Lab2A

Raft机制：

Leader election：

Raft使用心跳机制触发领导人选举。Leader为了维护自己的权威，向所有追随者发送周期性心跳(附录不带日志项的RPC)。如果一个追随者在一段时间内没有收到任何通信，称为选举超时，那么它假设没有可行的领导者，并开始选举以选择新的领导者。

为了开始选举，一个follower增加其当前的任期并过渡到candidate。然后它自己投票，并在集群中的每个其他服务器上并行地发出RequestVote RPC。candidate在这种状态下持续，直到三件事之一发生：

(a)它赢得选举。如果candidate在同一任期内从整个集群中的大多数服务器获得选票，则该candidate赢得选举。每个服务器在给定任期内最多投票给一个candidate，先到先得。多数票规则保证了至少有一名candidate能够赢得特定任期的选举。一旦candidate赢得选举，他就会成为领袖。然后，它向所有其他服务器发送心跳消息，以建立其权威并防止新的选举。

(b)另一个服务器建立自己作为领导者。在等待投票时，candidate可能会从另一个声称是leader的服务器收到一个AppendEntries RPC。如果leader的任期(包含在其RPC中)至少与candidate的当前任期一样大，那么candidate承认leader是合法的，并返回到follower状态。如果RPC中的项小于candidate当前的项，则该候选人拒绝RPC，继续处于候选状态。

(c)一段时间没有赢家。如果许多follower同时成为候选人，那么就可以分割选票，从而没有candidate获得多数。当这种情况发生时，每个candidate将超时并通过增加其任期和启动另一轮RequestVote RPCs来启动新的选举。然而，如果没有额外的措施，分裂投票可以无限期地重复。Raft使用随机的选举超时，以确保分裂投票是罕见的，并迅速解决。为了防止首先出现分裂投票，选举超时从固定间隔(如150 ~ 300ms)中随机选择。

设计细节：

结构体：

结构体这边基本按照的是论文中的格式设置的

```
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()

    votedFor int
    log       []int
    CurrentTerm int
    LastHeartBeatTime time.Time
    state     int // leader or follower or candidate
    //applyCh chan ApplyMsg

    commitIndex int
    lastApplied int


    nextIndex  []int
    matchIndex []int
    // Your data here (2A, 2B, 2C).
    // Look at the paper's Figure 2 for a description of what
    // state a Raft server must maintain.

}
```

Make

make函数主要是新建一个raft结构，所以主要的职责就是进行初始化。这里要注意的是所有的新建节点都是follower，当这些follower没有收到心跳而超时时，会自行去竞选而生成leader。所以在初始化之后要新开一个go协程来做循环操作，即不停的查询是否有收到心跳。那此处的心跳超时实际上是通过记录上一次心跳时间来判断的，如果发现当前时间减去上一次心跳时间超时了，就需要进行选举了。

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{}
    rf.peers = peers
    rf.persister = persister
    rf.me = me
    //rf.applyCh = applyCh
    fmt.Println("make raft ",me,rf.CurrentTerm)
    rf.votedFor = -1
    rf.log = []int{}
    rf.commitIndex = 0
    rf.lastApplied = 0
    rf.state = Follower
    rf.CurrentTerm = 0
    rf.LastHeartBeatTime = time.Now()
    go rf.doLoopjob()

    // Your initialization code here (2A, 2B, 2C).

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState())

    return rf
}
```

doloopjob

这个函数其实是把各个状态下的重复操作合在一起了。如果是leader，那他就需要不断的发送心跳报文。如果是follower或者是candidate，那他们就需要不断的进行超时查询，这里我把folloer和candidate合在一起考虑了，因为candidate如果在一段时间内没有变成leader或是降级为follower，也会重新设置随机的超时时间发起竞选。

```
func (rf *Raft) doLoopjob() {
    for{
        if rf.killed(){
            return
        }
        if rf.state == Leader{//sendHeartBeat

            args := AppendEntriesArgs{
                Term:     rf.CurrentTerm,
                Leaderid: rf.me,
            }
            time.Sleep(time.Millisecond * 100)
            for i:=0;i<len(rf.peers);i++{
                if i==rf.me{
                    continue
                }
                go func(p int) {
                    reply := AppendEntriesReply{}
                    ok := rf.sendAppendEntries(p, &args, &reply)
                    if !ok {
                        return
                    }
                    rf.mu.Lock()
                    defer rf.mu.Unlock()
                    if !reply.Success {
                        if reply.Term > rf.CurrentTerm {
                            rf.becomeFollower(reply.Term)
                        }
                    }
                    return
                }(i)
            }
        }else {
            during := time.Since(rf.LastHeartBeatTime)
            if during< time.Duration(randTimeout(700,900)) * time.Millisecond{
                continue
            }else{
                //fmt.Println(during,time.Duration(randTimeout(700,900)) *
time.Millisecond)
                go rf.Handleelection()
            }
        }
    }
    // Your code here, if desired.
}
```

可以看到，如果不是leader的话，就会设置一个区间内的随机时间段，如果超时了，就进入选举环节，即Handleelection。

如果是leader的话，重复工作就是发送心跳。也就是每隔一段时间（100ms）发送AppendEntries。同样是要遍历其他所有的节点。因为2A实验部分只包含选举，因此只需要发送当前的term以及自己作为leader的ID就可以。

Handleelection

```
func (rf *Raft) Handleelection() {

    rf.mu.Lock()
    defer rf.mu.Unlock()
    granted := 1
    reply := RequestVoteReply{}
    request := RequestVoteArgs{}

    rf.becomeCandidate()
    rf.LastHeartBeatTime = time.Now()
    request.Term = rf.CurrentTerm
    request.CandidateId = rf.me
    //request.LastLogIndex =
    //request.LastLogTerm =
    //fmt.Println(rf.me,"start election")
    for i:=0;i<len(rf.peers);i++{
        if i==rf.me{
            continue
        }
        go func(p int){
            ok := rf.sendRequestVote(p,&request,&reply)
            if ok {
                rf.mu.Lock()
                defer rf.mu.Unlock()
                if !reply.VoteGranted{
                    if reply.Term>rf.CurrentTerm{
                        rf.becomeFollower(reply.Term)
                    }
                    return
                }
                granted++
                if granted > len(rf.peers) / 2 {
                    rf.becomeLeader()
                    go rf.doLoopjob()
                    return
                }
            }
        }(i)
        //fmt.Println(rf.me,granted)
    }
    // Your code here, if desired.
}
```

这部分函数的主要工作就是发起竞选。首先要做的就是将自己的状态转变为candidate，同时需要把自己的term任期进行加一。然后对所有其他节点发送RequestVote请求，来请求他们的投票。如果在此期间，发现存在回复中的term比自己大，那很可能说明已经存在了新的leader，此时就把自己重新设置为follower，继续等待心跳。如果得到的票数超过一半，那就说明竞选成功，自己转换成leader，同时以leader的身份重新进入doLoopjob。

RequestVote

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {

    rf.mu.Lock()
    defer rf.mu.Unlock()
    rf.LastHeartBeatTime = time.Now()
    //fmt.Printf("CurrentTerm:%d args:%d\n",rf.CurrentTerm,args.Term)
    if args.Term < rf.CurrentTerm{
        reply.VoteGranted = false
        reply.Term = rf.CurrentTerm
    }

    if args.Term >=rf.CurrentTerm{
        rf.becomeFollower(args.Term)
    }
    //fmt.Println("candidate: ",args.CandidateId,args.Term,"me:
",rf.me,rf.CurrentTerm,rf.votedFor)
    //??
    if rf.votedFor == -1 || rf.votedFor == args.CandidateId {
        rf.votedFor = args.CandidateId
        reply.VoteGranted = true
    }


    // Your code here (2A, 2B).
}
```

这部分是指当raft节点接受到请求投票时需要怎么做。首先就是判断请求体中的term是否大于自身的term，如果大于等于则说明接收到的请求来自于更早发出竞选的节点，因此无论是follower还是candidate都需要更新自己的状态。然后如果查看自己是否已经投过票，如果投过，是否投给的就是请求的ID，因为可能存在多个candidate的情况，但是每个节点只有一票。

遇到的问题

1、就是锁的问题，之前没有太在意锁的使用，导致代码写完之后一直无法正常运行，总是处于一种死循环的状态，最后发现是由于在Handleelection函数中已经加了锁，但是在becomefollower的函数中仍然加了锁，这就导致无法进入becomefollower，因为需要Handle election结束之后才会释放锁。

2、就是votedfor的属性。之前test2A的第二部分总是过不了，也就是没有办法进行第二次投票。最后发现是因为没有重置votedfor的缘故，因为不重置的话，节点始终保存着投票给第一次当选leader的ID。当这个leader崩溃时，其他节点无法得到选票。所以在每次接收到心跳时，都重置vote为初始值，表示先前的投票过程已经结束了，因为已经存在新的leader节点了。

```
lyj@ubuntu:~/Desktop/6.824/src/raft$ go test -run 2A
make raft  0 0
make raft  1 0
make raft  2 0
Test (2A): initial election ...
labgob warning: Decoding into a non-default variable/field VoteGranted may not work
  ... Passed --   3.6  3  170   34584    0
make raft  0 0
make raft  1 0
make raft  2 0
Test (2A): election after network failure ...
  ... Passed --   6.9  3 19961 3109252    0
PASS
ok      _/home/lyj/Desktop/6.824/src/raft       10.450s
lyj@ubuntu:~/Desktop/6.824/src/raft$
```