分布式系统实验报告

简介

本项目在现有框架的基础上实现了Raft算法，并通过了`raft/test_test.go`文件中的所有测试样例（即：完成了实验1、2、3）。具体效果见「实现演示」部分。

流程介绍、分析与设计

Raft算法是一个使多个节点对某一日志达成一致的共识算法（consensus algorithm）。相比传统的Paxo算法，Raft的主要优点在于它的简洁与可理解性。本项目主要基于Ongaro等人介绍Raft算法的拓展版论文[1]（特别是文章中的图表2）完成。下面我们依次介绍本项目中Raft三个模块的具体分析与设计，即leader election、log replication和safety。

Leader Election

Leader选举的核心逻辑是：Follower在长时间没有收到来自Leader发送的心跳RPC后，会转变为Candidate，并开启选举。为了实现的简洁，我们采用了Golang提供的`Timer`机制实现计时。如下列代码所示，计时器每次重置时会随机重置为300~600ms中的某一个值，以尽量保证每次只有一个成员超时并发起选举。另外，我们把心跳间隔设置为100ms，比计时器的下限小200ms，以防止成员在暂时没有收到心跳的情况下发起选举。

```
const (
    ElectionTimeoutFloor = 300
    ElectionTimeoutRange = 300
    HeartbeatInterval    = 100
)

func (rf *Raft) resetTimer() {
    duration :=
time.Duration(ElectionTimeoutFloor+rand.Intn(ElectionTimeoutRange)) *
time.Millisecond

    if rf.timer == nil {
        rf.timer = time.NewTimer(duration)
    } else {
        // 通过timer.Reset来复用计时器，减少资源开销
        rf.timer.Reset(duration)
    }
}
```

需要注意的是，根据[1]中的图表2，当且仅当“收到来自**当前Leader**的AppendEntries RPC”或“**投票**给Candidate”时才应该重置计时器，而不是在每次收到RPC包时都重置[2]。

> If election timeout elapses without receiving AppendEntries RPC **from current leader** or **granting vote** to candidate: convert to candidate.

计时器超时（通过Golang的`select...case...`机制监测`channel`来实现，具体代码见`timerMonitor`方法）后，Follower转变角色为Candidate并启动选举。选举的具体流程是：

1. `currentTerm`加一；
2. `votedFor`置为me，即投票给自己；
3. `votes`置为1，表示自己给自己投的票；

4. 为选举重置计时器，以在选举长时间没有结果时（如发生split votes问题）再次启动选举。
5. 分别给所有其他节点发送Request Vote RPC，请求投票。

RequestVote RPC的args和reply结构如[1]中的图表2所示，此处不再赘述。Candidate填充好args后，调用`sendRequestVote`接口将其发送给其他所有成员，并对返回的reply进行分析。若对方给自己投票了，则将票数加一，并判断是否已经达成票数需求，若已达成则转变为新的Leader。

```
result := rf.sendRequestVote(target, args, &reply)
if result { //首先判断RPC包是否发送成功
    if rf.currentTerm == args.Term && reply.VoteGranted { //其次判断是否收到投票
        // 若收到投票，则票数加1，并判断是否已经满足“大多数”
        rf.votes++
        if rf.votes >= len(rf.peers)/2+1 && rf.roleState == Candidate {
            // 满足“大多数”则转为Leader
            rf.changeRoleState(Leader)
        }
    }
}
```

若长时间没有新的Leader产生（即：要么收到其他新Leader的心跳RPC，要么自己成为新Leader），选举开始时设定的定时器会超时，Candidate进入下个term，重新启动选举。这样的策略防止系统因为一直没有新Leader而宕机。

需要注意的是，RequestVote RPC（AppendEntries RPC同理）的args和reply均包含一个称为`term`的域，表明发送这个RPC（或其回复RPC）的成员的`currentTerm`。当接收方发现对方的`term`比自己的高时，应该更新自己的`term`，并转变为Follower。即[1]中图表2所说的

> If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

关于选举流程中与safety相关的逻辑，我们将在后文中介绍。

Log Replication

Leader被选出后，会启动一个新的goroutine，用于每隔100ms发送一个AppendEntries RPC给所有其他成员。若当前没有新的log需要发送，则该RPC不含任何payload，仅作为心跳用于维持每个成员的当前角色。

```
// changeRoleState用于角色状态装换，此处主要介绍它转换为Leader时执行的一些动作
func (rf *Raft) changeRoleState(newRoleState int) {
    // do something for differnet newRoleState...
    // ...
    } else if newRoleState == Leader {
        rf.roleState = Leader
        // do something...
        go func() {
            // check whether should send RPC
            for !rf.killed && rf.roleState == Leader {
                // send to all
                rf.sendAppendEntriesToAll()
                // wait for next round
                time.Sleep(HeartbeatInterval * time.Millisecond)
            }
        }()
    }
}
```

接下来我们详细解释发送AppendEntries RPC的流程，即`sendAppendEntriesToAll`方法的具体实现。

同样的，我们会首先填充args，其结构如[1]中的图表2所示，此处不再赘述。值得注意的是，RPC中的`entries`参数为`log[nextIndex[peerId]:]`，而Leader所维护的`nextIndex`数组的维护与前序AppendEntries RPC包的返回结果有关，即：每次AppendEntries RPC被拒绝后，Leader会把对应成员在`nextIndex`中的减1，在下次发送时就会包含更多的log，这样不断尝试最终会到底匹配的位置，此时对方即会接受这些新Entries，两者达成一致。

在Leader收到reply后，它会分析返回结果。如果reply的结果是成功（即对方接收了上一次发送的所有log），则更新`nextIndex`和`matchIndex`中的值。更新逻辑比较简单，取出args中的`PrevLogIndex`和`len(Entries)`即可进行计算。需要注意的是，这里通过`max`方法做了一个验证，防止在旧的RPC的delay过高的情况下，把更新的数据覆盖掉的情况。

```
rf.nextIndex[target] = max(args.PrevLogIndex+len(args.Entries)+1,
rf.nextIndex[target])
rf.matchIndex[target] = max(args.PrevLogIndex+len(args.Entries),
rf.matchIndex[target])
```

在有成员接收log之后，Leader需要判断`commitIndex`是否需要更新。这里同样是采用"大多数"的判断策略，具体如[1]中的图表2所说：

> If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N

由于该判断流程是因为`matchIndex[target]`被更新而触发的，显然此时`commitIndex`要么可以更新为`matchIndex[target]`，要么不能更新。换言之，**N若存在，则必然为**`matchIndex[target]`。因此我们的判断流程可以简化为：

```
N := rf.matchIndex[target]
if N > rf.commitIndex && rf.log[N-1].Term == rf.currentTerm {
    // check how many peers have get log[:N], save as deliveredCnt
    deliveredCnt := 0
    for i := 0; i < len(rf.matchIndex); i++ {
        if i == rf.me || rf.matchIndex[i] >= N {
            deliveredCnt++
        }
    }
    // if majority then commit
    if deliveredCnt >= len(rf.peers)/2+1 {
        rf.updateCommitIndex(N)
    }
}
```

由于Leader发的AppendEntries RPC（无论是心跳RPC还是包含log的RPC）都会有一个`LeaderCommit`的字段，因此其他成员在后续接收到新的RPC时可以了解到Leader已经commit了新的log，从而做出相应的措施。

在这一模块的实现中，还涉及到log堆积的问题（尤其是Figure8Unreliable这一样例），需要采取一些优化措施。具体将在后文中的「问题与解决方案」部分介绍。

Safety

上述介绍的主要是Raft算法的基本流程。在这些流程正常运转的前提下，Raft还增加了一些约束用于保证数据一致性。

1. 首先是对选举的约束。为了避免[1]中图表8所示的情况，我们必须确保被选举出的成员包含所有已经commit的log。Raft通过对选举过程的约束来解决这一问题，即：所有成员只给那些log至少和自己一样新的Candidate投票。此处两个成员log的最新程度的比较（“as up-to-date as”）是由最后一个log的`index`和`term`来判断的（具体逻辑不再赘述，见[1]），因此Request Vote RPC的args会携带`LastLogIndex`和`LastLogTerm`这两个字段，然后接收方做一个判断，仅当满足

   ```
   logEntryCompare(rf.log[len(rf.log)-1].Term, len(rf.log), args.LastLogTerm, args.LastLogIndex) <= 0
   ```

   时才可能投票（该条件必要不充分，还有其他因素可能导致拒绝投票，如上文所述）。

2. 其次是对过去的term的log的提交。为了避免[1]中图表8所示的问题，Raft保证不会**仅**因为大多数成员已经获得某条来自previous terms的log entry就将这条log entry提交。这一"大多数"的策略只对来自当前term的log entry有效。而来自previous terms的log entry的提交是间接的，即：在提交当前的term的log entry时，会把该entry之前的所有entries也提交，此时会间接地安全提交所有来自previous terms的entries。如[1]所说的

   > If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, **and log[N].term == currentTerm**: set commitIndex = N

   因此在代码中我们也做了如下判断：

   ```
   if N > rf.commitIndex && rf.log[N-1].Term == rf.currentTerm {
       // count deliverCnt
       if deliveredCnt >= len(rf.peers)/2+1 {
           rf.updateCommitIndex(N)
       }
   }
   ```

Persist

实验3主要通过gob包实现数据持久化，使成员服务器在crash之后能恢复到原本的状态。我们主要通过以下操作完成`currentTerm`、`votedFor`和`log`三个部分的存储：

```
w := new(bytes.Buffer)
e := gob.NewEncoder(w)
e.Encode(rf.currentTerm)
e.Encode(rf.votedFor)
e.Encode(rf.log)
data := w.Bytes()
rf.persister.SaveRaftState(data)
```

读取过程是存储过程的逆操作，采用类似的方法完成。

之后，每次修改以上三个字段的任意一个后，我们都需要调用存储函数，以将最新数据持久化保存。

实现演示

为了便于测试，我们在`/usr/bin/`文件夹中创建了`raft`脚本，写入以下内容（即`test_test.go`文件中的所有测试样例）。测试时，只需要在终端输入`raft`指令即可。每个样例各占一行的主要目的是方便样例的开启与关闭，注释掉特定的内容即可关闭部分样例，便于局部测试。

```
#!/bin/bash
export GOPATH=/home/jcchen/ds
cd /home/jcchen/ds/src/raft
go test -run InitialElection
go test -run ReElection
go test -run BasicAgree
go test -run FailAgree
go test -run FailNoAgree
go test -run ConcurrentStarts
go test -run Rejoin
go test -run Backup
go test -run Count
go test -run Persist1
go test -run Persist2
go test -run Persist3
go test -run Figure8
go test -run UnreliableAgree
go test -run Figure8Unreliable
go test -run ReliableChurn
go test -run UnreliableChurn
cd -
```

实验结果如图所示，通过了所有测试样例。

```
jcchen@DESKTOP-04JRHOD:~$ raft
Test: initial election ...
  ... Passed
PASS
ok      raft    2.517s
Test: election after network failure ...
  ... Passed
PASS
ok      raft    4.521s
Test: basic agreement ...
  ... Passed
PASS
ok      raft    0.985s
Test: agreement despite follower failure ...
  ... Passed
PASS
ok      raft    6.277s
Test: no agreement if too many followers fail ...
  ... Passed
PASS
ok      raft    3.734s
Test: concurrent Start()s ...
  ... Passed
PASS
ok      raft    0.674s
Test: rejoin of partitioned leader ...
  ... Passed
PASS
ok      raft    4.532s
Test: leader backs up quickly over incorrect follower logs ...
  ... Passed
PASS
ok      raft    25.305s
Test: RPC counts aren't too high ...
  ... Passed
PASS
ok      raft    2.335s
Test: basic persistence ...
  ... Passed
PASS
ok      raft    4.300s
Test: more persistence ...
  ... Passed
PASS
ok      raft    26.257s
```

```
Test: partitioned leader and one follower crash, leader restarts ...
  ... Passed
PASS
ok      raft    2.594s
Test: Figure 8 ...
  ... Passed
Test: Figure 8 (unreliable) ...
  ... Passed
PASS
ok      raft    69.863s
Test: unreliable agreement ...
  ... Passed
PASS
ok      raft    5.637s
Test: Figure 8 (unreliable) ...
  ... Passed
PASS
ok      raft    31.037s
Test: churn ...
  ... Passed
PASS
ok      raft    16.491s
Test: unreliable churn ...
  ... Passed
PASS
ok      raft    16.273s
```

问题与解决方案

系统异常commit了很多未获得majority的log entry

- 原因：在实现过程中，错误地用`nextIndex[i]-1`代替`matchIndex[i]`。尽管两者在某些情况下是相同的，但他们的更新策略不同，`nextIndex`只是对下一个应该发送的log的猜测，而`matchIndex`是确定的值；`nextIndex`初始化为尽可能大的值（即`len(log)+1`），而`matchIndex`初始化为0。引用来自[2]的一段话：

  > In a way, `nextIndex` is used for performance – you only need to send these things to this peer. `matchIndex` is used for safety.

- 解决方法：分别实现`nextIndex`和`matchIndex`，严格按照[1]中对两者的要求实现。

log entry累积，`nextIndex`需要尝试多次才能到达正确值，耗时过多（优化问题）

- 原因：在`Figure8Unreliable`测试用例中，会一次性写入给Leader写入多个log entries，因此Leader会累积许多个未同步的log entries，若每次AppendEntries RPC失败后使`nextIndex`减一，则需要与每个成员通讯多次才能最终发送正确的log。
- 解决方法：
  1. 方法1（参考自[2]）：在AppendEntries RPC reply中携带`conflictTerm`和`conflictIndex`，表明发生矛盾的log的位置。
     - 若Follower的log的长度小于`prevLogIndex`，则设置`conflictIndex=len(log)`且`conflictTerm=None`；否则，若只是`prevLogIndex`处的`term`不匹配，则`conflictTerm=log[prevLogInex].Term`，且`conflictIndex`指向log中第一个`term`为`conflictTerm`的log entry。
     - Leader收到回复后，首先搜索自己的log中是否有entry为`conflictTerm`，若有则设`nextIndex`为最后一个`term`为`conflictTerm`的entry的index加1；否则，`nextIndex = conflictIndex`。
  2. 方法2：在某个成员拒绝后，直接将`nextIndex`置为1。由于测试样例中的数据量较小，该方法不会带来太多的额外运行时间。

总结

我们在给定的框架上实现了Raft算法，并通过了`raft/test_test.go`文件中的所有测试样例。本文主要讨论并分析具体的实现思路和细节。对于遇到的问题，我们参考了[1][2]中的解决方案。后续，我们可以进一步优化该框架的的效率和通用性，同时添加更多功能的支持。

参考资料

1. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm (extended version)[J]. Retrieved July, 2016, 20: 2018.
2. https://thesquareplanet.com/blog/students-guide-to-raft/
3. https://tour.golang.org/