# Raft

## 引言

Raft是一种用来管理日志复制的协议，它和Paxos协议的功能是一样的，并且在性能上和Paxos相差无几。相信了解过Paxos协议的人都会觉得其晦涩难懂，工程实现和协议相差甚远。而Raft算法的核心设计理念就是简单，并且更易于建立实际的系统。Raft算法为了更便于理解，将问题拆分为Leader选举、日志复制、安全性这几个部分。还提供了动态改变集群成员的机制。

## 复制状态机

一致性协议都是在复制状态机的背景下提出的。基于这样一个理论：一组初始状态相同的状态机，如果按照相同顺序执行一批命令，那么这一组状态机的最终结果一定相同。通俗来讲，可以认为分布式系统中的每个节点内部有一个独立的状态机，每个节点通过执行自己日志中的命令，来达到系统中各个节点的一致性(如图1)。使用复制状态机的例子有Chubby和ZooKeeper等。Raft算法就是解决分布式系统中节点日志一致性问题的。

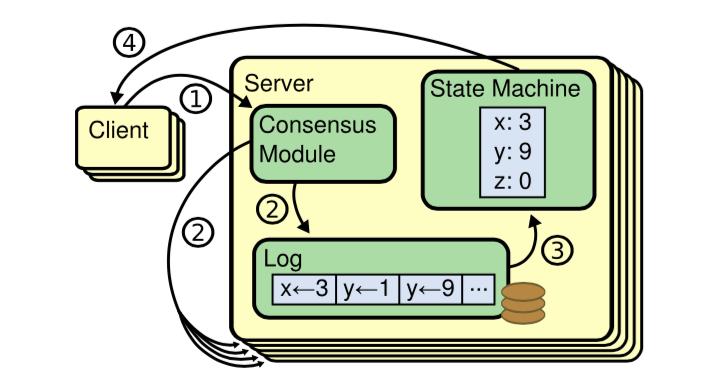


图1

## Paxos算法的不足

Paxos算法一直被认为分布式一致性算法中的鼻祖，无论是Raft算法还是其他的分布式协议，或多或少都借鉴了Paxos算法。

但是Paxos有两个缺点：

1. Paxos算法的晦涩难懂是一直以来被诟病的问题。
2. 难以在实际环境中实现，其中一个原因是对于多决策，大家还没有一个一致同意的算法。Lamport的描述大部分都是有关于单决策；他仅仅描述了实现多决策的可能的方法，缺少许多细节。有许多实现 Paxos 和优化 Paxos 的尝试，但是他们都和 Lamport 的描述有些出入。例如，Chubby 实现的是一个类似 Paxos 的算法，但是在许多情况下的细节没有公开。

还有一点是Paxos使用点对点的实现作为它的核心(尽管提出了一种弱Leader的形式来优化性能)，这种方法通常比较适用于那些只需要一个决策被制定的情况下。如果分布式系统需要做很多决策，那么通过选择一个Leader，由Leader来协调是更简单高效的方法。

## Raft一致性算法

Raft是用来管理日志复制一致性的算法，通过选出一个Leader，由Leader完全负责管理复制日志的责任。Leader接收客户端的命令，并将命令转换为日志条目后复制到其他服务器上，Leader还需要告诉Follower什么时候将日志条目应用到状态机是安全的。在系统中Leader拥有对日志的绝对管理权，能够决定将新的日志条目放到哪，并不需要和其它服务器协商，这样能够简化复制日志的流程，当Leader宕机或者与其它服务器无法连接时，需要选取下一个Leader。接下来我们就来了解一下Raft算法。

### 角色

在Raft算法中定义了三种角色，分别为Leader、Follower和Candidate，在任意时间每个服务器一定会处于三种角色中的一个。正常情况下，只有一台服务器是Leader，剩下的服务器都是Follower。

* Leader：分布式系统中的管理者，负责处理客户端的读写请求，将写请求转换为日志条目复制到Follower上。
* Follower：Leader服务器的跟随者，转发写请求给Leader，通常为了性能考虑而允许Leader与Follower存在弱一致性的情况，会允许Follower处理读请求。接收Leader服务器发送过来的日志条目。当机器刚启动时默认为Follower，如果收到Leader的请求则与Leader建立连接，或经过超时时间后，转换为Candidate。
* Candidate：当系统中的节点经过超时时间后仍无法与Leader通信，则会转换为Candidate，发起Leader选举，根据投票结果成为Leader或者Follower。

图2展示了三种角色之间的转换。

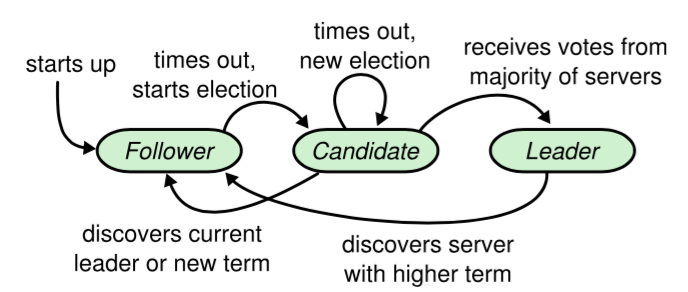


图2

简单描述一下图2：

1. 服务器启动后默认处于Follower角色，当发现Leader后与其保持跟随。
2. 当Follower经过超时时间后，转换为Candidate，开始选举，如果获得了大多数服务器的投票，则成为Leader；如果经过超时时间后仍没有获得大多数服务器的投票，则开始新一轮的选举；如果收到来自Leader的RPC请求或者收到了更高term的Candidate发来的投票请求，则转换为Follower与其保持跟随。
3. 当Leader发现更高term的服务器后，转换为Follower与其保持跟随。

### 任期term

Raft将时间分隔到任意长度的term中，每一个任期开始于选举，在选举成功后，Leader将负责管理集群直到任期结束(比如Leader宕机)，如果选举失败(比如选票被两个服务器平分)，则该任期就会因为没有Leader而结束，然后开始下一任期。

在某些情况下一台服务器可能看不到一次选举或者一个完整的任期。任期在Raft中充当逻辑时钟的角色，并且允许服务器检测过期信息，比如过时的Leader。每一台服务器都存储着当前任期的数字，这个数字会单调递增，当服务器之间进行通信时，会互相交换任期号，如果一个服务器的当前任期号比其它服务器的小，那么则更新为较大的任期号。如果一个Leader或者Candidate意识到自己的任期号过时了，那么就会转换为Follower。如果一台服务器接收到的请求中任期号是过时的，那么就会拒绝该请求。

### 状态

在Raft算法中，每个节点都有自己的一些状态，如下所示：

|  |  |
| --- | --- |
| 在所有服务器上稳定的状态(在响应RPCs请求前稳定存储的) | |
| currentTerm | 服务器已经知道的最新的任期（初始化为0在第一次启动时，单调增加） |
| votedFor | 在当前任期内收到选票的候选人 id（如果没有就为 null） |
| log[] | 日志条目集合，每个日志条目都包含状态机要执行的命令和产生该日志条目的任期 |

|  |  |
| --- | --- |
| 在所有服务器上不稳定的状态 | |
| commitIndex | 已知的被提交的最大日志条目索引值(从0开始，单调递增) |
| lastApplied | 被状态机执行的最大日志条目索引值(从0开始，单调递增) |

|  |  |
| --- | --- |
| Leader上不稳定的状态(在选举后重新初始化) | |
| nextIndex[] | 对于每一个服务器，记录需要发给它的下一个日志条目的索引（初始化为领导人上一条日志的索引值+1） |
| matchIndex[] | 对于每一个服务器，记录已经复制到该服务器的日志的最高索引值（从0开始递增），通常在收到该服务器的ack后递增。 |

### 通信

Raft算法中各个服务器间通过RPC进行通信，基本的Raft算法只需要两种RPC，AppendEntries RPC是Leader触发的，负责复制日志和心跳消息；RequestVote RPC是由Candidate在选举期间触发的，用来进行选举。还有第三种RPC用来在服务器之间传输快照。

#### AppendEntries RPC

附加日志的RPC请求与响应，Leader用来复制日志和发送心跳。

请求：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | Leader的任期 |
| leaderId | Leader的 id，为了其他服务器能重定向到客户端 |
| prevLogIndex | 当前发送的日志条目的前一条日志条目的索引 |
| prevLogTerm | 当前发送的日志条目的前一条日志条目的任期 |
| entries[] | 当前发送的日志条目集合，为了效率可能会包含多条日志，心跳请求时为空 |
| leaderCommit | Leader已经提交了的日志条目的索引 |

响应：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | currentTerm，用于Leader更新自己的任期 |
| success | 如果Follower服务器包含能够匹配上 prevLogIndex 和 prevLogTerm 的日志时为true |

接收者需要实现：

1. 如果term< currentTerm，返回false。
2. 如果在prevLogIndex处的日志的任期号与prevLogTerm不匹配时，返回 false。
3. 如果一条已经存在的日志与新的冲突（index 相同但是任期号 term 不同），则删除已经存在的日志和它之后所有的日志。
4. 添加任何在log[]中不存在的日志条目。
5. 如果leaderCommit>commitIndex，那么将commitIndex更新为leaderCommit和“最新日志条目的索引”中较小的一个，即commitIndex=MIN(leaderCommit, 最新一条日志条目的索引)。

这里解释一下第5条，因为如果leaderCommit>commitIndex，说明Follower提交的日志条目落后于Leader，需要更新Follower的commitIndex，之所以在leaderCommit和“最新一条日志的索引”选择较小的一个是因为Follower可能接收到的日志条目落后于leaderCommit，这时对于Follower来说leaderCommit索引处的日志条目是不存在的，自然无法提交，所以只能将commitIndex更新为自己最新的日志条目的索引。反之则将commitIndex更新为leaderCommit。

#### RequestVote RPC

上面了解了复制日志的RPC请求与响应，接下来再看一下Leader选举投票的RPC请求与响应。

请求：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | candidate的任期 |
| candidateId | candidate的Id |
| lastLogIndex | 最后一条日志的索引 |
| lastLogTerm | 最后一条日志的任期 |

响应：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | currentTerm，用于candidate更新自己的任期 |
| voteGranted | true表示candidate获得投票 |

接收者需要实现：

1. 如果term<currentTerm，返回false
2. 如果votedFor为null或者与candidateId相同，并且candidate的日志至少跟接收者一样新，那么candidate获得投票，返回true。

### 服务器需要遵守的规则

Raft算法定义了分布式系统中服务器需要遵守的规则，不同角色的服务器需要遵守不同的规则，才能确保分布式系统的一致性。

#### 所有服务器的规则

* 如果commitIndex>lastApplied，那么自增lastApplied，然后将log[lastApplied]应用到状态机。
* 如果RPC请求或者响应中的term T>currentTerm，那么设置currentTerm=T，然后将自己转换为Follower。

#### Followers

* 响应来自Leader和Candidate的RPC请求。
* 如果超过“选举超时时间”还没有收到来自Leader的AppendEntries RPC请求或者没有给candidate投票，则转换为candidate。

#### Candidates

* 转换为candidate后，开始选举

1. 自增currentTerm
2. 投票给自己
3. 重置选举超时时间
4. 发送RequestVote RPC请求给其它所有服务器

* 如果获得超过半数的投票，那么自己就转换为Leader。
* 如果收到了来自新Leader的AppendEntries RPC请求，那么将自己转换为Follower。
* 如果选举时间超过了“选举超时时间”，则开始新一轮的选举。

#### Leader

* 一旦成为Leader就发送空的AppendEntries RPC请求(心跳)给其它所有服务器，并且需要在空闲时间重复发送，以防止其它服务器选举超时。
* 如果收到了来自客户端的写请求，则将写请求转换为日志条目写入到日志，在该日志条目应用到状态机之后响应客户端。
* 如果Leader的最后一条日志的索引大于Follower的nextIndex，则将从nextIndex位置开始之后所有的日志条目通过AppendEntries RPC请求发送给该Follower。

1. 如果Follower返回成功，则更新该Follower的nextIndex和matchIndex。
2. 如果因为日志不一致而返回失败，则递减nextIndex后重试。

* 如果存在一个N, N>commitIndex，并且大多数match[i]≥N,并且log[N].term==currentTerm，那么更新commitIndex=N。这条规则简单来说就是当一条在当前Leader周期提出的日志被超过半数的Follower收到后，就可以提交这条日志了。后面会解释为什么是当前Leader周期提出的日志条目。

### Raft提供的保证

* **选举安全性**：同一任期最多有一个Leader。
* **Leader只能追加：**Leader绝不覆盖或者删除它自己的日志条目，只能追加日志条目。
* **日志匹配：**如果两条日志拥有相同的index和term，那么这两条日志完全相同，并且从日志开头到这两条日志的index处的所有日志条目也完全相同。
* **Leader完整性：**如果一条日志在给定的term被提交了，那么这条日志一定会出现在更高term的Leader的日志中。
* **状态机安全性：**如果一台服务器将某个索引位置的日志条目应用到了状态机，那么其他服务器不会在相同索引位置应用不同的日志条目到状态机。

### Raft算法深入分析

我们已经了解了Raft算法的一些基本概念，接下来就来深入了解一下Raft算法。在上面我们提到了Raft算法为了方便理解，基础Raft算法主要分为Leader选举、日志复制和安全性三个部分，下面就会从这三个方面来深入了解Raft算法。

为了方便理解，我们假设有一个由五台服务器组成的分布式系统，这样一个系统最多允许两台服务器宕机(Leader选举的过半原则)。

#### Leader选举