Raft 一致性协议的实现

刘芳新

中山大学 智能科学与技术 21312482

2023 年 12 月 16 日

学号: 21312482
课程: 分布式计算实验
日期：2023 年 12 月 16 日
Email:liufx7@mail2.sysu.edu.cn

目录

1 lab2-A leader election

1.1 任务分析

Raft 算法是一种分布式一致性算法，设计用于确保在分布式系统中的多个节点之间达成一致性。Raft 将整个系统的状态分为任期（term）、领导者（leader）和跟随者（follower）三个状态，并通过领导者选举机制和心跳机制来维持整个系统的一致性。

在 Raft 中，系统中的节点通过领导者选举机制来选出一个领导者。节点首先以跟随者的身份启动，在一段时间内未收到领导者的心跳时，它们会发起一次选举。选举的过程中，节点会向其他节点发送投票请求，其他节点通过比较各自的任期和候选节点的任期来决定是否投票。如果候选节点获得多数票，它将成为新的领导者。

Raft 中的领导者通过定期向其他节点发送心跳来维持其领导地位。这些心跳包告诉其他节点当前的领导者仍然处于活跃状态。如果一个节点在一定时间内没有收到来自领导者的心跳，它就会启动新的选举过程，试图成为新的领导者。

在 Lab 2A 中，我们需要实现 Raft 领导者选举和心跳机制的一部分。具体来说涉及到以下几个方面：

- 实现节点状态的切换：节点在不同的状态（跟随者、候选者、领导者）之间切换，具体取决于收到的消息和定时器的触发。
- 实现选举过程：包括候选者的发起、投票的请求和回复等。
- 实现心跳机制：领导者定期向其他节点发送心跳消息。

1.2 功能设计

1.2.1 完善 Raft 结构体

在 raft 结构体内，我添加了 paper 中 Figure2 中提及到的变量。

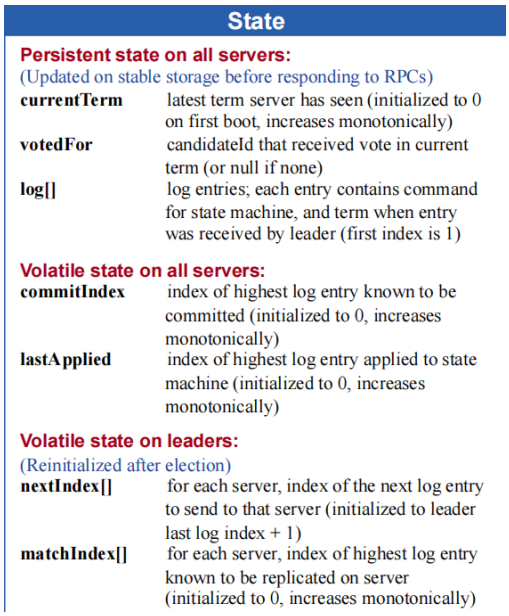


图 1: paper 中 figure2 关于 Raft 结构体的部分

除此之外，我还添加了以下变量：

- state：节点状态 (leader/follower/candidate)
- lastLogIndex：最后一个日志条目的索引
- lastLogTerm：最后一个日志条目的任期
- ElectionStatus：这个变量用于表示发生了什么事件。

  =0：这个变量的初始化

  =1：follower 阶段的计时器超时，应当转变为 candidate 发起 election

  =2：candidate 收集到了足够的票数，应当转变为 leader

  =3：candidae 在投票选举过程中超时了，应当更新 term，发起新的一轮 election

  = -1：表示 raft 实例接受到了消息，需要重置 timer，并且自己的身份已经转变为了 follower。例如：收到来自 leader 的 ae 消息、投票给了其他 raft 实例等。

- mesMutex：这是一个互斥锁，由于 raft 结构体中的变量诸多，这么多变量都只用一个锁的话，我觉得会导致 go routine 之间的性能降低，因此添加了这个互斥锁用于访问 ElectionStatus 这个使用频率很高的变量。
- messageCond：这是一个条件变量，以 mesMutex 作为它的互斥锁。一个 raft 实例在等待 ElectionStatus 的变化来决定后续的操作的时候，可以通过 wait 这个条件变量，当 ElectionStatus 发生改变的时候，就需要对 messageCond 进行广播，对等待 ElectionStatus 的 go routine 即可被唤醒继续执行。举例来说：一个 follower 在开始阶段，ElectionStatus=0，需要等待后续事件，如果发生 timeout，则 ElectionStatus=1; 如果收到投票要求并成功投票，则 ElectionStatus=-1，后续需要重置 timer 等。

- changeElectionStatus 函数：由于 mesMutex 这是一个针对 ElectionStatus 变量的互斥锁，每次修改这个变量都需要对这个锁操作，因此将修改操作封装到了一个函数中。

注意：在同时查看 ElectionStatus 变量和调用 changeElectionStatus 函数时存在潜在的死锁风险。这是因为在查看 ElectionStatus 变量之前需要获取锁，如果在没有释放锁的情况下直接调用 changeElectionStatus，由于 changeElectionStatus 函数也需要获取锁，可能会导致无法获得所需的锁，从而产生死锁。

1.2.2 完善 GetState 函数

GetState 返回当前节点的任期和该节点是否认为自己是领导者

1.2.3 完善 RequestVote 功能

先按照 paper 中的 figure2 实现 RequestVoteArgs 和 RequestVoteReply 结构体。

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)

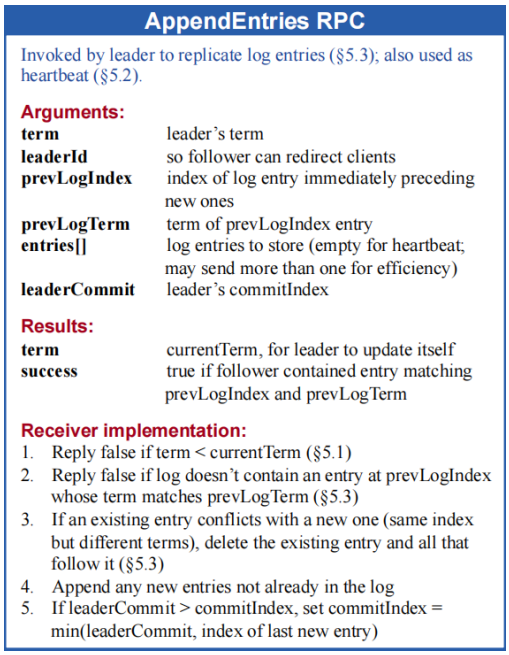
图 2: paper 中 figure2 关于 RequestVote 的部分

然后按照以下规则完善 RequestVote 函数：

- 如果请求的任期小于当前任期，则拒绝投票
- 如果请求的最后一个日志条目的任期小于节点的最后一个日志条目的任期，则拒绝投票
- 如果节点已经投过票给其他节点，则拒绝投票
- 如果请求的任期大于当前任期，则更新当前任期，同意投票，并转变为 follower 状态

1.2.4 实现 AppendEntries 功能

Leader 会周期性地给其他节点发送 ae 消息，这个 RPC 调用首先需要实现参数和返回值的结构体设计，结构体的设计直接参照 paper 即可。同时仿照 sendRequestVote 写了 sendAppendEntries 函数。



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)

图 3: paper 中 figure2 关于 AppendEntries 的部分

设计 AppendEntries 函数，当节点收到 AE 消息后，执行以下步骤：

1. 检查 Leader 的 Term 是否更大。如果 Leader 的 Term 不大于当前节点的 Term，则直接返回 false 并携带当前节点的 Term。如果 Leader 的 Term 较大，则进行下一步骤。

2. 认可新的 Leader，并将节点的 Term 和 Leader 的 Term 保持一致。节点将自身状态重置为 Follower 的初始状态，重新启动选举计时器。随后，通过 messageCond.Broadcast() 通知节点及时处理后续消息。

1.2.5 完善 ticker 函数

在 ticker() 函数中，该函数是大多数节点一直在运行的。以下是该函数的任务流程：

1. 作为 Follower，初始化 ElectionStatus 为 0，并设置计时器以检测超时。如果收到 AppendEntries 或者投票给了其他节点，ElectionStatus 将被设定为-1。如果计时器时间到了，发现 ElectionStatus 不等于-1，说明在这段时间内，节点没有收到 AppendEntries 或者投票，那么就是发生了超时，将 ElectionStatus 设定为 1。否则，节点在这段时间内发生了某些活动，已经重新发起了一个新的计时器，这个计时器没有实际意义，因此无需将 ElectionStatus 设定为 1。

2. 如果发现 ElectionStatus 为 1，则将状态转变为 Candidate，并向其他节点发起 RequestVote RPC 调用，然后进入步骤 3。

3. 如果发现 ElectionStatus 为-1，返回到初始步骤 1。

4. 在投票阶段，再次启动一个计时器。如果计时器时间到了，发现 ElectionStatus 不等于-1，说明发生了超时，将 ElectionStatus 设定为 3，表示投票阶段超时。

5. 在投票阶段，ticker() 函数再启动两个新的协程：一个用于处理投票的协程 vThread，另一个用于投票阶段的计时器协程 tThread。然后进入阻塞状态，等待消息。

负责投票的协程 vThread：通过创建新的协程向其他 Raft 节点发起投票 RPC 调用，并创建一个条件变量 voteCond。voteCond.Wait() 用于等待每个投票 RPC 调用的成功，并在成功后通过 voteCond.Broadcast 唤醒 vThread。

投票阶段计时器协程 tThread：类似于初始阶段的计时器，设置一个随机时间。如果到时了，尝试将 ElectionStatus 设置为 3。如果 ElectionStatus 为-1 或 4，表示节点已经不是 Candidate，投票阶段已经结束，直接退出。否则，将 ElectionStatus 设置为 3，并通知 ticker() 协程重新启动一次投票。

6. ticker() 被唤醒后，根据 ElectionStatus 的不同值进行不同的处理：

   如果 ElectionStatus 为 2，表示节点已经是 Leader。此时，启动一个新的协程周期性地发送心跳，然后阻塞本协程直到节点不再是 Leader。然后返回到初始状态的步骤 1。

   如果 ElectionStatus 为 3，表示投票阶段超时，需要重新发起新的投票。

   如果 ElectionStatus 为 4 或-1，表示节点不再是 Candidate，需要回到初始状态的步骤 1。

1.2.6 实现 heartbeat 函数

heartBeats 函数由 Leader 调用，用于周期性向其他 Raft 节点发送 AppendEntries 消息。

根据实验规定，心跳的周期不得超过每秒 10 次。因此，我设置了每秒 10 次的频率，同样通过使用 time.Sleep 来实现。

由于 Lab2A 仅需要实现选举功能，无需考虑日志的事务。因此，该函数的逻辑非常简单。在每个周期内，向其他的 Raft 节点发送 AppendEntries 消息。注意，这个发送 AppendEntries 消息是一个 RPC 调用，因此应当新开一个协程来进行发送，以避免阻塞 heartBeats 协程。

如果在发送 AppendEntries 消息后，收到的返回消息中发现有某个 Raft 节点的 Term 值比自身还高，说明自身已经过时。此时，将自己变为 Follower，并更新自身的 Term 值，同时将 voteFor 初始化为-1。

1.2.7 完善 Make 函数

Make 函数就是完成 raft 节点的初始化即可。

1.3 代码实现

- 完善 Raft 结构体

```go
// A Go object implementing a single Raft peer.
type Raft struct {
    mu        sync.Mutex          // Lock to protect shared access to this peer's state 用于保护对节点状态的共享访问的锁
    peers     []*labrpc.ClientEnd // RPC end points of all peers                        所有节点的 RPC 终端点
    persister *Persister          // Object to hold this peer's persisted state         用于保存节点持久状态的对象
    me        int                 // this peer's index into peers[] //                  节点在 peers[] 中的索引
    dead      int32               // set by Kill()                                      由 Kill() 设置

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

    // persistent state
    peerNum           int   // 节点数量
    currentTerm       int   // 当前任期
    voteFor           int   // 投票给哪个节点
    log               []LogEntry // 日志条目

    // volatile state
    commitIndex int // 已经被提交的日志的最大索引
    lastApplied int // 已经被应用到状态机的最大索引
    state       string // 节点状态
    lastLogIndex int // 最后一个日志条目的索引
    lastLogTerm  int // 最后一个日志条目的任期


    // Candidate 使用条件变量同步选举
    mesMutex    sync.Mutex // 用于锁定变量 ElectionStatus
    messageCond *sync.Cond // 条件变量

    // ElectionStatus == 1 -> 开始选举
    // ElectionStatus == -1 -> 保持不动
    // ElectionStatus == 2 -> 成为领导者
    // ElectionStatus == 3 -> 选举超时
    ElectionStatus int

    // volatile state on leaders
    nextIndex  []int // 用于每个节点的下一个日志条目的索引
    matchIndex []int // 用于每个节点已匹配的最高日志条目的索引
}
```

图 4: Raft 结构体

```go
// changeElectionStatus 用于更改节点的ElectionStatus
func (rf *Raft) changeElectionStatus(num int) {
    rf.mesMutex.Lock()
    rf.ElectionStatus = num
    rf.mesMutex.Unlock()
}
```

图 5: changeElectionStatus 函数

- 完善 GetState 函数

```go
// return currentTerm and whether this server
// believes it is the leader.
// GetState 返回当前节点的任期和该节点是否认为自己是领导者。
func (rf *Raft) GetState() (int, bool) {

    var term int        // 当前节点的任期
    var isleader bool   // 当前节点是否认为自己是领导者
    // Your code here (2A).
    // 使用互斥锁保护共享状态，确保在读取状态期间不会发生竞态条件
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if rf.state == "leader" {
        isleader = true
    }
    term = rf.currentTerm
    return term, isleader
}
```

图 6: GstState 函数

- 完善 Request Vote 功能

```go
// RequestVoteArgs 结构定义了 RequestVote RPC 请求的参数。
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    Term int // 当前候选者的任期
    CandidateId int // 请求投票的候选者的ID
    LastLogIndex int // 候选者的最后一个日志条目的索引
    LastLogTerm int // 候选者的最后一个日志条目的任期
}
```

图 7: RequestVoteArgs 结构体

```go
// RequestVoteReply 结构定义了 RequestVote RPC 响应的参数。
type RequestVoteReply struct {
    Term int // 对候选者投票的节点的当前任期
    VoteGrand bool // 是否授予投票，如果为 true，则表示投票成功
}
```

图 8: RequestVoteReply 结构体

```go
// example RequestVote RPC handler.
// RequestVote 处理来自其他节点的 RequestVote RPC 请求。
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    // Your code here (2A, 2B).

    // 使用互斥锁保护对节点状态的共享访问，确保在处理请求期间不会发生竞态条件
    rf.mu.Lock()
    defer rf.mu.Unlock()

    // 设置响应中的当前任期为节点的当前任期
    reply.Term = rf.currentTerm

    // 如果请求的任期小于当前任期，则拒绝投票
    if args.Term < rf.currentTerm {
        reply.VoteGrand = false
    }

    // 如果请求的最后一个日志条目的任期小于节点的最后一个日志条目的任期，则拒绝投票
    if args.LastLogTerm < rf.lastLogTerm {
        // 如果请求的任期大于当前任期，则更新当前任期，并转变为 follower 状态
        if args.Term > rf.currentTerm {
            rf.currentTerm = args.Term
            rf.voteFor = -1
            rf.changeElectionStatus(-1)
            rf.state = "follower"
            rf.messageCond.Broadcast()
        }
        reply.VoteGrand = false
    } else if args.LastLogTerm > rf.lastLogTerm {
        // 如果请求的最后一个日志条目的任期大于节点的最后一个日志条目的任期，则授予投票
        if rf.currentTerm == args.Term && rf.voteFor != -1 {
            // 如果节点已经投过票给其他节点，则拒绝投票
            reply.VoteGrand = false
        } else {
            // 否则，更新当前任期，投票给请求的候选者，并转变为 follower 状态
            if rf.currentTerm < args.Term {
                rf.currentTerm = args.Term
            }
            rf.voteFor = args.CandidateId
            reply.VoteGrand = true
            rf.changeElectionStatus(-1)
            rf.state = "follower"
            rf.messageCond.Broadcast()
        }
    } else {
        // 如果最后一个日志条目的任期相等，则需要进一步比较索引
        if args.LastLogIndex >= rf.lastLogIndex {
            if rf.currentTerm == args.Term && rf.voteFor != -1 {
                // 如果节点已经投过票给其他节点，则拒绝投票
                reply.VoteGrand = false
            } else {
                // 否则，更新当前任期，投票给请求的候选者，并转变为 follower 状态
                if rf.currentTerm < args.Term {
                    rf.currentTerm = args.Term
                }
                rf.voteFor = args.CandidateId
                reply.VoteGrand = true
                rf.changeElectionStatus(-1)
                rf.state = "follower"
                rf.messageCond.Broadcast()
            }
        } else {
            // 如果最后一个日志条目的索引小于节点的最后一个日志条目的索引，则拒绝投票
            reply.VoteGrand = false
            // 如果请求的任期大于当前任期，则更新当前任期，并转变为 follower 状态
            if args.Term > rf.currentTerm {
                rf.currentTerm = args.Term
                rf.voteFor = -1
                rf.changeElectionStatus(-1)
                rf.state = "follower"
                rf.messageCond.Broadcast()
            }
        }
    }
}
```

图 9: RequestVote 函数

- 实现 AppendEntries 功能

```go
// AppendEntriesArgs 结构定义了 AppendEntries RPC 请求的参数。
type AppendEntriesArgs struct {
    Term int // 领导者的任期
    LeaderId int // 领导者的ID
    PrevLogIndex int // 前一个日志条目的索引
    PrevLogTerm int // 前一个日志条目的任期
    Entries []LogEntry // 要追加到日志中的日志条目
    LeaderCommit int // 领导者已经提交的日志的最大索引
}
```

图 10: AppendEntriesArgs 结构体

```go
// AppendEntriesReply 结构定义了 AppendEntries RPC 响应的参数。
type AppendEntriesReply struct {
    Term int // 节点的当前任期
    ConflictIndex int // 冲突的日志条目的索引，如果有冲突
    Success bool // 如果追加成功，则为 true
}
```

图 11: AppendEntriesReply 结构体

```go
// AppendEntries 处理来自领导者的 AppendEntries RPC 请求
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    reply.Term = rf.currentTerm // 设置响应中的当前任期为节点的当前任期
    if args.Term < rf.currentTerm { // 如果请求的任期小于当前任期，则拒绝追加日志条目
        reply.Success = false
    } else {
        reply.Success = true
        // 更新节点的当前任期，并转变为 follower 状态
        rf.currentTerm = args.Term
        rf.state = "follower"
        rf.changeElectionStatus(-1)
        rf.messageCond.Broadcast()
    }
}
```

图 12: AppendEntries 函数

- 完善 ticker 函数

```go
// 该函数在一个独立的协程中运行，定期检查是否需要启动选举，以及通过随机睡眠来模拟计时器。
func (rf *Raft) ticker() {
    rf.mu.Lock() // 加锁，读取当前节点的当前任期
    var recordTerm = rf.currentTerm
    rf.mu.Unlock()
    // 初始化互斥锁和计数器，用于确保唯一性
    var timeMutex sync.Mutex
    var currentTermTimes = 0
    // 循环执行计时器的逻辑
    for rf.killed() == false {
        // Your code here to check if a leader election should
        // be started and to randomize sleeping time using
        // time.Sleep().
        // 在这里检查是否应该启动选举，并通过 time.Sleep() 随机睡眠一段时间
        rf.mu.Lock()
        timeMutex.Lock()
        if recordTerm != rf.currentTerm {
            recordTerm = rf.currentTerm
            currentTermTimes = 0
        }
        currentTermTimes++

        rand.Seed(time.Now().UnixNano()) // 生成随机的睡眠时间
        sleepTime := rand.Intn(TimeOutInterval) + TimeOutInterval
        rf.changeElectionStatus(0) // 初始化 ElectionStatus 变量，用于测试是否超时
        go func(currentTerm int, cTermTimes int) { // 启动协程，用于检测是否超时
            time.Sleep(time.Duration(sleepTime) * time.Millisecond)
            rf.mu.Lock()
            defer rf.mu.Unlock()
            rf.mesMutex.Lock()
            defer rf.mesMutex.Unlock()
            timeMutex.Lock()
            defer timeMutex.Unlock()
            // 如果 ElectionStatus 不为 -1 且当前任期和计数器未更改，则将 ElectionStatus 设置为 1 并广播条件变量
            if rf.ElectionStatus != -1 && currentTerm == rf.currentTerm && cTermTimes == currentTermTimes {
                rf.ElectionStatus = 1
                rf.messageCond.Broadcast()
            }
        }(rf.currentTerm, currentTermTimes)
        // 如果接收到附加日志或请求投票消息，则 ElectionStatus 变为 -1；如果超时，ElectionStatus 变为 1
        rf.mu.Unlock()
        timeMutex.Unlock()
        rf.mesMutex.Lock()
        rf.messageCond.Wait() // 等待 ElectionStatus 变为 -1 或 1
        for rf.ElectionStatus == 2 || rf.ElectionStatus == 3 {
            rf.messageCond.Wait()
        }
        // 如果 ElectionStatus 为 -1，则表示接收到了有效的消息并重置了计时器
        if rf.ElectionStatus == -1 {
            rf.mesMutex.Unlock()
            continue
        } else if rf.ElectionStatus == 1 {
            // 开始选举
            rf.mesMutex.Unlock()
            // 启动两个线程
            // 一个线程是计时器，检查选举是否超时
            // 另一个线程负责启动选举
        voteLoop:
            for rf.killed() == false {
                // 如果超时，发起新的选举
                // ch 用于确定选举是否成功或超时
                rf.mu.Lock()
                rf.state = "candidate"
                rf.currentTerm++
                rf.voteFor = rf.me
                var voteArgs = &RequestVoteArgs{}
                voteArgs.Term = rf.currentTerm
                voteArgs.LastLogTerm = rf.lastLogTerm
                voteArgs.LastLogIndex = rf.lastLogIndex
                voteArgs.CandidateId = rf.me

                var voteMutex sync.Mutex
                var voteCond = sync.NewCond(&voteMutex)

                rf.changeElectionStatus(0)
                // 启动协程发起选举
```

```
        go func(currentTerm int, currentTermTimes int, pNum int) {
            // 启动选举
            var grandNum = 1
            var refuseNum = 0
            var term = currentTerm
            var times = currentTermTimes
            var peerNum = pNum
            // 遍历所有节点，向它们发送请求投票消息
            for index := 0; index < peerNum; index++ {
                if index == rf.me {
                    continue
                }
                go func(index int) {
                    var reply = &RequestVoteReply{}
                    if rf.sendRequestVote(index, voteArgs, reply) {
                        rf.mu.Lock()
                        if reply.Term > rf.currentTerm {
                            // 如果回复的任期大于当前任期，则更新节点状态为 follower
                            if term == rf.currentTerm {
                                rf.currentTerm = reply.Term
                                rf.state = "follower"
                                // 发现更高的任期并更新节点状态
                                rf.mu.Unlock()
                                rf.changeElectionStatus(4)
                                rf.messageCond.Broadcast()
                                voteCond.Broadcast()
                                return
                            } else {
                                rf.currentTerm = reply.Term
                                rf.state = "follower"
                                // 标记陈旧的投票请求发现更高的任期并更新节点状态
                                rf.mu.Unlock()
                                rf.changeElectionStatus(-1)
                                rf.messageCond.Broadcast()
                                return
                            }
                        }
                        rf.mu.Unlock()
                        if reply.VoteGrand { // 处理投票回复
                            // 接收到其他节点的投票
                            voteCond.L.Lock()
                            grandNum++
                            voteCond.L.Unlock()
                            voteCond.Broadcast()
                        } else {
                            // 接收到其他节点的拒绝
                            voteCond.L.Lock()
                            refuseNum++
                            voteCond.L.Unlock()
                            voteCond.Broadcast()
                        }
                    }
                }(index)
            }
            // 等待投票结果
            var voteSuccess = false
            for {
                voteCond.L.Lock()
                voteCond.Wait()
                rf.mu.Lock()
                if rf.state == "follower" {
                    rf.mu.Unlock()
                    voteCond.L.Unlock()
                    return
                }
                // 获取节点的总数
                var peerNum = rf.peerNum
                rf.mu.Unlock()
                // 判断是否成功获取超过半数的投票
                if grandNum >= (peerNum/2)+1 {
                    voteSuccess = true
                    // 成功获取大多数的投票
                    voteCond.L.Unlock()
                    break
                } else if refuseNum+grandNum == peerNum || refuseNum >= (peerNum/2)+1 {
                    // 没有获取到大多数的投票
```

```
                    voteCond.L.Unlock()
                    break
                }
                voteCond.L.Unlock()
            }
            // 如果选举成功，更新节点状态为 leader
            if voteSuccess {
                rf.mu.Lock()
                defer rf.mu.Unlock()
                rf.mesMutex.Lock()
                defer rf.mesMutex.Unlock()
                timeMutex.Lock()
                defer timeMutex.Unlock()
                // 如果 ElectionStatus 不为 -1 且不为 4，且当前任期和计数器未更改，则将 ElectionStatus 设置为 2 并广播条件变量
                if rf.ElectionStatus != -1 && rf.ElectionStatus != 4 && term == rf.currentTerm && times == currentTermTimes {
                    // 选举成功表示节点成功成为领导者
                    rf.ElectionStatus = 2
                    rf.state = "leader"
                    rf.messageCond.Broadcast()
                }
                // 如果一个节点收集了正好半数的投票，它将等待直到超时并启动下一次选举
            }
        }(rf.currentTerm, currentTermTimes, rf.peerNum)
        // 启动协程，用于等待选举超时
        go func(currentTerm int, cTermTimes int) {
            rand.Seed(time.Now().UnixNano())
            sleepTime := rand.Intn(TimeOutInterval) + TimeOutInterval
            // 等待选举超时
            time.Sleep(time.Duration(sleepTime) * time.Millisecond)
            rf.mu.Lock()
            defer rf.mu.Unlock()
            rf.mesMutex.Lock()
            defer rf.mesMutex.Unlock()
            timeMutex.Lock()
            defer timeMutex.Unlock()
            // 如果 ElectionStatus 不为 -1 且不为 4，且当前任期和计数器未更改，则将 ElectionStatus 设置为 3 并广播条件变量
            if rf.ElectionStatus != -1 &&
            rf.ElectionStatus != 4 &&
            currentTerm == rf.currentTerm &&
            cTermTimes == currentTermTimes {
                rf.ElectionStatus = 3
                rf.messageCond.Broadcast()
            }
        }(rf.currentTerm, currentTermTimes)
        // 解锁，等待条件变量
        rf.mu.Unlock()
        rf.mesMutex.Lock()
        rf.messageCond.Wait()
        // 等待 ElectionStatus 变为 1 或 0
        for rf.ElectionStatus == 1 || rf.ElectionStatus == 0 {
            rf.messageCond.Wait()
        }
        // 根据 ElectionStatus 的值采取不同的行动
        switch rf.ElectionStatus {
        case 2: // 如果 ElectionStatus 为 2，表示节点成为领导者
            rf.mesMutex.Unlock() // 解锁消息互斥锁，以便让 leader 发送心跳
            // 启动心跳协程
            var leaderch = make(chan int)
            go rf.heartBeats(leaderch)
            <-leaderch
            break voteLoop // 退出选举循环 voteLoop
        case 3: // 如果 ElectionStatus 为 3，表示选举超时
            rf.mesMutex.Unlock() // 解锁消息互斥锁，重新开始选举
            continue
        case 4:
            fallthrough
        case -1:
            // 如果 ElectionStatus 为 4 或 -1，表示停止选举
            // 解锁消息互斥锁，并退出选举循环 voteLoop
            rf.mesMutex.Unlock()
            break voteLoop
        }
    }
}
}
```

图 13: ticker 函数

- 实现 heartBeats 函数

```go
// heartBeats 用于定期发送心跳消息以维持领导者地位。
// 这个函数在一个独立的协程中运行，定期向其他节点发送心跳消息。
// 参数 ch 是一个通道，用于在函数执行完毕时发送信号。
func (rf *Raft) heartBeats(ch chan int) {
    // 加锁，读取当前节点的当前任期
    rf.mu.Lock()
    var term = rf.currentTerm
    rf.mu.Unlock()
    // 循环执行心跳消息的发送
    for rf.killed() == false {
        // 获取当前节点的状态和是否为领导者
        _, isLeader := rf.GetState()
        if !isLeader {
            // 如果当前节点不是领导者，则退出循环，停止发送心跳消息
            break
        }
        // 加锁，读取当前节点的信息
        rf.mu.Lock()
        var peerNum = rf.peerNum
        var args AppendEntriesArgs
        args.LeaderId = rf.me
        args.Term = rf.currentTerm
        rf.mu.Unlock()
        // 遍历所有节点，向它们发送心跳消息
        for index := 0; index < peerNum; index++ {
            if index == rf.me {
                continue
            }
            // 启动协程发送心跳消息
            go func(index int) {
                // 初始化 AppendEntries 回复
                var reply AppendEntriesReply
                // 发送心跳消息
                if rf.sendAppendEntries(index, &args, &reply) {
                    // 加锁，处理心跳回复
                    rf.mu.Lock()
                    defer rf.mu.Unlock()
                    // 检查回复的任期是否大于当前节点的任期
                    if reply.Term > term {
                        // 如果回复的任期大于当前节点的任期，更新当前节点的状态为 follower
                        if reply.Term > rf.currentTerm {
                            rf.state = "follower"
                            rf.currentTerm = reply.Term
                            // 更新投票状态，重置投票
                            rf.voteFor = -1
                        }
                    }
                }
            }(index)
        }
        time.Sleep(time.Duration(HeartBeatInterval) * time.Millisecond)
    }
    ch <- 1
}
```

图 14: heartBeats 函数

- 完善 Make 函数

```go
// Make 用于创建并初始化 Raft 节点的实例。
// 参数 peers 表示所有 Raft 节点的客户端端点，
// 参数 me 表示当前节点的标识，
// 参数 persister 为持久化存储对象，用于保存 Raft 节点状态和日志条目，
// 参数 applyCh 为应用层传入的通道，用于接收已提交的日志条目。
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{} // 创建一个 Raft 实例 rf
    rf.peers = peers
    rf.persister = persister
    rf.me = me

    // Your initialization code here (2A, 2B, 2C).
    rf.messageCond = sync.NewCond(&rf.mesMutex) // 初始化消息条件变量
    // rf.voteCond = sync.NewCond(&rf.mu)
    rf.peerNum = len(peers)
    rf.voteFor = -1

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState()) // 从持久化存储中恢复节点状态和日志

    // start ticker goroutine to start elections
    go rf.ticker() // 启动 ticker 协程，定期检查是否需要发起选举

    return rf
}
```

图 15: Make 函数

1.4 测试结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2A
Test (2A): initial election ...
  ... Passed --   3.1  3   60   17132    0
Test (2A): election after network failure ...
  ... Passed --   4.4  3  128   26424    0
Test (2A): multiple elections ...
  ... Passed --   5.7  7  624  121864    0
PASS
ok      6.824/raft      13.182s
```

图 16: 2A 运行结果

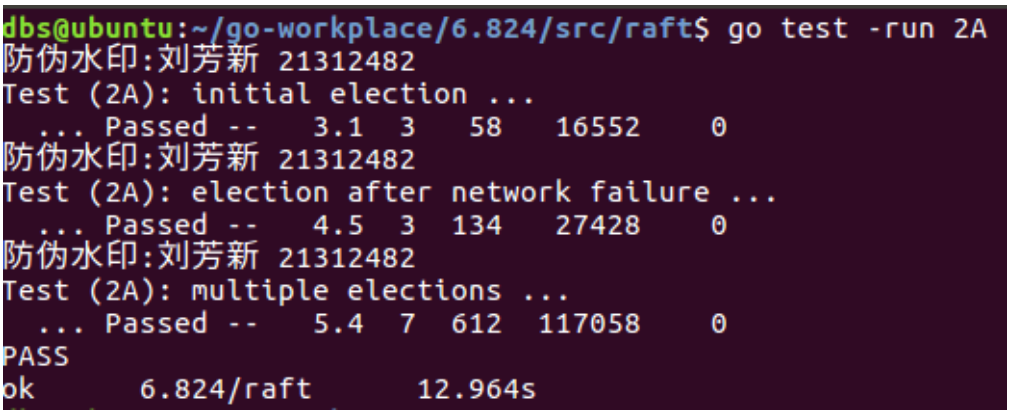


图 17: 2A 运行水印

1.5 个人总结

在进行 RPC 调用时切勿持有锁，因为持有锁可能导致 RPC 调用期间的阻塞，从而阻塞其他活动。释放锁是至关重要的，因为在 RPC 调用结束后，可能发生了大量的事件，而调用期间的锁已经不再具有实际意义。举个例子：假设在 term=5 时发起了一个 RPC 调用来收集选票，当 RPC 返回并收集到足够的选票时，持有锁的节点检查状态时发现 term=6。此时不能将节点变为 Leader，因为此 RPC 的结果是让该节点成为 term=5 的 Leader，而不是 term=6 的 Leader。

在进行 RPC 调用时，最好将其放在一个新开的协程中，以避免 RPC 的阻塞影响到协程的后续工作。例如，在投票阶段，候选者应该同时向其他 Raft 节点发起投票 RPC。为了实现同时发起，每次投票 RPC 都应该由一个新的协程负责处理，否则每向一个节点发起投票 RPC，就必须等待该节点的响应后才能向下一个节点发起投票 RPC。这种并发的设计可以提高系统性能，确保不会因为等待某个 RPC 调用而阻塞其他并发操作。

2 lab2-B log

2.1 任务分析

Lab 2B 旨在实现 Raft 日志的复制与一致性，是一个相对核心且具有一定难度的部分。主要涉及到日志复制的过程，包括日志 RPC 的交互、写入过程以及错误纠正的处理。

- 日志复制：在 Raft 中，每个节点都维护一份日志，用于记录系统状态的变化。每个日志条目包含一个命令，代表对状态机的一次更新。节点通过相互之间的日志复制来确保各自的状态机保持一致。
- 一致性：日志的复制需要确保整个系统中所有节点上的日志都保持一致。领导者负责向其他节点发送日志条目，以确保它们的日志保持一致。一旦大多数节点确认接收了一个条目，它就被认为是已提交的，可以应用到节点的状态机中。
- 日志条目的传输和确认：Lab 2B 要求实现日志条目的传输和确认机制。领导者向其他节点发送日志条目，其他节点需要确认接收。这可能涉及到网络通信、消息传递和节点状态的管理。

- 提交和应用：一旦一个日志条目被大多数节点确认接收，它就被视为已提交。在此之后，领导者会通知其他节点将已提交的日志应用到它们的状态机中，以保持整个系统的状态一致。

通过实现这些关键功能，Lab 2B 旨在确保 Raft 系统的日志同步、一致性和正确性。这涉及到处理复杂的分布式系统场景，对节点之间的通信和状态管理有着高要求。

2.2 功能设计

2.2.1 实现 LogEntry 结构体

在进行 log 结构体的设计时，需要确保能够保存 Command，同时记录该日志对应的 index 和 term。并且，为了灵活处理不同数据类型的 Command，可以使用 interface{}，类似于 ApplyMsg 结构体中对 Command 的设计。

2.2.2 实现 Start 函数

`Start()` 作为一个接口提供给外界，用于向 Raft 节点发送指令并请求其执行。下面是该函数的执行流程：

1. 函数收到一个命令时，首先判断自身是否为 Leader。若不是，则返回；若是，则继续执行下一步。
2. Leader 收到命令后,需要将该命令保存到本地的日志中,并更新 `matchIndex`、`nextIndex`、`lastLogIndex`、`lastLogTerm` 等变量。
3. 启动多个 goroutine 并发地向其他 Follower 发起日志复制请求。在每个 goroutine 中，应注意尽量只使用当前协程的局部变量。如果需要使用共享变量，应使用局部变量保存此刻的共享变量值。例如，在负责日志复制的 goroutine 中，将 `rf.nextIndex[id]` 这个共享变量的值赋给了局部变量 `nextIndex`。如果需要获取发起 goroutine 时的 `rf.currentTerm` 值，只需查看 `args.Term` 的值即可。
4. 负责 Follower 编号为 n 的节点的日志复制 goroutine 中，首先需要检查编号为 n 的 Raft 节点的 `nextIndex`。将 `nextIndex` 和 `index` 进行比较,其中 `index` 为最新的日志的索引值。如果两者存在差值，说明 Leader 和编号为 n 的 Raft 节点之间缺失了不止一个日志值，需要将 `nextIndex+1 index` 之间的日志也打包到日志数组中，一并发送给 Follower。随后，Leader 将 `nextIndex` 指定的索引号的日志包添加到日志数组中。
5. Leader 继续将 `nextIndex` 位置的日志打包到日志数组中，然后通过 `sendAppendEntries` RPC 调用将日志数组发送给 Follower。如果该 RPC 调用返回值为 `true`，表明日志同步成功，随后更新 `matchIndex[id]` 即可。在更新时，每个 goroutine 需要注意自身是否已经过时。如果一个 goroutine 想要将 `matchIndex[id]` 更新为 n，但是发现 `matchIndex[id]` 的数值大于 n，则表明该 goroutine 已经过时，不再更新 `matchIndex[id]` 的值。

若该 RPC 调用返回值为 `false`，则需要分情况进行处理：

- 如果 `reply.Term > args.Term`，表明该 goroutine 已经过时，需要停止该 goroutine 并检查 Raft 节点是否及时更新了 `Term` 值。若没有，则将 `rf.currentTerm` 更新为 `reply.Term`。

- 如果 `reply.Term <= args.Term`，表明 Follower 发现 `preLogTerm` 和 `preLogIndex` 不匹配，需要 Leader 发送更早的日志。这种情况下，Leader 可以逐步一个个地将前面的日志加入，并与 Follower 进行确认是否匹配。然而，一个个添加日志并询问 Follower 效率较低。应该利用 `reply` 中 Follower 的 `term` 值来加速定位，因为 Follower 中的日志的 `term` 值一定都不会大于 Follower 的 `term` 值。因此，那些 `term > follower.term` 的日志必然是 Follower 缺失的。随后尝试一个个添加更早的日志，直到 Follower 返回 `true`，表示找到了两者日志相同的地方，并将后续的日志都同步上了。

Leader 完成了对一个 Follower 的同步后，进行 `matchIndex` 检查，确定是否有新的日志已同步到大多数节点上。该检查通过收集各个节点的 `matchIndex`，对 `matchIndex` 出现的次数进行计数，形成一个键值对数组，其中键为 `matchIndex`，值为达到此 `matchIndex` 的节点数量。该数组按键从高到低排序，然后遍历数组进行累加值，直到值累加的值刚好超过 Raft 节点数的一半，此时的键值即为此刻半数以上节点都已同步的最新日志的索引值。

将这个索引值与 `commitIndex` 进行比较，若大于则表明有新的日志同步到半数以上节点，可以进行提交。随后更新 `commitIndex` 值，并将提交的指令发送到 `applyCh` 管道中，同时更新 `lastApplied` 变量的值。

2.2.3 继续完善 AppendEntries 函数

由于在 Lab2B 中，AE RPC 调用不再仅仅是心跳（heartbeat），因此，我在这里将日志复制（log replication）和心跳两者进行了区分处理。在心跳时，调用 `AppendEntries` 时传递的参数中是没有日志数组的；而在进行日志复制时，必然会发送带有日志数组的参数。因此，我将此作为区分点。

以下是这个函数的处理流程：

1. Term 的比较以及对应的处理，在 Lab 2A 中已经讲述，此处不赘述。
2. 若 `args.Entries` 数组为 `nil`，那么表明为心跳调用；反之，表明为日志复制调用。

心跳调用的 *AppendEntries* 时：

如果 `args.LeaderCommit > rf.commitIndex`，则更新 `rf.commitIndex = args.LeaderCommit`，并进入步骤 2。反之返回 `true`。

更新了 `rf.commitIndex` 表明有新的日志可以提交并执行了。随后，将那些日志发送到 `applyCh` 管道中，并更新 `lastApplied` 变量的值。

日志复制调用的 *AppendEntries* 时：

日志同步应该从双方日志一致的索引点开始，因此这里着重点在于反馈给 Leader 一致的索引在哪里。当然，里面还有很多特殊情况需要考虑。

1. Follower 需要检查发送来的 AE 包是否满足本节点的缺失情况，也就是检查 `args.PreLogIndex` 和 `rf.lastLogIndex`。
2. 如果 `args.PreLogIndex > rf.lastLogIndex` 表明肯定不满足本节点的缺失情况，返回 `false`；反之进入步骤 3。
3. 来到这儿表明 `args.PreLogIndex <= rf.lastLogIndex`，这依旧不能确定这个 AE 包是否满足本节点的缺失情况，可能本节点存在部分无效的日志。例如，一个 Leader 加了很多日志，但是断

开了连接，这些新加的日志都没有来得及发送给任何一个节点。后续重连回来就成为 Follower 了，那么这些独自拥有的日志就是无效日志。

4. Raft 算法的特点之一是，如果两个日志的索引和任期都是一样的，那么这两个日志必然是一致的。因此，Follower 仅需判断 `args.PrevLogTerm == rf.log[args.PreLogIndex]` 是否成立。如果成立，表明，在 `args.PreLogIndex` 这里以及之前的日志都是同步的了。否则，就是 `args.PreLogIndex < rf.lastLogIndex`，但是这个索引值为 `args.PreLogIndex` 处的日志，两个节点的日志依旧不一致，需要返回 `false`，让 Leader 继续往前找到一致的索引。

   注意：Follower 发现发来的这个 AE 包中的日志的 `PrevLogIndex` 和 `PrevLogTerm` 都是符合条件的，但是依旧不一定就要同步上来。需要注意部分 AE 是过时的，例如，1 号 AE 包负责将 Follower 同步到索引 n 处的日志；2 号 AE 包负责而将 Follower 同步到索引 $n+1$ 处的日志。但是 1 号 AE 比 2 号还迟，2 号 AE 包已经将各个 Follower 的日志同步到了索引 $n+1$ 处。当过时的 1 号 AE 包被 Follower 接收到时，Follower 依旧按照包中指示同步到索引 n 处，那么同步进度就倒退了，将会引发 bug。因此，Follower 还需要检测这个 AE 是否过时。

5. 到这里，表明这个 AE 包是合法的，同步的开始日志处，双方也都是一致的。但是需要判断这个 AE 是否是过时的。Follower 仅需检测，发来的 AE 包中，`args.Entries` 中最新的日志，本节点中是否存在。若存在，表明无需同步，这个包是过时的。若不存在则进入下一步进行同步。

6. 更新本地的 `rf.log`，`rf.lastLogIndex`，`rf.lastLogTerm` 等变量。

2.2.4 继续完善 heartBeats 函数

在 heartBeats 函数中增加一个功能，即通知 follower 更新 commitIndex。值得注意的是，需要考虑到部分节点尚未同步日志。如果 Leader 无条件地通知那些 follower 自身的 commitIndex 已经更新，要求它们也同步更新，可能导致混乱。因此，领导者应该根据 follower 的情况来进行选择性的通知。

Leader 应该通知那些同步进度超过 commitIndex 的节点。Leader 在 commitIndex 这个数组中记录了各个 follower 的同步进度，因此可以利用这个数组来进行判断。

在本实验中，若 Leader 要通知节点更新 commitIndex，仅需在发给 follower 的 AE 包中赋值 args.LeaderCommit = rf.commitIndex，随后 follower 就能收到最新的 commitIndex。

如果 Leader 发现节点的同步进度不够，并不通知节点进行更新 commitIndex，那么在发给 follower 的 AE 包中不赋值 args.LeaderCommit 即可。这个参数的默认值为 0，并不会对 follower 的 commitIndex 造成任何影响。

2.2.5 继续完善 ticker 函数

在本实验中，需要进行 log replication，Leader 需要使用 nextIndex 和 matchIndex 这两个变量，因此每个 raft 节点成为 leader 后都需要对这两个变量进行初始化。

按照 paper 中的说法，leader 开始需要将 matchIndex 数组全部初始化为 0，nextIndex 数组的初始化则按照自身的 nextIndex 初始化。

2.2.6 继续完善 Make 函数

根据实验说明，index 应当从 1 开始。为了使 index 的值和 log 数组的索引值对应起来，我使用一个 index 和 term 均为 0 的 log 占用了数组的 0 号索引位。这样的处理使得 index 为 1 的 log 存放在 log[1] 处，而非 log[0] 处。同时，这个 index 和 term 均为 0 的 log 作为数组的头部，可以在遍历 log 数组时用来判断是否到达数组的头部。

applyCh 的值可以直接从 Make 函数的参数中获取并赋值。

对于 nextIndex 和 matchIndex，由于它们都是切片，需要使用 make 进行初始化。此外，这些切片的长度应该和 Raft 节点数一样。

2.3 代码实现

- 实现 LogEntry 结构体

```go
// LogEntry 结构定义了 Raft 日志的条目
type LogEntry struct {
    Index   int         // 条目索引
    Term    int         // 条目所在任期
    Command interface{} // 条目的命令
}
```

图 18: LogEntry 结构体

- 实现 Start 函数

```go
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    // 初始化返回值
    index := -1
    term := -1
    isLeader := true
    // Your code here (2B).
    _, isLeader = rf.GetState() // 获取当前节点的状态
    if !isLeader { // 如果不是 leader 节点，则返回当前节点的状态
        return index, term, isLeader
    }
    rf.mu.Lock() // 加锁，保护共享资源
    // 设置 lastLogTerm 和 lastLogIndex 为当前节点的最后一条日志的任期和索引
    rf.lastLogTerm = rf.currentTerm
    rf.lastLogIndex = rf.nextIndex[rf.me]
    index = rf.nextIndex[rf.me]
    term = rf.lastLogTerm
    // 创建新的日志条目
    var peerNum = rf.peerNum
    var entry = LogEntry{Index: index, Term: term, Command: command}
    rf.log = append(rf.log, entry)
    // 更新匹配索引和下一个索引
    rf.matchIndex[rf.me] = index
    rf.nextIndex[rf.me] = index + 1
    // 遍历所有节点，向其他节点发送日志条目
    for i := 0; i < peerNum; i++ {
        if i == rf.me {
            continue
        }
        // 启动协程发送 AppendEntries RPC
        go func(id int, nextIndex int) {
            var args = &AppendEntriesArgs{} // 创建 AppendEntries 参数
            rf.mu.Lock()
            if rf.currentTerm > term { // 检查当前节点的任期是否大于待发送的任期
                rf.mu.Unlock()
                return
            }
            args.Entries = make([]LogEntry, 0) // 创建 Entries 切片，准备发送
            // 如果待发送的索引小于当前节点的索引，将需要发送的日志追加到 Entries 中
            if nextIndex < index {
                for j := nextIndex + 1; j <= index; j++ {
                    args.Entries = append(args.Entries, rf.log[j])
                }
            }
            // 设置 AppendEntries 参数的其他字段
            args.Term = term
            args.LeaderId = rf.me
            rf.mu.Unlock()
            for { // 向其他节点发送 AppendEntries RPC 请求
                var reply = &AppendEntriesReply{}
                rf.mu.Lock()
                if rf.currentTerm > term { // 检查当前节点的任期是否大于待发送的任期
                    rf.mu.Unlock()
                    return
                }
                // 设置 AppendEntries 参数的 PrevLogIndex 和 PrevLogTerm 字段
                args.PrevLogIndex = rf.log[nextIndex-1].Index
                args.PrevLogTerm = rf.log[nextIndex-1].Term
                args.Entries = rf.log[nextIndex : index+1]
                rf.mu.Unlock()
                // 发送 AppendEntries RPC
                var count = 0
                for {
                    if count == 3 {
                        return
                    }
                    // 如果send失败，重试，直到成功
                    if rf.sendAppendEntries(id, args, reply) {
                        break
                    }
                    count++
                }
                rf.mu.Lock()
                if reply.Term > args.Term { // 检查回复的任期是否大于待发送的任期
                    // 如果回复的任期大于当前节点的任期，更新当前节点的状态为 follower
                    if reply.Term > rf.currentTerm {
                        rf.currentTerm = reply.Term
```

```
                    rf.state = "follower"
                    rf.voteFor = -1
                    rf.mu.Unlock()
                    break
                }
                rf.mu.Unlock()
                break
            }
            if !reply.Success {
                if rf.log[nextIndex-1].Term > reply.Term {
                    for rf.log[nextIndex-1].Term > reply.Term {
                        nextIndex--
                    }
                } else {
                    if reply.ConflictIndex != 0 {
                        nextIndex = reply.ConflictIndex
                    } else {
                        nextIndex--
                    }
                }
                if nextIndex == 0 {
                    rf.mu.Unlock()
                    break
                }
                rf.mu.Unlock()
            } else { // 更新匹配索引
                if rf.matchIndex[id] < index {
                    rf.matchIndex[id] = index
                }
                // 检查是否达成一致，更新 commitIndex
                var mp = make(map[int]int)
                for _, val := range rf.matchIndex {
                    mp[val]++
                }
                var tempArray = make([]num2num, 0)
                for k, v := range mp {
                    tempArray = append(tempArray, num2num{key: k, val: v})
                }
                sort.Slice(tempArray, func(i, j int) bool {
                    return tempArray[i].key > tempArray[j].key
                })
                var voteAddNum = 0
                for j := 0; j < len(tempArray); j++ {
                    if tempArray[j].val+voteAddNum >= (rf.peerNum/2)+1 {
                        if rf.commitIndex < tempArray[j].key {
                            rf.commitIndex = tempArray[j].key
                            // 将已提交但未应用的日志条目应用到状态机
                            for rf.lastApplied < rf.commitIndex {
                                rf.lastApplied++
                                var applyMsg = ApplyMsg{}
                                applyMsg.Command = rf.log[rf.lastApplied].Command
                                applyMsg.CommandIndex = rf.log[rf.lastApplied].Index
                                applyMsg.CommandValid = true
                                rf.applyCh <- applyMsg
                            }
                            break
                        }
                    }
                    voteAddNum += tempArray[j].val
                }
                rf.mu.Unlock()
                break
            }

            time.Sleep(10 * time.Millisecond)
        }
    }(i, rf.nextIndex[i])
    // 更新 nextIndex 数组，即使跟随者还没有收到消息
    if index+1 > rf.nextIndex[i] {
        rf.nextIndex[i] = index + 1
    }
}
rf.mu.Unlock() // 解锁，释放锁资源
return index, term, isLeader
}
```

图 19: Start 函数

- AppendEntries 函数

```
    defer rf.mu.Unlock()
    reply.Term = rf.currentTerm // 设置响应中的当前任期为节点的当前任期
    if args.Term < rf.currentTerm { // 如果请求的任期小于当前任期，则拒绝追加日志条目
        reply.Success = false
    } else {
        if args.Entries == nil { // 如果参数中的 Entries 为空，表示这是一条心跳消息
            if args.LeaderCommit > rf.commitIndex { // 如果领导者的提交索引大于节点的提交索引，则更新节点的提交索引
                rf.commitIndex = args.LeaderCommit
                // 将已提交但未应用的日志条目应用到状态机
                for rf.lastApplied < rf.commitIndex {
                    rf.lastApplied++
                    // 生成 ApplyMsg，并发送到 applyCh 通道
                    var applyMsg = ApplyMsg{}
                    applyMsg.Command = rf.log[rf.lastApplied].Command
                    applyMsg.CommandIndex = rf.log[rf.lastApplied].Index
                    applyMsg.CommandValid = true
                    rf.applyCh <- applyMsg
                }
            }
            reply.Success = true
        } else { // 如果参数中的 Entries 不为空，表示领导者要同步日志条目
            // 检查前一个日志条目是否匹配
            var match bool = false
            if args.PrevLogTerm > rf.lastLogTerm {
                reply.Term = rf.lastLogTerm
                reply.Success = false
            } else if args.PrevLogTerm == rf.lastLogTerm {
                if args.PrevLogIndex <= rf.lastLogIndex {
                    match = true
                } else {
                    reply.Term = rf.lastLogTerm
                    reply.ConflictIndex = rf.lastLogIndex
                    reply.Success = false
                }
            } else if args.PrevLogTerm < rf.lastLogTerm {
                // 处理领导者的日志与跟随者的日志不匹配的情况
                var logIndex = len(rf.log) - 1
                for logIndex >= 0 {
                    if rf.log[logIndex].Term > args.PrevLogTerm {
                        logIndex--
                        continue
                    }
                    if rf.log[logIndex].Term == args.PrevLogTerm {
                        reply.Term = args.PrevLogTerm
                        if rf.log[logIndex].Index >= args.PrevLogIndex {
                            match = true
                            reply.Success = true
                        } else {
                            reply.ConflictIndex = rf.log[logIndex].Index
                            reply.Success = false
                        }
                        break
                    }
                    if rf.log[logIndex].Term < args.PrevLogTerm {
                        reply.Term = rf.log[logIndex].Term
                        reply.Success = false
                        break
                    }
                }
            }
            if match { // 匹配成功，追加领导者的日志到跟随者的日志中
                // 注意!!
                // 我们需要考虑一种特殊情况：追随者可能会收到一个较旧的日志复制请求，而追随者此时应该什么都不做
                // 因此关注者应该忽略那些过期的日志复制请求，否则关注者将选择同步并删除最新日志
                var length = len(args.Entries)
                var index = args.PrevLogIndex + length
                reply.Success = true
                if index < rf.lastLogIndex {
                    // 检查是否为过时的日志复制请求
                    if args.Entries[length-1].Term == rf.log[index].Term {
                        return
                    }
                }
                // 处理日志追加
                rf.log = rf.log[:args.PrevLogIndex+1]
                rf.log = append(rf.log, args.Entries...)
                // 更新节点的最后日志索引和任期
                var logLength = len(rf.log)
                rf.lastLogIndex = rf.log[logLength-1].Index
                rf.lastLogTerm = rf.log[logLength-1].Term
                rf.persist()
            }
        }
        // 更新节点的当前任期，并转变为 follower 状态
        rf.currentTerm = args.Term
        rf.state = "follower"
        rf.changeElectionStatus(-1)
        rf.messageCond.Broadcast()
    }
}

// example code to send a RequestVote RPC to a server
```

图 20: AppendEntries 函数

- heartBeats 函数

```go
func (rf *Raft) heartBeats(ch chan int) {
    // 加锁，读取当前节点的当前任期
    rf.mu.Lock()
    var term = rf.currentTerm
    rf.mu.Unlock()
    // 循环执行心跳消息的发送
    for rf.killed() == false {
        // 获取当前节点的状态和是否为领导者
        _, isLeader := rf.GetState()
        if !isLeader {
            // 如果当前节点不是领导者，则退出循环，停止发送心跳消息
            break
        }
        // 加锁，读取当前节点的信息
        rf.mu.Lock()
        var peerNum = rf.peerNum
        var args AppendEntriesArgs
        args.LeaderId = rf.me
        args.Term = rf.currentTerm
        rf.mu.Unlock()
        // 遍历所有节点，向它们发送心跳消息
        for index := 0; index < peerNum; index++ {
            if index == rf.me {
                continue
            }
            // 启动协程发送心跳消息
            go func(index int) {
                // 加锁，准备发送心跳消息的参数
                rf.mu.Lock()
                var args AppendEntriesArgs
                args.LeaderId = rf.me
                args.Term = term
                // 如果匹配索引大于等于提交索引，则包含 LeaderCommit 信息
                if rf.matchIndex[index] >= rf.commitIndex {
                    args.LeaderCommit = rf.commitIndex
                }
                rf.mu.Unlock()
                // 初始化 AppendEntries 回复
                var reply AppendEntriesReply
                // 发送心跳消息
                if rf.sendAppendEntries(index, &args, &reply) {
                    // 加锁，处理心跳回复
                    rf.mu.Lock()
                    defer rf.mu.Unlock()
                    // 检查回复的任期是否大于当前节点的任期
                    if reply.Term > term {
                        // 如果回复的任期大于当前节点的任期，更新当前节点的状态为 follower
                        if reply.Term > rf.currentTerm {
                            rf.state = "follower"
                            rf.currentTerm = reply.Term
                            // 更新投票状态，重置投票
                            rf.voteFor = -1
                        }
                    }
                }
            }(index)
        }
        time.Sleep(time.Duration(HeartBeatInterval) * time.Millisecond)
    }
    ch <- 1
}
```

图 21: heartBeats 函数

- ticker 函数

```
rf.mu.Lock() // 加锁，读取当前节点的当前任期
var recordTerm = rf.currentTerm
rf.mu.Unlock()
// 初始化互斥锁和计数器，用于确保唯一性
var timeMutex sync.Mutex
var currentTermTimes = 0
// 循环执行计时器的逻辑
for rf.killed() == false {
    // Your code here to check if a leader election should
    // be started and to randomize sleeping time using
    // time.Sleep().
    // 在这里检查是否应该启动选举，并通过 time.Sleep() 随机睡眠一段时间
    rf.mu.Lock()
    timeMutex.Lock()
    if recordTerm != rf.currentTerm {
        recordTerm = rf.currentTerm
        currentTermTimes = 0
    }
    currentTermTimes++

    rand.Seed(time.Now().UnixNano()) // 生成随机的睡眠时间
    sleepTime := rand.Intn(TimeOutInterval) + TimeOutInterval
    rf.changeElectionStatus(0) // 初始化 ElectionStatus 变量，用于测试是否超时
    go func(currentTerm int, cTermTimes int) { // 启动协程，用于检测是否超时
        time.Sleep(time.Duration(sleepTime) * time.Millisecond)
        rf.mu.Lock()
        defer rf.mu.Unlock()
        rf.mesMutex.Lock()
        defer rf.mesMutex.Unlock()
        timeMutex.Lock()
        defer timeMutex.Unlock()
        // 如果 ElectionStatus 不为 -1 且当前任期和计数器未更改，则将 ElectionStatus 设置为 1 并广播条件变量
        if rf.ElectionStatus != -1 && currentTerm == rf.currentTerm && cTermTimes == currentTermTimes {
            rf.ElectionStatus = 1
            rf.messageCond.Broadcast()
        }
    }(rf.currentTerm, currentTermTimes)
    // 如果接收到附加日志或请求投票消息，则 ElectionStatus 变为 -1；如果超时，ElectionStatus 变为 1
    rf.mu.Unlock()
    timeMutex.Unlock()
    rf.mesMutex.Lock()
    rf.messageCond.Wait() // 等待 ElectionStatus 变为 -1 或 1
    for rf.ElectionStatus == 2 || rf.ElectionStatus == 3 {
        rf.messageCond.Wait()
    }
    // 如果 ElectionStatus 为 -1，则表示接收到了有效的消息并重置了计时器
    if rf.ElectionStatus == -1 {
        rf.mesMutex.Unlock()
        continue
    } else if rf.ElectionStatus == 1 {
        // 开始选举
        rf.mesMutex.Unlock()
        // 启动两个线程
        // 一个线程是计时器，检查选举是否超时
        // 另一个线程负责启动选举
    voteLoop:
        for rf.killed() == false {
            // 如果超时，发起新的选举
            // ch 用于确定选举是否成功或超时
            rf.mu.Lock()
            rf.state = "candidate"
            rf.currentTerm++
            rf.voteFor = rf.me
            var voteArgs = &RequestVoteArgs{}
            voteArgs.Term = rf.currentTerm
            voteArgs.LastLogTerm = rf.lastLogTerm
            voteArgs.LastLogIndex = rf.lastLogIndex
            voteArgs.CandidateId = rf.me

            var voteMutex sync.Mutex
            var voteCond = sync.NewCond(&voteMutex)

            rf.changeElectionStatus(0)
            // 启动协程发起选举
            go func(currentTerm int, currentTermTimes int, pNum int) {
                // 启动选举
                var grandNum = 1
```

```
var refuseNum = 0
var term = currentTerm
var times = currentTermTimes
var peerNum = pNum
// 遍历所有节点，向它们发送请求投票消息
for index := 0; index < peerNum; index++ {
    if index == rf.me {
        continue
    }
    go func(index int) {
        var reply = &RequestVoteReply{}
        if rf.sendRequestVote(index, voteArgs, reply) {
            rf.mu.Lock()
            if reply.Term > rf.currentTerm {
                // 如果回复的任期大于当前任期，则更新节点状态为 follower
                if term == rf.currentTerm {
                    rf.currentTerm = reply.Term
                    rf.state = "follower"
                    // 发现更高的任期并更新节点状态
                    rf.mu.Unlock()
                    rf.changeElectionStatus(4)
                    rf.messageCond.Broadcast()
                    voteCond.Broadcast()
                    return
                } else {
                    rf.currentTerm = reply.Term
                    rf.state = "follower"
                    // 标记陈旧的投票请求发现更高的任期并更新节点状态
                    rf.mu.Unlock()
                    rf.changeElectionStatus(-1)
                    rf.messageCond.Broadcast()
                    return
                }
            }
            rf.mu.Unlock()
            if reply.VoteGrand { // 处理投票回复
                // 接收到其他节点的投票
                voteCond.L.Lock()
                grandNum++
                voteCond.L.Unlock()
                voteCond.Broadcast()
            } else {
                // 接收到其他节点的拒绝
                voteCond.L.Lock()
                refuseNum++
                voteCond.L.Unlock()
                voteCond.Broadcast()
            }
        }
    }(index)
}
// 等待投票结果
var voteSuccess = false
for {
    voteCond.L.Lock()
    voteCond.Wait()
    rf.mu.Lock()
    if rf.state == "follower" {
        rf.mu.Unlock()
        voteCond.L.Unlock()
        return
    }
    // 获取节点的总数
    var peerNum = rf.peerNum
    rf.mu.Unlock()
    // 判断是否成功获取超过半数的投票
    if grandNum >= (peerNum/2)+1 {
        voteSuccess = true
        // 成功获取大多数的投票
        voteCond.L.Unlock()
        break
    } else if refuseNum+grandNum == peerNum || refuseNum >= (peerNum/2)+1 {
        // 没有获取到大多数的投票
        voteCond.L.Unlock()
        break
    }
    voteCond.L.Unlock()
}
```

```
            }
            // 如果选举成功，更新节点状态为 leader
            if voteSuccess {
                rf.mu.Lock()
                defer rf.mu.Unlock()
                rf.mesMutex.Lock()
                defer rf.mesMutex.Unlock()
                timeMutex.Lock()
                defer timeMutex.Unlock()
                // 如果 ElectionStatus 不为 -1 且不为 4，且当前任期和计数器未更改，则将 ElectionStatus 设置为 2 并广播条件变量
                if rf.ElectionStatus != -1 && rf.ElectionStatus != 4 && term == rf.currentTerm && times == currentTermTimes {
                    // 选举成功表示节点成功成为领导者
                    rf.ElectionStatus = 2
                    rf.state = "leader"
                    var length = len(rf.log)
                    // 在选举后重新初始化 nextIndex 和 matchIndex 数组
                    for i := 0; i < rf.peerNum; i++ {
                        rf.nextIndex[i] = rf.log[length-1].Index + 1
                        rf.matchIndex[i] = 0
                    }
                    rf.messageCond.Broadcast()
                }
                // 如果一个节点收集了正好半数的投票，它将等待直到超时并启动下一次选举
            }
        }(rf.currentTerm, currentTermTimes, rf.peerNum)
        // 启动协程，用于等待选举超时
        go func(currentTerm int, cTermTimes int) {
            rand.Seed(time.Now().UnixNano())
            sleepTime := rand.Intn(TimeOutInterval) + TimeOutInterval
            // 等待选举超时
            time.Sleep(time.Duration(sleepTime) * time.Millisecond)
            rf.mu.Lock()
            defer rf.mu.Unlock()
            rf.mesMutex.Lock()
            defer rf.mesMutex.Unlock()
            timeMutex.Lock()
            defer timeMutex.Unlock()
            // 如果 ElectionStatus 不为 -1 且不为 4，且当前任期和计数器未更改，则将 ElectionStatus 设置为 3 并广播条件变量
            if rf.ElectionStatus != -1 &&
            rf.ElectionStatus != 4 &&
            currentTerm == rf.currentTerm &&
            cTermTimes == currentTermTimes {
                rf.ElectionStatus = 3
                rf.messageCond.Broadcast()
            }
        }(rf.currentTerm, currentTermTimes)
        // 解锁，等待条件变量
        rf.mu.Unlock()
        rf.mesMutex.Lock()
        rf.messageCond.Wait()
        // 等待 ElectionStatus 变为 1 或 0
        for rf.ElectionStatus == 1 || rf.ElectionStatus == 0 {
            rf.messageCond.Wait()
        }
        // 根据 ElectionStatus 的值采取不同的行动
        switch rf.ElectionStatus {
        case 2: // 如果 ElectionStatus 为 2，表示节点成为领导者
            rf.mesMutex.Unlock() // 解锁消息互斥锁，以便让 leader 发送心跳
            // 启动心跳协程
            var leaderch = make(chan int)
            go rf.heartBeats(leaderch)
            <-leaderch
            break voteLoop // 退出选举循环 voteLoop
        case 3: // 如果 ElectionStatus 为 3，表示选举超时
            rf.mesMutex.Unlock() // 解锁消息互斥锁，重新开始选举
            continue
        case 4:
            fallthrough
        case -1:
            // 如果 ElectionStatus 为 4 或 -1，表示停止选举
            // 解锁消息互斥锁，并退出选举循环 voteLoop
            rf.mesMutex.Unlock()
            break voteLoop
        }
    }
}
}
```

图 22: ticker 函数

- Make 函数

```go
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{} // 创建一个 Raft 实例 rf
    rf.peers = peers
    rf.persister = persister
    rf.me = me

    // Your initialization code here (2A, 2B, 2C).
    rf.messageCond = sync.NewCond(&rf.mesMutex) // 初始化消息条件变量
    rf.log = make([]LogEntry, 1) // 初始化日志，初始包含一个空的 LogEntry
    rf.log[0].Index = 0
    rf.log[0].Term = 0
    rf.peerNum = len(peers)
    rf.voteFor = -1
    rf.applyCh = applyCh
    rf.nextIndex = make([]int, rf.peerNum)
    rf.matchIndex = make([]int, rf.peerNum)
    // rf.voteCond = sync.NewCond(&rf.mu)

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState()) // 从持久化存储中恢复节点状态和日志

    // start ticker goroutine to start elections
    go rf.ticker() // 启动 ticker 协程，定期检查是否需要发起选举

    return rf
}
```

图 23: Make 函数

2.4 测试结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2B
Test (2B): basic agreement ...
  ... Passed --   0.6  3   16    4474    3
Test (2B): RPC byte count ...
  ... Passed --   1.4  3   48  114310   11
Test (2B): agreement after follower reconnects ...
  ... Passed --   5.4  3  134   35879    8
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   3.5  5  234   48434    3
Test (2B): concurrent Start()s ...
  ... Passed --   0.7  3   23    6798    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   6.0  3  195   48569    4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  16.8  5 2096  493407  103
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.1  3   44   13006   12
PASS
ok      6.824/raft      36.487s
```

图 24: 2B 运行结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2B
防伪水印:刘芳新 21312482
Test (2B): basic agreement ...
  ... Passed --   0.5  3   16    4506    3
防伪水印:刘芳新 21312482
Test (2B): RPC byte count ...
  ... Passed --   1.5  3   48  114214   11
防伪水印:刘芳新 21312482
Test (2B): agreement after follower reconnects ...
  ... Passed --   5.4  3  132   34997    8
防伪水印:刘芳新 21312482
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   3.5  5  230   46264    3
防伪水印:刘芳新 21312482
Test (2B): concurrent Start()s ...
  ... Passed --   0.7  3   18    5279    6
防伪水印:刘芳新 21312482
Test (2B): rejoin of partitioned leader ...
  ... Passed --   6.0  3  186   46026    4
防伪水印:刘芳新 21312482
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  16.8  5 2094  496744  102
防伪水印:刘芳新 21312482
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.2  3   46   13586   12
PASS
ok      6.824/raft      36.587s
```

图 25: 2B 运行水印

2.5 个人总结

在写 2B 的过程中，我找到了之前 2A 写的一些 bug，在处理多线程程序时，确实存在一些比较隐蔽的 bug，尤其是涉及到锁的使用和线程同步的情况。死锁是其中一个比较棘手的问题，因为它可能在程序运行的某个点触发，而不一定在每次运行都表现出来。可能突然某次运行就有一个 raft 节点发生卡死，没有任何响应了，大概率是因为锁设置的太多了，导致某处出现了死锁。因此我的经验是：

减少锁的使用：尽量减少对共享资源的锁定，可以考虑使用更细粒度的锁，或者使用无锁的数据结构。过多的锁可能导致竞争条件，增加了死锁的概率。

避免嵌套锁：当一个线程持有一个锁的同时尝试获取另一个锁，容易导致死锁。确保在获取锁的时候，尽量避免嵌套锁的情况。

3 lab2-C persistence

3.1 任务分析

Lab 2C 的任务是对 Raft 一致性算法中的关键状态信息进行持久化处理。这一操作的主要目的在于确保系统能够在节点重启或发生故障时从持久化存储中进行恢复，以维持一致性。以下是对 Lab 2C 任务的详细分析：

- 持久化的状态信息：在 Raft 中一些至关重要的状态信息，需要被持久化存储，以确保在节点重启后能够正确继续运行。这些信息包括日志（Log）、当前任期（term）以及投票对象（votedFor）。

- 存储和恢复逻辑：在 Lab 2C 实验中，并非直接存储于磁盘，而是通过一个代表性的 Persister 对象来完成。这个对象充当了持久化存储的代理，负责处理状态信息的存储和恢复逻辑。
- 一致性保障：通过持久化状态信息，Lab 2C 实验确保了在节点重启后系统能够正确继续运行，而不会发生数据丢失或不一致的情况。这种机制提供了强有力的一致性保障，增强了系统的可靠性和稳定性。

3.2 功能设计

3.2.1 完善 persist 函数

这个函数就是把节点必要的信息进行持久化保存，按照所给出的示例代码，以及论文上的描述即可写出来。

3.2.2 完善 readPersist 函数

这个函数就是把之前保存的信息重新读取并加载给节点，不过不能仅仅只是把需要持久化的那几次状态给恢复了。例如，我们这里持久化了 currentTerm、voteFor、log。但是我们恢复过程中，可以通过 log 信息来恢复节点的 lastLogTerm 以及 lastLogIndex，这两个信息也需要恢复，不然就会出现问题。

3.3 代码实现

- 完善 persist 函数

```go
func (rf *Raft) persist() {
    // Your code here (2C).
    // Example:
    // w := new(bytes.Buffer)
    // e := labgob.NewEncoder(w)
    // e.Encode(rf.xxx)
    // e.Encode(rf.yyy)
    // data := w.Bytes()
    // rf.persister.SaveRaftState(data)
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.voteFor)
    e.Encode(rf.log)
    data := w.Bytes()
    rf.persister.SaveRaftState(data)
}
```

图 26: persist 函数

- 完善 readPersist 函数

```go
func (rf *Raft) readPersist(data []byte) {
    if data == nil || len(data) < 1 { // bootstrap without any state?
        return
    }
    // Your code here (2C).
    // Example:
    // r := bytes.NewBuffer(data)
    // d := labgob.NewDecoder(r)
    // var xxx
    // var yyy
    // if d.Decode(&xxx) != nil ||
    //    d.Decode(&yyy) != nil {
    //   error...
    // } else {
    //   rf.xxx = xxx
    //   rf.yyy = yyy
    // }
    r := bytes.NewBuffer(data)
    d := labgob.NewDecoder(r)
    var term int
    var voteFor int
    var log []LogEntry
    if d.Decode(&term) != nil || d.Decode(&voteFor) != nil || d.Decode(&log) != nil {
    } else {
        rf.mu.Lock()
        rf.currentTerm = term
        rf.voteFor = voteFor
        rf.log = log
        var logLength = len(rf.log)
        rf.lastLogTerm = rf.log[logLength-1].Term
        rf.lastLogIndex = rf.log[logLength-1].Index
        rf.mu.Unlock()
    }
}
```

图 27: readPersist 函数

3.4 测试结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2C
Test (2C): basic persistence ...
  ... Passed --   4.4  3  100   27302    6
Test (2C): more persistence ...
  ... Passed --  19.2  5 1266  262498   16
Test (2C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   1.5  3   36    9644    4
Test (2C): Figure 8 ...
  ... Passed --  29.1  5 1180  267230   30
Test (2C): unreliable agreement ...
  ... Passed --   2.5  5  928  298400  246
Test (2C): Figure 8 (unreliable) ...
  ... Passed --  38.7  5 24134 5849143  518
Test (2C): churn ...
  ... Passed --  16.2  5 8524 2506184 1331
Test (2C): unreliable churn ...
  ... Passed --  16.1  5 6440 1921515  808
PASS
ok      6.824/raft      127.746s
```

图 28: 2C 运行结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2C
防伪水印:刘芳新 21312482
Test (2C): basic persistence ...
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
  ... Passed --   5.9  3  130   35842    6
防伪水印:刘芳新 21312482
Test (2C): more persistence ...
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
  ... Passed --  19.3  5 1255  262824   16
防伪水印:刘芳新 21312482
Test (2C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   1.7  3   38    9976    4
防伪水印:刘芳新 21312482
Test (2C): Figure 8 ...
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
防伪水印:刘芳新 21312482
  ... Passed --  31.3  5 1189  265540   26
防伪水印:刘芳新 21312482
Test (2C): unreliable agreement ...
  ... Passed --   2.6  5  959  306929  246
防伪水印:刘芳新 21312482
Test (2C): Figure 8 (unreliable) ...
  ... Passed --  39.5  5 17231 4227464  455
防伪水印:刘芳新 21312482
Test (2C): churn ...
  ... Passed --  16.1  5 8587 2712118 1406
防伪水印:刘芳新 21312482
Test (2C): unreliable churn ...
  ... Passed --  16.2  5 6732 2063067  850
PASS
ok      6.824/raft      132.484s
```

图 29: 2C 运行水印

3.5 个人总结

Lab 2C 目前似乎是对调试能力最大的考验，因为实验中的网络测试环境异常不稳定，极大地考验了程序在网络容错性方面的表现。在处理大量高延迟的网络包、频繁出现网络分区和节点崩溃等问题时，需要在代码中具备较强的容错性。

因此，这个实验的主要工作集中在调试，寻找之前代码中可能存在的漏洞。与此同时，新添加的功能代码量相对较少。主要需要改进的部分似乎在于 Log 同步阶段，需要我们迅速准确定位 leader 和 follower 之间日志一致的位置。

这一过程对调试技能提出了很高的要求，尤其需要高效地解决由于网络不稳定性引起的问题。在这种情况下，细致入微的调试和日志记录将变得尤为重要，以便更有效地定位和解决网络分区和崩溃相关的问题。

4 心得体会

实现 Raft 协议的过程无疑是对编程能力的一次重大考验。在这个过程中，我查找了大量的资料，经历了多次的尝试和重构。

在这次作业中，由于程序的运行环境是终端而非依赖于 IDE 调试功能，对于习惯了 IDE 调试的我感到非常不习惯。在调试的过程中，莫名其妙的 bug 似乎总是不期而至，为了解决问题，我选择使用插桩法，在每个功能内添加大量的 Printf 语句以监测程序的运行过程。

面对死锁问题，这似乎成为最大的困难，因为它会使程序在运行时卡住，而又不报错，从而浪费大量的时间。因此，采用 Printf 语句记录每个步骤的日志成为一种非常有效且必要的手段，帮助我更好地理解程序的运行过程，迅速定位问题并进行解决。

这种经验也反映了在复杂系统设计中，除了理论知识外，调试和日志记录的实际操作也是非常关键的一环。通过这个过程，我不仅仅提升了对 Raft 协议的理解，还培养了在复杂项目中解决问题的能力。

5 参考资料

- MIT 6.824 学习（二）【Raft】: https://blog.csdn.net/weixin_48922154/article/details/123956614
- MIT 6.824 Lab2 翻译（已完成）（Raft Base on Golang）: https://zhuanlan.zhihu.com/p/248686289
- Lecture 06 - Raft1 : https://mit-public-courses-cn-translatio.gitbook.io/mit6-824/lecture-06-raft1
- 分布式一致性算法:Raft 算法（论文翻译）: https://www.cnblogs.com/linbingdong/p/6442673.html