目　录

1 Part 2A: leader election

1.1 任务思路梳理

实现 Raft 领导者选举和心跳（没有日志条目的 ‘AppendEntries‘RPC）。暂时不用写日志的部分。Raft 的运行图如图 1-1 所示。

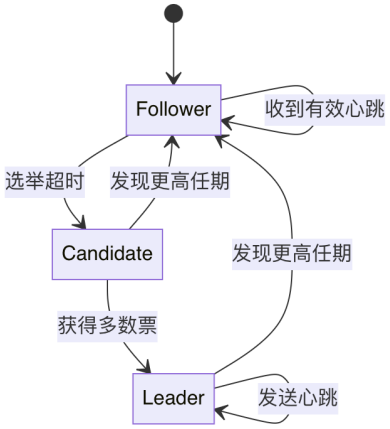


图 1-1 Raft 执行原理

在描述代码细节之前，我先梳理好了所有函数和函数之间的调用逻辑，如图 1-2 所示。

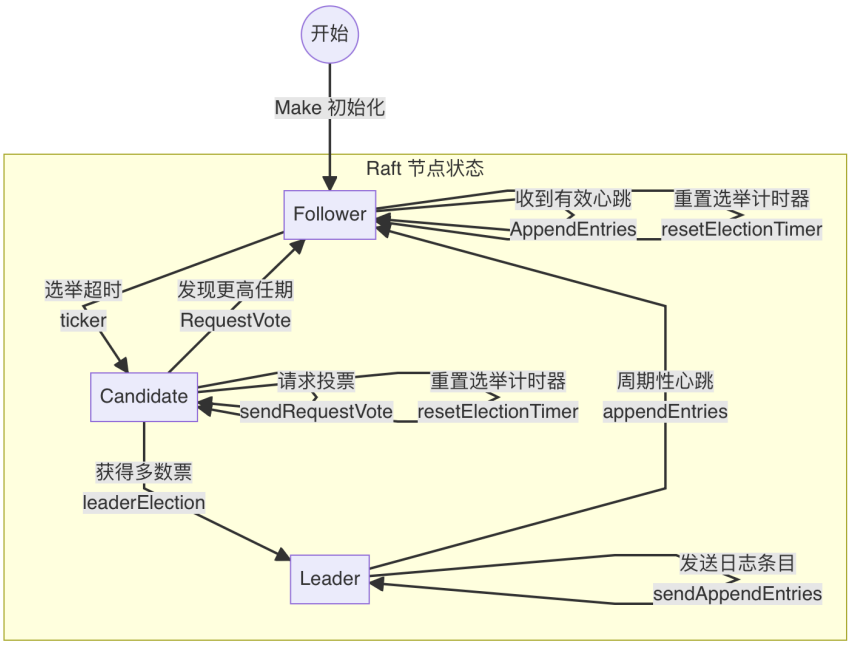


图 1-2 接口之间的调用逻辑

1.2 任务分析、实现流程和代码实现

我在实验过程中，是按照 mit6.824 给的实验手册一步步完成的。下面是按照步骤完成的代码。

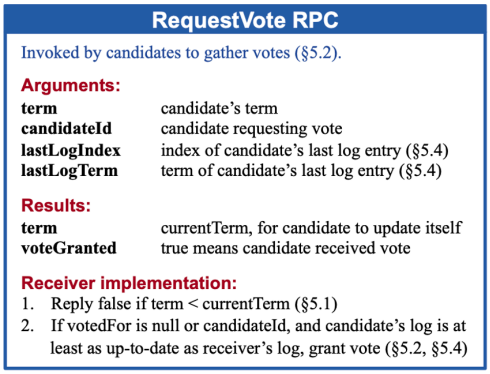
1.2.1 运行方式

不要直接运行 Raft 实现，而是通过测试器运行，使用 ‘go test -run 2A‘

现在运行出来肯定是 fail 的。因为还没实现。下面我们将一步步按照 MIT6.824 的手册一步步实现。

1.2.2 ReqeustVote RPC 定义

遵循论文 Figure2：关注发送和接收 ‘RequestVote‘ RPC（远程过程调用），涉及选举的服务器规则，以及与领导选举相关的状态。如图 1-3 所示。



Args字段：

- **term：候选人的任期**
- **cadidatedId：想要获取选票的人的id**
- **lastLogIndex：日志索引**
- **lastLogTerm：日志的任期**

Result字段：

- **term：当前的任期，用于候选人给自己setNewTerm的**
- **voteGranted：是否同意把票投给你**

图 1-3 RequestVoteRPC 解读

所以按照论文 Figure2 右上角去定义 ‘RequestVote RPC‘，按照论文去定义就行了。

代码如下所示。

language

```
// example RequestVote RPC arguments structure.
// field  names must start with capital letters!
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    Term         int     // 候选人当前的任期
    CandidateId int     // 候选人的id，你向别人去申请票，要告诉别人自己的id
}

// example RequestVote RPC reply structure.
// field  names must start with capital letters!
type RequestVoteReply struct {
```

```
    // Your data here (2A).
    Term        int     // 现在的任期，给候选人去更新自己的
    VoteGranted bool    // 如果是true代表候选人申请到选票了
}
```

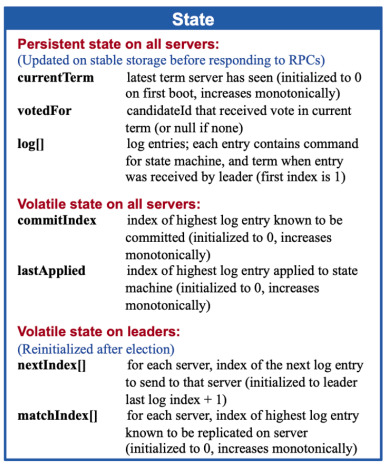
1.2.3 完善 Raft 结构体

在 Raft 结构中添加状态：在 ‘raft.go‘ 的 Raft 结构中加入图 2 所描述的领导选举状态。您还需要定义一个结构来保存每个日志条目的信息。

在这里，其实 2A 只需要这里添加保存每个日志条目信息的字段就行，但是我直接看论文的 Figure2，然后结合网上查阅的资料，直接把所有的字段添加好了。

然后在实现的时候，用一个枚举，然后里面是三种状态（Follower，候选人和领导），这个就是论文里面提到的。

在图中我会详细解释我对与 Figure2 左上角这个 Raft 结构题定义的理解。如图 1-4 所示。



State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	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	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

- state：就是当前节点的状态，就是刚才提到的枚举
- appendEntryCh：定义的是一个管道，然后这个管道是用来追加日志条目的
- heartBeat：定义了一个发送心跳包的频率，用time包里面的duration
- electionTime：下一次选举的计时器，用于触发新的领导者选举

- currentTerm：就是当前节点的任期，这个很好理解
- voteFor：表示在当前任期中，我的票是投给谁了？（一开始设置成-1，然后就能防止在同一任期里面给多个人投票）
- Log：就是存日志条目的日志结构，具体的一开始已经写好了，不用我来写
- lastApplied：已被应用到状态机的最高日志条目索引，可以确保状态机的更新是按照日志的顺序进行的（因为运行过程中是不阻塞的）
- nextIndex：对于每个同伴节点，要发送的下一个日志条目的索引。
- matchIndex：对于每个同伴节点，已知的已复制到该同伴节点的最高日志条目的索引。
- applyCh：我定义的用来发送applyMsg的通道
- applyCond：线程同步，条件变量来的

图 1-4 RequestVoteRPC 解读

Raft 结构代码如下所示。

language

```
type RaftState string


const (
    Follower  RaftState = "Follower"
    Candidate           = "Candidate"
    Leader              = "Leader"
)


// A Go object implementing a single Raft peer.
```

```
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()

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

    state         RaftState
    heartBeat     time.Duration
    electionTime  time.Time

    applyCh   chan ApplyMsg
    applyCond *sync.Cond
}
```

在图 1-4 中，论文 Figure2 里面部分字段，是为了 2B 以及以后的任务使用的，因此在实现 2A 的时候我在 ‘Raft‘里面只加入了和 2A 有关的字段。

1.2.4 ReqeustVoteRPC 和 Make() 等接口的完善

填充 RPC 结构体：填充 ‘RequestVoteArgs‘ 和 ‘RequestVoteReply‘ 结构体。修改 ‘Make()‘ 函数，创建一个后台协程，定期触发领导选举，通过发送 ‘RequestVote‘ RPC，在一段时间内没有从其他节点收到消息时发起选举。这样，节点可以了解是否已有领导者，或者自己成为领导者。

填充 ‘RequestVoteArgs‘结构体和 ‘RequestVoteReply‘结构体的工作上面做过了（同样，也是只添加了 2A 的内容）。然后下面是 ‘Make()‘函数的编写。

Make() 主要需要做的（2A 阶段）：

- **实例化** Raft **结构体**
- **填写** 2A **相关的字段，具体见下面代码所示。**
- **启动一个协程（计时器-用来管理选举和心跳的发送）**

具体细节见如下代码所示：

language

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    // 这里首先是创建一个Raft节点
```

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

    // Your initialization code here (2A, 2B, 2C).
    // 然后后面这里就是继续初始化节点里面的字段了
    rf.state = Follower                    // 一开始设置身份就是Follower
    rf.currentTerm = 0                     // 任期是0
    rf.votedFor = -1                       // 前面提到过了，先设置成-1，表示没有把票投给别人
    rf.heartBeat = 50 * time.Millisecond // 50ms的心跳周期
    rf.resetElectionTimer()                // 重置选举计时器，为发起新选举做准备

    rf.applyCh = applyCh                   // 送日志的通道
    rf.applyCond = sync.NewCond(&rf.mu) // 线程同步的条件变量
    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState()) // 如果是因为crash，重新启动的时候读取上一个状态
    // start ticker goroutine to start elections
    go rf.ticker()                         // 计时器，用于管理选举和心跳的发送
    // go rf.applier()                     // 弄日志的，在这里面就是把该弄的日志弄好，弄到状态机上
    return rf
}
```

首先 Make() 里面前面的部分都是在初始化一些字段，字段的含义我在代码块里面讲解了。

然后手册要求的：创建一个后台协程，定期触发领导选举，通过发送 RequestVote RPC，在一段时间内没有从其他节点收到消息时发起选举。

这一部分在 rf.ticker() 函数里面实现了。

ticker() 代码如下所示。

思路我感觉算比较好理解，就是如果是 Leader 就发送心跳，如果超时了，就进行新一轮选举。

language

```
func (rf *Raft) ticker() {
    for rf.killed() == false { // 先看看自己挂了没
        // Your code here to check if a leader election should
        // be started and to randomize sleeping time using
        // time.Sleep().
        time.Sleep(rf.heartBeat) // 然后进行心跳周期
        rf.mu.Lock()
        if rf.state == Leader {
```

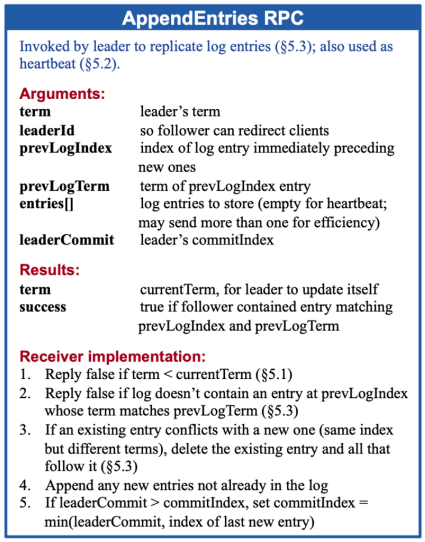
```
            rf.appendEntries(true) // 如果状态是Leader，就向其他人发送一波心跳包
        }
        if time.Now().After(rf.electionTime) {
    // 如果当前的时间戳超过了要进行新一轮选举是设定好的时间，就进行新一轮的选举
            rf.leaderElection()
        }
        rf.mu.Unlock()
    }
}
```

然后 appendEntries 和 leaderElection 都是要实现的。这也是手册指引的后面的步骤。

1.2.5 实现心跳机制

定义 'AppendEntries' RPC 结构体（可能还不需要所有参数），让领导者定期发送心跳。编写 'AppendEntries' RPC 处理方法，重置选举超时，以防其他服务器在已选出领导者时前来竞选。

如图 1-5 所示，rpc 结构和我的理解（这里面包括了 2A-2D 的内容，也包括优化的，但是代码实现暂时只是 2A 的）



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)

AppendEntriesArgs的字段

- Term：当前的任期号
- LeaderId：就是发送AppendEntries请求，也就是发送心跳包的领导的自己的id，告诉别人我是leader
- PrevLogIndex：新的日志条目紧接着前一个条目的索引
- PrevLogTerm：PrevLogIndex条目的任期号
- Entries：需要被复制到跟随者的日志条目数组
- LeaderCommit：领导者的commitIndex，告诉其他人领导者已经提交的日志条目的最高索引

AppendEntriesReply的字段

- Term：响应中返回的当前的任期号，可以给领导者更新自己的任期号
- Success：表示是否成功追加了日志条目/是否成功发送心跳
- Conflict：是否存在日志冲突
- XTerm：发生冲突的日志条目的任期号
- Xindex：日志中Xterm首次出现的索引
- XLen：跟随之日志的长度

图 1-5 AppendEntries 解释

定义代码如下所示。

language

```
type AppendEntriesArgs struct {
    Term        int
    LeaderId    int
```

```
    Entries      []Entry // 在2A中，以为append_entry只当发心跳用，所以Entry字段都是空
}
type AppendEntriesReply struct {
    Term    int
    Success bool
}
```

然后就要写 appendEntries 了。

具体的实现如下所示。

appendEntries 函数有两个作用:

- 作为领导者向其他节点发送心跳
- 用于复制日志条目到其他节点

所以其实是有两个作用的，不过是合在一个函数里面实现了，用参数)heartbeat bool) 来表示是不是心跳就行了。

然后前面 ticker 里面，是传了 True，因为那里是需要领导者发送心跳给其他人的。

代码如下所示。

language

```
func (rf *Raft) appendEntries(heartbeat bool) {
    for peer, _ := range rf.peers {
        if peer == rf.me {
            rf.resetElectionTimer() // 重新设置选举时间，防止别人进行无谓的玄机
            continue
        }
        // rules for leader 3
        if heartbeat { // 2A这里就只有heartbeat
      // 这里就是心跳包字段，告诉别人我是leader
            args := AppendEntriesArgs{
                Term:         rf.currentTerm,
                LeaderId:     rf.me,
            }
            go rf.leaderSendEntries(peer, &args)
        }
    }
}
```

对于 go rf.leaderSendEntries(peer, args) 是干嘛的。

在 appendEntries 函数中，对于每个同伴节点（peer），函数通过 go rf.leaderSendEntries(peer, args) 启动了一个新的协程。这里，leaderSendEntries 是实际发送 AppendEntriesRPC 请求的函数。

leaderSendEntries() 函数

这个函数的作用其实就是通过网络调用 AppendEntries 而已，只不过是这里走了很多层。

具体见代码中注释所示

language

```
func (rf *Raft) leaderSendEntries(serverId int, args *AppendEntriesArgs) {
    var reply AppendEntriesReply
    // 向指定的服务器发送 AppendEntries RPC。
    ok := rf.sendAppendEntries(serverId, args, &reply)
    if !ok {
        // 如果 RPC 调用失败，直接返回。
        return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // 检查响应中的任期号。如果大于当前任期号，更新为更高的任期号。
    if reply.Term > rf.currentTerm {
        rf.setNewTerm(reply.Term)
        return
    }
}
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // 初始化回复为失败，设置当前任期号。
    reply.Success = false
    reply.Term = rf.currentTerm
    // 如果 RPC 的任期号大于当前任期号，则更新为更高的任期号并返回。
    if args.Term > rf.currentTerm {
        rf.setNewTerm(args.Term)
        return
    }
    // 如果 RPC 的任期号小于当前任期号，直接返回。
    if args.Term < rf.currentTerm {
        return
    }
    // 收到有效的心跳，重置选举计时器。
    rf.resetElectionTimer()
    // 如果当前服务器是候选人，则转变为追随者。
    if rf.state == Candidate {
```

```
        rf.state = Follower
    }
    // 如果到达这里，意味着 AppendEntries RPC 成功，设置回复为成功。
    reply.Success = true
}
func (rf *Raft) sendAppendEntries(server int, args *AppendEntriesArgs, reply *AppendEntriesReply) bool {
    // 实际发送 RPC 调用。'Call' 是一个阻塞调用，直到收到响应或超时。
    ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
    return ok
}
```

1.2.6 选举机制

1.2.6.1 发起选票的逻辑

其实就是实现 leaderElection() 和相关的函数

就是用来发起新的选举，那主要的内容就是，向别人要选票。

步骤如下所示。

1. 自己的任期 ++
2. 把自己定义成候选人，并给自己投一票
3. reset 选举时间
4. 构建 RequestVoteRPC 的结构体
5. 循环向所有人索取选票（调用 candidateRequestVote）

language

```
func (rf *Raft) leaderElection() {
    // 这里先把字段定义好
    rf.currentTerm++                // 先任期++，自己当领导先让自己进行
    rf.state = Candidate            // 首先先把自己定义成候选人
    rf.votedFor = rf.me             // 然后给自己投一票
    rf.persist()                    // 2C的，调用了也没用（后面再详细实现）
    rf.resetElectionTimer() // 重新设置
    term := rf.currentTerm // 设置当前的任期
    voteCounter := 1                // 因为已经给自己投了一票了，所以现在已经有一票了
    DPrintf("[%v]: start leader election, term %d\n", rf.me, rf.currentTerm)
    // 定义要选票的RPC
    args := RequestVoteArgs{
        Term:        term,
        CandidateId: rf.me,
    }
```

```
    var becomeLeader sync.Once
    // 遍历所有的人，让他们给我选票，怎么要：调用candidateRequestVote
    for serverId, _ := range rf.peers {
        if serverId != rf.me {
            go rf.candidateRequestVote(serverId, &args, &voteCounter, &becomeLeader)
        }
    }
}
```

1.2.6.2 处理选票的逻辑

candidateRequestVote()：发起 election 的人向别人索取选票的函数。

- 首先要构建一个 Request 的响应
- 有几种情况是是不能给获取的人选票的，具体见代码注释
- 如果走到下面说面可以给选票，VoteGranted 为真，就把自己的选票数量 ++ 一下
- 如果判断自己选票数量超过半数，改变自己的状态
  1. 设置自己为 Leader
  2. 马上发送一个心跳包

language

```
func (rf *Raft) candidateRequestVote(serverId int, args *RequestVoteArgs, voteCounter *int, becomeLeader
    *sync.Once) {
    DPrintf("[%d]: term %v send vote request to %d\n", rf.me, args.Term, serverId)
    // 先构构建一个响应
    reply := RequestVoteReply{}
    // 这里是调用网络接口，发送请求
    ok := rf.sendRequestVote(serverId, args, &reply)
    if !ok {
        return // 如果网络调用失败，函数就能直接返回了
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if reply.Term > args.Term {
    // 如果发现，你去索要票的时候，被索要票的人的任期比你自己还大，说明自己落后了，直接setNewTerm
        然后返回就行了
        DPrintf("[%d]: %d 在新的term，更新term，结束\n", rf.me, serverId)
        rf.setNewTerm(reply.Term)
        return
    }
    // 如果发现，别人的任期比自己的小，说明别人落后了，不能给你投票，返回
```

```
    if reply.Term < args.Term {
        DPrintf("[%d]: %d 的term %d 已经失效，结束\n", rf.me, serverId, reply.Term)
        return
    }
    // reply.VoteGranted为假表明别人没有把票给你，也要return
    if !reply.VoteGranted {
        DPrintf("[%d]: %d 没有投给me，结束\n", rf.me, serverId)
        return
    }
    DPrintf("[%d]: from %d term一致，且投给%d\n", rf.me, serverId, rf.me)
    // 如果走到这里，说明成功获得选票了
    *voteCounter++
    // 如果选票大于一半人，说明你可以成为领导者了
    if *voteCounter > len(rf.peers)/2 &&
        rf.currentTerm == args.Term &&
        rf.state == Candidate {
        DPrintf("[%d]: 获得多数选票，可以提前结束\n", rf.me)
    // 成为领导者，要做下面这些事情
        becomeLeader.Do(func() {
            DPrintf("[%d]: 当前term %d 结束\n", rf.me, rf.currentTerm)
            rf.state = Leader // 把自己的状态设置成Leader
            rf.appendEntries(true) // 马上给别人发送一次心跳
        })
    }
}
```

1.2.6.3 最后完善 RequestVote 函数

- 首先先看任期，如果申请人任期落后了，不给票
- 如果自己已经把票投给别人了，也不给票
- 如果满足所有条件（具体见代码注释），给票，然后记得重新设置自己的选举时间

language

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    // Your code here (2A, 2B).
    rf.mu.Lock()                          // 保证临界资源安全
    defer rf.mu.Unlock()
    // rules for servers
    // all servers 2
    if args.Term > rf.currentTerm {    // 更新自己的任期
        rf.setNewTerm(args.Term)
```

```
    }
    // request vote rpc receiver 1
    if args.Term < rf.currentTerm {    // 如果申请人的任期更小，说明申请人落后了，不给这个人选票
        reply.Term = rf.currentTerm        // 通过reply告诉申请人现在最新的任期
        reply.VoteGranted = false              // 不给这个人选票
        return
    }
    if (rf.votedFor == -1 || rf.votedFor == args.CandidateId) {
    // 第一个条件判断，票是不是已经给出去了，-1表示自己还没投票
    // 第二个条件判断，自己投的人是不是这个申请人
    // 两个条件中一个即可
        reply.VoteGranted = true
        rf.votedFor = args.CandidateId
        rf.resetElectionTimer() // 接的重新设置自己的选举时间
        DPrintf("[%v]: term %v vote %v", rf.me, rf.currentTerm, rf.votedFor)
    } else {
        reply.VoteGranted = false
    }
    reply.Term = rf.currentTerm
}
```

1.3 2A 部分测试结果

如图 1-6 所示，运行结果。经过多次测试，2A 都是可以通过的。

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/6.824/src/raft$ ls
config.go  log.go  persister.go  raft-b.go  raft.go  test_test.go  util.go
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/6.824/src/raft$ go test -run 2A
Test (2A): initial election ...
  ... Passed --   3.1  3  118   25074    0
Test (2A): election after network failure ...
  ... Passed --   4.5  3  244   38670    0
Test (2A): multiple elections ...
  ... Passed --   5.7  7 1164  173388    0
PASS
ok      6.824/raft      13.276s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/6.824/src/raft$
```

图 1-6 2A 运行结果

2 Part 2B: log

2.1 任务思路梳理

2.1.1 raft 中日志机制的原理

Raft **日志机制的关键要素**

1. **日志条目**（Log Entry）:

每个日志条目包含一条命令和它所属的任期（Term）。命令通常是系统需要复制和执行的操作，而任期则是日志条目创建时的 Raft 任期。

2. **日志复制**:

领导者负责将日志条目复制到其他所有服务器（追随者）。这是通过 AppendEntries RPC 实现的，该 RPC 既用作日志条目的复制，也用作领导者发送心跳信号。

3. **任期**（Term）:

Raft 通过任期来避免过时的信息导致的错误。如果一个节点的任期落后于其他节点，它将转换为追随者状态。

4. **日志一致性检查**:

在复制日志条目之前，领导者会检查是否与追随者的日志在前一个条目上一致。这是通过比较两者的任期和索引来实现的。

Raft **日志机制的工作原理**

1. **日志条目的创建**:

当一个客户端发送命令给领导者时，领导者会创建一个新的日志条目，并将其附加到自己的日志中。

2. **日志条目的复制**:

领导者尝试通过发送 AppendEntries RPC 将新的日志条目复制到所有追随者。这个 RPC 包含了要复制的日志条目以及前一个日志条目的任期和索引，用于一致性检查。

3. **一致性检查**:

追随者在收到 AppendEntries RPC 后，会检查自己的日志是否包含与领导者相同任期和索引的条目。如果检查失败，追随者会拒绝接受新的日志条目。

4. **日志条目的确认和提交**:

一旦领导者从大多数追随者那里得到了对一个日志条目的确认，它就会将该条目标记为已提交。然后，领导者会通知追随者该日志条目已经被提交。

5. **应用到状态机**:

一旦日志条目被提交，每个服务器（包括领导者和追随者）都会按照日志条目的顺序将命令应用到它们的状态机中。

6. **持久化和恢复**:

Raft 通过持久化日志条目和当前任期等信息来处理服务器的崩溃和重启。这些信息在启动时会被重新读取，以恢复其状态。

2.1.2 思维导图

这一部分其实就是在 2A 的基础上添加 ** 实现领导者和跟随者代码以附加新的日志条目 **。

图 2-1 raft 中日志控制流程

2.2 准备工作

2.2.1 封装 log 和 Entry

为了后续日志方便管理，我把所有日志条目和日志的处理进行了封装，代码如下所示。

这一部分是我学习了 Github **上的开源思路，学习的一种形式，我觉得通过封装一个** log.go **可以让代码显示更清晰，结构更明确**。

关于每一部分的原理和细节解释，我都在代码**注释**中解释。

language

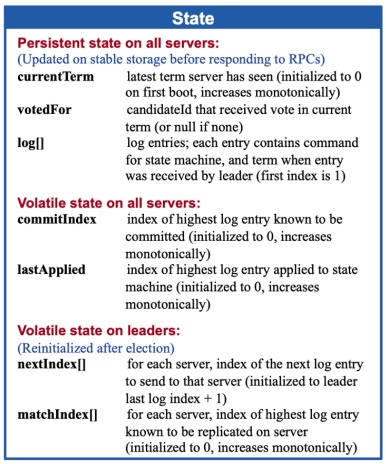
```
package raft
import (
    "fmt"
    "strings"
)
type Log struct {
    Entries  []Entry                // 存储日志条目的切片，每个条目是一个 Entry 类型
    Index0  int                     // 表示日志切片中第一个条目的索引，通常为0
}
type Entry struct {
    Command interface{}        // 存储实际的命令或数据
    Term    int                 // 表示该条目所属的任期
    Index   int                 // 表示该条目在日志中的索引
```

```
}
// 向日志中追加一个或多个条目 所以这里是用了可变参数的
func (l *Log) append(entries ...Entry) {
    l.Entries = append(l.Entries, entries...)
}
// 创建一个空的 Log 实例
func makeEmptyLog() Log {
    log := Log{
        Entries: make([]Entry, 0),      // 实例化一个空的Entry
        Index0:  0,                                  // 索引设置为0
    }
    return log
}
// 获取指定索引处的日志条目
func (l *Log) at(idx int) *Entry {
    return &l.Entries[idx]
}
// 截断日志，在指定索引处删除所有后续条目
func (l *Log) truncate(idx int) {
    l.Entries = l.Entries[:idx]
}
// 从指定索引开始返回日志条目的切片
func (l *Log) slice(idx int) []Entry {
    return l.Entries[idx:]
}
// 返回日志中的条目数量
func (l *Log) len() int {
    return len(l.Entries)
}
// 获取日志中的最后一个条目(的指针)
func (l *Log) lastLog() *Entry {
    return l.at(l.len() - 1)
}
// 将日志条目转换为字符串表示
func (e *Entry) String() string {
    return fmt.Sprint(e.Term)
}
// 将整个日志转换为字符串表示
func (l *Log) String() string {
    nums := []string{}
    for _, entry := range l.Entries {
```

```
        nums = append(nums, fmt.Sprintf("%4d", entry.Term))
    }
    return fmt.Sprint(strings.Join(nums, "|"))
}
```

2.2.2 完善 raft 结构

如图所示。



- state：就是当前节点的状态，就是刚才提到的枚举
- appendEntryCh：定义的是一个管道，然后这个管道是用来追加日志条目的
- heartBeat：定义了一个发送心跳包的频率，用time包里面的duration
- electionTime：下一次选举的计时器，用于触发新的领导者选举

- currentTerm：就是当前节点的任期，这个很好理解
- voteFor：表示在当前任期中，我的票是投给谁了？（一开始设置成-1，然后就能防止在同一任期里面给多个人投票）
- Log：就是存日志条目的日志结构，具体的一开始已经写好了，不用我来写
- lastApplied：已被应用到状态机的最高日志条目索引，可以确保状态机的更新是按照日志的顺序进行的（因为运行过程中是不阻塞的）
- nextIndex：对于每个同伴节点，要发送的下一个日志条目的索引。
- matchIndex：对于每个同伴节点，已知的已复制到该同伴节点的最高日志条目的索引。
- applyCh：我定义的用来发送applyMsg的通道
- applyCond：线程同步，条件变量来的

图 2-2 raft 结构分析图

language

```
// A Go object implementing a single Raft peer.
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()
    // Your data here (2A, 2B, 2C).
    // Look at the paper's Figure 2 for a description of what
    // state a Raft server must maintain.
    state          RaftState
    appendEntryCh chan *Entry                // 2B
    heartBeat      time.Duration
    electionTime  time.Time
    // Persistent state on all servers:
    currentTerm int
    votedFor    int
  // ------------------------------ 2B部分
      ------------------------------ //
```

```
    log           Log                                    // 存储日志的日志结构
    // Volatile state on all servers:
    commitIndex int                                      // 最后提交的日志的索引
    lastApplied int                                      // 被应用到状态机的最后的日志索引
    // Volatile state on leaders:
    nextIndex  [] int                                    // 对于每个同伴节点，要发送的下一个日志条目的
        索引
    matchIndex []int                                     // 对于每个同伴节点，已知的已复制到该同伴节点
        的最高日志条目的索引
  // ------------------------------ 2B部分
      ------------------------------ //
    applyCh  chan ApplyMsg        // 定义用来发送applyMsg的通道
    applyCond *sync.Cond
}
```

2.3 任务分析、实现流程和代码实现

2.3.1 实现 Start() 方法

开始时先实现 Start() 方法，然后编写代码通过 AppendEntries RPC 发送和接收新的日志条目，参照论文中的图 2。将每个已提交的条目在每个节点上通过 applyCh 发送。

这个步骤有两个部分:

1. 首先就是先把 Start() 方法写好。

2. 其次就是修改 appendEntries 函数，这个函数本来是只用来处理心跳包现在要求除了处理心跳包之外，还需要处理日志复制。

我们在 2A 部分实现的 appendEntries 的时候，参数都是传 True，表示都是心跳包，当时的 Entry 都是空的。现在我们在 Start() 里面是肯定要传递 False 的了。

先整理一下 Start() 的思路:

·判断当前是否是 Leader，如果是 Leader 才有资格添加新的日志条目

·创建一个新的日志条目。包括命令、索引和任期

·将新的日志条目添加到日志当中（我们前面写了管理日志的数据结构了，因此这里直接 append，非常方便）

·调用 appendEntries 把这个日志复制给他人

代码如下所示。

language

```
func (rf *Raft) Start(command interface{}) (int, int, bool) {
```

```
    /*
        (int, int, bool): 函数返回三个值: 日志条目的索引（int），当前任期（int），
        以及这个服务器是否认为自己是领导者（bool）
    */
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // 如果当前服务器不是领导者，则立即返回。在Raft中，只有领导者才能添加新的日志条目
    if rf.state != Leader {
        return -1, rf.currentTerm, false
    }
    // 这里计算新日志条目的索引（是最后一个日志条目的索引加1）和获取当前的任期
    index := rf.log.lastLog().Index + 1
    term := rf.currentTerm
    // 创建一个新的日志条目。包括命令、索引和任期
    log := Entry{
        Command: command,
        Index:   index,
        Term:    term,
    }
    // 将新的日志条目追加到日志中
    rf.log.append(log)
    // rf.persist()
    DPrintf("[%v]: term %v Start %v", rf.me, term, log)
    // 复制日志条目到其他服务器
    rf.appendEntries(false)
    // 返回新日志条目的索引、当前任期，以及一个表示该服务器是领导者的true值
    return index, term, true
}
```

2.3.2 实现选举限制

根据论文 5.4.1 节，实现选举限制。

其实就是在 2A 的基础上补充一些东西，如图 2-3 所示。

其实就是在 RequestVoteArgs 结构体里面添加和日志相关的信息即可。

在下列代码块中，除了标有注释的，其他的都是 2A 的内容，在 2A 的部分我已经详细分析过了这一份代码了。

language

```
func (rf *Raft) leaderElection() {
    rf.currentTerm++
    rf.state = Candidate
```

```go
func (rf *Raft) leaderElection() {
    rf.currentTerm++
    rf.state = Candidate
    rf.votedFor = rf.me
    rf.persist()
    rf.resetElectionTimer()
    term := rf.currentTerm
    voteCounter := 1
    // lastLog := rf.log.lastLog()
    DPrintf("[%v]: start leader election, term
    %d\n", rf.me, rf.currentTerm)
    args := RequestVoteArgs{
        Term: term,
        CandidateId: rf.me,
    }

    var becomeLeader sync.Once
    for serverId, _ := range rf.peers {
        if serverId != rf.me {
            go
            rf.candidateRequestVote(serverId,
            &args, &voteCounter,
            &becomeLeader)
        }
    }
}
```

```go
func (rf *Raft) leaderElection() {
    rf.currentTerm++
    rf.state = Candidate
    rf.votedFor = rf.me
    rf.persist()
    rf.resetElectionTimer()
    term := rf.currentTerm
    voteCounter := 1
    lastLog := rf.log.lastLog()
    DPrintf("[%v]: start leader election, term %d\n",
    rf.me, rf.currentTerm)
    args := RequestVoteArgs{
        Term: term,
        CandidateId: rf.me,
        LastLogIndex: lastLog.Index,
        LastLogTerm: lastLog.Term,
    }

    var becomeLeader sync.Once
    for serverId, _ := range rf.peers {
        if serverId != rf.me {
            go rf.candidateRequestVote(serverId,
            &args, &voteCounter, &becomeLeader)
        }
    }
}
```

图 2-3 2A 和 2B 的 leaderElection 对比

```go
    rf.votedFor = rf.me
    rf.persist() // 2C，还没写，调用了也没用
    rf.resetElectionTimer()
    term := rf.currentTerm
    voteCounter := 1
    lastLog := rf.log.lastLog() // 这里先获取上一次最后的log结构，因为申请票的时候要告诉别人这个的
    DPrintf("[%v]: start leader election, term %d\n", rf.me, rf.currentTerm)
    args := RequestVoteArgs{
        Term:         term,
        CandidateId:  rf.me,
        LastLogIndex: lastLog.Index, // 告诉别人日志信息，这里最新一条日志的下标
        LastLogTerm: lastLog.Term, // 这里是告诉别人上一条日志的任期
    }
    var becomeLeader sync.Once
    for serverId, _ := range rf.peers {
        if serverId != rf.me {
            go rf.candidateRequestVote(serverId, &args, &voteCounter, &becomeLeader)
        }
    }
}
```

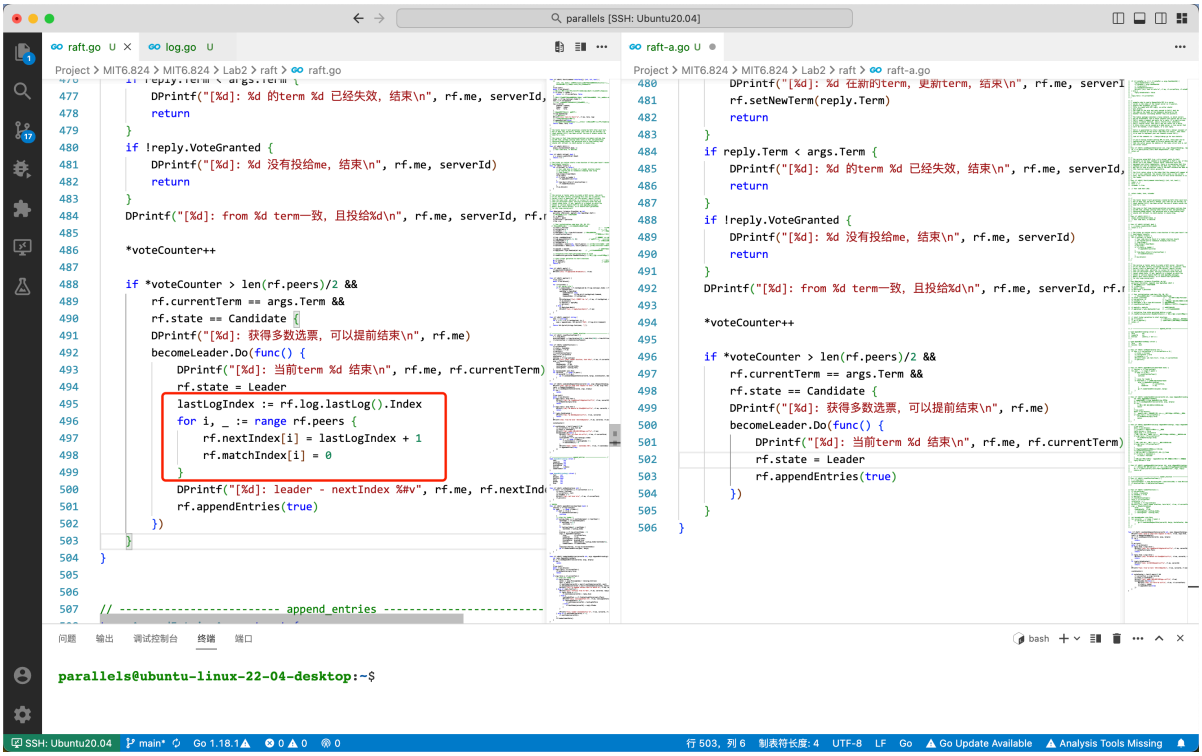


图 2-4 2A 和 2B 的 candidateRequestVote 对比

2.3.3 补充 candidateRequestVote 里面和日志相关的细节

在这个函数的前面部分和 2A 是一样的，首先就是要判断要不要给票给你，所以前面是 2A 的内容。

决定要给票之后，肯定是要更新日志的。

1. ‘lastLogIndex := rf.log.lastLog().Index‘：这一行获取当前日志的最后一个条目的索引。这是为了知道新领导者的日志中最后一个条目的位置。

2. 接下来的循环 ‘for i, _ := range rf.peers ...‘ 遍历所有的同伴节点（包括领导者自己）。

3. ‘rf.nextIndex[i] = lastLogIndex + 1‘：对于每个同伴，设置 ‘nextIndex‘ 为新领导者日志的最后一个条目的下一个位置。‘nextIndex‘ 数组用于跟踪领导者需要从哪个索引开始向每个同伴复制日志条目。因为新领导者刚上任，所以假设从最后一个条目之后的位置开始发送。

4. ‘rf.matchIndex[i] = 0‘：同时，将每个同伴的 ‘matchIndex‘ 设置为 0。‘matchIndex‘ 数组用于记录每个同伴已知的与领导者日志一致的最高日志条目的索引。由于新领导者刚上任，初始假设为没有与任何同伴的日志一致。

language

```
func (rf *Raft) candidateRequestVote(serverId int, args *RequestVoteArgs, voteCounter *int, becomeLeader
    *sync.Once) {
```

```
// ------------------- 2A部分 ---------------------- //
    reply := RequestVoteReply{}
    ok := rf.sendRequestVote(serverId, args, &reply)
    if !ok {
        return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if reply.Term > args.Term {
        rf.setNewTerm(reply.Term)
        return
    }
    if reply.Term < args.Term {
        return
    }
    if !reply.VoteGranted {
        return
    }
// ---------------------- 2A部分 ---------------------- //
    *voteCounter++
    if *voteCounter > len(rf.peers)/2 &&
        rf.currentTerm == args.Term &&
        rf.state == Candidate {
        becomeLeader.Do(func() {
            rf.state = Leader
      // 就是这些部分，是2B才有的
            lastLogIndex := rf.log.lastLog().Index
            for i, _ := range rf.peers {
                rf.nextIndex[i] = lastLogIndex + 1
                rf.matchIndex[i] = 0
            }
            DPrintf("[%d]: leader - nextIndex %#v", rf.me, rf.nextIndex)
            rf.appendEntries(true)
        })
    }
}
```

2.3.4 避免重复选举的问题

Lab 2B 早期测试中一个常见的失败原因是即使领导者还活着，也会持续进行重复选举。检查选举定时器的管理或者是否在赢得选举后立即发送心跳，这可能是导致问

题的原因。

在 2A 里面已经处理过了，在 candidateRequestVote 函数中，如果发现选票过半数，把自己升级成 Leader 之后，会马上向所有人发送一个心跳包。

2.3.5 循环检查事件的代码优化

如果代码中有循环不断地检查某些事件，不要让这些循环不间断地执行，因为这会减慢您的实现，导致测试失败。使用 Go 的条件变量，或者在每次循环迭代中插入 ‘time.Sleep(10 * time.Millisecond)‘。

在测试中我也遇到了非常多的问题，我采用的方式其实是日志 +Sleep 的方法进行 debug，这样我觉得对于我来说是最高效的。

2.3.6 补充 RequestVoteRPC 结构体

需要补充的只有 RequestVoteArgs 请求结构体，如下代码所示。

language

```
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    Term         int    // 候选人当前的任期
    CandidateId int    // 候选人的id，你向别人去申请票，要告诉别人自己的id
  // ------------------------------ 2B
      ------------------------------ //
    LastLogIndex int   // 上一条日志的索引
    LastLogTerm int    // 上一条日志的任期
  // ------------------------------ 2B
      ------------------------------ //
}
```

2.3.7 补充 RequestVote 函数

判断日志是否是最新的（这个是有两种情况的）

1. 如果获取到的日志中的 Term 比自己的 Term 更新，说明这是最新的日志

2. 如果获取到的日志和自己的是同期的，但是最后一条索引 >= 自己的，也表示获取到的是最新的日志）

language

```
// example RequestVote RPC handler.
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    // Your code here (2A, 2B).
```

```
// ---------------------------------------- 仅2A
    ---------------------------------------- //
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // rules for servers
    // all servers 2
    if args.Term > rf.currentTerm {
        rf.setNewTerm(args.Term)
    }
    // request vote rpc receiver 1
    if args.Term < rf.currentTerm {
        reply.Term = rf.currentTerm
        reply.VoteGranted = false
        return
    }
// ---------------------------------------- 仅2A
    ---------------------------------------- //
    // request vote rpc receiver 2
    myLastLog := rf.log.lastLog()                                    // 这里先获取最新的日志
    upToDate := args.LastLogTerm > myLastLog.Term ||
        (args.LastLogTerm == myLastLog.Term && args.LastLogIndex >= myLastLog.Index) // 判断日志
            是否是最新的（这个是有两种情况的 1. 如果获取到的日志中的Term比自己的Term更新，说明这是
            最新的日志 2. 如果获取到的日志和自己的是同期的，但是最后一条索引>=自己的，也表示获取到
            的是最新的日志）
    if (rf.votedFor == -1 || rf.votedFor == args.CandidateId) && upToDate {
    // 这里if的判断条件和2A是有不同的，这里能走到里面，必须满足我们获取到的日志已经是最新的了才行
        reply.VoteGranted = true
        rf.votedFor = args.CandidateId
        rf.persist() // 日志持久化，现在还没有实现，调用后是没有效果的
        rf.resetElectionTimer()
        DPrintf("[%v]: term %v vote %v", rf.me, rf.currentTerm, rf.votedFor)
    } else {
        reply.VoteGranted = false
    }
    reply.Term = rf.currentTerm
}
```

2.3.8 完善 Make 函数

需要完善的有两个重点的部分:

1. 初始化新加上的和日志相关的字段。

2. 还要启动一个新的协程, 把新创建的这个节点的日志推送到状态机上, 即 applier 函数。

language

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    // 这里首先是创建一个Raft节点
    rf := &Raft{}
    rf.peers = peers
    rf.persister = persister
    rf.me = me

    // Your initialization code here (2A, 2B, 2C).
    // 然后后面这里就是继续初始化节点里面的字段了
    // ---------------------------- 2A ---------------------------- //
    rf.state = Follower                    // 一开始设置身份就是Follower
    rf.currentTerm = 0                     // 任期是0
    rf.votedFor = -1                           // 前面提到过了，先设置成-1，表示没有把票投给别人
    rf.heartBeat = 50 * time.Millisecond // 50ms的心跳周期
    rf.resetElectionTimer()              // 重置选举计时器，为发起新选举做准备
    // ---------------------------- 2A ---------------------------- //
        // ---------------------- 2B ------------------------------ //
    rf.log = makeEmptyLog()              // 先生成一个空的日志
    rf.log.append(Entry{-1, 0, 0})       // 向日志中追加一个初始条目
    rf.commitIndex = 0                   // 设置相关索引
    rf.lastApplied = 0
    rf.nextIndex = make([]int, len(rf.peers)) // 初始化每个同伴的下一个日志索引数组
    rf.matchIndex = make([]int, len(rf.peers))// 初始化每个同伴已复制日志的索引数组
    // ---------------------- 2B ------------------------------ //
    rf.applyCh = applyCh                                  // 送日志的通道
    rf.applyCond = sync.NewCond(&rf.mu)  // 线程同步的条件变量

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState()) // 如果是因为crash，重新启动的时候读取上一个状态

    // start ticker goroutine to start elections
    go rf.ticker()                             // 计时器，用于管理选举和心跳的发送
  // ---------------------- 2B ------------------------------ //
    go rf.applier()                            // 弄日志的，在这里面就是把该弄的日志弄好，弄到状态机上
  // ---------------------- 2B ------------------------------ //
    return rf
}
```

2.3.9 编写 applier 函数

这个 ‘applier‘ 函数是 Raft 实现中的一个关键部分，负责将已提交的日志应用到状态机。它运行在一个独立的 goroutine 中，持续检查是否有新的日志条目可以应用。

1. 互斥锁的使用：函数开始时调用 ‘rf.mu.Lock()‘ 获取锁，并在函数退出前释放锁 ‘defer rf.mu.Unlock()‘。这是为了确保在访问和修改 Raft 状态（如 ‘commitIndex‘ 和 ‘lastApplied‘）时的线程安全。

2. 循环检查：‘for !rf.killed() ...‘ 这个循环会持续运行，直到 Raft 节点被明确停止（通过 ‘killed‘ 方法）。这确保了 Raft 服务器在其生命周期内持续处理日志条目。

3. 应用日志条目：

‘if rf.commitIndex > rf.lastApplied && rf.log.lastLog().Index > rf.lastApplied‘：这个条件检查是否有新的日志条目需要应用。‘commitIndex‘ 是已知的最大已提交日志条目索引，‘lastApplied‘ 是已应用到状态机的最后一个日志条目的索引。如果 ‘commitIndex‘ 大于 ‘lastApplied‘，则表示有新的日志条目需要应用。

‘rf.lastApplied++‘：增加 ‘lastApplied‘，准备应用下一个日志条目。

‘applyMsg := ApplyMsg...‘：创建一个包含要应用的日志命令的 ‘ApplyMsg‘ 结构体。

‘rf.applyCh <- applyMsg‘：通过 ‘applyCh‘ 通道发送 ‘ApplyMsg‘，这允许与 Raft 服务器交互的服务（如键值存储服务）接收并处理该命令。

4. 互斥锁和并发处理：

在发送 ‘applyMsg‘ 之前，代码先解锁 ‘rf.mu.Unlock()‘，然后在发送后重新加锁 ‘rf.mu.Lock()‘。这样做是为了避免在发送消息时持有锁，从而防止阻塞其他需要访问 Raft 状态的操作。

这种锁的手动释放和再次获取是必要的，因为发送操作可能会阻塞，我们不希望在此期间持有锁，这可能会导致死锁或性能问题。5. 等待新的日志条目：如果当前没有新的日志条目可供应用，函数会通过 ‘rf.applyCond.Wait()‘ 调用进入等待状态。这是一个条件变量等待，当新的日志条目被提交时，其他部分的代码（可能在不同的 goroutine 中）会通知这个条件变量，从而唤醒等待的 ‘applier‘ goroutine 继续处理新的日志条目。

代码如下。

language

```
func (rf *Raft) applier() {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    for !rf.killed() {
        // all server rule 1
    // 检查是否有新的日志条目需要应用
```

```
        if rf.commitIndex > rf.lastApplied && rf.log.lastLog().Index > rf.lastApplied {
            rf.lastApplied++
    // 创建一个包含要应用的日志命令的 'ApplyMsg' 结构体
            applyMsg := ApplyMsg{
                CommandValid: true,
                Command:    rf.log.at(rf.lastApplied).Command,
                CommandIndex: rf.lastApplied,
            }
            DPrintVerbose("[%v]: COMMIT %d: %v", rf.me, rf.lastApplied, rf.commits())
            rf.mu.Unlock()
            rf.applyCh <- applyMsg // 把这个日志放到通道中去
            rf.mu.Lock()
        } else {
            rf.applyCond.Wait() // 进入条件变量进行等待，如果有新的日志被提交的时候，这个才会被唤醒。
            DPrintf("[%v]: rf.applyCond.Wait()", rf.me)
        }
    }
}
```

2.3.10 论文里面提到的 2B 部分的优化方法

在 Raft 共识算法中，日志复制是一个核心部分，领导者（Leader）负责将自己的日志条目复制到所有的跟随者（Follower）。然而，日志条目可能在不同的服务器上出现不一致，这通常发生在领导者更换或网络分区恢复时。为了确保日志一致性，领导者需要找到一个与所有跟随者共同一致的日志点，然后从这个点开始重新复制日志条目。

在早期的 Raft 实现中，处理日志不一致的过程比较原始。如果一个跟随者发现 'AppendEntries' 请求中的 'PrevLogIndex' 或 'PrevLogTerm' 与自己的日志不匹配，它会简单地拒绝该请求。领导者收到拒绝后，会逐个减少 'nextIndex'（即准备发送给该跟随者的下一个日志条目的索引），并重试，直到找到匹配的点。这个过程可能非常低效，特别是当不一致发生在较早的日志条目时。

优化的做法是通过更详细的冲突信息来加速这个过程。当跟随者拒绝一个 'AppendEntries' 请求时，它不仅仅告诉领导者“我拒绝这个请求”，而是提供具体的冲突信息，包括：

- 'XTerm': 发生冲突的日志条目的任期。

- 'XIndex': 发生冲突的日志条目的索引。

- 'XLen': 跟随者日志的长度。

使用这些信息，领导者可以更快地定位到日志不一致的确切位置。具体来说：

1. 如果 'XTerm' 在领导者日志中找到了，领导者可以将 'nextIndex' 直接设置为

该任期在自己日志中最后一次出现的位置加一。

2. 如果 ‘XTerm‘ 在领导者日志中没有找到，领导者可以将 ‘nextIndex‘ 设置为 ‘XIndex‘。

这种方法比简单逐个递减 ‘nextIndex‘ 要高效得多，因为它减少了领导者和跟随者之间不必要的通信次数，尤其是在存在较大日志差异的情况下。

2.3.11 补充 appendEntriesRPC

所以为了实现日志功能和优化的功能，现在需要完善 appendEntries 的 RPC。

language

```
type AppendEntriesArgs struct {
    Term         int
    LeaderId     int
    PrevLogIndex int        // 新条目紧随之前的条目的索引值(这用于确保日志的一致性)
    PrevLogTerm int         // PrevLogIndex处条目的任期号(同样用于日志一致性校验)
    Entries      []Entry    // 要复制到跟随者日志中的日志条目数组, 如果是心跳检测，这个数组可能为空
    LeaderCommit int        // 领导者的commitIndex。如果跟随者的commitIndex较小，则会更新至此值
}

type AppendEntriesReply struct {
    Term    int
    Success bool
    Conflict bool   // 表示是否存在日志冲突, 如果为 true, 则跟随者的日志与领导者的日志不一致
    XTerm   int     // 发生冲突的日志条目的任期号。如果存在冲突，这有助于领导者快速定位问题
    XIndex  int     // 发生冲突的日志条目的索引。领导者可以使用这个信息来更新其对应跟随者的nextIndex
    XLen    int     // 跟随者日志的长度。如果领导者的日志比跟随者的长，这个信息有助于领导者正确处理日
        志复制
}
```

2.3.12 完善 appendEntries 函数

在里面补充 2B 的部分。

1. 获取最后一个日志条目: 函数首先获取当前节点（也就是 leader）的最后一个日志条目 ‘lastLog‘。

2. 遍历所有的对等节点（peers）：

- 对于每个对等节点（除了自身），函数会检查是否需要向该节点发送日志条目。

- 如果 ‘heartbeat‘ 参数为真，或者该对等节点的 ‘nextIndex‘（下一个需要发送的日志索引）小于或等于 ‘lastLog‘ 的索引，则表示需要向该节点发送数据。

3. 准备 ‘AppendEntries‘ 参数：

- 函数会准备一个 'AppendEntriesArgs' 结构体,其中包括当前任期 ('currentTerm'), 领导者的 ID ('LeaderId'), 以及关于之前日志条目的信息 ('PrevLogIndex' 和 'PrevLogTerm')。这些信息用于确保日志的一致性。

- 'Entries' 字段被初始化为从 'nextIndex' 开始的所有日志条目的副本。这是日志复制的核心部分。

- 'LeaderCommit' 字段设置为领导者的 'commitIndex', 这代表领导者已经提交的日志条目的最高索引。

4. 发送 'AppendEntries' 请求:

- 函数使用协程 ('go' 关键字) 来异步地向每个对等节点发送 'AppendEntries' RPC 请求。这是通过调用 'leaderSendEntries' 函数完成的, 它实际上执行网络调用来发送数据。

5. 重置选举计时器:

- 对于领导者自己, 会调用 'resetElectionTimer' 函数来重置选举计时器。这是为了防止领导者因超时而开始新的选举。

language

```
func (rf *Raft) appendEntries(heartbeat bool) {
    lastLog := rf.log.lastLog()
  // 遍历所有的节点
    for peer, _ := range rf.peers {
        if peer == rf.me {
            rf.resetElectionTimer()     // 如果遍历到自己的节点，先重新设置自己的计时器
            continue
        }
        // rules for leader 3
   // 判断是否需要向特定的对等节点（peer）发送信息
        if lastLog.Index >= rf.nextIndex[peer] || heartbeat {
            nextIndex := rf.nextIndex[peer]              // 获取要发送给该对等节点的下一个日志条目的索引
            if nextIndex <= 0 {
                nextIndex = 1                                                    // 这里保证日志的nextIndex是大
                    于0的
            }
            if lastLog.Index+1 < nextIndex {          // 确保 nextIndex 不超过领导者的最后一个日志条目索
                引
                nextIndex = lastLog.Index
            }
            prevLog := rf.log.at(nextIndex - 1)     // 获取紧邻要发送的日志条目之前的日志条目
      // 构建AppendEntriesArgs结构体，这个相对于2A部分多了PrevLogIndex, PrevLogTerm和
          LeaderCommit字段
      // 对比2A来说，Entries字段也从空的变成非空了
```

```
            args := AppendEntriesArgs{
                Term:         rf.currentTerm,
                LeaderId:     rf.me,
                PrevLogIndex: prevLog.Index,
                PrevLogTerm: prevLog.Term,
                Entries:      make([]Entry, lastLog.Index-nextIndex+1),
                LeaderCommit: rf.commitIndex,
            }
        // 将从nextIndex开始的日志条目复制到args.Entries
            copy(args.Entries, rf.log.slice(nextIndex))
            go rf.leaderSendEntries(peer, &args)
        }
    }
}
```

‘if lastLog.Index >= rf.nextIndex[peer] || heartbeat‘ 这一行判断是否需要向特定的对等节点（peer）发送信息。这里有两个条件：

如果领导者的最后日志条目索引（‘lastLog.Index‘）大于或等于要发送给该对等节点的下一个日志索引（‘rf.nextIndex[peer]‘），表示有新的日志条目需要被复制到该节点。

如果 ‘heartbeat‘ 为真，即使没有新的日志条目，也会发送心跳信息以维持领导者的状态。

2.3.13 完善 appendEntries 要调用的 leaderSendEntries 函数

这个函数和 2A 部分的相比较添加了很多内容。

1. 发送 ‘AppendEntries‘ 请求:

- ‘ok := rf.sendAppendEntries(serverId, args, reply)‘ 发送 ‘AppendEntries‘ RPC 请求到指定的服务器（通过 ‘serverId‘ 指定）。请求的参数包含在 ‘args‘ 中，而响应保存在 ‘reply‘ 中。

- 如果 ‘ok‘ 为 ‘false‘，表示请求未能成功发送或接收响应，函数返回。

2. 获取锁:

- ‘rf.mu.Lock()‘ 和 ‘defer rf.mu.Unlock()‘ 确保在处理响应的过程中，对 Raft 状态的访问是同步的，避免并发访问导致的问题。

3. 处理任期不匹配:

- 如果响应中的任期（‘reply.Term‘）大于当前任期（‘rf.currentTerm‘），则更新当前任期为响应中的任期，并返回。这意味着当前领导者可能过时，需要回退到跟随者状态。

4. 处理成功的响应:

- 如果响应表明成功 (reply.Success)，则进行以下更新：

- 更新 ‘rf.nextIndex[serverId]‘ 为 ‘max(rf.nextIndex[serverId], next)‘，其中 ‘next‘ 是确认接收的日志条目之后的下一个索引。- 更新 ‘rf.matchIndex[serverId]‘ 为 ‘max(rf.matchIndex[serverId], match)‘，其中 ‘match‘ 是确认接收的最后一个日志条目的索引。

5. 处理日志不一致的情况：

- 如果响应表明存在冲突 (reply.Conflict)，则进行以下处理：

- 如果 ‘reply.XTerm‘ 为 -1，表示跟随者没有包含在请求中的 ‘PrevLogIndex‘ 的日志条目。此时，将 ‘rf.nextIndex[serverId]‘ 设置为跟随者的日志长度 ‘reply.XLen‘。

- 否则，查找当前领导者日志中与 ‘reply.XTerm‘ 相同任期的最后一个日志条目索引。如果找到，将 ‘rf.nextIndex[serverId]‘ 设置为该索引，否则设置为 ‘reply.XIndex‘。

6. 更新领导者的 ‘nextIndex‘：

- 如果没有日志冲突且 ‘rf.nextIndex[serverId]‘ 大于 1，则将其减一。这是为了处理可能存在的日志不一致。

7. 更新领导者的提交索引：

- 调用 ‘rf.leaderCommitRule()‘，这个函数根据 Raft 的规则检查是否可以提交更多的日志条目。如果一半以上的服务器已经复制了特定的日志条目，并且该条目属于当前任期，则该条目可以被提交。

代码如下。

language

```
func (rf *Raft) leaderSendEntries(serverId int, args *AppendEntriesArgs) {
    var reply AppendEntriesReply
    // 发送AppendEntriesRPC请求到指定的人(2A)
    ok := rf.sendAppendEntries(serverId, args, &reply)
    if !ok {
        return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // 这里处理一下任期不匹配的问题，这里在2A部分已经详细解释过了
    if reply.Term > rf.currentTerm {
        rf.setNewTerm(reply.Term)     // 设置新的任期
        return
    }
    // ---------- 这里上面都是2A的部分，下面是2B要做的工作 ---------- //
    if args.Term == rf.currentTerm {
        // rules for leader 3.1
```

```
        if reply.Success {
            // 处理成功的响应
            match := args.PrevLogIndex + len(args.Entries)
            next := match + 1
            rf.nextIndex[serverId] = max(rf.nextIndex[serverId], next)
            rf.matchIndex[serverId] = max(rf.matchIndex[serverId], match)
            DPrintf("[%v]: %v append success next %v match %v", rf.me, serverId, rf.nextIndex[serverId],
                rf.matchIndex[serverId])
        } else if reply.Conflict {
            // 处理日志不一致的情况
            DPrintf("[%v]: Conflict from %v %#v", rf.me, serverId, reply)
            if reply.XTerm == -1 {
                // 跟随者没有包含在请求中的PrevLogIndex的日志条目
                rf.nextIndex[serverId] = reply.XLen
            } else {
                // 否则，查找当前领导者日志中与 reply.XTerm 相同任期的最后一个日志条目索引
                lastLogInXTerm := rf.findLastLogInTerm(reply.XTerm)
                DPrintf("[%v]: lastLogInXTerm %v", rf.me, lastLogInXTerm)
                if lastLogInXTerm > 0 {
                    rf.nextIndex[serverId] = lastLogInXTerm
                } else {
                    rf.nextIndex[serverId] = reply.XIndex
                }
            }
            DPrintf("[%v]: leader nextIndex[%v] %v", rf.me, serverId, rf.nextIndex[serverId])
        } else if rf.nextIndex[serverId] > 1 {
            // 更新领导者的 nextIndex
            rf.nextIndex[serverId]--
        }
        // 更新领导者的提交索引
        rf.leaderCommitRule() // 但是要检查，是否能够提交
    }
}
```

2.3.14 编写 leaderSendEntries 要调用的 leaderCommitRule 函数

这个函数的目的是：**帮助领导者决定哪些日志条目可以被安全地提交**。这个过程基于 Raft 的核心概念之一，即只有当大多数节点都复制了某个日志条目时，该条目才被认为是可提交的。

language

```
func (rf *Raft) leaderCommitRule() {
    // leader rule 4
    // 首先检查当前节点是否为领导者，只有领导者才能执行提交规则
    if rf.state != Leader {
        return
    }

    // 遍历所有尚未提交的日志条目
    for n := rf.commitIndex + 1; n <= rf.log.lastLog().Index; n++ {
        // 只考虑当前任期中的日志条目，跳过任期不匹配的条目
        if rf.log.at(n).Term != rf.currentTerm {
            continue
        }
        // 统计复制了该条目的节点数量，从1开始计算（包括领导者自己）
        counter := 1
        // 遍历所有对等节点（peers），检查它们的matchIndex
        for serverId := 0; serverId < len(rf.peers); serverId++ {
            // 如果该节点已经复制了这个日志条目（即matchIndex大于等于该日志条目的索引）
            if serverId != rf.me && rf.matchIndex[serverId] >= n {
                counter++
            }
            // 如果超过半数的节点已经复制了这个日志条目
            if counter > len(rf.peers)/2 {
                // 更新commitIndex为该条目的索引
                rf.commitIndex = n
                // 打印日志，用于调试
                DPrintf("[%v] leader尝试提交 index %v", rf.me, rf.commitIndex)
                // 应用这个日志条目到状态机
                rf.apply()
                // 一旦找到可以提交的日志条目，跳出循环
                break
            }
        }
    }
}
```

2.3.15 完善 AppendEntries 里面日志部分

这个函数中，相比于 2A，主要增加了对日志条目的处理逻辑，包括:

- 检查日志的一致性，确保领导者的上一个日志条目在跟随者的日志中存在且任期

匹配。

- 如果存在冲突，设置回复中的冲突信息，包括冲突日志条目的任期、索引和跟随者的日志长度。

- 如果接收的日志条目在本地日志中不存在，则将其追加到本地日志中。

- 更新本地的 ‘commitIndex‘ 以反映从领导者那里接收的已提交日志条目的最高索引。

language

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    DPrintf("[%d]: (term %d) follower 收到 [%v] AppendEntries %v, prevIndex %v, prevTerm %v", rf.me
        , rf.currentTerm, args.LeaderId, args.Entries, args.PrevLogIndex, args.PrevLogTerm)
    // rules for servers
    // all servers 2
    // 初始化回复中的成功标志为 false 和返回当前的任期
    reply.Success = false
    reply.Term = rf.currentTerm
    // 如果请求的任期大于当前任期，更新当前任期并返回
    if args.Term > rf.currentTerm {
        rf.setNewTerm(args.Term)
        return
    }
    // append entries rpc 1
    // 如果请求的任期小于当前任期，直接返回
    if args.Term < rf.currentTerm {
        return
    }
    // 重置选举计时器，因为接收到了领导者的消息
    rf.resetElectionTimer()

    // candidate rule 3
    // 如果当前节点的状态是候选者，那么转换成跟随者
    if rf.state == Candidate {
        rf.state = Follower
    }
    // append entries rpc 2
    // 检查日志一致性。如果本地日志中没有领导者的上一个日志条目，设置冲突标志
    if rf.log.lastLog().Index < args.PrevLogIndex {
        reply.Conflict = true // 设置冲突标志
        reply.XTerm = -1
```

```
        reply.XIndex = -1
        reply.XLen = rf.log.len()
        DPrintf("[%v]: Conflict XTerm %v, XIndex %v, XLen %v", rf.me, reply.XTerm, reply.XIndex,
            reply.XLen)
        return
    }
    // 检查领导者的上一个日志条目的任期是否与本地日志一致
    if rf.log.at(args.PrevLogIndex).Term != args.PrevLogTerm {
        reply.Conflict = true
        xTerm := rf.log.at(args.PrevLogIndex).Term
        // 查找本地日志中最后一个与 xTerm 任期匹配的日志条目索引
        // 这里直接用for循环遍历查找了
        for xIndex := args.PrevLogIndex; xIndex > 0; xIndex-- {
            if rf.log.at(xIndex-1).Term != xTerm {
                reply.XIndex = xIndex
                break
            }
        }
        reply.XTerm = xTerm
        reply.XLen = rf.log.len()
        DPrintf("[%v]: Conflict XTerm %v, XIndex %v, XLen %v", rf.me, reply.XTerm, reply.XIndex,
            reply.XLen)
        return
    }
    // 处理日志条目
    for idx, entry := range args.Entries {
        // append entries rpc 3
        // 如果本地日志中的条目与领导者的条目冲突，则截断本地日志
        if entry.Index <= rf.log.lastLog().Index && rf.log.at(entry.Index).Term != entry.Term {
            rf.log.truncate(entry.Index)     // 这里就用到了我们提前定义的方法了
            rf.persist()
        }
        // append entries rpc 4
        // 如果本地日志中没有领导者的条目，则追加到本地日志
        if entry.Index > rf.log.lastLog().Index {
            rf.log.append(args.Entries[idx :]...)
            DPrintf("[%d]: follower append [%v]", rf.me, args.Entries[idx:])
            rf.persist()
            break
        }
    }
```

```
    // append entries rpc 5
    // 如果领导者的提交索引大于本地的提交索引，更新本地的提交索引
    if args.LeaderCommit > rf.commitIndex {
        rf.commitIndex = min(args.LeaderCommit, rf.log.lastLog().Index)
        rf.apply()
    }
    reply.Success = true
}
```

2.4 2B 测试

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab2/raft$ go test -run 2B
Test (2B): basic agreement ...
  ... Passed --   0.4  3   20    5778    3
Test (2B): RPC byte count ...
  ... Passed --   0.8  3   50  115838   11
Test (2B): agreement despite follower disconnection ...
  ... Passed --   5.0  3  214   62942    8
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   3.4  5  372   84556    3
Test (2B): concurrent Start()s ...
  ... Passed --   0.6  3   32   10026    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   4.1  3  276   69275    4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  11.5  5 2508 2021365  103
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.2  3  104   34654   12
PASS
ok      6.824/raft      27.997s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab2/raft$
```

图 2-5 2B 的测试结果

经过多次测试，2B 都是可以完美通过的。

3 Part 2C: persistence

3.1 任务思路梳理

这一部分要实现的内容比较少，这一部分主要是持久化 Raft 节点的状态。

如果基于 Raft 的服务器重新启动，它应该从中断处恢复服务。这要求 Raft 保持在重启后仍然存在的持久状态。论文的图 2 提到了哪种状态应该是持久的。

按照 mit6.824 的手册，这部分只需要实现两个函数，并在多个地方调用即可。

思维导图如图 3-1 所示。

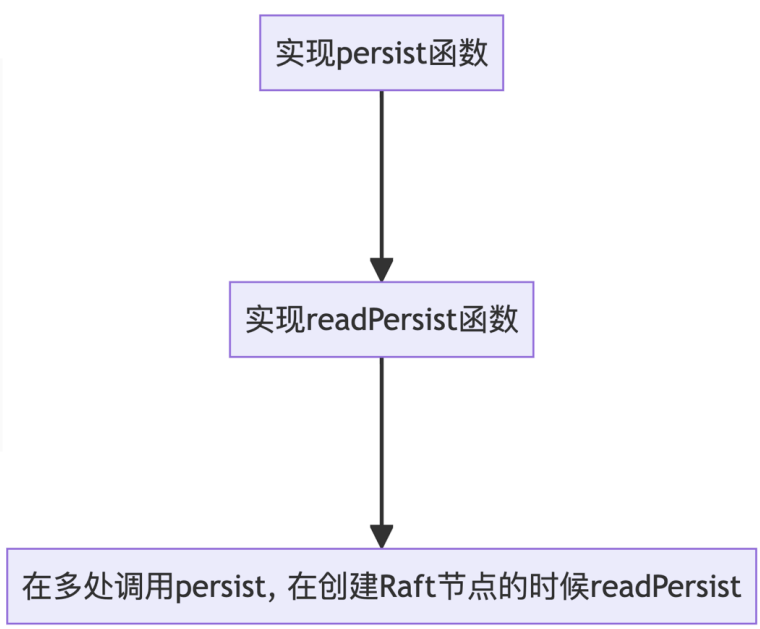


图 3-1 2C 部分实现思路

3.2 任务分析、实现流程和代码实现

3.2.1 实现 persist() 函数

1. 创建 ‘bytes.Buffer‘对象
2. 用 labgob 里面的编码器来处理这个字节流对象
3. 把节点的信息（任期，投票信息，日志）存放到字节流里面
4. 把缓冲区数据转化成字节数组
5. 调用 ‘rf.persister.SaveRaftState‘保存数据

代码如下。

language

```
func (rf *Raft) persist() {
    // 这里是打印日志
    DPrintVerbose("[%v]: STATE: %v", rf.me, rf.log.String())
    // 创建一个新的bytes.Buffer对象，Buffer是一个内存中的字节缓冲区，用于构建要保存的字节流
    w := new(bytes.Buffer)
    // 调用了labgob里面的编码器，这里是初始化一个编码器，然后传入w
    e := labgob.NewEncoder(w)
    // 把当前服务器的任期编码到缓冲区里面
    e.Encode(rf.currentTerm)
    // 投票信息也要编码进去
    e.Encode(rf.votedFor)
    // 日志非常重要，也要编码进去
    e.Encode(rf.log)
    // 这一行将缓冲区中的数据转换为字节数组
    data := w.Bytes()
    // 调用persister写好的函数的就行了
    rf.persister.SaveRaftState(data)
}
```

3.2.2 编写 readPersist 函数

这个函数很简单，读取编码后的信息，然后解码就行了。

最主要的是一些判断条件需要注意。

1. 如果 ‘data‘长度为空就说明没有可以 readPersist 的，返回即可

2. 如果读取的过程中出现任何的读取失败，都是错误的，都要提示错误（Decode 这里返回 nil 说明读取成功）

代码如下。

language

```
func (rf *Raft) readPersist(data []byte) {
    // 这行检查传入的 data 是否为空或长度小于 1
    if data == nil || len(data) < 1 { // bootstrap without any state?
        // 这里进来不一定是有错误，服务首次启动的时候也是要走到这里的
        return
    }
    // 创建一个接收字符流
    r := bytes.NewBuffer(data)
    // 创建一个对应的 解码器
    d := labgob.NewDecoder(r)
    // 创建raft节点信息里面的变量，用来接收
```

```
    var currentTerm int
    var votedFor int
    var logs Log

    // 只有有一部分读取不成功，都是错误的!
  // Decode这里返回nil说明读取成功
    if d.Decode(&currentTerm) != nil || d.Decode(&votedFor) != nil || d.Decode(&logs) != nil {
        log.Fatal("failed to read persist\n")
    } else {
        // 如果读取没有问题就可以完成赋值了
        rf.currentTerm = currentTerm
        rf.votedFor = votedFor
        rf.log = logs
    }
}
```

3.2.3 在 Make 中调用 readPersist 函数

肯定就是在创建节点的时候读取原来的状态（节点因为崩了退出了，重新 Make）

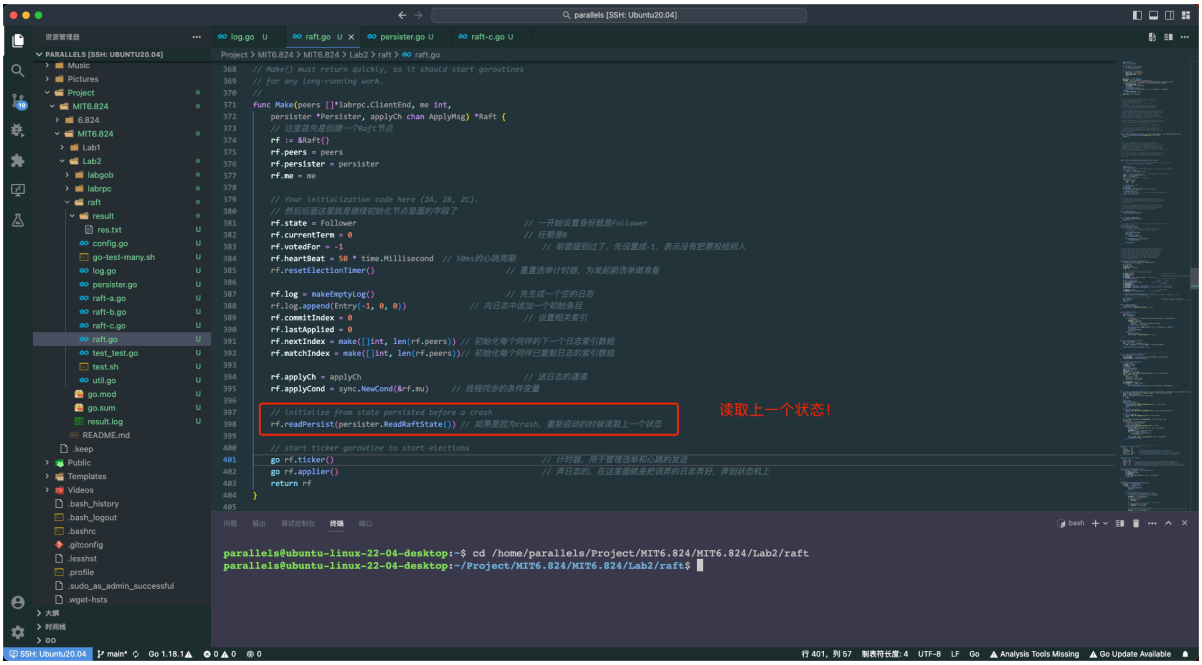


图 3-2 在 Make 中的调用

3.2.4 在多处中调用 persist 函数

要调用 persist 的地方有：

1. leaderElection 中

2. setNewTerm 中
3. AppendEntries 中
4. RequestVote 中
5. Start 中

这些地方需要调用，是因为这些地方可能会更改 raft 结构里面的重要字段，因此需要调用 persist 对状态进行存储。

3.3 2C 测试结果

经过多次实验，代码都可以通过测试。

```
parallels@ubuntu-linux-22-04-desktop:~$ cd /home/parallels/Project/MIT6.824/MIT6.824/Lab2/raft
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab2/raft$ go test -run 2C
Test (2C): basic persistence ...
  ... Passed --   2.9  3  126   36059    6
Test (2C): more persistence ...
  ... Passed --  14.3  5 1588  370598   16
Test (2C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   1.4  3   52   14761    4
Test (2C): Figure 8 ...
  ... Passed --  23.3  5 1604  373118   58
Test (2C): unreliable agreement ...
  ... Passed --   3.3  5 1248  435576  248
Test (2C): Figure 8 (unreliable) ...
  ... Passed --  34.8  5 13664 36146930  286
Test (2C): churn ...
  ... Passed --  16.1  5 18000 166838411 3429
Test (2C): unreliable churn ...
  ... Passed --  16.1  5 5808 9521004 1051
PASS
ok      6.824/raft      112.123s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab2/raft$
```

图 3-3 2C 的测试结果

4 调试过程中遇到的问题和对应的解决

1. **困难：选举过程中的多次超时**

具体问题：在 leaderElection 函数中，可能会遇到多个节点几乎同时成为候选者，导致选举超时。

代码位置：leaderElection 函数，特别是 resetElectionTimer 的调用。

解决方法：确保选举超时时间是随机的，以减少这种情况。您已经在 resetElectionTimer 函数中使用了随机超时，可能需要调整时间范围或检查是否在所有情况下都正确调用了此函数。

2. **困难：日志复制不一致**

具体问题：在 AppendEntries RPC 中，可能因为日志不一致导致附加条目失败。

代码位置：AppendEntries 函数，特别是对 args.PrevLogIndex 和 args.PrevLogTerm 的检查。

解决方法：增强对日志一致性的检查。如果发现不一致，可以更详细地记录哪些条目导致不一致，以便更快地解决问题。

3. **困难：心跳包发送失败**

具体问题：在 appendEntries 函数中，可能会因为网络问题或其他原因导致心跳包发送失败。

代码位置：appendEntries 函数，在发送心跳包的循环中。

解决方法：增加对 sendAppendEntries 调用的异常处理。在网络请求失败时，可以尝试重新发送或记录详细的错误信息。

4. **困难：提交索引更新不正确**

具体问题：在 leaderCommitRule 函数中，可能会出现由于错误的日志复制导致提交索引 (commitIndex) 更新不正确。

代码位置：leaderCommitRule 函数，特别是在更新 commitIndex 的逻辑部分。

解决方法：增加更多的日志输出，特别是在更新 commitIndex 之前和之后，以便跟踪其变化。同时，确保只在多数节点确认后更新 commitIndex。

5. **困难：状态同步问题**

具体问题：在并发场景下，可能会出现状态同步问题，如 currentTerm 和 state 的更新可能不同步。

代码位置：多个函数中对 currentTerm 和 state 的更新，如 RequestVote 和 AppendEntries。

解决方法：使用更细粒度的锁或其他同步机制来确保状态的一致性。检查所有修改这些变量的地方，确保在修改时持有正确的锁。

5 实验总结

在完成 MIT 6.824 Lab2 的过程中，我深入掌握了 Raft 一致性算法的核心原理和实践应用。通过实现领导者选举、日志复制、持久化状态和故障恢复等功能，我不仅增强了对分布式系统的理解，而且提升了我的编程技能和问题解决能力。特别是在处理选举超时和日志复制的一致性问题时，我学会了如何在复杂系统中进行有效的调试和优化。此外，实验过程中的挑战促使我更深入地理解了分布式系统的稳定性和高可用性的重要性。整个实验不仅加深了我对理论的理解，也让我在实践中获得了宝贵的经验，对我的技术成长和职业发展产生了积极影响。