MIT 6.824 Lab 2D&3A&3B

目录

前言

本报告是对完成MIT6.824 Lab3的详细说明，Lab3旨在使用Raft共识算法创建一个容错的键/值服务。

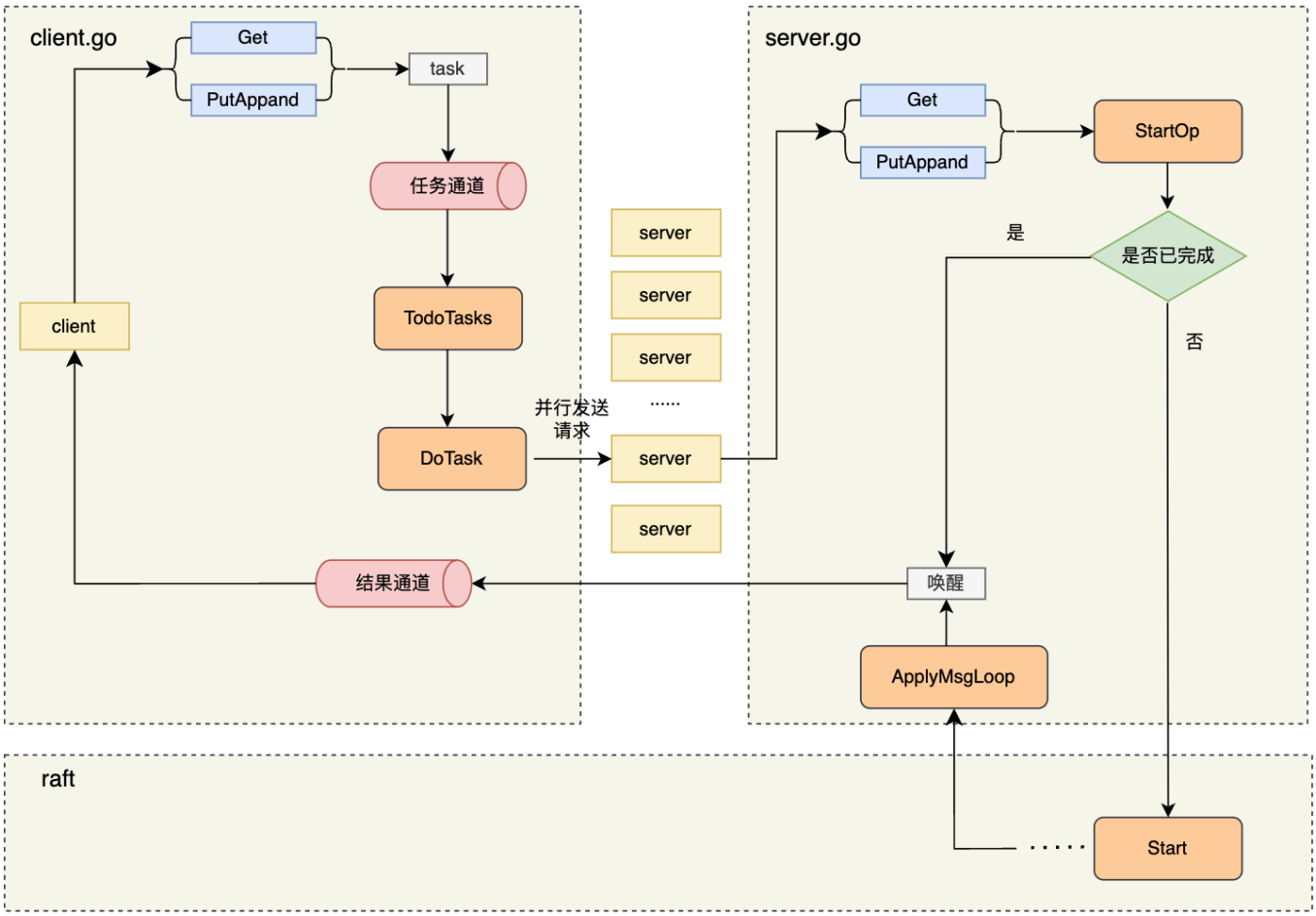
Lab 3A的任务是构建一个不使用快照的键/值服务。这包括实现处理Put、Append和Get操作的服务器，确保线性一致性，并处理网络和服务器故障时的领导选举和客户端请求。

Lab 3B在此基础上加入了快照功能。这涉及修改键/值服务器以创建和管理快照，从而优化日志空间使用并改善重启时间。该实验强调处理更大的状态大小，并确保在快照机制到位的情况下系统的功能性。

为了实现这一目标，我在客户端设计了Clerk结构体，作为与服务器交互的中介，管理客户端请求和服务器响应。它通过一个任务队列异步处理操作，使用结果通道同步返回操作结果。此外，我实现了一个后台运行的方法来处理队列中的任务，并通过并行发送RPC请求以提高效率。这种设计提高了客户端实现的灵活性和响应速度，同时确保了在网络波动和服务不稳定的情况下的鲁棒性。

在服务器端，我使用Op结构体封装客户端请求的操作，并通过KVServer结构体进行管理。这些操作通过一个持续运行的goroutine处理，监听Raft层的日志应用消息，并相应地更新服务器状态。此外，我还实现了处理RPC操作的方法，如Get和PutAppend，通过共识机制保证了系统的稳定性和一致性。整体来说，这些设计为构建一个高效、可靠的分布式键/值服务提供了坚实的基础。

我所设计的kvraft架构如下所示：



Lab3A

1 Client实现

在接下来的Client实现部分，我将介绍 `Clerk` 结构体及其核心方法和功能。这包括： `MakeClerk` 方法用于初始化客户端， `TodoTasks` 方法作为异步处理任务的中心，以及 `DoTask` 方法用于并行发送和处理服务器请求。此外，我还介绍了如何通过 `Get` ， `PutAppend` ， `Put` ，和 `Append` 等客户端接口方法，实现对分布式键值存储系统的高效、稳定且简洁的访问和操作。

1.1 准备工作

1.1.1 添加基本类型

```
// ClientId Client的标识
type ClientId int64

// RequestIndex Task的标识
type RequestIndex int64

// task 任务结构体 // 封装客户端的请求
type task struct {
  index    RequestIndex   // 对于当前客户端请求任务的标识
  op       string      // 任务类型，Get/Put/PutAppend
  key      string      // Get/PutAppend参数
  value    string      // PutAppend参数
  resultCh chan string // 传递Get的返回值并在Get/PutAppend上实现阻塞
}
```

1.1.2 Clerk结构体

Clerk结构体用于管理Client与多个Server的交互，由于我引入了任务队列机制，所以每个Client都有一个字段 `taskQueue` 。其余字段包括包含服务器列表、客户端标识、最后一次任务的索引以及上次成功的 Leader 服务器索引。实现如下：

```
  type Clerk struct {
  servers []*labrpc.ClientEnd
  // You will have to modify this struct.

  taskMu      sync.Mutex // 任务锁
  taskQueue   chan task // Client的任务队列
  clientId   ClientId  // Client的标识
  taskIndex   RequestIndex // 最后一条任务的下标(包括未完成的任务)
  leaderIndex int       // 上一次成功完成任务的Leader的Index，初始化为-1
}
```

1.2 MakeClerk方法

MakeClerk用于初始化Clerk结构体，并且调用Clerk的 `ToDoTasks` 方法来并发执行任务队列中的任务。

```
func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
  ck := &Clerk{
    servers:     servers,
    taskQueue:   make(chan task), // 通过make创建一个channel作为任务队列
    clientId:    nrand(),
    leaderIndex: -1,
  }
  go ck.ToDoTasks()
  return ck
}
```

其中，ClientId的生成方法是通过调用已有的函数`nrand()`（已重构，将返回值设置为ClientId）生成一个随机的标识。

1.3 TodoTask方法

`TodoTasks` 是 `Clerk` 的一个方法，它是任务队列机制的核心，负责持续从队列中获取任务，根据任务类型构建请求参数，发送 RPC 请求，并根据响应处理任务结果或重试。这个方法使得Client能够有效地与分布式存储系统进行交互，即使在网络延迟或服务器故障的情况下也能保持高效和稳定。

TodoTasks方法的实现逻辑如下：

1. 使用for构造无限循环，持续从`taskQueue`中获取任务，只要有任务在队列中就会一直运行。
2. 根据 `currentTask` 的操作类型（`op`），构造相应的 RPC 请求参数（`args`）。

   如果操作是 `Get`，则创建一个 `GetArgs` 结构体；如果是 `Put` 或 `Append`，则创建一个 `PutAppendArgs` 结构体。在common.go中为`GetArgs`和`PutAppendArgs`添加字段

   ```
   ClientId ClientId // 客户端标识
   TaskIndex RequestIndex // 任务索引
   ```
3. 加入内部for循环，持续调用`DoTask`，并行向所有的Servers发送该任务

   如果处理结果不是ErrNoleader，说明任务完成了，这时候可以将得到的value写入该任务的resultCh。如果处理结果是找不到Leader，则通过Sleep方法暂停调用一段等待时间，这样可以减少了向系统的不必要请求，从而降低了网络和服务器的负载。

具体实现如下：

```
func (ck *Clerk) TodoTasks() {
  //
  for {
    currentTask := <-ck.taskQueue // 从任务队列中取出任务
    DPrintf("Client %v: Get a task %v", ck.clientId, currentTask)
    var args interface{}
    // 根据任务类型，构造不同的参数
    switch currentTask.op {
    case "Get":
      args = &GetArgs{
        Key:       currentTask.key,
        TaskIndex: currentTask.index,
        ClientId:  ck.clientId,
```

```
        }
      case "Put", "Append", "PutAppend":
        args = &PutAppendArgs{
          Key:       currentTask.key,
          Value:     currentTask.value,
          Op:        currentTask.op,
          TaskIndex: currentTask.index,
          ClientId:  ck.clientId,
        }
      }
      for {
        // 调用DoTask()，并根据返回值判断任务是否完成
        // 如果任务返回值不是ErrNoLeader，说明任务完成，将返回值传给currentTask.resultCh，结束循环
        err, value := ck.DoTask(currentTask.op, args);
        if  err != ErrNoLeader {
          DPrintf("Client %v: Task %v finished", ck.clientId, currentTask)
          currentTask.resultCh <- value
          break
        }
        // 找不到Leader，等待一段时间后重试
        time.Sleep(clientNoLeaderSleepTime)
      }
    }
  }
```

1.4 DoTask方法

在2.3章节提到的TodoTasks方法中，Client在抽取任务后，是在一个无限循环中调用DoTask方法将任务并发到所有的Server上，直到得到有效回复后，此任务才被视为完成。所以下面要实现的事DoTask方法负责向分布式键值存储系统的服务器集群发送任务并处理响应。

DoTask的实现共分为两大步：**（1）发送任务**：`startTask` 向集群中的服务器发送一个指定的操作（如 `Get` 或 `Put/Append`），并等待响应。**（2）处理响应**：该函数处理服务器的响应，并确定操作是否成功执行或是否需要重试。

以下是该函数的功能和逻辑：

1. **并行发送任务**

   初始化通道和响应数组：创建 `replyCh` 和 `serverCh` 通道以接收服务器的响应和对应的服务器索引。同时，初始化 `replies` 数组以存储每个服务器的响应。

   发起 RPC 请求：定义了一个 `findServer` 函数，用于向指定服务器发送 RPC 请求，并将响应和服务器索引发送到 `replyCh` 和 `serverCh`。如果已知上次成功的 Leader 索引（`ck.leaderIndex`），则只向该服务器发送请求；否则，向所有服务器并行发送请求。

2. **处理响应和超时**

   超时设置：设定一个超时时间 `timeOut`，以避免无限期等待响应。

   接收和检查响应：使用 `select` 语句同时监听 `replyCh` 和超时信号。对于每个收到的响应，检查其有效性。

3. **解析并返回结果**

   处理 Get 请求的响应：如果操作是 `Get` 并且响应有效（没有错误或错误为 `ErrNoKey`），则更新 `ck.leaderIndex` 并返回 Get 的结果。

   处理 Put/Append 请求的响应：如果操作是 `Put` 或 `Append` 并且响应有效（没有错误），则更新 `ck.leaderIndex` 并返回一个空字符串表示操作成功。

   无响应或超时：如果所有尝试都失败（例如，没有服务器响应或都返回错误），或者操作超时，则函数返回 `ErrNoLeader`。

实现如下：

```
func (ck *Clerk) DoTask(op string, args interface{}) (Err, string) {
  /*
    op: 任务类型
    args: 任务参数

    返回值:
    Err: 错误类型
    string: Get返回值
  */

  // 1. 并行发送任务
  replyCh := make(chan interface{}, len(ck.servers)) // 接收响应的通道
  serverCh := make(chan int, len(ck.servers)) // 接收到该响应的服务器的索引
  replies := make([]interface{}, len(ck.servers)) // 初始化响应数组
  for index := range replies {
    if op == "Get" {
      replies[index] = &GetReply{}
    } else {
      replies[index] = &PutAppendReply{}
    }
  }

  // 定义一个匿名函数，用于发送任务
  findServer := func(server int){
    if op == "Get" {
      ck.servers[server].Call("KVServer.Get", args, replies[server])
    } else {
      ck.servers[server].Call("KVServer.PutAppend", args, replies[server])
    }
    replyCh <- replies[server]
    serverCh <- server
  }

  replyCount := 0 // 将收到响应的数量

  // 优先发给上一次成功完成任务的Leader
  if ck.leaderIndex != -1 {
    go findServer(ck.leaderIndex)
    replyCount = 1
```

```
  } else { // 如果上一次成功完成任务的Leader不存在，就发给所有服务器
    for server := 0; server < len(ck.servers); server++ {
      go findServer(server)
    }
    replyCount = len(ck.servers)
  }

  // 2. 处理响应
  // 持续检查replyCh，直到有可用的响应
  timeOut := time.After(clientDoTaskTimeOut)
  for ; replyCount > 0; replyCount-- {
    var reply interface{}
    select {
      case reply = <-replyCh:
      case <-timeOut:
        // 任务超时
        DPrintf("Client %v: Task %v time out, leader index: %v", ck.clientId, args,
ck.leaderIndex)
        ck.leaderIndex = -1
        return ErrNoLeader, ""
      }
    server := <-serverCh

    // 根据响应类型，处理响应
    if reply != nil {
      switch op {
      case "Get":
        getReply := reply.(*GetReply)
        if getReply.Err == OK || getReply.Err == ErrNoKey {
          ck.leaderIndex = server
          return getReply.Err, getReply.Value
          }
      case "Put", "Append", "PutAppend":
        putAppendReply := reply.(*PutAppendReply)
        if putAppendReply.Err == OK {
          ck.leaderIndex = server
          return putAppendReply.Err, ""
        }
      }
    }
  }
  // 没有可用的Leader或是保存的leaderIndex失效
  ck.leaderIndex = -1
  return ErrNoLeader, ""
}
```

1.5 操作函数

这部分主要是两个函数，Get函数和PutAppend函数，这两个函数在实现时，要注意以下两点：

1. 异步任务处理：通过将任务发送到一个队列并使用通道等待结果。这种方式使得任务的提交和结果的接收在时间上解耦，提高了系统的效率和响应性。

2. 线程安全：对共享资源的访问（如增加 `taskIndex` 和向 `taskQueue` 添加任务）需要被互斥锁保护，确保了在多线程环境下的安全性。

1.5.1 Get函数

Get用于获取指定键 `key` 的值，如果键不存在，返回空字符串（""）。实现如下：

```
func (ck *Clerk) Get(key string) string {
  // You will have to modify this function.
  resultCh := make(chan string)
  ck.taskMu.Lock()
  ck.taskQueue <- task{
    index:    ck.taskIndex + 1,
    op:       "Get",
    key:      key,
    value:    "",
    resultCh: resultCh,
  }
  ck.taskIndex++
  ck.taskMu.Unlock()
  return <-resultCh
}
```

1.5.2 PutAppend函数

PutAppend是一个共享函数，用于执行 `Put` 或 `Append` 操作。根据 `op` 参数的值（"Put" 或 "Append"），它将 `key` 和 `value` 发送到服务器以更新存储的数据。实现如下：

```
func (ck *Clerk) PutAppend(key string, value string, op string) {
  // You will have to modify this function.
  resultCh := make(chan string)
  ck.taskMu.Lock()
  ck.taskQueue <- task{
    index:    ck.taskIndex + 1,
    op:       op,
    key:      key,
    value:    value,
    resultCh: resultCh,
  }
  ck.taskIndex++
  ck.taskMu.Unlock()
  <-resultCh
}
```

1.6 Client实现总结

在Client的实现中，`Clerk` 结构体充当了与多个服务器交互的中介，管理客户端请求和服务器响应。每个 `Clerk` 实例都维护了一个任务队列 `taskQueue`，用于异步处理客户端操作，如 `Get`，`Put`，和 `Append`。这些操作被封装为 `task` 结构体，包含操作的类型、键值对参数以及一个结果通道 `resultCh`，后者用于同步返回操作结果。

核心方法 `TodoTasks` 在后台持续运行，从任务队列中取出任务并处理。它根据任务类型构造 RPC 请求，并通过 `DoTask` 方法并行地向服务器集群发送这些请求。`DoTask` 方法在处理服务器响应时考虑到了超时情况，确保了即使在网络延迟或服务器无响应的情况下也能保持稳定性。

Client接口方法如 `Get`，`Put`，和 `Append` 通过将任务添加到队列并等待 `resultCh` 通道的结果来简化Client与存储系统的交互。这种设计有效地分离了请求的发送和结果的等待，提高了客户端实现的灵活性和响应速度，同时也保证了在面对网络波动和服务不稳定时的鲁棒性。

2 Server实现

2.1 准备工作

2.1.1 Op结构体

Op结构体用于在 Raft 状态机和键值存储之间传递操作信息。它封装了Client请求的所有必要信息，使其成为可以在 Raft 系统中传播和共识的命令。它内部包括操作类型、要操作的键值对（`Key` 和 `Value`），以及客户端标识（`ClientId`）和该客户端请求的唯一索引（`RequestIndex`）。以下是我对Op结构体的定义：

```
type Op struct {
  // Your definitions here.
  // Field names must start with capital letters,
  // otherwise RPC will break.
  Type      string // 任务类型
  Key       string
  Value     string
  ClientId  ClientId  // 发送这条任务的Client的标识
  RequestId RequestIndex // 对应的Client的这条任务的下标
}
```

2.1.2 KVServer结构体

在分布式键值存储系统中，`KVServer` 对于Server端的作用与 `Clerk` 对于Client端的作用有着相似之处。它们都扮演着各自端点的中心协调者角色，管理其所在端点的关键操作和状态。`KVServer` 是Server端的核心组件，负责处理来自客户端的请求，维护键值存储的状态，以及与 Raft 协议集成以保证数据一致性。它需要实现以下功能：

- 处理客户端的 `Get`，`Put`，`Append` 等请求。
- 与 Raft 协议集成，确保跨多个服务器的数据一致性。
- 管理状态机的状态，如键值对的存储和更新。
- 维护与客户端请求相关的元数据，如客户端的最后任务索引。

我对于KVServer结构体的定义如下：

```
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.
  kv                  map[string]string      // kv数据库
  clientLastTaskIndex map[ClientId]RequestIndex // 每个Client已完成的最后一个任务的下标
  doneTask            map[int]int            // 已完成的任务(用于校验完成的Index对应的任务是不是自己发布的任务)
  doneCond            map[int]*sync.Cond     // Client发送到该Server的任务,任务完成后通知Cond回复Client
  persister    *raft.Persister
}
```

2.2 StartKVServer方法（初始化KVServer实例）

类似于`MakeClerk`，`StartKVServer`方法用于初始化`KVServer`，并且调用ApplyMsgLoop方法持续监听Raft 层的日志应用消息并更新服务器的状态，保证了Server端能够响应Client端的操作请求，并保持与集群中其他节点的状态同步。

在初始化一个KVServer实例后，要启动一个协程持续监听来自 Raft 层的日志应用消息，并根据这些消息来更新键值存储的状态，我将这个函数定义为ApplyMsgLoop，将在下一章节进行讲解。Loop是信息转接的意思，这个函数在一定程度上扮演了信息转接的作用。它作为服务器和 Raft 层之间的桥梁，接收来自 Raft 的日志应用消息，并根据这些消息来更新服务器的键值存储。

当一个KVServer处理了Client端的任务（比如 Put 或 Append 操作），它首先将这个任务通过 Raft 协议与集群中的其他节点进行同步。一旦该操作在多数节点上达成一致并被提交，其他节点就会通过ApplyMsgLoop从 Raft 层接收到这个操作的信息。

具体实现如下：

```
func StartKVServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister, maxraftstate int) *KVServer {
  // call labgob.Register on structures you want
  // Go's RPC library to marshall/unmarshall.
  labgob.Register(Op{})

  kv := new(KVServer)
  kv.me = me
  kv.maxraftstate = maxraftstate

  kv.applyCh = make(chan raft.ApplyMsg)
  kv.rf = raft.Make(servers, me, persister, kv.applyCh)

  kv.kv  = make(map[string]string)
  kv.clientLastTaskIndex = make(map[ClientId]RequestIndex)
  kv.doneTask = make(map[int]int)
  kv.doneCond = make(map[int]*sync.Cond)
  kv.persister = persister

  go kv.ApplyMsgLoop()

  return kv
}
```

2.3 ApplyMsgLoop方法（监听信息）

上一章节提到，ApplyMsgLoop方法是每个KVServer自初始化开始就进行的协程，持续监听来自 Raft 层的日志应用消息，并根据这些消息来更新键值存储的状态。

打个比方，想象这个kvraft集群是一家餐厅的管理团队。当一个顾客（Client）下达订单（例如 Put 或 Append 命令），他们首先会联系最近一次成功提供服务的服务员（Server）。服务员收到订单后，并不直接执行它。相反，他们首先要与其他服务员（集群中的其他服务器节点）协商，确保大多数服务员同意这个订单。这个协商过程由管理团队（Raft 协议）监督，以保证所有服务员对订单有一致的理解。一旦订单被多数服务员接受，管理团队就会指示原来的服务员执行这个订单。这确保了即使有服务员（服务器）出错或无法联系，餐厅（整个系统）依然能够提供一致和可靠的服务。

ApplyMsgLoop的逻辑可以概括为：

1. 持续监听 `applyCh` 通道，接收来自 Raft 的的日志消息。
2. 检查消息是否有效（`CommandValid`）并解析命令（`Op` 类型）。
3. 判断命令是否已执行（通过比较客户端最后任务索引）。
4. 对于未执行的 `Put` 或 `Append` 类型命令，更新键值存储并记录客户端任务索引。如果是Put类型命令，则直接添加键值对，如果是Append类型命令，键存在时直接追加该值，键不存在时创建该键值对。
5. 如果存在等待当前命令的 goroutine（通过 `doneCond` 检查），更新任务状态并通知它们。

具体实现如下：

```
func (kv *KVServer) ApplyMsgLoop() {
  // 只要节点没有终止就会一直接收信息
  for kv.killed() == false{
    applyMsg := <-kv.applyCh
    kv.mu.Lock()
    if applyMsg.CommandValid{ // 如果是有效的命令
      command,_ := applyMsg.Command.(Op) // 获取命令
      // 判断命令是否已经执行过
      taskLastIndex := kv.clientLastTaskIndex[command.ClientId]
      if command.Type != "Get" && taskLastIndex < command.RequestIndex {
        if command.Type == "Put" {
          kv.kv[command.Key] = command.Value
        } else {
          if _, ok := kv.kv[command.Key]; ok {
            kv.kv[command.Key] += command.Value
          } else {
            kv.kv[command.Key] = command.Value
          }
        }
        // 该任务的Index比之前存的任务Index大,更新
        kv.clientLastTaskIndex[command.ClientId] = command.RequestIndex
      }
      if cond, ok := kv.doneCond[applyMsg.CommandIndex]; ok {
        kv.doneTask[applyMsg.CommandIndex] = applyMsg.CommandTerm
        // 通知所有在等待该任务的goroutine
        cond.Broadcast()
```

```
      }
    }
    kv.mu.Unlock()
  }
}
```

2.4 任务处理RPC

2.4.1 StartOp方法

在之前的餐馆比喻中，其他服务员（节点）通过ApplyMsgLoop方法监听信息来更新自身状态，那让我们回到任务开始，当一个服务员（`KVServer`）收到顾客（Client）的请求后，会将这个请求传达给餐馆的管理团队（Raft 集群），进而与其他服务员（集群中的其他节点）进行协商，以达成关于如何处理这个请求的共识。只有当得到大多数服务员的同意后，请求才会被执行。所以我们需要一个函数来完成这种传达操作，我将这个方法定义为StartOp。

StartOp函数的逻辑如下：

1. 通过比较客户端的最后任务索引和当前任务索引，确定此任务是否已经执行过。如果已执行，直接返回，表示任务已完成。
2. 如果任务未执行过，使用 `kv.rf.Start` 将操作作为新的日志项提交给 Raft 层。这里创建了一个 `Op` 类型的结构体，包含操作类型、键值对、客户端标识和任务索引。
3. 检查当前服务器是否为 Raft 集群的领导者。如果不是领导者，返回相应的状态。
4. 如果这是一条新的日志项（即之前没有执行过这个操作），为这个日志项的索引创建一个新的条件变量，用于后续的同步处理。

具体实现如下：

```
func (kv *KVServer) StartOp(op string, key string, value string, clientTag ClientId,
clientTaskIndex RequestIndex) (bool, bool, int, int) {
  /*
    op: 任务类型
    key: 任务参数
    value: 任务参数
    clientTag: 发送这条任务的Client的标识
    clientTaskIndex: 对应的Client的这条任务的下标

    返回值:
    bool: 该任务是否已经完成过
    bool: 是否为Leader
    int: 该任务的Index
    int: 该任务的Term
  */


  // 这个任务已经完成过,直接返回
  if kv.clientLastTaskIndex[clientTag] >= clientTaskIndex {
    return true, false, 0, 0
  }
```

```
  // 提交Raft
  index, term, isLeader := kv.rf.Start(Op{
    Type:      op,
    Key:       key,
    Value:     value,
    ClientId:  clientTag,
    RequestIndex: clientTaskIndex,
  })

  if !isLeader {
    return false, false, 0, 0
  }
  if _, ok := kv.doneCond[index]; !ok {
    // 初始化该任务的Cond
    kv.doneCond[index] = &sync.Cond{L: &sync.Mutex{}}
  }
  return false, true, index, term
}
```

2.4.2 Get操作RPC

`Get` 函数用于处理客户端的 Get 请求，即获取特定键对应的值，该函数实现的逻辑如下：

1. 首先调用StartOp获取共识信息。Get函数就好比顾客的请求，在之前的餐馆比喻中，当一个服务员（`KVServer`）收到顾客（Client）的请求后，会通过StartOp方法将这个请求传达给餐馆的管理团队（Raft集群）。

2. 如果该请求已经被处理过，则直接从键值存储中返回对应的值，并设置响应状态。

3. 如果当前服务器不是领导者，则返回错误 `ErrWrongLeader`。在 Raft 协议中，只有领导者节点负责处理客户端的请求。如果当前服务器不是领导者，它无法保证处理请求的正确性和一致性，因为它可能没有最新的状态信息。返回 `ErrWrongLeader` 错误是一种告知客户端当前服务器不是领导者的方式，促使客户端重试请求并找到正确的领导者节点。

4. 如果是领导者并且请求尚未处理，则进行以下过程

   (a)获取与当前任务索引相关联的条件变量 `cond`。解锁 `kv.mu` 以允许其他操作进行，然后锁定 `cond` 来等待任务完成。

   (b)使用 `cond.Wait()` 使当前 goroutine 等待直到任务完成。任务完成后，重新加锁 `kv.mu`，检查任务的任期 `term` 是否与存储的任务任期 `kv.doneTask[index]` 匹配。

   (c)如果任务任期匹配，从键值存储中检索值并设置回复。如果键不存在，返回 `ErrNoKey`。如果任期不匹配，意味着领导者可能已改变，返回 `ErrWrongLeader`。

   (d)删除与任务索引相关的任期记录和条件变量，以避免资源泄漏。

5. 在操作完成后，检查该操作是否由当前服务器在相应的任期内处理。

具体实现如下：

```
func (kv *KVServer) Get(args *GetArgs, reply *GetReply) {
  // Your code here.
```

```
  kv.mu.Lock()
  done, isLeader, index, term := kv.StartOp("Get", args.Key, "", args.ClientId, args.TaskIndex)
  // 1. 任务已经完成
  if done == true {
    reply.Err = OK
    reply.Value = kv.kv[args.Key]
  } else if isLeader == false { // 2. 不是Leader
    reply.Err = ErrWrongLeader
  } else { // 3. 是Leader,等待任务完成
    cond := kv.doneCond[index]
    kv.mu.Unlock()
    cond.L.Lock() // 加锁，因为可能有多个Client在等待同一个任务
    cond.Wait()
    cond.L.Unlock()
    kv.mu.Lock()
    if term == kv.doneTask[index] { // 任务完成
      value, ok := kv.kv[args.Key]
      if ok {
        reply.Err = OK
        reply.Value = value
      } else {
        reply.Err = ErrNoKey
      }
    } else {
      reply.Err = ErrWrongLeader
    }
    delete(kv.doneTask, index)
    delete(kv.doneCond, index)
  }
  kv.mu.Unlock()
}
```

2.4.3 PutAppend操作RPC

PutAppend函数与Put的逻辑类似:

```
func (kv *KVServer) PutAppend(args *PutAppendArgs, reply *PutAppendReply) {
  // Your code here.
  kv.mu.Lock()
  done, isLeader, index, term := kv.StartOp(args.Op, args.Key, args.Value, args.ClientId,
args.TaskIndex)
  // 1. 任务已经完成
  if done == true {
    reply.Err = OK
  } else if isLeader == false { // 2. 不是Leader
    reply.Err = ErrWrongLeader
  } else { // 3. 是Leader,等待任务完成
    cond := kv.doneCond[index]
    kv.mu.Unlock()
    cond.L.Lock() // 加锁，因为可能有多个Client在等待同一个任务
    cond.Wait()
    cond.L.Unlock()
    kv.mu.Lock()
```

```
    if term == kv.doneTask[index] { // 任务完成
      reply.Err = OK
    } else {
      reply.Err = ErrWrongLeader
    }
    delete(kv.doneTask, index)
    delete(kv.doneCond, index)
  }
  kv.mu.Unlock()
}
```

2.5 Server实现总结

在Server端实现中，我使用Op结构体来封装客户端请求的操作，包括操作类型和键值对等信息。这些操作通过KVServer结构体进行管理，它是服务器的核心组件，负责处理客户端的Get、Put、Append等请求，并确保数据的一致性。服务器实例通过StartKVServer函数初始化，该函数启动一个持续运行的goroutine来监听Raft层的日志应用消息，以更新服务器状态。ApplyMsgLoop方法是这一过程的关键，它不断监听来自Raft的信息，并据此更新键值存储。此外，服务器还实现了处理RPC操作的方法，如Get和PutAppend，这些方法通过共识机制确保了系统的稳定性和一致性。

3 Lab3ADebug与测试

我在3A遇到的bug如下所示：

```
Test: one client (3A) ...
  ... Passed --  15.6  5  8182  127
Test: ops complete fast enough (3A) ...
--- FAIL: TestSpeed3A (121.23s)
    test_test.go:419: Operations completed too slowly 120.774201ms/op > 33.333333ms/op
Test: many clients (3A) ...
  ... Passed --  17.0  5 21182  635
Test: unreliable net, many clients (3A) ...
  ... Passed --  18.8  5  2939  352
Test: concurrent append to same key, unreliable (3A) ...
  ... Passed --   3.5  3   348   52
Test: progress in majority (3A) ...
  ... Passed --   1.0  5   369    2
Test: no progress in minority (3A) ...
  ... Passed --   1.8  5  2815    3
Test: completion after heal (3A) ...
```

这个bug显示我的代码实现需要150ms才完成一个op，我检查了很久，最终对比别人的代码发现，是我的raft提交日志的设计出现了一些问题，我的raft代码中将更新`apply`和`commitIndex`包含在Leader的ticker中，这样每次向Follower发送日志后，就会卡在select语句上，需要等下一次心跳后才会更新`apply`，后来我重构了代码，把更新`apply`和`commitIndex`的任务从raft的主协程中移出，设置一个单独的协程发送`applyMsg`给server，发送RPC的协程负责更新`commitIndex`就解决了问题。

重构的大致思路就是将每个函数解耦，然后把更新`apply`和`commitIndex`的任务从raft的主协程中移出，设置一个单独的协程发送`applyMsg`给server，发送RPC的协程负责更新`commitIndex`。由于重构的过程比较繁复，这里不详细解释。

```go
func Make(peers []*labrpc.ClientEnd, me int,
  persister *Persister, applyCh chan ApplyMsg) *Raft {
  /*
  其他代码
  */

  go rf.ticker()
  go rf.applier()
}
```

3A的测试通过如下：

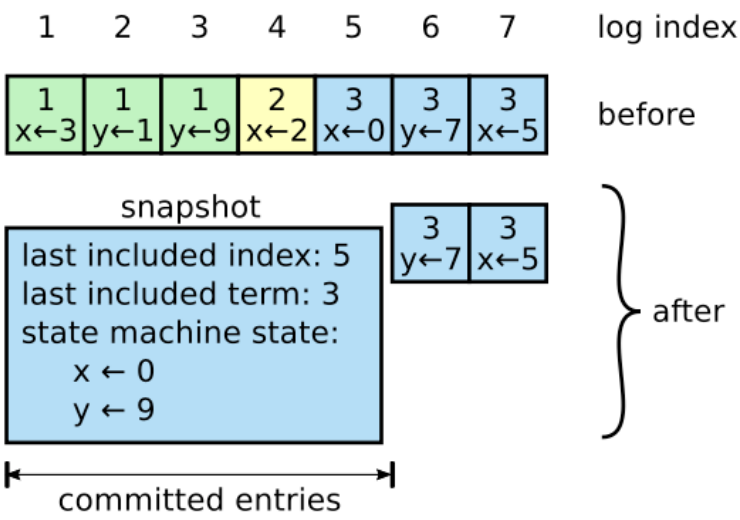
```
2024/01/12 22:55:12 Test3A Weichen Lyu
Test: one client (3A) ...
  ... Passed --  15.1  5 33383 6548
Test: ops complete fast enough (3A) ...
  ... Passed --   0.9  3  3043    0
Test: many clients (3A) ...
  ... Passed --  15.1  5 37676 7371
Test: unreliable net, many clients (3A) ...
  ... Passed --  16.3  5 11005 1540
Test: concurrent append to same key, unreliable (3A) ...
  ... Passed --   1.2  3   247   52
Test: progress in majority (3A) ...
  ... Passed --   0.5  5    58    2
Test: no progress in minority (3A) ...
  ... Passed --   1.0  5   131    3
Test: completion after heal (3A) ...
  ... Passed --   1.1  5    76    3
Test: partitions, one client (3A) ...
  ... Passed --  22.7  5 29519 5655
Test: partitions, many clients (3A) ...
  ... Passed --  22.7  5 38435 7270
Test: restarts, one client (3A) ...
  ... Passed --  19.9  5 32469 6313
Test: restarts, many clients (3A) ...
  ... Passed --  19.9  5 39713 7532
Test: unreliable net, restarts, many clients (3A) ...
  ... Passed --  21.2  5 11893 1483
Test: restarts, partitions, many clients (3A) ...
  ... Passed --  27.0  5 38596 7301
Test: unreliable net, restarts, partitions, many clients (3A) ...
  ... Passed --  27.0  5  8117  904
Test: unreliable net, restarts, partitions, random keys, many clients (3A) ...
  ... Passed --  29.5  7 46091 4563
PASS
ok      6.824/kvraft    241.934s
```

测试全部通过。

Lab2D

1　2D问题构思与回溯

对于 MIT 6.824 Lab 2D ，核心目标是实现 Raft 算法中的日志压缩功能。在当前基于Raft完成的键值系统中，日志不能无限制地增长。随着日志变长，它会占用更多空间，当节点重启时，将花费大量时间从头开始执行大量的log指令。所以Raft论文中要求，将快照之前的log都删掉。当节点重启时，仅需读取快照，并重新执行快照后的log指令即可恢复正确的状态。



上图展示了在Raft算法中日志压缩的过程。在进行快照之前，日志中有7个条目，索引从1到7，每个条目包含一个命令和一个任期号。快照记录了在日志索引5、任期3时的状态机状态，具体的状态值为x=0，y=9。进行快照之后，索引5之前的日志条目被丢弃，日志被压缩，只包含从索引5开始的条目，状态机状态反映了最后包含的索引和任期，以及x和y的当前值。这个过程减少了日志大小，并有效地存储了状态。

根据Lab 6.824提供的hints，对于2D部分，需要完成的主要任务有：完成Snapshot函数，该函数用于处理日志压缩；完成RPC函数 `InstallSnapshot` 和 `sendInstallSnapshot` ，这两个函数跟之前实现的 `RequestVote` / `sendRequestVote` 和 `AppendEntries` / `sendAppendEntries` 逻辑比较相似；根据要实现的日志压缩功能，对之前的代码进行修改，使整份代码能够正常工作。

在实现过程中需要注意以下几点：

由于不可能将所有日志永久地保存在内存中，因此对于已经应用到状态机的部分日志，就不再需要在 Raft 中维护。这意味着我们需要在 Raft 中记录额外的信息，比如 `lastIncludeIndex` 和 `lastIncludeTerm` ，这两者分别表示快照中最后一个日志的索引和任期。

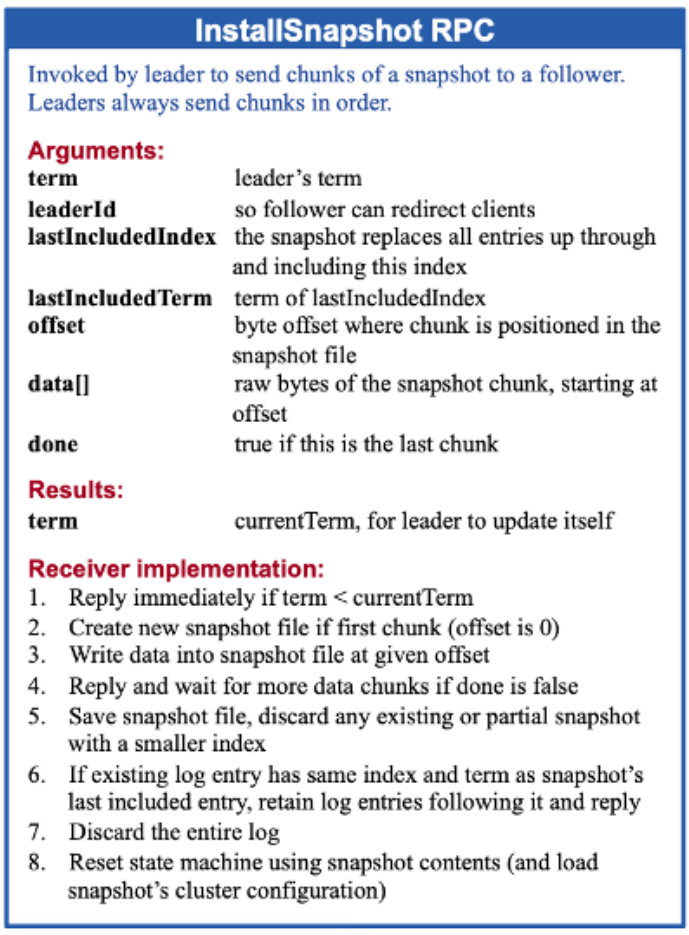
另外， `Snapshot()` 函数由状态机调用，传入的 `index` 表示 `lastIncludeIndex` ，而 `snapshot` 由状态机生成，需要 Raft 保存以便在必要时发送给 Follower。

在 `InstallSnapshot RPC` 中，Leader 需要在同步日志时考虑快照的存在。如果 Leader 中原本应该被同步到 Follower 的日志已经包含在快照中，则应该将快照发送给 Follower。

实验的过程中需要注意的是，对于 Raft 状态机的交互和日志处理逻辑，需要仔细设计以避免潜在的同步问题和死锁。

2 结构体修改

根据论文中InstallSnapshot的指引在Raft结构体中添加字段。



InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

字段：
•term：领导者的任期号。
•leaderId：领导者的ID，以便追随者可以重定向客户端。
•lastIncludedIndex：快照包含的最后日志条目的索引。
•lastIncludedTerm：与lastIncludedIndex对应的任期号。
•offset：快照文件中的字节偏移量，标识数据块的开始位置。
•data[]：从offset开始的快照数据块的原始字节。
•done：如果这是最后一个数据块，则为true。
结果：
•term：当前任期号，用于领导者更新自己。
接收者实现：
1.如果接收到的任期号小于当前任期号，则立即回复。
2.如果是第一个数据块（偏移量为0），则创建新的快照文件。
3.将数据写入快照文件的指定偏移量处。
4.如果done为false，回复并等待更多数据块。
5.保存快照文件，丢弃任何现有的或部分的快照，如果它们的索引更小。
6.如果存在具有相同索引和任期号的日志条目，则保留它之后的日志条目并回复。
7.丢弃整个日志。
8.使用快照内容重置状态机（并加载快照的集群配置）。

在Raft结构体中添加字段

```
// 2D需要的字段
lastIncludeIndex  int          // snapshot保存的最后一条日志的index
lastIncludeTerm   int          // snapshot保存的最后一条日志的term
snapshotCmd       []byte       // snapshot中的状态机快照
```

- `lastIncludeIndex`：快照中包含的最后一条日志条目的索引。通过这个索引，系统可以知道快照所代表的状态是基于日志中哪个点的数据。
- `lastIncludeTerm`：快照中包含的最后一条日志条目的任期号。它与`lastIncludeIndex`结合使用，可以确保一致性和日志条目的正确顺序。
- `snapshotCmd`：包含实际的快照数据，通常是状态机的序列化状态。当需要恢复状态机状态，或者将状态发送给其他节点时需要使用此数据。

这三个字段使得Raft节点能够创建快照，并在必要时将其发送给其他节点，或者在启动时从快照中恢复状态机的状态。

3 Snapshot实现

函数 `Snapshot` 的作用是在 Raft 状态机中创建一个快照，以便将当前状态机的状态压缩并保存，减少日志条目的数量。它的输入参数包括：`index`：快照包含的最后一条日志条目的索引；`snapshot`：状态机的当前状态的字节序列化形式，此处的数据类型与状态机快照相同，为`[]byte`

函数的处理逻辑如下：

- 首先检查节点是否已被关闭。
- 计算要保留的日志条目的位置。
- 更新 `lastIncludeIndex` 为快照中最后条目的索引。
- 更新 `lastIncludeTerm` 为该位置的日志条目的任期号。
- 截断日志条目，移除快照前的所有条目。
- 保存序列化的状态机状态到 `snapshotCmd`。

该函数不需要返回值，但函数会修改 Raft 状态机的内部状态。这样做后，节点的日志会从新的 `lastIncludeIndex` 开始，丢弃之前的所有条目，并以快照的形式保留状态机的状态。

具体实现如下：

```
func (rf *Raft) Snapshot(index int, snapshot []byte) {
  // Your code here (2D).
  if rf.killed() {
    return
  }
  pos := index - rf.lastIncludeIndex-1
  rf.lastIncludeIndex = index
  rf.lastIncludeTerm = rf.logs[pos].Term
  rf.logs = rf.logs[pos+1:]
  rf.snapshotCmd = snapshot
}
```

4 日志压缩RPC实现

4.1 RPC包定义

与之前选举RPC和日志复制RPC的逻辑相似，需要事先实现日志压缩请求的`InstallSnapshotRequest`和日志压缩处理的回复InstallSnapshotReply。对于二者我的定义如下所示：

```
// 日志压缩RPC的结构
type InstallSnapshotRequest struct {
  Term           int  // 任期号
  LeaderId       int  // Leader标识
  LastIncludeIndex int  // 快照中最后一条日志的索引
  LastIncludeTerm  int  // 快照中最后一条日志的任期号
  Data           []byte // 快照数据
}
```

```
// 对日志压缩请求的回复
type InstallSnapshotReply struct {
  Term         int  // 用于让leader更新
  InstallState          InstallSnapshotState  // 快照安装状态
}

// 枚举日志压缩状态
type InstallSnapshotState int
const (
  InstallNormal InstallSnapshotState = iota // 安装正常
  InstallTermExpire // 接收快照的任期已过期
  InstallLogExpire  // 表示接收快照的日志已过期或无效
)
```

4.2 InstallSnapshot实现

`InstallSnapshot` 函数用于Follower处理 InstallSnapshot RPC，决定是否日志压缩并返回快照安装的状态信息。当一个节点的日志条目缺失或与Leader的日志有显著差异时，它会将服务器的日志更新为与领导者的快照相匹配。如果请求的任期比当前任期旧，函数会立即返回。如果快照更新，则会更新服务器的日志、提交索引和最后应用的索引，并发送信号以将快照应用到状态机。

`InstallSnapshot` 的具体逻辑如下：

1. 检查请求的任期是否有效：如果请求的任期小于当前任期，则立即回复并表明请求已过期 `InstallTermExpire`
2. 检查是否需要安装快照：如果请求的快照索引 `args.LastIncludeIndex` 不超过当前已包含的最后一条日志索引 `rf.lastIncludeIndex`，说明该快照已经安装或过时，不需要再安装，并回复过时信息 `InstallLogExpire`
3. 安装快照：将当前任期更新为请求的任期，标记为跟随者，并重置选举超时。
4. 更新日志条目：根据快照中包含的最后日志索引和任期，裁剪或更新本地日志。
5. 应用快照：向状态机发送应用快照的信号，并更新最后应用的日志索引和提交索引。
6. 回复RPC：设置回复的任期为当前任期，并标记安装状态为正常。

`InstallSnapshot` 的实现如下：

```
func (rf *Raft) InstallSnapshot(args *InstallSnapshotRequest, reply *InstallSnapshotReply) {
  /*
    args：快照RPC
    reply：快照RPC的响应
  */

  DPrintf("[INFO][InstallSnapshot][收到 RPC]Node %d: InstallSnapshot called with args %+v\n",
rf.me, args)

  // 检查一次节点是否被终止
  if rf.killed() {
    reply.Term = args.Term
    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
    DPrintf("[INFO][InstallSnapshot][安装 失败]Node %d: InstallSnapshot - Current Term: %d,
Requested Term: %d, returning InstallTermExpire\n", rf.me, rf.currentTerm, args.Term)
    reply.InstallState = InstallTermExpire
    return
  }

  // 错误消息
  if args.LastIncludeIndex <= rf.lastIncludeIndex {
    reply.Term = rf.currentTerm
    DPrintf("[INFO][InstallSnapshot][安装 失败]Node %d: InstallSnapshot - Current
LastIncludeIndex: %d, Requested LastIncludeIndex: %d, returning InstallLogExpire\n", rf.me,
rf.lastIncludeIndex, args.LastIncludeIndex)
    reply.InstallState = InstallLogExpire
    rf.timer.Reset(rf.overtime)
    return
  }

  // 安装快照
  rf.currentTerm = args.Term
  rf.votedFor = args.LeaderId
  rf.status = Follower
  rf.timer.Reset(rf.overtime)

  if len(rf.logs)+rf.lastIncludeIndex <= args.LastIncludeIndex {
    rf.logs = []LogEntry{}
    rf.lastIncludeIndex = args.LastIncludeIndex
    rf.lastIncludeTerm = args.LastIncludeTerm
  } else {
    rf.logs = rf.logs[args.LastIncludeIndex-rf.lastIncludeIndex:]
    rf.lastIncludeIndex = args.LastIncludeIndex
    rf.lastIncludeTerm = args.LastIncludeTerm
  }

  rf.applyChan <- ApplyMsg{
    SnapshotValid: true,
    Snapshot:      args.Data,
    SnapshotTerm:  args.LastIncludeTerm,
    SnapshotIndex: args.LastIncludeIndex,
  }
  rf.lastApplied = args.LastIncludeIndex
  rf.commitIndex = rf.lastApplied

  reply.Term = rf.currentTerm
  reply.InstallState = InstallNormal
```

```
  DPrintf("[INFO][InstallSnapshot][安装 完成]Node %d: InstallSnapshot - Success, returning
InstallNormal\n", rf.me)
  return
}
```

4.3 sendInstallSnapshot实现

`sendInstallSnapshot` 函数负责Leader向Follower发送 InstallSnapshot RPC。它会重复尝试发送RPC，直到成功或服务器被关闭并且根据跟随者的响应来更新服务器对任期和下一个索引的了解，这表明快照是否安装成功。如果跟随者的任期更近，服务器会更新其状态并持久化这些信息。

`sendInstallSnapshot` 的具体逻辑如下：

1. 检查服务器是否已终止。
2. 调用`rf.peers[server].Call`方法发送InstallSnapshot RPC给指定的服务器。
3. 如果RPC调用失败，则重新尝试，直到成功或服务器被终止。
4. 在获取到回复后，如果回复的任期小于当前任期，则放弃这次RPC调用。
5. 根据回复的状态进行处理：
    - 如果状态为`InstallNormal`，并且回复的任期大于当前任期，更新服务器状态为Follower，并重置超时计时器。
    - 如果状态为`InstallLogExpire`，进行相应的日志索引更新。
    - 如果状态为`InstallTermExpire`，不做特殊处理。

`sendInstallSnapshot` 的实现如下：

```
func (rf *Raft) sendInstallSnapshot(server int, args *InstallSnapshotRequest, reply
*InstallSnapshotReply) bool {
  /*
    server: 目标服务器的索引
    args: 快照RPC
    reply: 快照RPC的响应
  */


  if rf.killed() {
    return false
  }

  // 调用rf.peers[server]的InstallSnapshot方法，发送快照RPC
  ok := rf.peers[server].Call("Raft.InstallSnapshot", args, reply)
  for !ok {
    if rf.killed() {
      return false
    }
    ok = rf.peers[server].Call("Raft.InstallSnapshot", args, reply)
  }
```

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

  if reply.Term < rf.currentTerm {
    return false
  }

  // 根据RPC的回复进行分支处理
  switch reply.InstallState {
  case InstallNormal:
    if reply.Term > rf.currentTerm {
      rf.status = Follower
      rf.currentTerm = reply.Term
      rf.votedFor = -1
      rf.timer.Reset(rf.overtime)
      rf.persist()
    }
    rf.nextIndex[server] = args.LastIncludeIndex+1
  case InstallLogExpire:
    if reply.Term > rf.currentTerm {
      rf.status = Follower
      rf.currentTerm = reply.Term
      rf.votedFor = -1
      rf.timer.Reset(rf.overtime)
      rf.persist()
    }
    rf.nextIndex[server] = len(rf.logs)+rf.lastIncludeIndex+1
  case InstallTermExpire:
  }

  return false
}
```

5 其他函数修改

5.1 Make函数修改

由于为Raft添加了字段，所以Make函数需要补充初始化的字段：

```
  rf.lastIncludeIndex = -1
  rf.lastIncludeTerm = 0
  rf.snapshotCmd = make([]byte, 0)
```

5.2 RequestVote选举限制修改

进行日志压缩后，原有的日志条目被压缩为一个快照，所以日志数组的开始索引将不再是0，而是压缩后的最后一条日志的索引。因此，需要调整选举时的日志比较逻辑，以确保选举时能正确处理压缩后的日志情况。

修改的主要逻辑如下：

1. **考虑快照的最后任期**：通过比较候选人的最后日志任期`LastLogTerm`与当前节点的快照中最后一条日志的任期`lastIncludeTerm`，以确保候选人的日志不落后于当前节点的快照。

2. **考虑快照的索引位置**：如果候选人的最后日志索引`LastLogIndex`小于当前节点的快照中最后一条日志的索引`lastIncludeIndex`，这意味着候选人的日志比当前节点的快照还要旧，不应给予选票。

实现如下：

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

  // 修改选举限制 2D
  if args.LastLogTerm < rf.lastIncludeTerm || args.LastLogTerm < lastLogTerm{

    rf.currentTerm = args.Term
    reply.Term = args.Term
    reply.VoteGranted = false
    reply.VoteState = LogExpire
    rf.persist() // 如果这个节点是一个过时的leader，那么它的日志可能是最新的，但是它的term已经过期了，所以需要持久化
    DPrintf("[INFO][RequestVote]Node %d: RequestVote - Vote Not Granted\n", rf.me)
    return
  }

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

5.3 AppendEntries日志复制修改

AppendEntries的RPC函数一共要进行两处修改：

第一处是对于请求任期判断的修改。当进行日志压缩后，节点的日志开始索引会发生变化。因此，需要增加一个判断条件`args.PrevLogIndex < rf.lastIncludeIndex`，确保当接收到的日志复制请求中的上一个日志索引`PrevLogIndex`小于当前节点的快照中最后一条日志的索引`lastIncludeIndex`时，该请求应被认为是过期的。

```
  if args.Term < rf.currentTerm  || args.PrevLogIndex < rf.lastIncludeIndex {
    reply.Term = rf.currentTerm
    reply.Success = false
    reply.AppState = AppExpire // 回复状态设置为过时
    reply.TmpNextIndex = -1
    DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - Current Term: %d, Requested Term: %d, returning AppExpire\n", rf.me, rf.currentTerm, args.Term)
    return
  }
```

由于节点的日志索引会发生变化，因此第二处修改是针对日志不匹配的情况适当地调整日志复制的逻辑以确保数据的一致性。

1. **对PrevLogIndex的处理**：当`args.PrevLogIndex`不等于`lastIncludeIndex`时，需要检查这个索引是否超出了当前日志的范围（即是否大于等于`len(rf.logs) + rf.lastIncludeIndex + 1`）或者该索引处的日志条目任期是否与请求中的`args.PrevLogTerm`不符。

2. **处理日志压缩情况**：当`args.PrevLogIndex`等于`lastIncludeIndex`但其任期与`lastIncludeTerm`不符时，也被认为是日志不匹配的情况。这种情况特别处理的原因是，在日志压缩之后，`lastIncludeIndex`之前的日志条目已经不在当前节点的日志数组中，因此需要单独判断这种情况。

修改后的逻辑如下：

```
  if (args.PrevLogIndex != rf.lastIncludeIndex && (args.PrevLogIndex >=
len(rf.logs)+rf.lastIncludeIndex+1 || args.PrevLogTerm != rf.logs[args.PrevLogIndex-
rf.lastIncludeIndex-1].Term)) ||
     (args.PrevLogIndex == rf.lastIncludeIndex && args.PrevLogTerm != rf.lastIncludeTerm){
     reply.Term = rf.currentTerm
     reply.Success = false
     reply.AppState = AppMismatch
     reply.TmpNextIndex = rf.lastApplied + 1
     rf.persist()
     DPrintf("[INFO][AppendEntries][同步 失败]Node %d: AppendEntries - AppMismatch, returning
Mismatch\n", rf.me)
     return
   }
```

5.4 sendAppendEntries修改

主要修改是针对日志复制正常中确认日志条目有效性的修改，此处修改依然是为了适应日志压缩后的索引调整。原先的代码比较的是`rf.logs[args.LogIndex].Term`和`rf.currentTerm`，这假设了日志索引`args.LogIndex`是从 0 开始连续的。但在引入日志压缩后，日志数组的起始索引可能不再是 0，因此需要调整这个比较逻辑。需要考虑一下两种情况：

1. `args.LogIndex`大于快照中包含的最后一个日志的索引`rf.lastIncludeIndex`时，它从调整后的日志数组中获取对应日志项（考虑到了日志压缩后的索引偏移）。

2. 当`args.LogIndex`等于快照中包含的最后一个日志的索引时，它直接使用快照中的最后一个日志项的任期来进行比较。

修改如下：

```
  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
      // 修改 2D
      if (args.LogIndex>rf.lastIncludeIndex && rf.logs[args.LogIndex-rf.lastIncludeIndex-1].Term
!= rf.currentTerm) ||
        (args.LogIndex == rf.lastIncludeIndex && rf.lastIncludeTerm != rf.currentTerm){
        return false
      }
      for rf.lastApplied < args.LogIndex {
        rf.lastApplied++
        applyMsg := ApplyMsg{
          CommandValid: true,
          Command:      rf.logs[rf.lastApplied-rf.lastIncludeIndex-1].Command,
          CommandIndex: rf.lastApplied,
        }
        rf.applyChan <- applyMsg
        rf.commitIndex = rf.lastApplied
      }
    }
```

5.5 ticker函数修改

ticker函数修改有两处。

由于引入了日志压缩，所以Candidate构造选举RPC时，对于LastLogIndex和LastLogTerm的初始化发生了变化：

```
      case Candidate:
        // 进行选举
        rf.currentTerm += 1
        rf.votedFor = rf.me
        // 每轮选举开始时，重新设置选举超时
        rf.overtime = time.Duration(rand.Intn(150)+200) * time.Millisecond
        voteNum := 1
        rf.persist()
        rf.timer.Reset(rf.overtime)
        // 构造msg
        for i, _ := range rf.peers {
          if i == rf.me {
            continue
          }
          // 构造投票RPC
          // 修改 2D
          voteArgs := &RequestVoteArgs{
            Term:         rf.currentTerm,
            CandidateId:  rf.me,
            LastLogIndex: len(rf.logs) +rf.lastIncludeIndex,              // 最后一条日志的索引
            LastLogTerm:  rf.lastIncludeTerm, // 最后一条日志的任期
          }
          if len(rf.logs) > 0 {
            voteArgs.LastLogTerm = rf.logs[len(rf.logs)-1].Term
          }
          voteReply := new(RequestVoteReply)
```

```
            go rf.sendRequestVote(i, voteArgs, voteReply, &voteNum) // 异步发送投票RPC
        }
```

第二处修改是对 Leader 心跳计时器超时后的操作进行了调整，以适应日志压缩的逻辑。修改的主要部分是在发送日志复制RPC 之前检查是否需要发送日志压缩RPC。如果 `rf.nextIndex[i]` 小于等于 `rf.lastIncludeIndex`，则代表目标节点的日志落后于快照，需要发送日志压缩RPC而不是常规的日志复制RPC。

具体修改的逻辑如下：

1. **检查是否需要发送快照**：添加一个检查 `if rf.nextIndex[i] <= rf.lastIncludeIndex`，这是用来判断是否需要向对应的节点发送日志压缩RPC。如果条件成立，说明目标节点的日志信息落后于当前的快照信息，需要发送RPC以更新其状态。

2. **修改心跳/日志复制RPC的构建**：对 `appendEntriesArgs` 的构造进行了调整，特别是在设置 `PrevLogIndex` 和 `PrevLogTerm` 时，考虑到了日志压缩的情况。如果 `PrevLogIndex` 等于 `rf.lastIncludeIndex`，则使用 `rf.lastIncludeTerm` 作为 `PrevLogTerm`。这样做是为了确保在日志压缩后，日志条目的索引和任期能够正确地匹配。

3. **对日志条目的复制进行修改**：在复制日志条目时，根据日志压缩调整了复制的起始点，确保日志的连续性和完整性。

```
    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
        // 修改 2D
        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更
新
          LeaderCommit: rf.commitIndex,    // Leader的提交索引
          LogIndex:     len(rf.logs)+rf.lastIncludeIndex, // 当前日志的最新索引
        }

        //installSnapshot，如果rf.nextIndex[i]小于等lastCludeIndex,则发送snapShot
        if rf.nextIndex[i] <= rf.lastIncludeIndex {
          installSnapshotReq := &InstallSnapshotRequest{
            Term:             rf.currentTerm,
            LeaderId:         rf.me,
```

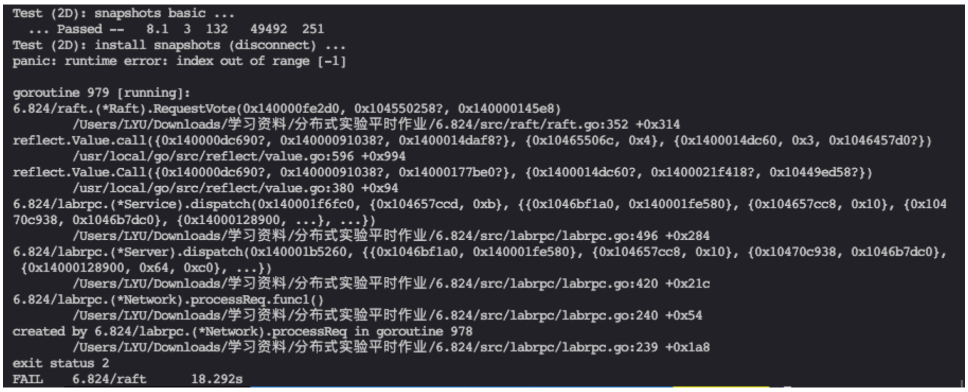
```
            LastIncludeIndex: rf.lastIncludeIndex,
            LastIncludeTerm:  rf.lastIncludeTerm,
            Data:             rf.snapshotCmd,
          }
          installSnapshotReply := &InstallSnapshotReply{}
          go rf.sendInstallSnapshot(i, installSnapshotReq, installSnapshotReply)
          continue
        }
        for rf.nextIndex[i] > rf.lastIncludeIndex {
          appendEntriesArgs.PrevLogIndex = rf.nextIndex[i]-1
          if appendEntriesArgs.PrevLogIndex >= len(rf.logs)+rf.lastIncludeIndex+1 {
            rf.nextIndex[i]--
            continue
          }
          if appendEntriesArgs.PrevLogIndex == rf.lastIncludeIndex {
            appendEntriesArgs.PrevLogTerm = rf.lastIncludeTerm
          } else {
            appendEntriesArgs.PrevLogTerm = rf.logs[appendEntriesArgs.PrevLogIndex-
rf.lastIncludeIndex-1].Term
          }
          break
        }
        if rf.nextIndex[i] < len(rf.logs)+rf.lastIncludeIndex+1 {
          appendEntriesArgs.Entries = make([]LogEntry,appendEntriesArgs.LogIndex+1-
rf.nextIndex[i])
          copy(appendEntriesArgs.Entries, rf.logs[rf.nextIndex[i]-
rf.lastIncludeIndex-1:appendEntriesArgs.LogIndex-rf.lastIncludeIndex])
        }

        appendEntriesReply := new(AppendEntriesReply)
        go rf.sendAppendEntries(i, appendEntriesArgs, appendEntriesReply, &appendNum)
      }
    }
```

这样的修改是为了处理日志压缩后的一致性问题。在日志压缩后，一些旧的日志条目可能被删除，而新加入的节点或落后的节点可能需要这些已压缩的日志来更新自己的状态。在这种情况下，通过发送快照（包含状态机的完整状态），而不是单个日志条目，可以更有效地同步落后的节点。

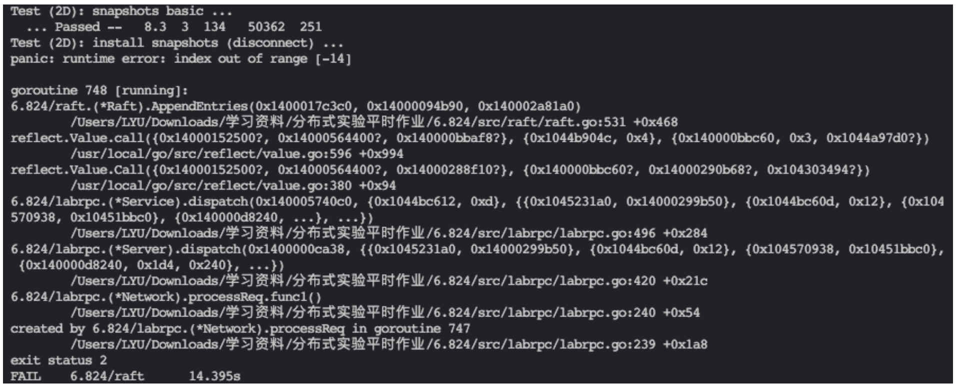
6 2D测试情况

6.1 Debug

初次测试时出现了一些bug，例如下面这几种情况数组越界的情况。这是由于我在为日志压缩修改之前的函数逻辑时，并没有考虑到全部情况（上文中已提及全部情况）。下面我将就Debug的过程展开一些阐述。



上图的bug是我起初没有考虑到要对选举限制进行调整，所以导致了数组越界。

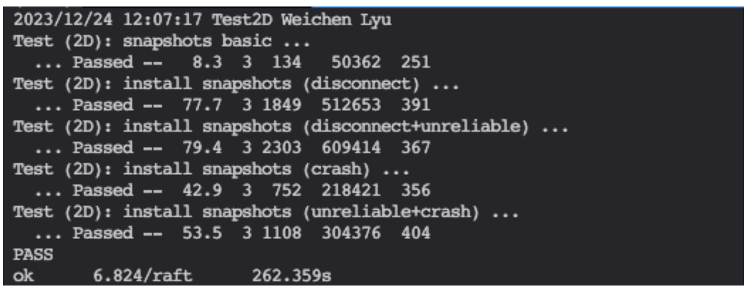


这里的bug是我在AppenEntries函数中没有考虑到请求过期的的第二种情况，即 `args.PrevLogIndex < rf.lastIncludeIndex` 而导致的数组越界.

测试过程中还出现了另一bug，如下图所示，但是这种情况的出现我一直没有找到是什么原因造成的，最终重启程序就解决了。

6.2 测试情况

测试样例全部通过。



Lab3B

1 SaveSnapshot实现

该函数的目的是为了创建并保存kv服务器的状态快照。实现逻辑如下：

1. 该函数接受一个参数`lastIndex`，这个参数表示要在快照中包含的最后一个日志条目的索引。
2. 使用`labgob.NewEncoder(writer)`创建一个新的编码器，这个编码器将用于将kv服务器的状态编码为字节序列。
3. 将服务器的键/值对（`kv.kv`）编码并且将跟踪客户端最后一个任务索引的数据（`kv.clientLastTaskIndex`）编码。
4. 如果上述两个编码操作都成功，即没有返回错误，那么`kv.rf.Snapshot(lastIndex, writer.Bytes())`会被调用。这个调用实际上是将编码后的状态以及与之关联的最后一个日志条目的索引传递给Raft实例，以便创建和保存快照。

```
func (kv *KVServer) SaveSnapshot(lastIndex int) {
  /*
  保存快照
  */

  writer := new(bytes.Buffer)
  encoder := labgob.NewEncoder(writer)
  if encoder.Encode(kv.kv) == nil &&
    encoder.Encode(kv.clientLastTaskIndex) == nil {
    kv.rf.Snapshot(lastIndex, writer.Bytes())
  }
}
```

2 ReadSnapshot实现

函数的主要作用是从保存的快照中恢复kv服务器的状态。实现逻辑如下：

1. 函数首先检查传入的`data`（代表快照数据）是否为`nil`或长度小于 1。如果是，函数直接返回，不执行任何操作。这是一个基本的安全检查，确保不会尝试解码无效或空的快照数据。
2. 使用`labgob.NewDecoder(bytes.NewBuffer(data))`创建一个新的解码器。这个解码器将用于将字节序列（即快照数据）解码回服务器的状态。
3. 使用`decoder.Decode(&kvMap)`将快照中的键值对数据解码到`kvMap`变量中。
4. 使用`decoder.Decode(&clientLastTaskIndex)`将快照中的客户端最后一个任务索引数据解码到`clientLastTaskIndex`变量中。
5. 如果上述解码操作都成功（即没有返回错误），则用解码出的数据更新服务器的状态。

```
func (kv *KVServer) ReadSnapshot(data []byte) {
  /*
```

```
  读取快照
  */

  if data == nil || len(data) < 1 {
    return
  }
  decoder := labgob.NewDecoder(bytes.NewBuffer(data))
  var kvMap map[string]string
  var clientLastTaskIndex map[ClientId]RequestIndex
  if decoder.Decode(&kvMap) == nil &&
    decoder.Decode(&clientLastTaskIndex) == nil {
    kv.kv = kvMap
    kv.clientLastTaskIndex = clientLastTaskIndex
  }
}
```

3 StartKVServer方法修改

需要在服务器启动时从持久化存储中加载并应用之前保存的快照。这确保了即使在服务器重启或崩溃后，它也能迅速恢复到之前的状态，而无需从头重新处理整个日志，从而提高启动效率并维持数据的一致性。

```
kv.ReadSnapshot(kv.persister.ReadSnapshot())
```

4 ApplyMsgLoop方法修改

在 Lab 3B 中，目标是优化kv服务器，使其能够与 Raft 的快照功能协同工作。这主要涉及两个方面：创建快照以节省日志空间和加快重启时间，以及在服务器重启时从快照中恢复状态。针对这个目标，需要对 `ApplyMsgLoop` 函数进行修改。

在修改之前，`ApplyMsgLoop` 函数主要负责处理来自 Raft 的普通命令消息。当 `applyMsg.CommandValid` 为真时，函数会解析并执行命令（例如 Put 或 Append），更新键值存储和Client最后请求索引（`kv.clientLastTaskIndex`）。

修改后的 `ApplyMsgLoop` 函数需要增加对快照消息的处理：

1. 检查是否需要Snapshot，当Raft状态的大小接近阈值,要求Raft进行Snapshot
2. 对 `applyMsg.SnapshotValid` 进行检查。这个字段为真时，表示接收到的是一个快照消息，而不是普通的命令。
3. 使用 `kv.rf.CondInstallSnapshot` 方法来判断是否应该安装这个快照。
4. 如果决定接受快照，通过调用 `kv.ReadSnapshot(applyMsg.Snapshot)` 来读取并应用快照数据。这一步骤涉及解析快照内容并更新服务器状态，包括键值存储和Client请求索引。

修改如下：

```
func (kv *KVServer) ApplyMsgLoop() {
  // 只要节点没有终止就会一直接收信息
  for kv.killed() == false {
    applyMsg := <-kv.applyCh
    kv.mu.Lock()
    if applyMsg.CommandValid { // 如果是有效的命令
```

```
        /*
        其他代码
        */
         // 检查是否需要Snapshot
        if kv.maxraftstate != -1 && float64(kv.persister.RaftStateSize()) >
float64(kv.maxraftstate)*serverSnapshotStatePercent {
          kv.SaveSnapshot(applyMsg.CommandIndex)
        }
      } else if applyMsg.SnapshotValid {
        if kv.rf.CondInstallSnapshot(applyMsg.SnapshotTerm, applyMsg.SnapshotIndex,
applyMsg.Snapshot) {
          kv.ReadSnapshot(applyMsg.Snapshot)
        }
      }
      kv.mu.Unlock()
    }
}
```

5 Lab3BDebug与测试

5.1 Debug过程

1. 在实现 `SaveSnapshot` 时，我面临了确保快照数据一致性的问题。我发现在编码状态之前未能正确地锁定状态，这可能导致读取到不一致的状态。为了解决这个问题，我增加了适当的锁定机制，确保在捕捉快照时，服务器状态不会被并发请求所修改。

2. 在处理快照消息时新的快照可能会覆盖正在进行的操作的状态。这主要是因为我在 `ApplyMsgLoop` 中未能正确处理现有状态，我增加了额外的逻辑来在安装新快照前保护当前状态，从而防止错误覆盖。

5.2 Lab3B测试

Lab3B通过情况如下：

```
2024/01/13 16:42:29 Test3B Weichen Lyu
Test: InstallSnapshot RPC (3B) ...
  ... Passed --   2.4  3   278   63
Test: snapshot size is reasonable (3B) ...
  ... Passed --   0.6  3  2429  800
Test: ops complete fast enough (3B) ...
  ... Passed --   0.7  3  3038    0
Test: restarts, snapshots, one client (3B) ...
info: linearizability check timed out, assuming history is ok
  ... Passed --  26.0  5 413932 82560
Test: restarts, snapshots, many clients (3B) ...
  ... Passed --  20.0  5 398338 78577
Test: unreliable net, snapshots, many clients (3B) ...
  ... Passed --  16.1  5 11486 1599
Test: unreliable net, restarts, snapshots, many clients (3B) ...
  ... Passed --  20.7  5 12432 1610
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
  ... Passed --  29.4  5  7963  817
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (3B) ...
  ... Passed --  28.8  7 37254 3191
PASS
ok      6.824/kvraft    145.274s
```

总结与心得

通过完成MIT 6.824 Lab3，我深入探索了分布式系统的核心概念，尤其是在使用Raft共识算法构建容错键/值服务方面。在Lab 3A中，我实现了一个基本的键/值存储系统，不仅处理了Put、Append和Get操作，还保证了系统的线性一致性。Lab 3B的挑战在于加入快照功能，优化了日志管理和重启效率。我在客户端和服务器端的实现中采用了清晰的结构和策略，确保了系统的高效性和可靠性。这个实验不仅加深了我对分布式系统的理解，也锻炼了我的编程和问题解决能力。

本次作业最困扰我的就是Lab2D了，Lab2D的实现要比整个Lab2都要难很多，感谢中文互联网上的很多博主，他们的代码深深启发了我。Lab3A测试中，由于我的Raft实现性能较差，导致3A一直有报错，所以我一度想放弃，最终汲取了许多优秀开源代码的精华，我得以重构Raft代码，顺利通过测试。我在做完这个作业后发现自己需要学的东西还有很多，但这次作业确实在某些方面上启发了我，改变了我思考问题的方式。