国光存储实习汇报

Author: 汪金璞

School: 华中科技大学

Email: itwangashao@qq.com

Version: v1.0

Date: 2020/1/12

Last-Modified: 2020/1/12

project1

project1就是要我们实现一个支持列族的单机键值存储系统, 指导书说列族貌似是在project4中会使用到

这个键值存储系统支持4种操作,分别是Put/Delete/Get/Scan

- Put操作其实就是我们熟悉的数据库的增和改, 当Put操作的键在数据库中不存在时, 就会执行增操作, 存在时, 就会执行改操作. !Note: 针对指定列族的指定键进行操作
- Delete就是执行删除操作, 删除指定列族的键的值
- Get就是查操作, 获得特定列族的键的值
- Scan就是扫描操作, 获取指定列族的一系列键的值

回到这个项目上来, 我们需要完成两个任务:

- 实现一个键值存储引擎
- 实现一个键值服务处理程序

一个存储引擎对外提供的接口如下所示:

```
type Storage interface {
   // Other stuffs
   Write(ctx *kvrpcpb.Context, batch []Modify) error
   Reader(ctx *kvrpcpb.Context) (StorageReader, error)
}
```

其中Write方法实现的就是对数据库的修改, 修改包括两个, 增, 改 和删除, 增和改属于Put操作, 而删属于Delete操作, 我们怎么知道执行的是什么操作呢? 通过Modify结构体, 我们可以看一下Modify结构体的定义:

```
type Modify struct {
   Data interface{}
}
```

它的里面有一个接口Data, 这里data传入的对象约定为两张类型, 如下所示:

```
type Put struct {
   Key   []byte
   Value []byte
   Cf    string
}

type Delete struct {
   Key []byte
   Cf  string
}
```

所以我们可以根据Modify结构体一个类型判断来知道此次Write操作具体执行什么操作.

我们再来看Reader方法, 重点关注它的返回值, 它返回的是一个StorageReader接口, 我们来看一下这个接口里面有什么东西:

```
type StorageReader interface {
   // When the key doesn't exist, return nil for the value
   GetCF(cf string, key []byte) ([]byte, error)
   IterCF(cf string) engine_util.DBIterator
   Close()
}
```

里面有三个方法, GetCF方法实现的就是Get操作, IterCF方法能够满足上层键值存储系统的Scan操作, 所以上层应用可以通过调用Reader返回一个reader对象, 然后操作这个对象的方法来对数据库进行读请求.

然后在具体实现上面需要注意的问题就是Reader方法需要我们使用badger.Txn来实现, 这个指导书里面提示了, 原因就是badger提供的事务处理能够提供一个保证了一致性的快照, 我们不用担心一致性的问题.

实现上调用的api主要就是util包里面的engine_util包里面的go文件里面的一些函数,这一步非常简单, 如果还有不懂的可以对照着测试文件来做

project2

project2aa

这一个项目就是要我们实现Raft一致性算法中的领导人选举, 需要我们填写的代码, 文档以及相关的有用的源代码全部在raft文件夹下面.

不过具体实现上面和raft算法还是有一些区别, 可能是为了测试更加方便吧, tinykv中采用的是逻辑时钟, 也就是一个叫做`tick`的函数来模拟时间, 上层应用会调用`RawNode.Tick`方法来驱动逻辑时钟, 此外, 消息的收发是一个异步的操作.

指导书上对这一部分的提示为从`raft.Raft.tick`函数开始编写, 这个函数的作用是驱动内部逻辑时钟摆动一个时钟, 来使得选举事件或者心跳事件发生超时. 一开始我对这个函数的实现就是简单的判断服务器状态, 然后再增加相应的heartbeatElapsed或者electionElapsed, 后来发现整个系统根本run不起来, 没有任何事件发生, 后来详细的阅读了一下指导书之后才发现, 这个tick函数可不简单哇, 我们需要在里面加入心跳超时判断以及选举超时判断的逻辑, 那么对应的事件触发了之后又会发生什么呢? 通过阅读文档以及测试函数, 我们可以知道有几个内部信息, 所谓内部信息, 就是这个类型的信息是不会通过网络发到其他机器的, 只是提醒本主机某个事件发生了, 举一个例子: MessageType_MsgBeat就属于内部消息, 它用于当发生心跳超时事件时, 通知本主机执行广播心跳包的操作, 通知是通过在tick函数中调用step函数做到的, 所以我们需要在step函数当中实现判断本主机类型以及判断消息类型的逻辑.

实现了tick函数之后, 我们接下来的可以选择做的事情就很多了, 比如我们可以把一些辅助函数先实现了, 比如那三个becomexxx函数, 其实逻辑都差不多, 都会对xxxElasped进行归零操作, 对状态赋新值, 将votes数组全部置为0等等, 但是也存在一些不同之处, 比如follower还需要对leader和term进行赋值, 而且还需要对lead和term传参. candidate自增term并且还需要向自己投票. leader目前没有什么特殊的, 不过等加入了log的逻辑之后, 它将会有特殊的地方.

然后我们可以开始处理一些消息请求, 我将挑一下我记得的重点来讲

- launchVote 发起选举请求

  就是候选人向其他节点发送选举请求, 希望它们投自己一票, 我们要做的就是构造一个msg, 然后将自己的一些信息放进去, 并且发给对方, 对方将根据一些条件判断是否接收自己的选举请求.

  这里需要注意一些比较毒瘤的情况, 比如单节点的情况, 一开始我是跳过了向自己发送请求包的选举请求, 但是如果只有一个节点, 那么会造成收不到response包的情况, 而我的很多逻辑都是放在处理voteResponse的函数里面的, 所以我的第一个尝试就是不跳过自己, 也向自己发送数据包, 但是后面还有一个测试点, 它过滤了发送给自己的数据, 准确来说, 它只允许了一些机器之间的通信, 这就导致我这种处理方法出现错误, 所以第二个尝试就是特判节点数量, 如果只有一个, 直接成为leader. 注意, 我们发送的选举请求的消息类型是MessageType_MsgRequestVote, 这个请求构造好之后我们直接把它append进raft结构体的msg数组里面即可

  > 'MessageType_MsgHup' is used for election. If a node is a follower or candidate, the 'tick' function in 'raft' struct is set as 'tickElection'. If a follower or candidate has not received any heartbeat before the election timeout, it passes 'MessageType_MsgHup' to its Step method and becomes (or remains) a candidate to start a new election.

- handleVote 处理选举请求

  主要根据下图的逻辑来实现

  接收者实现：

  1. 如果 term < currentTerm 返回 false （5.2 节）
  2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

  一个比较特殊的地方就是, 在2aa的测试里面混入了一个2ab的测试, 所以我们要想运行 `make project2aa` 获得PASS的结果, 我们还得加入一个简单的日志的处理逻辑, 下图是关于日志新旧的定义:

  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。

  对于投票逻辑的具体实现, 有如下几个步骤:

  1. 如果候选人的任期号比我们还要小, 那么直接丑拒
  2. 然后如果候选人的任期号比我们大, 那么我们要先执行becomeFollower的操作, 为什么要先执行这一步恐怕有点难以理解, 但是不这样就会出错,下面我尝试讲一下, 考虑这种情况, 如果第i轮里面有两个候选人平分秋色了, 加赛一轮, 那么投票的时候极有可能出现vote不为空, 然后无法投票的情况, 所以我们需要先执行这个, 当然也可能有其他原因.
  3. 然后比较日志的新旧, 如果不是至少一样新, 那么直接丑拒
  4. 最后判断自己是否已经投票了, 如果没有投票, 或者之前投票的那个人也是这个候选人, 那么继续投票给他
  5. 最后把信息append到msg里面即可
- handleVoteResponse 处理选举请求的回信

  这里有几个坑点

    - 节点的个数可能只有一个
    - 节点的个数可能是偶数
    - 如果拒绝的数量超过一半, 那么需要直接becomeFollower, 不然之后有一些测试会挂掉, 这个是我在做2ab的时候才知道的

  对于一个回应, 如果它投给了我们, 那么我们就把votes数组里面对应的位置设为true, 之后遍历一遍votes数组, 如果true的数量超过半数, 那么我们就成为了leader了

- broadcastHeartBeat leader广播心跳包

  当心跳超时的时候, 本机会发送beat消息到自己, 然后step函数中将会进入广播心跳包的逻辑, 调用broadcastHeartBeat函数, 这个函数遍历所有的peers, 然后对于每一个peer, 调用sendHeartbeat发送心跳包, 注意跳过自己, 之后需要重置计时器

  > 'MessageType_MsgBeat' is an internal type that signals the leader to send a heartbeat of
  > the 'MessageType_MsgHeartbeat' type. If a node is a leader, the 'tick' function in
  > the 'raft' struct is set as 'tickHeartbeat', and triggers the leader to
  > send periodic 'MessageType_MsgHeartbeat' messages to its followers.

- sendHeartbeat 发送心跳包到指定的peer

  这个函数的实现非常简单, 把From, to, term, MsgType填了, 然后append进msgs里面就可以了

- handleHeartbeat 处理leader发送过来的心跳请求

  如果term < currentTerm, 那么就返回一条reject为true的信息, 并且term填入自己的term, 否则的话调用becomeFollower函数, 然后msg的reject为false, append进msgs里面就可以了

- handleHeartbeatResponse 处理心跳包的回信

  被拒绝了那么就becomeFollower

- handleAppendEntries 处理附加日志请求

  1. 如果 `term < currentTerm` 就返回 false （5.1 节）
  2. 如果日志在 prevLogIndex 位置处的日志条目的任期号和 prevLogTerm 不匹配，则返回 false （5.3 节）
  3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的 （5.3 节）
  4. 附加日志中尚未存在的任何新条目
  5. 如果 `leaderCommit > commitIndex`，令 commitIndex 等于 leaderCommit 和 新日志条目索引值中较小的一个

  目前其实只需要实现第一条就可以了

其他的函数其实也都蛮简单的, 这里就不多加赘述了

另外需要注意

- 选举超时时间应该是一个随机值, 但是Raft结构体里面给的electionTimeOut却是一个定值, 所以我又在结构体里面多定义了一个变量, electionRandomTimeOut, 用来存储随机时间, 这个值在每一次发生选举超时之后发生更新, 更新的范围为[electionTimeout: electionTimeout*2), 为什么是这个范围呢?因为测试程序要求的就是这个范围.
- votes数组的重置时机: votes数组一定要记得重置, 否则我们在统计票数的时候会出现统计错误的情况, 我的处理是在每一次进行状态改变的时候重置, 这里注意一种情况的处理, 就是候选人出现平票的时候, 这样会再选举一轮, 选举超时时会收到hup消息, 我在对应的处理逻辑里面直接调用了becomeCandidate函数, 所以votes数组一样会清零
- vote成员的赋值: 这个成员变量是很重要的, 因为它可以保证一个任期内一个成员只投一张票, 所以不要忘了它的赋值, 另外一个需要注意的点就是在判断vote之前需要先判断候选人与自己的任期号的大小关系, 如果候选人额更大, 那么我们需要becomeFollower, 在becomeFollower的逻辑里面涉及将vote赋值为None的操作, 这样就会将票投给任期号更大的节点

总结: 这一部分现在看来其实很简单, 细节也并不多, 但是奠定了整个raft算法的代码框架, 所以需要注重一下代码的格式, 可维护性, 另外需要注意一些有用的调试手段:

- Dprintf
- 应该在哪些地方打log, 怎样打log更方便自己分析日志
- 二分定位错误法

project2ab

这一部分是实现日志的备份逻辑, 难度比前面的要大很多, 因为RaftLog这个结构体本身就很复杂, 而且处理日志的时候下标操作非常容易出现多一个1或者少一个1这样的错误, 此外这里还有一个坑点, 就是index不是entries数组的下标, 而是日志的下标, 这里绕了我半天

这一部分在2aa的基础之上加入了日志的操作, 所以有一些地方需要做出对应的修改

- 在newRaft函数里面对于RaftLog字段的初始化我们需要调用newLog方法, 我一开始是直接自己写了一个函数初始化, 结果没想到测试程序里面调用了newLog函数, 所以还是把初始化放在这个函数里面比较好. newLog函数根据指定的storage返回log, 这里的重点在于注释里面的这一句话:

  > It recovers the log to the state that it just commits and applies the latest snapshot.

  下面我来介绍一下几个重点字段:

  - storage保存自上次快照以来的所有稳定的条目, 应该就是已经被保存到大多数节点并且提交了的条目, storage里面有一个很有意思的地方就是它的FirstIndex()-1处的条目是一个空的东西, 为什么这样设计呢?因为为了匹配的方便, 关于这一点我们可以看storage.go的Term方法的注释

    > Term returns the term of entry i, which must be in the range [FirstIndex()-1, LastIndex()]. The term of the entry before FirstIndex is retained for matching purposes even though the rest of that entry may not be available.

    关于具体的实现我们可以看代码:

    ```
    // NewMemoryStorage creates an empty MemoryStorage.
    func NewMemoryStorage() *MemoryStorage {
      return &MemoryStorage{
        // When starting from scratch populate the list with a dummy entry at term zero.
        // 非常nice, 这里表面log的下标其实是从1开始的
        ents:    make([]pb.Entry, 1),
        snapshot: pb.Snapshot{Metadata: &pb.SnapshotMetadata{ConfState:
    &pb.ConfState{}}},
      }
    }
    ```

    关于接口Storage的一个实现就是MemoryStorage, 它有一些成员和方法, 主要介绍如下所示:

    - hardState: 包含了节点需要被持久化的一些状态, 比如当前的任期号, commit以及vote, 所谓硬状态就是不容易改变的, 所以需要持久化到storage当中

      > HardState contains the state of a node need to be peristed, including the current term, commit index and the vote record

      不过有一说一, 这里和论文出入很大, commit index应该是一个易变的状态

状态	所有服务器上持久存在的
currentTerm	服务器最后一次知道的任期号（初始化为 0，持续递增）
votedFor	在当前获得选票的候选人的 Id
log[]	日志条目集；每一个条目包含一个用户状态机执行的指令，和收到时的任期号

- snapShot: 一个Data和一个snapShotMetaData结构体, snapShotMetaData保存的是一些元数据信息, 比如最后一条被应用到这个快照的日志的index与任期号, 另外还有一个confState结构体, 这个结构体里面保存的是关于当前集群的成员信息

```
// SnapshotMetadata contains the log index and term of the last log applied to this
// Snapshot, along with the membership information of the time the last log applied.
type SnapshotMetadata struct {
  ConfState        *ConfState `protobuf:"bytes,1,opt,name=conf_state,json=confState" json:"conf_state,omitempty"`
  Index            uint64    `protobuf:"varint,2,opt,name=index,proto3" json:"index,omitempty"`
  Term             uint64    `protobuf:"varint,3,opt,name=term,proto3" json:"term,omitempty"`
  XXX_NoUnkeyedLiteral struct{}  `json:"-"`
  XXX_unrecognized     []byte    `json:"-"`
  XXX_sizecache        int32     `json:"-"`
}
// ConfState contains the current membership information of the raft group
type ConfState struct {
   // all node id
   Nodes            []uint64 `protobuf:"varint,1,rep,packed,name=nodes" json:"nodes,omitempty"`
   XXX_NoUnkeyedLiteral struct{} `json:"-"`
   XXX_unrecognized     []byte   `json:"-"`
   XXX_sizecache        int32    `json:"-"`
}
```

- ents: 这个就是storage里面已经持久化的一些日志的信息, 需要注意的最关键的一点就是;

> ents[i] has raft log position i+snapshot.Metadata.Index

什么意思呢? 这里需要理解快照的概念:

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧的信息，那么会带来可用性问题。

快照是最简单的压缩方法。在快照系统中，整个系统的状态都以快照的形式写入到稳定的持久化存储中，然后到那个时间点之前的日志全部丢弃。快照技术被使用在 Chubby 和 ZooKeeper 中，接下来的章节会介绍 Raft 中的快照技术。

就是这个之前的日志全部被扔了, 此下标并不是实际的下标, 它前面其实还有, 只不过被一个快照代替了

不过可能你对快照还是理解不够深刻, 没事, 论文为我们准备了一个例子:

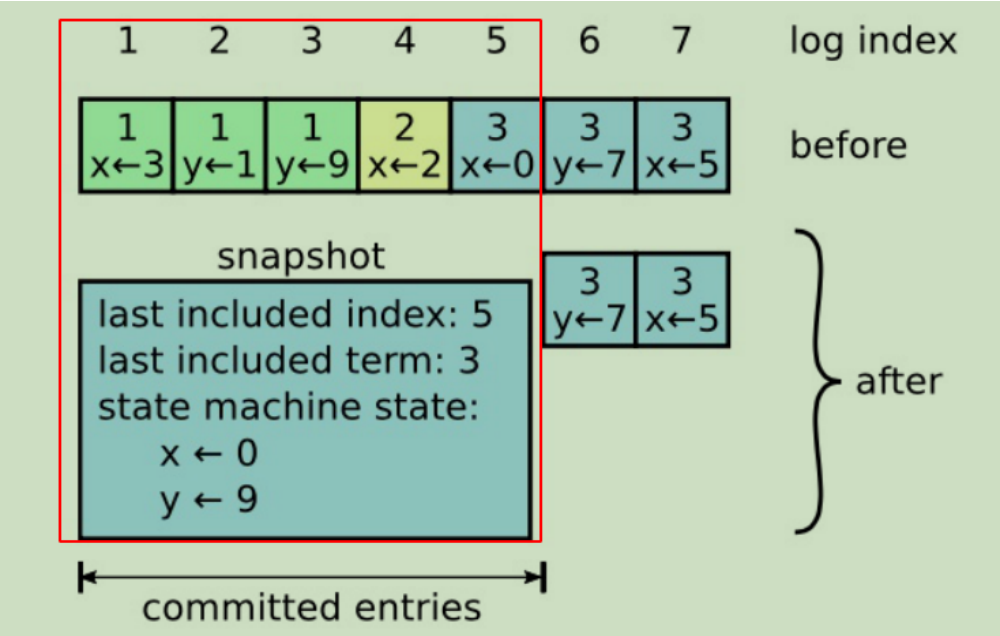


图 12：一个服务器用新的快照替换了从 1 到 5 的条目，快照值存储了当前的状态。快照中包含了最后的索引位置和任期号。

注意看我红色框框出来的部分, 5个日志条目被压缩成一个快照, 因为这个五个操作最终造成的结果就是x←0, y←9, 所以直接保存最终状态就可以了.

> 图 12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft 也包含一些少量的元数据到快照中：最后被包含索引指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），最后被包含的任期指的是该条目的任期号。保留这些数据是为了支持快照后紧接着的第一个条目的附加日志请求时的一致性检查，因为这个条目需要前一日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，他就可以删除最后索引位置之前的所有日志和快照了。

我在这介绍这些只是为了更加深刻的理解RaftLog结构体, 虽然现在用不着, 但是避免做的时候感到膈应, 我在做这个的时候就因为这个结构体没理解透彻踩了好多坑.

不过其实说到底这个memoryStorage还是一个在内存当中实现的storage, 至于在磁盘上的是啥样的, 这里先// TODO一下, 之后再写.

关于storage, 还有几个很有用的方法, 不过注释写的已经很详细, 我就直接贴过来了

```
// InitialState returns the saved HardState and ConfState information.
InitialState() (pb.HardState, pb.ConfState, error)
// Entries returns a slice of log entries in the range [lo,hi).
// MaxSize limits the total size of the log entries returned, but
// Entries returns at least one entry if any.
Entries(lo, hi uint64) ([]pb.Entry, error)
// Term returns the term of entry i, which must be in the range
// [FirstIndex()-1, LastIndex()]. The term of the entry before
// FirstIndex is retained for matching purposes even though the
// rest of that entry may not be available.
Term(i uint64) (uint64, error)
// LastIndex returns the index of the last entry in the log.
LastIndex() (uint64, error)
// FirstIndex returns the index of the first log entry that is
// possibly available via Entries (older entries have been incorporated
// into the latest Snapshot; if storage only contains the dummy entry the
// first log entry is not available).
FirstIndex() (uint64, error)
// Snapshot returns the most recent snapshot.
```

```
// If snapshot is temporarily unavailable, it should return ErrSnapshotTemporarilyUnavailable,
// so raft state machine could know that Storage needs some time to prepare
// snapshot and call Snapshot later.
Snapshot() (pb.Snapshot, error)
```

- 关于stabled变量, 注释里面是这样描述的

  > log entries with index <= stabled are persisted to storage. It is used to record the logs that are not persisted by storage yet.

  也就是说stabled是storage里面最后一个被持久化的条目的index, 使用这个变量, 可以区分已经被持久化的条目和没有被持久化的条目

- 关于applied变量, 注释里面是这样描述的

  > applied is the highest log position that the application has been instructed to apply to its state machine. Invariant: applied <= committed

  applied是已经被指示应用于状态机的最高的日志条目的index, 并且满足一个约束条件: applied <= committed, 也就是说只能应用已经提交的日志条目

- 关于commited变量, 就是已经提交的日志中Index最大的那一个, 注意提交需要满足的条件, 一是这个日志在大多数服务器上, 二是leader只能提交自己任期内的日志条目, 但是如果提交了自己任期的条目, 这个条目之前的也会被顺带提交

- 那么commited与stabled之间的关系是什么样的呢? 就是已经提交的条目什么时候会被持久化到storage呢.

  > 2ac里不是要生成一个ready结构体给上层吗
  >
  > 其中有一个成员是未持久化的entries
  >
  > 上层持久化后会返回到Raft模块
  >
  > Raft模块会更改stabled的值

一副有用的图:

```
// RaftLog manage the log entries, its struct look like:
//
//  snapshot/first.....applied....committed....stabled.....last
//  --------|------------------------------------------------|
//                            log entries
```

其实还有许多我也不是很清楚的地方, 先留个坑, 之后搞清楚了再过来修改// TODO

- 根据测试用例, 我们还需要在newRaft里面初始化Vote, Term, 以及commited的值, 这些值是存储在hardState里面的, 所以我们需要调用storage的InitialState方法获得hardState, 然后再初始化对应的值
- 增加了日志的操作之后, AppendEntry信息变得异常的复杂, 并且加入了AppendEntry的逻辑之后, 有一些地方还得加上一些特殊的处理, 不然可能会导致2aa里面的一些测试点挂掉, 下面我来重点谈一谈引入了日志之后的处理当中需要注意的一些地方

  - > The tests assume that the new elected leader should append a noop entry on its term

    这是指导书里面的一条提示, 意思就是当一个节点在一个新的任期赢得选举成为领导人之后, 他就立马在自己的entries当中添加一条空的日志条目项, 然而2aa中有一个cycle的测试点, 就是三台主机, 首先1发起选举, 成为leader, 然后2发起选举, 成为leader, 最后3发起选举, 成为leader. 我们添加了日志之后, 这个测试点就会挂掉, 因为1成为leader之后会导致它的日志更

加新, 导致拒绝掉其他人的选票, 最后3选举时, 1, 2都会拒绝它, 所以测试失败, 对此, 解决办法就是当获得一条新日志的时候, 立刻广播AppendEntry包.

- leader是通过收到MessageType_MsgPropose的信息来知道需要在自己的entries当中添加新的logs并且将它们广播的的, 于是收到这条消息对应的处理是先调用appendEntry方法, 把这个条目应用到自己的entries里面, 然后直接广播AppendEntry, 至于对于每一个peer, 传给它的Message里面对应的Entries字段的日志是什么, 是由`r.Prs[ro].Next`决定的, 这个Next代表的就是下一个需要发给这台主机的日志的index, 也就是切片[next:]
- 既然说到了ProPose, 那么我们就来谈一谈这个消息类型, 注释里面说的很清楚了, 我就不多加赘述了

> 'MessageType_MsgPropose' proposes to append data to its log entries. This is a special
> type to redirect proposals to leader. Therefore, send method overwrites
> eraftpb.Message's term with its HardState's term to avoid attaching its
> local term to 'MessageType_MsgPropose'. When 'MessageType_MsgPropose' is passed to the leader's 'Step'
> method, the leader first calls the 'appendEntry' method to append entries
> to its log, and then calls 'bcastAppend' method to send those entries to
> its peers. When passed to candidate, 'MessageType_MsgPropose' is dropped. When passed to
> follower, 'MessageType_MsgPropose' is stored in follower's mailbox(msgs) by the send
> method. It is stored with sender's ID and later forwarded to leader by
> rafthttp package.

- message里面的index并不是指entries的数组下标, 这一点非常坑, 在这里很容易犯例如数组下标越界的错误, 对此, 我的做法是先获得数组的下标为0的元素的index, 然后用Message里面的index减去这个就得到了数组的下标了, 当然, 这样做存在不足之处, 就是如果entries为空的情况, 所以后来我又封装了一个函数firstIndex用来处理这种情况.
- becomeLeader函数需要大改, 我们需要在这个里面初始化Next与Match, 这两个值的介绍如下所示

状态	在领导人里经常改变的 （选举后重新初始化）
nextIndex[]	对于每一个服务器，需要发送给他的下一个日志条目的索引值（初始化为领导人最后索引值加一）
matchIndex[]	对于每一个服务器，已经复制给他的日志的最高索引值

对于match, 我初始化为0了, 对于获取领导人最后的索引值, 在log.go里面有一个LastIndex方法, 这个是需要自己填充实现的, 我们直接调用这个方法就可以获得lastindex, 然后+1赋值给next就好, 这里不用担心next的值不对的情况, 因为next提供了机制让我们通过回信的反馈结果调整我们的next的值.

- 对于appendEntry方法, 我觉得最主要的一点就是term以及index要设置正确, 另外, 由于Message里面存储的entry指针, 所以我们需要通过遍历然后解引用的方式把这个entry给append 到我们的entries里面, 此外还有一点需要注意的就是, 每一次添加新的条目之后都需要更新自己的next以及match的值, 不然有一个测试会挂掉.
- 对于handleAppendEntries, 也就是接收方对AppendEntry的处理函数也是比较复杂的, 它收到的信息里面有两个字段是与2aa里面不同的, 一个叫做Logterm, 代表的是prevLogTerm, 一个是Index, 代表的是prevLogIndex, 这是什么意思呢? 就是为了让接受者判断prevLogIndex处的条目的term号是否保持一致, 保持一致就说明匹配成功, 否则就说明匹配不成功. 这里比较的时候需要注意一个请况, 就是接受者的日志没有那么长, 也就是说prevLogIndex比接受者的lastIndex还要大, 那么这种情况是直接拒绝的.

此处遵循了论文里面的这个规则.

接收者实现：

1. 如果 term < currentTerm 就返回 false （5.1 节）
2. 如果日志在 prevLogIndex 位置处的日志条目的任期号和 prevLogTerm 不匹配，则返回 false （5.3 节）
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的 （5.3 节）
4. 附加日志中尚未存在的任何新条目
5. 如果 leaderCommit > commitIndex，令 commitIndex 等于 leaderCommit 和 新日志条目索引值中较小的一个

依照的是下述原则:

日志匹配原则	如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这个日志从头到这个索引位置之间全部完全相同（5.3 节）

这里论文给出了证明, 有兴趣的可以自己去看一下论文即可.

- 我们返回给leader的信息里面在2ab中最重要的就是Index字段, 以及reject字段, 如果reject==false, 代表匹配成功, 那么leader就会用index更新对这个主机的match值以及next值, 否则会减少next重传, 目前的处理就是next-1, 不过论文里面给出了一种优化, 开会的时候有同学也说到了之后的project里面也会涉及到这种优化:

  如果需要的话，算法可以通过减少被拒绝的附加日志 RPCs 的次数来优化。例如，当附加日志 RPC 的请求被拒绝的时候，跟随者可以包含冲突的条目的任期号和自己存储的那个任期的最早的索引地址。借助这些信息，领导人可以减小 nextIndex 越过所有那个任期冲突的所有日志条目；这样就变成每个任期需要一次附加条目 RPC 而不是每个条目一次。在实践中，我们十分怀疑这种优化是否是必要的，因为失败是很少发生的并且也不大可能会有这么多不一致的日志。

- 接下来我们讨论一下匹配成功了的情况, 这里是整个2ab最复杂的一个地方: 在这里我们需要从匹配的地方开始一个个的往后判断冲突, 如果发生了冲突, 那么我们就需要截断, 然后把leader发送过来的entries附加在后面, 如果没有冲突, 那么直接附加在最后面就可以了. 此处贴一个我代码里面的注释方便理解:

```
// 如果已经存在的日志条目和新的产生冲突(索引值相同但是任期号不同),删除这一条和之后所有的 (5.3 节)
// 我的个人理解就是如果rf.log下标为prevLogIndex+1的地方如果存在元素并且term与传过来的entries中的term不同,那么这个以及后面的全部不要
// fix bug: 这里应该需要一个个的比较,考虑以下情况:
//         ↓ (prevIndex)
// logs: 1 1 1 2 2 2 2
// args:     1 1 2 2 2 2
//             ↑ (term不相等, 之后的都切掉)
```

  所以对于entries为空的情况, 你觉得接收方在prevLogIndex之后的条目需要截断吗? 答案是No, 不需要, 因为没有发生冲突.

  这一块说起来轻松, 但编写代码的时候可一点都不轻松, 我花了好久, 不过主要原因是因为我最开始没有搞清楚RaftLog的结构, 是在测试用例当中不断更正我对这个结构体的认识的, 前前后后一共迭代了大概五版, 太辛酸了.

- 之后我们需要判断是否需要commit, 这一块主要根据的是测试用例TestHandleMessageType_MsgAppend2AB的注释的提示的第三条

  > TestHandleMessageType_MsgAppend ensures:
  >
  > 1. Reply false if log doesn’ t contain an entry at prevLogIndex whose term matches prevLogTerm.
  > 2. If an existing entry conflicts with a new one (same index but different terms),
  > delete the existing entry and all that follow it; append any new entries not already in the log.
  > 3. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry).

  这里还是需要调一会bug的, 有一点绕, 测试也比较严格

目前还只是知道如何commit, 但是还不知道如何apply, 先留个坑, 等做了后面的之后知道了再来补充// TODO

- 最后就是handleAppendEntriesResponse了, 这个是leader用来处理AppendEntry的回信的, 这里分两张情况进行讨论, 一种是接受, 一种是拒绝
  - 如果回信是拒绝, 那么可能有以下几种被拒绝的理由
    - 任期号太小了, 那么leader需要becomeFollower
    - 日志不匹配, 那么我们需要对next做出调整, 然后立刻对这台主机重发AppendEntry, 为什么立刻重发呢? 因为不立刻重发测试会挂掉(笑, 但究其根本原因是因为在tinykv当中把AppendEntry与心跳包当做两个东西来处理, 如果心跳包就是entries为空的AppendEntry包的话, 那么等心跳超时再根据next发送对应的entries页未尝不可. 我是这么觉得的.
  - 如果回信是接受,那么我们需要更新next与match数组, match就更新为回信的值就行, next就更新为match+1, 之后我们需要判断是否有新的超过半数的提交, 对此我的解决方案是将所有peer的match都append到一个数组当中, 然后找到其中超过半数的最大的commit, 如果这个commit值比原来的大并且term号等于当前的任期号, 那么就可以提交, 记得从上次applied的地方开始提交, 另外当leader的commit更新之后, 我们还需要广播AppendEntry, 告诉其他节点更新commit, 不然这个测试用例会挂掉TestLeaderSyncFollowerLog2AB

- 对于log.go文件, 我们也需要在里面填充几个函数, 而且为了方便, 我还自己封装了一些函数在这个文件里面, 这个文件里面的函数非常容易搞错, 所以一定要细心一点
- 对于storage.go文件, 主要就是阅读一下就可以了, 倒不用填写啥代码
- 最最后就是阅读测试文件以及测试文件的注释了, 这是理解这个项目的一个非常好的途径

完

project2ac

首先贴上学长对这一部分的提示:

> 实现Raft模块与上层交互的相关函数
>
> Raft模块通过返回一个ready结构体将要发送的消息、要提交的日志条目、当前状态返回给上层进行处理。
>
> a．ready
>
> ready函数填充ready结构体相应数据，返回ready结构体
>
> b.HasReady
>
> 判断是否rn.Raft.RaftLog.unstableEntries()、rn.Raft.RaftLog.nextEnts()、rn.Raft.msgs是否都为空，若都为空返回false
>
> c.Advance
>
> 通过rawnode里的ready中的Entrie等数据来修改raft中的applied index, stabled log index等数据

实验指导书对这一部分的提示也还蛮多的, 这一部分我们实现的就是一个与上层应用交互的接口, 它提供了一些包装函数.

在这一部分一个比较重要的结构体就是Ready, 不过在这个结构体的声明当中, 注释已经写的很详细了, 所以我也不多加赘述了

```
type Ready struct {
  // The current volatile state of a Node.
```

```
    // SoftState will be nil if there is no update.
    // It is not required to consume or store SoftState.
    *SoftState

    // The current state of a Node to be saved to stable storage BEFORE
    // Messages are sent.
    // HardState will be equal to empty state if there is no update.
    pb.HardState

    // Entries specifies entries to be saved to stable storage BEFORE
    // Messages are sent.
    // 这个就是存储将要被持久化到stable storage中的日志条目, 所以之后的advaance操作会用这个里面的最后一条日志的index来更新stabled
    Entries []pb.Entry

    // Snapshot specifies the snapshot to be saved to stable storage.
    Snapshot pb.Snapshot

    // CommittedEntries specifies entries to be committed to a
    // store/state-machine. These have previously been committed to stable
    // store.
    // 这个就是存放已经提及但是还没有被应用的, 所以advace会用这个的最后一条的日志的index来更新applied
    CommittedEntries []pb.Entry

    // Messages specifies outbound messages to be sent AFTER Entries are
    // committed to stable storage.
    // If it contains a MessageType_MsgSnapshot message, the application MUST report back to raft
    // when the snapshot has been received or has failed by calling ReportSnapshot.
    Messages []pb.Message
}
```

下面简单的谈一谈我们需要实现的几个函数

- Ready函数

  返回的就是Ready结构体, 值得注意的一点是softState与hardState如果没有发生改变, 那么它们的赋值应该为空或者nil, 所以我们需要在Rawnode结构体中定义一些变量记录一下之前的状态, 方便我们做比较, 在这里我的设计是:

  ```
  //因为softstate与hardState的值取决于是否发生变化, 所以我们得保存它之前的值, 然后来比较是否发生变化
  prevSoftState *SoftState
  prevHardState pb.HardState
  ```

  然后我们需要两个辅助函数, 就是比较当前的hardStaete, softState与之前的是否相等的函数, 其实在util.go里面框架代码已经帮我们实现好了一个. 有了这些之后, 我们在这个函数当中要做的主要工作就是给ready结构体的字段进行赋值

  - SoftState如果当前状态与之前相同, 赋值为nil, 否则, 赋值为当前的状态
  - HardState如果当前状态与之前相同, 赋值为空结构体, 否则, 赋值为当前的状态
  - Entries 里面存放的是还没有被持久化到storage里面的日志条目
  - CommittedEntries里面存放的是已经提交但是还没有被应用到状态机里面的日志条目
- NewRawNode函数

  > NewRawNode returns a new RawNode given configuration and a list of raft peers.

首先我们按照给定的config生成一个raft实例, 然后获取hardState, 之后初始化prevHardstate, prevSoftState, 最后返回rawNode指针即可

- HasReady函数

  这个函数的实现不难, 但是我也是看了学长的提示之后才知道是要这样判断, 其实指导书以及注释并没有说的很清楚.

- Advance函数

  这函数的实现很难, 我花了好久才搞懂, 一开始测试出错的时候我甚至一度怀疑我是不是之前那几个变量的含义都搞错了.

  下面来谈一谈实现, 他要我们做的就是更新两个变量, stabled, applied的值, 具体做法如下所示:

```
if len(rd.Entries) > 0 {
   rn.Raft.RaftLog.stabled = rd.Entries[len(rd.Entries)-1].Index
}
if len(rd.CommittedEntries) > 0 {
   rn.Raft.RaftLog.applied = rd.CommittedEntries[len(rd.CommittedEntries)-1].Index
}
```

  原因

  > Entries []pb.Entry: 这个就是存储将要被持久化到stable storage中的日志条目, 所以之后的advaance操作会用这个里面的最后一条日志的index来更新stabled
  > CommittedEntries []pb.Entry: 这个就是存放已经提及但是还没有被应用的, 所以advace会用这个的最后一条的日志的index来更新applied

至此, 我们就圆满的完成了project2ac了. `make project2a` 通关截图如下所示:

```
--- PASS: TestAllServerStepdown2AB (0.00s)
=== RUN   TestCandidateResetTermMessageType_MsgHeartbeat2AA
--- PASS: TestCandidateResetTermMessageType_MsgHeartbeat2AA (0.00s)
=== RUN   TestCandidateResetTermMessageType_MsgAppend2AA
--- PASS: TestCandidateResetTermMessageType_MsgAppend2AA (0.00s)
=== RUN   TestDisruptiveFollower2AA
--- PASS: TestDisruptiveFollower2AA (0.00s)
=== RUN   TestHeartbeatUpdateCommit2AB
--- PASS: TestHeartbeatUpdateCommit2AB (0.00s)
=== RUN   TestRecvMessageType_MsgBeat2AA
--- PASS: TestRecvMessageType_MsgBeat2AA (0.00s)
=== RUN   TestLeaderIncreaseNext2AB
--- PASS: TestLeaderIncreaseNext2AB (0.00s)
=== RUN   TestCampaignWhileLeader2AA
--- PASS: TestCampaignWhileLeader2AA (0.00s)
=== RUN   TestSplitVote2AA
--- PASS: TestSplitVote2AA (0.00s)
=== RUN   TestRawNodeStart2AC
--- PASS: TestRawNodeStart2AC (0.00s)
=== RUN   TestRawNodeRestart2AC
--- PASS: TestRawNodeRestart2AC (0.00s)
PASS
ok      github.com/pingcap-incubator/tinykv/raft        0.022s
➜  tinykv git:(course) ✗
```

project2B

调试了好几天,终于解决了两个panic的bug

- 

这个bug的产生原因是我在之前的project里面在赋初始值的时候对snapshot赋值了,也就是下面这行代码

```
snapshot, _ := storage.Snapshot()
```

然后删了就好了, 之后我就把之前的project里面所有当前不必要但是由于goland的自动fill all fields的功能填写的字段全部删除了, 不过我之后又想了一下, 其实goland自动填充的应该没有影响, 反正都是赋的默认初值嘛, 不过删了也好, 不然代码太长了不便于阅读调试.

- 上面那个bug修好之后,又出现了一个新的bug

```
panic: find no region for 30203030303030303030

goroutine 296 [running]:
github.com/pingcap-incubator/tinykv/kv/test_raftstore.(*Cluster)
```

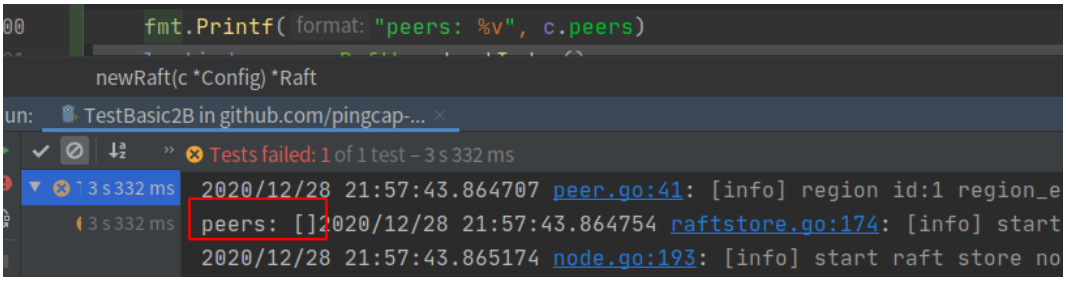
在调这个bug的过程当中发现了不少2a里面的bug

  - rawnode的ready函数没有给Message赋值, 导致消息没法递交给上层
  - 由于2b的测试里面的new一个peer的时候, cfg里面其实是没有给peers初始化的(发现没,我换主题啦,还是黑色对眼睛更加友好呢qwq)

```
raftCfg := &raft.Config{
    ID:              meta.GetId(),
    ElectionTick:    cfg.RaftElectionTimeoutTicks,
    HeartbeatTick:   cfg.RaftHeartbeatTicks,
    Applied:         appliedIndex,
    Storage:         ps,
}

raftGroup, err := raft.NewRawNode(raftCfg)
```

所以我们的peers得到的东西就会是一个空



所以我们就会出现找不到region的问题, 解决办法就是从confState里面读取

- 我还发现一个神奇的bug, 你看当我的msgs这样初始化时, rawnode的第二个测试点就会离谱的挂掉

```
// 这个bug藏得太深了,空数组应该是nil,不应该是[],我也不知道为什么
msgs:              []pb.Message{},
```

```
=== RUN   TestRawNodeRestart2AC
--- FAIL: TestRawNodeRestart2AC (0.00s)
    rawnode_test.go:212: g = {SoftState:<nil> HardState:{Term:0 Vote:0 Commit:0 XXX_NoUnkeyedLiteral:{} XXX_unre
                  w   {SoftState:<nil> HardState:{Term:0 Vote:0 Commit:0 XXX_NoUnkeyedLiteral:{} XXX_unrecogr
FAIL
```

这里可能看不清楚, 离谱的就在于g和w其实一模一样,我粘贴下来

```
=== RUN   TestRawNodeRestart2AC
--- FAIL: TestRawNodeRestart2AC (0.00s)
    rawnode_test.go:212: g = {SoftState:<nil> HardState:{Term:0 Vote:0 Commit:0 XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0} Entries:[] Snapshot:{Data:[] Metadata:<nil> XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0} CommittedEntries:[{EntryType:EntryNormal Term:1 Index:1 Data:[] XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0}] Messages:[]},
                w  {SoftState:<nil> HardState:{Term:0 Vote:0 Commit:0 XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0} Entries:[] Snapshot:{Data:[] Metadata:<nil> XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0} CommittedEntries:[{EntryType:EntryNormal Term:1 Index:1 Data:[] XXX_NoUnkeyedLiteral:{} XXX_unrecognized:[] XXX_sizecache:0}] Messages:[]}
FAIL
```

实在是太离谱了, 但是换成nil就好了, 于是我把之前一些初始化为空数组的全部换成了nil, 我对go还是不够熟练啊.

解决完上面的之后, 一切就都恢复正常啦!

- 在做2C的时候, 遇到了一个难以理解的bug, 但是感觉这个bug不像是2C的问题, 于是又回过头看2B去了

```
2021/01/02 13:48:04.944271 test_test.go:245: [info] 3: client scan 3 00000000-3 00000001

2021/01/02 13:48:04.944323 test_test.go:246: [fatal] get wrong value, client 3
want:x 3 0 y
got: x 3 0 yx 3 0 yx 3 0 yx 3 0 y
```

就是在scan的时候,这个范围的键值只有一个, 但是却打印出了多个

然后我又加了一些调试信息, 发现了问题没有, 48后面应该就是49,然后到达右边界停止scan才对, 但是我的48的下一个居然是一个诡异的值, 诡异的值后面居然还有值

```
outer***
inner***: iter.Item().Key() [51 32 48 48 48 48 48 48 48 48] end [51 32 48 48 48 48 48 48 48 49]
debug: value: [120 32 51 32 48 32 121], values: [[120 32 51 32 48 32 121]]
inner***: iter.Item().Key() [51 32 48 48 48 48 48 48 48 48 255 255 255 255 255 255 255 250] end [51 32 48 48 48 48 48 48 48 49]
debug: value: [120 32 51 32 48 32 121], values: [[120 32 51 32 48 32 121] [120 32 51 32 48 32 121]]
inner***: iter.Item().Key() [51 32 48 48 48 48 48 48 48 48 255 255 255 255 255 255 255 255 250 255 255 255 255 255 255 255 249] end
debug: value: [120 32 51 32 48 32 121], values: [[120 32 51 32 48 32 121] [120 32 51 32 48 32 121] [120 32 51 32 48 32 121]]
inner***: iter.Item().Key() [51 32 48 48 48 48 48 48 48 48 255 255 255 255 255 255 255 255 250 255 255 255 255 255 255 255 249 255
debug: value: [120 32 51 32 48 32 121], values: [[120 32 51 32 48 32 121] [120 32 51 32 48 32 121] [120 32 51 32 48 32 121] [12
inner***: iter.Item().Key() [52 32 48 48 48 48 48 48 48 48] end [51 32 48 48 48 48 48 48 48 49]
```

于是我断定是put的时候出问题了, 所以我把put的处理给注释掉了

```
2021/01/02 13:51:50.007941 test_test.go:246: [fatal] get wrong value, client 2
want:x 2 0 y
got:
```

果然是put出问题了, 问题找到了接下来就好办多了. 最终发现bug的原因是: 我在每一次setCf或者deleteCf之后都进行了writeToDB的操作, 不过我每次writeToDB之后都重新new了一个新的writeBatch, 按道理清空了应该就没事了啊, 这里我也不太清楚为什么会setCf出这么诡异的key出来. 不过最后同意writetoDB肯定是没问题的.所以去掉这里就好了.

project2C

- 在做这个实验的过程中遇到了不少问题, 体现出来就是2C的第一个测试就过不了, 通过不断的打印日志可以知道原因是那个日志比较旧的节点并没有成功获取到leader节点的snapshot.下面我就来剖析一下2C的第一个测试

```
首先是初始化
然后是向整个集群当中的节点都添加这两个键值对
```

```
  cluster.MustPutCF(cf, []byte("k1"), []byte("v1"))
  cluster.MustPutCF(cf, []byte("k2"), []byte("v2"))
然后是一个测试点的判断
  MustGetCfEqual(cluster.engines[1], cf, []byte("k1"), []byte("v1"))
  MustGetCfEqual(cluster.engines[1], cf, []byte("k2"), []byte("v2"))
然后是测试一些状态的初始值是不是正确的
  for _, engine := range cluster.engines {
    state, err := meta.GetApplyState(engine.Kv, 1)
    if err != nil {
      t.Fatal(err)
    }
    if state.TruncatedState.Index != meta.RaftInitLogIndex ||
      state.TruncatedState.Term != meta.RaftInitLogTerm {
      t.Fatalf("unexpected truncated state %v", state.TruncatedState)
    }
  }
然后是阻断1与2, 3之间的通信
  cluster.AddFilter(
    &PartitionFilter{
      s1: []uint64{1},
      s2: []uint64{2, 3},
    },
  )
然后PutCF一堆, 注意, 在这个过程中会触发compact
  // write some data to trigger snapshot
  for i := 100; i < 115; i++ {
    cluster.MustPutCF(cf, []byte(fmt.Sprintf("k%d", i)), []byte(fmt.Sprintf("v%d", i)))
  }
  cluster.MustDeleteCF(cf, []byte("k2"))
  time.Sleep(500 * time.Millisecond)
然后又是一个判断, 这时由于1被隔离了, 所以肯定获取到的是None
  MustGetCfNone(cluster.engines[1], cf, []byte("k100"))
然后恢复1与2,3之间的通信
  cluster.ClearFilters()
然后又是一堆测试
  // Now snapshot must applied on
  MustGetCfEqual(cluster.engines[1], cf, []byte("k1"), []byte("v1"))
  MustGetCfEqual(cluster.engines[1], cf, []byte("k100"), []byte("v100"))
  MustGetCfNone(cluster.engines[1], cf, []byte("k2"))
下面还有很多, 不过我的代码的问题在上面的测试当中就可以暴露出来了
...
```

在经过了我一整天找2C里面的代码的bug之后, 在晚上心疲力竭之际我在raft.go里面加了几个log.Infof, 然后奇迹出现了, 我一直以为日志旧的那台机器收不到snapshot消息是因为我在2C里面写的消息传递有问题, 结果发现又是2A里面的BUG, 我吐了.

原因: 这个sendsnapshot必须在sendappend里面触发, 然而我在2A的实现中, sendappend只有在propose的时候才能触发, 这样在测试当中由于在恢复通信之后没有propose, 所以snapshot的发送触发不了, 所以解决方案是每次leader收到心跳包的回信的时候都sendappend一下给对方.

- 问题2: 应该在收到了snapshot, 并且snapshot的index比自己的commit index大的时候, 丢弃掉整个entries

```
2020/12/31 13:14:37.492643 raft.go:965: [info] 我这里确实保存了快照文件: data:"\n&\010\001\"\004\0
panic: runtime error: index out of range [255] with length 90

goroutine 107 [running]:
github.com/pingcap-incubator/tinykv/raft.(*RaftLog).Term(0xc1ac5f4000, 0x580, 0xcf4ee0, 0xc1ad9
    /home/wang/文档/Lab/tinykv/raft/log.go:175 +0x185
github.com/pingcap-incubator/tinykv/raft.(*Raft).handleAppendEntries(0xc1a7065600, 0x3, 0x2, 0x
    /home/wang/文档/Lab/tinykv/raft/raft.go:833 +0x398
github.com/pingcap-incubator/tinykv/raft.(*Raft).Step(0xc1a7065600, 0x3, 0x2, 0x1, 0x22, 0x22,
    /home/wang/文档/Lab/tinykv/raft/raft.go:745 +0xd1
```

因为这些旧的日志属于已经被compact的,需要扔掉, 不然之后调用Term函数以及Index函数会出大问题.

- stop集群的时候卡死了

  原因: send runner.RegionTaskApply task to region worker through PeerStorage.regionSched and wait until region worker finish.的时候没有加&, 导致卡死了

- 2C的最后一个最难的测试,在测试的时候有一段时间会出现一台机器一直输出errnotleader的信息, 也就是它不是leader, 但是客户仍然一直给他发raftCmd的情况

```
2021/01/02 20:03:14.704992 peer_msg_handler.go:342: [info] 这个错误的提交为header:<region_id:1 peer:<id:2 store_id:2 > region_epoch:<conf_ver:1 version:1
2021/01/02 20:03:14.705024 callback.go:23: [info] resp is not nil: header:<error:<not_leader:<region_id:1 > > >
2021/01/02 20:03:14.705044 cluster.go:254: [info] run here: leader: id:2 store_id:2 , region: 1
2021/01/02 20:03:14.705058 peer_msg_handler.go:309: [info] [[region 1] 2] ErrNotLeader, leaderId: 0
2021/01/02 20:03:14.705069 peer_msg_handler.go:341: [info] [[region 1] 2] err preProposeRaftCommand
2021/01/02 20:03:14.705094 peer_msg_handler.go:342: [info] 这个错误的提交为header:<region_id:1 peer:<id:2 store_id:2 > region_epoch:<conf_ver:1 version:1
2021/01/02 20:03:14.705112 callback.go:23: [info] resp is not nil: header:<error:<not_leader:<region_id:1 > > >
2021/01/02 20:03:14.705128 cluster.go:254: [info] run here: leader: id:2 store_id:2 , region: 1
2021/01/02 20:03:14.705141 peer_msg_handler.go:309: [info] [[region 1] 2] ErrNotLeader, leaderId: 0
2021/01/02 20:03:14.705152 peer_msg_handler.go:341: [info] [[region 1] 2] err preProposeRaftCommand
2021/01/02 20:03:14.705180 peer_msg_handler.go:342: [info] 这个错误的提交为header:<region_id:1 peer:<id:2 store_id:2 > region_epoch:<conf_ver:1 version:1
2021/01/02 20:03:14.705197 callback.go:23: [info] resp is not nil: header:<error:<not_leader:<region_id:1 > > >
2021/01/02 20:03:14.705213 cluster.go:254: [info] run here: leader: id:2 store_id:2 , region: 1
2021/01/02 20:03:14.705226 peer_msg_handler.go:309: [info] [[region 1] 2] ErrNotLeader, leaderId: 0
2021/01/02 20:03:14.705236 peer_msg_handler.go:341: [info] [[region 1] 2] err preProposeRaftCommand
2021/01/02 20:03:14.705258 peer_msg_handler.go:342: [info] 这个错误的提交为header:<region_id:1 peer:<id:2 store_id:2 > region_epoch:<conf_ver:1 version:1
2021/01/02 20:03:14.705275 callback.go:23: [info] resp is not nil: header:<error:<not_leader:<region_id:1 > > >
2021/01/02 20:03:14.705291 cluster.go:254: [info] run here: leader: id:2 store_id:2 , region: 1
2021/01/02 20:03:14.705304 peer_msg_handler.go:309: [info] [[region 1] 2] ErrNotLeader, leaderId: 0
2021/01/02 20:03:14.705314 peer_msg_handler.go:341: [info] [[region 1] 2] err preProposeRaftCommand
2021/01/02 20:03:14.705337 peer_msg_handler.go:342: [info] 这个错误的提交为header:<region_id:1 peer:<id:2 store_id:2 > region_epoch:<conf_ver:1 version:1
2021/01/02 20:03:14.705353 callback.go:23: [info] resp is not nil: header:<error:<not_leader:<region_id:1 > > >
```

经过我不断的啃源码, 然后输出调试信息, 终于知道了为什么会这样

看下面这段框架代码, 在CallCommandOnLeader函数当中. 考虑这种情景, 一台follower网络连接断开了, 然后它成了候选人, 但是它却当不了leader, 因为没有足够的选票, 于是它一直是候选人, 候选人的leader是None, 于是夏敏这段代码就会一直continue. 我觉得这里应该加一个选其他节点的策略, 不然按照框架代码这样, 或许只能一直请求这个候选人了.

```go
if resp.Header.Error != nil {
    // log.Infof("CallCommandOnLeader resp.header.error")
    err := resp.Header.Error
    if err.GetStaleCommand() != nil || err.GetEpochNotMatch() != nil || err.GetNotLeader() != nil {
        log.Debugf( format: "encouter retryable err %+v", resp)
        if err.GetNotLeader() != nil && err.GetNotLeader().Leader != nil {
            leader = err.GetNotLeader().Leader
        } else {
            leader = c.LeaderOfRegion(regionID)
            log.Infof( format: "run here: leader: %v, region: %v", leader, regionID)
        }
        continue
    }
}
```

`make project2c` 大概是能过测试了, 不过这个是随机测试, 能过一次不代表代码没有bugqwq, 我把2C里面最难的一个测试点也就是最后一个测试点单独跑了好几次, 都PASS了, 应该是没有大问题.

```
2021/01/02 20:23:52.663065 peer_msg_handler.go:135: [info] [[region 1] 1] 正在执行DeleteCF: key: [52 32 48 48 48 48 48 48 49 54]
2021/01/02 20:23:52.663119 peer_msg_handler.go:138: [info] [[region 1] 1] 写入kvDB
2021/01/02 20:23:52.663152 peer_msg_handler.go:156: [info] [[region 1] 1] 正在执行回调,返回response, command type: Delete
2021/01/02 20:23:52.663285 callback.go:23: [info] resp is not nil: header:<current_term:36 > responses:<cmd_type:Delete delete:<> >
2021/01/02 20:23:52.666285 peer_msg_handler.go:249: [info] [[region 1] 3] 正在process Entry: term:36 index:939 data:"\n\016\010\001\022\004\010\001\02
0\001\"\004\010\001\020\001\022\031\010\004*\025\n\007default\022\n4 00000016"
2021/01/02 20:23:52.666305 peer_msg_handler.go:249: [info] [[region 1] 2] 正在process Entry: term:36 index:939 data:"\n\016\010\001\022\004\010\001\02
0\001\"\004\010\001\020\001\022\031\010\004*\025\n\007default\022\n4 00000016"
2021/01/02 20:23:52.666431 peer_msg_handler.go:126: [info] [[region 1] 2] 解析出来的命令: cmd_type:Delete delete:<cf:"default" key:"4 00000016" >
2021/01/02 20:23:52.666431 peer_msg_handler.go:126: [info] [[region 1] 3] 解析出来的命令: cmd_type:Delete delete:<cf:"default" key:"4 00000016" >
2021/01/02 20:23:52.666480 peer_msg_handler.go:135: [info] [[region 1] 2] 正在执行DeleteCF: key: [52 32 48 48 48 48 48 48 49 54]
2021/01/02 20:23:52.666291 peer_msg_handler.go:249: [info] [[region 1] 4] 正在process Entry: term:36 index:939 data:"\n\016\010\001\022\004\010\001\02
0\001\"\004\010\001\020\001\022\031\010\004*\025\n\007default\022\n4 00000016"
2021/01/02 20:23:52.666594 peer_msg_handler.go:138: [info] [[region 1] 2] 写入kvDB
2021/01/02 20:23:52.666651 peer_msg_handler.go:101: [info] [[region 1] 2] proposals为空
2021/01/02 20:23:52.666665 peer_msg_handler.go:126: [info] [[region 1] 4] 解析出来的命令: cmd_type:Delete delete:<cf:"default" key:"4 00000016" >
2021/01/02 20:23:52.666726 peer_msg_handler.go:135: [info] [[region 1] 4] 正在执行DeleteCF: key: [52 32 48 48 48 48 48 48 49 54]
2021/01/02 20:23:52.666500 peer_msg_handler.go:135: [info] [[region 1] 3] 正在执行DeleteCF: key: [52 32 48 48 48 48 48 48 49 54]
2021/01/02 20:23:52.666792 peer_msg_handler.go:138: [info] [[region 1] 4] 写入kvDB
2021/01/02 20:23:52.666848 peer_msg_handler.go:101: [info] [[region 1] 4] proposals为空
2021/01/02 20:23:52.666824 peer_msg_handler.go:138: [info] [[region 1] 3] 写入kvDB
2021/01/02 20:23:52.666901 peer_msg_handler.go:101: [info] [[region 1] 3] proposals为空
2021/01/02 20:23:52.669423 peer_msg_handler.go:249: [info] [[region 1] 5] 正在process Entry: term:36 index:939 data:"\n\016\010\001\022\004\010\001\02
0\001\"\004\010\001\020\001\022\031\010\004*\025\n\007default\022\n4 00000016"
2021/01/02 20:23:52.669553 peer_msg_handler.go:126: [info] [[region 1] 5] 解析出来的命令: cmd_type:Delete delete:<cf:"default" key:"4 00000016" >
2021/01/02 20:23:52.669583 peer_msg_handler.go:135: [info] [[region 1] 5] 正在执行DeleteCF: key: [52 32 48 48 48 48 48 48 49 54]
2021/01/02 20:23:52.669650 peer_msg_handler.go:138: [info] [[region 1] 5] 写入kvDB
2021/01/02 20:23:52.669675 peer_msg_handler.go:101: [info] [[region 1] 5] proposals为空
2021/01/02 20:23:53.663805 node.go:198: [info] stop raft store thread, storeID: 1
2021/01/02 20:23:53.664097 node.go:198: [info] stop raft store thread, storeID: 2
2021/01/02 20:23:53.664179 node.go:198: [info] stop raft store thread, storeID: 3
2021/01/02 20:23:53.664295 node.go:198: [info] stop raft store thread, storeID: 4
2021/01/02 20:23:53.664407 node.go:198: [info] stop raft store thread, storeID: 5
--- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2C (36.49s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   259.753s
➜  tinykv git:(course) ✗
```

接下来我想结合2B和2C这两个project来谈一谈tinykv的架构, 因为这两部分虽然要填的函数只有几个, 但是知道怎么填并且填对, 并且知道在哪里打log, 怎么看日志都是需要对整个系统有一个非常untrival的理解的.

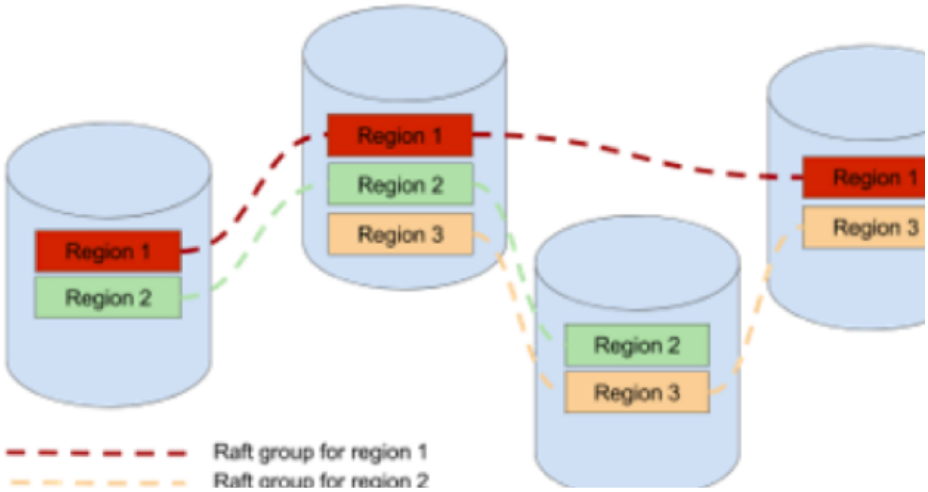
在2B当中, 我们使用在2A中实现的raft模块构建了一个具备高容错的键值存储服务. 回想一下, 我们在project1里面就实现了一个standaloneStorage, 不过那个是单机的, 也就是说那台机器挂了, 整个系统就提供不了服务了, 2B里面的高容错指的就是能够处理宕机, 网络隔离等异常条件, 并且能够正确的提供服务.那么是如何做到的呢? 就是利用Raft算法: Raft算法能够保证一个raft集群里面只要不超过半数的机器挂掉或者网络隔离, 系统就能够正常工作, 日志等信息也都是正常的. 但是TinyKV 目标是支持 100 TB+ 以上的数据，一个 Raft 集群是铁定没法支持这么多数据的，所以我们需要使用多个 Raft 集群，也就是 Multi Raft.

在介绍multi Raft之前, 我们需要知道几个术语: Store , Peer and Region

> - Store stands for an instance of tinykv-server
> - Peer stands for a Raft node which is running on a Store
> - Region is a collection of Peers, also called Raft group

那么这三个术语是什么意思呢? 我将结合下面这张图来解释一下:

- 指导书上面说store代表的就是一个tinykv的服务的实例, 我觉得你可以把store理解成电脑的硬盘, 一个store对应的就是下图中的一个圆柱体, 每个store都会有一个sotreid, 一个store里面可以存放多个peer, 每个peer属于一个region, 且这些peer都属于不同的region, regionid+storeid唯一确定一个peer
- 那么什么是peer呢? 我觉得一个peer就是一个Rawnode, 而一个Rawnode其实就是一个Raft节点的封装, 我们可以认为一个peer就是一个Raft集群里面的一个节点
- 那么什么是region呢? 一个region就是一个raftGroup, 比如下图中的三个红色的连起来就是一个region, 三个绿色的, 三个橙色的同理, 换句话来说, 一个region就是一个raft集群, 每个raft集群负责的事情就是存储对数据库操作的日志, 只不过不同region负责的键的range不一样, region就是按照键的range来划分的, 这样设计有利于水平扩展系统.



不过在project2里面远远没有这么复杂:

> For simplicity, there would be only one Peer on a Store and one Region in a cluster for project2. So you don’ t need to consider the range of Region now. Multiple regions will be further introduced in project3.

也就是说, 我们只有一个raft集群, 所有的键都存在这一个集群里面, 就像下图一样:

我们按照指导书的顺序接着分析, 下面进入The code部分:

StandaloneStorage是直接对底层的存储引擎进行读写, 因为只有一台机器. 但是RaftStorage就不一样了, 他会首先将读写请求发送给Raft, 然后等待raft提交了这个日志之后才会进行实际的读写操作, 因为raft提交了该日志意味着这个日志已经复制到大多数节点上面了, 我们可以安全的应用它到状态机里面了. 通过这种方式, 我们就可以保证multiple Stores之间的一致性了.

我们知道, 在project1里面, 我们操纵底层的数据库是通过storage接口里面的Reader与Write方法的, 同样, 我们在raftstore里也是通过这两个方法, 不同的是, 我们调用这两个方法之后, 我们作为上层应用发出的命令会被封装成一个RaftCmdRequest, 然后通过信道发送给raftWorker, raftWorker通过信道接收这个请求, 然后newPeerMsgHandler来处理这一个请求, 但是这个请求的回应会通过callback异步的返回, 当raft集群commit了然后apply之后, 就会在proposals里面找到这个cb, 然后发出回应, 否则的话, 如果过期了, 也会做出相应的过期的回应等等. 此时Reader与Writer方法里面的ctx参数也有了作用了, 它里面携带的就是region的一些信息.

至此, 整个消息处理逻辑可以划分成两个部分: raft worker从信道里面拿到消息, 这些消息包括一些驱动raft模块运转的基本时钟消息, 已经上层应用发送的需要被作为raft日志提交的raft command, 这些消息都会调用rawnode的step函数在raft集群里面做出进一步的处理, 然后我们会调用raft的ready函数获取ready结构体进行进一步的处理, 这里有必要详细的介绍一下Ready结构体:

方法/字段	作用
Entries（方法）	取出上一步发到 Raft 中，但尚未持久化的 Raft Log。
CommittedEntries	取出已经持久化，并经过集群确认的 Raft Log。
Messages	取出 Raft 产生的消息，以便真正发给其他节点。
Snapshot	取出上一步产生的快照
HardState	获得HardState,在 HardState 里面，保存着该 Raft 节点最后一次保存的 term 信息，之前 vote 的哪一个节点，以及已经 commit 的 log index。

应用程序在 propose 一个消息之后，应该调用 `RawNode::ready` 并在返回的 Ready 上继续进行处理：包括持久化 Raft Log，将 Raft 消息发送到网络上等. 而在 Follower 上，也不断运行着示例代码中与 Leader 相同的循环：接收 Raft 消息，从 Ready 中收集回复并发回给 Leader……对于 propose 过程而言，当 Leader 收到了足够的确认这一 Raft Log 的回复，便能够认为这一 Raft Log 已经被确认了. 在下一次这个 Raft 节点调用 `RawNode::ready` 时，便可以取出这部分被确认的消息，并应用到状态机中了。在将一个 Ready 结构体中的内容处理完成之后，应用程序即可调用这个方法更新 Raft 中的一些进度，包括 last index、commit index 和 apply index 等。

Implement peer storage

我们需要了解三个比较重要的状态:

> - RaftLocalState: Used to store HardState of the current Raft and the last Log Index.
> - RaftApplyState: Used to store the last Log index that Raft applies and some truncated Log information.
> - RegionLocalState: Used to store Region information and the corresponding Peer state on this Store. Normal indicates that this Peer is normal, Applying means this Peer hasn’t finished the apply snapshot operation and Tombstone shows that this Peer has been removed from Region and cannot join in Raft Group.

以及两个badger实例:

> - raftdb stores raft log and `RaftLocalState`
> - kvdb stores key-value data in different column families, `RegionLocalState` and `RaftApplyState`. You can regard kvdb as the state machine mentioned in Raft paper

这一部分我们要实现的函数就是SaveReadyState, 在上面我们已经详细介绍过了Ready结构体, 那么这里我就来说说我们在这个函数里面具体需要做什么?

> what this function does is to save the data in `raft.Ready` to badger, including append log entries and save the Raft hard state.

也就是取出上一步发到 Raft 中，但尚未持久化的 Raft Log, 将其持久化到storage里面, 以及看一看hardstate有没有发生改变, 有的话也把hardState持久化一下.

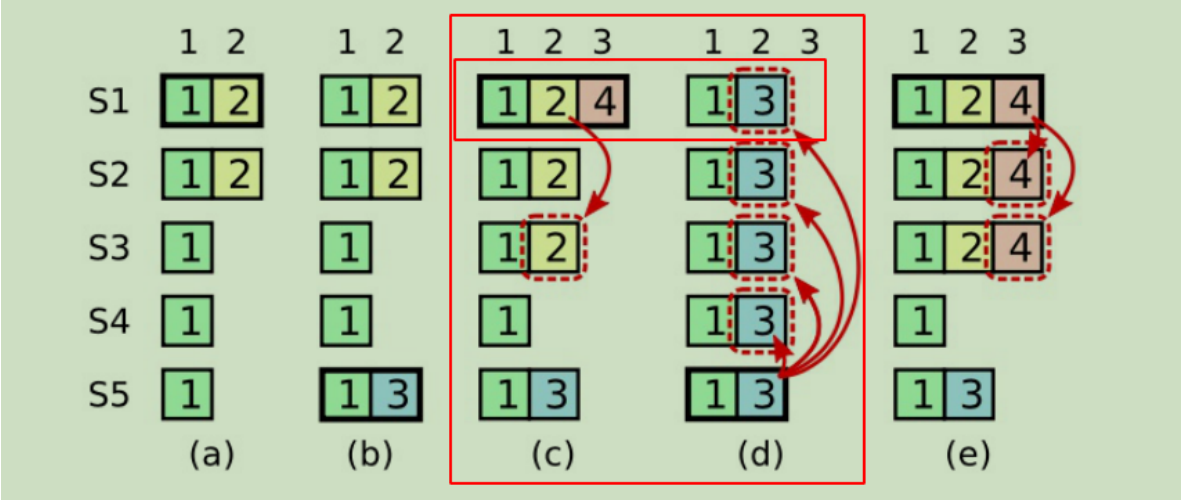
> Hints:
>
> - Use `WriteBatch` to save these states at once.
> - See other functions at `peer_storage.go` for how to read and write these states.

`WriteBatch` 在我的理解里面就是一个缓冲区的角色, 我们把一些需要在badger里面修改的东西先全部放到这个 `WriteBatch` 里面, 我看了源码之后发现这里的设计有一个很巧妙的地方, 比如你Put一个键进去, 它就是直接在ents后面append一个键值对, 但是如果你delete一个键, 那么它也在后面append一个键, 只不过值为空, 然后最后原子性的把所有东西写进db的时候, 就是从头到尾遍历 `WriteBatch` 里面的修改序列, 如果值为空, 就deletecf, 很巧妙.

至于如何使用setmeta这种, 需要注意的就是在keys.go里面有许多辅助函数, 可以帮助你把键构造好

另外需要注意的一点就是, 最好等到最后才writeToDB, 或者在writeToDb之后及时的reset也可以, 我在这里吃过亏.

指导书里面也给出了非常具体的做法, 这一部分难度不算太大, 对了, 关于delete any previously appended log entries which will never be committed这个东西还是有必要说明一下的, 我一开始看到这句话也没有明白, 其实说的是这种情况, 现在知道为什么要删以及该怎么删了吧.



> To append log entries, simply save all log entries at `raft.Ready.Entries` to raftdb and delete any previously appended log entries which will never be committed. Also update the peer storage's `RaftLocalState` and save it to raftdb.
>
> To save the hard state is also very easy, just update peer storage's `RaftLocalState.HardState` and save it to raftdb.

Implement Raft ready process

这一部分主要就是实现提交Raft命令以及处理Ready结构体的逻辑

需要填的函数只有两个: proposeRaftCommand以及HandleRaftReady

但是在填这两个函数之前, 我们需要知道一些先导知识:

我们知道peer就是对rawnode的包装, 其中一个重要的函数就是peerMsgHandler, 这个函数里面又分为两个重要的部分: HandleMsgs与HandleRaftReady, 我们首先HandleMsgs, 然后raft集群的状态就会发生变化, 然后上层可以通过Ready结构体获取这些变化在HandleRaftReady里面做出一些行为, 比如:

> persisting log entries, applying committed entries and sending raft messages to other peers through the network.

伪代码描述如下所示:

```
for {
  select {
  case <-s.Ticker:
    Node.Tick()
  default:
    if Node.HasReady() {
      rd := Node.Ready()
      saveToStorage(rd.State, rd.Entries, rd.Snapshot)
      send(rd.Messages)
      for _, entry := range rd.CommittedEntries {
        process(entry)
      }
```

```
        s.Node.Advance(rd)
    }
}
```

在将一个 Ready 中的所有更新处理完毕之后，使用 RawNode::advance 更新一些raft寄存的inner state

至此, 我们可以总结出服务端的运行逻辑:

服务端运行流程大致如下：

1 客户端调用RawGet / RawPut / RawDelete / RawScan RPC

2 RPC处理程序调用RaftStorage相关方法

3 RaftStorage 向Raftstore发送Raft命令请求，并等待响应

4 RaftStore 将Raft命令作为Raft日志进行处理

5 Raft模块附加日志，并通过 PeerStorage持久化到本地数据库

6 Raft模块提交日志

7 Raft worker在处理Raft模块返回的ready结构体时执行Raft命令，并通过调callback返回相应的响应

8 RaftStorage 接收返回的响应并返回到RPC处理程序

9 RPC处理程序将RPC响应返回给客户端。

Note: 注意错误处理

> - ErrNotLeader: the raft command is proposed on a follower. so use it to let client try other peers.
> - ErrStaleCommand: It may due to leader changes that some logs are not committed and overrided with new leaders’ logs. But client doesn’t know that and is still waiting for the response. So you should return this to let client knows and retries the command again.

emmm, 感觉我上面还是没有讲清楚, 有一些细碎的东西也不知道怎么放上去讲, 那就这样吧qwq

接下来谈一谈2C的一些点

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧的信息，那么会带来可用性问题。因此需定期对日志进行压缩, 2C需要我们做的工作就是实现日志压缩的逻辑, 策略就是使用快照.

日志压缩流程大致如下：

1raftlog-gc worker定时检测是否需要进行raftlog-gc

2若需要进行raftlog-gc则发送admin command CompactLogRequest

3proposeRaftCommand中调用propose进行处理

4HandleRaftReady中在apply命令时若发现该命令是CompactLogRequest，则修改TruncatedState并发送一个raftLogGcTask到raftLogGCTaskSender，在另一个线程中对raftdb中的日志进行压缩

由于日志压缩可能导致某些Follower没有完全接收到Leader所有的日志条目而Leader的一些日志条目已经被压缩了，因此此时Leader需要向Follower发送快照来使Follower日志同步，需要增加处理快照的相关逻辑，处理快照的大致流程如下：

1Raft中若需要发送Snapshot则调用Storage.Snapshot()生成日志，然后作为raft消息发送

2接收方收到Snapshot调用handleSnapshot进行处理，修改相关字段，将Snapshot保存到pendingSnapshot中

3在上层调用ready时将pendingSnapshot包装至ready中返回给上层

4SaveReadyState中若检测到有Snapshot则调用ApplySnapshot更新相关信息进行处理

project3

3A部分还算是比较简单,面向测试编程, 很快就能完成

3A的TestCommitAfterRemoveNode3A测试点

EntryType_EntryConfChange提交了之后, 集群的节点数目就发生了改变, 于是我们需要重新判断一些日志是否满足可以提交的条件

```go
// removeNode remove a node from raft group
func (r *Raft) removeNode(id uint64) {
    // Your Code Here (3A).
    for index, peer := range r.peers {
        if peer == id {
            r.peers = append(r.peers[:index], r.peers[index+1:]...)
            delete(r.Prs, peer)
            break
        }
    }
    // fix bug: TestCommitAfterRemoveNode3A这个测试挂掉了, 这个测试检测的就是集群成员变化提交之
    // 有一些日志也许就可以提交了, 所以我们得重新判断一下
    if r.State == StateLeader {
        r.updateCommit()
    }
    r.PendingConfIndex = None
}
```

```
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 3
--- PASS: TestLeaderTransferToSlowFollower3A (0.00s)
=== RUN   TestLeaderTransferAfterSnapshot3A
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 3
--- PASS: TestLeaderTransferAfterSnapshot3A (0.00s)
=== RUN   TestLeaderTransferToSelf3A
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 1
--- PASS: TestLeaderTransferToSelf3A (0.00s)
=== RUN   TestLeaderTransferToNonExistingNode3A
2021/01/03 22:29:02 raft.go:754: [info] LeaderTransferToNonExistingNode
--- PASS: TestLeaderTransferToNonExistingNode3A (0.00s)
=== RUN   TestLeaderTransferReceiveHigherTermVote3A
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 3
--- PASS: TestLeaderTransferReceiveHigherTermVote3A (0.00s)
=== RUN   TestLeaderTransferRemoveNode3A
2021/01/03 22:29:02 raft.go:754: [info] LeaderTransferToNonExistingNode
--- PASS: TestLeaderTransferRemoveNode3A (0.00s)
=== RUN   TestLeaderTransferBack3A
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 3
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 1
--- PASS: TestLeaderTransferBack3A (0.00s)
=== RUN   TestLeaderTransferSecondTransferToAnotherNode3A
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 3
2021/01/03 22:29:02 raft.go:769: [info] [1] send MessageType_MsgTimeoutNow to 2
--- PASS: TestLeaderTransferSecondTransferToAnotherNode3A (0.00s)
=== RUN   TestTransferNonMember3A
2021/01/03 22:29:02 raft.go:814: [info] [1] has been removed
2021/01/03 22:29:02 raft.go:814: [info] [1] has been removed
2021/01/03 22:29:02 raft.go:814: [info] [1] has been removed
--- PASS: TestTransferNonMember3A (0.00s)
=== RUN   TestRawNodeProposeAndConfChange3A
--- PASS: TestRawNodeProposeAndConfChange3A (0.00s)
=== RUN   TestRawNodeProposeAddDuplicateNode3A
--- PASS: TestRawNodeProposeAddDuplicateNode3A (0.00s)
PASS
ok      github.com/pingcap-incubator/tinykv/raft        0.002s
→ tinykv git:(course) ✗
```

3B部分遇到了非常多的问题

- Propose transfer leader
    - 这里发现我其实在raft模块实现的就有问题, 由于我之前对于leader自己的Prs的更新是单独更新的, 所以当leader被remove之后, 单独更新就会出现解引用nil指针的panic, 这个倒蛮好解决, 在更新Prs的循环里面特判自己就好了

- 另外在step函数里面, 我一开始的实现是在最开始特判自己的Prs是否为nil, 是的话直接返回, 这样是不对的, 因为这样就接收不到confchange等日志信息了
- 另外学长的笔记里面提到了一个指导书里面没有提到的一个优化, 我觉得是很有必要的, 也加上去了, 就是当存在未被应用的confchange时,拒绝MessageType_MsgHup操作。不然这个时候易主会很麻烦.
- 然后我还加了一个日志不匹配快速回退的优化, 因为我遇到了一个重传snap5次panic的bug, 我以为是日志压缩没处理好, 但事实证明和这个无关, 然后我又加了一个限制snapshot发送间隔的逻辑, 但是只是延长了这个panic触发的时间, 并没有根本上解决这个问题.
- 然后还修正了addnode里面新加入节点的next的值的bug, 一个新加入的节点的初始logindex是0, 那么下一条要发给他的就是1才对.

- Implement conf change in raftstore

  这一部分debug到头秃了

  - 我们需要避免多个confchange的情况, 如果PendingConfIndex的值大于p.peerStorage.AppliedIndex(),那么说明存在还未被应用的confchang指令。直接返回
  - 运行confchange的基本测试的时候报错, 报错的地方在newpeerStorage函数里面

    ```
    raftState, err := meta.InitRaftLocalState(engines.Raft, region)
    if err != nil : nil, err
    applyState, err := meta.InitApplyState(engines.Kv, region)
    if err != nil : nil, err
    if raftState.LastIndex < applyState.AppliedIndex {
        panic(fmt.Sprintf( format: "%s unexpected raft log index: lastIndex %d < appliedIndex %d",
            tag, raftState.LastIndex, applyState.AppliedIndex))
    }
    ```

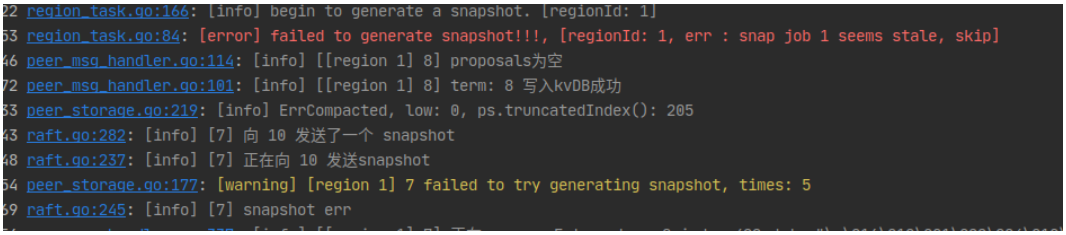
    错误在raftLocalState的LastIndex确实是0, 但是applyState的appliedIndex却不是0, 是一个在destroy之前的正确的applyindex的值

    当分析到这里的时候, 我的解决方向是向着完全错误的一方, 我认为storage里面的内容destroy之后应该还在, 那么读取出来的应该是destroy之前的正确的值才对, 不应该会是出现ErrKeyNotfound然后按照一定逻辑初始化为0, 于是在这个方向上调试了一天无果, 后面看了destroy的源码才发现应该是applyState错了, 而且错在在destroy之后又往底层的db写入了数据, 于是顺着这个思路终于找到了错误所在. HandleRaftReady里面process一个entry之后需要判断d.stop是否为真, 为真的话应该直接返回, 不然执行了后面的逻辑就会把applyState写入db
  - ```
    3 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    9 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    3 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    9 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    6 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    6 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    1 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    4 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    3 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    6 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    5 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    1 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    7 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    8 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    2 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    6 store_worker.go:159: [info] msg region epoch: conf_ver:6 version:1  handle raft message. from_peer:1, to_peer:2, store:2, regi
    4 store_worker.go:111: [info] localState: region:<id:1 region_epoch:<conf_ver:2 version:1 > peers:<id:1 store_id:1 > peers:<id:3
    9 store_worker.go:82: [error] handle raft message failed storeID 2, region 1 not exists but not tombstone: region:<id:1 region_e
    ```

    这个bug相当的诡异, 原因就是停机之后没有立刻return
  - ```
    2021/01/06 15:25:45.492936 peer_storage.go:403: [info] regionSched finished
    2021/01/06 15:25:45.493012 peer.go:231: [info] [region 1] 2 destroy itself, takes 7.263587ms
    panic: [region 1] 2 meta corruption detected

    goroutine 275 [running]:
    github.com/pingcap-incubator/tinykv/kv/raftstore.(*peerMsgHandler).destroyPeer(0xc17c869e08)
        /home/wang/文档/Lab/tinykv/kv/raftstore/peer_msg_handler.go:805 +0x417
    github.com/pingcap-incubator/tinykv/kv/raftstore.(*peerMsgHandler).processAdminRequest(0xc17c869e08, 0xc17c7d9c20, 0xc17c869b08, 0xc17c7d9c70)
    ```

    这个bug产生的原因: 应用了快照之后没有根据快照中的信息更新storeMeta的值
  -

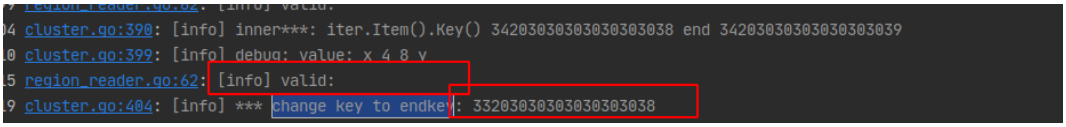


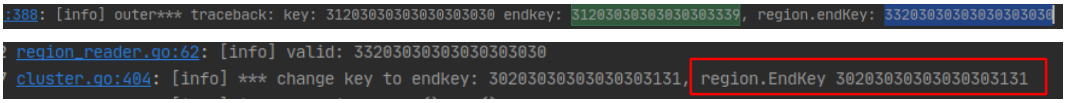
这个就是在前面介绍过的被我自己写的panic拦截下来的error, 因为我对于snapshot只处理了还未生成好, 暂时不可用的错误, 所以其他错误我就直接panic了, 这个错误后来找到的原因就是我在removeNode里面使用了

```
meta.WriteRegionState(kvWB, region, rspb.PeerState_Tombstone)
```

应该使用PeerState_Normal, 我之所以使用tombstone, 原因就是前面遇到的一个bug, 说no region and not tombstone啥的, 我在那个时候就这样设置了qwq

- split
  - 一个神奇的bug,scan的时候如果跨region了会出现重叠的现象, 后来终于找到原因了

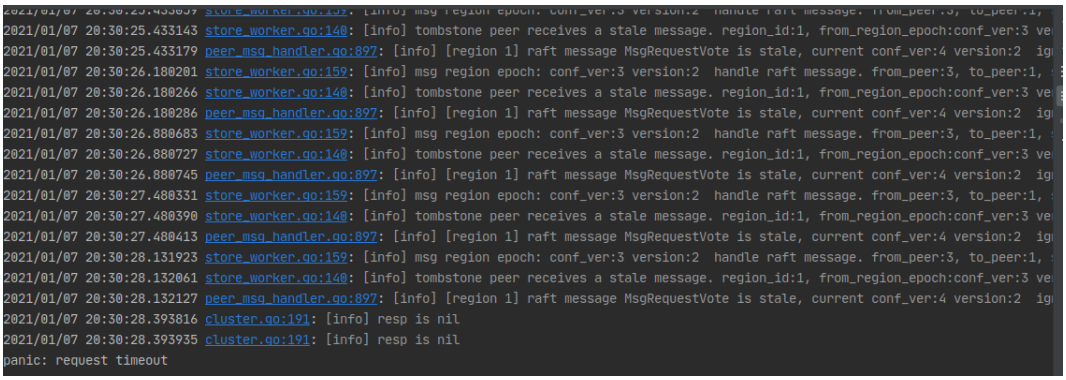




scan操作前与遍历完这一个scan更新key的值的时候region的endkey的值发生改变了

3B里面当正在scan的时候发生了split region该怎么处理啊? 因为跨region的scan它的逻辑是看当前region是否遍历完，遍历完之后更新key为endkey，然后获取下一个连续的新region，但是这个时候发生了split就会导致当前region的endkey发生变化，就会导致key更新出问题，结果错误了, 解决办法就是在response里面传region时传一个深拷贝.

  - 还遇到了许多偶尔触发的稀奇古怪的错误或者那种导致的原因很多难以排查的错误, 比如find no region; request timeout; meta corrputed, stop集群卡死等等, 太难调试了, 放弃了.



project3A

在Raft模块增加领导者更改和成员更改

在Raft模块增加MsgTransferLeader，EntryConfChange，MsgHup相关处理逻辑

Implement leader transfer
在step函数中:

1. 增加leader对pb.MessageType_MsgTransferLeader的处理逻辑。1. 首先判断当前leadTransferee是否为空,如果不为空,则比较消息的发送方是否相同,如
   果相同则忽略这条消息;如果不同则停止transfer leader操作。
2. 如果transfer leader的对象是自己,则忽略这条消息
3. 然后重置electionElapsed,并设置leadTransferee。

4. 根据leadTransferee的同步状态,选择发送MessageType_MsgTimeoutNow消息还是发送同步日志
5. 增加leader对于propose的处理逻辑:如果leadTransferee不为空,则拒绝接受propose消息。
6. 增加follower对MessageType_MsgTransferLeader和MessageType_MsgTimeoutNow的处理
7. 如果接受到MessageType_MsgTransferLeader消息,则向leader发送MessageType_MsgTransferLeader消息。
8. 如果受到MessageType_MsgTimeoutNow消息,则开始进行选举。

Implement conf change
在step函数中:

1. leader的propose处理中,增加对 EntryType_EntryConfChange的处理。如果消息的entries中存在EntryType_EntryConfChange类型,并且当前raft的pendingconf 还没有被应用,那么忽略这个entry,否则更新r.PendingConfIndex。
2. 增加follower和candidate对MessageType_MsgHup的处理,当存在未被应用的confchange时,拒绝MessageType_MsgHup操作。

project3B

上面谈到了在实现过程中我记得的一些bug(其实遇到了蛮多bug的,不过有一些没有及时记录下来,忘记了qwq)下面我来谈一谈3B部分要我们干什么以及一些具体的实现思路

需要我们完成的任务: 在RaftStore中增加领导者更改、成员更改、区域分割这几个功能

- Propose transfer leader

  这一步实现起来比较简单, 因为TransferLeader实际上是一个action, 所以不需要备份到其他的peers上面, 也就是说我们在proposeRaftCommand函数里面其实不用调用raftGroup的Propose函数, 而是直接调用TransferLeader函数并且直接返回一个response给client即可

- Implement conf change in raftstore

  在实现这一部分时, 一个重要的术语RegionEpoch: 它由两部分组成, conf_ver和version, conf_ver会随着addNode, removeNode这种操作递增, 而version会随着region的split和merge操作递增.

  实现这一部分需要特别注意的就是, 我们在propose的时候调用的不是raftGroup的propose函数, 而是ProposeConfChange函数, 另外当这个日志已经被复制到大多数的peer上面, 我们可以安全的提交的时候, 我们在process的时候还需要更新RegionLocalState, 主要更新的是两个成员RegionEpoch和peers, 在最后还要记得调用ApplyConfChange函数在raft模块中更新相应的状态. 下图是16级学长给出的完成建议与步骤

    1. 如果PendingConfIndex的值大于 p.peerStorage.AppliedIndex()，那么说明存在还未被应用的confchang指令。直接返回
    2. 然后对RaftCmdRequest进行检查。
       1. 如果当前只有一个节点(自身)，那么不需要进行检测，直接返回。
       2. 如果是添加节点类型，则向progress中增加对应id的空白progress。如果是删除操作则删除对用的progress。
       3. 然后检查，如果有超过一半的节点需要发送snapshot，那么则返回报错
    3. 对RaftCmdRequest进行压缩，通过ProposeConfChange提交这条指令
    4. 返回nextProposalIndex
    2. 在handleraftready中增加处理逻辑
       1. 根据confchange类型修改region的peer
       2. 更新RegionState
       3. 调用ApplyConfChange
       4. 设置PeersStartPendingTime和PeerCache
       5. 调用HeartbeatScheduler
       6. 判断是否需要调用destroyPeer

- Implement split region in raftstore

  Region中维护了该region的start key和end key，当Region中的数据过多时应对该Region进行分割，划分为两个Region从而提高并发度. 执行区域分割的大致流程如下：

  1split checker检测是否需要进行Region split，若需要则生成split key并发送相应命令

  2proposeRaftCommand执行propose操作

  3提交后在HandleRaftReady对该命令进行处理：

  检测split key是否在start key 和end key之间，不满足返回

  创建新region 原region和新region key范围为start~split，split~end

  更新d.ctx.storeMeta中的regionranges和regions

  为新region创建peer，调insertPeerCache插入peer信息

  发送MsgTypeStart消息

  下图是16级学长给出的完成建议与步骤(其中有几步指导书里面没有讲,我也不是很清楚为什么要这样QwQ)

project3C

3C就是要我们实现一个调度器

kv数据库中所有数据被分为几个Region，每个Region包含多个副本，调度程序负责调度每个副本的位置以及Store中Region的个数以避免Store中有过多Region。

为实现调度功能，region应向调度程序定期发送心跳，使调度程序能更新相关的信息从而能实现正确的调度，processRegionHeartbea函数实现处理region发送的心跳，该函数通过regioninfo更新raftcluster中的相关信息。

为避免一个store中有过多Region而影响性能，要对Store中的Region进行调度，通过Schedule函数实现

其实我们只需要填写两个函数就可以了: processRegionHeartbeat以及Schedule函数

- processRegionHeartbeat

  在这个函数的实现当中, 我们首先需要判断一个心跳包是否是可信的(credible), 判断的方法指导书里面说的已经很详细了, 我就直接搬运过来了

  > 1. Check whether there is a region with the same Id in local storage. If there is and at least one of the heartbeats' `conf_ver` and `version` is less than its, this heartbeat

> region is stale
> 2. If there isn’t, scan all regions that overlap with it. The heartbeats’ `conf_ver` and `version` should be greater or equal than all of them, or the region is stale.

当获悉一个心跳包是可信的之后, 我们就需要判断是否有必要根据这个心跳包所携带的一些信息来更新我们的调度系统的信息, 在这里指导书里面给出了四条必须更新的条件, 不过只写这四条测试会挂掉, 我们还需要加一些其他的不可跳过更新的判别条件才可以通过测试, 不过其实这里不判断, 直接所有可信心跳包都更新状态也是没问题的, 因为冗余更新不会影响系统的正确性, 只会影响系统的处理速度

> - If the new one’s `version` or `conf_ver` is greater than the original one, it cannot be skipped
> - If the leader changed, it cannot be skipped
> - If the new one or original one has pending peer, it cannot be skipped
> - If the ApproximateSize changed, it cannot be skipped
> - …

- Schedule

  实现调度, 将一个region从一个负载高的store移动到一个负载低的store, 具体的操作步骤指导书以及测试用例说的非常非常清楚了, 我这里来简单的总结一些:

  1. 选择出所有合适的stores

     那么什么是合适的store呢? 指导书给了我们答案

     > In short, a suitable store should be up and the down time cannot be longer than `MaxStoreDownTime` of the cluster, which you can get through `cluster.GetMaxStoreDownTime()`

  2. 将它们按照region size排序.

     这里我为StoreInfo结构体实现了Len, Swap以及Less方法, 这样我们就可以使用sort包来进行排序了.

  3. 然后尝试找到具有最大region size的region

     具体尝试次序指导书说的非常明白了

     > First, it will try to select a pending region because pending may mean the disk is overloaded. If there isn’t a pending region, it will try to find a follower region. If it still cannot pick out one region, it will try to pick leader regions. Finally, it will select out the region to move, or the Scheduler will try the next store which has a smaller region size until all stores will have been tried.

  4. 找到了一个用来move的region, 接下来我们需要寻找一个target
  5. 我们想要找到一个有着最小region size的store作为target, 同时需要保证这个target store里面没有我们想要move进去的region的id, 因为一个region最多只能有一个peer在一个store上面
  6. 等到我们找到了一个target之后, 我们需要判断一下我们的这个移动是否划算? 判断方法指导书里面说的已经很详细了

     > If the difference between the original and target stores’ region sizes is too small, after we move the region from the original store to the target store, the Scheduler may want to move back again next time. So we have to make sure that the difference has to be bigger than two times the approximate size of the region, which ensures that after moving, the target store’s region size is still smaller than the original store.

  7. 最后, 在target store上面分配一个new peer, 然后创建一个新的 move peer operator

project4

这一个project主要还是面向测试编程, 一些cornor case还是得看测试才知道, 比如在哪些情况下应该返回怎样的error等等.

这一部分的文档十分的详细, 提示里面的内容也非常有用, 可以避免踩很多坑, 当实现遇到困难的时候可以选择把文档和hints以及测试再读两遍.

一些比较重要的:

关于tinykv里面的列族

> TinyKV uses three column families (CFs): `default` to hold user values, `lock` to store locks, and `write` to record changes. The `lock` CF is accessed using the user key; it stores a serialized `Lock` data structure (defined in lock.go). The `default` CF is accessed using the user key and the start timestamp of the transaction in which it was written; it stores the user value only. The `write` CF is accessed using the user key and the commit timestamp of the transaction in which it was written; it stores a `Write` data structure (defined in write.go).

project4A

实现结构体MvccTxn的相关函数

对于Scan函数的理解一开始存在一定的问题, 后来问大佬之后才搞懂了

定位到第一个大于等于这个 Key-Version 的位置, 如果不存在这个key-version, 那么会定位到比它大的最小的那个
经过询问大佬, 终于搞懂了这个seek函数了, key的格式是这样的 keyN1-versionN2 -> value

Key1-Version3 -> Value
Key1-Version2 -> Value
Key1-Version1 -> Value
……
Key2-Version4 -> Value
Key2-Version3 -> Value
Key2-Version2 -> Value
Key2-Version1 -> Value
……
KeyN-Version2 -> Value
KeyN-Version1 -> Value
……

按照ukey升ts降，seek是按照整体排序seek的
具体来说就是如果key不存在, 那么就seek大的最小的那个, 如果key存在, 但是version不存在, 那么就seek到比version小的最大的那个

后来在指导书里面也找到了对应的出处

> A user key and timestamp are combined into an encoded key. Keys are encoded in such a way that the ascending order of encoded keys orders first by user key (ascending), then by timestamp (descending).

这一部分在实现的时候会遇到有几个函数可能不太明白具体需要我们做什么, 不过可以通过看测试以及函数上面的注释来搞懂, 这里不多加赘述.

另外transaction.go文件的最下面有几个很有用的辅助函数, 不要忘了.

project4B

实现server.go中的KvGet, KvPrewrite, KvCommit函数

这一部分的KvPrewrite与KvCommit就包含了分布式事务里面二阶段提交的逻辑, 这里简单的描述一下二阶段提交的主要流程(节选自官方博客):

第一步是 prewrite，即将此事务涉及写入的所有 key 上锁并写入 value。当 prewrite 全部完成时，client 便会取得 `commit_ts`，然后继续两阶段提交的第二阶段: 就是把写在 `CF_LOCK` 中的锁删掉，用 `commit_ts` 在 `CF_WRITE` 写入事务提交的记录。

实现这一部分时, 我们可以参考一下Coprocessor函数是怎样写的以及如何处理regionError的错误的

```
resp := new(coppb.Response)
  reader, err := server.storage.Reader(req.Context)
  if err != nil {
    if regionErr, ok := err.(*raft_storage.RegionError); ok {
      resp.RegionError = regionErr.RequestErr
      return resp, nil
    }
    return nil, err
  }
```

剩下的工作就是反复的看指导书以及测试, 这里就不多加赘述了.

project4C

实现server.go中的KvScan, KvCheckTxnStatus, KvBatchRollback,KvResolveLock函数

这一部分还是挺有难度的, 这里我着重讲一下KvScan的思路, 剩下的三个函数面向测试编程还是相对简单一些的. 但其实另外三个函数我也并没有太搞懂其原理, 只是能够达到通过测试的程度, 根据测试, 需要特判哪些特殊情形, 对应的情形需要返回什么样的错误罢了QwQ

说回KvScan, 我们为什么需要再实现一个KvScan而不能直接使用之前实现的RawScan呢?

因为底层的badger数据库里面存储的key-value有多个版本, 然而我们需要的只是每个key最适合我们当前version的版本, 所以我们自己封装的scanner的next函数就是要达到这个目的: 对于每一个key, 我们只返回最适合的那个版本, 并且iter指向下一个不同的userkey

按照这个思路, 应该就有方向了, 我就不多加赘述了.

总结体会

非常感谢万老师能够同意我来到实验室实习, 能够接触到tinykv这么优秀的开源项目, 非常感谢pingCAP团队开源这么好的项目以及社区的源码解析的系列博客, 让我学到了许多知识, 非常感谢学长以及同学们对我的帮助, 解决了我许多的疑问. 看了一下git log, 从提交project1到提交project4C, 这之间一共历时25天. 在这个过程中更深一步的了解了Raft算法, 了解了facebook公司开源的存储引擎rocksDB, 了解了tinykv的系统的架构, 了解了调度器的一些工作原理, 了解了分布式事务的二阶段提交, 以及MVCC等技术. 记得在做2B的时候, 有几天非常自闭, 也想过放弃, 但是非常感谢当时的自己坚持下来了! 这一次的实习收获非常多!