Lab2B

Log replication：

Leader 接收到 client 请求到应用其包含 command 到状态机大概需要这么几个过程：

1. 收到 client 的请求，包含一个 command 参数（`Start(command interface{})`）。
2. Leader 把该请求追加到本地日志（`rf.log = append(rf.log, &logEntry{rf.currentTerm, command})`）。
3. 通过心跳并行通知所有 Follower 写入该日志（AppendEntries RPC）。
4. 待大多数 Follower 成功写入后，提交该日志到状态机。

心跳是定时的，而**同步日志**则是在定时的心跳的过程中完成的。如果 RPC 参数中不携带日志条目，则就是一个简单心跳；如果 RPC 参数中携带日志条目，则是 Leader 告诉该 Follower，我认为你需要同步这些日志。

通过试探得到匹配点，Leader 在匹配点之后的日志便是需要同步给 Follower 的部分。试探匹配点即是 Leader 会依照自己的日志条目，从后往前，不断询问，你有没有存这一条日志？只要发现某条日志 Follower 存了，那么它就是个匹配点，其之前的日志必然一样 。为了实现这个逻辑，raft 论文主要使用了这几个变量： matchIndex[]、nextIndex[] 和 prevLogIndex、prevLogTerm。

其实2B实验主要要修改的部分是两个：

1、leader向各个follower发送heartbeat。在2A中，这部分不需要包含实际的日志，但是在2B中，leader和follower之间的日志同步是需要靠不断的心跳进行实现的。

2、appendentries。也就是follower或者candidate接收心跳的部分。在这部分中，follower和candidate需要跟踪leader的log进度。

HeartBeat部分：

```
func (rf *Raft) SendHeartbeat() {

    for i := 0; i < len(rf.peers); i++ {
        if i != rf.me {
            rf.pmu.Lock()
            //DPrintf("%d heartbeat to %d\n",rf.me,i)
            rf.pmu.Unlock()
            go rf.SendHeartbeatEntries(i)
            time.Sleep(time.Duration(10) * time.Millisecond)
        }
    }
}

func (rf *Raft) SendHeartbeatEntries(p int) {

        rf.mu.Lock()
        if rf.state != Leader {
            rf.mu.Unlock()
            return
        }
        nextIndex := rf.nextIndex[p]
        entries := make([]LogEntry, 0)
        entries = append(entries, rf.log[nextIndex:]...)
        args := AppendEntriesArgs{
            Term:         rf.CurrentTerm,
            Leaderid:     rf.me,
            Entries:      entries,
            Prevlogindex: rf.getPrevLogIndex(p),
            Prevlogterm:  rf.getPrevLogTerm(p),
            Leadercommit: rf.commitIndex,
        }
        reply := AppendEntriesReply{}
        rf.mu.Unlock()
        rf.pmu.Lock()
        //DPrintf("leader %d  rf.me:%d rf.state %d send entry: nextindex:%d
prelogindex:%d prelogterm:%d
leadercommit:%d\n",args.Leaderid,rf.me,rf.state,nextIndex,args.Prevlogindex,args.Prevl
ogterm,args.Leadercommit)
        //DPrintf("leader %d  rf.me:%d send entry: nextindex:%d prelogindex:%d
prelogterm:%d leadercommit:%d
entrylen:%d\n",args.Leaderid,rf.me,nextIndex,args.Prevlogindex,args.Prevlogterm,args.L
eadercommit,len(args.Entries))
        rf.pmu.Unlock()
        ok := rf.sendAppendEntries(p, &args, &reply)
        if !ok {
            rf.pmu.Lock()
            //DPrintf("fail %d\n",p)
            rf.pmu.Unlock()
            return
        }
```

```
        //
        //rf.pmu.Lock()
        //DPrintf("leader %d get, reply.term %d curterm %d\n",rf.me,)
        //rf.pmu.Unlock()


        rf.mu.Lock()
        defer rf.mu.Unlock()
        if rf.CurrentTerm != args.Term {
            rf.pmu.Lock()
            //DPrintf("cur term:%d ;  args term %d\n",rf.CurrentTerm,args.Term)
            rf.pmu.Unlock()
            return
        }
        if reply.Term > rf.CurrentTerm {
            rf.pmu.Lock()
            //DPrintf("heartbeat leader back to Follower: %d
term:%d\n",rf.me,rf.CurrentTerm)
            rf.pmu.Unlock()
            rf.becomeFollower(reply.Term)
        }

        if reply.Success {
            rf.pmu.Lock()
            //DPrintf("leader %d follower %d :reply success: prelogindex:%d
len:%d\n",rf.me,p,args.Prevlogindex,len(args.Entries))
            rf.pmu.Unlock()
            rf.matchIndex[p] = args.Prevlogindex + len(args.Entries)
            rf.nextIndex[p] = rf.matchIndex[p]+1
            //commit
            rf.advanceCommitIndex()
            return
        }else{
            rf.pmu.Lock()
            //DPrintf("leader %d follower %d :reply fail prelogindex:%d
len:%d\n",rf.me,p,args.Prevlogindex,len(args.Entries))
            rf.pmu.Unlock()
            //rf.nextIndex[p] = args.Prevlogindex
            newIndex := reply.ConflictIndex
            for i := 0; i < len(rf.log); i++ {
                entry := rf.log[i]
                if entry.Term == reply.ConflictTerm {
                    newIndex = i + 1
                }
            }
            rf.nextIndex[p] = intMax(1, newIndex)
        }
}
```

heartbeat的框架和2A的相同，但是为了更清晰，分成两个函数去看。然后这里加了很多DPrintf，调试这种并发程序，真的很难。。。。。。

首先看leader需要传递哪些信息：

```
        nextIndex := rf.nextIndex[p]
        entries := make([]LogEntry, 0)
        entries = append(entries, rf.log[nextIndex:]...)
        args := AppendEntriesArgs{
            Term:         rf.CurrentTerm,
            Leaderid:     rf.me,
            Entries:      entries,
            Prevlogindex: rf.getPrevLogIndex(p),
            Prevlogterm:  rf.getPrevLogTerm(p),
            Leadercommit: rf.commitIndex,
        }
```

nextIndex是指此时leader认为的下一个该给这个follower的log号，所以后面发送给follower的log就是这个log号的后续。这个nextindex的初始化是通过leader在刚刚竞选成功时进行的，此时leader只需要默认其他follower已经和自己对齐即可，所以初始化的时候这个nextindex其实就是leader的log长度。那此时就会遇到一个问题：follower其实并没有和leader对齐，导致leader没有给全log。这时，其实follower就会拒绝这次heartbeat，并反馈给leader，让leader进行调整。

Prevlogindex和Prevlogterm就是leader认为的follower上一次得到的log的index和term。用来和follower实际的进行对比。

commitindex则是leader现在已经提交的log。用来维护提交一致性。

然后看反馈：

```
if reply.Term > rf.CurrentTerm {
            rf.pmu.Lock()
            //DPrintf("heartbeat leader back to Follower: %d
term:%d\n",rf.me,rf.CurrentTerm)
            rf.pmu.Unlock()
            rf.becomeFollower(reply.Term)
        }

        if reply.Success {
            rf.pmu.Lock()
            //DPrintf("leader %d follower %d :reply success: prelogindex:%d
len:%d\n",rf.me,p,args.Prevlogindex,len(args.Entries))
            rf.pmu.Unlock()
            rf.matchIndex[p] = args.Prevlogindex + len(args.Entries)
            rf.nextIndex[p] = rf.matchIndex[p]+1
            //commit
            rf.advanceCommitIndex()
            return
        }else{
            rf.pmu.Lock()
            //DPrintf("leader %d follower %d :reply fail prelogindex:%d
len:%d\n",rf.me,p,args.Prevlogindex,len(args.Entries))
            rf.pmu.Unlock()
            //rf.nextIndex[p] = args.Prevlogindex
            newIndex := reply.ConflictIndex
            for i := 0; i < len(rf.log); i++ {
                entry := rf.log[i]
                if entry.Term == reply.ConflictTerm {
                    newIndex = i + 1
                }
            }
            rf.nextIndex[p] = intMax(1, newIndex)
        }
```

首先，如果得到的term大于自身的term，那说明自身可能已经经历过离线或者说网络故障，导致其他节点长时间没有受到自己的心跳，重新选举leader了。所以此时就需要把自己转换为follower，不需要对日志进行操作，等下一次收到新的leader的心跳，再和新leader对齐就可以了。

其次，如果follower认可了这次心跳，说明经过这次心跳，follower已经和leader对齐了log，所以此时就更新matchindex，这个matchindex就记录了各个follower已经对齐的部分，所有最终一致性就需要靠这个matchindex进行保证。同时更新nextindex。这个时候，leader其实已经出现了一次潜在的提交机会，所以进入advancecommitindex。

```
func (rf *Raft) advanceCommitIndex() {
    sortedMatchIndex := make([]int, len(rf.matchIndex))
    copy(sortedMatchIndex, rf.matchIndex)
    sortedMatchIndex[rf.me] = len(rf.log) - 1
    rf.pmu.Lock()
    //DPrintf("node %d matchindex %d   lastapplied :%d:
",rf.me,len(sortedMatchIndex),rf.lastApplied)
    rf.pmu.Unlock()
    sort.Ints(sortedMatchIndex)
    N := sortedMatchIndex[len(rf.peers)/2]

    rf.pmu.Lock()
    //DPrintf("N %d rf.commitIndex %d ",N,rf.commitIndex)
    rf.pmu.Unlock()
    if rf.state == Leader && N > rf.commitIndex && len(rf.log)>0 && rf.log[N].Term ==
rf.CurrentTerm  {
        rf.commitIndex = N

        rf.applyLog()
    }
}
```

这个函数其实就是来判断能否进行提交。raft中leader的提交规则是当这个log已经被超过半数的follower接收。又因为raft保证一个log被提交，则其前面的所有log都已经被提交，所以其实只需要去判断已经所有follower同步的最大logindex的中位数就可以。如果这个中位数大于leader现在已经提交的log号，那就进行一次提交。

接着上面，如果reply失败，也就说明follower没有完成同步，这就是先前提到的问题，此时出现了leader的log没给全的问题。所以这个时候，leader就需要把要发送的log往前推。这里最简单的做法就是每次往前推1。但是这样的做法太慢，可能需要很多次heartbeat才能能够完成同步，导致出现大量RPC。所以我们可以通过term来进行加速，我们可以找到和leader认为的prevlogterm相同的term下的第一个logindex来做跨幅度的加速。

Appendentries部分

```
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) {
    // Your code here (2A, 2B).
    rf.mu.Lock()
    defer rf.mu.Unlock()

    isSuccess := false
    conflictTerm := 0
    conflictIndex := 0
    rf.LastHeartBeatTime = time.Now()
    prevLogTerm :=0
    if args.Prevlogindex>=0 && args.Prevlogindex<len(rf.log){
        prevLogTerm = rf.log[args.Prevlogindex].Term
    }

    if args.Term > rf.CurrentTerm{
        rf.pmu.Lock()
        //DPrintf("get heartbeat becomeFollower: %d  term:%d\n",rf.me,rf.CurrentTerm)
        rf.pmu.Unlock()
        rf.becomeFollower(args.Term)
    }
    //false
    if args.Term < rf.CurrentTerm {
        goto  process
    }
    //false
    if len(rf.log)-1 < args.Prevlogindex {
        conflictIndex = len(rf.log)
        goto process
    }
    //false
    if args.Prevlogindex > 0 && rf.log[args.Prevlogindex].Term != args.Prevlogterm {
        goto process
    }


    if args.Prevlogindex>=0 && args.Prevlogterm != prevLogTerm {
        conflictTerm = rf.log[args.Prevlogindex].Term
        for i := 0; i < len(rf.log); i++ {
            if rf.log[i].Term == conflictTerm {
                conflictIndex = i
                break
            }
        }
    }
    //save log
    for i := 0; i < len(args.Entries); i++{
        index := args.Prevlogindex+i+1
        if index>len(rf.log) -1{
            rf.log = append(rf.log,args.Entries[i])
        }else{
```

```
            if rf.log[index].Term != args.Entries[i].Term{
                rf.log = rf.log[:index]
                rf.log = append(rf.log,args.Entries[i])
            }
        }

    }

    if args.Leadercommit > rf.commitIndex {
        rf.commitIndex = args.Leadercommit
        if len(rf.log)-1 < rf.commitIndex {
            rf.commitIndex = len(rf.log) - 1
        }
    }

    isSuccess = true
    goto process

process:
    rf.applyLog()
    reply.Success = isSuccess
    reply.Term = rf.CurrentTerm
    reply.ConflictIndex = conflictIndex
    reply.ConflictTerm = conflictTerm
    return
}
```

这部分其实比较简单。

完全按照的是论文中的五个原则：

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)

实验结果：

```
lyj@ubuntu:~/Desktop/6.824/src/raft$ go test -run 2B
Test (2B): basic agreement ...
  ... Passed --   0.9  3   20    6121    3
Test (2B): RPC byte count ...
  ... Passed --   2.2  3   68  191884   11
Test (2B): agreement despite follower disconnection ...
  ... Passed --   6.5  3  152   44146    7
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   4.9  5  382   99203    3
Test (2B): concurrent Start()s ...
  ... Passed --   1.2  3   22    6944    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   5.0  3  174   45040    4
Test (2B): leader backs up quickly over incorrect follower logs
  ... Passed --  27.6  5 3058 2861347  102
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.6  3   86   27042   12
PASS
ok      _/home/lyj/Desktop/6.824/src/raft       50.806s
```

```
Done 500/500; 438 ok, 62 failed
lyj@ubuntu:~/Desktop/6.824/src/raft$
```

```
Done 100/100; 90 ok, 10 failed
lyj@ubuntu:~/Desktop/6.824/src/raft$
```

虽然通过了测试，但是好像还是有差不多10%的故障率。故障大多如下：

too many RPC bytes; got 212230, expected 150000

还需要进一步查明。

实验遇到的问题：

做2B的时候，遇到的问题比2A多很多。

有一个最主要的问题困扰我很久。

遇到了很多次 one(%v) failed to reach agreement的问题。

通过大量的DPrintf，最后定位到问题是log编号和提交的问题。

因为我一开始设置的初始化如下：

```
rf.commitIndex = 0
rf.lastApplied = 0
```

而提交的部分中的逻辑应该就是rf.commitIndex > rf.lastApplied。

但是此时，当第一个log到达leader时，index为0，按照前面所述的逻辑，follower同步完之后，得到的中位数肯定也是0，这意味着leader下一步要进行提交的就是0号log。但是此时就会导致无法通过这个判断，最后没有产生任何提交。

但是简单的把大于改成大于等于也不行。

当leader还没有得到log时，follower也会收到心跳，因为此时没有log，所以会返回成功，这时候同样会进入提交环节，并且提交了原本并不存在的0号log。

最后的解决办法是把这两个值初始化为-1。并且在提交的判断中加上一些保护判断，比如说

&& len(rf.log)>0

保证必须日志有内容，日志有内容时，rf.lastApplied必定已经增加，所以不会出现问题。