常见的设计模式：工厂，单例，代理，观察者

## 1.Strut2含义及发展历史

英文意思是：支撑杆，支持物。

发展历史：

Model 1 : JSP+JavaBean模式

Model 2: JSP+ Servlet+JavaBean 模式

Strut2之前还有一个Struts1框架，但是Struts2是继承自Webwork框架，吸收了两种的优势。

## 2.Struts2框架结构





2.在上述问题进行升级，在多URL的情况下怎么办。

步骤分析：

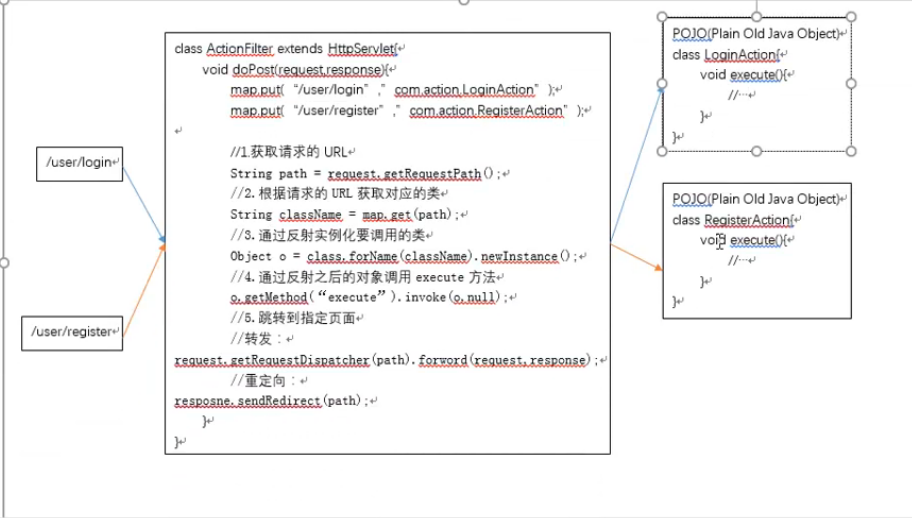
1.首先在map结构中存储url以及对应的类的全名

2.根据请求的URL获取对应的类。

3.通过反射实例化调用类。

4.通过反射之后的类调用excute方法

5.跳转到指定的页面（转发，重定向）



3.在上述问题进行升级，在多URL的情况下怎么办。

步骤分析：

1.首先在map结构中存储url以及对应的类的全名

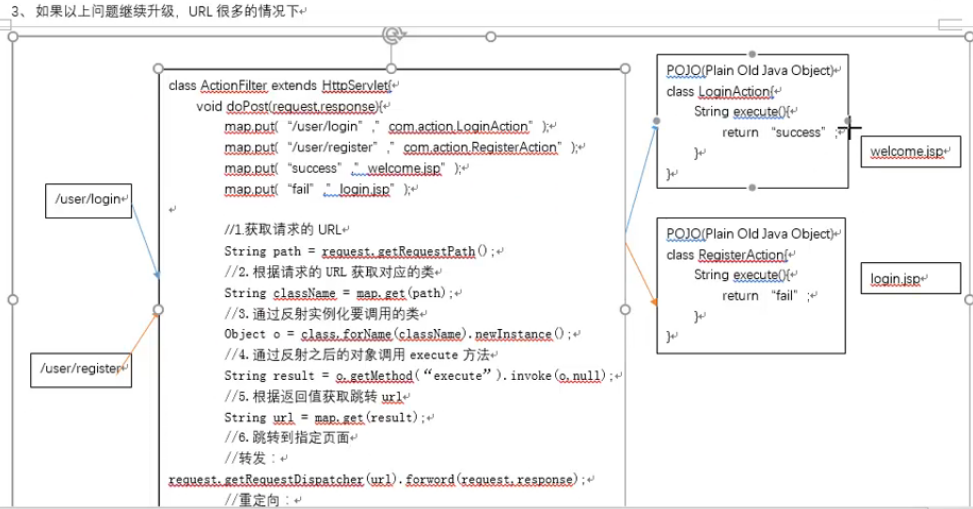
2.根据请求的URL获取对应的类。

3.通过反射实例化调用类。

4.通过反射之后的类调用excute方法

5.根据返回值回去URL

6.跳转到指定的页面（转发，重定向）



缺点：url映射，硬编码。 可以通过配置文件实现：properties, xml

<action name=”login” class=”com.action.LoginAction”>

<result name=”success”>/welcome.jsp</result>

<result name=”fail”>/login.jsp</result>

</action>

<action name=”register” class=”com.action.RegisterAction”>

<result name=”success”>/welcome.jsp</result>

<result name=”fail”>/register.jsp</result>

</action>

**配置文件**

Struts框架引入文件配置：

Struts.xml是核心配置文件放在web/classes目录下

Struts2框架是是基于拦截器的功能实现的

1. 创建web项目
2. 添加Struts包的支持
3. 创建各种Action

第一步创建POJO类

|  |
| --- |
| package com.demo.action;  public class LoginAction {  public String execute(){  return "success";  }  } |

1. 配置struts.xml文件

|  |
| --- |
| <struts>  <constant name="struts.i18n.reload" value="false"/>  <constant name="struts.devMode" value="false"/>  <!-- <include file="struts-default.xml"/>-->  <!--  action元素详解  name：action的访问资源名称。  class：action的完整类名。  method：指定调用action类中那个方法处理请求。  -->  <package name="default" extends="struts-default" namespace="/mynamespace">  <action name="login" class="com.demo.action.DemoAction" >  <result name="success">/index.jsp</result>  <result name="error">/login.jsp</result>  </action>  </package>  </struts> |

1. 启动运行看结果

**默认的配置文件：**

1.struts-default.xml Struts默认配置文件 在struts-cored-2.2.1.jar包中

2.defult.properties 默认常量配置 org.apache.struts2

|  |  |
| --- | --- |
| struts.i18n.encoding=UTF-8 | 编码格式 |
| # struts.objectFactory = spring | SSH集成时使用 |
| struts.objectFactory.spring.autoWire = name | 自动装配 |
| struts.multipart.parser=jakarta  struts.multipart.saveDir=  struts.multipart.maxSize=2097152 | 文件上传相关配置  上传临时目录  文件最大值 |
| struts.action.extension=action,, | URL 后缀配置 可自定义  支持xxx.action和没有后缀 |
| struts.enable.DynamicMethodInvocation = false | 动态调用方法 |

3.web.xml 项目配置文件

4.struts.xml Struts核心配置文件

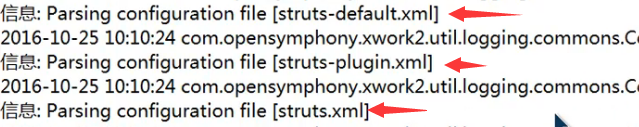
5.struts.properties 自定义常量配置文件

6.struts-plugin.xml 插件自定义

**几大配置文件优先级**

Web.xml > struts.properties > struts.xml > default.properties

**配置加载顺序**



核心配置文件

1. 常量配置

|  |
| --- |
| <constant name="struts.devMode" value="false"/> |

对default.properties中进行个性配置

1. Package元素

Action元素配置包，Java代码中的“包”，用来管理Java代码，防止命名冲突，package节点作用也是一样，防止action名冲突，在项目总指定唯一的name属性

|  |  |
| --- | --- |
| Name | 唯一标识 |
| Namespace | 命名空间，影响该包下属的action访问路径 |
| Extends | 指定继承xxx配置文件，默认继承自struts-default.xml文件 |
| Abstract | 定义为抽象包，只能别的包去继承此包，通常用于项目中的默认配置 |

1. Action元素

用于配置处理请求

结合 namespace+action<name属性>

|  |  |
| --- | --- |
| Name | Action名字，影响URL请求路径 |
| Class | 指定action全限定名 |
| method | 指定调用类方法 默认execute |
|  |  |

1. Result元素

配置action响应，可以配置多个

|  |  |
| --- | --- |
| Name | 类中方法返回值，要一致 |
| Type | 请求响应类型，默认是（转发dispacher）（可以改成重定向redirect） |

1. 协作配置文件

对于一个项目而言，通常配置 we年

|  |  |
| --- | --- |
| <include file=”zhangsan.xml”></include> | 文件路径 |

dtd文件对XML文件的约束

1. default-action-ref元素

默认action配置，通常用于防止URL输入错误，显示404错误页面，但该页面往往对用户不友好，可以通过使用该配置控制404显示页面

|  |  |
| --- | --- |
| <default-action-ref name=”default”></default-action-ref> | 当乱输URL时找不到action时,调用默认的action |

1. default-class-ref元素

Action找不到对应的Class的时候，会调用默认的Class类

|  |  |
| --- | --- |
| <default-class-ref class=”com.action.LoginAction”></default -class-ref> |  |

1. global-results元素

多个action共享的result

|  |
| --- |
| <global-results>  <result name=”error”>/500.jsp</result>  </global-results> |

**动态调用方法**

不使用action的method属性的情况下，通过URL <http://localhost:6060/struts2002/test/login!hello>

指定调用后台的xx方法 使用“action名！方法名”使用时需“struts.enable.DynamicMethodInvocation = true”设置为true，注意该配置在2.1以上的版本中默认为false .

**使用通配符**

在action的name属性中可以使用\*去命名，那么在其他属性中需要使用“{1}”或者“{2}”或者“{0}”取出URL路径。

范例：

|  |  |
| --- | --- |
| Name=”\*\_\*” | {1}取出第一个\*号  {2}取出第二个\*号 |
| url=/user\_add | {1}=user 通常出现在class属性中  {2}=add 通常出现在method属性中 |

{0}所取出来的数据就是类似如下效果：user\_add

**接受请求参数**

Struts2是MVC框架中的控制器（处理请求）那么在整个请求过程中，对参数处理就是Struts2的请求所在，接受和处理参数分为属性驱动，模型驱动，域驱动等

1. 属性驱动

直接在action的POJO类中声明与表单元素一致的属性，并生成匹配的get,set方法strut2会自动将表单数据提交到action中，并且直接给相应的属性赋值

1. 模型驱动

Struts2中提供了一个模型驱动的拦截器，要使用此模型驱动的拦截器填充参数，需要实现ModelDriven<XXX>泛型类

|  |
| --- |
| @override  Public User getModel(){  If(user==null)  User=new User();  Return user;  } |

（3）域驱动

通常情况下，表单元素往往对应一个实体，实体对象往往放在实体层中，那么需要在action中放实体、

Html代码

|  |
| --- |
|  |

Action代码LoginAction.java

|  |
| --- |
| xx |

（3）集合驱动

List<User> user

**获取ServletAPI**

ServletAPI提供什么东西？

Web容器提供各种内置对象，request,session,application等，在struts2中获取这些对象可以通过三种方式，ActionContext(Struts建议的)，ServletActionContext(可以获取真正的HttpServletRequest等)，特定接口（了解即可）

1. ActionContext

|  |
| --- |
| package com.demo.action;  import com.opensymphony.xwork2.ActionContext;  public class LoginAction {  public String execute(){  ActionContext act =ActionContext.getContext();  act.getSession().put("request","1111111111111");  act.getSession().put("session","22222222222222");  act.getSession().put("application","333333333333");  System.out.println("execute login.jsp文件");  return "success";  }  } |

通过以上代码，我们能给这些对象中放入数据，但是通过ActionContext操作的 对象并不是真正的HttpServletRequest，HttpServletResponse对象，只是ActionContext提供的一个map而已？是为了降低对ServletAPI耦合度

1. 通过ServletActionContext获取

|  |
| --- |
| package com.demo.action;  import com.opensymphony.xwork2.ActionContext;  import org.apache.struts2.ServletActionContext;  import javax.servlet.http.HttpServletRequest;  public class LoginAction {  public String demo(){  HttpServletRequest httpServletRequest= ServletActionContext.getRequest();  httpServletRequest.setAttribute("request","reqeust111");  httpServletRequest.getSession().setAttribute("session","session1111");  ServletActionContext.getServletContext().setAttribute("application","application1111");  return "success";  }  } |

1. 通过实现特定接口获取

实现接口

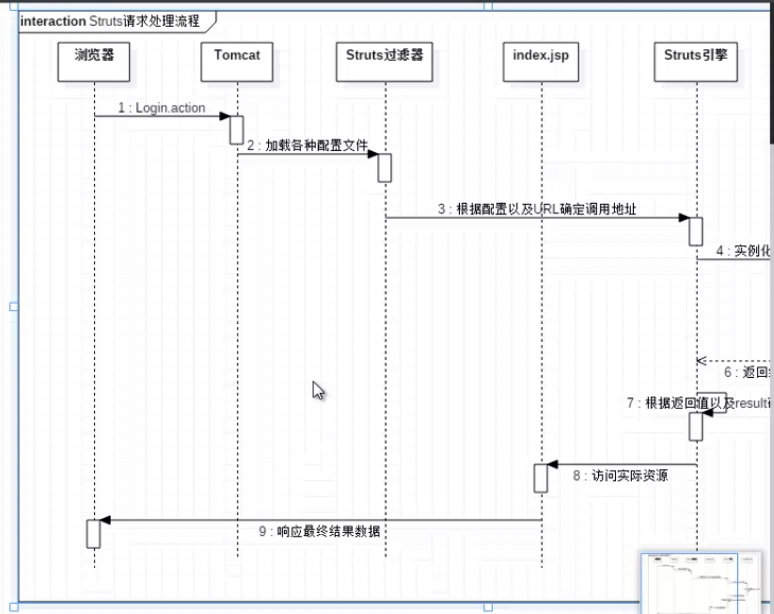
|  |
| --- |
| import org.apache.struts2.interceptor.ApplicationAware;  import org.apache.struts2.interceptor.RequestAware;  import org.apache.struts2.interceptor.SessionAware;  import java.util.Map;  public class TestAction implements RequestAware, SessionAware, ApplicationAware {  private Map<String, Object> request;  private Map<String, Object> session;  private Map<String, Object> appliction;  public String execute(){  return "success";  }  public void setApplication(Map<String, Object> application) {  application.put("application","aaaaaaaaa");  this.appliction=application;  }  public void setRequest(Map<String, Object> request) {  request.put("request","rrrrr");  this.request=request;  }  public void setSession(Map<String, Object> session) {  session.put("session","ssssss");  this.session=session;  }  } |

显示

|  |
| --- |
| ${sessionScope.user},您已经登录了<br/>  ${requestScope.tip}<br/>  从系统读取的cookie值是：${cookie.user.value},<br/>  request: ${requestScope.request},<br/>  session: ${sessionScope.session},<br/>  application: ${applicationScope.application},<br/> |

侵入式编程，没什么人用

**Struts2请求处理过程**



每次请求Struts会自动生成ActionContext,这个对象中就包含了请求所需的各种参数，并且自动化实例化Action类，调用指定方法或者默认方法、。

在ActionContext中主要包括了Map集合，主要包括6大对象：

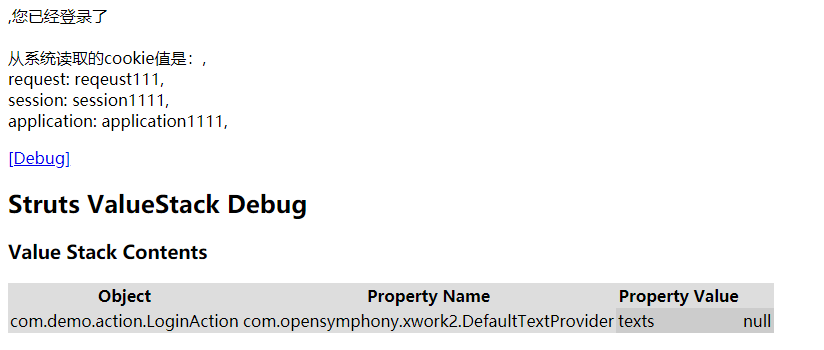
Request,session,application,attr, parameter,valuestack

**值栈**

值栈是Struts2框架提供的一个存储区域，是一个栈结构（FILO）往往存储Action实例对象。

观测值栈中内容：

|  |
| --- |
| <%@ taglib uri="/struts-tags" prefix="s"%>  <s:debug/> |



获取值栈内容，并且往里面加入内容（对值栈进行操作）。

|  |
| --- |
| System.out.println(user.getUsername()+" : "+user.getPassword());  //获取值栈  ValueStack stack=ActionContext.getContext().getValueStack();  //往值栈中放入数据  vs.push(new User("DEMO1","DEMO2"));  return "success"; |

通过以上演示，值栈是一个在请求的同时就自动创建的一个对象，往往不需要手动操作的，strut2框架会自动操作值栈（放入数据），取数据根据义务逻辑自行获取，往往与

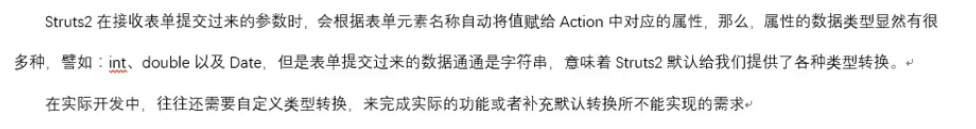
**OGNL配套使用的。**

对象图导航语言（Object graphic navigation language）,也类似于JSTL，用来访问值栈中的数据。

JSTL和EL表达式

**类型转换器**

Struts2接收输入参数的时候会自动进行类型转换。

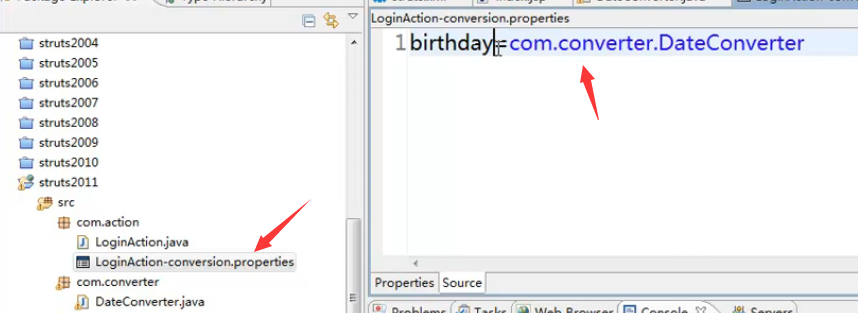


定义一个类继承DefaultTypeConverter

重写public Object convertValue(Object value,Class toType)方法

定义好转换器以后，需要在Action同目录下创建与该Action对应的配置文件。

Action类名-conversion.properties

范例

**拦截器**

拦截器，顾名思义，拦截请求（action）,可以达到增强Action的功能吗，框架本身默认提供了很多大概35中拦截器，但是实际开发过程中还是需要自定义拦截器，譬如：权限（访问）拦截

如果要实现拦截器，需要实现Interceptor接口，或者继承abstractInterceptor抽象类，后者也是实现了前者的类

开始准备：几个jsp页面

|  |
| --- |
| package com.action;  import com.opensymphony.xwork2.ActionContext;  public class UserAction {  public String add(){  //判断是否登录了  String user= (String) ActionContext.getContext().getSession().get("user");  if(user==null||"".equals(user)){  System.out.println("未登录！！！！！，跳转到登录页面");  return "fail";  }  return "success";  /\*if(user==null){  return "success";  }\*/  }  } |

定义拦截器

|  |
| --- |
| package com.interceptor;  import com.opensymphony.xwork2.ActionContext;  import com.opensymphony.xwork2.ActionInvocation;  import com.opensymphony.xwork2.interceptor.AbstractInterceptor;  public class InterceptorDemo1 extends AbstractInterceptor {  @Override  public String intercept(ActionInvocation actionInvocation) throws Exception {  //判断是否登录了  String user= (String) ActionContext.getContext().getSession().get("user");  if(user==null||"".equals(user)){  System.out.println("未登录！！！！！，跳转到登录页面");  return "fail";  }  return actionInvocation.invoke();  }  } |

到struts.xml中配置

|  |
| --- |
| <package name="user" namespace="/" extends="struts-default">  <!--配置拦截器名-->  <interceptors>  <interceptor name="Interceptor1" class="com.interceptor.InterceptorDemo1"></interceptor>  </interceptors>  <action name="login" class="com.action.LoginAction" method="login">  <result name="success">/login.jsp</result>  </action>  <action name="logout" class="com.action.LoginAction" method="logout">  <result name="success">/logout.jsp</result>  </action>  <action name="useradd" class="com.action.UserAction" method="add" >  <result name="success">/adduser.jsp</result>  <result name="fail">/login.jsp</result>  <!--如果项目中使用了自定义的拦截器，那么默认拦截器就不在使用了-->  <interceptor-ref name="Interceptor1"></interceptor-ref>  <interceptor-ref name="defaultStack"></interceptor-ref>  </action>  </package> |

通过action拦截，只能拦截部分请求，如果要大量添加就会出现大量重复代码。最好方法，将制定action或者部分action进行一起拦截

**全局拦截**

|  |
| --- |
| <package name="user" namespace="/" extends="struts-default">  <!--配置拦截器名-->  <interceptors>  <interceptor name="Interceptor1" class="com.interceptor.InterceptorDemo1"></interceptor>  <!--将自己的拦截器，加入拦截器中-->  <interceptor-stack name="myStack">  <interceptor-ref name="Interceptor1"></interceptor-ref>  <interceptor-ref name="defaultStack"></interceptor-ref>  </interceptor-stack>  </interceptors>  <default-interceptor-ref name="myStack"></default-interceptor-ref>  <action name="login" class="com.action.LoginAction" method="login">  <result name="success">/welcome.jsp</result>  </action>  <action name="logout" class="com.action.LoginAction" method="logout">  <result name="success">/logout.jsp</result>  </action>  <action name="useradd" class="com.action.UserAction" method="add" >  <result name="success">/adduser.jsp</result>  <result name="fail">/login.jsp</result>  <!--如果项目中使用了自定义的拦截器，那么默认拦截器就不在使用了-->  <interceptor-ref name="Interceptor1"></interceptor-ref>  <interceptor-ref name="defaultStack"></interceptor-ref>  </action>  </package> |

问题1：返回值问题，因为全局拦截将包中所有Action都拦截，但是拦截器的返回值只能固定一个，此时，所有被拦截的Action中都需要配置与拦截器返回值相同的result，可以将此返回值配置为全局结果集。

问题2：包含的某些个别的Action又不希望被拦截，譬如“登录”，“登出”，“注册”等两种结局方案。

1. 将这些功能直接放到jsp页面中 注意：不要暴露实际操作jsp页面地址
2. 将这些不需要拦截的Action单独放到另外一个包中，或者使用方法拦截

|  |
| --- |
| <struts>  <constant name="struts.i18n.reload" value="false"/>  <constant name="struts.devMode" value="false"/>  <package name="user" namespace="/" extends="struts-default">  <!--配置拦截器名-->  <interceptors>  <interceptor name="Interceptor1" class="com.interceptor.InterceptorDemo1"></interceptor>  <!--将自己的拦截器，加入拦截器中-->  <interceptor-stack name="myStack">  <interceptor-ref name="Interceptor1"></interceptor-ref>  <interceptor-ref name="defaultStack"></interceptor-ref>  </interceptor-stack>  </interceptors>  <default-interceptor-ref name="myStack"></default-interceptor-ref>  <global-results>  <result name="fail" type="redirect">/login.jsp</result>  <result name="success">/welcome.jsp</result>  </global-results>  <action name="useradd" class="com.action.UserAction" method="add" >  <result name="success">/adduser.jsp</result>  <!--如果项目中使用了自定义的拦截器，那么默认拦截器就不在使用了-->  <interceptor-ref name="Interceptor1"></interceptor-ref>  <interceptor-ref name="defaultStack"></interceptor-ref>  </action>  </package>  <package name="login" namespace="/" extends="struts-default">  <action name="login" class="com.action.LoginAction" method="login">  <result name="success">/welcome.jsp</result>  </action>  <action name="logout" class="com.action.LoginAction" method="logout">  <result name="success">/logout.jsp</result>  </action>  </package>  </struts> |

**方法拦截**

如果在一个Action中，如果有多个方法，那么拦截器就会拦截所有的方法。

范例继承拦截器：继承MethodFilterInterceptor

|  |
| --- |
| package com.interceptor;  import com.opensymphony.xwork2.ActionInvocation;  import com.opensymphony.xwork2.interceptor.MethodFilterInterceptor;  public class InterceptorDemo3 extends MethodFilterInterceptor {  @Override  protected String doIntercept(ActionInvocation actionInvocation) throws Exception {  System.out.println("执行方法拦截");  return actionInvocation.invoke();  }  } |

范例：配置struts.xml文件

|  |
| --- |
| <package name="user" namespace="/" extends="struts-default">  <interceptors>  <interceptor name="interceptor3" class="com.interceptor.InterceptorDemo3"/>  </interceptors>  <action name="edituser" class="com.action.UserAction" method="edit">  <result name="success">/edituser.jsp</result>  <interceptor-ref name="interceptor3"/>  <interceptor-ref name="defaultStack"/>  </action>  <action name="adduser" class="com.action.UserAction" method="add">  <result name="success">/adduser.jsp</result>  <interceptor-ref name="interceptor3">  <param name="excludeMethods">add</param>  </interceptor-ref>  <interceptor-ref name="defaultStack"/>  </action>  </package> |

|  |  |
| --- | --- |
| <param name="excludeMethods">add</param> | 不拦截 |
| <param name="includeMethods">add</param> | 拦截 |

**表单数据验验证**

表单验证分为三步：客户端验证，服务端验证，数据库验证

1. 手工验证：

范例：

|  |
| --- |
| Jsp代码 |

范例：Action代码 可以继承ActionSupport，会自动调用validate 等等



这里regex表达式：^开始$结束，[]单选{9}个数 \转义

以上代码，表单验证方法，会在execute方法执行之前被调用。

问题: Action方法中可能存在多个方法，有的方法我们需要验证，有的方法我们不需要验证

验证Action的指定方法，只需要将重写的vaildate方法名 改成validate+方法名就好了。方法名首字母需要大写。

**基于XML的验证**

如果要验证Action是LoginAction，那么配置文件名需命名为LoginAction-validation.xml

会对所有方法进行验证，要针对指定方法进行验证，只需要将配置文件名改成指定格式“ActionClassName-MethodName-validation.xml”

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE validators PUBLIC  "-//Apache Struts//XWork Validator 1.0.3//EN"  "http://struts.apache.org/dtds/xwork-validator-1.0.3.dtd">  <validators>  <field name="username">  <field-validator type="requiredstring">  <message>用户名不能为空</message>  </field-validator>  </field>  <field name="password">  <field-validator type="requiredstring">  <message>密码能为空</message>  </field-validator>  </field>  </validators> |

**防止表单重复提交**

由于用户网速问题，用户可能会不断点击提交按钮，或者刷新提交页面，已经提交成功，但是用户选择后退后恶意刷新页面，形成重复提交的问题。

**令牌机制**

在struts2中使用了令牌机制解决表单重复提交问题。产生一个随机字符串，在服务器端和客户端同时保留此字符串的副本，在客户端向服务器端发送请求过程中，会自动发送此副本，服务端接受到请求之后，判断本地令牌与客户端提交上来的令牌是否相符，如果相同就执行提交动作，完成数据库等业务逻辑操作，完成之后服务器端的副本会立马产生一个新的令牌，而客户端的字符串还是原字符串，那么，此时如果客户端再要提交，服务端与客户端的令牌就已经不一样了，则判断为重复提交！！！！！。

**使用struts2防止表单重复提交。**

范例：1.在jsp表单中加入 <s:token/>

|  |
| --- |
| <%@ taglib prefix="s" uri="/struts-tags" %>  <%--  Created by IntelliJ IDEA.  User: Sidewinder  Date: 2019/2/21  Time: 15:48  To change this template use File | Settings | File Templates.  --%>  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <html>  <head>  <title>Title</title>  </head>  <body>  <form action="login.action" method="post">  <s:token></s:token>  <table>  <tr>  <td>用户名：</td>  <td><input type="text" name="username" /> </td>  </tr>  <tr>  <td>密码：</td>  <td><input type="text" name="password" /> </td>  </tr>  <tr>  <td colspan="2">  <input type="submit" value="登录" />  <input type="reset" value="重置" /></td>  </tr>  </table>  </form>  </body>  </html> |

将token拦截器，加入到默认的拦截器中。

|  |
| --- |
| <package name="user" namespace="/" extends="struts-default">  <interceptors>  <interceptor-stack name="myStack">  <interceptor-ref name="token"/>  <interceptor-ref name="defaultStack"/>  </interceptor-stack>  </interceptors>  <default-interceptor-ref name="myStack"/>  <action name="login" class="com.action.LoginAction" method="login">  <result name="success">/welcome.jsp</result>  </action>  </package> |