Struts2 第四天

第1章 struts2 中的拦截器

1.1 Struts2 的拦截器基本概念

1.1.1 拦截器概述

在 Webwork 的中文文档的解释为——拦截器是动态拦截 Action 调用的对象。它提供了一种机制可以使开发者在定义的 action 执行的前后加入执行的代码，也可以在一个 action 执行前阻止其执行。也就是说它提供了一种可以提取 action 中可重代码，统一管理和执行的方式。

谈到拦截器，还要向大家提一个词——拦截器链 （Interceptor Chain，在 Struts 2 中称为拦截器栈 Interceptor Stack）。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 struts2 框架自己的，只有使用了 struts2 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。

拦截器它是只有进入 struts2 核心内部之后，才会起作用，如果访问的是 jsp，html,css,image 或者 js 是不会进行拦截的。

同时，拦截器还是 AOP 编程思想的具体体现形式。AOP（Aspect-Oriented Programming）简单的说就是：

在不修改源码的基础上，已有的方法进行动态增强。

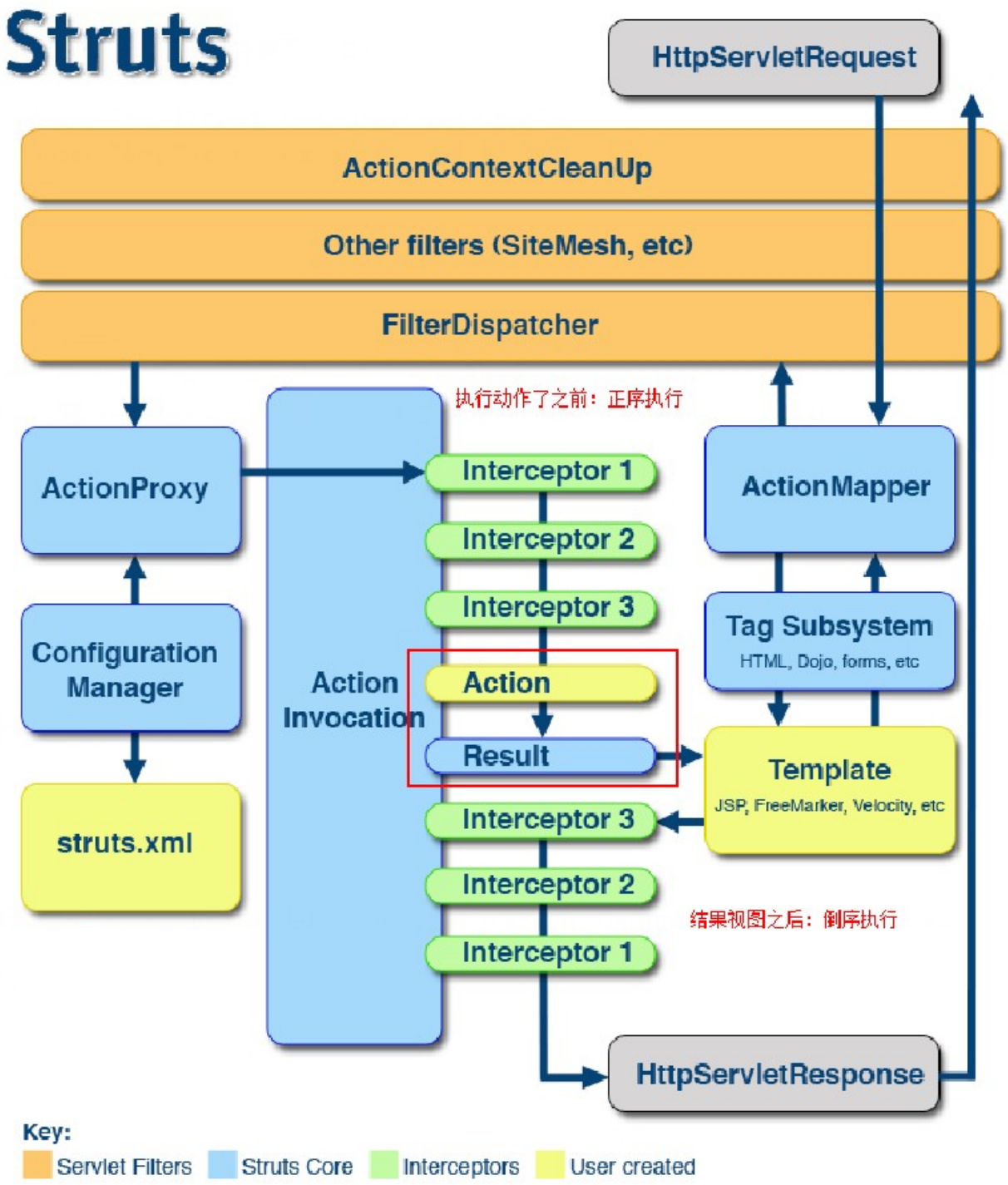
在 **struts2** 中，拦截器它就是对我们的动作方法进行增强。（其实就是把重复性的代码提取出来，然后放到拦截器中，统一管理，统一调用）

1.1.2 拦截器的作用

Struts2 中的很多功能都是由拦截器完成的。我们在第一天介绍 struts2 配置文件时，介绍了名称为 struts-default.xml 的配置文件，该配置文件中有 struts2 框架给我们提供的很多拦截器。比如：servletConfig，staticParam，params，modelDriven 等等。我们通过实现接口方式获取 ServletAPI 以及模型驱动封装请求参数，都是拦截器在帮我们做。

1.1.3 拦截器的执行时机

在访问 struts2 核心内部时，在动作方法执行之前先正序执行，然后执行动作方法，执行完动作方法和结果视图之后，再倒序执行。所以它是先进后出，是个栈的结构。具体可参考下图：



1.2 自定义拦截器

在程序开发过程中，如果需要开发自己的拦截器类，就需要直接或间接的实现 com.opensymphony.xwork2.interceptor.Interceptor 接口。其定义的代码如下：

```
public interface Interceptor extends Serializable {
    void init();
    void destroy();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

该接口提供了三个方法，其具体介绍如下。

- void init()：该方法在拦截器被创建后会立即被调用，它在拦截器的生命周期内只被调用一次. 可以在该方法中对相关资源进行必要的初始化。
- void destroy()：该方法与 init 方法相对应，在拦截器实例被销毁之前，将调用该方法来释放和拦截器相关的资源。它在拦截器的生命周期内，也只被调用一次。
- String intercept(ActionInvocation invocation) throws Exception：该方法是拦截器的核心方法，用来添加真正执行拦截工作的代码，实现具体的拦截操作。它返回一个字符串作为逻辑视图，系统根据返回的字符串跳转到对应的视图资源。每拦截一个动作请求，该方法就会被调用一次。该方法的 ActionInvocation 参数包含了被拦截的 Action 的引用，可以通过该参数的 invoke()方法，将控制权转给下一个拦截器或者转给 Action 的 execute()方法。

如果需要自定义拦截器，只需要实现 Interceptor 接口的三个方法即可。然而在实际开发过程中，除了实现 Interceptor 接口可以自定义拦截器外，更常用的一种方式是继承抽象拦截器类 AbstractIntercepter。该类实现了 Interceptor 接口，并且提供了 init()方法和 destroy()方法的空实现。使用时，可以直接继承该抽象类，而不用实现那些不必要的方法。拦截器类 AbstractInterceptor 中定义的方法如下所示：

```
public abstract class AbstractInterceptor implements Interceptor {
    public void init() {}
    public void destroy() {}
    public abstract String intercept(ActionInvocation invocation)
    throws Exception;
}
```

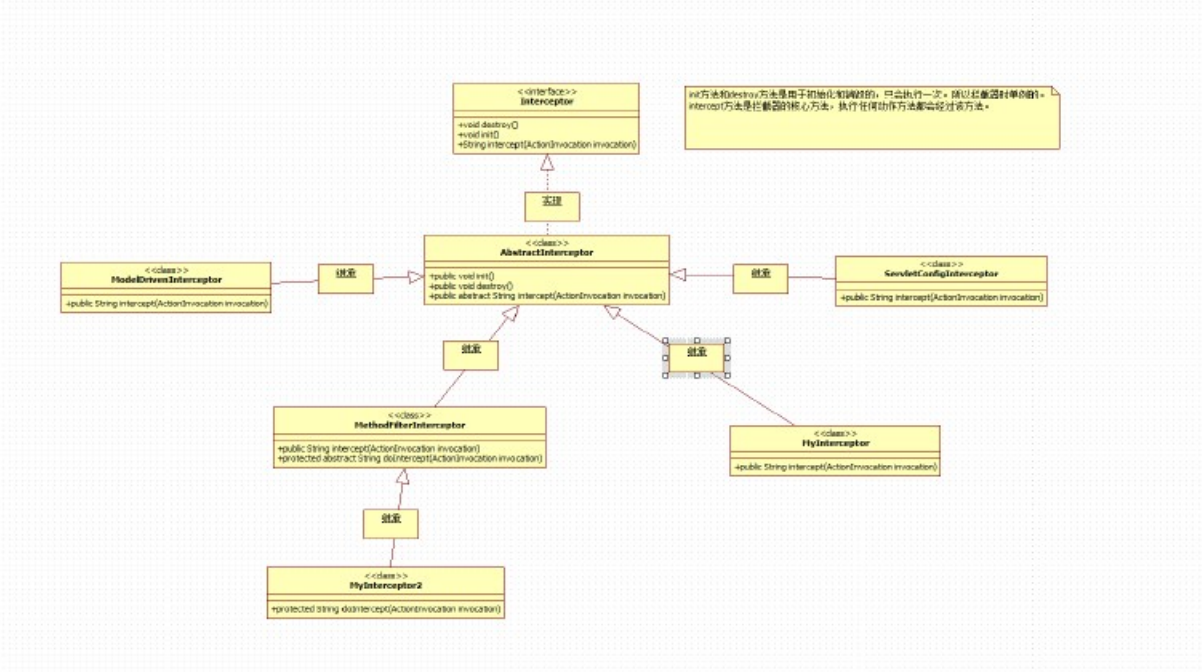
从上述代码中可以看出，AbstractInterceptor 类已经实现了 Interceptor 接口的所有方法，一般情况下，只需继承 AbstractInterceptor 类，实现 interceptor()方法就可以创建自定义拦截器。

只有当自定义的拦截器需要打开系统资源时，才需要覆盖 AbstractInterceptor 类的 init()方法和 destroy()方法。与实现 Interceptor 接口相比，继承 AbstractInterceptor 类的方法更为简单。

当然还有更简单的，AbstractInterceptor 还有一个子类，MethodFilterInterceptor，该类中提供了两个属性，可以告知拦截器对哪些方法进行拦截或者对哪些方法排除。

```
 * @version $Date$ $Id$
 */
public abstract class MethodFilterInterceptor extends AbstractInterceptor {
    protected transient Logger log = LoggerFactory.getLogger(getClass());

    protected Set<String> excludeMethods = Collections.emptySet();要排除的方法
    protected Set<String> includeMethods = Collections.emptySet();要拦截的方法

    public void setExcludeMethods(String excludeMethods) {
        this.excludeMethods = TextParseUtil.commaDelimitedStringToSet(excludeMethods);
    }

    public Set<String> getExcludeMethodsSet() {
        return excludeMethods;
    }

    public void setIncludeMethods(String includeMethods) {
        this.includeMethods = TextParseUtil.commaDelimitedStringToSet(includeMethods);
    }

    public Set<String> getIncludeMethodsSet() {
        return includeMethods;
    }
```

下图展示的就是拦截器的类结构视图：



1.2.1 自定义步骤

通过在拦截器类视图上我们可以得知，我们定义拦截器可以有三种办法：

第一种：定义一个类，实现 **Interceptor** 接口

第二种：定义一个类，继承 **AbstractInterceptor**

第三种：定义一个类，继承 **MethodFilterInterceptor**

在这三种方式中，我们选择第二种和第三种都可以。那么后两种有什么区别呢？

我们来看看 **AbstractorInteractor** 类中的代码：

```
/**
 * Provides default implementations of optional lifecycle methods
 */
public abstract class AbstractInterceptor implements Interceptor {

    /**
     * Does nothing.
     */
    public void init() {
    }

    /**
     * Does nothing.
     */
    public void destroy() {
    }

    /**
     * Override to handle interception.
     */
    public abstract String intercept(ActionInvocation invocation) throws Exception;
}
```

初始化方法，只执行一次
在应用加载读取了struts2配置文件后创建并初始化

拦截器是单例的

销毁方法，只执行一次
在应用卸载时，销毁

每次访问Action时，都执行此方法，此方法是拦截器的核心方法

我们再来看看 MethodFilterInterceptor 中的代码：

```
public abstract class MethodFilterInterceptor extends AbstractInterceptor {
    protected transient Logger log = LoggerFactory.getLogger(getClass());

    protected Set<String> excludeMethods = Collections.emptySet();
    protected Set<String> includeMethods = Collections.emptySet();

    public void setExcludeMethods(String excludeMethods) {
    }

    public Set<String> getExcludeMethodsSet() {
    }

    public void setIncludeMethods(String includeMethods) {
    }

    public Set<String> getIncludeMethodsSet() {
    }

    @Override
    public String intercept(ActionInvocation invocation) throws Exception {
        if (applyInterceptor(invocation)) {
            return doIntercept(invocation);
        }
        return invocation.invoke();
    }

    protected boolean applyInterceptor(ActionInvocation invocation) {
    }

    /**
     * Subclasses must override to implement the interceptor logic.
     *
     * @param invocation the action invocation
     * @return the result of invocation
     * @throws Exception
     */
    protected abstract String doIntercept(ActionInvocation invocation) throws Exception;

}
```

指定排除的方法
指定拦截的方法
写的都是动作方法名称。即action标签method属性的值

重写了父类的核心拦截方法

如果我们选择继承此类，则需要重写此抽象方法

看完两个类之后，我们有了结论。即：选择第三种方式，比第二种多了一个功能，就是告知拦截器哪些方法我们需要拦截，哪些方法我们不需要拦截。（注意：不要想着很傻的问题，在需要拦截和不需要拦截的属性中提供同一个方法）

根据以上的内容，我们开始编写我们自己的拦截器。

1.2.1.1 第一步：编写一个普通 java 类，继承 MethodFilterInterceptor

```
/**
 * 自定义拦截器
*/
public class MyInterceptor extends AbstractInterceptor {

    @Override
    public String intercept(ActionInvocation invocation) throws Exception {
        System.out.println("MyInterceptor 拦截了。。。。");
    }
}
```

1.2.1.2 第二步：在 struts.xml 中配置拦截器

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <!-- 开启开发者模式 -->
    <constant name="struts.devMode" value="true"></constant>

    <package name="p1" extends="struts-default">
        <!-- 声明拦截器 -->
        <interceptors>
            <interceptor name="myInterceptor"

class="com.itheima.web.interceptors.MyInterceptor"></interceptor>
        </interceptors>

        <action name="demo1"
            class="com.itheima.web.action.Demo1Action" method="demo1">
            <!-- 引用拦截器 -->
            <interceptor-ref name="myInterceptor"></interceptor-ref>
            <result name="error">/success.jsp</result>
        </action>
    </package>
</struts>
```

注意，此时我们创建动作类，配置动作类和编写访问动作了的 jsp 全都省略了。
代码如下：

```
/**
 * 一个动作类
 */
public class Demo1Action extends ActionSupport {
    public String demo1(){
        System.out.println("Demo1Action 的 demo1 方法执行了");
        return ERROR;
    }
}
```

```
<%@ 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>主页</title>
</head>
<body>
<a href="${pageContext.request.contextPath}/demo1.action">demo1</a>
</body>
</html>
```

```
<%@ 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>success.jsp</title>
</head>
<body>
<%System.out.println("success.jsp 执行了"); %>
执行成功!
</body>
</html>
```

这时候我们执行代码，发现控制的输出的只有【MyInterceptor 拦截了。。。。】，如下图：

而没有 action 的执行。首先，这说明我们的拦截器起作用了，同时也说明了我们没有放行。如何放行呢，请看下一章节。

1.2.2 拦截器的放行

上一章节，我们演示时，发现没有放行，在拦截器中如何放行呢？我们之前已经介绍过几个拦截器了，例如 ServletConfigInterceptor 拦截器，可以看看它是怎么放行的。拦截器的方式：**invocation.invoke()**方法。请看下面的代码：

```
/**
 * 自定义拦截器
*/
public class MyInterceptor extends AbstractInterceptor {

    @Override
    public String intercept(ActionInvocation invocation) throws Exception {
        System.out.println("执行动作方法之前：MyInterceptor 拦截了。。。。");
        String rtValue = invocation.invoke();//放行
        System.out.println("执行动作方法之后：MyInterceptor 拦截了。。。。");
        System.out.println(rtValue);
        return rtValue;
    }

}
```

当上面的代码执行完，发现 action 和 jsp 都执行了。那么这个 rtValue 值是什么呢？

1.2.3 拦截器的返回值

其实拦截器的返回值，就是我们 Action 中，执行的动作方法返回值。

1.2.4 多个拦截器的执行顺序

我们也可以自定义多个拦截器，那么多个拦截器之间执行顺序是靠引用拦截器时配置的顺序来决定的。

1.3 案例-检查登录拦截器

1.3.1 定义拦截器

代码如下：

```
    /**
     * 检查登录拦截器
    */
    public class CheckLoginInterceptor extends MethodFilterInterceptor {

        /**
         * 判断用户是否已经登录，没有登录的话，让他返回到登录页面
         */
        @Override
        public String doIntercept(ActionInvocation invocation) throws Exception
{
            //1.取出 Session 对象
            HttpSession                          session                          =
ServletActionContext.getRequest().getSession();
            //2.获取 session 域中的登录标记
            Object obj = session.getAttribute("userinfo");
            //3.判断 obj 是否为 null
            if(obj == null){
                return "login";
            }
            return invocation.invoke();//放行
        }
    }
```

1.3.2 配置拦截器

```
        <!-- 一个公共包 -->
        <package name="myDefault" extends="struts-default" abstract="true">
            <!-- 声明拦截器 -->
            <interceptors>
                <interceptor name="checkLogin"
                class="com.itheima.web.interceptors.CheckLoginInterceptor"/>
                <!-- 定义一个拦截器栈 -->
                <interceptor-stack name="myDefaultStack">
                    <interceptor-ref name="checkLogin">
                        <!-- 由于我们使用了继承 MethodFilterInterceptor，
                            此时我们可以告知拦截器，排除掉登录方法 -->
```

```
                    <param name="excludeMethods">login</param>
                </interceptor-ref>
                <interceptor-ref name="defaultStack"></interceptor-ref>
            </interceptor-stack>
        </interceptors>

        <!-- 修改默认拦截器栈，设置我们自定义的拦截器栈，
            这样的话我们写的所有动作都有了检查登录功能。并且排除了登录方法-->
        <default-interceptor-ref name="myDefaultStack"/>

        <!-- 全局结果视图 -->
        <global-results>
            <result name="login">/login.jsp</result>
        </global-results>
    </package>
```

1.3.3 编写和配置用于测试的 Action

action 和 jsp 使用的都是我们之前做过的功能，客户的查询列表和保存功能。

```
    /**
     * 客户的动作类
    */
    public class CustomerAction extends ActionSupport implements
ModelDriven<Customer>{

        private ICustomerService customerService = new CustomerServiceImpl();

        private Customer customer = new Customer();

        @Override
        public Customer getModel() {
            return customer;
        }
        /**
         * 此处模拟了一个登录，测试在访问此方法时，拦截器不拦截。
         * @return
         */

        public String login(){

    ServletActionContext.getRequest().getSession().setAttribute("userinfo",
"");
            return SUCCESS;
        }
```

```
    /**
     * 保存客户,使用模型驱动
     * @return
     */
    public String addCustomer(){
        //1.调用业务层，保存客户
        customerService.saveCustomer(customer);
        return "addCustomer";
    }

    /**
     * 前往添加客户页面
     * @return
     */
    public String addUICustomer(){
        return "addUICustomer";
    }

    /**
     * 查询所有客户
     * @return
     */
    private List<Customer> customers;
    public String findAllCustomer(){
        customers = customerService.findAllCustomer();
        return "findAllCustomer";
    }

    public List<Customer> getCustomers() {
        return customers;
    }

    public void setCustomers(List<Customer> customers) {
        this.customers = customers;
    }
}

<!-- 配置动作，让此包继承我们自己写的公共包 myDefault -->
<package name="customer" extends="myDefault" namespace="/customer">
    <!-- 查询所有客户 -->
    <action                                          name="findAllCustomer"
class="com.itheima.web.action.CustomerAction"
            method="findAllCustomer">
```

```
            <result name="findAllCustomer">/jsp/customer/list.jsp</result>
        </action>
        <!-- 获取客户添加页面 -->
        <action name="addUICustomer"
class="com.itheima.web.action.CustomerAction"
                method="addUICustomer">
            <result name="addUICustomer">/jsp/customer/add.jsp</result>
        </action>
        <!-- 添加客户 -->
        <action name="addCustomer"
class="com.itheima.web.action.CustomerAction"
                method="addCustomer">
            <result name="addCustomer"
type="redirect">/jsp/success.jsp</result>
        </action>
        <!-- 模拟登录 -->
        <action name="userLogin" class="com.itheima.web.action.CustomerAction"
                method="login">
            <result name="success" type="redirect">/index.jsp</result>
        </action>
    </package>
```

第2章 struts2 的注解配置

2.1 使用前提

struts2 框架，它不仅支持基于 XML 的配置方式，同时也支持基于注解配置的方式。接下来，我们就来讲解，struts2 框架如何基于注解配置。

首先我们要明确一件事：

注解和 XML 的配置，都是告知 struts2 框架，当我们 jsp 页面发送请求，根据配置执行对应动作类的方法，并根据返回值，前往指定的结果视图（jsp 页面或者其他动作）。它们只是配置的形式不一样。

其次要想使用 struts2 的注解，必须要导入一个新的 jar 包。该 jar 包是：

struts2-convention-plugin-2.3.24.jar

2.2 常用注解

2.2.1 @NameSpace

出现的位置：

它只能出现在 package 上或者 Action 类上。一般情况下都是写在 Action 类上。

作用：

指定当前 Action 中所有动作方法的名称空间。

属性：

value：指定名称空间的名称。写法和 xml 配置时一致。不指定的话，默认名称空间是""。

示例：

```
@Namespace("/customer")
public class CustomerAction extends ActionSupport implements
ModelDriven<Customer> {
    private Customer customer = new Customer();

    @Override
    public Customer getModel() {
        return customer;
    }
}
```

2.2.2 @ParentPackage

出现的位置：

它只能出现在 package 上或者 Action 类上。一般情况下都是写在 Action 类上。

作用：

指定当前动作类所在包的父包。由于我们已经是在类中配置了，所以无需在指定包名了。

属性：

value：指定父包的名称。

示例：

```
@ParentPackage("struts-default")
public class CustomerAction extends ActionSupport implements
ModelDriven<Customer> {
    private Customer customer = new Customer();

    @Override
    public Customer getModel() {
        return customer;
    }
}
```

2.2.3 @Action

出现的位置：

它只能出现在 Action 类上或者动作方法上。一般情况下都是写在动作方法上。

作用：

指定当前动作方法的动作名称。也就是 xml 配置时 action 标签的 name 属性。

属性：

value：指定动作名称。

results[]：它是一个数组，数据类型是注解。用于指定结果视图。此属性可以没有，当没有该属性时，表示不返回任何结果视图。即使用 response 输出响应正文。

interceptorRefs[]：它是一个数组，数据类型是注解。用于指定引用的拦截器。

示例：

```
/**
 * 获取添加客户页面
 * @return
 */
@Action(value="addUICustomer",results={
        @Result(name="addUICustomer",location="/jsp/customer/add.jsp")
})
public String addUICustomer(){
    return "addUICustomer";
}
```

2.2.4 @Result

出现的位置：

它可以出现在动作类上，也可以出现在 Action 注解中。

作用：

出现在类上，表示当前动作类中的所有动作方法都可以用此视图。

出现在 Action 注解中，表示当前 Action 可用此视图。

属性：

name：指定逻辑结果视图名称。

type：指定前往视图的方式。例如：请求转发，重定向，重定向到另外的动作。

location：指定前往的地址。可以是一个页面，也可以是一个动作。

示例：

```
/**
 * 保存客户
 * @return
 */
@Action(value="addCustomer",results={

@Result(name="addCustomer",type="redirect",location="/jsp/success.jsp")
})
public String addCustomer(){
    customerService.saveCustomer(customer);
    return "addCustomer";
}
```

2.2.5 @Results

出现的位置：

它可以出现在动作类上，也可以出现在 Action 注解中。

作用：

用于配置多个结果视图。

属性：

value：它是一个数组，数据类型是 result 注解。

示例：

```
@Results({
    @Result(name="login",location="/login.jsp"),
    @Result(name="error",location="/error.jsp")
})
public class CustomerAction extends ActionSupport implements
ModelDriven<Customer> {

    private Customer customer = new Customer();

    @Override
    public Customer getModel() {
        return customer;
    }
}
```

2.2.6 @InterceptorRef

出现的位置：

它可以出现在动作类上或者 Action 注解中。

作用：

用于配置要引用的拦截器或者拦截器栈

属性：

value：用于指定拦截器或者拦截器栈

示例：

出现在动作方法上：

```
/**
 * 查询所有客户
 * @return
 */
@Action(value="findAllCustomer",results={
        @Result(name="findAllCustomer",location="/jsp/customer/list.jsp")
},interceptorRefs={
        @InterceptorRef("myDefaultStack")
})
```

```
    public String findAllCustomer(){
        customers = customerService.findAllCustomer();
        return "findAllCustomer";
    }
```

出现在动作类上：

```
    @InterceptorRef("myDefaultStack")
    public class CustomerAction extends ActionSupport implements
ModelDriven<Customer> {
        private Customer customer = new Customer();

        @Override
        public Customer getModel() {
            return customer;
        }
    }
```

myDefaultInterceptor 的定义需要写在配置文件中，**struts.xml** 中定义的内容如下：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
        "http://struts.apache.org/dtds/struts-2.3.dtd">
    <struts>
        <!-- 开启开发者模式 -->
        <constant name="struts.devMode" value="true"></constant>

        <!-- 修改默认拦截后缀 -->
        <constant name="struts.action.extension" value="action"></constant>

        <!-- 修改 struts2 的主题 -->
        <constant name="struts.ui.theme" value="simple"></constant>

        <!-- 定义一个公共包，继承 struts2 的核心包 struts-default，并且声明为抽象包 -->
        <package name="myDefault" extends="struts-default" abstract="true">
            <interceptors>
                <!-- 声明拦截器 -->
                <interceptor name="checkLogin"

    class="cn.itcast.web.interceptors.CheckLoginInterceptor"/>
                <!-- 声明拦截器栈，把检查登录拦截器和默认拦截器栈组合成一个新的拦截器栈
-->
                <interceptor-stack name="myDefaultStack">
                    <interceptor-ref name="checkLogin"></interceptor-ref>
                    <interceptor-ref name="defaultStack"></interceptor-ref>
```

```
            </interceptor-stack>
        </interceptors>
        <!-- 把声明的拦截器栈设置为当前项目的默认拦截器栈 -->
        <default-interceptor-ref name="myDefaultStack"/>
    </package>
</struts>
```

2.3 注解实现客户保存和查询列表

2.3.1 前提

案例的选用，我们使用的是之前已经完成的客户保存和列表查询的案例。只是把 **struts2** 部分的配置改为用注解实现。

2.3.2 代码实现

2.3.2.1 拷贝必备 jar 包


crm_struts2_hibernate
JAX-WS Web Services
Deployment Descriptor: crm_struts2_hibernate
Java Resources
JavaScript Resources
build
WebContent
css
images
js
jsp
META-INF
WEB-INF
lib
antlr-2.7.7.jar
asm-3.3.jar
asm-commons-3.3.jar
asm-tree-3.3.jar
c3p0-0.9.2.1.jar
commons-fileupload-1.3.1.jar
commons-io-2.2.jar
commons-lang3-3.2.jar
dom4j-1.6.1.jar
freemarker-2.3.22.jar
geronimo-jta_1.1_spec-1.1.1.jar
hibernate-c3p0-5.0.7.Final.jar
hibernate-commons-annotations-5.0.1.Final.jar
hibernate-core-5.0.7.Final.jar
hibernate-jpa-2.1-api-1.0.0.Final.jar
jandex-2.0.0.Final.jar
javassist-3.11.0.GA.jar
javassist-3.18.1-GA.jar
jboss-logging-3.3.0.Final.jar
jstl-1.2.jar
junit-4.9.jar
log4j-1.2.16.jar
log4j-api-2.2.jar
log4j-core-2.2.jar
mchange-commons-java-0.2.3.4.jar
mysql-connector-java-5.1.7-bin.jar
ognl-3.0.6.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.7.2.jar
struts2-convention-plugin-2.3.24.jar
struts2-core-2.3.24.jar
xwork-core-2.3.24.jar
web.xml
index.jsp
login.jsp
menu.jsp
top.htm
welcome.htm

2.3.2.2 使用注解配置 Action

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <!-- 开启开发者模式 -->
    <constant name="struts.devMode" value="true"></constant>
</struts>

Action 中的代码
/**
 * 客户的动作类
*/
@Namespace("/customer")
@ParentPackage("struts-default")
public class CustomerAction extends ActionSupport implements
ModelDriven<Customer>{

    private ICustomerService customerService = new CustomerServiceImpl();

    private Customer customer = new Customer();

    @Override
    public Customer getModel() {
        return customer;
    }


    /**
     * 保存客户,使用模型驱动
     * @return
     */
    @Action(value="addCustomer",results={

@Result(name="addCustomer",type="redirect",location="/jsp/success.jsp")
    })
    public String addCustomer(){
        //1.调用业务层，保存客户
        customerService.saveCustomer(customer);
        return "addCustomer";
    }
```

```
        /**
         * 前往添加客户页面
         * @return
         */
        @Action(value="addUICustomer",results={

    @Result(name="addUICustomer",type="dispatcher",location="/jsp/customer/add.jsp")
        })
        public String addUICustomer(){
            return "addUICustomer";
        }

        /**
         * 查询所有客户
         * @return
         */
        @Action(value="findAllCustomer",results={

    @Result(name="findAllCustomer",location="/jsp/customer/list.jsp")
        })
        public String findAllCustomer(){
            //1.使用 service 对象查询所有客户
            List<Customer> customers = customerService.findAllCustomer();
            //2.获取 request 对象
            HttpServletRequest request = ServletActionContext.getRequest();
            //3.存入 request 域中
            request.setAttribute("customers", customers);
            return "findAllCustomer";
        }

    }
```