Paxos 算法的实现

1 介绍

本次实验使用 go 语言实现了 Paxos 共识算法，并进行了 7 节点规模下的测试。实验环境为 ubuntu linux 系统

2 Paxos 共识算法原理简述

Paxos 算法中的节点分成三种角色，提案发起者 Proposer，提案接收者 Acceptor 和提案学习者 Learner。算法流程可以分为两个阶段，第一个阶段叫准备 (Prepare) 阶段，第二个阶段叫接收 (Accept) 阶段。

2.1 Prepare 阶段

首先，Proposer 选择一个提案编号 n，向所有的 Acceptor 广播 Prepare(n) 请求。

Accptor 收到请求后，分为两种情况：

-如果 n 大于之前接收到所有 Prepare 请求的编号，则返回 Promise() 响应，承诺不会接受小于 n 的提案。如果已经有 Chosen 的提案，Promise() 响应还应包含前一次提案编号和对应的值。

-如果 n 小于等于之前收到的最大编号，就忽略。

2.2 Accept 阶段

Proposer 收到超过半数 Acceptor 的 Promise() 响应后，就向所有 Acceptor 发出 Propose(n, value) 请求，带上提案编号和值。如果之前在 Promise 中收到了值就使用该值，否则自己选定。

Acceptor 收到之后，如果没有对编号更大的 n 另行 Promise 了，就接受该提案。接着 Acceptor 会将接受的提案广播给所有 learner。learner 收到超过半数的提案之后就知道这个提案已被 Chosen。

3 代码实现

3.1 整体框架

首先，为了配合 Paxos 算法中的不同信息类型，在 ConsensusMsg 中加入属性 Tp，有 Prepare，Promise，Propose，Accept 四个值。

```
type MsgType uint8

const (
  Prepare MsgType = iota
  Promise
  Propose
  Accept
)

type ConsensusMsg struct {
  // Type MessageType
  Tp    MsgType
  From uint8
  Seq   uint64
  Data  []byte
}
```

节点启动后首先进入的是 Run() 函数,该函数前面部分保持不变,后面同时开启运行 c.proposeLoop(), c.Run_acceptor(),c.Run_learner() 函数，分别对应三种节点身份。每一轮 Paxos 中有一个 Proposer，需要在 c.proposeLoop() 函数中确定，而每个节点一直都同时是 Acceptor 和 Learner，因此另外两个函数是一直循环运行的。

```
func (c *Consensus) Run() {
  // wait for other node to start
  time.Sleep(time.Duration(1) * time.Second)
  //init rpc client
  rand.Seed(time.Now().UnixNano())
  for id := range c.peers {
    c.peers[id].Connect()
  }

  go c.proposeLoop()
  go c.Run_acceptor()
  c.Run_learner()
}
```

在 proposeLoop() 中循环运行主节点，使用 c.seq 来确定当前轮的主节点，如果 c.seq % 7 = c.id，该节点就成为主节点，运行 c.Run_proposer() 函数。

```
func (c *Consensus) proposeLoop() {
  for {
    // 如果当前节点是主节点
    flag := c.seq % 7
    if flag == uint64(c.id) {
      c.Run_proposer()
    }
  }
}
```

Paxos 算法中需要大量的信息传输，因此采用 RPC 配合信道。在 Consensus 结构体中加入三个信道 msgChan，acChan 和 lrChan，分别用于节点三种身份的信息传输，然后修改函数 OnReceiveMessage，将收到的不同信息放入对应的信道：

```
func (c *Consensus) OnReceiveMessage(args *myrpc.ConsensusMsg, reply *myrpc.
    ConsensusMsgReply) error {

  c.logger.DPrintf("Invoke RpcExample: receive message from %v at %v", args.From, time.Now
      ().Nanosecond())

  switch args.Tp {
    case myrpc.Prepare:
    c.acChan <- args
    case myrpc.Propose:
    c.acChan <- args
    case myrpc.Promise:
    c.msgChan <- args
    case myrpc.Accept:
    c.lrChan <- args
  }

  return nil
}
```

3.2 Prepare 阶段

Run_proposer() 函数：

```
func (c *Consensus) Run_proposer() {

  //生成新的区块
  block := c.blockChain.getBlock(c.seq)
  pre_msg := &myrpc.ConsensusMsg{
    Tp:   myrpc.Prepare,
    From: c.id,
    Seq:  c.seq,
    Data: block.Data,
  }

  //广播Prepare
  c.broadcastMessage(pre_msg)
  c.data = block.Data

  timeout := time.After(10 * time.Millisecond)
  l:
  for {
    select {
      case msg := <-c.msgChan:

      switch msg.Tp {
        case myrpc.Promise:
        c.promised += 1
        if msg.Seq > 0 {
          //c.seq = msg.Seq
```

```
            c.data = msg.Data
          }

          default:
          panic("UnSupport message.")
        }
        case <-timeout:

        break l
      }
    }
```

首先生成新的区块，作为这一轮 prepare message 的值，广播给所有 acceptor，然后循环接收 acceptor 的 Promise 答复。设置超时机制来跳出循环。

Run_acceptor() 函数：

```
func (c *Consensus) Run_acceptor() {
  for {
    timeout := time.After(60 * time.Millisecond)
    select {
      case msg := <-c.acChan:
      switch msg.Tp {
        case myrpc.Prepare:
        if c.promiseNum < msg.Seq {
          //只有当收到的seq大于已promised的数字才promise
          c.promiseNum = msg.Seq
          rep := &myrpc.ConsensusMsg{
            Tp:   myrpc.Promise,
            From: c.id,
            Seq:  c.acceptedNum,
            Data: c.acceptedData,
          }
          reply := &myrpc.ConsensusMsgReply{}
          c.peers[msg.From].Call("Consensus.OnReceiveMessage", rep, reply)
        }
```

该函数的内部设置一个一直循环的 for，不断从信道 acChan 中接收消息。接收到 Prepare 消息之后，比较消息编号 msg.Seq 与已经承诺过的编号，只有当收到的 seq 大于已 promised 的数字才 promise。在回复的信息 rep 中放入 acceptedNum 和 acceptedData，如果之前没有 Chosen 的提案这两个值都会是零值。然后使用 rpc 将 Promise 回复发给 Proposer。

3.3 Accept 阶段

Run_proposer() 函数：

```
if c.promised >= 4 {
pro_msg := &myrpc.ConsensusMsg{
  Tp:   myrpc.Propose,
  From: c.id,
  Seq:  c.seq,
  Data: c.data,
```

```
    }
    c.broadcastMessage(pro_msg)
    c.promised = 0
  }
  time.Sleep(20 * time.Millisecond)
```

如果收到了大多数 Acceptor 的 Promise，则构造 Propose 信息。其中 Data 取决于之前收到的 Promise。然后将该信息广播给所有 Acceptor，并将 c.promised 这个计数器的值恢复为零，然后启动 time.Sleep 等待一段时间让该轮共识完结，然后结束该函数。

Run_acceptor() 函数：

```
  case myrpc.Propose:
  num := msg.Seq
  if num >= c.promiseNum {
    c.acceptedNum = num
    c.acceptedData = msg.Data
    c.promiseNum = num
    ac := &myrpc.ConsensusMsg{
      Tp:   myrpc.Accept,
      From: c.id,
      Seq:  c.acceptedNum,
      Data: c.acceptedData,
    }
    c.broadcastMessage(ac)
  }
```

接收到 Propose 消息后，将消息的 seq 与之前 promise 过的值进行对比，如果大于等于则接受，并设置 acceptedNum 和 acceptedData 为相应的值。最后使用 broadcastMessage 函数将 Accpet 消息广播给所有 Learner。

Run_learner() 函数：

```
  func (c *Consensus) Run_learner() {
    for {
      msg := <-c.lrChan
      c.chosen(msg)

    }
  }
```

该函数循环不断从 lrChan 中接收 Accept 信息，接收到之后就用 chosen 函数判断是否符合条件，可以被 Chosen。

chosen() 函数：

```
  func (c *Consensus) chosen(msg *myrpc.ConsensusMsg) {

    if !reflect.DeepEqual(c.cuChose, msg.Data) {
      c.cuChose = msg.Data
      c.cuCount = 1
    } else {
```

```
        c.cuCount += 1
      }
      if c.cuCount == 4 {
        if msg.Seq != c.lastSeq && (msg.Seq == 0 || msg.Seq > c.lastSeq) {
          block := &Block{
            Seq:  msg.Seq,
            Data: msg.Data,
          }
          // 将构造的Block提交到区块链中
          c.blockChain.commitBlock(block)
          c.lastSeq = msg.Seq
        }
        c.seq = msg.Seq + 1
        c.acceptedNum = 0
        c.acceptedData = []byte{}
        c.cuChose = []byte{}
      }
    }
```

利用了两个属性 cuChose 和 cuCount，对收到的信息进行计数，到达大多数（4）之后就可以 Chosen 该提案。为了保证生成区块的顺序当前信息 seq 要大于等于上一个 seq。成功选择的提案就会作为区块提交到区块链中，代表该轮共识完成，然后重置几个属性的值。

3.4 crash 处理

如果该轮的 Proposer crash，程序会卡住，因此在 Acceptor 中利用 select 加入超时机制，信道里一段时间没有收到消息就进行超时处理。

```
    case <-timeout:
    t := rand.Intn(7)
    if t == int(c.id) {
      c.seq++
      fmt.Printf("seq increased")
    }
```

如果只有一个节点 crash 的话，其实可以设置让它的下一个节点来充当 Proposer，但是要应对两个节点的 crash，因此引入了随机机制，随机选一个节点来让它的 seq+1，这样总会有一个节点的 seq 增加到能充当 Proposer 的条件，成为新的 Proposer。

4 实验结果

```
njy@njy-virtual-machine:~/SimpleConsensus/scripts$ ./normal_test.sh
----------begin to check------------
node 0 commits 315 times
node 1 commits 315 times
node 2 commits 315 times
node 3 commits 315 times
node 4 commits 315 times
node 5 commits 315 times
node 6 commits 315 times
---------- check validity ------------
pass
---------- check safety -------------
pass
---------- end ----------------------
```

图 1: normal_test

```
njy@njy-virtual-machine:~/SimpleConsensus/scripts$ ./crash1_test.sh 2
----------begin to check------------
node 0 commits 160 times
node 1 commits 160 times
node 2 commits 91 times
node 3 commits 160 times
node 4 commits 160 times
node 5 commits 160 times
node 6 commits 160 times
---------- check validity ------------
pass
---------- check safety -------------
pass
---------- end ----------------------
```

图 2: crash1_test

```
njy@njy-virtual-machine:~/SimpleConsensus/scripts$ ./crash2_test.sh 0 1
----------begin to check------------
node 0 commits 107 times
node 1 commits 150 times
node 2 commits 216 times
node 3 commits 216 times
node 4 commits 216 times
node 5 commits 216 times
node 6 commits 216 times
---------- check validity ------------
pass
---------- check safety --------------
pass
---------- end -----------------------
```

图 3: crash2_test