分布式的键值存储系统

19335171 容浩民

题目

设计并实现一个分布式键值（key-value）存储系统，可以是基于磁盘的存储系统，也可以是基于内存的存储系统，可以是主从结构的集中式分布式系统，也可以是P2P式的非集中式分布式系统。能够完成基本的读、些、删除等功能，支持缓存、多用户和数据一致性保证, 提交时间为考试之后一周内。

要求：

1）、必须是分布式的键值存储系统，至少在两个节点或者两个进程中测试；

2）、可以是集中式的也可以是非集中式；

3）、能够完成基本的操作如：PUT、GET、DEL等；

4）、支持多用户同时操作；

5）、至少实现一种面向客户的一致性如单调写；

6）、需要完整的功能测试用例；

7）、涉及到节点通信时须采用RPC机制；

8)、提交源码和报告，压缩后命名方式为：学号*姓名*班级

加分项：

1）、具备性能优化措施如cache等；

2）、具备失效容错方法如：Paxos、Raft等；

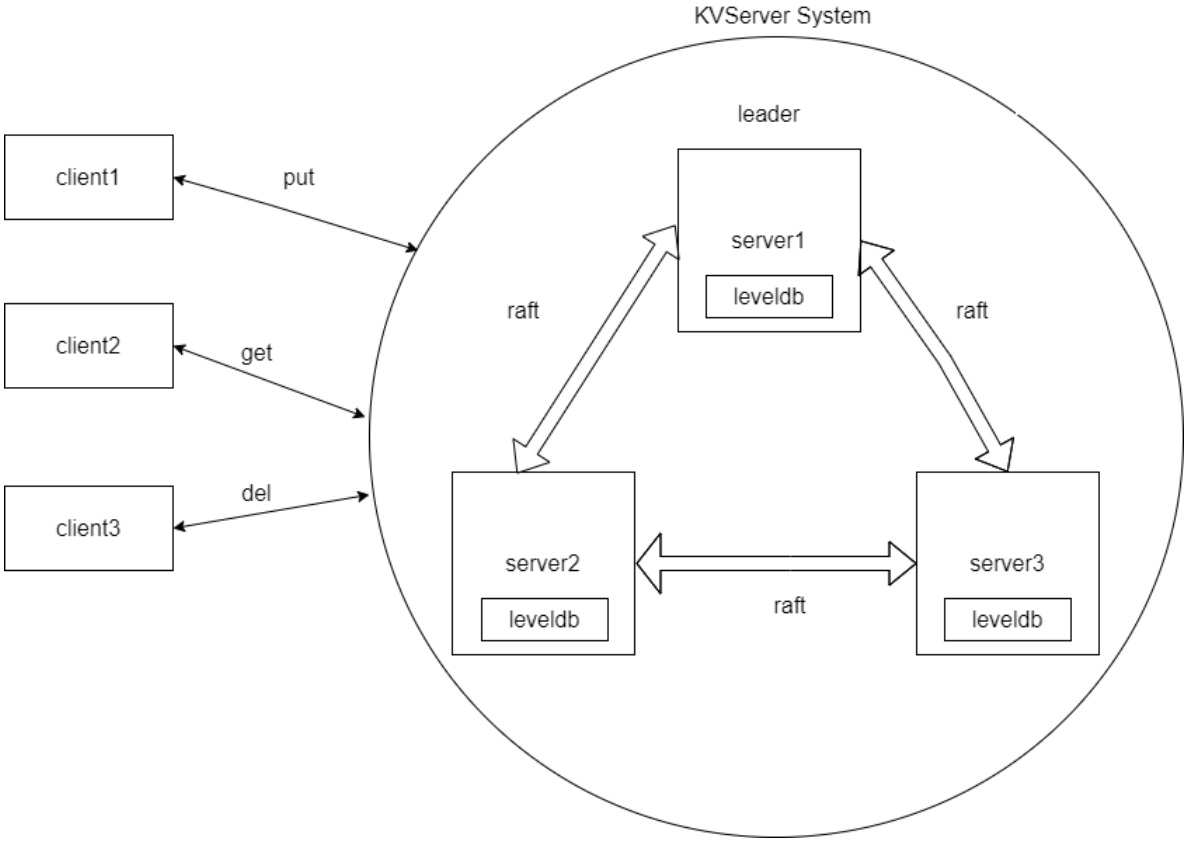
3）、具备安全防护功能；

4）、其他高级功能；

实现方案

实现了raft协议，所有通信均采用grpc实现，本地的数据库使用leveldb实现。编程语言使用go。

分布式结构为主备模式，即一个master服务器带有多个副本，客户的写请求发送到master服务器上执行，读请求可以在副本服务器上进行，从而加快访问速度。



Raft实现细节

Raft里grpc通信定义

raft组件里面的通信

- HandleMessage方法 负责接受请求来做出相应的响应。
- 枚举类型 MessageType，来定义消息的类型
  - MsgEmpty：不带有任何信息，不需要任何处理
  - MsgNewElection: 发起选举的信息
  - MsgHeartBeat：心跳信息，用来维持leader的权威
  - MsgAppend：日志复制信息，用来进行日志的复制
- Message：作为请求信息
  - msg_type：消息类型
  - term：发送的term
  - index：发送的index
  - entries：发送的日志
  - leader_address：leader的地址
  - match_check：日志匹配的index
- Response：返回的响应信息
  - msg_type：响应的消息类型
  - term：返回的term
  - match：只有msg_type=MsgAppend时有意义，返回日志是否匹配
  - match_index：只有msg_type=MsgAppend时有意义，返回日志匹配的index
  - address ：负责返回消息的服务器地址
  - append_success：只有msg_type=MsgAppend时有意义，返回日志是否附加成功
- Entry
  - term: 日志的term
  - index: 日志的index

- data: 日志的data，用bytes的形式来储存，可以保存不同类型的数据（本次实验保存的是json序列化之后的bytes）

```
syntax = "proto3";
package raftrpc;

option go_package="./;raftrpc";

service Raftrpc{
  rpc HandleMessage(Message) returns(Response){}
}

enum MessageType{
    MsgEmpty = 0;
    MsgNewElection = 1;
    MsgHeartBeat = 2;
    MsgAppend = 3;

}
message Response {
  MessageType msg_type = 1;
  int64 term = 2;
  bool match = 3;
  int64 match_index = 4;
  string address = 5;
  bool  append_success = 6;
}

message Entry {
  int64 term = 1;
  int64 index = 2;
  bytes data = 3;
}

message Message{
    MessageType msg_type = 1;
    int64 term = 2;
    int64 index = 3;
    repeated Entry entries = 4;
    string leader_address = 5;
    bool match_check = 6;
}
```

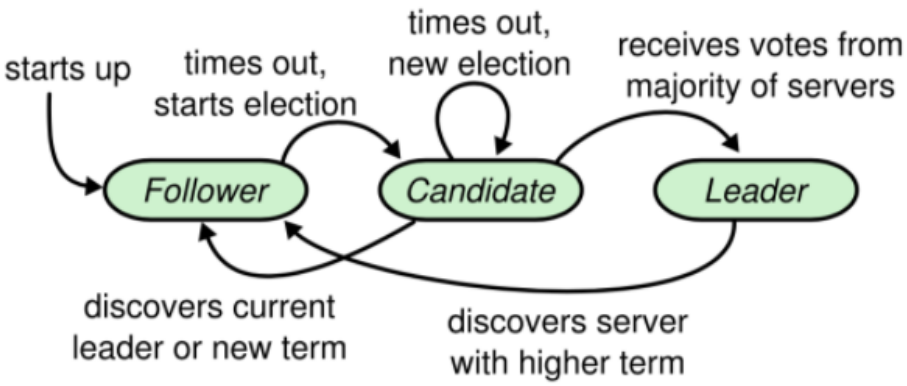
Raft结构体的定义

```
type Raft struct {
    //raft服务的地址
    NodeAddress string
    // raft集群中其他节点的地址
    PeerNodeAddress []string

    // raft中的日志，仅位于内存
    Log *RaftLog

    // 其他节点的log匹配信息
    Prs map[string]*Progress
```

```
    // 当前的term
    Terms int64

    // 心跳超时时间
    heartbeatTimeout int
    // 心跳逻辑时钟，只有leader才需要维护
    heartbeatElapsed int

    // 选举超时时间
    electionTimeout int
    // 选举逻辑时钟， 只有follower才需要维护
    electionElapsed int

    // leader的地址
    leaderAddress string

    // 当前的状态，leader, follower, candidate 中的一个
    State NodeStates

    // 选举选票的记录，超过一半才能当选
    voteRecord map[string]bool

    // 当前日志append记录，只有一半的节点返回成功，才能将日志commit
    appendRecord map[string]bool

    // 服务线程的关闭信号
    stopchan chan bool

    // 连接的本地leveldb数据库
    db *leveldb.DB
}
```

选举机制实现

每一个raft节点有三种角色，分别为leader，follower，candidate，每个节点只能是这三种角色中的一种。



leader心跳维持

每一个leader会维护一个心跳定时器，当发生超时，会发送心跳信息到raft组内的其他follower，当其他follower收到心跳信息后，会根据心跳信息当中的term来更新自己的term，并且重置自己的选举超时器，使其不会超时，也就不会成为candidate来发起新的选举。

```
// 收到心跳信息
    if message.GetMsgType() == raftpb.MessageType_MsgHeartBeat {
        s.r.leaderAddress = message.GetLeaderAddress()
        s.r.BecomeFollower(message.GetTerm())
        //log.Println("got heartBeat")
        s.r.electionElapsed = 0
        return &raftpb.Response{}, nil
    }
```

特别说明：本次实验为了方便调试，超时器采用了逻辑时钟实现，没有设置精确的超时时间。

```
// LogicalTimerChange 触发逻辑时钟加1
func (r *Raft) LogicalTimerChange() {
    if r.State == Leader {
        r.heartbeatElapsed += 1
    }
    if r.State == Follower {
        r.electionElapsed += 1
    }

    // 选举超时，发起新的选举
    if r.electionElapsed == r.electionTimeout {
        //log.Println("election timeout")
        r.electionElapsed = 0
        r.BecomeCandidate()
    }

    // leader每一个心跳周期发送heartbeat来维持权威
    if r.heartbeatElapsed == r.heartbeatTimeout {
        //log.Println("heartbeat timeout")
        r.sendHeartBeat()
        r.heartbeatElapsed = 0
    }
}
```

选举过程

如果一个follower在一段时间内没有接受到任何来自于leader的心跳信息，就会认为系统当中没有leader，会触发选举时钟超时，follower会将自己的状态转化为candidate，并且将自己的term加1来发起选举。发起选举的方式是向集群中的其他节点发送请求投票的rpc来让其他节点给自己投票。candidate会一致保持自己的状态，直到

- 他自己赢得了这一次的选举
- 其他的服务器成为领导
- 一段时间之后没有任何获胜的人。为了方便处理，我给每一个raft节点设置了不同的选举超时时间，防止出现同时超时但又没人胜出的循环现象。

为了记录选票，我使用了`voteRecord map[string]bool`数组来保存选票信息，一旦收到超过半数的服务器的同意，将状态切换为leader，并同时向其他节点发送心跳信息来维持权威。

```
// 收到投票信息，检查选票是否过半，如过半，成为leader
```

```
if response.GetMsgType() == raftpb.MessageType_MsgNewElection && r.State ==
Candidate {
   r.voteRecord[response.GetAddress()] = true
   log.Printf("%v get vote from %v\n", r.NodeAddress, response.GetAddress())
   count := 0
   for i := 0; i < len(r.PeerNodeAddress); i++ {
      if r.voteRecord[r.PeerNodeAddress[i]] == true {
         count += 1
      }
   }

   if count+1 >= (len(r.PeerNodeAddress)+1)/2 {
      r.BecomeLeader(r.Terms)
   }
}
```

日志复制实现

当leader收到一条附加日志时，会将该日志发送到其他follower中来进行附加，在日志附加之前，需要先找到follower节点与leader当中的日志相匹配的index。

日志的匹配与下列的日志匹配特性有关

- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。
- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也全部相同。

在寻找matchIndex时，leader针对每一个follower维护了一个nextIndex，这表示下一个需要发送给跟随者的日志条目的索引地址。当一个leader成为leader时，会初始化自身所有的nextIndex只为自己的最后一条日志的index加1。如果某个follower的日志与leader不一致，那么一致性的检查就会发生失败。在收到一致性检查失败后的信息，leader会减少nextIndex的值并进行重试。最后nextIndex会在某一个位置当中使得leader和follower的日志达成一致。极端情况下，日志全都不匹配，leader会把全部日志发送给follower去附加。

leader的操作：

```
// 处理日志附加操作
    if response.GetMsgType() == raftpb.MessageType_MsgAppend && r.State ==
Leader {
        // Append 成功，重置match值
        if response.GetAppendSuccess() == true {
            r.Prs[response.GetAddress()].Match = -2
            r.appendRecord[response.GetAddress()] = true
            count := 0
            for i := 0; i < len(r.PeerNodeAddress); i++ {
                if r.appendRecord[r.PeerNodeAddress[i]] == true {
                    count += 1
                }
            }
            if count+1 >= (len(r.PeerNodeAddress)+1)/2 {
                r.Log.committed = r.Log.LastIndex()
            }
            return
        }
        //没有match，减少对应follower的Next，继续重试
```

```
        if response.GetMatch() == false {
            r.Prs[response.GetAddress()].Next -= 1
            r.sendAppend(response.GetAddress())
        } else {
            r.Prs[response.GetAddress()].Match = response.GetMatchIndex()
            r.sendAppend(response.GetAddress())
        }
    }
```

follower的操作

收到append消息后，判断是否match，如果没有match，返回没有match的信息给leader，如果当前日志match，就将日志附加到本地的日志上，并且对本地数据库进行持久化操作。

```
// 收到append信息
    if message.GetMsgType() == raftpb.MessageType_MsgAppend {
        if message.GetMatchCheck() == true {
            // 根据term和index判断是否match

            // index = -1, 说明没有日志，回复match
            if message.GetIndex() == -1 {
                return &raftpb.Response{MsgType: raftpb.MessageType_MsgAppend,
                    Match:      true,
                    MatchIndex: message.GetIndex(),
                    Address:    s.r.NodeAddress}, nil
            }

            // 日志长度 - 1 小于发来的index， 不match
            if len(s.r.Log.entries)-1 < int(message.GetIndex()) {
                return &raftpb.Response{MsgType: raftpb.MessageType_MsgAppend,
                    Match:   false,
                    Address: s.r.NodeAddress}, nil
            }

            // 同一index下term不同，不match
            if s.r.Log.entries[message.GetIndex()].GetTerm() !=
message.GetTerm() {
                return &raftpb.Response{MsgType: raftpb.MessageType_MsgAppend,
                    Match:   false,
                    Address: s.r.NodeAddress}, nil
            }

            // 其他情况，返回match
            return &raftpb.Response{MsgType: raftpb.MessageType_MsgAppend,
                Match: true, MatchIndex: message.GetIndex(),
                Address: s.r.NodeAddress}, nil
        } else {
            // 收到leader的日志后，附加到自己的日志上
            s.r.Log.entries = s.r.Log.entries[:message.GetIndex()+1]
            for i := 0; i < len(message.Entries); i++ {
                s.r.Log.entries = append(s.r.Log.entries, *message.Entries[i])
            }
            s.r.Log.lastappliedIndex = message.GetIndex()

            // 写到本地数据库
            if s.r.db != nil {
                go func() {
```

```
                    for i := 0; i < len(message.Entries); i++ {
                        handleLogEntry(message.Entries[i], s.r.db)
                        s.r.Log.lastappliedIndex++
                    }

                }()
            }
            //log.Println(s.r.Log.entries)
        }
        return &raftpb.Response{MsgType: raftpb.MessageType_MsgAppend,
AppendSuccess: true, Address: s.r.NodeAddress}, nil
    }
```

日志附加导致的投票策略改变

Raft 使用投票的方式来阻止一个候选人赢得选举除非这个候选人包含了所有已经提交的日志条目。候选人为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果候选人的日志至少和大多数的服务器节点一样新，那么他一定持有了所有已经提交的日志条目。请求投票 RPC 实现了这样的限制： RPC 中包含了候选人的日志信息，然后投票人会拒绝掉那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号定义谁的日志比较新。如果两份日志最后的条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

```
// 收到选举信息
    if message.GetMsgType() == raftpb.MessageType_MsgNewElection {
        //log.Println(message.GetTerm(),
s.r.Log.entries[s.r.Log.LastIndex()].GetTerm())

        /*
            当前没有日志，或者最后一条日志的term小于leader的日志，或者term相同，日志的长度
较短
            投票，否则拒绝投票
        */
        if len(s.r.Log.entries) == 0 ||
s.r.Log.entries[s.r.Log.LastIndex()].GetTerm() < message.GetTerm() {
            return &raftpb.Response{MsgType: raftpb.MessageType_MsgNewElection,
Address: s.r.NodeAddress}, nil
        } else if s.r.Log.entries[s.r.Log.LastIndex()].GetTerm() ==
message.GetTerm() {
            if s.r.Log.LastIndex() <= message.GetIndex() {
                return &raftpb.Response{MsgType:
raftpb.MessageType_MsgNewElection, Address: s.r.NodeAddress}, nil
            }
        }

        return &raftpb.Response{MsgType: raftpb.MessageType_MsgEmpty}, nil
    }
```

节点实现

节点grcp通信定义

节点get，put，del的通信定义，分为RawPutRequest，RawGetRequest，RawDelRequest三种请求信息和RawGetResponse，RawPutResponse，RawDelResponse三种响应信息。其中请求信息中定义了key，value两种bytes，使用bytes来完成键值对的插入，查询或者删除。查询的响应信息有些不同，设置了not_found字符，来判断是否没有该键值对。

```
syntax = "proto3";
package kvrpc;

option go_package="./;kvrpc";

service KVrpc{
    rpc RawPut(RawPutRequest) returns (RawPutResponse){}
    rpc RawGet(RawGetRequest) returns (RawGetResponse){}
    rpc RawDel(RawDelRequest) returns (RawDelResponse){}
}

message RawPutRequest{
    bytes key = 1;
    bytes value = 2;
}

message RawPutResponse{
    string error = 1;
}


message RawGetRequest{
    bytes key = 1;
}

message RawGetResponse{
    bytes value = 1;
    string error = 2;
    bool not_found = 3;
}

message RawDelRequest{
    bytes key = 1;
}

message RawDelResponse{
    string error = 1;
}
```

服务器节点实现

```
type KVserver struct {
    pb.KVrpcServer
    r * raft.Raft
    db *leveldb.DB
}


type KVOperations int
```

```
const (
    _ KVOperations = iota
    Put
    Get
    Del
)


type KVentry struct{
    KVOperation KVOperations `json:"kv_operation"`
    Value []byte `json:"value"`
    Key   []byte `json:"key"`
}
```

将节点当中的leveldb数据库和raft模块封装到一个KVserver当中。

- RawPut：先将请求的bytes转化为定义好的KVentry，再进行序列化，转化为日志进行附加，如果是leader才附加到自己的日志中，再转发给其他follower，如果是follower，则不处理请求。
- RawGet：所有的服务器都可以响应请求，因为读操作不涉及到日志的附加。
- RawDel：于RawPut类似，只有leader才处理请求，再通过raft机制转发到其他follower上。

其中上述的服务端grpc服务实现如下

```
func (s *KVserver) RawPut(ctx context.Context, in *pb.RawPutRequest)
(*pb.RawPutResponse, error) {
    log.Printf("Received: key is %v, value is %v",
string(in.GetKey()),string(in.GetValue()))
    kventry := raft.KVentry{}
    kventry.KVOperation = raft.Put
    kventry.Key = in.GetKey()
    kventry.Value = in.GetValue()
    kventryByte,_ := json.Marshal(kventry)
    flag:= s.r.Append(kventryByte)
    if flag{
        err := s.db.Put(in.GetKey(), in.GetValue(), nil)
        if err != nil {
            log.Println("RawPut error is ",err)
            return &pb.RawPutResponse{Error: fmt.Sprintf("err is %v",err)}, nil
        }
        return &pb.RawPutResponse{Error: ""}, nil
    }else{
        return &pb.RawPutResponse{Error: "this server node can't exec put"}, nil
    }

}

func (s *KVserver) RawGet(ctx context.Context, in *pb.RawGetRequest)
(*pb.RawGetResponse, error) {
    log.Printf("Received: key is %v", string(in.GetKey()))

    value, err := s.db.Get(in.GetKey(), nil)
    if err != nil {
        log.Println("RawGet error is ",err)
        if err == leveldb.ErrNotFound {
```

```
            return &pb.RawGetResponse{Value: value, Error: "", NotFound: true},
nil
        }
        return &pb.RawGetResponse{Value: value,Error: fmt.Sprintf("err is
%v",err),NotFound: false}, nil
    }
    return &pb.RawGetResponse{Value: value,Error: "",NotFound: false}, nil
}


func (s *KVserver) RawDel(ctx context.Context, in *pb.RawDelRequest)
(*pb.RawDelResponse, error) {
    log.Printf("Received: key is %v", string(in.GetKey()))
    kventry := raft.KVentry{}
    kventry.KVOperation = raft.Del
    kventry.Key = in.GetKey()
    kventryByte,_ := json.Marshal(kventry)
    flag:= s.r.Append(kventryByte)
    if flag{
        err := s.db.Delete(in.GetKey(),nil)
        if err != nil {
            log.Println("RawPut error is ",err)
            return &pb.RawDelResponse{Error: fmt.Sprintf("err is %v",err)}, nil
        }
        return &pb.RawDelResponse{Error: ""}, nil
    }else{
        return &pb.RawDelResponse{Error: "this server node can't exec put"}, nil
    }
}
```

客户端节点实现

将其封装为简单的函数调用的形式，客户使用服务器地址和请求操作就能够进行Put，Get，Del操作。

```
func Put(address string,in *pb.RawPutRequest)(*pb.RawPutResponse, error){
    conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer func(conn *grpc.ClientConn) {
        err := conn.Close()
        if err != nil {
            return
        }
    }(conn)
    c := pb.NewKVrpcClient(conn)

    // Contact the server and print out its response.
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.RawPut(ctx,in)

    return r, err
}


func Get(address string,in *pb.RawGetRequest)(*pb.RawGetResponse, error){
```

```
    conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer func(conn *grpc.ClientConn) {
        err := conn.Close()
        if err != nil {
            return
        }
    }(conn)
    c := pb.NewKVrpcClient(conn)

    // Contact the server and print out its response.
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.RawGet(ctx,in)

    return r, err
}


func Del(address string,in *pb.RawDelRequest)(*pb.RawDelResponse, error){
    conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer func(conn *grpc.ClientConn) {
        err := conn.Close()
        if err != nil {
            return
        }
    }(conn)
    c := pb.NewKVrpcClient(conn)

    // Contact the server and print out its response.
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.RawDel(ctx,in)

    return r, err
}
```

单元测试

raft模块单元测试

raft模块的所有单元测试在raft_test.go文件当中，可以运行单元测试。

选举测试

使用三个节点进行测试，分别设置选举超时时间为3，5，10，因为第一个节点的超时时间设置成3，所以会率先超时，成为candidate，然后收到其他节点的投票成为leader。然后节点1关闭，会触发重新选举，又因为节点2的超时时间小于节点3，所以节点3会给节点2投票。然后节点2当选leader。

```
func TestRaftElection(t *testing.T) {
    setting := Setting{NodeAddress: ":50051", HeartbeatTimeout:
1,ElectionTimeout: 3,PeerNodeAddress:[]string{":50052",":50053"}}
    r1 := Raft{}
    r1.NewRaft(&setting)


    setting2 := Setting{NodeAddress: ":50052", HeartbeatTimeout:
1,ElectionTimeout: 5,PeerNodeAddress:[]string{":50051",":50053"}}
    r2 := Raft{}
    r2.NewRaft(&setting2)

    setting3 := Setting{NodeAddress: ":50053", HeartbeatTimeout:
1,ElectionTimeout: 10,PeerNodeAddress:[]string{":50051",":50052"}}
    r3 := Raft{}
    r3.NewRaft(&setting3)

    for i:= 0;i < 10;i++{
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
    }

    assert.Equal(t, r1.State, Leader)
    assert.Equal(t, r2.leaderAddress, ":50051")
    assert.Equal(t, r3.leaderAddress, ":50051")

    // leader怠机，会重新选举
    r1.StopRaft()

    for i:= 0;i < 10;i++{
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
    }

    assert.Equal(t, r2.State, Leader)
    assert.Equal(t, r3.leaderAddress, ":50052")
    r2.StopRaft()
    r3.StopRaft()
}
```

测试结果：

```
=== RUN   TestRaftElection
2022/01/19 11:33:01 raft server listening at [::]:50051
2022/01/19 11:33:01 raft server listening at [::]:50052
2022/01/19 11:33:01 raft server listening at [::]:50053
2022/01/19 11:33:01 :50051 get vote from :50052
2022/01/19 11:33:01 :50052 get vote from :50051
rpc error: code = DeadlineExceeded desc = context deadline exceeded
2022/01/19 11:33:01 :50052 get vote from :50053
rpc error: code = DeadlineExceeded desc = context deadline exceeded
rpc error: code = DeadlineExceeded desc = context deadline exceeded
rpc error: code = DeadlineExceeded desc = context deadline exceeded
rpc error: code = DeadlineExceeded desc = context deadline exceeded
rpc error: code = DeadlineExceeded desc = context deadline exceeded
rpc error: code = DeadlineExceeded desc = context deadline exceeded
```

```
--- PASS: TestRaftElection (0.30s)
```

测试通过，选举机制没有问题。

日志复制测试

日志没有不一致的情况下的测试

手动在leader上附加日志，检查节点上是否存在对应term，index，data的日志存在。

```
// 测试没有错误出现时的日志复制
func TestRaftAppend(t *testing.T) {
    ....

    for i:= 0;i < 20;i++{
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
        if i == 10{
            assert.Equal(t, r1.State,Leader)
            r1.Append(kventryByte)
        }

        if i == 13{
            r1.Append(kventryByte2)
        }

        if i == 15{
            success := r1.Append(kventryByte3)
            assert.Equal(t, r1.Log.committed, int64(2))
            assert.Equal(t, success, true)
        }

    }

    assert.Equal(t, r2.Log.entries[1].GetTerm(),int64(1))
    assert.Equal(t, r2.Log.entries[1].GetIndex(),int64(1))
    assert.Equal(t, r2.Log.entries[1].GetData(),kventryByte2)

    assert.Equal(t, r2.Log.entries[2].GetTerm(),int64(1))
    assert.Equal(t, r2.Log.entries[2].GetIndex(),int64(2))
    assert.Equal(t, r2.Log.entries[2].GetData(),kventryByte3)


    assert.Equal(t, r3.Log.entries[2].GetTerm(),int64(1))
    assert.Equal(t, r3.Log.entries[2].GetIndex(),int64(2))
    assert.Equal(t, r3.Log.entries[2].GetData(),kventryByte3)

    ...

}
```

测试当中打印了matchindex的匹配过程，与定义的过程一致，而且每一个节点上的日志都与leader上的日志相同，所以日志复制正确。

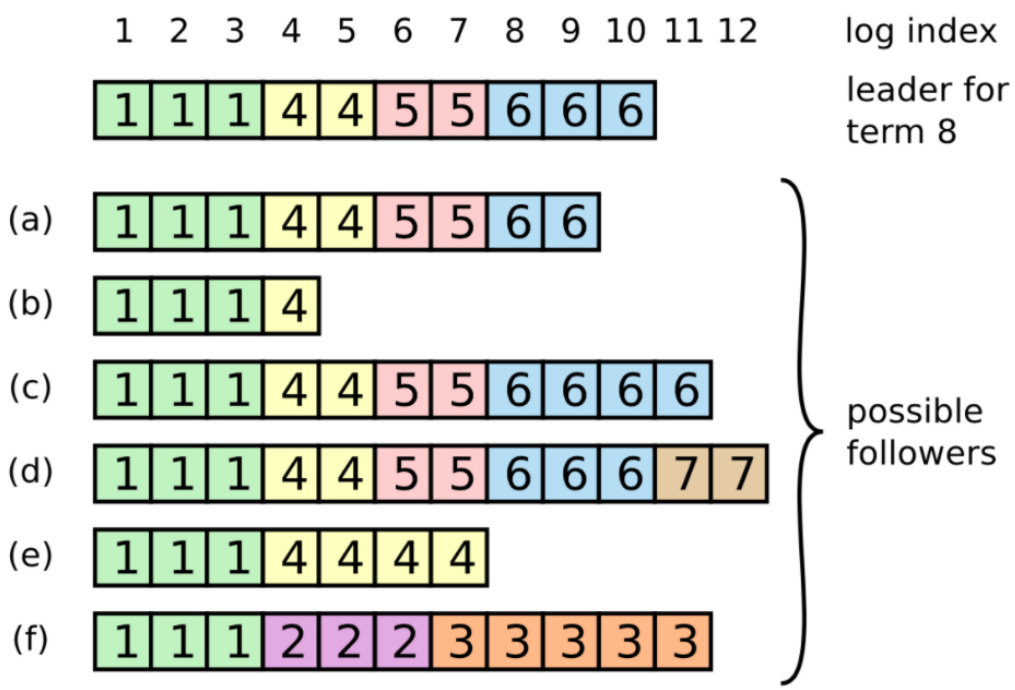
```
=== RUN   TestRaftAppend
2022/01/19 11:33:02 raft server listening at [::]:50051
```

```
2022/01/19 11:33:02 raft server listening at [::]:50052
2022/01/19 11:33:02 raft server listening at [::]:50053
2022/01/19 11:33:02 :50051 get vote from :50052
2022/01/19 11:33:02 send to :50052, match index is -2 , Next index is 0
2022/01/19 11:33:02 send to :50052, match index is -1 , Next index is 0
2022/01/19 11:33:02 send to :50053, match index is -2 , Next index is 0
2022/01/19 11:33:02 send to :50053, match index is -1 , Next index is 0
2022/01/19 11:33:02 send to :50052, match index is -2 , Next index is 1
2022/01/19 11:33:02 send to :50052, match index is 0 , Next index is 1
2022/01/19 11:33:02 send to :50053, match index is -2 , Next index is 1
2022/01/19 11:33:02 send to :50053, match index is 0 , Next index is 1
2022/01/19 11:33:02 send to :50052, match index is -2 , Next index is 2
2022/01/19 11:33:02 send to :50052, match index is 1 , Next index is 2
2022/01/19 11:33:02 send to :50053, match index is -2 , Next index is 2
2022/01/19 11:33:02 send to :50053, match index is 1 , Next index is 2
--- PASS: TestRaftAppend (0.05s)
```

日志出现了不一致的情况下的测试

日志的不一致有下列几种情况，分别为：

- 日志还没有附加
- 之前的日志缺失
- 日志多出一部分
- 日志多出一部分且term与leader不一致
- 日志缺失且term与leader不一致
- 日志多出一部分且之前的日志也不一致



分别对这几种情况进行测试

```go
// 测试日志不一致的情况
func TestRaftAppend2(t *testing.T) {
    ...

    for i:= 0;i < 20;i++{
```

```
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
        if i == 10{
            r1.Append(kventryByte)
        }
        if i == 13{
            r1.Append(kventryByte2)
        }

        if i == 15{
            r1.Append(kventryByte3)
        }
        // 强制修改日志使得日志不一致
        if i == 16{
            //日志的term不一致
            r3.Log.entries[2].Term = 3
            r3.Log.entries[2].Data = []byte("hhhhh")

            //日志缺少
            r2.Log.entries = r2.Log.entries[:1]
            log.Println(len(r2.Log.entries))
        }
        if i == 18{
            r1.Append(kventryByte4)
        }

    }

    assert.Equal(t, r2.Log.entries[2].GetTerm(),int64(1))
    assert.Equal(t, r2.Log.entries[2].GetIndex(),int64(2))
    assert.Equal(t, r2.Log.entries[2].GetData(),kventryByte3)

    assert.Equal(t, r2.Log.entries[3].GetTerm(),int64(1))
    assert.Equal(t, r2.Log.entries[3].GetIndex(),int64(3))
    assert.Equal(t, r2.Log.entries[3].GetData(),kventryByte4)
    assert.Equal(t, len(r2.Log.entries), 4)


    assert.Equal(t, r3.Log.entries[2].GetTerm(),int64(1))
    assert.Equal(t, r3.Log.entries[2].GetIndex(),int64(2))
    assert.Equal(t, r3.Log.entries[2].GetData(),kventryByte3)

    assert.Equal(t, r3.Log.entries[3].GetTerm(),int64(1))
    assert.Equal(t, r3.Log.entries[3].GetIndex(),int64(3))
    assert.Equal(t, r3.Log.entries[3].GetData(),kventryByte4)
    assert.Equal(t, len(r3.Log.entries), 4)

    //change leader and term
    r1.BecomeFollower(1)
    r2.BecomeCandidate()

    for i:= 0;i < 10;i++{
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()

        if i == 3{
```

```
            // log比leader长的情况
            assert.Equal(t, len(r2.Log.entries), 4)
            r3.Log.entries = append(r3.Log.entries, raftpb.Entry{})
            r3.Log.entries = append(r3.Log.entries, raftpb.Entry{})
            assert.Equal(t, len(r3.Log.entries), 6)

            // log变短且不一致
            r1.Log.entries = r1.Log.entries[:3]
            assert.Equal(t, len(r1.Log.entries), 3)
            r1.Log.entries[2].Term = 3
        }


        if i == 5{
            r2.Append(kventryByte5)
        }
    }

    assert.Equal(t, r2.Log.entries[4].GetTerm(),int64(2))
    assert.Equal(t, r2.Log.entries[4].GetIndex(),int64(4))
    assert.Equal(t, r2.Log.entries[4].GetData(),kventryByte5)
    assert.Equal(t, len(r2.Log.entries), 5)

    assert.Equal(t, r1.Log.entries[2].GetTerm(),int64(1))
    assert.Equal(t, r1.Log.entries[2].GetIndex(),int64(2))
    assert.Equal(t, r1.Log.entries[2].GetData(),kventryByte3)


    assert.Equal(t, r1.Log.entries[4].GetTerm(),int64(2))
    assert.Equal(t, r1.Log.entries[4].GetIndex(),int64(4))
    assert.Equal(t, r1.Log.entries[4].GetData(),kventryByte5)
    assert.Equal(t, len(r1.Log.entries), 5)

    assert.Equal(t, r3.Log.entries[4].GetTerm(),int64(2))
    assert.Equal(t, r3.Log.entries[4].GetIndex(),int64(4))
    assert.Equal(t, r3.Log.entries[4].GetData(),kventryByte5)
    assert.Equal(t, len(r3.Log.entries), 5)

    ...

}
```

检查出现各种日志不一致的情况时，附加后的结果是否与leader一致，测试发现都保持一致，即说明日志附加功能正确。

```
=== RUN   TestRaftAppend2
...
--- PASS: TestRaftAppend2 (0.07s)
```

日志选举限制测试

测试日志更加新的节点才能成为leader

```
// raft paper 5.4.1 Election restriction
// 只有日志更加新的才能成为leader
func TestRaftElectionRestriction(t *testing.T) {
```

```
    ....

    //term相同，日志较长的成为leader
    r1.Log.entries = append(r1.Log.entries, raftpb.Entry{Term: 0, Index: 0})


    r2.Log.entries = append(r2.Log.entries, raftpb.Entry{Term: 0, Index: 0})
    r2.Log.entries = append(r2.Log.entries, raftpb.Entry{Term: 0, Index: 1})

    r3.Log.entries = append(r3.Log.entries, raftpb.Entry{Term: 0, Index: 0})
    r3.Log.entries = append(r3.Log.entries, raftpb.Entry{Term: 0, Index: 1})

    r1.BecomeCandidate()
    r2.BecomeCandidate()


    for i:= 0;i < 5;i ++{
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
    }

    assert.Equal(t, r1.State, Follower)
    assert.Equal(t, r2.State, Leader)

    // term较新的成为leader，如果candidate的日志比较旧，拒绝投票
    r1.Log.entries[0].Term = 1
    r3.Log.entries[1].Term = 1

    r2.BecomeCandidate()
    r1.BecomeCandidate()


    for i:= 0;i < 5;i ++{
        r1.LogicalTimerChange()
        r2.LogicalTimerChange()
        r3.LogicalTimerChange()
    }

    assert.Equal(t, r1.State, Leader)
    assert.Equal(t, r2.State, Follower)
    ...
}
```

```
=== RUN   TestRaftElectionRestriction
2022/01/19 11:33:02 raft server listening at [::]:50051
2022/01/19 11:33:02 raft server listening at [::]:50052
2022/01/19 11:33:02 raft server listening at [::]:50053
2022/01/19 11:33:02 :50052 get vote from :50051
2022/01/19 11:33:02 :50051 get vote from :50052
--- PASS: TestRaftElectionRestriction (0.03s)
PASS
```

分布式存储节点单元测试

服务器端开启**三个**节点，客户端put{key:"hello",value:"world"}，检查数据库中是否存放有相应的数据，然后在其他非leader节点中调用get方法，检查是否能够get到相应的键值对，然后再插入一个新的键值对{key:"my_db",value:":ok"},再次get，检查是否能够获得新的键值对，然后测试删除，删除{key:"my_db",value:":ok"}键值对，然后调用get方法，检查是否返回键值对不存在的信息。

```
func TestServer(t *testing.T) {

    // 启动3个kvsever节点
    s1, kvserver1 := NewServer(&raft.Setting{NodeAddress: ":50051",
        HeartbeatTimeout: 1,
        ElectionTimeout:  3,
        PeerNodeAddress:  []string{":50052", ":50053"}},
        "mydb1",
        ":50041")

    s2, kvserver2 := NewServer(&raft.Setting{NodeAddress: ":50052",
        HeartbeatTimeout: 1,
        ElectionTimeout:  6,
        PeerNodeAddress:  []string{":50051", ":50053"}},
        "mydb2",
        ":50042")

    s3, kvserver3 := NewServer(&raft.Setting{NodeAddress: ":50053",
        HeartbeatTimeout: 1,
        ElectionTimeout:  10,
        PeerNodeAddress:  []string{":50051", ":50052"}},
        "mydb3",
        ":50043")

    // 逻辑时钟往后走10，使得选举出leader
    for i := 0; i < 10; i++ {
        kvserver1.r.LogicalTimerChange()
        kvserver2.r.LogicalTimerChange()
        kvserver3.r.LogicalTimerChange()
    }

    assert.Equal(t, kvserver1.r.State, raft.Leader)

    // 测试put方法
    r, err := kv_client.Put(":50041", &pb.RawPutRequest{Key: []byte("hello"),
Value: []byte("world")})
    if err != nil {
        log.Println(err)
    }

    log.Printf("error is: %s", r.GetError())

    // 检查节点的数据库当中是否存在数据
    get, err := kvserver1.db.Get([]byte("hello"), nil)
    if err != nil {
        log.Println(err)
    }
    assert.Equal(t, string(get), "world")

    get2, err := kvserver2.db.Get([]byte("hello"), nil)
    if err != nil {
        log.Println(err)
```

```
    }
    assert.Equal(t, string(get2), "world")

    // 检查调用Get方法
    r2, err := kv_client.Get(":50042", &pb.RawGetRequest{Key: []byte("hello")})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("value is %v, error is: %s", string(r2.GetValue()),
r2.GetError())
    assert.Equal(t, string(r2.GetValue()), "world")

    // 测试再次put一个新的键值对和再一次get
    r3, err := kv_client.Put(":50041", &pb.RawPutRequest{Key: []byte("my_db"),
Value: []byte("ok")})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("error is: %s", r3.GetError())

    r4, err := kv_client.Get(":50042", &pb.RawGetRequest{Key: []byte("my_db")})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("value is %v, error is: %s", string(r4.GetValue()),
r4.GetError())
    assert.Equal(t, string(r4.GetValue()), "ok")

    // 测试删除的用例
    r5, err := kv_client.Del(":50041", &pb.RawDelRequest{Key: []byte("my_db")})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("error is: %s", r5.GetError())

    r6, err := kv_client.Get(":50042", &pb.RawGetRequest{Key: []byte("my_db")})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    log.Printf("value is %v, error is: %s", string(r6.GetValue()),
r6.GetError())
    // 删除后，get返回找不到的错误信息
    assert.Equal(t, r6.NotFound, true)

    // 结束服务器线程
    kvserver1.r.StopRaft()
    kvserver2.r.StopRaft()
    kvserver3.r.StopRaft()

    s1.GracefulStop()
    s2.GracefulStop()
    s3.GracefulStop()
}
```

测试结果：

```
=== RUN   TestServer
```

```
2022/01/19 16:18:06 server listening at [::]:50041
2022/01/19 16:18:06 raft server listening at [::]:50051
2022/01/19 16:18:06 server listening at [::]:50042
2022/01/19 16:18:06 raft server listening at [::]:50052
2022/01/19 16:18:06 server listening at [::]:50043
2022/01/19 16:18:06 raft server listening at [::]:50053
2022/01/19 16:18:06 :50051 get vote from :50052
2022/01/19 16:18:06 Received: key is hello, value is world
2022/01/19 16:18:06 send to :50052, match index is -2 , Next index is 0
2022/01/19 16:18:06 send to :50052, match index is -1 , Next index is 0
2022/01/19 16:18:06 send to :50053, match index is -2 , Next index is 0
2022/01/19 16:18:06 send to :50053, match index is -1 , Next index is 0
2022/01/19 16:18:06 error is:
2022/01/19 16:18:06 Received: key is hello
2022/01/19 16:18:06 value is world, error is:
2022/01/19 16:18:06 Received: key is my_db, value is ok
2022/01/19 16:18:06 send to :50052, match index is -2 , Next index is 1
2022/01/19 16:18:06 send to :50052, match index is 0 , Next index is 1
2022/01/19 16:18:06 send to :50053, match index is -2 , Next index is 1
2022/01/19 16:18:06 send to :50053, match index is 0 , Next index is 1
2022/01/19 16:18:06 error is:
2022/01/19 16:18:06 Received: key is my_db
2022/01/19 16:18:06 value is ok, error is:
2022/01/19 16:18:06 Received: key is my_db
2022/01/19 16:18:06 send to :50052, match index is -2 , Next index is 2
2022/01/19 16:18:06 send to :50052, match index is 1 , Next index is 2
2022/01/19 16:18:06 send to :50053, match index is -2 , Next index is 2
2022/01/19 16:18:06 send to :50053, match index is 1 , Next index is 2
2022/01/19 16:18:06 error is:
2022/01/19 16:18:06 Received: key is my_db
2022/01/19 16:18:06 RawGet error is  leveldb: not found
2022/01/19 16:18:06 value is , error is:
--- PASS: TestServer (0.16s)
PASS
```

测试通过，节点的put，get，del功能实现。

总结

所实现的功能

- put，get，del功能
- raft选举机制
- raft日志复制
- 通过raft来保证不同节点数据库的一致性
- 多用户同时操作
- 单调写

可进一步的优化

- 日志的垃圾管理
- 缓存机制
- 日志持久化到硬盘

心得

通过编写键值对数据库，对raft协议有了更加深刻的理解，知道了分布式系统的初步实现是怎么样的，也熟练掌握了grpc的调用方法。

本次项目最大的困难是raft协议的实现，从定义需要的数据结构和grpc的接口时，就需要大量的时间构思，在实现时，也会出现许多bug或者一些细节，比如边界情况的处理。在测试时如果单纯人工来打印信息调试时不好调试的，所以我编写了单元测试，在实现下一个功能时都可以运行上次的单元测试来保证之前的功能是正确的，单元测试的结果也更加明显，能够清除地定位到错误的地方。

在完成项目后，提升了自己对于分布式系统的理解，提升了自己的编程能力和调试能力，同时也增长了编写单元测试的能力。