第5章 拦 截 器

本章导读

拦截器是动态拦截 Action 调用的对象，在执行 Action 的方法之前或之后，Struts2 会首先执行 struts.xml 中引用的拦截器。拦截器是 Struts2 框架的基石，许多功能的实现都是构建在拦截器的基础之上的。

本章要点

- 拦截器的编写
- 拦截器的配置
- 拦截器的应用

5.1 拦截器概述

在软件开发阶段，往往由于前期设计不合理，或者缺乏预见性，可能会导致系统在多个地方需要使用相同的代码，这会造成代码的重复编写，更重要的是给软件日后升级与维护带来极大的不便。例如有些 Action 需要输入验证，有的文件上传需要预处理，有的需要禁止重复提交，有的需要下拉列表等等，利用拦截器可以很好地解决这些问题。

拦截器指在 AOP 中用于在某个方法或字段被访问之前或之后，进行拦截然后加入某些操作，这是 Struts2 框架的一种通用解决方案。

5.1.1 AOP 简介

Aspect Oriented Programming(AOP)是面向切面编程，也叫面向方面编程，是目前软件开发中的一个热点。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

应用 AOP 主要的目的是将日志记录、性能统计、安全控制、事务处理、异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，将它们独立到方法中，进而改变这些行为的时候不影响业务逻辑的代码。

使用 Java 的开发人员比较熟悉 OOP(面向对象编程)的方法，那么 OOP 和 AOP 有什么区别呢？OOP 针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。而 AOP 则是针对业务处理过程中的切面进行提取，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。这两种设计思想在目标上有着本质的差异。

如果说面向对象编程是关注将需求功能划分为不同的并且相对独立，封装良好的类，并

让它们有着属于自己的行为，依靠继承和多态等来定义彼此的关系的话；面向切面编程则是将需求功能从不相关的类中分离出来，使得很多类共享一个行为，一旦发生变化，不必修改很多类，只需修改这个行为就可以。

5.1.2 拦截器原理

拦截器处理机制来源于 WebWork，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人员定义在一个 Action 执行的前后执行的代码，也可以在一个 Action 执行前阻止其执行。同时也是提供了一种可以提取 Action 中可重用部分的方式。

拦截器是 AOP 的一种实现策略。当用户请求 Action 时，Struts2 框架会激活 Action 对象。如果定义了拦截器，在 Action 对象被执行前，激活程序会被另一个对象拦截。在 Action 执行完毕之后，激活程序同样可以被拦截。Struts2 拦截器体系可以动态拦截 Action 调用的对象，开发人员只需要提供拦截器的实现类，并将其配置在 struts.xml 文件中即可。

当一个 Action 需要定义多个拦截器时，通常可以将多个拦截器按一定的顺序连接成一条链，在访问被拦截的方法或者字段时，拦截器链中的拦截器会按照定义的顺序被调用，如图 5.1 所示。

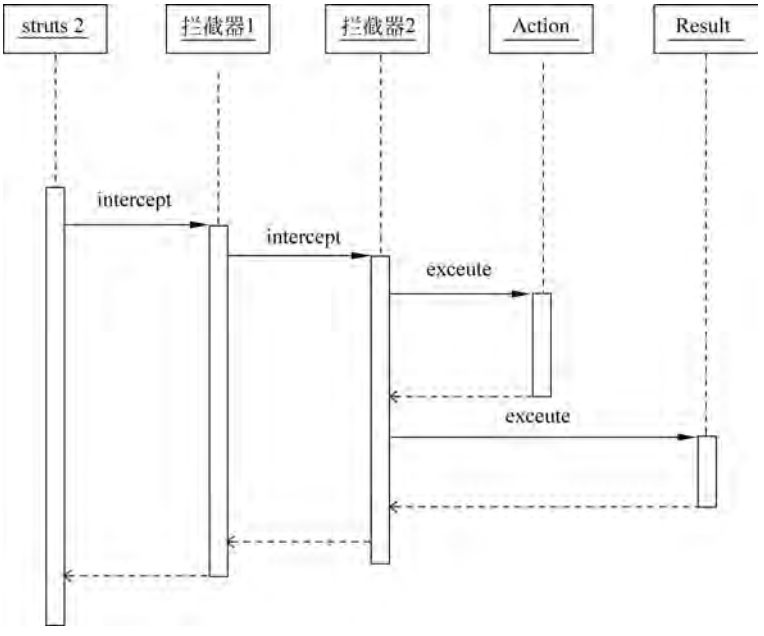

图 5.1 拦截器的工作时序图

Struts2 的拦截器一层一层把 Action 包裹在最里面，每个 Action 请求都包装在拦截器内部，整个结构就如同一个堆栈，Action 位于堆栈的底部，由于堆栈“后进先出”的特性，如果试图把 Action 拿出来执行，必须首先把位于 Action 上端的 Interceptor 拿出来执行。拦截器可以在 Action 执行之前做准备操作，也可以在 Action 执行之后做回收操作，这样整个执行就形成了一个递归调用。Action 既可以将操作转交给下面的拦截器，也可以直接退出

操作，返回客户已经定义好的视图资源。每个位于堆栈中的 Interceptor，除了需要完成它自身的逻辑，还需要完成一个特殊的执行职责。这个执行职责有 3 种选择：

（1）中止整个执行，直接返回一个字符串。

（2）通过递归调用负责调用堆栈中下一个 Interceptor 的执行。

（3）如果在堆栈内已经不存在任何的 Interceptor，调用 Action。

拦截器的执行是通过代理的方式来实现的，当请求到达 Struts2 框架时，Struts2 会查找配置文件，并根据配置实例化相应的拦截器对象，然后将这些对象串成一个列表，最后逐个调用列表中的拦截器。拦截器和 Action 之间的关系如图 5.2 所示。

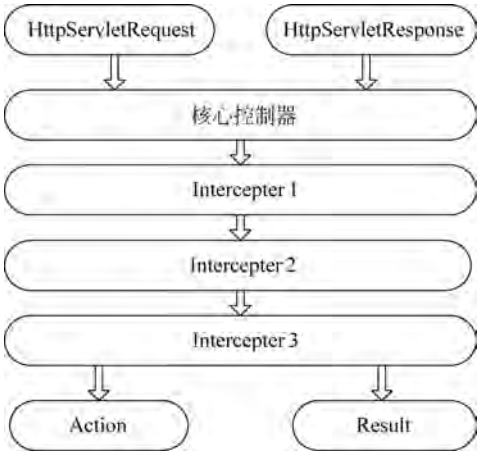

图 5.2　拦截器与 Action 的关系

5.1.3　内置拦截器

Struts2 内建了大量的拦截器，比如解析请求参数、参数类型转换、将请求参数值赋给 Action 的属性和数据校验等。这些拦截器以 name-class 对的形式配置在 struts-default.xml 文件中，其中 name 是拦截器的名字，就是以后使用该拦截器的唯一标识；class 则指定了该拦截器的实现类，常用的内置拦截器如下：

- alias：实现在不同请求中相似参数别名的转换。
- autowiring：这是个自动装配的拦截器，主要用于当 Struts2 和 Spring 整合时，Struts2 可以使用自动装配的方式来访问 Spring 容器中的 Bean。
- chain：构建一个 Action 链，使当前 Action 可以访问前一个 Action 的属性，一般和 <result type="chain" .../>一起使用。
- conversionError：这是一个负责处理类型转换错误的拦截器，它负责将类型转换错误从 ActionContext 中取出，并转换成 Action 的 FieldError 错误。
- createSession：该拦截器负责创建一个 HttpSession 对象，主要用于那些需要有 HttpSession 对象才能正常工作的拦截器中。
- debugging：当使用 Struts2 的开发模式时，这个拦截器会提供更多的调试信息。
- execAndWait：后台执行 Action，负责将等待画面发送给用户。
- exception：这个拦截器负责处理异常，它将异常映射为结果。

- fileUpload：这个拦截器主要用于文件上传，它负责解析表单中文件域的内容。
- i18n：这是支持国际化的拦截器，它负责把所选的语言、区域放入用户 Session 中。
- logger：这是一个负责日志记录的拦截器，主要是输出 Action 的名字。
- model-driven：这是一个用于模型驱动的拦截器，当某个 Action 类实现了 ModelDriven 接口时，它负责把 getModel()方法的结果堆入 ValueStack 中。
- scoped-model-driven：如果一个 Action 实现了一个 ScopedModelDriven 接口，这个拦截器负责从指定生存范围中找出指定的 Model，并将通过 setModel 方法将该 Model 传给 Action 实例。
- params：这是最基本的一个拦截器，它负责解析 HTTP 请求中的参数，并将参数值设置成 Action 对应的属性值。
- prepare：如果 Action 实现了 Preparable 接口，将会调用该拦截器的 prepare()方法。
- static-params：这个拦截器负责将 xml 中< action >标签下< param >标签中的参数传入 Action。
- scope：这是范围转换拦截器，它可以将 Action 状态信息保存到 HttpSession 范围，或者保存到 ServletContext 范围内。
- servlet-config：如果某个 Action 需要直接访问 Servlet API，就是通过这个拦截器实现的。注意：尽量避免在 Action 中直接访问 Servlet API，这样会导致 Action 与 Servlet 的高耦合。
- roles：这是一个 Java 授权和认证服务(Java Authentication and Authorization Service，JAAS)拦截器，只有当浏览者取得合适的授权后，才可以调用被该拦截器拦截的 Action。
- timer：这个拦截器负责输出 Action 的执行时间，这个拦截器在分析该 Action 的性能瓶颈时比较有用。
- token：这个拦截器主要用于阻止重复提交，它检查传到 Action 中的 token，从而防止多次提交。
- token-session：这个拦截器的作用与前一个基本类似，只是它把 token 保存在 HttpSession 中。
- validation：通过执行在 xxxAction-validation. xml 中定义的校验器，从而完成数据校验。
- workflow：这个拦截器负责调用 Action 类中的 validate 方法，如果校验失败，则返回 input 的逻辑视图。

大部分时候，开发人员无须手动控制这些拦截器，因为 struts-default. xml 文件中已经配置了这些拦截器，所以只需要在 struts. xml 文件中通过“< include file＝"struts-default. xml" />”将 struts-default. xml 文件包含进来，并继承其中的 struts-default 包，最后在定义 Action 时，使用“< interceptor-ref name＝"xx" />”引用拦截器或拦截器栈(interceptor stack)，就可以使用相应的拦截器。

5.1.4 案例

案例 1 本例演示拦截器 timer 的用途，用于显示执行某个 action 方法的耗时，在实际

开发中，开发人员能够借此做一个粗略的性能调试。

首先，新建 Action 类 TimerInterceptorAction.java，代码见例 5.1 和例 5.2。

例 5.1　TimerInterceptorAction.java

```
package com;
import com.opensymphony.xwork2.ActionSupport;
    public class TimerInterceptorAction extends ActionSupport {
      public String execute() {
          try {
             //模拟耗时的操作
            Thread.sleep( 500 );
          } catch (Exception e) {
             e.printStackTrace();
         }
          return SUCCESS;
    }
}
```

例 5.2　struts.xml

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE struts PUBLIC " - //Apache Software Foundation//DTD Struts Configuration 2.1//EN" "
http://struts.apache.org/dtds/struts - 2.1.dtd">
< struts >
  < include file = "struts - default.xml"/>
    < package name = "InterceptorDemo" extends = "struts - default" >
        < action name = "Timer" class = "com.TimerInterceptorAction" >
            < interceptor - ref name = "timer" />
            < result >/index.jsp </result >
        </action >
    </package >
</struts >
```

运行该程序，控制台输出结果见图 5.3。在不同的环境中执行 Timer! execute 的耗时，可能时间有些不同，这取决于 PC 的性能。第一次加载 Timer 时，需要进行一定的初始工作，时间可能会多些，当再次重新请求 Timer.action 时，时间就会变为 500ms。

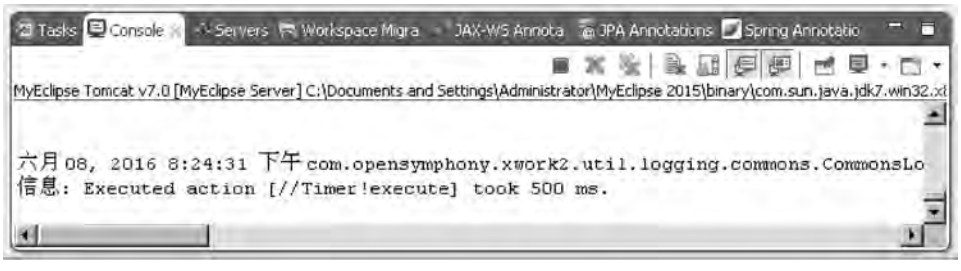

图 5.3　拦截器 timer 的使用

5.2 自定义拦截器

作为框架，可扩展性是不可或缺的。虽然 Struts 2 提供如此丰富的拦截器实现，但是这并不意味就失去创建自定义拦截器的能力，恰恰相反，Struts2 自定义拦截器是相当容易的一件事。

5.2.1 拦截器类的编写

Struts2 提供了 Interceptor 接口，Struts2 规定用户自定义拦截器必须直接或间接实现 com. opensymphony. xwork2. interceptor. Interceptor 接口，该接口的定义如下所示：

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

通过上面的代码可以看出，该接口中有三个方法：

- init()：该方法在拦截器被实例化之后，拦截器执行之前调用。对于每个拦截器而言，该方法只被执行一次，主要用于初始化资源，例如数据库连接参数等。
- intercept(ActionInvocation invocation)：该方法用于实现拦截的动作，方法会返回一个字符串作为逻辑视图。该方法的参数 ActionInvocation 包含了被拦截的 Action 的引用，用该参数调用其 invoke 方法，将控制权交给下一拦截器，或者交给 Action 类的方法。
- destroy()：该方法与 init()方法对应，拦截器实例被销毁之前调用，用于销毁在 init()方法中打开的资源。

另外，Struts2 提供了一个抽象拦截器类 AbstractInterceptor，这个类提供了 init()和 destroy()方法的空实现，因此继承该类实现拦截器类是更加简单的一种方式，因为并不是每次实现拦截器都要申请资源，那么只需要实现 intercept 方法就可以。

类 AbstractInterceptor 代码片段：

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

下面使用继承父类 AbstractInterceptor 的方式开发一个简单的拦截器，代码如下：

```
public classInterceptorTest extends AbstractInterceptor
{
```

```
public String intercept(ActionInvocation invocation) throws Exception {
    System.out.println("Action 执行前插入代码");
    //执行目标方法 (调用下一个拦截器, 或执行 Action)
    final String res = invocation.invoke();
    System.out.println("Action 执行后插入 代码");
    return res;
}
}
```

通过上面的代码可以看出,开发人员要自定义拦截器类,只需要继承 AbstractInterceptor 类,然后重写 intercept 方法就可以了。

注意:拦截器必须是无状态的,换句话说,在拦截器类中不应该有实例变量。这是因为 Struts2 对每一个 Action 的请求使用的是同一个拦截器实例来拦截调用,如果拦截器有状态,在多个线程(客户端的每个请求将由服务器端的一个线程来服务)同时访问一个拦截器实例的情况下,拦截器的状态将不可预测。

5.2.2 拦截器的配置

使用拦截器需要在 struts.xml 中配置拦截器,相关的配置元素如表 5.1 所示。

表 5.1 拦截器配置元素

元素名称	元素含义
<interceptors>	用来定义拦截器,所有拦截器与拦截器栈都在此元素下定义。可以包含子元素<interceptor>和<interceptor-stack>,分别用来定义拦截器和拦截器栈
<interceptor>	用来定义拦截器,需要指定拦截器的名字和拦截器的类
<interceptor-stack>	用来定义拦截器栈,可以引入其他拦截器或拦截器栈
<interceptor-ref>	用来引用其他拦截器或拦截器栈,作为<interceptor-stack>和<action>元素的子元素
<param>	用来为拦截器指定参数,可以作为<interceptor>或<interceptor-ref>的子元素

1. 拦截器的配置

拦截器的配置主要是在 struts.xml 文件中来定义的。定义拦截器使用<interceptor>元素。其基本语法格式为:

```
<interceptor name="拦截器名" class="拦截器实现类"></interceptor>
```

例如:

```
<package name="my" extends="struts-default" namespace="/manage">
    <interceptors>
        <!-- 定义拦截器 -->
        <interceptor name="checkLogin" class="com.LoginInterceptors"/>
    </interceptors>
</package>
```

例子中使用<interceptor>元素配置拦截器,拦截器名为 checkLogin,具体实现的拦截

器类是放在 com 包下的 LoginInterceptors.java 文件。

2. 传递参数

如果调用拦截器时需要传递参数，只要在< interceptor >与</interceptor >之间配置< param >子元素即可传入相应的参数。其格式如下：

```
< interceptor name = "myInterceptor" class = "org.tool.MyInterceptor">
    < param name = "参数名">参数值</param >
</interceptor >
```

例如：

```
< package name = "my" extends = "struts - default" namespace = "/manage">
    < interceptors >
        <!-- 定义拦截器 -- >
        < interceptor name = "checkLogin" class = "com.LoginInterceptors">
        < param name = "workNum"> 1008 </param >
        </interceptor >
</interceptors >
</package >
```

上例中调用拦截器类时传递了一个参数，参数名为 workNum，参数值为 1008。

3. 拦截器栈（Interceptor Stack）

拦截器栈就是将拦截器按一定的顺序联结成一个集合。

拦截器栈示例：

```
< package name = "my" extends = "struts - default" namespace = "/manage">
    < interceptors >
        <!-- 定义拦截器 -- >
        < interceptor name = "拦截器名" class = "拦截器实现类"/>
        <!-- 定义拦截器栈 -- >
        < interceptor - stack name = "拦截器栈名">
            < interceptor - ref name = "拦截器一"/>
            < interceptor - ref name = "拦截器二"/>
        </interceptor - stack >
    </interceptors >
        ……
</package >
```

例如：

```
< package name = "my" extends = "struts - default" namespace = "/manage">
    < interceptors >
        <!-- 定义拦截器 -- >
        < interceptor name = "BookInter1" class = "com. Inter1"/>
< interceptor name = "BookInter2" class = "com. Inter2"/>
        <!-- 定义拦截器栈 -- >
        < interceptor - stack name = " BookInterStack">
```

```
            <interceptor-ref name="BookInter1"/>
            <interceptor-ref name="BookInter1"/>
        </interceptor-stack>
    </interceptors>
```

上例中配置了两个拦截器 BookInter1 和 BookInter2，如果想两个拦截器都使用，那么就配置一个拦截器栈 BookInterStack，包含两个拦截器，然后只要应用拦截器栈 BookInterStack，拦截器 BookInter1 和 BookInter2 都会被调用。

4. Action 中应用拦截器

一旦定义了拦截器和拦截器栈后，就可以使用这个拦截器或拦截器栈来拦截 Action 了。使用<interceptor-ref>元素在 Action 类中应用拦截器，其配置的语法与在拦截器栈中引用拦截器的语法完全一样，语法如下：

```
<package name="my" extends="struts-default" namespace="/manage">
    <interceptors>
        <!-- 定义拦截器 -->
        <interceptor name="拦截器名" class="拦截器实现类"/>
        </interceptors>
        <!-- Action 中使用拦截器 -->
        <action name="action 类名" class="action 类路径">
            <result name="结果">/运行的文件</result>
            <interceptor-ref name="拦截器一"/>
            <interceptor-ref name="拦截器二"/>
        </action>
    </package>
```

示例如下：

```
<package name="default" extends="struts-default">
    <interceptors>
        <interceptor name="myInterceptor" class="org.tool.MyInterceptor"></interceptor>
    </interceptors>
    <action name="struts" class="org.action.StrutsAction">
        <result name="success">/welcome.jsp</result>
        <result name="error">/hello.jsp</result>
        <result name="input">/hello.jsp</result>
        <!-- 拦截配置在 result 后面 -->
        <!-- 使用系统默认拦截器栈 -->
        <interceptor-ref name="defaultStack"></interceptor-ref>
        <!-- 配置拦截器 -->
        <interceptor-ref name="myInterceptor"></interceptor-ref>
    </action>
</package>
```

示例中定义了一个拦截器 myInterceptor，在 Action 元素中应用<interceptor-ref>调用该拦截器进行拦截操作。注意配置拦截器，一般放在<result>元素之后。

5. 拦截器方法过滤

默认的情况下，如果为某个 Action 配置拦截器，则该拦截器会拦截 Action 中的所有方法。但是有时并不想拦截所有的方法，而是只需要拦截其中的某几个方法，此时就需要使用 Struts2 中的拦截器方法过滤特性。

Struts2 提供一个 MethodFilterInterceptor 抽象类，该类是 AbstractInterceptor 类的子类，重写了 intercept(ActionInvocation invocation)方法，但是提供了一个 doIntercept(ActionInvocation invocation)抽象方法。因此，开发人员实现过滤器方法过滤的特性，需要重写 doIntercept 方法。另外，MethodFilterInterceptor 类中增加了如下两个方法：

public void setExcludeMethods (String excludeMethods)：排除需要过滤的方法，设置方法的“黑名单”，所有在 excludeMethods 中列出的方法都不会被拦截。

public void setIncludeMethods (String includeMethods)：设置需要过滤的方法，设置方法的“白名单”，所有在 includeMethods 中列出的方法都会被拦截。

如果一个方法同时在 excludeMethods 和 includeMethods 中出现，则该方法会被拦截。

示例代码如下：

```
<interceptor name="myMethodInterceptor" class="interceptor.MyMethodIntercepto"/>
<interceptor-ref name="myMethodInterceptor">
    <param name="includeMethods">execute,login</param>
    <param name="excludeMethods">test</param>
</interceptor-ref>
```

上面配置文件中粗体代码设置了方法的过滤，includeMethods 属性指定了 execute，login 方法会被拦截，方法名之间以英文逗号隔开。excludeMethods 属性指定 test 方法不会被拦截器拦截。

5.2.3 默认拦截器

1. 系统默认拦截器栈

在 struts-default.xml 文件中定义了系统的默认拦截器栈 defaultStack。Struts2 框架的大部分功能都是通过这个默认拦截器栈来实现的，比如，开发人员没有配置参数解析的拦截器，但是系统却具有对客户端的请求参数进行解析等各种功能。这是因为如果开发人员没有为 Action 指定拦截器，系统就会以默认拦截器栈 defaultStack 来拦截 Action。如果开发人员为 Action 指定拦截器，那么系统就不再使用默认拦截器栈，需要开发人员手动配置。

```
<actionname="userOpt" class="org.gz. UserAction">
    <result name="success">/success.jsp</result>
    <result name="error">/error.jsp</result>
    <!-- 使用拦截器，一般配置在 result 之后 -->
    <!-- 引用系统默认的拦截器 -->
    <interceptor-ref name="defaultStack"/>
     <interceptor-ref name="拦截器名或拦截器栈名"/>
</action>
```

如果为 Action 指定了一个拦截器，则系统默认的拦截器栈将会失去作用。为了继续使

用默认拦截器，所以粗体代码为手动引入了系统的默认拦截器。

2. 自定义默认拦截器

用户在 struts.xml 文件中配置一个包时，可以为其指定默认拦截器，一旦为某个包指定了默认拦截器，如果该包中的某些 Action 没有显示指定其他拦截器，则默认拦截器就会起作用。

自定义默认拦截器语法：

```
<default-interceptor-ref name="拦截器(或拦截器栈)的名字">
```

配置用户自定义的默认拦截器需要使用<default-interceptor-ref>元素，此元素为<package>元素的子元素，配置该元素时，需要指定 name 属性，该 name 属性值必须是已经存在的拦截器名字，表明将该拦截器设置为默认的拦截器。

示例：

```
<package name="default" extends="struts-default">
<interceptors>
    <interceptor name="myInterceptor" class="org.tool.MyInterceptor"></interceptor>
</interceptors>
<default-interceptor-ref name="myInterceptor">
    <action name="struts" class="org. strutsAction">
        <result name="success">/success.jsp</result>
      <result name="error">/error.jsp</result>
    </action>
</package>
```

该例中定义 myInterceptor 为默认拦截器，在<action>中没有显示配置其他的拦截器，那么调用 strutsAction 时，会默认执行 myInterceptor 拦截器。

在一个包中只能配置一个默认拦截器，如果配置多个默认拦截器，那么系统就无法确认到底哪个才是默认拦截器，但是如果需要把多个拦截器都配置为默认拦截器，可以把这些拦截器定义为一个拦截器栈，然后把这个栈配置为默认拦截器。

5.2.4 案例

案例 2 文字过滤拦截器实例

网上论坛要求会员发帖的文字信息要文明，通常会进行过滤，如果发现不文明或者敏感的词语，会用“*”来代替。在 Struts2 框架中可以使用拦截器来实现这个功能。

例如某论坛中，网友们可以自由评论，如果出现“不文明”的字样，就通过拦截器对其过滤，以“*”显示出来。发表评论页面见例 5.3。

例 5.3 news.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="utf-8"%>
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
```

```
<title>论坛</title>
</head>
<body>
    <center>
        <h2>网友评论</h2>
        <s:form action="PublishAction"  method="post" theme="simple">
        <s:textfield name="title" lable="标题" size="45" style="vertical-align: top">
标题:</s:textfield><br/><br/>
        <s:textarea name="content" lable="内容" cols="40" rows="5" style="vertical-
align: top">内容:</s:textarea><br/><br/>
            <s:submit value="提交"/>
            <s:reset value="重置"/>
        </s:form>
    </center>
</body>
</html>
```

该页面表单中定义两个控件,标题 title 与评论的内容 content,单击【提交】按钮,就将表单中的内容提交给 Action 类来处理,代码见例 5.4。

例 5.4　PublishAction 类

```
package com;
import com.opensymphony.xwork2.*;
public class PublishAction extends ActionSupport {
    public PublishAction() {
    }
    private String title;
    private String content;
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    public String execute() {
        return SUCCESS;
    }
}
```

Action 类中,定义了两个属性 title 和 content,分别与表单的控件名一一对应,每个属性都有一对 set 和 get 方法,在 execute()方法中没有进行具体操作,直接返回一个 SUCCESS 的逻辑视图。代码见例 5.5。

例 5.5　MyInterceptor 类

```
package com;
import com.PublishAction;
import java.util.*;
import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
public class MyInterceptor extends AbstractInterceptor{
    public String intercept(ActionInvocation act) throws Exception {
        Object object = act.getAction();
        if (object != null) {
            if (object instanceof PublishAction) {
                PublishAction action = (PublishAction) object;
                String content = action.getContent();
                if (content.contains("不文明")) {
                    content = content.replaceAll("不文明", "*");
                    action.setContent(content);
                }
                return act.invoke();
            } else {
                return Action.LOGIN;
            }
        } else {
            return Action.LOGIN;
        }
    }
}
```

MyInterceptor 拦截器类中重写了 intercept()方法，这个方法有一个参数 ActionInvocation 对象，它是由框架传递过来的，通过这个对象可以获得相关联的 Action 对象。另外需要引用系统的默认拦截器栈 defaultStack，这样，用户提交的数据都被保存在 Action 的属性中，那么就可以检查属性 content 中是否包含需要过滤的文字，如果有，就替换成“*”，然后把替换后的结果赋值给 content。代码见例 5.6。运行结果见图 5.4 和图 5.5。

例 5.6　struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "
http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
<package name="default" extends="struts-default">
<interceptors>
        <interceptor name="myInterceptor" class="com.MyInterceptor"></interceptor>
    </interceptors>
    <action name="PublishAction" class="com.PublishAction">
        <result name="success">/success.jsp</result>
        <result name="login">/success.jsp</result>
        <interceptor-ref name="defaultStack"></interceptor-ref>
```

```
            < interceptor - ref name = "myInterceptor"></interceptor - ref >

        </action >
    </package >
    </struts >
```

图 5.4　发表评论页面

图 5.5　文字过滤效果

思考与练习

（1）简述什么是拦截器，它有什么作用。

（2）如何编写拦截器？

（3）什么是默认拦截器？