raft

Raft 是分布式一致性算法

简介

https://www.oschina.net/action/GoToLink?url=https%3A%2F%2Framcloud.atlassian.net%2Fwiki%2Fdownload%2Fattachments%2F6586375%2Fraft.pdf 是一种为了管理日志复制的分布式一致性算法。Raft 出现之前，Paxos 一直是分布式一致性算法的标准，Paxos 难以理解，更难以实现，Raft 的设计目标是简化 Paxos，使得算法既容易理解，也容易实现。

Paxos 和 Raft 都是分布式一致性算法，这个过程如同投票选举领袖（Leader），参选者（Candidate）需要说服大多数投票者（Follower）投票给他，一旦选举出领袖，就由领袖发号施令，Paxos 和 Raft 的区别在于选举的具体过程不同。

Raft 可以解决分布式 CAP 理论中的 CP，即 一致性（C：Consistency） 和 分区容忍性（P：Partition Tolerance），并不能解决 可用性（A：Availability） 的问题。

1.2 分布一致性

分布式一致性 (distributed consensus) 是分布式系统中最基本的问题，用来保证一个分布式系统的可靠性以及容错能力。简单来说，**分布式一致性是指多个服务器的保持状态一致**。

在分布式系统中，可能出现各种意外（断电、网络拥塞、CPU/内存耗尽等等），使得服务器宕机或无法访问，最终导致无法和其他服务器保持状态一致。为了应对这种情况，就需要有一种一致性协议来进行容错，使得分布式系统中即使有部分服务器宕机或无法访问，整体依然可以对外提供服务。

以容错方式达成一致，自然不能要求所有服务器都达成一致状态，只要**超过半数**以上的服务器达成一致就可以了。假设有 N 台服务器， 大于等于 N / 2 + 1 台服务器就算是半数以上了 。

1.3 复制状态机

复制状态机（Replicated State Machines） 是指一组服务器上的状态机产生相同状态的副本，并且在一些机器宕掉的情况下也可以继续运行。一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的，如上图。每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列。

保证复制日志相同就是一致性算法的工作了。在一台服务器上，一致性模块接收客户端发送来的指令然后增加到自己的日志中去。它和其他服务器上的一致性模块进行通信来保证每一个服务器上的日志最终都以相同的顺序包含相同的请求，尽管有些服务器会宕机。一旦指令被正确的复制，每一个服务器的状态机按照日志顺序处理他们，然后输出结果被返回给客户端。因此，服务器集群看起来形成一个高可靠的状态机。

实际系统中使用的一致性算法通常含有以下特性：

> **安全性保证**（绝对不会返回一个错误的结果）：在非拜占庭错误情况下，包括网络延迟、分区、丢包、冗余和乱序等错误都可以保证正确。
>
> **可用性：**集群中只要有大多数的机器可运行并且能够相互通信、和客户端通信，就可以保证可用。因此，

> 一个典型的包含 5 个节点的集群可以容忍两个节点的失败。服务器被停止就认为是失败。他们有稳定的存储的时候可以从状态中恢复回来并重新加入集群。
>
> **不依赖时序来保证一致性：**物理时钟错误或者极端的消息延迟只有在最坏情况下才会导致可用性问题。
>
> 通常情况下，一条指令可以尽可能快的在集群中大多数节点响应一轮远程过程调用时完成。小部分比较慢的节点不会影响系统整体的性能。

1.4 RAFT应用

通过 RAFT 提供的复制状态机，可以解决分布式系统的复制、修复、节点管理等问题Raft 极大的简化当前分布式系统的设计与实现，让开发者只关注于业务逻辑，将其抽象实现成对应的状态机即可。基于这套框架，可以构建很多分布式应用：

> 分布式锁服务
>
> 分布式存储系统，比如分布式消息队列、分布式块系统、分布式文件系统、分布式表格系统等，比如大名鼎鼎的 Redis 就是基于 Raft 实现分布式一致性
>
> 高可靠元信息管理，比如各类 Master 模块的 HA

二、 Raft基础

Raft 将一致性问题分解成了三个子问题：**选举 Leader日志复制安全性。**

在后续章节，会详细讲解这个子问题。现在，先了解一下 Raft 的一些核心概念。

2.1 服务器角色

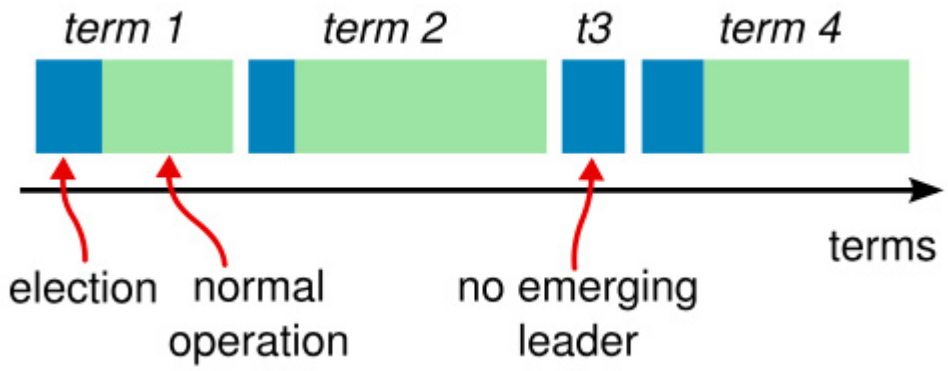
在 Raft 中，任何时刻，每个服务器都处于这三个角色之一 ：

> Leader - 领导者，通常一个系统中是一**主（Leader多从Follower**Leader 负责处理所有的客户端请求。
>
> Follower - 跟随者，不会发送任何请求，只是简单的 **响应来自 Leader 或者 Candidate 的请求**
>
> Candidate - 参选者，选举新 Leader 时的临时角色。

图示说明：

- Follower 只响应来自其他服务器的请求。在一定时限内，如果 Follower 接收不到消息，就会转变成 Candidate并发起选举。
- Candidate 向 Follower 发起投票请求，如果获得集群中半数以上的选票，就会转变为 Leader
- 在一个 Term 内，Leader 始终保持不变，直到下线了Leader 需要周期性向所有 Follower 发送心跳消息，以阻止 Follower 转变为 Candidate

2.2 任期



Raft 把时间分割成任意长度的 **任期（Term）**，任期用连续的整数标记。每一段任期从一次**选举**开始（Raft 保证了在一个给定的任期内，最多只有一个领导者。

> 如果选举成功（Leader 会管理整个集群直到任期结束。
>
> 如果选举失败，那么这个任期就会因为没有 Leader 而结束。

不同服务器节点观察到的任期转换状态可能不一样：

> 服务器节点可能观察到多次的任期转换。
>
> 服务器节点也可能观察不到任何一次任期转换。

任期在 Raft 算法中充当逻辑时钟的作用，使得服务器节点可以查明一些过期的信息（比如过期的 Leader）。每个服务器节点都会存储一个当前任期号，这一编号在整个时期内单调的增长。当服务器之间通信的时候会交换当前任期号。

> 如果一个服务器的当前任期号比其他人小，那么他会更新自己的编号到较大的编号值。
>
> 如果一个 Candidate 或者 Leader 发现自己的任期号过期了，那么他会立即恢复成跟随者状态。
>
> 如果一个节点接收到一个包含过期的任期号的请求，那么他会直接拒绝这个请求。

数据可视化的应用场景越来越广泛，数据可以呈现为更多丰富的可视化形式，使用户能够更加轻易、便捷的获取并理解数据传达的信息。

2.3 RPC

Raft 算法中服务器节点之间的通信使用 **远程过程调用（RPC）**。基本的一致性算法只需要两种 RPC：

> RequestVote RPC - 请求投票 RPC：由 Candidate 在选举期间发起。
>
> AppendEntries RPC - 附加条目 RPC：由 Leader 发起，用来复制日志和提供一种心跳机制。

三、选举Leader

3.1 选举规则

Raft 使用一种心跳机制来触发 Leader 选举。Leader 需要周期性的向所有 Follower 发送心跳消息，以此维持自己的权威并阻止新 Leader 的产生。

每个 Follower 都设置了一个**随机的竞选超时时间**，一般为 150ms ~ 300ms。如果在竞选超时时间内没有收到 Leader 的心跳消息，就会认为当前 Term 没有可用的 Leader。并发起选举来选出新的 Leader。开始一次选举过程。Follower 先要增加自己的当前 Term 号，并转换为 **Candidate**。

Candidate 会并行的向集群中的所有服务器节点**发送投票请求**。RequestVote RPC。它会保持当前状态直到以下三件事情之一发生：

> **自己成为 Leader**
>
> **其他的服务器成为 Leader**
>
> **没有任何服务器成为 Leader**
>
> 当一个 Candidate 从整个集群**半数以上**的服务器节点获得了针对同一个 Term 的选票，那么它就赢得了这次选举并成为 Leader。每个服务器最多会对一个 Term 投出一张选票，按照先来先服务。FIFO。的原则。要求半数以上选票的规则确保了最多只会有一个 Candidate 赢得此次选举。
>
> 一旦 Candidate 赢得选举，就立即成为 Leader。然后它会向其他的服务器发送心跳消息来建立自己的权威并且阻止新的领导人的产生。

等待投票期间。Candidate 可能会从其他的服务器接收到声明它是 Leader 的 AppendEntries RPC。

> 如果这个 Leader 的 Term 号（包含在此次的 RPC 中）不小于 Candidate 当前的 Term。那么 Candidate 会承认 Leader 合法并回到 Follower 状态。
>
> 如果此次 RPC 中的 Term 号比自己小，那么 Candidate 就会拒绝这个消息并继续保持 Candidate 状态。

如果有多个 Follower 同时成为 Candidate。那么选票可能会被瓜分以至于没有 Candidate 可以赢得半数以上的投票。当这种情况发生的时候，每一个 Candidate 都会竞选超时，然后通过增加当前 Term 号来开始一轮新的选举。然而，没有其他机制的话，选票可能会被无限的重复瓜分。

Raft 算法使用随机选举超时时间的方法来确保很少会发生选票瓜分的情况，就算发生也能很快的解决。为了阻止选票起初就被瓜分，竞选超时时间是一个**随机的时间**，在一个固定的区间（例如 150-300 毫秒）随机选择，这样可以把选举都分散开。

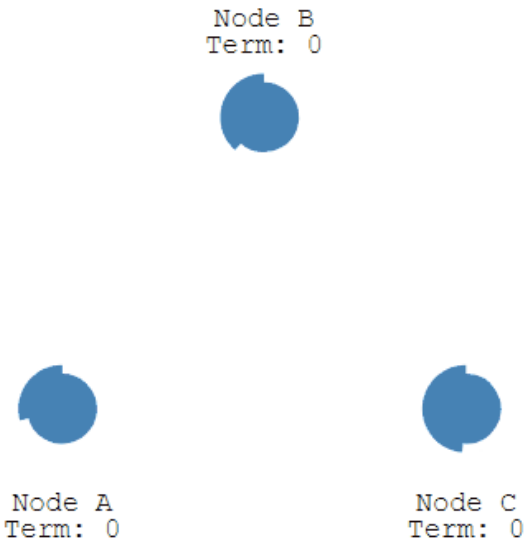
> 以至于在大多数情况下，只有一个服务器会超时，然后它赢得选举，成为 Leader。并在其他服务器超时之前发送心跳包。
>
> 同样的机制也被用在选票瓜分的情况下：每一个 Candidate 在开始一次选举的时候会重置一个随机的

> 选举超时时间，然后在超时时间内等待投票的结果；这样减少了在新的选举中另外的选票瓜分的可能性。

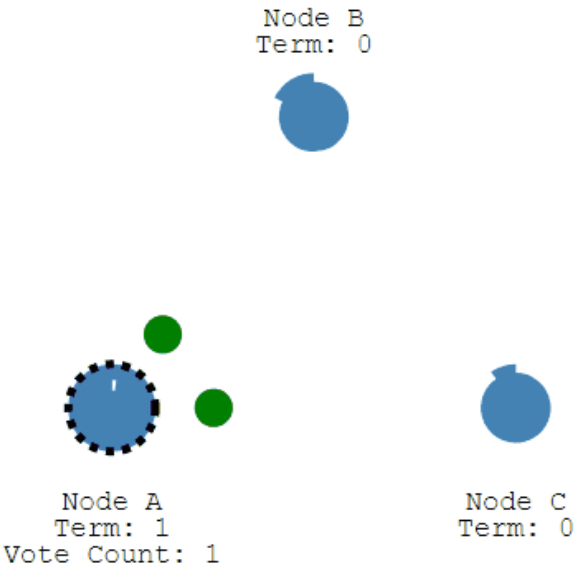
理解了上面的选举规则后，我们通过动图来加深认识。

3.2 单Candidate选举

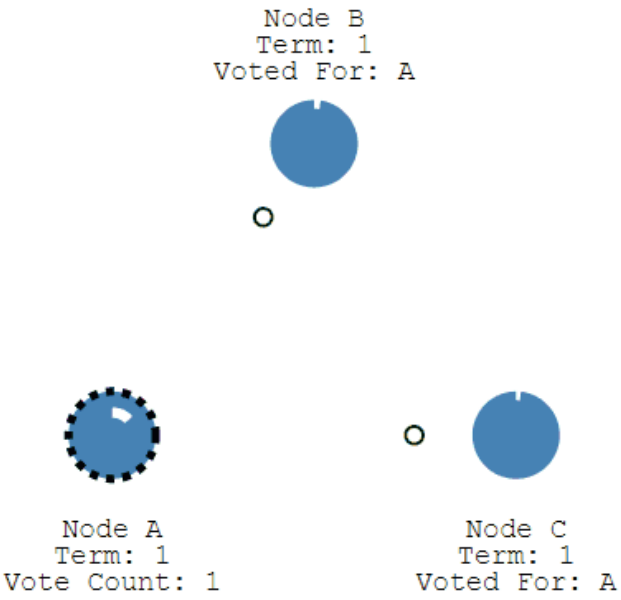
1. 下图表示一个分布式系统的最初阶段，此时只有 Follower没有 LeaderFollower A 等待一个随机的选举超时时间之后，没收到 Leader 发来的心跳消息。因此，将 Term 由 0 增加为 1，转换为 Candidate进入选举状态。



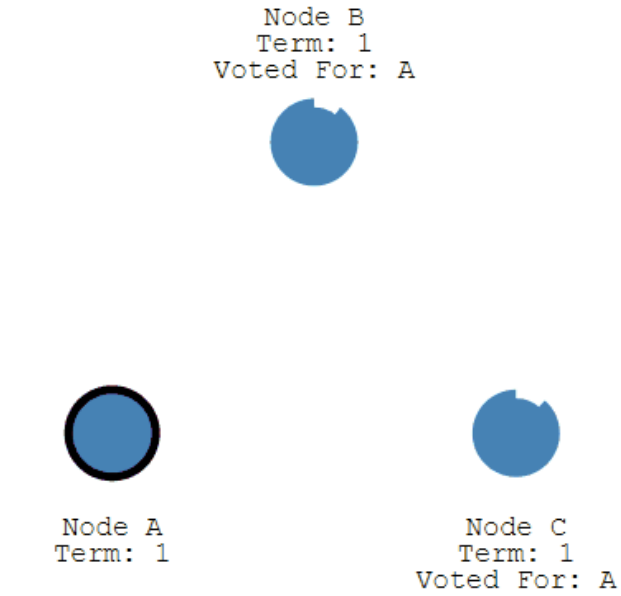
2)此时A 向所有其他节点发送投票请求。



1. 其它节点会对投票请求进行回复，如果超过半数以上的节点投票了，那么该 Candidate 就会立即变成 Term 为 1 的 Leader。

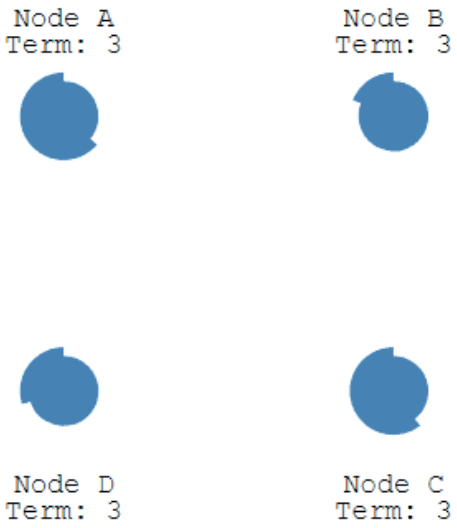


1. Leader 会周期性地发送心跳消息给所有 Follower。Follower 接收到心跳包，会重新开始计时。

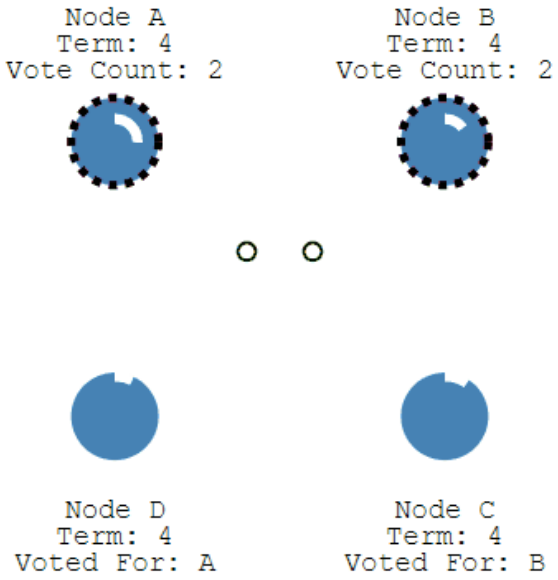


3.3 多 Candidate 选举

1. 1如果有多个 Follower 成为 Candidate，并且所获得票数相同，那么就需要重新开始投票。例如下图中 Candidate B 和 Candidate D 都发起 Term 为 4 的选举，且都获得两票，因此需要重新开始投票。



2) 当重新开始投票时，由于每个节点设置的随机竞选超时时间不同，因此能下一次再次出现多个 Candidate 并获得同样票数的概率很低。

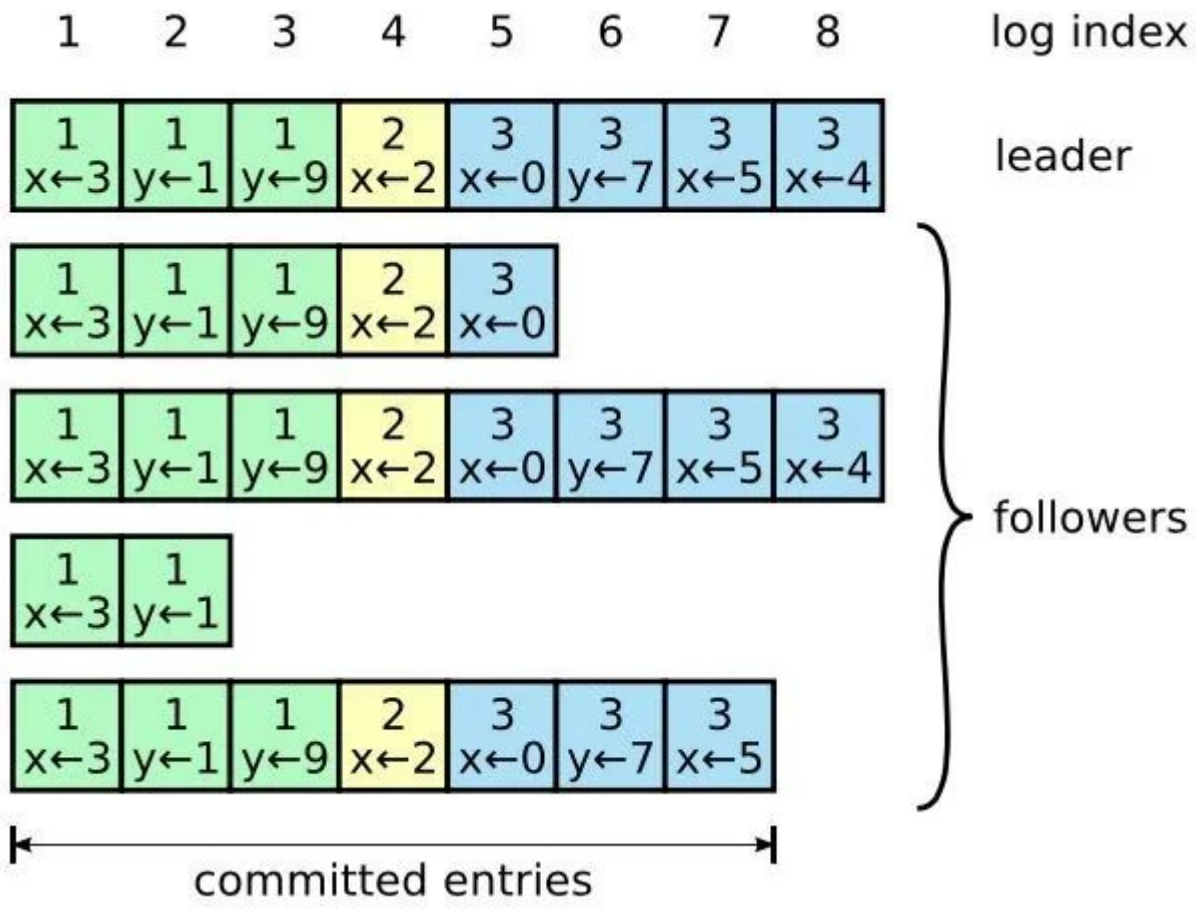


四、日志复制

4.1 日志格式

日志由含日志索引（log index）的日志条目（log entry）组成。每个日志条目包含它被创建时的 Term 号（下图中方框中的数字），和一个复制状态机需要执行的指令。如果一个日志条目被复制到半数以上的服务器上，就被认为可以提交（Commit）了。

> 日志条目中的 Term 号被用来检查是否出现不一致的情况。
>
> 日志条目中的日志索引（一个整数值）用来表明它在日志中的位置。



Raft 日志同步保证如下两点；图示说明：

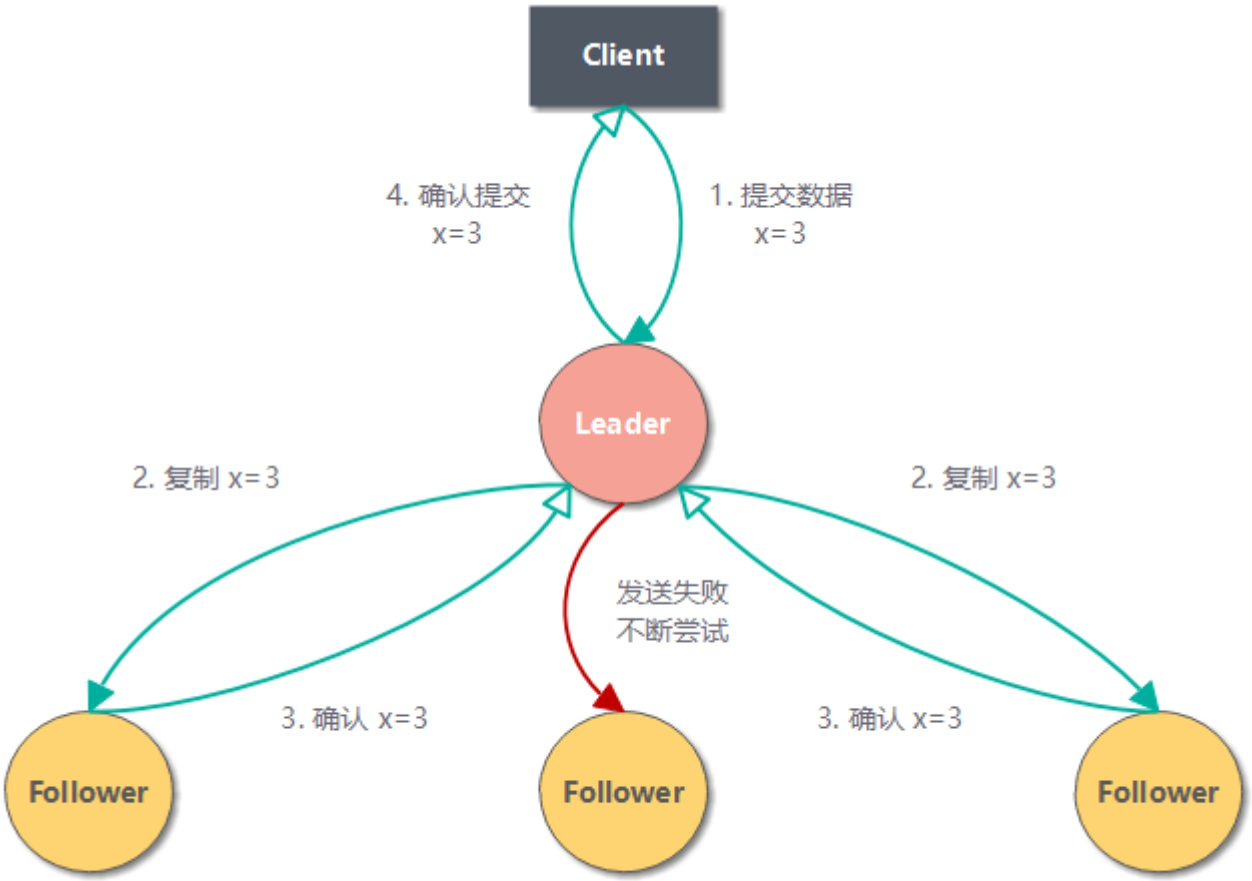
如果不同日志中的两个日志条目有着相同的日志索引和 Term，则**它们所存储的命令是相同的**。

> 这个特性基于这条原则：Leader 最多在一个 Term 内、在指定的一个日志索引上创建一条日志条目，同时日志条目在日志中的位置也从来不会改变。

如果不同日志中的两个日志条目有着相同的日志索引和 Term，则**它们之前的所有条目都是完全一样的**。

> 这个特性由 AppendEntries RPC 的一个简单的一致性检查所保证。在发送 AppendEntries RPC 时，Leader 会把新日志条目之前的日志条目的日志索引和 Term 号一起发送。如果 Follower 在它的日志中找不到包含相同日志索引和 Term 号的日志条目，它就会拒绝接收新的日志条目。

4.2 日志复制流程



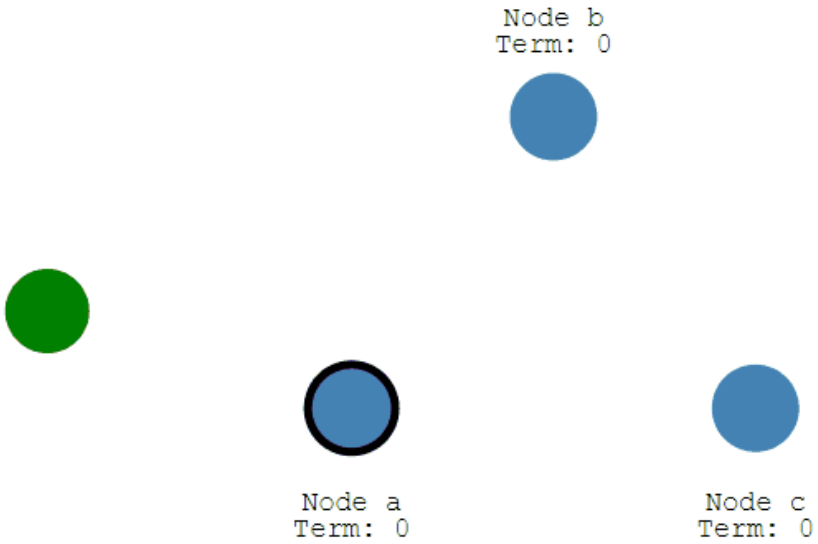
Raft 日志复制

> Leader 负责处理所有客户端的请求。
>
> Leader 把请求作为日志条目加入到它的日志中，然后并行的向其他服务器发送 AppendEntries RPC 请求，要求 Follower 复制日志条目。
>
> Follower 复制成功后，返回确认消息。
>
> 当这个日志条目被半数以上的服务器复制后，Leader 提交这个日志条目到它的复制状态机，并向客户端返回执行结果。

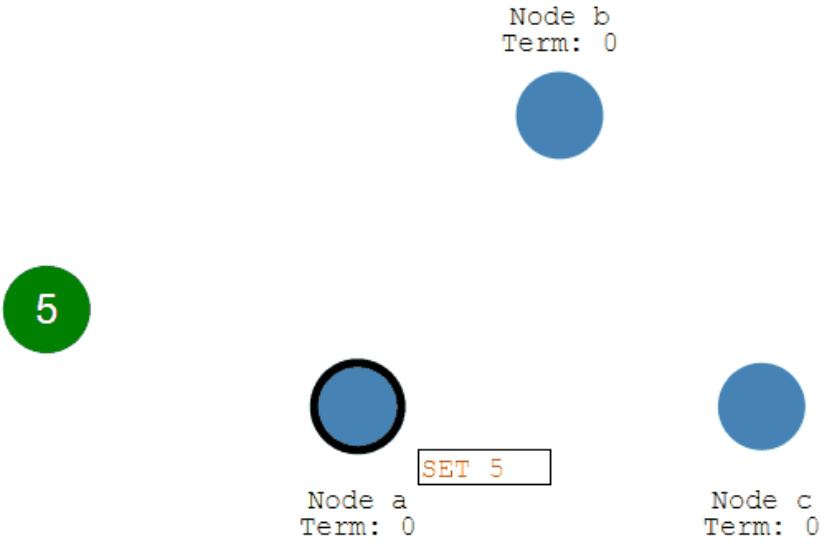
注意：如果 Follower 崩溃或者运行缓慢，再或者网络丢包，Leader 会不断的重复尝试发送 AppendEntries RPC 请求 （尽管已经回复了客户端），直到所有的跟随者都最终复制了所有的日志条目。

下面，通过一组动图来加深认识：

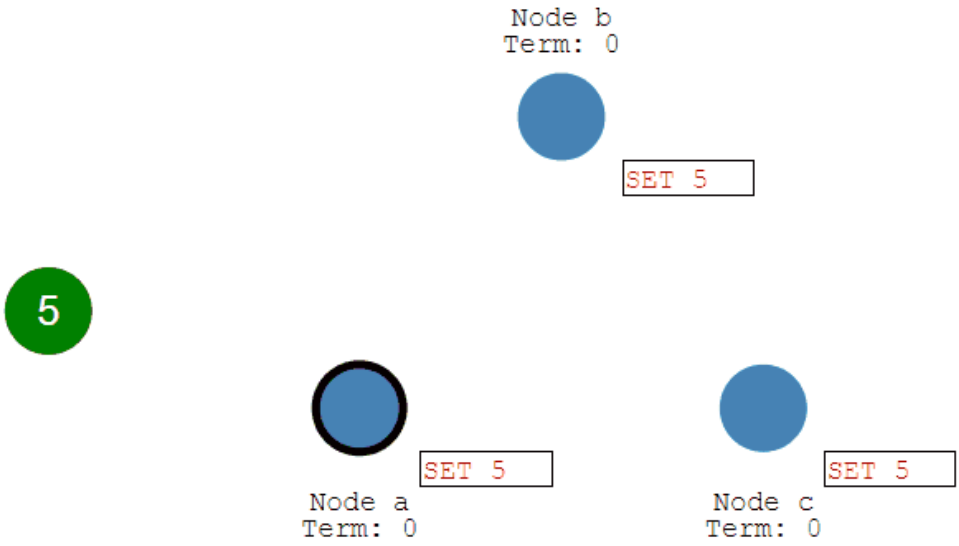
1)来自客户端的修改都会被传入 Leader，注意该修改还未被提交，只是写入日志中。



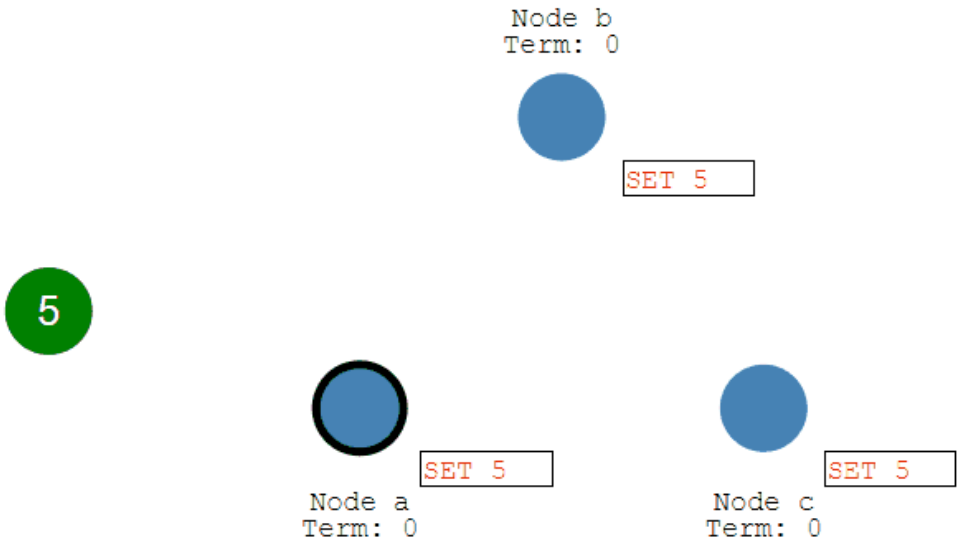
2)Leader 会把修改复制到所有 Follower



3)Leader 会等待大多数的 Follower 也进行了修改，然后才将修改提交。



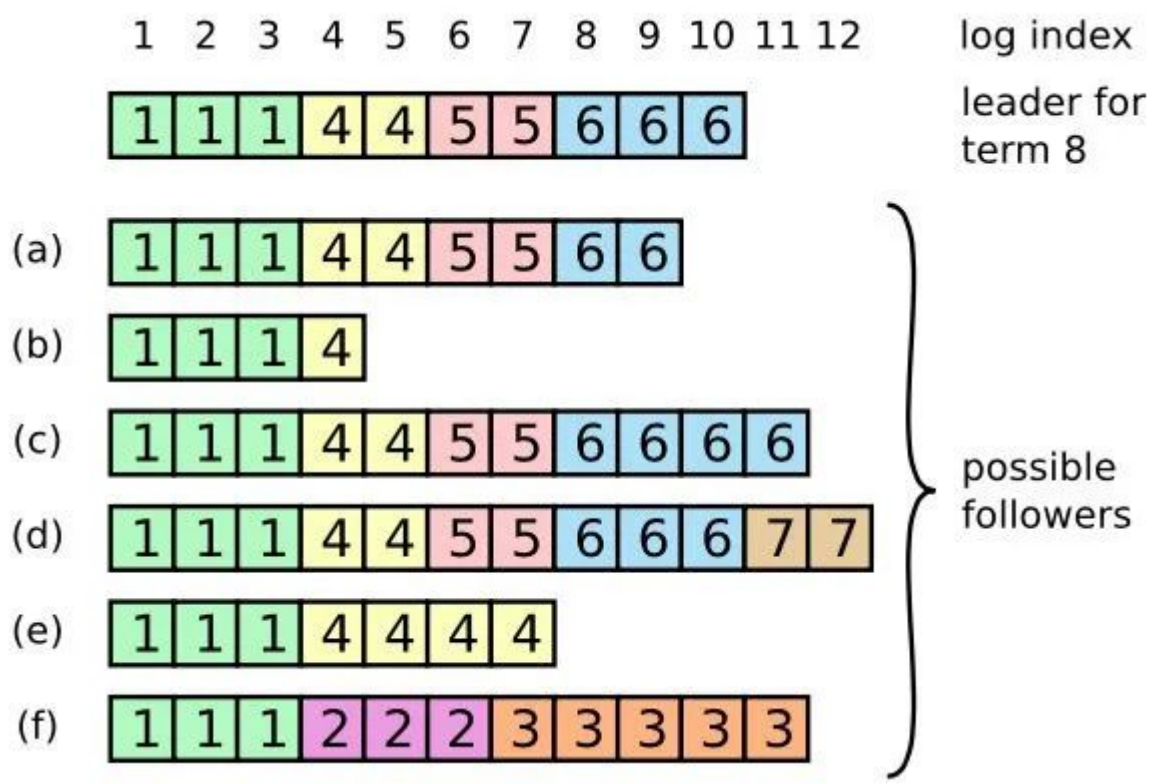
4)此时 Leader 会通知的所有 Follower 让它们也提交修改，此时所有节点的值达成一致。



4.3 日志一致性

一般情况下，Leader 和 Followers 的日志保持一致，因此日志条目一致性检查通常不会失败。然而，Leader 崩溃可能会导致日志不一致：旧的 Leader 可能没有完全复制完日志中的所有条目。

Leader 和 Follower 可能存在多种日志不一致的可能。



图示说明：上图阐述了 Leader 和 Follower 可能存在多种日志不一致的可能，每一个方框表示一个日志条目，里面的数字表示任期号 。

当一个 Leader 成功当选时，Follower 可能出现以下情况（a-f）：

> 存在未更新日志条目，如（a，b）；
>
> 存在未提交日志条目，如（c，d）；
>
> 或两种情况都存在，如（e，f）。

例如，场景 f 可能会这样发生，某服务器在 Term2 的时候是 Leader，已附加了一些日志条目到自己的日志中，但在提交之前就崩溃了；很快这个机器就被重启了，在 Term3 重新被选为 Leader，并且又增加了一些日志条目到自己的日志中；在 Term 2 和 Term 3 的日志被提交之前，这个服务器又宕机了，并且在接下来的几个任期里一直处于宕机状态。

Leader 通过强制 Followers 复制它的日志来处理日志的不一致，Followers 上的不一致的日志会被 Leader 的日志覆盖。

> Leader 为了使 Followers 的日志同自己的一致，Leader 需要找到 Followers 同它的日志一致的地方，然后覆盖 Followers 在该位置之后的条目。
>
> Leader 会从后往前试，每次日志条目失败后尝试前一个日志条目，直到成功找到每个 Follower 的日志一致位点，然后向后逐条覆盖 Followers 在该位置之后的条目。

五、安全性

前面描述了 Raft 算法是如何选举 Leader 和复制日志的。

Raft 还增加了一些限制来完善 Raft 算法，以保证安全性：保证了任意 Leader 对于给定的 Term，都拥有了之前 Term 的所有被提交的日志条目。

5.1 选举限制

拥有最新的已提交的日志条目的 Follower 才有资格成为 Leader。

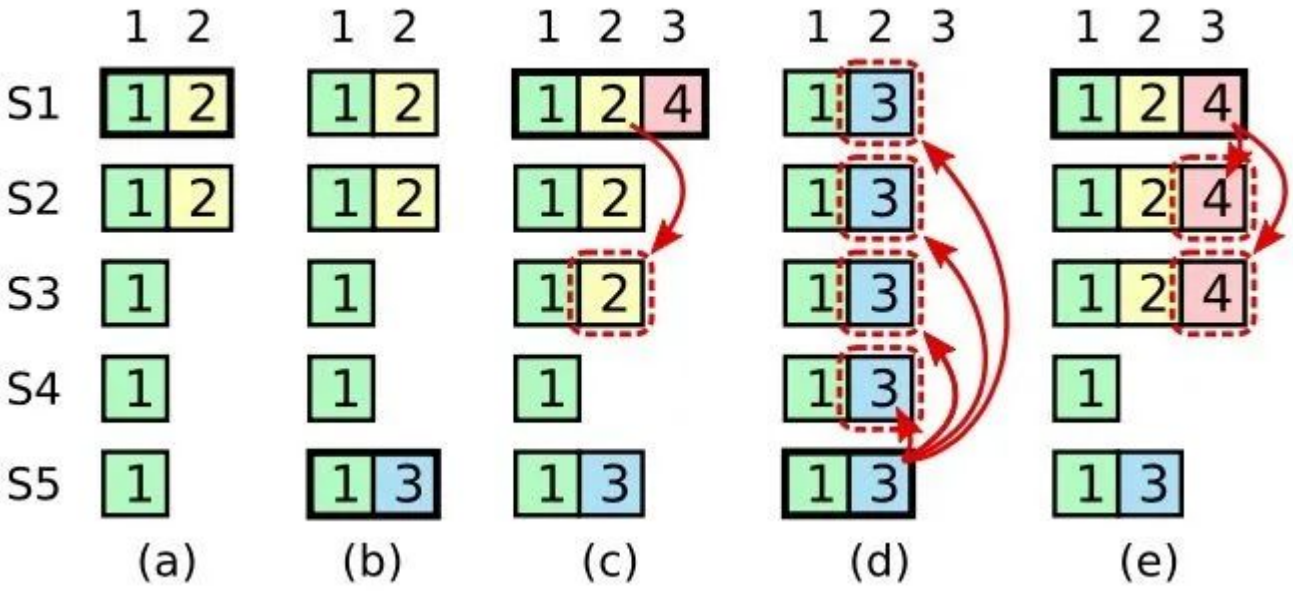
Raft 使用投票的方式来阻止一个 Candidate 赢得选举除非这个 Candidate 包含了所有已经提交的日志条目。Candidate 为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果 Candidate 的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么他一定持有了所有已经提交的日志条目。

RequestVote RPC 实现了这样的限制：RequestVote RPC 中包含了 Candidate 的日志信息：如果 Follower 会拒绝掉那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的日志索引和 Term 来判断哪个日志比较新。

> 先判断 Term：哪个数值大即代表哪个日志比较新。
>
> 如果 Term 相同，再比较 日志索引，哪个数值大即代表哪个日志比较新。

一个当前 Term 的日志条目被复制到了半数以上的服务器上：Leader 就认为它是可以被提交的。如果这个 Leader 在提交日志条目前就下线了，后续的 Leader 可能会覆盖掉这个日志条目。



图示说明：上图解释了为什么 Leader 无法对旧 Term 的日志条目进行提交。

- **阶段 (a)**：S1 是 Leader，且 S1 写入日志条目为 (Term 2，日志索引 2），只有 S2 复制了这个日志条目。
- **阶段 (b)**：S1 下线，S5 被选举为 Term3 的 Leader，S5 写入日志条目为 (Term 3，日志索引 2）。

- **阶段 (c)**：S5 下线，S1 重新上线，并被选举为 Term4 的 Leader，此时，Term 2 的那条日志条目已经被复制到了集群中的大多数节点上，但是还没有被提交。
- **阶段 (d)**：S1 再次下线，S5 重新上线，并被重新选举为 Term3 的 Leader，然后 S5 覆盖了日志索引 2 处的日志。
- **阶段 (e)**，如果阶段 (d) 还未发生，即 S1 再次下线之前，S1 把自己主导的日志条目复制到了大多数节点上，那么在后续 Term 里面这些新日志条目就会被提交。这样在同一时刻就同时保证了，之前的所有旧日志条目就会被提交。

Raft 永远不会通过计算副本数目的方式去提交一个之前 Term 内的日志条目。只有 Leader 当前 Term 里的日志条目通过计算副本数目可以被提交；一旦当前 Term 的日志条目以这种方式被提交，那么由于日志匹配特性，之前的日志条目也都会被间接的提交。

当 Leader 复制之前任期里的日志时，Raft 会为所有日志保留原始的 Term，这在提交规则上产生了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要重新复制之前的任期里的日志时，它必须使用当前新的任期号。

Raft 使用的方法更加容易辨别出日志，因为它可以随着时间和日志的变化对日志维护着同一个任期编号。另外，和其他的算法相比，Raft 中的新领导人只需要发送更少日志条目（其他算法中必须在他们被提交之前发送更多的冗余日志条目来为他们重新编号）。

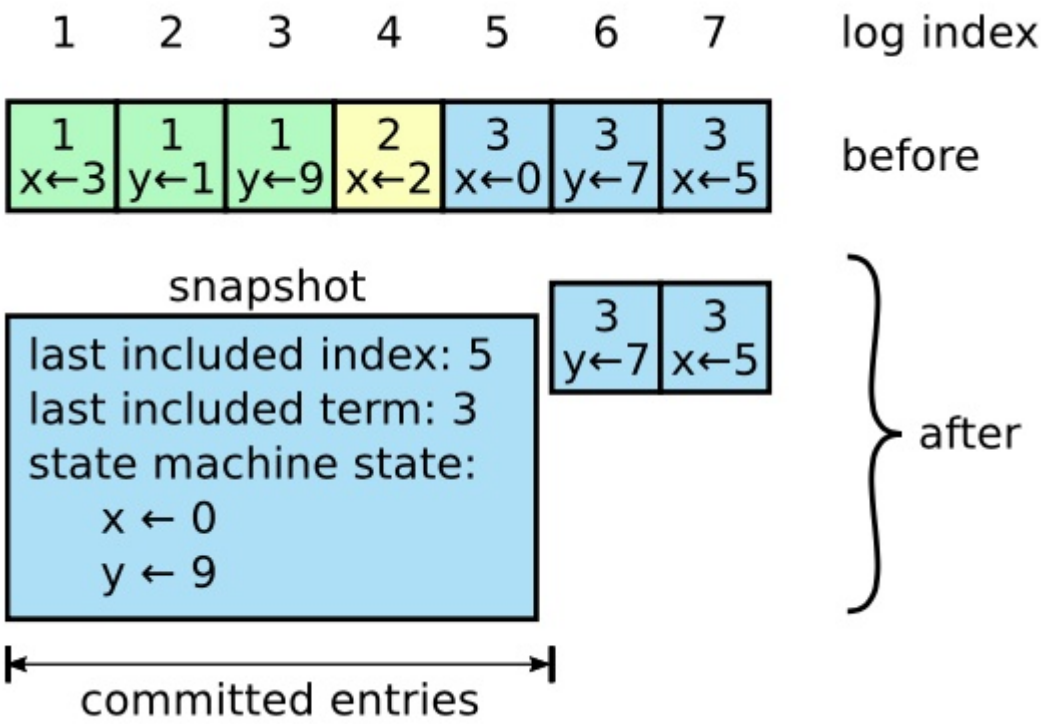
六、日志压缩

在实际的系统中，不能让日志无限膨胀，否则系统重启时需要花很长的时间进行恢复，从而影响可用性，Raft 采用对整个系统进行快照来解决，快照之前的日志都可以丢弃。

每个副本独立的对自己的系统状态生成快照，并且只能对已经提交的日志条目生成快照。快照包含以下内容：

> 日志元数据。最后一条已提交的日志条目的日志索引和 Term，这两个值在快照之后的第一条日志条目的 AppendEntries RPC 的完整性检查的时候会被用上。
>
> 系统当前状态。

当 Leader 要发送某个日志条目，落后太多的 Follower 的日志条目会被丢弃，Leader 会将快照发给 Follower，或者新上线一台机器时，也会发送快照给它。



生成快照的频率要适中，频率过高会消耗大量 I/O 带宽；频率过低，一旦需要执行恢复操作，会丢失大量数据，影响可用性。推荐当日志达到某个固定的大小时生成快照。生成一次快照可能耗时过长，影响正常日志同步。可以通过使用 copy-on-write 技术避免快照过程影响正常日志同步。

说明：本文仅阐述 Raft 算法的核心内容，不包括算法论证、评估等

七、参考资料

1.Raft 一致性算法论文原文

2.Raft 一致性算法论文译文

3.Raft 作者讲解视频

4.Raft 作者讲解视频对应的 PPT

5.分布式系统的 Raft 算法

6.Raft 算法详解

7.Raft: Understandable Distributed Consensus

8.sofa-jraft - 蚂蚁金服的 Raft 算法实现库□Java 版