一、产生的背景

生产过程中，线上的业务规则内嵌在系统的各处代码中，每次策略的调整都需要更新线上系统，进行从需求->设计->编码->测试->上线这种长周期的流程，满足不了业务规则的快速变化以及低成本的更新试错迭代。

因此需要有一种解决方案将**商业决策逻辑**和应用开发者的**技术决策**分离开，在系统运行时能去更新管理业务规则。

规则引擎(业务规则管理系统，英文名为BRMS（即Business Rule Management System））正是这样的解决方案。

二、实际业务场景：

一个小例子： 假如我们有个业务场景，当客户的积分位于一个区间A时，我们给予他一个头衔a，当一个客户的积分位于区间B时，我们给予他一个头衔b，当客户的积分位于区间C时，我们给予他一个 头衔c。如果我们使用if-else-then来写，是可以实现的，但是这里存在一个问题：规则定义和代码耦合在一起了。如果我们改变规则，把区间A，B，C 改成D，E，F，又或是将规则增加，从3组变为100组，那么我们改代码实在是太麻烦了。这时候规则引擎就派上用场了，我们希望把**规则和代码解耦**，形成一个规则引擎，以适应复杂多变的业务场景，或者更加精细化的运营。

🐵**实际业务场景：**

**BD绩效考核**是将各个战区每月针对bd制定的绩效考核方案线上化，支持绩效方案设置、bd绩效达成情况跟踪、绩效薪资计算等功能。

**现状：**绩效的考核方案和激励政策呈现出多样化、复杂化的特点，不同的战区绩效考核的指标及考核方案不同，每个战区每个月的指标及考核方案变化非常大，同一个指标每月对应的规则定义不同。

**方案：**为了应对战区绩效考核方案多变，本次需要实现指标规则的灵活配置。

**以有效团长数为例：**

**有效团长定义为：**团长每月有效团达标天数>=15天（有效团指的是每天成团的标准：5个下单用户，15件下单商品）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指标类型 | 指标名称 | 指标规则 | 实体 | 时间维度 |
| 原子指标 | 团用户数 | 团成交用户数 | 团 | 日 |
| 原子指标 | 团商品数 | 团成交商品数 | 团 | 日 |
| 复合指标 | 是否有效团 | 团用户数>=5且团商品数>=15 | 团 | 日 |
| 复合指标 | 有效团天数 | 是否有效团=1，求和sum | 团长 | 月 |
| 复合指标 | 是否有效团长 | 有效团天数>=15天 | 团长 | 月 |
| 复合指标 | 有效团长数 | 是否有效团长=1，求和sum | 组织/网格 | 月 |

**实际场景来源于：**[https://cf.jd.com/pages/viewpage.action?pageId=637690194](https://cf.jd.com/pages/viewpage.action?pageId=637690194" \t ")﻿

三、规则引擎价值及本质

3.1 规则引擎价值：

最大的价值就在于通过以下的三个过程，大大的缓解了频繁的需求变化给整个业务系统带来的灾难。

逼迫系统开发人员和业务专家梳理业务，定义统一的BOM（业务对象模型）。

业务专家可以快速的制定修改规则，然后交由规则引擎自动化地来处理分析。

规则引擎代替系统开发人员，解决由规则条件关联动作变化带来的开发工作。

**总结一句话：规则引擎就是将需要外部决策的业务规则加载到系统中，按照不同的输入条件进行不同的规则匹配组合后，执行符合规则的一个或者多个操作。**

3.2 规则引擎的本质

本质是“**专家决策系统规则引擎模型”和rete算法，为了解决的是大量重复的condition匹配效率的问题，以及规则冲突规范的问题，和脚本的性能比较不在同一个层面上。**

四、 Rete算法介绍

（1）规则内容

IF：

年级是三年级以上，

性别是男的，

年龄小于10岁，

身体健壮，

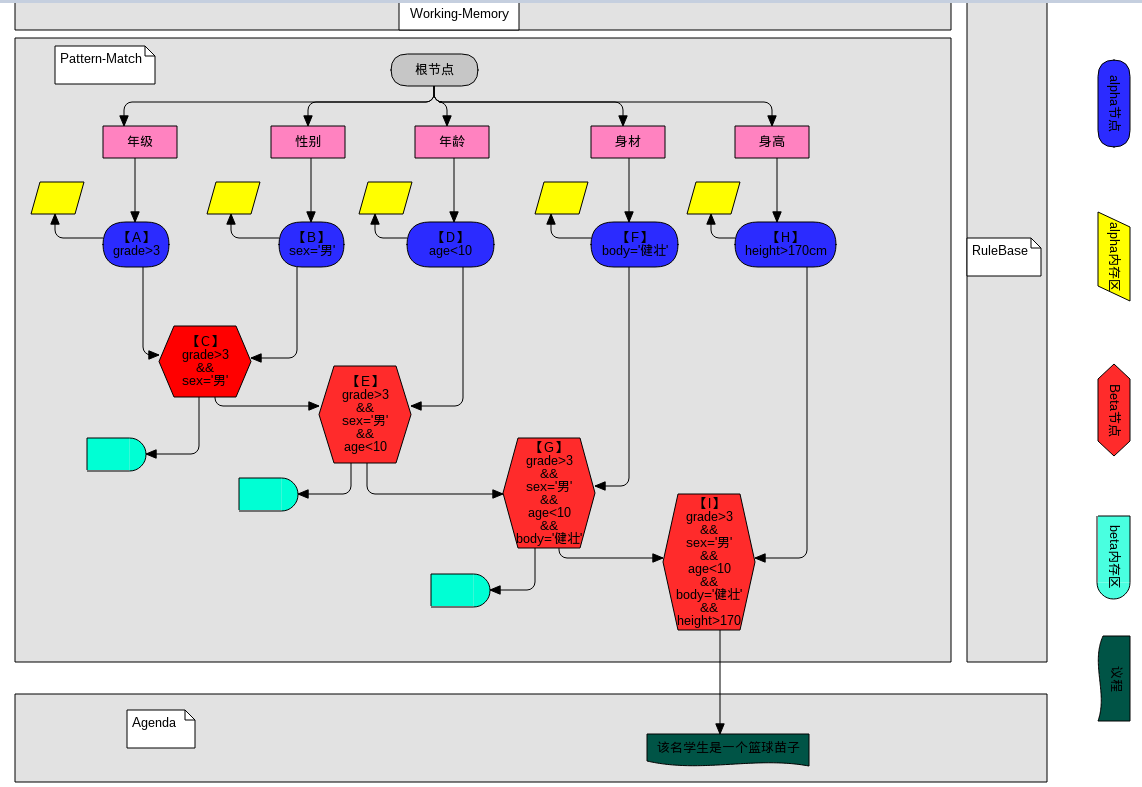
身高170cm以上，

THEN：

这个男孩是一个篮球苗子，需要培养

**网络构建：**

﻿



﻿﻿

**匹配过程：**

（1）匹配过程中事实在网络节点中的流转顺序为A-->B-->C-->D-->E-->F-->G-->H-->I--->规则匹配通过

（2）从working-Memory中拿出一个待匹配的StudentFact对象，进入根节点然后进行匹配，以下是fact在各个节点中的活动图

A节点：拿StudentFact的年级数值进行年级匹配，如果年级符合条件，则把该StudentFact的引用记录到A节点的alpha内存区中，退出年级匹配。

B节点：拿StudentFact的性别内容进行性别匹配，如果性别符合条件，则把该StudentFact的引用记录到B节点的alpha内存区中，然后找到B节点左引用的Beta节点，也就是C节点。

C节点：C节点找到自己的左引用也就是A节点，看看A节点的alpha内存区中是否存放了StudentFact的引用，如果存放，说明年级和性别两个条件都符合，则在C节点的Beta内存区中存放StudentFact的引用，退出性别匹配。

D节点：拿StudentFact的年龄数值进行年龄条件匹配，如果年龄符合条件，则把该StudentFact的引用记录到D节点的alpha的内存区中，然后找到D节点的左引用的Beta节点，也就是E节点。

E节点：E节点找到自己的左引用也就是C节点，看看C节点的Beta内存区中是否存放了StudentFact的引用，如果存放，说明年级，性别，年龄三个条件符合，则在E节点的Beta内存区中存放StudentFact的引用，退出年龄匹配。

F节点：拿StudentFact的身体数值进行身体条件匹配，如果身体条件符合，则把该StudentFact的引用记录到D节点的alpha的内存区中，然后找到F节点的左引用的Beta节点，也就是G节点。

G节点：G节点找到自己的左引用也就是E节点，看看E节点的Beta内存区中是否存放了StudentFact的引用，如果存放，说明年级，性别，年龄，身体四个条件符合，则在G节点的Beta内存区中存放StudentFact的引用，退出身体匹配

H节点：拿StudentFact的身高数值进行身高条件匹配，如果身高条件符合，则把该StudentFact的引用记录到H节点的alpha的内存区中，然后找到H节点的左引用的Beta节点，也就是I节点。

I节点：I节点找到自己的左引用也就是G节点，看看G节点的Beta内存区中是否存放了StudentFact的引用，如果存放了，说明年级，性别，年龄，身体，身高五个条件都符合，则在I节点的Beta内存区中存放StudentFact引用。

同时说明该StudentFact对象匹配了该规则，形成一个议程，加入到冲突区，执行该条件的结果部分：该学生是一个篮球苗子。

五、Rete算法优劣势分析

5.1 Rete算法优于传统的模式匹配算法

a．**状态保存。** Rete 算法是一种启发式算法，不同规则之间往往含有相同的模式，因此在 beta-network 中可以共享 BetaMemory 和 betanode。如果某个 betanode 被 N 条规则共享，则算法在此节点上效率会提高 N 倍。

b. **节点共享。**Rete 算法由于采用 AlphaMemory 和 BetaMemory 来存储事实，当事实集合变化不大时，保存在 alpha 和 beta 节点中的状态不需要太多变化，避免了大量的重复计算，提高了匹配效率。

c. 从 Rete 网络可以看出，Rete 匹配速度与规则数目无直接关系，这是因为事实只有满足本节点才会继续向下沿网络传递。

5.2 Rete算法的缺点

Rete算法使用了存储区存储已计算的中间结果，以**空间换取时间**，从而加快系统的速度。然而存储区根据规则的条件与事实的数目成指数级增长，极端情况下会耗尽系统资源。

**a. 容易变化的规则尽量置后匹配，可以减少规则的变化带来规则库的变化。**

**b. 约束性较为通用或较强的模式尽量置前匹配，可以避免不必要的匹配。**

六、规则引擎调研

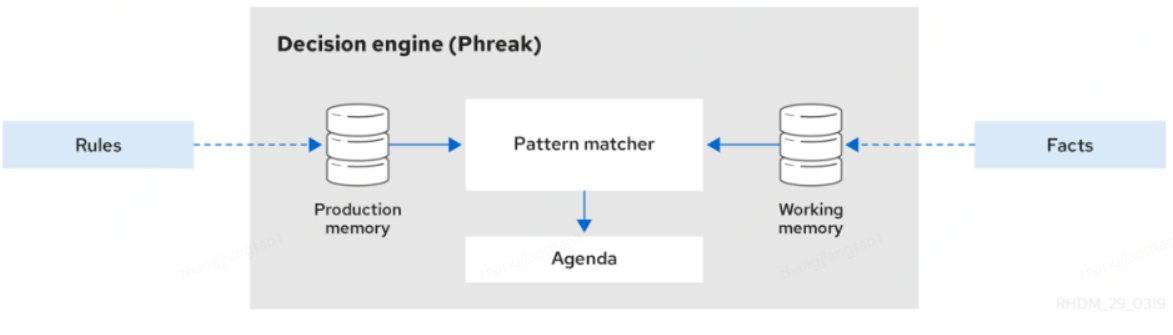
规则引擎有三个概念需要理解，如下：

**事实（Fact）：**对象之间及对象属性之间的关系

**规则（rule）：**是由条件和结论构成的推理语句，一般表示为if...Then。一个规则的if部分称为LHS，then部分称为RHS。

**模式（module）：**就是指IF语句的条件。这里IF条件可能是有几个更小的条件组成的大条件。模式就是指的不能在继续分割下去的最小的原子条件。

﻿



﻿﻿

根据业务人员编写的规则库和工作内存空间当前的状态，通过规则引擎匹配模式，把满足的规则放入议程表，将不满足的规则从议程表中删除。

a、将初始数据（fact）输入至工作内存(Working Memory)。

b、使用Pattern Matcher将规则库(Rules repository)的规则（rule）和数据（fact）比较。

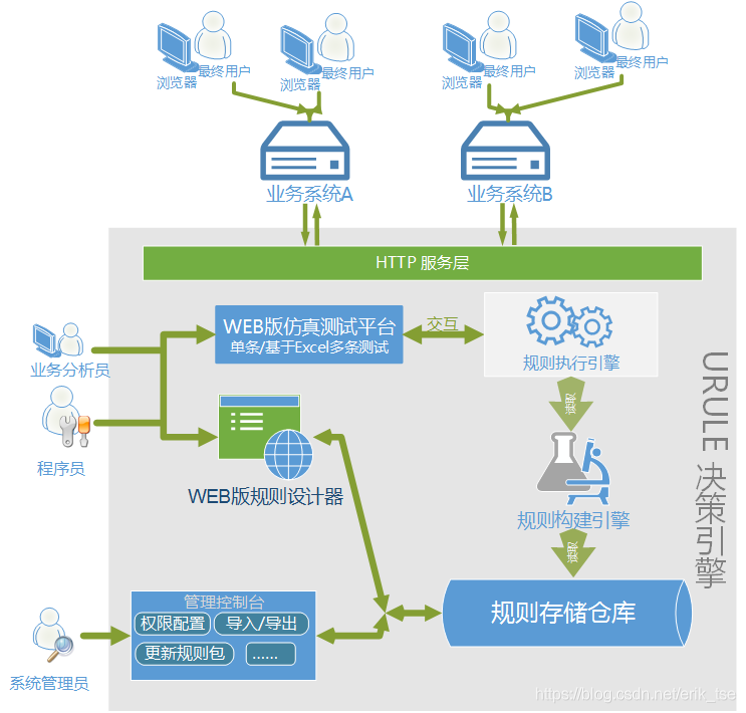
c、如果执行规则存在冲突（conflict），即同时激活了多个规则，将冲突的规则放入冲突集合。

d、解决冲突，将激活的规则按顺序放入Agenda。

e、执行Agenda中的规则。

f、重复步骤b至e，直到执行完毕Agenda中的所有规则。

•﻿



﻿﻿

6.1 规则引擎的核心问题

**任何一个规则引擎都需要很好地解决规则的推理机制和规则条件匹配的效率问题。**

a、规则引擎将逻辑与数据分离

b、数据在域对象中，逻辑在规则中。这从根本上打破了数据和逻辑的耦合，这可能是优点，也有可能是缺点。

但是**解耦逻辑**可以更容易维护。可以将逻辑全部组织在一个或多个非常不同的规则文件中，而不是将逻辑分布在许多**域对象或控制器**中。

6.2 内部规则引擎调研

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 产品 | 部门 | 特点 | ﻿ | ﻿ |
| 决策引擎-策略中心（智策） | 京东科技-风险管理中心-风险产品与研发部-风险平台产品组 | 支持实时计算：对于入参数据需要去调接口得到别的字段再做规则 | 中心化服务jsf调用 | 从入参数据到真正规则用到的数据，需要实时计算（拿着入参调接口、查缓存、流式计算）的加工，可能只要一个决策引擎不太行，得整体调用智策（其中包含特征中心） |
| 营销的一套规则引擎 | ﻿ | 主要场景是用在营销活动的规则处理。 | 中心化服务jsf调用 | ﻿ |
| 魔方ZCube | 京东科技-市场与平台运营中心-平台研发部-用户运营研发部-权益运营研发组 | 主要应用场景：会员场景我的优惠券场景 积分发放场景 白条交易数据准入规则 小金库交易数据准入规则 | 提供规则可视化搭建去中心化规则执行适用范围更广 | https://doc.jd.com/ZCube/book/ [ZCube使用介绍 神灯](http://xingyun.jd.com/shendeng/article/detail/4789?forumId=79&jdme_router=jdme%3A%2F%2Fweb%2F202206081297%3Furl%3Dhttps%3A%2F%2Fshendengh5.jd.com%2FarticleDetail%3Fid%3D4789) |

6.3 外部规则引擎框架调研

Java规则引擎主要有URule (pro) /Drools/easyRule/

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ﻿ | ﻿ | 介绍 | 优点 | 缺点 | 活跃性 | 综合评估 | ﻿ |
| 通过界面配置的成熟规则引擎： | URule | 纯Java规则引擎（RETE算法）为基础规 | 则定义方式：提供了向导式规则集、脚本式规则集、决策表、交叉决策表(PRO版提供)、决策树、评分卡决策流配合基于WEB的设计器，可快速实现规则的定义、维护与发布。 | 商用软件 | 高 | 五星 | https://github.com/youseries/urule |
| Drools | 决业务代码和业务规则分离；适用于大型应用系统 | 性能高 可整合 可维护 | 学习成本高 | 文档全 持续更新 流行 | 五星 | https://www.drools.org/ |
| 基于java代码的规则引擎 | Easy Rules | 一个比较简单的开源的规则引擎，使用简单的Java注解方式或者Java代码编程方式或者使用表达式语言或者用规则描述算子定义规则，然后使用非常简单的Java代码加载事实，规则，然后就可以在已知的事实上实现具体的行为了。 | 简单易用一个非常轻量级的框架定义规则的方式丰富多样基于POJO，支持复合规则 | Github维护者少 | ﻿ | 四星 | https://github.com/j-easy/easy-rules/releases/tag/easy-rules-4.1.0 |
| 基于JVM脚本语言： | Aviator | 各种表达式的动态求值 two pass 编译，最终生成 JVM 字节码保证性能比一般解释型脚本快 | 高性能；轻量级；支持多种类型 | Github维护者少 | 资料齐，例子多 | 四星 | https://code.google.com/archive/p/aviator/downloads |
| MVEL | 使用表达式语言定义规则 | 灵活，性能高，无类型限制 | 资料少 | Github更新少 | 三星 | ﻿ |
| ﻿ | RuleEngine | 一个可以使用SQL脚本来定义规则的中间件，如下的地址是github上基于RuleEngine的一个web可视化配置项目。 | ﻿ | ﻿ | ﻿ | ﻿ | https://github.com/rule-engine/rule-engine/releases/tag/v1.0-beta.1 |

七、规则引擎代码演示

**coding地址：**git@coding.jd.com:zhangjiangtao1/demo.git

八、业界其他公司的选择

1、美团点评(自研Maze框架)

在门店信息校验使用硬编码，门店审核流程使用Drools，绩效指标计算业务中使用绩效定制引擎。最后三个方案均放弃了，原因如下：

硬编码迭代成本高。

Drools维护门槛高。视图对非技术人员不友好，即使对于技术人员来说维护成本也不比硬编码低。

绩效定制引擎表达能力有限且扩展性差，无法推广到别的业务。

最后美团根据需求和业务特点，开发了Maze框架，其中包含两个主要的引擎分别是MazeGO（策略引擎）和MazeQL（结构化数据处理引擎）。

文章地址：https://tech.meituan.com/2017/06/09/maze-framework.html

2、美团酒旅(使用开源Aviator)

综合考虑了Esper，Drools，Aviator后选择了Aviator。

文章地址：https://tech.meituan.com/2018/04/19/hb-rt-operation.html

3、美团信息安全（自研框架Zeus）

文章地址：https://tech.meituan.com/2020/05/14/meituan-security-zeus.html

4、网易考拉（自研，名字未知）

一开始就放弃了Drools使用groovy脚本实现

文章地址：https://sq.163yun.com/blog/article/213006222321659904?spm=a2c4e.10696291.0.0.74c319a4n3RKZk

5、有赞（使用开源Drools）

文章地址：https://tech.youzan.com/rules-engine/?hmsr=toutiao.io&utm\_medium=toutiao.io&utm\_source=toutiao.io

6、携程（使用开源Drools）

文章地址：https://tech.youzan.com/rules-engine/?hmsr=toutiao.io&utm\_medium=toutiao.io&utm\_source=toutiao.io

7、Apache Nifi，Open Remote，Open Smart Register Platform （使用Easy Rules规则引擎）

九、规则引擎调研总结

规则引擎不是银弹，规则引擎只是把业务规则从应用程序代码中分离出来，通过配置文件独立管理，本质上就是把原来的Java代码转化成脚本来动态解析执行而已，还是需要写代码

解耦后虽然可以在一定程度上支持快速调整业务规则，但是由于为了实现通用性，避免与业务场景强关联，所以规则引擎都是以DSL（Domain Specific Language））或独立web页面进行维护，对开发人员和业务人员都具备一定的学习成本，而且调整也会比较繁琐，很多时候即使培训了业务人员也不懂。

**综上：**

a、如果只追求无需硬编码，并且配置人员懂得简单编码可以使用通用的规则引擎，引入规则引擎可以简化编码，而且让逻辑易于维护；

b、如果还要追求配置界面的可读性，配置人员无需了解代码，开发人员就必须往前走一步，做每个业务类型的配置界面，然后再做一个界面到规则DSL语句的转化功能。或者类型URule这样做一个通用的配置界面，但是通用界面也代表了牺牲可交互性。

十、部分内容借鉴文档：

**部分内容借鉴于：**

a、规则引擎rete算法介绍 <https://www.pudn.com/news/630367772d4eb809bf75ab38.html>﻿

b、urule介绍：[https://juejin.cn/post/6844903588725178376](https://juejin.cn/post/6844903588725178376" \t ")﻿

c、规则引擎闲谈：[https://blog.csdn.net/erik\_tse/article/details/119323719](https://blog.csdn.net/erik_tse/article/details/119323719" \t ")﻿

d、规则引擎闲谈：https://blog.csdn.net/erik\_tse/article/details/119323719

e、AviatorScript 编程指南（5.0）<https://www.yuque.com/boyan-avfmj/aviatorscript/cpow90>