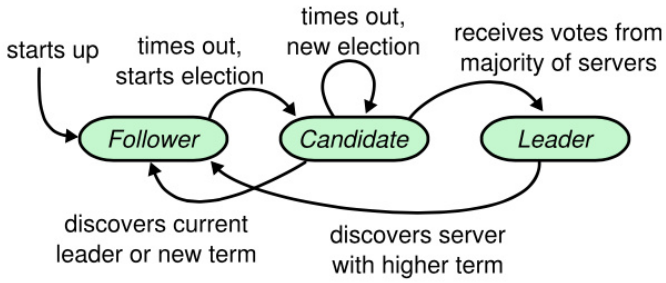
Raft

leader election

这一部分主要用于实现Raft的leader election。在Raft中，节点有三种状态，分别是Follower、Candidate和Leader。在通常情况下，系统中只有一个Leader并且其它的节点都是Follower。Follower不会发送任何请求，只是简单地响应来自Leader或者Candidate的请求。下图展现了这些状态和他们之间的转换关系：

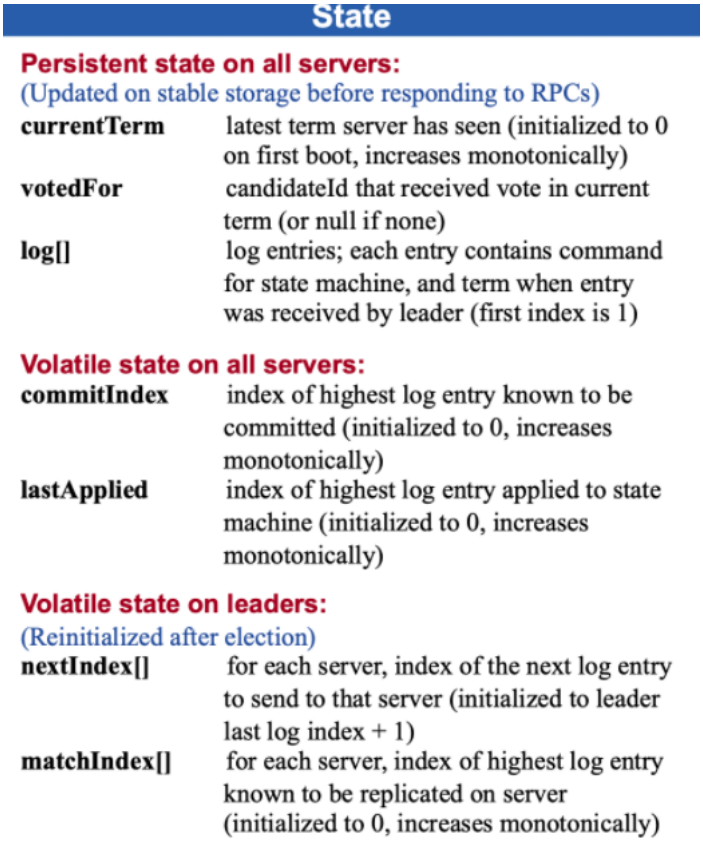


当程序启动时，所有节点都是Follewer状态，一个服务器节点会一直保持Follower状态只要他从Leader（AppendEntriesRPC）或Candidate（RequestVoteRPC）接收到有效的RPC。如果Follower在自己的electionTimeout内未能接收到这两种RPC，它就会认为系统中没有可用Leader，就会转变为Candidate发起选举。

具体实现

Raft structure

described in figure 2：



implemented in 6.824 lab:

```
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()

  // Your data here (2A, 2B, 2C).
  // Look at the paper's Figure 2 for a description of what
  // state a Raft server must maintain.
  currentTerm int
  votedFor    int               // 当前任期内收到选票的CandidateID，如果没有投给任何Candidate，则为None
  log         []LogEntry

  commitIndex int               // 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）
  lastApplied int               // 已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）

  nextIndex []int               // 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（初始值为领导者
                                // 最后的日志条目的索引+1）
  matchIndex []int              // 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（初始值为0，
                                // 单调递增

  state     int                 // 节点目前的状态：Follower·Candidate或Leader
  leader    int                 // 节点认为的当前系统中的Leader
  voteCount int                 // 节点所获得的选票数

  appendEntriesCh chan bool     // 如果节点接收到Leader的AppendEntriesRPC，则该信道置为true
  voteGrantedCh   chan bool     // 如果节点接收到Candidate的RequestVoteRPC，并给该Candidate投票，则该信道置为true
  leaderCh        chan bool     // 如果节点成功当选为Leader，则该信道置为true
  commitCh        chan bool     // 对于Leader来说，如果大多数节点成功复制日志，则该信道置为true
                                // 对于Follower来说，如果leader.CommitIndex大于自己的CommitIndex，则更新自己的CommitIndex，置该信道为true
  applyMsgCh      chan ApplyMsg // 将已提交的Log应用到state machine中，以方便客户端读取

  heartbeatTimeout time.Duration// 用于Leader定期向Follower发送AppendEntriesRPC
  electionTimeout  time.Duration// 用于Follower或Candidate在超时时重新发起选举
```

Implement detail

described in Figure 2

RequestVote RPC

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)

在节点启动时，首先会通过Make函数对节点进行初始化：

```
rf := &Raft{}
rf.peers = peers
rf.persister = persister
rf.me = me

// Your initialization code here (2A, 2B, 2C).
rf.currentTerm = STARTTERM
rf.votedFor = VOTENULL
rf.log = make([]LogEntry, 0)
rf.log = append(rf.log, LogEntry{LogIndex: 0, LogTerm: 0})

rf.nextIndex = make([]int, len(rf.peers))
rf.matchIndex = make([]int, len(rf.peers))

rf.state = Follower

rf.appendEntriesCh = make(chan bool, 1)
rf.voteGrantedCh = make(chan bool, 1)
rf.leaderCh = make(chan bool, 1)
rf.commitCh = make(chan bool, 1)
rf.applyMsgCh = applyCh

rf.heartbeatTimeout = time.Duration(HEARTBEATTIMEOUT) * time.Millisecond

// initialize from state persisted before a crash
rf.mu.Lock()
rf.readPersist(persister.ReadRaftState())
rf.mu.Unlock()
```

之后会在Make函数中实现Follower、Candidate和Leader超时和相应操作：

1. ElectionTimeout和heartbeatTimeout的时间段选取是非常重要的。
   - ElectionTimeout论文中建议大于300ms而且要随机设定
   - hearthbeatTimeout本Lab中要求1s内只能最多发送10次heartBeat，因此设置为100ms
2. 对于Follower来说，在ElectionTimeout时间内，信道appendEntriesCh或voteGrantedCh未被置为true，即未收到AppendEntriesRPC和RequestVoteRPC，则转变为Candidate准备发起选举。

```
func (rf *Raft) convertToCandidate() {
    // 转变state为Candidate，votedFor为自己，VoteCount为1
    rf.mu.Lock()
    defer rf.mu.Unlock()
    rf.state = Candidate
    rf.currentTerm += 1
    rf.votedFor = rf.me
    rf.leader = rf.me
    rf.voteCount = 1
    rf.persist()
}
```

3. 对于Candidate来说，首先开启leaderElection函数线程发起选举，并在ElectionTimeout时间内：

- 如果appendEntriesCh为true，那就说明系统中已经有了Leader，那么就应该放弃本次选举，转换状态为Follower。
- 如果voteGrantedCh为true，那说明系统中有了更高term的Candidate，自己也已经投票给了该server，那么就放弃本次选举，转换状态为Follower。
- 如果leaderCh为true，那说明自己已经获取到了系统中大多数server的选票，成为了当前Term的Leader。

否则，如果ElectionTimeout超时，则再次增加自己的Term，发起选举。

4. 对于Leader来说，需要定时（heartbeatTimeout）向系统中的所有server发送AppendEntriesRPC来表明自己的权威。

```
go func() {
   for !rf.killed() {
      rf.mu.Lock()
      state := rf.state
      electionTimeout := HEARTBEATTIMEOUT*2 + rand.Intn(HEARTBEATTIMEOUT)
      rf.electionTimeout = time.Duration(electionTimeout) * time.Millisecond
      rf.mu.Unlock()

      switch state {
      case Follower:
         select {
         //If follower not received appendEntries Request or vote request, then convert to Candidate
         case <-rf.appendEntriesCh:
         case <-rf.voteGrantedCh:
         case <-time.After(rf.electionTimeout):
            rf.convertToCandidate()
         }
      case Candidate:
         go rf.leaderElection()
         select {
         case <-rf.appendEntriesCh:
         case <-rf.voteGrantedCh:
         case <-rf.leaderCh:
         case <-time.After(rf.electionTimeout):
            rf.convertToCandidate()
         }
      case Leader:
         go rf.appendEntries()
         time.Sleep(rf.heartbeatTimeout)
      }
   }
}()
```

在server成为Candidate后，需要立即向系统中的其它server发送RequestVoteRPC来获取选票，leaderElection函数实现如下：

1. 首先应准备RequestVoteRPC的参数。

```
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    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
}
```

注：在go中，如果结构体或函数需要被外部访问，那么首字母需要大写。

2. 之后向系统中除了自己以外的server发送投票请求，并对Reply进行处理。

```
type RequestVoteReply struct {
    // Your data here (2A).
    Term      int  // currentTerm, for candidate to update itself
    VoteGrand bool // true means candidate received vote
}
```

3. 在对Reply参数进行处理之前，如果自己已经不是Candidate或者自己当前term已经发生改变，那么应该立即返回，不对reply参数进行处理。
4. 对reply参数进行处理：

- 若reply.VoteGrand为false且reply.Term>rf.currentTerm，说明系统中已有更高的Term，自己本次选举失败，转换为follower。
- 若reply.VoteGrand为true，则说明server投票给了自己，那么将自己的voteCount加1，并判断voteCount是否大于系统中节点的半数，如果是，那么转换为Leader，选举成功。

```
func (rf *Raft) leaderElection() {
   rf.mu.Lock()
   args := RequestVoteArgs{
      Term:         rf.currentTerm,
      CandidateId:  rf.me,
      LastLogIndex: rf.getLastIndex(),
      LastLogTerm:  rf.getLastTerm(),
   }
   rf.mu.Unlock()

   winThreshold := len(rf.peers)/2 + 1

   for i := 0; i < len(rf.peers); i++ {
      if i != rf.me{
         rf.mu.Lock()
         if rf.state!=Candidate{
            rf.mu.Unlock()
            return
         }
         rf.mu.Unlock()
         go func(server int, voteArgs RequestVoteArgs) {
            reply := RequestVoteReply{}
            ok := rf.sendRequestVote(server, &voteArgs, &reply)
            if !ok {
               return
            }

            rf.mu.Lock()
            defer rf.mu.Unlock()
            if rf.currentTerm != args.Term || rf.state!=Candidate {
               return
            }
            if reply.VoteGrand == false {
               if rf.currentTerm < reply.Term {
                  rf.convertToFollower(reply.Term)
               }
            } else {
               rf.voteCount += 1
               if rf.voteCount >= winThreshold {
                  rf.convertToLeader()
                  rf.leaderCh <- true
               }
            }
         }(i, args)
      }
   }
}
```

以上是Candidate发起RequestVote的处理，下面是Follower在收到RequestVote时的处理

1. 如果收到请求的Term<rf.currentTerm，说明该请求已经过时，那么将reply.Term置为rf.currentTerm（以用于请求者更新自己的term），将reply.VoteGrand置为false。
2. 如果收到请求的Term>rf.currentTerm，那么就要更新自己Term为args.Term，并转换自己的状态为Candidate。
3. 之后需要判断Candita的Log是否more up-to-date，在Raft中，判断Log新旧的标准是
   - If the logs have last entries with different terms, then the log with the later term is more up-to-date.
   - If the logs have last entries with same term, then the log with the longer index is more up-to-date.
4. 如果Candidate的Log和自己的一样，或是更新的。那么如果自己还没投票给其它Candidate，或者自己已经投票该Candidate，那么就将reply.VoteGrand置为true，将自己的状态置为Follower，将votedFor参数置为args.CandidateID。并将voteGrantedCh信道置为true。

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
   defer rf.persist()
   voteGrand := false
   uptoDate := false

   // 如果term<currentTerm term返回false
```

```
    rf.mu.Lock()
    if args.Term < rf.currentTerm {
       reply.Term = rf.currentTerm
       reply.VoteGrand = voteGrand
       rf.mu.Unlock()
       return
    }
    rf.mu.Unlock()

    // 如果term>currentTerm，那么将term设置为自己的currentTerm，变为follower
    // 如果该节点是candidate，那么重置votefor为VOTENULL，变为follower
    rf.mu.Lock()
    if args.Term > rf.currentTerm {
       rf.convertToFollower(args.Term)
    }
    rf.mu.Unlock()

    // If voteFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote
    // Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs.
    // If the logs have last entries with different terms, then the log with the later term is more up-to-date.
    // If the logs have last entries with same term, then the log with the longer index is more up-to-date.
    rf.mu.Lock()
    if args.LastLogTerm > rf.getLastTerm() ||
       (args.LastLogTerm == rf.getLastTerm() && args.LastLogIndex >= rf.getLastIndex()) {
       uptoDate = true
    }

    if (rf.votedFor == VOTENULL || rf.votedFor == args.CandidateId) && uptoDate {
       voteGrand = true
       rf.votedFor = args.CandidateId
       rf.state = Follower
       rf.leader = args.CandidateId
       rf.voteGrantedCh <- true
    }
    reply.Term = rf.currentTerm
    reply.VoteGrand = voteGrand
    rf.mu.Unlock()
}
```

log replication

一旦一个leader被选举出来，他就开始为客户端提供服务。客户端的每一个请求都包含一条被复制状态机执行的指令。leader把这条指令作为一条新的日志附加到Log中，然后并行的发起AppendEntriesRPCs给其它的server，让他们复制这条日志条目。当这条日志被安全的复制之后，leader会应用这条日志条目到它的状态机中然后把执行的结果返回给客户端。

Raft中每个日志条目（LogEntry）都包含一条**命令（command）**、**任期（Term）**，以及一个**正整数索引（LogIndex）**来标记其在日志中的位置。索引在不同任期内可以重复，但在同一任期内必须唯一。

日志条目在server中的状态通过两个变量控制：CommitIndex和LastApplied，通过这两个变量就可以让日志条目在以下三个状态中进行转换：

- **已写入**：日志条目 Append 到节点上，未进行其他操作，日志条目可以只存在内存中
- **已提交**：Leader 把日志条目复制到集群中大多数节点，Leader 通过更新 `CommitIndex` 将日志条目置为提交状态。
- **已应用**：Leader 通过 `AppendEntries` 请求将自己的 `CommitIndex` 通知给其他节点，其他节点进行提交操作，当大多数节点提交完成后，Leader 更新自己的 `LastApplied` 并将日志条目中的命令应用到状态机中。

具体实现

在该部分，需要另外实现AppendEntriesArgs结构体和AppendEntriesReply结构体

```go
type AppendEntriesArgs struct {
    Term         int        // 领导人的任期
    LeaderId     int        // 领导人的ID，以便于跟随重定向请求
    PrevLogIndex int        // 新的日志条目紧随之前的索引值
    PrevLogTerm  int        // PrevLogIndex条目的任期号
    Entries      []LogEntry // 准备存储的日志条目（表示心跳时为空；一次性发送多个是为了提高效率）
    LeaderCommit int        // 领导人已经提交的日志索引值
}

//
// AppendEntries RPC reply arguments structure
// field names must start with capital letters
//
type AppendEntriesReply struct {
    Term          int  // 当前任期，对于领导者而言，它会更新自己的任期
    ConflictIndex int  // 用于加速日志匹配
    ConflictTerm  int
    Success       bool // 如果follower所含有的条目与PreLogIndex和PreLogTerm匹配上了，就为真
}
```

需要实现Start函数，该函数是用于leader接收客户端的请求，并将请求加入到自己的log中。

```go
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    index := -1
    term := -1
    isLeader := true
    // Your code here (2B).
    term, isLeader = rf.GetState()

    rf.mu.Lock()
    if isLeader {
        index = rf.getLastIndex() + 1
        entry := LogEntry{
            LogIndex:   index,
            LogTerm:    term,
            LogCommand: command,
        }

        rf.log = append(rf.log, entry)
        rf.persist()
    }
    rf.mu.Unlock()
    return index, term, isLeader
}
```

leader为每个Follower都记录一个变量nextIndex，表示下一个需要发送个Follower的日志索引。当节点转变为Leader时，每个Follower的nextIndex均初始化为leader当前log最后一条日志条目的索引+1

```go
func (rf *Raft) convertToLeader() {
    rf.state = Leader
    for i := 0; i < len(rf.peers); i++ {
        rf.nextIndex[i] = rf.getLastIndex() + 1
    }
}
```

每个server都需要判断自己的commitCh信道是否为true，如果为true，说明日志已被提交，需要将最新提交的日志应用到state machine中，之后增加自己的lastApplied值为commitIndex：

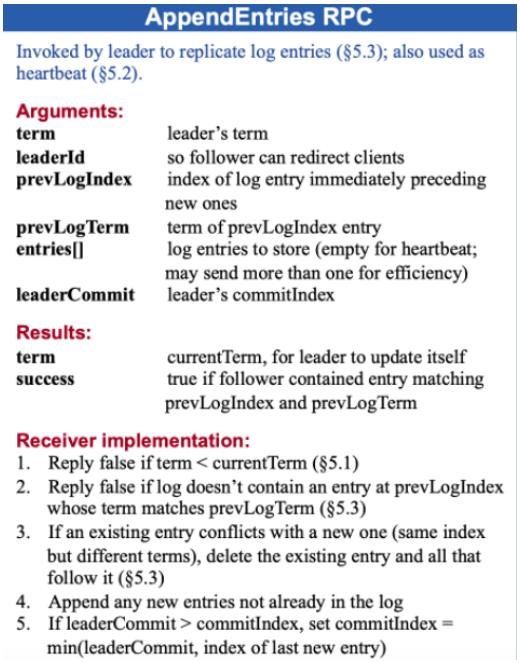
- 对于Leader来说，如果日志已经被复制到大多数节点中，则提交这些日志并增加自己的commitIndex，置自己的commitCh信道为true。
- 对于Follower来说，Leader会在AppendEntriesRPC中发送自己的commitIndex，follower需要判断，如果自己和leader的日志匹配，且commitIndex小于leader的commitIndex，则提交这些日志并根据规则增加自己的commitIndex，置自己的

commitCh信道为true

```
go func() {
    for !rf.killed() {
        select {
        case <-rf.commitCh:
            rf.mu.Lock()
            lastApplied := rf.lastApplied
            commitIndex := rf.commitIndex
            for i := lastApplied + 1; i <= commitIndex; i++ {
                msg := ApplyMsg{
                    CommandValid: true,
                    Command:      rf.log[i].LogCommand,
                    CommandIndex: rf.log[i].LogIndex,
                }
                //fmt.Println("server", rf.me, "apply commandIn
                applyCh <- msg
                rf.lastApplied = i
            }
            rf.mu.Unlock()
        }
    }
}()
```

Implement detail

described in Figure 2



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	leader's commitIndex

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)

实现appendEntries函数用于Leader向其它server发送AppendEntriesRPC：

1. leader在发送AppednEntriesRPC之前首先更新自己的commitIndex，就像代码中的注释一样，通过判断Leader为各个server维护的matchIndex值来更新自己commitIndex，同时判断log[N].term==currentTerm是为了避免论文中figure 8出现的现象。
2. 之后向每一个除了自己的server发送AppendEntriesRPC，并对reply进行处理。
3. 在对reply进行处理之前，也同样需要判断leader的状态是否有效，如果无效直接返回。
4. 对reply进行处理：
    - 如果reply.Term>rf.currentTerm，说明系统中已有更高的任期，leader失效，需要转换状态为follower

- 如果follower与leader的日志不匹配，leader需要根据reply.conflictIndex和reply.conflictTerm来更新该follower的nextIndex值。leader首先从日志尾部向前寻找reply.conflictTerm最后出现的位置findIndex。如果找到，则置该server的nextIndex为findIndex+1，否则，置为reply.conflictIndex。
- 如果日志匹配，则更新该follower的nextIndex值和matchIndex值。正确设置matchIndex值的方式应该时prevLogIndex+len(entries[])，然后设置nextIndex为matchIndex+1。

```
func (rf *Raft) appendEntries() {
   // If there exists an N such that N>commitIndex, a majority of matchIndex[i]>=N, and log[N].term==currentTerm:
   // Set commitIndex=N
   rf.mu.Lock()
   lastIndex := rf.getLastIndex()
   N := rf.commitIndex
   for i := rf.commitIndex + 1; i <= lastIndex; i++ {
      commitNum := 1
      for j := range rf.peers {
         // rf.log[i].LogTerm==rf.currentTerm
         // 这一点用于防止论文中figure 8出现的现象，新leader不会提交之前term的日志，而是提交当前term的日志。
         // 根据“日志一致性原则”，之前term的日志就也会被提交和应用。
         if j != rf.me && rf.matchIndex[j] >= i && rf.log[i].LogTerm == rf.currentTerm {
            commitNum++
         }
      }
      if commitNum >= (len(rf.peers)/2 + 1) {
         N = i
      }
   }
   if N > rf.commitIndex {
      rf.commitIndex = N
      rf.commitCh <- true
   }
   rf.mu.Unlock()

   for i := 0; i < len(rf.peers); i++ {
      if i != rf.me{
         rf.mu.Lock()
         if rf.state!=Leader{
            rf.mu.Unlock()
            return
         }
         rf.mu.Unlock()
         go func(server int) {
            rf.mu.Lock()
            prevLogIndex := rf.nextIndex[server] - 1
            prevLogTerm := rf.log[prevLogIndex].LogTerm
            entries := rf.log[prevLogIndex+1:]
            leaderCommit := rf.commitIndex
            args := AppendEntriesArgs{
               Term:         rf.currentTerm,
               LeaderId:     rf.me,
               PrevLogIndex: prevLogIndex,
               PrevLogTerm:  prevLogTerm,
               Entries:      entries,
               LeaderCommit: leaderCommit,
            }
            rf.mu.Unlock()

            reply := AppendEntriesReply{}
            ok := rf.sendAppendEntriesRequest(server, &args, &reply)

            if !ok {
               return
            }

            rf.mu.Lock()
            defer rf.mu.Unlock()
            if rf.currentTerm != args.Term || rf.state!=Leader {
               return
            }

            if reply.Success == false { // fail to replica log
               if reply.Term > rf.currentTerm {
                  rf.convertToFollower(reply.Term)
                  return
               } else {
                  // upon receiving a conflict response,
                  // the leader should first search its log for conflictTerm
                  findIndex := -1
                  for i := len(rf.log) - 1; i >= 0; i-- {
                     if rf.log[i].LogTerm == reply.ConflictTerm {
                        findIndex = rf.log[i].LogIndex
                        break
                     }
```

```
                }

                if findIndex == -1 {
                   // if it doesn't find an entry with that term, it should set nextIndex=conflictIndex
                   rf.nextIndex[server] = reply.ConflictIndex
                } else {
                   // if it finds, it should set nextIndex to be findIndex+1
                   rf.nextIndex[server] = findIndex + 1
                }
             }
          } else {
             // 如果成功：更新相应follower的nextIndex和matchIndex
             rf.matchIndex[server] = args.PrevLogIndex + len(args.Entries)
             rf.nextIndex[server] = rf.matchIndex[server] + 1
          }
       }(i)
    }
  }
}
```

Follower在收到AppendEntriesRPC后的处理流程：

1. 如果args.Term<rf.currentTerm，说明leader已经过时，那么设置reply.Term=rf.currentTerm，立即返回false。
2. 如果args.Term>rf.currentTerm，说明server的term已经过时，需要更新到当前leader的term，并转换状态为Follower。
3. 如果server日志中没有preLogIndex，那么设置reply.ConflictIndex=len(rf.log)，reply.ConflictTerm=-1。
4. 如果server日中中有preLogIndex：
   - 如果server中prevLogIndex的日志与leader不匹配，那设置reply.ConflictTerm=rf.log[prevLogIndex].LogTerm。之后在rf的日志中寻找第一个出现reply.ConflictTerm的日志条目的logIndex，并设置reply.ConflictIndex为该logIndex。
   - 如果server中日志与leader匹配，我们就可以将entries附加到server的日志当中去了，但是要注意，我们不能够删除server中匹配的日志，只需要删除不匹配的日志后再将server中缺少的日志附加到log中即可。
5. 无论日志是否匹配，都要将server的appendEntriesCh信道设置为true。

```
func (rf *Raft) AppendEntriesRequest(args *AppendEntriesArgs, reply *AppendEntriesReply) {
  defer rf.persist()
  success := false
  rf.mu.Lock()
  // Leader的term小于follower的term，说明leader已经过时
  if args.Term < rf.currentTerm {
    reply.Term = rf.currentTerm
    reply.Success = success
    rf.mu.Unlock()
    return
  }
  rf.mu.Unlock()

  // 如果leader的term大于该server的term，那么更新该server的term与leader一致
  rf.mu.Lock()
  if args.Term > rf.currentTerm {
    rf.convertToFollower(args.Term)
  }
  rf.mu.Unlock()

  // 日志匹配part
  rf.mu.Lock()
  //if follower doesn't have prevLogIndex in its log,
  //it should return with conflictIndex=len(log) and conflictTerm=None
  if args.PrevLogIndex > rf.getLastIndex() {
    reply.ConflictIndex = len(rf.log)
    reply.ConflictTerm = -1
  } else {
    // if a follower does have prevLogIndex in its log
    if args.PrevLogIndex == 0 || rf.log[args.PrevLogIndex].LogTerm == args.PrevLogTerm { //找到了日志匹配点，开始复制日志
      //fmt.Println("server",rf.me,"匹配成功，prevLogIndex is",args.PrevLogIndex,"PrevLogTerm is",args.PrevLogTerm,"日志长度是",len(rf.log
      success = true
      // 附加新的日志，在附加新的日志的过程中，应该避免删除server中已匹配的日志
      // logInsertIndex为rf.log中的插入位置，初始值为args.PrevLogIndex+1
      logInsertIndex:=args.PrevLogIndex+1
      // newEntriesIndex为args.Entries中需要插入到rf.log中的日志
      newEntriesIndex:=0
      for{
        if logInsertIndex>=len(rf.log)||newEntriesIndex>=len(args.Entries){
          break
        }
        if rf.log[logInsertIndex].LogTerm!=args.Entries[newEntriesIndex].LogTerm{
```

```
          break
        }
        logInsertIndex++
        newEntriesIndex++
      }
      if newEntriesIndex<len(args.Entries){
        rf.log=append(rf.log[:logInsertIndex],args.Entries[newEntriesIndex:]...)
      }

      if args.LeaderCommit > rf.commitIndex {
        //如果进入到此判断条件，说明leader已经知道大多数follower已经将新日志复制成功，并已将新日志应用到自己的state machine中返回给客户端
        //此时follower也可以更新自己的commitIndex，并将新日志应用到自己的state machine中
        rf.commitIndex = IntMin(args.LeaderCommit, rf.getLastIndex())
        rf.commitCh <- true
        //fmt.Println("server", rf.me, "update CommitIndex to", rf.commitIndex)
      }
    } else{
      // the term doesn't match, it should return conflictTerm=log[preLogIndex].Term,
      // and then search its log for the first index whose entry has term equal to conflictTerm
      if rf.log[args.PrevLogIndex].LogTerm != args.PrevLogTerm{
        reply.ConflictTerm = rf.log[args.PrevLogIndex].LogTerm
        for i := 0; i < len(rf.log); i++ {
          if rf.log[i].LogTerm == reply.ConflictTerm {
            reply.ConflictIndex = rf.log[i].LogIndex
            break
          }
        }
      }
    }
  }

  // 无论是否成功复制日志，都要把appendEntriesCh置为true表示收到了heartBeat
  rf.appendEntriesCh <- true
  rf.leader = args.LeaderId
  rf.state = Follower
  reply.Term = rf.currentTerm
  reply.Success = success
  rf.mu.Unlock()
}
```

持久化

论文中提到对于每个server来说，有三个持久化的state，分别是currentTerm、votedFor和log[]，因此在6.824的lab 3C中，我们要实现Persist()函数和readPersist()函数

```
func (rf *Raft) persist() {
   w := new(bytes.Buffer)
   e := labgob.NewEncoder(w)
   e.Encode(rf.currentTerm)
   e.Encode(rf.votedFor)
   e.Encode(rf.log)
   data := w.Bytes()
   rf.persister.SaveRaftState(data)
}

func (rf *Raft) readPersist(data []byte) {
  if data == nil || len(data) < 1 { // bootstrap without any state?
    return
  }
  r := bytes.NewBuffer(data)
  d := labgob.NewDecoder(r)
  var currentTerm int
  var votedFor int
  var log []LogEntry
  if d.Decode(&currentTerm) != nil ||
    d.Decode(&votedFor) != nil ||
    d.Decode(&log) != nil {
    fmt.Println("Decode fail")
  } else {
    rf.currentTerm = currentTerm
    rf.votedFor = votedFor
    rf.log = log
  }
}
```

如果基于Raft的服务器重新启动，则应从中断的位置恢复服务。这就要求Raft保持持久状态，使其在重启后仍然有效。

我们需要明确在什么时候需要持久化：

1. 成为Candidate/Follower时（currentTerm和votedFor变化）
2. follower投完票时（votedFor变化）
3. append Entries RPC更改log时(log 更改)
4. Start()追加leader日志时（log更改）