Drools开发环境构建

1 开发前提条件：

jdk 1.5+

ide eclipse

插件 org.drools.updatesite

相关软件包

Drools规则引擎 drools-distribution-5.5.0.Final.zip，下载地址：<http://download.jboss.org/drools/release/5.5.0.Final/drools-distribution-5.5.0.Final.zip>

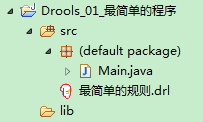
Drools jBPM5整合包 droolsjbpm-tools-distribution-5.5.0.Final.zip（下载地址：<http://download.jboss.org/drools/release/5.5.0.Final/droolsjbpm-tools-distribution-5.5.0.Final.zip>

），解压后这个目录下包含了一个org.drools.updatesite文件夹，里面就是Drools和jBPM5的插件 （安装eclipse插件会的哈）

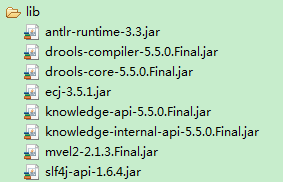
2 搭建一个最简单的Drools应用

1. 用eclipse建一个最简单的Java Project

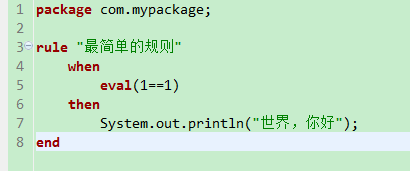
项目大致的结构



需要引入的jar包（必备的）



1. 上面定义了一个drl文件，这个就是Drools的规则文件，文件的图标显示为一个“智慧的脑袋”，说明Drools插件起作用了。用Drools插件提供的规则文件编辑器打开这个文件， 会看到内容：



这里声明了一条最简单的规则，每个规则都有这样的基本结构（其他的声明配置后面再介绍）。package表示规则所属的包名，类似于Java，详细内容见后面。when语句块中是规则条件（专业的叫法，模式匹配），then语句块就是满足该条件需要执行的结果（更准确说是规则的结论）。

1. 调用Drools相关API，装载、解析、执行上述流程定义。Main.java

|  |
| --- |
| **import** java.util.Collection;  **import** org.drools.KnowledgeBase;  **import** org.drools.KnowledgeBaseFactory;  **import** org.drools.builder.KnowledgeBuilder;  **import** org.drools.builder.KnowledgeBuilderFactory;  **import** org.drools.builder.ResourceType;  **import** org.drools.definition.KnowledgePackage;  **import** org.drools.io.ResourceFactory;  **import** org.drools.runtime.StatefulKnowledgeSession;  **public** **class** Main {  **public** **static** **void** main(String[] args) {  // 规则构建器对象，用于解析各种规则相关的资源  KnowledgeBuilder builder = KnowledgeBuilderFactory.*newKnowledgeBuilder*();  // 装载并解析规则文件，规则的资源文件很多，这里我们用的是最常用的规则定义文件（后缀drl）  builder.add(ResourceFactory.*newClassPathResource*("最简单的规则.drl"), ResourceType.*DRL*);  // 如果规则解析出错，直接用异常抛出规则解析的错误消息  **if** (builder.hasErrors()) {  **throw** **new** RuntimeException(builder.getErrors().toString());  }  // 创建一个空的规则库对象  KnowledgeBase knowledgeBase = KnowledgeBaseFactory.*newKnowledgeBase*();  // 获取规则构建器解析规则的产物（规则包）  Collection<KnowledgePackage> pkgs = builder.getKnowledgePackages();  // 把规则包填充到规则库中，这样这个规则库就准备好了，它是一个重量级的对象，构建过程蛮费时的。  knowledgeBase.addKnowledgePackages(pkgs);  // 创建用于和规则引擎之间进行交互的一次回话，这个回话对象很像Java web中servlet里的session的概念  // session的创建是轻量级的  StatefulKnowledgeSession session = knowledgeBase.newStatefulKnowledgeSession();    **try** {  // 执行规则  session.fireAllRules();  } **finally** {  // 最后释放资源  session.dispose();  }  }  } |

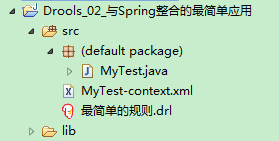
最后的输出结果：

|  |
| --- |
| 世界，你好 |

**Drools整合Spring运行一个最简单的应用程序**

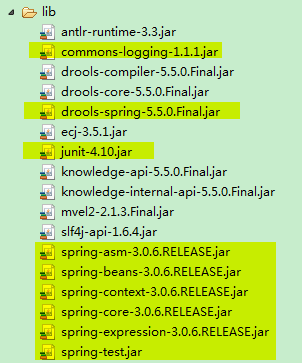
最开始，我们创建的第一个Drools应用程序完全是起main线程跑的。现在的Java项目中普遍用Spring来管理Bean。Drools同样可以和Spring进行整合。下面就来改写第一个例子（把上面的Java项目复制一份，来完成这个应用）。

这个应用的目录结构也很简单：



1. 补充jar包

因为需要和Spring整合，并且跑单元测试来执行规则，在原来的jar包基础上，我们需要引入额外的必备jar包（荧光色已标出）：



1. Spring配置文件

在Spring配置文件中需要定义跟Drools相关的bean，这里需要在xml配置的头部引入drools的命名空间。然后在配置drools的kbase（规则库bean）时，我们指定了规则库需要装载的规则资源。在Spring IoC容器加载这个配置时，会在内部完成类似于我们在第一个项目中的这部分操作，这样我们就获得了一个“有货”的规则库：

|  |
| --- |
| // 规则构建器对象，用于解析各种规则相关的资源  KnowledgeBuilder builder = KnowledgeBuilderFactory.*newKnowledgeBuilder*();  // 装载并解析规则文件，规则的资源文件很多，这里我们用的是最常用的规则定义文件（后缀drl）  builder.add(ResourceFactory.*newClassPathResource*("最简单的规则.drl"), ResourceType.*DRL*);  // 如果规则解析出错，直接用异常抛出规则解析的错误消息  **if** (builder.hasErrors()) {  **throw** **new** RuntimeException(builder.getErrors().toString());  }  // 创建一个空的规则库对象  KnowledgeBase knowledgeBase = KnowledgeBaseFactory.*newKnowledgeBase*();  // 获取规则构建器解析规则的产物（规则包）  Collection<KnowledgePackage> pkgs = builder.getKnowledgePackages();  // 把规则包填充到规则库中，这样这个规则库就准备好了，它是一个重量级的对象，构建过程蛮费时的。  knowledgeBase.addKnowledgePackages(pkgs); |

上面的这部分内容，我们在Spring配置中就如此简单了：MyTest-context.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:drools=*"http://drools.org/schema/drools-spring"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://drools.org/schema/drools-spring http://drools.org/schema/drools-spring-1.7.0.xsd"*>  <drools:kbase id=*"kbase"*>  <drools:resources>  <drools:resource type=*"DRL"* source=*"classpath:最简单的规则.drl"* />  </drools:resources>  </drools:kbase>  </beans> |

1. 最后是单元测试类（关于Spring和junit单元测试这块的知识点就省略了，可以参考其他资料）。MyTest.java

|  |
| --- |
| **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession();  **try** {  // 执行规则  session.fireAllRules();  } **finally** {  // 最后释放资源  session.dispose();  }  }    } |

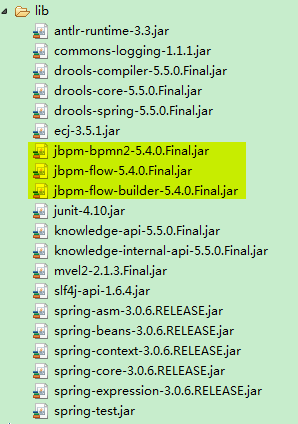
最后的执行结果同上一个项目。

**一个最简单的规则流（Rule Flow）应用**

在实际项目中使用规则引擎，一般会用它来做规则校验、规则计算，甚至是其他的一些可以用规则解决的地方，像新老系统之间的数据结构的转换，在数据量不是很大的情况下，也可以使用规则来做。这边说到的用规则引擎来实现校验、计算、数据转换，它们也是不无联系的，这些规则模块都是业务流程中的一个环节，有执行的先后顺序，还可能包括一些信息传递和共享。现在我们由规则引出了流程的概念。Drools规则引擎当然支持流程的形式（和jBoss旗下的另一个工作流产品jBPM5可以说是姊妹的关系，它们在API内部就做到了相互依赖和补充）。从Drools5.1版本开始，支持bpmn2的业务流程定义规范了。下面我们来看一个最简单的规则流应用。

我们依然在上一个与Spring整合的项目的基础上（复制一份来做），实现一个规则流应用。

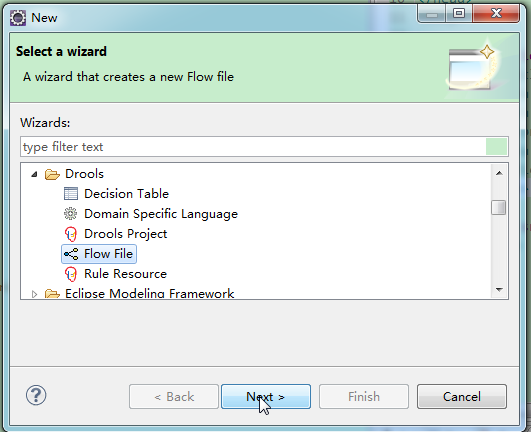
注意需要在原来基础上额外引入jBPM5相关的jar包用于支持流程：



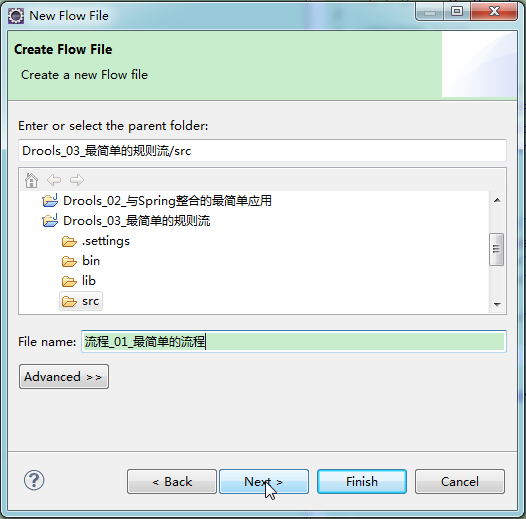
1. 定义流程文件

这里我们用eclipse的bpmn2流程编辑器创建并编辑一个规则流文件（基于bpmn2规范，以.bpmn作为拓展名）。步骤如下：

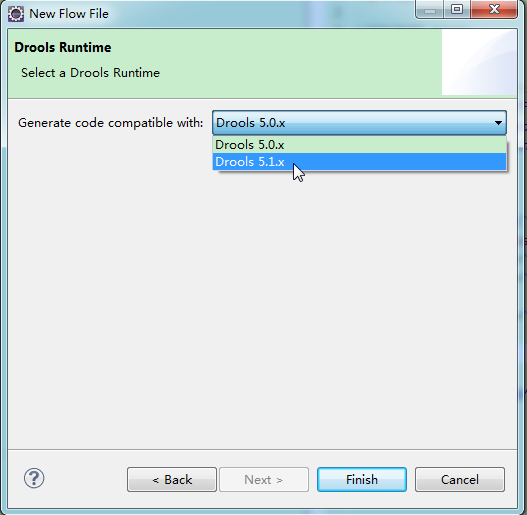
1. 在源代码包（src）下右键新建一个流程文件



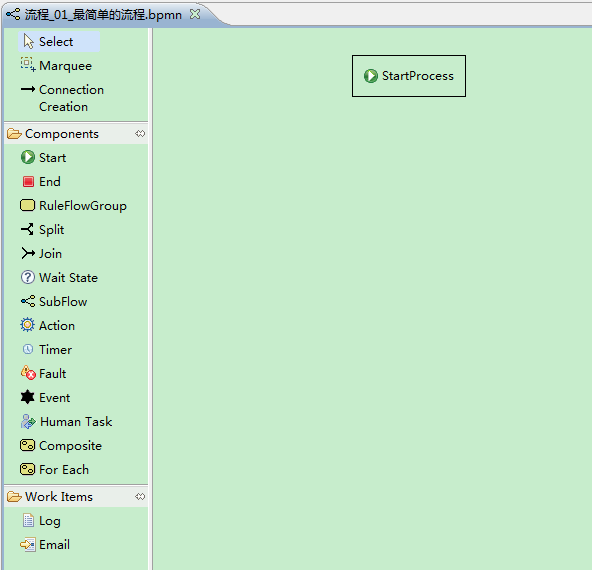
1. 输入流程文件的名称，点击下一步



1. 选择Drools5.1版本开始支持的bpmn流程定义规范，点击finish



1. 然后扑入我们眼帘的是我们再熟悉不过的流程编辑器窗口了，这着实让人兴奋啊（如果你之前也用过jBPM3或jBPM4的话）。以后我们的jBPM5流程定义也就靠这个编辑器了呢。

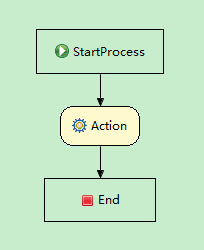


小贴士：这个编辑器窗口我还是喜欢这样的皮肤，有jBPM4控的童鞋可以这样设置来换肤哦。

Window→Preferences→Drools→在窗口右边区域设置

看到上面窗口左侧的“调色板”里面这么多货，真让人眼花缭乱。先不急嘛，以后我们都会一个个介绍到并实际使用的。

现在我们就是艺术家，就着左边的“调色板”，尽情的画吧（拖拖拽拽就行了，记住一定要连线哦）。最后我们“画”了这么个玩意儿：



中间的节点是用于执行脚本任务的，支持Java和MVEL表达式语言（相关语法我们后面会详细介绍），这个脚本任务节点是自动执行的。这里我们就只是配置这个节点在执行过后，在控制台上输出一句话。介绍下节点怎么玩儿的吧。

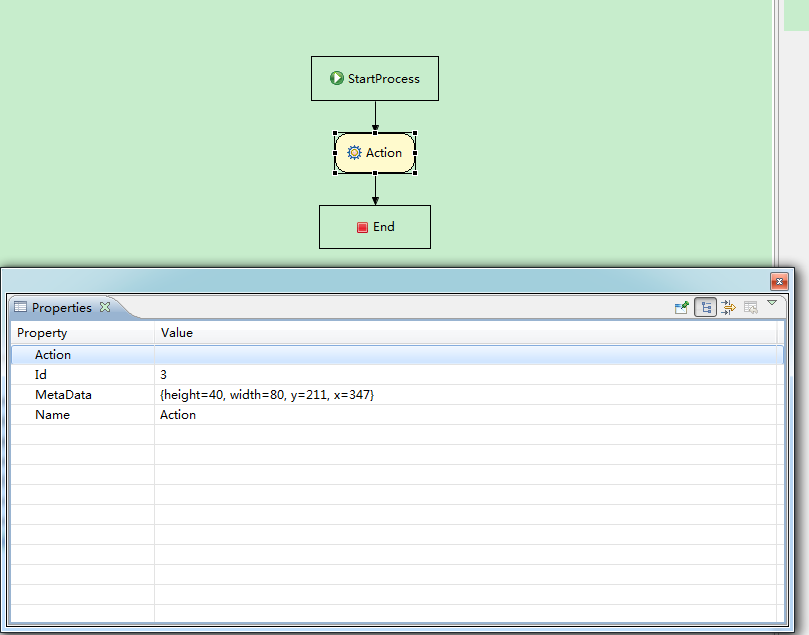
在流程可视化编辑界面上的每一个节点，我们选中它之后，只要单击节点上的文本，就可以编辑这个节点的名称（Name属性）了。

如果要编辑该节点的其他属性怎么办呢？尝试双击节点并不能弹出属性编辑窗口（属于工作项这一类的节点双击就有用，什么是工作项啊，先不说吧，嘿嘿。。）

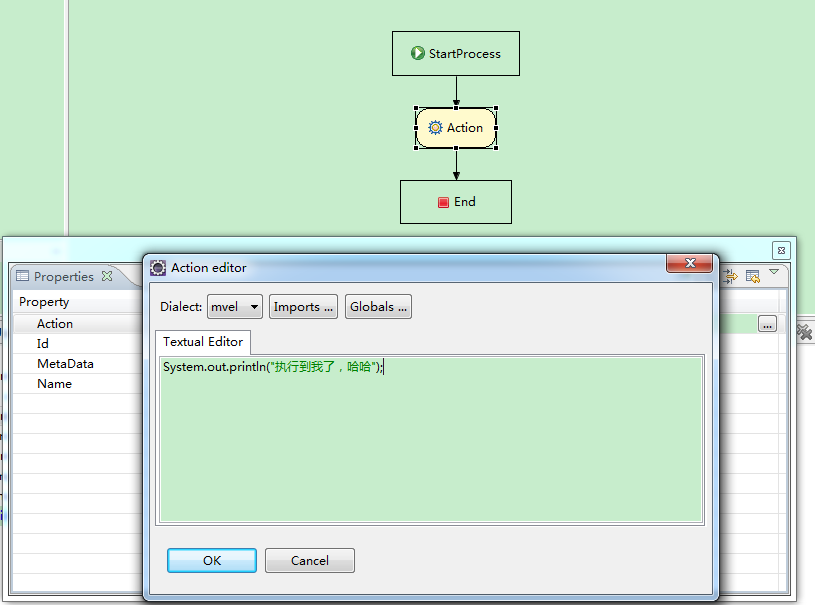
只要在eclipse的窗口视图中调出Properties属性窗口，在可视化流程编辑窗口中选中一个节点的同时，我们在Properties窗口中就可以看到，这个节点的所有属性了。

如果点击流程图的空白部分，Properties窗口中显示的就是整个流程定义的属性。

现在，选中“Action”节点：



这里我们只关注Action属性，点击其Value单元格后，右边会显示一个小按钮。点击按钮后弹出一个Action属性的编辑窗口，在这个窗口中我们只要在标准输入输出控制台打印一句话：



这个编辑窗口中的其他可操作部分，我们先不管，以后用到再说吧。

点击Ok，并Ctrl+S保存属性。这样这个流程文件就算定义好了。

1. 把上一步定义的bpmn流程文件的引用加入到Spring配置中“规则库”bean的定义里面（黄色高亮标出）：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:drools=*"http://drools.org/schema/drools-spring"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://drools.org/schema/drools-spring http://drools.org/schema/drools-spring-1.7.0.xsd"*>  <drools:kbase id=*"kbase"*>  <drools:resources>  <drools:resource type=*"DRL"* source=*"classpath:最简单的规则.drl"* />  <drools:resource type=*"BPMN2"* source=*"classpath:流程\_01\_最简单的流程.bpmn"* />  </drools:resources>  </drools:kbase>  </beans> |

3． 测试类 MyTest.java

获取到执行规则引擎的session对象后，调用它操作流程的startProcess相关方法，传的参数是整个流程的Id属性，这里是Sample：

|  |
| --- |
| **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession();  **try** {    session.startProcess("Sample");  // 执行规则  session.fireAllRules();  } **finally** {  // 最后释放资源  session.dispose();  }  }    } |

最后的输出结果：

|  |
| --- |
| 执行到我了，哈哈 |

我们发现session除了执行了流程，也执行了我们先前的drl规则定义。那是因为我们调用了session的fireAllRules 方法。这个方法会把规则库中加载到的所有规则定义都执行到，先这样理解吧。后面我们再用例子说明这个无参的fireAllRules ，默认情况下究竟执行了哪些规则定义。

**规则执行过程探究**

当我们创建了用于执行规则的session对象之后，在执行规则之前，可以通过session向规则引擎传入需要参与规则评估的事实对象。打个比方来说，学期末要对学生进行期末综合考评，Drools中定义的规则就相当于综合考评的各项评定标准，而一个学生的各科成绩就是一个参与规则执行的事实对象（Fact）。被执行到的每一个规则，会根据when语句块中定义的每一个条件对插入session的每一个Fact对象进行“模式匹配”。这种匹配方式随后会详细介绍。匹配成功的那些规则就变为了激活状态，这些激活状态的规则是按照什么顺序执行的呢？是不是激活的规则都会被执行呢？ 带着这样的疑问，我们开始下面的Drools之旅吧！

我们把Drools\_02项目复制一份，叫做Drools\_04，我们在这个项目下对规则执行过程进行探究。

我们会用用一个Student类作为域模型，它的对象会作为参与规则执行的Fact对象。现在我们在源码文件夹src下建一个包com.student，该包下建一个Student类：

|  |
| --- |
| **package** com.student;  **public** **class** Student {  **private** String name;  **private** **int** age;  **private** String address;    **public** Student() {    }    **public** Student(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **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** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address;  }  } |

其实，上面这个域模型Student就是一个最简单不过的POJO类。

在com.student包下面我们再建一个drl规则文件，01\_无序触发规则.drl，文件内容：

|  |
| --- |
| **package** com.student;  **rule** "匹配每个学生"  **when**  Student()  **then**  System.out.println("匹配到一个学生");  **end**  **rule** "匹配叫张三的学生"  **when**  Student(name == '张三')  **then**  System.out.println("匹配到一个学生叫张三");  **end**  **rule** "匹配每个未成年学生"  **when**  Student(age < 18)  **then**  System.out.println("匹配到一个未成年学生");  **end** |

注意一下，一般情况下这里的package名称可以任意起，但现在我们指定为com.student就意味着这个包下的Java类型如果在该规则文件定义中使用到，我们就无需再使用import语句导入相应的类型；否则必须要使用import导入类型，导入方式同Java的import语法。

现在我们在每个规则定义下的when语句块中都使用到了Student这个类型，类型后的括号中就是该类型的约束条件。这个约束条件的语法比较多，后面会介绍到。现在看到的是最简单的形式，约束条件为空就表示只匹配类型，如第一个规则，只匹配类型为Student的Fact对象。而第二、第三个规则分别是对Student的name和age属性进行了限制。

现在的MyTest.java测试类，我们要稍作调整，因为现在我们在执行规则之前，先调用session向规则引擎传入用于参与规则匹配的Student对象（也叫做Fact对象，以后就统一叫Fact对象吧。）

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  **import** com.student.Student;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession();    List facts = **null**;    facts = prepareFacts();    **try** {    **if** (**null** != facts && facts.size() > 0) {  **for** (Object fact : facts) {  session.insert(fact);  }  }    // 执行规则  session.fireAllRules();  } **finally** {  // 最后释放资源  session.dispose();  }  }    **private** List prepareFacts() {  List facts = **new** ArrayList();  facts.add(**new** Student("张三", 15));  **return** facts;  }    } |

这里我们只传入了一个Fact对象，也可以传入多个，后者我们后面会介绍。先看简单的情况。

还要注意别忘了在Spring配置中添加对上面定义的drl规则文件的引用。

运行测试，输出结果：

|  |
| --- |
| 匹配到一个未成年学生  匹配到一个学生  匹配到一个学生叫张三 |

按照我们本节开头将的匹配方式，三条规则都被激活了，但是最终的执行顺序却没有顺序可言。为了能控制这种情况我们可以对每个规则声明时加一个属性salience，属性值是一个数值，可以是正、0、负，数值越大，规则越优先触发。如：

|  |
| --- |
| **package** com.student;  **rule** "匹配每个学生" **salience** 99  **when**  Student()  **then**  System.out.println("匹配到一个学生");  **end**  **rule** "匹配叫张三的学生" **salience** 98  **when**  Student(name == '张三')  **then**  System.out.println("匹配到一个学生叫张三");  **end**  **rule** "匹配每个未成年学生" **salience** 97  **when**  Student(age < 18)  **then**  System.out.println("匹配到一个未成年学生");  **end** |

这样运行测试输出的结果就是：

|  |
| --- |
| 匹配到一个学生  匹配到一个学生叫张三  匹配到一个未成年学生 |

但是，这种方式的缺点是，如果我们修改这个规则文件在中间加一条新规则，同时我们希望规则还是按照从上到下的顺序执行，那么我们是不是需要重新为每个规则的salience编号呢？很麻烦。下面就介绍规则声明时的另一个属性**agenda-group** ，我们可以在上面drl规则定义中的每个规则上定义一个**agenda-group** ，但是这样之后，运行测试，我们发现一个规则都没有被触发到，这里我们还要指定一个属性**auto-focus ，**默认为false，要指定为true，这里我们为第一条规则指定。接下来我们想做的就是在第一条规则触发之后，能动态的指定下一条被触发的规则。那我们就在第一条规则的then语句块最后添加这样的程序代码：

|  |
| --- |
| kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup("group2").setFocus(); |

这样**agenda-group**被指定为group2 的规则就被触发了。注意了，这里的kcontext是一个Drools的隐式对象，我们还需要在drl文件头部import相应的类型：

|  |
| --- |
| **import** org.drools.runtime.**rule**.RuleContext; |

最终的规则文件：

|  |
| --- |
| **package** com.student;  **import** org.drools.runtime.**rule**.RuleContext;  **rule** "匹配每个学生"  **agenda-group** "group1"  **auto-focus** **true**  **when**  Student()  **then**  System.out.println("匹配到一个学生");  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup("group2").setFocus();  **end**  **rule** "匹配叫张三的学生"  **agenda-group** "group2"  **when**  Student(name == '张三')  **then**  System.out.println("匹配到一个学生叫张三");  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup("group3").setFocus();  **end**  **rule** "匹配每个未成年学生"  **agenda-group** "group3"  **when**  Student(age < 18)  **then**  System.out.println("匹配到一个未成年学生");  **end** |

最后的执行结果：

|  |
| --- |
| 匹配到一个学生  匹配到一个学生叫张三  匹配到一个未成年学生 |

在使用**agenda-group** 属性的时候要注意以下问题：

1. **这种动态指定下一个规则跳转的方式无法应用于在一个session插入多个同类型的Fact对象用于规则匹配的情况。比如说，要对一组学生同时根据一张综合评定表来进行考核。而应该每次通过session传入一个Fact对象来执行规则**。
2. 在1）中的限制前提下，可以给多个规则使用同一个**agenda-group** ，让它们位于同一个组。这样在执行到这组规则的时候，我们还可以使用**salience** 属性进一步指定在这组中规则的执行顺序。

上面说到**agenda-group** 属性应用于一组规则的情况，被分在同一组的规则先后都会被执行到。但是有时候会有这样的需求，对于同一组规则只要求匹配其中一条就返回，比如判断一个人的年龄段来决定他的年龄层次。比如规则是这样：

|  |  |
| --- | --- |
| 年龄段 | 类型 |
| age > 10 && age <18 | 青少年 |
| age >= 18 && age < 30 | 青年 |
| age >= 30 && age <60 | 中年 |
| age >= 60 | 老年 |

根据我们前面学过的知识，我们很容易就可以编写出这样的drl规则来：

|  |
| --- |
| **rule** "匹配青少年" **salience** 98  **when**  $p : Person(age >= 10 && < 18)  **then**  System.out.println("匹配到一个未成年人, id = " + $p.getId());  **end**  **rule** "匹配青年" **salience** 97  **when**  $p : Person(age < 30)  **then**  System.out.println("匹配到一个青年, id = " + $p.getId());  **end**  **rule** "匹配中年" **salience** 96  **when**  $p : Person(age < 60)  **then**  System.out.println("匹配到一个中年, id = " + $p.getId());  **end**  **rule** "匹配老年" **salience** 95  **when**  $p : Person(age >= 60)  **then**  System.out.println("匹配到一个老年, id = " + $p.getId());  **end** |

注：有可能你对上面规则中出现的“$p”不太理解，它是一个变量，用于保存“模式匹配”成功后的Fact对象。这个变量的作用域只是在一个rule定义块中。

同时执行规则前，我们插入一组Person的Fact对象列表：

|  |
| --- |
| **private** List prepareFacts() {  List<Person> facts = **new** ArrayList();  facts.add(**new** Person(1, "张三", 15));  facts.add(**new** Person(2, "李四", 28));  facts.add(**new** Person(3, "王五", 50));  facts.add(**new** Person(4, "赵六", 65));  **return** facts;  } |

（本实例请参考项目Drools\_04\_规则执行过程探究（续））

最后运行的结果却出乎我们的预料：

|  |
| --- |
| 匹配到一个未成年人, id = 1  匹配到一个青年, id = 2  匹配到一个青年, id = 1  匹配到一个中年, id = 3  匹配到一个中年, id = 2  匹配到一个中年, id = 1  匹配到一个老年, id = 4 |

我们发现“张三”在上面的规则中匹配了多条，并执行了多次。因为指定了**salience** ，上面每个独立的规则按照从上到下的顺序执行的，并没有对匹配完一条规则后就直接返回进行控制。这里我们要用到另外一个属性**activation-group** 。用这个属性将多个规则编为一组后，这组规则就变为互斥了。改进上面的drl规则定义后：

|  |
| --- |
| **rule** "匹配青少年" **salience** 98  **activation-group** "ageGroup"  **when**  $p : Person(age >= 10 && < 18)  **then**  System.out.println("匹配到一个未成年人, id = " + $p.getId());  **end**  **rule** "匹配青年" **salience** 97  **activation-group** "ageGroup"  **when**  $p : Person(age < 30)  **then**  System.out.println("匹配到一个青年, id = " + $p.getId());  **end**  **rule** "匹配中年" **salience** 96  **activation-group** "ageGroup"  **when**  $p : Person(age < 60)  **then**  System.out.println("匹配到一个中年, id = " + $p.getId());  **end**  **rule** "匹配老年" **salience** 95  **activation-group** "ageGroup"  **when**  $p : Person(age >= 60)  **then**  System.out.println("匹配到一个老年, id = " + $p.getId());  **end** |

再运行测试，得到结果：

|  |
| --- |
| 匹配到一个未成年人, id = 1 |

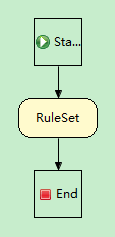
最后只对“张三”这个Fact对象进行了匹配并执行规则，而对于同类型的其他Fact对象并没有执行规则，这就类似于前面我们在讲**activation-group** 属性时提到的第一个注意点。使用group时，我们不能用session插入多个Fact对象来应用同一个规则文件定义，测试类改为这种方式就可以了：

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  **import** com.person.Person;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = **null**;    List facts = **null**;    facts = prepareFacts();    **if** (**null** != facts && facts.size() > 0) {  **for** (Object fact : facts) {  session = kbase.newStatefulKnowledgeSession();  **try** {  session.insert(fact);  // 执行规则  session.fireAllRules();  } **finally** {  // 最后释放资源  session.dispose();  }  }  }  }    **private** List prepareFacts() {  List<Person> facts = **new** ArrayList();  facts.add(**new** Person(1, "张三", 15));  facts.add(**new** Person(2, "李四", 28));  facts.add(**new** Person(3, "王五", 50));  facts.add(**new** Person(4, "赵六", 65));  **return** facts;  }    } |

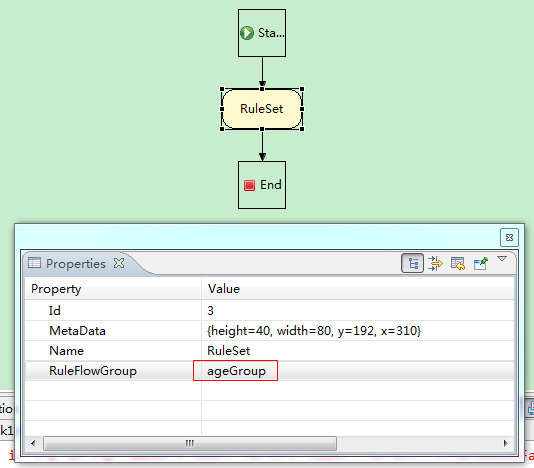
最后的执行结果：

|  |
| --- |
| 匹配到一个人，id = 1  匹配到一个未成年人, id = 1  匹配到一个人，id = 2  匹配到一个青年, id = 2  匹配到一个人，id = 3  匹配到一个中年, id = 3  匹配到一个人，id = 4  匹配到一个老年, id = 4 |

前面我们已经介绍了两个group（**agenda-group activation-group**）属性的用法。这里我们不得不提另一个group **ruleflow-group** 。这个用于在规则流中执行指定的一组规则。下面我们用一个实例来说明其用法。这里我们复制Drools\_03的规则流项目，命名为“Drools\_03\_最简单的规则流（续）\_执行规则”。我们新创建一个bpmn流程定义文件，在绘制流程图时，我们选择这个图标。定义完的流程图如下：



我们要对“RuleSet”节点编辑属性来绑定要执行的一组规则：



这里我们还是借用前面讲的“年龄段判断”的业务规则场景。上面截图中“RuleFlowGroup”属性指定为“ageGroup”，那么规则流在执行到这个规则节点时，会加载规则库中已装载的**ruleflow-group** 属性为“ageGroup”的一组规则。现在我们的drl规则定义如下：

|  |
| --- |
| **package** com.person;  **rule** "匹配每个人" **salience** 99  **when**  $p : Person()  **then**  System.out.println("匹配到一个人，id = " + $p.getId());  **end**  **rule** "匹配青少年" **salience** 98  **activation-group** "ageGroup"  **ruleflow-group** "ageGroup"  **when**  $p : Person(age >= 10 && < 18)  **then**  System.out.println("匹配到一个未成年人, id = " + $p.getId());  **end**  **rule** "匹配青年" **salience** 97  **activation-group** "ageGroup"  **ruleflow-group** "ageGroup"  **when**  $p : Person(age < 30)  **then**  System.out.println("匹配到一个青年, id = " + $p.getId());  **end**  **rule** "匹配中年" **salience** 96  **activation-group** "ageGroup"  **ruleflow-group** "ageGroup"  **when**  $p : Person(age < 60)  **then**  System.out.println("匹配到一个中年, id = " + $p.getId());  **end**  **rule** "匹配老年" **salience** 95  **activation-group** "ageGroup"  **ruleflow-group** "ageGroup"  **when**  $p : Person(age >= 60)  **then**  System.out.println("匹配到一个老年, id = " + $p.getId());  **end** |

上面指定了**ruleflow-group** 属性的规则，会在规则流中相应的规则节点触发时被执行到，注意了我们还声明了属性**activation-group** ，因为这组规则是互斥的。那么，前面我们一再强调，规则定义中使用了Group属性的，规则调用时插入Fact对象的方式：

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  **import** com.person.Person;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = **null**;    List facts = **null**;    facts = prepareFacts();    **if** (**null** != facts && facts.size() > 0) {  **for** (Object fact : facts) {    session = kbase.newStatefulKnowledgeSession();    **try** {    session.insert(fact);    session.startProcess("Sample");    // 执行规则  session.fireAllRules();    } **finally** {  // 最后释放资源  session.dispose();  }    }  }    }    **private** List prepareFacts() {  List<Person> facts = **new** ArrayList();  facts.add(**new** Person(1, "张三", 15));  facts.add(**new** Person(2, "李四", 28));  facts.add(**new** Person(3, "王五", 50));  facts.add(**new** Person(4, "赵六", 65));  **return** facts;  }    } |

最后运行的结果，就是我们想要的：

|  |
| --- |
| 匹配到一个人，id = 1  匹配到一个未成年人, id = 1  匹配到一个人，id = 2  匹配到一个青年, id = 2  匹配到一个人，id = 3  匹配到一个中年, id = 3  匹配到一个人，id = 4  匹配到一个老年, id = 4 |

让人兴奋的同时，这里有几个注意点：

1. session的insert方法一定要在startProcess方法之前调用，这是规则流中执行Drools规则的前提条件。
2. 不论是调用单独的drl规则还是规则流中的规则，最后一定要调用session的fireAllRules方法。

前面我们提到过一个问题，不知道你还记不记得，“是不是被激活的规则都会执行到呢”。从上面我们做的例子来看，答案是肯定的。但其实不尽然，如果我们在规则的执行过程中更新了Fact对象呢？

上面说到的情况是现实的业务规则场景中经常碰到的。比如，一个老师一手拿着一个学生的期末考试成绩表，一手拿着“期末综合评定标准”表，在给学生评优。。算到一半的时候，评定标准上说“贫困生”可以加分，老师头大了，怎么办，又要重新算。。。

其实上面描述的虚构场景，使用Drools规则引擎是完全可以实现的。规则引擎在执行Fact对象时，如果中途修改了Fact对象的信息，默认是有重新从头到尾匹配规则的特性的。那么当前导致Fact对象更新的规则触发后，其余的被激活等待触发的规则就都不“算数”了，也就是原来的“模式匹配”被废弃了，一切都重新来过吧。听的有点绕，下面我们用实例说明问题吧。

下面是“给张三改名儿”的一个drl规则定义：

|  |
| --- |
| **package** com.person;  **rule** "匹配叫张三儿的人"  **when**  Person(name == '张三儿')  **then**  System.out.println("匹配到一个叫张三儿的人");  **end**  **rule** "给张三改名儿规则" **salience** -1  **when**  $p : Person(name == '张三')  **then**  **modify**($p){  setName("张三儿");  }    System.out.println("叫张三的改名儿叫张三儿了");  **end** |

注意：这里我们给第二个规则指定**salience** 为“-1”，意味着先执行第一个规则，因为**salience** 默认是0。

这里我们用了一个新的关键字**modify** ，语法形式是：

|  |
| --- |
| **modify**(fact) {  setProp1(prop1),  setProp2(prop2)  } |

多个setXxx()调用之间用,分隔。

第一个规则不会被执行到，执行到第二个规则后，“张三”对象的name属性被修改了，规则引擎会重新匹配规则，结果第一个规则匹配上了，而第二个规则不再满足了。

这里要特别注意一个问题，如果条件控制不好，修改Fact对象重新触发规则可能会造成死循环！

比如：

|  |
| --- |
| **package** com.person;  **rule** "给一个人改名儿"  **when**  $p : Person()  **then**  **modify**($p){  setName("新名儿");  }    System.out.println("给一个人起了个新名儿");  **end** |

上面这个规则就会造成一个死循环，会不断重复执行这条规则。解决办法当然是有的，最容易想到的就是控制规则条件不再成立。当然，Drools给我们提供了不让规则重复触发的方法。

看这样一个规则：

|  |
| --- |
| **package** com.person;  **rule** "有人吗" **salience** 3  **when**  **exists** Person()  **then**  System.out.println("你好。我在的");  **end**  **rule** "确认是给张三送钱" **salience** 2  **when**  $p : Person(name == '张三')  **then**  $p.setMoneySender("李四"); // 设置送钱人  **update**($p);  System.out.println("李四给张三送钱来了");  **end**  **rule** "送钱规则" **salience** 1  **dialect** "mvel"  **when**  $p : Person(moneySender != **null** && != '')  **then**  $p.totalMoney = $p.totalMoney + 200.0;  **update**($p);  System.out.println($p.name + "现在手头的钱" + $p.totalMoney);  **end** |

（注：**update** 关键字，作用等同于前面介绍的**modify** 。第三个规则我们使用了关键字**dialect** 来指定了就这条规则使用何种脚本语言，默认是java，这里用的是一种更强大和灵活的表达式语言mvel，它有类似的对象导航图语言，可以直接用.访问属性。关于mvel相关的知识，根据学习的安排，我会另外提供相关的学习笔记，敬请期待哦~）

这个规则描述了这样一个场景：李四提着满满一麻袋钱，兴致冲冲的到张三家来送钱，他先敲门了，“有人在吗？”。“你好，在的”，一个声音冷冷回应。“你是张三吗，我给你送钱来了”，“哟原来是李四啊，快进来。。”。。。

运行这个程序，会发现规则引擎在不停的执行第二条规则（注意有些规则只会执行一次，像第一条规则，使用了关键字**exists** ，判断一个Fact对象是否存在）。送钱就送钱，不要寒暄太多啊。呵呵。。

要防止一个规则本身不被重复触发，可以使用属性**no-loop** 设置true。但是这之后，执行规则，发现第二和第三条规则在不断先后执行，很自然想到是第三条规则中的update语句修改Fact对象后，又触发了其他规则，当然和它本身。防止一个规则被其他规则的更新行为“带动”触发，可以使用属性**lock-on-active** ，设为true。最终我们想要的规则是：

|  |
| --- |
| **package** com.person;  **rule** "有人吗" **salience** 3  **when**  **exists** Person()  **then**  System.out.println("你好。我在的");  **end**  **rule** "确认是给张三送钱" **salience** 2  **no-loop** **true**  **lock-on-active** **true**  **when**  $p : Person(name == '张三')  **then**  $p.setMoneySender("李四"); // 设置送钱人  **update**($p);  System.out.println("李四给张三送钱来了");  **end**  **rule** "送钱规则" **salience** 1  **dialect** "mvel"  **no-loop** **true**  **when**  $p : Person(moneySender != **null** && != '')  **then**  $p.totalMoney = $p.totalMoney + 200.0;  **update**($p);  System.out.println($p.name + "现在手头的钱" + $p.totalMoney);  **end** |

上面我们花了很长的篇幅讲解了Drools规则引擎执行流程的过程，涉及到执行的顺序，触发规则的原则等等相关的细节，每种情况我们几乎都用一个实例来说明问题。下面我们就来对Drools规则语法进行全面的讲解。

**Drools规则语法详解**

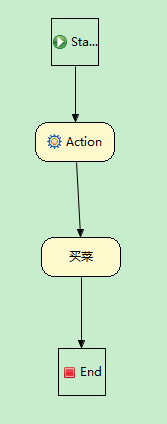
（敬请期待。。。）

**jBPM5.4学习进阶—基础篇**

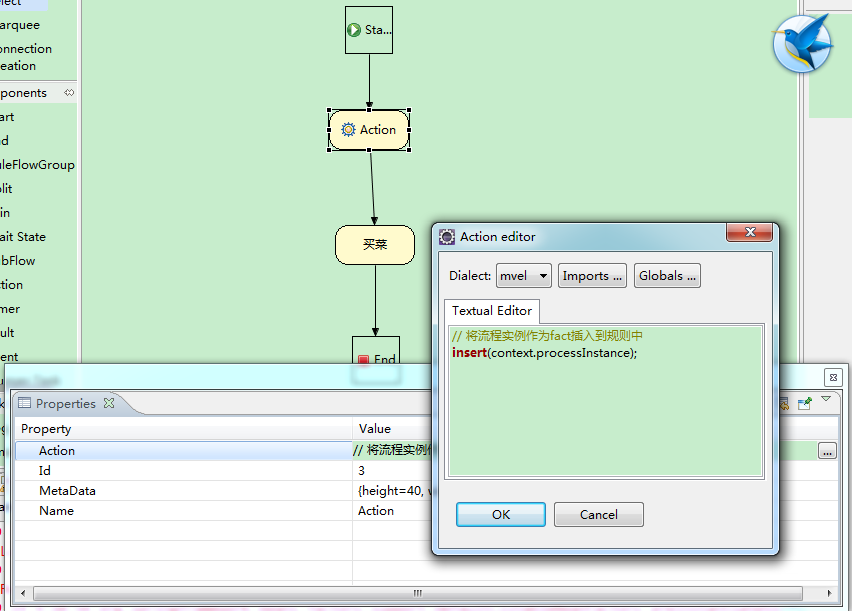
jBPM5是啥东东，自己网上去找，单是介绍其概念的网上一搜一箩筐。但是正真把每个功能点都用一个实例来反映的少之又少。废话少说，我们直接开始一个最简单的例子（现在还不涉及到人工任务）吧。

场景：周末一个人在宿舍，做什么好吃的呢。自己盘算了很久决定到菜市场上逛一逛。。

流程定义：



开始我们定义了一个Script任务节点，用于将jBPM5的流程实例插入到Drools规则定义中去，这里的processInstance对象现在就是我们用于执行Drools规则的一个Fact对象了。



（迅雷的小鸟也要抢镜头，没办法。。）

“买菜”的那个节点是“RuleSet”类型的，用于执行买菜的规则。对应的规则：

|  |
| --- |
| **package** com  **import** com.model.\*;  **import** org.jbpm.process.instance.impl.ProcessInstanceImpl;  **import** org.jbpm.process.instance.context.variable.VariableScopeInstance;  **import** org.jbpm.process.core.context.variable.VariableScope;  **import** org.jbpm.workflow.instance.WorkflowProcessInstance;  **rule** "买鸡" **salience** 3  **ruleflow-group** "buyVegetables"  **lock-on-active** **true**  **no-loop** **true**  **when**  processInstance : WorkflowProcessInstance()  **then**  **modify**(processInstance) {  setVariable("chicken", **new** Vegetable(Type.CHICKEN, 15.0f, 1));  };  System.out.println("买了鸡");  **end**  **rule** "买鸭" **salience** 2  **ruleflow-group** "buyVegetables"  **lock-on-active** **true**  **no-loop** **true**  **when**  processInstance : WorkflowProcessInstance()  **then**  **modify**(processInstance) {  setVariable("duck", **new** Vegetable(Type.DUCK, 14.0f, 1));  };  System.out.println("买了鸭");  **end**  **rule** "买鱼" **salience** 1  **ruleflow-group** "buyVegetables"  **lock-on-active** **true**  **no-loop** **true**  **when**  processInstance : WorkflowProcessInstance()  **then**  **modify**(processInstance) {  setVariable("fish", **new** Vegetable(Type.FISH, 9.0f, 2));  };  System.out.println("买了鱼");  **end** |

注意了，在这个drl定义中，我们实现了，jBPM5和Drools规则执行的交互。就是说用于执行jBPM的processInstance可以插入规则中（这个步骤我们在Action节点初始化了）作为匹配规则的一个Fact对象。当然随着规则的执行，我们可以将规则执行的结果保存到流程实例变量中。

最后看一下，测试用例：

|  |
| --- |
| **import** org.drools.KnowledgeBase;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.drools.runtime.process.ProcessInstance;  **import** org.drools.runtime.process.WorkflowProcessInstance;  **import** org.junit.Test;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;  @ContextConfiguration  **public** **class** MyTest **extends** AbstractJUnit4SpringContextTests {  @Autowired  **private** KnowledgeBase kbase;  @Test  **public** **void** test() {    StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession();  **try** {    ProcessInstance processInstance = session.startProcess("Sample");  // 执行规则  session.fireAllRules();    WorkflowProcessInstance process = (WorkflowProcessInstance) processInstance;    System.*out*.println(process.getVariable("chicken"));  System.*out*.println(process.getVariable("duck"));  System.*out*.println(process.getVariable("fish"));    } **finally** {  // 最后释放资源  session.dispose();  }  }    } |

大家发现启动了流程并触发了所有规则后，我们可以从流程实例变量中获取到跑规则流执行完的结果了。

最后的输出：

|  |
| --- |
| 买了鸡  买了鸭  买了鱼  Vegetable [type=CHICKEN, unitPrice=15.0, amount=1]  Vegetable [type=DUCK, unitPrice=14.0, amount=1]  Vegetable [type=FISH, unitPrice=9.0, amount=2] |

（注： 该实例请参考相应的项目源码）

**如何搭建jBPM5.4与Spring3集成的最简单的开发环境**

我们还是参照jBPM5.4官方提供文档来一步步搭建。

1. 项目中的依赖jar包

这个搭建过程最麻烦的就是需要添加哪些依赖的jar包，这个需要在启Spring的xml配置时，一点点的试，看控制台抛出的哪个类找不到，再一点点加。现在好了，我已经整合好了一个项目，项目中必需的jar包可以参考项目源码。

1. 核心的Spring配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"* xmlns:drools=*"http://drools.org/schema/drools-spring"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd*  *http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd*  *http://drools.org/schema/drools-spring http://drools.org/schema/drools-spring-1.7.0.xsd"*>  <bean id=*"jbpmEMF"*  class=*"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"*>  <property name=*"persistenceUnitName"* value=*"org.jbpm.persistence.local"* />  </bean>  <bean id=*"jbpmTxManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*>  <property name=*"entityManagerFactory"* ref=*"jbpmEMF"* />  <property name=*"nestedTransactionAllowed"* value=*"false"* />  </bean>  <drools:kbase id=*"kbase1"*>  <drools:resources>  <drools:resource type=*"BPMN2"*  source=*"classpath:com/test/流程\_01\_最简单的请假.bpmn"* />  </drools:resources>  </drools:kbase>  <drools:ksession id=*"ksession1"* type=*"stateful"* kbase=*"kbase1"*>  <drools:configuration>  <drools:jpa-persistence>  <drools:transaction-manager ref=*"jbpmTxManager"* />  <drools:entity-manager-factory ref=*"jbpmEMF"* />  </drools:jpa-persistence>  </drools:configuration>  </drools:ksession>  <bean id=*"systemEventListener"* class=*"org.drools.SystemEventListenerFactory"*  factory-method=*"getSystemEventListener"* />    <bean id=*"internalTaskService"* class=*"org.jbpm.task.service.TaskService"*>  <property name=*"systemEventListener"* ref=*"systemEventListener"* />  </bean>  <bean id=*"htTxManager"*  class=*"org.drools.container.spring.beans.persistence.HumanTaskSpringTransactionManager"*>  <constructor-arg ref=*"jbpmTxManager"* />  </bean>  <bean id=*"springTaskSessionFactory"*  class=*"org.jbpm.task.service.persistence.TaskSessionSpringFactoryImpl"*  init-method=*"initialize"* depends-on=*"internalTaskService"*>  <property name=*"entityManagerFactory"* ref=*"jbpmEMF"* />  <property name=*"transactionManager"* ref=*"htTxManager"* />  <property name=*"useJTA"* value=*"true"* />  <property name=*"taskService"* ref=*"internalTaskService"* />  </bean>  <bean id=*"taskService"* class=*"org.jbpm.task.service.local.LocalTaskService"*  depends-on=*"internalTaskService"*>  <constructor-arg ref=*"internalTaskService"* />  </bean>  </beans> |

这个配置主要就是参考了我翻译的jBPM5.4官方文档相关的章节搭建起来的，不理解的地方可以参照那个文档，也没什么好说的。

1. persistence.xml配置

因为jBPM5的底层的持久化用的是JPA来做的，我们需要在项目classpath下的META-INF目录下提供一个persistence.xml文件。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* standalone=*"yes"*?>  <persistence xmlns=*"http://java.sun.com/xml/ns/persistence"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"*  version=*"1.0"*>  <persistence-unit name=*"org.jbpm.persistence.local"*  transaction-type=*"RESOURCE\_LOCAL"*>  <provider>org.hibernate.ejb.HibernatePersistence  </provider>  <mapping-file>META-INF/ProcessInstanceInfo.hbm.xml  </mapping-file>  <mapping-file>META-INF/JBPMorm.xml</mapping-file>  <mapping-file>META-INF/Taskorm.xml</mapping-file>    <class>org.drools.persistence.info.SessionInfo</class>  <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>  <class>org.drools.persistence.info.WorkItemInfo</class>    <class>org.jbpm.task.Attachment</class>  <class>org.jbpm.task.Content</class>  <class>org.jbpm.task.BooleanExpression</class>  <class>org.jbpm.task.Comment</class>  <class>org.jbpm.task.Deadline</class>  <class>org.jbpm.task.Comment</class>  <class>org.jbpm.task.Deadline</class>  <class>org.jbpm.task.Delegation</class>  <class>org.jbpm.task.Escalation</class>  <class>org.jbpm.task.Group</class>  <class>org.jbpm.task.I18NText</class>  <class>org.jbpm.task.Notification</class>  <class>org.jbpm.task.EmailNotification</class>  <class>org.jbpm.task.EmailNotificationHeader</class>  <class>org.jbpm.task.PeopleAssignments</class>  <class>org.jbpm.task.Reassignment</class>  <class>org.jbpm.task.Status</class>  <class>org.jbpm.task.Task</class>  <class>org.jbpm.task.TaskData</class>  <class>org.jbpm.task.SubTasksStrategy</class>  <class>org.jbpm.task.OnParentAbortAllSubTasksEndStrategy  </class>  <class>org.jbpm.task.OnAllSubTasksEndParentEndStrategy  </class>  <class>org.jbpm.task.User</class>    <properties>  <property name=*"hibernate.dialect"*  value=*"org.hibernate.dialect.MySQL5Dialect"*/>  <property name=*"hibernate.max\_fetch\_depth"* value=*"3"*/>  <property name=*"hibernate.hbm2ddl.auto"*  value=*"create"*/>  <property name=*"hibernate.show\_sql"* value=*"true"*/>    <property name=*"hibernate.connection.driver\_class"*  value=*"com.mysql.jdbc.Driver"*/>  <property name=*"hibernate.connection.url"*  value=*"jdbc:mysql://localhost/jbpm5"*/>  <property name=*"hibernate.connection.username"*  value=*"root"*/>  <property name=*"hibernate.connection.password"*  value=*"sa"*/>  <property name=*"hibernate.connection.pool\_size"*  value=*"3"*/>  </properties>  </persistence-unit>  </persistence> |

上面的配置文件我们是基于JPA1.0规范的。使用的是MySQL数据库。加载这个配置之后，就会在我们的MySQL数据库中生成jBPM5.4相关的所有数据库表。

1. 新建一个最简单的bpmn流程定义文件

为了配合验证我们的整合是否正确、有效，我们建了一个简单的人工任务相关的流程定义。

|  |
| --- |
|  |

上面的流程定义里面有一个人工任务节点。这个节点，我们直接双击它就可以编辑属性了。现在我们只需要关注它的Name和Actor(s)，这两个属性（其他的属性我们在后面的学习笔记中会一个个的介绍到的）。Name属性随便命名一个，如“leaveApprove”。Actor属性指定了，谁要来执行这个人工任务，通常对应于系统中的一个登陆账号（当然也可以是一组人哦）。现在我们就把这个属性的值写死为“zhangsan”。

另外，人工任务节点，前后的两个节点是两个脚本任务节点，我们就简单的打印一句话。

1. 最后是我们的测试类：

|  |
| --- |
| **package** com.test;  **import** **static** org.junit.Assert.\*;  **import** java.util.List;  **import** org.drools.runtime.StatefulKnowledgeSession;  **import** org.jbpm.process.workitem.wsht.~~SyncWSHumanTaskHandler~~;  **import** org.jbpm.task.Status;  **import** org.jbpm.task.User;  **import** org.jbpm.task.query.TaskSummary;  **import** org.jbpm.task.service.TaskServiceSession;  **import** org.jbpm.task.service.local.LocalTaskService;  **import** org.jbpm.task.service.persistence.TaskSessionFactory;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** MyTest\_01 {    **static** ApplicationContext *applicationContext*;    **static** TaskSessionFactory *springTaskSessionFactory*;    **static** LocalTaskService *taskService*;    **static** StatefulKnowledgeSession *ksession1*;    @BeforeClass  **public** **static** **void** init() {  *applicationContext* = **new** ClassPathXmlApplicationContext(**new** String[]{  "classpath:com/test/app-context-01.xml"  });  *springTaskSessionFactory* = (TaskSessionFactory)*applicationContext*.getBean("springTaskSessionFactory");  *taskService* = (LocalTaskService)*applicationContext*.getBean("taskService");  *ksession1* = (StatefulKnowledgeSession)*applicationContext*.getBean("ksession1");  }    @Test  **public** **void** test() {    TaskServiceSession taskServiceSession = *springTaskSessionFactory*.createTaskServiceSession();  taskServiceSession.addUser(**new** User("Administrator"));  taskServiceSession.addUser(**new** User("zhangsan"));    ~~SyncWSHumanTaskHandler~~ humanTaskHandler = **new** SyncWSHumanTaskHandler(*taskService*, *ksession1*);  humanTaskHandler.setLocal(**true**);  humanTaskHandler.connect();  *ksession1*.getWorkItemManager().registerWorkItemHandler("Human Task", humanTaskHandler);    *ksession1*.startProcess("leaveProcess");    List<TaskSummary> tasks = *taskService*  .getTasksAssignedAsPotentialOwner("zhangsan", "en-UK");  *assertEquals*(1, tasks.size());    TaskSummary task = tasks.get(0);  *assertEquals*("leaveApprove", task.getName());  *assertEquals*(Status.*Reserved*, task.getStatus()); // 断言任务状态为预定了  *taskService*.start(task.getId(), "zhangsan"); // 开始执行任务  *taskService*.complete(task.getId(), "zhangsan", **null**); // 完成任务  }    } |

这里我们使用的是junit4跑的单元测试，一开始集成的是Spring对junit4的包装类，发现测试后，抛出来的异常信息根本就不明确，很难发现是jar少了还是其他什么原因。后来就改为直接使用junit4的方式了。注意一下这个问题就行了，不要走我的弯路就行了。

测试方法中的API在jBPM5.4的官方文档上也可以找得到。注意，开始流程之前我们先是，创建了需要审批流程的参与者的组织架构，这里千万要注意，开始一定要配置一个“Administrator”，这是我的教训。。呵呵、

接下来我们需要实例化一个人工任务工作项处理程序，用它来执行人工任务工作项，来完成流程引擎和人工任务服务之间的交互。

好了，这样一个例子就做好了，运行完全没有问题，兴奋吧。其实我还搞了大半天的呢。

jBPM5.4中自定义工作项

关于这一块的概念性的东西，还是请大家参考我翻译整理的相关文档《jBPM5.4翻译资料（用于辅助参考）》，以及相关的“奥特曼打怪兽”的小项目。

关于工作项和流程之间的信息传递（使用参数映射的方式），这里要说明下：

|  |
| --- |
| /\*  \* 从这里我们发现了这样一个流程实例变量中的参数和工作项参数进行映射的一个规律，就是  \* 只要在工作项中指定了Parameter Mapping，引擎就会把流程实例变量的引用拷贝给工作项参数变量  \* 而，如果工作项中还指定了Result Mapping，就会把工作项中参数变量的引用再拷贝回流程中，  \* 而流程向工作项根据匹配的情况，会找workItemHandler.completeWorkItem()方法中第二个map参数的信息，  \* 如果这个map为空，或者key对应的value为空，拷贝给流程的变量值也就为空。  \*/ |