Raft 算法

Raft 算法将一致性问题分解为 Leader 选举、日志同步、成员变更 等一系列子问题，Raft 算法相较于 Paxos 算法理解起来更加通俗易懂。

Raft 算法将节点分为三种角色：

1. Leader、领导者：负责接受客户端的写请求，并将请求日志同步给 Follower 追随者，当大多数 Follower 成功同步后，则允许 Follower 提交日志。
2. Follower、追随者：允许接受读请求，负责接受 Leader 的同步日志，在 Leader 告知日志可以提交后，提交日志。
3. Candidate、候选者：当检查到 Leader 下线时发起 Leader 选举竞选 Leader。

Leader 选举

动画演示：Raft Consensus Algorithm

Raft 算法保证任一任期内只能有一个领导者，领导者是 Follower 一致选举出来的。

作为 Leader，除了同步请求日志之外，还必须要周期性的向所有 Follower 发送心跳包，每个 Follower 都维护者一个定时器（超时时间是随机的），一旦指定时间内未收到 Leader 的心跳包，则认为 Leader 故障下线，Follower 作为 Candidate 发起选举竞选 Leader：

1. Candidate 任期自增，首先投票给自己，然后发送投票信息给其他所有节点。
2. 节点收到投票信息后，取出投票信息中 Candidate 最新的日志，日志条目由 任期 + 编号 组成，节点会将 Candidate 最新的日志与自身最新的日志进行对比，如果 Candidate 日志没有自己的新，则投反对篇，比较的规则是先比较任期，任期相同再比较编号。
3. 如果 Candidate 的日志确实新与自身，则比较 Candidate 任期和自身任期，只有当 Candidate 的任期大于自身，节点才会投赞成票，并将自身任期更改为 Candidate 的任期。
4. 当 Candidate 收到超过一半节点数目的赞成票后，Candidate 晋升为 Leader，开始定时发送心跳包。

5. 在任意时刻，一旦节点收到来自 **任期大于等于自身** 的 Leader 的心跳包，节点将成为该 Leader 的 Follower，并更新自身的任期为 Leader 的任期，同时刷新超时计时器。

要注意即使节点成为 Candidate，节点依然会维护超时计时器以期望收到 Leader 的消息，如果超时，将继续自增任期重复上述步骤。

在上述流程中，由于每个节点一旦投赞成票就会更改任期追随 Candiate，一旦任期更改，节点无法继续投给 Candidate 赞成票，所以隐含的规则是：

- 任意节点只能投给任意一个任期 Candidate 一票。

又因为对于任意一个 Candidate 而言，要想成为 Leader，就必须要有半数以上的赞成票，所以这两个规则保证了：**任意一个任期内，只会有一个 Leader。**

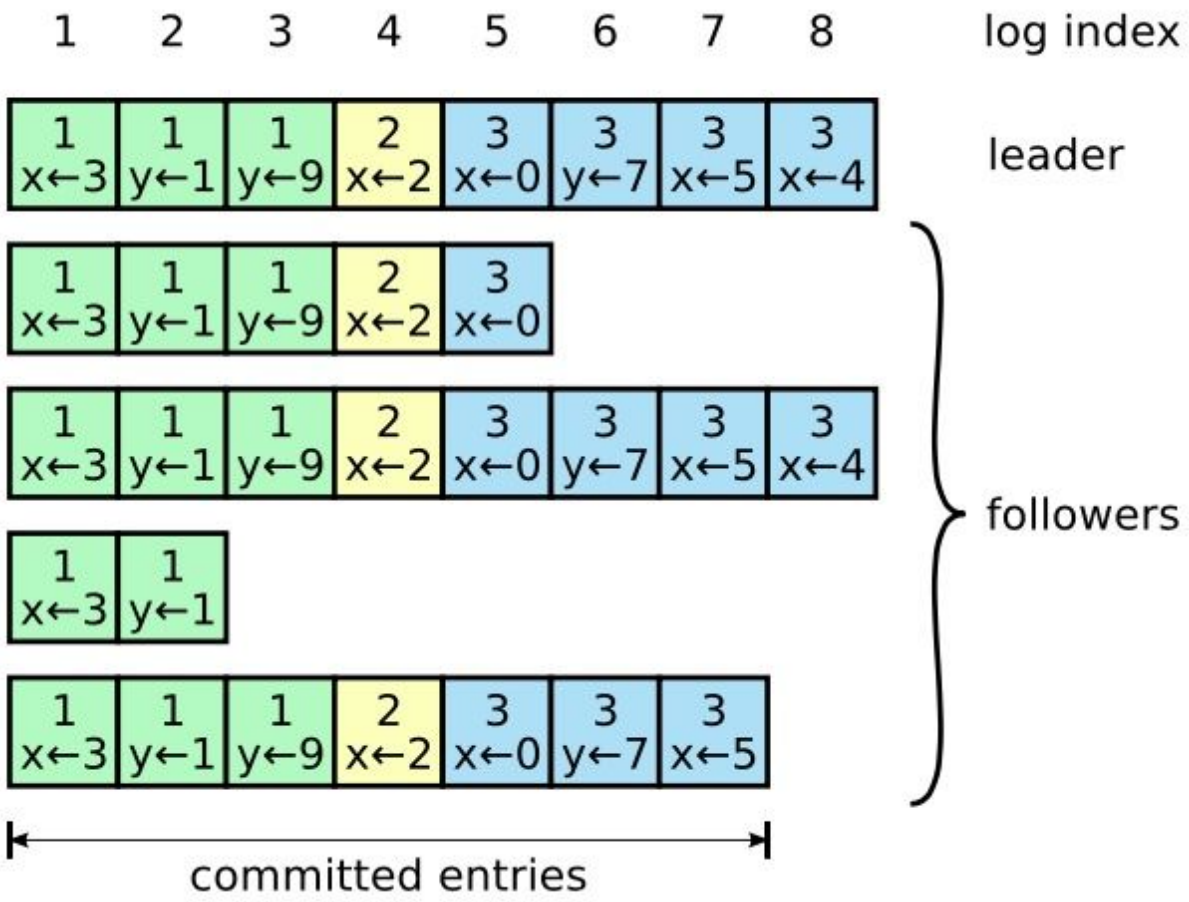
任期是为了防止 **脑裂** 的情况发生：某个 Leader 由于网络原因被隔绝，此时重新竞选 Leader，而后网络又恢复，这样就会存在多个 Leader。有了任期的概念，这个问题很好解决：**只服务任期等于自己的 Leader**，一旦收到任期大于自身的 Leader 的消息，节点会更新任期追随此 Leader（旧 Leader 就会成为 Follower 追随新 Leader）。

在上述流程中，第二点比较日志的新旧其实是为了防止这样一种情况发生：某个节点由于网络问题被隔绝，于是它收不到任何消息，将会不断自增任期并发起选举，于是此节点的任期将会非常大，但是数据可能是陈旧的。

所以 Raft 算法将日志新旧也作为投票的规范，**这意味着一个 Leader 中的数据至少等于或新于大多数节点的数据。**

日志同步

Leader 将客户端的请求作为日志条目加入到它的日志中，日志由有序编号的日志条目组成，每个日志条目都包含对应的任期。



Leader 会向所有 Follower 发起 RPC 广播日志条目，但为了保存一致性，只有当多数 Follower 回复 RPC 成功后，Leader 才会发起 commit 让 Follower 提交日志。

由于 Leader 崩溃，Follower 上的日志可能多于新 Leader 也可能少于新 Leader，这完全取决于 **多数节点的状态**，为保证一致性，Raft 算法一切以 Leader 为准，即 Leader 的日志可以覆盖 Follower 上的日志，这意味着最新的日志信息是可能被丢失的。

具体的做法：

1. 发生日志条目时，Leader 还会附加上这条日志条目的上一个日志条目的编号 prevIndex 和任期号 prevTerm，当 Follower 收到 RPC 信息时，它会检查本地日志中编号为 prevIndex 的日志条目任期是否为 prevTerm，如果相同，则将最新的信息覆盖在对应编号（prevIndex + 1）位置上。
2. 如果不同，节点返回一个错误码，当 Leader 收到错误码时，Leader 从后往前开始同步每一条日志，直到某条日志同步成功，从这条日志开始同步的后续的日志。这就是 Leader 强制性覆盖 Follower 日志的一致性检查的过程。

上述流程第二点的一致性检查保证了如下规则成立：

- index 和 term 都相同的两个日志条目存储的命令一定是相同的。

这是因为按照一致性检查的要求，Follower 中的日志最终会和 Leader 中的日志保持一致性，而一个 Leader 任意任期内在同一 index 内最多只会产生一条条目。

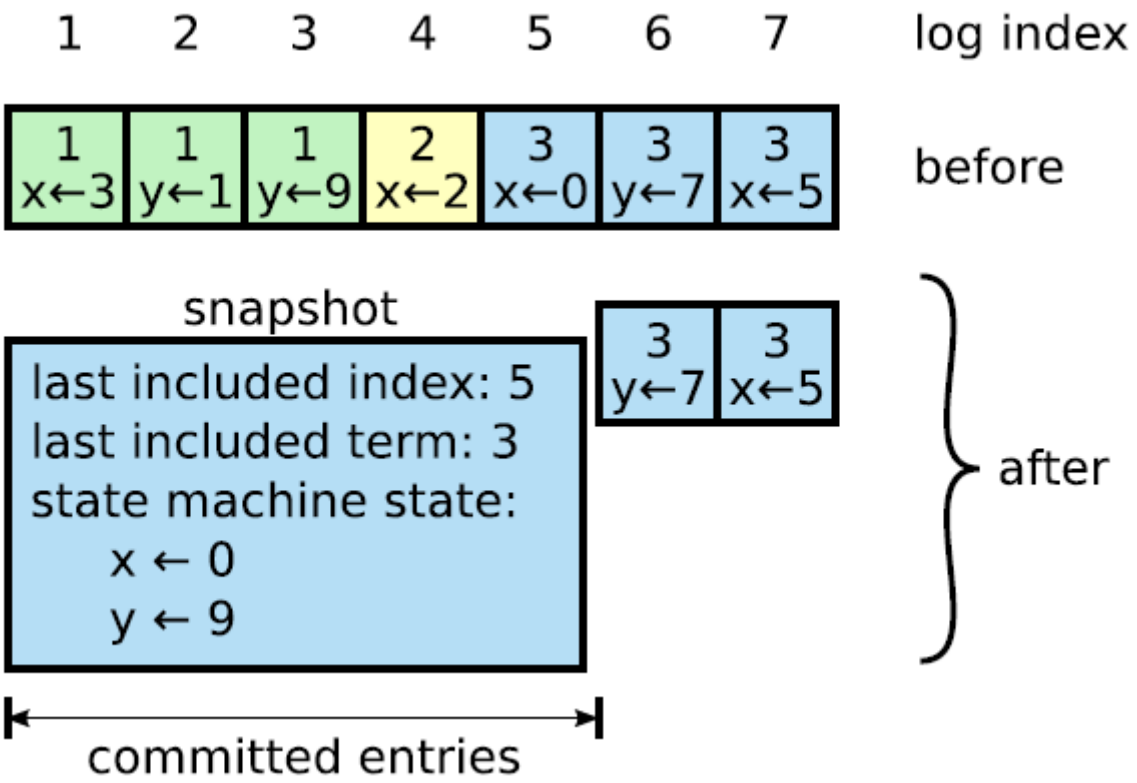
通过这种方式 Raft 能够保证 Leader 和 Follower 的日志一致性。

日志压缩

当新的 Follower 上线或者 Follower 丢失太多日志时，重放日志成本过高，因此 Raft 算法采用 Snapshot（快照）解决这个问题。

Snapshot 表示的是 Leader 的系统状态，日志是基于命令的，而 Snapshot 是 Leader 的系统状态，可以理解为数据，就好比 RDB 和 AOF 的区别，发送 Snapshot 使用 InstalledSnapshot RPC。

传输 Snapshot 会比传输日志命令快得多。



Leader 会不定期的对已提交的日志生成快照，快照包含如下内容：

- 最后的 log index
- 最后的 log term
- 数据

当 Follower 新上线或丢失过多消息时，Leader 会选择发送最新的 Snapshot 给 Follower，当 Follower 收到 Snapshot 时，如果当前日志旧与 Snapshot 中的最后的日志编号，那么 Follower 会全部替换 Snapshot 并丢弃所有日志；当然，Follower 的日志可能也会比 Snapshot 新，这种情况下只会丢弃 last index 之前的日志，并保留之后的日志。

在 Snapshot 同时，Leader 会并发的接受新消息，可以采用 Linux 自带的 CopyOnWrite 技术实现并发，利用 fork 函数在子进程内发送 Snapshot，而父进程继续接受新消息。

Follower 也需要不定期 Snapshot，因为 Follower 在未来的某一刻可能会成为 Leader。

节点再平衡

成员变更可能会造成最终选举出现两个 Leader，例如有两个节点 A、B 正在同时竞选，此时 A 已经获得了多数节点的赞成，而就在这一时刻，新增两个节点，由于延迟问题，A 不知道新增两个节点，于是 A 仍然认为自己成为了 Leader，而 B 发现自增两个节点，于是发送投票给他们，这两个节点都投了 B，现在 B 成为了多数派，B 也将成为 Leader！

如果我们严格控制每次新增或删除的节点数只有一个的话，那么我们是允许采用一般性的一致性解决方案，这意味着在上一次成员变更尚未完成的情况下，不允许接下来的成员变更。

因此我们可以按照一般化的方案解决：

- 同一时刻只允许一个节点变更，并且上一次变更未完成不允许下一次变更。
- 成员变更由 Leader 发起作为一条特殊的日志记录同步给 Follower。
- 当多数 Follower 确认后，成员变更成功，Leader 变更成员状态，同时 Leader 发送 commit 给 Follower。
- Follower 变更成员状态。

这是基于一个简单的数学道理，设总节点数为 N：

如果 N 是奇数：

$$\text{多数节点数目} = \frac{N+1}{2} \text{新增一个节点后多数节点数目} = \frac{N+2}{2}$$

这意味着如果 B 原本还差一个赞成票的话，即当前赞成票 $ P = \frac{N}{2}$，即使新增一个节点也无法让 B 得到多数赞成 $\frac{N + 2}{2}$ 。

偶数同样分析。