SpringMVC笔记

一、概述

SpringMVC是基于spring的一个框架，实际上就是spring的一个模块。这个模块专门做web开发的。即servlet的一个升级。

web开发底层是servlet,框架是再servlet基础上加入一些功能，让你做web开发方便。

SpringMVC就是一个Spring。Spring是容器，ioc能管理对象，使用,@Component，@Repository，@Service，@Controller。**SpringMVC能够创建对象，放到容器中，springmvc容器中放的是控制器对象。**

我们要做的就是**使用@Controller创建控制器对象，把对象放入到springmvc容器中，把创建的对象作为控制器使用。**这个控制器对象能接受用户的请求，显示处理结果，就当作是一个servlet使用。

使用@Controller注解创建的是一个普通类的对象，不是servlet。**springmvc赋予了控制器对象一些额外的功能。**

web开发底层是servlet，springmvc中有一个对象是servlet：**DispatherServlet（中央调度器）**

DispatherServlet:负责接受用户的所有请求，用户把请求给了**DispatherServlet**，之后**DispatherServlet**把请求转发给我们的Controller，最后是Controller对象处理请求。

index.jsp----------DispatherServlet(Servlet)----------转发，分配给Controller对象（@Controller注解创建的对象）

二、简介

SpringMVC简介

SpringMVC也叫做Spring web mvc。是Spring框架的一部分。

SpringMVC优点

1.基于MVC架构

基于MVC架构，功能分工明确。解耦合，

2.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。

3.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。

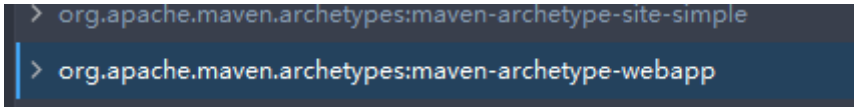
4.SpringMVC强化注解的使用，在控制器，Service，Dao都可以使用注解。方便灵活。

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,Service类中注入Dao。

SpringMVC入门

实现步骤：

1. **新建web maven工程**



2. **加入依赖**
    1. spring-webmvc依赖，间接把spring的依赖都加入到项目中
    2. jsp,servlet的依赖

```
<dependencies>
```

```
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
    <!--    servlet的依赖-->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
    <!--    spring-webmvc依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

3. **重点：在web.xml中注册springmvc框架的核心对象DispatcherServlet**
    1. DispatcherServlet叫做中央调度器，是一个servlet,它的父类是继承HttpServlet
    2. DispatcherServlet页叫做前端控制器（front controller）
    3. DispatcherServlet负责接受用户提交的请求，调用其他的控制器对象，并把请求的处理结果提示给用户

```
<!--    声明 注册springmvc的核心对象DispatcherServlet
        需要在tomcat服务器启动后，创建DispatcherServlet对象的实例。
        为什么要创建Dispactherservlet对象的实例呢？
        因为DispatcherServlet在他的创建过程中，会同时创建springmvc容器对象，
        读取sprinmvc的配置文件 ，把这个配置文件中的对象都创建好，当用户发起请求时就直接可以使用对象了.

        servlet的初始会执行init()方法。DispatcherServlet在init()中{
            //创建容器，读取配置文件
            WebApplicationContexyt ctx = new ClassPathXMLAppliactionContext("springmvc.xml")
            //把容器对象放到ServletContext中
            getServletContext().setAttribute(key,ctx);
        }

        启动tomcat报错，读取这个文件 /WEB-INF/springmvc-serlvet.xml
        springmvc创建容器对象时，读取的配置文件默认是/WEB-INF/<serlvet-name>-servlet.xml
-->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

    <!--        自定义springmvc的配置文件的位置的属性-->
    <init-param>
        <!--            springmvc的配置文件的位置的属性-->
        <param-name>contextConfigLocation</param-name>
        <!--            指定自定义文件的位置-->
        <param-value>classpath:springmvc.xml</param-value>
```

```
    </init-param>

    <!--         在tomcat启动后，创建Servlet对象
            load-on-startup:表示tomcat启动后创建对象的顺序。它的值是整数，数值越小，tomcat创建对象的时间越早。大于等于0的证书
    -->
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!--         使用框架的时候，url-pattern可以使用两种值-->
    <!--         1.扩展名的方式，语法 *.xxxx,xxxx是自定义的扩展名。常用的方式 *.do *.action *.mvc等-->
    <!--             http://localhost:8080/springmvc/springmvc1.do-->
    <!--             http://localhost:8080/springmvc/springmvc2.do-->
    <!--         2.使用斜杠 “/"-->
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

4. **创建一个发起请求的页面，index.jsp**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <p>第一个springmvc的项目</p>
    <p><a href="some.do">发起some.do请求</a></p>
</body>
</html>
```

5. **创建控制器类**

    1. 在类的上面加入@Controller注解，创建对象，并放入到springmvc容器中
    2. 在类中的方法上加入@RequestMapping注解

```
//Controller创建处理器对象，对象放在springmvc容器中
@Controller
public class MyController {
    //处理用户提交的请求，springmvc中是使用方法来处理的
    //方法是自定义的，可以有多种返回值，多种参数，方法名称自定义

//    准备使用doSome方法处理some.do请求
//    @RequestMapping 请求映射，作用是把一个请求地址和一个方法绑定在一起
//        一个请求指定一个方法
//        属性: value是一个String，表示请求的url地址 值必须是唯一值，不能重复。
//        位置：在方法的上面。在类的上面
//        使用@RequestMapping修饰的方法可以处理请求的，类似Servlet中的doGet，doPost
//        返回值 : ModelAndView = Model+View 数据加试图

    @RequestMapping(value = "/some.do")
    public ModelAndView doSome(){
        System.out.println("这是doSome的请求");
```

```
        ModelAndView modelAndView = new ModelAndView();
        //添加数据，框架在请求的最后把数据方法request作用域中
        //request.setAttribute("msg","欢迎使用mvc");
        modelAndView.addObject("msg","欢迎使用mvc");
        modelAndView.addObject("fun","执行doSome方法");
        //指定试听与，指定试图的完整路径
        //框架对试图执行的forword操作，
reqyest.getRequestDispatcher("/show.jsp").forword(...)
        modelAndView.setViewName("/show.jsp");
        return modelAndView;
    }
}
```

6. **创建一个作为结果的jsp,显示请求的处理结果。**

   show.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>show.jsp从request作用域中获取数据</h3>
    <h4>msg数据：${msg}</h4>
    <h4>fun数据：${fun}</h4>
</body>
</html>
```

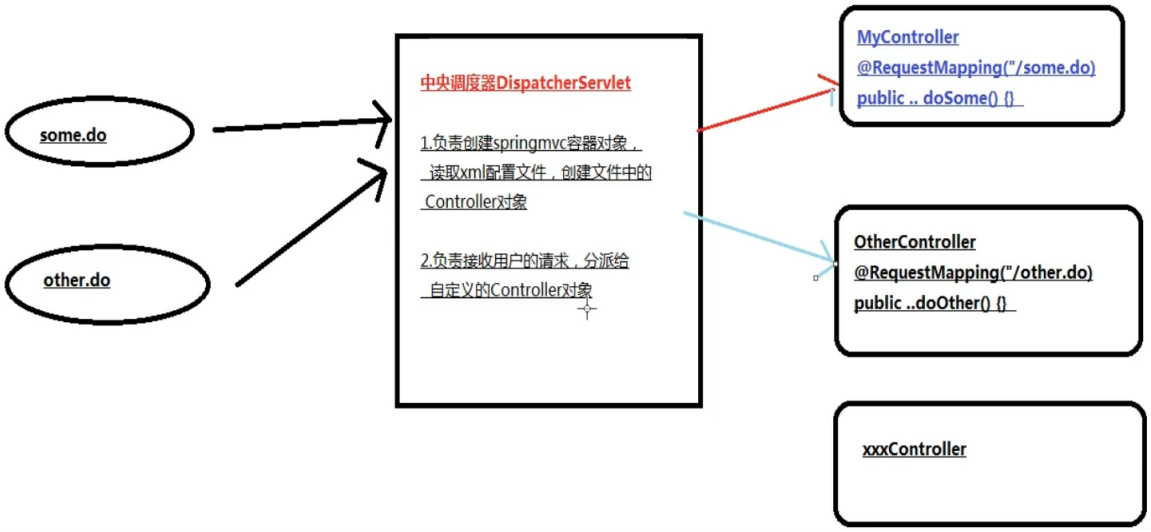
7. **创建springmvc的配置文件（spring的配置文件一样）**

   1. 声明组件扫描器，指定@Controller注解所在的包名
   2. 声明试图解析器。帮助处理视图的。

   springmvc.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.qiangliu8.controller"/>
</beans>
```

springmvc请求的处理流程



1. 发起some.do请求
2. tomcat（web.xml----url-pattern知道**.do请求给DisPatcherServlet）
3. DispatcherServlet（根据springmvc.xml配置知道some.do---doSome()）
4. DispatcherServlet把some.do转发给MyController.doSome（）方法
5. 框架执行doSome()把得到ModelAndView进行处理，转发到show.jsp

上面简化的过程：some.do----------DispatcherSerlvet----------MyController

springmvc执行过程源代码分析

1. **tomcat启动，创建容器的过程**

   通过load-on-start标签指定的1，创建DispatcherServlet对象，

   DispatcherServlet它的父类是继承HttpServlert的，他是一个servlet,在被创建时，会执行init()方法。

   在init()方法中

   ```
   WebApplicationContexyt ctx = new
   ClassPathXMLAppliactionContext("springmvc.xml")
   //把容器对象放到ServletContext中
   getServletContext().setAttribute(key,ctx);
   ```

   上面创建容器作用：创建@Controller注解所在的类的对象，创建MyController对象

   这个对象放入到springmvc的容器中，容器是map，类似map.put(“MyController”,MyController对象)

2. **请求的处理过程**

   执行servlet的service等方法

   protected void **service** (HttpservletHservlet request ， Response response)

   protected void **doService** (HttpServletRequest request,HttpServletResponse response)

   this.**doDispatch** (request, response){

   调用MyController的doSome()方法

   } ;

配置视图解析器

声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的路径

MyController.java

```
modelAndView.setViewName("/show.jsp");
modelAndView.setViewName("/WEB-INF/view/show.jsp");
modelAndView.setViewName("/WEB-INF/view/other.jsp");
```

当配置了视图解析器后，可以使用逻辑名称（文件名），指定视图

框架会使用视图解析器的**前缀+逻辑名称+后缀** 组成全程路径，这里就是字符连接操作

springmvc.xml

```
<!--声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的路径-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!--    前缀 视图文件的路径-->
    <property name="prefix" value="/WEB-INF/view"/>
    <!--        后缀 视图文件的扩展名-->
    <property name="suffix" value=".jsp"/>
</bean>
```

MyController.java

```
modelAndView.setViewName("/WEB-INF/view/show.jsp");
//等同于
modelAndView.setViewName("show");
```

三、SpringMVC注解式开发

ReuestMapping放到类上做公共部分

```
//@RequestMapping:
//value :所有请求地址的公共部分，叫做模块名称
//位置放在类的上面

//Controller创建处理器对象，对象放在springmvc容器中
@Controller
@RequestMapping("/test")
public class MyController {

    @RequestMapping(value = "/some.do")
    public ModelAndView  doSome(){
        ......
    }

    @RequestMapping(value ={"/other.do"} )
    public ModelAndView  doOther(){
        ......
    }
}
```

```
//可接受http://localhost:8080/springmvc/test/some.do
//http://localhost:8080/springmvc/test/other.do
```

指定请求方式Method是POST/GET

属性:method 表示请求的方法。

它的值RequestMethod类枚举值 例如RequestMethod.GET

```
@RequestMapping(value = "/some.do",method = RequestMethod.GET)
public ModelAndView  doSome(){
    ....
}

@RequestMapping(value ={"/other.do"} ,method=RequestMethod.POST)
public ModelAndView  doOther(){
    .....
}
```

处理器方法的参数

处理器方法可以包含一下四类参数，这些参数会在系统调用时由系统自动赋值，即程序员可在方法内直接使用。

1. **HttpServletRequest**
2. **HttpServletResponse**
3. **HttpSession**

```
@RequestMapping(value ={"/other.do"} ,method=RequestMethod.POST)
public ModelAndView  doOther(HttpServletRequest request,
HttpServletResponse response, HttpSession session){
    ModelAndView modelAndView = new ModelAndView();

    modelAndView.addObject("msg","欢迎使用mvc");

    System.out.println(httpServletRequest.getParameter("name"));

    modelAndView.setViewName("other");
    return modelAndView;
}
```

4. **请求中所携带的请求参数**

    1. 逐个接受

    逐个接收请求参数
    要求：处理器方法的形参名和请求中参数名必须一致。
    同名的请求参数赋值给同名的形参
    框架接受请求参数
    1.**使用request对象接受请求参数**
    String strName = request.getParameter("name");
    String strAge = request.getParameter("age");
    2.springmvc框架通过DispatcherServlet 调用MyController的doSome()方法

调用方法时，按名称对应，把接受的参数赋值给形参
doSome（strName,Interger,valueOf(strage)）
框架会提供类型转换的功能，能把String转换成int,long,float等类型

400状态码是客户端错误，表示提交请求参数过程中，发生了问题。

```
@RequestMapping(value ={"/receiveproperties.do"}
,method=RequestMethod.POST)
public ModelAndView  receiveProperties(String name,Integer age){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("name",name);
    modelAndView.addObject("age",age);
    modelAndView.setViewName("other");
    return modelAndView;
}
```

2. **对象接受**

Student.class

```
//保存请求参数值的一个类
public class Student {
    //属性名和请求中参数名要求一样
    private  String name;
    private Integer age;
}
```

控制器方法

```
//    处理器方法形参是Java对象 这个对象的属性名和请求中的参数名一样的
//    框架会创建形参的Java对象，给属性赋值。请求中的参数是name,框架会调用
setName()
@RequestMapping(value ={"/receiveobject.do"} )
public ModelAndView  receiveObject(Student student){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("name",student.getName());
    modelAndView.addObject("age",student.getAge());
    modelAndView.setViewName("other");
    return modelAndView;
}
```

GET/POST乱码解决

注意:
在提交请求参数时,get请求方式中文没有乱码。
使用post方式提交请求，中文有乱码，需要使用过滤器处理乱码问题。

过滤器可以自定义也可以使用框架中的过滤器。CharacterEncodingFilter

```
<!--    注册声明过滤器，解决post乱码问题-->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <!--        设置项目中使用的字符编码-->
    <init-param>
```

```
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
    <!--强制请求对象（HttpservletRequest）)使用encoding编码的值-->
    <init-param>
        <param-name>forceRequestEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
    <!--强制应答对象（HttpservletResponse）使用encoding编码的值-->
    <init-param>
        <param-name>forceResponseEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

请求中参数名和处理器方法的形参名不一样

@RequestParam:解决请求中参数名形参名不一样的问题

属性:1.**value请求中的参数名称**

2.**required** 是一个boolean 默认时true
true表示请求中必须包含此参数

位置:在处理器方法的形参定义的前面

```
@RequestMapping(value ={"/receiveparam.do"} ,method=RequestMethod.POST)
public ModelAndView  receiveParam(@RequestParam(value = "pname") String name,
@RequestParam(value = "page") Integer age){
    ModelAndView modelAndView = new ModelAndView();
    System.out.println(name);
    modelAndView.setViewName("other");
    return modelAndView;
}
```

400--Required String parameter 'pname' is not present

当未提交参数页面的网址 重新打开一个新窗口会出现的报错。需要将required改为false

```
public ModelAndView  receiveParam(@RequestParam(value = "pname",required =
false) String name, @RequestParam(value = "page",required = false) Integer age){

}
```

处理器方法的返回值

处理器方法的返回值表示请求的处理结果

ModelAndView:有数据和视图，对视图执行forward

String:表示视图，可以逻辑名称，也可以是完整视图路径

void:不能表示数据，也不能表示视图。

在处理ajax的时候，可以使用void返回值。通过HttpServletResponse输出数据。相应ajax请求。

ajax请求服务器端返回的就是数据，和视图无关。

返回Object:例如String，Integer，Map,List等等都是对象

对象有属性，属性就是数据。所以返回Object表示数据，和视图无关

可以使用对象的数据，响应ajax请求

现在做ajax，主要使用json的数据格式。**实现步骤(重点)**：

1. **加入处理json的工具库的依赖，springmvc默认使用的jackson。**
2. **在sprigmvc配置文件之间加入mvq:annotation-driven注解驱动。**

   json = om.writevalueAsstring(student);
3. **在处理器方法的上面加入@ResponseBody注解**

   response.setcontentType("application/json;charset=utf-8");

   Printwriter pw = response.getwriter();

   pw.println(json);

springmvc处理器方法返回Object,可以转为json输出到浏览器，响应ajax的内部原理。

1. mvq:annotation-driven注解驱动

   注解驱动实现的功能是完成java对象到json , xml , text，二进制等数据格式的转换。

   mvq:annotation-driven在加入到springmvc配置文件后，会自动创建HttpMessageConverter接口的7个实现类对象，包括MappingJackson2HttpMessageConverter(使用jackson工具库中的ObjectMapper实现Java对象转为json)

   HttpMessageConveter接口:消息转换器。

   功能:定义了java转为json , xml等数据格式的方法。这个接口有很多的实现类。

   这些实现类完成java对象多json , java对象到xml ， java对象到二进制数据的转换

```
例如处理器方法
@RequestMapping(value = "/returnString.do")
public Student doReturnView2(HttpServletRequest request,String name, Integer age){
        Student student = new Student();
        student.setName("lisi");
        student.setAge(20);
        return student;
}

1)canWrite作用检查处理器方法的返回值，能不能转为var2表示的数据格式。
  检查student(lisi，20)能不能转为var2表示的数据格式。如果检查能转为json，canWrite返回true
  MediaType：表示数格式的， 例如json， xml等等

2)write：把处理器方法的返回值对象，调用jackson中的ObjectMapper转为json字符串。
   json  = om.writeValueAsString(student);
```

```
没有加入注解驱动标签时的状态
org.springframework.http.converter.ByteArrayHttpMessageConverter
org.springframework.http.converter.StringHttpMessageConverter
org.springframework.http.converter.xml.SourceHttpMessageConverter
org.springframework.http.converter.support.AllEncompassingFormHttpMessageConverter

加入注解驱动标签时的状态
org.springframework.http.converter.ByteArrayHttpMessageConverter
org.springframework.http.converter.StringHttpMessageConverter
org.springframework.http.converter.ResourceHttpMessageConverter
org.springframework.http.converter.ResourceRegionHttpMessageConverter
org.springframework.http.converter.xml.SourceHttpMessageConverter
org.springframework.http.converter.support.AllEncompassingFormHttpMessageConverter
org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter
org.springframework.http.converter.json.MappingJackson2HttpMessageConverter
```

2. @ResponseBody注解

   放在处理器方法的上面，通过HttpservletResponse输出数据，响应ajax请求的。

   response.setcontentType("application/json;charset=utf-8");

   Printwriter pw = response.getwriter();

   pw.println(json);

返回ModelAndVew

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。

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。

返回内部资源逻辑视图名

若要跳转的资源为内部资源,则视图解析器可以使用InternalResourceViewResolver内部资源视图解析器。此时处理器方法返回的字符串就是要跳转页面的文件名去掉文件扩展名后的部分。这个字符串与视图解析器中的prefix、suffix相结合，即可形成要访问的URI。

```
//    处理器方法返回String 表示逻辑视图名称 需要配置视图解析器
    @RequestMapping("/returnString.do")
    public String returnString(Student student,HttpServletRequest request){

        //可以自己手工添加数据到request作用域
        request.setAttribute("name",student.getName());
        request.setAttribute("age",student.getAge());
        //show:逻辑视图名称，项目中配置了视图解析器
        //框架对视图直送forward转发操作
        return "show";
    }

//处理器方法返回String,表示完整视图路径，此时不能配置视图解析器
@RequestMapping("/returnString2.do")
public String returnString2(Student student,HttpServletRequest request){
```

```
    //可以自己手工添加数据到request作用域
    request.setAttribute("name",student.getName());
    request.setAttribute("age",student.getAge());
    //完整视图路径，项目中配置了视图解析器
    //框架对视图直送forward转发操作
    return "WEB-INF/view/show.jsp";
    //报错 文.件[/WEB-INF/view/WEB-INF/view/show.jsp.jsp] 未找到
}
```

使用逻辑名称就必须使用视图解析器，反之就不能使用完整视图路径

返回Void

前端请求

```
<script type="text/javascript">
    $(function (){
        $("#submit").click(function (){
            $.ajax({
                url:"returnVoid.do",
                data:{
                    name:"刘强",
                    age:23
                },
                type:"POST",
                dataType:"json",
                success:function (data){
                    console.log(data)
                }
            })
        })
    })
</script>
```

控制器方法

```
//处理器方法返回String,表示完整视图路径，此时不能配置视图解析器
@RequestMapping("/returnVoid.do")
public void returnVoid(Student student,HttpServletResponse response) throws
IOException {
    //处理ajax,使用json做数据的格式 把结果的对象转为json格式的数据
        String json = "";
        if (student!=null){
            ObjectMapper objectMapper = new ObjectMapper();
            json = objectMapper.writeValueAsString(student);
            System.out.println("json: "+json);
        }
    //输出数据，响应ajax的请求
    response.setContentType("application/json;charset=utf-8");
    PrintWriter printWriter = response.getWriter();
    printWriter.println(json);
    printWriter.flush();
    printWriter.close();
}
```

返回Object

返回自定义类型对象

1.加入jackson依赖

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
```

2.加入mvc:annotation-driven/注解驱动

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

3.加入@ResponseBody注解

```
//    处理器方法返回一个Student，通过框架转为json,响应ajax请求
@RequestMapping(value = "returnVoid2.do")
@ResponseBody
public Student returnObject(Student student){
    return student;//会被框架转为json
}
```

返回JsonArray

```
// 处理器返回List<Student>
@RequestMapping(value = "returnVoid3.do")
@ResponseBody
public List<Student> returnObject3(Student student){
    List<Student> studentList = new ArrayList<>();
    studentList.add(student);
    studentList.add(student);
    studentList.add(student);
    return studentList;
}
```

返回字符串对象

```
// 处理器参会String ,String表示数据的,不是视图
//         区分返回值Strin是数据还是视图,看有没有@Responsebody注解
//         如果有@Responsebody注解,返回String就是数据,反之就是视图
//    默认使用text/plain;charset=ISO-8859-1作为ContentType,导致中文有乱码
//    给RequestMapping增加一个属性produces,使用这个刷新指定新的contentType
@RequestMapping(value = "returnVoid4.do",produces = "text/plain;charset=utf-8")
@ResponseBody
public String returnString2(Student student){
    return "处理器参会String ,String表示数据的,不是视图";
}
```

解读<url-pattern/>

研究中央调度器的url-pattern设置为"/"

```
http://localhost:8080/url_pattern/index.jsp    //tomcat (jsp会转为servlet)
http://localhost:8080/url_pattern/js/jquery-1.7.2.js   //tomcat
http://localhost:8080/url_pattern/images/p1.png    //tomcat
http://localhost:8080/url_pattern/html/test.html    //tomcat
http://localhost:8080/url_pattern/some.do   //DispatcherServlet(springmvc框架处理的)
```

由此可得：tomcat本身能够处理静态资源的访问，像html,图片，js文件都是静态资源

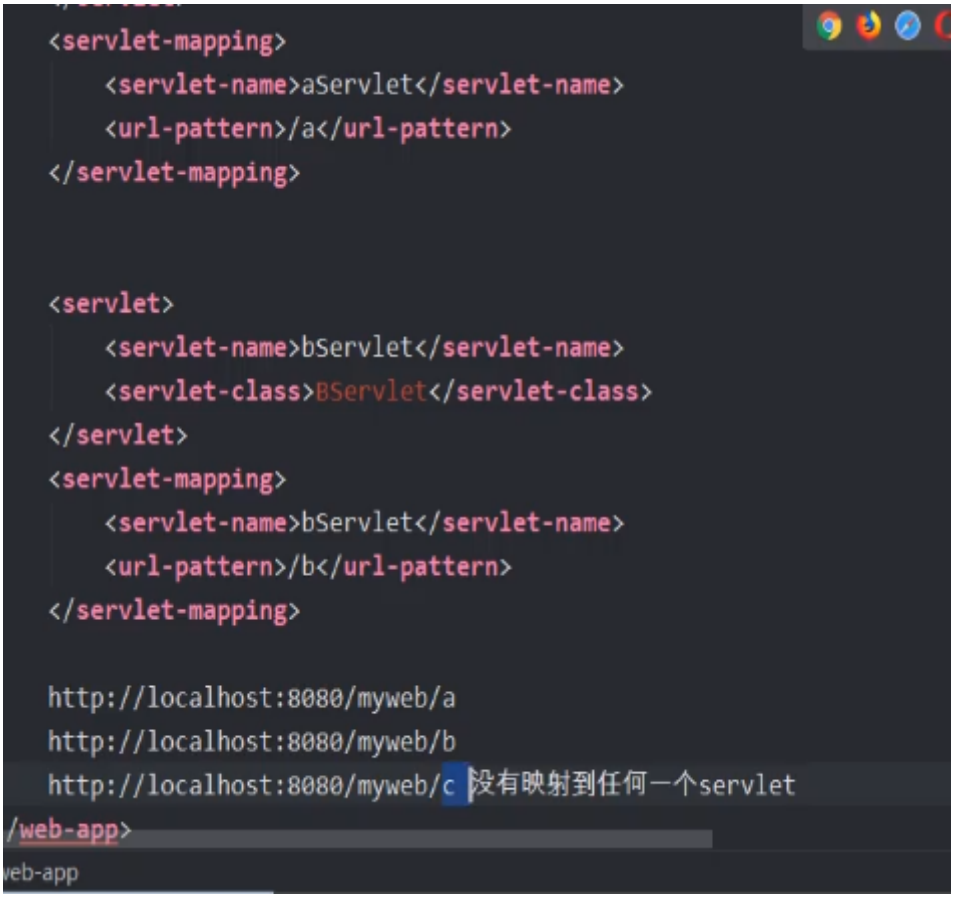
tomcat的web.xml文件有一个servlet 名称是 default ， 在服务器启动时创建的。

```
 <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>0</param-value>
        </init-param>
        <init-param>
            <param-name>listings</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

default这个servlet作用：

```
The default servlet for all web applications, that serves static
resources.  It processes all requests that are not mapped to other
servlets with servlet mappings (defined either here or in your own
web.xml file).
```

1. 处理静态资源、
2. 处理未映射到其他servlet的请求



所以使用斜杠去替代tomcat的default！这个时候所有静态资源就会出现404

```
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!--        使用框架的时候，url-pattern可以使用两种值-->
    <!--        1.扩展名的方式，语法 *.xxxx,xxxx是自定义的扩展名。常用的方式 *.do
*.action *.mvc等-->
    <!--            http://localhost:8080/springmvc/springmvc1.do-->
    <!--            http://localhost:8080/springmvc/springmvc2.do-->
    <!--        2.使用斜杠 “/"-->
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

导致所有的静态资源都给DispatcherServlet处理，默认情况下DispatcherServlet没有处理静态资源的能力。没有控制器对象能处理静态资源的访问。所以静态资源（html , js ,图片,css）都是404.

动态资源some.do

静态资源some.do ✕

第一种处理静态资源的方式mvc:default-servlet-handler/

需要加springmvc配置文件mvc:default-servlet-handler

原理是：加入这个标签后，框架会创建控制器对象DefaultservletHttpRequestHandler

DefaultServletHttpRequestHandler这个对象可以把接收的请求转发给 tomcat的default这个servlet,

```
<!--     加入注解驱动-->
<!-- default-servlet-handler和@RequestNapping注解有冲突，需要加入annotation-driven解决问题-->
<mvc:annotation-driven/>

<mvc:default-servlet-handler/>
```

第二种处理静态资源的方式mvc:resources/

mvc：resources 加入后框架会创建 ResourceHttpRequestHandler 处理器对象

让这个对象处理静态资源的访问，不依赖tomcat服务器

mapping :访问静态资源的uri地址，使用通配符**

location： 静态资源在你的项目中的目录位置

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

<mvc:annotation-driven/>
```

一条配置处理所有的静态资源

创建static目录，将js,html，images目录全放进去

```
<mvc:resources mapping="/static/**" location="/static/"></mvc:resources>
```

绝对路劲和相对路径

在你的页面中的,访问地址不加" /"

访问:http://localhost:8080/ch06path/index.jsp

路径:http://localhost:8080/ch06path/

资源: index.jsp

在index.jsp发起user/some.do请求，访问地址变为 http://localhost:8080/ch06path/user/some.do

当你的地址没有斜杠开头

例如**user/some.do**，当你点击链接时，访问地址是当前页面的地址加上链接的地址。即http://localhost:8080/ch06path/ + **user/some.do**

当你的地址没有斜杠开头,例如user/some.do

但是!

解决方案：

1. 加入${pageContext.request.contextPath}
2. 加入一个base标签，是html语言中的标签。表示当前页面中访问地址的基地址。

   你的页面中所有没有"/"开头的地址，都是以base标签中的地址为参考地址使用base中的地址+use/some.do

```
<%
    String basePath = request.getScheme()+
            "://"+request.getServerName()+":"+
            request.getServerPort()+
            request.getContextPath()+"/";
%>
<html>
<base href="<%=basePath%>">
```

当你的地址有斜杠开头

/user/some.do,参考地址就是你服务器的地址，即http://localhost:8080。即http://localhost:8080/ + **/user/some.do。**

如果资源不能访问，**需要加入${pageContext.request.contextPath}**

```
<a href="${pageContext.request.contextPath}/user/some.do">/user/some.do</a>
```

四、SSM整合开发

SSM = SpringMVC+Spring+Mybatis

SpringMVC：视图层，界面层，负责接受请求，显示处理结果的。

Spring:业务层，管理service,dao，工具类对象的

MyBatis:持久层，访问数据库的

用户发起请求--springMVC接收--spring中的service对象--MyBatis处理数据

SSM整合也叫SSI，整合中有容器。

1. 第一个容器**SpringMVC容器，管理Controller控制器对象的。**
2. 第二个容器**Spring容器，管理service,dao工具类对象的。**
3. 我们要做的把使用的对象交给合适的容器创建，管理。把Controller还有web开发的相关对象交给springmvc容器，这些web用的对象卸载springmvc配置文件中

   service，dao对象定义在spring的配置文件中，让spring管理这些对象

   springmvc容器和spring容器是有关系的。**springmvc容器是spring的子容器**，类似继承

   在子容器的Controller可以访问父容器中的Service对象，就可以实现controller使用service对象

实现步骤：

1. 使用springdb的mysql,表示用student
2. 新建maven项目

3. 加入依赖

springmvc , spring , mybatis三个框架的依赖，jackson依赖，mysql驱动 ， druid连接池，jsp , servlet依赖

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.qiangliu8</groupId>
  <artifactId>ssm</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>


  <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>
  </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>javax.annotation</groupId>
            <artifactId>javax.annotation-api</artifactId>
            <version>1.3.2</version>
        </dependency>
        <!--servlet依赖-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
        <!-- jsp依赖 -->
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.2.1-b03</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.5.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-tx</artifactId>
            <version>5.2.5.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.2.5.RELEASE</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-databind</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.1</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.1</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.13</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.12</version>
        </dependency>
    </dependencies>

    <build>
        <resources>
            <resource>
                <directory>src/main/java</directory><!--所在的目录-->
                <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
                    <include>**/*.properties</include>
                    <include>**/*.xml</include>
                </includes>
                <filtering>false</filtering>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
```

```
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

4. 写web.xml

    1. 注册DispatcherServlet，目的：1.创建springmvc容器对象，才能创建Controller类对象。

                                                2.创建的是servlet,才能接受用户的请求

    ```
    <?xml version="1.0" encoding="UTF-8"?>
    <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_4_0.xsd"
             version="4.0">
        <servlet>
            <servlet-name>myweb</servlet-name>
            <servlet-
    class>org.springframework.web.servlet.DispatcherServlet</servlet-
    class>
            <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>classpath:conf/springmvc.xml</param-
    value>
            </init-param>
            <load-on-startup>1</load-on-startup>
        </servlet>
    ```

    2. 注册spring的监听器：ContextLoaderListener,目的：创建spring的容器对象，才能创建service,dao等对象

    ```
    <!--注册spring的监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:conf/spring.xml</param-value>
    </context-param>
    <listener>
        <listener-
    class>org.springframework.web.context.ContextLoaderListener</list
    ener-class>
    </listener>
    ```

    3. 注册字符集过滤器，解决post请求乱码的问题

    ```
    <!--    注册声明过滤器，解决post乱码问题-->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-
    class>org.springframework.web.filter.CharacterEncodingFilter</fil
    ter-class>
        <!--         设置项目中使用的字符编码-->
        <init-param>
    ```

```
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
    <!--强制请求对象（HttpservletRequest）)使用encoding编码的值-->
    <init-param>
        <param-name>forceRequestEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
    <!--强制应答对象（HttpservletResponse）使用encoding编码的值-->
    <init-param>
        <param-name>forceResponseEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

4. 创建包，Controller包，service,dao，实体类包名创建好

5. 写springmvc,spring,mybatis的配置文件

    1. springmvc配置文件

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd">


    <context:component-scan base-
package="com.qiangliu8.controller"/>

    <!--    声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的
路径-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewR
esolver">
        <!--    前缀 视图文件的路径-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--        后缀 视图文件的扩展名-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <mvc:annotation-driven />
    <!--
      1. 响应ajax请求，返回json
      2. 解决静态资源访问问题。
```

```
    -->
</beans>
```

2. spring配置文件

```
<?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
https://www.springframework.org/schema/context/spring-
context.xsd">

    <!--spring配置文件： 声明service，dao，工具类等对象-->

    <context:property-placeholder
location="classpath:conf/jdbc.properties" />

    <!--声明数据源，连接数据库-->
    <bean id="dataSource"
class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>

    <!--SqlSessionFactoryBean创建SqlSessionFactory-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation"
 value="classpath:conf/mybatis.xml" />
    </bean>

    <!--声明mybatis的扫描器，创建dao对象-->
    <bean
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactory" />
        <property name="basePackage" value="com.qiangliu8.dao" />
    </bean>

    <!--声明service的注解@Service所在的包名位置-->
    <context:component-scan base-package="com.qiangliu8.service"
/>

    <!--事务配置：注解的配置， aspectj的配置-->
</beans>
```

3. mybatis主配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--settings：控制mybatis全局行为-->
    <!-- <settings>
         &lt;!&ndash;设置mybatis输出日志&ndash;&gt;
         <setting name="logImpl" value="STDOUT_LOGGING"/>
     </settings>-->

    <!--设置别名-->
    <typeAliases>
        <!--name:实体类所在的包名(不是实体类的包名也可以)-->
        <package name="com.qiangliu8.domain"/>
    </typeAliases>


    <!-- sql mapper(sql映射文件)的位置-->
    <mappers>
        <!--
          name：是包名， 这个包中的所有mapper.xml一次都能加载
          使用package的要求：
           1. mapper文件名称和dao接口名必须完全一样，包括大小写
           2. mapper文件和dao接口必须在同一目录
        -->
        <package name="com.qiangliu8.dao"/>
    </mappers>
</configuration>
```

4. 数据库的属性配置文件

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url =jdbc:mysql://localhost:3306/springdb?serverTimezone=UTC
jdbc.username=root
jdbc.password=Lq060528
```

6. 写代码，dao接口和mapper文件，service和实现类，controller，实体类

controller

```
package com.qiangliu8.controller;

import com.qiangliu8.domain.Student;
import com.qiangliu8.service.StudentService;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.ModelAndView;

import javax.annotation.Resource;
import javax.servlet.http.HttpServletRequest;
import java.util.List;

@Controller
```

```java
@RequestMapping("/student")
public class StudentController {

    @Resource
    private StudentService service;

    //注册学生

    @RequestMapping("/addStudent.do")
    public ModelAndView addStudent(Student student){
        ModelAndView modelAndView = new ModelAndView();
        String code = "注册失败";
        //调用service处理student
        int nums = service.addStudent(student);
        if( nums > 0 ){
            //注册成功
            code = "学生【" + student.getName() + "】注册成功";
        }
        modelAndView.addObject("code",code);
        modelAndView.setViewName("forward:/WEB-INF/jsp/result.jsp");

        System.out.println(modelAndView);

        return modelAndView;
    }

    //处理查询，响应ajax
    @RequestMapping("/queryStudent.do")
    @ResponseBody
    public List<Student> queryStudent(){
        //参数检查， 简单的数据处理
        List<Student> students = service.findStudents();
        return students;
    }
}
```

mapper文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.qiangliu8.dao.StudentDao">
    <select id="selectStudents" resultType="Student">
        select id,name,age from students order by id desc
    </select>

    <insert id="insertStudent">
        insert into students(name,age) values(#{name},#{age})
    </insert>
</mapper>
```

service类

```java
@Service
public class StudentServiceImpl implements StudentService {
```

```
    //引用类型自动注入@Autowired, @Resource
    @Resource
    private StudentDao studentDao;

    @Override
    public int addStudent(Student student) {
        int nums = studentDao.insertStudent(student);
        return nums;
    }

    @Override
    public List<Student> findStudents() {
        return studentDao.selectStudents();
    }
}
```

7. 写jsp页面

五、SpringMVC核心技术

请求重定向

SpringMVC框架把原来Servlet中的请求转发和重定向操作进行了封装。现在可以使用简单的方式实现转发和重定向。

forward：表示转发，实现request.getRequestDispatcher("xx.jsp").forward();

redirect：表示重定向，实现response.sendRedirect("xx.jsp")

共同特点：都是关键字，都不和视图解析器一起工作。

forward

```
//    处理器方法返回ModeLAndview,实现forward
@RequestMapping("/doForward.do")
public ModelAndView doForward(){
    ModelAndView mv = new ModelAndView();
    mv.addObject("msg","欢迎使用springmvc");
    mv.addObject("fun","doForward");
    //显示转发
    //mv.setViewName("forward:/WEB-INF/view/show.jsp");

    mv.setViewName("forward:/other.jsp");
    return mv;
}
```

它还可以跳转到视图解析器解析不到的地方 比如web-inf文件夹外面

redirect

```
//    处理器方法返回ModeLAndview,实现重定向redirect
//    redirect特点:不和视图解析器一同使用，就当项目中没有视图解析器
//        框架对重定向的操作：
//    框架会把Model中的简单类型的数据，转为Strings使用，作为hello.jsp的get请求参数使用
//    目的是：可以在doRedirect.do 和hello。jsp两次请求之间传递数据
```

```
//在目标helLo.jsp页面可以使用参数集合对象${param]获取请求参数值s{param.getParamter("参数名")}

@RequestMapping("/doRedirect.do")
public ModelAndView doRedirect(){
    ModelAndView mv = new ModelAndView();
    //数据放入到 request作用域
    mv.addObject("msg","欢迎使用springmvc");
    mv.addObject("fun","doForward");
    //         显示转发
    mv.setViewName("redirect:/show.jsp");
    //http://localhost:8080/forward_war_exploded/show.jsp?msg=%E6%AC%A2%E8%BF%8E%E4%BD%BF%E7%94%A8springmvc&fun=doForward
    return mv;
}
```

取参

```
<h3>取参数数据：<%=request.getParameter("msg")%>></h3>
<h4>msg数据：${param.msg}</h4>
<h4>fun数据：${param.fun}</h4>
```

但是：**redirect不可访问WEB-INF文件下的资源**

异常处理

springMVC框架处理异常的常用方式：使用@ExceptionHandler注解处理异常

springmvc框架采用的是统一的，全局的异常处理。

把controller中所有异常处理都集中到一个地方。**采用AOP的思想**，把业务逻辑和异常处理代码分开。解耦合。

使用两个注解

1. @ExceptionHandler
2. @ControllerAdvice

异常处理步骤

1. **新建maven web项目**
2. **加入依赖**
3. **新建一个自定义异常类 MyUserException,再定义它的子类NameException,AgeException**

```
public class MyUserException extends Exception{
    public MyUserException(){
        super();
    }

    public MyUserException(String message) {
        super(message);
    }
}
```

```
//当用户的年龄有异常
public class AgeException extends MyUserException{
    public AgeException() {
        super();
    }

    public AgeException(String message) {
        super(message);
    }
}
```

```
//当用户的姓名有异常
public class NameException extends MyUserException{
    public NameException() {
        super();
    }

    public NameException(String message) {
        super(message);
    }
}
```

4. **创建一个普通类，作用全局异常处理类**
    1. 在类的上面加入@ControllerAdvice
    2. 在类中定义方法，方法的上面加入@ExceptionHandler

```
/*
* @ControllerAdvice：控制器增强，也就是控制器类增加功能--异常处理功能
* 特点：必须让框架知道这个注解所在的包名，需要再springmcv配置文件声明组件扫描器
* 指定@ControllerAdvice所在的包名
* */
@ControllerAdvice
public class GlobalExceptionHandler {
    //定义方法，处理发生的异常
    /*
    * 处理异常的方法和控制器方法一样，可有有多个参数，可以有ModelAndView
    * String,void 对象类型的返回值
    * 形参：Exception 表示Controller中抛出的异常对象
    * 通过形参可以获取发生的异常信息
    * */
    /*@ExceptionHander(异常的class) 表示异常的类型，当发生此类型异常时，由当前方法处理
    * */
    @ExceptionHandler(value = NameException.class)
    public ModelAndView doNameException(Exception exception){
        //处理NameException的异常
        /*异常发生处理逻辑
        * 1.需要把异常记录下来，记录到数据库，日志文件
        * 记录日志发生的时间，哪个方法发生的，异常错误内容
        * 2.发送通过，把异常的信息通过邮件发送给相关人员
        * 3.给用户友好提示
        * */

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("msg","姓名必须是张三，其它不能访问");
```

```
        modelAndView.addObject("exception",exception);
        modelAndView.setViewName("nameError");
        return modelAndView;
    }

    @ExceptionHandler(value = AgeException.class)
    public ModelAndView doAgeException(Exception exception){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("msg","年龄不能小于0 也不能为空");
        modelAndView.addObject("exception",exception);
        modelAndView.setViewName("ageError");
        return modelAndView;
    }
    //处理其他异常，上面两个以外的异常  只能由一个这样的方法
    @ExceptionHandler
    public ModelAndView doOtherException(Exception exception){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("msg","其他异常");
        modelAndView.addObject("exception",exception);
        modelAndView.setViewName("otherError");
        return modelAndView;
    }

}
```

5. **在Controller抛出NameExcettion,AgeException**

```
@Controller
public class MyController {

    @RequestMapping("/some.do")
    public ModelAndView doSome(String name,Integer age) throws
MyUserException {
        ModelAndView mv = new ModelAndView();

        //根据请求参数抛出异常
        if (!"zs".equals(name)){
            throw new NameException("姓名不正确！");
        }
        if (age==null||age<0){
            throw new AgeException("性别错误！");
        }

        mv.addObject("name",name);
        mv.addObject("age",age);
        //显示转发
        mv.setViewName("forward:/WEB-INF/jsp/show.jsp");

        return mv;
    }
```

6. **创建处理异常的视图界面**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<h3>年龄异常</h3>
<h4>msg数据：${msg}</h4>
<h4>exception数据：${exception}</h4>
</body>
</html>
```

7. **创建springmvc的配置文件**

    1. 组件扫描器，扫描Controller注解
    2. 组件扫描器，扫描ControllerAdvice所在包名
    3. 声明注解驱动

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd">


    <context:component-scan base-package="com.qiangliu8.controller"/>

    <!--    声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的路径-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
>
        <!--    前缀 视图文件的路径-->
        <property name="prefix" value="/WEB-INF/exception/"/>
        <!--        后缀 视图文件的扩展名-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <mvc:annotation-driven />
    <!--
      1. 响应ajax请求，返回json
      2. 解决静态资源访问问题。
    -->
    <context:component-scan base-package="com.qiangliu8.handler"/>
</beans>
```

拦截器

1. 拦截器是Srpingmvc中的一种，需要实现HandlerInterceptor接口
2. 拦截器和过滤器类似，但功能方向侧重点不同。

   过滤器是用来过滤请求参数，设置编码字符集等工作。

   拦截器是拦截用户的秦桧去，对请求做判断处理
3. 拦截器是全局的，可以对多个Controller做拦截。

   一个项目中有0至多个，一起拦截用户的请求。
4. 拦截器常用在：用户登录处理，权限检查，记录日志。

拦截器的使用步骤

1. 定义类实现HanlderInterceptor接口
2. 在springmvc配置文件中，声明拦截器。

拦截器的执行时间

1. 在请求处理之前，也就是Controller类中的方法执行之前先被拦截
2. 在控制器方法执行之后，也会执行拦截器。
3. 在请求处理完成后，也会执行拦截器。

拦截器的实现步骤

1. **新建maven web项目**
2. **加入依赖**
3. **创建Controller类**

```
@Controller
public class MyController {

    @RequestMapping("/some.do")
    public ModelAndView doSome(String name,Integer age){
        System.out.println("====执行Controller中的doSome方法====");
        ModelAndView mv = new ModelAndView();
        mv.addObject("name",name);
        mv.addObject("age",age);
        mv.setViewName("show");
        return mv;
    }
}
```

4. **创建一个普通类，作为拦截器使用**
   1. 实现HandlerInterceptor接口
   2. 实现接口中的三个方法

```
//拦截器类：拦截用户的请求
public class MyInterceptor implements HandlerInterceptor {
    private long btime = 0;
    /*
    preHandle叫做预处理方法
    参数：
    Object handler 被拦截的控制器对象 返回值是bool值
    特点：
```

```
        1.方法在控制器方法（MyController的doSome）之前先执行的
         用户的请求首先到达此方法

         2.可以验证用户是否登录，验证用户是否有权限访问某个连接地址( url)如果验证失
败，
         可以截断请求，请求不能被处理。
         如果验证成功，可以放行请求，此时控制器方法才能执行
    */
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        btime = System.currentTimeMillis();
        System.out.println("preHandle方法");

 //request.getRequestDispatcher("/tips.jsp").forward(request,response);
        return true;
    }
    /*
    preHandle叫做后处理方法
    参数:
    Object handler 被拦截的控制器对象
    ModelAndView mv :处理器方法的返回值
    特点:
        1.方法在控制器方法（MyController的doSome）之后执行的
        2.能够获取到处理器方法的返回值ModelAndView，可以修改ModelAndView中的数据
和视图，可以影响到最后的执行结果
        3.主要对原来的执行结果做二次修正
    */
    @Override
    public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView modelAndView)
throws Exception {
        System.out.println("postHandle方法");
        if (modelAndView!=null){
            modelAndView.addObject("mydate",new Date());
            modelAndView.setViewName("other");
        }
    }
    /*
    afterCompletion
    参数:
    Object handler 被拦截的控制器对象
    Exception ex 程序中发生的异常
    特点:
        1.在请求处理完成后执行的。规定当你的是太古处理完成之后，对试图执行了
forward。就认为请求处理完成
        2.一般做资源回收工作，程序请求过程中创建了一些对象，在这里可以删除，把占用的
内存回收。
    */
    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("afterCompletion方法");
        long etime = System.currentTimeMillis();
        System.out.println("共花费时间为"+(etime-btime));
    }
}
```

5. **创建show.jsp**
6. **创建springmvc的配置文件**
    1. 组件扫描器，扫描Controller注解
    2. 声明拦截器，并指定拦截器的请求uri地址

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd">


    <context:component-scan base-package="com.qiangliu8.controller"/>

    <!--    声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的路径-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
>
        <!--    前缀 视图文件的路径-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--        后缀 视图文件的扩展名-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <mvc:annotation-driven />
    <!--
      1. 响应ajax请求，返回json
      2. 解决静态资源访问问题。
    -->
    <!--声明拦截器组-->
    <mvc:interceptors>
        <!--声明第一个拦截器-->
        <mvc:interceptor>
            <!--指定拦截的请求uri地址-->
            <mvc:mapping path="/**"/>
            <!--声明拦截器对象-->
            <bean class="com.qiangliu8.handler.MyInterceptor"></bean>
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**"/>
            <bean class="com.qiangliu8.handler.MyInterceptor2"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
</beans>
```

多个拦截器执行顺序

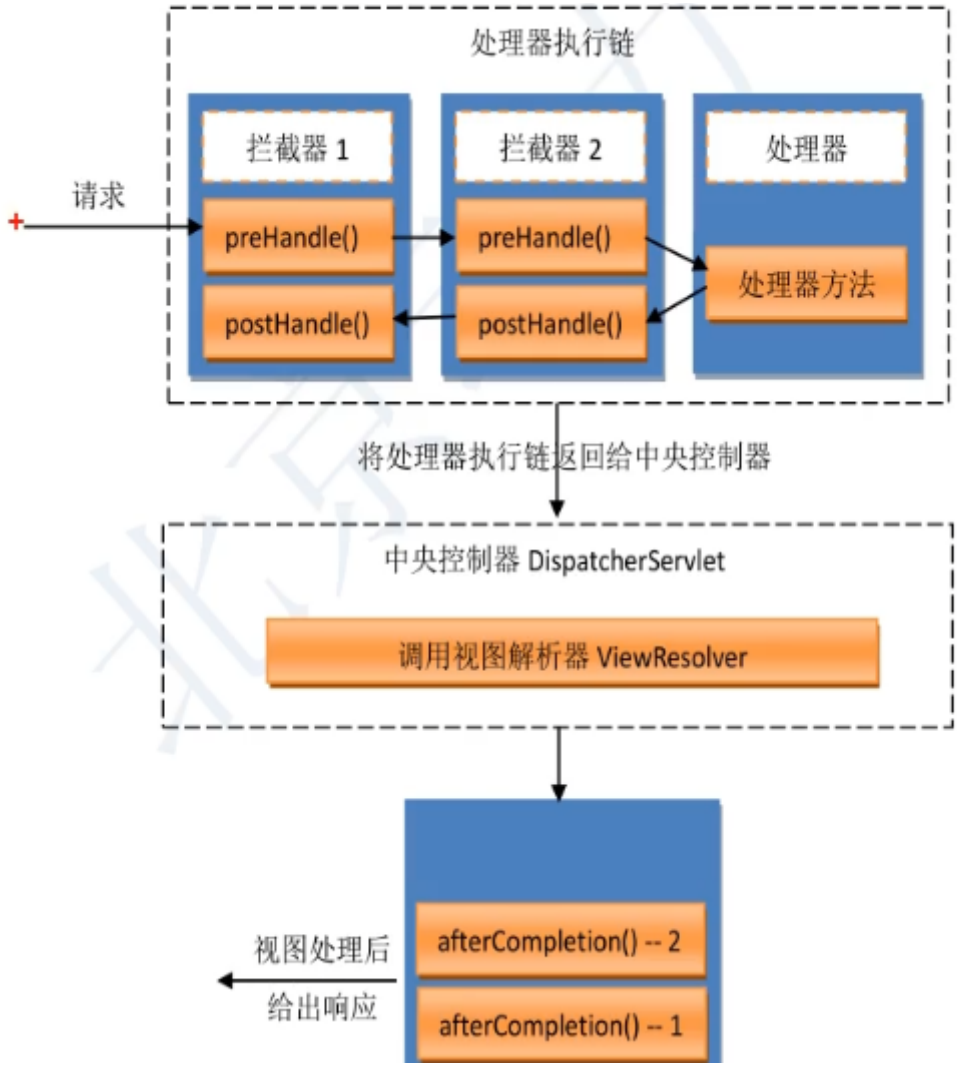
在框架中保存多个拦截器是ArrayList

按照声明的先后顺序放到ArrayList

```
<mvc:interceptors>
    <!--声明第一个拦截器-->
    <mvc:interceptor>
        <!--指定拦截的请求uri地址-->
        <mvc:mapping path="/**"/>
        <!--声明拦截器对象-->
        <bean class="com.qiangliu8.handler.MyInterceptor"></bean>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.qiangliu8.handler.MyInterceptor2"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

执行情况:

```
1111preHandle方法
2222preHandle方法
====执行Controller中的doSome方法====
2222postHandle方法
1111postHandle方法
2222afterCompletion方法
2222共花费时间为262
1111afterCompletion方法
1111共花费时间为262
```



当第一个拦截器preHandle返回true,第二个拦截器preHandle返回false，打印：

1111preHandle方法

2222preHandle方法

111afterCompletion方法

当第一个拦截器preHandle返回true,第二个拦截器preHandle返回false/true，打印：

1111preHandle方法

拦截器和过滤器的区别

过滤器	拦截器
过滤器是servlet中的对象	拦截器是框架中的对象
过滤器实现Filter接口	拦截器是实现HandlerInterceptor
过滤器设置request,response的参数、属性。侧重对数据过滤的	拦截器用来验证请求，能截断请求
过滤器在拦截器之前先执行的	
过滤器是tomcat服务器创建的对象	拦截器是springmvc容器中创建的对象
过滤器只有一个执行时间点	拦截器有三个执行时间点
过滤器可以处理jsp,js,html	拦截器侧重拦截对Controller对象。如果你的请求不能被Dispathcerservlet接收，这个请求不会执行拦截器内容

登录验证拦截器

1. **新建maven web项目**
2. **新建index.jsp发起请求**

```
<p>一个拦截器</p>
<form action="some.do" method="post">
    <input type="text" name="name" value="name"/>
    <input type="text" name="age" value=123/>
    <input type="submit" value="提交"/>
</form>
```

3. **创建MyController处理请求**

```
@Controller
public class MyController {

    @RequestMapping("/some.do")
    public ModelAndView doSome(String name,Integer age){
        System.out.println("====执行Controller中的doSome方法====");
        ModelAndView mv = new ModelAndView();
        mv.addObject("name",name);
        mv.addObject("age",age);
        mv.setViewName("show");
        return mv;
    }
}
```

4. **创建结果show.jsp**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>show.jsp从request作用域中获取数据</h3>
    <h4>name数据：${name}</h4>
    <h4>age数据：${age}</h4>
</body>
</html>
```

5. **创建一个login.jsp，模拟登录（把用户的信息放入到session）；**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    模拟登录
    <%  session.setAttribute("name","zs");
    %>
</body>
</html>
```

创建一个jsp, logout.jsp,模拟退出系统（从session中删除数据）

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
    <head>
        <title>Title</title>
    </head>
    <body>
        模拟推出
        <%  session.removeAttribute("name");%>
    </body>
</html>
```

6. **创建拦截器,从session中获取用户的登录数据,验证能否访问系统**

```
//拦截器类：拦截用户的请求
public class MyInterceptor implements HandlerInterceptor {
    //验证登录的用户信息
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {

        System.out.println("1111preHandle方法");
        Object attr = request.getSession().getAttribute("name");
        String loginName = "";
        if (attr!=null){
            loginName = (String) attr;
        }
        if ("zs".equals(loginName)){
```

```
            return true;
        }

 request.getRequestDispatcher("/tips.jsp").forward(request,response);
        return false;
    }

}
```

7. **创建一个验证的jsp，如果验证视图，给出提示**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
    <head>
        <title>Title</title>
    </head>
    <body>
        tips请求被拦截，不能被执行
    </body>
</html>
```

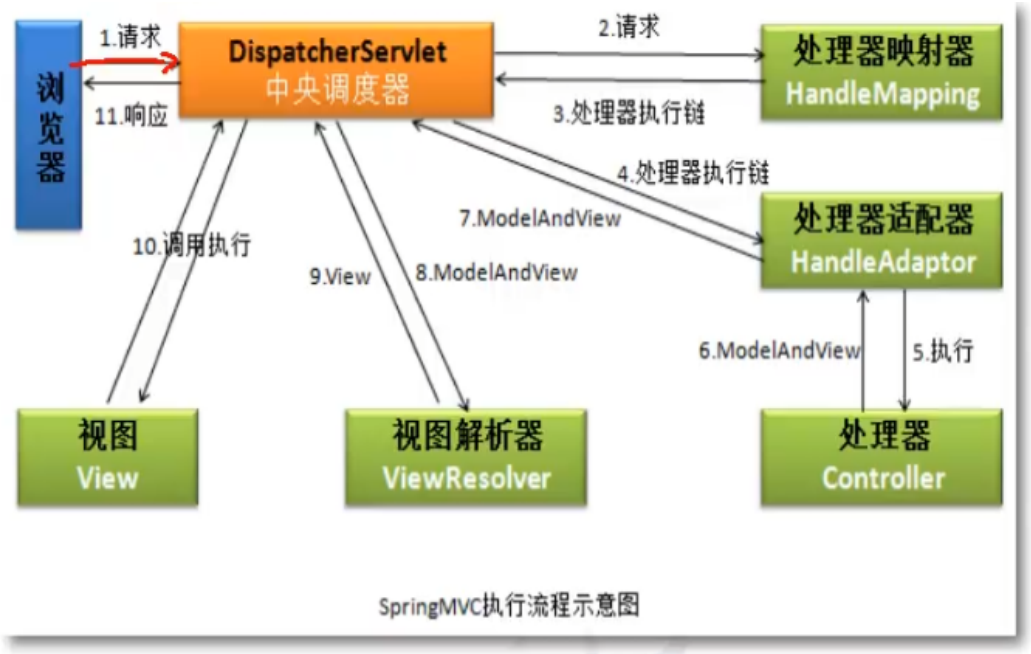
8. **创建springmvc的配置文件**

    1. 组件扫描器，扫描Controller注解
    2. 声明拦截器，并指定拦截器的请求uri地址

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd">


    <context:component-scan base-package="com.qiangliu8.controller"/>

    <!--    声明 springmvc框架中的视图解析器，帮助开发人员设置视图文件的路径-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
>
        <!--    前缀 视图文件的路径-->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!--        后缀 视图文件的扩展名-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <mvc:annotation-driven />
    <!--
      1. 响应ajax请求，返回json
      2. 解决静态资源访问问题。
    -->
    <!--声明拦截器组-->
```

```
    <mvc:interceptors>
        <!--声明第一个拦截器-->
        <mvc:interceptor>
            <!--指定拦截的请求uri地址-->
            <mvc:mapping path="/**"/>
            <!--声明拦截器对象-->
            <bean class="com.qiangliu8.handler.MyInterceptor"></bean>
        </mvc:interceptor>

    </mvc:interceptors>
</beans>
```

springMVC执行流程



内部请求的处理流程:

1. 用户发起请求some.do
2. **DispatcherServlet**接受请求some.do，把请求转交给处理器映射器

   处理器映射器：sprinmvc框架中的一种对象，框架把实现HandlerMapping接口的类叫做映射器

   处理器映射器作用：

   1. 根据请求，从springmvc容器对象中获取处理器对象（MyController controller = ctx.getBean("some.do")。
   2. 框架把找到的处理器对象放到一个叫做处理器执行链（HandlerExecutionChain）的类保存。

      HandlerExecutionChain类中保存着：处理器对象和项目中所有的拦截器对象
3. **DispatcherServlet**把2中的**HandlerExecutionChain**中的处理器对象交给了处理器适配器对象（多个）。

   处理器适配器对象：springmvc框架中的对象，需要实现HandlerAdapter接口

   处理器适配器作用：执行处理器方法（调用MyController.doSome()得到返回值ModelAndView）
4. **DispatcherServlet**把3中获取的ModelAndView交给了视图解析器对象

视图解析器：springmvc中的对象，需要实现ViewResoler接口（可以有多个）

视图解析器作用：组成视图完整路径，使用前缀，后缀。并创建View对象

View是一个接口，表示视图的。在框架中

InternalResourceView:视图类，表示jsp文件。视图解析器会创建InternalResourceView类对象。

这个对象里面，有一个属性url = /WEB-INF/view/show.jsp

5. **DispatcherServlet**把4步骤中创建的view对象获取到，调用View类自己的方法，把Model数据放入到request作用域。执行对象视图的forward。请求结束.