SpringMVC学习笔记

SpringMVC学习笔记

一、SpringMVC简介

1.1 三层架构及MVC设计模型

1.1.1 三层架构

> 我们的开发架构一般都是基于两种形式，- 种是C/S架构，也就是客户端/服务器，另一种是B/S 架构，也就是浏览器服务器。在JavaEE开发中，几乎全都是基于B/S架构的开发。那么在B/S架构中，系统标准的三层架构包括:**表现层、业务层、持久层**。三层架构在我们的实际开发中使用的非常多 三层架构中，每一层各司其职，接下来说说每层都负责哪些方面:

> 表现层: 也就是我们常说的web层。它负责接收客户端请求，向客户端响应结果，通常客户端使用http协议请求web层，web需要接收http请求，完成http响应。 表现层包括展示层和控制层:控制层负责接收请求，展示层负责结果的展示。 表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。 表现层的设计一般都使用MVC模型。(MVC 是表现层的设计模型，和其他层没有关系)

> 业务层: 也就是我们常说的service层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web层依赖业务层，但是业务层不依赖web层。 业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务-致性。(也就是我们说的，事务应该放到业务层来控制)

> 持久层: 也就是我们是常说的dao层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进 行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库 中。通俗的讲，持久层就是和数据库交互，对数据库表进行增删改查的。

1.1.2 MVC设计模型

> MVC全名是Model View Controller, 是模型(model)一视图(view)一控制器(controller)的缩写，是一种用于设计创建Web应用程序表现层的模式。MVC中每个部分各司其职:

> M: model 模型 javebean 一般用于封装数据

> V: view 视图 html/jsp等 一般用于展示数据

> C: controller 控制器 servlet等 用于前后端交互,前端传过来数据,经过一系列处理(数据验证,调用业务层方法等)将数据返回给前端

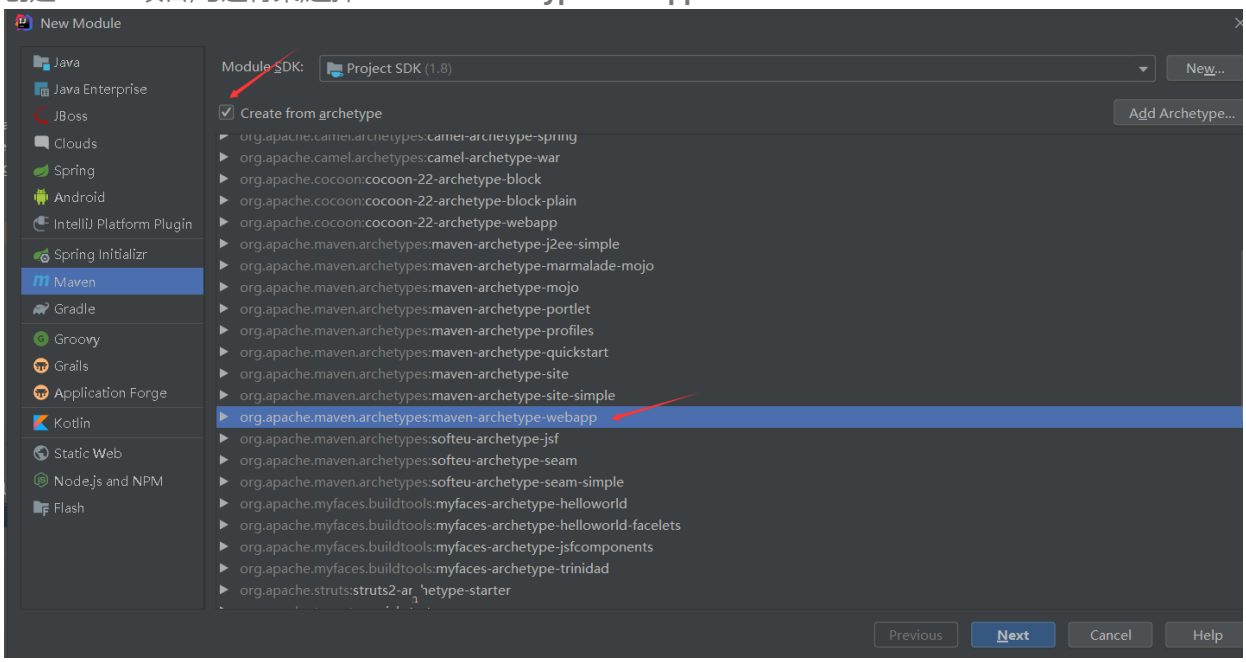
1.2 介绍SpringMVC框架

> - SpringMVC是一种基于Java 的实现MVC设计模型的请求驱动类型的轻量级Web框架，属于SpringFrameWork的后续产品，已经融合在Spring Web Flow 里面。Spring框架提供了构建Web应用程序的全功能MVC模块。使用Spring可插入的MVC架构，从而在使用Spring进行WEB开发时，可以选择使用Spring的Spring MVC框架或集成其他MVC开发框架，如Strutsl (现在一般不用)，Struts2 等。
> - SpringMVC已经成为目前最主流的MVC框架之一，并且随着Spring3.0的发布，全面超越Struts2, 成为最优秀的MVC框架。
> - 它通过一套注解，让一个简单的Java类成为处理请求的控制器，而无须实现任何接口。同时它还支持RESTful编程风格的请求。

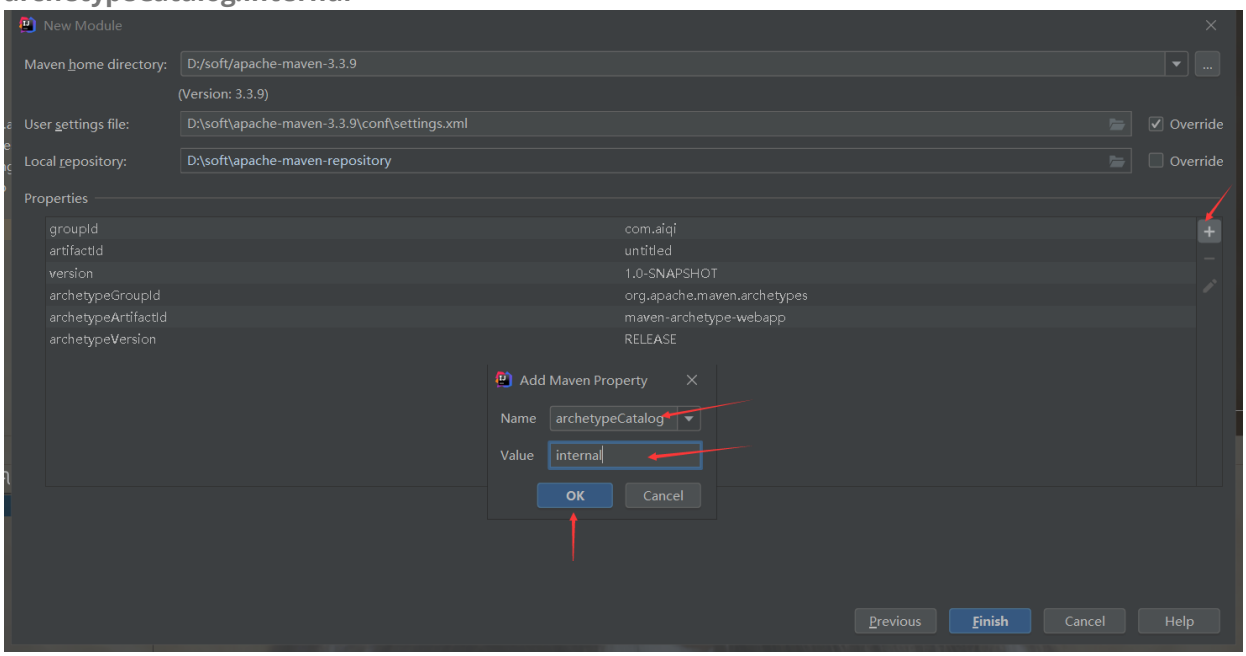
二、入门案例

2.1 搭建环境

> 创建maven项目,勾选骨架,选择**maven-archetype-webapp**



> 一直点击下一步到这里,为了解决创建javaweb项目时间过慢问题,添加一组键值对
> **archetypeCatalog:internal**



> 创建完成后,pom文件导入相关依赖

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <spring.version>5.1.5.RELEASE</spring.version>
</properties>

<dependency>
    <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.2.1</version>
    <scope>provided</scope>
</dependency>
```

2.2 代码编写

> 配置springmvc文件-->在resource下创建spring文件夹,在该文件夹下创建spring-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd
 http://www.springframework.org/schema/mvc
 http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd">

    <!-- HandlerMapping 无需配置，SpringMVC可以默认启动，DefaultAnnotationHandlerMapping
        annotation-driven HandlerMapping -->

    <!-- 配置SpringMVC -->
    <!-- 1.开启SpringMVC注解模式 -->
    <!-- 简化配置： (1)自动注册
DefaultAnootationHandlerMapping,AnotationMethodHandlerAdapter
        (2)提供一些列：数据绑定，数字和日期的format @NumberFormat, @DateTimeFormat, xml,json
默认读写支持 -->

    <mvc:annotation-driven/>
```

```
    <!-- 2.静态资源默认servlet配置 (1)加入对静态资源的处理：js,gif,png (2)允许使用"/"做整体映射，不会拦截，当为静态资源。
        这两个都是处理静态资源的，区别可以理解成一个是指定一个自定义的serlvet来专门处理相应的静态资源，如果不指定 会默认找default名字的servlet
        而<mvc:resources>的好处可以理解成是静态资源可以在我们项目中的任意位置配置，只需要将对应的位置声明即可 -->
    <mvc:resources mapping="/resources/**"
        location="/static/" />
    <mvc:default-servlet-handler />

    <!-- 3.定义视图解析器 -->
    <!-- ViewResolver:视图解析器。可以配置多个 但是一定要将这个ViewResolver(InternalResourceViewResolver)
        放到最后 -->
    <!-- 解析json格式的传参和封装数据到页面，注意spring的版本和对应的配置方式 -->
    <!-- spring-4.2以后 -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>


    <!-- 4.扫描controller相关的bean -->
    <!-- 激活组件扫描功能,扫描aop的相关组件 -->
    <context:component-scan
        base-package="com.aiqi.controller" />

</beans>
```

> 在webapp/WEB-INF/下创建pages文件夹,然后在该文件夹下创建success.jsp文件,显示成功信息,代码略

> 修改webapp/WEB-INF/web.xml配置文件

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <servlet>
    <servlet-name>spring-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 用于加载spring配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring/spring-*.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
```

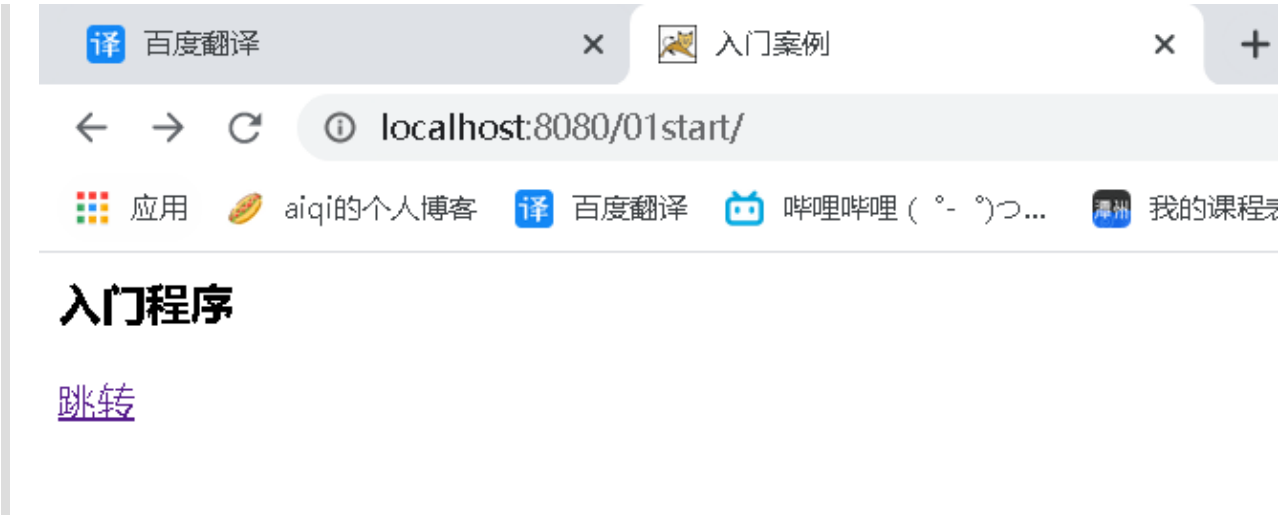
```
    <servlet-name>spring-dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <!-- 配置解决中文乱码的过滤器 -->
  <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>
</web-app>
```

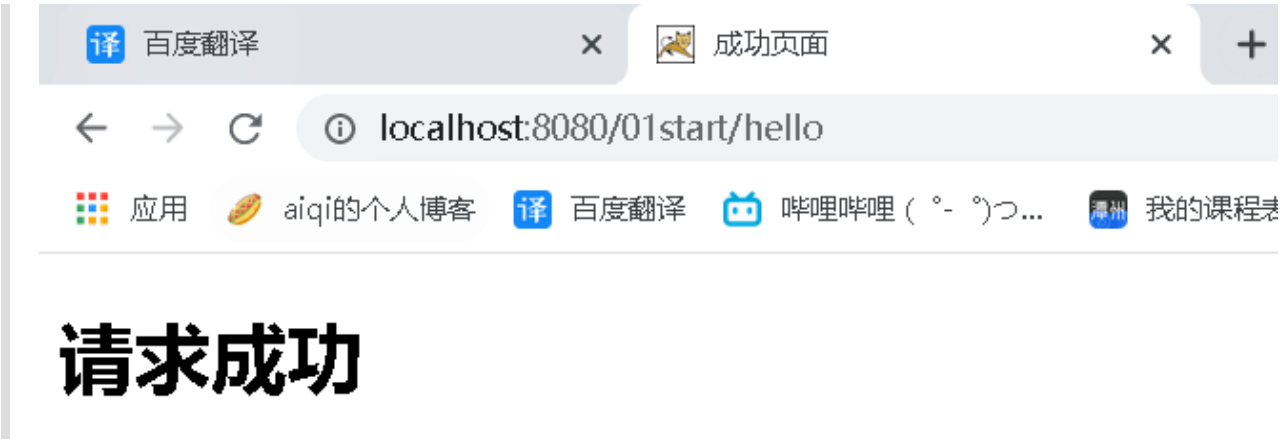
> 编写controller.java

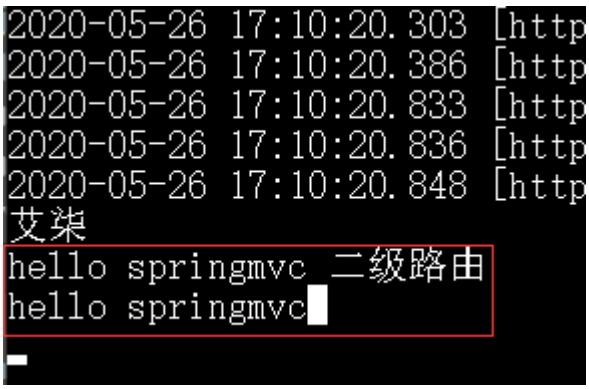
```
package com.aiqi.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * @description: 入门Controller
 * @author: aiqi
 * @create: 2020-05-11 22:25
 **/

//控制器类用于接收请求
@Controller
//一级路由
@RequestMapping(path = "/hello",method= RequestMethod.GET)
public class HelloController {
    //一级路由
    @RequestMapping(path = "",method= RequestMethod.GET)
    public String sayHello(){
        System.out.println("hello springmvc");
        return "success";
    }

    //二级路由
    @RequestMapping(path = "/L2",method= RequestMethod.GET)
    public String sayL2Hello(){
        System.out.println("hello springmvc 二级路由");
        return "success";
    }
}
```

2.3 测试

部分测试结果如下







三、参数绑定及自定义类型转换

3.1 请求参数的绑定

3.1.1 基本类型

```
<form action="param/login" method="post">
    账户:<input type="text" name="username"><br>
    密码:<input type="password" name="pwd"><br>
    <input type="submit" value="点击登录">
</form>
```

```
@RequestMapping(value = "/login",method = RequestMethod.POST)
public String login(String username,String pwd){
  System.out.println("账号:"+username+",密码:"+pwd);
  return "success";
}
```

3.1.2 javabean类型

```
<form action="param/register" method="post">
    账户:<input type="text" name="username"><br>
    密码:<input type="password" name="pwd"><br>
    金额:<input type="text" name="money"><br>
    用户名:<input type="text" name="user.uname"><br>
    用户年龄:<input type="text" name="user.age"><br>
    <input type="submit" value="点击提交">
</form>
```

```
@RequestMapping(value = "/register",method = RequestMethod.POST)
public String register(Account account){
    System.out.println(account);
    return "success";
}
```

3.1.3 集合类型

```
<form action="param/register" method="post">
    账户:<input type="text" name="username"><br>
    密码:<input type="password" name="pwd"><br>
    金额:<input type="text" name="money"><br>

    用户名:<input type="text" name="userList[0].uname"><br>
    用户年龄:<input type="text" name="userList[0].age"><br>

    用户名:<input type="text" name="userMap['key1'].uname"><br>
    用户年龄:<input type="text" name="userMap['key1'].age"><br>

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

```
@RequestMapping(value = "/register",method = RequestMethod.POST)
public String register(Account account){
    System.out.println(account);
    return "success";
}
```

3.2 数据类型转换

> 前端传过来的数据是String类型的,比如age字段"18",后端可用Integer接收,springmvc帮我们做了简单的类型转换封装,而有些数据,比如日期,用户输入的格式可能不正确,这就需要我们用到自定义类型转换.

> 首先第一步我们需要编写日期转换工具类(字符串-->日期) 这个类需要实现spring的Converter接口,并重写convert方法,S是source源,T是target目标,即从S类型转为T类型,在本例中即String转为Date,代码如下

```
package com.aiqi.utils;

import org.springframework.core.convert.converter.Converter;

import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @description: 自定义类型转换器工具类(字符串转日期)
 * @author: aiqi
 * @create: 2020-05-13 16:00
 **/
public class StringToDateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String s) {
        if(s==null){
            throw new RuntimeException("未传入数据");
        }
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        try {
            return sdf.parse(s);
        } catch (Exception e) {
            throw new RuntimeException("数据类型转换错误");
        }
    }
}
```

> 编写完转换工具类后,我们还需进行配置,修改springmvc的配置文件(spring-web.xml)

```
<!-- 注册自定义类型转换器 -->
<bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <set>
        <bean class="com.aiqi.utils.StringToDateConverter"></bean>
    </set>
  </property>
</bean>

<mvc:annotation-driven conversion-service="conversionService"/>
```

四、常用注解

4.1 RequestMapping

作用:

> 用来标识 http 请求地址与 Controller 类的方法之间的映射

属性:

> value:用于指定请求的URL.它和path属性的作用是一样的。

> method:用于指定请求的方式。

> params:用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的key和value必须和 配置的一模一样。 例如: params={" accountName" },表示请求参数必须有accountName params = { "moeny!100"},表示请求参数中money不能是100。

> headers:用于指定限制请求消息头的条件。

> **注意**: 以上四个属性只要出现2个或以上时，他们的关系是与的关系。

用法:

> 可写在Controller类上

```
@Controller
@RequestMapping(path = "/anno")
public class AnnoController {
  //...略
}
```

> 可写在方法上

```
@RequestMapping(path = "/m1",method= RequestMethod.GET)
public String m1(){
  return "success";
}
```

4.2 RequestParam

作用:

> 把请求中指定名称的参数给控制器中的形参赋值。用于指定前端传过来的参数和方法参数一一对应的关系

属性:

> value/name:请求参数中的名称。

> required:请求参数中是否必须提供此参数。默认值: true。表示必须提供，如果不提供将报错。

用法:

> 写在方法形式参数前面

```
public String m1(@RequestParam("username") String name){
    System.out.println(name);
    return "success";
}
```

4.3 RequestBody

作用:

> 用于获取请求体内容。直接使用得到是key=value&key=value.. .结构的数据。 get请求方式不适用。

属性:

> required:是否必须有请求体。默认值是:true。当取值为true 时,get请求方式会报错。如果取值 为false, get请求得到是null.

用法:

> 写在方法形式参数前面

```
@RequestMapping(path = "/RequestBody",method = RequestMethod.POST)
public String m2(@RequestBody String body){
  System.out.println(body);
  return "success";
}
```

4.4 PathVariable

作用:

> 用于绑定url中的占位符。例如:请求url中/delete/{id}, 这个{id}就是url占位符。 url支持占位符是spring3.0之后加入的。是springmvc支持restful风格URL的一个重要标志。

属性:

> value/name:用于指定url中占位符名称。

> required:是否必须提供占位符。

用法:

> 和@RequestMapping配合使用,写在方法形式参数前面

```
@RequestMapping(path = "/PathVariable/{id}",method = RequestMethod.GET)
public String m3(@PathVariable("id") String id){
System.out.println(id);
return "success";
}
```

4.5 RequestHeader

作用:

> 用于获取请求消息头。

属性:

> value/name:提供消息头名称

> required:是否必须有此消息头

用法:

> (在实际开发中一般不怎么用。)

```
//获取请求头中的Accept字段
public String m4(@RequestHeader("Accept") String accept){
  System.out.println(accept);
  return "success";
}
```

4.6 ResponseBody

作用:

> 如果方法声明了该注解,则会直接将返回值输出到页面。

属性:

> 无

用法:

> 一般使用它是为了返回json格式数据,前端ajax等异步请求

```
@RequestMapping("/login.do")
@ResponseBody
public Object login(String name, String password, HttpSession session) {
  user = userService.checkLogin(name, password);
  session.setAttribute("user", user);
  return new JsonResult(user);
}
```

4.7 CookieValue

作用:

> 用于把指定cookie名称的值传入控制器方法参数。

属性:

> value/name:指定cookie的名称。

> required:是否必须有此cookie。

用法:

> 写在方法形式参数前面

```
@RequestMapping(path = "/CookieValue",method = RequestMethod.GET)
public String m5(@CookieValue("JSESSIONID") String cookie){
    System.out.println(cookie);
    return "success";
}
```

4.8 ModelAttribute

作用:

> 该注解是SpringMVC4.3版本以后新加入的。它可以用于修饰方法和参数。 出现在方法上，表示当前方法会在控制器的方法执行之前，先执行。它可以修饰没有返回值的方法，也可以修饰有具体返回值的方法。 出现在参数上，获取指定的数据给参数赋值。

属性:

> value/name:用于获取数据的key。key可以是POJO的属性名称，也可以是map结构的key。

用法:

> 当表单提交数据不是完整的实体类数据时，保证没有提交数据的字段使用数据库对象原来的数据。 例如: 我们在编辑-一个用户时，用户有一个创建信息字段，该字段的值是不允许被修改的。在提交表单数 据是肯定没有此字段的内容，一旦更新会把该字段内容置为null,此时就可以使用此注解解决问题。

```
@RequestMapping(path = "/ModelAttribute",method = RequestMethod.GET)
    public String m6(){
        System.out.println("后执行");
        return "success";
    }
    @ModelAttribute
    public void show(){
        System.out.println("show方法先执行");
    }

    //有返回值
    @ModelAttribute
    public User show1(String uname,Integer age){
        System.out.println("show1方法先执行");
        //模拟从数据库查询数据
        User user = new User();
        user.setUname(uname);
        user.setAge(age);
        user.setBirthday(new Date());
```

```
        return user;
    }

    @RequestMapping(path = "/ModelAttribute1",method = RequestMethod.POST)
    public String m7(User user){
        System.out.println(user);
        return "success";
    }

    //无返回值
    @ModelAttribute
    public void show2(String uname, Integer age, Map<String,User> userMap){
        System.out.println("show2方法先执行");
        //模拟从数据库查询数据
        User user = new User();
        user.setUname(uname);
        user.setAge(age);
        user.setBirthday(new Date());
        userMap.put("userOne",user);
    }
    @RequestMapping(path = "/ModelAttribute2",method = RequestMethod.POST)
    public String m8(@ModelAttribute("userOne") User user){
        System.out.println(user);
        return "success";
    }
```

4.9 SessionAttributes

作用:

> 用于多次执行控制器方法间的参数共享。

属性:

> value/names:用于指定存入的属性名称

> type:用于指定存入的数据类型。

用法:

>

```
@SessionAttributes(value = {"userInfo_name"})
public class AnnoController {
    //存值Session
    @RequestMapping(path = "/SessionAttributes/add",method = RequestMethod.GET)
    public String m9(String username, Model model){
        model.addAttribute("userInfo_name",username);
        System.out.println("添加成功");
        return "success";
    }

    //取值Session

    @RequestMapping(path = "/SessionAttributes/get",method = RequestMethod.GET)
```

```
    public String m10(ModelMap modelMap){
        String name = (String)modelMap.get("userInfo_name");
        System.out.println(name);
        return "success";
    }

    //删值Session
    @RequestMapping(path = "/SessionAttributes/del",method = RequestMethod.GET)
    public String m11(SessionStatus status){
        status.setComplete();
        System.out.println("删除成功");
        return "success";
    }
}
```

4.10 HiddentHttpMethodFilter

作用:

> 由于浏览器form表单只支持GET与POST请求，而DELETE、 PUT等method 并不支持，Spring3.0 添加了一个过滤器， 可以将浏览器请求改为指定的请求方式，发送给我们的控制器方法，使得支持GET、POST、PUT与DELETE 请求。

用法:

> 第一步:在web. xml中配置该过滤器。 第二步:请求方式必须使用post请求。 第三步:按照要求提供method 请求参数，该参数的取值就是我们需要的请求方式。
>
> 它是一种过滤器(了解就行),更多见有关博客

五、返回值类型及响应数据类型

5.1 String

> controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。 forward转发或redirect重定向 controller方法在提供了String返回值后,默认就是采用请求转发,我们可以写下面这种写法 当使用forward或redirect关键词后,我们就用不了视图解析器,需要将文件路径或路由路径写完整 注意:当使用redirect重定向时,不用拼接项目名

```
@Controller
@RequestMapping("/return")
public class ReturnController {
    @RequestMapping(value = "/String",method = RequestMethod.GET)
    public String m1(Model model){
        System.out.println("返回一个页面");
        //模拟从数据库查询User对象
        User user = new User();
        user.setUsername("aiqi");
        user.setPassword("123");
        user.setAge(20);
        //将数据存入到request域中

        model.addAttribute("user",user);
```

```
        return "success";
    }

    //...
}
```

```
@RequestMapping(value = "/ForwardOrRedirect",method = RequestMethod.GET)
    public String m4(HttpServletRequest request){
//        System.out.println("执行了请求转发");
//        return "forward:/WEB-INF/pages/success.jsp";

        System.out.println("执行了重定向");
        return "redirect:/redirect.jsp";
    }
```

5.2 void

> 默认情况返回请求路径的jsp或html等视图文件 可以使用原生Servlet的request,response对象,执行请求转发或请求重定向

```
    @RequestMapping(value = "/void",method = RequestMethod.GET)
    public void m2(HttpServletRequest request, HttpServletResponse response) throws
Exception {

        //请求转发
        /*System.out.println("请求转发");
        request.getRequestDispatcher("/WEB-
INF/pages/success.jsp").forward(request,response);
        return;*/

        //重定向
        /*System.out.println("重定向");
        response.sendRedirect(request.getContextPath()+"/redirect.jsp");
        return;*/

        //直接响应
        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/html;charset=UTF-8");
        response.getWriter().println("哈哈");
    }
```

5.3 ModelAndView

> Spring提供的一个对象,可以用来调整具体的jsp视图, 和返回String差不多一样,只是写法不一样

```
@RequestMapping(value = "/ModelAndView",method = RequestMethod.GET)
public ModelAndView m3(Model model){
    //模拟从数据库查询User对象
    User user = new User();
    user.setUsername("奶妈");
    user.setPassword("123456");
    user.setAge(20);
    //将数据存入到request域中
    model.addAttribute("user",user);
    //返回视图+model
    return new ModelAndView("success", "model", model);
}
```

5.4 View

> 可以返回pdf excel等，暂时没详细了解。

5.5 Map

> 在jsp页面中可直通过${key}获得到值, map.put()相当于request.setAttribute方法。 一般用的最多的还是用@ResponseBody返回一个JSON

```
@RequestMapping(value = "/JSON2")
@ResponseBody
public Map<String, Object> m6(Integer id){
    Map<String, Object> modelMap = new HashMap<String, Object>();
    List<User> userList = new ArrayList<User>();
    //模拟数据库操作通过id查用户
    User user1 = new User();
    user1.setUsername("aiqi");
    user1.setPassword("123");
    user1.setAge(20);
    User user2 = new User();
    user2.setUsername("奶妈");
    user2.setPassword("456");
    user2.setAge(20);
    userList.add(user1);
    userList.add(user2);
    modelMap.put("success",true);
    modelMap.put("userList",userList);
    return modelMap;
}
```

```
@RequestMapping(value = "/JSON")
@ResponseBody
public User m5(@RequestBody User user){
    //客户端浏览器发送ajax请求,发送的是json数据,后端把json字符串封装到javabean中(User)
    System.out.println(user);

    //模拟数据库操作
    user.setAge(40);
    return user;
}
```

六、文件上传

> 搭建环境 在本节中,我们需要用到fileupload组件和跨服务器上传组件,所以我们在pom中导入相关依赖

```
<!-- 文件上传相关包 -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
</dependency>
<!-- 跨服务器上传有关包 -->
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.18.1</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.18.1</version>
</dependency>
```

6.1 传统上传

> 我们先来回顾下传统文件上传,使用fileupload第三方组件

前端:

```
<h3>传统上传</h3>
<form action="fileupload/method1" method="post" enctype="multipart/form-data">
    <input type="file" name="file1"><br>
    <input type="submit" value="点击上传">
</form>
```

后端控制器:

```java
@Controller
@RequestMapping("/fileupload")
public class FileUploadController {
    @RequestMapping("/method1")
    //使用fileupload组件完成文件上传功能
    public String m1(HttpServletRequest request) throws Exception {
        System.out.println("正在上传文件");

        //1.获取文件夹真实路径
        String path = request.getSession().getServletContext().getRealPath("/userimg/");
        System.out.println(path);
        //2.创建文件夹
        File file = new File(path);
        //判断该路径是否存在
        if (!file.exists()) {
            file.mkdirs();
        }
        //3.解析request对象,获取文件上传项
        DiskFileItemFactory factory = new DiskFileItemFactory();
        ServletFileUpload upload = new ServletFileUpload(factory);
        //拿到上传文件集合
        List<FileItem> fileItems = upload.parseRequest(request);
        //遍历
        for (FileItem item : fileItems) {
            //进行判断,判断当前item是否是上传文件项
            if (item.isFormField()) {
                //普通表单项
            } else {
                //上传文件项
                //获取上传文件名
                String filename = item.getName();
                //拼接唯一UUID作为文件名
                filename = UUID.randomUUID().toString().replaceAll("-", "") + "-" +
filename;
                //文件上传
                item.write(new File(path, filename));
                //删除临时文件
                item.delete();
                System.out.println("文件上传成功");
            }
        }
        return "success";
    }

    //m2,m3....
}
```

6.2 SpringMVC上传

> 首先我们需要配置下SpringMVC上传组件,打开springmvc的配置文件添加以下内容

```
<!--配置文件解析器对象，要求id名称必须是multipartResolver -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 10M = 10*1024*1024 以字节为单位 -->
    <property name="maxUploadSize" value="10485760"></property>
</bean>
```

> 编辑前端代码和后端控制器代码

```
<h3>SpringMVC上传</h3>
<form action="fileupload/method2" method="post" enctype="multipart/form-data">
    <input type="file" name="file2"><br>
    <input type="submit" value="点击上传">
</form>
```

```
    @RequestMapping("/method2")
    //使用SpringMVC完成文件上传功能
    public String m2(HttpServletRequest request, @RequestParam("file2") MultipartFile
uploadfile) throws Exception {
        System.out.println("正在上传文件");

        //1.获取文件夹真实路径
        String path = request.getSession().getServletContext().getRealPath("/userimg/");
        System.out.println(path);
        //2.创建文件夹
        File file = new File(path);
        //判断该路径是否存在
        if (!file.exists()) {
            file.mkdirs();
        }

        //自定义上传文件名字
        String filename = UUID.randomUUID().toString().replaceAll("-", "") + "-" +
uploadfile.getOriginalFilename();

        //文件上传
        uploadfile.transferTo(new File(path,filename));
        System.out.println("文件上传成功");
        return "success";
    }
```

6.2 SpringMVC跨服务器上传

> 我们知道,在真实的项目开发中,服务器分为应用服务器,资源服务器等等,应用服务器用来部署应用,而资源服务器用来存储一些资源,如图片音频视频等,这时如果要完成文件上传功能,不可能上传到应用服务器上,这时就需要用到跨服务器上传文件

> 首先我们模拟下资源服务器:开启第二个tomcat服务器,创建新的web项目,并部署到服务器上.开启端口8090

> 完成后编写代码

```
<h3>SpringMVC跨服务器上传</h3>
<form action="fileupload/method3" method="post" enctype="multipart/form-data">
    <input type="file" name="file3"><br>
    <input type="submit" value="点击上传">
</form>
```

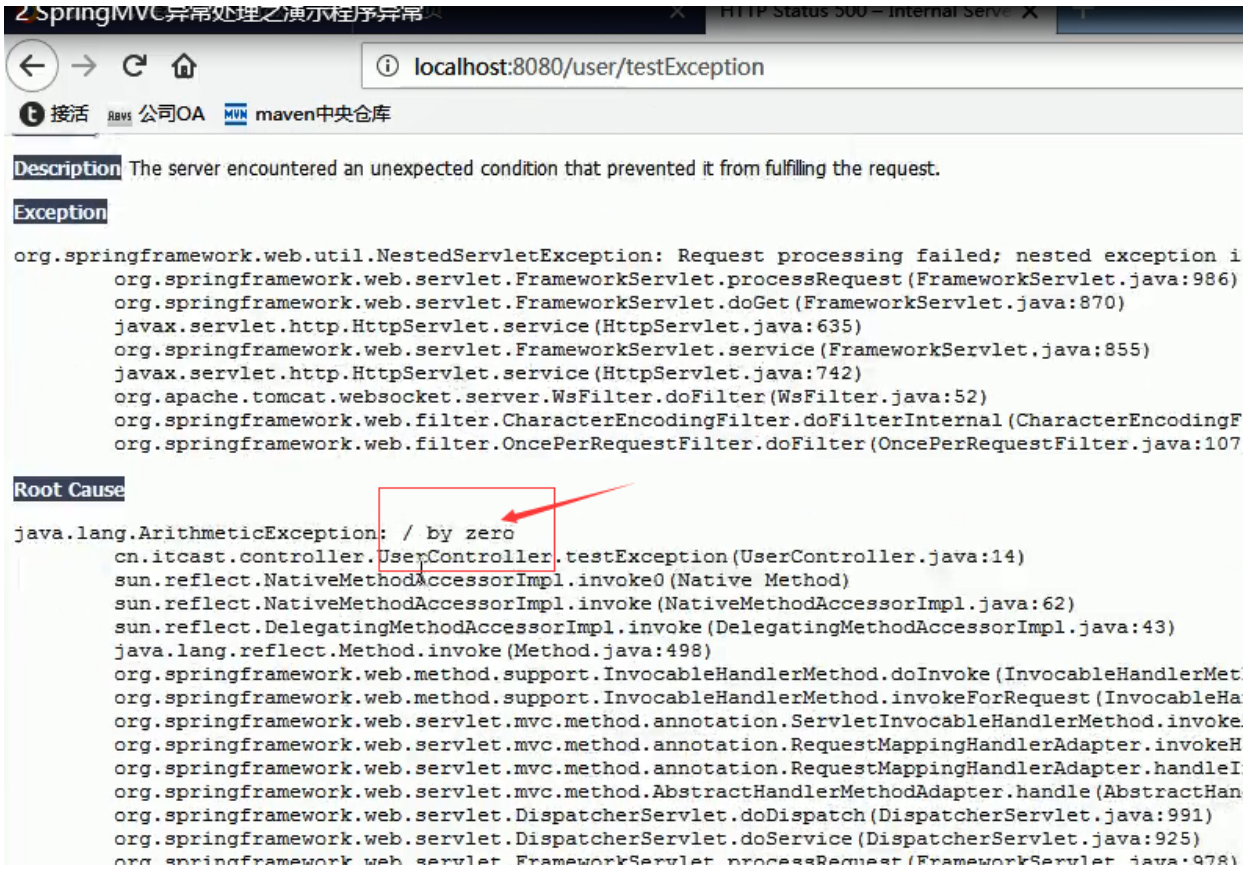
```
    @RequestMapping("/method3")
    //使用SpringMVC完成跨服务器文件上传功能
    public String m3(@RequestParam("file3") MultipartFile uploadfile) throws Exception {
        System.out.println("正在上传文件");

        //自定义上传文件名字
        String filename = UUID.randomUUID().toString().replaceAll("-", "") + "-" +
uploadfile.getOriginalFilename();

        //定义文件服务器路径
        String location = "http://localhost:8090/06fileServer/uploads";

        //跨服务器文件上传
        //1.创建客户端对象
        Client client = Client.create();
        //2.和文件服务器进行连接
        WebResource resource = client.resource(location + "/" + filename);
        //3.上传文件
        resource.put(uploadfile.getBytes());
        System.out.println("文件上传成功");
        return "success";
    }
```

七、异常处理及拦截器

7.1 异常处理

> 我们知道程序出现异常,如500错误等都会在前端页面上显示,给用户体验很不好,这时需要我们对异常进行处理,并对用户进行一定的反馈

> 我们模拟异常int i = 1/0; 访问后,出现异常



这时该怎么解决,首先我们自定义一个异常类,继承Exception

```
package com.aiqi.exception;

/**
 * @description: 自定义异常类
 * @author: aiqi
 * @create: 2020-05-15 19:34
 **/
public class MyException extends RuntimeException{
    private static final long serialVersionUID = 1L;

    private String msg;

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public MyException(String msg){
        this.msg = msg;
    }
}
```

> 自定义异常定义完了,我们还应该对这个异常进行处理,而springmvc给我们提供了一个HandlerExceptionResolver异常处理类接口,实现它,在之前,我们需要写一个异常视图,用来将错误信息反馈给用户,在本例中用了error.jsp

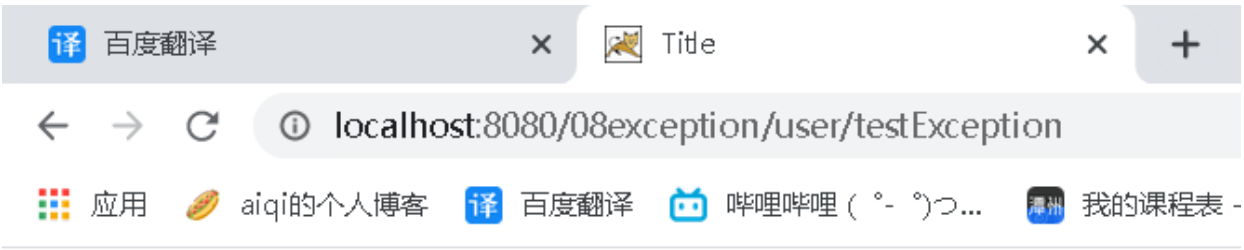
```
package com.aiqi.exception;

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @description: 自定义异常处理器类
 * @author: aiqi
 * @create: 2020-05-15 19:53
 **/
public class MyExceptionHandle implements HandlerExceptionResolver {

    /*
    * @Description: 处理异常业务逻辑
    * @Param: [httpServletRequest, httpServletResponse, o, e]
    * @return: org.springframework.web.servlet.ModelAndView
    **/
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o, Exception e) {
        //获取异常对象
        MyException myEx = null;
        if(e instanceof MyException){
            myEx = (MyException)e;
        }else {
            e = new MyException("服务器异常");
        }

        //创建ModelAndView对象
        ModelAndView mv = new ModelAndView();
        mv.addObject("errorMsg",myEx.getMsg());

        mv.setViewName("error");  //设置视图名称,对应error.jsp

        return mv;
    }
}
```

> 异常类和异常处理器类都写完了后,我们需要在springmvc的配置文件中配置下,也就是配bean

```
<!-- 配置异常处理器 -->
<bean id="myExceptionHandle" class="com.aiqi.exception.MyExceptionHandle"></bean>
```

> 配置完之后我们编写下控制器方法

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/testException")
    public String m1() throws MyException{
        System.out.println("执行了m1Controller方法");
        try {
            //模拟异常
            int i = 1/0;
        }catch (Exception e){
//            e.printStackTrace();
            //向上抛出自定义异常
            throw new MyException("出现了无法预知的错误");
        }
        return "success";
    }

}
```

> 用浏览器访问这个路由地址,在我们没写异常处理前,肯定是会报500错误的.我们测试下编写异常处理后的情况
>
> 
>
>
> # 错误页
>
> 错误信息:出现了无法预知的错误
>
> 成功,用户体验有了提高

7.2 拦截器

7.2.1 拦截器简介

> Spring web MVC的处理器拦截器类似于Servlet开发中的过滤器Filter，用于对处理器 进行预处理和后处理。

> **拦截器的作用**
>
> 用户可以自己定义一些拦截器来实现特定的功能。 谈到拦截器，还要向大家提一一个词一拦截器链(Interceptor Chain)。拦截器链就是将拦截器按一定的顺序联结成一条链。在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。 说到这里，可能大家脑海中有了一个疑问，这不是我们之前学的过滤器吗?是的它和过滤器是有几分相似，但是也有区别，接下来我们就来说说他们的区别: 过滤器是servlet规范中的一部分，任何java web工程都可以使用。 拦截器是SpringMVC框架自己的，只有使用了SpringMVC框架的工程才能用。 过滤器在url-pattern中配置了/*之后，可以对所有要访问的资源拦截。 拦截

> 器它是只会拦截访问的控制器方法，如果访问的是jsp, html,css, image或者js是不会进行拦 截的。 它也是AOP思想的具体应用。 我们要想自定义拦截器，要求必须实现: HandlerInterceptor 接口。

7.2.2 常见应用场景

> 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
>
> 更多见博客

7.2.3 了解和使用

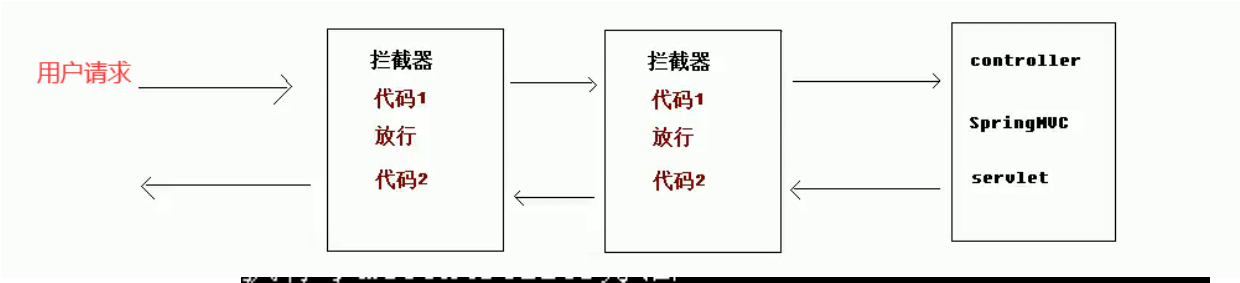
> 在本章中我们只为了了解预处理和后处理方法以及拦截器的配置,并没有用到具体的项目中去,如登录检测 首先我们编写拦截器类实现spring提供的HandlerInterceptor接口

```
package com.aiqi.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @description: 自定义拦截器
 * @author: aiqi
 * @create: 2020-05-15 18:56
 **/
public class MyInterceptor1 implements HandlerInterceptor {
    /*
    * @Description: 预处理,Controller方法执行前,执行该方法
    * @Param: [request, response, handler]
    * @return: boolean  true:放行,执行下一个拦截器,如果没有下一个拦截器了,就执行Controller方法
    **/
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("111--调用Controller方法前执行了前置拦截器");
        //request.getRequestDispatcher("/WEB-
INF/pages/error.jsp").forward(request,response);
        return true;
    }

    /*
    * @Description: 后处理,执行完Controller方法后执行此方法
    * @Param: [request, response, handler, modelAndView]
    * @return: void
    **/
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("111--Controller方法执行后该后置方法执行了");
        //request.getRequestDispatcher("/WEB-
INF/pages/error.jsp").forward(request,response);
    }
```

```
    /*
    * @Description: 返回的视图页面执行后,该方法执行
    * @Param: [request, response, handler, ex]
    * @return: void
    **/
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("111--所有方法完成后执行了此afterCompletion方法");
        //该方法已经不能再跳页面了,因为视图已经显示出来了
    }
}
```

> 再编写一个拦截器2,为了我们更好的知道拦截器是怎么执行的,代码和拦截器1差不多一样,就省略不写了 编写完之后,我们在springmvc的配置文件中进行相应的配置

```
    <!-- 配置拦截器 -->
    <mvc:interceptors>
        <mvc:interceptor>
            <!--要拦截的Controller方法 -->
            <mvc:mapping path="/user/*"/>  <!-- /** 表示拦截user路由下的方法  -->
            <!--不用拦截的Controller方法
            <mvc:exclude-mapping path=""/>-->
            <!--注入拦截器对象 -->
            <bean class="com.aiqi.interceptor.MyInterceptor1"></bean>
        </mvc:interceptor>
        <mvc:interceptor>
            <!--要拦截的Controller方法 -->
            <mvc:mapping path="/**"/>  <!-- /** 表示所有方法都拦截  -->
            <!--注入拦截器对象 -->
            <bean class="com.aiqi.interceptor.MyInterceptor2"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
```

> 在进行测试之前,我们回顾下servlet的filter过滤器,他的执行过程,和springmvc差不多一致,我们推导下interceptor的执行过程
>
> 
>
>
> 我们输入网址测试下
>
> ```
> 111--调用Controller方法前执行了前置拦截器
> 222--调用Controller方法前执行了前置拦截器
> 执行了mlController方法
> 222--Controller方法执行后该后置方法执行了
> 111--Controller方法执行后该后置方法执行了
> 执行了success.jsp
> 222--所有方法完成后执行了此afterCompletion方法
> 111--所有方法完成后执行了此afterCompletion方法
> ```
>
> 发现了拦截器是这样执行的

八、SSM整合

8.1 相关依赖

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <spring.version>5.1.5.RELEASE</spring.version>
</properties>

<dependencies>
    <!-- 打印日志有关包 -->
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.2.3</version>
    </dependency>
    <!-- 单元测试包 -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
    </dependency>
    <!-- spring有关包 -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
```

```
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <!-- 支持切入点表达式 -->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.13</version>
    </dependency>
    <!-- servlet有关包 -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>javax.servlet.jsp-api</artifactId>
      <version>2.2.1</version>
    </dependency>
    <!-- JSTL表达式 -->
    <dependency>
      <groupId>jstl</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
    </dependency>
    <!-- JSON有关包 -->
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>

      <artifactId>jackson-databind</artifactId>
```

```
      <version>2.9.0</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.9.0</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-annotations</artifactId>
      <version>2.9.0</version>
    </dependency>
    <!-- DAO:mybatis -->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.4</version>
    </dependency>
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis-spring</artifactId>
      <version>1.3.1</version>
    </dependency>
    <!-- mysql数据库及连接池 -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.6</version>
    </dependency>
    <dependency>
      <groupId>com.mchange</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.5.3</version>
    </dependency>
</dependencies>
```

8.2 业务层

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd
 http://www.springframework.org/schema/tx
 http://www.springframework.org/schema/tx/spring-tx.xsd">
<!-- 扫描service包下所有使用注解的类型 -->
<context:component-scan base-package="com.aiqi.service"/>
```

```
<!-- 配置事务管理器 -->
<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<!-- 注入数据库连接池 -->
<property name="dataSource" ref="dataSource"/>
</bean>

<!-- 配置基于注解的声明式事务 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

8.3 持久层

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
    <!-- 配置整合mybatis过程 -->
    <!-- 1.配置数据库相关参数properties的属性：${url} -->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!-- 2.数据库连接池 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <!-- 配置连接池属性 -->
        <property name="driverClass" value="${jdbc.driver}"/>
        <property name="jdbcUrl" value="${jdbc.url}"/>
        <property name="user" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>

        <!-- c3p0连接池的私有属性 -->
        <property name="maxPoolSize" value="30"/>
        <property name="minPoolSize" value="10"/>
        <!-- 关闭连接后不自动commit -->
        <property name="autoCommitOnClose" value="false"/>
        <!-- 获取连接超时时间 -->
        <property name="checkoutTimeout" value="10000"/>
        <!-- 当获取连接失败重试次数 -->
        <property name="acquireRetryAttempts" value="2"/>
    </bean>

    <!-- 3.配置SqlSessionFactory对象 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- 注入数据库连接池 -->
        <property name="dataSource" ref="dataSource"/>
        <!-- 配置MyBaties全局配置文件:mybatis-config.xml -->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <!-- 扫描entity包 使用别名 -->

        <property name="typeAliasesPackage" value="com.aiqi.pojo"/>
```

```
        <!-- 扫描sql配置文件:mapper需要的xml文件 -->
        <property name="mapperLocations" value="classpath:mapper/*.xml"/>
    </bean>

    <!-- 4.配置扫描Dao接口包，动态实现Dao接口，注入到spring容器中 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!-- 注入sqlSessionFactory -->
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
        <!-- 给出需要扫描Dao接口包 -->
        <property name="basePackage" value="com.aiqi.dao"/>
    </bean>
</beans>
```

8.4 控制层

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd
 http://www.springframework.org/schema/mvc
 http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd">

    <!-- HandlerMapping 无需配置，SpringMVC可以默认启动，DefaultAnnotationHandlerMapping
        annotation-driven HandlerMapping -->

    <!-- 配置SpringMVC -->
    <!-- 1.开启SpringMVC注解模式 -->
    <!-- 简化配置： (1)自动注册
DefaultAnootationHandlerMapping,AnotationMethodHandlerAdapter
        (2)提供一些列：数据绑定，数字和日期的format @NumberFormat, @DateTimeFormat, xml,json
默认读写支持 -->
    <mvc:annotation-driven />

    <!-- 2.静态资源默认servlet配置 (1)加入对静态资源的处理：js,gif,png (2)允许使用"/"做整体映射
, 不会拦截，当为静态资源。
        这两个都是处理静态资源的，区别可以理解成一个是指定一个自定义的serlvet来专门处理相应的静态资
源，如果不指定 会默认找default名字的servlet
        而<mvc:resources>的好处可以理解成是静态资源可以在我们项目中的任意位置配置，只需要将对应的位
置声明即可 -->
    <mvc:resources mapping="/resources/**"
        location="/resources/" />
    <mvc:default-servlet-handler />

    <!-- 3.定义视图解析器 -->
    <!-- ViewResolver:视图解析器。可以配置多个 但是一定要将这个
ViewResolver(InternalResourceViewResolver)

        放到最后 -->
```

```
    <!-- 解析json格式的传参和封装数据到页面，注意spring的版本和对应的配置方式 -->
    <!-- spring-4.2以后 -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <!--文件上传解析器-->
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <property name="defaultEncoding" value="utf-8"></property>
        <property name="maxUploadSize" value="20971520"></property>
        <property name="maxInMemorySize" value="20971520"></property>
    </bean>

    <!-- 4.扫描controller相关的bean -->
    <!-- 激活组件扫描功能,扫描aop的组件 -->
    <context:component-scan
        base-package="com.aiqi.controller" />

</beans>
```

8.5 web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1" metadata-complete="true">
  <display-name>Archetype Created Web Application</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <!-- 配置前端控制器 -->
  <servlet>
    <servlet-name>spring-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 用于加载spring配置文件 -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring/spring-*.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring-dispatcher</servlet-name>
    <!-- 默认匹配所有请求 -->
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <!-- 配置解决中文乱码的过滤器 -->

  <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>

  <!-- 配置session过期时间(min) -->
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
</web-app>
```

```
by aiqi
```