20 共识算法：一次性说清楚 Paxos、Raft 等算法的区别

现在，我们进入了分布式系统的最后一讲：共识算法。前面我们学习了各种分布式的技术，你可以和我一起回忆一下，其中我们讨论了失败模型、失败检测、领导选举和一致性模型。虽然这些技术可以被单独使用，但我们还是希望用一个技术栈就能实现上述全部功能，如果这样，将会是非常美妙的。于是，整个分布式数据库，乃至分布式领域的研究人员经过多年的努力，终于在这个问题上有所突破—共识算法由此诞生。

虽然共识算法是分布式系统理论的精华，但是通过之前的学习，其实你已经知道共识算法包含的内容了。它首先是要解决分布式系统比较棘手的失败问题，通过内置的失败检测机制可以发现失败节点、领导选举机制保证数据高效处理、一致性模式保证了消息的一致性。

这一讲，我会为你介绍几种常用的共识算法的特色。我不会深入到每种算法的详细执行过程，因为这些过程抽象且对使用没有特别的帮助。这一讲我的目的是从更高的维度为你解释这些算法，希望给你形象的记忆，并帮助你能够学以致用。至于算法实现细节，感兴趣的话你可以自行学习。

在介绍共识协议之前，我们要来聊聊它的三个属性。

1. 正确性（Validity）：诚实节点最终达成共识的值必须是来自诚实节点提议的值。
2. 一致性（Agreement）：所有的诚实节点都必须就相同的值达成共识。
3. 终止性（Termination）：诚实的节点必须最终就某个值达成共识。

你会发现共识算法中需要有“诚实”节点，它的概念是节点不能产生失败模型所描述的“任意失败”，或是“拜占庭失败”。因为数据库节点一般会满足这种假设，所以我们下面讨论的算法可以认为所有节点都是诚实的。

以上属性可以换个说法，实际上就是“15 | 领导选举：如何在分布式系统内安全地协调操作”介绍的安全性（Safety）和活跃性（Liveness），其中正确性（Validity）和一致性（Agreement）决定了安全性（Safety），而终止性（Termination）就是活跃性（Liveness）。让我们复习一下这两个特性。

1. 安全性（Safety）：在故障发生时，共识系统不能产生错误的结果。
2. 活跃性（Liveness）：系统能持续产生提交，也就是不会永远处于一个中间状态无法继续。

基于以上的特性，我们开始聊聊目前常见的共识算法。

原子广播与 ZAB

广播协议是一类将数据从一个节点同步到多个节点的协议。我在“17 | 数据可靠传播：反熵理论如何帮助数据库可靠工作”介绍过最终一致性系统通过各种反熵手段来保证数据的一致性传播，特别是其中的 Gossip 协议可以保障大规模的数据同步，而 Gossip 在正常情况下就是采用广播模式传播数据的。

以上的广播过程产生了一个问题，那就是这个协调节点是明显的单点，它的可靠性至关重要。要保障其可靠，首先要解决的问题是需要检查这个节点的健康状态。我们可以通过各种健康检查方式去发现其健康情况。

如果它失败了，会造成消息传播到一部分节点中，而另外一部分节点却没有这一份消息，这就违背了“一致性”。那么应该怎解决这个问题呢?

一个简单的算法就是使用“漫灌”机制，这种机制是一旦一个消息被广播到一个节点，该节点就有义务把该消息广播到其他未收到数据节点的义务。这就像水田灌溉一样，最终整个系统都收到了这份数据。

当然以上的模式有个明显的缺点，就是会产生N2的消息。其中 N 是目前系统剩下的未同步消息的节点，所以我们的一个优化目标就是要减少消息的总数量。

虽然广播可以可靠传递数据，但通过一致性的学习我们知道：需要保证各个节点接收到消息的顺序，才能实现较为严格的一致性。所以我们这里定义一个原子广播协议来满足。

1. 原子性：所有参与节点都收到并传播该消息；或相反，都不传播该消息。

2. 顺序性：所有参与节点传播消息的顺序都是一致的。

满足以上条件的协议我们称为原子广播协议，现在让我来介绍最为常见的原子广播协议：Zookeeper Atomic Broadcast（ZAB）。

ZAB

ZAB 协议由于 Zookeeper 的广泛使用变得非常流行。它是一种原子广播协议，可以保证消息顺序的传递，且消息广播时的原子性保障了消息的一致性。

ZAB 协议中，节点的角色有两种。

1. 领导节点。领导是一个临时角色，它是有任期的。这么做的目的是保证领导角色的活性。领导节点控制着算法执行的过程，广播消息并保证消息是按顺序传播的。读写操作都要经过它，从而保证操作的都是最新的数据。如果一个客户端连接的不是领导节点，它发送的消息也会转发到领导节点中。
2. 跟随节点。主要作用是接受领导发送的消息，并检测领导的健康状态。

既然需要有领导节点产生，我们就需要领导选举算法。这里我们要明确两个 ID：数据 ID 与节点 ID。前者可以看作消息的时间戳，后者是节点的优先级。选举的原则是：在同一任职周期内，节点的数据 ID 越大，表示该节点的数据越新，数据 ID 最大的节点优先被投票。所有节点的数据 ID 都相同，则节点 ID 最大的节点优先被投票。当一个节点的得票数超过节点半数，则该节点成为主节点。

一旦领导节点选举出来，它就需要做两件事。

1. 声明任期。领导节点通知所有的跟随节点当前的最新任期；而后由跟随节点确认当前任期是最新的任期，从而同步所有节点的状态。通过该过程，老任期的消息就不会被跟随节点所接受了。
2. 同步状态。这一步很关键，首先领导节点会通知所有跟随节点自己的领导身份，而后跟随节点不会再选举自己为领导了；然后领导节点会同步集群内的消息历史，保证最新的消息在所有节点中同步。因为新选举的领导节点很可能并没有最新被接受的数据，因此同步历史数据操作是很有必要的。

经过以上的初始化动作后，领导节点就可以正常接受消息，进行消息排序而后广播消息了。在广播消息的时候，需要 Quorum（集群中大多数的节点）的节点返回已经接受的消息才认为消息被正确广播了。同时为了保证顺序，需要前一个消息正常广播，后一个消息才能进行广播。

领导节点与跟随节点使用心跳算法检测彼此的健康情况。如果领导节点发现自己与 Quorum 节点们失去联系，比如网络分区，此时领导节点会主动下台，开始新一轮选举。同理，当跟随节点检测到领导节点延迟过大，也会触发新一轮选举。

ZAB 选举的优势是，如果领导节点一直健康，即使当前任期过期，选举后原领导节点还会承担领导角色，而不会触发领导节点切换，这保证了该算法的稳定。另外，它的节点恢复比较高效，通过比较各个节点的消息 ID，找到最大的消息 ID，就可以从上面恢复最新的数据了。最后，它的消息广播可以理解为没有投票过程的两阶段提交，只需要两轮消息就可以将消息广播出去。

那么原子广播协议与本讲重点介绍的共识算法是什么关系呢？这里我先留下一个“暗扣”，先介绍一下典型的共识算法 Paxos，而后再说明它们之间的关系。

Paxos

所谓的 Paxos 算法，是为了解决来自客户端的值被发送到集群中的任意一点，而后集群中的所有节点为该值达成共识的一种协调算法。同时这个值伴随一个版本号，可以保证消息是有顺序的，该顺序在集群中任何一点都是一致的。

基本的 Paxos 算法非常简单，它由三个角色组成。

1. Proposer: Proposer 可以有多个，Proposer 提出议案（value）。所谓 value，可以是任何操作，比如“设置某个变量的值为 value”。不同的 Proposer 可以提出不同的 value。但对同一轮 Paxos 过程，最多只有一个 value 被批准。
2. Acceptor: Acceptor 有 N 个，Proposer 提出的 value 必须获得 Quorum 的 Acceptor 批准后才能通过。Acceptor 之间完全对等独立。
3. Learner: 上面提到只要 Quorum 的 Accpetor 通过即可获得通过，那么 Learner 角色的目的就是把通过的确定性取值同步给其他未确定的 Acceptor。

这三个角色其实已经描述了一个值被提交的整个过程。其实基本的 Paxos 只是理论模型，因为在真实场景下，我们需要处理许多连续的值，并且这些值都是并发的。如果完全执行上面描述的过程，那性能消耗是任何生产系统都无法承受的，因此我们一般使用的是 Multi-Paxos。

Multi-Paxos 可以并发执行多个 Paxos 协议，它优化的重点是把 Propose 阶段进行了合并，这就引入了一个 Leader 的角色，也就是领导节点。而后读写全部由 Leader 处理，同时这里与 ZAB 类似，Leader 也有任期的概念，Leader 与其他节点之间也用心跳进行互相探活。是不是感觉有那个味道了？后面我就会比较两者的异同。

另外 Multi-Paxos 引入了两个重要的概念：replicated log 和 state snapshot。

1. replicated log：值被提交后写入到日志中。这种日志结构除了提供持久化存储外，更重要的是保证了消息保存的顺序性。而 Paxos 算法的目标是保证每个节点该日志内容的强一致性。
2. state snapshot：由于日志结构保存了所有值，随着时间推移，日志会越来越大。故算法实现了一种状态快照，可以保存最新的日志消息。当快照生成后，我们就可以安全删除快照之前的日志了。

熟悉 Raft 的同学会发现，上面的结构其实已经与 Raft 很接近了。在讨论完原子广播与共识之后 ，我们会接着介绍 Raft。

原子广播与共识

就像我开篇所说的，本讲不是介绍算法细节的，而是重点关注它们为什么是今天这个样子。从上面的粗略介绍中，我们已经发现：ZAB 其实与 Multi-Paxos 是非常类似的。本质上，它们都需要大部分节点“同意”一个值，并都有 Leader 节点，且 Leader 都是临时的。真是越说越相似，但本质上它们却又是不同的。

简单来说，ZAB 来源于主备复制场景，就是我们之前介绍的复制技术；而共识算法是状态机复制系统。

所谓状态机复制系统，是指集群中每个节点都是一个状态机，如果有一组客户端并发在系统中的不同状态机上提交不同的值，该系统保证每个状态机都可以保证执行相同顺序的客户端请求。可以看到请求一旦被提交，其顺序是有保障的。但是未提交之前，顺序是由 Leader 决定的，且这个顺序可以是任意的。一旦 Leader 被重选，新的 Leader 可以任意排序未提交的值。

而 ZAB 这种广播协议来自主备复制，强调的是消息的顺序是 Leader 产生的，并被 Follower 严格执行，其中没有协调的关系。更重要的区别是，Leader 重选后，新 Leader 依然会按照原 Leader 的排序来广播数据，而不会自己去排序。

因此可以说 ZAB 可以实现严格的线性一致性。而 Multi-Paxos 由于只是并发写，所以也没有所谓的线性一致，而是一种顺序一致结构，也就是数据被提交时才能确定顺序。而不是如 ZAB 那样有 Leader 首先分配了顺序，该顺序与数据提交的先后顺序保持了一致。关于线性一致和顺序一致，请参考“05 | 一致性与 CAP 模型：为什么需要分布式一致性？”

由于共识算法如 Paxos 为了效率的原因引入了 Leader。在正常情况下，两者差异不是很大，而差异主要在选举 Leader 的流程上。

那么学习完 ZAB 和 Multi-Paxos 后，我将要介绍这一讲的主角 Raft 算法，它是目前分布式数据库领域最重要的算法。

Raft 的特色

Raft 可以看成是 Multi-Paxos 的改进算法，因为其作者曾在斯坦福大学做过关于 Raft 与 Multi-Paxos 的比较演讲，因此我们可以将它们看作一类算法。

Raft 算法可以说是目前最成功的分布式共识算法，包括 TiDB、FaunaDB、Redis 等都使用了这种技术。原因是 Multi-Paxos 没有具体的实现细节，虽然它给了开发者想象空间，但共识算法一般居于核心位置，一旦存在潜在问题必然带给系统灾难性的后果。而 Raft 算法给出了大量的实现细节，且处理方式相比于 Multi-Paxos 有两点优势。

1. 发送的请求的是连续的，也就是说 Raft 的写日志操作必须是连续的；而 Multi-Paxos 可以并发修改日志，这也体现了“Multi”的特点。
2. 选主必须是最新、最全的日志节点才可以当选，这一点与 ZAB 算法有相同的原则；而 Multi-Paxo 是随机的。因此 Raft 可以看成是简化版本的 Multi-Paxos，正是这个简化，造就了 Raft 的流行。

Multi-Paxos 随机性使得没有一个节点有完整的最新的数据，因此其恢复流程非常复杂，需要同步节点间的历史记录；而 Raft 可以很容易地找到最新节点，从而加快恢复速度。当然乱序提交和日志的不连续也有好处，那就是写入并发性能会大大提高，从而提高吞吐量。所以这两个特性并不是缺点，而是权衡利弊的结果。当然 TiKV 在使用 Raft 的时候采用了多 RaftGroup 的模式，提高了单 Raft 结构的并发度，这可以被看作是向 Multi-Paxos 的一种借鉴。

同时 Raft 和 Multi-Paxos 都使用了任期形式的 Leader。好处是性能很高，缺点是在切主的时候会拒绝服务，造成可用性下降。因此一般我们认为共识服务是 CP 类服务（CAP 理论）。但是有些团队为了提高可用性 ，转而采用基础的 Paxos 算法，比如微信的 PaxosStore 都是用了每轮一个单独的 Paxos 这种策略。

以上两点改进使 Raft 更好地落地，可以说目前最新数据库几乎都在使用该算法。想了解算法更多细节，请参考 https://raft.github.io/。你从中不仅能学习到算法细节，更重要的是可以看到很多已经完成的实现，结合代码学习能为你带来更深刻的印象。

总结

共识算法是一个比较大的话题。本讲聚焦于常见的三种共识类算法，集中展示其最核心的功能。我通过比较它们之间的异同，来加深你对它们特性的记忆。

共识算法又是现代分布式数据库的核心组件，好在其 API 较为易懂，且目前有比较成熟的实现，所以我认为算法细节并不是本讲的重点。理解它们为什么如此，才能帮助我们理解数据库的选择依据。

到此，我们学习完了这个模块的所有知识点。下一讲我将会带领你复习这一模块的内容，同时通过几个案例来展示典型分布式数据库特性与咱们所学的知识点之间的关系，到时候见。

上一页 下一页