容错的分布式键值存储系统

刘芳新

中山大学 智能科学与技术 21312482

2023 年 12 月 25 日

学号: 21312482
课程: 分布式计算实验
日期：2023 年 12 月 25 日
Email:liufx7@mail2.sysu.edu.cn

目录

1 lab3-A kv server **3**
1.1 任务分析 . . . 3
1.2 common 功能设计 . . . 3
1.3 client 功能设计 . . . 3
1.3.1 完善 Clerk 结构体 . . . 3
1.3.2 完善 MakeClerk 函数 . . . 3
1.3.3 完善 Get 函数 . . . 4
1.3.4 完善 PutAppend 函数 . . . 4
1.4 server 功能设计 . . . 4
1.4.1 完善 Op 结构体 . . . 4
1.4.2 完善 KVServer 结构体 . . . 4
1.4.3 完善 Get 函数 . . . 5
1.4.4 完善 PutAppend 函数 . . . 5
1.4.5 设计 executeThread 函数 . . . 5
1.4.6 完善 StartKVServer 函数 . . . 6
1.5 代码实现 . . . 6
1.6 测试结果 . . . 14
1.7 个人总结 . . . 15

2 lab3-B snapshot **15**
2.1 lab2D 任务分析 . . . 15
2.2 lab3B 任务分析 . . . 15
2.3 lab2D 功能设计 . . . 16
2.3.1 调整 Raft 结构体 . . . 16
2.3.2 实现 Snapshot 函数 . . . 16
2.3.3 实现 InstallSnapshot 功能 . . . 16
2.3.4 调整 Start 函数 . . . 17
2.4 lab3B 功能设计 . . . 17
2.4.1 修改 executeThread 函数 . . . 17
2.4.2 修改 StartKVServer 函数 . . . 18
2.5 lab2D 代码实现 . . . 18
2.6 lab3B 代码实现 . . . 26
2.7 测试结果 . . . 27
2.8 个人总结 . . . 28

3 心得体会 **28**

4 参考资料 **29**

1 lab3-A kv server

1.1 任务分析

Lab 3A 的任务是在之前实现的 Raft 算法的基础上构建一个基于 Raft 的键值存储服务器（KV server）。这样的服务器可以提供分布式的键值存储服务，并通过 Raft 算法确保数据的一致性。以下是对 Lab 3A 的任务的分析：

- KV Server：需要实现一个简单的键值存储服务器，支持基本的键值对操作，如读取（Get）、写入（Put）、修改（Append）等。这个服务器将在多个节点上运行，通过 Raft 算法维护数据的一致性。
- Raft 集成：KV 服务器将使用之前实现的 Raft 算法来确保在分布式环境下数据的一致性。每个 KV 服务器节点都是 Raft 节点，参与领导者选举、日志复制、一致性维护等 Raft 算法的操作。
- 客户端请求处理：KV 服务器需要能够处理来自客户端的键值操作请求。这可能涉及到网络通信、请求解析、操作执行等。
- 状态机和数据存储：KV 服务器作为 Raft 的状态机，需要维护键值对的状态。你需要实现数据的存储和管理，确保数据在节点之间正确地复制和同步。

1.2 common 功能设计

公共部分需要完善 PutAppendArgs 和 GetArgs 结构体，我在这两个结构体中额外添加了 Id 属性，用来标识这个独一无二的请求，避免重复执行。

1.3 client 功能设计

客户端需要向服务器发送请求。

1.3.1 完善 Clerk 结构体

在这其中，我添加了三个字段：leader、peerNum 和 name。

- leader 字段：用于存储搜索到的领导者的索引。这个字段对于了解当前网络中的领导者是非常有用的。
- peerNum 字段：记录了服务器的数量。通过这个字段，我们可以知道整个系统中有多少个节点，这对于一致性算法的正常运行是必要的信息。
- name 字段：用于标识客户端。通过这个字段，我们可以清晰地辨识和区分不同的客户端，这在处理多个客户端请求时非常重要。

1.3.2 完善 MakeClerk 函数

MakeClerk 创建并返回一个 Clerk 客户端，当客户端刚刚启动时，由于还不知道领导者的位置，因此将 leader 字段设置为-1。这是一种合理的初始化，表示客户端在开始时对领导者的位置没有具体信息。

1.3.3 完善 Get 函数

客户端在发起 Get 请求时，首先需要检查当前是否已确认领导者。如果领导者为 -1，客户端需要通过轮询遍历的方式搜索新的领导者。这种搜索方式虽然直接简单，但由于 Raft 复制的数量通常不会很多，因此一次轮询的性能开销较小。

Get 请求的 RPC 参数应包含 Key 和一个 Id。

在客户端和服务端之间的交互中可能会遇到一些问题，考虑了以下几种情况：

- Consensus Error（共识错误）：表示服务端对该请求的共识过程发生错误，无法完成共识，因此无法执行该请求。可能原因是请求的服务端不再是合法的 leader，或者是 Raft 集群出现问题。客户端的处理措施是重新进行搜索，寻找另一个 leader。这种方式直观地解决了第一种情况，如果是第二种情况，客户端会重新搜索到相同的 leader 并再次尝试请求，直到 Raft 集群恢复。
- Not Leader（非领导者错误）：表示一个非 leader 的服务端接收到了客户端的请求。客户端需要重新向下一个服务器尝试发起请求。
- RPC 请求无法到达服务器：表示网络出现问题，无法和服务器建立连接进行 RPC 请求。客户端应该向下一个服务器尝试发起请求。

1.3.4 完善 PutAppend 函数

原理和 Get 函数同理，唯一的区别就是返回值的差别。PutAppend 函数的返回值中没有 value 值。

1.4 server 功能设计

服务器端需要接受请求，并通过底层 Raft 来保持 log 的一致性，操作存储在内存中。

1.4.1 完善 Op 结构体

这个 Op 结构体用于给我们服务端的简单数据库进行执行请求用的, 同时，这个结构体也被用作命令，发送到 Raft 中进行共识。

1.4.2 完善 KVServer 结构体

KVServer 表示键值储存服务器，我添加了以下字段：

- CommandLog：是一个映射型的数据结构，专门用于存储已经执行过的请求的 Id 值。其设计采用了 map 这个数据类型，以请求的 Id 为 key。这个变量的主要作用是提供一种快速查找的机制，用于判断某个指令 Id 是否已经有执行记录。
- CommandLogQueue：则是一个队列，用于追踪记录最近的几条请求。在进行快照时，需要将最近的请求记录保存在这个队列中。这样设计的目的是确保在服务端重启后，系统能够迅速恢复并回复那些最近可能发生的重复请求，从而保持数据的一致性。
- cond：一个条件变量，它被设计用于后续的同步执行。
- Data：一个代表简单数据库的变量，用于存储指令的执行结果。

1.4.3 完善 Get 函数

服务端在接收到客户端的请求后，首先通过 kv.CommandLog 判断该请求是否为重复请求，即是否已经有记录。如果是重复请求，则直接返回上次执行的结果。

对于非重复请求，服务端将 RPC 参数封装为 Op 结构体，作为命令传递给 Raft 模块进行共识。调用 Start 函数开始对该请求进行 Raft 共识。在这个过程中，需要考虑几种情况：

- 如果 Start 函数返回本节点非 leader，服务端返回错误信息：“Not leader”。
- 如果 Start 函数返回本节点为 leader，服务端将开启一个计时器协程，并阻塞等待请求的共识完成并成功执行。如何判断是否共识成功并执行呢？每当有一条请求共识成功并进入 applyCh 中，由 executeThread 线程进行执行，这时会唤醒等待的线程。被唤醒的线程判断是否为自己负责的请求，如果不是，则继续阻塞。这里可能有两种情况：（1）共识成功并成功执行；（2）本节点由于网络分区已经是过时的 leader，始终无法完成请求的共识成功。如果计时器超时后仍未完成，则返回错误信息：“Consensus error”。
- 如果完成 Raft 共识后，服务端等待该请求的执行。通过开启一个协程 executeThread 来监控 applyCh 管道中的请求，并执行管道中的请求。每有一条请求被执行，Get 函数都会查看是否为本请求被成功执行。若成功执行，则查看执行结果并返回。

在 KVServer 结构体中，设计了一个 cond 条件变量用于协调 executeThread 协程和 Get 协程。Get 协程在发现本请求成功共识后，会阻塞等待该请求是否被及时执行。而 executeThread 协程在每次执行完一条指令后，都会唤醒所有等待本请求执行结束的协程，使它们检查是否自己的请求被执行成功，若成功则返回结果给客户端。

1.4.4 完善 PutAppend 函数

逻辑和 Get 函数一模一样，此处不赘述。

1.4.5 设计 executeThread 函数

executeThread 函数是 KV 服务器中一个关键的处理函数，它处理 Raft 模块传递过来的 ApplyMsg。以下是 executeThread 函数的详细原理：

1. executeThread 是一个无限循环的 goroutine，它持续监听 applyCh 通道，该通道由 Raft 模块用于传递已经提交的日志条目（命令）。
2. 当有新的日志条目被提交时，executeThread 会从 applyCh 中接收到一个 ApplyMsg 对象。
3. 如果 ApplyMsg 中的 SnapshotValid 字段为 true，说明这是一个快照（snapshot），executeThread 会解码快照中的数据，包括键值对的状态和已经执行的命令日志。然后将这些数据恢复到 KV 服务器的状态中。
4. 如果 ApplyMsg 中的 CommandValid 字段为 true，说明这是一个客户端的命令。executeThread 提取出命令中的操作类型（Get、Put、Append）和相应的参数，然后执行相应的操作。
    - 对于 Get 操作，executeThread 从服务器状态中获取相应的值，将其放入命令结果中，并将结果保存到命令日志中，供后续查询使用。

- 对于 Put 操作，executeThread 更新服务器状态中的键值对，并将结果保存到命令日志中。
- 对于 Append 操作，executeThread 将指定键的值追加指定内容，并将结果保存到命令日志中。

5. 执行完命令后，executeThread 会通过条件变量 cond 发送信号通知可能在等待的线程，以便它们能够获取最新的状态或者获取相应的命令结果。

6. 这个循环不断地处理新的提交，确保服务器状态与 Raft 模块的日志保持一致。由于 applyCh 中的消息是有序的，因此 KV 服务器能够按照提交的顺序正确执行每个命令。

1.4.6 完善 StartKVServer 函数

在这里，我添加了我之前额外引入的一些变量和函数。

kv.CommandLog、kv.CommandLogQueue、kv.cond、kv.Data 都进行了初始化，并开启一个线程执行 executeThread 函数，以实时监控 applyCh 管道并执行其中的请求。

1.5 代码实现

- 公共部分

```go
// Put or Append
// PutAppendArgs 包含 Put 或 Append 请求的参数
type PutAppendArgs struct {
    Key   string
    Value string
    Op    string // "Put" or "Append"
    // You'll have to add definitions here.
    // Field names must start with capital letters,
    // otherwise RPC will break.
    Id int64 // 请求的唯一标识符
}
```

图 1: PutAppendArgs 结构体

```go
// GetArgs 包含 Get 请求的参数
type GetArgs struct {
    Key string
    // You'll have to add definitions here.
    Id int64 // 请求的唯一标识符
}
```

图 2: GetArgs 结构体

- 客户端部分

```go
// Clerk 结构体用于向 KV 服务器发起客户端请求。
type Clerk struct {
    servers []*labrpc.ClientEnd // KV 服务器的 RPC 客户端数组
    leader  int                 // 记录当前 KV 服务器的领导者
    peerNum int                 // KV 服务器的数量
    name    string              // 客户端的名称
}
```

图 3: Clerk 结构体

```go
// MakeClerk 创建并返回一个 Clerk 客户端。
func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
    ck := new(Clerk)
    ck.servers = servers
    // You'll have to add code here.
    ck.leader = -1
    ck.peerNum = len(servers)
    ck.name = strconv.FormatInt(nrand(), 10)
    return ck
}
```

图 4: MakeClerk 函数

```go
// Get 从 KV 服务器获取指定 key 对应的值。
func (ck *Clerk) Get(key string) string {
    // You will have to modify this function.
    if ck.leader == -1 {
        ck.leader = 0
    }
    args := &GetArgs{key, nrand()}
    reply := &GetReply{}
    for {
        // 调用 KV 服务器的 Get 方法
        ok := ck.servers[ck.leader].Call("KVServer.Get", args, reply)
        // 处理 Consensus error
        if reply.Err == "Consensus error" {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 处理 Not leader 错误
        if reply.Err == "Not leader" {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 处理调用失败情况
        if !ok {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 成功获取值，返回
        if ok && reply.Err == "" {
            return reply.Value
        }
        reply.Err = ""
    }
}
```

图 5: Get 函数

```go
//
// PutAppend 向 KV 服务器发送 Put 或 Append 请求。
func (ck *Clerk) PutAppend(key string, value string, op string) {
    // You will have to modify this function.
    if ck.leader == -1 {
        ck.leader = 0
    }
    args := &PutAppendArgs{key, value, op, nrand()}
    reply := &PutAppendReply{}
    for {
        // 调用 KV 服务器的 PutAppend 方法
        ok := ck.servers[ck.leader].Call("KVServer.PutAppend", args, reply)
        // 处理 Consensus error
        if reply.Err == "Consensus error" {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 处理 Not leader 错误
        if reply.Err == "Not leader" {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 处理调用失败情况
        if !ok {
            ck.leader = (ck.leader + 1) % ck.peerNum
        }
        // 成功发送请求，退出循环
        if ok && reply.Err == "" {
            break
        }
        reply.Err = ""
    }
}
```

图 6: PutAppend 函数

- 服务器端部分

```go
// Op 定义了 KV 服务的操作类型
type Op struct {
    // Your definitions here.
    // Field names must start with capital letters,
    // otherwise RPC will break.
    Key       string
    Value     string
    Operation string
    Id        int64
}
```

图 7: Op 结构体

```go
// KVServer 表示键值存储服务器
type KVServer struct {
    mu      sync.Mutex
    me      int
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg
    dead    int32 // set by Kill()  1 表示服务器已终止

    maxraftstate int // 达到此大小时进行快照

    // Your definitions here.
    CommandLog      map[int64]string // 记录每个客户端请求的结果
    CommandLogQueue List // 记录每个客户端请求的 Id，用于控制队列长度
    cond            *sync.Cond
    Data            map[string]string // 存储键值对
}
```

图 8: KVServer 结构体

```go
// Get 处理客户端的 Get 请求
func (kv *KVServer) Get(args *GetArgs, reply *GetReply) {
    // Your code here.
    kv.cond.L.Lock()
    result, ok := kv.CommandLog[args.Id]
    if ok {
        reply.Value = result
        kv.cond.L.Unlock()
        return
    }
    command := Op{Key: args.Key, Operation: "Get", Id: args.Id}
    kv.cond.L.Unlock()
    _, term, ok := kv.rf.Start(command)
    if !ok {
        reply.Err = "Not leader"
        return
    }
    var timerController bool = true
    var timeoutTime = 0
    kv.cond.L.Lock()
    for {
        // 判断是否请求被完成
        result, ok = kv.CommandLog[args.Id]
        if ok {
            reply.Value = result
            timerController = false
            kv.cond.L.Unlock()
            return
        }
        nowTerm, isLeder := kv.rf.GetState()
        // 请求并未完成，但是发现本raft节点已经不是leader节点，或者是等待请求结果已经超时3次了，
        // 或者发现本节点的状态已经和当时发起共识时不一样了则此次动作作废，
        // 以上几种情况均是失败的，直接返回共识错误给客户端
        if nowTerm != term || !isLeder || timeoutTime == 3 {
            reply.Err = "Consensus error"
            timerController = false
            kv.cond.L.Unlock()
            return
        }
        go func() {
            // 计时器线程，每经过100ms都没有完成请求的执行就由计时器来自动唤醒，查看是什么情况，
            // 若超过3次自动唤醒都没有完成，则直接终止，对应上面
            time.Sleep(time.Duration(100) * time.Millisecond)
            kv.cond.L.Lock()
            if timerController {
                timeoutTime++
                kv.cond.L.Unlock()
                kv.cond.Broadcast()
            } else {
                kv.cond.L.Unlock()
                return
            }
        }()
        kv.cond.Wait()
    }
}
```

图 9: Get 函数

```go
// PutAppend 处理客户端的 Put 或 Append 请求
func (kv *KVServer) PutAppend(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    kv.cond.L.Lock()
    _, ok := kv.CommandLog[args.Id]
    if ok {
        kv.cond.L.Unlock()
        return
    }
    command := Op{Key: args.Key, Operation: args.Op, Value: args.Value, Id: args.Id}
    kv.cond.L.Unlock()
    _, term, ok := kv.rf.Start(command)
    if !ok {
        reply.Err = "Not leader"
        return
    }
    var timerController bool = true
    var timeoutTime = 0
    kv.cond.L.Lock()
    for {
        _, ok = kv.CommandLog[args.Id]
        if ok {
            timerController = false
            kv.cond.L.Unlock()
            return
        }
        nowTerm, isLeder := kv.rf.GetState()
        if nowTerm != term || !isLeder || timeoutTime == 3 {
            reply.Err = "Consensus error"
            timerController = false
            kv.cond.L.Unlock()
            return
        }
        go func() {
            time.Sleep(time.Duration(100) * time.Millisecond)
            kv.cond.L.Lock()
            if timerController {
                timeoutTime++
                kv.cond.L.Unlock()
                kv.cond.Broadcast()
            } else {
                kv.cond.L.Unlock()
                return
            }
        }()
        kv.cond.Wait()
    }
}
```

图 10: PutAppend 函数

```go
// executeThread 处理 Raft 模块传递过来的 ApplyMsg
func (kv *KVServer) executeThread() {
    for {
        var command = <-kv.applyCh
        kv.mu.Lock()

        if command.CommandValid {
            var msg = command.Command.(Op)
            _, ok1 := kv.CommandLog[msg.Id]

            // 如果请求 Id 已经存在于 CommandLog 中，说明该请求已经被执行过
            if !ok1 {
                if msg.Operation == "Get" {
                    result, ok2 := kv.Data[msg.Key]
                    if ok2 {
                        msg.Value = result
                    } else {
                        msg.Value = ""
                    }
                }
                if msg.Operation == "Put" {
                    kv.Data[msg.Key] = msg.Value
                }
                if msg.Operation == "Append" {
                    kv.Data[msg.Key] += msg.Value
                }

                // 控制 CommandLogQueue 的长度
                if kv.CommandLogQueue.Len >= 5 {
                    kv.CommandLogQueue.Pop()
                }
                kv.CommandLogQueue.Push(msg.Id)
                kv.CommandLog[msg.Id] = msg.Value
            }
            kv.mu.Unlock()
            kv.cond.Broadcast()
        }
    }
}
```

图 11: executeThread 函数

```go
// StartKVServer 启动 KV 服务器
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{}) // 在 Go 的 RPC 库中注册 Op 结构

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

    // You may need initialization code here.

    kv.applyCh = make(chan raft.ApplyMsg)
    kv.rf = raft.Make(servers, me, persister, kv.applyCh)
    kv.CommandLog = make(map[int64]string)
    kv.cond = sync.NewCond(&kv.mu)
    kv.Data = make(map[string]string)
    kv.CommandLogQueue = List{}

    go kv.executeThread() // 启动执行线程
    // You may need initialization code here.

    return kv
}
```

图 12: StartKVServer 函数

1.6 测试结果

```
dbs@ubuntu:~/go-workplace/6.824/src/kvraft$ go test -run 3A
Test: one client (3A) ...
  ... Passed --  15.0  5 22311 3068
Test: ops complete fast enough (3A) ...
  ... Passed --   1.6  3  8919    0
Test: many clients (3A) ...
  ... Passed --  15.1  5 30833 3553
Test: unreliable net, many clients (3A) ...
  ... Passed --  15.6  5 13495 1344
Test: concurrent append to same key, unreliable (3A) ...
  ... Passed --   0.9  3   306   52
Test: progress in majority (3A) ...
  ... Passed --   0.3  5    48    2
Test: no progress in minority (3A) ...
  ... Passed --   1.0  5   182    3
Test: completion after heal (3A) ...
  ... Passed --   1.0  5    67    3
Test: partitions, one client (3A) ...
  ... Passed --  22.6  5 84756 3666
Test: partitions, many clients (3A) ...
  ... Passed --  22.4  5 81353 3265
Test: restarts, one client (3A) ...
  ... Passed --  19.7  5 46178 3489
Test: restarts, many clients (3A) ...
  ... Passed --  19.2  5 66865 3854
Test: unreliable net, restarts, many clients (3A) ...
  ... Passed --  21.2  5 14432 1302
Test: restarts, partitions, many clients (3A) ...
  ... Passed --  26.3  5 169377 3491
Test: unreliable net, restarts, partitions, many clients (3A) ...
  ... Passed --  26.8  5 13617  909
Test: unreliable net, restarts, partitions, random keys, many clients (3A) ...
  ... Passed --  28.9  7 44677 2213
PASS
ok      6.824/kvraft    237.908s
```

图 13: 3A 运行结果

1.7 个人总结

Lab3A 的主要目标是实现基于 Raft 的键值（KV）服务器。在这个系统中，客户端通过向服务器发送请求，服务器通过底层的 Raft 协议来维护日志的一致性，并将操作存储在内存中。Lab3A 在 Raft 论文中主要对应于第 8 节, 整体实现并不复杂。该实验的目的就是在 Lab2 中实现的 Raft 服务层之上建立一个服务，并与客户端进行交互。

2 lab3-B snapshot

2.1 lab2D 任务分析

Lab 2D 的任务是在之前实现的 Raft 算法中加入快照机制。快照机制是为了减小日志的大小，提高系统性能。

- 快照机制概述：快照机制是一种定期捕获整个系统状态的方法，以便在以后的某个时间点可以从这个快照中进行恢复。在 Raft 中，快照用于替代一系列的日志条目，从而减小存储的数据量。当日志变得很长时，通过快照机制截断部分旧的日志，只保留最新的快照以及之后的日志。
- 触发快照的条件：快照机制通常不是无条件地触发的，而是在满足一些条件时才执行。可能的触发条件包括：
  当日志达到一定的长度或条目数量时，执行快照。
  当系统空闲时，执行快照。
  当系统中有足够多的节点已经应用了相同的日志时，执行快照。
- 实现快照的逻辑：在 Lab 2D 中，需要实现将整个系统状态转化为一个快照，并将这个快照存储在持久化存储中。同时，还需要修改 Raft 的状态机，以支持从快照中进行恢复。
- 快照与日志的结合：在实现快照机制时，需要确保与之前实现的日志复制和一致性机制协同工作。快照可能会涉及到某一时刻的状态机快照，以及之后的一系列日志条目。当节点恢复时，需要能够根据快照和日志条目还原整个系统的状态。

2.2 lab3B 任务分析

Lab 3B 的任务是在 Lab 3A 的基础上为键值存储服务器（KV server）添加快照功能。快照机制将有助于减小系统状态的存储开销，特别是在处理大量的键值对时。以下是对 Lab 3B 的任务的分析：

- 快照机制的引入：在 Lab 3B 中，需要修改之前实现的 KV 服务器，以支持快照机制。这涉及到在 Raft 算法中触发和处理快照的逻辑。
- 快照触发条件：类似于 Lab 2D 中的 Raft 快照，需要确定何时触发 KV 服务器上的快照。触发条件可能包括：
  当系统中的键值对数量达到一定阈值时，执行快照。
  当系统空闲时，执行快照。
  当系统中有足够多的节点已经应用了相同的日志时，执行快照。

- 实现快照的逻辑：修改 Raft 算法和 KV 服务器的状态机，以支持将整个系统状态转化为一个快照，并将快照存储在持久化存储中。此外，需要考虑如何在节点恢复时正确加载和应用快照。
- 与日志的交互：确保快照机制与之前实现的 Raft 算法中的日志复制和一致性机制协同工作。快照可能会涉及到某一时刻的状态机快照，以及之后的一系列日志条目。在节点恢复时，需要能够根据快照和日志条目还原整个系统的状态。

2.3 lab2D 功能设计

2.3.1 调整 Raft 结构体

由于加入了快照机制，因此需要考虑 rf.log 被修剪的情况，需要记录 lastIncludedIndex 和 lastIncludedTerm 这两个参数。这两个变量指的是在经过快照后，被修剪的最后一个 log 的 index 和 term 值。在访问 rf.log 数组中的指定 index 的 log 时，需要使用这两个变量。例如，访问 index=n 的 log，通过 rf.log 数组来访问的话，就是 rf.log[n-rf.lastIncludedIndex] ，即为 index=n 的 log。因此，将这两个变量直接添加到 raft 结构体中，利用 rf.mu 这个互斥锁方便各个协程进行互斥使用。

2.3.2 实现 Snapshot 函数

该函数主要由节点自己调用，用于创建一个新的快照。

主要流程如下：检查要创建的快照是否已经过时，如果过时则直接返回。根据快照范围内 log 的 index，将节点自身的 log 数组进行修剪。创建好快照后，节点那些需要持久化的信息也会更新，例如 rf.log、rf.lastIncludedIndex、rf.lastIncludedTerm。然后需要将这些更新后的信息进行持久化保存。实验教程推荐使用 persister.SaveStateAndSnapshot 函数进行持久化保存快照和节点的信息。

注意：在 6.824 中，快照的间隔是每 10 条 command 进行一次快照。因此，在节点提交已经确认的指令到 applyCh 进行执行时，不能获取带有 rf.mu 这个互斥锁。因为在提交指令并将该指令发送到 applyCh 执行的同时，测试脚本会调用 Snapshot 函数进行快照。但是我设计的这个函数也需要获取 rf.mu 互斥锁，那么这个节点就会进入死锁状态：无法获取 rf.mu 互斥锁进行快照，另一边需要等快照结束才能继续提交指令并执行，以及后续动作。因此，需要注意在设计中解决这个死锁问题。

2.3.3 实现 InstallSnapshot 功能

当引入快照机制后，当领导者修剪日志后，在进行日志复制时，部分跟随者可能会缺少领导者已经通过快照修剪的日志。在这种情况下，领导者需要调用该 RPC 来将自身的快照发送给相应的跟随者，以解决这个问题。

需要注意的是，需要检查发送过来的快照是否是过时的，以避免旧的快照将本地新的快照覆盖，导致数据回滚。

同时，节点收到快照有两种可能性：

- 如果发送过来的 args.LastIncludedIndex 比本节点的 lastLogIndex 大，那么节点只需删除本地的日志，并通过 args 的数据更新本地信息。此外，节点的 lastLogIndex 也应当变更为 args.LastIncludedIndex，lastLogTerm 也应当变更为 args.LastIncludedTerm。
- 如果发送过来的 args.LastIncludedIndex 比本节点的 lastLogIndex 小，那么节点只需将包括 LastIncludedIndex 在内的所有日志修剪掉即可，无需更改 lastLogIndex。

实验教程中指出，当一个节点收到 InstallSnapshot 时，需要将该快照信息放到 applyMsg 中，然后再放到 applyCh 中。如果只是更新节点的本地信息而不将快照信息生成一个 applyMsg 并插入到 applyCh 中，测试将会出错。

因此，在生成 applyMsg 插入到 applyCh 时，需要注意上述提到的，插入到 applyCh 管道中时不能持有 rf.mu 互斥锁，以避免与测试脚本调用 snapshot 函数时发生死锁。

2.3.4 调整 Start 函数

在这里的调整主要针对那些日志落后的跟随者需要进行 InstallSnapshot 的情况。

需要注意的是，在 start 函数中存在一个循环，尝试发送 sendAppendEntries 给跟随者进行日志复制。在每次循环中都需要释放并重新获取锁。重新获取锁后，节点本身的状态可能发生了更新。在 2D 实验中，需要额外考虑的一个状态更新是通过快照修剪了 rf.log。在发送完 sendAppendEntries 后，重新获取锁之前，还需要检查 rf.lastIncludedIndex 是否发生了变化，否则对 rf.log 进行任何访问前都需要检查 rf.lastIncludedIndex 是否改变，以避免错误。

存在一种情况，当领导者尝试给跟随者进行日志复制以寻找匹配点时，进入下一次循环时，发现领导者已创建了快照，rf.log 经过修剪，nextIndex 已经小于 rf.lastIncludedIndex 即匹配点必然处于快照中，此时直接发送快照。

此外，在领导者给跟随者发送快照后，更新了跟随者的状态后，领导者本地对该跟随者的状态记录也需要更新，rf.matchIndex 和 rf.nextIndex 都需要更新。

在正常情况中，当领导者发送给跟随者，然后进行寻找日志复制的匹配点时，同样需要对 nextIndex 进行递减寻找。如果发现 nextIndex - 1 - rf.lastIncludedIndex == 0 且 rf.lastIncludedInex != 0，说明领导者本地的第一个日志也不是匹配点，领导者创建过快照对 rf.log 进行过修剪。在这种情况下，还需要查看快照中的最后一个日志是否匹配，若不匹配，则需要先发送快照来解决那些过于落后的日志，后续再发送本地有的日志给跟随者。

在 Start 函数中，当碰到 nextIndex - 1 - rf.lastIncludedIndex == 0 的情况，这种边缘情况，就需要考虑到快照对 rf.log 进行修剪的情况。

同时，在 Start 函数中，本地对那些已经提交的 msg 进行执行，并将 msg 插入到 applyCh 管道中时，务必不能持有 rf.mu 这个互斥锁，否则会导致死锁。

2.4 lab3B 功能设计

2.4.1 修改 executeThread 函数

这个函数启动一个线程，实时监控 applyCh 中已完成共识的请求，并立即执行这些请求。首先，通过 command.SnapshotValid 判断请求是否是执行快照的 applyMsg 的特殊消息。如果是，解码其中的信息并进行恢复。

若不是执行快照的消息，则通过类型断言将 interface 类型转换为 Op 类型，获取 command 中的信息并执行其中的操作。在每次请求执行结束后，执行 kv.cond.Broadcast() 进行广播，唤醒等待请求执行结果的线程，使它们返回执行结果给客户端。

此外，由于检测重复请求的机制，需要将已完成请求的 ID 插入队列中进行保存。在实验中，设计的队列长度为 5，保存最近执行的 5 条请求的 ID。当进行保存数据时进行快照时，将队列指定的 5 个请求和数据库的状态保存为快照。

Lab3B 要求实现快照机制，触发快照的条件是当 Raft 传递给 persister.SaveRaftState() 函数进行持久化状态的数据大小超过 maxraftstate 时。但是，快照动作由 kvserver 发起，但该线程无法访问 kv.rf.persister 对象的任何信息，因为该对象是小写开头，是私有属性。因此，需要在 Raft 中设计两个接口供 kvserver 调用 persister.RaftStateSize()，以获取 Raft 当前的持久化数据大小。

设计的接口名为 GetStateSizeAndSnapshotIndex()，该函数将返回 Raft 当前持久保存的数据大小以及当前快照中的日志索引值，即 lastIncludedIndex。

之所以需要 lastIncludedIndex，是因为存在一种特殊情况，即一个节点在断连很长时间后，再次连接后，领导者会向这个跟随者发送大量的日志。此时，跟随者进行持久化时，数据量大小必然超过 maxraftstate。然而，该节点的 kvserver 在执行第一个请求后发现数据量太大就会立即进行快照，即使仅执行了一条请求，那么快照仅会减少一个单位的 rf.log 大小，数据量仍然大于 maxraftstate。后续每执行一条命令都会触发一次快照，这就会出现问题。因此，需要确保进行快照时，距离上次快照的状态至少相差 20 个日志，即至少执行 20 条请求，才能再次执行快照。

2.4.2 修改 StartKVServer 函数

一个 KVserver 在启动时必须检查是否存在快照，如果有快照，则需要恢复快照中的状态。因此，我们需要获取 Raft 中的快照数据信息。我在 Raft 里面设计了 GetSnapshot() 接口以供 KVserver 调用以获取快照的数据大小，如果有快照，则数据大小不为 0。

2.5 lab2D 代码实现

- 调整 Raft 结构体

```go
// A Go object implementing a single Raft peer.
type Raft struct {
    mu        sync.Mutex          // Lock to protect shared access to this peer's state 用于保护对节点状态的共享访问的锁
    peers     []*labrpc.ClientEnd // RPC end points of all peers                        所有节点的 RPC 终端点
    persister *Persister          // Object to hold this peer's persisted state         用于保存节点持久状态的对象
    me        int                 // this peer's index into peers[] //                  节点在 peers[] 中的索引
    dead      int32               // set by Kill()                                      由 Kill() 设置

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

    // persistent state
    peerNum           int   // 节点数量
    currentTerm       int   // 当前任期
    voteFor           int   // 投票给哪个节点
    log               []LogEntry // 日志条目
    // 被经过快照后，被修剪的最后一个log的index和term值
    lastIncludedIndex int
    lastIncludedTerm  int
    // volatile state
    commitIndex int // 已经被提交的日志的最大索引
    lastApplied int // 已经被应用到状态机的最大索引
    state       string // 节点状态
    lastLogIndex int // 最后一个日志条目的索引
    lastLogTerm  int // 最后一个日志条目的任期

    // 将每个已提交的命令发送到 applyCh
    applyCh chan ApplyMsg

    // Candidate 使用条件变量同步选举
    mesMutex    sync.Mutex // 用于锁定变量 ElectionStatus
    messageCond *sync.Cond // 条件变量

    // ElectionStatus == 1 -> 开始选举
    // ElectionStatus == -1 -> 保持不动
    // ElectionStatus == 2 -> 成为领导者
    // ElectionStatus == 3 -> 选举超时
    ElectionStatus int

    // volatile state on leaders
    nextIndex  []int // 用于每个节点的下一个日志条目的索引
    matchIndex []int // 用于每个节点已匹配的最高日志条目的索引
}
```

图 14: Raft 结构体

- 实现 Snapshot 函数

```go
// Snapshot 用于生成 Raft 节点的快照。
// 参数 index 表示快照截断的日志条目索引，
// 参数 snapshot 为快照数据，即应用状态的快照。
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    // Your code here (2D).
    rf.mu.Lock()
    // 如果指定的索引小于等于已包含的最后一条日志的索引，
    // 则无需生成快照，直接释放锁并返回
    if index <= rf.lastIncludedIndex {
        rf.mu.Unlock()
        return
    }
    // 遍历日志，找到截断的位置，并更新节点的相关属性
    for cutIndex, val := range rf.log {
        if val.Index == index {
            rf.lastIncludedIndex = index
            rf.lastIncludedTerm = val.Term
            rf.log = rf.log[cutIndex+1:]
            var tempLogArray []LogEntry = make([]LogEntry, 1)
            // 确保日志数组从索引 1 开始有效
            rf.log = append(tempLogArray, rf.log...)
        }
    }
    // 使用 labgob 编码器将当前节点状态和快照数据序列化
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.voteFor)
    e.Encode(rf.log)
    e.Encode(rf.lastIncludedIndex)
    e.Encode(rf.lastIncludedTerm)
    data := w.Bytes()
    // 将节点状态和快照数据保存到持久化存储中
    rf.persister.SaveStateAndSnapshot(data, snapshot)
    // 释放互斥锁
    rf.mu.Unlock()
}
```

图 15: Snapshot 函数

- 实现 InstallSnapshot 功能

```go
// InstallSnapshotArgs 结构定义了 InstallSnapshot RPC 请求的参数。
type InstallSnapshotArgs struct {
    Term int // 快照的任期
    LeaderId int // 领导者的ID
    LastIncludedIndex int // 最后包含的日志条目的索引
    LastIncludedTerm int // 最后包含的日志条目的任期
    Data []byte // 快照数据
}
```

图 16: InstallSnapshotArgs 结构体

```go
// InstallSnapshotReply 结构定义了 InstallSnapshot RPC 响应的参数。
type InstallSnapshotReply struct {
    Term int // 节点的当前任期
}
```

图 17: InstallSnapshotReply 结构体

```go
// InstallSnapshot 用于安装快照，处理来自 Leader 的 InstallSnapshot RPC 请求。
// 参数 args 包含了 Leader 发送的 InstallSnapshot 请求的参数，
// 参数 reply 用于返回节点的响应信息。

func (rf *Raft) InstallSnapshot(args *InstallSnapshotArgs, reply *InstallSnapshotReply) {
    // 互斥锁保护对节点状态的修改
    rf.mu.Lock()
    // 设置响应中的当前任期
    reply.Term = rf.currentTerm
    // 如果当前节点的任期大于请求中的任期，则拒绝安装快照
    if reply.Term > args.Term {
        rf.mu.Unlock()
        return
    }
    // 更新节点的状态信息
    if args.LastIncludedIndex > rf.lastIncludedIndex {
        rf.lastIncludedIndex = args.LastIncludedIndex
        rf.lastIncludedTerm = args.LastIncludedTerm

        // 如果当前节点的日志不包含最后包含的索引，更新日志
        if rf.lastLogIndex < args.LastIncludedIndex {
            rf.lastLogIndex = args.LastIncludedIndex
            rf.lastLogTerm = args.LastIncludedTerm
            rf.log = rf.log[0:1]
        } else {
            for cutIndex, val := range rf.log {
                if val.Index == args.LastIncludedIndex {
                    rf.log = rf.log[cutIndex+1:]
                    var tempLogArray []LogEntry = make([]LogEntry, 1)
                    // 确保日志数组从索引 1 开始有效
                    rf.log = append(tempLogArray, rf.log...)
                }
            }
        }
        // 更新节点的 lastApplied 和 commitIndex
        if rf.lastApplied < rf.lastIncludedIndex {
            rf.lastApplied = rf.lastIncludedIndex
        }
        if rf.commitIndex < rf.lastIncludedIndex {
            rf.commitIndex = rf.lastIncludedIndex
        }
        // 使用 labgob 编码器将当前节点状态和快照数据序列化
        w := new(bytes.Buffer)
        e := labgob.NewEncoder(w)
        e.Encode(rf.currentTerm)
        e.Encode(rf.voteFor)
        e.Encode(rf.log)
        e.Encode(rf.lastIncludedIndex)
        e.Encode(rf.lastIncludedTerm)
        data := w.Bytes()
        // 将节点状态和快照数据保存到持久化存储中
        rf.persister.SaveStateAndSnapshot(data, args.Data)
        // 生成快照应用消息，并通过 applyCh 发送到状态机
        var snapApplyMsg ApplyMsg
        snapApplyMsg.SnapshotValid = true
        snapApplyMsg.SnapshotIndex = args.LastIncludedIndex
        snapApplyMsg.SnapshotTerm = args.LastIncludedTerm
        snapApplyMsg.Snapshot = args.Data
        rf.mu.Unlock()
        rf.applyCh <- snapApplyMsg
    } else {
        // 如果当前节点的最后包含的索引不小于请求中的最后包含的索引，则直接释放锁
        rf.mu.Unlock()
    }
}
```

图 18: InstallSnapshot 函数

- 调整 Start 函数

```go
// Start 启动一个新的 Raft 客户端请求
func (rf *Raft) Start(command interface{}) (int, int, bool) {
    // 初始化返回值
    index := -1
    term := -1
    isLeader := true
    // Your code here (2B).
    _, isLeader = rf.GetState() // 获取当前节点的状态
    if !isLeader { // 如果不是 leader 节点，则返回当前节点的状态
        return index, term, isLeader
    }
    rf.mu.Lock() // 加锁，保护共享资源
    // 设置 lastLogTerm 和 lastLogIndex 为当前节点的最后一条日志的任期和索引
    rf.lastLogTerm = rf.currentTerm
    rf.lastLogIndex = rf.nextIndex[rf.me]
    index = rf.nextIndex[rf.me]
    term = rf.lastLogTerm
    // 创建新的日志条目
    var peerNum = rf.peerNum
    var entry = LogEntry{Index: index, Term: term, Command: command}
    rf.log = append(rf.log, entry)
    // 更新匹配索引和下一个索引
    rf.matchIndex[rf.me] = index
    rf.nextIndex[rf.me] = index + 1
    rf.persist() // 持久化状态
    // 遍历所有节点，向其他节点发送日志条目
    for i := 0; i < peerNum; i++ {
        if i == rf.me {
            continue
        }
        // 启动协程发送 AppendEntries RPC
        go func(id int, nextIndex int) {
            var args = &AppendEntriesArgs{} // 创建 AppendEntries 参数
            rf.mu.Lock()
            if rf.currentTerm > term { // 检查当前节点的任期是否大于待发送的任期
                rf.mu.Unlock()
                return
            }
            if rf.nextIndex[id] > nextIndex+1 {
                // 如果待发送的下一个索引已经过时，不发送 RPC 以节省网络带宽
                rf.mu.Unlock()
                return
            }
            args.Entries = make([]LogEntry, 0) // 创建 Entries 切片，准备发送
            // 如果待发送的索引小于当前节点的索引，将需要发送的日志追加到 Entries 中
            if nextIndex < index {
                for j := nextIndex + 1; j <= index; j++ {
                    args.Entries = append(args.Entries, rf.log[j-rf.lastIncludedIndex])
                }
            }
            // 设置 AppendEntries 参数的其他字段
            args.Term = term
            args.LeaderId = rf.me
            rf.mu.Unlock()
            for { // 向其他节点发送 AppendEntries RPC 请求
                var reply = &AppendEntriesReply{}
                rf.mu.Lock()
                if rf.currentTerm > term { // 检查当前节点的任期是否大于待发送的任期
                    rf.mu.Unlock()
                    return
                }
                // 处理快照发送
                if nextIndex <= rf.lastIncludedIndex {
                    // 如果待发送的索引小于等于快照的最后包含索引，发送 InstallSnapshot RPC
                    if nextIndex != rf.lastIncludedIndex {
                        rf.mu.Unlock()
                        return
                    }
                    var snapArgs InstallSnapshotArgs
                    var snapReply InstallSnapshotReply
                    snapArgs.Term = rf.currentTerm
                    snapArgs.LastIncludedIndex = rf.lastIncludedIndex
                    snapArgs.LastIncludedTerm = rf.lastIncludedTerm
                    snapArgs.LeaderId = rf.me
                    snapArgs.Data = rf.persister.ReadSnapshot()
                    rf.mu.Unlock()
                    // 发送 InstallSnapshot RPC
                    var count = 0
                    for {
                        if count == 3 {
                            return
                        }
                        if rf.sendInstallSnapshot(id, &snapArgs, &snapReply) {
                            break
                        }
                        count++
                    }
                    rf.mu.Lock()
                    if rf.currentTerm < snapReply.Term {
                        // 检查 InstallSnapshot 回复的任期是否大于当前节点的任期
                        rf.currentTerm = snapReply.Term
                        rf.state = "follower"
```

```
                rf.voteFor = -1
            } else { // 更新匹配索引和下一个索引
                if rf.matchIndex[id] < snapArgs.LastIncludedIndex {
                    rf.matchIndex[id] = snapArgs.LastIncludedIndex
                }
                if rf.nextIndex[id] <= snapArgs.LastIncludedIndex {
                    rf.nextIndex[id] = snapArgs.LastIncludedIndex + 1
                }
            }
            rf.mu.Unlock()
            return
        }
        // 设置 AppendEntries 参数的 PrevLogIndex 和 PrevLogTerm 字段
        if nextIndex-1-rf.lastIncludedIndex == 0 && rf.lastIncludedIndex != 0 {
            args.PrevLogIndex = rf.lastIncludedIndex
            args.PrevLogTerm = rf.lastIncludedTerm
        } else {
            args.PrevLogIndex = rf.log[nextIndex-1-rf.lastIncludedIndex].Index
            args.PrevLogTerm = rf.log[nextIndex-1-rf.lastIncludedIndex].Term
        }
        // 设置 AppendEntries 参数的 Entries 字段
        var tempLog = rf.log[nextIndex-rf.lastIncludedIndex : index+1-rf.lastIncludedIndex]
        args.Entries = make([]LogEntry, len(tempLog))
        copy(args.Entries, tempLog)
        rf.mu.Unlock()
        // 发送 AppendEntries RPC
        var count = 0
        for {
            if count == 3 {
                return
            }
            // 如果send失败，重试，直到成功
            if rf.sendAppendEntries(id, args, reply) {
                break
            }
            count++
        }
        rf.mu.Lock()
        if reply.Term > args.Term { // 检查回复的任期是否大于待发送的任期
            // 如果回复的任期大于当前节点的任期，更新当前节点的状态为 follower
            if reply.Term > rf.currentTerm {
                rf.currentTerm = reply.Term
                rf.state = "follower"
                rf.voteFor = -1
                rf.mu.Unlock()
                break
            }
            rf.mu.Unlock()
            break
        }
        // 检查是否需要发送 InstallSnapshot RPC
        var ifSendInstallSnapshot bool
        if !reply.Success {
            if nextIndex <= rf.lastIncludedIndex {
                ifSendInstallSnapshot = true
            } else if rf.log[nextIndex-1-rf.lastIncludedIndex].Term > reply.Term {
                for rf.log[nextIndex-1-rf.lastIncludedIndex].Term > reply.Term {
                    nextIndex--
                    if nextIndex-1-rf.lastIncludedIndex == 0 && rf.lastIncludedIndex != 0 {
                        if rf.lastIncludedTerm != reply.Term {
                            ifSendInstallSnapshot = true
                        }
                    }
                }
                if reply.ConflictIndex != 0 {
                    nextIndex = reply.ConflictIndex + 1
                    if nextIndex <= rf.lastIncludedIndex {
                        ifSendInstallSnapshot = true
                    }
                }
            } else {
                if reply.ConflictIndex != 0 {
                    nextIndex = reply.ConflictIndex + 1
                    if nextIndex <= rf.lastIncludedIndex {
                        ifSendInstallSnapshot = true
                    }
                } else {
                    nextIndex--
                }
            }
            // 如果需要发送 InstallSnapshot RPC，执行相应操作
            if ifSendInstallSnapshot {
                var snapArgs InstallSnapshotArgs
                var snapReply InstallSnapshotReply
                snapArgs.Term = rf.currentTerm
                snapArgs.LastIncludedIndex = rf.lastIncludedIndex
                snapArgs.LastIncludedTerm = rf.lastIncludedTerm
                snapArgs.LeaderId = rf.me
                snapArgs.Data = rf.persister.ReadSnapshot()
                rf.mu.Unlock()
                // 发送 InstallSnapshot RPC
                var count = 0
```

```
                    for {
                        if count == 3 {
                            return
                        }
                        if rf.sendInstallSnapshot(id, &snapArgs, &snapReply) {
                            break
                        }
                        count++
                    }
                    rf.mu.Lock()
                    // 检查 InstallSnapshot 回复的任期是否大于当前节点的任期
                    if rf.currentTerm < snapReply.Term {
                        rf.currentTerm = snapReply.Term
                        rf.state = "follower"
                        rf.voteFor = -1
                        rf.mu.Unlock()
                        return
                    } else { // 更新匹配索引和下一个索引
                        if rf.matchIndex[id] < snapArgs.LastIncludedIndex {
                            rf.matchIndex[id] = snapArgs.LastIncludedIndex
                        }
                        if rf.nextIndex[id] <= snapArgs.LastIncludedIndex {
                            rf.nextIndex[id] = snapArgs.LastIncludedIndex + 1
                        }
                        // 更新 nextIndex，如果 index 大于快照的最后包含索引加 1
                        if index > snapArgs.LastIncludedIndex+1 {
                            nextIndex = snapArgs.LastIncludedIndex + 1
                        } else {
                            rf.mu.Unlock()
                            return
                        }
                    }
                }
                if nextIndex == 0 {
                    rf.mu.Unlock()
                    break
                }
                rf.mu.Unlock()
            } else { // 更新匹配索引
                if rf.matchIndex[id] < index {
                    rf.matchIndex[id] = index
                } else {
                    rf.mu.Unlock()
                    return
                }
                // 检查是否达成一致，更新 commitIndex
                var mp = make(map[int]int)
                for _, val := range rf.matchIndex {
                    mp[val]++
                }
                var tempArray = make([]num2num, 0)
                for k, v := range mp {
                    tempArray = append(tempArray, num2num{key: k, val: v})
                }
                sort.Slice(tempArray, func(i, j int) bool {
                    return tempArray[i].key > tempArray[j].key
                })
                var voteAddNum = 0
                for j := 0; j < len(tempArray); j++ {
                    if tempArray[j].val+voteAddNum >= (rf.peerNum/2)+1 {
                        if rf.commitIndex < tempArray[j].key {
                            rf.commitIndex = tempArray[j].key
                            // 将已提交但未应用的日志条目应用到状态机
                            for rf.lastApplied < rf.commitIndex {
                                rf.lastApplied++
                                var applyMsg = ApplyMsg{}
                                applyMsg.Command = rf.log[rf.lastApplied-rf.lastIncludedIndex].Command
                                applyMsg.CommandIndex = rf.log[rf.lastApplied-rf.lastIncludedIndex].Index
                                applyMsg.CommandValid = true
                                rf.mu.Unlock()
                                rf.applyCh <- applyMsg
                                rf.mu.Lock()
                            }
                            break
                        }
                    }
                    voteAddNum += tempArray[j].val
                }
                rf.mu.Unlock()
                break
            }
            time.Sleep(10 * time.Millisecond)
        }
    }(i, rf.nextIndex[i])
    // 更新 nextIndex 数组，即使跟随者还没有收到消息
    if index+1 > rf.nextIndex[i] {
        rf.nextIndex[i] = index + 1
    }
}
rf.mu.Unlock() // 解锁，释放锁资源
return index, term, isLeader
}
```

图 19: Start 函数

2.6 lab3B 代码实现

- 修改 executeThread 函数

```go
// executeThread 处理 Raft 模块传递过来的 ApplyMsg
func (kv *KVServer) executeThread() {
    for {
        // var command raft.ApplyMsg
        var command = <-kv.applyCh
        kv.mu.Lock()
        if command.SnapshotValid {
            // 从快照中恢复数据
            r := bytes.NewBuffer(command.Snapshot)
            d := labgob.NewDecoder(r)
            var data map[string]string
            var commandLog map[int64]string
            var commandLogQueue List
            if d.Decode(&data) != nil || d.Decode(&commandLog) != nil || d.Decode(&commandLogQueue) != nil {
                log.Printf("                                    Error: server%d readSnapshot.", kv.me)
            } else {
                kv.Data = data
                kv.CommandLog = commandLog
                kv.CommandLogQueue = commandLogQueue
            }
            kv.mu.Unlock()
            continue
        } else if command.CommandValid {
            var msg = command.Command.(Op)
            _, ok1 := kv.CommandLog[msg.Id]
            // 如果请求 Id 已经存在于 CommandLog 中，说明该请求已经被执行过
            if !ok1 {
                if msg.Operation == "Get" {
                    result, ok2 := kv.Data[msg.Key]
                    if ok2 {
                        msg.Value = result
                    } else {
                        msg.Value = ""
                    }
                }
                if msg.Operation == "Put" {
                    kv.Data[msg.Key] = msg.Value
                }
                if msg.Operation == "Append" {
                    kv.Data[msg.Key] += msg.Value
                }
                // 控制 CommandLogQueue 的长度
                if kv.CommandLogQueue.Len >= 5 {
                    kv.CommandLogQueue.Pop()
                }
                kv.CommandLogQueue.Push(msg.Id)
                kv.CommandLog[msg.Id] = msg.Value

                // 如果达到 maxraftstate，进行快照
                var statesize, snapshotIndex = kv.rf.GetStateSizeAndSnapshotIndex()
                if kv.maxraftstate != -1 && statesize >= kv.maxraftstate && command.CommandIndex-snapshotIndex >= 20 {
                    // the size of raftstate is approaching maxraftstate
                    w := new(bytes.Buffer)
                    e := labgob.NewEncoder(w)
                    var encodeCommandLog = make(map[int64]string, 5)
                    for i := 0; i < kv.CommandLogQueue.Len; i++ {
                        var msgId = kv.CommandLogQueue.Data[i].(int64)
                        encodeCommandLog[msgId] = kv.CommandLog[msgId]
                    }
                    e.Encode(kv.Data)
                    e.Encode(encodeCommandLog)
                    e.Encode(kv.CommandLogQueue)
                    data := w.Bytes()
                    kv.rf.Snapshot(command.CommandIndex, data)
                }
            }
            kv.mu.Unlock()
            kv.cond.Broadcast()
        }
    }
}
```

图 20: executeThread 函数

- 修改 StartKVServer 函数

```go
// StartKVServer 启动 KV 服务器
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{}) // 在 Go 的 RPC 库中注册 Op 结构

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

    // You may need initialization code here.

    kv.applyCh = make(chan raft.ApplyMsg)
    kv.rf = raft.Make(servers, me, persister, kv.applyCh)
    kv.CommandLog = make(map[int64]string)
    kv.cond = sync.NewCond(&kv.mu)
    kv.Data = make(map[string]string)
    kv.CommandLogQueue = List{}
    var snapshot = kv.rf.GetSnapshot() // 从快照中恢复数据
    if len(snapshot) != 0 {
        r := bytes.NewBuffer(snapshot)
        d := labgob.NewDecoder(r)
        var data map[string]string
        var commandLog map[int64]string
        var commandLogQueue List
        if d.Decode(&data) != nil || d.Decode(&commandLog) != nil || d.Decode(&commandLogQueue) != nil {
        } else {
            kv.Data = data
            kv.CommandLog = commandLog
            kv.CommandLogQueue = commandLogQueue
        }
    }
    go kv.executeThread() // 启动执行线程
    // You may need initialization code here.

    return kv
}
```

图 21: StartKVServer 函数

2.7 测试结果

```
dbs@ubuntu:~/go-workplace/6.824/src/raft$ go test -run 2D
Test (2D): snapshots basic ...
  ... Passed --  30.0  3  734  249406  235
Test (2D): install snapshots (disconnect) ...
  ... Passed --  48.7  3 1379  481118  328
Test (2D): install snapshots (disconnect+unreliable) ...
  ... Passed --  63.1  3 1754  557942  313
Test (2D): install snapshots (crash) ...
  ... Passed --  33.1  3  997  376236  328
Test (2D): install snapshots (unreliable+crash) ...
  ... Passed --  39.4  3 1146  382572  316
Test (2D): crash and restart all servers ...
  ... Passed --   7.7  3  262   79254   59
PASS
ok      6.824/raft      222.100s
```

图 22: 2D 运行结果

图 23: 3B 运行结果

2.8 个人总结

这个阶段就是引入了一个快照机制。然而，要成功实现这一机制，必须深入了解实验中快照是在何时以及如何进行的，否则就会遇到各种问题。例如，在测试 Lab2D 的过程中，我突然发现 learer 发生了死锁，测试结果还显示 lastapplied 的 index 值与 commandIndex 值不匹配等问题。

Lab2D 的代码修改范围相当大，因为涉及到了 rf.log 的索引值的修改以及 log replication 的运行逻辑的调整。在编写 Lab2D 的过程中，感觉就像以往的实验一样，运行测试，查找 bug，然后打补丁。

相比之下，Lab3B 的任务相对简单。首先，在当前储存的日志数量大于 maxraftstate 时，需要调用 snapshot() 函数，并传入需要保存的持久化数据。其次，在启动 KV 服务器时，需要读取可能存在的快照数据。

3 心得体会

lab3 比 lab2 更加自由一点，主要是没有可供参考的论文，也没有丰富的资料可供查询，同时调试的难度也明显提升。尽管如此，整个系统的实现思路基本上是一脉相承的，并没有太多自行发挥的空间。在查阅了一些网上的博客之后，我发现虽然不同的实现方式各有不同，但设计思路却是相似的。

相对于 Lab2，Lab3 的代码复杂性并不高，因此代码量也较为有限。在实现过程中，我尽量避免修改 raft 的源码，因为一旦对 raft 源码进行修改，可能会带来不必要的麻烦，尤其是在出现 bug 时修改起来更为头疼。这种小心谨慎的策略帮助我有效地应对了潜在的问题。

整个 Lab 3 的实践是我在分布式系统领域深入学习的重要一步。实现基于 Raft 的键值存储服务是这个实验的重要组成部分。这一部分让我学到了如何设计和构建一个分布式系统，考虑节点的协同工作、一致性维护以及客户端与服务器之间的通信。在这个实验中，我不仅巩固了 Raft 一致性算法的理论知识，还通过完成构建基于 Raft 的分布式键值存储服务的实验，将理论知识应用到实际场景，为我在分布式系统领域的学习和实践提供了宝贵的经验。我期待着在未来的学习中继续探索分布式系统的更多方面。

4 参考资料

- MIT 6.824 学习（二）【Raft】: https://blog.csdn.net/weixin_48922154/article/details/123956614
- MIT6.824 Lab3 实现与踩坑: https://blog.csdn.net/darkLight2017/article/details/122587599
- MIT 6.824 Lab3 翻译（Key/Value Server Base on Raft）:https://zhuanlan.zhihu.com/p/267685488
- Lecture 06 - Raft1 : https://mit-public-courses-cn-translatio.gitbook.io/mit6-824/lecture-06-raft1
- MIT 6.824 lab3 KVRaft : https://blog.csdn.net/Miracle_ma/article/details/80184594