

目 录

0.1	README (请老师和助教老师在看报告和源代码之前, 先看看 README, 谢谢!)	1
1	2D、3A 和 3B 的测试结果汇总	2
2	Part 3A: Key/value service without snapshots	3
2.1	准备工作	3
2.1.1	基础原理	3
2.1.2	功能和组件	3
2.2	编写 Client 部分	4
2.2.1	构建维护客户端的结构体	5
2.2.2	实例化 Clerk 函数 MakeClerk	6
2.2.3	编写 Get/Put/Append 函数	6
2.2.4	编写 Comamand 函数	7
2.2.5	编写需要用到的随机数生成算法	8
2.3	写 common.go	8
2.4	编写 server.go	11
2.4.1	确定 KVServer 的结构	11
2.4.2	编写服务端发送过来的 Command 请求	12
2.4.3	编写两个处理管道的工具函数	14
2.4.4	编写 StartKVServer 函数	14
2.4.5	applier 函数的编写和详解	16
2.5	3A 测试	17
3	Part 3B: Key/value service with snapshots	20
3.1	基本准备工作	20
3.2	raft 的 2D 部分编写: Part 2D: log compaction	20
3.2.1	准备工作	20
3.2.2	编写 Snapshot 函数	20
3.2.3	实现 InstallSnapshot RPC	23
3.2.4	处理 InstallSnapshot RPC	24
3.2.5	服务器重启时的状态恢复	25
3.2.6	一次性发送完整快照	26

3.2.7	丢弃旧日志条目	26
3.2.8	实现日志被修剪后的 AppendEntries	27
3.2.9	快照信息的引用	28
3.2.10	编写 CondInstallSnapshot 函数	28
3.2.11	编写网络调用	29
3.2.12	完善 replicateOneRound 函数	29
3.2.13	编写 genInstallSnapshotRequest 函数	32
3.2.14	编写 handleInstallSnapshotResponse 函数	32
3.2.15	2D 测试	34
3.3	3B 任务解读	34
3.4	完善 StartKVServer 函数	36
3.5	完善 applier 函数	37
3.6	完善快照相关工具函数	40
3.6.1	needSnapshot	40
3.6.2	takeSnapshot	41
3.6.3	restoreSnapshot	41
3.7	3B 测试	44
4	实验过程中遇到的问题和解决办法	45
4.1	困难: Command 函数中 RPC 调用失败	45
4.2	困难: applyLogToStateMachine 中的命令重复	45
4.3	困难: 快照恢复后数据不一致	45
4.4	困难: 处理 ‘InstallSnapshot’ RPC 导致的死锁	45
4.5	困难: Snapshot 方法中日志裁剪错误	45
4.6	困难: applier 循环中的快照处理问题	45
4.7	困难: 在 Raft 重启时未能恢复快照	46
4.8	困难: KVServer 快照阈值判断错误	46
4.9	困难: takeSnapshot 中快照创建失败	46
4.10	困难: 重启后 KVServer 状态与 Raft 不一致	46
5	实验总结	47

1 2D、3A 和 3B 的测试结果汇总

2D 部分、3A 部分和 3B 部分的测试截图分别如图 1-1, 1-2, 1-3 所示。

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/raft$ go test -run 2D
Test (2D): snapshots basic ...
... Passed -- 5.4 3 183 66206 251
Test (2D): install snapshots (disconnect) ...
... Passed -- 55.4 3 1071 292516 355
Test (2D): install snapshots (disconnect+unreliable) ...
... Passed -- 66.1 3 1271 325168 344
Test (2D): install snapshots (crash) ...
... Passed -- 43.4 3 687 193483 355
Test (2D): install snapshots (unreliable+crash) ...
... Passed -- 57.1 3 771 206500 355
PASS
ok      6.824/raft      227.434s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/raft$
```

图 1-1 2D 测试结果

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$ go test -run 3A -race
Test: one client (3A) ...
... Passed -- 15.1 5 8264 913
Test: ops complete fast enough (3A) ...
Test: many clients (3A) ...
... Passed -- 15.5 5 9773 1187
Test: unreliable net, many clients (3A) ...
... Passed -- 16.2 5 5334 906
Test: concurrent append to same key, unreliable (3A) ...
... Passed -- 1.8 3 245 52
Test: progress in majority (3A) ...
... Passed -- 1.3 5 128 2
Test: no progress in minority (3A) ...
... Passed -- 1.0 5 193 3
Test: completion after heal (3A) ...
... Passed -- 1.0 5 51 3
Test: partitions, one client (3A) ...
... Passed -- 22.9 5 14589 785
Test: partitions, many clients (3A) ...
... Passed -- 23.2 5 25916 1087
Test: restarts, one client (3A) ...
... Passed -- 22.0 5 18722 870
Test: restarts, many clients (3A) ...
... Passed -- 22.9 5 24752 1166
Test: unreliable net, restarts, many clients (3A) ...
... Passed -- 24.3 5 6767 891
Test: restarts, partitions, many clients (3A) ...
... Passed -- 29.8 5 29361 1046
Test: unreliable net, restarts, partitions, many clients (3A) ...
... Passed -- 30.0 5 5881 876
Test: unreliable net, restarts, partitions, random keys, many clients (3A) ...
... Passed -- 33.4 7 13429 947
PASS
ok      6.824/kvraft    274.002s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$
```

图 1-2 3A 测试结果

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$ go test -run 3B -race
Test: InstallSnapshot RPC (3B) ...
... Passed -- 4.4 3 3847 63
Test: snapshot size is reasonable (3B) ...
... Passed -- 2.4 3 6281 800
Test: ops complete fast enough (3B) ...
Test: restarts, snapshots, one client (3B) ...
... Passed -- 21.9 5 46775 6970
Test: restarts, snapshots, many clients (3B) ...
... Passed -- 22.2 5 44291 7575
Test: unreliable net, snapshots, many clients (3B) ...
... Passed -- 15.9 5 6845 1281
Test: unreliable net, restarts, snapshots, many clients (3B) ...
... Passed -- 23.2 5 8272 1260
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
... Passed -- 30.2 5 6377 587
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (3B) ...
... Passed -- 32.1 7 17542 1772
PASS
ok      6.824/kvraft    154.822s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$
```

图 1-3 3B 测试结果

2 Part 3A: Key/value service without snapshots

2.1 准备工作

在准备做 Lab3 之前我们要先搞清楚整个 kv 客户服务的架构。

下面是 Lab3 的整体原理和架构。

MIT 6.824 Lab 3 的核心任务是实现一个容错的、基于 Raft 协议的分布式键值存储系统。这个系统利用 Raft 算法的一致性保证，确保即使在服务器失败的情况下也能提供可靠的服务。以下是整体原理和功能的描述。

2.1.1 基础原理

1. **Raft 协议**: Raft 是一种分布式共识算法，用于管理分布式系统中的日志复制。它确保所有的节点能够以相同的顺序应用相同的日志条目，即使在节点故障或网络问题的情况下。

2. **日志复制**: Raft 通过日志复制来同步各个服务器节点的状态。每个键值对的操作（如 put 或 append）都被封装成日志条目，并通过 Raft 协议在集群中进行复制。

3. **领导者选举**: Raft 实现了领导者选举机制。在任何给定时间，集群中只有一个领导者节点负责处理客户端请求，并将操作日志复制到其他跟随者节点。

2.1.2 功能和组件

1. **KVServer**: KVServer 结构体代表了分布式键值存储系统中的单个服务器节点。它负责处理来自客户端的请求，通过 Raft 日志复制机制与其他节点同步，并维护本地状态机（键值存储）。

2. **状态机**: 系统中的每个 KVServer 都包含一个状态机，用于存储键值对数据。状态机根据从 Raft 层接收到的已提交日志条目进行更新。

3. **客户端 (Clerk)**: Clerk 结构体表示客户端，负责向集群发送键值存储操作（如 get、put 和 append 请求）。客户端通过 RPC 与服务器节点通信，并处理领导者更换等问题。

4. **容错和一致性**: 通过 Raft 算法，系统能够处理服务器节点的失败，并确保数据的一致性。如果领导者节点失败，会进行新的领导者选举，而客户端将尝试与新的领导者节点通信。

整个 Lab3 的整体架构如图 2-1 所示。

通过研究 Lab3 手册 (<http://nil.csail.mit.edu/6.824/2022/labs/lab-kvraft.html>) 的 3A 部分，对所有函数的调用逻辑进行了整理，调用逻辑如图 2-2 所示。

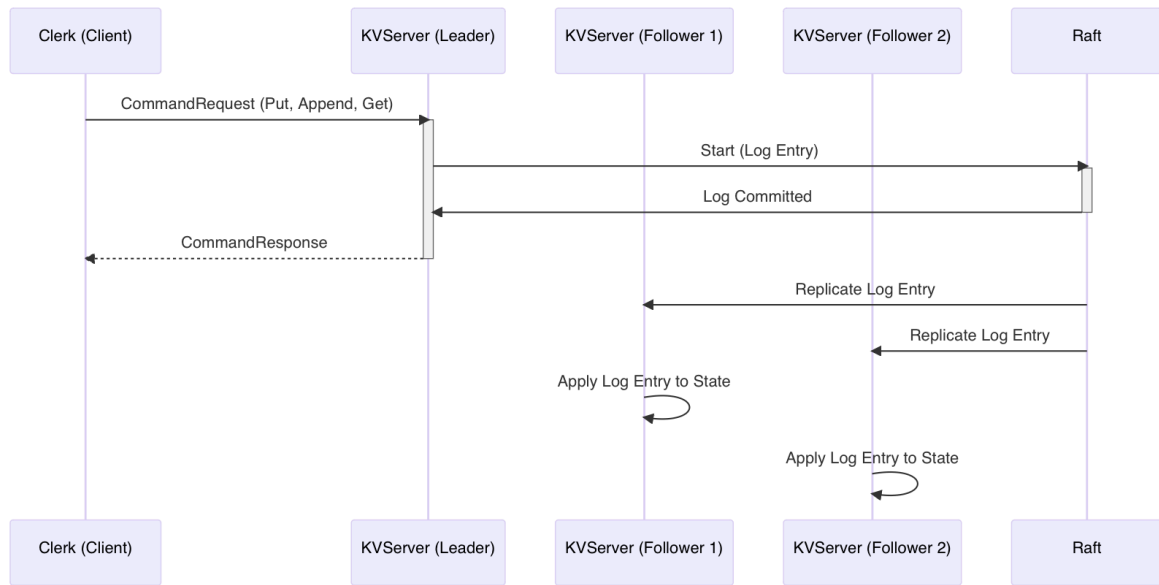


图 2-1 Lab3 的整体架构

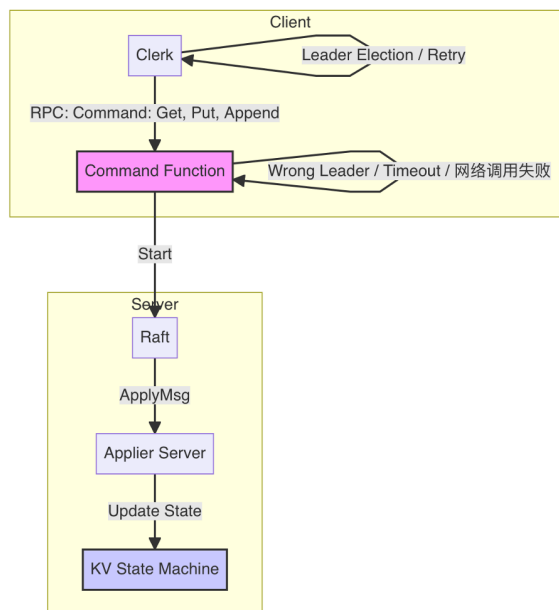


图 2-2 Lab3A 部分函数的调用逻辑

2.2 编写 Client 部分

我们先来看看整个 Client 的结构，如图 2-3 所示。

整体结构解释：上层调用 Get, put, append 三种方法，向服务端发送三种不同的请求。在原先直接 git clone 下来的 mit 代码中，Get/Put/Append 三个函数是直接向服务端发送网络调用的，我认为这样代码的维护性可能不是特别好，因此我对代码做了一定程度的修改。

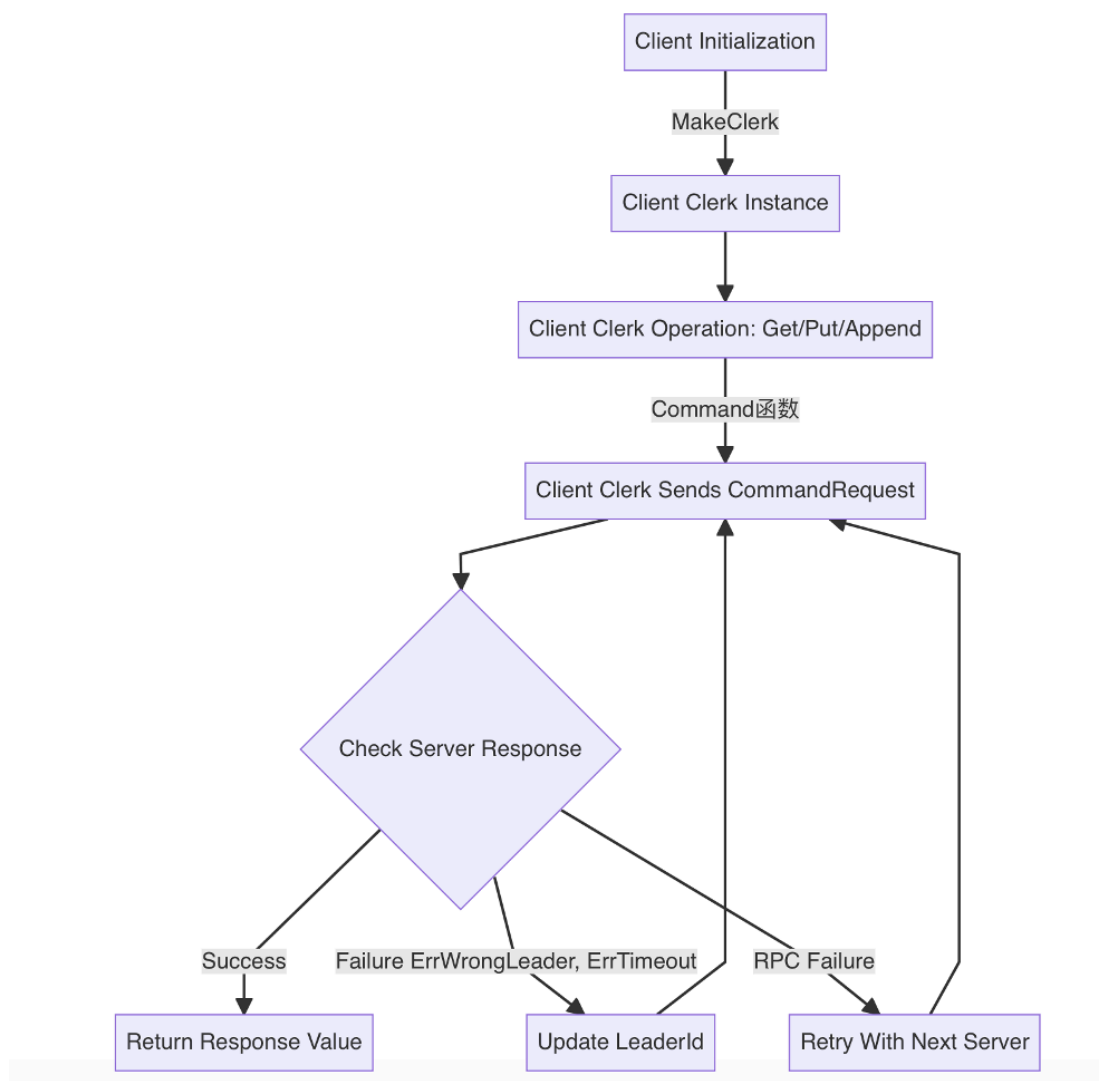


图 2-3 Client 整体结构

1. Get/Put/Append 不再直接向 Server 发送网络调用（请求），我封装多了一层 Command 函数，让 Command 函数统一处理 RPC 对发送和响应。
 2. Get/Put/Append 会创建 CommandRequest 交给 Command 函数。
 3. 我重新设计了 Op 结构体，使其可读性更好。
- 具体实现见下面的的代码。

2.2.1 构建维护客户端的结构体

按照手册要求，我们把构建维护客户端的结构体命名为 Clerk。具体结构如代码所示。

```
language
// Clerk就是客户端！
```

```
type Clerk struct {
    servers []*labrpc.ClientEnd // 指向通信服务的指针
    leaderId int64                // 当前领导者id
    clientId int64                // 这个字段是客户端的唯一标识符
    commandId int64              // (clientId, commandId) defines a operation uniquely
    // (clientId, commandId) 可以表示一个唯一的操作
}
```

2.2.2 实例化 Clerk 函数 MakeClerk

实现很简单，就是创建一个 Clerk 对象，然后往这个 Clerk 对象里面添加东西就可以了。

具体如代码所示。

```
language
// 建立一个客户端
func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
    // 先实例化一个Clerk对象
    ck := new(Clerk)
    // 然后向里面填写字段
    ck.servers = servers // 填写外面传来的servers指针
    // You'll have to add code here.
    ck.leaderId = 0      // 先让当前的leaderId为0
    ck.clientId = nrand() // 随机生成一个数作为客户端的id, 避免冲突
    ck.commandId = 0     // 先初始化command为0
    return ck
}
```

2.2.3 编写 Get/Put/Append 函数

过程很简单，直接创建 CommandRequest 结构体即可，然后传入设计好的参数即可。

代码如下所示。

```
language
// 提示是需要我们直接调用 KVServer.Get 的
// 但是因为server.go里面我把这三个方法封装了，所有都是统一向
// server发送CommandRequest请求
// 所以在这三个函数里面
// 1. 先创建CommandRequest结构体
// 2. 传给Command来发送CommandRequest给server
```

```

func (ck *Clerk) Get(key string) string {
    return ck.Command(&CommandRequest{Key: key, Op: OpGet})
}
func (ck *Clerk) Put(key string, value string) {
    ck.Command(&CommandRequest{Key: key, Value: value, Op: OpPut})
}
func (ck *Clerk) Append(key string, value string) {
    ck.Command(&CommandRequest{Key: key, Value: value, Op: OpAppend})
}

```

2.2.4 编写 Comamand 函数

Get/Put/Append 函数都会调用这个函数，这个函数要完成 Get/Put/Append 函数发送 RPC 的任务。

当 RPC 发送失败，或者遇到领导者错误或者超时错误的时候，重复发送即可。具体代码和注释解释如下所示。

```

language
// 向分布式键值存储系统发送一个命令，并处理响应
func (ck *Clerk) Command(request *CommandRequest) string {
    request.ClientId, request.CommandId = ck.clientId, ck.commandId
    for {
        // 这里有个for，但是这里如果正常来说是只执行一次的
        var response CommandResponse // 先构建CommandResponse结构
        if !ck.servers[ck.leaderId].Call("KVServer.Command", request, &response) || response.Err ==
            ErrWrongLeader || response.Err == ErrTimeout {
            // 走到这里说明调用失败了
            /*
                如果 RPC 调用失败或者返回了错误的领导者或超时错误，这行代码会更新 leaderId，
                以尝试下一个服务器。这是一个简单的循环尝试，不断遍历服务器列表
            */
            ck.leaderId = (ck.leaderId + 1) % int64(len(ck.servers))
            continue // 再次调用
        }
        ck.commandId++ // 走到这里说明网络调用成功了，commandId要++，因为这个是唯一标识符来的
        return response.Value
    }
}

```


2.2.5 编写需要用到的随机数生成算法

实现很简单，如代码所示。

```
language
// 这里是一个大数随机算法，生成一个随机数，避免冲突
func nrand() int64 {
    max := big.NewInt(int64(1) << 62)
    bigx, _ := rand.Int(rand.Reader, max)
    x := bigx.Int64()
    return x
}
```

2.3 写 common.go

这里面都是一些服务端和客户端都需要用到一些 rpc 定义，和一些结构，工具的定义。

关于所有字段的含义，所有结构的含义和解释，我都写在了注释里，相信这样可以给大家表达清楚！

代码如下所示。

```
language
package kvraft

import "fmt"
import "time"
import "log"

const Debug = false

func DPrintf(format string, a ... interface{}) (n int, err error) {
    if Debug {
        log.Printf(format, a...)
    }
    return
}

// 定义一些错误的类型
// 原来使用字符串的，我直接用数字了

type Err uint8    // 定义错误的类型
```

```

const (
    OK Err = iota // 未出错
    ErrNoKey      // 出现错误: 没有Key
    ErrWrongLeader // 出现错误: 错误的LeaderId
    ErrTimeout    // RPC网络调用超时
)
// 用字符串可视化Err类型
func (err Err) String() string {
    switch err {
    case OK:
        return "OK"
    case ErrNoKey:
        return "ErrNoKey"
    case ErrWrongLeader:
        return "ErrWrongLeader"
    case ErrTimeout:
        return "ErrTimeout"
    }
    panic(fmt.Sprintf("unexpected Err %d", err))
}

const ExecuteTimeout = 500 * time.Millisecond // 设置最大的响应时间, 如果超过了, 就是超时

// 这里定义操作的类型
/*
    Command 结构体是对 CommandRequest 的一个封装。
    它通过嵌入一个指向 CommandRequest 结构体的指针来继承所有 CommandRequest 的字段和方法。
    这样的设计通常用于简化代码结构或为了类型匹配, 特别是在进行 RPC 调用时。
*/
type Command struct {
    *CommandRequest
}

type OperationContext struct {
    /*
        这个字段记录了对应客户端最后一次成功应用(执行)的命令ID。
        这对于避免重复执行相同的操作很重要, 特别是在分布式系统中,
        客户端的同一个请求可能因为网络原因被多次发送
    */
    MaxAppliedCommandId int64
}

```

```

/*
    这个字段存储了最后一次操作的响应。如果客户端重复发送相同的请求,
    服务器可以直接返回这个已存储的响应, 而不是重新执行操作
*/
LastResponse      *CommandResponse
}

type OperationOp uint8    // 用整数定义操作类型
const (
    OpPut OperationOp = iota // 表示Put操作
    OpAppend                // 表示Append操作
    OpGet                   // 表示Get操作
)

// 把OperationOp打印成字符串表示出来
func (op OperationOp) String() string {
    switch op {
    case OpPut:
        return "OpPut"
    case OpAppend:
        return "OpAppend"
    case OpGet:
        return "OpGet"
    }
    panic(fmt.Sprintf("unexpected OperationOp %d", op))
}

// 这里是传递命令的RPC, 我把PutAppendArgs改掉了, 改个名字好听点
type CommandRequest struct {
    Key      string
    Value     string
    Op        OperationOp
    ClientId int64    // 客户端唯一标识符
    CommandId int64    // 命令唯一标识符
}

func (request CommandRequest) String() string {
    return fmt.Sprintf("{Key:%v,Value:%v,Op:%v,ClientId:%v,CommandId:%v}", request.Key, request.Value,
        request.Op, request.ClientId, request.CommandId)
}

type CommandResponse struct {
    Err  Err          // 运行状态
    Value string        // 运行结果
}

```

```

}
func (response CommandResponse) String() string {
    return fmt.Sprintf("{Err:%v,Value:%v}", response.Err, response.Value)
}

```

2.4 编写 server.go

2.4.1 确定 KVServer 的结构

KVServer 所需要的基本结构

1. 读写锁，保证线程安全
2. 标识自己的标识符
3. 标识自己是否已经被杀死的标识符
4. 指向 Raft 实例的指针
5. 用于接收 Raft 层消息的通道

```

language
type KVServer struct {
    mu      sync.RWMutex
    me      int
    dead    int32
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg

    maxRaftState int // 这个字段表示日志大小的阈值，用于决定何时对 Raft 日志进行快照处理。
                                // 如果日志大小超过这个阈值，KVServer 可能会执行快照以减少日志
                                大小
    lastApplied  int // 这个字段记录了最后一个被应用到状态机的日志条目的索引

    stateMachine KVStateMachine
    lastOperations map[int64]OperationContext
    notifyChans   map[int]chan *CommandResponse
}

```

stateMachine - KV 的状态机。

lastOperations - 这是一个映射，它记录了每个客户端的最后一个操作上下文。这用于防止日志的重复应用，确保每个客户端的操作只应用一次。

notifyChans - 这是一个映射，其键是日志条目的索引，值是通道，这些通道用于通知客户端有关其请求的响应。当一个命令被应用到状态机时，相应的通道被用来发送响应给客户端。

2.4.2 编写服务端发送过来的 Command 请求

对于 Command 请求的处理，我整理成了流程图，如图 2-4 所示。

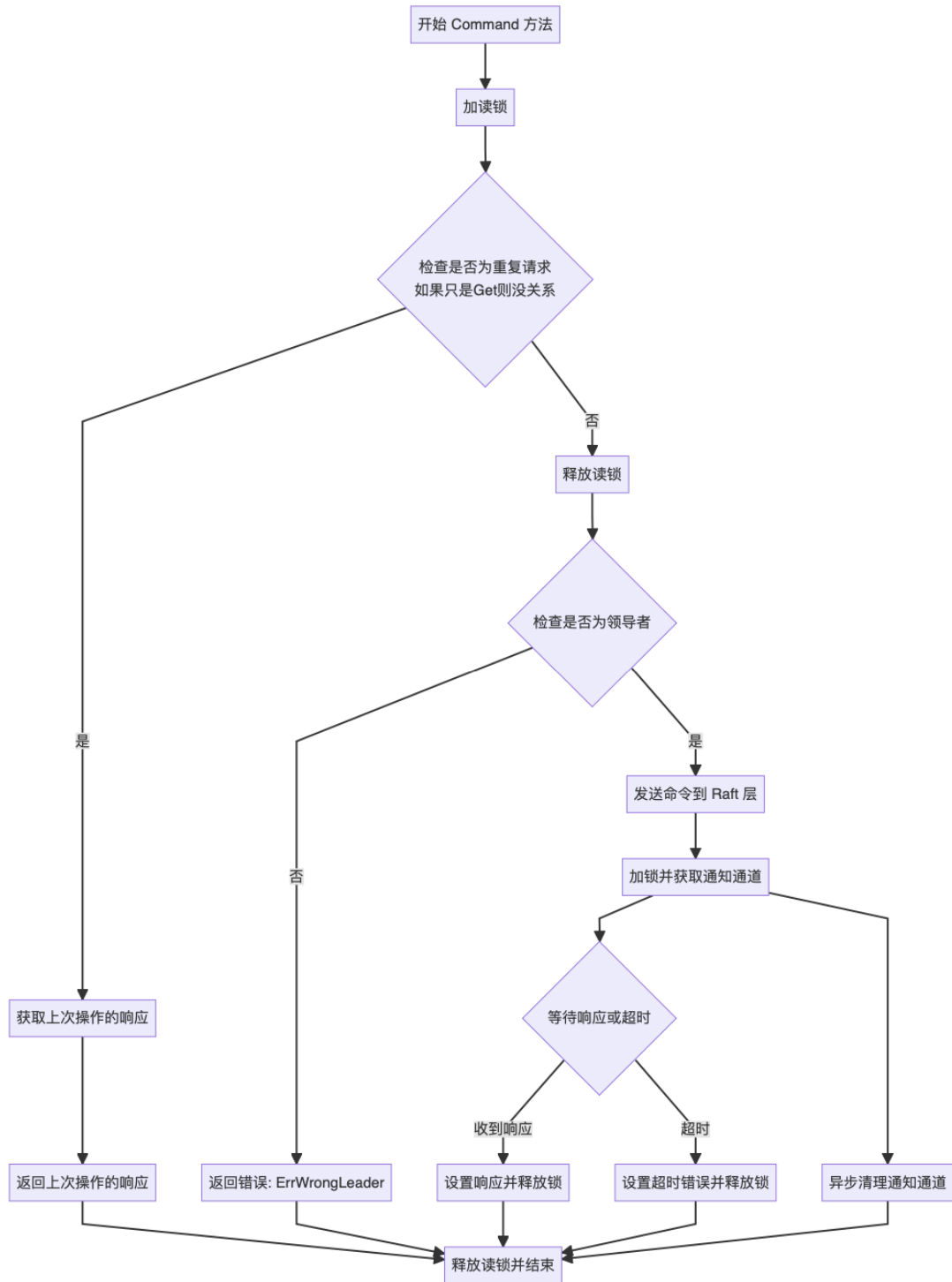


图 2-4 Command 请求的处理流程

要注意：提前 return 之前记得释放锁，不然会造成死锁导致代码崩溃。

代码如下所示。

language

```
// 这里把Get和Put封装成一个Command函数
func (kv *KVServer) Command(request *CommandRequest, response *CommandResponse) {
    defer DPrintf("{Node_%v}_processes_CommandRequest_%v_with_CommandResponse_%v", kv.rf.Me(),
        request, response)
    // return result directly without raft layer's participation if request is duplicated
    kv.mu.RLock()
    if request.Op != OpGet && kv.isDuplicateRequest(request.ClientId, request.CommandId) {
        // 判断是否是重复的操作
        // 但是如果是Get操作，那就无所谓，不用走进这个if里面
        lastResponse := kv.lastOperations[request.ClientId].LastResponse // 获取上一个操作
        response.Value, response.Err = lastResponse.Value, lastResponse.Err // 返回上一个操作即可
        kv.mu.RUnlock() // 一定要记住return之前
        return
    }
    kv.mu.RUnlock()
    // do not hold lock to improve throughput
    // when KVServer holds the lock to take snapshot, underlying raft can still commit raft logs

    /*
        调用 Raft 实例的 Start 方法，将命令添加到 Raft 日志中
        这个操作返回命令在日志中的索引和服务器是否是领导者的信息
        此外：调用raft不要加锁
    */
    index, _, isLeader := kv.rf.Start(Command{request})
    // 如果raft底层返回的leader不是现在的leader->报ErrWrongLeader错误
    if !isLeader {
        response.Err = ErrWrongLeader
        return
    }
    kv.mu.Lock()
    ch := kv.getNotifyChan(index) // 获取与命令索引关联的通知通道
    kv.mu.Unlock()
    select {
    case result := <-ch: // 如果从通道中接收到结果，设置响应值和错误
        response.Value, response.Err = result.Value, result.Err
    case <-time.After(ExecuteTimeout):
        response.Err = ErrTimeout // 如果等待超时，设置响应错误为超时
    }
    // release notifyChan to reduce memory footprint
}
```

```
// why asynchronously? to improve throughput, here is no need to block client request
go func() {
    // 使用 Go 例程异步清理过时的通知通道，以减少内存占用并提高吞吐量
    kv.mu.Lock()
    kv.removeOutdatedNotifyChan(index) // 清理通道
    kv.mu.Unlock()
}()
}
```

2.4.3 编写两个处理管道的工具函数

首先第一个是 getNotifyChan 函数。

作用：获取与特定日志条目索引关联的通知通道，如果该通道不存在则创建它。

```
language
func (kv *KVServer) getNotifyChan(index int) chan *CommandResponse {
    // 先判断我们kv维护的数组里面有没有这个管道，如果没有就创建，如果有就直接返回了
    if _, ok := kv.notifyChans[index]; !ok {
        kv.notifyChans[index] = make(chan *CommandResponse, 1) // 这里直接创建管道
    }
    return kv.notifyChans[index] // 然后返回即可
}
```

第二个是 removeOutdatedNotifyChan，它的作用是清理管道里面的东西。

在 Command 中是异步调用的。

```
language
func (kv *KVServer) removeOutdatedNotifyChan(index int) {
    delete(kv.notifyChans, index) // 删除映射中给定索引 index 的项
}
```

这两个函数很简单，不多说了。

2.4.4 编写 StartKVServer 函数

这个函数很重要，这个函数是启动整个服务器的主要函数。

流程也非常的简单，如图 2-5 所示。

代码如下所示。

```
language
func StartKVServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister, maxraftstate int) *
KVServer {
```

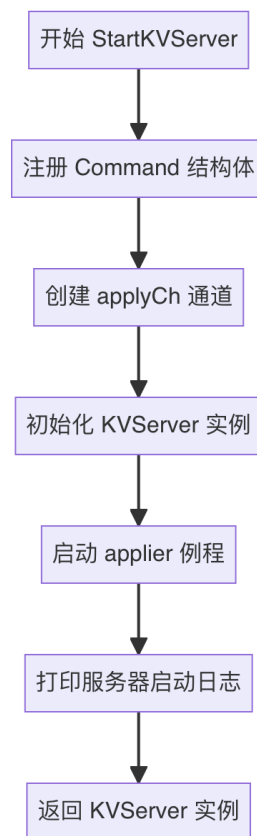


图 2-5 StartKVServer 函数执行流程

```

// call labgob.Register on structures you want
// Go's RPC library to marshall/unmarshall.

/*
    这行代码使用 labgob 包注册 Command 结构体，
    以便在 Go 的 RPC 系统中进行序列化和反序列化。这是分布式系统中节点间通信所必需的
*/
labgob.Register(Command{})
// 创建一个新的通道 applyCh，用于接收来自 Raft 层的 ApplyMsg 消息
applyCh := make(chan raft.ApplyMsg)

// 实例化一个KVServer，里面的字段也没啥好说的，就是按照设计的来初始化即可
kv := &KVServer{
    maxRaftState: maxraftstate,
    applyCh:      applyCh,
    dead:         0,
    lastApplied:  0,
    rf:           raft.Make(servers, me, persister, applyCh),
    stateMachine: NewMemoryKV(),

```



```

    lastOperations: make(map[int64]OperationContext),
    notifyChans:    make(map[int]chan *CommandResponse),
}
// start applier goroutine to apply committed logs to stateMachine
go kv.applier() // 启动applier协程

DPrintf("{Node_%v}_has_started", kv.rf.Me())
return kv
}

```

关于 applier 协程的细节，我在下一小节详细描述。

2.4.5 applier 函数的编写和详解

applier 方法是 kv 服务器的重要方法，负责监听 Raft 层的消息，并将日志条目应用到服务器的状态机。此外，它还负责处理重复请求和通知客户端关于其命令的结果。

通过了解和学习，我把 applier 函数的执行过程整理成了流程图，如图 2-6 所示。

相信这个图可以解释清楚我对这个函数的理解。

具体实现和具体代码解释如下所示。

```

language
func (kv *KVServer) applier() {
    // 只要服务器没有被杀死，就一直运行
    for kv.killed() == false {
        select {
            // 多路转接异步管理多个管道
            case message := <-kv.applyCh:
                // applyCh上是有来自raft的mesg的，如果有mesg来了，就进入到下面的函数中
                DPrintf("{Node_%v}_tries_to_apply_message_%v", kv.rf.Me(), message)
                if message.CommandValid {
                    // 检查接收到的消息是否包含有效的命令
                    kv.mu.Lock()
                    if message.CommandIndex <= kv.lastApplied {
                        // 检查命令索引是否已经被应用，以避免重复应用
                        DPrintf("{Node_%v}_discards_outdated_message_%v_because_a_newer_snapshot_which_
                            lastApplied_is_%v_has_been_restored", kv.rf.Me(), message, kv.lastApplied)
                        kv.mu.Unlock()
                        continue
                    }
                }
                kv.lastApplied = message.CommandIndex // 更新最后应用的日志条目索引
            }
        }
    }
}

```

```

var response *CommandResponse
command := message.Command.(Command) // 构建一个command
if command.Op != OpGet && kv.isDuplicateRequest(command.ClientId, command.
    CommandId) {
    // 判断是否为重复的命令请求
    DPrintf("{Node_%v} doesn't apply duplicated message %v to stateMachine because
        maxAppliedCommandId is %v for client %v", kv.rf.Me(), message, kv.
            lastOperations[command.ClientId], command.ClientId)
    response = kv.lastOperations[command.ClientId].LastResponse // 如果重复了就去获取旧
        的指令
} else {
    response = kv.applyLogToStateMachine(command) // 否则, 应用日志到状态机
    if command.Op != OpGet {
        // 如果不是 Get 操作, 更新 lastOperations 映射
        kv.lastOperations[command.ClientId] = OperationContext{command.CommandId,
            response}
    }
}

// only notify related channel for currentTerm's log when node is leader
if currentTerm, isLeader := kv.rf.GetState(); isLeader && message.CommandTerm ==
    currentTerm {
    // 如果当前节点是领导者并且命令属于当前任期
    ch := kv.getNotifyChan(message.CommandIndex)
    ch <- response
}
kv.mu.Unlock()
} else if message.SnapshotValid {
    kv.mu.Lock()
    kv.mu.Unlock()
} else {
    // 处理意外情况
    panic(fmt.Sprintf("unexpected_Message_%v", message))
}
}
}
}

```

2.5 3A 测试

如图 2-7 所示, 经过多次测试, 3A 均可以完美通过。

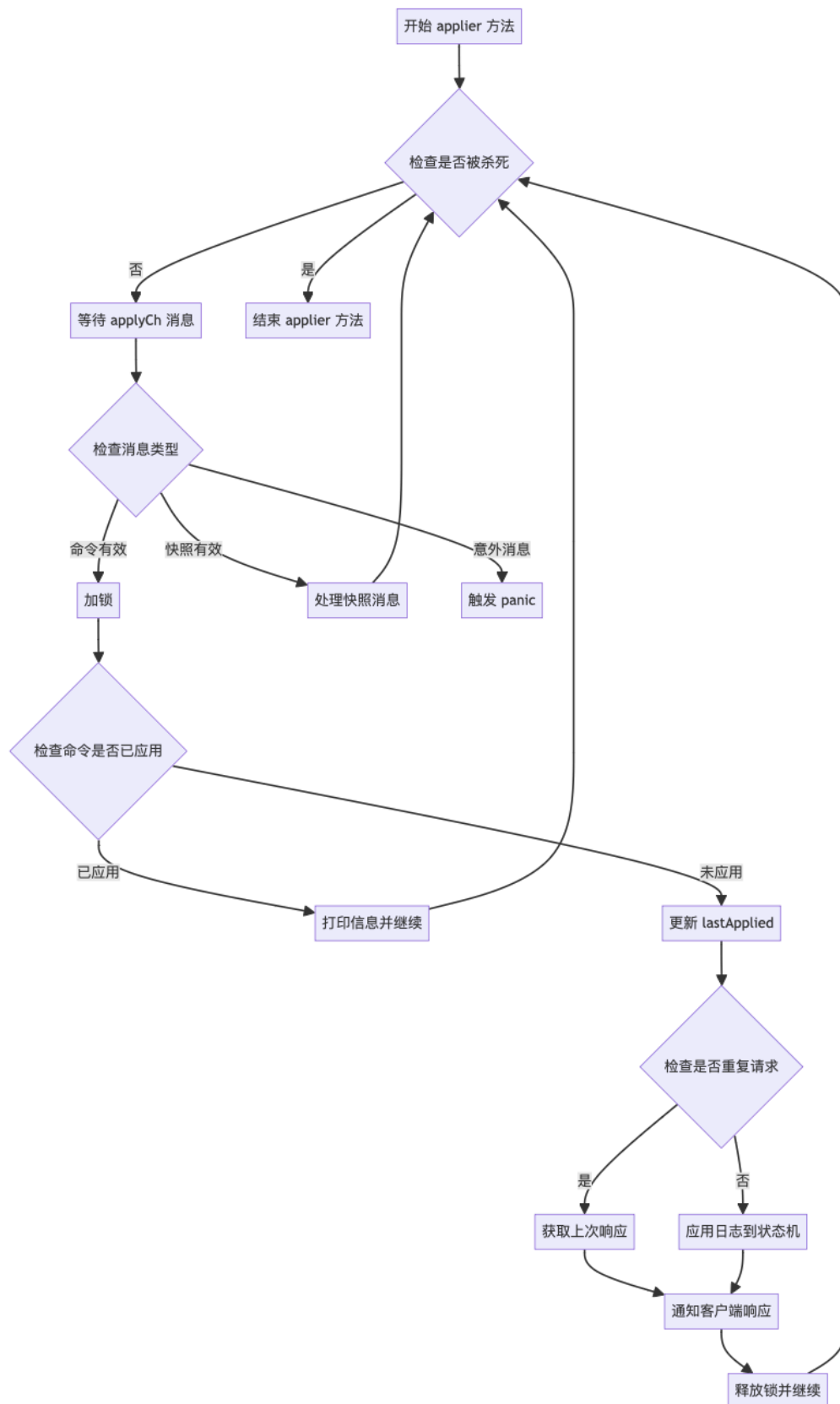


图 2-6 applier 执行流程图

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$ go test -run 3A -race
Test: one client (3A) ...
... Passed -- 15.1 5 8264 913
Test: ops complete fast enough (3A) ...
Test: many clients (3A) ...
... Passed -- 15.5 5 9773 1187
Test: unreliable net, many clients (3A) ...
... Passed -- 16.2 5 5334 906
Test: concurrent append to same key, unreliable (3A) ...
... Passed -- 1.8 3 245 52
Test: progress in majority (3A) ...
... Passed -- 1.3 5 128 2
Test: no progress in minority (3A) ...
... Passed -- 1.0 5 193 3
Test: completion after heal (3A) ...
... Passed -- 1.0 5 51 3
Test: partitions, one client (3A) ...
... Passed -- 22.9 5 14589 785
Test: partitions, many clients (3A) ...
... Passed -- 23.2 5 25916 1087
Test: restarts, one client (3A) ...
... Passed -- 22.0 5 18722 870
Test: restarts, many clients (3A) ...
... Passed -- 22.9 5 24752 1166
Test: unreliable net, restarts, many clients (3A) ...
... Passed -- 24.3 5 6767 891
Test: restarts, partitions, many clients (3A) ...
... Passed -- 29.8 5 29361 1046
Test: unreliable net, restarts, partitions, many clients (3A) ...
... Passed -- 30.0 5 5881 876
Test: unreliable net, restarts, partitions, random keys, many clients (3A) ...
... Passed -- 33.4 7 13429 947
PASS
ok      6.824/kvraft      274.002s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$
```

图 2-7 3A 部分测试

3 Part 3B: Key/value service with snapshots

3.1 基本准备工作

现在我们先解读一下 3B 工作和 3A 工作的区别，然后再理解 2D 在 3B 中的调用，然后再开始写代码。

就目前情况而言，现在 3A 的 kv 服务器不会调用 Raft 库的 ‘Snapshot()’ 方法，因此重新启动的服务器必须重播完整的持久 Raft 日志才能恢复其状态。现在，您将使用 Lab 2D 中的 Raft 的 ‘Snapshot()’ 修改 kvserver 以与 Raft 配合以节省日志空间并减少重新启动时间。

所以现在我们应先完成 2D 的工作。

3.2 raft 的 2D 部分编写：Part 2D: log compaction

3.2.1 准备工作

首先我们要了解为什么需要 2D？

按照目前的情况，重新启动的服务器会重放完整的 Raft 日志以恢复其状态。然而，对于一个长期运行的服务来说，永远记住完整的 Raft 日志是不切实际的。相反，您将修改 Raft 以与不时持久存储其状态“快照”的服务配合，此时 Raft 会丢弃快照之前的日志条目。结果是持久数据量更小，重启速度更快。然而，现在追随者可能会落后得太远，以至于领导者已经丢弃了它需要赶上的日志条目；然后，领导者必须发送快照以及从快照时开始的日志。扩展 Raft 论文第 7 节概述了该方案。

2D 部分工作流程图如图 3-1 所示。

通过学习 mit 的实验手册，网址为：<http://nil.csail.mit.edu/6.824/2022/labs/lab-raft.html>。我可以整理出整个实验过程的函数调用逻辑，如图 3-2 所示。

3.2.2 编写 Snapshot 函数

Snapshot 函数：在 Lab 2D 中，测试器会定期调用 Snapshot() 函数。在 Lab 3 中，您将编写一个键/值服务器，该服务器会调用 Snapshot()；快照将包含完整的键/值对表。服务层会在每个节点（不仅仅是领导者）上调用 Snapshot()。

当服务层决定创建快照以减少日志条目的数量时，这个函数被调用。

具体实现如代码所示。

language

```
// the service says it has created a snapshot that has
```

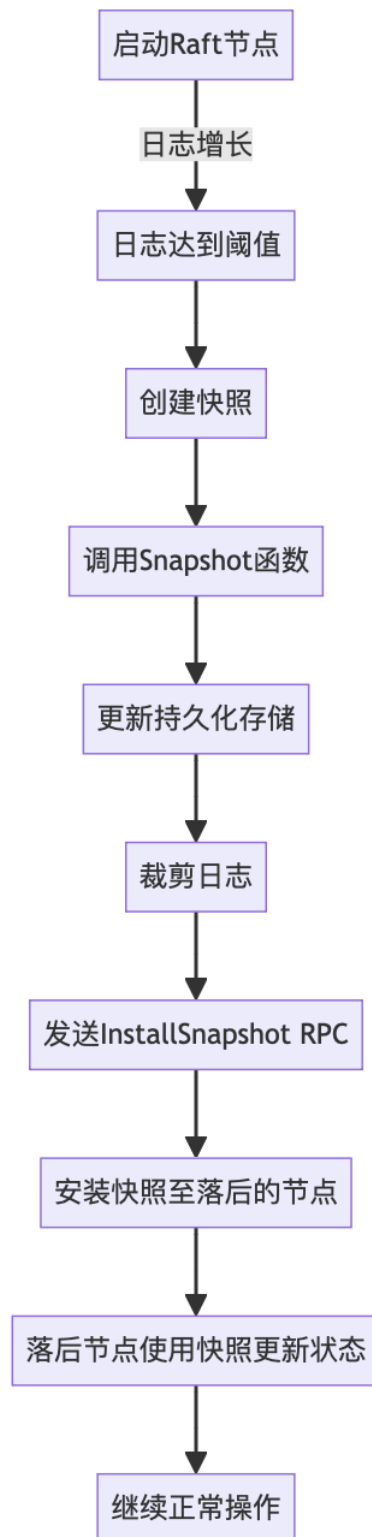


图 3-1 2D 执行流程

```
// all info up to and including index. this means the  
// service no longer needs the log through (and including)
```

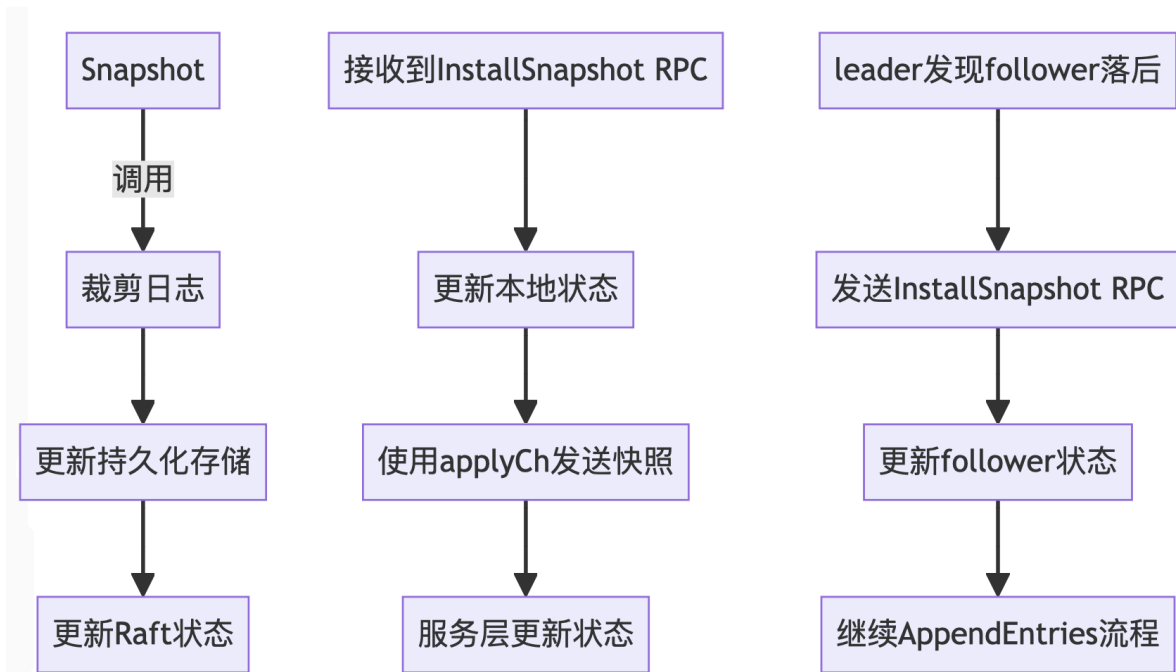


图 3-2 2D 中函数的调用逻辑

```
// that index. Raft should now trim its log as much as possible.
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    snapshotIndex := rf.getFirstLog().Index // 这里获取了当前日志中的第一个条目的索引。这是为了确定快照
    // 覆盖的日志部分
    if index <= snapshotIndex {
        /*
            这个条件检查确保传入的快照索引（即快照中包含的最后一个日志条目的索引）
            大于当前的第一个日志条目的索引。如果不是这样，函数会打印一条调试信息并返回。
        */
        DPrintf("{Node_%v} rejects replacing log with snapshotIndex_%v as current snapshotIndex_%v
            is larger in term_%v", rf.me, index, snapshotIndex, rf.currentTerm)
        return
    }
    // 这行代码裁剪日志 它保留从快照索引之后开始的所有日志条目，从而删除了较旧的、已经包含在快照中的
    // 日志条目
    rf.logs = shrinkEntriesArray(rf.logs[index-snapshotIndex:])
    // 这行代码将裁剪后的日志的第一个条目的命令字段设置为nil。这是为了标记这个条目现在作为新日志的起
    // 点
    rf.logs[0].Command = nil
    // 将Raft的当前状态和新的快照数据持久化存储
    // rf.encodeState()方法将Raft的状态（如当前任期和已投票的候选者）编码为字节流，以便持久化
}
```

```

rf.persister.SaveStateAndSnapshot(rf.encodeState(), snapshot)
DPrintf("{Node_%v}'s state is {state_%v, term_%v, commitIndex_%v, lastApplied_%v, firstLog_%v,
      lastLog_%v} after replacing log with snapshotIndex_%v as old snapshotIndex_%v is smaller", rf.
      me, rf.state, rf.currentTerm, rf.commitIndex, rf.lastApplied, rf.getFirstLog(), rf.getLastLog(), index,
      snapshotIndex)
}

```

其中, `rf.logs[0].Command = nil`, 这一句话, 是为了记录裁剪的位置, 将裁减后的第一个条目的命令字段设置为 `nil`, 这里是一个特殊设置, 这里是为了标记这个条目现在作为新日志的起点。

3.2.3 实现 InstallSnapshot RPC

根据论文, 我们需要实现 InstallSnapshot RPC, 允许 Raft 领导者告诉落后的 Raft 节点用快照替换其状态。我们可能需要思考 ‘InstallSnapshot’ 如何与图 2 中的状态和规则交互。

对于论文中 InstallSnapshotRPC 的解读如图 3-3 所示。

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)	

InstallSnapshot RPC 字段解读:

- **term**: 领导者的任期号
- **leaderId**: 发起请求的领导者ID
- **LastIncludedIndex**: 快照包含的最后日志条目的索引
- **LastIncludedTerm**: 快照包含的最后日志条目的任期号
- **Data**: 快照数据 (字节流类型)

回应结构体字段就只有 **Term**: 表示当前节点的任期号

图 3-3 InstallSnapshotRPC 的解读

代码如下。

```

language
// 用于安装快照的RPC请求
type InstallSnapshotRequest struct {
    Term          int    // 领导者的任期号
    LeaderId      int    // 发起请求的领导者ID
    LastIncludedIndex int  // 快照包含的最后日志条目的索引
    LastIncludedTerm int  // 快照包含的最后日志条目的任期号
    Data          []byte // 快照数据 (字节流类型)
}

```



```

}
// 用字符串可视化一个InstallSnapshotRequest
func (request InstallSnapshotRequest) String() string {
    return fmt.Sprintf("{Term:%v,LeaderId:%v,LastIncludedIndex:%v,LastIncludedTerm:%v,DataSize:%v}",
        request.Term, request.LeaderId, request.LastIncludedIndex, request.LastIncludedTerm, len(request.
            Data))
}
type InstallSnapshotResponse struct {
    Term int // 当前节点的任期号
}
// 用字符串可视化一个InstallSnapshotResponse
func (response InstallSnapshotResponse) String() string {
    return fmt.Sprintf("{Term:%v}", response.Term)
}

```

3.2.4 处理 InstallSnapshot RPC

当一个跟随者的 Raft 代码接收到 InstallSnapshot RPC 时，它可以使用 applyCh 将快照作为 ApplyMsg 发送给服务层。ApplyMsg 结构定义已包含您需要的字段（以及测试器期望的字段）。请确保这些快照只推进服务的状态，而不会导致它倒退。

处理 InstallSnapshot RPC 并将快照通过 applyCh 发送给服务层的任务是在 InstallSnapshot 函数中完成的。这个函数处理来自领导者的 InstallSnapshot 请求，并更新本地节点的状态以及将快照数据发送给上层服务。

InstallSnapshot 需要做的事情有：

1. 首先判断，这个 InstallSnapshot 是否被允许，如果发起 InstallSnapshot 的领导者是老领导，这个操作不被允许。
2. 如果 InstallSnapshot 的领导者是一个比我新的领导者，那就要更新自己的任期号了，因为这个情况说明我自己落后了，更新了之后记得要永久存储一下。
3. 然后更改自己的节点状态为跟随者
4. 重新设置计时器，避免当前节点在接收快照期间发起不必要的选举
5. 如果请求中包含的最后日志索引小于或等于当前节点的提交索引，则返回
6. 创建一个新的 goroutine 来异步发送快照信息到服务层

language

```

func (rf *Raft) InstallSnapshot(request *InstallSnapshotRequest, response *InstallSnapshotResponse) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer DPrintf("{Node:%v}'s state is {state:%v,term:%v,commitIndex:%v,lastApplied:%v,firstLog:%v,
        lastLog:%v} before processing InstallSnapshotRequest %v and reply InstallSnapshotResponse %v

```

```

    ", rf.me, rf.state, rf.currentTerm, rf.commitIndex, rf.lastApplied, rf.getFirstLog(), rf.getLastLog(),
    request, response)
response.Term = rf.currentTerm // 设置响应中的任期号为当前节点的任期号
if request.Term < rf.currentTerm {
    // 如果请求中的任期号小于当前节点的任期号, 则直接返回。这表示请求来自一个过时的领导者
    // 过时的领导者是肯定不允许InstallSnapshot
    return
}
if request.Term > rf.currentTerm {
    /*
        如果请求中的任期号大于当前节点的任期号, 则更新当前节点的任期号, 并重置投票信息。
        这表示当前节点落后于集群中的其他节点
    */
    rf.currentTerm, rf.votedFor = request.Term, -1
    rf.persist()
}
rf.ChangeState(StateFollower) // 将当前节点的状态更改为跟随者
rf.electionTimer.Reset(RandomizedElectionTimeout()) // 重置选举计时器, 避免当前节点在接收快照期间
    发起不必要的选举
// outdated snapshot
if request.LastIncludedIndex <= rf.commitIndex {
    // 如果请求中包含的最后日志索引小于或等于当前节点的提交索引, 则返回
    // 这表示当前节点已经有了请求中的所有信息
    return
}
}
go func() {
    // 创建一个新的goroutine来异步发送快照信息到服务层
    rf.applyCh <- ApplyMsg{
        SnapshotValid: true,
        Snapshot:      request.Data,
        SnapshotTerm: request.LastIncludedTerm,
        SnapshotIndex: request.LastIncludedIndex,
    }
}()
}

```

3.2.5 服务器重启时的状态恢复

当服务器重启时, 应用层会读取持久化的快照, 并恢复其保存的状态。

关于这部分, 我已经在 Snapshot 函数里面有所体现了, 我在 Snapshot 的最后部分调用了 rf.persister.SaveStateAndSnapshot 来进行快照状态的存储。

3.2.6 一次性发送完整快照

在单个 InstallSnapshot RPC 中发送整个快照。不要实现论文图 13 中用于分割快照的偏移机制。

这一部分的要求，我们在 InstallSnapshotRequest 部分已经有体现了，在 InstallSnapshotRequest 结构体中：

1. Data 字段包含了要发送的完整快照数据。当这个 RPC 请求被发送时，接收者（跟随者节点）会一次性收到整个快照数据，而不是分片或偏移量形式的数据。

2. 这种方式符合 Raft 论文中关于快照传输的简化处理，即一次性发送整个快照，不使用论文图 13 中提到的偏移机制。通过这种方法，可以简化处理过程，同时确保落后的节点能够接收并应用完整的快照来更新其状态。

3.2.7 丢弃旧日志条目

Raft 必须以一种方式丢弃旧日志条目，使得 Go 垃圾回收器能够释放并重用内存；这要求没有可达引用（指针）指向被丢弃的日志条目。

是通过 Snapshot 函数中对日志数组的处理实现的。具体来说，这是在裁剪日志条目时完成的。

代码片段如下所示。

```
language
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    snapshotIndex := rf.getFirstLog().Index

    if index <= snapshotIndex {
        return
    }

    rf.logs = shrinkEntriesArray(rf.logs[index-snapshotIndex:])
    rf.logs[0].Command = nil
    rf.persister.SaveStateAndSnapshot(rf.encodeState(), snapshot)
}
```

在 Snapshot 函数中，裁剪日志条目的关键在于以下几行代码：

1. `rf.logs = shrinkEntriesArray(rf.logs[index-snapshotIndex:])`: 这行代码使用自定义的 `shrinkEntriesArray` 函数（假设这是一个实现了适当逻辑的函数）来裁剪 `rf.logs` 数组，只保留从给定索引开始的日志条目。这样做的结果是移除了所有早于快照索引的日志条目。

2. `rf.logs[0].Command = nil`: 为了保证日志的连续性，第一个日志条目（现在是裁剪后的日志的第一个条目）的命令被设置为 `nil`。这通常是一个虚拟的占位条目，用于维持日志的索引连续性。

通过裁剪 `rf.logs` 数组并更新其内容，您的代码有效地移除了对旧日志条目的引用。由于没有其他部分的代码持有这些被裁剪日志条目的引用，Go 的垃圾回收器就可以安全地回收这部分内存。这种处理方式符合 Raft 算法对日志压缩的要求，并确保了内存的有效管理。

3.2.8 实现日志被修剪后的 `AppendEntries`

即使日志被修剪，实现仍然需要在 `AppendEntries` RPC 中正确发送新条目之前的条目的任期和索引；这可能需要保存并引用最新快照的 `lastIncludedTerm/lastIncludedIndex`（考虑是否应该持久化这些信息）。

`AppendEntries` 函数：

在 `AppendEntries` 函数中，我需要考虑日志可能已经被部分修剪的情况。当领导者发送 `AppendEntries` 请求时，请求中包含的 `PrevLogIndex` 和 `PrevLogTerm` 字段是用来确保日志的一致性。

修改后的代码片段如下所示。

```
language
func (rf *Raft) AppendEntries(request *AppendEntriesRequest, response *AppendEntriesResponse) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    // ...
    // 检查 PrevLogIndex 是否在快照中或之后
    if request.PrevLogIndex < rf.getFirstLog().Index {
        // 特殊处理：可能需要发送快照信息
        // ...
        return
    }

    // 正常处理 AppendEntries
    // ...
}
```

这段代码中，当 `PrevLogIndex` 小于当前日志的第一个条目的索引时，这意味着请求的日志条目已经被包含在快照中，或者日志已经被修剪。这时，我需要根据快照中的数据来响应，或者指示领导者发送快照（通过 `InstallSnapshotRPC`）。

3.2.9 快照信息的引用

在日志修剪时更新快照信息：

```
language
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    // ...

    // 更新快照信息
    rf.lastIncludedIndex = index
    rf.lastIncludedTerm = rf.logs[index].Term
    rf.logs = shrinkEntriesArray(rf.logs[index:])

    // 持久化快照信息
    rf.persist()
}
```

lastIncludedIndex 和 lastIncludedTerm 在创建快照时被更新，并通过 persist() 函数持久化存储。这确保了即使在日志被修剪后，AppendEntries 逻辑仍然能够正确地引用最新的快照信息。

3.2.10 编写 CondInstallSnapshot 函数

CondInstallSnapshot 函数用于处理安装快照的逻辑。当 Raft 节点接收到包含状态快照的请求时，这个函数负责决定是否接受这个快照并据此更新节点的状态。快照包含了 Raft 日志的压缩形式，用于在日志条目过多时减少存储和加快恢复速度。

```
language
func (rf *Raft) CondInstallSnapshot(lastIncludedTerm int, lastIncludedIndex int, snapshot []byte) bool {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    DPrintf("{Node_%v}_service_calls_CondInstallSnapshot_with_lastIncludedTerm_%v_and_
        lastIncludedIndex_%v_to_check_whether_snapshot_is_still_valid_in_term_%v", rf.me,
        lastIncludedTerm, lastIncludedIndex, rf.currentTerm)
    // outdated snapshot
    if lastIncludedIndex <= rf.commitIndex {
        // 检查快照是否过时。如果快照中包含的最后索引小于或等于已提交索引，则认为快照是过时的，拒绝
        接受
        DPrintf("{Node_%v}_rejects_the_snapshot_which_lastIncludedIndex_is_%v_because_commitIndex_%
            v_is_larger", rf.me, lastIncludedIndex, rf.commitIndex)
        return false
    }
}
```

```

/*
    如果快照的最后索引大于当前日志的最后索引，则清空整个日志并创建一个新条目；
    否则，保留快照索引之后的日志条目，并裁剪之前的部分
*/
*/
if lastIncludedIndex > rf.getLastLog().Index {
    rf.logs = make([]Entry, 1)
} else {
    // 清空日志数组的第一个条目的命令字段，因为这个条目现在作为新的日志起点
    rf.logs = shrinkEntriesArray(rf.logs[lastIncludedIndex-rf.getFirstLog().Index:])
    rf.logs[0].Command = nil
}
// update dummy entry with lastIncludedTerm and lastIncludedIndex
// 更新日志数组的第一个条目的任期和索引为快照中的值
rf.logs[0].Term, rf.logs[0].Index = lastIncludedTerm, lastIncludedIndex
// 更新节点的lastApplied和commitIndex为快照的最后索引
rf.lastApplied, rf.commitIndex = lastIncludedIndex, lastIncludedIndex
// 将节点的当前状态和快照数据持久化存储
rf.persister.SaveStateAndSnapshot(rf.encodeState(), snapshot)
DPrintf("{Node_%v}'s state is {state_%v, term_%v, commitIndex_%v, lastApplied_%v, firstLog_%v,
    lastLog_%v} after accepting the snapshot which lastIncludedTerm is %v, lastIncludedIndex is %v",
    rf.me, rf.state, rf.currentTerm, rf.commitIndex, rf.lastApplied, rf.getFirstLog(), rf.getLastLog(),
    lastIncludedTerm, lastIncludedIndex)
return true
}

```

3.2.11 编写网络调用

这个也要记得写一下，就是 Snapshot 的那些 RPC，需要网络调用的。

编写逻辑和之前 2A-2C 的 sendAppendEntries 和 sendRequestVote 是一样的。

```

language
func (rf *Raft) sendInstallSnapshot(server int, request *InstallSnapshotRequest, response *
    InstallSnapshotResponse) bool {
    return rf.peers[server].Call("Raft.InstallSnapshot", request, response)
}

```

3.2.12 完善 replicateOneRound 函数

用于处理将日志条目（或快照）复制到一个指定的同伴节点（peer）的逻辑。这个函数根据同伴节点的状态决定是发送日志条目还是快照，并处理相应的响应。

在 2A-2C 中，这个函数只需要将日志条目复制到一个指定的同伴节点中即可，现在还需要用于把快照复制到一个指定同伴节点中。

这个函数的流程如图 3-4 所示。

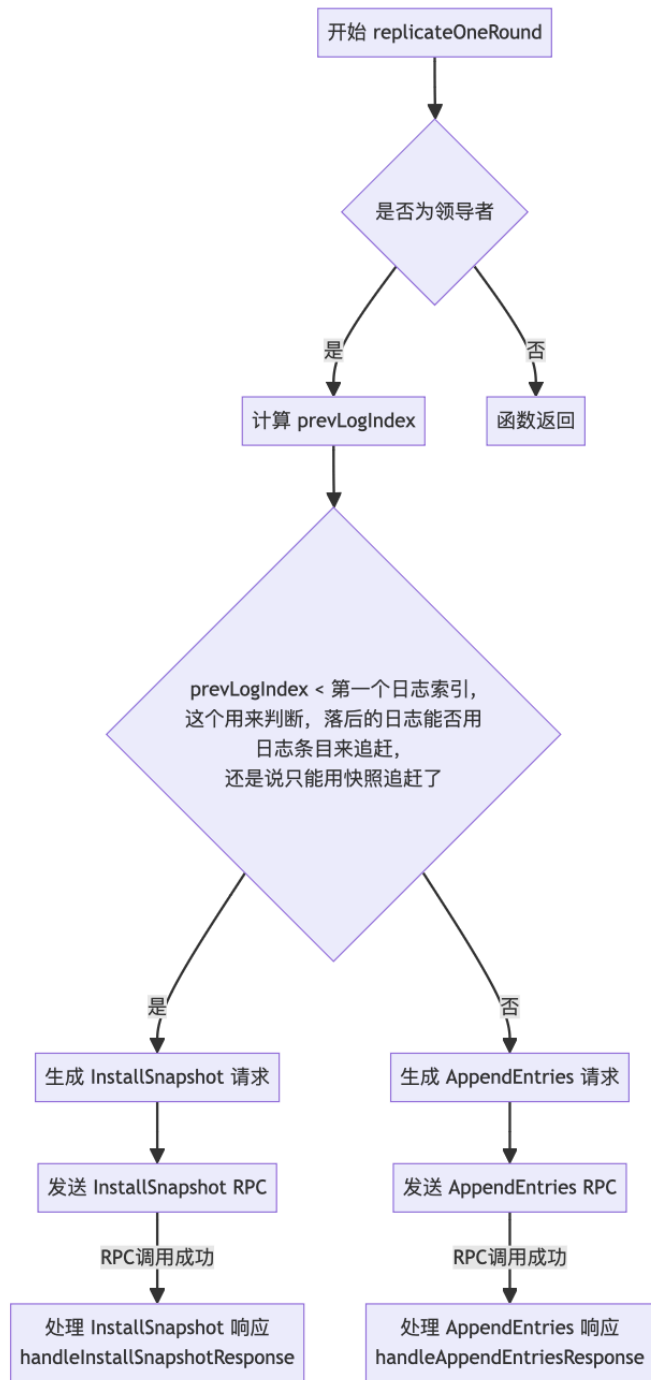


图 3-4 replicateOneRound 执行流程

代码如下所示。

language

```

func (rf *Raft) replicateOneRound(peer int) {
    rf.mu.RLock()
    if rf.state != StateLeader {
        rf.mu.RUnlock()
        return
    }
    // 计算要发送给同伴节点的下一个日志条目的前一个索引, nextIndex数组记录了每个同伴节点需要接收的下一个日志条目的索引
    prevLogIndex := rf.nextIndex[peer] - 1
    // 判断同伴节点是否需要快照来追赶。如果prevLogIndex小于当前日志的第一个条目的索引, 表示同伴节点落后得太多, 需要发送快照
    if prevLogIndex < rf.getFirstLog().Index {
        // only snapshot can catch up
        // 这里需要快照去追赶, 因为落后的太多了
        request := rf.genInstallSnapshotRequest()
        rf.mu.RUnlock()
        // 创建一个用于接收RPC响应的新对象
        response := new(InstallSnapshotResponse)
        // 发送快照安装RPC。如果RPC调用成功, 处理响应
        if rf.sendInstallSnapshot(peer, request, response) {
            rf.mu.Lock()
            rf.handleInstallSnapshotResponse(peer, request, response)
            rf.mu.Unlock()
        }
    } else {
        // just entries can catch up
        // 这里用日志就能完成同步, 不需要用快照追赶
        // 生成一个AppendEntries请求
        request := rf.genAppendEntriesRequest(prevLogIndex)
        rf.mu.RUnlock()
        // 创建一个接收响应的新对象
        response := new(AppendEntriesResponse)
        // 响应成功后就去处理响应
        if rf.sendAppendEntries(peer, request, response) {
            rf.mu.Lock()
            rf.handleAppendEntriesResponse(peer, request, response)
            rf.mu.Unlock()
        }
    }
}

```


3.2.13 编写 genInstallSnapshotRequest 函数

这个函数在很多地方都可能会用到，就是用来生成一个 InstallSnapshotRequest 的对象，思路很简单，直接写。

```
language
func (rf *Raft) genInstallSnapshotRequest() *InstallSnapshotRequest {
    firstLog := rf.getFirstLog()
    return &InstallSnapshotRequest{
        Term:          rf.currentTerm,
        LeaderId:       rf.me,
        LastIncludedIndex: firstLog.Index,
        LastIncludedTerm: firstLog.Term,
        Data:           rf.persister.ReadSnapshot(),
    }
}
```

3.2.14 编写 handleInstallSnapshotResponse 函数

handleInstallSnapshotResponse 函数，收到 InstallSnapshotResponse 之后，我应该怎么处理这个 resp。

流程如图 3-5 所示。

代码如下所示。

```
language
func (rf *Raft) handleInstallSnapshotResponse(peer int, request *InstallSnapshotRequest, response *
InstallSnapshotResponse) {
    // 检查当前节点是否仍是领导者，并且请求中的任期号与当前节点的任期号匹配。只有在这些条件满足时，
    // 才继续处理响应
    if rf.state == StateLeader && rf.currentTerm == request.Term {
        // 如果响应中的任期号大于当前节点的任期号，说明当前领导者已经落后，需要更新其状态
        if response.Term > rf.currentTerm {
            rf.ChangeState(StateFollower) // 将当前节点的状态更改为跟随者
            // 更新当前节点的任期号为响应中的任期号，并重置投票状态
            rf.currentTerm, rf.votedFor = response.Term, -1
            // 持久化当前节点的状态
            rf.persist()
        } else {
            // 如果响应中的任期号不大于当前节点的任期号，则说明当前领导者仍然有效
            // 更新与该同伴节点相关的matchIndex和nextIndex。这表示领导者认为该同伴至少拥有到
            LastIncludedIndex的所有日志。
        }
    }
}
```

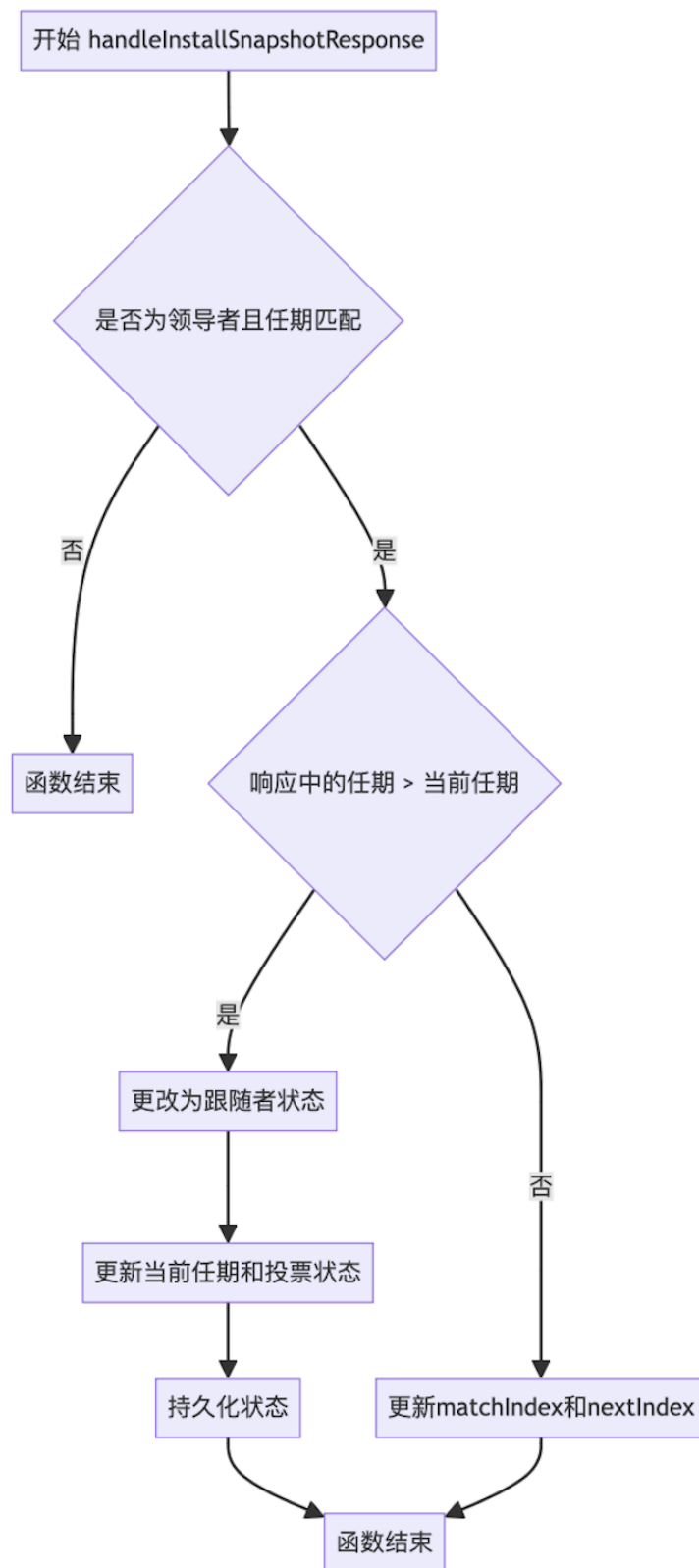


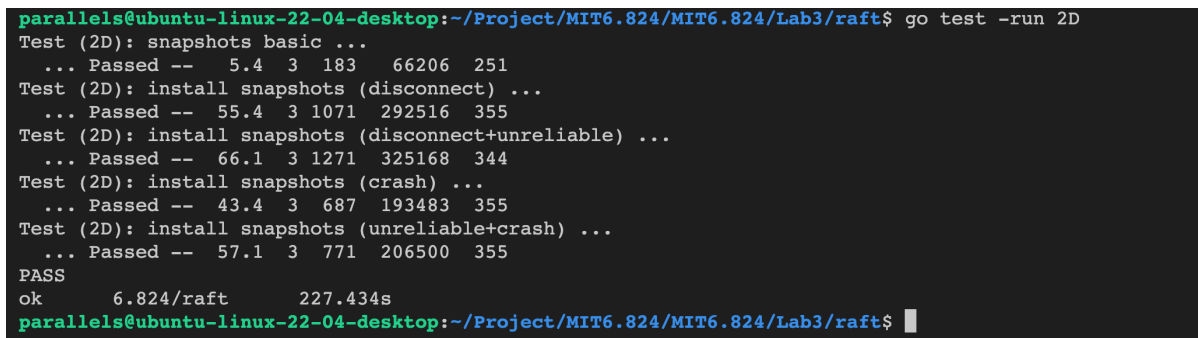
图 3-5 `handleInstallSnapshotResponse` 函数执行流程

```

        rf.matchIndex[peer], rf.nextIndex[peer] = request.LastIncludedIndex, request.LastIncludedIndex
            +1
    }
}
DPrintf("{Node_%v}'s state is {state_%v, term_%v, commitIndex_%v, lastApplied_%v, firstLog_%v,
lastLog_%v} after handling InstallSnapshotResponse_%v for InstallSnapshotRequest_%v", rf.me,
rf.state, rf.currentTerm, rf.commitIndex, rf.lastApplied, rf.getFirstLog(), rf.getLastLog(), response,
request)
}

```

3.2.15 2D 测试



```

parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/raft$ go test -run 2D
Test (2D): snapshots basic ...
... Passed -- 5.4 3 183 66206 251
Test (2D): install snapshots (disconnect) ...
... Passed -- 55.4 3 1071 292516 355
Test (2D): install snapshots (disconnect+unreliable) ...
... Passed -- 66.1 3 1271 325168 344
Test (2D): install snapshots (crash) ...
... Passed -- 43.4 3 687 193483 355
Test (2D): install snapshots (unreliable+crash) ...
... Passed -- 57.1 3 771 206500 355
PASS
ok      6.824/raft      227.434s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/raft$

```

图 3-6 2D 测试

如图 3-6 所示，可以通过所有的测试。

3.3 3B 任务解读

通过实验手册可以得出：

测试器将 `maxraftstate` 传递给您的 `StartKVServer()`。`maxraftstate` 指示持久 Raft 状态允许的最大大小(以字节为单位)(包括日志,但不包括快照)。您应该将 `maxraftstate` 与 `persister.RaftStateSize()` 进行比较。每当你的键/值服务器检测到 Raft 状态大小接近这个阈值时,它应该通过调用 Raft 的 `Snapshot` 来保存快照。如果 `maxraftstate` 为 -1, 则不必创建快照。`maxraftstate` 适用于 Raft 传递给 `persister.SaveRaftState()` 的 GOB 编码字节。

通过理解这个任务可以得出结论：

1. 客户端部分 (`client.go`) 部分是不用做出任何更改的, 因为快照只和服务端有关, 对用户是无感知的。
2. 相关 `rpc` 工具部分 (`common.go`) 是不用做出任何更改的, 因为 `common.go` 部分也算是服务端和客户端交互部分的代码, 因此也不用修改。
3. 所以最后只需要修改 `server.go` 即可。

在 3B 中, KVServer 实现了快照功能, 以处理日志的持续增长问题。当 Raft 日志达到一定大小时, 系统会生成快照, 并且这个快照会包含当前键值存储的状态。这样做可以减少日志的大小, 提高系统性能。3B 工作流程如图 3-7 所示。

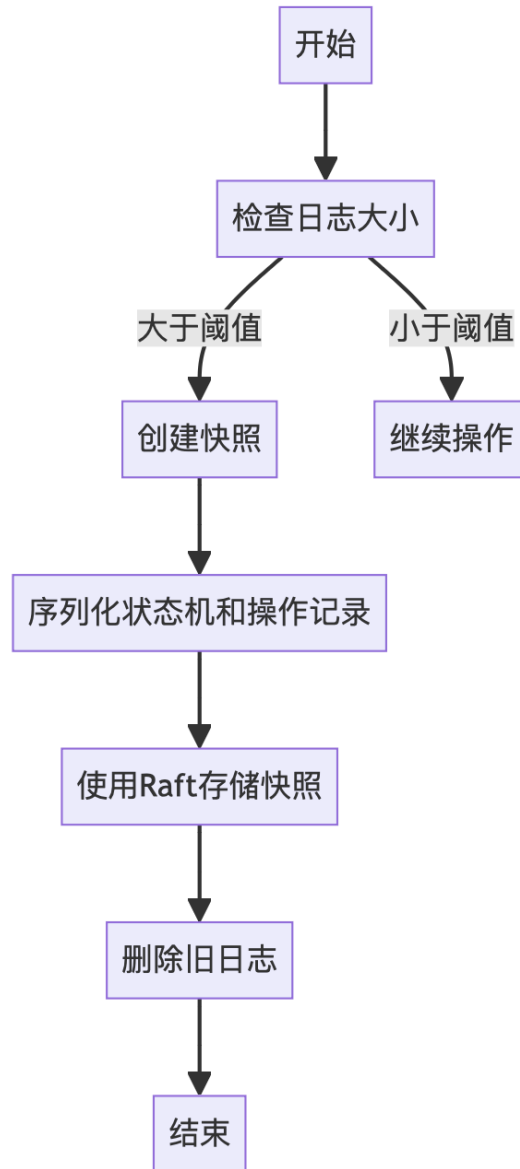


图 3-7 3B 工作流程

经过完善之后, 我可以写出一个 Lab3KV 服务器的完整调用逻辑图 3-8 了, 如图所示。这里包含了 3A-3B 的所有内容。

下面我将会在 3A 的基础上, 完善 server.go, 一步步完成 3B 的内容。

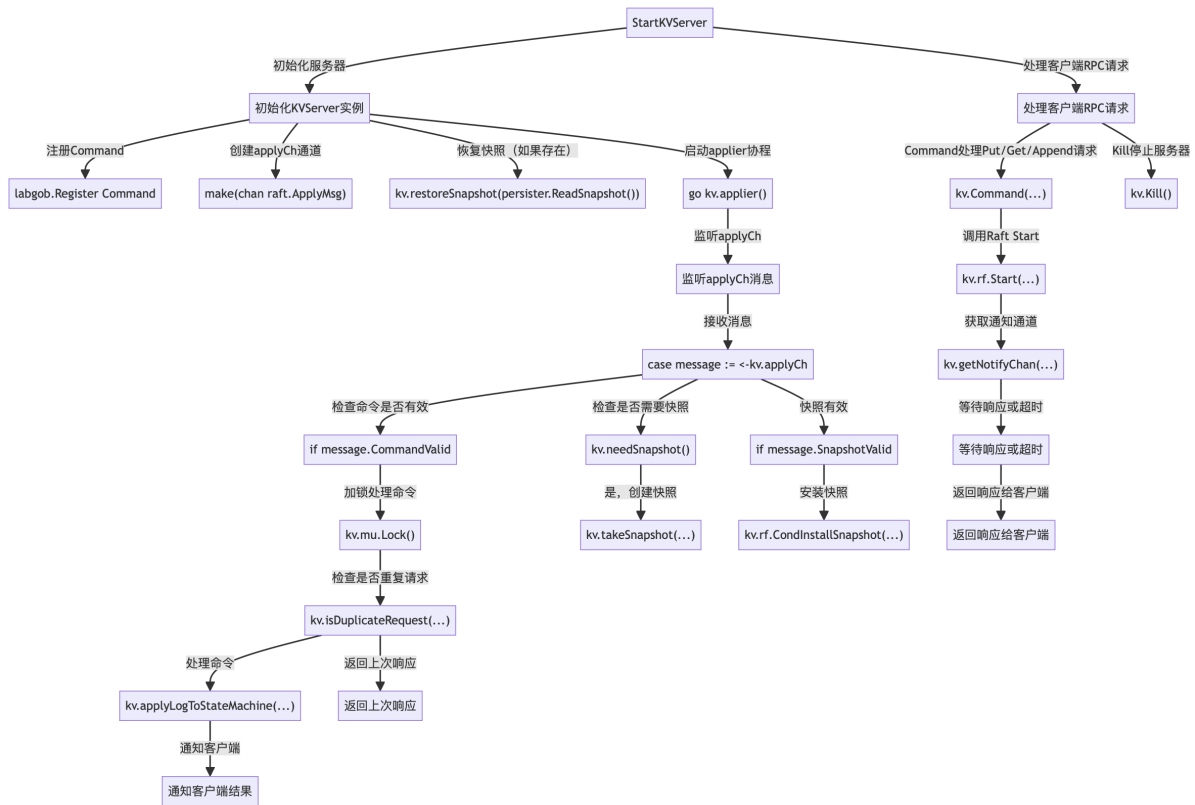


图 3-8 Lab3KV 服务器整体调用逻辑（包括 3A 和 3B）

3.4 完善 StartKVServer 函数

在 StartKVServer 函数中，3B 版本相比于 3A 版本多出的这行代码 kv.restoreSnapshot(persister.ReadSnapshot()) 执行了一个非常关键的操作：它从持久化存储中恢复之前保存的快照状态。

1. **读取快照**: persister.ReadSnapshot(): 这个调用从 persister（一个持久化存储接口）中读取最后保存的快照数据。Persister 对象通常负责管理 Raft 日志的持久化存储，包括快照数据。2. **恢复状态机**: kv.restoreSnapshot(...): 这个方法接收读取的快照数据作为参数，并恢复 KVServer 的状态机到快照保存时的状态。如果存在有效的快照数据，这个操作将确保 KVServer 在重新启动后能够从上次运行时的状态开始，而不是从初始状态开始。这对于保持分布式系统的持久性和一致性至关重要。

3. **处理无快照的情况**: 如果 persister.ReadSnapshot() 返回的快照为空或不存在，restoreSnapshot 方法将不会改变当前的状态机状态。这意味着 KVServer 将以初始状态开始。

完善后的 StartKVServer 函数完整代码如下所示。

```
language
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.

/*
    这行代码使用 labgob 包注册 Command 结构体,
    以便在 Go 的 RPC 系统中进行序列化和反序列化。这是分布式系统中节点间通信所必需的
*/
labgob.Register(Command{})
// 创建一个新的通道 applyCh, 用于接收来自 Raft 层的 ApplyMsg 消息
applyCh := make(chan raft.ApplyMsg)

// 实例化一个KVServer, 里面的字段也没啥好说的, 就是按照设计的来初始化即可
kv := &KVServer{
    maxRaftState: maxraftstate,
    applyCh:      applyCh,
    dead:         0,
    lastApplied:  0,
    rf:           raft.Make(servers, me, persister, applyCh),
    stateMachine: NewMemoryKV(),
    lastOperations: make(map[int64]OperationContext),
    notifyChans:   make(map[int]chan *CommandResponse),
}
kv.restoreSnapshot(persister.ReadSnapshot()) // 恢复之前的快照状态
// start applier goroutine to apply committed logs to stateMachine
go kv.applier() // 启动applier协程

DPrintf("{Node_%v}_has_started", kv.rf.Me())
return kv
}

```

3.5 完善 applier 函数

这个函数对比 3A 的多做了以下事情:

1. 检查和应用快照:

1. 在接收到的消息中检查是否包含有效的快照信息。
2. 如果存在有效快照, 则使用 CondInstallSnapshot 方法检查并安装这个快照。
3. 使用 restoreSnapshot 方法恢复快照中的状态机状态。

2. 判断是否需要创建快照:

1. 在处理完每个命令后, 检查当前的 Raft 状态是否达到了需要创建快照的条件。

2. 如果需要，调用 takeSnapshot 方法创建新的快照并更新 Raft 的状态。
整理过后 3B 的 applier 函数流程如图 3-9 所示。

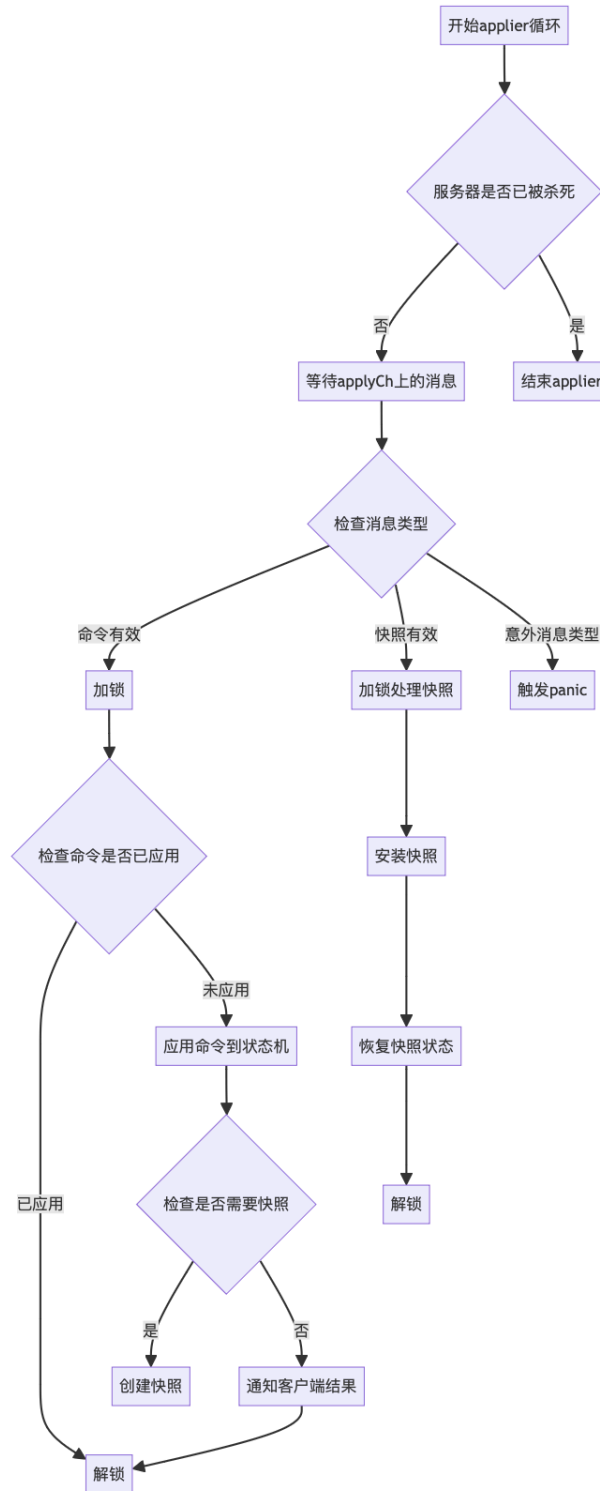


图 3-9 调整过后的 applier 流程

整理过后的 applier 函数代码如下所示。

```

language
// a dedicated applier goroutine to apply committed entries to stateMachine, take snapshot and apply
// snapshot from raft
// 用于应用来自 Raft 层的日志条目到键值存储的状态机
func (kv *KVServer) applier() {
    // 只要服务器没有被杀死，就一直运行
    for kv.killed() == false {
        select {
            // 多路转接异步管理多个管道
            case message := <-kv.applyCh:
                // applyCh上是有来自raft的mesg的，如果有mesg来了，就进入到下面的函数中
                DPrintf("{Node%v}_tries_to_apply_message%v", kv.rf.Me(), message)
                if message.CommandValid {
                    // 检查接收到的消息是否包含有效的命令
                    kv.mu.Lock()
                    if message.CommandIndex <= kv.lastApplied {
                        // 检查命令索引是否已经被应用，以避免重复应用
                        DPrintf("{Node%v}_discards_outdated_message%v_because_a_newer_snapshot_which_
                            lastApplied_is%v_has_been_restored", kv.rf.Me(), message, kv.lastApplied)
                        kv.mu.Unlock()
                        continue
                    }
                    kv.lastApplied = message.CommandIndex // 更新最后应用的日志条目索引

                    var response *CommandResponse
                    command := message.Command.(Command) // 构建一个command
                    if command.Op != OpGet && kv.isDuplicateRequest(command.ClientId, command.
                        CommandId) {
                        // 判断是否为重复的命令请求
                        DPrintf("{Node%v}_doesn't_apply_duplicated_message%v_to_stateMachine_because_
                            maxAppliedCommandId_is%v_for_client%v", kv.rf.Me(), message, kv.
                                lastOperations[command.ClientId], command.ClientId)
                        response = kv.lastOperations[command.ClientId].LastResponse // 如果重复了就去获取旧
                            的指令
                    } else {
                        response = kv.applyLogToStateMachine(command) // 否则，应用日志到状态机
                        if command.Op != OpGet {
                            // 如果不是 Get 操作，更新 lastOperations 映射
                            kv.lastOperations[command.ClientId] = OperationContext{command.CommandId,
                                response}
                        }
                    }
                }
            }
        }
    }
}

```



```

    }

    // only notify related channel for currentTerm's log when node is leader
    if currentTerm, isLeader := kv.rf.GetState(); isLeader && message.CommandTerm ==
        currentTerm {
        // 如果当前节点是领导者并且命令属于当前任期
        ch := kv.getNotifyChan(message.CommandIndex)
        ch <- response
    }
    needSnapshot := kv.needSnapshot()
    if needSnapshot {
        kv.takeSnapshot(message.CommandIndex)
    }
    kv.mu.Unlock()
} else if message.SnapshotValid {
    kv.mu.Lock()
    if kv.rf.CondInstallSnapshot(message.SnapshotTerm, message.SnapshotIndex, message.
        Snapshot) {
        kv.restoreSnapshot(message.Snapshot)
        kv.lastApplied = message.SnapshotIndex
    }
    kv.mu.Unlock()
} else {
    // 处理意外情况
    panic(fmt.Sprintf("unexpected Message %v", message))
}
}
}
}

```

因此我们还需要编写三个和快照相关的工具函数：

1. needSnapshot
2. takeSnapshot
3. restoreSnapshot

3.6 完善快照相关工具函数

3.6.1 needSnapshot

作用：判断是否需要创建快照。

思路很简单，和手册说的一样，超过了大小限制（kv.maxRaftState）就需要快照，

否则就不需要。

```
language
func (kv *KVServer) needSnapshot() bool {
    return kv.maxRaftState != -1 && kv.rf.GetRaftStateSize() >= kv.maxRaftState
}
```

注意：-1 这个条件不要漏了，按照规则，设置为-1 的时候是强制不使用快照的。

3.6.2 takeSnapshot

作用：在给定的日志索引处创建快照。

思路如图 3-10 所示。

代码如下所示。

```
language
func (kv *KVServer) takeSnapshot(index int) {
    // 创建一个新的字节缓冲区，用于存储序列化后的快照数据
    w := new(bytes.Buffer)
    // 创建一个新的 labgob 编码器，将数据编码到前面创建的缓冲区 w
    e := labgob.NewEncoder(w)
    // 将当前的键值存储状态机 (stateMachine) 和最后操作的记录 (lastOperations) 序列化到缓冲区
    e.Encode(kv.stateMachine)
    e.Encode(kv.lastOperations)
    // 调用底层的Snapshot
    kv.rf.Snapshot(index, w.Bytes())
}
```

3.6.3 restoreSnapshot

作用：从给定的字节切片中恢复快照。

思路如图 3-11 所示。

代码如下所示。

```
language
func (kv *KVServer) restoreSnapshot(snapshot []byte) {
    // 检查传入的快照数据是否为空。如果为空，直接返回，不进行恢复操作
    if snapshot == nil || len(snapshot) == 0 {
        return
    }
    // 创建一个新的字节缓冲区，并将快照数据填充进去，准备进行反序列化
    r := bytes.NewBuffer(snapshot)
```

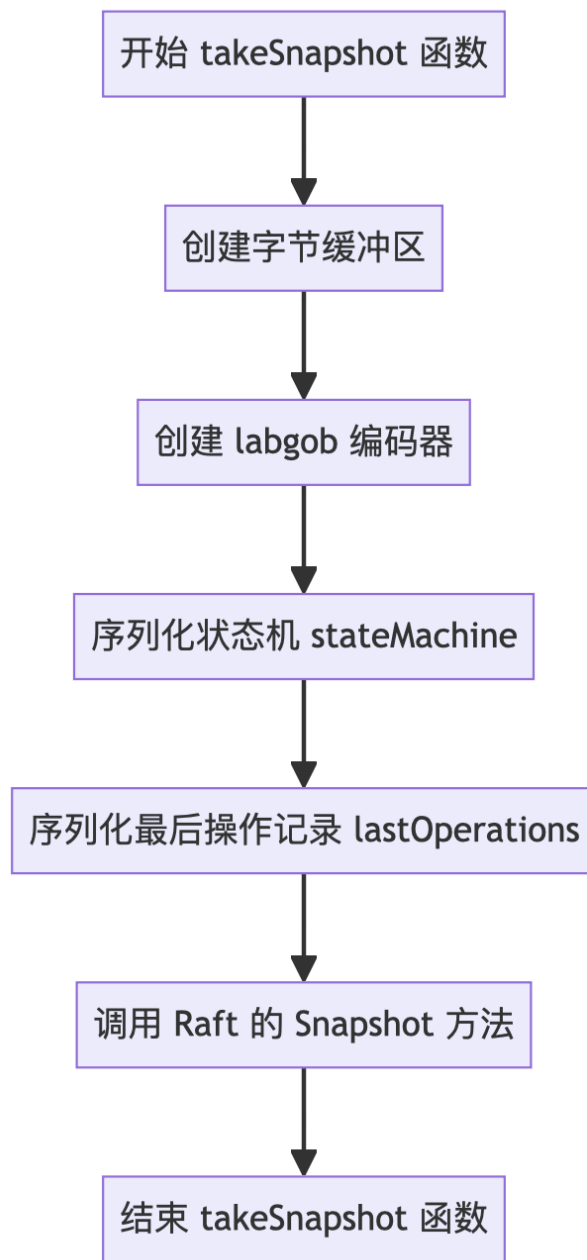


图 3-10 takeSnapshot 函数流程

```

// 创建一个新的 labgob 解码器，用于从缓冲区 r 中解码数据
d := labgob.NewDecoder(r)
var stateMachine MemoryKV
var lastOperations map[int64]OperationContext
// 从缓冲区反序列化出键值存储状态机（stateMachine）和最后操作的记录（lastOperations）
if d.Decode(&stateMachine) != nil ||
   d.Decode(&lastOperations) != nil {
    DPrintf("{Node_%v}_restores_snapshot_failed", kv.rf.Me())
}
    
```

```
// 更新 KVServer 的状态机和最后操作的记录为反序列化得到的数据
kv.stateMachine, kv.lastOperations = &stateMachine, lastOperations
}
```

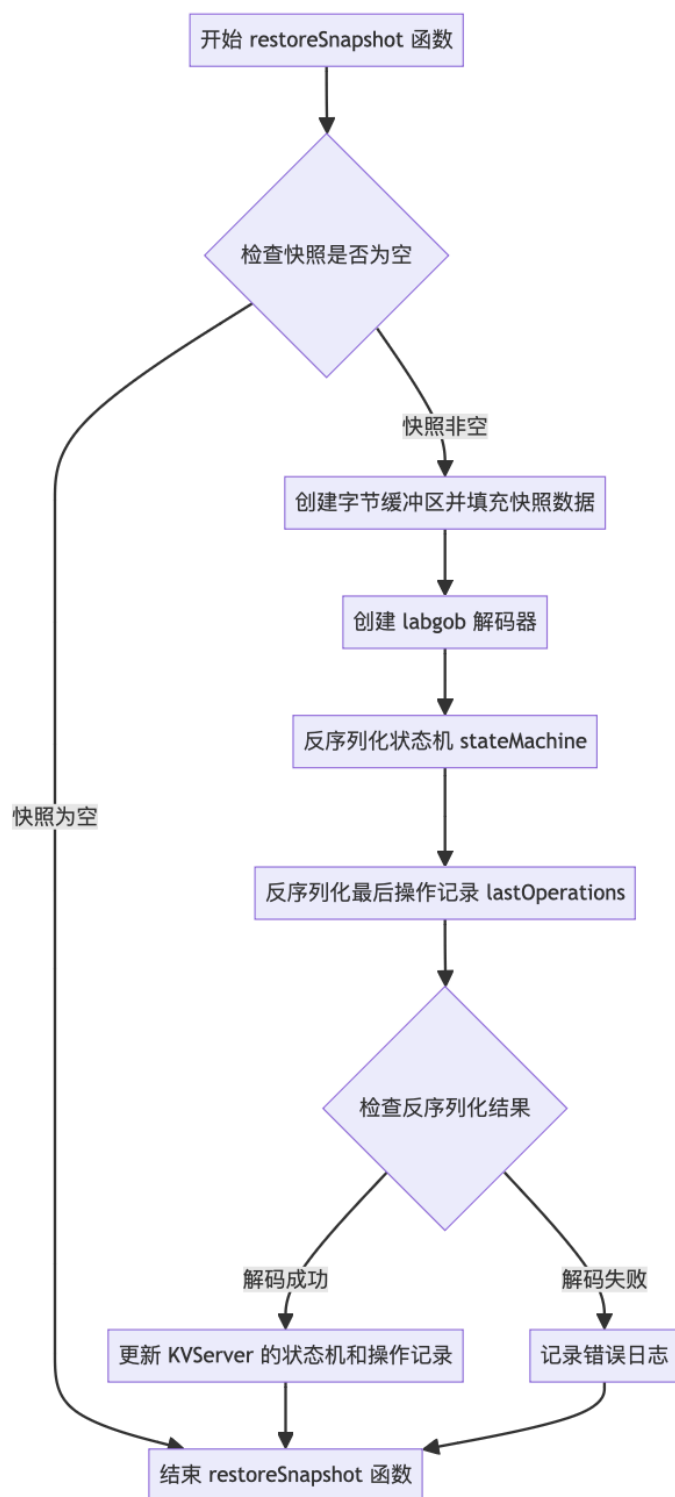


图 3-11 restoreSnapshot 处理流程

3.7 3B 测试

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$ go test -run 3B -race
Test: InstallSnapshot RPC (3B) ...
... Passed -- 4.4 3 3847 63
Test: snapshot size is reasonable (3B) ...
... Passed -- 2.4 3 6281 800
Test: ops complete fast enough (3B) ...
Test: restarts, snapshots, one client (3B) ...
... Passed -- 21.9 5 46775 6970
Test: restarts, snapshots, many clients (3B) ...
... Passed -- 22.2 5 44291 7575
Test: unreliable net, snapshots, many clients (3B) ...
... Passed -- 15.9 5 6845 1281
Test: unreliable net, restarts, snapshots, many clients (3B) ...
... Passed -- 23.2 5 8272 1260
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
... Passed -- 30.2 5 6377 587
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (3B) ...
... Passed -- 32.1 7 17542 1772
PASS
ok      6.824/kvraft      154.822s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab3/kvraft$
```

图 3-12 3B 测试

如图 3-12 所示，经过多次测试，3B 都可以通过。

4 实验过程中遇到的问题和解决办法

4.1 困难：Command 函数中 RPC 调用失败

问题：在 Clerk 的 Command 函数中，RPC 调用可能因网络问题或服务器故障失败。

解决办法：在函数中添加重试逻辑。当 RPC 调用失败时（如超时或连接错误），使用循环和延时重试机制，逐个尝试其他服务器，直到成功或达到最大重试次数。

4.2 困难：applyLogToStateMachine 中的命令重复

问题：由于网络延迟或重试，相同的命令可能被多次应用到状态机。

解决办法：在 KVServer 中实现命令去重逻辑。使用 lastOperations 记录每个客户端的最后一个操作上下文。在应用命令前，检查命令 ID 是否已经被应用。

4.3 困难：快照恢复后数据不一致

问题：在使用 ‘restoreSnapshot’ 恢复快照后，状态机数据与预期不一致。

解决办法：确保快照数据正确序列化和反序列化。检查 labgob 的使用是否正确，确保所有需要序列化的数据类型都已注册。反序列化后验证数据一致性。

4.4 困难：处理 ‘InstallSnapshot’ RPC 导致的死锁

问题：在处理 InstallSnapshot RPC 时，不当的锁操作可能导致死锁。

解决办法：审查 InstallSnapshot 中的锁操作。确保在所有可能的返回路径上正确释放锁，避免长时间持有锁。

4.5 困难：Snapshot 方法中日志裁剪错误

问题：在执行日志裁剪时，错误地删除了还未应用的日志条目。

解决办法：仔细检查裁剪逻辑，确保只裁剪已经包含在快照中的日志条目。使用日志的索引和任期作为裁剪的依据。

4.6 困难：applier 循环中的快照处理问题

问题：在 applier 循环中处理快照消息时，未能正确更新 lastApplied 和 commitIndex。

解决办法：在处理快照消息后，确保更新 lastApplied 和 commitIndex 到快照中包含的最后索引。

4.7 困难：在 Raft 重启时未能恢复快照

问题：Raft 重启后，未能从持久化存储中正确恢复快照。

解决办法：在 Raft 启动时，检查并加载持久化存储中的快照。如果存在快照，使用它来初始化 Raft 状态。

4.8 困难：KVServer 快照阈值判断错误

问题：KVServer 判断是否需要创建快照的逻辑错误，导致频繁或不必要的快照。

解决办法：仔细检查与 `maxRaftState` 相关的条件逻辑。确保只有当 Raft 状态大小接近 `maxRaftState` 时，才创建快照。

4.9 困难：takeSnapshot 中快照创建失败

问题：在 `takeSnapshot` 方法中，快照创建失败，无法正确存储到持久化存储。

解决办法：检查快照数据的序列化过程，确保正确调用 `labgob` 编码器。同时，验证 `raft.Snapshot` 调用是否正确处理快照数据。

4.10 困难：重启后 KVServer 状态与 Raft 不一致

问题：KVServer 重启后，其状态与 Raft 层的状态不一致。

解决办法：在 KVServer 启动时，确保从 Raft 层加载最新的快照和日志状态。使用 `applyCh` 来同步 KVServer 和 Raft 层的状态。

5 实验总结

完成 MIT 6.824 Lab 3 的过程不仅是一个技术挑战，更是对我分布式系统理解和实践能力的深化。这个实验分为两个部分：Lab 3A 和 Lab 3B，它们分别让我探索了键值存储的基本构建和快照功能的实现。

在 Lab 3A 中，我深入了解了 Raft 协议的工作机制和它在分布式键值存储系统中的应用。通过实现 ‘Clerk’ 结构体，我模拟了客户端与服务端的交互。这部分的挑战在于处理网络不稳定性和服务器故障情况。编写 RPC 调用和处理响应时，我学会了如何在分布式环境下保证稳定性和一致性。这不仅加深了我的网络编程知识，还锻炼了我面对不确定性时的问题解决能力。

进入 Lab 3B，我面临的主要任务是实现日志压缩和快照功能。这一部分让我更深刻地理解了分布式系统中状态管理的复杂性。实现快照功能时，我不得不考虑如何在不丢失关键信息的前提下，有效地减少状态的存储。这需要我对 Raft 日志结构和状态机的理解更加深入。编写代码时，我遇到了许多细节问题，如日志条目的有效裁剪、快照的序列化和反序列化。解决这些问题的过程不仅提升了我的编程技能，还加深了我对系统设计和性能优化的认识。

整体来说，Lab 3 不仅让我掌握了分布式系统的关键概念和技术，更重要的是，它教会了我如何在实际应用中应对挑战。面对复杂和不确定的问题，我学会了耐心地分析、理解并解决它们。这个实验不仅是对我技术能力的提升，更是对我的耐心、解决问题能力和系统思维的锻炼。它为我未来在更复杂的系统和项目中工作奠定了坚实的基础。