第3章 Struts2的拦截器

学习目标

- 理解拦截器的工作原理
- 掌握拦截器的配置和使用
- 学会使用自定义拦截器

拦截器是Struts2框架的重要组成部分，Struts2的很多功能都是构建在拦截器之上的，如数据校验、转换器、国际化等。Struts2利用其内建的拦截器可以完成大部分的操作，当内置拦截器不能满足时，开发者也可以自己扩展。可以说，Struts2框架之所以简单易用，与拦截器的作用是分不开的。本章将详细介绍Struts2拦截器的相关知识。

3.1 拦截器简介

3.1.1 拦截器概述

拦截器(Interceptor)是Struts2的核心组成部分，它可以动态拦截Action调用的对象，类似于Servlet中的过滤器。Struts2的拦截器是AOP(Aspect-Object-Programming，面向切面编程)的一种实现策略，是可插拔的，需要某一个功能时就“插入”这个功能的拦截器，不需要这个功能时就“拔出”这一拦截器。它可以任意地组合Action提供的附加功能，而不需要修改Action的代码，开发者只需要提供拦截器的实现类，并将其配置在struts.xml中即可。

Struts2中将各个功能对应的拦截器分开定义，每个拦截器完成单个功能，如果要运用某个功能就加入对应的拦截器。如果将这些拦截器组合在一起就形成了拦截器链(Interceptor Chain)或拦截器栈(Interceptor Stack)。所谓的拦截器链是指对应各个功能的拦截器按照一定的顺序排列在一起形成的链，而拦截器链组成的集合就是拦截器栈。当有适配连接器栈的访问请求进来时，这些拦截器就会按照之前定义的顺序被调用。

3.1.2 拦截器的工作原理

通常情况下，拦截器都是以代理方式调用的，它在一个Action执行前后进行拦截，围绕着Action和Result的执行而执行，其工作方式如图3-1所示。

从图3-1可以看出，Struts2拦截器的实现原理与Servlet过滤器的实现原理类似，它以链式执行，对真正要执行的方法(execute())进行拦截。首先执行Action配置的拦截器，在

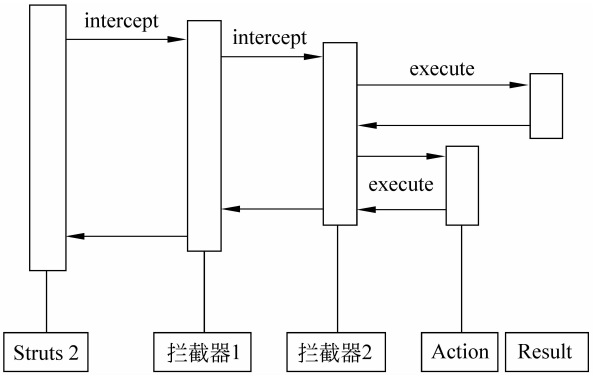

图 3-1 Struts2 拦截器的工作方式

Action 和 Result 执行之后，拦截器会再次执行（与先前调用顺序相反），在此链式执行的过程中，每一个拦截器都可以直接返回，从而终止余下的拦截器、Action 及 Result 的执行。

3.1.3 拦截器的配置

了解了拦截器的工作原理后，本节将介绍如何配置并使用拦截器。

1. 拦截器

要想让拦截器起作用，首先要对它进行配置。拦截器的配置是在 struts.xml 文件中完成的，它通常以＜interceptor＞标签开头，以＜/interceptor＞标签结束。定义拦截器的语法格式如下：

```
<interceptor name="interceptorName" class="interceptorClass">
    <param name="paramName">paramValue</param>
</interceptor>
```

上述语法格式中，name 属性用来指定拦截器的名称，class 属性用于指定拦截器的实现类。有时，在定义拦截器时需要传入参数，这时需要使用＜param＞标签，其中 name 属性用来指定参数的名称，paramValue 表示参数的值。

2. 拦截器栈

在实际开发中，经常需要在 Action 执行前同时执行多个拦截动作，如用户登录检查、登录日志记录以及权限检查等，这时，可以把多个拦截器组成一个拦截器栈。在使用时，可以将栈内的多个拦截器当成一个整体来引用。当拦截器栈被附加到一个 Action 上时，在执行 Action 之前必须先执行拦截器栈中的每一个拦截器。

定义拦截器栈使用＜interceptors＞元素和＜interceptor-stack＞子元素，当配置多个拦截器时，需要使用＜interceptor-ref ＞元素来指定多个拦截器，配置语法如下：

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

```
        …
    </interceptor-stack>
</interceptors>
```

在上述语法中，interceptorStackName 值表示配置的拦截器栈的名称，interceptorName 值表示拦截器的名称。除此之外，在一个拦截器栈中还可以包含另一个拦截器栈，示例代码如下：

```
<package name="default" namespace="/" extends="struts-default">
    <!--声明拦截器 -->
    <interceptors>
        <interceptor name="interceptor1" class="interceptorClass"/>
        <interceptor name="interceptor2" class="interceptorClass"/>
        <!--定义一个拦截器栈 myStack,该拦截器栈中包含两个拦截器和一个拦截器栈 -->
        <interceptor-stack name="myStack">
        <interceptor-ref name="defaultStack" />
        <interceptor-ref name="interceptor1" />
        <interceptor-ref name="interceptor2" />
        </interceptor-stack>
    </interceptors>
</package>
```

在上述代码中，定义的拦截器栈是 myStack，在 myStack 栈中，除了引用了两个自定义的拦截器 interceptor1 和 interceptor2 外，还引用了一个内置拦截器栈 defaultStack，这个拦截器是必须要引入的。

3. 默认拦截器

如果想对一个包下的 Action 使用相同的拦截器，则需要为该包中的每个 Action 都重复指定同一个拦截器，这样写显然过于烦琐。这时，可以使用默认拦截器，默认拦截器可以对其指定的包中所有的 Action 都能起到拦截的作用。一旦为某一个包指定了默认拦截器，并且该包中的 Action 未显式地指定拦截器，则会使用默认拦截器。反之，若此包中的 Action 显式地指定了某个拦截器，则该默认拦截器将会被屏蔽。此时，如果还想使用默认拦截器，则需要用户手动配置该默认拦截器的引用。

配置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。其语法格式如下：

```
<default-interceptor-ref name="拦截器(栈)的名称"/>
```

上述语法格式中，name 属性的值必须是已存在的拦截器或拦截器栈的名称。下面用该语法格式配置一个默认拦截器，示例代码如下：

```
<package name="default" namespace="/" extends="struts-default">
    <!--声明拦截器 -->
    <interceptors>
        <interceptor name="interceptor1" class="interceptorClass"/>
        <interceptor name="interceptor2" class="interceptorClass"/>
```

```
        <!--定义一个拦截器栈 myStack,该拦截器栈包含两个拦截器和一个拦截器栈 -->
        <interceptor-stack name="myStack">
            <interceptor-ref name="interceptor1" />
            <interceptor-ref name="interceptor2" />
            <interceptor-ref name="defaultStack" />
        </interceptor-stack>
    </interceptors>
    <!--配置包下的默认拦截器,既可以是拦截器,也可以是拦截器栈 -->
    <default-interceptor-ref name="myStack"/>
    <action name="login" class="cn.itcast.action.LoginAction">
        <result name="input">/login.jsp</result>
    </action>
</package>
```

在上面代码中,指定了包下面的默认拦截器为一个拦截器栈,该拦截器栈将会作用于包下所有的 Action。

注意:每一个包下只能定义一个默认拦截器,如果需要多个拦截器作为默认拦截器,则可以将这些拦截器定义为一个拦截器栈,再将这个拦截器栈作为默认拦截器即可。

3.2 Struts2 的内建拦截器

Struts2 中内置了许多拦截器,这些拦截器以 name-class 对的形式配置在 struts-default.xml 文件中,name 是拦截器的名称,也就是引用的名字;class 指定了该拦截器所对应的实现,只要自定义的包继承了 Struts2 的 struts-default 包,就可以使用默认包中定义的内建拦截器,否则需要自己定义拦截器。

3.2.1 内建拦截器的介绍

在 struts-default.xml 中,每一个拦截器都具有不同的意义,如表 3-1 所示。

表 3-1 内建拦截器的名称及说明

名 字	说 明
alias	在不同请求之间将请求参数在不同名字间转换,请求内容不变
autowiring	用来实现 Action 的自动装配
chain	让前一个 Action 的属性可以被后一个 Action 访问,现在和 chain 类型的 result ()结合使用
conversionError	将错误从 ActionContext 中添加到 Action 的属性字段中
cookies	使用配置的 name、value 来指定 cookies
cookieProvider	该类是一个 Cookie 工具,方便开发者向客户端写 Cookie
clearSession	用来清除一个 httpSession 实例
createSession	自动地创建 HttpSession,用来为需要使用到 HttpSession 的拦截器服务
debugging	提供不同的调试用的页面来展现内部的数据状况

续表

名　字	说　明
execAndWait	在后台执行 Action，同时将用户带到一个中间的等待页面
exception	将异常定位到一个画面
fileUpload	提供文件上传功能
i18n	记录用户选择的 locale
logger	输出 Action 的名字
model-driven	如果一个类实现了 ModelDriven，将 getModel 得到的结果放在 Value Stack 中
scoped-model-driven	如果一个 Action 实现了 ScopedModelDriven，则这个拦截器会从相应的 Scope 中取出 model 调用 Action 的 setModel 方法将其放入 Action 内部
params	将请求中的参数设置到 Action 中
actionMappingParams	用来负责 Action 配置中传递参数
prepare	如果 Action 实现了 Preparable，则该拦截器调用 Action 类的 prepare 方法
staticParams	从 struts. xml 文件中＜action＞标签的参数内容设置到对应的 Action 中
scope	将 Action 状态存入 session 和 application 范围
servletConfig	提供访问 HttpServletRequest 和 HttpServletResponse 的方法，以 Map 的方式访问
timer	输出 Action 执行的时间
token	通过 Token 来避免双击
tokenSession	和 Token Interceptor 一样，不过双击的时候把请求的数据存储在 Session 中
validation	使用 action-validation. xml 文件中定义的内容校验提交的数据
workflow	调用 Action 的 validate 方法，一旦有错误返回，重新定位到 INPUT 画面
store	存储或者访问实现 ValidationAware 接口的 Action 类出现的消息、错误和字段错误等
checkbox	添加了 checkbox 自动处理代码，将没有选中的 checkbox 的内容设定为 false，而 HTML 默认情况下不提交没有选中的 checkbox
datetime	日期拦截器
profiling	通过参数激活 profile
roles	确定用户是否具有 JAAS 指定的 Role，否则不予执行
annotationWorkflow	利用注解代替 XML 配置，使用 annotationWorkflow 拦截器可以使用注解，执行流程为 before→execute→fefore Result→after
multiselect	检测是否有像＜select＞标签被选中的多个值，然后添加一个空参数
deprecation	当日志级别设置为调试模式(debug)并且没有特殊参数时，在 devMode 模式，会检查应用程序使用过时的或未知的常量，并且显示警告

Struts2 框架除了提供这些有用的拦截器外，还定义了一些拦截器栈，在开发 Web 应用的时候，可以直接引用这些拦截器栈，而无须自定义拦截器。

注意：随着 Struts2 版本的发展，内建拦截器的数量也在相应地增多，不同版本的

Struts2 拦截器的数量有一些差异，此版本的 Struts2 内置拦截器共有 35 个。

3.2.2 内建拦截器的配置

3.2.1 节中已经了解了一些 Struts2 的内置拦截器以及它们的意义，下面看一下在 struts-default.xml 中定义的拦截器的部分配置。在 struts-core-2.3.24.jar 包中的根目录下找到 struts-default.xml 文件，打开后找到<interceptors>元素下的内建拦截器和拦截器栈，其部分代码如下所示：

```
<package name="struts-default" abstract="true">
    ⋮
    <interceptors>
        <!--系统内建拦截器部分,3.2.1 节介绍的内容-->
        <interceptor name="alias"
        class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
        <interceptor name="chain"
        class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
        ⋮
        <!--定义 Basic stack 拦截器栈-->
        <interceptor-stack name="basicStack">
            <!--引用系统定义的 exception 拦截器 -->
            <interceptor-ref name="exception"/>
            ⋮
        </interceptor-stack>
        ⋮
        <!-定义 Sample model-driven stack  -->
        <interceptor-stack name="modelDrivenStack">
            <!--引用系统定义的 modelDriven 拦截器-->
            <interceptor-ref name="modelDriven"/>
            <!--引用系统定义的 basicStack 拦截器栈 -->
            <interceptor-ref name="basicStack"/>
        </interceptor-stack>
            ⋮
        <!--定义 defaultStack 拦截器栈 -->
        <interceptor-stack name="defaultStack">
            <interceptor-ref name="exception"/>
            <interceptor-ref name="alias"/>
            <interceptor-ref name="i18n"/>
            ⋮
            <interceptor-ref name="validation">
                <param name="excludeMethods">input,back,cancel,browse</param>
            </interceptor-ref>
            ⋮
        </interceptor-stack>
    </interceptors>
    <!--将 defaultStack 拦截器栈配置为系统默认拦截器栈-->
    <default-interceptor-ref name="defaultStack"/>
    <!-默认 action 类是 ActionSupport-->
    <default-class-ref class="com.opensymphony.xwork2.ActionSupport" />
</package>
```

上面内建拦截器的配置代码中，defaultStack 拦截器组合了多个拦截器，这些拦截器的顺序经过精心的设计可以满足大部分 Web 应用程序的需求，只要定义包的过程中继承 struts-default 包，那么 defaultStack 拦截器栈就是默认拦截器的引用。因篇幅有限，这里没有列出所有的内建拦截器和拦截器栈，读者需要时，可以自行查阅 struts-default. xml 文件。

3.3　自定义拦截器

在实际的项目开发中，Struts2 的内置拦截器可以完成大部分的拦截任务，但是一些与系统逻辑相关的通用功能（如权限的控制、用户登录控制等），则需要通过自定义拦截器来实现。本节将详细讲解如何自定义拦截器。

3.3.1　实现自定义拦截器

在程序开发过程中，如果需要开发自己的拦截器类，就需要直接或间接地实现 com. opensymphony. xwork2. interceptor. Interceptor 接口，具体的代码如下：

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

该接口提供了以下三个方法。

- 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与拦截器相关的资源。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。

3.3.2 应用案例——使用拦截器实现权限控制

通过之前对拦截器的学习，可将自定义拦截器的使用过程分为以下 3 步：

(1) 用户自定义的拦截器类，必须实现 Interceptor 接口或继承 AbstractInterceptor 类；

(2) 需要在 struts.xml 中定义自定义的拦截器；

(3) 在 struts.xml 中的 Action 中使用拦截器。

为了让读者更好地掌握自定义拦截器的使用，下面通过一个具体案例来演示自定义拦截器的实现过程：

(1) 在 Eclipse 中创建一个 Web 项目 chapter03，将 Struts2 框架所需的 JAR 包添加到 WEB-INF 目录下的 lib 文件夹中，然后在 WEB-INF 目录下创建一个 web.xml 文件，并在其中注册过滤器和首页，如文件 3-1 所示。

文件 3-1 web.xml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
6       <!--定义 filter -->
7       <filter>
8           <!--filter 名字 -->
9           <filter-name>struts2</filter-name>
10          <!--fiter 的实现类,此处是 Struts2 的核心过滤器 -->
11          <filter-class>org.apache.struts2.dispatcher.ng.filter
12            StrutsPrepareAndExecuteFilter</filter-class>
13      </filter>
14      <filter-mapping>
15          <!--filter 的名字,必须是 filter 元素中已经声明过的过滤器的名字 -->
16          <filter-name>struts2</filter-name>
17          <!--定义 filter 负责拦截的 URL 地址 -->
18          <url-pattern>/*</url-pattern>
19      </filter-mapping>
20      <!--首页 -->
21      <welcome-file-list>
22          <welcome-file>main.jsp</welcome-file>
```

```
23      </welcome-file-list>
24  </web-app>
```

(2) 在 src 目录下创建一个名为 cn.itcast.domain 的包,在该包下创建 User.java 文件,如文件 3-2 所示。

文件 3-2　User.java

```
1   package cn.itcast.domain;
2   public class User {
3       private String username;                    //用户名
4       private String password;                    //密码
5       public String getUsername(){
6           return username;
7       }
8       public void setUsername(String username){
9           this.username=username;
10      }
11      public String getPassword(){
12          return password;
13      }
14      public void setPassword(String password){
15          this.password=password;
16      }
17  }
```

在文件 3-2 中,定义了 username 和 password 两个属性,及其 getters 和 setters 方法。

(3) 在 src 目录下创建包 cn.itcast.action,在该包下创建 LoginAction.java 文件,如文件 3-3 所示。

文件 3-3　LoginAction.java

```
1   package cn.itcast.action;
2   import cn.itcast.domain.User;
3   import com.opensymphony.xwork2.ActionContext;
4   import com.opensymphony.xwork2.ActionSupport;
5   import com.opensymphony.xwork2.ModelDriven;
6   public class LoginAction extends ActionSupport
7                           implements ModelDriven<User>        {
8       private static final long serialVersionUID=1L;
9       private User user=new User();
10      public User getModel(){
11          return user;
12      }
13      public String execute()throws Exception {
14          //获取 ActionContext
15          ActionContext actionContext=ActionContext.getContext();
16          if("tom".equals(user.getUsername())
17                  && "123".equals(user.getPassword())){
18              //将用户存储在 session 中
19              actionContext.getSession().put("user", user);
```

```
            return SUCCESS;
        } else {
            actionContext.put("msg", "用户名或密码不正确");
            return INPUT;
        }
    }
}
```

(4) 在 cn.itcast.action 包下创建 BookAction.java 文件,如文件 3-4 所示。

文件 3-4 BookAction.java

```
package cn.itcast.action;
import com.opensymphony.xwork2.ActionSupport;
public class BookAction extends ActionSupport {
    public String add(){
        System.out.println("book add");
        return SUCCESS;
    }
    public String del(){
        System.out.println("book del");
        return SUCCESS;
    }
    public String update(){
        System.out.println("book update");
        return SUCCESS;
    }
    public String find(){
        System.out.println("book find");
        return SUCCESS;
    }
}
```

(5) 在 src 目录下创建一个名称为 cn.itcast.interceptor 的包,在该包下创建一个 PrivilegeInterceptor.java 文件,如文件 3-5 所示。

文件 3-5 PrivilegeInterceptor.java

```
package cn.itcast.interceptor;
import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
public class PrivilegeInterceptor extends AbstractInterceptor {
private static final long serialVersionUID=1L;
public String intercept(ActionInvocation invocation)throws Exception {
        //得到 ActionContext
        ActionContext actionContext=invocation.getInvocationContext();
        //获取 user 对象
        Object user=actionContext.getSession().get("user");
        if(user !=null){
```

```
            return invocation.invoke();        //继续向下执行
        } else {
            actionContext.put("msg", "您还未登录,请先登录");
            return Action.LOGIN;               //如果用户不存在,返回 login 值
        }
    }
}
```

(6) 在 WebContent 目录下创建 3 个视图页面,分别为主页 main.jsp、登录页面 login.jsp 和操作成功页面 success.jsp,如文件 3-6、文件 3-7 和文件 3-8 所示。

文件 3-6　main.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>main.jsp</title>
  </head>
  <body>
      <a href="/chapter03/book_del">book del</a><br>
      <a href="/chapter03/book_add">book add</a><br>
      <a href="/chapter03/book_update">book update</a><br>
      <a href="/chapter03/book_find">book find</a><br>
  </body>
</html>
```

在文件 3-6 中,定义了 4 个链接,分别代表对图书的增、删、改、查操作。

文件 3-7　login.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>登录</title>
</head>
<body>
<center>
${requestScope.msg}<br>
    <form action="/chapter03/login.action" method="post">
        <table>
            <tr>
                <td><label style="text-align: right;">用户名:</label></td>
                <td><input type="text" name="username"></td>
            </tr>
            <tr>
                <td><label style="text-align: right;">密码:</label></td>
                <td><input type="password" name="password"></td>
            </tr>
            <tr>
```

```
                <td align="right" colspan="2">
                    <input type="submit" value="登录">
                </td>
            </tr>
        </table>
    </form>
</center>
</body>
</html>
```

在文件 3-7 中，分别定义了一个文本框、密码输入框和“登录”按钮，该页面是用户登录的页面。

文件 3-8 success.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>成功页面</title>
</head>
<body>
    用户${user.username}操作成功
</body>
</html>
```

在文件 3-8 中，使用了 EL 表达式获取用户名，并提示用户操作成功。

(7) 在 src 目录下创建 struts.xml 文件，此文件用于声明自定义拦截器、拦截器栈以及对 book 操作的 Action，如文件 3-9 所示。

文件 3-9 struts.xml

```
<?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="struts2" namespace="/" extends="struts-default">
        <!--声明拦截器 -->
        <interceptors>
            <interceptor name="privilege"
                class="cn.itcast.interceptor.PrivilegeInterceptor"/>
            <interceptor-stack name="myStack">
                <interceptor-ref name="defaultStack" />
                <interceptor-ref name="privilege" />
            </interceptor-stack>
        </interceptors>
        <!--用户登录操作 -->
        <action name="login" class="cn.itcast.action.LoginAction">
            <result>/main.jsp</result>
            <result name="input">/login.jsp</result>
```

```
20          </action>
21          <!--关于 book 操作 -->
22          <action name="book_*" class="cn.itcast.action.BookAction"
23              method="{1}">
24              <result>/success.jsp</result>
25              <result name="login">/login.jsp</result>
26              <!--在 action 中使用自定义拦截器 -->
27              <interceptor-ref name="myStack" />
28          </action>
29      </package>
30  </struts>
```

在文件 3-9 中，首先声明了一个名称为 privilege 的拦截器，并将该拦截器放入自定义的拦截器栈 myStack 中。然后配置了用户登录的 action 信息。最后定义了关于对页面 book 操作的 action 信息，并在该 action 中引用自定义的拦截器栈 myStack。

(8) 运行程序，查看结果。

启动 chapter03 程序，在浏览器地址栏中输入“http://localhost:8080/chapter03/main.jsp”，成功访问后，浏览器的显示结果如图 3-2 所示。

在单击页面中链接时，由于用户没有登录，所以没有相应的权限，页面会跳转到登录页面要求用户登录，浏览器的显示结果如图 3-3 所示。

图 3-2　主页

图 3-3　登录页面

如果用户输入错误的用户名和密码，系统也会有相应的错误提示信息，浏览器的显示结果如图 3-4 所示。

输入正确的用户名和密码后，系统会重新跳转到主页，此时单击页面中的链接，系统会跳转到成功页面，提示用户操作成功，浏览器的显示结果如图 3-5 所示。

图 3-4　登录失败

图 3-5　操作成功页面

分别单击首页中 book 操作的每个链接，可以看到 Eclipse 控制台输出的信息，如图 3-6 所示。

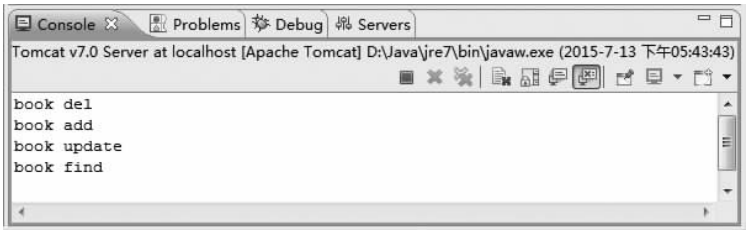

图 3-6　控制台信息

从图 3-6 中可以看出，登录后的用户对页面上的链接都可以进行操作，这说明配置文件中定义的权限拦截器已成功执行。

在上面的案例中，创建了一个方法过滤拦截器 PrivilegeInterceptor，然后在 struts. xml 中配置了该拦截器，如果用户没有登录，将无法对页面链接进行相应操作，只有登录后的用户才有权限操作页面相应功能。

3.4　本章小结

本章首先介绍拦截器的基础知识，讲解了拦截器的配置和使用方法，然后介绍 Struts2 的内置拦截器，最后介绍自定义拦截器的实现方式，并使用自定义拦截器，对用户登录进行权限控制。通过本章的学习，读者需对 Struts2 拦截器的工作原理有更深的了解，能够很好地掌握拦截器的配置和使用方法，并且学会如何配置和使用自定义拦截器。

【思考题】

1. 请简述拦截器的工作原理。
2. 请说明如何对拦截器进行配置。

扫描右方二维码，查看思考题答案！

第4章 Struts2的标签库

学习目标

- 了解 Struts2 的标签库
- 掌握 Struts2 常用标签的使用

对于一个 MVC 框架而言，重点是实现两部分：业务逻辑控制器部分和视图页面部分。Struts2 作为一个优秀的 MVC 框架，也把重点放在了这两部分上。控制器主要由 Action 来提供支持，而视图则是由大量的标签来提供支持。本章将详细介绍 Struts2 标签库的构成和常用标签的使用。

4.1 Struts2 标签库概述

在 JavaWeb 中，Struts2 标签库是一个比较完善，而且功能强大的标签库，它将所有标签都统一到一个标签库中，从而简化了标签的使用，它还提供主题和模板的支持，极大地简化了视图页面代码的编写，同时它还提供对 Ajax 的支持，大大丰富了视图的表现效果。与 JSTL(JSP Standard Library，JSP 标准标签库)相比，Struts2 标签库更加易用和强大。

4.1.1 Struts2 标签库的分类

早期的 JSP 页面需要嵌入大量的 Java 脚本来进行输出，这样使得一个简单的 JSP 页面加入了大量的代码，不利于代码的可维护性和可读性。随着技术的发展，逐渐地采用标签库来进行 JSP 页面的开发，这使得 JSP 页面能够在很短的时间内开发完成，而且代码通俗易懂，极大地方便了开发者，Struts2 的标签库就是这样发展起来的。

Struts2 框架对整个标签库进行了分类，按其功能大致可分为两类，如图 4-1 所示。

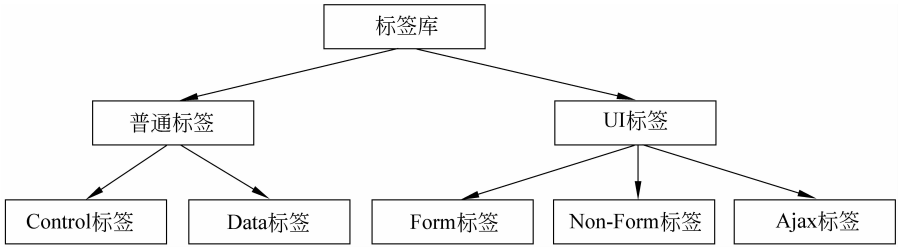

图 4-1 标签分类

由图 4-1 中可以看出，Struts2 标签库主要分为两类：普通标签和 UI 标签。普通标签

主要是在页面生成时，控制执行的流程。UI 标签则是以丰富而可复用的 HTML 文件来显示数据。

普通标签又分为控制标签(Control Tags)和数据标签(Data Tags)。控制标签用来完成条件逻辑、循环逻辑的控制，也可用来做集合的操作。数据标签用来输出后台的数据和完成其他数据访问功能。

UI 标签又分为表单标签(Form Tags)、非表单标签(Non-Form Tags)和 Ajax 标签。表单标签主要用来生成 HTML 页面中的表单元素，非表单标签主要用来生成 HTML 的 标签及输出 Action 中封装的信息等。Ajax 标签主要用来提供 Ajax 技术支持。

4.1.2 Struts2 标签的使用

Struts2 标签库被定义在 struts-tags.tld 文件中，读者可以在 struts-core-2.3.24.jar 中的 META-INF 目录下找到它。要使用 struts2 的标签库，一般只需在 JSP 文件使用 taglib 指令导入 Struts2 标签库，具体代码如下：

```
<%@taglib prefix="s" uri="/struts-tags" %>
```

在上述代码中，taglib 指令的 uri 属性用于指定引入标签库描述符文件的 URI(统一资源标识符)，prefix 属性用于指定引入标签库描述符文件的前缀。需要注意的是，在 JSP 文件中，所有的 Struts2 标签都建议使用 s 前缀。

4.2 Struts2 的控制标签

在程序开发中，经常要用流程控制实现分支、循环等操作，为此，Struts2 标签库中提供了控制标签，常用的逻辑控制标签主要包括<s:if>、<s:elseif>、<s:else>和<s:iterator>等。本节将详细介绍这 4 个常用标签。

4.2.1 <s:if>标签、<s:elseif>标签、<s:else>标签

与多数编程语言中的 if、elseif 和 else 语句的功能类似，<s:if>、<s:elseif>、<s:else>这 3 个标签用于程序的分支逻辑控制。其中，只有<s:if>标签可以单独使用，而<s:elseif>、<s:else>都必须与<s:if>标签结合使用，其语法格式如下所示：

```
<s:if test="表达式 1">
    标签体
</s:if>
<s: elseif test="表达式 2">
    标签体
</s: elseif >
<s: else >
    标签体
</s :else >
```

上述语法格式中，<s:if>和<s:elseif>标签必须指定 test 属性，该属性用于设置标签的判断条件，其值为 boolean 型的条件表达式。

4.2.2　<s:iterator>标签

<s:iterator>标签主要用于对集合中的数据进行迭代，它可以根据条件遍历集合中的数据。<s:iterator>标签的属性及相关说明如表 4-1 所示。

表 4-1　Iterator 标签的属性

属性	是否必须	默 认 值	类型	描　　述
begin	否	0	Integer	迭代数组或集合的起始位置
end	否	数组或集合的长度大小减 1，假如 step 为负则为 0	Integer	迭代数组或集合的结束位置
status	否	false	Boolean	迭代过程中的状态
step	否	1	Integer	指定每一次迭代后索引增加的值
value	否	无	String	迭代的数组或集合对象
var	否	无	String	将生成的 Iterator 设置为 page 范围的属性
id	否	无	String	指定了集合元素的 id，现已用 var 代替

在表 4-1 中，如果在<s:iterator>标签中指定 status 属性，那么通过该属性可以获取迭代过程中的状态信息，如元素数、当前索引值等。通过 status 属性获取信息的方法如表 4-2 所示（假设其属性值为 st）。

表 4-2　通过 status 属性获取状态信息

方　法	说　　明
st.count	返回当前已经遍历的集合元素的个数
st.first	返回当前遍历元素是否为集合的第一个元素
st.last	返回当前遍历元素是否为集合的最后一个元素
st.index	返回遍历元素的当前索引值
st.even	返回当前遍历的元素的索引是否为偶数
st.odd	返回当前遍历的元素的索引是否为奇数

为了让读者更好地掌握<s:if>、<s:else>和<s:iterator>标签的使用，下面通过一个案例来演示它们的使用方法。

在 Eclipse 中创建 Web 项目 chapter04，将 Struts2 框架所需的 JAR 包添加到 WEB-INF 目录下的 lib 文件夹中，然后在 WEB-INF 目录下添加 web.xml 文件，并在其中注册核心过滤器和首页信息，在 WebContent 下创建一个名称为 iteratorTags.jsp 的文件，如文件 4-1 所示。

文件 4-1　iteratorTags.jsp

```
1    <%@page language="java" contentType="text/html; charset=UTF-8"
2        pageEncoding="UTF-8"%>
```

```
<%@taglib prefix="s" uri="/struts-tags" %>
<!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>控制标签的使用</title>
</head>
<body>
  <center>
    <table border="1px" cellpadding="0" cellspacing="0">
        <s:iterator var="name"
                    value="{'Java','Java Web','Oracle','MySql'}"
                    status="st">
            <s:if test="#st.odd">
                <tr style="background-color: white;">
                    <td><s:property value="name"/></td>
                </tr>
            </s:if>
            <s:else>
                <tr style="background-color: gray;">
                    <td><s:property value="name"/></td>
                </tr>
            </s:else>
        </s:iterator>
    </table>
  </center>
</body>
</html>
```

在文件 4-1 的<table>标签内，首先使用<s:iterator>标签来循环输出集合中的值，然后通过该标签 status 属性的 odd 方法获取的值作为<s:if>、<s:else>标签的判断条件来对行数显示进行控制。需要注意的是，这里使用的是<s:property>标签输出的值，该标签会在 4.3 节中具体讲解，这里读者只需知道该标签的作用是输出值即可。

项目正确运行后，在浏览器地址栏中输入“http://localhost:8080/chapter04/iteratorTags.jsp”，成功访问后，浏览器的显示效果如图 4-2 所示。

图 4-2 <s:if>、<s:else>和<s:iterator>标签的使用效果

从图 4-2 的运行结果可以看出，表格的奇数行变为了白色，偶数行变为灰色。这是因为在文件 4-1 中，使用<s:iterator>遍历新创建的 List 集合时，通过判断其所在索引的奇偶来决定表格的颜色。

4.3　Struts2 的数据标签

在 Struts2 标签库中，数据标签主要用于各种数据访问相关的功能以及 Action 的调用等。常用的数据标签有＜s:property＞、＜s:a＞、＜ s:debug＞、＜s:include＞、＜ s:param＞等。

4.3.1　＜s:property＞标签

＜s:property＞标签用于输出指定的值，通常输出的是 value 属性指定的值，＜s:property＞标签的属性及属性说明如下所示。

- id：可选属性，指定该元素的标识。
- default：可选属性，如果要输出的属性值为 null，则显示 default 属性的指定值。
- escape：可选属性，指定是否忽略 HTML 代码。
- value：可选属性，指定需要输出的属性值，如果没有指定该属性，则默认输出 ValueStack 栈顶的值(关于值栈内容会在第 5 章进行讲解)。

接下来编写一个 propertyTags.jsp 页面，来演示 property 标签的使用，如文件 4-2 所示。

文件 4-2　propertyTags.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>property 标签</title>
</head>
 <body>
  输出字符串：
  <s:property value="'www.itcast.cn'"/><br>
  忽略 HTML 代码：
  <s:property value="'<h3>www.itcast.cn</h3>'" escape="true"/><br>
  不忽略 HTML 代码：
  <s:property value="'<h3>www.itcast.cn</h3>'" escape="false"/><br>
  输出默认值：
  <s:property value="" default="true"/><br>
</body>
</html>
```

在文件 4-2 中，定义了 4 个不同属性的＜s:property＞标签。第一个标签中，只有一个 value 属性，所以会直接输出 value 值；第二个标签，使用了 value 和 escape 两个属性，其中 value 属性值中包含了 html 标签，但其 escape 属性值为 true，表示忽略 HTML 代码，所以会直接输出 value 值；第三个标签，同样使用了 value 和 escape 两个属性，其 value 属性值中依然包含了 HTML 标签，但其 escape 属性值为 false，表示不能忽略 HTML 代码，所以输出的 value 值为 3 号标题的值；最后一个标签使用了 value 和 default 两个属性，但 value 的值为空，并且指定了 default 属性，所以最后会输出 default 属性指定的值。

在浏览器地址栏中输入“http://localhost:8080/chapter04/propertyTags.jsp”，成功访问后，浏览器的显示结果如图 4-3 所示。

图 4-3　property 标签的输出

从图 4-3 中可以看出，<s:property>标签的属性不同，其输出的结果也不同，读者在使用此标签时一定要注意其属性的设置。

4.3.2　<s:a>标签

<s:a>标签用于构造 HTML 页面中的超链接，其使用方式与 HTML 中的<a>标签类似。<s:a>标签的属性及相关说明如表 4-3 所示。

表 4-3　<s:a>标签的常用属性及描述

属　性	是否必须	类型	描　　述
action	否	String	指定超链接 Action 地址
href	否	String	超链接地址
namespace	否	String	指定 Action 地址
id	否	String	指定其 id
method	否	String	指定 Action 调用方法

<s:a>标签的使用格式如下所示：

```
<s:a href="链接地址"></s:a>
<s:a namespace="" action="">itcast.cn</s:a>
```

4.3.3　<s:debug>标签

<s:debug>标签用于在调试程序时输出更多的调试信息，主要输出 ValueStack 和 StackContext 中的信息，该标签只有一个 id 属性，且一般不使用。

在使用 debug 标签后，网页中会生成一个[Debug]的链接，单击该链接，网页中将输出各种服务器对象的信息，如图 4-4 所示。