# 包装类型POJO参数绑定

## 1.1需求：

商品查询Controller方法中实现商品查询条件的传入

注意：itemsCustomer和包装pojo中的属性一致即可。

## 1.2方法实现

方法一：在形参中，添加HttpServeltRequest request参数，通过request接收查询条件参数。

方法二：在形参中让包装的POJO接收查询条件参数

分析:

页面参数的特点：复杂、多样性（假如条件包括：用户账号、商品编号、订单信息、、、、）

如果将这些信息等全部放在简单的POJO（属性是简单类型）中，pojo类属性比较多，比较乱，建议使用包装类型的pojo，即pojo属性是pojo。

## 页面参数和Controller方法形参定义

通过 属性.属性 来设置name

<td>商品名称：<input name=*"itemsCustomer.name"* ></td>

<td><input type=*"submit"* value=*"查询"*></td>

Controller定义：

//商品的查询，修改不适用ModelAndView

@RequestMapping("/queryItems")

**public** ModelAndView queryItems(HttpServletRequest request,ItemsQueryVo itemsQueryVo) **throws** Exception {

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(itemsQueryVo);

ModelAndView modelAndView=**new** ModelAndView();

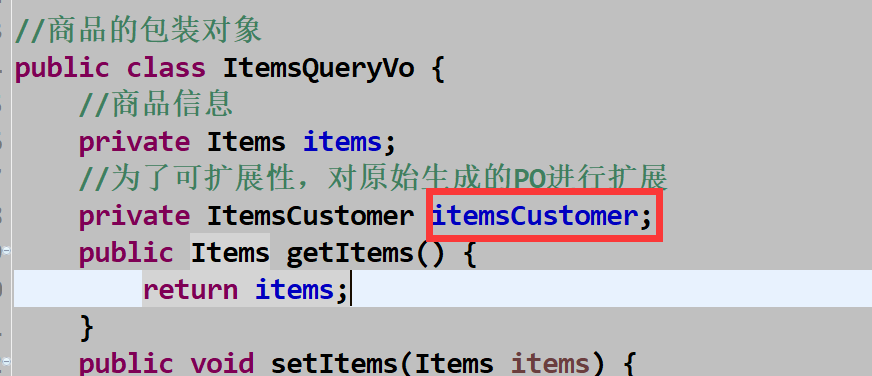
modelAndView.addObject("list",itemslist);

modelAndView.setViewName("items/itemslist");

**return** modelAndView;

}

页面中的必须和下边的po类中的属性对应：



# 集合类型的绑定

需求：

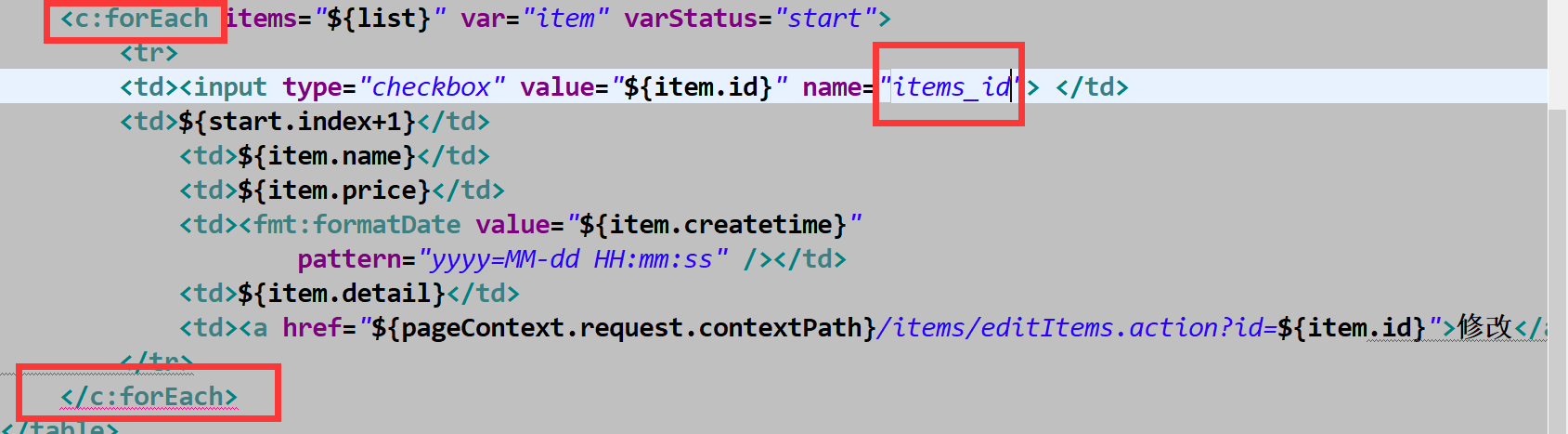
商品的删除，在用户界面选择对个商品，批量删除

## 2.1 数组绑定

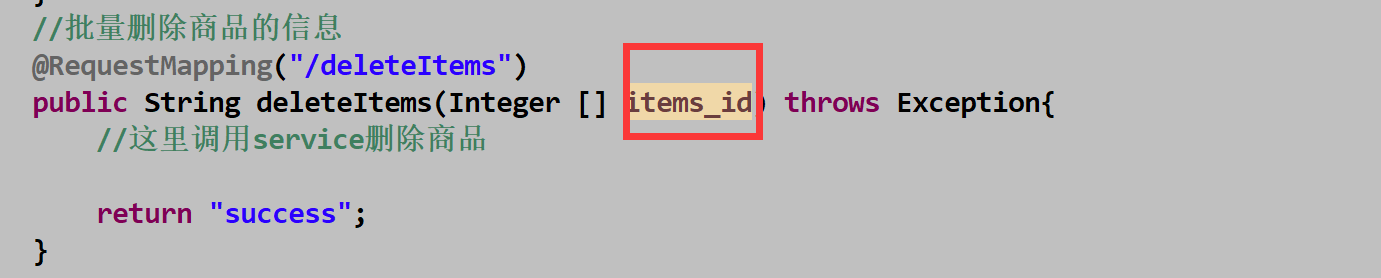
### 表现层的实现

关键：将页面选择(多选)的商品的id，传到controller方法的形参，方法的形参使用数组接收页面请求的多个商品的id.

页面定义：



### Controller方法定义：



**注意上边的参数名，一定要是一样的，但是页面传入的是forEach中的，所以传入的是多个items\_id.**

## 2.2 List绑定

需求:通常在批量提交数据时候，将提交的数据绑定到List<pojo>中，比如：成绩的录入（批量提交）。

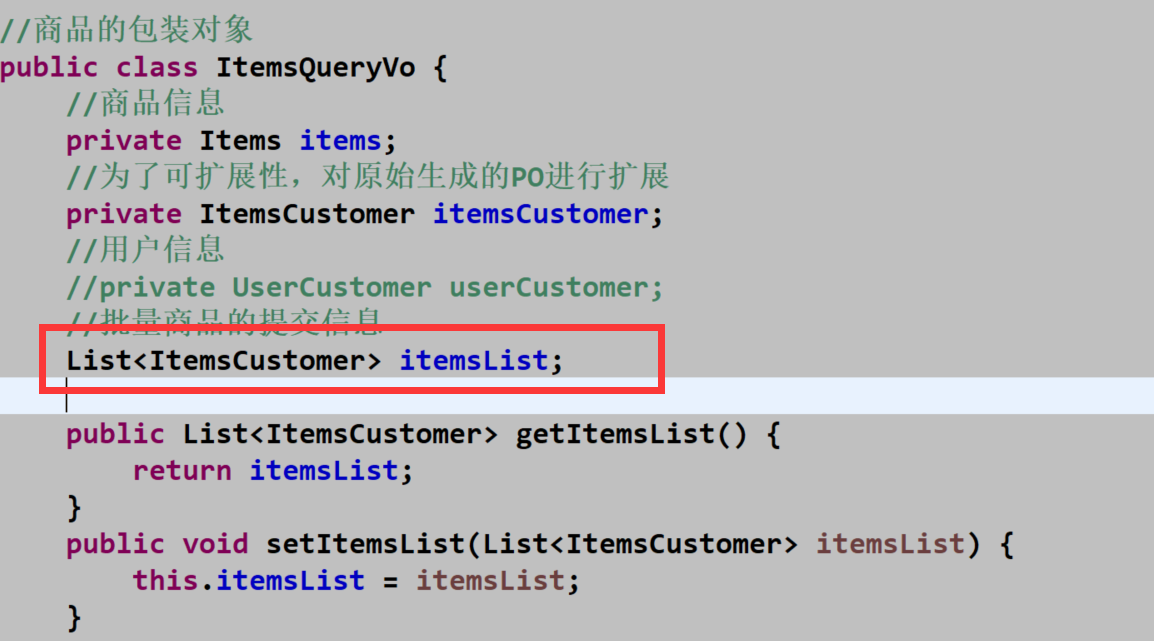
本例需求：批量修改商品的信息，在页面中输入多个商品的信息，将多个商品提交到Controller方法中。

### 表现层的实现：

controller方法定义：

1. 进入批量修改商品页面（参考商品列表实现）
2. 批量修改商品提交

使用List接受页面提交的批量数据，要通过包装的pojo接收，在包装pojo中定义List<pojo>属性：



//批量修改商品页面，将商品信息查询出来，在页面中可以编辑商品信息

@RequestMapping("/EditItemsquery")

**public** ModelAndView EditItemsquery(HttpServletRequest request,ItemsQueryVo itemsQueryVo) **throws** Exception {

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(itemsQueryVo);

ModelAndView modelAndView=**new** ModelAndView();

modelAndView.addObject("list",itemslist);

modelAndView.setViewName("items/EditItemsquery");

**return** modelAndView;

}

//批量修改商品提交

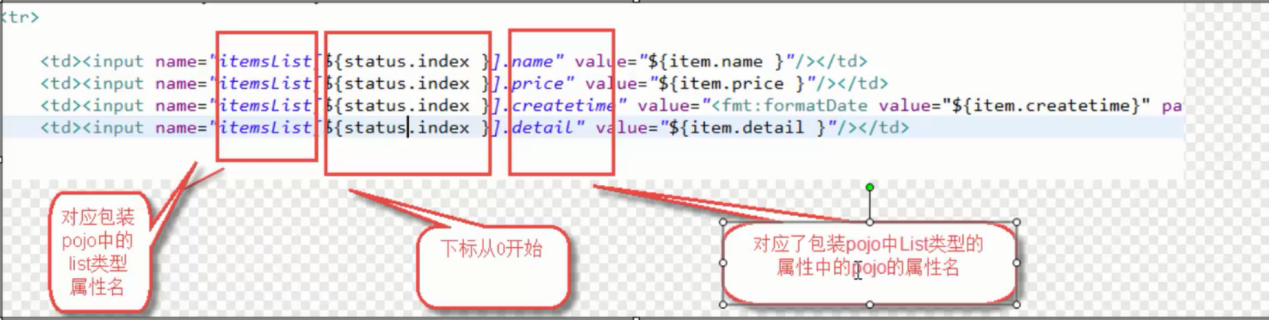
//ItemsQueryVo接受批量提交的商品信息，将其信息存储在ItemsQueryVo的itemsList属性中

**public** String EditItemsAllSubmit(ItemsQueryVo itemsQueryVo) **throws** Exception{

**return** "sucess";

}

页面定义：



## 2.3 map绑定

也是通过包装pojo中定义map类型属性。

在包装类型中定义map对象，并添加get/set方法，action使用包装类型接收，包装类型中定义map如下：

//map类型的定义

**private** Map<String,Object> itemsInfo=**new** HashMap<String,Object>();

**public** Map<String, Object> getItemsInfo() {

**return** itemsInfo;

}

**public** **void** setItemsInfo(Map<String, Object> itemsInfo) {

**this**.itemsInfo = itemsInfo;

}

页面的定义：

<tr>

<td><input type=*"text"* name=*"* itemsInfo *[*' *name*'*] "* value=*"*${item.name}*"*> </td>

<td><input type=*"text"* name=*"* itemsInfo *[*'price'*] "* value=*"*${item.price}*"*> </td>

<td><input type=*"text"* name=*"* itemsInfo *[*'details'*] "* value=*"*${item.detail}*"*> </td>

</tr>

# 3.校验

使用较多的是前端的校验，比如页面中js的校验，对于安全性较高的，需要服务端的校验。

服务端的校验：

控制层 controller：校验页面请求参数的合法性。在服务端控制层controller校验，不区分客户端的类型（浏览器、手机客户端、远程调用）。

业务层service（使用较多）：主要校验关键业务参数，仅限于service接口中使用的参数。

持久层 dao：一般是不校验的。

## 3.1 springmvc校验

springmvc使用hibernate的校验框架validation（和hibernate没有任何关系）

校验思路：

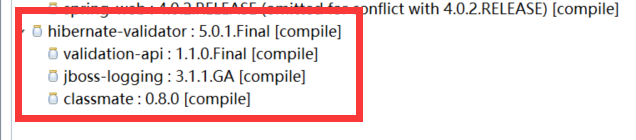
页面提交请求的参数，请求道controller方法中，使用validation校验，如果校验出错，将错误信息展示到页面。

具体需求：

商品修改，添加校验（校验商品名称长度、生产日期的非空校验），如果校验出错，在商品修改页面显示错误信息。

## 3.2 环境

hibernate的校验框架（hibernate-validate.jar）所需要的jar包：



## 3.3 配置校验器

<!-- 校验器 -->

<bean id=*"validator"* class=*"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"*>

<!-- 校验器 -->

<property name=*"providerClass"* value=*"org.hibernate.validator.HibernateValidator"*></property>

<!-- 指定校验使用的资源文件，在文件中配置校验错误信息，如果不指定则默认使用classpath下的validationMessages.properties -->

<property name=*"validationMessageSource"* ref=*"messageSource"*></property>

</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:CustomeValidationMessages</value>

</list>

</property>

<!-- 资源文件编码格式 -->

<property name=*"fileEncodings"* value=*"utf-8"*></property>

<!-- 对资源文件的内容缓存时间，单位秒 -->

<property name=*"cacheSeconds"* value=*"120"*></property>

</bean>

## 配置方方式2：（不用注解驱动）

<!-- 自定义webBinder -->

<bean id=*"customerBinder"* class=*"org.springframework.web.bind.support.ConfigurableWebBindingInitializer"*>

<property name=*"validator"* ref=*"validator"*></property>

</bean>

<!-- 注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*>

<property name=*"webBindingInitializer"* ref=*"customerBinder"*></property>

</bean>

<!-- 校验器 -->

<bean id=*"validator"* class=*"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"*>

<!-- 校验器 -->

<property name=*"providerClass"* value=*"org.hibernate.validator.HibernateValidator"*></property>

<!-- 指定校验使用的资源文件，在文件中配置校验错误信息，如果不指定则默认使用classpath下的validationMessages.properties -->

<property name=*"validationMessageSource"* ref=*"messageSource"*></property>

</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:CustomeValidationMessages</value>

</list>

</property>

<!-- 资源文件编码格式 -->

<property name=*"fileEncodings"* value=*"utf-8"*></property>

<!-- 对资源文件的内容缓存时间，单位秒 -->

<property name=*"cacheSeconds"* value=*"120"*></property>

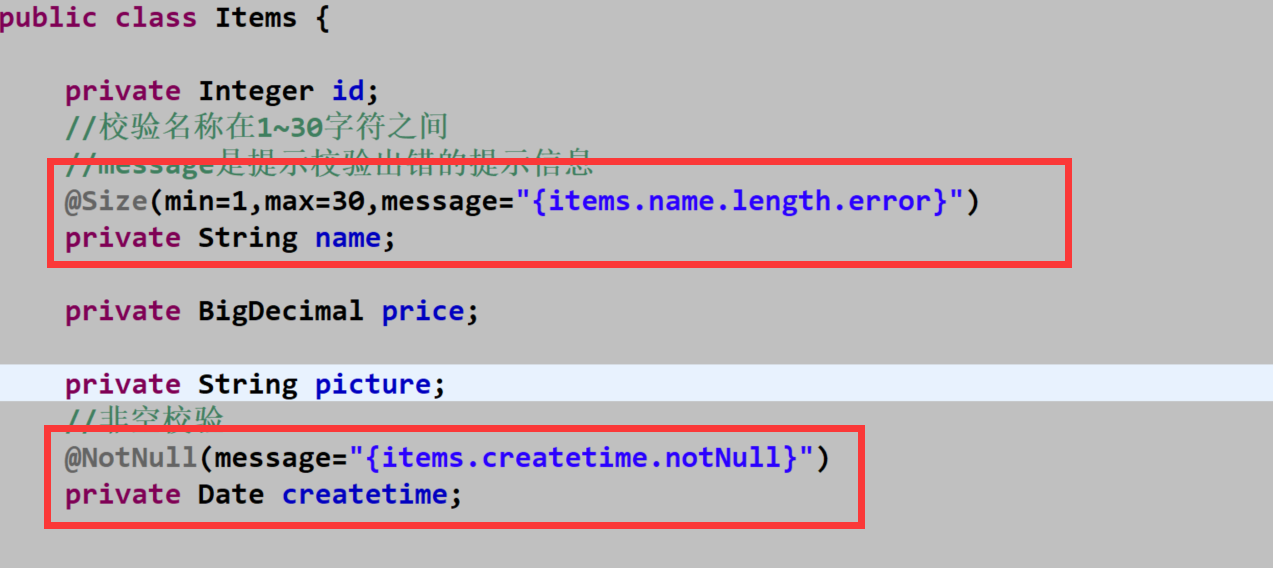
</bean>

## 3.4 将校验器注入处理器适配器中



## 3.5 添加校验规则

在ItemsCustomer.java中添加校验规则。

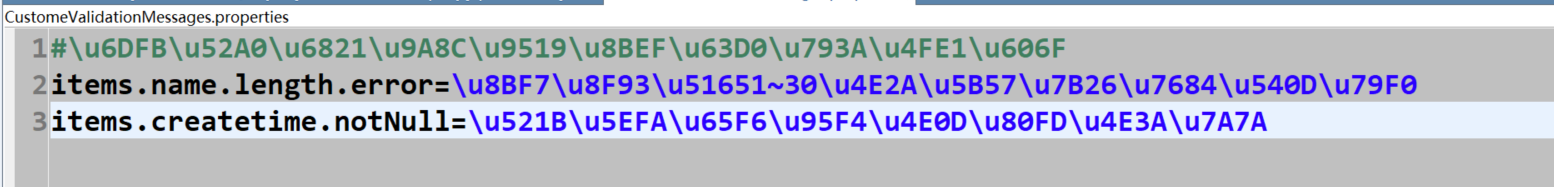


## 3.6 编写错误校验文件

CustomeValidationMessages2.properties：

Items.name.length.error=请输入1~30个字符的名称

Items.createtime.noteNull=创建时间不能为空



## 3.7 捕获校验信息

//在需要校验的pojo前添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息

//注意：@Validated和BindingResult bindingResult是配对出现的，并且顺序是固定的（一前一后）

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit( Model model,HttpServletRequest request,Integer id,

@Validated ItemsCustomer itemsCustomer,BindingResult bindingResult) **throws** Exception{

//获取校验信息

**if**(bindingResult.hasErrors()){

List<ObjectError> allErrors=bindingResult.getAllErrors();

**for**(ObjectError objectError:allErrors){

//输出错误信息

System.***out***.println(objectError.getDefaultMessage());

}

//将错误信息传到页面

model.addAttribute(“allerrors”, allErrors);

}

//调用service更新商品信息，页面需要将商品信息传递到此方法

itemsService.updateItems(id, itemsCustomer);

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

**return** "forward:queryItems.action";

}

## 3.8 页面显示校验信息

<!-- 显示错误信息 -->

<c:if test=*"*${allErrors!=null }*"*>

<c:forEach items=*"*${allErrors}*"* var=*"error"*>

${error.defaultMessage}

</c:forEach>

</c:if>

## 3.9 分组校验

### 3.9.1 需求

在pojo中定义校验规则，而pojo被多个controller方法所公用，当不同的controller对同一个pojo进行校验，但是每个controller需要不同的校验规则，该怎么办？

解决办法：

定义多个校验分组（其实是一个java接口），分组中定义有哪些规则。

每个controller方法使用不同的检验分组。

### 3.9.2校验分组的定义

//校验分组

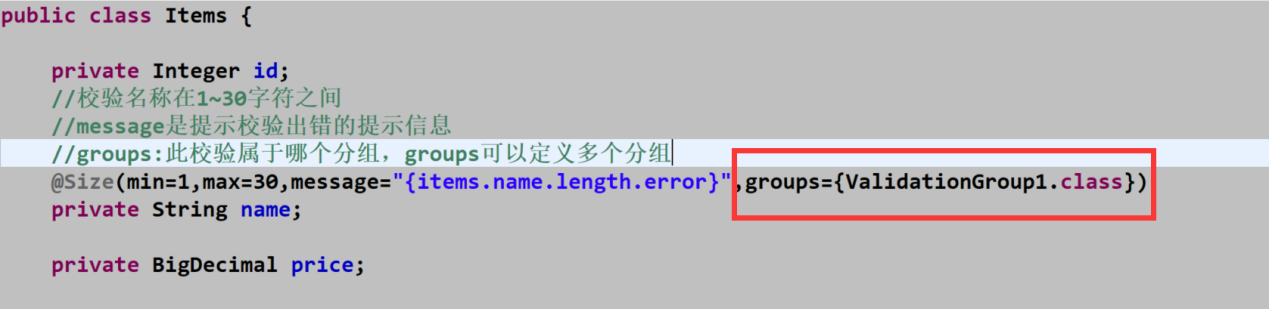
**public** **interface** ValidationGroup1 {

//在接口中不需要定义任何方法，因为它仅仅是对不同的校验规则进行分组

//此分组只校验商品名称的长度

}

### 3.9.3 在校验规则中添加分组



### 3.9.4 在controller中使用指定分组的校验

//商品信息修改的提交

//在需要校验的pojo前添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息

//注意：@Validated和BindingResult bindingResult是配对出现的，并且顺序是固定的（一前一后）

//value={ValidationGroup1.class}指定使用ValidationGroup1这个分组的校验

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit(Model model, HttpServletRequest request,Integer id,

@Validated(value={ValidationGroup1.**class**}) ItemsCustomer itemsCustomer,BindingResult bindingResult) **throws** Exception{

//获取校验信息

**if**(bindingResult.hasErrors()){

List<ObjectError> allErrors=bindingResult.getAllErrors();

**for**(ObjectError objectError:allErrors){

//输出错误信息

System.***out***.println(objectError.getDefaultMessage());

}

//将错误信息传到页面

model.addAttribute("allErrors", allErrors);

//出错重新到商品的修改页面

**return** "items/editItems";

}

//调用service更新商品信息，页面需要将商品信息传递到此方法

itemsService.updateItems(id, itemsCustomer);

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

**return** "forward:queryItems.action";

}

# 数据回显

## 4.1 什么是数据回显

提交后，如果出现错误，将刚才提交的数据回显到刚才提交的页面

## 4.2pojo数据回显方法

1.spring默认对pojo数据进行回显

pojo数据传入controller方法后，springmvc会自动将pojo的数据放到request域，key等于pojo类型（首字母小写）

//@ModelAttribute("items")可以指定pojo回显到页面在request中的key

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit(Model model, HttpServletRequest request,Integer id,

@ModelAttribute("items") @Validated(value={ValidationGroup1.**class**}) ItemsCustomer itemsCustomer,BindingResult bindingResult) **throws** Exception{

//获取校验信息

2.@ModelAttribute可以将方法的返回值传回到页面

example:

1. 在商品的查询列表页面，通过商品类型查询商品信息。
2. 在controller中定义商品查询的方法，最终将商品的类型传到页面。

//商品分类

//itemstypes表示最终将方法返回值放在request中的key

@ModelAttribute("itemstypes")

**public** Map<String ,String>getItemsRType(){

Map<String,String> itemsTypes=**new** HashMap<String,String>();

itemsTypes.put("101","数码");

itemsTypes.put("102","母婴");

**return** itemsTypes;

}

页面上可以得到itemstypes的数据

查询条件：

<table width=*"100%"* border=*"1px solid red"*>

<tr>

<td width=*"300px"*>商品名称：<input name=*"itemsCustomer.name"* ></td>

<td ><input type=*"button"* onclick="queryItems()" value=*"查询"*><input type=*"button"* onclick="deleteItems()" value=*"批量删除"*></td>

<td width=*"300px"*>商品类型：<input name=*"itemstypes"* ></td>

<td><select naem=*"itemstype"*>

<c:forEach items=*"*${itemtypes }*"* var=*"itemtype"*>

<option value=*"*${itemtype.key }*"*>${itemtype.value }</option>

</c:forEach>

</select></td>

</tr>

1. 最简单的方法是使用model，可以不用@ModelAttribute

//商品信息修改的提交

//在需要校验的pojo前添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息

//注意：@Validated和BindingResult bindingResult是配对出现的，并且顺序是固定的（一前一后）

//value={ValidationGroup1.class}指定使用ValidationGroup1这个分组的校验

//@ModelAttribute("items")可以指定pojo回显到页面在request中的key

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit(Model model, HttpServletRequest request,Integer id,

@ModelAttribute("items") @Validated(value={ValidationGroup1.**class**}) ItemsCustomer itemsCustomer,BindingResult bindingResult) **throws** Exception{

//获取校验信息

**if**(bindingResult.hasErrors()){

List<ObjectError> allErrors=bindingResult.getAllErrors();

**for**(ObjectError objectError:allErrors){

//输出错误信息

System.***out***.println(objectError.getDefaultMessage());

}

//将错误信息传到页面

model.addAttribute("allErrors", allErrors);

//可以直接使用model将提交pojo回显到页面

model.addAttribute("item",itemsCustomer);

//出错重新到商品的修改页面

**return** "items/editItems";

}

//调用service更新商品信息，页面需要将商品信息传递到此方法

itemsService.updateItems(id, itemsCustomer);

//return "sucess";

//重定向到商品的查询列表

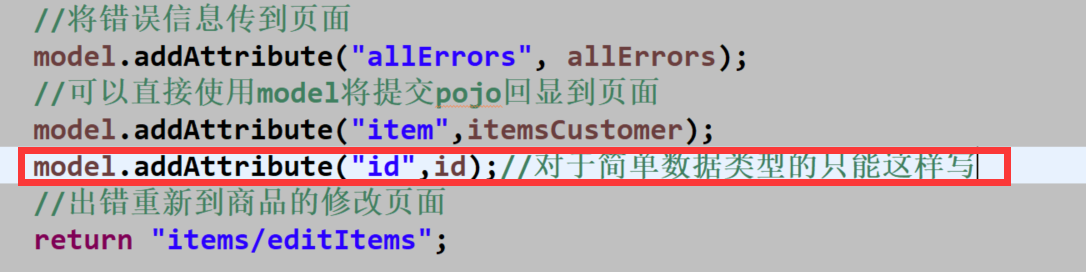
//return "redirect:queryItems.action";

//forward转发

**return** "forward:queryItems.action";

}

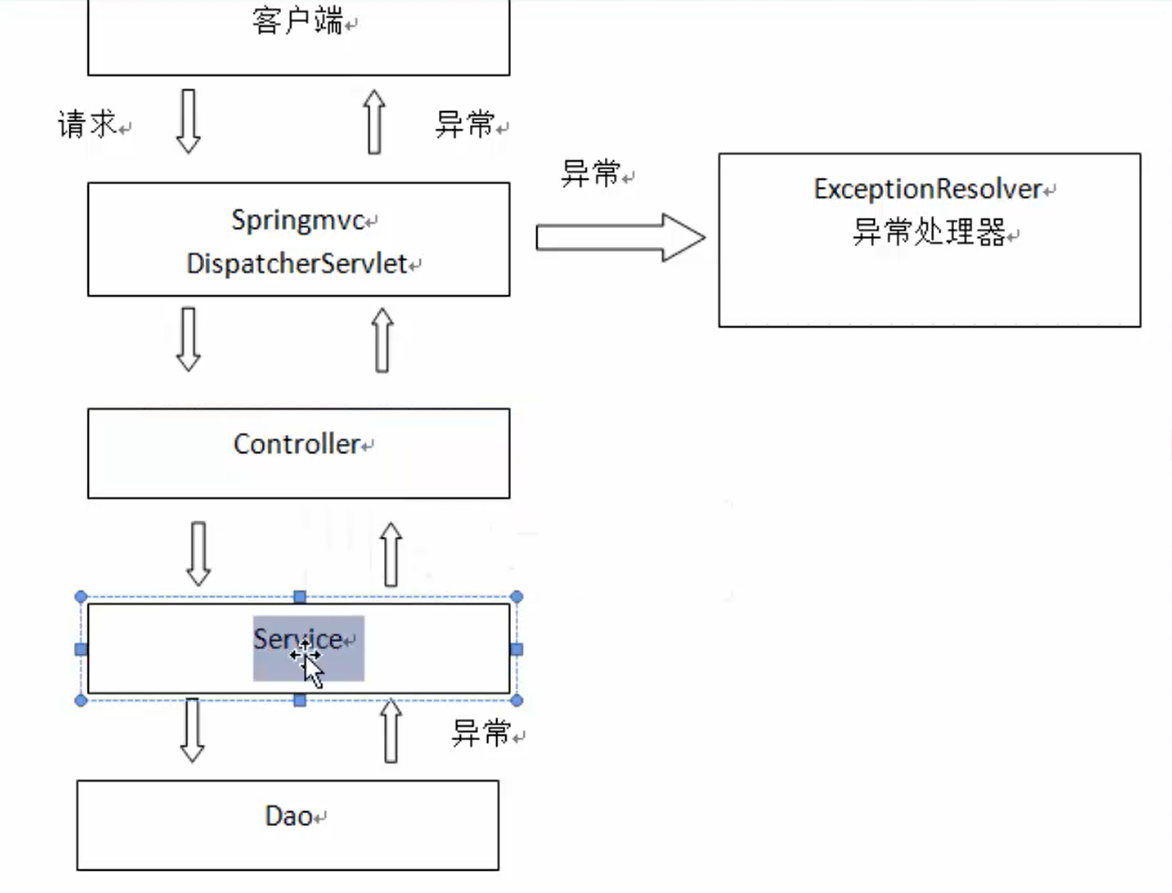
4.3简单类型的数据回显，使用model



# 4，异常处理

系统中异常包括两类，预期异常和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的dao、service、controller、出现都通过throws Exception 向上抛出异常，最后又springmvc前端控制器交由异常处理器进行异常处理，如图：



springmvc提供全局处理器（一个系统只有一个 异常处理器）进行统一异常处理。

## 4.1.自定义异常类

对不同的异常类型定义异常类，继承Exception

//针对一起的异常，在程序中抛出此异常的类

**public** **class** CustomerException **extends** Exception{

//异常信息

**public** String message;

**public** CustomerException(String msg){

**super**(msg);

**this**.message=msg;

}

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

## 4.2全局异常处理器

思路：

遇到异常时候，在程序中手动抛出，dao抛给service，service给controller，controller给前端控制器，前端控制器调用全局异常处理器。

全局异常处理器思路为：

解析出当前的异常类型，如果给异常类型是自定义的异常，直接取出异常信息，在错误的页面展示。

如果不是系统自定义的异常，构造一个自定义的异常类型（信息为“未知错误”）

//全局异常处理器

**public** **class** CustomerExceptionResolve **implements** HandlerExceptionResolver {

@Override

**public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,

Exception ex) {//ex是系统抛出的异常

// handler就是处理器适配器要执行的Handler对象（只有method方法）

/\* String message=null;

if(ex instanceof CustomerException){

//解析出当前的异常类型，如果给异常类型是自定义的异常，直接取出异常信息，在错误的页面展示。

//如果是自定义的异常

message=((CustomerException)ex).getMessage();

}else{

//如果不是系统自定义的异常，构造一个自定义的异常类型（信息为“未知错误”）

message="未知错误！";

}\*/

//上述的代码能够简化成为下边的

CustomerException ce=**null**;

**if**(ex **instanceof** CustomerException){

ce=(CustomerException)ex;

}**else**{

ce=**new** CustomerException("未知错误！");

}

//错误信息

String message=ce.getMessage();

ModelAndView modelAndView =**new** ModelAndView();

//将错误信息传到页面

modelAndView.addObject("message",message);

//定向到错误页面

modelAndView.setViewName("error");

**return** modelAndView;

}

}

## 4.3错误页面

<body>

<h1>${message}</h1>

</body>

</html>

4.4在springmvc.xml中配置全局异常处理器

<!-- 全局异常处理器,只要实现HandlerExceptionResolver接口，就是全局的异常处理器 -->

<bean class=*"com.wbs.exception.CustomerExceptionResolve"*> </bean>

## 4.5 异常的测试

在controller、service、dao任意一处手动抛出异常。

如果是程序中手动抛出的异常，在错误页面显示错误信息，如果不是手动抛出的异常，说明是一个运行时异常，只显示“未知错误”。

1. 在商品的修改的controller中抛出异常

//商品信息修改页面的展示

@RequestMapping("/editItems")

//@RequestParam里面指定request传入参数名称和形参进行绑定

//required指定参数是否必须要传入

//通过defaultValue可以设置默认值，如果id参数没有传入，将默认值和形参进行绑定

**public** String editItems(Model model,@RequestParam(value="id") Integer items\_id) **throws** Exception{

//调用service，根据id查询商品信息

ItemsCustomer itemsCustomer=itemsService.findItemsById(items\_id);

//在这里判断商品是否为空,根据Id没有查询到商品的信息，抛出异常，提示用户信息不存在

**if**(itemsCustomer==**null**){

**throw** **new** CustomerException("修改的商品的信息不存在");

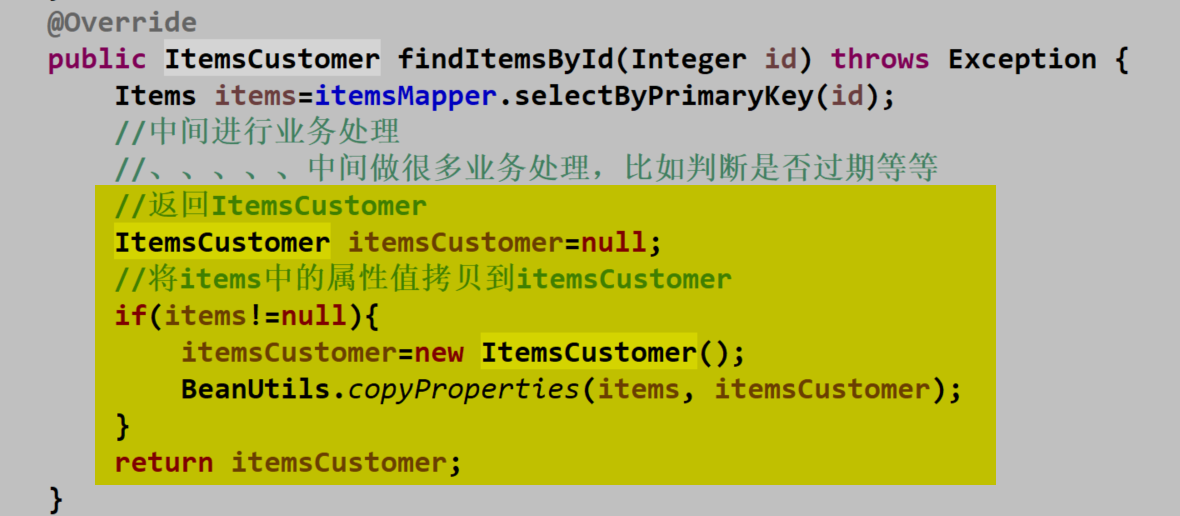
}

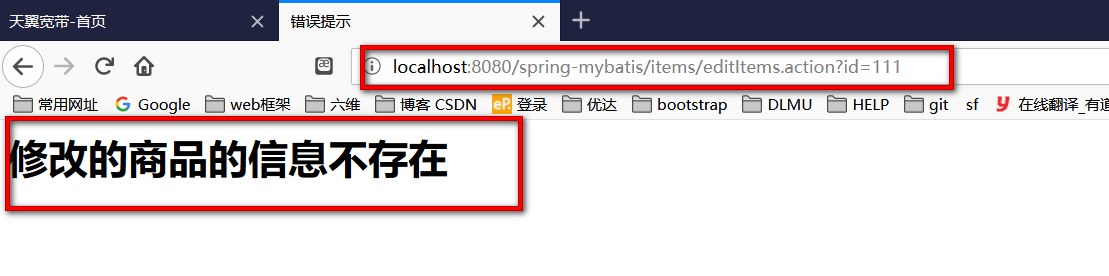
//通过形参中的model将model数据传到页面,相当于mav.addObject("list",itemslist)方法

model.addAttribute("itemsCustomer",itemsCustomer);

**return** "items/editItems";

}





通过上边的测试，测试成功。

1. service中抛出异常

如果与业务相关的异常，建议在service中抛出异常。

与业务功能没有关系的异常，建议在controller中抛出异常，dao层一般不处理异常。

@Override

**public** ItemsCustomer findItemsById(Integer id) **throws** Exception {

Items items=itemsMapper.selectByPrimaryKey(id);

**if**(items==**null**){

**throw** **new** CustomerException("商品信息不存在~~~~~~~");

}

//中间进行业务处理

//、、、、、中间做很多业务处理，比如判断是否过期等等

//返回ItemsCustomer

ItemsCustomer itemsCustomer=**null**;

//将items中的属性值拷贝到itemsCustomer

**if**(items!=**null**){

itemsCustomer=**new** ItemsCustomer();

BeanUtils.*copyProperties*(items, itemsCustomer);

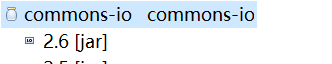
}

**return** itemsCustomer;

}

# 5.上传图片

加入上传图片的jar包： 解析器中会使用到下边的jar包



在修改商品页面中，上传商品的图片功能

## 5.1 springmvc中对多部件进行类型解析

在form表单中提交entype=”multipart/form-data”的数据时候，需要springmvc对这种类型的数据进行解析。

需要在springmvc中配置multipart类型的解析器

<!-- 文件上传 -->

<bean class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- s设置上传文件的最大尺寸为5M -->

<property name=*"maxUploadSize"*>

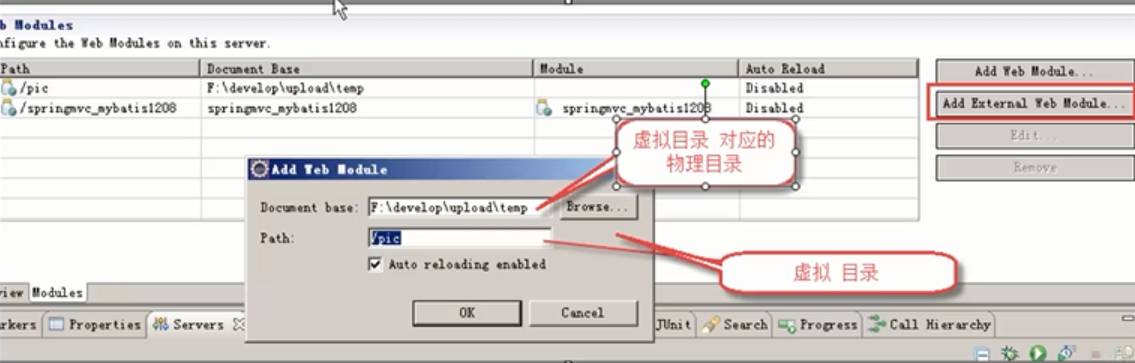
<value>5242880</value>

</property>

</bean>

## 5.2创建图片的虚拟目录，来存储图片

图形界面配置：



也可以直接修改tomcat的配置文件：  
修改tomcat中的conf/server.xml文件，添加虚拟目录：



注意：在建立的虚拟目录中，一定将图片目录分级创建（提高IO性能），一般采用按照日期（年、月、日）分级创建。

## 5.3修改商品的方法

修改商品的controller方法：

//商品信息修改的提交

//在需要校验的pojo前添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息

//注意：@Validated和BindingResult bindingResult是配对出现的，并且顺序是固定的（一前一后）

//value={ValidationGroup1.class}指定使用ValidationGroup1这个分组的校验

//@ModelAttribute("items")可以指定pojo回显到页面在request中的key

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit(Model model, HttpServletRequest request,Integer id,

@ModelAttribute("items") @Validated(value={ValidationGroup1.**class**}) ItemsCustomer itemsCustomer,

BindingResult bindingResult,

MultipartFile items\_pic//接受商品的图片，页面中的上传文件的name必须和这个是一样的

) **throws** Exception{

//获取校验信息

**if**(bindingResult.hasErrors()){

List<ObjectError> allErrors=bindingResult.getAllErrors();

**for**(ObjectError objectError:allErrors){

//输出错误信息

System.***out***.println(objectError.getDefaultMessage());

}

//将错误信息传到页面

model.addAttribute("allErrors", allErrors);

//可以直接使用model将提交pojo回显到页面

model.addAttribute("item",itemsCustomer);

model.addAttribute("id",id);//对于简单数据类型的只能这样写

//出错重新到商品的修改页面

**return** "items/editItems";

}

//上传图片

//得到图片的原始名称

String origin\_pic\_name=items\_pic.getOriginalFilename();

**if**(items\_pic!=**null &&** origin\_pic\_name！=null && origin\_pic\_name.length()>0){

//存储图片的物理路径

String pic\_path="F:\\develop\\upload\\temp\\";

//生成新的图片名称,UUID.randomUUID()是一个 随机数

String newFileName=UUID.*randomUUID*()+origin\_pic\_name.substring(origin\_pic\_name.lastIndexOf("."));

//新的图片

File newFile=**new** java.io.File(pic\_path+newFileName);

//将内存中的数据写入磁盘

items\_pic.transferTo(newFile);

//上传成功的话，将图片名称写到itemsCustomer中

itemsCustomer.setPicture(newFileName);

}

//调用service更新商品信息，页面需要将商品信息传递到此方法

itemsService.updateItems(id, itemsCustomer);

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

**return** "forward:queryItems.action";

}

## 5.4EditItems.jsp页面：

<tr>

<td>商品图片</td>

<td>

<c:if test=*"*${itemsCustomer.picture!=null }*"*>

<img src=*"/pic/*${itemsCustomer.picture}*"* width=*100* height=*100*><br>

</c:if>

<input type=*"file"* name=*"items\_pic"*><!-- 上传文件的选项，name必须要和controller中的参数中的名称一样 -->

</td>

</tr>

# 6.json数据交互

## 6.1为什么要用json数据交互

json数据格式在前台的页面中、接口调用中较为常用，json格式较为简单，解析比较简单。

比如：webservice接口，传输json数据。

## 6.2springmvc进行json交互

客户端请求

请求的是json串,需要指定

contenType=application/json

@RequestBody将json串转换成java对象

@ResponsBody将将java对象换换成json串输出

请求的是key/value

contenType=application/x-www-form-urlencoded

不需要@RequestBody将json串转换成java对象

@ResponsBody将将java对象换换成json串输出

**最后都输出json数据，在前端页面方便对请求的结果进行解析**

1. 请求json、输出json，要求请求的是json串，所以在前端页面中需要将请求的内容换换成json，不太方便。
2. 请求的是key/value，输出的是json，此方法比较常用，请求比较简单。

## 6.3环境准备

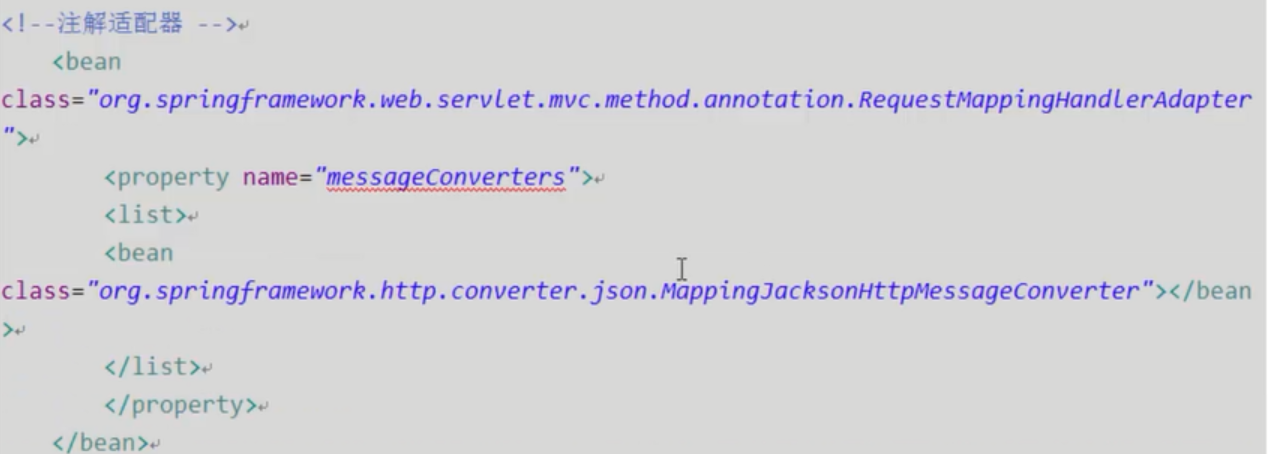
json的包如下：

springmvc中使用Jackson的包进行json转换（@RequestBody\@ResponseBody使用下边的包进行json转换），如下：



## 6.4 配置json转换器

在注解适配器中加入messageConverters



注意：如果使用<mvc:annotation-driven/>则不用定义上边的内容。

## 6.5json交互测试

### 6.5.1输入json串，输出json串

#### 6.5.1.1 jsp页面

<title>json交互测试</title>

<script type=*"text/javascript"* src=*"*${pageContext.request.contextPath}*/js/jquery.js"*></script>

<script type=*"text/javascript"*>

//请求json，输出json

**function** requestJson(){

$.ajax({

type:post,

url:'${pageContext.request.contextPath}/requestJson.action',

contentType:'application/json;charset=utf-8',

//数据格式是字符串,商品信息

data:'{"name":"手机","price":"99"}',

success:**function**(data){//返回json结果

alert(data);

}

})

}

//请求key/value，输出json

**function** responseJson(){

}

</script>

</head>

<body>

<input type=*"button"* onclick="requestJson()" value=*"请求json，输出json"*>

<input type=*"button"* onclick="responseJson()" value=*"请求key/value，输出json"*>

</body>

#### 6.5.2 controller方法

@Controller

//json交互测试

**public** **class** JsonTest {

//请求json（商品信息），输出json（商品信息）

@RequestMapping("requestJson")

//@RequestBody将请求的商品信息的json串转成ItemsCustomer对象

//@ResponseBody将ItemsCustomer对象转成json串输出

**public** @ResponseBody ItemsCustomer requestJson(@RequestBody ItemsCustomer itemsCustomer){

**return** itemsCustomer;

}

}

使用ｊＱｕｅｒｙ的ajax提交json串，对输出的json结果进行解析

### 6.5.2输入key/value，输出json串

#### 6.5.2.1 jsp页面

使用jquery的Ajax请求key/value串，对输出的json串进行解析

//请求key/value，输出json

**function** responseJson(){

$.ajax({

type:'post',

url:'/spring-mybatis/responseJson.action',

//请求的是key/value，这里不需要指定contextType,因为默认就是key/value的类型

//contentType:'application/json;charset=utf-8',

//数据格式是json字符串,商品信息

data:'name=手机&price=999',

success:**function**(data){//返回json结果

alert(data);

}

});

}

#### 6.5.2.2 controller方法

//请求key/value，输出json

@RequestMapping("/responseJson")

//@ResponseBody将ItemsCustomer对象转成json串输出

**public** @ResponseBody ItemsCustomer responseJson( ItemsCustomer itemsCustomer){

System.***out***.println("aaaa");

**return** itemsCustomer;

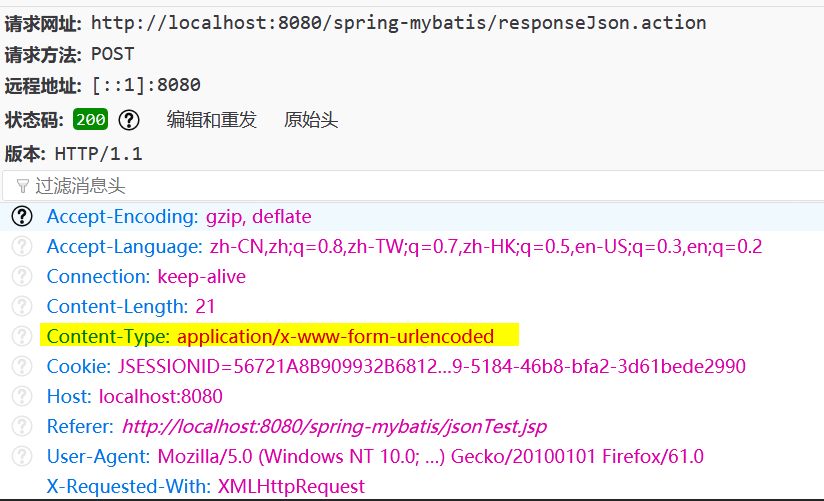
}

#### 6.5.2.3测试

响应的结果中返回了json格式的数据



请求的头中content-Type=application/x-www-form-urlencoded



# 7.RESTful支持

## 7.1什么是RESTful

RESTful是一个开发的理念，是对HTTP很好的一个诠释

1. 对url进行规范，都写成RESTful格式的url
   1. 非REST的URL：<http://localhost:80/aaa.action?id=1&type=T01>
   2. REST 的URL：<http://localhost:80/aaa/T01>

特定：url简洁，将参数通过url传到服务端

1. 对HTTP的方法进行规范

RESTful不管是添加、删除、修改，用的url是一致的，如果要进行删除，设置http的方法为delete，同理添加。。。。。。

1. 后台controller方法：判断http方法，如果是delete执行删除，如果是post，执行添加
2. 对HTTP的context Type进行规范
   1. 请求是指定context Type，要json数据，设置成为json格式的type

## 7.2REST的例子

### 需求：

查询商品的信息，返回json数据

### controller

定义方法，进行url映射，使用REST风格的url，将商品信息的id传入controller

输出json，使用@ResponseBody ，将Java对象输出json

//查询商品信息，输出json

@RequestMapping("/itemsView/{id}")

///itemsView/{id}中的id表示将这个位置的参数传到@PathVariable指定的名称中

**public** @ResponseBody ItemsCustomer itemsView(@PathVariable("id") Integer id) **throws** Exception{

//调用service查询商品信息

ItemsCustomer itemsCustomer =itemsService.findItemsById(id);

**return** itemsCustomer;

}

### 需要先配置号RESTful的前端控制器

<!-- 配置前端控制器，RESTful格式的 -->

<servlet>

<servlet-name>springmvc\_RESTful</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- contextConfigLocation配置spring加载的配置文件（配置处理器映射器、适配器等）

如果不配置contextConfigLocation，默认加载的是/WEB-INF/servlet名称-servlet.xml(springmvc-servlet.xml)

-->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/springmvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>springmvc\_RESTful</servlet-name>

<!-- 方法2；/ 所有访问的action都由DispatcherServlet进行解析，对于静态的文件不让DispatcherServlet进行解析

使用此种方法可以实现RESTful风俗的url -->

<url-pattern>/</url-pattern>

</servlet-mapping>

配置前端控制器的url-partten中指定/后，对静态资源的解析出现了问题，如下：



这个时候，需要在springmvc.xml中添加静态资源解析方法 ：

<!-- 静态资源的解析，包括：css js img -->

<mvc:resources location=*"/js/"* mapping=*"/js/\*\*"*/>

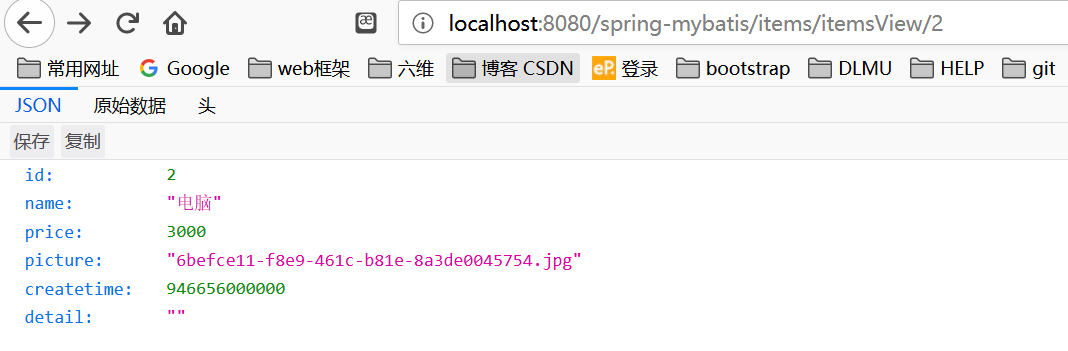
<!--\*\*是包括/js下边的所有文件-->

<mvc:resources location=*"/img/"* mapping=*"/img/\*\*"*/>

配置好之后再次访问即可成功，既可以访问RESTful类型的，又可以访问静态类型的，

如下：

**REST类型**



**静态类型**



# 8.拦截器

## 8.1拦截器定义

定义拦截器，实现HandlerInterceptor接口，接口中提供三个方法：

//测试拦截器1

**public** **class** HandlerIntercepter1 **implements** HandlerInterceptor {

@Override

//执行时机是在进入Handler之前

//用于身份认证、身份授权（登录认证，权限校验）

//比如身份认证，如果认证未通过表示用户未登录，需要此方法拦截，不在往下执行。

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

//return false表示拦截，return true表示放行

**return** **false**;

}

@Override

//进入Handler之后，在返回MOdelAndView之前执行

//应用场景从ModelAndView出发，将公用的模型数据（比如菜单导航）在这里传到视图，也可以在这里统一指定视图

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

}

@Override

//执行Handler完成，执行此方法

//使用统一的异常处理，用于统一的日志处理，

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

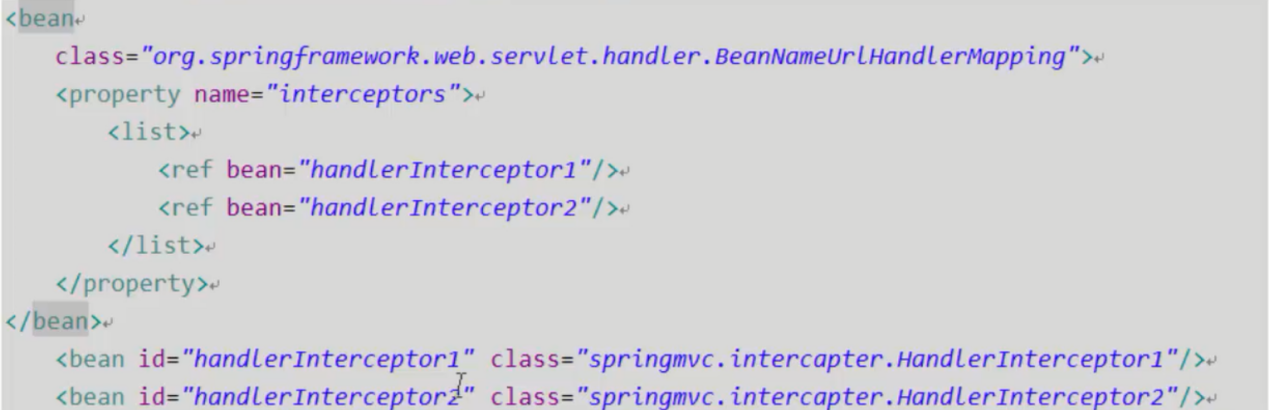
}

}

## 8.2拦截器的配置

**8.2.1 springmvc拦截器针对HandlerMapping进行拦截处理。**

如果在某个HandlerMapping中配置拦截器，经过该HandlerMapping映射成功的Handler最终才使用该拦截器。



一般不推荐使用。

**8.2.2 springmvc配置类似全局的拦截器**

springmvc框架将配置的类似全局的拦截器注入到每个HandlerMapping中。

<!-- 拦截器 -->

<mvc:interceptors>

<!--多个拦截器，顺序执行 -->

<mvc:interceptor>

<!-- /\*\*表示拦截url，包括所有的url路径，/\*表示只拦截跟的url -->

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"com.wbs.intercrptor.HandlerIntercepter1"*></bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"com.wbs.intercrptor.HandlerIntercepter2"*></bean>

</mvc:interceptor>

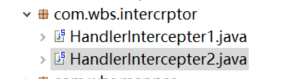
</mvc:interceptors>

## 8.3拦截器的测试：

### 8.3.1测试需求

测试多个拦截器各个执行时机

### 8.3.2编写两个拦截器







### 8.3.3两个拦截器都放行

测试打印结果：

HandlerIntercepter1 preHandle

HandlerIntercepter2 preHandle

HandlerIntercepter2 postHandle

HandlerIntercepter1 postHandle

HandlerIntercepter2 afterCompletion

HandlerIntercepter1 afterCompletion

总结：

**1，preHandle按照拦截器的顺序执行**

**2，postHandle和afterCompletion按照拦截器配置的逆向顺序执行**

### 8.3.4拦截器1放行，拦截器2不放行



只将拦截器2改为false

HandlerIntercepter1 preHandle

HandlerIntercepter2 preHandle

HandlerIntercepter1 afterCompletion

总结：

**拦截器1放行，拦截器2的preHandle才会执行。**

**拦截器2的preHandle不放行，拦截器2 postHandle 、afterCompletion这两个方法不会执行**

**只要有一个拦截器拦截不放行，postHandle不会执行**

### 8.3.5拦截器1 、2都不放行

HandlerIntercepter1 preHandle

总结：

**拦截器1 preHandle不放行，postHandle 、afterCompletion这两个方法不会执行**

**拦截器2 preHandle不放行，拦截器2不执行**

### 8.4小结

根据测试结果，对拦截器进行应用：

比如：统一日志处理拦截器，需要将该拦截器preHandle一定放行，且将它放在拦截器链的第一个位置。

比如：登录认证的拦截器，放在拦截器链中的第一个，权限校验拦截器，放在认证登录拦截器之后，（因为登录认证通过后才能校验权限）。  
9拦截器应用（登录认证拦截器）

## 9.1需求：

1. **用于请求url,拦截器进行拦截校验。**
2. **如果请求的url是公开地址（无需登录即可访问的url），则放行。**
3. **如果用户的session不存在，跳转到登录页面面。**
4. **如果用户的session存在，则放行**

## 9.2登录认证拦截器的实现

### 9.2.1登录的action

<body>

<h1>登录页面</h1>

<form action=*"${pageContext.request.contextPath}/login.action"*>

username:<input type=*"text"* name=*"username"*/>

password:<input type=*"password"* name=*"password"*/>

<input type=*"submit"* value=*"登录"*/>

</form>

</body>

</html>

### 9.2.2登录的controller

//登录页面

@Controller

**public** **class** LoginController {

//登录

@RequestMapping("/login")

**public** String login(HttpSession session, String username,String password) **throws** Exception{

//调用service进行用户身份验证

//.....

//在session中保存用户信息

session.setAttribute("username", username);

//重定向到商品的查询页面

**return** "redirect:/items/queryItems.action";

}

//退出

@RequestMapping("/logout")

**public** String logout(HttpSession session) **throws** Exception{

//清除session

session.invalidate();

//重定向到商品的查询页面

**return** "redirect:/items/queryItems.action";

}

}

### 9.2.3配置拦截器

<!-- 登录认证拦截器 -->

<mvc:interceptor>

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"com.wbs.intercrptor.LoginInterceptor"*></bean>

</mvc:interceptor>

### 9.2.3拦截器的实现

//登录认证拦截器

**public** **class** LoginInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

//获取url

String url=request.getRequestURI();

//判断url是否是公开地址，实际中将url配置在配置文件中

//这里公开地址是登录提交的地址

**if**(url.indexOf("login.action")>=0){

//url包括login.action，就是登录提交

**return** **true**;

}

//判断session

HttpSession session=request.getSession();

//从session中取出用于信息

String username=(String) session.getAttribute("username");

**if**(username!=**null**){

//身份信息存在，即用户已经存在，放行

**return** **true**;

}

//执行到这里表示用户信息需要验证，跳转到登录页面

request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);

**return** **false**;

}

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

System.***out***.println("HandlerIntercepter2 postHandle");

}

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

System.***out***.println("HandlerIntercepter2 afterCompletion");

}

}

9.2.4**在ItemsList.jsp页面加入下边的退出按钮，显示当前用户**

