springMVC笔记

第一部分：实现接口方式SpringMVC

1、需要的jar包

|  |
| --- |
|  |

使用注解需要额外添加AOP包

|  |
| --- |
|  |

2、基础配置式（非注解式）

1）web.xml

如果没有定义contextConfigLocation（springMVC配置文件位置）则会默认加载/WEB-INF/springmvc-servlet.xml的文件，如果该目录下没有创建这个配置文件，就会报错。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* id=*"WebApp\_ID"* version=*"3.0"*>    <!-- 配置前端控制器 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- 初始化时加载配置文件 -->  <init-param>  <!-- param-name固定 -->  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc-Config.xml</param-value>  </init-param>  <!-- 配置服务器启动后立即加载springmvc配置文件 -->  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  </web-app> |

2）springmvc配置文件

**注意！**4.0以前的版本需要配置处理器映射器，处理器适配器，视图解析器，4.0以后交由spring内部管理，可以无须再配，只要配置处理器映射请求就可以

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.3.xsd"*>    <!-- 配置处理器Handler,映射"/FirstController"请求 -->  <bean name=*"/FirstController"* class=*"com.zhangfd.controller.FirstController"*/>    <!-- 处理映射器，将Handler的name作为url进行查找 -->  <bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*/>    <!-- 处理器适配器，配置对处理器中的HandleRequest()方法调用 -->  <bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"*/>    <!-- 视图解析器 -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*/>  </beans> |

3）控制器类

**注意**! HttpServletRequest和HttpServletResponse需要添加jsp-api.jar，servlet.jar的依赖，可去tomcat的lib目录下找。

|  |
| --- |
| /\*\*  \* 控制器类  \* **@author** Administrator  \*  \*/  **public** **class** FirstController **implements** Controller {  @Override  **public** ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  //创建ModelAndView对象  ModelAndView mav = **new** ModelAndView();  //向模型对象中添加数据  mav.addObject("msg", "这是我的第一个springMVC文件");  //设置逻辑视图名  mav.setViewName("/WEB-INF/jsp/first.jsp");  //返回ModelAndView对象  **return** mav;  }  } |

4）jsp

用el表达式取出来

|  |
| --- |
|  |

第二部分：注解方式使用springMVC

1、@Controller

加在类上

2、@RequestMapping()

常用参数：

a.value

该属性为默认属性，可以加在类和方法上，括号里写value=”/地址”，当value为唯一属性时可以直接”/地址”，访问时地址为${xxx}/类上的地址/方法上的地址。

b.method

指定处理请求的方式，例如method=RequestMethod.GET表示只接受GET请求，如需支持多种请求，用{}写成数组形式，请求之间用逗号分开。

不常用参数：

name,params,headers,consumes,produces

3、组合注解

注：spring4.3以上可以使用

|  |
| --- |
|  |

4、请求处理方法的返回类型

1）常见的有：

a.ModelAndView

可以添加Model数据，并指定视图

b.Model

c.Map

d.View

e.String

可以跳转视图，但不携带数据

f.void

用于异步请求，只返回数据，不跳转视图

g.HttpEntity<?>或ResponseBody<?>

h.Callable<?>

i.DeferredResult<?>

2）返回类型为String：携带数据

由于ModelAndView类型不能实现数据和视图之间的解耦，所以在企业开发时，返回类型通常用String。用String时可以把数据通过方法model.addAttribute()放入Model里，再jsp中通过el表达式取出。

3）返回类型为String：重定向/转发

在返回的视图名前加一个redirect:或者forword:。

5、视图解析器（配置视图名前/后缀）

|  |
| --- |
| <!-- 视图解析器 -->  <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> |

第三部分：接收参数绑定

1、常见的默认参数类型

|  |
| --- |
|  |

2、绑定简单数据类型

直接在参数列表里写对应的参数名和类型。

3、@RequestParam注解

在前端请求参数名和后台控制器类方法中形参名不一致的时候可用。

属性：

|  |
| --- |
|  |

示例：

|  |
| --- |
|  |

4、绑定POJO类型

直接在方法参数列表里写一个POJO类型参数，springmvc会根据参数名将参数给POJO成员变量赋值。如果前端是用FORM表单提交，则表单项的name属性值必须与对应的POJO实体类成员变量一致。

5、绑定两个或者以上的POJO类型

1）POJO上加注解@Component

2）jsp

|  |
| --- |
|  |

3）Controller

|  |
| --- |
| @RequestMapping("/getTwoPOJO")  **public** String getTwoPOJO(Orders orders, User user) {  System.***out***.println(orders);  System.***out***.println(user);  **return** "success";  }  @InitBinder("user")  **public** **void** userBinder(WebDataBinder webDataBinder){  webDataBinder.setFieldDefaultPrefix("user.");  }  @InitBinder("orders")  **public** **void** userDetailBinder(WebDataBinder webDataBinder){  webDataBinder.setFieldDefaultPrefix("orders.");  } |

6、编码过滤器（处理前端传的POST中文乱码）

在web.xml里配置：

|  |
| --- |
| <!-- 配置编码过滤器 -->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

7、绑定包装POJO

当前端传过来的数据是包含两个不同的，相关联的类的数据时（例如传的订单编号属于订单类，用户id属于用户类，订单类与用户是多对一的关系），可以考虑让订单类持有用户对象，前端的表单项可以这么写：

|  |
| --- |
|  |

8、自定义数据绑定

有些数据不能直接绑定，例如时间，需要格式化后绑定的，就需要用**自定义转换器（Converter）**或**格式化（Formater）**来进行数据绑定绑定。两者都是用来转换格式的，不过Formater的源类型只能是**String**，Converter可以是**任意类型**。

1）Converter

定义一个实现Converter接口的转换类。

该接口是这个：

|  |
| --- |
| **import** org.springframework.core.convert.converter.Converter; |

示例：

a.日期转换类：

|  |
| --- |
| /\*\*  \* 自定义日期转换器  \* **@author** Administrator  \*  \*/  **public** **class** DateConvert **implements** Converter<String, Date> {  //定义日期格式  **private** String datePattern = "yyyy-MM-dd HH:mm:ss";    @Override  **public** Date convert(String source) {  //格式化日期  SimpleDateFormat sdf = **new** SimpleDateFormat(datePattern);    **try** {  **return** sdf.parse(source);  } **catch** (ParseException e) {  **throw** **new** IllegalArgumentException("无效的日期格式，请使用这种格式：" + datePattern);  }  }  } |

b.springmvc配置文件里加上：

**注意！需要加上mvc的命名空间**

|  |
| --- |
| <!-- 显示的装配自定义类型转化器 -->  <mvc:annotation-driven conversion-service=*"conversionService"*></mvc:annotation-driven>  <!-- 自定义类型转化器配置 -->  <bean id=*"conversionService"* class=*"org.springframework.context.support.ConversionServiceFactoryBean"*>  <property name=*"converters"*>  <set>  <bean class=*"com.zhangfd.covert.DateConvert"*></bean>  </set>  </property>  </bean> |

c.Controller类

|  |
| --- |
| @Controller  **public** **class** DateController {  /\*\*  \* 使用自定义类型数据绑定日期数据  \* **@param** date  \* **@return**  \*/  @RequestMapping("/customDate")  **public** String customDate(Date date) {  System.***out***.println("date=" + date);  **return** "success";  }  } |

URL:

|  |
| --- |
| http://localhost:8080/springmvc03/customDate?date=2018-04-12 16:26:00 |

结果：

|  |
| --- |
|  |

2）Formater

示例：

a.日期转换类：

|  |
| --- |
| /\*\*  \* 使用Formater日期转换器  \*  \* **@author** Administrator  \*  \*/  **public** **class** DateFormater **implements** Formatter<Date> {  // 定义日期格式  **private** String datePattern = "yyyy-MM-dd HH:mm:ss";  //声明SimpleDateFormat对象  **private** SimpleDateFormat simpleDateFormat;    @Override  **public** String print(Date date, Locale locale) {  **return** **new** SimpleDateFormat().format(date);  }  @Override  **public** Date parse(String source, Locale locale) **throws** ParseException {  simpleDateFormat = **new** SimpleDateFormat(datePattern);  **return** simpleDateFormat.parse(source);  }  } |

b.springmvc配置文件里加上：

|  |
| --- |
| <!-- 显示的装配自定义类型转化器 -->  <mvc:annotation-driven conversion-service=*"conversionService"*></mvc:annotation-driven>  <!-- 自定义类型格式化转换器配置 -->  <bean id=*"conversionService"*  class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>  <property name=*"formatters"*>  <set>  <bean class=*"com.zhangfd.covert.DateFormater"*></bean>  </set>  </property>  </bean> |

c.Controller类、URL:结果：

同Converter的。

9、复杂数据绑定

1）简单数据绑定

可以直接用数组接收相同name数据。例如：批量删除时用数组接收ids。

2）绑定复杂类型数据：list

定义一个VO接收。例如：修改时接收用户列表数据。

a.VO

|  |
| --- |
| /\*\*  \* 用户包装类  \* **@author** Administrator  \*  \*/  **public** **class** UserVo {  **private** List<User> users;  **public** List<User> getUsers() {  **return** users;  }  **public** **void** setUsers(List<User> users) {  **this**.users = users;  }    } |

b.Controller

|  |
| --- |
| /\*\*  \* 接受批量修改用户的方法  \* **@param** user  \* **@return**  \*/  @RequestMapping("/editUsers")  **public** String editUsers(UserVo userList) {  //将所有用户数据封装到集合中  List<User> users = userList.getUsers();  //循环输出所有的用户信息  **for** (User user : users) {  //如果接收的用户id不为空，则表示对该用户进行修改  **if** (user.getId() != **null**) {  System.***out***.println("修改了用户名为" + user.getUsername());  }  }  **return** "success";  } |

c.jsp

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath}*/editUsers"* method=*"post"*>  <table width=*"20%"* border=*"1"*>  <tr>  <td>选择</td>  <td>用户名</td>  </tr>  <tr>  <td><input name=*"users[0].id"* value=*"1"* type=*"checkbox"* /></td>  <td><input name=*"users[0].username"* value=*"tom"* type=*"text"* /></td>  </tr>  <tr>  <td><input name=*"users[1].id"* value=*"2"* type=*"checkbox"* /></td>  <td><input name=*"users[1].username"* value=*"jack"* type=*"text"* /></td>  </tr>  <tr>  <td><input name=*"users[2].id"* value=*"3"* type=*"checkbox"* /></td>  <td><input name=*"users[2].username"* value=*"luck"* type=*"text"* /></td>  </tr>  </table>  <input type=*"submit"* value=*"修改"*/>  </form> |

3）绑定复杂类型数据：Map

可以定义一个map的包装类，用tso那样接收。

第四部分：Json，restful

1、json绑定对象

json字符串转对象用fastjson的JSON.parseArray();和JSON.parseObject();方法也行，下面是在形参转换绑定。

1）需要jar包

|  |
| --- |
|  |

2）两个注解

|  |
| --- |
|  |

3)配置文件

在springmvc配置文件中加入

|  |
| --- |
| <!-- 配置注解驱动 -->  <mvc:annotation-driven/>  <!-- 配置静态资源的访问映射，此配置中的文件，被SpringMVC映射访问 -->  <mvc:resources location=*"/js/"* mapping=*"/js/\*\*"*></mvc:resources> |

4）jsp

|  |
| --- |
| **function** testJson() {  //获取输入用户名和密码  **var** username = $("#username").val();  **var** password = $("#password").val();  $.ajax({  url:"${pageContext.request.contextPath}/testJson",  type:"post",  //data表示发送的数据  data:JSON.stringify({username:username, password:password}),  //定义发送请求得数据格式为JSON字符串  contentType:"application/json;charset=UTF-8",  //定义回调响应的数据格式为JSON字符串，该属性可以省略  dataType:"json",  //成功响应的结果  success:**function**(data){  **if** (data !=**null**) {  alert("您输入的用户名为：" + data.username + "密码为：" + data.password)  }  }  });  } |

5)Controller

|  |
| --- |
| @RequestMapping("/testJson")  @ResponseBody  **public** User testJson(@RequestBody()User user) {  //打印接收的JSON格式数据  System.***out***.println(user);  //返回JSON格式的响应  **return** user;  } |

2、restful风格url

controller

|  |
| --- |
| /\*\*  \* 接收RESTful风格的请求，其接收方式为GET  \* **@param** user  \* **@return**  \*/  @RequestMapping(value="/user/{id}", method=RequestMethod.***GET***)  @ResponseBody  **public** User selectUser(@PathVariable("id")String id) {  //查看数据接收  System.***out***.println("id=" + id);  User user = **new** User();  //模拟根据id查询出到用户对象数据  **if** ("1234".equals(id)) {  user.setUsername("tom");  }  //返回JSON格式的数据  **return** user;  } |

第五部分：拦截器

1、配置

1）在springmvc配置文件中加入：

拦截器可以配置多个，但是节点需要遵守顺序

|  |
| --- |
|  |

2）使用方法

|  |
| --- |
|  |

3）接口方法

|  |
| --- |
|  |

2、拦截器的执行流程

1）拦截器的执行顺序

与配置文件里的配置的顺序相关

2）单个拦截器的执行流程

|  |
| --- |
| a.在访问控制器方法前会先执行拦截器的preHandle()方法==》  如果返回值为false则拦截该请求不执行controller方法，true则继续执行。  b.在执行完controller方法后执行拦截器的postHandle()方法==》  c.在前端控制器（DispatcherServlet）处理完请求后执行拦截器的afterCompletion ()方法 |

流程图：

|  |
| --- |
|  |

3）多个拦截器的执行流程

当有多个拦截器时，prehandle()方法会按配置顺序执行，而postHandle()和afterCompletion()会反序执行。

|  |
| --- |
|  |

3、拦截器案例

springmvc05

第六部分：文件上传/下载

1、需要jar包

|  |
| --- |
|  |

2、上传

1）jsp

|  |
| --- |
| 文件上传一般是用表单提交给服务器的，需要满足***三个条件***：  1）表单提交方式method必须为post；  2）表单的enctype属性必须为multipart/form-data；  3）提供<input type=”file” name=”filename”/>的文件上传输入框； |
| <form action=*"*${pageContext.request.contextPath}*/fileUpload"*  method=*"post"* enctype=*"multipart/form-data"* onsubmit="return check()">  上传人：<input id=*"name"* type=*"text"* name=*"name"* /><br />  <!-- multiple="multiple"HTML5多文件上传属性 -->  请选择文件：<input id=*"file"* type=*"file"* name=*"uploadfile"*  multiple=*"multiple"* /> <input type=*"submit"* value=*"上传"* />  </form> |

2）springmvc配置文件

|  |
| --- |
| <!-- 配置文件上传解析器multipartResolver(id必须是这个) -->  <bean id=*"multipartResolver"*  class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>  <!-- 设置请求编码格式 -->  <property name=*"defaultEncoding"* value=*"UTF-8"*></property>    </bean> |

3）controller

|  |
| --- |
| /\*\*  \* 执行文件上传  \*  \*/  @RequestMapping("/fileUpload")  **public** String handlerFormUpload(@RequestParam("name")String name,  @RequestParam("uploadfile")List<MultipartFile> uploadfile,  HttpServletRequest request) {  //判断所上传文件是否存在  **if** (!uploadfile.isEmpty() && uploadfile.size() > 0) {  //循环输出上传的文件  **for** (MultipartFile file : uploadfile) {  //获取上传文件的原始名称  String originalFilename = file.getOriginalFilename();  //设置上传文件的保存地址目录  String dirPath = request.getServletContext().getRealPath("/upload") + File.***separator***;  File filePath = **new** File(dirPath);  //如果保存的地址不存在，就先创建目录  **if** (!filePath.exists()) {  filePath.mkdirs();  }  //使用UUID重新命名上传的文件名称（上传人\_uuid\_原始名称）  String newFilename = name + "\_" + UUID.*randomUUID*() + "\_" + originalFilename;  //使用MultipartFile接口的方法完成文件上传到指定位置  **try** {  file.transferTo(**new** File(dirPath + newFilename));  System.***out***.println(dirPath + newFilename);  } **catch** (Exception e) {  e.printStackTrace();  **return** "error";  }  }  //跳转到成功页面  **return** "success";  } **else** {  **return** "error";  }  } |

MultipartFile接口方法

org.springframework.web.multipart.MultipartFile接口

|  |
| --- |
|  |

3、下载

1）jsp

|  |
| --- |
| <%@ page import=*"java.net.URLEncoder"*%>  <a href=*"*${pageContext.request.contextPath}*/download?filename=1.jpg"*>文件下载</a>  <a href=*"*${pageContext.request.contextPath}*/download?filename=*<%=URLEncoder.encode("壁纸.jpg", "UTF-8") %>*"*>中文文件下载</a> |

2）controller

|  |
| --- |
| @RequestMapping("/download")  **public** ResponseEntity<**byte**[]> fileDownload(HttpServletRequest request, String filename) **throws** Exception {  //指定要下载的文件所在路径  String path = request.getServletContext().getRealPath("/upload/");  //创建该文件对象  File file = **new** File(path + File.***separator*** + filename);  //对文件名编码，防止中文乱码  filename = **this**.getFilename(request, filename);  //设置响应头  HttpHeaders headers = **new** HttpHeaders();  //通知浏览器以下载的方式打开文件  headers.setContentDispositionFormData("attachment", filename);  //定义以流的形式下载返回文件数据  headers.setContentType(MediaType.***APPLICATION\_OCTET\_STREAM***);  //使用springMVC框架的ResponseEntity对象封装返回下载数据  **return** **new** ResponseEntity<**byte**[]>(FileUtils.*readFileToByteArray*(file), headers, HttpStatus.***OK***);  } |
| /\*\*  \* 根据流浪器的不同进行编码设置，返回编码后的文件名  \* **@throws** UnsupportedEncodingException  \*/  **private** String getFilename(HttpServletRequest request, String filename) **throws** UnsupportedEncodingException {  //IE不同版本User-Agent中文出现的关键词  String[] IEBrowserKeyWords = {"MSIE", "Trident", "Edge"};  //获取请求头代理信息  String userAgent = request.getHeader("User-Agent");  **for** (String keyWord : IEBrowserKeyWords) {  **if** (userAgent.contains(keyWord)) {  //IE内浏览器，统一为UTF-8编码显示  **return** URLEncoder.*encode*(filename, "UTF-8");  }  }  //火狐等其他浏览器统一为ISO-8859-1编码显示  **return** **new** String(filename.getBytes("UTF-8"), "ISO-8859-1");  } |