

Ulink 规则引擎

版本控制表

版本号	作者	参与者	日期	备注
1.0	李文		2019.06.26	

规则引擎

相关介绍

规则引擎起源于基于规则的专家系统，而基于规则的专家系统又是专家系统的其中一个分支。专家系统属于人工智能的范畴，它模仿人类的推理方式，使用试探性的方法进行推理，并使用人类能理解的术语解释和证明它的推理结论。

利用它就可以在应用系统中分离商业决策者的商业决策逻辑和应用开发者的技术决策，并把这些商业决策放在中心数据库或其他统一的地方，让它们能在运行时可以动态地管理和修改，从而为企业保持灵活性和竞争力提供有效的技术支持。

在需求里面我们往往把约束，完整性，校验，分支流等都可以算到业务规则里面。在规则引擎里面谈的业务规则重点是谈当满足什么样的条件的时候，需要执行什么样的操作。因此一个完整的业务规则包括了条件和触发操作两部分内容。而引擎是事物内部的重要的运行机制，规则引擎即重点是解决规则如何描述，如何执行，如何监控等一系列问题。

规则引擎由推理引擎发展而来，是一种嵌入在应用程序中的组件，实现了将业务决策从应用程序代码中分离出来，并使用预定义的语义模块编写业务决策。接受数据输入，解释业务规则，并根据业务规则做出业务决策。

规则引擎的优点

- 声明式编程
规则可以很容易地解决困难的问题，并得到解决方案的验证。与代码不同，规则以较不复杂的语言编写; 业务分析师可以轻松阅读和验证一套规则。
- 逻辑和数据分离
数据位于“域对象”中，业务逻辑位于“规则”中。根据项目的种类，这种分离是非常有利的。
- 速度和可扩展性
在规则引擎的帮助下，您的应用程序变得非常可扩展。如果频繁更改请求，可以添加新规则，而无需修改现有规则。

- 规则引擎
 - 相关介绍
 - 规则引擎的优点
- Ulink's Rule Engine
 - 1. 核心技术
 - 1.1 Akka Actor 模型
 - 1.1.1 什么是 Akka
 - 1.1.2 Actor 模型的起源
 - 1.1.3 什么是 Actor
 - 1.1.4 Actor 和消息传递
 - 1.1.5 Actor 的重要概念
 - 1.1.6 Actor 解决多线程的状态共享问题
 - 1.2 Akka Actor 简单使用
 - 1.2.1 构建maven依赖
 - 1.2.2 创建第一个 Actor
 - 1.2.3 Actor 创建详述
 - 1.2.4 Actor 支持远程
 - 1.2.5 Actor 的 tell
 - 1.2.6 Actor 的监督策略
 - 1.3 Akka Actor 生命周期
 - 1.3.1 Actor 生命周期
 - 1.3.2 重启和停止时的消息处理
 - 1.3.3 终止或 kill 一个 Actor
 - 1.3.4 生命周期监控和 DeathWatch
 - 1.3.5 安全重启
 - 1.4 Akka Actor Router
 - 1.4.1 Router 介绍
 - 1.4.2 路由逻辑
 - 1.4.3 向 Router 中的所有 Actor 发送消息
 - 1.4.4 监督 Router Pool 中的路由对象
 - 1.5 Akka Actor Dispatcher
 - 1.5.1 Dispatcher 解析
 - 1.5.2 Executor

- 知识集中化
通过使用规则，您创建一个可执行的知识库。这是商业政策的一个真理点。理想情况下，规则是可读的，它们也可以用作文档。

Ulink's Rule Engine

1. 核心技术

1.1 Akka Actor 模型

1.1.1 什么是 Akka

Akka一词据说来源于瑞典的一座山，我们说到 Akka 时，通常是指一个分布式工具集，用于协调远程计算资源来进行一些工作。Akka 是 Actor 并发模型的一种现代化实现。现在的 Akka 可以认为是从许多其他技术发展演化而来的，它借鉴了 Erlang 的 Actor 模型实现，同时又引入了许多新特性，帮助构建能够处理如今大规模问题的应用程序。

1.1.2 Actor 模型的起源

Actor 并发模型最早出现于一篇叫作《A Universal Modular Actor Formalism for Artificial Intelligence》的论文，该论文发表于 1973 年，提出了一种并发计算的理论模型，Actor 就源于该模型。Actor 模型的特性能够在并发计算中帮助我们解决共享状态带来的常见问题。

1.1.3 什么是 Actor

在 Actor 模型中，Actor 是一个并发原语：更简单地说，可以把一个 Actor 看作是一个工人，就像能够工作或是处理任务的进程和线程一样。把 Actor 看成是某个机构中拥有特定职位及职责的员工可能会对理解有所帮助。比如说一个寿司餐馆。餐馆的职员需要做各种各样不同的工作，给客人准备餐盘就是其中之一。

1.1.4 Actor 和消息传递

在面向对象编程语言中，对象的特点之一就是能够被直接调用：一个对象可以访问或修改另一个对象的属性，也可以直接调用另一个对象的方法。这在只有一个线程进行这些操作时是没有问题的，但是如果多个线程同时读取并修改同一个值，那么可能就需要进行同步并加锁。

Actor 和对象的不同之处在于其不能被直接读取、修改或是调用。反之，Actor 只能通过消息传递的方式与外界进行通信。简单来说，消息传递指的是一个 Actor 可以接收消息，本身可以发送消息，也可以对接收到的消息作出回复。尽管我们可以将这种方式与向某个方法传递参数并接收返回值进行类比，但是消息传递与方法调用在本质上是不同的：消息传递是异步的。无论是处理消息还是回复消息，Actor 对外界都没有依赖。

Actor 每次只同步处理一个消息。邮箱本质上是等待 Actor 处理的一个工作队列，**如图 1-1 所示**。处理一个消息时，为了能够做出响应，Actor 可以修改内部状态，创建更多 Actor 或是将消息发送给其他 Actor。

- 1.5.3 创建 Dispatcher
- 1.5.4 决定何时使用哪种 Dispatcher
- 1.5.6 并行最优化
- 1.6 Akka Actor 集群化
 - 1.6.1 Akka Cluster 介绍
 - 1.6.2 巨型单体应用 vs 微服务
 - 1.6.3 集群的定义
 - 1.6.4 集群的失败检测
 - 1.6.5 通过 gossip 协议达到最终一致性
 - 1.6.6 使用 Akka Cluster 构建系统
 - 1.6.7 创建集群
 - 1.6.8 启动集群
 - 1.6.9 通过路由向集群发送消息
- 1.7 Akka Actor 邮箱
 - 1.7.1 邮箱配置
 - 1.7.2 在部署配置中选择邮箱
 - 1.7.3 在代码中选择邮箱
 - 1.7.4 决定使用哪个邮箱
 - 1.7.5 提高邮箱中消息的优先级
- 1.8 参考资料
- 2. 架构概念
 - 2.1 整体架构图
 - 2.2 目录结构图
 - 2.3 概念描述
 - 2.3.1 设计概念
 - 2.3.2 链路概念
 - 2.3.3 关系概念
 - 2.3.4 节点概念
- 3. 原理与开发
 - 3.1 项目配置
 - 3.1.1 项目依赖
 - 3.1.2 配置文件
 - 3.1.3 数据库表
 - 3.1.4 服务依赖配置
 - 3.2 原理流程
 - 3.2.1 项目初始化
 - 3.2.2 规则链路的删除
 - 3.2.3 规则链路的更新
 - 3.2.4 NodeInitMessage 的处理

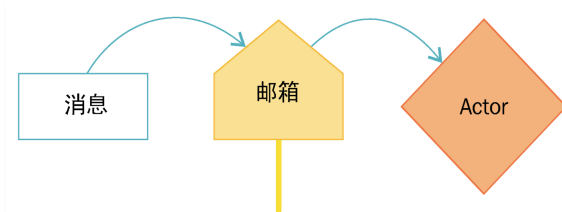


图1-1

1.1.5 Actor 的重要概念

在具体实现中，我们通常使用 Actor 系统这个术语来表示多个 Actor 的集合以及所有与该 Actor 集合相关的东西，包括地址、邮箱以及配置。下面再重申一下这几个重要的概念：

- Actor：一个表示工作节点的并发原语，同步处理接收到的消息。Actor 可以保存并修改内部状态。
- 消息：用于跨进程（比如多个 Actor 之间）通信的数据。
- 消息传递：一种软件开发范式，通过传递消息来触发各种行为，而不是直接触发行。
- 邮箱地址：消息传递的目标地址，当 Actor 空闲时会从该地址获取消息进行处理。
- 邮箱：在 Actor 处理消息前具体存储消息的地方。可以将其看作是一个消息队列。
- Actor系统：多个 Actor 的集合以及这些 Actor 的邮箱地址、邮箱和配置等。

虽然现在看来可能还不是太明显，但是 Actor 模型要比命令式的面向对象并发应用程序容易理解多了。我们可以举一个现实世界中的例子来比喻使用 Actor 模型来建模的过程，这会帮助我们理解它带来的好处。

比如有一个寿司餐馆，其中有 3 个 Actor：客人、服务员以及厨师。

首先，客人向服务员点单。服务员将客人点的菜品写在一张纸条上，然后将这张纸条放在厨师的邮箱中（将纸条贴在厨房的窗户上）。当厨师有空闲的时候，就会获取这条消息（客人点的菜品），然后就开始制作寿司（处理消息），直至寿司制作完成。寿司准备好以后，厨师会发送一条消息（盛放寿司的盘子）到服务员的邮箱（厨房的窗户），等待服务员来获取这条消息。此时厨师可以去处理其他客人的订单。

当服务员有空闲时，就可以从厨房的窗户获取食物的消息（盛放寿司的盘子），然后将其送到客人的邮箱（比如餐桌）。当客人准备好的时候，他们就会处理消息（吃寿司），如图 1-2 所示。

- 3.2.5 NodeDeleteMessage 的处理
- 3.2.6 NodeUpdateMessage 的处理
- 3.2.7 节点介绍
- 4. 兼容服务一体化平台
 - 4.1 服务一体化流程图及产品初始化
 - 4.2 兼容
 - 4.2.1 服务一体化添加产品
 - 4.2.2 服务一体化删除产品
 - 4.2.3 服务一体化变更数据端点
 - 4.2.4 服务一体化新增预警配置
 - 4.2.5 服务一体化更新预警配置
 - 4.2.6 服务一体化删除预警配置

- 结语

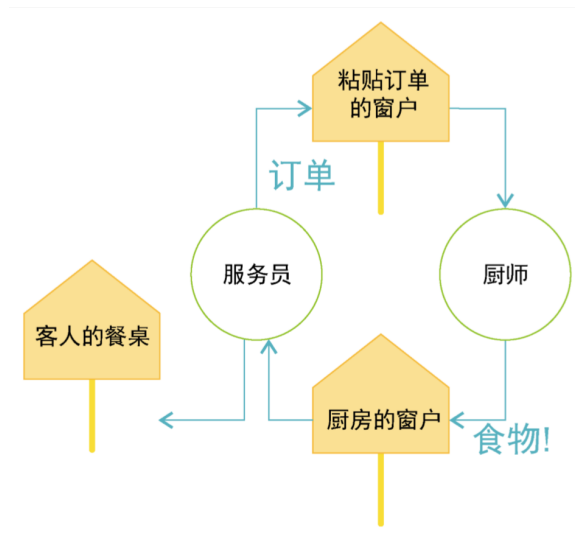


图1-2

运用餐厅的运作来理解 Actor 模型是很容易的。随着越来越多的客人来到餐厅，我们可以想象服务员每次接收一位客人的下单，并将订单交给厨房，接着厨师处理订单制作寿司，最后服务员将寿司交给客人。每个任务都可以并发进行。这就是 Actor 模型提供的最大好处之一：当每个人各司其职时，使用 Actor 模型分析并发事件非常容易。而使用 Actor 模型对真实应用程序的建模过程和本例中对寿司餐厅的建模过程并没有太大差异。

1.1.6 Actor 解决多线程的状态共享问题

Actor 模型的另一个好处就是可以消除共享状态。因为一个 Actor 每次只处理一条消息，所以可以在 Actor 内部安全地保存状态。如果读者此前没有接触过并发系统，那么可能不是很容易马上理解这一点。不过我们可以用一种简单的方式进行说明。假设我们尝试执行两个操作，同时读取、修改并保存一个变量，那么如果我们不进行同步操作并加锁的话，其中的一个操作结果将丢失。这是一个非常容易犯的错误。

在下面的例子中，有两个线程同时执行一个非原子的自增操作。让我们来看看在线程间共享状态会带来什么结果。会有多个线程从内存中读取一个变量值，将该变量自增，然后将结果写回内存。这就是竞态条件（Race Condition），可以通过保证某一时刻只有一个线程访问内存中的值来解决其带来的一部分问题。下面用一个 Java 的例子进行说明。

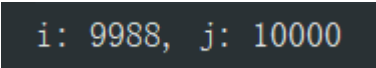
如果我们尝试使用多个线程并发地对一个整型变量执行 100 000 次自增操作，那么极有可能会丢失掉其中一些自增操作的结果。

```
private static int i, j;

public static void main(String... args) throws Exception {
    ExecutorService es = Executors.newFixedThreadPool(10);
    IntStream.range(0, 10000).forEach(i -> es.submit(Ex1::addI));
    IntStream.range(0, 10000).forEach(i -> addJ());
    Thread.sleep(1000);
    System.out.println(String.format("i: %d, j: %d", i, j));
}
```

```
private static void addI() {  
    i++;  
}  
  
private static void addJ() {  
    j++;  
}
```

我们使用 `x = x + 1` 这个简单的函数将 `i` 和 `j` 都自增了 100 000 次，其中 `i` 的自增操作通过多个线程并发执行，而 `j` 的自增操作则只通过一个线程来执行。等待 1 秒钟后，我们再打印运行结果，确保所有更新都已经完成。读者可能会认为运行结果是 100000 100000，然而结果却并非如此，如图 1-3 所示。



```
i: 9988, j: 10000
```

图1-3

共享状态是不安全的。如果两个线程同时读取一个值，将该值自增，然后写回内存，那么由于该值同时被多个线程读取，其中的某些操作结果将会丢失。这就是竞态条件，也是使用共享状态的并发模型存在的最基本的问题之一。

为了保证内存中的共享状态值不出现错误，我们可以使用锁和同步机制，防止多个线程同时读取并写入同一个值。这就会导致问题变得更为复杂，更难理解，也更难确保结果的正确。

使用共享状态带来的最大威胁就是代码在测试中看上去经常是正确的，但是一旦有多个线程并发运行时，就会时不时地出现一些错误。由于测试时通常都不会出现多线程开发的情况，因此这些 Bug 很容易被忽略。Dion Almaer 曾经在博客中写到过，大多数 Java 应用程序都存在大量的并发 Bug，因此有时能正确运行，有时却运行失败。Actor 通过减少共享状态来解决这一问题。如果我们把状态移到 Actor 内部，那么只有该 Actor 才能访问其内部的状态（实际上只有一个线程能够访问这些内部状态）。如果把所有消息都看做是不可变的，那么我们实际上可以去除 Actor 系统中的所有共享状态，构建更安全的应用程序。

1.2 Akka Actor 简单使用

1.2.1 构建maven依赖

Akka Actor 是一个高度集成的高并发解决方案，其中常用的依赖分为以下几个部分：

- akka-actor_2.12 : Actor 核心包。
- akka-testkit_2.12 : Actor 测试包，提供基础的测试工具，其中将 Actor 收发消息的操作改为同步最为常用。
- akka-remote_2.12 : Actor 远程调用包，用于搭建可以远程消息传递的 Actor。
- akka-cluster_2.12 : Actor 集群包，用于搭建 Actor 集群。
- akka-cluster-tools_2.12 : Actor 集群客户端，用于访问远程搭建的 Actor 集群。

```
<dependency>  
    <groupId>com.typesafe.akka</groupId>
```

```

    <artifactId>akka-actor_2.12</artifactId>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-testkit_2.12</artifactId>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_2.12</artifactId>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster_2.12</artifactId>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-tools_2.12</artifactId>
</dependency>
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-slf4j_2.12</artifactId>
</dependency>

```

1.2.2 创建第一个 Actor

首先构造消息

我们将从构造一个 SetRequest 消息开始。要记住的是，Actor 需要从其邮箱中获取消息，并查看消息中的操作指示。我们通过消息的类/类型来决定具体的操作。消息类型的内容具体描述了如何实现 API 协议。在本例中，我们在消息中使用 String 作为键，Object 作为值。

消息必须永远是不可变的，这样可以确保我们和我们的团队不通过多个执行上下文/线程来做一些不安全的操作，从而避免一些奇怪而又出人意料的行为。同样要记住这些消息除了会发送给本地的 Actor 以外，也可能会发送给另一台机器上的 Actor。如果可能的话，把所有值都定义为 final，并且使用不可变集合及类型，比如 Google Guava 所提供的集合及类型。

下面使用 Java 将 SetRequest 消息实现为一个不可变对象。这是在 Java 中实现不可变对象的一种相当标准的方法。任何熟练的 Java 开发者应该都对此很熟悉。一般来说，我们应该始终在所有代码中优先使用不可变对象。

```

public class SetRequest {
    private final String key;
    private final Object value;

    public SetRequest(String key, Object value) {
        this.key = key;
        this.value = value;
    }

    public String getKey() {
        return key;
    }
}

```

```

    public Object getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "SetRequest{" +
            "key='" + key + '\'' +
            ", value=" + value +
            '}';
    }
}

```

定义 Actor 收到消息后的响应

既然已经定义好了消息，现在我们就可以创建 Actor，并描述 Actor 接收到消息后如何做出响应。作为例子的开始，我们将在响应中做两件事：

1. 将消息输出到日志。
2. 将任何 Set 消息中的内容保存起来，以供之后使用。

下面的代码展示了使用 Java 实现的 Actor 收到消息后的响应

```

public class RequestReceiver extends AbstractActor {
    protected final LoggingAdapter log = Logging.getLogger(context().system(), this);
    protected final Map<String, Object> map = new HashMap<>();

    @Override
    public Receive createReceive() {
        return ReceiveBuilder.create()
            .match(SetRequest.class, message -> {
                log.info("Received Set request: {}", message);
                map.put(message.getKey(), message.getValue());
            }).matchAny(o -> log.info("received unknown message: {}", o))
            .build();
    }
}

```

Actor 是一个继承了 AbstractActor (Java8 的 Java Akka API) 的 Java 类。

我们重写了 createReceive 方法，在方法中，通过连续调用 ReceiveBuilder 的几个方法，生成最终的 ReceiveBuilder。这样就描述了 Actor 接收到不同类型消息时该如何做出响应。

ReceiveBuilder 的 match 方法有点儿像 case 语句，只不过 match 方法可以匹配类的类型。更正式地说，这其实就是模式匹配。在调用的 match 方法中，我们定义：如果消息的类型是 SetRequest.class，那么接受该消息，打印日志，并且将该 Set 消息的键和值作为一条新纪录插入到 map 中。其次，我们捕捉其他所有未知类型的消息，直接输出到日志。

使用单元测试验证代码

我们在这里将会使用 `testkit` 中提供的 `TestActorRef`，而不是普通的 `ActorRef`。`TestActorRef` 是通用的，有两个功能：首先，它提供的 `Actor API` 是同步的，这样我们就不需要在测试中考虑并发的問題；其次，我们可以通过 `TestActorRef` 访问其背后的 `Actor` 对象。

说得更清楚一点，Akka 隐藏了真实的 `Actor`（`RequestReceiver`），提供了指向 `Actor` 的引用，我们可以将消息发送至该引用。这样就对 `Actor` 进行了封装，保证了没有任何人可以直接访问真正的对象实例，只能进行消息传输。

接下来让我们来看看源代码，然后逐行解释。

```
public class RequestReceiverTest {
    ActorSystem system = ActorSystem.create("actorSystem");

    @Test
    public void itShouldPlaceKeyValueFromSetMessageIntoMap() {
        TestActorRef<RequestReceiver> actorRef = TestActorRef.create(system, Props.create(RequestReceiver.
class));
        actorRef.tell(new SetRequest("key", "value"), ActorRef.noSender());
        RequestReceiver requestReceiver = actorRef.underlyingActor();
        Assert.assertEquals(RequestReceiver.map.get("key"), "value");
    }
}
```

这是我们第一次讲到与 `Actor` 的交互，所以有一些之前没出现过的代码。其中一部分是专门用于测试的，另一部分是与 `Actor` 的交互相关的。我们已经介绍过，`Actor` 系统是包含了所有 `Actor` 及其地址的一

个地方，所以在创建 `Actor` 之前，需要做的第一件事就是获取 `Actor` 系统的引用。在上面的测试示例里，我们将该引用存在一个变量中：

```
ActorSystem system = ActorSystem.create();
```

在创建完 `Actor` 系统之后，我们就可以在 `Actor` 系统中创建 `Actor` 了。正如之前提到的那样，我们将使用 Akka Testkit 来创建一个 `TestActorRef`，提供同步 `API`，并且允许我们访问其指向的 `Actor`。下面，我们就在 `Actor` 系统中创建 `Actor`：

```
TestActorRef<RequestReceiver> actorRef = TestActorRef.create(system, Props.create(RequestReceiver.class));
```

我们调用 Akka Testkit 中 `TestActorRef` 提供的 `create` 方法，传入创建的 `Actor` 系统，以及指向 `Actor` 类的引用。由于真正的 `Actor` 实例被隐藏了，因此在我们的 `Actor` 系统中创建 `Actor` 返回的是一个 `ActorRef`（在本例中是 `TestActorRef`），我们可以将消息发送至该 `ActorRef`。有了 `Actor` 系统和 `Actor` 的类引用后，Akka 就足以在 `Actor` 系统的创建这个简单的 `Actor`。因此，现在我们已经成功地创建了第一个 `Actor` 了。与 `Actor` 之间的交互是通过消息传递来进行的。我们使用“`tell`”将消息放入 `Actor` 的邮箱中。我们通过 `tell` 的第二个参数定义该消息并不需要任何响应对象。

```
actorRef.tell(new SetRequest("key", "value"), ActorRef.noSender());
```

因为我们使用的是 `TestActorRef`，所以只有在 `tell` 调用请求处理完成后，才会继续执行后面的代码。对于我们的第一个 `Actor` 来说，这并没有问题，但是要注意的是，这个例子并没有展示出 `Actor API` 的异步特性，而这并不是常见的用法。通常情况下，`tell` 是一个异步操作，调用后会立即返回。最后，我们需要检查 `Actor` 是否将值存入了 `map` 中，确认其行为是否正确。为了进行这项检查，首先得到指向背后 `Actor` 实例的引用，然后调用 `get("key")` 检查 `map`，确认已经将值存入 `map` 中。

```
RequestReceiver requestReceiver = actorRef.underlyingActor();
Assert.assertEquals(RequestReceiver.map.get("key"), "value");
```

这样我们就完成了第一个简单的测试用例。这个基本的模式可以用于构建同步的 `Actor` 单元测试。

1.2.3 Actor 创建详述

访问 Actor 的方法和访问普通对象的方法有所不同。我们从来都不会得到 Actor 的实例，从不调用 Actor 的方法，也不直接改变 Actor 的状态，反之，只向 Actor 发送消息。除此之外，我们也不会直接访问 Actor 的成员，而是通过消息传递来请求获取关于 Actor 状态的信息。使用消息传递代替直接方法调用可以加强封装性。Alan Kay 是最早对面向对象编程进行描述的人，他实际上把消息传递也定义为面向对象编程的一部分。我想 Alan Kay 看到面向对象后来的发展情况，有时候一定会唏嘘不已。

我发明了“面向对象”这个术语，不过我可以告诉你，这跟 C++ 一点关系都没有。
——Alan Kay, OOPSLA '97

通过使用基于消息的方法，我们可以相当完整地将 Actor 的实例封装起来。如果只通过消息进行相互通信的话，那么永远都不会需要获取 Actor 的实例。我们只需要一种机制来支持向 Actor 发送消息并接收响应。

在 Akka 中，这个指向 Actor 实例的引用叫做 ActorRef。ActorRef 是一个无类型的引用，将其指向的 Actor 封装起来，提供了更高层的抽象，并且给用户提供了一种与 Actor 进行通信的机制。上文已经介绍过，Actor 系统就是包含所有 Actor 的地方。有一点可能相当明显：我们也正是在 Actor 系统中创建新的 Actor 并获取指向 Actor 的引用。actorOf 方法会生成一个新的 Actor，并返回指向该 Actor 的引用。

```
ActorSystem system = ActorSystem.create();
ActorRef actor = system.actorOf(Props.create(RequestReceiver.class));
```

Actor 都被封装起来了，不能够被直接访问。我们不能从外部代码中直接访问 Actor 的状态。创建 Actor 的模式保证了这一点，现在我们就来一窥究竟。

Props

为了保证能够将 Actor 的实例封装起来，不让其被外部直接访问，我们将所有构造函数的参数传给一个 Props 的实例。Props 允许我们传入 Actor 的类型以及一个变长的参数列表。

```
Props.create(RequestReceiver.class, arg1, arg2, argn);
```

actorOf 创建一个 Actor，并返回指向该 Actor 的引用 ActorRef。除此之外，还有另一种方法可以获取指向 Actor 的引用：actorSelection。为了理解 actorSelection，我们需要先来看一下 Actor 的路径。每个 Actor 在创建时都会有一个路径，我们可以通过 ActorRef.path 来查看该路径。该路径看上去如下所示：

- akka://default/user/BruceWillis
该路径是一个 URI，它甚至可以指向使用 akka.tcp 协议的远程 Actor。
- akka.tcp://my-sys@remotehost:5678/user/CharlieChaplin
要注意的是，路径的前缀说明使用的协议是 akka.tcp，并且指定了远程 Actor 系统的主机名和端口号。如果知道 Actor 的路径，就可以使用 actorSelection 来获取指向该 Actor 引用的 ActorSelection（无论该 Actor 在本地还是远程）。

```
ActorSystem system = ActorSystem.create();
ActorSelection selection = system.actorSelection("akka.tcp://actorSystem@host.jason-goodwin.com:5678/user/KeannaReeves");
```

这里的 ActorSelection 是一个指向 Actor 的引用，我们可以像使用 ActorRef 一样，使用 ActorSelection 进行网络间的通信。由此可以看出，使用 Akka 在网络上传递消息非常容易，而这也是 Akka 的位置透明性在实际应用中的一个例子。要对某个应用程序进行修改，使之与远程服务进行通信，几乎只需要修改对 Actor 位置的配置就足够了。

1.2.4 Actor 支持远程

有的时候我们需要支持远程应用程序通过网络远程访问上面定义的 Actor。

编写服务器端

我们需要支持远程应用程序通过网络远程访问上面定义的 Actor。幸运的是，这是件很简单的事。我们只需要添加配置文件，就可以启用 Actor 的远程访问功能了。在 src/main/resources 文件夹中新建一个文件，命名为 application.conf，然后把下面的配置内容添加到该文件中，其中包含了要监听的主机和端口。Akka 负责解析 application.conf。这是一个 HOCON 文件，是一种类型安全的配置文件，格式和 JSON 类似，与其他配置文件格式一样，使用方便。Akka 文档中经常提到该格式的配置文件，笔者认为如果要配置多个属性，HOCON 是一种相当不错的用于配置文件的格式。要注意的是，如果将其命名为 application.properties 并且使用属性配置文件的格式（比如 keypath.key=value），Akka 也是可以解析的。下面是 application.conf 的内容：

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
  }
}
```

接下来我们只需要在服务端用上文讲述过的 Actor 构建方法构建出一个远程监听 Actor。

```
ActorSystem system = ActorSystem.create("actorSystem");
system.actorOf(Props.create(RequestReceiver.class), "receiver");
```

Akka 会输出日志，表明其正在监听远程连接，并且给出当前服务器的地址（我们马上会在客户端中使用该地址）。

```
Starting remoting
Remoting started; listening on addresses :[akka.tcp://actorSystem@127.0.0.1:2552]
Remoting now listens on addresses: [akka.tcp://actorSystem@127.0.0.1:2552]
```

编写客户端

这一小节，我们将构建客户端，连接远程 Actor。只需要在客户端运用上文提到过的 system.actorSelection 方法即可拿到远程 Actor 的引用，接下来对 Actor 的消息发送相信读者已经很熟悉了。

```
ActorSystem system = ActorSystem.create("localSystem");
ActorSelection remote;
remote = system.actorSelection("akka.tcp://actorSystem@127.0.0.1:2552/user/receiver");
remote.tell(new SetRequest("key", "value"), remote.anchor());
```

可以看出，用户自建的 Actor 会被 ActorSystem 分配在路径 /user 下，这是读者在对 Actor 地址进行操作的时候需要注意的地方。

在服务器端，相信下方的接收日志已经被打印出来。

```
Received Set request: SetRequest[key='key', value=value]
```

1.2.5 Actor 的 tell

Tell 是最简单的消息模式，不过要花上一些时间才能够学会这种模式的最佳实践。Tell 通常被看做是一种 “fire and forget” 消息传递机制，无需指定发送者。不过通过一些巧妙的方法，也可以使用 tell 来完成 “request/reply” 风格的消息传递，如图 1-4 所示。

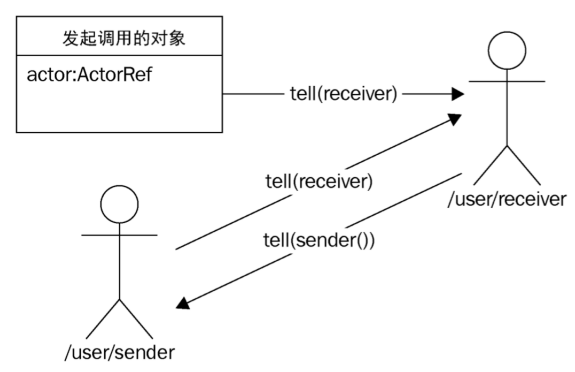


图1-4

Tell 是 ActorRef/ActorSelection 类的一个方法。它也可以接受一个响应地址作为参数，接收消息的 Actor 中的 sender() 其实就是这个响应地址。如果没有 sender（如在 Actor 外部发起请求），那么响应地址不会默认设置为任何邮箱（叫做 DeadLetters）。在 Java 中并没有隐式定义或是默认参数，所以必须提供 sender。如果不想指定任何特定的 sender 作为响应地址，那么应该遵循下述习惯。

- 从一个 Actor 内部发送消息，使用 self() : actor.tell(message, self());
- 从 Actor 系统外部发送消息，使用 noSender : actor.tell(message, ActorRef.noSender());

如果使用 tell，那么这样的做法是意料之中的。如果从 Actor 内部发送消息，那么响应地址的引用应该就是发送消息的 Actor，否则就应该没有响应地址。不过，在使用 tell 时也可以指定其他 sender，这一点相当重要：实际上我们可以通过一些颇具创造性的响应地址用法将状态存储在 Actor 中，以此减少请求的数量。

使用 Tell 处理响应

由于在返回消息时可以访问到指向发送者的引用，所以要对某条消息做出响应是很容易的。不过，在处理响应的时候，我们需要知道 Actor 收到的是哪一条消息的响应。如果我们在 Actor 中存储一些状态，记录 Actor 希望收到响应的消息，那么就能够高效地向 Actor 发送请求。

下面举一个非常简单的使用 tell 进行设计的例子。我们可以在 Actor 中将一些上下文信息存储在一个 map 中，将 map 的 key 放在消息中一起发送。然后，当有着相同 key 的消息返回时，就可以恢复上下文，完成消息的处理了，如图 1-5 所示。

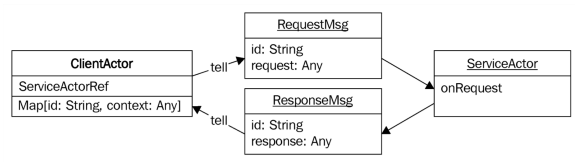


图1-5

1.2.6 Actor 的监督策略

Erlang 将容错性引入了 Actor 模型，它使用的概念叫做监督（supervision）。监督的核心思想就是把对于失败的响应和可能引起失败的组件分隔开，并且把可能发生错误的组件通过层级结构来组织，以便管理。

监督的层级结构

Akka 使用 Actor 层级结构来描述监督。当我们创建 Actor 时，新建的 Actor 都是作为另一个 Actor 的子 Actor，父 Actor 负责监督子 Actor。Actor 的路径结构就展示了它的层级结构，所以和文件系统中的文件夹有点像。

位于 Actor 层级结构顶部的是路径为/的根 Actor。然后是路径为/user 的守护 Actor。使用 actorSystem.actorOf() 函数创建的所有 Actor 都是守护 Actor 的子 Actor（/user/yourActor）。

如果在一个 Actor 内部创建另一个 Actor，那么可以通过调用 context().actorOf 使得新建的 Actor 成为原 Actor 的子 Actor。这个新建的 Actor 就成为了 Actor 树结构中的下一个叶节点（/user/yourActor/newChild）。

根 Actor 下面还有其他 Actor 层级结构。系统操作相关的 Actor 在路径为/system 的系统守护 Actor 下面。路径 /temp 下面还包含了一个临时的 Actor 层级结构。我们并不需要对这些细节担心过多，这些基本上都是 Akka 的内部实现，对于开发者是不可见的。

还记得寿司餐厅的例子吗？假设这个餐厅老板负责管理一个经理，而经理直接管理两个员工（寿司师和服务员）。那么我们就可能会有如下所示的层级结构，如图 1-6 所示。

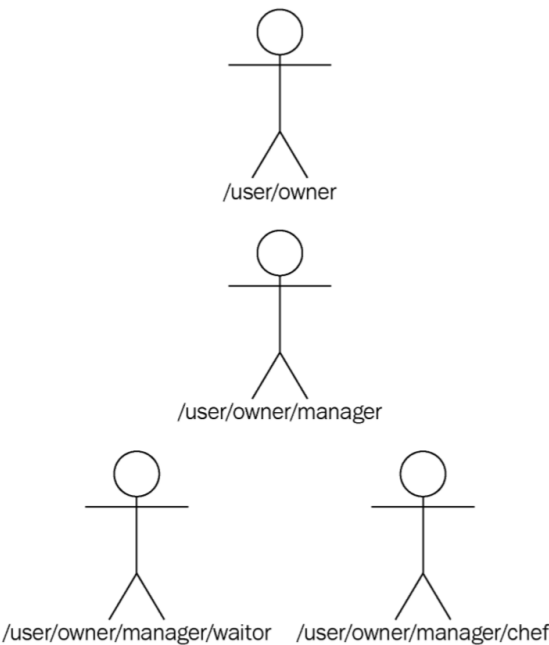


图1-6

每个负责监督其他 Actor 的 Actor 都可以自行定义监督策略。下面我们就来介绍监督策略。

监督策略和醉酒的寿司师

我们将通过一个例子来帮助理解监督策略。假设寿司师很喜欢喝酒。他喝了酒以后经常给自己惹麻烦，所以此时他的经理必需要采取一些措施。经理可以选择一些不同的做法。下面我们逐一介绍。

- 继续 (resume) : Actor 继续处理下一条消息。
- 停止 (stop) : 停止 Actor , 不再做任何操作。
- 重启 (restart) : 新建一个 Actor , 代替原来的 Actor 。
- 向上反映 (escalate) : 将异常信息传递给下一个监督者。

为了帮助我们理解上面的概念,假设表示寿司师的 Actor 必需要制作寿司。他非常熟练,每做完一盘寿司,都要喝一杯酒来庆祝一下自己的杰作。服务员把客人点的菜单叠起来交给寿司师,寿司师就不断地制作寿司并喝酒。如果寿司师从早忙到晚的话,最后就无法处在最佳工作状态了(说得轻一点)。

如果寿司师犯了一个错误(比如说切到了自己的手指或是掉了一个盘子),那么这可能是可以接受的。监督者将告诉寿司师在这种情况下 resume()。

如果寿司师累了,犯下了错误,这说明他可能需要休息一下。这时候经理就可能让寿司师 Actor 停止处理消息,休息一下。通常都由监督者来决定在这种情况下采取哪种必要的行为,比如说,监督者也需要告诉他监督的 Actor 何时重新继续开始工作。

一旦寿司师喝醉了,做了一盘巨丑无比的寿司,开始让一些客人不满了,那么经理(寿司师的监督者)就要负责对这种异常情况进行响应了。经理要说明寿司师喝醉了,并且把他辞退了。监督者请来另一个寿司师,接着根据下一个要做的菜单制作寿司。送走原来的寿司师,引进一个新寿司师的过程就相当于重启。

新来的寿司师比较年轻,忍不住也喝了起来。他彻底喝醉了,还打翻了一支蜡烛,餐厅着火了。经理处理不了这个错误——单单辞退喝醉的寿司师,再新请一个师傅可灭不了火!经理给他的监督者(餐厅老板)打电话,告诉他需要关门并且马上报警。这就是向上反映,此时经理的上级(饭店老板)就需要做决定。如果把异常向上反映给了守护 Actor,那么就会关闭应用程序。

定义监督策略

Actor 有默认的监督策略。如果没有修改监督策略,那么监督 Actor 的行为基本上和 喝醉的寿司师例子中一样:

- Actor 运行过程中抛出异常: restart()。
- Actor 运行过程中发生错误: escalate()。
- Actor 初始化过程中发生异常: stop()。

在默认监督策略中还定义了另一种情况: ActorKilledException。如果 Actor 被“杀”(kill)了,那么这个 Actor 的监督者会接收到一个 ActorKilledException,执行 stop()会接收到该异常。

还记得 /system 这个路径吗?所有的监督事件实际上都发生在这个路径下面的 Actor 中,而且异常事件并不是普通的消息。如果想要试着推导消息传递的顺序,那么理解这一点是十分重要的。

我们来看一下如果想要自己描述经理的监督行为,应该如何定义监督策略。

```
@Override
public akka.actor.SupervisorStrategy supervisorStrategy() {
    return new OneForOneStrategy(5, Duration.create("1 minute"),
        akka.japi.pf.DeciderBuilder
            .match(BrokenPlateException.class, e -> SupervisorStrategy.resume())
            .match(DrunkenFoolException.class, e -> SupervisorStrategy.restart())
            .match(RestaurantFireError.class, e -> SupervisorStrategy.escalate())
            .match(TiredChefException.class, e -> SupervisorStrategy.stop())
    );
}
```

```

        .matchAny(e -> SupervisorStrategy.escalate())
        .build());
    }
}

```

我们在 Actor 中重写了 supervisorStrategy 方法，该方法返回一个新的策略（OneForOneStrategy，我们马上会详细介绍）。在 Java 8 中，我们使用 DeciderBuilder 来创建一个 PartialFunction，用于表示策略，类似于在 receive 块中用来构造 PartialFunction 的 ReceiverBuilder。我们分别用一个 match 语句描述了寿司餐厅例子中的一种场景，最后还加入了一个 matchAny 语句，表示收到任何未知异常时，经理就给老板打电话。

我们都描述了当 Actor 抛出不同的 Throwable 时该采取什么样的处理方法：如果寿司师打破了盘子，就让他继续使用下一个盘子；如果寿司师喝醉了，就辞退他，再请一个新的寿司师；如果寿司师把餐厅弄着火了，就要让经理打电话给他老板；如果寿司师累了，就让他休息一下。

要注意的是，一般情况下使用默认的行为就可以了：如果 Actor 在运行中抛出异常，就重启 Actor；如果发生错误，就向上反映或是关闭应用程序。不过如果 Actor 在构造函数中抛出异常，那么会导致 ActorInitializationException，并最终导致 Actor 停止运行。因为在这种情况下应用程序不会继续运行，所以要对此特别注意。

1.3 Akka Actor 生命周期

1.3.1 Actor 生命周期

我们介绍了重启的概念，就好像把一个工人辞退，再雇一个新工人来代替他。现在我们就来学习 Actor 的生命周期中都发生了什么。

在 Actor 的生命周期中会调用几个方法，我们在需要时可以重写这些方法。

- prestart()：在构造函数之后调用。
- postStop()：在重启之前调用。
- preRestart(reason, message)：默认情况下会调用 postStop()。
- postRestart()：默认情况下会调用 preStart()。

在 Actor 生命周期中个事件的发生顺序，如图 1-7 所示。

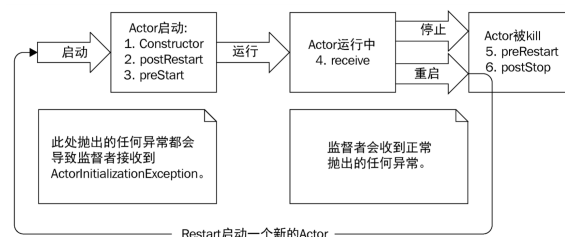


图1-7

这一点对于设计是很有帮助的。假设有一个聊天应用程序，每个用户都用一个 Actor 来表示，而 Actor 之间的消息传递就表示了用户之间的聊天。当一个用户进入聊天室的时候，就启动了一个 Actor，并发送一条消息，更新聊天室中的用户名单。当一个 Actor 停止的时候，发送另一条消息，将该用户从聊天室中显示的当前活跃名单中删除。如果使用默认实现，那么要是 Actor 遇到了异常并且重启，那么就会先将该用户从聊天窗口中删除，再将其添加进去。我们可以重写 preRestart 和 postRestart 方法，这样用户就只会在真正加入或离开聊天室的时候才被添加或删除到名单中，如图 1-8 所示。

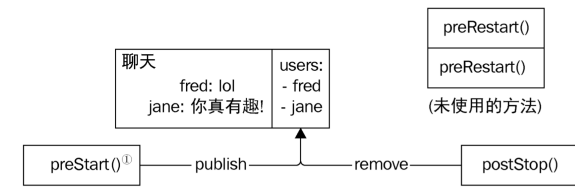


图1-8

1.3.2 重启和停止时的消息处理

我们需要理解发生异常时的消息处理方式，并且根据应用程序采取合适的操作。这一点十分重要。我们可以定义监督策略，在抛出异常前重新尝试发送失败的消息，重试次数没有限制。也可以设置时间限制，比如最多重试 10 次或 1 分钟，只要达到其中任一限制就停止重试：

```
new OneForOneStrategy(2, Duration.create("1 minute"), t -> {  
    if (t instanceof IOException) {  
        return SupervisorStrategy.restart();  
    } else {  
        return SupervisorStrategy.stop();  
    }  
});
```

一旦某个消息抛出了异常，该消息就会被丢弃，不会被重新发送。监督者会执行监督策略。在继续执行（resume）或是重启（restart）的情况下，就会处理下一条消息了。如果有一个工作队列的话，我们可能会想要重试若干次，然后彻底返回失败。

1.3.3 终止或 kill 一个 Actor

有多种不同的方法可以用来停止一个 Actor。下面任一方法都可以停止 Actor：

- 调用 `ActorSystem.stop(actorRef)`。
- 调用 `ActorContext.stop(actorRef)`。
- 给 Actor 发送一条 `PoisonPill` 消息，会在 Actor 完成消息处理后将其停止。
- 给 Actor 发送一条 `kill` 消息，会导致 Actor 抛出 `ActorKilledException` 异常。

调用 `context.stop` 或 `system.stop` 会导致 Actor 立即停止。发送 `PoisonPill` 消息则会在 Actor 处理完消息后将其停止。不同的是，`kill` 不会马上直接停止 Actor，而是会导致 Actor 抛出一个 `ActorKilledException`，这个异常由监督者的行为来处理——我们可以决定接收到这个异常时执行什么操作。

1.3.4 生命周期监控和 DeathWatch

监督机制描述了对子 Actor 的状态进行响应。而 Actor 也可以对其他任何 Actor 进行监督。通过调用 `context.watch(actorRef)` 注册后，Actor 就能够监控另一个 Actor 的终止，而调用 `context.unwatch(actorRef)` 就可以取消监控注册。如果被监控的 Actor 停止了，负责监控的 Actor 就会收到一条 `Terminated(ActorRef)` 消息。

1.3.5 安全重启

构建响应式系统时，要把失败处理列入设计考虑要素之中。我们要确保系统能够对失败情况做出正确的响应。笔者在实际项目中见到过的一个错误就是，启动 Actor 的时候没有处理失败情况，启动之后就对 Actor 的状态进行初始化。如果向 Actor 发送一些初始化消息，而消息中的信息对于 Actor 的状态至关重要，那么这些信息有可能会在重启的过程中丢失，一定要格外小心。

我们来看一个例子：假设表示数据库客户端的 Actor 没有构造参数，我们调用 tell，传入数据库的主机和端口来连接数据库。这看上去没有任何不合理之处——在 Java 中，我们经常会先创建对象，然后设置成员变量的值。我们也知道 Actor 能够安全地保存状态，所以这种用法似乎并没有什么问题：

```
ActorRef db = system.actorOf(DbActor.props());
actor.tell(new Connect("10.0.8.14", "9001");
```

现在我们就有了一个 Actor，一旦这个 Actor 处理了第一条消息，就完成初始化，连接到了远程数据库。

假设我们先向这个 Actor 发送消息，然后由于网络可靠性的问题，Actor 发生了一个异常。此时 Actor 会重新启动。当 Actor 重启的时候，会调用 preRestart()，然后原来的 Actor 就会停止。这个时候原来 Actor 内的所有状态就消失了。接着会新建一个 Actor 实例，运行新建的 Actor 的构造函数，然后调用 postRestart()，Actor 就启动完成并运行了。

现在，用于连接数据库的消息就丢失了。我们可以编写一些额外的代码来处理这种情况。可以编写一个监督策略，也可以使用 Akka 生命周期的监控功能（DeathWatch）来对这些失败情况做出响应。但是问题在于，我们只是为了这个简单的初始化消息，就必须立刻在客户端中处理错误情况，而且有可能会在处理的过程中犯下一些错误。更好的实现方式是通过 Props 把初始化信息传递给 Actor 的构造函数：

```
ActorRef db = system.actorOf(DbActor.props(host = "10.0.8.14", port = "9001"));
```

这样看上去更好：Actor 现在可以从构造函数中获取所有信息，并且自动连接。如果我们开始向 Actor 发送消息，然后 Actor 重启的话，由于构造函数的参数中包含了连接信息，因此 Actor 能够正确地从错误中恢复。不过如果 Actor 在初始化的过程中发生了异常（比如说在尝试连接远程数据库的过程中），那么默认的监督机制会停止 Actor。所以，有一种情况是：如果 Actor 已经处在运行中，那么就能够安全地重新启动；但是如果无法完成初始化，我们就必须编写额外的代码实现监督策略，来处理 Actor 无法在启动时连接到数据库的情况。

很多时候，我们不希望 Actor 在初始化的过程中发生错误，而是会给 Actor 发送一条初始化消息，然后在 Actor 运行的过程中处理状态的变化。可以在 preStart() 中向 Actor 自己发送一条 Connect 消息来达到这一效果：

```
public void preStart() {
    self().tell(new Connect(), null);
}
```

这样一来，Actor 只在运行中才可能会出现错误，而且能够成功重启。所以就可以不断尝试连接数据库直到连上为止了。

仔细想一想，读者可能会觉得应该在 preRestart 方法中给自己发送 Connect(host, port) 消息。如果 Actor 出于任意原因丢弃了这条消息（可能是因为邮箱已满，又或者是由于重启导致了消息丢失），那么这条 Connect 消息还是可能会丢失。对于重要的 Actor 功能，使用构造函数来初始化更简单，也更安全。

上面只是快速而粗略地介绍了如何构造可能不断重新请求连接的 Actor。

有一个很重要的地方需要注意：上面的介绍中还没有涉及到一些细节和问题。首先，我们没有处理 Actor 在连接成功之前接收到的消息；其次，如果 Actor 长时间无法连接的话，邮箱可能会被填满。

1.4 Akka Actor Router

1.4.1 Router 介绍

在 Akka 中，Router 是一个用于负载均衡和路由的抽象。创建 Router 时，必须要传入一个 Actor Group，或者由 Router 来创建一个 Actor Pool。

注意 Group 和 Pool 这两个词的用法。为 Actor 创建 Router 时，一定要理解，有两种用来创建 Router 背后的 Actor 集合的机制。一种是由 Router 来创建这些 Actor（一个 Pool），另一种是把一个 Actor 列表（Group）传递给 Router。

在创建了 Router 之后，当 Router 接收到消息时，就会将消息传递给 Group/Pool 中的一个或多个 Actor。有多种策略可以用来决定 Router 选择下一个消息发送对象的顺序。

在我们的例子中，所有的 Actor 都运行在本地，我们需要一个包含多个 Actor 的 Router 来支持使用多个 CPU 核进行并行运算。如果 Actor 运行在远程机器上，也可以使用 Router 在服务器集群上分发工作，**如图 1-9 所示**。

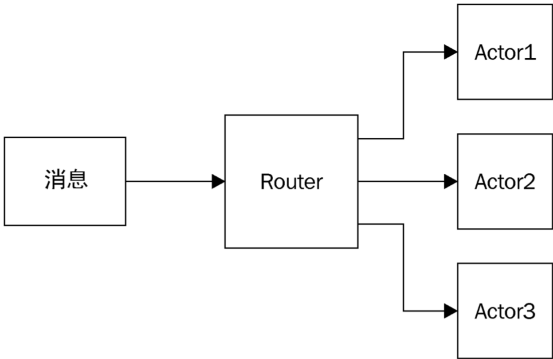


图1-9

在我们使用本地 Actor 的例子中，可以选择 Actor Pool 的方式来创建 Router，由 Router 来创建我们所需的所有 Actor。在这种情况下使用 Router 非常简单：照常实例化一个 Actor，然后调用 withRouter，并传入一个路由策略，以及希望 Pool 中包含的 Actor 数量：

```
ActorRef workerRouter = system.actorOf(Props.create(RequestReceiver.class).withRouter(new RoundRobinPool(8)));
```

也可以采用 Actor Group 的方式来创建 Router，传入一个包含 Actor 路径的列表：

```
ActorRef router = system.actorOf(new RoundRobinGroup(actors.map(actor -> actor.path()).props()));
```

现在，就有了将工作分发给不同 CPU 核所需的 Router 和 Actor。可以请求 Router 来处理列表中的每个消息，这样就可以并行执行操作了。我们将继续介绍 Router 的一些更高级的特性，深入理解 Router 的工作原理。

1.4.2 路由逻辑

注意到我们使用了 RoundRobinPool/RoundRobinGroup，用于指定 Router 将消息发送给各 Actor 的顺序。Akka 内置了一些路由策略，我们在这里将介绍一部分，**如图 1-10 所示**。对于一般情况来说，RoundRobin 和 Random 都是不错的选择。

路由策略	功能
Round Robin	依次向 Pool/Group 中的各个节点发送消息，循环往复。Random——随机向各个节点发送消息。
Smallest Mailbox	向当前包含消息数量最少的 Actor 发送消息。由于远程 Actor 的邮箱大小未知，因此假设它们的队列中已经有消息在排队。所以会优先将消息发送给空闲的本地 Actor。
Scatter Gather	向 Group/Pool 中的所有 Actor 都发送消息，使用接收到的第一个响应，丢弃之后收到的任何其他响应。如果需要确保能够尽快收到一个响应，那么可以使用 scatter/gather。
Tail Chopping	和 Scatter/Gather 类似，但是 Router 并不是一次性向 Group/Pool 中的所有 Actor 都发送一条消息，而是每向一个 Actor 发送消息后等待一小段时间。有着和 Scatter/Gather 类似的优点，但是相较而言有可能可以减少网络负载。
Consistent Hashing	给 Router 提供一个 key，Router 根据这个 key 生成哈希值。使用这个哈希值来决定给哪个节点发送数据。想要将特定的数据发送到特定的目标位置时，就可以使用哈希。在下章中，我们将讨论更多有关一致性哈希的问题。
BalancingPool	BalancingPool 这个路由策略有点特殊。只可以用于本地 Actor。多个 Actor 共享同一个邮箱，一有空闲就处理邮箱中的任务。这种策略可以确保所有 Actor 都处于繁忙状态。对于本地集群来说，经常会优先选择这个路由策略。

图1-10

我们也可以实现自己的路由策略，不过大多数情况下并不需要这么做。

1.4.3 向 Router 中的所有 Actor 发送消息

无论是使用 Group 还是 Pool 的形式来创建 Router，都可以通过广播，将一条消息发送给所有 Actor。例如，如果 Actor 都连接到一个远程数据库，运行中的系统由于发生了错误需要修改使用的数据库，那么就可以通过一条广播消息更新 Pool/Group 中的所有 Actor：

```
router.tell(new akka.routing.Broadcast(msg));
```

1.4.4 监督 Router Pool 中的路由对象

如果使用 Pool 的方式创建 Router，由 Router 负责创建 Actor，那么这些路由对象会成为 Router 的子节点。创建 Router 时，可以给 Router 提供一个自定义的监督策略。

创建 Router 的时候，可以调用 withSupervisorStrategy 方法指定 Router 对 Pool 中路由对象的监督策略。假设我们有一个叫做 strategy 的 SupervisorStrategy，那么可以使用下面的代码来创建 Router：

```
ActorRef router = system.actorOf(Props.create(RequestReceiver.class).withRouter(new RoundRobinPool(8).withSupervisorStrategy(strategy)));
```

由于使用 Group 方法创建 Router 的时候传入了事先已经存在的 Actor，所以没有办法用 Router 来监督 Group 中的 Actor。

监督 Pool 中的 Actor 是给 Router 指定监督策略的最常见的一种使用场景。除此之外，还有另一个场景会用到这种做法。如果有一个顶层的 Actor（使用 ActorSystem 的 actorOf 方法创建），那么这个 Actor 会将由守护 Actor 来监督。如果需要为这个 Actor 指定一个自定义的监督策略，那么一种方法就是创建另一个 Actor 来负责监督。除此之外，我们可以直接创建一个 Router，然后传递一个自定义的 SupervisorStrategy，由 Router 负责监督 Actor。由于这种方法不需要定义任何 Actor 行为，所以是最简单的为顶层 Actor 提供自定义监督策略的方法。

1.5 Akka Actor Dispatcher

现在我们有必要理解系统的所有瓶颈所在以提高应用程序的吞吐量和响应时间，了解请求/响应循环中时间都花在了什么地方。一旦给应用程序发送了任务之后，可用的线程就会试图处理所有的请求：我们需要理解如何去使用这些资源，这可以帮助服务使用尽可能少的延迟来达到尽可能多的吞吐量。

1.5.1 Dispatcher 解析

Dispatcher 将如何执行任务与何时运行任务两者解耦。一般来说，Dispatcher 会包含一些线程，这些线程会负责调度并运行任务，比如处理 Actor 的消息以及线程中的 Future 事件。Dispatcher 是 Akka 能够支持响应式编程的关键，是负责完成任务的机制。

所有 Actor 或 Future 的工作都是由 Executor/Dispatcher 分配的资源来完成的，如图1-11所示。

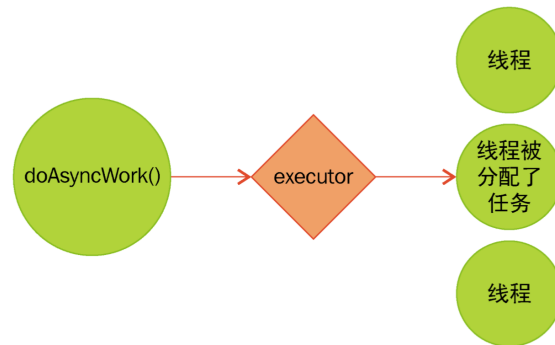


图1-11

Dispatcher 负责将工作分配给 Actor。除此之外 Dispatcher 还可以分配资源用于处理 Future 的回调函数。我们会发现 Future API 接受 Executor /ExecutionContext 作为参数。由于 Akka 的 Dispatcher 扩展了这些 API，因此 Dispatcher 具备双重功能。

在 Akka 中，Dispatcher 实现了 `scala.concurrent.ExecutionContextExecutor` 接口，而这个接口又扩展了 `java.util.concurrent.Executor` 和 `scala.concurrent.ExecutionContext`。可以将 Executor 传递给 Java 的 Future。

用于 Future 时，可以通过 ActorSystem 中的一个引用来获取 Dispatcher (`ActorSystem.dispatcher`)。可以在 ActorSystem 中通过 ID 查询得到配置文件中定义的某个 Dispatcher：

```
system.dispatchers.lookup("my-dispatcher");
```

由于我们能够创建并获取这些基于 Executor 的 Dispatcher，因此可以使用它们来定义 ThreadPool/ForkJoinPool 来隔离运行任务的环境。稍后讨论什么时候这么做以及为什么要这么做。虽然要高效地使用 Executor，其实并不需要理解所有的细节，但是我们还是会先做一些介绍。

1.5.2 Executor

Dispatcher 基于 Executor，所以在具体介绍 Dispatcher 之前，我们将介绍两种主要的 Executor 类型：ForkJoinPool 和 ThreadPool。

ThreadPool Executor 有一个工作队列，队列中包含了要分配给各线程的工作。线程空闲时就会从队列中认领工作。由于线程资源的创建和销毁开销很大，而 ThreadPool 允许线程的重用，所以就可以减少创建和销毁线程的次数，提高效率。

ForkJoinPool Executor 使用一种分治算法，递归地将任务分割成更小的子任务，然后把子任务分配给不同的线程运行。接着再把运行结果组合起来。由于提交的任务不一定都能够被递归地分割成 ForkJoinTask，所以 ForkJoinPool Executor 有一个工作窃取算法，允许空闲的线程“窃取”分配给另一个线程的工作。由于工作可能无法平均分配并完成，所以工作窃取算法能够更高效地利用硬件资源。

ForkJoinPool Executor 几乎总是比 ThreadPool 的 Executor 效率更高，是我们的默认选择。

1.5.3 创建 Dispatcher

要在 application.conf 中定义一个 Dispatcher，需要指定 Dispatcher 的类型和 Executor。还可以指定 Executor 的具体配置细节，比如使用线程的数量，或是每个 Actor 一次性处理的消息数量。

```
my-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    #Minimum threads
    parallelism-min = 2
    #Maximum total threads
    parallelism-max = 10
    #Maximum threads per core
    parallelism-factor = 2.0
    #Max messages to process in an actor before moving on.
    throughput = 100
  }
}
```

有四种类型的 Dispatcher 可以用于描述如何在 Actor 之间共享线程：

- Dispatcher：默认的 Dispatcher 类型。将会使用定义的 Executor，在 Actor 中处理消息。在大多数情况下，这种类型能够提供最好的性能。
- PinnedDispatcher：给每个 Actor 都分配自己独有的线程。这种类型的 Dispatcher 为每个 Actor 都创建一个 ThreadPool Executor，每个 Executor 中都包含一个线程。 如果希望确保每个 Actor 都能够立即响应，那么这似乎是个不错的方法。不过 PinnedDispatcher 比其他共享资源的方法效率更高的情况其实并不多。可以在单个 Actor 必须处理很多重要工作的时候试试这种类型的 Dispatcher，否则的话不推荐使用。
- CallingThreadDispatcher：这个 Dispatcher 比较特殊，它没有 Executor，而是在发起调用的线程上执行工作。这种 Dispatcher 主要用于测试，特别是调试。由于发起调用的线程负责完成工作，所以清楚地看到栈追踪信息，了解所执行方法的完整上下文。这对于理解异常是非常有用的。每个 Actor 会获取一个锁，所以每次只有一个线程可以在 Actor 中执行代码，而如果多个线程向一个 Actor 发送信息的话，就会导致除了拥有锁的线程之外的所有线程处于等待状态。本文前面介绍过的 TestActorRef 就是基于 CallingThreadDispatcher 实现支持在测试中同步执行工作的。
- BalancingDispatcher：我们会在一些 Akka 文档中看到 BalancingDispatcher。现在已经不推荐直接使用 BalancingDispatcher 了，应该使用前面介绍过的 BalancingPool Router。不过 Akka 中仍然使用了 BalancingDispatcher，但是只会通过 Router 间接使用。BalancingDispatcher 有一点很特殊：Pool 中的所有 Actor 都共享同一个邮箱，并且会为 Pool 中的每个 Actor 都创建一个线程。使用 BalancingDispatcher 的 Actor 从邮箱中拉取消息，所以只要有 Actor 处于空闲状态，就不会有任何 Actor 的工作队列中存在任务。这是工作窃取的一个变种，所有 Actor 都会从一个共享的邮箱中拉取任务。两者在性能上的优点也类似。

创建 Actor 的时候，可以给 Props 提供在 application.conf 中配置好的 Dispatcher 名称：

```
system.actorOf(Props[MyActor].withDispatcher("my-pinned-dispatcher"));
```

1.5.4 决定何时使用哪种 Dispatcher

现在已经介绍了如何创建 Dispatcher 和 Executor，但其实还不是很清楚要用它们来做什么。本小节的目的就是介绍如何最大化地利用硬件资源，所以现在我们就将开始学习如何使用 Dispatcher 来编写对用户响应更快、对可能发生的性能问题更游刃有余的应用程序。

现在有一个从网页中抽取、缓存并向用户返回文章内容体的例子。假设在这个例子中，需要从 RDBMS 获取用户的个人信息，而 JDBC 调用会阻塞线程。这个例子中线程的数量有限，而且会被阻塞，有助于强化解释本小节的内容。

进行纵向扩展的第一步是理解哪些情况的响应即时性最重要，以及对这些重要的请求做出响应时可能会发生资源竞争的地方。如果只使用默认的 Dispatcher，那么假如有 1000 个请求，其中 500 个会阻塞线程（如 JDBC），另外 500 个是非阻塞操作，那么我们不希望所有用于响应重要请求的线程都被阻塞操作占据，如图 1-12 所示。

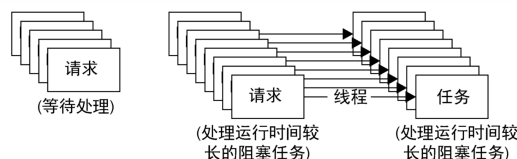


图1-12

在图 1-12 中，我们简化了例子的规模，只展示了 8 个运行长任务（比如阻塞 IO 操作或是大量的 CPU 处理）的线程。从图 1-12 中可以看出，由于所有可用的线程都已经被占据，所以其他请求都必须要等待。直到有线程资源被释放，才可以开始处理其他请求。而处于等待中的请求可能只需要进行很快速的缓存读取，但是却必须要排队等待空闲资源。

这就说明了只使用一个资源池来随意分配工作可能会导致应用程序的某些很耗资源的操作占尽了所有资源，而最重要的基本使用场景却无法得到资源。

要改善这种情况，可以把用于运行高风险任务的资源和运行重要任务的资源隔离开来。如果我们新建一些 Dispatcher，把运行时间较长或是会阻塞线程的任务都分配给这些 Dispatcher，就可以确保应用程序的剩余部分仍然能够保持响应的即时性。我们希望能够把所有需要大量计算、运行时间较长的任务分离到单独的 Dispatcher 中，确保在糟糕的情况下仍然能够有资源去运行其他任务。

这样一来，我们就可以把应用程序执行长任务时的延时也隔离开来。如果 MySQL 出了问题，花了 30 秒钟才返回响应，那么至少应用程序的其他部分还是能够迅速地做出响应。

要采用这种方法，就必需首先分析应用程序的性能，理解应用程序在什么地方可能会阻塞线程，耗尽系统资源。我们需要对应用程序执行的任务进行分类。

在我们的例子中，可以对应用程序执行的任务分类如下：

- 解析文章：运行时间较长的 CPU 密集型任务（10% 的请求）。
- 使用 JDBC 从 MySQL 读取用户信息：运行时间较长、会阻塞线程的 IO 操作（10% 的请求）。
- 获取文章：简单的非阻塞请求，从远程内存数据库中读取文章（80% 的请求）。

一旦我们知道了应用程序执行任务的类型，就可以开始理解这些任务是否可能引起性能问题——有没有可能会导致资源利用失控，影响应用程序的其他部分？分析阻塞线程和 CPU 密集型的任务，评估其是否可能导致应用程序其他重要部分无法获得资源：

- 解析文章：如果有人提交了一些发布在网上大型书籍，那么有可能所有线程都被占用，运行 CPU 密集型的长任务。这种情况风险中等，可以通过限制请求的大小来缓解。
- 使用 JDBC 读取用户信息：如果数据库需要 30 秒才能进行响应，那么所有线程可能都会处于等待状态。这种情况风险更高。
- 获取文章：读取文章的操作不会阻塞线程，不需要进行 CPU 密集型操作，风险较低。由于这种类型的操作占比最高，所以快速地对这类请求进行响应也很重要。

既然我们已经识别出了可能的高风险任务，那么当然希望使用单独的 Dispatcher 来负责这部分的工作，这样的话，发生问题的时候就不会对应用程序其他重要的部分产生负面影响。这是使用隔板法来隔离失败影响的另一个例子。不过在这里要提醒读者：对于变化可能对系统性能造成的影响，千万不要做任何假设，一定要实际测量。

我们已经事先对应用程序的工作和风险进行了分类，所以就可以把所有高风险任务分离出来，交给单独的 Dispatcher 来处理，如图 1-13 所示。

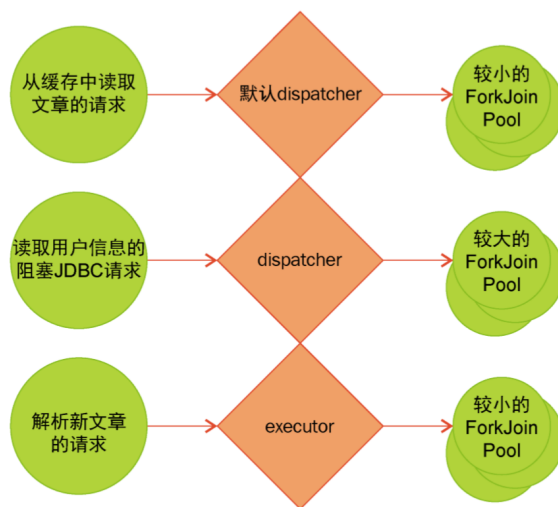


图1-13

对于解析文章请求，也由它自己的 Dispatcher 来处理，其中包含少量的线程。这里主要将 Dispatcher 用作隔离之用，以防用户提交的特别大的文章解析请求。如果提交的超大的请求，那么这个请求会导致队列中的请求都处于等待状态。所有为了处理这种极端情况，我们可以单独处理这类请求。在这种情况下，也可以使用 BalancingPool/BalancingDispatcher 将工作分配给 Pool 中所有用于解析文章的 Actor。除了能够提供隔离性以外，使用 BalancingPool 还有可能因为它原生的工作窃取特性改善资源利用效率。

1.5.5 默认 Dispatcher

有好多种使用默认 Dispatcher 的方法。既可以把所有工作都分离出去，只由 Akka 本身来使用默认 Dispatcher，也可以只在默认 Dispatcher 中执行异步操作，把高风险操作移到其他地方执行。无论怎么选择，都不能够阻塞运行默认 Dispatcher 的线程，而且对于运行在默认 Dispatcher 中的任务多加小心，防止资源饥饿情况的发生。

要创建或使用默认 Dispatcher/ThreadPool 的话，其实不需要做什么。如果需要的话，只要在 classpath 内的 application.conf 文件中定义并配置默认 Dispatcher 即可。如下所示：

```
akka {
  actor {
    default-dispatcher {
      # Min number of threads to cap factor-based parallelism number to
      parallelism-min = 8
      # 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 = 3.0
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 10
    }
  }
}
```

```
}  
}  
}
```

默认情况下，Actor 完成的所有工作都会在这个 Dispatcher 中执行。如果需要回去 ExecutionContext 并在其中创建 Future，那么可以通过 ActorSystem 访问到默认的线程池，然后将其传递给 Future：

```
ActorSystem system = ActorSystem.create();  
CompletableFuture.runAsync(() -> System.out.println("run in ec"), system.dispatcher());
```

1.5.6 并行最优化

要确定硬件上的最优并行度，方法只有一种：测试。如果没有真正进行测试以及调整，那么关于时间花在什么地方以及改变如何影响系统的所有猜想基本上都是错的。

如果使用的线程数量远远多于处理器核的个数，那么也会降低性能。所以不要觉得线程越多越好，选择包含很多 Actor 的 Pool。Akka 需要在 Actor 之间来回切换，而 Executor 也需要在线程之间平衡负载。操作系统需要进行调度，将 CPU 时间分配给运行中的线程，并通过交换运行线程与等待线程的状态进行上下文切换。也正是由于这些开销，最优的并行处理能够达到的处理时间实际上可能比我们想象的要长。要知道某个改变带来的影响，唯一的办法就是测试！

1.6 Akka Actor 集群化

在上一章中，我们介绍了如何通过并行操作来提高单个物理主机上的硬件利用率。还介绍了如何在单台主机上通过多个特定的 Dispatcher 来隔离性能问题。本章将介绍单台物理主机达到资源上限时的情况以及如何使用多台机器来处理工作。这听起来复杂得多，但是 Akka 提供的工具使得我们可以快速上手。

在本章中，我们将讨论下面的话题：

- 分布式系统中使用的一些基础概念。
- Akka Cluster 的介绍。
- 使用 Akka Cluster 来构建分布式系统。

1.6.1 Akka Cluster 介绍

本文已简要介绍过 Akka Remoting。理解 Remoting 对于学习 Akka 在网络上的通信方式很有帮助，不过在本章中，我们将使用 Akka Cluster 机制，而不是简单的点对点通信来提高部署系统的容错性和灵活性。本小节介绍一些集群的概念，以及 Akka Cluster 如何管理节点，支持应用程序按需横向扩展。

首先，我们来定义什么是集群。维基百科上的定义是：“计算机集群由一系列间接或 直接连接的计算机组成，这些计算机共同完成工作，从很多角度上来说，可以将这些计算机看作是单个系统。”具体到本章中对于集群的定义，我们可以认为一个集群就是一组机器（也可能是虚拟机），我们将这些机器称作节点（node）或是成员（member），同一个组中的所有节点遵循某个协议。

1.6.2 巨型单体应用 vs 微服务

如果我们在谈论微服务和分布式系统时，Martin Fowler 对相关问题的思考还是很具有参考价值，读者可以参读一下他的网站 <http://www.martinfowler.com/> 上的一些相关文章。

第一个有价值的观点是：在一个应用程序的生命周期中，当应用程序的复杂度还比较低的时候，和试图构建网络服务比起来，由一个小团队来构建单个大一点的应用程序（单体应用）可能效率更高。一开始可以先编写单个应用程序。随着应用程序变得越来越复杂，由于团队之间需要越来越多的协作来合并代码以及共同开发功能，生产效率会下降。这个时候，就可以开始看到使用小型网络服务给生产效率带来的好处。

第二个有价值的观点是：当我们需要将服务切分成许多更小的应用时，首先编写单体应用程序可以帮助我们理解应该将哪些部分分割出来。Fowler 认为如果没有真正开发系统一段时间的话，很有可能会选择把错误的服务实现成单独的应用。

一旦有了些在生产环境中运行系统的经验和数据，要把哪个部分分离出来才能在新增功能时带来性能和团队开发速度的好处就会比较清楚了。开始部署微服务之后，我们就可以用一种非平衡的方式对负载更高的服务进行扩展，给应用程序中使用更多的部分分配更大的服务器集群。

幸运的是，使用 Akka 时，更多时候只需要通过配置选项而不是代码就可以做这些部署决定。而使用多个代码库和让一个大团队使用同一个代码库比起来，也要简单得多。使用多个代码库解决了布鲁克斯定律（Brooks Law，给一个已经延期的项目添加资源会使其延期的时间更长）提到的问题，原因在于它减少了通信渠道，把代码提交分隔到不同的代码库中（不会再有合并代码带来的噩梦）。

而当单体应用变得极度复杂，难以管理，除非是有很好的基于数据支持的原因（性能或团队效率相关），否则最终我们很可能会考虑选择构建一个分布式系统，而不采取单体应用的方式来部署。

无论是选择构建单体应用还是微服务，我们可能都想从一开始就把不同的功能模块放在不同的库中。即使还没有真正准备部署多个独立的应用程序，让不同的团队使用不同的代码库也是有益的。一旦想要分割应用程序，如果一开始就使用多个代码库，那么管理起共享代码来要简单得多。

1.6.3 集群的定义

集群就是一组可以互相通信的服务器。集群中的每台服务器成为一个节点或成员。集群可以动态修改大小，并且在发生错误情况时继续运行，把影响降到最低。所以集群需要具备两个功能：失败发现以及使得集群中的所有成员最终能够提供一致的视图。

1.6.4 集群的失败检测

随着我们向集群中添加节点，节点很可能会发生错误，某些网络分区也可能暂时不可访问。因此集群是一个动态的实体，在服务器宕机或是不可用时会变小，而在添加服务器之后集群会变大（比如处理更高的负载）。集群中的节点通过互相发送消息来确定对方是否可用。节点基于能否获得其他节点的响应来确定其他节点是否可用。

如果集群中的每台服务器都需要和其他所有服务器进行通信，那么集群的性能不会随着节点的增加而线性提高。原因在于每增加一个节点，需要的通信开销都会指数增加。为了降低监控其他节点健康程度的复杂度，Akka 中的失败检测只会监控某个节点附近特定数目的节点。例如，在一个由 6 个节点环形排列组成的节点中，每个节点只会监控它后面的 2 个节点是否发生错误。在 Akka Cluster 中，每个节点监控的最大节点数默认是 5 个节点，如图 1-14 所示。

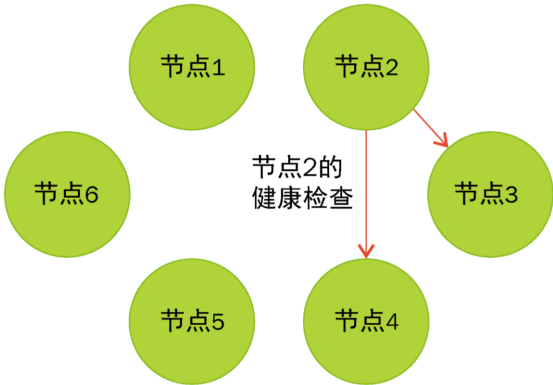


图1-14

在 Akka 中，失败检测是通过在节点间发送心跳消息并接收响应来完成的。Akka 会根据心跳的历史记录以及当前的心跳信息计算出某个节点可用的可能性。Akka 会依照这些数据和配置的容错限制得到计算结果，然后将节点标记为可用或不可用。

1.6.5 通过 gossip 协议达到最终一致性

思考一下失败检测的处理，集群中的每个节点和它的邻居节点互相交换已知的状态。接着，这些邻居节点再将已知的状态传递给它们的“邻居”，以此往复，直到某个节点的已知状态传递给了集群中的所有成员。节点会仔细计算并得到集群中其他成员的状态。

这种最终把状态传递给整个集群的机制叫做 gossip 协议或是 epidemic 协议（因为信息在集群中的传播就像病毒一样！）。许多最终一致性的数据库（比如 Riak 和 Cassandra）的实现方式都非常类似。在这些数据库中，Amazon 发表的关于 Dynamo 的论文影响力非常大。

我们暂时不需要知道太多 Akka Cluster 的内部原理，只需要知道 Akka 会负责确定集群中是否有状态变化，并且负责将发生的任何变化传递给集群中的所有节点。

本章剩余的章节会构建自己的集群。在这个过程中，我们将介绍许多集群的工作原理细节。如果读者想要了解更多关于 Akka Cluster 工作原理的背景知识，在 Akka 的文档中有一个叫做 Cluster Specification 的文档，可以去查阅。

1.6.6 使用 Akka Cluster 构建系统

我们将在本小节中介绍如何使用 Akka Cluster 分布到多台服务器上处理。假设我们已经构建了一个应用程序，这个程序需要能够在多个节点上扩展，并且具备高可用性，我们将学习到如何通过添加更多的节点来扩展我们的系统。

读者可能无法一口气就编写出 Cassandra，不过我们将在本章中编写出一个能够横向扩展的分布式服务。也会顺便介绍在分布式解决方案中的一些常用技术。

1.6.7 创建集群

本文前面简要介绍过 Remoting，现在我们就将更详细地学习集群的知识。Akka Cluster 基于 Remoting，但是更强大，用处也更大。如果使用 Remoting，那么我们需要在基础设施或者代码中自己考虑高可用性这样的问题。而 Akka Cluster 会负责解决很多这些问题，因此是我们构建分布式 Actor 系统的绝佳选择。

我们就需要在 src/main/resources/ application.conf 中添加集群配置。

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  remote {
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2551
    }
  }
  cluster {
    seed-nodes = [
      "akka.tcp://actorSystem@127.0.0.1:2552",
      "akka.tcp://actorSystem@127.0.0.1:2551"
    ]
  }
}
```

```
extensions = ["akka.cluster.client.ClusterClientReceptionist"]
}
```

下面是一些重要点。首先，Cluster 的配置和 Remoting 的配置非常像，但是我们把 provider 改成了 ClusterActorRefProvider。

我们要指定主机和端口。在这个例子中使用了 2551 端口，并且指定本地主机的 IP 用于测试。要在单台机器上测试一个集群，就需要在不同的端口上启动实例。我们可以通过在命令行内向 main 方法传递参数来完成这一点。如果需要传入任何参数，可以使用下面的命令：

```
-Dakka.remote.netty.tcp.port=#port
```

传入 port=0 表示由 Akka 随机分配一个端口。

我们还指定了种子节点。下文会介绍种子节点的具体定义，不过要注意主机、端口以及 ActorSystem 就是在配置种子节点时指定的。一定要确保指定的 ActorSystem 就是用于希望加入的集群。由于同一个实例内可以运行多个 ActorSystem，因此单单指定主机和端口并不足以确保成功连接。

最后一行指定了需要的扩展，我们在这行中加入了对于 contrib 包中 cluster 客户端的支持，稍后我们将详细介绍。

种子节点

读者可能很好奇 akka.cluster.seed-nodes 这个配置的作用。由于集群中可以包含任意数量的节点，我们可能不知道所有节点的地址。尤其是部署在云端的时候，部署的拓扑和 IP 地址可能经常变化。

还好我们有 gossip 协议，只要知道几个节点的位置就可以解决这个问题了。大多数技术（比如 Cassandra 和 Akka）把这些节点称作种子节点。除了知道它们的访问地址外，种子节点并无其他特殊之处。

为了理解种子节点的具体机制，我们将介绍节点加入集群的方式。当一个新节点加入集群时，该节点会尝试连接第一个种子节点。如果成功连接种子节点，新节点就会发布其地址（主机和端口）。种子节点会负责通过 gossip 协议将新节点的地址最终通知整个集群。如果连接第一个种子节点失败，新节点就会尝试连接第二个种子节点。只要成功连接任何一个种子节点，那么任何节点加入或离开集群时，我们都不需要对配置进行任何修改。

当部署到生产环境时，应该至少定义两个拥有固定 IP 的种子节点，并且保证任何时候都至少有一个种子节点可用。当一个节点尝试加入集群时，会试图顺序连接种子节点。如果所有的种子节点都不可用，那么新节点将无法加入到集群，如图 1-15 所示。

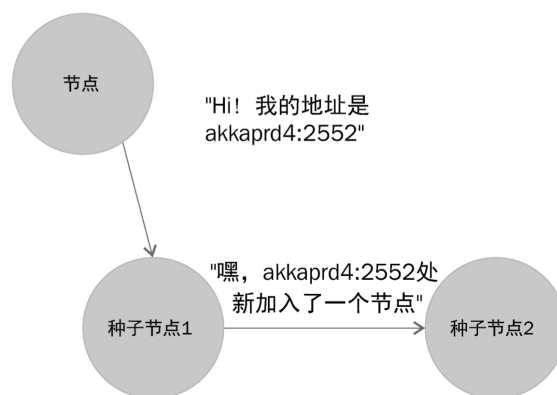


图1-15

订阅集群事件

现在已经完成了在运行时创建一个集群所需的全部配置，所以可以开始编写代码了。我们将订阅集群事件，将集群环形拓扑中发生的所有变化都记录到日志。代码编写完成后，我们会对代码进行测试，然后继续学习如何在此基础上设计分布式服务和数据库。

首先，创建一个 Actor，命名为 ClusterController，这个 Actor 会作为其他例子的基础。接着，编写代码，使之在事件发生时执行特定操作。我们将首先创建 Actor，实例化一个 Logger，接着创建集群对象。

```
public class ClusterController extends AbstractActor {
    protected final LoggingAdapter log = Logging.getLogger(context().system(), this);
    Cluster cluster = Cluster.get(getContext().system());

    @Override
    public void preStart() {
        cluster.subscribe(self(), ClusterEvent.initialStateAsEvents(), ClusterEvent.MemberEvent.class,
ClusterEvent.UnreachableMember.class);
        log.info("Actor {} Up, Path: {}", self().path().name(), self().path());
    }

    @Override
    public void postStop() {
        cluster.unsubscribe(self());
    }

    @Override
    public Receive createReceive() {
        return ReceiveBuilder.create()
            .match(ClusterEvent.MemberEvent.class, message -> {
                log.info("MemberEvent: {}", message);
            })
            .match(ClusterEvent.UnreachableMember.class, message -> {
                log.info("UnreachableMember: {}", message);
            })
            .match(ClusterLeaveMessage.class, message -> {
                log.info("Cluster Leave Message: {}", message);
                cluster.leave(Address.apply(message.getProtocol(), message.getSystemName(), message.
getHost(), message.getPort()));
            })
            .build();
    }
}
```

首先定义一个 Logger。然后获取指向 Cluster 对象的引用。对于 Cluster 对象及其可用方法的介绍会贯穿本章。

使用 Actor 的 preStart 和 postStop 方法来订阅感兴趣的事件，要记得在 postStop 方法中调用 unsubscribe，防止泄露。Actor 将订阅下面两个事件：

- MemberEvent：该事件会在集群状态发生变化时发出通知。
- UnreachableMember：该事件会在某个节点被标记为不可用时发出通知。

接着，定义 Actor 在接收到上述事件时的行为：暂时直接输出到日志。稍后会介绍集群中发生的不同事件。

1.6.8 启动集群

现在将启动几个节点，确认所有配置都准确无误。首先，我们需要一个 main 函数，作为以程序方式启动 Actor 系统的入口。然后，创建 ClusterController Actor。

```
public static void main(String... args) {
    ActorSystem system = ActorSystem.create("actorSystem");
    system.actorOf(Props.create(ClusterController.class), "clusterController");
}
```

不过为了节点能够安全退出集群，我们还将启用节点的 jmx 远程原理功能。所以我们还会在上面的命令中再传入几个参数：指定节点的 jmx 端口，并且在测试中关闭 jmx 的安全功能。稍后我们将介绍启用 jmx 远程管理功能的原因。加上 jmx 配置后，启动节点的命令如下：

```
-Dakka.remote.netty.tcp.port=2551
-Dcom.sun.management.jmxremote.port=9551
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

该命令会在配置好的 2551 端口上启动节点。我们将看到日志中输出了一些事件，表示第一个种子节点已经启动完成。也有可能由于节点尝试连接其他配置好的种子节点而看到几条 dead-letters 消息。

接下来更改 port 端口为 2552 以启动第二节点

最终，我们将看到接收到 MemberEvent 时输出到日志的内容：

```
MemberEvent: MemberUp(Member(address = akka.tcp://actorSystem@127.0.0.1:2552, status = Up))
```

现在我们就再添加一个节点，展示非种子节点的配置。我们给种子节点指定了端口，现在就将端口指定为 0，由 Akka 来随机分配一个端口。

稍微过一段时间后，就可以看到第三个节点连接到了集群。我们现在就有了构建分布式 Actor 系统的基础了。其中的很多步骤如果要自己实现的话还是有点困难的，但是 Akka 帮我们解决了这些问题。

优雅地退出集群

如果试图通过杀死 (kill) 进程来关闭某个节点的话，就会发现 Akka 会把这个节点标记为不可到达，然后输出一些错误信息。在这种情况下，关闭节点会导致该节点无法访问，Akka 最终会将其标记为关闭。原因在于我们并没有优雅地退出集群。在将一个节点从集群中删除之前，我们应该通知集群：我们要移除这个节点了。

要在程序中完成这一操作，我们可以调用 cluster.leave，传入想要删除的节点的地址：

```
Cluster cluster = Cluster.get(getContext().system())
cluster.leave(self().path().address());
```

集群成员的状态

加入集群后的节点可能会处在多种状态中的一种。在底层有一个逻辑上的 leader 节点，负责协调状态的变化。集群会从逻辑上对节点进行排序，集群中的所有节点都会遵循这个顺序。排序列表中的第一个节点就是 leader 节点。

Leader 节点会对加入和离开集群的请求做出响应，并修改集群成员的状态。

在加入集群时，将要加入的节点会将其状态设为 Joining。Leader 节点会做出响应，将其状态改为 Up。同样地，如果一个节点将状态设置为 Leaving，那么 leader 节点会做出响应，先将其状态改为 Exiting，然后再改成 Removed。所有这些状态改变都是通过 MemberEvent 在集群间发送的，我们订阅的就是 MemberEvent，并在发生该事件时输出日志。

失败检测

在 Actor 中还可以处理集群中可能出现的另一种情况：集群中的节点可以通过失败检测，检测其他节点是否可以到达。当某个节点出于任何原因被检测为不可到达时（比如发生崩溃或是暂时的网络错误），该节点的状态不会发生变化，但是会被标记为 MemberUnreachable。在 ClusterController 中订阅这个事件，这样每次标记 MemberUnreachable 的时候就能收到通知。如果在合理的时间段内该节点又重新变得可达，那么该节点就会重新运行。如果在配置的时间内始终不可到达，那么 leader 节点就会将该节点标记为 Down，该节点将无法重新加入集群。

从功能性的角度来看，也可以直接监控成员状态的变化，但是失败检测的实现其实是基于从集群中收集到的数据计算出的不可达可能性（phi）。如果读者想要进一步了解的话，Cluster Specification 中有一个失败检测实现方法的链接。

我们将使用默认配置，但是在实际部署中，一定要阅读文档，根据网络可靠性调整配置。Amazon EC2 Instance（AWS）的可靠性明显要比小型的局域网络差很多，所以和自己的虚拟设施及网络设备比起来，使用 AWS 的时候可能需要对发生临时分区访问故障时的响应速度做更多考虑。在云环境中进行大规模部署后，我发现由于临时网络故障导致的服务中断要比普通操作导致的异常更为常见。

值得注意的是，如果一个节点被标记为不可达，那么 Akka 集群不会改变任何节点的状态：也就是说，在该节点由于无法从不可达状态恢复，被标记为 Down 之前，Akka Cluster 将不会改变任何节点的状态。

如果节点无法访问并且被标记为 Down，那么该节点在这之后将服务重新加入集群。结果会产生两个分离的集群（形成所谓的“左右脑”的情况）。目前 Akka 没有解决这个问题。所以一旦某个节点被标记为 Down，就必须关闭该节点，重新启动，获得一个新的唯一 ID，才能重新加入集群。

1.6.9 通过路由向集群发送消息

我们已经学习了如何创建集群。现在就来学习如何向集群发送消息。

在启动集群时直接启动 Actor（或是 Pool 中的多个 Actor）。Main 函数如下所示：

```
public static void main(String... args) {
    ActorSystem system = ActorSystem.create("actorSystem");
    system.actorOf(Props.create(ClusterController.class), "clusterController");
    ActorRef workers = system.actorOf((new BalancingPool(20).props(Props.create(RequestReceiver.class))),
    "workers");
    ClusterClientReceptionist.get(system).registerService(workers);
}
```

用于集群服务的集群客户端

编写一个客户端用于与无状态集群服务进行通信是相当直接的。和使用前置负载均衡器的传统 Web 服务相比，Akka Cluster 提供了一些便利之处：集群可以动态扩展或收缩，无需修改负载均衡器的配置。因为客户端可以将消息路由至集群中的任意成员，所以对基础设施的要求就更低一些。由于客户端知道集群信息，所以当集群内包含的节点数增加或减少时，客户端也会修改其可以发送消息的服务列表。客户端内部通过负载均衡将请求发送到集群中所有节点的原理，如图 1-16 所示。

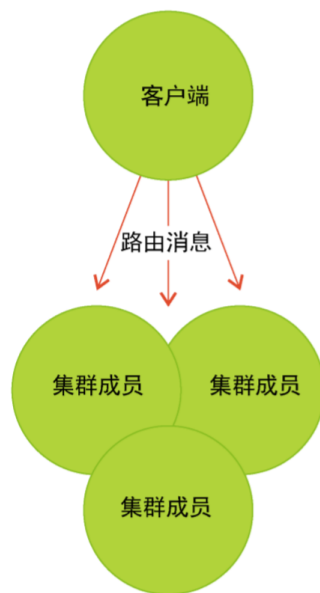


图1-16

我们运行了一个包含 3 个节点的集群，下面就是客户端与其通信的必需条件：

1. 在服务器项目中启用集群客户端。
2. 客户端中必须包含要发送给服务的消息。
3. 客户端本身不是集群成员，但是必需知道集群的拓扑结构。所以我们将使用 contrib 库中的 Akka Cluster Client。
4. 客户端必须知道要将消息发送到哪些 Actor 或 Router，以及由哪些 Actor 或 Router 来响应集群事件。

客户端中需要新建并编辑 application.conf，确保 provider 配置正确，并且设置 Cluster Client 的邮箱。

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  contrib.cluster.client {
    mailbox {
      mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
      stash-capacity = 1000
    }
  }
}
```

在连接成功之前，请注意不要往邮箱发送太多消息。Cluster Client 会暂存这些消息。

向集群发送消息

既然我们已经配置好了集群，可以和使用 contrib 包中 Cluster Client 的客户端进行通信，现在就可以向集群发送消息了。

在客户端的 main 方法中，加入以下代码：

```
public static void main(String... args) throws Exception {
    ActorSystem system = ActorSystem.create("clientSystem");
    ActorRef receptionist = system.actorOf(ClusterClient.props(ClusterClientSettings.apply(system).
withInitialContacts(initialContacts()))), "client");
    ClusterClient.Send msg = new ClusterClient.Send("/user/workers", new SetRequest("key", "value"), false);
    IntStream.range(0, 5).forEach(i -> receptionist.tell(msg, receptionist));
}

private static Set<ActorPath> initialContacts() {
    return Sets.newHashSet(
        ActorPaths.fromString("akka.tcp://actorSystem@127.0.0.1:2551/system/receptionist"),
        ActorPaths.fromString("akka.tcp://actorSystem@127.0.0.1:2552/system/receptionist"));
}
```

这其中，需要生成用于和集群中 Receptionist 进行通信的客户端。首先创建将会与客户端进行通信的包含种子节点的 Receptionist 清单。创建了包含种子节点的地址的 Set 后，我们就可以创建 ClusterClient Actor：

```
ActorRef receptionist = system.actorOf(ClusterClient.props(ClusterClientSettings.apply(system).withInitialContacts(initialContacts())), "client");
```

现在，应用程序就可以连接到集群，并获取任何集群拓扑结构的更改信息了。远程 Actor 系统中的 Receptionist 可以接收一些不同的消息：

- ClusterClient.Send：将消息发送至任意一个节点。
- ClusterClient.SendToAll：将消息发送至集群中的所有节点。
- ClusterClient.Publish：将消息发送给订阅了某个主题的所有 Actor。

只需要将消息发送给任意一个节点，所以对于这种情况用 Send 就可以了。我们构建 Send 对象，指定消息要发送至的目标 Actor（运行在集群中每个节点上的 Router），并且传入要发送的消息，执行 5 次：

```
ClusterClient.Send msg = new ClusterClient.Send("/user/workers", new SetRequest("key", "value"), false);
IntStream.range(0, 5).forEach(i -> receptionist.tell(msg, receptionist));
```

我们在两个服务器端会看到如下接收到消息的日志：

```
[akka://actorSystem/user/workers/$a] Received Set request: SetRequest[key='key', value=value]
[akka://actorSystem/user/workers/$a] Received Set request: SetRequest[key='key', value=value]
[akka://actorSystem/user/workers/$a] Received Set request: SetRequest[key='key', value=value]

[akka://actorSystem/user/workers/$a] Received Set request: SetRequest[key='key', value=value]
[akka://actorSystem/user/workers/$a] Received Set request: SetRequest[key='key', value=value]
```

要使用 Akka 来构建分布式 worker 节点，仅此而已。现在我们可以做很多事情来改进这个系统。如果可能会有非常多的消息，那么可能会想把消息放在其他地方，而不是内存中的信箱里。虽然对于实时处理来说，先使用临时内存可行，但是我们可能会要使用某种持久化队列或者是数据库。对于客户端来说，我们也可能需要编写超时和重试的语义，确保能够完成需要完成的任务。

1.7 Akka Actor 邮箱

直到现在为止，都没有详细介绍过邮箱。我们只是知道消息存储在邮箱里，然后被处理。在服务被大规模使用之前，只要知道这些就绰绰有余了。这也是为什么我把本章放在全文接近结尾的地方——我们很早就介绍了邮箱，但是只有在处理实际网络流量的时候才会需要对其进行优化。

1.7.1 邮箱配置

在 Akka 中，有多种方法可以对邮箱进行配置。如果想了解更多细节的话，可以查阅 Akka Mailboxes 的文档。

几乎在任何情况下，Actor 都有自己的邮箱。唯一的例外是使用 Balancing Dispatcher 的 BalancingPool。

BalancingPool 中的 Actor 共享同一个邮箱。所有可以在两个地方对邮箱进行配置：一个是在 Actor 里进行配置，另一个是在 Dispatcher 中进行配置。

我们将介绍配置邮箱的不同方法。

1.7.2 在部署配置中选择邮箱

我们已经知道如何使用程序来定义并实例化 Actor。除此之外也可以使用 Akka 的部署配置，在配置文件中配置 Actor 和 Router。

我们可以在部署配置中定义 Actor 的邮箱（通过 Actor 的路径）。部署配置中的定义在所有邮箱配置中优先级最高。可以在 application.conf 中对 Actor 邮箱定义如下：

```
akka.actor.deployment {  
  /myactor {  
    mailbox = default-mailbox  
  }  
}
```

这样创建于/user/myactor 的 Actor 就会使用默认邮箱：

```
ActorRef clusterController = system.actorOf(Props.create(MyActor.class), "myactor");
```

1.7.3 在代码中选择邮箱

我们也可以在代码中定义要使用哪个邮箱。Props 有一个 withMailbox 方法，我们可以在创建 Actor 的时候调用该方法，为 Actor 分配邮箱：

```
ActorRef clusterController = system.actorOf(Props.create(MyActor.class).withMailbox("default-mailbox"));
```

1.7.4 决定使用哪个邮箱

默认邮箱可以用于所有的用例，包括多个 Actor 之间会共享邮箱的 BalancingPool/BalancingDispatcher。

除非使用 BalancingPool，否则邮箱中的消息只会有一个消费者，如图 1-17 所示。

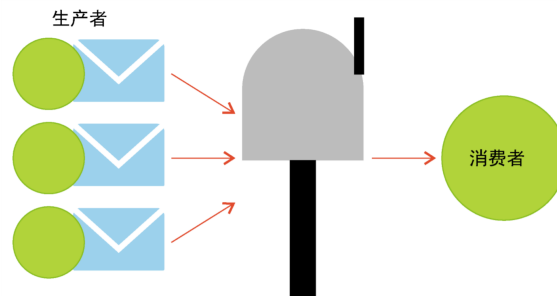


图1-17

在单个消费者的情况下，除了默认邮箱外，还可以使用 `SingleConsumerOnlyUnboundedMailbox`，该邮箱在大多数情况下效率优于默认邮箱（对于与效率有关的修改，一定要记得测试性能）。唯一不能使用该邮箱的情况就是 `BalancingPool/Balancing Dispatcher`，原因在于会有多个 Actor 读取同一个邮箱中的消息。使用队列的实现只支持单个消费者。

首先在 `application.conf` 中定义一个邮箱：

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

然后创建使用该邮箱的 Actor：

```
ActorRef clusterController = system.actorOf(Props.create(MyActor.class).withMailbox("akka.actor.mymailbox"));
```

我们经常会需要决定是否要限制邮箱中消息的最大数目。在大多数情况，没有消息数目限制的邮箱效率更好，推荐使用。不过这个例子将使用有消息数目限制的邮箱，会在达到限制时拒绝接收消息。我们不希望在负载突然变大时出现内存溢出，反之，我们会拒绝接收消息。

有两种类型的限制消息数量的邮箱——阻塞的和非阻塞的。所有的邮箱都基于队列。在这种情况下，阻塞意味着当邮箱已满时，再向邮箱发送消息会导致线程等待直至邮箱腾出空间，而非阻塞意味着此时消息会被丢弃。由于我们已经决定丢弃消息，所以可以使用 `NonBlockingBoundedMailbox`。我们在配置文件中添加该邮箱：

```
akka.actor.boundedmailbox {  
  mailbox-type = "akka.dispatch.NonBlockingBoundedMailbox"  
  mailbox-capacity = 1000000  
}
```

然后实例化使用该邮箱的 Actor：

```
ActorRef clusterController = system.actorOf(Props.create(MyActor.class).withMailbox("akka.actor.boundedmailbox"));
```

现在，如果系统负载突然变得很大，就将丢弃消息，但是系统仍将正常运行。

那么如果消息被丢弃，会发生什么呢？与下游系统进行通信的系统应该实现超时以及重试机制。我们要预设消息有时候可能会丢失，所以如果要保证消息至少能够被处理一次的话，构建系统的时候就需要记录请求的状态，无论任何原因导致请求失败，都要重发请求。在我们的示例应用程序中，客户端可能会发生错误，所以用户可以选择重试。而在我们自己的系统中，要实现超时和重试机制都很容易。

1.7.5 提高邮箱中消息的优先级

我们还应该了解另外两大类的邮箱：优先级邮箱和控制感知的邮箱。它们的目的相同：支持消息处理的顺序。

优先级邮箱在接收到消息之后，可以对消息进行排序，给每个消息赋予一个优先级。这会造成明显的额外性能开销：每次接收到消息时都要重新排序，这意味着消息队列的生产者和消费者都必须要等待消息重新排列完成。幸运的是，消息的重排序在大多数情况并不重要，所以大多数情况下我们都不需要使用这些邮箱。

不过有一种情况比较常见：如果需要给 Actor 提供某种控制消息，告诉 Actor 发生了某些改变，而这些改变会影响处理队列中消息的方式。

幸运的是，还有一种邮箱类型：ControlMessageAware 邮箱。该邮箱也通过一个高效队列来处理消息，但是允许将任意扩展了 akka.dispatch.ControlMessage 的消息插入到队列头部。如果 Actor 从邮箱队列的剩余消息中获取消息进行处理时不需要太复杂的时序逻辑，那么这种解决方案比优先级队列要好，应该优先选择使用。

这两种邮箱使用得很少，所以我们不需要深入了解，但是知道它们的存在还是很重要的。Akka 文档深入介绍了两者的使用。

最后值得一提的一个注意点是，如果所有邮箱都不符合我们的使用场景，那么可以创建自己的邮箱。

1.8 参考资料

《Learning Akka》- Jason Goodwin

《Akka 入门与实践》- 诸豪文 译 (2017 年 6 月第 1 版)

<https://akka.io/docs/> - Akka 官方文档

2. 架构概念

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。

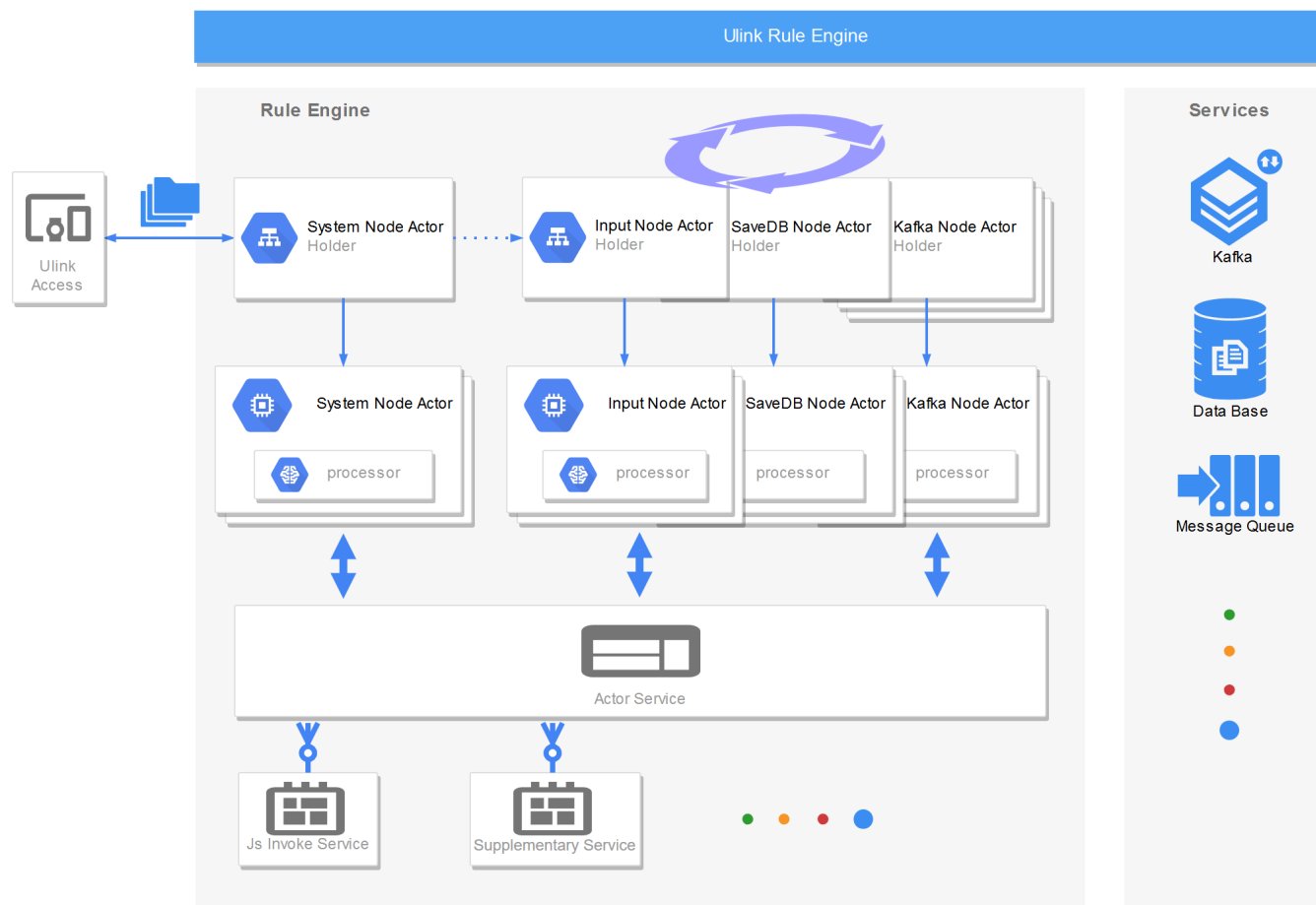


图2-1

2.2 目录结构图

下面我们从项目目录结构的角度，介绍一下 Rule Engine 的文件结构，如图2-2所示。

作为高度集成、支持可插拔式的规则引擎核心组件 Actor System 所有的文件都在目录 actorsystem 下。

所有与 actor 相关的内容都放置在 actor 目录中，由 nodeactor、nodeholderactor、processor 三个目录组成。而 processor 下又被分为 nodeholderprocessor 和 nodeprocessor 两部分。

actorservice 中是所有与 service 相关的文件，包括 Actor System 生命周期服务、外部数据库依赖服务等。

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。

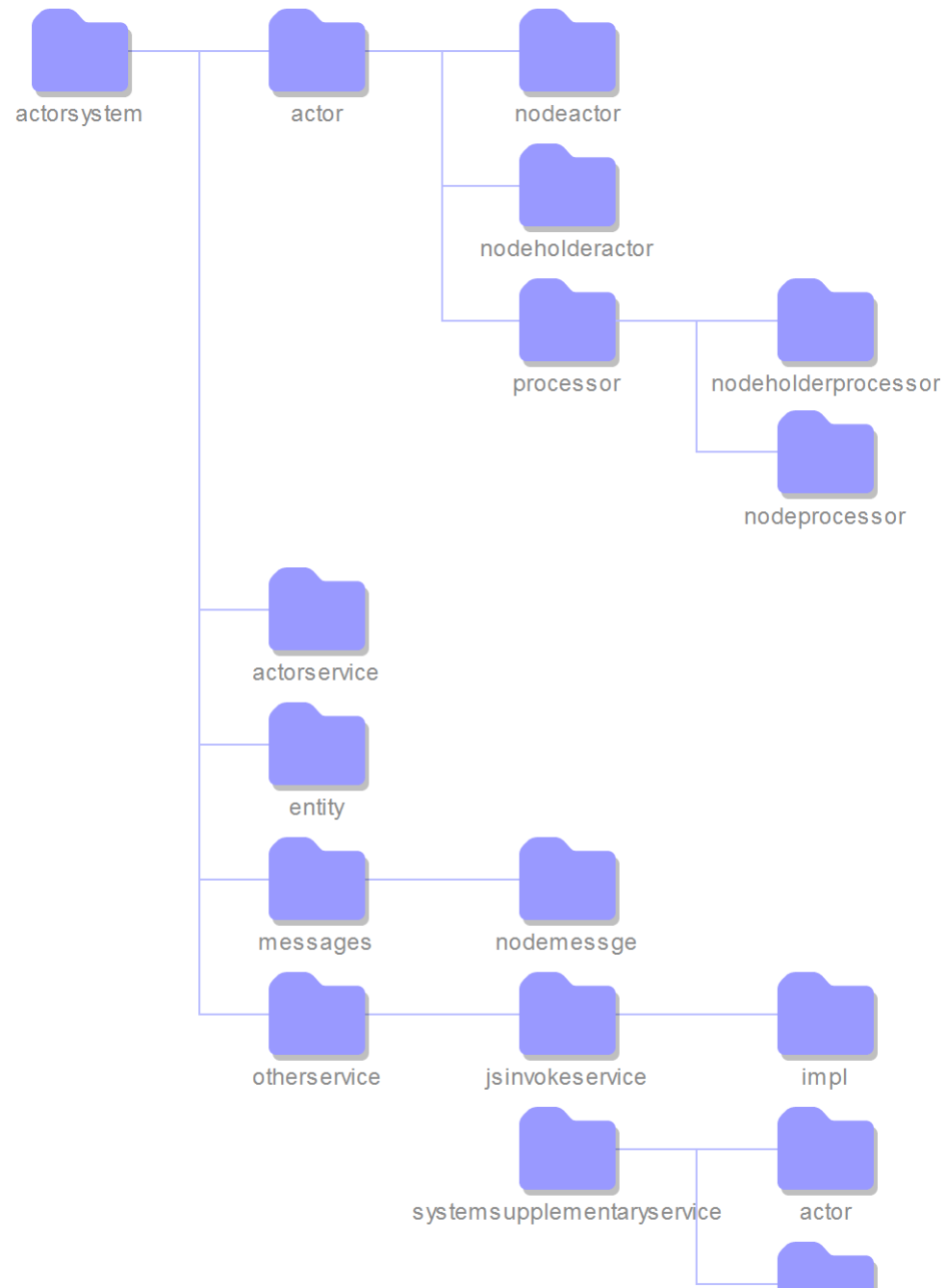




图2-2

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。

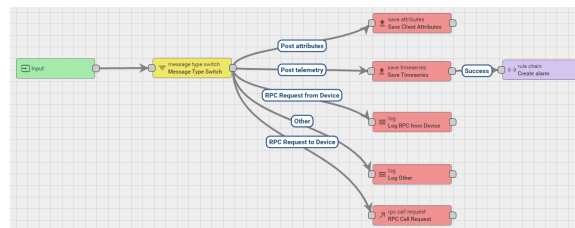


图2-3

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。

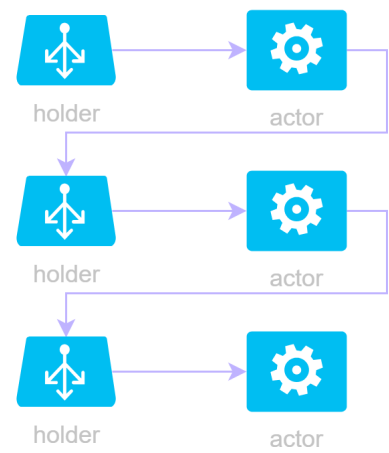
这样一看，读者可能已经明白了 Rule Engine 的概念。

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。

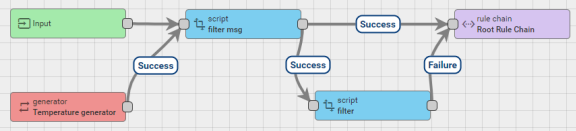


图2-4

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 Chain 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。

3. 原理与开发

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 Service，后面会介绍到。

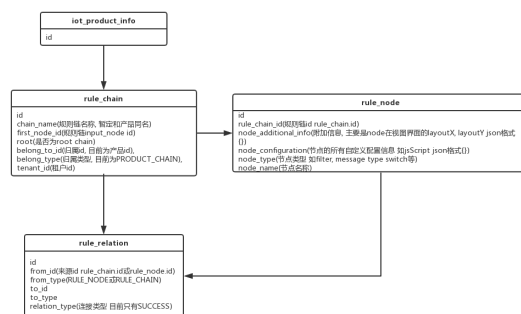


图3-1

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
NodeBuildPrepareData prepareData = this.prepareNodeBuildData(ruleChainIdList);
List<MongoRuleEngineRuleChainInitV> addedRuleChainVList = prepareData.getAddedRuleChainVList();
Map<String, Map<MongoRuleEngineRuleChainInitV> ruleChainIdMap> = prepareData.getRuleChainIdMap();
Map<String, List<MongoRuleEngineRuleChainInitV>> chainsGroupByBelongToIdMap = prepareData.getChainsGroupByBelongToIdMap();
Map<String, List<MongoRuleEngineRuleChainInitV>> nodesGroupByRuleChainIdMap = prepareData.getNodesGroupByRuleChainIdMap();
Map<String, List<MongoRuleEngineRuleRelationInitV>> relationsGroupByRuleChainIdMap = prepareData.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 用于 一个链路中 一个节点如果为多个上级节点的目标 则使看到所有上级节点
// 发来的init node messages，则会进行多次初始化，此标记为了标记该节点已被定时问题的链路初始化任务的初始化过 不需要再次初始化
long lastInitMsg = System.currentTimeMillis();

// send node init message to every rule chain to init all node actors
addRuleChainToInitParallelStream().forEach(ruleChain -> {
    String ruleChainId = ruleChain.getId();
    Map<String, List<WriteRuleEngineRuleRelationInitV1>> ruleChainRelationGroupByFromIdMap =
        relationGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(WriteRuleEngineRuleRelationInitV1::getFromId));
    Map<String, List<WriteRuleEngineRuleRelationInitV1>> ruleChainRelationGroupByToIdMap =
        relationGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(WriteRuleEngineRuleRelationInitV1::getToId));
    Map<String, WriteRuleEngineRuleRelationInitV1> ruleNodeInitMsg = nodeGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.toMap(WriteRuleEngineRuleRelationInitV1::getId,
        v -> v));
    ruleChainRelationGroupByFromIdMap.get(ruleChainId).forEach(ruleRelationInitV1 -> {
        String ruleNodeId = ruleRelationInitV1.getToId();
        ActorTypeRegisterEnum ruleNodeActorTypeEnum = ActorTypeRegisterEnum.valueOf(ruleNodeInitMsg.get(ruleNodeInitV1.getToId().getRuleType()));
        ActorSystemContext getRuleNodeMap(); getRuleNodeActorTypeEnum();
        willNew RuleMessageCarrier()new RuleId(ruleNodeInitV1.getRuleId(),
            null,
            new RuleIdInitMessage(ruleChainRelationGroupByFromIdMap,
                ruleChainRelationGroupByToIdMap,
                ruleNodeInitMsg,
                relationGroupByRuleChainIdMap,
                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 会做些什么事情，现在我们更关心整体的节奏。

```
// associate system holder actor with root chain first node and init rule chain context
addRuleChainToInitParallelStream().forEach(ruleChain -> {
    String ruleChainId = ruleChain.getId();
    String firstNodeId = ruleChain.getFirstNodeId();
    Map<String, WriteRuleEngineRuleRelationInitV1> ruleNodeInitMsg = nodeGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.toMap(WriteRuleEngineRuleRelationInitV1::getToId,
        v -> v));
    Map<String, List<WriteRuleEngineRuleRelationInitV1>> ruleChainRelationGroupByFromIdMap =
        relationGroupByRuleChainIdMap.get(ruleChainId).stream().collect(Collectors.groupingBy(WriteRuleEngineRuleRelationInitV1::getFromId));
    ActorSystemContext getRuleNodeMap(); getRuleNodeActorTypeEnum();
    willNew RuleMessageCarrier()new RuleId(ruleChainId,
        null,
        new RuleIdInitMessage(ruleChainRelationGroupByFromIdMap,
            null,
            ruleNodeInitMsg,
            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 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();
ruleChainList.parallelStream().forEach(ruleChain -> actorSystemContext.getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM)
    .tell(new NodeMessageCarrier<>(null,
        null,
        new NodeDeleteMessage(ruleChain, false, IorRuleChainRootEnum.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
ruleChainList.parallelStream().forEach(ruleChain -> {
    String ruleChainId = ruleChain.getId();
    String firstNodeId = ruleChain.getFirstNodeId();
    Map<String, List<IRuleEngineRuleRelationInit>> ruleChainRelationsGroupByFromIdMap =
        relationsGroupByRuleChainIdMap.get(ruleChainId, stream().collect(Collectors.groupingBy(IRuleEngineRuleRelationInit::getFromId));
    Map<String, List<IRuleEngineRuleRelationInit>> ruleChainRelationsGroupByToIdMap =
        relationsGroupByRuleChainIdMap.get(ruleChainId, stream().collect(Collectors.groupingBy(IRuleEngineRuleRelationInit::getToId));
    Map<String, IRuleEngineRuleNodeInit> ruleNodeIdMap =
        nodesGroupByRuleChainIdMap.get(ruleChainId, stream().collect(Collectors.toMap(IRuleEngineRuleNodeInit::getId, v -> v));
    actorSystemContext.getActorHolderMap().get(ActorTypeMapperEnum.SYSTEM)
        .tell(new NodeMessageCarrier<>(new NodeId(firstNodeId),
            null,
            new NodeUpdateMessage(new NodeInitMessage(
                ruleChainRelationsGroupByFromIdMap,
                ruleChainRelationsGroupByToIdMap,
                ruleNodeIdMap,
                chainIdGroupByToIdMap,
                ruleChainIdMap,
                timeStamp),
            new NodeDeleteMessage(ruleChainId, true, IorRuleChainRootEnum.ROOT.getCode().equals(ruleChainId.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是否存在
Optional<NodeId> nodeIdOpt = super.actor.getHolderIdMap().keySet().parallelStream().filter(targetNodeID -> targetNodeID.equals(nodeId)).findFirst();
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 targetNodeID = nodeIdOpt.get();
    NodeLifeCycleInfo nodeLifeCycleInfo = super.actor.getHolderIdCycleMap().get(targetNodeID);
    if (null == nodeLifeCycleInfo) {
        super.logNodeLifeCycleInitFull(super.actor.getType().getHolderIdActorClass().getSimpleName(), nodeInitMessage, nodeID);
    } else {
        if (nodeLifeCycleInfo.getInitTimestamp() == nodeInitMessage.getInitTimestamp()) {
            super.logLogNoOccurWhenWarn(super.actor.getType().getHolderIdActorClass().getSimpleName(), "a warn", String.format("process %s",
nodeInitMessage.getClass().getSimpleName()), "node actor already create, nodeId: " + targetNodeID);
            return;
        }
    }
}
ActorRef actorRef = createHolderActor(nodeID, nodeInitMessage);
if (null == actorRef) {
    return;
}
initHolderActor(actorRef, nodeInitMessage);
// 暂时缓存该message
if (checkIfNodeIDIsLocked(nodeID)) {
    releaseCacheMessage(nodeID);
}
```

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 (checkIfNodeIDIsLocked(messageCarrier.getTargetNodeID()) && !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.logReceiveFromKindOfMessage(super.logMsg, message);
List<MessageInboxHolderRelationInitV> toBeRelationList = super.actor.getToBeRelationList();
// 如果删除时消息记已标记过，并且等于本节点的删除标记，说明删除message已经下发到下游node，不需要再次下发
if (message.getMessageStamp() != super.actor.getMessageStamp() && CollectionUtil.isEmpty(toBeRelationList)) {
    super.actor.setLatestTimeStamp(message.getMessageStamp());
    super.sendToNextHolderActor(super.actor.getHolderId(), message, toBeRelationList, super.actor.getChainIdMap(), super.actor.getActorHolderMap(), false);
}
// 所有的上游delete message 都到达后，才会停止此node actor
if (super.isLatestMsg()) {
    List<MessageInboxHolderRelationInitV> 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
@MsgActorRef, NodeId nodeId;
if (nodeIdMap.containsKey(actorRef)) {
    NodeId nodeId = nodeIdMap.get(actorRef);
    NodeLifecycle nodeLifecycle = super.actor.getHolderLifecycleMap().remove(nodeId);
    if (null == nodeLifecycle) {
        super.logHolderLifecycleNull(super.actor.getType(), getHolderActorClass(), getSingleName(), nodeStopMessage, nodeId);
    }
    this.log.logHolderLifecycleDebug(super.actor.getType(), getHolderActorClass(), "a debug", String.format("process %s",
        nodeStopMessage.getClassName(), "remove actor ref: " + actorRef.path() + " node id: " + nodeId));
    super.actor.getActorSystemContext().getActorHolderMap().getActorTypeMap().tell(new NodeMessageCarrier<>(nodeId, nodeStopMessage, nodeId));
    else {
        this.log.logHolderLifecycleError(super.actor.getType(), getHolderActorClass(), "an error", String.format("process %s",
            nodeStopMessage.getClassName(), "find no actor ref: " + actorRef.path() + " node id: " + nodeId));
    }
}
```

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.ruleChainContextMap.get(ruleChainId);
if (null == ruleChainContext) {
    return;
    super.logRuleChainContextNotExist(super.actor.getType(), getHolderActorClass(), getSingleName(), nodeStopMessage, ruleChainId);
}
// system actor holder 中存在的两个链路的context 如果链路中的节点都stop done则集合为empty，说明链路全部node已经stop 可以进行初始化
List<MessageInboxHolderRelationInitV> nodeInitVList = ruleChainContext.getNodeInitVList();
nodeInitVList.removeIf(endsWithInitV -> endsWithInitV.getId().equals(nodeId));
if (CollectionUtil.isEmpty(nodeInitVList)) {
    // 进行初始化
    NodeInitMessage nodeInitMessage = ruleChainContext.getUpdateInitMessage();
    if (null == nodeInitMessage) {
        SystemNodeHolderActor.ruleChainContextMap.remove(ruleChainId);
    } else {
        Map<String, MessageInboxHolderRelationInitV> ruleModelMap = nodeInitMessage.getHolderMap();
        initializeChainContext(nodeInitMessage, getRelationGroupByFromMap(), ruleChainId, ruleModelMap);
        SystemNodeHolderActor.ruleChainContextMap.get(ruleChainId).getStartNodeInitVList().parallelStream().forEach(startNodeInitV -> {
            String startNodeId = startNodeInitV.getId();
            ActorTypeMapEnum startNodeActorTypeEnum = ActorTypeMapEnum.valueOf(ruleModelMap.get(startNodeId).getType());
            super.actor.getActorSystemContext().getActorHolderMap().get(startNodeActorTypeEnum)
                .tell(new NodeMessageCarrier<>(new NodeId(startNodeId),
                    null,
                    nodeInitMessage),
                    ActorRef.noSender());
        });
        ruleChainContext.getUpdateInitMessage(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。

```
private final boolean chainInit() {
    val chainInitV = nodeUpdateMessage.getDeleteInitMessage().getChainInitV();
    String ruleChainId = valChainInitV.getId();
    SystemNodeChainContext valChainContext = SystemNodeHolderActor.ruleChainContextMap.get(ruleChainId);
    if (null == valChainContext) {
        super.logInfoChainContextInitInfo(super.actor.getType().getHolderClassName(), getInitName(), nodeUpdateMessage, ruleChainId);
    } else {
        if (null != valChainContext.getUpdateInitMessage()) {
            // 上次更新还未完成
            super.logWarnChainContextInitInfo(super.actor.getType().getHolderClassName(), getInitName(), "in error", String.format("process %s",
                nodeUpdateMessage.getClassName(), getInitName(), "rule chain" + valChainId + "] context is updating, update init message" + valChainContext.getUpdateInitMessage()));
            valChainContext.setUpdateInitMessage(nodeUpdateMessage.getDeleteInitMessage());
            return;
        }
        // 这里需要更新旧message 结构
        valChainContext.setUpdateInitMessage(nodeUpdateMessage.getDeleteInitMessage());
        String firstNodeId = valChainInitV.getFirstNodeId();
        // 这里需要删除链路上所有非firstNode的message 非first node 会发送update message 其余的发送delete message
        valChainContext.getUpdateInitV().parallelStream().forEach(nodeInitV -> {
            ActorRef targetRef = super.actor.getActorSystemContext().getActorHolderMap().getActorTypeAppender.valueOf(nodeInitV.getNodeType());
            if (nodeInitV.getId() equals(firstNodeId)) {
                targetRef.tell(messageCarrier, super.actor.self());
            } else {
                targetRef.tell(new NodeDeleteMessageCarrier(new NodeId(nodeInitV.getId()), null, nodeUpdateMessage.getDeleteInitMessage(), 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。

```
// look this chain new message will forward to itself for further consuming (unlock in process init message)
super.actor.getActorHolderHolder().getHolder().lock();
// delete old chain via sending a delete message
ActorRef oldActorRef = super.actor.getHolderHolder().getHolder();
if (null == oldActorRef) {
    super.logWarnChainContextInitInfo(super.actor.getType().getHolderClassName(), getInitName(), "in error", String.format("process %s",
        nodeUpdateMessage.getClassName(), getInitName(), "target actor not exists, target node id: " + nodeId.getId() + ", source node id: " + senderHolder.getId()));
    this.tell(messageCarrier, nodeInitV);
    return;
}
oldActorRef.tell(new NodeDeleteMessageCarrier(new NodeId(nodeInitV.getId()), null, nodeUpdateMessage.getDeleteInitMessage(), 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。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数据检查节点，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。

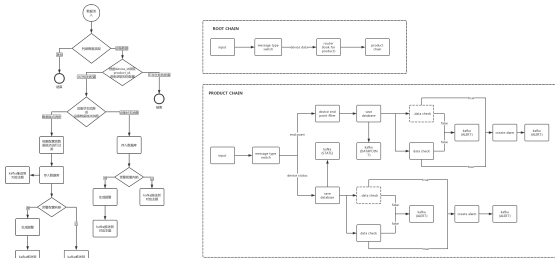


图4-1

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.isEmpty(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.J3_SCRIPT_IL, 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，根据不同情况，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<IoTRuleEngineRuleRelationInitV> ruleChainRelations = iotRuleRelationService.getRuleRelationsFromSystem(Lists.newArrayList(ruleChainId));
List<IoTRuleEngineRuleRelationInitV> neededRelations = Lists.newArrayList();
AtomicReference<String> endPointSaveDBId = new AtomicReference<>();
ruleChainRelations.forEach(relation -> {
    if (relation.getFromType().equals(IotRuleRelationFromTypeEnum.RULE_RULE.name())) {
        if (relation.getFromNode().equals(actorTypeMapperEnum.DATA_FILTER.name())) {
            endPointSaveDBId.set(relation.getToId());
        }
        if (relation.getFromNode().equals(actorTypeMapperEnum.SAVE_DB.name())) {
            All relation.getToNode().equals(actorTypeMapperEnum.DATA_CHECK.name()) {
                neededRelations.add(relation);
            }
        }
    }
});
if (StringUtils.isEmpty(endPointSaveDBId.get())) {
    throw new BusinessException(IotRuleDataServiceErrorCode.RELATION_DATA_ERROR);
}
boolean isRulePoint = triggerType == 1;
Optional<IoTRuleEngineRuleRelationInitV> saveDBRelationOpt = neededRelations.stream()
    .filter(relation -> isRulePoint == relation.getFromId().equals(endPointSaveDBId.get()))
    .findFirst();
if (!saveDBRelationOpt.isPresent()) {
    throw new BusinessException(IotRuleDataServiceErrorCode.SAVE_DB_RELATION_DATA_NOT_EXISTS);
}
IoTRuleEngineRuleRelationInitV 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。最后在此 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。