

MIT 6.824 Lab 2A&2B&2C

目录

[一、前言](#)

[二、Lab 2A](#)

[1、2A问题构思与回溯](#)

[2、Raft结构体定义](#)

[3、初始化Raft节点](#)

[4、选主](#)

[5、心跳](#)

[6、ticker方法](#)

[7、2A测试情况](#)

[三、Lab 2B](#)

[1、2B问题构思与回溯](#)

[2、Start方法](#)

[3、ticker方法修改](#)

[4、日志复制RPC](#)

[5、2B测试情况](#)

[四、Lab 2C](#)

[1、2C问题构思与回溯](#)

[2、persist方法](#)

[3、readPersist方法](#)

[4、持久化状态](#)

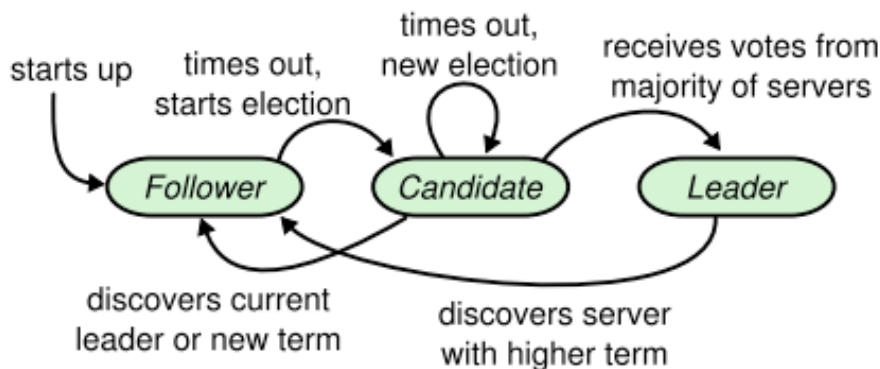
[5、2C测试情况](#)

[五、总结和心得](#)

前言

Raft 是一种为了实现分布式系统中数据一致性而设计的共识算法。它通过一系列规则确保一个集群中的多个服务器即使在出现故障的情况下也能保持状态同步。这种协议的关键特点在于它的易理解性和实用性，相比于Paxos算法，Raft 的结构更加直观，易于实现。

该协议的核心是领导者选举机制，它使得集群在任一时刻只有一个领导者负责管理日志复制。如下图所示，Raft 算法中节点在Follower、Candidate和Leader三种状态切换。领导者负责将日志条目复制到其他服务器，并在大多数节点同意后才提交更改。当领导者发生故障时，Raft 通过新的选举过程来保证集群的持续运作，并确保所有已提交的条目在新领导者上得到保留。



Raft 将共识问题细分为领导者选举、日志复制和安全三个子问题。它通过严格的日志匹配规则和多数派同意原则来保障系统的安全性，即使在领导者或服务器出现故障时也能保持系统状态的一致性。这种设计使得 Raft 不仅在理论上是可靠的，而且在实际应用中也广泛采用。

本次作业要完成的 MIT 6.824 Lab2 分为三个子实验：2A、2B 和 2C，分别对应 Raft 协议的不同组成部分。

在 Lab2A 中，任务是实现领导者选举和心跳机制。需要确保在分布式系统中，节点能够在没有领导者或领导者失败时进行选举，选出新的领导者，并通过心跳信息维持其权威。这个阶段不要求处理日志条目，仅确保领导者的选举和心跳机制的正确实施。

Lab2B 需要加入日志复制功能。在这一阶段，领导者需要将客户端的命令作为日志条目复制到集群中的其他节点，并在大多数节点写入日志后提交这些条目。此外，还需要处理日志不一致的情况，保证整个集群最终能够达成一致状态。

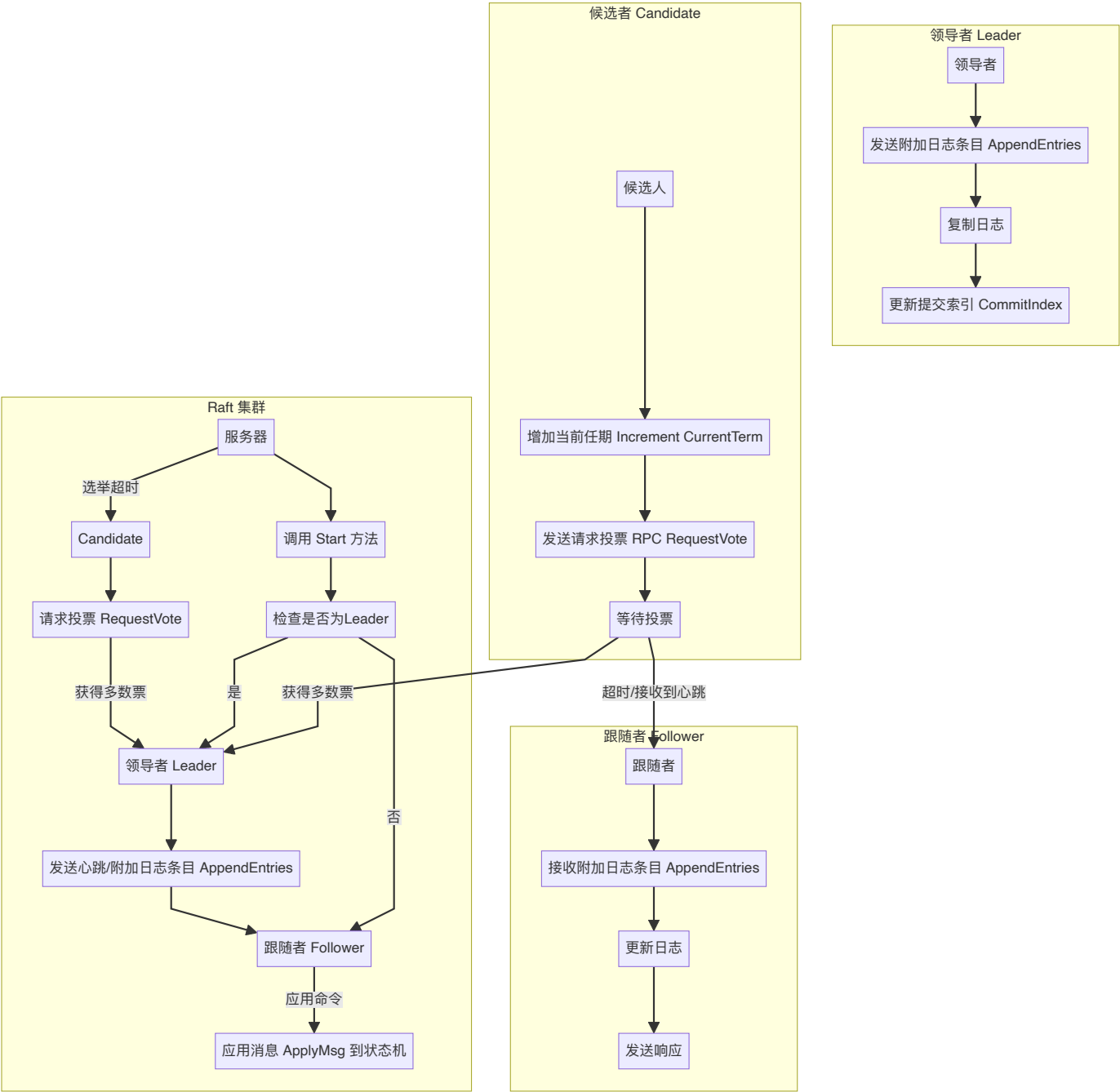
该Lab最难的也可以说是2B部分，因为它涉及到复杂的一致性问题 and 日志条目的复制机制。当领导者发生更换，或者网络分区导致日志不一致时，Raft协议必须能够妥善处理这些情况，确保数据最终一致，这就要求实现复杂的日志复制协商和回滚机制。领导者必须持续地追踪每个节点的日志状态，并在发现分歧时能够立即采取行动，使得所有节点能够快速收敛到相同的日志状态。

在实现日志复制时，首先要确保在正常运行的情况下，领导者能够有效地将日志条目追加到集群中的其他节点，并在大多数节点成功写入后提交这些日志条目。这个过程看似直接，但实际上要确保日志条目在所有节点上保持一致，尤其是在出现网络分裂或节点故障时。

除了常规的追加和提交，还需要设计和实现一套机制来处理一系列可能的异常情况。例如，当一个节点崩溃并重新启动时，它可能会错过一些日志条目的更新，或者它保存的日志可能与当前领导者的日志不一致。在网络延迟也可能导致日志条目的更新在不同节点上以不同的速度和顺序应用，造成日志状态的短暂分歧。所以设计的架构必须能够检测到这些不一致，并将节点上的日志同步到最新状态。日志冲突也会发生，如果两个节点针对同一个日志索引有不同的条目，或者一个新的领导者试图对已经存在的条目进行覆盖，设计的架构需要有策略来决定哪个条目是有效的，以及如何解决这种冲突。

Lab2C 的重点是实现状态的持久化。需要实现日志条目的持久化存储，以便在服务器宕机重启后能够恢复其状态。

最终完成的Raft架构可以用下图来表示：



Lab2A

2A问题构思和回溯

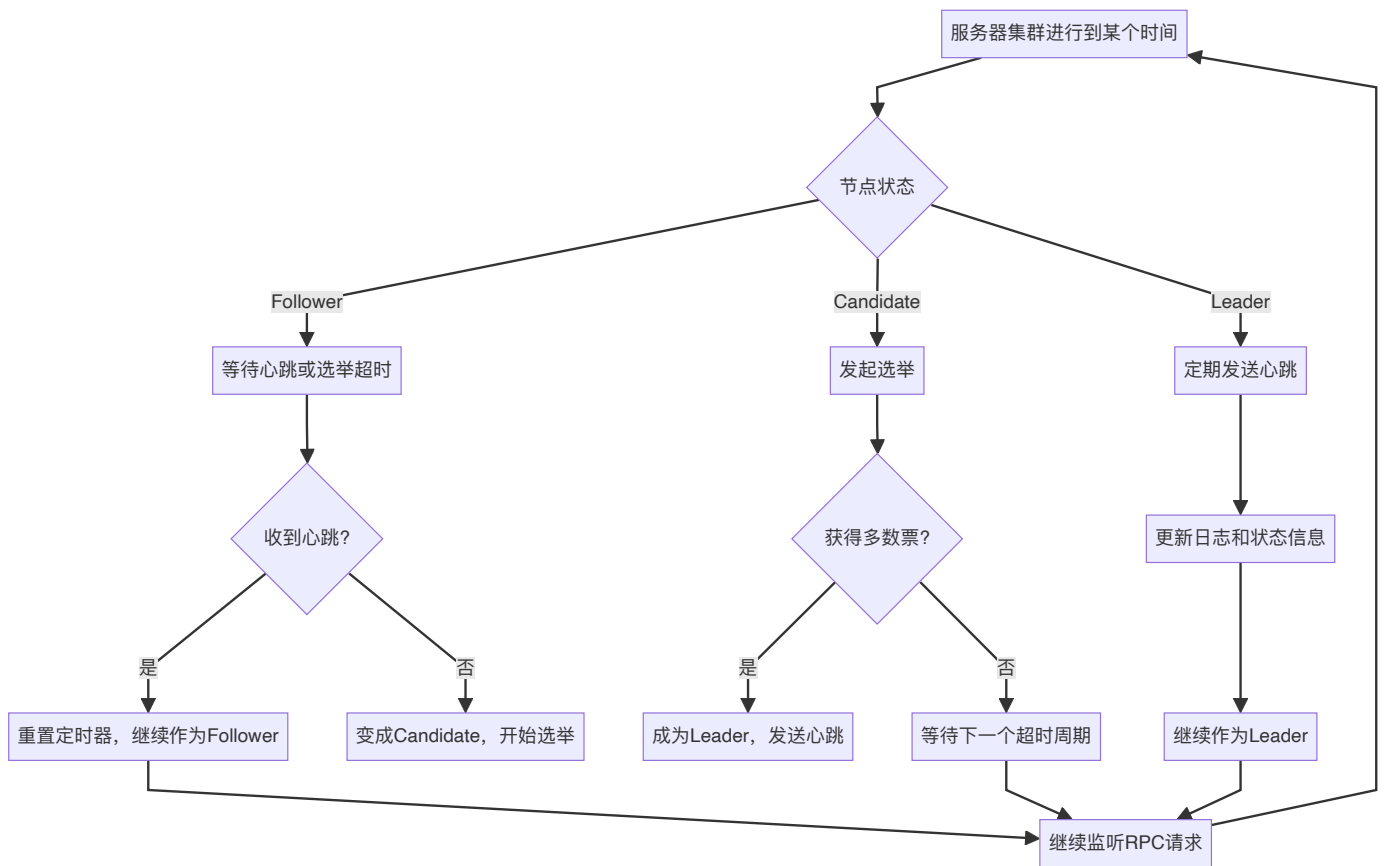
阅读Lab2A部分的Hint1可以知道核心任务是实现 Raft 分布式一致性算法中的 Leader 选举和心跳检测机制。这部分的主要目标包括：

1. **Leader 选举**:在无故障情况下，能够选举并维持一个 Leader。当原 Leader 故障或网络问题导致其数据包丢失时，能够选举出新的 Leader。
2. **心跳检测**:实现心跳机制，即 Leader 定期发送 AppendEntries RPC，2A的请求不包含日志条目。心跳的主要目的是防止其他服务器因为选举超时而发起不必要的 Leader 选举。

我对于2A的理解如下：

简单来说，2A其实就是先完成Raft的初始化，然后完成两种RPC的调用，对于Follower/Candidate来说，是实现选主RPC，它们有一个自身的定时器设置超时时间，当定时器结束之前没有收到来自Leader的心跳RPC，就要进行选举；对于Leader来说，是实现心跳RPC，Leader有一个心跳计时器，当计时器结束时，就要向其他节点发送心跳信息，以此来保证自己的领导地位。

该过程可以用如下流程图来表示：



Raft结构体定义

根据论文的Figure 2 可以定义节点的状态信息。

State	
Persistent state on all servers: (Updated on stable storage before responding to RPCs)	
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:	
commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on leaders: (Reinitialized after election)	
nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

所有服务器上的持久状态（在响应RPC之前更新到稳定存储）：

- **currentTerm**：服务器最后一次知道的任期号（初始化为0，以后单调增加）。这代表服务器所见过的最高任期号。
- **votedFor**：在当前任期内收到选票的候选人ID（如果没有则为null）。这记录了当前任期内该服务器投票给了哪个候选人。
- **log[]**：日志条目数组，每个条目包含用于状态机的命令以及领导者收到该条目时的任期号（第一个索引是1）。这是服务器上存储的所有日志条目的顺序集合。

所有服务器上的易失状态：

- **commitIndex**：已知的最高日志条目的索引号，已被提交到状态机（初始化为0，以后单调增加）。这代表了已经被复制到大多数服务器并可以安全应用到状态机的最高的日志条目索引。
- **lastApplied**：已经被应用到状态机的最高的日志条目的索引号（初始化为0，以后单调增加）。这表示了状态机最后一次应用日志条目的位置。

领导者上的易失状态（选举后重新初始化）：

- **nextIndex[]**：对于每个服务器，需要发送给它的下一条日志条目的索引号（初始化为领导者最后日志索引 + 1）。这个用于追踪要发送给每个服务器的下一条日志条目，以便日志复制。
- **matchIndex[]**：对于每个服务器，已知的被成功复制到该服务器的最高日志条目的索引号（初始化为0，以后单调增加）。这用于记录哪些日志条目已经被安全复制到特定的服务器。

```

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

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

    //——————论文Figure2中的状态——————//
    // 所有server都拥有的持久状态
    currentTerm int // 当前任期
    votedFor    int  // 为哪个节点投票
    logs        []LogEntry // 日志条目

    // 所有的servers都有的易失状态
    commitIndex int // 多于半数的节点同意更新的log的索引
    lastApplied int // 已经被应用到状态机的最高log的索引值

    // leader独有的易失状态
    nextIndex []int // 对于每一个服务器，需要发送给他的下一个log的索引值
    matchIndex []int // 对于每一个服务器，已经复制给他的log的最高索引值

    //——————自己定义的一些状态——————//
    // 本节点的状态
    status      Status // 本节点的状态
    overtime    time.Duration // 选举超时时间
    timer       *time.Ticker // 计时器

    applyChan chan ApplyMsg // 通道，用于存放已经提交的日志

```

```
}
```

对于节点身份的定义如下：

```
type Status int
const (
    Follower Status = iota
    Candidate
    Leader
)
```

对于日志条目的定义如下：

```
type LogEntry struct {
    Term    int           // Log的term
    Command interface{} // Log的command, 这里是一个interface, 可以存放任何类型的数据
}
```

初始化Raft节点（Make方法）

`Make()` 函数用于创建一个新的 Raft 节点。在这个函数中，要完成了 Raft 节点的初始化，包括各种状态的初始化、定时器的设置等。以下是编写 `Make()` 函数时的几点解释和关键点：

1. `rf` 就是 Raft 节点的实例对象，首先对其进行基本信息的初始化，包括节点的索引 `me`、与其他节点通信的 RPC 端点 `peers`、持久性存储 `persister` 以及日志应用通道 `applyCh`。
2. 对于 Leader 节点，需要初始化发送给其他服务器的下一个日志条目索引值 `nextIndex` 和已经复制给其他服务器的日志的最高索引值 `matchIndex`。
3. 每个节点的初始化状态都是Follower，并且通过 `Intn` 方法生成一个范围内的选举超时时间，这样的随机性是为了避免节点在相同的时间内触发选举，从而减少冲突和提高系统的稳定性。

以下是 `Make()` 函数的实现：

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

    // Your initialization code here (2A, 2B, 2C).
    rf.status = Follower
    rf.votedFor = -1
    // 随机选取选举超时时间
    rand.Seed(time.Now().UnixNano())
    rf.overtime = time.Duration(rand.Intn(150)+200) * time.Millisecond
    rf.currentTerm, rf.commitIndex, rf.lastApplied = 0, 0, 0
    rf.nextIndex, rf.matchIndex, rf.logs = nil, nil, []LogEntry{{0, nil}}
    rf.timer = time.NewTicker(rf.overtime)
    rf.applyChan = applyCh
```

```
// initialize from state persisted before a crash
rf.readPersist(persister.ReadRaftState())

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

return rf
}
```

`Make()` 函数确保了 Raft 节点在初始化时具备正确的状态和参数，为后续的协议执行奠定了基础。

选主（Follower和Candidate的动作）

该部分首先是定义两个基本的结构体，RequestVote RPC和其回复，然后是编写两个重要函数，`RequestVote` 和 `sendRequestVote`，`RequestVote` 用于处理来自其他节点的投票请求RPC，节点需要根据一系列繁琐的条件判断是否投票给请求的候选者；`sendRequestVote` 用于向其他节点发送投票请求，并处理对方的响应。

总的来说，选主需要遵循两条原则，第一条是竞选者的任期必须大于自己的任期。否则返回false。并且在出现网络分区时，可能两个分区分别产生了两个leader。那么我们认为应该是任期长的leader拥有的数据更完整。第二条是，投票成功的前提是，自己的票要投给别人并且竞选者的日志状态要和自己的一致。

RequestVote RPC的定义

RequestVote RPC和其回复的定义可以参照Figure 2。

RequestVote RPC	
Invoked by candidates to gather votes (§5.2).	
Arguments:	
term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)
Results:	
term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote
Receiver implementation:	
1. Reply false if term < currentTerm (§5.1)	
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)	

字段:

- term**: 候选者的任期号。
- candidateId**: 请求投票的候选者的ID。
- lastLogIndex**: 候选者的最后日志条目的索引值。
- lastLogTerm**: 候选者最后日志条目的任期号。

结果:

- term**: 返回给候选者的当前任期号，以便候选者更新自己的任期。
- voteGranted**: 如果为真，则表示候选者收到了选票。

接收者实现:

- 1.如果 term 小于 currentTerm，则回复 false。
- 2.如果 votedFor 是 null 或 candidateId，并且候选者的日志至少与接收者的日志一样新，则授予选票。

注意2A中hint的最后一条可知，定义RPC字段时要注意首字母要大写。

```
// RequestVote RPC 参数结构示例。
type RequestVoteArgs struct {
    // Your data here (2A, 2B).
    Term          int // 候选人的任期号
    CandidateId    int // 请求选票的候选人的 Id
    LastLogIndex   int // 候选人的最后日志条目的索引值
    LastLogTerm    int // 候选人最后日志条目的任期号
}

//
```



```
// example RequestVote RPC reply structure.
// field names must start with capital letters!
//
// 如果竞选者任期比自己小，拒绝投票
// 如果当前节点的votedFor为空或者是候选人，且候选人的日志至少和自己一样新，投票给候选人
type RequestVoteReply struct {
    // Your data here (2A).
    Term int // 当前任期号，以便于候选人去更新自己的任期号
    VoteGranted bool // 候选人赢得了此张选票时为真
    VoteState VoteState // 投票的状态
}
```

参照图中给出的投票RPC的返回字段可以定义 `RequestVoteReply`，我加入了 `VoteState` 字段，这是个枚举类型，用于之后选主的实现，根据不同的返回类型，Leader要进行不同的操作。

```
type RequestVoteReply struct {
    // Your data here (2A).
    Term int // 当前任期号，以便于候选人去更新自己的任期号
    VoteGranted bool // 候选人赢得了此张选票时为真
    VoteState VoteState // 投票的状态
}

// 枚举类型，节点投票的状态
type VoteState int
const (
    VoteNormal VoteState = iota //投票成功
    Killed //节点终止
    MsgExpire //投票(消息\竞选者)过期
    LogExpire //日志过期
    Voted //本Term内已经投过票
)
```

RequestVote 实现

我认为理解raft协议最关键的一点就是，每个节点在某个时间点所处的状态可能各不相同，这里的不相同并不单单指节点处于不同的status，而是说它们所处的“进程”不同，举个例子，当某个Follower的定时器到了后，它需要转变成Candidate让别的节点选它当Leader，这时候有很多节点可能仍处于倒计时的状态，所以对于这种异步的理解是非常重要的，这种异步的实现主要依靠每次节点自身的定时器都是一个范围内的随机时间。

在上面我提到，`RequestVote` 函数的作用主要是用于处理来自其他节点的投票请求RPC，通俗来说，收到RPC的节点这时可能处于倒计时状态，但是因为收到RPC而被打断，不得不进行处理，并根据RPC里的信息来采取一系列动作。

那了解 `RequestVote` 函数的功能后，如何编写它就十分清晰了，总的来说，`RequestVote` 函数需要先判断一次发送RPC的节点的任期和自身节点的任期，以此为准绳来考虑自身节点要不要改变状态。`RequestVote` 函数的逻辑是“判断任期——>分支1:对方比自己任期小，写回复——>分支2:对方比自己任期大，考虑是否投票，写回复——>重设计时器”

`RequestVote` 函数编写主要遵照以下几点：

1. 检查请求节点的任期是否小于自己的任期，如果是，拒绝投票并返回过期状态。

2. 如果请求节点的任期大于自己的任期，则将当前节点状态设置为 Follower，并更新当前任期，清除之前的投票。
3. 如果当前节点尚未投票，则检查请求候选者的日志是否足够新，如果是，则投票给该候选者，重置选举定时器，并返回投票成功状态。
4. 如果当前节点已经投票，返回已投票状态，并检查两种情况：
 - 若请求的候选者与当前节点上次投票的候选者相同，说明该请求可能是由于网络原因重复发送的，将当前节点状态设置为 Follower。
 - 否则，返回已投票状态。

在 `RequestVote` 函数中，我还添加了一些输出信息用于调试代码和观察节点的状态和动作，分别在节点收到 RPC 时和处理完后打印

`RequestVote` 函数的实现如下：

```
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    /*
        args: 请求投票 RPC
        reply: 用于处理请求的响应
    */

    // Your code here (2A, 2B).

    DPrintf("[INFO][RequestVote]Node %d: RequestVote called with args %+v\n", rf.me, args)
    // Your code here (2A, 2B).
    if rf.killed() {
        reply.Term = -1
        reply.VoteGranted = false
        reply.VoteState = Killed
        return
    }
    rf.mu.Lock()
    defer rf.mu.Unlock()

    // 对方任期小于自己，说明这个请求过期了
    if args.Term < rf.currentTerm { // 请求term更小，不投票
        reply.Term = rf.currentTerm
        reply.VoteGranted = false
        reply.VoteState = MsgExpire
        DPrintf("[INFO][RequestVote]Node %d: RequestVote - Vote Not Granted\n", rf.me)
        return
    }

    // 对方任期大于自己，需要更新自身状态
    if args.Term > rf.currentTerm {
        rf.status = Follower // 转变为追随者
        rf.currentTerm = args.Term // 更新最新任期
        rf.votedFor = -1 // 保存选票，还不要投出去，方便之后判断是否投了票
    }

    // 如果当前节点还没有投票
```

```

if rf.votedFor == -1 {
    lastLogIndex := len(rf.logs) - 1 // 获取当前节点最后一条日志的索引
    lastLogTerm := 0
    // 当前节点日志的最后一条日志的索引不为-1，表示日志不为空，那么将currentTerm设置为最后一条日志的任期
    if lastLogIndex >= 0 {
        lastLogTerm = rf.logs[lastLogIndex].Term
    }

    // If votedFor is null or candidateId, and candidate's log is at least as up-to-date
    as receiver's log, grant vote (§5.2, §5.4)
    // 论文(§5.2, §5.4)的一个条件判断，如果候选者的日志不至少和当前节点的日志一样新，就不授予投票
    // 如果候选者的 args.LastLogIndex 小于当前节点的 lastLogIndex 或者候选者的 args.LastLogTerm
    小于当前节点的 lastLogTerm，说明候选者的日志不足够新，无法给予投票。
    if args.LastLogIndex < lastLogIndex || args.LastLogTerm < lastLogTerm {
        reply.VoteState = LogExpire
        reply.VoteGranted = false
        reply.Term = rf.currentTerm

        // fmt.Printf("[INFO] Node %d: RequestVote - Condition (§5.2, §5.4) Not Met\n",
rf.me)
        return
    }

    // 一切给票的条件都满足后，就可以投出这张票了
    rf.votedFor = args.CandidateId // 投票给该候选者
    reply.VoteState = VoteNormal    // 投票状态成功
    reply.Term = rf.currentTerm     // 回复中的任期更新为当前节点的任期
    reply.VoteGranted = true        // 授予投票，也就是已投票
    rf.timer.Reset(rf.overtime)    // 重置定时器

    DPrintf("[INFO][RequestVote]Node %d: RequestVote - Vote Granted\n", rf.me)
} else { // 当前节点票已经给出去了（已经投出去了，但是由于网络或者其他原因又收到一个RPC）

    reply.VoteState = Voted // 标记为已经投过票了
    reply.VoteGranted = false // 不授予投票权

    // 检查一下投出去的这张票是不是就是当前的候选者
    if rf.votedFor != args.CandidateId {
        // DPrintf("[INFO] Node %d: RequestVote - Candidate ID Does Not Match\n", rf.me)
        return
    } else { // 是当前的候选者，但是因为某些原因重发了
        // 重置自身状态
        rf.status = Follower
    }

    // 重置定时器
    rf.timer.Reset(rf.overtime)

    DPrintf("[INFO][RequestVote]Node %d: RequestVote - Vote Not Granted\n", rf.me)
}

```

```
return
}
```

sendRequestVote 实现

上面提到，`sendRequestVote` 函数有两个功能，一个是Candidate向其他节点发送请求投票RPC的信息，二是Candidate要处理来自其他节点的回复，统计自身的票数看能否当Leader，那么代码逻辑就是“调用RPC发送请求，直到调用成功<—>处理来自其他节点的请求”，这两个功能的进行也不是顺序而是并行的，所以要在代码中用加锁进行隔离。

`sendRequestVote` 函数编写主要遵照以下几点：

1. 调用 RPC 向目标节点发送投票请求，并等待响应。
2. 若 RPC 调用失败，重试直到成功或节点被关闭。
3. 处理对方的响应，主要处理以下情况：
 - 如果响应表明对方请求的任期小于当前节点的任期，将当前节点状态设置为 Follower，更新当前任期，清除之前的投票，并重置选举定时器。
 - 如果响应表明对方请求的任期过期，根据相应状态进行处理。
 - 如果响应表明对方正常投票或已经投票，根据是否同意投票和当前任期，收集选票数量。
 - 如果选票数量超过半数，成为 Leader，初始化 `nextIndex` 并重置心跳定时器。

以下是 `sendRequestVote` 函数的实现：

```
func (rf *Raft) sendRequestVote(server int, args *RequestVoteArgs, reply
*RequestVoteReply, voteNum *int) bool {
    /*
        server: 目标服务器的索引
        args: 请求投票RPC
        reply: 接受请求投票RPC的响应
        voteNum: 记录已收到的投票数量

        加锁前：向目标服务器发送请求投票RPC的调用
        加锁后：对RPC的回复进行处理
    */

    // 判断一次节点是否被终止，如果是，则停止请求投票
    if rf.killed() {
        return false
    }

    // 调用rf.peers[server].Call方法，向目标服务器发送请求投票RPC的调用
    ok := rf.peers[server].Call("Raft.RequestVote", args, reply)

    // 失败了那就一直调用该方法，直到调用成功
    for !ok {
        if rf.killed() {
            return false
        }
        ok = rf.peers[server].Call("Raft.RequestVote", args, reply)
    }
}
```

```

}

// 多线程，加锁
rf.mu.Lock()
defer rf.mu.Unlock() // 函数执行完毕后解锁

if args.Term < rf.currentTerm { // 过期请求
    return false
}

switch reply.VoteState {
case MsgExpire, LogExpire: // 如果消息过期了
{
    rf.status = Follower // 转变自己的状态为Follower
    rf.timer.Reset(rf.overtime)
    if reply.Term > rf.currentTerm { // 更新当前节点的任期为回复中的任期
        rf.currentTerm = reply.Term
        rf.votedFor = -1
    }
}
case VoteNormal, Voted:
    //根据是否同意投票，收集选票数量
    if reply.VoteGranted && reply.Term == rf.currentTerm && *voteNum <= (len(rf.peers)/2)
{
        *voteNum++
    }

    if *voteNum >= (len(rf.peers)/2)+1 {
        DPrintf("[INFO][sendRequestVote]Node %d: Received majority votes, becoming
Leader\n", rf.me)

        *voteNum = 0
        // 如果该节点之前就是Leader
        if rf.status == Leader {
            return ok
        }

        // 如果之前不是Leader，那么需要初始化nextIndex数组
        // nextIndex 数组中的每个元素对应一个 Follower 节点，表示 Leader 向该节点发送日志的下一个位置。
        rf.status = Leader
        rf.nextIndex = make([]int, len(rf.peers))
        for i, _ := range rf.nextIndex {
            rf.nextIndex[i] = len(rf.logs) + 1
        }
        rf.timer.Reset(HeartBeatTimeout) // 重置节点的定时器为心跳定时时间
    }
case Killed:
    return false
}
return ok
}

```

心跳 (Leader和Follower的动作)

心跳机制，或者说是以心跳机制为基础实现的日志复制机制（2B），我认为跟选主的逻辑是一样的。为什么这么说呢？我思考了很久如何将这个问题答案讲明白。

实际上，选主只是Follower或Candidate的动作，心跳只是Leader的动作，打个比方，有一个王国，国王的唯一任务就是隔一段心跳时间就向国民发送同步日志的信息（虽然这里我没有为这种日志同步找一个合适的比方），而王国所有国民就一个动作，只要定时器到了就发动选举，听起来很荒谬，但Raft就是这样运行的，而不管是哪种动作，都是靠发送RPC和处理回复来实现的。

所以心跳仍然是先定义Append Entries和其回复，然后要完成两个函数，一个是发送RPC，处理回复的函数，另一个是处理RPC的函数，我把它们定义为 AppendEntries 和 sendAppendEntries。AppendEntries 用于处理其他节点发送的心跳或同步日志的请求，在2A中即实现心跳，sendAppendEntries 是向其他节点发送心跳或同步日志的请求。

AppendEntries RPC的定义

AppendEntries RPC和其回复的定义可以参照Figure 2。

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

字段:

- term: 领导者的任期号。
- leaderId: 领导者的ID，便于跟随者重定向客户端。
- prevLogIndex: 新日志条目之前的日志条目索引。
- prevLogTerm: prevLogIndex处日志条目的任期号。
- entries[]: 要存储的日志条目（心跳时为空；为了效率，可能会发送多个条目）。
- leaderCommit: 领导者已提交的日志的最高索引值。

结果:

- term: 当前任期号，供领导者更新自己。
- success: 如果跟随者包含与prevLogIndex和prevLogTerm匹配的条目，则为真。

接收者实现:

- 1.如果 term 小于 currentTerm，则回复 false。
- 2.如果日志中不包含与 prevLogIndex 匹配的 prevLogTerm 的条目，则回复 false。
- 3.如果现有的条目与新条目发生冲突（索引相同但任期号不同），删除现有的条目以及所有跟随它的条目。
- 4.追加任何尚未在日志中的新条目。
- 5.如果 leaderCommit 大于 commitIndex，则将 commitIndex 设置为 leaderCommit 与新条目最后索引中的最小值。

依然注意2A中hint的最后一条可知，定义RPC字段时要注意首字母要大写。

```

type AppendEntriesArgs struct {
    Term          int          // 任期号
    LeaderId      int          // Leader标识
    PrevLogIndex  int          // 每个节点的nextIndex前一处的日志的索引
    PrevLogTerm   int          // nextindex前一处的任期号
    Entries       []LogEntry // 需要复制的日志
    LeaderCommit  int          // leader已经提交的最大日志索引
    LogIndex      int
}

```

根据图中要求，心跳RPC的回复可以定义为如下结构，其中我添加了字段 `AppState`，用于Leader处理日志复制的不同情况。

```

type AppendEntriesReply struct {
    Term      int          // 用于让leader更新
    Success    bool          // 日志复制是否成功
    AppState  AppendEntriesState // 日志复制状态
}

// 枚举日志复制状态
type AppendEntriesState int
const (
    AppNormal      AppendEntriesState = iota // 复制正常
    AppExpire      // 复制过时
    AppKilled      // 节点终止
    AppRepeat      // 复制重复
    AppCommitted   // 复制的日志已经提交
    AppMismatch    // 复制不匹配
)

```

AppendEntries 实现

经过上面的分析，再通过论文，可以总结 `AppendEntries` 的逻辑“根据日志同步中的Term进行判断——>分支1：对方超时，告诉它已经有更高的Term了——>分支2：继续追随”。

`AppendEntries` 函数编写过程和关键点如下：

1. 先判断当前节点是否已经关闭，如果是，则直接返回相应的关闭状态。
2. 根据请求参数 `args` 中的 `Term` 与当前节点的 `currentTerm` 进行比较：
 - 如果请求的任期小于当前节点的任期，说明请求来自一个过时的 Leader 或网络分区，返回相应的状态。
 - 否则，更新当前节点的状态为 Follower，并重置相关状态，包括当前节点的任期、投票给的节点等。
3. 返回响应，包括当前节点的状态、当前节点的任期、响应状态等信息。

```

func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    /*
        args: 心跳/日志同步RPC
    */
}

```



```

    reply: 心跳/日志同步RPC的响应
    */

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

    // 收到了心跳/日志同步的调用
    DPrintf("[INFO][AppendEntries][收到 RPC]Node %d: AppendEntries called with args %+v\n",
rf.me, args)

    // 检查一次当前节点是否被终止
    if rf.killed() {
        reply.AppState = AppKilled // 回复状态设置为节点已经被终止
        reply.Term = -1
        reply.Success = false //

        // 打印信息, 表明节点已经终止
        DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Node is killed,
returning AppKilled\n", rf.me)
        return
    }

    // 如果RPC中的任期比自己的还小, 说明对方是一个过时的Leader, 或者是来自不同的网络分区
    if args.Term < rf.currentTerm {
        reply.AppState = AppOutOfDate // 当前节点回复的应用状态设置为过时
        reply.Term = rf.currentTerm
        reply.Success = false

        DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Current Term: %d,
Requested Term: %d, returning AppOutOfDate\n", rf.me, rf.currentTerm, args.Term)
        return
    }

    // 进行同步, 更新信息和Leader一致
    rf.currentTerm = args.Term
    rf.votedFor = args.LeaderId
    rf.status = Follower
    rf.timer.Reset(rf.overtime) // 重置定时器为超时时间

    DPrintf("[INFO][AppendEntries][同步 完成]Node %d: AppendEntries - Current Term: %d,
VotedFor: %d, Status: %d, Resetting timer, returning AppNormal\n", rf.me, rf.currentTerm,
rf.votedFor, rf.status)

    reply.AppState = AppNormal
    reply.Term = rf.currentTerm
    reply.Success = true
    return
}

```

sendAppendEntries 实现

`sendAppendEntries` 的逻辑是“发送心跳/日志同步的RPC<—>处理回复——>分支1: 正常返回——>分支2: 得知自己过期”。

`sendAppendEntries` 函数编写过程和关键点如下:

1. 首先判断当前节点是否已经被关闭, 如果是, 则直接返回 `false`。
2. 使用 `Call` 方法调用目标节点的 `AppendEntries` RPC, 传递请求参数 `args`, 并等待响应。
3. 如果 RPC 调用失败, 则进行重试, 直到成功或节点被关闭。
4. 在获取响应后, 根据响应的 `AppState` 进行分支处理:
 - 如果目标节点已经关闭 (`AppKilled`), 则返回 `false`。
 - 如果目标节点正常响应 (`AppNormal`), 说明心跳或同步成功, 返回 `true`。
 - 如果目标节点已经过时 (`AppOutOfDate`), 说明出现网络分区或 Leader 过时, 将当前节点状态设置为 `Follower`, 并重置相关状态, 返回 `true`。

```
func (rf *Raft) sendAppendEntries(server int, args *AppendEntriesArgs, reply
*AppendEntriesReply, appendNums *int) bool {
    /*
        server: 目标服务器的索引
        args: 日志同步RPC
        reply: 日志同步RPC的响应
        appendNums: 记录已收到的日志同步RPC的数量

        加锁前: 向目标服务器发送日志同步RPC的调用
        加锁后: 对RPC的回复进行处理
    */

    // 判断一次节点是否被终止, 如果是, 则停止发送日志同步RPC
    if rf.killed() {
        return false
    }

    // 使用 rf.peers[server].Call 方法, 向目标服务器发送日志同步 RPC 的调用
    ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
    // 失败了那就一直调用该方法, 直到调用成功
    for !ok {
        if rf.killed() {
            return false
        }
        ok = rf.peers[server].Call("Raft.AppendEntries", args, reply)
    }

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

    // 处理日志同步 RPC 的回复
    switch reply.AppState {
    case AppKilled: // 目标节点终止
        {
```

```

        DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d is killed\n", rf.me,
server)
        return false
    }

    case AppNormal: // 目标节点正常返回
    {
        DPrintf("[INFO][sendAppendEntries][成功]Node %d: Node %d returned normally,
returning true\n", rf.me, server)
        return true
    }

    case AppOutOfDate: // reason: 出现网络分区, 该 Leader 已经 OutOfDate (过时)

        // 该节点变成追随者, 重置节点状态
        rf.status = Follower
        rf.votedFor = -1
        rf.timer.Reset(rf.overtime)
        rf.currentTerm = reply.Term

        // 打印信息, 表明当前节点变成 Follower
        DPrintf("[INFO][sendAppendEntries][过期]Node %d: Node %d is OutOfDate, converting to
Follower\n", rf.me, server)
    }
    return ok
}

```

ticker方法（为节点的动作提供支持）

`Ticker()` 函数的编写是我遇到的一个难点，这里的难点主要是难以理解工作机制，在我充分理解论文和raft协议后，我总结了Ticker函数实际上是要完成两个计时器的功能，分别用于选举计时和心跳计时，但不不管是Follower还是Candidate，还是Leader，在定时器结束时都要调用这一个 `Ticker`。

1. **选举计时器**：对于跟随者（Follower）和候选者（Candidate）状态的节点，在定时器到期时会触发选举相关的操作，例如开始一轮新的选举。这部分的计时器在 `Follower` 和 `Candidate` 状态下进行处理。
2. **心跳计时器**：对于领导者（Leader）状态的节点，在定时器到期时会触发心跳（或日志同步）的操作，向其他节点发送 `AppendEntries` RPC请求。这部分的计时器在 `Leader` 状态下进行处理。

那了解了要实现的这两种计时器的功能后，编写函数代码就比较轻松了。虽然说是两种计时器，其实就是先通过判断节点是哪种状态来选择哪种计时器，如果是选举计时器，代码的整体逻辑就是“重设自身计时器的时间——>编写请求投票RPC——>发给其他节点，准备接收回复”，如果是心跳计时器，代码的整体逻辑就是“重设心跳计时器——>编写心跳/日志同步RPC——>发给其他节点，准备接收回复”。以下是对节点分别处于三种状态时，`Ticker()` 函数的具体操作：

1. 节点为Follower状态，定时器超时后，节点转变为Candidate状态；
2. 节点为Candidate状态：
 - 初始化选举参数，包括增加当前任期、为自己投票，以及记录自己已获得的一票。
 - 重置选举超时时间，以随机生成一个新的超时时间，避免多个节点在相同时间发起选举。
 - 向其他节点发送RequestVote RPC请求，尝试获得选票。

3. 节点为Leader状态:

- 定时器超时后, 进行心跳机制, 向其他节点发送AppendEntries RPC请求, 维持自己的领导地位。
- 构造AppendEntries参数, 包括当前任期、领导节点ID、空日志条目等。
- 重置心跳超时时间, 以保持固定的心跳间隔。
- 向其他节点并发发送心跳/日志同步请求。

在初始化阶段 (Make() 函数), 已经为每个 timer 设置了一个 overtime, 通过 select 监听, 时间超时后, 根据当前节点的状态执行相应的选举或日志同步操作。

```
func (rf *Raft) ticker() {  
    /*  
        每个节点都有一个定时器, 用于定时检查自己的状态, 如果发现自己的状态不对, 就会进行状态转换。  
    */  
  
    // 如果节点没有被终止, 那么就一直循环  
    for rf.killed() == false {  
  
        // Your code here to check if a leader election should  
        // be started and to randomize sleeping time using  
  
        // 当定时器结束进行超时选举  
        select {  
  
            // 定时器超时时会执行这个分支  
            case <-rf.timer.C:  
                if rf.killed() {  
                    return  
                }  
                // 多线程, 加锁  
                rf.mu.Lock()  
                // 根据节点的状态进行不同的分支操作  
                switch rf.status {  
  
                    //—————节点是Follower状态—————//  
                    case Follower:  
                        rf.status = Candidate // 转变为Candidate状态  
                        fallthrough           // 继续执行下面的代码  
  
                    //—————节点是Candidate状态—————//  
                    case Candidate:  
                        rf.currentTerm += 1 // 初始化自身的任期  
                        rf.votedFor = rf.me  
                        votedNums := 1 // 开始统计自身票数, 因为自己已经投给自己了, 所以票数为1  
  
                        // 每轮选举开始时, 重新设置选举超时  
                        rf.overtime = time.Duration(150+rand.Intn(200)) * time.Millisecond // 随机产生150-  
                        300ms的超时时间  
                        rf.timer.Reset(rf.overtime)
```

```

// 对自身以外的所有节点发送请求投票RPC
for i := 0; i < len(rf.peers); i++ {

    // 不用给自己发, 所以跳过
    if i == rf.me {
        continue
    }

    // 准备请求投票RPC的参数
    voteArgs := RequestVoteArgs{
        Term:          rf.currentTerm,
        CandidateId:   rf.me,
        LastLogIndex:  len(rf.logs) - 1,
        LastLogTerm:   0,
    }

    // 如果日志不为空, 那么就将最后一条日志的任期号传过去
    if len(rf.logs) > 0 {
        voteArgs.LastLogTerm = rf.logs[len(rf.logs)-1].Term
    }

    // 用于接收回复
    voteReply := RequestVoteReply{}

    // 使用go关键字启动一个新的goroutine, 异步发送请求投票RPC
    go rf.sendRequestVote(i, &voteArgs, &voteReply, &votedNums)
}

//—————节点是Leader状态—————//
case Leader:
    // 进行心跳/日志同步
    appendNums := 1 // 对于正确返回的节点数量
    rf.timer.Reset(HeartBeatTimeout)

    // 对自身以外的所有节点发送日志同步RPC
    for i := 0; i < len(rf.peers); i++ {

        // 不用跟自己同步, 所以跳过
        if i == rf.me {
            continue
        }

        // 准备日志同步RPC的参数
        appendEntriesArgs := AppendEntriesArgs{
            Term:          rf.currentTerm,
            LeaderId:      rf.me,
            PrevLogIndex:  0,
            PrevLogTerm:   0,
            Entries:       nil,
            LeaderCommit:  rf.commitIndex,
        }

```

```
// 用于接收回复
appendEntriesReply := AppendEntriesReply{}

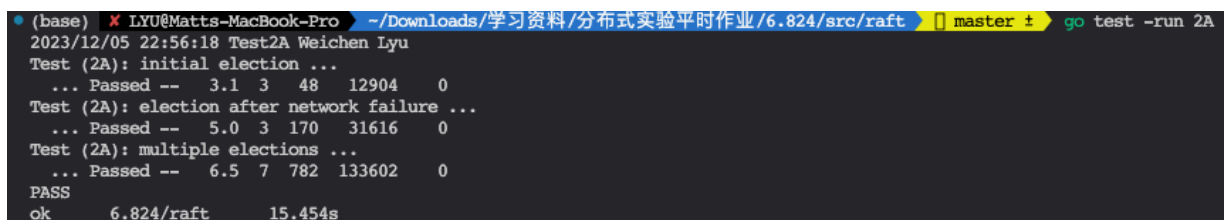
// 启动一个新的goroutine, 异步发送日志同步RPC
go rf.sendAppendEntries(i, &appendEntriesArgs, &appendEntriesReply, &appendNums)
}
}

rf.mu.Unlock()
}

}
}
```

2A测试情况

首先是通过情况（无输出日志）：



```
(base) x LYU@Matts-MacBook-Pro ~/Downloads/学习资料/分布式实验平时作业/6.824/src/raft [master ±] go test -run 2A
2023/12/05 22:56:18 Test2A Weichen Lyu
Test (2A): initial election ...
... Passed -- 3.1 3 48 12904 0
Test (2A): election after network failure ...
... Passed -- 5.0 3 170 31616 0
Test (2A): multiple elections ...
... Passed -- 6.5 7 782 133602 0
PASS
ok      6.824/raft      15.454s
```

可以看到，三个测试都完美通过。在raft/util.go中将参数const Debug改为true，可以打印之前编写的输出日志，执行下面这行命令可以将日志保存下来

```
go test -run 2A > test2A.log
```

Lab2B

2B问题构思与回溯

在实现Lab2B的过程中，我的目标是构建并完善 Raft 协议的日志复制功能。

首先需要实现的是 `start()` 方法。这个方法的核心是为新的日志条目启动一致性达成过程，这意味着每个条目都必须在大多数节点上得到复制并且提交，从而确保整个集群的状态机能够应用这些条目。在2A的实现中，`ticker()` 函数并没有考虑到日志条目的复制初始化，这是我们必须要解决的问题。`ticker()` 函数的作用是监控和触发关键事件，如选举超时和心跳信号的发送，但为了实现日志复制，我需要对其进行调整，确保它能够在领导者选举成功后，正确初始化并管理日志复制的过程。

2B中最为复杂的部分是日志复制机制的 RPC 调用修改。这涉及到 `sendAppendEntries` 和 `AppendEntries` 这两个关键函数。在这些函数中，不仅要实现基本的日志条目发送和接收逻辑，还要嵌入复杂的选举限制，正如 Raft 论文第 5.4.1 节所描述。这些限制的目的是为了确保集群在日志不一致的情况下，能够阻止错误的领导者节点进行状态更新，从而保护系统的数据一致性。

Start方法

根据Hint的第一条，首先要实现的函数是 `start`。


```
// the service using Raft (e.g. a k/v server) wants to start
// agreement on the next command to be appended to Raft's log. if this
// server isn't the leader, returns false. otherwise start the
// agreement and return immediately. there is no guarantee that this
// command will ever be committed to the Raft log, since the leader
// may fail or lose an election. even if the Raft instance has been killed,
// this function should return gracefully.
//
// the first return value is the index that the command will appear at
// if it's ever committed. the second return value is the current
// term. the third return value is true if this server believes it is
// the leader.
```

根据 `Start` 函数给出的注释，可以知道这个函数的目的是启动日志复制。当服务要提交一个新命令时，它调用 `Start` 函数，如果当前节点是领导者，它就会负责开始一致性协议。如果当前节点不是领导者，它将返回 `false`，表明该节点不具备启动一致性协议的资格。

以下是的 `Start` 函数的编写：

```
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    /*
        返回值1：日志同步命令的索引
        返回值2：当前的任期
        返回值3：是否是领导bool
    */
    index := -1
    term := -1
    isLeader := true

    // Your code here (2B).
    rf.mu.Lock()
    defer rf.mu.Unlock()

    // 不是Leader
    if rf.status != Leader {
        return index, term, false
    }

    isLeader = true

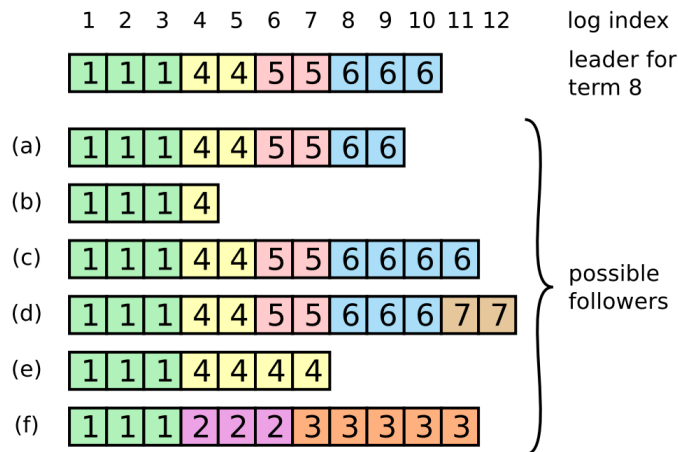
    // 初始化日志条目。并进行追加
    appendLog := LogEntry{Term: rf.currentTerm, Command: command}
    rf.logs = append(rf.logs, appendLog)
    index = len(rf.logs)
    term = rf.currentTerm

    return index, term, isLeader
}
```

ticker方法修改

在2A中并没有完成日志复制的相关部分，所以在2B中要进行补充，主要修改的内容是实现节点是Leader时要向除自身以外的其他节点发送用于日志复制的AppendEntries RPC，维持其的Leader状态。以下是对ticker函数修改的解释：

当定时器终止时，节点是Leader，也即Leader的心跳定时器终止时，Leader会遍历所有的Follower，向它们发送日志条目。在2A中，我已经构建了 AppendEntries RPC，但是没有初始化AppendEntries用于日志复制。根据论文图7，Entries要初始化为从 `nextIndex` 开始的所有日志条目。举个例子，在图7中，Leader要复制给 Follower 7的日志条目是从log index = 4开始的所有日志条目。但是，如果Leader不知道要从哪里开始向 Follower发送日志条目，也就是说 `nextIndex` 此时是初始设置（领导者日志的最后一个条目的索引加1），此时再把RPC发送给Follower，就会出现不匹配的情况，这种不匹配的情况我在之后章节中讨论。这里ticker函数的修改还不需要考虑不匹配的问题。



在构建日志复制RPC时，我添加了一个字段LogIndex，这个参数是Leader当前日志的最新索引。它可以帮助Leader确定向每个Follower发送哪些日志条目。Leader使用 `LogIndex` 来标记每个Follower应该接收的日志条目的末尾索引，从而确保所有Follower最终都能获取到最新的日志状态。当Follower收到带有 `LogIndex` 的 `AppendEntries` RPC时，它使用这个索引来检查自己的日志是否与Leader同步，特别是在处理日志不一致时。如果Follower的日志落后于 `LogIndex`，它将需要更新自己的日志以匹配Leader的状态。

ticker函数另外需要补充的是设置 `PrevLogIndex` 和 `PrevLogTerm`，`PrevLogIndex` 是领导者在发送所有复制日志条目前一个日志的索引，`PrevLogTerm` 是Leader在 `PrevLogIndex` 处的任期号，这两个参数将协助Follower进行一致性检验。我对于此的理解就是，Follower会根据 `PrevLogIndex` 和 `PrevLogTerm` 找到自己与当前Leader最后一次同步的日志索引，并告诉Leader。

关键代码的修改如下：

```
//-----节点是Leader状态-----//
case Leader:
    // 当前节点是leader，需要定期发送心跳
    appendNum := 1 // 初始化计数器

    rf.timer.Reset(HeartBeatTimeout) // 重置心跳计时器

    // 遍历所有的peers，发送心跳或日志条目
    for i, _ := range rf.peers {
        if i == rf.me {
            continue // 跳过自己，不需要给自己发送心跳
        }

        // 构造心跳/日志复制RPC
```

更新

```
appendEntriesArgs := &AppendEntriesArgs{
    Term:          rf.currentTerm,    // 当前Leader的任期
    LeaderId:      rf.me,             // Leader的ID (即当前节点的ID)
    PrevLogIndex:  0,                 // 初始化PrevLogIndex
    PrevLogTerm:   0,                 // 初始化PrevLogTerm
    Entries:       nil,               // 初始化Entries为空, 表示心跳; 如果有日志复制则会被

    LeaderCommit: rf.commitIndex,     // Leader的提交索引
    LogIndex:     len(rf.logs) - 1,    // 当前日志的最新索引
}

// 更新PrevLogIndex和PrevLogTerm以匹配下一个要发送的日志条目
for rf.nextIndex[i] > 0 {
    appendEntriesArgs.PrevLogIndex = rf.nextIndex[i] - 1
    if appendEntriesArgs.PrevLogIndex >= len(rf.logs) {
        rf.nextIndex[i]--
        continue
    }
    appendEntriesArgs.PrevLogTerm = rf.logs[appendEntriesArgs.PrevLogIndex].Term
    break
}

// 如果有新的日志条目要发送, 更新Entries字段
if rf.nextIndex[i] < len(rf.logs) {
    appendEntriesArgs.Entries = rf.logs[rf.nextIndex[i] :
appendEntriesArgs.LogIndex+1]
}

// 准备接收响应的结构体
appendEntriesReply := new(AppendEntriesReply)

// 异步发送心跳/日志复制请求
go rf.sendAppendEntries(i, appendEntriesArgs, appendEntriesReply, &appendNum)
}
```

日志复制RPC

同样, 实现日志复制RPC, 还是要围绕2A有关Leader动作的两个函数 `AppendEntries` 和 `sendAppendEntries`。

AppendEntries修改

之前我在章节3.3提到了日志不匹配的问题, 实际上, 根据论文, 不匹配的情况一共有下面两种:

- Follower首先检查自己的日志中是否有 `PrevLogIndex` 索引处的条目。也就是如果 `PrevLogIndex` 大于当前日志条目的最大下标, 说明Follower此时有一部分日志是缺失的, 这样Follower要回溯并告诉Leader正确的 `nextIndex` 值
- 如果该索引处有条目, 跟随者接着检查该条目的任期号是否与 `PrevLogTerm` 匹配, 不匹配的话, 也说明Follower有一部分日志是缺失的, 此时需要回溯 `nextIndex`

具体修改的逻辑如下:

1. 更新当前任期和节点状态：

当一个Follower收到 `AppendEntries` 请求时，它首先比较请求中的任期与自己的当前任期。如果请求的任期小于自己的当前任期，它会拒绝该请求。如果请求的任期大于或等于自己的当前任期，Follower会更新自己的任期，并将自己的状态更新为Follower，以接受来自新Leader的日志条目。

2. 处理日志不匹配的两种情况

3. 日志复制

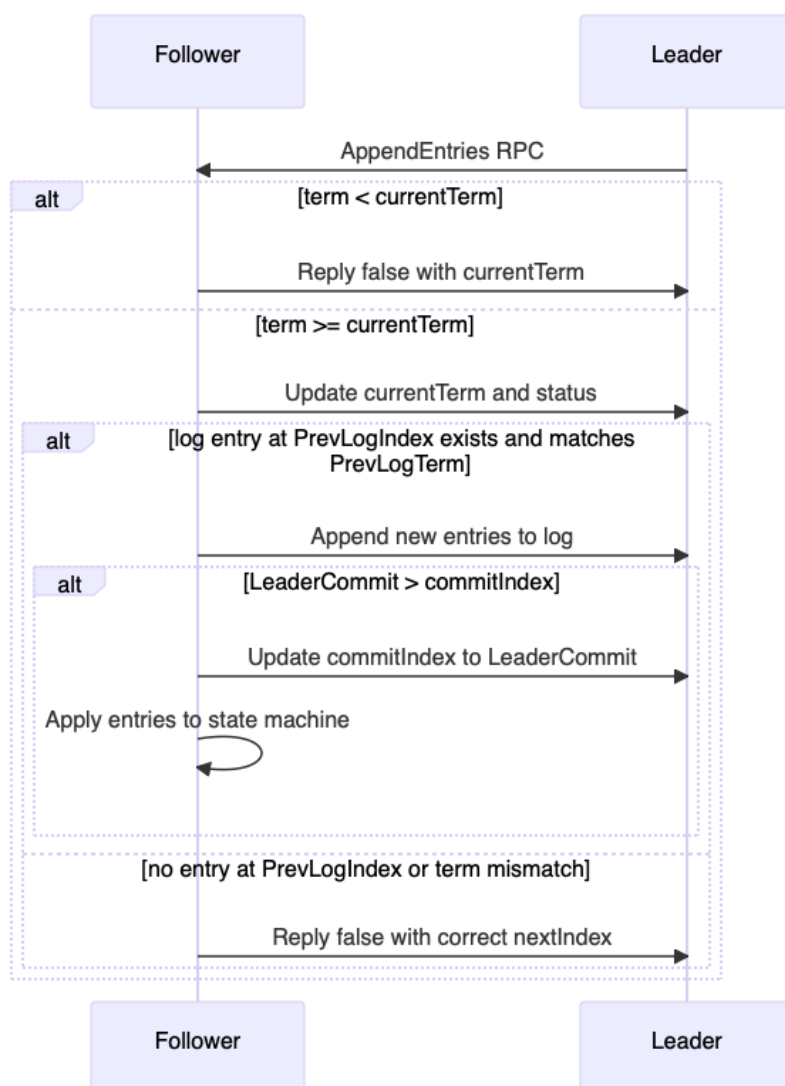
如果日志匹配，Follower会根据Leader的请求添加新的日志条目。这涉及到截断自己的日志（如果有冲突的条目），并追加来自Leader的新日志条目。

更新完日志后，如果 `LeaderCommit` 大于Follower的 `commitIndex`，Follower会更新自己的 `commitIndex`。

4. 处理日志已提交的情况

如果Follower的日志已经超前于Leader的请求（即 `PrevLogIndex` 小于Follower的 `lastApplied`），Follower会设置 `reply.AppState` 为 `AppCommitted`，指示Leader它已经提交了这部分日志。

`AppendEntries` 修改后的逻辑可以用下图来表示



修改后的 `AppendEntries` 函数如下：

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

```

/*
    args: 心跳/日志同步RPC
    reply: 心跳/日志同步RPC的响应
*/

// 收到了心跳/日志同步的调用
DPrintf("[INFO][AppendEntries][收到 RPC]Node %d: AppendEntries called with args %+v\n",
rf.me, args)

// 检查一次节点是否被终止
if rf.killed() {
    reply.Term = -1
    reply.AppState = AppKilled // 回复状态设置为节点被终止
    reply.Success = false
    DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Node is killed,
returning AppKilled\n", rf.me)
    return
}

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

// 如果请求的任期小于当前任期, 说明这个请求过期了
if args.Term < rf.currentTerm {
    reply.Term = rf.currentTerm
    reply.Success = false
    reply.AppState = AppExpire // 回复状态设置为过时
    DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Current Term: %d,
Requested Term: %d, returning AppOutOfDate\n", rf.me, rf.currentTerm, args.Term)
    return
}

rf.currentTerm = args.Term
rf.votedFor = args.LeaderId
rf.status = Follower
rf.timer.Reset(rf.overtime)

// 处理日志不匹配的情况
if args.PrevLogIndex >= len(rf.logs) || args.PrevLogTerm !=
rf.logs[args.PrevLogIndex].Term {
    reply.Term = rf.currentTerm
    reply.Success = false
    reply.AppState = AppMismatch
    DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Mismatch, returning
Mismatch\n", rf.me)
    return
}

if rf.lastApplied > args.PrevLogIndex {
    reply.Term = rf.currentTerm
    reply.Success = false
    reply.AppState = AppCommitted

```

```

DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - AppCommitted,
returning AppCommitted\n", rf.me)
    return
}

// 处理日志
if args.Entries != nil {
    rf.logs = rf.logs[:args.PrevLogIndex+1]
    rf.logs = append(rf.logs, args.Entries...)
}
for rf.lastApplied < args.LeaderCommit {
    rf.lastApplied++
    applyMsg := ApplyMsg{
        CommandValid: true,
        CommandIndex: rf.lastApplied,
        Command:      rf.logs[rf.lastApplied].Command,
    }
    rf.applyChan <- applyMsg
    rf.commitIndex = rf.lastApplied
}

reply.Term = rf.currentTerm
reply.Success = true
reply.AppState = AppNormal
DPrintf("[INFO][AppendEntries][同步 完成]Node %d: AppendEntries - Success, returning
AppNormal\n", rf.me)
    return
}

```

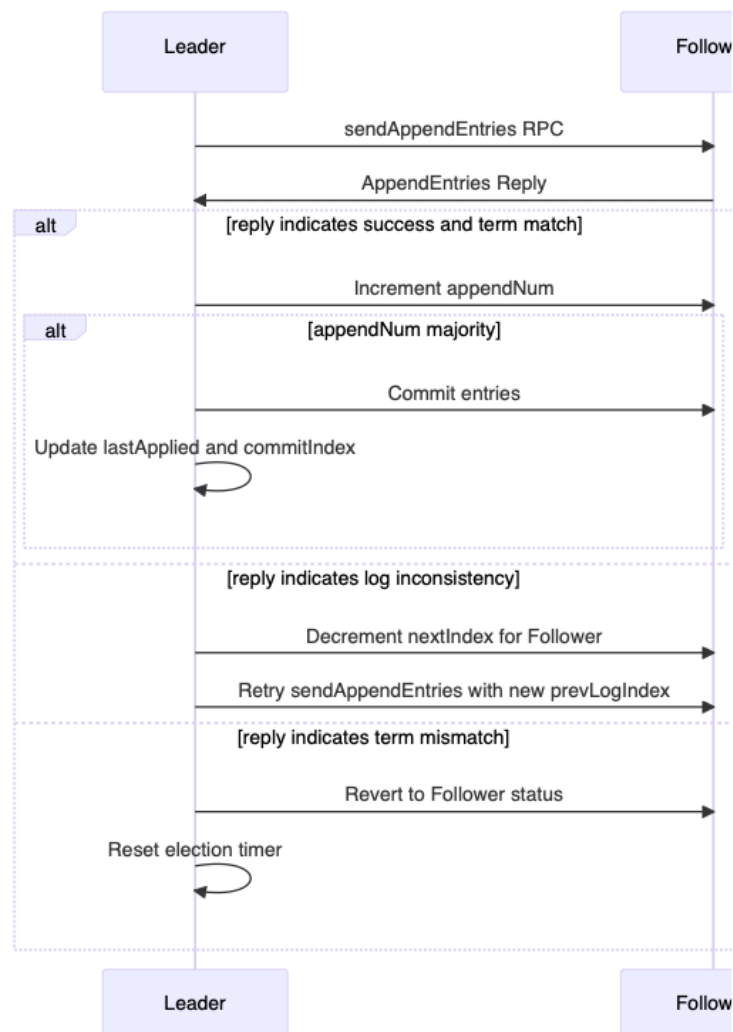
sendAppendEntries修改

`sendAppendEntries` 函数的2B版本中，主要针对不同的RPC响应类型进行了详细的处理，以下是这些情况的详细说明

- **AppNormal**：首先检查响应是否成功，并且响应中的任期与当前Leader的任期相同。如果成功并且任期匹配，则增加 `appendNum` 计数器，该计数器跟踪已成功接收心跳/日志的节点数量。如果 `appendNum` 超过集群的一半，则对日志条目进行提交，并更新 `lastApplied` 和 `commitIndex`。更新 `nextIndex[server]` 为 `args.LogIndex + 1`，表示下一次向该节点发送的日志起始位置。
- **AppMismatch**：首先检查响应的任期与当前Leader的任期是否一致。如果一致，则递减该节点的 `nextIndex[server]`，这意味着下次向该节点发送时，将包括更早的日志条目。重新构造 `AppendEntriesArgs`，设置新的 `PrevLogIndex` 和 `PrevLogTerm`，以及需要发送的日志条目。递归调用 `sendAppendEntries`，尝试重新同步日志。
- **AppCommitted**：同样先进行任期检查。如果响应的任期大于当前Leader的任期，将当前节点转换为 Follower，并重置选举超时。如果任期一致，则递增 `nextIndex[server]`，表示下一次向该节点发送时，将跳过已经提交的日志条目。类似于 `AppMismatch` 情况，重新构造 `AppendEntriesArgs` 并递归调用 `sendAppendEntries`。

- **AppExpire**：此时说明当前Leader的任期比响应节点的任期小。这时，当前节点需要将自己的状态更新为Follower，并重置选举超时。如果响应中的任期大于当前Leader的任期，还需要更新当前节点的任期为响应中的任期，并重置已投票的候选人ID为-1。这种情况通常表明存在更高任期的Leader，或者是网络分区导致的信息延迟。
- **AppRepeat**：此时可能是因为之前的响应丢失或者延迟。类似于 **AppExpire**，如果响应的任期大于当前Leader的任期，节点需要转换为Follower状态，更新任期，重置投票信息，并重置选举超时。这种情况下，当前Leader需要考虑集群中可能存在的任期更高的Leader或者网络问题。
- **AppKilled**：此时意味着目标节点已经停止运行。这种情况下，不进行任何状态更新，只简单地返回 false，因为没有必要继续与已终止的节点通信。这主要用于处理测试环境中节点被人为停止的情况。

修改后的逻辑可以用下图来表示：



修改的关键代码片段如下所示：

```

switch reply.AppState {
case AppNormal: // 复制正常
    DPrintf("[INFO][sendAppendEntries][成功]Node %d: Node %d returned normally, returning true\n", rf.me, server)
    if reply.Success && reply.Term == rf.currentTerm && *appendNum <= len(rf.peers)/2 {
        *appendNum++
    }
    if rf.nextIndex[server] >= args.LogIndex+1 {

```

```

        return ok
    }
    rf.nextIndex[server] = args.LogIndex + 1
    if *appendNum > len(rf.peers)/2 {
        *appendNum = 0
        if rf.logs[args.LogIndex].Term != rf.currentTerm {
            return false
        }
        for rf.lastApplied < args.LogIndex {
            rf.lastApplied++
            applyMsg := ApplyMsg{
                CommandValid: true,
                Command:      rf.logs[rf.lastApplied].Command,
                CommandIndex: rf.lastApplied,
            }
            rf.applyChan <- applyMsg
            rf.commitIndex = rf.lastApplied
        }
    }
    case AppExpire: // 复制过时
        DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned AppOutOfDate,
returning false\n", rf.me, server)
        rf.status = Follower
        rf.timer.Reset(rf.overtime)
        if reply.Term > rf.currentTerm {
            rf.currentTerm = reply.Term
            rf.votedFor = -1
        }
    case AppMismatch: // 复制不匹配
        DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned Mismatch, returning
false\n", rf.me, server)
        if args.Term != rf.currentTerm {
            return false
        }
    }
    rf.nextIndex[server]--
    argsNewa := &AppendEntriesArgs{
        Term:      args.Term,
        LeaderId:   rf.me,
        PrevLogIndex: 0,
        PrevLogTerm: 0,
        Entries:    nil,
        LeaderCommit: args.LeaderCommit,
        LogIndex:   args.LogIndex,
    }
    for rf.nextIndex[server] > 0 {
        argsNewa.PrevLogIndex = rf.nextIndex[server] - 1
        if argsNewa.PrevLogIndex >= len(rf.logs) {
            rf.nextIndex[server]--
            continue
        }
        argsNewa.PrevLogTerm = rf.logs[argsNewa.PrevLogIndex].Term
        break
    }

```

```

    }
    if rf.nextIndex[server] < args.LogIndex+1 {
        argsNewa.Entries = rf.logs[rf.nextIndex[server] : args.LogIndex+1]
    }
    reply := new(AppendEntriesReply)
    go rf.sendAppendEntries(server, argsNewa, reply, appendNum)
case AppRepeat: // 复制重复
    DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned AppRepeat, returning false\n", rf.me, server)
    if reply.Term > rf.currentTerm {
        rf.status = Follower
        rf.currentTerm = reply.Term
        rf.votedFor = -1
        rf.timer.Reset(rf.overtime)
    }
case AppCommitted: // 复制的日志已经提交
    DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned AppCommitted, returning false\n", rf.me, server)
    if args.Term != rf.currentTerm {
        return false
    }
    rf.nextIndex[server]++
    if reply.Term > rf.currentTerm {
        rf.status = Follower
        rf.currentTerm = reply.Term
        rf.votedFor = -1
        rf.timer.Reset(rf.overtime)
        return false
    }
}

argsNewa := &AppendEntriesArgs{
    Term:      args.Term,
    LeaderId:   rf.me,
    PrevLogIndex: 0,
    PrevLogTerm: 0,
    Entries:    nil,
    LeaderCommit: args.LeaderCommit,
    LogIndex:   args.LogIndex,
}

for rf.nextIndex[server] > 0 {
    argsNewa.PrevLogIndex = rf.nextIndex[server] - 1
    if argsNewa.PrevLogIndex >= len(rf.logs) {
        rf.nextIndex[server]--
        continue
    }
    argsNewa.PrevLogTerm = rf.logs[argsNewa.PrevLogIndex].Term
    break
}

if rf.nextIndex[server] < args.LogIndex+1 {
    argsNewa.Entries = rf.logs[rf.nextIndex[server] : args.LogIndex+1]
}

reply := new(AppendEntriesReply)

```

```
go rf.sendAppendEntries(server, argsNewa, reply, appendNum)
case AppKilled: // Raft程序终止
    DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d is killed\n", rf.me, server)
    return false
```

2B测试情况

Debug

2B测试时遇到了很多bug，典型的问题有下面两个。

1. 数组越界

```
Test (2B): basic agreement ...
panic: runtime error: index out of range [-1]

goroutine 31 [running]:
6.824/raft.(*Raft).AppendEntries(0x140000cc240, 0x140000cedc0, 0x140000f27e0)
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/raft/raft.go:457 +0x3d4
reflect.Value.call({0x140000ce500?, 0x140000925d8?, 0x14000193af8?}, {0x1030951ec, 0x4}, {0x14000193c60, 0x3, 0x1030857d0?})
/usr/local/go/src/reflect/value.go:596 +0x994
reflect.Value.Call({0x140000ce500?, 0x140000925d8?, 0x140000929e8?}, {0x14000193c60?, 0x0?, 0x0?})
/usr/local/go/src/reflect/value.go:380 +0x94
6.824/labrpc.(*Service).dispatch(0x14000097980, {0x1030987a4, 0xd}, {{0x1030ff060, 0x140000ab360}, {0x10309879f, 0x12}, {0x10314c718, 0x1030f7b80}, {0x1400019c000, ...}, ...})
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/labrpc/labrpc.go:496 +0x284
6.824/labrpc.(*Server).dispatch(0x140000a2738, {{0x1030ff060, 0x140000ab360}, {0x10309879f, 0x12}, {0x10314c718, 0x1030f7b80}, {0x1400019c000, 0xe7, 0x120}, ...})
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/labrpc/labrpc.go:420 +0x21c
6.824/labrpc.(*Network).processReq.func1()
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/labrpc/labrpc.go:240 +0x54
created by 6.824/labrpc.(*Network).processReq in goroutine 30
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/labrpc/labrpc.go:239 +0x1a8
panic: runtime error: index out of range [-1]
```

我在实现 `AppendEntries` 时，遇到了一个典型的运行时错误：`panic: runtime error: index out of range`。这个错误发生在尝试访问 Raft 日志数组时，具体表现为尝试获取一个不存在的数组元素，导致程序崩溃。问题的解决方法是将 `rf.logs[args.PrevLogIndex-1].Term` 修改为 `rf.logs[args.PrevLogIndex].Term`。

2. 无法成功选主

```
Ⓢ (base) ✖ LYU@Matts-MacBook-Pro ~/Downloads/学习资料/分布式实验平时作业/6.824/src/raft  master ➤ go test -run 2B
Test (2B): basic agreement ...
... Passed -- 0.9 3 14 4008 3
Test (2B): RPC byte count ...
... Passed -- 2.9 3 46 113940 11
Test (2B): agreement after follower reconnects ...
... Passed -- 6.3 3 134 34961 8
Test (2B): no agreement if too many followers disconnect ...
... Passed -- 6.0 5 372 77867 5
Test (2B): concurrent Start()s ...
... Passed -- 0.8 3 14 3772 6
Test (2B): rejoin of partitioned leader ...
--- FAIL: TestRejoin2B (12.05s)
    config.go:609: one(104) failed to reach agreement
Test (2B): leader backs up quickly over incorrect follower logs ...
... Passed -- 47.7 5 14978 11285778 108
Test (2B): RPC counts aren't too high ...
... Passed -- 2.3 3 36 10780 12
FAIL
exit status 1
FAIL 6.824/raft 81.944s
```

这个问题一度困扰了我很久，我打印了输出日志。日志显示，在测试中节点频繁发起选举（表现为 Term 的不断增长），但没有任何一个节点能稳定地成为领导者。这通常是由于节点在选举过程中没有获得足够的选票。根据 Raft 协议，一个节点在成为领导者之前，必须证明它拥有的日志信息至少和大多数节点一样新。

```

1129 2023/12/07 12:00:17 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:43 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1130 2023/12/07 12:00:17 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:44 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1131 2023/12/07 12:00:17 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:45 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1132 2023/12/07 12:00:18 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:46 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1133 2023/12/07 12:00:18 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:47 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1134 2023/12/07 12:00:18 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:48 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1135 2023/12/07 12:00:18 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:49 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1136 2023/12/07 12:00:18 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:50 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1137 2023/12/07 12:00:18 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:51 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1138 2023/12/07 12:00:18 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:52 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1139 2023/12/07 12:00:18 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:53 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1140 2023/12/07 12:00:19 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:54 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1141 2023/12/07 12:00:19 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:55 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1142 2023/12/07 12:00:19 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:56 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1143 2023/12/07 12:00:19 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:57 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1144 2023/12/07 12:00:19 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:58 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1145 2023/12/07 12:00:19 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:59 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1146 2023/12/07 12:00:19 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:60 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1147 2023/12/07 12:00:19 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:61 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1148 2023/12/07 12:00:20 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:62 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1149 2023/12/07 12:00:20 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:63 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1150 2023/12/07 12:00:20 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:64 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1151 2023/12/07 12:00:20 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:65 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1152 2023/12/07 12:00:20 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:66 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1153 2023/12/07 12:00:20 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:67 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1154 2023/12/07 12:00:20 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:68 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1155 2023/12/07 12:00:20 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:69 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1156 2023/12/07 12:00:21 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:70 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1157 2023/12/07 12:00:21 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:71 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1158 2023/12/07 12:00:21 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:72 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1159 2023/12/07 12:00:21 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:73 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1160 2023/12/07 12:00:21 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:74 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1161 2023/12/07 12:00:21 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:75 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1162 2023/12/07 12:00:21 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:76 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1163 2023/12/07 12:00:21 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:77 CandidateId:2 LastLogIndex:1 LastLogTerm:0}
1164 2023/12/07 12:00:21 [INFO] [RequestVote]Node 2: RequestVote called with args &{Term:78 CandidateId:1 LastLogIndex:4 LastLogTerm:0}
1165 2023/12/07 12:00:21 [INFO] [RequestVote]Node 1: RequestVote called with args &{Term:79 CandidateId:2 LastLogIndex:1 LastLogTerm:0}

```

问题的根源在于 `RequestVote` 函数中的日志一致性检查逻辑。以下是原本的代码片段：

```

lastLogIndex := len(rf.logs) - 1
lastLogTerm := 0
if lastLogIndex >= 0 {
    lastLogTerm = rf.logs[lastLogIndex].Term
}

if args.LastLogIndex < lastLogIndex || args.LastLogTerm < lastLogTerm {
    reply.VoteState = Expire
    reply.VoteGranted = false
    reply.Term = rf.currentTerm
    return
}

```

在原始的实现中，我是直接比较 `args.LastLogIndex` 和本地日志的最后一个索引。但这种比较方式并不能完全满足 Raft 协议中对日志“新旧”的定义，因为它没有充分考虑日志条目的任期号。

为了解决这个问题，我重新审视了 Raft 论文中关于日志一致性的要求。与论文的Figure 8类似，我构造了一个例子：

假设有一个 Raft 集群，包含五个服务器节点：A, B, C, D, 和 E。在某一时刻，这些节点的日志和任期如下：

节点 A (候选者) : 日志 [1, 2, 3]
节点 B: 日志 [1, 2]
节点 C: 日志 [1, 2, 3, 4]
节点 D: 日志 [1, 2, 3]
节点 E (候选者) : 日志 [1, 2, 3, 4, 5]

现在, 节点 A 和节点 E 都想成为领导者,

按照 Raft 协议, 我们需要比较两个候选者的日志一致性, 以确定哪个候选者更有资格成为领导者: **节点 A** 的最后一条日志是在第 3 任期。**节点 E** 的最后一条日志是在第 5 任期。虽然节点 E 的日志条目数比节点 A 多, 但更重要的是, 节点 E 的日志在任期上更为最新。根据 Raft 协议, 节点 E 将被视为更有资格成为领导者, 因为它的最后日志条目的任期号更高。

因此, 在这次选举中, 节点 B、C 和 D 很可能会给节点 E 投票, 因为节点 E 的日志比它们自己的都要新。尽管节点 A 的日志长度与节点 D 相同, 但由于其最后日志条目的任期号小于节点 E, 它不太可能赢得足够多的选票。

所以我改进了 `RequestVote` 函数中的日志比较逻辑, 使其首先检查候选节点的最后日志条目的任期号是否小于当前节点的相应任期号。如果任期号相同, 那么拥有更多日志条目的候选者将被认为更有资格成为领导者。具体修改如下:

```
lastLogIndex := len(rf.logs)-1
lastLogTerm := 0
if len(rf.logs) > 0 {
    lastLogTerm = rf.logs[lastLogIndex].Term
}

if args.LastLogTerm < lastLogTerm {
    rf.currentTerm = args.Term
    reply.Term = args.Term
    reply.VoteGranted = false
    reply.VoteState = LogExpire
    return
}

if args.LastLogTerm == rf.logs[len(rf.logs)-1].Term &&
args.LastLogIndex < len(rf.logs)-1 {
    rf.currentTerm = args.Term
    reply.Term = args.Term
    reply.VoteGranted = false
    reply.VoteState = LogExpire
    DPrintf("[INFO][RequestVote]Node %d: RequestVote - Vote Not Granted\n", rf.me)
    return
}
```

改进后的逻辑更严谨地实现了Raft的选举约束, 即优先考虑日志任期号, 其次是日志索引。这样做的目的是确保选出的Leader拥有所有已提交日志条目的最新副本。此处修改成功解决了无法成功选主的问题。

测试结果

全部通过:


```
(base) LYU@Matts-MacBook-Pro ~/Downloads/学习资料/分布式实验平时作业/6.824/src/raft master ± go test -run 2B
2023/12/08 10:16:00 Test2B Weichen Lyu
Test (2B): basic agreement ...
... Passed -- 0.9 3 14 3976 3
Test (2B): RPC byte count ...
... Passed -- 2.8 3 46 113844 11
Test (2B): agreement after follower reconnects ...
... Passed -- 6.2 3 138 35349 8
Test (2B): no agreement if too many followers disconnect ...
... Passed -- 3.6 5 252 46032 4
Test (2B): concurrent Start()s ...
... Passed -- 0.8 3 12 3436 6
Test (2B): rejoin of partitioned leader ...
... Passed -- 6.3 3 197 50620 4
Test (2B): leader backs up quickly over incorrect follower logs ...
... Passed -- 43.7 5 13252 9228153 106
Test (2B): RPC counts aren't too high ...
... Passed -- 2.3 3 34 10112 12
PASS
ok      6.824/raft      67.063s
```

执行下面这行命令可以将日志保存下来

```
go test -run 2B > test2B.log
```

Lab2C

2C问题构思与回溯

Lab2C的主要要求是实现Raft算法中的数据持久化，确保在服务器重启后，Raft节点能够从中断的地方恢复服务。以下是对完成这些要求的分析：

1. 持久化关键状态：

根据论文的Figure 2，需要持久化的状态包括 `currentTerm`（当前任期号）、`votedFor`（在当前获得选票的候选人ID），以及 `logs`（日志条目数组）。这些状态的持久化对于节点重启后能够正确恢复其在Raft集群中的状态至关重要。

2. 实现 `persist` 和 `readPersist` 函数：

完善 `raft.go` 中的 `persist()` 函数以在每次persistent state更改时保存状态，以及 `readPersist()` 函数以在服务器启动时从存储中恢复状态。使用 `labgob` 编码器（类似于Go语言的 `gob` 库）对persistent state进行序列化和反序列化。`labgob` 是为了在序列化过程中避免对非导出字段进行操作。

3. 使用 `Persister` 对象：

`Persister` 对象用于保存和恢复persistent state。在调用 `Raft.make()` 时，会提供一个包含Raft最近persistent state的 `Persister` 对象。

Raft节点需要在启动时使用 `ReadRaftState()` 从 `Persister` 初始化其状态，并在每次persistent state更改后使用 `SaveRaftState()` 保存状态。

persist方法

这个函数负责将Raft的持久化状态保存到稳定存储中，以便在崩溃和重启后可以恢复这些状态。实现步骤如下：

1. 创建一个 `bytes.Buffer` 实例 `w`，作为编码后数据的缓冲区。
2. 使用 `labgob.NewEncoder` 创建一个新的编码器 `e`，绑定到缓冲区 `w`。

- 依次调用 `e.Encode` 方法来序列化Raft的持久状态：`currentTerm`（当前任期）、`votedFor`（在当前任期内收到选票的候选人ID）、`logs`（日志条目数组）。
- 将编码后的数据从缓冲区 `w` 中提取出来，形成一个字节切片 `data`。
- 调用 `rf.persister.SaveRaftState(data)` 将 `data` 保存到稳定存储中。

```
func (rf *Raft) persist() {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.votedFor)
    e.Encode(rf.logs)
    data := w.Bytes()
    rf.persister.SaveRaftState(data)
}
```

readPersist方法

这个函数用于在Raft节点启动时从持久化存储中恢复之前保存的状态。实现步骤如下：

- 首先检查传入的数据 `data` 是否为空或长度为零，这种情况表明没有需要恢复的持久状态。
- 创建一个新的 `bytes.Buffer` 实例 `r`，并将 `data` 写入其中，用于解码。
- 使用 `labgob.NewDecoder` 创建一个新的解码器 `d`，绑定到缓冲区 `r`。
- 分别使用 `d.Decode` 方法尝试解码存储的 `currentTerm`、`votedFor` 和 `logs` 到临时变量 `Term`、`VoteFor` 和 `logs` 中。
- 如果解码过程中没有出现错误，则将这些临时变量的值赋给Raft实例的相应字段。

```
func (rf *Raft) readPersist(data []byte) {
    if data == nil || len(data) < 1 { // bootstrap without any state?
        return
    }
    r := bytes.NewBuffer(data)
    d := labgob.NewDecoder(r)
    var Term int
    var VoteFor int
    var logs []LogEntry
    if d.Decode(&Term) != nil ||
        d.Decode(&VoteFor) != nil ||
        d.Decode(&logs) != nil {
        fmt.Println("decode error")
    } else {
        rf.currentTerm = Term
        rf.votedFor = VoteFor
        rf.logs = logs
    }
}
```

持久化状态

在raft代码中，有以下地方需要调用 `rf.persist()` 进行持久化：

1. 在 `RequestVote` 中：每当Raft服务器更新其 `currentTerm` 或 `votedFor` 时（例如，收到更高任期的投票请求或更新投票状态），它调用 `rf.persist()` 以确保这些更改被持久化。这是为了确保在服务器崩溃和重启后，其任期和投票状态能够从持久化存储中恢复。
2. 在 `sendRequestVote` 中：当服务器在处理对其他服务器的投票请求时发现对方任期较高，它将更新自己的任期并调用 `rf.persist()`，以确保这一更改得到持久化。
3. 在 `AppendEntries` 中：类似于处理投票请求，当服务器在处理心跳或日志复制请求时更新任期、日志或其他状态时，它会调用 `rf.persist()` 来持久化这些更改。
4. 在 `Start` 中：当一个新的命令被添加到日志中时，服务器需要调用 `rf.persist()` 来持久化更新后的日志状态。这是为了确保即使在领导者崩溃后，新的领导者也能够从持久化存储中恢复最新的日志状态。
5. 在 `ticker` 中：当服务器作为候选人开始新一轮的选举并增加其任期时，它需要调用 `rf.persist()` 来持久化新的任期。这样做是为了确保在选举过程中发生崩溃后，任期信息不会丢失。
6. 在 `sendAppendEntries` 中：当服务器在发送日志复制请求后发现对方任期更高，它将转变为跟随者并更新自己的任期，然后调用 `rf.persist()` 来持久化这一更改。
7. 在 `Make` 中：在创建Raft服务器时，如果持久化存储中有先前的状态，服务器会调用 `rf.readPersist()` 从中恢复状态。

2C测试情况

Debug

测试时遇到了一个比较棘手的bug。

```
Test (2C): basic persistence ...
... Passed -- 4.0 3 66 17644 6
Test (2C): more persistence ...
... Passed -- 16.9 5 1157 237216 16
Test (2C): partitioned leader and one follower crash, leader restarts ...
... Passed -- 2.0 3 31 8147 4
Test (2C): Figure 8 ...
... Passed -- 30.2 5 931 181415 25
Test (2C): unreliable agreement ...
... Passed -- 6.4 5 237 84045 246
Test (2C): Figure 8 (unreliable) ...
panic: runtime error: slice bounds out of range [-1:]

goroutine 46304 [running]:
6.824/raft.(*Raft).sendAppendEntries(0x14000299500, 0x3, 0x14002084ff0, 0x1400144ae20, 0x14001715e98)
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/raft/raft.go:610 +0x72c
created by 6.824/raft.(*Raft).sendAppendEntries in goroutine 45931
/Users/LYU/Downloads/学习资料/分布式实验平时作业/6.824/src/raft/raft.go:613 +0x71c
exit status 2
FAIL 6.824/raft 101.486s
```

错误消息显示为“panic: runtime error: slice bounds out of range [-1:]”，这个错误发生在 `sendAppendEntries` 函数中，指示在处理数组或切片时索引越界。

通过打印日志，我发现问题出在处理 `sendAppendEntries` 函数中的 `Mismatch` 和 `AppCommitted` 两种情况时，出现了“out of slice”错误。这个问题在服务器集群中表现为在处理节点之间的日志不一致时，没有正确地更新各个节点的 `nextIndex`，这导致了错误的日志重复追加。

为了解决这个问题，我决定重构相关代码。我的重构策略主要围绕两个方面：改进 `AppendEntriesReply` 结构以更好地处理日志不一致的情况，以及优化 `sendAppendEntries` 函数中的逻辑。

1. 我在 `AppendEntriesReply` 结构体中增加了一个新字段 `TmpNextIndex`。这个字段用于在接收到来自其他节点的响应时，提供一个临时的 `nextIndex` 值。这样做的目的是为了在处理日志不一致或已提交的情况时，能够更精确地更新每个节点的 `nextIndex`。

2. 在 `AppendEntries` 函数中，我添加了逻辑来适当设置 `TmpNextIndex` 的值。当出现日志不匹配的情况时，我将 `TmpNextIndex` 设置为 `lastApplied + 1`，以确保在下次尝试追加日志时，可以从正确的位置开始。
3. 优化 `sendAppendEntries` 的处理逻辑，在处理 `Mismatch` 和 `AppCommitted` 的响应时，我使用了从 `TmpNextIndex` 获取的值来更新 `nextIndex`。这意味着，当接收到这些特定类型的响应时，我可以根据回复中的 `TmpNextIndex` 直接更新 `nextIndex`，而无需进行额外的计算或尝试。

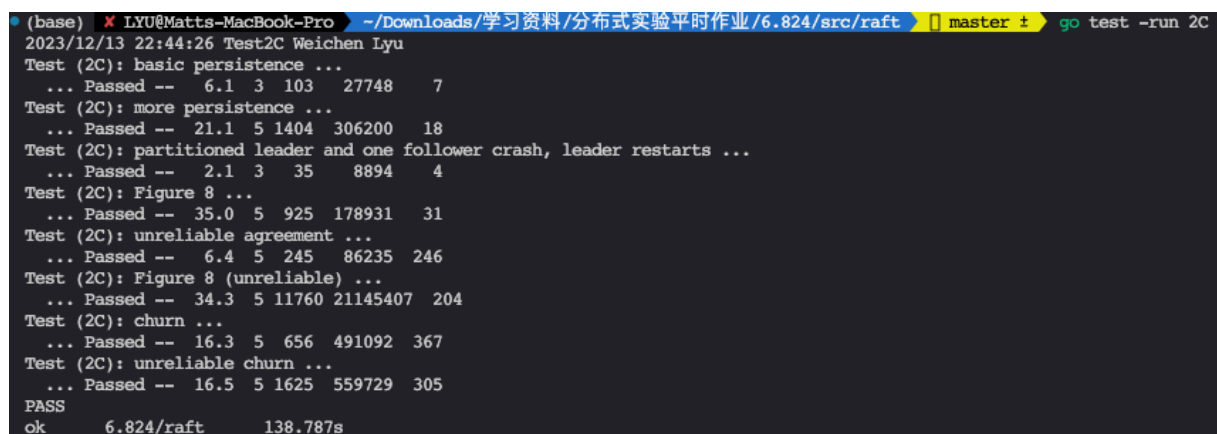
对于处理 `Mismatch` 和 `AppCommitted` 的具体修改如下：

```
case Mismatch: // 复制不匹配
    DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned Mismatch,
    returning false\n", rf.me, server)
    if args.Term != rf.currentTerm {
        rf.mu.Unlock()
        return false
    }
    rf.nextIndex[server] = reply.TmpNextIndex
case AppCommitted: // 复制的日志已经提交
    DPrintf("[INFO][sendAppendEntries][失败]Node %d: Node %d returned
    AppCommitted, returning false\n", rf.me, server)
    if args.Term != rf.currentTerm {
        rf.mu.Unlock()
        return false
    }
    rf.nextIndex[server] = reply.TmpNextIndex
```

通过这样的重构成功解决了该bug。

测试情况

测试样例全部通过：



```
(base) LYU@Matts-MacBook-Pro ~/Downloads/学习资料/分布式实验平时作业/6.824/src/raft [master] go test -run 2C
2023/12/13 22:44:26 Test2C Weichen Lyu
Test (2C): basic persistence ...
... Passed -- 6.1 3 103 27748 7
Test (2C): more persistence ...
... Passed -- 21.1 5 1404 306200 18
Test (2C): partitioned leader and one follower crash, leader restarts ...
... Passed -- 2.1 3 35 8894 4
Test (2C): Figure 8 ...
... Passed -- 35.0 5 925 178931 31
Test (2C): unreliable agreement ...
... Passed -- 6.4 5 245 86235 246
Test (2C): Figure 8 (unreliable) ...
... Passed -- 34.3 5 11760 21145407 204
Test (2C): churn ...
... Passed -- 16.3 5 656 491092 367
Test (2C): unreliable churn ...
... Passed -- 16.5 5 1625 559729 305
PASS
ok      6.824/raft    138.787s
```

执行下面这行命令可以将日志保存下来

```
go test -run 2C > test2C.log
```