Lab 3C：Raft 持久化总结

Lab 3C 要求您实现 Raft 的持久化机制，确保服务器在重启后能够恢复之前的状态。以下是您需要完成的主要任务：

1. 持久化状态的实现

需要持久化的状态（根据 Raft 论文图 2）：

- `currentTerm` - 当前任期
- `votedFor` - 在当前任期投票给的候选人 ID
- `log[]` - 日志条目数组

实现持久化的步骤：

1. 完成 `persist()` 函数：
   - 使用 `labgob` 编码器将状态序列化为字节数组
   - 调用 `persister.Save(state, nil)` 保存状态
2. 完成 `readPersist()` 函数：
   - 使用 `labgob` 解码从 `persister` 读取的状态
   - 恢复 Raft 的持久状态（`currentTerm`、`votedFor`、`logs`）
3. 在状态变化点调用 `persist()`，主要包括：
   - 修改 `currentTerm` 时
   - 修改 `votedFor` 时
   - 修改 `logs` 时（添加、删除或修改日志条目）

2. 优化日志一致性检查

实现论文第 7-8 页描述的优化，加速日志同步中的冲突恢复：

拒绝消息应包含的附加信息：

- `XTerm`：冲突条目的任期（如果有）
- `XIndex`：该任期的第一个条目索引（如果有）
- `XLen`：Follower 的日志长度

Leader 收到拒绝后的逻辑：

1. 如果 Leader 没有 `XTerm`（Case 1）：
   - 设置 `nextIndex = XIndex`
2. 如果 Leader 有 `XTerm`（Case 2）：
   - 设置 `nextIndex` 为 Leader 日志中该任期的最后一个条目的索引+1
3. 如果 Follower 的日志太短（Case 3）：
   - 设置 `nextIndex = XLen`

   代码修改

```
// replicateOnceRound
func (rf *Raft) replicateOnceRound(peer int) {
    rf.mu.RLock()
    if rf.state != Leader {
```

```
            rf.mu.RUnlock()
            return
        }
        prevLogIndex := rf.nextIndex[peer] - 1
        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()
    }
}
```

3. 代码修改点详解

1. **在 `AppendEntries` RPC 处理函数中**:
    - 当日志不匹配时，提供关于冲突的详细信息（XTerm, XIndex, XLen）
2. **在 `Start()` 函数中**:
    - 添加新日志后调用 `persist()`
3. **在状态变化时**:
    - 选举开始时（修改 `votedFor`）
    - 收到更高任期的 RPC 时（修改 `currentTerm` 和 `votedFor`）
    - 成为 Leader 时（可能会重置 `votedFor`）
    - 接受 AppendEntries 时（修改日志）
4. **在 `replicateOnceRound` 或处理 AppendEntries 响应时**:
    - 实现基于 XTerm, XIndex, XLen 的优化日志回滚逻辑

完成 Lab 3C 后，您的 Raft 实现将能够在服务器重启后恢复正常工作，这是构建可靠分布式系统的重要一步。

实现好后，我们只需要在入口处Make调用readPersist即可，关键需要在什么时候保存状态呢？其实很简单，只需要对我们需要持久化的三个字段修改的时候就进行persist操作。即persist()操作应当在以下几种情况下被触发：

1. 日志条目更新：当有新的日志条目被添加到logs中，或是已有条目被删除或替换时。
2. 任期变更：当currentTerm发生变化，比如在选举期间或接收到更高任期的领导者信息时。
3. 投票行为：当votedFor字段被更新，意味着节点投出了新的一票或取消了之前的投票。

```
// persist 持久化
func (rf *Raft) persist() {
    rf.persister.Save(rf.encodeState(), nil)

}

// restore previously persisted state.
func (rf *Raft) readPersist(data []byte) {
    if data == nil || len(data) < 1 { // bootstrap without any state?
        return
    }
    r := bytes.NewBuffer(data)
    d := labgob.NewDecoder(r)
    var currentTerm, voteFor int
    var logs []LogEntry
```

```
    if d.Decode(&currentTerm) != nil ||
        d.Decode(&voteFor) != nil || d.Decode(&logs) != nil {
        DPrintf("{Node %v} fails to decode persisted state", rf.me)
    } else {
        rf.currentTerm, rf.votedFor, rf.logs = currentTerm, voteFor, logs
        rf.lastApplied, rf.commitIndex = rf.getFirstlog().Index,
rf.getFirstlog().Index
    }
}
func (rf *Raft) encodeState() []byte {
    w := new(bytes.Buffer)
    e := labgob.NewEncoder(w)
    e.Encode(rf.currentTerm)
    e.Encode(rf.votedFor)
    e.Encode(rf.logs)
    return w.Bytes()

}
```

```
go test -run 3C
Test (3C): basic persistence ...
  ... Passed --   8.2  3  103   25033    6
Test (3C): more persistence ...
  ... Passed --  23.0  5  937  206389   16
Test (3C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   4.2  3   31    8017    4
Test (3C): Figure 8 ...
  ... Passed --  29.1  5  412  103885   20
Test (3C): unreliable agreement ...
  ... Passed --   2.4  5  343  119062  246
Test (3C): Figure 8 (unreliable) ...
  ... Passed --  31.3  5 1829 3584853  156
Test (3C): churn ...
  ... Passed --  16.4  5 1442 2297261  862
Test (3C): unreliable churn ...
  ... Passed --  16.6  5 1020 1072129  317
PASS
ok      6.5840/raft     131.206s
```