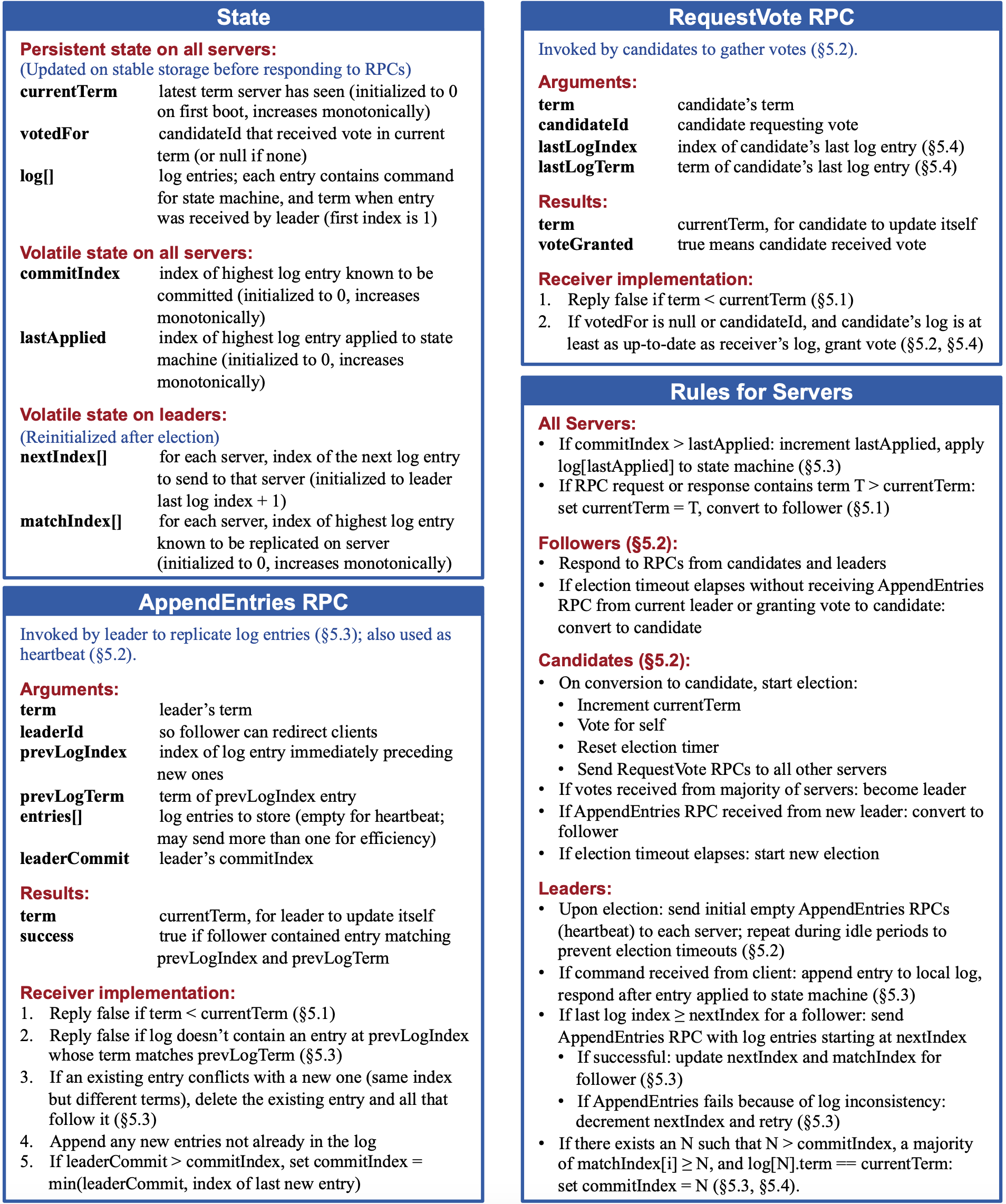
**[mit 6.824] lab2：Raft**

在做该实验前，需要熟读论文《**In Search of an Understandable Consensus Algorithm (Extended Version)**，主要根据 Figure 2 来实现代码。建议多读几遍论文，尤其是第五章。写之前可以首先参考 6.824 课程对 [locking](https://pdos.csail.mit.edu/6.824/labs/raft-locking.txt) 和 [structure](https://pdos.csail.mit.edu/6.824/labs/raft-structure.txt) 的描述，然后再参考 6.824 TA 的 [guidance](https://thesquareplanet.com/blog/students-guide-to-raft/) 。

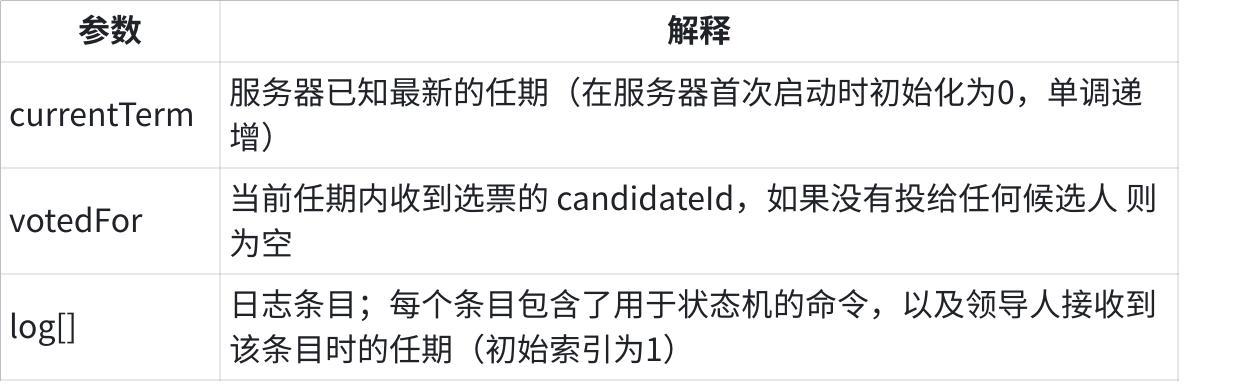
**1 算法核心**

文章给出了一张算法简略版本的图，这也是实现Raft算法的关键，包含以下几个部分。



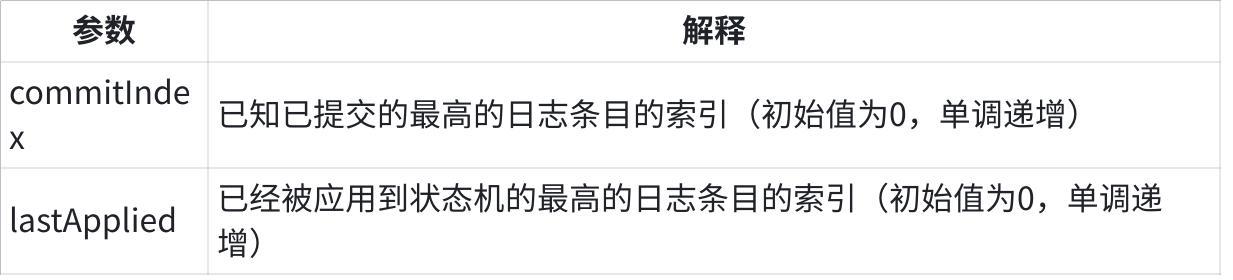
**状态**：

所有服务器上的持久性状态 (在响应 RPC 请求之前，已经更新到了稳定的存储设备)



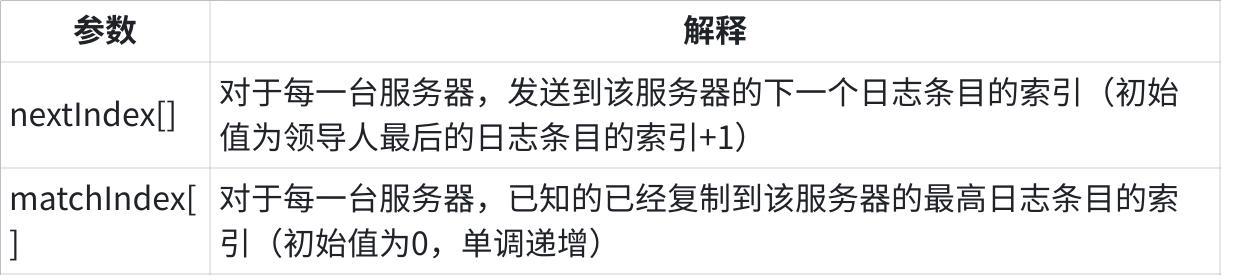
**点击图片可查看完整电子表格**

所有服务器上的易失性状态



**点击图片可查看完整电子表格**

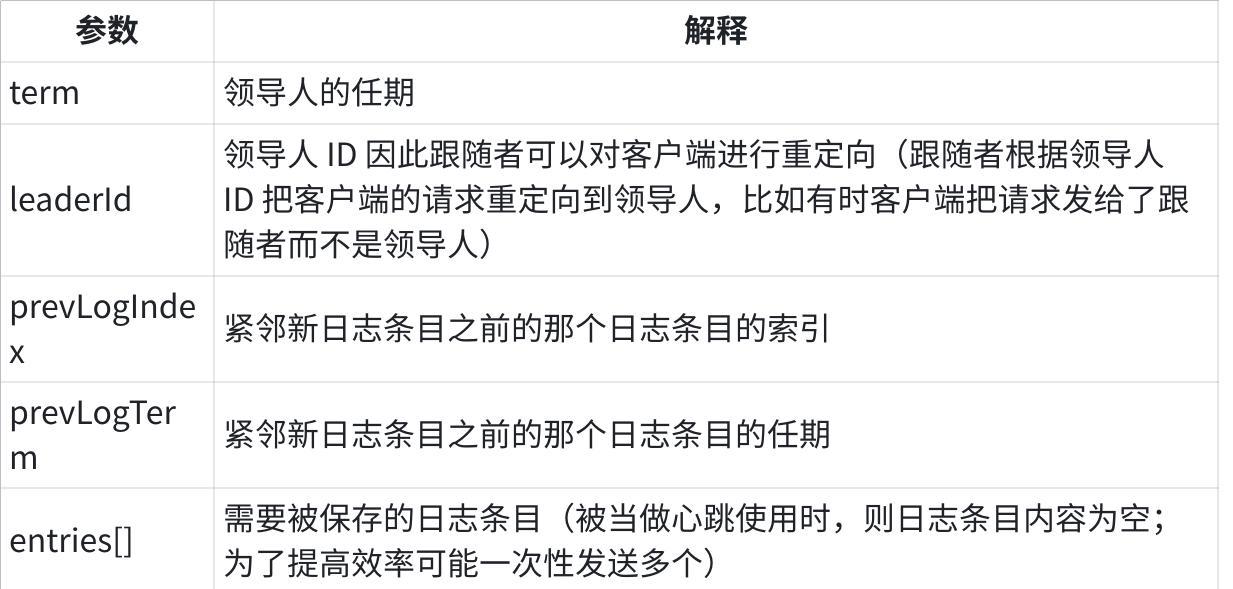
领导人（服务器）上的易失性状态 (选举后已经重新初始化)



**点击图片可查看完整电子表格**

**追加条目（AppendEntries）RPC**：

由领导人调用，用于日志条目的复制，同时也被当做心跳使用



**点击图片可查看完整电子表格**



**点击图片可查看完整电子表格**

接收者的实现：

1. 返回假 如果领导人的任期小于接收者的当前任期（这里的接收者是指跟随者或者候选人）
2. 返回假 如果接收者日志中没有包含这样一个条目 即该条目的任期在 prevLogIndex 上能和 prevLogTerm 匹配上 （在接收者日志中 如果能找到一个和 prevLogIndex 以及 prevLogTerm 一样的索引和任期的日志条目 则继续执行下面的步骤 否则返回假）
3. 如果一个已经存在的条目和新条目（即刚刚接收到的日志条目）发生了冲突（因为索引相同，任期不同），那么就删除这个已经存在的条目以及它之后的所有条目
4. 追加日志中尚未存在的任何新条目
5. 如果领导人的已知已提交的最高日志条目的索引大于接收者的已知已提交最高日志条目的索引（leaderCommit > commitIndex），则把接收者的已知已经提交的最高的日志条目的索引commitIndex 重置为 领导人的已知已经提交的最高的日志条目的索引 leaderCommit 或者是 上一个新条目的索引 取两者的最小值

**请求投票（RequestVote）RPC**：

由候选人负责调用用来征集选票



**点击图片可查看完整电子表格**



**点击图片可查看完整电子表格**

接收者实现：

1. 如果term < currentTerm返回 false
2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他

**所有服务器需遵守的规则**：

所有服务器：

* 如果commitIndex > lastApplied，则 lastApplied 递增，并将log[lastApplied]应用到状态机中
* 如果接收到的 RPC 请求或响应中，任期号T > currentTerm，则令 currentTerm = T，并切换为跟随者状态

跟随者：

* 响应来自候选人和领导人的请求
* 如果在超过选举超时时间的情况之前没有收到**当前领导人**（即该领导人的任期需与这个跟随者的当前任期相同）的心跳/附加日志，或者是给某个候选人投了票，就自己变成候选人

候选人：

* 在转变成候选人后就立即开始选举过程
* 自增当前的任期号（currentTerm）
* 给自己投票
* 重置选举超时计时器
* 发送请求投票的 RPC 给其他所有服务器
* 如果接收到大多数服务器的选票，那么就变成领导人
* 如果接收到来自新的领导人的附加日志（AppendEntries）RPC，则转变成跟随者
* 如果选举过程超时，则再次发起一轮选举

领导人：

* 一旦成为领导人：发送空的附加日志（AppendEntries）RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以防止跟随者超时
* 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端
* 如果对于一个跟随者，最后日志条目的索引值大于等于 nextIndex（lastLogIndex ≥ nextIndex），则发送从 nextIndex 开始的所有日志条目：
* 如果成功：更新相应跟随者的 nextIndex 和 matchIndex
* 如果因为日志不一致而失败，则 nextIndex 递减并重试
* 假设存在 N 满足N > commitIndex，使得大多数的 matchIndex[i] ≥ N以及log[N].term == currentTerm 成立，则令 commitIndex = N

**2 实现**

raft结构体，我和原文几乎定义是一样的。之后还需在Make初始化。

|  |
| --- |
| Go type Raft struct {  mu sync.Mutex // Lock to protect shared access to this peer's state  peers []\*labrpc.ClientEnd // RPC end points of all peers  persister \*Persister // Object to hold this peer's persisted state  me int // this peer's index into peers[]  dead int32 // set by Kill()   // state a Raft server must maintain.  //persistent state on all servers  currentTerm int //latest term server has seen (initialized to 0 on first boot, increases monotonically)  votedFor int //candidateId that received vote in current term (or null if none)  log []LogEntry //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 int //index of highest log entry known to be committed (initialized to 0, increases monotonically)  lastApplied int //index of highest log entry applied to state machine (initialized to 0, increases monotonically)   //Volatile state on leaders:  nextIndex []int //for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)  matchIndex []int //for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)   //something else  State int //状态  electiontimer \*time.Ticker //每个节点中的计时器,判断选举是否超时,选举计时器  applyCh chan ApplyMsg //client从applych取日志  cond \*sync.Cond //sync.Cond可以用于等待和通知goroutine，等待发送applyentry通知 } |

**2.1 选主**

选主的实现较为简单。首先定义RequestVoteArgs和RequestVoteReply，和原文一样。

|  |
| --- |
| Go type RequestVoteArgs struct {  Term int //candidate’s term  CandidateId int //candidate requesting vote  LastLogIndex int //index of candidate’s last log entry  LastLogTerm int //term of candidate’s last log entry }  type RequestVoteReply struct {  Term int //currentTerm, for candidate to update itself  VoteGranted bool //true means candidate received vote } |

需要注意的是只有在 grant 投票时才重置选举超时时间，这样有助于网络不稳定条件下选主的 liveness 问题。ticker中选举时间到期，follower提升为candidate发起选举。在 Raft 中，有两种超时机制，分别是**选举超时**和**心跳超时**，当节点选举超时后，它会增加自己当前的任期号并成为 candidate 竞选 leader，如果能获得超过半数的选票即赢得了选举，从而成为 leader。节点的选举超时是随机的，以避免节点同时发起选举。我们是需要周期性的进行选举的，因此肯定是需要实现一个选举计时器的，那么我们可以直接实现如下代码，使用time.Timer或time.Ticker来作为定时器。

|  |
| --- |
| Go //start a ticker to start election func (rf \*Raft) ticker() {  for rf.killed() == false {  select {  case <-rf.electiontimer.C:  //一段时间（election timeout）没收到其他节点的消息时，通过RequestVote RPC发起选举  rf.mu.Lock()  if rf.killed() {  rf.mu.Unlock()  break  }  if rf.State != Leader {  //fmt.Printf("id[%d].state[%v].term[%d]: 选举计时器到期\n", rf.me, rf.State, rf.currentTerm)  rf.StartElection()  }  rf.electiontimer.Reset(GetRamdomTimeout())  rf.mu.Unlock()  }  } } |

对于每个peers，发送RequestVote，等待回应。当reply.Term == rf.currentTerm && reply.VoteGranted && rf.State == Candidate，选票数加1。当选票数量超过1/2，则成为leader。如果回复的term大于当前term，说明已经有leader了，则变为follower。

|  |
| --- |
| Go func (rf \*Raft) StartElection() {  //candidate startelection  rf.BecomeCandidate()  defer rf.persist()  votenum := 1  for i := range rf.peers {  if i == rf.me || rf.State != Candidate {  continue  }  requestvoteargs := &RequestVoteArgs{  Term: rf.currentTerm,  CandidateId: rf.me,  LastLogIndex: rf.lastLogIndex(),  LastLogTerm: rf.lastLogTerm(),  }  go func(i int) {  //requestvote simulaneously  reply := &RequestVoteReply{}  ok := rf.sendRequestVote(i, requestvoteargs, reply)  if ok {  rf.mu.Lock()  defer rf.mu.Unlock()  if reply.Term == rf.currentTerm && reply.VoteGranted && rf.State == Candidate {  votenum++  if votenum > len(rf.peers)/2 {  rf.BecomeLeader()  }  rf.persist()  } else if reply.Term > rf.currentTerm {  rf.votedFor = -1  rf.currentTerm = reply.Term  rf.BecomeFollower()  rf.persist()  }  }  }(i)  } } |

RequestVote就是给Candidate投票。选举人term小于当前term，拒绝给选举人投票。选举人term大于当前term，当前rf变为follower。一定要匹配，即如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，才投票。什么叫和自己一样新：candidateTerm > 当前term 或者 (candidateTerm == 当前term && candidateLogIndex >= 当前Logindex)。

|  |
| --- |
| Go func (rf \*Raft) RequestVote(args \*RequestVoteArgs, reply \*RequestVoteReply) {  rf.mu.Lock()  defer rf.mu.Unlock()  defer rf.persist()  if args.Term < rf.currentTerm {  // reject request with stale term number  reply.Term = rf.currentTerm  reply.VoteGranted = false  return  }   if args.Term > rf.currentTerm {  rf.currentTerm = args.Term  rf.BecomeFollower()  rf.votedFor = -1  }   reply.Term = rf.currentTerm  reply.VoteGranted = false  if (rf.votedFor == -1 || rf.votedFor == args.CandidateId) && rf.Isuptodate(args.LastLogTerm, args.LastLogIndex) {  rf.votedFor = args.CandidateId  reply.VoteGranted = true  reply.Term = rf.currentTerm  rf.electiontimer.Reset(GetRamdomTimeout())  } } |

选举成功后，成为leader，leader 会周期性的向其它节点发送心跳，已告知其它节点现在已经存在 leader 了。心跳也会包含日志同步信息。当心跳超时，即follower 一段时间没有收到心跳，那么它就会认为系统中没有可用的 leader，当选举超时后就会发起选举。

**2.2 日志复制**

这一模块需要实现Raft之间的日志复制，重点是实现AppendEntries RPC以及发送AppendEntries RPC的方法sendAppendEntries。RPC定义和原文一致。

|  |
| --- |
| Go type RequestVoteArgs struct {  Term int //candidate’s term  CandidateId int //candidate requesting vote  LastLogIndex int //index of candidate’s last log entry  LastLogTerm int //term of candidate’s last log entry }  type RequestVoteReply struct {  Term int //currentTerm, for candidate to update itself  VoteGranted bool //true means candidate received vote } |

leader的周期HeartBeat发送函数会发送心跳或者日志给followers。通过调用sendAppendEntry，再由sendAppendEntry的rf.peers[server].Call("Raft.AppendEntries", args, reply)调用AppendEntries

* 如果领导人的任期小于接收者的当前任期，返回假。
* 如果接收到来自新的领导人的附加日志（AppendEntries）RPC，则转变成跟随者。
* 在接收者日志中 如果能找到一个和 prevLogIndex 以及 prevLogTerm 一样的索引和任期的日志条目，则继续执行下面的步骤，否则返回假。
* 如果一个已经存在的条目和新条目（即刚刚接收到的日志条目）发生了冲突（因为索引相同，任期不同），那么就删除这个已经存在的条目以及它之后的所有条目。
* 追加日志中尚未存在的任何新条目。
* 更新commitIndex。

|  |
| --- |
| Go func (rf \*Raft) AppendEntries(args \*AppendEntriesArgs, reply \*AppendEntriesReply) {  rf.mu.Lock()  defer rf.mu.Unlock()  defer rf.persist()   //1.返回假 如果领导人的任期小于接收者的当前任期  if args.Term < rf.currentTerm {  reply.Term = rf.currentTerm  reply.Success = false  reply.NextTryIndex = rf.lastLogIndex() + 1  return  }   if args.Term > rf.currentTerm {  //如果接收到来自新的领导人的附加日志（AppendEntries）RPC，则转变成跟随者  rf.currentTerm = args.Term  rf.BecomeFollower()  rf.votedFor = -1  }   reply.Term = rf.currentTerm  rf.electiontimer.Reset(GetRamdomTimeout())  //2.在接收者日志中 如果能找到一个和 prevLogIndex 以及 prevLogTerm 一样的索引和任期的日志条目 则继续执行下面的步骤 否则返回假  if args.PrevLogIndex > rf.lastLogIndex() {  reply.Success = false  reply.NextTryIndex = rf.lastLogIndex() + 1  return  }   //3.如果一个已经存在的条目和新条目（即刚刚接收到的日志条目）发生了冲突（因为索引相同，任期不同），那么就删除这个已经存在的条目以及它之后的所有条目  baseIndex := rf.log[0].Index  if args.PrevLogIndex >= baseIndex && args.PrevLogTerm != rf.log[args.PrevLogIndex-baseIndex].Term {  term := rf.log[args.PrevLogIndex-baseIndex].Term  for i := args.PrevLogIndex - 1; i >= baseIndex; i-- {  if rf.log[i-baseIndex].Term != term {  reply.NextTryIndex = i + 1  reply.Success = false  return  }  }  }   //4.追加日志中尚未存在的任何新条目  rf.log = rf.log[:args.PrevLogIndex+1]  rf.log = append(rf.log, args.Entries...)  //5.if leadercommit>commitdex,set commitindex=min(leadercommit,index of last new entry)  if args.LeaderCommit > rf.commitIndex {  lastlogIndex := rf.lastLogIndex()  rf.commitIndex = min(args.LeaderCommit, lastlogIndex)  rf.cond.Broadcast()  }   rf.electiontimer.Reset(GetRamdomTimeout())  reply.Term, reply.Success, reply.NextTryIndex = rf.currentTerm, true, args.PrevLogIndex+len(args.Entries)  return } |

sendAppendEntry还会接受AppendEntries返回的结果，判断leader是否apply。reply.Term > rf.currentTerm则变回为follower。如果成功：更新相应跟随者的 nextIndex 和 matchIndex，因为日志不一致而失败，则 nextIndex 递减并重试。然后根据论文“假设存在 N 满足N > commitIndex，使得大多数的 matchIndex[i] ≥ N以及log[N].term == currentTerm 成立，则令 commitIndex = N”，判断大多数follower已经apply，leader则apply。

|  |
| --- |
| Go func (rf \*Raft) sendAppendEntry(server int, args \*AppendEntriesArgs, reply \*AppendEntriesReply) bool {  ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)  rf.mu.Lock()  defer rf.mu.Unlock()   if !ok || reply.Term != rf.currentTerm || rf.State != Leader {  //fmt.Println("send append entries error")  return ok  }   if reply.Term > rf.currentTerm {  rf.currentTerm = reply.Term  rf.BecomeFollower()  rf.votedFor = -1  rf.persist()  return ok  }   //>1/2 and leader apply  if reply.Success {  //如果成功：更新相应跟随者的 nextIndex 和 matchIndex  if len(args.Entries) > 0 {  //rf.nextIndex[server] = args.Entries[len(args.Entries)-1].Index + 1  //rf.matchIndex[server] = rf.nextIndex[server] - 1  rf.matchIndex[server] = args.PrevLogIndex + len(args.Entries)  rf.nextIndex[server] = rf.matchIndex[server] + 1  }  } else {  //因为日志不一致而失败，则 nextIndex 递减并重试  rf.nextIndex[server] = reply.NextTryIndex  }   //假设存在 N 满足N > commitIndex，使得大多数的 matchIndex[i] ≥ N以及log[N].term == currentTerm 成立，则令 commitIndex = N  for N := rf.lastLogIndex(); N > rf.commitIndex && rf.log[N].Term == rf.currentTerm; N-- {  countvotes := 1  for i := range rf.peers {  if i == rf.me {  continue  }  if rf.matchIndex[i] >= N {  countvotes++  }  }  if countvotes > len(rf.peers)/2 {  rf.commitIndex = N  rf.cond.Broadcast()  break  }  }   return ok } |

不要单独启go apply()，因为可能导致apply顺序不一致。应该只有一个apply进程，才用信号量来判断是否apply。

|  |
| --- |
| Go func (rf \*Raft) ApplyEntries() {  //将logEntry应用到状态机  for {  rf.mu.Lock()  if rf.lastApplied >= rf.commitIndex {  rf.cond.Wait()  }  rf.lastApplied++  applymsg := ApplyMsg{  CommandValid: true,  Command: rf.log[rf.lastApplied].Command,  CommandIndex: rf.log[rf.lastApplied].Index,  }  rf.applyCh <- applymsg  rf.mu.Unlock()  } } |

**2.3 持久化**

由于我们现在都是保存在内存中的，那么断电即失，因此我们肯定是需要持久化保存起来的，比如说写入磁盘中。通过封装 gob 对象 persister 来模拟数据持久化到磁盘的过程，需要持久的状态有 3 个currentTerm、votedFor和log[]。对votedFor持久化是为了**保证一个节点在一个 term 中，最多只给一个 candidate 投票**，从而保证在每个 term 中最多只有一个 leader 能被选举出来。因此，currentTerm, voteFor 和 logs 这三个变量一旦发生变化就一定要在被其他协程感知到之前（释放锁之前，发送 rpc 之前）持久化，这样才能保证原子性。

**3 总结&采坑**

* defer的使用：注意看助教给的guidance给出容易死锁的情况。
* figure 8 （unreliable）：我在该测试中，我偶尔会失败一次，一直在找原因。由于我实现的时候是根据figure8不允许提交任期之前的日志来做的，所以我在找其他原因。我经过多次修改，发现在appendentries的时候，Becomefollower被我放在了外面，因此发生只要收到appendentryRPC，就会变成Follower，这是错误的。

|  |
| --- |
| Go  if args.Term > rf.currentTerm {  rf.currentTerm = args.Term  rf.BecomeFollower()  rf.votedFor = -1  } |

但是测试还是会偶尔会失败一次。说明这个bug不会影响其他测试，这是官方给的测试没有检测出来的地方。后来我发现我的heartbeattimeout设置的150ms，electiontimeout是200ms+random(200ms)。论文提到heartbeattimeout要远小于electiontimeout，我就改成60ms，就可以稳定通过的（100次测试）。

* apply：只能有一个go协程。
* apply error：发送 RPC 请求和处理 RPC 返回体时都需要判断一次自身状态是否为Leader。