本视频讲解了struts2的基础使用，包括环境搭建、配置详解、基础操作、数据校验、拦截器、文件操作、国际化及ajax交互等实际开发中常用的技能。

学习完本视频，可以熟练掌握struts2的使用，使用struts2完成web开发。

# 1. 环境搭建

## 1.1. 导入jar包

Struts2是Web层的框架，需要创建Web工程。

引入Struts2核心jar包

```
<dependencies>

    <dependency>

        <groupId>org.apache.struts</groupId>

        <artifactId>struts2-core</artifactId>

        <version>2.5.13</version>

    </dependency>

</dependencies>
```

## 1.2. 配置web.xml

```
<!--Struts2核心过滤器-->

<filter>

    <filter-name>Struts2</filter-name>

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

</filter>

<filter-mapping>

    <filter-name>Struts2</filter-name>

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

</filter-mapping>
```

## 1.3. Action类

```
package action;


import com.opensymphony.xwork2.Action;


public class HelloWorldAction implements Action {

    //请求中传递的参数和返回给页面的值都定义成属性

    //必须要给属性写getter/setter方法

    private String username;

    private String message;


    //getter/setter略，自己一定要写

    @Override

    public String execute() throws Exception {

        //查看请求中传递的参数

        System.out.println(username);

        //改变这个message,会自动传递给页面

        message = "hello:" + username;

        //SUCCESS是Action中的常量,值是success

        return SUCCESS;

    }
}
```

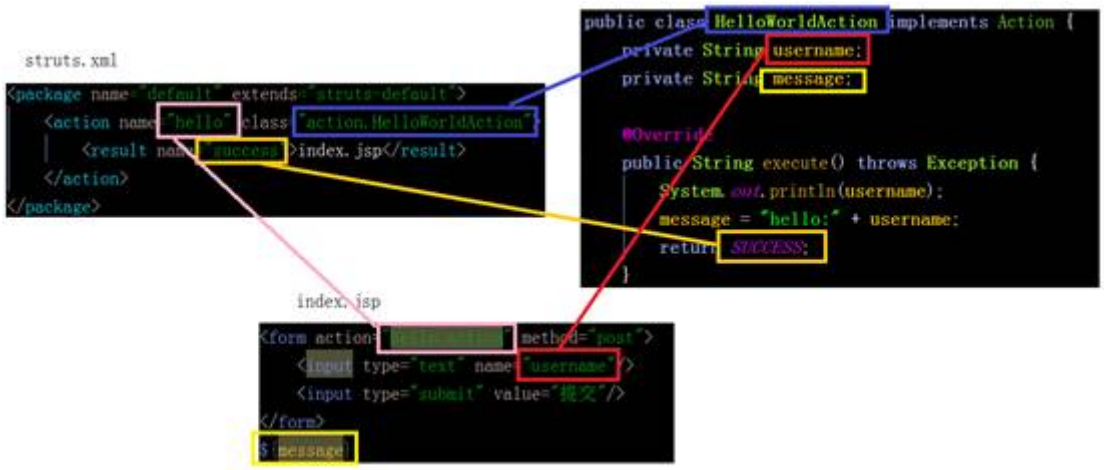
## 1.4. struts.xml

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

<!DOCTYPE struts PUBLIC

        "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"

        "http://struts.apache.org/dtds/struts-2.5.dtd">
```

```
<struts>
    <!--所有的action都放在package中，必须继承struts-default-->
    <!--struts-default中有默认的拦截器配置，能处理参数等信息-->
    <package name="default" extends="struts-default">
        <!--name对应的是请求的地址，class是处理请求的类-->
        <action name="hello" class="action.HelloWorldAction">
            <!--success是Action类中返回的字符串，根据不同字符串返回不同的页面-->
            <result name="success">index.jsp</result>
        </action>
    </package>
</struts>
```

## 1.5. 页面

```
<form action="hello.action" method="post">
    <input type="text" name="username"/>
    <input type="submit" value="提交"/>
</form>
${message}
```




注：struts默认支持的请求后缀是.action，如果需要配置其它的后缀，需要在struts.xml中配置：

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

# 2. struts.xml详解

## 2.1. constant标签

```
<!--设置请求后缀-->

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

<!--设置编码，解决中文乱码-->

<constant name="struts.i18n.encoding" value="utf-8"/>

<!--设置struts标签主题-->

<constant name="struts.ui.theme" value="simple"/>
```

constant用来配置常量。name属性是常量名，value属性是常量值。

constant常量可以改变Struts2的一些行为，比如UI标签的样式、编码格式等。

因为struts2默认的编码格式就是UTF-8，所以不用特意指定编码，中文也不会乱码

## 2.2. package标签

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

package是包。Struts2的package与java中的package类似，可以把同一个业务模块的action和result集中到一个包中，方便管理。不同的是Struts2的包可以继承。比如商品有增删改查操作，订单也有增删该查操作，我们可以将商品和订单的action分别放两个package中方便管理。

**name**属性是包的名字，一个struts.xml中可以有很多个package，通过name属性进行区分。

**namespace**是命名空间，/代表的是根目录。**namespace**的作用类似于SpringMVC中在Controller类上加@RequestMapping注解。相当于此包中所有的action前都加一个父路径。如：

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

    <action name="login" class="action.LoginAction">
```

上面这个name=login的action，在访问的时候路径就是/user/login.action

**extends**属性是继承，通常都会继承struts-default。在struts-default中定义了大量的struts特性，如拦截器和参数处理的功能，如果不继承struts-default，会遇到参数无法绑定或找不到action类。

## 2.3. action标签

```
<action name="login" class="action.LoginAction">

    <!--success是Action类中返回的字符串，根据不同字符串返回不同的页面-->

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

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

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

</action>
```

action标签用来处理请求和响应结果。

name属性是请求的名字，此处不需要加.action。同一个package下的action不能重名。

class属性指定处理该请求的类，是类的全路径。默认的处理请求时会去类中找名为execute的方法。如果不指定class，将默认ActionSupport为处理请求的类。

result标签用来处理请求结果，name属性是Action类中返回的字符串。标签的值是要跳转的页面地址。name如果不写的话，默认是success。

# 3. Action配置

## 3.1. Action简介

Struts2的业务核心是Action类。

```
import com.opensymphony.xwork2.Action;



public class HelloWorldAction implements Action {

    //请求中传递的参数和返回给页面的值都定义成属性

    //必须要给属性写getter/setter方法

    private String username;

    private String message;



    //getter/setter略，自己一定要写

    @Override

    public String execute() throws Exception {

        //查看请求中传递的参数

        System.out.println(username);

        //改变这个message,会自动传递给页面
```

```
        message = "hello:" + username;

        //SUCCESS是Action中的常量,值是success

        return SUCCESS;

    }
```

```
}
```

一个Action业务里可以实现Action接口，也可以继承ActionSupport类。在ActionSupport中提供了一些实现好的业务方法。在以后的编程中，建议全部继承ActionSupport类。

Action中的方法必须返回一个String类型的字符串，这个字符串与struts.xml中result标签的name属性相对应，struts.xml会根据返回的字符串查找对应的页面。

在Action接口中提供了5个常用的结果常量：


## 3.2. 自定义业务方法

在前面的学习中，Action类中只有一个execute方法。可不可以在一个Action类中定义多个业务方法呢？答案是肯定的。

假如用户有注册和登录两个功能，我们可以把这两个功能写进同一个Action类：

```
package action;



import com.opensymphony.xwork2.ActionSupport;

//继承ActionSupport类

public class UserAction extends ActionSupport{

    //处理登录

    public String login(){

        //参数和业务略
```

```
        System.out.println("我是登录");
        return SUCCESS;
    }
    //处理注册
    public String regist(){
        //参数和业务略
        System.out.println("我是注册");
        return SUCCESS;
    }
}
```

注意所有处理请求的业务方法必须是public的，而且要返回一个String

struts.xml中配置：

```
<package name="user" namespace="/user" extends="struts-default">
    <!--通过method指定调用类中的哪个方法-->
    <action name="login" class="action.UserAction" method="login">
        <result name="success">index.jsp</result><!--登录成功去首页-->
        <result name="error">login.jsp</result><!--登录失败回登录页-->
    </action>
    <action name="reg" class="action.UserAction" method="regist">
        <result name="success">index.jsp</result><!--注册成功去首页-->
        <result name="error">regist.jsp</result><!--注册失败回登录页-->
    </action>
</package>
```

method=”login”表示要调用类中的login()方法处理请求。如果找不到login()方法，Struts2会在类中查找doLogin()方法。如果都找不到，将报错。

## 3.3. 动态方法调用

注：Struts2支持动态方法调用，但是不建议使用。仅做考点学习。

如果一个类中有多个业务方法，又不想给每个业务方法都配置一个action标签，可以使用动态党法调用，语法是：请求名!方法名.action

当请求的格式是user!login.action时，代表调用UserAction中的login()方法处理当前请求。当请求的格式是user!regist.action时，代表调用UserAction中的regist()方法处理当前请求。

Struts2中可以开启调用动态方法，设置的常量为：

```
<!--允许调用动态方法-->

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

action的配置：

```
<!--允许动态调用的方法，新版里新增的设置-->

<global-allowed-methods>login,regist</global-allowed-methods>

<action name="user" class="action.UserAction">

    <result name="success">/index.jsp</result><!--成功去首页-->

    <result name="error">/error.jsp</result><!--失败去错误-->

</action>
```

## 3.4. 使用通配符

另一种减少action数量的方法是使用通配符：

```
<global-allowed-methods>login,regist</global-allowed-methods>
```

```
<action name="*User" class="action.UserAction" method="{1}">

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

    <result name="error">{1}.jsp</result><!--失败了就返回原来的页面-->

</action>
```

*User匹配所有已User结尾的请求，method="{1}"中的{1}匹配的就是* User中的*。如果请求的地址是loginUser.action，那么{1}匹配的就是login，就会去类中调用login()方法，并返回相应的结果。

## 3.5. 默认Action

如果在struts.xml中找不到匹配的action，将会报错。可以设置一个默认的action。当所有请求都不匹配时，将匹配默认action。

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

<action name="default">

    <result>error.jsp</result>

</action>
```

对当前的package有效。

action标签的class省略将调用ActionSupport类。result的name省略将默认为success。

注意default-action-ref必须在所有的action标签上面。也就是说default-action-ref出现在action标签之前。不然不符合DTD验证。

# 4. Result配置

常用结果有三种类型：dispatcher、redirect、redirectAction、chain

## 4.1. dispatcher

result的默认类型就是dispatcher。以下两个标签是等价的：

dispatcher的结果等同于Servlet中的请求转发，即：

request.getRequestDispatcher("success.jsp").forward(request, response);

请求转发的意思是当前请求中的参数、属性在下一个页面或请求中仍然可以使用。

```
<result name="success" type="dispatcher">index.jsp</result>

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

## 4.2. redirect

redirect是重定向，重定向之后，当前请求中的参数和属性在下一个页面或请求中将不能使用。

```
<result name="success" type="redirect">index.jsp</result>
```

相当于Servlet中的：response.sendRedirect("success.jsp");

## 4.3. redirectAction

redirectAction与redirect类似，不过redirectAction是重定向到某一个action

```
<action name="reg" class="action.UserAction" method="regist">

    <result name="success" type="redirectAction">login.action</result>

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

</action>
```

如果要调用不同package下的action,需要在result中传参数：

```
<action name="login" class="action.UserAction" method="login">

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

        <!--调用不同package下的action-->

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

```
        <param name="actionName">hello.action</param>
        <!--传递其它参数-->
        <param name="username">123</param>
    </result>
    <result name="error">login.jsp</result>
</action>
```

## 4.4. chain

redirectAction不能共享request中的数据，如果想共享数据，可以将type设置为chain。

```
<action name="reg" class="action.UserAction" method="regist">
    <!--注意chain的action后面没有后缀-->
    <result name="success" type="chain">login</result>
    <result name="error">regist.jsp</result>
</action>
```

## 4.5. 动态结果

```
private String username;
private String page;
//getter/setter方法略

public String login() {
    //参数和业务略
    System.out.println("我是登录");
    if ("admin".equals(username)) {//管理员去管理页
        page = "admin_page";
    } else {//其他人去用户页
        page = "user_page";
    }
```

```
    return SUCCESS;

}
```

result配置：

```
<action name="login" class="action.UserAction" method="login">

    <!--读取action中的属性值，返回不同页面-->

    <result name="success">${page}.jsp</result>

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

</action>
```

## 4.6. 全局结果

一个action中的result只能在当前action中有效。如果多数的action中都用到同一个结果，可以将此结果定义为全局结果。比如login代表用户没有登录，需要跳转到登录页面，那么可以设置成全局结果：

```
<global-results>

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

</global-results>
```

那么所有的用到login的action都不用再配置这个result了。如果action中定义了与全局结果同名的result，将优先使用action中的result。

注意global-results在action之前，default-action-ref之后。具体顺序为：


result-types?

interceptors?

default-interceptor-ref?

default-action-ref?

default-class-ref?

global-results?

global-exception-mappings?

action*

其中？代表出现0次或1次

*代表0次或多次。

# 5. 访问servletAPI

所谓Servlet API，即我们常用的Request、Session、Application等Sevlet对象。

Struts2访问Servlet API有解耦和耦合两种方式

## 5.1. 解耦方式

Struts2将部分Servlet API中的对象封装成Map，可以通过ActionContext获取。获取到的Servlet API对象全部是Map

```
public String regist() {

    ActionContext ac = ActionContext.getContext();

    Map request = (Map) ac.get("request");//获取request

    Map session = ac.getSession();//获取session

    Map application = ac.getApplication();//获取application

    session.put("session_user","123");//向session中存值

    return SUCCESS;

}
```

需要注意的是Session和Application都有对应的get方法，而request没有，需要使用get(“request”)获取。

页面上可以使用

```
${sessionScope.session_user}
```

取值

## 5.2. 耦合方式

耦合方式需要导入servlet-api的jar包

```
<dependency>

    <groupId>javax.servlet</groupId>

    <artifactId>servlet-api</artifactId>

    <version>2.5</version>

</dependency>
```

```
HttpServletRequest request = ServletActionContext.getRequest();

HttpSession session = request.getSession();

HttpServletResponse response = ServletActionContext.getResponse();
```

# 6. Struts标签

与JSTL标签类似，Struts2提供了强大的标签库。

Struts2标签分为UI标签和通用标签。使用Struts2标签需要在页面上引入标签库：

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

## 6.1. 表单标签

struts中的表单标签，语法如下：

```
<s:form action="/user/login.action" method="POST">

    <s:textfield name="username" label="用户名"/>

    <s:password name="password" label="密码"/>

    <s:submit value="登录"/>

</s:form>
```

以上代码等同于：

```
<form action="/user/login.action" method="post">

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

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

    <input type="submit" value="登录"/>

</form>
```

查看页面生成的源代码：

```
<form id="login" name="login" action="/user/login.action" method="POST">
  <table class="wwFormTable">
    <tbody>
      <tr>
        <td class="tdLabel">
          <label class="label" for="login_username">用户名:</label>
        </td>
        <td class="tdInput">...</td>
      </tr>
      <tr>
        <td class="tdLabel">
          <label class="label" for="login_password">密码:</label>
        </td>
        <td class="tdInput">
          <input id="login_password" name="password" type="password">
        </td>
      </tr>
      <tr>
        <td colspan="2">
          <div class="formButton">
            <input id="login_0" value="登录" type="submit">
          </div>
        </td>
      </tr>
    </tbody>
  </table>
</form>
```

可以看到Struts2将元素放入了table。原因是Struts2的标签都有默认主题，默认值为XHTML。如果想使用原始的HTML样式，可以在struts.xml中进行配置：

```
<constant name="struts.ui.theme" value="simple"/>
```

## 6.2. 通用标签

```
<s:if test=""></s:if>

<s:elseif test=""></s:elseif>

<s:else></s:else>
```

s:if标签用于条件判断，相当于jstl中的

```
<c:if test=""></c:if>
```

s:elseif和s:else标签必须与s:if结合使用。

```
<s:iterator value=""  var="" status=""></s:iterator>
```

s:iterator用于集合的遍历，相当于jstl中的

<c:forEach items= ”” var= ”” varStatus= ”” > /c:forEach

下面我们以各行变色为例，使用Struts2的通用标签。

需要注意的是当遍历的集合泛型是基本数据类型和对象类型时，写法不同。

编写IteratorAction类：

```java
public class IteratorAction extends ActionSupport {
    private List<String> strList;
    private List<User> userList;
    //getter/setter方法略

    public String execute() {
        strList = new ArrayList<String>();
        strList.add("元素1");
        strList.add("元素2");
        strList.add("元素3");
        strList.add("元素4");

        User user1=new User();
        user1.setUserName("用户1");
        userList=new ArrayList<User>();
        userList.add(user1);
        userList.add(user1);
        userList.add(user1);
        userList.add(user1);

        return "success";
    }
```

配置struts.xml

```xml
<action name="iterator" class="action.IteratorAction">

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

</action>
```

编写页面iterator.jsp

```jsp
<table>
  <!-- 遍历基本数据类型的集合需要借助id属性 -->
  <s:iterator value="strList" id="str" status="status">
    <!-- status属性可以访问对象的索引或奇偶 -->
    <s:if test="#status.even"><!-- even代表奇数，注意# -->
      <tr style="background-color: green">
        <td><s:property value="str"/></td>
      </tr>
    </s:if>
    <s:else>
      <tr><td><s:property value="str"/></td></tr>
    </s:else>
  </s:iterator>
</table>
```

遍历泛型是基本数据类型的集合，如List，需要通过id属性访问当前的元素。

运行效果：

元素1
元素2
元素3
元素4

```
<table>
  <s:iterator value="userList" status="status">
    <!-- odd代表偶数 -->
    <tr <s:if test="#status.odd">style="background-color: green"</s:if>>
      <td><s:property value="userName"/></td>
    </tr>
  </s:iterator>
</table>
```

遍历泛型是对象类型的集合，如List,不需要使用id，输出的时候直接写对象中的属性名即可：

```
<table>
  <s:iterator value="userList" status="status">
    <!-- odd代表偶数 -->
    <tr <s:if test="#status.odd">style="background-color: green"</s:if>>
      <td><s:property value="userName"/></td>
    </tr>
  </s:iterator>
</table>
```

运行效果：

用户1
用户1
用户1
用户1

# 7. 数据校验

## 7.1. 通用验证

Action类继承ActionSupport，ActionSupport中有一个validate方法进行数据校验

```
@Override

public void validate() {

    if (username == null || username.length() == 0) {

        //向页面中添加错误信息

        addFieldError("username", "用户名不能为空");

    }

}
```

addFieldError要求必须给result配置一个input类型的结果。所以在调用logout方法时，会报找不到result input。

解决办法：

```
<action name="login" class="action.UserAction" method="login">

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

    <!--addFieldError必须配一个input-->

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

</action>
```

页面上显示：

```
<s:fielderror /><!--显示所有错误信息-->

<s:fielderror name="username"/><!--显示指定字段错误信息-->
```

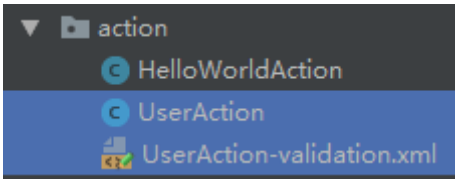
## 7.2. 方法验证

validate()方法会验证当前Action类中所有的方法，如果只想验证其中的一个方法，可以使用validateXxx()方法，其中Xxx是被验证的方法名，首字母大写。

```
//只验证login方法，不验证其它方法

public void validateLogin() {

    if (username == null || username.length() == 0) {

        //向页面中添加错误信息

        addFieldError("username", "用户名不能为空");

    }

}
```

## 7.3. 校验框架

验证框架是把验证信息都写在xml文件中，对某一个Action类进行验证，需要在Action类的同一个包下创建xml文件，文件命名为Action类的类名-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">
            <param name="trim">true</param>
            <message>用户名不能为空</message>
        </field-validator>
    </field>
    <field name="password">
        <field-validator type="requiredstring">
            <message>密码不能为空</message>
        </field-validator>
        <field-validator type="stringlength">
            <param name="maxLength">10</param>
            <param name="minLength">6</param>
            <message>密码必须在${minLength}和${maxLength}</message>
        </field-validator>
    </field>
    <field name="repwd">
        <field-validator type="fieldexpression">
            <param name="expression">password=repwd</param>
            <message>两次密码不一致</message>
        </field-validator>
    </field>
</validators>
```

validators标签:在校验框架中，所有的验证都写在validators标签中

field标签:每一个需要验证的属性都是一个field标签，name指定要验证那个属性。

field-validator标签：代表一种验证规则，通过type指定规则。

| required | 必填 |
| --- | --- |
| requiredstring | 必填字符串，长度大于0，可使用trim参数去除空格 |
| stringlength | 限制字符串长度，指定maxLength和minLength |
| regex | 正则表达式，通过参数regex指定正则表达式 |
| fieldexpression | 字段之间的逻辑关系 |

param标签：给校验器传递的参数

message标签：给页面的提示信息。

校验框架和validate()方法一样，会验证action中所有的方法，如果只验证某一个方法，需要将xml文件的名字命名为action类名-动作名-validation.xml。其中的动作名是指struts.xml中对应此方法的action的name。

多种验证方式并存时，执行顺序是：校验框架->validateXxx()->validate()

# 8. 拦截器

拦截器可以在请求进入action之前做预处理，比如判断用户是否登录。也可以在action执行之后进行处理。

## 8.1. 拦截器配置

例如利用拦截器计算action的执行时间：

在interceptor包下创建TimeInterceptor

```
package interceptor;



import com.opensymphony.xwork2.ActionInvocation;

import com.opensymphony.xwork2.interceptor.Interceptor;

//拦截器的类需要实现Interceptor

public class TimeInterceptor implements Interceptor {

    //执行拦截的方法

    @Override

    public String intercept(ActionInvocation actionInvocation) throws Exception {

        Long startTime=System.currentTimeMillis();

        System.out.println("action执行");

        //invoke方法将执行action，result是action中方法返回的字符串

        String result=actionInvocation.invoke();
```

```
        System.out.println("action执行结束");
        Long endTime=System.currentTimeMillis();
        System.out.println("共运行："+(endTime-startTime));
        return result;
    }

    @Override
    public void destroy() {
        System.out.println("destroy");
    }

    @Override
    public void init() {
        System.out.println("init");
    }
}
```

在struts.xml中配置拦截器，注意标签的位置

```
<interceptors>
    <interceptor name="time" class="interceptor.TimeInterceptor"/>
</interceptors>
```

在需要拦截的action中加入拦截器引用：

```
<action name="hello" class="action.HelloWorldAction">
    <interceptor-ref name="time"/>
    <result name="success">index.jsp</result>
</action>
```

经过以上配置，访问hello.action的时候就会先进入拦截器了。

如果当前包中所有的action都需要勘界，可以配置默认拦截器

```
<default-interceptor-ref name="time"/>
```

## 8.2. 登录拦截器

```
package interceptor;



import com.opensymphony.xwork2.Action;

import com.opensymphony.xwork2.ActionContext;

import com.opensymphony.xwork2.ActionInvocation;

import com.opensymphony.xwork2.interceptor.Interceptor;



import java.util.Map;



public class LoginInterceptor implements Interceptor{

    @Override

    public String intercept(ActionInvocation actionInvocation) throws Exception {

        ActionContext ac = ActionContext.getContext();

        Map session=ac.getSession();

        //如果session中没有用户，就返回登录

        if(session.get("SESSION_USER")==null){

            return Action.LOGIN;

        }else{//如果session中有用户，就进入action

            return actionInvocation.invoke();

        }

    }

    @Override

    public void destroy() {

    }



    @Override

    public void init() {
```

```
    }

}
```

## 8.3. 拦截器栈

```
<interceptors>

    <interceptor name="time" class="interceptor.TimeInterceptor"/>

    <interceptor name="login" class="interceptor.LoginInterceptor"/>

    <!--拦截器栈，多个拦截器捆绑在一起执行-->

    <interceptor-stack name="time-login">

        <interceptor-ref name="time"/>

        <interceptor-ref name="login"/>

        <interceptor-ref name="defaultStack"/>

    </interceptor-stack>

</interceptors>
```

如果自定义了拦截器栈，一定要在最后引用defaultStack，否则action中将无法获得请求中的参数。

# 9. 文件上传下载

## 9.1. 文件上传

```
public class FileAction extends ActionSupport {

    private File upload;//与页面中input的name对应

    private String uploadFileName;//与文件属性对应后面+FileName

    //getter/setter略



    public String upload() throws Exception {

        byte[] buffer = new byte[1024];

        FileInputStream fis = new FileInputStream(upload);

        FileOutputStream fos = new FileOutputStream("E:/upload/" + uploadFileName);

        int length = fis.read(buffer);
```

```
        while (length > 0) {
            fos.write(buffer, 0, length);
            length = fis.read(buffer);
        }
        fis.close();
        fos.flush();
        fos.close();
        return SUCCESS;
    }
}
```

Action代码中的upload与页面表单中的upload相对应。使用xxFileName封装文件名，其中的xx指代File变量的名字。

struts.xml中配置

```
<action name="upload" class="action.FileAction" method="upload">
    <result name="success">upload.jsp</result>
</action>
```

页面：

```
<form action="upload.action" method="post" enctype="multipart/form-data">
    <input type="file" name="upload"/><!--与action中属性对应-->
    <input type="submit" value="上传"/>
</form>
```

## 9.2. 文件下载

Action中的代码

```
private String fileName;//要下载的文件名
//getter/setter略
private InputStream inputStream;//文件流，只写getter方法

public InputStream getInputStream() throws Exception {
    return new BufferedInputStream(new FileInputStream("e:/upload/" + fileName));
```

```
}


public String download() {//这个方法里啥也不用干
    return SUCCESS;
}
```

struts中的配置

```
<action name="down" class="action.FileAction" method="download">
    <result name="success" type="stream">
        <param name="contentType">application/actet-stream</param>
        <param name="inputName">inputStream</param>
        <param name="contentDisposition">
            attachment;filename="${fileName}"
        </param>
    </result>
</action>
```

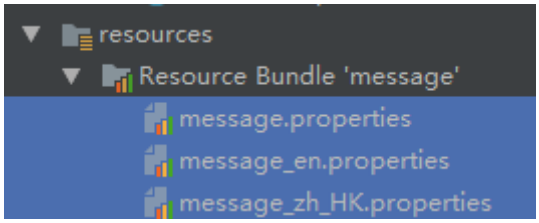
页面：

```
<a href="down.action?fileName=1509184730189.jpg">下载</a>
```

# 10. 国际化

国际化是可以根据用户浏览器默认语言的不同，显示不同的文字。比如用户默认是英语，那么就显示英文界面，是汉语就显示汉语界面。

实现国际化需要创建语言资源文件，我们以中文简体、中文繁体、英语者三种语言为例：




在src下创建三个资源文件,默认资源文件是message.properties，英语资源是message_en.properties，中文繁体是message_zh_HK.properties

如果默认资源文件叫xx.properties，那么英文文件叫xx_en.properties，中文繁体叫xx_zh_HK.properties，中文简体叫xx_zh_CN.properties。

文件内容如下：

message.properties | message_en.properties | message_zh_HK.properties

| name | value | name | value | name | value |
|---|---|---|---|---|---|
| username | 用户名 | username | username | username | 用戶名 |
| password | 密码 | password | password | password | 密碼 |
| submit | 登录 | submit | login | submit | 登錄 |

注意几个资源文件中的name值都是一样的，value转换成不同的语言。

在struts.xml中指定资源文件的名字：

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

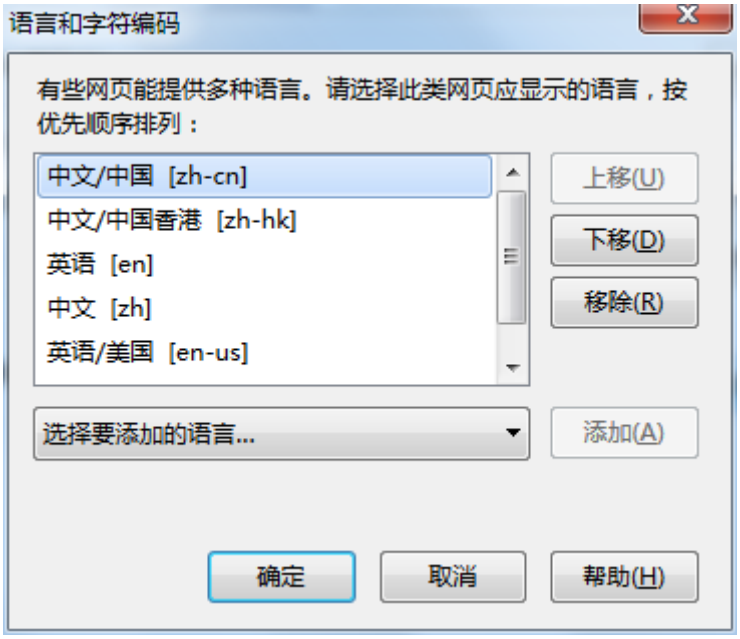
代表资源文件名是message开头的。

页面上：

```
<form action="login.action" method="post">
  <s:text name="username"/>: <input type="text" name="userName"/><br/>
  <s:text name="password"/>: <input type="password" name="password"/><br/>
  <input type="submit" value="<s:text name="submit"/>"/>
或者
  <s:submit value="%{getText('submit')}"/>
```

通过 s:text/ 标签从配置文件中读取相应的字符串，name值对应properties文件中的name。比较特殊的是submit标签，除了使用 s:text ，还可以使用%{getText(‘name值’)}

修改浏览器的语言，想要使用哪种语言就将该语种上移到第一个，刷新页面即可看到效果：




中文简体：

用户名：
密码：
登录

中文繁体：

用戶名：
密碼：
登錄

英文：


数据校验中的提示信息也可以实现国际化：

在验证方法中：

```
addFieldError("userName", getText("配置文件中的name"));
```

通过getText方法读取配置文件中的name。

在校验框架中：

```
<message key="配置文件中的name"/>
```

# 11. 返回json数据

需要添加json依赖包：

```
<dependency>

    <groupId>org.apache.struts</groupId>

    <artifactId>struts2-json-plugin</artifactId>

    <version>2.5.13</version>

</dependency>
```

Action类

```
public class JsonAction extends ActionSupport {

    private Map<String, Object> resultMap;

    //getter/setter略

    public String getJson() {

        resultMap=new HashMap<>();

        resultMap.put("key1",123);

        resultMap.put("key2","abc");

        return SUCCESS;

    }
```

```
}
```

这个action所在的包必须继承json-default

```
<package name="default" namespace="/" extends="struts-default,json-default">

    <action name="json" class="action.JsonAction" method="getJson">

        <result type="json">

            <param name="root">resultMap</param>

        </result>

    </action>
```

key1

```
<param name="includeProperties">key1</param>
```