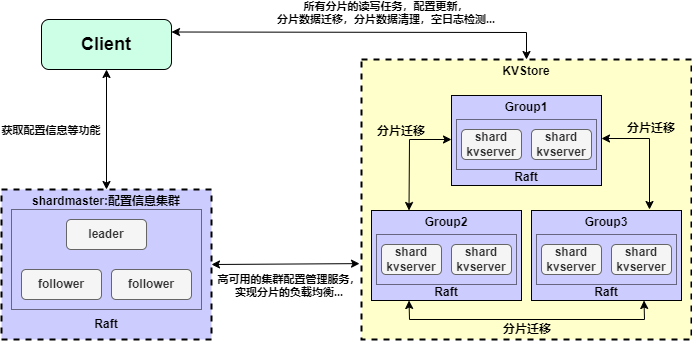
**[mit 6.824] lab4：ShardKV**

lab4 的内容是要在 lab2 的基础上实现一个 multi-raft 的 KV 存储服务，同时也要支持切片在不同 raft 组上的动态迁移而不违背线性一致性，不过其不需要实现集群的动态伸缩。

shard表示互不相交并且组成完整数据库的每一个数据库子集。group表示server的集合，包含一个或多个server。一个shard只可属于一个group，一个group可包含(管理)多个shard。

* lab4A实现ShardMaster服务，作用：提供高可用的集群配置管理服务，实现分片的负载均衡，并尽可能少地移动分片。
* lab4B实现ShardKVServer服务，ShardKVServer则需要实现所有分片的读写任务，需要具备配置更新，分片数据迁移，分片数据清理等功能。

整体架构如下：



**1 shardmaster**

shardmaster记录了每组（Group）ShardKVServer的集群信息和每个分片（shard）服务于哪组（Group）ShardKVServer。具体实现通过Raft维护 一个Configs数组，具体内容如下：

* Num：config number
* Shards：shard -> gid，分片位置信息
* Groups：gid -> servers[],集群成员信息

需要实现四个功能，Join, Leave, Move, Query 服务。

* Query： 查询最新的Config信息。
* Move 将数据库子集Shard分配给GID的Group。
* Join： 新加入的Group信息，要求在每一个group平衡分布shard，即任意两个group之间的shard数目相差不能为1，具体实现每一次找出含有shard数目最多的和最少的，最多的给最少的一个，循环直到满足条件为止。注意：GID = 0 是无效配置，一开始所有分片分配给GID=0，需要优先分配；map的迭代时无序的，不确定顺序的话，同一个命令在不同节点上计算出来的新配置不一致，按sort排序之后遍历即可。且 map 是引用对象，需要用深拷贝做复制。
* Leave： 移除Group，同样别忘记实现均衡，将移除的Group的shard每一次分配给数目最小的Group就行，如果全部删除，别忘记将shard置为无效的0。

基本上就是把lab3的代码抄过来。clerk定义如下：

|  |
| --- |
| Go type Clerk struct {  servers []\*labrpc.ClientEnd  mu sync.Mutex  clientId int64  commandId int64 } |

几个主要的函数如下：

|  |
| --- |
| Go func (sm \*ShardMaster) MakeNextConfig() Config {  //更新config配置  //println("Make Next Config")  curconf := sm.configs[len(sm.configs)-1]  newconf := Config{  Num: curconf.Num + 1,  Shards: curconf.Shards,  Groups: make(map[int][]string),  }  for k, v := range curconf.Groups {  newconf.Groups[k] = v  }  return newconf }  func (sm \*ShardMaster) QueryExec(op Op) (Config, Err) {  if op.Num < 0 || op.Num >= len(sm.configs) {  return sm.configs[len(sm.configs)-1], OK  }  return sm.configs[op.Num], OK }  func (sm \*ShardMaster) MoveExec(op Op) Err {  newconf := sm.MakeNextConfig()  newconf.Shards[op.Shard] = op.GID  sm.configs = append(sm.configs, newconf)  return OK }  func (sm \*ShardMaster) LeaveExec(op Op) Err {  newconf := sm.MakeNextConfig()  //find the remaining group  staygid := 0  for kgid := range newconf.Groups {  canstay := true  for \_, gid := range op.GIDs {  if gid == kgid {  canstay = false  break  }  }  if canstay {  staygid = kgid  break  }  }  //移除Group  for \_, gid := range op.GIDs {  for i := 0; i < len(newconf.Shards); i++ {  if newconf.Shards[i] == gid {  newconf.Shards[i] = staygid  }  }  delete(newconf.Groups, gid)   }  sm.ReBalanceGroupShards(&newconf)  sm.configs = append(sm.configs, newconf)   return OK }  func (sm \*ShardMaster) JoinExec(op Op) Err {  newconf := sm.MakeNextConfig()  for gid, servers := range op.Servers {  newconf.Groups[gid] = servers  for i := 0; i < len(newconf.Shards); i++ {  if newconf.Shards[i] == 0 {  newconf.Shards[i] = gid  }  }  }  sm.ReBalanceGroupShards(&newconf)  sm.configs = append(sm.configs, newconf)   return OK }  func (sm \*ShardMaster) ReBalanceGroupShards(conf \*Config) {  //should move as few shards as possible  //fmt.Println("Do rebalance...")  serverconf := make(map[int]int)  serverconfmap := make(map[int][]int)  for k := range conf.Groups {  serverconf[k] = 0  }  for i := 0; i < len(conf.Shards); i++ {  gid := conf.Shards[i]  serverconf[gid]++  if \_, ok := serverconfmap[gid]; !ok {  serverconfmap[gid] = make([]int, 0)  }  serverconfmap[gid] = append(serverconfmap[gid], i)  }  for {  //将最多的移向最少的，直到每个之间相差不到1  maxnum, maxgid := 0, 0  minnum, mingid := INT\_MAX, 0  for gid, shardnums := range serverconf {  if maxnum < shardnums {  maxnum = shardnums  maxgid = gid  }  if minnum > shardnums {  minnum = shardnums  mingid = gid  }  }  if maxnum-minnum <= 1 {  break  }  serverconf[maxgid]--  serverconf[mingid]++  serverconfmap[mingid] = append(serverconfmap[mingid], serverconfmap[maxgid][len(serverconfmap[maxgid])-1])  serverconfmap[maxgid] = serverconfmap[maxgid][:len(serverconfmap[maxgid])-1]  }  for gid, shards := range serverconfmap {  for \_, shard := range shards {  conf.Shards[shard] = gid  }  }  } |

**2 shardkv**

shardkv的运行方式如下：一开始系统会创建一个 shardmaster 组来负责配置更新，分片分配等任务，接着系统会创建多个 raft 组来承载所有分片的读写任务。此外，raft 组增删，节点宕机，节点重启，网络分区等各种情况都可能会出现。

对于集群内部，我们需要保证所有分片能够较为均匀的分配在所有 raft 组上，还需要能够支持动态迁移和容错。

对于集群外部，我们需要向用户保证整个集群表现的像一个永远不会挂的单节点 KV 服务一样，即具有线性一致性。

challenge1 要求及时清理不再属于本分片的数据，分片迁移成功之后，立马进行分片 GC 了，GC 完毕后再进入到配置更新阶段。challenge2 不仅要求分片迁移时不影响未迁移分片的读写服务，还要求不同地分片数据能够独立迁移。分片迁移以 group 为单位，这样即使一个 group挂了，也不会影响到另一个 group中的分片迁移。

**实验提示**

* 服务器将需要定期轮询 shardmaster 以监听新的配置。预期大约每100毫秒轮询一次；可以更频繁，但过少可能会导致 bug。
* 服务器需要互相发送rpc，以便在配置更改期间传输分片。shardctrler的Config结构包含服务器名，一个 Server 需要一个labrpc.ClientEnd，以便发送RPC。使用make\_end()函数传给StartServer()函数将服务器名转换为ClientEnd。shardkv /client.go需要实现这些逻辑。
* 重配置需要按流程执行唯一一次
* 当服务器转移到新配置后，它可以继续存储它不再负责的分片（生产环境中这是不允许的），但这个可以简化实现
* 你可以在整个 rpc 请求或回复中发送整个 map，这可以简化分片传输
* map 是引用类型，所以在发送 map 的时候，建议先拷贝一次，避免 data race（在 labrpc 框架下，接收 map 时也需要拷贝）
* 在配置更改期间，一对组可能需要互相传送分片，这可能会发生死锁

**Clerk**

主要实现为请求逻辑：

1. 使用key2shard()去找到一个 key 对应哪个分片Shard；
2. 根据Shard从当前配置config中获取的 gid；
3. 根据gid从当前配置config中获取 group 信息；
4. 在group循环查找leaderId，直到返回请求成功、ErrWrongGroup或整个 group 都遍历请求过；
5. Query 最新的配置，回到步骤1循环重复；

|  |
| --- |
| Go type Clerk struct {  sm \*shardmaster.Clerk  config shardmaster.Config  make\_end func(string) \*labrpc.ClientEnd  // You will have to modify this struct.  mu sync.Mutex  clientId int64  commandId int64 } |

**Server**

* 客户端首先和ShardMaster交互，获取最新的配置，根据最新配置找到对应key的shard，请求该shard的group。
* 服务端ShardKVServer会创建多个 raft 组来承载所有分片的读写任务。
* 服务端ShardKVServer需要定期和ShardMaster交互，保证更新到最新配置(monitor)。
* 服务端ShardKVServer需要根据最新配置完成配置更新，分片数据迁移，分片数据清理。

Shardkv server定义如下：

|  |
| --- |
| Go type ShardKV struct {  mu sync.Mutex  me int  rf \*raft.Raft //创建多个 raft 组来承载所有分片的读写任务  applyCh chan raft.ApplyMsg  make\_end func(string) \*labrpc.ClientEnd  gid int  masters []\*labrpc.ClientEnd  maxraftstate int // snapshot if log grows this big  statemachine [shardmaster.NShards]KVmemory //存储  resultCh map[int]chan Res // logindex对应位置的结果  lastopack map[int64]int64 // 记录一个 client 已经处理过的最大 requestId  config shardmaster.Config //配置,定期和Shardmaster交互，保证更新到最新配置(monitor)  mck \*shardmaster.Clerk //clerk } |

基本的Get、PutAppend、SubmitCommand和lab3 kvraft基本一致。主要新增的部分在配置更新、分片迁移和垃圾回收。

**配置更新**

配置更新服务需要不断循环检测最新配置，需要检测当前配置序号，更新最新配置。GetReConfigOp函数是获取分片迁移数据。

|  |
| --- |
| Go func (kv \*ShardKV) UpdateConfig() {  for {  if \_, isleader := kv.rf.GetState(); isleader {  //I am the leader,update the config  // get the latestconfig  lastconfig := kv.mck.Query(-1)  for i := kv.config.Num + 1; i <= lastconfig.Num; i++ {  nextconfig := kv.mck.Query(i)  op, ok := kv.GetReConfigOp(nextconfig)  if !ok {  break  }  res := kv.SubmitCommand(op)  if res.OK == false {  break  }  }  }  //kv.PrintCurConfig()  time.Sleep(TimeoutConfigUpdate)  } }  func (kv \*ShardKV) GetReConfigOp(nextconfig shardmaster.Config) (Op, bool) {  //update config: config-->nextconfig  op := Op{  Command: CommandUpdateConfig,  Config: nextconfig,  LastOpAck: make(map[int64]int64),  }  for i := 0; i < shardmaster.NShards; i++ {  op.Data[i] = KVmemory{  Store: make(map[string]string),  }  }  //分片数据迁移  var wg sync.WaitGroup  var ackmu sync.Mutex  ok := true  shardToTransfer := kv.GetShardsToTransfer(nextconfig)   for gid, shardids := range shardToTransfer {  wg.Add(1)  //eg, A ack B for shard1, get the shard data and insert them into op.Data  args := TransferShardArgs{  ShardIds: shardids,  Num: nextconfig.Num,  }  go func(gid int, args TransferShardArgs, reply TransferShardReply) {  defer wg.Done()  if kv.sendTransferShard(gid, &args, &reply) {  ackmu.Lock()  //deep copy  for \_, sharid := range args.ShardIds {  for k, v := range reply.Data[sharid].Store {  op.Data[sharid].Store[k] = v  }  }  for k := range reply.LastOpAck {  if \_, ok := op.LastOpAck[k]; !ok || op.LastOpAck[k] < reply.LastOpAck[k] {  op.LastOpAck[k] = reply.LastOpAck[k]  }  }   ackmu.Unlock()  } else {  ok = false  }  }(gid, args, TransferShardReply{})  }  wg.Wait()  return op, ok } |

**分片迁移**

分片迁移就是当server可以进行迁移（比如没有处理request），仅可由 leader 处理该请求，其次如果发现请求中的配置版本大于本地的版本，那说明请求拉取的是未来的数据，则返回 ErrNotReady 让其稍后重试，否则将分片数据和去重表都深度拷贝到 reply 即可。

在applyupdateconfig时，还需分片清理协程检测并尝试删除远端的分片。

|  |
| --- |
| Go // if this server is ready, copy shards required in args and ack to reply. func (kv \*ShardKV) TransferShard(args \*TransferShardArgs, reply \*TransferShardReply) {  kv.mu.Lock()  defer kv.mu.Unlock()   if kv.config.Num < args.Num {  // this server is not ready (may still handle the requested shards).  reply.Err = ErrNotReady  return  }   //copy shard to reply  for i := 0; i < shardmaster.NShards; i++ {  reply.Data[i] = KVmemory{  make(map[string]string),  }  }  for \_, shardid := range args.ShardIds {  for k, v := range kv.statemachine[shardid].Store {  reply.Data[shardid].Store[k] = v  }  }   reply.LastOpAck = make(map[int64]int64)  for k, v := range kv.lastopack {  reply.LastOpAck[k] = v  }   reply.Err = OK }  func (kv \*ShardKV) sendTransferShard(gid int, args \*TransferShardArgs, reply \*TransferShardReply) bool {  if servers, ok := kv.config.Groups[gid]; ok {  for si := 0; si < len(servers); si++ {  srv := kv.make\_end(servers[si])  cancall := srv.Call("ShardKV.TransferShard", args, reply)  if cancall {  if reply.Err == OK {  return true  }  if reply.Err == ErrNotReady {  return false  }  }  }  }  return true }  func (kv \*ShardKV) ApplyUpdateConfig(op Op, res \*Res) {  res.ConfigNum = op.Config.Num  if op.Config.Num == kv.config.Num+1 {  //copy the data  for shardid := 0; shardid < shardmaster.NShards; shardid++ {  for k, v := range op.Data[shardid].Store {  kv.statemachine[shardid].Store[k] = v  }  }   for k, v := range op.LastOpAck {  if \_, ok := kv.lastopack[k]; !ok || kv.lastopack[k] < v {  kv.lastopack[k] = v  }  }   lastcfg := kv.config  kv.config = op.Config  //use lastconf do garbage collection  for shardid, sharddata := range op.Data {  if len(sharddata.Store) > 0 {  lastshard2gid := lastcfg.Shards[shardid]  gcargs := GarbageCollectionArgs{  ShardId: shardid,  Num: lastcfg.Num,  }  go kv.sendGC(lastshard2gid, lastcfg, &gcargs, &GarbageCollectionReply{})  }  }  }  res.Err = OK } |

**垃圾回收**

利用 lastConfig 计算出对应 raft 组的 gid 和要拉取的分片，然后并行地去删除分片。仅可由 leader 处理该请求，其次如果发现请求中的配置版本小于本地的版本，那说明该请求已经执行过，否则本地的 config 也无法增大，此时直接返回 OK 即可，否则在本地提交一个删除分片的日志。

|  |
| --- |
| Go func (kv \*ShardKV) GC(args \*GarbageCollectionArgs, reply \*GarbageCollectionReply) {  if \_, isleader := kv.rf.GetState(); !isleader {  reply.WrongLeader = true  return  }   if kv.config.Num < args.Num {  // this server is not ready (may still handle the requested shards).  reply.Err = ErrNotReady  reply.WrongLeader = false  return  }   op := Op{  Command: CommandGC,  NumCfg: args.Num,  SharId: args.ShardId,  }  kv.SubmitCommand(op)   reply.WrongLeader = false  reply.Err = OK }  func (kv \*ShardKV) sendGC(gid int, lastcfg shardmaster.Config, args \*GarbageCollectionArgs, reply \*GarbageCollectionReply) bool {  if servers, ok := lastcfg.Groups[gid]; ok {  for si := 0; si < len(servers); si++ {  srv := kv.make\_end(servers[si])  cancall := srv.Call("ShardKV.GC", args, reply)  if cancall {  if reply.Err == OK {  return true  }  if reply.Err == ErrNotReady {  return false  }  }  }  }  return true }  func (kv \*ShardKV) ApplyGC(op Op, res \*Res) {  if op.NumCfg > kv.config.Num {  return  }  if kv.gid != kv.config.Shards[op.SharId] {  kv.statemachine[op.SharId] = KVmemory{  Store: make(map[string]string),  }  } } |

**3 总结**

在本课程中，我循序渐进，从raft到单机kv存储，再到分片kv存储，通过设计和实现一个分布式系统，从而应用课程中学到的概念和技术，收获良多。从本课程中学到了较多的内容，涵盖了分布式系统的基本概念、一致性、容错性、并发性、分布式存储和分布式计算等关键主题。同时，我也推荐《ddia》和《深入理解分布式系统》。通过这门课的学习，让我也进一步掌握分布式系统工程领域的知识和技能。接下来的话可以看看tinykv以及6.824-2015中paxos的实现。