**4-etcd**

**一、etcd概念**

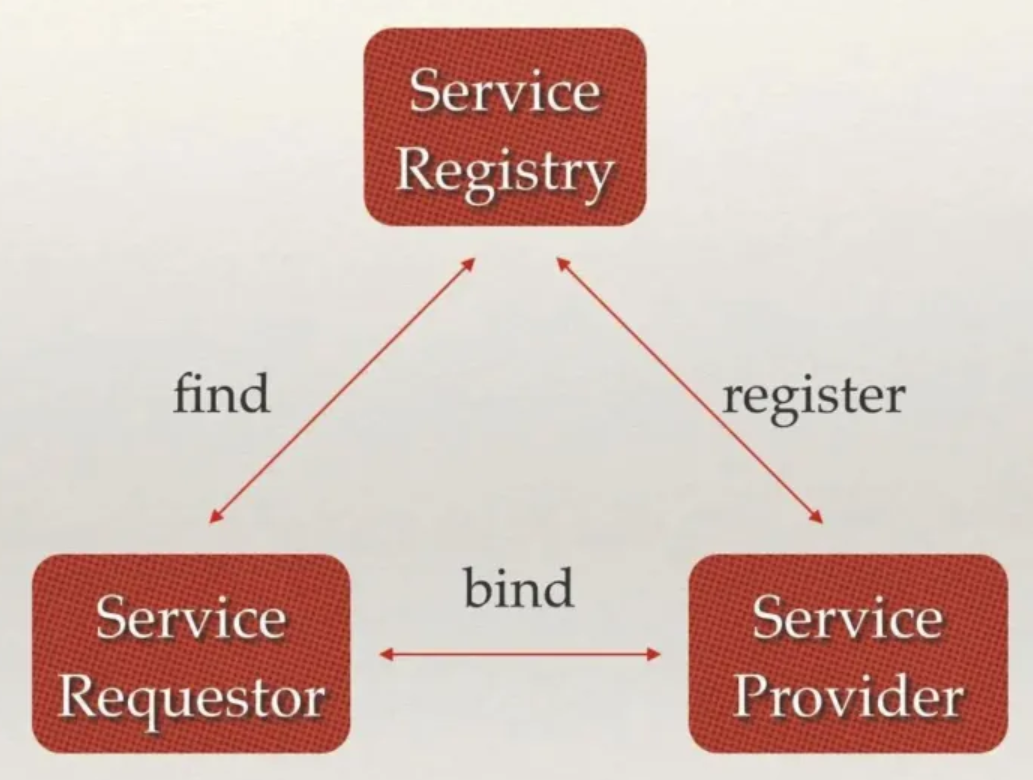
ETCD 是一个高可用的分布式键值key-value数据库，可用于服务发现。

ETCD 采用raft 一致性算法，基于 Go语言实现。

etcd作为一个高可用键值存储系统，天生就是为集群化而设计的。由于Raft算法在做决策时需要多数节点的投票，所以etcd一般部署集群推荐奇数个节点，推荐的数量为3、5或者7个节点构成一个集群。

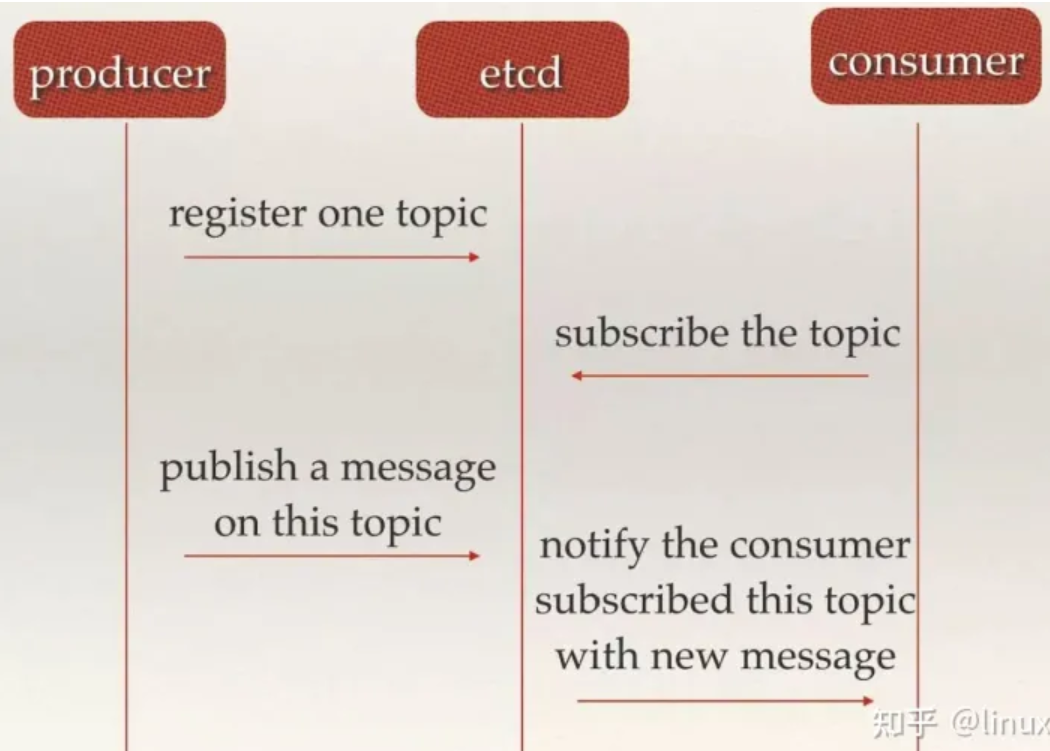
**二、etcd应用场景**

* 配置管理中心
* 服务注册发现



用户可以在 etcd 中注册服务，并且对注册的服务配置 key TTL，定时保持服务的心跳以达到监控健康状态的效果

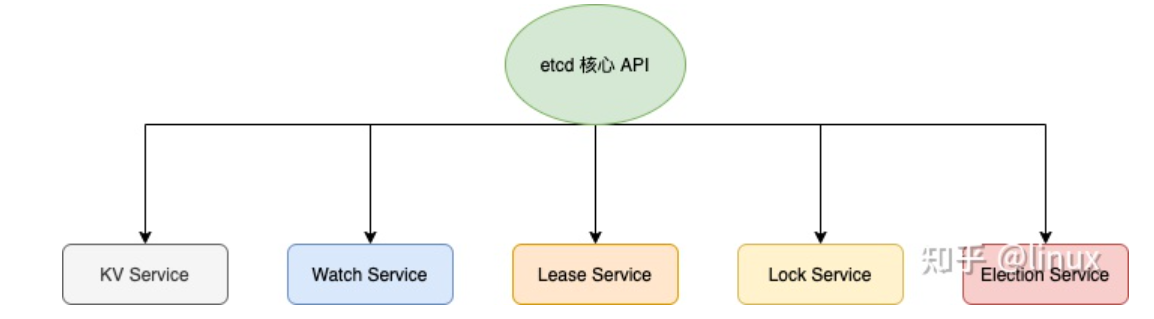
* 消息订阅发布



这类场景的使用方式通常是：

* 应用在启动的时候主动从etcd获取一次配置信息。
* 同时，在etcd节点上注册一个Watcher并等待
* 以后每次配置有更新的时候，etcd都会实时通知订阅者，以此达到获取最新配置信息的目的
* 选主、应用调度
* 分布式队列
* 分布式锁

**三、etcd 核心API**



* KV 服务，创建，更新，获取和删除键值对

|  |
| --- |
| C++ # ./etcdctl put key2 val2 OK # ./etcdctl get key2 key2 val2 # ./etcdctl get key2 -w json {"header":{"cluster\_id":14841639068965178418,"member\_id":10276657743932975437,"revision":9,"raft\_term":7},"kvs":[{"key":"a2V5Mg==","create\_revision":9,"mod\_revision":9,"version":1,"value":"dmFsMg=="}],"count":1} |

* 监视，监视键的更改。

|  |
| --- |
| C++ // 监听 key 的变动 watch key    1) 启两个 session 2) 在 session A 中执行：WATCH key 3) 在 session B 中执行操作 key 的命令，如：PUT key val，DEL key 等，则同时会在 session A 中显示具体操作   // 当前事件发生前先获取前一个 key val watch key --prev-kv   // 监听多个 key 的变动 watch key --prefix |

* 租约，消耗客户端保持活动消息的基元

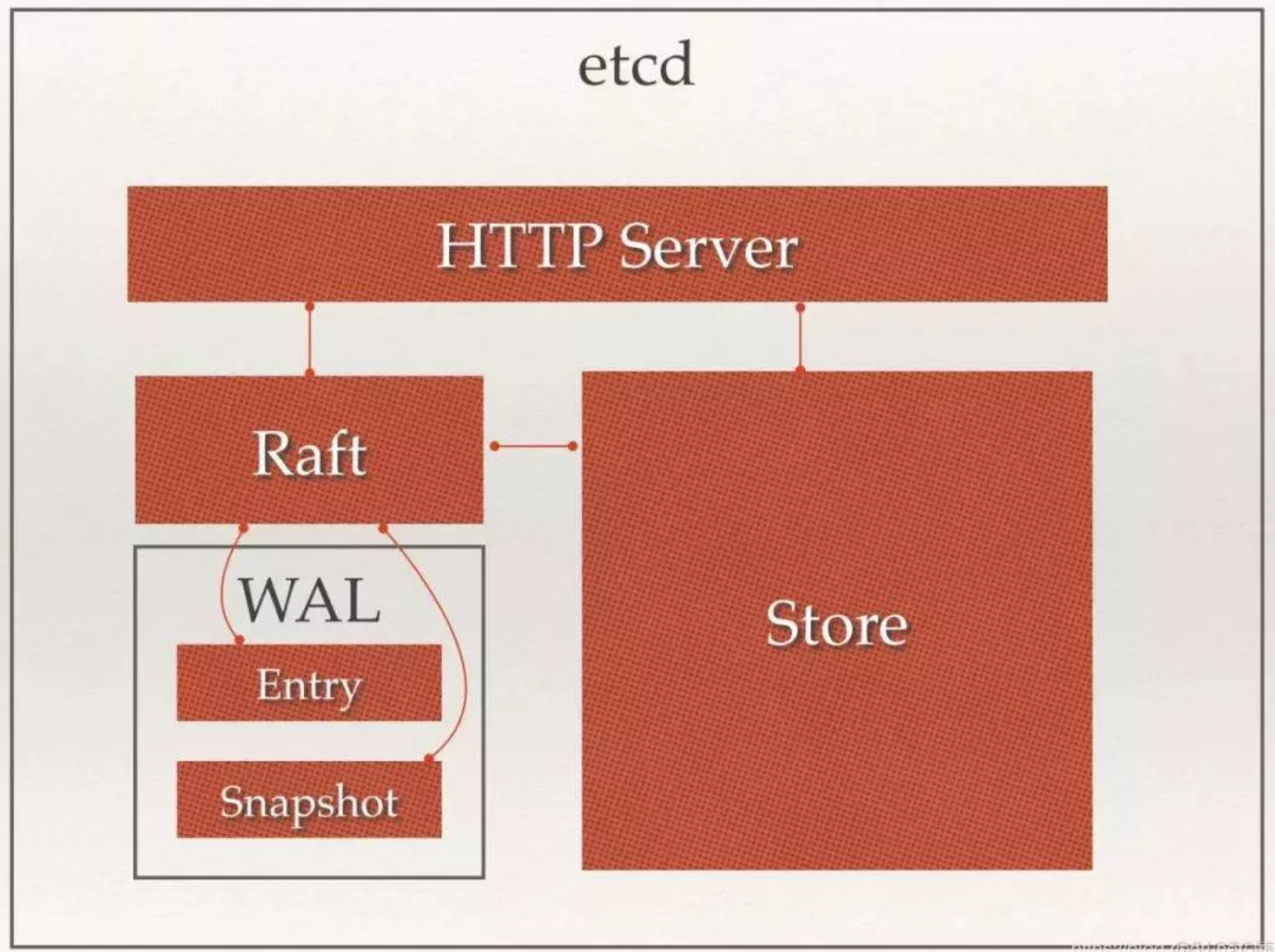
|  |
| --- |
| C++ // 创建一个 100 秒的租约 lease grant 100   // 如果租约创建成功会显示如下输出 lease 694d7b82c54a9309 granted with TTL(100s)   // 将多个 key 绑定到租约 put key1 vla1 --lease=694d7b82c54a9309 put key2 vla2 --lease=694d7b82c54a9309 put key3 vla3 --lease=694d7b82c54a9309   // 获取具有匹配前缀的 key(包括：绑定租约的 key 和未绑定租约的 key) get key --prefix    // 输出结果 key1 vla1  key2  vla2  key3  vla3    // 销毁租约 lease revoke 694d7b82c54a9309   // 获取具有匹配前缀的 key(因为租约已被销毁，所以此时返回的只有未绑定租约的 key) get key --prefix    // 获取租约信息(如果租约未过期，则输出结果会显示租约的剩余日期；如果租约已过期，则显示已过期) lease timetolive 694d7b82c54a9309   // 输出结果(租约已过期) lease 694d7b82c54a9309 already expired   // 续约(可以让租约剩余日期一直保持在设定时间；续约前提是当前租约未过期) lease keep-alive 694d7b82c54a9309 |

* 锁，etcd 提供分布式共享锁的支持，通过lock ttl更新锁的租约时长让锁保活

|  |
| --- |
| C++ USAGE:  etcdctl lock <lockname> [exec-command arg1 arg2 ...] [flags]   OPTIONS:  -h, --help[=false] help for lock  --ttl=10 timeout for session |

* 选举，暴露客户端选举机制

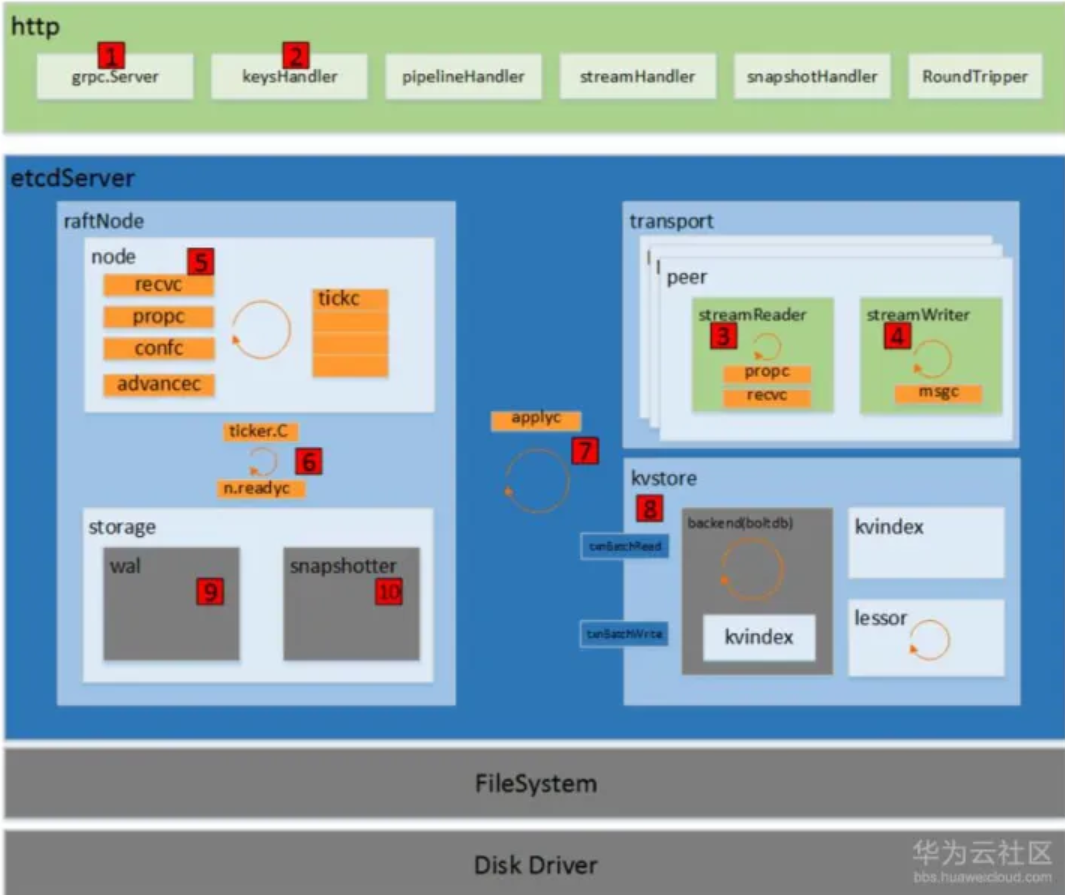
**四、etcd架构**



* HTTP Server：接受客户端发出的 API 请求以及其它 etcd 节点的同步与心跳信息请求。
* Store：kv数据的存储引擎，v3支持不同的后端存储，当前采用boltdb。通过boltdb支持事务操作。用于处理 etcd 支持的各类功能的事务，包括数据索引、节点状态变更、监控与反馈、事件处理与执行等等，是 etcd 对用户提供的大多数 API 功能的具体实现。
* Raft：强一致性算法的具体实现，是 etcd 的核心算法。
* WAL（Write Ahead Log，预写式日志）：是 etcd 的数据存储方式，etcd 会在内存中储存所有数据的状态以及节点的索引，此外，etcd 还会通过 WAL 进行持久化存储。WAL 中，所有的数据提交前都会事先记录日志。
* Snapshot 是为了防止数据过多而进行的状态快照
* Entry 表示存储的具体日志内容。

通常，一个用户的请求发送过来，会经由 HTTP Server 转发给 Store 进行具体的事务处理，如果涉及到节点数据的修改，则交给 Raft 模块进行状态的变更、日志的记录，然后再同步给别的 etcd 节点以确认数据提交，最后进行数据的提交，再次同步。

**五、etcd详细架构及数据流**



从大体上可以将其划分为以下4个模块

* - http：负责对外提供http访问接口和http client
* - raft 状态机：根据接受的raft消息进行状态转移，调用各状态下的动作。
* - wal 日志存储：持久化存储日志条目。
* - kv数据存储：kv数据的存储引擎，v3支持不同的后端存储，当前采用boltdb。通过boltdb支持事务操作。

写入数据流程：

* 客户端向Etcd集群发送写入请求，请求中包含要写入的K-V数据。
* Etcd集群的任意一个节点接收到请求后，将其转发给Leader节点。
* Leader节点根据配置决定是否需要生成WAL Log预写式日志。如果需要，它会向所有的Follower节点发送预写式日志条目（WAL Log Entry），并等待超过半数的Follower节点返回接收日志条目成功的信息。
* 如果预写式日志生成成功，Leader节点会将K-V数据写入到内存中的PageCache中，并通知客户端写入成功。
* 如果配置为"沃尔夫法则"（W+L=N）且N大于一半的节点，在超过半数的Follower节点返回接收Entry日志条目成功的信息后，Etcd会将K-V数据写入到磁盘中的Blot DB中。
* 写入磁盘中的Blot DB后，Etcd会生成一个唯一的Snapshot快照，并保存在指定的存储路径中。

需要注意的是，Etcd的写入流程是异步的，也就是说，写入请求的响应不是立即返回的。Etcd会使用后台线程或者异步操作来完成K-V数据写入磁盘和Snapshot快照的生成。这样可以提高系统的吞吐量和性能。

**六、etcd部分组件详解**

**1. Store**

ETCD的store是ETCD中的核心组件之一，负责管理键值对数据的存储和访问，并提供API接口供客户端进行读写操作。Store通过基于Raft算法的分布式一致性协议，确保配置信息的正确性和一致性。

Store的主要作用包括：

* 数据存储：Store负责存储键值对数据，可以将数据存储在内存中或者持久化到磁盘上。为了保证数据的正确性和一致性，Store会根据Raft协议进行数据同步和一致性检查。
* 键值查询：Store提供API接口，客户端可以通过这些接口对键值对进行查询操作。Store会根据客户端提供的键进行查询，并返回对应的值。
* 事件通知：当配置信息发生变化时，Store会通过事件通知机制及时通知相关的客户端进行更新。客户端可以通过订阅事件来接收配置信息的变更通知。
* 节点状态变更：Store还负责管理ETCD集群中节点的状态变更，包括节点的加入和离开。当有节点加入或离开时，Store会更新相应的节点状态信息，并通知其他相关的客户端。
* 监控与反馈：Store还提供一些监控和反馈功能，可以收集ETCD集群的运行状态和性能指标，并进行统计和报告。这些信息可以帮助管理员及时发现和解决问题。

**2. WAL**

etcd 的数据存储分为两个部分：

内存存储：内存中的存储除了顺序化的记录下所有用户对节点数据变更的记录外，还会对用户数据进行索引、建堆等方便查询的操作。

持久化（硬盘）存储：持久化则使用 WAL（Write Ahead Log，预写式日志）进行记录存储。

WAL 日志是二进制的，解析出来后是以上数据结构 LogEntry。其中：

* 第一个字段 type，只有两种
* 0 表示 Normal
* 1 表示 ConfChange，ConfChange 表示 etcd 本身的配置变更同步，比如有新的节点加入等。
* 第二个字段是 term，每个 term 代表一个 Leader 的任期，每次 Leader 变更 term 就会变化。
* 第三个字段是 index，这个序号是严格有序递增的，代表变更序号。
* 第四个字段是二进制的 data，将 Raft Request 对象的 pb 结构整个保存下。

etcd 源码下有个 tools/etcd-dump-logs 脚本工具，可以将 WAL 日志 dump 成文本查看，可以协助分析 Raft 协议。

Raft 协议本身不关心应用数据，也就是 data 中的部分，一致性都通过同步 WAL 日志来实现，每个 Node 将从 Leader 收到的 data apply 到本地的存储，Raft 只关心日志的同步状态，如果本地存储实现的有 Bug，比如没有正确的将 data apply 到本地，也可能会导致数据不一致。

在 WAL 的体系中，所有的数据在提交之前都会进行日志记录。在 etcd 的持久化存储目录中，有两个子目录：

一个是 WAL：存储着所有事务的变化记录；

另一个是 Snapshot：存储着某一个时刻 etcd 所有目录的数据。

通过 WAL 和 Snapshot 相结合的方式，etcd 可以有效的进行数据存储和节点故障恢复等操作。

**为什么需要 Snapshot（快照）？**

因为随着使用量的增加，WAL 存储的数据会暴增，为了防止磁盘很快就爆满，etcd 默认每 10000 条记录做一次 Snapshot，经过 Snapshot 以后的 WAL 文件就可以删除。所以，通过 API 可以查询的操作历史记录默认为 1000 条。

首次启动时，etcd 会把启动的配置信息存储到 data-dir 配置项指定的目录路径下。配置信息包括 Local Node ID、Cluster ID 和初始时的集群信息。用户需要避免 etcd 从一个过期的数据目录中重新启动，因为使用过期的数据目录启动的 Node 会与 Cluster 中的其他 Nodes 产生不一致性，例如：之前已经记录并同意 Leader Node 存储某个信息，重启后又向 Leader Node 申请这个信息。所以，为了最大化集群的安全性，一旦有任何数据损坏或丢失的可能性，你就应该把这个 Node 从 Cluster 中移除，然后加入一个不带数据目录的 New Node。

WAL（Write Ahead Log）最大的作用是记录了整个数据变化的全部历程。在 etcd 中，所有数据的修改在提交前，都要先写入到 WAL 中。使用 WAL 进行数据的存储使得 etcd 拥有两个重要功能：

故障快速恢复： 当你的数据遭到破坏时，就可以通过执行所有 WAL 中记录的修改操作，快速从最原始的数据恢复到数据损坏前的状态。

数据回滚（undo）或重做（redo）：因为所有的修改操作都被记录在 WAL 中，需要回滚或重做，只需要方向或正向执行日志中的操作即可。

**WAL 和 Snapshot 的命名规则？**

在 etcd 的数据目录中，WAL 文件以 $seq-$index.wal 的格式存储。最初始的 WAL 文件是 0000000000000000-0000000000000000.wal，表示这是所有 WAL 文件中的第 0 个，初始的 Raft 状态编号为 0。运行一段时间后可能需要进行日志切分，把新的条目放到一个新的 WAL 文件中。

假设，当集群运行到 Raft 状态为 20 时，需要进行 WAL 文件的切分时，下一份 WAL 文件就会变为 0000000000000001-0000000000000021.wal。如果在 10 次操作后又进行了一次日志切分，那么后一次的 WAL 文件名会变为 0000000000000002-0000000000000031.wal。可以看到 “-” 符号前面的数字是每次切分后自增 1，而 “-” 符号后面的数字则是根据实际存储的 Raft 起始状态来定。

而 Snapshot 的存储命名则比较容易理解，以 $term-$index.wal 格式进行命名存储。term 和 index 就表示存储 Snapshot 时数据所在的 Raft 节点状态，当前的任期编号以及数据项位置信息。

etcd 的数据模型

etcd 的设计目的是用来存放非频繁更新的数据，提供可靠的 Watch 插件，它暴露了键值对的历史版本，以支持低成本的快照、监控历史事件。这些设计目标要求它使用一个持久化的、多版本的、支持并发的数据数据模型。

当 etcd 键值对的新版本保存后，先前的版本依然存在。从效果上来说，键值对是不可变的，etcd 不会对其进行 in-place 的更新操作，而总是生成一个新的数据结构。为了防止历史版本无限增加，etcd 的存储支持压缩（Compact）以及删除老旧版本。

**七、Raft协议算法原理**

Raft算法为目标设计的一致性共识算法，涉及到共识算法就必然会提到Paxos，但是Paxos的实现和理解起来都非常复杂。Raft协议采用分治的思想，把分布式协同的问题分为3个问题：

选举： 一个新的集群启动时，或者老的leader故障时，会选举出一个新的leader。

日志同步： leader必须接受客户端的日志条目并且将他们同步到集群的所有机器。

安全： 保证任何节点只要在它的状态机中生效了一条日志，就不会在相同的key上生效另一条日志条目。

**7.1 Raft 定义**

首先，Raft是一种“算法”；其次，Raft 是一种为了管理“复制日志”算法；最后，Raft 是一种为了管理复制日志“一致性”算法。

**那么问题来了，什么是一致性？**

一致性是分布式系统容错的基本问题。一组机器像一个整体一样工作，即使其中小半部分机器(不大于N/2)出现故障也能够继续工作下去， 一旦他们就状态做出决定，该决定就是最终决定。 例如，即使2台服务器发生故障，5台服务器的集群也可以继续运行。 如果更多服务器失败，它们将停止进展（但永远不会返回错误的结果）

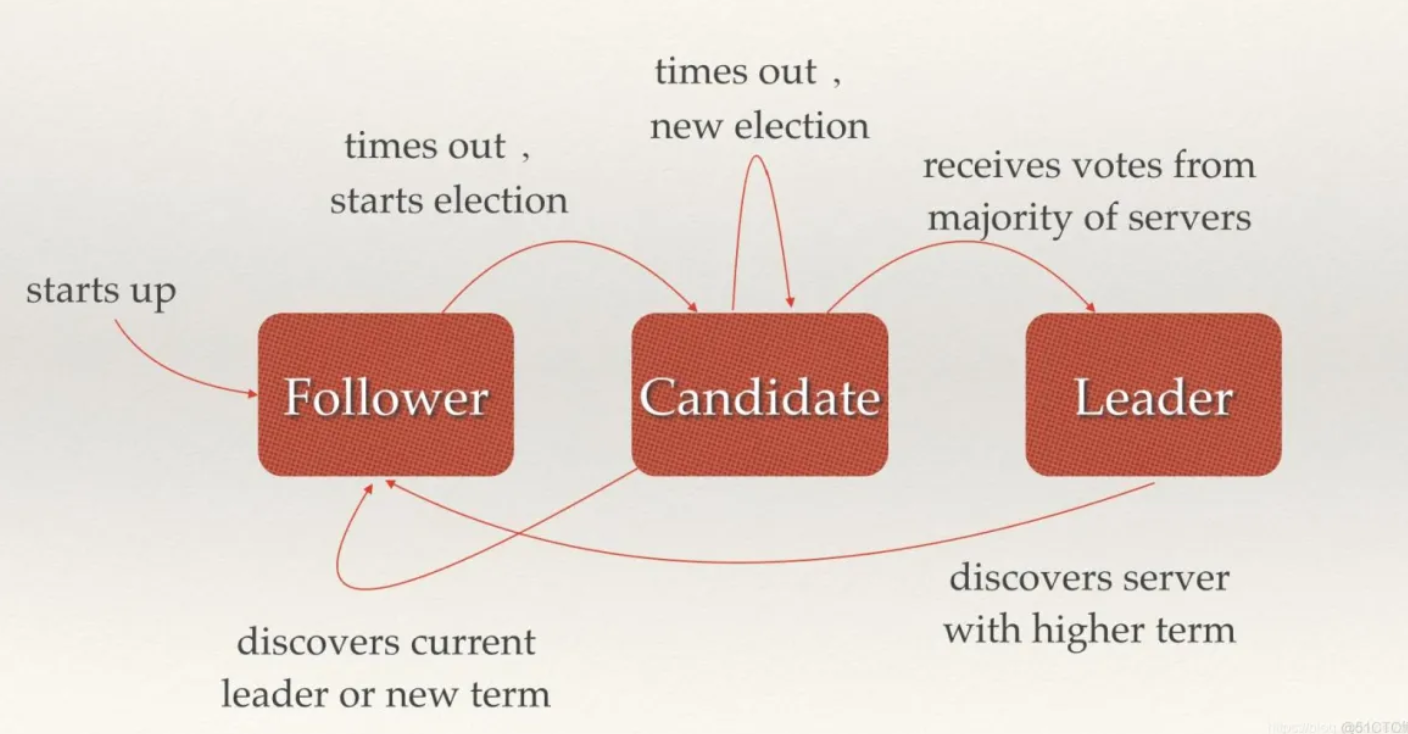
**7.2 Raft 三种角色**

根据 “拜占庭将军问题” ，我们可以提取出三种状态的角色：

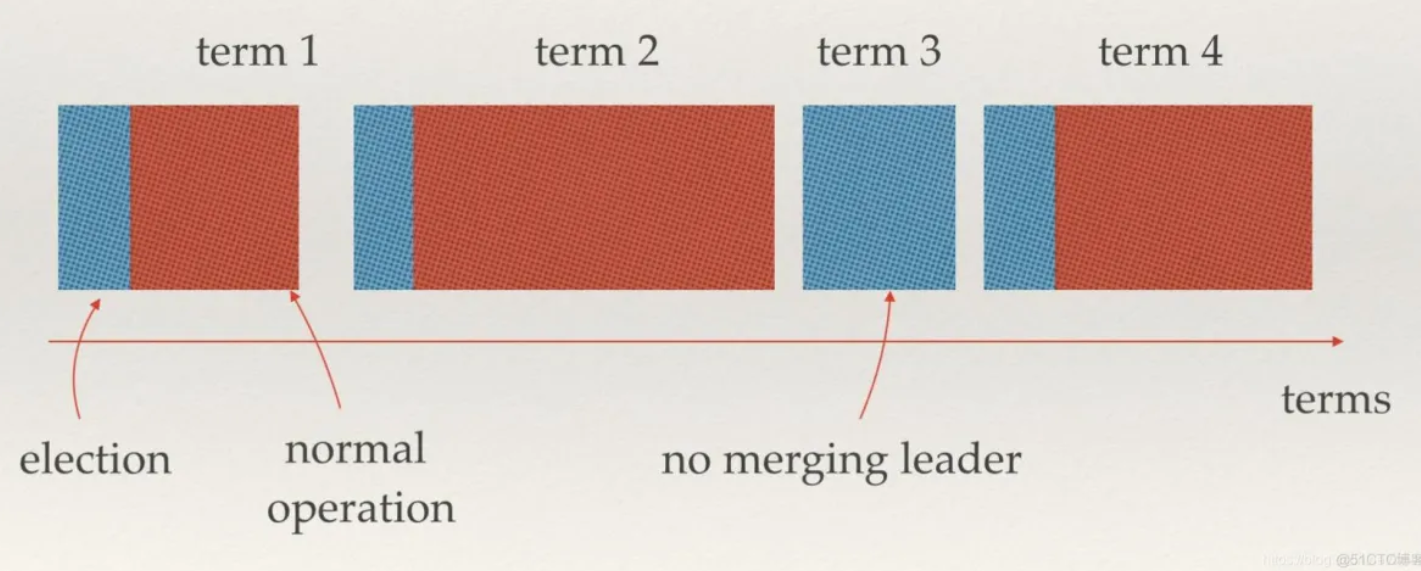
* 1. 追随者：将军B和C愿意投票给将军A, 将军B和C成为将军A的跟随者
* 2. 候选者：将军A倒计时结束，成为大将军候选者
* 3. 大将军：将军A收到大多数将军的投票后，成为大将军

在包含若干节点Raft集群中，其实也存在着相似的角色：Leader、Candidate、Follower。每种角色负责的任务也不一样，正常情况下，集群中的节点只存在 Leader 与 Follower 两种角色。

* 1. Leader（领导者）：处理所有客户端交互，日志复制等，一般一次只有一个Leader；
* 2. Follower（追随者）：响应 Leader 的日志同步请求，响应 Candidate 的邀票请求，以及把客户端请求到 Follower 的事务转发（重定向）给 Leader；
* 3. Candidate（候选者）：负责选举投票，集群刚启动或者 Leader 宕机时，角色为 Follower 的节点将转为 Candidate 并发起选举，选举胜出（获得超过半数节点的投票）后，从 Candidate 转为 Leader 角色；



Raft把时间划分为任期(Term)(如下图所示)，任期是一个递增的整数，一个任期是从开始选举leader到leader失效的这段时间。有点类似于一届总统任期，只是它的时间是不一定的，也就是说只要leader工作状态良好，它可能成为一个独裁者，一直不下台。



**7.3 Raft 算法问题分解**

根据上面的介绍，我们知道通常Raft集群中只有一个Leader，其它节点都是Follower。Follower都是被动的：他们不会发送任何请求，只是简单的响应来自Leader或者Candidate的请求。Leader负责处理所有的客户端请求（如果一个客户端和Follower联系，Follower会把请求重定向给Leader）。为了简化逻辑和实现，Raft将一致性问题分解成三个独立的子问题：

* 1. Leader election：当leader宕机或者集群创建时，需要选举一个新的Leader
* 2. Log replication：Leader接收来自客户端的请求并将其以日志的形式复制到集群中的其它节点，并且强制要求其它节点的日志和自己保持一致
* 3. Safety：如果有任何节点已经应用了一个确定的日志条目到它的状态机中，那么其它服务节点不能在同一个日志索引位置应用一个不用的指令

**7.4 Raft 算法原理**

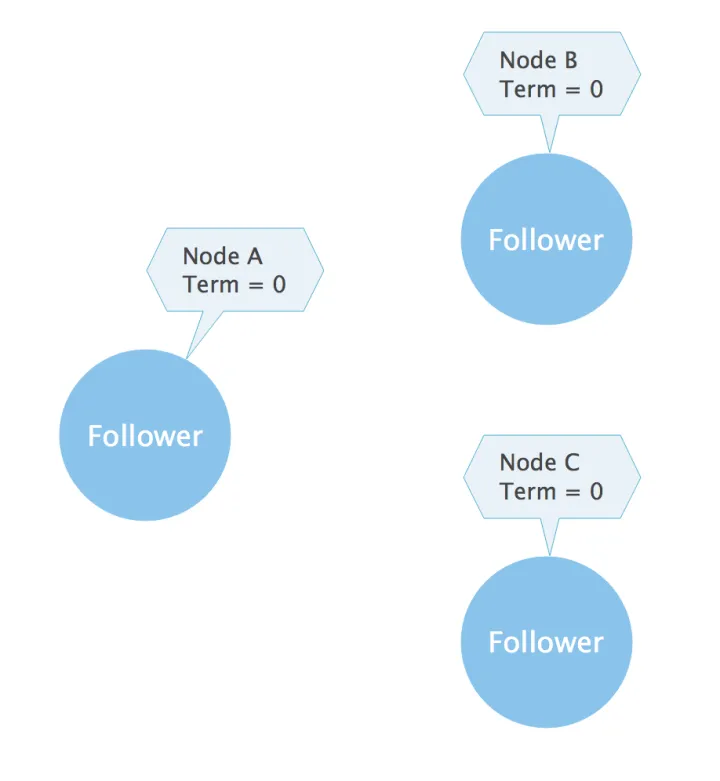
上面讲了这么多，其实都是伏笔 。接下来，本文的核心知识点来了。

**7.4.1 Raft 角色选举**

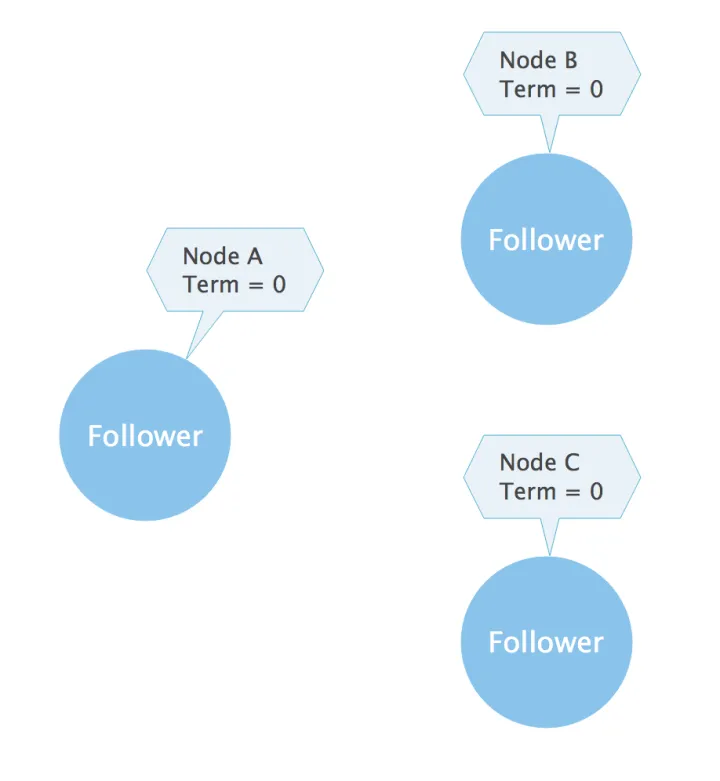
根据 Raft 协议，一个应用 Raft 协议的集群在刚启动时，所有节点状态都是Follower态，由于没有Leader，Follower 无法与 Leader 保持心跳（heart beat），Follower等待心跳超时（每个Follower的心跳超时时间不一样），Followers 会认为 Leader 已经 down 掉。最先超时的Follower进而转为 Candidate 状态，然后，Candidate 将向集群中的其它节点请求投票，同意自己升级为 Leader，如果 Candidate 收到超过半数节点的投票（N/2+1），它将获胜成为 Leader。

**角色选举详细流程如下：**

**第一阶段**：都是 Follower 状态



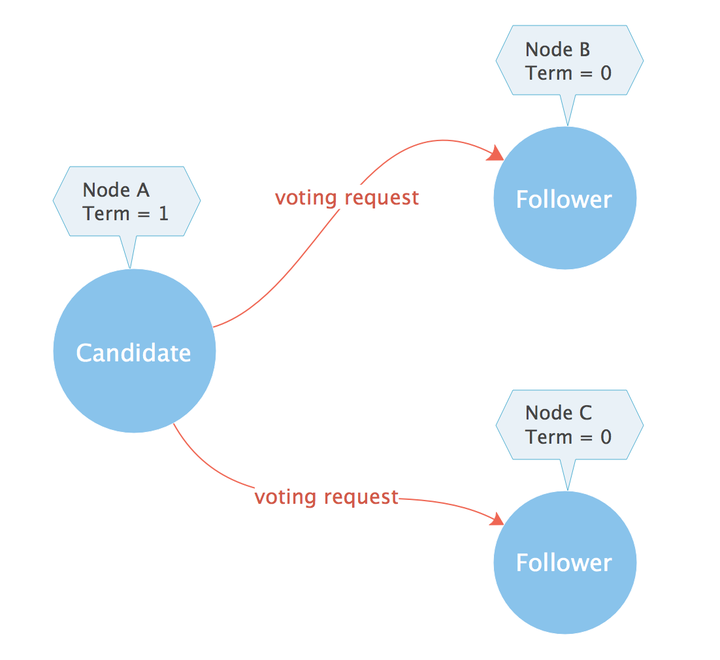
一个应用Raft协议的集群在刚开始启动时（或者 Leader 宕机重启时），所有的节点都是 Follower 状态，初始任期（Term，即某次选举的唯一标识）都是0。同时启动选举定时器，每个节点的选举定时器都不一致且都在100～500ms之间（避免同时发起选举）。



**第二阶段：**从 Follower 状态转换为Candidate，并发起投票

由于没有 Leader，Followers 无法与 Leader 保持心跳（heart beat），节点启动后在一个选举定时器周期内未收到心跳和投票请求，则状态转变为 Candidate 状态、Term 自增，并向集群中所有节点发送投票请求并且重置选举定时器。

注意：每个节点选举定时器超时时间都在 100 ～ 500 ms之内，且不一致。因此，可以避免所有的Follower同时转化为 Candidate状态，换言之，最先转为 Candidate 并发起投票请求的节点将具有成为 Leader 的先发优势。

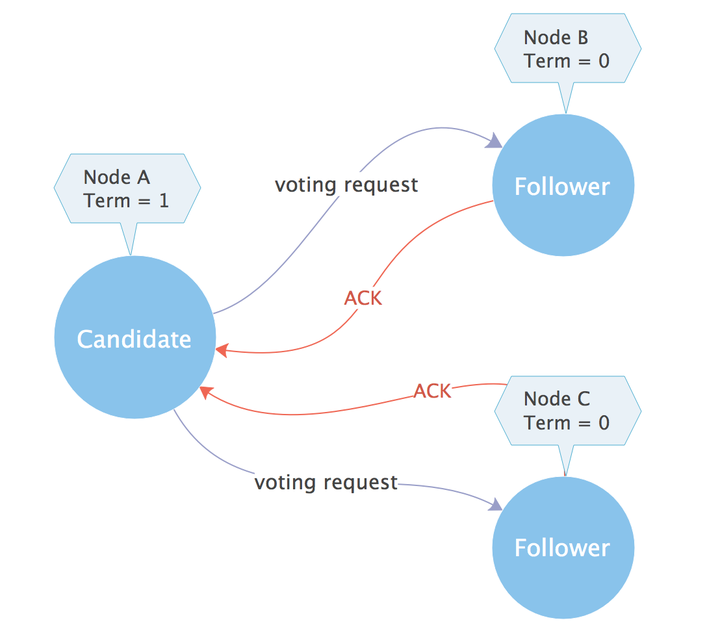


**第三阶段：**投票策略

Follower 节点收到投票请求后会根据以下情况决定是否接受投票请求：

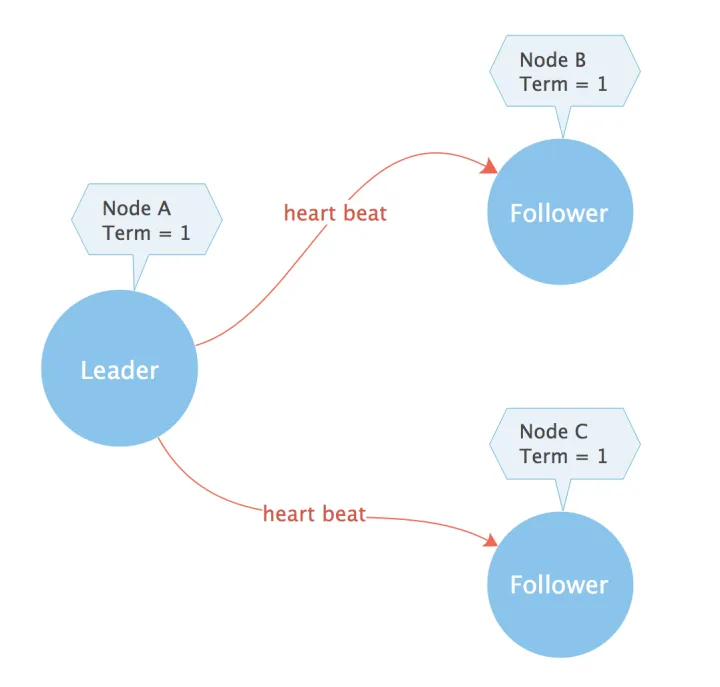
请求节点的 Term 大于自己的 Term，且自己尚未投票给其它节点，则接受请求，把票投给 Candidate 节点；

请求节点的 Term 小于自己的 Term，且自己尚未投票，则拒绝请求，将票投给自己。



**第四阶段：**Candidate 转换为 Leader

经过一轮选举后，正常情况，会有一个 Candidate 节点收到超过半数（N/2+1）其它节点的投票，那么它将胜出并升级为 Leader 节点，然后定时发送心跳给其它节点，其它节点会转化为 Follower 节点并与 Leader 保持同步，如此，本轮选举结束。如果一轮选举中，Candidate 节点收到的投票没有超过半数，那么将进行下一轮选举。



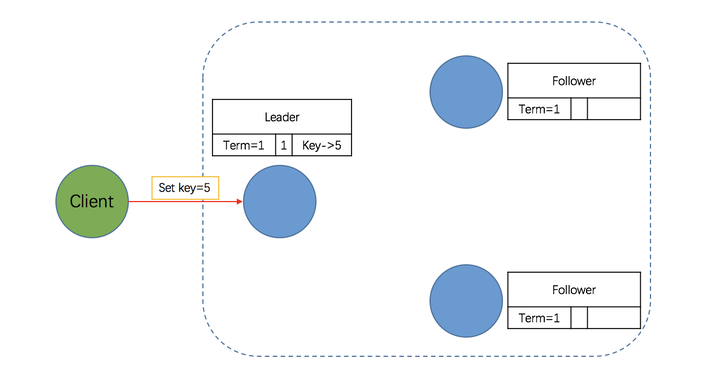
**7.4.2 Raft 日志同步**

一个 Raft 集群中只有 Leader 节点能够处理客户端的请求（如果客户端的请求发到了 Follower 节点，Follower 将会把请求重定向到 Leader），客户端的每一个请求都包含一条被复制到状态机执行的指令。Leader 把这条指令作为一条新的日志条目（Entry）附加到日志中去，然后并行的将附加条目发送给 Followers，让它们复制这条日志条目。当这条日志条目被 Followers 安全的复制，Leader 会应用这条日志条目到它的状态机中，然后把执行的结果返回给客户端。如果 Follower 崩溃或者运行缓慢，再或者是网络丢包，Leader 会不断的重复尝试附加日志条目（尽管已经回复了客户端）直到所有的 Follower 最终都存储了所有的日志条目，确保强一致性。

**日志复制详细流程如下：**

**第一阶段**：客户端请求提交到 Leader

Leader 收到客户端请求：如存储一个数据：5；Leader 收到请求后，会将它作为日志条目（Entry）写入本地日志中。此时该 Entry 是未提交状态（uncommitted），Leader并不会更新本地数据，因此它是不可读的。



**第二阶段**：Leader 将 Entry 发送到其它Follower

Leader 与 Followers 之间保持心跳联系，跟心跳 Leader 将追加的 Entry（AppendEntries） 并行的发送到其它 Follower 节点，并让它们复制这条日志条目，这一过程我们称为：复制（Replication）。

**1.为什么 Leader 向 Follower 发送的 Entry 是 AppendEntries 呢？**

因为 Leader 与 Follower 的心跳是周期性的，而一个周期 Leader 可能接收到客户端的多个请求，因此，随心跳向 Followers 发送的大概率是多个 Entry，即 AppendEntries。在本例中为了简单，只有一条请求，自然 只有一个 Enrety。

**2. Leader 向 Followers 发送的不仅仅是追加的 Entry （AppendEntries）**

在发送追加日志条目的时候，Leader 会把新日志条目之前的条目索引（前一个日志条目）位置（prevLogIndex）和Leader任期号（term）包含在里边。如果 Follower 在它的日志中找不到包含相同索引位置和任期号的条目，那么它会拒接这个新的日志条目。因为出现这种情况说明 Follower 和 Leader 是不一致的。

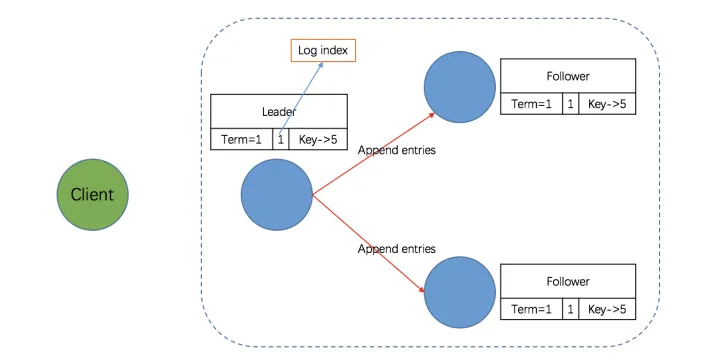
**3. 如何解决 Leader 和 Follower 不一致的问题？**

在正常情况下，Leader 和 Follower 的日志保持一致，所以追加日志的一致性从来不会失败。然后，Leader 和 Follower 的一系列崩溃情况下会使它们的日志处于不一致的状态。Follower 可能会丢失一些在新的 Leader 中有的日志条目，它也可能拥有一些 Leader 没有的日志条目，或者两者都有发生。丢失或者多出的日志条目可能会持续多个任期。

要使 Follower 的日志与 Leader 恢复一致，Leader 必须找到最后两者达成一致的地方，然后删除从那个节点之后的所有日志，发送自己的日志给 Follower。所有的这些操作都在进行附加日志一致性检查时完成。

Leader 节点针对每个 Follower 节点维护了一个 nextIndex，这表示下一个需要发送给 Follower 的日志条目的索引地址。当一个 Leader 刚获得权力的时候，它初始化所有的 nextIndex 值为自己的最后一条日志的 index + 1。如果一个 Follower 日志和 Leader 不一致，那么在下一次附加日志的时候就会检查失败。在被 Follower 拒绝之后，Leader 就会减小该 Follower 对应的 nextIndex 值并进行重试（即回溯）。

最终 nextIndex 会在某个位置使得 Leader 和 Follower 的日志达成一致。当这种情况发生，附加日志就会成功，这时就会把 Follower 冲突的日志条目全部删除并且附加上 Leader 的日志。一旦附加成功，那么 Follower 的日志就会和 Leader 保持一致，并且在接下来的任期里一致继续保持。



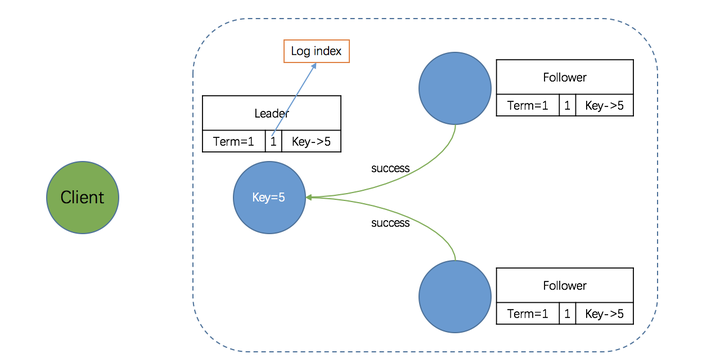
**第三阶段**：Leader 等待 Followers 回应

Followers 接收到 Leader 发来的复制请求后，有两种可能的回应：

写入本地日志，返回 Success

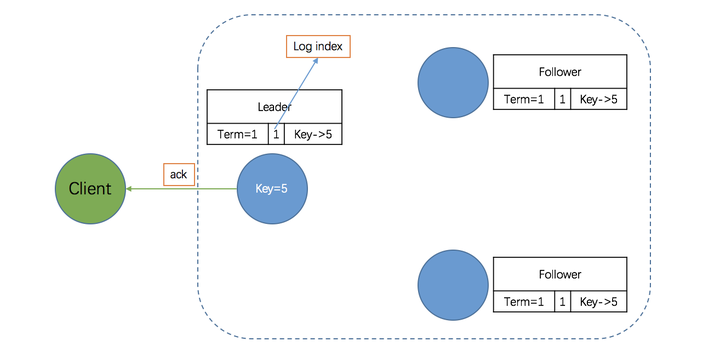
一致性检查失败，拒绝写入，返回 false。原因和解决办法上面已经详细说明。

注：此时该 Entry 的状态也是未提交（uncommitted）。完成上述步骤后，Followers 会向 Leader 发出回应 - success，当 Leader 收到大多数 Followers 的回应后，会将第一阶段写入的 Entry 标记为提交状态（committed），并把这条日志条目应用到它的状态机中。



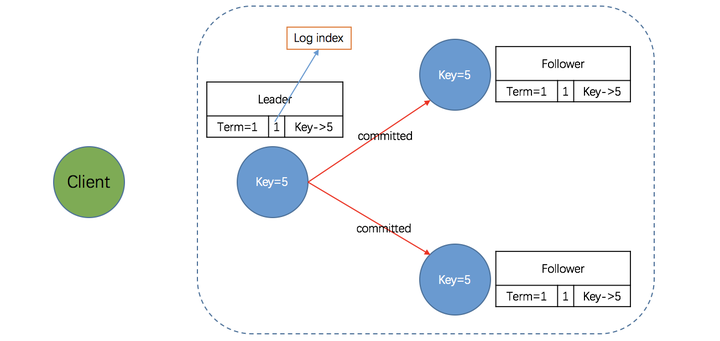
**第四阶段**：Leader回应客户端

完成前三个阶段后，Leader 会回应客户端 - OK，写操作成功。



**第五阶段**：Leader 通知 Followers Entry 已提交

Leader 回应客户端后，将随着下一个心跳通知 Followers，Followers 收到通知后也会将 Entry 标记为提交状态。至此，Raft 集群超过半数节点已经达到一致状态，可以确保强一致性。需要注意的是，由于网络、性能、故障等各种原因导致的“反应慢”、“不一致”等问题的节点，也会最终与 Leader 达成一致。



**7.4.3 Raft 安全性保证**

前面的章节里描述了 Raft 算法是如何选举 Leader 和 日志复制。然而，到目前为止描述的机制并不能充分保证每一个状态机会按照相同的顺序执行相同的指令。例如：一个 Follower 可能处于不可用的状态，同时 Leader 已经提交了若干的日志条目；然后这个 Follower 恢复（尚未与 Leader 达成一致）而 Leader 故障，如果该 Follower 被选举为 Leader 并且覆盖这些日志条目，就会出现问题：不同的状态机执行不同的指令序列。

鉴于此，在 Leader 选举的时候需要增加一些限制来完善 Raft 算法。这些限制可保证任何的 Leader 对于给定的任期号（Term），都拥有之前任期的所有被提交的日志条目（所谓 Leader 的完整特性）。

**7.4.3.1 选举限制**

对于所有基于 Leader 机制一致性算法，Leader 都必须存储所有已经提交的日志条目。为了保障这一点，Raft 使用了一种简单而有效的办法，以保证之前任期号中已提交的日志条目在选举的时候都会出现在新的Leader中。换言之，日志条目的传送是单向的，只从 Leader 传给 Follower， 并且 Leader 从不会覆盖自身本地日志中已经存在的条目。

Raft 使用投票的方式来阻止一个 Candidate 赢得选举，除非这个 Candidate 包含了所有已经提交的日志条目。Candiate 为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目都在这些服务器节点中肯定存在于至少一个节点上。如果 Candidate 的日志至少和大多数的服务器节点一样新，那么它一定持有了所有已经提交的日志条目。投票请求的限制：请求中包含了 Candidate 的日志信息，然后投票人会拒绝那些日志没有自己日志新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号，确定谁的日志比较新。如果两份日志最后的条目和任期号不同，那么任期号大的日志更加新一些。如果两份日志最后的任期号相同，那么日志比较长的那个就更加新。

**7.4.3.2 提交之前任期内的日志条目**

Leader 知道一条当前任期内的日志记录是可以被提交的，只要它被复制到了大多数 Follower 节点上。如果一个Leader 在提交日志条目之前崩溃了，继任的 Leader 会继续尝试复制这条日志记录。然而，一个 Leader 并不能断定之前任期里的日志条目被保存到大多数 Follower 上就一定已经提交了。很明显，从日志复制的过程可以看出。

鉴于上述情况，Raft 算法不会通过计算副本数的方式去提交一个之前任期内的日志条目。只有 Leader 当前任期里的日志条目通过计算副本数目可以被提交；一旦当前任期的日志条目以这种方式提交，由于日志匹配特性，之前的日志条目也都会被间接提交。在某些情况下， Leader 可以安全的知道一个老的日志条目是否已经被提交（只需判断该条目是否存储到所有的节点上），但是 Raft 为了简化问题使用一种更加保守的方式。

当 Leader 复制之前任期里的日志时，Raft 会为所有的日志保留原始任期号，这在提交规则上产生了额外的复杂性。但是，这种策略更加容易辨别出日志，因为它可以随着时间和日志变化对日志维护着同一个任期号。此外，该策略使得新 Leader 只需要发送较少的日志条目。

**八、etcd集群搭建**

**配置ETCD文件**

|  |
| --- |
| Bash $vim /opt/kubernetes/etcd/cfg/etcd.conf #[Member] ETCD\_NAME="etcd-1" # k8s01为etcd-1，k8s02为etcd-2。。。每个节点唯一标识符 ETCD\_DATA\_DIR="/opt/kubernetes/etcd/data/default.etcd" ETCD\_LISTEN\_PEER\_URLS="https://192.168.1.241:2380" # 修改对应ip，k8s01为241，k8s02为242... ETCD\_LISTEN\_CLIENT\_URLS="https://192.168.1.241:2379" # 修改对应ip，k8s01为241，k8s02为242...   #[Clustering] ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS="https://192.168.1.241:2380" # 修改对应ip，k8s01为241，k8s02为242... ETCD\_ADVERTISE\_CLIENT\_URLS="https://192.168.1.241:2379" # 修改对应ip，k8s01为241，k8s02为242... ETCD\_INITIAL\_CLUSTER="etcd-1=https://192.168.1.241:2380,etcd-2=https://192.168.1.242:2380,etcd-3=https://192.168.1.243:2380" ETCD\_INITIAL\_CLUSTER\_TOKEN="etcd-cluster" ETCD\_INITIAL\_CLUSTER\_STATE="new" |

ETCD\_NAME：节点名称，集群中唯一

ETCD\_DATA\_DIR：数据目录

ETCD\_LISTEN\_PEER\_URLS：集群通信监听地址

ETCD\_LISTEN\_CLIENT\_URLS：客户端访问监听地址

ETCD\_INITIAL\_ADVERTISE\_PEERURLS：集群通告地址

ETCD\_ADVERTISE\_CLIENT\_URLS：客户端通告地址

ETCD\_INITIAL\_CLUSTER：集群节点地址

ETCD\_INITIALCLUSTER\_TOKEN：集群Token

ETCD\_INITIALCLUSTER\_STATE：加入集群的当前状态，new是新集群，existing表示加入已有集群

**查看集群**

|  |
| --- |
| Bash [root@kubenode01 etcd]# ./bin/etcdctl --cacert=/opt/kubernetes/etcd/ssl/ca.pem --cert=/opt/kubernetes/etcd/ssl/server.pem --key=/opt/kubernetes/etcd/ssl/server-key.pem --endpoints="https://10.10.21.73:2379,https://10.10.21.74:2379,https://10.10.21.75:2379" endpoint health --write-out=table   +--------------------------+--------+-------------+-------+ | ENDPOINT | HEALTH | TOOK | ERROR | +--------------------------+--------+-------------+-------+ | https://10.10.21.75:2379 | true | 13.407895ms | | | https://10.10.21.74:2379 | true | 13.61133ms | | | https://10.10.21.73:2379 | true | 14.868649ms | | +--------------------------+--------+-------------+-------+ |

**九、etcd运维**

**磁盘打满：**

调整etcd配置扩大容量，重启服务

压缩老版本数据清理

etcd磁盘打满，参考：ETCD故障排查，ETCD存储满了如何处理

**节点宕机：**

正常来说，宕机机器重新拉起服务即可，raft会同步最新的节点数据

**内存爆满：**

查看etcd的日志，哪个目录有异常，或者是否有异常日志