springmvc第二天 高级知识

复习：

springmvc框架：

DispatcherServlet前端控制器：接收request，进行response

HandlerMapping处理器映射器：根据url查找Handler。（可以通过xml配置方式，注解方式）

HandlerAdapter处理器适配器：根据特定规则去执行Handler，编写Handler时需要按照HandlerAdapter的要求去编写。

Handler处理器（后端控制器）：需要程序员去编写，**常用注解开发方式。**

Handler处理器执行后结果 是ModelAndView，具体开发时Handler返回方法值类型包括 ：ModelAndView、String（逻辑视图名）、void（通过在Handler形参中添加request和response，类似原始 servlet开发方式，注意：可以通过指定response响应的结果类型实现json数据输出）

View resolver视图解析器：根据逻辑视图名生成真正的视图（在springmvc中使用View对象表示）

View视图:jsp页面，仅是数据展示，没有业务逻辑。

注解开发：

使用注解方式的处理器映射器和适配器：

<!--注解映射器 -->

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

<!--注解适配器 -->

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

在实际开发，使用<mvc:annotation-driven>代替上边处理器映射器和适配器配置。

@controller注解必须要加，作用标识类是一个Handler处理器。

@requestMapping注解必须要加，作用：

1、对url和Handler的方法进行映射。

2、可以窄化请求映射，设置Handler的根路径，url就是根路径+子路径请求方式

3、可以限制http请求的方法

映射成功后，springmvc框架生成一个Handler对象，对象中只包括 一个映射成功的method。

注解开发中参数绑定：

将request请求过来的key/value的数据（理解一个串），通过转换（参数绑定的一部分），将key/value串转成形参，将转换后的结果传给形参（整个参数绑定过程）。

springmvc所支持参数绑定：

1、默认支持很多类型，HttpServletRequest、response、session、

model/modelMap(将模型数据填充到request域)

2、支持简单数据类型，整型、字符串、日期。。

只要保证request请求的参数名和形参名称一致，自动绑定成功

如果request请求的参数名和形参名称不一致，可以使用@RequestParam（指定request请求的参数名），@RequestParam加在形参的前边。

3、支持pojo类型

只要保证request请求的参数名称和pojo中的属性名一致，自动将request请求的参数设置到pojo的属性中。

注意：形参中即有pojo类型又有简单类型，参数绑定互不影响。

自定义参数绑定：

日期类型绑定自定义：

定义的Converter<源类型，目标类型>接口实现类，比如：

Converter<String,Date>表示：将请求的日期数据串转成java中的日期类型。

注意：要转换的目标类型一定和接收的pojo中的属性类型一致。

将定义的Converter实现类注入到处理器适配器中。

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

</mvc:annotation-driven>

<!-- conversionService -->

<bean id=*"conversionService"*

class=*"**org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<!-- 转换器 -->

<property name=*"converters"*>

<list>

<bean class=*"cn.itcast.ssm.controller.converter.CustomDateConverter"*/>

</list>

</property>

</bean>

springmvc和struts2区别：

springmvc面向方法开发的（更接近service接口的开发方式），struts2面向类开发。

springmvc可以单例开发，struts2只能是多例开发。

# 课程安排

上午：

在商品查询和商品修改功能案例驱动下进行学习：

包装类型pojo参数绑定（掌握）。

集合类型的参数绑定：数组、list、map..

商品修改添加校验，学习springmvc提供校验validation（使用的是hibernate校验框架）

数据回显

统一异常处理（掌握）

下午：

上传图片

json数据交互

RESTful支持

拦截器

# 包装类型pojo参数绑定

## 需求

商品查询controller方法中实现商品查询条件传入。

## 实现方法

第一种方法：在形参中 添加HttpServletRequest request参数，通过request接收查询条件参数。

第二种方法：在形参中让包装类型的pojo接收查询条件参数。

分析：

页面传参数的特点：复杂，多样性。条件包括 ：用户账号、商品编号、订单信息。。。

如果将用户账号、商品编号、订单信息等放在简单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，pojo中属性是pojo。

## 页面参数和controller方法形参定义

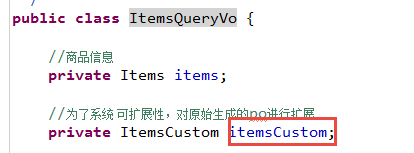
页面参数：

商品名称：<input name="itemsCustom.name" />

注意：itemsCustom和包装pojo中的属性一致即可。

controller方法形参：

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



# 集合类型绑定

## 数组绑定

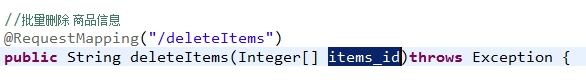
### 需求

商品批量删除，用户在页面选择多个商品，批量删除。

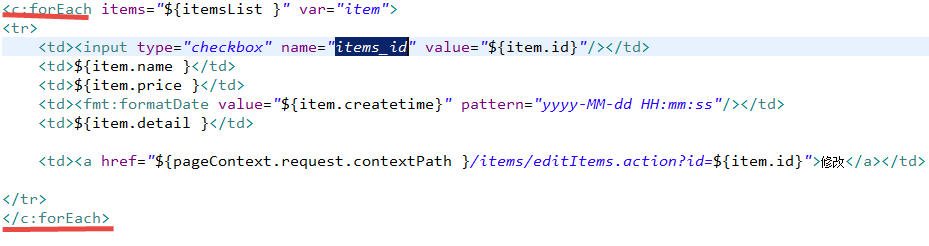
### 表现层实现

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

controller方法定义：



页面定义：



## list绑定

### 需求

通常在需要批量提交数据时，将提交的数据绑定到list<pojo>中，比如：成绩录入（录入多门课成绩，批量提交），

本例子需求：批量商品修改，在页面输入多个商品信息，将多个商品信息提交到controller方法中。

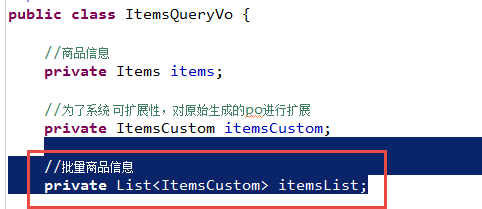
### 表现层实现

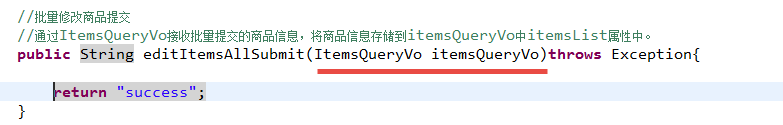
controller方法定义：

1、进入批量商品修改页面(页面样式参考商品列表实现)

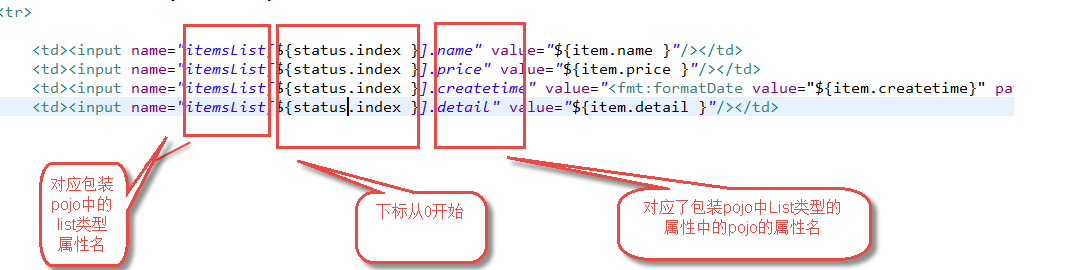
2、批量修改商品提交

使用List接收页面提交的批量数据，通过包装pojo接收，在包装pojo中定义list<pojo>属性





页面定义：



## map绑定

也通过在包装pojo中定义map类型属性。

在包装类中定义Map对象，并添加get/set方法，action使用包装对象接收。

包装类中定义Map对象如下：

**Public class** QueryVo {

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

//get/set方法..

}

页面定义如下：

<tr>

<td>学生信息：</td>

<td>

姓名：<inputtype=*"text"*name=*"itemInfo['name']"*/>

年龄：<inputtype=*"text"*name=*"itemInfo['price']"*/>

.. .. ..

</td>

</tr>

Contrller方法定义如下：

public String useraddsubmit(Model model,QueryVo queryVo)throws Exception{

System.out.println(queryVo.getStudentinfo());

}

# springmvc校验

## 校验理解

项目中，通常使用较多是前端的校验，比如页面中js校验。对于安全要求较高点建议在服务端进行校验。

服务端校验：

控制层conroller：校验页面请求的参数的合法性。在服务端控制层conroller校验，不区分客户端类型（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。

## springmvc校验需求

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

校验思路：

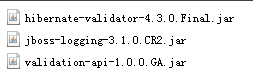
页面提交请求的参数，请求到controller方法中，使用validation进行校验。如果校验出错，将错误信息展示到页面。

具体需求：

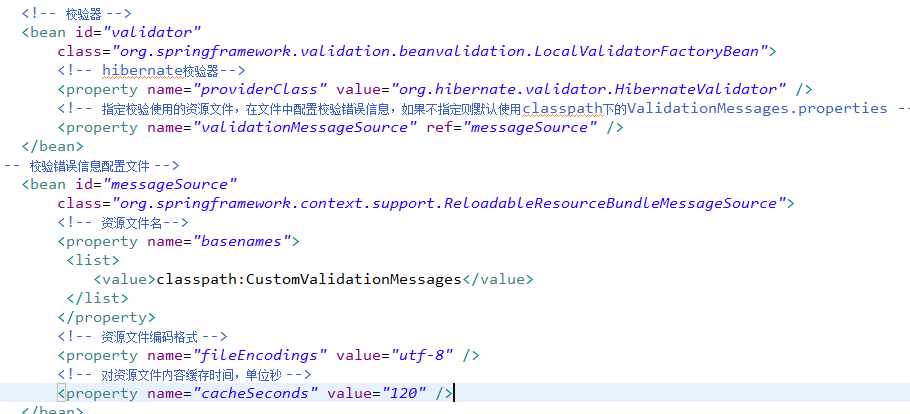
商品修改，添加校验（校验商品名称长度，生产日期的非空校验），如果校验出错，在商品修改页面显示错误信息。

## 环境准备

hibernate的校验框架validation所需要jar包：



## 配置校验器

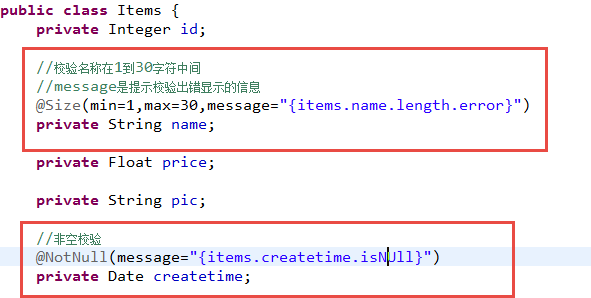


## 校验器注入到处理器适配器中



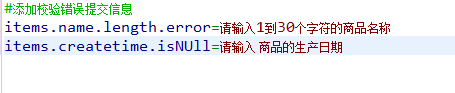
## 在pojo中添加校验规则

在ItemsCustom.java中添加校验规则：

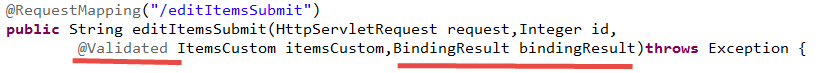


## CustomValidationMessages.properties

在CustomValidationMessages.properties配置校验错误信息：



## 捕获校验错误信息

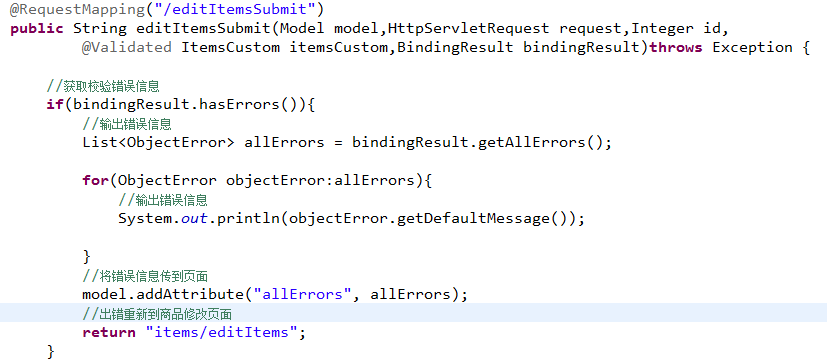


//在需要校验的pojo前边添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息

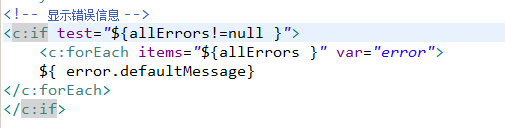
//注意：@Validated和BindingResult bindingResult是配对出现，并且形参顺序是固定的（一前一后）。

## 在页面显示校验错误信息

在controller中将错误信息传到页面即可。



页面显示错误信息：



## 分组校验

### 需求

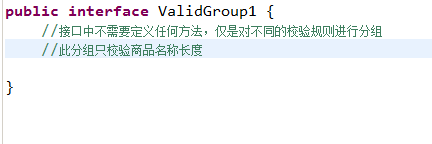
在pojo中定义校验规则，而pojo是被多个 controller所共用，当不同的controller方法对同一个pojo进行校验，但是每个controller方法需要不同的校验。

解决方法：

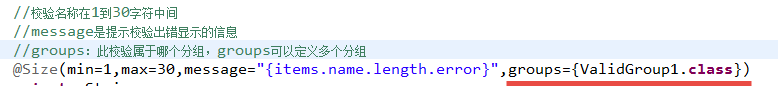
定义多个校验分组（其实是一个java接口），分组中定义有哪些规则

每个controller方法使用不同的校验分组

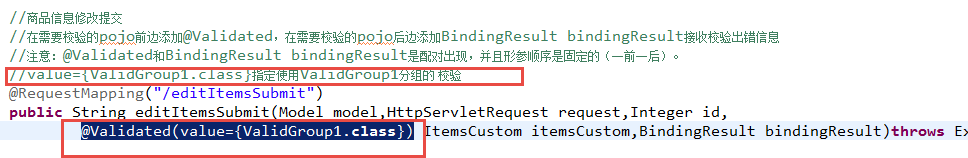
### 校验分组



### 在校验规则中添加分组



### 在controller方法使用指定分组的校验



# 数据回显

## 什么数据回显

提交后，如果出现错误，将刚才提交的数据回显到刚才的提交页面。

## pojo数据回显方法

1、springmvc默认对pojo数据进行回显。

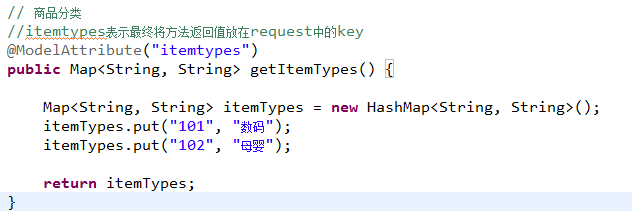
pojo数据传入controller方法后，springmvc自动将pojo数据放到request域，key等于pojo类型（首字母小写）

使用@ModelAttribute指定pojo回显到页面在request中的key

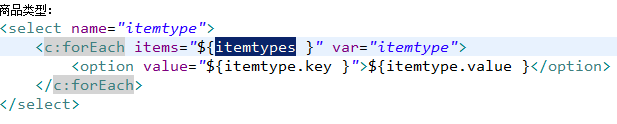
2、@ModelAttribute还可以将方法的返回值传到页面

在商品查询列表页面，通过商品类型查询商品信息。

在controller中定义商品类型查询方法，最终将商品类型传到页面。



页面上可以得到itemTypes数据。



3、使用最简单方法使用model，可以不用@ModelAttribute



## 简单类型数据回显

使用最简单方法使用model。

model.addAttribute("id", id);

# 异常处理

## 异常处理思路

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，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：

Controller

客户端

Service

Dao

Springmvc

DispatcherServlet

请求

异常

ExceptionResolver

异常处理器

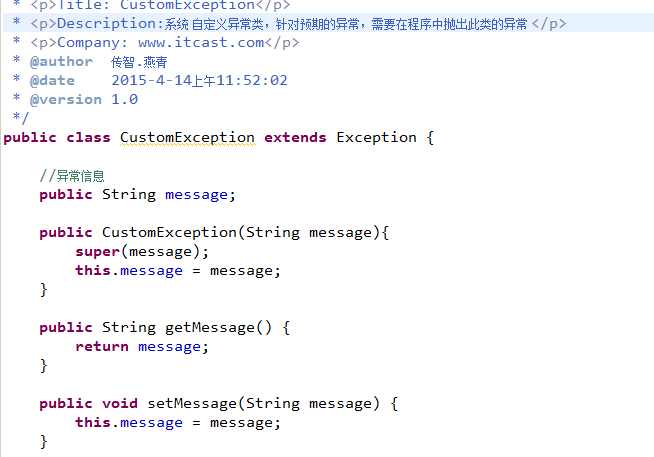
异常

异常

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

## 自定义异常类

对不同的异常类型定义异常类，继承Exception。



## 全局异常处理器

思路：

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

全局异常处理器处理思路：

解析出异常类型

如果该 异常类型是系统 自定义的异常，直接取出异常信息，在错误页面展示

如果该 异常类型不是系统 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”）

springmvc提供一个HandlerExceptionResolver接口

@Override

**public** ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex) {

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

// 解析出异常类型

// 如果该 异常类型是系统 自定义的异常，直接取出异常信息，在错误页面展示

// String message = null;

// if(ex instanceof CustomException){

// message = ((CustomException)ex).getMessage();

// }else{

//// 如果该 异常类型不是系统 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";

// }

//上边代码变为

CustomException customException = **null**;

**if**(ex **instanceof** CustomException){

customException = (CustomException)ex;

}**else**{

customException = **new** CustomException("未知错误");

}

//错误信息

String message = customException.getMessage();

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

//将错误信息传到页面

modelAndView.addObject("message", message);

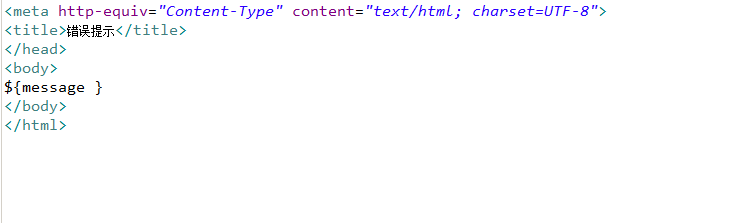
//指向错误页面

modelAndView.setViewName("error");

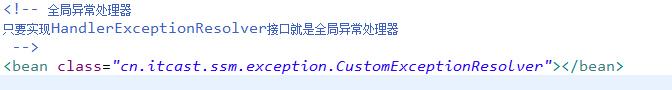
**return** modelAndView;

}

## 错误页面



## 在springmvc.xml配置全局异常处理器

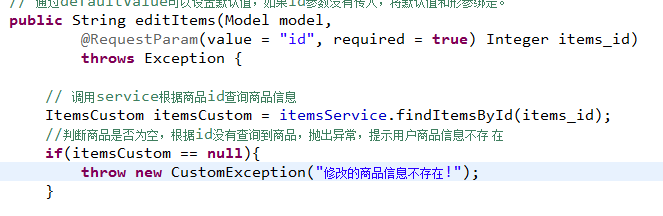


## 异常测试

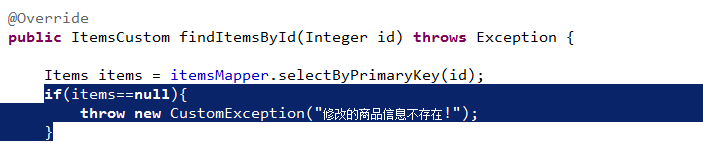
在controller、service、dao中任意一处需要手动抛出异常。

如果是程序中手动抛出的异常，在错误页面中显示自定义的异常信息，如果不是手动抛出异常说明是一个运行时异常，在错误页面只显示“未知错误”。

在商品修改的controller方法中抛出异常 .



在service接口中抛出异常：



如果与业务功能相关的异常，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。

上边的功能，建议在service中抛出异常。

# 上传图片

## 需求

在修改商品页面，添加上传商品图片功能。

## springmvc中对多部件类型解析

在 页面form中提交enctype="multipart/form-data"的数据时，需要springmvc对multipart类型的数据进行解析。

在springmvc.xml中配置multipart类型解析器。



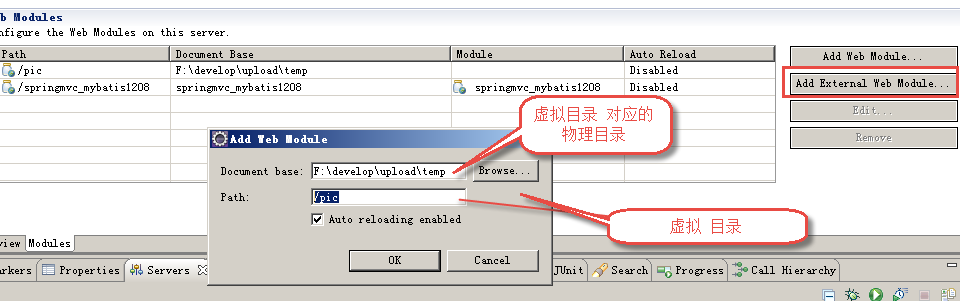
## 加入上传图片的jar

上边的解析内部使用下边的jar进行图片上传。



## 创建图片虚拟 目录 存储图片

通过图形界面配置：



也可以直接修改tomcat的配置：

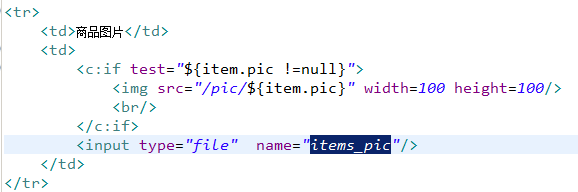
在conf/server.xml文件，添加虚拟 目录 ：



注意：在图片虚拟目录 中，一定将图片目录分级创建（提高i/o性能），一般我们采用按日期(年、月、日)进行分级创建。

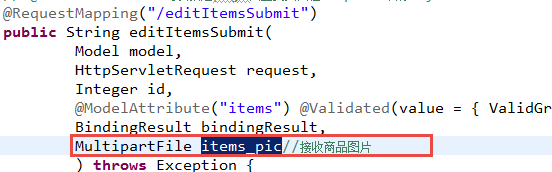
## 上传图片代码

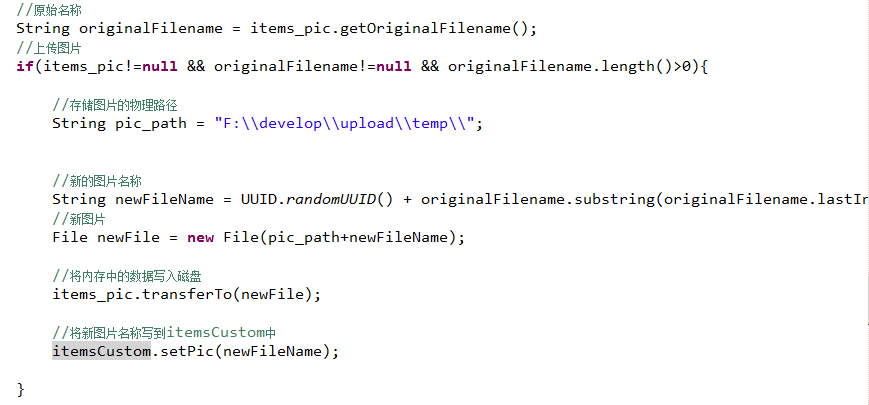
### 页面



### controller方法

修改：商品修改controller方法：





# json数据交互

## 为什么要进行json数据交互

json数据格式在接口调用中、html页面中较常用，json格式比较简单，解析还比较方便。

比如：webservice接口，传输json数据.

## springmvc进行json交互

客户端请求

请求的是json串

需要指定contentType=

application/json

@RequestBody将json串转成java对象

@ResponseBody将java对象转成json串输出

请求的是key/value

contentType=

application/x-www-form-urlencoded

不需要@RequestBody将json串转成java对象

@ResponseBody将java对象转成json串输出

最终都输出json数据，为了在前端页面方便对请求结果进行解析

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转成json，不太方便。

2、请求key/value、输出json。此方法比较常用。

## 环境准备

### 加载json转的jar包

springmvc中使用jackson的包进行json转换（@requestBody和@responseBody使用下边的包进行json转），如下：



### 配置json转换器

在注解适配器中加入messageConverters

<!--注解适配器 -->

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

<property name=*"messageConverters"*>

<list>

<bean class=*"org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"*></bean>

</list>

</property>

</bean>

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

## json交互测试

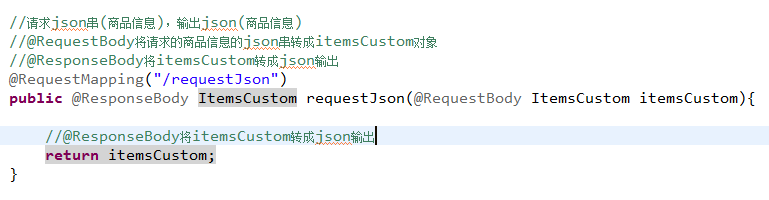
### 输入json串，输出是json串

#### jsp页面

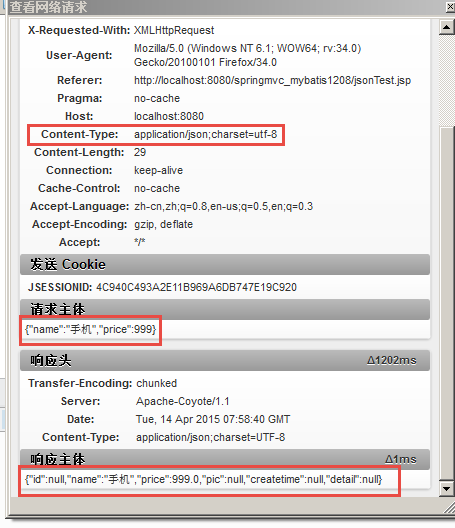
使用jquery的ajax提交json串，对输出的json结果进行解析。



#### controller



#### 测试结果



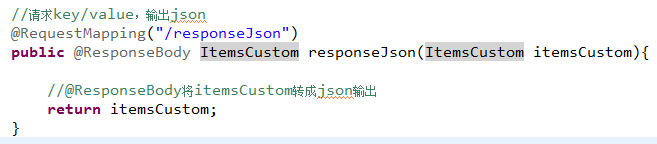
### 输入key/value，输出是json串

#### jsp页面

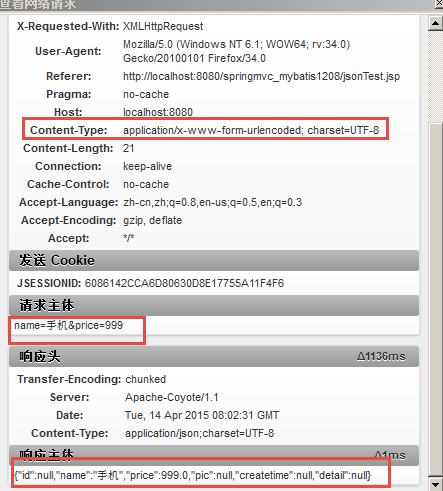
使用jquery的ajax提交key/value串，对输出的json结果进行解析。



#### controller



#### 测试



# RESTful支持

## 什么是RESTful

RESTful架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

RESTful（即Representational State Transfer的缩写）其实是一个开发理念，是对http的很好的诠释。

1、对url进行规范，写RESTful格式的url

非REST的url：http://...../queryItems.action?id=001&type=T01

REST的url风格：http://..../items/001

特点：url简洁，将参数通过url传到服务端

2、http的方法规范

不管是删除、添加、更新。。使用url是一致的，如果进行删除，需要设置http的方法为delete，同理添加。。。

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。

3、对http的contentType规范

请求时指定contentType，要json数据，设置成json格式的type。。

## REST的例子

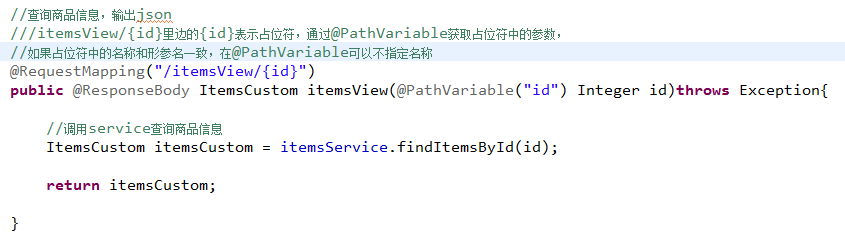
### 需求

查询商品信息，返回json数据。

### controller

定义方法，进行url映射使用REST风格的url，将查询商品信息的id传入controller .

输出json使用@ResponseBody将java对象输出json。



@RequestMapping(value="/ itemsView/{id}")：{×××}占位符，请求的URL可以是“/viewItems/1”或“/viewItems/2”，通过在方法中使用@PathVariable获取{×××}中的×××变量。

@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。

如果RequestMapping中表示为"/ itemsView /{id}"，id和形参名称一致，@PathVariable不用指定名称。

### REST方法的前端控制器配置

在web.xml配置：

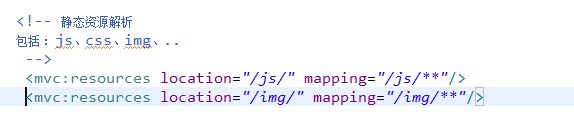


## 对静态资源的解析

配置前端控制器的url-parttern中指定/，对静态资源的解析出现问题：



在springmvc.xml中添加静态资源解析方法。



# 拦截器

## 拦截定义

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

**public** **class** HandlerInterceptor1 **implements** HandlerInterceptor {

//进入 Handler方法之前执行

//用于身份认证、身份授权

//比如身份认证，如果认证通过表示当前用户没有登陆，需要此方法拦截不再向下执行

@Override

**public** **boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

//return false表示拦截，不向下执行

//return true表示放行

**return** **false**;

}

//进入Handler方法之后，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

@Override

**public** **void** postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

}

//执行Handler完成执行此方法

//应用场景：统一异常处理，统一日志处理

@Override

**public** **void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

}

}

## 拦截器配置

### 针对HandlerMapping配置

**springmvc拦截器针对HandlerMapping进行拦截设置，**如果在某个HandlerMapping中配置拦截，经过该 HandlerMapping映射成功的handler最终使用该 拦截器。

<bean

class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*>

<property name=*"interceptors"*>

<list>

<ref bean=*"handlerInterceptor1"*/>

<ref bean=*"handlerInterceptor2"*/>

</list>

</property>

</bean>

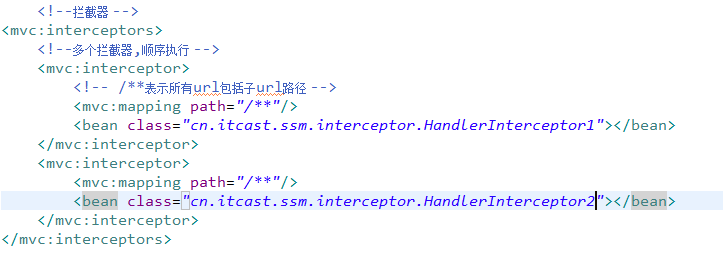
<bean id=*"handlerInterceptor1"* class=*"springmvc.intercapter.HandlerInterceptor1"*/>

<bean id=*"handlerInterceptor2"* class=*"springmvc.intercapter.HandlerInterceptor2"*/>

一般不推荐使用。

### 类似全局的拦截器

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。



## 拦截测试

### 测试需求

测试多个拦截器各各方法执行时机。

### 编写两个拦截



### 两个拦截器都放行

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor2...postHandle

HandlerInterceptor1...postHandle

HandlerInterceptor2...afterCompletion

HandlerInterceptor1...afterCompletion

总结：

preHandle方法按顺序执行，

postHandle和afterCompletion按拦截器配置的逆向顺序执行。

### 拦截器1放行，拦截器2不放行

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor1...afterCompletion

总结：

拦截器1放行，拦截器2 preHandle才会执行。

拦截器2 preHandle不放行，拦截器2 postHandle和afterCompletion不会执行。

只要有一个拦截器不放行，postHandle不会执行。

### 拦截器1不放行，拦截器2不放行

HandlerInterceptor1...preHandle

拦截器1 preHandle不放行，postHandle和afterCompletion不会执行。

拦截器1 preHandle不放行，拦截器2不执行。

### 小结

根据测试结果，对拦截器应用。

比如：统一日志处理拦截器，需要该 拦截器preHandle一定要放行，且将它放在拦截器链接中第一个位置。

比如：登陆认证拦截器，放在拦截器链接中第一个位置。权限校验拦截器，放在登陆认证拦截器之后。（因为登陆通过后才校验权限）

## 拦截器应用（实现登陆认证）

### 需求

1、用户请求url

2、拦截器进行拦截校验

如果请求的url是公开地址（无需登陆即可访问的url），让放行

如果用户session 不存在跳转到登陆页面

如果用户session存在放行，继续操作。

### 登陆controller方法

@Controller

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

// 登陆

@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中保存用户身份信息

session.setAttribute("username", username);

// 重定向到商品列表页面

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

}

// 退出

@RequestMapping("/logout")

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

// 清除session

session.invalidate();

// 重定向到商品列表页面

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

}

}

### 登陆认证拦截实现

#### 代码实现

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

//进入 Handler方法之前执行

//用于身份认证、身份授权

//比如身份认证，如果认证通过表示当前用户没有登陆，需要此方法拦截不再向下执行

@Override

**public** **boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

//获取请求的url

String url = request.getRequestURI();

//判断url是否是公开 地址（实际使用时将公开 地址配置配置文件中）

//这里公开地址是登陆提交的地址

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

//如果进行登陆提交，放行

**return** **true**;

}

//判断session

HttpSession session = request.getSession();

//从session中取出用户身份信息

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

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

//身份存在，放行

**return** **true**;

}

//执行这里表示用户身份需要认证，跳转登陆页面

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

//return false表示拦截，不向下执行

//return true表示放行

**return** **false**;

}

#### 拦截器配置

