Lab5B

前言

实现了一个分布式分片键值存储系统，基于 Raft 一致性协议和分片控制器（`ShardCtrler`）管理动态分片和副本组

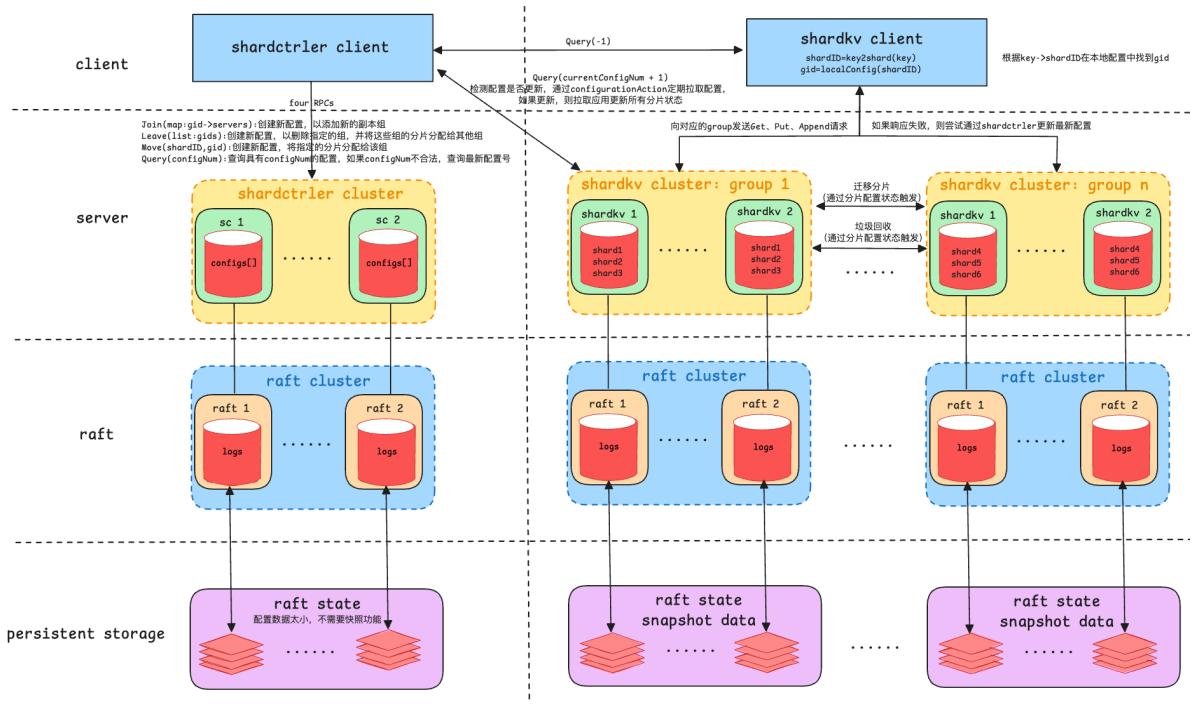
整体思路

`ShardKV` 是一个支持动态分片迁移的键值存储服务，通过 Raft 保证一致性，结合 `ShardCtrler` 管理分片分配。核心目标是：

- **键值操作**：处理客户端的 `Get`、`Put`、`Append` 请求，确保数据一致性和分片正确性。
- **分片管理**：响应配置变更，执行分片迁移（拉取数据）和垃圾收集（清理旧数据）。
- **一致性保证**：通过 Raft 日志和快照机制，确保副本组内和跨组操作的正确性。
- **并发优化**：使用读写锁、异步任务和定时任务，提高吞吐量。

系统运行流程：

1. **初始化**：启动服务器，恢复快照，初始化状态机和 Raft。
2. **客户端请求**：通过 `Command` 处理键值操作，验证分片和服务状态。
3. **配置变更**：定时检查新配置（`configurationAction`），更新分片分配。
4. **分片迁移**：拉取数据（`migrationAction`）和清理旧数据（`gcAction`）。
5. **日志应用**：通过 `applier` 将 Raft 日志应用到状态机，处理操作、配置、迁移等命令。
6. **快照管理**：监控 Raft 日志大小，触发快照以优化存储。



1. **客户端层**：
   - **Shardctrler客户端**：通过Join、Leave、Move、Query四个RPC接口与shardctrler集群交互，管理分片配置。
   - **Shardkv客户端**：根据分片ID和本地配置，将Get、Put、Append请求发送到对应的shardkv服务器。若服务不可用，则向shardctrler请求最新配置并重定向。
2. **服务器层**：

    - **Shardctrler集群**：管理分片配置，维护configs[]数组（含configNum、shard->gid映射、gid->servers映射）。通过RPC处理配置变更，生成新配置并重新分配分片。
    - **Shardkv集群**：处理键值存储，划分为多个分片组（Group），每个组管理部分分片。客户端请求根据分片ID路由，服务器根据配置操作数据。分片迁移时，数据从旧组传输到新组，迁移完成后清理旧数据。
3. **Raft层**：Shardctrler和Shardkv集群依赖Raft协议实现分布式日志一致性复制。
4. **持久化存储层**：Raft状态和日志需持久化以支持故障恢复。Shardctrler配置数据较小，通常无需快照。

ShardKV 结构体的核心组件（`stateMachine`、`notifyChans`、`lastOperations`）协同工作，支持分布式键值存储和分片管理：

- `stateMachine` 通过分片状态（`Serving`、`Pulling`、`BePulling`、`GCing`）管理键值数据和迁移过程，确保数据操作和迁移的正确性。
- `notifyChans` 提供异步通知机制，保证客户端及时获取操作结果。
- `lastOperations` 实现操作去重，提升系统可靠性和效率。

Raft日志命令类型

客户端命令 (Command)：包括 Put、Append、Get 等键值操作，通过 Raft 日志提交，确保多副本一致性。

配置变更 (Config)：记录从 shardctrler 获取的新分片配置，通过 Raft 日志同步，确保所有副本组分片分配一致。

分片操作 (ShardOperation)：分片迁移相关的操作记录在 Raft 日志中，保证迁移过程的顺序和一致性。

空日志条目：Raft 生成空日志以维持领导者状态和活动性，防止集群长时间无操作。

代码实现

客户端

负责发起三种KV命令

```
type Clerk struct {
    sm       *shardctrler.Clerk
    config   shardctrler.Config
    make_end func(string) *labrpc.ClientEnd
    // You will have to modify this struct.
    leaderIds map[int]int //gid->leaderID
    clientID  int64
    commandID int64
}

// the tester calls MakeClerk.
//
// ctrlers[] is needed to call shardctrler.MakeClerk().
//
// make_end(servername) turns a server name from a
// Config.Groups[gid][i] into a labrpc.ClientEnd on which you can
// send RPCs.
func MakeClerk(ctrlers []*labrpc.ClientEnd, make_end func(string)
*labrpc.ClientEnd) *Clerk {
    InitLogger()
```

```
    ck := &Clerk{
        sm:        shardctrler.MakeClerk(ctrlers),
        make_end:  make_end,
        leaderIds: make(map[int]int),
        clientID:  nrand(),
        commandID: 0,
    }
    //从shardctrler中查询最新配置
    // DPrintf("客户端%v:初始化完成，查询最新配置", ck.clientID)
    ck.config = ck.sm.Query(-1)
    // DPrintf("客户端%v:查询最新配置完成,配置号为%d", ck.clientID, ck.config.Num)
    return ck
}
```

三种命令的RPC调用

```
func (ck *Clerk) Get(key string) string {
    return ck.Command(&CommandArgs{Key: key, Op: Get})
}

func (ck *Clerk) Put(key string, value string) {
    ck.Command(&CommandArgs{Key: key, Value: value, Op: Put})
}
func (ck *Clerk) Append(key string, value string) {
    ck.Command(&CommandArgs{Key: key, Value: value, Op: Append})
}

func (ck *Clerk) Command(args *CommandArgs) string {
    DPrintf("客户端%v发起%v请求", ck.clientID, args.Op)
    args.ClientId, args.CommandId = ck.clientID, ck.commandID
    for {
        shard := key2shard(args.Key)
        gid := ck.config.Shards[shard]
        if servers, ok := ck.config.Groups[gid]; ok {
            //如果没有设置，则设置默认的leader id为0
            if _, ok = ck.leaderIds[gid]; !ok {
                ck.leaderIds[gid] = 0
            }
            oldLeaderId := ck.leaderIds[gid]
            newLeader := oldLeaderId
            for {
                reply := new(CommandReply)
                //发送请求到领导服务器
                ok := ck.make_end(servers[newLeader]).Call("ShardKV.Command",
args, reply)
                if ok && (reply.Err == OK || reply.Err == ErrNoKey) {
                    DPrintf("客户端%v执行%v请求成功", ck.clientID, args.Op)
                    ck.commandID++
                    return reply.Value
                } else if ok && reply.Err == ErrWrongGroup {
                    DPrintf("客户端%v执行%v请求失败，重试中", ck.clientID, args.Op)
                    break
                } else {
                    //尝试下一个服务器
                    DPrintf("客户端%v执行%v请求失败，", ck.clientID, args.Op)
                    newLeader = (newLeader + 1) % len(servers)
                    //检查是否已尝试所有服务器
```

```
                    if newLeader == oldLeaderId {
                        break
                    }
                }
            }
        }
        time.Sleep(100 * time.Millisecond)
        //从shardctrler中查询最新配置
        ck.config = ck.sm.Query(-1)
    }
}
```

日志

```
package shardkv

import (
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

var ShardkvLogger *zap.SugaredLogger

func InitLogger() {
    writeSyncer := getLogWriter()
    encoder := getEncoder()
    core := zapcore.NewCore(encoder, writeSyncer, zapcore.DebugLevel)

    logger := zap.New(core)
    ShardkvLogger = logger.Sugar()
}

func getEncoder() zapcore.Encoder {
    return zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig())
}

func getLogWriter() zapcore.WriteSyncer {
    //如果想要追加写入可以查看我的博客文件操作那一章
    file, _ := os.OpenFile("/home/xy/mit2024/6.5840/src/shardkv/test.log",
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0666)
    return zapcore.AddSync(file)
}

// Debugging
const Debug = false

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

参数

用于检测Server 4个动作是否超时

```
const (
    ExecuteTimeout              = 500 * time.Millisecond
    ConfigurationMonitorTimeout = 100 * time.Millisecond
    MigrationMonitorTimeout     = 50 * time.Millisecond
    GCMonitorTimeout            = 50 * time.Millisecond
    EmptyEntryDetectorTimeout   = 200 * time.Millisecond
)
```

几种Server汇报给客户端的状态

```
type Err uint8

const (
    OK Err = iota
    ErrNoKey
    ErrWrongGroup
    ErrWrongLeader
    ErrOutDated
    ErrTimeout
    ErrNotReady
)

func (err Err) String() string {
    switch err {
    case OK:
        return "OK"
    case ErrNoKey:
        return "ErrNoKey"
    case ErrWrongGroup:
        return "ErrWrongGroup"
    case ErrWrongLeader:
        return "ErrWrongLeader"
    case ErrOutDated:
        return "ErrOutDated"
    case ErrTimeout:
        return "ErrTimeout"
    case ErrNotReady:
        return "ErrNotReady"
    default:
        panic(fmt.Sprintf("Unknown error: %d", err))
    }
}
```

命令类型，同时也同步到Raft集群

```
type CommandType uint8

const (
    Operation     CommandType = iota //处理客户端请求，修改 stateMachine 的键值数据
    Configuration                    //更新分片配置（currentConfig），触发分片迁移
    InsertShards                     //将新分片数据插入 stateMachine，对应 Pulling 状
态
```

```
    DeleteShards                        //移除分片数据，完成迁移后的 GCing 清理
    EmptyShards                         //清空分片数据，处理特殊情况
)

func (commandType CommandType) String() string {
    switch commandType {
    case Operation:
        return "Operation"
    case Configuration:
        return "Configuration"
    case InsertShards:
        return "InsertShards"
    case DeleteShards:
        return "DeleteShards"
    case EmptyShards:
        return "EmptyShards"
    default:
        panic(fmt.Sprintf("Unknown CommandType: %d", commandType))
    }
}
```

客户端的三种命令状态

```
type OperationType uint8

const (
    Get OperationType = iota
    Put
    Append
)

func (op OperationType) String() string {
    switch op {
    case Get:
        return "Get"
    case Put:
        return "Put"
    case Append:
        return "Append"
    default:
        panic(fmt.Sprintf("Unknown OperationType: %d", op))
    }
}
```

RPC参数结构体

```
type CommandArgs struct {
    Key       string
    Value     string
    Op        OperationType
    ClientId  int64
    CommandId int64
}

func (args *CommandArgs) String() string {
    return fmt.Sprintf("CommandArgs{Key: %s, Value: %s, Op: %s, ClientId: %d,
CommandId: %d}", args.Key, args.Value, args.Op, args.ClientId, args.CommandId)
}
```

RPC响应结构体

```
type CommandReply struct {
    Err   Err
    Value string
}

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

Server检查是否重复命令的结构

```
// OperationContext 存储有关操作执行上下文的信息
type OperationContext struct {
    MaxAppliedCommandId int64
    LastReply           *CommandReply
}

func (operationContext OperationContext) String() string {
    return fmt.Sprintf("OperationContext{MaxAppliedCommandId: %d, LastReply:
%v}", operationContext.MaxAppliedCommandId, operationContext.LastReply)
}
// deepCopy 分片操作的RPC回复中需要携带的参数
func (operationContext OperationContext) deepCopy() OperationContext {
    return OperationContext{
        MaxAppliedCommandId: operationContext.MaxAppliedCommandId,
        LastReply: &CommandReply{
            Err:   operationContext.LastReply.Err,
            Value: operationContext.LastReply.Value,
        },
    }
}
```

用于分片操作的参数（删除对应分片，拉取对应分片）

```
// ShardOperationArgs结构，用于与分片操作相关的参数
type ShardOperationArgs struct {
    ConfigNum int
    ShardIDs  []int
}

func (args *ShardOperationArgs) String() string {
    return fmt.Sprintf("ShardOperationArgs{ConfigNum: %d, ShardIDs: %v}",
args.ConfigNum, args.ShardIDs)
}
```

分片操作的RPC回复

```
// ShardOperationReply 用于来自分片操作的回复
type ShardOperationReply struct {
    Err            Err
    ConfigNum      int
    Shards         map[int]map[string]string
    LastOperations map[int64]OperationContext
}

func (reply *ShardOperationReply) String() string {
    return fmt.Sprintf("ShardOperationReply{Err: %s, ConfigNum: %d, Shards: %v,
LastOperations: %v}", reply.Err, reply.ConfigNum, reply.Shards,
reply.LastOperations)
}
```

封装参数类型，送入Raft命令用于保存

```
// 表示要执行的命令的命令结构
type Command struct {
    CommandType CommandType
    Data        interface{}
}
func (command Command) String() string {
    return fmt.Sprintf("Command{commandType: %s, Data: %v}",
command.CommandType, command.Data)
}
```

几种raft命令的参数设置

```
//在 ShardKV 系统中，Raft 日志包含了以下几种不同类型的操作：

// 客户端命令 (Command)：包含对键值存储的Put、Append、Get等操作，这些操作会通过 Raft 日志
提交来保证多副本的一致性。
// 配置变更 (Config)：当从 shardctrler 获取到新的分片配置时，会通过 Raft 日志来记录配置的
变化。所有副本组通过 Raft 日志共享同一配置，确保分片的一致分配。
// 分片操作 (ShardOperation)：当进行分片迁移时，涉及到的操作也会记录在 Raft 日志中，保证分
片迁移过程的顺序和一致性。
// 空日志条目：Raft 有时会生成空日志条目以保持领导者的状态和活动性，避免在某些情况下集群处于无
操作状态。

// NewOperationCommand 从CommandArgs中创建一个新的操作命令
func NewOperationCommand(args *CommandArgs) Command {
```

```
    // 客户端命令 (Command): 包含对键值存储的Put、Append、Get等操作，这些操作会通过 Raft 日志提交来保证多副本的一致性
    return Command{Operation, *args}
}

// NewConfigurationCommand创建一个新的配置命令
func NewConfigurationCommand(config *shardctrler.Config) Command {
    //// 配置变更 (Config): 当从 shardctrler 获取到新的分片配置时，会通过 Raft 日志来记录配置的变化。所有副本组通过 Raft 日志共享同一配置，确保分片的一致分配
    return Command{Configuration, *config}
}

// NewInsertShardsCommand创建一个新命令来插入分片
func NewInsertShardsCommand(reply *ShardOperationReply) Command {
    // 分片操作 (ShardOperation): 当进行分片迁移时，涉及到的操作也会记录在 Raft 日志中，保证分片迁移过程的顺序和一致性
    //创建一个 Command 类型的命令，用于将分片数据插入到当前 ShardKV 节点的 Raft 日志中
    return Command{InsertShards, *reply}
}

// NewDeleteShardsCommand创建一个删除分片的新命令
func NewDeleteShardsCommand(args *ShardOperationArgs) Command {

    //创建一个 Command 类型的命令，用于从当前 ShardKV 节点的 Raft 日志中删除分片数据
    return Command{DeleteShards, *args}
}

// NewEmptyShardsCommand创建一个新的命令，表示没有shardscommand
func NewEmptyShardsCommand() Command {
    //// 空日志条目: Raft 有时会生成空日志条目以保持领导者的状态和活动性，避免在某些情况下集群处于无操作状态
    return Command{EmptyShards, nil}
}
```

快照

```
package shardkv

import (
    "bytes"

    "6.5840/labgob"
    "6.5840/shardctrler"
)

// restoreSnapShot 从快照中恢复 shardkv状态
func (kv *ShardKV) restoreSnapShot(snapshot []byte) {
    // 从快照中存储状态机。如果快照为nil或空，则初始化状态机
    if len(snapshot) < 1 {
        kv.initStateMachines()
        return
    }
    r := bytes.NewBuffer(snapshot)
    d := labgob.NewDecoder(r)
    var stateMachine map[int]*Shard
    var lastOperations map[int64]OperationContext
    var lastConfig shardctrler.Config
```

```
    var currentConfig shardctrler.Config

    if d.Decode(&lastConfig) != nil || d.Decode(&currentConfig) != nil ||
d.Decode(&stateMachine) != nil || d.Decode(&lastOperations) != nil {
        DPrintf("{Node %v}{Group %v} fails to restore state machine from
snapshot", kv.rf.GetId(), kv.gid)
    }
    kv.lastConfig, kv.currentConfig, kv.lastOperations, kv.stateMachine =
lastConfig, currentConfig, lastOperations, stateMachine
}

// needSnapShot 查看是否需要快照
func (kv *ShardKV) needSnapShot() bool {
    return kv.maxRaftState != -1 && kv.rf.GetRaftStateSize() >= kv.maxRaftState
}

// takeSnapshot 快照当前状态
func (kv *ShardKV) takeSnapshot(index int) {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(kv.lastConfig)
    e.Encode(kv.currentConfig)
    e.Encode(kv.stateMachine)
    e.Encode(kv.lastOperations)
    data := w.Bytes()
    kv.rf.Snapshot(index, data)
}
```

分片设计

设置每个分片的状态

```
type ShardStatus uint8

// ShardStatus表示分片状态的类型

const (
    Serving          ShardStatus = iota //表示该分片正在正常地为客户端提供读写服务
    Pulling                             //表明该分片正在从其他服务器拉取数据
    BePulling                           //BePulling状态意味着该分片的数据正在被其他服务
器拉取
    GCing                               //GCing即垃圾回收（Garbage Collection）状态
    confirmMigration                    //完成迁移
)

func (status ShardStatus) String() string {
    switch status {
    case Serving:
        return "Serving"
    case Pulling:
        return "Pulling"
    case BePulling:
        return "BePulling"
    case GCing:
        return "GCing"
    case confirmMigration:
        return "confirmMigration"
```

```
    default:
        panic(fmt.Sprintf("Unknown ShardStatus: %d", status))
    }
}
```

分片操作

每个分片包含多个kv信息

```
// Shard表示一个键值存储及其状态
type Shard struct {
    KV     map[string]string
    Status ShardStatus //当前分片的状态
}

// NewShard 创建并初始化一个新的Shard实例
func NewShard() *Shard {
    return &Shard{
        KV:     make(map[string]string),
        Status: Serving,
    }
}

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

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

// deepCopy创建一个分片键值对的副本。
// 返回一个包含shard中所有键值对的新映射
func (shard *Shard) deepCopy() map[string]string {
    newShard := make(map[string]string)
    for k, v := range shard.KV {
        newShard[k] = v
    }
    return newShard
}
```

ServerKV服务器

```
type ShardKV struct {
    mu      sync.RWMutex        //互斥锁，用于保护 ShardKV 结构体的共享数据
    dead    int32               //指示节点是否被杀死
    rf      *raft.Raft          //指向 Raft 实例的指针
    applyCh chan raft.ApplyMsg //接收 Raft 提交的日志条目
```

```
    make_end func(string) *labrpc.ClientEnd //将服务器地址（string）转换为 RPC 客户端端点（*labrpc.ClientEnd）
    gid      int                            //副本组的唯一标识
    sc       *shardctrler.Clerk             //分片控制器的客户端，用于查询和更新分片配置

    maxRaftState int //Raft 日志的最大字节数，触发快照的阈值。
    lastApplied  int //最后应用的日志条目的索引，以防止statemmachine回滚
    // Your definitions here.

    lastConfig    shardctrler.Config // 上一次的分片配置（shardctrler.Config），由分片控制器提供。
    currentConfig shardctrler.Config //当前生效的分片配置

    stateMachine   map[int]*Shard             //键值存储的状态机，按分片（shard）组织，映射分片编号（int）到分片数据（*Shard）。
    lastOperations map[int64]OperationContext //记录每个客户端的最新操作，用于去重
    notifyChans    map[int]chan *CommandReply //映射 Raft 日志索引（int）到通知通道（*chan *CommandReply），用于异步通知客户端请求结果
}
```

处理来自客户端的RPC，并送入Raft

```
// Command 处理来自客户端的RPC
func (kv *ShardKV) Command(args *CommandArgs, reply *CommandReply) {
    kv.mu.RLock()
    //如果命令是重复的，直接返回结果，没有raft层的参与
    if args.Op != Get && kv.isDuplicateRequest(args.ClientId, args.CommandId) {
        lastReply := kv.lastOperations[args.ClientId].LastReply
        reply.Err, reply.Value = lastReply.Err, lastReply.Value
        kv.mu.RUnlock()
        return
    }
    //检查服务器是否可以提供请求的分片
    if !kv.canServe(key2shard(args.Key)) {
        DPrintf("服务器不能提供请求分片")
        reply.Err = ErrWrongGroup
        kv.mu.RUnlock()
        return
    }

    kv.mu.RUnlock()
    kv.Execute(NewOperationCommand(args), reply)
}
```

送入Raft集群，等待服务器的异步通知，并返回给客户端操作完成情况

```
// Execute 处理命令并通过reply参数返回结果
func (kv *ShardKV) Execute(command Command, reply *CommandReply) {
    // 不持有锁以提高吞吐量
    // 当KVServer持有锁以获取快照时，底层raft仍然可以提交raft日志
    index, _, isLeader := kv.rf.Start(command)
    if !isLeader {
        reply.Err = ErrWrongLeader
        return
```

```
    }
    defer DPrintf("{Node %v}{Group %v} Execute Command %v with CommandReply %v",
kv.rf.GetId(), kv.gid, command, reply)

    kv.mu.Lock()
    notifyChan := kv.getNotifyChan(index)
    kv.mu.Unlock()
    // 等待结果返回

    select {
    case result := <-notifyChan:
        reply.Value, reply.Err = result.Value, result.Err
    case <-time.After(ExecuteTimeout):
        reply.Err = ErrTimeout
    }
    //释放notifyChan以减少内存占用
    //为什么异步？为了提高吞吐量，这里不需要阻止客户机请求
    go func() {
        kv.mu.Lock()
        kv.removeOutdatedNotifyChan(index)
        kv.mu.Unlock()
    }()
}
```

Server端的核心调度程序

```
// 应用程序不断地将Raft日志中的命令应用到状态机。
func (kv *ShardKV) applier() {
    for !kv.killed() {
        select {
        //在apply通道中等待新消息
        case message := <-kv.applyCh:
            DPrintf("{Node %v}{Group %v} tries to applfy message %v",
kv.rf.GetId(), kv.gid, message)
            if message.CommandValid {
                kv.mu.Lock()
                //检查命令是否被应用
                if message.CommandIndex <= kv.lastApplied {
                    DPrintf("{Node %v}{Group %v} discards outdated message %v
because a newer snapshot which lastApplied is %v has been applied",
kv.rf.GetId(), kv.gid, message, kv.lastApplied)
                    kv.mu.Unlock()
                    continue
                }
                //更新最新的日志提交索引
                kv.lastApplied = message.CommandIndex

                reply := new(CommandReply)
                //断言命令
                command := message.Command.(Command)
                switch command.CommandType {
                case Operation:
                    //提取操作数据并将操作应用于状态机
                    operation := command.Data.(CommandArgs)
                    reply = kv.applyOperation(&operation)
                case Configuration:
                    nextConfig := command.Data.(shardctrler.Config)
```

```
                    reply = kv.applyConfiguration(&nextConfig)
                case InsertShards:
                    shardsInfo := command.Data.(ShardOperationReply)
                    reply = kv.applyInsertShards(&shardsInfo)
                case DeleteShards:
                    // time.Sleep(10*time.Millisecond)
                    shardsInfo := command.Data.(ShardOperationArgs)
                    reply = kv.applyDeleteShards(&shardsInfo)
                case EmptyShards:
                    DPrintf("{Node %d}{Group %v} 应用空分片信息%v", kv.rf.GetId(),
kv.gid, message)
                    reply = kv.applyEmptyShards()
                }

                if currentTerm, isLeader := kv.rf.GetState(); isLeader &&
message.CommandTerm == currentTerm {
                    notifyChan := kv.getNotifyChan(message.CommandIndex)
                    notifyChan <- reply
                }

                if kv.needSnapShot() {
                    kv.takeSnapshot(message.CommandIndex)
                }
                kv.mu.Unlock()
            } else if message.SnapshotValid {
                //从快照恢复状态机
                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 {
                panic(fmt.Sprintf("{Node %v}{Group %v} invalid apply message
%v", kv.rf.GetId(), kv.gid, message))
            }
        }
    }
}
```

一些功能函数

```
// removeOutdatedNotifyChan 删除给定索引的通知通道。
func (kv *ShardKV) removeOutdatedNotifyChan(index int) {
    // delete(kv.notifyChans, index)
    if ch, ok := kv.notifyChans[index]; ok {
        close(ch) // ☑ 先关闭通道
        delete(kv.notifyChans, index)
    }
}

// getNotifyChan 返回给定索引的通知通道
func (kv *ShardKV) getNotifyChan(index int) chan *CommandReply {
    if _, ok := kv.notifyChans[index]; !ok {
        kv.notifyChans[index] = make(chan *CommandReply, 1)
    }
```

```
    return kv.notifyChans[index]
}

// canServe 检查指定分片是否由当前副本组（GID）负责，并且分片状态是否允许服务（Serving 或
GCing）
func (kv *ShardKV) canServe(shardID int) bool {
    //GCing：分片已迁移到其他组但尚未清理，数据仍可读
    return kv.currentConfig.Shards[shardID] == kv.gid &&
(kv.stateMachine[shardID].Status == Serving || kv.stateMachine[shardID].Status
== GCing)
}

// Command 处理来自客户端的RPC
func (kv *ShardKV) isDuplicateRequest(clientId int64, commandId int64) bool {
    operationContext, ok := kv.lastOperations[clientId]
    return ok && commandId <= operationContext.MaxAppliedCommandId
}
// initStateMachines 用于初始化ShardKV状态机
func (kv *ShardKV) initStateMachines() {
    for shardID := 0; shardID < shardctrler.NShards; shardID++ {
        if _, ok := kv.stateMachine[shardID]; !ok {
            kv.stateMachine[shardID] = NewShard()
        }
    }
}

// Monitor 如果服务器是领导者，则监视特定的操作并在固定的时间间隔内重复执行该操作。
func (kv *ShardKV) Monitor(action func(), timeout time.Duration) {
    for !kv.killed() {
        if _, isLeader := kv.rf.GetState(); isLeader {
            action()
        }
        time.Sleep(timeout)
    }
}

// 返回指定状态的shard的GID到shard id的映射
func (kv *ShardKV) getShardIDsByStatus(status ShardStatus) map[int][]int {
    gid2shardIDs := make(map[int][]int)
    for shardID, shard := range kv.stateMachine {
        //过滤给定状态的分片
        if shard.Status == status {
            //找到负责该分片的最后一个gid，并从该gid中提取数据
            gid := kv.lastConfig.Shards[shardID]
            if gid != 0 {
                //按GID对分片id进行分组
                if _, ok := gid2shardIDs[gid]; !ok {
                    gid2shardIDs[gid] = make([]int, 0)
                }
                gid2shardIDs[gid] = append(gid2shardIDs[gid], shardID)
            }
        }
    }
    return gid2shardIDs
}
```

开启服务器

1. 注册RPC
2. 创建通道与对应raft绑定
3. 创建ServerKv实例
4. 启动协程用于应用
5. 启动检测服务端4中动作的协程

```
func StartServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister,
maxraftstate int, gid int, ctrlers []*labrpc.ClientEnd, make_end func(string)
*labrpc.ClientEnd) *ShardKV {
    // call labgob.Register on structures you want
    // Go's RPC library to marshall/unmarshall.
    // InitLogger()
    labgob.Register(Command{})
    labgob.Register(CommandArgs{})
    labgob.Register(shardctrler.Config{})
    labgob.Register(ShardOperationArgs{})
    labgob.Register(ShardOperationReply{})

    //创建一个通道来接收Raft应用的消息
    applyCh := make(chan raft.ApplyMsg)

    kv := &ShardKV{
        dead:           0,
        rf:             raft.Make(servers, me, persister, applyCh),
        applyCh:        applyCh,
        make_end:       make_end,
        gid:            gid,
        sc:             shardctrler.MakeClerk(ctrlers),
        maxRaftState:   maxraftstate,
        lastApplied:    0,
        lastConfig:     shardctrler.DefaultConfig(),
        currentConfig:  shardctrler.DefaultConfig(),
        stateMachine:   make(map[int]*Shard),
        lastOperations: make(map[int64]OperationContext),
        notifyChans:    make(map[int]chan *CommandReply),
    }
    kv.restoreSnapShot(persister.ReadSnapshot())
    // Your initialization code here.
    go kv.applier()

    go kv.Monitor(kv.configurationAction, ConfigurationMonitorTimeout)
    go kv.Monitor(kv.migrationAction, MigrationMonitorTimeout)
    go kv.Monitor(kv.gcAction, GCMonitorTimeout)
    go kv.Monitor(kv.checkEntryInCurrentTermAction, EmptyEntryDetectorTimeout)

    DPrintf("{Node %v}{Group %v} started", kv.rf.GetId(), kv.gid)
    return kv
}
```

服务端四种行为

发送空条目

```
// checkEntryInCurrentTermAction 确保日志条目在当前期限中存在，以保持日志活动。
func (kv *ShardKV) checkEntryInCurrentTermAction() {
    // 如果当前期限内没有日志条目，则执行空命令
    if !kv.rf.HasLogInCurrentTerm() {
        kv.Execute(NewEmptyShardsCommand(), new(CommandReply))
    }
}
```

配置变更检测

通过检查所有分片的状态是否全部是Server状态，来检测配置是否改变。如果满足条件则想ctr查询最新配置，如果当前配置老旧，则发起更新配置RPC

```
// configurationAction 检查是否可以执行下一个配置,如果所有的shard都处于服务状态，则查询并应用下一个配置。
func (kv *ShardKV) configurationAction() {
    canPerformNextConfig := true
    kv.mu.RLock()
    //如果没有分片处于服务状态，则不能应用下一个配置
    for _, shard := range kv.stateMachine {
        if shard.Status != Serving {
            canPerformNextConfig = false
            DPrintf("{Node %v}{Group %v} will not try to fetch latest
configuration because shards status are %v when currentConfig is %v",
kv.rf.GetId(), kv.gid, kv.stateMachine, kv.currentConfig)
            break
        }
    }
    currentConfigNum := kv.currentConfig.Num
    kv.mu.RUnlock()

    //如果允许，查询并应用下一个配置
    if canPerformNextConfig {
        nextConfig := kv.sc.Query(currentConfigNum + 1)
        if nextConfig.Num == currentConfigNum+1 {
            DPrintf("{Node %v}{Group %v} fetches latest configuration %v when
currentConfigNum is %v", kv.rf.GetId(), kv.gid, nextConfig, currentConfigNum)
            kv.Execute(NewConfigurationCommand(&nextConfig), new(CommandReply))
        }
    }
}
```

迁移任务

获取Pulling状态的分片旧组Gid->分片ID的映射，遍历每个组，从每个组对应KVserver获取分片信息，发送插入分片命令到Raft

```
// migrationAction 执行迁移任务，从其他组中拉出分片数据
// func (kv *ShardKV) migrationAction() {
//  kv.mu.RLock()
```

```
//  gid2Shards := kv.getShardIDsByStatus(Pulling) //需要从旧组拉出哪些shardID gid->shardId
//  var wg sync.WaitGroup                         //创建 sync.WaitGroup，用于等待所有异步拉取任务（goroutine）完成
//  // 为每个组（GID）创建pull task
//  for gid, shardIDs := range gid2Shards {
//      DPrintf("{Node %v}{Group %v} starts a PullTask to get shards %v from group %v when config is %v", kv.rf.GetId(), kv.gid, shardIDs, gid, kv.currentConfig)
//      wg.Add(1)
//      go func(servers []string, configNum int, shardIDs []int) {
//          defer wg.Done()
//          pullTaskArgs := ShardOperationArgs{configNum, shardIDs}
//          //尝试从组中的每个服务器提取分片数据
//          for _, server := range servers {
//              pullTaskReply := new(ShardOperationReply)
//              srv := kv.make_end(server)
//              // ok := srv.Call("ShardKV.GetShardsData", &pullTaskArgs, pullTaskReply)
//              // if !ok || pullTaskReply.Err == ErrWrongLeader {
//              //  // 试下一个副本
//              //  break
//              // }
//              // if pullTaskReply.Err == ErrNotReady {
//              //  // 对方还没更新到 configNum，等会儿重试
//              //  time.Sleep(10 * time.Millisecond)
//              //  continue
//              // }
//              // if pullTaskReply.Err == OK {
//              //  // 数据准备好了，提交插入命令
//              //  kv.Execute(NewInsertShardsCommand(pullTaskReply), new(CommandReply))
//              // }
//              // break
//              if srv.Call("ShardKV.GetShardsData", &pullTaskArgs, pullTaskReply) && pullTaskReply.Err == OK {
//                  //从这些服务器提取数据
//                  DPrintf("{Node %v}{Group %v} gets a PullTaskReply %v and tries to commit it when currentConfigNum is %v", kv.rf.GetId(), kv.gid, pullTaskReply, configNum)
//                  kv.Execute(NewInsertShardsCommand(pullTaskReply), new(CommandReply))
//              }
//          }
//      }(kv.lastConfig.Groups[gid], kv.currentConfig.Num, shardIDs)
//  }
//  kv.mu.RUnlock()
//  wg.Wait() //等待所有的pull任务完成
// }
// Executes the migration task to pull shard data from other groups.
func (kv *ShardKV) migrationAction() {
    kv.mu.RLock()
    gid2Shards := kv.getShardIDsByStatus(Pulling)
    var wg sync.WaitGroup

    // Create pull tasks for each group (GID)
    for gid, shardIDs := range gid2Shards {
```

```
        DPrintf("{Node %v}{Group %v} starts a PullTask to get shards %v from
group %v when config is %v", kv.rf.GetId(), kv.gid, shardIDs, gid,
kv.currentConfig)
        wg.Add(1)
        go func(servers []string, configNum int, shardIDs []int) {
            defer wg.Done()
            pullTaskArgs := ShardOperationArgs{configNum, shardIDs}
            // Try to pull shard data from each server in the group
            for _, server := range servers {
                pullTaskReply := new(ShardOperationReply)
                srv := kv.make_end(server)
                if srv.Call("ShardKV.GetShardsData", &pullTaskArgs,
pullTaskReply) && pullTaskReply.Err == OK {
                    //Pulling data from these servers
                    DPrintf("{Node %v}{Group %v} gets a PullTaskReply %v and
tries to commit it when currentConfigNum is %v", kv.rf.GetId(), kv.gid,
pullTaskReply, configNum)
                    kv.Execute(NewInsertShardsCommand(pullTaskReply),
new(CommandReply))
                }
            }
        }(kv.lastConfig.Groups[gid], kv.currentConfig.Num, shardIDs)
    }
    kv.mu.RUnlock()
    wg.Wait() // Wait for all pull tasks to complete
}
```

GC行为

获取GCing状态的gid->分片ids的对应，对于GCing状态（表明新组已经拉取完成，修改为服务状态）。

在这里调用两次删除，第一次是将旧组中的BePulling状态的数据删除，因为，只有在Gcing状态代表该分片已经迁移成功，可以直接将旧组对应分片进行删除；第二次是将新组中对应分片的状态修改为Server状态。

```
// gcAction 执行垃圾收集（GC）任务，从其他组中删除分片数据。
func (kv *ShardKV) gcAction() {
    kv.mu.RLock()
    //获取之前负责这些分片的组，并清理不再负责的分片。
    gid2Shards := kv.getShardIDsByStatus(GCing)
    var wg sync.WaitGroup

    // 为每个组创建GCtask
    for gid, shardIDs := range gid2Shards {
        DPrintf("{Node %v}{Group %v} starts a GCTask to delete shards %v from
group %v when config is %v", kv.rf.GetId(), kv.gid, shardIDs, gid,
kv.currentConfig)
        wg.Add(1)
        go func(servers []string, configNum int, shardIDs []int) {
            defer wg.Done()
            gcTaskArgs := ShardOperationArgs{configNum, shardIDs}
            for _, server := range servers {
                gcTaskReply := new(ShardOperationReply)
                srv := kv.make_end(server)
                //这里RpC是将旧组中的BePulling状态的数据删除
```

```
                if srv.Call("ShardKV.DeleteShardsData", &gcTaskArgs,
gcTaskReply) && gcTaskReply.Err == OK {
                    DPrintf("{Node %v}{Group %v} deletes shards %v in remote
group successfully when currentConfigNum is %v", kv.rf.GetId(), kv.gid,
shardIDs, configNum)
                    // 不是重复调用，这里是将新组中的Gc状态改为服务，
                    kv.Execute(NewDeleteShardsCommand(&gcTaskArgs),
new(CommandReply))
                }
            }
        }(kv.lastConfig.Groups[gid], kv.currentConfig.Num, shardIDs)
    }
    kv.mu.RUnlock()
    wg.Wait()
}
```

应用操作

应用KV操作

将四种kv操作应用到状态机，首先检查该分片是否可以分组，然后检查是否重复命令，最后检查通过，应用到状态机

```
// applyOperation 将给定的操作应用于KV状态机
func (kv *ShardKV) applyOperation(operation *CommandArgs) *CommandReply {

    reply := new(CommandReply)
    shardID := key2shard(operation.Key)

    // 检查服务器是否可以提供请求的分片
    if !kv.canServe(shardID) {
        DPrintf("错误分组")
        reply.Err = ErrWrongGroup
    } else {
        //检查操作是否重复（仅针对非get操作）
        if operation.Op != Get && kv.isDuplicateRequest(operation.ClientId,
operation.CommandId) {
            DPrintf("{Node %v}{Group %v} does not apply duplicated commandId %v
to stateMachine because maxAppliedCommandId is %v for clientId %v",
kv.rf.GetId(), kv.gid, operation.CommandId,
kv.lastOperations[operation.ClientId].MaxAppliedCommandId, operation.ClientId)
            lastReply := kv.lastOperations[operation.ClientId].LastReply
            reply.Value, reply.Err = lastReply.Value, lastReply.Err
        } else {
            //将操作应用到状态机
            reply = kv.applyLogToStateMachine(operation, shardID)
            //更新最后提交
            if operation.Op != Get {
                kv.lastOperations[operation.ClientId] = OperationContext{
                    operation.CommandId,
                    reply,
                }
            }
        }
    }
    return reply
}
```

设置状态机要拉取的分片id设置为Pulling，被拉取的分片Id设置为Bepulling

```
// updateShardStatus 根据下次配置更新分片状态。
func (kv *ShardKV) updateShardStatus(nextConfig *shardctrler.Config) {
    for shardID := 0; shardID < shardctrler.NShards; shardID++ {
        //检查shard的组是否从当前配置更改到下一个配置。
        //这个shard不负责这个gid，但是下一个配置中的gid负责这个shard，所以需要拉出这个
shard。
        if kv.currentConfig.Shards[shardID] != kv.gid &&
nextConfig.Shards[shardID] == kv.gid {
            //获取新组Id
            gid := kv.currentConfig.Shards[shardID]
            //如果group为0，则跳过该shard，因为这意味着该shard没有分配给任何组
            if gid != 0 {
                kv.stateMachine[shardID].Status = Pulling
            }
        }
        //检查shard的组是否从next更改为当前配置
        //这个shard由这个gid负责，但是下一个配置中的gid不负责这个shard，所以需要由其他组来拉

        if kv.currentConfig.Shards[shardID] == kv.gid &&
nextConfig.Shards[shardID] != kv.gid {
            //获取新组Id
            gid := nextConfig.Shards[shardID]
            //如果group为0，则跳过该shard，因为这意味着该shard没有分配给任何组
            if gid != 0 {
                kv.stateMachine[shardID].Status = BePulling
            }
        }
    }
}
```

将kv操作应用到对应分片中去

```
// applylogtostatemmachine 将操作日志应用到状态机
func (kv *ShardKV) applyLogToStateMachine(operation *CommandArgs, shardID int)
*CommandReply {
    reply := new(CommandReply)
    switch operation.Op {
    case Get:
        reply.Value, reply.Err = kv.stateMachine[shardID].Get(operation.Key)
    case Put:
        reply.Err = kv.stateMachine[shardID].Put(operation.Key, operation.Value)
    case Append:
        reply.Err = kv.stateMachine[shardID].Append(operation.Key,
operation.Value)
    }
    return reply
}
```

应用配置更新

```
// applyConfiguration应用一个新的配置到shard。
func (kv *ShardKV) applyConfiguration(nextConfig *shardctrler.Config)
*CommandReply {
    reply := new(CommandReply)
    //检查新配置是否是当前配置的下一个配置
    if nextConfig.Num == kv.currentConfig.Num+1 {
        DPrintf("{Node %v}{Group %v} updates currentConfig from %v to %v",
kv.rf.GetId(), kv.gid, kv.currentConfig, nextConfig)
        //基于新配置更新分片的状态
        kv.updateShardStatus(nextConfig)

        //保存最后的配置
        kv.lastConfig = kv.currentConfig
        kv.currentConfig = *nextConfig
        reply.Err = OK
    } else {
        DPrintf("{Node %v}{Group %v} discards outdated configuration %v when
currentConfig is %v", kv.rf.GetId(), kv.gid, nextConfig, kv.currentConfig)
        reply.Err = ErrOutDated
    }
    return reply
}
```

应用插入分片

```
// applyInsertShards应用分片数据插入
func (kv *ShardKV) applyInsertShards(shardsInfo *ShardOperationReply)
*CommandReply {
    reply := new(CommandReply)
    //检查配置号是否匹配当前配置号
    if shardsInfo.ConfigNum == kv.currentConfig.Num {
        DPrintf("{Node %v}{Group %v} accepts shards insertion %v when
currentConfig is %v", kv.rf.GetId(), kv.gid, shardsInfo, kv.currentConfig)
        for shardID, shardData := range shardsInfo.Shards {
            shard := kv.stateMachine[shardID]
            //只有当分片处于pull状态时才会pull
            if shard.Status == Pulling {
                for key, value := range shardData {
                    shard.Put(key, value)
                }
                //更新shard状态为Garbage Collecting
                shard.Status = GCing
                reply.Err = OK
            } else {
                DPrintf("{Node %v}{Group %v} encounters duplicated shards
insertion %v when currentConfig is %v", kv.rf.GetId(), kv.gid, shardsInfo,
kv.currentConfig)
                break
            }
        }
        for clientID, operationContext := range shardsInfo.LastOperations {
            if lastOperation, ok := kv.lastOperations[clientID]; !ok ||
lastOperation.MaxAppliedCommandId < operationContext.MaxAppliedCommandId {
                kv.lastOperations[clientID] = operationContext
```

```
            }
        }
    } else {
        DPrintf("{Node %v}{Group %v} discards outdated shards insertion %v when
currentConfig is %v", kv.rf.GetId(), kv.gid, shardsInfo, kv.currentConfig)
        reply.Err = ErrOutDated
    }
    return reply
}
```

应用删除分片

```
// applydeletesards 应用删除分片数据
func (kv *ShardKV) applyDeleteShards(shardsInfo *ShardOperationArgs)
*CommandReply {
    //检查配置号是否一致
    if shardsInfo.ConfigNum == kv.currentConfig.Num {
        DPrintf("{Node %v}{Group %v}'s shards status are %v before accepting
shards deletion %v when currentConfig is %v", kv.rf.GetId(), kv.gid,
kv.stateMachine, shardsInfo, kv.currentConfig)
        for _, shardID := range shardsInfo.ShardIDs {
            //删除指定分片
            shard := kv.stateMachine[shardID]
            if shard.Status == GCing { //如果该分片正在被垃圾收集，则更新状态为正在服务。
                shard.Status = Serving
            } else if shard.Status == BePulling { //如果shard正在被拉，重置shard为一
个新的shard
                kv.stateMachine[shardID] = NewShard()
            } else { //如果碎片不在预期状态，则退出。
                DPrintf("{Node %v}{Group %v} encounters duplicated shards
deletion %v when currentConfig is %v", kv.rf.GetId(), kv.gid, shardsInfo,
kv.currentConfig)
                break
            }
        }
        DPrintf("{Node %v}{Group %v}'s shards status are %v after accepting
shards deletion %v when currentConfig is %v", kv.rf.GetId(), kv.gid,
kv.stateMachine, shardsInfo, kv.currentConfig)
        return &CommandReply{Err: OK}
    }

    DPrintf("{Node %v}{Group %v} discards outdated shards deletion %v when
currentConfig is %v", kv.rf.GetId(), kv.gid, shardsInfo, kv.currentConfig)
    return &CommandReply{Err: OK}
}
```

空碎片

```

// applyEmptyShards 处理空碎片的情况。这是为了防止状态机回滚
func (kv *ShardKV) applyEmptyShards() *CommandReply {
    return &CommandReply{Err: OK}
}
```

获取分片数据

```
// GetShardsData 处理 从Leader服务器获取分片数据。
func (kv *ShardKV) GetShardsData(args *ShardOperationArgs, reply
*ShardOperationReply) {
    // 场景：配置变更（Configuration 命令）导致分片 0 从副本组 GID 1 重新分配到 GID 2：
    // GID 1 的分片 0 进入 BePulling 状态，准备发送数据。
    // GID 2 的分片 0 进入 Pulling 状态，需要通过 RPC 调用 GID 1 的 GetShardsData 获取
数据。
    // 目标：展示 GetShardsData 如何处理 GID 2 的请求，返回分片 0 的数据和上下文

    //只从leader处提取碎片
    if _, isLeader := kv.rf.GetState(); !isLeader {
        reply.Err = ErrWrongLeader
        return
    }

    kv.mu.RLock()
    defer kv.mu.RUnlock()
    defer DPrintf("{Node %v}{Group %v} processes PullTaskRequest %v with
PullTaskReply %v", kv.rf.GetId(), kv.gid, args, reply)

    // 检查当前配置是否与请求中的配置一致
    if kv.currentConfig.Num < args.ConfigNum {
        reply.Err = ErrNotReady
        return
    }

    //检查当前配置是否为所请求的操作做好了准备
    reply.Shards = make(map[int]map[string]string)
    for _, shardId := range args.ShardIDs {
        //将分片种的kv拿出来，存入reply中的shards
        reply.Shards[shardId] = kv.stateMachine[shardId].deepCopy()
    }

    reply.LastOperations = make(map[int64]OperationContext)
    for clientId, operationContext := range kv.lastOperations {
        reply.LastOperations[clientId] = operationContext.deepCopy()
    }

    // 设置配置号并返回OK
    reply.ConfigNum, reply.Err = args.ConfigNum, OK
}
```

删除分片数据

```
/ 用于处理从 Leader 节点获取分片数据的请求，通常在分片迁移过程中由新组（Pulling 状态）向旧组
（BePulling 状态）发起
func (kv *ShardKV) DeleteShardsData(args *ShardOperationArgs, reply
*ShardOperationReply) {
    //只有是leader时才会删除
    if _, isLeader := kv.rf.GetState(); !isLeader {
        reply.Err = ErrWrongLeader
        return
    }
```

```
    defer DPrintf("{Node %v}{Group %v} processes GCTaskRequest %v with
GCTaskReply %v", kv.rf.GetId(), kv.gid, args, reply)
    kv.mu.RLock()

    //检查当前配置是否大于请求的配置
    if kv.currentConfig.Num > args.ConfigNum {
        DPrintf("{Node %v}{Group %v} encounters duplicated shards deletion
request %v when currentConfig is %v", kv.rf.GetId(), kv.gid, args,
kv.currentConfig)
        reply.Err = ErrOutDated
        kv.mu.RUnlock()
        return
    }
    if kv.currentConfig.Num < args.ConfigNum {
        DPrintf("{Node %v}{Group %v} encounters notready shards deletion request
%v when currentConfig is %v", kv.rf.GetId(), kv.gid, args, kv.currentConfig)
        reply.Err = ErrNotReady
        kv.mu.RUnlock()
        return
    }
    kv.mu.RUnlock()

    //DeleteShardsData
    commandReply := new(CommandReply)
    kv.Execute(NewDeleteShardsCommand(args), commandReply)
    reply.Err = commandReply.Err
}
```

结果

```
cd src/shardkv
```

```
go test
Test (5A): static shards ...
  ... Passed
Test (5A): rejection ...
  ... Passed
Test (5B): join then leave ...
  ... Passed
Test (5B): snapshots, join, and leave ...
  ... Passed
Test (5B): servers miss configuration changes...
  ... Passed
Test (5B): concurrent puts and configuration changes...
  ... Passed
Test (5B): more concurrent puts and configuration changes...
  ... Passed
Test (5B): concurrent configuration change and restart...
  ... Passed
Test (5B): unreliable 1...
  ... Passed
Test (5B): unreliable 2...
  ... Passed
Test (5B): unreliable 3...
  ... Passed
```

```
Test: shard deletion (challenge 1) ...
  ... Passed
Test: unaffected shard access (challenge 2) ...
  ... Passed
Test: partial migration shard access (challenge 2) ...
  ... Passed
PASS
ok      6.5840/shardkv  131.602s
```