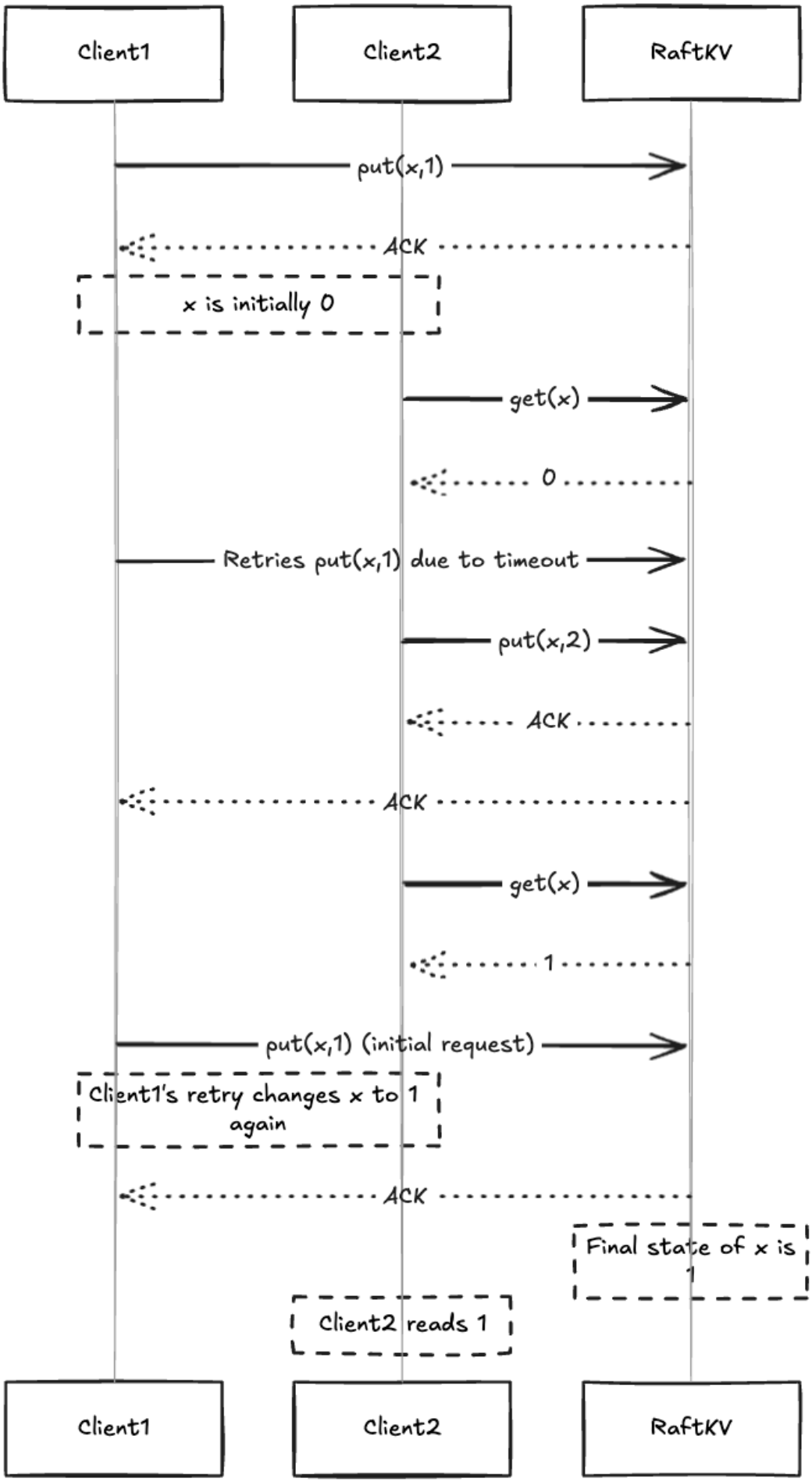
Lab4A

总体

lab4需要我们基于lab3实现的Raft，实现一个可用的KV服务，这意味着我们需要保证线性一致性**（要求从外部观察者的角度来看，所有操作都按照某个全局顺序执行，并且结果与这些操作按该顺序串行执行的结果相同）**。尽管 Raft 共识算法本身支持线性化语义，但要真正保证线性化语义在整个系统中生效，仍然需要上层服务的配合。

例如，在下面这张图中：x初始值为0，client1发送put请求(x,1)，client2发送put请求(x,2)，并在put请求前后发送get请求，此时如果put请求因为超时不断重发，如果在client2的put请求之后才被应用，则导致最后client2读到的是1，RaftKV的结果也是1，这就违背了线性一致性。


Client1
Client2
RaftKV
put(x,1)
ACK
x is initially 0
get(x)
0
Retries put(x,1) due to timeout
put(x,2)
ACK
ACK
get(x)
1
put(x,1) (initial request)
Client1's retry changes x to 1 again
ACK
Final state of x is 1
Client2 reads 1
Client1
Client2
RaftKV

这是因为当客户端向服务端提交command时，服务端在Raft层中同步、提交并应用后，客户端因为没有收到请求回复，会重试此操作，这种重试机制会导致相同的命令被执行多次。注意，这里讨论的都是写请求，因为读请求不会改变系统状态，可以重复执行多次。

为了解决重复执行命令导致线性一致性破坏的问题，Raft 作者提出了一种解决方案：**客户端为每个命令分配一个唯一的序列号。状态机会记录每个客户端的最新序列号及其对应的执行结果。如果一个命令的序列号已经被处理过，则系统会直接返回先前的结果，而不会重新执行该命令。**这样可以确保每个命令只被应用到状态机一次，避免了重复执行可能带来的线性一致性问题。

在这个lab中，我们可以按照如下机制具体实现：

1. 客户端命令唯一化：每个客户端发送给服务端的每个command请求都携带一个由ClientId和CommandId组成的二元组。ClientId是客户端的唯一标识符，CommandId是一个递增的整数，用于唯一标识客户端发出的每一个命令。
2. 服务器端状态记录：在服务器端，维护一个映射表，这个映射表以ClientId作为主键，其值是一个结构体包含：
    - 最近执行的来自该客户端的CommandId。
    - 对应的命令执行结果。
3. 重复命令检测与处理：
    - 当一个新命令到达时，首先检查映射表中是否存在对应的ClientId条目。
    - 如果存在，则比较新命令的CommandId与映射表中记录的CommandId。
        - 如果新命令的CommandId小于或等于记录的CommandId，则说明这是一个重复命令，服务器可以直接返回之前存储的结果。
        - 如果新命令的CommandId大于记录的CommandId，则说明这是新的命令，服务器应该正常处理这个命令，并更新映射表中对应ClientId的CommandId及结果。
    - 如果不存在对应的ClientId条目，则将此命令视为首次出现的命令进行处理，并添加一个新的条目到映射表中。

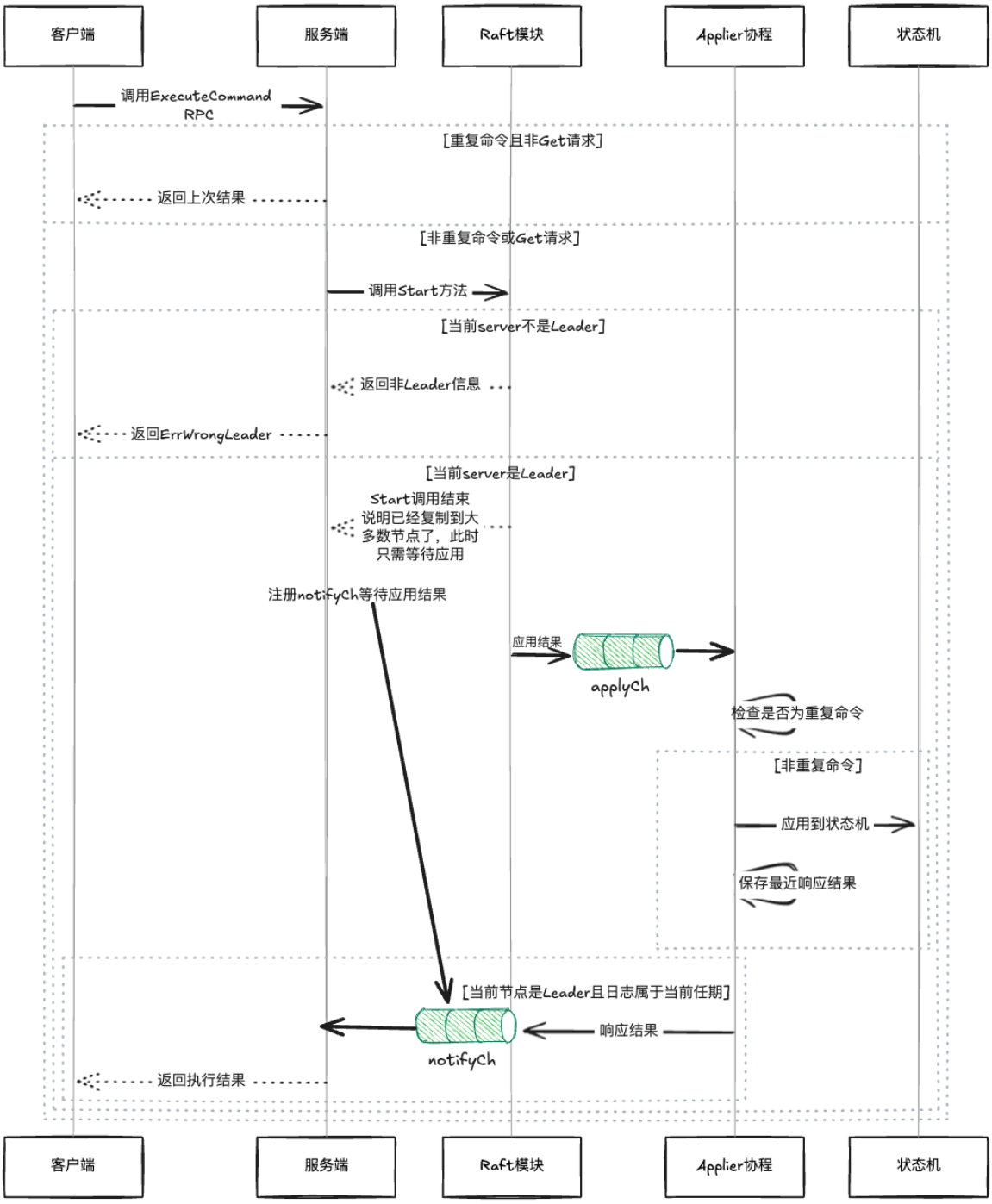
Raft 的架构

- Raft 协议将状态机放在服务端：
    - 客户端发送命令（如 `Put("x", "5")`）给 Raft 领导者。
    - 领导者将命令写入日志，复制到所有节点。
    - 各节点的 Raft 层提交日志后，调用本地状态机执行命令。
- 在 Lab 4 中，`KVServer` 是 Raft 的上层应用，`KVStateMachine` 作为其状态机，运行在服务端。

客户端的职责

- 客户端（`Clerk`）只负责：
    - 构造命令（`CommandArgs`）。
    - 通过 RPC 发送命令到服务端（`KVServer.ExecuteCommand`）。
    - 处理响应（`CommandReply`）。

Lab4A:无快照



客户端

对于客户端，需要有(clientId, commandId)来标识唯一命令，对于clientId，通过lab提供的随机数生成器nrand生成即可，对于commandId，可以采用递增的方式进行管理。这意味着每当客户端发送一个新的命令时，commandId都会递增一次，从而确保每个命令都有一个唯一的标识符，这样也需要保证如果这条命令没处理完（请求的server不是leader或者请求超时）需重复执行的时候，不能改变commandId。
客户段结构体涉及如下

```
type Clerk struct {
    servers []*labrpc.ClientEnd
    // You will have to modify this struct.
    leaderId  int
    clientId  int64
    commandId int64
}
```

```
// MakeClerk 构造客户端
func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
    InitKVLogger()
    ck := &Clerk{
        servers:   servers,
        leaderId:  0,
        clientId:  nrand(),
        commandId: 0,
    }
    return ck
}
```

客户端调用RPC

```
// ExecuteCommand RPC调用
//src/kvraft/kvsm.go
func (ck *Clerk) ExecuteCommand(args *CommandArgs) string {
    args.ClientId, args.CommandId = ck.clientId, ck.commandId
    for {
        reply := new(CommandReply)
        if !ck.servers[ck.leaderId].Call("KVServer.ExecuteCommand", args, reply)
|| reply.Err == ErrWrongLeader || reply.Err == ErrTimeout {
            ck.leaderId = (ck.leaderId + 1) % len(ck.servers)
            continue
        }
        ck.commandId += 1
        return reply.Value
    }
}
```

三种操作的实现

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

KV状态机

```
package kvraft
//src/kvraft/kvsm.go
// KVStateMachine kv状态机接口类型
type KVStateMachine interface {
    Get(key string) (string, Err)
    Put(key, value string) Err
    Append(key, value string) Err
}

type MemoryKV struct {
    KV map[string]string
}

func (memoryKV *MemoryKV) Get(key string) (string, Err) {
    if value, ok := memoryKV.KV[key]; ok {
        return value, OK
    }
    return "", ErrNoKey
}

func (memoryKV *MemoryKV) Put(key, value string) Err {
    memoryKV.KV[key] = value
    return OK
}

func (memoryKV *MemoryKV) Append(key, value string) Err {
    memoryKV.KV[key] += value
    return OK
}
```

公用资源

```
package kvraft

import (
    "fmt"
    "time"
)
//src/kvraft/common.go
// 用来设置超时时间
const ExecuteTimeout = 1000 * time.Millisecond

// Debugging
const Debug = true

func DPrintf(format string, a ...interface{}) {
    if Debug {
        SugarKVLogger.Infof(format, a...)
        // log.Printf(format, a...)
    }
}

// 设置三种操作状态
type OpType uint8
```

```go
const (
    OpPut OpType = iota
    OpAppend
    OpGet
)

func (opType OpType) String() string {
    switch opType {
    case OpPut:
        return "Put"
    case OpAppend:
        return "Append"
    case OpGet:
        return "Get"
    }
    panic(fmt.Sprintf("unexpected OpType %d", opType))
}

// CommandArgs 客户端请求执行命令RPC参数
type CommandArgs struct {
    Key       string
    Value     string
    Op        OpType
    ClientId  int64
    CommandId int64
}

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

type CommandReply struct {
    Err   Err
    Value string
}

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

// OperationContext 用于记录某个客户端的操作历史和状态
type OperationContext struct {
    MaxAppliedCommandId int64
    LastReply           *CommandReply
}

type Command struct {
    *CommandArgs
}

// kvraft响应客户端的错误状态
type Err uint8

const (
    OK Err = iota
    ErrNoKey
```

```
    ErrWrongLeader
    ErrTimeout
)

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))
}
```

服务端

```
// KVServer 负责协调 Raft 协议和上层键值存储逻辑，确保线性一致性
type KVServer struct {
    mu      sync.RWMutex
    me      int
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg
    dead    int32 // set by Kill()

    maxraftstate int // snapshot if log grows this big
    lastApplied  int //避免重复应用，记录最后的应用索引

    // Your definitions here.
    stateMachine   KVStateMachine             //表示键值存储的状态机
    lastOperations map[int64]OperationContext //记录每个客户端的最新操作上下文，用于去重和结果缓存
    notifyChs      map[int]chan *CommandReply //用于通知等待命令执行结果的客户端
}
```

KV服务端RPC调用逻辑

```
// ExecuteCommand 处理客户端发送的命令请求（例如 Get、Put 或 Append 操作），并确保命令在分布式系统中正确执行。
func (kv *KVServer) ExecuteCommand(args *CommandArgs, reply *CommandReply) {
    kv.mu.RLock()
    //如果不是Get命令，且是重复命令， 则直接返回旧值
    if args.Op != OpGet && kv.isDuplicatedCommand(args.ClientId, args.CommandId)
{
        lastReply := kv.lastOperations[args.ClientId].LastReply
        reply.Value, reply.Err = lastReply.Value, lastReply.Err
        kv.mu.RUnlock()
        return
    }
    kv.mu.RUnlock()
    //是Get命令，不确定是不是以及执行过的

    // 如果是Get命令，重复执行也没事
```

```
    // 如果不是Get命令，则说明该命令不是重复命令需要执行
    //提交命令到 Raft
    index, _, isLeader := kv.rf.Start(Command{args})
    if !isLeader {
        reply.Err = ErrWrongLeader
        return
    }

    //创建通知通道并等待结果
    kv.mu.Lock()
    ch := kv.getNotifyCh(index)
    kv.mu.Unlock()

    // 使用 select 等待：
    // 如果通道 ch 返回结果（result），说明命令已成功应用，填充 reply。
    // 如果超时（ExecuteTimeout），返回 ErrTimeout。
    select {
    case result := <-ch:
        reply.Value, reply.Err = result.Value, result.Err
    case <-time.After(ExecuteTimeout):
        reply.Err = ErrTimeout
    }

    //清理通知通道
    go func() {
        kv.mu.Lock()
        kv.deleteNotifyCh(index)
        kv.mu.Unlock()
    }()
}
```

检查是否是重复命令

```
// isDuplicatedCommand 检查是否是重复命令
func (kv *KVServer) isDuplicatedCommand(clientId, commandId int64) bool {
    OperationContext, ok := kv.lastOperations[clientId]
    return ok && commandId <= OperationContext.MaxAppliedCommandId
}
```

创建和删除响应通道，用于监已提交的日志信息

```
// getNotifyCh 检查 notifyChs 中是否已有 index 对应的通道。
func (kv *KVServer) getNotifyCh(index int) chan *CommandReply {
    if _, ok := kv.notifyChs[index]; !ok {
        kv.notifyChs[index] = make(chan *CommandReply, 1)
    }
    return kv.notifyChs[index]
}

// deleteNotifyCh 从 notifyChs 中删除指定索引的通道
func (kv *KVServer) deleteNotifyCh(index int) {
    delete(kv.notifyChs, index)
}
```

`applier()` 是一个无限循环的后台线程，监听 `kv.applyCh` 通道，从 Raft 层接收已提交的日志消息（`ApplyMsg`），并：

1. 将命令应用到状态机（`stateMachine`），更新键值存储状态。
2. 处理重复命令（去重），确保线性一致性。
3. 通知客户端命令执行结果。

```
func (kv *KVServer) applier() {
    for !kv.killed() {
        select {
        case message := <-kv.applyCh:
            DPrintf("{Node %v} tries to apply message %v", kv.rf.GetId(),
message)
            // 检查 message.CommandValid 是否为 true，表示这是一个命令消息（而不是快照消
息）
            if message.CommandValid {
                kv.mu.Lock()
                //当前条目，已经被raft应用到集群  （可能是快照恢复导致的重复消息）
                if message.CommandIndex <= kv.lastApplied {
                    DPrintf("{Node %v} discards outdated message %v because a
newer snapshot which lastApplied is %v has been restored", kv.rf.GetId(),
message, kv.lastApplied)
                    kv.mu.Unlock()
                    continue //为什么continue？ continue为继续执行循环
                }
                kv.lastApplied = message.CommandIndex

                reply := new(CommandReply)
                command := message.Command.(Command) // type assertion
                //如果不是Get命令，且是重复命令， 则直接返回旧值
                if command.Op != OpGet &&
kv.isDuplicatedCommand(command.ClientId, command.CommandId) {
                    DPrintf("{Node %v} doesn't apply duplicated message %v to
stateMachine because maxAppliedCommandId is %v for client %v", kv.rf.GetId(),
message, kv.lastOperations[command.ClientId], command.ClientId)
                    reply = kv.lastOperations[command.ClientId].LastReply
                } else {
                    //要么是get命令，要么是新命令; Get 是只读操作，无副作用，可重复执行，不
需去重。
                    //应用到状态机
                    reply = kv.applyLogToStateMachine(command)
                    if command.Op != OpGet { //该命令是新index的命令
                        //保存最后应用
                        kv.lastOperations[command.ClientId] = OperationContext{
                            MaxAppliedCommandId: command.CommandId,
                            LastReply:           reply,
                        }
                    }
                }

                //
                if currentTerm, isLeader := kv.rf.GetState(); isLeader &&
message.CommandTerm == currentTerm {
                    ch := kv.getNotifyCh(message.CommandIndex)
                    ch <- reply
                }
                kv.mu.Unlock()
            }
        }
    }
```

```
}
```

KVServer启动逻辑

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

    kv := &KVServer{
        mu:             sync.RWMutex{},
        me:             me,
        rf:             raft.Make(servers, me, persister, applyCh),
        applyCh:        applyCh,
        dead:           0,
        maxraftstate:   maxraftstate,
        stateMachine:   &MemoryKV{KV: make(map[string]string)},
        lastOperations: make(map[int64]OperationContext),
        notifyChs:      make(map[int]chan *CommandReply),
    }
    // You may need initialization code here.

    go kv.applier()

    return kv
}
```

测试

```
go test -run 4A
```

```
go test -run 4A
Test: one client (4A) ...
  ... Passed --  15.0  5 50729 5196
Test: ops complete fast enough (4A) ...
  ... Passed --   1.8  3 24128    0
Test: many clients (4A) ...
  ... Passed --  15.1  5 115679 6625
Test: unreliable net, many clients (4A) ...
  ... Passed --  16.0  5  7584 1421
Test: concurrent append to same key, unreliable (4A) ...
  ... Passed --   1.9  3   260   52
Test: progress in majority (4A) ...
  ... Passed --   1.2  5    96    2
Test: no progress in minority (4A) ...
  ... Passed --   1.0  5   173    3
Test: completion after heal (4A) ...
  ... Passed --   1.0  5    53    3
Test: partitions, one client (4A) ...
  ... Passed --  22.7  5 85424 4087
Test: partitions, many clients (4A) ...
  ... Passed --  22.4  5 372709 6269
Test: restarts, one client (4A) ...
```

```
  ... Passed --  21.8  5 121415 5182
Test: restarts, many clients (4A) ...
  ... Passed --  23.6  5 526431 6910
Test: unreliable net, restarts, many clients (4A) ...
  ... Passed --  22.8  5  9335 1402
Test: restarts, partitions, many clients (4A) ...
  ... Passed --  29.1  5 274878 6010
Test: unreliable net, restarts, partitions, many clients (4A) ...
  ... Passed --  29.7  5  6814  608
Test: unreliable net, restarts, partitions, random keys, many clients (4A) ...
  ... Passed --  31.2  7 22028 2284
PASS
ok      6.5840/kvraft   257.589s
```