# 尚马教育 JAVA 课程

# springMVC基础

文档编号：C05

创建日期： 2017-07-07

最后修改日期：2021-01-21

版 本 号：V3.5

电子版文件名：尚马教育-第三阶段-5.springMVC基础.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Mybatis版本更新 | V2.0 |
| 2019-11-06 | 冯勇涛 | 课件格式以及课程深度加深 | V3.0 |
| 2021-01-21 | 冯勇涛 | 优化课程深度，优化案例，调整顺序 | V3.5 |

目录

[1. springMVC介绍 2](#_Toc27932_WPSOffice_Level1)

[1.1. 与struts2对比： 3](#_Toc23051_WPSOffice_Level2)

[1.2. springMVC的特点： 3](#_Toc10568_WPSOffice_Level2)

[2. 核心类与接口： 3](#_Toc23051_WPSOffice_Level1)

[2.1. DispatcherServlet前端控制器 3](#_Toc1317_WPSOffice_Level2)

[2.2. HandlerMapping处理器映射器 4](#_Toc7746_WPSOffice_Level2)

[2.3. HandlerAdapter处理器适配器 4](#_Toc28796_WPSOffice_Level2)

[2.4. Handler处理器对象 4](#_Toc30221_WPSOffice_Level2)

[2.5. ModelAndView模型与视图对象 4](#_Toc12751_WPSOffice_Level2)

[2.6. ViewResolver视图解析器对象 4](#_Toc5530_WPSOffice_Level2)

[3. springMVC请求处理流程图 5](#_Toc10568_WPSOffice_Level1)

[4. 体验springMVC 5](#_Toc1317_WPSOffice_Level1)

[4.1. 使用RequestMappingHandlerMapping处理器映射器 5](#_Toc28594_WPSOffice_Level2)

[4.1.1. Jar包依赖 5](#_Toc23051_WPSOffice_Level3)

[4.1.2. 编写配置文件以及代码 6](#_Toc10568_WPSOffice_Level3)

[4.1.3. 编写Handler处理器类 6](#_Toc1317_WPSOffice_Level3)

[4.1.4. Handler处理器中常用注解总结： 6](#_Toc7746_WPSOffice_Level3)

[5. 转发与重定向 7](#_Toc7746_WPSOffice_Level1)

[6. 数据乱码 7](#_Toc28796_WPSOffice_Level1)

[7. Servlet对象的获取 7](#_Toc30221_WPSOffice_Level1)

[8. Rest风格url 8](#_Toc12751_WPSOffice_Level1)

[8.1. 配置过滤器 8](#_Toc12530_WPSOffice_Level2)

[8.2. 常用注解 8](#_Toc7876_WPSOffice_Level2)

知识点：

|  |
| --- |
| 掌握SpringMVC的核心组件结构；  掌握SpringMVC环境的搭建方法；  掌握SpringMVC的处理流程； |

## springMVC介绍

Spring Web MVC是基于Servlet API构建的原始Web框架，并从一开始就包含在Spring Framework中。正式名称“ Spring Web MVC”来自其源模块[spring-webmvc](https://github.com/spring-projects/spring-framework/tree/master/spring-webmvc)的名称， 但它通常被称为“ Spring MVC”。简而言之，springMVC是对servlet的封装框架，避免了servlet的代码弊端，提高开发效率。

Spring 为展现层提供的基于 MVC 设计理念的优秀的Web 框架，是目前最主流的 MVC 框架之一：

Spring3.0 后全面超越 Struts2，成为最优秀的 MVC 框架。

Spring MVC 通过一套 MVC 注解，让 POJO 成为处理请求的控制器，而无须实现任何接口。

支持 REST 风格的 URL 请求。

采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性。

## springMVC执行原理图

|  |
| --- |
|  |

## 原理解析

1.客户端向服务器发送请求，请求被SpringMVC前端控制器DispatcherServlet捕获；

2.DispatcherServlet对请求URL进行解析，得到请求资源标识符URI，然后根据URI去调用HandlerMapping获得该URI对应的相关对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象返回；

3.DispatcherServlet根据获得的Handler处理器，选择一个合适的HandlerAdapter处理器适配器，该适配器对象执行Handler（获得HandlerAdapter后，将开始执行拦截器的前置拦截方法）；

4.提取Request中的请求数据，填充Handler入参，开始执行Handler。在填充Handler的入参过程中，SpringMVC将帮我们做一些很棒的工作：如通过HttpMessageConveter消息转化器对象生效：

* 将请求数据（如Json字符串）转换成对象，将响应对象转换为指定的数据格式。
* 数据转换：对请求数据进行格式转换。如String转换成Integer、Double等
* 数据格式化：对请求数据进行格式化。如将字符串转换成格式化数字或格式化日期等。
* 数据验证：验证数据的有效性（长度、格式等），验证结果存储BindingResult或Error。

1. Handler执行完成后，向DispatcherServlet返回一个ModelAndView对象；
2. 根据返回的ModelAndView，选择一个适合的ViewResolver（已注册到SpringMVC容器中的ViewResolver)返回给DispatcherServlet；
3. ViewResolver结合Model和View，来渲染视图；
4. 将渲染结果返回给客户端；
5. 响应结束。

## 核心类与接口

### DispatcherServlet前端控制器

|  |
| --- |
| 类图： |
| 作用：接收请求，响应结果，相当于转发器，是springMVC的中央处理器。 |

### HandlerMapping处理器映射器

|  |
| --- |
| 类图： |
| 作用：HandlerMapping负责查找Handler处理器对象与对应的拦截器对象 |

### HandlerAdapter处理器适配器

|  |
| --- |
| 类图： |
| 作用：找到Handler处理器对象后，由DispatcherServlet前端控制器调用HandlerAdapter来执行HandlerMethod,Handler处理器方法返回给适配器ModelAndView对象 |

### Handler处理器对象

|  |
| --- |
| 需要开发人员自己编写，不需要继承任何接口。 |
| 作用：相当于原来的Servlet控制器 |

### HttpMessageConverter消息转换器对象

|  |
| --- |
|  |
| 消息转换器主要两个作用，1.接收请求参数时，把请求参数转换成不同类型的对象（实体对象，Integer，Double等）；2.返回响应数据时，把返回数据转成指定的数据格式（json字符串） |

### ModelAndView模型与视图对象

|  |
| --- |
| 作用：是springmvc框架的一个底层对象，包括 Model和View，代表数据与视图部分。Handler执行完成后，返回给Adapter的是ModelAndView对象。Adapter再把该对象返回给DispatherServlet前端控制器。 |

### ViewResolver视图解析器对象

|  |
| --- |
| 作用：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)，  视图解析器向前端控制器返回View，DispatcherServlet负责渲染视图，将模型数据(在ModelAndView对象中)填充到request作用域，便于显示数据,最终响应用户。 |

## springMVC入门

### 创建web项目，导入springMVC依赖

|  |
| --- |
|  |

### 创建springmvc.xml配置

在类路径下创建springmvc.xml配置文件，该文件时spring风格的配置，在该文件中配置处理器映射器，处理器适配器，视图解析器对象等。内容如下：

|  |
| --- |
| *<!--开启ioc与di注解的识别-->* <**context:component-scan base-package="com.javasm"**></**context:component-scan**> *<!--处理器映射器对象：解析bean对象上的RequestMapping注解进行url映射-->* <**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"**></**bean**> *<!--处理器适配器对象：执行处理器方法-->* <**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**></**bean**> *<!--视图解析器对象-->* <**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  <**property name="prefix" value="/page/"**></**property**>  <**property name="suffix" value=".jsp"**></**property**> </**bean**> |

注意点1：必须配置<**context:component-scan base-package="com.javasm"**></**context:component-scan**>，用来识别基础的spring注解(Controller,Service,Repository,Component,Resource,Autowired等)。

注意点2：一般HandlerMapping与HandlerAdapter不手工配置，使用<mvc:annotation-driven></mvc:annotation-driven>标签替代，作用一致。如下：

|  |
| --- |
| <**mvc:annotation-driven**></**mvc:annotation-driven**> *<!--视图解析器对象-->* <**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  <**property name="prefix" value="/page/"**></**property**>  <**property name="suffix" value=".jsp"**></**property**> </**bean**> |

### 配置前端控制器

在Web.xml中配置DispatcherServlet，映射路径：/

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

### 创建page目录

在web目录按照视图解析器路径创建page文件夹，并创建main.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**>  <**head**>  <**title**>Title</**title**>  </**head**>  <**body**>  hello springMVC!  </**body**> </**html**> |

### 编写Handler处理器类

|  |
| --- |
| @Controller @RequestMapping(**"user"**) **public class** SysuserHandler {  @RequestMapping(**"main"**)  **public** String main(){  System.***out***.println(**"main方法"**);  **return "main"**;  } |

注意点：这里的返回值字符串main是jsp视图的名称。

### 部署项目启动tomcat

启动tomcat后，浏览器访问http://localhost:8080/user/main。

### 入门总结

1. @Controller  
    负责注册一个bean 到springMVC 上下文中。
2. @RequestMapping

注解到类与方法上，springMVC解析requestMapping注解，进行uri映射，一个uri映射到一个HandlerMethod。

1. 返回值

Main方法返回的”main”字符串，与视图解析器对象中的视图前缀：/page/，视图后缀：.jsp，组合完整视图路径:/page/main.jsp。

## Kv键值对数据处理

### 接收key=value数据

#### 定义login.jsp与main.jsp

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**>  这里是login.jsp  <**form action="user/login" method="post"**>  用户名:<**input type="text" name="uname"**>  密码：<**input type="password" name="upwd"**>  <**button type="submit"**>登录</**button**>  </**form**> </**body**> </**html**> |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**>  登录成功，进入main 页面 </**body**> </**html**> |

#### springmvc定义登录服务

|  |
| --- |
| @Controller @RequestMapping(**"user"**) **public class** SysuserHandler {  *//处理器方法添加形参，形参名与表单参数名一致。* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***)  **public** String dologin(String uname, String upwd) {  System.***out***.println(**"用户名："** + uname + **",密码："** + upwd);  **if** (**"admin"**.equals(uname) && **"upwd"**.equals(upwd)) {  Sysuser u = **new** Sysuser();  u.setUname(uname);  u.setUpwd(upwd);  *//登录成功，进入main页面* **return "main"**;  } **else** {  *//登录失败，进入login页面*  **return "login"**;  }  }  @RequestMapping(path = **"userlist"**,method = RequestMethod.***GET***) **public** String userlist(Sysuser u, @RequestParam(defaultValue =**"1"**) Integer pageNum,@RequestParam(defaultValue = **"10"**) Integer pageSize){  System.***out***.println(**"userlist方法"**+u+**"--"**+pageNum+**"--"**+pageSize);  **return "main"**; } } |

注意点：

注意点1：RequestMapping注解通过method属性限定请求方法；

注意点2：表单参数名一定要与方法形参名一致；

注意点3：登录服务的完整uri：/user/login；

注意点4：在page目录下要有对应的main.jsp与login.jsp视图。

注意点5：可以通过RequestParam注解为形参指定默认值。但该注解只能用在简单类型的形参（String，Double，Integer等），不能注解复杂对象类型。

#### 测试登录服务

启动tomcat，浏览器访问：[http://localhost:8080/login.jsp，](http://localhost:8080/login.jsp，输入用户名密码hou) 输入用户名与密码后点击登录按钮进行测试

### 返回视图数据

如果后端服务需要返回数据给客户端进行展示，需要使用Model或ModelAndView对象，以下分别展示两个对象的使用方法：

|  |
| --- |
| *//处理器方法添加形参，形参名与表单参数名一致。* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***) **public** String dologin(String uname, String upwd, Model model) {  System.***out***.println(**"用户名："** + uname + **",密码："** + upwd);  **if** (**"admin"**.equals(uname) && **"upwd"**.equals(upwd)) {  Sysuser u = **new** Sysuser();  u.setUname(uname);  u.setUpwd(upwd);  *//向model中加入数据等价于request.setAttribute* model.addAttribute(**"msg"**, **"suc"**);  model.addAttribute(**"loginuser"**, u);  *//登录成功，进入main页面* **return "main"**;  } **else** {  *//登录失败，进入login页面* model.addAttribute(**"msg"**, **"error"**);  **return "login"**;  } } |
| *//处理器方法添加形参，形参名与表单参数名一致。* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***) **public** ModelAndView dologin(String uname, String upwd) {  System.***out***.println(**"用户名："** + uname + **",密码："** + upwd);  ModelAndView modelAndView = **new** ModelAndView();   **if** (**"admin"**.equals(uname) && **"upwd"**.equals(upwd)) {  Sysuser u = **new** Sysuser();  u.setUname(uname);  u.setUpwd(upwd);  *//向model中加入数据等价于request.setAttribute* modelAndView.addObject(**"msg"**, **"suc"**);  modelAndView.addObject(**"loginuser"**, u);  *//登录成功，进入main页面*  modelAndView.setViewName(**"main"**);} **else** {  *//登录失败，进入login页面*  modelAndView.setViewName(**"login"**);modelAndView.addObject(**"msg"**, **"error"**);  }  **return** modelAndView; } |

### 注意点

#### 注意点1

一般后端服务接收表单参数，属于同一个实体类的数据，在方法的形参直接添加对应的形参对象，方法改造如下：

|  |
| --- |
| *//处理器方法添加形参，形参名与表单参数名一致。* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***) **public** ModelAndView dologin(Sysuser user) {  System.***out***.println(**"用户名："** + user.getUname() + **",密码："** + user.getUpwd());  ModelAndView modelAndView = **new** ModelAndView();   **if** (**"admin"**.equals(user.getUname()) && **"upwd"**.equals(user.getUpwd())) {  *//向model中加入数据等价于request.setAttribute* modelAndView.addObject(**"msg"**, **"suc"**);  modelAndView.addObject(**"loginuser"**,user);  *//登录成功，进入main页面* modelAndView.setViewName(**"main"**);  } **else** {  *//登录失败，进入login页面* modelAndView.setViewName(**"login"**);  modelAndView.addObject(**"msg"**, **"error"**);  }  **return** modelAndView; } |

#### 注意点2

Springmvc会对表单参数进行自动转型，不需要进行繁琐的String转Double等操作。

代码如下：

|  |
| --- |
| @GetMapping(**"test"**) **public** String testCast(String uname,Double uprice){  System.***out***.println(uname);  System.***out***.println(uprice);  **return "main"**; } |

浏览器输入url:http://localhost:8080/user/test?uname=fyt&uprice=100.12

#### 注意点3

后端服务方法的形参类型不允许出现基本类型（byte,short,int,long等），必须指定为包装类对象，因为包装类对象可以有默认值null，基本类型无法指定默认值。

## json数据处理

目前前端主流的异步提交请求组件ajax与axios。

* Ajax提交默认的ContentType的值为:application/x-www-form-urlencoded; charset=UTF-8   
  此格式为表单提交格式，数据为key1=value1&key2=value2的格式，以上章节后端服务获取数据方式适合前端使用ajax组件的情况。
* axios提交默认的ContentType的值为:application/json;charset=utf-8，

此格式为json提交格式，数据为{key1:value1,key2:value2}的格式，以上章节代码写法无法获取json格式数据。具体代码如下：

### 加入json转换器

Springmvc默认使用jackson组件作为json转换器。因此需要加入jackson相关的核心包。

|  |
| --- |
|  |

### 接收json数据

#### 定义响应枚举类与响应体对象

|  |
| --- |
| **package** com.javasm.sys.entity; **public enum** Status {  ***SUC***(20000,**"成功"**),  ***ERROR***(50000,**"失败"**),  ;  **private** Integer **code**;  **private** String **msg**;  Status(Integer code, String msg) {  **this**.**code** = code;  **this**.**msg** = msg;  }  **public** Integer getCode() {  **return code**;  }   **public void** setCode(Integer code) {  **this**.**code** = code;  }  **public** String getMsg() {  **return msg**;  }  **public void** setMsg(String msg) {  **this**.**msg** = msg;  } } |
| **package** com.javasm.sys.entity;  **import** com.fasterxml.jackson.annotation.JsonInclude;  *//null值不返回* @JsonInclude(JsonInclude.Include.***NON\_NULL***) **public class** ResultBean {  **private** Integer **code**;  **private** String **msg**;  **private** Object **data**;  **public** ResultBean(Status s, Object data) {  **this**.**code** = s.getCode();  **this**.**msg** = s.getMsg();  **this**.**data** = data;  }  **public** ResultBean(Status s) {  **this**.**code** = s.getCode();  **this**.**msg** = s.getMsg();  }  **public static** ResultBean data(Status s,Object data){  **return new** ResultBean(s,data);  }  **public static** ResultBean status(Status s){  **return new** ResultBean(s);  }  **public** Integer getCode() {  **return code**;  }  **public void** setCode(Integer code) {  **this**.**code** = code;  }  **public** String getMsg() {  **return msg**;  }  **public void** setMsg(String msg) {  **this**.**msg** = msg;  }  **public** Object getData() {  **return data**;  }  **public void** setData(Object data) {  **this**.**data** = data;  } } |

注意点：

注意点1：@JsonInclude(JsonInclude.Include.***NON\_NULL***)表示null值不返回前端；

#### 重新定义登录接口

由于异步提交，后端接口不需要进行视图的转发，仅需要返回json数据。因此不再执行视图解析器步骤，通过json消息转换器把方法的返回值转json字符串返回前端，springMVC三种方法支持返回数据转json字符串。

##### 使用ResponseBody注解

|  |
| --- |
| *//ResponseBody表示当前方法返回值通过json转换器，转json字符串返回前端* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***) @ResponseBody **public** ResultBean dologin(@RequestBody Sysuser user) {  System.***out***.println(**"收到数据："**+user);  **if**(**"admin"**.equals(user.getUname())&&**"admin"**.equals(user.getUpwd())){  user.setUid(100);  **return** ResultBean.*data*(Status.***SUC***,user);  }**else**{  **return** ResultBean.*status*(Status.***ERROR***);  } } |

注意点：RepsosneBody注解还可以注解到类上，表示该类中所有方法返回值通过json转换器转字符串。

##### 使用ResponseEntity作为返回值类型

|  |
| --- |
| @RequestMapping(path = **"login"**, method = RequestMethod.***POST***) **public** ResponseEntity dologin(@RequestBody Sysuser user) {  System.***out***.println(**"收到数据："**+user);  **if**(**"admin"**.equals(user.getUname())&&**"admin"**.equals(user.getUpwd())){  user.setUid(100);  ResultBean data = ResultBean.*data*(Status.***SUC***,user);  HttpHeaders headers = **new** HttpHeaders();  headers.add(**"testHeader"**,**"自定义响应头"**);  **return new** ResponseEntity(data,headers,HttpStatus.***OK***);  }**else**{  **return** ResponseEntity.*ok*(ResultBean.*status*(Status.***ERROR***));  } } |

注意点：ResponseEntity 相比ReposneBody更加强大，能够灵活的指定响应头，响应体，状态行。

##### 使用RestController注解控制层bean

RestController表示当前类中所有方法的返回值通过json转换器转字符串。

|  |
| --- |
| @RestController @RequestMapping(**"user"**) **public class** SysuserHandler {} |

注意点：用过RestController后，方法不再需要加ResponseBody注解。

#### 测试登录服务

以下所有接口均使用postman测试.

|  |
| --- |
|  |

#### 注意点

前端传递数据包含日期格式，后端接口可以使用String或Date接收，String接收则正常接收前端数据，Date接收需要使用JsonFormat注解指定日期格式

|  |
| --- |
| **public class** Sysuser {  **private** Integer **uid**;  **private** String **uname**;  **private** String **upwd**;  **private** Integer **uage**;  **private** String **uphone**;  **private** Double **uweight**;  @JsonFormat(pattern = **"yyyy-M**M-dd",timezone = "GMT+8")  **private** Date **udate**;  } |

## 服务的转发与重定向

视图解析器默认是做服务端跳转到指定视图。

forward:跳转到指定handlerMethod

redirect:重定向到指定handlerMethod。

处理器结果返回字符串以forward或redirect结尾

|  |
| --- |
| *//处理器方法添加形参，形参名与表单参数名一致。* @RequestMapping(path = **"login"**, method = RequestMethod.***POST***)  **public** String dologin(Sysuser user,Model model) {  System.***out***.println(**"用户名："** + user.getUname() + **",密码："** + user.getUpwd());   **if** (**"admin"**.equals(user.getUname()) && **"upwd"**.equals(user.getUpwd())) {  *//向model中加入数据等价于request.setAttribute* model.addAttribute(**"msg"**, **"suc"**);  model.addAttribute(**"loginuser"**,user);  *//登录成功，进入main页面 // return "forward:main";//视图名是forward:开头，表示转发到/user/main服务中* **return "redirect:main"**;*//视图名是redirect:开头，表示重定向到/user/main* } **else** {  *//登录失败，进入login页面* model.addAttribute(**"msg"**, **"error"**);  **return "login"**;  }  }  @RequestMapping(**"main"**) **public** String main(){  System.***out***.println(**"main方法"**);  **return "main"**; } |

## 数据乱码

### get提交

tomcat7以下需要在tomcat/conf/server.xml中的connector标签下配置属性URIEncoding="UTF-8"。

tomcat7及以上不需要配置，url默认支持中文数据,不会出现乱码。

### post提交

#### 提交kv键值对数据

Springmvc对键值对数据的处理会产生乱码问题，需要指定编码格式。

在web.xml中配置CharacterEncodingFilter过滤器进行编码统一过滤，配置如下：

|  |
| --- |
| <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value><!-- 与项目编码一致-->  </init-param>  <init-param>  <param-name>forceResponseEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

#### 提交json格式数据

Springmvc的提交的json数据处理通过jackson转换器，该转换器对象底层编码格式默认是UTF-8，默认支持中文，不会出现乱码问题。因此建议使用json格式进行数据传输。

## Servlet对象的获取

springMVC达到与servlet核心对象解耦，便于测试。

获取方式：以处理器方法形参方式进行传入，便于获取HttpServletRequest，HttpServletResponse，HttpSession。

|  |
| --- |
|  |

## Rest风格url

REST风格：Representational State Transfer表述性状态转移。

传统的查、改、删的URL与REST风格的增删改URL对比

|  |  |  |
| --- | --- | --- |
| 传统url | Rest风格URL | 提交方法 |
| 查询操作 /user/getUserById?id=12 | /user/12 | Get，数据跟在url后面 |
| 删除操作 /user/deleteUserById?id=12 | /user/12 | DELETE，数据一般跟在url后，也可以在请求体 |
| 添加操作 /user/addUser | /user | POST，数据在请求体 |
| 编辑操作 /user/updateUser?id=12 | /user/12 | PUT，数据再请求体或url后 |

请求方式：GET负责查询、POST负责添加、DELETE负责删除、PUT负责更新

他强调的是一个url资源可以对应多种视图

### 常用注解

GetMapping、PostMapping、PutMapping、DeleteMapping

四种注解对应四种请求方式。

|  |
| --- |
| @RestController  @RequestMapping("/goods")  **public** **class** GoodsHandler {  //@RequestMapping(path="{goodsId}",method=RequestMethod.GET)  @GetMapping("{goodsId}")  **public** String getGoods(@PathVariable("goodsId")**int** gid) {  System.***out***.println("查询商品："+gid);  **return** "查询商品："+gid;  }  @PostMapping  **public** **void** addGoods(Goods g) {  System.***out***.println("添加商品："+g);  }  @DeleteMapping("{goodsId}")  **public** **void** delGoodsById(@PathVariable("goodsId")**int** gid) {  System.***out***.println("删除商品："+gid);  }  @PutMapping("{goodsId}")  **public** **void** updateGoodsById(@PathVariable("goodsId")**int** gid,Goods g) {  System.***out***.println("更新商品："+g);  }  } |