SpringMVC

1. SpringMVC用处

SpringMVC用于处理请求,回复页面和数据给客户端,SpringMVC是一个组件组成的框架,其中有以下组件

- 前端控制器
- 处理器映射器
- 处理器适配器
- 处理器
- 视图解析器
- 视图

2. SpringMVC的基本配置

1. 首先,SpringMVC接收请求是使用前端控制器,这个DispatchServlet本质上便是一个Servlet,配置在web.xml文件中,当tomcat一启动便创建该类

```
     <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.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>
```

2. 当创建前端控制器时,加载SpringMVC的配置文件,创建SpringMVC容器,在容器中配置视图解析器和开启注解扫描的配置,用于生成控制器类,在SpringMVC中开启<mvc:annotation-driven />后便自动创建出处理器适配器和处理器映射器

```
    <context:component-scan base-package="com.itheima" /

    <bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
<!--    开启SpringMVC的注解支持,这个相当于配置了视图解析器和处理器适配器-->
    <mvc:annotation-driven />
```

3. 在web.xml文件中配置过滤器,用于解决中文乱码的问题,将发送过来的字符先进行解码,再按照指定的编码方式对字符编码

```
<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>
```

3. SpringMVC配置问题

1. Controller层的对象为什么只能由SpringMVC容器来扫描生成呢?

为了将来的扩展,将来即使是改用了Struct2框架,而不是采用SpringMVC框架,也无需改动原有的配置

如果让Spring管理所有层的对象,请求到来将找不到对应的Handler,此时可以说明,处理器映射器只会在SpringMVC容器中寻找对应的Controller层的对象,并不会到Spring容器中寻找Controller层的对象

在让SpringMVC扫描包生成对象时,如果不加过滤器,将会导致Controller层所依赖的Service层的对象都是SpringMVC扫描生成的对象,而不是Spring中已经加上事务的对象,这样将会导致一些不可预见的问题的出现

1. 依赖注入的配置在Spring配置文件中已经加入,所以Controller层的对象能够成功注入Service层的对象。

4. SpringMVC处理请求的基本步骤

1. 首先前端控制器拦截到请求
2. 前端控制器调用处理器映射器,找到处理该请求对应的类
3. 调用处理器适配器生成适配对象
4. 调用真正的Handler对请求进行处理返回
5. 将字符串返回给前端控制器,前端控制器调用视图解析器获取对应的视图
6. 调用视图组件返回视图

5. URL与处理方法之间的对应

@RequestMapper注解

1. 用于处理URL请求与Handler之间的对应管理
2. 具有属性

```
@AliasFor("path")
String[] value() default {};

@AliasFor("value")
String[] path() default {};

RequestMethod[] method() default {};

String[] params() default {};

String[] headers() default {};
```

   - 其中value和path是相同的,这个是用于请求路径与方法之间的映射
   - method：设置请求方法,只处理使用特定请求方法的请求
   - params: 要求一定要有传入的参数,如果key和value都有的话,就都要保持一致
   - headers:设置了的话,请求中一定要有设置的请求头
3. 示例

```
    @RequestMapping(path = "/hello",method = {RequestMethod.POST,RequestMethod.GET},params = {"name=bbb"})
       public String sayHello(){
           System.out.println("开启处理请求");
           return "success";
       }
```

   上面方法对应的请求路径是/hello,处理请求方法为POST和GET的请求,必须传入一个参数名为那么,参数值为bbb的参数

6. 参数获取

概述

可以看出,在使用处理器适配器时,其实在内部已经使用request对象获取到了传递过来的参数,并且使用反射给方法中的参数通过反射或者是对应的注解,

将请求过来的参数赋值给方法的参数

1. 当请求参数与处理方法中的参数名相同时,SpringMVC将会对方法进行自动赋值,使用反射进行赋值

示例

1. 请求的编写

```
<a href="param/simpleParam?name=aaa&password=123">简单参数的封装示例</a>
```

2. Handler的编写

```
 @RequestMapping(path = "/simpleParam")
 public String simpleParam(String name,String password){
     System.out.println(name);
     System.out.println(password);
     return "success";
 }
```

3. 结果

```
能够接受到请求传递过来的参数
```

2. 传入参数为javaBean对象,在对其进行封装时,将会通过传入参数的名字使用反射来调用javaBean的setter方法进行赋值,如果传入参数名与bean对象中的参数名不一致,将会导致赋值失败

1. 请求的编写

```
<form  action="param/beanParam" method="post">
   姓名: <input type="text" name = "name"><br>
   密码: <input type="text" name= "password"><br>

   <input type="submit" value="提交"><br>
</form>
```

2. Handler的编写

```
@RequestMapping(path = "/beanParam")
public String beanParam(User user){
    System.out.println(user);
    return "success";
}
```

3. 结果

```
传入的参数能够自动注入对象中
```

3. 如果javaBean中存在一个引用类型,那么传入参数名称应该为 javaBean中 引用属性名.属性名 ,此时能够调用反射来赋值

1. 对应的javaBean

```
public class UserPlus {

    private String name;
    private String password;
    private Account account;
}
public class Account {

    private String idCard;
```

```
    private Float money;
}
```

2. 对应的请求方法,请求方法传入的参数名必须与接收的Bean对象中属性名保持一致

```
<form  action="param/beanParamPlus" method="post">
    姓名: <input type="text" name = "name"><br>
    密码: <input type="text" name= "password"><br>
    账号: <input type="text" name = "account.idCard"> <br>
    剩余金额: <input type="text" name="account.money">
    <input type="submit" value="提交"><br>
</form>
```

3. 处理器的编写,只需要编写JavaBean对象,便能实现参数的自动封装

```
@RequestMapping(path = "/beanParamPlus")
public String beanParamPlus(UserPlus user){
    System.out.println(user);
    return "success";
}
```

4. javaBean对象中存在list集合和map集合时

概述

```
当传入参数为List和map集合时,是无法直接使用List集合和map来接收参数的,SpringMVC最根本的获取参数的
方法便是request.getParameter("name"),设置也只能是同setter方法，那问题的关键是如何找到这个setter
方法？肯定是setName中的name和request中的name对应。这才能找到。你想，如果你单纯接收一个list参数，
list虽然有get和set方法，但是没有名字呀，只能根据数组下标来判断参数位置
```

1. list集合:接收参数的bean对象中List集合的属性名与传入的参数名保持一致

1. 对应的bean对象

```
public class UserList {

    private List<User> userList;
    public void setUserList(List<User> userList) {
        this.userList = userList;
    }
}
```

2. 对应的请求

```
 <form  action="param/listParam" method="post">
        姓名: <input type="text" name = "userList[0].name"><br>
        密码: <input type="text" name= "userList[0].password"><br>


        姓名: <input type="text" name = "userList[1].name"><br>
        密码: <input type="text" name= "userList[1].password"><br>
        <input type="submit" value="提交"><br>
</form>
```

3. Handler的方法

```
@RequestMapping(path = "/listParam")
public String listParam(UserList user){
    System.out.println(user);
    return "success";
}
```

传入的是List集合,此时需要一个对象来封装这个list集合,获取参数可以使用request.getParameter("name"),设置肯定是调用对应的setter方法进行赋值,

所以接收参数的bean对象中的集合的属性名需要与传入参数的名称相同

2. map集合:接收参数的bean对象的Map集合的属性名与传入的参数名保持一致

1. 对应的bean对象

```
public class UserMap {

    private Map<String,User> userMap;
}
```

2. 对应的请求

```
<form  action="param/mapParam" method="post">
    姓名: <input type="text" name = "userMap['aaa'].name"><br>
    密码: <input type="text" name = "userMap['aaa'].password"><br>


    姓名: <input type="text" name = "userMap['bbb'].name"><br>
    密码: <input type="text" name = "userMap['bbb'].password"><br>
    <input type="submit" value="提交"><br>
</form>
```

3.Handler方法

```
    @RequestMapping(path = "/mapParam")
    public String mapParam(UserMap map){
        System.out.println(map);
        return "success";
    }
```

7.Handler中的常用注解

1. RequestParams:用于解决传入参数与处理方法的参数不同的情况,并且使用这个注解时默是有对应的属性传入的

- 属性内容

```
 @AliasFor("name")
String value() default "";

@AliasFor("value")
String name() default "";

在方法处使用该属性表明将传入的哪个参数赋给方法参数

boolean required() default true;

声明传入的参数必须存在这个属性,不存在将会报错
```

- 示例

1. 请求方法

```
<form  action="annotation/testRequestParams" method="post">
    姓名: <input type="text" name = "userName"><br>
    密码: <input type="text" name = "password"><br>
    <input type="submit" value="提交"><br>
</form>
```

2. 处理方法

```
@RequestMapping(path = "/testRequestParams")
```

```
public String testRequestParams(@RequestParam("userName") String name, String password){
    System.out.println(name);
    System.out.println(password);
    return "success";
}
```

3. 结果

```
传入的参数名为userName最终能够赋值给属性名为name的属性
```

2. RequestBody:可以获得请求体

- 示例

1. 请求方法

```
<form  action="annotation/testRequestBody" method="post">
    姓名: <input type="text" name = "userName"><br>
    密码: <input type="text" name = "password"><br>
    <input type="submit" value="提交"><br>
</form>
```

2. 对应的Handler

```
@RequestMapping(path = "/testRequestBody")
public String testRequestBody(@RequestBody String name){
    System.out.println(name);
    return "success";
}
```

3. 结果:

```
打印出userName=fadf&password=aec5,获得了请求体
```

3. @RequestHeader可以获得请求头中的某个属性,并且赋值给方法的参数

- 示例

1. 请求方法

```
<form  action="annotation/testRequestHeader" method="post">
    姓名: <input type="text" name = "userName"><br>
    密码: <input type="text" name = "password"><br>
    <input type="submit" value="提交"><br>
</form>
```

2. 对应的处理方法

```
@RequestMapping(path = "/testRequestHeader")
public String testRequestHeader(@RequestHeader(name = "Accept") String name){
    System.out.println(name);
    return "success";
}
```

3. 结果

```
成功获取了请求头中的某个属性,上面程序打印出结果
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0
```

4. @CookieValue

1. 用处

获得cookie的值,可以使用value或者name来指明获取对应key的value,建立一个连接时,服务器会创建一个session,然后以cookie的形式返回一个值,用于唯一确定一个session,sessionid是一个会话的key，浏览器第一次访问服务器会在服务器端生成一个session，有一个sessionid和它对应。tomcat生成的sessionid叫做jsessionid。第一次访问便自动生成Cookie的值,并且自动返回

- 示例

    1. 请求方法

    ```
    <form  action="annotation/testCookieValue" method="post">
        姓名: <input type="text" name = "userName"><br>
        密码: <input type="text" name = "password"><br>
        <input type="submit" value="提交"><br>
    </form>
    ```

    2. 处理方法

    ```
    @RequestMapping(path = "/testCookieValue")
    public String testCookieValue(@CookieValue(name = "JSESSIONID") String name){
        System.out.println(name);
        return "success";
    }
    ```

    3. 结果

    ```
    返回对应的Cookie的值
    ```

5. @ModelAttribute

6. @SessionAtribute

8. Handle返回值类型

String类型

1. 概述

Handler方法返回值为String类型,String类型的值为对应视图的名称,调用视图解析器返回对应的视图 将一个对象存储到model对象中,model的底层实现是一个map,这个将会被存储到request域对象中,可以使用EL表达式从request域中取出值

2.处理器方法,这个处理器方法会返回success页面

```
@RequestMapping(path = "/testString")
public String testString(Model model){
    User user = new User();
    user.setName("aaa");
    user.setAge(20);
    model.addAttribute("user",user);
    return "success";
}
```

上面的model参数将会在处理器适配器时由框架为我们注入,使用model添加的属性会被加入到request对象中去

3.在jsp页面中使用EL表达式取值的话,取值顺序

```
application.setAttribute("name", "applicationName");//Context域，作用域最大
session.setAttribute("name", "sessionName");//session域，作用域大（当前session有效）
request.setAttribute("name", "requestName"); //request域，作用域小（对当前请求有效）
pageContext.setAttribute("name", "pageContextName");//page域，作用域最小（只对当前页面有效）
```

void类型,需要手动进行转发

1. 可以使用重定向:两次请求,重定向的路径要编写完整,加上项目路径与具体名称

```
 @RequestMapping(path = "/testVoid")
public void testVoid(Model model, HttpServletRequest request, HttpServletResponse response){
    User user = new User();
    user.setName("aaa");
    user.setAge(20);
    model.addAttribute("user",user);
    request.setAttribute("user",user);
    try {
        response.sendRedirect(request.getContextPath()+"/index.jsp");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2. 可以使用转发：一次请求,重定向不需要编写项目的路径和名称

```
 try {
        request.getRequestDispatcher("/WEB-INF/pages/success.jsp").forward(request,response);
    } catch (ServletException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
 }
```

3. 重定向与请求转发之间的区别

```
重定向和转发有一个重要的不同：当使用转发时，JSP容器将使用一个内部的方法来调用目标页面，新的页面继
续处理同一个请求，而浏览器将不会知道这个过程。 与之相反，重定向方式的含义是第一个页面通知浏览器发
送一个新的页面请求。因为，当你使用重定向时，浏览器中所显示的URL会变成新页面的URL， 而当使用转发时，
该URL会保持不变。重定向的速度比转发慢，因为浏览器还得发出一个新的请求。同时，由于重定向方式产生了一
个新的请求，所以经过一次重 定向后，request内的对象将无法使用。
转发和重定向的区别
不要仅仅为了把变量传到下一个页面而使用session作用域，那会无故增大变量的作用域，转发也许可以帮助你解决这个问题。
重定向：以前的request中存放的变量全部失效，并进入一个新的request作用域。
转发：以前的request中存放的变量不会失效，就像把两个页面拼到了一起。
```

4. 使用关键字进行请求转发和重定向

- return "forward:转发的路径" 固定写法,转发的路径需要编写完全,将不会使用视图解析器 请求转发
- return "redirect:项目路径+确切页面的坐标" 重定向,项目名可加可不加 无法访问WEB-INF内部的页面,因为请求无法直接访问里面的页面

ModelAndView

1. 概述:将会使用视图解析器进行解析,跟直接返回String类型差不多,方法参数有Model对象的情况

2. Handler

```
 @RequestMapping(path = "/testModelAndView")
public ModelAndView testModelAndView(){
    User user = new User();
    user.setName("aaa");
    user.setAge(20);
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("user",user);
    modelAndView.setViewName("success");
    return modelAndView;
}
```

9.Ajax发送请求和响应的方式

1. 一个Ajax的请求

```
$.ajax({
               type:"post",// get或者post
               url:"returnValue/testJson",// 请求的url地址
               data:'{"name":"你好","age":33}',//请求的参数
               contentType:"application/json;charset=UTF-8",//json写了jq会帮我们转换成数组或者对象 他已经用JSON.parse弄好了
               dataType:"json",
               timeout:3000,//3秒后提示错误
               success:function(data){ // 成功拿到结果放到这个函数 data就是拿到的结果
                   alert(data.name)
                   alert(data.age)
               },
               error:function(){//失败的函数
               },
               complete:function(){//不管成功还是失败 都会进这个函数
               }
           });
       });
```

2. 可以看到是用json字符串格式向后台传请求参数，那么后台需要采用@RequestBody来处理请求的json格式数据，将json数据转换为java对象，否则springmvc就不能解析导致传空参的结果

```
   @RequestMapping(path = "/testJson")
    public @ResponseBody User testJson(@RequestBody User user){
        System.out.println("你好");
        System.out.println(user.toString());
        user.setAge(30);
        user.setName("你好呀");
        return user;
    }
```

3. 一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。而 @ResponseBody就可以理解成将java的对象转换成json字符串的格式给前端解析（json数据格式解析比较简单） 如果加上@ResponseBody注解，就不会走视图解析器，不会返回页面，目前返回的json数据。如果不加，就走视图解析器，返回页面

4. 将json字符串转化为对象或者对象转化为字符串,此时需要加上注解@RequestBody或者@ResponseBody

5. 现在在开发项目有喜欢用key/value的格式传给后台。此方法比较常用。在ajax的data中写 'name=我&age=12'这个就不用就可以不用加@RequestBody
6. Spring默认的json协议解析由Jackson完成,所以需要导入Jackson的jar包

10. 取消对静态资源的拦截

在项目中引入jQuery 在webapp处创建js文件夹,将jquery.mn.js导入其中 在jsp页面的header中将该文件导入即可

在编写好之后,并不会有反应,因为前端控制器会对所有的请求进行拦截,此时对于静态资源,我们并不想对其进行拦截 此时可以在springmvc文件中配置取消对静态资源的拦截

<mvc:resources mapping="/js/**" location="/js/" />

这样就可以去掉对这个文件夹下资源的拦截

11. 拦截器设置:aop思想的体现

1. 编写一个实现了HandlerInterceptor接口的类,可以crtl+o查看接口中的方法

```
public class MyInterceptor implements HandlerInterceptor {
```

```
   public  boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
       System.out.println("preHandle方法执行了");
         return true;
    }

     public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView m
         System.out.println("方法执行了");
         request.getRequestDispatcher("/WEB-INF/pages/error.jsp").forward(request,response);
    }

     public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception

         System.out.println("最终执行的方法");
     }

}
```

2. 在SpringMVC的配置文件中编写过滤器,拦截对应的路径的请求,此时就能够对这个请求的处理方法进行拦截增强

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/interceptor/*"/>
        <bean class="com.itheima.controller.Interceptor.MyInterceptor"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

12. 过滤器的设置,解决请求的中文乱码问题

基本理念

```
   传入中文参数时,会出现中文乱码,可以设置过滤器,CharacterEncodingFilter来对编码方式进行设置,在web.xml中对其进行配置
   1)完全匹配 /servlet1
   2）目录匹配 /* 或者 /aaa/* ----匹配资源最多的
   3）扩展名匹配 *.abc *.jsp

   pageEncoding="UTF-8"的作用是设置JSP编译成Servlet时使用的编码。
   2、contentType="text/html;charset=UTF-8"的作用是指定对服务器响应进行重新编码的编码。
   3、request.setCharacterEncoding("UTF-8")的作用是设置对客户端请求进行重新编码的编码。
   4、response.setCharacterEncoding("UTF-8")的作用是指定对服务器响应进行重新编码的编码。
   response.setCharacterEncoding("UTF-8")的作用是指定对服务器响应进行重新编码的编码。
   同时，浏览器也是根据这个参数来对其接收到的数据进行重新编码（或者称为解码）。
   所以在无论你在JSP中设置response.setCharacterEncoding ("UTF-8")或者response.setCharacterEncoding("GBK"),
   浏览器均能正确显示中文（前提是你发送到浏览器的数据编码是正确的，比如正确设置了pageEncoding参数等）。

   过滤器的原理,在tomcat服务器之前设置一个过滤器,对应请求到来先经过这个过滤器
```

具体配置

```
   <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>
```