



中山大學
SUN YAT-SEN UNIVERSITY

分布式计算-实验报告

Raft 2A-2C



中山大學
SUN YAT-SEN UNIVERSITY

题 目: Raft 实验-2A,2B,2C

课程名称: 分布式计算

授课老师: 余晨韵

学生姓名: 张瑞程

学 号: 22354189

2024 年 12 月

目录

1. Pre-Checking.....	3
1.1 环境配置.....	3
1.2 实验准备.....	3
1.3 实验说明.....	3
1.4 快速测试:	3
1.5 实验结果:	3
2. Lab2A-Leader Election-选举.....	5
2.1 任务要求.....	5
2.2 任务分析.....	5
2.3 代码详解.....	6
2.3.1 Raft 基本架构.....	6
2.3.2 raft 基本函数.....	8
<1> Make 函数.....	8
<2> ticker 函数.....	9
<3> Kill & Killed 函数.....	10
<4> changeRole 函数.....	11
<5> 其他函数.....	12
2.3.3 vote rpc 部分.....	12
<1> 结构体定义.....	12
2.3.4 心跳的实现.....	16
2.4 Lab2A-测试结果.....	18
3. Lab2B-Log Replication-日志复制.....	18
3.1 任务要求:	18
3.3 代码分析:.....	19
3.3.1 Raft 基本函数-2B 部分.....	19
3.3.2 append entries rpc 参数.....	21
3.3.2 append entries rpc-回复部分.....	22
3.4 Lab2B-测试结果.....	26
4. Lab2C-Persistence-持久化.....	27
4.1 任务要求.....	27
4.2 任务分析.....	27
4.3 代码详解.....	27
4.4 Lab2C-测试结果.....	30
5. 实验心得.....	30
6. 相关参考资料.....	31

1. Pre-Checking

1.1 环境配置

- 操作系统: Ubuntu 16.04 虚拟机
- 编辑器: Vscode 2023。

1.2 实验准备

通过执行命令 `git clone git://g.csail.mit.edu/6.824-golabs-2022 6.824`, 按照提示下载实验所需文件。若在后续实验过程中发现程序无法正常运行, 需要更新电脑上的 `gcc` 版本, 可使用命令 `sudo apt-get install gcc-5 g++-5` 进行安装。

1.3 实验说明:

本次实验依据 MIT-6.824 的课程安排, 要求完成 Lab 2-ABC, 使用 Go 语言实现。在实验开始前, 需提前阅读论文 [《raft-extended.pdf》](#)。

- Lab-2A: 实现 Raft 领导者选举以及发送心跳信息
- Lab-2B: 假设客户需要写入新记录, 领导者如何管理日志复制
- Lab-2C: 记录 Raft 集群状态, 并在工作过程中更新, 保证节点故障重启后依然能正常工作。

1.4 快速测试

打开命令框, 输入:

1. <code>cd ~/6.824/src/x</code>	#进入特定 x 文件
2. <code>go test -run x</code>	#测试特定 x 模块
3. <code>go test</code>	#测试 x 文件中的所有实验模块

1.5 实验结果

```
db5@ubuntu: ~/6.824/src/Raft2A
db5@ubuntu:~/6.824/src/Raft2A$ go test -run 2A
Only for: ZhangRuichengTest (2A): initial election ...
... Passed -- 3.0 3 36 9840 0
Test (2A): election after network failure ...
... Passed -- 4.5 3 109 19770 0
Test (2A): multiple elections ...
ok 6.824/Raft2A 12.651s
```

Lab2A 的测试结果为 12.651s



```
db@ubuntu: ~/6.824/src/Raft2B
db@ubuntu:~/6.824/src/Raft2B$ go test -run 2B
Only for: ZhangRuichengTest (2B): basic agreement ...
... Passed -- 0.9 3 16 4362 3
Test (2B): RPC byte count ...
... Passed -- 2.2 3 48 113846 11
Test (2B): agreement after follower reconnects ...
... Passed -- 6.0 3 111 28457 8
Test (2B): no agreement if too many followers disconnect ...
... Passed -- 3.7 5 194 37978 4
Test (2B): concurrent Start()s ...
... Passed -- 0.7 3 12 3308 6
Test (2B): rejoin of partitioned leader ...
... Passed -- 6.5 3 189 45379 4
Test (2B): leader backs up quickly over incorrect follower logs ...
... Passed -- 21.6 5 3888 2905444 102
Test (2B): RPC counts aren't too high ...
ok      6.824/Raft2B      38.012s
```

Lab2B 的测试结果为 38.012s

```
db@ubuntu: ~/6.824/src/Raft2C
db@ubuntu:~/6.824/src/Raft2C$ go test -run 2C
Only for: ZhangRuichengTest (2C): basic persistence ...
... Passed -- 4.0 3 65 16144 6
Test (2C): more persistence ...
... Passed -- 17.1 5 881 185938 16
Test (2C): partitioned leader and one follower crash, leader restarts ...
... Passed -- 2.1 3 32 8025 4
Test (2C): Figure 8 ...
... Passed -- 32.3 5 849 181781 55
Test (2C): Figure 8 (unreliable) ...
... Passed -- 31.0 5 8906 20691883 200
Test (2C): churn ...
... Passed -- 16.3 5 3756 2139781 570
Test (2C): unreliable churn ...
... Passed -- 16.2 5 2840 1078282 940
PASS
ok      6.824/Raft2C      119.132s
```

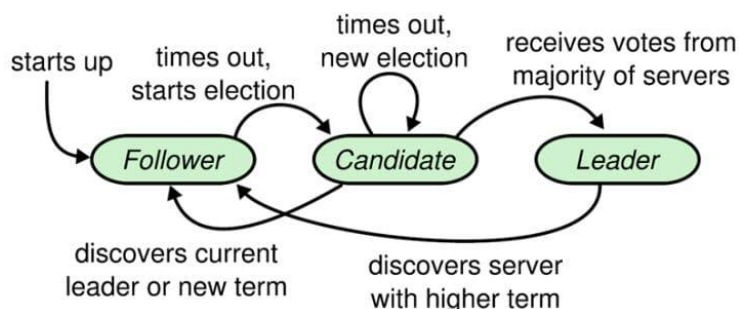
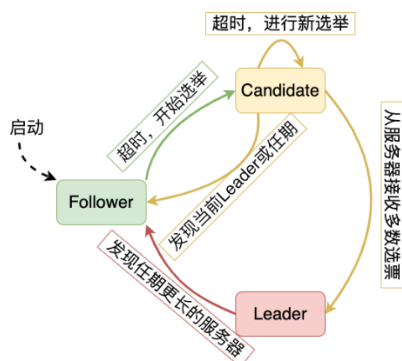
Lab2C 的测试结果为 119.132s

总时长: 12.651s + 38.012s + 119.132s = 169.785s < 170s

2. Lab2A-Leader Election-选举

2.1 任务要求:

实现 Raft 领导者选举和心跳 (AppendEntries RPCs 暂时不用考虑日志条目)。



2.2 任务分析:

第 2A 部分的目标是选举单个领导者, 如果没有失败, 领导者仍然是领导者, 如果旧领导者失败或数据包发送到, 则新领导者接管。具体实现步骤如下:

- 实现 Raft 结构定义与初始化:** 编写自己的 Raft 结构定义及初始化函数 (Make), 设置相关定时器, 并完成 ticker 函数的编写。此函数将定期触发, 以驱动 Raft 状态机的运转。
- 实现领导者选举机制:**
 - 完善 RequestVote RPC 相关结构:** 填写 RequestVoteArgs 和 RequestVoteReply 结构体, 用于领导者选举过程中的请求与响应。修改 Make() 函数, 创建一个后台 goroutine。该 goroutine 会在一段时间内未收到其他节点消息时, 定期发送 RequestVote RPC, 从而触发领导者选举。通过这种方式, 节点能够知晓谁是领导者, 或者在已有领导者的情况下, 尝试自己成为领导者。
 - 实现 RequestVote RPC 处理程序:** 编写 RequestVote() RPC 处理程序, 使服务器之间能够相互投票, 从而完成领导者选举的流程。
- 实现心跳机制:**
 - 填写 AppendEntries RPC 相关结构:** 完善 AppendEntriesArgs 和 AppendEntriesReply 结构体。此处无需完全实现该 RPC 的全部功能, 仅需使其具备心跳的作用。
 - 编写 AppendEntries RPC 处理程序:** 编写 AppendEntries RPC 处理程序方法, 用于重置选举超时时间。这样, 当其他服务器已被选举为领导者时, 不会继续尝试成为领导者。同时, 要确保不同节点的选举超时时间不会总是同时触发, 否则可能导致所有节点都只为自己投票, 而无法成功选出领导者。
- 控制心跳发送频率:** 测试代码要求领导者每秒发送心跳 RPC 的次数不得超过十次, 需合理控制心跳发送的频率, 以满足这一要求。



5. **确保新领导者及时选举：**测试代码要求在旧领导者出现故障后，若大多数节点仍可正常通信，新的领导者需在 5 秒内被选举出来。但需注意，在出现分裂投票的情况下，领导者选举可能需要多轮才能完成。这可能发生在数据包丢失或候选人不幸选择了相同的随机退避时间时。因此，我们必须选择足够短的选举超时时间以及心跳间隔，以确保即使选举需要多轮，也极有可能在不到 5 秒的时间内完成选举。
6. **处理 Raft 实例关闭情况：**测试代码在永久关闭 Raft 实例时，会调用 Raft 的 `rf.Kill()` 方法。我们可以通过 `rf.killed()` 方法检查是否已调用 `Kill()`。在所有循环中都需要进行此类检查，以避免已关闭的 Raft 实例输出错误信息。

2.3 代码详解：

2.3.1 Raft 基本架构

此部分任务目标是选举单个领导者，如果没有失败，领导者仍然是领导者，如果旧领导者失败或数据包发送到，则新领导者接管。下图展示了 Raft 的结构体的定义，注释标明了其在任务中的作用：

```
1. type Raft struct {
2.     mu          sync.Mutex          // 锁，用于保护对节点状态的共享访问
3.     peers       []*labrpc.ClientEnd // 所有对等节点的 RPC 端点，每个 clientEnd 对应一个通信端点
4.     persister   *Persister         // 用于保存节点持久化状态的对象
5.     me          int                // 当前节点在 peers[] 中的索引
6.     dead        int32              // 由 Kill() 设置的标志位
7.
8.     // 2A, 2B, 2C 实验所需的数据
9.     role        Role               // 当前节点的角色（领导者、跟随者或候选者）
10.    currentTerm int                 // 当前任期号
11.    votedFor     int                 // 当前任期内投票给的候选人
12.    logs         []LogEntry          // 日志条目的列表
13.    commitIndex  int                 // 已提交日志的最大索引
14.    lastApplied  int                 // 最后应用到状态机的日志索引
15.    nextIndex    []int               // 下一个要发送给每个跟随者的日志索引
16.    matchIndex   []int               // 每个跟随者匹配的最大日志索引
17.
18.    electionTimer *time.Timer          // 用于触发领导者选举的定时器
19.    appendEntriesTimers []*time.Timer // 用于触发心跳（AppendEntries RPC）的定时器数组
20.    applyTimer     *time.Timer          // 用于触发日志应用的定时器
21.    applyCh        chan ApplyMsg    // 用于提交应用日志的通道，具体处理在 config.go 文件中
22.    notifyApplyCh  chan struct{} // 用于通知日志应用完成的通道
23.    stopCh         chan struct{} // 用于停止 Raft 实例的通道
24.
25.    lastSnapshotIndex int           // 快照中最后一条日志的索引，是真正的 index，不是存储在 logs 中的 index
}
```

```
26.     lastSnapshotTerm    int           // 快照中最后一条日志的任期号
27. }
```

三种节点类型:

本实验中，节点具有三种不同的状态：

- ①跟随者(Follower):相当于普通群众，接收和处理来自领导者的消息，当等待领导者心跳超时的时候，就推举自己当候选人。
- ②候选人(Candidate):向其他节点发送请求投票的 RPC 消息，通知其他节点来投票，若赢得大多数选票，则晋升为领导者。
- ③领导者(Leader):霸道总裁，具有主导权，管理日志复制。

状态变量

(1) 所有服务器上的持久性状态(在响应 RPC 请求之前，已经更新到了稳定的存储设备)

参数	解释
currentTerm	服务器已知最新的任期(在服务器首次启动时初始化为 0, 单调递增)
votedFor	当前任期内收到选票的 candidateId, 如果没有投给任何候选人则为空
log[]	日志条目; 每个条目包含了用于状态机的命令，以及领导人接收到该条目时的任期

(2) 所有服务器上的易失性状态

参数	解释
commitIndex	已知已提交的最高的日志条目的索引
lastApplied	已经被应用到状态机的最高的日志条目的索引

(3) 领导人(服务器)上的易失性状态

参数	解释
nextIndex[]	对于每一台服务器，发送到该服务器的下一个日志条目的索引
matchIndex[]	对于每一台服务器，已知的已经复制到该服务器的最高日志条目的索引

(4) 其他参数

参数	解释
term	领导人的任期
leaderId	领导人 ID 因此跟随者可以对客户端进行重定向(跟随者根据领导人 ID 把客户端的请求重定向到领导人，比如有时客户端把请求发给了跟随者而不是领导人)
prevLogIndex	紧邻新日志条目之前的那个日志条目的索引
prevLogTerm	紧邻新日志条目之前的那个日志条目的任期
entries[]	需要被保存的日志条目(被当做心跳使用时，则日志条目内容为空；为了提高效率可能一次性发送多个)
leaderCommit	领导人的已知已提交的最高的日志条目的索引
term	当前任期，对于领导人而言它会更新自己的任期
success	如果跟随者所含有的条目和 prevLogIndex 以及 prevLogTerm 相匹配，则为 true, 否则为 false

```
const (  
    Role_Follower = 0  
    Role_Candidate = 1  
    Role_Leader = 2  
)  
  
const (  
    ElectionTimeout = time.Millisecond * 300 // 选举超时时间/心跳超时时间  
    HeartBeatInterval = time.Millisecond * 150 // leader 发送心跳  
    ApplyInterval = time.Millisecond * 100 // apply log  
    RPCTimeout = time.Millisecond * 100  
    MaxLockTime = time.Millisecond * 10 // debug  
)
```

针对每一个节点的状态和时间定义了如上的一些参数。

2.3.2 raft 基本函数

<1> Make 函数

先来看一下 Make 函数，都需要调用 Make 函数进行初始化：

```
func Make(peers []*labrpc.ClientEnd, me int,  
    persister *Persister, applyCh chan ApplyMsg) *Raft {  
    DPrintf("make a raft, me: %v", me)  
    rf := &Raft{}  
    rf.peers = peers  
    rf.persister = persister  
    rf.me = me  
  
    // Your initialization code here (2A, 2B, 2C).  
    rf.role = Role_Follower  
    rf.currentTerm = 0  
    rf.votedFor = -1  
    rf.logs = make([]LogEntry, 1) //下标为0存储快照  
    // initialize from state persisted before a crash  
    rf.commitIndex = 0  
    rf.lastApplied = 0  
    rf.nextIndex = make([]int, len(rf.peers))  
    rf.matchIndex = make([]int, len(rf.peers))  
    //读取持久化数据  
    rf.readPersist(persister.ReadRaftState())  
  
    rf.electionTimer = time.NewTimer(rf.getElectionTimeout())  
    rf.appendEntriesTimers = make([]*time.Timer, len(rf.peers))  
    for i := 0; i < len(rf.peers); i++ {  
        rf.appendEntriesTimers[i] = time.NewTimer(HeartBeatInterval)  
    }  
    rf.applyTimer = time.NewTimer(ApplyInterval)  
    rf.applyCh = applyCh  
    rf.notifyApplyCh = make(chan struct{}, 100)  
    rf.stopCh = make(chan struct{})  
  
    // start ticker goroutine to start elections  
    rf.ticker()  
  
    return rf  
}
```

其实 Make 函数主要就做了三件事情：

- (1) 初始化 Raft 的基本结构;
 - (2) 根据传入的 Persister, 读入持久化数据; (这一步暂时可以不用实现)
 - (3) 定义多个定时器, 并调用 ticker() 函数进行处理。
- 这个 ticker 函数很重要, 它是 Raft 节点在初始化后, 真正要运行的部分。

<2> ticker 函数

```
1. // The ticker go routine starts a new election if this peer hasn't received
2. // heartbeats recently.
3. func (rf *Raft) ticker() {

4.     go func() { // 启动一个goroutine, 用于处理日志应用和选举超时

5.         for {
6.             select {

7.                 case <-rf.stopCh: // 如果接收到停止信号, 退出goroutine

8.                     return
9.                     case <-rf.applyTimer.C:

10.                        rf.notifyApplyCh <- struct{}{} // 如果应用定时器触发, 通知可以应用日志

11.                        case <-rf.notifyApplyCh: // 当有日志记录提交了, 要进行应用
12.                            rf.startApplyLogs()
13.                        }
14.                    }
15. }()

16. // 选举定时: 启动一个goroutine 来处理选举定时器

17. go func() {
18.     for rf.killed() == false {
19.         // Your code here to check if a leader election should
20.         // be started and to randomize sleeping time using
21.         // time.Sleep().
22.         select {

23.             case <-rf.stopCh: // 如果接收到停止信号, 退出goroutine

24.                 return

25.                 case <-rf.electionTimer.C: // 如果选举定时器触发, 启动选举

26.                     rf.startElection()
```

```
27.     }
28.   }
29. }()
30. // leader 发送日志定时
31. for i, _ := range rf.peers {
32.     if i == rf.me {
33.         continue
34.     }
35.     go func(cur int) {
36.         for rf.killed() == false {
37.             select {
38.             case <-rf.stopCh:
39.                 return
40.             case <-rf.appendEntriesTimers[cur].C: // 如果发送日志定时器触发, 向对应的跟随者发送日志
41.                 rf.sendAppendEntriesToPeer(cur)
42.             }
43.         }
44.     }(i)
45. }
46. }
```

ticker 函数的核心作用是启动两个 goroutine 来分别处理三种不同的任务:

1. 第一部分: 领导者选举

- 当超过预定的心跳时间间隔时, 将调用 startElection 函数来发起领导者选举。这一步骤是 Raft 算法中确保集群中存在一个活跃领导者的关键机制。

2. 第二部分: 日志复制

- 定期调用 sendAppendEntriesToPeer 函数, 向其他节点发送 AppendEntries RPC。这一操作仅由当前的领导者执行, 目的是将领导者的日志条目复制到所有跟随者节点上, 以保持日志的一致性。

以上两个部分是 raft.go 文件在 2A 部分的主要逻辑了, 除此之外, 还需一些辅助功能函数来支持这些核心逻辑的实现, 下面就是一些功能函数的编写:

<3> Kill & Killed 函数

```
1. func (rf *Raft) Kill() {
2.     atomic.StoreInt32(&rf.dead, 1) // 设置 dead 为 1, 表示 Raft 节点已被杀死
3.     // Your code here, if desired.
4. }
```



这个方法用于标记 Raft 节点为“已杀死”状态。这是通过将 `rf.dead` 变量设置为 1 来实现的。使用 `atomic.StoreInt32` 函数确保这个操作是原子的，即在多线程环境中安全。这样可以防止竞态条件，确保 `dead` 状态的一致性。

```
1. func (rf *Raft) killed() bool {
2.     z := atomic.LoadInt32(&rf.dead) // 读取 dead 的值
3.     return z == 1 // 如果 dead 为 1, 返回 true, 表示节点已被杀死
4. }
```

这个方法用于检查 Raft 节点是否已被标记为“已杀死”。使用 `atomic.LoadInt32` 函数安全地读取 `rf.dead` 的值。如果读取的值等于 1，则返回 `true`，表示节点已被杀死；否则返回 `false`。

<4> changeRole 函数

该函数是每一个 raft 节点要改变角色时需要做出的变化。需要注意的是，对于改变角色时的动作，一部分在此函数进行，一部分则直接被写在了对应的节点里。

```
1. func (rf *Raft) changeRole(newRole Role) {
2.     // 检查新角色是否有效, 如果不在有效范围内, 触发 panic
3.     if newRole < 0 || newRole > 3 {
4.         panic("unknown role")
5.     }
6.     rf.role = newRole // 更新节点的角色为新角色
7.     switch newRole {
8.     case Role_Follower:
9.         // 跟随者角色可能需要的特定初始化代码 (当前未实现)
10.    case Role_Candidate:
11.        rf.currentTerm++ // 增加当前任期号
12.        rf.votedFor = rf.me // 在新的任期开始时, 候选者投票给自己
13.        rf.resetElectionTimer() // 重置选举定时器
14.    case Role_Leader:
15.        // 领导者只有两个特殊的数据结构: nextIndex, matchIndex
16.        lastLogIndex := rf.getLastLogTermAndIndex() // 获取最后一条日志的索引和任期
17.        for i := 0; i < len(rf.peers); i++ {
18.            rf.nextIndex[i] = lastLogIndex + 1 // 设置每个跟随者的下一个日志索引
19.            rf.matchIndex[i] = lastLogIndex // 设置每个跟随者匹配的最大日志索引
20.        }
21.        rf.resetElectionTimer() // 重置选举定时器
22.    default:
23.        panic("unknown role") // 如果角色未知, 触发 panic
24.    }
25. }
```

角色特定操作:

1. 跟随者 (Follower): 无操作。

2. 候选者 (Candidate)：增加当前任期号，设置自己为本任期的投票对象，并重置选举定时器。
3. 领导者 (Leader)：获取最后一条日志的索引和任期，设置每个跟随者的 nextIndex 和 matchIndex，并重置选举定时器。

<5> 其他函数

- (1) GetState()：获取当前节点的 term 和 leader 标记；
- (2) getElectionTimeout()：获取一个心跳过期时间，每一个节点的心跳过期时间大部分情况下是要不一样的；
- (3) resetElectionTimer()：重置选举定时器；
- (4) getLastLogTermAndIndex()：获取最后一条日志的 term 和 index 信息。

2.3.3 vote rpc 部分

<1> 结构体定义

接下来要实现的函数就是跟进行选举相关的。先看下 rpc 的参数和回复结构体：

```
1. type RequestVoteArgs struct {  
2.     // Your data here (2A, 2B).  
3.     Term          int  
4.     CandidateId   int  
5.     LastLogIndex  int  
6.     LastLogTerm   int  
7. }
```

```
8. type RequestVoteReply struct {  
9.     // Your data here (2A).  
10.    Term          int  
11.    VoteGranted   bool  
12. }  
13.
```

这个没什么好说的，基本都是根据论文来实现的。

<2> 发起选举

```
1. startElection()用于启动领导者选举：  
2. func (rf *Raft) startElection() {  
3.     rf.mu.Lock()           // 加锁以保护共享资源  
4.     rf.resetElectionTimer() // 重置选举定时器  
5.     if rf.role == Role_Leader { // 如果当前节点已经是领导者  
6.         rf.mu.Unlock()      // 解锁  
7.         return              // 直接返回，不进行选举  
8.     }  
9. }
```

这个函数通常在以下几种情况下被调用：

- 当跟随者在一定时间内没有收到领导者的心跳时，会调用 startElection 来尝试成为新的领导者。
- 在某些特定条件下，例如网络分区恢复后，可能需要重新选举领导者。

〈3〉 重置选举时间

resetElectionTimer 函数用于重置领导者选举的定时器:

```
1. func (rf *Raft) resetElectionTimer() {  
2.     rf.electionTimer.Stop() // 停止当前的选举定时器  
3.     rf.electionTimer.Reset(rf.getElectionTimeout()) // 重置定时器为新的选举超时时间  
4. }
```

这里, getElectionTimeout 函数通常会返回一个随机化的选举超时时间, 以防止所有节点同时触发选举, 这有助于稳定集群。

这个函数通常在以下几种情况下被调用:

- 当节点的角色发生变化时 (例如, 从跟随者变为候选者)。
- 在处理完某些 RPC 请求后, 需要重置选举定时器以避免过早触发新的选举。
- 在领导者定期发送心跳信息时, 重置选举定时器以延长跟随者的选举超时时间。

〈4〉 投票选举

领导者投票选举过程的思路如下:

- (1) 判断是否是 leader, 因为 leader 是不会收到心跳的, 因此如果是 leader 就没必要再此发起选举;
- (2) 将当前节点的角色修改为候选人 (Candidate), 并根据当前状态创建一个 rpc 参数, 包括任期号 (Term)、候选人 ID (CandidateId)、最后一条日志的索引 (LastLogIndex) 和最后一条日志的任期号 (LastLogTerm);
- (3) 分别创建一个 goroutine 调用 sendRequestVote 函数发送给每一个其它节点, 结果由 grantedChan 进行接收;
- (4) 当前 goroutine 就进入一个循环, 主要分为三种情况:
 - 1) 在候选人角色的情况下, 如果当前收到的投票超过了半数, 就选举成功, 成为 leader;
 - 2) 如果在收到某个节点的回复时, 变成了 follower, 就暂停选举;
 - 3) 当前获取所有节点的回复, 但是还没有超过半数的投票, 就结束选举。

以下是详细代码:

```
1. // 将当前节点的角色修改为候选人  
2. rf.changeRole(Role_Candidate)  
3. DPrintf("%v role %v, start election, term: %v", rf.me, rf.role, rf.currentTerm)  
4. // 获取最后一条日志的任期号和索引  
5. lastLogTerm, lastLogIndex := rf.getLastLogTermAndIndex()  
6. // 准备请求投票的参数  
7. args := RequestVoteArgs{  
8.     CandidateId: rf.me,  
9.     Term:        rf.currentTerm,  
10.    LastLogTerm: lastLogTerm,  
11.    LastLogIndex: lastLogIndex,  
12. }
```



```
13. // 持久化当前状态
14. rf.persist()
15. rf.mu.Unlock()
16. // 初始化计数器
17. allCount := len(rf.peers)
18. grantedCount := 1
19. resCount := 1
20. // 创建用于接收投票结果的通道
21. grantedChan := make(chan bool, len(rf.peers)-1)
22.
23. // 遍历所有节点，发送请求投票 RPC
24. for i := 0; i < allCount; i++ {
25.     if i == rf.me {
26.         continue // 跳过自身
27.     }
28.     // 对每一个其他节点都要发送 RPC
29.     go func(gch chan bool, index int) {
30.         reply := RequestVoteReply{}
31.         rf.sendRequestVote(index, &args, &reply)
32.         gch <- reply.VoteGranted
33.         if reply.Term > args.Term {
34.             rf.mu.Lock()
35.             if reply.Term > rf.currentTerm {
36.                 // 放弃选举
37.                 rf.currentTerm = reply.Term
38.                 rf.changeRole(Role_Follower)
39.                 rf.votedFor = -1
40.                 rf.resetElectionTimer()
41.                 rf.persist()
42.             }
43.             rf.mu.Unlock()
44.         }
45.     }(grantedChan, i)
46. }
47. // 等待投票结果
48. for rf.role == Role_Candidate {
49.     flag := <-grantedChan
50.     resCount++
51.     if flag {
52.         grantedCount++
53.     }
54.     DPrintf("vote: %v, allCount: %v, resCount: %v, grantedCount: %v", flag, allCount, resCount, grantedCount)
```

```
55.
56.     if grantedCount > allCount/2 {
57.         // 竞选成功
58.         rf.mu.Lock()
59.         DPrintf("before try change to leader, count: %d, args: %v, currentTerm: %v, argsTerm: %v",
grantedCount, args, rf.currentTerm, args.Term)
60.         if rf.role == Role_Candidate && rf.currentTerm == args.Term {
61.             rf.changeRole(Role_Leader)
62.         }
63.         if rf.role == Role_Leader {
64.             rf.resetAppendEntriesTimersZero()
65.             rf.persist()
66.             rf.mu.Unlock()
67.             DPrintf("%v current role: %v", rf.me, rf.role)
68.         } else {
69.             rf.mu.Unlock()
70.         }
71.     } else if resCount == allCount || resCount-grantedCount > allCount/2 {
72.         DPrintf("grant fail! grantedCount <= len/2: count: %d", grantedCount)
73.         return
74.     }
75. }
```

以上代码存在两个关键的操作：投票请求和投票回复。

其中，发送请求投票 RPC 到指定的服务器由 `sendRequestVote()` 函数实现，它会尝试 10 次向节点发送 rpc，直到成功或者超过了此次 rpc 超时时间。

```
1. go func() {
2.     for i := 0; i < 10 && !rf.killed(); i++ {
3.         // 尝试向目标服务器发送 RequestVote RPC
4.         ok := rf.peers[server].Call("Raft.RequestVote", args, reply)
5.         if !ok {
6.             continue // 如果调用失败，继续尝试下一次
7.         } else {
8.             ch <- ok // 如果调用成功，将结果发送到通道并返回
9.             return
10.        }
11.    }
12. }()
```

而每一个节点在收到 vote rpc 是要进行相应的处理，这由 `RequestVote()` 函数实现，主要就是进行如下判断：

(1) 如果 `currentTerm > term`，返回 `false`;

- (2) 如果 $term = currentTerm$, 如果该节点为 leader, 返回 false; 如果 $votedFor$ 不为空且不为 $candidateId$, 返回 false; 如果 $votedFor$ 不为空且为 $candidateId$, 返回 true;
- (3) 如果 $currentTerm < term$, 直接更新 $currentTerm$, 变为 Follower 并设 $votedFor$ 为空;
- (4) 判断日志完整性, 如果当前节点的日志完整性大于等于候选人日志完整性 ($lastLogTerm > args.lastLogTerm$ 或者 $lastLogTerm = args.lastLogTerm$ & $lastLogIndex > args.LastLogIndex$), 那么就拒绝投票给它;
- (5) 否者, 就投票给它。

2.3.4 心跳的实现

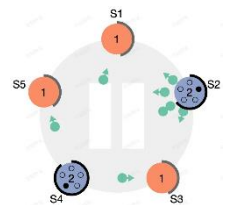
<1> rpc 的参数和回复结构体:

```
1. type AppendEntriesArgs struct {
2.     Term          int //当前任期号。
3.     LeaderId      int
4.     PrevLogIndex  int // 前一个日志条目的索引,
//用于检查日志的连续性。
5.     PrevLogTerm   int
6.     Entries       []LogEntry // 要追加的日志
//条目数组。
7.     LeaderCommit  int
```

```
8. }
9. type AppendEntriesReply struct {
10.     Term          int
11.     Success       bool
12.     NextLogTerm   int // 跟随者下一个期望的
//日志条目的任期号。
13.     NextLogIndex  int // 跟随者下一个期望的日
//志条目的索引。
14. }
```

AppendEntriesArgs 结构体用于封装领导者向跟随者发送日志条目的参数。AppendEntriesReply 结构体用于封装跟随者对领导者日志复制请求的响应。这些基本都是根据论文来实现的。

当向某个节点发送心跳时, 会调用 `sendAppendEntriesToPeer` 向指定的节点发送 rpc:



```
1. func (rf *Raft) sendAppendEntriesToPeer(peerId int) {
2.     if rf.killed() {
3.         return
4.     }
5.     rf.mu.Lock()

6.     if rf.role != Role_Leader { // 如果当前节点不是领导者, 重置 AppendEntries 定时器并解锁返回

7.         rf.resetAppendEntriesTimer(peerId)
8.         rf.mu.Unlock()
9.         return
10.    }
11.    DPrintf("%v send append entries to peer %v", rf.me, peerId)
12. }
```



```

13. prevLogIndex, prevLogTerm, logEntries := rf.getAppendLogs(peerId) // 获取要发送的日志信息

14. args := AppendEntriesArgs{ / 创建 AppendEntries 请求参数

15.     Term:         rf.currentTerm,
16.     LeaderId:     rf.me,
17.     PrevLogIndex: prevLogIndex,
18.     PrevLogTerm:  prevLogTerm,
19.     Entries:      logEntries,
20.     LeaderCommit: rf.commitIndex,
21. }
22. reply := AppendEntriesReply{}
23. rf.resetAppendEntriesTimer(peerId)
24. rf.mu.Unlock()
25. rf.sendAppendEntries(peerId, &args, &reply)
26. DPrintf("%v role: %v, send append entries to peer finish, %v, reply = %v", rf.me, rf.role, args,
reply)
27. rf.mu.Lock()

28. if reply.Term > rf.currentTerm { // 如果回复中的任期号大于当前任期号, 更新任期号并变更角色为跟随者

29.     rf.changeRole(Role_Follower)
30.     rf.currentTerm = reply.Term
31.     rf.resetElectionTimer()
32.     rf.persist()
33.     rf.mu.Unlock()
34.     return
35. }

```

在 2A 部分中, 当前要做的处理很简单, 就是创建一个 AppendEntriesArgs, 并调用 sendAppendEntries 函数进行发送, 而对于发送的结果我们并不需要管, 因此要实现心跳, 并不需要日志, 因此也就不需要进行判断了。通过这种方式, sendAppendEntriesToPeer 函数确保了领导者能够定期向跟随者发送心跳信息, 以维持其领导地位并确保日志的一致性。

其中 sendAppendEntries 函数用于向指定的服务器发送 AppendEntries RPC, 用于日志复制。它会尝试 10 次向节点发送 rpc, 直到成功或者超过了此次 rpc 超时时间。

```

1. go func() {
2.     // 尝试 10 次
3.     for i := 0; i < 10 && !rf.killed(); i++ {
4.         ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
5.         if !ok {
6.             time.Sleep(time.Millisecond * 10) // 如果调用失败, 等待 10 毫秒后重试
7.             continue
8.         } else {

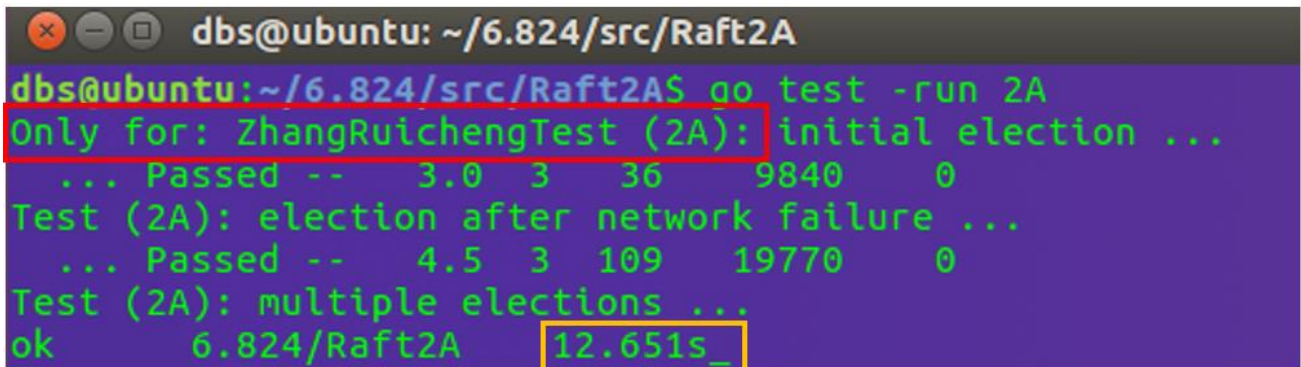
```

```
9.         ch <- ok // 如果调用成功，将结果发送到通道并返回
10.        return
11.    }
12. }
13. }()
```

而每一个节点在收到 AppendEntries rpc 后，要进行相应的处理。仅仅是心跳的话，接收方实现很简单，只需要判断一下 term 其实就可以了，并重置选举定时器。

```
1. reply.Term = rf.currentTerm // 设置回复中的任期号为当前任期号
2.
3. // 如果请求中的任期号小于当前任期号，解锁并返回
4. if args.Term < rf.currentTerm {
5.     rf.mu.Unlock()
6.     return
7. }
8.
9. // 如果请求中的任期号大于当前任期号，更新当前任期号并变更角色为跟随者
10. rf.currentTerm = args.Term
11. rf.changeRole(Role_Follower) // 变更角色为跟随者
```

2.4 Lab2A-测试结果



```
db5@ubuntu: ~/6.824/src/Raft2A
db5@ubuntu:~/6.824/src/Raft2AS go test -run 2A
Only for: ZhangRuichengTest (2A): initial election ...
... Passed -- 3.0 3 36 9840 0
Test (2A): election after network failure ...
... Passed -- 4.5 3 109 19770 0
Test (2A): multiple elections ...
ok 6.824/Raft2A 12.651s_
```

Lab2A 的测试结果为 12.651 s

3. Lab2B-Log Replication-日志复制

3.1 任务要求:

在 2A 的基础上实现日志复制功能。在 2B 部分相比于 2A 部分，需要处理的条件判断更多，容易出错的地方也相应增多。有时在编写条件判断时，可能会因为疏忽而忘记在某些情况下解锁！

3.2 任务分析:

这一部分的核心任务是实现日志复制功能。实际上, 在 2A 部分中, 我们已经利用 AppendEntries RPC 实现了基本的心跳机制。在此基础上, 我们需要完成以下三个主要部分:

1. **扩展心跳功能:** 在 AppendEntries RPC 的心跳功能基础上, 增加日志的发送和接收功能。这意味着领导者不仅需要定期发送心跳信息, 还需要将日志条目发送给跟随者, 以确保日志的一致性。
2. **实现 Start(command interface{}) 函数:** 该函数接受一个 command 参数。如果当前节点是领导者, 它将该 command 添加到日志中, 并立即发送一次 AppendEntries RPC 来同步日志。这一步是确保新日志条目能够及时复制到所有跟随者的关键。
3. **日志接收情况的判断:** 需要对日志的接收情况进行判断。当满足多数派(即超过半数的跟随者)确认接收了日志条目时, 领导者将该日志条目提交到本地状态机, 并通过 ApplyCh 发送消息以应用该日志条目。这一步确保了日志条目的持久化和应用。

3.3 代码分析:

3.3.1 Raft 基本函数-2B 部分

此部分最主要的函数就是 start 函数和对 ApplyCh 的处理:

```
//客户端请求, leader添加日志
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    index := -1
    term := -1
    isleader := false
    // Your code here (2B).
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if rf.role != Role_Leader {
        return index, term, isleader
    }

    rf.logs = append(rf.logs, LogEntry{
        Term: rf.currentTerm,
        Command: command,
    })
    _, lastIndex := rf.getLastLogTermAndIndex()
    index = lastIndex
    rf.matchIndex[rf.me] = lastIndex
    rf.nextIndex[rf.me] = lastIndex + 1

    term = rf.currentTerm
    isleader = true

    DPrintf("%v start command %v %v: %v", rf.me, rf.lastSnapshotIndex, lastIndex, command)

    rf.resetAppendEntriesTimersZero()

    return index, term, isleader
}
```

Start 函数的主要作用是首先判断当前节点是否为领导者(Leader)。只有领导者节点才有权限接收并处理新的命令(command), 并将这些命令写入到日志中。完成日志写入后, 该



函数会重置 AppendEntries RPC 的定时器，并立即发送 AppendEntries RPC 以确保日志条目能够及时地复制到所有的跟随者节点上。

以下是该过程的详细描述：

1. **领导者判断：**函数开始时首先检查当前节点的角色。如果当前节点不是领导者，则无法处理新的命令，函数直接返回。
2. **命令写入日志：**如果当前节点是领导者，那么它将接收到的命令添加到本地日志中。这一步骤确保了所有新的命令首先被记录和存储。
3. **重置和发送 AppendEntries RPC：**
 - 写入日志后，领导者会重置用于触发 AppendEntries RPC 的定时器。这是为了确保在写入新日志后，能够尽快地将这些变更同步到集群中的其他节点。
 - 重置定时器后，领导者会立即发送 AppendEntries RPC 给所有的跟随者。这一步是日志复制过程的关键，它确保了所有节点上的日志保持一致。

接下来再看下 ApplyCh 的处理：

```
func (rf *Raft) ticker() {
    go func() {
        for {
            select {
            case <-rf.stopCh:
                return
            case <-rf.applyTimer.C:
                rf.notifyApplyCh <- struct{}{}
            case <-rf.notifyApplyCh: //当有日志记录提交了，要进行应用
                rf.startApplyLogs()
            }
        }
    }()
}
```

在 raft 的 ticker 中，其中创建的一个 goroutine 就是用来处理 ApplyCh，每隔一段时间、notifyApplyCh 有消息就调用一次 startApplyLogs 来处理新的提交日志：

```
//处理要应用的日志，快照的命令比较特殊，不在此处提交
func (rf *Raft) startApplyLogs() {
    defer rf.applyTimer.Reset(ApplyInterval)

    rf.mu.Lock()
    var msgs []ApplyMsg
    if rf.lastApplied < rf.lastSnapshotIndex {
        //此时要安装快照，命令在接收到快照时就发布过了，等待处理
        msgs = make([]ApplyMsg, 0)
        rf.mu.Unlock()
        rf.InstallSnapshot(rf.lastSnapshotTerm, rf.lastSnapshotIndex, rf.persister.snapshot)
        return
    } else if rf.commitIndex <= rf.lastApplied {
        // snapshot 没有更新 commitid
        msgs = make([]ApplyMsg, 0)
    } else {
        msgs = make([]ApplyMsg, 0, rf.commitIndex-rf.lastApplied)
        for i := rf.lastApplied + 1; i <= rf.commitIndex; i++ {
            msgs = append(msgs, ApplyMsg{
                CommandValid: true,
                Command:      rf.logs[rf.getStoreIndexByLogIndex(i)].Command,
                CommandIndex: i,
            })
        }
    }
    rf.mu.Unlock()
}
```

核心处理流程主要涉及比较 `commitIndex` 和 `lastApplied` 两个索引值。如果发现有新的已提交日志尚未应用，那么就依次将这些日志发送到 `applyCh` 通道，以便进行进一步的应用处理。

在此过程中，有一个关于快照的判断步骤，但在实现中可以忽略不计。这是因为在我的实现中，快照命令与日志中记录的命令是两种完全不同的命令类型，它们不会占用日志的索引。当跟随者 (Follower) 接收到一份快照时，它会单独地将快照命令发送给 `applyCh` 进行处理。这种设计确保了日志命令和快照命令在系统中被清晰地区分和独立处理，从而简化了日志管理和状态机应用的流程。

3.3.2 append entries rpc 参数

`append entries` rpc 参数如下，都是根据论文来设计的：

```
1. type AppendEntriesArgs struct {
2.     Term          int
3.     LeaderId      int
4.     PrevLogIndex  int
5.     PrevLogTerm   int
6.     Entries       []LogEntry
7.     LeaderCommit  int
8. }
9. type AppendEntriesReply struct {
10.    Term          int
11.    Success       bool
12.    NextLogTerm   int
13.    NextLogIndex  int
14. }
```

在介绍 rpc 的发送和接收的相关处理之前，先来讲下要用到的几个函数：

```
1. // 重置所有 AppendEntries 定时器
2. func (rf *Raft) resetAppendEntriesTimersZero() {
3.     for _, timer := range rf.appendEntriesTimers {
4.         timer.Stop() // 停止定时器
5.         timer.Reset(0) // 重置定时器为 0，表示立即发送心跳
6.     }
7. }
8. // 重置特定跟随者的 AppendEntries 定时器
9. func (rf *Raft) resetAppendEntriesTimerZero(peerId int) {
10.    rf.appendEntriesTimers[peerId].Stop() // 停止定时器
11.    rf.appendEntriesTimers[peerId].Reset(0) // 重置定时器为 0，表示立即发送心跳
12. }
13. // 重置单个 AppendEntries 定时器
14. func (rf *Raft) resetAppendEntriesTimer(peerId int) {
15.    rf.appendEntriesTimers[peerId].Stop() // 停止定时器
16.    rf.appendEntriesTimers[peerId].Reset(HeartBeatInterval) // 重置定时器为心跳间隔时间
17. }
```

这三个函数主要是在不同的情况下进行调用来重置全部或某个节点的 `AppendEntries` 定时器。比如：

- (1) 在 start 中，leader 接收到一条 command，需要调用 resetAppendEntriesTimersZero 重置对所有节点的定时器，立马发送该日志；
- (2) 在 leader 发送一次 rpc 后，如果收到节点的回复信息，发现该节点并没有接收 log 的日志，且 NextLogIndex 表明了要接受的下一条日志，为了一致性，需要调用 resetAppendEntriesTimerZero 立马在进行一次发送；
- (3) 在 leader 发送一次 rpc 后，就需要调用 resetAppendEntriesTimer 重置对该节点的下一次 rpc 发送间隔时间。

```
1. // 判断当前 raft 的日志记录是否超过发送过来的日志记录
2. func (rf *Raft) isOutOfArgsAppendEntries(args *AppendEntriesArgs) bool {
3.     argsLastLogIndex := args.PrevLogIndex + len(args.Entries)
4.     lastLogTerm, lastLogIndex := rf.getLastLogTermAndIndex()
5.     if lastLogTerm == args.Term && argsLastLogIndex < lastLogIndex {
6.         return true
7.     }
8.     return false
9. }
10. // 获取当前存储位置的索引
11. func (rf *Raft) getStoreIndexByLogIndex(logIndex int) int {
12.     storeIndex := logIndex - rf.lastSnapshotIndex
13.     if storeIndex < 0 {
14.         return -1
15.     }
16.     return storeIndex
17. }
```

isOutOfArgsAppendEntries 函数函数的主要目的是在相同任期 (term) 的情况下，判断当前节点的最后一条日志的索引是否大于领导者发送过来的最后一条日志的索引。这个函数在日志复制过程中非常重要，因为它帮助确定当前节点的日志是否已经包含了领导者发送的所有日志。如果当前节点的日志索引更大，这意味着当前节点的日志已经覆盖了领导者发送的日志，从而可以避免不必要的日志复制。

getStoreIndexByLogIndex 函数的作用是根据日志的索引来获取在日志数组 (logs) 中真正的存储索引。由于日志数组中可能会包含快照 (snapshot)，这些快照不占用日志索引，因此需要通过计算来找到实际的存储位置。这个函数通过从日志索引中减去最后快照的索引来实现这一点。如果计算出的存储索引小于 0，表示该日志索引无效，函数返回 -1；否则，返回计算出的存储索引。

3.3.2 append entries rpc-回复部分

先来看一下 rpc 的发送函数：

```
func (rf *Raft) sendAppendEntriesToPeer(peerId int) {
    if rf.killed() {
        return
    }

    rf.mu.Lock()
    if rf.role != Role_Leader {
        rf.resetAppendEntriesTimer(peerId)
        rf.mu.Unlock()
        return
    }
    DPrintf("%v send append entries to peer %v", rf.me, peerId)

    prevLogIndex, prevLogTerm, logEntries := rf.getAppendLogs(peerId)
    args := AppendEntriesArgs{
        Term:      rf.currentTerm,
        LeaderId:   rf.me,
        PrevLogIndex: prevLogIndex,
        PrevLogTerm: prevLogTerm,
        Entries:    logEntries,
        LeaderCommit: rf.commitIndex,
    }
    reply := AppendEntriesReply{}
    rf.resetAppendEntriesTimer(peerId)
    rf.mu.Unlock()

    rf.sendAppendEntries(peerId, &args, &reply)

    DPrintf("%v role: %v, send append entries to peer finish,%v,args = %v,reply = %v",
        rf.me, rf.role, rf.me, args, reply)

    rf.mu.Lock()
    if reply.Term > rf.currentTerm {
        rf.changeRole(Role_Follower)
        rf.currentTerm = reply.Term
        rf.resetElectionTimer()
        rf.persist()
        rf.mu.Unlock()
        return
    }
}
```

主要步骤如下：

- (1) 如果当前节点不是 leader，则重置发送时间并返回；
- (2) 根据当前的信息生成 AppendEntriesArgs，其中要根据 getAppendLogs 函数来获取要发送节点的日志信息：prevLogIndex, prevLogTerm, logEntries；
- (3) 调用 sendAppendEntries 函数发送 rpc；
- (4) 根据 rpc 的返回结果，进行相应的处理：
 - 1) 如果返回的 term 大于当前节点的 term，就立马切换成 follower；
 - 2) 如果相应成功了，说明发送的数据全部接收了，或者根本没有数据，如果有发送数据，可以修改 nextIndex 和 matchIndex，以及调用 tryCommitLog 尝试提交一次日志；
 - 3) 如果相应失败了，且 NextLogIndex 不为 0，则判断是立马再发送一次 rpc 还是发送一次快照(sendInstallSnapshotToPeer)；如果 NextLogIndex 为 0，则不进行处理，此时如果插入会导致乱序。

上面涉及了三个函数调用，具体看下：


```
//获取要向指定节点发送的日志
func (rf *Raft) getAppendLogs(peerId int) (prevLogIndex int, prevLogTerm int, logEntries []LogEntry) {
    nextIndex := rf.nextIndex[peerId]
    lastLogTerm, lastLogIndex := rf.getLastLogTermAndIndex()
    if nextIndex <= rf.lastSnapshotIndex || nextIndex > lastLogIndex {
        //没有要发送的log
        prevLogTerm = lastLogTerm
        prevLogIndex = lastLogIndex
        return
    }
    //这里一定要进行深拷贝，不然会和Snapshot()发生数据上的冲突
    //logEntries = rf.logs[nextIndex-rf.lastSnapshotIndex:]
    logEntries = make([]LogEntry, lastLogIndex-nextIndex+1)
    copy(logEntries, rf.logs[nextIndex-rf.lastSnapshotIndex:])
    prevLogIndex = nextIndex - 1
    if prevLogIndex == rf.lastSnapshotIndex {
        prevLogTerm = rf.lastSnapshotTerm
    } else {
        prevLogTerm = rf.logs[prevLogIndex-rf.lastSnapshotIndex].Term
    }
    return
}
```

getAppendLogs 是获取要发送给对应节点的日志信息。如果下一条信息的 index 小于快照所包含的最后一条日志 index，或者大于最后一条日志 index，则说明没有要发送的日志；否则，进行计算。

```
//发送端发送数据
func (rf *Raft) sendAppendEntries(server int, args *AppendEntriesArgs, reply *AppendEntryResponse) {
    rpcTimer := time.NewTimer(RPCTimeout)
    defer rpcTimer.Stop()

    ch := make(chan bool, 1)
    go func() {
        //尝试10次
        for i := 0; i < 10 && !rf.killed(); i++ {
            ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
            if !ok {
                time.Sleep(time.Millisecond * 10)
                continue
            } else {
                ch <- ok
                return
            }
        }
    }()

    select {
    case <-rpcTimer.C:
        DPrintf("%v role: %v, send append entries to peer %v TIME OUT!!!", rf.me, rf.role, server)
        return
    case <-ch:
        return
    }
}
```

sendAppendEntries 会尝试 10 次向节点发送 rpc，直到成功或者超过了此次 rpc 超时时间。

```
//尝试去提交日志
//会依次判断，可以提交多个，但不能有间断
func (rf *Raft) tryCommitLog() {
    _, lastLogIndex := rf.getLastLogTermAndIndex()
    hasCommit := false

    for i := rf.commitIndex + 1; i <= lastLogIndex; i++ {
        count := 0
        for _, m := range rf.matchIndex {
            if m >= i {
                count += 1
            }
        }
        //提交数达到多数派
    }
}
```

```
//提交数达到多数派
if count > len(rf.peers)/2 {
    rf.commitIndex = i
    hasCommit = true
    DPrintf("%v role: %v,commit index %v", rf.me, rf.role, i)
    break
}

if rf.commitIndex != i {
    break
}

if hasCommit {
    rf.notifyApplyCh <- struct{}{}
}
```

tryCommitLog 是从已经提交的 index+1 开始遍历后面的日志,只要一个节点的 matchIndex 大于等于当前日志的 index, 就表明该节点已经接收了该日志, 如果日志达到了多数派, 就可以提交了。并且将新提交的日志进行应用。遍历的过程中, 只要有一条日志没有达到多数派, 后面的日志也可以不用计算了。

最后一个部分就是接收方的处理;

```
//接收端处理rpc
//主要进行三个处理:
// 1. 判断任期
// 2. 判断是否接收数据, success: 数据全部接受, 或者根本就没有数据
// 3. 判断是否提交数据
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    DPrintf("%v receive a appendEntries: %v", rf.me, args)
    reply.Term = rf.currentTerm
    if args.Term < rf.currentTerm {
        rf.mu.Unlock()
        return
    }
    rf.currentTerm = args.Term
    rf.changeRole(Role_Follower)
    rf.resetElectionTimer()

    _, lastLogIndex := rf.getLastLogTermAndIndex()
    //先判断两边, 再判断刚好从快照开始, 再判断中间的情况
    if args.PrevLogIndex < rf.lastSnapshotIndex {
        //1. 要插入的前一个index小于快照index, 几乎不会发生
        reply.Success = false
        reply.NextLogIndex = rf.lastSnapshotIndex + 1
    } else if args.PrevLogIndex > lastLogIndex {
        //2. 要插入的前一个index大于最后一个log的index, 说明中间还有log
        reply.Success = false
        reply.NextLogIndex = lastLogIndex + 1
    } else if args.PrevLogIndex == rf.lastSnapshotIndex {
        //3. 要插入的前一个index刚好等于快照的index, 说明可以全覆盖, 但要判断是否是全覆盖
        if rf.isOutOfArgsAppendEntries(args) {
            reply.Success = false
            reply.NextLogIndex = 0 //0代表插入会导致乱序
        } else {
            reply.Success = true
            rf.logs = append(rf.logs[:1], args.Entries...)
            _, currentLogIndex := rf.getLastLogTermAndIndex()
            reply.NextLogIndex = currentLogIndex + 1
        }
    } else if args.PrevLogTerm == rf.logs[rf.getStoreIndexByLogIndex(args.PrevLogIndex)].Term {
        //4. 要插入的前一个index等于快照的index, 且term等于快照的term, 说明可以全覆盖
        reply.Success = true
        rf.logs = append(rf.logs[:1], args.Entries...)
        _, currentLogIndex := rf.getLastLogTermAndIndex()
        reply.NextLogIndex = currentLogIndex + 1
    }
}
```

```
//4. 中间的情况: 索引处的两个term相同
if rf.isOutOfArgsAppendEntries(args) {
    reply.Success = false
    reply.NextLogIndex = 0
} else {
    reply.Success = true
    rf.logs = append(rf.logs[:rf.getStoreIndexByLogIndex(args.PrevLogIndex)+1], args)
    _, currentLogIndex := rf.getLastLogTermAndIndex()
    reply.NextLogIndex = currentLogIndex + 1
}
} else {
    //5. 中间的情况: 索引处的两个term不相同, 跳过一个term
    term := rf.logs[rf.getStoreIndexByLogIndex(args.PrevLogIndex)].Term
    index := args.PrevLogIndex
    for index > rf.commitIndex && index > rf.lastSnapshotIndex && rf.logs[rf.getStoreIndexByLogIndex(index)].Term == term {
        index--
    }
    reply.Success = false
    reply.NextLogIndex = index + 1
}

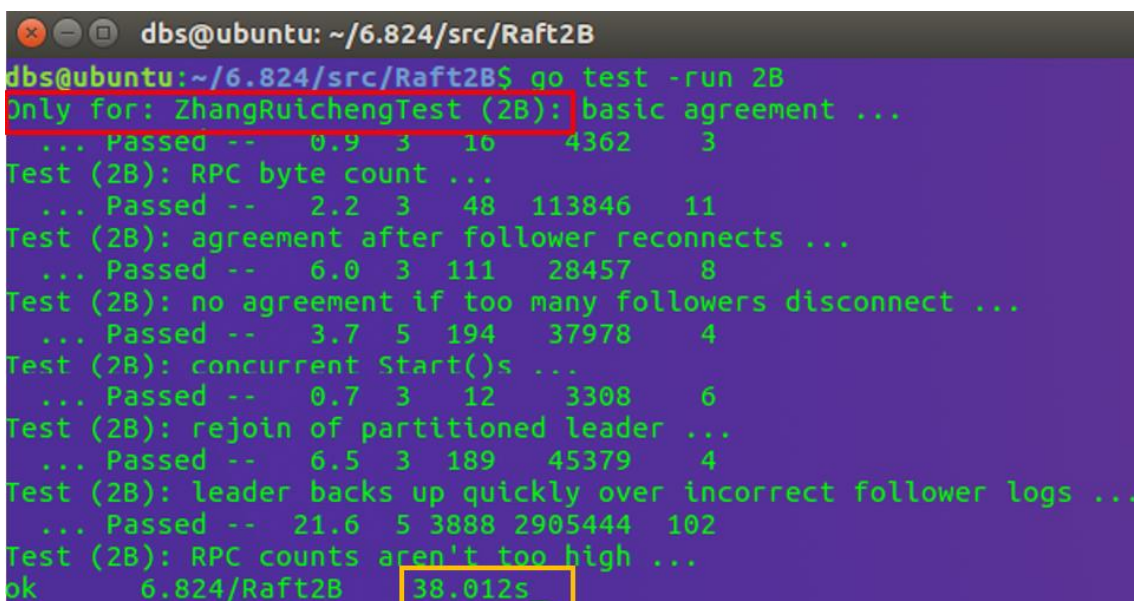
if reply.Success {
    DPrintf("%v current commit: %v, try to commit %v", rf.me, rf.commitIndex, args.LeaderCommit)
    if rf.commitIndex < args.LeaderCommit {
        rf.commitIndex = args.LeaderCommit
        rf.notifyApplyCh <- struct{}{}
    }
}

rf.persist()
DPrintf("%v role: %v, get appendentries finish,args = %v,reply = %v", rf.me, rf.role, args, reply)
rf.mu.Unlock()
}
```

主要分为三个步骤:

1. 判断任期, 如果当前 rpc 的 term 小于自身的 term, 可以不进行处理;
2. 判断是否接收数据, 返回 success 表明数据全部接受, 或者根本就没有数据。分为以下五种情况:
3. 判断是否提交数据, 如果自身记录的 commitIndex 小于 leader 记录的 commitIndex, 就可以更新 commitIndex, 然后发送命令到 ApplyCh, 表示可以应用了。

3.4 Lab2B-测试结果



```
db@ubuntu: ~/6.824/src/Raft2B
db@ubuntu:~/6.824/src/Raft2B$ go test -run 2B
Only for: ZhangRuichengTest (2B): basic agreement ...
... Passed -- 0.9 3 16 4362 3
Test (2B): RPC byte count ...
... Passed -- 2.2 3 48 113846 11
Test (2B): agreement after follower reconnects ...
... Passed -- 6.0 3 111 28457 8
Test (2B): no agreement if too many followers disconnect ...
... Passed -- 3.7 5 194 37978 4
Test (2B): concurrent Start()'s ...
... Passed -- 0.7 3 12 3308 6
Test (2B): rejoin of partitioned leader ...
... Passed -- 6.5 3 189 45379 4
Test (2B): leader backs up quickly over incorrect follower logs ...
... Passed -- 21.6 5 3888 2905444 102
Test (2B): RPC counts aren't too high ...
ok      6.824/Raft2B      38.012s
```

Lab2B 的测试结果为 38.012s

4. Lab2C-Persistence-持久化

4.1 任务要求:

如果一个基于 Raft 的服务器重新启动, 它应该从中断的地方恢复服务。这要求 Raft 在重启后保持持久状态。

一个真正的实现是将 Raft 的持久状态在每次更改时写入磁盘, 并在重新启动后重新启动时从磁盘读取状态。但是在我们的实现中, 并不会真正的在磁盘中进行操作; 相反, 我们将从 `Persister` 对象保存和恢复持久状态 (请参阅 `persister.go`)。 `Persister` 最初保存了 Raft 最近持久化的状态 (如果有的话)。 Raft 应该从那个 `Persister` 初始化它的状态, 并且应该在每次状态改变时使用它来保存它的持久状态。使用 `Persister` 的 `ReadRaftState()` 和 `SaveRaftState()` 方法。

4.2 任务分析:

在 2C 里我们的任务就是通过添加代码来保存和恢复持久状态:

1. 完成 `raft.go` 中的 `persist()` 和 `readPersist()` 函数: `readPersist()`, 当节点重启时, 会重新读取状态恢复; `persist()`, 将状态存储下来 (实际工作中会写到硬盘, Lab 里写入内存);
2. 在状态改变的时候, 调用 `persist()` 函数存储状态。

我们需要将状态编码 (或 “序列化”) 为字节数组, 以便将其传递给 `Persister`; 使用 `labgob` 编码器; 具体的使用方法可以查看官方给出的 `persist()` 和 `readPersist()` 中的注释。

Lab2C 真正难的地方在于它严苛的测试, 涉及到节点的反复选举、宕机与重新上线、状态恢复、网络断开、日志未提交等等问题。2A 和 2B 中很难测出的错误很有可能在 2C 的测试里暴露出来。(深有体会!)

4.3 代码详解:

代码部分其实很简单, 首先来看下 `Persist` 的数据结构:

```
1. type Persister struct {  
2.     mu      sync.Mutex  
3.     raftstate []byte // 存储当前 raft 的状态  
4.     snapshot []byte // 存储当前的快照  
5. }
```

一共是三个属性: 互斥锁 `mu`、当前 raft 的状态、当前的快照。以这一个构造体来存储持久化数据, 代替了实际上的磁盘 IO。

该部分主要内容就是 `persist()` 和 `readPersist()` 这两个函数:

`persist()`:

```
1. // 保存持久化当前 Raft 状态  
2. func (rf *Raft) persist() {  
3.     data := rf.getPersistData() // 获取需要持久化的数据  
4.     rf.persister.SaveRaftState(data) // 保存 Raft 状态  
5. }
```

这个函数用于将当前 Raft 节点的状态持久化到存储中。它首先调用 `getPersistData` 获取需要持久化的数据，然后调用 `Persister` 结构的 `SaveRaftState` 方法将数据保存到持久化存储中。

`readPersist()`:

```
1. // 读取持久化数据
2. func (rf *Raft) readPersist(data []byte) {
3.     if data == nil || len(data) < 1 { // 如果没有持久化数据，可能是启动时没有状态
4.         return
5.     }
6.     r := bytes.NewBuffer(data)
7.     d := labgob.NewDecoder(r)
8.
9.     var (
10.         currentTerm    int
11.         votedFor       int
12.         logs            []LogEntry
13.         commitIndex    int
14.         lastSnapshotTerm int
15.         lastSnapshotIndex int
16.     )
17.
18.     if d.Decode(&term) != nil ||
19.        d.Decode(&votedFor) != nil ||
20.        d.Decode(&commitIndex) != nil ||
21.        d.Decode(&lastSnapshotIndex) != nil ||
22.        d.Decode(&lastSnapshotTerm) != nil ||
23.        d.Decode(&logs) != nil {
24.         log.Fatal("rf read persist err!") // 如果读取过程中出现错误，记录错误并退出
25.     } else {
26.         rf.currentTerm = currentTerm
27.         rf.votedFor = votedFor
28.         rf.commitIndex = commitIndex
29.         rf.lastSnapshotIndex = lastSnapshotIndex
30.         rf.lastSnapshotTerm = lastSnapshotTerm
31.         rf.logs = logs // 更新日志
32.     }
33. }
```

这个函数用于从持久化存储中读取 Raft 节点的状态。它首先检查数据是否有效，然后使用 `labgob.NewDecoder` 解码读取持久化的数据。如果解码过程中出现错误，记录错误并退出；否则，更新 Raft 节点的状态，包括任期号、投票信息、提交索引、快照索引和日志。

在我们持久化的状态任何一个改变的时候，我们就需要调用 `persist()` 来进行持久化，这里涉及之处很多，就不一一列出了，举几个例子：

1. 在接收到 `RequestVote` rpc 时，如果 rpc 的 `term` 大于当前节点的 `term`，就要修改状态，并且如果要投票，也要修改一次状态：

```
if rf.currentTerm < args.Term {
    rf.currentTerm = args.Term
    rf.changeRole(Role_Follower)
    rf.votedFor = -1
    reply.Term = rf.currentTerm
    rf.persist()
}

//判断日志完整性
if lastLogTerm > args.LastLogTerm || (lastLogTerm == args.LastLogTerm && lastLogIndex
    return
}

rf.votedFor = args.CandidateId
reply.VoteGranted = true
rf.changeRole(Role_Follower)
rf.resetElectionTimer()
rf.persist()
DPrintf("%v role: %v voteFor: %v", rf.me, rf.role, rf.votedFor)
```

在发送 `AppendEntries` rpc，收到回复，要修改回复节点的 `nextIndex`、`matchIndex`，可能还要修改 `commitIndex`，因此要进行持久化一次：

```
//响应：成功了，即：发送的数据全部接收了，或者根本没有数据
if reply.Success {
    if reply.NextLogIndex > rf.nextIndex[peerId] {
        rf.nextIndex[peerId] = reply.NextLogIndex
        rf.matchIndex[peerId] = reply.NextLogIndex - 1
    }
    if len(args.Entries) > 0 && args.Entries[len(args.Entries)-1].Term == rf.currentTerm
        //每个leader只能提交自己任期的日志
        rf.tryCommitLog()
    }
    rf.persist() //持久化
    rf.mu.Unlock()
    return
}
```

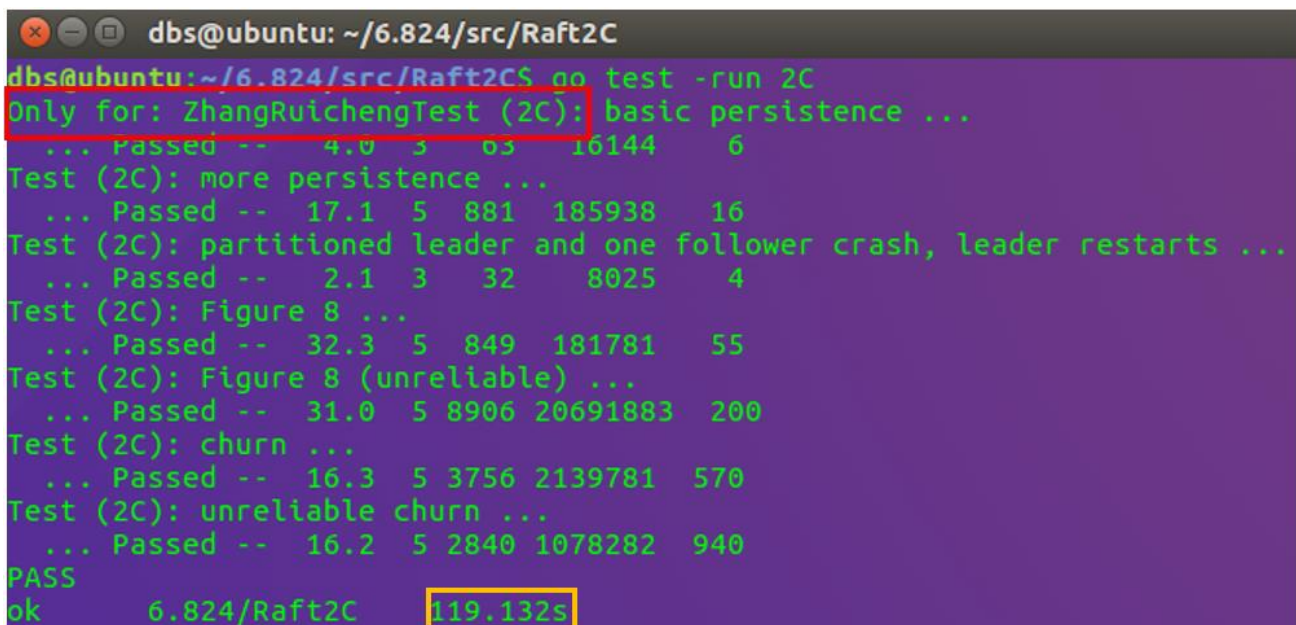
而在当前节点启动后，也要读取一次持久化的状态来进行恢复，具体的调用点并不固定，只需要在持久化相关属性初始化后就可以了：

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    DPrintf("make a raft,me: %v", me)
    rf := &Raft{}
    rf.peers = peers
    rf.persister = persister
    rf.me = me

    // Your initialization code here (2A, 2B, 2C).
    rf.role = Role_Follower
    rf.currentTerm = 0
    rf.votedFor = -1
    rf.logs = make([]*LogEntry, 1) //下标为0存储快照
    // initialize from state persisted before a crash
    rf.commitIndex = 0
    rf.lastApplied = 0
    rf.nextIndex = make([]int, len(rf.peers))
    rf.matchIndex = make([]int, len(rf.peers))
    //读取持久化数据
    rf.readPersist(persister.ReadRaftState())

    rf.electionTimer = time.NewTimer(rf.getElectionTimeout())
    rf.appendEntriesTimers = make([]*time.Timer, len(rf.peers))
    for i := 0; i < len(rf.peers); i++ {
        rf.appendEntriesTimers[i] = time.NewTimer(HeartBeatInterval)
    }
}
```

4.4 Lab2C-测试结果



```
dbcs@ubuntu: ~/6.824/src/Raft2C
dbcs@ubuntu:~/6.824/src/Raft2C$ go test -run 2C
Only for: ZhangRuichengTest (2C): basic persistence ...
... Passed -- 4.0 3 03 16144 6
Test (2C): more persistence ...
... Passed -- 17.1 5 881 185938 16
Test (2C): partitioned leader and one follower crash, leader restarts ...
... Passed -- 2.1 3 32 8025 4
Test (2C): Figure 8 ...
... Passed -- 32.3 5 849 181781 55
Test (2C): Figure 8 (unreliable) ...
... Passed -- 31.0 5 8906 20691883 200
Test (2C): churn ...
... Passed -- 16.3 5 3756 2139781 570
Test (2C): unreliable churn ...
... Passed -- 16.2 5 2840 1078282 940
PASS
ok      6.824/Raft2C      119.132s
```

Lab2C 的测试结果为 119.132s

5. 实验心得

这次实验不仅是对分布式系统和计算机网络的深入探索，更是一次对容错性、一致性保障和系统性能深层次理解的旅程。在开始编写和调试 Raft 代码时，发现真的好难，无论是对论文的理解，还是对系统实现，都有些有心无力。还好有时代福利，可以参考其他人的代码和解

决方法。如老师所言，别人的终究是别人的，参考完还是一头雾水。最后，我决定，自己动手，在结合网络上的视频资料，大概理解了八成左右。以下是我的一些实验心得与感受：

1. 通过这次实验，我不仅加深了对 Raft 协议的理解，还掌握了 Go 语言中信号通道（chan）的使用。在分布式系统中，理解和运用这样的并发机制对于实现高效、可靠的通信至关重要。
2. 在实现 Raft 算法时，我注意到一些关键的实现细节对算法的稳定性和性能有显著影响。特别是在 Candidate 转变为 Leader 的过程中，立即发送心跳信号或追加日志请求可能会导致系统达成一致性协议的困难。这一发现提示我在设计状态转换和消息传递策略时需要更加谨慎。
3. 在实现票数计算和日志提交数计算时，我意识到在并发环境下对共享变量进行操作的复杂性。为了保证计数的准确性，我采用了互斥锁或 Go 语言提供的原子操作函数。这一实践强调了在分布式系统设计中同步和一致性保证的重要性。

这次实验、我学会了如何在复杂的分布式环境中处理并发、同步和状态一致性问题。通过实际编码和问题解决，我对分布式系统设计的微妙之处有了更深的认识。此外，我还意识到了代码优化和性能调整的重要性，这对于构建高效、可靠的分布式系统至关重要。总体来说，这次实验是一个极具价值的学习经历，它加深了我的技术知识，思考一个整体系统的能力。

6. 相关参考资料

1. https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md
2. <https://raft.github.io/raft.pdf>
3. <https://pdos.csail.mit.edu/6.824/papers/mapreduce.pdf>
4. <https://github.com/OneSizeFitsQuorum/MIT6.824-2021>
5. <https://github.com/chaozh/MIT-6.824>
6. <https://www.bilibili.com/video/av87684880>
7. <https://shimo.im/docs/xwqvh3kGppJKvHvX?fallback=1>