

## 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。

## 1 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

## 2 包装类型pojo参数绑定

### 2.1 需求

商品查询 controller 方法中实现商品查询条件传入。

### 2.2 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。

### 2.3 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
```

```
public class ItemsQueryVo {  
  
    //商品信息  
    private Items items;  
  
    //为了系统可扩展性，对原始生成的pojo进行扩展  
    private ItemsCustom itemsCustom;  
}
```

## 3 集合类型绑定

## 3.1 数组绑定

### 3.1.1 需求

商品批量删除，用户在页面选择多个商品，批量删除。

### 3.1.2 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：

```
//批量删除 商品信息
@RequestMapping("/deleteItems")
public String deleteItems(Integer[] items_id) throws Exception {
```

页面定义：

```
<c:forEach items="${itemsList }" var="item">
<tr>
<td><input type="checkbox" name="items_id" value="${item.id}"/></td>
<td>${item.name }</td>
<td>${item.price }</td>
<td><fmt:formatDate value="${item.createtime}" pattern="yyyy-MM-dd HH:mm:ss"/></td>
<td>${item.detail }</td>

<td><a href="${pageContext.request.contextPath }/items/editItems.action?id=${item.id}">修改</a></td>

</tr>
</c:forEach>
```

## 3.2 list绑定

### 3.2.1 需求

通常需要在批量提交数据时，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。

### 3.2.2 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

```
public class ItemsQueryVo {
```

```
    //商品信息
```

```
    private Items items;
```

```
    //为了系统 可扩展性, 对原始生成的po进行扩展
```

```
    private ItemsCustom itemsCustom;
```

```
    //批里商品信息
```

```
    private List<ItemsCustom> itemsList;
```

```
    //批里修改商品提交
```

```
    //通过ItemsQueryVo接收批里提交的商品信息, 将商品信息存储到itemsQueryVo中itemsList属性中。
```

```
    public String editItemsAllSubmit(ItemsQueryVo itemsQueryVo) throws Exception{
```

```
        return "success";
```

```
    }
```

页面定义:

```
<tr>
```

```
<td><input name="itemsList" ${status.index}].name" value="${item.name }"/></td>
```

```
<td><input name="itemsList" ${status.index}].price" value="${item.price }"/></td>
```

```
<td><input name="itemsList" ${status.index}].createtime" value="<fmt:formatDate value="${item.createtime}" pa
```

```
<td><input name="itemsList" ${status.index}].detail" value="${item.detail }"/></td>
```

对应包装  
pojo中的  
list类型  
属性名

下标从0开始

对应了包装pojo中List类型的  
属性中的pojo的属性名

### 3.3 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: <input type="text" name="itemInfo['name']"/>
年龄: <input type="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());
}
```

## 4 springmvc校验

### 4.1 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 **controller**：校验页面请求的参数的合法性。在服务端控制层 **controller** 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。

### 4.2 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。

## 4.3 环境准备

hibernate 的校验框架 validation 所需要 jar 包：

---

 hibernate-validator-4.3.0.Final.jar  
 jboss-logging-3.1.0.CR2.jar  
 validation-api-1.0.0.GA.jar

## 4.4 配置校验器

```
<!-- 校验器 -->
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <!-- hibernate校验器-->
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator" />
    <!-- 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" />
</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>
```

## 4.5 校验器注入到处理器适配器中

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

## 4.6 在pojo中添加校验规则

在 ItemsCustom.java 中添加校验规则：

```

public class Items {
    private Integer id;

    //校验名称在1到30字符中间
    //message是提示校验出错显示的信息
    @Size(min=1,max=30,message="{items.name.length.error}")
    private String name;

    private Float price;

    private String pic;

    //非空校验
    @NotNull(message="{items.createtime.isNull}")
    private Date createtime;

```

## 4.7 CustomValidationMessages.properties

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

```

#添加校验错误提交信息
items.name.length.error=请输入1到30个字符的商品名称
items.createtime.isNull=请输入 商品的生产日期

```

## 4.8 捕获校验错误信息

```

@RequestMapping("/editItemsSubmit")
public String editItemsSubmit(HttpServletRequest request,Integer id,
    @Validated ItemsCustom itemsCustom,BindingResult bindingResult)throws Exception {

```

//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）。

## 4.9 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。

```
@RequestMapping("/editItemsSubmit")
public String editItemsSubmit(Model model,HttpServletRequest request,Integer id,
    @Validated ItemsCustom itemsCustom,BindingResult bindingResult)throws Exception {

    //获取校验错误信息
    if(bindingResult.hasErrors()){
        //输出错误信息
        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);
        //出错重新到商品修改页面
        return "items/editItems";
    }
}
```

页面显示错误信息：

```
<!-- 显示错误信息 -->
<c:if test="${allErrors!=null }">
    <c:forEach items="${allErrors }" var="error">
        ${ error.defaultMessage}
    </c:forEach>
</c:if>
```

## 4.10分组校验

### 4.10.1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

### 4.10.2校验分组

```
public interface ValidGroup1 {
    //接口中不需要定义任何方法，仅是对不同的校验规则进行分组
    //此分组只校验商品名称长度
}
```

### 4.10.3在校验规则中添加分组

```
//校验名称在1到30字符中间
//message是提示校验出错显示的信息
//groups: 此校验属于哪个分组，groups可以定义多个分组
@Size(min=1,max=30,message="{items.name.length.error}",groups={ValidGroup1.class})
```

### 4.10.4在controller方法使用指定分组的校验

```
//商品信息修改提交
//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）。
//value={ValidGroup1.class}指定使用ValidGroup1分组的 校验
@RequestMapping("/editItemsSubmit")
public String editItemsSubmit(Model model,HttpServletRequest request,Integer id,
    @Validated(value={ValidGroup1.class}) ItemsCustom itemsCustom,BindingResult bindingResult)throws E>
```

## 5 数据回显

### 5.1 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。

### 5.2 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。

```
// 商品分类
//itemtypes表示最终将方法返回值放在request中的key
@ModelAttribute("itemtypes")
public Map<String, String> getItemTypes() {

    Map<String, String> itemTypes = new HashMap<String, String>();
    itemTypes.put("101", "数码");
    itemTypes.put("102", "母婴");

    return itemTypes;
}
```

页面上可以得到 itemTypes 数据。

商品类型:

```
<select name="itemtype">
    <c:forEach items="${itemtypes}" var="itemtype">
        <option value="${itemtype.key}">${itemtype.value}</option>
    </c:forEach>
</select>
```

## 3、使用最简单方法使用 `model`，可以不用 `@ModelAttribute`

```
// 获取校验错误信息
if (bindingResult.hasErrors()) {
    // 输出错误信息
    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("items", itemsCustom);

    // 出错重新到商品修改页面
    return "items/editItems";
}
```

## 5.3 简单类型数据回显

使用最简单方法使用 `model`。

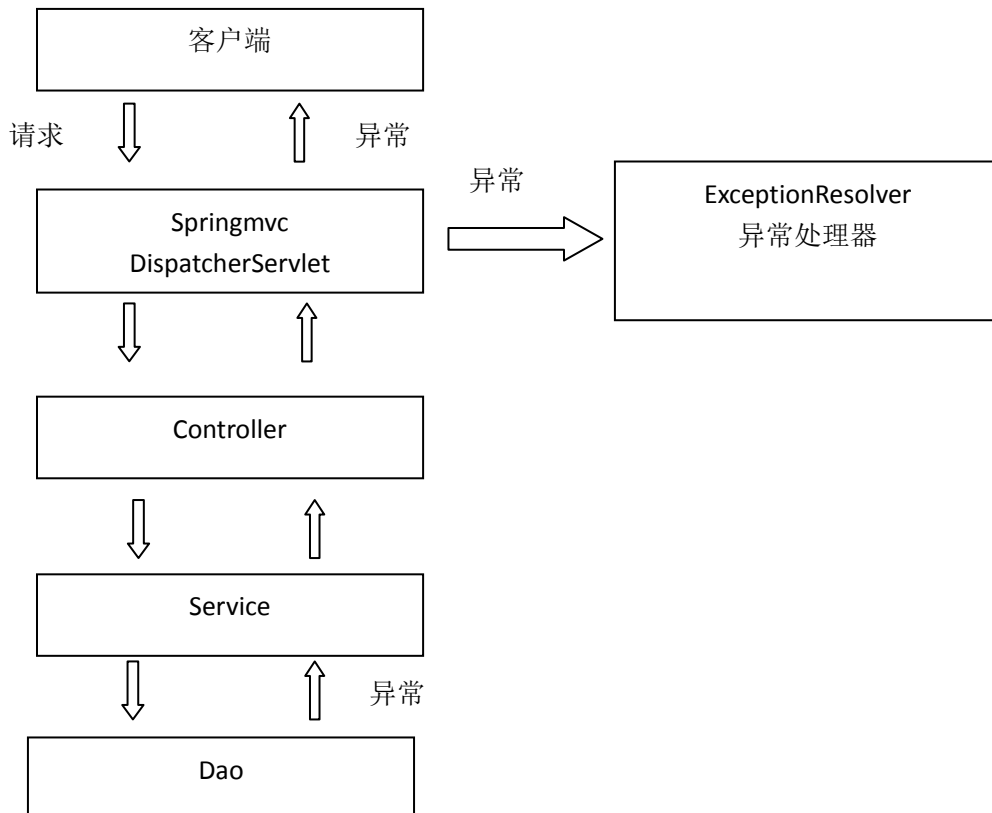
```
model.addAttribute("id", id);
```

# 6 异常处理

## 6.1 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：



`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。

## 6.2 自定义异常类

对不同的异常类型定义异常类，继承 `Exception`。

```
 * <p>Title: CustomException</p>
 * <p>Description:系统 自定义异常类，针对预期的异常，需要在程序中抛出此类的异常 </p>
 * <p>Company: www.itcast.com</p>
 * @author 传智·燕青
 * @date 2015-4-14上午11:52:02
 * @version 1.0
 */
public class CustomException extends Exception {

    //异常信息
    public String message;

    public CustomException(String message){
        super(message);
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

## 6.3 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;
}
```

## 6.4 错误页面

```

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>错误提示</title>
</head>
<body>
${message }
</body>
</html>

```

## 6.5 在springmvc.xml配置全局异常处理器

```

<!-- 全局异常处理器
只要实现HandlerExceptionResolver接口就是全局异常处理器
-->
<bean class="cn.itcast.ssm.exception.CustomExceptionHandler"></bean>

```

## 6.6 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。

```

// 通过defaultValue可以设置默认值，如果id参数没有传入，将默认值和id参数绑定。
public String editItems(Model model,
    @RequestParam(value = "id", required = true) Integer items_id)
    throws Exception {

    // 调用service根据商品id查询商品信息
    ItemsCustom itemsCustom = itemsService.findItemsById(items_id);
    //判断商品是否为空，根据id没有查询到商品，抛出异常，提示用户商品信息不存在
    if(itemsCustom == null){
        throw new CustomException("修改的商品信息不存在!");
    }
}

```

在 service 接口中抛出异常：

```

@Override
public ItemsCustom findById(Integer id) throws Exception {

    Items items = itemsMapper.selectByPrimaryKey(id);
    if(items==null){
        throw new CustomException("修改的商品信息不存在!");
    }
}

```

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。

## 7 上传图片

### 7.1 需求

在修改商品页面，添加上传商品图片功能。

### 7.2 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。



```

<!-- 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</value>
    </property>
</bean>

```

### 7.3 加入上传图片的jar

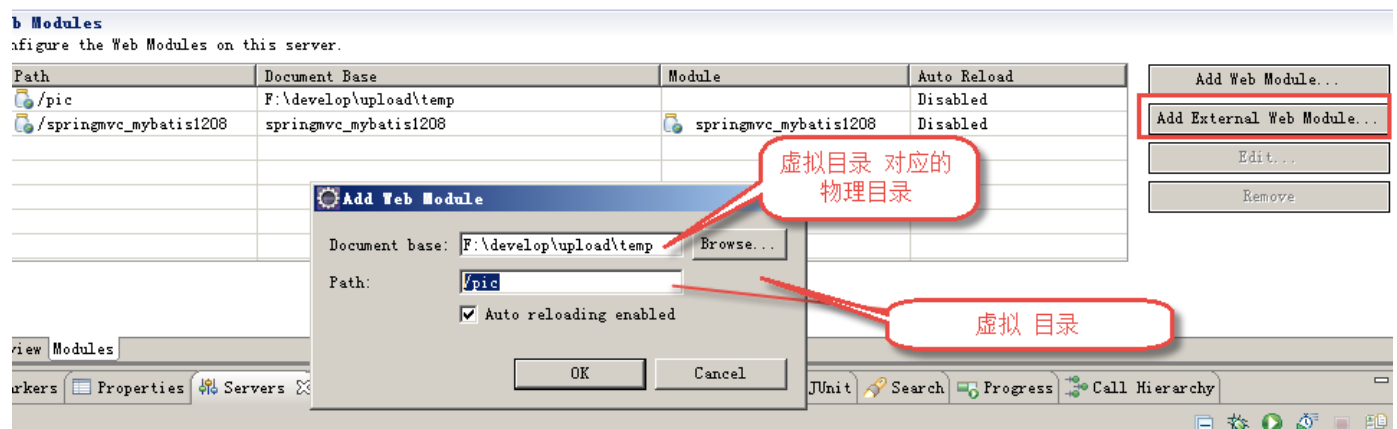
上边的解析内部使用下边的 jar 进行图片上传。

 commons-fileupload-1.2.2.jar  
 commons-io-2.4.jar



## 7.4 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：

```
<Context docBase="F:\develop\upload\temp" path="/pic" reloadable="false"/>
```

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。

## 7.5 上传图片代码

### 7.5.1 页面

```
<tr>
  <td>商品图片</td>
  <td>
    <c:if test="${item.pic != null}">
      
      <br/>
    </c:if>
    <input type="file" name="items_pic"/>
  </td>
</tr>
```

## 7.5.2 controller方法

修改：商品修改 controller 方法：

```
@RequestMapping("/editItemsSubmit")
public String editItemsSubmit(
    Model model,
    HttpServletRequest request,
    Integer id,
    @ModelAttribute("items") @Validated(value = { ValidGr
    BindingResult bindingResult,
    MultipartFile items_pic//接收商品图片
) throws Exception {

    //原始名称
    String originalFilename = items_pic.getOriginalFilename();
    //上传图片
    if(items_pic!=null && originalFilename!=null && originalFilename.length()>0){

        //存储图片的物理路径
        String pic_path = "F:\\develop\\upload\\temp\\";

        //新的图片名称
        String newFileName = UUID.randomUUID() + originalFilename.substring(originalFilename.lastIr
        //新图片
        File newFile = new File(pic_path+newFileName);

        //将内存中的数据写入磁盘
        items_pic.transferTo(newFile);

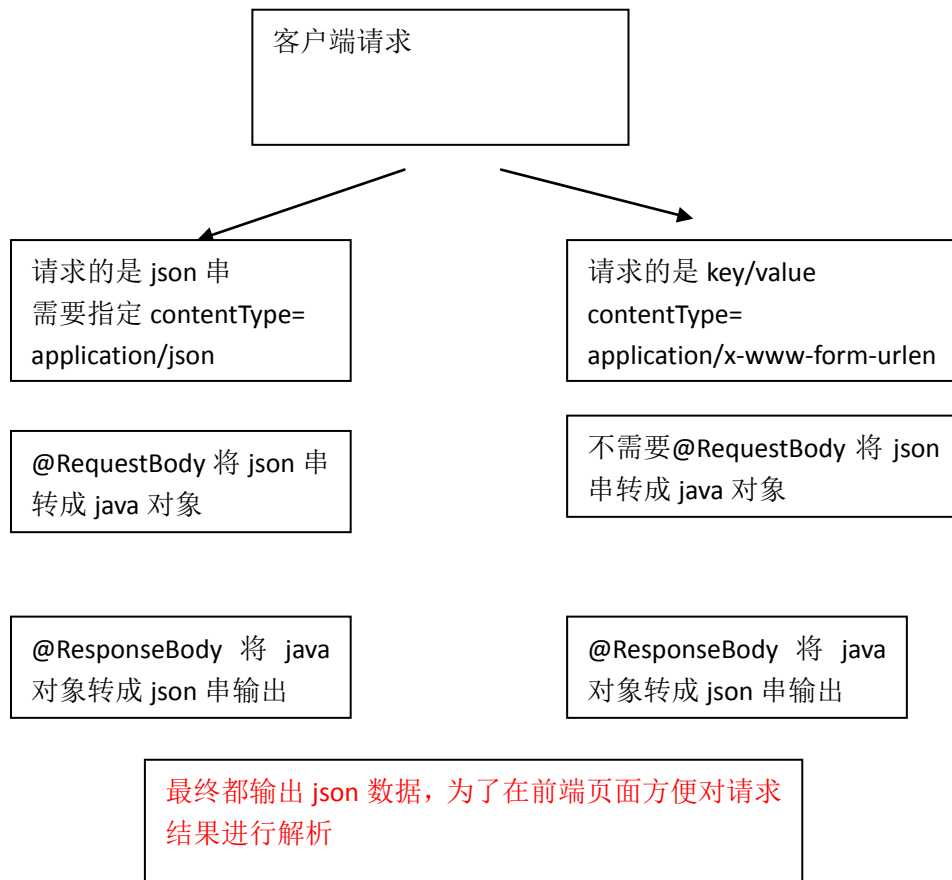
        //将新图片名称写到itemsCustom中
        itemsCustom.setPic(newFileName);
    }
}
```

## 8 json数据交互

### 8.1 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。

## 8.2 springmvc进行json交互



- 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。

## 8.3 环境准备

### 8.3.1 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：

```
jackson-core-asl-1.9.11.jar -  
jackson-mapper-asl-1.9.11.jar
```

## 8.3.2 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。

## 8.4 json交互测试

### 8.4.1 输入json串，输出是json串

#### 8.4.1.1 jsp页面

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

```

//请求json，输出是json
function requestJson(){

    $.ajax({
        type: 'post',
        url: '${pageContext.request.contextPath }/requestJson.action',
        contentType: 'application/json;charset=utf-8',
        //数据格式是json串，商品信息
        data: '{"name":"手机","price":999}',
        success: function(data){//返回json结果
            alert(data);
        }

    });

}
}

```

#### 8.4.1.2 controller

```

//请求json串(商品信息)，输出json(商品信息)
//@RequestBody将请求的商品信息的json串转成ItemsCustom对象
//@ResponseBody将ItemsCustom转成json输出
@RequestMapping("/requestJson")
public @ResponseBody ItemsCustom requestJson(@RequestBody ItemsCustom itemsCustom){

    //@ResponseBody将ItemsCustom转成json输出
    return itemsCustom;
}

```

### 8.4.1.3 测试结果



## 8.4.2 输入key/value，输出是json串

### 8.4.2.1 jsp页面

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

```
//请求key/value, 输出是json
function responseJson(){

    $.ajax({
        type: 'post',
        url: '${pageContext.request.contextPath }/responseJson.action',
        //请求是key/value这里不需要指定contentType, 因为默认就是key/value类型
        //contentType: 'application/json;charset=utf-8',
        //数据格式是json串, 商品信息
        data: 'name=手机&price=999',
        success: function(data){//返回json结果
            alert(data);
        }

    });

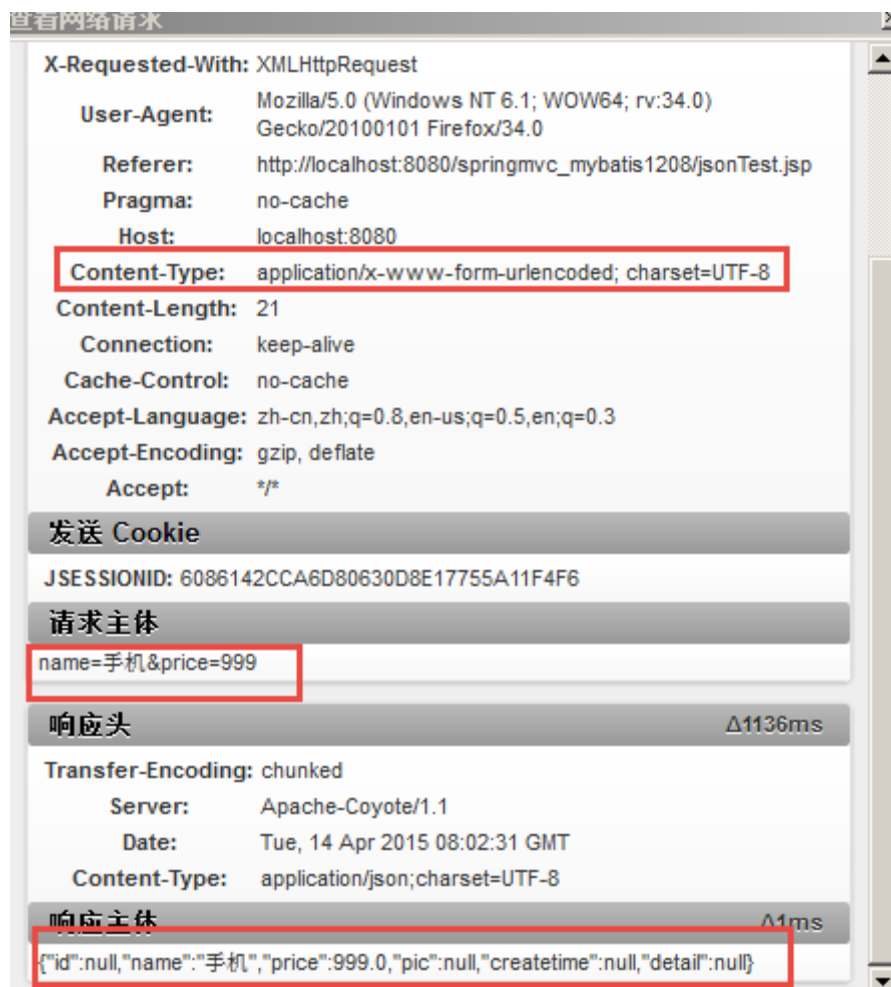
}
```

#### 8.4.2.2 controller

```
//请求key/value, 输出json
@RequestMapping("/responseJson")
public @ResponseBody ItemsCustom responseJson(ItemsCustom itemsCustom){

    //@ResponseBody将itemsCustom转成json输出
    return itemsCustom;
}
```

#### 8.4.2.3 测试



## 9 RESTful支持

### 9.1 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。。

# 9.2 REST的例子

## 9.2.1 需求

查询商品信息，返回 json 数据。

## 9.2.2 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。

```
//查询商品信息，输出json
//itemsView/{id}里边的{id}表示占位符，通过@PathVariable获取占位符中的参数，
//如果占位符中的名称和形参名一致，在@PathVariable可以不指定名称
@RequestMapping("/itemsView/{id}")
public @ResponseBody ItemsCustom itemsView(@PathVariable("id") Integer id)throws Exception{

    //调用service查询商品信息
    ItemsCustom itemsCustom = itemsService.findItemsById(id);

    return itemsCustom;
}
```

@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。

## 9.2.3 REST方法的前端控制器配置

在 web.xml 配置：

```

<!-- springmvc前端控制器, rest配置 -->
<servlet>
    <servlet-name>springmvc_rest</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- contextConfigLocation配置springmvc加载的配置文件（配置处理器映射器、适配器等等）如果不配置contextConfig
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>springmvc_rest</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

```

## 9.3 对静态资源的解析

配置前端控制器的 url-pattern 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。

```

<!-- 静态资源解析
包括: js、css、img、..
-->
<mvc:resources location="/js/" mapping="/js/**"/>
<mvc:resources location="/img/" mapping="/img/**"/>

```

# 10 拦截器

## 10.1 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 {

    }
}
```

## 10.2 拦截器配置

### 10.2.1 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。

### 10.2.2 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。

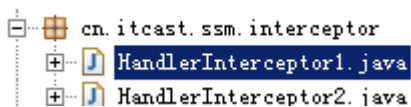
```
<!--拦截器-->
<mvc:interceptors>
    <!--多个拦截器,顺序执行-->
    <mvc:interceptor>
        <!-- /**表示所有url包括子url路径 -->
        <mvc:mapping path="/**"/>
        <bean class="cn.itcast.ssm.interceptor.HandlerInterceptor1"></bean>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="cn.itcast.ssm.interceptor.HandlerInterceptor2"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

## 10.3 拦截测试

### 10.3.1 测试需求

测试多个拦截器各方法执行时机。

### 10.3.2 编写两个拦截



### 10.3.3 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。

### 10.3.4 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。

### 10.3.1 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。

### 10.3.2 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）

## 10.4 拦截器应用（实现登陆认证）

### 10.4.1 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。

### 10.4.2 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";
}
}

```

## 10.4.3 登陆认证拦截实现

### 10.4.3.1 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;
}

```

### 10.4.3.2 拦截器配置

```

<!--拦截器-->
<mvc:interceptors>
    <!--多个拦截器，顺序执行-->
    <!-- 登陆认证拦截器 -->
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="cn.itcast.ssm.interceptor.LoginInterceptor"></bean>
    </mvc:interceptor>

```