# SpringMVC学习笔记

# 第一章

## @RequestMapping注解

### @RequestMapping注解的功能

从注解名称上我们可以看到，@RequestMapping注解的作用就是将请求和处理请求的控制器方法关联起来，建立映射关系

### @RequestMapping注解的位置

@RequestMapping标识一个类：设置映射请求的请求路径的初始信息

@RequestMapping标识一个方法：设置请求请求路径的具体信息

### @RequestMapping注解的value属性

@RequestMapping注解的value属性通过请求的请求地址匹配请求映射

@RequestMapping注解的value属性是一个字符串类型的数组，表示该请求映射能够匹配多个请求地址所对应的请求

@RequestMapping注解的value属性必须设置，至少通过请求地址匹配请求映射

### @RequestMapping注解的method属性

@RequestMapping注解的method属性通过请求的请求方式（get或post）匹配请求映射

@RequestMapping注解的method属性是一个RequestMethod类型的数组，表示该请求映射能够匹配多种请求方式的请求

若当前请求的请求地址满足请求映射的value属性，但是请求方式不满足method属性，即浏览器报405错误：Request method‘POST’notsupported

### @RequestMapping注解的params属性（了解）

@RequestMapping注解的params属性通过请求的请求参数匹配请求映射

@RequestMapping注解的params属性是一个字符串类型的数组，可以通过四种表达式设置请求参数和请求映射的匹配关系

“param”:要求请求映射所匹配的请求必须携带param请求参数

“!param”：要求请求映射所匹配的请求必须不能携带param请求参数

“param=value”：要求请求映射所匹配的请求必须携带param请求参数且param=value

“param!=value”:要求请求映射所匹配的请求必须携带param请求参数但是param！=value

### @RequestMapping注解的headers属性（了解）

@RequestMapping注解的headers属性通过请求的请求头信息匹配请求映射

@RequestMapping注解的headers属性是一个字符串类型的数组，可以通过四种表达式设置请求头信息和请求映射的匹配关系

“header”：要求请求映射所匹配的请求必须携带header请求头信息

“！header”：要求请求·映射所匹配的请求必须不能携带header请求头信息

“header=value”：要求请求映射所匹配的请求必须携带header请求头信息且header=value

“header！=value”：要求请求映射所匹配的请求必须携带header请求头信息且header！=value

若当前请求满足@RequestMapping注解的value和method属性，但是不满足headers属性，此时页面显示404错误，即资源未找到

### springMVC支持ant风格的路径

？：表示任意的单个字节

\*：表示任意的0个或多个字符

\*\*：表示任意的一层或多层目录

注意：在使用\*\*时，只能使用/\*\*/xxx的方式

### SpringMVC支持路径中的占位符（重点）

原始方式：/deleteUser？id=1

Rest方式：/deleteUser/1

springMVC路径汇总的占位符常用于restful风格中，当请求路径中将某些数据通过路径的方式传输到服务器中，就可以将相应的@RequestMapping属性中通过占位符（xxx）表示传输的数据，在通过@PathVariable注解，将占位符所表示的数据赋值给控制器方法的形参

|  |
| --- |
| <a th:href="@{/testPath/1/admin}">测试@RequestMapping支持路径中的占位符---->/testPath</a>  @RequestMapping("/testPath/{id}/{username}") public String testPath(@PathVariable("id")Integer id,@PathVariable("username") String username){  System.*out*.println("id:"+id);  System.*out*.println("username："+username);  return "success"; |

# SpringMVC获取请求参数

## 通过servletAPI获取

|  |
| --- |
| @Controller public class ParamController {  @RequestMapping("/testServletAPI")  //形参位置的request表示当前请求  public String testServletAPI(HttpServletRequest request){  String username = request.getParameter("username");  String password = request.getParameter("password");  System.*out*.println("username:"+username+",password:"+password);  return "success";  } } |

## 通过控制器方法的形参获取请求参数

在控制器方法的形参位置，设置和请求参数同名的形参，当浏览器发送请求，匹配到请求映射时，在DispatcherServlet中就会将请求参数赋值给响应的形参

## @RequestHeader

@RequestHeader是将请求头信息和控制方法的形参创建映射关系

@RequestHeader注解一共有三个属性：value，required，defaultValue，方法同@RequestParam

## @CookieValue

@CookieValue是将cookie数据和控制器方法的形参创建映射关系

@CookieValue注解一共有三个属性：value，required，defaultValue，用法同@Request

## 通过POJO获取请求参数

可以在控制器方法的形参位置设置一个实体类类型的形参，此时若浏览器传输的请求参数的参数名和实体类中的属性名一致，那么请求参数就会为此属性赋值

|  |
| --- |
| @RequestMapping("/testBean") public String testBean(User user){  System.*out*.println(user);  return "success"; } |

## 当请求字符集出现乱码的时候

添加字符过滤器

|  |
| --- |
| <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>  <init-param>  <param-name>forceResponseEncoding</param-name>  <param-value>true</param-value>  </init-param> </filter> |

# 域对象共享数据

## 使用servletAPI向request域对象共享数据

|  |
| --- |
| @Controller public class ScopeController { // 使用servletAPI向request域对象共享数据  @RequestMapping("/testRequestByServletAPI")  public String testRequestByServletAPI(HttpServletRequest request){  request.setAttribute("testRequestScope","hello,servletAPI");  return "success";  } |

## 使用ModelAndView向request域对象共享数据

|  |
| --- |
| @RequestMapping("/testModelAndView") public ModelAndView testModelAndView(){  ModelAndView mav=new ModelAndView();  //处理模型数据，即向请求域共享数据  mav.addObject("testRequestScope","hello,ModelAndView");  //设置视图名称  mav.setViewName("success");  return mav; } |

## 使用Model向向request域对象共享数据

|  |
| --- |
| @RequestMapping("/testModel") public String testModel(Model model){  model.addAttribute("testRequestScope","hello,model");  return "success"; } |

## 使用map向request域对象共享数据

|  |
| --- |
| @RequestMapping("/testMap") public String testMap(Map<String,Object> map){  map.put("testRequestScope","hello,Map");  return "success"; } |

## 使用ModelMap向request域对象共享数据

|  |
| --- |
| @RequestMapping("/testModelMap") public String testModelMap(ModelMap modelMap){  modelMap.addAttribute("testRequestScope","hello,ModelMap");  return "success"; } |

## Model，ModelMap，Map的关系

Model，ModelMap，Map类型的参数本质上都是BindingAwareModelMap类型的

|  |
| --- |
| public interface Model {}  public class ModelMap extends LinkedHashMap<String, Object> {}  public class ExtendedModelMap extends ModelMap implements Model {}  public class BindingAwareModelMap extends ExtendedModelMap {} |

## 向session域共享数据

|  |
| --- |
| @RequestMapping("/testSession") public String testSession(HttpSession session){  session.setAttribute("testSessionScope","hello session");  return "success";  } |

## 向application域共享数据

# SpringMVC的视图

SpringMVC中的视图是VIew接口，视图的作用渲染数据，将模型中的数据显示给用户

SpringMVC视图的种类很多，默认有转发视图和重定向视图

当工程引入jstl的依赖，转发视图会自动转换为jstlView

若使用的视图技术为ThyMeleaf，在SpringMVC的配置文件汇总配置了ThyMeleaf的视图解析器，由此视图解析器解析之后所得到的是ThyMeleafVIew

## ThymeleafVIew

当控制器方法中所设置的视图名称没有任何前缀时，此时的视图名称会被SpringMVC配置文件中所配置的视图解析器解析，视图名称拼接视图前缀和视图后缀所得到的最终路径，会通过转发的方式实现跳转

|  |
| --- |
| @Controller public class ViewController {  @RequestMapping("/testThymeleafView")  public String testThyMeleafView(){  return "success";  } |

## 转发视图

SpringMVC中默认的转发视图是internalResourceView

SpringMVC中创建转发视图的情况：

当控制器方法中设置的视图名称以“forword:”为前缀时，创建internalResorceView视图，此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是将前缀“forword：”去掉，剩余部分作为最终路径通过转发的方式实现跳转

例如”forword:/”,”forword:/employee”

|  |
| --- |
| @RequestMapping("/testForward") public String testForward(){  return "forward:/testThymeleafView"; } |

## 重定向视图

SpringMVC中默认的重定向视图是RedirectView

当控制器方法中所设置的视图名称以“redirect”为前缀时，创建RedirectView视图，此时的视图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀“redirect”去掉，，剩余部分作为最终路径通过重定向的方式实现跳转

例如“redirect:/”，“redirect:/employee”

## 视图控制器view-controller

当控制器方法中，仅仅用来实现页面跳转，即只需要设置视图名称时，可以将处理器方法使用view-controller标签进行表示

|  |
| --- |
| <mvc:view-controller path="/" view-name="index"></mvc:view-controller> <!--开启MVC的注解驱动--> <mvc:annotation-driven/> |

# RESTFul

REST：表现层资源状态转移

**a>资源**

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个

可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、

数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端

应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是名词。一个

资源可以由一个或多个URI来标识。URI既是资源的名称，也是资源在Web上的地址。对某个资源感兴

趣的客户端应用，可以通过资源的URI与其进行交互。

**b>资源的表述**

资源的表述是一段对于资源在某个特定时刻的状态的描述。可以在客户端-服务器端之间转移（交

换）。资源的表述可以有多种格式，例如HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格

式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

**c>状态转移**

状态转移说的是：在客户端和服务器端之间转移（transfer）代表资源状态的表述。通过转移和操作资

源的表述，来间接实现操作资源的目的。

## RESTFul的实现

RESTFul风格提倡URL地址使用统一的风格设计，从前到后各个单词使用斜杠分开，不使用问号键值对方式携带请求参数，而是将要发送给服务器的数据作为URL地址的一部分，以保证整体风格的一致性

|  |  |  |
| --- | --- | --- |
| 操作 | 传统方式 | REST方式 |
| 查询操作 | getUserByid?id=1 | User/1-🡪get请求方式 |
| 保存操作 | saveUser | User—>post请求方式 |
| 删除操作 | deleteUser?id=1 | User/1🡪delete请求方式 |
| 更新操作 | updateUser | User—>put请求方式 |

## 路径拼接的两种方式

|  |
| --- |
| <td> <!--方式一-->  <!-- <a th:href="@{/employee/}+${employee.id}">delete</a>-->  <!--方式二,此时单引号里面的内容会被当做路径解析，${}里的内容会被当做数据解析-->  <a th:href="@{'/employee/'+${employee.id}}">delete</a>  <a href="">update</a> </td> |

# HttpMessageConverter

HttpMessageConverter，报文信息转换器，将请求报文转换为JAVA对象，或将java对象转换为响应报文，HttpMessageConverter提供了两个注解和两个类型：@RequestBody，RequestEntity，ReponseEntity

### @RequestBody

@RequestBody可以获取请求体，需要在控制器方法设置一个形参，使用@RequestBody进行标识，当前请求的请求体就会为当前注解所标识的形参赋值

|  |
| --- |
| @Controller public class HttpController {   @RequestMapping("/testRequestBody")  public String testRequestBody(@RequestBody String requestBody){  System.*out*.println("requestBody:"+requestBody);  return "success";  } } |

### RequestEntity

RequestEntity封装请求报文的一种类型，需要在控制器方法的形参中设置该类型的形参，当前请求的请求报文就会赋值给该形参，可以通过getHeaders（）获取请求头信息，通过getBody（）获取请求体信息

|  |
| --- |
| @RequestMapping("/testRequestEntity") public String testRequestEntity(RequestEntity<String> requestEntity){  //当前requestEntity表示整个请求报文的信息  System.*out*.println("请求头信息："+requestEntity.getHeaders());  System.*out*.println("请求体："+requestEntity.getBody());  return "success"; } |

### ReponseBody

@ResponseBody用于标识一个控制器方法，可以将该方法的返回值直接作为响应报文的响应体响应到浏览器

|  |
| --- |
| /\*原生方式\*/  @RequestMapping("/testResponse")  public void testRespoonse(HttpServletResponse response) throws IOException {  response.getWriter().print("hello,response");   }  \\方式二：通过添加@ResponseBody注解将返回值显示在浏览器上面  @RequestMapping("/testResponseBody") @ResponseBody() public String testResponseBody(){  return "success"; } |

### SpringMVC处理json

@ResponseBody处理json的步骤：

1. 导入jackson的依赖

|  |
| --- |
| 1. <dependency> 2. <groupId>com.fasterxml.jackson.core</groupId> 3. <artifactId>jackson-databind</artifactId> 4. <version>2.12.1</version> 5. </dependency> |

b>在SpringMVC的核心配置文件中开启mvc的注解驱动，此时在HandlerAdaptor中会自动装

配一个消息转换器：MappingJackson2HttpMessageConverter，可以将响应到浏览器的Java对象转换为Json格式的字符串

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

c>在处理器方法上使用@ResponseBody注解进行标识

d>将Java对象直接作为控制器方法的返回值返回，就会自动转换为Json格式的字符串浏览器的页面中展示的结果：

|  |
| --- |
| @RequestMapping("/testResponseUser")  @ResponseBody  public User testResponseUser(){  return new User(1001,"admin","123456",23,"男");  } |

{"id":1001,"username":"admin","password":"123456","age":23,"sex":"男"}

### springMVC处理ajax

@RestController注解

@RestController注解是springMVC提供的一个复合注解，标识在控制器的类上，就相当于为类添加了@Controller注解，并且为其中的每个方法添加了@ResponseBody注解

### ResponseEntity

ResponseEntity用于控制器方法的返回值类型，该控制器方法的返回值就是响应到浏览器的响应报文

# 拦截器

## 拦截器的配置

springMVC中的拦截器用于拦截控制器方法的执行

springMVC中的拦截器需要实现HandlerInterceptor或者继承HandlerInterceptorAdapter类

SpringMVC的拦截器必须在SpringMVC的配置文件中进行配置；

|  |
| --- |
| <!--配置拦截器-->  <mvc:interceptors> <!-- <bean class="com.lyy.mvc.interceptor.FirstInterceptor"></bean>--> <!-- <ref bean="firstInterceptor"></ref>--> <!-- //以上两个对所有的请求进行拦截-->  <!--下面的拦截器可以指定拦截路径-->  <mvc:interceptor>  <mvc:mapping path="/\*"/> <!-- <mvc:mapping path="/\*"/>-->  <mvc:exclude-mapping path="/"/>  <ref bean="firstInterceptor"></ref>  </mvc:interceptor>  </mvc:interceptors> |

## 拦截器的三个抽象方法

SpringMVC中的拦截器有三个抽象方法：

preHandle:控制器方法执行之前执行preHandle（），其boolean类型的返回值表示是否拦截或放行，返回true为放行，即调用控制器方法：返回false表示拦截，即不调用控制器方法

postHandle：控制器方法执行之后执行postHandle（）

afterComplation：处理完视图和模型数据，渲染视图完毕之后执行afterComplation（）

## 多个拦截器的执行顺序

1. 若每个拦截器的preHandle（）都返回true

此时多个拦截器的执行顺序和拦截器在SPringMVC的配置文件的配置顺序有关：

preHandle（）会按照配置的顺序执行，而postHandle（）和afterComplate（）会按照配置的反序执行

1. 若每个拦截器的preHandle（）返回false

preHandle（）返回false和它之前的拦截器的preHandle（）都会执行，postHandle（）都不会执行，返回false的拦截器之前的拦截器的afterComplate（）会执行

# 异常处理器

## 基于配置的异常处理

springMVC提供了一个处理控制器方法执行过程中所出现的异常的接口：HandlerExceptionResolver

HandlerExceptionResolver接口的实现类有：DefaultHandleExceptionResolve和SimpleMappingExceptionResolver

SpringMVC提供了自定义的异常处理器SimpleMappingExceptionResolver，使用方式：

|  |
| --- |
| <!--配置异常处理--> <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <property name="exceptionMappings">  <props>  <prop key="java.lang.ArithmeticException">error</prop>  </props>  </property>  <!--设置异常信息共享在请求域中的键-->  <property name="exceptionAttribute" value="ex"></property> </bean> |

# 注解配置SpringMVC

使用配置类和注解代替web.xml和SpringMVC配置文件的功能

创建初始化类，代替web.xml

|  |
| --- |
| package com.lyy.mvc.config;  import org.springframework.web.filter.CharacterEncodingFilter; import org.springframework.web.filter.HiddenHttpMethodFilter; import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;  import javax.servlet.Filter;  */\*\*  \** ***@program:*** *springMVC  \** ***@description:*** *\** ***@author:*** *ly  \** ***@create:*** *2021-10-29 14:17  \*\*/* //web工程的初始化类，用来代替web.xml public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer {  */\*\*  \* 指定spring的配置类  \** ***@return*** *\*/* @Override  protected Class<?>[] getRootConfigClasses() {  return new Class[]{SpringConfig.class};  }   */\*\*  \* 指定springMVC的配置类  \** ***@return*** *\*/* @Override  protected Class<?>[] getServletConfigClasses() {  return new Class[]{WebConfig.class};  }   */\*\*  \* 指定DispatcherServlet的映射规则，即url-pattern  \** ***@return*** *\*/* @Override  protected String[] getServletMappings() {  return new String[]{"/"};  }   */\*\*  \*注册过滤器  \** ***@return*** *\*/* @Override  protected Filter[] getServletFilters() {  CharacterEncodingFilter characterEncodingFilter=new CharacterEncodingFilter();  characterEncodingFilter.setEncoding("UTF-8");  characterEncodingFilter.setForceResponseEncoding(true);  HiddenHttpMethodFilter hiddenHttpMethodFilter=new HiddenHttpMethodFilter();  return new Filter[]{characterEncodingFilter,hiddenHttpMethodFilter};  } } |

# SpringMVC常用插件

## springMVC常用插件

* DispatcherServlet：前端控制器，不需要工程师开发，由框架提供

作用：统一处理请求和响应，整个流程控制的中心，由它调用其他组件处理用户的请求

* HandlerMapping：处理器映射器，不需要工程师开发，由框架提供  
  作用：根据请求的url、method等信息查找Handler，即控制器方法
* Handler：处理器，需要工程师开发  
  作用：在DispatcherServlet的控制下Handler对具体的用户请求进行处理
* HandlerAdapet：处理器适配器，不需要工程师开发，由框架提供  
  作用：通过HandlerAdapter对处理器（控制器方法）进行执行
* ViewResovler：视图解析器，不需要工程师开发，由框架提供  
  作用：进行视图解析，得到相应的视图，例如：ThymeleafView、InternalResourceView、RedirectView
* View：视图，不需要工程师开发，由框架或者视图提供

作用：将模型数据通过页面展示给用户

## DispatcherServlet初始化过程

DispatcherServlet本质上是一个Servlet，所以天然的遵循Servlet的生命周期。所以宏观上是Servlet生命周期来进行调度

## DispatcherServlet调用组件处理请求

1. processRequest()  
   FrameworkServlet重写HttpServlet中的service（） 和DoXxx（），这些方法中调用了processRequest（request，response）  
   所在类：org.springframework.web.servlet.FrameworkServlet