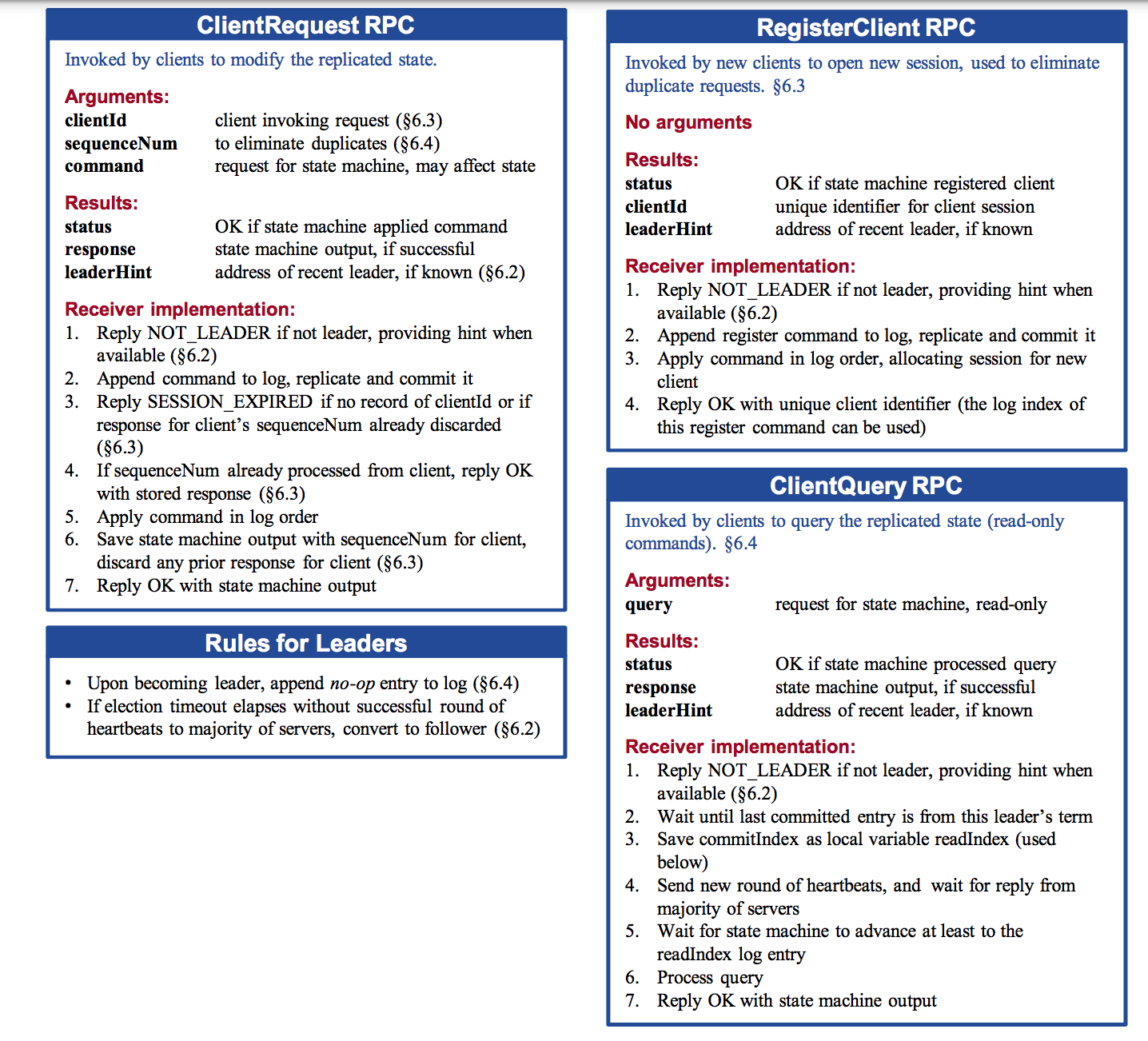
**[mit 6.824] lab3：KVRaft**

lab3 的内容是要在 lab2 的基础上实现一个高可用的 KV 存储服务，需要提供强一致性保证。强一致性的解释如下：对于单个请求，整个服务需要表现得像个单机服务，并且对状态机的修改基于之前所有的请求。对于并发的请求，返回的值和最终的状态必须相同，就好像所有请求都是串行的一样。即使有些请求发生在了同一时间，那么也应当一个一个响应。此外，在一个请求被执行之前，这之前的请求都必须已经被完成（在技术上我们也叫做线性化（linearizability））。

在做Lab3A前，需要熟读raft 作者[博士论文](https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf)的第6章。同时需要阅读raft论文的第8章。在完成Lab3B前，需仔细阅读raft论文的第7章。

**1 算法核心**

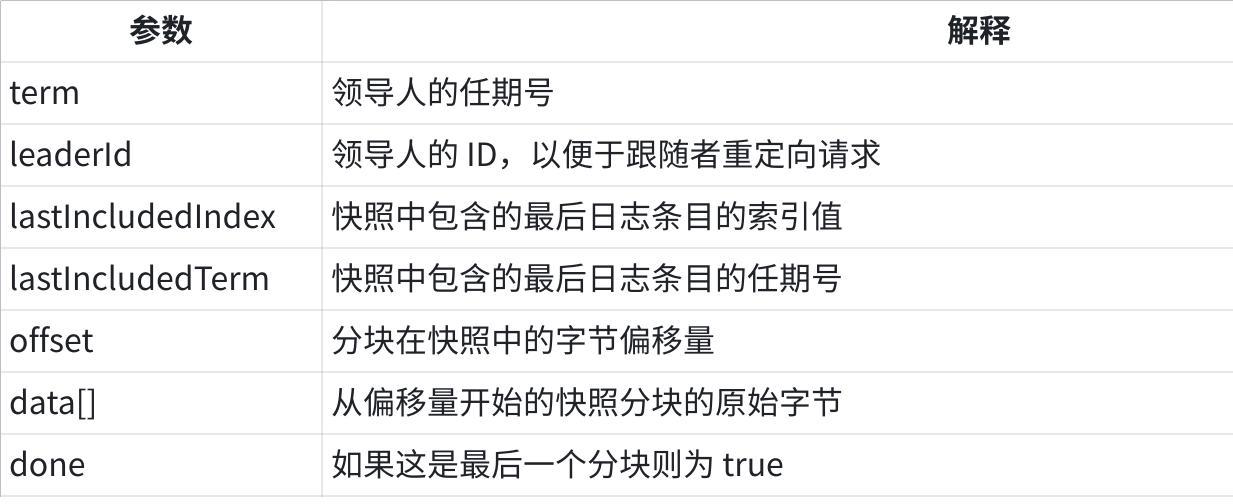
将raft应用于KV 存储服务的内容在raft 作者[博士论文](https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf)的第6章，请仔细阅读。



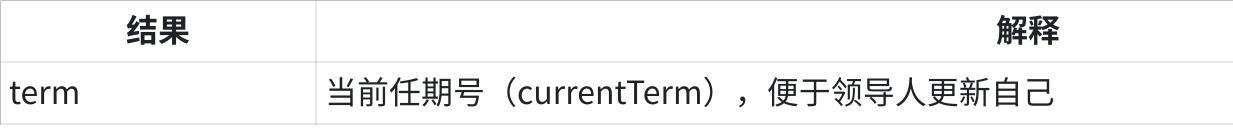
实现日志压缩在raft论文第八章。

**安装快照 RPC**：

由领导人调用以将快照的分块发送给跟随者。领导人总是按顺序发送分块。



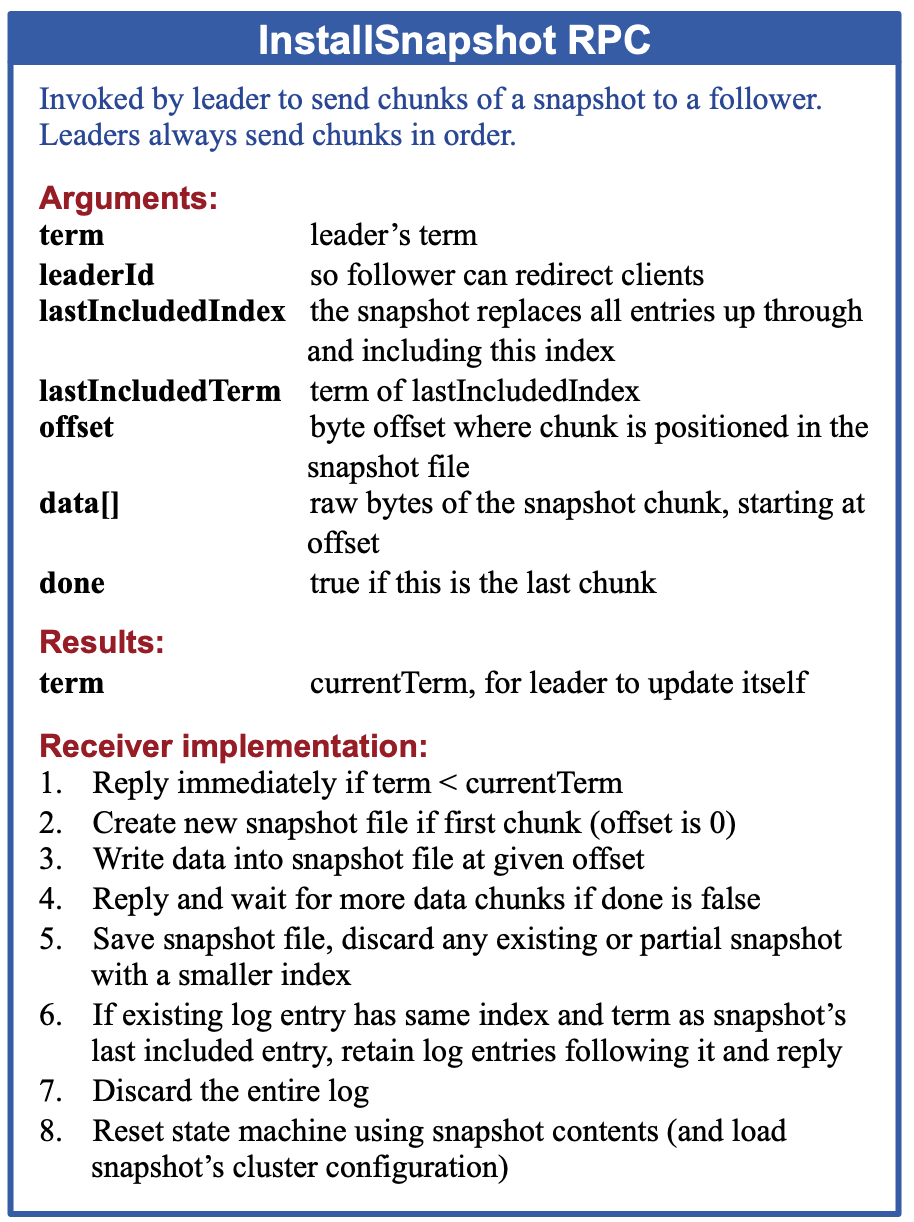
**点击图片可查看完整电子表格**



**点击图片可查看完整电子表格**

**接收者实现**：

1. 如果term < currentTerm就立即回复
2. 如果是第一个分块（offset 为 0）就创建一个新的快照
3. 在指定偏移量写入数据
4. 如果 done 是 false，则继续等待更多的数据
5. 保存快照文件，丢弃具有较小索引的任何现有或部分快照
6. 如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复
7. 丢弃整个日志
8. 使用快照重置状态机（并加载快照的集群配置）



**2 KV存储**

client比较简单。主要是通过RPC调用Server的get和putappend接口。

|  |
| --- |
| Go type Clerk struct {  servers []\*labrpc.ClientEnd  mu sync.Mutex  clientId int64  leaderId int  commandId int64 }  func (ck \*Clerk) Get(key string) string {  args := GetArgs{  Key: key,  }  ck.mu.Lock()  args.ClientId = ck.clientId  args.CommandId = ck.commandId  ck.commandId++  ck.mu.Unlock()   for {  //fmt.Println("get")  reply := GetReply{}  ok := ck.servers[ck.leaderId].Call("KVServer.Get", &args, &reply)  if ok && !reply.WrongLeader {  return reply.Value  }  ck.mu.Lock()  ck.leaderId = (ck.leaderId + 1) % len(ck.servers)  ck.mu.Unlock()  } }  func (ck \*Clerk) PutAppend(key string, value string, op string) {  args := PutAppendArgs{  Key: key,  Value: value,  Op: op,  }  ck.mu.Lock()  args.ClientId = ck.clientId  args.CommandId = ck.commandId  ck.commandId++  ck.mu.Unlock()   for {  reply := PutAppendReply{}  ok := ck.servers[ck.leaderId].Call("KVServer.PutAppend", &args, &reply)  if ok && !reply.WrongLeader {  return  }  ck.mu.Lock()  ck.leaderId = (ck.leaderId + 1) % len(ck.servers)  ck.mu.Unlock()  } } |

server端稍微复杂些。你需要定义一个状态机，就是最简易的kv存储。resultCh是logindex对应位置的结果。lastopack记录一个 client 已经处理过的最大 requestId。

|  |
| --- |
| Go type KVServer struct {  mu sync.Mutex  me int  rf \*raft.Raft  applyCh chan raft.ApplyMsg  dead int32 // set by Kill()   maxraftstate int // snapshot if log grows this big  // Your definitions here.  statemachine KVmemory  resultCh map[int]chan Res // logindex对应位置的结果  lastopack map[int64]int64 // 记录一个 client 已经处理过的最大 requestId } |

首先就是要完成get和putappend接口。他们接收get和putappend的请求，并传递给server的applier执行，并返回给client结果。Server层和Raft的交互流程为：Server(leader)收到RPC请求 → Start() → Raft层共识 → Server(全部)监听applyCh → Server(全部)执行操作 → Server(leader)返回RPC请求。

因此SubmitCommand函数的主要功能就是执行 Start() ，将op传递给Raft层共识，再等待处理结果。需要处理超时情况。

|  |
| --- |
| Go func (kv \*KVServer) SubmitCommand(op Op) Res {  //submit commands to the Raft log using Start()  //ApplyEntries() to rf.applych->KVSerer.applych  index, \_, isLeader := kv.rf.Start(op)  if !isLeader {  return Res{OK: false}  }   kv.mu.Lock()  \_, ok := kv.resultCh[index]  if !ok {  kv.resultCh[index] = make(chan Res, 1)  }  kv.mu.Unlock()   select {  case result := <-kv.resultCh[index]:  //fmt.Printf("client[%d] command[%d] get data\n", op.ClientId, op.CommandId)  if op.ClientId == result.ClientId && op.CommandId == result.CommandId {  return result  }  return Res{OK: false}  case <-time.After(TimeoutApply):  //fmt.Printf("client[%d] command[%d] timeout\n", op.ClientId, op.CommandId)  return Res{OK: false}  } } |

在 Raft层共识达成后，会将op传递到applyCh，那么这时就 Server的工作就是监听applyCh，实现如下函数：

|  |
| --- |
| Go func (kv \*KVServer) Applier() {  // keep reading applyCh while PutAppend() and Get() handlers submit commands to the Raft log using Start()  for {  applymsg := <-kv.applyCh  kv.mu.Lock()  if applymsg.UseSnapshot {  r := bytes.NewBuffer(applymsg.Snapshot)  d := labgob.NewDecoder(r)  var lastIncludedIndex, lastIncludedTerm int   if d.Decode(&lastIncludedIndex) != nil || d.Decode(&lastIncludedTerm) != nil || d.Decode(&kv.statemachine) != nil || d.Decode(&kv.lastopack) != nil {  panic("applier decode snapshot error")  }  } else {  op := applymsg.Command.(Op)  res := Res{  ClientId: op.ClientId,  Err: OK,  CommandId: op.CommandId,  OK: true,  WrongLeader: false,  }  if op.Command == CommandGet {  kv.lastopack[op.ClientId] = op.CommandId  res.Value, res.Err = kv.statemachine.Get(op.Key)  } else if op.Command == CommandPut {  if \_, ok := kv.lastopack[op.ClientId]; !ok {  res.Err = kv.statemachine.Put(op.Key, op.Value)  kv.lastopack[op.ClientId] = op.CommandId  } else {  if kv.lastopack[op.ClientId] >= op.CommandId {  res.Err = OK  } else {  kv.lastopack[op.ClientId] = op.CommandId  res.Err = kv.statemachine.Put(op.Key, op.Value)  }  }  } else {  if \_, ok := kv.lastopack[op.ClientId]; !ok {  kv.lastopack[op.ClientId] = op.CommandId  res.Err = kv.statemachine.Append(op.Key, op.Value)  } else {  if kv.lastopack[op.ClientId] >= op.CommandId {  res.Err = OK  } else {  kv.lastopack[op.ClientId] = op.CommandId  res.Err = kv.statemachine.Append(op.Key, op.Value)  }  }  }   if ch, ok := kv.resultCh[applymsg.CommandIndex]; ok {  select {  case <-ch: // drain bad data  default:  }  } else {  kv.resultCh[applymsg.CommandIndex] = make(chan Res, 1)  }  kv.resultCh[applymsg.CommandIndex] <- res   //the Raft state size is approaching maxraftsize, it should save a snapshot,  //and tell the Raft library that it has snapshotted, so that Raft can discard old log entries.  if kv.maxraftstate != -1 && kv.rf.GetRaftStateSize() > kv.maxraftstate {  //fmt.Println("reach maxraftstate")  w := new(bytes.Buffer)  e := labgob.NewEncoder(w)  e.Encode(kv.statemachine)  e.Encode(kv.lastopack)  snapshot := w.Bytes()  go kv.rf.CreateSnapshot(applymsg.CommandIndex, snapshot)  }  }  kv.mu.Unlock()  } } |

kv.applier协程：单独开一个go routine来远程监视apply channel，一旦底层的Raft commit一个到apply channel，状态机就立马执行且通过commandIndex通知到该客户端。要注意raft同步完成后，也需要判断请求是否为重复请求。因为同一请求可能由于重试会被同步多次。

**3 日志压缩**

日志肯定是不可以持续的增长下去的，因为当我们日志数量达到很大的时候，比如说我们的日志数据已经达到了几千万条的时候，我们和一个还没有多少数据的跟随者进行同步的话，需要将这些日志全部发送，其实是十分浪费资源和时间的。因此可以使用快照的方式，也就是对leader某一个时刻它的状态机的数据进行保存，然后将这个快照发送给那些很落后的节点进行快速的同步，同时由于快照已经记录此时的所有必要数据，那么我们可以将这些日志删除，避免日志无限度的增长下去。

Lab3B要求实现KV数据的持久化和raft log的抛弃，对Server的修改不多，主要的问题在于对Raft代码的修改，需要增加一个InstallSnapshot RPC方法，并考虑log index转换的问题。

在服务端需要持久化的是状态机kv键值对stateMachine和去重的 lastOperations 哈希表，然后通知raft进行日志压缩。同时有两种情况的持久化，分为：

（1）检测一次是否达到了maxraftstate，是则主动调用Snapshot

（2）Raft层检测rf.nextIndex[i]是否大于baseIndex（rf.log[0].Index），如果小于等于则调用InstallSnapshot从快照恢复数据。

InstallSnapshot定义和论文相同。但是是一次直接发送一整个快照过去。

|  |
| --- |
| Go type InstallSnapshotArgs struct {  Term int  LeaderId int  LastIncludedIndex int  LastIncludedTerm int  Data []byte  //Offset int  //Done bool }  type InstallSnapshotReply struct {  Term int } |

首先是对server的修改，状态机发现自己的目前的存储数据过大，那么就保存当前的状态机必须状态以及日志和Raft的必须状态到快照中。然后通知Raft对自己的日志进行丢弃，也就是调用Raft的Snapshot()。在applier中增加。

|  |
| --- |
| Go  if kv.maxraftstate != -1 && kv.rf.GetRaftStateSize() > kv.maxraftstate {  //fmt.Println("reach maxraftstate")  w := new(bytes.Buffer)  e := labgob.NewEncoder(w)  e.Encode(kv.statemachine)  e.Encode(kv.lastopack)  snapshot := w.Bytes()  go kv.rf.CreateSnapshot(applymsg.CommandIndex, snapshot)  } |

* 当领袖发送ApppendEntries RPC的时候，发现需要跟随者的nextIndex <= 日志数组中第一个日志的索引的时候，也就是需要发送的日志已经被丢弃了，那么就调用InstallSnapshot()来安装快照。InstallSnapshot()主要就是需要将lastIncludedIndex, lastIncludedTerm之前的log删除。当跟随者接收到领袖发来的快照的时候，若快照是正确的，那么就接收，并通过applyCh传递给状态机。
* 状态机接收到安装快照的请求，进行快照数据的应用，并且通知Raft去更新到该快照。

|  |
| --- |
| Go func (rf \*Raft) InstallSnapshot(args \*InstallSnapshotArgs, reply \*InstallSnapshotReply) {  rf.mu.Lock()  defer rf.mu.Unlock()   if args.Term < rf.currentTerm {  reply.Term = rf.currentTerm  return  }  if args.Term > rf.currentTerm {  rf.currentTerm = args.Term  rf.BecomeFollower()  rf.votedFor = -1  rf.persist()  }  reply.Term = rf.currentTerm  rf.electiontimer.Reset(GetRamdomTimeout())  if args.LastIncludedIndex > rf.commitIndex {  newlog := make([]LogEntry, 0)  lastIncludedIndex, lastIncludedTerm := args.LastIncludedIndex, args.LastIncludedTerm  newlog = append(newlog, LogEntry{Index: lastIncludedIndex, Term: lastIncludedTerm})   for i := len(rf.log) - 1; i >= 0; i-- {  if rf.log[i].Index == lastIncludedIndex && rf.log[i].Term == lastIncludedTerm {  newlog = append(newlog, rf.log[i+1:]...)  break  }  }  rf.log = newlog   rf.commitIndex = args.LastIncludedIndex  rf.lastApplied = args.LastIncludedIndex  rf.persister.SaveStateAndSnapshot(rf.GetRfState(), args.Data)   applymsg := ApplyMsg{  CommandValid: true,  UseSnapshot: true,  Snapshot: args.Data,  }  rf.applyCh <- applymsg  } }  func (rf \*Raft) sendInstallSnapshot(server int, args \*InstallSnapshotArgs, reply \*InstallSnapshotReply) bool {  ok := rf.peers[server].Call("Raft.InstallSnapshot", args, reply)  rf.mu.Lock()  defer rf.mu.Unlock()   if !ok || args.Term != rf.currentTerm || rf.State != Leader {  return ok  }  if reply.Term > rf.currentTerm {  rf.currentTerm = reply.Term  rf.BecomeFollower()  rf.votedFor = -1  rf.persist()  return ok  }  rf.matchIndex[server] = args.LastIncludedIndex  rf.nextIndex[server] = rf.matchIndex[server] + 1  return ok } |

**4 总结&采坑**

在lab3中，实现3A中的client server难度不大，但是修改Raft的时候还是遇到一些坑。

在每次丢弃已经commit的log时，都要rf.log[0].Index作为baseIndex。

InstallSnapshot后，更新rf.commitIndex = args.LastIncludedIndex，rf.lastApplied = args.LastIncludedIndex，同样，在RecoverSnapshot时也是要这样。

我定义了一个statemachine interface作为kv存储，和直接采用map[string]string作为kv存储是，采用gob encode有所不同。具体可以参考https://www.coder.work/article/194804。