基于Raft选举服务实验报告

实验目的

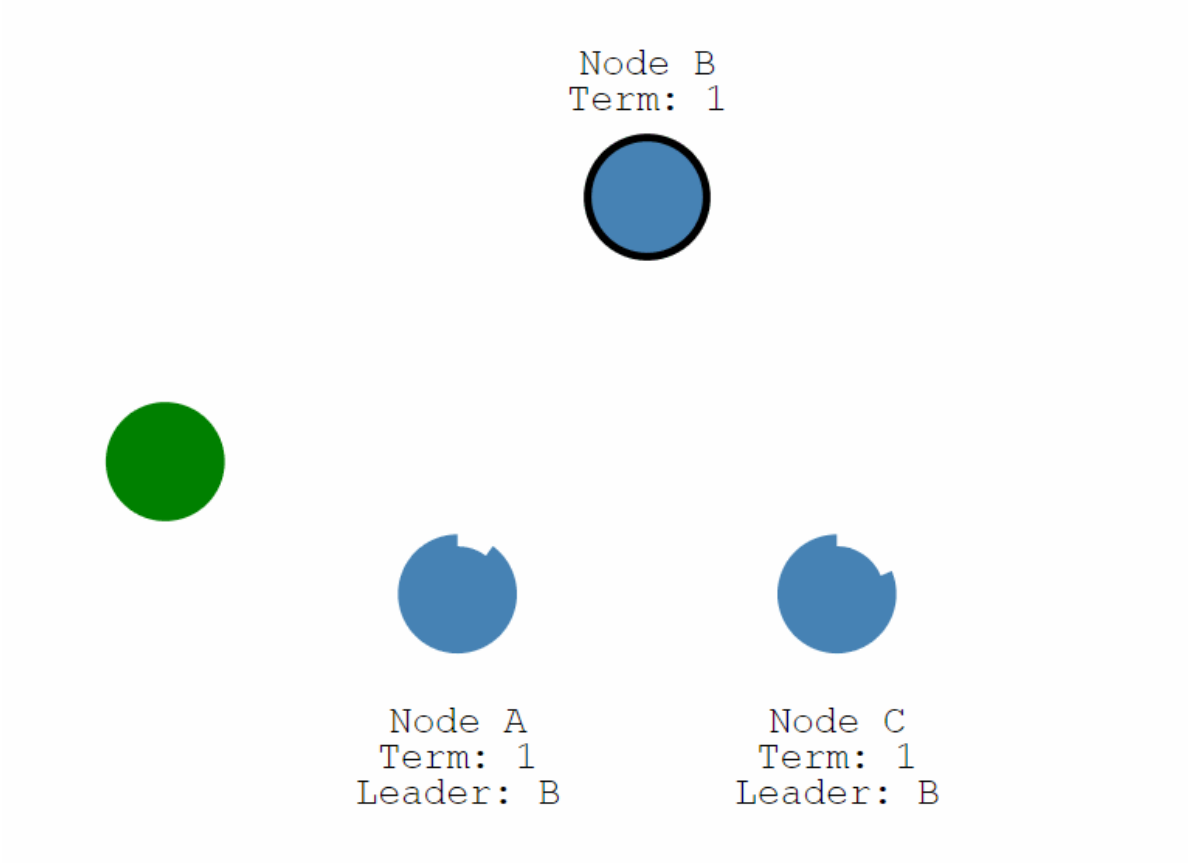
本次实验的目的是通过深入了解Raft算法原理，掌握其设计和实现方法。Raft算法是一种一致性算法，旨在通过确保在分布式系统中多个节点之间的数据一致性，从而保障系统的高可用性与容错性。本实验通过go语言实现一个基于Raft的选举服务，模拟了实现初始选举，以及实现Leader挂掉后重新选举。

实验内容

技术原理

Raft原理

Raft中使用日志来记录所有操作，所有节点都有自己的日志列表来记录所有请求。算法将机器分成三种角色，分别为Leader，Follower和Candidate。正常情况下只存在一个Leader，其余均为Follower，所有客户端都与Leader进行交互。



该过程采用类似两阶段提交的机制。在接收到客户端请求后，Leader 不立即执行操作，而是先将操作记录到自己的日志中，并将该操作传播给所有的 Follower。Follower 接收到请求后，也仅将操作记录到自己的日志中，并返回给 Leader。只有当超过半数的节点确认写入日志后，Leader 才会将操作提交并返回结果给客户端，同时通知其他节点也执行提交操作。通过这种方式，确保了操作一旦提交，就会在大多数节点的日志中留下记录，从而避免了数据丢失的风险。

由此我们可知，Raft将共识问题分解为三个问题，分别为：

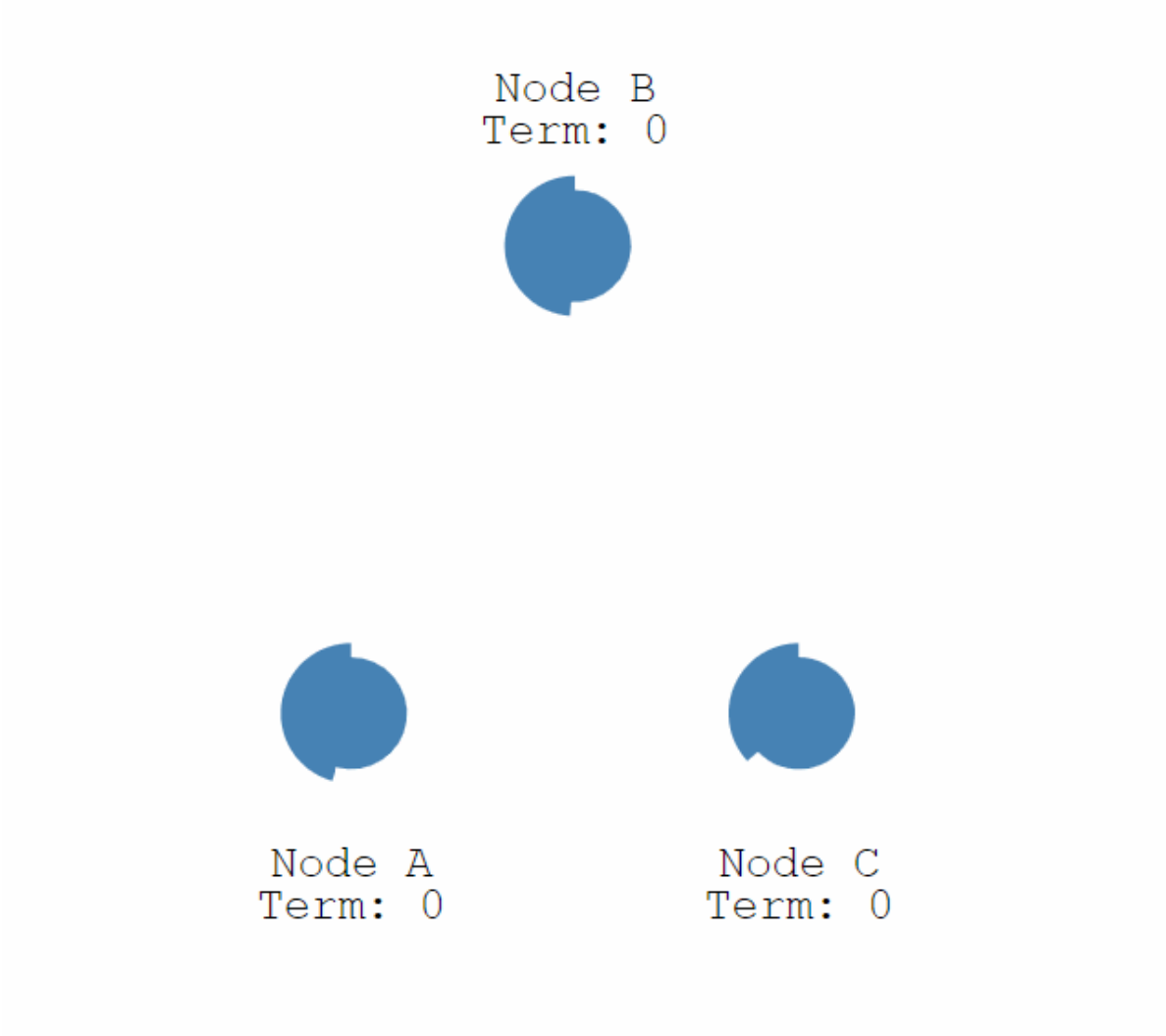
- 领导选举：有且仅有一个Leader节点，如果Leader宕机，通过选举机制选出新的Leader。
- 日志复制：Leader从客户端接收数据更新/删除请求，然后日志复制到Follower节点，从而保证一致性。

- 安全性：通过安全性原则来处理一些特殊 case，保证 Raft 算法的完备性。

  在本次实验中，将主要考虑领导选举问题。

领导选举问题

初始选举



在第一次选举时，集群中所有节点均为Follower角色，由于Raft算法随机超时时间的特性，会有随机一个节点的超时时间是最小的，那么他会最先没有等到Leader的心跳信息，发生超时。此时，该节点就会增加自己的任期编号，转换角色为Candidate，为自己投票，然后向其他节点发送请求投票的RPC消息，请他们推举自己为领导者。

如果其他节点接收到Candidate的请求投票 RPC 消息，在当前编号的任期内也还没有进行过投票，那么它将把选票投给该Candidate，并增加自己的任期编号。如果Candidate在选举超时时间内赢得了大多数的选票，将会成为本届任期内新的Leader；如果没有获得足够的投票，那么就会进行一轮新的选举，直至本节点或其他节点成为了Leader，则其余均变为Follower角色。

Leader挂掉后重新选举

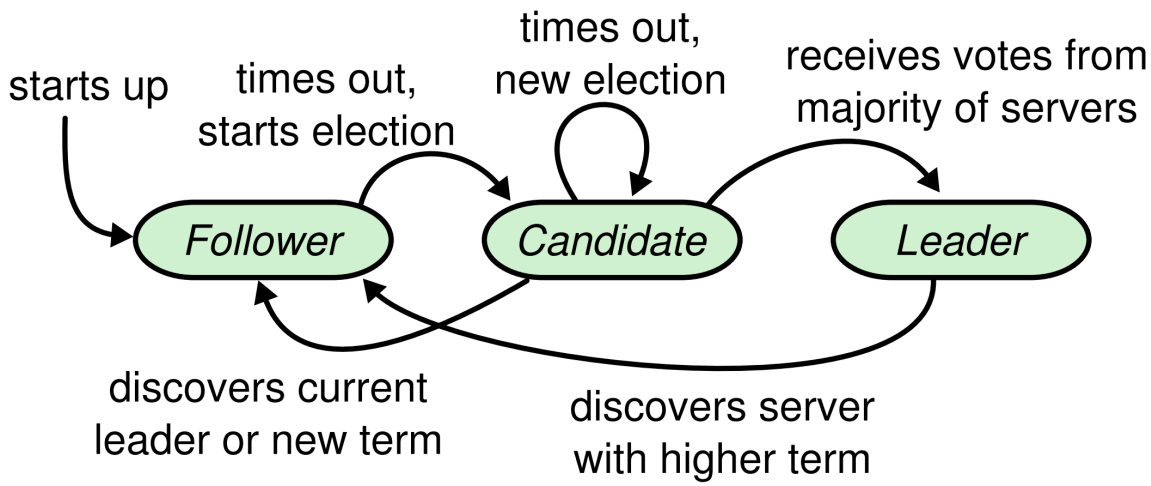
如果Leader下线或因为网络问题产生分区时会导致再次选举。此时将分成以下几种情况。

情况1：Leader下线

Leader 下线，此时所有其他节点的计时器不会被重置直到一个节点成为了 Candidate，和上述一样开始一轮新的选举选出一个新的 Leader。

情况 2：某一 Follower 结点与 Leader 间通信发生问题，导致发生了分区

这时没有 Leader 的那个分区就会进行一次选举。这种情况下，因为要求获得多数的投票才可以成为 Leader，因此只有拥有多数结点的分区可以正常工作。而对于少数结点的分区，即使仍存在 Leader，但由于写入日志的结点数量不可能超过半数因此不可能提交操作。



技术方案与实现

实现初始选举

初始化Raft服务器

在`Make`函数中，创建一个新的Raft服务器实例，并初始化其状态。需要设置当前任期，投票对象（-1表示还没有投票），日志等。

```
func Make(peers []*labrpc.ClientEnd, me int,
          persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{}
    // ... ...
    rf.currentTerm = 0
    rf.votedFor = -1
    rf.log = make([]Entries, 0)
    // ......
}
```

启动定时器

在`Make`函数中，启动定时器`Triker`，用于模拟网络延迟和Leader的心跳。定时器会在随机时间后触发，触发后会检查当前节点的状态，如果是Follower或Candidate状态，则可能触发新的选举。

```
Ticker := time.NewTicker(time.Duration(300+(rand.Int63()%150)) *
time.Millisecond)
go rf.ticker()
```

触发定时器

`ticker`函数是定时器触发时执行的函数，它检查当前节点的状态。如果是Follower状态，并且当前没有Leader（即没有收到心跳），则节点会转换为候选人状态并开始新的选举。

```go
func (rf *Raft) ticker() {
    for rf.killed() == false {
        rf.mu.Lock()
        if rf.state == Leader {
            // 如果是领导者，发送心跳
        } else if rf.state == Follower || rf.state == Candidate {
            // 如果是跟随者或候选人，可能触发新的选举
        }
        rf.mu.Unlock()
        time.Sleep(50 * time.Millisecond)
    }
}
```

开始选举

如果检测到需要开始选举（即当前节点是跟随者且没有收到心跳），节点会转换为候选人状态，并增加当前任期`currentTerm`。然后，节点会为自己投票，并发送请求投票`RequestVote`RPC给其他所有节点。

```go
if rf.state == Follower || rf.state == Candidate {
    rf.state = Candidate
    rf.currentTerm = rf.currentTerm + 1
    // ......
    // 先投自己一票
    rf.votedFor = rf.me
    rf.persist()
    count := 1
    for v := range rf.peers {
        if v == rf.me {
            continue
        }
        go func(args *RequestVoteArgs, v int, count *int) {
            // 发送请求投票RPC
        }(args, v, &count)
    }
}
```

发送请求投票

`RequestVote`函数是处理请求投票RPC的函数。它首先检查请求的任期是否大于当前任期，如果是，则会为请求者投票。

```go
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    if args.Term > rf.currentTerm {
        // 为请求者投票
        rf.votedFor = args.CandidateId
        reply.Votegranted = true
    }
    // ......
}
```

统计投票结果

在`RequestVote`的goroutine中，每个Candidate会等待其他节点的投票结果。如果收到超过半数节点的投票，则成为Leader。

```
go func(args *RequestVoteArgs, v int, count *int) {
    reply := RequestVoteReply{}
    ok := rf.sendRequestVote(v, args, &reply)
    if ok {
        if reply.Votegranted {
            *count += 1
            if *count > len(rf.peers)/2 {
                rf.isleader = true
                rf.state = Leader
                // ......
            }
        }
    }
}(args, v, &count)
```

Leader挂掉后重新选举

判断Leader挂掉

每个Follower节点都有一个定时器，定期检查是否需要进行Leader选举。如果Follower在选举超时时间内没有收到来自Leader的心跳，它会重置定时器并开始新的选举。

```
// 定时器回调函数
func (rf *Raft) ticker() {
    for rf.killed() == false {
        // ... 省略无关代码 ...
        if rf.state == Leader {
            // 如果是Leader，发送心跳
        } else {
            // 如果不是Leader，可能需要开始选举
            if rf.state == Follower || rf.state == Candidate {
                // 开始选举
            }
        }
        // 暂停一段时间，模拟网络延迟
        time.Sleep(50 * time.Millisecond)
    }
}
```

Leader挂掉后重新选举

当Follower没有在选举超时时间内收到Leader的心跳时，它会进行以下步骤：

- 增加当前任期`currentTerm`
- 将自己设置为候选人Candidate
- 重置选举超时计时器
- 请求其他服务器的选票`RequestVote`

```
go func() {
    for {
        select {
```

```
        case <-Ticker.C:
            // 开始选举
            if rf.state == Follower || rf.state == Candidate {
                rf.state = Candidate
                rf.currentTerm = rf.currentTerm + 1
                // ......
                // 发送RequestVote RPCs到其他服务器
                for v := range rf.peers {
                    if v == rf.me {
                        continue
                    }
                    go func(args *RequestVoteArgs, v int, count *int) {
                        // ......
                        ok = rf.sendRequestVote(v, args, &reply)
                        // ......
                        if reply.Votegranted {
                            // 如果收到大多数服务器的选票，成为Leader
                        }
                    }(args, v, &count)
                }
            }
            // ......
        case <-rf.IsGetHeartbeat:
            // 收到心跳，重置选举超时计时器
        }
    }
}()
```

实验总体思路

Lab 2A

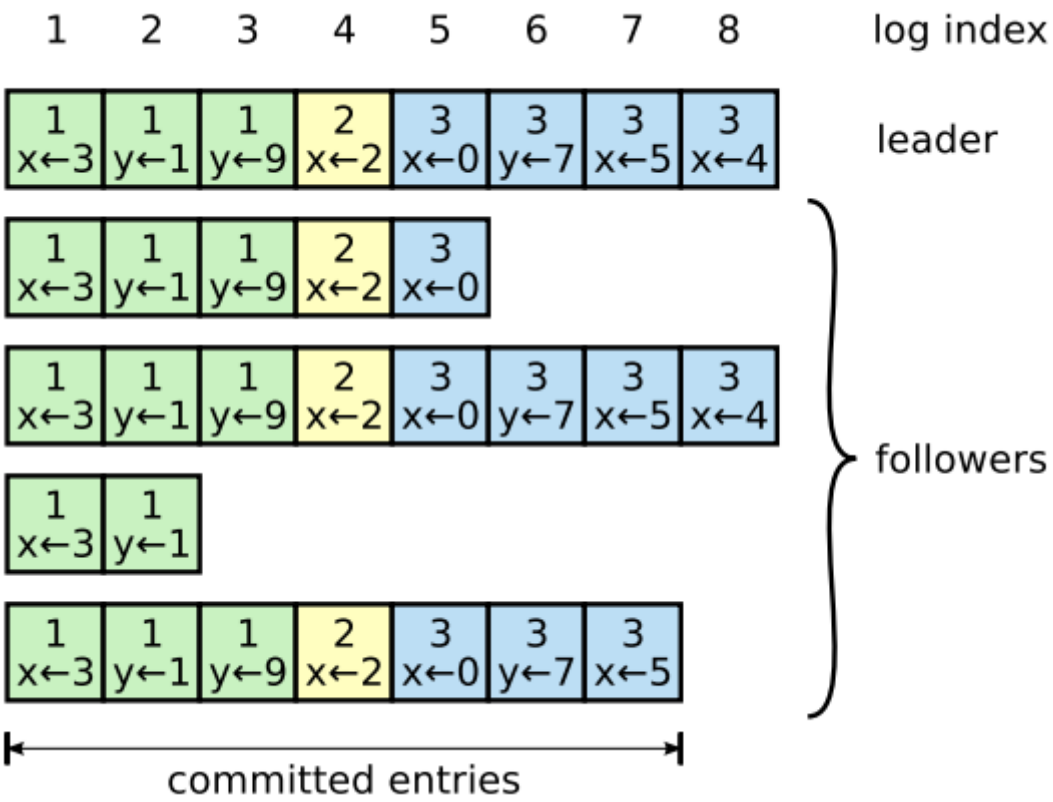
实现Raft的leader选举和心跳机制（没有log entries的AppendEntriesRPCs）。

Part 2A的目标是：选出一个leader，在没有故障的情况下，这个leader仍然是leader；如果旧leader故障，或者旧leader的数据包丢失，那么新的leader接管。

Lab 2B

实现raft的日志复制机制，即：**在无需第三方协调的前提下**，各个节点在任何情况下存储同一组系列日志，且对日志的顺序与内容达成共识。

日志条目的结构如下图，每个日志条目包含：执行的指令、领导人收到指令时的**任期号**、以及日志条目的**索引值。已提交的日志条目**是那些被领导人成功复制到大多数节点并应用到状态机的日志条目。领导人跟踪并通过附加日志的RPC告知其他节点当前已提交日志的索引。



日志匹配特性:

- 如果两个日志条目具有相同的索引和任期号，则它们包含的指令是相同的；
- 如果两个日志条目在相同的索引和任期号下相同，那么它们之前的所有日志条目也必定相同。

每次领导人附加新日志条目时，都进行一致性检查，确保日志条目不会冲突。

Leader的日志复制流程:

1. **Leader** 被选举出来后，开始接受客户端请求，**将每个请求转化为一个日志条目，附加到自己的日志**中；
2. **Leader**将日志条目**并行复制到集群中的其他节点**中，确保大多数**Follower**接收到该日志条目；
3. 一旦大多数**Follower**成功复制了日志条目，**Leader**将**该条目应用到自己的状态机中，并将执行结果返回给客户端**；
4. 如果某个**Follower**由于崩溃、网络延迟等原因未能及时复制日志条目，**Leader**会继续尝试，直到所有**Follower**都存储了该日志条目。

日志不一致的处理:

Leader和Follower之间的日志可能会因为Leader崩溃或网络问题而不一致。可能的情况包括：

- Follower丢失某些日志条目，或拥有Leader没有的条目。
- Leader在不同任期内记录了不同的日志条目。

Raft 通过**强制覆盖**不一致的日志条目来解决这个问题。Leader会找到日志一致性恢复的“最后一致点”，并将其之后的日志条目复制到Follower。所有的这些操作都在进行附加日志 RPCs 的一致性检查时完成。

Leader针对每一个Follower维护一个 nextIndex，表示下一个需要复制到Follower的日志条目的索引地址；当Follower的日志与Leader不一致时，Leader会减少 `nextIndex` 并重试，直到日志一致。

实验过程

Lab 2A

根据Raft论文中的Figure 2，此阶段，我们关心的是发送和接收RequestVote RPCs、与选举相关的服务器规则以及与leader选举相关的状态。

将Figure 2中的leader选举的状态添加到`raft.go`中的Raft结构中：

```
// A Go object implementing a single Raft peer.
type Raft struct {
    mu        sync.Mutex          // Lock to protect shared access to this
peer's state
    peers     []*labrpc.ClientEnd // RPC end points of all peers
    persister *Persister          // Object to hold this peer's persisted state
    me        int                 // this peer's index into peers[]
    dead      int32               // set by Kill()

    state State

    // Your data here (2A, 2B, 2C).
    IsGetHeartbeat chan int
    currentTerm    int       // incr default 0
    votedFor       int       // 投票给了谁
    log            []Entries
    isleader       bool
    Ticker         *time.Ticker
    nextIndex      []int
    matchIndex     []int
    commitIndex    int
    lastApplied    int
    applyCond      *sync.Cond

    lastIncludedIndex int
    lastIncludedTerm  int

    applyCh chan ApplyMsg
    // Look at the paper's Figure 2 for a description of what
    // state a Raft server must maintain.
    waitingSnapshot []byte
    waitingIndex    int // lastIncludedIndex
    waitingTerm     int // lastIncludedTerm
}
```

填充`RequestVoteArgs`和`RequestVoteReply`结构体，其用于 RequestVote RPC 的请求和回复信息：

- `RequestVoteArgs` 中包含了候选人的信息，包括候选人的任期号、日志索引、日志任期号，以及候选人的 ID，用于请求其他节点的投票。
- `RequestVoteReply` 中包含了当前节点的任期号以及是否同意投票的信息，帮助候选人判断是否能够成功赢得选举。

```
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    Term         int
    CandidateId  int
    LastLogIndex int
    LastLogTerm  int
}

type RequestVoteReply struct {
    // Your data here (2A).
    Term        int
    Votegranted bool
}
```

实现`RequestVote()` RPC处理程序，以便服务器相互投票。

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
//candidate 在选举期间发起
    // Your code here (2A, 2B).
    if rf.killed() {
       reply.Votegranted = false
       return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term > rf.currentTerm {
       rf.IsGetHeartbeat <- 0
       rf.isleader = false
       rf.state = Follower
       rf.votedFor = -1
       rf.currentTerm = args.Term
       lv := rf.logVote(args)
       if lv {
          rf.votedFor = args.CandidateId
          reply.Votegranted = true
       }

    } else {
       if args.Term == rf.currentTerm { //如果是投给的那个人来要票，那就再给他投一次
          rf.IsGetHeartbeat <- 0
          lv := rf.logVote(args)
          if (rf.votedFor == args.CandidateId || rf.votedFor == -1) && lv {
             rf.votedFor = args.CandidateId
             reply.Votegranted = true
          }
       } else {
          reply.Votegranted = false
       }
    }
    rf.persist()
    reply.Term = rf.currentTerm
    return
}
```

为了实现心跳（heartbeats），定义一个`AppendEntries` RPC结构，并让leader定期发送它们。

```
type AppendEntriesArgs struct {
    // Your data here (2A, 2B).
    Term         int  // 当前任期号
    LeaderId     int  // 领导者的 ID
    PrevLogIndex int  // 上一个日志条目的索引
    PrevLogTerm  int  // 上一个日志条目的任期号
    Entries      []Entries  // 新日志条目列表
    LeaderCommit int  //领导者已经提交的日志条目的索引
}
type AppendEntriesReply struct {
    // Your data here (2A).
    Term            int   // 当前任期号
    Success         bool  // 是否成功应用日志条目
    FastGoBackIndex int // 快速回退index
    FastGoBackTerm  int
}
```

编写一个`AppendEntries` RPC处理方法，重置选举超时，以便当一个server已经当选时，其他servers不会仍想成为leader。

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply
*AppendEntriesReply) { //heartbeat timeout implementation 由leader发送
    // Your code here (2A, 2B).
    //接收到heartbeat，刷新计时器
    if rf.killed() {
       return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if args.Term < rf.currentTerm {
       reply.Success = false
       reply.Term = rf.currentTerm
       return
    }
    rf.currentTerm = args.Term
    rf.isleader = false
    rf.state = Follower
    rf.persist()
    //if args.PrevLogIndex >= len(rf.log) {
    // DPrintf("not %d want result\n", rf.me)
    // reply.FastGoBackTerm = 0
    // reply.Term = rf.currentTerm
    // reply.Success = false
    // return
    //}

    //心跳处理：
    if len(args.Entries) == 0 {
       rf.IsGetHeartbeat <- 0
       reply.Success = true
       reply.Term = rf.currentTerm

       curIndex := len(rf.log) + rf.lastIncludedIndex
       if rf.lastIncludedIndex > args.PrevLogIndex { //日志一致性检查
          //DPrintf("%d rf.lastIncludedIndex= %d > args.PrevLogIndex=%d\n",
rf.me, rf.lastIncludedIndex, args.PrevLogIndex)
```

```
            reply.Success = false
            reply.FastGoBackIndex = rf.lastIncludedIndex + 1
            return
        }
        if curIndex > args.PrevLogIndex {
            if rf.CheckTermIsFirst(args.PrevLogIndex) { // 检查日志条目任期一致性
                reply.FastGoBackTerm = 0
                reply.FastGoBackIndex = 0
            } else {
                reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                index := args.PrevLogIndex - 1
                for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                    index--
                }
                reply.FastGoBackIndex = index + 1
            }
            reply.Success = false
            return
        }

        if curIndex < args.PrevLogIndex {
            reply.Success = false
            reply.FastGoBackTerm = 0
            reply.FastGoBackIndex = curIndex
            return
        }

        if curIndex == args.PrevLogIndex { // 如果日志索引一致，检查是否需要更新日志
            if !rf.CheckTermIsFirst(args.PrevLogIndex) {
                if rf.log[args.PrevLogIndex-rf.lastIncludedIndex-1].Term !=
args.PrevLogTerm {
                    reply.Success = false
                    reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                    index := args.PrevLogIndex - 1
                    for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                        index--
                    }
                    reply.FastGoBackIndex = index + 1
                    return
                }
            }
        }

        //DPrintf("know lastApplied me; %d,lastapp: %d, len: %d,index ;%d log:
%v\n", rf.me, rf.lastApplied, len(rf.log), args.PrevLogIndex, rf.log)

        if args.LeaderCommit > rf.commitIndex {
            rf.commitIndex = args.LeaderCommit
        }
        rf.applyCond.Broadcast()

    } else {
        rf.IsGetHeartbeat <- 0
        reply.Term = rf.currentTerm
```

```
        if !rf.CheckPreLogIndex(args) {
            curIndex := len(rf.log) + rf.lastIncludedIndex
            if rf.lastIncludedIndex > args.PrevLogIndex {
                //DPrintf("%d rf.lastIncludedIndex= %d > args.PrevLogIndex=%d\n",
rf.me, rf.lastIncludedIndex, args.PrevLogIndex)
                reply.Success = false
                reply.FastGoBackIndex = rf.lastIncludedIndex + 1
                return
            }

            if curIndex > args.PrevLogIndex {
                if !rf.CheckTermIsFirst(args.PrevLogIndex) {
                    //DPrintf("%d get args.PrevLogIndex= %d,rf.lastIncludedIndex=
%d\n", rf.me, args.PrevLogIndex, rf.lastIncludedIndex)
                    reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                    index := args.PrevLogIndex - 1
                    for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                        index--
                    }
                    reply.FastGoBackIndex = index + 1
                }
                reply.Success = false
                return
            }

            if curIndex < args.PrevLogIndex {
                reply.Success = false
                reply.FastGoBackTerm = 0
                reply.FastGoBackIndex = curIndex
                return
            }

            if curIndex == args.PrevLogIndex {
                if !rf.CheckTermIsFirst(curIndex) {
                    if rf.log[args.PrevLogIndex-rf.lastIncludedIndex-1].Term !=
args.PrevLogTerm {
                        reply.Success = false
                        reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                        index := args.PrevLogIndex - 1
                        for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                            index--
                        }
                        reply.FastGoBackIndex = index + 1
                        return
                    }
                }
            }

        }
        reply.Success = true
        for index, entry := range args.Entries {
            if entry.Index >= rf.GetFirstIndex() && (entry.Index >=
rf.GetAllLogLen() || rf.log[entry.Index-rf.lastIncludedIndex-1].Term !=
entry.Term) {
```

```
            rf.log = append(rf.log[:entry.Index-rf.lastIncludedIndex-1],
args.Entries[index:]...)
            break
         }
      }
      rf.persist()
      if args.LeaderCommit > rf.commitIndex {
         rf.commitIndex = args.LeaderCommit
      }
      rf.applyCond.Broadcast()
   }
   reply.Term = rf.currentTerm
   return
}
```

编写 `ticker` 函数，实现周期性地检查和触发心跳机制以及启动领导人选举。

```
func (rf *Raft) ticker() {
   for rf.killed() == false {
      // Your code here (2A)
      // Check if a leader election should be started.
      //在ticker里发心跳
      rf.mu.Lock()
      if rf.state == Leader {
         args := &AppendEntriesArgs{Term: rf.currentTerm, LeaderId: rf.me}
         reply := &AppendEntriesReply{}
         rf.mu.Unlock()
         rf.sendAppendEntries(args, reply)
      } else {
         rf.mu.Unlock()
      }
      // pause for a random amount of time between 50 and 350
      // milliseconds.
      // ms := 50 + (rand.Int63() % 300)
      time.Sleep(50 * time.Millisecond)
   }
}
```

完成`Make`函数的编写，初始化 Raft 节点的状态、启动并配置定时器（`ticker`）、处理选举、心跳机制以及日志应用。修改`Make`以创建一个后台groutine，当它有一段时间没有收到另一个peer的消息时，通过发送RequestVote RPCs周期性地启动leader选举。

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

   // Your initialization code here (2A, 2B, 2C).
   rf.isleader = false
   rf.state = Follower
   rf.currentTerm = 0
   rf.votedFor = -1
   rf.nextIndex = make([]int, len(rf.peers))
```

```
    rf.matchIndex = make([]int, len(rf.peers))
    rf.log = make([]Entries, 0)
    rf.applyCh = applyCh
    rf.applyCond = sync.NewCond(&rf.mu)

    Ticker := time.NewTicker(time.Duration(300+(rand.Int63()%150)) *
time.Millisecond)
    rf.Ticker = Ticker
    rf.log = make([]Entries, 0)
    rf.lastIncludedIndex = -1
    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState())
    if rf.lastIncludedIndex > 0 {
       rf.lastApplied = rf.lastIncludedIndex + 1
       rf.commitIndex = rf.lastApplied
    }

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

    go rf.applier()

    go func() {

       for {
          select {
          case <-Ticker.C:
             //开始选举

             rf.Ticker.Reset(time.Duration(300+(rand.Int63()%150)) *
time.Millisecond)
             if rf.state == Follower || rf.state == Candidate {
                rf.state = Candidate
                rf.currentTerm = rf.currentTerm + 1
                llIndex := 0
                llTerm := 0
                if rf.GetAllLogLen() == 0 {
                   llIndex = -1
                   llTerm = 0
                } else {
                   if rf.lastIncludedIndex == -1 {

                   }
                   if len(rf.log) == 0 {
                      llIndex = rf.lastIncludedIndex
                      llTerm = rf.lastIncludedTerm
                   } else {
                      llIndex = rf.GetLastIndex()
                      llTerm = rf.GetLastTerm()
                   }

                }

                args := &RequestVoteArgs{
                   Term:         rf.currentTerm,
                   CandidateId:  rf.me,
                   LastLogIndex: llIndex,
                   LastLogTerm:  llTerm,
```

```
                }

                //先投自己一票
                rf.votedFor = me
                rf.persist()
                count := 1
                //fmt.Println("[candidate]", rf.me, rf.currentTerm, time.Now())
                for v := range rf.peers {
                    if v == rf.me {
                        continue
                    }
                    go func(args *RequestVoteArgs, v int, count *int) {
                        reply := RequestVoteReply{}
                        var ok bool = false
                        ok = rf.sendRequestVote(v, args, &reply)
                        fmt.Println("who send vote ", v, reply, rf.isleader,
rf.state, rf.currentTerm)
                        if ok {
                            rf.mu.Lock()

                            if reply.Term > rf.currentTerm {

                                rf.currentTerm = reply.Term
                                rf.votedFor = -1
                                rf.state = Follower
                                rf.persist()
                            }

                            if reply.Votegranted {
                                if rf.currentTerm == args.Term {
                                    *count += 1
                                }
                                fmt.Println("[count num]", *count, rf.me)
                                if *count > len(rf.peers)/2 {
                                    *count = 0
                                    rf.isleader = true
                                    rf.state = Leader

                                    rf.persist()
                                    fmt.Println("[election leader]", rf.me)
                                    for i := range rf.nextIndex {
                                        rf.nextIndex[i] = rf.GetAllLogLen()
                                        rf.matchIndex[i] = -1
                                    }

                                    args := &AppendEntriesArgs{
                                        Term:     rf.currentTerm,
                                        LeaderId: rf.me,
                                    }

                                    reply := &AppendEntriesReply{}
                                    rf.mu.Unlock()
                                    rf.sendAppendEntries(args, reply)
                                } else {
                                    rf.mu.Unlock()
                                }
                            } else {
                                rf.mu.Unlock()
```

```
                        }

                    }
                }(args, v, &count)

            }

        }

        case <-rf.IsGetHeartbeat: //收到心跳
            rf.Ticker.Reset(time.Duration(300+(rand.Int63()%150)) *
time.Millisecond)
        }
    }
}()

    return rf
}
```

Lab 2B

实现Start函数

- 客户端传输指令到下层raft集群的**入口**：
  - 由Leader接收客户端提交的命令，将该命令记录到日志中；触发日志复制过程，利用raft算法，将新的日志条目从Leader复制到所有的Follower。

1. 锁定共享资源

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

2. 获取当前任期与Leader身份
   - `term`：当前节点的任期（`currentTerm`），用于确保日志条目的顺序一致性。
   - `isLeader`：当前节点是否是领导者（`isleader`），只有领导者才能执行日志复制和同步。

```
term := rf.currentTerm
isLeader := rf.isleader
```

3. 如果是Leader，处理日志条目；如果不是Leader，返回错误
   - `Entries{}`：构造一个新的日志条目。这个条目包括：
     - `Term`：当前的任期。
     - `Command`：客户端提交的命令。
     - `Index`：日志条目的索引，这里使用 `len(rf.log) + rf.lastIncludedIndex + 1` 计算日志的下一个索引。
   - **日志追加**：将新的日志条目 `en` 追加到 `rf.log` 中。`rf.log` 是一个存储所有日志条目的切片。
   - **更新 `nextIndex`**：`rf.nextIndex[rf.me]` 用来记录下一个要发送给其他Follower的日志条目的索引。这里将其设置为当前日志长度（加上快照的偏移量 `lastIncludedIndex`）。
   - **持久化日志**：`rf.persist()` 将日志状态保存到磁盘，以便在节点重启时恢复。
   - **日志打印**：`DPrintf` 打印当前Leader的日志信息。

```
if isLeader {
    en := Entries{Term: term, Command: command, Index: len(rf.log) +
rf.lastIncludedIndex + 1}
    rf.log = append(rf.log, en)
    rf.nextIndex[rf.me] = len(rf.log) + rf.lastIncludedIndex + 1
    rf.persist()
    DPrintf("[start]:me: %d,log:%v\n", rf.me, rf.log)
} else {
    return -1, term, isLeader
}
```

4. 向其他节点发送 `AppendEntries` 请求
    - **构造** `AppendEntriesArgs`：Leader构造一个 `AppendEntries` 请求，准备将新的日志条目同步到所有Follower。请求包括：
        - `Term`：当前任期。
        - `LeaderId`：当前节点的 ID（即领导者的 ID），用于标识请求来自哪个Leader。
    - **异步发送日志复制请求**：`go rf.sendAppendEntries(args, reply)` 启动一个协程，通过 RPC 向其他节点发送 `AppendEntries` 请求。`sendAppendEntries` 方法会负责将新的日志条目传递给Follower，并根据Follower的响应调整日志复制进度。

```
args := &AppendEntriesArgs{
    Term:     rf.currentTerm,
    LeaderId: rf.me,
}

reply := &AppendEntriesReply{}
go rf.sendAppendEntries(args, reply)
```

5. 返回值
    - `rf.GetAllLogLen()`：返回当前日志的长度，表示当前日志中包含的条目数。
    - `term`：返回当前的任期，供客户端使用。
    - `isLeader`：返回当前节点是否是领导者，供客户端使用。

```
return rf.GetAllLogLen(), term, isLeader
```

完善AppendEntries()函数

- 由Leader发送，向Follower发送日志条目，或仅发送心跳（没有日志条目）
- 将心跳和日志复制功能分开处理：
    - **心跳处理**：如果没有日志条目（`args.Entries == 0`），转入心跳机制，告知Follower，Leader依然存活并维持地位。如果心跳成功，跟随者的 `commitIndex` 会同步。
    - **日志复制**：如果 `args.Entries` 非空，Leader尝试将新的日志条目追加到Follower的日志中。如果Follower的日志存在冲突（如日志条目索引不同或者日志条目的 term 不一致），Leader会要求Follower回滚到一致的日志位置。

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply
*AppendEntriesReply) {
    if rf.killed() {
        return
    }
```

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

    // 如果收到的 term 小于当前 term，直接拒绝
    if args.Term < rf.currentTerm {
        reply.Success = false
        reply.Term = rf.currentTerm
        return
    }

    // 更新当前 term 和状态，转为 Follower
    rf.currentTerm = args.Term
    rf.isleader = false
    rf.state = Follower
    rf.persist()

    // 处理心跳的情况（args.Entries为空）
    if len(args.Entries) == 0 {
        rf.IsGetHeartbeat <- 0
        reply.Success = true
        reply.Term = rf.currentTerm
        // 确认跟随者日志是否同步
        curIndex := len(rf.log) + rf.lastIncludedIndex
        if rf.lastIncludedIndex > args.PrevLogIndex {
            reply.Success = false
            reply.FastGoBackIndex = rf.lastIncludedIndex + 1
            return
        }
        // 判断日志是否一致
        if curIndex > args.PrevLogIndex {
            if rf.CheckTermIsFirst(args.PrevLogIndex) {
                reply.FastGoBackTerm = 0
                reply.FastGoBackIndex = 0
            } else {
                reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                index := args.PrevLogIndex - 1
                for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                    index--
                }
                reply.FastGoBackIndex = index + 1
            }
            reply.Success = false
            return
        }
        // 更新提交的日志条目
        if args.LeaderCommit > rf.commitIndex {
            rf.commitIndex = args.LeaderCommit
        }
        rf.applyCond.Broadcast()
    } else {
        rf.IsGetHeartbeat <- 0
        reply.Term = rf.currentTerm

        // 检查前一个日志条目的正确性
        if !rf.CheckPreLogIndex(args) {
            curIndex := len(rf.log) + rf.lastIncludedIndex
```

```
            if rf.lastIncludedIndex > args.PrevLogIndex {
                reply.Success = false
                reply.FastGoBackIndex = rf.lastIncludedIndex + 1
                return
            }
            if curIndex > args.PrevLogIndex {
                if !rf.CheckTermIsFirst(args.PrevLogIndex) {
                    reply.FastGoBackTerm = rf.log[args.PrevLogIndex-
rf.lastIncludedIndex-1].Term
                    index := args.PrevLogIndex - 1
                    for index >= rf.GetFirstIndex() && rf.log[index-
rf.lastIncludedIndex-1].Term == reply.FastGoBackTerm {
                        index--
                    }
                    reply.FastGoBackIndex = index + 1
                }
                reply.Success = false
                return
            }
        }
        // 如果日志一致，复制新的日志条目
        reply.Success = true
        for index, entry := range args.Entries {
            if entry.Index >= rf.GetFirstIndex() && (entry.Index >=
rf.GetAllLogLen() || rf.log[entry.Index-rf.lastIncludedIndex-1].Term !=
entry.Term) {
                rf.log = append(rf.log[:entry.Index-rf.lastIncludedIndex-1],
args.Entries[index:]...)
                break
            }
        }
        rf.persist()
        // 更新提交的日志条目
        if args.LeaderCommit > rf.commitIndex {
            rf.commitIndex = args.LeaderCommit
        }
        rf.applyCond.Broadcast()
    }
    reply.Term = rf.currentTerm
    return
}
```

实现sendAppendEntries函数，发送日志条目

- Leader向所有Follower发送追加日志条目的请求：
    - Leader通过 `sendAppendEntries` 向每个跟随者发送日志条目；
    - 如果Follower的日志已同步，更新 `nextIndex` 和 `matchIndex`，否则调整其日志指针；
    - 如果Leader发现Follower的日志不一致，会请求回退到一个一致的位置。

```
func (rf *Raft) sendAppendEntries(args *AppendEntriesArgs, reply
*AppendEntriesReply) {
    if rf.killed() {
        return
```

```
    }

    for i, v := range rf.peers {
        if i == rf.me {
            continue
        }
        rf.mu.Lock()
        if rf.state != Leader {
            rf.mu.Unlock()
            break
        }

        next := rf.nextIndex[i]
        args.LeaderCommit = rf.commitIndex
        args.PrevLogIndex = next - 1

        // 如果需要安装快照，跳过日志复制
        if next <= rf.lastIncludedIndex {
            rf.mu.Unlock()
            go rf.SendInstallSnapshot(i)
            continue
        }

        if rf.CheckTermIsFirst(args.PrevLogIndex) {
            args.Entries = make([]Entries, len(rf.log))
            args.PrevLogTerm = rf.GetFirstTerm()
        } else {
            args.Entries = make([]Entries, rf.GetLastIndex()-args.PrevLogIndex)
            args.PrevLogTerm = rf.log[args.PrevLogIndex-rf.lastIncludedIndex-
1].Term
        }
        if next < rf.GetAllLogLen() {
            copy(args.Entries, rf.log[next-rf.lastIncludedIndex-1:])
        }
        rf.mu.Unlock()
        go func(v *labrpc.ClientEnd, i int, args AppendEntriesArgs, reply
AppendEntriesReply) {
            ok := v.Call("Raft.AppendEntries", &args, &reply)
            rf.mu.Lock()
            defer rf.mu.Unlock()
            if ok {
                if reply.Success {
                    newNext := args.PrevLogIndex + len(args.Entries) + 1
                    newMatch := args.PrevLogIndex + len(args.Entries)
                    if newNext > rf.nextIndex[i] {
                        rf.nextIndex[i] = newNext
                    }
                    if newMatch > rf.matchIndex[i] {
                        rf.matchIndex[i] = newMatch
                    }

                    for end := len(rf.log) - 1; end >= 0; end-- {
                        if rf.log[end].Term != rf.currentTerm {
                            break
                        }
                        n := 1
                        for k := range rf.matchIndex {
```

```
                            if rf.me != k && rf.matchIndex[k] >=
end+rf.lastIncludedIndex+1 {
                                n++
                            }
                        }
                        if n > len(rf.peers)/2 {
                            rf.commitIndex = end + rf.lastIncludedIndex + 2
                            break
                        }
                    }

                    rf.applyCond.Broadcast()
                } else {
                    if reply.Term > rf.currentTerm {
                        rf.currentTerm = reply.Term
                        rf.votedFor = -1
                        rf.state = Follower
                        rf.isleader = false
                        rf.persist()
                        return
                    }
                    if reply.Term == rf.currentTerm {
                        if reply.FastGoBackIndex > 0 {
                            rf.nextIndex[i] = reply.FastGoBackIndex
                        }
                    }
                }
            }
        }(v, i, *args, *reply)
    }
}
```

测试样例

- **TestBasicAgree2B**：测试Raft协议的基本一致性，检查在多个迭代中日志项的提交和一致性。
  - **核心流程**：每次迭代时，首先检查在调用 `Start()` 之前是否有节点已经提交了日志项（期望没有）。然后检查不同节点的日志索引是否按预期顺序进行存储。
  - **考虑情况**：节点是否能够在没有提前提交日志的情况下正确处理日志请求，确保索引的一致性。

```
for index := 1; index < iters+1; index++ {
        DPrintf(11, "\nthis is the %d th iter...\n", index)
        //检测start函数调用前，有几个节点已经提交了日志，因为调用start函数就相当于
tester
        //生成日志项并且投放给主节点，所以这里的没有启动start方法就没有日志项产生
        nd, _ := cfg.nCommitted(index)
        if nd > 0 {
            t.Fatalf("some have committed before Start()")
        }
        // 检查索引大多数节点对待同一个日志，存储的位置（即索引）是否和生产时的顺序相
同
        xindex := cfg.one(index*100, servers, false)
        if xindex != index {
            t.Fatalf("got index %v but expected %v", xindex, index)
        }
```

```
        DPrintf(11, "\nfinished the %d th iter...\n", index)

    }
```

- **TestRPCBytes2B**：测试RPC字节数，确保每个命令只发送一次。
  - **考虑情况**：确保没有不必要的重复RPC调用，避免过多的网络开销。
- **TestFollowerFailure2B**：测试Follower的逐步故障
  - **核心流程**：该测试模拟了一个Follower节点的故障，并验证Leader和剩余的Follower是否能够继续进行日志的复制。在测试中，首先提交命令，然后逐步断开一个Follower节点，检查剩余FollowerFollower是否仍然能够继续进行日志复制和提交。
  - **考虑情况**：即使有Follower故障，Raft协议也应能保证集群的稳定性，确保Leader和剩余Follower能够继续达成一致并进行日志复制。

```
    cfg.one(101, servers, false)

    // disconnect one follower from the network.
    leader1 := cfg.checkOneLeader()
    cfg.disconnect((leader1 + 1) % servers)

    // the leader and remaining follower should be
    // able to agree despite the disconnected follower.
    cfg.one(102, servers-1, false)
    time.Sleep(RaftElectionTimeout)
    cfg.one(103, servers-1, false)

    // disconnect the remaining follower
    leader2 := cfg.checkOneLeader()
    cfg.disconnect((leader2 + 1) % servers)
    cfg.disconnect((leader2 + 2) % servers)
```

- **TestLeaderFailure2B**：测试Leader的故障处理
  - **核心流程**：模拟一个Leader的故障，并验证剩余的节点是否能够通过选举产生新的Leader并继续提交日志。
  - **考虑情况**：即使Leader失败，剩余的节点应能够选举出新的Leader，并且新的Leader能够继续处理客户端请求。

```
cfg.one(101, servers, false)

    // disconnect the first leader.
    leader1 := cfg.checkOneLeader()
    cfg.disconnect(leader1)

    // the remaining followers should elect
    // a new leader.
    cfg.one(102, servers-1, false)
    time.Sleep(RaftElectionTimeout)
    cfg.one(103, servers-1, false)

    // disconnect the new leader.
    leader2 := cfg.checkOneLeader()
    cfg.disconnect(leader2)
```

- **TestFailAgree2B**：测试当一个Follower重新连接到集群后，是否能够恢复一致性。

- **核心流程**：首先断开一个Follower，然后提交多个命令，确保其他节点能够继续处理请求。随后重新连接断开的节点，验证它是否能够同步以前的日志并继续提交新的命令。
- **考虑情况**：当一个Follower恢复时，它需要能够同步之前的日志并且继续与集群达成一致。

```
    cfg.one(101, servers, false)
    // disconnect one follower from the network.
    leader := cfg.checkOneLeader()
    cfg.disconnect((leader + 1) % servers)
    DPrintf(111, "already let the node %d offline to add new entries...",
(leader+1)%servers)
    // the leader and remaining follower should be
    // able to agree despite the disconnected follower.
    cfg.one(102, servers-1, false)
    cfg.one(103, servers-1, false)
    time.Sleep(RaftElectionTimeout)
    cfg.one(104, servers-1, false)
    cfg.one(105, servers-1, false)
    // re-connect
    cfg.connect((leader + 1) % servers)
    DPrintf(111, "after connected, check whether previous added entries can
sync to %d...", (leader+1)%servers)

    // the full set of servers should preserve
    // previous agreements, and be able to agree
    // on new commands.
    cfg.one(106, servers, true)
    time.Sleep(RaftElectionTimeout)
    cfg.one(107, servers, true)
```

- **TestFailNoAgree2B**：测试当有过多的Follower节点失联时，集群是否能够保持一致性。
  - **核心流程**：模拟多个Follower节点的失联，验证即使没有大多数节点，集群依然无法继续提交命令。然后修复网络并检查是否能恢复一致性。
  - **考虑情况**：当集群中的节点失联过多时，无法达成多数派一致性，命令不能被提交，直到恢复足够的节点数量。

```
  cfg.one(10, servers, false)

    // 3 of 5 followers disconnect
    leader := cfg.checkOneLeader()
    cfg.disconnect((leader + 1) % servers)
    cfg.disconnect((leader + 2) % servers)
    cfg.disconnect((leader + 3) % servers)

  /*......*/

  // repair
    cfg.connect((leader + 1) % servers)
    cfg.connect((leader + 2) % servers)
    cfg.connect((leader + 3) % servers)

    // the disconnected majority may have chosen a leader from
    // among their own ranks, forgetting index 2.
    leader2 := cfg.checkOneLeader()
    index2, _, ok2 := cfg.rafts[leader2].Start(30)
    if ok2 == false {
```

```
        t.Fatalf("leader2 rejected Start()")
    }
    if index2 < 2 || index2 > 3 {
        t.Fatalf("unexpected index %v", index2)
    }

    cfg.one(1000, servers, true)
```

- **TestRejoin2B**：测试在Leader分区和恢复后的行为。
  - **核心流程**：模拟Leader断开连接并重新连接，验证Leader和Follower在重新连接后是否能够继续提交日志并保持一致性。
  - **考虑情况**：当Leader或Follower因网络分区断开连接后恢复时，必须确保它能够与集群同步之前的日志，并继续执行新的命令。

```
    cfg.one(101, servers, true)

    // leader network failure
    leader1 := cfg.checkOneLeader()
    cfg.disconnect(leader1)

    // make old leader try to agree on some entries
    cfg.rafts[leader1].Start(102)
    cfg.rafts[leader1].Start(103)
    cfg.rafts[leader1].Start(104)

    // new leader commits, also for index=2
    cfg.one(103, 2, true)

    // new leader network failure
    leader2 := cfg.checkOneLeader()
    cfg.disconnect(leader2)

    // old leader connected again
    cfg.connect(leader1)

    cfg.one(104, 2, true)

    // all together now
    cfg.connect(leader2)

    cfg.one(105, servers, true)
```

- **TestConcurrentStarts2B**：测试Raft协议在并发情况下的行为。
  - **核心流程**：该测试模拟了多个并发的 `Start()` 调用，确保即使多个命令同时提交，集群仍然能维持一致性，且每个命令都会按顺序执行。并通过 `wait()` 方法来验证每个命令是否成功提交。
  - **考虑情况**：在高并发情况下，节点应该保持一致性，并且能够正确处理并发的命令提交。
- **TestBackup2B**：测试Leader快速回滚错误的Follower日志。
  - **核心流程**：模拟一个Leader与多个Follower分区，并且提交多个不会提交的日志；然后通过重新连接大多数节点，验证Leader是否能够迅速恢复并继续提交日志。
  - **考虑情况**：当出现日志不一致的情况时，Leader应该能够修复日志并快速恢复正常操作，避免数据丢失。

```
    cfg.one(rand.Int(), servers, true)
```

```
    // put leader and one follower in a partition
    leader1 := cfg.checkOneLeader()
    DPrintf(11, "check one leader alive with id %d...\n",
cfg.rafts[leader1].me)
    cfg.disconnect((leader1 + 2) % servers)
    cfg.disconnect((leader1 + 3) % servers)
    cfg.disconnect((leader1 + 4) % servers)
    DPrintf(11, "let followers whose id is greater than leader %d were
forced offline...\n", cfg.rafts[leader1].me)

    // submit lots of commands that won't commit
    // 意思是因为前面强制下线了多数节点，所以这里的添加的日志都会丢失，不会被提交
    for i := 0; i < 50; i++ {
        cfg.rafts[leader1].Start(rand.Int())
    }
    DPrintf(11, "after corruption, 50 cmds are inserted and it should not be
successfully done...\n")
    time.Sleep(RaftElectionTimeout / 2)

    cfg.disconnect((leader1 + 0) % servers)
    cfg.disconnect((leader1 + 1) % servers)
    DPrintf(11, "last 2 instances are done now !!!! \n")
    // allow other partition to recover
    cfg.connect((leader1 + 2) % servers)
    cfg.connect((leader1 + 3) % servers)
    cfg.connect((leader1 + 4) % servers)
    DPrintf(11, "3 nodes are now reconnected and new 50 cmds is gonna be
checked whether be refused...\n")
    // lots of successful commands to new group.
    for i := 0; i < 50; i++ {
        cfg.one(rand.Int(), 3, true)
    }
    // now another partitioned leader and one follower
    leader2 := cfg.checkOneLeader()
    other := (leader1 + 2) % servers
    if leader2 == other {
        other = (leader2 + 1) % servers // 另一个从节点
    }

    cfg.disconnect(other) // 下线另一个从节点

    // lots more commands that won't commit
    for i := 0; i < 50; i++ {
        cfg.rafts[leader2].Start(rand.Int())
    }

    time.Sleep(RaftElectionTimeout / 2)
    // bring original leader back to life,
    for i := 0; i < servers; i++ {
        cfg.disconnect(i)
    }
    DPrintf(11, "all nodes are down....\n")

    cfg.connect((leader1 + 0) % servers)
    cfg.connect((leader1 + 1) % servers)
    cfg.connect(other)
    DPrintf(11, "3 are reconnected....\n")
```

```
    // lots of successful commands to new group.
    for i := 0; i < 50; i++ {
        cfg.one(rand.Int(), 3, true)
    }
    DPrintf(11, "check successfully!....\n")
    // now everyone
    for i := 0; i < servers; i++ {
        cfg.connect(i)
    }

    cfg.one(rand.Int(), servers, true)
```

- **TestCount2B**：测试RPC计数，确保RPC请求数量不会过多。
    - **核心流程**：通过发送多个命令，检查每个节点的RPC请求数量，确保RPC调用的次数不会过多。每次测试都会重新选举Leader，并通过RPC计数来验证是否存在不必要的RPC调用。

      考虑情况：RPC的数量应该控制在合理范围内，避免因频繁的RPC请求而造成不必要的开销。

实验结果

Lab 2A

通过Test-2A测试。测试结果的每个"Passed"行包括5个数字，它们分别是：

1. 测试花费的时间（以秒为单位）
2. Raft peers的数量
3. 在测试期间发送的RPC的数量
4. RPC消息中的总字节数
5. Raft报告提交的日志条目的数量。

```
hechenrui@MacBook-Air-2 raft % go test -run 2A
Test (2A): initial election ...
  ... Passed --   3.1  3  102   32218    0
Test (2A): election after network failure ...
  ... Passed --   4.6  3  188   42322    0
Test (2A): multiple elections ...
  ... Passed --   6.4  7  954  201678    0
PASS
ok      6.5840/raft     14.475s
hechenrui@MacBook-Air-2 raft %
```

Lab 2B

```
lisa@Lisa:/mnt/c/Users/16584/Documents/Github/MIT6.824-6.5840/src/raft$ go test -run 2B
Test (2B1): basic agreement ...
  ... Passed --   1.1  3   22    6866    3
Test (2B2): RPC byte count ...
  ... Passed --   1.2  3   28   38884    4
Test (2B3): test progressive failure of followers ...
  ... Passed --   5.0  3  126   28747    3
Test (2B4): test failure of leaders ...
  ... Passed --   5.4  3  202   46743    3
Test (2B5): agreement after follower reconnects ...
  ... Passed --   5.1  3  112   31797    7
Test (2B6): no agreement if too many followers disconnect ...
  ... Passed --   3.6  5  194   44394    3
Test (2B7): concurrent Start()s ...
  ... Passed --   0.5  7   54   17214    6
Test (2B8): rejoin of partitioned leader ...
  ... Passed --   6.7  3  194   50979    4
Test (2B9): leader backs up quickly over incorrect follower logs ...
  ... Passed --  27.0  5 2988 1799938  103
Test (2B_10): RPC counts aren't too high ...
  ... Passed --   2.1  3   58   19359   12
PASS
ok      MIT6.824-6.5840/raft    57.736s
```

效果和问题分析

必答问题

1.可以用Raft做什么?

- 分布式数据库
- 基于 DLedger 实现多节点的缓存同步更新
- 基于日志复制的副本容错处理

2.像Raft这样的系统是否可以在集群中只有少数处于活动状态时生存并继续运行?

先说明结论：不可以。

如果客户端和服务器准确知道集群中哪些节点处于活动状态，则可以构建一个系统，在至少一个服务器存活的情况下继续运行，并选择已知活跃的服务器作为Leader。但是计算机很难判断服务器是挂了，还是因为网络问题导致消息丢失。所以实现方式有两种，1.人工决定，2.允许“分脑”操作，并为服务器提供一种方式使其在网络分区恢复后能够调和分期。

因此，对于像Raft这样的分布式一致性系统来说，只有少数服务器处于活跃状态时，系统是否能继续运行并保持一致性，取决于集群的配置和容错机制。在Raft中，系统通过多数派原则来维护一致性。具体而言，Raft要求大多数服务器处于活跃状态，并且能够参与选举和日志复制。若集群中只有少数服务器存活，无法形成多数派，那么Raft协议将无法正常工作，因为无法选举出一个领导者，或者无法保证日志复制的一致性。**因此，Raft这样的系统通常需要至少一半以上的服务器保持活跃状态才能继续运行和维持一致性。**如果活跃服务器不足以满足多数派要求，集群将无法继续正常工作。

3.什么是拜占庭问题？它会使Raft失效吗？请阐述理由。

拜占庭问题是指在一个分布式系统中，一些节点可能会故意提供错误的、欺骗性的或者不一致的消息，导致系统无法达成一致性决策的问题。在拜占庭容错的场景中，系统需要能够容忍一定数量的恶意节点，这些节点可能会发送虚假信息，破坏系统的一致性。

拜占庭问题会使Raft协议失效。发送虚假的日志条目或伪造投票信息，进而影响选举过程或导致日志的不一致，使得Raft协议无法正常工作。在Raft协议中节点之间的选举和日志复制是基于信任的假设进行的，如果某些节点故意发送不一致的或伪造的消息，Raft将无法区分这些恶意行为，进而导致选举失败、系统状态不一致，或者数据丢失。

4.为什么随机设置选举timeout?

- 避免多个节点同时触发选举：Raft协议中的选举机制依赖于超时触发。在选举过程中，每个候选人会等待一个随机的超时时间，一旦超时没有收到Leader的心跳，它会发起选举。如果所有节点的选举超时时间完全相同，那么在同一时刻，所有节点可能都会同时触发选举，导致选举冲突和分割选票。通过随机化每个节点的超时时间，可以使得节点们在不同的时间触发选举，从而避免多次选举同时发生。
- 减少分割选票：可能会导致两个或多个候选人在同一时刻发起选举，导致没有任何候选人能获得超过半数的选票，从而选举无法顺利完成。随机化选举超时时间可以有效减少这种情况，因为随机设置会使得选举时刻分布更均匀，增加了某一个节点在选举中胜出的机会，从而降低分割选票的概率。
- 增加系统的稳定性：如果选举超时时间总是相同，可能导致系统在某些情况下无法及时选举出Leader，或者会频繁地发生选举冲突，影响系统的稳定性。通过随机化超时时间，可以使得选举过程更加灵活，并且避免节点频繁发起选举，提升系统的容错性和稳定性。
- 提高容错性和故障恢复速度：在Raft中，Leader故障后需要尽快重新选举出新的Leader，系统才可以继续提供服务。通过随机设置超时时间，可以确保在Leader失效后，系统中的某些节点会在不同的时间触发选举，增加选举过程的并行性，从而加速Leader的选举和故障恢复。

5.选举timeout设置太小会有什么影响?

如果选举超时时间设置得太小，可能会导致节点在收到Leader的正常心跳之前就触发选举，从而引发不必要的选举过程。这种情况会显著增加系统发生选举冲突和分割选票的概率，进而影响集群的稳定性。由于每个节点的选举定时器超时时间是随机的，如果超时时间过短，节点更容易在没有充分等待心跳的情况下触发选举，可能导致多个节点几乎同时发起选举请求，结果可能会出现选举分割的局面，增加系统的选举开销，降低系统的性能。同时，过短的超时时间还可能使得选举过程变得不可靠，影响系统的容错能力和稳定性。因此，选举超时时间必须足够长，以避免在收到心跳之前就触发选举，确保系统的正常运行。

体会和建议

在设计实现基于Raft算法的选举服务中，认识到了Raft算法是如何实现分布式一致性的。通过对Raft算法的深入研究和实际编码实现，使我们对算法的理论基础和实际应用有了更为全面的认识。

在实验的理论学习阶段，深入探讨了Raft算法的工作原理，包括其如何利用日志系统记录操作历史，并在分布式节点间维持状态一致性。详细分析了Raft中的三种角色——Leader、Follower和Candidate——及其相互转换的动态过程。通过对初始选举和Leader故障后的重新选举过程的模拟，我对Raft选举机制的理解更加深刻，尤其是其在处理分布式系统中的节点故障和网络分区问题时的鲁棒性。