

Drools5 规则引擎 开发教程

高杰

上海锐道信息技术有限公司

2009-8-20

1. 学习前的准备

Drools 是一款基于 Java 的开源规则引擎，所以在使用 Drools 之前需要在开发机器上安装好 JDK 环境，Drools5 要求的 JDK 版本要在 1.5 或以上。

1.1. 开发环境搭建

大多数软件学习的第一步就是搭建这个软件的开发环境，Drools 也不例外。本小节的内容就是介绍如何搭建一个 Drools5 的开发、运行、调试环境。

1.1.1. 下载开发工具

Drools5 提供了一个基于 Eclipse3.4 的一个 IDE 开发工具，所以在使用之前需要到 <http://eclipse.org> 网站下载一个 3.4.x 版本的 Eclipse，下载完成之后，再到 <http://jboss.org/drools/downloads.html> 网站，下载 Drools5 的 Eclipse 插件版 IDE 及 Drools5 的开发工具包，如图 1-1 所示。

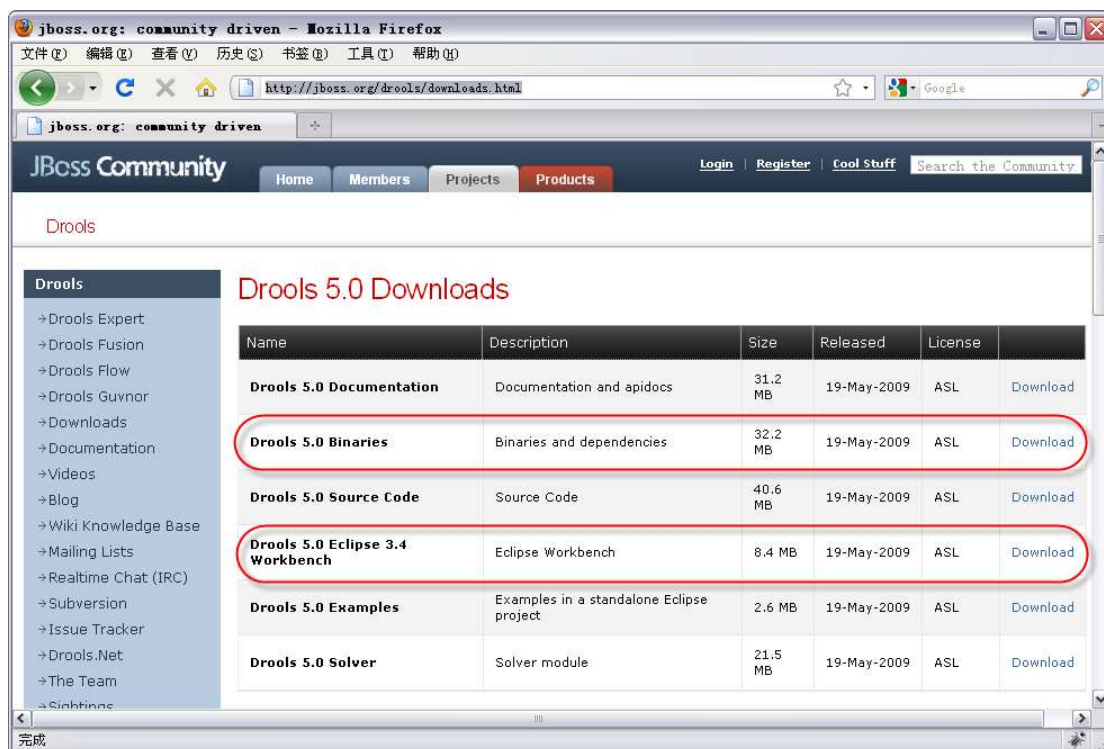


图 1-1

除这两个下载包以外，还可以把 Drools5 的相关文档、源码和示例的包下载下来参考学习使用。

将下载的开发工具包及 IDE 包解压到一个非中文目录下，解压完成后就可以在 Eclipse3.4 上安装 Drools5 提供的开发工具 IDE 了。

1.1.2. 安装 Drools IDE

打开 Eclipse3.4 所在目录下的 links 目录（如果该目录不存在可以手工在其目录下创建一个 links 目录），在 links 目录下创建一个文本文件，并改名为 drools5-ide.link，用记事本打开该文件，按照下面的版本输入 Drools5 Eclipse Plugin 文件所在目录：

```
path=D:\\eclipse\\drools-5.0-eclipse-all
```

这个值表示 Drools5 Eclipse Plugin 文件位于 D 盘 eclipse 目录下的 drools-5.0-eclipse-all 下面，这里有一点需要注意，那就是 drools-5.0-eclipse-all 文件夹下必须再包含一个 eclipse 目录，所有的插件文件都应该位于该 eclipse 目录之下，接下来要在 win dos 下重启 Eclipse 3.4，检验 Drools5 IDE 是否安装成功。

进入 win dos，进入 Eclipse3.4 所在目录，输入 eclipse -clean 启动 Eclipse3.4。启动完成

后打开菜单 Window→Preferences，在弹出的窗口当中如果能在左边导航树中发现 Drools 节点就表示 Drools5 IDE 安装成功了，如图 1-2 所示。

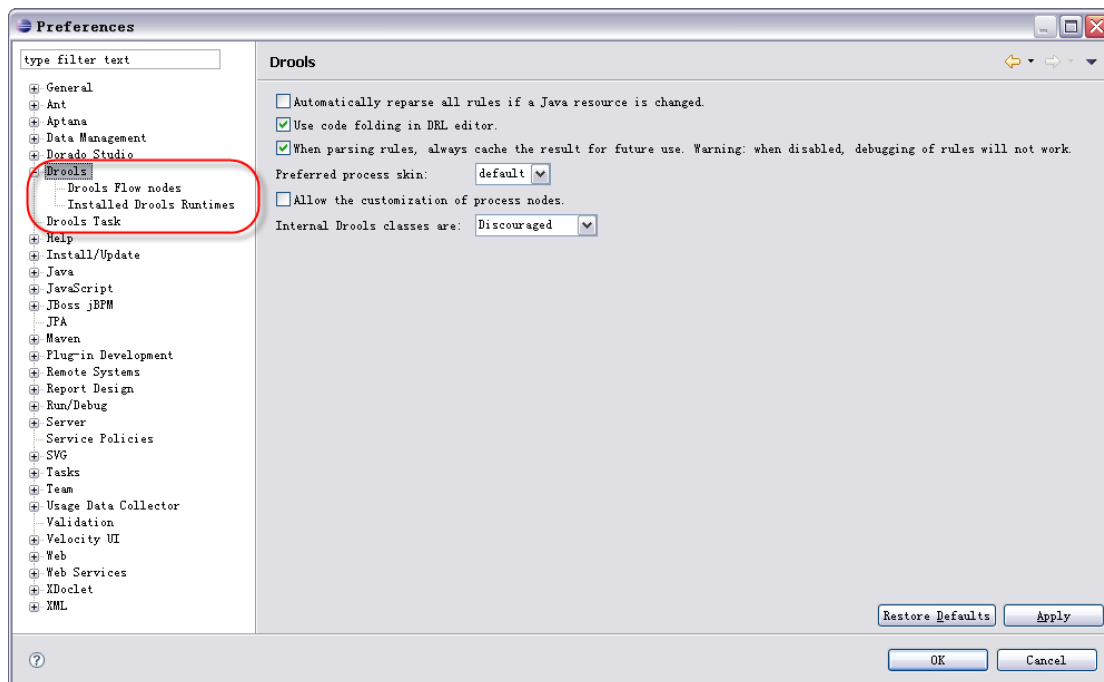


图 1-2

IDE 安装完成后，接下来需要对 Drools5 的 IDE 环境进行简单的配置，打开菜单 Window→Preferences，在弹出的窗口当中选择左边导航树菜单 Drools→Installed Drools Runtimes 设置 Drools5 IDE 运行时依赖的开发工具包，点击 “Add...” 按钮添加一个开发工具包，如图 1-3 所示。

KnowledgeBuilder、KnowledgeBase、StatefulKnowledgeSession、StatelessKnowledgeSession、等，它们起到了对规则文件进行收集、编译、查错、插入 fact、设置 global、执行规则或规则流等作用，在正式接触各种类型的规则文件编写方式及语法讲解之前，我们有必要先熟悉一下这些 API 的基本含义及使用方法。

1.3.1. KnowledgeBuilder

规则编写完成之后，接下来的工作就是在应用的代码当中调用这些规则，利用这些编写好的规则帮助我们处理业务问题。KnowledgeBuilder 的作用就是用来在业务代码当中收集已经编写好的规则，然后对这些规则文件进行编译，最终产生一批编译好的规则包（KnowledgePackage）给其它的应用程序使用。KnowledgeBuilder 在编译规则的时候可以通过其提供的 hasErrors()方法得到编译规则过程中发现规则是否有错误，如果有的话通过其提供的 getErrors()方法将错误打印出来，以帮助我们找到规则当中的错误信息。

创建 KnowledgeBuilder 对象使用的是 KnowledgeBuilderFactory 的 newKnowledgeBuilder 方法。代码清单 1-1 就演示了 KnowledgeBuilder 的用法。

代码清单 1-1

```
package test;

import java.util.Collection;

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;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kbuilder=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
Test.class),ResourceType.DRL);
        Collection<KnowledgePackage>
kpackage=kbuilder.getKnowledgePackages();//产生规则包的集合
    }
}
```

通过 KnowledgeBuilder 编译的规则文件的类型可以有很多种，如.drl 文件、.dslr 文件或一个 xls 文件等。产生的规则包可以是具体的规则文件形成的，也可以是规则流（rule flow）文件形成的，在添加规则文件时，需要通过使用 ResourceType 的枚举值来指定规则文件的类型；同时在指定规则文件的时候 drools 还提供了一个名为 ResourceFactory 的对象，通过该对象可以实现从 Classpath、URL、File、ByteArray、Reader 或诸如 XLS 的二进制文件里添加加载规则。

在规则文件添加完成后，可以通过使用 hasErrors()方法来检测已添加进去的规则当中有没有错误，如果不通过该方法检测错误，那么如果规则当中存在错误，最终在使用的时候也会将错误抛出。代码清单 1-2 就演示了通过 KnowledgeBuilder 来检测规则当中有没有错误。

代码清单 1-2

```
package test;

import java.util.Collection;
import java.util.Iterator;

import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderErrors;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.definition.KnowledgePackage;
import org.drools.io.ResourceFactory;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kbuilder=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
Test.class),ResourceType.DRL);
        if(kbuilder.hasErrors()){
            System.out.println("规则中存在错误，错误消息如下：");
            KnowledgeBuilderErrors kbuidlerErrors=kbuilder.getErrors();
            for(Iterator
iter=kbuidlerErrors.iterator();iter.hasNext();){
                System.out.println(iter.next());
            }
        }
        Collection<KnowledgePackage>
kpackage=kbuilder.getKnowledgePackages();//产生规则包的集合
    }
```

```
}
```

后面随着介绍的深入我们还会看到 KnowledgeBuilder 的一些其它用法。

1.3.2. KnowledgeBase

KnowledgeBase 是 Drools 提供的用来收集应用当中知识（knowledge）定义的知识库对象，在一个 KnowledgeBase 当中可以包含普通的规则（rule）、规则流(rule flow)、函数定义(function)、用户自定义对象（type model）等。KnowledgeBase 本身不包含任何业务数据对象（fact 对象，后面有相应章节着重介绍 fact 对象），业务对象都是插入到由 KnowledgeBase 产生的两种类型的 session 对象当中(StatefulKnowledgeSession 和 StatelessKnowledgeSession，后面会有对应的章节对这两种类型的对象进行介绍)，通过 session 对象可以触发规则执行或开始一个规则流执行。

创建一个 KnowledgeBase 要通过 KnowledgeBuilderFactory 对象提供的 newKnowledgeBase() 方法来实现，这其中创建的时候还可以为其指定一个 KnowledgeBaseConfiguration 对象，KnowledgeBaseConfiguration 对象是一个用来存放规则引擎运行时相关环境参数定义的配置对象，代码清单 1-3 演示了一个简单的 KnowledgeBase 对象的创建过程。

代码清单 1-3

```
package test;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;

public class Test {
    public static void main(String[] args) {
        KnowledgeBase kbase=KnowledgeBuilderFactory.newKnowledgeBase();
    }
}
```

代码清单 1-4 演示了创建 KnowledgeBase 过程当中，使用一个 KnowledgeBaseConfiguration 对象来设置环境参数。

代码清单 1-4

```
package test;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseConfiguration;
import org.drools.KnowledgeBuilderFactory;
```



```
public class Test {  
  
    public static void main(String[] args) {  
        KnowledgeBaseConfiguration kbConf =  
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();  
        kbConf.setProperty( "org.drools.sequential", "true");  
        KnowledgeBase kbase =  
KnowledgeBaseFactory.newKnowledgeBase(kbConf);  
    }  
}
```

从代码清单 1-4 中可以看到，创建一个 `KnowledgeBaseConfiguration` 对象的方法也是使用 `KnowledgeBaseFactory`，使用的是其提供的 `newKnowledgeBaseConfiguration()` 方法，该方法创建好的 `KnowledgeBaseConfiguration` 对象默认情况下会加载 `drools-core-5.0.1.jar` 包下 `META-INF/drools.default.rulebase.conf` 文件里的规则运行环境配置信息，加载完成后，我们可以在代码中对这些默认的信息重新赋值，以覆盖加载的默认值，比如这里我们就把 `org.drools.sequential` 的值修改为 `true`，它的默认值为 `false`。

除了这种方式创建 `KnowledgeBaseConfiguration` 方法之外，我们还可以为其显示的指定一个 `Properties` 对象，在该对象中设置好需要覆盖默认值的相关属性的值，然后再通过 `newKnowledgeBaseConfiguration(Properties prop, ClassLoader loader)` 方法创建一个 `KnowledgeBaseConfiguration` 对象。该方法方法当中第一个参数就是我们要设置的 `Properties` 对象，第二个参数用来设置加载 `META-INF/drools.default.rulebase.conf` 文件的 `ClassLoader`，因为该文件在 `ClassPath` 下，所以采用的是 `ClassLoader` 方法进行加载，如果不指定这个参数，那么就取默认的 `ClassLoader` 对象，如果两个参数都为 `null`，那么就和 `newKnowledgeBaseConfiguration()` 方法的作用相同了，代码清单代码清单 1-5 演示了这种用法。

代码清单 1-5

```
package test;  
  
import java.util.Properties;  
  
import org.drools.KnowledgeBase;  
import org.drools.KnowledgeBaseConfiguration;  
import org.drools.KnowledgeBaseFactory;  
  
public class Test {  
  
    public static void main(String[] args) {
```

```
Properties properties = new Properties();
properties.setProperty( "org.drools.sequential", "true");
KnowledgeBaseConfiguration kbConf =
KnowledgeBuilderFactory.newKnowledgeBaseConfiguration(properties, null);
KnowledgeBase kbase =
KnowledgeBuilderFactory.newKnowledgeBase(kbConf);
}
}
```

用来设置默认规则运行环境文件 `drools.default.rulebase.conf` 里面所涉及到的具体项内容如代码清单 1-6 所示:

代码清单 1-6

```
drools.maintainTms = <true|false>
drools.assertBehaviour = <identity|equality>
drools.logicalOverride = <discard|preserve>
drools.sequential = <true|false>
drools.sequential.agenda = <sequential|dynamic>
drools.removeIdentities = <true|false>
drools.shareAlphaNodes = <true|false>
drools.shareBetaNodes = <true|false>
drools.alphaNodeHashingThreshold = <1...n>
drools.compositeKeyDepth = <1..3>
drools.indexLeftBetaMemory = <true|false>
drools.indexRightBetaMemory = <true|false>
drools.consequenceExceptionHandler = <qualified class name>
drools.maxThreads = <-1|1..n>
drools.multithreadEvaluation = <true|false>
```

在后面的内容讲解过程当中, 对于代码清单 1-6 里列出的属性会逐个涉及到, 这里就不再多讲了。

KnowledgeBase 创建完成之后, 接下来就可以将我们前面使用 KnowledgeBuilder 生成的 KnowledgePackage 的集合添加到 KnowledgeBase 当中, 以备使用, 如代码清单 1-7 所示。

代码清单 1-7

```
package test;

import java.util.Collection;
```

```
import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseConfiguration;
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;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
            .newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
            Test.class), ResourceType.DRL);
        Collection<KnowledgePackage> kpackage =
kbuilder.getKnowledgePackages();

        KnowledgeBaseConfiguration kbConf = KnowledgeBaseFactory
            .newKnowledgeBaseConfiguration();
        kbConf.setProperty("org.drools.sequential", "true");
        KnowledgeBase kbase =
KnowledgeBaseFactory.newKnowledgeBase(kbConf);
        kbase.addKnowledgePackages(kpackage); //将KnowledgePackage集合添
加到KnowledgeBase当中
    }
}
```

1.3.3. StatefulKnowledgeSession

规则编译完成之后，接下来就需要使用一个 API 使编译好的规则包文件在规则引擎当中运行起来。在 Drools5 当中提供了两个对象与规则引擎进行交互：StatefulKnowledgeSession 和 StatelessKnowledgeSession，本小节当中要介绍的是 StatefulKnowledgeSession 对象，下面的一节将对 StatelessKnowledgeSession 对象进行讨论。

StatefulKnowledgeSession 对象是一种最常用的与规则引擎进行交互的方式，它可以与规则引擎建立一个持续的交互通道，在推理计算的过程当中可能会多次触发同一数据集。在用户的代码当中，最后使用完 StatefulKnowledgeSession 对象之后，一定要调用其 dispose()方

法以释放相关内存资源。

`StatefulKnowledgeSession` 可以接受外部插入 (insert) 的业务数据——也叫 `fact`，一个 `fact` 对象通常是一个普通的 Java 的 POJO，一般它们会有若干个属性，每一个属性都会对应 `getter` 和 `setter` 方法，用来对外提供数据的设置与访问。一般来说，在 Drools 规则引擎当中，`fact` 所承担的作用就是将规则当中要用到的业务数据从应用当中传入进来，对于规则当中产生的数据及状态的变化通常不用 `fact` 传出。如果在规则当中需要有数据传出，那么可以通过在 `StatefulKnowledgeSession` 当中设置 `global` 对象来实现，一个 `global` 对象也是一个普通的 Java 对象，在向 `StatefulKnowledgeSession` 当中设置 `global` 对象时不用 `insert` 方法而用 `setGlobal` 方法实现。

创建一个 `StatefulKnowledgeSession` 要通过 `KnowledgeBase` 对象来实现，下面的代码清单 1-8 就演示了 `StatefulKnowledgeSession` 的用法：

代码清单 1-8

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseConfiguration;
import org.drools.KnowledgeBuilderFactory;
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 Test {

    public static void main(String[] args) {
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
            .newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
            Test.class), ResourceType.DRL);
        Collection<KnowledgePackage> kpackage =
kbuilder.getKnowledgePackages();

        KnowledgeBaseConfiguration kbConf = KnowledgeBuilderFactory
            .newKnowledgeBaseConfiguration();
        kbConf.setProperty("org.drools.sequential", "true");
```

```

        KnowledgeBase kbase =
KnowledgeBuilderFactory.newKnowledgeBase(kbConf);

        kbase.addKnowledgePackages(kpackage); //将KnowledgePackage集合添
加到KnowledgeBase当中

        StatefulKnowledgeSession
statefulKSession=kbase.newStatefulKnowledgeSession();

        statefulKSession.setGlobal("globalTest", new Object()); //设置一
个global对象

        statefulKSession.insert(new Object()); //插入一个fact对象
        statefulKSession.fireAllRules();
        statefulKSession.dispose();

    }
}

```

代码清单 1-8 当中同时也演示了规则完整的运行处理过程，可以看到，它的过程是首先需要通过使用 **KnowledgeBuilder** 将相关的规则文件进行编译，产生对应的 **KnowledgePackage** 集合，接下来再通过 **KnowledgeBase** 把产生的 **KnowledgePackage** 集合收集起来，最后再产生 **StatefulKnowledgeSession** 将规则当中需要使用的 **fact** 对象插入进去、将规则当中需要用到的 **global** 设置进去，然后调用 **fireAllRules()** 方法触发所有的规则执行，最后调用 **dispose()** 方法将内存资源释放。

1.3.4. StatelessKnowledgeSession

StatelessKnowledgeSession 的作用与 **StatefulKnowledgeSession** 相仿，它们都是用来接收业务数据、执行规则的。事实上，**StatelessKnowledgeSession** 对 **StatefulKnowledgeSession** 做了包装，使得在使用 **StatelessKnowledgeSession** 对象时不需要再调用 **dispose()** 方法释放内存资源了。

因为 **StatelessKnowledgeSession** 本身所具有的一些特性，决定了它的使用有一定的局限性。在使用 **StatelessKnowledgeSession** 时不能进行重复插入 **fact** 的操作、也不能重复的调用 **fireAllRules()** 方法来执行所有的规则，对应这些要完成的工作在 **StatelessKnowledgeSession** 当中只有 **execute(...)** 方法，通过这个方法可以实现插入所有的 **fact** 并且可以同时执行所有的规则或规则流，事实上也就是在执行 **execute(...)** 方法的时候就在 **StatelessKnowledgeSession** 内部执行了 **insert()** 方法、**fireAllRules()** 方法和 **dispose()** 方法。

代码清单 1-9 演示了 **StatelessKnowledgeSession** 对象的用法。

代码清单 1-9

```
package test;

import java.util.ArrayList;
import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseConfiguration;
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.StatelessKnowledgeSession;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
            .newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
            Test.class), ResourceType.DRL);
        Collection<KnowledgePackage> kpackage =
kbuilder.getKnowledgePackages();

        KnowledgeBaseConfiguration kbConf = KnowledgeBaseFactory
            .newKnowledgeBaseConfiguration();
        kbConf.setProperty("org.drools.sequential", "true");
        KnowledgeBase kbase =
KnowledgeBaseFactory.newKnowledgeBase(kbConf);
        kbase.addKnowledgePackages(kpackage); //将KnowledgePackage集合添
加到KnowledgeBase当中

        StatelessKnowledgeSession
statelessKSession=kbase.newStatelessKnowledgeSession();
        ArrayList list=new ArrayList();
        list.add(new Object());
        list.add(new Object());
        statelessKSession.execute(list);
    }
}
```

代码清单 1-9 当中，通过新建了一个 ArrayList 对象，将需要插入到

StatelessKnowledgeSession 当中的对象放到这个 ArrayList 当中，将这个 ArrayList 作为参数传给 execute(...)方法，这样在 StatelessKnowledgeSession 内部会对这个 ArrayList 进行迭代，取出其中的每一个 Element，将其作为 fact，调用 StatelessKnowledgeSession 对象内部的 StatefulKnowledgeSession 对象的 insert() 方法将产生的 fact 逐个插入到 StatefulKnowledgeSession 当中，然后调用 StatefulKnowledgeSession 的 fireAllRules()方法，最后执行 dispose()方法释放内存资源。

在代码清单 1-9 当中，如果我们要插入的 fact 就是这个 ArrayList 而不是它内部的 Element 那该怎么做呢？在 StatelessKnowledgeSession 当中，还提供了 execute(Command cmd)的方法，在该方法中通过 CommandFactory 可以创建各种类型的 Command，比如前面的需求要直接将这个 ArrayList 作为一个 fact 插入，那么就可以采用 CommandFactory.newInsert(Object obj)来实现，代码清单 1-9 当中 execute 方法可做如代码清单 1-10 所示的修改。

代码清单 1-10

```
statelessKSession.execute(CommandFactory.newInsert(list));
```

如果需要通过 StatelessKnowledgeSession 设置 global 的话，可以使用 CommandFactory.newSetGlobal("key",Object obj)来实现；如果即要插入若干个 fact，又要设置相关的 global，那么可以将 CommandFactory 产生的 Command 对象放在一个 Collection 当中，然后再通过 CommandFactory.newBatchExecution(Collection collection)方法实现。代码清单 1-11 演示了这种做法。

代码清单 1-11

```
package test;

import java.util.ArrayList;
import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseConfiguration;
import org.drools.KnowledgeBaseFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.command.Command;
import org.drools.command.CommandFactory;
import org.drools.definition.KnowledgePackage;
import org.drools.io.ResourceFactory;
import org.drools.runtime.StatelessKnowledgeSession;
```

```
public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
            .newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("test.drl",
            Test.class), ResourceType.DRL);
        Collection<KnowledgePackage> kpackage =
kbuilder.getKnowledgePackages();

        KnowledgeBaseConfiguration kbConf = KnowledgeBaseFactory
            .newKnowledgeBaseConfiguration();
        kbConf.setProperty("org.drools.sequential", "true");
        KnowledgeBase kbase =
KnowledgeBaseFactory.newKnowledgeBase(kbConf);
        kbase.addKnowledgePackages(kpackage); //将KnowledgePackage集合添
加到KnowledgeBase当中

        StatelessKnowledgeSession
statelessKSession=kbase.newStatelessKnowledgeSession();
        ArrayList<Command> list=new ArrayList<Command>();
        list.add(CommandFactory.newInsert(new Object()));
        list.add(CommandFactory.newInsert(new Object()));
        list.add(CommandFactory.newSetGlobal("key1", new Object()));
        list.add(CommandFactory.newSetGlobal("key2", new Object()));

        statelessKSession.execute(CommandFactory.newBatchExecution(list))
;
    }
}
```

1.4. Fact 对象

Fact 是指在 Drools 规则应用当中, 将一个普通的 JavaBean 插入到规则的 WorkingMemory 当中后的对象。规则可以对 Fact 对象进行任意的读写操作, 当一个 JavaBean 插入到 WorkingMemory 当中变成 Fact 之后, Fact 对象不是对原来的 JavaBean 对象进行 Clon, 而是原来 JavaBean 对象的引用。规则在进行计算的时候需要用到应用系统当中的数据, 这些数据设置在 Fact 对象当中, 然后将其插入到规则的 WorkingMemory 当中, 这样在规则当中就可以通过对 Fact 对象数据的读写, 从而实现对应用数据的读写操作。一个 Fact 对象通常是

一个具有 `getter` 和 `setter` 方法的 POJO 对象，通过这些 `getter` 和 `setter` 方法可以方便的实现对 `Fact` 对象的读写操作，所以我们可以简单的把 `Fact` 对象理解为规则与应用系统数据交互的桥梁或通道。

当 `Fact` 对象插入到 `WorkingMemory` 当中后，会与当前 `WorkingMemory` 当中所有的规则进行匹配，同时返回一个 `FactHandler` 对象。`FactHandler` 对象是插入到 `WorkingMemory` 当中 `Fact` 对象的引用句柄，通过 `FactHandler` 对象可以实现对对应的 `Fact` 对象的删除及修改等操作。

在前面介绍 `StatefulKnowledgeSession` 和 `StatelessKnowledgeSession` 两个对象的时候也提到了插入 `Fact` 对象的方法，在 `StatefulKnowledgeSession` 当中直接使用 `insert` 方法就可以将一个 `Java` 对象插入到 `WorkingMemory` 当中，如果有多个 `Fact` 需要插入，那么多个调用 `insert` 方法即可；对于 `StatelessKnowledgeSession` 对象可利用 `CommandFactory` 实现单个 `Fact` 对象或多个 `Fact` 对象的插入。

2. 规则

学习 Drools 规则语法的目的是为了在应用当中帮助我们解决实际的问题，所以学会并灵活的在规则当中使用就显的尤为重要。本章的内容包括规则的基本的约束部分语法讲解（LHS）、规则动作执行部分语法讲解及规则的各种属性介绍。

2.1. 规则文件

在 Drools 当中，一个标准的规则文件就是一个以 “.drl” 结尾的文本文件，由于它是一个标准的文本文件，所以可以通过一些记事本工具对其进行打开、查看和编辑。规则是放在规则文件当中的，一个规则文件可以存放多个规则，除此之外，在规则文件当中还可以存放用户自定义的函数、数据对象及自定义查询等相关在规则当中可能会用到的一些对象。

一个标准的规则文件的结构如代码清单 2-1 所示。

代码清单 2-1

```
package package-name

imports

globals

functions

queries

rules
```

对于一个规则文件而言，首先声明 **package** 是必须的，除 **package** 之外，其它对象在规则文件中的顺序是任意的，也就是说在规则文件当中必须要有一个 **package** 声明，同时 **package** 声明必须要放在规则文件的第一行。

规则文件当中的 **package** 和 Java 语言当中的 **package** 有相似之处，也有不同之处。在 Java 当中 **package** 的作用是用来对功能相似或相关的文件放在同一个 **package** 下进行管理，这种 **package** 管理既有物理上 Java 文件位置的管理也有逻辑上的文件位置的管理，在 Java 当中这种通过 **package** 管理文件要求在文件位置在逻辑上与物理上要保持一致；在 Drools 的规则

文件当中 **package** 对于规则文件中规则的管理只限于逻辑上的管理，而不管其在物理上的位置如何，这点是规则与 Java 文件的 **package** 的区别。

对于同一 **package** 下的用户自定义函数、自定义的查询等，不管这些函数与查询是否在同一个规则文件里面，在规则里面是可以直接使用的，这点和 Java 的同一 **package** 里的 Java 类调用是一样的。

2.2. 规则语言

规则是在规则文件当中编写，所以要编写一个规则首先需要先创建一个存放规则的规则文件。一个规则文件可以存放若干个规则，每一个规则通过规则名称来进行标识。代码清单 2-2 说明了一个标准规则的结构。

代码清单 2-2

```
rule "name"
  attributes
  when
    LHS
  then
    RHS
end
```

从代码清单 2-2 中可以看到，一个规则通常包括三个部分：属性部分（**attribute**）、条件部分（**LHS**）和结果部分（**RHS**）。对于一个完整的规则来说，这三个部分都是可选的，也就是说如代码清单 2-3 所示的规则是合法的。

代码清单 2-3

```
rule "name"
  when
  then
end
```

对于代码清单 2-3 所示的这种规则，因为其没有条件部分，默认它的条件部分是满足的，因为其结果执行部分为空，所以即使条件部分满足该规则也什么都不做。接下来我们就分别来对规则的条件部分、结果部分和属性部分进行介绍。

2.2.1. 条件部分

条件部分又被称之为 **Left Hand Side**，简称为 **LHS**，下文当中，如果没有特别指出，那么所说的 **LHS** 均指规则的条件部分，在一个规则当中 **when** 与 **then** 中间的部分就是 **LHS** 部

分。在 LHS 当中，可以包含 0~n 个条件，如果 LHS 部分没空的话，那么引擎会自动添加一个 eval(true)的条件，由于该条件总是返回 true，所以 LHS 为空的规则总是返回 true。所以代码清单 2-3 所示的规则在执行的时候引擎会修改成如代码清单 2-4 所示的内容。

代码清单 2-4

```
rule "name"
  when
    eval(true)
  then
  end
```

LHS 部分是由一个或多个条件组成，条件又称之为 pattern（匹配模式），多个 pattern 之间用可以使用 and 或 or 来进行连接，同时还可以使用小括号来确定 pattern 的优先级。

一个 pattern 的语法如代码清单 2-5 所示：

代码清单 2-5

```
[绑定变量名: ]Object([field 约束])
```

对于一个 pattern 来说“绑定变量名”是可选的，如果在当前规则的 LHS 部分的其它的 pattern 要用到这个对象，那么可以通过为该对象设定一个绑定变量名来实现对其引用，对于绑定变量的命名，通常的作法是为其添加一个“\$”符号作为前缀，这样可以很好的与 Fact 的属性区别开来；绑定变量不仅可以用在对象上，也可以用在对象的属性上面，命名方法与对象的命名方法相同；“field 约束”是指当前对象里相关字段的条件限制，代码清单 2-6 规则中 LHS 部分单个 pattern 的情形。

代码清单 2-6

```
rule "rule1"
  when
    $customer:Customer()
  then
    <action>...
  end
```

代码清单 2-6 规则中“\$customer”是就是一个绑定到 Customer 对象的“绑定变量名”，该规则的 LHS 部分表示，要求 Fact 对象必须是 Customer 类型，该条件满足了那么它的 LHS 会返回 true。

代码清单 2-7 演示了多个约束构成的 LHS 部分。

代码清单 2-7

```
rule "rule1"
  when
```

```

    $customer:Customer(age>20,gender==' male' )
    Order(customer==$customer,price>1000)
  then
    <action>...
end

```

代码清单 2-7 中的规则就包含两个 pattern，第一个 pattern 有三个约束，分别是：对象类型必须是 Customer；同时 Customer 的 age 要大于 20 且 gender 要是 male；第二个 pattern 也有三个约束，分别是：对象类型必须是 Order，同时 Order 对应的 Customer 必须是前面的那个 Customer 且当前这个 Order 的 price 要大于 1000。在这两个 pattern 没有符号连接，在 Drools 当中在 pattern 中没有连接符号，那么就用 and 来作为默认连接，所以在该规则的 LHS 部分中两个 pattern 只有都满足了才会返回 true。默认情况下，每行可以用“;”来作为结束符（和 Java 的结束一样），当然行尾也可以不加“;”结尾。

2.2.1.1. 约束连接

对于对象内部的多个约束的连接，可以采用“&&”（and）、“||”（or）和“,”（and）来实现，代码清单 2-7 中规则的 LHS 部分的两个 pattern 就里对象内部约束就采用“,”来实现，“&&”（and）、“||”（or）和“,”这三个连接符号如果没有用小括号来显示的定义优先级的话，那么它们的执行顺序是：“&&”（and）、“||”（or）和“,”。代码清单 2-8 演示了使用“&&”（and）、“||”（or）来连接约束的情形。

代码清单 2-8

```

rule "rule1"
  when
    Customer(age>20 || gender==' male' && city==' sh' )
  then
    <action>...
End

```

代码清单 2-8 中规则的 LHS 部分只有一个 pattern，在这个 pattern 中有四个约束，首先必须是一个 Customer 对象，然后要么该对象 gender='male' 且 city='sh'，要么 age>20，在 Customer 对象的字段约束当中，age>20 和 gender='male' 且 city='sh' 这两个有一个满足就可以了，这是因为“&&”连接符的优先级要高于“||”，所以代码清单 2-8 中规则的 LHS 部分 pattern 也可以写成代码清单 2-9 的样子，用一小括号括起来，这样看起来就更加直观了。

代码清单 2-9

```

rule "rule1"

```

```

when
    Customer(age>20 || (gender==' male' && city==' sh' ))
then
    <action>...
End

```

表面上看“,”与“&&”具有相同的含义,但是有一点需要注意,“,”与“&&”和“||”不能混合使用,也就是说在有“&&”或“||”出现的 LHS 当中,是不可以有“,”连接符出现的,反之亦然。

2.2.1.2. 比较操作符

在前面规则例子当中,我们已经接触到了诸如“>”、“=”之类的比较操作符,在 Drools5 当中共提供了十二种类型的比较操作符,分别是: >、>=、<、<=、==、!=、contains、not contains、memberof、not memberof、matches、not matches; 在这十二种类型的比较操作符当中,前六个是比较常见也是用的比较多的比较操作符,本小节当中将着重对后六种类型的比较操作符进行介绍。

2.2.1.2.1. contains

比较操作符 contains 是用来检查一个 Fact 对象的某个字段(该字段要是一个 Collection 或是一个 Array 类型的对象)是否包含一个指定的对象。代码清单 2-10 演示了 contains 比较操作符的用法。

代码清单 2-10

```

#created on: 2009-8-26
package test

rule "rule1"

    when
        $order:Order();
        $customer:Customer(age >20, orders contains $order);
    then
        System.out.println($customer.getName());
    end

```

contains 操作符的语法如下:

Object(field[Collection/Array] contains value)

`contains` 只能用于对象的某个 `Collection/Array` 类型的字段与另外一个值进行比较, 作为比较的值可以是一个静态的值, 也可以是一个变量(绑定变量或者是一个 `global` 对象), 在代码清单 2-10 当中比较值 `$order` 就是一个绑定变量。

2.2.1.2.2. not contains

`not contains` 作用与 `contains` 作用相反, `not contains` 是用来判断一个 `Fact` 对象的某个字段 (`Collection/Array` 类型) 是不是包含一个指定的对象, 和 `contains` 比较符相同, 它也只能用在对象的 `field` 当中, 代码清单 2-11 演示了 `not contains` 用法。

代码清单 2-11

```
#created on: 2009-8-26
package test

rule "rule1"

    when
        $order:Order(items not contains "手机");
    then
        System.out.println($order.getName());
    end
```

代码清单 2-11 的规则当中, 在判断订单 (`Order`) 的时候, 要求订单当中不能包含有“手机”的货物, 这里的规则对于比较的项就是一个表态固定字符串。

2.2.1.2.3. memberOf

`memberOf` 是用来判断某个 `Fact` 对象的某个字段是否在一个集合 (`Collection/Array`) 当中, 用法与 `contains` 有些类似, 但也有不同, `memberOf` 的语法如下:

Object(fieldName memberOf value[Collection/Array])

可以看到 `memberOf` 中集合类型的数据是作为被比较项的, 集合类型的数据对象位于 `memberOf` 操作符后面, 同时在使用 `memberOf` 比较操作符时被比较项一定要是一个变量(绑定变量或者是一个 `global` 对象), 而不能是一个静态值。代码清单 2-12 是一个演示 `memberOf` 使用的规则示例。

代码清单 2-12

```
#created on: 2009-8-26
package test

global String[] orderNames;

rule "rule1"

    when
        $order:Order(name memberOf orderNames);
    then
        System.out.println($order.getName());
    end
```

代码清单 2-12 中被比较对象是一个 String Array 类型的 global 对象。

2.2.1.2.4. not memberOf

该操作符与 memberOf 作用恰恰相反，是用来判断 Fact 对象当中某个字段值是不是中某个集合(Collection/Array)当中，同时被比较的集合对象只能是一个变量(绑定变量或 global 对象)，代码清单 2-13 演示了 not memberOf 的用法。

代码清单 2-13

```
#created on: 2009-8-26
package test
import java.util.List;
rule "rule1"
    when
        $orderList:String[]();
        $order:Order(name not memberOf $orderList);
    then
        System.out.println($order.getName());
    end
```

代码清单 2-13 中表示只有订单 (Order) 的名字 (name) 在订单集合 (\$orderList) 时 LHS 才能返回 true，该例子当中被比较的集合 \$orderList 是一个字符串数组的类型的绑定变量对象。

2.2.1.2.5. matches

matches 是用来对某个 Fact 的字段与标准的 Java 正则表达式进行相似匹配，被比较的字

字符串可以是一个标准的 Java 正则表达式，但有一点需要注意，那就是正则表达式字符串当中不用考虑 “\” 的转义问题。matches 使用语法如下：

Object(fieldName matches “正则表达式”)

代码清单 2-14 演示了 matches 的用法

代码清单 2-14

```
#created on: 2009-8-26
package test
import java.util.List;
rule "rule1"
    when
        $customer:Customer(name matches "李.*");
    then
        System.out.println($customer.getName());
end
```

在清单代码清单 2-14 中示例的规则就像我们展示了 matches 的用法，该规则是用来查找所有 Customer 对象的 name 属性是不是以“李”字开头，如果满足这一条件那么就将该 Customer 对象的 name 属性打印出来，代码清单 2-15 是对该规则的测试类源码。

代码清单 2-15

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
        kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
        ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
        knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
```

```

        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        Customer cus1=new Customer();
        cus1.setName("张三");
        Customer cus2=new Customer();
        cus2.setName("李四");
        Customer cus3=new Customer();
        cus3.setName("王二");
        Customer cus4=new Customer();
        cus4.setName("李小龙");

        statefulSession.insert(cus1);
        statefulSession.insert(cus2);
        statefulSession.insert(cus3);
        statefulSession.insert(cus4);

        statefulSession.fireAllRules();
        statefulSession.dispose();
        System.out.println("end.....");
    }
}

```

测试类运行结果如图 2-1 所示。

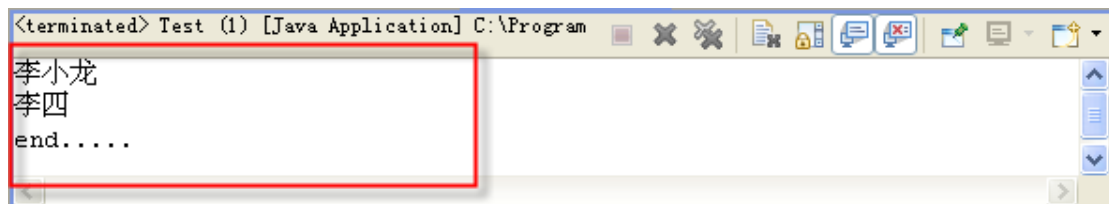


图 2-1

从测试结果中可以看到，该规则运用 `matches` 操作符已经将所有以“李”字开头的 `Customer` 对象找到并打印出来。

2.2.1.2.6. not matches

与 `matches` 作用相反，是用来将某个 `Fact` 的字段与一个 `Java` 标准正则表达式进行匹配，看是不是能与正则表达式匹配。`not matches` 使用语法如下：

```
Object(fieldname not matches “正则表达式”)
```

因为 `not matches` 用法与 `matches` 一样，只是作用相反，所以这里就不再举例了，有兴趣

趣的读者可以用 `not matches` 写个例子测试一下。

2.2.2. 结果部分

条件部分又被称之为 **Right Hand Side**，简称为 **RHS**，在一个规则当中 **then** 后面部分就是 **RHS**，只有在 **LHS** 的所有条件都满足时 **RHS** 部分才会执行。

RHS 部分是规则真正要做事情的部分，可以将因条件满足而要触发的动作写在该部分当中，在 **RHS** 当中可以使用 **LHS** 部分当中定义的绑定变量名、设置的全局变量、或者是直接编写 **Java** 代码（对于要用到的 **Java** 类，需要在规则文件当中用 `import` 将类导入后方能使用，这点和 **Java** 文件的编写规则相同）。

我们知道，在规则当中 **LHS** 就是用来放置条件的，所以在 **RHS** 当中虽然可以直接编写 **Java** 代码，但不建议在代码当中有条件判断，如果需要条件判断，那么请重新考虑将其放在 **LHS** 当中，否则就违背了使用规则的初衷。

在 **Drools** 当中，在 **RHS** 里面，提供了一些对当前 **Working Memory** 实现快速操作的宏函数或对象，比如 `insert/insertLogical`、`update` 和 `retract` 就可以实现对当前 **Working Memory** 中的 **Fact** 对象进行新增、删除或者是修改；如果您觉得还要使用 **Drools** 当中提供的其它方法，那么您还可以使用另一外宏对象 `drools`，通过该对象可以使用更多的操作当前 **Working Memory** 的方法；同时 **Drools** 还提供了一个名为 `kcontext` 的宏对象，使我们可以通过该对象直接访问当前 **Working Memory** 的 **KnowledgeRuntime**。下面我们就来详细讨论一下这些宏函数的用法。

2.2.2.1. insert

函数 `insert` 的作用与我们在 **Java** 类当中调用 **StatefulKnowledgeSession** 对象的 `insert` 方法的作用相同，都是用来将一个 **Fact** 对象插入到当前的 **Working Memory** 当中。它的基本用法格式如下：

```
insert(new Object());
```

一旦调用 `insert` 宏函数，那么 **Drools** 会重新与所有的规则再重新匹配一次，对于没有设置 `no-loop` 属性为 `true` 的规则，如果条件满足，不管其之前是否执行过都会再执行一次，这个特性不仅存在于 `insert` 宏函数上，后面介绍的 `update`、`retract` 宏函数同样具有该特性，所

以在某些情况下因考虑不周调用 insert、update 或 retract 容易发生死循环，这点大家需要注意。

代码清单 2-16 演示了 insert 函数的用法。

代码清单 2-16

```
#created on: 2009-8-26
package test
import java.util.List;
rule "rule1"
    salience 1
    when
        eval(true);
    then
        Customer cus=new Customer();
        cus.setName("张三");
        insert(cus);
    end
rule "rule2"
    salience 2
    when
        $customer:Customer(name == "张三");
    then
        System.out.println("rule2----"+$customer.getName());
    end
end
```

代码清单 2-16 中有两个规则 rule1 和 rule2，在 rule1 这个规则当中，LHS 是 eval(true)，表示没有任何条件限制（关于 eval 的用法，在后续的内容当中会有详细的介绍），RHS 当中创建了一个 Customer 对象，并设置它的 name 属性为“张三”，完成后调用 insert 宏函数将其插入到当前的 Working Memory 当中；在名为 rule2 这个规则当中，首先判断当前的 Working Memory 当中没有一个 name 属性的值为“张三”的 Customer 对象，如果有的话，那么就在其 RHS 部分将这个 Customer 对象的 name 属性打印出来。

很明显我们希望 rule1 这个规则先执行，rule2 这个规则后执行，为了达成这个希望，我们为这两个规则添加了一个名为“salience”的属性，该属性的作用是通过一个数字来确认规则执行的优先级，数字越大，执行越靠前。这里为 rule1 规则设置它的 salience 属性值为 2，rule2 为 1 那么就表示 rule1 会先执行，rule2 会后执行。实际上，我们前面讲到，因为一旦调用 insert、update 或 retract 函数，Drools 会重新与所有的规则再重新匹配一次，对于没有设置 no-loop 属性为 true 的规则，如果条件满足，不管其之前是否执行过都会再执行一次。所以这里的优先级如果设置也可以达到相同的效果。

编写测试类，测试这两个规则的执行情况，测试类源码如代码清单 2-17 所示。

代码清单 2-17

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        statefulSession.fireAllRules();
        statefulSession.dispose();
        System.out.println("end.....");
    }
}
```

运行之后，结果如图 2-3 所示。



从执行的结果可以看出，正如我们前面论述的那样，rule1 先执行，向 Working Memory 当中插入一个 name 属性为“张三”的 Customer 对象，然后 rule2 再执行，将该对象的 name 属性在控制台打印出来。

2.2.2.2. insertLogical

insertLogical 作用与 insert 类似，它的作用也是将一个 Fact 对象插入到当前的 Working Memory 当中，

2.2.2.3. update

update 函数意义与其名称一样，用来实现对当前 Working Memory 当中的 Fact 进行更新，update 宏函数的作用与 StatefulSession 对象的 update 方法的作用基本相同，都是用来告诉当前的 Working Memory 该 Fact 对象已经发生了变化。它的用法有两种形式，一种是直接更新一个 Fact 对象，另一种为通过指定 FactHandle 来更新与指定 FactHandle 对应的 Fact 对象，下面我们就来通过两个实例来说明 update 的这两种用法。

先来看第一种用法，直接更新一个 Fact 对象。第一种用法的格式如下：

```
update(new Object());
```

示例规则如代码清单 2-18 所示。

代码清单 2-18

```
#created on: 2009-8-26
package test
import java.util.List;

query "query fact count"
    Customer();
end

rule "rule1"
```

```

salience 2
when
    eval(true);
then
    Customer cus=new Customer();
    cus.setName("张三");
    cus.setAge(1);
    insert(cus);
end
rule "rule2"
    salience 1
    when
        $customer:Customer(name=="张三",age<10);
    then
        $customer.setAge($customer.getAge()+1);
        update($customer);
        System.out.println("-----"+$customer.getName());
    end
end

```

在代码清单 2-18 当中，有两个规则：rule1 和 rule2，在 rule1 当中我们通过使用 insert 宏函数向当前 Working Memory 当中插入了一个 name 属性为“张三”、age 属性为“1”的 Customer 对象；rule2 的规则当中首先判断当前的 Working Memory 当中有没有一个 name 属性为“张三”同时 age 属性值小于“10”的 Customer，如果有的话，那么就重新设置当前这个 Customer 对象的 age 属性的值：将当前 age 属性的值加 1，然后调用 update 宏函数，对这个修改后的 Customer 对象进行更新。

因为 rule1 的优先级为 2，所以它会先执行；rule2 的优先级为 1，所以它会在 rule1 执行完成后执行。在第一次 rule1 规则执行完成后，Working Memory 当中就会有一个 name 属性为“张三”、age 属性为“1”的 Customer 对象，正好满足 rule2 规则的条件，所以 rule2 的 RHS 部分会执行，在 rule2 当中一旦使用 update 宏函数对 Customer 对象进行了更新，Drools 会重新检查所有的规则，看看有没有条件满足，这时只要 Customer 对象的 age 属性值在没有更新的 10 之前 rule2 都会再次触发，直到 Customer 对象的 age 属性值更新到大于等于 10，这时所有的规则执行才算完成。

为了测试在多次调用 update 宏函数更新 Customer 对象后 Working Memory 当中还只存在一个 Customer 对象，所以我们还添加了一个名为“query fact count”的 query 查询（关于 query 查询后面的章节会有详细介绍），该查询的作用是用来检索当中 Working Memory 当中有多少个 Working Memory 对象，在该示例当中，Customer 对象应该只有一个。

编写测试类，测试这两个规则执行是不是符合我们的期望。测试类代码如代码清单 2-19 所示。

代码清单 2-19

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.QueryResults;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        statefulSession.fireAllRules();
        statefulSession.dispose();
        QueryResults qr=statefulSession.getQueryResults("query fact
count");
        System.out.println("customer 对象数目: "+qr.size());
        System.out.println("end.....");
    }
}
```

运行测试类，结果如图 2-2 所示。

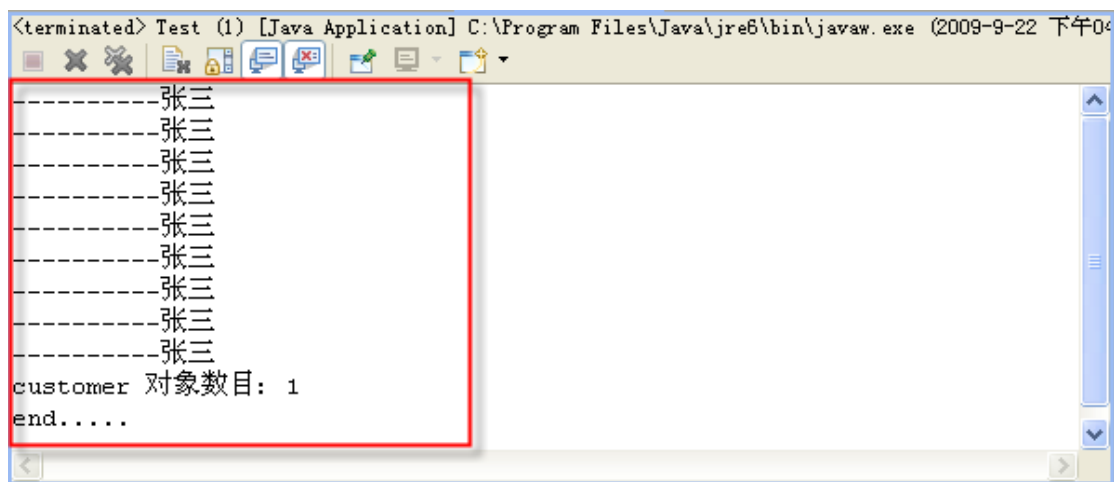


图 2-2

从运行结果图中可以看出，Customer 对象的 name 属性共输出了 9 次，同时最后检索出来的 Customer 对象的结果也只是一个，符合我们的预期。

在上面的这个例子当中，一旦使用 **update** 宏函数，那么符合条件的规则又会重新触发，不管该规则是否执行过，如果您希望规则只执行一次，那么可以通过设置规则的 **no-loop** 属性为 **true** 来实现，如上面示例当中的 **rule2** 规则，如果添加 **no-loop** 属性为 **true**，那么 Customer 的 name 属性将只会输出一次。添加 **no-loop** 属性后的 **rule2** 规则如代码清单 2-20 所示。

代码清单 2-20

```
rule "rule2"
    salience 1
    no-loop true
    when
        $customer:Customer(name=="张三",age<10);
    then
        $customer.setAge($customer.getAge()+1);
        update($customer);
        System.out.println("-----"+$customer.getName());
    end
```

再次运行测试类，可以得到如图 2-3 所示的结果。

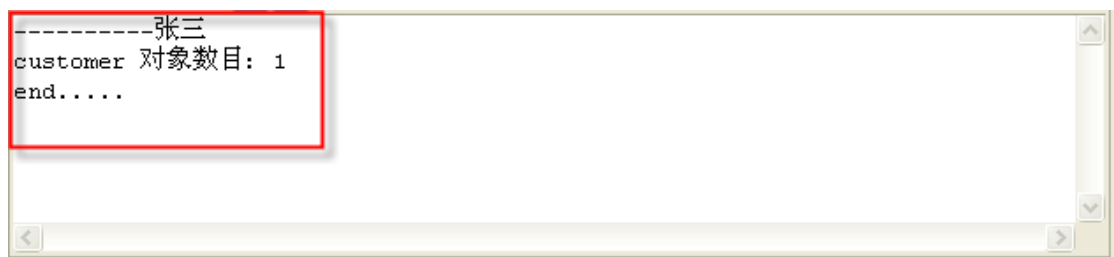


图 2-3

可以看到，由于添加了 **no-loop** 属性为 **true**，**rule2** 规则只执行了一次。下面我们来看

一下 update 宏函数的第二种用法。

第二种用法的格式如下：

```
update(new FactHandle(),new Object());
```

从第二种用法格式上可以看出，它可以支持创建一个新的 Fact 对象，从而把 FactHandle 对象指定的 Fact 对象替换掉，从而实现对象的全新更新，代码清单 2-21 里的规则演示这种用法。

代码清单 2-21

```
#created on: 2009-8-26
package test
import java.util.List;

query "query fact count"
    Customer();
end

rule "rule1"
    salience 2
    when
        eval(true);
    then
        Customer cus=new Customer();
        cus.setName("张三");
        cus.setAge(1);
        insert(cus);
    end
rule "rule2"
    salience 1
    when
        $customer:Customer(name=="张三",age<10);
    then
        Customer customer=new Customer();
        customer.setName("张三");
        customer.setAge($customer.getAge()+1);

        update(drools.getWorkingMemory().getFactHandleByIdentity($customer),customer);
        System.out.println("-----"+$customer.getName());
    end
```

和前面的规则相比，更改了 rule2 的 RHS 部分，在这个部分当中，创建了一个新的 Customer 对象，并设置了它的 name 属性为“张三”、age 属性为当前 Working Memory 当中

的 Customer 对象的 age 属性值加 1，设置完成后将这个新的 Customer 对象通过使用 update 方法替换了原来的在 Working Memory 当中的 Customer 对象。编写测试类，可以发现运行结果同前面的结果相同。

在这个规则当中我们使用了一个名为 drools 的宏对象，通过该对象获取当前的 Working Memory 对象，再通过 WorkingMemory 得到指定的 Fact 对象的 FactHandle。关于 drools 宏函数后面会有详细介绍。

2.2.2.4. retract

和 StatefulSession 的 retract 方法一样，宏函数 retract 也是用来将 Working Memory 当中某个 Fact 对象从 Working Memory 当中删除，下面就通过一个例子来说明 retract 宏函数的用法。代码清单 2-22 为示例规则。

代码清单 2-22

```
#created on: 2009-8-26
package test
import java.util.List;

query "query fact count"
    Customer();
end

rule "rule1"
    salience 2
    when
        eval(true);
    then
        Customer cus=new Customer();
        cus.setName("张三");
        cus.setAge(1);
        insert(cus);
    end
rule "rule2"
    salience 1
    when
        $customer:Customer(name=="张三");
    then
        retract($customer);
    end
```

代码清单 2-22 中有两个规则和前面的示例基本相同，在 rule2 当中 RHS 部分，通过使

用 `retract` 宏函数对符合条件的 `Customer` 对象进行了删除，将其从当前的 `Working Memory` 当中清除，这样在执行完所有的规则之后，调用名为“`query fact count`”的 `query` 查询，查询到的结果数量应该是 0，编写测试类，验证我们推理的结果。测试类代码如代码清单 2-23 所示。

代码清单 2-23

```
package test;

import java.util.Collection;


import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.QueryResults;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        statefulSession.fireAllRules();
        statefulSession.dispose();
        QueryResults qr=statefulSession.getQueryResults("query fact
count");
        System.out.println("customer 对象数目: "+qr.size());
        System.out.println("end.....");
    }
}
```

运行测试类，可以看到如图 2-4 所示的结果。



```
customer 对象数目: 0  
end.....
```

正如推理的那样，因为使用了 `retract` 将 `Customer` 对象从当前的 `Working Memory` 当中删除，所以运行结果中看到的 `Customer` 对象的数目为 0。

2.2.2.5. drools

如果您希望在规则文件里更多的实现对当前的 `Working Memory` 控制，那么可以使用 `drools` 宏对象实现，通过使用 `drools` 宏对象可以实现在规则文件里直接访问 `Working Memory`。在前面介绍 `update` 宏函数的时候我们就使用 `drools` 宏对象来访问当前的 `Working Memory`，得到一个指定的 `Fact` 对象的 `FactHandle`。同时前面介绍的 `insert`、`insertLogical`、`update` 和 `retract` 宏函数的功能皆可以通过使用 `drools` 宏对象来实现。代码清单 2-24 和代码清单 2-25 所示的两个规则在功能上是完全一样的。

代码清单 2-24

```
rule "rule1"  
  salience 11  
  when  
    eval(true);  
  then  
    Customer cus=new Customer();  
    cus.setName("张三");  
    insert(cus);  
end
```

代码清单 2-25

```
rule "rule1"  
  salience 11  
  when  
    eval(true);  
  then  
    Customer cus=new Customer();  
    cus.setName("张三");  
    drools.insert(cus)
```

end

代码清单 2-24 中向当前的 Working Memory 当中插入一个 Customer 对象采用的是宏函数 insert 实现的，代码清单 2-25 则是采用宏对象 drools 的 insert 方法来实现，两个规则向当前 Working Memory 当中插入对象的方法不同，但最终实现的功能是一样的。

使用 drools 宏对象，可以得到很多操纵 Working Memory 的方法，如果想查看 drools 宏对象具有哪些方法可以使用，可以通过按“ALT”+“/”这两个功能键实现。具体做法是先输入“drools.”然后按“ALT”+“/”这两个功能键就可以看到 drools 宏函数的方法列表，如果在操作过程中，您没有看到方法列表，那应该是您操作系统里的“ALT”+“/”这两个功能键组合被其它软件占用。

表 2-1 中罗列了 drools 宏对象的常用方法。

表格 2-1

方法名称	用法格式	含义
getWorkingMemory()	drools.getWorkingMemory()	获取当前的 WorkingMemory 对象
halt()	drools.halt()	在当前规则执行完成后，不再执行其它未执行的规则。
getRule()	drools.getRule()	得到当前的规则对象
insert(new Object)	drools.insert(new Object)	向当前的 WorkingMemory 当中插入指定的对象，功能与宏函数 insert 相同。
update(new Object)	drools.update(new Object)	更新当前的 WorkingMemory 中指定的对象，功能与宏函数 update 相同。
update(FactHandle Object)	drools.update(FactHandle Object)	更新当前的 WorkingMemory 中指定的对象，功能与宏函数 update 相同。
retract(new Object)	drools.retract(new Object)	从当前的 WorkingMemory 中删除指定的对象，功能与宏函数 retract 相同。

2.2.2.6. kcontext

kcontext 也是 Drools 提供的一个宏对象，它的作用主要是用来得到当前的 KnowledgeRuntime 对象，KnowledgeRuntime 对象可以实现与引擎的各种交互，关于 KnowledgeRuntime 对象的介绍，您可以参考后面的章节内容。

2.2.2.7. modify

modify 是一个表达式块，它可以快速实现对 Fact 对象多个属性进行修改，修改完成后会自动更新到当前的 Working Memory 当中。它的基本语法格式如下：

```
modify(fact-expression){  
    <修改 Fact 属性的表达式>[,<修改 Fact 属性的表达式>*]  
}
```

下面我们通过一个实例来说明 modify 表达式块的用法。

代码清单 2-26 中规则 rule1 里用使用 modify 表达式块来对 Customer 对象的属性进行了修改。

代码清单 2-26

```
#created on: 2009-8-26  
package test  
import java.util.List;  
rule "rule1"  
    salience 2  
    when  
        $customer:Customer(name=="张三",age==20);  
    then  
        System.out.println("modify before customer  
id:"+$customer.getId()+"age:"+$customer.getAge());  
        modify($customer){  
            setId("super man"),  
            setAge(30)  
        }  
    end  
rule "rule2"  
    salience 1  
    when
```

```
$customer:Customer(name=="张三");  
then  
    System.out.println("modify after customer  
id:"+$customer.getId()+" ;age:"+$customer.getAge());  
end
```

在这两个规则当中，我们通过使用 `salience` 属性来控制他们的执行顺序，让 `rule1` 先执行，执行时先打印出当前 `Customer` 对象的 `id` 与 `age` 属性的值，然后利用 `modify` 块对 `Customer` 对象的这两个属性进行修改，接下来再执行 `rule2`，再次打印 `Customer` 对象的 `id` 与 `age` 属性的值，这时的值应该是 `modify` 块修改后的值。编写测试类，测试类代码如代码清单 2-27 所示。

代码清单 2-27

```
package test;  
  
import java.util.Collection;  
  
import org.drools.KnowledgeBase;  
import org.drools.KnowledgeBuilderFactory;  
import org.drools.builder.KnowledgeBuilder;  
import org.drools.builder.KnowledgeBuilderFactory;  
import org.drools.builder.ResourceType;  
import org.drools.io.impl.ClassPathResource;  
import org.drools.runtime.StatefulKnowledgeSession;  
  
public class Test {  
  
    public static void main(String[] args) {  
        KnowledgeBuilder  
kb=KnowledgeBuilderFactory.newKnowledgeBuilder();  
        kb.add(new ClassPathResource("test/test.drl"),  
ResourceType.DRL);  
        Collection collection=kb.getKnowledgePackages();  
        KnowledgeBase  
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();  
        knowledgeBase.addKnowledgePackages(collection);  
        StatefulKnowledgeSession  
statefulSession=knowledgeBase.newStatefulKnowledgeSession();  
  
        Customer cus=new Customer();  
        cus.setAge(20);  
        cus.setId("ZhangeShan");  
        cus.setName("张三");  
        statefulSession.insert(cus);  
    }  
}
```



```

        statefulSession.fireAllRules();
        statefulSession.dispose();
        System.out.println("end.....");
    }
}

```

运行测试类，可以看到如图 2-所示的结果。

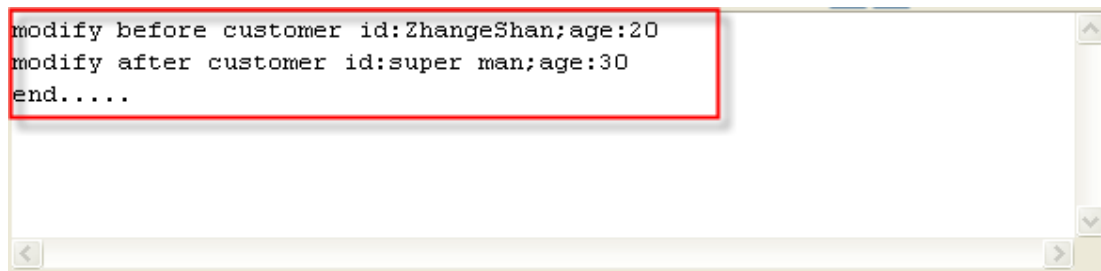


图 2-4

这里有一点需要注意，那就是和 insert、update、retract 对 Working Memory 的操作一样，一旦使用了 modify 块对某个 Fact 的属性进行了修改，那么会导致引擎重新检查所有规则是否匹配条件，而不管其之前是否执行过，所以这个例子当中，在 rule1 这个规则当中，LHS 条件约束部分我们添加了两个条件 name == “张三”和 age == 20，如果去掉 age == 20 那么就会形成死循环，至于原因应该是比较容易理解的。

2.2.3. 属性部分

规则属性是用来控制规则执行的重要工具，在前面举出的关于规则的例子当中，已经接触了如控制规则执行优先级的 salience，和是否允许规则执行一次的 no-loop 等。在目前的 Drools5 当中，规则的属性共有 13 个，它们分别是：activation-group、agenda-group、auto-focus、date-effective、date-expires、dialect、duration、enabled、lock-on-active、no-loop、ruleflow-group、salience、when，这些属性分别适用于不同的场景，下面我们就来分别介绍这些属性的含义及用法。

2.2.3.1. salience

该属性的作用我们在前面的内容也有涉及，它的作用是用来设置规则执行的优先级，salience 属性的值是一个数字，数字越大执行优先级越高，同时它的值可以是一个负数。默认情况下，规则的 salience 默认值为 0，所以如果我们不手动设置规则的 salience 属性，那

么它的执行顺序是随机的。

代码清单 2-28

```
package test
rule "rule1"
    salience 1
    when
        eval(true)
    then
        System.out.println("rule1");
    end

rule "rule2"
    salience 2
    when
        eval(true)
    then
        System.out.println("rule2");
    end
```

在代码清单 2-28 中两个规则，虽然 rule1 位于前面，但因为它的 salience 为 1，而 rule2 的 salience 属性为 2，所以 rule2 会先执行，然后 rule1 才会执行。

2.2.3.2. no-loop

在一个规则当中如果条件满足就对 Working Memory 当中的某个 Fact 对象进行了修改，比如使用 update 将其更新到当前的 Working Memory 当中，这时引擎会再次检查所有的规则是否满足条件，如果满足会再次执行，代码清单 2-29 中的规则演示了这种用法。

代码清单 2-29

```
package test
rule "rule1"
    salience 1
    when
        $customer:Customer(name=="张三")
    then
        update($customer);
        System.out.println("customer name:"+$customer.getName());
    end
```

编写测试类，测试类代码如代码清单 2-30 所示。

代码清单 2-30

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {

    public static void main(String[] args) {
        KnowledgeBuilder
kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        Customer cus=new Customer();
        cus.setName("张三");
        statefulSession.insert(cus);

        statefulSession.fireAllRules();
        statefulSession.dispose();
        System.out.println("end....");
    }
}
```

运行测试代码，可以看到，控制台会不断输出打印 Custom 的 name 属性值的信息，而且永无止境，很明显这是一个死循环。造成这个死循环的原因就是在代码清单 2-29 中规则的 RHS 中使用了 update 宏函数，对当前的 Customer 的 Fact 对象进行更新操作，这样就导致引擎再次重新检查所有的规则是否符合条件，如果符合条件，那么就继续执行，那么再次对 Customer 进行更新.....。

如何避免这种情况呢，这时可以引入 `no-loop` 属性来解决这个问题。`no-loop` 属性的作用是用来控制已经执行过的规则在条件再次满足时是否再次执行，前面的示例当中也用到该属性帮我们解决应该当中的问题。`no-loop` 属性的值是一个布尔型，默认情况下规则的 `no-loop` 属性的值为 `false`，如果 `no-loop` 属性值为 `true`，那么就表示该规则只会被引擎检查一次，如果满足条件就执行规则的 RHS 部分，如果引擎内部因为对 Fact 更新引起引擎再次启动检查规则，那么它会忽略掉所有的 `no-loop` 属性设置为 `true` 的规则。

现在对规则进行修改，为其添加一个 `no-loop` 属性，属性的值为 `true`，修改后的规则如代码清单 2-31 所示。

代码清单 2-31

```
package test
rule "rule1"
    salience 1
    no-loop true
    when
        $customer:Customer(name=="张三")
    then
        update($customer);
        System.out.println("customer name:"+$customer.getName());
    End
```

再次运行测试类，可以看到控制台只会打印一次 Custom 的 name 属性值的信息。

2.2.3.3. date-effective

该属性是用来控制规则只有在到达后才会触发，在规则运行时，引擎会自动拿当前操作系统的时候与 `date-effective` 设置的时间值进行比对，只有当系统时间 \geq `date-effective` 设置的时间值时，规则才会触发执行，否则执行将不执行。在没有设置该属性的情况下，规则随时可以触发，没有这种限制。

`date-effective` 的值为一个日期型的字符串，默认情况下，`date-effective` 可接受的日期格式为“dd-MMM-yyyy”，例如 2009 年 9 月 25 日在设置为 `date-effective` 的值时，如果您的操作系统为英文的，那么应该写成“25-Sep-2009”；如果是英文操作系统“25-九月-2009”，那么就是，代码清单 2-32 就演示了 `date-effective` 属性的用法。

代码清单 2-32

```
package test
```

```
rule "rule1"
    date-effective " 25-九月-2009"
    when
        eval(true);
    then
        System.out.println("rule1 is execution!");
end
```

该规则里的 date-effective 的属性值为“09-九月-2009”，因为我的操作系统是中文，所以这里直接用的汉字来表示九月。

在实际使用的过程当中，如果您不想用这种时间的格式，那么可以在调用的 Java 代码中通过使用 System.setProperty(String key,String value)方法来修改默认的时间格式，如代码清单 2-33 的规则就是采用格式化后的日期。

代码清单 2-33

```
package test
rule "rule1"
    date-effective "2009-09-25"
    when
        eval(true);
    then
        System.out.println("rule1 is execution!");
end
```

在编写测试代码时，需要调用 System.setProperty(String key,String value)方法来修改默认的时间格式，调用的测试类如代码清单 2-34 所示。

代码清单 2-34

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        System.setProperty("drools.dateformat","yyyy-MM-dd");
        kb.add(new ClassPathResource("test/test.drl"),
```

```

ResourceType.DRL);
    Collection collection=kb.getKnowledgePackages();

    KnowledgeBase
knowledgeBase=KnowledgeBaseFactory.newKnowledgeBase();
    knowledgeBase.addKnowledgePackages(collection);
    StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

    statefulSession.fireAllRules();
    statefulSession.dispose();
    System.out.println("end....");
}
}

```

测试代码中`System.setProperty("drools.dateformat","yyyy-MM-dd");`这句就是用来修改当前系统默认的时间格式的。在进行这个操作的时候有两个地方需要注意：一是设置的key必须是“drools.dateformat”，值的话遵循标准的Java日期格式；二是修改当前系统默认的时间格式的这句代码必须放在向KnowledgeBuilder里添加规则文件之前，否则将不起作用。

2.2.3.4. date-expires

该属性的作用与 date-effective 属性恰恰相反， date-expires 的作用是用来设置规则的有效期，引擎在执行规则的时候，会检查规则有没有 date-expires 属性，如果有的话，那么会将这个属性的值与当前系统时间进行比对，如果大于系统时间，那么规则就执行，否则就不执行。该属性的值同样也是一个日期类型，默认格式也是“dd-MMM-yyyy”，具体用法与 date-effective 属性相同。

代码清单 2-35 演示了 date-expires 属性的用法。

代码清单 2-35

```

package test
rule "rule1"
    date-expires "2009-09-27"
    when
        eval(true);
    then
        System.out.println("rule1 is execution!");
    end

```

在代码清单 2-35 中，名为 rule1 的规则的有效期为“2009-09-27”，当然该值遵循的是“yyyy-MM-dd”格式，所以在执行该规则的时候需要设置 drools.dateformat 环境变量的值。

2.2.3.5. enabled

`enabled` 属性比较简单，它是用来定义一个规则是否可用的。该属性的值是一个布尔值，默认该属性的值为 `true`，表示规则是可用的，如果手工为一个规则添加一个 `enabled` 属性，并且设置其 `enabled` 属性值为 `false`，那么引擎就不会执行该规则。

2.2.3.6. dialect

该属性用来定义规则当中要使用的语言类型，目前 Drools5 版本当中支持两种类型的语言：`mvel` 和 `java`，默认情况下，如果没有手工设置规则的 `dialect`，那么使用的 `java` 语言。

代码清单 2-36 演示了如何查看当前规则的 `dialect` 的值。

代码清单 2-36

```
package test
rule "rule1"
  when
    eval(true)
  then
    System.out.println("dialect:"+drools.getRule().getDialect());
  end
```

代码清单 2-36 中名为 `rule1` 的规则 RHS 当中利用规则当中内置的宏对象 `drools` 得到当前的 `Rule` 对象，再通过这个 `Rule` 对象得到其 `dialect` 属性的值。

执行该规则我们可以看如图 2-5 所示的结果。

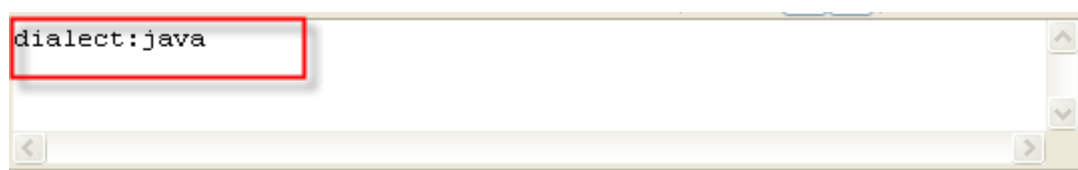


图 2-5

从执行输出的结果当中可以看出，规则默认情况下使用的 `dialect` 值为 “`java`”。

我们知道，规则是存放于规则文件当中，在规则文件当中也可以定义一个 `dialect` 属性，如果一个规则没有定义 `dialect` 属性，那么它就采用规则文件里设置的 `dialect` 属性的值。规则文件里 `dialect` 属性默认值为 “`java`”，所以规则中默认 `dialect` 属性的值也为 “`java`”。`java` 语言的语法大家都比较清楚，这里就简单的介绍一下 `mvel` 的语法。

`mvel` 在本书当中有专门的章节加一介绍，

2.2.3.7. duration

对于一个规则来说，如果设置了该属性，那么规则将在该属性指定的值之后在另外一个线程里触发。该属性对应的值为一个长整型，单位是毫秒，代码清单 2-37 里的规则 rule1 添加了 duration 属性，它的值为 3000，表示该规则将在 3000 毫秒之后在另外一个线程里触发。

代码清单 2-37

```
package test
rule "rule1"
    duration 3000
    when
        eval(true)
    then
        System.out.println("rule thread
id:"+Thread.currentThread().getId());
end
```

对应的测试 Java 类源码如代码清单 2-38 所示。

代码清单 2-38

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb=KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"), ResourceType.DRL);
        Collection collection=kb.getKnowledgePackages();
        KnowledgeBase
knowledgeBase=KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession
statefulSession=knowledgeBase.newStatefulKnowledgeSession();

        statefulSession.fireAllRules();
    }
}
```



```

        statefulSession.dispose();
        System.out.println("current thread
id:"+Thread.currentThread().getId());

    }
}

```

运行测试类，可以得到如图 2-6 所示的结果。



```

current thread id:1
rule thread id:8

```

图 2-6

从运行结果可以看到，执行规则和线程与测试类运行的线程的 ID 不同，表示规则不是在与测试类的线程里运行的，同时，又因为规则执行产生了一个新的线程没有结束，所以可以看到测试类的执行时一直处于未结束状态，需要手动结束主线程才行。

2.2.3.8. lock-on-active

当在规则上使用 `ruleflow-group` 属性或 `agenda-group` 属性的时候，将 `lock-on-action` 属性的值设置为 `true`，可能避免因某些 `Fact` 对象被修改而使已经执行过的规则再次被激活执行。

可以看出该属性与 `no-loop` 属性有相似之处，`no-loop` 属性是为了避免 `Fact` 修改或调用了 `insert`、`retract`、`update` 之类而导致规则再次激活执行，这里的 `lock-on-action` 属性也是起这个作用，`lock-on-active` 是 `no-loop` 的增强版属性，它主要作用在使用 `ruleflow-group` 属性或 `agenda-group` 属性的时候。`lock-on-active` 属性默认值为 `false`。关于该属性的介绍，我们在后面讲解规则流的时候还涉及到。

2.2.3.9. activation-group

该属性的作用是将若干个规则划分成一个组，用一个字符串来给这个组命名，这样在执行的时候，具有相同 `activation-group` 属性的规则中只要有一个会被执行，其它的规则都将不再执行。也就是说，在一组具有相同 `activation-group` 属性的规则当中，只有一个规则会被执行，其它规则都将不会被执行。当然对于具有相同 `activation-group` 属性的规则当中究竟哪一个会先执行，则可以用类似 `salience` 之类属性来实现。代码清单 2-39 就演示了

activation-group 属性的用法。

代码清单 2-39

```
package test
rule "rule1"
    activation-group "test"
    when
        eval(true)
    then
        System.out.println("rule1 execute");
    end

rule "rule 2"
    activation-group "test"
    when
        eval(true)
    then
        System.out.println("rule2 execute");
    end
```

代码清单 2-39 中有两个规则：rule1 和 rule2，这两个规则具有相同的 activation-group 属性，这个属性的值为“test”，前面我们讲过，具有相同 activation-group 属性的规则只会有一个被执行，其它规则将会被忽略掉，所以这里的 rule1 和 rule2 这两个规则因为具体相同名称的 activation-group 属性，所以它们只有一个会被执行。

2.2.3.10. agenda-group

规则的调用与执行是通过 StatelessSession 或 StatefulSession 来实现的，一般的顺序是创建一个 StatelessSession 或 StatefulSession，将各种经过编译的规则 package 添加到 session 当中，接下来将规则当中可能用到的 Global 对象和 Fact 对象插入到 Session 当中，最后调用 fireAllRules 方法来触发、执行规则。在没有调用最后一步 fireAllRules 方法之前，所有的规则及插入的 Fact 对象都存放在一个名叫 Agenda 表的对象当中，这个 Agenda 表中每一个规则及与其匹配相关业务数据叫做 Activation，在调用 fireAllRules 方法后，这些 Activation 会依次执行，这些位于 Agenda 表中的 Activation 的执行顺序在没有设置相关用来控制顺序的属性时（比如 salience 属性），它的执行顺序是随机的，不确定的。

Agenda Group 是用来在 Agenda 的基础之上，对现在的规则进行再次分组，具体的分组方法可以采用为规则添加 agenda-group 属性来实现。

agenda-group 属性的值也是一个字符串，通过这个字符串，可以将规则分为若干个 Agenda Group，默认情况下，引擎在调用这些设置了 agenda-group 属性的规则的时候需要显示的指定某个 Agenda Group 得到 Focus（焦点），这样位于该 Agenda Group 当中的规则才会触发执行，否则将不执行。

代码清单 2-40 当中有两个规则 rule1 和 rule2，rule1 的 agenda-group 属性值为 001；rule2 的 agenda-group 属性值为 002，这就表示 rule1 和 rule2 这两个规则分别被划分到名为 001 和 002 的两个 Agenda Group 当中，这样引擎执行这两个规则的时候必须显示的设置名为 001 或 002 的 Agenda Group 得到焦点，这样位于 001 或 002 的 Agenda Group 里的规则才会触发执行。

代码清单 2-40

```
package test
rule "rule1"
    agenda-group "001"
    when
        eval(true)
    then
        System.out.println("rule1 execute");
    end

rule "rule 2"
    agenda-group "002"
    when
        eval(true)
    then
        System.out.println("rule2 execute");
    end
```

为了测试这两个规则的执行情况，编写测试类，测试类代码如代码清单 2-41 所示。

代码清单 2-41

```
package test;

import java.util.Collection;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.base.RuleNameStartsWithAgendaFilter;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
```

```

import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.AgendaFilter;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        Collection collection = kb.getKnowledgePackages();
        KnowledgeBase knowledgeBase =
KnowledgeBaseFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(collection);
        StatefulKnowledgeSession statefulSession = knowledgeBase
            .newStatefulKnowledgeSession();
        statefulSession.getAgenda().getAgendaGroup("002").setFocus();
        statefulSession.fireAllRules();
        statefulSession.dispose();
    }
}

```

在代码清单 2-41 当中，注意如下这句：

```
statefulSession.getAgenda().getAgendaGroup("002").setFocus();
```

这句代码的作用是先得到当前的 Agenda，再通过 Agenda 得到名为 002 的 Agenda Group 对象，最后把 Focus 设置到名为 002 的 Agenda Group 当中，这个位于名为 002 的 Agenda Group 中的 rule2 规则会执行，而位于名为 001 的 Agenda Group 当中的 rule1 则不会被执行，因为名为 001 的 Agenda Group 没有得到 Focus。

运行测试点，可以看到如图 2-7 所示。

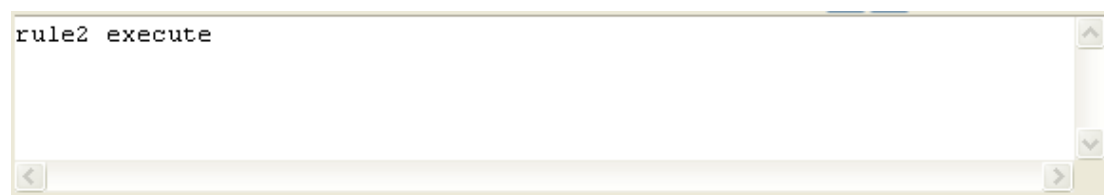


图 2-7

实际应用当中 agenda-group 可以和 auto-focus 属性一起使用，这样就不会在代码当中显示的为某个 Agenda Group 设置 Focus 了。一旦将某个规则的 auto-focus 属性设置为 true，那么即使该规则设置了 agenda-group 属性，我们也不需要再在代码当中显示的设置 Agenda Group 的 Focus 了。

2.2.3.11. auto-focus

前面我们也提到 auto-focus 属性，它的作用是用来在已设置了 agenda-group 的规则上设置该规则是否可以自动独取 Focus，如果该属性设置为 true，那么在引擎执行时，就不需要显示的为某个 Agenda Group 设置 Focus，否则需要。

对于规则的执行的控制，还可以使用 Agenda Filter 来实现。在 Drools 当中，提供了一个名为 org.drools.runtime.rule.AgendaFilter 的 Agenda Filter 接口，用户可以实现该接口，通过规则当中的某些属性来控制规则要不要执行。org.drools.runtime.rule.AgendaFilter 接口只有一个方法需要实现，方法体如下：

```
public boolean accept(Activation activation);
```

在该方法当中提供了一个 Activation 参数，通过该参数我们可以得到当前正在执行的规则对象或其它一些属性，该方法要返回一个布尔值，该布尔值就决定了要不要执行当前这个规则，返回 true 就执行规则，否则就不执行。

代码清单 2-42 中的两个规则设置了 agenda-group 属性，设置了 auto-focus 属性为 true，这样在引擎在执行这两个规则的时候就不需要显示的为 Agenda Group 设置 Focus。

代码清单 2-42

```
package test
rule "rule1"
    agenda-group "001"
    auto-focus true
    when
        eval(true)
    then
        System.out.println("rule1 execute");
    end

rule "rule2"
    agenda-group "002"
    auto-focus true
    when
        eval(true)
    then
        System.out.println("rule2 execute");
    end
```

在引擎执行规则的时候，我们希望使用规则名来对要执行的规则做一个过滤，此时就可以通过 AgendaFilter 来实现，代码清单 2-43 既为我们实现的一个 AgendaFilter 类源码。

代码清单 2-43

```
package test;

import org.drools.runtime.rule.Activation;
import org.drools.runtime.rule.AgendaFilter;

public class TestAgendaFilter implements AgendaFilter {
    private String startName;
    public TestAgendaFilter(String startName){
        this.startName=startName;
    }
    public boolean accept(Activation activation) {
        String ruleName=activation.getRule().getName();
        if(ruleName.startsWith(this.startName)){
            return true;
        }else{
            return false;
        }
    }
}
```

从实现类中可以看到，我们采用的过滤方法是规则名的前缀，通过 Activation 得到当前的 Rule 对象，然后得到当前规则的 name，再用这个 name 与给定的 name 前缀进行比较，如果相同就返回 true，否则就返回 false。

编写测试类，测试类代码如代码清单 2-44 所示。

代码清单 2-44

```
package test;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.AgendaFilter;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
    }
}
```

```
KnowledgeBase knowledgeBase =
KnowledgeBaseFactory.newKnowledgeBase();
knowledgeBase.addKnowledgePackages(kb.getKnowledgePackages());
StatefulKnowledgeSession statefulSession = knowledgeBase
    .newStatefulKnowledgeSession();
AgendaFilter filter=new TestAgendaFilter("rule1");
statefulSession.fireAllRules(filter);
statefulSession.dispose();
}
}
```

在测试类当中，给出的过滤规则名称的前缀是“rule1”，这样只要当前的 Agenda 当中有名称以“rule1”开头的规则，那么就会执行，否则将不会被执行，本例子当中符合要求的规则就是名为 rule1 的规则。运行测试类，运行结果如图 2-8 所示。

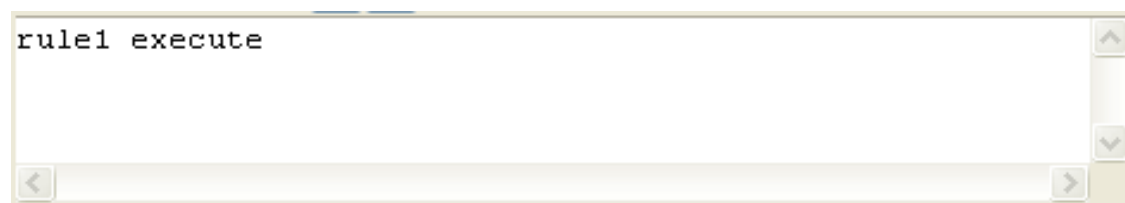


图 2-8

从测试结果来看，我们编写的 AgendaFilter 实现类起到了作用。

2.2.3.12. ruleflow-group

在使用规则流的时候要用到 ruleflow-group 属性，该属性的值为一个字符串，作用是用来将规则划分为一个个的组，然后在规则流当中通过使用 ruleflow-group 属性的值，从而使用对应的规则。后面在讨论规则流的时候还要对该属性进行详细介绍。

2.2.4. 注释

在编写规则的时候适当添加注释是一种好的习惯，好的注释不仅仅可以帮助别人理解你编写的规则也可以帮助你自己日后维护规则。在 Drools 当中注释的写法与编写 Java 类的注释的写法完全相同，注释的写法分两种：单行注释与多行注释。

2.2.4.1. 单行注释

单行注释可以采用“#”或者“//”来进行标记，如代码片段 2-45 中的规则就添加了两个注释，一个以“//”开头，一个是以“#”开头。

代码清单 2-45

```
//规则rule1的注释
rule "rule1"
    when
        eval(true) #没有条件判断
    then
        System.out.println("rule1 execute");
    end
```

对于单行注释来说，您可以根据自己的喜好来选择注释的标记符，我个人的习惯所有的单行注释全部采用“//”来实现，这样就和 Java 语法一致了。

2.2.4.2. 多行注释

如果要注释的内容较多，可以采用 Drools 当中的多行注释标记来实现。Drools 当中的多行注释标记与 Java 语法完全一样，以“/*”开始，以“*/”结束，代码清单 2-46 演示了这种用法。

代码清单 2-46

```
/*
规则rule1的注释
这是一个测试用规则
*/
rule "rule1"
    when
        eval(true) #没有条件判断
    then
        System.out.println("rule1 execute");
    end
```

代码清单 2-46 当中，在规则名称部分，添加了一个多行注释文本，用于表明该规则的作用，实际应用当中我们推荐在每一个规则开始前添加这么一个多行注释，用于说明该规则的作用。

2.3. 函数

函数是定义在规则文件当中一段代码块，作用是将在规则文件当中若干个规则都会用到的业务操作封装起来，实现业务代码的复用，减少规则编写的工作量。

函数的编写位置可以是规则文件当中 `package` 声明后的任何地方，Drools 当中函数声明的语法格式如代码清单 2-47 所示：

代码清单 2-47

```
function void/Object functionName(Type arg...) {  
    /*函数体的业务代码*/  
}
```

Drools 当中的函数以 `function` 标记开头，如果函数体没有返回值，那么 `function` 后面就是 `void`，如果有返回值这里的 `void` 要换成对应的返回值对象，接下来就是函数的名称。函数名称的定义可以参考 Java 类当中方法的命名原则，对于一个函数可以有若干个输入参数，所以函数名后面的括号当中可以定义若干个输入参数。定义输入参数的方法是先声明参数类型，然后接上参数名，这点和 Java 当中方法的输入参数定义是完全一样的，最后就是用“{...}”括起来的业务逻辑代码，业务代码的书写采用的是标准的 Java 语法。

代码清单 2-48 演示了一个简单的函数的编写方法。

代码清单 2-48

```
function void printName(String name) {  
    System.out.println("您的名字是：" + name);  
}
```

该函数的名称为 `printName`，不需要返回值，它需要一个 `String` 类型的输入参数，函数体的业务逻辑比较简单，将输入参数组合后在控制台打印出来。

可以看到 Drools 当中函数的定义与 Java 当中方法的定义极为相似，与 Java 当中定义方法相比，唯一不同之处就是在 Drools 的函数定义当中没有可见范围的设定，而 Java 当中可以通过 `public`、`private` 之类来设置方法的可见范围。在 Drools 当中函数的可见范围是当前的函数所在的规则文件，位于其它规则文件当中的规则是不可以调用不在本规则文件当中的函数的。所以 Drools 当中函数的可见范围可以简单将其与 Java 当中方法的 `private` 类型划上等号。下面我们来看一个函数应用的例子，加深对 Drools 函数的理解。

代码清单 2-49 当中的规则文件包含一个函数和两个规则，在这两个规则当中都调用了这个函数。

代码清单 2-49

```
package test

import java.util.List;
import java.util.ArrayList;

/*
一个测试函数
用来向Customer对象当中添加指定数量的Order对象的函数
*/
function void setOrder(Customer customer,int orderSize) {
    List ls=new ArrayList();
    for(int i=0;i<orderSize;i++){
        Order order=new Order();
        ls.add(order);
    }
    customer.setOrders(ls);
}

/*
测试规则
*/
rule "rule1"
    when
        $customer :Customer();
    then
        setOrder($customer,5);
        System.out.println("rule 1 customer has order
size:"+$customer.getOrders().size());
end

/*
测试规则
*/
rule "rule2"
    when
        $customer :Customer();
    then
        setOrder($customer,10);
        System.out.println("rule 2 customer has order
size:"+$customer.getOrders().size());
end
```

这个函数没有返回值，它有两个输入参数，第一个参数为一个 Customer 对象，第二个

为一个数字，用来决定添加到 Customer 对象当中 Order 的数量。可以看到这个函数体就是用标准的 Java 语法编写，很容易理解。

在 Drools 当中，函数的调用通常是在规则的 RHS 部分中完成，在名为 rule1 和 rule2 的这两个规则当中 RHS 部分都调用了 setOrder 函数，调用完成后将当前的 Customer 对象所拥有的 Order 数量在控制台打印出来，从而验证函数调用后对 Customer 对象产生的影响。

编写测试类，测试类代码如代码清单 2-50 所示。

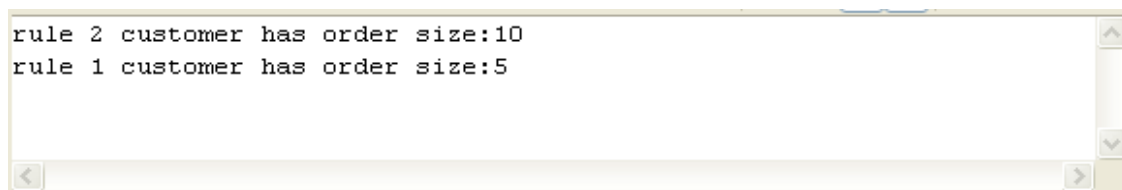
代码清单 2-50

```
package test;

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.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        KnowledgeBase knowledgeBase =
KnowledgeBaseFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(kb.getKnowledgePackages());
        StatefulKnowledgeSession statefulSession = knowledgeBase
            .newStatefulKnowledgeSession();
        statefulSession.insert(new Customer());
        statefulSession.fireAllRules();
        statefulSession.dispose();
    }
}
```

运行测试类，可以看到如图 2-9 所示的结果。



```
rule 2 customer has order size:10
rule 1 customer has order size:5
```

图 2-9

从运行结果中可以看到, 经过函数的调用, Customer 对象当中 Order 的数量已发生变化。

实际应用当中, 可以考虑使用在 Java 类当中定义静态方法的办法来替代在规则文件中定义函数。我们知道 Java 类当中的静态方法, 不需要将该类实例化就可以使用该方法, 利用这种特性, Drools 为我们提供了一个特殊的 import 语句: import function, 通过该 import 语句, 可以实现将一个 Java 类中静态方法引入到一个规则文件当中, 使得该文件当中的规则可以像使用普通的 Drools 函数一样来使用 Java 类中某个静态方法。

代码清单 2-51 是一个包括名为 printInfo 静态方法的 Java 类。

代码清单 2-51

```
package test;

public class RuleTools {
    public static void printInfo(String name){
        System.out.println("your name is :"+name);
    }
}
```

名为 RuleTools 类中静态方法 printInfo 在规则当中调用方法如代码清单 2-52 所示。

代码清单 2-52

```
package test

import function test.RuleTools.printInfo;

/*
测试规则
*/
rule "rule1"
    when
        eval(true);
    then
        printInfo("高杰");
end
```

在代码清单 2-52 中, 通过使用 import function 关键字, 将 test.RuleTools 类中静态方法 printInfo 引入到当前规则文件中, 在名为 rule1 的规则 RHS 当中, 将 printInfo 静态方法作为一个普通的 Drools 函数来进行使用, 有兴趣的读者可以编写测试类, 测试一下该规则的运行结果。

2.4. 查询

查询是 Drools 当中提供的一种根据条件在当前的 WorkingMemory 当中查找 Fact 的方法。查询是定义在规则文件当中，和函数一样，查询的定义可以是 package 语句下的任意位置，在 Drools 当中查询可分为两种：一种是不需要外部传入参数；一种是需要外部传入参数。下面我们就分别对这两种类型的查询进行介绍。

2.4.1. 无参数查询

在 Drools 当中查询以 query 关键字开始，以 end 关键字结束，在 package 当中一个查询要有唯一的名称，查询的内容就是查询的条件部分，条件部分内容的写法与规则的 LHS 部分写法完全相同。查询的格式如下：

```
query "query name"
    #conditions
end
```

代码清单 2-53 是一个简单的查询示例。

代码清单 2-53

```
query "testQuery"
    customer:Customer(age>30,orders.size >10)
end
```

在这个查询当中，查询的名称为 testQuery，条件是取到所有的 age>30，并且拥有 Order 数量大于 10 的 Customer 对象的集合。从这个示例当中可以看出，条件的写法与规则的 LHS 部分完全相同。

查询的调用是由 StatefulSession 完成的，通过调用 StatefulSession 对象的 getQueryResults(String queryName)方法实现对查询的调用，该方法的调用会返回一个 QueryResults 对象，QueryResults 是一个类似于 Collection 接口的集合对象，在它当中存放在若干个 QueryResultsRow 对象，通过 QueryResultsRow 可以得到对应的 Fact 对象，从而实现根据条件对当前 WorkingMemory 当中 Fact 对象的查询。我们来编写一个针对于代码清单 2-53 当中查询的测试用例。

代码清单 2-54

```
package test;

import java.util.ArrayList;
```

```
import java.util.List;

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.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.QueryResults;
import org.drools.runtime.rule.QueryResultsRow;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        KnowledgeBase knowledgeBase =
KnowledgeBaseFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(kb.getKnowledgePackages());
        StatefulKnowledgeSession statefulSession = knowledgeBase
            .newStatefulKnowledgeSession();
        //向当前WorkingMemory当中插入Customer对象
        statefulSession.insert(generateCustomer("张三", 20, 21));
        statefulSession.insert(generateCustomer("李四", 33, 11));
        statefulSession.insert(generateCustomer("王二", 43, 12));
        //调用查询
        QueryResults
queryResults=statefulSession.getQueryResults("testQuery");
        for(QueryResultsRow qr:queryResults){
            Customer cus=(Customer)qr.get("customer");
            //打印查询结果
            System.out.println("customer name :"+cus.getName());
        }
        statefulSession.dispose();
    }

    /**
     * 产生包括指定数量Order的Customer
     * */
    public static Customer generateCustomer(String name,int age,int
orderId){
        Customer cus=new Customer();
    }
}
```

```

        cus.setName(name);
        cus.setAge(age);
        List ls=new ArrayList();
        for (int i = 0; i < orderSize; i++) {
            ls.add(new Order());
        }
        cus.setOrders(ls);
        return cus;
    }
}

```

在代码清单 2-54 的测试用例当中，向当前 WorkingMemory 中插入了三个 Customer 对象，其中只有后两个能满足 testQuery 的查询条件，所以在查询结果输入当中只能在控制台输出后两个 Customer 的 name 属性的值。测试用例运行结果如图 2-10 所示。

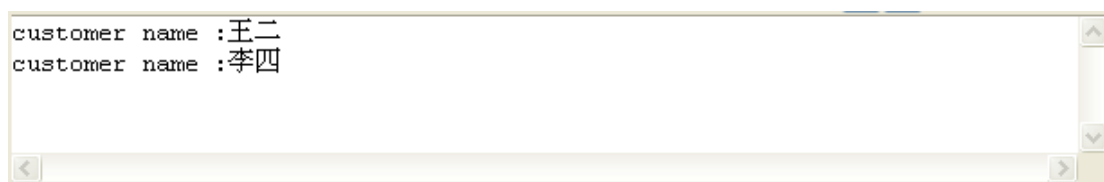


图 2-10

2.4.2. 参数查询

和函数一样，查询也可以接收外部传入参数，对于可以接收外部参数的查询格式如下：

```

query "query name" (Object obj,...)
    #conditions
end

```

和不带参数的查询相比，唯一不同之处就是在查询名称后面多了一个用括号括起来的输入参数，查询可接收多个参数，多个参数之间用“,”分隔，每个参数都要有对应的类型声明，代码清单 2-55 演示了一个简单的带参数的查询示例。

代码清单 2-55

```

query "testQuery"(int $age,String $gender)
    customer:Customer(age>$age,gender==$gender)
end

```

在这个查询当中，有两个外部参数需要传入，一个是类型为 int 的 \$age；一个是类型为 String 的 \$gender（这里传入参数变量名前添加前缀“\$”符号，是为了和条件表达式中相关变量区别开来）。对于带参数的查询，可以采用 StatefulSession 提供的 getQueryResults(String

queryName,new Object[]{}))方法来实现,这个方法中第一个参数为查询的名称,第二个 Object 对象数组既为要输入的参数集合。代码清单 2-56 演示了代码清单 2-55 中查询的调用方法。

代码清单 2-56

```
package test;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;
import org.drools.runtime.rule.QueryResults;
import org.drools.runtime.rule.QueryResultsRow;

public class Test {
    public static void main(String[] args) {
        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/test.drl"),
ResourceType.DRL);
        KnowledgeBase knowledgeBase =
KnowledgeBuilderFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(kb.getKnowledgePackages());
        StatefulKnowledgeSession statefulSession = knowledgeBase
            .newStatefulKnowledgeSession();
        //向当前WorkingMemory当中插入Customer对象
        statefulSession.insert(generateCustomer("张三",20,"F"));
        statefulSession.insert(generateCustomer("李四",33,"M"));
        statefulSession.insert(generateCustomer("王二",43,"F"));
        //调用查询
        QueryResults
queryResults=statefulSession.getQueryResults("testQuery", new
Object[] {new Integer(20),"F"});
        for(QueryResultsRow qr:queryResults){
            Customer cus=(Customer)qr.get("customer");
            //打印查询结果
            System.out.println("customer name :"+cus.getName());
        }
        statefulSession.dispose();
    }

    /**
```



```

* 产生Customer对象
* */
public static Customer generateCustomer(String name,int age,String
gender){
    Customer cus=new Customer();
    cus.setAge(age);
    cus.setName(name);
    cus.setGender(gender);
    return cus;
}
}

```

测试用例当中，在调用 `getQueryResults` 方法，执行名为 `testQuery` 的查询结果时，我们给出的参数是 `age>20`，且 `gender="F"` 的条件，执行测试用例可以看到如图 2-11 所示的输出结果。

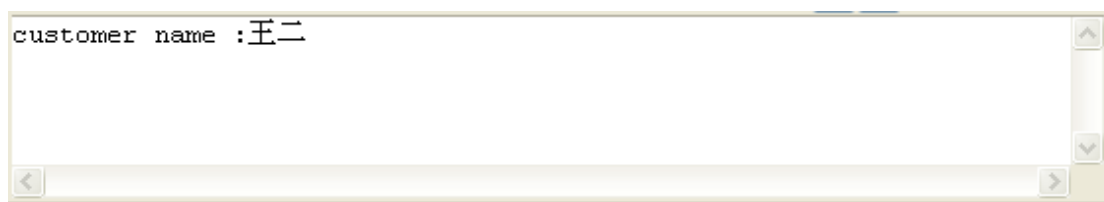


图 2-11

从输出结果来看，满足条件的只有 `name` 属性为“王二”的 `Customer` 对象。

2.5. 对象定义

在 Drools 当中，可以定义两种类型的对象：一种是普通的类型 `Java Fact` 的对象；另一种是用来描述 `Fact` 对象或其属性的元数据对象。

2.5.1. Fact 对象定义

我们知道在 Drools 当中是通过向 `WorkingMemory` 中插入 `Fact` 对象的方式来实现规则引擎与业务数据的交互，对于 `Fact` 对象就是普通的具有若干个属性及其对应的 `getter` 与 `setter` 方法的 `JavaBean` 对象。Drools 除了可以接受用户在外向 `WorkingMemory` 当中插入现成的 `Fact` 对象，还允许用户在规则文件当中定义一个新的 `Fact` 对象。

在规则文件当中定义 `Fact` 对象要以 `declare` 关键字开头，以 `end` 关键字结尾，中间部分就是该 `Fact` 对象的属性名及其类型等信息的声明。代码清单 2-57 是一个简单的在规则当中

定义的 Fact 对象示例。

代码清单 2-57

```
declare Address
    city : String
    addressName : String

end
```

在代码清单 2-57 当中定义的 Fact 对象名为 Address，它有两个皆为 String 类型的，名为 city 和 addressName 属性，该对象与代码清单 2-58 当中定义务 Address 对象是等效的。

代码清单 2-58

```
public class Address {
    private String city;
    private String addressName;
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getAddressName() {
        return addressName;
    }
    public void setAddressName(String addressName) {
        this.addressName = addressName;
    }
}
```

相比代码清单 2-58 当中定义的 Address 对象，在代码清单 2-57 当中定义的 Address 对象只有属性的声明，没有与这些属性对应的 getter 与 setter 方法的定义。

通过 declare 关键字声明了一个 Fact 对象之后，在规则当中可以像使用普通的 JavaBean 一样来使用我们声明好的 Fact 对象，代码清单 2-59 演示了在规则当中使用 Address 对象的情形。

代码清单 2-59

```
declare Address
    city : String
    addressName : String

end

rule "rule1"
    salience 2
    when
```

```

        eval(true);
    then
        #使用定义的Address对象，并利用insert操作符将其插入到当前的
        WorkingMemory当中
        Address add=new Address();
        add.setCity("中国上海");
        add.setAddressName("中国上海松江区");
        insert(add);
    end

    rule "rule2"
        salience 1
        when
            $add:Address()
        then
            System.out.println($add.getCity()+" "+$add.getAddressName());
        end
end

```

在代码清单 2-59 当中，有一个通过 declare 关键字定义的名为 Address 的 Fact 对象，在名为 rule1 的规则当中 RHS 部分创建了这个 Address 对象的实例，分别设置了该实例对象的 city 属性及 addressName 属性的值，最后将其插入到当前的 WorkingMemory 当中；名为 rule2 的规则首先判断了当前的 WorkingMemory 当中有没有 Address 对象的实例，如果有就在其 RHS 部分当中在控制台打印出它的 city 属性及 addressName 属性的值，

在 rule1 当中，我们设置了它的 salience 属性值为 2，rule2 的 salience 属性值为 1，因为 rule1 的 salience 属性值大于 rule2 的，所以它的优先级较高，会先于 rule2 执行。我们这样做的目的是想让 rule1 先执行，通过它的 RHS 部分向当前的 WorkingMemory 当中插入一个 Address 对象，这样 rule2 再执行的时候就可以检测到当前 WorkingMemory 当中有一个 Address 对象，就可以通过其 RHS 部分打印出 Address 对象的 city 与 addressName 属性的值。事实上，因为我们在 rule1 当中通过 insert 操作符将 Address 对象的实例插入到当前的 WorkingMemory 当中，根据前面对于 insert 操作符的介绍我们知道，不管 rule2 有没有执行过，因为有新对象插入到当前 WorkingMemory 当中，所以 rule2 都会再执行一次，所以这里 rule1 与 rule2 的属性不设置也同样可以达到目的。

编写测试类运行代码清单 2-59 当中的规则，可以看到如图 2-12 所示的结果。

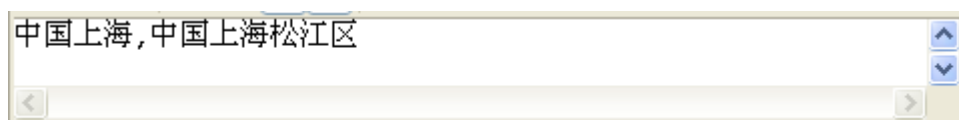


图 2-12

在使用 `declare` 关键字来声明一个 `Fact` 对象时，对于属性类型的定义，除了可以使用类似 `java.lang.String` 之类的简单类型之外，也可以使用另外一些 `JavaBean` 作为某个属性的类型，代码清单 2-60 演示了这种用法。

代码清单 2-60

```
import java.util.Date;

declare Address
    city : String
    addressName : String
end

declare Person
    name:String
    birthday:Date
    address:Address //使用declare声明的对象作为address属性类型
    order:Order //使用名为Order的JavaBean作为order属性类型
end

rule "rule1"
    when
        eval(true);
    then
        #使用定义的Address对象，并利用insert操作符将其插入到当前的
        WorkingMemory当中
        Address add=new Address();
        add.setCity("中国上海");
        add.setAddressName("中国上海松江区");

        Order order=new Order();
        order.setName("测试订单");

        Person person=new Person();
        person.setName("高杰");
        person.setBirthday(new Date());
        person.setAddress(add); //将Address对象的实例赋给address属性
        person.setOrder(order); //将Order对象实例赋给order属性
        insert(person);
    end

rule "rule2"
    when
        $person:Person()
    then
```

```

        System.out.println("姓名:"+$person.getName()+"\r住址:
"+$person.getAddress().getAddressName()+"\r拥有的订单:
"+$person.getOrder().getName());
end

```

代码清单 2-60 当中，通过 `declare` 关键字我们声明了一个名为 `Person` 的 `Fact` 对象，该对象的 `birthday` 属性类型是 `java.util.Date` 类型，它的 `address` 属性类型是一个通过 `declare` 关键字声明的名为 `Address` 的 `Fact` 对象，它的 `order` 属性类型是一个在外部定义的名为 `Order` 的 `JavaBean` 对象。

在 `rule1` 的 `RHS` 部分中，对 `Person` 对象的所有属性进行了初始化，最后将其插入到当前的 `WorkingMemory` 当中，因为使用 `insert` 操作符插入到 `WorkingMemory`，所以在这里就没有使用设置优先级的 `salience` 属性。名为 `rule2` 规则的 `LHS` 部分首先也是判断当前 `WorkingMemory` 当中有没有 `Person` 对象的实例，如果存在，那么就在其 `RHS` 部分中向控制台打印该 `Person` 对象的 `name` 属性、`address` 属性及 `order` 属性，运行这两个规则可以看到如图 2-13 所示的结果。

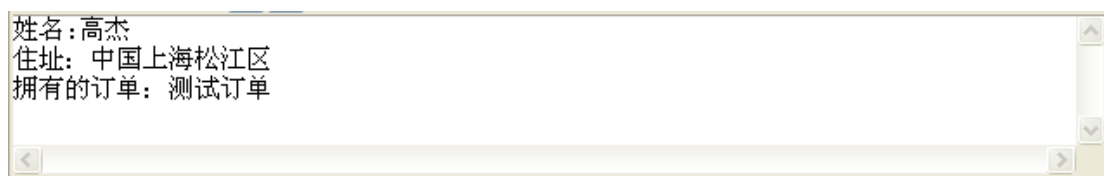


图 2-13

在 `Java` 类中，可以通过 `KnowledgeBase` 对象得到在规则文件当中使用 `declare` 关键字定义的 `Fact` 对象、可以将其实例化、可以为该实例化对象赋值、同时可以将其插入到当前的 `WorkingMemory` 当中。代码清单 2-61 演示了这种用法。

代码清单 2-61

```

package test;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBuilderFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.definition.type.FactType;
import org.drools.io.impl.ClassPathResource;
import org.drools.runtime.StatefulKnowledgeSession;

public class Test {
    public static void main(String[] args) throws Exception{

```

```

        KnowledgeBuilder kb =
KnowledgeBuilderFactory.newKnowledgeBuilder();
        kb.add(new ClassPathResource("test/demo.drl"),
ResourceType.DRL);
        KnowledgeBase knowledgeBase =
KnowledgeBaseFactory.newKnowledgeBase();
        knowledgeBase.addKnowledgePackages(kb.getKnowledgePackages());

        Order order=new Order();
        order.setName("测试定单");

        //获取规则文件当中定义的Address对象并对其进行实例化
        FactType
addressType=knowledgeBase.getFactType("test","Address");
        Object add=addressType.newInstance();
        addressType.set(add, "city", "中国上海");
        addressType.set(add, "addressName", "中国上海松江区");

        //获取规则文件当中定义的Person对象并对其进行实例化
        FactType
personFact=knowledgeBase.getFactType("test","Person");
        Object obj=personFact.newInstance();//实例化该对象f
        personFact.set(obj, "name", "高杰");//设置该对象的name属性
        personFact.set(obj, "order", order);
        personFact.set(obj, "address", add);

        StatefulKnowledgeSession statefulSession = knowledgeBase
                .newStatefulKnowledgeSession();
        //将实例化好的Person对象插入到当前的WorkingMemory当中
        statefulSession.insert(obj);

        statefulSession.fireAllRules();
        statefulSession.dispose();
    }
}

```

此时删除 2-60 代码当中 rule1 这个规则，运行这个测试类，可以得到如图 2-13 所示的结果。

2.5.2. 元数据定义

我们知道，元数据就是用来描述数据的数据。在 Drools5 当中，可以为 Fact 对象、Fact 对象的属性或者是规则来定义元数据，元数据定义采用的是“@”符号开头，后面是元数据

的属性名（属性名可以是任意的），然后是括号，括号当中是该元数据属性对应的具体值（值是任意的）。具体格式如下：

```
@metadata_key( metadata_value )
```

在这里 `metadata_value` 属性值是可选的。具体实例如下：

```
@author(gaojie)
```

这个元数据属性名为 `author`，值为 `gaojie`。

代码清单 2-62

```
declare User
    @author(gaojie)
    @createTime(2009-10-25)
    username : String @maxLenth(30)
    userid : String @key
    birthday : java.util.Date
end
```

代码清单 2-62 是一个元数据在通过 `declare` 关键字声明对象时的具体应用实例，在这个例子当中，一共有四个元数据的声明，首先对于 `User` 对象有两个元数据的声明：`author` 说明这个对象的创建者是 `gaojie`；`createTime` 说明创建的时间为 2009-10-25 日。接下来是对 `User` 对象的 `username` 属性的描述，说明该属性的 `maxLenth` 的值为 30。最后是对 `userid` 属性的描述，注意这个描述只有一个无数据的属性名 `key`，而没有为这个属性 `key` 赋予具体的值。

3. DSL

DSL 的全称是 Domain Specific Language，翻译过来可称之为“域特定语言”，是 Drools5 当中的一种模版，这个模版可以描述规则当中的条件（LHS 部分）或规则当中的条件（RHS 部分），这样在使用的时候可以可以在规则当中多处引用这个模版，同时因为模版的定义方式采用的是 `key-value` 键值对对应的方式，所以通过在 `key` 当中使用自然语言就可以实现用自然语言的方式来描述规则的方法。

我们知道，规则文件是一个文本文件，除了使用 Drools5 提供的规则编辑器之外，如果对规则编写语法特别熟悉，完全可以采用记事本来编写业务规则。DSL 的定义也是一个文本文件，一个具有属性文件结构的文本文件，一旦在这个文件当中定义好 DSL 后，就可以在规则文件当中引用这个 DSL 文件，使用这个文件当中定义好的 DSL。Drools5 也为我们提供了编写 DSL 文件的编辑器，通过这个编辑器，可以实现对 DSL 的快速定义。

3.1.DSL 文件的创建

DSL 文件的建立，可以借助于 Drools5 提供的 Eclipse 插件来完成，打开 Eclipse，选择菜单 File→New→Other，在弹出的窗口中展开列表树当中的 Drools 节点，选择其下的 Domain Specific Language，如图 3-1 所示。

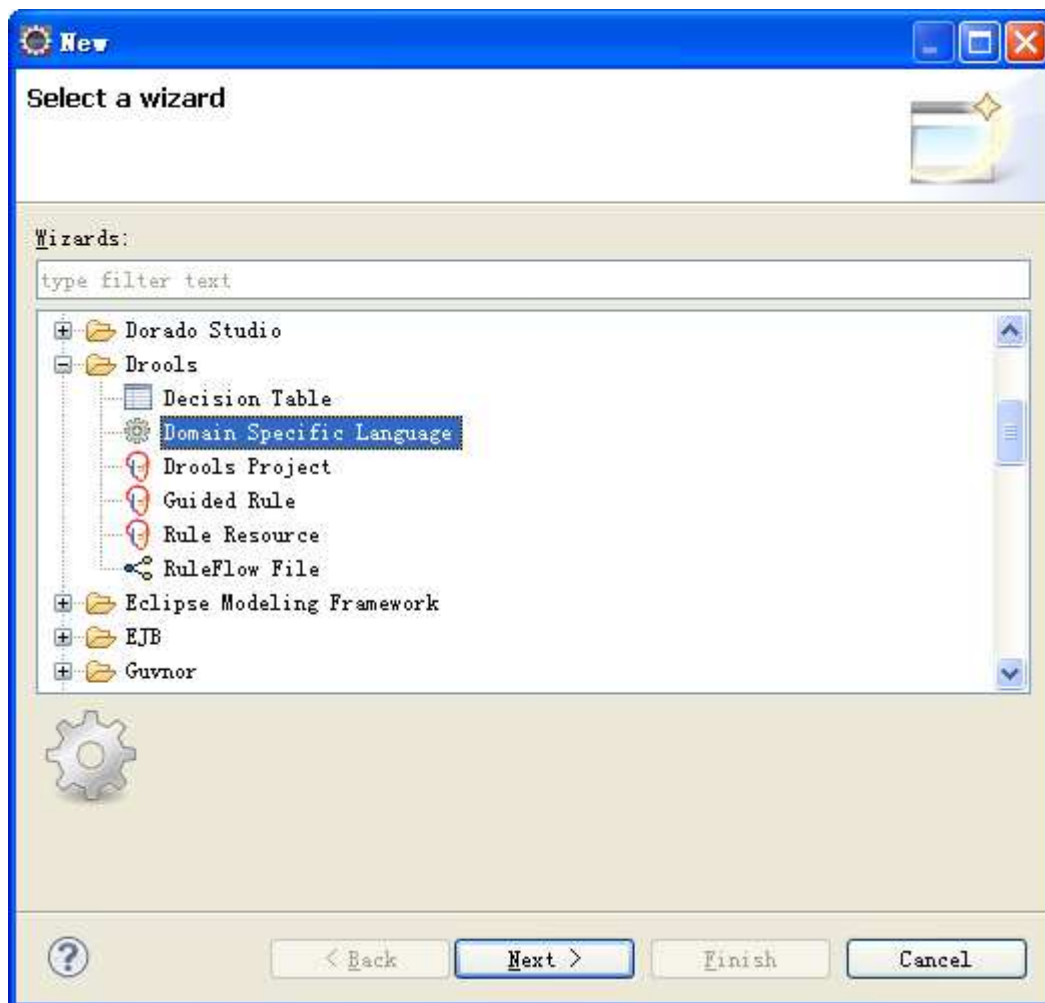


图 3-1

点击 Next 按钮，输入 DSL 文件名及其要存放的文件夹后，点击 Finish 按钮后 IDE 会自动将当前新建的 DSL 文件用 Drools5 提供的 IDE 打开，如图 3-2 所示。

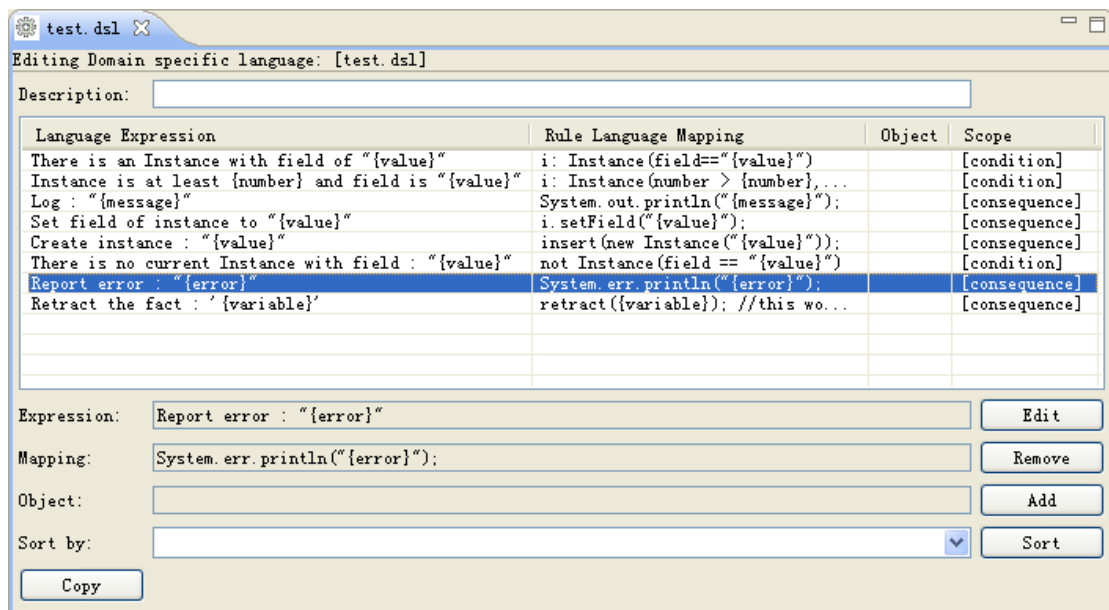


图 3-2

默认情况下，新创建的 DSL 文件会包含若干个 DSL 语句的示例，这些 DSL 示例语句向我们演示了 DSL 的语法规范。

通过使用 Drools5 为提供的 DSL 文件编辑器，可以实现对 DSL 语句的增加、删除、修改及排序操作。

3.2. 使用 DSL

4. 规则流