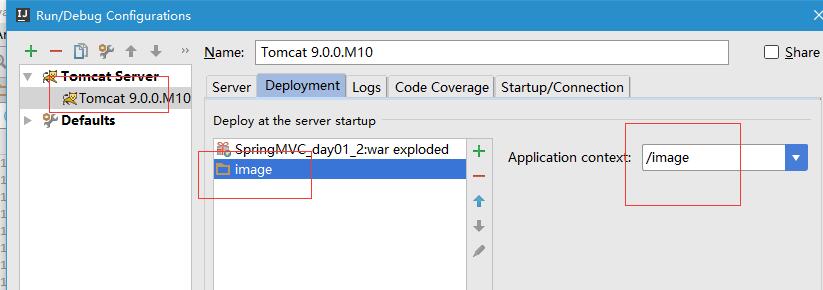
切记：不要把图片上传到工程目录 ，不方便进行工程维护。

实际电商项目中使用专门图片服务器(http，比如apache、tomcat)。

本教程使用图片虚拟目录，通过虚拟目录 访问硬盘上存储的图片目录 。

虚拟 目录 设置：





注意：

图片目录中尽量进行目录分级存储，提高访问速度（提交i/o）。

## 配置图片上传解析器

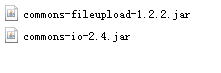
springmvc使用commons-fileupload进行图片上传。

commons-fileupload对应的springmvc的图片上传解析器：

*org.springframework.web.multipart.commons.CommonsMultipartResolver*

*<!-- 文件上传 -->*<**bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  
 *<!-- 设置上传文件的最大尺寸为5MB -->* <**property name="maxUploadSize" value="5242880"**/>  
</**bean**>

加入commons-fileupload的jar包



## 编写上传图片的页面

<**tr**>  
 <**td**>商品图片</**td**>  
 <**td**>  
 <**c:if test="${**product.pic !=**null}"**>  
 <**img src="/image/${**product.pic**}" width=100 height=100/**>  
 <**br**/>  
 </**c:if**>  
 <**input type="file" name="pictureFile"**/>  
 </**td**>  
</**tr**>

设置了虚拟图片目录，所以必须以/开头，不加工程名。

## 编写controller方法

@RequestMapping(**"/editProductSubmit"**)  
**public** String editProductSubmit(Model model, Integer id,  
 @ModelAttribute(**"product"**) ProductCustom productCustom,  
 *// 上传图片* MultipartFile pictureFile) **throws** Exception {  
  
 *//进行图片上传* **if** (pictureFile != **null** && pictureFile.getOriginalFilename() != **null** && pictureFile.getOriginalFilename().length() > 0) {  
 *//图片上传成功后，将图片的地址写到数据库* String filePath = **"F:\\upload\\image\\"**;  
 *//上传文件原始名称* String originalFilename = pictureFile.getOriginalFilename();  
 *//新的图片名称* String newFileName = UUID.*randomUUID*() + originalFilename.substring(originalFilename.lastIndexOf(**"."**));  
 *//新文件* File file = **new** java.io.File(filePath + newFileName);  
  
 *//将内存中的文件写入磁盘* pictureFile.transferTo(file);  
  
 *//图片上传成功，将新图片地址写入数据库* productCustom.setPic(newFileName);  
 }

# json数据的交互

## 需求

json数据格式是比较简单容易理解，json数据格式常用于远程接口传输，http传输json数据，非常方便页面进行提交/请求结果解析，对json数据的解析。

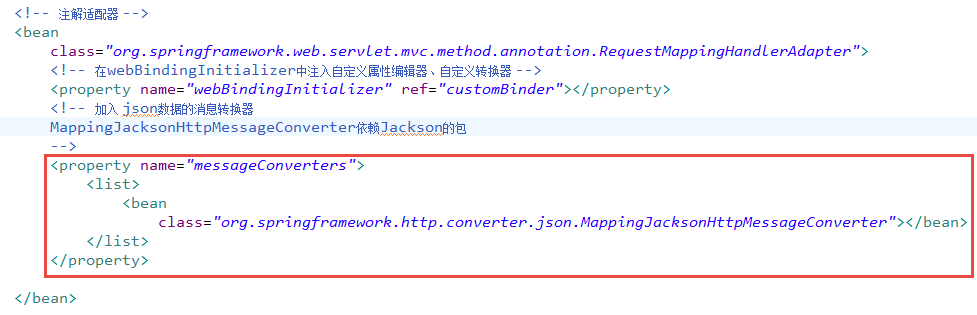
## springmvc解析json加入json解析包

Springmvc默认用MappingJacksonHttpMessageConverter对json数据进行转换，需要加入jackson的包，如下：



## 在处理器适配器中注入MappingJacksonHttpMessageConverter

让处理器适配器支持json数据解析，需要注入MappingJacksonHttpMessageConverter。



上面是以前版本配置

*<!-- 【注解适配器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**>  
 *<!-- 在webBindingInitializer中注入自定义属性编辑器、自定义转换器 -->* <**property name="webBindingInitializer" ref="customBinder"**/>  
  
 *<!-- 加入 json数据的消息转换器 MappingJacksonHttpMessageConverter依赖Jackson的包 -->* <**property name="messageConverters"**>  
 <**list**>  
 <**ref bean="mappingJacksonHttpMessageConverter"**/>  
 </**list**>  
 </**property**>  
</**bean**>

*<!-- json配置 -->*<**bean id="mappingJacksonHttpMessageConverter"  
 class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"**>  
 <**property name="supportedMediaTypes"**>  
 <**list**>

**<value>text/html;charset=UTF-8</value>  
 <value>application/json;charset=utf-8</value>**

</**list**>  
 </**property**>  
</**bean**>

## @RequestBody和@ResponseBody

**@RequestBody：**将请求的json数据转成java对象

**@ResponseBody：**将java对象转成json数据输出。

http请求数据是json格式（json串）

contentType=appliation/json;charset=utf-8

@ResponseBody方法返回值 controller方法形参（@RequestBody pojo对象）

使用@RequestBody将json转成java对象(pojo)

@ResponseBody可以将java对象转成json输出

如果请求的是key/value就不需要使用@RequestBody

## 请求json响应json

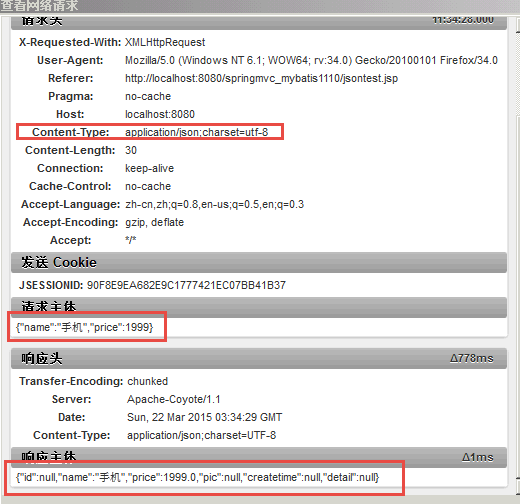
controller方法：

*//请求的json响应json，请求商品信息，商品信息用json格式，输出商品信息*@RequestMapping(**"/requestJson"**)  
**public**@ResponseBody  
ProductCustom requestJson(@RequestBody ProductCustom productCustom) **throws** Exception {  
  
 **return** productCustom;  
}

页面：

*//请求json响应json***function** *requestJson*() {  
 **$**.ajax({  
 **url**: **"${**pageContext.request.contextPath**}/requestJson.action"**,  
 **type**: **"post"**,  
 **contentType**: **"application/json;charset=utf-8"**,  
 *//请求json数据,使用json表示商品信息* **data**: **'{"name":"手机", "price":1999}'**,  
 success: **function** (data) {  
 alert(data.name);  
 }  
 });  
}

测试跟踪：



## 请求key/value响应json

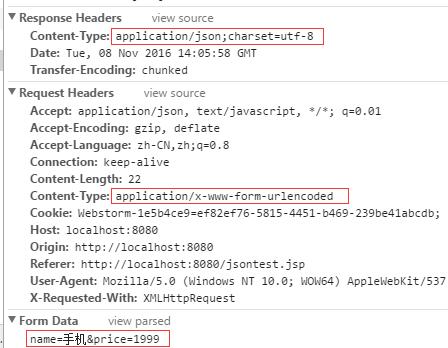
controller方法：

*//请求key/value响应json*@RequestMapping(**"/responseJson.action"**)  
**public**@ResponseBody  
ProductCustom responseJson(ProductCustom productCustom) **throws** Exception {  
  
 **return** productCustom;  
}

页面：

*//请求key/value响应json***function** *responseJson*() {  
 **$**.ajax({  
 **url**: **"${**pageContext.request.contextPath **}/responseJson.action"**,  
 **type**: **"post"**,  
 **dataType**: **"json"**,  
 *//contentType:"application/json;charset=utf-8",  
 //请求key/value数据* **data**: **"name=手机&price=1999"**,  
 success: **function** (data) {  
 alert(data.name);  
 }  
 });  
}

测试：



## 小结

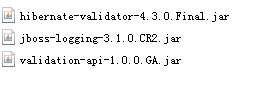
如果前端处理没有特殊要求建议使用第二种，请求key/value，响应json，方便客户端解析请求结果 。

# validation校验（了解）

对前端的校验大多数通过js在页面校验，这种方法比较简单，如果对安全性考虑，还要在后台校验。

springmvc使用JSR-303（javaEE6规范的一部分）校验规范，springmvc使用的是Hibernate Validator（和Hibernate的ORM不是一回事）

## 加入Hibernate Validator的jar



## 在处理器适配器中配置校验器

*<!-- 【注解适配器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**>  
 *<!-- 在webBindingInitializer中注入自定义属性编辑器、自定义转换器 -->* <**property name="webBindingInitializer" ref="customBinder"**/>  
  
 *<!-- 加入 json数据的消息转换器 MappingJacksonHttpMessageConverter依赖Jackson的包 -->* <**property name="messageConverters"**>  
 <**list**>  
 <**ref bean="mappingJacksonHttpMessageConverter"**/>  
 </**list**>  
 </**property**>  
</**bean**>

*<!-- 自定义webBinder -->*<**bean id="customBinder" class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer"**>  
 *<!-- 使用converter进行参数转 -->* <**property name="conversionService" ref="conversionService"**/>  
  
 *<!-- 配置validator -->* <**property name="validator" ref="validator"**/>  
</**bean**>

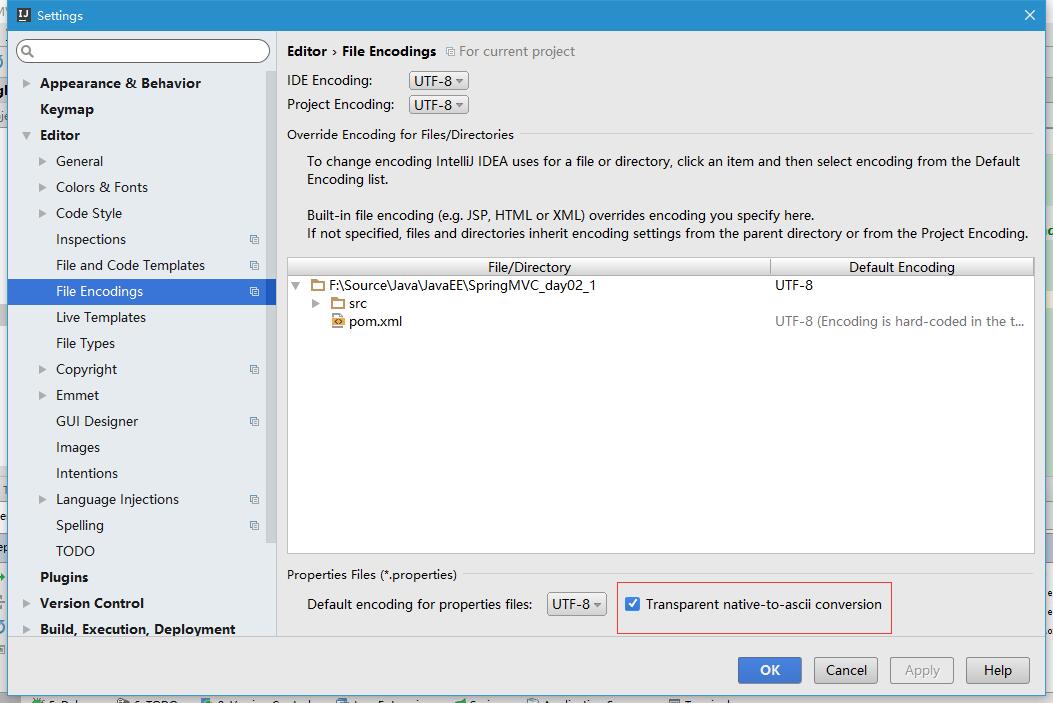
*<!-- 校验器 -->*<**bean id="validator"  
 class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"**>  
 *<!-- 校验器 -->* <**property name="providerClass" value="org.hibernate.validator.HibernateValidator"**/>  
 *<!-- 指定校验使用的资源文件，如果不指定则默认使用classpath下的ValidationMessages.properties -->* <**property name="validationMessageSource" ref="messageSource"**/>  
</**bean**>  
*<!-- 校验错误信息配置文件 -->*<**bean id="messageSource"  
 class="org.springframework.context.support.ReloadableResourceBundleMessageSource"**>  
 *<!-- 资源文件名 -->* <**property name="basenames"**>  
 <**list**>  
 <**value**>classpath:CustomValidationMessages</**value**>  
 </**list**>  
 </**property**>  
 *<!-- 资源文件编码格式 -->* <**property name="fileEncodings" value="utf-8"**/>  
 *<!-- 对资源文件内容缓存时间，单位秒 -->* <**property name="cacheSeconds" value="120"**/>  
</**bean**>

## 创建CustomValidationMessages

在classpath下创建CustomValidationMessages.properties

*#检验信息 product.name.length.error写在java代码中***product.name.length.error**=**商品名称的长度限制在1到30个字符  
product.createtime.is.notnull**=**请输入商品生产日期**

注意，在intellij idea中需要勾选这个选项：



否则在页面会造成乱码！

## 校验规则

需求：

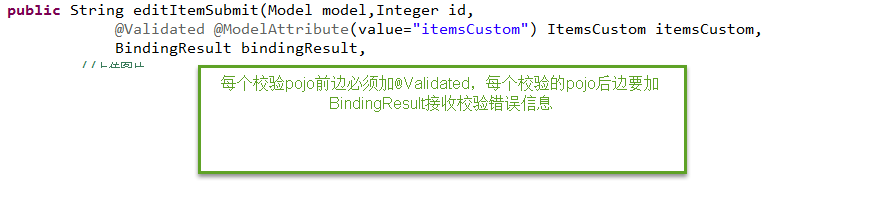
商品信息提交时校验 ，商品生产日期不能为空，商品名称长度在1到30字符之间

**public class** Product **implements** Serializable {  
  
 **private static final long *serialVersionUID*** = 1L;  
  
 **private** Integer **id**;   
  
 @Size(min = 1, max = 30, message = **"product.name.length.error"**)  
 **private** String **name**;  
  
 **private** Float **price**;  
  
 **private** String **pic**;  
  
 @NotNull(message = **"product.createtime.is.notnull"**)  
 **private** Date **createtime**;  
   
 **private** String **detail**;

## 捕获错误

需要修改controller方法，在要校验的pojo前边加上@Validated，

@RequestMapping(**"/editProductSubmit.action"**)  
**public** String editProductSubmit(Model model,  
 Integer id,  
 @Validated @ModelAttribute(**"product"**) ProductCustom productCustom,  
 BindingResult bindingResult,  
 *// 上传图片* MultipartFile pictureFile) **throws** Exception {



错误信息输出：

*//输出校验错误信息  
//如果参数绑定时有错***if** (bindingResult.hasErrors()) {  
 *//获取错误* List<ObjectError> errors = bindingResult.getAllErrors();  
 *//准备在页面输出errors，页面使用jstl遍历* model.addAttribute(**"errors"**, errors);  
 **for** (ObjectError error : errors) {  
 *//输出错误信息* System.***out***.println(error.getDefaultMessage());  
 }  
 *//如果校验错误，回到商品修改页面* **return "editProduct"**;  
}

## 在页面上展示错误

*<!-- 错误信息 -->*<**c:forEach items="${**errors**}" var="error"**>  
 **${**error.defaultMessage**}**<**br**/>  
</**c:forEach**>

## 分组校验

需求：

针对不同的controller方法通过分组校验达到个性化校验的目的，修改商品修改功能，只校验生产日期不能为空。

第一步：创建分组接口

*/\*\*  
 \* 校验分组，用于商品修改的校验  
 \* create:2016-10-30 20:51  
 \* <p>  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public interface** ValidGroup1 {  
 *//接口不定义方法，就是只标识 哪些校验 规则属于该 ValidGroup1分组*}

*/\*\*  
 \* 校验分组，用于其它的校验  
 \* create:2016-10-30 20:51  
 \* <p>  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public interface** ValidGroup2 {  
 *//接口不定义方法，就是只标识 哪些校验 规则属于该 ValidGroup1分组*}

第二步：定义校验规则属于哪个分组

@NotNull(message = **"{product.createtime.is.notnull}"**, groups = {ValidGroup1.**class**})  
**private** Date **createtime**;

第三步：在controller方法定义使用校验的分组

@RequestMapping(**"/editProductSubmit.action"**)  
**public** String editProductSubmit(Model model,  
 Integer id,  
 @Validated(value = {ValidGroup1.**class**}) @ModelAttribute(**"product"**) ProductCustom productCustom,  
 BindingResult bindingResult,  
 *// 上传图片* MultipartFile pictureFile) **throws** Exception {

# 统一异常处理

## 需求

一般项目中都需要作异常处理，基于系统架构的设计考虑，使用统一的异常处理方法。

系统中异常类型有哪些？

包括预期可能发生的异常、运行时异常（RuntimeException），运行时异常不是预期会发生的。

针对预期可能发生的异常，在代码手动处理异常可以try/catch捕获，可以向上抛出。

针对运行时异常，只能通过规范代码质量、在系统测试时详细测试等排除运行时异常。

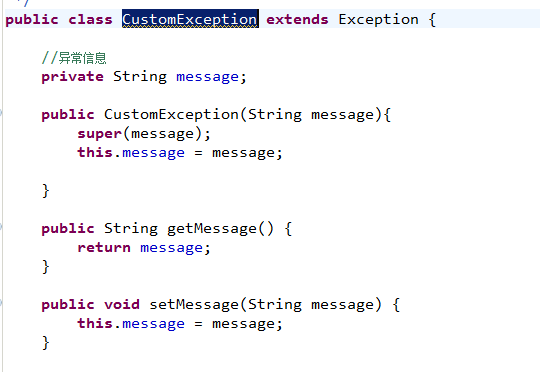
## 统一异常处理解决方案

### 定义异常

针对预期可能发生的异常，定义很多异常类型，这些异常类型通常继承于Exception。

这里定义一个系统自定义异常类：

CustomException，用于测试。

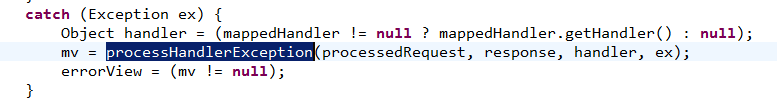


### 异常处理

要在一个统一异常处理的类中要处理系统抛出的所有异常，根据异常类型来处理。

统一异常处理的类是什么？

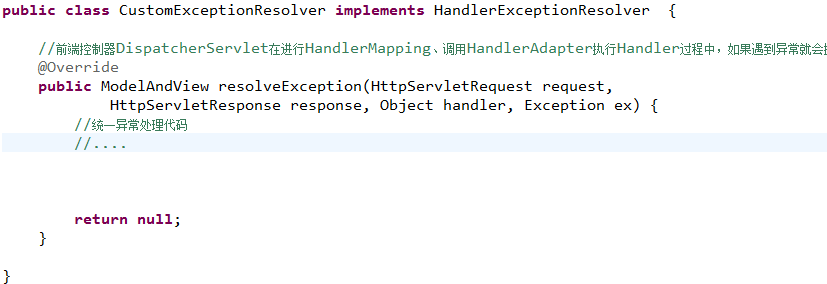
前端控制器DispatcherServlet在进行HandlerMapping、调用HandlerAdapter执行Handler过程中，如果遇到异常，进行异常处理。



在系统中自定义统一的异常处理器，写系统自己的异常处理代码。

#### 定义统一异常处理器类

统一异常处理器实现HandlerExceptionResolver接口。



#### 配置统一异常处理器

*<!-- 定义统一异常处理器 -->*<**bean class="cn.qdgxy.ssm.exception.CustomExceptionResolver"**/>

#### 异常处理逻辑

根据不同的异常类型进行异常处理。

系统自定义的异常类是CustomException，在controller方法中、service方法中、dao方法中手动抛出此类异常。

针对系统自定义的CustomException异常，就可以直接从异常类中获取异常信息，将异常处理在错误页面展示。

针对非CustomException异常，对这类重新构造成一个CustomException，异常信息为“未知错误”，此类错误需要在系统测试阶段去排除。

在统一异常处理器CustomExceptionResolver中实现上边的逻辑。

*/\*\*  
 \* 自定义异常处理器  
 \* create:2016-10-30 20:31  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** CustomExceptionResolver **implements** HandlerExceptionResolver {  
  
 */\*\*  
 \* 前端控制器DispatcherServlet在进行HandlerMapping、调用HandlerAdapter执行Handler过程中，如果遇到异常就会执行此方法  
 \*  
 \** ***@param request*** *request  
 \** ***@param response*** *response  
 \** ***@param handler*** *最终要执行的Handler，它的真实身份是HandlerMethod  
 \** ***@param ex*** *接收到的异常信息  
 \** ***@return*** *ModelAndView  
 \*/* @Override  
 **public** ModelAndView resolveException(HttpServletRequest request,  
 HttpServletResponse response, Object handler, Exception ex) {  
 *//输出异常* ex.printStackTrace();  
  
 *//统一异常处理代码  
 //针对系统自定义的CustomException异常，就可以直接从异常类中获取异常信息，将异常处理在错误页面展示* String message; *//异常信息* CustomException customException;  
  
 **if** (ex **instanceof** CustomException) { *//如果ex是系统自定义的异常，直接取出异常信息* customException = (CustomException) ex;  
 } **else** { *//针对非CustomException异常，对这类重新构造成一个CustomException，异常信息为“未知错误”* customException = **new** CustomException(**"未知错误"**);  
 }  
  
 *//错误 信息* message = customException.getMessage();  
  
 request.setAttribute(**"message"**, message);  
  
  
 **try** {  
 *//转向到错误页面* request.getRequestDispatcher(**"/WEB-INF/jsp/error.jsp"**).forward(request, response);  
 } **catch** (ServletException | IOException e) {  
 e.printStackTrace();  
 }  
  
 **return new** ModelAndView();  
 }  
  
}

#### 测试抛出异常由统一异常处理器捕获

可以在controller方法、service方法、dao实现类中抛出异常，要求dao、service、controller遇到异常全部向上抛出异常，方法向 上抛出异常throws Exception

*/\*\*  
 \* 根据商品id查询商品信息  
 \*  
 \** ***@param id*** *id  
 \** ***@return*** *商品包装类  
 \** ***@throws*** *Exception 异常  
 \*/*@Override  
**public** ProductCustom findProductById(Integer id) **throws** Exception {  
 Product product = **productMapper**.selectByPrimaryKey(id);  
  
 **if** (product == **null**) {  
 **throw new** CustomException(**"此商品信息不存在"**);  
 }  
  
 *//在这里随着需求的变量，需要查询商品的其它的相关信息，返回到controller* ProductCustom productCustom = **new** ProductCustom();  
 *//将product的属性拷贝到productCustom* BeanUtils.*copyProperties*(product, productCustom);  
  
 **return** productCustom;  
}

图解：

http请求

DispatcherServlet

Handler(Controller)

service

mapper

左边的箭头就是请求处理的过程

有异常向上抛出

有异常向上抛出

有异常向上抛出

DispatcherServlet统一try/catch捕获异常调用异常处理器（全系统只有一个）

异常处理器需要实现HandlerExceptionResolver接口

要求dao、service、controller遇到异常全部向上抛出异常，方法向 上抛出异常throws Exception

# RESTful支持

## 什么是RESTful

RESTful软件开发理念，RESTful对http进行非常好的诠释。

RESTful即Representational State Transfer的缩写。

综合上面的解释，我们总结一下什么是RESTful架构：

　　（1）每一个URI代表一种资源；

　　（2）客户端和服务器之间，传递这种资源的某种表现层；

　　（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

## url的RESTful实现

非RESTful的http的url：http://localhost:8080/items/editItems.action?id=1&....

RESTful的url是简洁的：http:// localhost:8080/items/editItems/1

参数通过url传递，rest接口返回json数据

### 需求

根据id查看商品信息，商品信息查看的连接使用RESTful方式实现，商品信息以json返回。

### 第一步更改DispatcherServlet配置

*<!-- RESTful -->*<**servlet**>  
 <**servlet-name**>SpringMVC\_RESTful</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 *<!-- 加载springmvc配置 -->* <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 *<!-- 配置文件的地址  
 如果不配置contextConfigLocation，  
 默认查找的配置文件名称classpath下的：servlet名称+"-serlvet.xml"即：springmvc-serlvet.xml  
 -->* <**param-value**>classpath:spring/springmvc.xml</**param-value**>  
 </**init-param**>  
</**servlet**>  
  
<**servlet-mapping**>  
 <**servlet-name**>SpringMVC\_RESTful</**servlet-name**>  
 *<!--  
 可以配置/ ，此工程 所有请求全部由springmvc解析，此种方式可以实现 RESTful方式，需要特殊处理对静态文件的解析不能由springmvc解析  
 可以配置\*.do或\*.action，所有请求的url扩展名为.do或.action由springmvc解析，此种方法常用  
 不可以/\*，如果配置/\*，返回jsp也由springmvc解析，这是不对的。  
 -->  
 <!-- RESTful配置为/ -->* <**url-pattern**>/</**url-pattern**>  
</**servlet-mapping**>

### 第二步参数通过url传递

*/\*\*  
 \* 根据商品id查看商品信息rest接口  
 \*  
 \** ***@param id*** *@PathVariable将url中的{}包起参数和形参进行绑定  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/*@RequestMapping(**"/viewItems/{id}"**) *//@RequestMapping中指定restful方式的url中的参数，参数需要用{}包起来***public**@ResponseBody  
ProductCustom productCustom(@PathVariable(**"id"**) Integer id) **throws** Exception {  
 *//调用 service查询商品信息* ProductCustom productCustom = **productService**.findProductById(id);  
  
 **return** productCustom;  
}

多个参数：

@RequestMapping(**"/viewItems/{id}/{name}"**)

@PathVariable(**"id"**) Integer id, @PathVariable(**"name"**) String name

### 设置静态资源解析

当DispatcherServlet拦截/开头的所有请求，对静态资源的访问就报错：



需要通过设置对静态资源进行解析.

*<!-- 静态资源 解析 -->*<**mvc:resources location="/WEB-INF/js/" mapping="/js/\*\*"**/>  
<**mvc:resources location="/WEB-INF/css/" mapping="/css/\*\*"**/>  
<**mvc:resources location="/WEB-INF/image/" mapping="/image/\*\*"**/>

访问/js/\*\*的url从工程下/WEB-INF/js/下解析。

# springmvc拦截器

## 拦截器的异常场合

用户请求到DispatherServlet中，DispatherServlet调用HandlerMapping查找Handler，HandlerMapping返回一个拦截的链儿（多个拦截），springmvc中的拦截器是通过HandlerMapping发起的。

在企业开发，使用拦截器实现用户认证（用户登陆后进行身份校验拦截），用户权限拦截。

## springmvc拦截器方法

*/\*\*  
 \* 测试拦截器1  
 \* create:2016-10-30 20:51  
 \* <p>  
 \* author 李欣  
 \*  
 \** ***@version*** *1.0  
 \*/*@Controller  
**public class** HandlerInterceptor1 **implements** HandlerInterceptor {  
  
 */\*\*  
 \* 在执行handler之前来执行的  
 \* <p>  
 \* 用于用户认证校验、用户权限校验  
 \*  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 **public boolean** preHandle(HttpServletRequest request,  
 HttpServletResponse response, Object handler) **throws** Exception {  
  
 }  
  
  
 */\*\*  
 \* 在执行handler返回modelAndView之前来执行  
 \* <p>  
 \* 如果需要向页面提供一些公用 的数据或配置一些视图信息，使用此方法实现 从modelAndView入手  
 \*  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param modelAndView*** *\** ***@throws*** *Exception  
 \*/* @Override  
 **public void** postHandle(HttpServletRequest request,  
 HttpServletResponse response, Object handler,  
 ModelAndView modelAndView) **throws** Exception {  
 }  
  
 */\*\*  
 \* 执行handler之后执行此方法  
 \* <p>  
 \* 作系统 统一异常处理，进行方法执行性能监控，在preHandle中设置一个时间点，在afterCompletion设置一个时间，两个时间点的差就是执行时长  
 \* <p>  
 \* 实现 系统 统一日志记录  
 \*  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@throws*** *Exception  
 \*/* @Override  
 **public void** afterCompletion(HttpServletRequest request,  
 HttpServletResponse response, Object handler, Exception ex)  
 **throws** Exception {  
 }  
  
}

## 测试拦截器

### 定义两个拦截器



### 配置拦截器

配置全局的拦截器，DispatcherServlet将配置的全局拦截器加载到所有的HandlerMapping。

在springmvc.xml中配置：

*<!--拦截器 -->*<**mvc:interceptors**>  
 *<!--多个拦截器,顺序执行 -->* <**mvc:interceptor**>  
 <**mvc:mapping path="/\*\*"**/>  
 <**bean class="cn.qdgxy.ssm.controller.interceptor.HandlerInterceptor1"**/>  
 </**mvc:interceptor**>  
   
 <**mvc:interceptor**>  
 <**mvc:mapping path="/\*\*"**/>  
 <**bean class="cn.qdgxy.ssm.controller.interceptor.HandlerInterceptor2"**/>  
 </**mvc:interceptor**></**mvc:interceptors**>

### 测试1 （1 号和2号都放行）

测试结果：

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor2...postHandle

HandlerInterceptor1...postHandle

HandlerInterceptor2...afterCompletion

HandlerInterceptor1...afterCompletion

总结：

执行preHandle是顺序执行。

执行postHandle、afterCompletion是倒序执行

### 测试2 （1 号放行和2号不放行）

测试结果：

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor1...afterCompletion

总结：

如果preHandle不放行，postHandle、afterCompletion都不执行。

只要有一个拦截器不放行，controller不能执行完成

### 测试3 （1 号不放行和2号不放行）

测试结果：

HandlerInterceptor1...preHandle

总结：

只有前边的拦截器preHandle方法放行，下边的拦截器的preHandle才执行。

### 日志拦截器或异常拦截器要求

将日志拦截器或异常拦截器放在拦截器链儿中第一个位置，且preHandle方法放行

## 拦截器应用（用户认证拦截）

### 需求

用户访问系统的资源(url)，如果用户没有进行身份认证，进行拦截，系统跳转登陆页面，如果用户已经认证通过，用户可以继续访问系统的资源。

### 用户登陆及退出功能开发

*/\*\*  
 \* 登陆和退出  
 \* create:2016-10-30 20:51  
 \* <p>  
 \* author 李欣  
 \*  
 \** ***@version*** *1.0  
 \*/*@Controller  
**public class** LoginController {  
  
 *//用户登陆提交方法* @RequestMapping(**"/login"**)  
 **public** String login(HttpSession session, String username, String password) **throws** Exception {  
 *//调用service校验用户账号和密码的正确性  
 //..  
  
 //如果service校验通过，将用户身份记录到session* session.setAttribute(**"username"**, username);  
 *//重定向到商品查询页面* **return "redirect:/product/findAllProduct.action"**;  
 }  
  
 *//用户退出* @RequestMapping(**"/logout"**)  
 **public** String logout(HttpSession session) **throws** Exception {  
  
 *//session失效* session.invalidate();  
 *//重定向到商品查询页面* **return "redirect:/product/findAllProduct.action"**;  
 }  
  
}

### 用户身份认证校验拦截器

拦截实现思路：

http请求url

如果url是公开地址（无需登录）

放行

如果用户在session中存在

放行

公开地址：

不需要认证即可访问的url

如果用户在session中不存在

登陆页面

### 拦截器

*/\*\*  
 \* 用户身份认证拦截器  
 \* create:2016-10-30 20:51  
 \* <p>  
 \* author 李欣  
 \*  
 \** ***@version*** *1.0  
 \*/***public class** LoginInterceptor **implements** HandlerInterceptor {  
  
 */\*\*  
 \* 在执行handler之前来执行的  
 \* <p>  
 \* 用于用户认证校验、用户权限校验  
 \*  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 **public boolean** preHandle(HttpServletRequest request,  
 HttpServletResponse response, Object handler) **throws** Exception {  
  
 *//得到请求的url* String url = request.getRequestURI();  
  
 *//判断是否是公开 地址  
 //实际开发中需要公开 地址配置在配置文件中  
 //...* **if** (url.contains(**"login.action"**)) {  
 *//如果是公开 地址则放行* **return true**;  
 }  
  
 *//判断用户身份在session中是否存在* HttpSession session = request.getSession();  
 String usercode = (String) session.getAttribute(**"username"**);  
 *//如果用户身份在session中存在放行* **if** (usercode != **null**) {  
 **return true**;  
 }  
  
 *//执行到这里拦截，跳转到登陆页面，用户进行身份认证* request.getRequestDispatcher(**"/WEB-INF/jsps/user/login.jsp"**).forward(request, response);  
  
 *//如果返回false表示拦截不继续执行handler，如果返回true表示放行* **return false**;  
 }

在springmvc.xml中配置拦截器：

*<!--拦截器 -->*<**mvc:interceptors**>  
 *<!--多个拦截器,顺序执行 -->* <**mvc:interceptor**>  
 <**mvc:mapping path="/\*\*"**/>  
 <**bean class="cn.qdgxy.ssm.controller.interceptor.HandlerInterceptor1"**/>  
 </**mvc:interceptor**>  
  
 <**mvc:interceptor**>  
 <**mvc:mapping path="/\*\*"**/>  
 <**bean class="cn.qdgxy.ssm.controller.interceptor.HandlerInterceptor2"**/>  
 </**mvc:interceptor**>  
  
 <**mvc:interceptor**>  
 *<!-- /\*\*可以拦截路径不管多少层 -->* <**mvc:mapping path="/\*\*"**/>  
 <**bean class="cn.qdgxy.ssm.controller.interceptor.LoginInterceptor"**/>  
 </**mvc:interceptor**>  
</**mvc:interceptors**>

# 学生练习

商品查询添加查询条件：商品名称、价格范围

添加商品功能开发

删除商品功能开发