实验三、 Raft 协议的实现之一

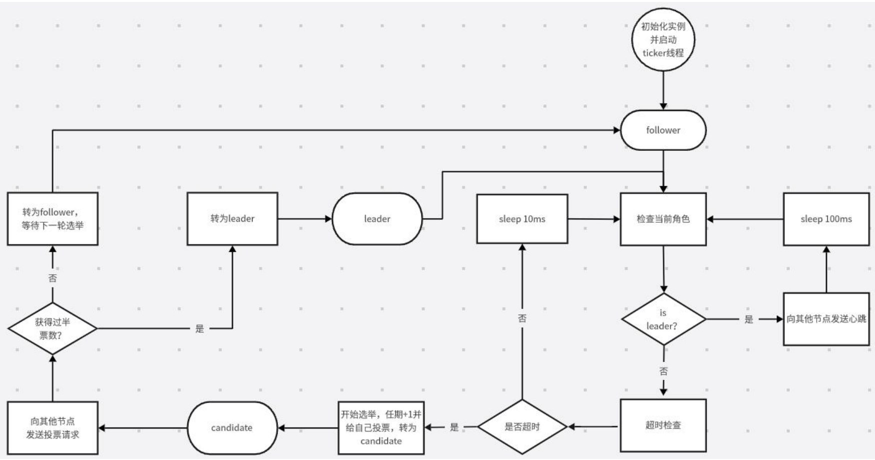
一、Part2A 实现 Leader 选举

1.1 流程与代码分析

要求：通过图片的形式，结合自身代码实现，分析 Raft 中 Leader 选举以及日志追加的具体流程。

（注：由于我做完 2A 就先写了这部分的实验报告，而后面做 2B 的时候调整了部分代码，导致这里有些代码截图与最终版本的 raft.go 中有所出入）

首先，给出心跳保活和超时选举的主要流程图如下：



其在代码中主要对应 raft 的实例中的 ticker 函数及其调用的 sendHeartbeat、checkElectionTimeout、beginAnElection 等方法。

（1）首先，客户端会调用 Make 函数生成一个 Raft 实例，初始都是 follower，Make 函数中会启动一个线程执行 ticker 函数：

```
// start ticker goroutine to start elections
go rf.ticker()
```

（2）ticker 函数在 Raft 实例销毁前会不断地执行，其主要工作为检查当前实例在集群中的角色类型，若为 leader，则以 100ms 的间隔调用 sendHeartbeat 函数向其他实例发送心跳进行保活；若不是 leader，则不断检查当前是否超时（一段时间没有收到 leader 或 candidate 的任何消息），若超时，则调用 beginAnElection 函数开始进行选举：

```
func (rf *Raft) ticker() {  2 usages  lvyy1999
    for rf.killed() == false {
        /.../
        if rf.role == RaftRoleLeader {
            // send heartbeat
            rf.sendHeartbeat()
            // according to lab requirement, sleep for 100 ms to send heartbeat again
            time.Sleep(time.Millisecond * time.Duration(HeartbeatInterval))
        } else {
            // if election timeout, to begin an election
            if rf.checkElectionTimeout() {
                rf.beginAnElection()
            }
            // sleep for a little time to check again
            time.Sleep(time.Millisecond * time.Duration(CheckTimeoutInterval))
        }
    }
}
```

（3）其中，sendHeartbeat 函数负责向其他实例异步发送心跳并接收回复：

```go
func (rf *Raft) sendHeartbeat() {  1 usage  lvyy1999 *
	rf.mu.Lock()
	args := AppendEntriesArgs{
		Term:         rf.currentTerm,
		Entries:      make([]Log, 0),
		LeaderId:     rf.me,
		LeaderCommit: rf.commitIndex,
	}
	rf.mu.Unlock()

	//fmt.Println(time.Now().UnixMilli(), "leader ", rf.me, " begin to send heartbeat",
	//  ", term = ", args.Term)

	// Send heartbeat to all other servers asynchronously
	for i := 0; i < len(rf.peers); i++ {
		if i != rf.me {
			reply := AppendEntriesReply{}
			go func(server int, args *AppendEntriesArgs, reply *AppendEntriesReply) {
				ok := rf.sendAppendEntries(server, args, reply)
				if ok && reply.Term > rf.currentTerm {
					rf.checkAndUpdateCurrentTerm(reply.Term)
				}
			}(i, &args, &reply)
		}
	}
}
```

（4）而 beginAnElection 函数负责开启选举，先使任期+1，转为 candidate 并为自己投票，然后异步向其他实例发送投票请求并收集投票结果，若投票数量过半则赢得选举并转为 leader，若未投票数量过半则输掉选举，转为 follower 并等待其他 candidate 赢得选举或进行下一轮超时后投票：

```go
// To begin an election, a follower increments its current term and transitions to candidate state.
// It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster.
// Return true only if this candidate wins the election.
func (rf *Raft) beginAnElection() {  1 usage  lvyy1999 *
	//fmt.Println(time.Now().UnixMilli(), "server ", rf.me, " begin an election")

	// Update rf's state
	rf.mu.Lock()
	rf.currentTerm++
	rf.votedFor = rf.me
	rf.resetElectionTimeout()
	rf.setRole(RaftRoleCandidate)
	half := int32(len(rf.peers) / 2) // need at least (half + 1) votes to win the election
	lastLog := rf.getLastLog()
	args := RequestVoteArgs{
		Term:         rf.currentTerm,
		CandidateId:  rf.me,
		LastLogTerm:  lastLog.Term,
		LastLogIndex: lastLog.Index,
	}
	rf.mu.Unlock()
```

```go
// Send RequestVote RPCs to all other servers asynchronously and collect the votes
var votedCount, unvotedCount int32 = 1, 0
for i := 0; i < len(rf.peers); i++ {
    if i != rf.me {
        reply := RequestVoteReply{}
        go func(server int, args *RequestVoteArgs, reply *RequestVoteReply, votedCount, unvotedCount *int32) {
            ok := rf.sendRequestVote(server, args, reply)
            if ok && reply.Term > rf.currentTerm {
                rf.checkAndUpdateCurrentTerm(reply.Term)
            }
            if ok && reply.VoteGranted {
                atomic.AddInt32(votedCount, 1)
            } else {
                atomic.AddInt32(unvotedCount, 1)
            }
        }(i, &args, &reply, &votedCount, &unvotedCount)
    }
}
```

```go
// Collect the votes asynchronously
go func(votedCount, unvotedCount *int32) {
    // if received votes is not enough, to spin wait
    for atomic.LoadInt32(votedCount) <= half && atomic.LoadInt32(unvotedCount) <= half {
        time.Sleep(time.Millisecond * time.Duration(10))
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()
    // may receive heartbeat from a new leader during waiting for voting, so need to check state
    if rf.currentTerm == args.Term && rf.role == RaftRoleCandidate {
        // check the result of election and transit role
        if atomic.LoadInt32(votedCount) > half {
            rf.setRole(RaftRoleLeader)
            //fmt.Println("candidate ", rf.me, " wins the election", ", term = ", rf.currentTerm)
        } else {
            //rf.votedFor = -1
            rf.setRole(RaftRoleFollower)
            //fmt.Println("candidate ", rf.me, " fails the election", ", term = ", rf.currentTerm)
        }
    }
}(&votedCount, &unvotedCount)
```

（5）此外，Raft 实例中的 rpc 线程会持续监听其他实例发来的消息，这些消息分为两类，RequestVote（投票请求）和 AppendEntries（追加日志请求，心跳消息就是日志列表为空的追加日志消息）；其中，投票请求的处理过程为先检查候选人的任期、最近日志索引、本实例当前投票信息等内容，若不通过则返回 false，都通过则为该候选人投票并返回 true：

```go
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    if args.Term > rf.currentTerm {
        rf.checkAndUpdateCurrentTerm(args.Term)
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()
    lastLog := rf.getLastLog()
    if args.Term < rf.currentTerm {
        // Reply false if term < currentTerm
        reply.VoteGranted = false
    } else if rf.votedFor != -1 && rf.votedFor != args.CandidateId {
        // Reply false if already voted for another candidate
        reply.VoteGranted = false
    } else if args.LastLogTerm > lastLog.Term || (args.LastLogTerm == lastLog.Term && args.LastLogIndex >= lastLog.Index) {
        // Then if candidate's log is at least as up-to-date as receiver's log, grant vote
        reply.VoteGranted = true
        rf.resetElectionTimeout()
        rf.setRole(RaftRoleFollower)
        rf.votedFor = args.CandidateId
    }

    reply.Term = rf.currentTerm
}
```

（6）追加日志消息的处理位于 AppendEntries 函数中，且当前阶段只有日志列表为空的消息用来作为心跳，follower 每次收到有效的心跳时，会更新超时时间：

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {  no usages  lvyy1999
    if args.Term > rf.currentTerm {
        rf.checkAndUpdateCurrentTerm(args.Term)
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term < rf.currentTerm {
        // Reply false if term < currentTerm
        reply.Success = false
    } else {
        reply.Success = true
        rf.resetElectionTimeout()
        rf.setRole(RaftRoleFollower)
    }

    //fmt.Println(time.Now().UnixMilli(), "server ", rf.me, " receive AppendEntries from leader ", args.LeaderId,
    //  ", args.Term = ", args.Term,
    //  ", rf.currentTerm = ", rf.currentTerm,
    //  ", rf.votedFor = ", rf.votedFor)

    reply.Term = rf.currentTerm
}
```

注 1. 关于超时时间的管理

我的代码中使用了两个函数来控制对超时时间的操作，checkElectionTimeout 用来检查是否超时，resetElectionTimeout 用来更新超时时间，其他的函数必须调用者两个函数而不能直接读写相关变量。具体实现为，用一个原子变量维护下一次超时的时间，每次更新时，用当前时间加上一个 250ms-500ms 之间的随机值（使用随机值是为了避免频繁出现分票问题），即为下一次超时时间，每次检查时，若当前时间大于超时时间则返回 true，否则返回 false：

```go
// will br called in three cases : receive leader's heartbeat; begin an election; vote for a candidate
// reset the nextElectionTime to the time of now + a random value between MinElectionTimeout and 2 * MinElectionTimeout
func (rf *Raft) resetElectionTimeout() {  5 usages  lvyy1999
    timeout := int64(MinElectionTimeout + rand.Intn(MinElectionTimeout))
    atomic.StoreInt64(&rf.nextElectionTime, time.Now().UnixMilli()+timeout)
}

func (rf *Raft) checkElectionTimeout() bool {  2 usages  lvyy1999
    nextElectionTime := atomic.LoadInt64(&rf.nextElectionTime)
    return time.Now().After(time.UnixMilli(nextElectionTime))
}
```

注 2. 关于超时时间的更新时机

根据论文和实验要求进行分析，有三个时机可以更新超时时间，分别是收到有效心跳、作为候选人开启选举和为其他候选人投票，代码应该遵守这一规则：

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {  no usages  lvyy1999
    if args.Term > rf.currentTerm {...}

    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term < rf.currentTerm {...} else {
        reply.Success = true
        rf.resetElectionTimeout()        收到有效心跳
        rf.setRole(RaftRoleFollower)
    }
```

```go
func (rf *Raft) beginAnElection() {  1 usage  lvyy1999 *
    //fmt.Println(time.Now().UnixMilli(), "server ", rf.me, " begin an election")

    // Update rf's state
    rf.mu.Lock()
    rf.currentTerm++
    rf.votedFor = rf.me
    rf.resetElectionTimeout()
    rf.setRole(RaftRoleCandidate)
```

开启选举时

```go
    reply.VoteGranted = false
} else if args.LastLogTerm > lastLog.Term || (args.LastLogTerm == lastLog.Term && args.LastLogIndex >= lastLog.Index) {
    // Then if candidate's log is at least as up-to-date as receiver's log, grant vote
    reply.VoteGranted = true
    rf.resetElectionTimeout()
    rf.setRole(RaftRoleFollower)
    rf.votedFor = args.CandidateId
}
```

为其他candidate投票时

注 3. 关于任期的更新

首先，开启选举时当前任期应该加一。此外，根据论文，对于收到任意的 rpc 请求与回复，若其中传递的任期大于本地任期，都应当用更大的这个任期更新本地任期，为此，专门用一个 checkAndUpdateCurrentTerm 函数进行任期的检查与更新，并在所有收到请求和回复的地方首先执行这个函数。需要注意的是，由于每个任期可以为一个候选人投票，所以 votedFor 应该重置；而旧任期的超时时间在新任期显然不应该生效，所以超时时间也可以重置；并且对于 leader，进入新任期应该自动失去 leader 身份，所以 role 也应该重置为 follower：

```go
// If RPC request or response contains term T > currentTerm:
// set currentTerm = T, convert to follower, and reset voteFor.
// Call this function when receive a request or response firstly.
func (rf *Raft) checkAndUpdateCurrentTerm(term int) {  4 usages  lvyy1999
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if term > rf.currentTerm {
        rf.votedFor = -1
        rf.currentTerm = term
        rf.resetElectionTimeout()
        rf.setRole(RaftRoleFollower)
    }
}
```

```go
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {  3 usages  lvyy1999 *
    if args.Term > rf.currentTerm {
        rf.checkAndUpdateCurrentTerm(args.Term)
    }
```

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {  no usages  lvyy1999
    if args.Term > rf.currentTerm {
        rf.checkAndUpdateCurrentTerm(args.Term)
    }
```

```go
ok := rf.sendRequestVote(server, args, reply)
if ok && reply.Term > rf.currentTerm {
    rf.checkAndUpdateCurrentTerm(reply.Term)
}
```

```go
ok := rf.sendAppendEntries(server, args, reply)
if ok && reply.Term > rf.currentTerm {
    rf.checkAndUpdateCurrentTerm(reply.Term)
}
```

注 4. 在开启选举时，由于发送请求和收集选票是异步运行的，在收集选票期间，可能有其他候选人已经获胜并向本实例发送了心跳使本实例成为 follower，因此，收集到足够选票并加锁后，应该先再次检查当前状态是否还是候选人，任期与开始选举时是否一致，才能对选票结果进行处理：

```go
// Collect the votes asynchronously
go func(votedCount, unvotedCount *int32) {
    // if received votes is not enough, to spin wait
    for atomic.LoadInt32(votedCount) <= half && atomic.LoadInt32(unvotedCount) <= half {
        time.Sleep(time.Millisecond * time.Duration(10))
    }

    rf.mu.Lock()
    defer rf.mu.Unlock()
    // may receive heartbeat from a new leader during waiting for voting, so need to check state
    if rf.currentTerm == args.Term && rf.role == RaftRoleCandidate {
        // check the result of election and transit role
        if atomic.LoadInt32(votedCount) > half {
            rf.setRole(RaftRoleLeader)
```

注 5. 论文中指出，日志条目的索引从 1 开始，而程序中数组下标从 0 开始，因此，为了简化处理，可以在初始化时先向 log 组中写入一条虚拟的日志条目，这样一来，若想访问下标为 i 的日志，就可以直接使用 log[i]，而要访问最新的一条日志，可以使用 log[len(log)-1]：

```go
// Your initialization code here (2A, 2B, 2C).
rf.log = make([]Log, 0)
// log's first index is 1, so append a virtual log
rf.log = append(rf.log, Log{ Term: -1, Index: 0, Command: nil})
rf.role = RaftRoleFollower
```

```go
func (rf *Raft) getLastLog() Log {
    return rf.log[len(rf.log)-1]
}
```

注 6. 代码中有一些地方使用了冗余代码，是为了提高可读性。

1.2 测试说明与结果

测试环境：由于本实验未要求在 linux 进行，因此直接在 windows 本机进行实验。

测试方式：编写了一个 windows 批处理脚本 test2A.bat，循环测试一千次并保存日志，脚本内容如下：

```
@echo off
set COUNT=0
set INTERVAL=2
set LOG_FILE=raft.log

:LOOP_START
set /a COUNT+=1
echo [%DATE% %TIME%] start test %COUNT%/1000 >> "%LOG_FILE%"
go test -run 2A  >> "%LOG_FILE%" 2>&1

if %COUNT% lss 1000 (
    ping -n %INTERVAL% 127.0.0.1 > nul
    goto LOOP_START
)
```

测试结果：

执行 1000 次

```
Test (2A): election after network failure ...
  ... Passed --   4.6  3  118   23164    0
Test (2A): multiple elections ...
  ... Passed --   6.1  7  594  111272    0
PASS
ok      6.824/raft  14.318s
[2025/11/09 周日  3:48:21.25] start test 1000/1000
Test (2A): initial election ...
  ... Passed --   3.1  3   54   13948    0
Test (2A): election after network failure ...
  ... Passed --   4.5  3  116   22780    0
Test (2A): multiple elections ...
  ... Passed --   5.5  7  570  107190    0
PASS
ok      6.824/raft  13.792s
```

成功 1000 次

搜索 "PASS" （1个文件中匹配到1000次，总计查找1次）

失败 0 次

搜索结果 - （匹配0次）
搜索 'FAIL' （0个文件中匹配到0次，总计查找1次）
搜索 "PASS" （1个文件中匹配到1000次，总计查找1次）

注：以上测试为不加 -race 竞争检测机制下进行的，加上 -race 之后会超时

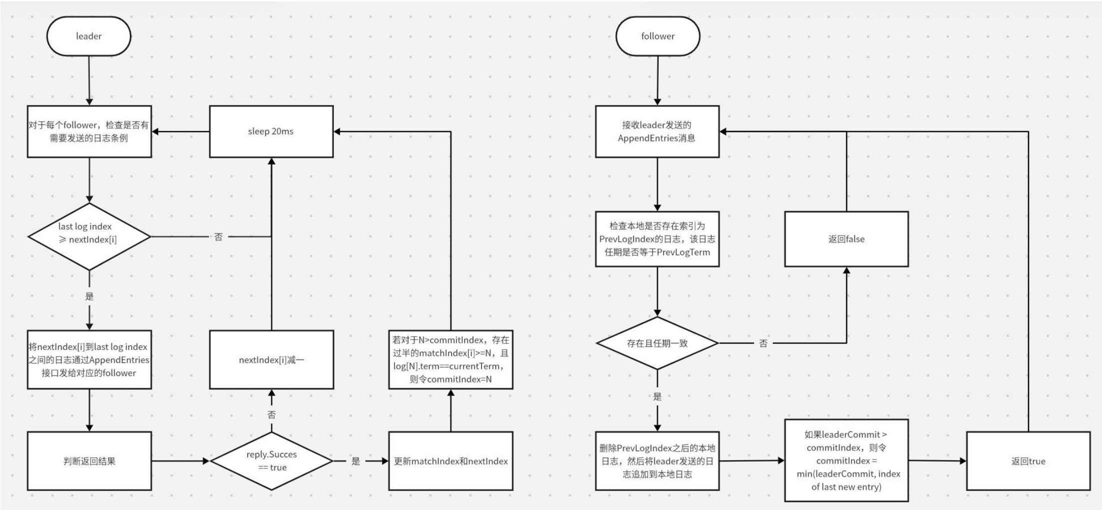
二、Part2B 实现日志追加

2.1 流程与代码分析

要求：通过图片的形式，结合自身代码实现，分析 Raft 中 Leader 选举以及日志追加的具体流程。

（注：由于我做完 2B 后又做了 2C 和 2D，导致这里有些代码截图与最终版本的 raft.go 中有所出入，以 raft.go 中为准，这里主要说思路）

首先，给出 leader 和 follower 关于日志追加的主要流程图如下：



其中 leader 的部分主要对应 checkAndReplicateLog 和 checkAndCommitLog 两个函数，follower 的部分主要对应 AppendEntries 函数。

2.1.1 对于 leader：

（1）客户端通过调用 Start 函数添加新日志；

```go
func (rf *Raft) Start(command interface{}) (int, int, bool) {  18 usages  lvyy1999 *
    term := -1
    index := -1
    isLeader := false

    // Your code here (2B).
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if rf.role == RaftRoleLeader {
        isLeader = true
        term = rf.currentTerm
        index = rf.getLastLog().Index + 1
        rf.log = append(rf.log, Log{
            Term:    term,
            Index:   index,
            Command: command,
        })
        DWriteInfoLog( v...: "leader", rf.me, "received a command", command,
            ", term =", term,
            ", index =", index)
    }

    return index, term, isLeader
```

（2）leader 运行时，ticker 函数中每隔 20ms 调用一次 checkAndReplicateLog，为了降低 rpc 消息并发数量，当发送心跳时先不执行 checkAndReplicateLog（另一种做法是干脆将日志复制和心跳合并到一起，每 100ms 发一次，这样的优点是减少消息数量，缺点是延长日志达成一致的时间，实测下来是可行的）；

```go
func (rf *Raft) ticker() {
    for rf.killed() == false {
        // Your code here to check if a leader election should
        // be started and to randomize sleeping time using
        // time.Sleep().
        if rf.role == RaftRoleLeader {
            // send heartbeat
            if rf.needToSendHeartbeat() {
                rf.sendHeartbeat()
            } else { // avoid to send heartbeat and log replication at the same time, reduce rpc callings
                rf.checkAndReplicateLog()
            }
            time.Sleep(time.Millisecond * time.Duration(CheckLogInterval))
        } else {
```

（3）在 checkAndReplicateLog 函数内部，用 leader 的最新日志与每个不同 follower 的 nextIndex 进行对比，若 lastLog.Index >= rf.nextIndex[i]，则将这个范围内的日志发给对应 follower；

```go
func (rf *Raft) checkAndReplicateLog() {

    // Try to replicate log entries to the followers asynchronously
    lastLog := rf.getLastLog()
    for i := 0; i < len(rf.peers); i++ {
        // If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
        if i != rf.me && lastLog.Index >= rf.nextIndex[i] {
            prevLog := rf.log[rf.nextIndex[i]-1]
            args := AppendEntriesArgs{
                Term:         rf.currentTerm,
                Entries:      rf.log[rf.nextIndex[i] : lastLog.Index+1],
                LeaderId:     rf.me,
                PrevLogTerm:  prevLog.Term,
                PrevLogIndex: prevLog.Index,
                LeaderCommit: rf.commitIndex,
            }

            DWriteDebugLog("leader", rf.me, "send AppendEntries to follower", i,
                ", term =", rf.currentTerm,
                ", startIndex =", rf.nextIndex[i],
                ", entries.len =", len(args.Entries))

            go func(server int, args *AppendEntriesArgs) {
                reply := AppendEntriesReply{}
                ok := rf.sendAppendEntries(server, args, &reply)
```

（4）收到 follower 的回复后，若追加失败，将 nextIndex 减一并等待下次重新发送；若追加成功，更新 matchIndex 和 nextIndex，并调用 checkAndCommitLog 函数对 commitIndex 进行更新；

```go
    ok := rf.sendAppendEntries(server, args, &reply)
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if ok {
        if reply.Term > rf.currentTerm {
            rf.checkAndUpdateCurrentTerm(reply.Term)
        } else if reply.Success { // If successful: update nextIndex and matchIndex for follower
            rf.matchIndex[server] = max(rf.matchIndex[server], lastLog.Index) // the reply maybe disor
            rf.nextIndex[server] = rf.matchIndex[server] + 1
            rf.checkAndCommitLog()
        } else { // If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
            rf.nextIndex[server]--
        }
    } // end if ok
}(i, &args)
```

（5）在 checkAndCommitLog 函数中，根据论文图 2 中 Rules for Servers 里的要求：如果存在一个 N>commitIndex，有半数以上的 matchIndex[i] ⩾ N，且 log[N].term == currentTerm，那么更新 commitIndex = N。

```go
func (rf *Raft) checkAndCommitLog() {
    // If there exists an N such that N > commitIndex, a majority
    // of matchIndex[i] ≥ N, and log[N].term == currentTerm:
    // set commitIndex = N.
    half := len(rf.peers) / 2
    lastLog := rf.getLastLog()
    commitIndex := rf.commitIndex
    for N := commitIndex + 1; N <= lastLog.Index; N++ {
        count := 1 // the first count is leader itself
        if rf.log[N].Term == rf.currentTerm {
            for i := 0; i < len(rf.peers); i++ {
                if i != rf.me && rf.matchIndex[i] >= N {
                    count++
                }
            } // end for i
        }

        if count > half {
            rf.commitIndex = N
        }
    } // end for N
```

2.2.2 对于 follower：

（1）当 follower 收到 leader 发来的 AppendEntries 消息时，首先检查 PrevLogIndex 是否存在，若存在，其对应任期是否等于 PrevLogTerm，若不存在或任期不匹配，则返回 false；

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term > rf.currentTerm {
        rf.checkAndUpdateCurrentTerm(args.Term)
    }

    if args.Term < rf.currentTerm { // reply false if term < currentTerm
        reply.Success = false
    } else {
        // handle heartbeat
        rf.resetElectionTimeout()
        rf.setRole(RaftRoleFollower)

        // handle log replication
        lastLog := rf.getLastLog()
        if lastLog.Index < args.PrevLogIndex || rf.log[args.PrevLogIndex].Term != args.PrevLogTerm {
            // Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
            reply.Success = false
        } else {
            reply.Success = true
```

（2）若匹配成功，根据论文，应该将首个发生冲突（索引一致而任期不一致即为冲突）及之后的 log 删除，再将 leader 发来的本地不存在的 log 都追加进去，然后返回 true；这里我简化为直接把 PrevLogIndex 之后的都删掉，再把 args.Entries 全部追加到本地，可以减少一些判断过程；

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
            reply.Success = false
        } else {
            reply.Success = true

            // If an existing entry conflicts with a new one (same index but
            // different terms), delete the existing entry and all that follow it.
            // Append any new entries not already in the log.
            rf.log = rf.log[:args.PrevLogIndex+1]
            rf.log = append(rf.log, args.Entries...)
```

（3）然后根据论文中的规则更新 follower 的 commitIndex：若 leaderCommit > commitIndex，则令 commitIndex 等于 leaderCommit 和最近一条日志索引之间的最小值；

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {  no usages  lvyy1999 *
        rf.log = append(rf.log, args.Entries...)

        // If leaderCommit > commitIndex, set commitIndex =
        // min(leaderCommit, index of last new entry)
        if args.LeaderCommit > rf.commitIndex {
            rf.commitIndex = min(args.LeaderCommit, rf.getLastLog().Index)
        }
    }
}
```

（4）等待下一条消息。

2.2.3 对于所有 server：

不管是 leader 还是 follower，在本地更新完 commitIndex 后，都需要通过 client 传入的通道 applyCh 去提交到状态机，刚开始我是在更新 commitIndex 后就立即去提交，而由于 applyCh 可能阻塞，我只能异步进行，这会增加许多不必要的冲突，且 leader 和 follower 需要不同的实现。于是，根据学生指南 https://thesquareplanet.com/blog/students-guide-to-raft/中的提示，无论 leader 还是 follower，都单独使用一个线程去负责状态机的提交，从而保证提交顺序和简化代码：

（1）在用 Make 函数新建 Raft 实例时单独启动一个 applier 线程；

```
func Make(peers []*labrpc.ClientEnd, me int,  6 usages  lvyy1999 *
    // start applier goroutine to apply logs
    go rf.applier()
```

（2）applier 函数中会循环检查并提交日志到状态机，提交规则为论文图 2 中的 Rules for Servers 里面约定的：如果 commitIndex > lastApplied，则自增 lastApplied 并提交 log[lastApplied]到状态机；

```
// According to the students' guide, use a applier go routine to check and apply log.
// If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine.
func (rf *Raft) applier() {  2 usages  new *
    for rf.killed() == false {
        rf.mu.Lock()
        if rf.commitIndex > rf.lastApplied {
            rf.lastApplied++
            applyLog := rf.log[rf.lastApplied]
            DWriteInfoLog( v...: "server", rf.me, "applies the log which index is", applyLog.Index)
            applyMsg := ApplyMsg{
                CommandValid: true,
                Command:      applyLog.Command,
                CommandIndex: applyLog.Index,
            }
            rf.mu.Unlock()
            rf.applyCh <- applyMsg
        } else {
            rf.mu.Unlock()
            time.Sleep(time.Millisecond * time.Duration(CheckLogInterval))
        }
    }
```

注 1. 关于选举限制

为了保证日志一致性，使已提交的日志不被覆盖，根据论文 5.3.1 的要求，投票过程中应该将不包含所有已提交日志的候选人排除。由于做 2A 时，已经提前实现了这个限制，本节就不再赘述。

注 2. 关于日志冲突的优化

论文 5.3 提到了一种加速收敛日志冲突情况的优化：follower 在未正确匹配 PrevLogTerm 和 PrevLogIndex 时，在冲突任期中寻找产生冲突的最小索引 conflictingIndex，并返回给 leader，这样 leader 就可以直接用 conflictingIndex 来更新 nextIndex[i]，而不是逐个日志向前尝试。

```go
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {

    // handle log replication
    lastLog := rf.getLastLog()
    if lastLog.Index < args.PrevLogIndex || rf.log[args.PrevLogIndex].Term != args.PrevLogTerm {
        // Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
        reply.Success = false
        if lastLog.Index < args.PrevLogIndex {
            reply.ConflictingIndex = lastLog.Index + 1
        } else if rf.log[args.PrevLogIndex].Term != args.PrevLogTerm {
            conflictingIndex := args.PrevLogIndex
            conflictingTerm := rf.log[args.PrevLogIndex].Term
            for conflictingIndex > 1 && rf.log[conflictingIndex].Term == conflictingTerm {
                // find the minimum index in the conflicting term
                conflictingIndex--
            }
            reply.ConflictingIndex = conflictingIndex
        }
    } else {
```

```go
func (rf *Raft) checkAndReplicateLog() {
            go func(server int, args *AppendEntriesArgs) {
                reply := AppendEntriesReply{}
                ok := rf.sendAppendEntries(server, args, &reply)
                rf.mu.Lock()
                defer rf.mu.Unlock()
                if ok {
                    if reply.Term > rf.currentTerm {
                        rf.checkAndUpdateCurrentTerm(reply.Term)
                    } else if reply.Success { // If successful: update nextIndex and matchIndex for follower
                        rf.matchIndex[server] = max(rf.matchIndex[server], lastLog.Index) // the reply maybe disorder, so need to
                        rf.nextIndex[server] = rf.matchIndex[server] + 1
                        rf.checkAndCommitLog()
                    } else { // If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
                        // rf.nextIndex[server]--
                        rf.nextIndex[server] = reply.ConflictingIndex
                    }
```

leader

注 3. 关于 leader 中 matchIndex 和 nextIndex 的初值

当一个 candidate 当选为 leader 时，需要设置 matchIndex 和 nextIndex 数组的值。nextIndex 可以初始为最新一条日志的索引，来尽可能减少每次同步日志的消息长度，日志匹配机制会将其纠正到正确的位置；而 matchIndex 不能直接初始为 nextIndex-1，而应该初始为当前已确认提交的最大值 commitIndex 或干脆初始为 0，防止将未真正达成多数一致的日志直接提交。

```go
if rf.currentTerm == args.Term && rf.role == RaftRoleCandidate {
    // check the result of election and transit role
    if atomic.LoadInt32(votedCount) > half {
        DWriteInfoLog("candidate", rf.me, " wins the election, term =", rf.currentTerm)
        rf.setRole(RaftRoleLeader)
        atomic.StoreInt64(&rf.lastSendHeartBeat, 0) // ensure to send heartbeat immediately
        for i := 0; i < len(rf.peers); i++ {
            rf.matchIndex[i] = 0
            rf.nextIndex[i] = args.LastLogIndex + 1
        }
```

2.2 测试说明与结果

测试环境:由于本实验未要求在 linux 进行,因此直接在 windows 本机进行实验。

测试方式：编写了一个 windows 批处理脚本 test2B.bat，循环测试一千次并保存日志，脚本内容如下：

```
@echo off
set COUNT=0
set INTERVAL=2
set LOG_FILE=raft.log

:LOOP_START
set /a COUNT+=1
echo [%DATE% %TIME%] start test %COUNT%/1000 >> "%LOG_FILE%"
go test -run 2B  >> "%LOG_FILE%" 2>&1

if %COUNT% lss 1000 (
    ping -n %INTERVAL% 127.0.0.1 > nul
    goto LOOP_START
)
```

测试结果：

执行 1000 次

```
ok      6.824/raft   41.692s
[2025/11/13 周四 11:42:20.19] start test 999/1000
Test (2B): basic agreement ...
  ... Passed --   0.8  3   16    4428    3
Test (2B): RPC byte count ...
  ... Passed --   1.8  3   48  114136   11
Test (2B): agreement after follower reconnects ...
  ... Passed --   6.0  3  149   38404    7
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   3.6  5  331   69900    3
Test (2B): concurrent Start()s ...
  ... Passed --   0.7  3   10    2794    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   6.2  3  268   64674    4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  19.6  5 3362 2938225  102
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.1  3   34    9946   12
PASS
ok      6.824/raft   41.994s
[2025/11/13 周四 11:43:05.32] start test 1000/1000
Test (2B): basic agreement ...
  ... Passed --   0.8  3   16    4428    3
Test (2B): RPC byte count ...
```

成功 1000 次

搜索 "ok " （1个文件中匹配到1000次 总计查找1次）

失败 0 次

搜索 "FAIL" （0个文件中匹配到0次，总计查找1次）

回归测试 2A，成功通过

```
Test (2A): initial election ...
  ... Passed --   3.0  3   46   12864    0
Test (2A): election after network failure ...
  ... Passed --   4.6  3  106   21089    0
Test (2A): multiple elections ...
  ... Passed --   5.6  7  480   89821    0
PASS
ok      6.824/raft      14.224s
```

注：以上测试为不加 -race 竞争检测机制下进行的，加上 -race 之后会超时

三、完整代码

由于代码较长，这里不贴出，可以直接看文件夹内的源代码文件，或从我的 github 仓库获取：https://github.com/lvyy1999/My6.824 。