Lab 3D

Lab 3D 的主题是 **日志压缩（Log Compaction）**，旨在解决 Raft 服务器在长期运行中日志无限增长的问题。通过引入快照（Snapshot）机制，Raft 可以丢弃早期的日志条目，从而减少持久化数据量并加速重启。以下是主要内容和任务的总结：

方案

- **快照机制**：
  - 服务层定期生成状态快照（Snapshot），并通知 Raft。
  - Raft 丢弃快照之前的日志，只保留快照之后的日志尾部。
- **安装快照 RPC（InstallSnapshot RPC）**：
  - 当 Follower 落后太多，Leader 已丢弃其所需的日志时，Leader 发送快照给 Follower。
  - Follower 通过 `applyCh` 将快照应用到服务层，更新状态。

任务

1. **实现 `Snapshot(index int, snapshot []byte)` 函数**：
   - 服务层调用此函数，提供状态快照和对应的最高日志索引（`index`）。
   - Raft 丢弃 `index` 之前的日志，更新日志起点（例如通过变量 `X` 表示）。
   - 确保丢弃的日志不再被引用，以便 Go 垃圾回收器释放内存。
2. **实现 `InstallSnapshot` RPC**：
   - Leader 检测 Follower 所需的日志已被丢弃时，发送快照。
   - Follower 接收快照后，通过 `applyCh` 发送 `ApplyMsg`（包含快照数据）给服务层。
   - 确保快照只推进服务状态，不回退。
3. **持久化快照和 Raft 状态**：
   - 使用 `persister.Save(state, snapshot)` 保存 Raft 状态和快照。
   - 重启时，从 `Persister` 读取快照和状态，恢复 Raft 和服务状态。
4. **修改 Raft 操作以支持裁剪日志**：
   - Raft 只存储从某个索引（`X`）开始的日志尾部。
   - 更新日志索引计算（如 `getFirstLog()`、`getLastLog()`）以适应裁剪后的日志。

实现步骤建议

1. **支持裁剪日志**：
   - 修改 Raft 结构体，添加 `snapshotIndex`（或类似变量）表示快照覆盖的最高日志索引。
   - 调整日志数组，只存储 `snapshotIndex` 之后的条目。
   - 测试 3B/3C，确保基础功能仍正常（初始 `snapshotIndex = 0`）。
2. **实现 `Snapshot()`**：
   - 丢弃 `index` 之前的日志，更新 `snapshotIndex = index`。
   - 保存快照到 `Persister`。
   - 通过第一个 3D 测试。
3. **实现 `InstallSnapshot` RPC**：
   - Leader 在 `nextIndex[peer]` 小于日志起点时发送快照。
   - Follower 接收后应用快照并更新状态。

总结

- **实现日志压缩**：通过快照机制丢弃旧日志，减少存储需求并加速重启。
- **支持落后 Follower 的恢复**：通过 `InstallSnapshot` RPC 让 Leader 将快照发送给落后太多的 Follower。
- **确保正确性**：在裁剪日志后，Raft 仍能正常运行并保持一致性，同时正确持久化和恢复快照。

代码实现

Snapshot实现

主要作用：接受一个快照，修剪 Raft 日志（丢弃 `index` 及之前的日志条目），并将状态持久化。

```
func (rf *Raft) Snapshot(index int, snapshot []byte) {
    // Your code here (3D).
    rf.mu.Lock()
    defer rf.mu.Unlock()

    snapshotIndex := rf.getFirstlog().Index
    if index <= snapshotIndex || index > rf.getLastlog().Index {
        DPrintf("{Node %v} rejects replacing log with snapshotIndex %v as
current snapshotIndex %v is larger in term %v", rf.me, index, snapshotIndex,
rf.currentTerm)
        return
    }

    rf.logs = shrinkEntries(rf.logs[index-snapshotIndex:])
    rf.logs[0].Command = nil
    rf.persister.Save(rf.encodeState(), snapshot)

    DPrintf("{Node %v}'s state is {state %v,term %v,commitIndex %v,lastApplied
%v,firstLog %v,lastLog %v} after accepting the snapshot with index %v", rf.me,
rf.state, rf.currentTerm, rf.commitIndex, rf.lastApplied, rf.getFirstlog(),
rf.getLastlog(), index)

}
```

同时要修改persister.go中的Save函数，如果快照为nil的话，在测试install snapshots (disconnect)以及后面的都会出错。

```
// Save both Raft state and K/V snapshot as a single atomic action,
// to help avoid them getting out of sync.
func (ps *Persister) Save(raftstate []byte, snapshot []byte) {
    ps.mu.Lock()
    defer ps.mu.Unlock()
    ps.raftstate = clone(raftstate)
    if snapshot != nil {
        ps.snapshot = clone(snapshot)
    }
```

CondInstallSnapshot实现

```
// CondInstallSnapshot 用来peer判断leader发过来的快照是否满足条件，如果满足，则安装快照
func (rf *Raft) CondInstallSnapshot(lastIncludedTerm int, lastIncludedIndex int,
snapshot []byte) bool {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    //过期快照
    if lastIncludedIndex <= rf.commitIndex {
        DPrintf("{Node %v} rejects outdated snapshot with lastIncludeIndex %v as
current commitIndex %v is larger in term %v", rf.me, lastIncludedIndex,
rf.commitIndex, rf.currentTerm)
        return false
    }

    if lastIncludedIndex > rf.getLastlog().Index {
        rf.logs = make([]LogEntry, 1)
    } else {
        rf.logs = shrinkEntries(rf.logs[lastIncludedIndex-
rf.getFirstlog().Index:])
    }

    rf.logs[0].Term, rf.logs[0].Index = lastIncludedTerm, lastIncludedIndex
    rf.commitIndex, rf.lastApplied = lastIncludedIndex, lastIncludedIndex

    rf.persister.Save(rf.encodeState(), snapshot)

    DPrintf("{Node %v}'s state is {state %v,term %v,commitIndex %v,lastApplied
%v,firstLog %v,lastLog %v} after accepting the snapshot which lastIncludedTerm is
%v, lastIncludedIndex is %v", rf.me, rf.state, rf.currentTerm, rf.commitIndex,
rf.lastApplied, rf.getFirstlog(), rf.getLastlog(), lastIncludedTerm,
lastIncludedIndex)
    return true
}
```

同时修改config.go中的applierSnap函数

该函数在一个独立的 goroutine 中运行，监听 `applyCh` 通道，处理提交的日志条目（`CommandValid`）或快照（`SnapshotValid`），并定期生成快照。

```
// periodically snapshot raft state
func (cfg *config) applierSnap(i int, applyCh chan ApplyMsg) {
    cfg.mu.Lock()
    rf := cfg.rafts[i]
    cfg.mu.Unlock()
    if rf == nil {
        return // ???
    }

    for m := range applyCh {
        err_msg := ""
        if m.SnapshotValid {
            cfg.mu.Lock()
            if rf.CondInstallSnapshot(m.SnapshotTerm, m.SnapshotIndex,
m.Snapshot) {
```

```
                err_msg = cfg.ingestSnap(i, m.Snapshot, m.SnapshotIndex)
            }
            cfg.mu.Unlock()
        } else if m.CommandValid {
            if m.CommandIndex != cfg.lastApplied[i]+1 {
                err_msg = fmt.Sprintf("server %v apply out of order, expected
index %v, got %v", i, cfg.lastApplied[i]+1, m.CommandIndex)
            }

            if err_msg == "" {
                cfg.mu.Lock()
                var prevok bool
                err_msg, prevok = cfg.checkLogs(i, m)
                cfg.mu.Unlock()
                if m.CommandIndex > 1 && prevok == false {
                    err_msg = fmt.Sprintf("server %v apply out of order %v", i,
m.CommandIndex)
                }
            }

            cfg.mu.Lock()
            cfg.lastApplied[i] = m.CommandIndex
            cfg.mu.Unlock()

            if (m.CommandIndex+1)%SnapShotInterval == 0 {
                w := new(bytes.Buffer)
                e := labgob.NewEncoder(w)
                e.Encode(m.CommandIndex)
                var xlog []interface{}
                for j := 0; j <= m.CommandIndex; j++ {
                    xlog = append(xlog, cfg.logs[i][j])
                }
                e.Encode(xlog)
                rf.Snapshot(m.CommandIndex, w.Bytes())
            }
        } else {
            // Ignore other types of ApplyMsg.
        }
        if err_msg != "" {
            log.Fatalf("apply error: %v", err_msg)
            cfg.applyErr[i] = err_msg
            // keep reading after error so that Raft doesn't block
            // holding locks...
        }
    }
}
```

实现快照RPC

首先修改 ApplyMsg结构体

```
type ApplyMsg struct {
    CommandValid bool
    Command      interface{}
    CommandIndex int

    // For 3D:
    SnapshotValid bool
    Snapshot      []byte
    SnapshotTerm  int
    SnapshotIndex int
}
```

快照RPC参数和实际处理函数

```
type InstallSnapshotArgs struct {
    Term              int
    LeaderId          int
    LastIncludedIndex int
    LastIncludedTerm  int
    Data              []byte

    // unused fields
    // Offset int   // byte offset where chunk is positioned in the snapshot
file
    // Done   bool  // true if this is the last chunk
}

type InstallSnapshotReply struct {
    Term int
}

// genInstallSnapshotArgs 产生快照
func (rf *Raft) genInstallSnapshotArgs() *InstallSnapshotArgs {
    firstLog := rf.getFirstlog()
    args := &InstallSnapshotArgs{
        Term:              rf.currentTerm,
        LeaderId:          rf.me,
        LastIncludedIndex: firstLog.Index,
        LastIncludedTerm:  firstLog.Term,
        Data:              rf.persister.ReadSnapshot(),
    }
    return args
}

// InstallSnapshot 处理快照RPC
func (rf *Raft) InstallSnapshot(args *InstallSnapshotArgs, reply
*InstallSnapshotReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer DPrintf("{Node %v}'s state is {state %v, term %v}} after processing
InstallSnapshot,  InstallSnapshotArgs %v and InstallSnapshotReply %v ", rf.me,
rf.state, rf.currentTerm, args, reply)

    reply.Term = rf.currentTerm
    if rf.currentTerm > args.Term {
        return
```

```
    }

    if rf.currentTerm < args.Term {
        rf.currentTerm, rf.votedFor = args.Term, -1
        rf.persist()
    }
    rf.ChangeState(Follower)
    rf.electionTimer.Reset(RandomElectionTimeout())

    //查看快照是否比当前新
    if args.LastIncludedIndex <= rf.commitIndex {
        //Raft 要求快照只推进状态，不回退。如果快照的索引 ≤ 已提交索引
        //应用它可能会导致状态回退，违反 Lab 3D 的要求：“快照不应使服务状态回退”
        return
    }
    //这里发送给applych 后再config.go中继续判断是否符合快照应用，并在CondInstallSnapshot
进行状态修改
    go func() {
        rf.applych <- ApplyMsg{

            SnapshotValid: true,
            Snapshot:      args.Data,
            SnapshotTerm:  args.LastIncludedTerm,
            SnapshotIndex: args.LastIncludedIndex,
        }
    }()

}

// sendInstallSnapshot 发送快照RPC
func (rf *Raft) sendInstallSnapshot(peer int, args *InstallSnapshotArgs, reply
*InstallSnapshotReply) bool {
    ok := rf.peers[peer].Call("Raft.InstallSnapshot", args, reply)
    return ok
}
```

shrinkEntries函数实现

```
// shrinkEntriesArray 会丢弃 entries 切片底层使用的数组，
// 如果这个数组的大部分空间没有被使用的话。这样可以避免保留对一大堆可能很大的、不再需要的条目的
引用。
// 简单地清空 entries 是不安全的，因为客户端可能仍在使用这些条目。
func shrinkEntries(entries []LogEntry) []LogEntry {
    const lenMultiple = 2
    if cap(entries) > len(entries)*lenMultiple {
        newEntries := make([]LogEntry, len(entries))
        copy(newEntries, entries)
        return newEntries
    }
    return entries
}
```

心跳发送逻辑修改

当发现peer的日志比leader的快照后的日志还要旧。那就直接发送leader日志给follower，让其直接复制leader的快照

```
// replicateOnceRound
func (rf *Raft) replicateOnceRound(peer int) {
    rf.mu.RLock()
    if rf.state != Leader {
        rf.mu.RUnlock()
        return
    }
    prevLogIndex := rf.nextIndex[peer] - 1
    if prevLogIndex < rf.getFirstlog().Index { //如果Follow落后太多直接发送快照给他
        //only send InstallSnapshot RPC
        args := rf.genInstallSnapshotArgs()
        rf.mu.RUnlock()
        reply := new(InstallSnapshotReply)
        if rf.sendInstallSnapshot(peer, args, reply) {
            rf.mu.Lock()
            if rf.state == Leader && rf.currentTerm == args.Term {
                if reply.Term > rf.currentTerm {
                    rf.ChangeState(Follower)
                    rf.currentTerm, rf.votedFor = reply.Term, -1
                    rf.persist()
                    // rf.electionTimer.Reset(RandomElectionTimeout())
                } else {
                    rf.nextIndex[peer] = args.LastIncludedIndex + 1
                    rf.matchIndex[peer] = args.LastIncludedIndex
                }
            }
            rf.mu.Unlock()
            DPrintf("{Node %v} sends InstallSnapshotArgs %v to {Node %v} and
receives InstallSnapshotReply %v", rf.me, args, peer, reply)
        }
    } else {
        args := rf.genAppendEntriesArgs(prevLogIndex)
        rf.mu.RUnlock()
        reply := new(AppendEntriesReply)
        if rf.sendAppendEntries(peer, args, reply) {
            rf.mu.Lock()
            //如果rpc 后 还是leader 且周期没变化
            if args.Term == rf.currentTerm && rf.state == Leader {
                if !reply.Success { //日志一致性检查失败
                    if reply.Term > rf.currentTerm { //脑裂，或者已经宕机的leader突然
又活过来  可能发生了网络分区，或者这个 Leader 是一个刚刚恢复但任期落后的旧 Leader
                        //标签当前服务器 已经过时了，重新变成follower
                        rf.ChangeState(Follower)
                        rf.currentTerm, rf.votedFor = reply.Term, -1
                        rf.persist()
                    } else if reply.Term == rf.currentTerm { //说明follow的周期和
leader周期一致，说明是条目出了问题
                        // //减少nextIndex并重试
                        // rf.nextIndex[peer] = reply.ConfictIndex
                        // // TODO: optimize the nextIndex finding, maybe use
binary search
                        // if reply.ConfictTerm != -1 {
```

```
                        //  firstLogIndex := rf.getFirstlog().Index
                        //  for index := args.PrevLogIndex; index >=
firstLogIndex; index-- {
                        //      if rf.logs[index-firstLogIndex].Term ==
reply.ConfictTerm {
                        //          rf.nextIndex[peer] = index
                        //          break
                        //      }
                        //  }
                        // }
                        firstLogIndex := rf.getFirstlog().Index
                        if reply.ConfictTerm != -1 {
                            lastIndex := -1
                            for index := args.PrevLogIndex; index >=
firstLogIndex; index-- {
                                if rf.logs[index-firstLogIndex].Term ==
reply.ConfictTerm {
                                    lastIndex = index
                                    break
                                }
                            }
                            if lastIndex != -1 {
                                rf.nextIndex[peer] = lastIndex + 1 // Case 2
Leader 有 XTerm，nextIndex = XTerm 最后一个条目 + 1。
                            } else {
                                rf.nextIndex[peer] = reply.ConfictIndex // Case
1 Leader 无 XTerm，nextIndex = ConfictIndex。
                            }
                        } else {
                            rf.nextIndex[peer] = reply.ConfictIndex // Case 3
Follower 日志太短，nextIndex = XLen。
                        }
                    }
                } else { //日志匹配成功 提交
                    rf.matchIndex[peer] = args.PrevLogIndex + len(args.Entries)
                    rf.nextIndex[peer] = rf.matchIndex[peer] + 1
                    //advance commitIndex if possible
                    rf.advanceCommitIndexForLeader()
                }
            }
            rf.mu.Unlock()
            DPrintf("{Node %v} sends AppendEntriesArgs %v to {Node %v} and
receives AppendEntriesReply %v", rf.me, args, peer, reply)
        }
    }
}
```

打印日志

需要去test_test.go下加上InitLogger()，可以打印日志方便查看bug

```
func snapcommon(t *testing.T, name string, disconnect bool, reliable bool, crash
bool) {

    InitLogger()
    iters := 30
    servers := 3
    cfg := make_config(t, servers, !reliable, true)
    defer cfg.cleanup()
    .......
```

测试

```
 go test -run 3D
Test (3D): snapshots basic ...
  ... Passed --   6.1  3  174   61294  212
Test (3D): install snapshots (disconnect) ...
  ... Passed --  53.9  3 1052  539010  323
Test (3D): install snapshots (disconnect+unreliable) ...
  ... Passed --  64.8  3 1247  592347  324
Test (3D): install snapshots (crash) ...
  ... Passed --  44.5  3  674  425975  288
Test (3D): install snapshots (unreliable+crash) ...
  ... Passed --  52.8  3  785  462227  301
Test (3D): crash and restart all servers ...
  ... Passed --  15.0  3  298   84388   62
Test (3D): snapshot initialization after crash ...
  ... Passed --   5.2  3   78   21162   14
PASS
ok      6.5840/raft     242.407s
```

3C和3D的不同点

- **3C** 只专注于 **持久化Raft状态**，在重启时恢复原始状态。
- **3D** 则专注于 **日志压缩** 和 **快照机制**，目的是减少存储需求并加速节点的恢复速度。

3C: 持久化 (Persistence)

1. `persist()`：
    - 这个函数负责将Raft节点的状态序列化（编码）并保存到 `Persister`。Raft的状态通常包括：当前任期、投票信息、日志信息等。
    - 你可以使用 `labgob` 库来编码状态（`encodeState()`）。`labgob` 是Go中的一种编码方式，能够确保结构体字段名是以大写字母开头（否则会报错）。
2. `readPersist()`：
    - 这个函数负责从 `Persister` 读取保存的Raft状态，并初始化Raft节点的状态。它应恢复Raft的任期、日志信息等，以便节点在重启后能够恢复。
3. **在Raft中插入 `persist()` 调用**：
    - 在Raft的状态发生变化（例如选举新领导者、追加日志条目等）时，调用 `persist()` 保存当前状态。

3D: 日志压缩 (Log Compaction)

1. `Snapshot()`：
    - 这个方法负责生成当前状态的快照，并丢弃所有在快照之前的日志条目。
    - 快照通常包含Raft节点的最新状态（例如任期、日志索引等），并且与应用程序的服务状态（如键值对存储）紧密相关。
2. `InstallSnapshot` **RPC**：
    - 这是Raft协议中的一个RPC，用于将快照传递给滞后的节点。目标是将节点状态更新为最新的快照，而不需要回放过多的日志条目。
3. **持久化快照**：
    - 在Raft重启后，除了恢复Raft状态外，还需要恢复快照。`Persister.Save()` 的第二个参数传递的就是快照数据。

思路

1. 在**3C**中，主要是利用 `Persister` 存储Raft的状态，包括日志和选举信息。
2. 在**3D**中，Raft通过快照的方式减少日志的存储，并确保通过 `InstallSnapshot` RPC 保证滞后节点的快速恢复。
3. **性能优化**：日志压缩不仅能减少存储需求，也能加速重启和恢复过程。

总结

- **3C** 更关注如何在节点重启时恢复到最新状态，而 **3D** 更多地解决了如何优化日志存储和加速恢复。实现这两个部分都需要良好的状态管理和对 `Persister` 的有效使用，尤其是在快照的实现上。