SpringMVC测试

- MVC概念
  - M：Model，模型层主要是工程中的javaBean
    - javaBean
      - 实体类Bean：存储数据，mybatis中的User
      - 业务处理Bean：Service或DAO对象
  - V：VIew，视图层，主要值工程中html代码等
  - C：Controller，控制层，工程中的servlet，主要与用户交互
- SpringMVC简介
  - 开发底层：servlet
  - 本身是spring，而spring是容器，IOC管理对象，使用注解创建控制器对象，SpringMVC则是控制器容器，而注解本身（Controller）创建的对象是一个普通的对象，而SpringMVC会赋予它servlet的功能。
  - DispatcherServlet（中央调度器，前端控制器）
    - DispatcherServlet的父类时httpservlet
    - springMVC提供的一个servlet，网页发送请求给DispatcherServlet，DispatcherServlet将请求转发给Controller对象（@Controller创建的对象，可以有很多个）处理，响应同理
  - springmvc是spring的一部分，可以使用spring中的IOC和AOP，可以整合mybatis等多种框架
- Maven依赖
  - MVC-web依赖，日志包，servlet包，Thymeleaf和spring5的整合包

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>



    <!-- https://mvnrepository.com/artifact/org.springframework/spring-
webmvc
Springmvc的依赖-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.3.18</version>
    </dependency>

  <!-- https://mvnrepository.com/artifact/ch.qos.logback/logback-classic
日志（可选）
-->
```

```
  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
    <scope>test</scope>
  </dependency>

  <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-
api
servlet包
-->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.thymeleaf/thymeleaf-
spring5
thymeleaf和spring的整合包（可选）
-->
  <dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
    <version>3.0.12.RELEASE</version>
  </dependency>

</dependencies>
```

- SpringMVC的基本使用步骤
  - (1）web.xml的编写
  - web.xml文件配置springmvc的两种方式
  - a>默认配置方式

```
<servlet>
    <servlet-name>spring_mvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>spring_mvc</servlet-name>
    <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

    - org.springframework.web.servlet.DispatcherServlet是DispatcherServlet的全包名
    - url-pattern中“/”代表的是包含除了.jsp文件以外的所有请求，主要是因为jsp本身就是servlet，如果mvc接受到jsp文件网页将不会表现jsp的信息
    - 这种配置方法的springmvc配置文件将默认放在WEB-INF下其名称为对应的-servlet.xml,所以这种方式较少使用
  - b>扩展配置方式

```

```

```
<servlet>
    <servlet-name>spring_mvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring_mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>spring_mvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- 在src目录下建立resource资源目录并创建spring_mvc.xml文件，该文件和spring的xml文件一样
- 同时需要load-on-startup 标签让DispatcherServlet在服务器启动时创建，这是由于servlet初始化会调用init方法，在DispatcherServlet这个类中的init方法会大致调用一下方法

```
WebApplicationContext ctx= new
ClassPathXmlApplicationContext("spring_mvc.xml");
```

  - 该方法和spring中的大致类似
- url-pattern的编写
  - 以斜杠作为url
  - 以***.do,do可以是任意自定义后缀，访问路径包含所有含有do为后缀的路径，推荐使用这种方式

- （2）编写请求页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<p><a href="some.action">我的第一个mvc</a></p>
</body>
</html>
```

- (3)编写控制器类和控制器方法

```
@Controller
public class first_mvc {

    @RequestMapping(value = "/some.action")
   public ModelAndView dosome()
    {
        //service方法....
```

```
        //向ModelAndView中注入数据
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg","a Object");
        mv.addObject("msg2","a Object");
        mv.setViewName("/action.jsp");
        //返回ModelAndView
        return mv;
    }

}
```

- @Controller:标识这个类是控制类，这种控制类也被称为后端控制器
- @RequestMapping("/some.action"):标识这个方法是控制器方法，
  - @RequestMapping用value属性作为该控制器方法的请求路径
  - @RequestMapping可以出现在类上也可以出现在方法上，在方法上居多
- ModelAndView类
  - ModelAndView是springmvc中内置的类，其中Model表示数据，View表示视图，说明这个方法返回处理结束后的数据和响应的页面
  - 向ModelAndView中注入值
    - `ModelAndView addObject(String attributeName, @Nullable Object attributeValue)`
      - 注入Model
    - `void setViewName(@Nullable String viewName)`
      - viewName指的是响应页面的绝对路径，从webapp开始
  - 在springmvc底层中会先将Model的数据放在Request，通过setAttribute()方法，把数据放在request作用域中，对于view会创建一个view对象并在响应时完成重定向（getRequestDispather().forward())

- （4）编写springmvc_config.xml文件

```
<context:component-scan base-package="springtest.mvc.Controller"/>
```

- （5）编写响应页面action.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<h3>msg:${msg}</h3><br/>
<h4>msg2:${msg2}</h4><br/>
</body>
</html>
```

- SpringMVC请求流程

    - （1）发起some.action给服务器
    - （2）Tomcat根据xml文件中的url-pattern给到中央调度器
    - （3）中央调度器读取spring_mvc.xml并读取到组件扫描器
    - （4）根据将请求转发给dosome方法
    - （5）dosome转发到 action.jsp
    - 底层原码
        - （1）Tomcat启动创建容器
            - 创建DispatcherServlet对象，DispatcherServlet调用intit方法

```
WebApplicationContext ctx= new
ClassPathXmlApplicationContext("spring_mvc.xml");
//创建容器
getServletContext().setAttribute(key,ctx);
//将容器对象放入到ServletContext中
```

        - （2）请求处理
            - DispatcherServlet在service方法中调用doservice方法，doservice调用doDispatch方法，而在doDispatch中会调用Controller类中的dosome方法
- SpringMVC补充内容
    - 对于某些页面我们不想让用户直接访问，可以将这些页面存放在WEB-INF目录下
        - 在服务器内部中可直接访问
    - 视图解析器
        - 对于ModelAndView对于页面的设置有时需要设置很多个，路径也比较长，可以采用视图解析器

```
<bean
class="org.springframework.web.servlet.view.InternalResourceViewReso
lver">
    <!--指定文件前缀-->
    <property name="prefix" value="/WEB-INF/view/"/>
    <!--指定文件后缀-->
    <property name="suffix" value=".jsp"/>
</bean>
```

            - 指定了视图解析器后，mv.setViewName("action")只需要获取文件名即可
- SpringMVC注解开发
    - ###@RequestMapping
        - 对于请求路径相同的部分可以放在Controller类上，用于表示模块，而以下的dosome方法则可以省略相同的路径

```
@Controller
@RequestMapping(value = "/test")
public class first_mvc {

    @RequestMapping(value = "/some.action")
   public ModelAndView dosome()
    {
   }
}
```

- 控制请求方式
  - method属性

```
@RequestMapping(value = "/some.action",method =
RequestMethod.GET)
public ModelAndView dosome()
 {
     //service方法....
     //向ModelAndView中注入数据
     ModelAndView mv = new ModelAndView();
     mv.addObject("msg","a Object");
     mv.addObject("msg2","a Object");
     mv.setViewName("action");
     //返回ModelAndView
     return mv;
 }
```

    - RequestMethod是个枚举类型，如果不添加method那么不会对请求方式进行限制
- 接受请求参数
  - 直接在@RequestMapping注解下的方法的参数中设置，springmvc会自动为其赋值

```
@RequestMapping(value = "/some.action",method =
RequestMethod.POST)
   public ModelAndView dosome(HttpServletRequest request,
HttpServletResponse response , HttpSession session)
    {

        Map<String, String[]> map = request.getParameterMap();
        //service方法....
        //向ModelAndView中注入数据
        ModelAndView mv = new ModelAndView();
        String[] names = map.get("name");
        String[] passwords = map.get("password");
        mv.addObject("name",names[0]);
        mv.addObject("password",passwords[0]);
        mv.setViewName("action");
        //返回ModelAndView
        return mv;
    }
```

    - 上例中采用了Request直接获取表单提交信息，但实际开发中不常用这种方式
- ####获取用户提交信息
    - 逐个接收

```
@RequestMapping(value = "/some.action",method =
RequestMethod.POST)
public ModelAndView dosome(String name,String password)
 {

     ModelAndView mv = new ModelAndView();
     mv.addObject("name",name);
     mv.addObject("password",password);
     mv.setViewName("action");
     //返回ModelAndView
     return mv;
 }
```

        - 在控制器方法中直接将用户传递的信息作为参数传入，要求是表单中的name必须和形参的名字相同，由于表单提交的类型都是字符串，所以mvc框架会为传参完成类型转换，但是对于int等类型作为形参时，如果表单提交为空，比如age提交时为空，而传参时为int则会报400错误，这时可以采用Integer等装箱类型
        - 如果表单中的name和形参的名字不相同，可以使用@RequestParam，使用操作和mybatis中的@Param相同

```
public ModelAndView dosome(@RequestParam("uname")String
name, String password, Integer age)
```

            - 在控制器方法参数行上指定
            - 除此之外@RequestParam还有另一个作用，@RequestParam含有一个成员boolean required，默认为true，代表表单提交时该参数必须有值否则400错误
        - 接受参数中如果包含中文POST请求会发生乱码问题，可以通过过滤器（CharacterEncodingFilter）解决
        - CharacterEncodingFilter
            - 变量
                - String encoding:编码方式
                - boolean forceRequestEncoding:强制Request使用指定编码，默认为false
                - boolean forceResponseEncding:强制Response使用指定编码，默认为false
            - 使用

```
<filter>
    <filter-name>encoding</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncod
ingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
```

```
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceRequestEncoding</param-
name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>forceResponseEncoding</param-
name>
        <param-value>true</param-value>
    </init-param>
</filter>

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

- **对象接收**
  - 对于参数较多的可以采用对象封装
    - User类

```
public class User {
    private String name;
    private String password;
    private int age;
    //省略set方法和无参构造，这两个必须有
}
```

      - 对象中的成员名称需要和提交的参数名称相同
    - 控制器方法

```
@RequestMapping(value = "/some.action",method =
RequestMethod.POST)
public ModelAndView dosomebyUser(User user)
{

    ModelAndView mv = new ModelAndView();
   mv.addObject("user",user);
    mv.setViewName("action");
    //返回ModelAndView
    return mv;
}
```

    - 重定向输出

```
<h3>name:${user.name}</h3><br/>
<h4>password:${user.password}</h4><br/>
<h5>age:${user.age}</h5><br/>
```

- 对于对象可以采用“.”来访问成员变量，这种访问方式要求User类中必须有get方法不然500错误

ModelAndView

- 控制器返回值
  - String
    - 如果只是需要重定向到指定页面可以采用直接返回String，如果spring_mvc.xml文件中含有视图解析器，可以直接放回页面文件名，否则需要返回绝对路径，以webapp为根。

```
@Controller
@RequestMapping(value = "/test")
public class first_mvc {
   /* @RequestMapping("/some.action")
    public String dosome()
    {
        return "/WEB-INF/view/action.jsp";
    }
    这种方式采用的就是没有视图解析器返回的方式
    */

    @RequestMapping("/some.action")
    public String dosome()
    {
        return "action";
    }

}
```

  - void
    - 既不返回数据也不返回页面，常用于ajax响应，响应的内容由response通过json格式直接输出
    - ajax常用json，需要添加jackson的依赖

```
<!--
https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core
jackson的依赖
-->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.13.1</version>
</dependency>
```

    - 使用步骤
      - （1）获取数据
      - （2）处理数据并转化为json字符串
      - （3）输出数据
  - 返回对象Object

- 对于返回void中的处理Ajax请求，第二步和第三步都是重复的，可以代码复用，而框架也提供了复用的功能
- 在spring_mvc.xml中添加

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

  - 注解驱动代替第二步，主要完成的是将Object对象转换为json格式、xml格式、text格式、二进制格式等
  - 该功能的主要实现类是HttpMessageConverter接口：消息转换器
  - 该接口主要4个方法，两两一对，canRead和read（请求发送有关），canWrite和write（响应有关），对于canWrite和write为步骤二的主要实现

```
 boolean canWrite(Class<?> clazz, @Nullable MediaType
mediaType);

void write(T t, @Nullable MediaType contentType,
HttpOutputMessage outputMessage) throws IOException,
HttpMessageNotWritableException;
```

    - canWrite会判断clazz是否可以转换为mediaType，MediaType指的是转换后的格式
    - Write将处理器方法返回值转换为指定格式
    - 该接口有许多实现类，比如StringHttpMessageConverter，MappingJackson2HttpMessageConverter（默认以json传输），而注解驱动在添加到xml文件中时会自动创建8个HttpMessageConverter接口的实现类对象，如果不添加注解驱动标签则mvc只会准备4个实现类其中包括StringHttpMessageConverter，添加后会增加MappingJackson2HttpMessageConverter等一系列实现类
- @ResponseBody注解完成第三步骤
  - @ResponseBody放在控制器方法上，通过HttpServletResponse输出数据
- 案例测试（通过返回Object实现ajax请求）
  - 后端处理类

```
@Controller
@RequestMapping(value = "/test")
public class first_mvc {
    @RequestMapping("/some.action")
    @ResponseBody
    public User dosome() throws JsonProcessingException
{
        User user = new User("zhangsan", "123", 21);
       return user;
    }
}
```

  - 前端页面

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
<html>
<head>
    <title>Title</title>
    <script type="text/javascript">
        function doajax()
        {
            var http=new XMLHttpRequest();
            http.onreadystatechange=function () {
                if(http.readyState==4 &&
http.status==200)
                {
                    var
response=JSON.parse(http.responseText);

document.getElementById('p1').innerText=response.name;

 document.getElementById('p2').innerText=response.age;

 document.getElementById('p3').innerText=response.passw
ord;
                }
            }
            http.open("get","test/some.action",true);
            http.send();
        }
    </script>
</head>
<body>

姓名<p id="p1">123</p>
年龄<p id="p2"></p>
密码<p id="p3"></p>
<input type="button" name="button" value="ajax"
onclick="doajax()"/>
</body>
</html>
```

    - 除此以外，控制类可返回对象数组，前端在接受是也是以数组接受
- 返回String类型
    - 如果只是返回字符串而不是地址，只需要在控制器方法上添加@ResponseBody即可
    - 前端在接受数据时不能以json解析
    - 如果返回字符串存在中文可能会存在乱码，RequestMapping中的属性produces可以指定编码和解析格式

```
@RequestMapping(value = "/test" ,produces
="text/plain;charset=utf-8")
```

Tomcat

- 底层
  - 对于某些资源比如图片，html，jsp等静态资源文件可以直接通过Tomcat管理访问，在Tomcat在conf下的web.xml文件中可以看到以下代码

```
<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>


!-- The mapping for the default servlet -->
    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- The mappings for the JSP servlet -->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>
```

  - Tomcat对于静态资源创建了servlet，并使用自己的servlet实现类，除此以外tomcat为其他未映射的路径提供了处理，第一个servlet-mapping即是对于这方面的处理
  - 而对于请求的处理优先交给项目中自己的web.xml文件，如果web.xml没有才会在Tomcat中查找，所以中央调度器的url最好不要设置为'/'，这样会覆盖Tomcat中的url，但是同样有办法在使用斜杠的同时享受Tomcat的处理，
  - 处理方式一：在SPRING_MVC.xml下添加

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

    - 使用这个标签后框架会自动创建一个调度器DefaultServletHttpRequestHandler将所有路径重定向转发给Tomcat，但是这也会导致我们自己的调度器收不到请求，这时可以添加注解驱动解决该问题
  - 处理方式二：使用mvc:resources标签

```
<mvc:resources mapping="访问静态资源的uri地址" location="静态资源在项目中的位置"/>
```

- 使用该标签后框架会创建ResourceHttpRequestHandler，让这个对象处理对于静态资源的访问，不依赖Tomcat
- mapping：如果webapp下存在images目录，且该目录中放置着所有图片，mapping="/images/**"，这代表images下的所有文件
- location:以mapping的例子，location可设置为/images/
- 这种方式同样和RequestMapping有冲突，需要添加注解驱动

- 关于访问路径的'/'问题
  - 相对地址:jsp,html中的地址都是前端中的地址，都是相对地址
  - 当你在触发事件时，访问地址将是当前页面的地址http://localhost:8080/untitled36/加上链接中的地址，也就是不加'/'
    - 例：

```
<a href="/test/some.action" >this is a href</a>
<br/>
<a href="test/some.action" >this is a href</a>
```

    - 最终的访问路径

```
http://localhost:8080/test/some.action
http://localhost:8080/untitled36_mv/test/some.action
```

    - 也就是说如果添加斜杠那么相对路径的根路径是http://localhost:8080，如果不添加则是http://localhost:8080/untitled36_mv，
    - 在添加斜杠的情况下还想要访问成功可以使用EL表达式

```
<a
href="${pageContext.request.contextPath}/test/some.action"
>this is a href</a>
```

  - 特殊情况

```
<a href="test/some.action" >this is a href</a>
```

    - 控制器方法

```
@Controller
@RequestMapping(value = "/test" ,produces ="text/plain;charset=utf-8")
public class first_mvc {
    @RequestMapping("/some.action")
    public ModelAndView dosome() throws JsonProcessingException {
       return new ModelAndView("/index.jsp");
    }
}
```

- 当点击链接后访问地址变为http://localhost:8080/untitled36_mv/test/some.action，此时再次点击地址变为http://localhost:8080/untitled36_mv/test/test/some.action，这样会导致访问错误
  - 解决方法一：使用EL表达式
  - 解决方法二：使用html中的base标签
    - base地址：添加在head标签中，存在base标签的页面中的所有没有'/'的连接都以base中的地址为参考地址

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
<html>
<head>
    <title>Title</title>
    <base href="http://localhost:8080/untitled36_mv/"
/>
</head>
<body>
<a href="test/some.action" >this is a href</a>
</body>
</html>
```

    - 但有时项目的地址会发生改变，此时可以动态编写base地址

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
<%
    String
basePath=request.getScheme()+"://"+request.getServerNam
e()+":"+request.getServerPort()+
    request.getContextPath()+"/";
%>
<html>
<head>
    <title>Title</title>
    <base href="<%=basePath%>" />
</head>
<body>
<a href="test/some.action" >this is a href</a>
</body>
</html>
```

- Springmvc中转发与重定向
  - 转发可以访问WEB-INF,重定向不能
  - 转发关键字：forward

```
public ModelAndView addUser(User user)
{
    int result =us.addUser(user);
    ModelAndView view = new ModelAndView();
    if(result>0)
```

```
    {
        view.setViewName("forward:/WEB-INF/view/addUser_safe.jsp");
    }
    else
    {
        view.setViewName("forward:/WEB-
INF/view/addUser_defeat.jsp");
    }
    return view;
}
```

- 重定向redirect关键字和forward使用相同但是不能访问WEB-INF下的内容
- 但是对于重定向而言，当用户发起一次请求,时，假设request为r1，而发生重定向时，servlet会再次发送请求，假设这个请求为r2，r2中不包含r1的提交数据，但是springmvc框架会将ModelAndView中的Model值通过get请求一起放入r2中，如果想要获得该数据可以使用jsp中的param关键字

```
${param.name}
```

异常处理

- 在项目中常常会有大量的异常处理的代码，框架可以使用AOP的思想将try/catch作为一个模板，集中处理异常
- 操作步骤
  - 创建自定义异常类（最好是先定义一个父类MyException，后定义各个不同的子类，方便在方法中抛出）
    - 在自定义异常类中重写带参（String）和无参构造方法
  - 创建一个普通类作为集中处理异常的类

```
@ControllerAdvice
public class MyExceptionHandler {

    @ExceptionHandler(NameException.class)
    public ModelAndView NameErrorHandler(NameException n)
    {
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception",n);
        mv.setViewName("addUser_defeat");
        return mv;
    }
    @ExceptionHandler(PwdException.class)
    public ModelAndView PwdErrorHandler(PwdException p)
    {
        ModelAndView mv = new ModelAndView();
        mv.addObject("exception",p);
        mv.setViewName("addUser_defeat");
        return mv;
    }
    @ExceptionHandler
    public ModelAndView OtherErrorHandler(Exception exce)
    {
        ModelAndView mv = new ModelAndView();
```

```
            mv.addObject("exception",exce);
            mv.setViewName("addUser_defeat");
            return mv;
        }

    }
```

- 集中处理异常的类的方法
  - 同样可以返回ModelAndVeiw，String，void等，参数中需要有一个异常类
  - 在该方法中需要将异常的时间，内容记录到日志文件中，为用户重定向页面
  - 假如定义了三个自定义异常，除了这三个异常之外，其他的异常统一用一个方法处理，该方法上的@ExceptionHandler注解的value不赋值。
- 两个注解
  - @ExceptionHandler：添加在集中处理异常的类的方法上，其中有一个类型为Class[]的value，默认为空，当发生value中的异常时会自动执行该注解下的方法
  - @ControllerAdvice：添加在集中处理异常的类上，控制器增强注解
- 创建异常的处理页面
- 编写springmvc配置文件
  - 在springMVC配置文件中添加组件扫描器：用于扫描@ControllerAdvice所在的包，用于扫描@Controller所在的包
    - 同样使用context:component-scan，添加这两个注解所在的包即可
  - 声明注解驱动

拦截器

- 属于springmvc中的一种，需要实现HandlerInterceptor接口
- 拦截器用于拦截用户请求，过滤器用来过滤请求参数
- 拦截器可以对多个Controller做拦截，常用于用户登录处理，权限检查，记录日志
- 使用步骤
  - 定义类实现HandlerInterceptor接口
    - 实现接口中的三个方法（可以选择性的实现）
      - preHandler前处理方法

```
default boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler)
```

        - Object handler表示被拦截的控制器对象
        - boolean作为返回值
        - 特点
          - 在控制器方法执行前执行，用于验证用户是否登录，验证请求是否符合要求，验证是否权限是否符合
      - postHandler后处理方法

```
default void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, @Nullable
ModelAndView modelAndView)
```

    - 在处理器方法之后执行
    - Object handler表示被拦截的控制器对象
    - ModelAndView modelAndView表示控制器对象的返回值，可以通过修改modelAndView影响之后的执行
- afterCompletion最后执行的方法

```
default void afterCompletion(HttpServletRequest
request, HttpServletResponse response, Object handler,
@Nullable Exception ex) throws Exception
```

    - Exception ex：程序中的异常
    - 在请求处理完成后执行，也就是在视图跳转后，一般做资源回收工作
- 四个方法执行顺序

```
preHandler（返回真，代表通过拦截器前处理方法）->controller类方法->postHandler->afterCompletion

preHandler（返回假，代表不通过拦截器前处理方法）->空白页面（可以在preHandler方法中重定向）
```

- 修改springmvc配置文件
    - 声明拦截器，并指定拦截的uri地址

```
<!--声明拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <!--表示需要拦截的uri地址，可以使用通配符-->
        <mvc:mapping path="/**"/>
         <!--拦截器的类-->
        <bean class="com.SSM01.Interceptor.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

- 多个拦截器的存储方式是ArrayList，执行时按声明的先后顺序执行，比如Interceptor1先声明，Interceptor2后声明，以下为各方法执行顺序

```
Interceptor1.preHandler
Interceptor2.preHandler
Controller方法
Interceptor2.postHandler
Interceptor1.postHandler
Interceptor2.afterCompletion
Interceptor1.afterCompletion
```

SpringMVC执行流程

- 底层原理有关类
  - 处理器映射器（实现了HandlerMapping接口的类）
    - 在MVC容器中含有多个处理器映射器
    - 处理器映射器作用是从容器中获取对应请求的控制器对象和拦截器，并将其放置在处理器执行链（HandlerExecutionChain）中保存，最后将处理器执行链返回至中央调度器

      ```
      HandlerExecutionChain
      mappedHandler=getHandler(processedRequest);
      ```

    - HandlerExecutionChain的成员
      - 控制器
      - 拦截器的list集合
  - 处理器适配器（实现了HandlerAdaptor接口的类）
    - 调用对应的控制器的对应的方法并得到结果并将结果返回给中央调度器

      ```
      HandlerAdapter ha=getHandlerAdapter(mappedHander.getHandler());
      mv=ha.handle(processedRequest,response,mappedHandler.getHandler
      ())
      ```

  - 视图解析器（InternalResourceViewResolver，实现了ViewResolver接口的类都称为视图解析器）
    - 解析中央调度器中传递的ModelAndView
    - 作用：通过前缀和后缀组成视图的完整路径并创建View对象并返回给中央调度器
      - View接口
        - 用于表示视图，比如jsp，html等页面
        - 中央调度器先将Model中的值放入Request中并执行View的forward方法，请求就此结束