

# 1 Mvc 与 servlet

## 1.1 Servlet 的优点

- 1、是 mvc 的基础，其他的框架比如 struts1，struts2，webwork 都是从 servlet 基础上发展过来的。所以掌握 servlet 是掌握 mvc 的关键。
- 2、Servlet 把最底层的 api 暴露给程序员，使程序员更能清楚的了解 mvc 的各个特点。
- 3、程序员可以对 servlet 进行封装。Struts2 就是从 servlet 中封装以后得到的结果。
- 4、市场上任何一个 mvc 的框架都是 servlet 发展过来的，所以要想学好 struts2 这个框架，了解 servlet 的运行机制很关键。

## 1.2 Servlet 的缺点

- 1、每写一个 servlet 在 web.xml 中都要做相应的配置。如果有多很 servlet，会导致 web.xml 内容过于繁多。
- 2、这样的结构不利于分组开发。
- 3、在 servlet 中，doGet 方法和 doPost 方法有 HttpServletRequest 和 HttpServletResponse 参数。这两个参数与容器相关，如果想在 servlet 中作单元测试，则必须初始化这两个参数。
- 4、如果一个 servlet 中有很多个方法，则必须采用传递参数的形式，分解到每一个方法中。

# 2 重构 servlet

针对 servlet 以上的特点，我们可以对 servlet 进行重构，使其开发起来更简单。更容易，更适合团队协作。

重构的目标:

- 1、只写一个 `serlvet` 或者过滤器，我们这里选择过滤器。
- 2、不用再写任何的 `servlet`，这样在 `web.xml` 中写的代码就很少了。
- 3、原来需要写 `serlvet`,现在改写 `action`。
- 4、在 `action` 中把 `HttpServletRequest` 参数和 `HttpServletResponse` 参数传递过去。
- 5、在过滤器中通过 `java` 的反射机制调用 `action`。
- 6、详细过程参照 `cn.itcast.action` 包中的内容

## 3 Struts2 介绍

- 1、`struts2` 是 `apache` 组织发明的开源框架。是 `struts` 的第二代产品。
- 2、`struts2` 是在 `struts` 和 `webwork` 基础上整合的全新的框架。
- 3、`struts2` 的配置文件组织更合理，是企业开发很好的选择。
- 4、`struts2` 的拦截器为 `mvc` 框架注入了全新的概念。

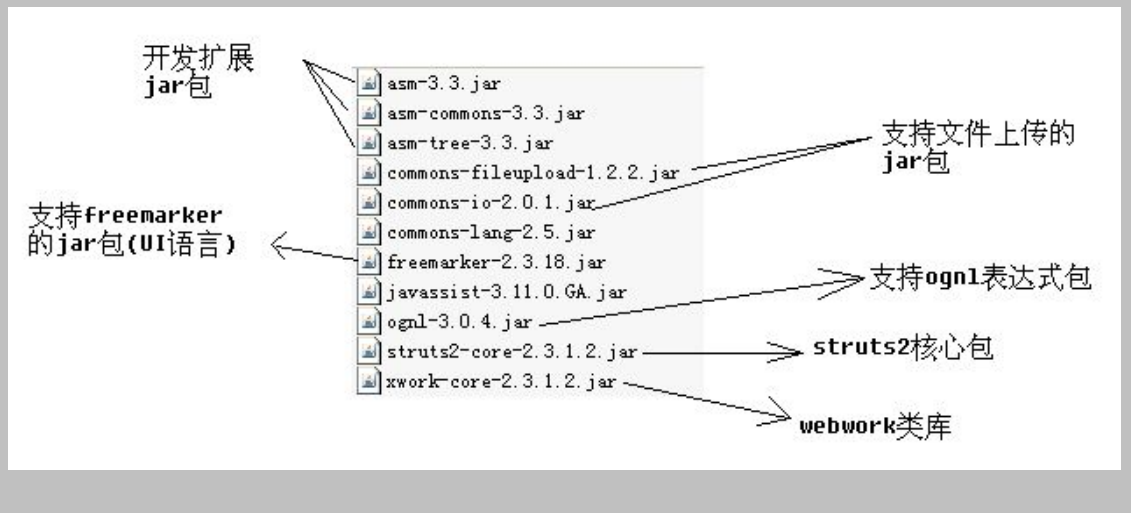
## 4 Struts2 入门

### 4.1 新建一个工程为 `struts2`

### 4.2 修改工程的编码为 `utf-8`

### 4.3 导入 `jar` 包

在新建的工程下创建一个文件夹名为 lib,把相应的 jar 包到入到 lib 文件夹中,并且放入到 classpath 中,jar 包有:



## 4.4 创建 test.jsp 文件

```
测试struts2,输出有命名空间的helloWorld<br>
<a href="${pageContext.request.contextPath}/base/helloWorld.action">helloWorld</a><br>
测试struts2,输出没有命名空间的helloWorld<br>
<a href="${pageContext.request.contextPath}/helloWorld.action">helloWorld</a>
```

## 4.5 创建 HelloWorldAction

```
package cn.itcast.struts2.action;

import com.opensymphony.xwork2.Action;

public class HelloWorldAction implements Action {

    public String execute() throws Exception {
        // TODO Auto-generated method stub
        System.out.println("hello world action");
        return "success";
    }
}
```

## 4.6 编写 success.jsp 文件

```
<body>
    This is my JSP page. <br>
    base 命名空间下的 HelloWorld 运行完成!
</body>
```

## 4.7 编写 struts 配置文件

该文件放在 src 下即可

```
<?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>
    <!--
        name:
        package的名称, 名称是唯一的, 主要作用用于继承
        namespace:
        和url对应:http://localhost:8080/struts2/base/helloWorldAction.action
    -->
    <package name="base" namespace="/base" extends="struts-default">
        <!--
            name:
            每一个action必须对应一个name属性, 和请求的url对应
            class:
            类的完整路径
        -->
        <action name="helloWorldAction"
            class="cn.itcast.struts2.action.HelloWorldAction">
            <!--
                result标签建立result与action返回值之间的关系
                name:
                默认值为success
            -->
            <result name="success">/base/success.jsp</result>
        </action>
    </package>
</struts>
```

```
public String execute() throws Exception {
    // TODO Auto-generated method stub
    System.out.println("hello world action");
    return "success";
}
```

方法的返回值和result标签对应

## 4.8 编写 web.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <filter>
    <filter-name>StrutsPrepareAndExecuteFilter</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>StrutsPrepareAndExecuteFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

## 4.9 运行结果

```
<body>
  测试 struts2,输出有命名空间的 helloWorld
  <br>
  <a
    href="${pageContext.request.contextPath}/base/helloWorldAction.action">helloWorld</a>
  <br>
  测试 struts2,输出没有命名空间的 helloWorld
  <br>
  <a href="${pageContext.request.contextPath}/helloWorldAction.action">helloWorld</a>
</body>
```

## 4.10 加载 struts.xml 过程

在org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter  
类中

```
public void init(FilterConfig filterConfig) throws ServletException {
    InitOperations init = new InitOperations();
    try {
        FilterHostConfig config = new FilterHostConfig(filterConfig);
        init.initLogging(config);
        dispatcher = init.initDispatcher(config);
        init.initStaticContentLoader(config, dispatcher);
    }
```

在org.apache.struts2.dispatcher.Dispatcher中

private void init(Optional<WebConfigurations> webConfigurations)

说明:

- 1、在 web 服务器启动的时候，执行的是过滤器中的 init 方法。[在这里回顾一个问题：一个过滤器中的 init 方法在什么时候执行？总共执行几次？](#)
- 2、在启动服务器的时候，执行了过滤器中的 init 方法，加载了三个配置文件  
struts-default.xml、struts-plugin.xml、struts.xml
- 3、因为这三个文件的 dtd 约束是一样的，所以如果这三个文件有相同的项，后面覆盖前面的。因为先加载前面的，后加载后面的。
- 4、struts.xml 文件必须放在 src 下才能找到。

## 5 Struts2 基本配置

### 5.1 Extends 用法

在struts.xml中

```
<!--
    name:
    package的名称，名称是唯一的，主要作用用于继承
    namespace:
    和url对应:http://localhost:8080/struts2/base/helloWorldAction.action
-->
<package name="base" namespace="/base" extends="struts-default">
    <!--
```

在struts-default.xml中

```
<package name="struts-default" abstract="true">
    <result-types>
```

...

说明：

- 1、上述内容中，因为在 struts.xml 文件中，所有的包都继承了 struts-default 包（在 struts-default.xml 文件中），所以程序员开发的 action 具有 struts-default 包中所有类的功能。
- 2、而 struts-default.xml 文件在 web 服务器启动的时候就加载了。
- 3、在 struts-default.xml 文件中，定义了 struts2 容器的核心内容。
- 4、可以做这样的尝试：把 extends="struts-default" 改为 extends="" 会怎么样呢？

### 5.1.1 例子

在 struts.xml 文件中

```
<package name="test" namespace="/test" extends="base"></package>
```

访问 <http://localhost:8080/struts2/test/helloWorldAction.action> 时也能输出正确的结果，并且命名空间和 base 包中的命名空间没有任何关系了已经。

如果在 struts2 的配置文件中不写 extends="struts-default" 会出现什么样的结构呢？

## 5.2 ActionSupport

在 struts 框架中，准备了一个 ActionSupport 类

代码段一：

```
public class ActionSupport implements Action, Validateable, ValidationAware, TextProvider, LocaleProvider, Serializable {
```

代码段二：

```
/**
```

```
 * A default implementation that does nothing and returns "success".
```

```
 * <p> ActionSupport 是一个默认的动作实现，但是只返回了一个字符串 success
```

```
 * Subclasses should override this method to provide their business logic.
```

```
 * <p>子类需要重新覆盖整个方法，在这个方法中写相应的逻辑
```

```
 * See also {@link com.opensymphony.xwork2.Action#execute()}.
```

```
 *
```

```
 * @return returns {@link #SUCCESS}
```

```
 * @throws Exception can be thrown by subclasses.
```

```
 */
```

```
public String execute() throws Exception {
```

说明：

- 1、 代码段一说明了 ActionSupport 也实现了 Action 接口（以前写的类实现了 Action 接口）
- 2、 代码段二说明如果程序员写自己的 action 继承了 ActionSupport, 需要重新覆盖 execute 方法即可。
- 3、 这个方法默认的返回的是 success;

在配置文件中，还可以这么写：

```
<action name="actionSupportAction">  
    <result name="success">/baseconfig/successActionSupport.jsp</result>  
</action>
```

可以看到 action 标签中没有 class 属性，在 struts-default.xml 中，

```
<default-class-ref class="com.opensymphony.xwork2.ActionSupport" />
```

说明：如果在 action 标签中没有写 class 属性。那么根据上述的配置文件，struts2 会启用 ActionSupport 这个类。所以 action 标签中的 class 属性可以不写。

## 5.3 include

在 struts.xml 中可以按照如下的形式引入别的 struts 的配置文件

```
<include file="cn/itcast/struts2/action/include/struts-include.xml"></include>
```

这样在加载 struts.xml 文件的时候，struts-include.xml 文件也能被加载进来。实例见 Baseconfig/testInclude.jsp 文件

## 5.4 命名空间

在说明 extends 用法的时候，我们引用了这样一个



[url:http://localhost:8080/struts2/base/helloWorld.action](http://localhost:8080/struts2/base/helloWorld.action)。如果我们把 url 改成这样：  
<http://localhost:8080/struts2/base/a/helloWorld.action>。行吗？答案是可以的。再改成这样的  
[url:http://localhost:8080/struts2/base/a/b/helloWorld.action](http://localhost:8080/struts2/base/a/b/helloWorld.action) ,行吗？答案也是可以的。如果这样呢 <http://localhost:8080/struts2/helloWorld.action> 可以吗？这样就不行了。为什么？

步骤：

- 1、struts2 会在当前的命名空间下查找相应的 action
- 2、如果找不到，则会在上级目录中查找，一直查找到根目录
- 3、如果根目录也找不到，报错。
- 4、如果直接访问的就是根目录，找不到，这样的情况就会直接报错。不会再去子目录中进行查找。

## 6 结果类型

### 6.1 说明

- 1、每个 action 方法都返回一个 String 类型的值，struts 一次请求返回什么值是由这个值确定的。
- 2、在配置文件中，每一个 action 元素的配置都必须有 result 元素，每一个 result 对应一个 action 的返回值。
- 3、Result 有两个属性：  
name:结果的名字，和 action 中的返回值一样，默认值为 success;  
type:响应结果类型，默认值为 dispatcher.

### 6.2 类型列表

在 struts-default.xml 文件中，如下面所示：

```
<result-types>
  <result-type                                name="chain"
class="com.opensymphony.xwork2.ActionChainResult"/>
  <result-type                                name="dispatcher"
class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true"/>
  <result-type                                name="freemarker"
class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
  <result-type                                name="httpheader"
class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
  <result-type                                name="redirect"
class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
  <result-type                                name="redirectAction"
class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
  <result-type                                name="stream"
class="org.apache.struts2.dispatcher.StreamResult"/>
  <result-type                                name="velocity"
class="org.apache.struts2.dispatcher.VelocityResult"/>
  <result-type                                name="xslt"
class="org.apache.struts2.views.xslt.XSLTResult"/>
  <result-type                                name="plainText"
```

说明：

- 1、从上述可以看出总共 10 种类型
- 2、默认类型为 `ServletDispatcherResult` 即转发。
- 3、结果类型可以是这 10 种结果类型的任何一种。

## 6.2.1 Dispatcher 类型

### 6.2.1.1 说明

Dispatcher 类型是最常用的结果类型，也是 struts 框架默认的结果类型。

### 6.2.1.2 例子

页面参照：resulttype/testDispatcher.jsp

Action 参照：DispatcherAction

配置文件：struts-resulttype.xml

在配置文件中，可以有两种写法：

第一种写法：

```
<result name="success">/resulttype/successDispatcher.jsp</result>
```

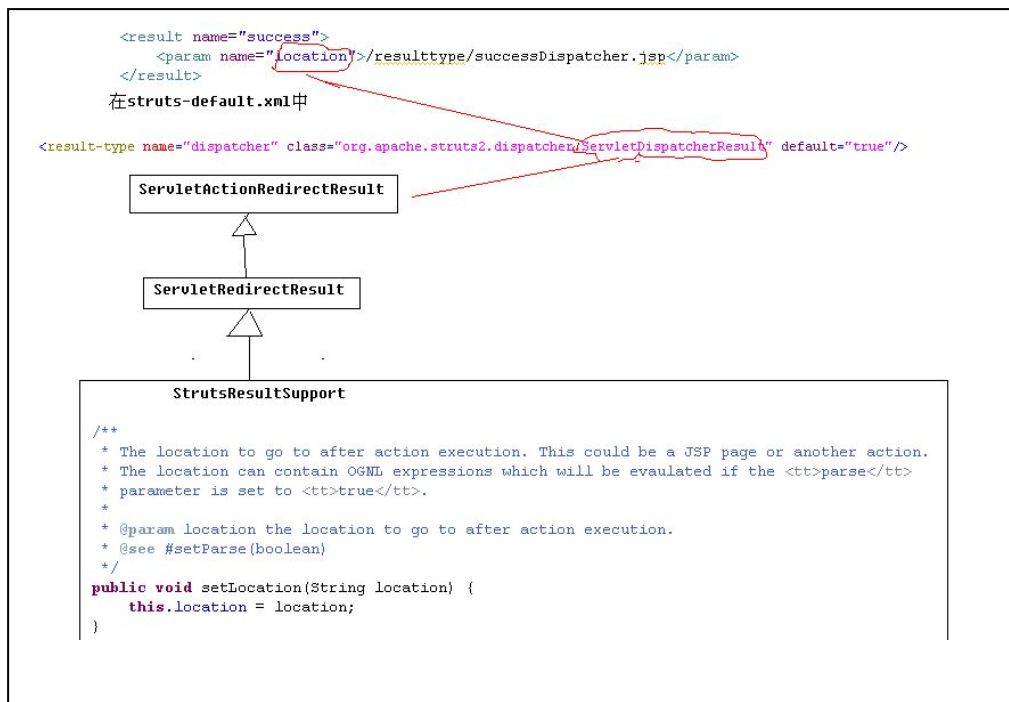
第二种写法：

```
<result name="success">
```

```
    <param name="location">/resulttype/successDispatcher.jsp</param>
```

```
</result>
```

下面的图说明了 location 的来历:



## 6.2.2 Redirect 类型

### 6.2.2.1说明

Redirect 属于重定向。如果用 redirect 类型，则在 request 作用域的值不能传递到前台。

### 6.2.2.2例子

页面: resulttype/testRedirect.jsp

Action: RedirectAction

配置文件: struts-resulttype.xml

## 6.2.3 redirectAction 类型

### 6.2.3.1 说明

- 1、把结果类型重新定向到 action
- 2、可以接受两种参数
  - a) actionName: action 的名字
  - b) namespace:命名空间

### 6.2.3.2 例子

第一种方式:

```
<result name="success" type="redirectAction">resulttype/redirectactionAction.action</result>
```

第二种方式:

```
<result name="success" type="redirectAction">
```

```
  <!--
```

```
    actionName:
```

```
      请求的 action 的路径
```

```
    namespace:
```

```
      如果不写，默认就是请求的 action 的路径，如果写，路径将被重新赋值
```

```
  -->
```

```
  <param name="actionName">
```

```
    resulttype/redirectactionAction.action
```

```
  </param>
```

```
</result>
```

## 7 Action 原型模式

### 7.1 回顾 servlet

在 servlet 中，一个 servlet 只有一个对象，也就是说 servlet 是单例模式。如果把一个集合写在 servlet 属性中，则要考虑线程安全的问题。

## 7.2 Action 多例模式

但是在 struts2 的框架中，并不存在这种情况，也就是说 struts2 中的 action，只要访问一次就要实例化一个对象。这样不会存在数据共享的问题。这也是 struts2 框架的一个好处。

## 7.3 实验

可以写一个类，如下：

```
package cn.itcast.struts2.action.moreinstance;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class MoreInstanceAction extends ActionSupport {
    public MoreInstanceAction() {
        System.out.println("create new action");
    }
    public String execute() {
        System.out.println("more instance action");
        return SUCCESS;
    }
}
```

```
<struts>
  <package name="moreinstance" namespace="/moreinstance">
    <action name="moreinstanceAction"
      class="cn.itcast.struts2.action.moreinstance.MoreInstanceAction">
    </action>
  </package>
</struts>
```

## 8 通配符

### 8.1 Execute 方法的弊端

假设有这样的需求：

- 1、有一个 action 为 PersonAction。
- 2、在 PersonAction 中要实现增、删、改、查四个方法。
- 3、但是在 action 中方法的入口只有一个 execute 方法
- 4、所以要想完成这样的功能，有一种方法就是在 url 链接中加参数。

那么在 action 中的代码可能是这样的：

```
PatternAction
public class PatternAction extends ActionSupport{
    private String method;
    public String execute(){
        if(method.equals("add")){
            //增加操作
        }else if(method.equals("update")){
            //修改操作
        }else if(method.equals("delete")){
            //删除操作
        }else{
            //查询操作
        }
        return "";
    }
}
```

可以看出这样写结构并不是很好。而通配符的出现解决了这个问题。

## 8.2 method 属性

```
Pattern.jsp
访问 PersonAction 的 add 方法:<br>
    <a href="${pageContext.request.contextPath}/pattern/patternAction.action"> 测
试</a><br>
Struts-pattern.xml
<package name="pattern" namespace="/pattern"
    extends="struts-default">
    <action name="patternAction" method="add"
        class="cn.itcast.struts2.action.pattern.PatternAction">
    </action>
</package>
PatternAction
public String add(){
    return "add";
}
```

说明：从上述代码可以看出只要配置文件中的 `method` 属性的值和方法名称一样就可以了。但是这种写法有弊端。如果 `action` 中需要 5 个方法。则在 `struts` 的配置文件中需要写 5 个配置，这样会导致配置文件的篇幅很大。而且重复性也很大

## 8.3 动态调用方法

在 url 中通过 `action` 名称!方法名称可以动态调用方法。

```
Pattern.jsp
动态调用 PatternAction 中的 add 方法:<br>
<a
    href="${pageContext.request.contextPath}/pattern/patternAction!add.ac
tion">测试</a>
struts-pattern.xml
<action name="patternAction"
    class="cn.itcast.struts2.action.pattern.PatternAction">
</action>
说明：这样的情况在配置文件中不需要 method 属性
```

## 8.4 通配符映射

### 8.4.1 映射一

需求: a\_add.action、b\_add.action、c\_add.action 全部请求 PatternAction 的 add 方法  
Pattern.jsp  
通配符映射实例 1:<br>  
<a href="{pageContext.request.contextPath}/pattern/a\_add.action">测试</a>  
<a href="{pageContext.request.contextPath}/pattern/b\_add.action">测试</a>  
<a href="{pageContext.request.contextPath}/pattern/c\_add.action">测试</a>  
说明: 不管是 a\_add 还是 b\_add 还是 c\_add 的路径都指向 PatternAction 的 add 方法。  
struts-pattern.xml  
<action name="a\_add" method="add"  
    class="cn.itcast.struts2.action.pattern.PatternAction">  
</action>  
<action name="b\_add" method="add"  
    class="cn.itcast.struts2.action.pattern.PatternAction">  
</action>  
<action name="c\_add" method="add"  
    class="cn.itcast.struts2.action.pattern.PatternAction">  
</action>  
上述结构是很差的, 经过改进如下:  
<action name="\*\_add" method="add"  
    class="cn.itcast.struts2.action.pattern.PatternAction">

### 8.4.2 映射二

请求 PersonAction 和 StudentAction 的 add 方法  
Pattern.jsp  
通配符映射实例 2:  
<br>  
<a  
    href="{pageContext.request.contextPath}/pattern/personAction\_add.action">  
请求 personAction 的 add 方法</a>  
<a  
    href="{pageContext.request.contextPath}/pattern/studentAction\_add.action"  
>请求 studentAction 的 add 方法</a>



Struts-pattern.xml

```
<action name="personAction_add" method="add"
class="cn.itcast.struts2.action.pattern.PersonAction"></action>
```

```
<action name="studentAction_add" method="add"
class="cn.itcast.struts2.action.pattern.StudentAction"></action>
```

改进如下：

```
<action name="*_add" method="add" class="
cn.itcast.struts2.action.pattern.{1}">
```

说明：\*和{1}是相对应的关系。

### 8.4.3 映射三

需求：在 TeacherAction 中有增、删、改、查的方法。这个时候配置文件怎么写比较简单？

Pattern.jsp

通配符映射实例 3:

```
<a
```

```
href="{pageContext.request.contextPath}/pattern/PersonAction_add.action"> 请 求
teacherAction 的 add 方法</a>
```

```
<a
```

```
href="{pageContext.request.contextPath}/pattern/StudentAction_update.action"> 请 求
teacherAction 的 update 方法</a>
```

```
<a
```

```
href="{pageContext.request.contextPath}/pattern/StudentAction_delete.action"> 请 求
teacherAction 的 delete 方法</a>
```

```
<a
```

```
href="{pageContext.request.contextPath}/pattern/StudentAction_query.action"> 请 求
teacherAction 的 query 方法</a>
```

struts-pattern.xml

```
<action name="teacherAction_*" method="{1}"
```

```
class="cn.itcast.struts2.action.pattern.TeacherAction">
```

```
</action>
```

说明：\*和 method 的属性值保持一致。

延伸：

```
<action name="*_*" method="{2}"
```

```
class="cn.itcast.struts2.action.pattern.{1}">
```

```
</action>
```

第一个\*匹配{1}，第二个\*匹配{2}

## 9 全局结果类型

### 9.1 说明

当很多提交请求跳转到相同的页面，这个时候，这个页面就可以成为全局的页面。在 struts2 中提供了全局页面的配置方法。

### 9.2 例子

Struts-pattern.xml

```
<global-results>
```

```
    <result name="success">success.jsp</result>
```

```
</global-results>
```

注意：

- \* 这个配置必须写在 action 配置的上边。dtd 约束的规定。
- \* 如果在 action 的 result 中的 name 属性也有 success 值，顺序为先局部后全局。

### 9.3 错误的统一处理

#### 9.3.1 xml 文件

```
<package name="struts-global" namespace="/" extends="json-default">
    <global-results>
        <result name="errHandler" type="chain">
            <param name="actionName">errorProcessor</param>
        </result>
    </global-results>
    <global-exception-mappings>
        <exception-mapping exception="java.lang.Exception"
            result="errHandler" />
    </global-exception-mappings>

    <action name="errorProcessor" class="cn.itheima01.oa.exception.ErrorProcessor">
        <result></result>
    </action>
</package>
```

## 9.3.2 Java 文件

```
public class ErrorProcessor extends ActionSupport {
    private String ss = "";
    public String getSs() {
        return ss;
    }
    public void setSs(String ss) {
        this.ss = ss;
    }
    private Exception exception;
    @JSON(serialize=false)
    public Exception getException() {
        return exception;
    }
    public void setException(Exception exception) {
        this.exception = exception;
    }
    @Override
    public String execute(){
        this.ss = this.exception.getMessage();
        return SUCCESS;
    }
}
```

# 10 Struts2 与 servlet 接口

## 10.1 说明

通过前面的练习大家都知道,在 action 的方法中与 servlet 的所有的 API 是没有任何关系的。所以在 struts2 中做到了 action 与 servlet 的松耦合,这点是非常强大的。但是如果没有 HttpServletRequest,HttpServletResponse,ServletContext 有些功能是没有办法完成的。比如购物车程序,需要把购买的物品放入 session 中。所以就得找一些路径使得在 struts2 中和 servlet 的 API 相结合。

## 10.2 实现一

Struts2 中提供了 ServletActionContext 类访问 servlet 的 api。

```
Servlet.jsp
通过 ServletActionContext 类访问 servlet 的 API: <br>
<a
href="{pageContext.request.contextPath}/servlet/servletAction_testServletAPI.action">测试 struts2 中访问 servletAPI</a>
ServletAction
```

## 10.3 实现二

```
ServletAction
public class ServletAction extends ActionSupport implements ServletContextAware,
    SessionAware,ServletRequestAware{
    private HttpServletRequest request;
    private Map sessionMap;
    private ServletContext servletContext;

    public String testServletAPI2(){
        System.out.println(this.servletContext);
        System.out.println(this.sessionMap);
        System.out.println(this.request);
        return "";
    }

    public void setServletContext(ServletContext context) {
        // TODO Auto-generated method stub
        this.servletContext = context;
    }
    public void setSession(Map<String, Object> session) {
        // TODO Auto-generated method stub
        this.sessionMap = session;
    }
    public void setServletRequest(HttpServletRequest request) {
        // TODO Auto-generated method stub
        this.request = request;
    }
}
```

# 11 拦截器

假设有一种场景：

在 action 的一个方法中，要进行权限的控制。如果是 admin 用户登入，就执行该方法，如果不是 admin 用户登入，就不能执行该方法。

## 11.1 实现方案一

```
AccessAction
public String testAccess(){
    if(this.username.equals("admin")){
        //执行业务逻辑方法
        return SUCCESS;
    }else{
        //不执行
        return "failed";
    }
}
```

说明：

- 5、这样做，程序的结构不是很好。原因是权限的判断和业务逻辑的方法紧密耦合在了一起。如果权限的判断很复杂或者是业务逻辑很复杂会造成后期维护的非常困难。所以结构不是很好
- 6、这种形式只能控制一个 action 中的一个方法。如果很多 action 中的很多方法都需要这种控制。会导致大量的重复代码的编写。

## 11.2 实现方案二

动态代理可以实现。请参见 cn.itcast.struts.jdkproxy 包下的类。

## 11.3 实现方案三

在 struts2 中，用拦截器(interceptor)完美的实现了这一需求。在 struts2 中，内置了很多拦截器，在 struts-default.xml 文件中可以看出。用户还可以自定义

自己的拦截器。自定义拦截器需要以下几点：

1、在配置文件中：

包括两个部分：声明拦截器栈和使用拦截器栈

```
struts-interceptor.xml
<!--
    声明拦截器
-->
<interceptors>
    <!--
        定义自己的拦截器
    -->
    <interceptor name="accessInterceptor"
        class="cn.itcast.struts2.action.interceptor.PrivilegeInterceptor">
    </interceptor>
    <!--
        声明拦截器栈
    -->
    <interceptor-stack name="accessInterceptorStack">
        <!--
            引用自定义的拦截器
        -->
        <interceptor-ref name="accessInterceptor"></interceptor-ref>
        <!--
            引用 struts2 内部的拦截器栈
        -->
        <interceptor-ref name="defaultStack"></interceptor-ref>
    </interceptor-stack>
```

一个类如果是拦截器，必须实现 `Interceptor` 接口。

```
public class PrivilegeInterceptor implements Interceptor{

    public void destroy() {
        // TODO Auto-generated method stub

    }

    public void init() {
        // TODO Auto-generated method stub

    }

    public String intercept(ActionInvocation invocation) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("aaaa");
        //得到当前正在访问的 action
        System.out.println(invocation.getAction().toString());
        //得到 Ognl 值栈
        System.out.println(invocation.getInvocationContext().getValueStack());
        //请求路径 action 的名称，包括方法
        System.out.println(invocation.getProxy().getActionName());
        //方法名称
        System.out.println(invocation.getProxy().getMethod());
        //命名空间
        System.out.println(invocation.getProxy().getNamespace());
        String method = invocation.invoke();
        return null;
    }

}
```

说明：

- 1、这个类中 `init`、`intercept` 和 `destroy` 三个方法说明了一个拦截器的生命周期。
- 2、在 `interceptor` 方法中的参数 `invocation` 是执行 `action` 的上下文，可以从这里得到正在访问的 `action`、`Ognl` 值栈、请求路径、方法名称、命名空间等信息。以帮助程序员在拦截器中做相应的处理工作。
- 3、红色部分是关键部分，就是调用 `action` 的方法。这里可以成为目标类的

# 12 验证

## 12.1 需求

验证用户名不能为空，密码也不能为空，并且长度不能小于 6 位数。

## 12.2 基本验证

```
ValidateAction.java
public class ValidateAction extends ActionSupport{
    private String username;
    private String password;
    //set 和 get 方法
    public String testValidate(){
        return "success";
    }
    public String aaa(){
        return "success";
    }
    public void validate() {
        if(this.username.equals("")){
            this.addFieldError("username", "用户名不能为空");
        }
        if(this.password.equals("")){
            this.addFieldError("password", "密码不能为空");
        }else if(this.password.length()<6){
            this.addFieldError("password", "密码不能小于 6 个字符");
        }
    }
}
```

说明：

- 在 ActionSupport 类中有这样的描述：

```
/**
 * A default implementation that validates nothing.
 * Subclasses should override this method to provide validations.
 */
public void validate() {
}
```

这是一个默认的实现，子类应该覆盖整个方法去完成验证逻辑。



所以我们只需要去重写 validate 这个方法就可以了。

- 从下面的图中可以看出错误信息的封装机制。

```
    this.addFieldError("username", "用户名不能为空");  
  
validationAware.addFieldError(fieldName, errorMessage);  
  
public synchronized void addFieldError(String fieldName, String errorMessage) {  
    final Map<String, List<String>> errors = internalGetFieldErrors();  
    List<String> thisFieldErrors = errors.get(fieldName);  
  
    if (thisFieldErrors == null) {  
        thisFieldErrors = new ArrayList<String>();  
        errors.put(fieldName, thisFieldErrors);  
    }  
  
    thisFieldErrors.add(errorMessage);  
}
```

总的结构为 Map<String,List<String>>

String 为要验证的字段

List<String> 封装错误信息

- 错误信息会出现在 input 指向的页面。
- validate 是针对 action 中所有的方法进行验证。如果想针对某一个方法进行验证，应该把 validate 方法改为 validate 方法名称(方法名称的第一个字母大写)。

## 12.3 框架验证

- 1、编写 xml 文件，名字的输出规则为:

<ActionClassName>-<aliasName\_methodName>-validation.xml

其中 alias 为配置文件中 action 元素 name 属性的值。

- 2、在该文件中填写需要校验的内容

# 13 国际化

## 13.1 说明

一个系统的国际化就是根据操作系统的语言，页面上的表现形式发生相应的变化。比如如果操作系统是英文，页面的文字应该用英语，如果操作系统是中文，页面的语言应该是中文。

## 13.2 步骤

### 13.2.1 建立资源文件

资源文件的命名规则：

默认的命名为：

文件名前缀.properties

根据语言的命名为：

文件名前缀.语言种类.properties

例如：

中文：

resource\_zh\_CN.properties

内容：

item.username=用户名

item.password=密码

英文：

resource\_en\_US.properties

内容：

item.username=username\_en

item.password=password\_en

默认：

resource.properties

内容：

item.username=username

item.password=password

### 13.2.2 配置文件中

需要在配置文件中加入：

```
<constant                                name="struts.custom.i18n.resources"
value="cn.itcast.struts2.action.i18n.resource">
</constant>
```

说明：

- 1、这样 struts2 就会去找你写的资源文件
- 2、name 的值可以在 org/apache/struts2/default.properties 中找到。
- 3、如果放置的资源文件在 src 下，则 value 的值可以直接写，如果在包中则可以写成包名.resource。

4、在这里 resource 是个基名,也就是说可以加载以 resource 开头的文件。

### 13.2.3 页面中

利用<s:text name="item.username"/>就可以把资源文件中的内容输出来。

```
I18n/login.jsp
<s:form action="i18n/loginAction_login.action" method="post">
    <table border="1">
        <tr>
            <td><s:text name="item.username"/></td>
            <td><s:textfield name="username"/></td>
        </tr>
        <tr>
            <td><s:text name="item.password"/></td>
            <td><s:password name="password"/></td>
        </tr>
        <tr>
            <td><s:submit
                                name="submit"
value="%{getText('item.username')}" /></td>
            <td></td>
        </tr>
    </table>
</s:form>
```

说明:

- 1、标红色部分的是要从资源文件中找的内容。item.username 和 item.password 代码 key 的值。
- 2、也可以利用 %{getText('item.username')} 方式来获取资源。采取的是 OGNL 表达式的方式。
- 3、getText 的来源:

从源代码可以看出 ActionSupport 实现了 TextProvider 接口。  
Provides access to {@link ResourceBundle}s and their underlying text messages. 意思是说提供了访问资源文件的入口。而 TextProvider 中提供了 getText 方法, 根据 key 可以得到 value。

## 13.2.4 在 action 中

可以利用 `ActionSupport` 中的 `getText()` 方法获取资源文件的 `value` 值。

```
118n/LoginAction
public class LoginAction extends ActionSupport{
    public String login(){
        String username = this.getText("item.username");
        System.out.println(username);
        String password = this.getText("item.password");
        System.out.println(password);
        return "";
    }
}
```

说明：通过 `this.getText()` 方法可以获取资源文件的值。

# 14 OGNL

## 14.1 介绍

OGNL 表达式是(Object-Graph Navigation Language)是对象图形化导航语言。OGNL 是一个开源的项目，struts2 中默认使用 OGNL 表达式语言来显示数据。与 `serlvet` 中的 `el` 表达式的作用是一样的。OGNL 表达式有下面以下特点：

- 1、支持对象方法调用，例如：`objName.methodName()`;
- 2、支持类静态的方法调用和值访问，表达式的格式为

`@[类全名（包括包路径）]`

`@[方法名 | 值名]`

例如：

`@java.lang.String@format('foo%s','bar')`

`@tutorial.MyConstant@APP_NAME;`

- 3、支持赋值操作和表达式串联，例如：

`price=100, discount=0.8, calculatePrice()`，这个表达式会返回 80;

- 4、访问 OGNL 上下文（OGNL context）和 `ActionContext`

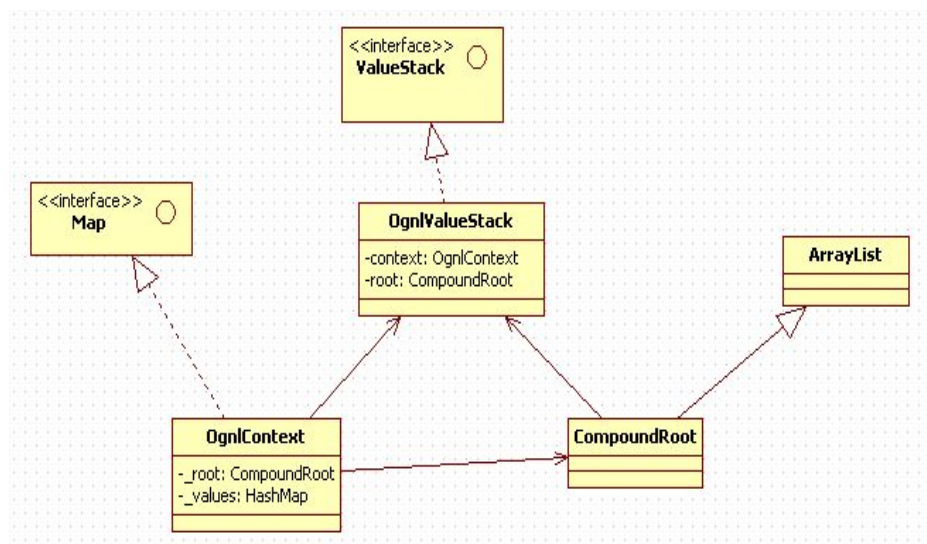
## 5、操作集合对象

### 14.2 回顾 el 表达式

在 servlet 中学习的 el 表达式实际上有两步操作：

- 1、把需要表现出来的数据放入到相应的作用域中  
(req,res,session,application)。
- 2、利用 el 表达式把作用域的值表现在页面上

### 14.3 ognl 类图

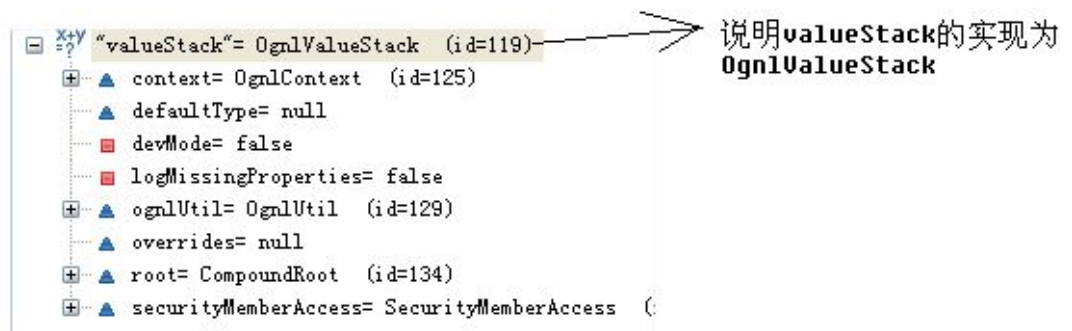


### 14.4 ValueStack

#### 14.4.1 说明

- 1、ValueStack 是一个接口，在 struts2 中使用 OGNL 表达式实际上是使用实现了 ValueStack 接口的类 OgnlValueStack,这个类是 OgnlValueStack 的基础。
- 2、ValueStack 贯穿整个 action 的生命周期。每一个 action 实例都拥有一个 ValueStack 对象。其中保存了当前 action 对象和其他相关对象。
- 3、Struts2 把 ValueStack 对象保存中名为 struts.valueStack 的 request 域中。

## 14.4.2 ValueStack 内存图



## 14.4.3 ValueStack 的组织结构



从图上可以看出 OgnlValueStack 和我们有关的内容有两部分：即 OgnlContext 和 CompoundRoot。所以把这两部分搞清楚很重要。

## 14.4.4 OgnlContext 组织结构

```
public class OgnlContext implements Map
{
    private Object _root;
    private Map _values = new HashMap(23);
}
```

#### 14.4.4.1 \_values

从上述可以看出，OgnlContext 实际上有一部分功能是 Map。所以可以看出 \_values 就是一个 Map 属性。而运行一下下面的代码就可以看到：

```
//在 request 域中设置一个参数
ServletActionContext.getRequest().setAttribute("req_username","req_username");
//在 request 域中设置一个参数
ServletActionContext.getRequest().setAttribute("req_psw", "req_psw");
//在 session 域中设置一个参数
ActionContext.getContext().getSession().put("session_username",
"session_username");
//在 session 域中设置一个参数
ActionContext.getContext().getSession().put("session_psw",
"session_psw");
//获取 OGNL 值栈
ValueStack valueStack = ActionContext.getContext().getValueStack();
```

在 \_values 的 map 中：

key	value
application	ApplicationMap
request	RequestMap
action	自己写的 action
com.opensymphony.xwork2.ActionContext.session	SessionMap

而 request 中的值为：

```
{
    req_psw=req_psw,
    req_username=req_username,
    __cleanup_recursion_counter=1,
    struts.valueStack=com.opensymphony.xwork2.ognl.OgnlValueStack@3d58b2,
    struts.actionMapping=org.apache.struts2.dispatcher.mapper.ActionMapping@1f713ed
}
```

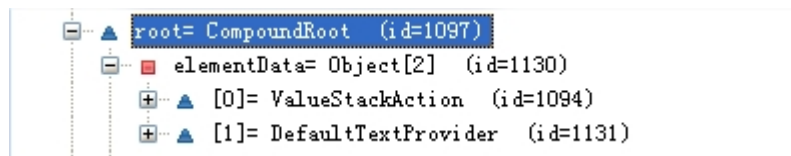
可以看出在程序中存放在 request 作用域的值被放入到了 \_values 的 request 域中。

而  
com.opensymphony.xwork2.ActionContext.session 的  
值为:

```
{session_username=session_username,  
session_psw=session_psw}
```

从上图中可以看出在程序中被加入到 session 的值在 \_values 中也体现出来。

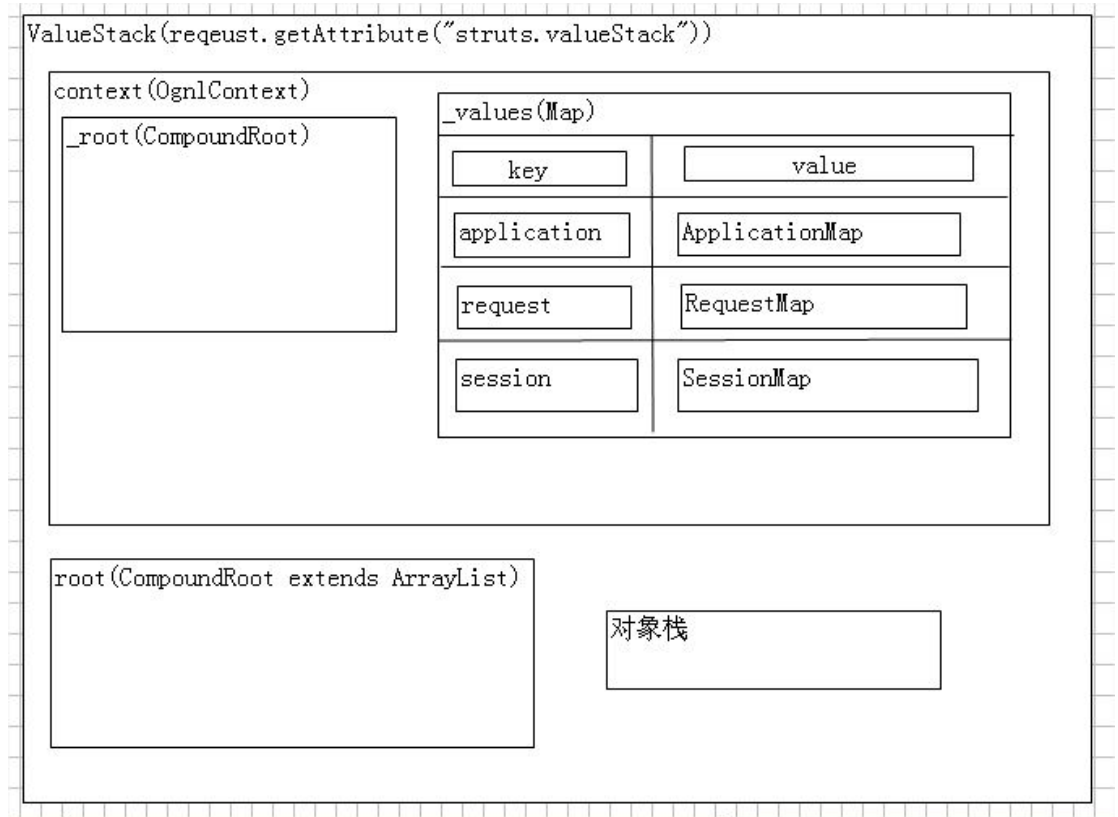
#### 14.4.4.2 \_root



从图中可以看出 \_root 实际上是 CompoundRoot 类，从类的组织结构图中可以看出，这个类实际上是继承了 ArrayList 类，也就是说这个类具有集合的功能。而且在默认情况下，集合类的第一个为 ValueStackAction，也就是我们自己写的 action。



## 14.5 总图



说明：

- 1、上图是 ognl 完整的数据结构图，可以清晰得看出数据的组成。
- 2、Context 中的 `_root` 和 ValueStack 中的 `root`(对象栈)里的数据结构和值是一样的。
- 3、这就意味着我们只需要操作 `OgnlContext` 就可以完成对数据的存和取的操作。
- 4、ValueStack 内部有两个逻辑的组成部分：
  - a) ObjectStack  
Struts 会把动作和相关的对象压入到 ObjectStack 中。
  - b) ContextMap  
Struts 会把一些映射关系压入到 ContextMap 中

## 14.6 存数据

### 14.6.1 Map 中存数据

#### 14.6.1.1 方法 1

```
//向 map 中存放数据
ServletContext.getRequest().setAttribute("req_username","req_username");
ServletContext.getRequest().setAttribute("req_psw", "req_psw");
ActionContext.getContext().getSession().put("session_username", "session_username");
ActionContext.getContext().getSession().put("session_psw", "session_psw");
```

上面的代码都是往 ContextMap 中存放数据。因为这些值都是具有映射关系的。

#### 14.6.1.2 方法 2

```
ActionContext.getContext().put("msg", "msg_object");
```

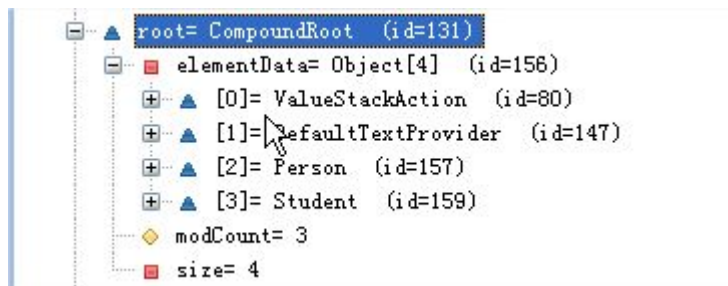
通过执行上述代码把”msg”和”msg\_object”放入到了 ContextMap 中。

## 14.6.2 值栈中存数据

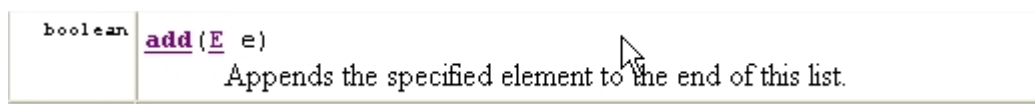
### 14.6.2.1 方法 1

```
/*
 * 把对象放入到值栈中
 */
//方法一：先得到 root，把一个对象压入到 root 中
ValueStack valueStack = ActionContext.getContext().getValueStack();
valueStack.getRoot().add(new Person());
valueStack.getRoot().add(new Student());
```

运行以后的值栈结构图：



从内存图中可以看出最后被压入到 list 中的对象在最下面。



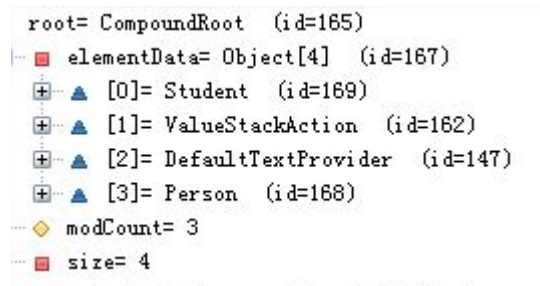
图为 ArrayList 中的 add 方法的解释：

追加的指定的元素放到集合的最下面。

### 14.6.2.2 方法 2

```
/*
 * 方法二：先得到 root，利用 add(index,Object)把一个对象压入到 root 中
 * 这里 index 指的是集合中的位置
 */
ValueStack valueStack = ActionContext.getContext().getValueStack();
valueStack.getRoot().add(new Person());
valueStack.getRoot().add(0, new Student());
```

运行后的结构图为：



从图中可以很明显看出新创建的 Student 对象被放到了第一个位置，因为 Index 的值为 0，所以是第一个位置。

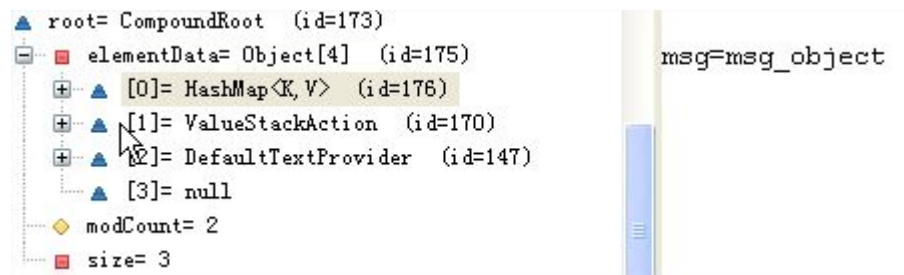
### 14.6.2.3 方法 3

把一个键值对存放在对象栈中，做法为：

```
/*
 * 方法三：
 *      把一个键值对存放在对象栈中
 */

ValueStack valueStack = ActionContext.getContext().getValueStack();
valueStack.set("msg","msg_object");
```

对象栈图为：



从图中可以看出上面的代码执行过程为：

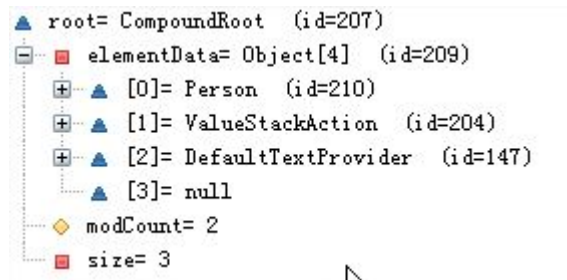
- 1、先把”msg”和”msg\_object”两个字符串封装成 Map
- 2、再把封装以后的 Map 放入到对象栈中。

### 14.6.2.4 方法 4

```
/*
 * 方法 4
 *      利用 ValueStack 的 push 方法可以把一个对象直接压入对象栈的第一个位置
 */

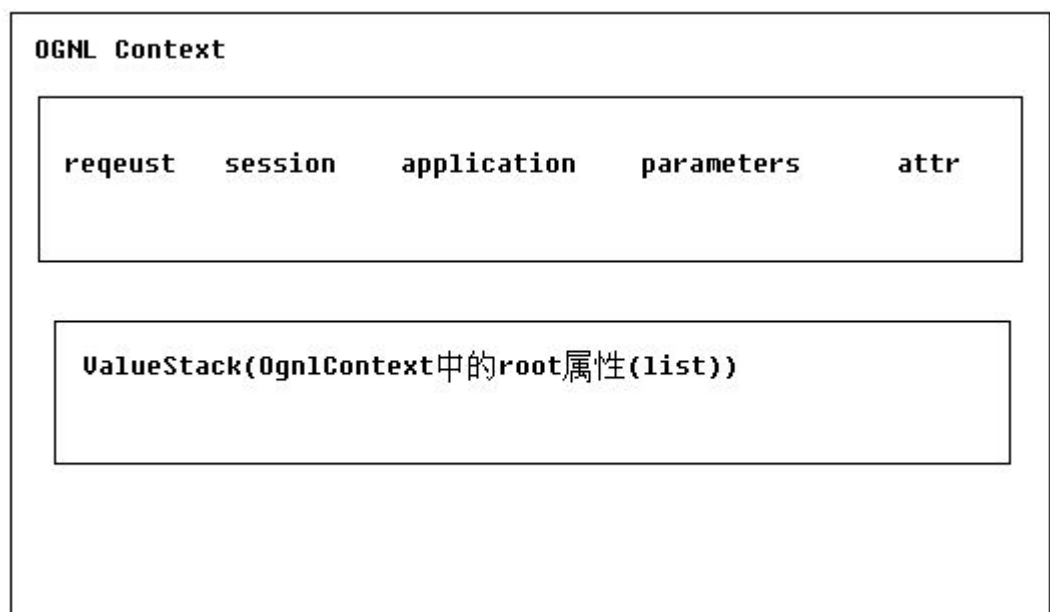
ValueStack valueStack = ActionContext.getContext().getValueStack();
valueStack.push(new Person());
```

执行完 push 方法以后，对象栈的情况如下：



Push 方法把新创建的 Person 对象放入到了对象栈的首个位置。

## 14.7 OGNL Context



### 14.7.1 说明

- 1、上图为 OGNL Context 的结构图
- 2、当 struts2 接受一个请求时，会迅速创建 `ActionContext`, `ValueStack`, `action`。然后把 `action` 压入到值栈中。所以 `action` 的实例变量可以被 ognl 访问。所以利用 ognl 表达式可以访问 `action`。

## 14.8 ActionContext

```
/*
 * ActionContext 的作用
 */
ActionContext.getContext().getSession().put("session_username", "session_username");
ActionContext.getContext().getSession().put("session_psw", "session_psw");
ValueStack valueStack = ActionContext.getContext().getValueStack();
```

### 14.8.1 说明

从上面的代码中可以看出，struts2 中的 ActionContext 的作用是提供了对 ognl 数据的操作。并且可以通过 ActionContext 获取到经过 struts2 封装了的 session 等参数。

## 14.9 ServletActionContext

```
/*
 * 返回 servlet 中的 request 和 servletcontext
 */
ServletActionContext.getRequest().setAttribute("req_username", "req_username");
ServletActionContext.getRequest().setAttribute("req_psw", "req_psw");
ServletActionContext.getServletContext();
//得到 ActionContext
ServletActionContext.getContext();
```

### 14.9.1 说明

- 1、 可以通过 ServletActionContext 得到 servlet 中的一些类，比如 HttpServletRequest, ServletContext 等
- 2、 可以通过 ServletActionContext 返回 ActionContext

## 14.10 Ognl 表达式

### 14.10.1 简述

从 9.6 到 9.9 讨论了 ognl 的结构、如何存数据。9.10 重点讨论如何把 ognl 结构中的数据呈现在页面上。所以 Ognl 表达式的作用就是把 OgnlContext 中的数据输出到页面上。

### 14.10.2 el 表达式例子

参照课堂演示例子

### 14.10.3 ognl 表达式例子

#### 14.10.3.1 用法 1(#号用法)

说明：

- 1、访问 OGNL 上下文和 action 上下文，#相当于 `ActionContext.getContext()`;
- 2、如果访问的是 map 中的值而不是对象栈中的值，由于 map 中的数据不是根对象，所以在访问时需要添加#前缀。

名称	作用	例子
parameters	包含当前 HTTP 请求的 Map	<code>#parameters.id[0]=request.getParameter("id")</code>
request	包含当前 <code>HttpServletRequest</code> 属性的 Map	<code>#request.username=request.getAttribute("username");</code>
session	包含当前 <code>HttpSession</code> 属性的 Map	<code>#session.username=session.getAttribute("username");</code>
application	包含当前 <code>ServletContext</code> 属性的 Map	<code>#application.username=application.getAttribute("username");</code>
attr	用于按照 <code>request&gt;session&gt;application</code> 顺序访问其属性	<code>#attr.username</code> 相当于按照顺序在以上三个范围内读取 username 的属性，直到找到为止。

注：也可以写为 `#request['username']` `#session['username']`  
`#application['username']`

主要步骤:

在 action 中

```
public String testScope() {
    // 把字符串"request_msg"放入到 request 域中

    ServletActionContext.getRequest().setAttribute("request_username",
        "request_username");
    // 把字符串"session_msg"放入到 session 域中
    ServletActionContext.getRequest().getSession().setAttribute(
        "session_username", "session_username");
    // 把字符串"application_msg"放入到 application 域中
    ServletActionContext.getServletContext().setAttribute(
        "application_username", "application_username");
    return "ognl_scope";
}
```

在页面中:

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="/struts-tags" prefix="s"%> //导入 struts2 的标签
ognl 表达式关于 scope(request,session,application)的输出<br>
request:<s:property value="#request.request_username"/><br>
session:<s:property value="#session.session_username"/><br>
application:<s:property value="#application.application_username"/><br>
```

### 14.10.3.2 用法 2

OGNL 会设定一个对象(root 对象), 在 struts2 中根对象就是 CompoundRoot, 或者为 OgnlValueStack 中的 root, 通常被叫做 ValueStack(值栈或者对象栈), 如果要访问根对象的属性, 则可以省略去#,直接访问对象的属性即可。

在 action 中

```
public String testObjectValue(){
    //得到 ognl 的上下文
    ValueStack valueStack =
    ActionContext.getContext().getValueStack();
    valueStack.set("msg", "object_msg");
    return "ognl_objectstack";
}
```



在页面中：

```
对象栈的内容：<s:property value="msg"/>
//把对象栈中 key 值为 msg 的 value 的值输出
```

问题：如果把一个字符串 push 进对象栈，怎么样取？  
在 14.11.1 中将解决这个问题。

### 14.10.3.3 用法 3(深入理解对象栈)

有三个类：Person.java, Student.java, OgnlAction.java

Person 类

```
package cn.itcast.struts2.valuestack.bean;
public class Person {
    private Integer pid;
    private String pname;
    private String comment;
}
```

Student 类

```
package cn.itcast.struts2.valuestack.bean;
public class Student {
    private Integer sid;
    private String sname;
    private String comment;
}
```

OgnlAction 类

```
public class OgnlAction extends ActionSupport {
    private String comment="55";
    private String id = "3";
}
```

把 Person 类和 Student 类创建出对象，然后放入到对象栈中，代码为：

```
ValueStack valueStack = ActionContext.getContext().getValueStack();
Person person = new Person();
person.setPid(1);
person.setPname("zz");
person.setComment("person");
valueStack.getRoot().add(0, person);
Student student = new Student();
student.setSid(2);
student.setSname("student");
student.setComment("student");
valueStack.getRoot().add(0, student);
```

从以前学过的 OGNLContext 结构可以看出，对象栈中的分布如图所示：

```
root= CompoundRoot (id=115)
  elementData= Object[4] (id=129)
    [0]= Student (id=131)
    [1]= Person (id=133)
    [2]= OgnlAction (id=63)
    [3]= DefaultTextProvider (id=141)
  modCount= 3
  size= 4
```

但是大家注意一个现象：在 student 对象中有 comment 属性，在 person 对象中也有 comment 属性，在 OgnlAction 中还有 comment 属性，如果页面输出 Comment 属性应该选择哪种呢？

结论：

对于对象栈中存放的属性，struts2 会从最顶部开始寻找，如果找到则赋值，如果找不到，则会依次往下寻找，直到找到为止。所以应该输出的是 student 对象的 comment 的值。

#### 14.10.3.4 用法 4(构造 map)

还可以利用 ognl 表达式构造 Map,如#{'foo1': 'bar1', 'foo2': 'bar2'};这种用法经常用在给 radio,checkbox 等标签赋值上。

1、在页面上可以这样输出：

```
<s:radio list="#{'foo1': 'bar1', 'foo2': 'bar2'}" name="sex" label="性别"></s:radio>
上面的代码相当于
性别: <input type="radio" name="sex" value="foo1">bar1
      <input type="radio" name="sex" value="foo2">bar2
```

2、也可以这样使用：

```
后台代码：
Map<String, String> map = new HashMap<String, String>();
map.put("male", "男");
map.put("female", "女");
ServletContext.getRequest().setAttribute("map", map);//1
ActionContext.getContext().put("map", map);//2
jsp 页面：
如果执行的是 1，则为
<s:property value="#request.map.male"/><br>
<s:property value="#request.map.female"/><br>
如果执行的是 2，则为
<s:property value="map.male"/><br>
<s:property value="map.female"/><br>
```

### 14.10.3.5 用法 5(%)

“%”符号的用途是在标签的属性值给理解为字符串类型时，执行环境%{}中添加的是 OGNL 表达式。

{} 中用 OGNL 表达式

在代码中：

```
ServletActionContext.getRequest().setAttribute("request_username",  
"username");
```

在页面上：

```
<s:textfield name="name"  
label="%{#request.request_username}"></s:textfield>
```

用{}来表示字符串

```
<s:textfield name="username" label="%{'用户名'}"></s:textfield>
```

相当于

```
用户名: <input type="text" name="username">
```

### 14.10.3.6 用法 6(\$)

\$主要有两个用途：

用于在国际化资源文件中引用 OGNL 表达式

在 struts2 的配置文件中引用 OGNL 表达式

在 action 中：

```
ServletActionContext.getRequest().setAttribute("request_username",  
"username");
```

在配置文件中：

```
<result  
name="dollar">successOgnl.jsp?msg=%{#request.request_username}</result>
```

在页面中：

```
<s:property value="#parameters.msg[0]"/>
```

如果把 action 中的值放入到对象栈中呢？

## 14.11 标签用法

### 14.11.1 Property 标签

#### 14.11.1.1 说明

用于输出指定的值

#### 14.11.1.2 属性

1、 default:

可选属性，如果输出的值为 null,则显示该属性指定的值。

2、 escape

可选属性，指定是否格式化为 html 代码

3、 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。

#### 14.11.1.3 例 1(默认值)

```
在 testOgnlTag.jsp 中
利用 property 标签获取栈顶的值:
<br>
<a
    href="${pageContext.request.contextPath}/ognl/ognlTagAction_testProperty1.
action">测试</a>
在 OgnlTagAction 中
public String testProperty1(){
    return "ognlTag";
}
在 successOgnlTag.jsp 中:
利用 property 标签获取栈顶的值:<br>
<s:property/> //没有 value 属性，所以应该输出的是栈顶元素
结果:
cn.itcast.struts2.action.ognltag.OgnlTagAction@5a82b2
```

#### 14.11.1.4 例 2(default 和 value)

在 testOgnlTag.jsp 中

测试 property 中的 default 的值: <br>

<a

href="{pageContext.request.contextPath}/ognl/ognlTagAction\_testDefault.action">测试</a><br>

在 OgnlTagAction 中

```
public String testDefault() {
```

```
    ServletActionContext.getRequest().setAttribute("request_username",  
        "username");
```

```
    return "ognlDefault";
```

```
}
```

在 successOgnlTag.jsp 中

当 property 的 value 没有值的情况下, 输出 default 的值:<br>

```
<s:property value="#request.request_username11" default="default value"/>
```

```
<s:property value="#request.request_username" default="default value"/>
```

说明: 因为在后台 request 中的 key 值为.request\_username, 而页面上的输出为.request\_username11, 不对应, 所以应该输出 default 的值。

#### 14.11.1.5 例 3(escape)

在 testOgnlTag.jsp 中:

测试 property 中的 escape 的值:<br>

```
<s:property value="%{'<hr>hr 的解析'}"/>
```

说明: 因为 escape 默认值为 true, 也就是说<hr>hr 的解析会当作一个字符串处理。

```
<s:property value="%{'<hr>hr 的解析'}" escape="false"/>
```

说明: 因为如果 escape 为 false, 则把字符串中符合 html 标签的语法, 会当作 html 标签来进行处理。

#### 14.11.1.6 例 4(输出栈顶 String 的值)

```
在 testOgnlTag.jsp 中:  
取出栈顶的值: <br>  
<a  
    href="${pageContext.request.contextPath}/ognl/ognlTagAction_testString.action  
>测试</a><br>  
在 OgnlTagAction 中  
public String testString(){  
    ActionContext.getContext().getValueStack().push("msg");  
    return "ognlString";  
}  
在 successOgnlTag 中  
取出栈顶的值: <br>  
<s:property/>
```

### 14.11.2 Debug 标签

#### 14.11.2.1 说明

利用 debug 标签可以输出 OGNLContext 所有的值

#### 14.11.2.2 例子

```
在 testOgnlTag.jsp 中:  
利用 debug 标签输出 ognl 的所有的值: <br>  
<a  
    href="${pageContext.request.contextPath}/ognl/ognlTagAction_testDebug.action">测试  
</a><br>  
在 OgnlTagAction 中  
public String testDebug(){  
    return "ognlDebug";  
}  
在 successOgnlTag.jsp 中:  
利用 debug 标签输出 OgnlContext 中所有的值: <br>  
<s:debug></s:debug>//利用这个标签可以输出 ognlcontext 中所有的值
```

## 14.11.3 Set 标签

### 14.11.3.1 说明

用于将某个指定的值放入到指定的范围

### 14.11.3.2 属性

var:

变量的名字，name、id 与 var 表达的含义是一样的。Var 已经取代了 name,id;

Scope:

指定变量被放置的范围。该属性可以接受 application,session,request,page 或 Action。如果没有设置该属性，则默认会放在 action 中。

Value:

赋值给变量的值。如果没有设置该属性，则将 ValueStack 栈顶的值赋给变量。

### 14.11.3.3 例 1

```
在 testOgnlTag.jsp 中
测试 set 标签: <br>
<a
    href="${pageContext.request.contextPath}/ognl/ognlTagAction_testSet.action
">测试</a><br>
在 OgnlTagAction 中
public String testSet() {
    ServletActionContext.getRequest().setAttribute("request_username",
        "username");
    return "ognlSet";
}
在 successOgnlTag.jsp 中
测试 set 标签:<br>
<s:set value="#request.request_username" var="aaaaa" scope="request"></s:set>
<s:property value="#request.aaaaa"/>
说明: 这样的情况, aaaaa 的 key 值会出现在 request 域中。
<s:set value="#request.request_username" var="aaaaa"></s:set>
<s:property value="#request.aaaaa"/>
说明: 这种写法 aaaaa 会作为一个 key 值存在 ognl 的 map 中。所以利用
<s:property value="#aaaaa"/>也能输出值。
```

## 14.11.4 Push 标签

### 14.11.4.1 说明

把对象放入到栈顶，不能放入到其他的范围，当标签结束时，会从栈顶删除。

### 14.11.4.2 例子

```
在 testOgnlTag.jsp 中
用 push 方法把一个对象放入到栈顶: <br>
<a
    href="${pageContext.request.contextPath}/ognl/ognlTagAction_testPUSH.action"> 测
试</a><br>
在 OgnlTagAction 中
public String testPUSH(){
    ServletActionContext.getRequest().setAttribute("request_username", "username");
    return "ognlPUSH";
}
在 successOgnlTag.jsp 中
利用 push 方法把对象放入到栈顶:<br>
<s:push value="#request.request_username">
    <s:property/>
    <s:debug></s:debug>//注意 debug 标签的位置，这个位置是正确的
</s:push>
<s:debug></s:debug>//这个位置的标签是错误的。因为 s:push 标签已经结束，所以栈顶
元素已经被删除掉了。
```

## 14.11.5 Bean 标签

### 14.11.5.1 说明

实例化一个符合 javabean 规范的 class，标签体内可以包含几个 param 元素，可用于调用 set 方法，给 class 的属性赋值。

### 14.11.5.2 属性

Name:



要被实例化的 class 的名字，符合 javabean 规范。

Var:

赋值给变量的值。放置在 request 作用域中。如果没有设置该属性，对象被设置到栈顶。

### 14.11.5.3 例子

在 testOgnlTag.jsp 中

利用 bean 标签实例化一个 person 对象:<br>

<a

href="{pageContext.request.contextPath}/ognl/ognlTagAction\_testBean.action"> 测 试

</a><br>

在 OgnlTagAction 中

public String testBean(){

ServletActionContext.getRequest().setAttribute("pid", 1);

ServletActionContext.getRequest().setAttribute("request\_username", "username");

ServletActionContext.getRequest().setAttribute("pname", "username");

ServletActionContext.getRequest().setAttribute("comment", "person");

return "ognlBean";

}

在 successOgnlTag.jsp 中

给一个 person 赋值，值来自于后台:<br>

<s:bean name="cn.itcast.struts2.valuestack.bean.Person" var="myperson">

<s:param name="pid" value="#request.pid"></s:param>

<s:param name="pname" value="#request.pname"></s:param>

<s:param name="comment" value="#request.comment"></s:param>

<!--

因为 person 在栈顶，所以输出栈顶

-->

<s:property value="pid"/>

<s:property value="pname"/>

<s:property value="comment"/>

<s:debug></s:debug>

</s:bean>

<s:property/> //值不再是 Person 对象

<!--

前提条件：不加 var="myperson" 属性

由于 bean 标签在栈顶，所以当 bean 标签结束的时候，

栈顶的 person 对象就被删除掉了。所以在 bean 标签的外部输出栈顶的 person 值是不行的。

-->

```
</s:property/>
```

```
<!--
```

当 bean 标签中加 var 属性值为 myperson,用 debug 标签再观察其值的分布情况  
在 map 中出现了"myperson":person 对象

```
-->
```

```
<s:property value="#person.pid"/><br>
```

```
<s:property value="#person.pname"/><br>
```

```
<s:property value="#person.comment"/><br>
```

说明：当加上 var="myperson"属性时，myperson 对象出现在了 map 中，这个时候  
就可以在 bean 标签的外部获取 person 的属性了。

## 14.11.6 Action 标签

### 14.11.6.1 说明

通过指定命名空间和 action 的名称，可以直接调用后台的 action.

### 14.11.6.2 属性

Name:

Action 的名字

Namespace:

Action 所在的命名空间(action 的名称后不加.action)

executeResult:

Action 的 result 是否需要被执行，默认值为 false,不执行

### 14.11.6.3 例子

在 testOgnlTag.jsp 中

通过 action 标签访问后台的 action:<br>

```
<s:action          name="ognlTagAction_testBean"          namespace="/ognl"
executeResult="true"></s:action>
```

在 OgnlTagAction 中

```
public String testBean(){
    ServletActionContext.getRequest().setAttribute("pid", 1);
    ServletActionContext.getRequest().setAttribute("request_username", "username");
    ServletActionContext.getRequest().setAttribute("pname", "username");
    ServletActionContext.getRequest().setAttribute("comment", "person");
    return "ognlBean";
}
```

说明:

如果 executeResult 的属性值为 false,会执行到相应的 action,但是 action 跳转后的页面将不再执行。如果 executeResult 的属性值为 true,则会在当前页面包含跳转后的页面值。

## 14.11.7 Iterator 标签

### 14.11.7.1 说明

该标签用于对集合进行迭代。这里的集合包括: list,set 和数组

### 14.11.7.2 属性

Value:

可选属性, 指定被迭代的集合。如果没有设置该属性, 则使用对象栈顶的集合。

Var:

可选属性, 引用变量的名称

Status:

可选属性, 该属性指定迭代时的 IteratorStatus 实例。该实例包含如下的方法:

int getCount() 返回当前迭代的元素个数

int getIndex() 返回当前迭代元素的索引

boolean isEven() 返回当前迭代元素的索引是否是偶数

boolean isOdd() 返回当前迭代元素的索引是否是奇数

boolean isFirst() 返回当前迭代元素是否为第一个元素

boolean isLast() 返回当前迭代元素是否为最后一个元素

### 14.11.7.3 例 1(push)

在 testOgnlTag.jsp 中

通过 iterator 标签把后台的 list 集合遍历(list 通过 push 方法压入到栈顶)<br>

```
<a href="{pageContext.request.contextPath}/ognl/ognlTagAction_testList1.action"> 测 试  
</a><br>
```

在 OgnlTagAction 中

```
/*  
    * 把生成的 list 利用 push 方法压入到栈顶  
    */  
public String testList1(){  
    ServletActionContext.getRequest().setAttribute("request_username", "username");  
    List<Person> personList = new ArrayList<Person>();  
    List<Person> person1List = new ArrayList<Person>();  
    for(int i=0;i<10;i++){  
        Person person = new Person();  
        person.setPid(i);  
        person.setPname("person"+i);  
        person.setComment("person"+i);  
        personList.add(person);  
    }  
  
    for(int i=11;i<15;i++){  
        Person person = new Person();  
        person.setPid(i);  
        person.setPname("person"+i);  
        person.setComment("person"+i);  
        person1List.add(person);  
    }  
    /*  
    * 说明:  
    * 1、执行完三次压栈以后, 对象栈的栈顶存放的元素为 String 类型的 aaa;  
    * 而这个时候如果页面用<s:iterator value="top"></s:iterator>  
    * top 代表对象栈的栈顶的元素  
    * 进行迭代的时候, 针对的元素是 aaa;  
    */  
}
```

```

*      2、执行第一次压栈和第二次压栈，对象栈的栈顶存放的元素为
person1List;
*      页面上用
*      <s:iterator value="top"></s:iterator>
*      top 代表对象栈的栈顶的元素
*      进行迭代的时候，针对的元素是 person1List;
*      3、执行第一次压栈，对象栈的栈顶存放的元素为 personList;
*      页面上用
*      <s:iterator value="top"></s:iterator>
*      top 代表对象栈的栈顶的元素
*      进行迭代的时候，针对的元素是 personList;
*/
//把 personList 压入到栈顶
ActionContext.getContext().getValueStack().push(personList);
/*
* 把 perosn1List 压入到栈顶
*/
ActionContext.getContext().getValueStack().push(person1List);
/*
* 把 aaa 压入到栈顶
*/
ActionContext.getContext().getValueStack().push("aaa");
return "ognlList1";
}

```

#### 14.11.7.4 例 2(action 属性)

在 testOgnlTag.jsp 中

通过iterator标签把后台的list集合遍历(list作为action中的属性) <br>

```
<a  
href="{pageContext.request.contextPath}/ognl/ognlTagAction_testList2.action">测试</a><br>
```

在OgnlTagAction中

```
private List<Person> pList;  
public List<Person> getPList() {  
    return pList;  
}  
public void setPList(List<Person> list) {  
    pList = list;  
}  
public String testList2(){  
    ServletActionContext.getRequest().setAttribute("request_username", "username");//在successOgnlTag.jsp中  
    <s:push value="#request.request_username">标签需要用到后台的这个操作。  
  
    this.pList = new ArrayList<Person>();  
    for(int i=0;i<10;i++){  
        Person person = new Person();  
        person.setPid(i);  
        person.setPname("person"+i);  
        person.setComment("person"+i);  
        pList.add(person);  
    }  
    return "ognlList2";  
}
```

说明：通过代码可以看出来，这个例子中把pList作为action的属性，然后给其赋值。

在successOgnlTag.jsp中

用iterator标签迭代后台list中的数据(list作为action中的属性):<br>

```
<s:iterator value="pList">  
    <s:property value="pname"/>  
    <s:property/>  
    <s:debug></s:debug>  
</s:iterator><br>  
    <s:debug></s:debug>
```

说明：因为OgnlTagAction在对象栈，所以value中的pList可以不加#号。从第一个debug标签可以看出，这个标签把当前正在迭代的对象临时放入了栈顶。如果iterator元素结束迭代时，栈顶的对象就消失了。所以第一次的debug和第二次的Debug内容是不一样的。

### 14.11.7.5 例 3(Map 中)

在 testOgnlTag.jsp 中

通过 iterator 标签把后台的 list 集合遍历(list 通过 ActionContext.getContext().put 放入 OgnlContext 的 Map 中)<br>

```
<a
    href="{pageContext.request.contextPath}/ognl/ognlTagAction_testList3.action"
>测试</a><br>
```

在 OgnlTagAction 中

```
/*
    * 把 personList 通过 put 方法放入到 OgnlContext 的 map 中
    */
public String testList3(){
    ServletActionContext.getRequest().setAttribute("request_username",
"username");
    List<Person> personList = new ArrayList<Person>();
    for(int i=0;i<10;i++){
        Person person = new Person();
        person.setPid(i);
        person.setPname("person"+i);
        person.setComment("person"+i);
        personList.add(person);
    }
    ActionContext.getContext().put("personList", personList);
    return "ognlList3";
}
```

在 successOgnlTag.jsp 中

用 iterator 标签迭代后台 list 中的数据(list 经过 ActionContext.getContext().put 方法放入到 OgnlContext 中 )<br>

```
<s:iterator value="personList">
    <s:property value="pname"/>
</s:property/>
<s:debug></s:debug>
</s:iterator><br>
<s:debug></s:debug>
```

说明：用 iterator 进行迭代，迭代的集合如果来自 map,value 的值可以加#也可以不加#,这点需要注意。Debug 的现象和例 2 一样。

#### 14.11.7.6 例 4(begin,end,step)

在 testOgnlTag.jsp 中

利用 iterator 的 begin、end、step 属性控制数据的显示:<br>

<a

href="{pageContext.request.contextPath}/ognl/ognlTagAction\_testList4.action">测试</a><br>

在 OgnlTagAction 中

```
/*
 * 在页面上通过 iterator 的 begin,end,step 属性控制页面的显示
 */
public String testList4(){

    ServletActionContext.getRequest().setAttribute("request_username",
"username");

    List<Person> personList = new ArrayList<Person>();
    for(int i=0;i<10;i++){
        Person person = new Person();
        person.setPid(i);
        person.setPname("person"+i);
        person.setComment("person"+i);
        personList.add(person);
    }

    ServletActionContext.getRequest().setAttribute("first", 2);
    ServletActionContext.getRequest().setAttribute("end", 8);
    ServletActionContext.getRequest().setAttribute("step", 2);
    ActionContext.getContext().put("personList", personList);
    return "ognlList4";
}
```

在 successOgnlTag.jsp 中

用 iterator 标签属性 begin,end,step 控制数据的显示: <br>

```
<s:iterator value="personList" begin="%{#request.first}"
end="%{#request.end}" step="%{#request.step}">
    <s:property value="pname"/><br>
</s:iterator><br>
```

说明: begin 属性为开始位置,end 属性为结束位置, step 为步长。这里要注意 begin 的值是通过 ognl 表达式传递过来的。



### 14.11.7.7 例 5(status)

在 testOgnlTag.jsp 中

测试 iterator 的 status 属性: <br>

<a

href="{pageContext.request.contextPath}/ognl/ognlTagAction\_testList5.action"> 测 试

</a><br>

在 OgnlTagAction 中

/\*

\* 测试 iterator 的 status 属性

\*/

public String testList5(){

ServletActionContext.getRequest().setAttribute("request\_username", "username");

List<Person> personList = new ArrayList<Person>();

for(int i=0;i<10;i++){

Person person = new Person();

person.setPid(i);

person.setPname("person"+i);

person.setComment("person"+i);

personList.add(person);

}

ActionContext.getContext().put("personList", personList);

return "ognlList5";

}

在 successOgnlTag.jsp 中

测试 iterator 的 status 属性的值: <br>

<!--

如果提供 status 每次迭代时候将生成一个 IteratorStatus 实例并放入堆栈中  
用 debug 标签查看堆栈, 在 map 中, 存在一个 st, 类型为

org.apache.struts2.views.jsp.IteratorStatus

st 对应一个 IteratorStatus 对象, 有如下属性

int getCount() 返回当前迭代的元素个数

int getIndex() 返回当前迭代元素的索引

boolean isEven() 返回当前迭代元素的索引是否是偶数

boolean isOdd() 返回当前迭代元素的索引是否是奇数

boolean isFirst() 返回当前迭代元素是否为第一个元素

boolean isLast() 返回当前迭代元素是否为最后一个元素-->

<table border="1">

<s:iterator value="personList" var="person" status="st">

<tr>

<td><s:property value="#st.count"/></td>

<td><s:property value="#st.getIndex()"/></td>

<td><s:property value="#person.pname"/></td>

</tr>

</s:iterator>

</table>

说明：上述标明 status 属性，就会在栈中的 map 中有一个 key 值为 st，而 st 对应的值为 org.apache.struts2.views.jsp.IteratorStatus 的对象。在迭代每一个元素的时候，都存在这个对象，我们可以根据这个对象获取到该对象的属性。从而得到遍历中的相关信息。

#### 14.11.7.8 例 6(奇偶行变色)

```
testOgnlTag.jsp
利用 iterator 的 status 属性完成隔行变色:<br>
<a
href="{pageContext.request.contextPath}/ognl/ognlTagAction_testList6.action">    测    试
</a><br>
OgnlTagAction
/*
    * 利用 iterator 的 status 属性完成表格的隔行变色
    */
public String testList6(){
    ServletActionContext.getRequest().setAttribute("request_username", "username");
    List<Person> personList = new ArrayList<Person>();
    for(int i=0;i<10;i++){
        Person person = new Person();
        person.setPid(i);
        person.setPname("person"+i);
        person.setComment("person"+i);
        personList.add(person);
    }
    ActionContext.getContext().put("personList", personList);
    return "ognlList6";
}
successOgnlTag.jsp
<style type="text/css">
    .odd{
        background-color:red;
    }
    .even{
        background-color:blue;
    }
</style>
```

利用 iterator 的 status 属性隔行变色: <br>

```
<table border="1">
  <s:iterator value="personList" var="person" status="st">
    <!--
      value 的值可以跟双目表达式
    -->
    <tr class="<s:property value="#st.even?'even':'odd'"/>">
      <td><s:property value="#st.count"/></td>
      <td><s:property value="#st.getIndex()"/></td>
      <td><s:property value="#st.isEven()"/></td>
      <td><s:property value="#st.isOdd()"/></td>
      <td><s:property value="#st.isFirst()"/></td>
      <td><s:property value="#st.isLast()"/></td>
      <td><s:property value="#person.pid"/></td>
      <td><s:property value="#person.pname"/></td>
    </tr>
  </s:iterator>
</table>
```

说明: <s:property value="#st.even?'even':'odd'"/>这是一个双目表达式。

## 14.11.8 If/elseif/else 标签

### 14.11.8.1 说明

基本的流程控制标签。If 标签可以单独使用, 也可以结合 elseif 或 else 标签使用。

### 14.11.8.2 属性

test: 后面跟判断表达式。

### 14.11.8.3 例子

```
testOgnlTag.jsp
测试 if/elseif/else 标签的使用:<br>
<a
href="${pageContext.request.contextPath}/ognl/ognlTagAction_testIF.action"> 测
试</a><br>
OgnlTagAction
/*
    * 测试 if 标签
    */
public String testIF(){
    ServletActionContext.getRequest().setAttribute("request_username",
"username");
    List<Person> personList = new ArrayList<Person>();
    for(int i=0;i<10;i++){
        Person person = new Person();
        person.setPid(i);
        person.setPname("person"+i);
        person.setComment("person"+i);
        personList.add(person);
    }
    ActionContext.getContext().put("personList", personList);
    return "ognlIF";
}
successOgnlTag.jsp
测试 if/elseif/else 标签: <br>
<table border="1">
    <s:iterator value="personList" var="person" status="st">
        <!--
            value 的值可以跟双目表达式
        -->
        <tr class="<s:property value="#st.even?'even':'odd'"/>">
            <td><s:property value="#person.pid"/></td>
            <td><s:property value="#person.pname"/></td>
            <td>
                <s:if test="#person.pid<3">这个 id 号小于 3</s:if>
                <s:elseif test="#person.pid<5">这个 id 号大于等于 3
小于 5</s:elseif>
                <s:else>这个 id 号大于等于 5</s:else>
            </td>
        </tr>
    </s:iterator>
</table>
```

## 14.11.9 url 标签

### 14.11.9.1 说明

该标签用于创建 url,可以通过”param”标签提供 request 参数。

### 14.11.9.2 属性

Value:

如果没有值,就用当前的 action,使用 value 后必须加.action.

Action:

用来生成 url 的 action.如果没有使用则用 value;

Namespace:

命名空间

Var:

引用变量的名称。

### 14.11.9.3 例子

```
testOgnlTag.jsp
测试 url 标签的使用:<br>
<a
href="${pageContext.request.contextPath}/ognl/ognlTagAction_testURL.action"> 测 试
</a><br>
OgnlTagAction
/*
    * 测试标签 url
    */
    public String testURL(){
        ServletActionContext.getRequest().setAttribute("request_username",
"username");
        return "ognlURL";
    }
}
```

successOgnlTag.jsp

测试 url 标签的使用:<br>

```
<!--
```

作用：直接输出 url 地址

```
-->
```

```
<s:url action="ognlTagAction.action" namespace="/ognl" var="myurl">
```

```
<s:param name="pid" value="12"></s:param>
```

```
<s:param name="pname" value="%{'zd'}"></s:param>
```

```
</s:url>
```

```
<s:debug></s:debug>
```

```
<a href="<s:property value="#myurl"/>">test</a>
```

说明：如果 url 标签不写 var 属性，则 url 标签输出 url 的路径。如果写上 var 变量，则会在 OgnlContext 中出现相应的 key 值：myurl。所以在程序的最后一行引用了 #myurl 变量从而得到路径。Param 是传递的参数。这里需要注意：value="%{'zd'}" 可以，但是 value="zd" 是不行的。后面的写法会去栈顶查找 zd。所以 s:url 标签就相当于声明一个 url，然后在外部使用。

## 14.11.10 Ognl 操作集合

Java 代码	OGNL 表达式
list.get(0)	List[0]
array[0]	array[0]
((User)list.get(0)).getName()	list[0].name
Array.length	Array.length
List.size()	List.size
List.isEmpty()	List.isEmpty

testOgnlTag.jsp

测试集合的长度：<br>

```
<a
```

```
href="${pageContext.request.contextPath}/ognl/ognlTagAction_testListLength.action"
```

```
>测试</a><br>
```

OgnlTagAction

```
/*
```

```
 * 测试集合的长度
```

```
*/
```

```
public String testListLength(){
```

```
    ServletActionContext.getRequest().setAttribute("request_username",  
"username");
```

```
    List<Person> personList = new ArrayList<Person>();
```

```
for(int i=0;i<10;i++){
    Person person = new Person();
    person.setPid(i);
    person.setPname("person"+i);
    person.setComment("person"+i);
    personList.add(person);
}
ActionContext.getContext().put("personList", personList);
return "ognlListLength";
}
successOgnlTag.jsp
测试集合的长度: <br>
<s:property value="#personList.size"/>
直接写一个集合:<br>
<s:iterator value="{1,2,3,4}">
    <s:property/>
</s:iterator>
```

### 14.11.11 Ognl 操作 Map

Java 代码	Ognl 表达式
map.get("foo")	Map['foo']
Map.get(new Integer(1));	Map[1]
User user = (User)map.get("user"); Return user.getName()	Map['user'].name
Map.size()	Map.size
Map.isEmpty()	Map.isEmpty
Map.get("foo")	Map.foo

Java 代码	Ognl 表达式
Map map = new HashMap(); Map.put("foo","bar"); Map.put("1","2"); Return map;	#{"foo":"bar","1":"2"}
Map map = new HashMap(); Map.put(new Integer(1),"a"); Map.put(new Integer(2),"b"); Map.put(new Integer(3),"c");	#{1:"a",2:"b",3:"c"}

```
testOgnlTag.jsp
写一个 map 集合:<br>
    <s:iterator value="#{1:'a',2:'b',3:'c'}">
        <!--
            获取 map 中的 key 和 value
        -->
        <s:property value="value"/>
        <s:property value="key"/>
        <s:debug></s:debug>
    </s:iterator>
<br>
    <s:iterator value="#{1:'a',2:'b',3:'c'}" var="map">
        <!--
            获取 map 中的 key 和 value
        -->
        <s:property value="value"/><br>
        <s:property value="key"/><br>
        <s:property value="#map.value"/><br>
    </s:iterator>
```

## 15 UI 标签

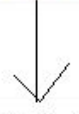
### 15.1 说明

- 1、UI 标签在 html 文件中表现为一个表单元素。
- 2、使用 struts2 的 ui 标签有如下好处：
  - 1、 可以进行表单的回显
  - 2、 对页面进行布局和排版
- 3、 标签的属性可以被赋值为一个静态的值，或者一个 OGNL 表达式。



## 15.2 Form 标签

```
<s:form id="myform" name="myform" action="uiTagAction" method="post"></s:form>
```




```
<form id="myform" name="myform" action="uiTagAction" method="post">  
  <table class="wwFormTable"></table>  
</form>
```

### 15.2.1 说明

- 1、id 属性为 s:form 的唯一标识。可以用 document.getElementById() 获取。
- 2、name 属性为 s:form 的名字，可以用 document.getElementsByName() 获取。
- 3、在默认情况下，s:form 将显示表格的形式。

## 15.3 Textfield 标签

```
<s:textfield name="username" id="username" label="姓名" value="王二麻子"></s:textfield>
```




```
<tr>  
  <td class="tdLabel">  
    <label for="username" class="label">姓名:</label>  
  </td>  
  <td>  
    <input type="text" name="username" value="王二麻子" id="username"/>  
  </td>  
</tr>
```

### 15.3.1 说明

实际上相当于在表格中多了一行，在这行中多了两列。其变化从上述图中可以很明显的看出。

## 15.4 Password 标签

```
<s:password name="password" id="password" label="密码" value="aaaa" showPassword="true"></s:password>
```




```
<tr>  
  <td class="tdLabel">  
    <label for="password" class="label">密码:</label>  
  </td>  
  <td>  
    <input type="password" name="password" value="aaaa" id="password"/>  
  </td>  
</tr>
```

### 15.4.1 说明

如果不加 showPassword 属性,则密码不会显示,把 showPassword 属性的值设置为 true,就能显示密码。

## 15.5 Hidden 标签

```
<s:hidden name="myhidden" id="myhidden" value="myhidden"></s:hidden>
```



```
<input type="hidden" name="myhidden" value="myhidden" id="myhidden"/>
```


### 15.5.1 说明

Hidden 标签并没有加 tr 和 td

## 15.6 Submit 标签

### 15.6.1 情况一

```
<s:submit type="submit" value="提交" name="submit"></s:submit>
```




```
<tr>  
  <td colspan="2">  
    <div align="right">  
      <input type="submit" id="myform_submit" name="submit" value="提交"/>  
    </div>  
  </td>  
</tr>
```

### 15.6.2 说明一

这种情况为 submit 的 type 属性为 submit 类型。

### 15.6.3 情况二：

```
<s:submit type="button" value="提交" name="submit_button"></s:submit>
```



```
<tr>  
  <td colspan="2">  
    <div align="right">  
      <button type="submit" id="myform_submit_button" name="submit_button" value="提交"/>  
    </div>  
  </td>  
</tr>
```

### 15.6.4 说明二

这种情况 submit 的 type 属性为 button.

### 15.6.5 情况三

```
<s:submit type="image" value="提交" name="submit_image"></s:submit>
```

↓

```
<tr>
  <td colspan="2">
    <div align="right">
      <input type="image" alt="提交" id="myform_submit_image" name="submit_image" value="提交"/>
    </div>
  </td>
</tr>
```

### 15.6.6 说明三

该 type 类型为 image。

### 15.6.7 综合

以上三种情况说明，当 type 为 submit、button、image 都能完成提交。

## 15.7 Reset 标签

```
<s:reset type="input" value="重置" name="reset"></s:reset>
```

↓

```
<tr>
  <td colspan="2">
    <div align="right">
      <input type="reset" name="reset" value="重置"/>
    </div>
  </td>
</tr>
```

```
<s:reset type="button" value="重置" name="reset"></s:reset>
```

↓

```
<tr>
  <td colspan="2">
    <div align="right">
      <button type="reset" name="reset" value="重置"/>
    </div>
  </td>
</tr>
```

## 15.8 Textarea 标签

```
<s:textarea name="mytextarea" id="mytextarea" value="aaaa" cols="10"
            rows="10" label="文本域"></s:textarea>
```

```
<tr>
  <td class="tdLabel">
    <label for="mytextarea" class="label">文本域:</label>
  </td>
  <td>
    <textarea name="mytextarea" cols="10" rows="10" id="mytextarea">aaaa</textarea>
  </td>
</tr>
```

## 15.9 Checkbox 标签

```
<s:checkbox name="mycheckbox" value="true" label="aaa" fieldValue="1"></s:checkbox>
<s:checkbox name="mycheckbox" value="true" label="bbb" fieldValue="2"></s:checkbox>
<s:checkbox name="mycheckbox" value="true" label="ccc" fieldValue="3"></s:checkbox>
```

```
<tr>
  <td valign="top" align="right"></td>
  <td valign="top" align="left">
    <input type="checkbox" name="mycheckbox" value="1" checked="checked" id="myform_mycheckbox"/>
    <input type="hidden" id="__checkbox_myform_mycheckbox" name="__checkbox_mycheckbox" value="1" />
    <label for="myform_mycheckbox" class="checkboxLabel">aaa</label>
  </td>
</tr>

<tr>
  <td valign="top" align="right"></td>
  <td valign="top" align="left">
    <input type="checkbox" name="mycheckbox" value="2" checked="checked" id="myform_mycheckbox"/>
    <input type="hidden" id="__checkbox_myform_mycheckbox" name="__checkbox_mycheckbox" value="2" />
    <label for="myform_mycheckbox" class="checkboxLabel">bbb</label>
  </td>
</tr>

<tr>
  <td valign="top" align="right"></td>
  <td valign="top" align="left">
    <input type="checkbox" name="mycheckbox" value="3" checked="checked" id="myform_mycheckbox"/>
    <input type="hidden" id="__checkbox_myform_mycheckbox" name="__checkbox_mycheckbox" value="3" />
    <label for="myform_mycheckbox" class="checkboxLabel">ccc</label>
  </td>
</tr>
```

## 15.10 Checkboxlist 标签

### 15.10.1 集合为 list

```
<s:checkboxlist list="{ '篮球', '足球', '排球', '羽毛球' }" name="hobby" label="爱好"
value="{ '篮球', '足球' }"></s:checkboxlist>
```

```
<tr>
  <td class="tdLabel"><label for="myform_hobby" class="label">爱好:</label></td>
  <td>
    <input type="checkbox" name="hobby" value="篮球"
      id="hobby-1" checked="checked"/>
    <label for="hobby-1" class="checkboxLabel">篮球</label>
    <input type="checkbox" name="hobby" value="足球"
      id="hobby-2" checked="checked" />
    <label for="hobby-2" class="checkboxLabel">足球</label>
    <input type="checkbox" name="hobby" value="排球"
      id="hobby-3" />
    <label for="hobby-3" class="checkboxLabel">排球</label>
    <input type="checkbox" name="hobby" value="羽毛球"
      id="hobby-4" />
    <label for="hobby-4" class="checkboxLabel">羽毛球</label>
    <input type="hidden" id="__multiselect_myform_hobby" name="__multiselect_hobby"
      value="" />
  </td>
</tr>
```

## 15.10.2 集合为 map

```
<s:checkboxlist list="#{1: '专科学位', 2: '本科学位', 3: '硕士学位', 4: '幼儿园学位'}"
name="学位" label="学位" value="#{4: '幼儿园学位'}"></s:checkboxlist>
```

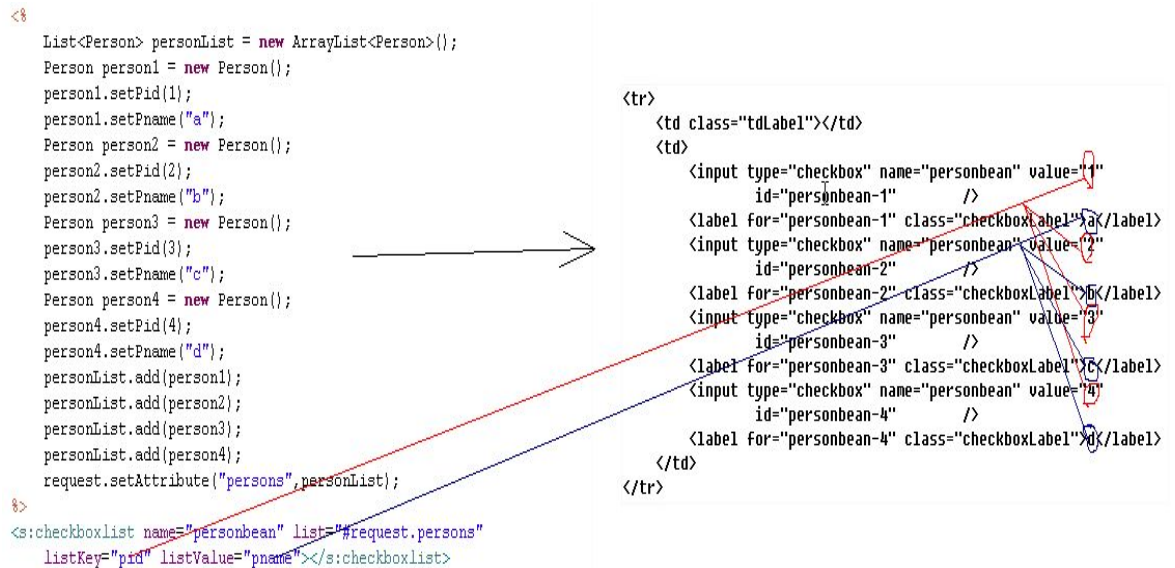
```
<tr>
  <td class="tdLabel"><label for="myform_学位" class="label">学位:</label></td>
  <td>
    <input type="checkbox" name="学位" value="1"
      id="学位-1" />
    <label for="学位-1" class="checkboxLabel">专科学位</label>
    <input type="checkbox" name="学位" value="2"
      id="学位-2" />
    <label for="学位-2" class="checkboxLabel">本科学位</label>
    <input type="checkbox" name="学位" value="3"
      id="学位-3" />
    <label for="学位-3" class="checkboxLabel">硕士学位</label>
    <input type="checkbox" name="学位" value="4"
      id="学位-4" checked="checked" />
    <label for="学位-4" class="checkboxLabel">幼儿园学位</label>
    <input type="hidden" id="__multiselect_myform_学位" name="__multiselect_学位"
      value="" />
  </td>
</tr>
```

上述的 checkboxlist 标签也可以表示为:

```
<s:checkboxlist list="#{1: '专科学位', 2: '本科学位', 3: '硕士学位', 4: '幼儿园学位'}"
name="学位" label="学位" value="#{4: '幼儿园学位'}" listKey="key" listValue="value"></s:checkboxlist>
```

listKey 相当于<input type="checkbox">中的 value, listValue 相当于 label 的显示值。

### 15.10.3 List 中是 javabean



从图中可以看出：listKey 的值对应☐中的 value;

而 listValue 的值对应 label 的内容。

s:select, s:radio 的应用与 checkbox 相同。

### 15.11 标签回显

```
BackValueAction:
private String username = "aaa";
private String password = "bbb";
public String getUsername() {
    return username;
}
public void setUsername(String username) {
    this.username = username;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
```

Backvalue.jsp

```
<s:textfield name="username"></s:textfield>
```

```
<s:password name="password" showPassword="true"></s:password>
```

注意：这里标签中的 **username** 和 **action** 中的属性名称必须保持一致。

其他的标签也是根据 **name** 进行回显。

## 16 属性驱动

### 16.1 说明

在 `servlet` 中获取页面传递过来的数据的方式是：  
`request.getParameter("username")`;这个代码可以获取到页面的 `username` 的数据。在 `action` 中可以通过属性驱动的方式来获取页面的值。

### 16.2 例子

Propertydriver/login.jsp

```
<form          action="propertydriver/propertyDriverAction_testPropertyDriver.action"
method="post">
```

```
    用户名:<input type="text" name="username"/>
```

```
    密码: <input type="password" name="password"/>
```

```
    <input type="submit"/>
```

```
</form>
```

PropertyDriverAction

```
public class PropertyDriverAction extends ActionSupport{
```

```
    private String username;
```

```
    private String password;
```

```
    public String getUsername() {
```

```
        return username;
```

```
    }
```

```
    public void setUsername(String username) {
```

```
        this.username = username;
```

```
    }
```

```
    public String getPassword() {
```

```
        return password;
```

```
    }
```

```
    public void setPassword(String password) {
```

```
        this.password = password;
```

```
    }
```

```
    public String testPropertyDriver(){
```

```
        System.out.println(this.username);
```

```
        System.out.println(this.password);
```

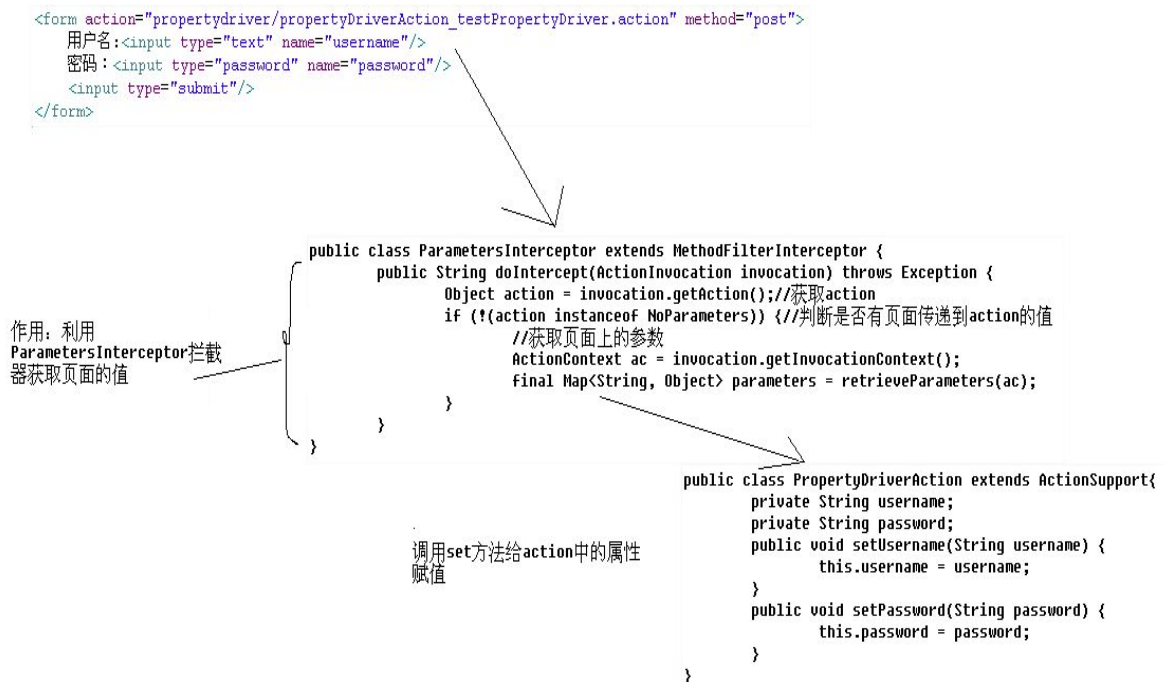
```
        return SUCCESS;
```



说明：

- 1、页面中 name 的属性和 action 中的属性必须保持一致。
- 2、Action 中的属性必须有 get 和 set 方法。
- 3、满足这两个条件就实现了属性驱动。

## 16.3 原理



- 1、当执行所有的拦截器的时候，当前请求的 action 已经放在了栈顶。
- 2、放在对象栈的对象的特点是其属性能够直接访问。
- 3、也就是说当执行 ParameterInterceptor 拦截器的时候，action 的所有的属性在栈顶。
- 4、所以只需要给栈顶的 action 的属性赋值就可以了。
- 5、而 ParameterInterceptor 拦截器正好完成了此功能。

如果属性中要求接受的不是 String 类型，而是其他类型呢？struts2 将做自动的转化。

## 17 类型转化

### 17.1 问题

```
testConverter.jsp
<form                action="converter/converterAction_testConverter.action"
method="post">
    年龄:<input type="text" name="age"><br>
    姓名:<input type="text" name="name"><br>
    出生日期:<input type="text" name="birthday"><br>
    <input type="submit"/>
</form>
```

通过属性驱动可以得出，只要在 action 中有 age 和 name 属性，有 set 和 get 方法就能得到页面上 age 和 name 的值。如果 action 中是日期类型呢？在 struts2 中可以自动隐式得到转化。比如在 struts2 中可以把字符串类型转化为日期类型。但是必须要求是 yyyy-mm-dd 的格式。其他格式转换不了。如果页面上传过来的是 yyyyMMdd，应该怎么办呢？类型转化将解决这个问题。

### 17.2 解决方案

#### 17.2.1 DateConverter 类

```
public class DateConverter extends DefaultTypeConverter{
    @Override
    public Object convertValue(Map context, Object value, Class toType) {
        // TODO Auto-generated method stub
        String dateStr = ((String[])value)[0];
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyyMMdd");
        try {
            return simpleDateFormat.parse(dateStr);
        } catch (ParseException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

说明：

6、这个类必须继承 `DefaultTypeConverter` 或者实现 `TypeConverter` 接口。

7、在 `TypeConverter` 接口中，声明的方法是这样的：

```
public Object convertValue(Map context, Object target,  
                           Member member, String propertyName,  
                           Object value, Class toType);
```

可以看出里面实际上有六个参数。

但是这个类中，只有三个参数。

在 `DefaultTypeConverter` 类中：

```
public class DefaultTypeConverter implements TypeConverter  
{  
    public DefaultTypeConverter()  
    {  
        super();  
    }  
  
    public Object convertValue(Map context, Object value, Class toType)  
    {  
        return OgnlOps.convertValue(value, toType);  
    }  
  
    public Object convertValue(Map context, Object target, Member member, String propertyName, Object value, Class toType)  
    {  
        return convertValue(context, value, toType);  
    }  
}
```

可以看出在实现了 `convertValue` 的方法中调用了 `convertValue` 有三个参数的方法，所以能够执行。这样做的好处是程序员即可以用三个参数的方法，

也可以用 6 个参数的方法。

8、在 `convertType` 方法中：

`value` 为从页面上传过来的值

`toType` 为转换以后的类型

## 17.2.2 properties 文件

做完 17.2.1 以后，`DateConverter` 类仅仅是一般的类，struts2 框架不能把这个类看作数据结构的转换类。所以需要把 `DateConverter` 类用配置文件进行注册。

注意事项：

1、这个文件必须和相应的 `action` 类放在同一个目录下。

2、文件的名称为：类名-conversion.properties。

3、配置文件中的内容为：

`action` 中的属性名称=`DateConverter` 全名

4、在执行的时候，一定要让浏览器的环境是中文的执行环境。在默认情况下，`yyyy-MM-dd` 只有在中文的浏览器环境下才能识别，如果是英文识别不了。  
可以参照 `XWorkBasicConverter` 这个类说明问题。

## 17.2.3 全局 properties 文件

除了 17.2.2 的做法，还可以考虑全局的配置文件。

步骤：

1、在 `src` 下新建一个 `properties` 文件，为 `xwork-conversion.properties`。

2、在文件中键值对是这样的：

`java.util.Date= cn.itcast.struts2.action.converter.DateConverter`

这样配置，只要是 `java.util.Date` 类型都会通过 `DateConverter` 这个类进行转化。适合于所有的 `action` 中的属性。

# 18 模型驱动

## 18.1 说明

假设你正在完成一个网站的注册功能。在后台需要得到 20 多个属性，才能完成注册。如果用 action 中的属性获取值，就是这样的情况：

- 1、在 action 中会写 20 个属性
- 2、这 20 个属性添加 set 和 get 方法。

这样会导致 action 中的代码结构不是很好。模型驱动很好的解决了这个问题。

## 18.2 例子

Modeldriver.jsp

```
<s:form action="modeldriver/modelDriverAction_modeldriver.action" method="post">
  <s:textfield name="username"></s:textfield>
  <s:password name="password"></s:password>
  <s:submit></s:submit>
</s:form>
```

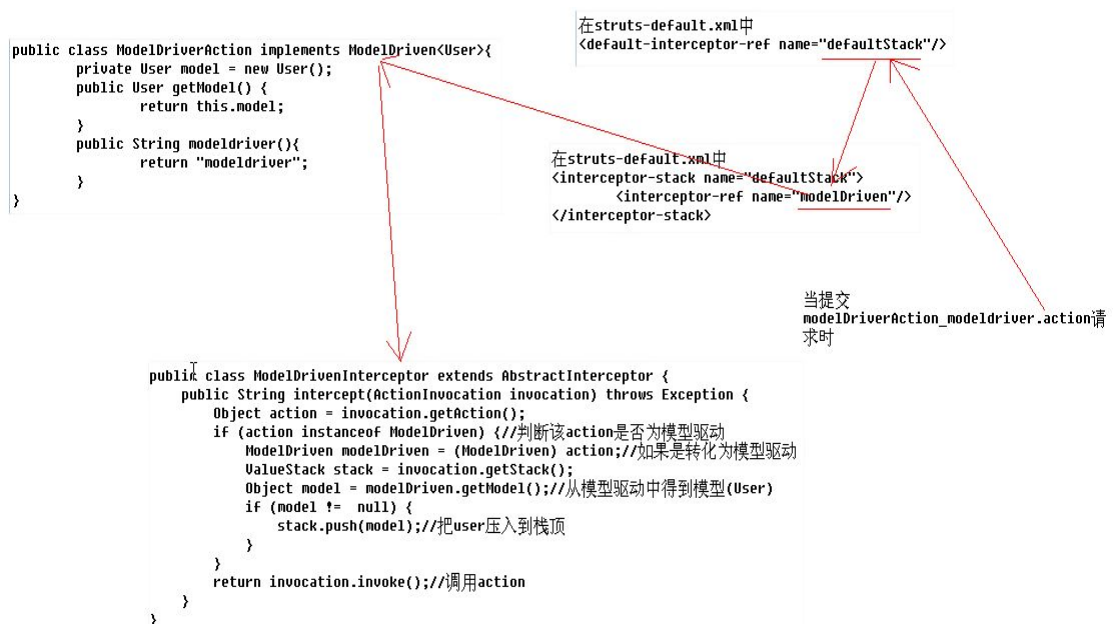
ModelDriverAction

```
public class ModelDriverAction extends ActionSupport implements ModelDriven<User>{
    private User model = new User();
    public User getModel() {
        // TODO Auto-generated method stub
        return this.model;
    }
    public String modeldriver(){
        return "modeldriver";
    }
}
```

Modeldriver\_value.jsp

实现回显的功能

## 18.3 原理



过程为：当浏览器提交 modelDriverAction\_modeldriver.action 请求时，先经过拦截器。其中有一个拦截器为 ModelDrivenInterceptor，从这个源代码可以看出，这个拦截器的作用就是获取实现了 ModelDriver 接口的 action 的模型驱动。在这里为 user。然后把模型驱动利用 push 方法压入到栈顶。这样我们就能直接通过属性进行回显和赋值了。

```
<interceptor-stack name="defaultStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="alias"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="i18n"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="chain"/>
  <interceptor-ref name="scopedModelDriven"/>
  <interceptor-ref name="modelDriven"/>
  <interceptor-ref name="fileUpload"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="multiselect"/>
  <interceptor-ref name="staticParams"/>
  <interceptor-ref name="actionMappingParams"/>
  <interceptor-ref name="params">
    <param name="excludeParams">dojo\..*,^struts\..*</param>
  </interceptor-ref>
</interceptor-stack>
```

通过这个图也可以看出模型驱动的拦截器在参数拦截器前面，也就是先把模型驱动压入栈顶，再进行赋值。

## 19 文件上传

struts2 对文件上传做了更进一步的封装，使得开发更加简单。

### 19.1 页面

```
testUpload.jsp
<form action="upload/uploadAction_testUpload.action" method="post"
      enctype="multipart/form-data">
    要上传的文件:
    <input type="file" name="upload">
    <br>
    <input type="submit">
</form>
```

注意：这里 `enctype="multipart/form-data"` 是必须写的，以二进制的形式进行上传。

### 19.2 UploadAction 中

```
public class UploadAction extends ActionSupport {
    private File upload;
    private String uploadContentType;
    private String uploadFileName;
    public File getUpload() {
        return upload;
    }
    public String getUploadContentType() {
        return uploadContentType;
    }
    public void setUploadContentType(String uploadContentType) {
        this.uploadContentType = uploadContentType;
    }
    public String getUploadFileName() {
        return uploadFileName;
    }
    public void setUploadFileName(String uploadFileName) {
        this.uploadFileName = uploadFileName;
    }
    public void setUpload(File upload) {
        this.upload = upload;
    }
}
```

```

public String testUpload() throws Exception {
    String path = ServletActionContext.getServletContext().getRealPath("/upload");
    File destFile = new File(path, this.uploadFileName);
    FileUtils.copyFile(upload, destFile);
    return SUCCESS;
}
}

```

说明：

- upload 代表要上传的文件，和页面上 name 的值对应
- uploadContentType 为上传的类型。这个属性的组成为{name}ContentType。  
ContentType 是固定写法。原因查看 UploadFileInterceptor 源代码。
- uploadFileName 为上传的文件名称。这个属性的组成为{name}FileName。  
FileName 为固定写法。
- 在默认情况下，上传的文件的最大限制为 2M，所以如果想上传更大的，就得改变其默认的设置。
- 在上传的时候，会在 tomcat 中的 work\Catalina\localhost\struts2 目录下保存一个临时文件，后缀名为 tmp。

default.properties 中

**struts.multipart.maxSize=2097152** 这个配置代表上传文件的最大限制。想改变如下：

第一种方式：

struts-upload.xml 中

```
<constant name="struts.multipart.maxSize" value="8097152"></constant>
```

为改变上传文件的最大限制为 8M。

第二种方式：

通过引入 FileUploadInterceptor 拦截器中的三个参数

maximumSize: 上传文件的最大限制

allowedTypes: 上传文件允许的类型

allowedExtensions: 允许上传文件的扩展名

通过这三个参数可以更加细粒度的控制上传文件。

所以我们通常采用第二种方式，但是必须注意以下几点：

- 如果采用第二种方式，第一种方式必须写。再写第二种方式，这样让第二种方式覆盖第一种方式。



- 在上传的过程中，如果出现错误，则系统会自动跳转到 input 指向的页面。这点也可以从 FileUploadInterceptor 源代码中看出来。
- 在 input 参数指定的页面中，编写<s:fielderror></s:fielderror>可以看到错误的信息。但是是英文的。如：  
不能上传一个类型错误信息为：

Content-Type not allowed: upload "xwork-2.1.6-src.zip" "upload\_6fff0830\_13174e12471\_\_8000\_00000000.tmp"  
application/x-zip-compressed

这个信息的组成查看 org.apache.struts2 包下的 struts-messages.properties 文件。在这个文件中，有三个键值对：

struts.messages.error.file.too.large=File too large: {0} "{1}" "{2}" {3}

struts.messages.error.content.type.not.allowed=Content-Type not allowed: {0} "{1}" "{2}" {3}

struts.messages.error.file.extension.not.allowed=File extension not allowed: {0} "{1}" "{2}" {3}

{0}：代表页面上<input name="upload" type="file"/>中的 name 的值

{1}：代表文件上传的名称

{2}：文件保存在临时目录的名称。临时目录为 work\Catalina\localhost\struts2

{3}：代表文件上传类型，或者文件上传大小。上面的错误代表文件上传类型。如果报第一个错误，则代表文件上传大小。

如果报错了，显示的是英文，怎样才能显示中文呢？

步骤：

1、建立一个 properties 文件。这个文件的名称可以任意取。

2、在这个配置文件中，添入如下的内容：

struts.messages.error.file.too.large=文件超过了规定的大小: {0} "{1}" "{2}" {3}

struts.messages.error.content.type.not.allowed=该类型不允许被上传: {0} "{1}" "{2}" {3}

struts.messages.error.file.extension.not.allowed=不能上传该扩展名类型的文件: {0} "{1}" "{2}" {3}

3、在 struts-upload.xml 中，指定配置文件的位置

```
<constant name="struts.custom.i18n.resources" value="cn.itcast.struts2.action.upload.fileuploadmessage"></constant>
```

如果配置文件放在 src 下，则这样指定：

```
<constant name="struts.custom.i18n.resources" value="fileuploadmessage"></constant>
```

## 19.3 多文件上传

testUploads.jsp

```
<form action="upload/uploadsAction_testUpload.action" method="post"
      enctype="multipart/form-data">
    要上传的文件:
    <input type="file" name="uploads">
    <br>
    要上传的文件:
    <input type="file" name="uploads">
    <br>
    要上传的文件:
    <input type="file" name="uploads">
    <br>
    <input type="submit">
</form>
```

注意：所有的 `type="file"` 的 `name` 属性都为 `uploads`, 所以后台获取的 `uploads` 是一个数组。

UploadsAction.java

```
public class UploadsAction extends ActionSupport{
    private File[] uploads;

    private String[] uploadsContentType;

    private String[] uploadsFileName;
    public String testUpload() throws Exception{
        String path = ServletActionContext.getServletContext().getRealPath("/upload");

        for(int i=0;i<this.uploads.length;i++){

            File destFile = new File(path,this.uploadsFileName[i]);

            FileUtils.copyFile(uploads[i], destFile);
        }
        return SUCCESS;
    }
}
```

注：这里的原理和上传一个文件的原理是一样的，不同的是 `uploads`, `uploadsContentType`, `uploadsFileName` 为数组。

## 20 高级部分