3. 详细设计

这个章节我们将结合源码开始介绍 Rule Engine 的原理，以及如何利用先用的框架进行二次开发。

3.1 项目配置

3.1.1 项目依赖

下面是截止笔者撰文时，项目的依赖情况和版本。其中 delight-nashorn-sandbox 是 delight 针对 java8 nashorn script engine 所做的一个开源 js 安全运行沙箱。

```
<dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
        <version>27.1-jre</version>
</dependency>
<dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-actor_2.12</artifactId>
        <version>2.5.21</version>
</dependency>
<dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-testkit_2.12</artifactId>
        <version>2.5.21</version>
</dependency>
<dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-slf4j_2.12</artifactId>
        <version>2.5.21</version>
</dependency>
<dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka_2.12</artifactId>
        <version>2.2.0</version>
</dependency>
<dependency>
        <groupId>org.javadelight</groupId>
        <artifactId>delight-nashorn-sandbox</artifactId>
        <version>0.1.22</version>
</dependency>
```

3.1.2 配置文件

actor-system.conf 是 Akka Actor 的配置文件，其中 blockBoundedMailBox 为 System Node Holder Actor 的背压设计，具体如何进行调优配置可以参考 Akka 官方文档。

```
akka {
  # JVM shutdown, System.exit(-1), in case of a fatal error,
  # such as OutOfMemoryError
  jvm-exit-on-fatal-error = off

  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"

  log-dead-letters = 955
  log-dead-letters-during-shutdown = on
}

akka.actor.default-mailbox {
  mailbox-type = "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
}

akka.actor.deployment {
  /SystemNodeHolderActor {
    mailbox = blockBoundedMailBox
  }
}
blockBoundedMailBox {
  mailbox-type = "akka.dispatch.BoundedMailbox"
  mailbox-capacity = 996
  mailbox-push-timeout-time = 5s
}

# This dispatcher is used for system
system-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 1
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 2

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.1
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}
```

```
# This dispatcher is used for input node
input-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 4
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 8

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.25
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for message type switch node
messagetypeswitch-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 1
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 2

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.1
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for router node
router-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 1
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 2
```

```
    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.1
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for data filter node
datafilter-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 4
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 8

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.25
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for save db node
savedb-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 4
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 8

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.25
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for data check node
```

```
datacheck-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 4
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 8

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.25
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for create alarm node
createalarm-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 4

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.2
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}

# This dispatcher is used for kafka node
kafka-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 16
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 64

    # The parallelism factor is used to determine thread pool size using the
    # following formula: ceil(available processors * factor). Resulting size
```

```
    # is then bounded by the parallelism-min and parallelism-max values.
    parallelism-factor = 0.5
  }
  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 50
}
```

bootstrap-native.properties 配置中，actor.actorsystem.actortypes 为整个 Actor System 系统启用的 node 类型。这里必须至少填一个，不然项目会启动报错。没有在此列中的节点类型，不会初始化对应的 node holder actor，那么自然不会有任何子 actor 生成在系统中。

kafka ulink 中的所有配置仅仅是为了兼容旧版服务一体化所硬性配置的 kafka 配置项。理想情况下，这些配置项应该是可视化界面中 node 的 configuration 中手动填写配置。

mongo.service.dao.thread_pool_size，mongo 存储的异步线程池大小。

```
#akka
actor.actorsystem.name = uama-akka
actor.actorsystem.actortypes = SYSTEM, INPUT, MESSAGE_TYPE_SWITCH, ROUTER, SAVE_DB, DATA_FILTER, DATA_CHECK,
CREATE_ALARM, KAFKA
actor.actorsystem.config.filename = actor-system.conf

#js evaluator config
js.java.sandbox_thread_pool_size = 32
js.java.max_cpu_time = 300
js.java.max_errors = 3

#kafka ulink
spring.kafka.ulink.bootstrap-server = kafka.test.ulink.uama.cc:9092
spring.kafka.ulink.username = producer
spring.kafka.ulink.password = prod-sec
spring.kafka.consumer.device.alert.topic = UAMA_DEVICE_ALERT
spring.kafka.consumer.device.datapoint.topic = UAMA_DEVICE_DATAPOINT
spring.kafka.consumer.device.state.topic = UAMA_DEVICE_STATE

#mongo async
mongo.service.dao.thread_pool_size = 4
```

3.1.3 数据库表

Rule Engine 依赖三张关系型数据库表，**如图3-1所示。**同时项目还依赖这三张表的 RUID Serivce，后面会介绍到。

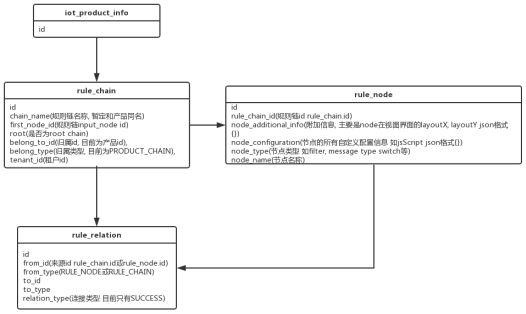

图3-1

初始化表和数据

init_sql.sql

3.1.4 服务依赖配置

最后我们还有一项准备工作，就是给 Actor System 配置数据源的服务依赖。在 DefaultActorDependenceService 类中，我们可以看到三张关系型数据库表和 mongo 中两个 collection 的 service 依赖注入，需要读者注意。其中的 RedisHelperWithNoExpire 的依赖注入可以省略，目前系统中并没有用到。

3.2 原理流程

3.2.1 项目初始化

准备工作就绪，现在让我们启动项目。接下来我们将逐步分析在启动过程中，Actor System 都做了什么。

前文提到过，actorservice 包中存放着控制 Actor System 生命周期的类。那么如果提到项目初始化，我们第一个想到的就是在这个包下一探究竟。

略过接口，我们找到 DefaultActorService 类。

来到 init 方法，我们可以清晰的看到项目的初始化分为 4 个步骤。

初始化 Actor System

```
// 基本校验
this.check(systemActorTypeList);
// 获得actor system配置
Config config = this.getActorSystemConfig(configFileName);
// 生成actor system
actorSystemContext = ActorSystemContext.create(actorSystemName, config);
```

基本校验目前只是对上文中提到的配置文件中的 actor.actorsystem.actortypes 配置项校验，不能为空。

读取初始化 Actor System 需要的配置。

初始化 Actor System，得到引用存放在 Actor System Context 中。

初始化增强拓补功能

```
// 初始化deadLetter增强，嵌入在actor system中 回调为如何处理系统中的dead letter
this.systemSupplementaryService.initDeadLetterActor(this.actorSystemContext.getActorSystem(), deadLetter -> {
    Object message = deadLetter.message();
    if (message instanceof NodeMessage) {
        DefaultActorService.actorDependenceService.getDeadLetterMongoDbService().asyncSave((NodeMessage)message);
    }
    return null;
});
```

这里的注释写的很清楚，主要是初始化对于 Dead Letter 的处理，灵活的通过回调函数对未送达的消息做处理。这里可以看到目前系统只会对 Node Message 也就是各类节点消息进行处理，而未送达的 Actor System 系统消息则会抛弃。事实上系统消息未送达的情况极少，而且没有什么存储需求。

初始化 node holder actor

```
if (null == actorSystemContext) {
    throw new IllegalArgumentException("actor system context might not have been initialized");
}
// 初始化配置文件中的每一个node holder actor
systemActorTypeList.forEach(actorTypeMapperEnum -> {
    ActorRef nodeHolderActorRef = new NodeHolderActor.ActorCreator(actorSystemContext, actorTypeMapperEnum).create();
    actorSystemContext.getActorHolderMap().put(actorTypeMapperEnum, nodeHolderActorRef);
});
```

这里会初始化配置文件中所配置的每一种 node 的 holder。可以看到在初始化的时候，我们会调用 create 方法得到 actor ref，然后将所有的 node holder actor 的引用存放在 context 的 map 中。

在 create 方法中，读者或许已经注意到 ActorTypeMapperEnum 这个类。这个类是所有 Actor 使用情况的映射，包括他们所使用的 dispatcher 名字。所以如果开发者想要添加一个新品种的 node，除了创建 holder actor 和 actor 之外，还需要在这个类中添加好映射。

初始化所有链路

holder actor 生成后，我们就可以生成所有的规则链路了。因为规则链路中的每一个节点，都是一个 actor，而这个 actor 的父监督，便是已经创建好的 holder actor。

这里会初始化数据库中所有的规则链路，包括 Root Chain。

使用 addRuleChainsToActorSystem 方法进行添加特定的规则链路到 Actor System 系统内存中。和传统的做法一样，如果这里传入的参数是空集合或者 null，则视为添加所有链路。这个方法有点复杂，我们会逐步讲解。

```
// prepare data
NodeBuildPreparedData preparedData = this.preparedNodeBuildData(ruleChainIdList);
List<MIotRuleEngineRuleChainInitV> addedRuleChainVList = preparedData.getAddedRuleChainVList();
Map<String, MIotRuleEngineRuleChainInitV> ruleChainIdMap = preparedData.getRuleChainIdMap();
Map<String, List<MIotRuleEngineRuleChainInitV>> chainsGroupByBelongToIdMap = preparedData.getChainsGroupByBelongToIdMap();
Map<String, List<MIotRuleEngineRuleNodeInitV>> nodesGroupByRuleChainIdMap = preparedData.getNodesGroupByRuleChainIdMap();
Map<String, List<MIotRuleEngineRuleRelationInitV>> relationsGroupByRuleChainIdMap = preparedData.getRelationsGroupByRuleChainIdMap();
```

首先是 prepare data。这里准备的数据都是后面生成 actor 所必须用到的。大致可以理解为以下几部分：

- 所要添加的所有链路 addedRuleChainVList，这里主要是为了后面的循环 forEach。
- 链路信息 ruleChainIdMap，用于获取对应链路的信息。其中比较重要的信息是该链路的 First Node Id 和 First Node Type。
- 根据 belong_to_id 进行分组的 chainsGroupByBelongToIdMap，主要用于 Router Node。Router Node 会根据数据流的产品 id 决定该数据流流向哪一条/几条规则链。因为目前的情况是产品与规则链是一对一的关系，为了适应一对多，Rule Engine 已经考虑到一个产品 id 可能会把数据引向多个规则链。

- 链路中所有节点的信息 nodesGroupByRuleChainIdMap。

- 链路中所有关系的信息 relationsGroupByRuleChainIdMap。

注意

这些准备数据看起来很繁杂，其实每一个在后续都是不可或缺的。如果开发者对此系统还不是很了解，建议不要在此处做任何更改，因为此处的更改是系统级的，可能会导致系统无法正常启动。

```
// 初始化时间戳 用于标记同一时间初始化的各个链路 所于 一个链路中 一个节点如果为多个上层节点的目标 则会收到所有上层节点
// 发来的init node message，则会进行多次初始化。此标记为了标记该节点已经该时间戳的链路初始化任务初始化过 不需要再次初始化
long initTimeStamp = System.currentTimeMillis();

// send node init message to every rule chain to init all node actors
addedRuleChainMList.parallelStream().forEach(ruleChainV -> {
    String ruleChainId = ruleChainV.getId();
    Map<String, List<MIotRuleEngineRuleRelationInitV>> ruleChainRelationsGroupByFromIdMap =
relationsGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(MIotRuleEngineRuleRelationInitV::getFromId));
    Map<String, List<MIotRuleEngineRuleRelationInitV>> ruleChainRelationsGroupByToIdMap =
relationsGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(MIotRuleEngineRuleRelationInitV::getToId));
    Map<String, MIotRuleEngineRuleNodeInitV> ruleNodeIdMap = nodesGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.toMap(MIotRuleEngineRuleNodeInitV::getId,
v -> v));
    ruleChainRelationsGroupByFromIdMap.get(ruleChainId).forEach(ruleRelationInitV -> {
        String startNodeId = ruleRelationInitV.getToId();
        ActorTypeMappersEnum startNodeActorTypeEnum = ActorTypeMappersEnum.valueOf(ruleNodeIdMap.get(startNodeId).getNodeType());
        actorSystemContext.getActorHolderMap().get(startNodeActorTypeEnum)
                .tell(new NodeMessageCarrier<>(new NodeId(startNodeId),
                        null,
                        new NodeInitMessage(ruleChainRelationsGroupByFromIdMap,
                                ruleChainRelationsGroupByToIdMap,
                                ruleNodeIdMap,
                                chainsGroupByBelongToIdMap,
                                ruleChainIdMap,
                                initTimeStamp)),
                        ActorRef.noSender());
    });
});
```

接下来我们会对每一条规则链进行初始化，方法是向每一条需要添加到 Actor System 内存的规则链的 first node 发送一条 NodeInitMessage（实际上会先发送给其 holder，再由 holder 进行转发）。

我们先了解一下消息结构。Rule Engine 中的大多数消息结构都是一个 NodeMessageCarrier。其中包含该消息实际的目标 node 的 id，根据这个 id，holder 才会准确的分发给目标子 actor。其次就是包含整个消息体，每一个消息体有各自不同的种类，上图中的 NodeInitMessage 是一种，actor 的 processor 会根据这个种类去做对应的处理。

提示

根据 Akka Actor 的规范，消息为不可变元素，故所有的消息成员变量应该为 final 类型。如果需要在传递的过程中变更消息的成员变量状态，则可以重新 new 出 message，如后文的 NodeDeleteMessage 就存在这种在传递过程中被重新 new 出的情况。这种情况为特殊的情况，建议谨慎使用。

这时候我们认为所有的链路已经被初始化完成，实际上由于是异步，可能消息还未传递到每一个链路的节点。不过这根本不影响我们后续的操作，由于消息的队列性和有序性，每一个节点在收到下一个消息的时候一定是在收到 NodeInitMessage 之后。后面我们会介绍 holder 在接收到 NodeInitMessage 会做哪些事情，现在我们更关心整体的节奏。

```
// associate system holder actor with root chains first node and init rule chain context
addedRuleChainMList.parallelStream().forEach(ruleChainV -> {
    String ruleChainId = ruleChainV.getId();
    String firstNodeId = ruleChainV.getFirstNodeId();
    Map<String, MIotRuleEngineRuleNodeInitV> ruleNodeIdMap = nodesGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.toMap(MIotRuleEngineRuleNodeInitV::getId,
v -> v));
    Map<String, List<MIotRuleEngineRuleRelationInitV>> ruleChainRelationsGroupByFromIdMap =
relationsGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(MIotRuleEngineRuleRelationInitV::getFromId));
    actorSystemContext.getActorHolderMap().get(ActorTypeMappersEnum.SYSTEM)
            .tell(new NodeMessageCarrier<>(new NodeId(firstNodeId),
                    null,
                    new NodeInitMessage(ruleChainRelationsGroupByFromIdMap,
                            null,
                            ruleNodeIdMap,
                            null,
                            ruleChainIdMap,
                            initTimeStamp)),
                    ActorRef.noSender());
});
```

下一步我们开始把刚才初始化好的每一条规则链同 System Node Holder Actor 联系起来。前文中我们多次提到过，System Node Holder Actor 是对于 Actor System 偏向链路的把控。如果没有获取，保存所有链路的状态，做到把控是很困难的。

这里我们会把所有 Root Chain（目前一个系统只有一条 Root Chain）的 first node 在 System Node Holder Actor 中生成映射——System Node Actor。这样在数据流入的时候，System Node Holder Actor 会将数据转发给所有的子 actor。是的，这里我们默认数据会给所有的 Root Chain 发送一份，如果想要有定制化的配置，可以给 System Node Actor 做对应的 Configuration 配置。

之后我们还会把目前在系统中的所有链路的信息存放在 System Node Holder Actor 中，以 map 的形式保存每一条规则链路的上下文。此举主要是用于之后的链路更新操作。

```
// update root chains router couse new chains added
actorSystemContext.getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM)
        .tell(new NodeMessageCarrier<>(null,
                        null,
                        new RootRouterUpdateMessage(addedRuleChainVList)),
                ActorRef.noSender());
```

最后我们会把新加入的 Rule Chain(s) 加入到 Root Chain 中 Router Node 的内存中。这样在数据流入的时候，数据就会顺利的动态转发到新生成的链路中。这里以 System Node Holder Actor 作为入口只是为了给所有的 Root Chain 做转发。直接给所有的 Root Chain 的 first node （实际上是 holder，然后再由 holder 转发）发送此消息也可以完成同样的功能。

至此系统初始化完成，我们会看到系统已经可以正常的运行起来了。

接下来我们会继续从宏观的角度去介绍系统的生命周期，至于系统中每一个 actor 在收到某一种生命周期类型的 message 会做什么我们会在之后详细介绍。

3.2.2 规则链路的删除

在 DefaultActorService 类中，我们应该注意到了还有规则链路删除的方法 deleteRuleChainsFromActorSystem。

```
long deleteTimeStamp = System.currentTimeMillis();
    ruleChainVList.parallelStream().forEach(ruleChain -> actorSystemContext.getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM)
            .tell(new NodeMessageCarrier<>(null,
                            null,
                            new NodeDeleteMessage(ruleChain, false, IotRuleChainRootEnum.ROOT.getCode().equals(ruleChain.getRoot()), deleteTimeStamp)),
                    ActorRef.noSender()));
```

这里的删除时间戳 deleteTimeStamp 原理同初始化链路时的初始化时间戳。如果开发者还不了解可以回到上文中详细阅读注释。

对比初始化规则链路，删除链路则显得相当简单。系统只是给 System Node Holder Actor 发送一条 NodeDeleteMessage，这时候我们认为链路已经从 Actor System 内存中删除掉，包括 Router Node 的路由内存。

注意

这里的编写已经考虑到删除 Root Chain 的情况，只是目前为止，在接收到删除 Root Chain 的消息时，系统会报出错误提示，并不会真正的从内存中删除掉。

3.2.3 规则链路的更新

同样在 DefaultActorService 类中，存在方法 updateRuleChainsInActorSystem 用于更新内存中已存在的规则链路。

```
// send node update message to every rule chain to update all node actors
    ruleChainVList.parallelStream().forEach(ruleChainV -> {
        String ruleChainId = ruleChainV.getId();
        String firstNodeId = ruleChainV.getFirstNodeId();
        Map<String, List<MIotRuleEngineRuleRelationInitV>> ruleChainRelationsGroupByFromIdMap =
relationsGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(MIotRuleEngineRuleRelationInitV::getFromId));
        Map<String, List<MIotRuleEngineRuleRelationInitV>> ruleChainRelationsGroupByToIdMap =
relationsGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(MIotRuleEngineRuleRelationInitV::getToId));
        Map<String, MIotRuleEngineRuleNodeInitV> ruleNodeIdMap =
nodesGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.toMap(MIotRuleEngineRuleNodeInitV::getId, v -> v));
        actorSystemContext.getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM)
                .tell(new NodeMessageCarrier<>(new NodeId(firstNodeId),
                                null,
                                new NodeUpdateMessage(new NodeInitMessage(
                                        ruleChainRelationsGroupByFromIdMap,
                                        ruleChainRelationsGroupByToIdMap,
                                        ruleNodeIdMap,
                                        chainsGroupByBelongToIdMap,
                                        ruleChainIdMap,
                                        timeStamp),
                                        new NodeDeleteMessage(ruleChainV, true, IotRuleChainRootEnum.ROOT.getCode().equals(ruleChainV.getRoot()), timeStamp)
                                )),
                        ActorRef.noSender());
    });
```

我们跳过准备数据和构建更新时间戳(实际上还是初始化时间戳)的部分，改方法会给 System Node Holder Actor 发送一个 NodeUpdateMessage 消息，里面又包含了两条消息：NodeInitMessage 和 NodeDeleteMessage。

更新看起来也很简单，但实际上是工作最为复杂的生命周期过程，笔者遇到的很多 bug 都是在这个生命周期中产生的。

看到 NodeUpdateMessage 的结构相信部分开发者已经感觉到我们要如何对于内存中存在的产品链进行更新了。简单来说我们会将内存中需要更新的链路先删除掉，然后再重新初始化，不过这个过程是相当复杂和繁琐的，而且极容易出错，开发者在修改此功能时需要谨慎处理。

3.2.4 NodeInitMessage 的处理

在这一小节我们会介绍在接收到 NodeInitMessage，我们会做什么。

```
NodeInitMessage nodeInitMessage = (NodeInitMessage) messageCarrier.getMessage();
// 需要初始化的node actor是否已经存在
Optional<NodeId> nodeIdOpt = super.actor.getNodeIdBiMap().keySet().parallelStream().filter(tarNodeIdV -> tarNodeIdV.equals(nodeIdV)).findAny();
if (nodeIdOpt.isPresent()) {
    // 已存在 查看该actor初始化时间戳是否与本次需要初始化的时间戳相同
    // 相同则跳过初始化 不同则进行初始化并且替换
    NodeId tarNodeIdV = nodeIdOpt.get();
    NodeLifeCycleInfo nodeLifeCycleInfo = super.actor.getNodeLifeCycleMap().get(tarNodeIdV);
    if (null == nodeLifeCycleInfo) {
        super.logNodeLifeCycleIsNull(super.actor.getType().getNodeHolderActorClass().getSimpleName(), nodeInitMessage, nodeIdV);
    } else {
        if (nodeLifeCycleInfo.getInitTimeStamp() == nodeInitMessage.getInitTimeStamp()) {
            super.bLog.logWhoOccursWhenWarn(super.actor.getType().getNodeHolderActorClass().getSimpleName(), "a warn", String.format("process %s",
nodeInitMessage.getClass().getSimpleName()), "node actor already create, nodeId: " + tarNodeIdV);
            return;
        }
    }
}
ActorRef actorRef = createNodeActor(nodeIdV, nodeInitMessage);
if (null == actorRef) {
    return;
}
initNodeActor(actorRef, nodeInitMessage);
// 释放缓存的message
if (checkIfNodeIdVLocked(nodeIdV)) {
    releaseCachedMessage(nodeIdV);
}
```

各 Node Holder Actor 在接收到 NodeInitMessage 后，会检查该 actor 是否已经被初始化。为什么会出现此情况呢？

在一条单线规则链路的情况下，Node Holder Actor 在收到初始化消息时，目标 actor 是一定没有被初始化的。这时候 Node Holder Actor 会利用初始化消息中的基础信息来初始化目标 actor，并且会将 actor ref 放置到自身的 map 中维护起来。最后会释放缓存在该 holder 中，且目标为该 actor 的所有消息。这些消息会在该 actor 为锁定状态时缓存起来。而在更新链路的时候，会锁定该链路的 first node。在更新链路的处理时，我们会详细解释。

```
// if locked, cache message
if (checkIfNodeIdVLocked(messageCarrier.getTargetNodeIdV()) && !messageType.equals(NodeMessageType.NODE_INIT_MESSAGE)){
    putMessageToCache(messageCarrier);
    return;
}
```

每个 Node Holder Actor 在收到合法消息的第一步，会做此 lock 检查。如果已被锁定，则缓存消息并且直接返回，不会对消息做任何进一步处理。

现在回到之前的问题，何时 actor 会在已经被初始化的情况下接收到 NodeInitMessage 呢。试想如果在一个规则链路中，一个节点有多个上游节点，那么这几个上游节点都会给这个节点发送一个 NodeInitMessage。最快到达的第一个消息会被成功的执行，而第二个和第三个消息抵达时就会出现目标 actor 已经被初始化的情况。此时我们就会通过初始化时间戳判断这次抵达的初始化消息与 actor 被初始化时的消息是否为同一批消息。如果不为同一批，则重新初始化（目前这种情况不会出现，不为同一批的初始化会发生在更新链路时，但是在更新链路的时候会提前删除掉 actor 再重新初始化）。

在 createNodeActor 方法中，Node Holder Actor 会对初始化的 actor 进行 death watch，监控该 actor 的停止状态。

在 initNodeActor 方法中 Node Holder Actor 还做了比较关键的一步，就是将该 NodeInitMessage 发送给刚初始化好的 actor，目的只是为了让其转发给其所有下游 actor，进而完成类似水流的规则链初始化模式。

3.2.5 NodeDeleteMessage 的处理

接下来是针对 NodeDeleteMessage 的处理介绍。

首先，System Node Holder Actor 会接收到 NodeDeleteMessage，此时的 deleteFlag 为false，所以在消息的传递过程中不会有任何节点会做删除的操作，仅仅只是做消息传递。System Node Holder Actor 会将消息转发给所有的 Root Chain 来确保每条 Root Chain 中的 Router Node 都会从内存中删

除该规则链路。Router Node 收到该消息后，会复用消息中的所有成员变量（new 一个新的NodeDeleteMessage），然后置 deleteFlag 为 true，转发此消息到删除链路的 first node（实际上为 holder actor）。所以在后续的传递中，所有节点 actor 会认为这是一条需要执行删除操作的消息，根据情况进行自停止。

在收到 NodeDeleteMessage 后，Node Holder Actor 仅仅会转发该消息到目标（需要删除的）actor。

actor 在收到消息后，首先会判断是否收到过此批次的删除消息。同上文提到过的，如果一个节点存在多个上游节点，那么它会收到多个删除消息，那么在对其下游节点进行转发消息时只需要发送一次删除消息即可。

第二步，在 actor 初始化的时候会在其内部维护一个上游节点的集合。此时的 deleteFlag 已经被 Root Chain 中的 Router Node 置为 true，actor 会从上游节点集合中删除第一个元素。这里尚不需要做到精确匹配到需要 remove 哪一个上游节点元素。只有在收到所有的上游节点的删除消息后，actor 才会自停止。防止如果提前终止，那么可能会存在尚未到达的消息形成 Dead Letter。

```
super.logReceivedSomeKindOfMessage(super.logWho, message);
List<MlotRuleEngineRuleRelationInitV> toRelationList = super.actor.getToRelationList();
// 如果删除时间标记已被标记 并且等于本次的删除标记 则认为删除message已经下发到下层node 不需要再次下发
if (message.getDeleteTimeStamp() != super.actor.getDeleteTimeStamp() && CollectionUtil.isNotEmpty(toRelationList)) {
    super.actor.setDeleteTimeStamp(message.getDeleteTimeStamp());
    super.sendToNextNodeHolderActor(super.actor.getNodeId(), message, toRelationList, super.actor.getChainIdMap(), super.actor.getActorHolderMap(), false);
}
// 所有的上层delete message 都到达后 才会停止此node actor
if (message.isDeleteFlag()) {
    List<MlotRuleEngineRuleRelationInitV> fromRelationList = super.actor.getFromRelationList();
    fromRelationList.remove(0);
    if (CollectionUtil.isEmpty(fromRelationList)) {
        super.stopActor(super.actor.self());
    }
}
```

actor 自停止后，由于 death watch 的存在，负责监管其的 Node Holder Actor 会收到特定的消息 NodeStopMessage。然后 Node Holder Actor 会将此 actor 从自己的内存 map 中删除，并且发送一条 NodeStopDoneMessage 给 System Node Holder Actor。

```
// 从监管map中移除actor
BiMap<ActorRef, NodeId> nodeIdBiMapInv = super.actor.getNodeIdBiMap().inverse();
if (nodeIdBiMapInv.containsKey(actorRef)) {
    NodeId nodeIdV = nodeIdBiMapInv.remove(actorRef);
    NodeLifeCycleInfo nodeLifeCycleInfo = super.actor.getNodeLifeCycleMap().remove(nodeIdV);
    if (null == nodeLifeCycleInfo) {
        super.logNodeLifeCycleIsNull(super.actor.getType(),getNodeHolderActorClass().getSimpleName(), nodeStopMessage, nodeIdV);
    }
    this.bLog.logWhoOccursWhenDebug(super.actor.getType(),getNodeHolderActorClass().getSimpleName(), "a debug", String.format("process %s",
nodeStopMessage.getClass().getSimpleName()), "remove actor ref: " + actorRef.path() + " node id: " + senderNodeIdV.getId());

    super.actor.getActorSystemContext().getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM).tell(new NodeMessageCarrier<>(senderNodeIdV, senderNodeIdV, new
NodeStopDoneMessage(nodeStopMessage.getNodeInitV())), super.actor.self());
} else {
    this.bLog.logWhoOccursWhenError(super.actor.getType(),getNodeHolderActorClass().getSimpleName(), "an error", String.format("process %s",
nodeStopMessage.getClass().getSimpleName()), "find no actor ref: " + actorRef.path() + " node id: " + senderNodeIdV.getId());
}
```

System Node Holder Actor 在接收到 NodeStopDoneMessage 后，会从内部维护的 Rule Chain 上下文中取出该链路的信息，从信息中移除该已经停止的节点。如果该链路的所有节点都被移除掉，则表示该链路已经被完全从 Actor System 内存中删除。如果仅仅是删除链路，此时方法就会结束。如果是对于链路的更新，System Node Holder Actor 会更新该链路的上下文信息，然后从上下文中取出更新后的链路信息——NodeInitMessage，发送此消息到更新链路的 first node 来完成对新链路的初始化。

```
SystemRuleChainContext ruleChainContext = SystemNodeHolderActor.ruleChainIdContextMap.get(ruleChainId);
if (null == ruleChainContext) {
    super.logRuleChainContextNotExists(super.actor.getType(),getNodeHolderActorClass().getSimpleName(), nodeStopDoneMessage, ruleChainId);
    return;
}
// system actor holder 中缓存的每个链路的context 如果链路中的节点都stop done则此集合为empty, 说明链路全部node已经stop 可以进行初始化
List<MlotRuleEngineRuleNodeInitV> nodeInitVList = ruleChainContext.getNodeInitVList();
nodeInitVList.removeIf(endNodeInitV -> endNodeInitV.getId().equals(nodeId));
if (CollectionUtil.isEmpty(nodeInitVList)) {
    // 进行初始化
    NodeInitMessage nodeInitMessage = ruleChainContext.getUpdateInitMessage();
    if (null == nodeInitMessage) {
        SystemNodeHolderActor.ruleChainIdContextMap.remove(ruleChainId);
    } else {
        Map<String, MlotRuleEngineRuleNodeInitV> ruleNodeIdMap = nodeInitMessage.getRuleNodeIdMap();
        initRuleChainContext(nodeInitMessage.getRelationGroupByFromIdMap(), ruleChainId, ruleNodeIdMap);
        SystemNodeHolderActor.ruleChainIdContextMap.get(ruleChainId).getStartNodeInitVList().parallelStream().forEach(startNodeInitV -> {
            String startNodeId = startNodeInitV.getId();
            ActorTypeMapperEnum startNodeActorTypeEnum = ActorTypeMapperEnum.valueOf(ruleNodeIdMap.get(startNodeId).getNodeType());
            super.actor.getActorSystemContext().getActorHolderMap().get(startNodeActorTypeEnum)
                    .tell(new NodeMessageCarrier<>(new NodeId(startNodeId),
                                    null,
                                    nodeInitMessage),
                            ActorRef.noSender());
        });
        ruleChainContext.setUpdateInitMessage(null);
    }
}
```

提示

这里体现了一个顺序性的概念，即在更新链路的时候，Actor System 会保证旧的链路已经完全从内存中删除，再去初始化新的链路。由于 actor 消息处理的异步性，保证此顺序性概念的严谨是非常重要的。开发者可在后续的开发中慢慢体会。

3.2.6 NodeUpdateMessage 的处理

这一小节，我们会结合之前两个小节的内容来对 NodeUpdateMesssage 的处理行为做详细解读。

同样是 System Node Holder Actor 会收到这一类消息。首先会从自身的规则链缓存中找到需要更新的规则链的上下文。如果上下文中的 updateInitMessage 成员变量不为 null，则说明上一次的更新还未进行完成。这时，只需要将本次更新的规则链的最新初始化信息替换掉，等待更新进程调用即可。

通常，上述情况并不会发生，代码会继续运行。我们会设置好 updateInitMessage 成员变量以供后续在删除了规则链后，对新规则链初始化进行使用。最后我们拿到需要更新的链路的 first node，给其发送 NodeUpdateMessage，对于其余的链路 head node，我们发送的是 NodeDeleteMessage。需要注意的是，这个时候的 deleteFlag 已经为true。

提示

这里重申一下 head node 的定义。即为链路中没有任何上游关系的节点，故 first node 也是一种 head node。

```
MlotRuleEngineRuleChainInitV ruleChainInitV = nodeUpdateMessage.getNodeDeleteMessage().getRuleChainV();
String ruleChainId = ruleChainInitV.getId();
SystemRuleChainContext ruleChainContext = SystemNodeHolderActor.ruleChainIdContextMap.get(ruleChainId);
if (null == ruleChainContext) {
    super.logRuleChainContextNotExists(super.actor.getType().getNodeHolderActorClass().getSimpleName(), nodeUpdateMessage, ruleChainId);
} else {
    if (null != ruleChainContext.getUpdateInitMessage()) {
        // 上次更新还未完成
        super.bLog.logWhoOccursWhenError(super.actor.getType().getNodeHolderActorClass().getSimpleName(), "an error", String.format("process %s",
nodeUpdateMessage.getClass().getSimpleName()), "rule chain[" + ruleChainId + "] context is updating, update init message: " + ruleChainContext.getUpdateInitMessage());
        ruleChainContext.setUpdateInitMessage(nodeUpdateMessage.getNodeInitMessage());
        return;
    }
    // 设置需要更新的message 环境
    ruleChainContext.setUpdateInitMessage(nodeUpdateMessage.getNodeInitMessage());
    String firstNodeId = ruleChainInitV.getFirstNodeId();
    // 给需要更新的链路所有从node发送message first node 会发送update message 其余的发送delete message
    ruleChainContext.getStartNodeInitVList().parallelStream().forEach(nodeInitV -> {
        ActorRef targetRef = super.actor.getActorSystemContext().getActorHolderMap().get(ActorTypeMapperEnum.valueOf(nodeInitV.getNodeType()));
        if (nodeInitV.getId().equals(firstNodeId)) {
            targetRef.tell(messageCarrier, super.actor.self());
        } else {
            targetRef.tell(new NodeMessageCarrier<>(new NodeId(nodeInitV.getId()), null, nodeUpdateMessage.getNodeDeleteMessage()), super.actor.self());
        }
    });
}
```

开发者可能有所疑惑，这里为什么发送 NodeUpdateMessage，而不是 NodeDeleteMessage，这里解释一下原因。

更新链路操作是一种比较特殊的操作，我们需要 Actor System 在获得更新通知的时候，下一个目标为该链路（在 Ulink 中即为该产品）的数据消息就会按照新的链路规则来进行处理，也就是说，在链路尚未更新成功的时候，我们既不能丢失消息，也不能用旧的方式去处理消息，这里有很强的同步性和顺序性。既然 first node 是每个链路的唯一入口，那么其他的 head node 我们可以发送 NodeDeleteMessage 来完成链路的删除操作。而 first node 我们需要做一些特殊的处理，来保证我们所需要的非常严谨的功能实现。这个处理便是 NodeUpdateMessage。实际上也就是告诉 holder 此目标 actor 需要锁定，待后续更新完成（即初始化完成）后再解锁。锁定期间发送至该 actor 所有的消息都会被存到holder的内存中，解锁后 holder 会首先按照原顺序释放这些消息到 first node 中，然后再处理后续消息。这样做既可以保证消息不会被丢弃，又能做到消息的顺序不被打乱。

现在我们看一下 holder 在接收到 NodeUpdateMessage 会做什么。

```
// lock this chain (new message will forward to itself for futher consuming) (unlock in process init message)
super.actor.getLockedFirstNodeIdVMap().put(nodeId9, Lists.newArrayList());
// delete old chain via sending a delete message
ActorRef oldActorRef = super.actor.getNodeIdActorMap().get(nodeId9);
if (null == oldActorRef) {
    super.bLog.logWhoOccursWhenError(super.actor.getType().getNodeHolderActorClass().getSimpleName(), "an error", String.format("process %s",
nodeUpdateMessage.getClass().getSimpleName()), "target actor ref not exists, target node id: " + nodeId9.getId() + ", source node id: " + senderNodeId9.getId());
    this.tellSenderNodeNotExists(nodeId9);
    return;
}
oldActorRef.tell(nodeUpdateMessage.getNodeDeleteMessage(), super.actor.self());
```

首先，正如上述所说，holder 会锁定目标 actor 也就是需要更新的规则链路的 first node。然后再 tell actor 一个 NodeDeleteMessage。

接下来的事情相信开发者已经很清楚了。first node 后续的所有 actor 会同该链路的其余 head node 的后续 actor 一样，按照 message 的流动顺序逐个删除，逻辑和前面章节讲述的 NodeDeleteMessage 完全一样。

之前我们也说过，删除成功后，System Node Holder Actor 会收到 NodeStopDoneMessage，在 remove 规则链上下文中的节点集合的最后一个元素时，会检查是否有需要更新的初始化信息 updateInitMessage 成员变量，如果有，则发送该变量的中的初始化消息给新规则链的 first node，并且将 updateInitMessage 置为 null，表示更新操作完成。

这里如果开发者还有疑问，可以回顾 3.2.5 NodeDeleteMessage 的处理。

3.2.7 节点介绍

由于 Rule Engine 节点的种类繁多，每个节点的功能也并不会固定不变，所以笔者不打算详细介绍每个节点的功能和其处理器的原理。感兴趣的开发者可以自行参考源码，源码中对于每类节点的生命周期，processor 都有做必要的解释。如果有开发新类节点的必要，还请遵循 Rule Engine 的原设计理念。

4. 兼容服务一体化平台

Ulink 在编写之初是要替代原有的物联网平台，这样就会对 Rule Engine 有所要求，可以兼容现有服务一体化平台的内容，例如规则配置为图表式，而非 ThingsBoard 的拖拽式。本章会围绕着兼容的功能模块做逐一介绍，如果今后的 Ulink 做更新升级，可考虑忽略本章。

注意

在阅读此章内容前，有必要先掌握服务一体化平台对于物联网的大部分需求和业务逻辑，本章内对于服务一体化相关的业务逻辑不会做过多解释。

4.1 服务一体化流程图及产品初始化

如图4-1所示，目前服务一体化中物联网规则引擎的业务流程还不是很复杂，只是用到几个基础节点类型。由于服务一体化没有规则引擎拖拽式建立的界面，故我们除了会在数据库中添加一条 Root Chain外，还会在新增产品的时候为该产品创建一条默认的规则链路，该链路满足所有新建产品的功能，并可以新增数据端点和预警配置。其中，每个产品在创建的时候会添加两个默认的 Data Check Node，这两个节点不会做任何 check 行为，而且不会将业务数据流转到下游节点。作用仅为服务一体化添加预警配置时，可以方便获取到数据端点/设备状态的某些必要节点上下文信息。

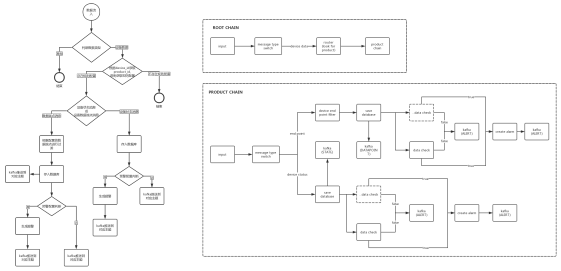

图4-1

4.2 兼容

4.2.1 服务一体化添加产品

服务一体化添加产品成功后会调用 Ulink 接口，在 Rule Engine 里会做两步：

1. 向数据库中添加初始化产品的规则链路、节点、关系三种信息。
2. 向 Actor System 系统内存中添加该产品链（规则链）。

```
// init default product rule chain in db
String ruleChainId = iotRuleChainService.addNewProductChain(mIotRuleEngineF);
// add rule chain to actor system
defaultActorService.addRuleChainsToActorSystem(Lists.newArrayList(ruleChainId));
```

像数据库中添加三种信息代码略长，但是逻辑很简单。仅仅是按照图4-1中产品链的链路关系图构建一个规则链信息，感兴趣的开发者可以参阅源码中相关部分。

第二步，向内存中添加产品链也就是前文提到过的初始化规则链路，有疑问可回顾 3.2.1 项目初始化 中的**初始化所有链路**部分。

4.2.2 服务一体化删除产品

同添加产品类似，删除产品，在 Rule Engine 中也会做两个步骤：

1. 从数据库中删除有关该产品的所有规则链路信息。
2. 从 Actor System 系统内存中删除所有对应的规则链路信息。

```
// delete product rule chains in db
List<MIotRuleEngineRuleChainInitV> ruleChainVList = iotRuleChainService.deleteProduct(productIdList);
// delete rule chains from actor system
defaultActorService.deleteRuleChainsFromActorSystem(ruleChainVList);
```

其中的逻辑也并不复杂，由于拿到了 productId，那么我们可以获取到 Rule Engine 中保存的有关该产品的所有信息，数据库中删除后，再从 Actor System 中删除，此处有疑问可回顾 3.2.2 规则链路的删除。

注意

这里不会删除 Root Chain，因为拿到的是产品id，根据产品id删除对应的规则链路，而 Root Chain 没有任何 belong_to_id。

4.2.3 服务一体化变更数据端点

变更数据端点包含了数据端点的添加、更新、删除。所以传进来的表单是目前该产品的所有数据端点，如传入的数据端点为空集合，则为该产品没有任何数据端点。

逻辑依旧是先操作数据库，然后再去更新 Actor System 中的这条规则链路。这里我们已经开始用到规则链路的更新了，有疑问可回顾 3.2.3 规则链路的更新。

```
// update product rule chain's datafilter node configuration
List<MIotRuleEngineRuleChainInitV> ruleChainVList = iotRuleChainService.updateEndPointFilter(mIotRuleEngineF);
// update rule chain in actor system
defaultActorService.updateRuleChainsInActorSystem(ruleChainVList.parallelStream()
    .filter(Objects::nonNull).map(MIotRuleEngineRuleChainInitV::getId).collect(Collectors.toList()));
```

4.2.4 服务一体化新增预警配置

新增预警配置较为复杂，因为我们需要在数据端点/设备状态的初始化 Data Check Node 并联一个 Data Check Node，并添加对应的 js 规则。

node 构建

首先我们会添加好需要构建的 node。第一步则是将服务一体化的预警规则转化为 Rule Engine 的 check js。

```
String judgingConditionConvert = null;
Byte deviceStatusConvert = 0;
if (deviceStatus != null) {
    deviceStatusConvert = deviceStatus == 2 ? 0 : deviceStatus;
}
if (!StringUtils.isNumeric(value)) {
    value = "\"" + value + "\"";
}
if (triggerType == 1) {
    switch(judgingCondition) {
        case "≠":
            judgingConditionConvert = "!=";
            break;
        case "=":
            judgingConditionConvert = "==";
            break;
        default:
            judgingConditionConvert = judgingCondition;
            break;
    }
}
String endPointNameConvert = triggerType == 1 ? endPointName : ConstantFields.STATUS;
String valueConvert = triggerType == 1 ? value : deviceStatusConvert.toString();
judgingConditionConvert = triggerType == 1 ? judgingConditionConvert : "==";
return String.format(ConstantFields.JS_SCRIPT_II, endPointNameConvert + judgingConditionConvert + valueConvert.toLowerCase());
```

这里需要注意的点很多，首先如果规则的 value 不是数值型，而是字符串型，或者是布尔型（TRUE/FALSE）我们需要对 value 加上双引号处理，在 js 中就会构建成例如 a = “true”，这样的形式。对于字符串型开发者可能比较容易理解，对于布尔型则是因为目前从设备上报的情况来看，布尔型的数据端点或者设备状态也是将 value 转成字符串传到 Rule Engine。如果有设备这里的传入的是布尔值，这里的做法可能需要重新考虑。

其次由于服务一体化的关系符号与 js 中的不尽相同，所以需要进行转换，例如“≠"、“=”。

之后对于状态的描述服务一体化值的范围为 1、2，需要转换为 Rule Engine 的 0、1。

最后把所有大写字母转换为小写，主要是针对布尔类型的。因为服务一体化的布尔都为大写，而设备端上传的而小写，故做转换。

将生成好的 js 字符串传入到 node 的 configuration 中，我们就可以直接插入该 node，完成 node 的构建了。

relation 构建

构建 relation 时，我们会从该规则链所有的关系中找到数据端点的Save DB Node 的 id 和 所有从 Save DB Node → Data Check Node 的关系。

然后再判断该预警规则为数据端点，还是设备状态，根据情况不同，最终取出对应的可能为多个的 Data Check Node 中的一个 id。这样做的目的也是为了获取构建新关系所需要的下游 Create Alarm Node 与 Kafka Node 的 id。

```
List<MIotRuleEngineRuleRelationInitV> ruleChainRelations = iotRuleRelationService.getRuleRelationsForActorSystem(Lists.newArrayList(ruleChainId));
List<MIotRuleEngineRuleRelationInitV> neededRelations = Lists.newArrayList();
AtomicReference<String> endPointSaveDBId = new AtomicReference<>();
ruleChainRelations.forEach(relation -> {
    if (relation.getFromType().equals(IotRuleRelationFromTypeEnum.RULE_NODE.name())) {
        if (relation.getFromNodeType().equals(ActorTypeMapperEnum.DATA_FILTER.name())) {
            endPointSaveDBId.set(relation.getToId());
        }
        if (relation.getFromNodeType().equals(ActorTypeMapperEnum.SAVE_DB.name())
                && relation.getToNodeType().equals(ActorTypeMapperEnum.DATA_CHECK.name())) {
            neededRelations.add(relation);
        }
    }
});
if (StringUtils.isBlank(endPointSaveDBId.get()) || neededRelations.size() < 2) {
    throw new BizException(IotRuleNodeServiceErrorMessage.RELATION_DATA_ERROR);
}

boolean isEndPoint = triggerType == 1;
Optional<MIotRuleEngineRuleRelationInitV> saveDBRelationOpt = neededRelations.stream()
        .filter(relation -> isEndPoint == relation.getFromId().equals(endPointSaveDBId.get())).findAny();
if (!saveDBRelationOpt.isPresent()) {
    throw new BizException(IotRuleNodeServiceErrorMessage.SAVE_DB_RELATION_DATA_NOT_EXISTS);
}
MIotRuleEngineRuleRelationInitV saveDBRelation = saveDBRelationOpt.get();
String dataCheckId = saveDBRelation.getToId();
```

这时我们也有了上游关系 id，即对应的 Save DB Node 的 id。就可以构建三个关系：

- 上游关系：Save DB Node → new Data Check Node。
- 下游 check false 关系：new Data Check Node → Kafka Node。
- 下游 check true 关系：new Data Check Node → Create Alarm Node。

关系构建完成之后，数据库的任务也随之完成，接下来就是前文提到过的更新 Actor System 内存中的该规则链路，可回顾 3.2.3 规则链路的更新。

4.2.5 服务一体化更新预警配置

更新预警配置中，我们只会把新的预警规则按照上文提过的 js 转换，转换成 Rule Engine 可识别的 js，然后修改该 Data Check Node 的 configuration。最后在 Actor System 中更新此规则链路。

4.2.6 服务一体化删除预警配置

由于我们在新增预警配置的时候都是在原先的预警配置基础上进行 Data Check Node 的并联操作，所以删除这些 node 的时候就会变得很简单。

我们只需要找到需要删除的 Data Check Node，然后找到它的所有上、下游关系，删除掉即可，这样做对该链路其他的 Data Check Node 不会有任何影响。当然，Actor System 中还是做更新规则链路的操作。

4.2.7 服务一体化更新产品

目前服务一体化在更新产品的时候，Rule Engine 所需要做出的反应仅仅是针对产品名称的变更。因为链路的名称与产品名称是一致的，为了问题的查询方便以及数据的一致性，我们还是会同步更改规则链路名称，这里的操作只是针对数据库，对于 Actor System 我们不会做任何事情。

结语

至此，Ulink 规则引擎的技术介绍全部结束，不过 Rule Engine 的发展道路才刚刚开始。如果读者通读此文，应该会为 Akka Actor 与 Rule Engine 的完美结合感到振奋，这其中的通用化、高度低耦合的核心设计使得 Rule Engine 会有很大的发展空间，也注定了 Rule Engine 的道路会是极其漫长与艰辛。对于目前 Rule Engine 的状态，笔者认为只是简单的框架完成，甚至基础功能的构建也还只是刚刚起步。在撰文的过程中，也发现很多性能不足、程序 bug 的问题点。好在我们有这样一篇记录着 Rule Engine 成长的史篇，可以在大多数情况下给予开发者、爱好者一定的技术帮助，也感谢后续的开发者能够继续完善此文，让 Rule Engine 成为一个日渐成熟的技术架构。