基本概念：

1.实体：包含基本实体和衍生实体。基本实体是当前系统中，可以作为规则引擎操作对象的实体，例如（进件实体，反欺诈结果实体）。衍生实体，在规则引擎执行过程中需要的辅助实体，例如（信用管理中间实体）。

基本实体需要在管理系统中声明，供每个规则组调用。衍生实体根据规则依赖的方法自动添加到规则组中。

2.方法：规则组中对实体的操作，包含数据方法和工具方法。数据方法需要连接数据库，根据规则的需要操作数据库。工具方法只是对目标实体的属性操作，不需要连接数据库。这两种类型的方法都可能操作两种类型的实体，两种方法在使用的时候都需要在规则组中进行声明。

3.规则：某一种业务情况的处理，例如（用户的性别若为女性，用户的分数加1）。

规则组：对于某一个或几个实体形成的某种类型的业务的综合处理，例如（当用户的性别为女性，用户的分数加1；当用户的性别为男性，用户的积分加2），因此一个规则组中包含至少一种的并行规则。

4.场景：每个url请求接口都是一个场景。一个场景中可以包含多个串行的规则组，例如（当用户登录的时候，首先根据用户性别规则组给用户加分，然后根据用户权限规则组确定用户的页面元素）。

核心问题：

1.如何确定每个接口（场景）的参数即为该场景下所有规则组的参数。

解决这个问题需要约定两个方面：保证借口的参数直接或者间接的能够找到规则组需要的数据；保证各个规则组之间的操作合法性。这样做其实放大了业务人员的权限，有了操作数据的能力，因为接口中不能确定业务人员究竟想做什么事情。业务人员做事前一定要想清楚，这件事情到底有没有问题。可以说这里是安全稳定性和灵活性之间的制约。

2.规则引擎的并发稳定性

这个我在drools技术群里遇到几个朋友，他们说应该没有这问题，但是需要调试。

几个需要遵守的问题：

1.各个接口各司其职。一个接口负责一项功能，同时也有处理返回结果的义务，当接口知道自己应该做什么，才不会出现变化的规则影响结果的数据，导致接口异常。每个接口都有约定的结果处理办法。

规则引擎结构例图：（每一层之间都是多对多的关系）

场景1

场景2

动作1

实体1

规则组3

规则组2

规则组1

动作3

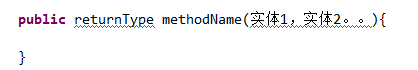
动作2

实体3

实体2

补充：

1.方法的基本结构规定如下



原因：1.方便管理。根据参数的类型可以为业务人员提供有效的规则方法，因为当业务人员确定了实体之后，可以使用的方法就已经限定了范围。

2.方便规则的定义和生成。统一的对象实体参数，这也是最高程度的抽象，业务围绕实体展开，方法拥有参数实体的所有属性操作权限，实现这样一个原则：业务人员配置规则引擎要它去做什么，达到什么目的，具体的操作在方法中实现。也就产生了下面的3.

3.系统需要准确的实体和方法仓库，这是支持业务人员添加规则的重点。系统可以通过的自省的方式在项目启动的时候校验所有的实体和方法。

2.业务人员配置规则流的基本流程：

创建规则组

向规则组中添加规则

编辑每个规则

（向规则中添加实体-🡪编辑规则的条件--🡪编辑规则的结果）

创建规则流

向规则流中添加规则组同时确定顺序

所有基础实体的属性直接使用，衍生实体的属性都是通过方法的方式获得。原因：考虑到各个规则组之间的有序性，因此衍生实体都是属于懒汉性格，在使用衍生实体属性时立即生成，保证数据的有效性。

3.所有传入规则流的实体在规则引擎运行之后，属性会更新。但是每个接口的结果返回值的处理还没有想清楚。这个和规则引擎的关系不大，主要由业务环境决定。