第 3 章　拦截器、转换器与校验器

本章学习目标

- 了解拦截器的原理。
- 了解系统自带拦截器。
- 掌握配置拦截器拦截指定的方法。
- 拦截器有关的案例——权限控制。
- 了解转换器与校验器的使用。

本章首先介绍 Struts2 的核心拦截器，让读者能够使用 Struts2 的拦截器实现权限控制、日志打印、密码 MD5 加密等功能。通过转换器与校验器的使用，增加项目的安全验证。

3.1　拦截器的基本原理

拦截器的工作原理如图 3.1 所示，每一个 Action 请求都包装在一系列的拦截器的内部。拦截器可以在 Action 执行前做相似的操作，也可以在 Action 执行后做回收操作。

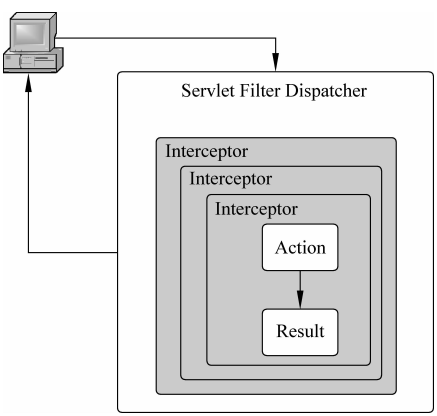

图 3.1　拦截器的工作原理图

每一个 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指定的画面。

3.2　Struts2（XWork）提供的拦截器

拦截器能在 action 被调用之前和被调用之后执行一些“代码”。Struts2 框架的大部分核心功能都是通过拦截器来实现的，如防止重复提交、类型转换、对象封装、校验、文件上传、页面预装载等，都是在拦截器的帮助下实现的。每一个拦截器都是独立装载的，人们可以根据实际需要为每一个 action 配置它所需要的拦截器。Struts2 自带拦截器如表 3.1 所示。

表 3.1 Struts2 自带拦截器

拦 截 器	名 字	说 明
Alias Interceptor	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
Chaining Interceptor	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(<result type="chain">)结合使用
Checkbox Interceptor	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
Conversion Error Interceptor	conversionError	将错误从 ActionContext 中添加到 Action 的属性字段中
Create Session Interceptor	createSession	自动地创建 HttpSession，用来为需要使用到 HttpSession 的拦截器服务
Debugging Interceptor	debugging	提供不同的调试用的页面来展现内部的数据状况
Execute and Wait Interceptor	execAndWait	在后台执行 Action，同时将用户带到一个中间的等待页面
Exception Interceptor	exception	将异常定位到一个画面
File Upload Interceptor	fileUpload	提供文件上传功能
I18n Interceptor	i18n	记录用户选择的 locale
Logger Interceptor	logger	输出 Action 的名字
Message Store Interceptor	store	存储或者访问实现 ValidationAware 接口的 Action 类出现的消息、错误、字段错误等
Model Driven Interceptor	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	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
Parameters Interceptor	params	将请求中的参数设置到 Action 中去
Prepare Interceptor	prepare	如果 Action 实现了 Preparable，则该拦截器调用 Action 类的 prepare 方法
Scope Interceptor	scope	将 Action 状态存入 session 和 application 的简单方法
Servlet Config Interceptor	servletConfig	提供访问 HttpServletRequest 和 HttpServletResponse 的方法，以 Map 的方式访问
Static Parameters Interceptor	staticParams	从 struts.xml 文件中将<action>中的<param>中的内容设置到对应的 Action 中
Roles Interceptor	roles	确定用户是否具有 JAAS 指定的 Role，否则不予执行

续表

拦 截 器	名 字	说 明
Timer Interceptor	timer	输出 Action 执行的时间
Token Interceptor	token	通过 Token 来避免双击
Token Session Interceptor	tokenSession	和 Token Interceptor 一样，不过双击时把请求的数据存储在 Session 中
Validation Interceptor	validation	使用 action-validation. xml 文件中定义的内容校验提交的数据
Workflow Interceptor	workflow	调用 Action 的 validate 方法，一旦有错误返回，重新定位到 INPUT 画面
Parameter Filter Interceptor	N/A	从参数列表中删除不必要的参数
Profiling Interceptor	profiling	通过参数激活 profile

3.3 自定义拦截器

系统自带的拦截功能已经比较强，如果没有特殊的要求一般可以满足，如果有特殊的要求，可以自定义拦截器，自定义拦截器的步骤一般分为 3 步。

3.3.1 建立拦截器的实现类

自定义一个实现 Interceptor 接口（或者继承自 AbstractInterceptor）的类。在 com. yz . interceptor 包下建立 LoggerInterceptor，让该类实现 Interceptor（com. opensymphony . xwork2. interceptor. Interceptor）接口。

LoggerInterceptor 拦截器实现的代码如下：

```
package com.yz.interceptor;

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.Interceptor;

public class LoggerInterceptor implements Interceptor {
    public void destroy() {
    }
    public void init() {
    }
    public String intercept(ActionInvocation chain) throws Exception {
        System.out.println("=========前置拦截操作==========");
        String result=chain.invoke();
        System.out.println("=========后置拦截操作==========");
        return result;
    }
}
```

3.3.2 在 struts.xml 中注册自定义的拦截器

如下代码所示，声明了一个新的子包 cmmonpackage，该包继承自默认包 struts-default。在 cmmonpackage 中定义了一个拦截器，拦截器的名字为 mylogger，对应的实现类为 com.yz.interceptor.LoggerInterceptor。

```
<package name="cmmonpackage" extends="struts-default">
        <!--声明拦截器 -->
    <interceptors>
        <interceptor name="mylogger" class="com.yz.interceptor.LoggerInterceptor">
            </interceptor>
    </interceptors>
</package>
```

3.3.3 在 Action 中引用拦截器

在 struts.xml 中 action 的配置处增加拦截器的映射，示例代码如下：

```
<package name="mypackage" extends="cmmonpackage">
        <action name="user_*" class="com.yz.action.UserAction" method="{1}">
            <result>ok.jsp</result>
            <interceptor-ref name="mylogger"></interceptor-ref>
            <interceptor-ref name="defaultStack"></interceptor-ref>
        </action>
</package>
```

＜interceptor-ref name＝"mylogger"＞＜/interceptor-ref＞这句代码指定了 action 使用 mylogger 这个拦截器，此处的 name 对应 3.3.2 节中声明拦截器的 name 属性。

注意：虽然 extends 继承的 struts-default 自带 params 拦截器，但是在人们自己引用拦截器时，继承 struts-default 将不会再分配默认的拦截器（有点类似构造器），不过仍然可以通过＜interceptor-ref name＝"defaultStack"/＞来继续使用 struts-defalut 的拦截器。

3.3.4 拦截器执行效果

在地址栏中输入 http://localhost:8087/struts2002/user_add.action，按 Enter 键后返回结果如图 3.2 所示。

```
==========前置拦截操作===========
执行用户添加操作
==========后置拦截操作===========
```

图 3.2 添加用户中使用拦截器后的效果

通过测试发现，此方法配置后，不管请求 user_add.action 或 user_del.action 等 action，拦截器都可以成功在 action 执行函数调用前和调用后执行相应的拦截器操作。

3.4 拦截器拦截指定方法

自定义拦截器虽然可以实现给指定的 action 配置拦截器，但是存在一个问题，如果给 action 配置了拦截器，则 action 中所有的方法都会应用该拦截器。这样就会造成 action 中某些函数可能不需要拦截器处理，但是现在拦截器依然会处理 action 中的所有方法。如何解决该问题呢？

3.4.1 使拦截器继承自 MethodFilterInteceptor

修改拦截器实现类，让该类继承自 MethodFilterInteceptor，具体代码如下：

```
package com.yz.interceptor;

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.MethodFilterInterceptor;

public class LoggerInterceptor extends MethodFilterInterceptor {
    public String doIntercept(ActionInvocation chain) throws Exception {
        System.out.println("=========前置拦截操作==========");
        String result=chain.invoke();
        System.out.println("=========后置拦截操作==========");
        return result;
    }
}
```

3.4.2 在 action 中配置相应的拦截器

在 Action 中配置相应的拦截器代码如下：

```
<?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="cmmonpackage" extends="struts-default">
        <!--声明拦截器 -->
    <interceptors>
        <interceptor name="mylogger" class="com.yz.interceptor.LoggerInterceptor">
</interceptor>
    </interceptors>
    </package>

    <package name="mypackage" extends="cmmonpackage">
        <action name="user_*" class="com.yz.action.UserAction" method="{1}">
            <result>ok.jsp</result>
```

```
            <interceptor-ref name="mylogger">
                <param name="includeMethods">add</param>
            </interceptor-ref>
            <interceptor-ref name="defaultStack"></interceptor-ref>
        </action>
    </package>
</struts>
```

在上述代码中，配置的 mylogger 拦截器只拦截 add 方法，对 del 和 update 方法不进行拦截。只需在＜param name＝"includeMethods"＞add＜/param＞中，指定拦截哪些方法即可。另外还可以通过配置＜param name＝"excludeMethods"＞del，update＜/param＞，来控制拦截器不拦截哪些方法。注意 includeMethods 与 excludeMethods 只可以使用其中一个来控制。拦截器使用后的效果图如图 3.3 所示。

```
==========前置拦截操作============
执行用户添加操作
==========后置拦截操作============
执行用户删除操作
执行用户修改操作
```

图 3.3　拦截器使用后的效果图

3.5　拦截器有关的案例——权限控制

为了说明此问题，下面建立 Struts2auth 项目，拦截器流程图如图 3.4 所示。

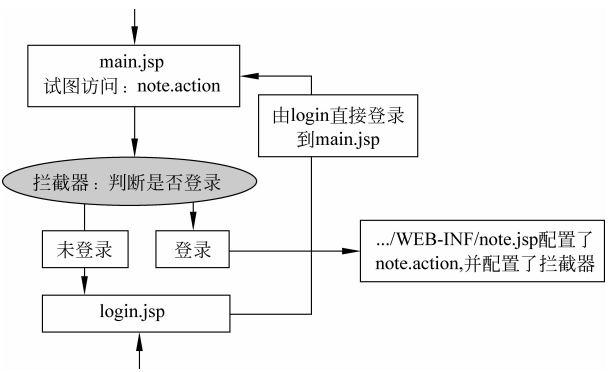

图 3.4　拦截器流程图

说明：当人们访问 main.jsp 页面并试图通过此页面中的链接地址 http://127.0.0.1:8080/security/note.action 来访问到.../WEB-INF/note.jsp 页面时，由于访问的 note.action 配置了拦截器，所以会被拦截，如果拦截器判断已登录则可以访问，否则会跳到登录页面。如果从登录页面直接到 main.jsp 页面，再来访问 note.action 时，同样被拦截，但是由于登录过，所以可访问此 action 对应的内容。由这里的分析可以看出关键点就是登录成功时要给出标志，以便拦截器判断是否成功登录。

(1) 搭建好相关的开发环境，并准备好登录页面 login.jsp，代码如下：

```
<form action="<%=request.getContextPath()%>/login.action" method="post">
        姓名:<input type="text" name="username"><br>
        密码:<input type="password" name="password"><br>
        <input type="submit" value="登录">
</form>
```

(2) 建立相应的 Action:LoginAction,代码如下:

```
package com.asm;
public class LoginAction extends ActionSupport {
    private String username;
    Map session;
    public String execute() throws Exception {
        if(username.equals("admin")){
            session=ActionContext.getContext().getSession();
            session.put("loginSign", "loginSuccess");
            return SUCCESS;
        }else{
            return LOGIN;
        }
}
... //省略 username 的 get/set 方法
}
```

说明:这里设定了只有登录用户名为 admin 时,此 Action 才设置登录标志。另外,这里获取 Session 对象采取的是"与 Servlet 解耦合的非 IOC 方式"。

(3) 编写拦截器类,代码如下:

```
package com.asm.interceptor;
public class AuthInterceptor extends AbstractInterceptor {
    public String intercept(ActionInvocation invocation) throws Exception {
        Map session=invocation.getInvocationContext().getSession();
        //session=ActionContext.getContext().getSession();
        if (session.get("loginSign")==null) {
            return "login";
        } else {
            String result=invocation.invoke();
            return result;
        }
    }
}
```

(4) 配置此 Action,主要配置内容如下:

```
<struts>
    <package name="tokenTest" extends="struts-default">
        <interceptors>
```

```
            <interceptor name="auth"
                class="com.asm.interceptor.AuthInterceptor">
            </interceptor>
            <interceptor-stack name="authStack">
                <interceptor-ref name="auth"></interceptor-ref>
                <interceptor-ref name="defaultStack"></interceptor-ref>
            </interceptor-stack>
        </interceptors>
        <action name="login" class="com.asm.LoginAction">
            <result name="success">/main.jsp</result>
            <result name="login">/login.jsp</result>
        </action>

        <action name="note">
            <result>/WEB-INF/note.jsp</result>
            <result name="login">/login.jsp</result>
            <interceptor-ref name="authStack"></interceptor-ref>
        </action>
    </package>
</struts>
```

说明：结合前面的一些代码来看，当为 note.action 配置了前面所写的 AuthInterceptor 拦截器时，如果人们要访问 note.action，拦截器会首先判断是否登录，如果登录则继续把请求传递下去，如果没有登录则会返回到登录页面。

(5) 编写相关的其他 JSP 页面，然后发布测试。此实例重点是进一步掌握拦截器的配置使用。作为“实现资源权限访问”，此实例不具参考价值。

3.6 转换器与校验器

3.6.1 转换器

Struts2 提供了强有力的表现层类型转换机制，无须程序员过多干预即可自动完成转换。Struts 能自动处理类型转换过程中出现的未知异常。

下面的数据类型会自动转换：boolean、char、int、long、float、double 基础类型，包括封装类型和对应数组；Date、String 数组，元素类型为 String 的 List。

图 3.5 点坐标提交

点坐标提交如图 3.5 所示。

点坐标提交的代码如下：

```
package com.anbo.action;

import com.anbo.vo.Pointer;

public class UserAction {
```

```
    private Pointer positon;
    //增加用户的操作
    public String addUser(){
        System.out.println("点坐标的信息为: "+positon.getX()+","+positon.getY());
        return "addsucc";
    }
    //查看用户的操作
    public String showUser(){
        return "showsucc";
    }
    //删除用户的操作
    public String delUser(){

        return "delsucc";
    }
    public Pointer getPositon() {
        return positon;
    }
    public void setPositon(Pointer positon) {
        this.positon=positon;
    }
}
```

如图 3.5 所示,页面中提交的是一个字符串,如果使用 Struts2 自带的转换器则无法将字符串型的数据转换成 Pointer 类型(该类型是作者自定义的)。

下面通过自定义转换器来实现上述功能,首先类型转换器是一个继承自 StrutsTypeConverter (抽象类)的类,必须重写两个方法。

从字符串转换成目标类型:

```
public Object convertFromString(Map context, String[] values, Class toClass)
```

从目标类型转换成字符串:

```
public String convertToString(Map context, Object o)
```

注意:上面两个方法都必须重写。

```
package com.anbo.convertor;

import java.util.Map;

import org.apache.struts2.util.StrutsTypeConverter;

import com.anbo.vo.Pointer;

public class PointerCovertor extends StrutsTypeConverter {
```

```
public Object convertFromString(Map arg0, String[] values, Class arg2) {
    Pointer pp=new Pointer();
    String strs[]=values[0].split(",");
    pp.setX(strs[0]);
    pp.setY(strs[1]);
    return pp;
}

public String convertToString(Map arg0, Object obj) {
    Pointer pp=(Pointer)obj;
    return "点坐标["+pp.getX()+","+pp.getY()+"]";
}

}
```

然后配置注册自定义类型转换器,有两种方法。

(1) 局部类型转换器:只有特定的 Action 才可以使用。

注册方法:在 Action 所在的包中建立 properties 文件,文件名格式为“Action 类名-conversion.properties”,内容如下:

```
inver=com.sdhandson.conversition.IntArrayConverter
```

其中,inver 是 action 的属性名。

(2) 全局类型转换器:所有 Action 都可以使用。

注册方法:在 classpath 下建立名为 xwork-conversion.properties 的文件,内容如下:

```
目标类型(全限定名)=转换器全限定名
```

3.6.2 校验器

框架中的校验器分为手工校验与配置文件校验两种。要想实现校验,action 必须继承自 ActionSupport 类。

1. 基于手工编码的校验

建立 struts2validate 项目,其中 reg.jsp 页面的主要代码如下:

```
<body>
    <s:head/>
    <h3>注册页面</h3>
    <s:form method="post" action="reg" >
        <s:bean name="com.asm.AgeAction" id="aa"></s:bean>
        <s:textfield name="user.username" label="用户名"/>
        <s:property value="errors.user.username"/>
        <s:password name="user.password" label="密码"/>
        <s:password name="user.password2" label="确认密码"/>
        <s:select list="#aa.ageMap" name="user.age" label="年龄" headerValue="填
写真实年龄" headerKey="0"/>
```

```
        <s:reset value="重置" align="left" />
        <s:submit value="注册" align="left"/>
    </s:form>
</body>
```

说明：<s:head/>可以用来对验证信息进行一些美化效果处理，另外，此页面用到了一个 AgeAction 用来动态生成“年龄”表单项，在前面的表单标签中已用过类似的做法。AgeAction 的代码如下：

```
package com.asm;
public class AgeAction extends ActionSupport {
    private Map<Integer, String>ageMap;
    public AgeAction() {
        ageMap=new HashMap();
        for (int i=1; i<=120; i++) {
            ageMap.put(new Integer(i), i +"");
        }
    }
    ...//省略 ageMap 的 get/set 方法
}
```

RegAction 的配置如下：

```
<package name="validate" extends="struts-default">
        <action name="reg" class="com.asm.RegAndLoginAction" method="reg">
            <result name="success">/regSuc.jsp</result>
            <result name="login">/reg.jsp</result>
</action>
</package>
```

根据配置，我们来看它的对应 Action：RegAndLoginAction，代码如下：

```
package com.asm;
public class RegAndLoginAction extends ActionSupport {
    private User user;
    public String reg() throws Exception {
        if (user.getUsername()==null || user.getUsername().equals("")) {
            this.addFieldError("user.username", "用户名不能为空");
         } else if (!Pattern.matches("^[a-zA-Z][a-zA-Z0-9_]{3,14}$", user
.getUsername())) {
            this.addFieldError("user.username", "用户名只能是以字母开头，后面可以跟
字母、数字或下划线，长度只能是 4~15 位");
        } else if (user.getPassword()==null || user.getPassword().equals("")) {
            this.addFieldError("user.password", "密码不能为空");
        } else if (!user.getPassword().equals(user.getPassword2())) {
            this.addFieldError("user.password2", "两次输入的密码不一致，请重新输
入");
        } else if (user.getAge()<16) {
```

```
            this.addFieldError("user.age", "未满 16 岁,不能注册");
        }

        if (this.hasFieldErrors()) {
            return LOGIN;
        }
        System.out.println("reg success…");
        return SUCCESS;
    }
    ...//省略 user 的 get/set 方法
}
```

说明：当 reg.jsp 提交给此 Action 对应的 reg 方法处理时，它会调用 addFieldError 把错误信息加到 FiledError 中去，关于这点，在前台 reg.jsp 页面中用＜s:debug＞调试时，可以看到值栈中此 Action 对象中的 fieldErrors 对应着我们添加的错误信息，因此这点也就为我们取出验证信息提供一个参考，即可以取出此验证信息，对它进行美化处理，而不是按 Struts2 默认来显示。后面，我们接着对登录页面用 login 方法进行了类似的验证(在此省略)，所以此 action 取名为 regAndLoginAction。

补充：当把 login.jsp 页面的验证写完后，可以发现 reg 和 login 这两个方法的显示相当烦琐，对此可以专门把验证分别放在 validateReg 和 validateLogin 方法中去。我们新建一个 Action 来演示，新的 RegAndLogin2Action 的主要代码如下：

```
package com.asm;
public class RegAndLogin2Action extends ActionSupport {
    private User user;

    @Override
    public void validate() {
    System.out.println("校验的统一出口,对所有方法进行校验: 这里可以放一些公共的验证");
    }

    public void validateReg() {
        ... //省略,对 reg 方法进行验证
    }

    public void validateLogin() {
        ... //省略,对 login 方法进行验证
    }

    public String reg() throws Exception {
        System.out.println("reg success…");
        return SUCCESS;
    }
```

```
    public String login() throws Exception {
        System.out.println("login success…");
        return SUCCESS;
    }
    … //省略 user 的 get/set 方法
}
```

说明：当 reg.jsp 提交给此 Action 对应的 reg 方法处理时，它会首先调用此 reg 方法专属的验证方法 valiadteReg(注意取名规则：validate＋方法名＜首字母大写＞)，此方法验证完成后，会调用 validate 方法，此方法完成后才会调用 reg 方法。因此，一般情况下人们会把一些公共的验证放在 validate 方法中，而这些所有的验证方法也只进行验证处理，并把错误信息封装到 fieldError 字段中(或者其他字段)。reg 这些真正执行的方法只进行一些其他处理(比如把注册信息写进数据库)。测试时需要修改，把前面的配置注释掉，写上下面的配置：

```
<action name="login" class="com.asm.RegAndLogin2Action"  method="login">
            <result name="success">/logSuc.jsp</result>
            <result name="input">/login.jsp</result>
</action>
<action name="reg" class="com.asm.RegAndLogin2Action" method="reg">
        <result name="success">/regSuc.jsp</result>
        <result name="input">/reg.jsp</result>
</action>
```

说明：配置中有一个 input result 的配置，因为带有 validate 的方法进行验证时，如果验证失败，会返回 input 所对应的 result 结果集。

简析校验流程如下。

(1) 类型转换器请求参数执行类型转换，并把转换后的值赋给 action 中的属性。

(2) 如果在执行类型转换过程中出现异常，系统会将异常信息保存到 ActionContext，conversionError 拦截器将异常信息添加到 fieldErrors 里，不管类型转换是否出现异常都会进入第(3)步。

(3) 系统通过反射技术调用 action 中的 validateXxx()方法。

(4) 再调用 action 中的 validate()方法。

(5) 经过上面 4 步，如果系统中的 fieldErrors 存在错误信息(即存放错误信息的集合 size 大于 0)，系统自动将请求转发至名为 input 的视图。如果系统中的 fieldErrors 没有任何错误信息，系统将执行 action 中的处理方法。

注意：经过以上过程的分析，可以知道如果类型转换失败，也会到名为 input 的视图。

2. 基于 XML 配置形式的校验

新建 struts2validateXML 项目，在此项目中，基本的代码和上面的 struts2validate 项目相似，只是在上一个项目中我们在 Action 的具体方法中进行了验证处理，现在先修改 RegAndLoginAction 的代码如下：

```
package com.asm;
```

```
public class RegAndLoginAction extends ActionSupport {
    private User user;

    public String reg() throws Exception {
        System.out.println("reg success…");
        return SUCCESS;
    }

    public String login() throws Exception {
        System.out.println("login success…");
        return SUCCESS;
    }
    … //省略 user 的 get/set 方法
}
```

下面在 action 所在的包下建立一个对此 Action 进行校验的 XML 文件，文件名为 RegAndLoginAction-validation. xml，取名原则就是 actionClassName-validation. xml。它会对此 Action 中的所有方法进行校验，主要代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
        "-//OpenSymphony Group//XWork Validator 1.0.3//EN"
        "http://www.opensymphony.com/xwork/xwork-validator-1.0.3.dtd">
<validators>
    <field name="user.username">
        <field-validator type="requiredstring">
            <message>用户名不能为空</message>
        </field-validator>
        <field-validator type="regex">
            <param name="expression">^[a-zA-Z][a-zA-Z0-9_]{3,14}$</param>
<message>
```

用户名只能是以字母开头，后面可以跟字母、数字或下划线，长度只能是 4～15 位：

```
</message>
        </field-validator>
    </field>

    <field name="user.password">
        <field-validator type="requiredstring">
            <message>密码不能为空</message>
        </field-validator>
    </field>
</validators>
```

进行此配置，相当于在 RegAndLoginAciton 中增加用 validate 方法进行验证。如果想对某个方法进行验证，配置文件应取名为 actionClassName-ActionName-validation. xml，比

如对 reg 方法进行验证，在前面用到 validateReg 方法，这里只需增加 RegAndLoginAction-reg-validation.xml 配置文件即可，它的作用和 validateReg 方法相同，在此省略此配置文件的内容。

关于验证的配置文件中用到的验证类型可以参看文档或者参看压缩包中的配置参照文件，下面对校验器类型进行简单说明。

Required-必需校验器：要求 field 的值不能为 null。

Requiredstring-必需字符串校验器：不能为 null，长度大于 0，默认情况下会对字串去前后空格。

int[long、short、double]：整型[long 型、短整型、double 型]值必须在指定范围。参数 min 指定最小值，参数 max 指定最大值。

date-日期校验器：日期校验类型，符合日期格式，可以使用 min/max 来指定日期范围。

expression-OGNL 表达式校验器：expression 参数指定 ognl 表达式，该逻辑表达式基于值栈进行求值，返回 true 时校验通过，否则不通过，该校验器不可用在字段校验器风格的配置中。

fieldexpression-字段 ognl 表达式校验器：要求 field 满足一个 ognl 表达式，expression 参数指定 ognl 表达式，该逻辑表达式基于值栈进行求值，返回 true 校验通过，否则不通过。

email-邮件地址校验器：非空用为合法的邮件地址。

url-网址校验器：非空用为合法的 URL 地址。

visitor-复合属性校验器：它指定一个校验文件用于校验复合属性中的属性。

conversion-转换校验器：指定在类型转换失败时，提示的错误信息。

stringlength-字符长度校验器：要求字段必须在指定的范围内，否则校验失败。minLength 参数指定最小长度，maxLength 参数指定最大长度。Trim 参数指定校验 field 之前是否去除字串前后的空格。

regex-正则表达式校验器：校验字段是否与 expression 参数指定的正则表达式匹配。caseSensitive 参数指定进行匹配时是否区分大小写，默认为 true，即区分大小写。

第 4 章　OGNL 与 Struts2 标签库

本章学习目标

- 了解值栈的概念。
- 了解 OGNL 表达式的定义。
- 掌握 OGNL 的几种访问方法。
- 掌握 Struts2 的几种常用的标签。

本章先向读者介绍 Struts2 中 OGNL 表达式以及几种常用的标签。通过本章让读者能够使用 OGNL 表达式获取 action 中的数据信息，利用 Struts2 的控制标签、UI 标签实现数据的获取和展示。

4.1　Struts2 值栈

简单地说，值栈是对应每一个请求对象的轻量级的数据存储中心，在这里统一管理着数据，供 Action、Result、Interceptor 等 Struts2 的其他部分使用，这样数据被集中管理起来而不凌乱。

简单地说，值栈能够线程安全地为每个请求提供公共的数据存取服务。

当有请求的时候，Struts2 会为每个请求创建一个新的值栈，也就是说，栈和请求是一一对应的，不同的请求，值栈也不一样，而值栈封装了一次请求所有需要操作的相关数据。

正是因为值栈和请求的对应关系，因此值栈能保证线程安全地为每个请求提供公共的数据存取服务。

通常是指 com. opensymphony. xwork2. util. ValueStack 接口的对象，目前就是 com. opensymphony. xwork2. ognl. OgnlValueStack 对象。

狭义值栈主要用来存放动态 EL（表达式语言）运算需要的值和结果，当然 OgnlValueStack 对象主要是用来支持 OGNL(对象图导航语言)运算的。

狭义值栈中存放着一些 OGNL 可以访问的数据，这些数据如下。

(1) action 的实例，这样就可以通过 OGNL 来访问 Action 实例中的属性的值了。

(2) OGNL 表达式运算的值，可以设置到值栈中，可以主动访问值栈对象，强行设置。

(3) OGNL 表达式产生的中间变量，比如使用 Struts2 标签的时候，使用循环标签，自然会有循环变量，这些都放在值栈中。

广义值栈通常是 ActionContext 对象，ActionContext 是 Action 运行的上下文，每个 ActionContext 是一个基本的容器，包含着 Action 运行需要的数据，比如请求参数、会话等。

ActionContext 也是线程安全的，每个线程都有一个独立的 ActionContext，这样就不用担心值栈中值的线程安全问题了。

ActionContext 里面存储着很多值，这些值如下。

(1) Request 的 Parameters，这是请求中的参数，注意这里的数据是从数据对象中复制

来的，因此这里的数据变化不会影响请求对象里面的参数的值。

(2) Request 的 Attribute，这是请求中的属性，这里是一个 Map，存放着请求对象的属性数据，这些数据和请求对象的 Attribute 是联动的。

(3) Application 的 Attribute，这是应用的属性，这里是一个 Map，存放着应用对象的属性数据，这些数据和应用对象的 attribute 是联动的。

(4) ValueStack，也就是狭义值栈，ActionContext 是以 value stack 作为被 OGNL 访问的根，简单地说，OGNL 在没有特别指明的情况下，访问的就是 value stack 的值。

(5) attr，在所有的属性范围中获取值，依次搜索 page、request、session 和 application。

4.2 OGNL 表达式

OGNL(Object-Graph Navigation Language)是一个功能强大的表达式语言，用来获取和设置 Java 对象的属性，它旨在提供一个更高、更抽象的层次来对 Java 对象图进行导航。

OGNL 表达式的基本单位是“导航链”，一般导航链由如下几部分组成。

(1) 属性名称(property)。

(2) 方法调用(method invoke)。

(3) 数组元素。

4.2.1 OGNL 普通方法访问

首先在 User 中增加一个成员方法，代码如下：

```
public String get(){
    return "这是 User 中的 get 方法";
}
```

在 LoginAction 中也有类似的 get 方法，随后再在 loginSuc.jsp 中增加如下代码。

调用值栈对象中的普通方法(2)：

```
<s:property value="user.username.length()"/><br>
```

调用值栈对象中的普通方法(1)：

```
<s:property value="user.get()"/><br>
```

调用 LoginAction 中的普通方法：

```
<s:property value="get()"/><br>
```

最后测试，发现这些方法都可以访问到。

4.2.2 OGNL 静态方法访问

在 LoginAction 中增加如下方法：

```
public static String getSta() {
    return "这是 LoginAction 中的静态方法";
}
```

然后在 loginSuc.jsp 中增加如下代码。

调用 Action 中的静态方法：

```
<s:property value="@com.asm.LoginAction@getSta()"/><br>
```

调用 LoginAction 中的静态方法的方式(2)：

```
<s:property value="@vs@getSta()"/><br>
```

说明：我们在方式(2)中用到@vs，只有那些值栈中的对象才可以这样写。

然后访问，发现访问不到页面，因为在 Struts 2.1.6 的版本中，struts.ognl.allowStaticMethodAccess 的默认值为 false，我们只需在 struts.xml 中增加如下内容：

```
<constant name="struts.ognl.allowStaticMethodAccess" value="true"/>
```

再来访问时便可以访问到页面。

4.2.3 OGNL 默认类 Math 的访问

在 loginSuc.jsp 中增加如下代码。

调用 Math 类中的静态方法：

```
<s:property value="@java.lang.Math@min(1,2)"/><br>
```

调用 Math 类中的静态方法的方式(2)：

```
<s:property value="@@min(1,2)"/><br>
```

调用 Math 类中的字段：

```
<s:property value="@@PI"/><br>
```

说明：因为是默认的类，所以可以省略类名。

4.2.4 OGNL 调用普通类的构造方法

建立一个新的类，名为 Student，在此省略代码。

然后在 loginSuc.jsp 中增加如下代码。

调用普通类中的构造方法：

```
<s:property value="new com.asm.vo.Student('jack','20','85.5')"/><br>
```

调用普通类中的构造方法并访问其字段：

```
<s:property value="new com.asm.vo.Student('jack','20','85.5').name"/>
```

说明：第一种是只创建出对象，显示的时候其实是调用对象的 toString 方法。

4.2.5 OGNL 集合对象

首先在 LoginAction 中增加如下字段并提供相应的 get/set 方法：

```
private List myList=new ArrayList();
```

```
private Set mySet =new HashSet();
private Map myMap =new HashMap();
```

然后再在 execute 方法中初始化这些集合对象,代码如下:

```
myList.add("list1");
myList.add("list2");
myList.add("list3");
myList.add("list4");
mySet.add("set1");
mySet.add("set3");
mySet.add("set1");
mySet.add("set2");
myMap.put("m1", "map1");
myMap.put("m3", "map3");
myMap.put("m2", "map2");
```

最后在 loginSuc.jsp 中增加如下代码。

```
获取 List: <s:property value="myList"/><br>
获取 List 中的第一个元素: <s:property value="myList[0]"/><br>
获取 Set: <s:property value="mySet"/><br>
获取 Set 中的第一个元素(set 无序,不能取到): <s:property value="mySet[0]"/><br>
获取 Map: <s:property value="myMap"/><br>
获取 Map 中的 key=m1 的元素的值: <br>
方式一: <s:property value="myMap.m1"/>
方式二: <s:property value="myMap['m1']"/><br><hr>
获取 List 的大小:
<s:property value="myList.size"/>|<s:property value="myList.size()"/><br>
获取 Map 中所有的键: <s:property value="myMap.keys"/><br>
获取 Map 中所有的值: <s:property value="myMap.values"/><br>
```

4.2.6 OGNL 中 top 用法

在 loginSuc.jsp 中增加如下代码。

```
N 语法[0]: <s:property value="[0]"/><br>
N 语法[1]: <s:property value="[1]"/><br>
N 语法[0].top: <s:property value="[0].top"/><br>
N 语法[1].top: <s:property value="[1].top"/><br>
N 语法 top: <s:property value="top"/><br>
N 语法取值: <s:property value="[0].user.username"/><br>
N 语法取值: <s:property value="top.user.username"/><br>
```

说明:规定栈顶的对象为[0],而我们只使用[0]的意思是从值栈中第一个对象取,一直取至栈底。N 的意思是从值栈中的第 N 个对象开始,取到栈底为止。如果要想访问某个对象,需要使用[N].top,它的意思是取出符合 N 语法的栈顶对象,比如在这里,[0]会取出两个对象,而[0].top 是取出这两个对象的栈顶对象。纯 top 可以简洁地取出值栈中的栈顶

对象。

为什么要提出 N 语法？当人们通过 chain 链访问时，值栈中可能有两个以上的 Action 对象，如果这些对象中存在相同的属性，N 便能正确地区分它们。通常，这些 Action 对象的入栈顺序是先访问先入栈。

从上面的 N 语法取值实例中，可知[N].top 语法的一个重要作用就是能通过它们引用值栈对象中的属性。结合前面的 5 种[N].top 语法实例，不难理解这里的取值实例。

补充：在此实例中，使用 < s：debug > 调试会发现，值栈中还有一个 DefaultTextProvider 对象(因为此 Action 继承自 ActionSupport)，它的作用是获取资源文件中的内容(其实本质是 ActionSupport 重写了 getText()方法)，这也是在国际化问题中人们能直接调用它的 getText()方法的原因。

4.3 标签库

Struts2 标签基本可以分为用户界面标签、非界面标签和 Ajax 标签。对于 Struts2 标签来说，执行的效率并不是很高，所以很多时候企业都会采用自己封装标签库。这样做一方面可以保证界面的统一，另一方面标签的执行效率比较高。

使用 Struts2 标签前，应该引入标签库：

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

标签库分类如图 4.1 所示。

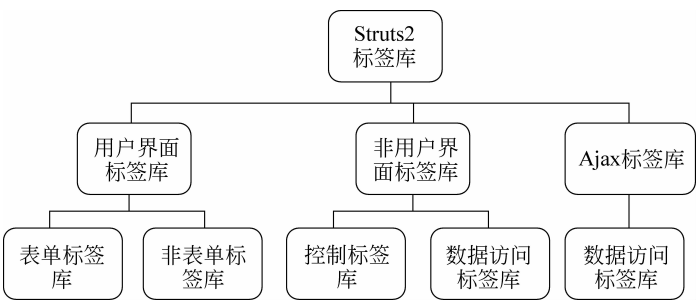

图 4.1 标签库分类

4.3.1 Struts2 的表单标签

Struts2 的表单标签可分为两种：Form 表单本身和单个表单元素的标签。Form 标签本身的行为不同于表单元素的标签。Struts2 的表单元素标签都包含了非常多的属性，但有很多属性完全是通用的。

1. 表单标签的通用属性

所有表单标签处理类都继承了 UIBean 类，UIBean 包含了一些通用属性，分为 3 种。

(1) 模板相关属性。

(2) JavaScript 相关属性。

(3) 通用属性。