本章学习目标

- SpringMVC 异常处理
- SpringMVC 文件上传
- SpringMVC 处理 JSON 格式数据
- SpringMVC 拦截器
- SpringMVC 对 restful 风格的支持

1. SpringMVC 异常处理

1.1. @ExceptionHandler 注解处理异常

@ExceptionHandler 该注解使用在异常处理方法上面

1.1.1. 直接在 Controller 类里面使用

```
@Controller
@RequestMapping("/exception")
public class ExceptionController {

    /**
     * 异常处理方法
     * value: 该异常方法可以处理的异常类型
     * @return
     */
    @ExceptionHandler(value={java.lang.NullPointerException.class})
    public ModelAndView exceptionHandler(Exception ex){
        ModelAndView mv = new ModelAndView();

        StringWriter writer = new StringWriter();
```

```
        PrintWriter s = new PrintWriter(writer);
        ex.printStackTrace(s);

        mv.addObject("exception", writer.toString());
        mv.setViewName("error");
        return mv;
    }

    @RequestMapping("/test1")
    public String test1(){
        System.out.println("ExceptionController 的 test1");
        //模拟异常
        String name = null;
        //java.lang.NullPointerException
        name.substring(0);
        return "success";
    }
}
```

1.1.2. 单独编写独立类使用

```
@ControllerAdvice
public class MyExcptionHandler {


    /**
     * 异常处理方法
     * value: 该异常方法可以处理的异常类型
     * @return
     */
    @ExceptionHandler(value={java.lang.NullPointerException.class})
```

```
    public ModelAndView exceptionHandler(Exception ex){
        ModelAndView mv = new ModelAndView();

        StringWriter writer = new StringWriter();
        PrintWriter s = new PrintWriter(writer);
        ex.printStackTrace(s);

        mv.addObject("exception", writer.toString());
        mv.setViewName("error");
        return mv;
    }
}
```

1.2. SimpleMappingExceptionResolver 处理异常

好处：可以通过配置文件的方式处理不同的异常

1.2.1. 配置 spring-mvc.xml

```
<!-- SimpleMappingExceptionResolver 配置 -->
    <bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionRe
solver">
        <!-- 支持异常列表 -->
        <property name="exceptionMappings">
            <props>
                <prop key="java.lang.ArithmeticException">error</prop>
            </props>
        </property>
```

```
        <!-- 默认就会把异常信息封装到 exception 名称属性 -->
    </bean>
```

1.3. HandlerExceptionResolver 处理异常

1.3.1. 需要编写一个自定义异常类

```
/**
 * 自定义异常类
 * @author lenovo
 *
 */
public class MyHandlerExeptionResolver implements
HandlerExceptionResolver{

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
            HttpServletResponse response, Object arg2, Exception ex) {

        //取出异常栈信息
        StringWriter writer = new StringWriter();
        PrintWriter s = new PrintWriter(writer);
        ex.printStackTrace(s);

        ModelAndView mv = new ModelAndView();
        mv.addObject("ex", writer.toString());
        mv.setViewName("error");
        return mv;
    }
```

```
}
```

1.3.2. 配置 spring-mvc.xml

```
<bean class="cn.sm1234.exception.MyHandlerExeptionResolver"/>
```

2. SpringMVC 文件上传

2.1. 导入文件上传的支持 jar 包

2.2. 页面

```
<form action="${pageContext.request.contextPath }/upload/test1.action"
method="post" enctype="multipart/form-data">
    用户名：<input type="text" name="userName"/><br/>
    密码：<input type="password" name="userPass"/><br/>
    头像:<input type="file" name="headIcon"/><br/>
    <input type="submit" value="保存"/>
  </form>
```

2.3. Controller

```
@Controller
@RequestMapping("/uplad")
public class UploadController {
```

```
    @RequestMapping("test1")
    public String test1(HttpServletRequest request,User user,MultipartFile
headIcon) throws Exception{
        System.out.println(user.getUserName());
        System.out.println(user.getUserPass());

        //保存文件
        String uploadPath =
request.getServletContext().getRealPath("/upload");
        headIcon.transferTo(new
File(uploadPath+"/"+headIcon.getOriginalFilename()));

        return "success";
    }
}
```

2.4. 配置 spring-mvc.xml

```
<!-- 处理文件上传 -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolv
er">

    </bean>
```

注意：id 必须为 *multipartResolver*。

3. SpringMVC 处理 json 数据

3.1. 页面

```
<script type="text/javascript">
	$(function(){
		$("#putAndGetJson").click(function(){
			//发出一个请求，请头中携带 json 格式数据
			$.ajax({
				url:"json/test1.action",
				data:'{"userName":"jack","userPass":"123456"}',
				contentType:"application/json",
				type:"post",
				success:function(data){  // data:服务端返回的数据
					alert(JSON.stringify(data));
				},
				dataType:"json"
			});
		});

	});

</script>
```

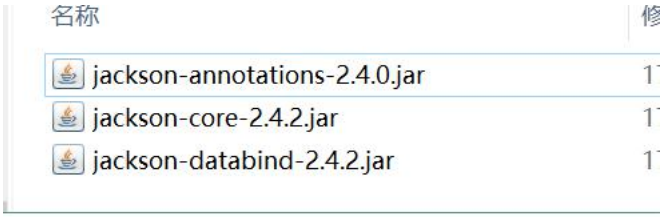
3.2. Controller

```
@Controller
@RequestMapping("/json")
```

```
public class JsonController {


    //@RequestBody：代表接收页面的 json 数据
    //@ResponseBody：代表 Controller 返回 json 数据
    @RequestMapping("/test1")
    @ResponseBody
    public User test1(@RequestBody User user){
        return user;
    }
}
```

这时导入 jackson 的插件，才可以 json 转换为 JavaBean（或 JavaBean 转换为 json）



4. SpringMVC 拦截器

拦截器的作用？

拦截器可以实现在 Controller 的方法执行的之前 和 之后加入业务逻辑。

（例如，权限判断逻辑，记录日志等）

4.1. 目标 Controller 方法

```
@Controller
@RequestMapping("/interceptor")
public class InterceptorController {
```

```
    @RequestMapping("/test1")
    public String test1(){
        System.out.println("InterceptorController 的 test1 方法");
        return "success";
    }
}
```

4.2. 编写一个 SpringMVC 拦截器

```
public class MyInterceptor1 implements HandlerInterceptor{


    //在 Controller 方法完成之后被执行
    @Override
    public void afterCompletion(HttpServletRequest arg0,
            HttpServletResponse arg1, Object arg2, Exception arg3)
            throws Exception {
        System.out.println("执行 MyInterceptor1 的 afterCompletion");
    }


    //在 Controller 方法执行之后被执行
    @Override
    public void postHandle(HttpServletRequest arg0, HttpServletResponse
arg1,
            Object arg2, ModelAndView arg3) throws Exception {
        System.out.println("执行 MyInterceptor1 的 postHandle");
    }


    //在 Controller 方法执行之前被执行
    /**
```

```
     * 该方法的返回值，代表是否继续执行 Controller 方法
     *     true：继续执行
     *     false:不执行
     */
    @Override
    public boolean preHandle(HttpServletRequest arg0, HttpServletResponse
arg1,
            Object arg2) throws Exception {
        System.out.println("执行 MyInterceptor1 的 preHandle");
        return true;
    }

}
```

4.3. 配置 spring-mvc.xml

```
<!-- 配置拦截器 -->
    <mvc:interceptors>
        <mvc:interceptor>
            <!-- 拦截的路径 -->
            <mvc:mapping path="/interceptor/test1.action"/>
            <bean class="cn.sm1234.interceptor.MyInterceptor1"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
```

4.4. 执行效果

```
执行 MyInterceptor1 的 preHandle
InterceptorController 的 test1 方法
```

```
执行 MyInterceptor1 的 postHandle
执行 MyInterceptor1 的 afterCompletion
```

4.5. 如果有多个拦截器，配置如下：

```
<!-- 配置拦截器 -->
    <mvc:interceptors>
        <mvc:interceptor>
            <!-- 拦截的路径
                注意：path 路径必须是完整的路径（包括后缀名称）
             -->
            <mvc:mapping path="/interceptor/test1.action"/>
            <bean class="cn.sm1234.interceptor.MyInterceptor1"></bean>
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/interceptor/test1.action"/>
            <bean class="cn.sm1234.interceptor.MyInterceptor2"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
```

那么多个拦截器和 Controller 执行顺序：

```
执行 MyInterceptor1 的 preHandle
执行 MyInterceptor2 的 preHandle
InterceptorController 的 test1 方法
执行 MyInterceptor2 的 postHandle
执行 MyInterceptor1 的 postHandle
执行 MyInterceptor2 的 afterCompletion
执行 MyInterceptor1 的 afterCompletion
```

5. SpringMVC 对 restful 风格的支持

5.1. 什么是 restful?

Restful 风格的 API 是一种软件架构风格，设计风格而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

在 Restful 风格中，用户请求的 url 使用同一个 url 而用请求方式：get，post，delete，put...等方式对请求的处理方法进行区分，这样可以在前后台分离式的开发中使得前端开发人员不会对请求的资源地址产生混淆和大量的检查方法名的麻烦，形成一个统一的接口。

在 Restful 风格中，现有规定如下：

GET（SELECT）：从服务器查询，可以在服务器通过请求的参数区分查询的方式。
POST（CREATE）：在服务器新建一个资源，调用 insert 操作。
PUT（UPDATE）：在服务器更新资源，调用 update 操作。
DELETE（DELETE）：从服务器删除资源，调用 delete 语句。
了解这个风格定义以后，我们举个例子：

如果当前 url 是 http://localhost:8080/User

那么用户只要请求这样同一个 URL 就可以实现不同的增删改查操作，例如

http://localhost:8080/User?_method=get&id=1001 这样就可以通过 get 请求获取到数据库 user 表里面 id=1001 的用户信息

http://localhost:8080/User?_method=post&id=1001&name=zhangsan 这样可

以向数据库 user 表里面插入一条记录

http://localhost:8080/User?_method=put&id=1001&name=lisi 这样可以将 user 表里面 id=1001 的用户名改为 lisi

http://localhost:8080/User?_method=delete&id=1001 这样用于将数据库 user 表里面的 id=1001 的信息删除

这样定义的规范我们就可以称之为 restful 风格的 API 接口，我们可以通过同一个 url 来实现各种操作。

5.2. SpringMVC 对 restful 风格的支持

5.2.1. 配置 restful 风格支持过滤器

org.springframework.web.filter.HiddenHttpMethodFilter

在 web.xml 配置这个过滤器：

```
<!-- restful 风格支持的过滤器：把 POST 请求转为指定请求方式（PUT,DELETE 等）-->
 <filter>
   <filter-name>HiddenHttpMethodFilter</filter-name>

   <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter<
/filter-class>
 </filter>
 <filter-mapping>
   <filter-name>HiddenHttpMethodFilter</filter-name>
   <url-pattern>/*</url-pattern>
 </filter-mapping>
```

5.2.2. 页面

```
  <h3>查询操作</h3>
  <form action="${pageContext.request.contextPath}/restful.action"
method="get">
    <input type="submit" value="提交"/>
  </form>

  <hr/>

  <h3>新增操作</h3>
  <form action="${pageContext.request.contextPath}/restful.action"
method="post">
    用户名：<input type="text" name="userName"/><br/>
    密码：<input type="password" name="userPass"/><br/>
    <input type="submit" value="提交"/>
  </form>

   <h3>更新操作</h3>
  <form action="${pageContext.request.contextPath}/restful.action"
method="post">
    <input type="hidden" name="_method" value="put">
    <input type="hidden" name="userId" value="1">
    用户名：<input type="text" name="userName"/><br/>
    密码：<input type="password" name="userPass"/><br/>
    <input type="submit" value="提交"/>
  </form>

  <h3>删除操作</h3>
```

```
  <form action="${pageContext.request.contextPath}/restful/1.action"
method="post">
    <input type="hidden" name="_method" value="delete">
    <input type="submit" value="提交"/>
  </form>
  </body>
```

5.2.3. Controller

```
@Controller
@RequestMapping("/restful")
public class RestfulController {


    /**
     * 查询操作
     */
    @RequestMapping(method=RequestMethod.GET)
    public String testGet(){
        System.out.println("执行 RestfulController 的 testGet");
        return "success";
    }


    /**
     * 新增操作
     */
    @RequestMapping(method=RequestMethod.POST)
    public String testPost(User user){
        System.out.println("执行 RestfulController 的 testPost");
        System.out.println(user);
```

```
        return "success";
    }

    /**
     * 更新操作
     */
    @RequestMapping(method=RequestMethod.PUT)
    public String testPut(User user){
        System.out.println("执行 RestfulController 的 testPut");
        System.out.println(user);
        return "success";
    }

    /**
     * 删除操作
     */
    @RequestMapping(value="/{id}",method=RequestMethod.DELETE)
    public String testDelete(@PathVariable("id")Integer id){
        System.out.println("执行 RestfulController 的 testDelete");
        System.out.println(id);
        return "success";
    }

}
```