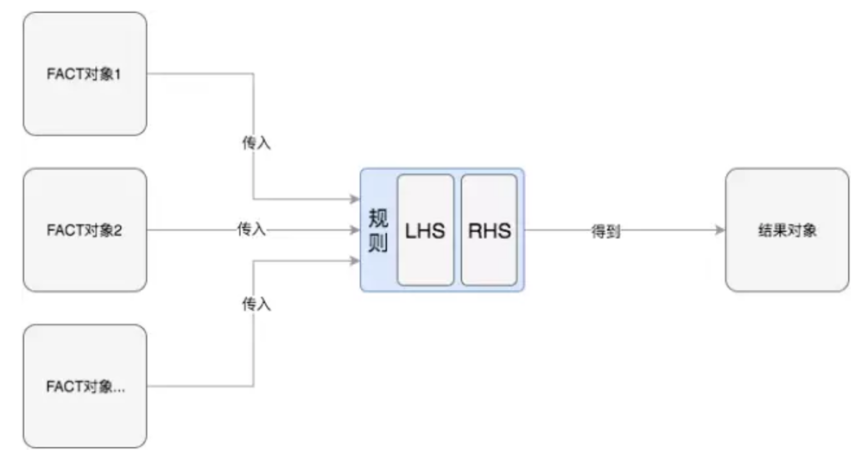
规则本质上是：

LHR -> RHR

~~Condition-1 &&Condition-2 &&... &&Condition-n -> actions~~

~~其中condition-n可表示为：fn(data-n) compare-method condition-n-data~~

~~其中compare-method包含==，>=, >, <=, <, contains等~~



~~主要由三部分构成：~~

* ~~FACT对象：用户输入的事实对象，作为决策因子使用。~~
* ~~规则：LHS（Left Hand Side）部分即条件分支逻辑。RHS（Right Hand Side）部分即执行逻辑。LHS和RHS部分是由一个或多个模式构成的。模式是规则内最小单位。模式的输入参数可以是另一个模式或FACT对象（比如逻辑与运算[参数1] && [参数2]中参数1可以是另一个表达式）。模式需要支持以下3种类别：~~
  + ~~客户定义方法：FACT对象的实例方法、静态方法。~~
  + ~~常规表达式：逻辑运算、算数运算、关系运算、对象属性处理等。~~
  + ~~结构化查询。~~
* ~~结果对象：规则处理完毕后的结果。需要支持自定义类型或者简单类型（Integer、Long、Float、Double、Short、String、Boolean等）。~~

一、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”

这里面包含两个需求：

1 提供任意数据，在数据之上构建任意规则条件

2 构建的规则条件热部署，并能够根据输入数据和规则条件执行正确的行为

比如存在一组刻画用户各方面信息的数据，如用户等级、投资额、注册渠道等，运营人员可能随时想要构建这组数据上的任意规则，比如等级为5，投资额大于1w，注册渠道在(A,B,C)范围内，则给该用户发送特定奖励。

1. 数据和规则的抽象

2.1 数据的抽象

以DR用户为例，说明数据可以如何抽象。

（1）基础数据

针对用户来说，基础数据包含用户等级、注册渠道、投资金额、贷款金额、投资团等，基于这些基础数据可以构建的规则条件包括投资额大于x元，贷款金额大于y元，投资团范围在(a,b,c,d)内，注册渠道等于dianrong1。

规则的判断就是输出一组基础数据，判定这组数据是否满足在其之上构建的规则。

（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、实名记录等行为记录。这类数据的特征是具备线性时间特性，可以在其之上应用时间窗口(最近x天/y次，当前自然日/周/月等)筛选数据，以及做count/sum/average等规约型的计算。以用户的投资行为记录为例，可以衍生出诸如如下类型的条件：

1. 最近10天投资次数；
2. 投资大于1w的次数；
3. 最近10天投资大于2w万的次数；
4. 最近30次投资大于5w的次数；

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计算得出表达式左边值。

由此规则可以抽象为：

基础数据：data = <d1=v1, d2=v2, d3=v3,..., dn=vn>

流式数据：data = [s1,s2,...sn] = [<e1=v1, e2=v2, e3=v3,..., en=vn, timestamp>, ...]

基础规则：condition(dn) = valueOf(dn) compare with((>,>=,<,<=, members of) ) valueOf(condition(dn))

流式规则：duration([s1,s2,...sn]).foreach(condition(sn)).count/sum

规则的抽象：

针对规则还需要支持：

1 优先级规则

2 互斥规则

2.1 执行相同action：

Person( age > 30 && < 40 || location == "london" )

2.2 执行不同的action：

比如给近期活跃用户体验金奖励，以下几种情况都可以定义为活跃用户：（1）一次性投资超过5w且已经完成风险测评；（2）近10天投资次数超过5次；（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，比如满足情况1的有可能情况2也满足，但是情况1的规则通过后，其他规则就不能再被触发。

===============================================

支持的功能：

1 活动（规则组）为载体，包含多个有关联关系的规则，一个规则由条件集合和行为集合组成

2 一个活动内多个规则具备优先级，可以提前结束(实现互斥)

3 一个规则内条件通过后，可以随机、所有、轮训等方式执行行为集合

4 活动面向的人群因子，活动执行的方式(MQ触发、定时触发、HTTP触发等)等属性

典型场景举例：

我们以实际的例子来说明需要支持哪些功能。

基础条件：

需要计算的基础条件：

是否首投、~~累计投资金额小于x元~~、首次投资且发生在注册后20天内、是否已经实名

采用LHS中引入service变量实现。

一定要懒计算（需要时计算），而不是在跑规则前将所有条件需要的数据预先计算好。

基础条件的延伸：

~~实际上是针对时间段的筛选：~~

行为：

1 用户投资、退团、注册、实名记录

2 用户推荐的投资、退团、注册、实名记录

基于用户，数据量可控。

存储到redis，设置阈值，比如1年过期。



最后三个通过自定义accumulate函数实现。

================================

基础条件集合 + 行为集合组成一条规则。

针对规则还需要支持：

1 优先级规则

2 互斥规则

2.1 执行相同action：

Person( age > 30 && < 40 || location == "london" )

2.2 执行不同的action：

比如给近期活跃用户体验金奖励，以下几种情况都可以定义为活跃用户：（1）一次性投资超过5w且已经完成风险测评；（2）近10天投资次数超过5次；（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，比如满足情况1的有可能情况2也满足，但是情况1的规则通过后，其他规则就不能再被触发。

~~区间规则~~

~~投资额在[a,b]之间发100元满减券~~

~~投资额在[b,c]之间发200元满减券~~

3 带时间特性的规则：

30min内浏览某个团产品超过3次（浏览团触发近30min浏览次数计算）

10天内退团总金额大于5w（退团行为触发近10天退团总额计算）

注册后30天内没有任何投资（X行为后Y行为判定）（注册行为触发定时30天后判断）这个场景需要延迟timer支持

退团前24小时未浏览过A产品（Y行为前X行为判定）（退团行为触发近24小时浏览次数计算）

规则有了，规则的触发时间点：

1 用户行为实时触发

2 延迟触发

3 周期性触发（不是运营指定用户，针对对指定用户群体的运营）

activation-group

默认值：N/A

类型：String

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。

salience

默认值：0

类型：integer

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. 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.

滑动窗口

Sliding Windows are a way to scope the events of interest by defining a window that is constantly moving. The two most common types of sliding window implementations are time based windows and length based windows.

定时器

Rules now support both interval and cron based timers, which replace the now deprecated duration attribute.

Example 132. Sample timer attribute uses

timer ( int: <initial delay> <repeat interval>? )

timer ( int: 30s )

timer ( int: 30s 5m )

timer ( cron: <cron expression> )

timer ( cron:\* 0/15 \* \* \* ? )

Accumulate CE (preferred syntax)

The top level accumulate syntax is the most compact and flexible syntax. The simplified syntax is as follows:

accumulate( <source pattern>; <functions> [;<constraints>] )

==================================================

Storm准备数据，实时更新数据模型（用户画像）

规则引擎 vs 动态语言

规则引擎的实现方式很多：

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。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。

另外开源的java规则引擎包括Drools、OFBiz Rule Engine、JLisa 、Algernon、TyRuBa、JTP、JEOPS、InfoSapient、RDFExpert、Jena 2、Euler、JLog、Pellet OWL Reasoner、Prova、OpenRules、SweetRules、JShop2等等。

重要的不是规则有多灵活，而是在于针对自己的业务形成一套框架，规则语言再灵活，让运营人员编写规则都是不现实的。我们要方便运营人员快速新增自己的规则，开发人员能够按照统一的规范，轻松开发出对应的规则。用简单的原子规则组成复杂的场景。因此最重要的是模式。

最终呈现出来应该是这样：

1 开发人员根据业务编写受版本控制的fn(data-n)

用户年龄，注册日期，投资金额等不需要复杂计算，直接传入即可，这类数据无需开发直接提供给运营人员选择，但用户的

2 运营人员通过Web UI构建规则

通过UI操作运营人员组合已有条件形成规则，以及规则满足后执行的动作集合。

3 规则发布，规则引擎加载、执行规则，根据结果执行actions

选择规则引擎的标准：

1业务的快速响应

可以动态调整规则，并立即热加载生效，达到与动态语言同样的效果。

2 支持从数据库读取规则

规则存入数据库，更加容易管理，且由运营人员通过Web UI操作快速构建和调整业务规则。

2执行效率

3灵活性

4社区活跃度

6学习成本

7后期维护

8版本管理

~~使用Java,Groovy等原生语言来做判断和执行语句，让程序员很容易过渡、移植，学习曲线很低。~~

~~使用Java，Groovy等原生语言编写规则，很容易让开发人员陷入规则中写业务的困境，后期的维护较为困难。~~

~~很多业务系统一般会使用某种脚本语言来实现，比如groovy，jruby，javascript，原因很简单单，因为脚本语言本质上可以解决变量识别和自动运行的问题，而且扩展性强，表达能力强，动态部署生效。~~

动态语言的好处在于不需要像静态语言那样预先写好，而是可以在程序运行时间动态拼装，根据数据生成规则，达到较高的灵活性。而灵活性并不是规则引擎的重要指标，毕竟我们不是为了能够动态构造一堆复杂的业务代码，更重要的是形成自己的一套数据标准，在此基础上开发人员和运营人员能够很方便地增加业务规则。

选择drools

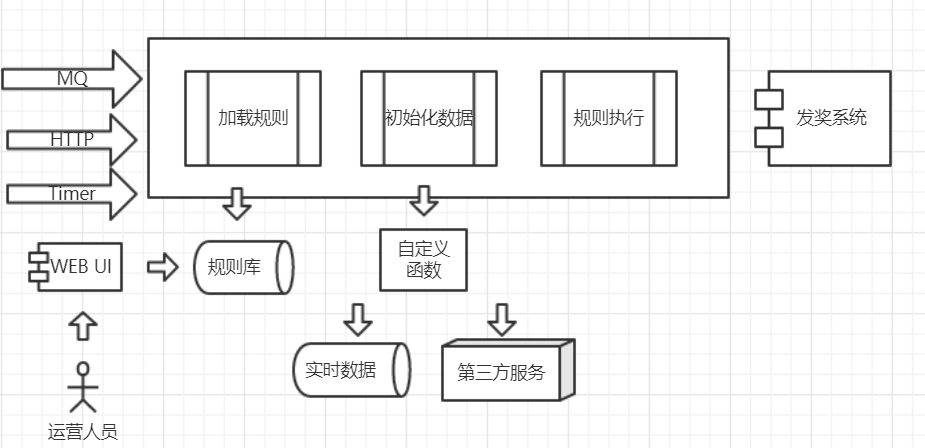
做规则引擎自研的公司，一般会把学习成本太高，二次开发困难，编写新的规则需要开发人员介入，

架构图：

规则库

规则引擎

发奖系统



记住，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。