# 简介

Structs2是一个基于MVC模式开发框架,对servlet技术的封装

MVC模式:

M:Model模型.使用javabean技术,对业务数据的一种封装

V:View 试图.使用jsp技术,对业务数据的显示

C:Control控制器.使用servlet技术

作用:

1. 接受请求,获取用户的参数
2. 调用业务逻辑处理方法
3. 获取结果,导航试图

需求:用户登陆和注册功能(用户模块),使用MVC模式进行开发

设计:

dao层:

queryUser(User user);//查询登陆用户

addUser(User user);//添加用户

service层:

UserService类:

login(User user);//登陆业务

register(User user);//注册业务

web 层:

一个功能对应一个Servlet类

登陆功能:LoginServlet类:/login访问

注册功能:RegisterServlet类:/register访问

**问题:**

一个功能对应一个Servlet处理类,这样的话项目管理的servlet就会很多,web.xml配置很麻烦,更多的servlet对象会消耗更多的内存

# 自定义struts

项目:Demo1

# Struts2的使用

struts2的版本是2.3.34

## 3.1使用步骤

1)导入struts2的jar包

迷你的jar包有9个:

commons-beanutils-1.8.0 :beanutils工具包

commons-fileupload-1.3.2:文件上传

commons-io-2.2

commons-lang3-3.2:struts2对java.lang.\*类的支持

freemarker-2.3.22:视图技术

javassist-3.11.0.GA:struts对字节码运算的支持

ognl-3.0.21:struts2对ognl表达式的一个支持

struts2-core-2.3.34:struts2的核心包

xwork-core-2.3.34:struts2的前身是webwork框架,xwork-core是webwork的核心包

2)配置启动的全局的过滤器(Filter) 和ActionServlet类似

|  |
| --- |
| <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>  <!--过滤所有请求-->  <url-pattern>/\*</url-pattern>  </filter-mapping> |

3)编写Action类,这个类要实现Action接口

|  |
| --- |
| package lw.pers.action;  import com.opensymphony.xwork2.Action;  /\*\*  \*登陆逻辑action  \*/  public class LoginAction implements Action {  @Override  public String execute() throws Exception {  System.out.println("执行了LoginAction的execute方法");  return "success";  }  } |

4)在src目录下建立struts.xml文件，配置Action对象

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="xxx" extends="struts-default" namespace="/">  <!--配置Action-->  <action name="login" class="lw.pers.action.LoginAction">  <result name="success" type="redirect">/index.jsp</result>  </action>  </package>  </struts> |

5)访问:localhost:8080/test/login.action

## 3.2struts2框架的作用

struts2是基于MVC开发模型的web层框架,是基于webwork的升级版,struts2=webwork+struts1

struts1也是MVC开发模式的框架,struts2并不是struts1的升级版,而是webwork

**struts2的具体功能:**

1. 请求数据封装
2. 简化国际化
3. 文件上传
4. 后台数据校验

......

**项目的三层结构**

DB

dao层

(数据访问层)

service层

(业务逻辑层)

web层

(表现层)

servlet+jsp技术 普通java类 jdbc/dbutil

struts2框架代替web层的

Spring 框架代替Service层

Hibernate框架代替dao层

## 3.3Struts2的执行过程

项目启动:

1)创建核心过滤器StrutsPrepareAndExecuteFilter对象

2)执行核心过滤器的init方法,在执行init之前读依次读取了

struts-default.xml [struts2框架的默认配置文件]

struts-plguin.xml [struts2的插件配置文件]

struts.xml [我们写的业务的配置文件]

访问资源:

3)在内存中查询对应的Action配置,得到class内容创建Action对象

4)执行Action配置的method内容,执行Action对象对应的方法

## 3.4struts-default.xml文件详解

**1)声明struts2运行过程中使用到的一些对象**

|  |
| --- |
| <bean class="com.opensymphony.xwork2.ObjectFactory" name="xwork" />  <bean type="com.opensymphony.xwork2.ObjectFactory" name="struts" class="org.apache.struts2.impl.StrutsObjectFactory" /> |

**2)默认包,包名叫作struts-default(注意我们自己写的package必须继承这个默认包,继承了这**个包才能使用默认包下的功能)

<package name=”struts-default” abstract=”true”>

a.跳转类型:

redirect:重定向到页面

dispatcher:转发到页面

redirecAction:重定向到Action

chain:转发到Action

b.拦截器(Interceptor)

struts2默认的拦截器(32个):完成struts2的核心功能(请求封装,国际化...)

**3)拦截器和过滤器**

拦截器(Interceptor) vs 过滤器(Filter)

过滤器:可以过滤任何类型的请求(html,Servlet,jsp)和响应,加入通用的逻辑代码

拦截器:是struts2的特有功能,它只能过滤Action,在执行Action的之后加入通用的代码

声明一个拦截器:

<interceptor name="alias" class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>

**4)拦截器栈**

<interceptor-stack name=”defaultstack”> (默认拦截器栈)

<interceptor-ref name="exception"/>

<interceptor-ref name="alias"/>

<interceptor-ref name="servletConfig"/>

......

</interceptor-stack>

**struts-default包使用的拦截器栈为:**

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

**5)使用的Action**

struts-default包默认使用的Action是

<default-class-rel class=”com.opensymphony.xwork2.ActionSupport”/>

## 3.5struts.xml

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="xxx" extends="struts-default" >  <!--配置Action-->  <action name="login" class="lw.pers.action.LoginAction">  <result name="success" type="redirect">/index.jsp</result>  </action>  </package>  </struts> |

1. package:表示一个包,管理action的配置,一个包下面不能含有同名的action

package标签:

name:包名,在一个项目中不能出现同名的包

extend:继承,类似于类的继承,如果一个包继承另一个包,那么就会把父包的功能继承下来,一般要继承struts-default包

namespace:名称空间,区分不用包的访问路径,默认值就是”/”

用户访问Axtion的路径搜索规则:

例子:如果namespace=”user”

用户访问:http://localhost:8080/test/user/a/b/login.action

先搜索:http://localhost:8080/test/user/a/b/login.action 没有

接着搜索:http://localhost:8080/test/user/a/login.action 没有

再搜索:http://localhost:8080/test/user/login.action 有

abstract:表示当前包是否抽象,如果当前包是抽象包,则不能含有action,抽象包一般用来定义拦截器,公共视图,不做具体的业务

2)action标签

name:action的名称,用户访问Action的路径

class:Action业务对象的类名,一定是全名,默认值:ActionSupport类,该类继承自struts-default的配置

method:指定的业务方法,默认值是execute

3)result标签

name:视图的标记,在一个Action中不要重复,默认值是success

type:跳转的类型

redirect:重定向到页面

dispatcher:转发到页面(默认值)

redirecAction:重定向到Action

chain:转发到Action

stream:用在下载文件的时候

文本内容:跳转到的页面或者Action

4) global-results标签

**全局视图**

如果有局部,则局部覆盖全局

|  |
| --- |
| <struts>  <package name="xxx" extends="struts-default" >  <!--全局视图,把该包下的所有action共用的视图集中写在这里-->  <global-results>  <result name="success">/index.jsp</result>  </global-results>  <!--配置Action-->  <action name="login" class="lw.pers.action.UserAction3" method="login">  <!--<result name="success" type="redirect">/index.jsp</result>--> 局部  </action>  <action name="book" class="lw.pers.action.BookAction" method="add">  </action>  </package>  </struts> |

## 3.6配置文件分开

struts.xml中:

|  |
| --- |
| <struts>  //这个include的顺序,就是package声明的顺序  <include file=”config/struts-book.xml”></include>  <include file=”config/struts-user.xml”></include>  </struts> |

# 4.struts2的Action三种使用方式

## 4.1方式1.不实现Action接口

|  |
| --- |
| public class UserAction2 {  public String login() throws Exception{  System.out.println("UserAction2.login()");  return "success";  }  } |

## 4.2方式2.实现Action接口

|  |
| --- |
| import com.opensymphony.xwork2.Action;  /\*\*  \*登陆逻辑action  \*/  public class LoginAction implements Action {  @Override  public String execute() throws Exception {  System.out.println("执行了LoginAction的execute方法");  return Action.SUCCESS;  }  } |

## 4.3继承了一个类（推荐）

1)提供了常用的视图标记

2)提供了数据校验功能

|  |
| --- |
| import com.opensymphony.xwork2.ActionSupport;  public class UserAction3 extends ActionSupport{  @Override  public String execute() throws Exception {  return super.execute();  }  } |

# 5.Action路径通配符

\* 和{1}相对应,\*是什么{1}就是什么,可以推断,第二个\*对应的是{2}

这样写,可以减少action的配置.

**1)一个模块,一个action**

<action name="user\_\*" class="lw.pers.action.UserAction3" method="{1}">

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

</action>

**2)多个模块使用一个action配置**

<!--第一个\*表示模块名称,第二个\*代表方法-->

<action name="\*\_\*" class="lw.pers.action.{1}" method="{2}">

<result name="{2}">/{1}/{2}.jsp</result>

</action>

# 6.struts.xml配置常量

这个常量的配置可以放进一个单独的xml文件中

首先是读取default.properties 在依次读取三个xml文件,这个default.properties文件

里保存着常量的信息,可以到struts.xml文件里去修改default.properties文件的信息

**1) struts.action.extension**

<constant name="struts.action.extension" value="action,do,"></constant>

这样写的话,xxx.action xxx.do xxx都能访问

**2) struts.i18n.encoding**

相当于:写了request.setCharacterEncoding("xxx");即设置请求的编码

**3) struts.devMode**

如果设置为true,则在错误页面上能展示更加详细的信息

**4) struts.objectFactory**

如果为spring表示由Spring负责action的创建

**5) struts.multipart.maxSize**

设置单个上传文件的大小限制

**6)struts.enable.DynamicMethodInvocation**

动态方法调用,默认值是true,如果是true则可以通过 action的name加上感叹号加上方法名去访问这个方法,好处是这个方法不用在xml文件里配置,例如 /test/login!register.action

不过这个特点会导致暴露方法名称,所以一般不用，但是在开发时可以使用

# 7.action的默认配置

关于一些默认值 3.5struts.xml中有讲解

默认情况:/WEB-INF目录下的文件是只能通过转发才能访问到.

|  |
| --- |
| <struts>  <package name="xxx" extends="struts-default" namespace="/">  <!--配置Action-->  <action name="login">  <result>/WEB-INF/jsp/01.jsp</result>  </action>  </package>  </struts> |

# 8.action的属性注入

如果一个action里想要使用一个属性,这个属性的值是通过xml文件来设置的,这样的方式就是属性注入(自己理解的啊)

注入步骤:

1. 在Action类中声明一个成员变量,用于接受xml配置文件传入的内容
2. 在Action类提供一个该变量的setter方法,该方法接收了xml配置的内容
3. 在对应的struts.xml文件中,找到对应的action对象的配置,然后在action中使用

UploadAction.java:

|  |
| --- |
| public class UploadAction extends ActionSupport{  //1)在action中提供一个属性  private String savePath;  //2)提供属性的一个set方法,用于外部的action的参数进行注入  public void setSavePath(String savePath) {  this.savePath = savePath;  }  //文件上传的方法  public String upload() throws Exception{  //1)已经获取了用户的文件  File file = new File("F:/01.txt");  //2)保存到服务器硬盘  FileUtils.copyFile(file,new File(savePath + file.getName()));  System.out.println(savePath);  return SUCCESS;  }  } |

struts.xml中:

|  |
| --- |
| <struts>  <package name="xxx" extends="struts-default" namespace="/">  <!--配置Action-->  <action name="upload" class="lw.pers.action.UploadAction" method="upload">  <!--//使用该配置可以给action对象的属性注入内容-->  <!--param的属性name的值是set方法的名字,不过要去掉set,且名字要遵守变量的命名-->  <!--setSavePath > savePath-->  <param name="savePath">E:/</param>  <result>/login.jsp</result>  </action>  </package>  </struts> |

# 9.struts2的数据共享的三种方式

当业务方法比较多的时候,用方式3,很少的时候用方式2

如果要获取其他的api,则使用方式1

## 9.1方式1

**使用ServletActionContext对象**

|  |
| --- |
| public class UserAction extends ActionSupport{  public String list()throws Exception {  //1)从数据库得到数据  List<String> list = new ArrayList<>();  list.add("eric");  list.add("jacky");  list.add("rose");  //2)用request,session,context域对象来共享数据  /\*\*  \* struts2提供了一种使用域对象的方法  \* ServletActionContext对象,可以在Struts的action方法中使用域名=对象  \*/  //获取request域对象  HttpServletRequest request = ServletActionContext.getRequest();  request.setAttribute("request\_list",list);  //获取session域对象  HttpSession session = request.getSession(true);  session.setAttribute("session\_list",list);  //获取ServletContext域对象  ServletContext servletContext = ServletActionContext.getServletContext();  servletContext.setAttribute("context\_list",list);  return SUCCESS;  }  } |

## 9.2方式2

**使用ActionContext对象:这个对象表示action的上下文,在ActionContext对象中提供操作不同域对象数据的Map集合**

**相对于方式1的特点:**

1)不依赖于servlet原生的api

2)操作数据方便

请注意:ServletActionContext是继承了ActionContext

|  |
| --- |
| public class UserAction extends ActionSupport{  //业务方法是在拦截器之后执行的,所以ActionContext被拦截器成功的创建  //构造方法是在拦截器创建之前就被创建了的,导致不能在构造函数中创建各种域对象,这样就使得每个业务方法都要创建域,代码重复.  public String list()throws Exception {  //1)从数据库得到数据  List<String> list = new ArrayList<>();  list.add("eric");  list.add("jacky");  list.add("rose");  //2)用request,session,context域对象来共享数据  ActionContext context = ActionContext.getContext();  //得到操作request域的map集合,操作这个Map集合就等同于操作了request域  //request域  Map<String, Object> requestMap = context.getContextMap();  requestMap.put("request\_list",list);//存放到这个request域中    //session域  Map<String, Object> session = context.getSession();  session.put("session\_list",list);  //Context域  Map<String, Object> application = context.getApplication();  application.put("context\_list",list);  return SUCCESS;  }  } |

## 9.3方式3

实现接口

|  |
| --- |
| public class UserAction extends ActionSupport implements RequestAware,SessionAware,ApplicationAware {  //strut2会把操作request域的map集合传入  private Map<String,Object> requestMap;  private Map<String,Object> sessionMap;  private Map<String,Object> contextMap;  @Override  public void setRequest(Map<String, Object> request) {  this.requestMap = request;  }  @Override  public void setSession(Map<String, Object> session) {  this.sessionMap = session;  }  @Override  public void setApplication(Map<String, Object> application) {  this.contextMap = application;  }    public String list() throws Exception {  //1)从数据库得到数据  List<String> list = new ArrayList<>();  list.add("eric");  list.add("jacky");  list.add("rose");  //2)用request,session,context域对象来共享数据  //request域  requestMap.put("request\_list",list);  //session域  sessionMap.put("session\_list",list);  //context域  contextMap.put("context\_list",list);  return SUCCESS;  }  } |

# 10.请求参数数据的封装

接受来自jsp页面提交的数据,在原来的Servlet中使用getParamter(“xxx”)来接受

这里是采用拦截器,拦截器(ParametersInterceptor)会调用action的set方法来设置参数值

## 10.1方式1

普通变量的接受

|  |
| --- |
| public class UserAction extends ActionSupport{  private String name;  private String password;  private String gender;  private String[] hobit;  //参数通过这个set方法注入到Action中  public void setPassword(String password) {  this.password = password;  }  public void setGender(String gender) {  this.gender = gender;  }  public void setHobit(String[] hobit) {  this.hobit = hobit;  }  public void setName(String name) {  this.name = name;  }  public String register() throws Exception{  //获取表单的数据  System.out.println("name:" + name);  System.out.println("password:" + password);  System.out.println("gender:" + gender);  System.out.println("hobit" + Arrays.toString(hobit));  return SUCCESS;  }  } |

|  |
| --- |
| <body>  <form action="${pageContext.request.contextPath}/data/user\_register.action" method="post">  用户名:<input type="text" name="name"><br>  密码:<input type="password" name="password"><br>  性别:<input type="radio" name="gender" value="男">男  <input type="radio" name="gender" value="女">女  <br>  爱好:<br>  <input type="checkbox" name="hobit" value="篮球">篮球  <input type="checkbox" name="hobit" value="足球">足球  <input type="checkbox" name="hobit" value="羽毛球">羽毛球  <input type="submit" value="注册">  </form>  </body> |

## 10.2方式2(推荐)

javabean封装

User.java:

|  |
| --- |
| public class User {  @Override  public String toString() {  return "User{" +  "name='" + name + '\'' +  ", password='" + password + '\'' +  ", gender='" + gender + '\'' +  ", hobit=" + Arrays.toString(hobit) +  '}';  }  //注意getter方法也要有  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getPassword() {  return password;  }  public void setPassword(String password) {  this.password = password;  }  public String getGender() {  return gender;  }  public void setGender(String gender) {  this.gender = gender;  }  public String[] getHobit() {  return hobit;  }  public void setHobit(String[] hobit) {  this.hobit = hobit;  }  private String name;  private String password;  private String gender;  private String[] hobit;  } |

|  |
| --- |
| public class UserAction extends ActionSupport{  private User user;  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  public String register() throws Exception{  System.out.println(user);  return SUCCESS;  }  } |

|  |
| --- |
| <body>  <form action="${pageContext.request.contextPath}/data/user\_register.action" method="post">  用户名:<input type="text" name="user.name"><br>  //这里的user就是action里的user  密码:<input type="password" name="user.password"><br>  性别:<input type="radio" name="user.gender" value="男">男  <input type="radio" name="user.gender" value="女">女  <br>  爱好:<br>  <input type="checkbox" name="user.hobit" value="篮球">篮球  <input type="checkbox" name="user.hobit" value="足球">足球  <input type="checkbox" name="user.hobit" value="羽毛球">羽毛球  <input type="submit" value="注册">  </form>  </body> |

## 10.3自动类型的转换

struts2把接受到的String类型的参数自动转化为不同的基本类型和BeanUtils组件的功能一样

|  |
| --- |
| public class User {  private String name;  private int age;  private double score;  private Date birth;  @Override  public String toString() {  return "User{" +  "name='" + name + '\'' +  ", age=" + age +  ", score=" + score +  ", birth=" + birth +  '}';  }  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 double getScore() {  return score;  }  public void setScore(double score) {  this.score = score;  }  public Date getBirth() {  return birth;  }  public void setBirth(Date birth) {  this.birth = birth;  }  } |

## 10.4自定义类型转换

10.3中例如input 标签中

<input type=”text” name=”birth”>

在javabean中的数据类型是Date

如果这个birth是”1994-03-22”则是可以的,如果是1994/03/22则报错,因此我们需要

自定义类型转换

StrutsTypeConverter

绑定自定义转换器的方式:

1. 局部绑定(只能绑定一个action)
2. 建立一个Action的文件名-conversion.properties
3. 必须要将上面的文件放到和要绑定的Action放在同一个目录下

示例:

MyDateConverter.java

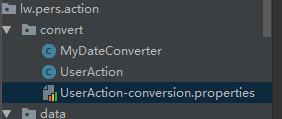
|  |
| --- |
| package lw.pers.action.convert;  import org.apache.struts2.util.StrutsTypeConverter;  import java.text.ParseException;  import java.text.SimpleDateFormat;  import java.util.Date;  import java.util.Map;  /\*\*  \* 自定义日期类型转换器  \*/  public class MyDateConverter extends StrutsTypeConverter{  /\*\*  \* 从页面的数据到服务器的转换  \* 参数1 context:值栈上下文对象  \* 参数2 values:从页面传递过来的参数值  \* 参数3 toClass:转换到的类型 String->java.util.Date  \*/  private SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");  @Override  public Object convertFromString(Map context, String[] values, Class toClass) {  System.out.println("进入了转换器");  //1)判断values是否为空  if (values == null || values.length == 0) {  return null;  }  if (values.length > 1) {  //多个值的情况  Date[] dates = new Date[values.length];  //2)取出数据进行转换  for (int i = 0; i < dates.length; i++) {  Date date = null;  try {  date = sdf.parse(values[i]);  } catch (ParseException e) {  e.printStackTrace();  }  dates[i] = date;  }  return dates;  } else {  //一个值的情况  Date date = null;  try {  date = sdf.parse(values[0]);  } catch (ParseException e) {  e.printStackTrace();  }  return date;  }  }  /\*\*  \* 从服务器的数据到页面的转换  \*/  @Override  public String convertToString(Map context, Object o) {  return null;  }  } |

UserAction.java:

|  |
| --- |
| package lw.pers.action.convert;  import com.opensymphony.xwork2.ActionSupport;  import lw.pers.action.entity.User;  public class UserAction extends ActionSupport{  private User user;  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  public String register() throws Exception{  System.out.println(user);  return SUCCESS;  }  } |

UserAction-conversion.properties:

|  |
| --- |
| user.birth=lw.pers.action.convert.MyDateConverter |



1. 全局绑定(绑定整个项目的action)
2. 建立一个xwork-conversion.properties
3. 该文件一定要放在src目录下

例子:

xwork-conversion.properties:

|  |
| --- |
| java.util.Date=lw.pers.action.convert.MyDateConverter |

# 11.文件的上传和下载

## 11.1上传

和请求参数数据的封装差不多,文件上传也是这样的

请求页面:<input type=”file” name=”attach”>

服务器的action里要定义几个变量和setter方法来接受客户端上传过来的文件信息.

action:

|  |
| --- |
| public class UploadAction extends ActionSupport{  //1)这个变量名字是和<input type="file" name="attache"> 中的name的值一样  private File attach; //接受上传文件  private String attachContentType;//接受文件类型,写法固定的  private String attachFileName;//接受文件名称,固定写法  private String info; //接受描述  private String savePath;  public void setSavePath(String savePath) {  this.savePath = savePath;  }  //给setter方法  public void setAttach(File attach) {  this.attach = attach;  }  public void setAttachContentType(String attachContentType) {  this.attachContentType = attachContentType;  }  public void setAttachFileName(String attachFileName) {  this.attachFileName = attachFileName;  }  public void setInfo(String info) {  this.info = info;  }  @Override  public String execute() throws Exception {  System.out.println(attach);  System.out.println(attachContentType);  System.out.println(attachFileName);  System.out.println(info);  FileUtils.copyFile(attach,new File(savePath+attachFileName));  return super.execute();  }  } |

上传配置:

1. 色画质临时文件存放路径

<constant name="struts.multipart.saveDir" value="E:/tmp/"></constant>

1. 设置单个上传文件大小的限制

<constant name="struts.multipart.maxSize" value="100000000"></constant>

1. 对上传文件类型的限制

在action里配置:

|  |
| --- |
| <package name="upload" extends="struts-default" namespace="/upload">  <action name="upload" class="struts.UploadAction" method="execute">  <interceptor-ref name="defaultStack">  <!--//文件上传的类型-->  <param name="fileUpload.allowedTypes">  image/jpg,text/plain  </param>  <!--//文件后缀名-->  <param name="fileUpload.allowedExtensions">  jpg,txt,py  </param>  </interceptor-ref>  <result name="upload">/index.jsp</result>  <param name="savePath">E:/tmp/</param>  </action>  </package> |

注意:如果有两个配置的话,取他们的交集

upload.jsp

|  |
| --- |
| <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <html>  <head>  <title>上传</title>  </head>  <body>  <form action="${pageContext.request.contextPath}/upload/upload.action" method="post" enctype="multipart/form-data">  文件:<input type="file" name="attach"> 说明:<input type="text" name="info"><br>  <input type="submit" value="提交">  </form>  </body>  </html> |

## 11.2下载

action:

|  |
| --- |
| public class DownAction extends ActionSupport{  private String serverPath;  //接受要下载的文件名  private String name;  public void setServerPath(String serverPath) {  this.serverPath = serverPath;  }  //显示文件列表  public String list()throws Exception{  File file = new File(serverPath);  String[] list = file.list();//如果是文件则返回null,如果是目录则返回目录下的所有文件名(包括隐藏的)  //把文件列表转到jsp页面去显示  ActionContext context = ActionContext.getContext();  Map<String, Object> contextMap = context.getContextMap();  contextMap.put("list",list);  return "list";  }  //下载的方法,这里需要提供给struts2写出数据的输入流  private InputStream inputStream; //这里inputStream和struts.xml中的inputStream一样  public InputStream getInputStream() {  FileInputStream fileInputStream = null;  try {  fileInputStream = new FileInputStream(new File(serverPath+name));  } catch (FileNotFoundException e) {  e.printStackTrace();  }  return fileInputStream;  }  public String down()throws Exception{  return "down";  }  public String getName() {  try {  //注意这里要进行URL编码,就像访问一个网站时的URL会被编码一样,只不过这是浏览器自动帮你编码了,但是这里需要手动编码  return URLEncoder.encode(name,"utf-8");  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  }  public void setName(String name) {  this.name = name;  }  } |

struts.xml:

|  |
| --- |
| <package name="down" extends="struts-default" namespace="/down">  <action name="down\_\*" class="lw.pers.action.upload\_down.DownAction" method="{1}">  <param name="serverPath" >  C:\Users\lw\Pictures\archlinux\  </param>  <result name="list">/listFile.jsp</result>  <!--视图的类型一定要是stream-->  <result name="down" type="stream">  <!--往StreamResult类中的属性注入内容-->  <!--返回给浏览器的文件类型,返回通用二进制类型-->  <param name="contentType">application/octet-stream</param>  <!--返回给浏览器的输入流-->  <!—和action的getter一样-->  <param name="inputName">inputStream</param>  <!--告诉浏览器以下载的方式下载资源-->  <!--<param name="contentDisposition">attachment;filename="document.pdf"</param>-->  <!--这里的${name}相当于是调用action里的getName()方法-->  <param name="contentDisposition">attachment;filename=${name}</param>  <!--缓存大小-->  <param name="bufferSize">1024</param>  </result>  </action>  </package> |

listFile.jsp:

|  |
| --- |
| <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core\_rt" %>  <html>  <head>  <title>显示所有的文件</title>  </head>  <body>  <table>  <tr>  <th>  编号  </th>  <th>  文件名称  </th>  <th>  操作  </th>  </tr>  <c:forEach items="${list}" var="file" varStatus="varSta">  <tr>  <td>${varSta.count}</td>  <td>${file}</td>  <td><a href="${pageContext.request.contextPath}/down/down\_down?name=${file}">下载</a></td>  </tr>  </c:forEach>  </table>  </body>  </html> |

# 12.拦截器(Interceptor)

拦截器类似于过滤器的功能

过滤器:可以过滤项目的任何请求(servlet/jsp/html/img)

拦截器:只能拦截Action资源,拦截完Action,添加逻辑代码.

## 12.1拦截器编写规则

struts2提供了Interceptor接口用于自定义拦截器

1)编写类,实现Interceptor接口

|  |
| --- |
| package lw.pers.action.Interceptor;  import com.opensymphony.xwork2.ActionInvocation;  import com.opensymphony.xwork2.interceptor.Interceptor;  public class MyInterceptor1 implements Interceptor {  public MyInterceptor1(){  System.out.println("1)创建了拦截器1对象");  }  @Override  public void destroy() {  System.out.println("destroy");  }  @Override  public void init() {  System.out.println("2)init");  }  @Override  public String intercept(ActionInvocation invocation) throws Exception {  System.out.println("4)拦截了Action--前面的代码");  invocation.invoke();//放行,调用下一个拦截器,如果没有下一个拦截器,则调用action  System.out.println("6)拦截了Action--后面的代码");  return null;  }  } |

2)在struts.xml文件中定义和使用拦截器

局部拦截器

|  |
| --- |
| <package name="inter" extends="struts-default" namespace="/inter">  <!--定义拦截器-->  <interceptors>  <interceptor name="inter1" class="lw.pers.action.Interceptor.MyInterceptor1">  </interceptor>  </interceptors>  <action name="user\_\*" class="lw.pers.action.Interceptor.UserAction" method="{1}">  <!--使用定义好了的拦截器,注意这个拦截器是局部的,值对这个Action起作用-->  <interceptor-ref name="inter1"></interceptor-ref>  <result>/index.jsp</result>  </action>  </package> |

全局拦截器配置:

|  |
| --- |
| <package name="inter" extends="struts-default" namespace="/inter">  <!--定义拦截器-->  <interceptors>  <interceptor name="inter1" class="lw.pers.action.Interceptor.MyInterceptor1">  </interceptor>  </interceptors>  <!--全局作用的拦截器,这么写会覆盖struts-defalut.xml中的默认拦截器-->  <default-interceptor-ref name="inter1"></default-interceptor-ref>  </package> |

## 12.2拦截器的生命周期

启动:

1. 拦截器对象创建,调用拦截器的init方法

访问:

1. 创建Action对象
2. 执行拦截器的intercept方法
3. 执行拦截器前面的代码(invocation.invoke();方法之前的)
4. 执行invocation.invoke();放行(执行下一个拦截器或Action方法)
5. 执行拦截器后面的代码(invocationinvoke(方法之后的)

## 12.3拦截器栈

概念:当一个或多个action同时被多个拦截器所拦截,就可以使用拦截器

|  |
| --- |
| <package name="inter" extends="struts-default" namespace="/inter">  <!--定义拦截器-->  <interceptors>  <interceptor name="inter1" class="lw.pers.action.Interceptor.MyInterceptor1"></interceptor>  <interceptor name="inter2" class="lw.pers.action.Interceptor.MyInterceptor2"></interceptor>  <!--定义拦截器栈,可以将拦截器栈当作拦截器-->  <interceptor-stack name="interStack">  <interceptor-ref name="inter1"></interceptor-ref>  <interceptor-ref name="inter2"></interceptor-ref>  </interceptor-stack>  </interceptors>  <!--全局作用的拦截器,这里也会覆盖默认的拦截器栈-->  <default-interceptor-ref name="interStack"></default-interceptor-ref>  <action name="user\_\*" class="lw.pers.action.Interceptor.UserAction" method="{1}">  <result>/index.jsp</result>  </action>  <action name="book\_\*" class="lw.pers.action.Interceptor.BookAction" method="{1}">  <result>/index.jsp</result>  </action>  </package> |

注意1:定义拦截器栈的时候,引用拦截器栈的顺序决定了创建拦截器对象的顺序.

注意2:当有了拦截器栈的执行顺序:

启动:

1. 创建拦截器inter1对象,调用init方法
2. 创建拦截器inter2对象,调用init方法

访问:

1. Action对象的构造
2. 执行inter1的interceptor方法前面代码
3. 执行inter2的interceptor方法前面代码
4. 执行Action的业务方法
5. 执行inter2的interceptor方法后面代码
6. 执行inter1的interceptor方法后面代码

**正确不覆盖默认拦截器的配置如下**:

|  |
| --- |
| <package name="inter" extends="struts-default" namespace="/inter">  <!--定义拦截器-->  <interceptors>  <interceptor name="inter1" class="lw.pers.action.Interceptor.MyInterceptor1"></interceptor>  <interceptor name="inter2" class="lw.pers.action.Interceptor.MyInterceptor2"></interceptor>  <!--定义拦截器栈,可以将拦截器栈当作拦截器-->  <interceptor-stack name="interStack">  <interceptor-ref name="inter1"></interceptor-ref>  <interceptor-ref name="inter2"></interceptor-ref>  </interceptor-stack>  <interceptor-stack name="myStack">  <interceptor-ref name="defaultStack"></interceptor-ref>  <interceptor-ref name="interStack"></interceptor-ref>  </interceptor-stack>  </interceptors>  <!--全局作用的拦截器-->  <default-interceptor-ref name="myStack"></default-interceptor-ref>  <action name="user\_\*" class="lw.pers.action.Interceptor.UserAction" method="{1}">  <result>/index.jsp</result>  </action>  <action name="book\_\*" class="lw.pers.action.Interceptor.BookAction" method="{1}">  <result>/index.jsp</result>  </action>  </package> |

## 12.4案例-登陆权限

参见:login项目

# 13.值栈(value stack)

## 13.1ONGL表达式

Ongl表达式简介:Ognl是Object Graphic Navigation Language(对象图导航语言)的缩写,它是一个开源项目,struts2框架使用Ognl作为默认的表达式语言

需要的jar包:

ognl-3.xxx.jar

**OGNL工具由三部分构成，分别是**

a. OGNL引擎：负责解释执行OGNL表达式，按表达式区root或map区域寻找数据。

b. Root根对象区域：访问该区域数据OGNL表达式简洁方便。适合放置访问频繁的对象元素。

c. Map对象存储区 ：访问该区域需要使用“#key”访问。适合放置访问不频繁的对象元素

**ognl表达式 vs EL表达式**

EL表达式:获取域对象的数据,不能存放数据(存放数据需要jstl),不能调用方法

ognl表达式:获取域对象的数据,可以存放数据,可以调用方法

ognl表达式的使用:

a.访问Root对象某个简单属性表达式： "属性名"

b.访问Root对象某个数组或者List集合表达式："属性名[0]"

c.访问Root对象某个Map集合表达式："属性名.key"或者"属性名['key']"

d.访问Root对象某个复合类型属性address表达式： "address.子属性"

e.访问Root对象某个复杂属性 List<Address> addresses表达式："addresses[0].子属性"

f.访问Root对象的某个属性的方法表达式："属性名.方法()"

g.支持Java中的表达式和运算符

h.定义一个List常量 表达式：{"A","B","C"}

i.定义一个Map常量表达式：#{"key1":value1,"key2":value2}

j.访问静态常量或方法表达式："@包名.类型@常量名或方法"

上述语法如果访问Map区域一样使用，只要注意追加"#key".

1. 不使用Ognl表达式存取数据

|  |
| --- |
| User user = new User();  user.setName("eric");  user.setAge(20);  //1)创建一个OgnlContext对象  OgnlContext context = new OgnlContext();  //2)把user对象存入OgnlContext对象  context.put("user",user);  //3)从OgnlContext对象取出数据  User user1 = (User) context.get("user");  System.out.println(user1); |

1. 使用Ognl表达式存取属性

|  |
| --- |
| User user = new User();  user.setName("eric");  user.setAge(20);  //1)创建一个OgnlContext对象  OgnlContext context = new OgnlContext();  //2)把user对象存入OgnlContext对象  //context.put("user",user);//往OgnlContext非根对象存数据,注意这里有个key值(即"user"),Map  context.setRoot(user);//往OgnlContext根对象存数据,List  //3)使用Ognl表达式从OgnlContext对象取出数据  try {  //创建一个表达式对象  //Object o = Ognl.parseExpression("#user.name");//从OgnlContext的非根对象中取数据  Object o = Ognl.parseExpression("name");//从OgnlContext根对象中取数据  String name = (String)Ognl.getValue(o,context,context.getRoot());  System.out.println(name);  } catch (OgnlException e) {  e.printStackTrace();  } |

1. Ognl表达式调用静态方法

|  |
| --- |
| Math.round(345.5);  //1)创建一个OgnlContext对象  OgnlContext context = new OgnlContext();  //2)使用Ognl表达式从OgnlContext对象取出数据  Object ognlObj = Ognl.parseExpression("@java.lang.Math@round(45.6)");  Object result = Ognl.getValue(ognlObj,context,context.getRoot());  System.out.println(result); |

jsp:

|  |
| --- |
| <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <%@ taglib prefix="s" uri="/struts-tags" %>  <html>  <head>  <title>查看值栈的数据</title>  </head>  <body>  <s:debug>  </s:debug>  <%--1)取出List栈的数据--%>  <s:property value="user.name"></s:property><br>  <s:property value="a"></s:property><br>  <%--2)取出Map栈的数据--%>  <s:property value="#request.request\_data"></s:property><br>  <s:property value="#session.session\_data"></s:property><br>  <s:property value="#application.application\_data"></s:property><br>  <%--3)取出paramter的数据--%>  <%--如果在Action里有个普通的变量为a=100,此时#parameters.a是没有值的,  如果你在url上写?a=200,此时List栈里的a=会由100变为200,paramters也会变为200 --%>  <%--也就是说客户端的get/post参数会被存放在parameters里和Action的变量里面(List栈)--%>  <s:property value="#parameters.a"></s:property>  <s:property value="#attr.request.request\_data"></s:property>  </body>  </html> |

## 13.2struts2的值栈对象

值栈对象就是实现了ValueStack接口的对象,即OgnlValueStack

ActionContext context = ActionContext.getContext();

ValueStack valueStack = context.getValueStack();

OgnlValueStack

OgnlValueStack对象

根对象(List栈) 非根对象(Map栈)

valueStack.push(obj):存放元素 context.getContextMap.put(“xx”,”xx”)

valueStack.pop(obj):取出元素 context.getSession().put(“xx”,”xx”)

context.getApplication().put(“xx”,”xx”)

Action对象

Provider对象

request域

session域

application域

attr

parameters(from表单)

13.3页面获取值

<s:property>标签可以使用Ognl表达式获取值栈的数据

使用Ognl表达式取OgnlValueStack的数据

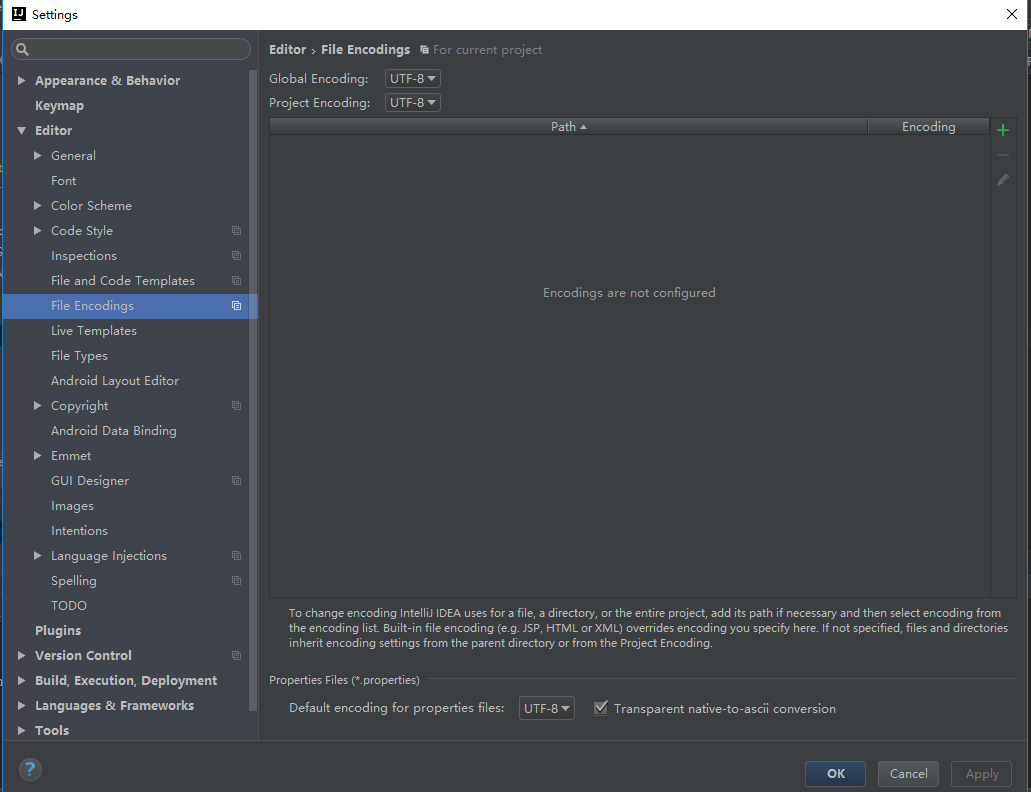
1. List栈(不需要#号,不需要key),直接到list栈的每个对象去搜索对应的属性

搜索规则:

从List栈的栈顶开始搜索,如果有属性->返回,没有的话到栈的第二个元素去搜索属性...

# 14.国际化

注意:如果有中文乱码问题: 右下角的那个勾勾



1) 准备不同的国家资源包

中国: message\_zh\_CN.properties

user=用户民

美国: message\_en\_US.properties

user=user

英国:message\_en\_GB.properties

2)在struts.xml文件中指定资源包加载路径的常量

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

1. 在页面上使用资源包的内容

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

<s:text name="user"></s:text>

5） 在Action里使用资源包的内容

|  |
| --- |
| public class UserAction extends ActionSupport{  @Override  public String execute() throws Exception {  String user = getText("user");  // String user = super.getText("user");  System.out.println(user);  return SUCCESS;  }  } |

# 15.数据验证

**前台校验**:也称之为客户端校验,主要是通过javascript编程方式进行表单数据的验证

**后台校验:**也称之为服务器端校验,指的是使用struts2通过XML配置的方式进行保单数据的验证

struts2提供了4种进行表单 后台校验:

## 15.1方式1

用代码方式对action的所有方法进行验证

步骤:因为Action继承了ActionSupport类,所以只需要在Action类里重写validate()方法就可以了

执行顺序:

1)validate();

2)判断错误Map集合是否有错误,有错误就抛出异常,不执行业务方法,返回input，

如果没有错误,就继续执行业务方法

Action类:

|  |
| --- |
| package lw.pers;  import com.opensymphony.xwork2.ActionSupport;  public class UserAction extends ActionSupport{  private User user;  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  /\*\*  \* 如果validate方法中出现异常,那么就会直接返回错误页面  \* 如果validate没有出现异常,那么直接运行业务方法  \* 注意:struts2是根据异常的Map集合有没有数据来抛出异常的,有数据则抛出异常,没有则不抛出,去执行execute方法  \*/  @Override  public void validate() {  //写表单数据验证的逻辑  if(user.getName()==null||user.getName().equals("")){  //把错误的信息放入信息Map集合  super.addFieldError("user.error.required","用户名不能为空");  }  if(user.getPassword()==null||user.getPassword().equals("")){  //把错误的信息放入信息Map集合  super.addFieldError("password.error.required","密码不能为空");  }  }  @Override  public String execute() throws Exception {  System.out.println("execute");  return SUCCESS;  }  } |

struts.xml:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="user" extends="struts-default" namespace="/user">  <action class="lw.pers.UserAction" name="user" method="execute">  <result>/index.jsp</result>  <result name="input">/login.jsp</result>  </action>  </package>  </struts> |

login.jsp:

|  |
| --- |
| <%@ taglib prefix="s" uri="/struts-tags" %>  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <html>  <head>  <title>Title</title>  <style type="text/css">  ul{  display: inline-block;  list-style-type: none;  }  </style>  </head>  <body>  login界面  <%--显示所有错误信息--%>  <s:fielderror></s:fielderror>  <form action="${pageContext.request.contextPath}/user/user.action">  <input type="text" name="user.name"> <s:fielderror fieldName="user.error.required"></s:fielderror><br>  <input type="password" name="user.password"> <s:fielderror fieldName="password.error.required"></s:fielderror><br>  <input type="submit" value="登陆">  </form>  </body>  </html> |

## 15.2方式2

用代码方式对action的指定方法进行验证

和方式1差不多,不过要将validate 这个方法改名称

改为 validate + 需要验证的方法名称

例如:如果要验证register方法,则 validateRegister()

## 15.3方式3

用xml配置方式对action的所有方法进行验证

方式1和方式2都是使用代码在程序中写死了的,不灵活,不好扩展

xml配置数据验证的规则:

1. 编写一个xml文件,名称:Action文件名-validation.xml
2. 该xml文件必须要放在Action文件的同一个目录

UserAction.java

|  |
| --- |
| package lw.pers;  import com.opensymphony.xwork2.ActionSupport;  public class UserAction extends ActionSupport{  private User user;  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  @Override  public String execute() throws Exception {  System.out.println("execute");  return SUCCESS;  }  public String register() throws Exception {  System.out.println("register");  return SUCCESS;  }  } |

UserAction-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>  <!--一个field能加多个验证规则-->  <!--验证用户名-->  <field name="user.name">  <!--type属性表示xwork中定义的可以使用的验证的类型-->  <field-validator type="requiredstring">  <!--当出现错误的时候,提示的错误信息-->  <message>用户名不能为空</message>  </field-validator>  </field>  <!--验证密码-->  <field name="user.password">  <field-validator type="requiredstring">  <message>密码不能用空</message>  </field-validator>  <field-validator type="regex">  <!--注入正则的字符串-->  <param name="regexExpression">  ^[a-zA-Z0-9]{4,12}  </param>  <message>密码格式不正确</message>  </field-validator>  </field>  </validators> |

login.jsp

|  |
| --- |
| <%@ taglib prefix="s" uri="/struts-tags" %>  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <html>  <head>  <title>Title</title>  <style type="text/css">  ul{  display: inline-block;  list-style-type: none;  }  </style>  </head>  <body>  login界面  <form action="${pageContext.request.contextPath}/user/user.action">  <input type="text" name="user.name"> <s:fielderror fieldName="user.name"></s:fielderror><br>  <input type="password" name="user.password"> <s:fielderror fieldName="user.password"></s:fielderror><br>  <input type="submit" value="登陆">  </form>  </body>  </html> |

jsp中的 <s:fielderror>中的fieldName属性的值和 验证器的.xml中的field标签的name属性的值一样

F:\untitled11\web\WEB-INF\lib\xwork-core-2.3.34.jar!\com\opensymphony\xwork2\validator\validators\default.xml中

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE validators PUBLIC  "-//Apache Struts//XWork Validator Definition 1.0//EN"  "http://struts.apache.org/dtds/xwork-validator-definition-1.0.dtd">  <!-- START SNIPPET: validators-default -->  <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> |

## 15.4方式4

用xml配置方式对action的指定方法进行验证

方式4和方式3一样,只是验证器文件名要改变

Action文件名-访问方法的路径-validation.xml

例如:Useraction的register方法:user\_register

UserAction-user\_execute-validation.xml

UserAction.java:

|  |
| --- |
| package lw.pers;  import com.opensymphony.xwork2.ActionSupport;  public class UserAction extends ActionSupport{  private User user;  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  @Override  public String execute() throws Exception {  System.out.println("execute");  return SUCCESS;  }  public String register() throws Exception {  System.out.println("register");  return SUCCESS;  }  } |

struts.xml:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="user" extends="struts-default" namespace="/user">  <action class="lw.pers.UserAction" name="user\_execute" method="execute">  <result>/index.jsp</result>  <result name="input">/login.jsp</result>  </action>  <action name="user\_register" class="lw.pers.UserAction" method="register">  <result>/index.jsp</result>  </action>  </package>  </struts> |

UserAction-user\_execute-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>  <!--一个field能加多个验证规则-->  <!--验证用户名-->  <field name="user.name">  <!--type属性表示xwork中定义的可以使用的验证的类型-->  <field-validator type="requiredstring">  <!--当出现错误的时候,提示的错误信息-->  <message>用户名不能为空</message>  </field-validator>  </field>  <!--验证密码-->  <field name="user.password">  <field-validator type="requiredstring">  <message>密码不能用空</message>  </field-validator>  <field-validator type="regex">  <!--注入正则的字符串-->  <param name="regexExpression">  ^[a-zA-Z0-9]{4,12}  </param>  <message>密码格式不正确</message>  </field-validator>  </field>  </validators> |

# 16.模型驱动

请求参数数据的封装前面有两种:

方式1:表单: name -> 基本数据类型 : String name

方式2:表单:user.name-> javabean数据类型:User user (User:String name)

方式3:表单:name-> javabean数据类型:User user(User:String name) 模型驱动方式

模型驱动的好处:表单的name值和javabean的属性名是一样,如果Action存在多个对象,则使用方式2

UserAction.java:

|  |
| --- |
| package lw.pers.modedriver;  import com.opensymphony.xwork2.ActionSupport;  import com.opensymphony.xwork2.ModelDriven;  import lw.pers.User;  /\*\*  \* 使用模型驱动方式获取请求数据  \*/  public class UserAction extends ActionSupport implements ModelDriven<User>{  //注意:使用模型驱动的方式,存放数据的模型引用不能为空  private User user = new User();  /\*\*  \* 该方法struts2可以把值栈中的请求参数封装到User对象中,不需要写setter和getter  \*/  @Override  public User getModel() {  return user;  }  @Override  public String execute() throws Exception {  System.out.println(user);  return SUCCESS;  }  } |

struts.xml:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="model" extends="struts-default" namespace="/model">  <action name="model\_\*" class="lw.pers.modedriver.UserAction" method="{1}">  <result name="success">/index.jsp</result>  </action>  </package>  </struts> |

model.jsp

|  |
| --- |
| <%@ taglib prefix="s" uri="/struts-tags" %>  <%@ page contentType="text/html;charset=UTF-8" language="java" %>  <html>  <head>  <title>Title</title>  </head>  <body>  login界面  <form action="${pageContext.request.contextPath}/model/model\_execute.action">  <input type="text" name="name"> <br>  <input type="password" name="password"> <br>  <input type="submit" value="登陆">  </form>  </body>  </html> |

# 17.struts2标签

分为:逻辑标签和UI标签(节省html代码)

1)创建List集合

|  |
| --- |
| <%--ognl表达式:在jsp页面中创建一个List集合--%>  <s:iterator value="{'a','b','c'}" var="name">  <s:property value="#name"></s:property>  </s:iterator> |

2)创建Map集合

|  |
| --- |
| <%--ognl表达式:jsp页面中创建Map集合--%>  <s:iterator value="#{1:'a',2:'b',3:'c'}" var="entry">  编码:<s:property value="#entry.key"></s:property>  姓名:<s:property value="#entry.value"></s:property><br>  </s:iterator> |

# 18.防止重复提交

用户提交的第一次:

1. 得到用户页面的token值
2. 把token放入session中
3. 放行

再次提交时:

1. 对比session中的token和当前用户传过来的token对比,如果一样则处理(表示重复提交了)
2. 不一样,那么就是不重复提交,放行

默认情况下struts2是不支持防止重复提交,需要在struts.xml中添加token的拦截器

struts.xml

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN" "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  <package name="user" extends="struts-default" namespace="/user">  <interceptors>  <interceptor-stack name="myStack">  <interceptor-ref name="defaultStack"></interceptor-ref>  <interceptor-ref name="token"></interceptor-ref>  </interceptor-stack>  </interceptors>  <default-interceptor-ref name="myStack"></default-interceptor-ref>  <action name="user\_\*" class="lw.pers.modedriver.UserAction" method="{1}">  <!--当发生了重复提交后,会返回invalid.token-->  <result name="invalid.token">/error.jsp</result>  <result name="success">/index.jsp</result>  </action>  </package>  </struts> |

login.jsp

|  |
| --- |
| <form action="${pageContext.request.contextPath}/user/user\_execute.action">  <input type="text" name="user.name">  <input type="password" name="user.password">  <%--在当前页面生成一个token值--%>  <s:token></s:token>  <input type="submit" value="登陆">  </form> |