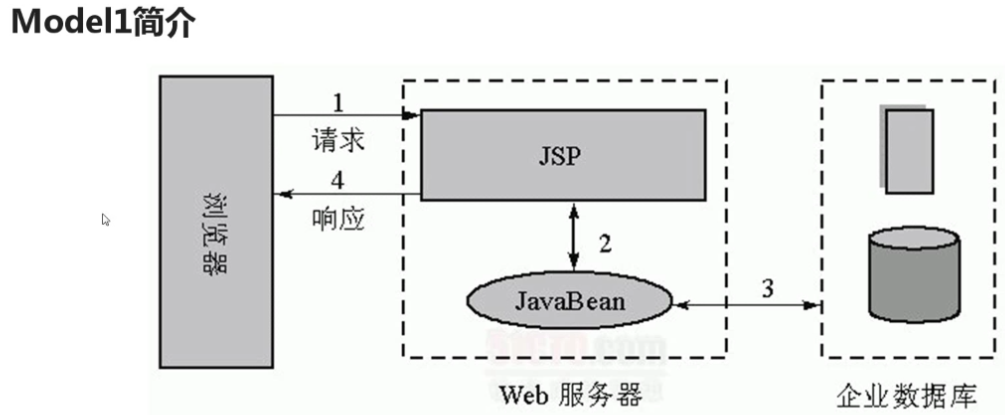
# Struts2 note

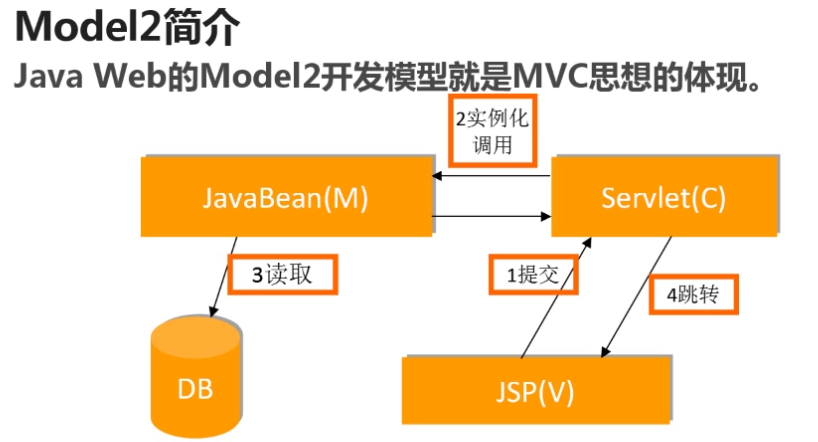
<http://www.jellythink.com/archives/category/javaframework>

## Model1&Model2&MVC

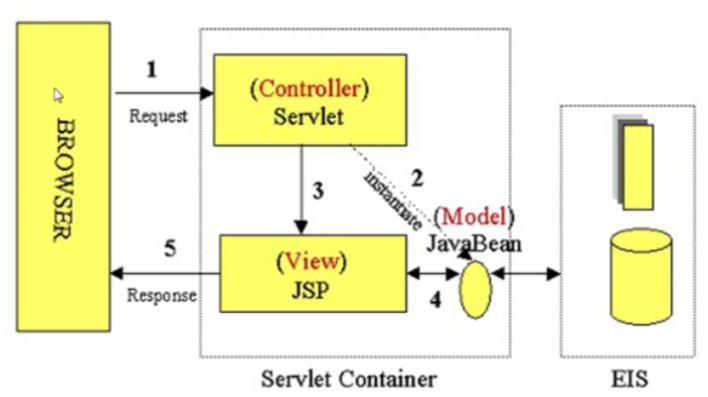
#### Model1：jsp(+JavaBean)



#### Model2：jsp+Servlet+JavaBean



#### MVC:

(EIS：Enterprise Information System)

在MVC架构中，Servlet作为前端中枢控制器（Controller），负责接收客户端发送过来的请求，在Servlet中只包含控制逻辑和简单的前端处理。Struts2就起了这样一个作用，为了最大程度的将Model、View和Controller这三部分进行解耦，从而提高应用的可扩展性及可维护性，struts2在这个当中占据着关键的Controller的位置；

## 安装

1. 将struts目录下的lib文件夹中几个关键jar包拷贝至项目WEB-INF\lib目录下：

xwork-core-2.3.24.1.jar

struts2-core-2.3.24.1.jar

ognl-3.0.6.jar

javassist-3.11.0.GA.jar

commons-io-2.2.jar

commons-lang3-3.2.jar

commons-fileupload-1.3.1.jar

freemarker-2.3.22.jar

1. 在web.xml中配置一个Filter，将所有对该项目请求拦截到struts2框架中：

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

<!-- 以下为可选项 -->

*<init-param>*

*<param-name>actionPackages</init-param><!-- 配置框架默认加载的action包结构 -->*

*<param-value>org.crazyit.auction.action</param-value>*

*<!-- 该参数的值也是一个以英文逗号隔开的字符串，每个字符串都是一个包空间， Struts 2 框架将扫描这些包空间下的 Action 类 -->*

*</init-param>*

*<init-param>*

*<param-name>configProviders</init-param><!-- 配置框架配置提供者类 -->*

*<param-value>lee.MyConfigurationProvider</param-value>*

*<!-- 如果用户需要实现自己的 ConfigurationProvider 类，用户可以提供一个或多个实现了 ConfigurationProvider 接口的类，然后将这些类的类名设置成该属性的值，多个类名之间以英文逗号隔开 -->*

*</init-param>*

*<init-param>*

*<param-name>config</init-param><!-- 配置框架加载的配置文件 -->*

*<param-value>xxx</param-value>*

*<!-- 该参数的值是一个以英文逗号 C ，)隔开的字符串，每个字符串都是一个 XML配置文件的位置。 Struts 2 框架将自动加载该属性指定的系列配置文件 -->*

*</init-param>*

</filter>

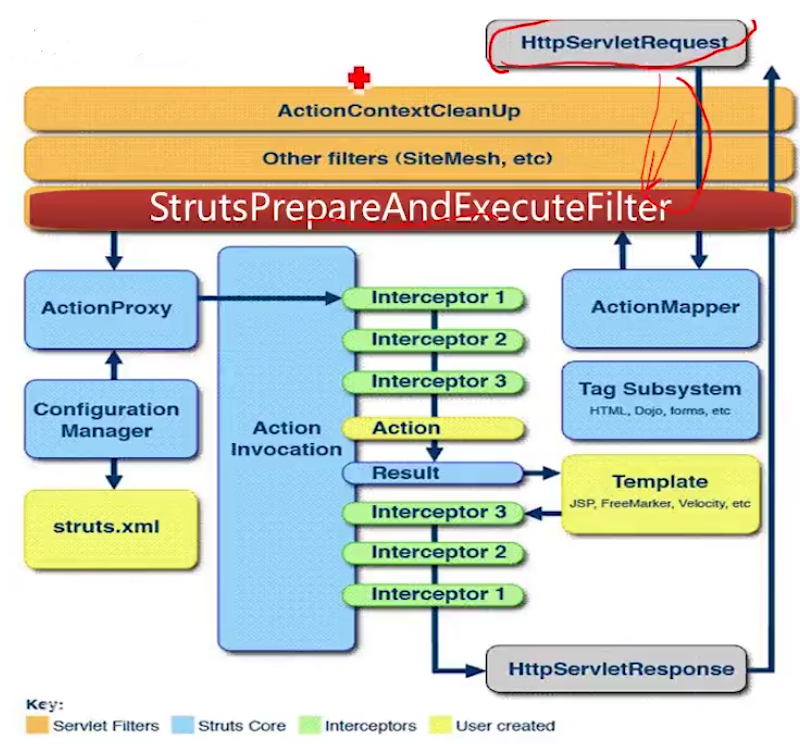
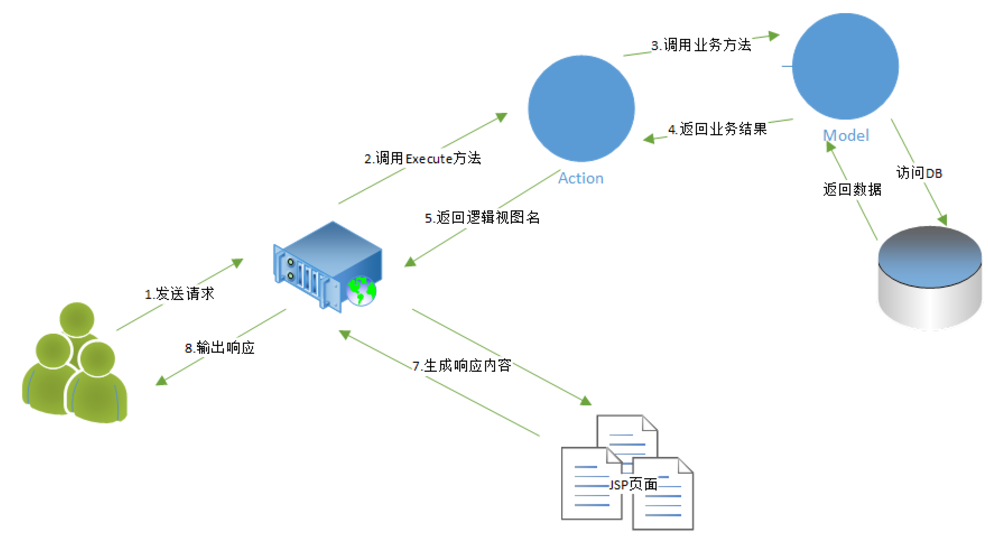
<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## 工作流程



1. 用户通过浏览器访问Java Web应用；
2. 用户的请求到达服务器以后，被配置在web.xml中的Struts2的核心Filter所拦截：

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

由于Web应用都是基于请求—响应架构的应用，所以不管哪个MVC WEB框架，都需要在web.xml中配置该框架的核心Servlet或Filter，这样才可以让该框架介入Web应用中；

1. 定义处理用户请求的Action类；这一步是非常重要的，这里定义的Action类就是MVC中的“C”，就是所谓的控制器，该控制器负责调用Model里的方法来处理请求，对于一些非常简单的请求可以在Action中直接处理；Struts2根据对应的请求，在struts.xml中找到对应的Action处理类：

<action name="login" class="com.jellythink.practise.LoginAction">

...

</action>

对于寻找对应的Action处理类，有一个Action的寻找过程：

当访问<http://localhost:8080/struts/path1/path2/path3/test.action> 时：

系统先判断是否存在package：path1.path2.path3；

1. 如果存在：
   1. 在包配置中指定的namespace中寻找Action处理类；
   2. 当在指定的namespace中找不到的情况下，则在默认namespace中去找Action处理类；
   3. 如果都没有找到，则出现系统错误404；
2. 如果不存在：
3. 判断上一级路径的package是否存在，直到默认namespace（即path1.path2和path1），当存在时，进行步骤a；
4. 如果都没有找到，则出现系统错误404；

找到了对应的Action处理类以后，就会调用对应的public String execute方法，处理业务逻辑；

1. Action根据业务返回结果，返回对应的逻辑视图名；

例如下列代码中的return SUCCESS和return ERROR：

**public String execute() throws Exception**

{

if (getUserName().equals("jellythink")

&& getPassword().equals("123456"))

{

ActionContext.getContext().getSession().put("user", getUserName());

return SUCCESS;

}

else

{

return ERROR;

}

}

这里的SUCCESS和ERROR都是字符串常量，对应的是:success和error；

1. Struts2根据Action返回的逻辑视图名，在配置文件中对应物理视图资源；

例如：

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="input">/login.jsp</result>

<result name="error">/error.jsp</result>

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

</action>

当Action控制器返回input字符串时，则进入/login.jsp页面；当返回error字符串时，则进入/error.jsp页面；当返回success字符串时，则进入/success.jsp页面。

通过这种方法，将View和Controller进行解耦；

1. 服务器将对应的页面响应给用户，从而完成一次请求—响应；

## 配置文件

为了尽最大程度、最大努力的解耦每个模块，各个模块之间的关联基本上都是通过配置文件来完成的，这也是Struts2这个框架设计的关键；

#### 几个关键配置文件

##### struts-default.xml

Struts2框架的基本/默认配置文件，主要用于配置很多基本配置项的默认值；一般情况下，我们几乎不会直接和这个文件打交道；

位置：使用解压缩工具查看struts2-core-2.3.24.1.jar，可以发现struts-default.xml配置文件；

例：Action的默认处理类就是在该配置文件中配置的：

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

##### struts-plugin.xml

Struts2框架插件的配置文件，由于Struts2框架有很多丰富的插件，所以很多在struts-default.xml配置文件中定义的默认值对于这些插件来说，已经不能胜任了，所以只好在插件的struts-plugin.xml配置文件中覆盖它们；

位置：在Struts2的安装包的lib/目录下，可以发现很多类似这样struts2-xxx-plugin-xxx.jar命名的Jar包，这些就是struts插件，使用解压缩工具查看这些Jar包，就会发现struts-plugin.xml配置文件；

##### struts.properties

struts.properties文件是一个标准的Properties文件，该文件包含了一系列的key-value对，每个key就是一个Struts2常量，该key对应的value就是一个Struts2的常量值，如：

struts.multipart.parser=jakarta

# uses javax.servlet.context.tempdir by default

struts.multipart.saveDir=

struts.multipart.maxSize=2097152

该文件定义了Struts 2框架的大量属性，开发者可以通过改变这些属性来满足应用的需求；

位置：在struts2-core-2.3.24.1.jar包的\org\apache\struts2目录下，有一个default.properties文件；web项目中 struts.properties文件通常放在Web应用的WEB-INF/classes路径下。实际上，只要将该文件放在Web应用的CLASSPATH路径下，Struts 2框架就可以加载该文件。

* **常用的配置项**：（配置项详解看<https://www.cnblogs.com/pigtail/archive/2013/02/12/2910348.html> ）

**一般修改配置项都不会直接在struts.properties文件中修改，而是在struts.xml中通过<constant . . . />标签覆盖来修改；**

struts.i18n.encoding：指定Web应用的默认编码集(相当于调用HttpServletRequest的setCharacterEncoding方法）。该属性对于处理中文请求参数非常有用，对于获取中文请求参数值，应该将该属性值设置为GBK或者GB2312，默认值是UTF-8；

struts.i18n.reload：该属性设置是否每次HTTP请求到达时，系统都重新加载国际化资源文件。该属性默认值是false。在开发阶段将该属性设置为true会更有利于开发，但在产品发布阶段应将该属性设置为false。（开发阶段将该属性设置了true，将可以在每次请求时都重新加载国际化资源文件，从而可以让开发者看到实时开发效果；产品发布阶段应该将该属性设置为false，是为了提供响应性能，每次请求都需要重新加载资源文件会大大降低应用的性能）；

struts.configuration.xml.reload：该属性设置当struts.xml文件改变后，系统是否自动重新加载该文件。该属性的默认值是false，在开发阶段建议设置未true；

struts.devMode：该属性设置Struts 2应用是否使用开发模式。如果设置该属性为true，则可以在应用出错时显示更多、更友好的出错提示。该属性只接受true和flase两个值，该属性的默认值是false。通常，应用在开发阶段，将该属性设置为true，当进入产品发布阶段后，则该属性设置为false；

struts.serve.static.browserCache：该属性设置浏览器是否缓存静态内容。当应用处于开发阶段时，我们希望每次请求都获得服务器的最新响应，则可设置该属性为false，默认为true；

struts.action.extension：该属性指定需要Struts 2处理的请求后缀，该属性的默认值是action，即所有匹配\*.action的请求都由Struts 2处理。如果用户需要指定多个请求后缀，则多个后缀之间以英文逗号（,）隔开，如：“struts.action.extension=action,do,html,“；不做修改时默认支持”.action“和不加后缀访问；

struts.url.http.port：该属性指定Web应用所在的监听端口，一般情况下不做修改，只是当Struts 2需要生成URL时（例如Url标签），该属性才提供Web应用的默认端口，默认值80；

struts.url.https.port：该属性类似于struts.url.http.port属性的作用，区别是该属性指定的是Web应用的加密服务端口；

##### struts.xml

Struts2框架的核心配置文件，在开发过程中利用率最高。用户请求和Action的映射、逻辑视图result与物理视图的映射、全局常量值、各种拦截器的配置都在这个配置文件中完成。有时候，在读别人的代码时，也会根据struts.xml中的配置，理清逻辑思路；

位置：WEB-INF/classes/

文件结构：

struts2.0中struts.xml配置文件详解 <https://www.cnblogs.com/kay/archive/2007/11/28/976120.html>

<?xml version="1.0" encoding="GBK"?>  
<!-- 下面指定Struts 2配置文件的DTD信息 -->  
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN" "<http://struts.apache.org/dtds/struts-2.0.dtd>">  
<!-- struts是Struts 2配置文件的根元素 -->  
<struts>  
 <constant name="" value="" />  
 <bean type="" name="" class="" scope="" static="" optional="" />  
 <include file="" />  
 <!-- package元素是Struts配置文件的核心，该元素可以出现0次，或者无限多次 -->  
 <package name="必填的包名" extends="" namespace="" abstract="" externalReferenceResolver>  
 <result-types>  
 <result-type name="" class="" default="true|false">  
 <param name="参数名">参数值</param>\*  
 </result-type>  
 </result-types>  
  
 <interceptors>  
 <!-- 该元素的interceptor元素和interceptor-stack至少出现其中之一，也可以二者都出现 -->  
 <interceptor name="" class="">  
 <param name="参数名">参数值</param>\*  
 </interceptor>  
 <interceptor-stack name="">  
 <interceptor-ref name="">  
 <param name="参数名">参数值</param>\*  
 </interceptor-ref>  
 </interceptor-stack>  
 </interceptors>  
  
 <default-interceptor-ref name="">  
 <param name="参数名">参数值</param>  
 </default-interceptor-ref>  
 <default-action-ref name="">  
 <param name="参数名">参数值</param>\*  
 </default-action-ref>?  
 <global-results>  
 <result name="" type="">  
 <param name="参数名">参数值</param>\*//映射资源  
 </result>  
 </global-results>  
  
 <global-exception-mappings>  
 <exception-mapping name="" exception="" result="">异常处理资源  
 <param name="参数名">参数值</param>\*  
 </exception-mapping>  
 </global-exception-mappings>

**<action name="" class="" method="" converter="">** <param name="参数名">参数值</param>\* **<result name="" type="">** <param name="参数名">参数值</param>\*

**映射资源，如/login.jsp**

**</result>** <interceptor-ref name="">  
 <param name="参数名">参数值</param>\*  
 </interceptor-ref>  
 <exception-mapping name="" exception="" result="">  
 <param name="参数名">参数值</param>\*

异常处理资源  
 </exception-mapping> **</action>**

</package>\*  
<struts>

#### Bean配置

#### 常量配置

##### 全局常量配置

在Structs2框架中配置全局的常量，有以下几种方式：

1. 在struts.xml中配置常量：

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

1. 在struts.properties中配置常量(该文件专门用于配置常量)：

struts.custom.i18n.resources=jellythink

1. web.xml中配置核心Filter时通过初始化参数来配置常量：

<init-param>

<param-name>struts.custom.i18n.resources</init-param>

<param-value>jellythink</param-value>

</init-param>

通常，在struts2框架中是按照如下顺序依次加载struts2常量的：

struts-default.xml -> struts-plugin.xml -> struts.xml -> struts.properties -> web.xml

如果在多个文件中配置了同一个Struts2常量，则后一个文件中配置的常量值会覆盖前面文件中配置的常量值；

#### 包配置

Struts2框架中核心组件是Action、拦截器等，而Struts2框架使用包来管理Action和拦截器，每个包就是多个Action、拦截器的集合；

在struts.xml 文件中， package 标签用于定义包配置，每个 package定义了一个包配置。定义 package 元素时可以指定以下四个属性:

* name：必须属性，表示包的名字，其它包可以通过该属性引用该包
* extends：可选属性，表示该包继承自其它包；通过继承其它包，可以继承其它包中的Action定义、拦截器定义等；其中，struts-default包是在struts2-core-2.3.24.1.jar里定义的一个抽象包，它里面包含了大量结果类型定义、拦截器定义等，这些定义是配置普通Action的基础，所以我们在定义package时，都应该继承struts-default包
* namespace：可选属性，表示该包的命名空间，具体请参见命名空间小节
* abstract：可选属性，表示该包是否为抽象包，抽象包中不能包含Action定义

#### 命名空间配置

为了处理同一个Web应用中包含同名Action的情形（如同C++）。Struts2以命名空间的方式来管理Action，虽然同一个命名空间里不能有同名的Action，但是不同的命名空间里可以有同名的Action；namespace相当于模块的作用，它允许以模块化的方式来组织Action.

例：

<package name="lee" extends="struts-default">

<action name="login" class="com.jellythink.practise.LoginAction">

. . .

</action>

</package>

则访问地址为：<http://localhost:8080/Struts2Demo/login.action>

<package name="lee" extends="struts-default" **namespace="/Login"**>

...

</package>

则访问地址变为：<http://localhost:8080/Struts2Demo/Login/login.action>

当指定了命名空间以后，在前台页面中的FORM表单也要指定对应的命名属性，如下：

<s:form action="login" **namespace="/Login"**>

<s:textfield name="userName" key="user" />

<s:password name="password" key="pass" />

<s:submit key="login" />

</s:form>

#### 包含配置

##### 模块化管理——配置文件包含机制

在开发大型应用的时候。随着Action的增多，配置项随之增多，struts.xml文件越来越大，为解决此问题，Struts2提供了配置文件包含机制：

可以将struts.xml文件分成struts-part1.xml、struts-part2.xml(子配置文件命名没有要求)等子配置文件，如：

struts-part1.xml代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"

"http://struts.apache.org/dtds/struts-2.1.7.dtd">

<struts>

...

</struts>

在struts.xml文件中包含其他配置文件：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"

"http://struts.apache.org/dtds/struts-2.1.7.dtd">

<struts>

<!-- 通过include元素导入其他配置文件 -->

**<include file="struts-part1.xml" />**

...

</struts>

struts.xml文件由Struts2框架负责自动加载，从而加载到所有子配置文件；

#### 拦截器配置

#### 通配符的使用

在配置<action .../>元素时，允许在指定name属性时使用模式字符串（即用“\*”代表一个或多个任意字符），接下来就可以在class、method属性及<result .../>子元素中使用{N}的形式来代表前面第N个星号所匹配的子串；

例1：如果用户请求的URL为loginAction.action，则调用对应的login方法；如果为registAction.action，则调用regist方法

<struts>

<constant name="struts.enable.DynamicMethodInvocation" value="true" />

<package name="lee" extends="struts-default" namespace="/">

<action name="\*Action" class="com.jellythink.practise.LoginAction" method="{1}">

<result name="input">/{1}.jsp</result>

<result name="error">/error.jsp</result>

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

</action>

</package>

</struts>

例2：**官方推荐使用**

<struts>

<constant name="struts.enable.DynamicMethodInvocation" value="true" />

<package name="lee" extends="struts-default" namespace="/">

<action name="**\*\_\***" class="com.jellythink.practise**.{1}Action**" method**="{2}">**

<result name="input">/**{2}.jsp**</result>

<result name="error">/error.jsp</result>

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

</action>

</package>

</struts>

在上述配置代码中，action的name属性为\*\_\*，这需要我们在前台页面设置action的值时按照指定的规则进行设定，例如：类名\_方法名。当用户请求的URL是Login\_login.action的模式时，则{1}代表的值为Login，而对应的{2}代表的值为login；同理，当请求为Regist\_regist.action的模式时，则{1}代表的值为Regist，而对应的{2}代表的值为regist。

通过这种方式，配置代码的灵活性更高，写法更简单；

注意：

1. 若出现了多个带有\*的action定义，\*Action并不会比\*更优先匹配getBookInfoAction的请求，这取决于这两个Action定义的先后顺序；
2. 在配置<result .../>元素时，除了可以使用上述的模式匹配方式以外，还可以使用${属性名}的方式来指定视图资源。在${属性名}这种方式下，${属性名}里的属性名就是对应Action实例里的属性。而且，不仅允许使用这种简单表达式形式，还可以使用完全的OGNL表达式，即以下这种形式：

${属性名.属性名}；

## Action

对于Struts2的项目而言，Action承担着Controller的角色，是项目的核心；

### Action类和ActionSupport类

为了让开发的Action类更规范化，统一execute方法的返回值，Struts2提供了一个Action接口，这个接口定义了Struts2的Action处理类应该实现的规范：

package com.opensymphony.xwork2;

public interface Action {

public static final String SUCCESS = "success";

public static final String NONE = "none";

public static final String ERROR = "error";

public static final String INPUT = "input";

public static final String LOGIN = "login";

public String execute() throws Exception;

}

而ActionSupport是一个默认的Action实现类，它提供了许多默认方法，这些默认方法包括获取国际化信息的方法、数据校验的方法、默认的处理用户请求的方法等；我们在编写Action类时，如果继承了ActionSupport类，则会大大简化Action的开发：

/\*\*

\* Provides a default implementation for the most common actions.

\*/

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

protected static Logger LOG = LoggerFactory.getLogger(ActionSupport.class);

private final ValidationAwareSupport validationAware = new ValidationAwareSupport();

private transient TextProvider textProvider;

private Container container;

public void setActionErrors(Collection<String> errorMessages) {

validationAware.setActionErrors(errorMessages);

}

public Collection<String> getActionErrors() {

return validationAware.getActionErrors();

}

public void setActionMessages(Collection<String> messages) {

validationAware.setActionMessages(messages);

}

public Collection<String> getActionMessages() {

return validationAware.getActionMessages();

}

...

}

由于ActionSupport实现了Action接口，完全符合一个Action类的编写要求，所以我们可以直接使用ActionSupport作为业务控制器；实际上，如果我们在struts.xml中配置Action时，没有指定class属性，系统将默认使用ActionSupport类作为Action处理类；

### Action的实现

1. 编写Action：
2. 通常继承于ActionSupport类（为了最大程度的解耦和重用代码，Struts2没有强制要求Action类必须继承某个基类或者接口，就是说即便不继承，仍旧是合法的Action类）；
3. 包含与请求参数相对应的private属性，并提供setXxx和getXxx方法；
4. 实现处理客户端请求的execute()方法，不带参数，返回处理结果字符串；
5. 在struts.xml中注册Action：

例：

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="input">/login.jsp</result>

<result name="error">/error.jsp</result>

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

</action>

* <action... />有以下三个属性：
  + name：必须属性，外部就是通过这个名字找到对应的Action；
  + class：可选属性，该属性指定了Action的实现类；如果不指定该属性，系统则默认使用ActionSupport类作为处理用户请求的Action；
  + method：（可选）指定method属性，这样就可以让Action调用指定方法，而不是默认的execute方法来处理用户请求（默认值是execute）；
* <result... />元素定义了逻辑视图和物理视图之间的映射关系，每个<result... />元素定义逻辑视图和物理视图之间的一次映射。对于<result... />的name属性就是execute方法返回的结果字符串，对于<result... />的值就是对应的物理视图。“/“表示项目根目录。

### 访问Action

#### URL访问：<http://localhost:8080/项目名/包名层次/xxx.action> （[后缀可通过修改struts.action.extension常量修改](#_struts.properties)）

#### 在JSP中

例：form提交到Action

<s:form **action="LoginAction" namespace="/"**>

<s:textfield name="userName" key="user" />

<s:password name="password" key="pass" />

<s:submit key="login" value="登陆"></s:submit>

<s:submit key="regist" value="注册" onclick="registFunc()"></s:submit>

</s:form>

### Action访问ServletAPI

<http://www.jellythink.com/archives/1451>

为使Struts2的项目充分解耦，Action没有与任何Servlet API耦合，因此，无法在Action中直接访问ServletAPI，通常在Action中访问ServletAPI有以下三种方式：

#### 通过ActionContext类访问

Struts2提供了ActionContext类，Struts2的Action可以通过该类来访问Servlet API。下面是ActionContext类中几个常用的方法：

* static ActionContext getContext()：静态方法，获取系统的ActionContext实例（使用ActionServlet类的前提是必须先通过该静态方法获取该类对象ActionContext.getContext()）
* Object get(Object key)：该方法类似于调用HttpServletRequest的getAttribute(String name)方法
* Map getApplication()：返回一个Map对象，该对象模拟了该应用的ServletContext实例
* Map getParameters()：获取所有的请求参数；类似于调用HttpServletRequest对象的getParameterMap()方法
* Map getSession()：返回一个Map对象，该Map对象模拟了HttpSession实例
* void setApplication(Map application)：直接传入一个Map实例，将该Map实例里的key-value对转换成application的属性名、属性值
* void setSession(Map session)：直接传入一个Map实例，将该Map实例里的key-value对转换成session的属性名、属性值

#### 实现“\*Aware“接口后在Action中直接访问

为了可以在Action中直接访问Servlet API，Struts2还提供了以下几个接口：

|  |  |
| --- | --- |
| 接口名称 | 接口说明 |
| ServletContextAware | 实现该接口的Action可以直接访问Web应用的ServletContext实例 |
| ServletRequestAware | 实现该接口的Action可以直接访问Web应用的HttpServletRequest实例 |
| ServletResponseAware | 实现该接口的Action可以直接访问服务器响应的HttpServletResponse实例 |

只需在定义Action时，实现这几个接口，即可获取原生的servlet实例，例：

public class LoginAction extends ActionSupport

implements ServletResponseAware, ServletRequestAware, ServletContextAware

{

private String userName;

private String password;

private HttpServletResponse response;

private HttpServletRequest request;

private ServletContext sc;

// 实现ServletResponseAware接口的方法，设置response属性值

public void setServletResponse(HttpServletResponse response)

{

this.response = response;

}

// 实现ServletRequestAware接口的方法，设置request属性值

public void setServletRequest(HttpServletRequest request)

{

this.request = request;

}

// 实现ServletContextAware接口的方法，设置sc属性值

public void setServletContext(ServletContext sc)

{

this.sc = sc;

}

public String execute() throws Exception

{

// ......

}

#### 通过ServletActionContext访问

为了能直接访问Servlet API，Struts2还提供了一个ServletActionContext工具类，该工具类提供了以下几个静态方法：

|  |  |
| --- | --- |
| 声明 | 说明 |
| static PageContext getPageContext() | 取得Web应用的PageContext对象 |
| static HttpServletRequest getRequest() | 取得Web应用的HttpServletRequest对象 |
| static HttpServletResponse getResponse() | 取得Web应用的HttpServletResponse对象 |
| static ServletContext getServletContext() | 取得Web应用的ServletContext对象 |

但通过这种方式，会加强Action和Servlet API的耦合，不利于Action的再利用。

#### 比较

|  |  |  |
| --- | --- | --- |
| 方法 | 优点 | 缺点 |
| 通过ActionContext类访问 | Action和Servlet API完全解耦 | 并不能调用原生的Servlet API |
| Action直接访问 | 可以调用原生的Servlet API | 需要实现对应的接口 |
| 通过ServletActionContext访问 | 可以调用原生的Servlet API | 增加了耦合 |

### 单个Action多控制处理逻辑

#### 动态方法调用**（不推荐使用）**

动态方法调用是指表单元素的action并不是直接等于某个Action的名字，而是以如下形式来指定表单的action属性。

action="ActionName!methodName"

其中ActionName指定提交到哪个Action，而methodName指定提交到指定方法；

使用动态方法调用前必须设置Struts2允许动态方法调用。将struts.enable.DynamicMethodInvocation设置为true开启动态方法调用；否则将关闭动态方法调用（默认为关闭状态）。在struts.xml中添加以下配置即可：

<constant name="struts.enable.DynamicMethodInvocation" value="true" />

例：

通过前台页面控制action的属性值：

<body>

<s:form action="login" namespace="/">

<s:textfield name="userName" key="user" />

<s:password name="password" key="pass" />

<s:submit key="login" value="登陆"></s:submit>

<s:submit key="regist" value="注册" onclick="registFunc()"></s:submit>

</s:form>

<script type="text/javascript">

function registFunc(){

// 获取页面第一个表单

var targetForm = document.forms[0];

// 动态修改表单的action属性

**targetForm.action = "login!regist";//若用url访问，为**<http://xxx/xxx/login!regist.action>

}

</script>

</body>

对于后台Action，则添加一个regist方法即可，方法的声明和execute方法一模一样（public、返回String、无参）：

public String regist() throws Exception

{

return "regist";

}

#### 在struts.xml中指定method属性

Struts2还提供了另外一种方法，就是将一个Action类配置成多个逻辑Action，具体的做法是在配置<action .../>，可以为它指定method属性，这样就可以让Action调用指定方法，而不是默认的execute方法来处理用户请求，但前提是这些方法声明要满足**public、返回String、无参**的条件；

例1：

struts.xml配置如下：

（配置的两个Action，对应的class为同一个class，只是Action名和method不一样）

<struts>

<constant name="struts.enable.DynamicMethodInvocation" value="true" />

<package name="lee" extends="struts-default" namespace="/">

**<action name="loginAction" class="com.jellythink.practise.LoginAction" method="login">**

<result name="input">/login.jsp</result>

<result name="error">/error.jsp</result>

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

</action>

**<action name="registAction" class="com.jellythink.practise.LoginAction" method="regist">**

<result name="regist">/regist.jsp</result>

</action>

</package>

</struts>

前台JSP页面代码如下：

<body>

<s:form action="**loginAction**" namespace="/">

<s:textfield name="userName" key="user" />

<s:password name="password" key="pass" />

<s:submit key="login" value="登陆" onclick="loginFunc()"></s:submit>

<s:submit key="regist" value="注册" onclick="registFunc()"></s:submit>

</s:form>

<script type="text/javascript">

function loginFunc(){

var targetForm = document.forms[0];

targetForm.action = "**loginAction**";

}

function registFunc(){

var targetForm = document.forms[0];

targetForm.action = "**registAction**";

}

</script>

</body>

#### 直接在<s:submit .../>中指定method

Action中直接定义login方法和regist方法即可？

<body>

<s:form action="**loginAction**" namespace="/">

<s:textfield name="userName" key="user" />

<s:password name="password" key="pass" />

<s:submit key="login" value="登陆" **method="login"**></s:submit>

<s:submit key="regist" value="注册" **method="regist"**></s:submit>

</s:form>

</body>

struts.xml配置如下：

<package name="lee" extends="struts-default" namespace="/">

<action name="loginAction" class="com.jellythink.practise.LoginAction">

<result name="input">/login.jsp</result>

<result name="error">/error.jsp</result>

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

<result name="regist">/regist.jsp</result>

</action>

</package>

### 配置默认Action

即使用户的请求找不到对应的Action时，存在一个默认的Action可以来接管用户的请求，这是我们希望看到的。在Struts2中，可以通过配置<default-action-ref ... />设置默认的Action：

<struts>

<package name="lee" extends="struts-default">

**<default-action-ref name="default" />**

<action **name="default"** class="com.jellythink.practise.LoginAction">

<result name="default">/default.jsp</result>

</action>

</package>

</struts>

注意：将默认Action配置在默认命名空间里就可以让该Action处理所有用户请求，因为默认命名空间的Action可以处理任何命名空间的请求；

另：配置Action的默认处理类，在struts2-core-2.3.24.1.jar中的struts-default.xml文件中：

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

### 接收参数

#### request对象

在Action中访问ServletAPI后，即可通过request对象接收请求的参数；

#### Action属性

在Action类中定义要接收的属性（如：private String username;），并实现所有属性的set/get方法，当使用URL或在JSP中提交form访问该Action时，params拦截器会自动为所有与请求参数命名对应的属性赋值；

#### DomainModel

将要接收的参数封装成一个标准的JavaBean实体类，再在Action中声明该实体类为Action类的属性（不需要实例化），并实现set/get方法；在前端提交请求时指定好实体类，即可为所有与请求参数命名对应的属性赋值；

例：

将username 和password两个属性封装为一个类User（必须是标准的JavaBean）；

在LoginAction中声明属性（不需要实例化）：private User user; 同时为user设置getter和setter方法；

JSP中form的input标签的name需要设置为user.name和user.password,才能映射成功；

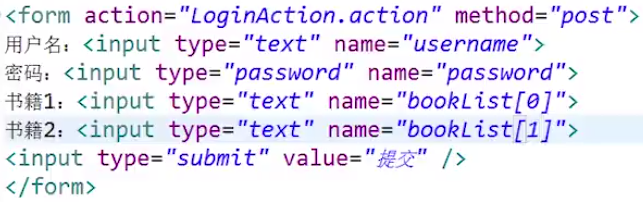
缺点：前后端耦合加强，若实体类名user发生改变，前端input标签的name都需要修改；

#### ModelDriver（推荐使用）

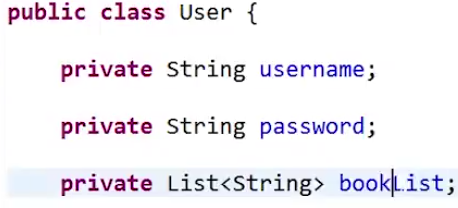
将要接收的参数封装成一个标准的JavaBean实体类，再在Action中声明该实体类为Action类的属性（必须实例化，不用实现set/get方法），同时使Action类实现ModelDriver<实体类名>接口，并实现这个接口的public T getModel()方法，；在前端提交请求时不许需要再指定实体类，即可为所有与请求参数命名对应的属性赋值；解决了DomainModel方法的前后端耦合问题；

例：

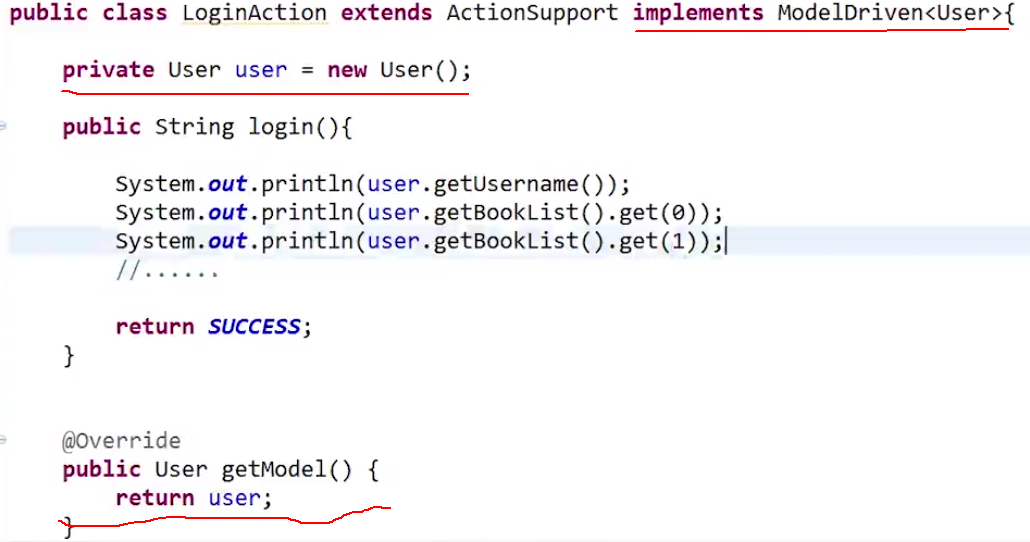
前端页面：



封装要接收的参数成一个实体类：



在Action中使用ModelDriver方式接收参数：



## 处理结果

结果类型决定了Action处理结束后，下一步将调用哪种视图资源来呈现处理结果。Struts2的结果类型要求实现com.opensymphony.xwork2.Result，这个结果是所有结果类型的通用接口。如果我们需要自己的结果类型（一般不需要），我们应该提供一个实现该接口的类，并在struts.xml文件中配置该结果类型；

### 所有支持的结果类型

<result-types>

<!--Action链式处理的结果类型-->

<result-type name="chain" class="com.opensymphony.xwork2.ActionChainResult"/>

<!--用于与JSP整合的结果类型，作为默认的结果类型-->

<result-type name="dispatcher" class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true"/>

<!--用于与FreeMarket整合的结果类型-->

<result-type name="freemarker" class="org.apache.struts2.views.freemarker.FreemarkerResult"/>

<!--用户控制特殊的HTTP行为的结果类型-->

<result-type name="httpheader" class="org.apache.struts2.dispatcher.HttpHeaderResult"/>

<!--用于直接跳转到其它URL的结果类型-->

<result-type name="redirect" class="org.apache.struts2.dispatcher.ServletRedirectResult"/>

<!--用于直接跳转到其它Action的结果类型-->

<result-type name="redirectAction" class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>

<!--用于向浏览器返回一个InputStream的结果类型-->

<result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>

<!--用于整合Velocity的结果类型-->

<result-type name="velocity" class="org.apache.struts2.dispatcher.VelocityResult"/>

<!--用于整合XML/XSLT的结果类型-->

<result-type name="xslt" class="org.apache.struts2.views.xslt.XSLTResult"/>

<!--用于显示某个页面原始代码的结果类型-->

<result-type name="plainText" class="org.apache.struts2.dispatcher.PlainTextResult" />

<!--将请求参数以form的形式提交到指定地点的结果类型-->

<result-type name="postback" class="org.apache.struts2.dispatcher.PostbackResult" />

</result-types>

### 结果类型作用域

#### 局部结果

将<result .../>作为<action .../>元素的子元素配置，例：

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="default">/default.jsp</result>

</action>

实际上，上边的这种写法简化的写法，完整写法如下：

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="success" type="dispatcher">

<param name="location">/success.jsp</param>

</result>

</action>

#### 全局结果

将<result .../>作为<global-results .../>元素的子元素配置，例：

<package name="lee" extends="struts-default" namespace="/Login" >

<!--定义全局结果-->

<global-results>

<result name="error">/error.jsp</result>

</global-results>

<action name="login" class="com.jellythink.practise.LoginAction" />

</package>

注：如果一个Action里包含了与全局结果里同名的结果，则Action里的局部Result会覆盖全局的Result；

### dispatcher结果类型

dispatcher结果类型是Struts2框架中默认的结果类型，dispatcher主要用于返回JSP，HTML等以页面为基础View视图。在使用dispatcher时，唯一需要指定的是JSP或者HTML页面的位置，这个位置将被用于定位返回的页面；

### redirect结果类型

dispatcher和redirect的主要差别就是转发和重定向的差别：重定向会丢失所有的请求参数、请求属性、Action的处理结果；

使用redirect结果类型的效果是，系统将调用HttpServletResponse.sendRedirect(String URL)方法来重定向指定视图资源，这种重定向的效果就是重新产生一个请求，因此所有的请求参数、请求属性、Action实例和Action中封装的属性全部丢失；

例：

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="error" **type="redirect"**>/error.html?info=${info}</result>//在重定向中需要携带参数

</action>

### redirectAction结果类型

当需要让一个Action处理结束后，直接将请求重定向（是重定向，而不是转发）到另一个Action时，应该使用redirectAction结果类型；

例：

<package name="Lee" extends="struts-default" namespace="/Login" >

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="error" **type="redirectAction"**>

<param name="actionName">RedirectActionTest</param>

<param name="namespace">/Test</param>

<param name="userName">${userName}</param>

<param name="password">${password}</param>

</result>

</action>

</package>

<package name="RedirectAction" extends="struts-default" namespace="/Test">

<action name="RedirectActionTest" class="com.jellythink.practise.RedirectAction">

<result name="error">/error.jsp</result>

</action>

</package>

### chain结果类型

chain结果类型是将在一个Action处理完成以后，将请求再**转发**至另一个Action（请求参数都不会丢失）；

例：

<package name="Lee" extends="struts-default" namespace="/Login" >

<action name="login" class="com.jellythink.practise.LoginAction">

<result name="error" **type="chain"**>

<param name="actionName">chainTest</param>

<param name="namespace">/Test</param>

</result>

</action>

</package>

<package name="Chain" extends="struts-default" namespace="/Test">

<action name="chainTest" class="com.jellythink.practise.ChainAction">

<result name="error">/error.jsp</result>

</action>

</package>

### input结果类型

当返回INPUT类型时，系统先判断前端表单是否使用了<s:form>（即struts内置标签），若使用的是<s:form>，则将addFileError的错误信息直接显示在表单上方（FileError的表现形式：在jsp页面中使用<s:fielderror/>标签，该标签name属性为addFieldError方法中的参数fieldName），若使用的是普通表单标签，则将请求转发到Action定义的input结果页面，若没有提供name值为input的<result/>，那么发生错误时，将直接在浏览器中提示404错误；

Action返回input类型的情况：

1. 当参数类型转换错误时（如age输入框（要求输入int）中的类型是字母），Action自动返回input；
2. Action的validate方法中中存在addFiledError方法时，Action返回input；
3. Action的非validate方法中存在addFiledError方法，且其后有“return INPUT“语句时，Action返回input；

## 异常处理（未完成）

<http://www.jellythink.com/archives/1507>

在重写execute方法时，从它的定义中可以看出：

public String execute() throws Exception

execute方法可以抛出全部异常，这也意味着我们完全无须进行任何异常处理，而是把异常直接抛给Struts2框架处理即可；Struts2框架接收到Action抛出的异常之后，将根据struts.xml文件配置的异常映射，转入指定的视图资源；

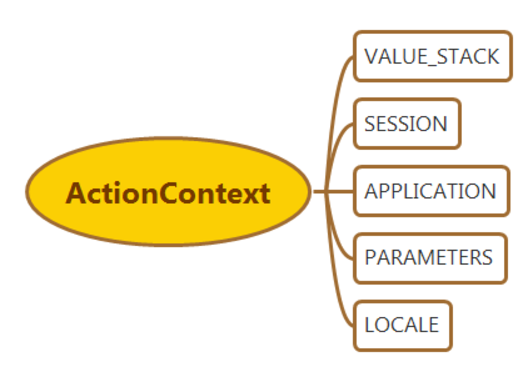
## OGNL

OGNL(Object Graphic Navigation Language，对象图导航语言) 是一个开源的表达式引擎。通过使用OGNL的表达式语法可以存取Java对象树的任意属性和调用Java对象树的方法等；

Struts2框架使用OGNL作为默认的表达式语言；

### ActionContext

OGNL的操作都是基于ActionContext类而进行的，ActionContext表示的是Action的上下文环境，它封装了一个Action运行所需要的环境，比如session、parameters、locale等，对于ActionContext来说，每一个请求对应一个Action，因此不用担心线程问题；



在ActionContext封装的环境中，ValueStack被称为值栈，是对OGNL的扩展，Struts2就是通过ValueStack来使用OGNL进行赋值和取值操作的。ValueStack内部封装了一个CompoundRoot类型的对象作为root属性，CompoundRoot是一个继承ArrayList的栈存储结构。而所有被压入栈中的对象，都会被视为OGNL的Root对象。在使用OGNL计算表达式时，首先会将栈顶元素作为Root对象，进行表达式匹配，匹配不成功则会依次向下匹配，最后返回第一个成功匹配的表达式计算结果。因此，Struts2通过ValueStack实现了多Root对象的OGNL操作。

当你提交一个请求，会为这个请求创建一个和web容器交互的ActionContext，与此同时会创建ValueStack，并置于ActionContext之中。而实例化Action之后，就会将这个action对象压入ValueStack中。在请求“映射”过程中，Struts2则是通过ParametersInterceptor拦截器将提交的参数值封装入对应的Action属性中。因此action实例可以作为OGNL的Root对象，对于Action中的属性、方法都可以使用OGNL来获取。

### 基本操作

1. 访问ValueStack

由于ValueStack是Struts2中OGNL的根对象，如果用户需要访问值栈中的对象，在JSP页面可以直接通过下面的OGNL表达式访问ValueStack(值栈)中对象的属性：

<s:property value="userName" />

在ValueStack中处于第一位的对象叫栈顶对象。通常我们在OGNL表达式里直接写上属性的名称即可访问ValueStack变量里对象的属性，搜索顺序从栈顶对象开始寻找，依次往下访问，直到找到为止；

1. 访问application对象

访问ValueStack以外其它对象时，由于他们不是根对象，所以在访问时，需要添加#前缀，相当于调用ServletContext.getAttribute("userName")方法：

<s:property value="#application.userName" />

<s:property value="#application['userName']" />

1. 访问session对象

session对象用于访问HttpSession的属性，相当于调用HttpSession.getAttribute("userName")方法:

<s:property value="#session.userName" />

<s:property value="#session['userName']" />

1. 访问request对象

request对象用于访问HttpServletRequest的属性，相当于调用HttpServletRequest.getAttribute("userName")方法：

<s:property value="#request.userName" />

<s:property value="#request['userName']" />

1. 访问parameters对象

parameters对象用于访问HTTP请求参数，相当于调用HttpServletRequest.getParameter("userName")方法：

<s:property value="#parameters.userName" />

<s:property value="#parameters['userName']" />

1. 访问attr对象

attr对象依次搜索以下对象中的属性，找到便返回：

PageContext

HttpServletRequest

HttpSession

ServletContext

### 集合操作

#### 创建集合

例：

<!-- 创建一个List：{value1, value2, value3} -->

<s:set name="varList" value="{'www.google.com','www.facebook.com','www.jellythink.com'}" />

<s:iterator value="#varList" id="n">

<s:property value="n"/><br>

</s:iterator>

<!-- 创建一个Map：#{key1:value1,key2:value2,key3:value3} -->

<s:set name="varMap" value="#{'谷歌':'www.jellythink.com', '脸书':'www.facebook.com', '果冻想':'www.jellythink.com'}" />

<s:iterator value="#varMap" >

<s:property value="key"/>=<s:property value="value"/><br>

</s:iterator>

#### 集合运算符

1. in运算符：判断某个元素是否在集合中；
2. not in运算符：判断某个元素是否不在集合中；

示例：

<s:if test="100 in {100,200,300}">

包含

</s:if>

<s:else>

不包含

</s:else>

#### 访问静态成员

Struts2默认是关闭访问静态方法的，只允许通过OGNL表达式访问静态变量。为了让OGNL表达式可以访问静态方法，需要在Struts2配置文件中添加如下配置：

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

示例：

Action中：

public static String strWebSite = "http://www.jellythink.com";

public static String getStrWebSite()

{

return "果冻想 - " + strWebSite;

}

JSP中：

<s:property value="@com.jellythink.practise.LoginAction@strWebSite" />

<s:property value="@com.jellythink.practise.LoginAction@getStrWebSite()" />

## 标签库（未完成）

## 拦截器

### 概念

拦截器在概念上和servlet过滤器或JDKs代理类一样。拦截器允许横切功能在动作和框架中单独实现。你可以使用拦截器实现在动作被调用之前提供预处理逻辑、在动作被调用之后提供预处理逻辑、捕获异常，以便可以执行交替处理；

拦截器实现了面向切面编程（AOP，Aspect-Oriented Programming）的设计思想，拦截是AOP的一种实现策略（利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑的各个部分耦合度降低，提高程序可重用性，降低维护成本）；

Struts 2 框架提供的许多功能都是使用拦截实现的；例如包括异常处理，文件上传，生命周期回调和验证等。事实上，由于 Struts 2 是许多拦截器功能的基础，所以每次动作不是不可能有 7 个或 8 个拦截器被分配；

### 使用

#### 引用单个拦截器

为一个Action引用一个拦截器十分简单，需要使用时在**struts.xml**中配置。不需要使用则取消配置即可；

例：为该Action配置使用params和timer拦截器

<action name="hello" class="com.tutorialspoint.Struts.HelloWorldAction">

**<interceptor-ref name="params"/>**

**<interceptor-ref name="timer" />**

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

</action>

#### 引用拦截器栈

为每个动作配置多个拦截器很快就会变得非常难以管理。为此，可以使用拦截器栈进行管理；

例：

<interceptor-stack name="basicStack">

<interceptor-ref name="exception"/>

<interceptor-ref name="servlet-config"/>

<interceptor-ref name="prepare"/>

<interceptor-ref name="checkbox"/>

<interceptor-ref name="params"/>

<interceptor-ref name="conversionError"/>

</interceptor-stack>

//当配置初始的拦截器和拦截器栈时，必须确保在所有拦截器和拦截器栈配置中这个名称是唯一的

使用：应用拦截器栈与应用拦截器没有什么不同

<action name="hello" class="com.tutorialspoint.Struts 2.MyAction">

**<interceptor-ref name="basicStack"/>**

<result>view.jsp</result>

</action>

//上述注册的 “basicStack” 将完整地注册所有带有 hello 动作的 6 个拦截器。

**//应该指出的是，拦截器栈中各拦截器的定义顺序很重要，拦截器按照已配置的顺序执行。例如，在上述情况下，将首先执行异常，第二个执行的是 servlet 配置等等**

#### 拦截器的默认引用

在struts-default.xml中有以下一行代码：

<default-interceptor-ref name="defaultStack"/>

这是struts2的默认拦截器配置，表示对于任何Action，都会自动使用defaultStack这个拦截器栈，defaultStack是struts-default包指定的默认拦截器栈，这个包的继承者内所有的Action都将与这个拦截器栈中的所有拦截器关联：

struts2默认拦截器栈defaultStack：

<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="debugging"/>

<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-ref name="conversionError"/>

<interceptor-ref name="validation">

<param name="excludeMethods">input,back,cancel,browse</param>

</interceptor-ref>

<interceptor-ref name="workflow">

<param name="excludeMethods">input,back,cancel,browse</param>

</interceptor-ref>

</interceptor-stack>

注意：

如果在配置文件中为Action显式引用了任意拦截器后，默认拦截器栈将不再起作用，为了继续使用，必须显式引用默拦截器；

### 内置拦截器

Struts2框架内置了一系列拦截器，下面列出了一些重要的内置拦截器：

* alias：允许参数有不同的跨请求的别名；
* checkbox：通过为没有被检查的复选框添加一个参数值 false 来协助管理复选框。
* conversionError：把从字符串转化为参数类型的错误信息放置到动作的字段错误中。
* createSession：如果不存在 HTTP 会话，则自动创建一个 HTTP 会话。
* debugging：为开发人员提供几种不同的调试屏幕。
* execAndWait：当动作在后台执行的时侯，把用户定向到一个中间的等待页面。
* exception：映射动作抛出的异常到一个结果中，通过重定向允许自动异常处理。
* fileUpload：有利于简单的文件上传。
* i18n：在用户的会话期间，跟踪选定的语言环境。
* logger：通过输出被执行的动作的名称提供简单的日志。
* params：设置动作的请求参数。
* prepare：它通常是用来做预处理工作，如设置数据库连接。
* profile
* scope：在会话或应用程序的范围中存储和检索动作的状态。
* ServletConfig：为行动提供了各种基于 servlet 信息的访问。
* timer：以动作需要多长时间执行的形式提供了简单的配置信息。
* token：为有效的标记检查动作用来防止重复地表单提交。
* validation；为动作提供了验证支持。

### 自定义拦截器

1. 编写Interceptor类：

在你的应用程序中使用自定义的拦截器是一种提供横切的应用功能的简洁的方式。创建一个自定义的拦截器需要扩展Interceptor 接口：

public interface Interceptor extends Serializable{

void destroy();

void init();

String intercept(ActionInvocation invocation)

throws Exception;

}

init() 方法提供了一种初始化拦截器的方法，而destroy() 方法提供了一种清理拦截器的工具。与动作不同的是，拦截器在请求之间被重用，而且需要是线程安全的，尤其是 intercept() 方法，参数ActionInvocation 对象提供运行时环境的访问（它允许访问动作本身和方法来调用该动作，并判定动作是否已经被调用），返回字符串作为逻辑视图；

如果不需要初始化或清理代码，可以**直接扩展 AbstractInterceptor 类**，它提供了一个对 init() 和 destroy() 方法的默认的无操作实现，开发者直接重写intercept方法即可，这也是最常用的拦截器编写方法；

例：

package com.tutorialspoint.Struts 2;

import java.util.\*;

import com.opensymphony.xwork2.ActionInvocation;

import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

**public class MyInterceptor extends AbstractInterceptor** {

**public String intercept(ActionInvocation invocation)throws Exception**{

/\* let us do some **pre-processing** \*/

String output = "Pre-Processing";

System.out.println(output);

/\* let us call action or next interceptor \*/

**String result = invocation.invoke();//实际的动作将通过使用拦截器调用 invocation.invoke() 来执行**

/\* let us do some **post-processing** \*/

output = "Post-Processing";

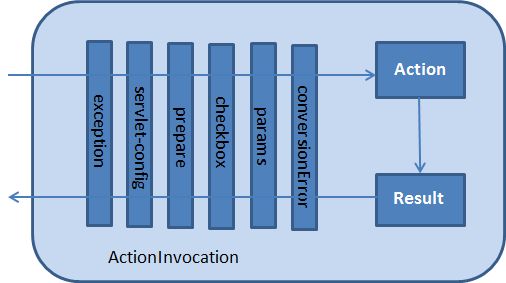
System.out.println(output);

return result;

}

}

//每次ActionInvocation 对象的 invoke() 被调用，ActionInvocation 查询它的状态，并且执行接下来的拦截器。当所有已配置的拦截器已经被配置时，invoke() 方法将引发这个动作本身被执行



1. 在struts.xml中注册自定义的拦截器并指定Action使用该拦截器

<struts>

<constant name="struts.devMode" value="true" />

<package name="helloworld" extends="struts-default">

**<interceptors>**

**<interceptor name="myinterceptor" class="com.tutorialspoint.Struts 2.MyInterceptor" />**

**</interceptors>**

<action name="hello"

class="com.tutorialspoint.Struts 2.HelloWorldAction">

<interceptor-ref name="params"/>

**<interceptor-ref name="myinterceptor" />**

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

</action>

//可以在interceptor标签内注册多个拦截器，并且同时你可以在action标签内调用多个拦截器，可以用不同的动作调用相同的拦截器

</package>

</struts>

## 校验器

### 使用校验方法

<http://www.blogjava.net/nokiaguy/archive/2008/04/17/193776.html>

#### 重写validate方法

ActionSupport类实现了一个Validateable接口。这个接口只有一个validate方法。如果Action类实现了这个接口，Struts2在调用execute方法之前首先会调用这个方法，我们可以在validate方法中编写验证代码，如果验证错误，可以根据错误的level选择字段级错误this.addFieldError("pwd", "错误信息")，还是动作级错误this.addActionError("错误信息")。并且可使用addFieldError或addActionError加入相应的错误信息，如果存在Action或Field错误，Struts2会返回“input”（这个并不用开发人员写，由Struts2自动返回），如果返回了“input”，Struts2就不会再调用execute方法了。如果不存在错误信息，Struts2在最后会调用execute方法。

工作原理：

当对Action发起请求时，Struts2 会自动执行该Action中的验证方法**validate()**，若遇到调用了addFieldError 方法（说明验证不成功），将不会继续调用 execute 方法，而 Struts 2 框架将返回input类型的result作为调用这个行动的结果；

为了处理返回值，我们需要在 struts.xml 添加下面的结果到我们的动作节点：

<result name="input">/index.jsp</result>

所以当验证失败和 Struts2 返回input时，Struts 2 框架将重新显示 index.jsp 文件。而如果使用了 Struts 2 的表单标签，Struts2 将会自动添加addFieldError 方法调用中指定的错误信息（直接显示在输入框上方）；

例：

public class Employee **extends ActionSupport**{//确保你的动作类扩展了 ActionSupport 类，否则你的 validate 方法将不会被执行

private String name;

private int age;

public String execute()

{

return SUCCESS;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

**public void validate()**

**{**

**if (name == null || name.trim().equals(""))**

**{**

**addFieldError("name","The name is required");**

**//** addFieldError 方法第一个参数是错误应用的表单字段名，第二个参数是显示上述表单字段的错误信息，如果前端表单是用<s:form/>编写的，Struts2会将错误信息直接显示在对应的表单字段输入框上方，如果前端表单是用普通<form>编写的，则将请求转发到Action定义的input结果页面，若没有提供name值为input的<result/>，那么发生错误时，将直接在浏览器中提示404错误

**}**

**if (age < 28 || age > 65)**

**{**

**addFieldError("age","Age must be in between 28 and 65");**

**}**

**}**

}

ps：addActionError 、addFieldError、addActionMessage方法辨析

1. addActionError是 Action级别的错误消息，使用：this.addActionError("错误信息"); 在前端显示： <s:actionerror />显示全部的Action级别的错误消息,可以加CSS；
2. addFieldError给一个字段(属性)添加错误消息，使用：this.addFieldError("pwd", "错误信息");在前端显示：
3. <s:fielderror />显示全部的错误消息；
4. <s:fielderror name=“”>，name属性为addFieldError方法中的参数fieldName；
5. 若什么都不写则默认显示在相应字段上方；
6. addActionMessage对参数进行校验，若校验失败则直接写错误信息，需要在action中重写父类的validate方法；使用：this.addActionMessage("信息"); 在前端显示：<s2:actionmessage/>可以显示全部的消息
7. 注意点

在添加第一个信息之前加上this.clearErrorsAndMessages();用来清空之前加载的错误信息；

若<head></head>之间加上<s:head/>，<s:actionerror />和<s:fielderror />会自动加上CSS，<s:actionmessage/>则不会加上；

#### 重写ValidateXxx方法

对于指定了method的Action，Struts2通过重写validateXxx方法专门校验Action种对应的xxx方法；

如果Action中存在validate方法和validateXxx方法，运行时先执行validateXxx方法，再执行validate方法，进行两次校验；

### 使用校验框架

<http://www.blogjava.net/nokiaguy/archive/2008/04/21/194599.html>

使用框架来进行数据校验的优点是允许验证从应用程序代码中分离，使用xml文件存储配置从而使用内置的校验器；

验证框架的包位置:

部署struts2项目有一个必备jar包：xwork-core-\*.jar，其中com.opensymphony.xwork2.validator.validators，所有官方验证类就放在这下面的，在这个包下面有一个"default.xml"文件，这就是验证框架的核心文件，里面配置了所有的验证类数据。

XML 文件会和类文件一起保存在 CLASSPATH 中，需要被命名为 '[action-class]'-validation.xml。所以，在我们的例子中，我们创建一个文件，名为 Employee-validation.xml，示例：

<!DOCTYPE validators PUBLIC

"-//OpenSymphony Group//XWork Validator 1.0.2//EN"

"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>

<!-- 校验Action的name属性 -->

<field name="name">

<!-- 指定name属性必须满足的必填规则 -->

<field-validator type="requiredstring">

<param name="trim">true</param>

<message>必须输入名字</message>

</field-validator>

<!-- 指定name属性必须匹配正则表达式 -->

<field-validator type="regex">

<param name="expression"><![CDATA[(\w{4,25})]]></param>

<message>您输入的用户名只能是字母和数字，且长度必须在4到25之间</message>

</field-validator>

</field>

<!-- 校验Action的password属性 -->

<field name="password">

<!-- 指定password属性必须满足必填的规则 -->

<field-validator type="requiredstring">

<param name="trim">true</param>

<message>必须输入密码</message>

</field-validator>

<!-- 指定password属性必须满足匹配指定的正则表达式 -->

<field-validator type="regex">

<param name="expression"><![CDATA[(\w{4,25})]]></param>

<message>您输入的密码指定是字母和数字，且长度必须在4到25之间</message>

</field-validator>

</field>

<!-- 校验Action的age属性 -->

<field name="age">

<field-validator type="int">

<param name="min">1</param>

<param name="max">150</param>

<message>年龄必须在1到150之间</message>

</field-validator>

</field>

<validators>

配置风格

Struts 2提供了两种方式来配置校验规则：

1. 字段校验器器配置

对于字段校验器配置，这是一种以字段优先的配置方式

使用字段校验器配置风格时，每个字段校验规则大致遵守如下形式：

<field name="被校验的字段">

<field-validator type="校验器名">

<!-- 此处需要为不同校验器指定数量不等的校验参数-->

<param name="参数名">参数值</param>

......

<!-- 校验失败后的提示信息，其中key指定国际化信息的key-->

<message key="I18Nkey">校验失败后的提示信息</message>

</field-validator>

<!-- 如果该字段需要满足多个规则，下面可以配置多个校验器-->

</field>

1. 非字段校验器配置

对于非字段校验器配置风格，是一种以校验器优先的配置方式。

对于采用非字段校验器风格的校验规则文件，<validators.../>元素下有多个<validator.../>元素，每个<validator.../>元素都有如下格式：

<validator type="校验器名">

<param name="fieldname">需要被校验的字段</param>

<!-- 此处需要为不同校验器指定数量不等的校验参数-->

<param name="参数名">参数值</param>

.....

<!-- 校验失败后的提示信息，其中key指定国际化信息的key-->

<message key="I18Nkey">校验失败后的提示信息</message>

</validator>

内建校验器

Struts 2提供了大量的内建校验器，这些内建校验器可以满足大部分应用的校验需求。

在xwork-core-2.2.1.jar文件中的com\opensymphony\xwork2\validator\validators路径下的default.xml文件，该文件就是Struts 2默认的校验器注册文件。

<validators>

<validator name="required" class="com.opensymphony.xwork2.validator.validators.RequiredFieldValidator"/>

<validator name="requiredstring" class="com.opensymphony.xwork2.validator.validators.RequiredStringValidator"/>

<validator name="int" class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"/>

<validator name="long" class="com.opensymphony.xwork2.validator.validators.LongRangeFieldValidator"/>

<validator name="short" class="com.opensymphony.xwork2.validator.validators.ShortRangeFieldValidator"/>

<validator name="double" class="com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"/>

<validator name="date" class="com.opensymphony.xwork2.validator.validators.DateRangeFieldValidator"/>

<validator name="expression" class="com.opensymphony.xwork2.validator.validators.ExpressionValidator"/>

<validator name="fieldexpression" class="com.opensymphony.xwork2.validator.validators.FieldExpressionValidator"/>

<validator name="email" class="com.opensymphony.xwork2.validator.validators.EmailValidator"/>

<validator name="url" class="com.opensymphony.xwork2.validator.validators.URLValidator"/>

<validator name="visitor" class="com.opensymphony.xwork2.validator.validators.VisitorFieldValidator"/>

<validator name="conversion" class="com.opensymphony.xwork2.validator.validators.ConversionErrorFieldValidator"/>

<validator name="stringlength" class="com.opensymphony.xwork2.validator.validators.StringLengthFieldValidator"/>

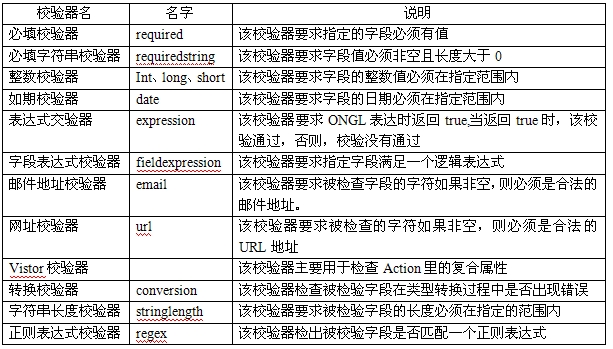
<validator name="regex" class="com.opensymphony.xwork2.validator.validators.RegexFieldValidator"/>

<validator name="conditionalvisitor" class="com.opensymphony.xwork2.validator.validators.ConditionalVisitorFieldValidator"/>

</validators>

通过上面可以发现，注册一个校验器只需要通过一个<validator.../>元素即可注册一个校验器，每一个<validator.../>元素的name属性指定该校验器的名字，class属性指定该校验器的实现类。

常用的内建校验器：



## 国际化（未完成）