# 高级参数绑定

## 绑定数组

使用场景：批量删除

|  |
| --- |
| <c:for Eachitems="${itemList }" var="item">  <td><inputtype="checkbox" name=**"ids"** value="${item.id}"/></td>  <td>**<fmt:formatDatevalue="${item.createtime}"pattern="yyyy-MM-ddHH:mm:ss"/>**</td>  </c:forEach>  **@RequestMapping("queryItem")**  **public String queryItem(Integer[] ids) {**  **....**  **return"success";**  **}** |

## 绑定到List

适用场景：批量修改。

|  |
| --- |
| List |

|  |
| --- |
| 传参 |

varStatus属性常用参数总结下：

${status.index} 输出行号，从0开始。

${status.count} 输出行号，从1开始。

${status.first} 判断当前项是否为集合中的第一项，返回值为true或false

${status.last} 判断当前项是否为集合中的最后一项，返回值为true或false

# @RequestMapping

## URL路径映射

**value的值是数组，可以将多个url映射到同一个方法.**

/\*\*

\* 查询商品列表

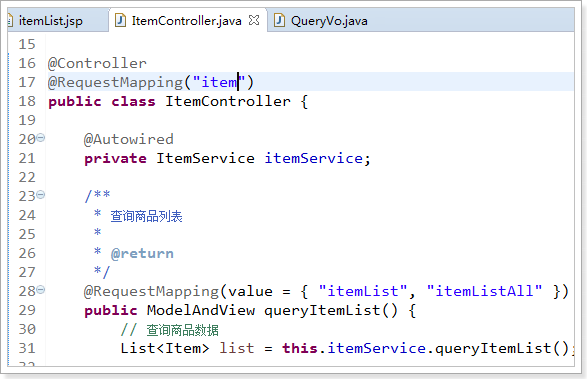
\*/

@RequestMapping(value = { "itemList", "itemListAll" })

## 添加在类上面

在class上添加@RequestMapping(url)指定通用请求前缀，限制此类下的所有方法请求url必须以请求前缀开头

可以使用此方法对url进行分类管理，如下图：



此时需要进入queryItemList()方法的请求url为：

http://127.0.0.1:8080/springmvc-web2/item/itemList.action

或者

http://127.0.0.1:8080/springmvc-web2/item/itemListAll.action

## 请求方法限定

除了可以对url进行设置，还可以限定请求进来的方法

* 限定GET方法

@RequestMapping(method = RequestMethod.***GET***)

如果通过POST访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

@RequestMapping(value = "itemList",method = RequestMethod.***POST***)

* 限定POST方法

@RequestMapping(method = RequestMethod.***POST***)

如果通过GET访问则报错：

HTTP Status 405 - Request method 'GET' not supported

* GET和POST都可以

@RequestMapping(method = {RequestMethod.***GET***,RequestMethod.***POST***})

# Controller方法返回值



## 返回ModelAndView

## 返回void或自定义类型

常用于ajax异步请求时json数据交互，详细情况看后面json数据交互。

## 返回字符串

|  |
| --- |
| 返回字符串：   1. 返回值即视图名 2. 数据放在model或modelmap中（modelmap是model的实现类，使用上无区别） 3. 重定向：Redirect   如果需要传参："redirect:/itemEdit.action?itemId=" + item.getId();   1. 转发：forward |

# 异常处理器

springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个系统的异常处理逻辑。

## 异常处理思路

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.

## 自定义异常类

为了区别不同的异常,通常根据异常类型进行区分，这里我们创建一个自定义系统异常。

如果controller、service、dao抛出此类异常说明是**系统预期处理的异常信息**。

**Publicclass** MyException **extends** Exception {

// 异常信息

**private** String message;

**public**MyException() {

**super**();

}

**Public**MyException(String message) {

**super**();

**this**.message = message;

}

**public** String **getMessage() {**

**return**message;

}

**PublicvoidsetMessage(S**tring message) {

**this**.message = message;

}

}

## 自定义异常处理器

**Public class** CustomHandleException **implements** HandlerExceptionResolver {

@Override

**Public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception exception) {

// 定义异常信息

String msg;

// 判断异常类型

**if** (exception **instanceof** MyException) {

// 如果是自定义异常，读取异常信息

msg = exception.getMessage();

} **else** {

// 如果是运行时异常，则取错误堆栈，从堆栈中获取异常信息

Writer out = **new** StringWriter();

PrintWriter s = **new** PrintWriter(out);

exception.printStackTrace(s);

msg = out.toString();

}

// 把错误信息记录到日志中/发邮件给相应的开发人员等方式

// 返回错误页面，给用户友好页面显示错误信息

ModelAndViewmodelAndView = **new**ModelAndView();

modelAndView.addObject("msg", msg);

modelAndView.setViewName("error");

**return**modelAndView;

}

}

## 异常处理器配置

在springmvc.xml中添加：

<!-- 配置全局异常处理器 -->

<bean

id=*"customHandleException"* class=*"cn.itcast.ssm.exception.CustomHandleException"*/>

|  |
| --- |
| 看到这里会发现：配置转换器converter（时间）、异常处理器这些自定义的XX器，都是：   * 1. 自定义一个类继承SpringMVC中的某个类或者接口   2. 接着在SpringMVC.xml中配置 |

# 上传图片

## 配置虚拟目录

在tomcat上配置图片虚拟目录，在tomcat下conf/server.xml中添加：

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

访问http://localhost:8080/pic即可访问D:\develop\upload\temp下的图片。

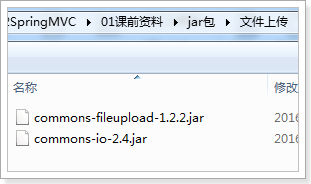
复制一张图片到存放图片的文件夹，使用浏览器访问

测试效果，如下图：



## 加入jar包

实现图片上传需要加入的jar包，如下图：



把两个jar包放到工程的lib文件夹中

## 配置上传解析器

在springmvc.xml中配置文件上传解析器

<beanid=*"****multipartResolver****"*

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

<!-- 设置文件上传大小 -->

<propertyname=*"maxUploadSize"*value=*"5000000"*/>

文件名编码格式

<property name="defaultEncoding" value="UTF-8"/>

</bean>

|  |
| --- |
| <!-- 文件上传,id必须设置为multipartResolver -->  原因：controller层的MultipartFile是个接口，它必须指明实现类，*CommonsMultipartResolver是它的实现类，*MultipartFile接口就是根据这个id在spring容器中查找*CommonsMultipartResolver。* |

## jsp页面修改

设置表单可以进行文件上传，如下图：



## 图片上传

在更新商品方法中添加图片上传逻辑

/\*\*

\* 更新商品

\*

\* **@param** item

\* **@return**

\* **@throws** Exception

\*/

@RequestMapping("updateItem")

**public** String updateItemById(Item item, MultipartFilepictureFile) **throws** Exception {

// 图片上传

// 设置图片名称，不能重复，可以使用uuid

String picName = UUID.*randomUUID*().toString().replaceAll(“-”,“”);

// 获取文件名

String oriName = pictureFile.getOriginalFilename();

// 获取图片后缀

String extName = oriName.substring(oriName.lastIndexOf("."));

// 开始上传

pictureFile.transferTo(**new** File("C:/upload/image/" + picName + extName));

// 设置图片名到商品中

item.setPic(picName + extName);

// ---------------------------------------------

// 更新商品

**this**.itemService.updateItemById(item);

**return**"forward:/itemEdit.action";

}

也可以在实体类中定义MultipartFile pictureFile属性，用来接收上传的文件。

# json数据交互

## @RequestBody

作用：

|  |
| --- |
| @RequestBody是作用在形参列表上，用于将前台发送过来固定格式的数据【xml 格式或者json等】封装为对应的 JavaBean 对象，封装时使用到的一个对象是系统默认配置的HttpMessageConverter进行解析，然后封装到形参上。 |

例子：



## @ResponseBody

作用：

|  |
| --- |
| @responseBody注解的作用是将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据。加上@responsebody后，返回结果直接写入**HTTP response body**中，不会被解析为跳转路径。 |

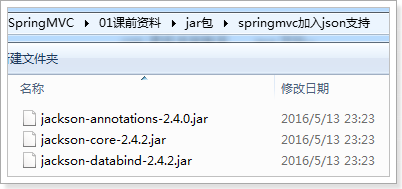
例子：可以是返回String（转成json格式的字符串），map。

## 请求json，响应json实现：

### 加入jar包

如果需要springMVC支持json，必须加入json的处理jar

我们使用Jackson这个jar，如下图：



### ItemController编写

/\*\*

\* 测试json的交互

\* **@param** item

\* **@return**

\*/

@RequestMapping("testJson")

// @ResponseBody

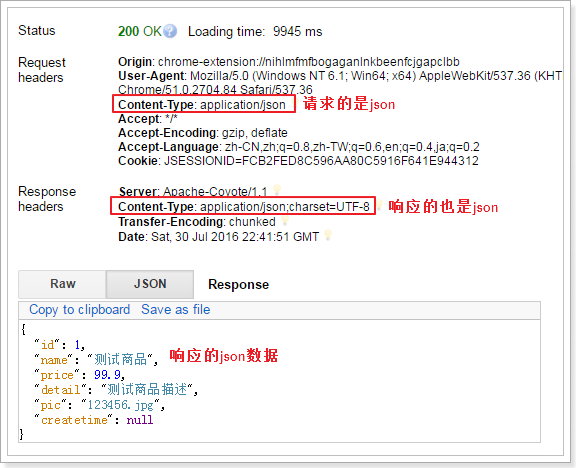
**public**@ResponseBody Item testJson(@RequestBody Item item) {

**return**item;

}

### 测试结果

如下图：



### 配置json转换器

如果不使用注解驱动<mvc:annotation-driven />，就需要给处理器适配器配置json转换器，参考之前学习的自定义参数绑定。**如果使用了注解驱动就不配了。**

在springmvc.xml配置文件中，给处理器适配器加入json转换器：

<!--处理器适配器 -->

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

<propertyname=*"messageConverters"*>

<list>

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

</list>

</property>

</bean>

# RESTful支持

## 什么是restful？

Restful就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种风格。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

资源：互联网所有的事物都可以被抽象为资源

资源操作：使用POST、DELETE、PUT、GET，使用不同方法对资源进行操作。

分别对应添加、删除、修改、查询。

传统方式操作资源

http://127.0.0.1/item/queryItem.action?id=1 查询,GET

http://127.0.0.1/item/saveItem.action 新增,POST

http://127.0.0.1/item/updateItem.action 更新,POST

http://127.0.0.1/item/deleteItem.action?id=1 删除,GET或POST

使用RESTful操作资源

http://127.0.0.1/item/1 查询,GET

http://127.0.0.1/item 新增,POST

http://127.0.0.1/item 更新,PUT

http://127.0.0.1/item/1 删除,DELETE

## 从URL上获取参数

使用RESTful风格开发的接口，根据id查询商品，接口地址是：

http://127.0.0.1/item/1

我们需要从url上获取商品id，步骤如下：

1. 使用注解@RequestMapping("item/{id}")声明请求的url

{xxx}叫做占位符，请求的URL可以是“item /1”或“item/2”

1. 使用(@PathVariable() Integer id)获取url上的数据

/\*\*

\* 使用RESTful风格开发接口，实现根据id查询商品

\*

\* **@param** id

\* **@return**

\*/

@RequestMapping("item/{id}")

@ResponseBody

**public** Item queryItemById(@PathVariable() Integer id) {

Item item = **this**.itemService.queryItemById(id);

**return**item;

}

如果@RequestMapping中表示为"item/{id}"，id和形参名称一致，@PathVariable不用指定名称。如果不一致，例如"item/{ItemId}"则需要指定名称@PathVariable("itemId")。

http://127.0.0.1/item/123?id=1

注意两个区别

1. @PathVariable是获取url上数据的。@RequestParam获取请求参数的（包括post表单提交）
2. 如果加上@ResponseBody注解，就不会走视图解析器，不会返回页面，目前返回的json数据。如果不加，就走视图解析器，返回页面

# 拦截器

## 定义

Spring Web MVC 的处理器拦截器类似于Servlet 开发中的过滤器Filter，用于对处理器进行预处理和后处理。

## 拦截器定义

实现HandlerInterceptor接口，如下：

**publicclass** HandlerInterceptor1 **implements** HandlerInterceptor {

// 发生于：页面渲染后

// 可用于：记录操作日志、得到执行controller时的异常信息

@Override

**Public void** afterCompletion(HttpServletRequest arg0, HttpServletResponsearg1, Object arg2, Exception arg3)

**throws** Exception {

System.***out***.println("页面渲染后");

}

// 发生于：controller执行后但未返回视图前调用此方法

// 可用于：这里可在返回用户前对模型数据进行加工处理，比如这里加入公用信息以便页面显示

@Override

**Public void** postHandle(HttpServletRequestarg0, HttpServletResponsearg1, Object arg2, ModelAndViewarg3)

**throws** Exception {

System.***out***.println("方法后，页面渲染前");

}

// 发生于：执行处理器映射器之前

// 返回true表示继续执行，返回false中止执行

// 这里可以加入登录校验、权限拦截等

@Override

**Public boolean** preHandle(HttpServletRequestarg0, HttpServletResponsearg1, Object arg2) **throws** Exception {

System.***out***.println("方法前");

// 设置为true，测试使用

**Return true**;

}

}

## 拦截器配置

在springmvc.xml中配置拦截器

<!-- 配置拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!--所有的请求都进入拦截器 -->

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

<!-- 配置具体的拦截器 -->

<bean class=*"cn.itcast.ssm.interceptor.HandlerInterceptor1"*/>

</mvc:interceptor>

<mvc:interceptor>

。。可以配多个拦截器

</mvc:interceptor>

</mvc:interceptors>

## 正常流程测试

浏览器访问地址

http://127.0.0.1:8080/springmvc-web2/itemList.action

### 运行流程

控制台打印：

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor2..postHandle..

HandlerInterceptor1..postHandle..

HandlerInterceptor2..afterCompletion..

HandlerInterceptor1..afterCompletion..

## 中断流程测试

浏览器访问地址

http://127.0.0.1:8080/springmvc-web2/itemList.action

### 运行流程

HandlerInterceptor1的preHandler方法返回false，HandlerInterceptor2返回true，运行流程如下：

HandlerInterceptor1..preHandle..

从日志看出第一个拦截器的preHandler方法返回false后第一个拦截器只执行了preHandler方法，其它两个方法没有执行，第二个拦截器的所有方法不执行，且Controller也不执行了。

HandlerInterceptor1的preHandler方法返回true，HandlerInterceptor2返回false，运行流程如下：

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor1..afterCompletion..

从日志看出第二个拦截器的preHandler方法返回false后第一个拦截器的postHandler没有执行，第二个拦截器的postHandler和afterCompletion没有执行，且controller也不执行了。

总结：

preHandle按拦截器定义顺序调用

postHandler和afterCompletion按拦截器定义逆序调用

postHandler在拦截器链内所有拦截器返成功调用

afterCompletion只有preHandle返回true才调用

## 拦截器应用

### 处理流程

1. 有一个登录页面，需要写一个Controller访问登录页面
2. 登录页面有一提交表单的动作。需要在Controller中处理。
   1. 判断用户名密码是否正确（在控制台打印）
   2. 如果正确,向session中写入用户信息（写入用户名username）
   3. 跳转到商品列表
3. 拦截器。
   1. 拦截用户请求，判断用户是否登录（登录请求不能拦截）
   2. 如果用户已经登录。放行
   3. 如果用户未登录，跳转到登录页面。

### 编写拦截器

@Override

**publicboolean**preHandle(HttpServletRequestrequest, HttpServletResponseresponse, Object arg2) **throws** Exception {

// 从request中获取session

HttpSessionsession = request.getSession();

// 从session中获取username

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

// 判断username是否为null

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

// 如果不为空则放行

**Return true**;

} **else** {

// 如果为空则跳转到登录页面

response.sendRedirect(request.getContextPath() + "/user/toLogin.action");

}

**Returnfalse**;

}