何时使用规则引擎？

并非所有应用程序都应使用规则引擎。如果业务逻辑代码包括很多 if-else 语句，则应考虑使用一个规则引擎。**维护复杂的 Boolean 逻辑**可能是非常困难的任务，而规则引擎可以帮助您组织该逻辑。当您可以使用声明方法而非命令编程语言表达逻辑时，变化引入错误的可能性会大大降低。

对客户的了解也是该决策的一个因素。在开发周期期间甚至部署之后添加和更改业务逻辑需求的倾向。

使用规则引擎场景特点：

单条规则相对简单

规则数量相对庞大

规则之间会有冲突

一条规则本身会触发另一条规则

规则有可能会产生变动（很多时候可能唯一需要变动的就是规则）

有些逻辑需要的参数我们并不能定义在规则中，而是在数据库表中进行配置。因此我们常见的业务逻辑层的开发，并不能先设计出一个数据模型，然后再在此基础上抽象逻辑。

因此我们发现Drools等规则引擎很难用

规则引擎:严格来说，它是一种嵌入到应用程序中的一个组件，能很好的把业务决策从应用程序框架中分离出来，然后使用预定义的方言（dialect）编写语义模块和业务决策模块，使用约定好的语法规范，接受用户的输入，然后解析用户的业务规则，然后根据解析好的业务规则，作出业务决策。

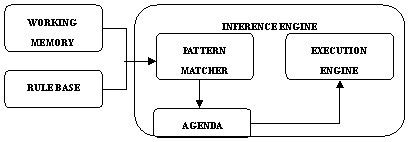
Drools是基于Java的规则引擎框架，是JBoss开源社区中的一个为Java量身定制的、基于RETE算法的产生式规则引擎的实现。大致的工作原理是，基于XML、DRL（Drools规则配置文件）的基础上，通过一个内置的解析器，把业务规则翻译成AST（Abstract Syntax Tree），最终会映射编译成Java的代码包，然后在程序运行的时候，加载这些代码包中的业务规则，并把在工作内存空间的规则和事实进行匹配，看下事实是否符合业务规则的约定

规则引擎与工作流引擎概念

业务流程执行语言(BPEL)

中央规则信息库

基于规则的专家系统（RBES）包括三部分：Rule Base（knowledge base）、Working Memory（fact base）和Inference Engine



工作流引擎现在演化为 业务过程管理，最基本的概念还是业务过程的流转，

业务规则侧重业务逻辑的控制，比如零售行业的价格规则，它会针对 客户 商品/品类 地理位置等设计一个非常复杂的价格管理体系；与业务规则非常紧密的是事件引擎，比如政府部门的应急处理系统

规则引擎使用了匹配规则的方式来进行，因此在应用这些规则引擎时。首先需要将我们具体应用中的业务逻辑做抽象，抽象成一条条规则之后，再打包成一个规则包。一个规则包相当于一个智能块。当数据传递给这个智能块后，系统会以匹配的方式应用满足条件的逻辑处理。

Drools is a Business Rules Management System (BRMS) solution

Drools Workbench (web UI for authoring and management) + Drools Expert (business rules engine)

https://www.drools.org/#

应用程序中业务逻辑最常见的方法是编写 Java 代码来实现需求文档的规则和逻辑

规则引擎试图降低应用程序业务逻辑的开发和维护的困难。可以将**规则引擎看作实现复杂业务逻辑的框架**。大多数规则引擎允许您使用声明性编程来表达对于某些给定信息或知识有效的结果

Drools 是用 Java 语言编写的开放源码规则引擎, Drools允许使用声明方式表达业务逻辑。可以使用非 XML 的本地语言编写规则

<http://blog.csdn.net/quzishen/article/details/6163012/>

接口：传参数，接口实现执行，获取结果

drools: 传递数据，规则检查和执行，获取结果

在drools,传递数据称为fact对象（java bean),当一个java bean插入到workingMemory中，规则使用的是原有对象的引用，规则通过对fact对象的读写，实现对应用数据的读写，对于其中的属性，需要提供getter setter访问器，规则中，可以动态的往当前workingMemory中插入删除新的fact对象。

规则文件: .drl

Drl中可以通过Import的方式引入Model类，也可以调用Java的各种函数，也可以自己定义Class and Function

Drl

package package-name

imports

globals

functions

queries

rules

Rules就是规则的部分，结构如下：

rule "name"

attributes

when

LHS(The Rule Language)

then

RHS(Java, Pthyon, Groovy)

End

每一条规则都有名称

每一条规则都有属性, 定义当前规则执行的一些属性等，比如是否可被重复执行、过期时间、生效时间等。

When定义当前规则的条件，等同于if里的条件判断，使用Drools自己的语法规则

Then当前规则条件满足后执行的操作,等同于if里的执行语句，可以写普通java代码，也可以直接调用Fact对象的方法来操作应用

***属性：***

**no-loop true**: 定义当前的规则是否不允许多次循环执行，默认是false

**lock-on-active true**：通过这个标签，可以控制当前的规则只会被执行一次，因为一个规则的重复执行不一定是本身触发的，也可能是其他规则触发的，所以这个是no-loop的加强版

**date-expires**：设置规则的过期时间，如"2011-01-31 23:59:59"

**date-effective**：设置规则的生效时间

**duration**：规则定时，duration 3000 3秒后执行规则

**salience**：优先级，数值越大越先执行，这个可以控制规则的执行顺序。

***条件：***

当前规则只有在条件都匹配的时候才会执行

Drools提供了十二中类型比较操作符：

**> >= < <= == != contains / not contains / memberOf / not memberOf /matches/ not matches**

如：

when

$customer:Customer()

$message:Message(status==0 || (status > 1 && status <=100))

当前规则只有在这三个条件都匹配的时候才会执行RHS部分

$message:Message(status==0 || (status > 1 && status <=100))：当前的workingMemory存在status为0或在[1 100]范围内的Ｍessage对象，这个对象通常是通过外部java代码插入或者自己在前面已经执行的规则的RHS部分中insert进去的。$message代表着当前条件的引用变量，在后续的条件部分和RHS部分中，可以使用当前的变量去引用符合条件的FACT对象，修改属性或者调用方法等

如果条件全部是 &&关系，可以使用“,”来替代

***执行：***

**调用Fact的方法：**如 $message.execute();操作数据库等等一切操作

调用Drools API方法：

**insert**：往当前workingMemory中插入一个新的Fact对象

**update**：更新

**modify**：修改

**retract**：删除

insert update modify and restract会触发规则的再次执行，除非使用no-loop限定；

即停止执行剩下的规则，重新执行所有规则

**调用规则文件自定义方法和类：**

function void console {

System.out.println();

StringUtils.getId(); // 调用外部静态方法

}

declare Address

@author(quzishen) // 元数据，仅用于描述信息

@createTime(2011-1-24)

city : String @maxLengh(100)

postno : int

end

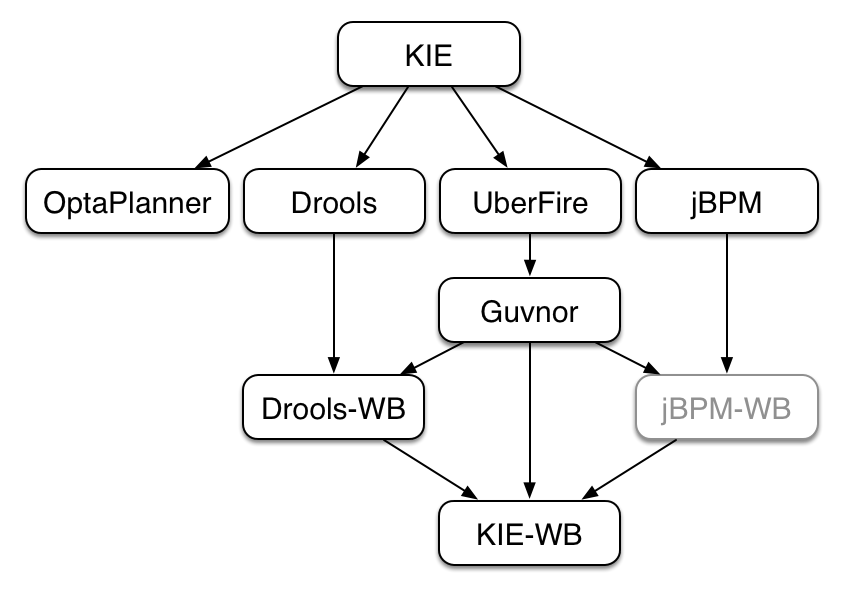
Address address = new Address();

Drools 环境

eclipse + Drools Engine + Drools and jBPM tools

<http://blog.csdn.net/yeomer/article/details/54291263>

Drools Documents



KIE (Knowledge Is Everything) = Drools + jBPM

Authoring of knowledge using a UI metaphor, such as: DRL, BPMN2, decision table, class models.

User interaction with the KieSession, via command line or UI.

System interaction with the KieSession, via API.

The loading of a JAR to provide a KIE session (KieSession), for which the application can interact with.

KIE exposes the JAR at runtime via a KIE container (KieContainer).

KieSessions, for the runtime's to interact with, are created from the KieContainer.

KIE will scan the classpath to find all the JARs with a kmodule.xml in it.

A Kie Project = Maven project + kmodule.xml (from which the KieBases and KieSessions that can be created)

all the Java sources and the Kie resources are compiled and deployed into the KieContainer which makes its contents available for use at runtime

kmodule.xml configure the KieBase(s) and KieSession(s)

KieBase contains all the application's knowledge definitions such as rules, processes, functions, and type models

KieSession are created from the KieBase into which data can be inserted and from which process instances may be started, the KieSession stores and executes on the runtime data. It is created from the KieBase or more easily can be created directly from the KieContainer if it has been defined in the **kmodule.xml** file

<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://www.drools.org/xsd/kmodule">

<kbase name="KBase1" default="true" declarativeAgenda="enabled" packages="org.domain.pkg1">

　<**ksession** name="KSession2\_1" type="stateful" default="true"/>

　　<ksession name="KSession2\_2" type="stateless" default="false" />

</kbase>

</kmodule>

//KieServices access all the Kie building and runtime facilities

KieServices kieServices = KieServices.Factory.get();

//KieContainer reads the files to be built from the classpath

KieContainer kContainer = kieServices.getKieClasspathContainer();

//retrieve the KieBases and KieSessions from the KieContainer using their names.

KieBase kBase1 = kContainer.getKieBase("KBase1");

KieSession kieSession1 = **kContainer.newKieSession**("KSession2\_1");

StatelessKieSession kieSession2 = kContainer.newStatelessKieSession("KSession2\_2");

例子1: 积分发放操作

https://wenku.baidu.com/view/b874963cf524ccbff0218402.html

发放积分可能伴随不同的运营策略和季节性调整，发放数目和规则完全不同，如果使用硬编码的方式去伴随业务调整而修改，代码的修改、管理、优化、测试、上线将是一件非常麻烦的事情，所以，将发放规则部分提取出来，交给Drools管理，可以极大程度的解决这个问题。

发放规则：

积分的发放参考因素有：交易笔数、交易金额数目、信用卡还款次数、生日特别优惠等。

定义规则：

1. 过生日，则加10分，并且将当月交易笔数翻倍后再计算积分 priority 1
2. 2011-某几个月，每月信用卡还款3次以上，每满3笔赠送30分
3. 当月购物总金额100以上，每100元赠送10分
4. 当月购物次数5次以上，每五次赠送50分
5. 特别的，如果全部满足了要求，则额外奖励100分
6. 发生退货，扣减10分
7. 退货金额大于100，扣减100分

**// fact**

**public** **class** Point {

**private** **boolean** isBirthday;

**private** **int** payNum;

**private** **double** buyAmount;

**private** **int** buyNum;

**private** **double** backAmount;

**private** **long** point;

}

**rules.point.AddPoint.drl**

**package** rules.point

**import** com.sample.point.Point;

**rule** "birth day"

**salience** 1

**lock-on-active** **true**

**when**

$p: Point(isBirthday == **true**)

**then**

$p.setPoint($p.getPoint()+10);

$p.setPayNum($p.getPayNum()\*2);

$p.setBuyAmount($p.getBuyAmount()\*2);

$p.setBuyNum($p.getBuyNum()\*2);

**update**($p);

**end**

**rule** "special month"

**date-effective** "1-April-2017"

**date-expires** "1-May-2017"

**when**

$p: Point(payNum >= 3)

**then**

$p.setPoint($p.getPoint() + $p.getPayNum()/3\*30);

**end**

**rule** "large amount"

**when**

$p: Point(buyAmount >= 100)

**then**

$p.setPoint($p.getPoint() + (**long**)$p.getBuyAmount()/100\*10);

**end**

**rule** "frequency buy"

**when**

$p: Point(buyNum >= 5)

**then**

$p.setPoint($p.getPoint() + $p.getBuyNum()/5\*50);

**end**

**rule** "large amount and frequency buy"

**when**

$p: Point(buyAmount>=100, buyNum>=5)

**then**

$p.setPoint($p.getPoint()+100);

**end**

**rules.point.SubstractPoint.drl**

**package** rules.point

**import** com.sample.point.Point;

**rule** " back"

**when**

$p: Point(backAmount>0)

**then**

$p.setPoint($p.getPoint()-10);

**end**

**rule** "large back"

**when**

$p: Point(backAmount>=100)

**then**

$p.setPoint($p.getPoint()-100);

**end**

**kmodule.xml**

**<**kbase name=*"rules"* packages=*"rules"*>

<ksession name=*"ksession-rules"*/>

</kbase>

<kbase name=*"point"* packages=*"rules.point"*>

<ksession name=*"ksession-rules-point"*/>

</kbase>

**Test.java**

// load up the knowledge base

KieServices ks = KieServices.Factory.*get*();

KieContainer kContainer = ks.getKieClasspathContainer();

KieSession kSession = kContainer.newKieSession("ksession-rules-point");

Point point = **new** Point();

point.setIsBirthday(**true**);

point.setPayNum(5);

point.setBuyAmount(500);

point.setBuyNum(5);

point.setBackAmount(100);

kSession.insert(point);

Message message = **new** Message();

message.setMessage("Hello World");

message.setStatus(Message.***HELLO***);

kSession.insert(message);

kSession.fireAllRules();

System.***out***.println("after rule");

System.***out***.println("the user's current point = " + point.getPoint());

注意：kSession　=> ksession-rules-point => rules.point => rules.point.AddPoint.drl, rules.point.SubstractPoint.drl

所以kSession workingMemeory里不论有什么fact,只能执行rules.point包下的规则文件

若要执行rules包下的规则文件，需要新开一个session

KieSession kSession = kContainer.newKieSession("ksession-rules");

kSession.insert(point);

kSession.insert(message);

kSession.fireAllRules();

当然把所有的规则放在一个包里面，session塞进所有的fact，从事执行包下面的所有规则文件，满足条件就触发规则