第2章 Struts2 核心组件

2.1 Struts2 动作（Action）

2.1.1 Action 的作用

Struts2 的动作（Action）组件是 Struts2 框架的核心。Action 主要有三个作用：第一，为给定的请求封装需要做的实际工作；第二，在从请求到视图的框架自动数据传输中作为中介；第三，帮助框架确定要返回的视图。

1. 动作封装工作单元

动作在框架中可作为 MVC 模式的模型。这个角色的主要职责是控制业务逻辑，动作使用 execute()方法来实现业务逻辑。这个方法中的代码应该只关注与请求相关的工作逻辑。

2. 动作为数据转移提供场所

动作是框架的模型组件，使得动作能够携带数据。由于数据保存在动作本地，在执行业务逻辑的过程中就可以很方便地访问到它们。或许一系列的 JavaBean 属性会增加动作组件的代码量，但是当 execute()方法引用这些属性中存储的数据时，邻近的数据会让动作的代码变得更简洁。

3. 动作为结果路由选择返回控制字符串

动作组件的最后一个职责是返回控制字符串以选择应该被呈现的结果页面。先前的框架将路由对象（ActionMapping）整体传递给了动作组件的入口方法。返回的控制字符串消除了对这些对象的依赖，使得方法签名更简洁，并且降低了动作与具体路由代码的耦合。返回字符串的值必须与配置在声明性架构中期望的结果的名字匹配。例如，HelloWorldAction 返回“SUCCESS”字符串，可以从 XML 配置文件中看出，SUCCESS 是某一个结果组件的名字。

2.1.2　Action 类的编写

为了让用户开发的 Action 类更加规范，Struts2 提供了一个 Action 接口，这个接口定义了 Struts2 的 Action 类应该实现的规范。下面是标准 Action 接口的代码：

```
public interface Action {
    //定义 Action 接口里包含的一些结果字符串
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";
    public static final String NONE = "none";
    public static final String SUCCESS = "success";
    //定义处理用户请求的 execute()方法
    public String execute() throws Exception;
}
```

上面的 Action 接口里只定义了一个 execute()方法，该接口规范规定了 Action 类应该包含一个 execute()方法，该方法返回一个字符串，此外，该接口还定义了 5 个字符串常量，它的作用是统一 execute()方法的返回值。例如，当 Action 类处理完用户请求后，有人喜欢返回 welcome 字符串，有人喜欢返回 success 字符串，如此不利于项目的统一管理，Struts2 的 Action 接口定义加上了如上的 5 个字符串常量：ERROR、NONE、INPUT、LOGIN、SUCCESS 等，分别代表了特定的含义。当然，如果开发者依然希望使用特定的字符串作为逻辑视图名，开发者依然可以返回自己的视图名。

另外，Struts2 还为 Action 接口提供了一个实现类：ActionSupport，下面是该实现类部分的代码，具体的实现代码读者可以参考 Struts2 的源码包。

```
public class ActionSupport implements Action, Validateable, ValidationAware,
        TextProvider, LocaleProvider, Serializable {
    // 默认处理用户请求的方法，直接返回 SUCCESS 字符串
    public String execute() throws Exception {
        return SUCCESS;
    }
    @Override
    public Locale getLocale() {
        return null;
    }
    @Override
    public String getText(String arg0) {
        return null;
    }
    @Override
    public ResourceBundle getTexts() {
        return null;
    }
    @Override
    public void addActionError(String arg0) {}
    @Override
```

```
        public void validate() { }
        //省略部分需要实现接口的方法
    }
```

ActionSupport 是一个默认的 Action 实现类，该类里已经提供了许多默认方法，这些方法包括获取国际化信息的方法、数据校验的方法、默认的处理用户请求的方法等。实际上，ActionSupport 是 Struts2 默认的 Action 处理类，如果让开发者的 Action 类继承该 ActionSupport 类，则会大大简化 Action 的开发。

2.1.3 Action 的使用与配置

现在可以开始开发动作了。在本节中，首先讲解 Action 的配置，接着讲解 Action 类的具体编写使用。Struts2 的 Action 的配置是在 struts.xml 中进行的，通过<action>元素进行配置，<action>元素常用的属性有如下几个：

- name：该属性用来指定客户端发送请求的地址映射名称；
- class：该属性用来指定业务逻辑处理的 Action 类名称；
- method：该属性用来指定进行业务逻辑处理的 Action 类中的方法名称。

一个典型的 action 配置如下：

```
<action name="sayHello" class="cap.action.HelloWorldAction" method="sayHello">
   <result name="success">/output.jsp</result>
</action>
```

在上述 Action 代码配置中，为 action 指定 name，class 和 method 属性。method 属性值就是 Action 类中定义的方法名，在默认情况下是 execute()方法。

通常要为 Action 指定一个或多个视图，这些视图的名称或类型通过 result 元素来配置，该元素主要有 type 和 name 属性。result 元素的 name 属性用来指定 Action 中方法返回的名称，例如上面的代码片段返回 SUCCESS。

1. 动态 Action 调用

实际应用中，每个 Action 都要处理很多业务，所以每个 Action 都要包含多个处理业务逻辑的方法。针对不同的客户端请求，Action 都会调用不同的方法来处理。Struts2 提供使用 method 属性实现方法的动态调用。实现的步骤如下：

首先在表单中的 action 中指定 action 的名称，例如下面的代码：

```
<form action="login" method="post">
```

其次在 struts.xml 中配置 action 名称：

```
<action name="login" class="cap.action.LoginAction" method="login">
    <result name="success">/output.jsp</result>
</action>
```

下面将通过一个具体的例子讲解动态 Action 调用的方法。

（1）继续在 struts1 工程中，在 HelloWorldAction 中添加 SayHello 的方法，在 struts2 中定义方法的原型如下：

```
public String someMethod(){}
```

（2）在 src 下的 cap.action 子包中的 HelloWorldAction.java 文件中添加 SayHello 方法，实现代码如下：

```
public String sayHello(){
        username="欢迎您：" +username;
        return SUCCESS;
    }
```

（3）在 src 中的 sturts.xml 文件中写入以下的代码：

```
<action name="sayHello" class="cap.action.HelloWorldAction" method="sayHello">
    <result name="success">/output.jsp</result>
</action>
```

（4）在 WebContent 下复制 sayHi.jsp，重命名为 sayHello.jsp。将 form 标签中 action 属性的值修改为 sayHello。

（5）运行工程：在浏览器中输入 http://localhost:8080/struts1/sayHello.action 地址来访问 HelloWorld Action。如图 2-1 所示，输入“starlee2008@163.com”文本，单击“提交”按钮。返回结果页面如图 2-2 所示。

图 2-1　sayHello 运行结果 1

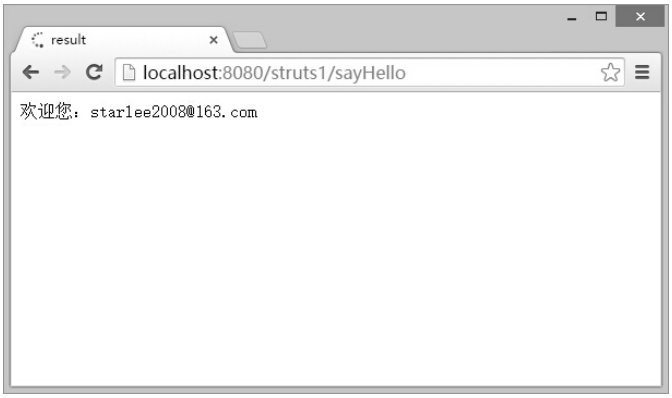

图 2-2　sayHello 运行结果 2

通过上面的例子，可以发现 src/cap.action 子包中 HelloWorldAction.java 类的 Action 方法

（execute 和 sayHello）返回的都是 SUCCESS。这个属性变量在 ActionSupport 类或其父类中定义。

2. 向对象传递数据

使用 Struts2，用对象传递数据将变得非常方便，和普通的 POJO（Plain Ordinary Java Object，简单的 Java 对象，实际就是普通 JavaBeans，是为了避免和 EJB 混淆所创造的简称词）一样在 Action 编写 Getter 和 Setter，然后在 JSP 的 UI 标志的 name 与其对应，在提交表单到 Action 时，就可以获得其值。

下面我们看一个具体向对象传递数据的例子。

（1）按照第 1 章的内容在 Eclipse 中新建 Dynamic Web Project 工程 struts2。首先在 WebContent 下新建 login.jsp 页面，在 body 标签中添加下面的代码：

```
<form action="login" method="post">
   <input type="text" name="admin.username"><br>
   <input type="password" name="admin.password"><br>
   <input type="submit" value="登录">
</form>
```

（2）在 src 的 cap.bean 包中新建 Admin.java 类，编辑后的代码如下：

```
package cap.bean;
public class Admin {
        private int id;
        private String username;
        private String password;
        //省略 getters 和 setters
}
```

（3）在 src 的 cap.action 包中新建 LoginAction 类，编辑后的代码如下：

```
package cap.action;
import cap.bean.Admin;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
        private Admin admin;
        public Admin getAdmin() {
                return admin;
        }
        public void setAdmin(Admin admin) {
                this.admin = admin;
        }
        public String login(){
                if(admin.getUsername().equals("starlee2008")&&admin.getPassword().equals("starlee2008"))
                {
                return SUCCESS;
                }else{
                        return ERROR;
                }
        }
}
```

代码解释：这里没有采用数据库验证登录的方式，采用的是静态判断，只要输入的用户名和密码都为“starlee2008”时，返回登录成功的 SUCCESS 字符串，反之返回 ERROR 的字符串。

（4）在 src 的 struts 配置文件 struts.xml 中添加下面的代码：

```
<action name="login" class="cap.action.LoginAction" method="login">
        <result name="success">/result.jsp</result>
        <result name="error">/error.jsp</result>
</action>
```

（5）还需添加登录判断后的两种跳转页面，分别是 result.jsp 和 error.jsp，可查看随书提供的示例源码。运行 Tomcat，在浏览器地址栏中键入 http://localhost:8080struts2/login.jsp，出现如图 2-3 所示页面。在 User 和 name 中分别输入“cap”和“cap”，单击“登录”按钮，出现如图 2-4 所示页面。如果输入了错误的用户名和密码，会提示登录失败。

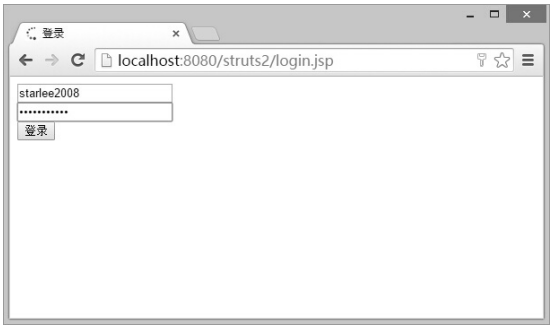

图 2-3　login.jsp 运行

图 2-4　登录结果

3. 模型驱动

所谓模型驱动，就是使用单独的 JavaBean 实例来贯穿整个 MVC 流程。简单地说，模型驱动就是使用单独的 VO(Value Object，值对象)来封装请求参数和处理结果。

使用模型驱动模式时，Action 必须实现 ModelDriven 接口，实现该接口则必须实现 getModel 方法，该方法用于把 Action 和与之对应的 Model 实例关联起来。

下面继续通过一个案例讲解模型驱动的使用。

（1）继续在工程 struts2 中操作。在 src 的 cap.action 包中新建 LoginxAction 类，编辑后的代码如下：

```
package cap.action;
import cap.bean.Admin;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
@SuppressWarnings("serial")
public class LoginxAction extends ActionSupport implements ModelDriven<Admin>{
	private Admin admin;
	public Admin getAdmin() {
		return admin;
	}
	public void setAdmin(Admin admin) {
		this.admin = admin;
	}
	public String loginx(){
		if(admin.getUsername().equals("cap")&&admin.getPassword().equals("cap"))
		return SUCCESS;
		else
		return ERROR;
	}
	@Override
	public Admin getModel() {
		if(admin==null){
		admin=new Admin();
		}
		return admin;
	}
}
```

代码解释：loginx 方法和前一节的方法功能相同，实现静态的用户名和密码判断，getModel 方法主要是创建一个 Admin 的对象 admin，并根据页面 loginx.jsp 传递过来的 username 和 password 值，通过调用对象 admin 的 setUsername 方法和 setPassword 方法设置值。

（2）在 WebContent 中新建 loginx.jsp，在 body 标签中添加如下的代码：

```
<form action="loginx" method="post">
	<input type="text" name="username"><br>
	<input type="password" name="password"><br>
	<input type="submit" value="登录">
</form>
```

代码解释：和 login.jsp 页面的主要区别是本页面采用传递的是属性值，而不是采用对象名.属性值（admin.username）的方法传递值。实现 ModelDriven 接口后，会通过 getModel 创建 Admin 对象 admin，并初始化其值。

（3）在 src 下的 struts.xml 文件中添加下面的代码：

```
<action name="loginx" class="cap.action.LoginxAction" method="loginx">
    <result name="success">/result.jsp</result>
    <result name="error">/error.jsp</result>
</action>
```

2.2 Struts2 拦截器（Interceptor）

2.2.1 拦截器

拦截器实质就是 AOP(面向切面编程)的编程思想。拦截器允许在 Action 处理之前，或者 Action 处理结束之后，插入开发者自定义的代码。

拦截器体系是 Struts2 的一个重要组成部分，对于 Struts2 框架而言，可以将其理解为一个空容器，正是大量的内建拦截器完成了该框架的大部分操作。

对于 Struts2 的拦截器体系而言，当需要使用某个拦截器时，只需要在配置文件中应用该拦截器即可，如果不需要使用该拦截器，也只需要在配置文件中取消该拦截器，不管是否应用某个拦截器，对于整个 Struts2 框架不会有任何影响。这种设计方式非常灵活，可扩展性好。

因为 Struts2 框架的拦截器是动态配置的，所以开发者可以非常方便地扩展 Struts2 框架，只要实现 Interceptor 接口或者继承 AbstractInterceptor 类，并将其配置在 struts.xml 文件中即可。实际上，在 Struts2 中开发自定义拦截器非常便利，因此，开发者可非常方便地将多个 Action 中需要重复执行的动作放在拦截器中定义，从而提供更好的代码复用。

虽然拦截器很重要，但是一般不会去编写很多拦截器。实际上，Web 应用程序领域常见任务已经编写和捆绑进了 struts-default 包。

1. 拦截器的工作原理

现在来看看拦截器是如何工作的。拦截器 ActionInvocation 类负责指挥着动作的完整执行，以及与之相关的拦截器栈的顺序触发。

ActionInvocation 类封装了与特定动作执行相关的所有处理细节。当框架收到一个请求时，它首先必须决定这个 URL 映射到哪个动作。这个动作的一个实例会被加入到一个新创建的 ActionInvocation 实例中。接着，框架查询声明性架构(通过应用程序的 XML 或者 Java 注解创建)，以发现哪些拦截器应该触发，以及按照什么样的顺序触发。指向这些拦截器的引用被加入到 ActionInvocation 类中。除了这些核心元素，ActionInvocation 还具有其他重要信息。

ActionInvocation 类创建好并且填充了需要的所有对象和信息，就可以开始调用。ActionInvocation 类公开了 invoke()方法，框架通过调用这个方法开始动作的执行。当框架调用这个方法时，ActionInvocation 通过执行拦截器栈中的第一个拦截器开始这个调用过程，但需注意，invoke()方法并不总是映射到第一个拦截器。

如图 2-5 所示，第一个被触发的拦截器是 exception 拦截器。从这里开始，每一个拦截器按照与从上到下相同的顺序触发。因此，最后一个被触发的拦截器会是 workflow 拦截器。在结果被呈现之后，拦截器会按照相反的顺序再触发一遍，使它们有机会做后续处理。

后续拦截器继续执行，最终执行动作，这些都通过递归调用 ActionInvocation 的 invoke()方法实现。每次 invoke()方法被调用时，ActionInvocation 都会查询自身的状态，调用接下来的任何一个拦截器。在所有的拦截器都被调用之后，invoke ()方法会促使动作类执行。

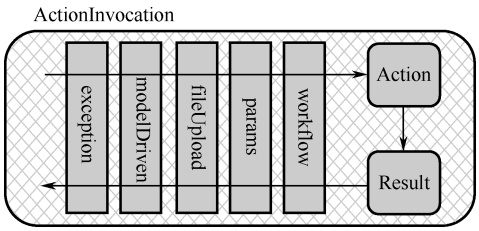

图 2-5　ActionInvocation 封装了动作及与之关联的拦截器和结果的执行

拦截器的执行周期，主要分为三个阶段：

第一阶段，做一些预处理。

第二阶段，通过调用 invoke()方法将控制转移给后续的拦截器，最后直到动作，或者通过返回一个控制字符串中断执行。

第三阶段，做一些后加工。

通常，使用拦截器可以解决如权限控制、跟踪日志、跟踪系统的性能瓶颈等问题。

2. 拦截器的配置

Struts2 框架自带很多内置栈，使内置拦截器方便排列。可以通过扩展 struts-default.xml 文件中定义的 struts-default 包来继承包含 defaultStack 在内的所有内建的栈。拦截器的配置代码片段如下：

```
<action name="timer" class="cap.action.TimerAction">
        <interceptor-ref name="timer"></interceptor-ref>
        <interceptor-ref name="logger"></interceptor-ref>
        <result name="success">/result.jsp</result>
</action>
```

上述代码实现将 timer 和 logger 拦截器与动作（timer）关联起来，它们会按被列出的顺序触发。Struts2 已经提供丰富多样的，功能齐全的拦截器实现。在 struts2-core-2.x.x.jar 包的 struts-default.xml 文件中可以查看关于默认的拦截器与拦截器链的配置。

下面列出了在 struts-default.xml 文件定义的部分拦截器：

```
<interceptors>
        <interceptor name="alias" class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
        <interceptor name="autowiring" class="com.opensymphony.xwork2.spring.interceptor.
ActionAutowiringInterceptor"/>
        <interceptor name="chain" class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
        <interceptor name="conversionError" class="org.apache.struts2.interceptor.
StrutsConversionErrorInterceptor"/>
        <interceptor name="cookie" class="org.apache.struts2.interceptor.CookieInterceptor"/>
        <interceptor name="cookieProvider" class="org.apache.struts2.interceptor.
CookieProviderInterceptor"/>
        <interceptor name="clearSession" class="org.apache.struts2.interceptor.
ClearSessionInterceptor" />
        <interceptor name="createSession" class="org.apache.struts2.interceptor.
CreateSessionInterceptor" />
        <interceptor name="debugging" class="org.apache.struts2.interceptor.debugging.
DebuggingInterceptor" />
```

```
                <interceptor name="execAndWait" class="org.apache.struts2.interceptor.
ExecuteAndWaitInterceptor"/>
                <interceptor name="exception" class="com.opensymphony.xwork2.interceptor.
ExceptionMappingInterceptor"/>
                <interceptor name="fileUpload" class="org.apache.struts2.interceptor.FileUploadInterceptor"/>
                <interceptor name="i18n" class="com.opensymphony.xwork2.interceptor.I18nInterceptor"/>
                <interceptor name="logger" class="com.opensymphony.xwork2.interceptor.LoggingInterceptor"/>
                <interceptor name="modelDriven" class="com.opensymphony.xwork2.interceptor.
ModelDrivenInterceptor"/>
                <interceptor name="scopedModelDriven" class="com.opensymphony.xwork2.interceptor.
ScopedModelDrivenInterceptor"/>
                <interceptor name="params" class="com.opensymphony.xwork2.interceptor.
ParametersInterceptor"/>
                <interceptor name="actionMappingParams" class="org.apache.struts2.interceptor.
ActionMappingParametersInteceptor"/>
                <interceptor name="prepare" class="com.opensymphony.xwork2.interceptor.
PrepareInterceptor"/>
                <interceptor name="staticParams" class="com.opensymphony.xwork2.interceptor.
StaticParametersInterceptor"/>
                <interceptor name="scope" class="org.apache.struts2.interceptor.ScopeInterceptor"/>
                <interceptor name="servletConfig" class="org.apache.struts2.interceptor.
ServletConfigInterceptor"/>
                <interceptor name="timer" class="com.opensymphony.xwork2.interceptor.TimerInterceptor"/>
                <interceptor name="token" class="org.apache.struts2.interceptor.TokenInterceptor"/>
                <interceptor name="tokenSession" class="org.apache.struts2.interceptor.
TokenSessionStoreInterceptor"/>
                <interceptor name="validation" class="org.apache.struts2.interceptor.validation.
AnnotationValidationInterceptor"/>
                <interceptor name="workflow" class="com.opensymphony.xwork2.interceptor.
DefaultWorkflowInterceptor"/>
                <interceptor name="store" class="org.apache.struts2.interceptor.MessageStoreInterceptor" />
                <interceptor name="checkbox" class="org.apache.struts2.interceptor.CheckboxInterceptor" />
                <interceptor name="profiling" class="org.apache.struts2.interceptor.
ProfilingActivationInterceptor" />
                <interceptor name="roles" class="org.apache.struts2.interceptor.RolesInterceptor" />
                <interceptor name="annotationWorkflow" class="com.opensymphony.xwork2.interceptor.
annotations.AnnotationWorkflowInterceptor" />
                <interceptor name="multiselect" class="org.apache.struts2.interceptor.MultiselectInterceptor" />
```

如果您想要使用 struts-default.xml 文件中定义的上述拦截器，还需要在应用程序 struts.xml 文件中进行一些设置，后续将用具体案例来讲解。

3. 使用拦截器栈

Struts2 允许将多个拦截器组合在一起，形成一个拦截器栈。对于 Struts2 系统而言，多个拦截器组成的拦截器栈对外也表现为一个拦截器。例如，在 Action 执行前同时做登录检查、安全检查和记录日志，则可以把这三个动作对应的拦截器组成一个拦截器栈。定义拦截器栈使用<intercepter-stack .../>元素，并且需要在其中使用<intercepter-ref…/>元素来定义多个拦截器的引用，即该拦截器栈由多个<intercepter-ref .../>元素指定的拦截器组成。下面是典型的拦截器栈的

定义代码片段。

```
<interceptors>
        <!-- 添加登录拦截器 -->
        <interceptor name="checkLogin" class="cap.util.CheckLoginInterceptor"/>
        <!-- 新建一个栈，把登录拦截器和默认的栈放进去 -->
        <interceptor-stack name="mystack">
            <interceptor-ref name="defaultStack"/>
            <interceptor-ref name="checkLogin"/>
        </interceptor-stack>
</interceptors>
```

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，拦截器（包含拦截器栈）的拦截行为将会在 Action 的 execute 方法执行之前被执行。通过<interceptor-ref.../>元素可以在 Action 内使用拦截器。

2.2.2 拦截器的使用

1. 预定义拦截器案例

下面将使用实例讲解拦截器 timer 的使用。

（1）在 Eclipse 中新建工程 struts3，在 src 的 cap.action 包中新建 TimerAction 类，编辑后的代码如下：

```
package cap.action;
import com.opensymphony.xwork2.ActionSupport;
@SuppressWarnings("serial")
public class TimerAction extends ActionSupport{
    public String timer()
    {
        try {
            Thread.sleep(500);//代码①
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return SUCCESS;
    }
}
```

代码解释：代码①调用线程 Thread 类的静态方法实现睡眠 5 秒钟。

（2）在 src 的 struts.xml 文件中 package 标签处添加如下的代码：

```
<action name="timer" class="cap.action.TimerAction">
        <interceptor-ref name="timer"></interceptor-ref>
        <result name="success">/result.jsp</result>
</action>
```

（3）发布运行应用程序，在浏览器的地址栏键入 http://localhost:8080/struts3/timer，在出现 result.jsp 页面后，查看服务器的后台输出。

```
十一月 12, 2013 3:49:30 下午 com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
信息: Executed action [//timer!timer] took 508 ms.
```

在自己电脑上执行 Timer 拦截器所花的时间，可能与上述时间有些不同，其缘由主要是计算机性能差异所造成的。当第一次加载 Timer 时，需要进行一定的初始工作。当你再次运行，会发现时间接近 500ms。

2. 自定义拦截器案例

对于框架（Framework），可扩展性是必不可少的特性。虽然，Struts2 提供丰富多样的拦截器实现，但我们依然可以创建自定义拦截器，并且也很容易实现。

Struts2 的拦截器直接或间接实现接口 com.opensymphony.xwork2.interceptor.Interceptor，另外，我们也习惯性继承 com.opensymphony.xwork2.interceptor.AbstractInterceptor 类。AbstractInterceptor 是 Interceptor 接口的实现类。

从 com.opensymphony.xwork2.interceptor.Interceptor 接口定义来看，Interceptor 接口只定义了 3 个方法。前 2 个方法是典型的生命周期方法，作用是初始化或者清理资源。真正的业务逻辑发生在 intercept()方法中。可以看到，这个方法被递归的 ActionInvocation.invoke()方法调用。

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

Intercept（ActionInvocation() invocation）：该方法是用户需要实现的拦截动作。就像 Action 的 execute 方法一样，intercept()方法会返回一个字符串作为逻辑视图。如果该方法直接返回了一个字符串，系统将会跳转到该逻辑视图对应的实际视图资源，不会调用被拦截的 Action。该方法的 ActionInvocation 参数包含了被拦截的 Action 的引用，可以通过调用该参数的 invoke 方法，将控制权转给下一个拦截器，或者转给 Action 的 execute 方法。

下面的例子将展示通过继承 AbstractInterceptor，实现授权拦截器。

3. 自定义拦截器案例

（1）继续在工程 struts3 中编辑，在 src 的 cap.util 包下创建登录拦截器 CheckLoginInterceptor 类，编辑后的代码如下：

```
package cap.util;
import java.util.Map;
import cap.action.LoginAction;
import cap.bean.Admin;
import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
@SuppressWarnings("serial")
public class CheckLoginInterceptor extends AbstractInterceptor{
    @Override
    public String intercept(ActionInvocation ai) throws Exception {
        System.out.println("开始拦截器拦截");
        Object obj=ai.getAction();
        if(obj instanceof LoginAction){
```

```
                System.out.println("登录 action 不需要拦截");
                return ai.invoke();
            }
            Map<String,Object> session=ai.getInvocationContext().getSession();
            Admin admin=(Admin)session.get("admin");
            if(admin!=null){
                System.out.println("已经登录。不需要拦截");
                return ai.invoke();
            }else{
                System.out.println("你还没有登录。跳转到登录页面");
                return Action.LOGIN;
            }
        }
    }
```

代码解释：在 intercept()方法中，首先通过 ActionInvocation 的 getAction 获得对象 object，接着判断对象 object 是否为 LoginAction 的实例，如果是，不需要拦截，并返回。如果不是 LoginAction 对象的实例，首先获得 session 对象，通过检查 session 是否存在键为“admin”的对象 admin，判断用户是否登录。如果用户已经登录，将角色放到 Action 中，调用 Action；否则，拦截直接返回 Action.LOGIN 字符串并调转到相应的视图页面。

（2）在 src 的 cap.action 中创建 LoginAction 类，编辑后的代码如下：

```
package cap.action;
import java.util.Map;
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionSupport;
import cap.bean.Admin;
public class LoginAction extends ActionSupport implements SessionAware{//代码①
    private Admin admin;
    private Map<String,Object> session;
    public Admin getAdmin() {
        return admin;
    }
    public void setAdmin(Admin admin) {
        this.admin = admin;
    }
    public String login(){
        if(admin.getUsername().equals("cap")&&admin.getPassword().equals("cap")){
            session.put("admin", admin);
            return SUCCESS;
        }else
            return ERROR;
    }

    @Override
    public void setSession(Map<String, Object> session) {
        this.session=session;
```

```
        }
    }
```

代码解释：要在 Action 类中使用 Map<String,Object> session 对象，需要实现 SessionAware 接口，login()方法首先判断用户名和密码是否匹配，如果匹配的话使用 session 将 Admin 的对象 admin 存放在 Map 中名为 admin。如果不匹配，返回 ERROR 常量。

（3）在 src 的 cap.bean 中创建 Admin 类，编辑后的代码如下：

```
package cap.bean;
public class Admin {
    private Integer id;
    private String username;
    private String password;
   //省略 getters 和 setters
}
```

（4）修改 src 下的 struts2 的配置文件 struts.xml，在 struts.xml 中包含 test.xml，代码如下：

```
<?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>
<package name="login" extends="struts-default">
        <!-- 添加拦截器 -->
        <interceptors>
            <!-- 添加登录拦截器 -->
            <interceptor name="checkLogin" class="cap.util.CheckLoginInterceptor"/>
            <!-- 新建一个栈，把登录拦截器和默认的栈添加进去 -->
            <interceptor-stack name="mystack">
                <interceptor-ref name="defaultStack"/>
                <interceptor-ref name="checkLogin"/>
            </interceptor-stack>
        </interceptors>
        <!-- 修改默认拦截器 -->
        <default-interceptor-ref name="mystack"/>
        <!-- 将 result 设置为全局的，这样就不用在每一个 package 里面都添加拦截器了 -->
        <global-results>
            <result name="login">/login.jsp</result>
        </global-results>
    </package>
    <include file="test.xml"></include>
</struts>
```

代码解释：<package>之间的代码定义了一个包，其名称为 login，namespace 默认值为 “/”，并且继承了 Struts2 的 struts-default 包。接着定义了拦截器栈，并将拦截器栈的名称命名为 mystack，其中包含了自定义的拦截器 checkLogin，<default-interceptor-ref>标签把 mystack 拦截器栈设置为默认的拦截器栈，<global-results>标签定义了一个全局的返回值，如果返回的字符

串为 LOGIN，那么将返回到 loing.jsp 视图。<include>标签表示包含一个名为 test.xml 的配置文件。

（5）在 src 下创建 test.xml，编辑后的代码如下：

```
<?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>
    <package name="default" namespace="/" extends="login">
        <action name="timer" class="cap.action.TimerAction" method="timer">
            <result name="success">/result.jsp</result>
        </action>
        <action name="login" class="cap.action.LoginAction" method="login">
            <result name="success">/index.jsp</result>
            <result name="error">/error.jsp</result>
        </action>
    </package>
</struts>
```

代码解释: 这里重新定义了一个 Struts2 的配置文件，包的名称为 default，并且继承了 login 包，这样 login 包里的全局拦截器就会在此包中起到拦截的作用。

（6）在 WebContent 下创建 login.jsp 页面， 在 body 标签中添加下面的代码：

```
<form action="login" method="post">
    <input type="text" name="admin.username"><br>
    <input type="password" name="admin.password"><br>
    <input type="submit" value="登录">
</form>
```

（7）在 WebContent 下创建 index.jsp 页面，在 body 标签中添加下面的代码（效果见图 2-6）：

```
<p>全局拦截器，只有登录后才能使得 timer 生效</p>
<a href="timer">使用内置的 timer 拦截器，会在控制台下打印出 action 执行的时间</a>
```

（8）发布运行应用程序，在浏览器地址栏中输入：http://localhost:8080/struts3/timer 。单击页面中的超链接，由于没有登录，所以会跳转到 login.jsp 页面，如图 2-7 所示。

图 2-6　index.jsp 运行

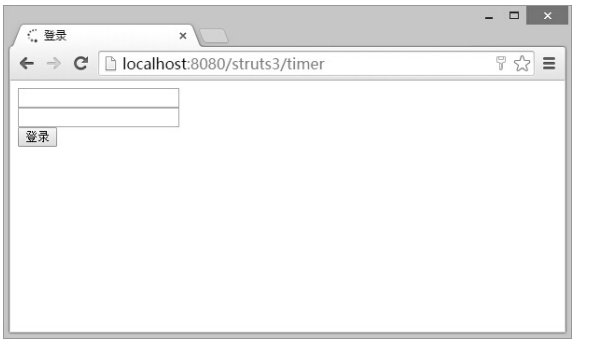

图 2-7　未登录被拦截的结果

2.3　Struts2 注解（Annotation）

注解（Annotation）是 Java 语言比较新的一个特性，它允许在 Java 源代码文件中直接添加元数据。

通常情况下，Struts2 是通过 struts.xml 文件配置的。但是随着系统规模的扩大，所需要配置的文件就会比较多，虽然可以根据不同的系统功能将不同模块的配置文件单独书写，然后通过<include>节点将不同的配置文件引入最终的 struts.xml 文件中，但是毕竟还是要维护和管理这些文件，使得维护工作变得艰巨。

Struts2 声明性架构机制可以配置用来扫描 Java 类以获得与 Struts2 相关的注解。

如果要使用 struts2 的注解功能，必须使用一个包 struts2-convention-plugin-2.x.x.x.jar，其中 xxx 代表版本号。

2.3.1　常用注解

Struts2 中注解的主要概念包括 package、action 以及 Interceptor 等，下面将分别讲解这些概念：

（1）@ParentPackage：此注解对应 xml 文件中的 package 节点，它只有一个属性叫 value，其实就是 package 的 name 属性；

（2）@Namespace：命名空间，也就是 xml 文件中<package>的 namespace 属性；

（3）@Action：此注解对应 <action> 节点，可以应用于 action 类上，也可以应用于方法上。此注解中的常用属性见表 2-1。

表 2-1　@Action 注解的常用属性

属　性	含　义
value	表示 action 的 URL，也就是<action>节点中的 name 属性
results	表示 action 的多个 result，这个属性是一个数组属性，因此可以定义多个 result
interceptorRefs	表示 action 的多个拦截器。这个属性也是一个数组属性，因此可以定义多个拦截器
params	这是一个 String 类型的数组，它按照 name/value 的形式组织，是传给 action 的参数
exceptionMappings	这是异常属性，它是一个 ExceptionMapping 的数组属性，表示 action 的异常，在使用时必须引用相应的拦截器

（4）@Result：这个注解对应 <result> 节点，只能应用于 action 类上。这个注解中常用属性见表 2-2。

表 2-2　@Result 注解的常用属性

属　性	含　义
name	表示 action 方法的返回值，也就是<result>节点的 name 属性，默认情况下是 success
location	表示 view 层文件的位置，可以是相对路径，也可以是绝对路径
type	是 action 的类型，如 redirect，forward
params	是一个 String 数组，也是以 name/value 形式传送给 result 的参数

2.3.2　注解的使用

下面将通过一个具体的案例来讲解 Struts2 中注解（Annotation）的使用。

（1）在 Eclipse 中，复制 struts3 工程，将工程重新命名为 struts4，如图 2-8 所示。

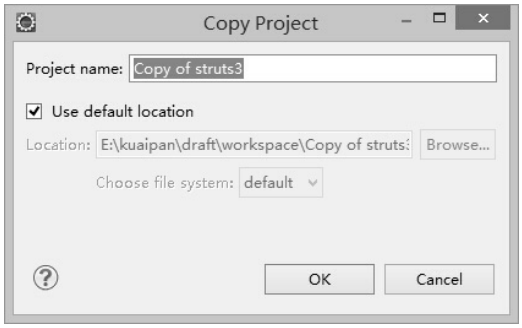

图 2-8　工程复制

（2）将 src 目录下的 test.xml 配置文件删除，然后将 struts.xml 文件中的下面代码删除。

```
<include file="test.xml"></include>
```

（3）编辑 cap.action 下的 LoginActon 类，编辑后的代码如下：

```
package cap.action;
import java.util.Map;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.InterceptorRef;
import org.apache.struts2.convention.annotation.InterceptorRefs;
import org.apache.struts2.convention.annotation.ParentPackage;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionSupport;
import cap.bean.Admin;
@SuppressWarnings("serial")
@ParentPackage("login")
@InterceptorRefs(@InterceptorRef("mystack"))
@Results( { @Result(name = "success", location = "/index.jsp"),
```

```
        @Result(name = "input", location = "/login.jsp") })
public class LoginAction extends ActionSupport implements SessionAware{
    private Admin admin;
    private Map<String,Object> session;
    public Admin getAdmin() {
        return admin;
    }
    public void setAdmin(Admin admin) {
        this.admin = admin;
    }
    @Action(value="login",results={@Result(name="success",location="/index.jsp"),
            @Result(name="error",location="/error.jsp"),
            @Result(name="input",location="/login.jsp")})
    public String login(){
        if(admin.getUsername().equals("cap")&&admin.getPassword().equals("cap")){
            session.put("admin", admin);
            return SUCCESS;
        }else
            return ERROR;
    }
    @Override
    public void setSession(Map<String, Object> session) {
        this.session=session;
    }
}
```

代码解释：注释的具体使用详见表 2-1 和表 2-2。其余的 Java 实现代码和上一节功能相同。

（4）编辑 cap.action 下的 TimerActon 类，编辑后的代码如下：

```
package cap.action;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.InterceptorRef;
import org.apache.struts2.convention.annotation.InterceptorRefs;
import org.apache.struts2.convention.annotation.ParentPackage;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;
import com.opensymphony.xwork2.ActionSupport;
@SuppressWarnings("serial")
@ParentPackage("login")
@InterceptorRefs(@InterceptorRef("mystack"))
@Results( { @Result(name = "success", location = "/index.jsp"),
        @Result(name = "input", location = "/login.jsp") })
public class TimerAction extends ActionSupport{
    @Action(value="timer",results={@Result(name="success",location="/result.jsp")})
    public String timer()
    {
        try {
            Thread.sleep(500);
```