第七讲 SpringMVC 高级特性

一、响应数据和结果视图

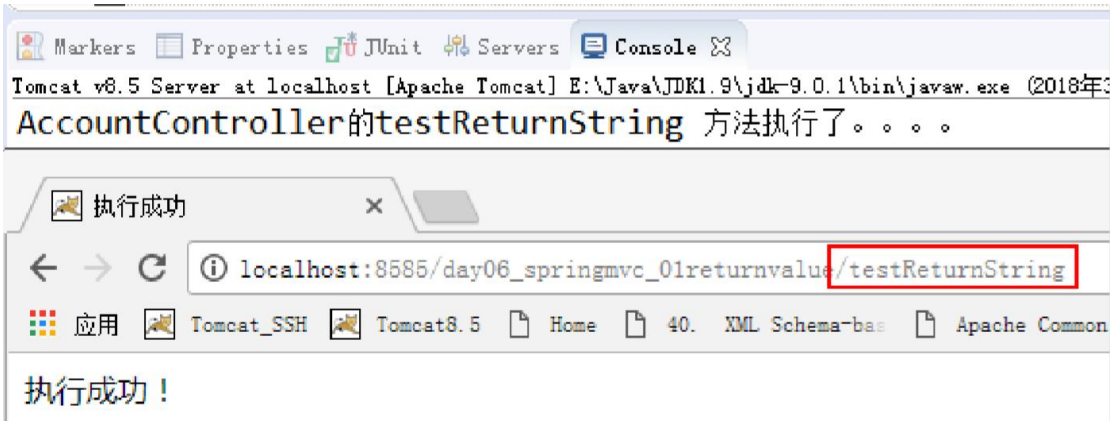
1. 返回值分类

① 字符串

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。

```
//指定逻辑视图名，经过视图解析器解析为 jsp 物理路径：/WEB-INF/pages/success.jsp
@RequestMapping("/testReturnString")
 public String testReturnString() {
  System.out.println("AccountController 的 testReturnString 方法执行了。。。。");
  return "success";
 }
```

运行结果：



② void

在 controller 方法形参上可以定义 request 和 response，使用 request 或 response 指定响应结果：

```
@RequestMapping("/testReturnVoid")
```

```
public void testReturnVoid(HttpServletRequest request,HttpServletResponse response) throws Exception {
  }
```

- 使用 request 转向页面，如下：

```
request.getRequestDispatcher("/WEB-INF/pages/success.jsp").forward(request, response);
```

- 通过 response 页面重定向：

```
response.sendRedirect("testRetrunString")
```

- 通过 response 指定响应结果，例如响应 json 数据：

```
response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
response.getWriter().write("json 串");
```

③ ModelAndView

ModelAndView 是 SpringMVC 为我们提供的一个对象,该对象也可以用作控制器方法的返回值。

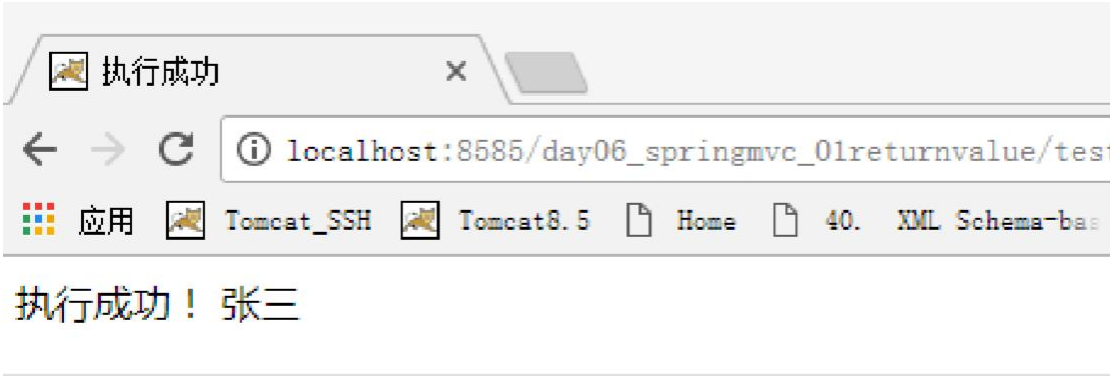
- 示例代码：

```
/**
 * 返回 ModeAndView  * @return  */
@RequestMapping("/testReturnModelAndView") public ModelAndView testReturnModelAndView() { ModelAndView mv = new ModelAndView(); mv.addObject("username", "张三"); mv.setViewName("success");
 return mv; }
```

- 响应的 jsp 代码：

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html>
<head> <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>执行成功</title> </head> <body>
执行成功！

${requestScope.username} </body> </html>
```

- 输出结果：



注意： 我们在页面上上获取使用的是 requestScope.username 取的，所以返回 ModelAndView 类型时，浏 览器跳转只能是请求转发。

2. 转发和重定向

① forward 转发

controller 方法在提供了 String 类型的返回值之后，默认就是请求转发。我们也可以写成：

```
/**
  * 转发
  * @return
  */
 @RequestMapping("/testForward")
  public String testForward() {
    System.out.println("AccountController 的 testForward 方法执行了。。。。");
  return "forward:/WEB-INF/pages/success.jsp";
  }
```

需要注意的是，如果用了 formward：则路径必须写成实际视图 url，不能写逻辑视图。它相当于“request.getRequestDispatcher("url").forward(request,response)”。使用请求 转发，既可以转发到 jsp，也可以转发到其他的控制器方法。

② Redirect 重定向

contrller 方法提供了一个 String 类型返回值之后，它需要在返回值里使用:redirect:

```
/**
```

```
 * 重定向
 * @return
 */
 @RequestMapping("/testRedirect")
 public String testRedirect() {
   System.out.println("AccountController 的 testRedirect 方法执行了。。。。");
   return "redirect:testReturnModelAndView";
}
```

它相当于“response.sendRedirect(url)”。需要注意的是，如果是重定向到 jsp 页面，则 jsp 页面不 能写在 WEB-INF 目录中，否则无法找到。

3. ResponseBody 响应 json 数据

① 使用说明

作用：

该注解用于将 Controller 的方法返回的对象，通过 HttpMessageConverter 接口转换为指定格式的 数据如：json,xml 等，通过 Response 响应给客户端

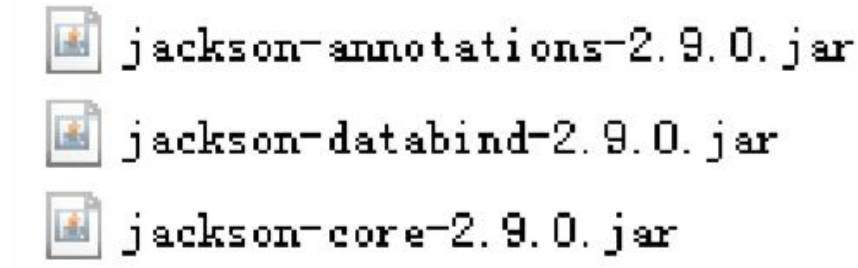
② 使用示例

- 需求：

使用@ResponseBody 注解实现将 controller 方法返回对象转换为 json 响应给客户端。

- 前置知识点：

Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要加入 jackson 的包。



- jsp 中的代码：

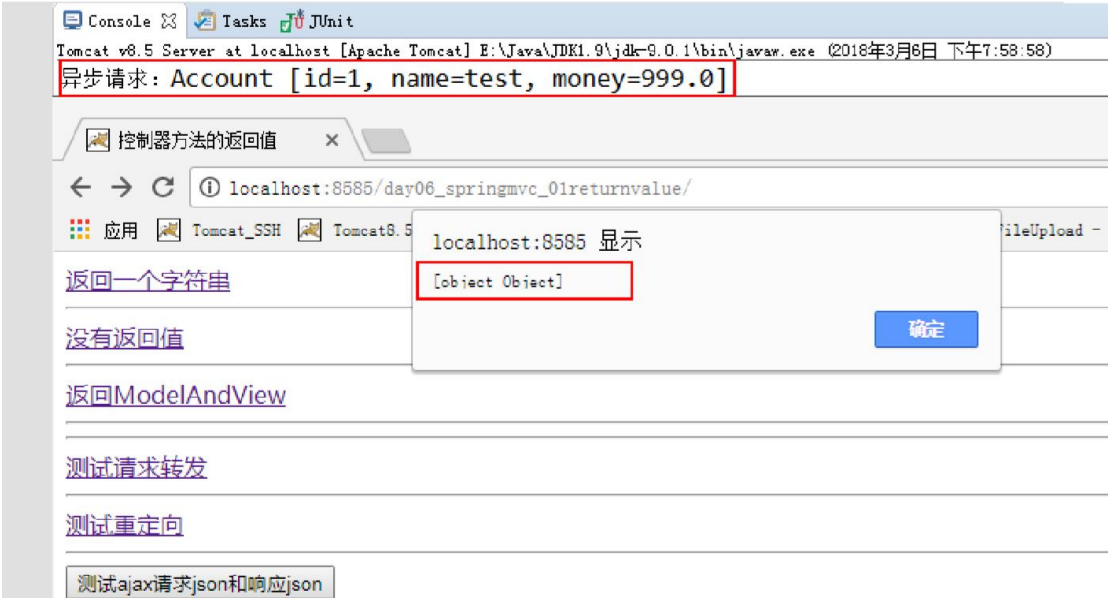
```
<script type="text/javascript" src="${pageContext.request.contextPath}/js/jquery.min.js"></script>
<script type="text/javascript">
  $(function(){
    $("#testJson").click(function(){
      $.ajax({
        type:"post",
      url:"${pageContext.request.contextPath}/testResponseJson",
        contentType:"application/json;charset=utf-8",
      data:'{"id":1,"name":"test","money":999.0}',
        dataType:"json",
        success:function(data){
          alert(data);
      }
      });
    });
  })
 </script>
<!-- 测试异步请求 -->
 <input type="button" value=" 测试 ajax 请求 json 和响应 json" id="testJson"/>
```

- 控制器中的代码：

```
/**
 * 响应 json 数据的控制器
 * @Version 1.0
 */
@Controller("jsonController")
 public class JsonController {
     /**
  * 测试响应 json 数据
  */
  @RequestMapping("/testResponseJson")
  public @ResponseBody Account testResponseJson(@RequestBody Account account) {
    System.out.println("异步请求："+account);
```

```
    return account;
  }
}
```

- 运行结果：



二、 SpringMVC 实现文件上传

1. 必要前提

- form 表单的 enctype 取值必须是：multipart/form-data (默认值是:application/x-www-form-urlencoded) enctype:是表单请求正文的类型
- method 属性取值必须是 Post
- 提供一个文件选择域<input type=”file” />

2. 借助第三方组件实现文件上传

使用 Commons-fileupload 组件实现文件上传，需要导入该组件相应的支撑 jar 包：Commons-fileupload 和 commons-io。commons-io 不属于文件上传组件的开发 jar 文件，但 Commons-fileupload 组件从 1.1 版本开始，它工作时需要 commons-io 包的支持。


commons-fileupload-1.3.1.jar


commons-io-2.4.jar

3. 实现步骤

- 编写 jsp 页面

```
<form action="/fileUpload" method="post" enctype="multipart/form-data">
 名称：<input type="text" name="picname"/><br/>
  图片：<input type="file" name="uploadFile"/><br/>
  <input type="submit" value=" 上传 "/>
 </form>
```

- 编写控制器

```
/**
 * 文件上传的的控制器
 * @Version 1.0
  */
@Controller("fileUploadController")
 public class FileUploadController {
     /**
  * 文件上传
  */
 @RequestMapping("/fileUpload")
  public String testResponseJson(String picname,MultipartFile
uploadFile,HttpServletRequest request) throws Exception{
    //定义文件名
    String fileName = "";
  //1.获取原始文件名
    String uploadFileName =  uploadFile.getOriginalFilename();
  //2.截取文件扩展名
    String extendName =
uploadFileName.substring(uploadFileName.lastIndexOf(".")+1,
uploadFileName.length());
```

```
  //3.把文件加上随机数，防止文件重复
    String uuid = UUID.randomUUID().toString().replace("-", "").toUpperCase();
  //4.判断是否输入了文件名
    if(!StringUtils.isEmpty(picname)) {
      fileName = uuid+"_"+picname+"."+extendName;
    }else {
      fileName = uuid+"_"+uploadFileName;
  }
    System.out.println(fileName);
  //2.获取文件路径
  ServletContext context = request.getServletContext();
  String basePath = context.getRealPath("/uploads");
  //3.解决同一文件夹中文件过多问题
    String datePath = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
//4.判断路径是否存在
    File file = new File(basePath+"/"+datePath);
    if(!file.exists()) {
      file.mkdirs();
  }
  //5.使用 MulitpartFile 接口中方法，把上传的文件写到指定位置
    uploadFile.transferTo(new File(file,fileName));
    return "success";
  }
 }
```

- 配置文件解析器

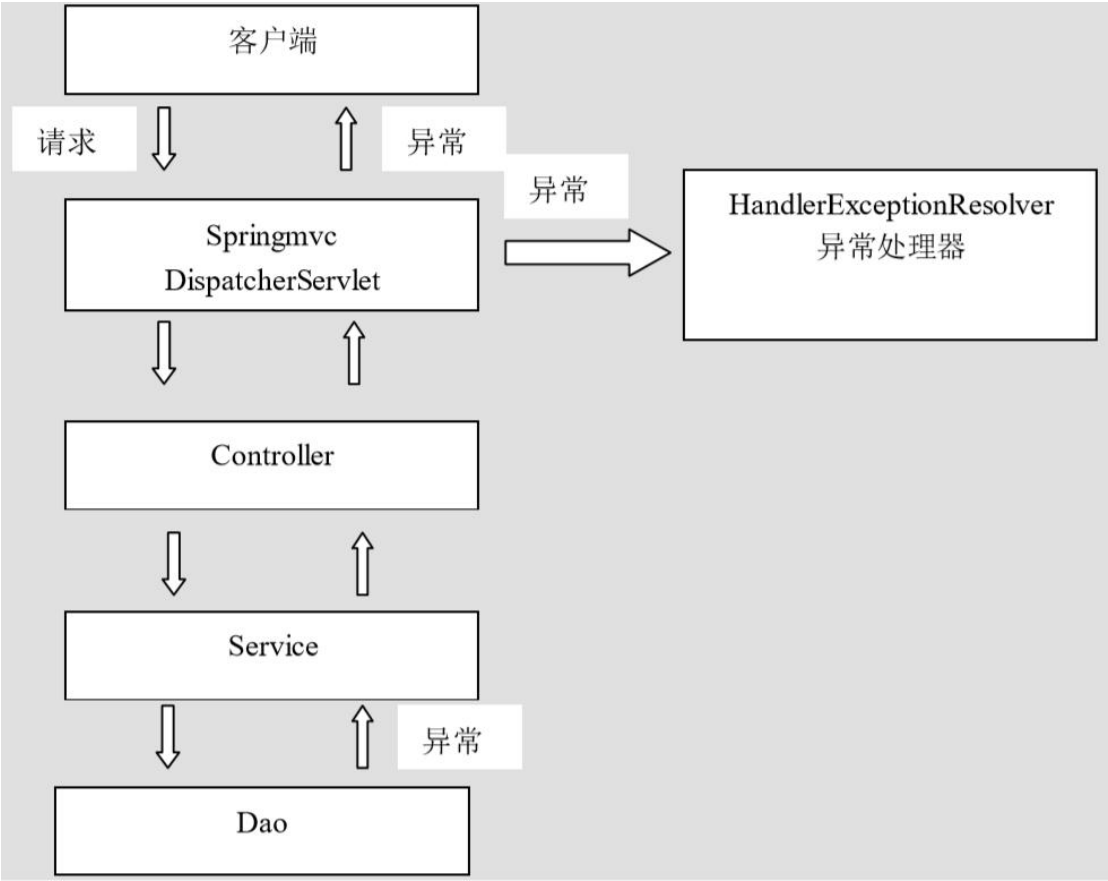
```
<!-- 配置文件上传解析器 -->
<bean id="multipartResolver"   <!-- id 的值是固定的-->
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
 <!-- 设置上传文件的最大尺寸为 5MB -->
  <property name="maxUploadSize">
   <value>5242880</value>
  </property>
</bean>
```

注意： 文件上传的解析器 id 是固定的，不能起别的名称，否则无法实现请求参数的绑定。（不光是文件，其他 字段也将无法绑定）

三、SpringMVC 中的异常处理

1. 异常处理的思路

系统中异常包括两类：预期异常和运行时异常 RuntimeException，前者通过捕获异常从而获取异常信息， 后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。 系统的 dao、service、controller 出现都通过 throws Exception 向上抛出，最后由 springmvc 前端 控制器交由异常处理器进行异常处理，如下图：



2. 实现步骤

- 编写异常类

```
/**
 * 自定义异常
 * @Version 1.0
 */
```

```
public class CustomException extends Exception {

 private String message;

 public CustomException(String message) {
    this.message = message;
  }

 public String getMessage() {
    return message;
  }
 }
```

- jsp 错误页面：

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html> <head> <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>执行失败</title>
 </head>
 <body>
执行失败!

${message }
</body>
</html>
```

- 自定义异常处理器

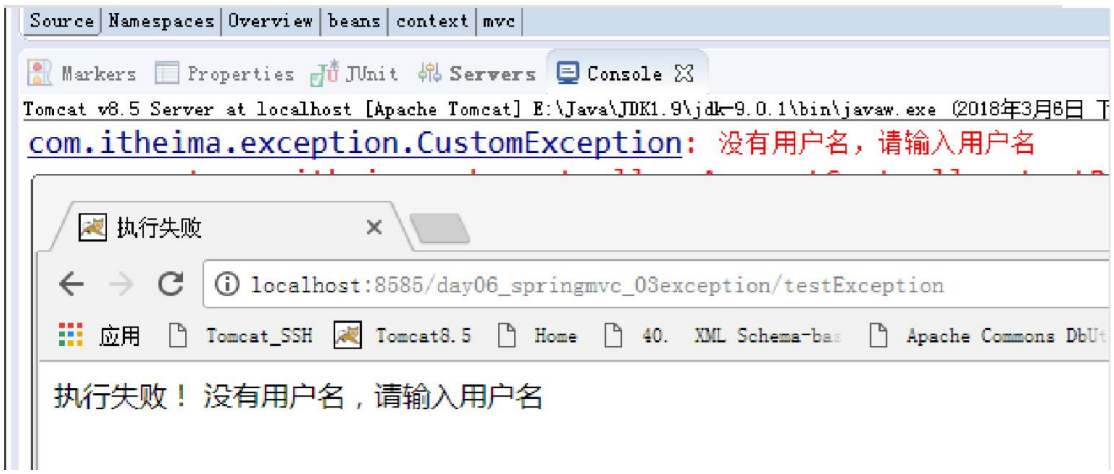
```
/**
 * 自定义异常处理器
 * @Version 1.0
  */
 public class CustomExceptionResolver implements HandlerExceptionResolver {

 @Override
 public ModelAndView resolveException(HttpServletRequest request,
```

```
HttpServletResponse response, Object handler, Exception ex) {


  ex.printStackTrace();
    CustomException customException = null;
  //如果抛出的是系统自定义异常则直接转换
    if(ex instanceof CustomException){
      customException = (CustomException)ex;
    }else{
      //如果抛出的不是系统自定义异常则重新构造一个系统错误异常。
      customException = new CustomException("系统错误，请与系统管理 员联系！");
    }
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("message", customException.getMessage());
  modelAndView.setViewName("error");
    return modelAndView;
  }
 }
```

- 配置异常处理器

```
<!-- 配置自定义异常处理器 -->
<bean id="handlerExceptionResolver"
  class="com.itheima.exception.CustomExceptionResolver"/>
```

- 运行结果：



四、 SpringMVC 中的拦截器

1. 拦截器的作用

Spring 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。

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，还要向大家提一个词——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。

2. 自定义拦截器的步骤

- 第一步：编写一个普通类实现 HandlerInterceptor 接口

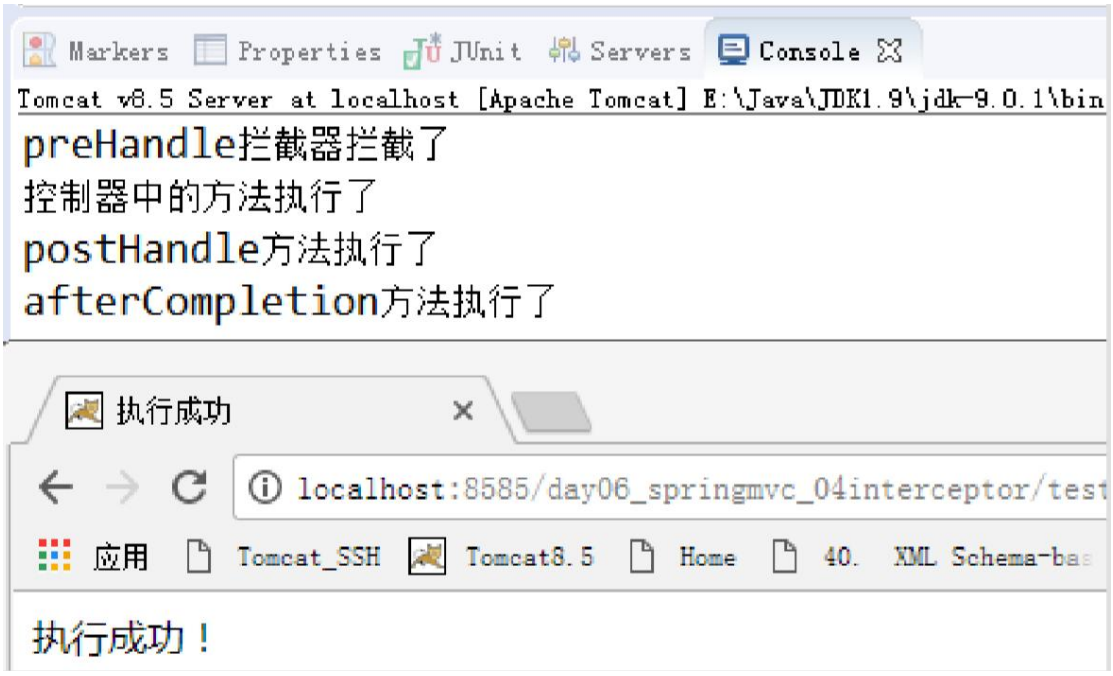
```
/**
 * 自定义拦截器
 * @Version 1.0  */
 public class HandlerInterceptorDemo1 implements HandlerInterceptor {

 @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)      throws Exception {
    System.out.println("preHandle 拦截器拦截了");
 return true;
 }
```

```
 @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,      ModelAndView modelAndView) throws Exception {
    System.out.println("postHandle 方法执行了");
 }

 @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)      throws Exception {
    System.out.println("afterCompletion 方法执行了");  }
}
```

- 第二步：配置拦截器

```
<!-- 配置拦截器 -->
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**"/>
  <bean                                                    id="handlerInterceptorDemo1"
class="com.itheima.web.interceptor.HandlerInterceptorDemo1"></bean>
  </mvc:interceptor>
</mvc:interceptors>
```

- 测试运行结果：



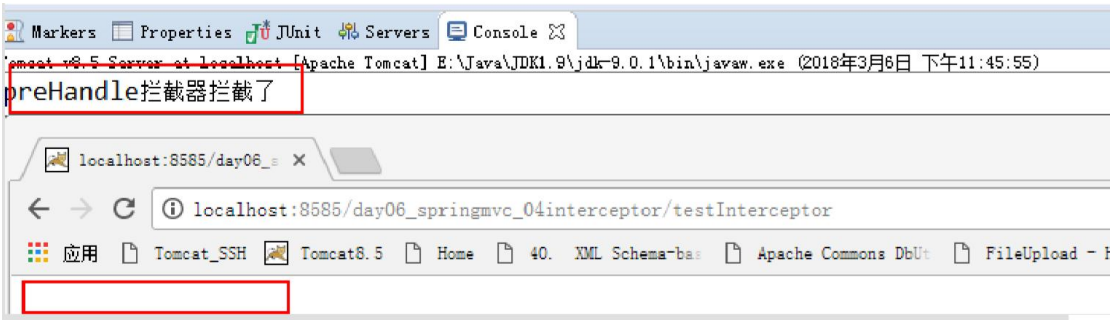
3. 拦截器的细节

① 拦截器的放行

放行的含义是指，如果有下一个拦截器就执行下一个，如果该拦截器处于拦截器链的最后一个，则执行控制器中的方法

```
16
17      @Override
18      public boolean preHandle(HttpServletRequest request, HttpServletR
19              throws Exception {
20          System.out.println("preHandle拦截器拦截了");
21          return false;
22      }
```

只有当此方法返回true的时候，程序才能继续执行



② 拦截器中方法的说明

```
public interface HandlerInterceptor {
   /**
   * 如何调用：
   *    按拦截器定义顺序调用
   * 何时调用：
   *   只要配置了都会调用
   * 有什么用：
   *   如果程序员决定该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去
   *    进行处理，则返回 true。
   *   如果程序员决定不需要再调用其他的组件去处理请求，则返回 false。
   */
  default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
```

```
handler)      throws Exception {
    return true;
  }
 /**
  * 如何调用：
  *    按拦截器定义逆序调用
  * 何时调用：
  *   在拦截器链内所有拦截器返成功调用
  * 有什么用：
  *   在业务处理器处理完请求后，但是 DispatcherServlet 向客户端返回响应前被调用，
   *   在该方法中对用户请求 request 进行处理。
     */
  default void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler,      @Nullable ModelAndView modelAndView) throws Exception
{  }

 /**
  * 如何调用：
  *    按拦截器定义逆序调用
  * 何时调用：
  *   只有 preHandle 返回 true 才调用
 * 有什么用：
  *   在 DispatcherServlet 完全处理完请求后被调用，
  *    可以在该方法中进行一些资源清理的操作。
  */
  default void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler,      @Nullable Exception ex) throws Exception {  }
 }
```

思考： 如果有多个拦截器，这时拦截器 1 的 preHandle 方法返回 true，但是拦截器 2 的 preHandle 方法返 回 false，而此时拦截器 1 的 afterCompletion 方法是否执行?

③ 拦截器的作用路径

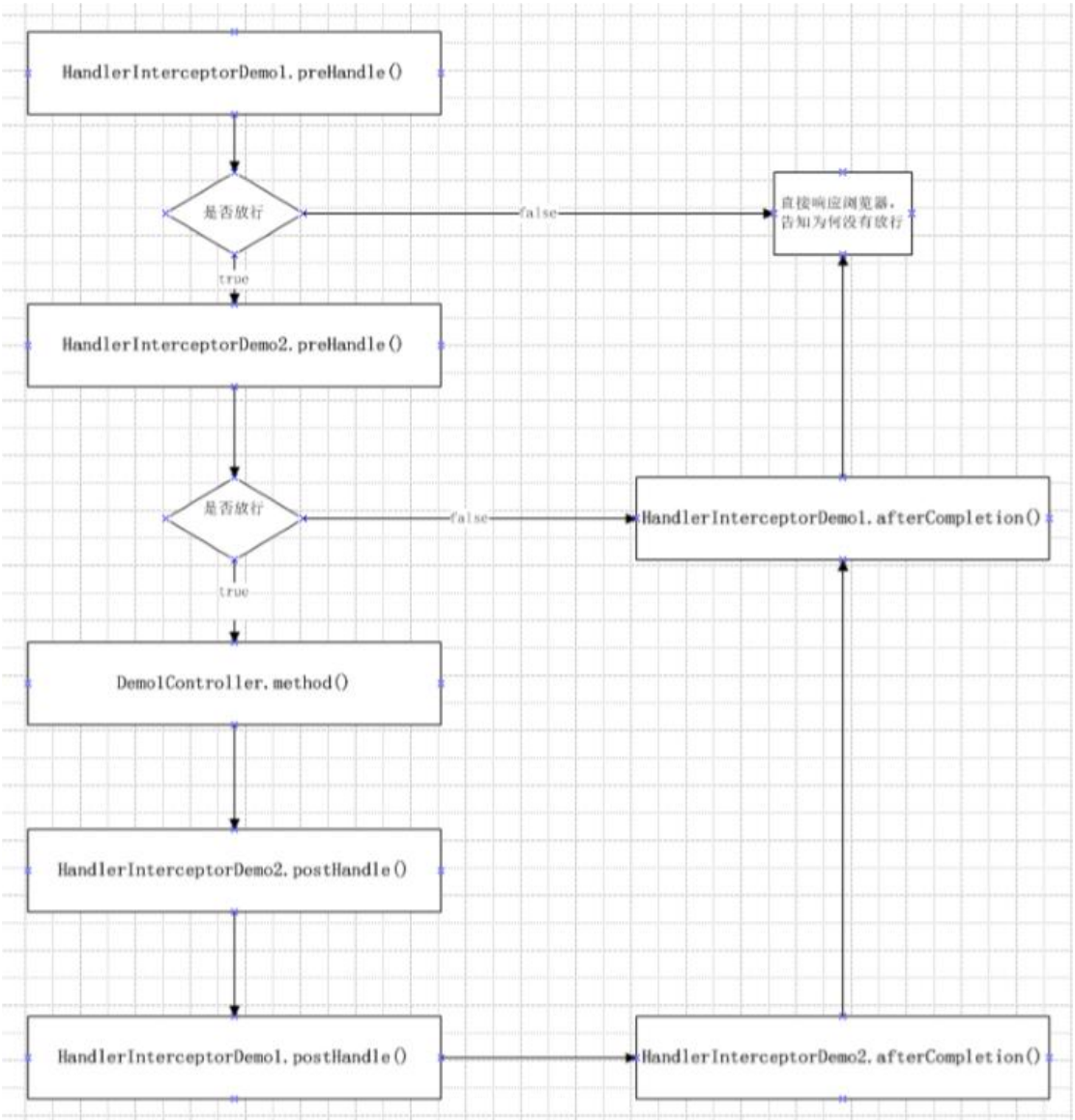
作用路径可以通过在配置文件中配置。

```
<!-- 配置拦截器的作用范围 -->
```

```
<mvc:interceptors>
  <mvc:interceptor>
  <mvc:mapping path="/**" /><!-- 用于指定对拦截的 url -->
    <mvc:exclude-mapping path=""/><!-- 用于指定排除的 url-->
    <bean id="handlerInterceptorDemo1"
    class="com.itheima.web.interceptor.HandlerInterceptorDemo1"></bean>
</mvc:interceptor>
</mvc:interceptors>
```

④ 多个拦截器的执行顺序

多个拦截器是按照配置的顺序决定的。



4. 正常流程测试

- 配置文件：

```
<!-- 配置拦截器的作用范围 -->
  <mvc:interceptors>
   <mvc:interceptor>
   <mvc:mapping path="/**" />
<!-- 用于指定对拦截的 url -->
    <bean id="handlerInterceptorDemo1"
class="com.itheima.web.interceptor.HandlerInterceptorDemo1"></bean>
  </mvc:interceptor>
   <mvc:interceptor>
   <mvc:mapping path="/**" />
    <bean id="handlerInterceptorDemo2"
class="com.itheima.web.interceptor.HandlerInterceptorDemo2"></bean>
  </mvc:interceptor>
  </mvc:interceptors>
```

- 拦截器 1 的代码：

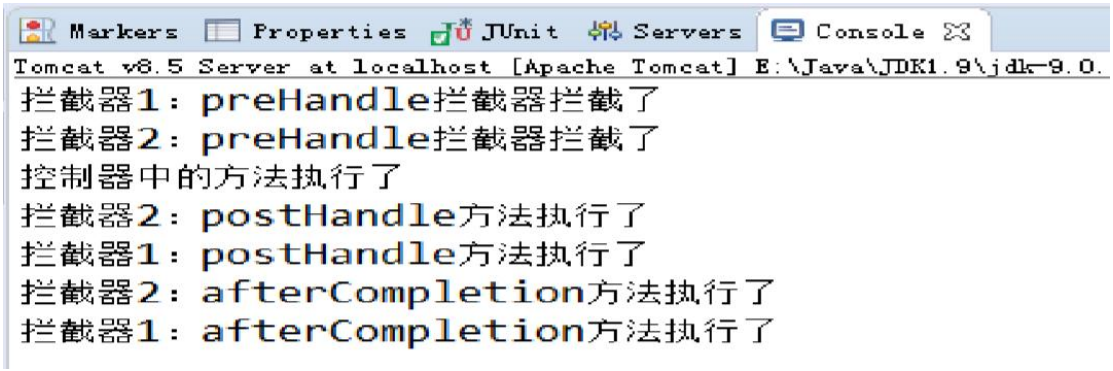
```
/**
 * 自定义拦截器
 * @Version 1.0
 */
public class HandlerInterceptorDemo1 implements HandlerInterceptor {

 @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)      throws Exception {
   System.out.println("拦截器 1：preHandle 拦截器拦截了");
   return true;
 }

 @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler,      ModelAndView modelAndView) throws Exception {
   System.out.println("拦截器 1：postHandle 方法执行了");
```

```
  }

 @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)      throws Exception {
    System.out.println("拦截器 1：afterCompletion 方法执行了");
  }
 }
```

- 拦截器 2 的代码：

```
/**
 * 自定义拦截器
 * @Version 1.0
 */
public class HandlerInterceptorDemo2 implements HandlerInterceptor {

 @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)      throws Exception {
    System.out.println("拦截器 2：preHandle 拦截器拦截了");
    return true;
 }

 @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,      ModelAndView modelAndView) throws Exception {
    System.out.println("拦截器 2：postHandle 方法执行了");
 }

 @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)      throws Exception {
    System.out.println("拦截器 2：afterCompletion 方法执行了");
  }
 }
```

- 运行结果：



5. 中断流程测试

- 配置文件：

```
<!-- 配置拦截器的作用范围 -->
<mvc:interceptors>
  <mvc:interceptor>
  <mvc:mapping path="/**" />
<!-- 用于指定对拦截的 url -->
    <bean id="handlerInterceptorDemo1"
class="com.itheima.web.interceptor.HandlerInterceptorDemo1"></bean>
</mvc:interceptor>
 <mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean id="handlerInterceptorDemo2"
class="com.itheima.web.interceptor.HandlerInterceptorDemo2"></bean>
</mvc:interceptor>
 </mvc:interceptors>
```

- 拦截器 1 的代码：

```
/**
 * 自定义拦截器
 */
 public class HandlerInterceptorDemo1 implements HandlerInterceptor {
 @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)      throws Exception {
    System.out.println("拦截器 1：preHandle 拦截器拦截了");
```

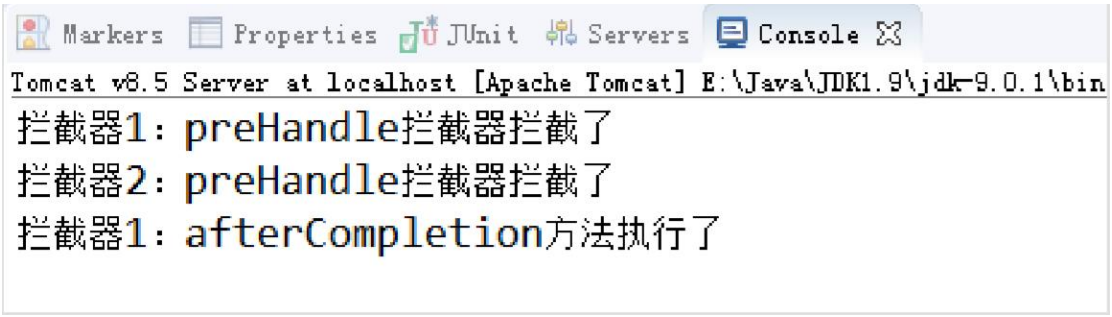
```
    return true;
 }

 @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,      ModelAndView modelAndView) throws Exception {
    System.out.println("拦截器 1：postHandle 方法执行了");
  }

 @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)      throws Exception {
    System.out.println("拦截器 1：afterCompletion 方法执行了");
  }
 }
```

- 拦截器 2 的代码：

```
/**
 * 自定义拦截器
 */
public class HandlerInterceptorDemo2 implements HandlerInterceptor {

 @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)      throws Exception {
    System.out.println("拦截器 2：preHandle 拦截器拦截了");
    return false;
 }

 @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,      ModelAndView modelAndView) throws Exception {
    System.out.println("拦截器 2：postHandle 方法执行了");
 }

 @Override
```

```
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)       throws Exception {
        System.out.println("拦截器 2：afterCompletion 方法执行了");
    }
}
```

- 运行结果：



6. 拦截器的简单案例（验证用户是否登录）

- 实现思路

  有一个登录页面，需要写一个 controller 访问页面

  登录页面有一提交表单的动作。需要在 controller 中处理。

  判断用户名密码是否正确

  如果正确 向 session 中写入用户信息

  返回登录成功。

  拦截用户请求，判断用户是否登录

  如果用户已经登录。放行

  如果用户未登录，跳转到登录页面

- 控制器代码

```
//登陆页面
 @RequestMapping("/login")
 public String login(Model model)throws Exception{
    return "login";
}
//登陆提交
 //userid：用户账号，pwd：密码
@RequestMapping("/loginsubmit")
 public String loginsubmit(HttpSession session,String userid,String pwd)throws
```

```
Exception{

 //向 session 记录用户身份信息
   session.setAttribute("activeUser", userid);
   return "redirect:/main.jsp";
}

//退出
 @RequestMapping("/logout")
 public String logout(HttpSession session)throws Exception{
 //session 过期
   session.invalidate();
       return "redirect:index.jsp";
 }
```

拦截器代码

```
public class LoginInterceptor implements HandlerInterceptor{

 @Override
   Public boolean preHandle(HttpServletRequest request,      HttpServletResponse response, Object handler) throws Exception {

   //如果是登录页面则放行
    if(request.getRequestURI().indexOf("login.action")>=0){
     return true;
    }
   HttpSession session = request.getSession();
   //如果用户已登录也放行
    if(session.getAttribute("user")!=null){
     return true;
    }
   //用户没有登录挑战到登录页面
    request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);
    return false;
   }
```

```
}
```