Day03

# 回顾

### Result:

在struts中,Reuslt用于处理Action返回的结果.将我们之前在Servlet中耦合的功能代码.解耦了.将结果处理的代码封装到了Result中.

Struts2已经预置了一些处理类.

Dispatcher 转发

Redirect 重定向

redirectAction 重定向到一个Action(与redirect配合)

plaintext 纯文本

stream 文件下载

### 另外一种注入ServletApi的方式.

让Action去实现

ServletRequestAware

ServletResponseAware

SessionAware....

这些接口实现后,struts2 会在启动的拦截器中注入你想要的对应对象.

servletConfig拦截器.

### 自动封装表单参数

1. 提交的键与Action中的属性名称一模一样
2. 属性提供get/set方法

### 自动类型转换

表单提交的参数,类型全都是String.而我们在Action中接收这些参数,不使用String类型也可以接受,strus2可以自动帮我们转换类型.

8个基本数据类型以及对应的包装类

date类型 => yy-MM-dd

### 自定义转换器

实现TypeConverter接口(太麻烦,不用)

继承DefaultTypeConverter类

convertValue(Object object,Class toType)

参数1:待转换的对象

参数2: 需要转换成的类型

配置类型转换器:

局部配置: Action所在包下.Action简单类名-conversion.properties

birthday=转换器完整类名

全局配置: src下xwork- conversion.properties

转换到的类型=转换器完整类名

### 表单参数校验

校验流程:

Validation拦截器负责判断Action是否实现Validateable接口.如果实现就调用validate方法.

workflow拦截器判断Action是否实现ValidationAware接口. 如果实现就会判断Action中是否包含错误信息.如果包含错误信息, 停止拦截器递归调用,返回结果到名为input的结果页面中.

编程式校验

实现Validateable接口.接口中只有一个方法validate方法.我们可以在该方法中添加错误信息.

添加错误信息需要实现ValidationAware接口.该接口中有一些判断是否含有错误消息.获得错误消息.添加错误消息的方法.

配置式校验

# 类型转换补充

|  |
| --- |
|  |

　　我们发现Struts2打印的类型转换错误信息是英文的，这说明我们需要自定义错误信息。自定义错误信息需要在Action所在目录下创建*ActionName*.properties文件（与Action同名的properties文件），然后在该文件中给出：invalid.fieldvalue.*属性名*=错误信息，其中invalid.fieldvalue是固定的。例如：invalid.fieldvalue.person=无法将请求参数转换成Person类型！

PersonAction.proeprties

|  |
| --- |
| invalid.fieldvalue.person=无法把表单参数转换成Person类型 |

# 输入校验

###### 1　什么是输入校验

在Action封装了请求参数后，还需要对其进行校验。例如name不能为空，age只能在18~60之间等等！我们一定要搞清楚，输入校验是在类型转换成功之后，才可能执行的。

校验分类：

* JavaScript客户端校验（改善用户体验）；
* 服务器端校验（保证安全性），即使用Struts2输入校验。

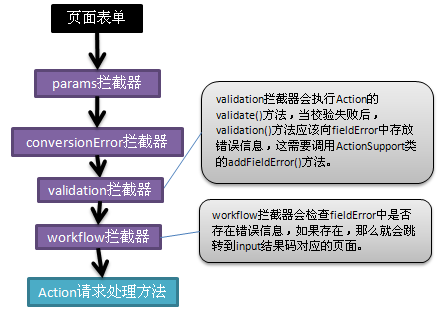
###### 2　Struts2输入校验分类

Struts2输入校验分为两种：

* 编程式校验；
* 配置校验：
* XML配置校验（了解）；

###### 3　编程式的输入校验

覆盖ActionSupport类的validate()方法，在这个方法中完成对参数的校验。validate()方法会在参数封装之后，在execute()方法之前执行。如果validate()方法中向fieldError中添加了错误信息，那么就不会再执行execute()方法，而是跳转到input结果码对应的页面。



|  |
| --- |
| <s:fielderror />  <form action=*"*<c:url value=*'/Demo1Action.action'*/>*"* >  用户名：<input type=*"text"* name=*"username"*/><br/>  密　码：<input type=*"password"* name=*"password"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
| **public** **class** Demo1Action **extends** ActionSupport {  **private** String username;  **private** String password;  @Override  **public** **void** validate() {  **if**(username == **null** || username.trim().length() == 0) {  **this**.addFieldError("username", "用户名不能为空");  }  **if**(password == **null** || password.trim().length() == 0) {  **this**.addFieldError("password", "密码不能为空");  }  }  **public** String execute() {  System.*out*.println(username + ", " + password);  **return** *NONE*;  }  } |

在validate方法中

使用addFieldError(fieldName, errorMessage)方法存入字段的错误信息.

使用addActionError(anErrorMessage)方法存入action的错误信息

使用addActionMessage(aMessage)方法存入action提示信息

在页面

使用<s:fielderror fieldName="xxx" />取出字段的错误信息

使用<s:actionerror/>取出action的错误信息

使用<s:actionmessage/>取出action提示信息

当然如果你的页面使用了struts2提供的表单标签.表单标签会自动显示字段的错误信息

###### 4　校验进阶

##### 跳过指定的校验方法:

如果想跳过某个Action方法的校验,在不需要校验的Action方法上加入@SkipValidation

我们都知道，一个Action中可以存在多个请求处理方法，不同的请求处理方法应该有不同的校验逻辑，所以我们应该为每个请求处理方法提供自己独有的校验方法。而validate()方法是所有请求处理方法公共的校验方法。

#### 也可以指定校验某个方法:

* public的，没有返回值，没有参数（public void xxx()）；
* 方法名称前缀为validate（public void validateXxx()）；
* 方法名后缀必须与请求处理方法名相同，例如请求处理方法为login()，那么它的私有校验方法为：public validateLogin()。

注意，私有校验方法会在公共校验方法（validate()）之前被调用。如果你要为execute()提供私有校验方法，那么这个方法名为validateExecute()。

|  |
| --- |
| **public** **class** Demo2Action **extends** ActionSupport {  **public** **void** validateLogin() {  System.*out*.println("validateLogin()...");  }  **public** **void** validateRegist() {  System.*out*.println("validateRegist()...");  }  **public** **void** validate() {  System.*out*.println("validate()...");  }  **public** String login() {  System.*out*.println("login()");  **return** *NONE*;  }  **public** String regist() {  System.*out*.println("regist()");  **return** *NONE*;  }  } |
|  |

###### 5　XML配置方式校验（了解）

使用XML配置方式是先把常用的校验规则写好，然后在XML配置中指定要使用的校验规则。当然Struts2已经帮我们写好了很多的校验规则。我们只需要指在XML文档中配置当前的请求处理方法需要哪些校验规则。

5.1　XML配置方式校验要求

要使用XML配置方式校验，你的Action类必须实现Validateable接口。ActionSupport类已经实现了Validateable接口，所以我们通常是直接继承ActionSupport类。

为属性提供getXXX()和setXXX()方法！代码校验是在Action本类中来完成校验，这说明我们可以直接使用本类的private属性，但如果使用XML配置方式校验，这需要使用校验框架的代码来完成校验工作，那么校验框架需要调用Action的getXXX()方法来获取被校验的属性，所以一定要为被校验的属性提供getXXX()方法。

5.2　创建校验文件

1. 校验文件的命名必须为：*ActionName*-validation.xml。例如LoginAction的校验文件命名为：LoginAction-validation.xml。

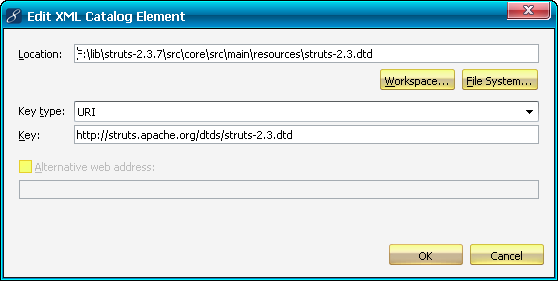
2. 校验文件的路径：必须与Action在同包下。

3. 校验文件的DTD：在xwork-core-x.x.x.jar中找到xwork-validator-x.x.x.dtd，打开它，内部会有一段DTD，我们把它copy过来，放到我们的校验文件中。

xwork-validator-1.0.3.dtd

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!--  XWork Validators DTD.  Used the following DOCTYPE.  <!DOCTYPE validators PUBLIC  "-//Apache Struts//XWork Validator 1.0.3//EN"  "http://struts.apache.org/dtds/xwork-validator-1.0.3.dtd">  -->  <!ELEMENT validators (field|validator)+>  <!ELEMENT field (field-validator+)>  <!ATTLIST field  name CDATA #REQUIRED  >  <!ELEMENT field-validator (param\*, message)>  <!ATTLIST field-validator  type CDATA #REQUIRED  short-circuit (true|false) *"false"*  >  <!ELEMENT validator (param\*, message)>  <!ATTLIST validator  type CDATA #REQUIRED  short-circuit (true|false) *"false"*  >  <!ELEMENT param (#PCDATA)>  <!ATTLIST param  name CDATA #REQUIRED  >  <!ELEMENT message (#PCDATA|param)\*>  <!ATTLIST message  key CDATA #IMPLIED  > |

4. 为了在MyEclipse中对XML有提示功能，那么还需要让MyEclipse导入DTD文件的位置。



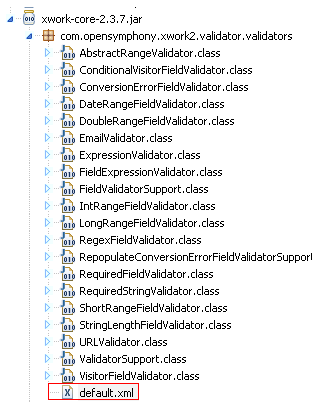
5.3　编写校验文件

校验文件的元素结果如下：

|  |
| --- |
| <validators>  <field name=*""*>  <field-validator type=*""*>  <param name=*""*></param>  <message></message>  </field-validator>  </field>  </validators> |

<field>的name属性指定要校验的属性，例如<feld name=”username”>，表示要校验的属性是username属性。

<field-validator>的type属性指定校验规则，校验规则由Struts2提供，Struts2提供的所有校验规则在：



打印default.xml文件，内部如下：

defualt.xml

|  |
| --- |
| <?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> |

上面文件中每个<validator>元素都是一个校验规则，校验规则对应一些已经写好的方法，他们有校验属性非空的规则，有校验字符串属性长度的规则，有校验int属性范围的规则等等。通常我们不需要自己来编写校验规范，因为上面的校验规则已经足够了。

每个规则都有自己的名字，校验文件中<field-validator>的type就是用来指定校验规则的名称。例如下面的代码是对username属性的非空校验：

|  |
| --- |
| <validators>  <field name=*"username"*>  <field-validator type=*"requiredString"*>  ……  </field-validator>  </field>  </validators> |

其中type=”requiredString”是校验规则的名称，它必须对应defualt.xml文件中<validator>元素的name属性值。requiredString校验规则是校验字符串属性是否长度为0，如果长度为0，它会向fieldError中添加错误信息。

<message>元素指定的是错误信息，例如：

|  |
| --- |
| <validators>  <field name=*"username"*>  <field-validator type=*"requiredstring"*>  <message>用户名不能为空</message>  </field-validator>  </field>  </validators> |

每个校验规则还都有自己的参数，如果想知道每个校验规则有什么参数，那么最好的方法是去查看校验规则的源代码。

例如requiredstring校验规则有一个trim参数，它是boolean类型的参数，当trim为true时，requiredString校验器会先调用属性的trim()方法（去掉前后空白），然后再校验长度是否为0。trim参数的默认值就是true。

|  |
| --- |
| **public** **class** RequiredStringValidator **extends** FieldValidatorSupport {  **private** **boolean** doTrim = **true**;  **public** **void** setTrim(**boolean** trim) {  doTrim = trim;  }  **public** **boolean** getTrim() {  **return** doTrim;  }  **public** **void** validate(Object object) **throws** ValidationException {  String fieldName = getFieldName();  Object value = **this**.getFieldValue(fieldName, object);  **if** (!(value **instanceof** String)) {  addFieldError(fieldName, object);  } **else** {  String s = (String) value;  **if** (doTrim) {  s = s.trim();  }  **if** (s.length() == 0) {  addFieldError(fieldName, object);  }  }  }  } |

　　下面是完整的LoginActoin的校验文件：

LoginAction-validation.xml

|  |
| --- |
| <validators>  <field name=*"username"*>  <field-validator type=*"requiredstring"*>  <!-- 调用校验器的setTrim()方法传递参数false -->  <param name=*"trim"*>false</param>  <message>用户名不能为空</message>  </field-validator>  <field-validator type=*"stringlength"*>  <!-- 调用校验器的setMaxLength()方法传递参数16 -->  <param name=*"maxLength"*>16</param>  <!-- 调用校验器的setMinLength()方法传递参数3 -->  <param name=*"minLength"*>3</param>  <message>用户名长度必须在3~16之间</message>  </field-validator>  </field>  <field name=*"password"*>  <field-validator type=*"requiredstring"*>  <param name=*"trim"*>false</param>  <message>密码不能为空</message>  </field-validator>  </field>  </validators> |

5.4　校验规则介绍

* required：当属性为null时校验失败；
* requiredstring：当字符串属性长度为0时校验失败：
* 参数trim：默认值为true，表示去除前后空白后再校验长度。
* stringlength：当字符串长度不在指定范围内时校验失败：
* minLength：指定字符串的最小长度；
* maxLength：指定字符串的最大长度。
* regex：属性不能匹配正则表达式时校验失败：
* expression：指定正则表达式；
* caseSensitive：默认值为true，表示不忽略大小写。
* int：当int属性不在指定范围内校验失败：
* min：最小值；
* max：最大值。
* double：当double属性不在指定范围内校验失败：
* min：最小值；
* max：最大值。
* fieldexpression：属性必须是OGNL表达式：
* expression：用来校验的ONGL表达式，例如pass == repass，其中pass和repass是两个属性名称，当这两个属性的值相等时校验通过，否则失败。
* email：属性必须是合法的邮件地址；
* url：属性必须是合法的网址；
* date：属性必须是合法的日期格式。

## 拦截器

###### 1　理解拦截器

来自AOP（面向切面编程）思想。它看起来与JavaWeb中的Filter极其相似。我们已经知道请求会先过一系列拦截器，最终到达Action，或者中途中断。也就是说，每个拦截器都有中断请求的能力。

因为现在还不是讲AOP的时候，所以我们现在只要把拦截器理解为JavaWeb中Filter即可。你回忆一下Filter与Servlet的关系，那么Interceptor与Action就是相同的关系。

###### 2　Struts2中的拦截器

　　在Struts2中定义了很多拦截器，你可以去struts-default.xml文件中查看。

###### 3　自定义拦截器

我们也可以自定义拦截器，Struts2要求所有拦截器必须实现Interceptor接口。

Interceptor.java

|  |
| --- |
| **public** **interface** Interceptor **extends** Serializable {  **void** destroy();  **void** init();  String intercept(ActionInvocation invocation) **throws** Exception;  } |

Struts2还提供了一个Interceptor接口的实现类：AbstractInterceptor，通常我们自定义拦截器都是通过继承AbstractInterceptor类，而不是实现Interceptor接口。

AbstractInterceptor.java

|  |
| --- |
| **public** **abstract** **class** AbstractInterceptor **implements** Interceptor {  **public** **void** init() {}  **public** **void** destroy() {}  **public** **abstract** String intercept(ActionInvocation invocation) **throws** Exception;  } |

继承AbstractInterceptor类时，不需要“被迫”实现init()和destroy()方法，而只需要关注intercept()方法即可。

下面我们自定义一个拦截器

MyInterceptor

|  |
| --- |
| **public** **class** MyInterceptor **extends** AbstractInterceptor {  @Override  **public** String intercept(ActionInvocation invaction) **throws** Exception {  System.*out*.println("MyInterceptor...");  **return** invaction.invoke();  }  } |

###### 4　注册拦截器

注册拦截器一共分为两步：

* 在<package>中声明拦截器；
* 在<action>中引用拦截器。

|  |
| --- |
| <package name=*"s8"* namespace=*"/"* extends=*"struts-default"*>  <interceptors>  <interceptor name=*"MyInterceptor"* class=*"cn.itcast.interceptor.MyInterceptor"* />  </interceptors>  <action name=*"LoginAction"*>  <result>/index.jsp</result>  <result name=*"input"*>/login.jsp</result>  <interceptor-ref name=*"MyInterceptor"* />  </action>  </package> |

上面的代码虽然可以执行MyInterceptor了，但因为Struts2有这么一种机制，一旦为Action指定了拦截器，那么就不会再为这个Action执行默认拦截器了，即defaultStack这个拦截器栈中的拦截器都不会执行，也就是说，这个Action没有输入校验、没有参数注入、没有国际化、没有…，这是不行的，所以我们需要在这个<action>元素中再引用defaultStack拦截器栈。

|  |
| --- |
| <package name=*"s8"* namespace=*"/"* extends=*"struts-default"*>  <interceptors>  <interceptor name=*"MyInterceptor"* class=*"cn.itcast.interceptor.MyInterceptor"* />  </interceptors>  <action name=*"LoginAction"*>  <result>/index.jsp</result>  <result name=*"input"*>/login.jsp</result>  <interceptor-ref name=*"defaultStack"* />  <interceptor-ref name=*"MyInterceptor"* />  </action>  </package> |

在<aciton>元素中引用拦截器的顺序决定了拦截器的执行顺序，上例中会先执行defaultStack中的所有拦截器，再执行MyInterceptor拦截器。

上面的方式虽然可以注册拦截器，但比较麻烦。因为如果当前包中所有<action>都需要执行MyInterceptor拦截器，那么就需要在每个<action>元素中引入拦截器。其实还有另一种方式，就是为当前包指定默认拦截器栈！

我们都知道，因为我们的包继承了struts-default包，所以默认的拦截器栈是defaultStack，但没有为<action>元素指定拦截器时，那么就会执行defaultStack拦截器栈。我们可以在<package>中声明一个拦截器栈，然后在去替换默认拦截器栈即可。

|  |
| --- |
| <package name=*"s8"* namespace=*"/"* extends=*"struts-default"*>  <interceptors>  <interceptor name=*"MyInterceptor"* class=*"cn.itcast.interceptor.MyInterceptor"* />  <interceptor-stack name=*"myStack"*>  <interceptor-ref name=*"defaultStack"* />  <interceptor-ref name=*"MyInterceptor"* />  </interceptor-stack>  </interceptors>  <default-interceptor-ref name=*"myStack"* />  <action name=*"LoginAction"*>  <result>/index.jsp</result>  <result name=*"input"*>/login.jsp</result>  </action>  </package> |

## 5.拦截器进阶(MethodFilterInterceptor)

## OGNL表达式

###### 什么是OGNL语言

OGNL的全称为Object Graphic Navigation Language（对象图导航语言）。它是Struts2的默认表达式语言！

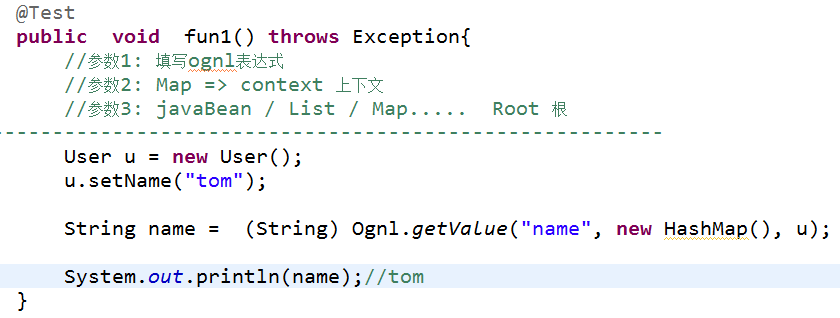
使用OGNL需要导入OGNL的Jar包：ognl-3.0.5.jar

OGNL的功能介绍：

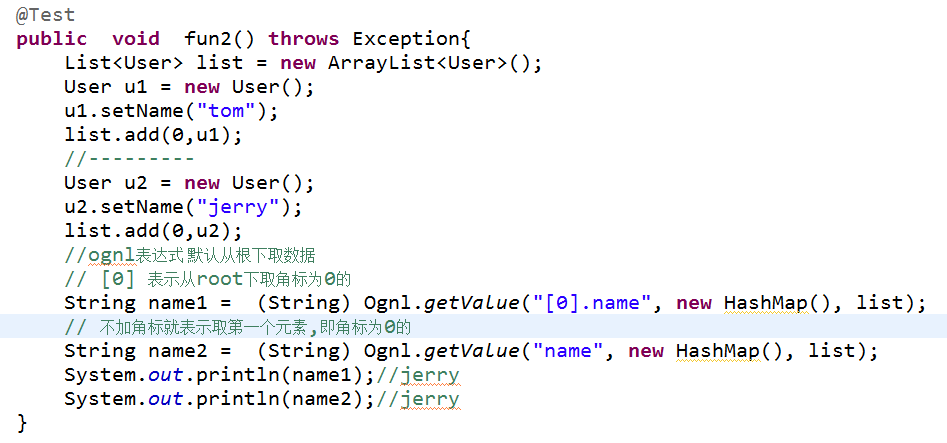
* EL一样的JavaBean导航；
* 调用对象方法；
* 调用类的静态方法；
* 索引数组元素；
* 操作集合；

###### OGNL功能演示

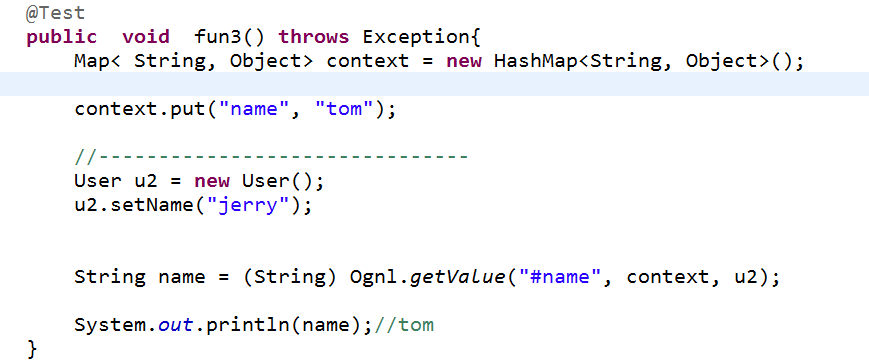
1. **取值**
   1. **根中取**
      1. **根是javabean**



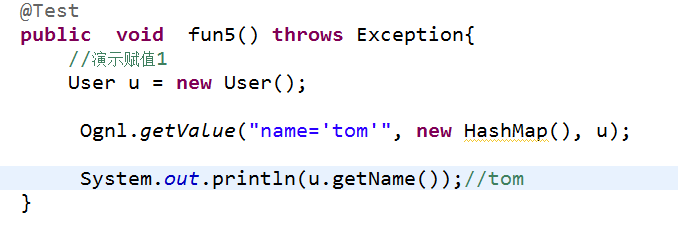
* + 1. **根是list([n]语法)**



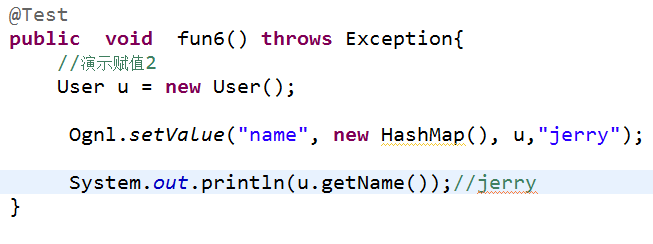
* 1. **Map中取**



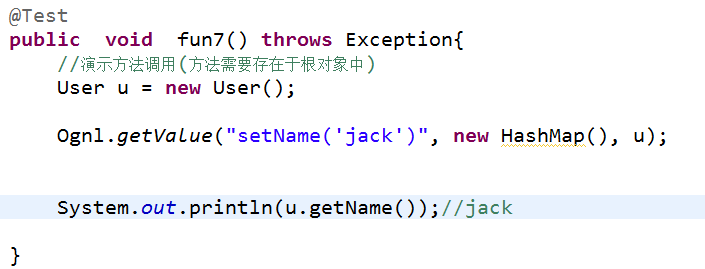
1. **赋值**
   1. **表达式赋值**



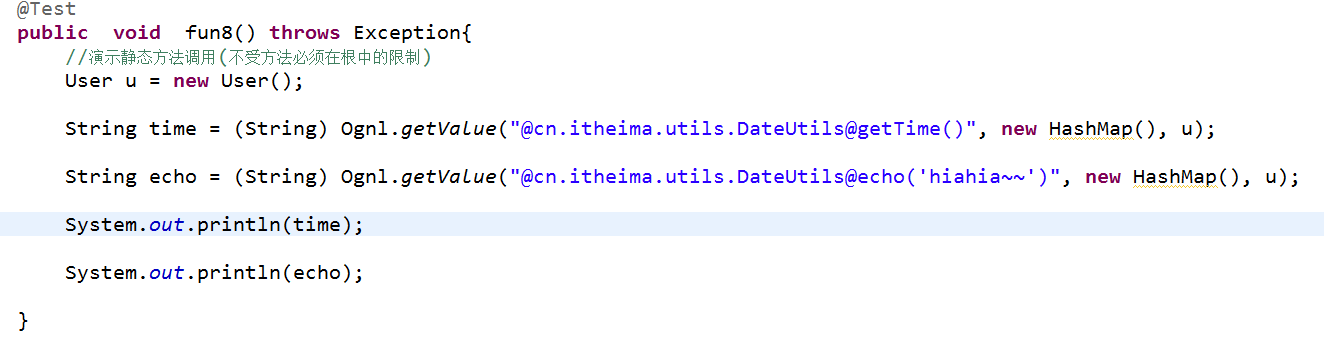
* 1. **SetValue方法赋值**



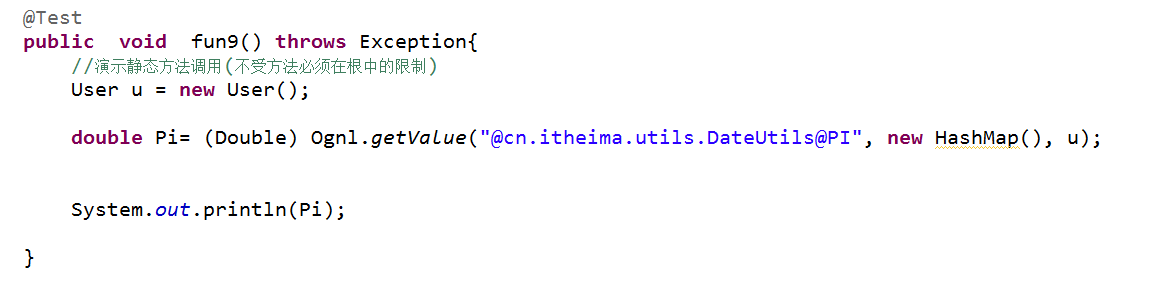
1. **调用方法**



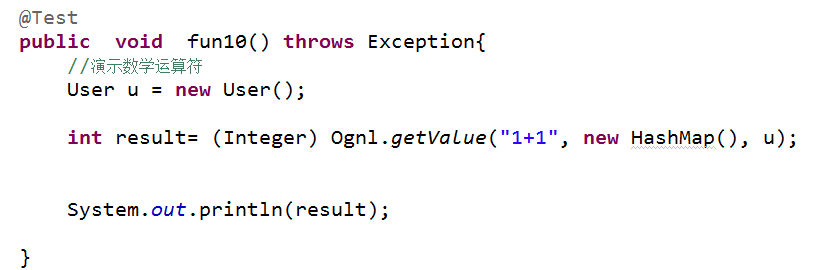
1. **调用静态方法**



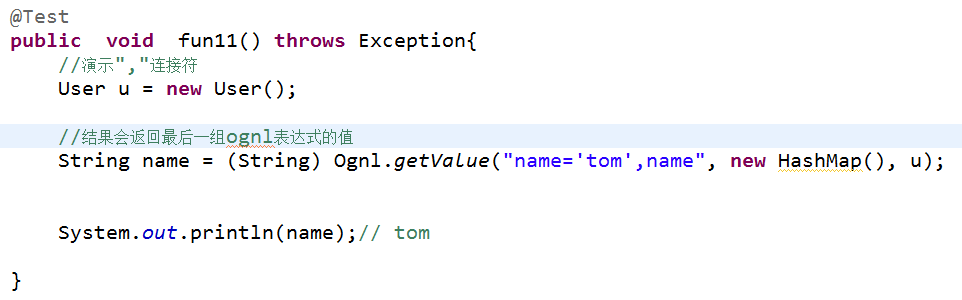
1. **访问静态变量**



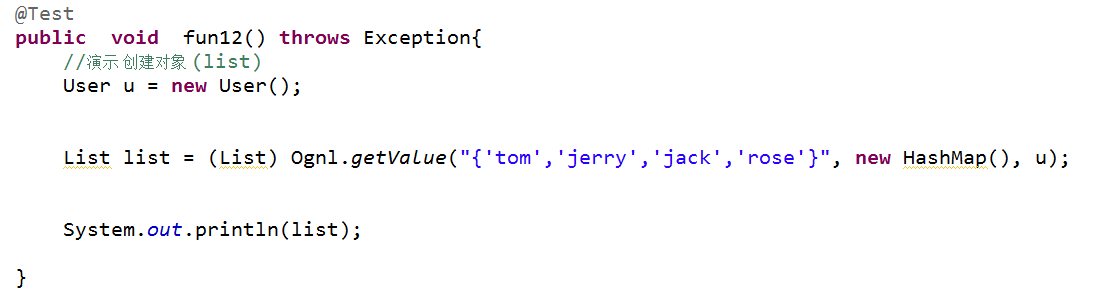
1. **数学运算符**



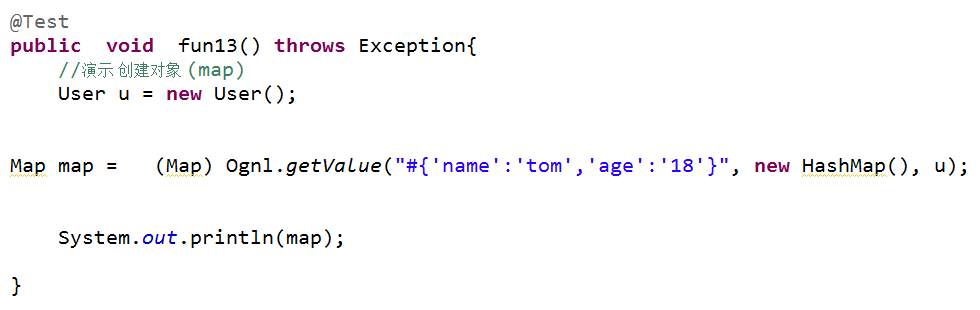
1. **“,”号连接**



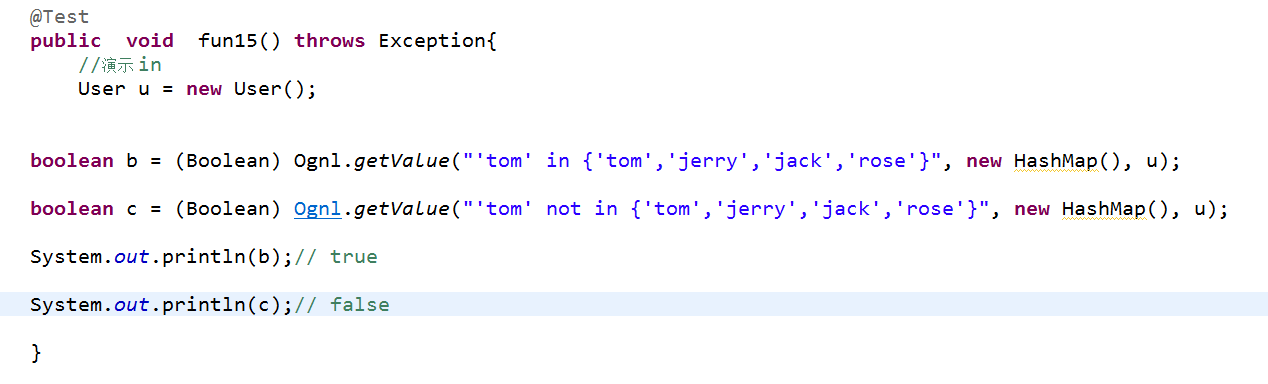
1. **创建list**



1. **创建map**



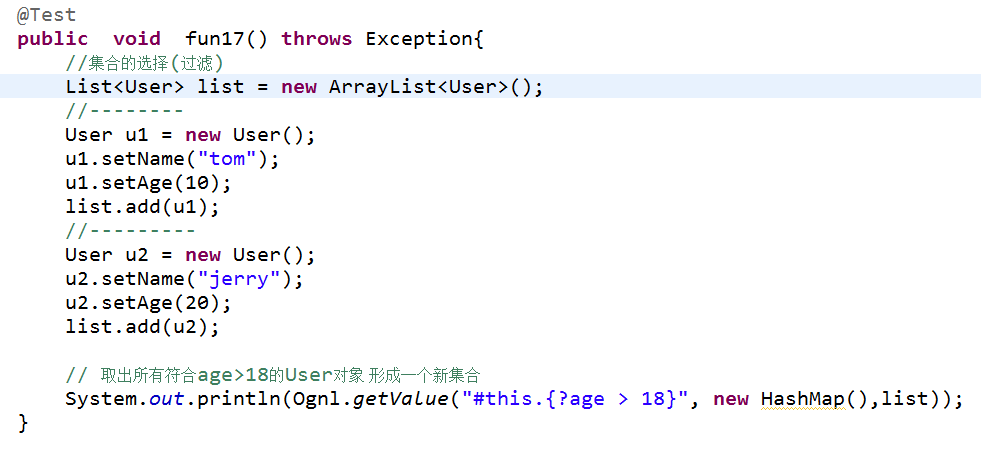
1. **In与not in运算符**



1. **投影(了解)**



1. **过滤(了解)**



## OGNL与Struts2的结合

###### Struts2中OGNL操作的上下文和根对象

* 上下文对象是ActionContext；
* 根对象是ValueStack，它是一个栈结构，提供了压栈和弹栈等方法，通常栈顶元素是当前方法的Action对象；
* 每个请求都有自己的ActionContext，每个ActionContext中都有自己的ValueStack；

Struts2中的上下文对象就是ActionContext对象，而根对象是ValueStack对象。但Struts2中根对象ValueStack很特殊，它不是一个对象，而是一组对象（底层是ArrayList的栈结构）。你也可以理解为Struts2的根对象不是一个对象，而是一堆对象。

可以通过ActionContext来获取ValueStack：ActionContext.getContext().getValueStack()，也可以从request中获取ValueStack：request.getAttribute(“struts.valueStack”)。

ActionContext内部的数据：

* 请求参数和域属性：
* parameters：Map类型，对应HttpServletRequest中的请求参数；
* request：Map类型，对应HttpServletRequest中的属性；
* session：Map类型，对应HttpSession中的属性；
* application：Map类型，对应ServletContext中的属性；
* attr：Map类型，对应request、session、application三个域，即全域查询的Map；
* 域对象：
* com.opensymphony.xwork2.dispatcher.HttpServletRequest：HttpServletRequest对象
* com.opensymphony.xwork2.dispatcher.HttpServletResponse：HttpServletResposne对象
* com.opensymphony.xwork2.dispatcher.ServletContext：ServletContext对象；
* 其他信息：
* com.opensymphony.xwork2.ActionContext.name：<action>的name属性值；
* action：当前Action对象；
* com.opensymphony.xwork2.util.ValueStack.ValueStack：ValueStack对象；
* com.opensymphony.xwork2.ActionContext.locale：当前的Locale。

ValueStack内部的数据：

* Action对象；
* 模型对象（ModelDriven）。

　　不同的请求对应不同的ValueStack，通过在请求一个Action时，都会把当前Action的引用放到ValueStack中。一般情况下，当前Action就在ValueStack的栈顶！

## 结合演示:

1. **<s:debug>标签介绍**
2. **将表单的参数赋值到对象中(ModelDriver方式)**

让Action实现ModelDriver接口.将希望赋值到的对象返回.那么表单在提交时参数将会优先赋值到该对象上.

原理: struts2的表单赋值实际上是依赖ognl,提交参数的键相当于ognl表达式

1. **将表单的参数赋值到对象中(表达式方式)**
2. **文件下载中文件名乱码问题解决(配置文件中使用OGNL)**
3. **Struts2标签中OGNL表达式(<s:properties>)**