

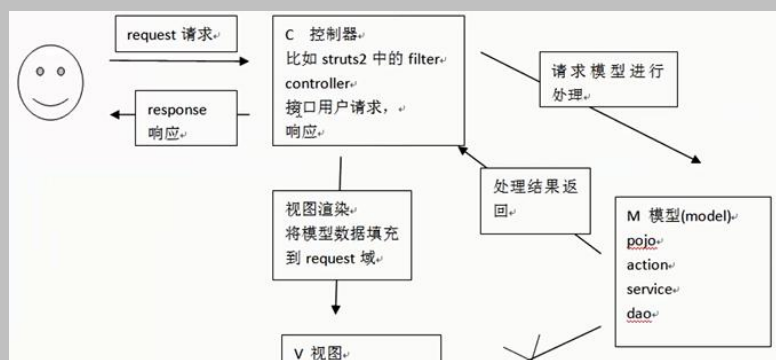
# 1. Spring MVC 框架

## 1.1. 什么是 spring MVC

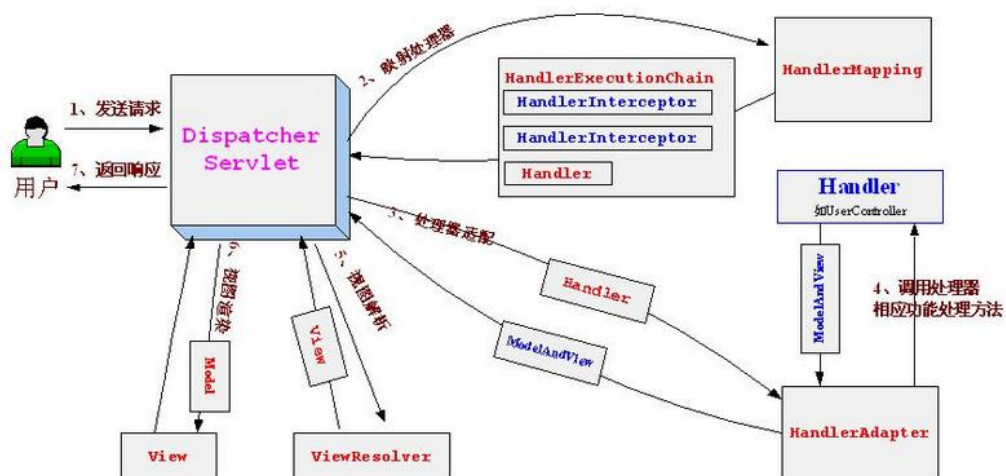
- 1、 SpringMVC 是 spring 的一个模块，springMVC 和 spring 无需通过中间层进行整合。
- 2、 SpringMVC 是一个基于 MVC 的 web 框架。

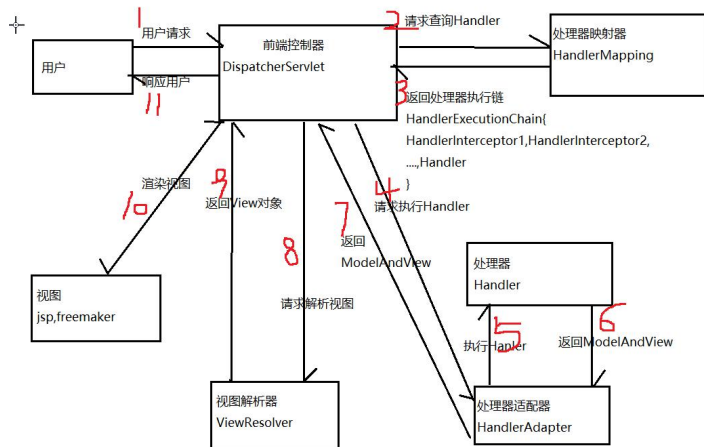
## 1.2. 什么是 MVC

1、 MVC 是一种设计思想



## 1.3. SpringMVC 框架原理





第一步：发起请求到前端控制器(DispatcherServlet)。

第二步：前端控制器请求 HandlerMapping 查找 Handler。

可以根据 xml 配置、注解进行查找。

第三步：处理器映射器 HandlerMapping 向前端控制器返回 Handler。

第四步：前端控制器调用处理器适配器去执行 Handler。

第五步：处理器适配器去执行 Handler。

第六步：Handler 执行完成给适配器返回 ModelAndView。

第七步：处理器适配器向前端控制器返回 ModelAndView。

ModelAndView 是 springmvc 框架的一个底层对象，包括 Model 和 view。

第八步：前端控制器请求视图解析器去进行视图解析。

根据逻辑视图名解析成真正的视图(jsp)。

第九步：视图解析器向前端控制器返回 View。

第十步：前端控制器进行视图渲染。

视图渲染将模型数据(在 ModelAndView 对象中)填充到 request 域。

第十一步：前端控制器向用户响应结果。

组件：

1、前端控制器 DispatcherServlet (不需要程序员开发)。

作用接收请求，响应结果，相当于转发器，中央处理器。

有了 DispatcherServlet 减少了其它组件之间的耦合度。

2、处理器映射器 HandlerMapping(不需要程序员开发)↵

作用：根据请求的 url 查找 Handler↵

↵

↵

↵

3、处理器适配器 HandlerAdapter↵

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler↵

↵

4、处理器 Handler(需要程序员开发)↵

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler↵

↵

5、视图解析器 View resolver(不需要程序员开发)↵

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）↵

↵

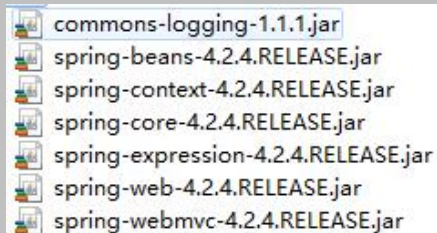
6、视图 View(需要程序员开发 jsp)↵

View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker、pdf...）↵

↵

## 1. 4. SpringMVC 框架搭建

### 1. 4. 1. 必须 jar 包



commons-logging-1.1.1.jar  
spring-beans-4.2.4.RELEASE.jar  
spring-context-4.2.4.RELEASE.jar  
spring-core-4.2.4.RELEASE.jar  
spring-expression-4.2.4.RELEASE.jar  
spring-web-4.2.4.RELEASE.jar  
spring-webmvc-4.2.4.RELEASE.jar

### 1. 4. 2. Springmvc 配置文件约束头

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:mvc="http://www.springframework.org/schema/mvc"

    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
```

```

http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd">
</beans>

```

### 1. 4. 3. 配置前端控制器

```

<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param><!-- 配置文件主要配置处理器映射器、适配器等等 -->
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:springmvc.xml</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <!--
    url-pattern:三种配置方式
    1、*.action 访问以.action结尾的请求由DispatcherServlet来解析
    2、/ 所有请求包括静态资源(js、css、image等等)都有DispatcherServlet解析,对于静态文件的解析需要配置不让DispatcherServlet解析
    3、/*这种配置方式不对
    -->
  <url-pattern>*.action</url-pattern>
</servlet-mapping>

```

关于 url-pattern 说明:

- ①、**【/】**：它会拦截所有的 url，如：/test、/test1.html、/1.jpg.....，但是除了以 jsp 结尾的 url 不会交给前端控制器。
- ②、**【/\*】**：他是包含 **【/】** 的，可以多拦截以 \*.jsp 结尾的 url
- ③、**【\*.xxx】**：这个拦截固定结尾的 url 请求，常见的有 \*.do、\*.json、\*.action

### 1. 4. 4. 配置处理器映射器

```

<!-- 处理器映射器 将bean的name作为url进行查找，需要在配置handler时指定beanname(就是url) -->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

```

### 1. 4. 5. 配置处理器适配器

```

<!-- 处理器适配器 所有处理器适配器都实现HandlerAdapter接口 -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

```

能处理实现了 controller 接口的 controller

### 1. 4. 6. 创建并配置处理器

```
public class StudentController implements Controller{
    @Override
    public ModelAndView handleRequest(javax.servlet.http.HttpServletRequest httpServl
        ModelAndView modelAndView=new ModelAndView();
        List<String> list=new ArrayList<>();
        list.add("张三");
        list.add("李四");
        modelAndView.addObject( attributeName: "list",list);
        modelAndView.setViewName("/list");
        return modelAndView;
    }
}
```

类似 struts2 的 action

```
<!-- 配置处理器 -->
<bean name="/user.action" class="com.lx.springmvc.controller.UserController"></bean>
```

### 1. 4. 7. 配置视图解析器

```
<!--
    视图解析器 解析jsp 默认使用jstl
    prefix:在视图逻辑名前面加上一部分,
    suffix:在视图逻辑名后面加上一部分;
-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="view" />
    <property name="suffix" value=".jsp" />
</bean>
```

### 1. 4. 8. 非注解的处理器映射器和适配器（了解）

#### 1. 4. 8. 1. 映射器

##### 1. 4. 8. 1. 1. BeanNameUrlHandlerMapping

```
<!-- 处理器映射器 将bean的name作为url进行查找, 需要在配置handler时指定beanname(就是url) -->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
```

##### 1. 4. 8. 1. 2. SimpleUrlHandlerMapping

```
<!-- 简单url映射 需要配置请求和响应 prop的key值就是请求, 后面指定处理这个请求的Controller-->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/user1.action">user</prop>
        </props>
    </property>
</bean>
```



```

<!-- 配置处理器 -->
<bean id="user" name="/user.action" class="com.lx.springmvc.controller.UserController"></bean>

<!-- 简单url映射 SimpleUrlHandlerMapping需要配置请求和响应 prop的key值就是请求，后面指定处理这个请求的Controller-->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/user1.action">user</prop>
        </props>
    </property>
</bean>

```

结论：多个映射器可以并存，前端控制器判断url能让哪些映射器映射，就让正确的映射器处理。

## 1. 4. 8. 2. 适配器

### 1. 4. 8. 2. 1. SimpleControllerHandlerAdapter

要求编写的 controller 必须实现 controller 接口

<!-- 处理器适配器 所有处理器适配器都实现HandlerAdapter接口 -->

```
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"></bean>
```

```

public class UserController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<String> list=new ArrayList<>();
        list.add("zhangsan");
        list.add("lisi");
        ModelAndView model=new ModelAndView();
        model.addObject("list", list);
        model.setViewName("list.jsp");
        return model;
    }
}

```

### 1. 4. 8. 2. 2. HttpRequestHandlerAdapter

要求编写的 controller 实现 HttpRequestHandler 接口

<!-- http请求处理器适配器 -->

```
<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"></bean>
```

```

public class User1Controller implements HttpRequestHandler {

    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    }

}

```

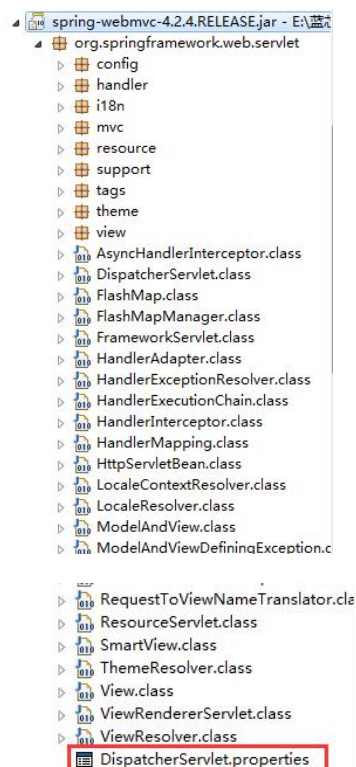
从上面可以看出此适配器的handleRequest方法没有返回ModelAndView,可通过response修改定义内容,比如返回json数据:

```

response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
response.getWriter().write("json串");

```

## 1. 4. 9. DispatcherServlet.properties



总结：前端控制器从上面的配置文件中加载处理映射器、适配器、视图解析器等组件。如果不在 springmvc.xml 配置，使用默认加载的。

## 2. 注解处理器映射器和适配器（重点掌握）

### 2.1. 映射器

在 spring3.1 之前用 org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

在 spring3.1 之后用 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

### 2.2. 适配器

在 spring3.1 之前用 org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

在 spring3.1 之后用 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

### 2.3. 配置注解映射器和适配器（重点掌握）

```
<!-- 注解的映射器 -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"></bean>
<!-- 注解适配器 -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"></bean>

<!-- 使用mvc:annotation-driven代替上面注解的映射器和适配器 实际开发时用这个 -->
<mvc:annotation-driven></mvc:annotation-driven>
```

### 2.3.1. 常用注解

注意：注解的映射器和适配器要配套使用

#### 2.3.1.1. @Controller

```
//Controller标识它是一个控制器
@Controller
public class UserController2{
    //RequestMapping实现对queryUser方法进行映射，一个方法对应一个url
    @RequestMapping("/queryUser.action")
    public ModelAndView queryUser(){
        List<String> list=new ArrayList<>();
        list.add("zhangsan");
        list.add("lisi");
        ModelAndView model=new ModelAndView();
        model.addObject("list", list);
        model.setViewName("list.jsp");
        return model;
    }
}
```

#### 2.3.1.2. 开启注解扫描

在 springmvc.xml 配置注解扫描

```
<!-- 扫描带有注解的类，base-package:controller所在的包-->
<context:component-scan base-package="com.Lx.springmvc.controller"></context:component-scan>
```

#### 2.3.1.3. @RequestMapping

##### 2.3.1.3.1. url 映射

最根本作用，上面 2.3.1.1 有介绍

##### 2.3.1.3.2. 通配符访问

@RequestMapping 中还支持通配符 “\* ”

```
@Controller
@RequestMapping ( "/myTest" )
public class MyController {
    @RequestMapping ( "*/wildcard" )
    public String testWildcard() {
        System.out.println( "wildcard-----" );
        return "wildcard" ;
    }
}
```



### 2.3.1.3.3. 窄化请求映射

```
@Controller
//为了对url进行分类管理，可以在这里定义根路径，最终访问url是跟路径+子路径
//用户列表就变成: /user/queryUser.action
@RequestMapping("/user")
public class UserController2{
```

### 2.3.1.3.4. 限制http的请求方式

出于安全考虑

如果限制为 post 方法，进行 get 请求报错。

```
@RequestMapping(value="/queryUser.action",method={RequestMethod.POST})
public ModelAndView queryUser(){
    List<String> list=new ArrayList<>();
    list.add("zhangsan");
    list.add("lisi");
    ModelAndView model=new ModelAndView();
    model.addObject("list", list);
    model.setViewName("list.jsp");
    return model;
}
```



### 2.3.1.4. @ModelAttribute

被@ModelAttribute 注释的方法会在此 controller 每个方法执行前被执行

#### 2.3.1.4.1. 注释void 返回值的方法

```
@Controller
public class HelloModelController {

    @ModelAttribute
    public void populateModel(@RequestParam String abc, Model model) {
        model.addAttribute("attributeName", abc);
    }

    @RequestMapping(value = "/helloWorld")
    public String helloWorld() {
        return "helloWorld.jsp";
    }
}
```

在这个代码中，访问控制器方法helloWorld时，会首先调用populateModel方法，将页面参数abc(/helloWorld.ht?abc=text)放到model的attributeName属性中，在视图中可以直接访问。

### 2.3.1.4.2. 注释返回具体类的方法

```
@Controller
public class Hello2ModelController {

    @ModelAttribute
    public User populateModel() {
        User user=new User();
        user.setAccount("ray");
        return user;
    }
    @RequestMapping(value = "/helloWorld2")
    public String helloWorld() {
        return "helloWorld.jsp";
    }
}
```

当用户请求 `http://localhost:8080/test/helloWorld2`时，首先访问`populateModel`方法，返回`User`对象，`model`属性的名称没有指定，它由返回类型隐含表示，如这个方法返回`User`类型，那么这个`model`属性的名称是`user`。这个例子中`model`属性名称有返回对象类型隐含表示，`model`属性对象就是方法的返回值。它无需要特定的参数。

### 2.3.1.4.3. 也可以指定属性名称

```
@Controller
public class Hello2ModelController {

    @ModelAttribute(value="myUser")
    public User populateModel() {
        User user=new User();
        user.setAccount("ray");
        return user;
    }
    @RequestMapping(value = "/helloWorld2")
    public String helloWorld(Model map) {
        return "helloWorld.jsp";
    }
}
```

**jsp中如下访问：**

```
<c:out value="${myUser.account}"></c:out>
```

### 2.3.1.4.4. 标记在方法的形参上

运用在参数上，会将客户端传递过来的参数按名称注入到指定对象中，并且会将这个对象自动加入 `ModelMap` 中，便于 `View` 层使用：

```
@RequestMapping("/save.do")
public String save(@ModelAttribute("stu") Student student){
    return "list.jsp";
}
```

可以使用在表单提交失败需要再回到表单页面重新填写，原来提交的数据需要重新在页面上显示。

## 2. 3. 1. 5. @SessionAttrbute

### 2. 3. 1. 5. 1. 标注在类上

Model 当中 key 为 name 的存在了 session 当中

```
@Controller
// 把model当中key为name的存在了session当中
@SessionAttributes("name")
public class MyController {
    @RequestMapping("/testsession")
    public String etstsession(Model model){
        // 把数据存到了request域
        model.addAttribute("name","dell");
        return "/result2.jsp";
    }
}
```

```
@Controller
// 把model当中String类型的属性都存在了session当中
@SessionAttributes(types=String.class)
public class MyController {
    @RequestMapping("/testsession")
    public String etstsession(Model model){
        // 把数据存到了request域
        model.addAttribute("name","dell");
        return "/result2.jsp";
    }
}
```

### 2. 3. 1. 5. 2. 标注在方法的形参上

@SessionAttribute 在 spring5 以后才有。

使用SessionAttribute来访问预先存在的全局会话属性，注解标记在处理器映射方法上来获取参数

假设提前在session域中存了key值为name的键值对      如若没有，服务器会报错

```
1  @RequestMapping("testsession2")
2      public String testsession2(@SessionAttribute("name") String name){
3          System.out.println(name);
4          return "/result2.jsp";
5      }
```

## 2. 3. 1. 6. @RequestParam

当请求的参数名称和处理器形参名称一致时会将请求参数与形参进行绑定，若不一致，就需要@RequestParam

value: 参数名字，即入参的请求参数名字，如 value="itemId"表示请求的参数 区中的名字为 itemId 的参数的值将传入

required: 是否必须，默认是 true，表示请求中一定要有相应的参数，否则将报错

HTTP Status 400 - Required Integer parameter 'XXXX' is not present

defaultValue: 默认值, 表示如果请求中没有同名参数时的默认值

```
public String queryItemById(@RequestParam(value = "itemId", required = true, defaultValue = "1") Integer id, ModelMap modelMap) {}
```

这里需要传入的是 id, 实际传入的是 itemId, 需要用 @RequestParam 转换一下

```
@Controller
@RequestMapping("/jsontest")
public class JsonTestController {
    @RequestMapping(value="/getJson.action")
    public String getJson(@RequestParam Map<String, Object> map) {
        System.out.println("----"+map);
        return "";
    }
}
```

## 2.3.1.7. @RequestBody

### 2.3.1.7.1. 程序流程

1. 前台使用 ajax 技术, 传递 json 字符串到后台;
2. 后台使用 Spring MVC 注解 @RequestBody 接受前台传递的 json 字符串, 并返回新的 json 字符串到前台;
3. 前台接受后台传递过来的 json 数据, 并显示。

### 2.3.1.7.2. 前端页面

```
function loadData() {
    var actionUrl = basePath + "test/isonDataReq.action";
    var params = JSON.stringify([ {
        username : "ZhangSan",
        passwd : "123456"
    }, {
        username : "LiSi",
        passwd : "8888"
    } ] );
    $.ajax({
        type: 'POST',
        url: actionUrl,
        data: params,
        dataType: "json", // 后台返回的数据类型
        cache: false,
        contentType: "application/json", // 必须, 不可少 @RequestBody 需要根据这个来判定
        // 需要使用 jsonHttpMessageCon
        error: function(textStatus, errorThrown) {
            alert("系统请求错误: " + textStatus);
        },
        success: function(data, textStatus) {
            alert("请求成功==>姓名: "+data.username+", 密码: "+data.passwd);
        }
    });
}
```

### 2.3.1.7.3. 后端程序

**@RequestBody 接收Json数组对象**

点击后会请求jsonDataReq

```

/**
 * SpringMVC @RequestBody 接收Json数组对象
 * @param persons 写jsonDataReq.action也可以，见配置
 */
@RequestMapping(value = "/jsonDataReq", method = { RequestMethod.POST })
@ResponseBody 完成person对象到json数据的转换
public Person jsonDataReq(@RequestBody Person[] persons) {
    for (Person person : persons) { 接受前台传递的json对象数组
        System.out.println("用户名: " + person.getUsername() + ", 密码: "
            + person.getPasswd());
    }
    return new Person("王五", "666666");
}

```

## 2.3.1.7.4. 实体类

```

package com.ll.model;

public class Person {
    private String username;
    private String passwd;

    public Person() {
        super();
    }
}

```

## 2.3.1.7.5. Jackson

```

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.4</version>
</dependency>

```

**注意：** jackson 的版本和 spring 的版本对应关系。

## 2.3.1.8. @ResponseBody

作用： 该注解用于将 Controller 的方法返回的对象，通过适当的 `HttpMessageConverter` 转换为指定格式后，写入到 Response 对象的 body 数据区。

使用时机： 返回的数据不是 html 标签的页面，而是其他某种格式的数据时（如 json、xml 等）使用；

#### 2.3.1.8.1. 在方法上添加@ResponseBody 注解

```
@RequestMapping("/test")
public class TestController{

    @RequestMapping(value="/list.action")
    @ResponseBody
    public List list(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse)
    {
        List<String> list=new ArrayList<String>();
        list.add("wangwu");
        list.add("zhangsan");
        return list;
    }
}
```

#### 2.3.1.8.2. 添加 jackson 依赖

```
<dependency>

    <groupId>com.fasterxml.jackson.core</groupId>

    <artifactId>jackson-databind</artifactId>

    <version>2.9.4</version>

</dependency>
```

#### 2.3.1.8.3. 实现原理

当一个处理请求的方法标记为@ResponseBody 时,就说明该方法需要输出其他视图 (json、xml), SpringMVC 通过已定义转换器做转化输出,默认输出 json。

其实是注解驱动帮我们做了这件事情。

```
<!-- 注解驱动 -->
<mvc:annotation-driven/>
```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
258
```



```

http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd">
<!-- 可以代替注解的映射器和注解的适配器-->
<mvc:annotation-driven conversion-service="formattingConversionService">
<!-- 关闭默认的消息转换器-->
<mvc:message-converters register-defaults="false">
<!-- 配置fastjson转换器-->
<bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter"
<property name="supportedMediaTypes">
<list>
<!-- 这里顺序不能反，一定要先写text/html，不然IE下会出现下载提示-->
<value>text/html; charset=UTF-8</value>
<value>application/json; charset=UTF-8</value>
</list>
</property>
</bean>
</mvc:message-converters>
</mvc:annotation-driven>

```

## 2.3.1.9. @PathVariable

带占位符的 URL 是 Spring3.0 新增的功能，该功能在 SpringMVC 向 REST 目标挺进发展过程中具有里程碑的意义

通过 @PathVariable 可以将 URL 中占位符参数绑定到控制器处理方法的入参中：URL 中的 {xxx} 占位符可以通过 @PathVariable(“xxx”) 绑定到操作方法的入参中。

用来接收路径参数，如/user/001，可接收 001 作为参数，此注解放置在参数前。

//@PathVariable 可以用来映射 URL 中的占位符到目标方法的参数中

```

@Controller
public class TestController {

    @RequestMapping(value="/user/{userId}/roles/{roleId}",method = RequestMethod.GET)

    public String getLogin(@PathVariable("userId") String userId,

        @PathVariable("roleId") String roleId){

        System.out.println("User Id : " + userId);

        System.out.println("Role Id : " + roleId);

        return "hello";

    }

    @RequestMapping(value="/product/{productId}",method = RequestMethod.GET)

    public String getProduct(@PathVariable("productId") String productId){

        System.out.println("Product Id : " + productId);

        return "hello";

    }

    @RequestMapping(value="/javabeat/{regex1:[a-z-]+}",

        method = RequestMethod.GET)

    public String getRegExp(@PathVariable("regex1") String regex1){

        System.out.println("URI Part 1 : " + regex1);

        return "hello";

    }

}

```

### 2. 3. 1. 10. @RestController

是一个组合注解，组合了@Controller和@ResponseBody,这就意味着当你开发一个和页面交互数据的控制的时候，需要使用此注解。

```
@RestController
@RequestMapping(value="/user",produces =
{MediaType.APPLICATION_JSON_VALUE})
public class TestController {

    @RequestMapping(value = "{name}", method = RequestMethod.GET)
    public User getUser(@PathVariable String name) {
        User user = new User();
        user.setUserName(name);
        return user;
    }
}
```

### 2. 3. 1. 11. @PostMapping

Spring4.3 中引进了 { @GetMapping、@PostMapping、@PutMapping、@DeleteMapping、@PatchMapping }，来帮助简化常用的 HTTP 方法的映射，并更好地表达被注解方法的语义。

### 2. 3. 1. 12. @GetMapping

## 2. 3. 2. HttpMessageConverter

## 2. 3. 3. Controller 方法返回值

### 2. 3. 3. 1. 返回 ModelAndView

需要在方法结束时，定义 ModelAndView，将 model 和 view 分别进行设置，由视图解析器进行解析。

```
public class UserController implements Controller{

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<String> list=new ArrayList<>();
        list.add("zhangsan");
        list.add("lisi");
        ModelAndView model=new ModelAndView();
        model.addObject("list", list);
        model.setViewName("list.jsp");
        return model;
    }
}
```



### 2.3.3.2. 返回 String

如果 controller 方法返回 String,表示返回逻辑视图名。

真正视图（jsp 路径）=前缀+逻辑视图名+后缀

```
@RequestMapping(value="/addUser.action")
public String addUser(Model model){
    List<String> list=new ArrayList<>();
    list.add("zhangsan1");
    list.add("lisi1");
    model.addAttribute("list", list);
    return "/list.jsp";
}
```

#### 2.3.3.2.1. 转发 forward

```
@RequestMapping(value="/addUser.action")
public String addUser(Model model){
    List<String> list=new ArrayList<>();
    list.add("zhangsan1");
    list.add("lisi1");
    model.addAttribute("list", list);
    return "forward:queryUser.action";
}
```

#### 2.3.3.2.2. 重定向 redirect

```
@RequestMapping(value="/addUser.action")
public String addUser(Model model){
    List<String> list=new ArrayList<>();
    list.add("zhangsan1");
    list.add("lisi1");
    model.addAttribute("list", list);
    return "redirect:queryUser.action";
}
```

### 2.3.3.3. 返回 void

一般适用于服务器端响应 json 格式数据给客户端。

在 controller 方法形参上可以定义 request 和 response，使用 request 或 response 指定响应结果：↵

1、使用 request 转向页面，如下：↵

```
request.getRequestDispatcher("页面路径").forward(request, response);↵
```

↵

2、也可以通过 response 页面重定向：↵

```
response.sendRedirect("url");↵
```

↵

3、也可以通过 response 指定响应结果，例如响应 json 数据如下：↵

```
response.setCharacterEncoding("utf-8");↵
```

```
response.setContentType("application/json;charset=utf-8");↵
```

```
response.getWriter().write("json 串");↵
```

```
@RequestMapping(value="/isUser.action")
public void isUser(HttpServletRequest req, HttpServletResponse resp) throws IOException{
    List<String> list=new ArrayList<String>();
    list.add("zhangsan1");
    list.add("lisi1");
    resp.setContentType("application/json;charset=utf-8");
    resp.getWriter().write("{\"result\":true}");
}
```

## 3. Springmvc 参数绑定

### 3.1. Springmvc 编码过滤器


```
<filter>
    <filter-name>encoding</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

### 3.2. 参数绑定过程

从客户端请求 key/value 数据，经过参数绑定，将 key/value 数据绑定到 controller 方法形参上。

Springmvc 中，接收页面提交的数据是通过方法形参来接收，而不是在 controller 中定义成员变量来接收。

```
graph LR
    A[客户端请求  
Key/value] --> B[Controller Method Parameter]
```



处理器适配器调用 springmvc 提供的参数绑定组件将参数赋值给形参。

### 3.3. 默认支持的类型

直接 controller 方法形参上定义下面的类型，就可以直接使用这些对象。在参数绑定的过程中，如果遇到下面的类型则直接进行绑定。

#### 3.3.1. HttpServletRequest

#### 3.3.2. HttpServletResponse

#### 3.3.3. HttpSession

#### 3.3.4. Model/ModelMap

Model 是一个接口，ModelMap 是一个接口实现。

作用：把模型数据填充到 request 域。

### 3.4. 简单类型绑定

通过@RequestParam 对简单类型进行绑定

如果没有使用@RequestParam，要求 request 传入的参数名必须和 controller 方法形参一致，方可绑定成功。

如果使用@RequestParam 不用限制 request 传入参数名称和 controller 方法形参一致

例如：

```
@RequestMapping(value="/testParam.action")
public String testParam(@RequestParam(value="id") Integer ids){
    System.out.println("id:"+ids);
    return "forward:queryUser.action";
}
```

把请求参数中参数名叫 id 的参数值赋值给 ids 这个形参。

### 3.4.1. 整形

### 3.4.2. 字符串

### 3.4.3. 单精度/双精度

### 3.4.4. 布尔类型

对于布尔类型的参数，请求的参数值为 true 或 false。或者 1 或 0

## 3.5. Pojo 绑定

页面中 input 的 name 和 controller 的 pojo 形参中的属性名称一致，将页面中数据绑定到 pojo。

页面：

```
<form action="user/addUser.action" method="post">
  <table border="1">
    <tr>
      <td>用户名: </td><td><input type="text" name="username" /> </td>
      <td>年龄: </td><td><input type="text" name="age" /> </td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="保存" /> </td>
    </tr>
  </table>
</form>
```

Controller:

```
@RequestMapping(value="/addUser.action")
public String addUser(Model model User user) {
    System.out.println(user.getUsername()+"--"+user.getAge());
    return "forward:queryUser.action";
}
```

实体类:

```
public class User {
    private String username;
    private Integer age;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
}
```



### 3.6. 数组

localhost/test/query.do?hobby=1&hobby=3&hobby=8

```
@RequestMapping(value = "/query.do", method = {RequestMethod.POST, Req  
public void query(String[] hobby){  
    System.out.println("ids:" + Arrays.toString(hobby));  
}
```

### 3.7. 集合

```
<form action="/test/query.do" method="get">  
    <input type="text" name="studentList[0].id" value="1">  
    <input type="text" name="studentList[0].name" value="3">  
    <input type="text" name="studentList[0].sex" value="6">  
    <input type="text" name="studentList[0].age" value="8">  
  
    <input type="text" name="studentList[1].id" value="1">  
    <input type="text" name="studentList[1].name" value="3">  
    <input type="text" name="studentList[1].sex" value="6">  
    <input type="text" name="studentList[1].age" value="8">  
  
    <input type="text" name="studentList[2].id" value="1">  
    <input type="text" name="studentList[2].name" value="3">  
    <input type="text" name="studentList[2].sex" value="6">  
    <input type="text" name="studentList[2].age" value="8">  
  
    <input type="submit">  
  
</form>
```

```
@RequestMapping(value = "/query.do", method = {RequestMethod.PO  
public void query(Students student){  
    for(Student student1:student.getStudentList()){  
        System.out.println(student1);  
    }  
}
```

```
public class Students {  
    private List<Student> studentList;  
  
    public List<Student> getStudentList() {  
        return studentList;  
    }  
  
    public void setStudentList(List<Student> studentList) {  
        this.studentList = studentList;  
    }  
}
```

### 3. 8. Map

```
<form action="/test/query.do" method="get">
  <input type="text" name="studentList[0].id" value="1">
  <input type="text" name="studentList[0].name" value="3">
  <input type="text" name="studentList[0].sex" value="6">
  <input type="text" name="studentList[0].age" value="8">

  <input type="text" name="studentList[1].id" value="1">
  <input type="text" name="studentList[1].name" value="3">
  <input type="text" name="studentList[1].sex" value="6">
  <input type="text" name="studentList[1].age" value="8">

  <input type="text" name="studentList[2].id" value="1">
  <input type="text" name="studentList[2].name" value="3">
  <input type="text" name="studentList[2].sex" value="6">
  <input type="text" name="studentList[2].age" value="8">

  <input type="submit">

</form>

@RequestMapping(value = "/query.do",method = {RequestMethod.POST,RequestMethod.GET})
public void query(Students student){
    Set<String> list=student.getStudentMap().keySet();
    Iterator<String> iterator=list.iterator();
    while (iterator.hasNext()){
        System.out.println(student.getStudentMap().get(iterator.next()));
    }
}

public class Students {
    private List<Student> studentList;

    private Map<String,Student> studentMap;

    public Map<String, Student> getStudentMap() {
        return studentMap;
    }

    public void setStudentMap(Map<String, Student> studentMap) {
        this.studentMap = studentMap;
    }

    public List<Student> getStudentList() {
```

### 3. 9. 前端传JSON

#### 3. 9. 1. 传 json 对象

### 3.9.2. 传 json 数组

#### 3.9.2.1. 后端用数组接收

#### 3.9.2.2. 后端用集合接收

## 3.10. 自定义参数绑定

由于日期数据有很多种格式，springmvc 没办法把字符串转换成日期类型。所以需要自定义参数绑定  
一般使用<mvc:annotation-driven/>注解驱动加载处理器适配器，可以在此标签上进行配置。

### 3.10.1. 注解方式

传递的参数为日期，[spring](#) 不知道该以什么格式转换为 Date 类型，解决办法为在实体类的日期属性上加上  
@DateTimeFormat(pattern="yyyy-MM-dd")注解即可

```
public class User{  
    private String id;  
    private String name;  
    private String sex;  
    @DateTimeFormat(pattern="yyyy-MM-dd")  
    private Date birthday;
```

### 3.10.2. 自定义类型转换器

第一步

定义类型转换类实现 org.springframework.core.convert.converter.Converter 接口

```
import org.springframework.core.convert.converter.Converter;  
  
public class DateConverter implements Converter<String, Date> {  
    @Override  
    public Date convert(String source) {  
        //实现 将日期串转成日期类型(格式是yyyy-MM-dd HH:mm:ss)  
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
        try {  
            //转成直接返回  
            return simpleDateFormat.parse(source);  
        } catch (ParseException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        //如果参数绑定失败返回null  
        return null;  
    }  
}
```

第二步

在 springmvc.xml 里面配置自定义参数绑定

```
<!-- 自定义参数绑定 -->
<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <!-- 转换器 -->
    <property name="converters">
        <list>
            <!-- 日期类型转换 -->
            <bean class="com.Lx.springmvc.Converter.DateConverter"/>
        </list>
    </property>
</bean>
```

### 第三步

在注解驱动器里面引用这个自定义参数绑定

```
<!-- 使用mvc:annotation-driven代替上面注解的映射器和适配器 实际开发时用这个-->
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>
```

## 3. 11. 高级类型参数绑定

### 3. 11. 1. 集合类型绑定

适用于批量添加和批量修改这样的场景。

```
public class User {
    private String username;
    private Integer age;
    private Date birthday;
    private List<User> list;
    public List<User> getList() {
        return list;
    }
    public void setList(List<User> list) {
        this.list = list;
    }
    public String getUsername() {
        return username;
    }
    public Date getBirthday() {
        return birthday;
    }
}

@RequestMapping(value="/addUser.action")
public String addUser(Model model,User user){
    for (int i = 0; i < user.getList().size(); i++) {
        System.out.println(user.getList().get(i).getUsername());
    }
    return "forward:queryUser.action";
}
```



```

<form action="user/addUser.action" method="post">
  <table border="1">
    <tr>
      <td>用户名: </td><td><input type="text" name="list[0].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[0].age"> </td>
      <td>生日: </td><td><input type="text" name="list[0].birthday"> </td>
    </tr>
    <tr>
      <td>用户名: </td><td><input type="text" name="list[1].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[1].age"> </td>
      <td>生日: </td><td><input type="text" name="list[1].birthday"> </td>
    </tr>
    <tr>
      <td>用户名: </td><td><input type="text" name="list[2].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[2].age"> </td>
      <td>生日: </td><td><input type="text" name="list[2].birthday"> </td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="保存"> </td>
    </tr>
  </table>
</form>

```

下标

### 3.11.2. 数组绑定

适应于批量删除数据，得到一组 id。

```

@RequestMapping(value="/addUser.action")
public String addUser(Model model, User user, Integer[] userid){
  /*for (int i = 0; i < user.getList().size(); i++) {
    System.out.println(user.getList().get(i).getUsername());
  }*/

  for (int i = 0; i < userid.length; i++) {
    System.out.println(userid[i]);
  }
  return "forward:queryUser.action";
}

```

```

<form action="user/addUser.action" method="post">
  <table border="1">
    <tr>
      <input type="hidden" name="userid" value="1">
      <td>用户名: </td><td><input type="text" name="list[0].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[0].age"> </td>
      <td>生日: </td><td><input type="text" name="list[0].birthday"> </td>
    </tr>
    <tr>
      <input type="hidden" name="userid" value="2">
      <td>用户名: </td><td><input type="text" name="list[1].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[1].age"> </td>
      <td>生日: </td><td><input type="text" name="list[1].birthday"> </td>
    </tr>
    <tr>
      <input type="hidden" name="userid" value="3">
      <td>用户名: </td><td><input type="text" name="list[2].username"> </td>
      <td>年龄: </td><td><input type="text" name="list[2].age"> </td>
      <td>生日: </td><td><input type="text" name="list[2].birthday"> </td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="保存"> </td>
    </tr>
  </table>
</form>

```

三个隐藏表单域name都一样，值会自动绑定到controller方法的形参数组里面

## 4. SpringMvc 数据校验

### 4.1. 什么是数据校验

这个比较好理解，就是用来验证客户输入的数据是否合法，比如客户登录时，用户名不能为空，或者不能超出指定长度等要求，这就叫做数据校验。

数据校验分为客户端校验和服务端校验




客户端校验: js 校验

服务端校验: springmvc 使用 validation 校验, struts2 使用 validation 校验。都有自己的一套校验规则。

## 4.2. springmvc 的 validation 校验

Springmvc 本身没有校验功能，它使用 hibernate 的校验框架，hibernate 的校验框架和 orm 没有关系

### 4.2.1. 添加 jar

名称	修改日期	类型
 hibernate-validator-4.3.0.Final.jar	2015/3/5 11:05	Executab
 jboss-logging-3.1.0.CR2.jar	2015/3/5 11:43	Executab
 validation-api-1.0.0.GA.jar	2015/3/5 11:05	Executab

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.3.0.Final</version>
</dependency>
```

### 4.2.2. 在 springmvc.xml 中配置 validator 校验器

配置以下这些，相当于有人帮我们写好了校验代码，我们拿过来直接用就行了，所以需要进行配置。

```
38 <!-- 校验器，配置validator -->
39 <bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
40     <property name="providerClass" value="org.hibernate.validator.HibernateValidator"></property>
41     <property name="validationMessageSource" ref="validationMessageSource"></property>
42 </bean>
43
44 <!-- 配置validationMessageSource -->
45 <bean id="validationMessageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
46     <!-- 指定校验信息的资源文件的基本文件名称，不包括后缀，后缀默认是properties -->
47     <property name="basenames">
48         <list>
49             <value>classpath:validationMessageSource</value>
50         </list>
51     </property>
52     <!-- 指定文件的编码 -->
53     <property name="fileEncodings" value="utf8"></property>
54     <!-- 对资源文件内容缓存的时间，单位秒 -->
55     <property name="cacheSeconds" value="120"></property>
56 </bean>
```

### 4.2.3. 将 validator 注册到适配器中

方式一(推荐)

```
18 <!-- 配置处理器映射器和处理器适配器 -->
19 <mvc:annotation-driven conversion-service="conversionService" validator="validator">
20 |
```

方式二：如果配置文件中使用的是非注解方式编写的适配器，则这样配置

```
16 <!-- 自定义WebBinder -->
17 <bean id="customBinder"
18     class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
19     <property name="validator" ref="validator"/>
20 </bean>
21
22 <!-- 注解适配器 -->
23 <bean
24     class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
25     <property name="webBindingInitializer" ref="customBinder"></property>
26 </bean>
27
```

### 4.2.4. 在 pojo 中指定校验规则

列举两个校验规则(使用的是注解校验)，nonnull 和 size



```

8 public class Items {
9     private Integer id;
10     @Size(min=1,max=20,message="{items.name.size}")
11     private String name;
12
13     private Float price;
14
15     private String pic;
16     @NotNull(message="{items.createtime.notNull}")
17     private Date createtime;
18
19     private String detail;
20
21     public Integer getId() {
22         return id;
23     }

```

1、items.name.size 和 items.createtime.notNull:就是读取 validationMessageSource.properties 中的配置信息。从这里就可以理解该配置文件的意义，防止硬编码。

2、使用注解对需要进行校验的属性进行绑定，而能够使这些注解生效的前提就是配置此前的几个步骤，2.1、2.2、2.3 都必不可少

其他校验规则摘抄自网上

验证注解	验证的数据类型	说明
@AssertFalse	Boolean,boolean	验证注解的元素值是false
@AssertTrue	Boolean,boolean	验证注解的元素值是true
@NotNull	任意类型	验证注解的元素值不是null
@Null	任意类型	验证注解的元素值是null
@Min(value=值)	BigDecimal , BigInteger, byte, short, int, long , 等任何Number或CharSequence ( 存储的是数字 ) 子类型	验证注解的元素值大于等于@Min指定的value值
@Max ( value=值 )	和@Min要求一样	验证注解的元素值小于等于@Max指定的value值
@DecimalMin(value=值)	和@Min要求一样	验证注解的元素值大于等于@DecimalMin指定的value值
@DecimalMax(value=值)	和@Min要求一样	验证注解的元素值小于等于@DecimalMax指定的value值
@Digits(integer=整数位数, fraction=小数位数)	和@Min要求一样	验证注解的元素值的整数位数和小数位数上限
@Size(min=下限, max=上限)	字符串、Collection、Map、数组等	验证注解的元素值的在min和max ( 包含 ) 指定区间之内，如字符串长度、集合大小
@Past	java.util.Date, java.util.Calendar; Joda Time类库的日期类型	验证注解的元素值 ( 日期类型 ) 比当前时间早
@Future	与@Past要求一样	验证注解的元素值 ( 日期类型 ) 比当前时间晚
@NotBlank	CharSequence子类型	验证注解的元素值不为空 ( 不为null、去除首位空格后长度为0 ) ，不同于@NotEmpty , @NotBlank只应用于字符串且在比较时会去除字符串的首位空

@Length(min=下限, max=上限)	CharSequence子类型	验证注解的元素值长度在min和max区间内
@NotEmpty	CharSequence子类型、Collection、Map、数组	验证注解的元素值不为null且不为空 ( 字符串长度不为0、集合大小不为0 )
@Range(min=最小值, max=最大值)	BigDecimal,BigInteger,CharSequence, byte, short, int, long等原子类型和包装类型	验证注解的元素值在最小值和最大值之间
@Email(regex=正则表达式, flag=标志的模式)	CharSequence子类型 ( 如String )	验证注解的元素值是Email，也可以通过regex和flag指定自定义的email格式
@Pattern(regex=正则表达式, flag=标志的模式)	String, 任何CharSequence的子类型	验证注解的元素值与指定的正则表达式匹配
@Valid	任何非原子类型	指定递归验证关联的对象；如用户对象中有个地址对象属性，如果想在验证用户对象时一起验证地址对象的话，在地址对象上加@Valid注解即可级联验证

## 4. 2. 5. controller 中对其校验绑定进行使用

```
//@Validated: 对它注解的pojo进行校验
//BindingResult: 通过这个对象可以获取到校验失败的信息, 它和validated注解必须配对使用, 而且一前一后
@RequestMapping("/editItemsSubmit")
public String editItemsSubmit(Integer id, @Validated Items items, BindingResult result) {
    if (result.hasErrors()) {
        List<ObjectError> allErrors = result.getAllErrors();
        for (ObjectError objectError : allErrors) {
            System.out.println(objectError.getDefaultMessage());
        }
        model.addAttribute("errors", allErrors);
    }
    return "items/editItems";
}
```

- 1、@Validated 作用就是将 pojo 内的注解数据校验规则(@NotNull 等)生效, 如果没有该注解的声明, pojo 内有注解数据校验规则也不会生效
- 2、BindingResult 对象用来获取校验失败的信息(@NotNull 中的 message), 与@Validated 注解必须配对使用, 一前一后
- 3、代码中的逻辑应该很容易看懂, 就是将 result 中所有的错误信息取出来, 然后到原先的页面将错误信息进行显示, 注意, 要使用 model 对象, 则需要在形参中声明 Model model, 然后才能使用

## 4. 2. 6. validationMessageSource.properties

该配置文件的作用就是存储校验失败时的提示文字信息的, 也就是相当于将其提取出来放到配置文件中,

name	value
items.name.size	商品名称必须在1到20个字符之间
items.createTime.notNull	商品创建时间不能为空

## 5. 全局异常处理

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。

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

### 5. 1. 自定义异常类

```
public class MyException extends Exception{
    private String msg;

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public MyException() {
        this.msg = msg;
    }
}
```

```

    }

    public MyException(String msg) {
        super(msg);
        this.msg = msg;
    }
}

```

## 5.2. 自定义异常处理器

```

public class ExceptionHandler implements HandlerExceptionResolver{

    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse,
Object o, Exception e) {

        //1、判断是哪一种异常

        String msg="";

        if(e instanceof MyException){
            MyException myException=(MyException)e;
            msg=myException.getMsg();
        }

        //2、发送邮件和短信通知到相关人员

        //3、跳转到友好的页面，并展示描述信息

        ModelAndView modelAndView=new ModelAndView();

        modelAndView.addObject("error",msg);

        modelAndView.setViewName("/500");

        return modelAndView;
    }
}

```

## 5.3. 在 springmvc.xml 里面配置

```

<!-- 配置异常处理器-->
<bean class="com.hy.springmvc.exception.ExceptionHandler"></bean>

```

## 6. 拦截器

拦截器其实就是我们的 filter

### 6.1. 自定义拦截器

```

public class LoginInterceptor implements HandlerInterceptor {

    // controler 执行后且视图返回后调用此方法

```

```

// 这里可得到执行 controller 时的异常信息
// 这里可记录操作日志
@Override
public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
Exception arg3)
    throws Exception {
    System.out.println("HandlerInterceptor1...afterCompletion");
}

// controller 执行后但未返回视图前调用此方法
// 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(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2,
ModelAndView arg3)
    throws Exception {
    System.out.println("HandlerInterceptor1...postHandle");
}

// Controller 执行前调用此方法
// 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(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2) throws
Exception {
    System.out.println("HandlerInterceptor1...preHandle");
    // 设置为 true，测试使用
    return true;
}
}

```

## 6.2. 配置拦截器

```

<mvc:interceptors>
    <mvc:interceptor>
        <!-- 表示所有路径包括子路径 -->
        <mvc:mapping path="/**"/>
        <bean class="com.lx.ssm.interceptor.LoginInterceptor"></bean>
    </mvc:interceptor>
    <!-- 如果有多个拦截器继续配置 -->
    <mvc:interceptor>
        <!-- 表示所有路径包括子路径 -->
        <mvc:mapping path="/**"/>
        <bean class="com.lx.ssm.interceptor.AuthorityInterceptor"></bean>
    </mvc:interceptor>
</mvc:interceptors>

```

## 6.3. 代码示

- 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。

例



AuthorityInterceptor.java



```
package com.lx.ssm.interceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

/**
 * 登录认证拦截器
 * @author wangsq
 * @date 2017 年 10 月 19 日下午 2:40:09
 * version V1.0
 */
public class LoginInterceptor implements HandlerInterceptor{

    /**
     * controller 方法执行完之后执行
     * 可以做全局异常信息处理
     */
    @Override
    public void afterCompletion(HttpServletRequest arg0,
                               HttpServletResponse arg1, Object arg2, Exception arg3)
                               throws Exception {
        System.out.println("afterCompletion");
    }

    /**
     * controller 方法执行执行完且返回 ModelAndView 之前执行
     */
    @Override
    public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1,
                           Object arg2, ModelAndView arg3) throws Exception {
        System.out.println("postHandle");
    }
}
```

```

    }

    /**
     * 此方法是在 controller 方法执行之前执行
     * return false 不放行 return true 放行
     * 1、登录认证
     * 2、权限鉴权
     */

    @Override

    public boolean preHandle(HttpServletRequest req, HttpServletResponse resp,

                               Object obj) throws Exception {

        System.out.println(obj);

        String username=(String)req.getSession().getAttribute("username");

        if(req.getRequestURI().indexOf("login/login.action")!=1){

            return true;

        }

        if(username==null || "".equals(username)){

            req.getRequestDispatcher("/login.jsp").forward(req, resp);

            return false;

        }

        return true;

    }

}

```

## 7. Springmvc 静态文件访问处理

SpringMVC 提供<mvc:resources>来设置静态资源，但是增加该设置如果采用通配符的方式增加拦截器的话仍然会被拦截器拦截，可采用如下方案进行解决：

### 7.1. 拦截器中增加针对静态资源不进行过滤

```

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

<mvc:resources location="/" mapping="/**/*.*.css"/>

<mvc:resources location="/assets/" mapping="/assets/**/*.*"/>

<mvc:resources location="/images/" mapping="/images/*.*" cache-period="360000"/>

<mvc:interceptors>

    <mvc:interceptor>

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

        <mvc:exclude-mapping path="/**/fonts/*.*"/>

        <mvc:exclude-mapping path="/**/*.*.css"/>

        <mvc:exclude-mapping path="/**/*.*.js"/>

        <mvc:exclude-mapping path="/**/*.*.png"/>

        <mvc:exclude-mapping path="/**/*.*.gif"/>
    
```



```

<mvc:exclude-mapping path="/**/*.*jpg"/>

<mvc:exclude-mapping path="/**/*.*jpeg"/>

<mvc:exclude-mapping path="/**/*.*login*"/>

<mvc:exclude-mapping path="/**/*.*Login*"/>

<bean class="com.luwei.console.mg.interceptor.VisitInterceptor"></bean>

</mvc:interceptor>

</mvc:interceptors>

```

## 7.2. 使用默认的静态资源处理 Servlet 处理静态资源(涉及 *spring-mvc.xml*, *web.xml*)

在 *spring-mvc.xml* 中启用默认 Servlet

```
1 <mvc:default-servlet-handler/>
```

在 *web.xml* 中增加对静态资源的处理

```

<servlet-mapping>

    <servlet-name>default</servlet-name>

    <url-pattern>*.js</url-pattern>

    <url-pattern>*.css</url-pattern>

    <url-pattern>/assets/*</url-pattern>

    <url-pattern>/images/*</url-pattern>

</servlet-mapping>

```

但是当前的设置必须在 **Spring** 的 **Dispatcher** 的前面

## 7.3. 修改 *Spring* 的全局拦截设置为\*.do 的拦截

```

<servlet>

    <servlet-name>SpringMVC</servlet-name>

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

    <init-param>

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

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

    </init-param>

    <load-on-startup>1</load-on-startup>

    <async-supported>true</async-supported>

</servlet>

<servlet-mapping>

    <servlet-name>SpringMVC</servlet-name>

    <url-pattern>*.do</url-pattern>

</servlet-mapping>

```

这样设置，**Spring** 就会只针对以'.do'结尾的请求进行处理，不再维护静态资源

## 7.4. 总结

针对这三种方案的优劣分析：

第一种方案配置比较臃肿，多个拦截器时增加文件行数，不推荐使用；第二种方案使用默认的 **Servlet** 进行资源文件的访问，**Spring** 拦截所有请求，然后再将资源文件交由默认的 **Servlet** 进行处理，性能上少有损耗；第三种方案 **Spring** 只是处理以'.do'结尾的访问，性能上更加高效，但是再访问路径上必须都以'.do'结尾，URL 不太文雅；

综上所述，推荐使用第二和第三种方案

## 8. restful 支持

### 8.1. 资源操作

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
```

### 8.2. 从 url 上获取参数

```
@RequestMapping("item/{id}")
@ResponseBody
public Item queryItemById(@PathVariable() Integer id) {}
```

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

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

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")

注意两个区别

1. @PathVariable 是获取 url 路径上数据的。@RequestParam 获取静态的 url 请求参数的（包括 post 表单提交）

1. @Controller

2. @RequestMapping("/owners/{ownerId}")

3. public class RelativePathUriTemplateController {

4.

5. @RequestMapping("/pets/{petId}")

```
6. public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {  
7. // implementation omitted  
8. }  
9. }
```

上面代码把 URI template 中变量 `ownerId` 的值和 `petId` 的值，绑定到方法的参数上。若方法参数名称和需要绑定的 uri template 中变量名称不一致，需要在 `@PathVariable("name")` 指定 uri template 中的名称

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

## 8.3. 最佳实践

### 8.3.1. 在 web.xml 配置过滤器

```
<!-- 将 POST 请求转化为 DELETE 或者是 PUT 要用 _method 指定真正的请求参数 -->  
  
<filter>  
    <filter-name>HiddenHttpMethodFilter</filter-name>  
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>  
</filter>  
  
<filter-mapping>  
    <filter-name>HiddenHttpMethodFilter</filter-name>  
    <url-pattern>/*</url-pattern>  
</filter-mapping>
```

### 8.3.2. 表单中发送 post 请求

method="post"

添加隐藏域(name="\_method" value="PUT/DELETE")

```
<input type="hidden" name="_method" value="DELETE" />
```

### 8.3.3. 配置支持对应的请求方式

```
@RequestMapping(method = RequestMethod.DELETE)
```

## 9. Springmvc 整合 mybatis

### 9.1. 整合思路

第一步：整合 dao 层

Mybatis 和 spring 整合，通过 spring 管理 mapper 接口。

使用 mapper 的扫描器自动扫描 mapper 接口在 spring 中进行注册。

第二步：整合 service 层

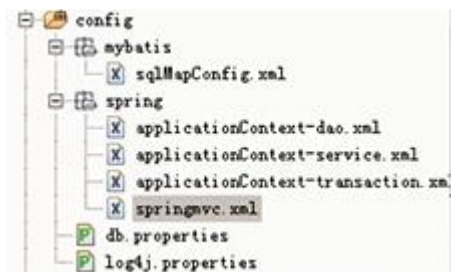
通过 spring 管理 service 接口。

使用配置方式将 service 接口配置在 spring 配置文件中。

实现事物控制

第三步：整合 springmvc

由于 springmvc 是 spring 的一个模块，不需要整合



### 9.2. 整合 dao

#### 9.2.1. SqlMapConfig.xml

```
<configuration>
  <!-- 启用log4j -->
  <settings>
    <setting name="logImpl" value="LOG4J"/>
  </settings>
  <!-- 批量别名 -->
  <typeAliases>
    <package name="com.lx.ssm.po"/>
  </typeAliases>
</configuration>
```

#### 9.2.2. ApplicationContext-dao.xml

```
<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  <property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3306/mybatis0203"/>
  <property name="user" value="root"/>
  <property name="password" value="123456"/>
</bean>
```

```

<!-- 创建sqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <!-- 加载mybatis核心配置文件-->
    <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml"></property>
</bean>

<!-- mapper批量扫描，从mapper包中扫描出mapper接口，自动创建代理对象并且在spring容器中注册
遵循规范：mapper.java和mapper.xml映射文件名字保持一致，且在一个目录中，
自动扫描出来的mapper的bean的id为mapper类名（首字母小写）
-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

    <!-- 指定扫描的包名
    如果扫描多个包，每个包中间使用半角逗号分隔 -->
    <property name="basePackage" value="com.lx.ssm.mapper"></property>
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>

```

### 9.2.3. Mapper.xml

### 9.2.4. 整合 service

#### 9.2.4.1. 定义 service 接口

#### 9.2.4.2. 在 Spring 容器中配置 service

```

<bean id="userService" class="com.lx.ssm.service.UserServiceImpl">
    <property name="userMapper" ref="userMapper"></property>
</bean>

```

#### 9.2.4.3. 事物控制

- <bean id="transactionManager"
- class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
- <prop
- ransaction-manager="transactionManager" />

```

<!-- 配置事物管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 开启事物注解 -->
<tx:annotation-driven transaction-manager="transactionManager"/>

```

## 9.2.5. 整合 springmvc

### 9.2.5.1. Springmvc.xml

```

<mvc:annotation-driven></mvc:annotation-driven>

<context:component-scan base-package="com.lx.ssm.controller"></context:component-scan>

<!--
    视图解析器 解析jsp 默认使用jstl
    prefix:在视图逻辑名前面加上一部分,
    suffix:在视图逻辑名后面加上一部分;
-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/" />
    <property name="suffix" value=".jsp" />
</bean>

```

### 9.2.5.2. 加载 spring 容器

Spring 文件很多，使用通配符方式加载。

```

<!-- 加载spring容器 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/spring/applicationContext-*.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<filter>
    <filter-name>encoding</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```



```

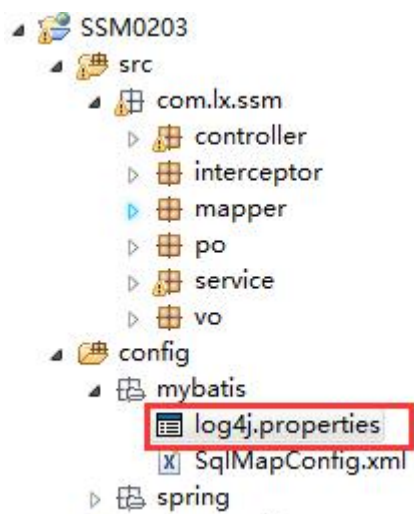
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param><!-- 配置文件主要配置处理器映射器、适配器等 -->
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring/springmvc.xml</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <!--
    url-pattern:三种配置方式
    1、*.action 访问以.action结尾的请求由DispatcherServlet来解析
    2、/ 所有请求包括静态资源（js、css、image等等）都有DispatcherServlet解析，对于静态文件的解析需
    3、/*这种配置方式不对
  -->
  <url-pattern>*.action</url-pattern>
</servlet-mapping>

```

## 9.2.6. 加载 log4j 配置文件

### 9.2.6.1. 配置 log4j 配置文件



```
#定义日志输出目的地为控制台
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.Target=System.out
#可以灵活的指定日志输出格式，下面一行是指定具体的格式
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=[%c]-%m%n

#mybatis显示SQL语句日志配置
#log4j.logger.org.mybatis=DEBUG
log4j.logger.com.lx.ssm.mapper=DEBUG

#文件大小到达指定尺寸的时候产生一个新的文件
log4j.appender.File=org.apache.log4j.RollingFileAppender
#指定输出目录
log4j.appender.File.File=G:/logs/ssm.log
#定义文件最大大小
log4j.appender.File.MaxFileSize=10MB
#输出所有日志，如果换成DEBUG表示输出DEBUG以上级别日志
log4j.appender.File.Threshold=ALL
log4j.appender.File.layout=org.apache.log4j.PatternLayout
log4j.appender.File.layout.ConversionPattern=[%p][%d{yyyy-MM-dd HH:mm:ss}][%c]%m%n

#定义LOG输出级别

log4j.rootLogger=INFO,Console,File

#定义日志输出目的地为控制台

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.Target=System.out

#可以灵活的指定日志输出格式，下面一行是指定具体的格式

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern=[%c]-%m%n

#mybatis显示SQL语句日志配置

#log4j.logger.org.mybatis=DEBUG

log4j.logger.com.lx.ssm.mapper=DEBUG

#文件大小到达指定尺寸的时候产生一个新的文件

log4j.appender.File=org.apache.log4j.RollingFileAppender

#指定输出目录

log4j.appender.File.File=G:/logs/ssm.log

#定义文件最大大小

log4j.appender.File.MaxFileSize=10MB

#输出所有日志，如果换成DEBUG表示输出DEBUG以上级别日志

log4j.appender.File.Threshold=ALL

log4j.appender.File.layout=org.apache.log4j.PatternLayout

log4j.appender.File.layout.ConversionPattern=[%p][%d{yyyy-MM-dd HH:mm:ss}][%c]%m%n
```

### 9.2.6.2. 在 web.xml 里面配置监听器

```
<!-- 加载log4j配置文件 -->
<context-param>
  <param-name>log4jConfigLocation</param-name>
  <param-value>classpath:mybatis/log4j.properties</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
</listener>
```

### 9.2.6.3. 让 mybatis 使用 log4j 作为日志输出

配置 sqlmapconfig.xml

```
<!-- 启用log4j -->
<settings>
  <setting name="logImpl" value="LOG4J"/>
</settings>
```

## 10. Springmvc 实现文件上传

我们使用 apache fileupload 上传文件，springmvc 集成此组件

### 10.1. 添加坐标

```
<dependency>
  <groupId>commons-fileupload</groupId>
  <artifactId>commons-fileupload</artifactId>
  <version>1.3.1</version>
</dependency>
```

### 10.2. 在 spring\_mvc.xml 配置上传的组件

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <!-- set the max upload size100MB -->
  <property name="maxUploadSize">
    <value>104857600</value>
  </property>
  <property name="maxInMemorySize">
    <value>4096</value>
  </property>
</bean>
```

### 10.3. 在 controller 方法中绑定文件流参数

```
@RequestMapping("/fileuploadExecl")

    public void fileuploadExecl(@RequestParam("filename") MultipartFile pictureFile){

        try {

            // 图片上传

            // 设置图片名称, 不能重复, 可以使用 uuid

            String picName = UUID.randomUUID().toString();

            // 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));

        } catch (IOException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

    }

}
```

### 10.4. jsp 页面

```
<form action="user/fileuploadExecl" enctype="multipart/form-data" method="post">

    <input type="file" name="filename">

    <input type="submit">

</form>
```

总结:

通过以上两个步骤就可以在 controller 方法中得到输入的文件流。

### 10.5. 表单里面有文件上传同时又有普通属性

## 11. Springmvc 与 Struts2 的不同

springmvc 与 struts2 不同

- 1、 springmvc 的入口是一个 servlet 即前端控制器, 而 struts2 入口是一个 filter 过滤器。
- 2、 springmvc 是基于方法开发(一个 url 对应一个方法), 请求参数传递到方法的形参, 可以设计为单例或多例(建议单例), struts2 是基于类开发, 传递参数是通过类的属性, 只能设计为多例。

3、 Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据， springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 request 域传输到页面。Jsp 视图解析器默认使用 jstl。