



下载APP

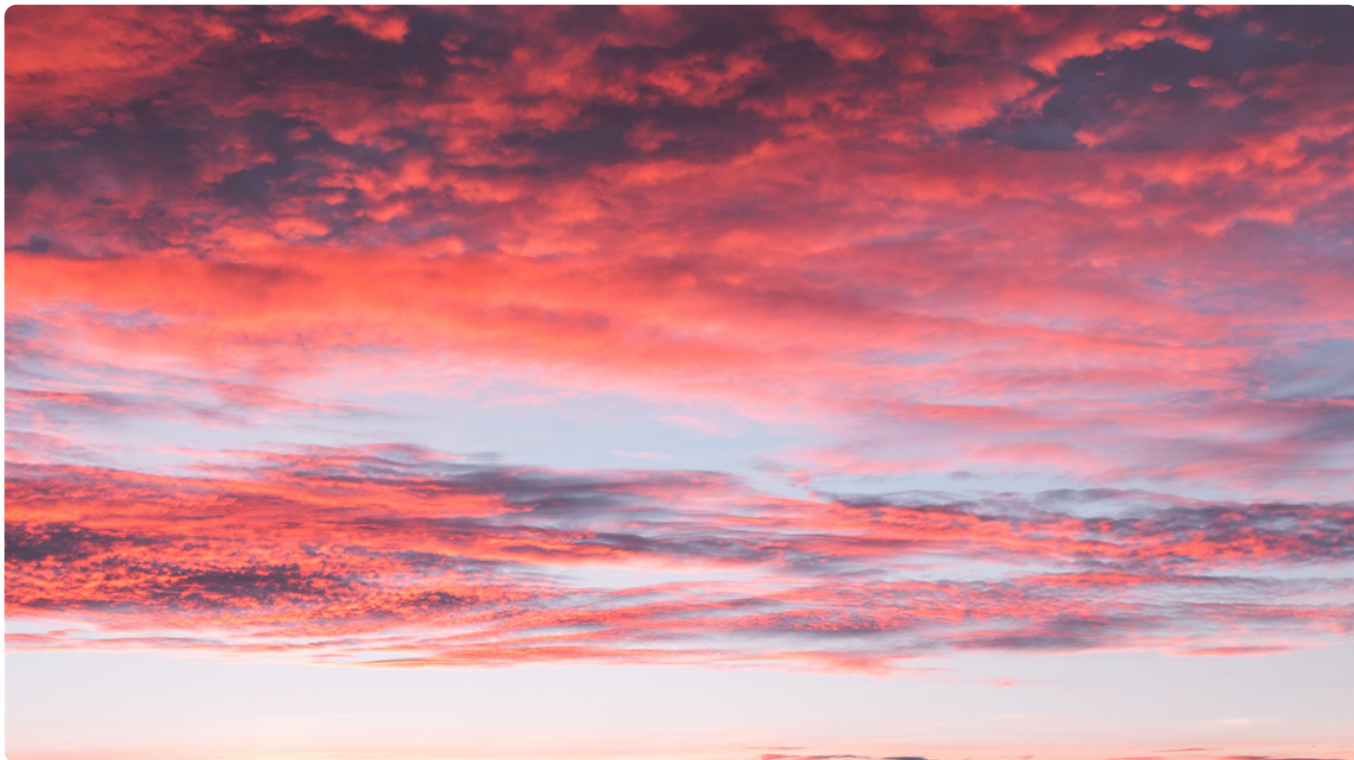


## 04 | Raft协议：etcd如何实现高可用、数据强一致的？

2021-01-27 唐聪

etcd实战课

[进入课程 >](#)



讲述：王超凡


时长 21:18 大小 19.52M



你好，我是唐聪。

在前面的 etcd 读写流程学习中，我和你多次提到了 etcd 是基于 Raft 协议实现高可用、数据强一致性的。

那么 etcd 是如何基于 Raft 来实现高可用、数据强一致性的呢？

这节课我们就以上一节中的 hello 写请求为案例，深入分析 etcd 在遇到 Leader 节点 crash 等异常后，Follower 节点如何快速感知到异常，并高效选举出新的 Leader，对， 提供高可用服务的。

同时, 我将通过一个日志复制整体流程图, 为你介绍 etcd 如何保障各节点数据一致性, 并介绍 Raft 算法为了确保数据一致性、完整性, 对 Leader 选举和日志复制所增加的一系列安全规则。希望通过这节课, 让你了解 etcd 在节点故障、网络分区等异常场景下是如何基于 Raft 算法实现高可用、数据强一致的。

## 如何避免单点故障

在介绍 Raft 算法之前, 我们首先了解下它的诞生背景, Raft 解决了分布式系统什么痛点呢?

首先我们回想下, 早期我们使用的数据存储服务, 它们往往是部署在单节点上的。但是单节点存在单点故障, 一宕机就整个服务不可用, 对业务影响非常大。

随后, 为了解决单点问题, 软件系统工程师引入了数据复制技术, 实现多副本。通过数据复制方案, 一方面我们可以提高服务可用性, 避免单点故障。另一方面, 多副本可以提升读吞吐量、甚至就近部署在业务所在的地理位置, 降低访问延迟。

## 多副本复制是如何实现的呢?

多副本常用的技术方案主要有主从复制和去中心化复制。主从复制, 又分为全同步复制、异步复制、半同步复制, 比如 MySQL/Redis 单机主备版就基于主从复制实现的。

**全同步复制**是指主收到一个写请求后, 必须等待全部从节点确认返回后, 才能返回给客户端成功。因此如果一个从节点故障, 整个系统就会不可用。这种方案为了保证多副本的一致性, 而牺牲了可用性, 一般使用不多。

**异步复制**是指主收到一个写请求后, 可及时返回给 client, 异步将请求转发给各个副本, 若还未将请求转发到副本前就故障了, 则可能导致数据丢失, 但是可用性是最高的。

**半同步复制**介于全同步复制、异步复制之间, 它是指主收到一个写请求后, 至少有一个副本接收数据后, 就可以返回给客户端成功, 在数据一致性、可用性上实现了平衡和取舍。

跟主从复制相反的就是**去中心化复制**, 它是指在一个  $n$  副本节点集群中, 任意节点都可接受写请求, 但一个成功的写入需要  $w$  个节点确认, 读取也必须查询至少  $r$  个节点。

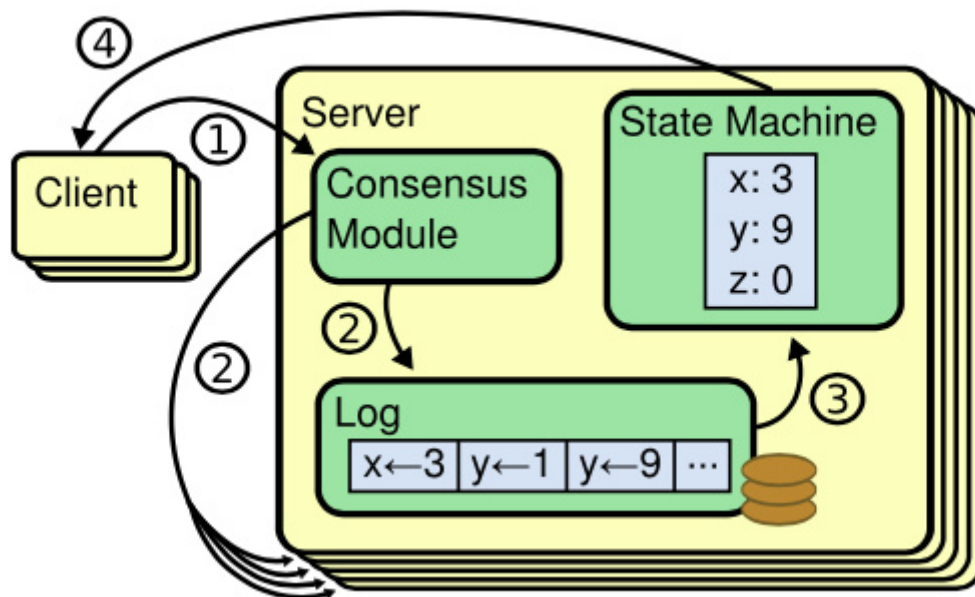
你可以根据实际业务场景对数据一致性的敏感度，设置合适  $w/r$  参数。比如你希望每次写入后，任意 client 都能读取到新值，如果  $n$  是 3 个副本，你可以将  $w$  和  $r$  设置为 2，这样当你读两个节点时候，必有一个节点含有最近写入的新值，这种读我们称之为法定票数读 (quorum read)。

AWS 的 Dynamo 系统就是基于去中心化的复制算法实现的。它的优点是节点角色都是平等的，降低运维复杂度，可用性更高。但是缺陷是去中心化复制，势必会导致各种写入冲突，业务需要关注冲突处理。

从以上分析中，为了解决单点故障，从而引入了多副本。但基于复制算法实现的数据库，为了保证服务可用性，大多数提供的是最终一致性，总而言之，不管是主从复制还是异步复制，都存在一定的缺陷。

### 如何解决以上复制算法的困境呢？

答案就是共识算法，它最早是基于复制状态机背景下提出来的。下图是复制状态机的结构（引用自 Raft paper），它由共识模块、日志模块、状态机组成。通过共识模块保证各个节点日志的一致性，然后各个节点基于同样的日志、顺序执行指令，最终各个复制状态机的结果实现一致。



共识算法的祖师爷是 Paxos，但是由于它过于复杂，难于理解，工程实践上也较难落地，导致在工程界落地较慢。standford 大学的 Diego 提出的 Raft 算法正是为了可理解性、易实现而诞生的，它通过问题分解，将复杂的共识问题拆分成三个子问题，分别是：

Leader 选举, Leader 故障后集群能快速选出新 Leader;

日志复制, 集群只有 Leader 能写入日志, Leader 负责复制日志到 Follower 节点, 并强制 Follower 节点与自己保持相同;

安全性, 一个任期内集群只能产生一个 Leader、已提交的日志条目在发生 Leader 选举时, 一定会存在更高任期的新 Leader 日志中、各个节点的状态机应用的任意位置的日志条目内容应一样等。

下面我以实际场景为案例, 分别和你深入讨论这三个子问题, 看看 Raft 是如何解决这三个问题, 以及在 etcd 中的应用实现。

## Leader 选举

当 etcd server 收到 client 发起的 put hello 写请求后, KV 模块会向 Raft 模块提交一个 put 提案, 我们知道只有集群 Leader 才能处理写提案, 如果此时集群中无 Leader, 整个请求就会超时。

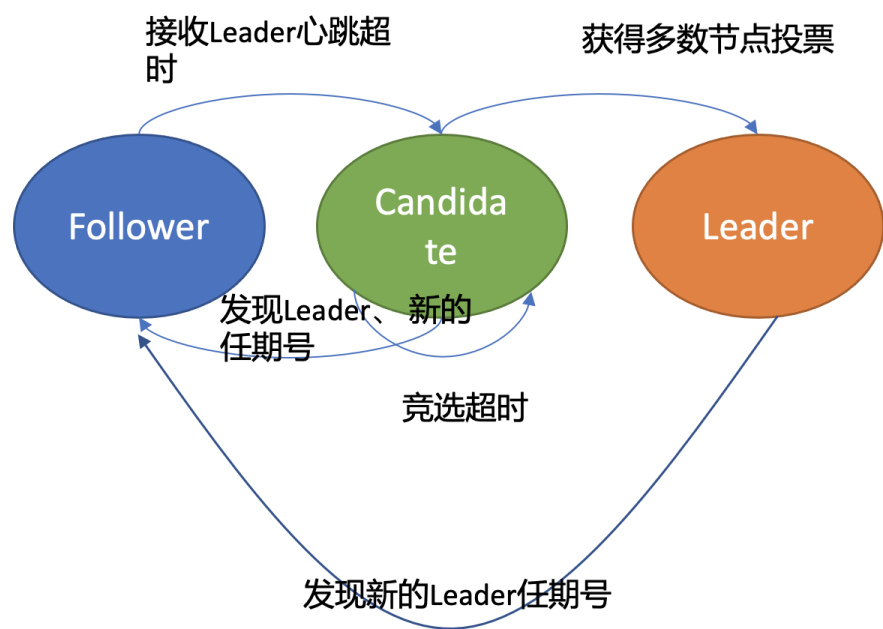
那么 Leader 是怎么诞生的呢? Leader crash 之后其他节点如何竞选呢?

首先在 Raft 协议中它定义了集群中的如下节点状态, 任何时刻, 每个节点肯定处于其中一个状态:

Follower, 跟随者, 同步从 Leader 收到的日志, etcd 启动的时候默认为此状态;

Candidate, 竞选者, 可以发起 Leader 选举;

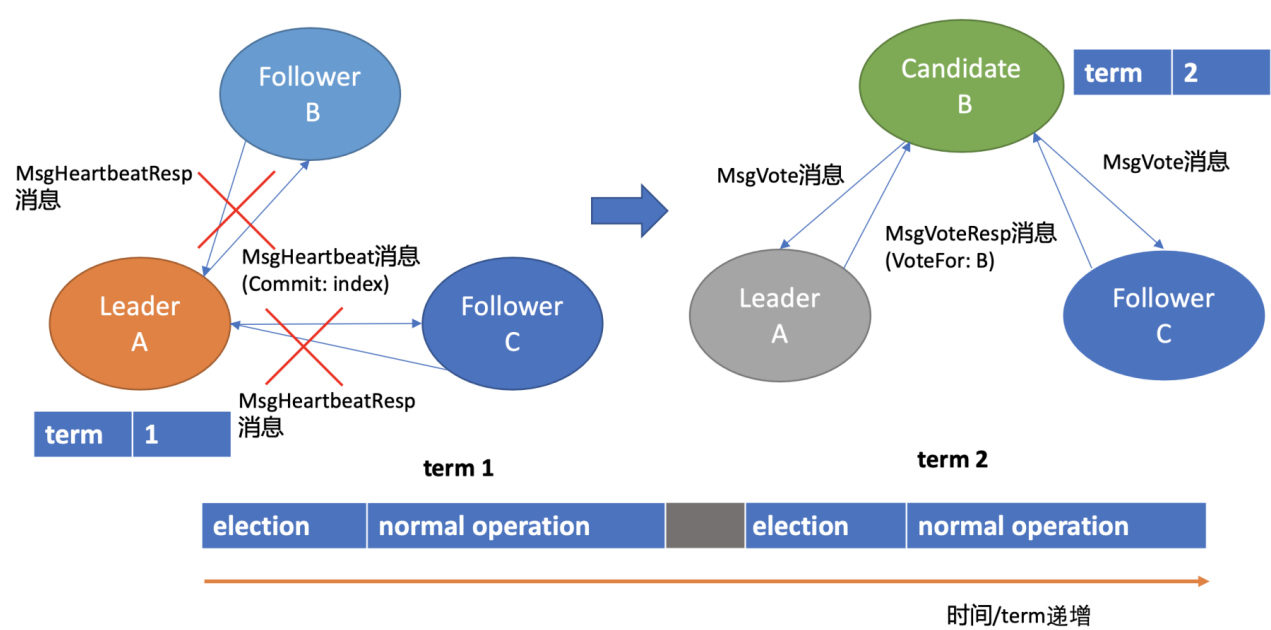
Leader, 集群领导者, 唯一性, 拥有同步日志的特权, 需定时广播心跳给 Follower 节点, 以维持领导者身份。



上图是节点状态变化关系图，当 Follower 节点接收 Leader 节点心跳消息超时后，它会转变成 Candidate 节点，并可发起竞选 Leader 投票，若获得集群多数节点的支持后，它就可转变成 Leader 节点。

下面我以 Leader crash 场景为案例，给你详细介绍一下 etcd Leader 选举原理。

假设集群总共 3 个节点，A 节点为 Leader，B、C 节点为 Follower。



如上 Leader 选举图左边部分所示, 正常情况下, Leader 节点会按照心跳间隔时间, 定时广播心跳消息 (MsgHeartbeat 消息) 给 Follower 节点, 以维持 Leader 身份。Follower 收到后回复心跳应答包消息 (MsgHeartbeatResp 消息) 给 Leader。

细心的你可能注意到上图中的 Leader 节点下方有一个任期号 (term), 它具有什么样的作用呢?

这是因为 Raft 将时间划分成一个个任期, 任期用连续的整数表示, 每个任期从一次选举开始, 赢得选举的节点在该任期内充当 Leader 的职责, 随着时间的消逝, 集群可能会发生新的选举, 任期号也会单调递增。

通过任期号, 可以比较各个节点的数据新旧、识别过期的 Leader 等, 它在 Raft 算法中充当逻辑时钟, 发挥着重要作用。

了解完正常情况下 Leader 维持身份的原理后, 我们再看异常情况下, 也就 Leader crash 后, etcd 是如何自愈的呢?

如上 Leader 选举图右边部分所示, 当 Leader 节点异常后, Follower 节点会接收 Leader 的心跳消息超时, 当超时时间大于竞选超时时间后, 它们会进入 Candidate 状态。

这里要提醒下你, etcd 默认心跳间隔时间 (heartbeat-interval) 是 100ms, 默认竞选超时时间 (election timeout) 是 1000ms, 你需要根据实际部署环境、业务场景适当调优, 否则就很可能频繁发生 Leader 选举切换, 导致服务稳定性下降, 后面我们实践篇会再详细介绍。

进入 Candidate 状态的节点, 会立即发起选举流程, 自增任期号, 投票给自己, 并向其他节点发送竞选 Leader 投票消息 (MsgVote)。

C 节点收到 Follower B 节点竞选 Leader 消息后, 这时候可能会出现如下两种情况:

第一种情况是 C 节点判断 B 节点的数据至少和自己一样新、B 节点任期号大于 C 当前任期号、并且 C 未投票给其他候选者, 就可投票给 B。这时 B 节点获得了集群多数节点支持, 于是成为了新的 Leader。

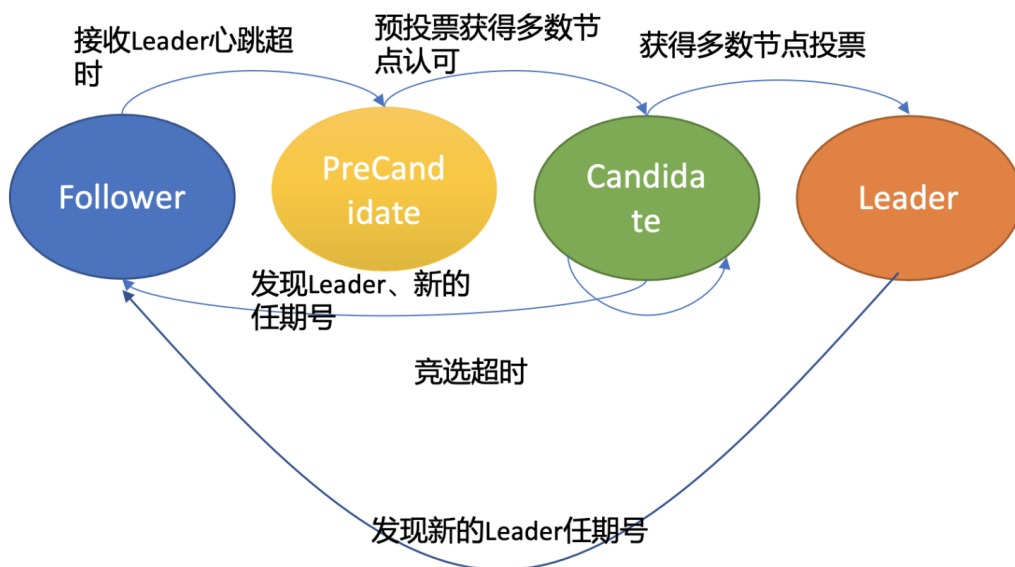
第二种情况是，恰好 C 也心跳超时超过竞选时间了，它也发起了选举，并投票给了自己，那么它将拒绝投票给 B，这时谁也无法获取集群多数派支持，只能等待竞选超时，开启新一轮选举。Raft 为了优化选票被瓜分导致选举失败的问题，引入了随机数，每个节点等待发起选举的时间点不一致，优雅的解决了潜在的竞选活锁，同时易于理解。

Leader 选出来后，它什么时候又会变成 Follower 状态呢？从上面的状态转换关系图中你可以看到，如果现有 Leader 发现了新的 Leader 任期号，那么它就需要转换到 Follower 节点。A 节点 crash 后，再次启动成为 Follower，假设因为网络问题无法连通 B、C 节点，这时候根据状态图，我们知道它将不停自增任期号，发起选举。等 A 节点网络异常恢复后，那么现有 Leader 收到了新的任期号，就会触发新一轮 Leader 选举，影响服务的可用性。

然而 A 节点的数据是远远落后 B、C 的，是无法获得集群 Leader 地位的，发起的选举无效且对集群稳定性有伤害。

那如何避免以上场景中的无效的选举呢？

在 etcd 3.4 中，etcd 引入了一个 PreVote 参数（默认 false），可以用来启用 PreCandidate 状态解决此问题，如下图所示。Follower 在转换成 Candidate 状态前，先进入 PreCandidate 状态，不自增任期号，发起预投票。若获得集群多数节点认可，确定有概率成为 Leader 才能进入 Candidate 状态，发起选举流程。





因 A 节点数据落后较多，预投票请求无法获得多数节点认可，因此它就不会进入 Candidate 状态，导致集群重新选举。

这就是 Raft Leader 选举核心原理，使用心跳机制维持 Leader 身份、触发 Leader 选举，etcd 基于它实现了高可用，只要集群一半以上节点存活、可相互通信，Leader 宕机后，就能快速选举出新的 Leader，继续对外提供服务。

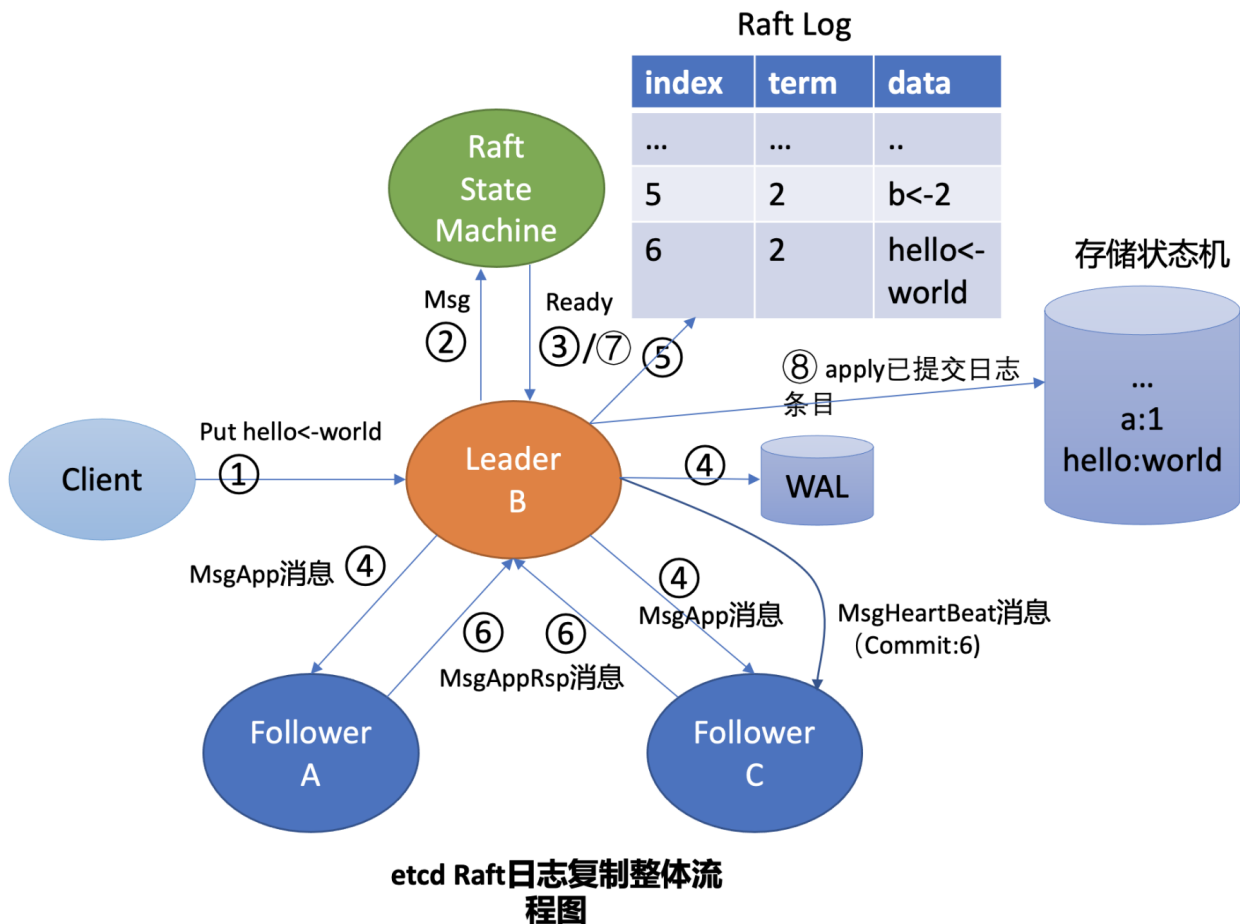
## 日志复制

假设在上面的 Leader 选举流程中，B 成为了新的 Leader，它收到 put 提案后，它是如何将日志同步给 Follower 节点的呢？什么时候它可以确定一个日志条目为已提交，通知 etcdserver 模块应用日志条目指令到状态机呢？

这就涉及到 Raft 日志复制原理，为了帮助你理解日志复制的原理，下面我给你画了一幅 Leader 收到 put 请求后，向 Follower 节点复制日志的整体流程图，简称流程图，在图中我用序号给你标识了核心流程。

我将结合流程图、后面的 Raft 的日志图和你简要分析 Leader B 收到 put hello 为 world 的请求后，是如何将此请求同步给其他 Follower 节点的。





首先 Leader 收到 client 的请求后，etcdserver 的 KV 模块会向 Raft 模块提交一个 put hello 为 world 提案消息（流程图中的序号 2 流程），它的消息类型是 MsgProp。

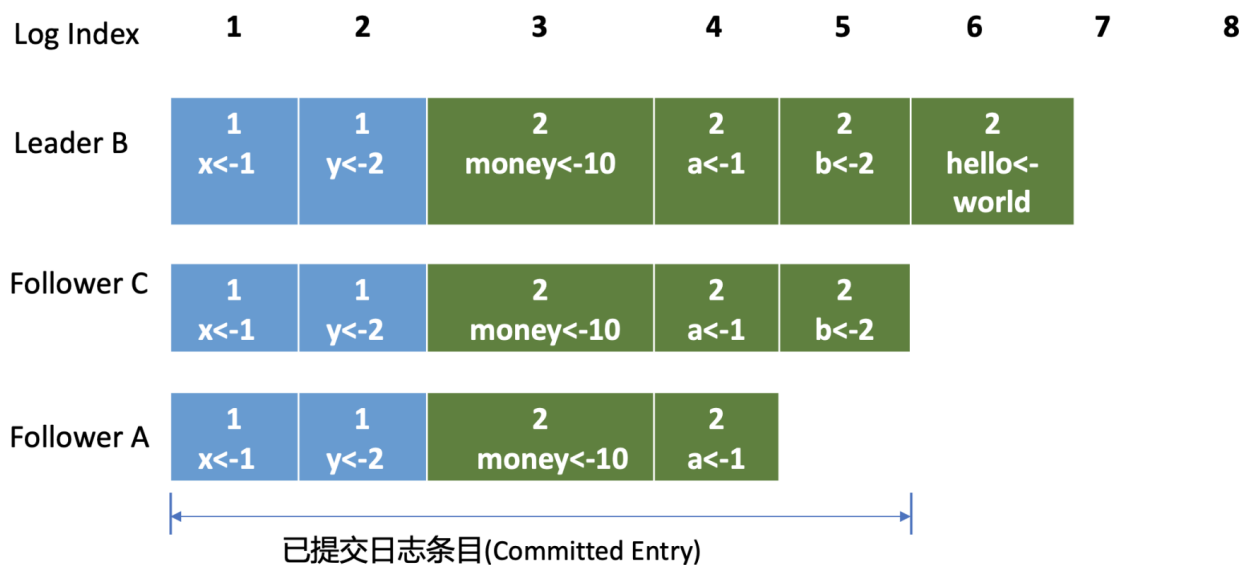
Leader 的 Raft 模块获取到 MsgProp 提案消息后，为此提案生成一个日志条目，追加到未持久化、不稳定的 Raft 日志中，随后会遍历集群 Follower 列表和进度信息，为每个 Follower 生成追加（MsgApp）类型的 RPC 消息，此消息中包含待复制给 Follower 的日志条目。

这里就出现两个疑问了。第一，Leader 是如何知道从哪个索引位置发送日志条目给 Follower，以及 Follower 已复制的日志最大索引是多少呢？第二，日志条目什么时候才会追加到稳定的 Raft 日志中呢？Raft 模块负责持久化吗？

首先我来给你介绍下什么是 Raft 日志。下图是 Raft 日志复制过程中的日志细节图，简称日志图 1。

在日志图中，最上方的是日志条目序号 / 索引，日志由有序号标识的一个个条目组成，每个日志条目内容保存了 Leader 任期号和提案内容。最开始的时候，A 节点是 Leader，任期号为 1，A 节点 crash 后，B 节点通过选举成为新的 Leader，任期号为 2。

日志图 1 描述的是 hello 日志条目未提交前的各节点 Raft 日志状态。



Raft日志图1(hello日志条目未提交前)

我们现在就可以来回答第一个疑问了。Leader 会维护两个核心字段来追踪各个 Follower 的进度信息，一个字段是 NextIndex，它表示 Leader 发送给 Follower 节点的下一个日志条目索引。一个字段是 MatchIndex，它表示 Follower 节点已复制的最大日志条目的索引，比如上面的日志图 1 中 C 节点的已复制最大日志条目索引为 5，A 节点为 4。

我们再看第二个疑问。etcd Raft 模块设计实现上抽象了网络、存储、日志等模块，它本身并不会进行网络、存储相关的操作，上层应用需结合自己业务场景选择内置的模块或自定义实现网络、存储、日志等模块。

上层应用通过 Raft 模块的输出接口（如 Ready 结构），获取到待持久化的日志条目和待发送给 Peer 节点的消息后（如上面的 MsgApp 日志消息），需持久化日志条目到自定义的 WAL 模块，通过自定义的网络模块将消息发送给 Peer 节点。

日志条目持久化到稳定存储中后，这时候你就可以将日志条目追加到稳定的 Raft 日志中。即便这个日志是内存存储，节点重启时也不会丢失任何日志条目，因为 WAL 模块已持久化此日志条目，可通过它重建 Raft 日志。

etcd Raft 模块提供了一个内置的内存存储（MemoryStorage）模块实现，etcd 使用的就是它，Raft 日志条目保存在内存中。网络模块并未提供内置的实现，etcd 基于 HTTP 协议

实现了 peer 节点间的网络通信，并根据消息类型，支持选择 pipeline、stream 等模式发送，显著提高了网络吞吐量、降低了延时。

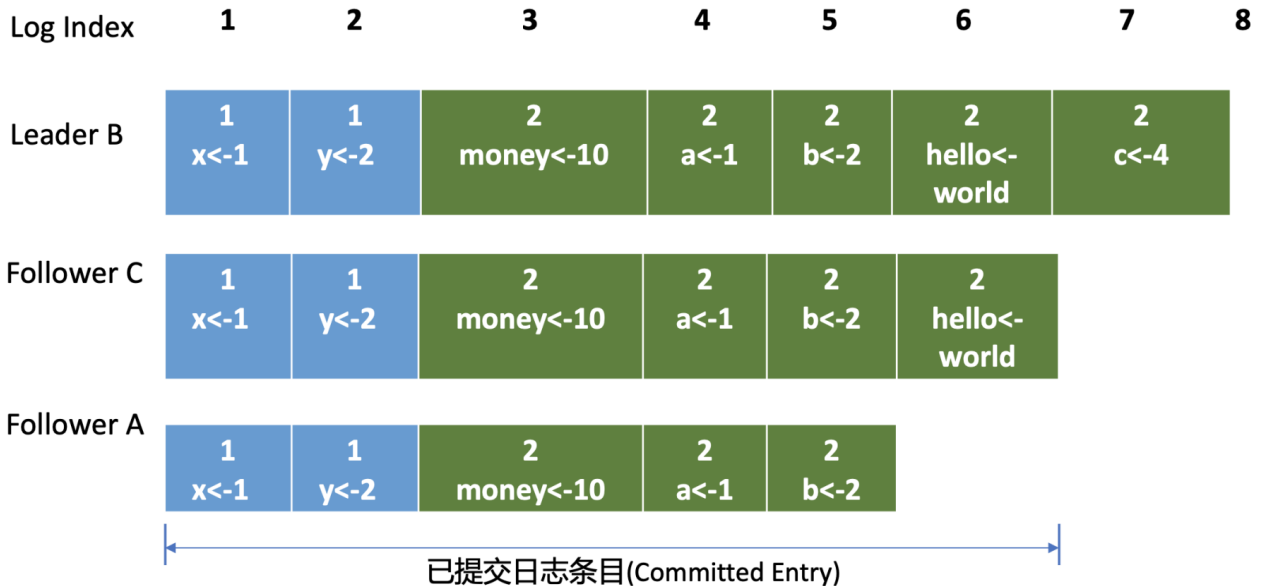
解答完以上两个疑问后，我们继续分析 etcd 是如何与 Raft 模块交互，获取待持久化的日志条目和发送给 peer 节点的消息。

正如刚刚讲到的，Raft 模块输入是 Msg 消息，输出是一个 Ready 结构，它包含待持久化的日志条目、发送给 peer 节点的消息、已提交的日志条目内容、线性查询结果等 Raft 输出核心信息。

etcdserver 模块通过 channel 从 Raft 模块获取到 Ready 结构后（流程图中的序号 3 流程），因 B 节点是 Leader，它首先会通过基于 HTTP 协议的网络模块将追加日志条目消息 (MsgApp) 广播给 Follower，并同时 will 待持久化的日志条目持久化到 WAL 文件中（流程图中的序号 4 流程），最后将日志条目追加到稳定的 Raft 日志存储中（流程图中的序号 5 流程）。

各个 Follower 收到追加日志条目 (MsgApp) 消息，并通过安全检查后，它会持久化消息到 WAL 日志中，并将消息追加到 Raft 日志存储，随后会向 Leader 回复一个应答追加日志条目 (MsgAppResp) 的消息，告知 Leader 当前已复制的日志最大索引（流程图中的序号 6 流程）。

Leader 收到应答追加日志条目 (MsgAppResp) 消息后，会将 Follower 回复的已复制日志最大索引更新到跟踪 Follower 进展的 Match Index 字段，如下面的日志图 2 中的 Follower C MatchIndex 为 6，Follower A 为 5，日志图 2 描述的是 hello 日志条目提交后的各节点 Raft 日志状态。



Raft日志图2(hello日志条目提交后)

最后 Leader 根据 Follower 的 MatchIndex 信息，计算出一个位置，如果这个位置已经被一半以上节点持久化，那么这个位置之前的日志条目都可以被标记为已提交。

在我们这个案例中日志图 2 里 6 号索引位置之前的日志条目已被多数节点复制，那么他们状态都可被设置为已提交。Leader 可通过在发送心跳消息 (MsgHeartbeat) 给 Follower 节点时，告知它已经提交的日志索引位置。

最后各个节点的 etcdserver 模块，可通过 channel 从 Raft 模块获取到已提交的日志条目（流程图中的序号 7 流程），应用日志条目内容到存储状态机（流程图中的序号 8 流程），返回结果给 client。

通过以上流程，Leader 就完成了同步日志条目给 Follower 的任务，一个日志条目被确定为已提交的前提是，它需要被 Leader 同步到一半以上节点上。以上就是 etcd Raft 日志复制的核心原理。

## 安全性

介绍完 Leader 选举和日志复制后，最后我们再来看看 Raft 是如何保证安全性的。

如果在上面的日志图 2 中，Leader B 在应用日志指令 put hello 为 world 到状态机，并返回给 client 成功后，突然 crash 了，那么 Follower A 和 C 是否都有资格选举成为 Leader 呢？

从日志图 2 中我们可以看到, 如果 A 成为了 Leader 那么就会导致数据丢失, 因为它并未含有刚刚 client 已经写入成功的 put hello 为 world 指令。

Raft 算法如何确保面对这类问题时不丢数据和各节点数据一致性呢?

这就是 Raft 的第三个子问题需要解决的。Raft 通过给选举和日志复制增加一系列规则, 来实现 Raft 算法的安全性。

## 选举规则

当节点收到选举投票的时候, 需检查候选者的最后一条日志中的任期号, 若小于自己则拒绝投票。如果任期号相同, 日志却比自己短, 也拒绝为其投票。

比如在日志图 2 中, Follower A 和 C 任期号相同, 但是 Follower C 的数据比 Follower A 要长, 那么在选举的时候, Follower C 将拒绝投票给 A, 因为它的不是最新的。

同时, 对于一个给定的任期号, 最多只会有一个 leader 被选举出来, leader 的诞生需获得集群一半以上的节点支持。每个节点在同一个任期内只能为一个节点投票, 节点需要将投票信息持久化, 防止异常重启后再投票给其他节点。

通过以上规则就可防止日志图 2 中的 Follower A 节点成为 Leader。

## 日志复制规则

在日志图 2 中, Leader B 返回给 client 成功后若突然 crash 了, 此时可能还并未将 6 号日志条目已提交的消息通知到 Follower A 和 C, 那么如何确保 6 号日志条目不被新 Leader 删除呢? 同时在 etcd 集群运行过程中, Leader 节点若频繁发生 crash 后, 可能会导致 Follower 节点与 Leader 节点日志条目冲突, 如何保证各个节点的同 Raft 日志位置含有同样的日志条目?

以上各类异常场景的安全性是通过 Raft 算法中的 Leader 完全特性和只附加原则、日志匹配等安全机制来保证的。

**Leader 完全特性**是指如果某个日志条目在某个任期号中已经被提交, 那么这个条目必然出现在更大任期号的所有 Leader 中。

Leader 只能追加日志条目, 不能删除已持久化的日志条目 (**只附加原则**), 因此 Follower C 成为新 Leader 后, 会将前任的 6 号日志条目复制到 A 节点。

为了保证各个节点日志一致性, Raft 算法在追加日志的时候, 引入了一致性检查。Leader 在发送追加日志 RPC 消息时, 会把新的日志条目紧接着之前的条目的索引位置和任期号包含在里面。Follower 节点会检查相同索引位置的任期号是否与 Leader 一致, 一致才能追加, 这就是**日志匹配特性**。它本质上是一种归纳法, 一开始日志空满足匹配特性, 随后每增加一个日志条目时, 都要求上一个日志条目信息与 Leader 一致, 那么最终整个日志集肯定是一致的。

通过以上的 Leader 选举限制、Leader 完全特性、只附加原则、日志匹配等安全特性, Raft 就实现了一个可严格通过数学反证法、归纳法证明的高可用、一致性算法, 为 etcd 的安全性保驾护航。

## 小结

最后我们来小结下今天的内容。我从如何避免单点故障说起, 给你介绍了分布式系统中实现多副本技术的一系列方案, 从主从复制到去中心化复制、再到状态机、共识算法, 让你了解了各个方案的优缺点, 以及主流存储产品的选择。

Raft 虽然诞生晚, 但它却是共识算法里面在工程界应用最广泛的。它将一个复杂问题拆分成三个子问题, 分别是 Leader 选举、日志复制和安全性。

Raft 通过心跳机制、随机化等实现了 Leader 选举, 只要集群半数以上节点存活可相互通信, etcd 就可对外提供高可用服务。

Raft 日志复制确保了 etcd 多节点间的数据一致性, 我通过一个 etcd 日志复制整体流程图为你详细介绍了 etcd 写请求从提交到 Raft 模块, 到被应用到状态机执行的各个流程, 剖析了日志复制的核心原理, 即一个日志条目只有被 Leader 同步到一半以上节点上, 此日志条目才能称之为成功复制、已提交。Raft 的安全性, 通过对 Leader 选举和日志复制增加一系列规则, 保证了整个集群的一致性、完整性。

## 思考题

好了, 这节课到这里也就结束了, 最后我给你留了一个思考题。

哪些场景会出现 Follower 日志与 Leader 冲突，我们知道 etcd WAL 模块只能持续追加日志条目，那冲突后 Follower 是如何删除无效的日志条目呢？

感谢你阅读，也欢迎你把这篇文章分享给更多的朋友一起阅读。

### 03 思考题答案

在上一节课中，我给大家留了一个思考题：expensive request 是否影响写请求性能。要搞懂这个问题，我们得回顾下 etcd 读写性能优化历史。

在 etcd 3.0 中，线性读请求需要走一遍 Raft 协议持久化到 WAL 日志中，因此读性能非常差，写请求肯定也会受影响。

在 etcd 3.1 中，引入了 ReadIndex 机制提升读性能，读请求无需再持久化到 WAL 中。

在 etcd 3.2 中，优化思路转移到了 MVCC/boltdb 模块，boltdb 的事务锁由粗粒度的互斥锁，优化成读写锁，实现 “N reads or 1 write” 的并行，同时引入了 buffer 来提升吞吐量。问题就出在这个 buffer，读事务会加读锁，写事务结束时要升级锁更新 buffer，但是 expensive request 导致读事务长时间持有锁，最终导致写请求超时。

在 etcd 3.4 中，实现了全并发读，创建读事务的时候会全量拷贝 buffer，读写事务不再因为 buffer 阻塞，大大缓解了 expensive request 对 etcd 性能的影响。尤其是 Kubernetes List Pod 等资源场景来说，etcd 稳定性显著提升。在后面的实践篇中，我会和你再次深入讨论以上问题。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 基础架构：etcd 一个写请求是如何执行的？



## 精选留言 (8)

写留言



blentle

2021-01-27

老师，如果一个日志完整度相对较高的节点因为自己随机时间比其他节点的长，没能最先发起竞选，其他节点当上leader后同步自己的日志岂不是冲突了？

展开 ∨

作者回复: 你所说的这个日志完整度相对较高的节点，投票时有竞选规则安全限制，如果它的节点比较新会拒绝投票，至于最终先发起选举的节点能否赢得选举，要看其他节点数据情况，如果多数节点的数据比它新，那么先发起选举的节点就无法获得多数选票，如果5个节点中，只有一个节点数据比较长，那的确会被覆盖，但是这是安全的，说明这个数据并未被集群节点多数确认



5

不瘦二十斤  
不改头像

jeffery

2021-01-28

讲的太好了、图文并茂、形象生动、raft这章就够本了！老师问下.es 也用raft！为啥会出现数据不一致性？部署etcd高可用有运维有最佳实践建议吗？谢谢老师

展开 ∨

作者回复: 感谢你的认可，有收获就好，数据不一致原因和高可用运维最佳实践，实践篇都有，稍等，一起学下去，加油，保证让你收获满满



3



春风

2021-01-27

假如老的 Leader A 因为网络问题无法连通 B、C 节点，这时候根据状态图，我们知道它将不停自增任期号，发起选举。等 A 节点网络异常恢复后，那么现有 Leader 收到了新的任期号，就会触发新一轮 Leader 选举，影响服务的可用性。

老师，这里没太懂，A不是本身是leader吗，为什么还要发起选举？

展开 ∨

作者回复: 感谢反馈，是我描述不严谨，让你误会了，在上文中老的Leader A crash后，重启会变成Follower节点，已完善描述，谢谢你



2

**写点啥呢**

2021-01-27

关于选举过程和节点崩溃后恢复有几个问题请教老师:

1. 如果多数follower崩溃后重启恢复(比如极端情况只剩下Leader其它follower同时重启), 根据选举规则是不是会出现重启follower占多数投票给了一个数据不是最新的节点而导致数据丢失。我理解这种情况不满足法定票算法前提, 所以是无法保障数据一致的。
  2. 对于少数节点崩溃恢复后, 它是如何追上leader的最新数据的呢? 比如对于日志复制...
- 展开 ∨

作者回复: 好问题, 第一个问题不能说是数据丢失, 正如你所说的, 可能存在一个新日志条目, 这些异常重启后的follower中没有, 因为异常的follower数达到了法定数, 那么这个新日志条目肯定没同步到多数节点上, 因此是合理的。

第二个问题, 看落后数据程度, 通过内存中的raft log和快照文件追赶, 因篇幅关系, 未介绍日志压缩和快照, 后面我根据情况是否单独加餐篇。如果提交过程中, 不能达到法定数, 这时会超时, 依赖客户端重试

第三个问题, 描述可以更详细点吗, 没太懂, 按我初步理解回答下, 其他节点收到此日志条目后会持久化, 选出leader也有此条记录, 新老leader如果数据有冲突, 以新leader为准



1

**QSkerry**

2021-01-27

主从复制缺点是因为主节点崩溃后, 没有选主机制(是否可以考虑redis的哨兵选主)呢? 还是因为数据一致性呢(raft保持强一致性的话, 也是通过某些机制保证强一致性(主节点读或者ReadIndex))

作者回复: 嗯, 主从复制我个人认为最大的缺点还是数据一致性, Raft的保证一致性背后的核心数学原理其实是抽屉原理, 你可以详细了解下, 不管是集群选举还是数据复制, 都要求一半以上节点确认



4



1

**云原生工程师**

2021-01-27

日志复制流程图很赞, 清晰易懂

展开 ∨



1

**童末**

2021-01-27

额 老师 上节课的思考题答案呢.,是xensive read会导致频繁升级读写锁.导致写请求往后推迟,然后就超时了.吗

展开 ∨

作者回复: 嗯, 晚上更新到04

**Simon**

2021-01-27

在日志图 2 中, Follower B 返回给 client 成功后若突然 crash 了, 此时可能还并未将 6 号日志条目已提交的消息通知到 Follower A 和 C

老师, 这里的Follower B是不是Leader B啊?

展开 ∨

作者回复: 是的, 笔误, 已修正, 谢谢

