

# SpringMVC5框架讲义

# SpringMVC 概述

## SpringMVC简介

SpringMVC 也叫Spring web mvc。是Spring 框架的一部分，是在Spring3.0 后发布的。

## SpringMVC的优点

### 基于MVC 架构

基于 MVC 架构，功能分工明确。解耦合。

### 容易理解，上手快，使用简单

就可以开发一个注解的 SpringMVC 项目，SpringMVC 也是轻量级的，jar 很小。不依赖的特定的接口和类。

### 作为Spring框架一部分，能够使用Spring的IOC和AOP

方便整合Strtus,MyBatis,Hiberate,JPA 等其他框架。

### SpringMVC 强化注解的使用

在Controller, Service, Dao 都可以使用注解。方便灵活。使用@Controller 创建处理器对象,@Service 创建业务对象，@Autowired 或者@Resource 在控制器类中注入 Service,在Service 类中注入 Dao。

## SpringMVC优化的方向

|  |
| --- |
|  |

## SpringMVC执行的流程

|  |
| --- |
|  |

执行流程说明：

1. 向服务器发送HTTP请求，请求被前端控制器 DispatcherServlet 捕获。
2. DispatcherServlet 根据<servlet-name>中的配置对请求的URL进行解析，得到请求资源标识符（URI）。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），最后以 HandlerExecutionChain 对象的形式返回。
3. DispatcherServlet 根据获得的Handler，选择一个合适的 HandlerAdapter。
4. 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

HttpMessageConveter：将请求消息（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：对请求消息进行数据格式化。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），验证结果存储到BindingResult或Error中。

1. Handler(Controller)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（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet。
3. ViewResolver 结合Model和View，来渲染视图。
4. 视图负责将渲染结果返回给客户端

## 基于注解的SpringMVC程序

所谓 SpringMVC 的注解式开发是指，在代码中通过对类与方法的注解，便可完成处理器在 springmvc 容器的注册。注解式开发是重点。

项目案例功能：用户提交一个请求，服务端处理器在接收到这个请求后，给出一条欢迎信息，在响应页面中显示该信息。

创建步骤：

1. 新建maven\_web项目

|  |
| --- |
|  |

1. 添加依赖

|  |
| --- |
| <dependency> <groupId>org.springframework</groupId> <artifactId>spring-webmvc</artifactId> <version>5.2.5.RELEASE</version> </dependency> <dependency> <groupId>javax.servlet</groupId> <artifactId>javax.servlet-api</artifactId> <version>3.1.0</version> </dependency> |

1. 删除web.xml文件重新添加，因为自动生成的web.xml文件版本太低了。

|  |
| --- |
|  |

1. 在web.xml文件中注册SpringMvc框架。因为web的请求都是由Servlet来进行处理的，而SpringMVC的核心处理器就是一个DispatcherServlet，它负责接收客户端的请求，并根据请求的路径分派给对应的action（控制器）进行处理，处理结束后依然由核心处理器DispatcherServlet进行响应返回。

中央调度器的全限定性类名在导入的 Jar 文件 spring-webmvc-5.2.5.RELEASE.jar 的第一个包org.springframework.web.servlet下可找到。

|  |
| --- |
| <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:springmvc.xml</param-value> </init-param> </servlet> <servlet-mapping> <servlet-name>springmvc</servlet-name> <url-pattern>\*.action</url-pattern> </servlet-mapping> |

<param-value>classpath:springmvc.xml</param-value>表示从类路径下加载SpringMVC的配置文件。

<url-pattern>指定拦截以.action结尾的请求，交给核心处理器DispatcherServlet处理。

1. 删除index.jsp页面，重新建index.jsp页面，因为自动生成的页面缺失指令设置。
2. 开发页面，发出请求。

|  |
| --- |
| <a href="${pageContext.request.contextPath}/zar/hello.action">访问action</a> |

其中：

/zar 是类上的注解路径

/hello 是方法上的注解路径

1. 在webapp目录上新添目录/admin。
2. 在/admin目录下新建main.jsp页面。用来进行服务器处理完毕后数据的回显。
3. 开发HelloSpringMvc.java-->控制器（相当于以前的servlet）。这是一个普通的类，不用继承和实现接口。类中的每个方法就是一个具体的action控制器。

类中的方法定义有规范：

1. 访问权限是public。
2. 方法名自定义。
3. 方法的参数可以有多个，任意类型，用来接收客户端提交上来的数据。
4. 方法的返回值任意。以返回String居多。

|  |
| --- |
| @Controller @RequestMapping("/zar") public class HelloSpringMvc { @RequestMapping("/hello") public String one(){ return "main"; } } |

@Controller：表示当前类为处理器，交给Spring容器去创建对象。

@RequestMapping：表示路径映射。该注解可以加在类上相当于包名，还可以加在方法上相当于action的名称，都是来指定映射路径的。

1. 完成springmvc.xml文件的配置。在工程的类路径即resources目录下创建 SpringMVC 的配置文件 springmvc.xml。该文件名可以任意命名。推荐使用springmvc.xml.

|  |
| --- |
| <context:component-scan base-package="com.bjpowernode.controller"></context:component-scan> <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"> <property name="prefix" value="/WEB-INF/jsp/"></property> <property name="suffix" value=".jsp"></property> </bean> |

SpringMVC框架为了避免对于请求资源路径与扩展名上的冗余，在视图解析器

InternalResouceViewResolver 中引入了请求的前辍与后辍。而action中只需给出要跳转页面的文件名即可，对于具体的文件路径与文件扩展名，视图解析器会自动完成拼接。

<context:component-scan>:用来进行包扫描，这里用于指定@Controller注解所在的包路径。

# SpringMVC注解式开发

## @RequestMapping定义请求规则

### 指定模块名称

通过@RequestMapping 注解可以定义处理器对于请求的映射规则。该注解可以注解在方法上，也可以注解在类上，但意义是不同的。value 属性值常以“/”开始。@RequestMapping 的 value 属性用于定义所匹配请求的 URI。

一个@Controller 所注解的类中，可以定义多个处理器方法。当然，不同的处理器方法所匹配的 URI 是不同的。这些不同的 URI 被指定在注解于方法之上的@RequestMapping 的value 属性中。但若这些请求具有相同的 URI 部分，则这些相同的 URI部分可以被抽取到注解在类之上的@RequestMapping 的 value 属性中。此时的这个 URI 表示模块（相当于包）的名称。URI 的请求是相对于 Web 的根目录。换个角度说，要访问处理器的指定方法，必须要在方法指定 URI 之前加上处理器类前定义的模块名称。

示例：

|  |
| --- |
| 提取后  @Controller @RequestMapping("/zar") public class HelloSpringMvc { //相当于一个控制器处理的方法 @RequestMapping("/hello") public String one() { return "main"; } @RequestMapping("/two") public String two() { return "main"; } //客户端的请求： // <form action="${pageContext.request.contextPath}/zar/hello.action">  // <form action="${pageContext.request.contextPath}/zar/two.action"> } |

### 对请求提交方式的定义

对于@RequestMapping，其有一个属性 method，用于对被注解方法所处理请求的提交

方式进行限制，即只有满足该 method 属性指定的提交方式的请求，才会执行该被注解方法。Method 属性的取值为 RequestMethod 枚举常量。常用的为 RequestMethod.GET 与

RequestMethod.POST，分别表示提交方式的匹配规则为 GET 与 POST 提交。

|  |
| --- |
| @RequestMapping(value = "/hello",method = RequestMethod.*POST*) public String one() { return "main"; } |

以上处理器方法只能处理 POST 方式提交的请求。

客户端浏览器常用的请求方式，及其提交方式有以下几种：

|  |
| --- |
|  |

也就是说，只要指定了处理器方法匹配的请求提交方式为 POST，则相当于指定了请求发送的方式：要么使用表单请求，要么使用 AJAX 请求。其它请求方式被禁用。

当然，若不指定 method 属性，则无论是 GET 还是 POST 提交方式，均可匹配。即对于请求的提交方式无要求。

#### post提交方式

|  |
| --- |
|  |

#### get提交方式

|  |
| --- |
|  |

## 五种数据提交的方式

前四种数据注入的方式，会自动进行类型转换。但无法自动转换日期类型。

### 单个数据注入

在方法中声明一个和表单提交的参数名称相同的参数，由框架按照名称直接注入。

|  |
| --- |
|  |

### 对象封装注入

在方法中声明一个自定义的实体类参数，框架调用实体类中相应的setter方法注入属性值，只要保证实体类中成员变量的名称与提交请求的name属性值一致即可。

|  |
| --- |
|  |

### 动态占位符提交（仅用于超链接）

使用框架提供的一个注解@PathVariable，将请求url中的值作为参数进行提取，只能是超链接。restful风格下的数据提取方式。restful是一种软件架构风格、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。

|  |
| --- |
|  |

### 请求参数名称与形参名称不一致

请求与形参中的名字不对应，可以使用

@RequestParam(value="name1",required=**true**)String namea来进行参数名称绑定。

|  |
| --- |
|  |

### 使用HttpServletRequest对象提取

在方法参数中声明一个request对象，使用request的getParameter()获取表单提交的数据，这样得到的数据还要手工进行数据类型的转换。

|  |
| --- |
| public String five(HttpServletRequest request){ int age=new Integer(request.getParameter("stuage")); String name=request.getParameter("stuname"); System.*out*.println(age+"\*\*\*\*\*\*\*\*\*"+name);  return "main"; } |

## 请求参数中文乱码解决

对于前面所接收的请求参数，若含有中文，则会出现中文乱码问题。Spring 对于请求参数中的中文乱码问题，给出了专门的字符集过滤器： spring-web-5.2.5.RELEASE.jar 的org.springframework.web.filter 包下的 CharacterEncodingFilter 类。

|  |
| --- |
|  |

### （1）解决方案

在 web.xml 中注册字符集过滤器，即可解决 Spring 的请求参数的中文乱码问题。不过，最好将该过滤器注册在其它过滤器之前。因为过滤器的执行是按照其注册顺序进行的。

|  |
| --- |
|  |

### 源码分析

|  |
| --- |
|  |

## 处理器方法的返回值

使用@Controller 注解的处理器的方法，其返回值常用的有四种类型：

* 第一种：ModelAndView
* 第二种：String
* 第三种：无返回值void
* 第四种：返回对象类型

### 2.4.1 返回 ModelAndView

若处理器方法处理完后，需要跳转到其它资源，且又要在跳转的资源间传递数据，此时处理器方法返回 ModelAndView 比较好。当然，若要返回 ModelAndView，则处理器方法中需要定义 ModelAndView 对象。在使用时，若该处理器方法只是进行跳转而不传递数据，或只是传递数据而并不向任何资源跳转（如对页面的 Ajax 异步响应），此时若返回 ModelAndView，则将总是有一部分多余：要么 Model 多余，要么 View 多余。即此时返回 ModelAndView 将不合适。较少使用。

### 2.4.2 返回 String

处理器方法返回的字符串可以指定逻辑视图名，通过视图解析器解析可以将其转换为物理视图地址。

|  |
| --- |
|  |

当然，也可以直接返回资源的物理视图名。不过，此时就不需要再在视图解析器中再配置前辍与后辍了。

|  |
| --- |
|  |

### 无返回值void

对于处理器方法返回 void 的应用场景，应用在AJAX 响应处理。若处理器对请求处理后，无需跳转到其它任何资源，此时可以让处理器方法返回 void。我们SSM整合案例中的分页使用的就是无返回值。代码见后面。

### 返回对象Object

处理器方法也可以返回 Object 对象。这个 Object 可以是 Integer，自定义对象，Map，List 等。但返回的对象不是作为逻辑视图出现的，而是作为直接在页面显示的数据出现的。返回对象，需要使用@ResponseBody 注解，将转换后的 JSON 数据放入到响应体中。Ajax请求多用于Object返回值类型。由于转换器底层使用了Jackson 转换方式将对象转换为JSON 数据，所以需要添加Jackson的相关依赖。

项目案例:使用ajax请求返回一个JSON结构的学生.

实现步骤:

A.在pom.xml文件中添加依赖

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.9.8</version>

</dependency>

B.添加jQuery的函数库,在webapp目录下,新建js目录,拷贝jquery-3.3.1.js到目录下

C.在页面添加jQuery的函数库的引用

<script src="js/jquery-3.3.1.js"></script>

D.发送ajax请求

function show() {

$.ajax({

url:"${pageContext.request.contextPath}/ajax.action",

type:"post",

dataType:"json",

success:function (stu) {

$("#oneStu").html(stu.name+"------"+stu.age);

}

});

}

E.开发action

@Controller

public class AjaxDemo {

@RequestMapping("/ajax")

@ResponseBody //此注解用来解析ajax请求

public Object ajax(){

Student stu = new Student("张三",22);

return stu;

}

}

F.在springmvc.xml文件中添加注解驱动

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

G.index.jsp页面

<a href="javascript:show()">ajax访问服务器,返回一个学生</a>

<br>

<div id="oneStu"></div>

## SpringMVC的四种跳转方式

默认的跳转是请求转发，直接跳转到jsp页面展示，还可以使用框架提供的关键字redirect:，进行一个重定向操作，包括重定向页面和重定向action，使用框架提供的关键字forward:，进行服务器内部转发操作，包括转发页面和转发action。当使用redirect:和forward:关键字时，视图解析器中前缀后缀的拼接就无效了。

页面部分：

|  |
| --- |
| <!--ctrl+d:复制当前行--> <a href="**${**pageContext.request.contextPath**}**/one.action">请求转发页面(默认)</a><br> <a href="**${**pageContext.request.contextPath**}**/two.action">请求转发action</a><br> <a href="**${**pageContext.request.contextPath**}**/three.action">重定向页面</a><br> <a href="**${**pageContext.request.contextPath**}**/four.action">重定向action</a><br> |

Controller部分：

|  |
| --- |
| @Controller public class JumpAction {  @RequestMapping("/one")  public String one(){  System.*out*.println("请求转发页面(默认)");  //以前的访问方式  //request.getRequestDispatcher("/admin/main.jsp").forward(request,response);  //观察地址栏的变化: http://localhost:8080/one.action  //return "main"; //默认的访问方式是自动拼接前缀和后缀进行跳转  return "forward:/fore/user.jsp";//只要使用了forward:就可以屏蔽前缀和后缀的拼接,自己手工构建返回的全部路径+.jsp  }  @RequestMapping("/two")  public String two(){  System.*out*.println("请求转发action");  //观察地址栏的变化: http://localhost:8080/two.action  return "forward:/other.action"; //不使用forward:,就会是这样的路径 /admin/other.action/.jsp  }  @RequestMapping("/three")  public String three(){  System.*out*.println("重定向页面");  //观察地址栏的变化 http://localhost:8080/admin/main.jsp  return "redirect:/admin/main.jsp";//只要使用了redirect:就可以屏蔽前缀和后缀的拼接  }  @RequestMapping("/four")  public String four(){  System.*out*.println("重定向action");  //观察地址栏的变化 http://localhost:8080/other.action  return "redirect:/other.action";//只要使用了redirect:就可以屏蔽前缀和后缀的拼接  } } |

## SpringMVC支持的默认参数类型

这些类型只要写在方法参数中就可以使用了。

**1）HttpServletRequest 对象**

**2）HttpServletResponse 对象**

**3）HttpSession 对象**

**4）Model/ModelMap 对象**

**5）Map<String,Object>对象**

示例：

|  |
| --- |
| @Controller public class ParamAction {  @RequestMapping("/param")  public String param(HttpServletRequest request,  HttpServletResponse response,  HttpSession session,  Model model,  ModelMap modelMap,  Map map){  //Map ,Model,ModelMap,request都使用请求作用域进行传值,  //所以必须使用请求转发的方式进行跳转,否则丢失数据  Student stu = new Student("张三",22);  request.setAttribute("requestStu",stu);  session.setAttribute("sessionStu",stu);  modelMap.addAttribute("modelMapStu",stu);  model.addAttribute("modelStu",stu);  map.put("mapStu",stu);  return "main"; //切记请求转发跳  // return "redirect:/admin/main.jsp";//会丢失数据  } } |

**注意Model,Map,ModelMap都使用的是request请求作用域,意味着只能是请求转发后,页面才可以得到值。**

## 日期处理

### 2.7.1 日期注入

日期类型不能自动注入到方法的参数中。需要单独做转换处理。使用@DateTimeFormat注解，需要在springmvc.xml文件中添加<mvc:annotation-driven/>标签。

### 在方法的参数上使用@DateTimeFormat注解

|  |
| --- |
| @RequestMapping("/submitone") public String submitdateone( @DateTimeFormat(pattern="yyyy-MM-dd")  Date mydate){  System.*out*.println(mydate);  return "dateShow"; } |

### （2）在类的成员setXXX()方法上使用@DateTimeFormat注解

|  |
| --- |
| @DateTimeFormat(pattern="yyyy-MM-dd") public void setDate(Date date) { this.date = date; } |

但这种解决方案要在每个使用日期类型的地方都去添加使用@DateTimeFormat注解，比较麻烦，我们可以使用@InitBinder注解来进行类中统一日期类型的处理。

### （3）@InitBinder注解解决类中日期问题

|  |
| --- |
| @InitBinder public void initBinder(WebDataBinder dataBinder) {  SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd"); dataBinder.registerCustomEditor(Date.class, new CustomDateEditor(sf, true)); } |

这样在类中出现的所有日期都可以进行转换了。

### 2.7.2 日期显示

#### （1）JSON中的日期显示

需要在类中的成员变量的getXXX方法上加注解.

|  |
| --- |
| @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss") public Date getDate() { return date; } |

#### JSP页面的日期显示

需要使用国际化标签，先添加依赖

|  |
| --- |
| <dependency> <groupId>jstl</groupId> <artifactId>jstl</artifactId> <version>1.2</version> </dependency> |

导入国际化的标签库

|  |
| --- |
| **<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>** |

再使用标签显示日期

|  |
| --- |
| <div id="stulistgood"> <**c:forEach** items="**${**list**}**" var="stu"> <p>**${**stu.name**}**-------**${**stu.age**}**-------<**fmt:formatDate** value="**${**stu.date**}**" pattern="yyyy-MM-dd"></**fmt:formatDate**></p> </**c:forEach**> </div> |

## <mvc:annotation-driven/>标签的使用

<mvc:annotation-driven/>会自动注册两个bean，分别为**DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter。**是springmvc为@controller分发请求所必须的。除了注册了这两个bean，还提供了很多支持。

1）支持使用ConversionService 实例对表单参数进行类型转换；

2）支持使用 @NumberFormat 、@DateTimeFormat；

3）注解完成数据类型的格式化；

4）支持使用 @RequestBody 和 @ResponseBody 注解；

5）静态资源的分流也使用这个标签;

## 资源在WEB-INF目录下

很多企业会将动态资源放在WEB-INF目录下，这样可以保证资源的安全性。在WEB-INF目录下的动态资源不可以直接访问，必须要通过请求转发的方式进行访问。这样避免了通过地址栏直接对资源的访问。重定向也无法访问动态资源。

项目案例：

|  |
| --- |
| 页面结构图： |

action:

|  |
| --- |
| @Controller public class ShowAction {  @RequestMapping("/showIndex")  public String showIndex(){  System.*out*.println("index.............");  return "index";  }  @RequestMapping("/showMain")  public String showMain(){  System.*out*.println("main.............");  return "main";  }  @RequestMapping("/showLogin")  public String showLogin(){  System.*out*.println("login.............");  return "login";  }  @RequestMapping("/login")  public String login(String name, String pwd, HttpServletRequest request){  if("admin".equals(name) && "123".equals(pwd)){  return "main";  }  request.setAttribute("msg","用户名或密码不正确!");  return "login";  } } |

运行结果：

|  |
| --- |
|  |

# SpringMVC拦截器

SpringMVC 中的 Interceptor 拦截器，它的主要作用是拦截指定的用户请求，并进行相应的预处理与后处理。其拦截的时间点在“处理器映射器根据用户提交的请求映射出了所要执行的处理器类，并且也找到了要执行该处理器类的处理器适配器，在处理器适配器执行处理器之前”。当然，在处理器映射器映射出所要执行的处理器类时，已经将拦截器与处理器组合为了一个处理器执行链，并返回给了中央调度器。

## 拦截器介绍

### 拦截器的应用场景

1、日志记录：记录请求信息的日志

2、权限检查，如登录检查

3、性能检测：检测方法的执行时间

### 拦截器的执行原理

|  |
| --- |
|  |

### 拦截器执行的时机

1)preHandle():在请求被处理之前进行操作

2)postHandle():在请求被处理之后,但结果还没有渲染前进行操作,可以改变响应结果

3)afterCompletion:所有的请求响应结束后执行善后工作,清理对象,关闭资源

### 拦截器实现的两种方式

1)继承HandlerInterceptorAdapter的父类

2)实现HandlerInterceptor接口,实现的接口,推荐使用实现接口的方式

## HandlerInterceptor接口分析

自定义拦截器，需要实现 HandlerInterceptor 接口。而该接口中含有三个方法：

#### preHandle

该方法在处理器方法执行之前执行。其返回值为 boolean，若为 true，则紧接着会执行处理器方法，且会将 afterCompletion()方法放入到一个专门的方法栈中等待执行。

#### postHandle

该方法在处理器方法执行之后执行。处理器方法若最终未被执行，则该方法不会执行。由于该方法是在处理器方法执行完后执行，且该方法参数中包含 ModelAndView，所以该方法可以修改处理器方法的处理结果数据，且可以修改跳转方向。

#### (3)afterCompletion

当preHandle()方法返回 true 时，会将该方法放到专门的方法栈中，等到对请求进行响应的所有工作完成之后才执行该方法。即该方法是在中央调度器渲染（数据填充）了响应页面之后执行的，此时对 ModelAndView 再操作也对响应无济于事。afterCompletion 最后执行的方法，清除资源，例如在 Controller 方法中加入数据等。

## 自定义拦截器实现权限验证

实现一个权限验证拦截器。

1. 修改web.xml文件中请求路径

|  |
| --- |
|  |

1. 将所有的页面放入WEB-INF目录下

|  |
| --- |
|  |

1. 开发登录action

|  |
| --- |
|  |

1. 开发拦截器

|  |
| --- |
|  |

1. 配置springmvc.xml文件

|  |
| --- |
| <!--注册拦截器--> <mvc:interceptors>  <mvc:interceptor>  <!--配置拦截的路径(哪些请求被拦截)-->  <mvc:mapping path="/\*\*"/>  <!--设置放行的请求-->  <mvc:exclude-mapping path="/login"></mvc:exclude-mapping>  <mvc:exclude-mapping path="/showLogin"></mvc:exclude-mapping>  <!--设置进行功能处理的拦截器类-->  <bean class="com.bjpowernode.interceptor.LoginInterceptor"></bean>  </mvc:interceptor> </mvc:interceptors> |

# SSM整合

## SSM整合后台功能

#### 新建Maven项目，添加依赖

|  |
| --- |
| <!-- 集中定义依赖版本号--> <properties> <junit.version>4.12</junit.version> <spring.version>5.1.2.RELEASE</spring.version> <mybatis.version>3.2.8</mybatis.version> <mybatis.spring.version>1.2.2</mybatis.spring.version> <mybatis.paginator.version>1.2.15</mybatis.paginator.version> <mysql.version>8.0.22</mysql.version> <slf4j.version>1.6.4</slf4j.version> <druid.version>1.0.9</druid.version> <pagehelper.version>5.1.2</pagehelper.version> <jstl.version>1.2</jstl.version> <servlet-api.version>3.0.1</servlet-api.version> <jsp-api.version>2.0</jsp-api.version> <jackson.version>2.9.6</jackson.version> </properties> <dependencies> <dependency> <groupId>org.aspectj</groupId> <artifactId>aspectjweaver</artifactId> <version>1.6.11</version> </dependency><dependency> <groupId>org.json</groupId> <artifactId>json</artifactId> <version>20140107</version> </dependency><!-- spring --> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-context</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-beans</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-webmvc</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-jdbc</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-aspects</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-jms</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-context-support</artifactId> <version>${spring.version}</version> </dependency> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-test</artifactId> <version>${spring.version}</version> </dependency> <!-- Mybatis --> <dependency> <groupId>org.mybatis</groupId> <artifactId>mybatis</artifactId> <version>${mybatis.version}</version> </dependency> <dependency> <groupId>org.mybatis</groupId> <artifactId>mybatis-spring</artifactId> <version>${mybatis.spring.version}</version> </dependency> <dependency> <groupId>com.github.miemiedev</groupId> <artifactId>mybatis-paginator</artifactId> <version>${mybatis.paginator.version}</version> </dependency> <dependency> <groupId>com.github.pagehelper</groupId> <artifactId>pagehelper</artifactId> <version>${pagehelper.version}</version> </dependency> <!-- MySql --> <dependency> <groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> <version>${mysql.version}</version> </dependency> <!-- 连接池--> <dependency> <groupId>com.alibaba</groupId> <artifactId>druid</artifactId> <version>${druid.version}</version> </dependency><!-- junit --> <dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.12</version> <scope>test</scope> </dependency> <!-- JSP相关--> <dependency> <groupId>jstl</groupId> <artifactId>jstl</artifactId> <version>${jstl.version}</version> </dependency> <dependency> <groupId>javax.servlet</groupId> <artifactId>javax.servlet-api</artifactId> <version>3.0.1</version> <scope>provided</scope> </dependency>  <dependency> <groupId>javax.servlet</groupId> <artifactId>jsp-api</artifactId> <scope>provided</scope> <version>${jsp-api.version}</version> </dependency> <!-- Jackson Json处理工具包--> <dependency> <groupId>com.fasterxml.jackson.core</groupId> <artifactId>jackson-databind</artifactId> <version>${jackson.version}</version> </dependency> <!-- fastjson--> <dependency> <groupId>com.alibaba</groupId> <artifactId>fastjson</artifactId> <version>1.2.28</version> </dependency> <!-- 文件上传用--> <dependency> <groupId>commons-io</groupId> <artifactId>commons-io</artifactId> <version>2.4</version> </dependency> <dependency> <groupId>commons-fileupload</groupId> <artifactId>commons-fileupload</artifactId> <version>1.3.1</version> </dependency> </dependencies>  <!-- 插件配置--> <build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <configuration> <source>1.8</source> <target>1.8</target> <encoding>UTF-8</encoding> </configuration> </plugin>  </plugins> <!--识别所有的配置文件--> <resources> <resource> <directory>src/main/java</directory> <includes> <include>\*\*/\*.properties</include> <include>\*\*/\*.xml</include> </includes> <filtering>false</filtering> </resource> <resource> <directory>src/main/resources</directory> <includes> <include>\*\*/\*.properties</include> <include>\*\*/\*.xml</include> </includes> <filtering>false</filtering> </resource> </resources> </build> |

#### 拷贝所有的配置文件到config，开发每个配置文件

applicationContext-dao.xml

|  |
| --- |
| <!-- 读取jdbc.properties属性文件 -->  <context:property-placeholder location="classpath:jdbc.properties"></context:property-placeholder> <!-- 创建数据源-->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="driverClassName" value="${jdbc.driver}"></property>  <property name="url" value="${jdbc.url}"></property>  <property name="username" value="${jdbc.username}"></property>  <property name="password" value="${jdbc.password}"></property>  </bean> <!-- 创建SqlSessionFactoryBean-->  <bean class="org.mybatis.spring.SqlSessionFactoryBean"> <!-- 配置数据源-->  <property name="dataSource" ref="dataSource"></property> <!-- 配置MyBatis的核心配置文件 -->  <property name="configLocation" value="classpath:SqlMapConfig.xml"></property> <!-- 配置实体类-->  <property name="typeAliasesPackage" value="com.bjpowernode.pojo"></property>  </bean> <!-- 创建mapper文件的扫描器-->  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="com.bjpowernode.mapper"></property>  </bean> |

applicationContext-service.xml

|  |
| --- |
| <!-- 设置业务逻辑层的包扫描器,目的是在指定的路径下,  使用@Service注解的类,Spring负责创建对象,并添加依赖 -->  <context:component-scan base-package="com.bjpowernode.service"></context:component-scan> <!-- 设置事务管理器-->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean> <!-- 添加事务的切面-->  <tx:advice id="myadvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="\*select\*" read-only="true"/>  <tx:method name="\*find\*" read-only="true"/>  <tx:method name="\*get\*" read-only="true"/>  <tx:method name="\*search\*" read-only="true"/>  <tx:method name="\*insert\*" propagation="REQUIRED"/>  <tx:method name="\*save\*" propagation="REQUIRED"/>  <tx:method name="\*add\*" propagation="REQUIRED"/>  <tx:method name="\*delete\*" propagation="REQUIRED"/>  <tx:method name="\*remove\*" propagation="REQUIRED"/>  <tx:method name="\*clear\*" propagation="REQUIRED"/>  <tx:method name="\*update\*" propagation="REQUIRED"/>  <tx:method name="\*modify\*" propagation="REQUIRED"/>  <tx:method name="\*change\*" propagation="REQUIRED"/>  <tx:method name="\*set\*" propagation="REQUIRED"/>  <tx:method name="\*" propagation="SUPPORTS"/>  </tx:attributes>  </tx:advice> <!-- 完成切面和切入点的织入-->  <aop:config>  <aop:pointcut id="mypointcut" expression="execution(\* com.bjpowernode.service.\*.\*(..))"/>  <aop:advisor advice-ref="myadvice" pointcut-ref="mypointcut"></aop:advisor>  </aop:config> |

Springmvc.xml

|  |
| --- |
| <!-- 设置包扫描器-->  <context:component-scan base-package="com.bjpowernode.controller"></context:component-scan> <!-- 设置视图解析器 /admin/ main .jsp -->  <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/admin/"></property>  <property name="suffix" value=".jsp"></property>  </bean> <!-- 设置文件上传核心组件-->  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  </bean> <!-- 设置注解驱动-->  <mvc:annotation-driven></mvc:annotation-driven>  <!-- 解决跨域问题-->  <mvc:cors>  <mvc:mapping path="/\*\*"  allowed-origins="\*"  allowed-methods="POST, GET, OPTIONS, DELETE, PUT"  allowed-headers="Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With"  allow-credentials="true" />  </mvc:cors> |

注意解决跨域问题.

什么是跨域？

浏览器从一个域名的网页去请求另一个域名的资源时，域名、端口、协议任一不同，都是跨域.

域名：

　主域名不同 http://www.baidu.com/index.html -->http://www.sina.com/test.js

　子域名不同 http://www.666.baidu.com/index.html -->http://www.555.baidu.com/test.js

　域名和域名ip http://www.baidu.com/index.html -->http://180.149.132.47/test.js

端口：

　http://www.baidu.com:8080/index.html–> http://www.baidu.com:8081/test.js

协议：

　http://www.baidu.com:8080/index.html–> https://www.baidu.com:8080/test.js

备注：

　1、端口和协议的不同，只能通过后台来解决

　2、localhost和127.0.0.1虽然都指向本机，但也属于跨域

另外一种解决方案是在控制器上添加@CrossOrigin注解.

或者自定义过滤器,进行跨域处理.

SqlMapConfig.xml

|  |
| --- |
|  |

#### 在web.xml文件中完成springmvc,spring两个框架的注册

|  |
| --- |
| <!-- 添加字符编码过滤器--> <filter>  <filter-name>encode</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>  <init-param>  <param-name>forceRequestEncoding</param-name>  <param-value>true</param-value>  </init-param>  <init-param>  <param-name>forceResponseEncoding</param-name>  <param-value>true</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>encode</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> <!-- 注册SpringMVC框架--> <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:springmvc.xml</param-value>  </init-param> </servlet> <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern> </servlet-mapping> <!-- 注册Spring框架--> <listener> <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext\_\*.xml</param-value> </context-param> |

#### 创建实体类对象User

|  |
| --- |
| public class User {  private String userId;//用户id  private String cardType;//证件类型  private String cardNo;//证件号码  private String userName;//用户姓名  private String userSex;//用户性别  private String userAge;//用户年龄  private String userRole;//用户角色  } |

#### 创建UserMapper接口和实现功能的UserMapper.xml

|  |
| --- |
| public interface UserMapper {  */\*\*  \* 分页查询 User  \** ***@param*** *startRows 起始页  \** ***@return*** *List*<*User*>  *\*/* List<User> queryUserPage(Integer startRows);  */\*\*  \* 分页查询 User 带条件  \** ***@param*** *userName  \** ***@param*** *userSex  \** ***@param*** *startRows  \** ***@return*** *\*/* List<User> selectUserPage(@Param("userName")String userName, @Param("userSex")String userSex, @Param("startRows")Integer startRows);  */\*\*  \* 查询 User 个数  \** ***@param*** *userName  \** ***@param*** *userSex  \** ***@return*** *\*/* Integer getRowCount(@Param("userName")String userName, @Param("userSex")String userSex);  */\*\*  \* 添加 User  \** ***@param*** *user  \** ***@return*** *返回码  \*/* Integer createUser(User user);  */\*\*  \* 根据 userId 删除用户  \** ***@return*** *返回码  \*/* Integer deleteUserById(String userId);  */\*\*  \* 根据 userId 批量删除用户  \** ***@param*** *userIds  \** ***@return*** *\*/* Integer deleteUserByIdList(@Param("list") List userIds);  */\*\*  \* 根据 userId 更新用户  \** ***@return*** *返回码  \*/* Integer updateUserById(User user); }  UserMapper.xml  <mapper namespace="com.bjpowernode.mapper.UserMapper" >  <resultMap id="BaseResultMap" type="com.bjpowernode.pojo.User" >  <id property="userId" column="user\_id" jdbcType="VARCHAR" /><!--用户id-->  <result property="cardType" column="card\_type" jdbcType="VARCHAR" /><!--证件类型-->  <result property="cardNo" column="card\_no" jdbcType="VARCHAR" /><!--证件号码-->  <result property="userName" column="user\_name" jdbcType="VARCHAR" /><!--用户姓名-->  <result property="userSex" column="user\_sex" jdbcType="VARCHAR" /><!--用户性别-->  <result property="userAge" column="user\_age" jdbcType="VARCHAR" /><!--用户年龄-->  <result property="userRole" column="user\_role" jdbcType="VARCHAR" /><!--用户角色-->  </resultMap>   <sql id="Base\_Column\_List" >  user\_id, card\_type, card\_no, user\_name, user\_sex, user\_age, user\_role  </sql>  <!--分页查询用户-->  <select id="queryUserPage" resultMap="BaseResultMap" parameterType="java.lang.Integer">  select  <include refid="Base\_Column\_List" />  from user  order by user\_id desc  limit #{startRows,jdbcType=INTEGER},5  </select>  <!--分页查询用户-->  <select id="selectUserPage" resultMap="BaseResultMap">  select  <include refid="Base\_Column\_List" />  from user  <where>  <if test="userName != null and userName !=''">  and user\_name like concat('%', #{userName}, '%')  </if>  <if test="userSex != null and userSex !=''">  and user\_sex = #{userSex}  </if>  </where>  order by user\_id desc  limit #{startRows,jdbcType=INTEGER},5  </select>  <!--查询用户个数-->  <select id="getRowCount" resultType="java.lang.Integer">  select *count*(\*) from user  <where>  <if test="userName != null and userName !=''">  and user\_name like concat('%', #{userName}, '%')  </if>  <if test="userSex != null and userSex !=''">  and user\_sex = #{userSex}  </if>  </where>  </select>  <!--添加用户信息-->  <insert id="createUser" parameterType="com.bjpowernode.pojo.User" >  insert into User(<include refid="Base\_Column\_List" />)  values(#{userId}, #{cardType}, #{cardNo}, #{userName}, #{userSex}, #{userAge}, #{userRole})  </insert>  <!--根据 user\_id 删除用户-->  <delete id="deleteUserById" parameterType="String" >  delete from user  where user\_id = #{userId,jdbcType=VARCHAR}  </delete>  <!--根据 user\_id 批量删除用户-->  <delete id="deleteUserByIdList" parameterType="java.util.List">  delete from user where user\_id in <foreach collection="list" item="item" index="index" open="(" close=")" separator=",">#{item,jdbcType=VARCHAR}</foreach>  </delete>  <!--根据 user\_id 更新用户信息-->  <update id="updateUserById" parameterType="com.bjpowernode.pojo.User" >  update user  <set >  <if test="cardNo != null" >  card\_no = #{cardNo,jdbcType=VARCHAR},  </if>  <if test="cardType != null" >  card\_type = #{cardType,jdbcType=VARCHAR},  </if>  <if test="userName != null" >  user\_name = #{userName,jdbcType=VARCHAR},  </if>  <if test="userSex != null" >  user\_sex = #{userSex,jdbcType=VARCHAR},  </if>  <if test="userAge != null" >  user\_age = #{userAge,jdbcType=VARCHAR},  </if>  <if test="userRole != null" >  user\_role = #{userRole,jdbcType=VARCHAR}  </if>  </set>  where 1 = 1  and user\_id = #{userId,jdbcType=VARCHAR}  </update> </mapper> |

#### 创建业务逻辑层UserService接口和实现类

|  |
| --- |
| public interface UserService {  */\*\*  \* 分页查询 User  \** ***@param*** *startRows 起始页  \** ***@return*** *List*<*User*>  *\*/* List<User> queryUserPage(Integer startRows);  */\*\*  \* 分页查询 User 带条件  \** ***@param*** *userName  \** ***@param*** *userSex  \** ***@param*** *startRows  \** ***@return*** *\*/* List<User> selectUserPage(String userName, String userSex, Integer startRows);  */\*\*  \* 查询 User 个数  \** ***@param*** *userName  \** ***@param*** *userSex  \** ***@return*** *\*/* Integer getRowCount(String userName,String userSex);  */\*\*  \* 添加 User  \** ***@param*** *user  \** ***@return*** *返回码  \*/* Integer createUser(User user);  */\*\*  \* 根据 userId 删除用户  \** ***@return*** *返回码  \*/* Integer deleteUserById(String userId);  */\*\*  \* 根据 userId 批量删除用户  \** ***@param*** *userIds  \** ***@return*** *\*/* Integer deleteUserByIdList( List userIds);  */\*\*  \* 根据 userId 更新用户  \** ***@return*** *返回码  \*/* Integer updateUserById(User user); }  @Service public class UserServiceImpl implements UserService {  @Autowired  private UserMapper userMapper;  @Override  public List<User> queryUserPage(Integer startRows) {  return userMapper.queryUserPage(startRows);  }  @Override  public List<User> selectUserPage(String userName, String userSex, Integer startRows) {  return userMapper.selectUserPage(userName, userSex, startRows);  }  @Override  public Integer getRowCount(String userName, String userSex) {  return userMapper.getRowCount(userName, userSex);  }  @Override  public Integer createUser(User user) {  return userMapper.createUser(user);  }  @Override  public Integer deleteUserById(String userId) {  return userMapper.deleteUserById(userId);  }  @Override  public Integer deleteUserByIdList(@Param("list") List userIds) {  return userMapper.deleteUserByIdList(userIds);  }  @Override  public Integer updateUserById(User user) {  return userMapper.updateUserById(user);  } } |

#### 创建测试类进行功能测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration(locations = {"classpath:applicationContext\_dao.xml","classpath:applicationContext\_service.xml"}) public class MyTest {  @Autowired  UserService userService;  @Test  public void testQueryUserPage(){  List<User> list = userService.selectUserPage(null,null,1);  list.forEach(user -> System.*out*.println(user));  }  @Test  public void testGetRowconunt(){  Integer num = userService.getRowCount("","");  System.*out*.println(num);  }  @Test  public void testDelete(){  Integer num = userService.deleteUserById("15968162087363060");  System.*out*.println(num);  } } |

#### 创建action进行分页显示,查询用户个数,删除处理

|  |
| --- |
| @CrossOrigin @RestController @RequestMapping("/user") public class UserController {  @Autowired  private UserService userService;   @RequestMapping("/queryUserPage")  public List<User> queryUserPage(Integer page) {  int pageNow = page == null ? 1 : page;  int pageSize = 5;  int startRows = pageSize\*(pageNow-1);  return userService.queryUserPage(startRows);  }  @RequestMapping("/selectUserPage")  public List<User> selectUserPage(String userName, String userSex, Integer page) {  int pageNow = page == null ? 1 : page;  int pageSize = 5;  int startRows = pageSize\*(pageNow-1);  return userService.selectUserPage(userName, userSex, startRows);  }  @RequestMapping("/getRowCount")  public Integer getRowCount(String userName, String userSex) {  return userService.getRowCount(userName, userSex);  }  @RequestMapping("/createUser")  public Integer createUser(User user) {  Random random = new Random();  Integer number = random.nextInt(9000) + 1000;  user.setUserId(System.*currentTimeMillis*() + String.*valueOf*(number));  return userService.createUser(user);  }  @RequestMapping("/deleteUserById")  public Integer deleteUserById(String userId) {  return userService.deleteUserById(userId);  }  @RequestMapping(value = "/deleteUserByIdList")  public Integer deleteUserByIdList(String userIdList) {  String userIdListSub = userIdList.substring(0, userIdList.length()-1); // String[] userIds = userIdList.split(",");  List userIds = new ArrayList();  for (String userIdStr: userIdListSub.split(",")){  userIds.add(userIdStr.trim());  }  return userService.deleteUserByIdList(userIds);  }  @RequestMapping("/updateUserById")  public Integer updateUserById(User user, Date date) {  return userService.updateUserById(user);  } } |

## Vue实现前台功能

Element UI官网地址:

https://element.eleme.cn/#/zh-CN/component/installation

Element UI是Vue使用的前端的框架,通过官网可以自行学习.

#### 安装node.js===> node-v12.16.2-x64.msi

查看版本编号

node -v

npm –v

node.js的安装是为了使当前的计算机使用vue的框架,预安装的工具.有点类似于运行java程序时必须安装JDK一样的道理.

#### 构建项目

使用命令行进入到当前要运行的vue的项目的目录下,运行以下命令进行项目搭建.

cd E:\idea\_workspace\vuedemo01 进入到当前项目的目录下

npm i element -ui -S 下载elementUI的框架

npm install //打包项目

npm install --save vue-axios //下载跨域访问组件axios

#### 为idea添加vue插件

|  |
| --- |
|  |

#### 配置启动项

|  |
| --- |
|  |

#### 项目结构

|  |
| --- |
|  |

项目结构解析:

build 项目构建(webpack)相关代码

config 配置目录，包括端口号等。我们初学可以使用默认的。

node\_modules npm 加载的项目依赖模块

src

这里是我们要开发的目录，基本上要做的事情都在这个目录里。里面包含了几个目录及文件：

assets: 放置一些图片，如logo等。

components: 目录里面放了一个组件文件，可以不用。

App.vue: 项目入口文件，我们也可以直接将组件写这里，而不使用 components 目录。

main.js: 项目的核心文件。

static 静态资源目录，如图片、字体等。

test 初始测试目录，可删除

.xxxx文件 这些是一些配置文件，包括语法配置，git配置等。

index.html 首页入口文件，你可以添加一些 meta 信息或统计代码啥的。

package.json 项目配置文件。

README.md 项目的说明文档，markdown 格式

#### UserHome.vue解读

UserHome.vue是所有功能实现的组件.与后台跨域访问,分页显示数据,并实现增加,按主键删除,批量删除,更新,多条件查询等功能.

A.首先确定钩子函数中的启动访问函数

created() {

this.handlePageChange(); ===>分页

this.getRowCount(); ===>计算总行数

}

B.分页函数解析

handlePageChange() {

//定义变量,封装将要提交的数据

let postData=this.qs.stringify({

page:this.currentPage,

userName:this.formInline.search1,

userSex:this.formInline.search2

});

this.$axios({ ==>发出跨域访问的请求,参考$.ajax();

method:'post', ==>请求提交的方式

url:'/api/user/selectUserPage',==>服务器的地址

data:postData //this.qs.stringify==>{"page":1,"userName":"","userSex":""} ==>提交的数据

}).then(response=>{ ==>成功后进入到这里

this.tableData=response.data; ==>数据绑定,返回的5个用户的json数据,一下子绑定给表格

}).catch(error=>{ ==>出错了进入到这里

console.log(error);

})

}

C.计算总行数函数分析

getRowCount() {

//创建变量,提取文本框和下拉列表框中的数据

let postData=this.qs.stringify({

userName:this.formInline.search1,

userSex:this.formInline.search2

});

this.$axios({ ==>发出跨域访问的请求,参考$.ajax();

method:'post', ==>请求提交的方式

url:'/api/user/getRowCount', ==>服务器的地址

data:postData ===>提交的数据

}).then(response=>{

this.total=response.data; ==>返回的总行数赋值给变量total

}).catch(error=>{

console.log(error);

})

},

D.按主键删除分析

//弹出提示框,让用户确定是否删除

this.$confirm('删除操作, 是否继续?', '提示', {

confirmButtonText: '确定',

cancelButtonText: '取消',

type: 'warning' ==>黄色的警告图标

}).then(() => { ==>用户点击确定后进入到这里

let postData = this.qs.stringify({ ==>封装成JSON数据格式

userId: row.userId, ==>将要提交的主键值

});

this.$axios({ ==>发出跨域访问的请求,参考$.ajax();

method: 'post', ==>请求提交的方式

url: '/api/user/deleteUserById',

data: postData //{"userId":15968162893439470}

}).then(response => { ==>跨域请求成功后进入这里

this.getRowCount(); ==>计算删除后的总行数,进行分页插件的页码变化

...

this.handlePageChange(); ==>删除后重新分页

this.$message({ ==>删除成功后弹框

type: 'success',

message: '删除成功!'

});

}).catch(error => {

console.log(error);

});

}).catch(() => { ===>用户点击取消后进入到这里

this.$message({

type: 'info',

message: '已取消删除'

});

});

},