回顾：

Struts第二天:

Struts配置

# 通配符、动态方法调用

# 全局跳转配置、配置的默认值、常量配置

Struts核心业务

# 请求数据的自动封装 (param 拦截器)

# 类型转换

自动类型转换(struts提供)

类型转换器

$ 自定义局部

---》 ActionClassName-conversion.properties

$ 自定义全局

---》xwork-conversion.properties

# 数据处理

|--ActionContext

|-- ServletActionContext

# 文件上传与下载

Jsp: <input type=file name=file1 >

Action:

(上传，拦截器完成的)

Privat File file1; // 拿到上传的文件对象

Private String file1FileName; // 文件名

Private String file1ContentType;//类型

目标：

拦截器

(国际化)

Ognl表达式语言与Struts标签

# 1. 拦截器

## 1.1 概述

* **基本概念**

Intercetor, 即为拦截器。

1） 在Struts2中，把每一个功能都用一个个的拦截器实现；用户想用struts的哪个功能的时候，可以自由组装使用。

2）Struts2中，为了方法用户对拦截器的引用，提供了拦截器栈的定义，里面可以包含多个拦截器。 文件夹(文件, 文件2) 拦截器栈(拦截器,拦截器2)

3）Struts2中，如果用户没有指定执行哪些拦截器，struts2有一个默认执行的栈，defaultStack;

**一旦如果用户有指定执行哪些拦截器，默认的拦截器栈就不会被执行**

拦截器的设计，就是基于组件设计的应用！

* **拦截器配置举例**

**s**truts-default.xml文件中，定义了struts提供的所有拦截器！

|  |
| --- |
| //1. 定义拦截器以及拦截器栈  <interceptors>  1.1 拦截器定义  <interceptor name="" class="" />    1.2 拦截器栈的定义  <interceptor-stack name="defaultStack">  引用了上面拦截器(1.1)  </interceptor-stack>  </interceptors>  2. 默认执行的拦截器(栈)  <default-interceptor-ref name="defaultStack"/> |

* **API**

**|-- Interceptor 拦截器接口**

**|-- AbstractInterceptor 拦截器默认实现的抽象类； 一般用户只需要继承此类即可继续拦截器开发**

**|-- ActionInvocation 拦截器的执行状态，调用下一个拦截器或Action**

拦截器与过滤器区别：

….

## 1.2 自定义一个拦截器案例

步骤：

1. 写拦截器类 (看生命周期)

流程：

1.Action实例类创建类

2.执行Action之前

3.执行了请求处理方法：execute

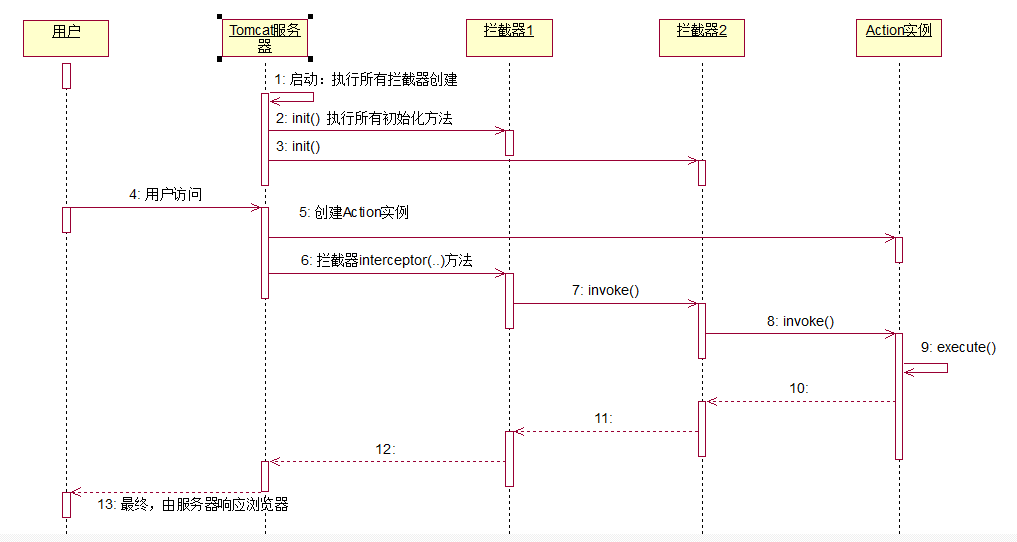
4.拦截器，处理业务-结束

2. 配置

|  |
| --- |
| /\*\*  \* 自定义拦截器  \* **@author** Jie.Yuan  \*/  **public** **class** HelloInterceptor **implements** Interceptor{  // 启动时候执行  **public** HelloInterceptor(){  System.*out*.println("创建了拦截器对象");  }  // 启动时候执行  @Override  **public** **void** init() {  System.*out*.println("执行了拦截器的初始化方法");  }  // 拦截器业务处理方法 （在访问action时候执行？ 在execute之前执行？）  @Override  **public** String intercept(ActionInvocation invocation) **throws** Exception {  System.*out*.println("2. 拦截器，业务处理-开始");    // 调用下一个拦截器或执行Action (相当于chain.doFilter(..)  // 获取的是： execute方法的返回值  String resultFlag = invocation.invoke();    System.*out*.println("4. 拦截器，业务处理-结束");    **return** resultFlag;  }    @Override  **public** **void** destroy() {  System.*out*.println("销毁....");  }  } |
| <?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=*"hello"* extends=*"struts-default"*>  <!-- 【拦截器配置】 -->  <interceptors>  <!-- 配置用户自定义的拦截器 -->  <interceptor  name=*"helloInterceptor"* class=*"cn.itcast.a\_interceptor.HelloInterceptor"*></interceptor>    <!-- 自定义一个栈： 要引用默认栈、自定义的拦截器 -->  <interceptor-stack name=*"helloStack"*>  <!-- 引用默认栈 (一定要放到第一行)-->  <interceptor-ref name=*"defaultStack"*></interceptor-ref>  <!-- 引用自定义拦截器 -->  <interceptor-ref name=*"helloInterceptor"*></interceptor-ref>  </interceptor-stack>    </interceptors>    <!-- 【执行拦截器】 -->  <default-interceptor-ref name=*"helloStack"*></default-interceptor-ref>      <!-- Action配置 -->  <action name=*"hello"* class=*"cn.itcast.a\_interceptor.HelloAction"*>  <result name=*"success"*></result>  </action>      </package>  </struts> |

## 1.2 拦截器执行流程

UML (时序图)



启动：

创建所有拦截器、执行init()

访问:

先创建Action，

再执行拦截器，

最后：拦截器放行，执行execute();

## 1.3 拦截器案例

需求：

登陆后，显示列表！

案例准备：

Struts jar文件

DbUtils组件

数据库连接池/ 驱动包

|  |
| --- |
| --> login.jsp |
| <body>  <form method=*"post"* action=*"*${pageContext.request.contextPath }*/user\_login.action"*>  用户名：<input type=*"text"* name=*"admin.userName"*><br/>  密码：<input type=*"text"* name=*"admin.pwd"*><br/>  <input type=*"submit"* value=*"登陆"*><br/>  </form>  </body> |
| 🡪 UserAction.java |
| **public** **class** UserAction **extends** ActionSupport {  // ---------1. 封装请求数据-----------  **private** Admin admin;  **public** Admin getAdmin() {  **return** admin;  }  **public** **void** setAdmin(Admin admin) {  **this**.admin = admin;  }  // ---------2. 调用的Service-----------  **private** AdminService adminService = **new** AdminService();  // 登陆  **public** String login() {  **try** {  Admin userInfo = adminService.login(admin);  // 判断  **if** (userInfo == **null**){  // 登陆失败  **return** "input";  }  // 登陆成功：数据保存在session中  ActionContext.*getContext*().getSession().put("userInfo", userInfo);  // 登陆成功  **return** "loginSuccess";  } **catch** (Exception e) {  **return** *ERROR*;  }  }  // 列表  **public** String list() {  **try** {  // 查询全部  List<Admin> list = adminService.getAll();  // 保存到request  ActionContext.*getContext*().getContextMap().put("listAdmin", list);  **return** "list";  } **catch** (Exception e) {  **return** *ERROR*;  }  }  **public** String add() {  **return** **null**;  }  } |
| 🡪 list.jsp |
| <body>  <h1>欢迎你，${userInfo.userName }</h1>  <table align=*"center"* border=*"1"*>  <tr>  <td>序号</td>  <td>编号</td>  <td>用户名</td>  <td>密码</td>  </tr>  <%--@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" --%>  <!-- 用struts标签迭代数据 -->  <%@taglib uri=*"/struts-tags"* prefix=*"s"* %>  <s:iterator var=*"admin"* value=*"#request.listAdmin"* status=*"st"*>  <tr>  <td>  <s:property value=*"#st.count"*/>  </td>  <td>  <s:property value=*"#admin.id"*/>  </td>  <td>  <s:property value=*"#admin.userName"*/>  </td>  <td>  <s:property value=*"#admin.pwd"*/>  </td>  </tr>  </s:iterator>    </table>  </body> |
| 🡪 自定义拦截器 |
| **public** **class** UserCheckInterceptor **extends** AbstractInterceptor{  /\*\*  \* 拦截器业务处理方法  \*/  **public** String intercept(ActionInvocation invocation) **throws** Exception {  // 拿到当前执行的方法名：判断，只有当前方法名不是login,就进行验证    // 获取ActionContext对象  ActionContext ac = invocation.getInvocationContext();    // 获取action的代理对象  ActionProxy proxy = invocation.getProxy();  // 获取当前执行的方法名  String methodName = proxy.getMethod();  // 判断  **if** (!"login".equals(methodName)) {  // 先获取当前登陆的用户  Object obj = ac.getSession().get("userInfo");  **if** (obj == **null**) {  // 没有登陆  **return** "input";  } **else** {  // 当前用户有登陆  **return** invocation.invoke();  }  } **else** {  // 说明当前用户正在登陆  **return** invocation.invoke();  }  }  } |
| 🡪 配置拦截器 |
| <?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"*>    <!-- 【拦截器配置】 -->  <interceptors>  <interceptor name=*"loginCheck"* class=*"cn.itcast.interceptor.UserCheckInterceptor"*></interceptor>  <interceptor-stack name=*"myStack"*>  <interceptor-ref name=*"defaultStack"*></interceptor-ref>  <interceptor-ref name=*"loginCheck"*></interceptor-ref>  </interceptor-stack>  </interceptors>  <!-- 【执行拦截器:第一种写法： 当前包下所有的acntion都执行myStack栈】  <default-interceptor-ref name="myStack"></default-interceptor-ref>  -->    <!-- 全局配置 -->  <global-results>  <result name=*"error"*>/error.jsp</result>  </global-results>    <action name=*"user\_\*"* class=*"cn.itcast.action.UserAction"* method=*"{1}"*>    <!--第二种写法： 只是在这一个Action中执行myStack栈  <interceptor-ref name="defaultStackt"></interceptor-ref>  <interceptor-ref name="loginCheck"></interceptor-ref>  -->    <!-- 第三种写法：执行用户栈(与第二种写法一样， 只在当前aciton中执行自定义栈) -->  <interceptor-ref name=*"myStack"*></interceptor-ref>        <!-- 1. 登陆失败 -->  <result name=*"input"*>/login.jsp</result>    <!-- 2. 登陆成功 -->  <result name=*"loginSuccess"* type=*"redirectAction"*>user\_list</result>    <!-- 3. 列表展示 -->  <result name=*"list"*>/WEB-INF/list.jsp</result>    </action>    </package>  </struts> |

# 2. Struts2中的国际化

回顾：Servlet 中国际化：

1. 写资源文件

基础名.properties 【默认的语言环境的配置】

基础名\_语言简称\_国家简称.properties

2. 读取资源文件，再使用

程序：ResourceBundle

Jsp: jstl提供的格式化与国际化标签库。

Struts2中国际化：

1. 写资源文件 (同servlet)

2. 读资源文件

程序：ResourceBundle (同servlet)

JSP:

1）jstl表亲啊 （同servlet）

2）struts标签获取资源文件内容

区别：

Struts2加载资源文件更加简单！通过常量加载即可！再在jsp页面直接使用！

|  |
| --- |
| 🡪1. 写资源文件 |
| Msg.properties 默认的语言环境； 找不到配置就找它 |
| Msg\_en\_US.properties 美国 |
| -🡪2. 加载 |
| <constant name="struts.custom.i18n.resources" value="cn.itcast.config.msg"></constant> |
| 🡪3. 使用: 标签name值直接写配置文件中的key |
| <s:text name="title"></s:text> |

另外一点，

(推荐)加载资源文件通过常量加载

还可以在页面加载, 这样用：

<s:i18n name="cn.itcast.config.msg">

<s:text> 标签必须放到标签体中。

</s:i18n>

# 3. Ognl表达式语言

## 概述

* **OGNL表达式**

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。

* **OGNL优势**

1、支持对象方法调用，如xxx.doSomeSpecial()；

2、支持类静态的方法调用和值访问，表达式的格式:

@[类全名（包括包路径）]@[方法名 |  值名]，例如：

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

或@tutorial.MyConstant@APP\_NAME；

3、支持赋值操作和表达式串联，如price=100, discount=0.8,

calculatePrice()，这个表达式会返回80；

4、访问OGNL上下文（OGNL context）和ActionContext；

5、操作集合对象。

* **总结**

OGNL 有一个上下文（Context）概念，说白了上下文就是一个MAP结构，它实现了 java.utils.Map 的接口。 OgnlContext对象

分析：

* **Struts框架默认就支持Ognl表达式语言。**

**(struts必须引用的包：ognl.jar)**

* **作用**

**页面取值用。**

**El表达式语言，用于页面取值，jsp页面取值的标准。(默认直接可以使用)**

**(应用范围更广。)**

**Ognl表达式语言， struts标签默认支持的表达式语言。**

**必须配置struts标签用，不能离开struts标签直接用。**

## OgnlContext对象(了解)

OgnlContext对象是ognl表达式语言的核心。

源码类：

public class OgnlContext extends Object implements Map{..}

硬编码方式，了解OgnlContext对象：

|  |
| --- |
| // OgnlContext用法  **public** **class** OgnlDemo1 {  /\*\*  \* 1. Ognl表达式语言语言取值，取非根元素的值，必须用#号  \* **@throws** Exception  \*/  @Test  **public** **void** testOgnl() **throws** Exception {  // 创建一个Ognl上下文对象  OgnlContext context = **new** OgnlContext();  // 放入数据  User user = **new** User();  user.setId(100);  user.setName("Jack");  // 【往非根元素放入数据， 取值的时候表达式要用"#"】  context.put("user", user);    // 获取数据(map)  // 先构建一个Ognl表达式, 再解析表达式 可用于jsp取值<s:a value=”#user.name”>  Object ognl = Ognl.*parseExpression*("#user.name");  Object value = Ognl.*getValue*(ognl, context, context.getRoot());    System.*out*.println(value);  }  /\*\*  \* 2. Ognl表达式语言语言取值，取根元素的值，不用带#号  \* **@throws** Exception  \* 下面： 新建一个Address对象类  \*/  @Test  **public** **void** testOgn2() **throws** Exception {  // 创建一个Ognl上下文对象  OgnlContext context = **new** OgnlContext();  // 放入数据  User user = **new** User();  user.setId(100);  user.setName("Jack");  // 【往根元素放入数据】  context.setRoot(user);    // 获取数据(map)  // 先构建一个Ognl表达式, 再解析表达式 对象.值  Object ognl = Ognl.*parseExpression*("address.province");  Object value = Ognl.*getValue*(ognl, context, context.getRoot());    System.*out*.println(value);  }  /\*\*  \* 3.Ognl对 静态方法调用的支持  \* **@throws** Exception  \*/  @Test  **public** **void** testOgn3() **throws** Exception {  // 创建一个Ognl上下文对象  OgnlContext context = **new** OgnlContext();    // Ognl表单式语言，调用类的静态方法  //Object ognl = Ognl.parseExpression("@Math@floor(10.9)");  // 由于Math类在开发中比较常用，所以也可以这样写  Object ognl = Ognl.*parseExpression*("@@floor(10.9)");  Object value = Ognl.*getValue*(ognl, context, context.getRoot());  System.*out*.println(value);  }  } |

## ValueStack对象

ValueStack, 即值栈对象。

值栈对象：

是整个struts数据存储的核心，或者叫中转站。

用户每次访问struts的action，都会创建一个Action对象、值栈对象、ActionContext对象； 然后把Action对象放入值栈中； 最后再把值栈对象放入request中，传入jsp页面。

(key: struts.valueStack)； 开发者只需要通过ActionContext对象就可以访问struts的其他的关键对象。 （ActionContext是给开发者用的，便于学习与使用。）

问题：

OgnlContext与ValueStack对象的关系？

## Struts标签

Struts标签取值，就使用了Ognl表达式语言。