# 零基础入门Drools(一)

# 一、Drools简介

## 某电商平台促销活动

活动规则是根据用户购买订单的金额给用户送相应的积分，购买的越多送的积分越多，用户可以使用积分来兑换相应的商品，我们这次活动的力度很大，肯定会吸引很多的用户参加，产品经理小王兴高采烈唾液横飞的对小明讲到。

用户购买的金额和对应送多少积分的规则如下:

Plain Text 复制代码

```
100元以下, 不加分
100元-500元 加100分
500元-1000元 加500分
1000元 以上 加1000分
```

小明一看，这需求果然简单呀，作为一个工作了两三年的程序员来讲，这不就是小case，半天搞定，送积分的心代码如下：

```java
public void execute() throws Exception {

    List<Order> orderList = getInitData();
    for (int i=0; i<orderList.size(); i++){
        Order order = orderList.get(i);
        if (order.getAmout() <= 100){
            order.setScore(0);
            addScore(order);
        }else if(order.getAmout() > 100 && order.getAmout() <= 500){
            order.setScore(100);
            addScore(order);
        }else if(order.getAmout() > 500 && order.getAmout() <= 1000){
            order.setScore(500);
            addScore(order);
        }else{
            order.setScore(1000);
            addScore(order);
        }
    }

}
```

上线运行了半天之后，财务部的小财突然监测到活动账户的金额大为减少，发现产品做活动竟然没有通知到他，非常不爽，于是给领导小马说，这样大规模的活动，对公司财务有压力，领导小马权衡了一番说，这样吧活动继续，但是金额翻倍在送积分，于是规则变成了这样：200元以下不给积分，1000元以下给100积分...

小明看领导都发话了，没办法改呀，不过也简单，就是将里面的值都翻了倍，在投产上去，只是挨了不少测试的白眼。

活动又进行了一天，运营人员通过后台监控发现提到2倍以后，用户积极性变的很差，活动效果不理想，和领导商议了一下，改为最初规则的1.5倍，及150元一下不给积分，750元以下给100积分...

没办法还得改不是，当然这次小明可学乖了，将这些数据（多少元送多少分）存到了数据库中，当老板再改主意的时候，只要改一下数据库的值就可以了，小明为自己的明聪明到有点小高兴。
核心代码编程了这样

```java
public void execute() throws Exception {

    List<Order> orderList = getInitData();
    List<int> values = getTableValues();
    for (int i=0; i<orderList.size(); i++){
        Order order = orderList.get(i);
        if (order.getAmout() <= values.get(0)){
            order.setScore(values.get(3));
            addScore(order);
        }else if(order.getAmout() > values.get(0) && order.getAmout() <= values.get(1)){
            order.setScore(values.get(4));
            addScore(order);
        }else if(order.getAmout() > values.get(1) && order.getAmout() <= values.get(2)){
            order.setScore(values.get(5));
            addScore(order);
        }else{
            order.setScore(values.get(6));
            addScore(order);
        }
    }

}
```

正当小明得意洋洋的打了个最新版本投产上线之后，产品经理小王说积分规则层次太少了，由以前的4组变成8组，小明此刻的心情：kao …

小明想这样下去非得被他们弄死，必须要找找有什么技术可以将活动规则和代码解耦，不管规则如何变化，执行端不用动。小明搜了半天还真有这样的东西，那就是规则引擎，那么规则引擎到底是什么东西呢？我们来看看。

## 规则引擎

规则引擎：全称为**业务规则管理系统**，英文名为BRMS(即Business Rule Management System)。规则引擎的主要思想是将应用程序中的业务决策部分分离出来，并使用预定义的语义模块编写业务决策（业务规则），由用户或开发者在需要时进行配置、管理。 需要注意的是规则引擎并不是一个具体的技术框架，而是指的一类系统，即业务规则管理系统。 目前市面上具体的规则引擎产品有：drools、VisualRules、iLog等，使用最为广泛并且开源的是Drools。 规则引

擎实现了将业务决策从应用程序代码中分离出来，接收数据输入，解释业务规则，并根据业务规则做出业务决策。规则引擎其实就是一个输入输出平台。

**规则引擎主要应用场景**

对于一些存在比较复杂的业务规则并且业务规则会频繁变动的系统比较适合使用规则引擎，如下：

1、风险控制系统----风险贷款、风险评估

2、反欺诈项目----银行贷款、征信验证

3、决策平台系统----财务计算

4、促销平台系统----满减、打折、加价购

# Drools介绍

drools是一款由JBoss组织提供的基于Java语言开发的开源规则引擎，可以**将复杂且多变的业务规则从硬编码中解放出来，以规则脚本的形式存放在文件或特定的存储介质中(例如存放在数据库中)**，使得业务规则的变更**不需要修改项目代码、不用重启服务器**就可以在线上环境立即生效。

`drools官网地址`：https://drools.org/

`drools源码下载地址`：https://github.com/kiegroup/drools

- 使用规则引擎能够解决什么问题？ 针对复杂的业务规则代码处理，往往存在一下问题：

1、硬编码实现业务规则难以维护；

2、硬编码实现业务规则难以应对变化；

3、业务规则发生变化需要修改代码，重启服务后才能生效； 于是规则引擎Drools便诞生在项目中。。。

- 使用规则引擎的优势如下：

1、业务规则与系统代码分离，实现业务规则的集中管理

2、在不重启服务的情况下可随时对业务规则进行扩展和维护

3、可以动态修改业务规则，从而快速响应需求变更

4、规则引擎是相对独立的，只关心业务规则，使得业务分析人员也可以参与编辑、维护系统的业务规则

5、减少了硬编码业务规则的成本和风险

6、使用规则引擎提供的规则编辑工具，使复杂的业务规则实现变得的简单

在项目中使用drools时，即可以单独使用也可以整合spring使用。如果单独使用只需要导入如下maven坐标即可：

```xml
<dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>7.6.0.Final</version>
</dependency>
```

如果我们使用IDEA开发drools应用，IDEA中已经集成了drools插件。如果使用eclipse开发drools应用还需要单独安装drools插件。9

drools API开发步骤如下：


# 二、Drools快速入门

## 业务场景说明

用户购买的金额和对应送多少积分的规则如下:

```
100元以下, 不加分
100元-500元 加100分
500元-1000元 加500分
1000元 以上 加1000分
```

## 开发实现

第一步：创建maven工程droolsDemo并导入drools相关maven坐标

```xml
<dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>7.10.0.Final</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>

```

第二步：根据drools要求创建resources/META-INF/kmodule.xml配置文件

需要有一个配置文件告诉代码规则文件drl在哪里，在drools中这个文件就是kmodule.xml，放置到resources/META-INF目录下。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
    <!--
        name:指定kbase的名称，可以任意，但是需要唯一
        packages:指定规则文件的目录，需要根据实际情况填写，否则无法加载到规则文件
        default:指定当前kbase是否为默认
    -->
    <kbase name="myKbase1" packages="rules" default="true">
        <!--
            name:指定ksession名称，可以任意，但是需要唯一
            default:指定当前session是否为默认
        -->
        <ksession name="ksession-rule" default="true"/>
    </kbase>
</kmodule>

```

注意：上面配置文件的名称和位置都是固定写法，不能更改

Kmodule 中可以包含一个到多个 kbase,分别对应 drl 的规则文件。

Kbase 需要一个唯一的 name,可以取任意字符串。

packages 为drl文件所在resource目录下的路径。注意区分drl文件中的package与此处的package不一定相同。多个包用逗号分隔。默认情况下会扫描 resources目录下 所有(包含子目录)规则文件。

kbase的default属性,标示当前KieBase是不是默认的,如果是默认的则不用名称就可以查找到该KieBase,但每个 module 最多只能有一个默认 KieBase。

kbase 下面可以有一个或多个 ksession, ksession 的 name 属性必须设置,且必须唯一。

第三步：创建实体类Order

Java 复制代码

```java
package com.chenj.entity;

public class Order {
    private int amout;//订单原价金额
    private int score;//积分


    public int getAmout() {
        return amout;
    }

    public void setAmout(int amout) {
        this.amout = amout;
    }

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Order{" +
                "amout=" + amout +
                ", score=" + score +
                '}';
    }
}

```

第四步：创建规则文件resources/rules/score-rules.drl

Plain Text 复制代码

```
package rules

import com.chenj.entity.Order

//规则1：100元以下, 不加分
rule "score_1"
    when
        $s : Order(amout <= 100)
    then
        $s.setScore(0);
        System.out.println("成功匹配到规则1：100元以下, 不加分 ");
end

//规则2：100元-500元 加100分
rule "score_2"
    when
        $s : Order(amout > 100 && amout <= 500)
    then
        $s.setScore(100);
        System.out.println("成功匹配到规则2：100元-500元 加100分 ");
end
//规则3：500元-1000元 加500分
rule "score_3"
    when
        $s : Order(amout > 500 && amout <= 1000)
    then
        $s.setScore(500);
        System.out.println("成功匹配到规则3：500元-1000元 加500分 ");
end
//规则4：1000元 以上 加1000分
rule "score_4"
    when
        $s : Order(amout > 1000)
    then
        $s.setScore(1000);
        System.out.println("成功匹配到规则4：1000元 以上 加1000分 ");
end
```

这个规则文件就是描述了，当符合什么条件的时候，应该去做什么事情，每当规则有变动的时候，我们只需要修改规则文件，然后重新加载即可生效。

第五步：编写单元测试

Java 复制代码

```java
package com.chenj.test;

import com.chenj.entity.Order;
import org.junit.Test;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class TestDrools {
    @Test
    public void test1(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        //构造订单对象，设置订单金额，由规则引擎计算获得的积分
        Order order = new Order();
        order.setAmout(200);

        //将数据交给规则引擎，规则引擎会根据提供的数据进行规则匹配
        kieSession.insert(order);

        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //关闭会话
        kieSession.dispose();
        //打印结果;
        System.out.println("订单提交之后积分："+order.getScore());
    }

}

```

## 案例小结

我们在操作Drools时经常使用的API以及它们之间的关系如下图：


**Kie全称为Knowledge Is Everything，即"知识就是一切"的缩写，是Jboss一系列项目的总称**

### 规则引擎构成

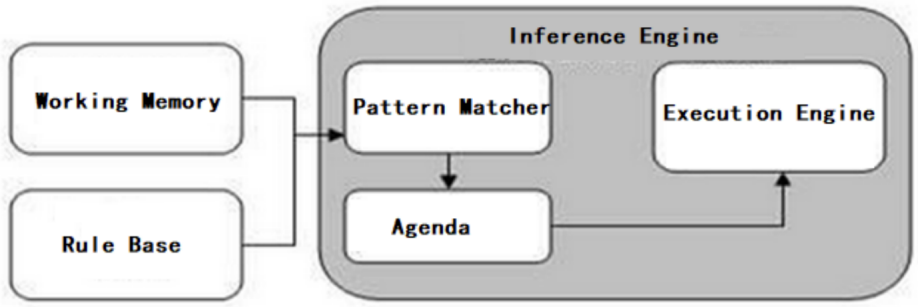
drools规则引擎由以下三部分构成：

- Working Memory（工作内存）
- Rule Base（规则库）
- Inference Engine（推理引擎）

其中Inference Engine（推理引擎）又包括：

- Pattern Matcher（匹配器） 具体匹配哪一个规则，由这个完成
- Agenda(议程)
- Execution Engine（执行引擎）

如下图所示：




### 相关概念说明

**Working Memory**：工作内存，drools规则引擎会从Working Memory中获取数据并和规则文件中定义的规则进行模式匹配，所以我们开发的应用程序只需要将我们的数据插入到Working Memory中即可，例如本案例中我们调用kieSession.insert(order)就是将order对象插入到了工作内存中。

**Fact**：事实，是指在drools 规则应用当中，将一个**普通的JavaBean插入到Working Memory后的对象**就是Fact对象，例如本案例中的Order对象就属于Fact对象。Fact对象是我们的应用和规则引擎进行数据交互的桥梁或通道。

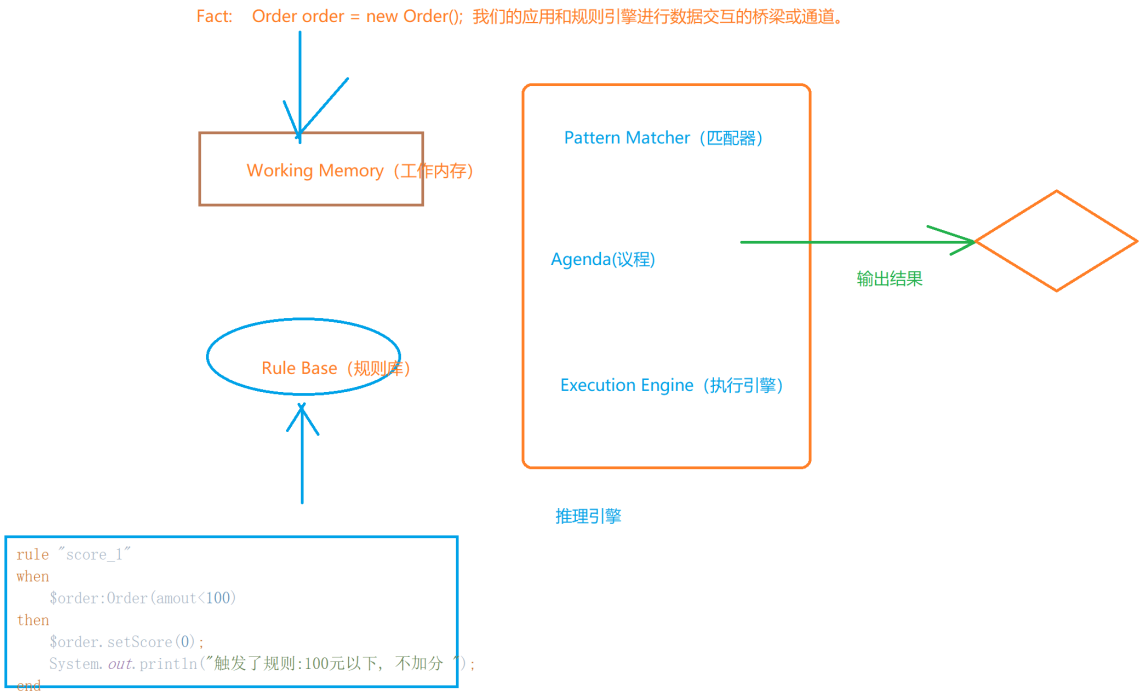
**Rule Base**：规则库，我们在规则文件中定义的规则都会被加载到规则库中。

**Pattern Matcher**：匹配器，将Rule Base中的所有规则与Working Memory中的Fact对象进行模式匹配，匹配成功的规则将被激活并放入Agenda中。

**Agenda**：议程，用于存放通过匹配器进行模式匹配后被激活的规则。

**Execution Engine**：执行引擎，执行Agenda中被激活的规则。

### 规则引擎执行过程




# 三、Drools基础语法

## 规则文件的构成

**drl是Drools Rule Language的缩写**。在规则文件中编写具体的规则内容。

一套完整的规则文件内容构成如下：

- package：包名，package对应的不一定是真正的目录，可以任意写com.abc，同一个包下的drl文件可以相互访问
- import：用于导入类或者静态方法
- global：全局变量
- function：自定义函数
- query：查询
- rule end：规则体

## 规则体语法结构

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）

```
rule "ruleName"         //rule关键字，表示规则开始，参数为规则的唯一名称
    attributes          //规则属性，是rule与when之间的参数，为可选项
    when                //关键字，后面是规则的条件部分
        LHS             //Left Hand Side，是规则的条件部分
    then                //后面跟规则的结果部分
        RHS             //是规则的结果或行为
end                     //表示一个规则的结束
```

## 条件部分

**LHS(Left Hand Side)：是规则的条件部分的通用名称。它由零个或多个条件元素组成。如果LHS为空，则它将被视为始终为true的条件元素。** （左手边）

LHS部分由一个或者多个条件组成，条件又称为pattern。

```
pattern的语法结构为：绑定变量名:Object(Field约束)
```

其中绑定变量名可以省略，通常绑定变量名的命名一般建议以$开始。如果定义了绑定变量名，就可以在规则体的RHS部分使用此绑定变量名来操作相应的Fact对象。Field约束部分是需要返回true或者false的0个或多个表达式。

```
//规则1：100元以下，不加分
rule "score_1"
    when
        //工作内存中必须存在Order这种类型的Fact对象-----类型约束
        //Fact对象的amout属性值必须小于等于100------属性约束
        $s : Order(amout <= 100)
    then
        $s.setScore(0);
        System.out.println("成功匹配到规则1：100元以下，不加分 ");
end
```

如果 LHS 部分为空的话，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

1.约束连接

在 LHS 当中，可以包含 0~n 个条件，多个pattern之间可以采用“&&” （and） 、 “||”(or)和“,”(and)来实现，也可以不写，默认连接为and。

Plain Text 复制代码

```
//规则2：100元-500元 加100分
rule "score_2"
    when
        $s : Order(amout > 100 && amout <= 500)
    then
        $s.setScore(100);
        System.out.println("成功匹配到规则2：100元-500元 加100分 ");
end
```

2.比较操作符

在 Drools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

| 符号 | 说明 |
| --- | --- |
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |
| == | 等于 |
| != | 不等于 |
| contains | 检查一个Fact对象的某个属性值是否包含一个指定的对象值 |
| not contains | 检查一个Fact对象的某个属性值是否不包含一个指定的对象值 |
| memberOf | 判断一个Fact对象的某个属性是否在一个或多个集合中 |
| not memberOf | 判断一个Fact对象的某个属性是否不在一个或多个集合中 |
| matches | 判断一个Fact对象的属性是否与提供的标准的Java正则表达式进行匹配 |
| not matches | 判断一个Fact对象的属性是否不与提供的标准的Java正则表达式进行匹配 |

示例：

第一步：创建实体类，用于测试比较操作符

Java 复制代码

```java
package com.chenj.entity;

import java.util.List;

public class Customer {
    //客户姓名
    private String name;
    private List<Order> orderList;//订单集合

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Order> getOrderList() {
        return orderList;
    }

    public void setOrderList(List<Order> orderList) {
        this.orderList = orderList;
    }
}

```

第二步：在/resources/rules下创建规则文件customer-rules.drl

Plain Text 复制代码

```
package rules

import com.chenj.entity.*;

//测试contains规则
rule "rule1"
    when
    $order:Order();
    $customer:Customer(orderList contains $order);
    then
    System.out.println("测试contains规则触发："+$customer.getName());
end

//测试not contains规则
rule "rule2"
    when
    $order:Order();
    $customer:Customer(orderList not contains $order);
    then
    System.out.println("测试not contains规则触发："+$customer.getName());
end


//测试比较操作符matches
rule "rule3"
    when
        Customer(name matches "张.*")
    then
        System.out.println("测试比较操作符matches触发...");
end

//测试比较操作符not matches
rule "rule4"
    when
        Customer(name not matches "张.*")
    then
        System.out.println("测试比较操作符not matches触发...");
end
```

第三步：编写单元测试

Java 复制代码

```java
 @Test
    public void test2(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        //构造订单对象，设置订单金额，由规则引擎计算获得的积分
        Order order = new Order();
        //匹配规则：$order:Order();
        kieSession.insert(order);



        Customer customer = new Customer();
        List<Order> orderList = new ArrayList<>();
        //匹配规则：  $customer:Customer(orderList contains $order);
        //orderList.add(order);
        customer.setOrderList(orderList);
        customer.setName("Jack");


        //将数据交给规则引擎，规则引擎会根据提供的数据进行规则匹配
        kieSession.insert(customer);


        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //关闭会话
        kieSession.dispose();
    }
```

注意：

执行指定规则

我们在调用规则代码时，满足条件的规则都会被执行。那么如果我们只想执行其中的某个规则如何实现呢?

Drools给我们提供的方式是通过规则过滤器来实现执行指定规则。对于规则文件不用做任何修改，只需要修改Java代码即可，

如下：

```java
 @Test
    public void test2(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        //构造订单对象，设置订单金额，由规则引擎计算获得的积分
        Order order = new Order();
        kieSession.insert(order);


        Customer customer = new Customer();
        List<Order> orderList = new ArrayList<>();
        //orderList.add(order);
        customer.setOrderList(orderList);
        customer.setName("Jack");


        //将数据交给规则引擎，规则引擎会根据提供的数据进行规则匹配
        kieSession.insert(customer);


        //通过规则过滤器实现只执行指定规则
        kieSession.fireAllRules(new RuleNameEqualsAgendaFilter("rule4"));
        //关闭会话
        kieSession.dispose();
    }
```

关键语句：

```java
 //通过规则过滤器实现只执行指定规则
kieSession.fireAllRules(new RuleNameEqualsAgendaFilter("rule4"));
```

## 结果部分

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

### 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

需注意：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

示例：

Plain Text 复制代码

```
//Drools提供的内置方法insert

rule "rule5"
    when
        eval(true); //默认成立
    then
        Customer cus=new Customer();
        cus.setName("张三");
        insert(cus);
        System.out.println("测试Drools提供的内置方法insert 触发...");
end

rule "rule6"
    when
        $customer:Customer(name =="张三");
    then
        System.out.println("测试Drools提供的内置方法insert 触发..."+$customer.getName());
end
```

测试：

```java
    @Test
    public void test3(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //关闭会话
        kieSession.dispose();
    }
```

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 Memroy 当中

### 2.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，用来告诉当前的 Working Memory 该 Fact 对象已经发生了变化。

示例：

```java
rule "rule7"
    //no-loop true
    when
        $customer:Customer(name =="李四");
    then
        $customer.setName("张三");
        update($customer);
        System.out.println("测试Drools提供的内置方法update 触发...");
end

rule "rule8"
    when
        $customer:Customer(name =="张三");
    then
        System.out.println("测试Drools提供的内置方法update 触
发..."+$customer.getName());
end
```

测试

```java
    @Test
    public void test3(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        Customer customer = new Customer();
        customer.setName("李四");
        kieSession.insert(customer);

        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //关闭会话
        kieSession.dispose();
    }
```

### 3.retract

retract用来将 Working Memory 当中某个 Fact 对象从 Working Memory 当中删除

```
//Drools提供的内置方法retract
rule "rule9"
    when
        $customer:Customer(name =="李四");
    then
        //retract($customer);
        System.out.println("测试Drools提供的内置方法retract 触发...");
end

rule "rule10"
    when
        $customer:Customer();
    then
        System.out.println("测试Drools提供的内置方法retract 触
发..."+$customer.getName());
end
```

测试

Java 复制代码

```java
 @Test
    public void test3(){
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        Customer customer = new Customer();
        customer.setName("李四");
        kieSession.insert(customer);

        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //通过规则过滤器实现只执行指定规则
        //kieSession.fireAllRules(new
RuleNameEqualsAgendaFilter("rule5"));

        //关闭会话
        kieSession.dispose();
    }
```

## 属性部分

Drools中提供的属性如下表：

| 属性名 | 说明 |
| --- | --- |
| salience | 指定规则执行优先级 |
| dialect | 指定规则使用的语言类型，取值为java和mvel |
| enabled | 指定规则是否启用 |
| date-effective | 指定规则生效时间 |
| date-expires | 指定规则失效时间 |
| activation-group | 激活分组，具有相同分组名称的规则只能有一个规则触发 |
| agenda-group | 议程分组，只有获取焦点的组中的规则才有可能触发 |
| timer | 定时器，指定规则触发的时间 |
| auto-focus | 自动获取焦点，一般结合agenda-group一起使用 |
| no-loop | 防止死循环 |

### 1.salience

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。默认情况下，规则的 salience 默认值为 0。如果不设置salience属性，规则体的执行顺序为由上到下。
示例：

```
rule "attributes_rule1"
salience 1
    when
        eval(true)
    then
        System.out.println("rule1....");
end

rule "attributes_rule2"
salience 2
    when
        eval(true)
    then
        System.out.println("rule2....");
end
```

### 2.no-loop

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。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。

示例：

```
//测试no-loop
rule "attributes_rule3"
    salience 1
    no-loop true
    when
        $customer:Customer(name=="张三")
    then
        update($customer);
        System.out.println("customer name:"+$customer.getName());
end
```

### 3.date-effective

作用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>=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”
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
在java文件中添加此条命令: System.setProperty("drools.dateformat","yyyy-MM-dd");
示例:

Java 复制代码

```
//测试date-effective
rule "attributes_rule4"
    date-effective "2021-11-20" //当前日期不小于2021-11-25时可以执行
    when
        eval(true);
    then
        System.out.println("attributes_rule4 is execution!");
end
```

测试

Java 复制代码

```
 //测试属性部分
    @Test
    public void test4(){
        //设置修改默认的时间格式
        System.setProperty("drools.dateformat","yyyy-MM-dd");
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();
        Customer customer = new Customer();
        customer.setName("张三");
        kieSession.insert(customer);
        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //通过规则过滤器实现只执行指定规则
        //kieSession.fireAllRules(new
RuleNameEqualsAgendaFilter("rule5"));

        //关闭会话
        kieSession.dispose();
    }
```

### 4.date-expires

作用是与 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。

### 5.enabled

作用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。设置其 enabled 属性值为 false，那么引擎就不会执行该规则

### 6.dialect

作用是用来定义规则当中要使用的语言类型，目前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

### 7.activation-group

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。
示例：

Java 复制代码

```
//测试activation-group
rule "attributes_rule5"
    activation-group "test"
    when
        eval(true)
    then
        System.out.println("attributes_rule5 execute...");
end

rule "attributes_rule6"
    activation-group "test"
    when
        eval(true)
    then
        System.out.println("attributes_rule6 execute...");
end
```

### 8.agenda-group

作用是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。

示例：

Java 复制代码

```
//测试agenda-group
rule "attributes_rule7"
    agenda-group "001"
    when
        eval(true)
    then
        System.out.println("attributes_rule7 execute");
end

rule "attributes_rule8"
    agenda-group "002"
    when
        eval(true)
    then
        System.out.println("attributes_rule8 execute...");
end
```

测试：

Java 复制代码

```java
//测试属性部分
    @Test
    public void test4(){
        //设置修改默认的时间格式
        System.setProperty("drools.dateformat","yyyy-MM-dd");
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieContainer =
kieServices.getKieClasspathContainer();
        //会话对象,用于和规则引擎交互
        KieSession kieSession = kieContainer.newKieSession();

        kieSession.getAgenda().getAgendaGroup("002").setFocus(); //获得执行
焦点

        //激活规则引擎，如果匹配成功则执行规则
        kieSession.fireAllRules();
        //通过规则过滤器实现只执行指定规则
        //kieSession.fireAllRules(new
RuleNameEqualsAgendaFilter("rule5"));

        //关闭会话
        kieSession.dispose();
    }
```

### 9.auto-focus

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。

### 10.timer属性

timer属性可以通过定时器的方式指定规则执行的时间，使用方式有两种：

**方式一**：timer (int: ?)

此种方式遵循java.util.Timer对象的使用方式，第一个参数表示几秒后执行，第二个参数表示每隔几秒执行一次，第二个参数为可选。

**方式二**：timer(cron: )

此种方式使用标准的unix cron表达式的使用方式来定义规则执行的时间。

示例：

Java 复制代码

```
import java.text.SimpleDateFormat
import java.util.Date
/*
    此规则文件用于测试timer属性
*/

rule "rule_timer_1"
    timer (5s 2s) //含义：5秒后触发，然后每隔2秒触发一次
    when
    then
        System.out.println("规则rule_timer_1触发，触发时间为：" +
                           new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()));
end

rule "rule_timer_2"
    timer (cron:0/1 * * * * ?) //含义：每隔1秒触发一次
    when
    then
        System.out.println("规则rule_timer_2触发，触发时间为：" +
                           new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()));
end
```

测试：

Java 复制代码

```java
//测试属性部分
    @Test
    public void test5() throws InterruptedException {
        //设置修改默认的时间格式
        System.setProperty("drools.dateformat","yyyy-MM-dd");
        KieServices kieServices = KieServices.Factory.get();
        KieContainer kieClasspathContainer =
kieServices.getKieClasspathContainer();
        final KieSession kieSession =
kieClasspathContainer.newKieSession();

        new Thread(new Runnable() {
            public void run() {
                //启动规则引擎进行规则匹配，直到调用halt方法才结束规则引擎
                kieSession.fireUntilHalt();
            }
        }).start();

        Thread.sleep(10000);
        //结束规则引擎
        kieSession.halt();
        kieSession.dispose();
    }
```

注意：单元测试的代码和以前的有所不同，因为我们规则文件中使用到了timer进行定时执行，需要程序能够持续一段时间才能够看到定时器触发的效果。