Raft -论文导读 与 ETCD源码解读

1. 算法的介绍
2. 算法的详细实现
3. 算法原理与证明
4. 工程优化
5. ETCD 源码实现
6. 一个小作业

简介

如何多快好省的对大规模数据集进行存储和计算?

a. 更好的机器
b. 更多的机器

如何让跨网络的机器之间协调一致的工作?

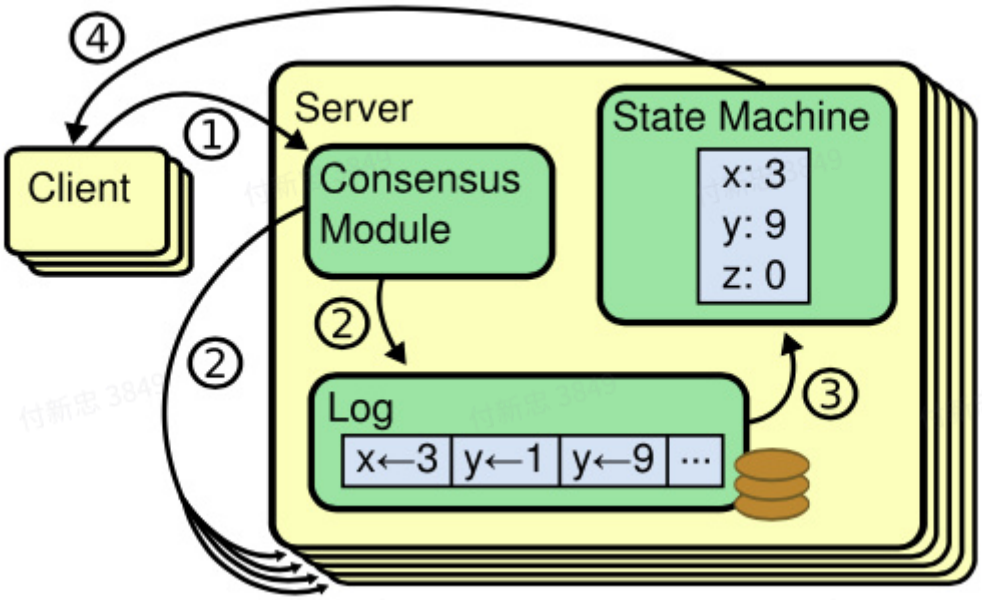
a. 状态的立即一致
b. 状态的最终一致

如何应对网络的不可靠以及节点的失效?

a. 可读写
b. 可读
c. 不可用

- 组织机器使其状态最终一致并允许局部失败的算法称之为**一致性算法**.
- Paxos算法由来已久,目前是功能和性能最完善的一致性算法，然而他难以理解与实现。 raft简化了paxos,它是以易于理解为首要目标,尽量提供与paxos一样的功能与性能。

复制状态机



一致性算法的目标就是保证集群上所有节点的状态一致,节点要执行的指令可以分为两种,读与写。只有写指令会改变节点状态,因此为了保证集群各个节点状态的一致,那就必须将写指令同步给所有节点。

理想状态下,我们期望**任意节点**发生写命令都会**立即**的在其他节点上变更状态,这其中没有任何时延,所有节点都好像是单机一样被变更状态。

网络延迟要远远慢于内存操作, 写入命令不可能被同时执行,因此如果在不同节点发生不同的写命令,那么在其他节点上这些写命令被应用的顺序很可能完全不同。

如果我们不要求所有节点的写命令立即被执行,而仅仅是保证所有的写命令在所有的节点上按同样的顺序最终被执行呢? 第一, 仅仅允许一个节点处理写命令,第二,所有的节点维护一份顺序一致的日志。

每个节点上的状态机按照自己的节奏,逐条应用日志上的写命令来变更状态。

定义问题

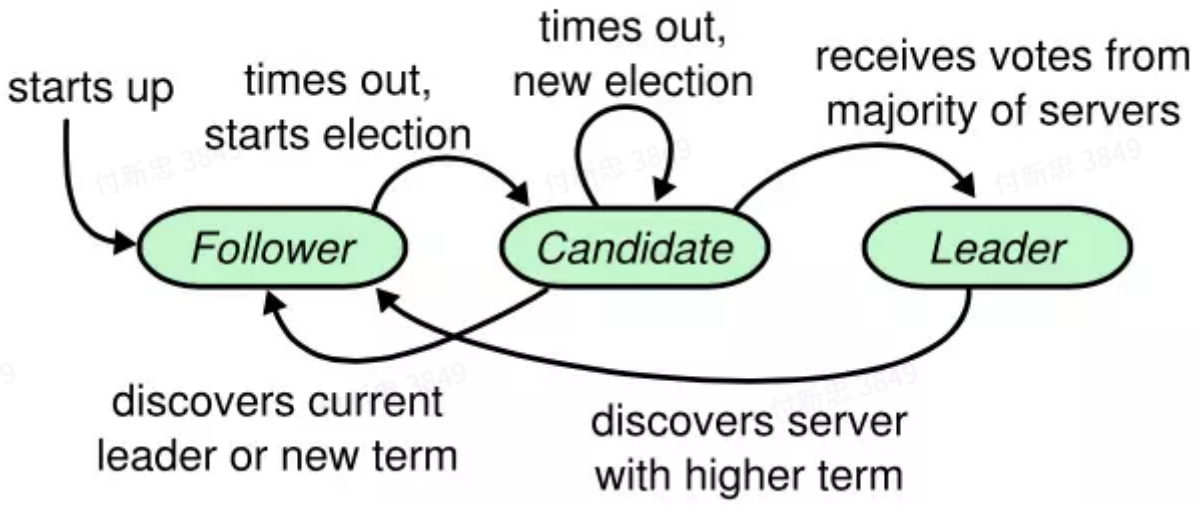
1. 输入: 写入命令
2. 输出: 所有节点最终处于相同的状态
3. 约束
    a. 网络不确定性: 在非拜占庭情况下,出现网络 分区/冗余/丢失/乱序 等问题下要保证正确。
    b. 基本可用性: 集群中大部分节点能够保持相互通信,那么集群就应该能够正确响应客户端
    c. 不依赖时序: 不依赖物理时钟或极端的消息延迟来保证一致性
    d. 快速响应: 对客户端请求的响应不能依赖集群中最慢的节点

一个可行解

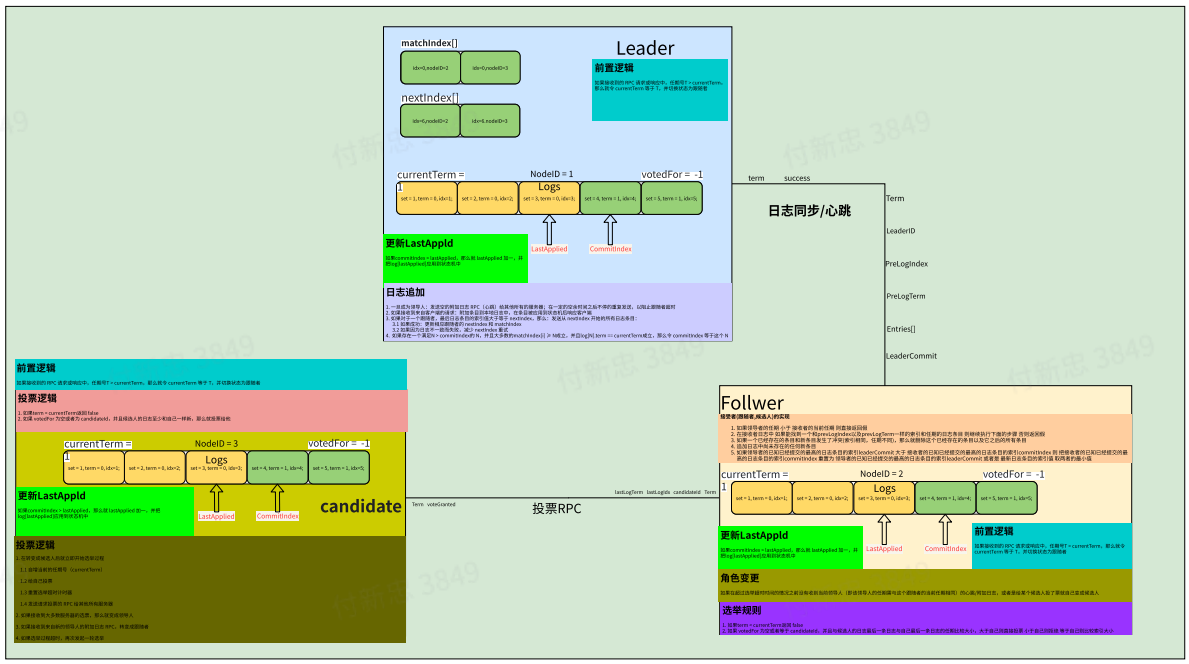
1. 初始化的时候有一个领导者节点，负责发送日志到其他跟随者,并决定日志的顺序
2. 当读请求到来时,在任意节点都可以读,而写请求只能重定向到领导者进行
3. 领导者先写入自己的日志,然后同步给半数以上节点,跟随者表示都ok了,领导者才提交日志
4. 日志最终由领导者先按顺序应用于状态机,其他跟随者随机应用到状态机
5. 当领导者崩溃后,其他跟随者通过心跳感知并选举出新的领导者继续集群的正常运转
6. 当有新的节点加入或退出集群,需要将配置信息同步给整个集群

raft的详细实现

状态机



数据结构



通用持久性状态

	A	B
1	**参数**	**解释**
2	currentTerm	服务器已知最新的任期（在服务器首次启动的时候初始化为0，单调递增）
3	votedFor	当前任期内收到选票的候选者id 如果没有投给任何候选者 则为空
4	log[]	日志条目;每个条目包含了用于状态机的命令，以及领导者接收到该条目时的任期（

通用易失性状态

	A	B
1	**参数**	**解释**
2	commitIndex	已知已提交的最高的日志条目的索引（初始值为0，单调递增）
3	lastApplied	已经被应用到状态机的最高的日志条目的索引（初始值为0，单调递增）

领导者上的易失性状态

	A	B
1	参数	解释
2	nextIndex[]	对于每一台服务器，发送到该服务器的下一个日志条目的索引（初始值为领
3	matchIndex[]	对于每一台服务器，已知的已经复制到该服务器的最高日志条目的索引（初

RPC

> 候选人发起**选举投票**RPC到跟随者或候选人
>
> 由领导者发起RPC到跟随者
>
> a. **日志追加**
> b. **心跳通知**

请求投票

> 1. 跟随者变更为候选人后
> 2. 选举超时后

请求参数

	A	B
1	**参数**	**解释**
2	term	候选人的任期号
3	candidateId	请求选票的候选人的 Id
4	lastLogIndex	候选人的最后日志条目的索引值
5	lastLogTerm	候选人最后日志条目的任期号

返回值

	A	B
1	**返回值**	**解释**
2	term	当前任期号，以便于候选人去更新自己的任期号
3	voteGranted	候选人赢得了此张选票时为真

追加日志&心跳(领导者调用)

1. 客户端发起写命令请求时
2. 发送心跳时
3. 日志匹配失败时

请求参数

	A	B
1	**参数**	**解释**
2	term	当前领导者的任期
3	leaderId	领导者ID 因此跟随者可以对客户端进行重定向
4	prevLogIndex	紧邻新日志条目之前的那个日志条目的索引
5	prevLogTerm	紧邻新日志条目之前的那个日志条目的任期
6	entries[]	需要被保存的日志条目（被当做心跳使用是 则日志条目内容为
7	leaderCommit	领导者的已知已提交的最高的日志条目的索引

返回值

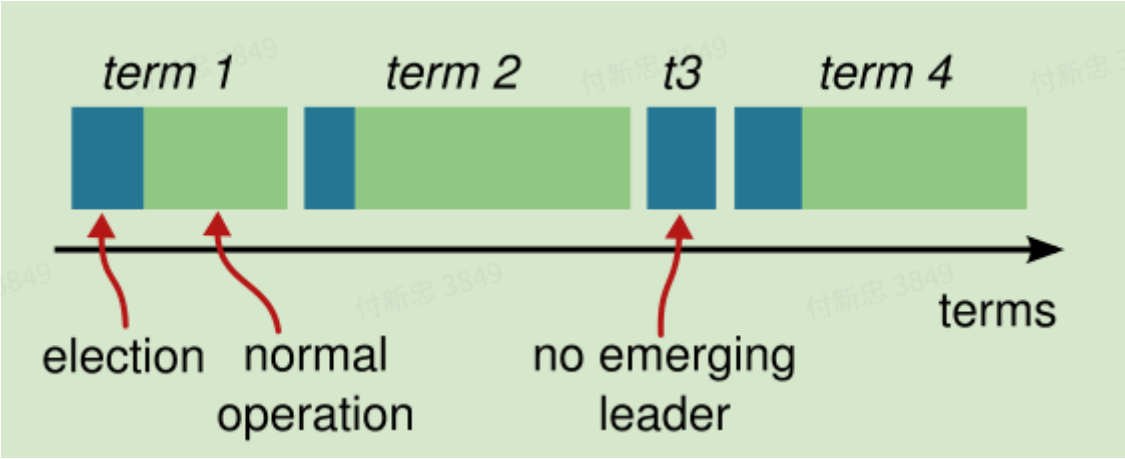
	A	B
1	**返回值**	**解释**
2	term	当前任期,对于领导者而言 它会更新自己的任期
3	success	结果为真 如果跟随者所含有的条目和prevLogIndex以及prevLogTerm匹配上了

算法的原理与证明

五条公理

	A	B
1	**特性**	**解释**
2	选举安全特性	对于一个给定的任期号，最多只会有一个领导人被选举出来
3	领导人只附加原则	领导人绝对不会删除或者覆盖自己的日志，只会增加
4	日志匹配原则	如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这
5	领导人完全特性	如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大
6	状态机安全特性	如果一个领导人已经将给定的索引值位置的日志条目应用到状态机中，那么其

选举安全特性



> 在一个任期内半数以上的票数才能当选，保证每个任期要么0个领导要么1个领导。

日志复制过程的完全匹配

1. **因为** 集群在任意时刻最多有一个leader存在, leader在一个任期内只会在同一个索引处写入一次日志
2. **又因为** 领导者从来不会删除或者覆盖自己的日志, 并且日志一旦写入就不允许修改
3. **所以** 只要任期和索引相同,那么在任何节点上的日志也都相同
4. **因为** 跟随者每次只会从与leader的PreLog匹配处追加日志,如果不匹配 则nextIndex - 1 重试
5. **所以** 由递归的性质可知 一旦跟随者和leader在PreLog处匹配,那么之前的所有日志就都是匹配的
6. **所以** 只要把preLog之后的日志全部按此次Leader同步RPC的日志顺序覆盖即可保证 二者的一致性

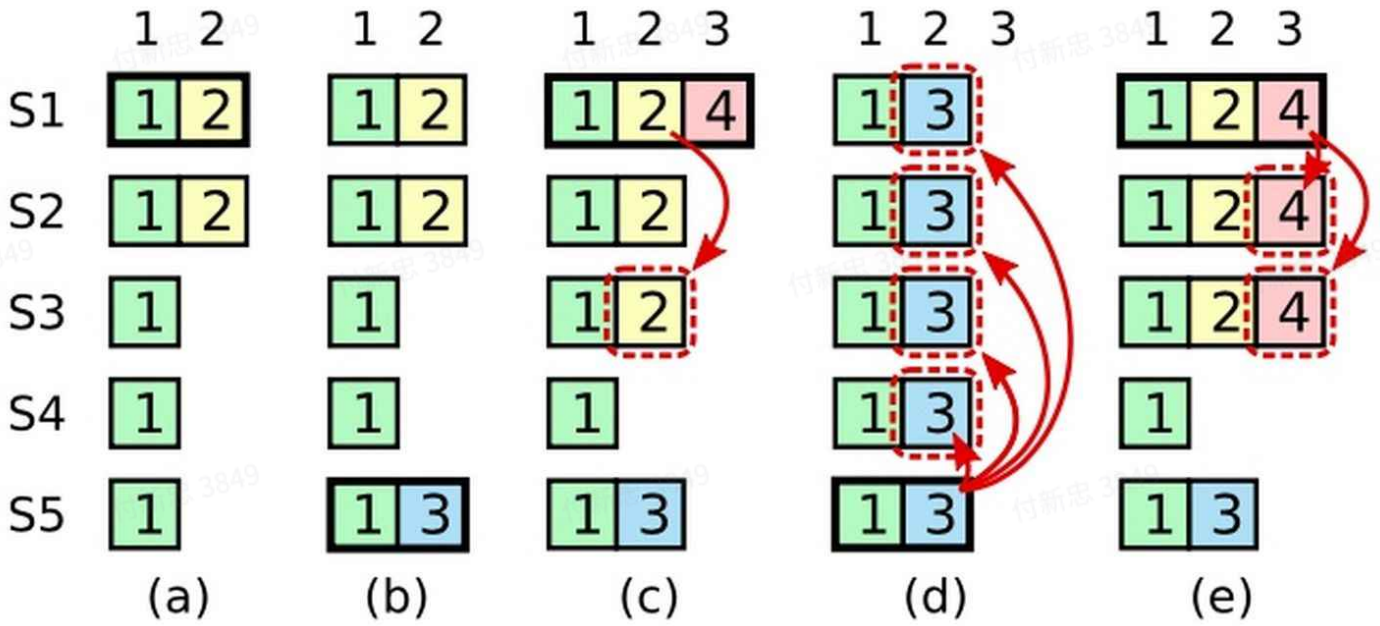
安全性

每一任的领导者 一定会有所有任期内领导者的全部已提交日志吗?

选举限制

选民只会投票给任期比自己大，最后一条日志比自己新(任期大于 或者 等于时索引更大)的候选人。

但这真的正确吗?



1. 时刻a，Sl是任期2的领导人并且向部分节点（S1和S2）复制了2号位置的日志条目，然后宕机
2. 时刻b，S5获得了S3、S4(S5的日志与S3和S4的一样新，最新的日志的任期号都是1）和自己的选票赢得了选举，成了3号任期的领导人，并且在2号位置上写人了一条任期号为3的日志条目。在新日志条目复制到其他节点之前，S5若机了
3. 时刻c，S1重启，并且通过S2、S3、S4和自己的选票赢得了选举，成了4号任期的领导人，并且继续向S3复制2号位置的日志。此时，任期2的日志条目已经在大多数节点上完成了复制
4. 时刻d，S1发生故障，S5通过S2、S3、”的选票再次成为领导人（因为S5最后一条日志条目的任期号是3，比S2、S3、S4中任意一个节点上的日志都更加新），任期号为5。然后S5用自己的本地日志夜写了其他节点上的日志
5. 上面这个例子生动地说明了，即使日志条目被半数以上的节点写盘（复制）了，也并不代表它已经被提交（commited）到Raft集群了——因为一旦某条日志被提交，那么它将永远没法被删除或修改。这个例子同时也说明了，领导人无法单纯地依靠之前任期的日志条目信息判断它的提交状态
6. 因此，针对以上场景，Raft算法对日志提交条件增加了一个额外的限制：要求Leader在当前任期至少有一条日志被提交，即被超过半数的节点写盘
7. 正如上图中e描述的那样，Sl作为Leader，在崩溃之前，将3号位置的日志（任期号为4）在大多数节点上复制了一条日志条目（指的是条目3，term 4），那么即使这时·Sl若机了，SS也不可能赢得选举一一因为S2和白的最新日志条目的任期号为4，比SS的3要大，S3无法获得超过半数的选票。“无法赢得选举，这就意味着2号位置的日志条目不会被覆写

所以新上任的领导者在接受客户端写入命令之前 需要提交一个no-op(空命令)，携带自己任期号的日志复制到大多数集群节点上才能真正的保证选举限制的成立。

状态机安全性证明(三段论)

1. **定义** A为上个任期最后一条已提交日志，B为当前任期的leader
2. **因为** A必然同步到了集群中的半数以上节点
3. **又因为** B只有获得集群中半数以上节点的选票后才能成为leader
4. **所以** B的选民中必然存在拥有A日志的节点
5. **又因为 选举限制**， B成为leader的前提是比给它投票的所有选民都要新
6. **所以** B的日志中必然要包含A
7. **又因为 日志完全匹配规则** 如果A被B包含，那么比A小的所有日志都被B包含
8. **因为** lastApplied <= commitIndex
9. **又因为** raft保证已提交日志在所有集群节点上的顺序一致
10. **所以** 应用日志必然在在所有节点上顺序一致
11. **因为** 状态机只能按序执行应用日志部分
12. **得证** 状态机在整个集群所有节点上必然 **最终一致**。

状态机安全性证明(反证法)

1. 当日志条目L被同步给半数以上节点时,leaderA会移动commitIndex指针提交日志,此时的日志被提交
2. 当leader崩溃后, 由一个新节点成为leaderB，假设leaderB是第一个未包含leaderA最后已提交日志的领导者
3. 选举过程中,只有获得半数以上节点认可才能成为leader,因此至少有一个投票给当前leaderB的节点中含有已经提交的那条日志L。
4. 那么根据选举限制, 节点只会将选票投给至少与自己一样新的节点
    a. 节点C作为包含leaderA最后提交日志条目的投票者, 如果leaderB与节点C的最后一条日志的任期号一样大时，节点C的条目数一定大于leaderB，因为leaderB是第一个未包含最后一条LeaderA日志的领导者。这与选举限制相矛盾，节点C不会投票给leaderB、
    b. 如果leaderB最后一条日志的任期号大于节点C最后一条日志的任期号, 那么leaderB的前任领导中必然包含了leaderA已经提交的日志(leaderB是第一个不包含leaderA已提交日志的领导者 这一假设) 根据 日志匹配特性 leaderB也必须包含leaderA最后的已提交日志,这与假设矛盾。
5. 所以证明 未来所有的领导者必然包含过去领导者已提交的日志,并且日志匹配原则,所有已提交日志的顺序一定是一致的。

> 6. 又因为 任意节点仅会将已提交日志按顺序应用于自身的状态机,更新lastApplied指针,因此所有节点的状态机都会最终顺序一致。
> 7. 得证 raft 算法能够保证节点之间的协同工作。

工程优化

容错性

1. 领导者崩溃通过选举可以解决,但跟随者与候选人崩溃呢?

> 基础的raft算法,通过无限次幂等的附加复制rpc进行重试来解决。

2. 当平均故障时间大于信息交换时间,系统将没有一个稳定的领导者,集群无法工作

> 广播时间 << 心跳超时时间 << 平均故障时间

3. 客户端如何连接raft的server节点?

> 1. 客户端随机选择一个节点去访问,如果是跟随者,跟随者会把自己知道的领导者告知客户端

4. 领导者提交后返回时崩溃,客户端重试不就导致相同的命令反复执行了吗?

> 客户端为每次请求标记唯一序列号,服务端在状态中维护客户端最新的序列号标记 进行幂等处理

5. 客户端给领导者set a=3 并进行了提交,此时客户端如果从一个未被同步的节点读取a 读不到写后的值

> 每个客户端应该维持一个latestIdx值,每个节点在接受读请求的时候与自己的lastApplied值比较,如果这个值大于自己的lastApplied,则拒绝此次请求,客户端重定向到一个lastApplied大于等于自己latestIdx的请求,并且每次读取请求都会返回这个节点的lastApplied值,客户端将latestIdx更新为此值,保证读取的线性一致。

6. 如果leader被孤立, 其他跟随者选举出leader,但是当前leader还是向外提供脏数据怎么办?

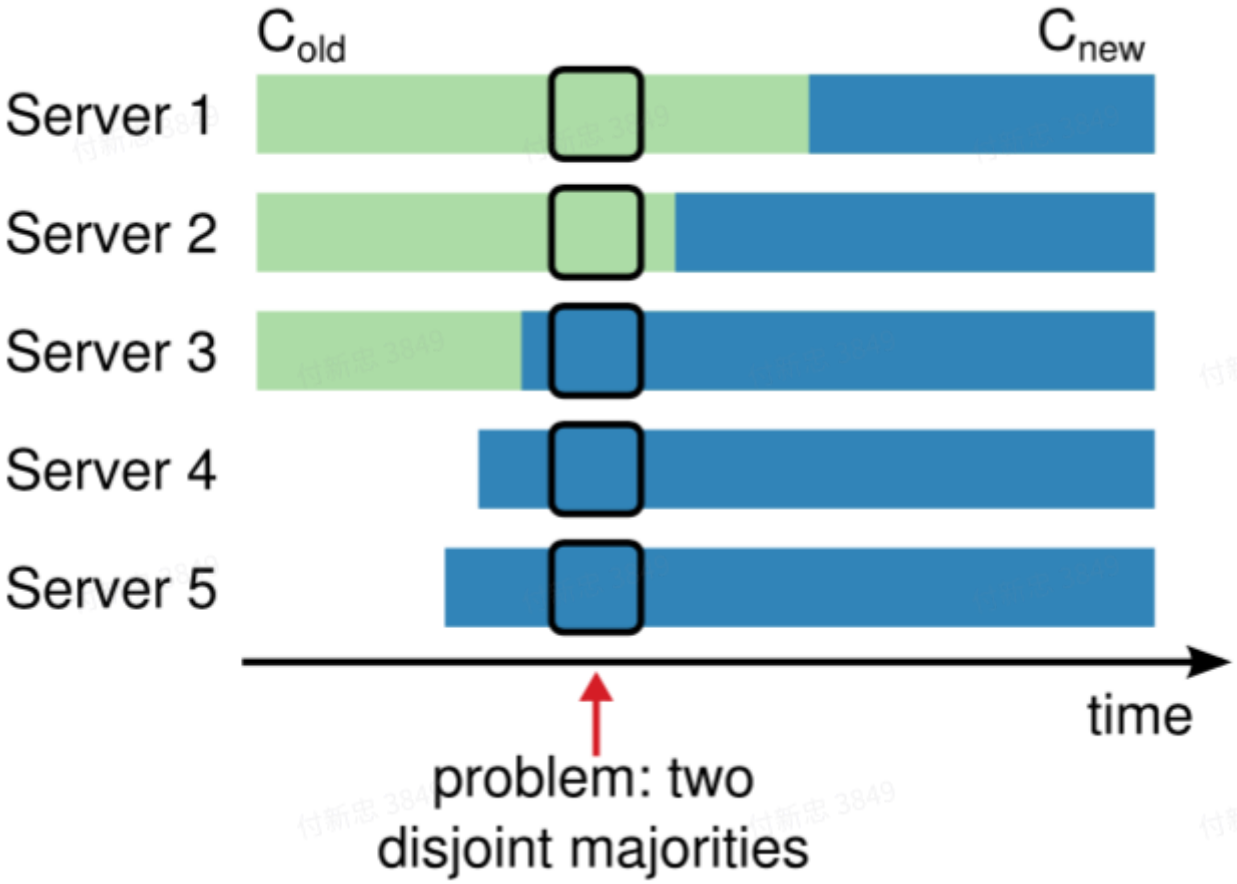
> 写入数据由于无法提交,因此会立即失败，但无法防止读到脏数据
>
> 解决办法是:心跳超过半数失败,leader感知到自己处于少数分区而被孤立进而拒绝提供读写服务

7. 当出现网络分区后， 被孤立少数集合的节点无法选举，只会不断的增加自己的任期 分区恢复后由于失联的节点任期更大，会强行更新所有节点的任期,触发一次重新选举,而又因为其日志不够新，被孤立的节点不可能成为新的leader所以，其状态机是安全的，只是触发了一次重新选举，使得集群有一定时间的不可用。这是完全可以避免的

> 在跟随者成为候选人时，先发送一轮pre-vote rpc 来判断自己是否在大多数分区内(是否有半数节点回应自己），如果是则任期加1进行选举。否则的话就不断尝试pre-vote请求。

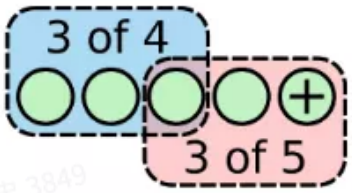
扩展性

1. 集群的成员发生变化时,存在某一时刻新老配置共存,进而有选举出两个领导者的可能

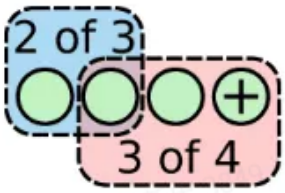


1. **新集群节点在配置变更期间必须获得老配置的多数派投票才能成为leader**
2. 发送新配置c-new给集群的领导者
3. 领导者将自己的c-old配置与c-new合并为一个 c-old-new配置 【123-45】
4. 然后下发给其他所有跟随者
   a. 当c-old-new被同步给半数以上节点后 那么此配置已经提交 遵循raft安全性机制
   b. 当leader在将c-old-new写入半数以上跟随者之前崩溃了，那么选举出来的新leader 会退回到老的配置,此时重试更新配置即可
5. 当c-old-new被提交之后,leader会真正的提交c-new配置
   a. 如果提交给了半数节点 则c-new真正的被提交
   b. 如果未提交给半数节点时崩溃 则新选举的leader 必定包含已提交的c-old-new 那么接着更新配置即可

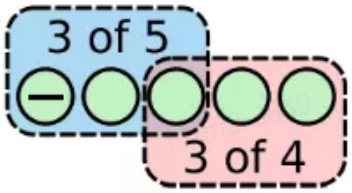
集群变更过于复杂,因此可以简化这一过程,使用单节点变更机制,即每一次只添加或删除一个节点



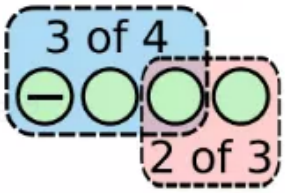
(a) Adding one server to a 4-server cluster.



(b) Adding one server to a 3-server cluster.



(c) Removing one server from a 5-server cluster.



(d) Removing one server from a 4-server cluster.

2. 单节点变更时 如果leader挂了 造成一致性问题(丢失已提交日志)如何处理?

> 新leader先发一条no-op 日志再开始配置变更
>
> 1. Raft成员变更的工程实践

2. 单节点变更时偶数节点遇到网络分区,则没有办法选举leader了怎么办?

> 重新定义偶数节点情况下的 法定人数模型下的大多数情况(n/2 或者 n/2-1)
>
> 1. TiDB 在 Raft 成员变更上踩的坑
> 2. 后分布式时代: 多数派读写的’少数派’实现

3. 新的服务器没有存储任何日志, 领导要复制很长一段时间此时不能参加选举否则会使得整体不可用

> 新加入的节点设置一个保护期, 在此保护期内不会参加选举与日志提交决策 只用来同步日志

4. 如果集群的领导不是新集群中的一员，该如何处理?

> 在提交c-new时 不将自己算作半数提交,并且在提交后要主动退位

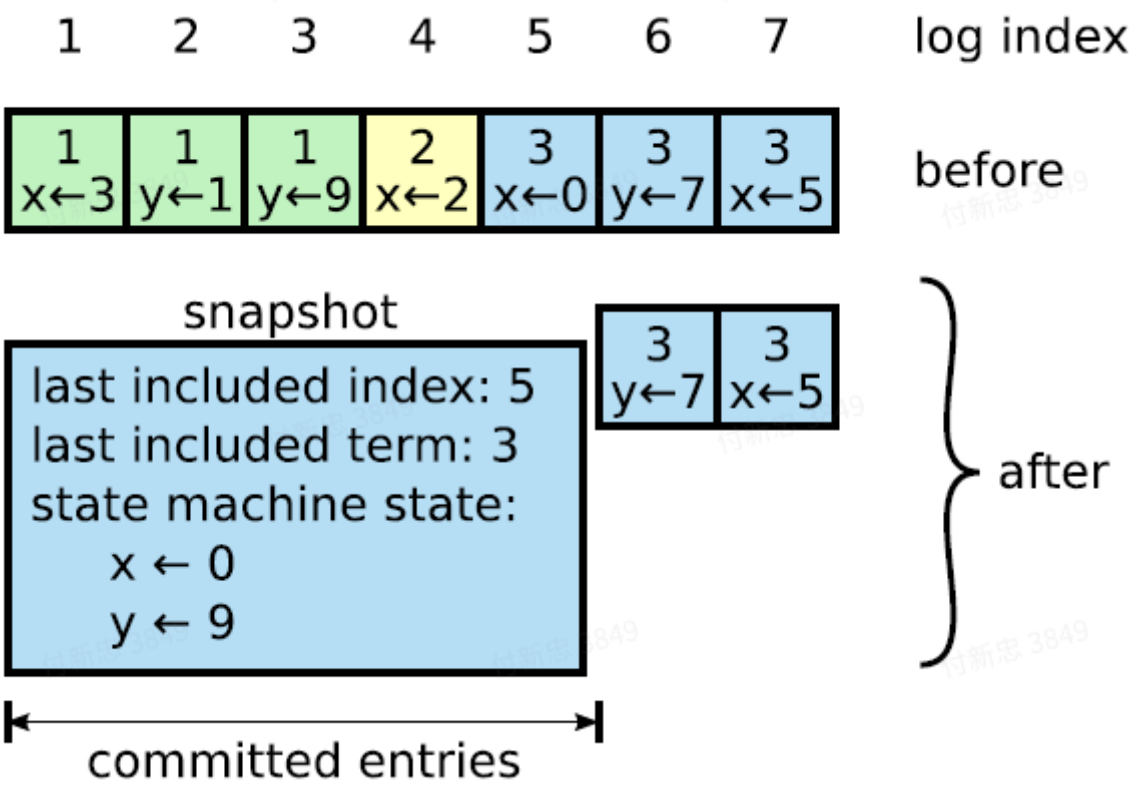
5. 被移除的节点如果不及时关闭,会导致选举超时后强行发起投票请求干扰在线集群

> 每个节点如果未达到最小心跳超时时间,则不会进行投票。

性能

1. 生成快照

- 日志如果无限增长会将本地磁盘打满,这会造成可用性问题



> 定时的将状态机中的状态生成快照,而将之前的日志全部删除,是一种常见的压缩方式
>
> 1. 将节点的状态保存为LSM Tree,然后存储最后应用日志的索引与任期,以保证日志匹配特性
> 2. 为支持集群的配置更新,快照中也要将最后应用的集群配置也当做状态保存下来
> 3. 当跟随者需要的日志已经在领导者上面被删除时(nextIndex--),需要将快照通过RPC发送过
>
> 注意: 由领导人调用以将快照的分块发送给跟随者。领导者总是按顺序发送分块。

	A	B
1	参数	解释
2	term	领导人的任期号
3	leaderId	领导人的 Id，以便于跟随者重定向请求
4	lastIncludedIndex	快照中包含的最后日志条目的索引值
5	lastIncludedTerm	快照中包含的最后日志条目的任期号
6	offset	分块在快照中的字节偏移量
7	data[]	从偏移量开始的快照分块的原始字节
8	done	如果这是最后一个分块则为 true

	A	B
1	**结果**	**解释**
2	term	当前任期号（currentTerm），便于领导人更新自己

C++

```
如果term < currentTerm就立即拒绝
如果是第一个分块（offset 为 0）就创建一个新的快照
在指定偏移量写入数据
如果 done 是 false，则继续等待更多的数据 ack
保存快照文件，丢弃具有较小索引的任何现有或部分快照
如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复
否则 丢弃整个日志
使用快照重置状态机（并加载快照的集群配置）
```

- 快照何时创建? 过于频繁会浪费性能,过于低频日志占用磁盘的量更大,重建时间更长。

> 限定日志文件大小到达某一个阈值后立刻生成快照

- 写入快照花费的时间昂贵如何处理?如何保证不影响节点的正常工作?

> 使用写时复制技术, 状态机的函数式顺序性天然支持

---

2. 调节参数

> 1. 心跳的随机时间， 过快会增加网络负载，过慢则会导致感知领导者崩溃的时间更长
> 2. 选举的随机时间， 如果大部分跟随者同时变为候选人则会导致选票被瓜分

3. 流批结合

> 首先可以做的就是 batch，大家知道，在很多情况下面，使用 batch 能明显提升性能，譬如对于 RocksDB 的写入来说，我们通常不会每次写入一个值，而是会用一个 WriteBatch 缓存一批修改，然后在整个写入。 对于 Raft 来说，Leader 可以一次收集多个 requests，然后一批发送给 Follower。当然，我们也需要有一个最大发送 size 来限制每次最多可以发送多少数据。
>
> 如果只是用 batch，Leader 还是需要等待 Follower 返回才能继续后面的流程，我们这里还可以使用 Pipeline 来进行加速。大家知道，Leader 会维护一个 NextIndex 的变量来表示下一个给 Follower 发送的 log 位置，通常情况下面，只要 Leader 跟 Follower 建立起了连接，我们都会认为网络是稳定互通的。所以当 Leader 给 Follower 发送了一批 log 之后，它可以直接更新

NextIndex，并且立刻发送后面的 log，不需要等待 Follower 的返回。如果网络出现了错误，或者 Follower 返回一些错误，Leader 就需要重新调整 NextIndex，然后重新发送 log 了。

4. 并行追加

对于上面提到的一次 request 简易 Raft 流程来说， Leader 可以先并行的将 log 发送给 Followers，然后再将 log append。为什么可以这么做，主要是因为在 Raft 里面，如果一个 log 被大多数的节点append，我们就可以认为这个 log 是被 committed 了，所以即使 Leader 再给 Follower 发送 log 之后，自己 append log 失败 panic 了，只要 `N / 2 + 1` 个 Follower 能接收到这个 log 并成功 append，我们仍然可以认为这个 log 是被 committed 了，被 committed 的 log 后续就一定能被成功 apply。

那为什么我们要这么做呢？主要是因为 append log 会涉及到落盘，有开销，所以我们完全可以在 Leader 落盘的同时让 Follower 也尽快的收到 log 并 append。

这里我们还需要注意，虽然 Leader 能在 append log 之前给 Follower 发 log，但是 Follower 却不能在 append log 之前告诉 Leader 已经成功 append 这个 log。如果 Follower 提前告诉 Leader 说已经成功 append，但实际后面 append log 的时候失败了，Leader 仍然会认为这个 log 是被 committed 了，这样系统就有丢失数据的风险了。

5. 异步应用

上面提到，当一个 log 被大部分节点 append 之后，我们就可以认为这个 log 被 committed 了，被 committed 的 log 在什么时候被 apply 都不会再影响数据的一致性。所以当一个 log 被 committed 之后，我们可以用另一个线程去异步的 apply 这个 log。

所以整个 Raft 流程就可以变成：

1. Leader 接受一个 client 发送的 request。
2. Leader 将对应的 log 发送给其他 follower 并本地 append。
3. Leader 继续接受其他 client 的 requests，持续进行步骤 2。
4. Leader 发现 log 已经被 committed，在另一个线程 apply。
5. Leader 异步 apply log 之后，返回结果给对应的 client。

使用 asychronous apply 的好处在于我们现在可以完全的并行处理 append log 和 apply log，虽然对于一个 client 来说，它的一次 request 仍然要走完完整的 Raft 流程，但对于多个 clients 来说，整体的并发和吞吐量是上去了。

ETCD 中 Raft 库源码分析

Entry

从整体上来说，一个集群中的每个节点都是一个状态机，而raft管理的就是对这个状态机进行更改的一些操作，这些操作在代码中被封装为一个个`Entry`。

Haskell

```
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/raftpb/raft.pb.go#L203
type Entry struct {
    Term             uint64
    Index            uint64
    Type             EntryType // EntryNormal, EntryConfChange
    Data             []byte // kv pair, ConfChange
}
```

Message

Raft集群中节点之间的通讯都是通过传递不同的`Message`来完成的，这个`Message`结构就是一个非常general的大容器，它涵盖了各种消息所需的字段。

Haskell

```
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/raftpb/raft.pb.go#L239
type Message struct {
    Type             MessageType // 心跳，追加日志，投票,上层应用消息等很多个
    To               uint64 // 接受者
    From             uint64 // 发送者
    Term             uint64 // 任期 逻辑时钟
    LogTerm          uint64 // 发送者最后一条日志的任期号
    Index            uint64 // 如果是投票请求时 其表示发送者最后一条日志的索引号
    Entries          []Entry // 需要存储的日志
    Commit           uint64 // 已提交的所偶音质
    Snapshot         Snapshot // 存放快照
    Reject           bool   // 对方节点拒绝了 当前的请求
    RejectHint       uint64 // 对方节点拒绝了 当前的请求
    Context          []byte 上下文信息 用于跟踪
}
```

log_unstable.go

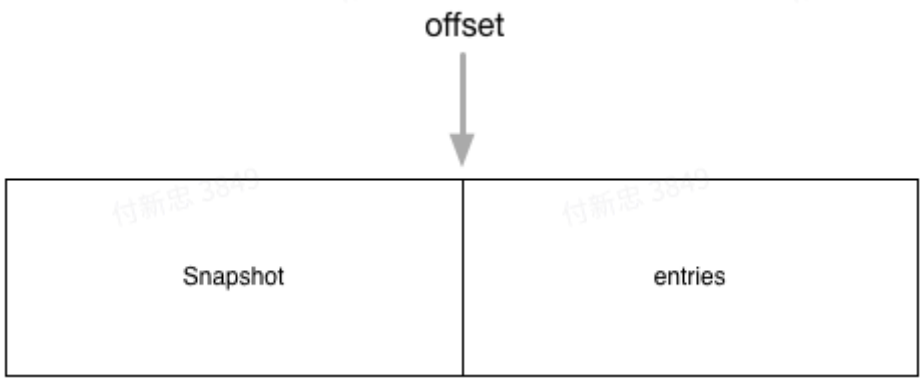
顾名思义，unstable数据结构用于还没有被用户层持久化的数据，它维护了两部分内容`snapshot`和`entries`：

Rust

```rust
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/log_unstable.go#L23
type unstable struct {
    // the incoming unstable snapshot, if any.
    snapshot *pb.Snapshot
    // all entries that have not yet been written to storage.
    entries []pb.Entry
    offset  uint64

    logger Logger
}
```

`entries`代表的是要进行操作的日志，但日志不可能无限增长，在特定的情况下，某些过期的日志会被清空。那这就引入一个新问题了，如果此后一个新的`follower`加入，而`leader`只有一部分操作日志，那这个新`follower`不是没法跟别人同步了吗？所以这个时候`snapshot`就登场了 - 我无法给你之前的日志，但我给你所有之前日志应用后的结果，之后的日志你再以这个`snapshot`为基础进行应用，那我们的状态就可以同步了。因此它们的结构关系可以用下图表示2：



这里的前半部分是快照数据，而后半部分是日志条目组成的数组entries，另外unstable.offset成员保存的是entries数组中的第一条数据在raft日志中的索引，即第i条entries在raft日志中的索引为`i + unstable.offset`。

storage.go

这个文件定义了一个Storage接口，因为etcd中的raft实现并不负责数据的持久化，所以它希望上面的应用层能实现这个接口，以便提供给它查询log的能力。

另外，这个文件也提供了`Storage`接口的一个内存版本的实现MemoryStorage，这个实现同样也维护了`snapshot`和`entries`这两部分，他们的排列跟`unstable`中的类似，也是`snapshot`在前，`entries`在后。从代码中看来`etcdserver`和`raftexample`都是直接用的这个实现来提供log的查询功能的。

log.go

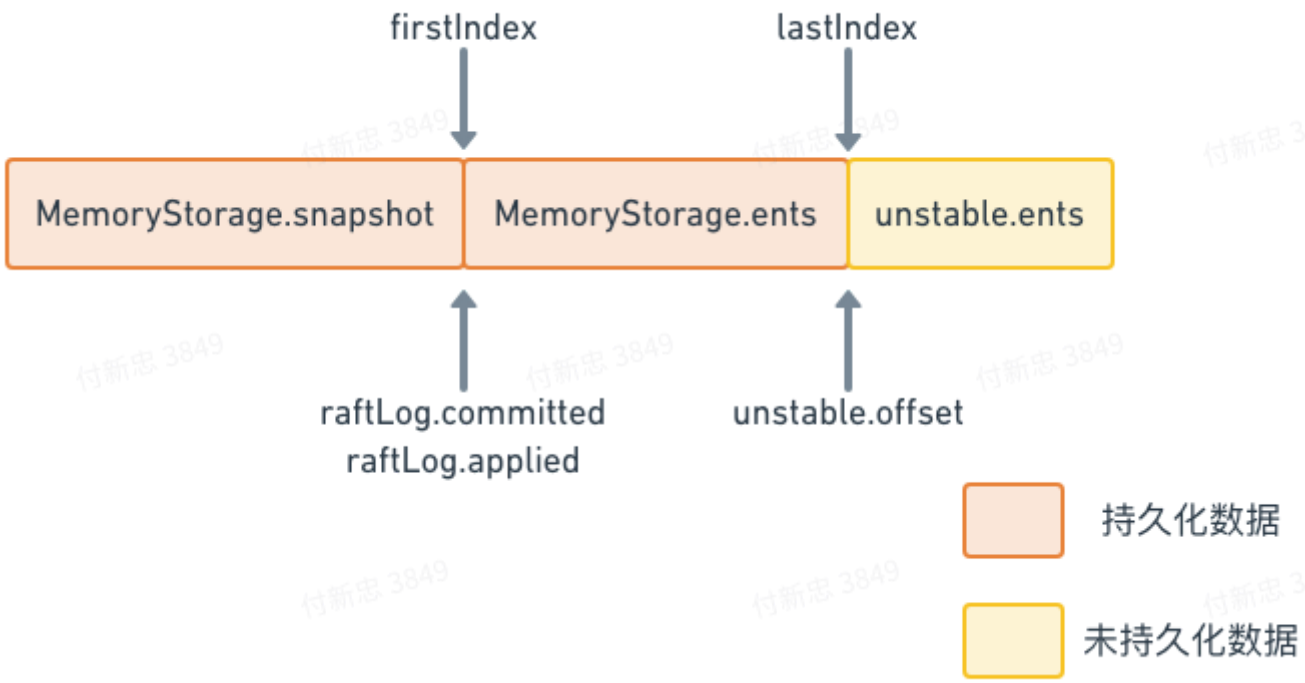
有了以上的介绍unstable、Storage的准备之后，下面可以来介绍raftLog的实现，这个结构体承担了raft日志相关的操作。

raftLog由以下成员组成：

- storage Storage：前面提到的存放已经持久化数据的Storage接口。
- unstable unstable：前面分析过的unstable结构体，用于保存应用层还没有持久化的数据。
- committed uint64：保存当前提交的日志数据索引。
- applied uint64：保存当前传入状态机的数据最高索引。

需要说明的是，一条日志数据，首先需要被提交（committed）成功，然后才能被应用（applied）到状态机中。因此，以下不等式一直成立：`applied <= committed`。

raftLog结构体中，几部分数据的排列如下图所示2：



RaftLog Layout

这个数据排布的情况，可以从raftLog的初始化函数中看出来：

Go

```go
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/log.go#L45
// newLog returns log using the given storage. It recovers the log to the state
// that it just commits and applies the latest snapshot.
func newLog(storage Storage, logger Logger) *raftLog {
    if storage == nil {
        log.Panic("storage must not be nil")
    }
    log := &raftLog{
        storage: storage,
        logger:  logger,
    }
    firstIndex, err := storage.FirstIndex()
    if err != nil {
        panic(err) // TODO(bdarnell)
    }
    lastIndex, err := storage.LastIndex()
    if err != nil {
        panic(err) // TODO(bdarnell)
    }
    log.unstable.offset = lastIndex + 1
    log.unstable.logger = logger
    // Initialize our committed and applied pointers to the time of the last compaction.
    log.committed = firstIndex - 1
    log.applied = firstIndex - 1

    return log
}
```

因此，从这里的代码可以看出，raftLog的两部分，持久化存储和非持久化存储，它们之间的分界线就是lastIndex，在此之前都是`Storage`管理的已经持久化的数据，而在此之后都是`unstable`管理的还没有持久化的数据。

以上分析中还有一个疑问，为什么并没有初始化unstable.snapshot成员，也就是unstable结构体的快照数据？原因在于，上面这个是初始化函数，也就是节点刚启动的时候调用来初始化存储状态的函数，而unstable.snapshot数据，是在启动之后同步数据的过程中，如果需要同步快照数据时才会去进行赋值修改的数据，因此在这里并没有对它进行操作的地方。

progress.go

Leader通过`Progress`这个数据结构来追踪一个follower的状态，并根据`Progress`里的信息来决定每次同步的日志项。这里介绍三个比较重要的属性：

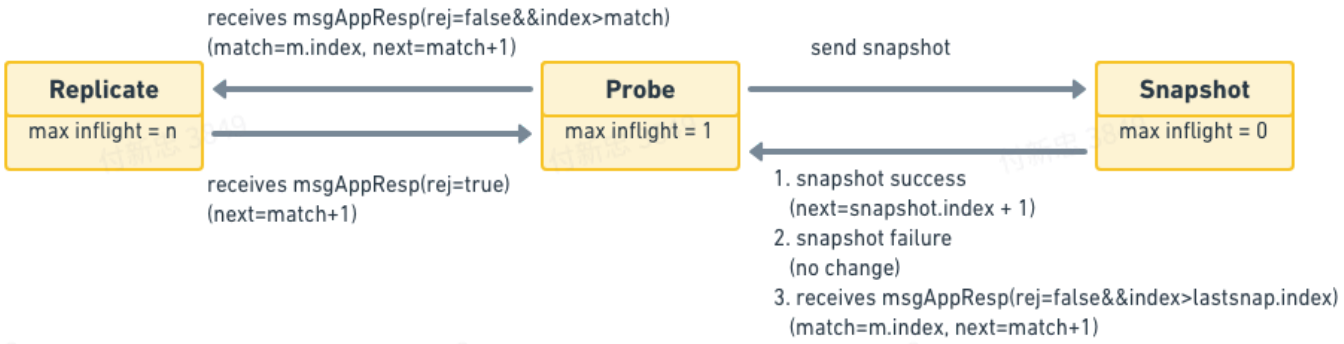
Rust

```rust
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/progress.go#L37
// Progress represents a follower’s progress in the view of the leader. Leader maintains
// progresses of all followers, and sends entries to the follower based on its progress.
type Progress struct {
    Match, Next uint64

    State ProgressStateType // ProgressStateProbe(探索) ProgressStateReplicate(复制) ProgressStateSnapshot(快照)

    ins *inflights // 流量控制 类似tcp滑窗
}
```

综上，`Progress`其实也是个状态机，下面是它的状态转移图：



raft.go

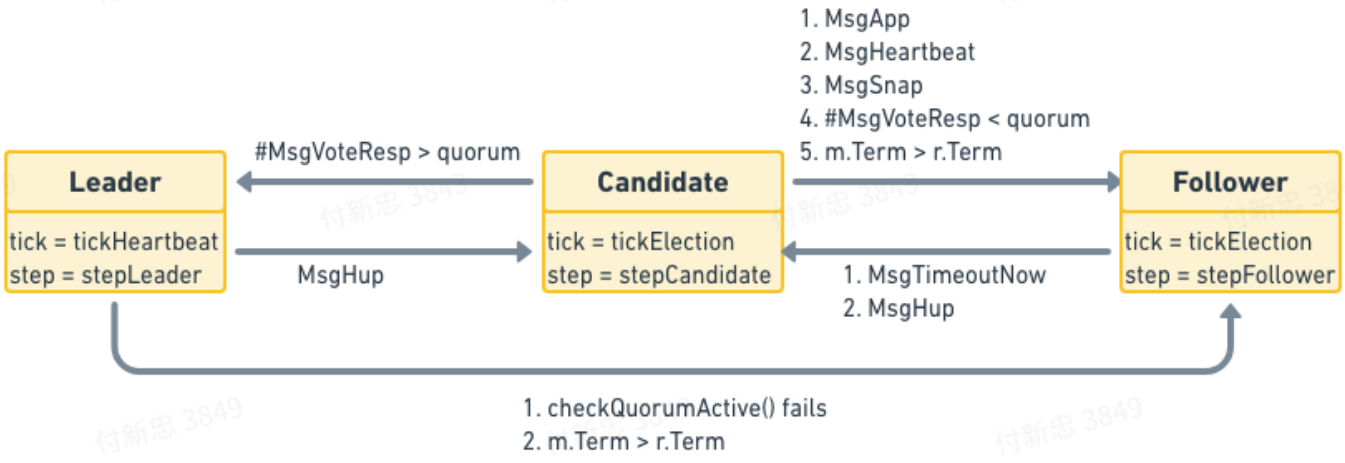
前面铺设了一大堆概念，现在终于轮到实现逻辑了。从名字也可以看出，raft协议的具体实现就在这个文件里。这其中，大部分的逻辑是由`Step`函数驱动的。

Go

```go
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/raft.go#L752
func (r *raft) Step(m pb.Message) error {
    //...
    switch m.Type {
        case pb.MsgHup:
        //...
        case pb.MsgVote, pb.MsgPreVote:
        //...
        default:
            r.step(r, m)
    }
}
```

`Step` 的主要作用是处理不同的消息，所以以后当我们想知道raft对某种消息的处理逻辑时，到这里找就对了。在函数的最后，有个 `default` 语句，即所有上面不能处理的消息都落入这里，由一个小写的 `step` 函数处理，这个设计的原因是什么呢?

其实是因为这里的raft也被实现为一个状态机，它的 `step` 属性是一个函数指针，根据当前节点的不同角色，指向不同的消息处理函数：stepLeader/stepFollower/stepCandidate。与它类似的还有一个 `tick` 函数指针，根据角色的不同，也会在tickHeartbeat和tickElection之间来回切换，分别用来触发定时心跳和选举检测。这里的函数指针感觉像实现了 `OOP` 里的多态。



node.go

`node` 的主要作用是应用层（etcdserver）和共识模块（raft）的衔接。将应用层的消息传递给底层共识模块，并将底层共识模块共识后的结果反馈给应用层。所以它的初始化函数创建了很多用来通信

的`channel`，然后就在另一个`goroutine`里面开始了事件循环，不停的在各种`channel`中倒腾数据（貌似这种由`for-select-channel`组成的事件循环在Go里面很受欢迎）。

Groovy

```
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/node.go#L286
for {
  select {
    case m := <-propc:   // 应用层 输入
        r.Step(m)
    case m := <-n.recvc: // 应用层 输入
        r.Step(m)
    case cc := <-n.confc:
        // Add/remove/update node according to cc.Type
    case <-n.tickc:
        r.tick()
    case readyc <- rd: // 输出
        // Cleaning after result is consumed by application
    case <-advancec:
        // Stablize logs
    case c := <-n.status:
        // Update status
    case <-n.stop:
        close(n.done)
        return
    }
}
```

下面来解释下`readyc`的作用。在etcd的这个实现中，`node`并不负责数据的持久化、网络消息的通信、以及将已经提交的log应用到状态机中，所以`node`使用`readyc`这个`channel`对外通知有数据要处理了，并将这些需要外部处理的数据打包到一个`Ready`结构体中：

SCSS

```
// https://github.com/etcd-io/etcd/blob/v3.3.10/raft/node.go#L52
// Ready encapsulates the entries and messages that are ready to read,
// be saved to stable storage, committed or sent to other peers.
// All fields in Ready are read-only.
type Ready struct {
    // The current volatile state of a Node.
    // SoftState will be nil if there is no update.
    // It is not required to consume or store SoftState.
    *SoftState

```

```
    // The current state of a Node to be saved to stable storage BEFORE
    // Messages are sent.
    // HardState will be equal to empty state if there is no update.
    pb.HardState

    // ReadStates can be used for node to serve linearizable read requests locally
    // when its applied index is greater than the index in ReadState.
    // Note that the readState will be returned when raft receives msgReadIndex.
    // The returned is only valid for the request that requested to read.
    ReadStates []ReadState

    // Entries specifies entries to be saved to stable storage BEFORE
    // Messages are sent.
    Entries []pb.Entry

    // Snapshot specifies the snapshot to be saved to stable storage.
    Snapshot pb.Snapshot

    // CommittedEntries specifies entries to be committed to a
    // store/state-machine. These have previously been committed to stable
    // store.
    CommittedEntries []pb.Entry

    // Messages specifies outbound messages to be sent AFTER Entries are
    // committed to stable storage.
    // If it contains a MsgSnap message, the application MUST report back to raft
    // when the snapshot has been received or has failed by calling ReportSnapshot.
    Messages []pb.Message

    // MustSync indicates whether the HardState and Entries must be synchronously
    // written to disk or if an asynchronous write is permissible.
    MustSync bool
}
```

应用程序得到这个`Ready`之后，需要：

1. 将HardState, Entries, Snapshot持久化到storage。
2. 将Messages广播给其他节点。
3. 将CommittedEntries（已经commit还没有apply）应用到状态机。

4. 如果发现CommittedEntries中有成员变更类型的entry，调用`node.ApplyConfChange()`方法让`node`知道。
5. 最后再调用`node.Advance()`告诉raft，这批状态更新处理完了，状态已经演进了，可以给我下一批Ready让我处理。

Life of a Request

前面我们把整个包的结构过了一遍，下面来结合具体的代码看看raft对一个请求的处理过程是怎样的。我一直觉得，如果能从代码的层面追踪到一个请求的处理过程，那无论是从宏观还是微观的角度，对理解整个系统都是非常有帮助的。

Life of a Vote Request

1. 首先，在`node`的大循环里，有一个会定时输出的`tick channel`，它来触发`raft.tick()`函数。根据上面的介绍可知，如果当前节点是follower，那它的`tick`函数会指向`tickElection`。`tickElection`的处理逻辑是给自己发送一个`MsgHup`的内部消息，`Step`函数看到这个消息后会调用`campaign`函数，进入竞选状态。

Go

```go
// tickElection is run by followers and candidates after r.electionTimeout.
func (r *raft) tickElection() {
    r.electionElapsed++

    if r.promotable() && r.pastElectionTimeout() {
        r.electionElapsed = 0
        r.Step(pb.Message{From: r.id, Type: pb.MsgHup})
    }
}

func (r *raft) Step(m pb.Message) error {
    //...
    switch m.Type {
    case pb.MsgHup:
        r.campaign(campaignElection)
    }
}
```

2. `campaign`则会调用`becomeCandidate`把自己切换到candidate模式，并递增`Term`值。然后再将自己的`Term`及日志信息发送给其他的节点，请求投票。

Groovy

```
func (r *raft) campaign(t CampaignType) {
    //...
    r.becomeCandidate()
    // Get peer id from progress
    for id := range r.prs {
        //...
        r.send(pb.Message{Term: term, To: id, Type: voteMsg, Index: r.raftLog.lastIndex(), LogTerm: r.raftLog.lastTerm(), Context: ctx})
    }
}
```

3. 另一方面，其他节点在接受到这个请求后，会首先比较接收到的`Term`是不是比自己的大，以及接受到的日志信息是不是比自己的要新，从而决定是否投票。这个逻辑我们还是可以从`Step`函数中找到：

Groovy

```
func (r *raft) Step(m pb.Message) error {
    //...
    switch m.Type {
    case pb.MsgVote, pb.MsgPreVote:
        // We can vote if this is a repeat of a vote we've already cast...
        canVote := r.Vote == m.From ||
            // ...we haven't voted and we don't think there's a leader yet in this term...
            (r.Vote == None && r.lead == None) ||
            // ...or this is a PreVote for a future term...
            (m.Type == pb.MsgPreVote && m.Term > r.Term)
        // ...and we believe the candidate is up to date.
        if canVote && r.raftLog.isUpToDate(m.Index, m.LogTerm) {
            r.send(pb.Message{To: m.From, Term: m.Term, Type: voteRespMsgType(m.Type)})
        } else {
            r.send(pb.Message{To: m.From, Term: r.Term, Type: voteRespMsgType(m.Type), Reject: true})
        }
    }
}
```

4. 最后当candidate节点收到投票回复后，就会计算收到的选票数目是否大于所有节点数的一半，如果大于则自己成为leader，并昭告天下，否则将自己置为follower：

Groovy

```
unc (r *raft) Step(m pb.Message) error {
    //...
    switch m.Type {
    case myVoteRespType:
        gr := r.poll(m.From, m.Type, !m.Reject)
        switch r.quorum() { // 是否满足法定人数
        case gr:
            if r.state == StatePreCandidate {
                r.campaign(campaignElection)
            } else {
                r.becomeLeader()
                r.bcastAppend()
            }
        case len(r.votes) - gr:
            r.becomeFollower(r.Term, None)
    }
}
```

Life of a Write Request

1. 一个写请求一般会通过调用`node.Propose`开始，`Propose`方法将这个写请求封装到一个`MsgProp`消息里面，发送给自己处理。
2. 消息处理函数`Step`无法直接处理这个消息，它会调用那个小写的`step`函数，来根据当前的状态进行处理。

如果当前是follower，那它会把这个消息转发给leader。

Go

```
func stepFollower(r *raft, m pb.Message) error {
    switch m.Type {
    case pb.MsgProp:
        //...
        m.To = r.lead
        r.send(m)
    }
}
```

3. Leader收到这个消息后（不管是follower转发过来的还是自己内部产生的）会有两步操作：
    a. 将这个消息添加到自己的log里

b. 向其他follower广播这个消息

Go

```go
func stepLeader(r *raft, m pb.Message) error {
   switch m.Type {
     case pb.MsgProp:
         //...
         if !r.appendEntry(m.Entries...) {
             return ErrProposalDropped
         }
         r.bcastAppend()
         return nil
     }
}
```

4. 在follower接收完这个log后，会返回一个`MsgAppResp`消息。
5. 当leader确认已经有足够多的follower接受了这个log后，它首先会commit这个log，然后再广播一次，告诉别人它的commit状态。这里的实现就有点像两阶段提交了。

Go

```go
func stepLeader(r *raft, m pb.Message) error {
    switch m.Type {
    case pb.MsgAppResp:
        //...
        if r.maybeCommit() {
            r.bcastAppend()
        }
    }
}

// maybeCommit attempts to advance the commit index. Returns true if
// the commit index changed (in which case the caller should call
// r.bcastAppend).
func (r *raft) maybeCommit() bool {
    //...
    mis := r.matchBuf[:len(r.prs)]
    idx := 0
    for _, p := range r.prs {
        mis[idx] = p.Match
        idx++
    }
    sort.Sort(mis)
    mci := mis[len(mis)-r.quorum()]
    return r.raftLog.maybeCommit(mci, r.Term)
}
```

小作业

> 动手实现一个raft算法包

参考资料

1. Raft 官网
2. raft 算法动画讲解
3. raft算法演示
4. TIKV优化
5. ETCD源码分析
6. Raft的Bug

7. Raft 算法原理
8. Etcd raft库实现
9. Raft 源码