



Raft算法详解



祥光

分布式 / 存储 / 数据库 / 区块链

803 人赞同了该文章

收起

[Paxos算法详解](#)一文讲述了晦涩难懂的Paxos算法，以可理解性和易于实现为目标的Raft算法极大的帮助了我们的理解，推动了分布式一致性算法的工程应用，本文试图以通俗易懂的语言讲述Raft算法。

算法概述

Leader选举

同步

性能

日志压缩

成员变更

与Multi-Paxos的异同

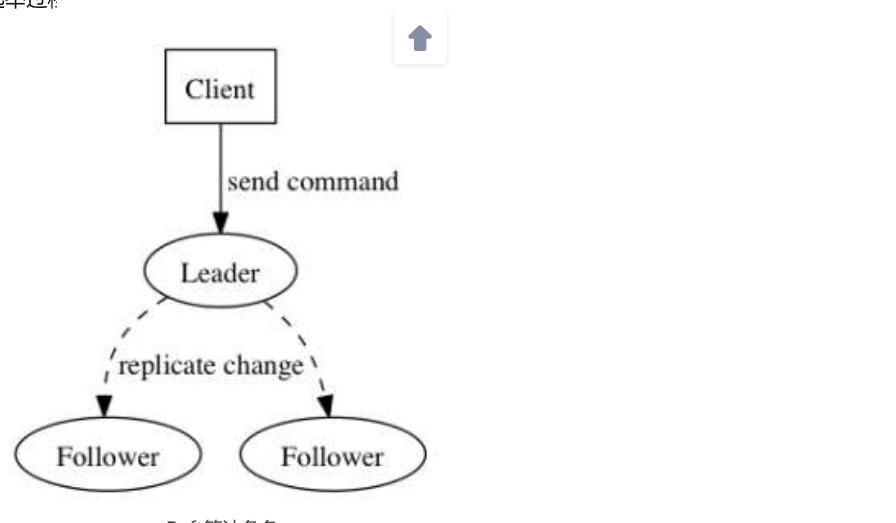
算法总结

一、Raft算法概述

不同于Paxos算法直接从分布式一致性问题出发推导出来，Raft算法则是从多副本状态机的角度提出，用于管理多副本状态机的日志复制。Raft实现了和Paxos相同的功能，它将一致性分解为多个子问题：Leader选举（Leader election）、日志同步（Log replication）、安全性（Safety）、日志压缩（Log compaction）、成员变更（Membership change）等。同时，Raft算法使用了更强的假设来减少了需要考虑的状态，使之变的易于理解和实现。

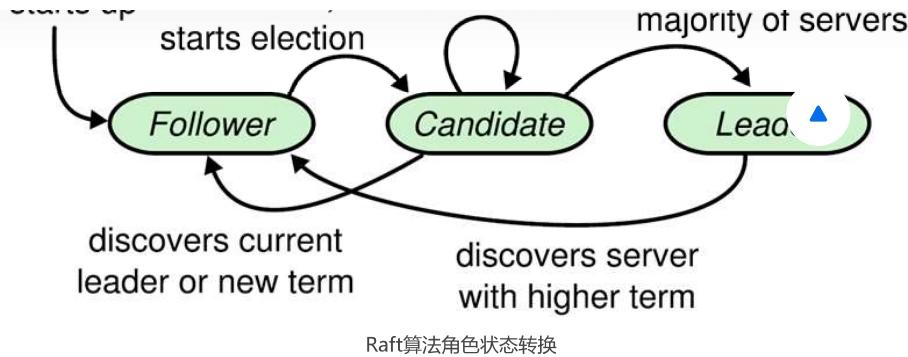
Raft将系统中的角色分为领导者（Leader）、跟从者（Follower）和候选人（Candidate）：

- **Leader**: 接受客户端请求，并向Follower同步请求日志，当日志同步到大多数节点上后告诉Follower提交日志。
- **Follower**: 接受并持久化Leader发来的日志。
- **Candidate**: Leader选举过程中的临时角色。

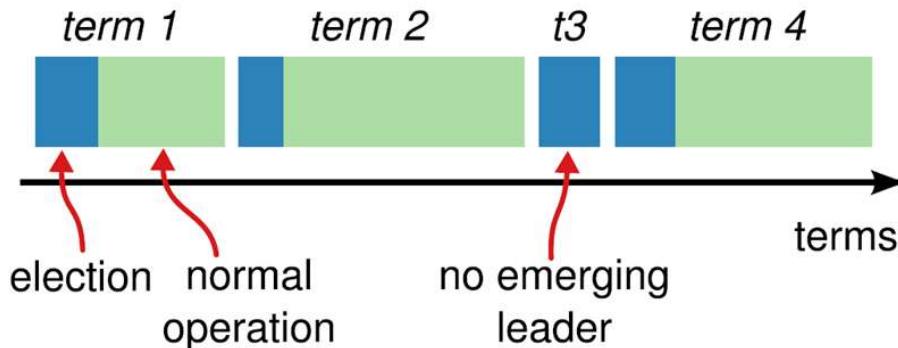


Raft要求系统在任意时刻最多只有一个Leader，正常工作期间只有Leader和Followers。

Raft算法角色状态转换如下：



Follower只响应其他服务器的请求。如果Follower超时没有收到Leader的消息，它会成为一个Candidate并且开始一次Leader选举。收到大多数服务器投票的Candidate会成为新的Leader。Leader在宕机之前会一直保持Leader的状态。



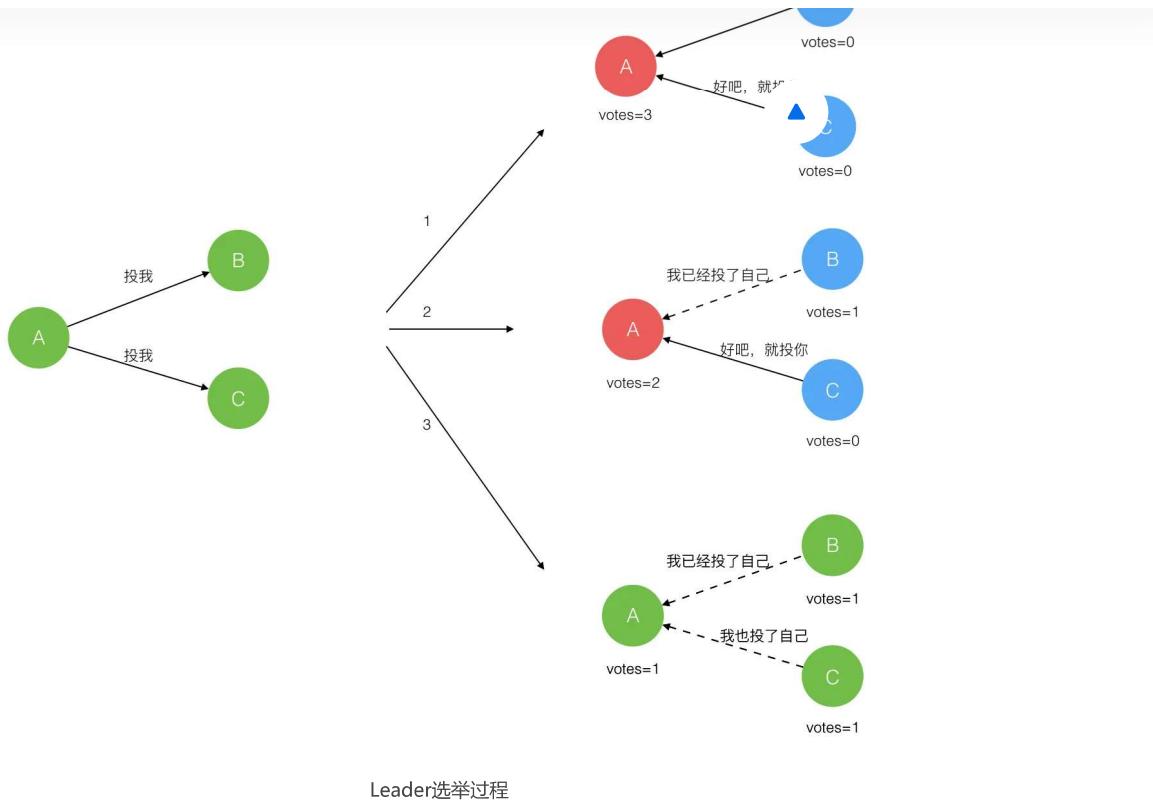
Raft算法将时间分为一个个的任期 (term)，每一个term的开始都是Leader选举。在成功选举Leader之后，Leader会在整个term内管理整个集群。如果Leader选举失败，该term就会因为没有Leader而结束。

二、Leader选举

Raft 使用心跳 (heartbeat) 触发Leader选举。当服务器启动时，初始化为Follower。Leader向所有Followers周期性发送heartbeat。如果Follower在选举超时时间内没有收到Leader的heartbeat，就会等待一段随机的时间后发起一次Leader选举。

Follower将其当前term加一然后发送 RequestVote RPC (RPC细节参见八、Raft具体总结)。结果有以下两种情况：

- 赢得了多数的选票，成功选举为Leader；
- 收到了Leader的消息，表示有其它服务器已经抢先当选了Leader；
- 没有服务器赢得多数的选票，Leader选举失败，等待选举时间超时后发起下一次选举。

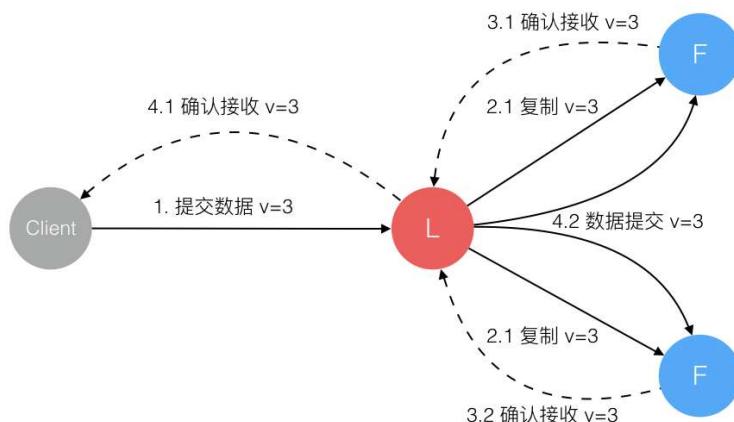


选举出Leader后，Leader通过定期向所有Followers发送心跳信息维持其统治。若Follower一段时
间未收到Leader的心跳则认为Leader可能已经挂了，再次发起Leader选举过程。

Raft保证选举出的Leader上一定具有最新的已提交的日志，这一点将在四、安全性中说明。

三、日志同步

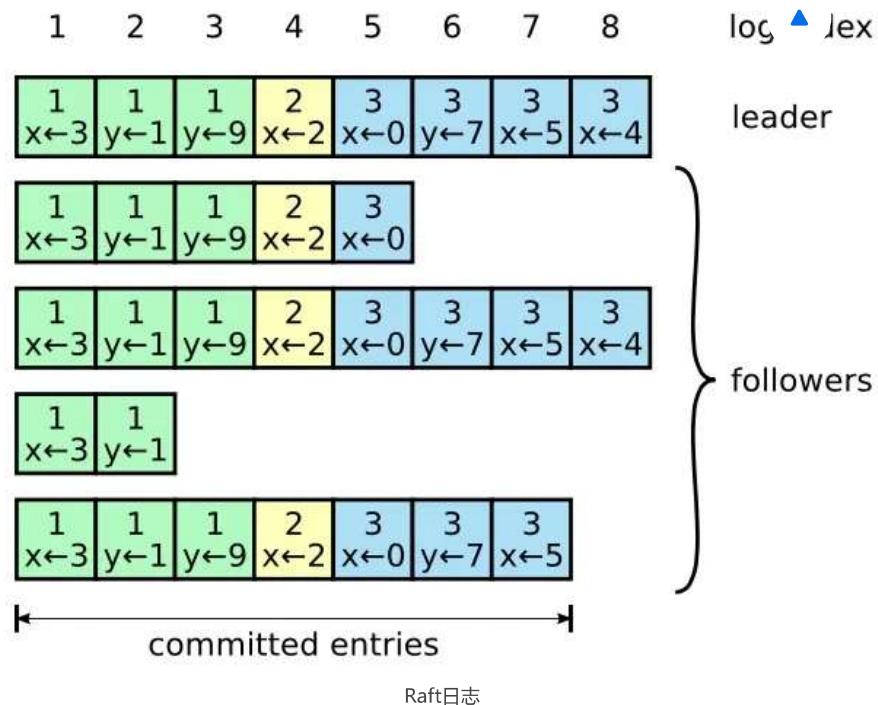
Leader选出后，就开始接收客户端的请求。Leader把请求作为日志条目（Log entries）加入到它的
日志中，然后并行的向其他服务器发起 AppendEntries RPC （RPC细节参见八、Raft算法总
结）复制日志条目。当这条日志被复制到大多数服务器上，Leader将这条日志应用到它的状态机
并向客户端返回执行结果。



Raft日志同步过程

某些Followers可能没有成功的复制日志，Leader会无限的重试 AppendEntries RPC直到所有的
Followers最终存储了所有的日志条目。

以提交 (commit) 了。



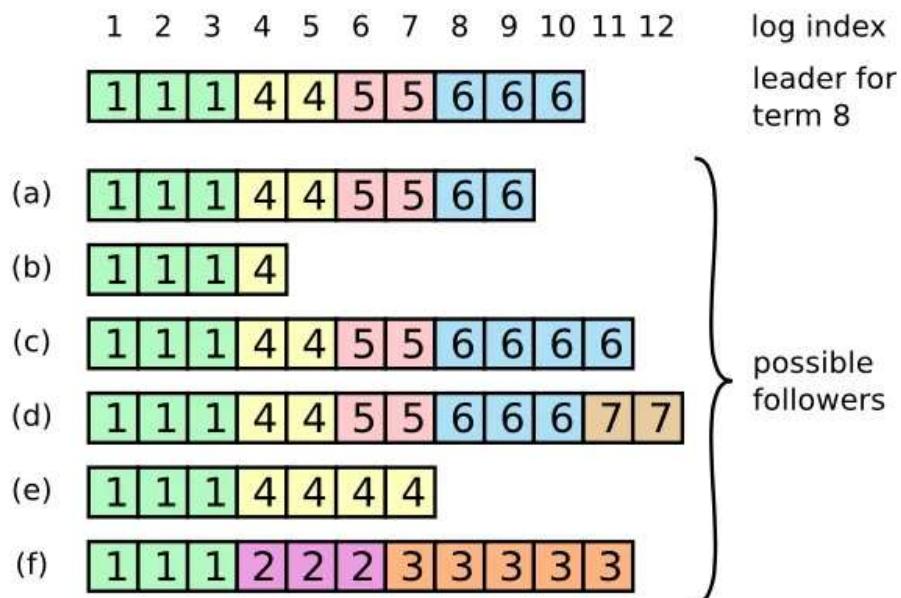
Raft日志同步保证如下两点：

- 如果不同日志中的两个条目有着相同的索引和任期号，则它们所存储的命令是相同的。
- 如果不同日志中的两个条目有着相同的索引和任期号，则它们之前的所有条目都是完全一样的。

第一条特性源于Leader在一个term内在给定的一个log index最多创建一条日志条目，同时该条目在日志中的位置也从来不会改变。

第二条特性源于 AppendEntries 的一个简单的一致性检查。当发送一个 AppendEntries RPC 时，Leader会把新日志条目紧接着之前的条目的log index和term都包含在里面。如果Follower没有在它的日志中找到log index和term都相同的日志，它就会拒绝新的日志条目。

一般情况下，Leader和Followers的日志保持一致，因此 AppendEntries 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。



Leader和Followers上日志不一致

来的条目可能会持续多个任期。

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AppendEntries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。

四、安全性

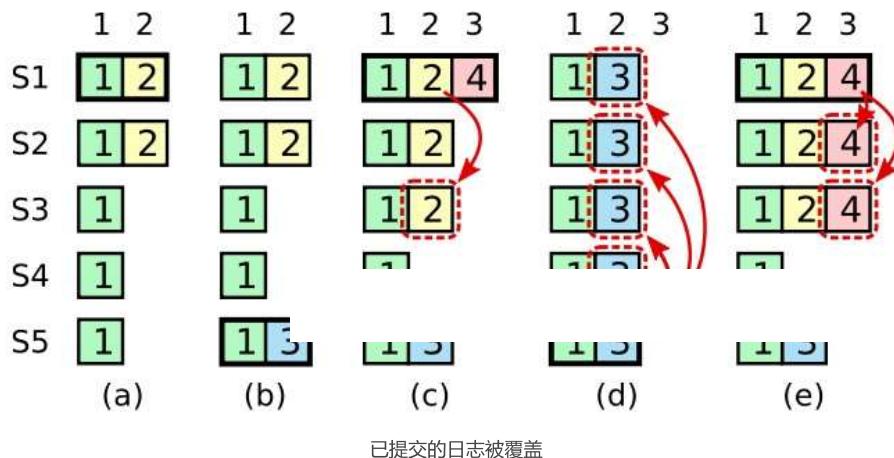
Raft增加了如下两条限制以保证安全性：

- 拥有最新的已提交的log entry的Follower才有资格成为Leader。

这个保证是在RequestVote RPC中做的，Candidate在发送RequestVote RPC时，要带上自己的最后一条日志的term和log index，其他节点收到消息时，如果发现自己的日志比请求中携带的更新，则拒绝投票。日志比较的原则是，如果本地的最后一条log entry的term更大，则term大的更新，如果term一样大，则log index更大的更新。

- Leader只能推进commit index来提交当前term的已经复制到大多数服务器上的日志，旧term日志的提交要等到提交当前term的日志来间接提交（log index 小于 commit index的日志被间接提交）。

之所以要这样，是因为可能会出现已提交的日志又被覆盖的情况：



在阶段a，term为2，S1是Leader，且S1写入日志 (term, index) 为(2, 2)，并且日志被同步写入了S2；

在阶段b，S1离线，触发一次新的选主，此时S5被选为新的Leader，此时系统term为3，且写入了日志 (term, index) 为 (3, 2)；

S5尚未将日志推送到Followers就离线了，进而触发了一次新的选主，而之前离线的S1经过重新上线后被选中变成Leader，此时系统term为4，此时S1会将自己的日志同步到Followers，按照上图就是将日志 (2, 2) 同步到了S3，而此时由于该日志已经被同步到了多数节点 (S1, S2, S3)，因此，此时日志 (2, 2) 可以被提交了。；

在阶段d，S1又下线了，触发一次选主，而S5有可能被选为新的Leader（这是因为S5可以满足作为主的一切条件：1. term = 5 > 4，2. 最新的日志为 (3, 2)，比大多数节点（如S2/S3/S4的日志都新），然后S5会将自己的日志更新到Followers，于是S2、S3中已经被提交的日志 (2, 2) 被截断了。）

数Followers确认，S1方可提交日志（4, 4）这条日志，当然，根据Raft定义，（4, 4）之前的所有日志也会被提交。此时即使S1再下线，重新选主时S5不可能成为Leader，因为它没有包含大多数节点已经拥有的日志（4, 4）。



五、日志压缩

在实际的系统中，不能让日志无限增长，否则系统重启时需要花很长的时间进行回放，从而影响可用性。Raft采用对整个系统进行snapshot来解决，snapshot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进行snapshot，并且只能对已经提交的日志记录进行snapshot。

Snapshot中包含以下内容：

- 日志元数据。最后一条已提交的log entry的log index和term。这两个值在snapshot之后的第一条log entry的AppendEntries RPC的完整性检查的时候会被用上。
- 系统当前状态。

当Leader要发给某个日志落后太多的Follower的log entry被丢弃，Leader会将snapshot发给Follower。或者当新加进一台机器时，也会发送snapshot给它。发送snapshot使用InstalledSnapshot RPC (RPC细节参见八、Raft算法总结)。

做snapshot既不要做的太频繁，否则消耗磁盘带宽，也不要做的太不频繁，否则一旦节点重启需要回放大量日志，影响可用性。推荐当日志达到某个固定的大小做一次snapshot。

做一次snapshot可能耗时过长，会影响正常日志同步。可以通过使用copy-on-write技术避免snapshot过程影响正常日志同步。

六、成员变更

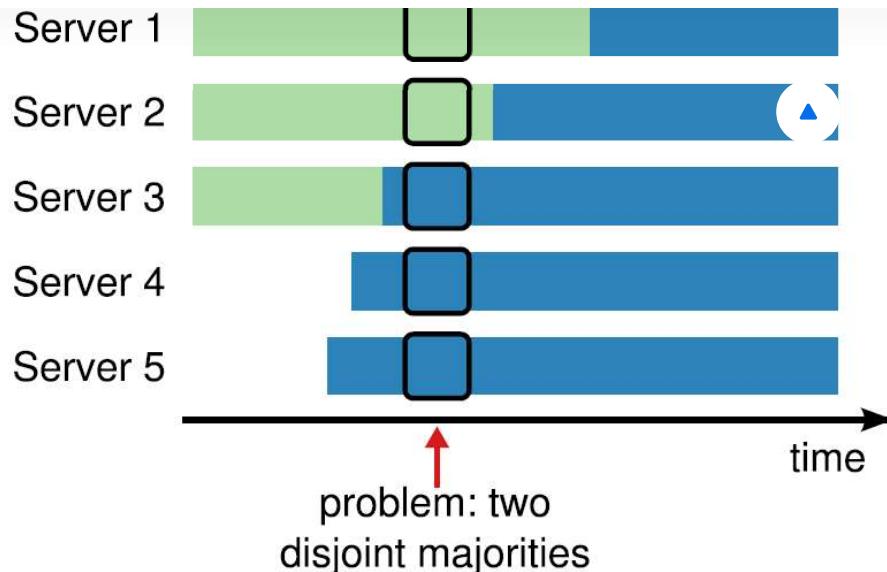
成员变更是指集群运行过程中副本发生变化，如增加/减少副本数、节点替换等。

成员变更也是一个分布式一致性问题，既所有服务器对新成员达成一致。但是成员变更又有其特殊性，因为在成员变更的一致性达成的过程中，参与投票的进程会发生变化。

如果将成员变更当成一般的一致日志，达成多数派之后提交，各配置（Cold）切换到新成员配置（Cnew）。

因为各个服务器提交成员变更日志的时刻可能不同，造成各个服务器从旧成员配置（Cold）切换到新成员配置（Cnew）的时刻不同。

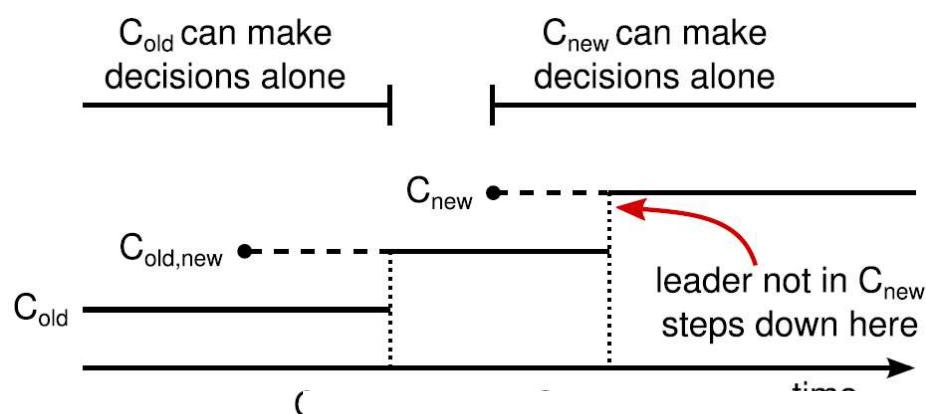
成员变更不能影响服务的可用性，但是成员变更过程的某一时刻，可能出现在Cold和Cnew中同时存在两个不相交的多数派，进而可能选出两个Leader，形成不同的决议，破坏安全性。



成员变更的某一时刻 C_{old} 和 C_{new} 中同时存在两个不相交的多数派

由于成员变更的这一特殊性，成员变更不能当成一般的一致性问题去解决。

为了解决这一问题，Raft提出了两阶段的成员变更方法。集群先从旧成员配置 C_{old} 切换到一个过渡成员配置，称为共同一致（joint consensus），共同一致是旧成员配置 C_{old} 和新成员配置 C_{new} 的组合 $C_{old} \cup C_{new}$ ，一旦共同一致 $C_{old} \cup C_{new}$ 被提交，系统再切换到新成员配置 C_{new} 。



Raft两阶段成员变更

Raft两阶段成员变更过程如下：

1. Leader收到成员变更请求从 C_{old} 切成 $C_{old,new}$ ；
2. Leader在本地生成一个新的log entry，其内容是 $C_{old} \cup C_{new}$ ，代表当前时刻新旧成员配置共存，写入本地日志，同时将该log entry复制至 $C_{old} \cup C_{new}$ 中的所有副本。在此之后新的日志同步需要保证得到 C_{old} 和 C_{new} 两个多数派的确认；
3. Follower收到 $C_{old} \cup C_{new}$ 的log entry后更新本地日志，并且此时就以该配置作为自己的成员配置；
4. 如果 C_{old} 和 C_{new} 中的两个多数派确认了 $C_{old} \cup C_{new}$ 这条日志，Leader就提交这条log entry并切换到 C_{new} ；
5. 接下来Leader生成一条新的log entry，其内容是新成员配置 C_{new} ，同样将该log entry写入本地日志，同时复制到Follower上；
6. Follower收到新成员配置 C_{new} 后，将其写入日志，并且从此刻起，就以该配置作为自己的成员配置，并且如果发现自己不在 C_{new} 这个成员配置中会自动退出；
7. Leader收到 C_{new} 的多数派确认后，表示成员变更成功，后续的日志只要得到 C_{new} 多数派确认即可。Leader给客户端回复成员变更执行成功。

异常分析：

- 如果Leader的Cold U Cnew推送到大部分的Follower后就挂了，此后选出的新Leader可能是Cold也可能是Cnew中的某个Follower。
- 如果Leader在推送Cnew配置的过程中挂了，那么同样，新选出来的Leader可能是Cold也可能是Cnew中的某一个，此后客户端继续执行一次改变配置的命令即可。
- 如果大多数的Follower确认了Cnew这个消息后，那么接下来即使Leader挂了，新选出来的Leader肯定位于Cnew中。

两阶段成员变更比较通用且容易理解，但是实现比较复杂，同时两阶段的变更协议也会在一定程度上影响变更过程中的服务可用性，因此我们期望增强成员变更的限制，以简化操作流程。

两阶段成员变更，之所以分为两个阶段，是因为对Cold与Cnew的关系没有做任何假设，为了避免Cold和Cnew各自形成不相交的多数派选出两个Leader，才引入了两阶段方案。

如果增强成员变更的限制，假设Cold与Cnew任意的多数派交集不为空，这两个成员配置就无法各自形成多数派，那么成员变更方案就可能简化为一阶段。

那么如何限制Cold与Cnew，使之任意的多数派交集不为空呢？方法就是每次成员变更只允许增加或删除一个成员。

可从数学上严格证明，只要每次只允许增加或删除一个成员，Cold与Cnew不可能形成两个不相交的多数派。

一阶段成员变更：

- 成员变更限制每次只能增加或删除一个成员（如果要变更多个成员，连续变更多次）。
- 成员变更由Leader发起，Cnew得到多数派确认后，返回客户端成员变更成功。
- 一次成员变更成功前不允许开始下一次成员变更，因此新任Leader在开始提供服务前要将自己本地保存的最新成员配置重新投票形成多数派确认。
- Leader只要开始同步新成员配置，即可开始使用新的成员配置进行日志同步。

七、Raft与Multi-Paxos的异同

Raft与Multi-Paxos都是基于领导者的一致性算法，乍一看有很多地方相同，下面总结一下Raft与Multi-Paxos的异同。

Raft与Multi-Paxos中相似的概念：

Raft	
Leader	Proposer
Term	Proposal ID
Log	Proposal Value
Log index	Instance ID
RequestVote	Prepare阶段
AppendEntries	Accept阶段

Raft与Multi-Paxos中相似的概念

Raft与Multi-Paxos的不同：

领导者	唯一Leader	允许多Proposer
领导者选举权	具有最新提交的日志的副本	任意副本
日志连续性	保证连续	允许空洞
日志提交	推进commit index	异步的Commit消息

Raft与Multi-Paxos的不同

八、Raft算法总结

Raft算法各节点维护的状态：

State	
Persistent state on all servers:	
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:	
commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on leaders:	
nextIndex[]	to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

Raft各节点维护的状态

Leader选举：

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Leader选举

日志同步:

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)

leaderCommit

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

日志同步

Rules for Servers

All Servers:

- If $\text{commitIndex} > \text{lastApplied}$: increment lastApplied , apply $\log[\text{lastApplied}]$ to state machine (§5.3)
- If RPC request or response contains term $T > \text{currentTerm}$: set $\text{currentTerm} = T$, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
 - Increment currentTerm
 - Vote for self
 - Reset election timer
 - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index $\geq \text{nextIndex}$ for a follower: send AppendEntries RPC with log entries starting at nextIndex
 - If successful: follower (§5.3)
 - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that $N > \text{commitIndex}$, a majority of $\text{matchIndex}[i] \geq N$, and $\log[N].\text{term} == \text{currentTerm}$: set $\text{commitIndex} = N$ (§5.3, §5.4).

Raft状态机

安装snapshot:

Invoked by leader to send chunks of a snapshot to a follower.
Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself
-------------	--

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

安装snapshot

参考

[In Search of an Understandable Consensus Algorithm \(Extended Version\)](#)

[CONSENSUS: BRIDGING THEORY AND PRACTICE](#)

编辑于 2022-02-17 12:13

[分布式系统](#) [分布式一致性](#) [Paxos](#)

写下你的评论...

100 条评论

默认 最新



river

并不是有且仅有一个leader,而是任意一个term最多一个leader

2018-01-26

回复 22



南蛮小子

知乎

首发于
Paxos、Raft分布式一致性最佳实践

2018-01-31

回复 8



祥光 作者

表达不够严谨，谢谢指正
2018-01-26

回复 3

展开其他 2 条回复 >



CHENJING

raft原英文版论文：raft.github.io/raft.pdf

raft中文翻译版：object.redisant.com/doc...
04-24

回复 5



梦清璇 作者
可以维基下拜占庭问题

06-17

回复 喜欢



枫叶飘零

中文是机翻吗，啥东西翻译成了拜占庭…
06-14

回复 喜欢



Tao

安全性中的一段话“因此，此时日志 (2, 2) 可以被提交了。”，此处表述不妥，原文中是说，此日志未提交。“but it is not committed”

2021-02-10

回复 4



byebye

是的 区别还蛮大了
08-21

回复 喜欢



陈江

这是github上的原文啊
2018-01-28

回复 2



祥光 作者

github链接发出来看看
2018-01-28

回复 14



kevin

leader接收客户端所有请求？岂不压力很大？如果leader被打挂，即便选出新的leader还是被打挂？
2018-02-07

三年不开张 作者 kevin

说白了，就是提供个replica log，上层怎么用，和它没啥关系，你这请求过多情况，跟 raft 这种基本是八竿子打不着

2018-09-15

回复 7



祥光 作者 bingo

工程实践中，可以实现为，客户端可以访问任意节点，非leader节点可以将请求重定向至leader节点
2018-11-28

回复 6

查看全部 13 条回复 >



Tao

raft已经提交的log entries，还是可以被重写的吗
2021-02-10

回复 1



孟凡宇

不可以
06-09

回复 喜欢



无痕

怎么知道一共有多少节点呢？
2022-04-05

回复 1

知乎

首发于
Paxos、Raft分布式一致性最佳实践

2022-07-14

● 回复 1



心姐

选举标准: term id, log term, log index。选取选期最前的, 最后一条记录, 记录我旧。持有所有已经提交成功的日志, 才能成功选举leader, 注意若无之前已经提交的日志, 是不可能成为leader的。删掉上没有正常提交给用户的记录是可以的

2021-05-11

● 回复 1



张睿

写得很好, 挑一个小毛病: "如果本地的最后一条log entry的term更大, 则term大的更新, 如果term一样大, 则log index更大的更新", 其实如果一个node term大但是log entry index不是最新也不能更新, 比如一个isolation node可能一直TO之后reelect, 然后不停增加term, 等到重新连接上cluster之后它的term可能比其他高很多, 但是log index小于现在的, 也不能选为leader并且更新

2019-07-08

● 回复 1

张睿 → 祥光

又看了一下, 我上面说的也不是很对, 关键不在于比较log的index, 而在于比较最高index的term, 你之前写的是对的, 修改一下我的例子, 比如比如现在网络上term是10, 但是isolation node的term=20, isolation node back, 那么isolation node首先requestforvote, 带上term20, 但是因为它之前isolate, 所以比如它虽然还一直bump term (因为timeout), 但是log上带的是term 9的最后一个log, 这里9 < 10, 所以它会被拒绝

2019-07-08

● 回复 喜欢

祥光 [作者]

谢谢细化

2019-07-08

● 回复 喜欢



click

您好楼主, 请教个问题:

网络分区, 一个follower被隔离。这个follower的后续行为会怎么样? 会切换为candidate不停地发起选举吧? 当这个candidate的term大于主分区后整个网络连通了, 这个时候会怎么样?

敬盼回复。

2019-04-18

● 回复 1

张睿 → 祥光

其实是这样, 比如现在网络上term是10, logindex是5000, 但是isolation node的term=20, logindex=3000, 那么isolation node首先requestforvote, 带上term20, 但是其他node发现这个index 3000 > 5000, 不会同意, 但是同时他们都更新term到20, 下一次选举就是term21开始

2019-07-08

zeimi33 [皇冠]

现在已经有prevote解决这个问题了

2019-12-31

● 回复 1

展开其他 3 条回复 >



Coastline

raft算法不是强一致性吗, leader会保证follower落后的情况吧

04-01

● 回复 喜欢



FearofEternity

这个是我能聊的吗

2022-11-08

● 回复 喜欢



格兰德

有点复杂

2022-10-18

● 回复 喜欢



莞风

leader强行复制给follower的过程, follower会undo旧日志+redo leader新日志吧

2022-07-02

● 回复 喜欢



恋喵大鲤鱼

用户口令已成功设置。

2022-06-22

回复 喜欢

**恋猫大鲤鱼**

"这是因为S5可以满足作为主的一切条件: 1. term = 5 > 4", 这个力应该是 3 > 2

2022-06-21

回复 喜欢

**马丁**

应该是3>2吧，作者能帮着解答么

2022-09-29

回复 喜欢

**蜡笔小新爱好者**

成员变更，虚线是创建未提交状态吗？可是提交前上面显示Cold make decision alone呀。

也就是提交前不必Cold，Cnew一起必须majority，是这样吗？但是和你说的不符合呀？

2022-04-17

回复 喜欢

**wfw**

写的挺好的。尤其安全性那块。收获很大。

2021-08-05

回复 喜欢

**心姐**

leader与follower不一致情况下f的情况出现原因：f赢得term选举，第一步append数据后，第二步commit失败，与其他断开连接。然后f又连接上了，参与term3选举，且参选成功，又append成功，但是commit失败。

2021-05-11

回复 喜欢

**心姐**

解决配置更新可能带来的问题：当有新配置时，保证candidate在新旧选区都赢得多数派，可以保证一个term只有一个leader。

2021-05-11

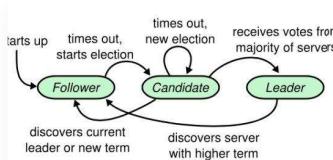
回复 喜欢

[点击查看全部评论 >](#)

写下你的评论...

**Paxos、Raft分布式一致性最佳实践**

分布式一致性最佳实践分享，跨越理论与实践的鸿沟

**raft算法浅析**

bloom...

发表于网络与存储...

**Raft算法很难吗？这篇是最易懂的.....**

dbapl...

发表于数据库全能...

Raft算法

liuyi...

发表于区块链碎碎...

Abstract

he Paxos algorithm, when presented in plain English, is very simple.

RaftRaft
(cor
Raft
篇文
文、
面、

一页