Lab4B

前言

要求在 Lab 4A 的键值存储服务（RaftKV）基础上添加快照（Snapshot）支持，以优化 Raft 日志的空间使用和重启恢复时间。以下是对 Lab 4B 任务的详细解释，涵盖目标、实现要点和需要解决的问题。

Lab 4B 的任务目标

Lab 4B 的核心目标是让 `KVServer` 与 Raft 的快照机制协作，减少持久化日志的大小并加速服务器重启时的状态恢复。具体任务包括：

1. **检测 Raft 状态大小并触发快照**：
   - 当 Raft 的持久化状态（日志等）接近 `maxraftstate` 阈值时，生成快照并调用 Raft 的 `Snapshot()` 方法。
2. **保存和恢复快照**：
   - 将键值服务的状态保存到快照中，通过 `persister.Save()` 存储。
   - 重启时从 `persister.ReadSnapshot()` 读取快照，恢复服务状态。
3. **保持去重能力**：
   - 确保快照后仍能检测重复操作（例如客户端重试的命令）。
4. **性能要求**：
   - 测试运行时间需控制在 400 秒（实际时间）和 700 秒（CPU 时间）内，`TestSnapshotSize` 小于 20 秒。

实现要点

以下是实现 Lab 4B 的关键步骤和需要修改的组件：

1. 检测 Raft 状态大小

- **输入**：
  - `maxraftstate`：在 `StartKVServer()` 中传入，表示 Raft 状态的最大字节数（包括日志，不包括快照）。
  - 如果 `maxraftstate == -1`，无需快照。
- **检查方法**：
  - 使用 `persister.RaftStateSize()` 获取当前 Raft 状态的字节大小。
  - 在 `KVServer` 中定期比较 `RaftStateSize()` 与 `maxraftstate`。
- **触发时机**：
  - 在 `applier()` 或其他合适位置（例如每次应用日志后），检查状态大小。
  - 如果接近阈值（例如 `RaftStateSize() >= maxraftstate * 0.9`），生成快照。

2. 生成快照

- **快照内容**：
  - **状态机状态**：`KVStateMachine` 的当前键值对（例如 `MemoryKV.KV`）。
  - **去重信息**：`lastOperations`（每个客户端的最新操作记录），用于检测重复命令。
  - **最后应用索引**：`lastApplied`，标记已应用的日志位置。
- **字段要求**：

- 快照中存储的结构体字段必须全部大写（例如 `KV`、`LastOperations`、`LastApplied`）。

代码实现

applier

对于Kvserver的快照，在applier函数中对其进行是实现

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                    DPrintf("{Node %v} discards outdated message %v because a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

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

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

```
                //检查是否需要触发KV快照
                if kv.needSnapshot() {
                    kv.takeSnapshot(message.CommandIndex)
                }
                kv.mu.Unlock()
            } else if message.SnapshotValid { //如果消息种携带快照
                kv.mu.Lock()
                //将快照应用到raft上，且从快照中恢复kvserver的状态
                if kv.rf.CondInstallSnapshot(message.SnapshotTerm,
message.SnapshotIndex, message.Snapshot) {
                    kv.restoreStateFromSnapshot(message.Snapshot)
                    kv.lastApplied = message.SnapshotIndex
                }
                kv.mu.Unlock()
            } else {
                panic(fmt.Sprintf("Invalid ApplyMsg %v", message))
            }
        }
    }
}
```

needSnapShot

如果在applier中，检查如果日志条目比当前的kv要求的最大长度还大则进行快照处理

```
// needSnapshot 检查是否需要触发KV快照
func (kv *KVServer) needSnapshot() bool {
    return kv.maxraftstate != -1 && kv.rf.GetRaftStateSize() >= kv.maxraftstate
}
```

takeSnapshot

将kvServer打包成存入

```
// takeSnapshot 生成kv快照
func (kv *KVServer) takeSnapshot(index int) {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(kv.stateMachine)
    e.Encode(kv.lastOperations)
    data := w.Bytes()
    kv.rf.Snapshot(index, data)
}
```

对之前的代码进行了改进

```
//src/raft/raft.go
// applier 应用日志条目
func (rf *Raft) applier() {
    for !rf.killed() {
        rf.mu.Lock()
        // 检查commitIndex是否可用
        if rf.commitIndex <= rf.lastApplied {
            //需要等待commitIndex被推进
            rf.applyCond.Wait()
```

```
        }
        // 应用日志条目到状态机
        firstLogIndex, commitIndex, lastApplied := rf.getFirstlog().Index,
rf.commitIndex, rf.lastApplied
        entries := make([]LogEntry, commitIndex-lastApplied)
        copy(entries, rf.logs[lastApplied-firstLogIndex+1:commitIndex-
firstLogIndex+1])
        rf.mu.Unlock()

        //将申请消息发送到applyCh以获取服务/状态机副本
        for _, entry := range entries {
            rf.applych <- ApplyMsg{
                CommandValid: true,
                Command:      entry.Command,
                CommandIndex: entry.Index,
                CommandTerm:  entry.Term,
            }
        }
        rf.mu.Lock()
        DPrintf("{Node %v} applies log entries from index %v to %v in term %v",
rf.me, lastApplied+1, commitIndex, rf.currentTerm)
        // 使用commitIndex与rf.commitIndex最大值而不是commitIndex，因为rf.commitIndex
可能在Unlock（）和Lock（）期间发生变化。
        //  如果直接赋值 rf.lastApplied = commitIndex，
        // 而 rf.commitIndex 在解锁期间减小（例如由于某种异常或竞争条件）
        // f.lastApplied 可能会被设置为一个较小的值，导致状态回退。
        rf.lastApplied = Max(rf.lastApplied, commitIndex)
        rf.mu.Unlock()
    }
}
```

测试

```
go test -run 4B
Test: InstallSnapshot RPC (4B) ...
  ... Passed --   3.5  3 24388   63
Test: snapshot size is reasonable (4B) ...
  ... Passed --   1.4  3 22093  800
Test: ops complete fast enough (4B) ...
  ... Passed --   1.7  3 26513    0
Test: restarts, snapshots, one client (4B) ...
info: linearizability check timed out, assuming history is ok
  ... Passed --  23.2  5 237328 34697
Test: restarts, snapshots, many clients (4B) ...
  ... Passed --  21.8  5 494187 49509
Test: unreliable net, snapshots, many clients (4B) ...
  ... Passed --  15.8  5  7459 1278
Test: unreliable net, restarts, snapshots, many clients (4B) ...
  ... Passed --  23.9  5  9878 1357
Test: unreliable net, restarts, partitions, snapshots, many clients (4B) ...
  ... Passed --  30.0  5  6777  740
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients
(4B) ...
  ... Passed --  32.0  7 18579 1976
PASS
ok      6.5840/kvraft   153.454s
```