大数据时代的软件架构范式：Reactive 架构及 Akka 实践

本文原载于《程序员》杂志 2015　2A 期

钟翔

近一两年来，Reactive 突然成为分布式系统设计领域的热词，在 2013 年，业内发布了 Reactive 宣言，到 2014 年，又第一次出现了以 Reactive 为主题的行业会议 Reactconf。那什么是 Reactive，Reactive 又能解决什么问题？本文希望能抛砖引玉，引起读者一些思考。

Reactive 架构要解决什么问题

Reactive 要解决的问题是分布式系统设计中的难点问题，包括：

1. 如何处理各种各样的错误，比如硬件错误、软件错误、网络错误、硬盘错误、内存错误等。错误交给谁处理、怎样处理、怎样交出去、要不要恢复、如何恢复这些问题都很复杂。
2. 软件耦合过紧导致维护复杂。比如要升级一个公用组件，牵一发而动全身。
3. 编程复杂。多线程编程中用锁来保护共享数据，分布式编程中的用事务来实现并发读写，这两个都非常复杂。
4. 性能调优复杂。我们通常很习惯过程式的编程，但调用链上可能有各种阻塞运算比如 IO 等，把它们绑在一个线程栈上是不合适的。而要做优化需要修改大量代码。

这些难点并不直接和业务逻辑相关，但却占用了开发人员大量的时间。Reactive 架构就想解决这些难点，把开发人员从这些业务无关的编程工作中解放出来。Reactive 架构既是一个方法论，又是一个工具集，能够系统性的化解这些复杂性。

什么是 Reactive 架构

Reactive 架构是以消息驱动为核心的架构。实现 Reactive 架构的应用是由许许多多的微型服务编织而成。每个微型服务粒度很小，只做一件简单的事情；微型服务间通过异步的消息发送驱动对方工作，一起组合实现应用所需的功能。

Reactive 架构极像我们人类社会的架构，每个人独立承担自己的工作，相互间通过电子邮件等消息通道来驱动对方工作，实现协同合作。而且这个系统容错性良好，一方面，每个人都可以失败，地球不因某一个人就不转；另一方面，地球上每一个人都在运转，几十亿人分布式工作，忙碌而和谐。人类社会也许是已知的最大的分布式系统。

2013 年，Akka 的发明人 Jonas Bonér 等发布了 Reactive 宣言[1]，从定义上明确了 Reactive 架构必备的四个属性:

- Message driven: 消息驱动。

[1] Reactive 宣言 http://www.reactivemanifesto.org/

- Elastic:负载变化时弹性增减资源。
- Resilient: 失败时能从中恢复。
- Responsive: 按 QoS 质量要求响应用户请求。

如果一个应用满足这四个属性，那我们就称这个应用符合 Reactive 架构。Reactive 技术上最显著的特点就是消息驱动。

Reactive 架构的历史和现实意义

Reactive 本身而言不是一个新的技术。早在上世纪 70 年代，就有科学家对以消息驱动为核心的架构做过细致的理论分析[2]。但在大数据时代的今天，这是一个新的命题，重新认识 Reactive 具有时代价值。一方面，大数据的应用特点决定它正是 Reactive 架构善于解决的问题领域；第二方面，在现今大数据领域按照 Reactive 思想建构的软件还太少；第三方面， 建构 Reactive 应用的工具链正在快速成熟，现在使用正当其时。

历史上，Reactive 架构有两次大发展。第一次是在上世纪 80 年代的电信领域，当时 Ericsson 的 Joe Armstrong 发明了 Erlang 语言，实现了消息驱动的 Actor 模型，用 Actor 实现的 ATM 交换机 AXD301[3]，达到了 99.9999999%[4]的在线率，Actor 模型成为高性能高容错的代表。

第二次大发展是在上世纪末的互联网领域。从 1993 年开始的 10 年期间，互联网用户数爆发增长了 54 倍[5]，在巨大并发访问情况下，许多网站都遇到了性能的瓶颈，即著名的 C10K[6]问题(同时并发请求数无法超过 10K)。为解决这个问题，网站架构逐渐切换到以事件驱动为核心。这里面最负盛名的是 Nginx，它采用了异步 IO 和事件驱动的模型，成功解决了 C10K 问题。Nginx 因此也快速流行起来，2014 年在 Top 1000 流量的网站中，市场占有率达 42%[7]。

Reactive 架构在电信和互联网行业率先落地，原因在于这两个行业对高可用性和高并发性都有非常高的要求，于是相对其他行业更早触碰到了传统架构的瓶颈。

现在已经进入大数据和物联网的新时代，据报道，全球每年的新生成数据将以 140%的速度指数增长[8]；物联网方面，到 2020 年，将有 260 亿设备[9]。大规模和分布式计算将成为一种新常态。Reactive 架构的应用将超越电信和网站系统的范围，影响更多的垂直行业，深入人们生活每一处。正如 Scala 语言发明者 Martin Odersky 所说，“Reactive 编程无所不在[10]”。这正在成为 Reactive 架构发展的第三波浪潮。现在业界已经看到了这股趋势，2014 年，来自 Typesafe、Twitter、Oracle 等公司的有识之士提出了 Reactive Stream[11]接口规范，定义了不同厂商的服务间如何以 Reactive 的方式互联，这样整个行业能够相互兼容的发展。

---

[2] Carl Hewitt http://worrydream.com/refs/Hewitt-ActorModel.pdf
[3] AXD301 http://ll2.ai.mit.edu/talks/armstrong.pdf
[4] AXD301 NINE nines reliability https://pragprog.com/articles/erlang
[5] Internet user trend http://www.internetlivestats.com/internet-users/
[6] C10K http://en.wikipedia.org/wiki/C10k_problem
[7] W3Techs 2014 http://w3techs.com/technologies/cross/web_server/ranking
[8] IDC http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf
[9] Gartner report on IOT
http://www.zdnet.com/article/internet-of-things-devices-will-dwarf-number-of-pcs-tablets-and-smartphones/
[10] Reactive programming is ubiquitous https://class.coursera.org/reactive-001/lecture/3
[11] Reactive Stream http://www.reactive-streams.org/

Reactive 架构的技术要求和设计模式

Reactive 定义的四个属性中，Responsive 是最终目的，Message-Driven, Elastic, Resilient 是手段。本节将按四个属性分组，介绍 Reactive 架构的设计模式和实践经验：

属性一：Message-Driven

Message-Driven 是整个 Reactive 架构的技术基础。Message-Driven 是指系统由消息推动，实现并发和隔离。具体有以下要求：

1. 消息是**唯一**的通信手段。
2. 使用**不可变**的消息。
3. **异步非阻塞**。异步非阻塞是指发送方不用等待接收方处理完消息。异步非阻塞能提高应用的并行度。

要实现这些要求，有以下几种典型设计模式:

函数式编程语言

函数式编程语言非常适合实现事件驱动的编程。函数式编程之前，我们只能用事件回调函数，而回调函数之间不容易组合，比如第二个回调想依赖第一个回调的计算结果，代码会有多层嵌套，写起来很复杂。而函数式编程支持高阶函数，比如 foo1.map(foo2).map(foo3)线性的串联三个回调，编程可以大大简化，其中的 map 就是高阶函数。很多函数式编程语言还提供了 Future Monad，Future Monad 能够串联多个异步非阻塞的调用。比如我们需要先做磁盘 IO，再做数据库访问，使用 Monad 可以这么写：future {do disk IO} flatmap {disk => future {do database query}}

消息队列模式

我们可以用消息队列来异步非阻塞的传送消息。发送方和接受方通过消息队列完全解耦，互相之间没有依赖。Apache Storm 就是一个很好的应用例子，它用到了两类消息队列，一类是 ZeroMQ(Storm 0.8 以前)做跨进程的消息通道，另一个是 Disruptor queue 做进程内的消息通道。

属性二：Responsive

Responsive 是指应用在大负载和软硬件错误的情况下仍然能保持实时的、交互式的响应，它有以下需求：

1. 服务必须在确定的时间边界内响应，无论负载状况如何，无论发生何种错误。
2. 服务间能组合。从前端到后端，一个用户请求可能涉及多个服务协同工作。服务间要能组合，确保从全局来看，系统整体仍然是 Responsive 的。要做到这一点，需要做流速控制，小心设计服务边界。
3. 在线不停机，系统升级必须平滑，不影响服务的可用性。

要实现这些要求，有以下几种典型设计模式:

一问一答模式

一问一答模式的目的是为了确保系统能在确定的时间边界内响应用户请求。一问一答模式是指对每一个“问”的消息都必须“答”一个消息，无论这个“答”的消息代表成功还是失败。比如用户 U 问服务 A 一个问题，而服务 A 在回答前，需要先咨询服务 B， 数据流向是 U -> A ->B。该模式要求：无论 A 是否已经收到 B 的回应，A 都必须在给定的时间边界内回应用户 U。如果 B 没有响应，则 A 必须自己创造一个超时消息发送给用户 U。

路由器模式

路由器模式是为了解决数据流不同节点处理能力不匹配的问题，提高响应速度。假设数据流 A->B->C 中 A 和 C 比较快，而 B 比较慢，整个系统的响应性能受 B 的拖累。路由器模式可以根据负载变化动态调整组件的并发度来解决这个问题。假设 B 是瓶颈，我们可以把 B 转成一个路由器[图 1]，把 B 的工作量分发到多个 B 的子任务上，提高 B 的并发度，同时对 A 和 C 保持透明，即不中断 A 和 C 的计算，这样整个系统的响应时间可以更快。

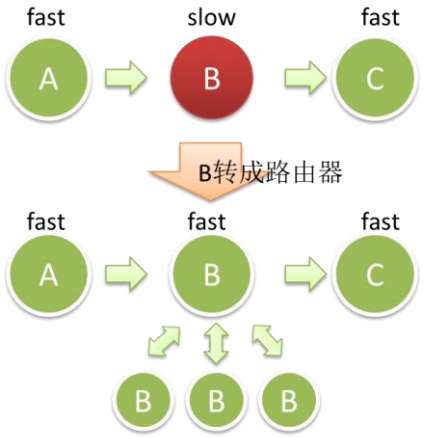

图 1 Router Pattern

开关模式 (Circuit Breaker)

路由器模式仍然假定整个系统有空余资源可用，如果系统已经到了极限，无资源可用，这个时候需要使用断路器模式来使服务平滑降级，保持对用户响应。如图 2 所示，如果负载过大，断路器会切到 Fail Fast 模式，如果负载恢复正常，则会切回到正常 Service 模式。Fail-Fast 模式能及时返回给用户一些信息，缓解用户的焦虑。

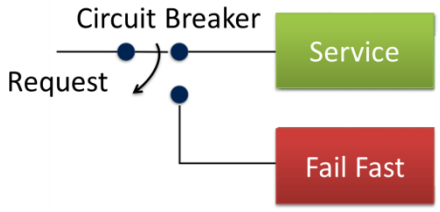

图 2 Circuit Breaker Pattern

问答流控模式

如果两个服务间通信量极大，一方可能耗尽另一方的所有资源怎么办？这个时候必须做流速控制。流控的理论基础是 Little 法则[12]，Little 法则是指可以根据系统中存量消息量和每个消息的处理时间来推算系统的服务容量。流控可以用滑动窗来实现[图 3]，A 发送给 B 已

[12] http://en.wikipedia.org/wiki/Little's_law

发送消息的序列号，B 回复 A 已接收消息的序列号，A 根据 B 的回复来确定是否继续发送消息。通过这一问一答，我们就可以控制传输中消息的滑动窗的大小。

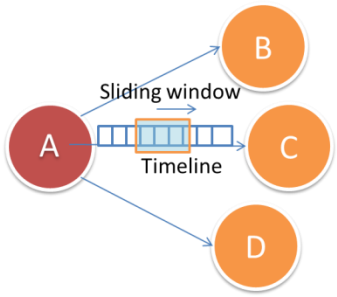

图 3 Sliding Window

属性三: Elastic

Elastic 指应用能弹性使用计算资源，它有以下要求：

1. 增加机器能提高性能。
2. 升级 CPU 能提升性能。
3. 能**响应负载变化**，弹性伸缩。

要实现 Elastic，最重要的是数据的切分和计算的水平扩展。除此之外，还有以下几种设计模式可供参考:

负载状态即消息

系统需要监控负载的变化，并把它定义成显式的消息，让它像正常消息一样在系统中传递。这样的好处是我们能够灵活的选择应对策略以响应负载变化。

无锁设计

每个组件状态都应该是私有的，不允许别的组件访问。这样大部分时候我们可以避免使用锁，因为锁对性能的影响是破坏性的。根据 Amdahl 法则[13]，并行(指多线程运行)化后程序性能加速的上限只取决于代码并发度(指不同的任务逻辑上不相互依赖)，如果代码并发度是 50%，则即使有 100 个 CPU，性能上限仍然是 2 倍。

那有需要共享的数据时如何避免使用锁呢？

锁的目的是控制并发修改，有的时候我们目的只是共享数据，需要修改的是引用而不是数据本身，这时可以用 Immutable 的数据结构解决。比如线程 A 往线程 B 传一个 Immutable Map，B 可以修改，但修改的是一个新的数据副本，和原始数据无关。Scala 语言的 Immutable 数据结构可以在内存上很高效的保存副本。

最终一致性模式

如果一定要对同一份数据做并发修改怎么办？这时还可以用最终一致性模型来解决，同样可

[13] http://en.wikipedia.org/wiki/Amdahl%27s_law

以避免使用锁。这里的典型代表是 CRDT[14]数据结构，CRDT 数据结构允许不同系统对同一份数据写，而且能保证写的结果是收敛一致的。图 4 是用 CRDT 的 GCounter 做分布式计数的例子，a 和 b 代表两个系统，它们各自维护一个计数器，经过两次异步消息交换，计数收敛到了相同的值(a:1, b:3)，而最终的计数是两者相加的结果 4。

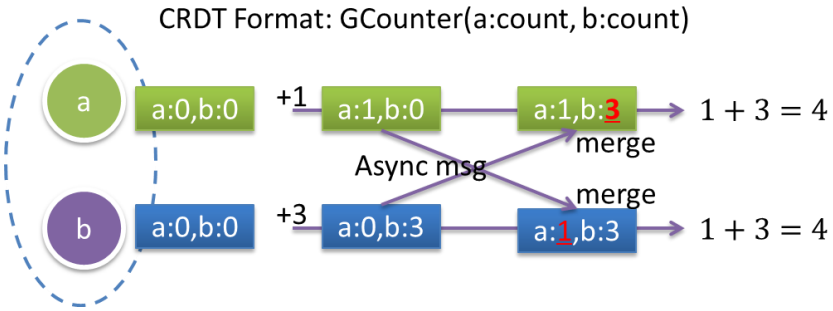

图 4 CRDT GCounter

CRDT 数据类型使用非常广泛，比如 Akka Cluster 集群管理功能用到了 CRDT 数据类型来保存整个集群的状态。除了 GCounter 外，CRDT 还有多种数据类型，比如做分布式加减计数 PNCounter,标记全局状态的 Flag，全局收敛的 Set 和 Map 等。更多数据类型可以参考 NoSQL 数据库 Riak 的实现[15]。

位置透明模式

位置透明是指一个应用的不同组件既可以在一个 JVM 内运行，也可以在一个分布式环境中运行，组件间的交互方式是一样的，和部署环境无关。有了位置透明模式，应用逻辑就和部署环境完全隔离开来，降低了应用的复杂性。

属性四：Resilient

Resilient 是指服务能够容忍错误发生，它有以下要求：

1. 软件异常、硬件错误、网络错误发生时，服务仍然保持可用。
2. 必须响应错误，对错误做恰当的处理和恢复。
3. 一个错误不应链式的传播到整个系统，引起锁链式的失败。

要实现一个 Resilient 的系统，从设计之初就要考虑容错，系统开发好后再增加容错逻辑会非常困难。实现容错的核心是隔离和控制，有以下几种设计模式可供参考。

错误即消息

每一个错误都要显式地定义成消息。比如软件异常、硬件错误，网络连接错误等，这些错误传统上都是隐式定义的消息，在函数的返回值上是不体现的，需要专门的 try catch 语句才能捕捉到。把它们显式定义成消息后，错误就可以像正常消息一样被标记、传输和处理了。

异步错误链模式

异步错误链模式可以系统的解决错误传输和错误处理问题。传统编程实践中，错误发生时，我们都是抛给调用方处理，但调用方未必知道该如何处理。异步错误链模式则不同，它的核

[14] CRDT http://pagesperso-systeme.lip6.fr/Marc.Shapiro/papers/RR-6956.pdf
[15] Riak http://basho.com/distributed-data-types-riak-2-0/

心是分离业务调用链和错误处理链，调用链传递正常业务数据，而错误链负责传递和处理错误消息。错误发生时，并不把错误抛给调用者，而是封装成显式消息发给它专属的 Supervisor，如图 5 所示，如果第一级 Supervisor 处理不了，则发给上一级 Supervisor 做处理，级联下去形成一个错误处理链条，不同的链条还可以合并成为一颗树，称为 Supervision 树。有了异步错误链，错误就可以按照受控的方式传播并被集中处理，而不需要在程序中处处插入错误处理代码。

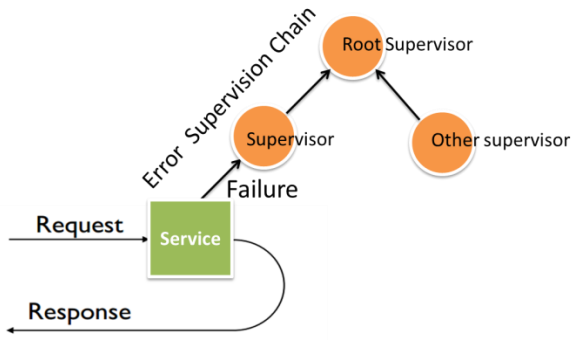

图 5 Async Error Supervision Chain

超时闹钟模式

超时闹钟模式是为了解决请求超时的问题。比如 A 通过 UDP 往 B 发消息，但 B 也许永远没收到这条消息，也无法回复。使用超时闹钟模式[图 6]后，A 在给 B 发正常请求消息的同时，也给自己设个闹钟，如果没收到 B 的回应，则闹钟会显式给自己发一条超时消息。这样，这样，B 没回应这个异常本身也被显式的定义成了消息。

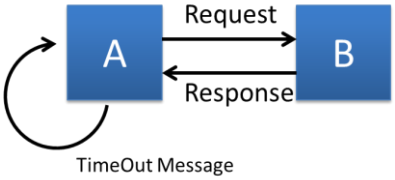

图 6 Timeout Timer

替身模式

如果一个服务很复杂，但这个服务同时又很重要，不能失败怎么办？这个时候可以用替身模式，危险的事情要交给替身去做。对每一件危险任务都可以创建一个替身，如果替身殉职了，可以原地复活，本尊则去监督替身有没有完成工作。危险的任务包括但不限于 1）状态多逻辑复杂的代码 2) 频繁修改未经充分测试的代码 3)用户上传的自定义的代码 4) IO 网络操作等可能失败的代码 5)其他可能抛出未预期异常的代码。通过使用替身模式，我们可以把错误封禁在临时替身这个“错误内核”里，而本尊则专司简单的，稳定的，不易出错的工作，以及管理“替身”们。通过这种隔离设计，可以提高整个系统的健壮性。

有这些设计模式，我们就可以确保错误可以得到隔离和控制，每条错误都被显式定义、显式传递、和显式处理。

Reactive 架构的开发工具

Reactive 应用最显著的技术特点就是 Message-Driven。过去 5 年，Reactive 开发工具中要属

Typesafe 的 Akka 最具光彩。Akka 在 Scala 语言上实现了 1973 年 Carl Hewitt 提出的 Actor 模型。Actor 是一种细粒度的并发编程单位，用起来和线程类似，但比线程的粒度要小得多，一个 JVM 里面可以启动数百万个 Actor。然后 Actor 之间是完全隔离的，不共享任何状态数据，如果要交换信息，只能像发短信一样发条消息给对方。Actor 还是位置透明的，即不论 Actor 是部署在单机上还是在分布式环境中，Actor 之间都能以一致性的方式交互，和 Actor 所在的物理位置无关。有了位置透明，一个应用不论运行在单个 JVM 里面或在分布式集群中，代码逻辑都是一样的，这让基于 Actor 实现的程序天生能 scale up 和 Scale out。

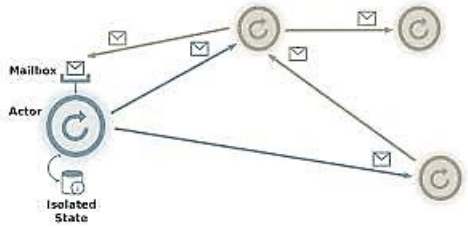

图 7 Akka Actor

Actor 的行为类似一个状态机，如图 7 所示，首先接收输入消息，处理消息然后改变自己的状态，最后发送消息给其他的 Actor。结构上 Actor 由三部分构成，状态（State），收件箱(Inbox)，和消息处理函数(Message Handler)。

1. **状态：**Actor 的状态是私有的，只对自己可见，对外完全隔离。一个 Actor 看不到另一个 Actor 的任何状态。
2. **收件箱：**Actor 之间的通信的唯一通道是消息，消息被保存到收件箱里。消息发送是异步非阻塞的。当某个 Actor 的收件箱有消息时，系统会调度该 Actor 的消息处理函数到线程池中运行，处理新消息。如果收件箱没有消息，则 Actor 不占用任何 CPU 资源。
3. **消息处理函数:** 消息处理函数由用户提供，用来处理收件箱的消息。Akka 的消息处理函数是纯函数式的，可以通过 or、and 等方式组合使用。

Actor 有良好的容错设计。如果一个 Actor 崩溃了，“崩溃”这个错误本身会被封装成显式的消息，然后发送给该 Actor 的 supervisor，由 Supervisor Actor 决定如何处理错误。Supervisor 可以选择重启 Actor，恢复 Actor，或上报给上游的 Supervisor，直到该错误得到正确处理。

除了隔离和容错，Akka Actor 在实践中还有非常优异的性能，一个 JVM 里面可以启动几百万个 Actor；分布式环境中， 2013 年的一组测试[16]显示，Akka Actor 可以 scale out 到 2400 个 CPU core 上。

Reactive 架构的应用实例 – GearPump 流处理系统

GearPump 是 Intel 开源的基于 Akka 开发的流处理平台，是一个完全按照 Reactive 思想设计的大数据产品。

[16] 2400 Akka Nodes http://typesafe.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine

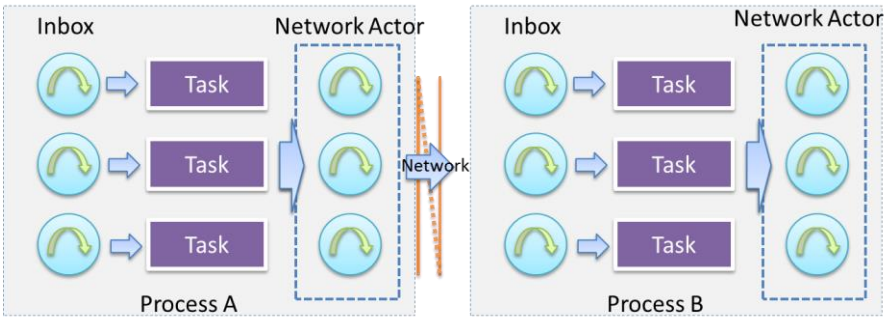

图 8 Gearpump Task Parallelism

Gearpump 是完完全全以消息驱动为核心的流处理引擎，设计时大量采用了前文提到的各种设计模式。

如图 8 所示，每一个 Task 都有一个 Inbox， Inbox 是一个**消息队列**。整个系统完全由**消息驱动**，当 Inbox 有消息时，Task 开始运算，生成的消息送给输出 Network Actor，再经网络送给下级 Task 的 Inbox，推动下级 Task 运算，相互间的数据传输按照**滑动窗做流速控制**。Task 之间是完全隔离的，不共享任何状态，完全**无锁设计**。应用的多个 Task 可以在一个进程内运行，也可以在不同机器上运行，是**位置透明**的，很容易实现 Scale out 和 Scale up。

为了实现高容错性和可管理性，GearPump 实现了一套清晰的 Supervision 架构，如图 9 所示。

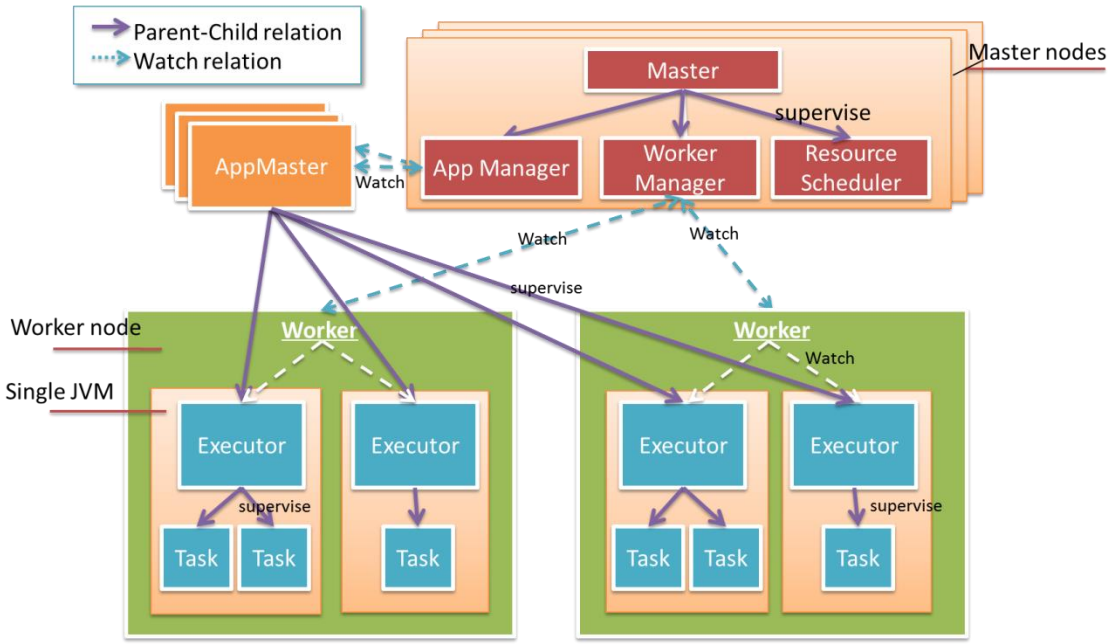

图 9 Actor Supervision Hierarchy

图中的每一个框都是一个 Akka Actor，它们通过 Supervision 和 Watch 关系编织在一起。Actor 间的相互通信都遵循**一问一答模式**和**超时闹钟模式**，确保整个系统是 Responsive 的。Master 和 Worker 是集群相关的 Actor，用于管理集群资源，Master 管理整个集群的资源，worker 管理单个机器的资源。AppMaster，Executor，Task 是应用相关的 Actor。 集群部分，Master 中的危险工作通过**替身模式**交给了 AppManager, Scheduler 等子 Actor，这样 Master 本身可以

更加健壮。应用部分，Task，Executor 和 AppMaster 连成一条**异步错误链**，通过增加中间层 Executor，我们可以做更细粒度的错误处理。我们还支持 Master HA。多个 Master 节点形成一个 Akka Cluster 集群，Master 间通过 Gossip 协议交换数据，用**最终一致性模式**的 **CRDT** 数据类型来保存共享的状态，一个 Master 挂掉，其他 Master 能从 CRDT 中恢复数据。

令人惊喜的是，通过采用 Reactive 架构的设计方法，GearPump 有非常好的性能。在四个节点，用 SOL[17] Benchmark，我们可以达到 1100 万消息/秒(100 字节每消息)，每条消息处理的平均延时只有 17 毫秒[图 10]。

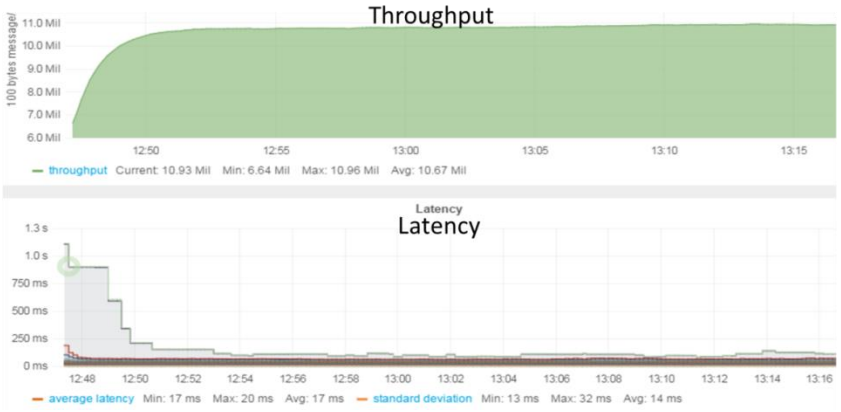

图 10 Gearpump performance

总结

Reactive 技术虽然不是新出现的，但在大数据和物联网的时代具有深远意义。Reactive 架构的 Message-Driven、Elastic、Resilient、Responsive 特性非常适合开发分布式应用。现在已经有一些非常成功的语言和工具，比如 Scala、Akka 等，设计上转向 Reactive 架构正当其时。

参考资料：

http://www.infoq.com/interviews/cesarini-klang-reactive-manifesto
http://www.infoq.com/presentations/reactive-principles
https://typesafe.com/company/casestudies

[17] https://github.com/yahoo/storm-perf-test