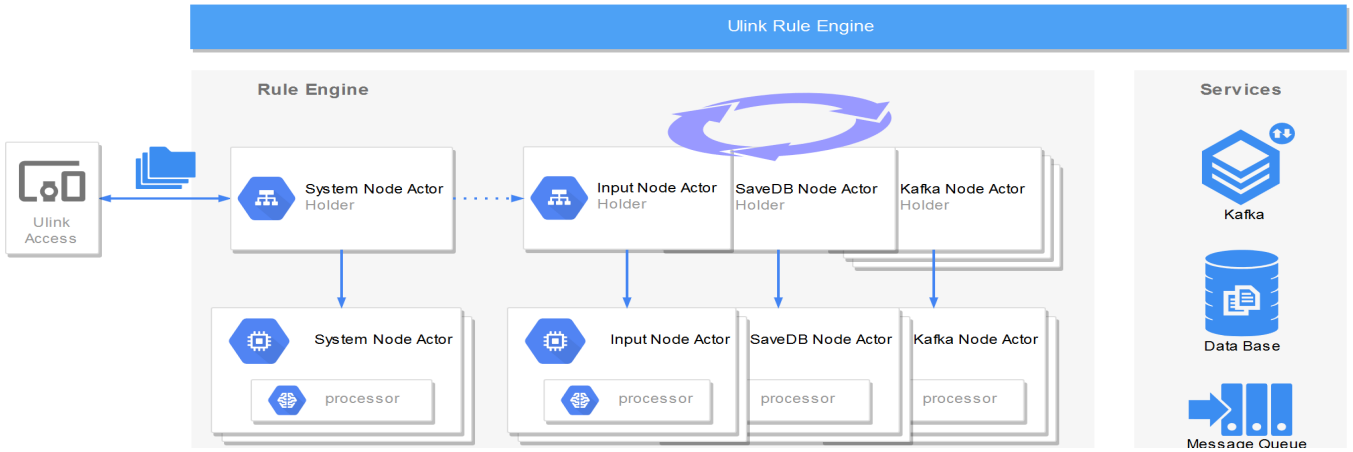
2. 架构设计

2.1 整体架构图

Rule Engine 是嵌入在应用程序中的高扩展、可插拔式的轻量级组件，**如图2-1所示**，其大体结构分为四部分：

- Actor：所有规则、流转、处理的影射，其内部的 Processor 为业务处理核心。所有 Actor 会根据自己特有的、可定制的 Configuration 来进行初始化，进而支持 Processor 对于数据流的处理。
- Holder Actor：Actor 的父级，负责管辖下的 Actor 各项生命周期及 Death Watch，也会将流入的数据分发到目标 Actor。
- Actor Service：Actor System 所依赖的各项服务，负责整个系统的初始化及销毁，也是外界与 Actor System 联系的唯一通道。
- Other Service：Actor System 的补充增强，用于对整个系统附加功能的扩展与支撑。



entity 是放置所有实体 bean 的目录。

messages 中存放着所有 actor 通信的消息类与接口。

otherservice 则是存放所有增强拓补功能的目录。目前有两项功能。

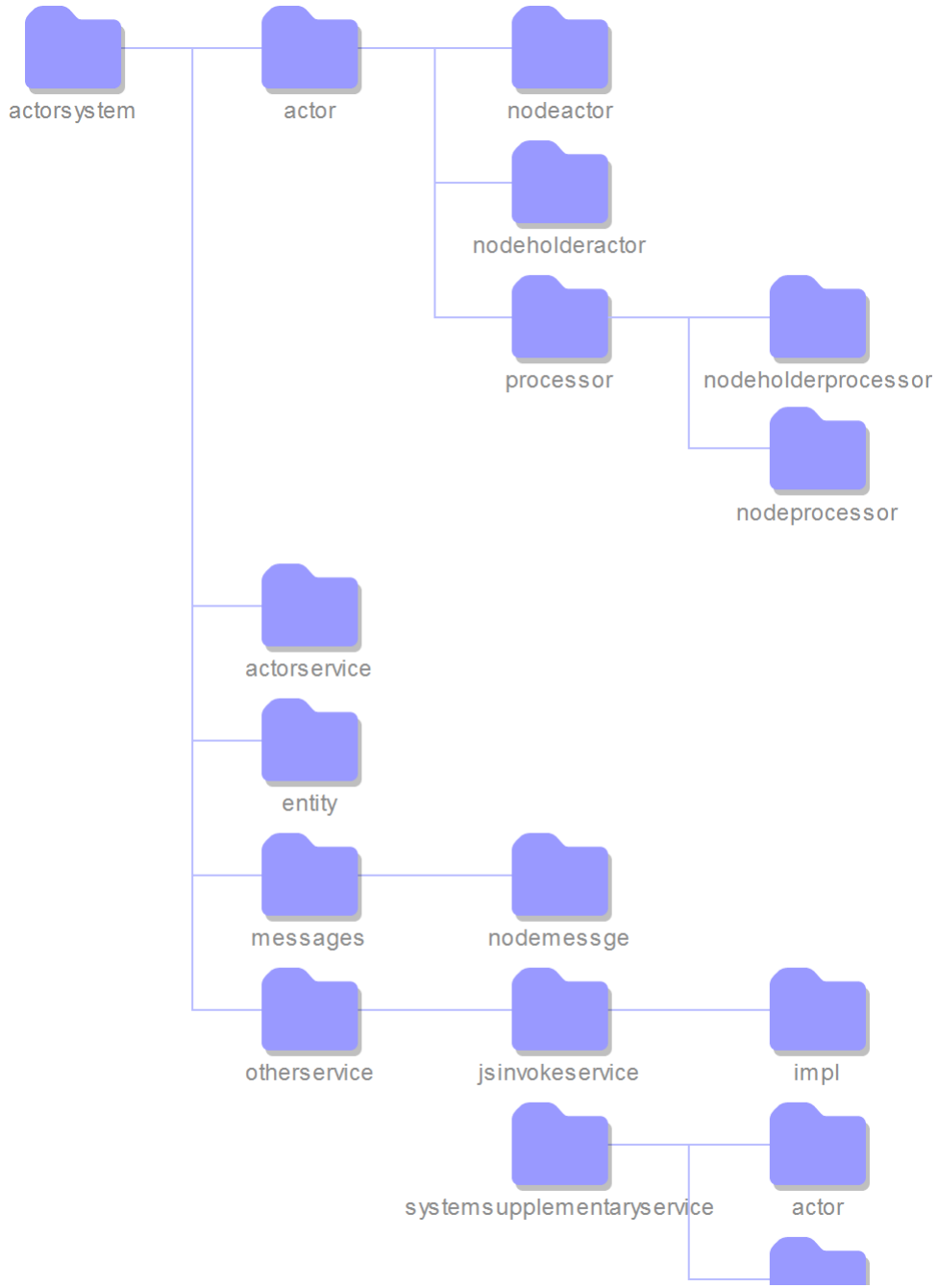
从目录结构我们可以看出，如果整个规则引擎要开发一个新类型的 node 组件，首先要创建 node holder actor，然后创建 node actor，并且配置好 node actor 的 configuration，编写初始化。最后为两个actor 各编写两个 processor。目前为止，所有 node holder actor 共用一个 node holder processor（system node holder 除外）。如果读者有需要也可以为新的 holder actor 单独配置新的 processor，这里面的编程可以做到相当灵活。

同样，如果想要新加一种通信消息，只需要在 nodemessage 中添加对应的消息类和 message type 即可。

如果对于整个 Actor System 有什么新的需求和改进，则可以考虑在 actorservice 中或者 System Node Holder Actor 中编写有关整个 Actor System 的控制。这里或许读者会对于 System Node Holder Actor 有所疑问。这个 actor 是作为消息流入庞大的 node 系统中的入口，某种意义上也有控制整个 Actor System 的作用。只不过 actorservice 中的控制更偏向为系统方面，而这个 actor 则是偏向 node、链路方向的整体把控。

最后，如果对于 Actor System 有什么更好的拓补或者需求，则可以在 otherservice 中编写相关服务，只需要注意服务的可插拔式，尽可能做到与 Actor System 的不耦合。

在各项功能改进或者需求完成的过程中，所有新增的实体类放置到entity里，这样，整个 Actor System 的目录结构就已经清晰可见，对于开发者来说已经是非常有序的归整结构。


actorsystem
actor
nodeactor
nodeholderactor
processor
nodeholderprocessor
nodeprocessor
actorservice
entity
messages
nodemessge
otherservice
jsinvokeservice
impl
systemsupplementaryservice
actor

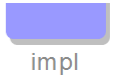


图2-2

2.3 概念描述

Rule Engine 在设计之初就曾考虑如何高效、高扩展性的实现对于数据流的处理、分发。笔者在参考了阿里云、SiteWhere、ThingsBoard 等物联网项目后，选择了以 ThingsBoard 为参照的 Actor 模型设计模式。其主要优势有以下几点：

1. 作为国外优秀的 IoT 平台在GitHub上已有 3,249 Star，可谓有良好的技术用户基础。
2. 平台功能齐全，包括 Asset、Device、Rule Engine、Dashboard、Security 等 IoT 所必需的基本要素模块。
3. 开源，文档齐全并配有详细的视频演示。
4. 代码架构清晰，利于开发者在自行研究开发时进行参考。
5. 规则引擎可视化拖拽形成数据流式处理链路，配合 Js 脚本可达到高度自定义。

2.3.1 设计概念

本文进行到这里，需要读者对于 Akka Actor 有了一定的掌握。如果是对于 Akka Actor 没有任何了解的，笔者强烈建议读者通读第一章节的核心技术，其中所有的技术要点将会贯穿于接下来的文章。

首先我们看一张 ThingsBoard 的规则引擎图，**如图2-3所示。**

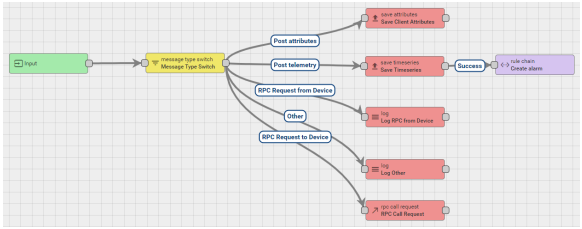


图2-3

笔者在看到这个 UI 界面的时候也有些为之所动，在后续的源码研读之后，发现整个规则引擎链路设计的很巧妙，技术栈并不复杂。

这张图清晰的展示出 ThingsBoard 的规则引擎中的三要素：**节点、关系、链路。**这三个要素也确定了规则引擎所需要依赖的三张数据库表。

这条链路是 ThingsBoard 中的 Root Chain，意为所有的数据都会从 Input 节点中流入，跟随着箭头达到定义的每一个节点做对应的处理，这也是 Ulink Rule Engine 所要达到的目标——可配置高度自定义的可视化数据流式处理。

所以也可以说 Rule Engine 是由一条条 Rule Chain 构成的。哪怕是 Root Chain，其实也只是一条 Rule Chain，只不过被 Rule Engine 赋予了特殊的功能罢了。

图中每一种节点代表每一个 node actor，而所有 chain 的同一种节点，都会被它们的一个共同的 node holder actor 监督管理着。每一种节点有着自己独特的功能，对于功能的配置则是每一个节点的 configuration，然后就是每一个 node actor 所特有的 processor 会根据 node actor 的 configuration 来对数据流做分析、处理、转发。

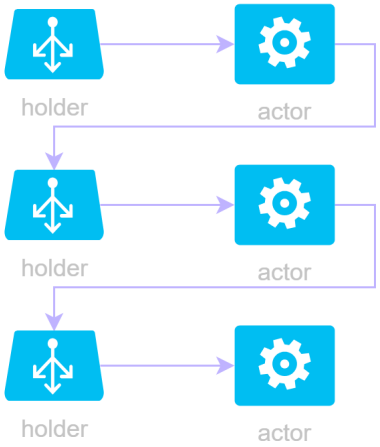
这样一看，读者可能已经明白了 **Rule Engine 的概念。**

数据流在 Actor System 外部转化为对应的 actor 消息结构，并通过 actor service 的 input 方法流入 System Node Holder Actor。

System Node Holder Actor 会把消息发送给其掌管的所有子节点 System Node Actor，每一个 System Node Actor 都是一个 Root Chain Input 节点的映射，它们会把消息转发到自己所映射的那个 Root Chain 的 Input 节点。当然，实际上它们不是直接转发给 Input Node Actor，而是转发给 Input Node Actor 的唯一掌管者——Input Node Holder Actor。

holder 中存放着所有子节点的地址 map，将每一条 System Node Actor 转发过来的消息发送给消息的目标 node actor。 node actor 在接收到消息之后，根据消息类型做出对应的处理，然后转发给它的所有下游 actor 的 holder，最终消息像水流一般流遍整个链路。

简单的流程如下：



2.3.2 链路概念

上文提到过，在 Rule Engine 其实是有多个规则链路构成。那么每一条链路，可以根据客户自己的需要与自身业务的概念进行关联。如目前 Ulink 平台所服务的服务一体化平台，每一条 Rule Chain 对应的是一个产品的规则。如果有其他的需要，Rule Chain 甚至可以成为 Web Application 开发的一条 Controller 至 Data Base 的映射，规则引擎中的链路是一个相当抽象的概念，可以具体化到生活中的很多应用场景，工厂中的一条生产线，学校中一个学生一天的学习历程，生活中处理事务的一条线索等等。而链路和链路的可连接性也大大提高了链路对于复杂场景的可用性。

每条链路是由多个节点构成，原则上每条链路都会有一个唯一外部入口，目前这个入口被规定为 Input 节点，当然代码中已经灵活的可以定为其他任一类型的节点。举一个通用的链路例子我们来了解链路中较为复杂的情况，**如图2-4所示。**

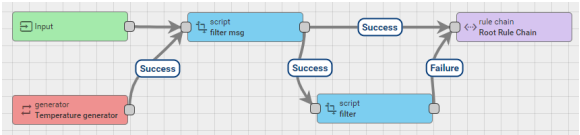


图2-4

每一条链路都会有一个 First Node——Input 为唯一外部数据流入入口，链路的最后会将数据流转到 Root Chain。Input 的下游节点只有一个 filter msg，而二者的关系没有任何标签，表示数据会无条件的流入到 filter msg 中。而 filter msg 有两个下游节点：filter，Root Chain 的入口节点。而这两个下游节点的关系都是 Success 标签，表示只有 filter msg 的处理结果为 Success 时，数据才会流入下游。同理 filter 的处理结果为 Failure 时，数据才会流入下游。

这里我们要单独介绍一个 generator 节点。介绍它不是因为这个节点的特殊性，而是因为这个节点的特殊位置。这个节点与 Input 非常类似，都没有上游节点。Rule Engine 中对于此类没有上游关系的节点统称为 head node。这里需要提一下，Input 节点的下游关系想必读者已经很清楚，但是它是否没有上游关系呢？其实 Input 节点是有一个上游关系，这个关系的 from 为这条 Rule Chain，而 to 为 Input 节点，表示 Input 节点是一个在此 Rule Chian 中的没有上游关系的节点。这么一说，读者可能清楚了 generator 节点实际上也存在这么一个上游关系，from 为这条 Rule Chain。至于为何做出这样的设计下一小节对于关系的解读中，和后续对于整个链路的初始化、更新操作，读者可以慢慢体会。

注意

ThingsBoard 中，无法添加 filter node 的下游为 filter msg，也就是无法形成一个闭环。形成闭环则会造成数据在链路中循环，猜测ThingsBoard是在前端做出的控制，但是相信后端也会有相应的控制。目前的 Rule Engine 还未做控制，后续开发者需要注意此类情况的发生。

2.3.3 关系概念

Rule Engine 中的关系概念很简单，却也很巧妙，看似复杂的规则链路由这些关系构成。而 ThingsBoard 中对于关系的运用不仅仅体现在 Rule Chain 中，事实上，对于任何类似于从属的关系他们都是运用这套关系概念去做持久化和映射的。但是目前在 Ulink 中，只有 Rule Chain 会运用这套关系概念，其他的例如规则链从属于哪个租户这种从属关系则还是依照之前的常见关系逻辑。

所以在 Rule Engine 中，关系都是由 from，to 构成。关系只有规则链至节点，节点至节点，节点至规则链三种：

- from 规则链 - to 节点
  这套关系在上文中已有简述，用于描述一个节点在一个规则链中如果没有任何其他上游节点，则会用这种关系进行表述。

- from 节点 - to 节点
  最常见的关系，这种关系表述节点与节点之前的联系，需要注意的是这其实可以是一个多对多的关系。

- from 节点 - to 规则链
  当节点连接到其他规则链时采用这种关系描述，规则引擎在处理这种关系时，会找到下游规则链的 First Node 作为数据真实的目标 node。

关系中还有标签属性，对于一般的 node 来说只有 Success 和 Failure 标签，目前的规则引擎对于这两种标签也没有做过多的解析处理和运用。使用较多的是 Check Node，用于 check 数据流，将数据流运行于 configuration 配置的 js 语句，返回 true/false 结果。所以 check node 的下游关系会有两种标签 TRUE/FALSE。类似的还有 Message Type Switch Node，根据标签路由分发数据流。

2.3.4 节点概念

节点概念在整个规则引擎中是基础，也是最简单的概念。但是每一个节点中的 processor 其实是整个规则引擎最为复杂的部分，也是消耗内存和 CPU 最多的部分。

Rule Engine 中，每一个节点都有自己独特的 Configuration。那些 Configuration 还是空白的节点，只是因为节点的作用还没有具体细化，不代表它们不需要自己独特的 Configuration。对于 processor 也是一样的逻辑，每一个节点，甚至每一个节点的父监管都有各自的 processor。虽然目前，所有业务相关的 node holder actor 都是共用一个 processor，不代表今后我们不会对它进行定制化的处理。

ThingsBoard 中的节点有很多，目前的 Rule Engine 只是集成编写了一些主要节点，每一种节点都有一个 node holder actor。上文也提到过，所有同类型的 node actor 都会被这个类型节点的 node holder actor 监管，流入 node 的数据只会从他们的监管者来源，坚持这一点虽然有些苦难，但是是非常必要的。