目录

1 4A 和 4B 测试结果 **3**

2 Part A: The Shard controller **4**

2.1 准备工作 4

2.2 Lab4-4A 的核心组件和调用逻辑 4

2.3 common.go 编写 7

2.3.1 分片系统相关配置 7

2.3.2 定义执行超时的时间 8

2.3.3 定义调试所需要用到的函数 8

2.3.4 请求 RPC 调用的结构 8

2.3.5 回应 RPC 结构 9

2.3.6 Client 中 4 个操作的定义 10

2.4 Client 部分编写 10

2.4.1 client.go 基本架构 10

2.4.2 clerk 结构的定义和实例化函数 MakeClerk 11

2.4.3 随机数算法 nrand 12

2.4.4 客户端可以进行的 4 个操作接口 12

2.4.5 Client 的核心：Command 函数编写 13

2.5 Server 部分编写 14

2.5.1 server.go 部分基本架构和函数间的调用逻辑 14

2.5.2 分片控制器结构定义 15

2.5.3 实例化 ShardCtrler 结构的 StartServer 函数 16

2.5.4 编写 applier 函数 17

2.5.5 编写 applier 中所需要调用的三个方法 19

2.5.6 编写处理来自客户端请求的 Command 函数 20

2.6 configStateMachine 状态机结构部分 22

2.6.1 状态机解读 22

2.6.2 定义状态机结构 23

2.6.3 定义 MemoryConfigStateMachine 23

2.6.4 Join 方法实现 24

2.6.5 Leave 方法实现 25

2.6.6 Move 方法实现 27

2.6.7 Query 方法实现 28

2.6.8 Group2Shards 方法实现 29

2.6.9 GetGIDWithMinimumShards 方法实现 30

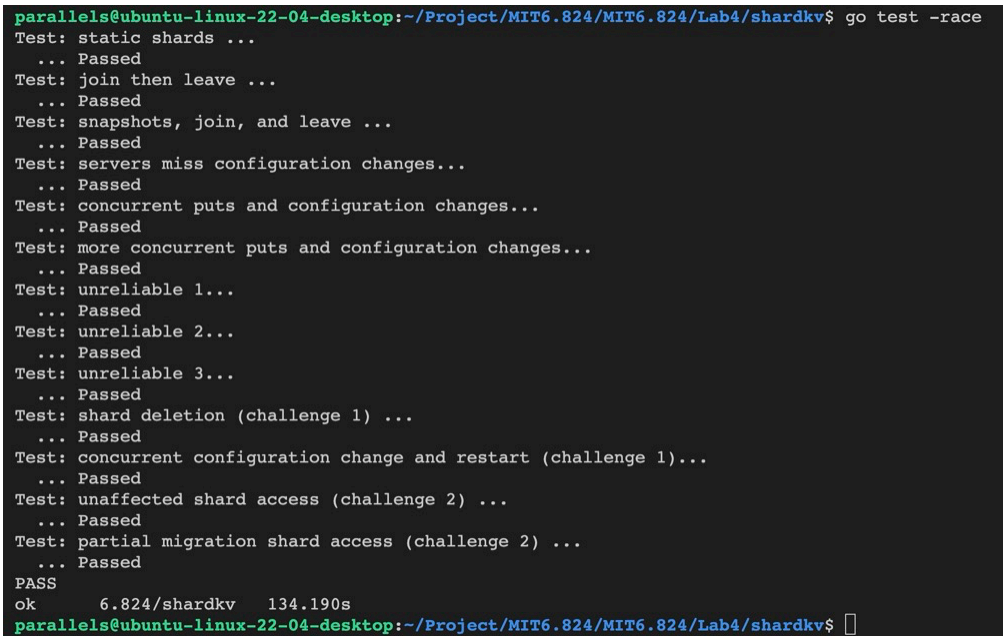
2.6.10 deepCopy 方法的实现 31

2.7 Lab4-4A 部分测试结果 31

3 Part B: Sharded Key/Value Server **31**
3.1 准备工作 . . . 31
3.2 Lab4-4B 的核心组件和调用逻辑 . . . 32
3.3 common.go 编写 . . . 33
3.3.1 common.go 所要编写的组件整理 . . . 33
3.3.2 一些常量的定义 . . . 34
3.3.3 定义分片的状态 . . . 35
3.3.4 OperationContext 部分编写 . . . 36
3.3.5 定义命令的类型 . . . 36
3.3.6 用 Command 结构对命令类型进行封装 . . . 37
3.3.7 3.3.7 定义核心 rpc 结构 . . . 38
3.3.8 定义分片核心 rpc 结构 . . . 38
3.3.9 分片部分的一些代码定义和功能的实现 . . . 39
3.4 编写 Client 部分代码 . . . 39
3.4.1 client 部分解读 . . . 39
3.4.2 先编写两个工具函数 . . . 40
3.4.3 Client(Clerk) 结构体编写和对应实例化的函数 . . . 40
3.4.4 定义操作命令的三个操作对应的接口 . . . 41
3.4.5 定义发送网络调用的 Command 接口 . . . 42
3.5 编写 Server 部分代码 . . . 43
3.5.1 Server 部分解读 . . . 43
3.5.2 ShardKV 结构定义 . . . 45
3.5.3 处理来自客户端的请求的 Command 函数 . . . 46
3.5.4 Execute 函数编写 . . . 47
3.5.5 GetShardsData 和 DeleteShardsData . . . 49
3.5.6 isDuplicateRequest 编写 . . . 50
3.5.7 canServe 编写 . . . 51
3.5.8 server 核心结构 applier 的详解和编写 . . . 51
3.5.9 applier 里面需要调用的对应分片操作的函数 . . . 54
3.5.10 快照相关方法编写 . . . 57
3.5.11 一次性初始化所有分片的状态机函数 . . . 59
3.5.12 处理配置更新的 configureAction 函数编写 . . . 59
3.5.13 从其他组拉取分片数据 migrationAction 的编写 . . . 60
3.5.14 删除不需要的分片数据接口 gcAction 编写 . . . 62
3.5.15 StartServer 实例化一个 KV 机器 . . . 63
3.6 4B 部分测试 . . . 65

4 实验过程中遇到的困难和对应的解决办法 **66**

5 实验总结 **67**



```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab4/shardkv$ go test -race
Test: static shards ...
  ... Passed
Test: join then leave ...
  ... Passed
Test: snapshots, join, and leave ...
  ... Passed
Test: servers miss configuration changes...
  ... Passed
Test: concurrent puts and configuration changes...
  ... Passed
Test: more concurrent puts and configuration changes...
  ... Passed
Test: unreliable 1...
  ... Passed
Test: unreliable 2...
  ... Passed
Test: unreliable 3...
  ... Passed
Test: shard deletion (challenge 1) ...
  ... Passed
Test: concurrent configuration change and restart (challenge 1)...
  ... Passed
Test: unaffected shard access (challenge 2) ...
  ... Passed
Test: partial migration shard access (challenge 2) ...
  ... Passed
PASS
ok      6.824/shardkv   134.190s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab4/shardkv$
```

Fig 2. 4B 测试结果

2 Part A: The Shard controller

2.1 准备工作

在开始之前，我先通过阅读 MIT 6.824 的官方手册整理出了整个 Lab4-4A 部分的基本解读和原理。如下所示。

MIT 6.824 Lab 4 的 Part A 部分主要是关于实现一个分片控制器（Shard Controller）。

在分布式系统中，特别是在键值存储系统中，分片（Sharding）是一种重要的技术，用于将数据分散存储在多个节点上，从而提高系统的可扩展性和处理能力。分片控制器（Shard Controller）是负责管理这些分片的配置和分配的组件。

在这个实验的 4A 部分，你需要实现的分片控制器主要完成以下功能：

1. **管理分片配置**：分片控制器需要跟踪不同的分片配置，这些配置描述了哪些副本组负责哪些分片。分片配置可以随时间变化，以响应系统的变化需求。

2. **处理 RPC 请求**：分片控制器需要能够响应客户端的 RPC 请求，这些请求可能包括加入新的副本组（Join），删除现有的副本组（Leave），手动移动分片到特定的副本组（Move），以及查询当前或特定的配置信息（Query）。

3. **保持一致性和容错性**：通过使用 Raft 算法，分片控制器要保持其状态的一致性，即使在面临节点故障或网络分区的情况下也能正常工作。

2.2 Lab4-4A 的核心组件和调用逻辑

核心组件如下所示。

1. 分片控制器（ShardCtrler）：

(1) 负责管理键值存储系统中分片的配置。

(2) 通过 Raft 协议实现容错性和一致性。

2. 客户端（Clerk）：

用于与分片控制器交互，发送请求如加入（Join）、离开（Leave）、移动（Move）分片，以及查询（Query）配置。

3. 配置状态机（ConfigStateMachine）：

(1) 在内存中维护配置状态的历史记录。

(2) 处理分片控制器收到的 Join, Leave, Move, 和 Query 操作。

4. Raft 实例:

用于在分片控制器节点间同步状态，确保容错性和一致性。

5. RPC（远程过程调用）：

分片控制器与客户端之间的通信机制。

调用逻辑如下所示。

1. 客户端发起请求:

(1) 客户端（Clerk）发起 Join, Leave, Move, 或 Query 请求。

(2) 客户端持续轮询，直到找到分片控制器的领导者，并发送 RPC 请求。

2. 分片控制器接收和处理请求:

(1) 分片控制器（ShardCtrler）接收到 RPC 请求。

(2) 如果是非 Query 操作，分片控制器首先检查请求是否重复。

(3) 通过 Raft 协议将操作日志复制到其他节点，确保状态一致。

3. 配置状态机更新状态:

(1) 分片控制器中的配置状态机（ConfigStateMachine）根据接收到的操作更新内部状态。

(2) Join 操作添加新的副本组，Leave 操作移除副本组，Move 操作改变特定分片的副本组，Query 操作返回特定配置。

4. 响应客户端请求:

完成状态更新后，分片控制器响应客户端的 RPC 请求，返回操作结果。

5. 状态机历史记录更新:

每个操作都会产生一个新的配置版本，这些版本被添加到状态机的历史记录中。

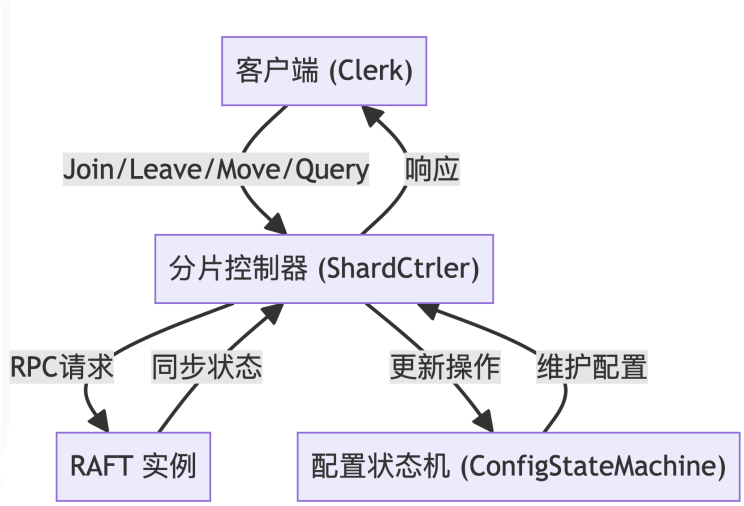


Fig 3. 4A 整体架构

理解整个 4A 部分的设计架构之后，我们需要通过阅读项目已经提供的源代码，来理清楚所有函数之间的调用逻辑。

1. **客户端（Clerk）**通过 Join、Leave、Move 或 Query 方法向 **分片控制器（ShardCtrler）**发起 RPC 请求。

2. **分片控制器**的 Command 方法接收这些请求，并通过 Start 方法将命令传递给 **Raft 实例**进行日志复制。

3. 当 **Raft 实例**确认日志条目已被复制到多数节点时，它通过 applych 通道将日志应用信息发送回**分片控制器**。4. 分片控制器中的 applier goroutine 从 applych 接收日志应用消息，并调用 applyLogToStateMachine 方法来更新 **配置状态机（ConfigStateMachine）**的状态。5. **配置状态机**通过实现的 Join、Leave、Move 和 Query 方法来更新其内部状态并响应分片控制器的请求。

可以整理成流程图，如 Fig 4所示。

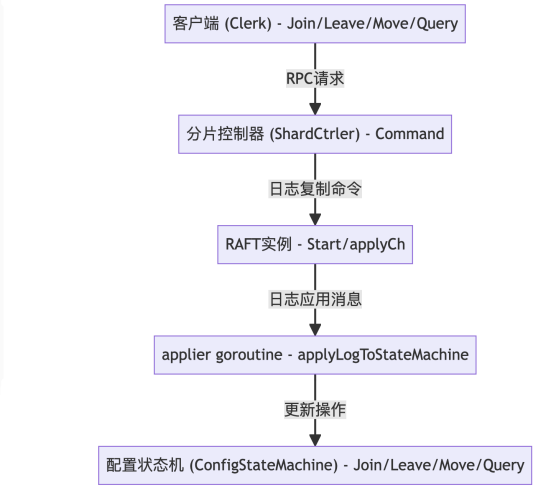


Fig 4. 4A 整体流程图

2.3 common.go 编写

这段代码主要定义了分片控制器的数据结构和 RPC 接口所需的请求和响应类型，以及一些辅助函数。它是分片控制器的核心部分，用于管理和响应客户端的请求。所需要定义的内容如下：

- 整个分片系统相关的配置, 即 Config 相关
- 调试所需要用到的一些接口
- 请求 RPC 相关定义, 即 CommandRequest 相关
- 回应 RPC 相关定义, 即 CommandResponse 相关
- Client 中 4 个操作的定义, 即 OperationOp 相关定义

common.go 中需要完成的组件，如图所示。

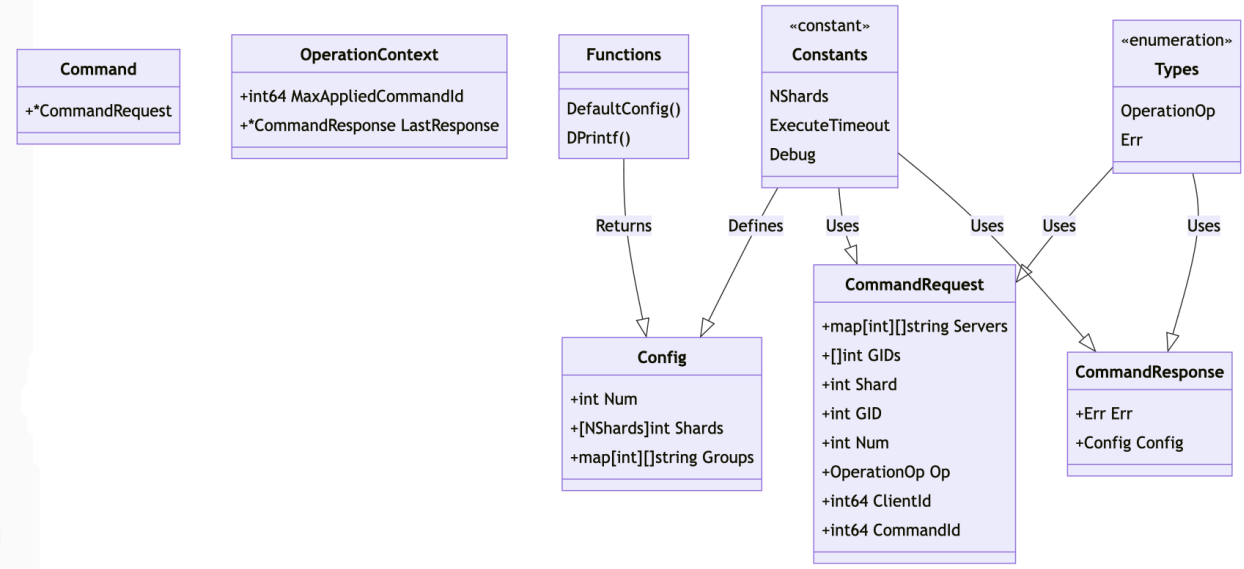


Fig 5. common.go 的逻辑

2.3.1 分片系统相关配置

这一部分代码定义了分片系统的分片数量，和描述配置的 Config 结构体，同时，这部分还定义了配置的生成函数和字符串的表示函数。

```go
// Config 描述了一组副本组以及每个 shard 的负责副本组。配置有编号。配置 #0 ...
    是初始配置，没有组，所有 shards 分配给组 0（无效组）。
// 请不要更改此结构。
const NShards = 10  // 分片的数量。

type Config struct {
    Num    int               // 配置编号
    Shards [NShards]int      // shard -> gid
    Groups map[int][]string // gid -> 服务器列表
```

```go
}
// 默认配置生成函数。
func DefaultConfig() Config {
    return Config{Groups: make(map[int][]string)}
}
// Config 结构的字符串表示形式。
func (cf Config) String() string {
    return fmt.Sprintf("{Num:%v,Shards:%v,Groups:%v}", cf.Num, cf.Shards, ...
        cf.Groups)
}
```

2.3.2 定义执行超时的时间

```go
const ExecuteTimeout = 500 * time.Millisecond  // 执行超时时间。
```

2.3.3 定义调试所需要用到的函数

这里定义调试所需要的一些日志相关函数。

```go
const Debug = false  // 调试标志。

// 调试打印函数。
func DPrintf(format string, a ...interface{}) (n int, err error) {
    if Debug {
        log.Printf(format, a...)
    }
    return
}
```

2.3.4 请求 RPC 调用的结构

客户端向服务端发起请求所要用到的 rpc 结构就是 CommandRequest 结构体，这个结构体会被 Client 里的四个操作所创建，并通过网络调用发送给服务端，具体定义如下所示。

```go
// RPC 请求的结构体。
type CommandRequest struct {
    Servers   map[int][]string // Join 请求的服务器映射。
    GIDs      []int            // Leave 请求的 GID 列表。
    Shard     int              // Move 请求的分片。
    GID       int              // Move 请求的组ID。
    Num       int              // Query 请求的配置编号。
    Op        OperationOp      // 操作类型。
    ClientId  int64            // 客户端ID。
```

```go
    CommandId int64             // 命令ID。
}
```

其中 Servers 是客户端中 Join 请求特有的，表示请求的服务器的映射，是一个 map 类型。GIDs 是 Leave 请求特有的，表示 GID 列表。Shard 是 Move 请求特有的，表示请求的分片，GID 表示 Move 请求的组 ID。Num 表示 Query 请求的配置编号。

CommandRequest 对应的字符串表示代码如下所示。

```go
// CommandRequest 的字符串表示。
func (request CommandRequest) String() string {
    switch request.Op {
    case OpJoin:
        return fmt.Sprintf("{Servers:%v,Op:%v,ClientId:%v,CommandId:%v}", ...
            request.Servers, request.Op, request.ClientId, request.CommandId)
    case OpLeave:
        return fmt.Sprintf("{GIDs:%v,Op:%v,ClientId:%v,CommandId:%v}", ...
            request.GIDs, request.Op, request.ClientId, request.CommandId)
    case OpMove:
        return fmt.Sprintf("{Shard:%v,Num:%v,Op:%v,ClientId:%v,CommandId:%v}", ...
            request.Shard, request.Num, request.Op, request.ClientId, ...
            request.CommandId)
    case OpQuery:
        return fmt.Sprintf("{Num:%v,Op:%v,ClientId:%v,CommandId:%v}", ...
            request.Num, request.Op, request.ClientId, request.CommandId)
    }
    panic(fmt.Sprintf("unexpected CommandOp %d", request.Op))
}
```

为了调用方便，对 CommandRequest 进行进一步封装。

```go
// Command 结构封装了 CommandRequest。
type Command struct {
    *CommandRequest
}
```

2.3.5 回应 RPC 结构

调用非常简单，只需要表示响应的配置即可。

```go
// RPC 响应的结构体。
type CommandResponse struct {
    Err    Err    // 响应的错误类型。
    Config Config // 响应的配置。
}

```

```go
// CommandResponse 的字符串表示。
func (response CommandResponse) String() string {
    return fmt.Sprintf("{Err:%v,Config:%v}", response.Err, response.Config)
}
```

2.3.6 Client 中 4 个操作的定义

Client 中 4 个操作的定义采用常量枚举的方式进行定义，实现思路很简单，如下代码所示。

```go
// 操作类型枚举。
type OperationOp uint8

const (
    OpJoin OperationOp = iota  // 加入操作
    OpLeave                    // 离开操作
    OpMove                     // 移动操作
    OpQuery                    // 查询操作
)

// 操作类型的字符串表示。
func (op OperationOp) String() string {
    switch op {
    case OpJoin:
        return "OpJoin"
    case OpLeave:
        return "OpLeave"
    case OpMove:
        return "OpMove"
    case OpQuery:
        return "OpQuery"
    }
    panic(fmt.Sprintf("unexpected CommandOp %d", op))
}
```

2.4 Client 部分编写

2.4.1 client.go 基本架构

client.go 的部分编写非常简单，我们只需要维护一个 client 结构体，使用 Makeclient 函数来实例化一个 client 对象，然后编写 client 的 4 个操作（对应的方法）即可。具体流程架构如 Fig 6所示。

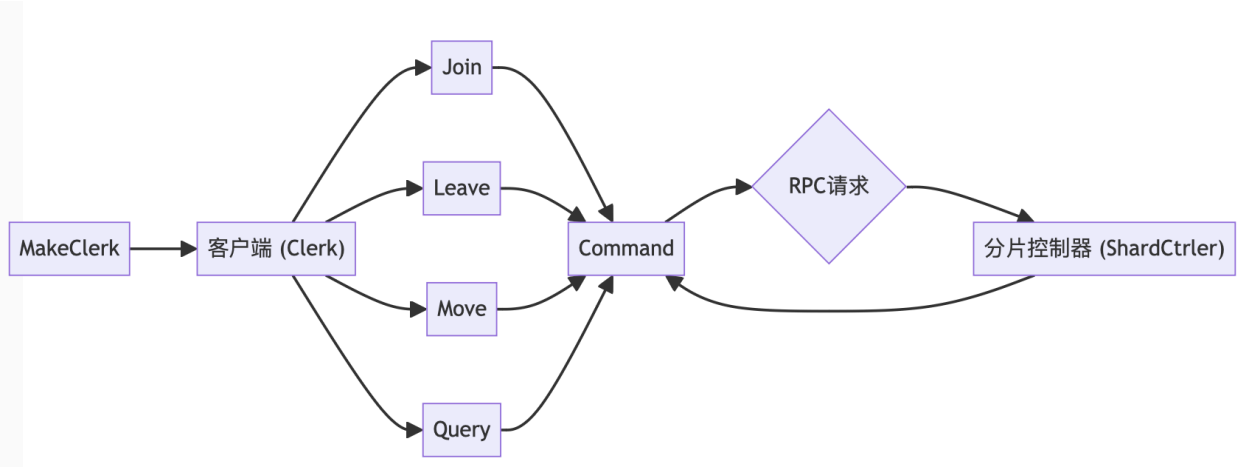


Fig 6. client.go 基本架构

2.4.2 clerk 结构的定义和实例化函数 MakeClerk

所需要的字段如下：

1. servers: 分片控制器的 rpc 服务器的列表, 也就是当前集群机子的列表
2. leaderld: 当前谁是领导者, 领导者的 id
3. clientld: 标识自己的 id
4. commandld: 标识命令的 id, (clientld, commandld) 元组可以标识定义一个唯一的操作。

代码如下所示。

```go
// Clerk 是客户端（Clerk）的结构体，用于与分片控制器进行交互。
type Clerk struct {
    servers   []*labrpc.ClientEnd // 分片控制器的RPC服务器列表。
    leaderId  int64               // 当前认为是领导者的服务器ID。
    clientId  int64               // ...
        客户端ID，用于唯一标识客户端。通常通过随机生成。
    commandId int64               // 命令ID，与clientId一起唯一定义一个操作。
}
```

MakeClerk 函数的作用就是创建一个 Clerk 对象，然后进行初始化即可。

```go
// MakeClerk 创建并返回一个新的Clerk实例。
// 这个函数接收分片控制器的RPC服务器列表并初始化一个Clerk结构体。
func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
    return &Clerk{
        servers:   servers,   // 设置RPC服务器列表。
        leaderId:  0,         // 初始时，将第一个服务器作为领导者。
        clientId:  nrand(),   // 生成一个随机的客户端ID。
        commandId: 0,         // 初始化命令ID为0。
    }
}
```

其中 clientId 需要一个唯一的随机数，用 nrand() 函数进行生成。

2.4.3 随机数算法 nrand

该随机数算法和之前 Lab3 的算法是相同的，代码如下所示。

```go
// nrand 生成一个随机的int64值，用作客户端ID。
// 这个函数使用加密安全的随机数生成器来确保ID的唯一性和随机性。
func nrand() int64 {
    max := big.NewInt(int64(1) << 62) // 设置生成数字的最大值。
    bigx, _ := rand.Int(rand.Reader, max) // 生成一个随机数。
    return bigx.Int64() // 返回生成的随机数。
}
```

2.4.4 客户端可以进行的 4 个操作接口

四个操作接口分别为如下：

- Query 请求分片控制器返回特定编号的配置
- Join 通知分片控制器有新的副本组加入
- Leave 通知分片控制器指定的副本组离开
- Move 允许手动指定一个分片分配到特定的副本组

四个接口都是需要调用 Command 函数进行网络调用的发送的，因为这四个接口都只是一个封装而已。具体实现如下所示。

```go
// Query 请求分片控制器返回特定编号的配置。
// 如果num为-1或大于已知的最大配置号，分片控制器应回复最新配置。
func (ck *Clerk) Query(num int) Config {
    return ck.Command(&CommandRequest{Num: num, Op: OpQuery})
}

// Join 通知分片控制器有新的副本组加入。
// servers参数是一个映射，键是副本组的GID，值是服务器名称列表。
// 加入的副本组将被分配一部分分片。
func (ck *Clerk) Join(servers map[int][]string) {
    ck.Command(&CommandRequest{Servers: servers, Op: OpJoin})
}

// Leave 通知分片控制器指定的副本组离开。
// gids是一个包含需要离开的副本组GID的列表。
// 离开的副本组将释放其分片，这些分片需要被重新分配给其它副本组。
func (ck *Clerk) Leave(gids []int) {
```

```go
    ck.Command(&CommandRequest{GIDs: gids, Op: OpLeave})
}

// Move 允许手动指定一个分片分配到特定的副本组。
// shard是分片号，gid是副本组的GID。
// 这个操作主要用于测试，实际环境中不常用。
func (ck *Clerk) Move(shard int, gid int) {
    ck.Command(&CommandRequest{Shard: shard, GID: gid, Op: OpMove})
}
```

2.4.5 Client 的核心：Command 函数编写

Command 函数是客户端部分重要的组件，负责向服务端发送网络请求（发起 rpc 调用）。具体流程如下所示。

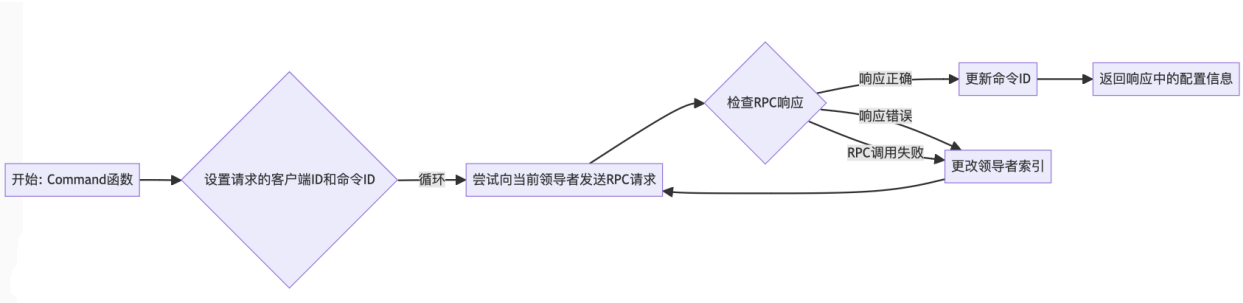


Fig 7. Command 函数逻辑

代码如下所示。

```go
func (ck *Clerk) Command(request *CommandRequest) Config {
    // 设置客户端ID和命令ID，用于识别请求和处理重复请求。
    request.ClientId, request.CommandId = ck.clientId, ck.commandId

    // 循环尝试，直到成功发送RPC请求并获得响应。
    for {
        var response CommandResponse
        // 尝试向当前认为是领导者的服务器发送RPC请求。
        // 如果RPC调用失败，或者响应表明领导者错误或请求超时，则尝试下一个服务器。
        if !ck.servers[ck.leaderId].Call("ShardCtrler.Command", request, ...
            &response) || response.Err == ErrWrongLeader || response.Err == ...
            ErrTimeout {
            ck.leaderId = (ck.leaderId + 1) % int64(len(ck.servers))
            continue
        }
        // 命令ID自增，用于下一个命令。
        ck.commandId++
        // 返回响应中包含的配置。
```

```
        return response.Config
    }
}
```

2.5 Server 部分编写

2.5.1 server.go 部分基本架构和函数间的调用逻辑

server.go 在分片控制器(ShardCtrler)中定义了服务器的主要逻辑。它负责处理来自客户端的 RPC 请求，与 Raft 实例交互以保持一致性，并管理内部状态。

主要组件

1. **ShardCtrler:** 作为主要结构, 包含了必要的字段, 如 Raft 实例、配置状态机、客户端请求的上下文等。

2. **Raft:** 用于在不同的 ShardCtrler 实例之间保持状态的一致性。

具体细节的一些组件如图所示。

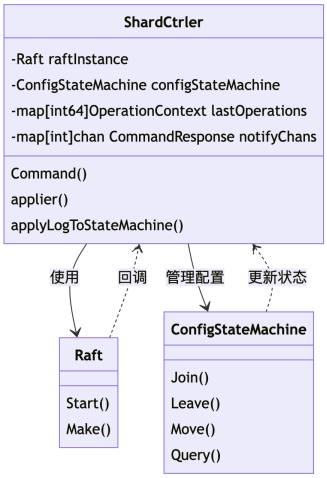


Fig 8. server.go 基本架构

函数调用逻辑:

1. **StartServer：** 初始化并启动 ShardCtrler 服务。

2. **Command：** 处理来自客户端的 RPC 请求。

3. **Raft.Start:** 由 Command 调用, 向 Raft 实例提交命令。

4. **applier:** 一个后台 goroutine, 监听 Raft 应用的日志消息。

5. **applyLogToStateMachine:** 由 applier 调用, 将 Raft 日志应用到配置状态机。

6. **配置状态机的方法**（如 Join, Leave, Move, Query）：由 applyLogToStateMachine 调用, 根据 Raft 日志更新配置状态。

调用逻辑流程如图所示。

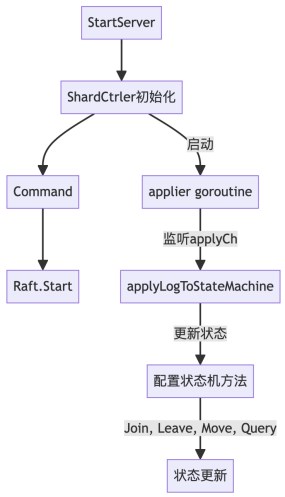


Fig 9. server.go 调用逻辑

有了这两幅图，其实整个 server 部分的编写逻辑已经非常清晰了。

2.5.2 分片控制器结构定义

分片控制器就是 ShardCtrler，现在先定义这个结构体，里面所需要的字段有：

- 用于同步访问的读写锁
- rf 结构的实例
- 和 Raft 进行通信的管道
- 状态机，即用来标识分片控制器状态的结构
- 记录客户端最后一个操作的 map，用于检测重复的请求
- 通知客户端 goroutine 以响应应用程序 goroutine 的映射 map

代码如下所示。

```go
// ShardCtrler 是分片控制器的结构体。
type ShardCtrler struct {
    mu      sync.RWMutex        // 用于同步访问的读写互斥锁。
    dead    int32               // 标记控制器是否已停止工作。
    rf      *raft.Raft          // Raft协onsensus算法的实例，用于保证一致性。
    applyCh chan raft.ApplyMsg // 从Raft层接收日志应用消息的通道。

    stateMachine   ConfigStateMachine            // 配置状态机，管理分片的配置。
    lastOperations map[int64]OperationContext    // ...
        记录每个客户端的最后一个操作，用于检测重复请求。
    notifyChans    map[int]chan *CommandResponse // ...
        通知客户端goroutine以响应应用程序goroutine。
}
```

2.5.3 实例化 ShardCtrler 结构的 StartServer 函数

StartServer 函数是 server.go 中的一个关键函数，它负责初始化并启动一个 ShardCtrler（分片控制器）实例。下面是函数的详细解释：

1. **注册 RPC 结构:**

调用 labgob.Register (Command{}) 注册 Command 结构体, 确保 Go 的 RPC 框架可以正确地序列化和反序列化该结构体。

2. **创建 applyCh 通道:**

使用 make(chan raft.ApplyMsg) 创建一个通道 applych, 用于接收来自 Raft 层的日志应用消息。

3. **初始化 ShardCtrler 结构体:**

- 创建一个新的 ShardCtrler 实例 sc。
- 设置 applych 用于接收应用日志消息。
- 初始化 dead 标志为 0，表示服务活跃。
- 调用 raft.Make(...) 创建 Raft 实例，并传入服务器数组、当前服务器索引、持久化存储器和 applych。
- 初始化 stateMachine 为一个新的配置状态机（NewMemoryConfigStateMachine()）。
- 初始化 lastoperations 和 notifychans，用于跟踪客户端的最后操作和响应通知。

4. **启动 applier goroutine :**

通过 go sc.applier() 启动一个 goroutine, 负责监听 applych 并将 Raft 应用的日志更新到状态机。

5. **打印调试信息:**

如果调试模式开启, 使用 DPrintf 打印 ShardCtrler 服务器启动的信息。

6. **返回 ShardCtrler 实例:**

函数返回初始化好的 ShardCtrler 实例。

代码如下所示。

```go
// StartServer 初始化并启动一个ShardCtrler服务器实例。
func StartServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister) *ShardCtrler {
    // 在Go的RPC库中注册Command结构体，
        // 以便于该结构体可以被RPC框架正确地序列化和反序列化。
    labgob.Register(Command{})
    // 创建一个用于接收来自Raft层的应用日志消息的通道。
    applyCh := make(chan raft.ApplyMsg)

    // 初始化ShardCtrler结构体实例。
    sc := &ShardCtrler{
        applyCh:        applyCh,                                    // 设置接收应用日志消息的通道。
        dead:           0,                                          // 初始化死亡标志为0（表示活跃）。
```

```go
        rf:              raft.Make(servers, me, persister, applyCh), // 创建Raft实例。
        stateMachine:    NewMemoryConfigStateMachine(),              // 创建新的配置状态机。
        lastOperations: make(map[int64]OperationContext),            // 初始化记录客户端操作的映射。
        notifyChans:     make(map[int]chan *CommandResponse),        // 初始化通知通道映射。
    }
    // 启动一个goroutine来应用提交的Raft日志到状态机。
    go sc.applier()

    // 打印调试信息，表明ShardCtrler服务器已启动。
    DPrintf("{ShardCtrler %v} has started", sc.rf.Me())
    // 返回初始化的ShardCtrler实例。
    return sc
}
```

2.5.4 编写 applier 函数

applier 函数的详细解读如下所示。

applier 函数是 server.go 中的一个关键函数, 用于将已提交的 Raft 日志应用到分片控制器的状态机。这是一个在后台运行的 goroutine, 用于处理来自 Raft 的日志消息。以下是其处理流程:

1. **无限循环:** applier 在 for 循环中运行, 直到 ShardCtrler 被标记为停止。
2. **监听 Raft 日志消息:** 从 applych 通道接收 Raft 层传递的日志应用消息。
3. **处理每条消息:**

- 对于每条接收到的消息，首先检查其有效性（message.Commandvalid）。
- 如果消息有效，则将其转换为 Command 类型并加锁以确保同步。

4. **检查重复请求:**

- 检查当前命令是否重复（即先前已处理过相同的客户端命令）。
- 如果是重复的，使用之前保存的响应而不是重新应用日志。

5. **应用日志到状态机:**

- 如果命令不是重复的，则调用 applyLogToStateMachine 方法来更新状态机。
- 更新客户端的最后操作记录。

6. **通知等待的 goroutine:**

- 如果当前节点是 Raft 集群的领导者，并且消息属于当前任期，则通过通知通道发送响应。
- 通过 getNotifyChan 方法获取与命令索引相关联的通知通道，并发送响应。

7. **异常处理:**

如果接收到意外的消息类型, 则触发 panic。

执行流程如图所示。

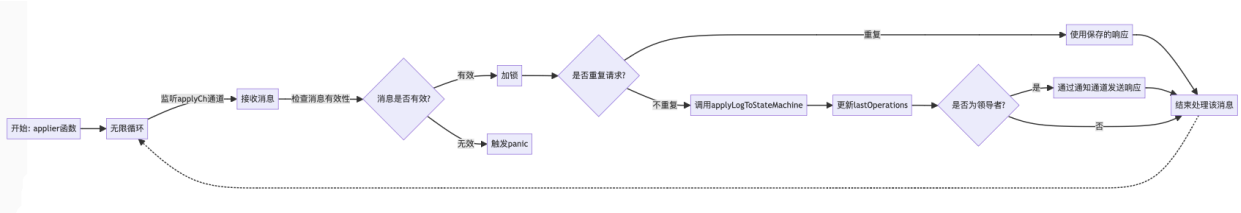


Fig 10. applier 基本架构

代码如下所示。

```go
// applier 是一个专门的goroutine，用于将已提交的Raft日志条目应用到状态机。
func (sc *ShardCtrler) applier() {
    // 循环运行，直到ShardCtrler被标记为停止。
    for sc.killed() == false {
        select {
        case message := <-sc.applyCh:
            // 打印调试信息。
            DPrintf("{Node %v} tries to apply message %v", sc.rf.Me(), message)
            // 确认日志条目是有效的。
            if message.CommandValid {
                var response *CommandResponse
                command := message.Command.(Command)
                sc.mu.Lock()

                // 检查命令是否重复。
                if command.Op != OpQuery && ...
                    sc.isDuplicateRequest(command.ClientId, command.CommandId) {
                    // 如果是重复的，使用保存的响应而不是重新应用日志。
                    DPrintf("{Node %v} doesn't apply duplicated message %v to ...
                        stateMachine because maxAppliedCommandId is %v for client ...
                        %v", sc.rf.Me(), message, ...
                        sc.lastOperations[command.ClientId], command.ClientId)
                    response = sc.lastOperations[command.ClientId].LastResponse
                } else {
                    // 应用日志到状态机，并保存响应。
                    response = sc.applyLogToStateMachine(command)
                    if command.Op != OpQuery {
                        sc.lastOperations[command.ClientId] = ...
                            OperationContext{command.CommandId, response}
                    }
                }

                // 仅当节点为领导者且日志条目属于当前任期时，通知相关的通道。
                if currentTerm, isLeader := sc.rf.GetState(); isLeader && ...
                    message.CommandTerm == currentTerm {
```

```go
                    ch := sc.getNotifyChan(message.CommandIndex)
                    ch <- response
                }

                sc.mu.Unlock()
            } else {
                // 如果接收到意外的消息类型，触发panic。
                panic(fmt.Sprintf("unexpected Message %v", message))
            }
        }
    }
}
```

2.5.5 编写 applier 中所需要调用的三个方法

在 applier 函数中，调用了 isDuplicateRequest、applyLogToStateMachine 和 getNotifyChan 这三个方法，每个都有特定的作用：

isDuplicateRequest:

- **功能:** 检查当前处理的命令是否是重复的命令。
- **用途:** 在分布式系统中，由于网络延迟或其他因素，同一个客户端的命令可能会被重复接收。这个函数通过检查命令的 ID 和客户端的 ID 来确定是否已经处理过相同的命令。
- **处理流程:** 如果当前命令的 ID 小于或等于已记录的最大命令 ID，那么它被认为是重复的，因此不需要再次应用到状态机。

代码如下：

```go
// 它根据日志条目的操作类型执行相应的状态机操作，并返回结果。
func (sc *ShardCtrler) applyLogToStateMachine(command Command) *CommandResponse {
    var config Config
    var err Err
    // 根据命令类型执行相应的状态机操作。
    switch command.Op {
    case OpJoin:
        err = sc.stateMachine.Join(command.Servers)
    case OpLeave:
        err = sc.stateMachine.Leave(command.GIDs)
    case OpMove:
        err = sc.stateMachine.Move(command.Shard, command.GID)
    case OpQuery:
        config, err = sc.stateMachine.Query(command.Num)
    }
    // 返回操作的结果。
    return &CommandResponse{err, config}
}
```

getNotifyChan:

- **功能:** 获取与给定 Raft 日志索引相关联的通知通道。
- **用途:** 在 Raft 确认日志条目被复制到多数节点后，applier 通过这个通道通知相应的等待响应的 goroutine。这是一种机制，用于在操作被应用到状态机后，通知那些等待命令结果的客户端。
- **处理流程:** 每个命令在被 Raft 处理时都有一个唯一的索引。getNotifyChan 根据这个索引返回一个通道，如果这个通道不存在，它会创建一个新的通道。然后，可以通过这个通道发送命令的处理结果。

```go
// getNotifyChan 根据提供的日志索引获取相应的通知通道。
// 如果该索引的通道不存在，则创建一个新的通道。
func (sc *ShardCtrler) getNotifyChan(index int) chan *CommandResponse {
    // 检查是否已经存在与该日志索引对应的通道。
    if _, ok := sc.notifyChans[index]; !ok {
        // 如果不存在，创建一个新的通道并将其添加到映射中。
        sc.notifyChans[index] = make(chan *CommandResponse, 1)
    }
    // 返回该索引对应的通道。
    return sc.notifyChans[index]
}
```

2.5.6 编写处理来自客户端请求的 Command 函数

Command 函数是分片控制器（ShardCtrler）中处理来自客户端的 RPC 请求的核心函数。它的主要职责是处理请求、与 Raft 层交互以保持状态的一致性，并返回响应给客户端。以下是其处理流程：

1. 检查重复请求:

- 使用读锁来检查请求是否重复。
- 如果是非查询 (OpQuery) 的重复请求，则直接返回上次的响应结果。

2. 提交命令到 Raft 层:

- 如果不是重复请求，将命令提交给 Raft 实例进行处理。
- 使用 Start 方法返回一个索引（用于追踪命令）和是否是领导者的标志。

3. 处理非领导者节点:

如果当前节点不是领导者, 则返回 ErrWrongLeader 错误。

4. 等待命令结果:

- 获取与 Raft 日志索引关联的通知通道。
- 等待直到从通道接收到命令结果或超时。

5. 处理命令结果:

- 如果在超时前收到结果，则将结果复制到响应中。

- 如果超时，则设置响应错误为 ErrTimeout。

6. 异步释放通知通道:
异步地移除过时的通知通道, 以防止内存泄漏。

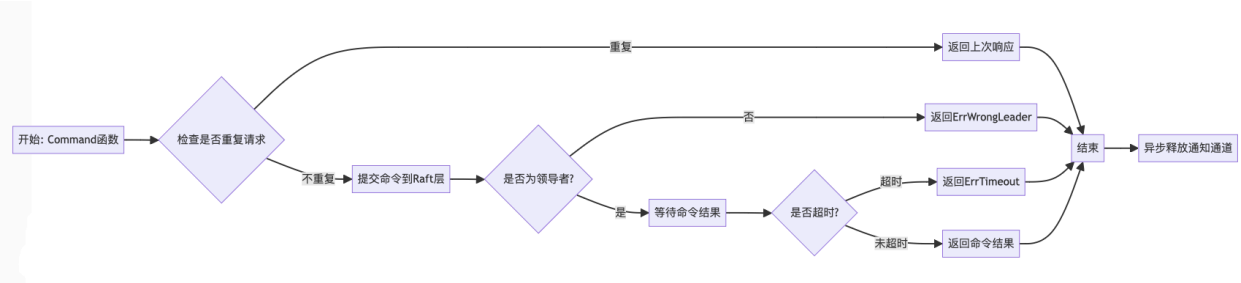


Fig 11. 客户端 Command 函数逻辑

代码如下所示。

```go
// Command 处理来自客户端的命令请求。
func (sc *ShardCtrler) Command(request *CommandRequest, response *CommandResponse) {
    defer DPrintf("{Node %v}'s state is {}, processes CommandRequest %v with CommandResponse %v", sc.rf.Me(), request, response)

    // 读锁定以查询此请求是否重复。
    sc.mu.RLock()
    // 检查请求是否重复。对于非查询操作（即除了OpQuery以外的所有操作），
        // 如果发现是重复请求，则直接从lastOperations映射中获取上一次
        // 的响应并返回，不需要再次经过Raft层处理。
    if request.Op != OpQuery && sc.isDuplicateRequest(request.ClientId, request.CommandId) {
        lastResponse := sc.lastOperations[request.ClientId].LastResponse
        response.Config, response.Err = lastResponse.Config, lastResponse.Err
        sc.mu.RUnlock() // 释放读锁。
        return
    }
    sc.mu.RUnlock() // 释放读锁。

    // 向Raft层提交命令。这里不持有锁是为了提高吞吐量，
        // 因为Raft的Start方法本身是并发安全的。
    index, _, isLeader := sc.rf.Start(Command{request})
    // 如果当前节点不是领导者，则立即返回错误。
    if !isLeader {
        response.Err = ErrWrongLeader
        return
    }

```

```go
    // 锁定以安全地访问notifyChans映射。
    sc.mu.Lock()
    // ...
        获取与Raft日志索引关联的通知通道。如果该通道不存在，则getNotifyChan会创建它。
    ch := sc.getNotifyChan(index)
    sc.mu.Unlock() // 解锁。

    // 等待命令结果或超时。
    select {
    case result := <-ch:
        // 从通道接收到结果后，将其复制到响应中。
        response.Config, response.Err = result.Config, result.Err
    case <-time.After(ExecuteTimeout):
        // 如果等待超过了预定的超时时间，则设置响应错误为超时。
        response.Err = ErrTimeout
    }

    // 异步释放notifyChan，以减少内存占用。此操作不需要阻塞客户端请求，
    // 因此通过一个新的goroutine来完成，以提高系统吞吐量。
    go func() {
        sc.mu.Lock()
        // 移除已过时的通知通道，这样可以防止内存泄漏。
        sc.removeOutdatedNotifyChan(index)
        sc.mu.Unlock()
    }()
}
```

2.6 configStateMachine 状态机结构部分

2.6.1 状态机解读

configStateMachine.go 中的代码定义了分片控制器（ShardCtrler）的配置状态机（ConfigStateMachine）。这个状态机的主要作用是维护和管理分片（Shard）的配置，包括如何将分片分配给不同的副本组（replication groups）。具体来说，它包含以下几个关键功能：

1. 配置存储: 维护分片配置的历史记录，每个配置包括对副本组的分片分配和副本组的服务器列表。

2. Join 方法: 处理加入新副本组的请求。当新的副本组加入时，状态机更新当前的分片配置，将新的副本组纳入系统，并重新平衡分片分配以确保分片在各个副本组之间均匀分布。

3. Leave 方法: 处理副本组的离开请求。当副本组离开时，状态机从当前配置中移除这些副本组，并重新分配它们负责的分片到剩余的副本组，以保持分片分配的均衡。

4. Move 方法: 处理手动移动特定分片到指定副本组的请求。这通常用于测试或特定的管理操作。

5. Query 方法: 查询特定编号的配置。如果请求的编号超出历史记录范围，状态机将返回最新的配置。

通过这些方法，configStateMachine.go 中的配置状态机为分片控制器提供了处理分片配置变化的核心逻辑。

这部分的整体架构如图所示。

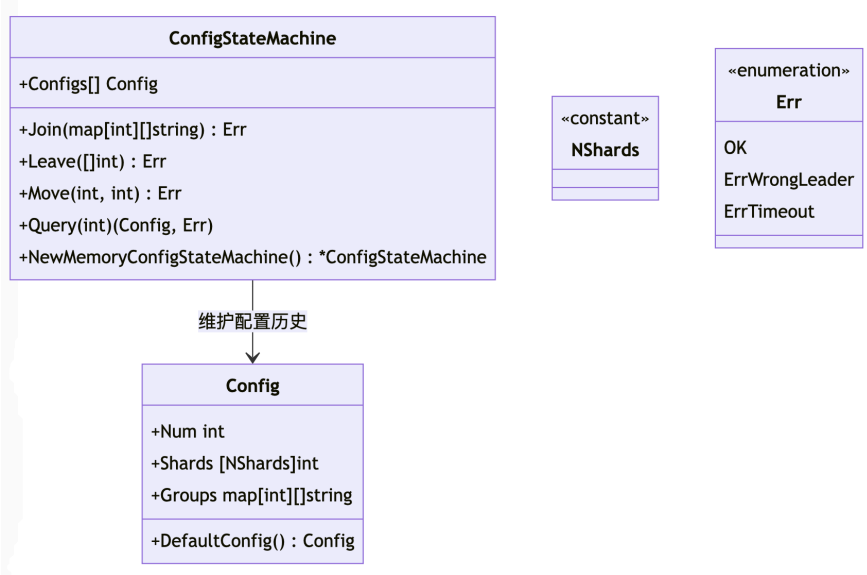


Fig 12. configStateMachine 基本架构

2.6.2 定义状态机结构

代码如下所示。

```
// ConfigStateMachine 是一个接口，定义了配置状态机必须实现的方法。
// 这些方法对应于分片控制器需要处理的不同类型的操作。
type ConfigStateMachine interface {
    Join(groups map[int][]string) Err // Join 方法用于处理加入新副本组的操作。
    Leave(gids []int) Err             // Leave 方法用于处理副本组离开的操作。
    Move(shard, gid int) Err          // Move ...
        方法用于将特定分片移动到指定副本组的操作。
    Query(num int) (Config, Err)      // Query 方法用于查询特定编号的配置。
}
```

2.6.3 定义 MemoryConfigStateMachine

MemoryConfigStateMachine 是 ConfigStateMachine 接口的一个实现。

代码实现如下所示。

```
// MemoryConfigStateMachine 是ConfigStateMachine接口的一个实现。
// 它在内存中维护配置状态的历史记录。
type MemoryConfigStateMachine struct {
```

```go
    Configs []Config // Configs 是一个配置数组，存储所有配置的历史记录。
}
// NewMemoryConfigStateMachine 创建并初始化一个MemoryConfigStateMachine实例。
// 它将配置数组初始化为包含一个默认配置。
func NewMemoryConfigStateMachine() *MemoryConfigStateMachine {
    // 创建MemoryConfigStateMachine实例，并预分配一个配置槽。
    cf := &MemoryConfigStateMachine{make([]Config, 1)}
    // 初始化第一个配置为默认配置。
    cf.Configs[0] = DefaultConfig()
    // 返回初始化的MemoryConfigStateMachine实例。
    return cf
}
```

2.6.4 Join 方法实现

join 过程处理流程如下。

1. 获取最后一个配置：作为更新的基础，获取当前配置状态机中的最后一个配置。

2. 创建新配置：基于最后一个配置创建一个新的 Config 结构体实例，包括分配新的配置编号和复制现有的副本组。

3. 处理加入的组:

- 遍历 groups 参数中的每个副本组。
- 如果新配置中还不存在该副本组，就将其添加到新配置中。

4. 重新平衡分片分配:

- 调用 Group2Shards 函数将新配置中的分片映射到副本组。
- 调用 GetGIDWithMaximumShards 和 GetGIDWithMinimumShards 函数找出分片最多和最少的副本组。
- 通过移动分片，重新平衡各副本组之间的分片分配。

5. 更新新配置的分片映射：更新 newConfig 中的 Shards 字段, 反映重新平衡后的分片分配。

6. 更新配置历史记录：将新配置添加到 Configs 列表中, 完成对新副本组的加入处理。

7. 返回成功：函数返回 OK, 表示成功处理了加入请求。

流程图如图所示。

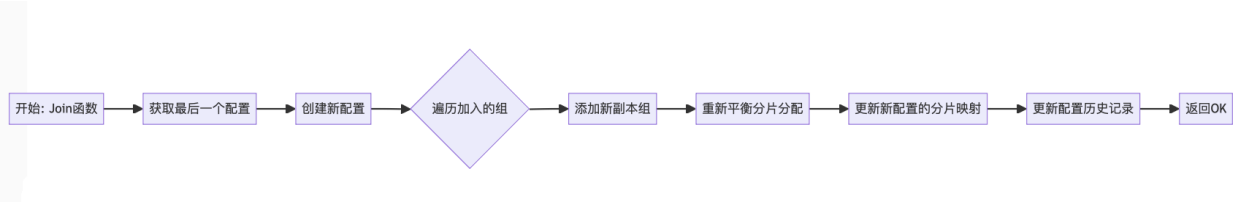


Fig 13. Join 方法基本逻辑

```go
// Join 方法处理加入新副本组的请求。
func (cf *MemoryConfigStateMachine) Join(groups map[int][]string) Err {
    // 获取最后一个配置作为基础进行更新。
    lastConfig := cf.Configs[len(cf.Configs)-1]
    // 创建新配置，初始化编号和副本组。
    newConfig := Config{len(cf.Configs), lastConfig.Shards, ...
        deepCopy(lastConfig.Groups)}
    // 遍历加入的组并添加到新配置中。
    for gid, servers := range groups {
        if _, ok := newConfig.Groups[gid]; !ok {
            newServers := make([]string, len(servers))
            copy(newServers, servers)
            newConfig.Groups[gid] = newServers
        }
    }
    // 重新平衡分片分配。
    s2g := Group2Shards(newConfig)
    for {
        source, target := GetGIDWithMaximumShards(s2g), ...
            GetGIDWithMinimumShards(s2g)
        if source != 0 && len(s2g[source])-len(s2g[target]) ≤ 1 {
            break
        }
        s2g[target] = append(s2g[target], s2g[source][0])
        s2g[source] = s2g[source][1:]
    }
    var newShards [NShards]int
    for gid, shards := range s2g {
        for _, shard := range shards {
            newShards[shard] = gid
        }
    }
    newConfig.Shards = newShards
    // 将新配置添加到历史记录中。
    cf.Configs = append(cf.Configs, newConfig)
    return OK
}
```

2.6.5 Leave 方法实现

Leave 函数是 MemoryConfigStateMachine 结构体的方法，处理副本组离开的请求。以下是其处理流程：

1. **获取最后一个配置：**作为更新的基础，获取当前配置状态机中的最后一个配置。
2. **创建新配置：**基于最后一个配置创建一个新的 Config 结构体实例。

3. 移除离开的组:

- 遍历 gids 参数中的每个副本组 ID。
- 如果副本组在当前配置中存在，则从新配置的 Groups 中删除该副本组。
- 同时收集由于副本组离开而变成孤立的分片。

4. 进行负载均衡:

- 如果仍有副本组存在，对孤立的分片进行重新分配。
- 通过 GetGIDWithMinimumShards 找到分片最少的副本组，并将孤立的分片分配给它们。

5. 更新新配置的分片映射: 更新 newConfig 中的 Shards 字段, 反映重新分配后的分片映射。

6. 更新配置历史记录: 将新配置添加到 Configs 列表中, 完成对离开副本组的处理。

7. 返回成功: 函数返回 OK, 表示成功处理了离开请求。

流程图如图所示。

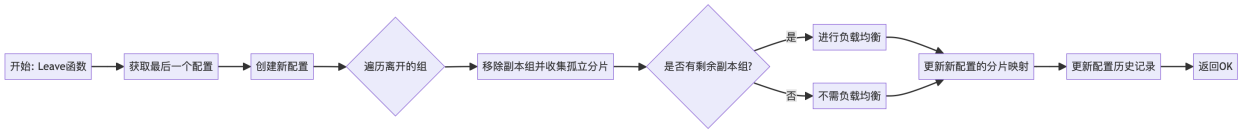


Fig 14. Leave 方法基本架构

```go
// Leave 方法处理副本组离开的请求。
func (cf *MemoryConfigStateMachine) Leave(gids []int) Err {
    // 获取最后一个配置作为更新基础。
    lastConfig := cf.Configs[len(cf.Configs)-1]
    newConfig := Config{len(cf.Configs), lastConfig.Shards, ...
        deepCopy(lastConfig.Groups)}
    s2g := Group2Shards(newConfig)
    orphanShards := make([]int, 0)
    // 移除离开的组并收集孤立分片。
    for _, gid := range gids {
        if _, ok := newConfig.Groups[gid]; ok {
            delete(newConfig.Groups, gid)
        }
        if shards, ok := s2g[gid]; ok {
            orphanShards = append(orphanShards, shards...)
            delete(s2g, gid)
        }
    }
    var newShards [NShards]int
    // 仅在存在剩余副本组时进行负载均衡。
    if len(newConfig.Groups) != 0 {
```

```go
        for _, shard := range orphanShards {
            target := GetGIDWithMinimumShards(s2g)
            s2g[target] = append(s2g[target], shard)
        }
        for gid, shards := range s2g {
            for _, shard := range shards {
                newShards[shard] = gid
            }
        }
    }
    newConfig.Shards = newShards
    // 将新配置添加到历史记录中。
    cf.Configs = append(cf.Configs, newConfig)
    return OK
}
```

2.6.6 Move 方法实现

Move 函数是 MemoryConfigStateMachine 结构体的一个方法, 用于处理将特定分片（shard）移动到指定的副本组 (replication group) 的请求。以下是其处理流程:

1. 获取最后一个配置: 作为更新的基础, 获取当前配置状态机中的最后一个配置。

2. 创建新配置: 基于最后一个配置创建一个新的 Config 结构体实例。这包括复制现有的分片映射和副本组信息。

3. 更新分片分配: 将指定的分片 shard 分配给指定的副本组 gid, 更新 newConfig 中的 Shards 数组。

4. 更新配置历史记录: 将新的配置添加到 Configs 列表中, 完成对移动分片请求的处理。

5. 返回成功: 函数返回 OK, 表示成功处理了移动分片的请求。

流程图如下所示。

Fig 15. Leave 方法基本架构

代码如下所示。

```go
// Move 方法处理将特定分片移动到指定副本组的请求。
func (cf *MemoryConfigStateMachine) Move(shard, gid int) Err {
    // 获取最后一个配置作为更新基础。
    lastConfig := cf.Configs[len(cf.Configs)-1]
```

```go
    newConfig := Config{len(cf.Configs), lastConfig.Shards, ...
        deepCopy(lastConfig.Groups)}
    // 更新指定分片的副本组分配。
    newConfig.Shards[shard] = gid
    // 将新配置添加到历史记录中。
    cf.Configs = append(cf.Configs, newConfig)
    return OK
}
```

2.6.7 Query 方法实现

Query 函数是 MemoryConfigStateMachine 结构体的一个方法, 用于查询指定编号的配置。以下是其处理流程:

1. **检查请求编号:** 判断请求的配置编号 num 是否在有效范围内。
2. **返回配置:**

- 如果请求的编号超出配置历史记录的范围（即 num 小于 0 或大于等于配置列表的长度），则返回最新的配置。
- 如果请求的编号在有效范围内，则直接返回对应编号的配置。

3. **返回成功:** 函数返回请求的配置以及 OK 状态, 表示查询成功。

流程图如下所示。

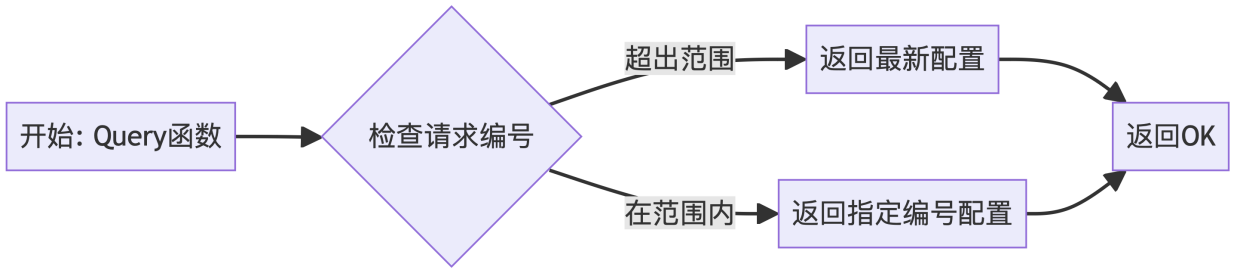


Fig 16. Query 方法基本架构

代码如下所示。

```go
// Query 方法查询指定编号的配置。
func (cf *MemoryConfigStateMachine) Query(num int) (Config, Err) {
    // 如果请求的编号超出历史记录范围，返回最新的配置。
    if num < 0 || num ≥ len(cf.Configs) {
        return cf.Configs[len(cf.Configs)-1], OK
    }
    // 返回指定编号的配置。
    return cf.Configs[num], OK
}
```

2.6.8 Group2Shards 方法实现

Group2Shards 函数用于将配置中的分片映射到各个副本组。它是一个帮助函数，通常用于配置状态机中分片重新平衡的过程。以下是其处理流程:

1. 初始化分片到副本组的映射: 创建一个新的映射 $s2g$ ，用于存储每个副本组及其对应的分片列表。

2. 为每个副本组初始化分片列表: 遍

- 遍历配置历史中的所有副本组 `config.Groups`。
- 对于每个副本组 `gid`，在 `s2g` 中初始化一个空的分片列表。

3. **遍历分片并映射到副本组:**

- 遍历配置中的所有分片 `config.Shards`。
- 对于每个分片 `shard` 及其对应的副本组 `gid`，将分片添加到 `$s2g[gid]$` 的列表中。

4. **返回分片到副本组的映射:** 函数返回映射 s2g。

流程图如下所示。

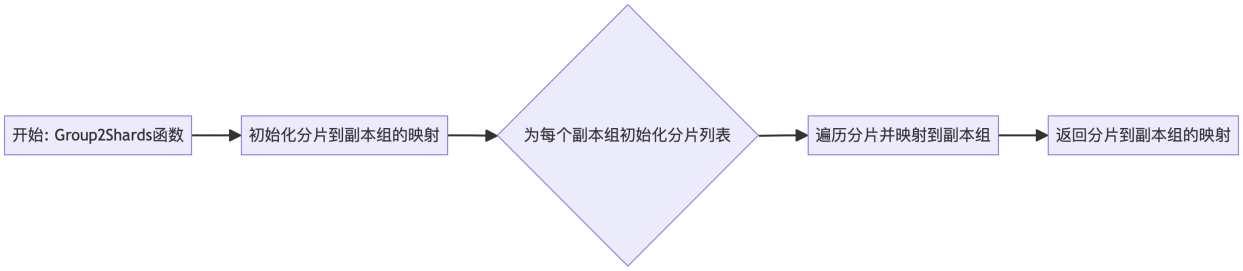


Fig 17. Group2Shards 方法基本架构

代码如下所示。

```go
// Group2Shards 将配置中的分片映射到各个副本组。
func Group2Shards(config Config) map[int][]int {
    s2g := make(map[int][]int)
    // 为每个副本组初始化一个空的分片列表。
    for gid := range config.Groups {
        s2g[gid] = make([]int, 0)
    }
    // 遍历分片，将每个分片映射到其对应的副本组。
    for shard, gid := range config.Shards {
        s2g[gid] = append(s2g[gid], shard)
    }
    return s2g
}
```

2.6.9 GetGIDWithMinimumShards 方法实现

GetGIDWithMinimumShards 函数用于在给定的分片到副本组的映射中查找拥有最少分片的副本组。以下是其处理流程: (找最多和最少是一样的, 这里不重复了)。

1. 排序副本组键:

- 创建一个 `keys` 数组来存储映射 s2g 中所有的副本组键 (GID)。
- 对 `keys` 进行排序，以确保迭代顺序是确定的。

2. 初始化查找变量: 初始化 index 为 -1 和 min 为 NShards +1 , 用于跟踪拥有最少分片的副本组。

3. 遍历并查找:

- 遍历排序后的副本组 keys。
- 对于每个副本组 gid，检查它拥有的分片数量 len(s2g[gid]) 是否小于当前记录的最少分片数 min。
- 如果是，则更新 index 和 min 为当前副本组的 GID 和它的分片数量。

4. 返回结果： 返回拥有最少分片的副本组的 GID。

Fig 18. GetGIDWithMinimumShards 方法基本架构

代码如下所示。

```go
// GetGIDWithMinimumShards 查找拥有最少分片的副本组。
func GetGIDWithMinimumShards(s2g map[int][]int) int {
    // 确保迭代顺序是确定的。
    var keys []int
    for k := range s2g {
        keys = append(keys, k)
    }
    sort.Ints(keys)

    // 查找拥有最少分片的副本组。
    index, min := -1, NShards+1
    for _, gid := range keys {
        if gid != 0 && len(s2g[gid]) < min {
            index, min = gid, len(s2g[gid])
        }
    }
    return index
}
```

2.6.10 deepCopy 方法的实现

作用就是对副本组映射进行深拷贝。

代码如下所示。

```go
// deepCopy 对副本组映射进行深拷贝。
func deepCopy(groups map[int][]string) map[int][]string {
    newGroups := make(map[int][]string)
    // 遍历副本组映射，并为每个组创建一个服务器列表的副本。
    for gid, servers := range groups {
        newServers := make([]string, len(servers))
        copy(newServers, servers)
        newGroups[gid] = newServers
    }
    return newGroups
}
```

2.7 Lab4-4A 部分测试结果

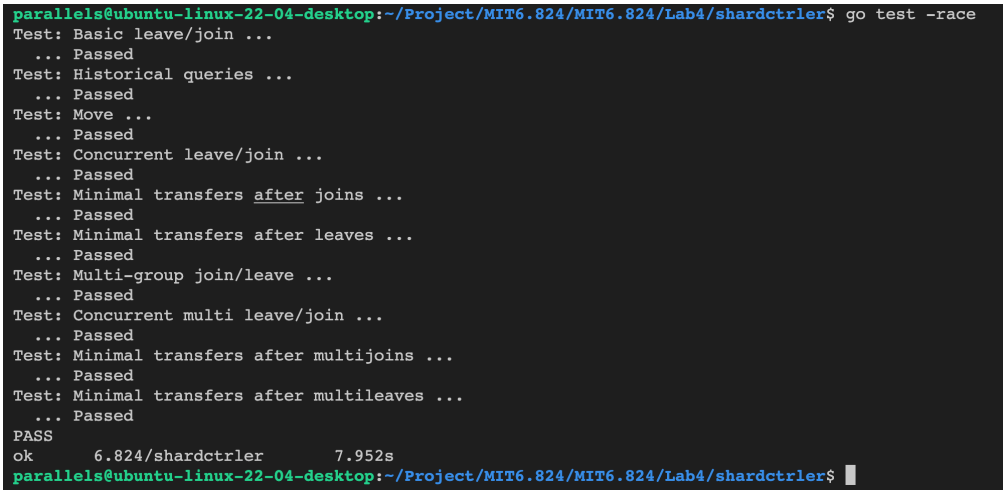


Fig 19. 4A 测试结果

通过多次实验，代码都可以通过测试。

3 Part B: Sharded Key/Value Server

3.1 准备工作

在开始之前，我先通过阅读 Mit6.824 的官方手册整理出了整个 Lab4-4B 部分的基本解读和原理。如下所示。4 B 部分旨在构建一个分片的键值存储系统。这个系统被设计为可以在分布式环境中运行，具备容错性和伸缩性。以下是这个实验的核心内容和任务:

1. 实现分片键值存储: 系统将键值对数据分布在不同的“分片”中，每个分片由一个副本组管理。分片的目的是为了提高系统的可伸缩性和性能，因为不同的分片可以并行处理请求。

2. 分片控制器: 分片控制器是一个特殊的组件, 负责决定哪个副本组负责哪个分片。控制器在副本组加入、离开或者负载需要重新平衡时, 会重新分配分片。

3. 处理配置更改: 当分片控制器更改分片到副本组的映射时, 系统需要能够处理这些更改。这可能涉及将分片的数据从一个副本组迁移到另一个副本组。

4. 保证一致性和容错性: 系统需要使用如 Raft 这样的一致性协议来在副本之间同步状态，同时保证即使在部分节点失败的情况下也能正常工作。

5. 客户端操作: 系统应该支持基本的键值存储操作, 如 Get、Put 和 Append。即使在进行分片迁移和配置更改的时候，这些操作也应该是可用的，并保持一致性。

6. 处理重复和网络不稳定: 实现机制以处理网络延迟或重复请求所带来的问题, 确保系统的稳定性和一致性。

3.2 Lab4-4B 的核心组件和调用逻辑

通过分析和解读 4B 部分的任务，可以学习到，4B 部分的核心组件如下所示。

1. ShardKV: 实现了分片的键值存储服务器。每个 ShardKV 实例运行在一个副本组内, 管理一部分分片 (shards) 的数据。ShardKV 负责处理客户端的 Get、Put、Append 等请求。

2. ShardCtrler (Shard Controller): 分片控制器是一个独立的服务, 负责维护分片到副本组的映射。ShardKV 实例定期向 ShardCtrler 查询最新的分片配置。

3. Raft: 用于在 ShardKV 副本组内实现一致性的协议。每个 ShardKV 实例都是 Raft 集群的一部分, 用于确保跨多个副本的状态一致性。

4. Client: 客户端向 ShardKV 发送键值存储操作请求（如 Get、Put、Append）。客户端首先查询 ShardCtrler 以确定负责特定键的 ShardKV 副本组。

关键组件之间的调用逻辑如下所示。

1. 客户端请求: 客户端（client.go 中的 Clerk 结构）发起请求（Get、Put、Append）。

2. 确定分片位置: 客户端首先通过 shardctrler 确定键所在的分片和负责该分片的 ShardKV 服务器。

3. 请求发送: 客户端向负责相应分片的 ShardKV 服务器发送请求。

4. ShardKV 服务器处理请求: 在 server.go 中, ShardKV 服务器接收到请求并调用相应的处理函数（例如 Command 函数）。

5. Raft 一致性: 修改性请求（Put、Append）会通过 raft 实例进行一致性协调。

6. 应用到状态机: 确认通过 Raft 后, 请求被应用到相应的分片状态机（在 shard.go 中定义）。

7. 返回结果: 处理完成后, 结果被发送回客户端。

8. 配置和迁移: 分片配置的更改和分片迁移逻辑也在 server.go 中处理。

4B 部分的核心架构如图所示。

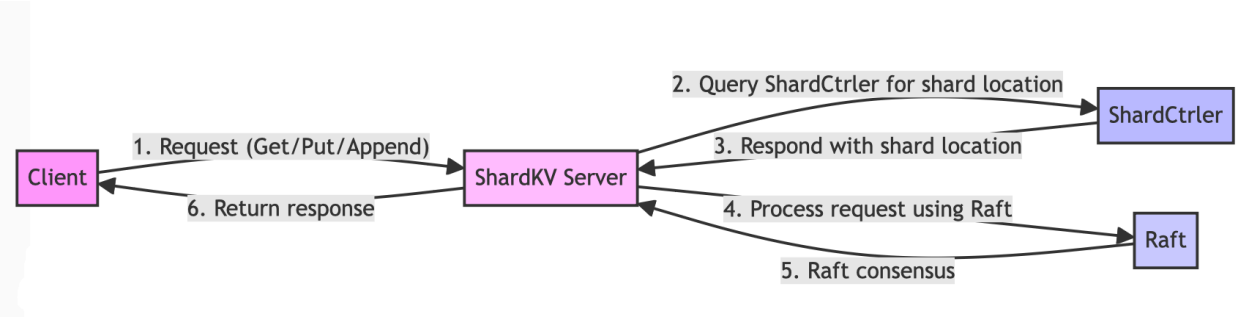


Fig 20. 4B 核心架构

函数之间的调用逻辑如下图所示。

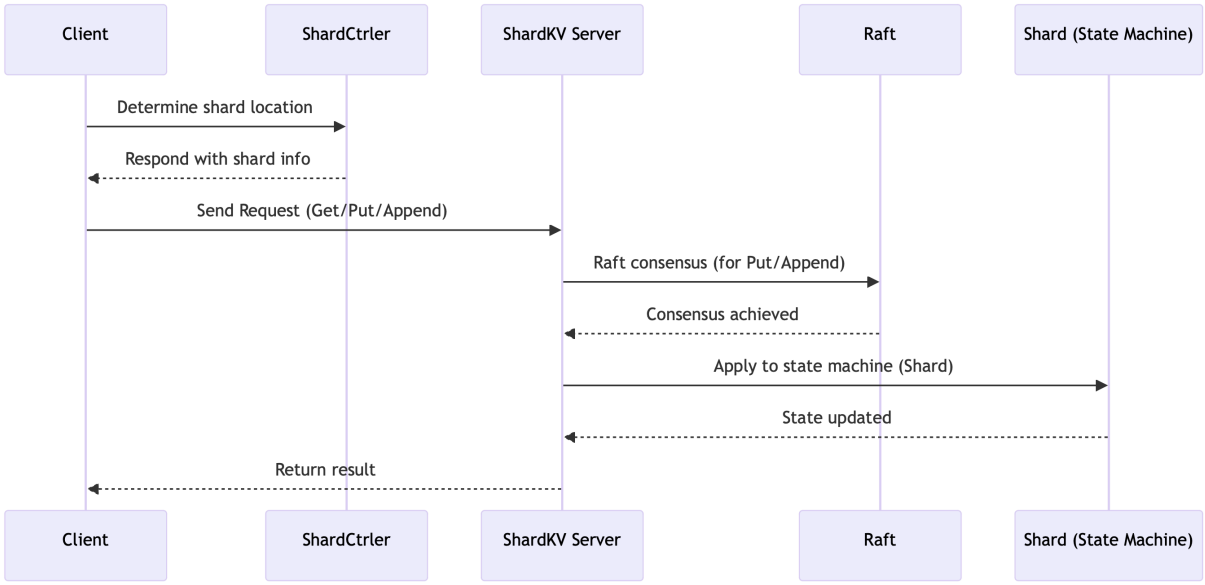


Fig 21. 4B 函数调用逻辑

3.3 common.go 编写

这一部分我在原始 6.824 给的代码基础上进行了一定程度的封装，所以没有使用原始的形式，这样可以得到更好的代码维护性和可读性。

3.3.1 common.go 所要编写的组件整理

common.go 文件中定义了在 MIT 6.824 Lab 4-4B 分片键值存储系统中用到的一些共通的结构和类型。以下是该文件中定义的主要内容和它们的用途：

1. 常量定义：

- `ExecuteTimeout`：执行超时时间，用于控制操作的超时。
- `ConfigureMonitorTimeout`：配置监控超时时间，用于监控配置更新。

- `MigrationMonitorTimeout`: 迁移监控超时时间，用于监控分片迁移。
- `GCMonitorTimeout`: 垃圾回收监控超时时间，用于数据清理。
- `EmptyEntryDetectorTimeout`: 空条目检测超时时间，用于 Raft 的日志同步。
- `Debug`: 用于控制是否输出调试信息。

2. Err 类型定义: 定义了一系列可能的错误类型, 如 OK, ErrNoKey, ErrWrongGroup 等, 用于表示操作的结果状态。

3. ShardStatus 类型定义: 定义了分片的状态, 如 Serving，Pulling，BePulling，GCing, 用于表示分片在不同操作中的状态。

4. OperationContext 结构: 存储操作的上下文信息, 包括最大已应用的命令 ID 和最后的响应。

5. Command 结构: 定义了命令的结构, 包括操作类型和数据。

6. CommandType 枚举: 定义了命令的类型, 如 Operation, Configuration, InsertShards, DeleteShards，EmptyEntry。

7. OperationOp 枚举: 定义了操作命令的具体操作类型, 如 OpPut, OpAppend, OpGet 。

8. CommandRequest 结构: 定义了客户端命令请求的结构, 包括键、值、操作类型、客户端 ID 和命令 ID。

9. CommandResponse 结构: 定义了命令响应的结构, 包括错误信息和返回值。

10. ShardOperationRequest 结构: 定义了分片操作请求的结构，包括配置编号和分片 ID 列表。

11. ShardOperationResponse 结构: 定义了分片操作响应的结构，包括错误信息、配置编号、分片数据和操作上下文。

整个 common.go 里面的组件和组件之间的关系，我都总结道了一个类图里，如下所示。

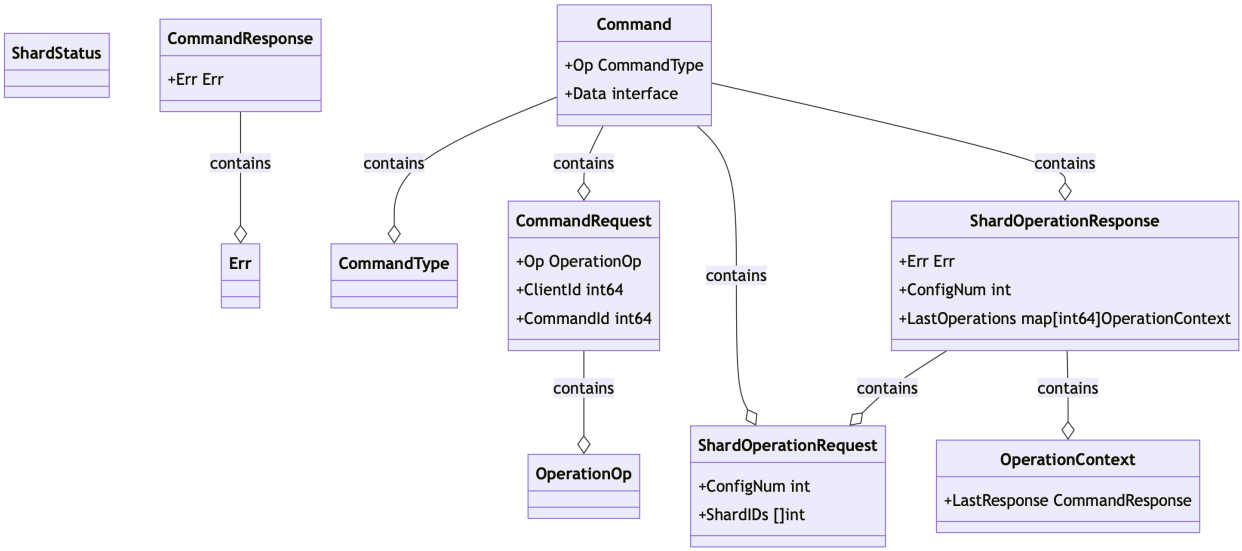


Fig 22. common.go 的组件关系

3.3.2 一些常量的定义

```go
// 常量定义，用于设置不同超时和时间间隔。
const (
    ExecuteTimeout            = 500 * time.Millisecond // 执行超时时间。
    ConfigureMonitorTimeout   = 100 * time.Millisecond // 配置监控超时时间。
    MigrationMonitorTimeout   = 50 * time.Millisecond  // 迁移监控超时时间。
    GCMonitorTimeout          = 50 * time.Millisecond  // 垃圾回收监控超时时间。
    EmptyEntryDetectorTimeout = 200 * time.Millisecond // 空条目检测超时时间。
)
```

3.3.3 定义分片的状态

分片的状态有四种：

1. 正在服务
2. 正在拉取数据
3. 被拉取数据
4. 正在进行垃圾回收

在这里是直接使用整型来标识状态的。

```go
// ShardStatus 定义了分片的状态。
type ShardStatus uint8
const (
    Serving   ShardStatus = iota // 正在服务
    Pulling                      // 正在拉取数据
    BePulling                    // 被拉取数据
    GCing                        // 正在进行垃圾回收
)
// String 方法将 ShardStatus 转换为字符串。
func (status ShardStatus) String() string {
    switch status {
    case Serving:
        return "Serving"
    case Pulling:
        return "Pulling"
    case BePulling:
        return "BePulling"
    case GCing:
        return "GCing"
    }
    panic(fmt.Sprintf("unexpected ShardStatus %d", status))
}
```

3.3.4 OperationContext 部分编写

OperationContext 用于存储有关特定操作的上下文信息。这在分布式系统中尤其重要，因为它有助于处理网络延迟、重复请求和恢复操作等问题。

字段:

- `MaxAppliedCommandId`（类型为 `int64`）：记录了对于特定客户端已经应用的最大命令 ID。这个字段用于检测和处理重复的命令请求，确保每个命令只被执行一次。
- `LastResponse`（类型为指向 `CommandResponse` 的指针）：存储了最后一次操作的响应。这对于快速响应重复的请求非常有用，因为可以直接返回先前已经计算过的结果，而不需要重新执行整个命令。

当然同时也需要实现一个 OperationContext 的深拷贝，用于创建一个新的，和之前一样的 OperationContext 副本。

代码如下所示。

```go
// OperationContext 存储操作的上下文信息。
type OperationContext struct {
    MaxAppliedCommandId int64            // 已应用的最大命令ID。
    LastResponse        *CommandResponse // 最后的响应。
}

// deepCopy 方法对 OperationContext 进行深拷贝。
func (context OperationContext) deepCopy() OperationContext {
    return OperationContext{context.MaxAppliedCommandId, …
        &CommandResponse{context.LastResponse.Err, context.LastResponse.Value}}
}
```

3.3.5 定义命令的类型

这是 rpc 中关键的一部分，命令类型有如下几种：

1. 操作命令（而操作命令分为三种，和 Lab3 的相同）

- put 操作
- append 操作
- get 操作

2. 配置命令

3. 插入分片命令

4. 删除分片命令

5. 空条目命令

操作命令的定义代码如下所示。

```go
// OperationOp 定义了操作命令的具体操作类型。
type OperationOp uint8
const (
    OpPut    OperationOp = iota // PUT 操作
    OpAppend                    // APPEND 操作
    OpGet                       // GET 操作
)
```

命令类型的定义如下所示。

```go
// CommandType 定义了命令的类型。
type CommandType uint8

const (
    Operation     CommandType = iota // 操作命令
    Configuration                    // 配置命令
    InsertShards                     // 插入分片命令
    DeleteShards                     // 删除分片命令
    EmptyEntry                       // 空条目命令
)
```

3.3.6 用 Command 结构对命令类型进行封装

封装的内容其实就是命令类型 + 命令类型所需要对应的数据。

而每一种命令所需要的命令类型是不同的，因此 Data 应该设计为一个接口类型，这样 Command 就可以接收不同类型的命令类型数据。

代码如下所示。

```go
// Command 定义了命令的结构。
type Command struct {
    Op   CommandType // 命令类型。
    Data interface{} // 命令数据。
}
```

当然不同类型的命令的创建是不一样的，即 Command 本质上是有 5 种的，对应 5 种不同的命令类型，因此 Command 的实例化需要分成五种情况，代码如下所示。

```go
// NewOperationCommand 创建一个新的操作命令。
func NewOperationCommand(request *CommandRequest) Command {
    return Command{Operation, *request}
}
// NewConfigurationCommand 创建一个新的配置命令。
func NewConfigurationCommand(config *shardctrler.Config) Command {
    return Command{Configuration, *config}
```

```go
}
// NewInsertShardsCommand 创建一个新的插入分片命令。
func NewInsertShardsCommand(response *ShardOperationResponse) Command {
    return Command{InsertShards, *response}
}
// NewDeleteShardsCommand 创建一个新的删除分片命令。
func NewDeleteShardsCommand(request *ShardOperationRequest) Command {
    return Command{DeleteShards, *request}
}
// NewEmptyEntryCommand 创建一个新的空条目命令。
func NewEmptyEntryCommand() Command {
    return Command{EmptyEntry, nil}
}
```

3.3.7 3.3.7 定义核心 rpc 结构

上面所有部分都是为这一部分作准备，这一部分才是在定义核心 rpc 的结构。

CommandRequest 结构代码如下所示，内容都非常简单，基本上结合 Lab3 和 4A 部分就能写出来。

```go
// CommandRequest 定义了客户端命令请求的结构。
type CommandRequest struct {
    Key       string      // 键名。
    Value     string      // 值。
    Op        OperationOp // 操作类型。
    ClientId  int64       // 客户端ID。
    CommandId int64       // 命令ID。
}
```

CommandResponse 结构代码如下所示，内容都非常简单，基本上结合 Lab3 和 4A 部分就能写出来。

```go
// CommandResponse 定义了命令响应的结构。
type CommandResponse struct {
    Err   Err    // 错误信息。
    Value string // 返回值。
}
```

3.3.8 定义分片核心 rpc 结构

这一部分的 rpc 有别于上一部分的 rpc 结构，这一部分是专门为分片服务的 rpc 结构。

ShardOperationRequest 结构下所示。

```go
// ShardOperationRequest 定义了分片操作请求的结构。
```

```go
type ShardOperationRequest struct {
    ConfigNum int   // 配置编号。
    ShardIDs  []int // 分片ID列表。
}
```

ShardOperationResponse 结构如下所示。

```go
// ShardOperationResponse 定义了分片操作响应的结构。
type ShardOperationResponse struct {
    Err            Err                        // 错误信息。
    ConfigNum      int                        // 配置编号。
    Shards         map[int]map[string]string  // 分片数据。
    LastOperations map[int64]OperationContext // 操作上下文。
}
```

3.3.9 分片部分的一些代码定义和功能的实现

定义了与分片（Shard）相关的数据结构和方法，这是构建分片键值存储系统的一个关键部分。在分片键值存储系统中，整个键值存储空间被分割成多个分片，每个分片包含一部分键值对。这一部分代码非常简单，其实就是对分片部分的一些具体实现而已，在报告中不展开详细讲了，具体可以看源代码(shard.go)。

3.4 编写 Client 部分代码

3.4.1 client 部分解读

client.go 中的 Clerk 结构体实现了分片键值存储系统的客户端逻辑。这个客户端主要负责与 ShardKv 服务通信, 以执行键值存储操作, 如 Get、Put 和 Append。以下是 Clerk 的主要运行原理和函数调用逻辑:

1. 初始化: 客户端 (Clerk) 通过 MakeClerk 函数初始化。这个过程中, 客户端创建了一个分片控制器 (shardctrler.Clerk) 的实例, 用于查询键所属的分片和负责该分片的 ShardKv 服务器。

2. 发送请求: 当客户端想要执行 Get、Put 或 Append 操作时, 它调用相应的方法 (clerk.Get、Clerk.Put 、Clerk.Append)。这些方法内部都调用 Clerk.Command 方法来实际发送请求。

3. 确定分片位置: 在 Clerk. Command 方法中, 客户端首先通过分片控制器查询键所在的分片位置及负责该分片的 Shardkv 服务器组。

4. 与 ShardKv 服务器交互: 客户端根据查询到的分片信息, 找到对应的 ShardKv 服务器, 并向其发送请求。如果服务器响应表示不是负责的服务器 (例如因为分片迁移), 客户端会重新查询分片控制器, 并重试请求。

5. 处理响应: 客户端从 ShardKv 服务器接收响应并返回给原始调用者。

这一部分的整体架构如图所示。

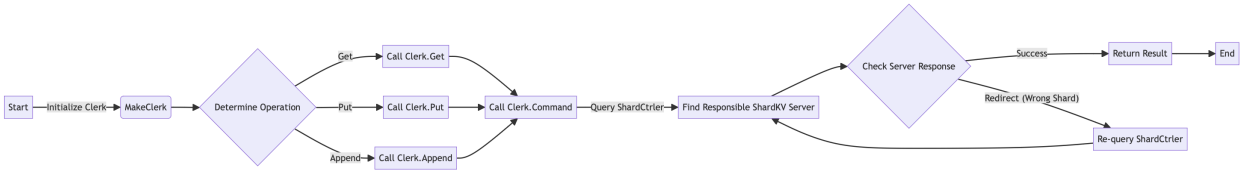


Fig 23. Client 基本架构

3.4.2 先编写两个工具函数

实现很简单，直接看代码就可以了。

```go
// key2shard 函数用于确定一个键属于哪个分片。
// 根据键的第一个字符计算分片号，确保分片的分布是均匀的。
func key2shard(key string) int {
    shard := 0
    if len(key) > 0 {
        shard = int(key[0])
    }
    shard %= shardctrler.NShards // 使用模运算来确定分片号。
    return shard
}

// nrand 生成一个随机的 int64 值。
// 这个函数使用加密安全的随机数生成器，以确保生成的ID是唯一和随机的。
// 这对于区分不同的客户端操作非常重要。
func nrand() int64 {
    max := big.NewInt(int64(1) << 62)
    bigx, _ := rand.Int(rand.Reader, max)
    return bigx.Int64()
}
```

3.4.3 Client(Clerk) 结构体编写和对应实例化的函数

客户端需要维护字段有以下几个字段：

- `makeEnd` 函数回调：根据服务器名称生成 RPC 端点。
- `leaderIds`：与 4A 一样，领导者的 ID。
- `clientId` 和 `commandId`：与 4A 一样。

MakeClerk 就是实例化一个 Clerk 对象，和 4A 是一样的，具体细节见代码所示。

```go
// Clerk 结构体代表客户端。
// 它用于与分片键/值存储系统的不同部分交互。
type Clerk struct {
```

```go
    sm        *shardctrler.Clerk             // 分片控制器的客户端，用于查询分片信息。
    config    shardctrler.Config             // 当前的分片配置信息。
    makeEnd   func(string) *labrpc.ClientEnd // 函数，根据服务器名称生成RPC端点。
    leaderIds map[int]int                    // 记录每个组的领导者ID，用于优化RPC调用。
    clientId  int64                          // 客户端ID，由 nrand() 生成，用于唯一标识客户端。
    commandId int64                          // 操作ID，与 clientId 配合，唯一定义一个操作。
}
// MakeClerk 创建并初始化一个新的 Clerk 实例。
// ctrlers[] 是创建 shardctrler.Clerk 所需的。
// make_end 函数用于将服务器名称转换为 RPC 客户端端点。
func MakeClerk(ctrlers []*labrpc.ClientEnd, makeEnd func(string) *labrpc.ClientEnd) *Clerk {
    ck := &Clerk{
        sm:        shardctrler.MakeClerk(ctrlers), // 创建分片控制器的客户端。
        makeEnd:   makeEnd,                        // 设置 makeEnd 函数。
        leaderIds: make(map[int]int),              // 初始化领导者ID映射。
        clientId:  nrand(),                        // 生成客户端ID。
        commandId: 0,                              // 初始化命令ID为0。
    }
    ck.config = ck.sm.Query(-1) // 查询最新的分片配置。
    return ck
}
```

3.4.4 定义操作命令的三个操作对应的接口

和 4A 同样，三个接口都需要直接调用 Command 接口进行网络调用的发送。因此其实这三个函数也是被封装了多一层的。

具体代码如下所示。

```go
// Get 方法用于从键/值存储中获取键的值。
func (ck *Clerk) Get(key string) string {
    return ck.Command(&CommandRequest{Key: key, Op: OpGet})
}
// Put 方法用于向键/值存储中添加或更新键的值。
func (ck *Clerk) Put(key string, value string) {
    ck.Command(&CommandRequest{Key: key, Value: value, Op: OpPut})
}
// Append 方法用于向键/值存储中的现有键添加值。
func (ck *Clerk) Append(key string, value string) {
    ck.Command(&CommandRequest{Key: key, Value: value, Op: OpAppend})
}
```

3.4.5 定义发送网络调用的 Command 接口

Command 函数基本流程如下所示。

1. 确定分片位置: 首先, Command 函数通过分片控制器 (ShardCtrler) 查询请求的键所在的分片及其负责的 ShardKV 服务器组。

2. 发送请求至 ShardKV 服务器: 根据分片控制器的回应, Command 函数会找到负责该分片的 ShardKV 服务器, 并向其发送请求。

3. 处理 ShardKV 服务器的响应:

- 如果 ShardKV 服务器成功处理了请求，`Command` 函数将返回结果。
- 如果 ShardKV 服务器指示请求被发送到了错误的分片（可能由于分片迁移），`Command` 函数需要重新查询分片控制器，然后重试请求。

4. 返回结果: 一旦请求被正确的 ShardKV 服务器处理, Command 函数将把服务器的响应结果返回给客户端。

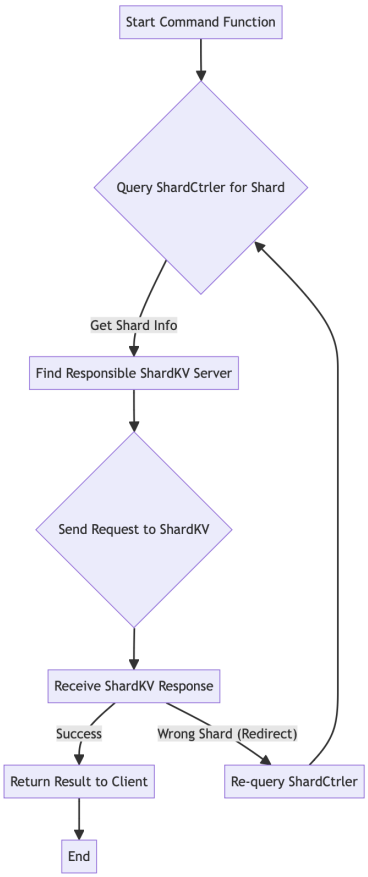


Fig 24. Command 接口逻辑

代码如下所示。

```go
// Command 方法处理客户端的所有命令请求，包括 Get、Put 和 Append。
func (ck *Clerk) Command(request *CommandRequest) string {
```

```go
    // 设置客户端和命令ID。
    request.ClientId, request.CommandId = ck.clientId, ck.commandId

    for {
        // 根据键计算分片号，并找到负责该分片的组ID。
        shard := key2shard(request.Key)
        gid := ck.config.Shards[shard]
        if servers, ok := ck.config.Groups[gid]; ok {
            if _, ok = ck.leaderIds[gid]; !ok {
                ck.leaderIds[gid] = 0 // 初始时，假设第一个服务器是领导者。
            }
            oldLeaderId := ck.leaderIds[gid]
            newLeaderId := oldLeaderId

            // 循环尝试，直到找到正确的领导者并获取响应。
            for {
                var response CommandResponse
                ok := ck.makeEnd(servers[newLeaderId]).Call("ShardKV.Command", ...
                    request, &response)
                if ok && (response.Err == OK || response.Err == ErrNoKey) {
                    ck.commandId++
                    return response.Value
                } else if ok && response.Err == ErrWrongGroup {
                    break // 错误的组，需要重新查询配置。
                } else {
                    // 尝试下一个服务器。
                    newLeaderId = (newLeaderId + 1) % len(servers)
                    if newLeaderId == oldLeaderId {
                        break // 如果尝试了所有服务器，重新查询配置。
                    }
                    continue
                }
            }
        }
        time.Sleep(100 * time.Millisecond) // 短暂暂停，然后重新查询配置。
        ck.config = ck.sm.Query(-1)        // 从分片控制器查询最新配置。
    }
}
```

3.5 编写 Server 部分代码

3.5.1 Server 部分解读

server.go 文件中实现的是分片键值存储系统的服务端逻辑，即 ShardKV 服务器。这个服务器处理客户端的请求，并通过 Raft 协议保持数据一致性。以下是其基本架构和函数调用逻辑的描述。

基本架构: 1. ShardKV 结构体: 代表了一个 ShardKV 服务器实例，包含了服务器状态、配置信息、Raft 实例等。

2. Raft 实例: 用于在 ShardKV 的副本间保持数据一致性。

3. 分片控制器客户端 (ShardCtrler Clerk): 用于与分片控制器交互, 获取分片信息。

4. RPC 服务器: 处理来自客户端的 RPC 请求（如 Get、Put、Append）。

设计架构图如下所示。

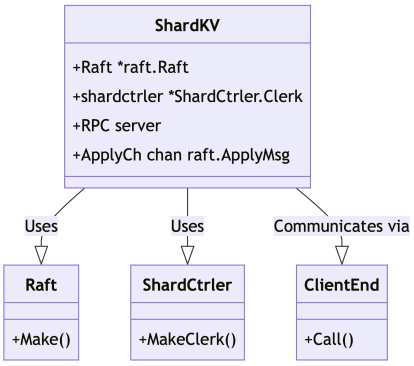


Fig 25. Server 部分基本架构

函数间的调用逻辑:

1. StartServer: 初始化 ShardKV 服务器。

2. applier: 处理来自 Raft 的日志应用。

3. Command: 处理来自客户端的命令请求。

4. GetShardsData 和 DeleteShardsData: 处理分片数据的迁移。

5. 其他内部函数: 处理状态管理、分片迁移、快照等。

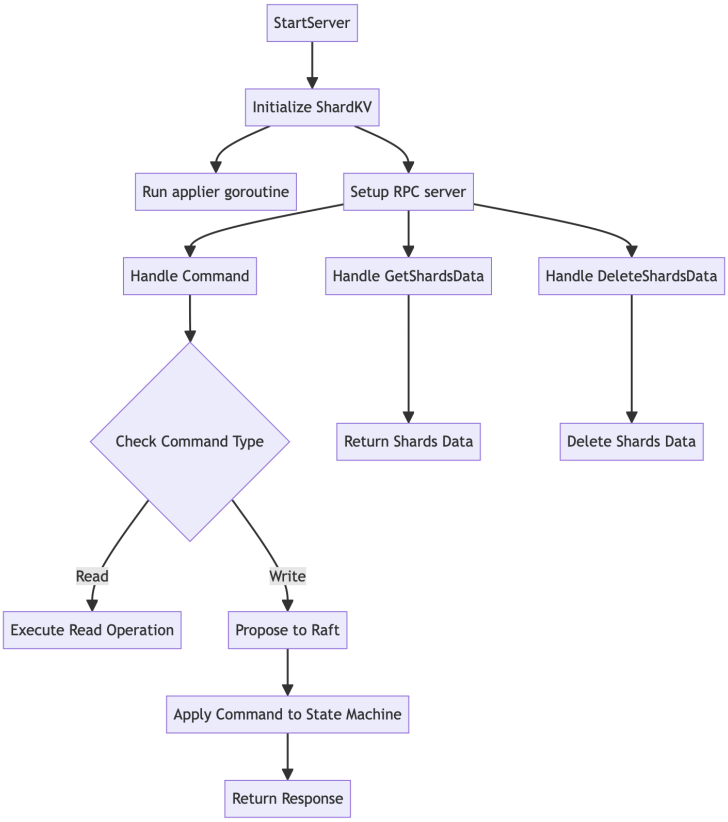


Fig 26. ShardKV 整体架构

3.5.2 ShardKV 结构定义

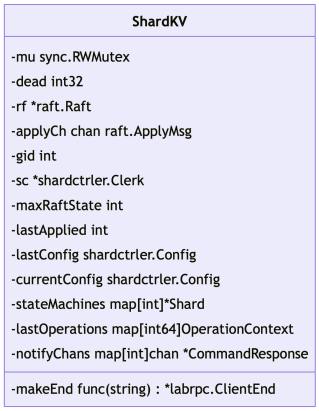


Fig 27. ShardKV 结构定义

代码如下所示。

```go
type ShardKV struct {
    mu      sync.RWMutex       // 读写互斥锁，用于在并发环境下保护共享资源。
    dead    int32              // 用于标记服务是否已经终止。
    rf      *raft.Raft         // Raft协议实例，用于在多个副本间保持一致性。
    applyCh chan raft.ApplyMsg // 从Raft层接收已提交的日志的通道。

    makeEnd func(string) *labrpc.ClientEnd // 函数，用于根据服务器名称创建RPC端点。
    gid     int                            // 组ID，用于标识这个服务属于哪个群组。
    sc      *shardctrler.Clerk             // 分片控制器的客户端，用于与分片控制器交互。

    maxRaftState int // 日志达到这个大小时进行快照。
    lastApplied  int // 最后应用的日志索引，用于防止状态机回滚。

    lastConfig    shardctrler.Config // 上一个配置版本。
    currentConfig shardctrler.Config // 当前配置版本。

    stateMachines  map[int]*Shard                // 每个分片的键值存储状态机。
    lastOperations map[int64]OperationContext    // 记录每个客户端的最后一次操作，用于检测重复请求。
    notifyChans    map[int]chan *CommandResponse // 用于通知客户端操作结果的通道。
}
```

3.5.3 处理来自客户端的请求的 Command 函数

Command 的工作流程和 4A 部分其实是差不多的。

1. 请求验证:

- Command 函数首先检查请求是否为重复请求。如果是重复的非 Get 请求，它会直接返回之前已

经计算过的结果，而不需要重新通过 Raft 协议执行。

- 然后，它检查当前分片 (Shard) 是否可以服务于此请求的键。如果不能（可能是因为键所在的分片不在当前服务器组中），它会返回一个 ErrWrongGroup 错误。

2. Raft 一致性协议: 对于新的或有效的请求, Command 函数会将请求封装为一个命令, 并通过其内部的 Raft 实例来达成一致性。这是为了确保在分布式系统中的所有副本之间维持数据的一致性。

3. 等待执行结果: 请求被发送到 Raft 层后, Command 函数会等待该命令被应用到状态机。这是通过监听一个专门的通道（applych）来完成的，该通道会接收来自 Raft 的已应用日志条目。

4. 返回响应: 一旦命令被应用, Command 函数会根据操作的结果创建一个响应, 并将这个响应返回给客户端。

```go
// Command 方法处理来自客户端的命令请求。
func (kv *ShardKV) Command(request *CommandRequest, response *CommandResponse) {
    // 对请求加读锁。
    kv.mu.RLock()
    // 如果是重复的非Get请求，则直接返回之前的结果，不需要Raft层参与。
    if request.Op != OpGet && kv.isDuplicateRequest(request.ClientId, ...
        request.CommandId) {
        lastResponse := kv.lastOperations[request.ClientId].LastResponse
        response.Value, response.Err = lastResponse.Value, lastResponse.Err
        kv.mu.RUnlock()
        return
    }
    // 如果当前分片无法服务此键，返回ErrWrongGroup错误。
    if !kv.canServe(key2shard(request.Key)) {
        response.Err = ErrWrongGroup
        kv.mu.RUnlock()
        return
    }
    kv.mu.RUnlock()

    // 执行命令。
    kv.Execute(NewOperationCommand(request), response)
}
```

在这个函数结束的时候，执行了 Execute 接口。

3.5.4 Execute 函数编写

Execute 用于执行客户端的命令并等待响应，流程如下所示。

1. 提交命令至 Raft 层: 首先, 函数通过调用 kv.rf.start (command) 将命令提交到 Raft 层。这个过程中, Start 方法返回三个值：提交的命令的索引 (index)、当前的任期 (term) 和该节点是否为领导者 (isLeader)。

2. 检查领导者身份: 如果当前节点不是领导者, 函数将返回一个 ErrWrongLeader 错误, 表示客户端应该向其他服务器节点发送请求。

3. 等待命令执行结果:

- 如果当前节点是领导者，函数会锁定互斥锁并获取与命令索引相关联的通知通道 (ch)。这个通道用于接收来自 Raft 层的命令执行结果。
- 函数使用 select 语句等待通道的响应或超时。如果在指定的 ExecuteTimeout 时间内收到响应，它会将结果设置到响应参数 `response` 中；如果超时，则设置一个 `ErrTimeout` 错误。

4. 异步释放通知通道: 为了防止内存泄漏，函数启动一个新的 goroutine 来异步移除过时的通知通道。

流程可以整理成流程图，如图所示。

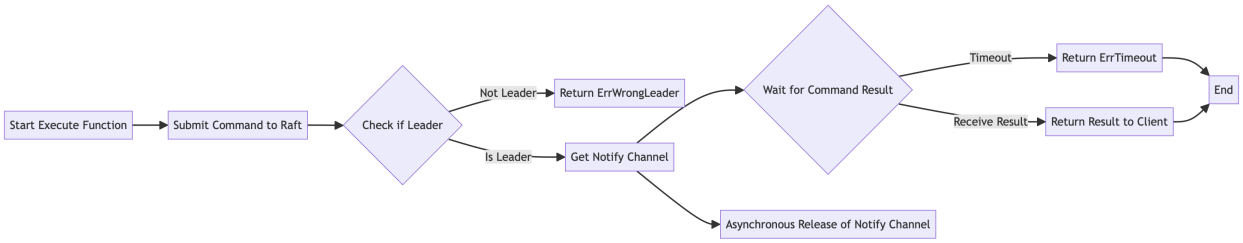


Fig 28. Execute 方法基本框架

代码如下所示。

```go
// Execute 方法执行命令，并等待响应。
func (kv *ShardKV) Execute(command Command, response *CommandResponse) {
    // 向Raft层提交命令。
    index, _, isLeader := kv.rf.Start(command)
    if !isLeader {
        response.Err = ErrWrongLeader
        return
    }

    // 等待命令执行结果。
    kv.mu.Lock()
    ch := kv.getNotifyChan(index)
    kv.mu.Unlock()
    select {
    case result := <-ch:
        response.Value, response.Err = result.Value, result.Err
    case <-time.After(ExecuteTimeout):
        response.Err = ErrTimeout
    }

    // 异步释放通知通道。
    go func() {
```

```
        kv.mu.Lock()
        kv.removeOutdatedNotifyChan(index)
        kv.mu.Unlock()
    }()
}
```

3.5.5 GetShardsData 和 DeleteShardsData

GetShardsData

作用: 获取指定分片的数据。

1. 领导者检查: 首先确认当前 ShardKv 实例是否为 Raft 集群的领导者。只有领导者才能执行此操作, 以确保数据的一致性。

2. 锁定资源: 使用读锁定来保护共享资源，防止在收集数据的过程中数据被修改。

3. 检查配置: 检查当前的配置版本是否已经包含了请求的分片。如果请求的分片属于未来的配置, 返回 ErrNotReady 错误。

4. 收集分片数据: 对于请求中包含的每个分片 ID, 收集该分片的数据并存储在响应中。这些数据包括分片内的所有键值对。

5. 收集操作上下文: 同时, 收集有关这些分片的最近操作的上下文，用于处理可能的重复请求。

6. 设置响应: 最后, 设置响应中的配置版本和错误状态。

DeleteShardsData

作用: 删除特定配置下的分片数据。

1. 领导者检查: 同样首先检查当前 ‘Shardkv‘实例是否为 Raft 集群的领导者。

2. 锁定资源: 使用读锁定来保护共享资源。

3. 检查配置: 如果当前配置版本大于请求中的配置版本，直接返回 OK 。这意味着分片的数据已经不再需要。

4. 删除分片数据: 如果当前配置版本不大于请求中的配置版本, 使用 Execute 方法执行删除分片数据的操作。这通常涉及到更新状态机以及通过 Raft 协议确保操作在所有副本上一致执行。

5. 设置响应: 设置响应中的错误状态。

```
// GetShardsData 方法用于获取指定分片的数据。
func (kv *ShardKV) GetShardsData(request *ShardOperationRequest, response ...
    *ShardOperationResponse) {
    // 确保只有领导者执行此操作。
    if _, isLeader := kv.rf.GetState(); !isLeader {
        response.Err = ErrWrongLeader
        return
    }
    // 锁定以保护共享资源。
    kv.mu.RLock()
    defer kv.mu.RUnlock()
    // 检查当前配置是否已经包含请求的分片。
    if kv.currentConfig.Num < request.ConfigNum {
```

```go
        response.Err = ErrNotReady
        return
    }
    // 收集请求的分片数据。
    response.Shards = make(map[int]map[string]string)
    for _, shardID := range request.ShardIDs {
        response.Shards[shardID] = kv.stateMachines[shardID].deepCopy()
    }
    // 收集分片的操作上下文。
    response.LastOperations = make(map[int64]OperationContext)
    for clientID, operation := range kv.lastOperations {
        response.LastOperations[clientID] = operation.deepCopy()
    }
    response.ConfigNum, response.Err = request.ConfigNum, OK
}
// DeleteShardsData 方法用于删除分片数据。
func (kv *ShardKV) DeleteShardsData(request *ShardOperationRequest, response ...
    *ShardOperationResponse) {
    // 确保只有领导者执行此操作。
    if _, isLeader := kv.rf.GetState(); !isLeader {
        response.Err = ErrWrongLeader
        return

    kv.mu.RLock()
    if kv.currentConfig.Num > request.ConfigNum {
        response.Err = OK
        kv.mu.RUnlock()
        return
    }
    kv.mu.RUnlock()
    var commandResponse CommandResponse
    kv.Execute(NewDeleteShardsCommand(request), &commandResponse)
    response.Err = commandResponse.Err
}
```

3.5.6 isDuplicateRequest 编写

这个函数和 4A 部分是完全一样的，这里不做过多解释，代码如下所示。

```go
// isDuplicateRequest 方法检查是否为重复请求。
func (kv *ShardKV) isDuplicateRequest(clientId int64, requestId int64) bool {
    operationContext, ok := kv.lastOperations[clientId]
    return ok && requestId ≤ operationContext.MaxAppliedCommandId
}
```

3.5.7 canServe 编写

用于判断当前的 Raft 组（即当前的 ShardKv 实例）是否能够服务于指定的分片（shard）。这个函数的主要目的是确保只有负责特定分片的 Raft 组才处理对该分片的请求。以下是它的工作原理：

检查分片归属:

- 函数首先检查 `currentConfig.Shards[shardID]` 是否等于当前 ShardKV 实例的组 ID (`kv.gid`)。这一步是为了确认当前实例是否是负责处理指定分片 `shardID` 的 Raft 组。
- `currentConfig` 是当前配置，其中包含有关哪个 Raft 组负责哪个分片的信息。`Shards` 是一个数组，其索引对应分片 ID，值是负责该分片的组 ID。

检查分片状态:

- 函数接着检查该分片的状态是否为 Serving 或 GCing。Serving 状态表示分片是活动的且可以服务请求，而 GCing (Garbage Collecting) 状态表示分片正在进行垃圾收集但仍然可以处理某些类型的请求。
- `stateMachines` 是一个映射，其键是分片 ID，值是存储每个分片状态和数据的 `Shard` 结构。

返回结果:

- 如果上述两个条件都满足，则函数返回 `true`，表明当前的 ShardKV 实例可以服务于该分片。
- 否则，返回 `false`，表明当前实例不应该处理该分片的请求。

代码写起来其实可以总结成一个语句来写。

```go
// canServe 方法检查此 raft 组是否能服务于此分片。
func (kv *ShardKV) canServe(shardID int) bool {
    return kv.currentConfig.Shards[shardID] == kv.gid && ...
        (kv.stateMachines[shardID].Status == Serving || ...
        kv.stateMachines[shardID].Status == GCing)
}
```

3.5.8 server 核心结构 applier 的详解和编写

applier 函数是 Shardkv 结构体中定义的一个方法, 充当一个持续运行的后台 goroutine。它的主要职责是应用由 Raft 协议提交的日志条目到 ShardKV 的状态机，并在必要时处理快照。

1. 持续监听: applier 函数在一个循环中运行, 持续监听 Raft 层提交的日志消息。这是通过从 applyCh 通道接收消息来实现的。

2. 处理日志消息:

- 对于每个接收到的消息，`applier` 首先检查它是否包含有效的命令 (`CommandValid`)。
- 如果命令有效，它会锁定 `ShardKV` 实例来同步访问共享资源。

3. 命令索引检查: 如果命令的索引小于或等于 lastApplied（表示已经应用过），则跳过该命令。

4. 应用命令到状态机: 根据命令的类型（如 Operation, Configuration, InsertShards, DeleteShards, EmptyEntry），执行相应的操作方法, 如 applyOperation, applyConfiguration 等。

5. 通知结果: 如果当前节点是领导者且命令属于当前任期, 它会通知与该命令索引相关联的通知通道。

6. 快照处理: 检查是否需要根据 Raft 日志的大小创建快照。如果是, 则创建快照。

7. 处理包含快照的消息: 如果消息包含有效快照 (SnapshotValid), 则尝试安装快照并更新 lastApplied 。

8. 异常处理: 如果收到非预期类型的消息, 触发 panic。

applier 部分的流程图如图所示，流程图可以很好的描述 applier 和其他函数之间的调用逻辑和调用条件等信息，非常有利于我们理解整个 server 的核心工作原理。

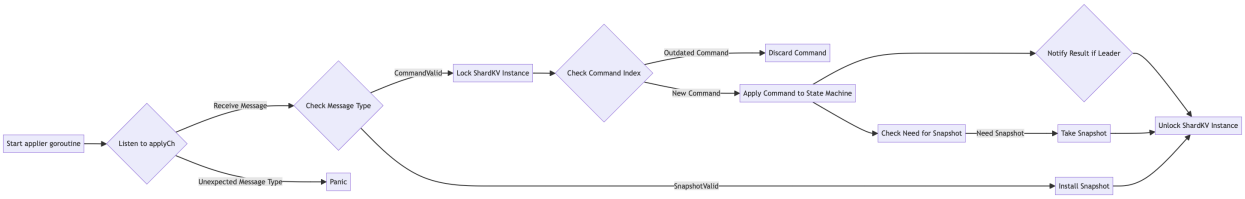


Fig 29. applier 的基本架构

代码如下所示。

```go
// applier goroutine 负责处理由 Raft 提交的日志条目。
// 它将日志条目应用到状态机，并在必要时处理快照。
func (kv *ShardKV) applier() {
    // 持续循环，直到实例被标记为终止。
    for !kv.killed() {
        select {
        case message := <-kv.applyCh: // 从 applyCh 通道接收 raft ...
            层提交的日志消息。
            // 记录接收到的消息，用于调试。
            DPrintf("{Node %v}{Group %v} tries to apply message %v", kv.rf.Me(), ...
                kv.gid, message)
            if message.CommandValid { // 检查消息是否包含有效的命令。
                kv.mu.Lock() // 加锁，以同步对共享资源的访问。
                // 如果消息已过时（即命令索引小于或等于 lastApplied），则丢弃它。
                if message.CommandIndex ≤ kv.lastApplied {
                    DPrintf("{Node %v}{Group %v} discards outdated message %v ...
                        because a newer snapshot which lastApplied is %v has been ...
                        restored", kv.rf.Me(), kv.gid, message, kv.lastApplied)
                    kv.mu.Unlock()
                    continue
                }
```

```go
            kv.lastApplied = message.CommandIndex // 更新 lastApplied 为当前消息的命令索引。

            var response *CommandResponse
            command := message.Command.(Command) // 类型断言，获取 Command 实例。
            // 根据命令类型执行相应操作。
            switch command.Op {
            case Operation: // 处理普通操作命令。
                operation := command.Data.(CommandRequest)
                response = kv.applyOperation(&message, &operation)
            case Configuration: // 处理配置更改命令。
                nextConfig := command.Data.(shardctrler.Config)
                response = kv.applyConfiguration(&nextConfig)
            case InsertShards: // 处理插入分片命令。
                shardsInfo := command.Data.(ShardOperationResponse)
                response = kv.applyInsertShards(&shardsInfo)
            case DeleteShards: // 处理删除分片命令。
                shardsInfo := command.Data.(ShardOperationRequest)
                response = kv.applyDeleteShards(&shardsInfo)
            case EmptyEntry: // 处理空条目命令。
                response = kv.applyEmptyEntry()
            }

            // 仅在当前节点为领导者且消息属于当前任期时，通知相关的通道。
            if currentTerm, isLeader := kv.rf.GetState(); isLeader && message.CommandTerm == currentTerm {
                ch := kv.getNotifyChan(message.CommandIndex)
                ch <- response // 向通道发送响应。
            }

            // 检查是否需要创建快照。
            needSnapshot := kv.needSnapshot()
            if needSnapshot {
                kv.takeSnapshot(message.CommandIndex) // 创建快照。
            }
            kv.mu.Unlock()
        } else if message.SnapshotValid { // 处理包含有效快照的消息。
            kv.mu.Lock()
            // 如果可以安装快照，则恢复快照并更新 lastApplied。
            if kv.rf.CondInstallSnapshot(message.SnapshotTerm, message.SnapshotIndex, message.Snapshot) {
                kv.restoreSnapshot(message.Snapshot)
                kv.lastApplied = message.SnapshotIndex
            }
            kv.mu.Unlock()
```

```
            } else {
                // 如果收到意外的消息类型，则触发 panic。
                panic(fmt.Sprintf("unexpected Message %v", message))
            }
        }
    }
}
```

3.5.9 applier 里面需要调用的对应分片操作的函数

applier 里面所要根据命令类型调用的操作函数有：

- applyOperation 处理普通操作命令。
- applyConfiguration 处理配置更改命令。
- applyInsertShards 处理插入分片命令。
- applyDeleteShards 处理删除分片命令。
- applyEmptyEntry 处理空条目命令。

下面所有函数的实现细节，我们都写在了代码的注释里面。

applyOperation applyOperation 方法应用一个操作到对应的分片状态机。

```
func (kv *ShardKV) applyOperation(message *raft.ApplyMsg, operation ...
    *CommandRequest) *CommandResponse {
    // 初始化响应结构体。
    var response *CommandResponse
    // 计算键所在的分片 ID。
    shardID := key2shard(operation.Key)
    // 检查当前 ShardKV 实例是否可以服务于这个分片。
    if kv.canServe(shardID) {
        // 检查是否是重复的非Get请求。
        if operation.Op != OpGet && kv.isDuplicateRequest(operation.ClientId, ...
            operation.CommandId) {
            // 如果是重复的请求，则直接返回之前的响应。
            DPrintf("{Node %v}{Group %v} doesn't apply duplicated message %v to ...
                stateMachine because maxAppliedCommandId is %v for client %v", ...
                kv.rf.Me(), kv.gid, message, ...
                kv.lastOperations[operation.ClientId], operation.ClientId)
            return kv.lastOperations[operation.ClientId].LastResponse
        } else {
            // 应用日志到状态机，并返回响应。
            response = kv.applyLogToStateMachines(operation, shardID)
            // 如果不是 Get 操作，则更新操作上下文。
            if operation.Op != OpGet {
                kv.lastOperations[operation.ClientId] = ...
                    OperationContext{operation.CommandId, response}
```

```go
            }
            return response
        }
    }
    // 如果当前实例不能服务于这个分片，则返回错误。
    return &CommandResponse{ErrWrongGroup, ""}
}
```

applyConfiguration

applyConfiguration 方法应用新的分片配置。

```go
// applyConfiguration 方法应用新的分片配置。
func (kv *ShardKV) applyConfiguration(nextConfig *shardctrler.Config) *CommandResponse {
    // 检查配置是否是连续的下一个版本。
    if nextConfig.Num == kv.currentConfig.Num+1 {
        // 记录配置更新的日志信息。
        DPrintf("{Node %v}{Group %v} updates currentConfig from %v to %v", kv.rf.Me(), kv.gid, kv.currentConfig, nextConfig)
        // 更新分片状态。
        kv.updateShardStatus(nextConfig)
        // 更新配置信息。
        kv.lastConfig = kv.currentConfig
        kv.currentConfig = *nextConfig
        return &CommandResponse{OK, ""}
    }
    // 如果配置不是预期的下一个版本，则记录日志并返回错误。
    DPrintf("{Node %v}{Group %v} rejects outdated config %v when currentConfig is %v", kv.rf.Me(), kv.gid, nextConfig, kv.currentConfig)
    return &CommandResponse{ErrOutDated, ""}
}
```

applyInsertShards

applyInsertShards 方法应用插入分片的操作。

```go
func (kv *ShardKV) applyInsertShards(shardsInfo *ShardOperationResponse) *CommandResponse {
    // 检查插入的分片是否适用于当前配置版本。
    if shardsInfo.ConfigNum == kv.currentConfig.Num {
        // 记录接受插入分片的日志信息。
        DPrintf("{Node %v}{Group %v} accepts shards insertion %v when currentConfig is %v", kv.rf.Me(), kv.gid, shardsInfo, kv.currentConfig)
        // 遍历所有分片数据并插入到状态机中。
        for shardId, shardData := range shardsInfo.Shards {
            shard := kv.stateMachines[shardId]
            // 检查分片状态是否为 Pulling。
```

```go
            if shard.Status == Pulling {
                // 插入分片数据。
                for key, value := range shardData {
                    shard.KV[key] = value
                }
                // 更新分片状态为 GCing。
                shard.Status = GCing
            } else {
                // 如果分片状态不是 Pulling，则记录日志并跳出循环。
                DPrintf("{Node %v}{Group %v} encounters duplicated shards insertion %v when currentConfig is %v", kv.rf.Me(), kv.gid, shardsInfo, kv.currentConfig)
                break
            }
        }
        // 更新操作上下文。
        for clientId, operationContext := range shardsInfo.LastOperations {
            if lastOperation, ok := kv.lastOperations[clientId]; !ok || lastOperation.MaxAppliedCommandId < operationContext.MaxAppliedCommandId {
                kv.lastOperations[clientId] = operationContext
            }
        }
        return &CommandResponse{OK, ""}
    }
    // 如果插入的分片不适用于当前配置，则记录日志并返回错误。
    DPrintf("{Node %v}{Group %v} rejects outdated shards insertion %v when currentConfig is %v", kv.rf.Me(), kv.gid, shardsInfo, kv.currentConfig)
    return &CommandResponse{ErrOutDated, ""}
}
```

applyDeleteShards

applyDeleteShards 方法应用删除分片的操作。

```go
func (kv *ShardKV) applyDeleteShards(shardsInfo *ShardOperationRequest) *CommandResponse {
    // 检查删除的分片是否适用于当前配置版本。
    if shardsInfo.ConfigNum == kv.currentConfig.Num {
        // 记录接受删除分片的日志信息。
        DPrintf("{Node %v}{Group %v}'s shards status are %v before accepting shards deletion %v when currentConfig is %v", kv.rf.Me(), kv.gid, kv.getShardStatus(), shardsInfo, kv.currentConfig)
        // 遍历所有分片ID并更新状态。
        for _, shardId := range shardsInfo.ShardIDs {
            shard := kv.stateMachines[shardId]
            // 检查分片状态并进行相应的更新。
```

```go
            if shard.Status == GCing {
                shard.Status = Serving
            } else if shard.Status == BePulling {
                kv.stateMachines[shardId] = NewShard()
            } else {
                // 如果分片状态不是 GCing 或 BePulling，则记录日志并跳出循环。
                DPrintf("{Node %v}{Group %v} encounters duplicated shards deletion %v when currentConfig is %v", kv.rf.Me(), kv.gid, shardsInfo, kv.currentConfig)
                break
            }
        }
        // 记录完成删除分片后的日志信息。
        DPrintf("{Node %v}{Group %v}'s shards status are %v after accepting shards deletion %v when currentConfig is %v", kv.rf.Me(), kv.gid, kv.getShardStatus(), shardsInfo, kv.currentConfig)
        return &CommandResponse{OK, ""}
    }
    // 如果删除的分片不适用于当前配置，则记录日志并返回错误。
    DPrintf("{Node %v}{Group %v}'s encounters duplicated shards deletion %v when currentConfig is %v", kv.rf.Me(), kv.gid, shardsInfo, kv.currentConfig)
    return &CommandResponse{OK, ""}
}
```

applyEmptyEntry

applyEmptyEntry 方法用于处理空条目的情况，通常用于领导者选举后的日志同步。

```go
func (kv *ShardKV) applyEmptyEntry() *CommandResponse {
    // 返回一个表示成功的响应，不进行任何状态修改。
    return &CommandResponse{OK, ""}
}
```

3.5.10 快照相关方法编写

needSnapshot

检查是否需要根据日志大小创建快照。

这一部分和 2D，Lab3 的 3B 的原理都是相同的，超过阈值就创建。

```go
// needSnapshot 方法检查是否需要根据日志大小创建快照。
func (kv *ShardKV) needSnapshot() bool {
    // 如果设置了快照阈值并且 Raft 状态大小超过该阈值，则返回 true。
    return kv.maxRaftState != -1 && kv.rf.GetRaftStateSize() ≥ kv.maxRaftState
}
```

takeSnapshot

这一部分和 2D，Lab3 的 3B 的原理都是相同的。

```go
// takeSnapshot 方法创建快照，并通过 Raft 存储快照。
func (kv *ShardKV) takeSnapshot(index int) {
    // 创建一个新的缓冲区以存储快照数据。
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    // 对状态机、操作记录等进行编码。
    e.Encode(kv.stateMachines)
    e.Encode(kv.lastOperations)
    e.Encode(kv.currentConfig)
    e.Encode(kv.lastConfig)
    // 通过 Raft 存储快照。
    kv.rf.Snapshot(index, w.Bytes())
}
```

restoreSnapshot

这一部分和 2D，Lab3 的 3B 的原理都是相同的。

```go
// restoreSnapshot 方法从快照中恢复状态。
func (kv *ShardKV) restoreSnapshot(snapshot []byte) {
    // 检查快照是否有效。
    if snapshot == nil || len(snapshot) == 0 {
        // 如果快照无效，则初始化状态机。
        kv.initStateMachines()
        return
    }
    r := bytes.NewBuffer(snapshot)
    d := labgob.NewDecoder(r)
    var stateMachines map[int]*Shard
    var lastOperations map[int64]OperationContext
    var currentConfig shardctrler.Config
    var lastConfig shardctrler.Config
    // 从快照中解码状态机和其他状态信息。
    if d.Decode(&stateMachines) != nil ||
        d.Decode(&lastOperations) != nil ||
        d.Decode(&currentConfig) != nil ||
        d.Decode(&lastConfig) != nil {
        // 如果解码失败，记录错误日志。
        DPrintf("{Node %v}{Group %v} restores snapshot failed", kv.rf.Me(), kv.gid)
    }
    // 恢复解码后的状态。
    kv.stateMachines, kv.lastOperations, kv.currentConfig, kv.lastConfig = ...
        stateMachines, lastOperations, currentConfig, lastConfig
}
```

3.5.11 一次性初始化所有分片的状态机函数

思路非常简单，直接 for 循环遍历所有的分片的状态机即可。

```go
// initStateMachines 方法初始化所有分片的状态机。
func (kv *ShardKV) initStateMachines() {
    // 遍历所有分片，初始化状态机。
    for i := 0; i < shardctrler.NShards; i++ {
        if _, ok := kv.stateMachines[i]; !ok {
            // 如果某个分片的状态机尚未初始化，则创建一个新的状态机。
            kv.stateMachines[i] = NewShard()
        }
    }
}
```

3.5.12 处理配置更新的 configureAction 函数编写

configureAction 函数是 ShardKV 结构体中的一个方法, 用于处理配置更新。在分片键值存储系统中, 配置更新通常涉及分片的重新分配。当分片控制器更改分片到组的映射时, ShardKv 服务器需要更新其本地配置以反映这些更改。

1. 检查分片状态: 函数首先确定当前 ShardKv 实例是否可以获取新的配置。这通常意味着所有分片都处于服务状态，没有正在进行的迁移。

2. 查询最新配置: 如果可以获取新配置, configureAction 会通过分片控制器客户端查询最新的配置。

3. 更新配置: 如果获取到的新配置是当前配置的下一个版本, Shardkv 实例会更新其本地配置, 以反映新的分片分配。

4. 执行配置更新命令: 更新配置后, configureAction 会执行必要的命令来处理配置变更, 这可能涉及更新分片状态或者迁移分片数据。

流程图如下所示。

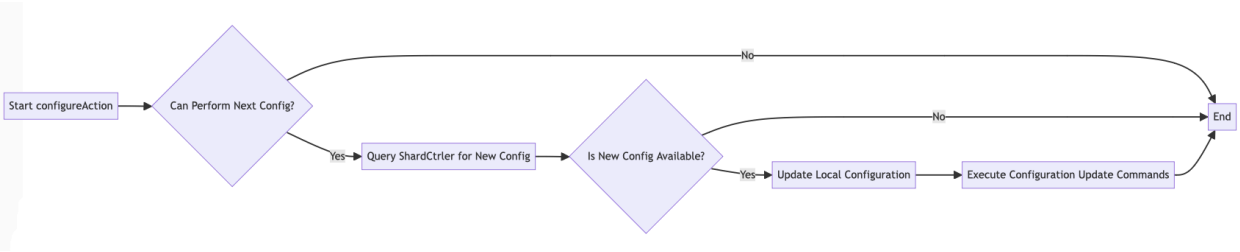


Fig 30. configureAction 方法逻辑

代码如下所示。

```go
// configureAction 方法用于在当前 ShardKV ...
    实例可以服务于所有分片时，获取最新的配置。
```

```go
func (kv *ShardKV) configureAction() {
    // 初始设置为可以获取新配置。
    canPerformNextConfig := true
    kv.mu.RLock()
    // 遍历所有分片以确认它们都处于服务状态。
    for _, shard := range kv.stateMachines {
        if shard.Status != Serving {
            // 如果有任何分片不在服务状态，标记为不可获取新配置。
            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.Me(), kv.gid, kv.getShardStatus(), kv.currentConfig)
            break
        }
    }
    currentConfigNum := kv.currentConfig.Num
    kv.mu.RUnlock()
    // 如果可以获取新配置，则查询下一个配置。
    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.Me(), kv.gid, nextConfig, currentConfigNum)
            // 执行配置更新命令。
            kv.Execute(NewConfigurationCommand(&nextConfig), &CommandResponse{})
        }
    }
}
```

3.5.13 从其他组拉取分片数据 migrationAction 的编写

工作流程如下所示。

1. 检索需要迁移的分片: 函数首先检索所有状态为 Pulling 的分片, 即那些需要从其他 ShardKV 组拉取数据的分片。

2. 分片数据拉取: 对于每个需要迁移的分片, migrationAction 创建一个单独的 goroutine 来处理迁移。这些 goroutine 会向原分片所属的组发送 RPC 请求，以获取分片的数据。

3. 处理迁移数据: 当迁移数据被成功拉取后, 通过 Execute 函数执行插入分片数据的命令, 将其合并到当前 ShardKV 实例的状态机中。

4. 等待所有迁移完成: migrationAction 等待所有分片迁移任务完成, 确保所有必要的数据都已经迁移到当前 Shardkv 实例。

流程图如下所示。

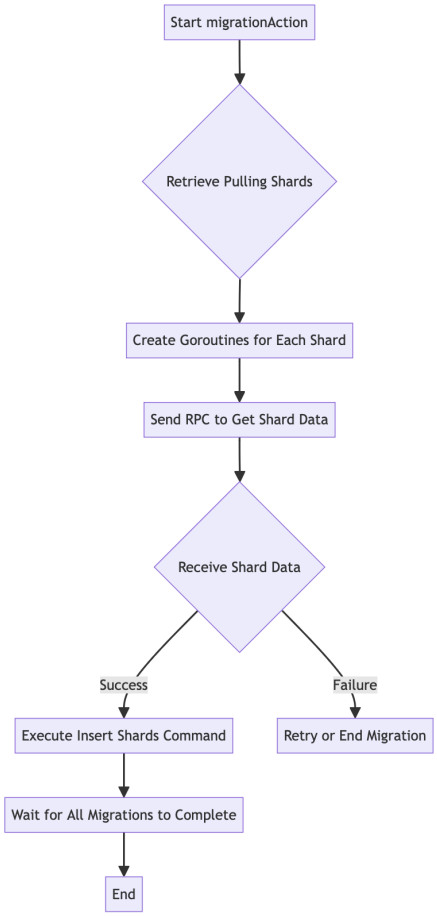


Fig 31. migrationAction 基本逻辑

代码如下所示。

```go
// migrationAction 方法用于从其他组拉取分片数据。
func (kv *ShardKV) migrationAction() {
    kv.mu.RLock()
    // 获取所有需要拉取的分片ID。
    gid2shardIDs := kv.getShardIDsByStatus(Pulling)
    var wg sync.WaitGroup
    // 遍历每个组和它们的分片ID。
    for gid, shardIDs := range gid2shardIDs {
        DPrintf("{Node %v}{Group %v} starts a PullTask to get shards %v from group %v when config is %v", kv.rf.Me(), kv.gid, shardIDs, gid, kv.currentConfig)
        wg.Add(1)
        // 创建 goroutine 来执行拉取任务。
        go func(servers []string, configNum int, shardIDs []int) {
            defer wg.Done()
            pullTaskRequest := ShardOperationRequest{configNum, shardIDs}
```

```go
            for _, server := range servers {
                var pullTaskResponse ShardOperationResponse
                srv := kv.makeEnd(server)
                // 发送 RPC 请求获取分片数据。
                if srv.Call("ShardKV.GetShardsData", &pullTaskRequest, ...
                    &pullTaskResponse) && pullTaskResponse.Err == OK {
                    DPrintf("{Node %v}{Group %v} gets a PullTaskResponse %v and ...
                        tries to commit it when currentConfigNum is %v", ...
                        kv.rf.Me(), kv.gid, pullTaskResponse, configNum)
                    // 提交拉取到的分片数据。
                    kv.Execute(NewInsertShardsCommand(&pullTaskResponse), ...
                        &CommandResponse{})
                }
            }
        }(kv.lastConfig.Groups[gid], kv.currentConfig.Num, shardIDs)
    }
    kv.mu.RUnlock()
    wg.Wait()
}
```

3.5.14 删除不需要的分片数据接口 gcAction 编写

工作流程如下所示。

1. 分片检查函数首先检索所有状态为 GCing 的分片, 这些分片的数据需要从当前 ShardKV 实例中删除。

2. 分片数据删除: 对于每个需要垃圾回收的分片, gcAction 创建一个单独的 goroutine 来处理删除。这些 goroutine 会向原分片所属的组发送 RPC 请求，以指示删除分片数据。

3. 处理删除确认: 当接收到确认分片数据已被成功删除的响应时, 通过 Execute 函数执行删除分片数据的命令, 将其从当前 ShardKV 实例的状态机中移除。

4. 等待所有垃圾回收完成: gcAction 等待所有分片垃圾回收任务完成, 确保所有不再需要的数据都已经从当前 ShardKv 实例中删除。

代码如下所示。

```go
// gcAction 方法用于删除已经不再需要的分片数据。
func (kv *ShardKV) gcAction() {
    kv.mu.RLock()
    // 获取所有需要删除的分片ID。
    gid2shardIDs := kv.getShardIDsByStatus(GCing)
    var wg sync.WaitGroup
    // 遍历每个组和它们的分片ID。
    for gid, shardIDs := range gid2shardIDs {
        DPrintf("{Node %v}{Group %v} starts a GCTask to delete shards %v in group ...
            %v when config is %v", kv.rf.Me(), kv.gid, shardIDs, gid, ...
            kv.currentConfig)
```

```go
        wg.Add(1)
        // 创建 goroutine 来执行删除任务。
        go func(servers []string, configNum int, shardIDs []int) {
            defer wg.Done()
            gcTaskRequest := ShardOperationRequest{configNum, shardIDs}
            for _, server := range servers {
                var gcTaskResponse ShardOperationResponse
                srv := kv.makeEnd(server)
                // 发送 RPC 请求删除分片数据。
                if srv.Call("ShardKV.DeleteShardsData", &gcTaskRequest, …
                    &gcTaskResponse) && gcTaskResponse.Err == OK {
                    DPrintf("{Node %v}{Group %v} deletes shards %v in remote …
                        group successfully when currentConfigNum is %v", …
                        kv.rf.Me(), kv.gid, shardIDs, configNum)
                    // 提交删除分片的命令。
                    kv.Execute(NewDeleteShardsCommand(&gcTaskRequest), …
                        &CommandResponse{})
                }
            }
        }(kv.lastConfig.Groups[gid], kv.currentConfig.Num, shardIDs)
    }
    kv.mu.RUnlock()
    wg.Wait()
}
```

3.5.15 StartServer 实例化一个 KV 机器

这个也是 server 部分的核心函数, 负责构建实例化一个 KV 服务器, 具体的详细流程如下所示。**1. 初始化 ShardKV 结构体:** 创建并初始化 ShardKv 结构体实例, 设置必要的字段, 如 Raft 实例、分片控制器客户端、状态机等。

2. 恢复快照 (如果有) : 如果存在持久化的快照数据, 恢复该快照到服务器的状态机。

3. 启动 Raft 应用日志处理器: 启动 applier goroutine, 用于持续监听并应用 Raft 层提交的日志条目。

4. 启动配置、迁移、垃圾回收监控器: 启动 configureAction, migrationAction, 和 gcAction 等后台 goroutines, 分别负责处理配置更新、数据迁移和垃圾回收。

5. RPC 服务器注册和启动: 设置并启动 RPC 服务器, 注册 ShardKv 结构体的方法作为 RPC 处理函数。

流程图如下所示。

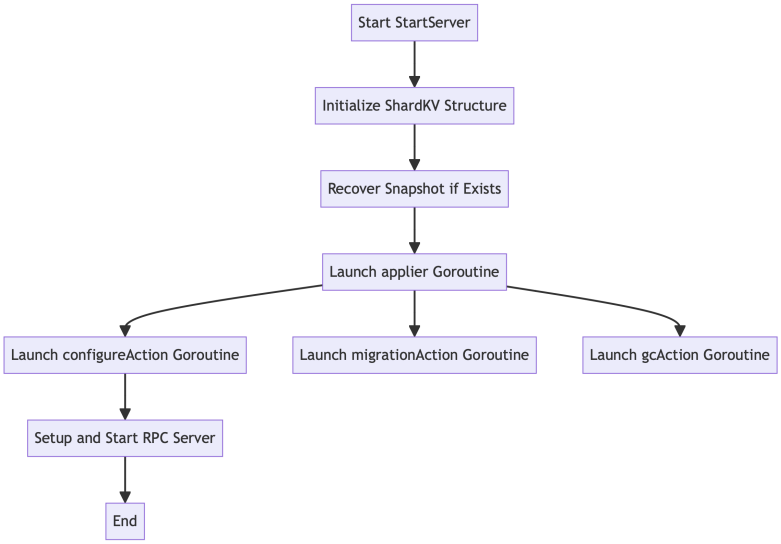


Fig 32. StartServer 基本逻辑

具体代码如下所示。

```go
func StartServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister, maxRaftState int, gid int, ctrlers []*labrpc.ClientEnd, makeEnd func(string) *labrpc.ClientEnd) *ShardKV {
    // 在 Go 的 RPC 库中注册需要序列化/反序列化的结构体
    labgob.Register(Command{})
    labgob.Register(CommandRequest{})
    labgob.Register(shardctrler.Config{})
    labgob.Register(ShardOperationResponse{})
    labgob.Register(ShardOperationRequest{})

    // 创建一个通道，用于接收 Raft 层提交的日志消息
    applyCh := make(chan raft.ApplyMsg)

    // 初始化一个 ShardKV 结构体实例
    kv := &ShardKV{
        dead:           0, // 标记服务是否已终止
        rf:             raft.Make(servers, me, persister, applyCh), // 创建 Raft 实例
        applyCh:        applyCh, // Raft 提交的日志消息通道
        makeEnd:        makeEnd, // 函数，用于创建 RPC 端点
        gid:            gid, // 组 ID
        sc:             shardctrler.MakeClerk(ctrlers), // 创建分片控制器客户端
        lastApplied:    0, // 最后应用的日志索引
        maxRaftState:   maxRaftState, // 日志大小达到该值时进行快照
        currentConfig:  shardctrler.DefaultConfig(), // 当前配置
        lastConfig:     shardctrler.DefaultConfig(), // 上一个配置
```

```go
        stateMachines:  make(map[int]*Shard), // 每个分片的状态机
        lastOperations: make(map[int64]OperationContext), // 记录每个客户端的最后一次操作
        notifyChans:    make(map[int]chan *CommandResponse), // 通知客户端操作结果的通道
    }
    // 恢复快照 (如果有)
    kv.restoreSnapshot(persister.ReadSnapshot())
    // 启动一个 goroutine 来应用 Raft 提交的日志到状态机
    go kv.applier()
    // 启动配置更新监控器 goroutine
    go kv.Monitor(kv.configureAction, ConfigureMonitorTimeout)
    // 启动数据迁移监控器 goroutine
    go kv.Monitor(kv.migrationAction, MigrationMonitorTimeout)
    // 启动垃圾回收监控器 goroutine
    go kv.Monitor(kv.gcAction, GCMonitorTimeout)
    // 启动检测当前任期内是否有日志条目的监控器 goroutine, 以避免死锁
    go kv.Monitor(kv.checkEntryInCurrentTermAction, EmptyEntryDetectorTimeout)
    // 打印日志, 表示节点启动
    DPrintf("{Node %v}{Group %v} has started", kv.rf.Me(), kv.gid)
    return kv
}
```

3.6 4B 部分测试

```
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab4/shardkv$ go test -race
Test: static shards ...
  ... Passed
Test: join then leave ...
  ... Passed
Test: snapshots, join, and leave ...
  ... Passed
Test: servers miss configuration changes...
  ... Passed
Test: concurrent puts and configuration changes...
  ... Passed
Test: more concurrent puts and configuration changes...
  ... Passed
Test: unreliable 1...
  ... Passed
Test: unreliable 2...
  ... Passed
Test: unreliable 3...
  ... Passed
Test: shard deletion (challenge 1) ...
  ... Passed
Test: concurrent configuration change and restart (challenge 1)...
  ... Passed
Test: unaffected shard access (challenge 2) ...
  ... Passed
Test: partial migration shard access (challenge 2) ...
  ... Passed
PASS
ok      6.824/shardkv   134.190s
parallels@ubuntu-linux-22-04-desktop:~/Project/MIT6.824/MIT6.824/Lab4/shardkv$
```

Fig 33. 4B 测试结果

4B 部分结果如图所示。可以通过多次测试。

4 实验过程中遇到的困难和对应的解决办法

困难 1: Raft 日志应用顺序不一致

问题现象: 在 `applier` 函数中，可能会遇到 Raft 日志应用顺序与提交顺序不一致的问题。这可能导致状态机的状态更新不一致，从而影响分片控制器的正确性。

解决方法: 确保 `applyLogToStateMachine` 函数在更新状态机时遵循 Raft 日志的顺序。可以通过在状态机中维护一个递增的日志索引，并在应用每个日志条目时检查其索引是否正确。

困难 2: 分片控制器中的重复 RPC 请求

问题现象: 在 `Command` 函数中，可能会遇到重复处理相同 RPC 请求的问题，尤其是在网络不稳定时。这会导致不必要的资源消耗和潜在的状态不一致。

解决方法: 加强 `isDuplicateRequest` 函数的逻辑，确保能够准确识别和过滤掉重复的 RPC 请求。可以通过为每个客户端请求维护一个唯一的序列号，并在服务器端记录每个客户端处理的最后一个序列号。

困难 3: 分片重平衡算法的效率问题

问题现象: 在 `Join` 或 `Leave` 方法中，分片重平衡算法可能效率不高，导致分片分配不均或响应时间过长。

解决方法: 优化分片重平衡算法，例如使用更高效的数据结构或算法逻辑。可以考虑在计算分片迁移时采用更加高效的负载均衡策略，如平均分配法或权重分配法。

困难 4: 分片控制器状态机的并发问题

问题现象: 在多个 goroutine 同时访问或修改 `MemoryConfigStateMachine` 的状态时，可能会出现竞态条件或数据不一致。

解决方法: 在状态机的操作中添加适当的锁机制，确保并发访问的安全性。例如，使用读写锁（`sync.RWMutex`）来控制对状态机的读写访问。

困难 5: 超时和网络问题导致的服务不可用

问题现象: 在处理 RPC 请求时，由于网络延迟或服务超时，客户端可能会收到错误的响应或无响应。

解决方法: 增强客户端和服务器的错误处理和超时机制。例如，实现超时重试逻辑，在 RPC 调用超时时自动重试。同时，优化网络通信和错误处理代码，提高服务的健壮性和容错能力。

5 实验总结

本实验是在 Lab2 的基础上实现一个 multi-raft 的 KV 存储服务，并支持切片在不同 raft 组上的动态迁移，同时保持线性一致性。总体来说，lab4 的总体难度是远高于 Lab3 的。

lab4 没有太多可参考的资料，需要我们自己设计整个迁移流程并保证线性一致性。对于才学习分布式一个学期的我们来说，这是一个非常困难的过程，有时即使浪费大量时间，也找不到报错存在的原因。此外，lab4 的两个 challenge 要求代码更加完善，需要考虑许多边缘情况，如果一开始没有考虑清楚好一些边缘的情况，那么最终实现的代码就基本会出现 bug。总之，Lab4 是我们在 6.824 课程中花费时间最多的一个实验。

不过幸好，我们最终成功地克服了 lab4 这个难题，能够通过 shardctrler、shardkv 以及两个 challenge 的所有测试。在实现过程中，我们几乎每一行都添加了适当的代码来增加代码的可读性。

在 shardctrler 实验中，它实际上是一个高可用的集群配置管理服务，主要记录了当前每个 raft 组中的副本数和节点的 endpoint，以及当前每个 shard 被分配到哪个 raft 组。shardctrler 还可以通过用户手动或内置策略的方式来动态增加或删除 raft 组，以更有效地利用集群资源。此外客户端的每次请求都可以通过查询 shardctrler 来路由到正确的数据节点，这有点类似于 HDFS Master 的角色。当然，客户端也可以缓存配置来减轻 shardctrler 的负担。

在 ShardKV 实验中，ShardKV 是一个分布式键值存储系统，使用了 Raft 协议来保证数据的一致性和可用性。ShardKV 系统的目标是支持配置更改、分片迁移和容错，同时向用户提供线性一致性的服务。我们一开始需要设计包括 hardctrler 和多个 Raft 组用于处理数据操作。每个 Raft 组负责一组分片的读写请求。我们还设定了配置更新、分片迁移和状态维护等操作必须由 Raft 组的 Leader 来执行，确保一致性。在这个过程中，我们也遇到了不少问题，例如动态迁移、幂等性和数据一致性。为了保证数据迁移的正确性，我们使用了配置版本和状态变量，并确保每个分片的状态更改都是幂等的。此外，我们还需要考虑节点重启和网络分区等情况，不然就很容易出现各种未知的 bug。此外，在实现 ShardKV 系统时一定要特别注意空日志的问题，确保 Raft 组的领导者能够迅速达到最新状态，这个问题主要在节点频繁重启时出现，需要特殊考虑并且解决。

总的来说，整个 lab4 这个实验都涉及到了许多复杂的、交叉的、边缘的情况，如配置管理、数据

迁移、幂等性和活锁处理等等。想要完成这个实验，不仅需要对这个实验整体有一个很清晰的脉络，还需要对实验中遇到的各种特殊情况和 bug 有足够的应对方法。总得来说，这个实验让我们不仅可以构建出一个高可用性和一致性的分布式键值存储系统，还让我们都深入理解了分布式系统的设计与实施以及面对不同问题的应对方法。这门分布式实验课程算是 over 了，希望在比 lab4 更加复杂的挑战时，这种解决问题的勇气依然能够继续激励我们去迎接并克服这些它们。