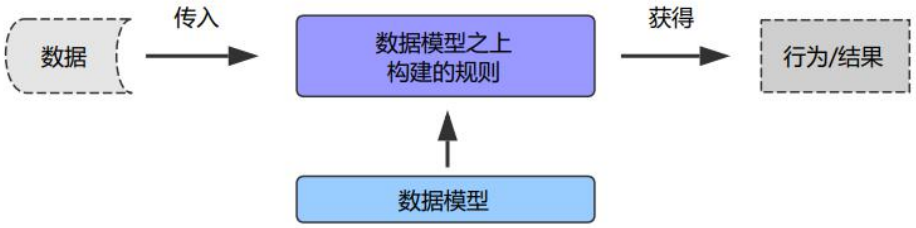
目录

作者： 季义钦，13 年毕业于南京大学计算机系，现为 DR 促销团队高级软件开发工程师。

一 需求或目标

目标：提供一组数据，以及基于这组数据编写的任意规则条件，快速得出结果。



主要包含几个部分：

1 任意数据模型；

2 基于数据模型构造的任意规则；

3 根据传入数据运行规则、得到正确的行为或结果；

二 数据和规则的抽象及举例

2.1 数据的抽象

数据可以抽象为两类：一类是基础数据，一类是具有时间特性的流式数据，第一类数据可以直接应用于表达式中，而第二类数据需要在其基础之上做 reduce 规约操作得到规约结果数据。下面以 DR lender 用户为例详细说明两类数据：

类别 1：基础数据

针对用户来说，基础数据包含用户 id、姓名、手机号、等级、注册渠道、注册时间、是否绑卡、是否实名、绑卡时间、实名时间、风险偏好、投资金额、投资团等单值数据。

类别 2：行为数据

针对用户来说，行为数据包含用户的投资、退团、注册、实名、邀请的人的投资记录等。这类数据的特征是具备时间特性。可以在其之上：(1)应用时间窗口(最近 x 天/y 次，当前自然日/周/月等)限制; (2)进一步用规则条件筛选数据; (3)最后做 count/sum/average 等 reduce 规约型计算得出“规约结果”。以用户的投资行为记录为例，规约结果如：

A. 最近 10 天的平均投资额；

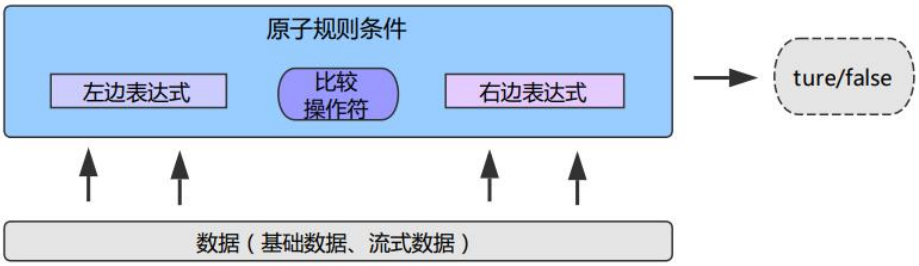
B. 投资大于 1w 的总次数；

C. 最近 30 天投资大于 2w 的总次数；
D. 最近 10 次投资大于 2w 的总次数；
E. 当前自然月的首投大于 1w；
F. 首次投资发生在注册后 20 天内；
以用户邀请的人的投资记录为例，规约结果为：
A. 所有单笔大于 5W、且在[A,B,C]团范围内的邀请投资的总和；
B. 所有邀请的注册时间满 20 天的人的投资的总和；
C. 从指定活动注册进来的、完成首投大于 1w 的人数；

2.2 规则的抽象

2.2.1 原子规则条件

前面已经介绍了“基础数据”和“行为数据的规约结果”两类数据，基于这些数据可以构造表达式，诸如%、+、-、*、/、甚至 java 库中数学运算 Math，commons 包的 DateUtils 等任意第三方库都可用于构造表达式。比较操作符诸如>、>=、<、<=、contains、not contains、start with、end with、between 等。左右表达式结合比较操作符构成一个原子的规则条件：



在“基础数据”上构建的原子规则条件如：
A. ${用户等级}等于 x 级；
B. ${注册时间}在<begin, end>之间；
C. ${是否完成实名绑卡}等于 true；
D. ${实名绑卡时间}发生在${注册时间}后 30 天内；
E. ${投资额}大于 x 元；
F. ${投资团范围}在(a,b,c,d)内；
G. ${用户手机号}以 181 开头；
H.“当前投资金额”大于“风险评级”的 10 倍；
在“行为数据的规约结果”上构建的原子规则条件如：
A. ${最近 10 天平均投资额}大于 1w。
B. ${投资大于 1w 的总次数}大于 5 次；
C. ${最近 30 天投资大于 2w 的总次数}大于 3 次；
D. ${最近 10 次投资大于 2w 的总次数}大于 3 次；
E. ${当前自然月的首投大于 1w 的次数}等于 1；
F. ${首次投资发生在注册后 20 天内的次数}等于 1；

需要注意的是单条行为数据本身也是一组简单数据，同样可以在其基础上按照基础数据的方式构建规则，比如想要判断用户是否满足“已实名用户近 20 天单次投资超过 5w 的总次数”大于 3，此规则可以抽象为先在投资行为记录上应用时间限制（最近 20 天）筛选、进一步做基础数据筛选（用户已经实名，且投资额大于 5w），最后做 count 计算得出表达

式所需的“行为数据规约结果”。

2.2.2 规则

规则组成

一条完整的规则由多个“原子规则条件”的 AND/OR 组成，规则通过后执行 action 集合。

优先级规则

一条简单的规则由多个原子规则条件组成，规则通过后执行对应的 action 集合。但有时候需要多条规则，通过其中一条就跳出去执行 action 集合，且先跑哪个规则要有优先级顺序。以给近期活跃用户发体验金奖励为例，以下几种情况都可以定义为活跃用户：

规则 1：一次性投资超过 5w、且已经完成风险测评；

规则 2：近 7 天投资次数超过 5 次；

规则 3：近 5 天跟帖次数超过 10 次且完成过至少一次投资；

以上几种情况完成任意一种都会发放体验金，但是满足其中一种，其他规则就不能继续执行，比如用户可能规则 1 和规则 2 同时满足，但是要求规则 1 通过后，其他规则就不能再被触发，这就要求规则之间支持优先级、且互斥存在。

延迟规则

存在这样的场景，一个规则由用户某个行为触发，但需要延迟一定的时间后再执行。比如要判断用户满足投资后 20 天内未发生过任何退团，则给其发放体验金奖励，或者判断用户注册后 30 天内没有任何投资，则给其发放代金券鼓励其投资。

这就要求支持延迟执行的规则，用户注册后触发延迟规则，在 30 天后运行此延迟规则判断用户是否没有任何投资行为。

三 自建规则引擎如何满足需求

3.1 需要什么？

针对规则引擎，需要满足如下几个需求：

1 易于理解、易于编写、且表达能力强的规则描述语言；

2 支持动态构造规则、并立即热部署生效；

3 支持从数据库等外部存储读取规则；

4 支持基础数据、流式数据上构建规则；

5 数据的计算和加载延迟到规则运行过程中，避免不必要的计算，且支持外部服务调用；

3.2 技术选型

实现规则引擎的方式有很多，可以采用原生的脚本语言如 Groovy、Python、jruby、javascript 等，好处是表达能力强、可动态修改并生效，但很容易让开发人员陷入规则中写业务的困境，后期的维护困难。更进一步说其毕竟不是为规则引擎而生、要形成规则引擎框架需要大量的工作。而开源的 java 规则引擎包括 Drools、Mandarax、JLisa、JEOPS 、Prova 、OpenRules 、Open Lexicon、SweetRules 、JRuleEngine 、Zilonis 、DTRules 、OpenL Tablets 等几十种，它们的好处是为规则引擎而生，已经为此做了大量工作，缺点在于学习成本。

我们决定站在巨人的肩上，选择基于社区活跃、文档丰富、功能强大的 Drools 来构建规则引擎，其最近一次发布是 2018/04/04 的 7.7.0 版。

Drools 本身能很好地支持支持 java 表达式、外部依赖数据运行时加载、基于时间的流式数据计算、规则互斥、规则优先级等功能，同时还和动态脚本语言一样可以动态生成、并动态执行，基于 Drools 构建的规则引擎框架可以完全满足 3.1 中提到的所有需求。

3.3 如何使用

步骤一：提供数据模型

以 lender 用户为例，首先定义其数据模型，如下表所示为部分数据模型定义：

（1）基础数据

用户 ID	aId	是否实名	hasRealName	当前行为	actionType
姓名	userName	是否绑卡	hasBindCard	当前投资额	investAmount
年龄	age	绑卡时间	bindCardTime	当前投资团	investPlan
电话	cellPhone	注册日期	registerTime	当前退团额	quitAmount
等级	level	注册活动	regActivityId	当前退出团	quitPlan
风险偏好	riskPrefe	注册渠道	regChannel	行为发生时间	triggerTime

（2）流式数据

投资行为数据流	STREAM_INVEST
邀请首投行为数据流	STREAM_REFEREE_FIRST_INVEST

步骤二：构造规则

然后就是基于数据模型构造规则，下面举几个规则的例子，其中**规则 1-4** 的数据主要是来自于前面提到的“基础数据”，**规则 5** 开始涉及到“流式数据的规约结果”，**规则 3 和规则 7** 演示了运行时加载外部数据：

规则 1：手机号 181 开头的用户、投资金额大于 1w 且排除特定的投团范围

```
//准备数据
Map<String, Object> data = new HashMap<>();
data.put("aId", 1110011);
data.put("actionType", "INVEST");
data.put("investAmount", 50000);
data.put("investPlan", "33222");
data.put("cellPhone", "18190800520");
data.put("level", 4);
//构造规则
DrCondition d1 = BaseCondition.newBaseCondition(new MetaCondition( left: "${actionType}", CompareMethod.EQUAL, Arrays.asList( ...a: "INVEST")));
DrCondition d2 = BaseCondition.newBaseCondition(new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000)));
DrCondition d3 = BaseCondition.newBaseCondition(new MetaCondition( left: "${investPlan}", CompareMethod.NOT_IN, Arrays.asList( ...a: "33221", "33222")));
DrCondition d4 = BaseCondition.newBaseCondition(new MetaCondition( left: "${cellPhone}", CompareMethod.STR_STARTS_WITH, Arrays.asList( ...a: "181")));
//设置action
DrRule rule = new DrRule( id: "testRule1");
rule.setConditions(Arrays.asList(Arrays.asList(d1, d2, d3, d4)));
rule.setActions(Arrays.asList(new SendAwardAction( awardName: "award1"), new SendAwardAction( awardName: "award2")));
//运行规则
drRuleEngine.runRuleEngine(data, Arrays.asList(rule));
```

这里演示了从准备数据，构造规则，到运行规则的整个过程。规则部分由“原子规则条件”组成，然后是设置规则通过后要执行的 Action 集合，最后运行规则。需要注意的是数据需用${}标记，接下来演示更多更强大的规则，将只贴出来具体规则部分的代码。

规则 2：投资额为偶数、且投资额大于年龄的 30 倍和用户等级的 100 倍中最小的那个值

```
//任意java表达式支持
DrCondition d5 = BaseCondition.newBaseCondition(new MetaCondition( left: "${investAmount} % 2", CompareMethod.EQUAL, Arrays.asList( ...a: 0)));
DrCondition d6 = BaseCondition.newBaseCondition(new MetaCondition( left: "${investAmount}", CompareMethod.GRATER,
        Arrays.asList( ...a: "Math.min(${age} * 30, ${level} * 100)")));
```

比较的左右两边都可以支持任意${}数据、以及任意 java 表达式，包括 java.lang 包、java.util 包等一切可以引入的 jar 包功能。

规则 3：注册时间在指定范围内、已经完成实名、且在注册后的 20 天内完成绑卡

```
//运行时计算并加载外部数据
DrCondition drCondition7 = BaseCondition.newBaseCondition(new MetaCondition( left: "${registerTime}", CompareMethod.BETWEEN,
        Arrays.asList( ...a: "2018-01-01 00:00:00", "2018-06-01 00:00:00")));
DrCondition drCondition8 = BaseCondition.newBaseCondition(new MetaCondition( left: "${registerChannel}", CompareMethod.IN,
        Arrays.asList( ...a: "channelA", "channelB")));
DrCondition drCondition9 = BaseCondition.newBaseCondition(new MetaCondition( left: "${bindCardTime}", CompareMethod.LESS_AND_EQUAL,
        Arrays.asList( ...a: "DateUtils.addDays(${registerTime}, 20)")));
```

运行时一旦发现${}数据是传入的数据集合中不存在的，则会尝试加载以该数据变量命名的外部数据加载器计算并加载数据到规则内存中，这里 registerTime、registerChannel 等数据都是当前传入的数据中没有的，所以会调用对应的数据加载器加载数据到规则内存中，我们需要做的就是针对 registerTime、registerChannel 编写对应的数据加载器：

```
@Component
public class RegisterTimeDataInitializer extends AbstractDataInitializer {
    @Override
    public String dataKey() { return "registerTime"; }

    @Autowired
    private UserInfoRepository userInfoRepository;

    @Override
    public Object initialize(Map<String, Object> data) throws Exception {
        if (!data.containsKey("aId") || data.get("aId") == null) {
            return null;
        }
        //search user info from database
        return userInfoRepository.getUserInfoByAid(data.get("aId").toString()).getRegisterTime();
    }
}
```

规则 4：完成实名、或完成绑卡

```
//"或"关系支持
MetaCondition metaConditionA = new MetaCondition( left: "${hasRealName}", CompareMethod.EQUAL, Arrays.asList( ...a: true));
MetaCondition metaConditionB = new MetaCondition( left: "${hasBindCard}", CompareMethod.EQUAL, Arrays.asList( ...a: true));
DrCondition drCondition10 = BaseCondition.newBaseCondition(metaConditionA, metaConditionB);
```

注意，下面从规则 5 开始，会演示表达式中的数据来自“流式数据的规约结果”。

规则 5：用户大于 1w 的投资的次数、大于 3 次

```
//流式数据支持
MetaCondition filterCondition11 = new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000));
StreamCondition.Reduce reduce1 = StreamCondition.newReduce( streamKey: "STREAM_INVEST",  duration: null, Arrays.asList(filterCondition11),
        ReduceType.COUNT,  reduceKey: null);
DrCondition drCondition11 = StreamCondition.newStreamCondition(reduce1, new MetaCondition( left: "${reduceValue}", CompareMethod.GRATER,
        Arrays.asList( ...a: 3)));
```

前面的规则都是基于基础数据的比较，而这里是针对流式数据的操作，正如前面介绍的，首先执行 reduce 操作，包括筛选用户投资记录数据流中投资额超过 1w 的记录，并执行 count 操作，然后与 condition 进行比较，判断次数是否大于三次。

需要注意的是，流式数据也是外部依赖数据，并不是在规则执行前加载到规则内存中的，而是需要运行时动态加载，所以我们需要针对“STREAM_INEST”编写对应的流式数据加载器，根据 reduce 操作传入的 Duration 限制(包括时间和长度的限制)返回数据流，流式数据加载器的例子参见规则 7。

规则 6：用户大于 1w 的投资的总额、大于 5w

```
//流式数据支持
MetaCondition filterCondition21 = new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000));
StreamCondition.Reduce reduce2 = StreamCondition.newReduce( streamKey: "STREAM_INVEST", duration: null, Arrays.asList(filterCondition21),
        ReduceType.SUM, reduceKey: "${InvestAmount}");
DrCondition drCondition12 = StreamCondition.newStreamCondition(reduce2, new MetaCondition( left: "${reduceValue}", CompareMethod.GRATER,
        Arrays.asList( ...a: 50000)));
```

规则 7：最近 7 天投资额大于 1w 的总次数、大于 3 次

```
//带时间限制的流式数据支持
MetaCondition filterCondition31 = new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000));
StreamCondition.Reduce reduce3 = StreamCondition.newReduce( streamKey: "STREAM_INVEST", new Duration(DurationType.LAST_DAYS, value: 7),
        Arrays.asList(filterCondition31), ReduceType.COUNT, reduceKey: null);
DrCondition drCondition13 = StreamCondition.newStreamCondition(reduce3, new MetaCondition( left: "${reduceValue}", CompareMethod.GRATER,
        Arrays.asList( ...a: 3));
```

可以对流式数据增加时间窗口限制，其他支持的时间窗口包括最近 x 秒、最近 x 分钟、最近 x 小时、最近 x 月、最近 x 年、当前自然周、当前自然月、当前自然年等。下面是用户投资流式数据加载器实现，所有类型时间限制都会转换成<beginDate、endDate、limit>三个字段，加载器根据这三个字段的值查询数据返回。

```
@Component
public class InvestStreamDataFilter extends AbstractStreamFilter {
    @Override
    public String streamKey() { return "STREAM_INVEST"; }

    @Autowired
    private UserInvestMessageRepository uimRepository;

    @Override
    public List<Map<String, Object>> filter(Map<String, Object> data, Date beginDate, Date endDate, Integer limit)
        if (!data.containsKey("aId") || data.get("aId") == null) {
            return null;
        }
        //search invest records from dataBase
        List<InvestMessage> list = uimRepository.select(data.get("aId").toString(), beginDate, endDate, limit);
        return list.stream().map(m -> {
            Map<String, Object> map = new HashMap<>();
            map.put("investAmount", m.getInvestAmount());
            return map;
        }).collect(Collectors.toList());
    }
}
```

规则 8：存在发生在注册后 20 天内、且投资金额大于 1w 的首次投资

```
//带时间限制的流式数据支持
MetaCondition filterCondition41 = new MetaCondition( left: "${investTime}", CompareMethod.LESS_AND_EQUAL,
        Arrays.asList( ...a: "DateUtils.addDays(${registerTime}, 20)"));
MetaCondition filterCondition42 = new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000));
StreamCondition.Reduce reduce4 = StreamCondition.newReduce( streamKey: "STREAM_INVEST", new Duration(DurationType.AHEAD_LENGTH, value: 1),
        Arrays.asList(filterCondition41, filterCondition42), ReduceType.COUNT, reduceKey: null);
DrCondition drCondition14 = StreamCondition.newStreamCondition(reduce4, new MetaCondition( left: "${reduceValue}", CompareMethod.GRATER,
        Arrays.asList( ...a: 0));
```

除了最近 x 天这样的时间限制，还有最近 x 次、最开头的 x 次这样针对流式数据长度的限制，这里取的是用户投资记录中的首次投资。

规则 9：存在当前自然周内首次、且投团范围在指定团范围内、金额大于 1w 的投资

```
//带双重时间限制的流式数据支持
MetaCondition filterCondition51 = new MetaCondition( left: "${investPlan}", CompareMethod.IN, Arrays.asList( ...a: "33221", "33222", "11891"));
MetaCondition filterCondition52 = new MetaCondition( left: "${investAmount}", CompareMethod.GRATER, Arrays.asList( ...a: 10000));
StreamCondition.Reduce reduce5 = StreamCondition.newReduce( streamKey: "STREAM_INVEST", new Duration(DurationType.NATURE_WEEK_AHEAD_LENGTH, value: 1),
        Arrays.asList(filterCondition51, filterCondition52), ReduceType.COUNT, reduceKey: null);
DrCondition drCondition15 = StreamCondition.newStreamCondition(reduce5, new MetaCondition( left: "${reduceValue}", CompareMethod.GRATER,
        Arrays.asList( ...a: 0));
```

可以看到 reduce 操作支持的时间限制还包括时间+长度。

3.4 实现原理

规则引擎执行的入口即前面提到的 drRuleEngine.runRuleEngine(data,DrRule 集合)，大致思路是将 DrRule 规则集合转换为 Drools 规则语言并执行，大致分为如下步骤：

//步骤 1：将规则集合转换为 DRL 规则语言并 add 到 KieHelper；

//步骤 2：基于 KieHelper 和 KieBaseConfiguration 构建知识库 KieBase；

//步骤 3：基于知识库 KieBase 创建到 Drools 引擎的会话 KieSession；

//步骤 4：插入 Fact 数据，设置规则收集器、外部数据加载器等为全局变量；

//步骤 5：执行规则，得到运行通过的规则集合；

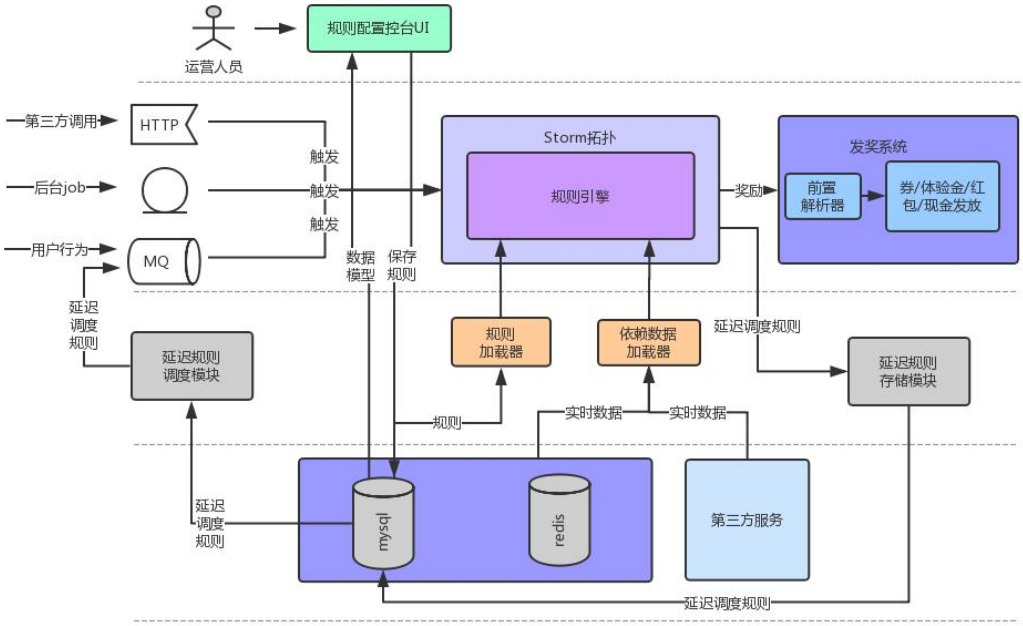
//步骤 6：收集步骤 5 中得到的规则集合所属的 action 集合，依次执行或返回；

如何将 DrRule 规则集合转换为 Drools 规则语言，通过 Pattern、操作符、java 表达式、from、accumulate、salience、activation-group 等丰富的 Drools 功能支撑我们的需求至关重要。另外可以考虑将规则的加载，到转换为知识库 KieBase 的过程异步化，而不是每次运行规则都实时构建知识库，这样有助于提升性能，后面介绍的基于自建规则引擎构建的促销系统就是这样做的。下面是运行时从 DrRule 规则集合动态生成的 Drools 规则语言的片段：

```
import org.apache.commons.lang3.time.DateUtils;
global java.util.Set passRuleSet;
global com.example.drools.engine.engine.initializer.GeneralDataInitializer dataInitializer;
global com.example.drools.engine.engine.initializer.StreamDataFilter dataFilter;
rule "drools_rule_123412"
when
$data: Map();
Map(this["actionType"] == "INVEST");
Map(this["investAmount"] > 10000);
Map(this["investPlan"] in ("33221", "33222"));
Map(this["cellPhone"] str[startsWith] "181");
Map(this["registerTime"] > DateUtils.parseDate("2018-01-01 00:00:00", "yyyy-MM-dd HH:mm:ss") &&
Map(this["registerChannel"] in ("channelA", "channelB")) from dataInitializer.initialize($data, "registerC
Map(this["bindCardTime"] <= DateUtils.addDays(this["registerTime"], 20)) from dataInitializer.initialize
Map(this["hasRealName"] == true || this["hasBindCard"] == true) from dataInitializer.initialize($data, "I
Map(this["investAmount"] > 10000);
Number(doubleValue > 50000) from accumulate(
   Map(this["investAmount"] > 10000, $investAmount: this["investAmount"]) from dataFilter.filter($dat
   sum($investAmount));
Number(doubleValue > 3) from accumulate(
   Map(this["investAmount"] > 1000) from dataFilter.filter($data, "STREAM_INVEST", "LAST_DAYS", 7, r
   count(1));
Number(doubleValue > 0) from accumulate(
   Map(this["investTime"] <= DateUtils.addDays(this["registerTime"], 20), this["investAmount"] > 1000
   count(1));
then
passRuleSet.add("rule_123412");
end
```

四 基于自建规则引擎构建促销系统

如下图所示为促销系统规则引擎和发奖相关的架构图：



1.规则配置控台

从数据库读取支持的数据模型呈现给运营人员，运营人员基于数据模型，通过 UI 界面创建规则及对应的奖励，创建好的规则将存储到数据库中。

前面已经提到了 lender 用户的数据模型，展示到 UI 提供给运营人员的就是数据模型的中文描述，如用户${注册渠道}、${注册时间}、${投资额}、${投资团等}，操作符如大于、大于等于/小于/小于等于/包含于/闭区间/开区间等，运营人员只需在界面上选择对应的数据、操作符、以及创建对应的奖励，即可生成对应规则。

2.规则引擎

来自 MQ 的用户行为、来自定时 Job 的调用、或来自第三方系统的 http 调用都可以触发规则引擎实时加载并执行对应的规则。

3.延迟规则存储和加载模块

针对一些需要延迟触发规则二次执行的特殊场景，在触发行为发生后，将需要延迟调度的规则存储到库中，由延迟规则调度模块在正确的时间触发规则引擎执行。

4.发奖系统

发奖系统只业务落地的核心，当用户满足特定规则后，需要根据配置发放对应奖励，这里面包括优惠券、体验金、现金、DR 币等的发放，甚至包含 app/短信/微信等不同渠道的用户消息促达。

促销系统的核心不是规则引擎，而是数据、以及运营模式，规则引擎只是工具。如何构

造并完善用户数据模型，如何让运营人员快速方便地构建，修改规则，并能够立刻生效，如何让开发人员快速响应新的业务需求开发新的条件和调整已有条件，才是更应该去思考的。