Raft算法总结

1. 3个不同的角色：

Follower -> Candidate -> Leader
(Follower 超时转为Candidate, Candidate收到majority的回复转为Leader)
Leader -> Follower
(Leader收到比自己更新的term reply,则降级为follower)
Candidate-> Follower
(如果reply.Term> rf.Term,则证明有其他server term更新一些,那么转为follower)

2. State和持久化

Persistent on all servers:
CurrentTerm: 当前这台raft server目前的term
VotedFor:这台raft server在目前的term 投票给了哪个
Log[]: First Index is 1, and all log received from Leaders

Volatile on all servers:
CommitIndex:最大的已经提交的索引号,初始化为0
LastApplied: 最后被应用到状态的日志条目索引,initial to 0

Volatile on Leaders:
nextIndex[]:对于每一个服务器，需要发送给他的下一个日志条目的索引值，初始化为当前领导人的最后的日志索引值加1
matchIndex[]:对于每一个服务器，已经复制给他的日志的最高索引值,initial to 0

RequestVote(candidate发出):
Term:candidate当前的term
CandidateID:candidate当前的term
LastLogIndex:candidate最后一条log Entry的Index
LastLogTerm:candidate最后一条log Entry的Term

RequestVoteResults(candidate接收):
Term:接收到的voter的当前term
voteGranted:voter是否投票成功

RequestAppendEntries(Leader发出):
Term: Leader当前的term
LeaderID: Leader当前的term
PrevLogIndex:当前要附加的日志entries的上一条的日志索引
PrevLogTerm:当前要附加的日志entries的上一条的日志任期号
Entries[]: 需要附加的日志条目（心跳时为空）
LeaderCommit://// 当前领导人已经提交的最大的日志索引值

RequestAppendEntriesResults(Leader接收):
Term: **Follower**的current Term
Success:如果返回yes,那么证明follower的日志在PrevLogIndex处匹配成功

3. 两个Timer on each server,每个server都有
(Election Timer过时开始选举,HeartBeat Timer过时开始发送心跳)
Candidate 和 Follower：开启Election Timer 但是关闭 HeartBeatTimer
leader：开启Election Timer 但是关闭 HeartBeatTimer

4. 角色的职责：

Follower: Stop heartBeatTimer+Reset Election timer
Candidate: Start Election +Reset Election timer
Follower: initialize matchIndex and NextIndex 并且 start to send heartbeat

5. 2个被动方法，每个server都有, 通过RPC进行通信：

5.1 RequestVote(Voter处理Candidate的请求)：

5.1.1.

请求者的任期比自己小，那么拒绝投票
OR
voter并没有投票给candidate,那么拒绝投票
reply.term总是指向最新的term

5.1.2.

如果请求的任期比自己大，那么自己切换为Follower
如果请求中的任期比自己的大，对方的日志并不一定比自己的新，还需要进一步判断

5.1.3.

Candidate最新的一条日志的term 不可以 比voter最新的一条日志的term更旧
OR
两者的term是同样的新，但是candidate的日志长度不可以比voter的短

5.2 AppendEntries(Follower处理Leader的请求)：

5.2.1.

判断附加日志任期Term和当前的Term是否相同：
如果请求的Term小于当前的Term，那么说明收到了来自过期的领导人的附加日志请求，那么拒接处理。
如果请求的Term大于当前的Term，那么更新当前的Term为请求的Term,并且转为follower。

5.2.2.

判断preLogIndex是否大于当前的日志长度:

如果preLogIndex的长度大于当前的日志的长度，那么说明跟随者缺失日志，那么拒绝附加日志，返回false,

并且ConflictTerm =-1 and ConflictIndex=follower的日志长度

或者

preLogIndex位置处的任期是否和preLogTerm相等以检测要附加的日志之前的日志是否匹配:

如果preLogIndex处的任期和preLogTerm不相等，那么说明日志有冲突，拒绝附加日志，返回false

并且ConflictTerm =发生冲突的follower的log term

ConflictIndex=这个term在follower log 中的第一条Log Index

5.2.3.

如果term和prevlogindex都没有问题，那么开始append Log

从PrevlogIndex+1的位置开始，

检查此处位置follower是否存在log，

如果不存在直接把leader的log append进去

如果存在开始逐个查看follower和leader的log，有不一样的位置，直接截断后面的所有log，把leader的加进去

5.2.4.

Append Log结束之后，进行提交commitIndex，

rf.commitIndex= Min(leader.leaderCommitIndex, len(rf.log)-1)

6.两个主动方法(
sendAppendEntries(leader)+
sendVoteRequest(Candidate))

6.1 sendVoteRequest Candidate处理来自voter的回复

所有的接受消息处理例程都要考虑到 network delay，也就是说收到的消息是过时的，这个要正确处理
Candidate 检查是否自己仍然是Candidate,并且VoteGranted超过majority,如果是那么就升级为leader.
否则如果是voter.Term> Candidate.Term,那么就降级为 Follower

6.2 sendAppendEntries Leader处理来自Follower的回复

所有的接受消息处理例程都要考虑到 network delay，也就是说收到的消息是过时的，这个要正确处理
6.2.1
Leader先检查自己是否仍然是Leader，
6.2.2
然后检查Term
比对响应的Term和当前的Term以确认自己是否过期：
如果响应的Term大于当前的Term，那么说明当前的领导人已经过期，马上将自己切换为跟随者。
如果响应的Term小于当前的Term，那么说明当前的收到了过期的响应（可能网路延迟导致，那么忽略过期任期的响应
6.2.3
然后判断reply.Success，并且更新nextIndex
如果Success=False 要判断两种情况
第一种：follower log 长度不够，更新nextIndex为follower的日志长度
第二种长度没有问题, 但是在prevLogIndex位置的term不一样
所以在leader中从prevLogIndex往后找, unitl找到和conflict term一样term的第一个index 作为 nextIndex

假如找不到，nextIndex 作为follower中的conflict term中的第一条的index
如果Success=True,
那么nextIndex=PrevLogIndex+len(entries)
matchIndex= nextIndex+1

6.2.4
前面的步骤都完成之后，可以进行commitIndex,
从len(log)-1到commitIndex,
查找leader上的每一条logEntry,
去判断有多少个server的matchIndex大于这条LogEntry的index,
假如在某个commitIndex上找到了数量超过n/2+1，那么认为这条logEntry已经被提交了
那么就把这个commitIndex和lastApplied之间的command发送给ApplyChannel,
Raft layer 通过ApplyChannel 把命令传输给 KV layer, 上层的数据库开始进行操作。