Raft_3B

写在前面

在3A阶段，实现了领导人的选举，和简单的心跳机制，Lab 3B 的核心任务是实现 Raft 协议中的日志复制机制，确保集群中所有节点的状态机能够以相同的顺序执行相同的命令，从而保持一致性。

关键点

1. **日志条目结构**：每个日志条目包含命令、任期和索引信息。
2. **日志一致性检查**：Follower 需要验证 Leader 发送的日志是否与自己的日志匹配，如果不匹配则拒绝接受。
3. **冲突恢复优化**：当 Follower 拒绝日志条目时，通过提供额外的冲突信息（ConflictIndex 和 ConflictTerm）帮助 Leader 更快地调整 nextIndex。
4. **提交规则**：Leader 只能提交当前任期的日志条目，并且只有在该条目被大多数节点复制后才能提交。
5. **应用程序协程**：单独的协程负责将已提交的日志应用到状态机，确保应用顺序和持久性。
6. **并发控制**：使用互斥锁保护共享状态，使用条件变量协调不同协程的操作。

详细实现步骤

1. Raft初始化函数 (Make函数)

```
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{
        mu:        sync.RWMutex{},
        peers:     peers,
        persister: persister,
        me:        me,
        dead:      0,

        currentTerm: 0,
        votedFor:    -1,
        logs:        make([]LogEntry, 1), //索引0处的虚拟条目

        commitIndex: 0,
        lastApplied: 0,

        nextIndex:  make([]int, len(peers)),
        matchIndex: make([]int, len(peers)),

        state: Follower,
        //心跳超时时间<<小于选举超时(follower需要成为candidate所需要等待的时间)<<平均故障时间
        electionTimer:  time.NewTimer(RandomElectionTimeout()),
        heartbeatTimer: time.NewTimer(StableHeartbeatTimeout()),

        applych:        applyCh,
        replicatorCond: make([]*sync.Cond, len(peers)),
    }
```

```
    // SugarLogger.Infof("%v raft Server Initialization", me)
    // Your initialization code here (3A, 3B, 3C).

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

    // should use mu to protect applyCond, avoid other goroutine to change the
critical section
    rf.applyCond = sync.NewCond(&rf.mu)

    // initialize nextIndex and matchIndex, and start replicator goroutine
    // 应该使用mu来保护应用程序，避免其他例程更改临界区
    for peer := range peers {
        rf.matchIndex[peer], rf.nextIndex[peer] = 0, rf.getLastlog().Index+1
        if peer != rf.me {
            rf.replicatorCond[peer] = sync.NewCond(&sync.Mutex{})
            // 启动复制程序例程以向对等节点发送日志项
            go rf.replicator(peer)
        }
    }

    SugarLogger.Infof("%v raft Server Initialization replicatorCond Finish", me)
    // start ticker goroutine to start elections
    go rf.ticker()

    SugarLogger.Infof("%v raft Server Initialization select and heartbeat Finish
", me)
    // start apply goroutine to apply log entries to state machine
    go rf.applier()
    SugarLogger.Infof("%v raft Server Initialization Log Applier Finish ", me)

    return rf
}
```

2. 实现客户端命令接口 (Start 函数)

`Start` 函数是应用层与 Raft 通信的主要接口：

```
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if rf.state != Leader {
        return -1, -1, false
    }
    //每个raft服务器的第一条日志是对于他自己的
    newLogIndex := rf.getLastlog().Index + 1
    rf.logs = append(rf.logs, LogEntry{
        Term:    rf.currentTerm,
        Index:   newLogIndex,
        Command: command,
    })
    rf.matchIndex[rf.me], rf.nextIndex[rf.me] = newLogIndex, newLogIndex+1
    DPrintf("{Node %v} starts agreement on a new log entry with command %v in
term %v", rf.me, command, rf.currentTerm)

    // 为所有服务器广播日志添加条目
    rf.BroadcastHeartbeat(false)
```

```
    // Your code here (3B).

    return newLogIndex, rf.currentTerm, true
}
```

3. 实现 Leader 对每个 Follower 的日志复制处理

每个 Follower 需要一个专用的日志复制器：

```
// replicator 日志复制
func (rf *Raft) replicator(peer int) {
    rf.replicatorCond[peer].L.Lock()
    defer rf.replicatorCond[peer].L.Unlock()
    for !rf.killed() {
        for !rf.needReplicating(peer) {
            // send log entries to peer
            SugarLogger.Infof("%v raft wait ", peer)
            rf.replicatorCond[peer].Wait()
        }
        // send log entries to peer
        SugarLogger.Infof("%v raft send Log entries ", rf.me)
        rf.replicateOnceRound(peer)
    }
}
```

```
// needReplicating 判断peer是否需要日志条目复制
func (rf *Raft) needReplicating(peer int) bool {
    rf.mu.RLock()
    defer rf.mu.RUnlock()
    return rf.state == Leader && rf.matchIndex[peer] < rf.getLastlog().Index
}

// replicateOnceRound  新的心跳机制，负责日志复制，和日志回退重发
func (rf *Raft) replicateOnceRound(peer int) {
    rf.mu.RLock()
    if rf.state != Leader {
        rf.mu.RUnlock()
        return
    }
    prevLogIndex := rf.nextIndex[peer] - 1
    args := rf.genAppendEntriesArgs(prevLogIndex)
    rf.mu.RUnlock()
    reply := new(AppendEntriesReply)
    if rf.sendAppendEntries(peer, args, reply) {
        rf.mu.Lock()
        //如果rpc 后 还是leader 且周期没变化
        if args.Term == rf.currentTerm && rf.state == Leader {
            if !reply.Success { //日志一致性检查失败
                if reply.Term > rf.currentTerm { //脑裂，或者已经宕机的leader突然又活过来  可能发生了网络分区，或者这个 Leader 是一个刚刚恢复但任期落后的旧 Leader
                    //标签当前服务器 已经过时了，重新变成follower
                    rf.ChangeState(Follower)
                    rf.currentTerm, rf.votedFor = reply.Term, -1
                } else if reply.Term == rf.currentTerm { //说明follow的周期和leader周期一致，说明是条目出了问题
```

```
                    //减少nextIndex并重试
                    rf.nextIndex[peer] = reply.ConfictIndex
                    // TODO: optimize the nextIndex finding, maybe use binary
search
                    if reply.ConfictTerm != -1 {
                        firstLogIndex := rf.getFirstlog().Index
                        for index := args.PrevLogIndex - 1; index >=
firstLogIndex; index-- {
                            if rf.logs[index-firstLogIndex].Term ==
reply.ConfictTerm {
                                rf.nextIndex[peer] = index
                                break
                            }
                        }
                    }
                }
            } else { //日志匹配成功 提交
                rf.matchIndex[peer] = args.PrevLogIndex + len(args.Entries)
                rf.nextIndex[peer] = rf.matchIndex[peer] + 1
                //advance commitIndex if possible
                rf.advanceCommitIndexForLeader()
            }
        }
        rf.mu.Unlock()
    }
}
```

4. 更新选举和心跳机制

```
// ticker 选举超时与心跳超时逻辑
func (rf *Raft) ticker() {
    for !rf.killed() {
        select {
        case <-rf.electionTimer.C:
            //选举超时逻辑
            rf.mu.Lock()
            rf.ChangeState(Candidate)
            rf.currentTerm += 1
            // rf.persist()
            // start election
            rf.StartElection()
            rf.electionTimer.Reset(RandomElectionTimeout()) //在分裂投票的情况下，重
置选举计时器
            rf.mu.Unlock()
        case <-rf.heartbeatTimer.C:
            //心跳超时逻辑
            rf.mu.Lock()
            if rf.state == Leader {
                //重新发送心跳
                DPrintf("{Node %v} BroadcastHeartbeat", rf.me)
                rf.BroadcastHeartbeat(true)

                rf.heartbeatTimer.Reset(StableHeartbeatTimeout())
            }
            rf.mu.Unlock()
        }
```

```
        // Your code here (3A)
        // Check if a leader election should be started.

        // pause for a random amount of time between 50 and 350
        // milliseconds.
        // ms := 50 + (rand.Int63() % 300)
        // time.Sleep(time.Duration(ms) * time.Millisecond)
    }
}
```

广播心跳机制

```
// BroadcastHeartbeat Leader发送心跳  新的心跳机制，负责日志复制，和日志回退重发
func (rf *Raft) BroadcastHeartbeat(isHeartbeat bool) {
    for peer := range rf.peers {
        if peer == rf.me {
            continue
        }
        if isHeartbeat {
            //应该立即将心跳发送给所有的对等体
            go rf.replicateOnceRound(peer)
        } else {
            //只需要向复制器发送信号，将日志条目发送给对等体
            rf.replicatorCond[peer].Signal()
        }
    }
}
```

5. 实现日志应用协程

一个单独的协程负责将已提交的日志应用到状态机：

```
// applier 应用日志条目
func (rf *Raft) applier() {
    for !rf.killed() {
        rf.mu.Lock()
        // 检查commitIndex是否可用
        if rf.commitIndex <= rf.lastApplied {
            //需要等待commitIndex被推进
            rf.applyCond.Wait()
        }
        // 应用日志条目到状态机
        firstLogIndex, commitIndex, lastApplied := rf.getFirstlog().Index,
rf.commitIndex, rf.lastApplied
        entries := make([]LogEntry, commitIndex-lastApplied)
        copy(entries, rf.logs[lastApplied-firstLogIndex+1:commitIndex-
firstLogIndex+1])
        rf.mu.Unlock()

        //将申请消息发送到applyCh以获取服务/状态机副本
        for _, entry := range entries {
            rf.applych <- ApplyMsg{
                CommandValid: true,
                Command:      entry.Command,
                CommandIndex: entry.Index,
```

```
            }
        }
        rf.mu.Lock()
        DPrintf("{Node %v} applies log entries from index %v to %v in term %v",
rf.me, lastApplied+1, commitIndex, rf.currentTerm)
        // 使用commitIndex而不是rf.commitIndex，因为rf.commitIndex可能在Unlock（）和
Lock（）期间发生变化。
        rf.lastApplied = commitIndex
        rf.mu.Unlock()
    }
}
```

6. 实现日志复制机制

主要通过 `AppendEntries` RPC 实现：

```
// AppendEntries 处理心跳日志
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply
*AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer DPrintf("{Node %v}'s state is {state %v, term %v}} after processing
AppendEntries,  AppendEntriesArgs %v and AppendEntriesReply %v ", rf.me,
rf.state, rf.currentTerm, args, reply)

    //如果当前周期大于 发送周期，说明发送周期已经落后。设置返回消息，立刻返回
    if args.Term < rf.currentTerm {
        reply.Term, reply.Success = rf.currentTerm, false
        return
    }

    //如果当前周期落后发送周期，说明当前Server未更新leader,首先设置该Raft的周期
    if args.Term > rf.currentTerm {
        rf.currentTerm, rf.votedFor = args.Term, -1
    }
    // 改为Follow
    rf.ChangeState(Follower)
    rf.electionTimer.Reset(RandomElectionTimeout())

    //如果在当前peer的 prevLogIndex中没有一个词条匹配prevLogTerm的条目，则返回false
    //args.PrevLogIndex "我认为你的日志在索引 PrevLogIndex 处的条目应该和我的一致"
    if args.PrevLogIndex < rf.getFirstlog().Index {
        reply.Term, reply.Success = rf.currentTerm, false
        return
    }
    //检查日志是否匹配，如果不匹配，返回冲突索引和词条
    //如果现有项与新项冲突(相同的索引，但不同周期）删除现有条目及其后面的所有条目
    if !rf.IsLogMatch(args.PrevLogIndex, args.PrevLogTerm) {
        reply.Term, reply.Success = rf.currentTerm, false
        lastLogIndex := rf.getLastlog().Index
        //找出冲突项的第一个索引
        if lastLogIndex < args.PrevLogIndex { //如果follower的最后一个索引比leader期
望的索引小，则最后一个的前一个位置为矛盾位置
            reply.ConfictIndex, reply.ConfictTerm = lastLogIndex+1, -1
        } else { //如果follower的最后一个索引比leader期望的索引大，且第一个索引比leader期
望的索引小，则说明能follow存在该索引，是该索引的周期不匹配
            firtstLogIndex := rf.getFirstlog().Index
```

```
            //找出冲突项的第一个索引
            index := args.PrevLogIndex
            //这里index需不需要减1
            for index >= firtstLogIndex && rf.logs[index-firtstLogIndex].Term ==
args.PrevLogTerm {
                index--
            }
            //告诉 Leader：“我有一段连续的、任期为 args.PrevLogTerm 的日志，从索引
ConflictIndex 开始”  也可以用这个实现
            // reply.ConfictIndex, reply.ConfictTerm = index+1, args.PrevLogTerm

            //更标准的 Raft 优化实现有时会返回 Follower 在 args.PrevLogIndex 处实际的任
期 rf.logs[args.PrevLogIndex - firtstLogIndex].Term 作为 ConfictTerm，
            //让 Leader 可以更快地在其自身日志中查找匹配的任期。
            reply.ConfictIndex, reply.ConfictTerm = index+1,
rf.logs[args.PrevLogIndex-firtstLogIndex].Term
        }
        return
    }
    //追加日志中没有的任何新条目
    firstLogIndex := rf.getFirstlog().Index
    for index, entry := range args.Entries {
        // 找到现有日志和附加日志的连接点

        // 对每个条目检查两种情况：
        // 超出范围：entry.Index-firstLogIndex >= len(rf.logs)
        // 任期冲突：rf.logs[entry.Index-firstLogIndex].Term != entry.Term
        // 当发现首个不匹配点，执行日志截断和追加，然后退出循环
        if entry.Index-firstLogIndex >= len(rf.logs) || rf.logs[entry.Index-
firstLogIndex].Term != entry.Term {
            rf.logs = append(rf.logs[:entry.Index-firstLogIndex],
args.Entries[index:]...)
            break
        }
    }

    //如果leaderCommit > commitIndex，设置commitIndex = min（leaderCommit，最后一个新
条目的索引）（论文）
    newCommitIndex := Min(args.LeaderCommit, rf.getLastlog().Index)
    if newCommitIndex > rf.commitIndex {
        DPrintf("{Node %v} advances commitIndex from %v to %v with leaderCommit
%v in term %v", rf.me, rf.commitIndex, newCommitIndex, args.LeaderCommit,
rf.currentTerm)
        rf.commitIndex = newCommitIndex
        //通知有新的已提交日志需要应用到状态机
        rf.applyCond.Signal()
    }

    reply.Term, reply.Success = rf.currentTerm, true
}
```

7. 实现日志提交和推进 commitIndex

Leader 在收到大多数 Follower 的成功回复后，需要推进 commitIndex：

```
// advanceCommitIndexForLeader 通过检查大多数节点的日志复制状态，安全地推进 Leader 的提交索引
func (rf *Raft) advanceCommitIndexForLeader() {
    n := len(rf.matchIndex)
    sortMatchIndex := make([]int, n)
    copy(sortMatchIndex, rf.matchIndex)
    sort.Ints(sortMatchIndex)
    //获取已知在大多数服务器上复制的索引最高的日志条目的索引
    newCommitIndex := sortMatchIndex[n-(n/2+1)]
    if newCommitIndex > rf.commitIndex {
        if rf.IsLogMatch(newCommitIndex, rf.currentTerm) { //只能提交本周期内的日志
            DPrintf("{Node %v} advances commitIndex from %v to %v in term %v", rf.me, rf.commitIndex, newCommitIndex, rf.currentTerm)
            rf.commitIndex = newCommitIndex
            rf.applyCond.Signal()
        }
    }
}
```

8. 投票机制没有多大变化

```
func (rf *Raft) StartElection() {
    //这里应该不加锁
    rf.votedFor = rf.me //为自己投票
    args := rf.genRequestVoteArgs()
    grantedVotes := 1
    DPrintf("{Node %v} starts election with RequestVoteArgs %v", rf.me, args)
    for peer := range rf.peers {
        // fmt.Println(peer)
        if peer == rf.me {
            continue
        }
        go func(peer int) {
            reply := new(RequestVoteReply) // 返回一个指向类型 T 的指针
            if rf.sendRequestVote(peer, args, reply) {
                rf.mu.Lock()
                defer rf.mu.Unlock()
                DPrintf("{Node %v} receives RequestVoteReply %v from {Node %v} after sending RequestVoteArgs %v", rf.me, reply, peer, args)
                if args.Term == rf.currentTerm && rf.state == Candidate { //这里再次检查是避免rpc在调用过程中超时导致，当前Term和state发生了变化
                    if reply.VoteGranted { //该peer 投票给了这个候选人
                        grantedVotes += 1
                        if grantedVotes > len(rf.peers)/2 {
                            DPrintf("{Node %v} receives over half of the votes", rf.me)
                            rf.ChangeState(Leader)
                            rf.BroadcastHeartbeat(true)
                        }
                    } else if reply.Term > rf.currentTerm { //没有投票给候选人，说明选人要么日志落后 要么周期落后
                        rf.ChangeState(Follower)
```

```
                        rf.currentTerm, rf.votedFor = reply.Term, -1
                    }
                }
            }
        }(peer)
    }
}
```

测试与结果

```
time go test --run 3B
```