# 概述

Spring MVC框架是一个开源的Java平台，为开发强大的基于JavaWeb应用程序提供全面的基础架构支持，并且使用起来非常简单容易。

SpringMVC框架提供了MVC(模型-视图-控制器)架构，用于开发灵活和松散耦合的Web应用程序的组件。 MVC模式使应用程序的不同组件(输入逻辑，业务逻辑和UI逻辑)合理有效的分离，同时又有效的将各组件组合一起完成功能。

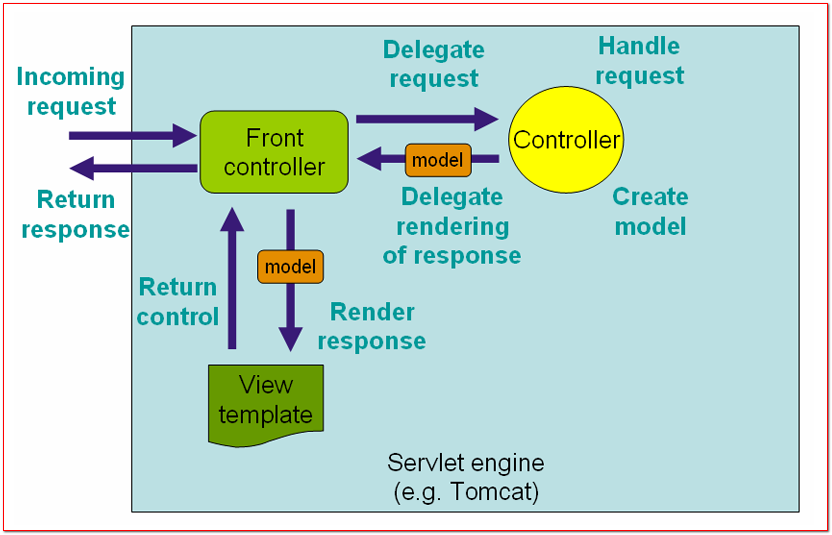
模型(**Model**) 封装了应用程序数据，通常它们将由JavaBean类组成。

视图(**View**) 负责渲染模型数据，一般来说它负责生成客户端浏览器可以解释HTML输出。

控制器(**Controller**) 负责处理用户请求并构建适当的模型，并将其传递给视图进行渲染。

## 核心执行步骤

DispacherServlet程序是SpringMVC中负责请求分发的一个核心程序，也叫前端控制器（Front Controller）。



## SpringMVC的工作流程

user

DispatcherServlet

Controller

Handler Mapping

Handler Adapter

View Resolver

1、request

2、request

10、response

9、model，view

3、controller

4、controller

7、ModelAndView

8、ModelAndView

5、method

6、ModelAndView

1、用户向服务器发送请求，请求被SpringMVC的前端控制器DispatcherServlet截获。

2、DispatcherServlet对请求的URL（统一资源定位符）进行解析，得到URI(请求资源标识符)，然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象，包括Handler对象以及Handler对象对应的拦截器，这些对象都会被封装到一个HandlerExecutionChain对象当中返回。

3、DispatcherServlet根据获得的Handler，选择一个合适的HandlerAdapter。HandlerAdapter的设计符合面向对象中的单一职责原则，代码结构清晰，便于维护，最为重要的是，代码的可复制性高。HandlerAdapter会被用于处理多种Handler，调用Handler实际处理请求的方法。

4、提取请求中的模型数据，开始执行Handler(Controller)。在填充Handler的入参过程中，根据配置，spring将帮助做一些额外的工作

消息转换：将请求的消息，如json、xml等数据转换成一个对象，将对象转换为指定的响应信息。

数据转换：对请求消息进行数据转换，如String转换成Integer、Double等。

数据格式化：对请求的消息进行数据格式化，如将字符串转换为格式化数字或格式化日期等。

数据验证：验证数据的有效性如长度、格式等，验证结果存储到BindingResult或Error中。

5、Handler执行完成后，向DispatcherServlet返回一个ModelAndView对象，ModelAndView对象中应该包含视图名或视图模型。

6、根据返回的ModelAndView对象，选择一个合适的ViewResolver(视图解析器)返回给DispatcherServlet。

7、ViewResolver结合Model和View来渲染视图。

8、将视图渲染结果返回给客户端。

以上8个步骤，DispatcherServlet、HandlerMapping、HandlerAdapter和ViewResolver等对象协同工作，完成SpringMVC请求—>响应的整个工作流程，这些对象完成的工作对于开发者来说都是不可见的，开发者并不需要关心这些对象是如何工作的，开发者，只需要在Handler(Controller)当中完成对请求的业务处理。

## 简单实现

### 源码目录配置

创建web工程，在源码目录config中配置Springmvc.xml配置文件，然后在web.xml配置文件中配置前端控制器。

全部配置完成后可以创建测试用的Controller类和jsp页面测试。

**Springmvc.xml**

|  |
| --- |
| <?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:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  <!--配置包扫描器-->  <context:component-scan base-package="com"/> </beans> |

**web.xml**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0">  <!—配置前端控制器 -->  <servlet>  <servlet-name>springDispatcherServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!--contextConfigLocation初始化参数，用来配置SpringMVC配置文件路径-->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <!--load-on-startup标签配置web工程启动时创建的servlet程序实例-->  <load-on-startup>1</load-on-startup>  </servlet>  <!-- 配置的前端控制器路径和Tomcat服务器系统中的DefaultServlet程序路径冲突  <servlet-name>default</servlet-name>  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>  覆盖了原来的DefaultServlet程序，该程序专门用于处理静态资源 -->  <servlet-mapping>  <servlet-name>springDispatcherServlet</servlet-name>  <!--此处设置为“/”表示将所有请求都转交给SpringMVC框架处理-->  <url-pattern>/</url-pattern>  </servlet-mapping> </web-app> |

#### SpringMVC处理静态资源

因为在web.xml中配置DispatcherServlet前端控制器时使用的地址是/，而这个斜杠刚好和Tomcat服务器中的DefaultServlet程序（用于处理静态资源）使用的地址相同，直接覆盖了原来Tomcat中DefaultServlet程序的静态资源处理功能，从而导致访问静态资源时出现404错误。

因此需要在springmvc.xml中配置SpringMVC对静态资源的处理和对注解功能的支持，这两个配置只要使用SpringMVC开发就要对其进行配置。

**方法一（推荐）**

|  |
| --- |
| <!-- 以下两个标签是SpringMVC开发的标配 --> <!-- 配置SpringMVC对静态资源的处理支持 --> <mvc:default-servlet-handler/> <!-- 配置SpringMVC的高级功能（即支持静态资源，又支持springMVC的高级功能） --> <mvc:annotation-driven/> |

**方法二**

可以修改SpringMVC前端控制器的请求地址。让SpringMVC的拦截地址和Tomcat中的处理器拦截地址不相同。这个当有静态资源请求进来的时候Tomcat原来的Servlet程序还可以继续处理静态资源。

SpringMVC前端控制器的拦截地址可以改成为：\*.do | \*.action | \*.xhtml | \*.mvc

|  |
| --- |
| <servlet>  <servlet-name>springDispatcherServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springDispatcherServlet</servlet-name>  <url-pattern>\*.action</url-pattern>  </servlet-mapping> |

这种方式的缺点是会导致SpringMVC不能支持Restful风格的请求。

**方法三**

激活tomcat默认的资源处理器，需要在web.xml种进行如下配置：

|  |
| --- |
| <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.jpg</url-pattern> </servlet-mapping> <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.js</url-pattern> </servlet-mapping> <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.css</url-pattern> </servlet-mapping> <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.html</url-pattern> </servlet-mapping> |

这种方式的缺点是需要对web工程种使用的所有静态资源后缀都进行配置，过于繁琐。

**方法四**

使用标签的方式指定将静态资源交给tomcat服务器来处理。

|  |
| --- |
| <!-- 表示启用SpringMVC的注解映射访问 --> <mvc:annotation-driven></mvc:annotation-driven> <!-- 表示resources标签所指定的资源都交给Tomcat服务器来处理  location指的是工程下的哪个目录  mapping 指的是这个目录下的哪些资源 --> <mvc:resources location="/static/" mapping="/static/\*\*"/> |

**Controller测试类**

|  |
| --- |
| @Controller public class HelloController {  //hello方法处理用户的请求，每一个方法处理一个请求  //@RequestMapping 注解表示给当前方法配置一个访问地址  //"/hello" 表示http://ip:port/工程名/hello  @RequestMapping("/hello")  public String hello(){  //返回值表示转发到http://ip:port/工程名/ok.jsp 路径  return "/ok.jsp";  } } |

**index.jsp**

|  |
| --- |
| <body> <a href="http://localhost:8080/springMVC\_hello/hello">hello请求</a> </body> |

**ok.jsp**

|  |
| --- |
| <body> 请求完成！！！ </body> |

**log4j.properties**

|  |
| --- |
| # Global logging configuration log4j.rootLogger=INFO, stdout # Console output... log4j.appender.stdout=org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout=org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

### WEB-INF目录配置

SpringMVC配置文件也可以放置在WEB-INF目录下，此时需要遵循一个配置文件的命名规则，即将该配置文件的文件名修改为web.xml中**servlet-name标签的值+servlet**。

例如此前servlet-name标签的值为<servlet-name>springDispatcherServlet</servlet-name>，因此该配置文件的文件名应修改为springDispatcherServlet-servlet。

使用这种配置方式时必须将web.xml中配置的DispatcherServlet程序中的init-param标签删除掉才有效，否则SpringMVC会按照init-param中的值来配置SpringMVC配置文件。

### 视图解析器

在SpringMVC配置文件中可以配置视图解析器，可以简化代码的编写。视图解析器可以将Controller类中的方法返回的地址进行简化，避免每次都要写资源后缀名和目录。

|  |
| --- |
| <!--配置视图解析器--> <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <!--配置视图名前缀-->  <property name="prefix" value="/pages/"/>  <!--配置视图名后缀-->  <property name="suffix" value=".jsp"/> </bean> |

完成后Controller类中的方法返回值可以简化书写，如下所示：

|  |
| --- |
| @RequestMapping("/hello") public String hello(){  //因为配置了视图解析器，因此会自动为返回值添加上前缀和后缀，字符串“ok”即是视图  return "ok"; } |

# @RequestMapping注解

@RequestMapping注解可以给标注的方法配置一个访问地址，通过该地址就可以调用该方法。就如同web中的servlet程序，在web.xml中配置了访问地址后，就和该servlet之间产生了一个访问映射。

## 使用方式

**标注在方法上**

@RequestMapping注解标注在方法上表示给该方法配置一个访问地址。

**标注在Controller类上**

@RequestMapping注解标注在类上表示给该Controller类配置一个模块的路径，例如设置模块路径为“/user”。

有模块路径时请求地址：http://ip:port/工程名/user/delete

无模块路径时请求地址：http://ip:port/工程名/delete

|  |
| --- |
| @RequestMapping(value = "/user") @Controller public class HelloController {  @RequestMapping(value = "/delete")  public String hello(){  return "ok";  } } |

## 属性

### value和path属性

@RequestMapping注解的value属性用于配置访问地址，当只有一个属性时可以省略value属性而直接写该属性的值。path属性的作用和value属性的作用相同，都是用于配置访问地址，且配置的访问地址书写方式必须要以“/”开头。

|  |
| --- |
| @RequestMapping(value = "/hello") public String hello(){  return "ok"; }  @RequestMapping(path = "/hello") public String hello(){  return "ok"; } |

### params属性

@RequestMapping注解的params属性表示对请求参数的匹配。

params = "username"表示请求地址中必须带有参数username

params = "username=abc"表示请求地址中必须带有参数username且值必须为abc

params = "username!=abc"表示请求地址中不能有参数username或有参数username但其值不能为abc

params = "!username"表示请求地址中不能有参数username

params属性可以有多个值，每个值之间是&&的关系，多个值之间用逗号隔开，如下所示：

params = {"username!=abc","!password"}表示请求地址中不能有username参数或有参数username但其值不能为abc并且 请求地址中不能有参数password。

|  |
| --- |
| @RequestMapping(params = "username",path = "/params") public String hello(){  return "ok"; } |

### headers属性

@RequestMapping注解的headers属性表示对请求头的匹配，其中请求头的属性和其值之间使用“=”连接，不要使用“:”进行连接。通过设置特殊的请求头，可以达到简单加密的效果，避免恶意请求访问。

|  |
| --- |
| @RequestMapping(headers="User-Agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36",path = "/headers") public String hello(){  return "ok"; } |

### method属性

@RequestMapping注解的method属性表示对请求方式的匹配，其值有RequestMethod.GET| RequestMethod.POST等，表示分别以GET或POST请求访问时才允许被访问。

共有如下几种请求方式：GET、POST、PUT、DELETE、HEAD、PATCH、OPTIONS、TRACE。

|  |
| --- |
| @RequestMapping(value = "/method",method = RequestMethod.*GET*) public String hello(){  System.*out*.println("------hello world------");  return "ok"; }  @RequestMapping(value = "/method",method = RequestMethod.*POST*) public String hello(){  System.*out*.println("------hello world------");  return "ok"; } |

## 通配符

开发中经常不是一个请求地址对应一个处理请求的方法，而是多个请求地址对应一个方法，此时就可以使用通配符来解决这个问题。

当一个路径匹配多个规则时，调用方法的优先顺序的原则是越精确越优先，即绝对匹配优先于？匹配，？匹配优先于\*匹配。

### 绝对匹配

绝对匹配也叫做精确匹配，表示的是一个请求地址对应一个处理请求方法的关系。

|  |
| --- |
| //绝对匹配表示请求地址必须是http://ip:port/工程名/path1 @RequestMapping("/path1") public String path1(){  return "ok"; } |

### 资源路径匹配

URL地址的最后一个“/”后的内容表示资源路径。

**一个字符匹配**

进行资源路径匹配时，“？”表示任意一个字符，比如匹配path[0-9a-zA-Z]

|  |
| --- |
| //此时value属性的值为/path?，表示path后只能有一个任意字符，匹配规则如下： //http://ip:port/工程名/path1 匹配 //http://ip:port/工程名/path2 匹配 //http://ip:port/工程名/path12 不匹配 @RequestMapping("/path?") public String path2(){  return "ok"; } |

任意字符匹配

进行资源路径匹配时，“\*”表示多个任意字符

|  |
| --- |
| //此时value属性的值为/path\*，表示path后面可以有多个任意字符，匹配规则如下： //http://ip:port/工程名/patha 匹配 //http://ip:port/工程名/pathab 匹配 //http://ip:port/工程名/pathabc 匹配 @RequestMapping("/path\*") public String path3(){  System.*out*.println("path3方法被调用");  return "ok"; } |

### 目录路径匹配

URL地址中http://ip:port/工程名/ 和 资源路径之间的内容称之为目录路径。

一层一个字符目录

|  |
| --- |
| //此时value属性的值为/?/path，匹配规则如下： //http://ip:port/工程名/a/path 匹配 //http://ip:port/工程名/b/path 匹配 //http://ip:port/工程名/cd/path 不匹配 @RequestMapping("/?/path") public String path4(){  return "ok"; } |

一层任意字符目录

|  |
| --- |
| //此时value属性的值为/\*/path，匹配规则如下： //http://ip:port/工程名/ads/path 匹配 //http://ip:port/工程名/123d/path 匹配 //http://ip:port/工程名/asd1/path 匹配 @RequestMapping("/\*/path") public String path5(){  return "ok"; } |

多层任意字符目录

|  |
| --- |
| //此时value属性的值为/\*\*/path，匹配规则如下： //http://ip:port/工程名/ads/af/path 匹配 //http://ip:port/工程名/123/df/path 匹配 //http://ip:port/工程名/afg/123/path 匹配 @RequestMapping("/\*\*/path") public String path6(){  return "ok"; } |

# 获取请求信息

## 获取请求参数信息

### 原生API参数类型

**HttpServletResquest类**

要获取原生的HttpServletRequest类对象，只需要在请求的方法上写上参数即可。

|  |
| --- |
| @RequestMapping("/param1") public String param1(HttpServletRequest request) {  System.*out*.println("param1被调用了 --- " + request);  System.*out*.println("请求头：" + request.getHeader("User-Agent"));  System.*out*.println("请求方式：" + request.getMethod());  System.*out*.println("请求参数username的值：" + request.getParameter("username"));  return "ok"; } |

**HttpSession类**

要获取原生的HttpSession类对象，只需要在请求的方法上写上参数即可。

|  |
| --- |
| @RequestMapping("/param2") public String param2(HttpSession session) {  System.*out*.println("param2被调用了 --- " + session);  System.*out*.println("当前Session是否是新创建的：" + session.isNew());  System.*out*.println("当前Session的id值是：" + session.getId());  return "ok"; } |

**HttpservletResponse类**

要获取原生的HttpServletResponse类对象，只需要在请求的方法上写上参数即可。

|  |
| --- |
| @RequestMapping("/param3") public String param3(HttpServletResponse response) {  System.*out*.println("param3被调用了 --- " + response);  return "ok"; } |

如果想要同时获得HttpServletRequest、HttpSession和HttpServletResponse类的对象只需要在方法的参数上一起写上即可获得。

### 普通参数类型

只需要在方法上输入需要的参数即可，要求方法中的参数名必须和请求过来的参数名对应，如果没有找到对应的参数，则使用该类型参数的默认值。

|  |
| --- |
| @RequestMapping("/param5") public String param5(String username, String password, Integer age) {  System.*out*.println("param5方法： username参数的值是：" + username);  System.*out*.println("param5方法： password参数的值是：" + password);  System.*out*.println("param5方法： age参数的值是：" + age);  return "ok"; } |

上述请求地址为：http://localhost:8080/工程名/param5?username=xmm&password=123，其获取到的结果是username的值为xmm，password的值为123，而age的值为null。

### 普通数组参数类型

如果要获取数组类型的普通参数，在方法中写入的参数类型定义为该类型的数组即可。

|  |
| --- |
| @RequestMapping("/param6") public String param6(String username, String[] hobby) {  System.*out*.println("param6方法： username参数的值是：" + username);  System.*out*.println("param6方法： hobby参数的值是：" + Arrays.*asList*(hobby));  return "ok"; } |

### @RequestParam注解

@RequestParam注解的作用是将客户端传递过来的参数注入到指定的方法参数中。

value表示客户端发送过来的参数名

defaultValue表示默认值，如果客户端没有发送过来参数值，就使用默认值

required默认值为true，表示标注的参数不能为null，如果为null就会报错

|  |
| --- |
| @RequestMapping("/param7") public String param7(@RequestParam(value = "user", required = false) String username) {  System.*out*.println("param7方法： username参数的值是：" + username);  return "ok"; } |

## 获取请求头信息

@RequestHeader注解的作用是将请求头中指定key的值注入到方法的参数中。

value表示客户端发送过来的请求头的key值

defaultValue表示默认值，如果客户端没有发送过来参数值，就使用默认值

required默认值为true，表示标注的参数不能为null，如果为null就会报错

|  |
| --- |
| @RequestMapping("/param8") public String param8(@RequestHeader("User-Agent") String userAgent, @RequestHeader("host") String host) {  System.*out*.println("param8方法 userAgent请求头的值是：" + userAgent);  System.*out*.println("param8方法 host请求头的值是：" + host);  return "ok"; } |

## 获取Cookie信息

@CookieValue注解的作用是将指定的cookie的值注入到方法中。

|  |
| --- |
| //将key是JSESSIONID的cookie值，注入到方法参数jsessionId中 @RequestMapping("/param9") public String param9(@CookieValue(value = "JSESSIONID", defaultValue = "null") String jsessionId) {  System.*out*.println("param9方法 jsessionId=" + jsessionId);  return "ok"; } |

## 获取JavaBean信息

在Controller控制器的方法中添加参数是JavaBean时，要求客户端发送过来的参数名必须和JavaBean的属性名保持一致。

|  |
| --- |
| public class Person {  private Integer id;  private String name;  private String phone;  private Integer age;  } |

|  |
| --- |
| <form action="http://localhost:8080/spring\_mvc\_hello/param10">  <label for="id">编号：<input id="id" name="id" /></label><br/>  <label for="name">姓名：<input id="name" name="name" /></label><br/>  <label for="phone">电话：<input id="phone" name="phone" /></label><br/>  <label for="age">年龄：<input id="age" name="age" /></label><br/>  <input type="submit" /> </form> |

|  |
| --- |
| @RequestMapping("/param10") public String param10(Person person) {  System.*out*.println("param10方法:person的值是：" + person);  return "ok"; } |

### 获取级联对象信息

创建Book的JavaBean类，并在Person类中添加Book属性。

|  |
| --- |
| public class Book {  private String name;  private BigDecimal price;  } |

|  |
| --- |
| <%--级联属性的获取，通过“.”来获取对象属性的属性--%> <label for="book.name">书名：<input id="book.name" name="book.name" /></label><br/> <label for="book.price">价格：<input id="book.price" name="book.price" /></label><br/> |

|  |
| --- |
| @RequestMapping("/param10") public String param10(Person person) {  System.*out*.println("param10方法:person的值是：" + person);  return "ok"; } |

# 方法返回值

被@RequestMapping注解标注的方法根据其返回值的不同会有不同的功能。

## 返回String

返回值为String类型时，只有默认情况下如果有视图解析器就会进行请求转发地址的拼接操作，显示转发和显示重定向都会忽略视图解析器。

### 默认情况

默认情况下根据有没有视图解析器分为两种情况，有视图解析器时，返回值是前缀+视图名（return后面返回的字符串）+后缀拼接得到的请求转发地址，没有视图解析器时则没有转发地址的拼接操作。最后SpringMVC底层使用请求转发进行跳转。

**有视图解析器时**

|  |
| --- |
| @RequestMapping("/return0") public String return0() {  return "ok"; } |

**无视图解析器时**

|  |
| --- |
| @RequestMapping("/return0") public String return0() {  return "/pages/ok.jsp"; } |

### 显示转发

进行显示转发只需要在返回值的字符串（即路径地址）最前面添加forward:即可，显示转发时，SpringMVC会忽略掉视图解析器。

|  |
| --- |
| @RequestMapping("/return1") public String return1() {  return "forward:/pages/ok.jsp"; } |

### 显示重定向

进行显示重定向只需要在返回值的字符串（即路径地址）最前面添加redirect:即可，显示转发时，SpringMVC会忽略掉视图解析器。

|  |
| --- |
| @RequestMapping("/return2") public String return2() {  return "redirect:/pages/ok.jsp"; } |

进行显示转发和显示重定向时都会忽略视图解析器InternalResourceViewResolver。

## 返回ModelAndView

返回值为ModelAndView类型时，只有默认情况下如果有视图解析器就会进行请求转发地址的拼接操作，显示转发和显示重定向都会忽略视图解析器。

### 默认情况

返回值为ModelAndView时可以有两种方式返回带有路径地址的ModelAndView对象，第一种可以在创建ModelAndView对象时直接在其构造器中传入路径地址的字符串形式；第二种是通过ModelAndView的空参构造器创建对象，然后调用该对象的setViewName方法设置视图名（即路径地址），最后将该对象返回。

|  |
| --- |
| @RequestMapping("/return3") public ModelAndView return3() {  return new ModelAndView("ok"); } |

|  |
| --- |
| @RequestMapping("/return3") public ModelAndView return3() {  ModelAndView modelAndView = new ModelAndView();  modelAndView.setViewName("ok");  return modelAndView; } |

### 显示转发

进行显示转发只需要设置视图名（即路径地址）时在最前面添加forward:即可，显示转发时，SpringMVC会忽略掉视图解析器。

|  |
| --- |
| @RequestMapping("/return4") public ModelAndView return4() {  ModelAndView modelAndView = new ModelAndView();  modelAndView.setViewName("forward:/pages/ok.jsp");  return modelAndView; } |

### 显示重定向

进行显示转发只需要设置视图名（即路径地址）时在最前面添加redirect:即可，显示转发时，SpringMVC会忽略掉视图解析器。

|  |
| --- |
| @RequestMapping("/return5") public ModelAndView return5() {  ModelAndView modelAndView = new ModelAndView();  modelAndView.setViewName("redirect:/pages/ok.jsp");  return modelAndView; } |

## 返回void（不推荐使用）

返回值为void时会使用默认的请求资源名作为默认转发的逻辑资源名。

在开发中，一般不会使用这种方式，因为这种方式的可读性太差，程序的跳转无法直接通过代码看出；而且破坏了SpringMVC的结构。

### 没有返回值

|  |
| --- |
| @RequestMapping("/return6") public void return6() {  System.*out*.println("方法被调用了"); } |

### 显示转发

|  |
| --- |
| @RequestMapping("/return7") public void return7(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  request.getRequestDispatcher("/pages/ok.jsp").forward(request, response); } |

### 显示重定向

|  |
| --- |
| @RequestMapping("/return8") public void return8(HttpServletRequest request, HttpServletResponse response) throws IOException {  response.sendRedirect(request.getContextPath() + "/pages/ok.jsp"); } |

### 输出响应数据

|  |
| --- |
| @RequestMapping("/return9") public void return9(HttpServletRequest request, HttpServletResponse response) throws IOException {  response.getWriter().write("hello world!"); } |

## 返回值路径

无论返回值是String类型的还是ModelAndView类型的，无论是显示转发还是显示重定向，都会根据路径地址是否是**"/"**开头来将路径地址分为相对路径和绝对路径。在

### 相对路径

路径地址不是**"/"**开头时为相对路径，相对路径在跳转时会参照浏览器当前的地址栏中的地址

示例：forward:pages/ok.jsp

当前浏览器栏地址是：http://localhost:8080/spring\_mvc\_hello/abc/return1

将参数参照地址后得到：http://localhost:8080/spring\_mvc\_hello/abc/pages/ok.jsp

### 绝对路径

路径地址不是**"/"**开头时为绝对路径，绝对路径在跳转时不会参照浏览器当前的地址栏中的地址，而是将**"http://ip:port/工程名/"**和**"/"**后面的路径地址进行拼接得到跳转的路径地址。

示例：forward:/pages/ok.jsp

实际跳转的路径地址： http://ip:port/工程名/pages/ok.jsp

开发过程中要使用绝对路径，不要使用相对路径，因为相对路径存在二义性，从而会产生不必要的错误。

### 重定向路径

在服务器代码中，重定向只有在调用response.sendRedirect("/")时的斜杠会被解析为http://ip:port/，在SpringMVC中路径地址以斜杠开头则该斜杠都会被解析为http://ip:port/工程名/。

# 域数据存储

|  |
| --- |
| <body> showData.jsp页面<br/><hr/> request域key数据：${ requestScope.key }<br/><hr/> session域key数据：${ sessionScope.key }<br/><hr/> servletContext域key数据：${ applicationScope.key }<br/><hr/> </body> |

## 原生API存储（不推荐使用）

Request域

|  |
| --- |
| @RequestMapping("/requestScope") public String requestScope(HttpServletRequest request){  request.setAttribute("key", "requestValue");  return "showData"; } |

Session域

|  |
| --- |
| @RequestMapping("/sessionScope") public String sessionScope(HttpSession session){  session.setAttribute("key", "sessionValue");  return "showData"; } |

ServletContext域

在Controller中获取ServletContext对象有两种方法，一种是通过Session对象获取，另一种是使用@Autowired注解注入。

|  |
| --- |
| @RequestMapping("/applicationScope") public String applicationScope(HttpSession session){  ServletContext servletContext = session.getServletContext();  servletContext.setAttribute("key", "applicationValue");  return "showData"; } |

|  |
| --- |
| @Autowired ServletContext servletContext; @RequestMapping("/applicationScope2") public String applicationScope(){  servletContext.setAttribute("key", "applicationValue2");  return "showData"; } |

## 隐含模型

BindingAwareModelMap类又叫做隐含模型，可以通过传入其接口或父类的方式来存储数据到request域中，即以Map、Model或ModelMap形式(其底层都是BindingAwareModelMap的)保存数据到域中。将数据存储到map、model或modelMap中时，SpringMVC底层会自动的将存储的数据保存到request域中。

|  |
| --- |
| @RequestMapping("/mapToRequest") public String mapToRequest(Map<String,Object> map){  map.put("key", "mapToRequestValue");  return "showData"; } @RequestMapping("/modelToRequest") public String modelToRequest(Model model){  model.addAttribute("key", "modelToRequestValue");  return "showData"; } @RequestMapping("/modelMapToRequest") public String modelMapToRequest(ModelMap modelMap){  modelMap.addAttribute("key", "modelMapToRequestValue");  return "showData"; } |

BindingAwareModelMap和Map、Model和ModelMap的关系：

|  |
| --- |
| public class BindingAwareModelMap extends ExtendedModelMap  public class ExtendedModelMap extends ModelMap implements Model  public class ModelMap extends LinkedHashMap<String, Object>  public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> |

## ModelAndView

除了上述方式可以将数据存储到request域中外，还可以使用ModelAndView类将数据存储到request域中。其方式是先通过向ModelAndView类构造器中传入转发的访问地址创建该类对象，然后调用addObject方法传入要存储的值即可。

|  |
| --- |
| @RequestMapping("/modelAndViewToRequest") public ModelAndView modelAndViewToRequest(){  ModelAndView modelAndView = new ModelAndView("showData");  modelAndView.addObject("key", "modelAndViewValue");  return modelAndView; } |

## @SessionAttributes

@SessionAttributes注解可以标注在类上，可以指定隐含模型中的哪些数据可以存储到session域中。

value|names属性：表示将request域中指定的key的键值对信息存储到session域中，如果有多个key则用数组方式。（value和names属性互为别名，因此两者功能相同）

value = {“key1”,”key2”...“keyn”}

types属性：表示将request域中指定的value类型的键值对信息保存到session域中，如果有多个value类型则用数组方式。

types = {String.class,Book.class...}

|  |
| --- |
| //types = Integer.class表示将隐含模型中的Integer类型的数据保存到session域中  @SessionAttributes(names = "key" ,types = Integer.class) @Controller public class ScopeController {  @RequestMapping("/sessionTest")  public String sessionTest(Map<String,Object> map){  //map.put("key", "sessionTest");  map.put("key", new Integer(100));  return "showData";  }  } |

实际开发中，@SessionAttributes注解禁止使用，因为其本身存在bug。

## @ModelAttribute

@ModelAttribute注解可以标注在方法和参数上，该注解有三个常见的作用：

1、被标注了@ModelAttribute的方法会在Controller的目标方法之前执行；

2、目标方法的参数（JavaBean对象）会先从隐含模型中获取值传入，如果客户端传入了值就会覆盖；

|  |
| --- |
| @ModelAttribute public void modelAttributeFub(Map<String,Object> map){  map.put("book", new Book(99, "java")); } @RequestMapping("/test2") //默认情况下SpringMVC会先将参数类型转成小字母小写从隐含模型中寻找key取值传入参数中 public String test2(Book book){  System.*out*.println("参数book的值："+book);  return "showData"; } |

3、被标注在参数上，参数值会按照指定的key从隐含模型中获取值；

|  |
| --- |
| @ModelAttribute public void modelAttributeFub(Map<String,Object> map){  map.put("bookABC", new Book(99, "java")); } @RequestMapping("/test2") //@ModelAttribute注解标注的参数可以按指定的key在request域中获取值 public String test2(@ModelAttribute("bookABC") Book book){  System.*out*.println("参数book的值："+book);  return "showData"; } |

# 传统CRUD

## 准备操作

### 配置文件

创建数据库

drop database if exists springmvc;  
create database springmvc;  
use springmvc;  
##创建图书表  
create table t\_book  
(  
 `id` int(11) primary key auto\_increment, ## 主键  
 `name` varchar(50) not null, ## 书名   
 `author` varchar(50) not null, ## 作者  
 `price` decimal(11, 2) not null, ## 价格  
 `sales` int(11) not null, ## 销量  
 `stock` int(11) not null ## 库存  
);  
## 插入初始化测试数据  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'java从入门到放弃', '国哥', 80, 9999, 9);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '数据结构与算法', '严敏君', 78.5, 6, 13);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'C语言程序设计', '谭浩强', 28, 52, 74);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, 'Lua语言程序设计', '雷丰阳', 51.5, 48, 82);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '西游记', '罗贯中', 12, 19, 9999);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '水浒传', '施耐庵', 33.05, 22, 88);  
insert into t\_book(`id`, `name`, `author`, `price`, `sales`, `stock`)  
values (null, '操作系统原理', '刘优', 133.05, 122, 188);  
## 查看表内容  
select id, name, author, price, sales, stock  
from t\_book;

log4j.properties

|  |
| --- |
| # Global logging configuration log4j.rootLogger=INFO, stdout # Console output... log4j.appender.stdout=org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout=org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

jdbc.properties

|  |
| --- |
| user=root password=root url=jdbc:mysql://localhost:3306/springmvc driverClassName=com.mysql.jdbc.Driver initialSize=5 maxActive=10 |

web.xml

|  |
| --- |
| <servlet>  <servlet-name>dispatcher</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup> </servlet> <servlet-mapping>  <servlet-name>dispatcher</servlet-name>  <url-pattern>/</url-pattern> </servlet-mapping> |

springmvc.xml

|  |
| --- |
| <!-- 配置包扫描 --> <context:component-scan base-package="com"></context:component-scan> <!-- 加载jdbc.properties属性配置文件 --> <context:property-placeholder location="classpath:jdbc.properties" /> <!-- 配置视图解析器 --> <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/book/" />  <property name="suffix" value=".jsp" /> </bean> <!-- 配置数据库连接池 --> <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="username" value="${user}" />  <property name="password" value="${password}" />  <property name="url" value="${url}" />  <property name="driverClassName" value="${driverClassName}" />  <property name="initialSize" value="${initialSize}" />  <property name="maxActive" value="${maxActive}" /> </bean> <!-- 配置JdbcTemplate工具类操作数据库 --> <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource"></property> </bean> |

### 前端页面

#### bookList.jsp

|  |
| --- |
| <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>图书列表</title>  <% String basePath = request.getScheme() + "://" + request.getServerName() + ":" + request.getServerPort()  + request.getContextPath() + "/"; %>  <base href="<%=basePath %>"/>  <script type="text/javascript" src="${ pageContext.request.contextPath }/script/jquery-1.7.2.js"></script>  <script type="text/javascript">  // 页面加载完成之后  $(function () {  $("a.deleteAClass").click(function () {  var name = $(this).parent().parent().find("td:first").text();  return confirm("你确定要删除【" + name + "】吗?");  });  });  </script>  <style type="text/css">  table { border: 1px blue solid; width: 700px; border-collapse: collapse; }  td, th { border: 1px green solid; }  div.menu { width: 700px; text-align: right; }  </style> </head> <body> <center>  <h2>图书列表管理页面</h2>  <div class="menu"><a href="${ pageContext.request.contextPath }/book/bookEdit.jsp">添加图书</a></div>  <table>  <tr bgcolor="#FF8888">  <th>书名</th>  <th>作者</th>  <th>价格</th>  <th>销量</th>  <th>库存</th>  <th>操作</th>  </tr>  <**c:forEach** items="${ requestScope.list }" var="book">  <tr>  <td>${ book.name }</td>  <td>${ book.author }</td>  <td>${ book.price }</td>  <td>${ book.sales }</td>  <td>${ book.stock }</td>  <td>  <a class="deleteAClass" href="${ pageContext.request.contextPath }/book/delete?id=${ book.id} ">删除</a>、  <a href="${ pageContext.request.contextPath }/book/getBook?id=${ book.id }">修改</a>  </td>  </tr>  </**c:forEach**>  </table> </center> </body> |

#### bookEdit.jsp

|  |
| --- |
| <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>图书列表</title>  <% String basePath = request.getScheme() + "://" + request.getServerName() + ":" + request.getServerPort()   + request.getContextPath() + "/"; %>  <base href="<%=basePath %>" /> </head> <body>  <center>  <h3>添加图书</h3>  <form action="${ pageContext.request.contextPath }/book/${ empty param.id ? "add" : "update" }" method="post">  <**c:if** test="${ not empty param.id }">  <input type="hidden" name="id" value="${ book.id }" />  </**c:if**>  <table>  <tr>  <td>书名</td>  <td><input name="name" type="text" value="${ book.name }"/></td>  </tr>  <tr>  <td>作者</td>  <td><input name="author" type="text" value="${ book.author }"/></td>  </tr>  <tr>  <td>价格</td>  <td><input name="price" type="text" value="${ book.price }"/></td>  </tr>  <tr>  <td>销量</td>  <td><input name="sales" type="text" value="${ book.sales }"/></td>  </tr>  <tr>  <td>库存</td>  <td><input name="stock" type="text" value="${ book.stock }"/></td>  </tr>  <tr>  <td align="center" colspan="2">  <input type="submit" />  </td>  </tr>  </table>  </form>  </center> </body> |

### Dao和Service

#### Book.java

|  |
| --- |
| public class Book {  private Integer id;  private String name;  private String author;  private BigDecimal price;  private Integer sales;  private Integer stock; } |

#### BookDao.java

|  |
| --- |
| @Repository public class BookDao {  @Autowired  private JdbcTemplate jdbcTemplate;  public int saveBook(Book book) {  String sql = "insert into t\_book(`name` , `author` , `price` , `sales` , `stock`) values(?,?,?,?,?)";  return jdbcTemplate.update(sql, book.getName(), book.getAuthor(), book.getPrice(), book.getSales(), book.getStock());  }  public int deleteBookById(Integer id) {  String sql = "delete from t\_book where id = ?";  return jdbcTemplate.update(sql, id);  }   public int updateBook(Book book) {  String sql = "update t\_book set name=?,author=?,price=?,sales=?,stock=? where id = ?";  return jdbcTemplate.update(sql, book.getName(), book.getAuthor(), book.getPrice(), book.getSales(),  book.getStock(), book.getId());  }  public Book queryBookById(Integer id) {  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock` from t\_book where id = ?";  return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Book>(Book.class), id);  }  public List<Book> queryBooks() {  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock` from t\_book";  return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Book>(Book.class));  } } |

#### BookService.java

|  |
| --- |
| @Service public class BookService {  @Autowired  private BookDao bookDao;  public void addBook(Book book) {  bookDao.saveBook(book);  }  public void deleteBookById(Integer id) {  bookDao.deleteBookById(id);  }  public void updateBook(Book book) {  bookDao.updateBook(book);  }  public Book queryBookById(Integer id) {  return bookDao.queryBookById(id);  }  public List<Book> queryBooks() {  return bookDao.queryBooks();  } } |

## CRUD

### 添加

#### 中文乱码

在添加图书时，如果客户端（浏览器）传递的参数中有中文，则会产生乱码，此时可以通过在web.xml配置文件中配置SpringMVC提供的Filter过滤器来解决乱码问题。

|  |
| --- |
| <!-- 配置中文乱码的Filter过滤器 --> <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <!-- 配置字符集 -->  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  <init-param>  <!--配置请求字符集-->  <param-name>forceRequestEncoding</param-name>  <param-value>true</param-value>  </init-param>  <init-param>  <!--配置响应字符集-->  <param-name>forceResponseEncoding</param-name>  <param-value>true</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> |

|  |
| --- |
| @RequestMapping("/book") @Controller public class BookController {  @Autowired  private BookService bookService;  @RequestMapping("/add")  public String add(Book book) {  // 调用bookService添加图书  bookService.addBook(book);  // 重定向回图书列表管理页面  return "redirect:/book/list";  } } |

### 删除

|  |
| --- |
| @RequestMapping("/delete")  public String delete(Integer id) {  bookService.deleteBookById(id);  return "redirect:/book/list";  } |

### 修改

修改图书信息前需要进行图书信息的回显操作，然后才调用controller的update方法进行修改数据库中的图书信息。

|  |
| --- |
| @RequestMapping("/getBook")  public ModelAndView getBook(Integer id) {  ModelAndView modelAndView = new ModelAndView();  // 查询你需要修改的图书信息，保存到request域中  Book book = bookService.queryBookById(id);  modelAndView.addObject("book", book);  // 转发到/book/bookEdit.jsp页面去显示  modelAndView.setViewName("bookEdit");  return modelAndView;  } |

|  |
| --- |
| @RequestMapping("/update")  public String update(Book book) {  bookService.updateBook(book);  return "redirect:/book/list";  } |

### 查询

|  |
| --- |
| @RequestMapping("/list")  public String list(Map<String, Object> map) {  // 1查询全部的图书，保存到request域中。  map.put("list", bookService.queryBooks());  // 转发到/book/bookList.jsp页面  return "bookList";  } |

## 小结

Dao层使用了JdbcTemplate工具类，功能和之前的DbUtils类似；Service功能和之前一样；而Web层则是使用Controller控制器替代了Servlet程序，不同于Servlet使用BaseServlet程序根据action=xxx功能方法分发到不同的模块，SpringMVC使用DispatcherServlet程序处理请求的分发到不同的Controller控制器以及方法。

# Restful风格

## 概述

Restful是一种软件架构设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。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。

### 原则条件

Restful指的是一组架构约束条件和原则，满足这些约束条件和原则的应用程序或设计就是 Restful。Web 应用程序最重要的 Restful原则是客户端和服务器之间的交互在请求之间是无状态的。从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点重启，客户端不会得到通知。此外，无状态请求可以由任何可用服务器回答，这十分适合云计算之类的环境。客户端可以缓存数据以改进性能。

在服务器端，应用程序状态和功能可以分为各种资源。资源是一个有趣的概念实体，它向客户端公开。资源的例子有：应用程序对象、数据库记录、算法等等。每个资源都使用 URI (Universal Resource Identifier) 得到一个唯一的地址。所有资源都共享统一的接口，以便在客户端和服务器之间传输状态。使用的是标准的 HTTP 方法，比如 GET、PUT、POST 和 DELETE。Hypermedia 是应用程序状态的引擎，资源表示通过超链接互联。

简而言之，Restful是一种设计风格，就是使用一个url地址表示一个唯一的资源，把原来的请求参数加入到请求资源地址中，然后原来请求的增，删，改，查操作改为使用HTTP协议中请求方式GET、POST、PUT、DELETE表示。

### 与传统请求方式比较

传统的方式是：

比如：http://ip:port/工程名/资源名?请求参数

举例：http://127.0.0.1:8080/springmvc/book?action=delete&id=1

restful风格是：

比如：http://ip:port/工程名/资源名/请求参数/请求参数

举例：http://127.0.0.1:8080/springmvc/book/1

请求的动作由请求方式决定，如果请求方式是DELETE，则表示删除。

### 请求方式与CRUD

GET请求 对应 查询

http://ip:port/工程名/book/1 HTTP请求GET 表示要查询id为1的图书

http://ip:port/工程名/book HTTP请求GET 表示查询全部的图书

POST请求 对应 添加

http://ip:port/工程名/book HTTP请求POST 表示要添加一个图书

PUT请求 对应 修改

http://ip:port/工程名/book/1 HTTP请求PUT 表示要修改id为1的图书信息

DELETE请求 对应 删除

http://ip:port/工程名/book/1 HTTP请求DELETE 表示要删除id为1的图书信息

### 四种请求的发送方式

GET和POST请求只要在form标签中设置method属性即可。

PUT和DELETE请求需要在method属性为POST的form表单中添加额外的隐藏域，并且该隐藏域的name属性的值为\_method，value属性的值根据请求方式设置为PUT或DELETE，最后在web.xml配置文件中配置一个Filter过滤器（org.springframework.web.filter.HiddenHttpMethodFilter），该过滤器只能位于处理中文乱码的Filter过滤器后面。

### 过滤器位置

HiddenHttpMethodFilter过滤器只能位于处理中文乱码的CharacterEncodingFilter过滤器后面原因在于HiddenHttpMethodFilter底层调用了request.getParameter(this.methodParam)方法，如果其位置在处理乱码问题的过滤器前面会导致CharacterEncodingFilter过滤器失效。

## 简单实现

配置支持Restful风格请求的过滤器。

|  |
| --- |
| <!-- 配置支持restful风格请求的过滤器，将隐藏域value属性的值转为请求方式-->  <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> |

|  |
| --- |
| <a href="${pageContext.request.contextPath}/book/abc/1">查询指定编号和图书</a><br/> <a href="${pageContext.request.contextPath}/book">查询全部图书</a><br/> <form action="${pageContext.request.contextPath}/book" method="post">  <input type="submit" value="添加图书"> </form> <form action="${pageContext.request.contextPath}/book/1" method="post">  <%--设置请求方式为PUT时，form标签的method属性值必须为post  添加隐藏域如下，标识为put请求，然后配置restful风格支持的Filter过滤器--%>  <input type="hidden" name="\_method" value="PUT">  <input type="submit" value="修改图书"> </form> <form action="${pageContext.request.contextPath}/book/1" method="post">  <%--设置请求方式为DELETE时，form标签的method属性值必须为post  添加隐藏域如下，标识为delete请求，然后配置restful风格支持的Filter过滤器--%>  <input type="hidden" name="\_method" value="DELETE">  <input type="submit" value="删除图书"> </form> |

|  |
| --- |
| @Controller public class RestfulController {  @RequestMapping(value = "/book",method = RequestMethod.*POST*)  public String add(){  System.*out*.println("addBook");  return "forward:/index.jsp";  }  @RequestMapping(value = "/book/1",method = RequestMethod.*DELETE*)  public String delete(){  System.*out*.println("deleteBook");  return "forward:/index.jsp";  }  @RequestMapping(value = "/book/1",method = RequestMethod.*PUT*)  public String update(){  System.*out*.println("updateBook");  return "forward:/index.jsp";  }  @RequestMapping(value = "/book" , method = RequestMethod.*GET*)  public String list(){  System.*out*.println("queryAll");  return "forward:/index.jsp";  } } |

## Tomcat高版本无法转发

在Tomcat8之后的一些高版本，使用Restful风格访问然后无法转发到jsp页面，可以通过如下两种方式解决：

1、使用重定向而不是请求转发

2、将请求转发的页面设置为错误页面

|  |
| --- |
| <%@ **page** contentType="text/html;charset=UTF-8" language="java" isErrorPage="true" %> |

## 请求信息获取

对于Restful风格的请求信息可以通过@PathVariable注解来获取，在方法的参数上标注该注解并指定名称就可获取请求参数。

|  |
| --- |
| //在请求路径中使用{id}标识路径参数，其中id是这个参数的名称  //在参数中使用@PathVariable("id")注解获取请求路径中指定名称的路径变量的值  @RequestMapping(value = "/book/{name}/{id}",method=RequestMethod.*GET*)  public String queryBookById(@PathVariable("id") Integer id,@PathVariable("name") String name){  System.*out*.println("id:" + id);  System.*out*.println("name:" + name);  return "forward:/index.jsp";  } |

## Restful风格CRUD

### 查询列表功能

|  |
| --- |
| <body> <a href="${ pageContext.request.contextPath }/book">图书列表管理页面</a> </body> |

|  |
| --- |
| @Controller public class BookController {  @Autowired  private BookService bookService;  //请求方式必须为GET方式才会调用该方法  @RequestMapping(value = "/book",method = RequestMethod.*GET*)  public String list(Map<String, Object> map) {  map.put("list", bookService.queryBooks());  return "bookList";  }  } |

### 添加图书功能

|  |
| --- |
| //请求方式必须为POST方式才会调用该方法 @RequestMapping(value = "/book", method = RequestMethod.*POST*) public String add(Book book) {  bookService.addBook(book);  return "redirect:/book"; } |

|  |
| --- |
| <%--使用restful风格进行添加操作就不用再用隐藏域的方式来判断当前是添加还是修改操作了--%> <form action="${ pageContext.request.contextPath }/book" method="post"> |

### 删除图书功能

|  |
| --- |
| <script type="text/javascript">  $(function () {  $("a.deleteAClass").click(function () {  let name = $(this).parent().parent().find("td:first").text();  if( confirm("你确定要删除【" + name + "】吗?")){  //prev()方法的作用是获取前一个兄弟标签  $(this).prev().submit();  return false;  }else{  return false;  }  });  }); </script>  <%--restful风格发送DELETE请求需要将form表单的method属性设置为POST，然后添加隐藏域 最后还要在web.xml中配置支持restful风格请求的过滤器，且该过滤器只能位于处理乱码过滤器后面--%> <form action="${pageContext.request.contextPath}/book/${book.id}" method="post">  <%--该隐藏域的name属性只能设置为\_method，value属性只能设置为DELETE--%>  <input type="hidden" name="\_method" value="DELETE"/> </form> <a class="deleteAClass">删除</a> |

|  |
| --- |
| @RequestMapping(value = "/book/{id}",method = RequestMethod.*DELETE*) //必须要在delete方法的参数上添加@PathVariable注解，否则无法获取请求参数信息 public String delete(@PathVariable("id") Integer id) {  bookService.deleteBookById(id);  return "redirect:/book"; } |

### 修改图书功能

首先要查询需要修改的图书回显到页面。

|  |
| --- |
| @RequestMapping(value = "/book/{id}",method = RequestMethod.*GET*) public ModelAndView getBook(@PathVariable("id") Integer id) {  ModelAndView modelAndView = new ModelAndView();  Book book = bookService.queryBookById(id);  modelAndView.addObject("book", book);  modelAndView.setViewName("bookEdit");  return modelAndView; } |

|  |
| --- |
| <a href="${pageContext.request.contextPath}/book/${book.id}">修改</a> |

然后对图书信息进行修改并提交到服务器。

|  |
| --- |
| @RequestMapping(value="/book/{id}",method = RequestMethod.*PUT*) public String update(Book book) {  bookService.updateBook(book);  return "redirect:/book"; } |

|  |
| --- |
| <%--判断是添加图书还是修改图书，如果request域中的book对象不为null，则表示为修改图书--%> <**c:if** test="${ not empty requestScope.book }"><h3>修改图书</h3>  <form action="${pageContext.request.contextPath}/book/${book.id}" method="POST">  <input type="hidden" name="\_method" value="put"/>  <input type="hidden" name="id" value="${book.id}"/> </**c:if**> |

# 自定义参数转换器

在SpringMVC中有WebDataBinder类，这个类专门用来负责将请求参数类型转换，以及请求参数数据验证，错误信息绑定等功能。

在WebDataBinder类中有三个组件分别处理三种不同的功能，三大组件分别如下所示：

conversionService 负责处理参数类型转换。把请求的参数转换成为Controller中的方法参数值。

converters 在ConversionService组件中需要各种类型转换器，依赖于各种转换器类去实现转换工作。

validators 负责验证传入的参数值是否合法。

bindingResult 负责接收验证后的错误信息。

converters是conversionService的子属性，conversionService是WebDataBinder的子属性。

## 自定义String到java.util.Date类型

### xml配置方式

自定义参数转换器步骤如下：

1、实现Converter接口，定义目标功能的类型转换器

2、在springmvc配置文件中配置FormattingConversionServiceFactoryBean

3、将自定义的类型转换器注入到FarmattingConversionServiceFactoryBean的set集合属性converters中

4、将自定义的类型转换器service服务器注入到WebDataBinder类中

|  |
| --- |
| public class MyStringToDateConverter implements Converter<String, Date> {  @Override  public Date convert(String source) {  if (source == null) {  return null;  }  String time = source.trim();  if ("".equals(time)) {  return null;  }  SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");  try {  return simpleDateFormat.parse(time);  } catch (ParseException e) {  e.printStackTrace();  throw new IllegalArgumentException("Invalid Date value" + source);  }  } } |

|  |
| --- |
| <!--配置转换器service组件--> <bean id="formattingConversionService"  class="org.springframework.format.support.FormattingConversionServiceFactoryBean">  <!--将自定义的类型转换器注到转换器service组件中-->  <property name="converters">  <set>  <bean class="com.crud.converter.MyStringToDateConverter"/>  </set>  </property> </bean> <!--使用自定义的类型转换器service组件--> <mvc:annotation-driven conversion-service="formattingConversionService"/> |

|  |
| --- |
| <form action="${ pageContext.request.contextPath }/person/add">  编号：<input type="text" name="id"/><br/>  姓名：<input type="text" name="name"/><br/>  电话：<input type="text" name="phone"/><br/>  生日：<input type="text" name="birthDate"/><br/>  <input type="submit" /> </form> |

|  |
| --- |
| @Controller public class PersonController {  @RequestMapping("/person/add")  public String add(Person person){  System.*out*.println("add Person" + person);  return "redirect:/index.jsp";  } } |

### 注解方式

可以使用@DateTimeFormat注解标注在JavaBean需要转换的属性上，可以实现字符串和Date类型的转换，pattern=="yyyy-MM-dd"表示转换的格式，此时就不用在springmvc.xml中配置类型转换器。

|  |
| --- |
| public class Person {  private Integer id;  private String name;  private String phone;  /\* @DateTimeFormat 注解可以实现字符串和日期时间类型的转换  pattern="yyyy-MM-dd"属性设置你客户端发送过来的日期时间的格式 \*/  @DateTimeFormat(pattern="yyyy-MM-dd")  private Date birthDate; } |

# 文件上传和下载

## 文件上传

在SpringMVC中实现文件上传需要配置文件上传解析器CommonsMultipartResolver，然后编写Controller控制器的代码。

需要的jar包有commons-fileupload、commons-io

配置前端文件上传表单：

|  |
| --- |
| <form action="${ pageContext.request.contextPath }/upload"   method="post" enctype="multipart/form-data">  用户名：<input type="text" name="username"/><br/>  头像：<input type="file" name="photo"/><br/>  <input type="submit" /> </form> |

配置文件上传解析器：

|  |
| --- |
| <!-- 配置用来解析上传数据的解析器，id值必须是：multipartResolver --> <bean id="multipartResolver"   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <!-- 配置字符串的字符集 -->  <property name="defaultEncoding" value="UTF-8" /> </bean> |

编写文件上传的Controller控制器，使用MultipartFile类型来接收文件，将服务器获取到的上传文件存储到E盘：

|  |
| --- |
| @RequestMapping("/upload") public String upload(@RequestParam("username") String username, @RequestParam("photo") MultipartFile photo) {  System.*out*.println("用户名是：" + username);  if (photo != null && !photo.isEmpty()) {  System.*out*.println("上传的文件名：" + photo.getOriginalFilename());  try {  photo.transferTo(new File("e:\\" + photo.getOriginalFilename()));  } catch (Exception e) {  e.printStackTrace();  }  }  return "/upload.jsp"; } |

## 文件下载

要实现文件下载功能则需要操作响应头和响应体，可以使用ResponseEntity类来实现这个功能。

|  |
| --- |
| @RequestMapping("/download") public ResponseEntity<byte[]> download(HttpSession session){  // 下载要读取到下载的文件并返回给客户端，还要设置响应头，通知客户端，返回数据以附件形式处理，设置返回的数据类型Content-Type响应头  String filePath = "/file/spring-core.jar";  // 获取servletContext对象  ServletContext servletContext = session.getServletContext();  // 读取需要下载的文件，得到输入流  InputStream inputStream = servletContext.getResourceAsStream(filePath);  byte[] body = null;  try {  body = IOUtils.*toByteArray*(inputStream);// 请求体 == 需要下载的文件内容  } catch (IOException e) {  e.printStackTrace();  }   // 得到需要下载的数据类型  String mimeType = servletContext.getMimeType(filePath);  HttpHeaders httpHeaders = new HttpHeaders();// 请求头或响应头  httpHeaders.add("Content-Type", mimeType);// 添加响应头  httpHeaders.add("Content-disposition", "attachment; filename=spring-core.jar");  ResponseEntity<byte[]> responseEntity = new ResponseEntity(body,httpHeaders,HttpStatus.*OK*);  return responseEntity; } |

# 普通数据和json互转

## 数据转为json

首先在Controller中编写方法接收请求并返回数据对象，然后在方法上使用@ResponseBody注解使返回值自动转换为json数据。

|  |
| --- |
| //@ResponseBody注解可以将controller方法的返回值转成为json字符串返回给客户端 @ResponseBody @RequestMapping("/queryPersonById") public Person queryPersonById() {  // 调用xxService.xxxx  return new Person(1, "中国", "18610541354", new Date()); } @ResponseBody @RequestMapping("/queryPersons") public List<Person> queryPersons() {  List<Person> personList = new ArrayList<Person>();  for (int i = 0; i < 10; i++) {  personList.add(new Person(i, "name" + i, "phone" + i, new Date()));  }  return personList; } |

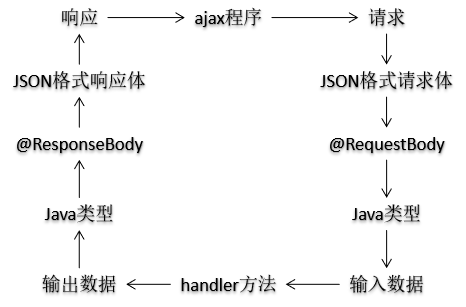
## json转为普通数据

可以使用@RequestBody注解将Ajax请求中的json格式数据转换为Java类型的数据传递给handler进行处理。

|  |
| --- |
| @ResponseBody @RequestMapping("/admin/batch/remove.json") public ResultEntity<String> doBatchRemove(@RequestBody List<Integer> adminList,  HttpSession session){  …… } |

## @ResponseBody和@RequestBody

如果handler类中每一个方法都标记了@ResponseBody注解，那么该注解就可以提取到类上。



## @RestController

如果一个类上同时被@ResponseBody和@Controller注解标注，则可以使用@RestController注解替换这两个注解来达到同样的效果。

@RestController = @ResponseBody + @Controller

## 修改时间在json中的格式

将数据转为json格式时，Date类型的数据转换为的是long类型的长整数，可以使用@JsonFormat注解标注在需要转换的JavaBean的Date时间属性或该属性对应的getXxx方法上，即可将其转换为json格式。

|  |
| --- |
| /\* @JsonFormat 可以将属性转换成为我们需要的格式  pattern属性设置Date类型的数据转换为的时间的格式  timezone属性设置时区的的时差(北京时间跟格林时间差8小时) \*/ @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss",timezone = "GMT+8") private Date birthDate; |

|  |
| --- |
| @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss",timezone = "GMT+8") public Date getBirthDate() {  return birthDate; } |

# 获取请求信息

## @RequestBody注解

使用@RequestBody注解标注在Contorller的方法的方法参数上，即可将请求体的数据都注入到该参数中。

|  |
| --- |
| <form action="${ pageContext.request.contextPath }/httpEntityTest"   method="post" >  用户名：<input type="text" name="username"/><br/>  密码：<input type="password" name="password"/><br/>  <input type="submit" /> </form> |

|  |
| --- |
| //@RequestBody注解标注在请求方法的参数上，表示把全部的请求体数据都注入到此参数中 @RequestMapping("/getRequestBody") public String getRequestBody(@RequestBody String requstBody) {  System.*out*.println("请求体是：" + requstBody);  return "/upload.jsp"; } |

## HttpEntity类

使用HttpEntity类作为Controller方法参数可以获取当前请求的请求头和请求体，SpringMVC会在接收请求后将数据封装到HttpEntity的对象中。

|  |
| --- |
| @RequestMapping("/httpEntityTest") public String httpEntityTest(HttpEntity<String> entity) {  //HttpEntity可以获取当前请求的请求头，和请求体（SpringMVC接收到请求之后都封装进去了）  System.*out*.println( "请求体：" + entity.getBody() );  System.*out*.println( "请求头：" + entity.getHeaders() );  return "/upload.jsp"; } |

# 拦截器HandlerInterceptor

要使用HandlerInterceptor拦截器首先要实现HandlerInterceptor接口，然后重写该接口中的三个方法afterCompletion、postHandle和preHandle,最后要在springmvc配置文件中配置拦截路径。

|  |
| --- |
| public class MyHelloHandlerInterceptor implements HandlerInterceptor {  /\* afterCompletion在拦截的请求页面渲染完成之后执行 \*/  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object arg2, Exception arg3) {  System.*out*.println("5、afterCompletion 方法 页面渲染完成");  }  /\* postHandle在拦截到的目标方法执行之后执行<br/> \*/  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object arg2, ModelAndView arg3) {  System.*out*.println("3、postHandle方法 目标方法之后");  }  /\* preHandle在拦截到的目标方法执行之前执行 \*/  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object arg2) {  System.*out*.println("1、preHandle方法 目标方法之前");  /\* 返回false表示阻止程序继续运行，返回true，表示放行，让程序继续往后走 \*/  return true;  } } |

Controller控制器中被拦截的方法：

|  |
| --- |
| @RequestMapping("/hello") public String hello() {  System.*out*.println("2. 这是目标方法 hello () ");  return "/ok.jsp"; } |

被拦截跳转的页面：

|  |
| --- |
| <body>  <%  System.*out*.println("4、页面开始渲染");  %> </body> |

## 拦截器执行顺序

正常情况其执行顺序如下所示：

preHandle方法🡪目标方法hello🡪postHandle方法🡪页面开始渲染（加载）🡪afterCompletion方法

preHandle方法返回false：

preHandle方法异常返回false时，后面四步均不执行。

preHandle方法返回true，目标方法异常

目标方法异常时，postHandle方法不执行，页面渲染异常，但是afterCompletion方法会执行。

目标方法正常，postHandle方法异常

postHandle方法异常时，页面渲染异常，但是afterCompletion方法会执行。

postHandle方法正常，页面渲染异常

页面渲染异常时，afterCompletion方法正常执行。

# 异常处理

## @ExceptionHandler

可以使用@ExceptionHandler注解标注在方法上，只要出了异常就会自动调用该方法处理异常。并且一般该方法的返回值是一个页面，表示跳转到显示异常信息的页面。

|  |
| --- |
| @RequestMapping("/hello") public String hello() {  System.*out*.println("2. 这是目标方法 hello () ");  int i = 12 / 0;  return "/ok.jsp"; }  //@ExceptionHandler 注解标注在方法上之后，只在出了异常，就会自动的调用此方法处理异常 @ExceptionHandler public String exceptionHandler(Exception e) {  System.*out*.println("exceptionHandler 抛出的异常：" + e);  return "/error500.jsp"; } |

## @ControllerAdvice

可以使用@ControllerAdvice注解标注在Controller控制器上，表示可以处理所有的Controller控制器中发生的异常。

|  |
| --- |
| //@ControllerAdvice表示该Controller类中异常处理方法可以处理全部Controller的异常 @ControllerAdvice @Controller public class TempController {  @ExceptionHandler  public String exceptionHandler(Exception e) {  System.*out*.println("TempController exceptionHandler 抛出的异常：" + e);  return "/error500.jsp";  }  } |

## 异常处理优先顺序

在局部异常处理和全局异常处理同时存在时，其优先顺序如下：

1、局部优先：发生异常的类中如果有可以处理该异常的方法则交给该方法处理，无视全局异常处理中存在的更精确的异常处理方法。

2、精确优化：经过局部优先原则选择处理异常方法后，再根据精确优化原则选择异常处理方法，即该方法中的异常参数类型与发生的异常类型越接近/匹配则越优先选择该方法。

|  |
| --- |
| @ExceptionHandler public String exceptionHandler1(Exception e) {  System.*out*.println("exceptionHandler1 抛出的异常：" + e);  return "/error500.jsp"; } //当有多个@ExceptionHandler异常处理方法的时候，异常类型越小（越匹配）抛出的异常，就会执行它 @ExceptionHandler public String exceptionHandler2(RuntimeException e) {  System.*out*.println("exceptionHandler2 抛出的异常：" + e);  return "/error500.jsp"; } //当有多个@ExceptionHandler异常处理方法的时候，异常类型越小（越匹配）抛出的异常，就会执行它 @ExceptionHandler public String exceptionHandler3(ArithmeticException e) {  System.*out*.println("exceptionHandler3 抛出的异常：" + e);  return "/error500.jsp"; } |

## 映射异常跳转

发生异常时跳转页面除了使用注解的方式来处理，还可以在springmvc.xml配置文件中配置异常映射解析器，用来将发生的异常类型和跳转页面进行匹配。但是这种方式只能用于跳转页面，而不能对异常进行处理。

|  |
| --- |
| <!-- 配置异常映射解析器 --> <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <property name="exceptionMappings">  <props>  <!-- 每个prop标签都是一个异常和跳转路径的映射 -->  <prop key="java.lang.Exception">/error500.jsp</prop>  <prop key="java.lang.RuntimeException">/error501.jsp</prop>  <prop key="java.lang.ArithmeticException">/error502.jsp</prop>  </props>  </property> </bean> |

使用这种方式时异常处理优先原则也是越精确越优先。

# 表单标签

在使用SpringMVC时可以使用Spring封装的一系列表单标签，这些标签都可以访问到ModelMap中的内容。在使用SpringMVC的表单标签之前，需要先在JSP中声明使用的标签，具体做法是在JSP文件的顶部加入以下指令：

|  |
| --- |
| <%@ **taglib** uri="http://www.springframework.org/tags/form" prefix="form" %> |

## form

使用Spring的form标签主要有两个作用

第一是它会自动的绑定来自Model中的一个属性值到当前form对应的实体对象，默认是command属性，这样就可以在form表单体里面方便的使用该对象的属性了；

第二是它支持在提交表单的时候使用除GET和POST之外的其他方法进行提交，包括DELETE和PUT等。

### 支持绑定表单对象

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post">  <table>  <tr>  <td>Name:</td>  <td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td>  <td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

这个时候如果Model中存在一个属性名称为command的javaBean，而且该javaBean拥有属性name和age的时候，在渲染上面的代码时就会取command的对应属性值赋给对应标签的值。如在上面的代码中，假设Model中存在一个属性名称为command的javaBean，且它的name和age属性分别为“Zhangsan”和“36”时，那么它在渲染时就会生成如下一段代码：

|  |
| --- |
| <form id="command" action="formTag/form.do" method="post">  <table>  <tr>  <td>Name:</td><td><input id="name" name="name" type="text" value="ZhangSan"/></td>  </tr>  <tr>  <td>Age:</td><td><input id="age" name="age" type="text" value="36"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </form> |

从上面生成的代码中可以看出，当没有指定form标签的id时它会自动获取该form标签绑定的Model中对应属性名称作为id，而对于input标签在没有指定id的情况下它会自动获取path指定的属性作为id和name。

指定form默认自动绑定的是Model的command属性值，当form对象对应的属性名称不是command时，Spring提供了一个commandName属性，可以通过该属性来指定使用Model中的哪个属性作为form需要绑定的command对象。除了commandName属性外，指定modelAttribute属性也可以达到相同的效果。这里假设上面代码中存放在Model中的是user对象而不是默认的command对象，那么代码就可以如下定义了：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>Name:</td><td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td><td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

### 支持全部的HTTP请求方法

|  |
| --- |
| <**form:form** action="formTag/form.do" method="delete" modelAttribute="user">  <table>  <tr>  <td>Name:</td><td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td><td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

在上面代码中设定了该form的提交方法是delete，这样在后台就可以给对应的请求方法的RequestMapping加上method为RequestMethod.DELETE的限制。上面的代码在进行渲染的时候生成的HTML代码如下所示：

|  |
| --- |
| <form id="user" action="formTag/form.do" method="post">  <input type="hidden" name="\_method" value="delete"/>  <table>  <tr>  <td>Name:</td><td><input id="name" name="name" type="text" value="ZhangSan"/></td>  </tr>  <tr>  <td>Age:</td><td><input id="age" name="age" type="text" value="36"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </form> |

从它生成的代码可以看出，Spring在实现除GET和POST之外的请求方法时，还是使用的POST方法进行请求，然后给表单加上了一个隐藏域，用以表示真正的请求方法，这个隐藏域的名称默认是“\_method”。这样定义后仍不能以delete方式访问到“formTag/form.do”，这样定义只是多加了一个用以表示请求方法的隐藏域而已，实际的请求方式还是POST。Spring提供HiddenHttpMethodFilter，通过这个Filter可以把以POST方式传递过来的表示实际请求方式的参数转换为对应的真正的Http请求方法。所以这时还需要在web.xml中加上如下代码：

|  |
| --- |
| <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> |

HiddenHttpMethodFilter只能对以POST方式进行传递的表示请求方式的隐藏域转换为真正的Http请求方式。直接在form：form标签的method中使用除GET和POST方法以外的其他方法时，Spring会自动生成以POST方式进行传递的表单以及对应的隐藏域。所以当我们需要手动的设置表示请求方法的隐藏域时，我们就需要指定表单的请求方式为POST，为GET将不会生效。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" modelAttribute="user">  <input type="hidden" name="\_method" value="head"/>  <table>  <tr>  <td>Name:</td><td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td><td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

上面代码就是一个手动定义请求方式的隐藏域的示例。这里表示请求方式的隐藏域的名称默认是“\_method”，如果不想使用这个默认值的话，也可以通过form:form标签的methodParam属性来指定。如下面这个示例：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" methodParam="requestMethod" modelAttribute="user">  <input type="hidden" name="requestMethod" value="head"/>  <table>  <tr>  <td>Name:</td><td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td><td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

同时要告诉HiddenHttpMethodFilter使用哪个表单参数作为methodParam，所以需要在配置HiddenHttpMethodFilter的时候指明methodParam对应的值。

|  |
| --- |
| <filter>  <filter-name>hiddenHttpMethodFilter</filter-name>  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>  <init-param>  <param-name>methodParam</param-name>  <param-value>requestMethod</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>hiddenHttpMethodFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> |

在有Multipart请求处理时，HiddenHttpMethodFilter需要在Multipart处理之后执行，因为在处理Multipart时需要从POST请求体中获取参数。所以通常会在HiddenHttpMethodFilter之前设立一个MultipartFilter。MultipartFilter默认会去寻找一个名称为filterMultipartResolver的MultipartResolver bean对象来对当前的请求进行封装。所以定义的MultipartResolver的名称不为filterMultipartResolver的时候就需要在定义MultipartFilter的时候通过参数multipartResolverBeanName来指定。

|  |
| --- |
| <filter>  <filter-name>multipartFilter</filter-name>  <filter-class>org.springframework.web.multipart.support.MultipartFilter</filter-class>  <init-param>  <param-name>multipartResolverBeanName</param-name>  <param-value>multipartResolver</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>multipartFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> <filter>  <filter-name>hiddenHttpMethodFilter</filter-name>  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>  <init-param>  <param-name>methodParam</param-name>  <param-value>requestMethod</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>hiddenHttpMethodFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> |

## input

SpringMVC的input标签会被渲染为一个type为text的普通Html input标签。使用SpringMVC的input标签的唯一作用就是它能绑定表单数据。SpringMVC表单标签最大的好处就是它支持数据绑定，当表单标签不需要绑定的数据时，应该使用普通Html标签。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="head" modelAttribute="user" methodParam="requestMethod">  <table>  <tr>  <td>Name:</td><td><**form:input** path="name"/></td>  </tr>  <tr>  <td>Age:</td><td><**form:input** path="age"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

## hidden

hidden标签会被渲染为一个type为hidden的普通Html input标签。用法跟input标签一样，也能绑定表单数据，只是它生成的是一个隐藏域。

## checkbox

checkbox标签会被渲染为一个type为checkbox的普通HTML input标签。checkbox标签也是支持绑定数据的。checkbox是一个复选框，有选中和不选中两种状态，checkbox标签的选中与否状态是根据它绑定的值来判断的。

### 绑定boolean数据

当checkbox绑定的是一个boolean数据的时候，那么checkbox的状态跟该boolean数据的状态是一样的，即true对应选中，false对应不选中。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>Male:</td><td><**form:checkbox** path="male"/></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

上面这段代码，假设在渲染该视图之前往ModelMap中添加了一个user属性，并且该user对象有一个类型为boolean的属性male，那么这个时候如果male属性为true，则Male那一栏的复选框将会被选中。

### 绑定列表数据

这里的列表数据包括数组、List和Set。下面将以List为例讲一下checkbox是如何根据绑定的列表数据来设定选中状态的。现在假设有一个类User，其有一个类型为List的属性roles，如下所示：

|  |
| --- |
| public class User {  private List<String> roles;  public List<String> getRoles() {  return roles;  }  public void setRoles(List<String> roles) {  this.roles = roles;  } } |

当需要展现该User是否拥有某一个Role的时候，可以使用checkbox标签来绑定roles数据进行展现。当checkbox标签的value在绑定的列表数据中存在的时候该checkbox将为选中状态。来看下面一段代码：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>Roles:</td>  <td>  <**form:checkbox** path="roles" value="role1"/>Role1<br/>  <**form:checkbox** path="roles" value="role2"/>Role2<br/>  <**form:checkbox** path="roles" value="role3"/>Role3  </td>  </tr>  </table> </**form:form**> |

就上面代码而言，当User拥有role1时对应的<form:checkbox path=*"roles"* value=*"role1"*/>就会为选中状态，也就是说roles列表中包含role1的时候该checkbox就会为选中状态。

### 绑定Object数据

checkbox还支持绑定数据类型为Object的数据，这种情况下Spring会拿所绑定对象数据的toString结果跟当前checkbox的value进行比较，如果能够进行匹配则该checkbox将为选中状态。看这样一个例子，有一个User类代码如下：

|  |
| --- |
| public class User {  private Blog blog;  public Blog getBlog() {  return blog;  }  public void setBlog(Blog blog) {  this.blog = blog;  } }  public class Blog {  public String toString() {  return "HelloWorld";  } } |

Blog类的toString方法已经被写死为“HelloWorld”了。这时往ModelMap中放一个user对象，而且给该user对象设定了一个blog属性，那么使用该ModelMap对象渲染如下视图代码时，checkbox标签将为选中状态。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>HelloWorld:</td>  <td>  <**form:checkbox** path="blog" value="HelloWorld"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

## checkboxes

相对于一个checkbox标签只能生成一个对应的复选框而言，一个checkboxes标签将根据其绑定的数据生成N个复选框。checkboxes绑定的数据可以是数组、集合和Map。在使用checkboxes时有两个属性是必须指定的，一个是path，另一个是items。Items表示当前要用来展现的项有哪些，而path所绑定的表单对象的属性表示当前表单对象拥有的项，即在items所展现的所有项中表单对象拥有的项会被设定为选中状态。先来看以下一段代码：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>Roles:</td>  <td>  <**form:checkboxes** path="roles" items="${roleList}"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

|  |
| --- |
| @RequestMapping(value="form", method=RequestMethod.GET) public String formTag(Map<String, Object> map) {  User user = new User();  List<String> roles = new ArrayList<String>();  roles.add("role1");  roles.add("role3");  user.setRoles(roles);  List<String> roleList = new ArrayList<String>();  roleList.add("role1");  roleList.add("role2");  roleList.add("role3");  map.put("user", user);  map.put("roleList", roleList);  return "formTag/form"; } |

以上代码ModelMap中的roleList对象有三个元素，分别是role1、role2和role3，而表单对象User的roles属性只拥有两个元素，分别是role1和role3,，所以访问该处理器方法返回如上所示的视图页面时，要展现的复选框项是roleList，也就是role1、role2和role3，而表单对象只拥有role1和role3，所以在页面进行渲染的时候会展示3个复选框项，但只有role1和role3会被设定为选中状态。

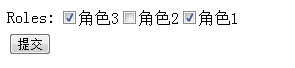


上面使用List作为展现复选框项的数据源，标签Label和它的值是一样的。使用Array和Set作为数据源也是这种情况。如果要让checkboxes呈现出来的Label和实际上的value不同，这时可以使用Map作为数据源。使用Map作为checkboxes的items属性的数据源时Key将作为真正的复选框的value，而Map的value将作为Label进行展示。当使用Map作为checkboxes的items属性的数据源时绑定的表单对象属性的类型可以是Array、集合和Map，这种情况就是判断items Map中是否含有对应的key来决定当前的复选框是否处于选中状态。

|  |
| --- |
| @RequestMapping(value="form", method=RequestMethod.GET) public String formTag(Map<String, Object> map) {  User user = new User();  List<String> roles = new ArrayList<String>();  roles.add("role1");  roles.add("role3");  user.setRoles(roles);  Map<String, String> roleMap = new HashMap<String, String>();  roleMap.put("role1", "角色1");  roleMap.put("role2", "角色2");  roleMap.put("role3", "角色3");  map.put("user", user);  map.put("roleMap", roleMap);  return "formTag/form"; } |

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>Roles:</td>  <td>  <**form:checkboxes** path="roles" items="${roleMap}"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

这时会呈现出3个复选框，而checkboxes绑定的表单对象user的roles属性是一个集合对象，其包含的两个元素都能在checkboxes的items数据源中找到对应的Key，所以以这两个元素为value的checkbox将处于选中状态。效果如下：



当使用Array或集合作为数据源，且里面的元素都是POJO时，还可以使用checkboxes标签的itemLabel和itemValue属性来表示使用数组或者集合中元素对象的哪一个属性作为需要呈现的单选框的label和value。

## radiobutton

radiobutton标签会被渲染为一个type为radio的普通HTML input标签。radiobutton标签也是可以绑定数据的。以下是一个radiobutton的简单应用示例：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>性别:</td>  <td>  <**form:radiobutton** path="sex" value="1"/>男  <**form:radiobutton** path="sex" value="0"/>女  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

在上面代码中的radiobutton标签都是绑定了表单对象user的sex属性，当sex为1的时候就代表性别为男，上面性别为男的那一行就会被选中，当sex为0的时候就代表性别为女，上面性别为女的那一行就会被选中。

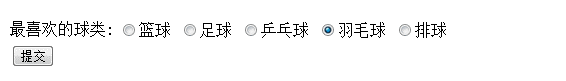
## radiobuttons

radiobuttons标签跟radiobutton标签的区别如同checkbox标签对checkboxes标签的区别。使用radiobuttons标签的时候将生成多个单选按钮。使用radiobuttons有两个属性也是必须指定的，一个是path属性，表示绑定的表单对象对应的属性，另一个是items属性，表示用于生成单选按钮的数据源。跟checkboxes一样，radiobuttons的items属性和path属性都可以是Array、集合或者是Map。假设user在篮球、足球、乒乓球、羽毛球和排球这5种运动中选择一种作为自己最喜欢的球类运动。处理器方法和返回的对应的视图代码如下：

|  |
| --- |
| @RequestMapping(value="form", method=RequestMethod.GET) public String formTag(Map<String, Object> map) {  User user = new User();  user.setFavoriteBall(4);//设置我最喜爱的球类运动是4羽毛球   Map<Integer, String> ballMap = new HashMap<Integer, String>();  ballMap.put(1, "篮球");  ballMap.put(2, "足球");  ballMap.put(3, "乒乓球");  ballMap.put(4, "羽毛球");  ballMap.put(5, "排球");  map.put("user", user);  map.put("ballMap", ballMap);  return "formTag/form"; } |

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的球类:</td>  <td>  <**form:radiobuttons** path="favoriteBall" items="${ballMap}"  delimiter="&nbsp;"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

 在上述代码中使用了radiobuttons的delimiter属性，该属性表示进行展示的radiobutton之间的分隔符。这里用的是一个空格。结果页面如下所示：



## password

password标签将会被渲染为一个type为password的普通HTML input标签。

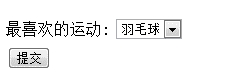
## select

select标签将会被渲染为一个普通的HTML select标签。这里还拿前面的user最喜欢的球类运动来做示例，有如下这样一个处理器方法和对应的视图页面：

|  |
| --- |
| @RequestMapping(value="form", method=RequestMethod.GET) public String formTag(Map<String, Object> map) {  User user = new User();  user.setFavoriteBall(4);//设置我最喜爱的球类运动是4羽毛球   Map<Integer, String> ballMap = new HashMap<Integer, String>();  ballMap.put(1, "篮球");  ballMap.put(2, "足球");  ballMap.put(3, "乒乓球");  ballMap.put(4, "羽毛球");  ballMap.put(5, "排球");  map.put("user", user);  map.put("ballMap", ballMap);  return "formTag/form"; } |

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的运动:</td>  <td>  <**form:select** path="favoriteBall" items="${ballMap}"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

这个时候会渲染出如下结果



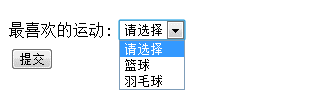
通过items属性给select标签指定了一个数据源，并且绑定了表单对象user的favoriteBall属性。Items属性是用于指定当前select的所有可选项的，对于select标签而言不是必须的，还可以手动的在select标签中间加上option标签来指定select可选的option。Select标签支持的items属性的数据类型可以是Array、Collection和Map，当数据类型为Array或Collection时且其中的元素为一个POJO时，可以通过属性itemLabel和itemValue来指定将用于呈现的option Label和Value，其他情况下Array和Collection数据源中的元素将既作为可选项option的value又作为它的Label。当items的数据类型为Map时，Map的key将作为可选项option的value，而Map的value将作为option的Label标签。

## option

option标签会被渲染为一个普通的HTML option标签。当一个SpringMVC select标签没有通过items属性指定自己的数据源的时候，就可以在select标签中通过普通HTML option标签或者SpringMVC option标签来指定可以选择的项。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的运动:</td>  <td>  <**form:select** path="favoriteBall">  <option>请选择</option>  <**form:option** value="1">篮球</**form:option**>  <option value="4">羽毛球</option>  </**form:select**>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

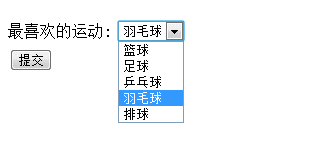
上面代码中没有指定select标签的数据源的，而是通过在select标签体里面指定普通HTML option标签和SpringMVC option标签来指定可选项。其渲染的效果如下：



同时通过items属性指定其数据源和使用option标签时，items优先级较高，当select标签指定了items属性的时候，它会忽略其标签体的内容，而使用items指定的内容来渲染出可选项。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的运动:</td>  <td>  <**form:select** path="favoriteBall" items="${ballMap}">  <option>请选择</option>  <**form:option** value="1">篮球</**form:option**>  <option value="4">羽毛球</option>  </**form:select**>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

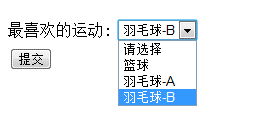
 从上述代码中我们可以看出来我们就是给select标签加了一个items属性，然后指定其数据源为当前pageContext的ballMap属性。此时，将渲染出如下效果：



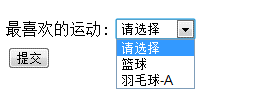
SpringMVC option标签和普通HTML option标签的区别就在于普通HTML option标签不具备数据绑定功能，而SpringMVC option标签具有数据绑定功能，它能把当前绑定的表单对象的属性对应的值对应的option置为选中状态。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的运动:</td>  <td>  <**form:select** path="favoriteBall">  <option>请选择</option>  <**form:option** value="1">篮球</**form:option**>  <option value="4">羽毛球-A</option>  <**form:option** value="4">羽毛球-B</**form:option**>  </**form:select**>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

在上面代码中定义一个select标签，其绑定了当前表单对象user的favoriteBall属性，而且没有给该select指定items数据源。在该select标签体中通过普通HTML option和SpringMVC option标签定义了两个value均为4的option元素，而且当前表单对象user的favoriteBall属性的值是4。上面代码渲染出的效果：



把上述代码中以SpringMVC option标签定义的option给删除，再看一下其渲染出的效果如下：

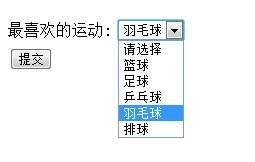


## options

使用options标签时需要指定其items属性，它会根据其items属性生成一系列的普通HTML option标签。这里的items属性的可取数据类型及其对应的渲染规则跟select的items属性是一样的。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>最喜欢的运动:</td>  <td>  <**form:select** path="favoriteBall">  <option>请选择</option>  <**form:options** items="${ballMap}"/>  </**form:select**>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

上面代码将渲染出如下效果：



## textarea

SpringMVC textarea标签将被渲染为普通HTML textarea标签。简单示例如下：

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table>  <tr>  <td>自我介绍:</td>  <td>  <**form:textarea** path="introduction" cols="20" rows="10"/>  </td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

## errors

SpringMVC errors标签是对应于SpringMVC的Errors对象的。它的作用就是用于展现Errors对象中包含的错误信息。利用errors标签来展现Errors时是通过errors标签的path属性来绑定一个错误信息的。可以通过path属性来展现两种类型的错误信息。

绑定所有的错误信息时path的值应该设置为“\*”，当前对象的某一个域的错误信息时path的值应为所需展现的域的名称。

看下面这样一个例子，定义了一个UserValidator对象，专门用来对User对象进行验证，其代码如下：

|  |
| --- |
| import org.springframework.validation.Errors; import org.springframework.validation.ValidationUtils; import org.springframework.validation.Validator;  public class UserValidator implements Validator {  @Override  public boolean supports(Class<?> clazz) {  return User.class.equals(clazz);  }  @Override  public void validate(Object target, Errors errors) {  ValidationUtils.rejectIfEmpty(errors, "name", null, "Name Is Empty");  ValidationUtils.rejectIfEmpty(errors, "username", null, "Username Is Empty.");  } } |

|  |
| --- |
| @Controller @RequestMapping("formTag") public class FormTagController {   @RequestMapping(value="form", method= RequestMethod.*GET*)  public String formTag(Map<String, Object> map) {  User user = new User();  map.put("user", user);  return "formTag/form";  }   @InitBinder  public void initBinder(DataBinder binder) {  binder.setValidator(new UserValidator());  }   @RequestMapping(value="form", method=RequestMethod.*POST*)  public String form(@Valid User user, Errors errors) {  if (errors.hasFieldErrors())  return "formTag/form";  return "formTag/submit";  } } |

上述控制器类中通过DataBinder对象给该类设定了一个用于验证的UserValidator，当请求该控制器时UserValidator将生效。

|  |
| --- |
| <**form:form** action="formTag/form.do" method="post" commandName="user">  <table border="1px" bordercolor="blue">  <tr align="center">  <td width="100">姓名:</td>  <td width="150"><**form:input** path="name"/></td>  </tr>  <tr align="center">  <td>用户名:</td>  <td><**form:input** path="username"/></td>  </tr>  <tr>  <td>所有错误信息:</td>  <td><**form:errors** path="\*"/></td>  </tr>  <tr>  <td>Name的错误信息:</td>  <td><**form:errors** path="name"/></td>  </tr>  <tr align="center">  <td colspan="2"><input type="submit" value="提交"/></td>  </tr>  </table> </**form:form**> |

提交表单时会往Errors中注入两个错误信息，展示的页面信息将如下所示：

