



中山大學
SUN YAT-SEN UNIVERSITY

题 目: Kraft Lab3A,3B

课程名称: 分布式计算

授课老师: 余晨韵

学生姓名: 张瑞程

学 号: 22354189

2024 年 12 月

目录

1. Pre-Checking.....	3
1.1 环境配置:	3
1.2 实验准备:	3
1.3 实验说明:	3
1.4 实验测试:	3
打开命令框, 输入:	3
1.5 实验结果:	3
2. Lab3A-Key/value service-基本 K-V 服务	4
2.1 任务要求:	4
2.2 任务分析:	5
2.3 功能实现:	5
2.3.1 RPC 相关 (common 部分)	5
2.3.2 Client-客户端部分	6
2.3.3 Server-服务端部分	10
2.4 Lab3A-测试结果	22
3. Lab3B-Key/value service with snapshots-实现快照的 K-V 服务	22
3.1 任务目标	22
3.2 任务要求	23
3.3 代码详解	23
3.3.1 Raft 结构体调整	23
3.3.2 四个基础 API-Lab2D 部分	25
3.3.3 4 个执行 API	29
3.4 Lab3B-测试结果	32
5. 实验总结	32
5.1 困难总结	32
5.2 实验心得	33
6. 相关参考资料	33

1. Pre-Checking

1.1 环境配置:

- 操作系统: Ubuntu 16.04 虚拟机
- 编辑器: Vscode 2023.

1.2 实验准备:

通过执行命令 `git clone git://g.csail.mit.edu/6.824-golabs-2022 6.824`, 按照提示下载实验所需文件。若在后续实验过程中发现程序无法正常运行, 需要更新电脑上的 `gcc` 版本, 可使用命令 `sudo apt-get install gcc-5 g++-5` 进行安装。

1.3 实验说明:

此次实验主要, 要求我们按照 MIT-6.824 的安排, 使用 Go 语言完成 Lab 3-AB。

- **Lab3A:** 基于 Raft 协议, 实现一个不包含快照功能的键值 (KV) 服务。
- **Lab3B:** 实现一个具备快照功能的键值 (KV) 服务, 快照的实现可参考 Lab2D 的相关内容。

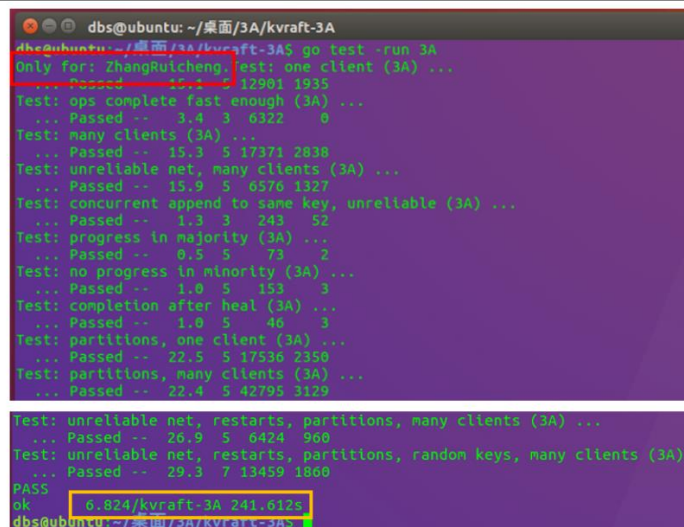
1.4 实验测试:

打开命令框, 输入:

- | | |
|----------------------------------|------------------|
| 1. <code>cd ~/6.824/src/x</code> | #进入特定 x 文件 |
| 2. <code>go test -run x</code> | #测试特定 x 模块 |
| 3. <code>go test</code> | #测试 x 文件中的所有实验模块 |

1.5 实验结果:

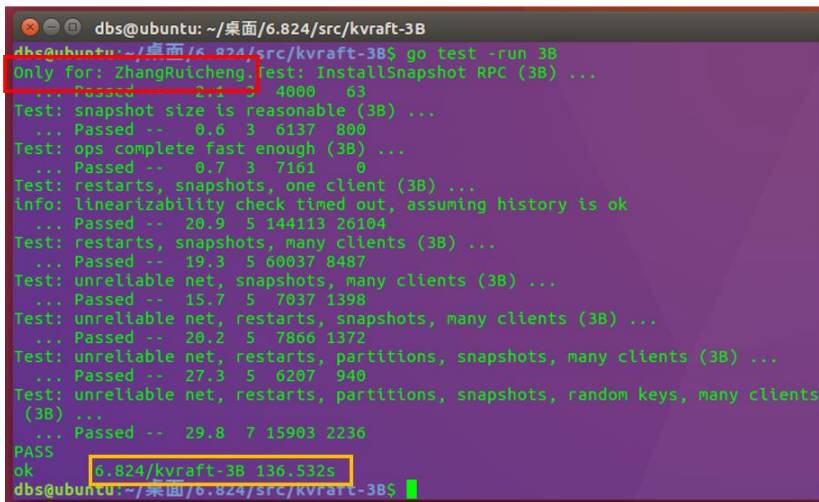
- | |
|---------------------------------------|
| 1. <code>cd ~/6.824/src/kvraft</code> |
| 2. <code>go test -run 3A</code> |



```
dbx@ubuntu: ~/桌面/3A/kvraft-3A
$ go test -run 3A
Only for: ZhangRuicheng test: one client (3A) ...
... Passed -- 12.1 12901 1935
Test: ops complete fast enough (3A) ...
... Passed -- 3.4 3 6322 0
Test: many clients (3A) ...
... Passed -- 15.3 5 17371 2038
Test: unreliable net, many clients (3A) ...
... Passed -- 15.9 5 6576 1327
Test: concurrent append to same key, unreliable (3A) ...
... Passed -- 1.3 3 243 52
Test: progress in majority (3A) ...
... Passed -- 0.5 5 73 2
Test: no progress in minority (3A) ...
... Passed -- 1.0 5 153 3
Test: completion after heal (3A) ...
... Passed -- 1.0 5 46 3
Test: partitions, one client (3A) ...
... Passed -- 22.5 5 17536 2350
Test: partitions, many clients (3A) ...
... Passed -- 22.4 5 42795 3129
Test: unreliable net, restarts, partitions, many clients (3A) ...
... Passed -- 26.9 5 6424 960
Test: unreliable net, restarts, partitions, random keys, many clients (3A) ...
... Passed -- 29.3 7 13459 1860
PASS
ok 6.824/kvraft-3A 241.612s
dbx@ubuntu: ~/桌面/3A/kvraft-3A
```

Lab3A 的测试结果为 241.612s

```
1. go test -run 3B
```



```
dbb@ubuntu: ~/桌面/6.824/src/kvraft-3B
dbb@ubuntu:~/桌面/6.824/src/kvraft-3B$ go test -run 3B
Only for: ZhangRuicheng.test: InstallSnapshot RPC (3B) ...
... Passed -- 2.1 3 4000 63
Test: snapshot size is reasonable (3B) ...
... Passed -- 0.6 3 6137 800
Test: ops complete fast enough (3B) ...
... Passed -- 0.7 3 7161 0
Test: restarts, snapshots, one client (3B) ...
Info: linearizability check timed out, assuming history is ok
... Passed -- 20.9 5 144113 26104
Test: restarts, snapshots, many clients (3B) ...
... Passed -- 19.3 5 60037 8487
Test: unreliable net, snapshots, many clients (3B) ...
... Passed -- 15.7 5 7037 1398
Test: unreliable net, restarts, snapshots, many clients (3B) ...
... Passed -- 20.2 5 7866 1372
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
... Passed -- 27.3 5 6207 940
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (3B) ...
... Passed -- 29.8 7 15903 2236
PASS
ok      6.824/kvraft-3B 136.532s
dbb@ubuntu:~/桌面/6.824/src/kvraft-3B$
```

Lab3B 的测试结果为 136.532s

2. Lab3A-Key/value service-基本 K-V 服务

2.1 任务要求:

本部分将实现 MIT 6.824 课程的 3A 部分功能，即构建一个基本的、不会丢失消息且不会失败的键值 (KV) 服务器，并实现一个无日志压缩的键值服务。具体要求如下：

1. 实现 Client 客户端

Client 是系统测试用例的接口，模拟客户端能够连接到 Leader 的 kvservers，并发送 Put、Append 和 get 三种 RPC 信号。这些信号对应于键值存储系统中的以下功能：

- 键值存放
- 键值对添加
- 键值对查询

在此部分，需要完成以下函数的补充：

- A. Clerk 结构体
- B. func MakeClerk() 初始化函数
- C. func (ck Clerk) Get 键值对查询函数
- D. func (ck Clerk) PutAppend() 键值对添加函数

2. 实现 kvserver 键值对服务器

kvserver 是一个具备 raft 协议功能的 key-value 存储服务器。它接收来自 client 的处理请求，并将这些请求在 raft 集群中通过一致性协议来处理和存储数据。此外，kvserver 的实现需要能够抵抗网络分区和确保数据的持久化。具体要求如下：

1. 网络分区和持久化能力:

- 只要大多数服务器在线且网络通畅，即使其他服务器失败或发生网络分区，系统也应能持续处理客户端的请求。
- 当服务器集群重启时，应能够在重启后恢复原有数据。

2. 具体功能实现：

- A. 在 `common.go` 中定义相关 RPC 结构体，包括操作信息、`PutAppendArgs` 和 `GetArgs` 等。
- B. 完成 `Kvserver` 的结构体定义。
- C. 基于 raft 一致性协议，实现 `PutAppend()` 的数据库查询与存储服务功能。
- D. 确保集群在网络异常和服务器崩溃中工作正常，即在 Clerk 多次发送同一个重复请求时，保证该请求只执行一次。
- E. 利用持久化技术实现系统的持久化服务。

2.2 任务分析：

基本思路如下：每个客户端都需要一个唯一的标识符，并且每个不同的命令都需要一个顺序递增的命令 ID。客户端 ID 和命令 ID 可以唯一确定一个命令，从而使得各个 Raft 节点能够记录并保存每个命令是否已被应用以及应用后的结果。如果默认客户端只能串行执行请求，那么服务端只需要维护一个映射（map），其键是客户端 ID，值是该客户端执行的最后一条日志的命令 ID 和状态机的输出即可。具体步骤如下：

- (1) 完成 `client` 和 `server` 基本构造；
- (2) 客户端方面主要完成 `Get`、`Put`、`PutAppend` 三个函数的处理。客户端可以向键/值服务发送三种不同的 RPC：`Put(key, value)`、`Append(key, arg)` 和 `Get(key)`。该服务维护一个简单的键/值对数据库。键和值是字符串。`Put(key, value)` 替换数据库中特定键的值，`Append(key, arg)` 将 `arg` 附加到键的值，`Get(key)` 获取键的当前值。一个不存在的键的 `Get` 应该返回一个空字符串。`Append` 到不存在的键应该像 `Put` 一样；
- (3) `server` 中实现 `PutAppend()` 和 `Get()` RPC 处理程序。这些处理程序应该使用 `Start()` 在 Raft 日志中输入一个 `Op`，在 `server` 完成 `Op` 结构定义，以便它描述 `Put/Append/Get` 操作；
- (4) `server` 要完成 `applyCh` 的处理，对于每一个要引用的命令，要进行处理。每个服务器都应该在 Raft 提交它们时执行 `Op` 命令，即当它们出现在 `applyCh` 上时。当 Raft 提交其 `Op` 时，RPC 处理程序应该注意到，然后回复 RPC。

2.3 功能实现：

2.3.1 RPC 相关（common 部分）

实验中，要求 `clerk` 与 `kvserver` 间通过 RPC 进行通信，因此，首先需要完善相关 RPC 结构体。`rpc` 中服务器返回给客户端的五种状态码，不同的状态码会进行不同的处理：

状态变量

参数	解释
----	----

OK	命令执行成功，可以退出了
ErrNoKey	如没有这个 key，直接退出
ErrWrong Leader	请求的节点并不是 leader，因此按照某种规则换一个节点再进行请求（我这里使用的策略是轮询请求）
ErrTimeOut	请求超时，继续请求
ErrServer	server 出现了一些问题 crash 了，可以换一个节点继续请求。

为了实现集群中请求的非重复性处理，我们在 PutAppendArgs 和 GetArgs 结构体中添加了 MsgId 和 ClientId 两项信息。这两项信息能够唯一标识一条存储请求，对于 kvserver 在处理请求时判断该指令是否为重复指令至关重要。

```

1. type PutAppendArgs struct {
2.     Key    string // 键
3.     Value  string // 值
4.     Op     string // 操作类型, "Put" 或 "Append"
5.     MsgId  int64  // 用于唯一标识一个请求的 Args
6.     ClientId int64 // 客户端唯一标识符
7. }
8.
9. type GetArgs struct {
10.    Key    string // 键
11.    MsgId  int64  // 同样添加标识符，用于唯一标识请求
12.    ClientId int64 // 客户端唯一标识符
13. }
14.
15. type PutAppendReply struct {
16.     Err error // 错误信息
17. }
  
```

2.3.2 Client-客户端部分

Client 作为系统测试用例的接口，负责模拟客户端与能够连接到 Leader 的 kvservers 进行交互。它能够发送三种主要的 RPC 请求：Put、Append 和 Get，分别对应键值存储系统中的以下功能：

- 键值存放：通过 Put 请求将键值对存储到系统中。
- 键值对添加：通过 Append 请求将额外的值附加到已存在的键的值上。
- 键值对查询：通过 Get 请求查询特定键的当前值。

(1) Clerk 结构体定义及其初始化

定义：Clerk 结构体存储了服务器列表 servers，用以在每次发出请求时搜索到 leader 并发送请求到对方服务器上。为避免每次发送时都需要重复搜索 Leader，此处定义 LeaderId 标记 leader 的 ID 值。此外，为了避免由于网络故障等导致的重复请求，此处还

定义 `clientId` 标识自身 ID，结合服务器端存储的自己上一次发送的 `MsgId` 信息，可以判断本次请求是否为本机的第二次发送。

```
1. type Clerk struct {
2.     servers []*labrpc.ClientEnd // 存储服务器列表，用于在每次发出请求时搜索到 leader
3.     LeaderId int                // 记录搜索到的 leader 的 ID，用于访问
4.     clientId int64              // 标识自身，用来给 server 检查我发的最后一条请求，避免重复
5. }
```

需要注意的是，`leaderId` 在这里并不一定是 leader，仅仅是一个请求标识。

初始化：在初始化部分，使用了 `rand()` 函数，该函数用于生成一个大范围的随机数。由于 `rand.Reader` 是一个密码学安全的随机数生成器，它能够确保生成的随机数在概率上是唯一的，从而保证每个 `Clerk` 拥有一个唯一的 ID 值。以下是具体的代码实现：

```
1. func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
2.     ck := new(Clerk)
3.     ck.servers = servers
4.     ck.LeaderId = 0
5.     ck.clientId = rand() // 生成唯一的客户端 ID
6.     return ck
7. }
```

(2) Clerk 三种动作

`Get`、`Put` 和 `Append` 三个函数负责处理查询请求或键值对存储请求。这些函数的处理逻辑如下：

- A. 生成 RPC 信号 首先，生成相应的 RPC 信号以准备发送请求。
- B. 向当前认为的 Leader 发送请求 请求的结果可能存在以下几种情况：
 1. 发送成功且对方是 Leader：在这种情况下，请求已经成功处理，程序将退出请求循环。
 2. 发送成功但对方不是 Leader：此时，使用轮询策略，将请求发送给下一个服务器。这是因为当前的服务器不是 Leader，无法处理该请求。
 3. 发送失败：判断为网络延迟或对方出现分区。在这种情况下，同样使用轮询策略，将请求发送给下一个服务器，以确保请求能够被处理。

通过这种策略，只有在请求成功被服务器处理的情况下，程序才会退出。否则，程序将持续进行查询，直到找到可以处理请求的服务器。

• 发送一个 get 请求

```
1. // Get 方法用于从服务器获取键对应的值
2. func (ck *Clerk) Get(key string) string {
3.     // 创建 GetArgs 参数结构体
4.     args := GetArgs{
5.         Key:    key,
6.         ClientId: ck.clientId,
7.         CommandId: rand(), // 生成唯一的命令 ID
```



```
8.     }
9.     // 初始 LeaderId 设置为 ck.leaderId
10.    leaderId := ck.leaderId
11.    for {
12.        // 调用服务器的 Get 方法
13.        reply := GetReply{}
14.        ok := ck.servers[leaderId].Call("KVServer.Get", &args, &reply)
15.        if ok {
16.            // 如果请求失败，等待一段时间后重新请求，换一个节点再请求
17.            DPrintf("%v client get key %v from server %v, not ok.", ck.clientId, key, leaderId)
18.            time.Sleep(ChangeLeaderInterval)
19.            leaderId = (leaderId + 1) % len(ck.servers)
20.            continue
21.        } else if reply.Err != OK {
22.            // 如果请求返回错误
23.            DPrintf("%v client get key %v from server %v, reply err = %v!", ck.clientId, key,
leaderId, reply.Err)
24.        }
25.        // 根据返回的错误类型进行处理
26.        switch reply.Err {
27.        case OK:
28.            // 如果请求成功
29.            DPrintf("%v client get key %v from server %v, value: %v, OK.", ck.clientId, key,
leaderId, reply.Value)
30.            ck.leaderId = leaderId
31.            return reply.Value
32.        case ErrNoKey:
33.            // 如果键不存在
34.            DPrintf("%v client get key %v from server %v, NO KEY!", ck.clientId, key, leaderId)
35.            ck.leaderId = leaderId
36.            return ""
37.        case ErrTimeOut:
38.            // 如果请求超时，继续尝试下一个服务器
39.            continue
40.        default:
41.            // 如果发生其他错误，等待一段时间后重新请求，换一个节点再请求
42.            time.Sleep(ChangeLeaderInterval)
43.            leaderId = (leaderId + 1) % len(ck.servers)
44.            continue
45.        }
46.    }
47. }
```

主要就是发送 Get RPC，获取指定 key 的 value，以及状态码，根据不同的状态码进行不同的处理，前面对状态码也有介绍。



• 发送 Put 和 Append 请求:

```

1. / 1. / Put 方法用于将键值对存储到服务器
2. func (ck *Clerk) Put(key string, value string) {
3.     ck.PutAppend(key, value, "Put")
4. }
5. // Append 方法用于将值附加到已存在的键的值上
6. func (ck *Clerk) Append(key string, value string) {
7.     ck.PutAppend(key, value, "Append")
8. }

```

Put 和 Append 两个函数分别用于处理键值对存储请求和附加请求。这两种命令都是基于 PutAppend() 函数实现, 和 Get 的处理大体相同, 发送 PutAppend RPC, 获取指定 key 的 value, 以及状态码, 根据不同的状态码进行不同的处理。

```

1. // PutAppend 方法用于处理键值对的存储 (Put) 或附加 (Append) 操作
2. func (ck *Clerk) PutAppend(key string, value string, op string) {
3.     // 打印调试信息, 显示客户端 ID、键、值和操作类型
4.     DPrintf("%v client PutAppend, key: %v, value: %v, op: %v", ck.clientId, key, value, op)
5.     // 创建 PutAppendArgs 参数结构体
6.     args := PutAppendArgs{
7.         Key:    key,        // 键
8.         Value:   value,      // 值
9.         Op:     op,          // 操作类型, "Put" 或 "Append"
10.        ClientId: ck.clientId, // 客户端 ID
11.        CommandId: nrand(),    // 生成唯一的命令 ID
12.    }
13.    // 初始化 LeaderId 为当前记录的 LeaderId
14.    leaderId := ck.leaderId
15.    // 循环尝试发送请求直到成功或所有服务器都尝试完毕
16.    for {
17.        // 创建用于接收回复的结构体
18.        reply := PutAppendReply{}
19.        // 向当前 Leader 发送请求
20.        ok := ck.servers[leaderId].Call("KVServer.PutAppend", &args, &reply)
21.        // 如果请求发送失败
22.        if !ok {
23.            // 打印错误信息并等待一段时间后尝试下一个服务器
24.            DPrintf("%v client set key %v to %v to server %v, not ok.", ck.clientId, key,
value, leaderId)
25.            time.Sleep(ChangeLeaderInterval) // 等待一段时间后重试
26.            leaderId = (leaderId + 1) % len(ck.servers) // 尝试下一个服务器
27.            continue // 继续循环
28.        } else if reply.Err != OK {
29.            // 如果请求发送成功但返回错误
30.            DPrintf("%v client set key %v to %v to server %v, reply err = %v!", ck.clientId,

```

```
key, value, leaderId, reply.Err)
31.     }
32.     switch reply.Err {    // 根据返回的错误类型进行处理
33.     case OK:
34.         // 如果请求成功
35.         DPrintf("%v client set key %v to %v to server %v, OK.", ck.clientId, key, value,
leaderId)
36.         ck.leaderId = leaderId // 更新 LeaderId 为当前成功的服务器
37.         return // 成功后退出函数
38.     case ErrNoKey:
39.         // 如果键不存在
40.         DPrintf("%v client set key %v to %v to server %v, NO KEY!", ck.clientId, key,
value, leaderId)
41.         return // 键不存在, 退出函数
42.     case ErrTimeOut:
43.         // 如果请求超时, 继续尝试下一个服务器
44.         continue
45.     case ErrWrongLeader:
46.         // 如果当前服务器不是 Leader, 等待一段时间后尝试下一个服务器
47.         time.Sleep(ChangeLeaderInterval)
48.         leaderId = (leaderId + 1) % len(ck.servers)
49.         continue
50.     case ErrServer:
51.         // 如果服务器错误, 等待一段时间后尝试下一个服务器
52.         time.Sleep(ChangeLeaderInterval)
53.         leaderId = (leaderId + 1) % len(ck.servers)
54.         continue
55.     default:
56.         // 如果发生未知错误, 记录致命错误并退出程序
57.         log.Fatal("client rev unknown err", reply.Err)
58.     }
59. }
60. }
```

PutAppend 方法是 Clerk 结构体中的一个核心功能, 用于处理键值对的存储 (Put) 或附加 (Append) 操作。该方法通过向当前认为的 Leader 发送请求, 并根据返回的结果进行相应的处理。如果请求成功且对方是 Leader, 则操作完成并退出; 如果对方不是 Leader 或请求失败, 则使用轮询策略尝试向其他服务器发送请求。此外, 该方法还处理了多种错误情况, 包括键不存在、超时、错误的 Leader 和服务器错误, 确保了操作的可靠性和一致性。

2.3.3 Server-服务端部分

kvserver 是具有 raft 协议功能的 key-value 存储服务器, 其接收来自 client 的处理请求, 并将请求在 raft 集群中通过一致状态机存储数据。另外, kvserver 的实现要求可抵抗

网络分区与可持久化：只要大多数服务器在线并且网络通畅，就算其他服务器失败或者网络分区，都应该可以持续处理客户端的请求；另外，当服务器集群重启，也应当能够在重启后恢复原有数据。

服务端的代码主要分为：1.数据结构定义；2.初始化代码；3.RPC 接收处理代码；4.Applych 处理代码，即命令应用代码。

（1）数据结构

```
const WaitCmdTimeOut = time.Millisecond * 500 // cmd执行
const MaxLockTime = time.Millisecond * 10 // debug

type Op struct {
    // Your definitions here.
    // Field names must start with capital letters,
    // otherwise RPC will break.
    ReqId    int64 //用来标识commandNotify
    CommandId int64
    ClientId  int64
    Key       string
    Value     string
    Method    string
}

type CommandResult struct {
    Err    Err
    Value  string
}

type KVServer struct {
    mu      sync.Mutex
    me      int
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg
    dead    int32 // set by Kill()
    stopCh  chan struct{}

    maxraftstate int // snapshot if log grows this big

    // Your definitions here.
    commandNotifyCh map[int64]chan CommandResult
    lastApplies     map[int64]int64 //k-v: ClientId-CommandId
    data            map[string]string

    //持久化
    persister *raft.Persister

    //用于互斥锁
    lockStartTime time.Time
    lockEndTime   time.Time
    lockMsg       string
}
```

数据结构变量大致作用：

参数	解释
Op	对接收到的命令进行的一个解析封装，发送给 raft，便于对 Applych 的处理
CommandResult	对每一个命令应用结果的封装
maxraftstate	当 raft 的日志数量达到这个数量，就进行一次 snapshot
commandNotifyCh	用于命令应用后对请求协程的唤醒
lastApplies	实现线性化语义，k-v = clientid-commandid
persister	用于保存初始化 server 的 persister，其实它的用处只有一点：获取 raft 的日志长度用于 snapshot 判断。因为 raft 的属性都是私有的，没法访问，为了保证不在 raft 中进行修改，因此保存一个 persister 用于调用。

（2）初始化代码

StartKVServer 函数初始化并启动一个键值存储服务器（KVServer），该服务器集成了 Raft 协议以确保数据的一致性和高可用性。函数首先注册了需要进行序列化和反序列化的结构体，然后创建了一个新的 KVServer 实例，并设置了服务器的索引（me）、最大 Raft 状态（maxraftstate）和持久化存储（persister）。接着，函数初始化了一些必要的数据结构，包括命令通知通道、应用通道和数据存储。此外，它还从持久化存储中读取了快照数据，并启动了一个 goroutine 来处理应用通道。最后，函数创建了一个 Raft 实例并返回了 KVServer 实例。

1. // StartKVServer 初始化并启动一个 KVServer 实例
2. func StartKVServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister,

```
maxraftstate int) *KVServer {
3.    // 注册 Op 结构体以便于 Go 的 RPC 库进行编组和解组
4.    labgob.Register(Op{})
5.    // 创建一个新的 KVServer 实例
6.    kv := new(KVServer)
7.    kv.me = me
8.    kv.maxraftstate = maxraftstate
9.    kv.persister = persister
10.   // 初始化 KVServer 的一些成员变量
11.   kv.lastApplies = make(map[int64]int64)
12.   kv.data = make(map[string]string)
13.   // 创建一个停止通道
14.   kv.stopCh = make(chan struct{})
15.   // 从持久化存储中读取快照
16.   kv.readPersist(true, 0, 0, kv.persister.ReadSnapshot())
17.   // 创建命令通知通道和应用消息通道
18.   kv.commandNotifyCh = make(map[int64]chan CommandResult)
19.   kv.applyCh = make(chan raft.ApplyMsg)
20.   // 创建一个 Raft 实例
21.   kv.rf = raft.Make(servers, me, persister, kv.applyCh)
22.   // 启动一个 goroutine 来处理应用通道
23.   go kv.handleApplyCh()
24.   // 返回 KVServer 实例
25.   return kv
26. }
```

(3) RPC 接收处理代码 (Get&put Append 函数)

Get 和 PutAppend 函数作为 RPC 调用的接口函数，负责接收来自 Clerk 的任务请求。其主要功能是将 Clerk 发送的服务请求封装到 **Op 结构体**中，然后通过调用 `kv.rf.StartOp(op)` 将任务指令提交到 Raft 集群中进行共识性处理。首先我们先展示一下我们定义的 Op 结构体：

① 结构体 Op 设计

Op 结构体用于封装发送给服务端以执行数据库请求的操作。

```
1. type Op struct {
2.     Key    string // 键
3.     Value  string // 值
4.     Operation string // 操作类型, "put" 或 "get"
5.     MsgId  int64 // 当前封装的来自 Clerk 的指令消息号
6.     ReqId  int64 // 标识当前 Op 的消息号
7.     ClientId int64 // 客户端唯一标识符
8. }
```

对其中部分成员变量解释如下：

- **Operation string**: 记录当前指令类型，可选值包括 "put"、"append" 或 "get"。

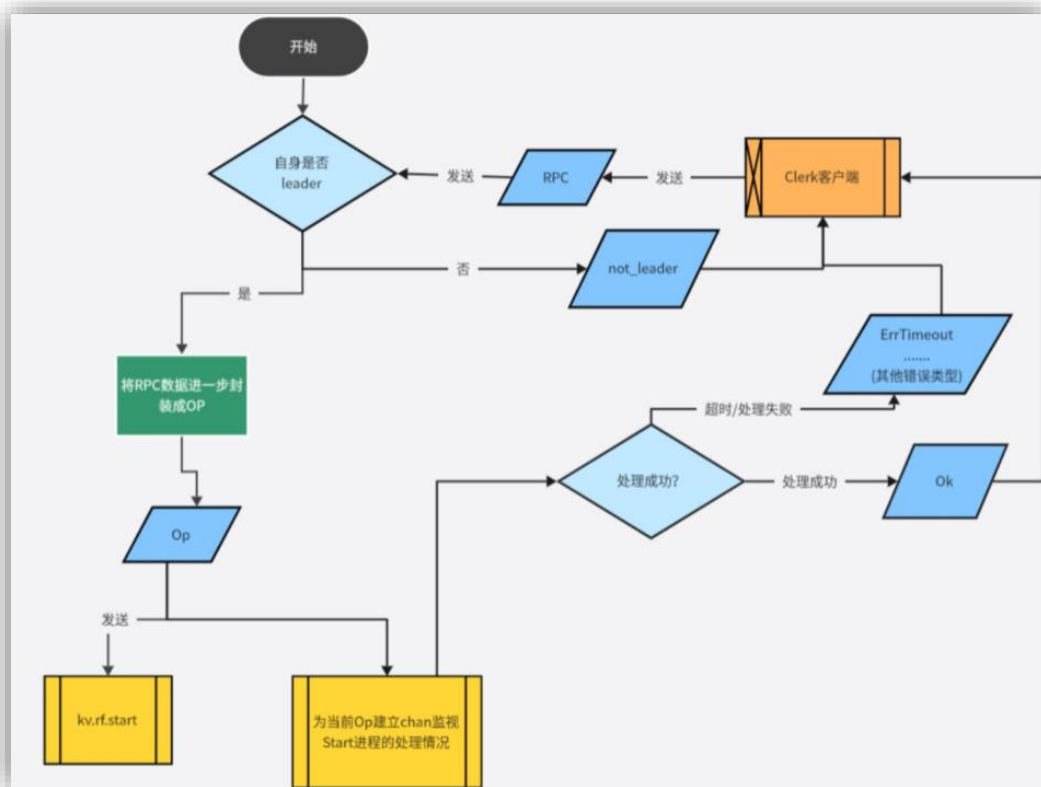
- MsgId int64: 存储当前 Op 封装的来自 Clerk 的请求消息号。
- ReqId int64: 唯一标识每一条 Op, 需要另外为当前 Op 生成请求号, 用于查询当前消息处理进度。
- persister *raft.Persister: 实现数据持久化。
- msgNotify map[int64]chan NotifyMsg: 消息管道, 在 Get 和 PutAppend 中为每条待处理消息建立管道以追踪处理结果。

在了解过 Op 结构体后, 下面我们将正式介绍 Get&put Append 函数。

② Get 与 PutAppend 函数设计

两个函数的处理逻辑大致相同, 它们接收来自 Clerk 的调用并返回相应的结果。具体流程如下:

- ✚ 当接收到来自 Clerk 的消息时, 首先检查当前服务器是否为 Leader (没有这个判断会出问题!)。如果不是 Leader, 则返回 ErrWrongLeader 错误。
- ✚ 如果是 Leader, 将请求的 RPC 封装成一个 Op 结构体, 并将其传递给 kv.rf.Start(op) 以进行共识性日志处理。
- ✚ 同时, 为当前的 Op 建立一个从 ReqId 到 chan NotifyMsg 的映射关系。在另一个并发进程中, 会根据 Start 进程的处理结果, 将处理结果发送到对应的通道中。在当前进程中, 通过 ReqId 监控通道的数据, 从而判断当前操作的处理结果。
- ✚ 如果处理成功, 则返回 OK 信号; 如果处理失败, 则返回 NotifyMsg 中包含的错误信息。



Get 函数

```
1. func (kv *KVServer) Get(args *GetArgs, reply *GetReply) {
2.     // 收到 get 请求, 则调用 start 交给 raft 集群达成共识。start 包含检查自身是否 leader 的代码
3.     op := Op{
4.         MsgId:  args.MsgId,
5.         ReqId:  nrand(),
6.         Key:    args.Key,
7.         Operation: "GET",
8.         ClientId: args.ClientId,
9.     }
10.    var res NotifyMsg
11.    _, isLeader := kv.rf.GetState()
12.    if !isLeader { // 不是 leader 直接返回
13.        res.Err = ErrWrongLeader
14.    } else {
15.        kv.rf.Start(op) // 交给 start 进行一致性共识
16.        ch := make(chan NotifyMsg, 1)
17.        kv.mu.Lock()
18.        kv.msgNotify[op.ReqId] = ch // 将管道存放到 map 中, 当 start 处理结束后, 其他进程会存放结果
到该管道中
19.        kv.mu.Unlock()
20.        t := time.NewTimer(ChangeLeaderInterval) // 定时器, 避免超时
21.        defer t.Stop()
22.        select {
23.        case res = <-ch: // 处理结束后, 另一个进程会将结果存放到 ch 中。
24.            kv.mu.Lock()
25.            delete(kv.msgNotify, op.MsgId)
26.            kv.mu.Unlock()
27.        case <-t.C: // 超时, 删除管道。
28.            kv.mu.Lock()
29.            delete(kv.msgNotify, op.MsgId)
30.            kv.mu.Unlock()
31.            res.Err = ErrTimeOut
32.        }
33.    }
34.    reply.Err = res.Err
35.    reply.Value = res.Value
36. }
```

上面的代码段展示了 KVServer 结构体的 Get 方法, 用于处理来自客户端的获取 (Get) 请求。方法首先检查当前服务器是否为 Leader, 如果不是, 则直接返回错误。如果是 Leader, 则将请求封装成 Op 结构体, 并提交给 Raft 集群进行共识处理。然后, 方法等待处理结果, 并将结果返回给客户端。

PutAppend 函数

```
1. func (kv *KVServer) PutAppend(args *PutAppendArgs, reply *PutAppendReply) {
```

```
2.   op := Op{
3.       MsgId:  args.MsgId,
4.       ReqId:  nrand(),
5.       Key:    args.Key,
6.       Value:  args.Value,
7.       Operation: args.Op,
8.       ClientId: args.ClientId,
9.   }
10.  var res NotifyMsg
11.  _, isLeader := kv.rf.GetState()
12.  if !isLeader { // 不是 leader 直接返回
13.      res.Err = ErrWrongLeader
14.  } else {
15.      kv.rf.Start(op)
16.      ch := make(chan NotifyMsg, 1)
17.      kv.mu.Lock()
18.      kv.msgNotify[op.ReqId] = ch // 管道等待一致性结果，如果有调用成功则将结果放在 ch 里
19.      kv.mu.Unlock()
20.      t := time.NewTimer(WaitTimeOut)
21.      defer t.Stop()
22.      select {
23.      case res = <-ch:
24.          kv.mu.Lock()
25.          delete(kv.msgNotify, op.ReqId)
26.          kv.mu.Unlock()
27.      case <-t.C: // 超时，同样
28.          kv.mu.Lock()
29.          delete(kv.msgNotify, op.ReqId)
30.          kv.mu.Unlock()
31.          res.Err = ErrTimeOut
32.      }
33.  }
34.  reply.Err = res.Err
35. }
```

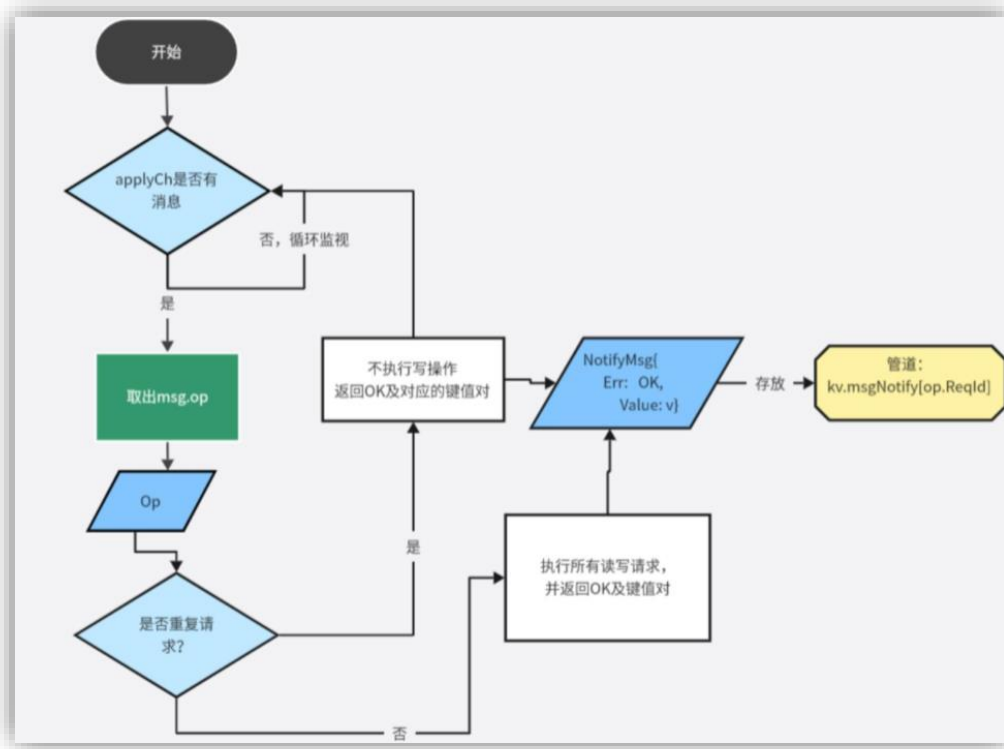
上面的代码段展示了 KVServer 结构体的 PutAppend 方法，用于处理来自客户端的存储（Put）或附加（Append）请求。方法首先检查当前服务器是否为 Leader，如果不是，则直接返回错误。如果是 Leader，则将请求封装成 Op 结构体，并提交给 Raft 集群进行共识处理。然后，方法等待处理结果，并将结果返回给客户端。

（4）WaitApplyCh 日志应用监视函数

RPC 还要一个 waitCmd 部分-负责调用 start 向 raft 请求命令。该程序主要用于监视 raft 集群对请求的共识结果并执行共识后的操作。当 raft 成功处理某条日志，会将日志应用结果与



相关信息存放到 kv.applyCh 中。WaitApplyCh 进程不停取出 apply 通道的信息，并执行日志共识后的数据库读写操作。大致流程如下图所示。



```

1. func (kv *KVServer) waitCmd(op Op) (res CommandResult) {
2.     // 打印调试信息, 显示等待命令的开始
3.     DPrintf("server %v wait cmd start, Op: %+v.\n", kv.me, op)
4.
5.     // 提交命令到 Raft 状态机, 无论是否是 Leader 节点都可以发起命令
6.     index, term, isLeader := kv.rf.Start(op)
7.     if !isLeader {
8.         // 如果当前节点不是 Leader, 返回错误
9.         res.Err = ErrWrongLeader
10.        return
11.    }
12.    kv.lock("waitCmd")
13.    ch := make(chan CommandResult, 1) // 创建一个通道用于接收命令结果
14.    kv.commandNotifyCh[op.ReqId] = ch // 将通道与请求 ID 关联, 以便后续可以访问结果
15.    kv.unlock("waitCmd")
16.    DPrintf("start cmd: index:%d, term:%d, op:%+v", index, term, op)
17.    t := time.NewTimer(WaitCmdTimeOut) // 创建一个定时器, 用于设置超时时间
18.    defer t.Stop() // 确保函数退出时停止定时器
19.
20.    // 使用 select 语句等待命令结果或超时
21.    select {

```

```
22.     case <-kv.stopCh:
23.         // 如果服务器停止, 打印信息, 删除通道, 并返回服务器错误
24.         DPrintf("stop ch waitCmd")
25.         kv.removeCh(op.ReqId)
26.         res.Err = ErrServer
27.         return
28.     case res = <-ch:
29.         // 如果从通道接收到结果, 删除通道并返回结果
30.         kv.removeCh(op.ReqId)
31.         return
32.     case <-t.C:
33.         // 如果超时, 删除通道, 设置超时错误, 并返回
34.         kv.removeCh(op.ReqId)
35.         res.Err = ErrTimeOut
36.         return
37. }
38. }
```

需要注意的是, 在 `select` 语句中, 代码处理了两种情况: 从通道接收结果和定时器超时。在这两种情况下, 都需要从 `kv.msgNotify` 中删除对应的通道, 以避免内存泄漏。这是一个重要的细节, 因为如果不及时清理, 可能会导致内存中积累大量的无用通道, 最终耗尽内存资源。

另外, 重复请求不执行写操作, 但仍需正常返回处理结果到 `clerk` 中, 否则 `clerk` 会判断为处理失败从而无限循环地发送请求。

(5) 命令执行代码

接下来就是整个 `server` 中最重要的函数了——`KVServer` 结构体的 `handleApplyCh` 函数, 该函数是一个事件循环, 用于处理来自 `applyCh` 通道的命令。

```
1. func (kv *KVServer) handleApplyCh() {
2.     for {
3.         select {
4.             case <-kv.stopCh:
5.                 DPrintf("get from stopCh, server-%v stop!", kv.me)
6.                 return
7.             case cmd := <-kv.applyCh:
8.                 // 处理快照命令, 读取快照的内容
9.                 if cmd.SnapshotValid {
10.                    DPrintf("%v get install sn,%v %v", kv.me, cmd.SnapshotIndex, cmd.SnapshotTerm)
11.                    kv.lock("waitApplyCh_sn")
12.                    kv.readPersist(false, cmd.SnapshotTerm, cmd.SnapshotIndex, cmd.Snapshot)
13.                    kv.unlock("waitApplyCh_sn")
14.                    continue
15.                }
```



```
16.      // 处理普通命令
17.      if !cmd.CommandValid {
18.          continue
19.      }
20.      cmdIdx := cmd.CommandIndex
21.      DPrintf("server %v start apply command %v: %v", kv.me, cmdIdx, cmd.Command)
22.      op := cmd.Command.(Op)
23.      kv.lock("handleApplyCh")
24.
25.      if op.Method == "Get" {
26.          // 处理读
27.          e, v := kv.getValueByKey(op.Key)
28.          kv.notifyWaitCommand(op.ReqId, e, v)
29.      } else if op.Method == "Put" || op.Method == "Append" {
30.          // 处理写
31.          // 判断命令是否重复
32.          isRepeated := false
33.          if v, ok := kv.lastApplies[op.ClientId]; ok {
34.              if v == op.CommandId {
35.                  isRepeated = true
36.              }
37.          }
38.          if !isRepeated {
39.              switch op.Method {
40.              case "Put":
41.                  kv.data[op.Key] = op.Value
42.                  kv.lastApplies[op.ClientId] = op.CommandId
43.              case "Append":
44.                  e, v := kv.getValueByKey(op.Key)
45.                  if e == ErrNoKey {
46.                      // 按 put 处理
47.                      kv.data[op.Key] = op.Value
48.                      kv.lastApplies[op.ClientId] = op.CommandId
49.                  } else {
50.                      // 追加
51.                      kv.data[op.Key] = v + op.Value
52.                      kv.lastApplies[op.ClientId] = op.CommandId
53.                  }
54.              }
55.          }
56.          default:
57.              kv.unlock("handleApplyCh")
58.              panic("unknown method " + op.Method)
59.      }
```

```
60.         // 命令处理成功
61.         kv.notifyWaitCommand(op.ReqId, OK, "")
62.         kv.unlock("handleApplyCh")
63.         DPrintf("apply op: cmdIdx:%d, op:%+v, data:%v", cmdIdx, op, kv.data[op.Key])
64.         // 每应用一条命令，就判断是否进行持久化
65.         kv.saveSnapshot(cmdIdx)
66.     }
67. }
68. }
```

处理逻辑在一个 for 循环中进行，从 applyCh 通道中获取两种类型的命令：快照命令和普通命令。具体的结构可以参考 Raft 的实现。关于持久化部分的细节可以暂时忽略，只需了解在哪个位置需要进行相应的处理即可。以下是普通命令执行步骤的简要介绍：

- ✚ 从 applyCh 通道中获取数据后，首先进行转换，将其转换成我们的 Op 结构体。
- ✚ 如果是 Get 操作，根据 key 直接获取 value，并唤醒等待该操作完成的协程。
- ✚ 如果是 Put 或 Append 操作，首先需要判断该命令是否重复。如果不重复，表示可以执行，因此根据 Put 和 Append 的不同分别进行处理。需要注意的是，在 Append 命令中，如果 key 不存在，则实际上会执行 Put 操作。最后，唤醒等待该操作完成的协程。

（6）可持久化实现

在 3A 部分中，我们还需要实现 kvserver 的数据持久化功能。这意味着即使在 server 集群出现宕机的情况下，我们仍然能够恢复数据库中的数据。由于 Raft 已经实现了日志部分的持久化，因此我们可以使用 Raft 中的 persister 来实现这一功能。然而，由于 kvserver 与 Raft 的数据持久化时机并不相同（即在各自的数据出现修改时进行持久化存储），我们不能直接共用 Raft 中的 persister 持久化函数，否则会造成数据相互覆盖，从而引发冲突。因此，我们需要特别处理这一部分。

I. 持久化数据

在 kvserver 中，data 表示数据库部分，必须进行保存。此外，由于 lastApplies 记录了每个 clerk 的上次处理的请求信息，这对于避免重复请求的处理至关重要，因此也需要保存。

- data: 一个映射，用于存储键值对数据。
- lastApplies: 一个映射，记录每个节点上次提交的日志，用于查重以避免重复执行。

II. 数据存储

数据存储部分通过两个步骤实现。首先，在 raft.persister.go 文件中添加对应的 kvserver 数据对象及数据存储函数：

```
1. type Persister struct {
2.     // 3A 添加，用于存储 kvserver 数据
3.     kv_data []byte
4. }
5. // SaveKvdata 将 kvserver 数据持久化到磁盘
```

```
6. func (ps *Persister) SaveKvdata(data []byte) {
7.     ps.mu.Lock()
8.     defer ps.mu.Unlock()
9.     ps.kv_data = clone(data) // 将数据存储到 kv_data 中，实现持久化
10. }
```

接下来，在 `server.go` 文件中编写数据持久化函数，打包数据并调用 `SaveKvdata()` 进行存储：

```
1. // persist_kvdata 将 kvserver 数据持久化到磁盘
2. func (kv *KVServer) persist_kvdata() {
3.     w := new(bytes.Buffer)
4.     e := labgob.NewEncoder(w)
5.     // 存储所有内容
6.     e.Encode(kv.data)
7.     e.Encode(kv.lastApplies)
8.     data := w.Bytes()
9.     kv.persister.SaveKvdata(data)
10. }
```

III. 数据读取

为了实现 `kvserver` 的数据持久化功能，我们需要在 `raft.persister.go` 文件中定义 `kvserver` 数据读取函数，以便从持久化存储中提取 `kvserver` 数据。然后，在 `server.go` 文件中对提取的数据进行解码和读取。

在 `server.go` 中定义数据读取函数

在 `server.go` 中，`readPersist` 函数负责从持久化存储中读取 `kvserver` 数据。它首先检查数据是否为空或长度不足，然后使用 `labgob` 解码器从字节缓冲区中解码数据。解码成功后，将数据赋值给 `kv.data` 和 `kv.lastApplies`。

```
1. // readPersist 从持久化存储中读取 kvserver 数据
2. func (kv *KVServer) readPersist(data []byte) {
3.     if data == nil || len(data) < 1 {
4.         return // 如果数据为空或长度不足，直接返回
5.     }
6.     r := bytes.NewBuffer(data)
7.     d := labgob.NewDecoder(r)
8.     var kvData map[string]string // 创建变量作为容器
9.     var lastApplies map[int64]int64 // 创建变量作为容器
10.    // 解码数据
11.    if d.Decode(&kvData) != nil || d.Decode(&lastApplies) != nil {
12.        log.Fatal("kv read persist err") // 如果解码失败，记录致命错误
13.    } else {
14.        kv.data = kvData // 将解码后的数据赋值给 kv.data
15.        kv.lastApplies = lastApplies // 将解码后的 lastApplies 赋值给 kv.lastApplies
16.    }
17. }
```

在 raft.persister.go 中定义数据提取函数

在 raft.persister.go 中, Readkvdata 函数负责从持久化存储中提取 kvserver 数据。它通过互斥锁保护共享资源, 并返回克隆的 kv_data 数据, 以确保数据的一致性和安全性。

```
1. // Readkvdata 从持久化存储中提取 kvserver 数据
2. func (ps *Persister) Readkvdata() []byte {
3.     ps.mu.Lock()
4.     defer ps.mu.Unlock()
5.     return clone(ps.kv_data) // 返回克隆的 kv_data 数据
6. }
```

(7) 服务器启动函数 StartKVServer

为了实现键值对存储服务器的最终功能, 我们需要将各个子功能在服务器启动函数 StartKVServer 中进行组合。以下是该函数的详细实现:

```
1. // StartKVServer 初始化并启动一个键值存储服务器 (KVServer)。
2. func StartKVServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister,
maxraftstate int) *KVServer {
3.     // 注册 Op 结构体以便于 Go 的 RPC 库进行编组和解组。
4.     labgob.Register(Op{})
5.     // 创建一个新的 KVServer 实例。
6.     kv := new(KVServer)
7.     kv.me = me
8.     // 设置最大 Raft 状态。
9.     kv.maxraftstate = maxraftstate
10.    // 创建应用通道。
11.    kv.applyCh = make(chan raft.ApplyMsg)
12.    // 初始化数据存储。
13.    kv.data = make(map[string]string)
14.    // 初始化最后应用的索引。
15.    kv.lastApplies = make(map[int64]int64)
16.    // 初始化消息通知通道映射。
17.    kv.msgNotify = make(map[int64]chan NotifyMsg)
18.    // 设置持久化存储器。
19.    kv.persister = persister
20.    // 从持久化存储中读取数据和最后应用的索引。
21.    kv.readPersist(kv.persister.ReadSnapshot())
22.    // 启动一个 goroutine 来处理应用通道。
23.    go kv.handleApplyCh()
24.    return kv
25. }
```




2.4 Lab3A-测试结果

```
dbb@ubuntu: ~/桌面/3A/kvraft-3A
dbb@ubuntu: ~/桌面/3A/kvraft-3A$ go test -run 3A
Only for: ZhangRuicheng. Test: one client (3A) ...
... Passed -- 15.1 5 12901 1935
Test: ops complete fast enough (3A) ...
... Passed -- 3.4 3 6322 0
Test: many clients (3A) ...
... Passed -- 15.3 5 17371 2838
Test: unreliable net, many clients (3A) ...
... Passed -- 15.9 5 6576 1327
Test: concurrent append to same key, unreliable (3A) ...
... Passed -- 1.3 3 243 52
Test: progress in majority (3A) ...
... Passed -- 0.5 5 73 2
Test: no progress in minority (3A) ...
... Passed -- 1.0 5 153 3
Test: completion after heal (3A) ...
... Passed -- 1.0 5 46 3
Test: partitions, one client (3A) ...
... Passed -- 22.5 5 17536 2350
Test: partitions, many clients (3A) ...
... Passed -- 22.4 5 42795 3129

Test: unreliable net, restarts, partitions, many clients (3A) ...
... Passed -- 26.9 5 6424 960
Test: unreliable net, restarts, partitions, random keys, many clients (3A)
... Passed -- 29.3 7 13459 1860
PASS
ok      6.824/kvraft-3A 241.612s
dbb@ubuntu: ~/桌面/3A/kvraft-3A$
```

Lab3A 的测试结果为 241.612s

3. Lab3B-Key/value service with snapshots-实现快照的 K-V 服务

3.1 任务目标

本部分实验，要求完成 MIT6.824 的 3B 部分功能，即在 3A 基础上实现包含日志压缩的 key/value 服务。为了实现 kvserver 的数据持久化和快照功能，我们需要结合 Lab2 中的 D 部分，完成 saveSnapshot 函数以创建快照。初始化时传入的 maxraftstate 表示持久化 Raft 状态的最大允许大小。当当前 Raft 日志的大小超过该值时，就可以进行一次快照。

Lab2D 主要处理实验中重启的服务器会重放完整的 Raft 日志以恢复其状态。然而，对于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。因此，我们可以创建一个快照，快照存储了在某一个索引之前的所有日志的一个状态，这样 Raft 就可以丢弃快照之前的日志条目。结果是更少量的持久化数据和更快的重启。

具体要完成的任务如下：

1. **修改 Raft**: 完成 Raft 的 2D 功能, 实现快照, 使得 Raft 仅保留尾部的一部分日志。每次更新数据时丢弃快照前的日志, 并让 Go 的垃圾回收站释放并重用这部分内存。
2. **kvserver 端**: kvserver 需要检查 Raft 日志大小是否超过阈值。当超过时, 应该保存快照并告知 Raft 丢弃旧日志。

3.2 任务要求

3B 实验中, 我们可以将任务分为以下几个关键步骤:

1. **实现基础 API**: 在 Lab2D 中实现四个基础 API, 这些 API 是快照功能的核心:
 - `sendInstallSnapshotToPeer(server int)`: 向指定的节点发送快照。
 - `InstallSnapshot(args *InstallSnapshotArgs, reply *InstallSnapshotReply)`: 节点接收到快照并进行处理。
 - `Snapshot(index int, snapshot []byte)`: `index` 是快照所包含的最后一个日志的索引, `snapshot` 是已经处理好的快照字节流, 用于生成一次快照。
 - `CondInstallSnapshot(lastIncludedTerm int, lastIncludedIndex int, snapshot []byte) bool`: 当 follower 收到 leader 的快照并判断接收后, 要生成一个快照命令, 应用该快照命令时会调用 `CondInstallSnapshot()` 函数进行处理。
2. **保存关键数据**: 在快照部分, 需要保存的主要数据包括: 服务端的 `data` 数据和用于线性化语义的 `lastApplies` 数据结构。这些数据的具体保存操作将调用 Raft 的 `Snapshot` 方法来生成快照。
3. **读取快照**: 实现 `readPersist` 函数以读取快照。该函数主要在初始化和接收到快照命令时调用, 其核心任务是读取快照中的 `data` 数据和 `lastApplies` 数据。需要注意的是, 如果不是在初始化中调用, 还需要调用 Raft 的 `CondInstallSnapshot` 来处理日志和状态同步。
4. **合理创建和读取快照**: 在系统中合理地进行快照创建和读取操作, 以确保数据的一致性和系统的高效性。
5. **整体同步调整**: 对其余函数进行整体同步调整, 包括 `AppendEntries` 函数、`start` 函数、`ticker` 函数等, 以确保整个系统在引入快照功能后的协调性和稳定性。

3.3 代码详解

3.3.1 Raft 结构体调整

在 raft 中添加 `lastSnapshotIndex` 与 `lastSnapshotTerm` 两项用于唯一标识上一个快照。该两项信息将被提供给 raft 进行快照号检查等处理。

```
1. type Raft struct {  
2.     .....2A、2B 的相关定义在此省略, 只展示 2D 新加入的数据结构  
3.     lastSnapshotIndex int // 快照中的 index
```

```
4.     lastSnapshotTerm int
5.     lastApplied      int // 此 server 的 log commit
6. }
```

接着，我们对初始化函数 `startApplyLogs` 进行相应修改。在 Raft 协议的实现中，`startApplyLogs` 函数负责定期处理已提交的日志并将它们放入 `applyCh` 通道中。此函数还包含一个重要的判断逻辑：如果发现 `rf.lastApplied` 小于 `rf.lastSnapshotIndex`，则需要进行一次快照读取。这主要是为了应对崩溃后快照的读取问题，并确保在 Raft 初始化时也能正确处理。其关键改变如下：

1. **快照处理**：如果 `lastApplied` 小于 `lastSnapshotIndex`，则调用 `CondInstallSnapshot` 安装快照并返回，避免重复处理。
2. **日志应用**：如果 `commitIndex` 大于 `lastApplied`，则创建一个 `ApplyMsg` 数组，包含从 `lastApplied` 到 `commitIndex` 的所有日志，并将它们放入 `applyCh` 通道中。
3. **更新 lastApplied**：在每次日志应用后，更新 `lastApplied` 以反映最新的已应用日志索引。

```
1. func (rf *Raft) startApplyLogs() {
2.     defer rf.applyTimer.Reset(ApplyInterval)
3.
4.     rf.mu.Lock()
5.     var msgs []ApplyMsg
6.     if rf.lastApplied < rf.lastSnapshotIndex {
7.         // 如果 lastApplied 小于 lastSnapshotIndex，表示需要安装快照
8.         msgs = make([]ApplyMsg, 0)
9.         rf.mu.Unlock()
10.        rf.CondInstallSnapshot(rf.lastSnapshotTerm, rf.lastSnapshotIndex,
rf.persister.snapshot)
11.        return
12.    } else if rf.commitIndex <= rf.lastApplied {
13.        // 如果 commitIndex 小于等于 lastApplied，表示没有新的 commit 需要应用
14.        msgs = make([]ApplyMsg, 0)
15.    } else {
16.        // 否则，创建一个 ApplyMsg 数组，包含从 lastApplied 到 commitIndex 的所有日志
17.        msgs = make([]ApplyMsg, rf.commitIndex-rf.lastApplied)
18.        for i := rf.lastApplied + 1; i <= rf.commitIndex; i++ {
19.            msgs = append(msgs, ApplyMsg{
20.                CommandValid: true,
21.                Command: rf.logs[rf.getStoreIndexByLogIndex(i)].Command,
22.                CommandIndex: i,
23.            })
24.        }
25.    }
26.    rf.mu.Unlock()
27.    for _, msg := range msgs {
```

```
28.     rf.applyCh <- msg
29.     rf.mu.Lock()
30.     rf.lastApplied = msg.CommandIndex
31.     rf.mu.Unlock()
32. }
33. }
```

3.3.2 四个基础 API-Lab2D 部分

• 1.sendInstallSnapshotToPeer-快照发送

负责将快照发送到指定的对等节点。快照的发送函数与心跳包发送函数 `appendEntries()` 流程一致，大体流程分为三个步骤：

1. 生成 `InstallSnapshotArgs`：首先，根据当前节点的状态生成 `InstallSnapshotArgs`，包括任期、领导者 ID、最后一个包含的索引和任期、数据等信息。
2. 在 RPC 超时时间内发送 RPC：使用 `time.NewTimer(RPCTimeout)` 创建一个定时器，以避免无限期等待。在超时时间内，不断地向指定节点发送 RPC 请求，直到发送成功或超时。（超时未得到回复：重发；发送失败，`ok=false`：重发）
3. 根据发送结果更新数据结构：根据从对等节点接收到的结果，进行判断并更新相应的数据结构，特别是 `matchIndex` 和 `nextIndex`。

```
1. func (rf *Raft) sendInstallSnapshotToPeer(server int) {
2.     rf.mu.Lock()
3.     args := InstallSnapshotArgs{
4.         Term:         rf.currentTerm,
5.         LeaderId:     rf.me,
6.         LastIncludedIndex: rf.lastSnapshotIndex,
7.         LastIncludedTerm: rf.lastSnapshotTerm,
8.         Data:         rf.persister.ReadSnapshot(),
9.     }
10.    rf.mu.Unlock()
11.
12.    timer := time.NewTimer(RPCTimeout)
13.    defer timer.Stop()
14.    DPrintf("%v role: %v, send snapshot to peer %v, args = %v", rf.me, rf.role, server, args)
15.
16.    for {
17.        timer.Stop()
18.        timer.Reset(RPCTimeout)
19.        ch := make(chan bool, 1)
20.        reply := &InstallSnapshotReply{}
21.        go func() {
22.            ok := rf.peers[server].Call("Raft.InstallSnapshot", &args, reply)
23.            if !ok {
24.                time.Sleep(time.Millisecond * 10)
25.            }

```

```
26.         ch <- ok
27.     }()
28.     select {
29.     case ok := <-ch:
30.         if !ok {
31.             continue
32.         }
33.         rf.mu.Lock()
34.         defer rf.mu.Unlock()
35.         if rf.role != Role_Leader || args.Term != rf.currentTerm {
36.             return
37.         }
38.         if reply.Term > rf.currentTerm {
39.             rf.changeRole(Role_Follower)
40.             rf.currentTerm = reply.Term
41.             rf.resetElectionTimer()
42.             rf.persist()
43.             return()
44.         }
45.         if args.LastIncludedIndex > rf.matchIndex[server] {
46.             rf.matchIndex[server] = args.LastIncludedIndex
47.         }
48.         if args.LastIncludedIndex+1 > rf.nextIndex[server] {
49.             rf.nextIndex[server] = args.LastIncludedIndex + 1
50.         }
51.         return
52.     }
53. }
54. }
```

• 2.InstallSnapshot-处理从收到的快照

该函数负责处理从领导者接收到的快照。当接收方接收到快照，应该执行如下策略：

1. **检查和更新任期**：当节点收到快照后，首先检查自身的任期（Term）是否大于发送方的任期。如果是，则拒绝快照。如果发送方的任期大于当前节点的任期，或者当前节点不是跟随者（Role_Follower），则更新自身的任期和角色等信息。
2. **检查快照的日志**：如果当前节点的快照包含的最后一个日志大于等于领导者快照包含的最后一个日志，则没有必要接受该快照。
3. **日志更新**：首先计算从哪个索引开始保留现有日志。
 - 如果需要保留的日志超出了现有日志的范围，则重置日志并设置为快照的最后项。
 - 如果需要保留的日志在现有日志范围内，则裁剪现有日志，只保留必要的部分。
4. **更新快照信息和持久化状态**：更新节点的快照索引和任期信息，然后将新状态和快照数据持久化存储。

```
1. func (rf *Raft) InstallSnapshot(args *InstallSnapshotArgs, reply *InstallSnapshotReply) {
```

```
2.  rf.mu.Lock()
3.  defer rf.mu.Unlock()
4.  reply.Term = rf.currentTerm
5.  if rf.currentTerm > args.Term {
6.      return
7.  }
8.  if args.Term > rf.currentTerm || rf.role != Role_Follower {
9.      rf.changeRole(Role_Follower)
10.     rf.votedFor = -1
11.     rf.currentTerm = args.Term
12.     rf.resetElectionTimer()
13.     rf.persist()
14. }
15. // 如果自身快照包含的最后一个日志 >= leader 快照包含的最后一个日志，就没必要接受了
16. if rf.lastSnapshotIndex >= args.LastIncludedIndex {
17.     return
18. }
19. // 接收发来的快照，并提交一个命令处理
20. rf.applyCh <- ApplyMsg{
21.     SnapshotValid: true,
22.     Snapshot:      args.Data,
23.     SnapshotTerm:  args.LastIncludedTerm,
24.     SnapshotIndex: args.LastIncludedIndex,
25. }
26. }
```

• 3.Snapshot-快照的创建和应用

函数负责处理快照的创建和应用。该函数在生成新的快照时，更新节点的快照信息、截断旧日志，将快照命令存入日志，并将状态和快照数据保存在持久化存储中，以减小存储开销并确保一致性。

```
1. func (rf *Raft) Snapshot(index int, snapshot []byte) {
2.     rf.mu.Lock()
3.     defer rf.mu.Unlock()
4.     snapshotIndex := rf.lastSnapshotIndex
5.     if snapshotIndex >= index {
6.         DPrintf("(Node %v) rejects replacing log with snapshotIndex %v as current
snapshotIndex %v is larger in term %v", rf.me, index, snapshotIndex)
7.         return
8.     }
9.     oldLastSnapshotIndex := rf.lastSnapshotIndex
10.    rf.lastSnapshotTerm = rf.logs[rf.getStoreIndexByLogIndex(index)].Term
11.    rf.lastSnapshotIndex = index
12.    // 删除 index 前的所有日志
13.    rf.logs = rf.logs[index-oldLastSnapshotIndex:]
}
```

```
14. // e 位置就是快照命令
15. rf.logs[0].Term = rf.lastSnapshotTerm
16. rf.logs[0].Command = nil
17. rf.persister.SaveStateAndSnapshot(rf.getPersistData(), snapshot)
18. DPrintf("(Node %v)'s state is {role %v, term %v, commitIndex %v, lastApplied %v} after
replacing log with snapshotIndex %v as old snapshotIndex %v",
19. rf.me, rf.role, rf.currentTerm, rf.commitIndex, rf.lastApplied, index,
oldLastSnapshotIndex)
20. }
```

需要指出的是，如果 `snapshotIndex` 大于等于参数 `index`，则表示新的快照的日志索引比当前已保存的快照的日志索引小或相等，因此没有必要生成新的快照。此时会打印一条日志，并直接返回，不执行后续的操作。

• 4.CondInstallSnapshot

该函数负责条件性地安装快照。该函数首先锁定节点互斥锁，然后根据传入的快照索引判断是否需要进行快照安装。如果需要安装，它会清空或截取节点的日志，将快照信息和状态更新到节点，最后将快照和状态保存到持久化存储中，并返回 `true` 表示成功安装了快照。这有助于减小存储开销并确保节点状态的一致性。

```
1. func (rf *Raft) CondInstallSnapshot(lastIncludedTerm int, lastIncludedIndex int, snapshot
[]byte) bool {
2.     rf.mu.Lock()
3.     defer rf.mu.Unlock()
4.
5.     // 判断接收快照后 logs 中还有没有多余的日志并进行不同的处理
6.     _, lastIndex := rf.getLastLogTermAndIndex()
7.     if lastIncludedIndex > lastIndex {
8.         rf.logs = make([]LogEntry, 1)
9.     } else {
10.         installLen := lastIncludedIndex - rf.lastSnapshotIndex
11.         rf.logs = rf.logs[installLen:]
12.         rf.logs[0].Command = nil
13.     }
14.
15.     // 0 处是空日志，代表了快照日志的标记
16.     rf.logs[0].Term = lastIncludedTerm
17.     rf.lastSnapshotIndex, rf.lastSnapshotTerm = lastIncludedIndex, lastIncludedTerm
18.     rf.lastApplied, rf.commitIndex = lastIncludedIndex, lastIncludedIndex
19.     // 保存快照和状态
20.     rf.persister.SaveStateAndSnapshot(rf.getPersistData(), snapshot)
21.     return true
22. }
```

3.3.3 4 个执行 API

• 1.saveSnapshot-保存快照

```
1. // saveSnapshot 函数用于在满足条件时创建并保存快照
2. func (kv *KVServer) saveSnapshot(logIndex int) {
3.     // 检查是否需要创建快照
4.     if kv.maxraftstate == -1 || kv.persister.RaftStateSize() < kv.maxraftstate {
5.         return
6.     }
7.
8.     // 生成快照数据
9.     w := new(bytes.Buffer) // 创建一个字节缓冲区用于存储快照数据
10.    e := labgob.NewEncoder(w) // 创建一个编码器，用于将数据编码为字节流
11.    if err := e.Encode(kv.data); err != nil { // 编码 kv.data 失败
12.        panic(err)
13.    }
14.    if err := e.Encode(kv.lastApplies); err != nil { // 编码 kv.lastApplies 失败
15.        panic(err)
16.    }
17.    data := w.Bytes() // 获取编码后的字节流数据
18.
19.    // 调用 Raft 的 Snapshot 方法保存快照
20.    kv.rf.Snapshot(logIndex, data)
21. }
```

该函数用于生成并保存快照：分为三步处理：

- 快照生成判断。如果 maxraftstate 不为-1，且当前 raft 的日志数量大于等于 maxraftstate，就进行一次快照生成；
- 生成快照数据。数据就是两种：服务端的 data 数据、用于线性化语义的 lastApplies 数据结构；
- 调用 Snapshot 生成一次快照。具体逻辑就不列出了，是 2D 部分的代码，主要就是：删除多余日志、保存快照内容、修改 raft 状态。

• 2.readPersist-读取快照

在 kvserver 端，一旦接收到 applych 管道中的 res == True 消息（快照应用指令），则应及时调用 readPersist 进行数据库的快照读取。代码如下所示

```
1. func (kv *KVServer) readPersist(isInit bool, snapshotTerm, snapshotIndex int, data []byte) {
2.     if data == nil || len(data) < 1 {
3.         return
4.     }
5.     // 只要不是初始化调用，即如果收到一个 Snapshot 命令，就要执行该函数
6.     if !isInit {
7.         res := kv.rf.CondInstallSnapshot(snapshotTerm, snapshotIndex, data)
8.         if !res {
```



```
9.         log.Panicln("kv read persist err in CondInstallSnapshot!")
10.        return
11.    }
12. }
13. // 对数据进行同步
14. r := bytes.NewBuffer(data)
15. d := labgob.NewDecoder(r)
16. var kvData map[string]string
17. var lastApplies map[int64]int64
18. if d.Decode(&kvData) != nil || d.Decode(&lastApplies) != nil {
19.     log.Fatal("kv read persist err!")
20. } else {
21.     kv.data = kvData
22.     kv.lastApplies = lastApplies
23. }
24. }
```

该函数用于读取快照内容，两处调用：初始化阶段；从 applyCh 中收到 Snapshot 命令，即接收了 leader 的 Snapshot。主要逻辑分为两步：

- 判断是否是初始化，如果不是，就要调用 CondInstallSnapshot 来处理日志和状态同步，不调用的话，后续节点就会同步失败；至于为什么初始化阶段不需要调用，因为 raft 初始化的一段时间内，也会调用 CondInstallSnapshot 函数进行初始化；
- 从快照中获取服务端的数据、用于线性化语义的 lastApplies 数据结构。

• 3.StartKVserver-调用时机

在上述 2D 功能中，已经实现了对 raft 的快照压缩功能。为了实现 kvserver 的快照应用，还需要对 kvserver 持久化部分内容进行修改，包括在 raft 节点快照更新时，及时调用 kvserver 的持续化函数进行数据库的更新。该部分功能需要联合 raft 与 kvserver 共同实现。

每一次 server 初始化时，就要保存传入的 persister 和 maxraftstate，以及根据 snapshot 读取快照的内容。

```
1. func StartKVServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister,
maxraftstate int) *KVServer {
2.     labgob.Register(Op{})
3.     kv := new(KVServer)
4.     kv.me = me
5.     kv.maxraftstate = maxraftstate
6.     kv.persister = persister
7.     kv.lastApplies = make(map[int64]int64)
8.     kv.data = make(map[string]string)
9.     kv.stopCh = make(chan struct{})
10.    kv.readPersist(true, 0, 0, kv.persister.ReadSnapshot())
11.
12.    kv.commandNotifyCh = make(map[int64]chan CommandResult)
```

```
13. kv.applyCh = make(chan raft.ApplyMsg)
14. kv.rf = raft.Make(servers, me, persister, kv.applyCh)
15.
16. go kv.handleApplyCh()
17. return kv
18. }
```

StartKVServer 函数用于初始化并启动键值存储服务器 (KVServer)。该函数首先注册了 Op 结构体，然后创建一个新的 KVServer 实例并设置其属性。它初始化了 lastApplies 和 data 数据结构，用于存储命令索引和键值对数据。接着，它创建了 stopCh 通道和 commandNotifyCh 映射，用于处理命令通知和应用消息。然后，函数创建了一个 Raft 实例 kv.rf，用于处理一致性协议。最后，启动了一个 goroutine 来处理应用通道，确保命令能够被正确应用。函数返回初始化后的 KVServer 实例，准备接收和处理来自客户端的请求。

• 4.handleApplyCh-执行命令

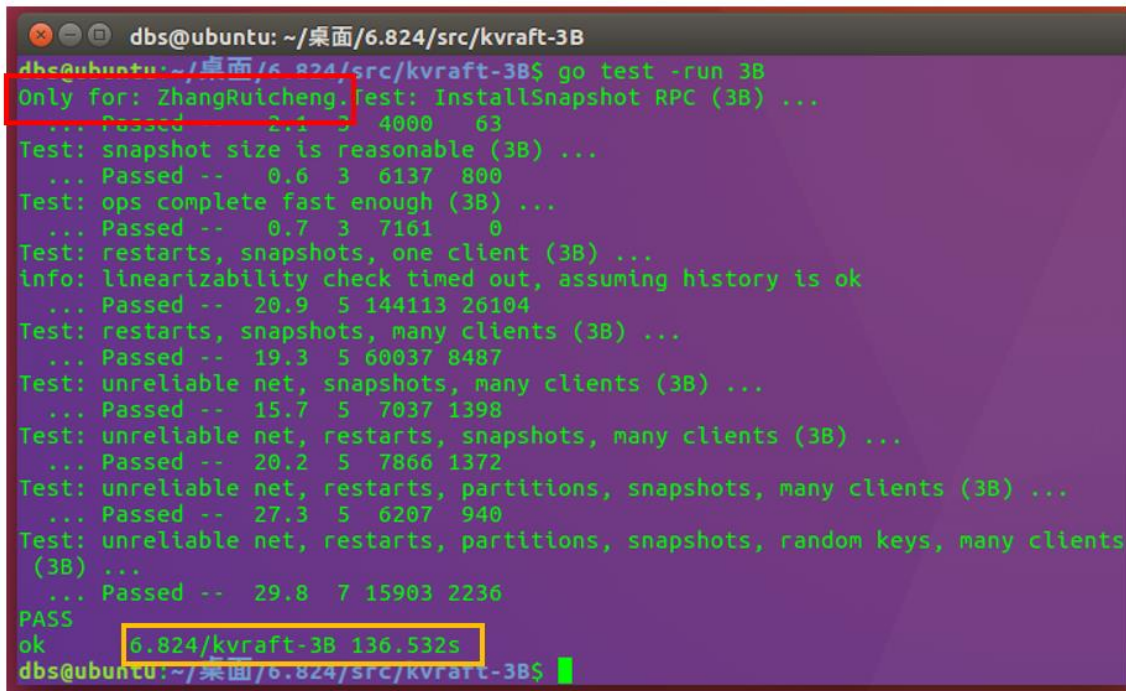
handleApplyCh 函数负责处理来自 applyCh 通道的命令。该函数在一个无限循环中监听 applyCh 通道，从中接收命令。对于每个接收到的命令，如果命令是快照命令，则读取快照内容并更新状态。对于其他命令，函数会打印命令信息，并根据需要调用 saveSnapshot 函数进行数据持久化。这确保了系统在处理命令时能够正确地更新状态，并且在必要时进行数据的持久化操作。

在 applyCh 的处理中，如果接收到一条快照命令，就要调用 readPersist 函数进行一次快照的读取，每执行完一条普通命令，就要调用 saveSnapshot 函数判断是否进行一次快照生成。

```
1. func (kv *KVServer) handleApplyCh() {
2.     for {
3.         select {
4.             case <-kv.stopCh:
5.                 DPrintf("get from stopCh, server-%v stop!", kv.me)
6.                 return
7.             case cmd := <-kv.applyCh:
8.                 // 处理快照命令，读取快照的内容
9.                 if cmd.SnapshotValid {
10.                    DPrintf("%v get install sn,%v %v", kv.me, cmd.SnapshotIndex, cmd.SnapshotTerm)
11.                    kv.lock("waitApplyCh_sn")
12.                    kv.readPersist(false, cmd.SnapshotTerm, cmd.SnapshotIndex, cmd.Snapshot)
13.                    kv.unlock("waitApplyCh_sn")
14.                    continue
15.                }
16.                // 每应用一条命令，就判断是否进行持久化
17.                DPrintf("apply op: cmdId:%d, op: %v, data:%v", cmdIdx, op, kv.data[op.Key])
18.                kv.saveSnapshot(cmdIdx)
19.                kv.unlock("handleApplyCh")
20.            }
21.        }
22.    }
```

22. }

3.4 Lab3B-测试结果



```
dbx@ubuntu: ~/桌面/6.824/src/kvraft-3B
dbx@ubuntu:~/桌面/6.824/src/kvraft-3B$ go test -run 3B
Only for: ZhangRuicheng, test: InstallSnapshot RPC (3B) ...
... Passed -- 2.1 3 4000 63
Test: snapshot size is reasonable (3B) ...
... Passed -- 0.6 3 6137 800
Test: ops complete fast enough (3B) ...
... Passed -- 0.7 3 7161 0
Test: restarts, snapshots, one client (3B) ...
info: linearizability check timed out, assuming history is ok
... Passed -- 20.9 5 144113 26104
Test: restarts, snapshots, many clients (3B) ...
... Passed -- 19.3 5 60037 8487
Test: unreliable net, snapshots, many clients (3B) ...
... Passed -- 15.7 5 7037 1398
Test: unreliable net, restarts, snapshots, many clients (3B) ...
... Passed -- 20.2 5 7866 1372
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
... Passed -- 27.3 5 6207 940
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (3B) ...
... Passed -- 29.8 7 15903 2236
PASS
ok      6.824/kvraft-3B 136.532s
dbx@ubuntu:~/桌面/6.824/src/kvraft-3B$
```

Lab3B 的测试结果为 136.532s

5. 实验总结

5.1 困难总结

本次实验不仅是对分布式系统和计算机网络的深入探索，更是一次对容错性、一致性保障和系统性能的深层次理解之旅。以下是我在实验中遇到的几点印象深刻的困难：

实验 3A 部分

<1> 无论用户是否重复请求，首先应将请求消息交给 Raft 集群进行一致性共识并存储日志。在此过程中，我最初犯了原理性的错误，后来通过参考示例代码才艰难地调试成功。因此，应先熟悉基本原理与知识。

<2> 超时机制对 RPC 的发送效率具有很大的提高作用。在超时机制下，节点不必因 RPC 发送卡顿而苦苦等待，降低发送效率，而是在超时后可以及时重发，尽管这会导致一些重复的 RPC 数据包。因此，本次实验中我尝试在发送 RPC 的地方都尽量加入超时机制。然而，加入超时机制后，应对相关接收管道进行删除。但在 3A 中，删除的管道是与其他进程共用

的，导致一个进程删除该管道后其他进程出现资源冲突。因此，对公共资源的删除应做好检测，避免造成资源冲突。

<3> 在 3A 中，我最初将 `kvserver` 与 `Raft` 共用相同的持久化函数，但由于两者数据写入的时间与内容不一致，导致数据覆盖和冲突。最终通过分离数据解决了问题。

实验 3B 部分

<1> 经过之前几次任务，实验 3B 部分已经具有一定经验，虽然不是特别难编写，但其离谱之处在于它会暴露我们前面 `Raft` 部分的一些问题，因此碰到问题时，我们通过查看日志进行查找和解决。

<2> `AppendEntries` RPC 日志内容的深拷贝问题在 2B 中未被发现，而是在 3B 中被发现。切片默认进行的是浅拷贝，因此在 `leader` 向某个 `follower` 发送 `AppendEntries` RPC 时，在生成 `args` 到发送 RPC 这段时间内（这段时间没有上锁，上锁就太浪费资源了），如果进行一次快照（生成快照），那么就可能导致 `args` 的 `logEntries` 发生变化，原本存在的数据，进行一次快照后可能会进行删除，进而导致发送给 `follower` 的 `logEntries` 的某些日志的 `Command` 变为 `nil` 了。在 `kvServer` 应用这个命令时，进行接口的转化（`op := cmd.Command.(Op)`）就会报错。

5.2 实验心得

本次实验是对 `Raft` 协议的一次实战应用。在实践中，我进一步加深了对 `Raft` 协议的认识和理解，将之前学习的 `Raft` 协议理论知识应用于实际的工程实践中。这个实验不仅仅是对 `Raft` 协议概念的一个编程实现，更是一个深入理解和体验分布式系统核心原理的过程。在实现 2D 时，我需要反复回顾 lab2 中的设计思路，这让我对领导者选举、日志复制、安全性以及日志压缩等概念有了更深刻的理解。

在整个实验过程中，我不断地进行调试和测试，这不仅提高了我的编程技能，也极大提升了我的 `debug` 能力。在调试过程中，我学会了如何更有效地识别和解决问题，这对于我未来在分布式系统设计和实现中将是一个宝贵的技能。

此外，通过这次实验，我体会到了分布式系统设计和实现的乐趣。在解决实际的工程实践中，我能够看到理论知识如何在现实世界中发挥作用，这让我对分布式计算领域产生了更浓厚的兴趣。我期待在未来的学习和工作中，能够进一步探索和应用这些概念，为构建更加复杂和强大的分布式系统做出贡献。

6. 相关参考资料

1. https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md
2. <https://raft.github.io/raft.pdf>
3. <https://pdos.csail.mit.edu/6.824/papers/mapreduce.pdf>

4. <https://github.com/OneSizeFitsQuorum/MIT6.824-2021>
5. <https://github.com/chaozh/MIT-6.824>
6. <https://www.bilibili.com/video/av87684880>
7. <https://shimo.im/docs/xwqvh3kGppJKvHvX?fallback=1>