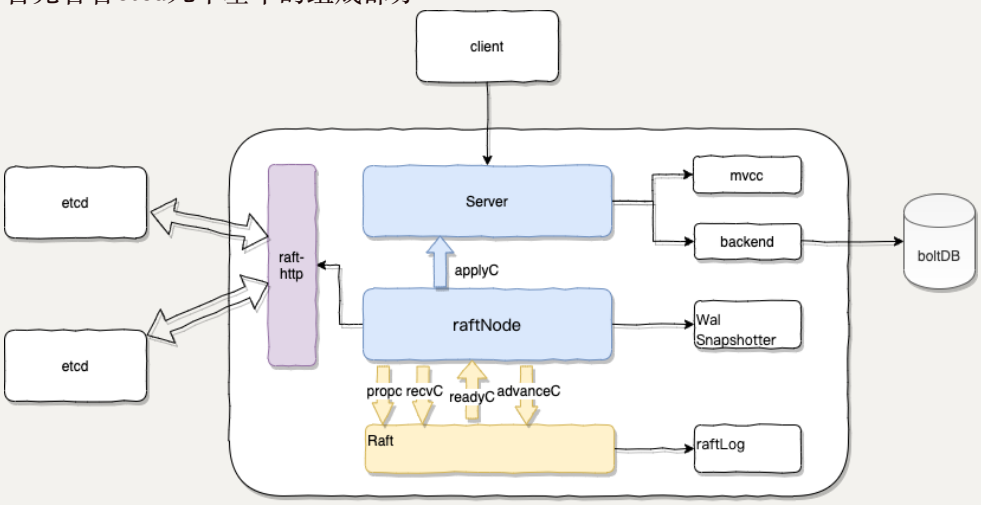
etcd 通过raft实现分布式一致性，实现参照raft的论文并做了很少的修改(优化)，本次文章整理raft的基本原理以及etcd的实现，raft具体实现可以参照论文翻译版raft-中文

1. 基本架构

之前在网上看到一些关于etcd的架构，将raft等作为etcd的底层而storage作为了上层实现,但我习惯采用领域驱动的方式分析架构。raft和server是etcd的2个subdomain，而raft-http作为支撑模块其实可以分离出来，但是目前etcd是把它放在了server中的api中，这种设计如果考虑到修改网络层的设计可能需要修改server包，单独分离出来可能更合理。

首先看看etcd几个基本的组成部分：



- httpserver
  etcd node之间进行通信，接收来自其他node的消息；
- raft
  实现分布式一致性raft协议，raft模块与server模块的通信采用了四个channel：
  - `propc`： 处理client来的命令
  - `recvc`： 处理http消息
  - `readyc`：消息经过raft处理之后封装成Ready交给server处理
  - `advanceC`： server处理一条消息之后通知raft
- WAL
  server为了防止数据丢失而实现的write ahead log，与很多数据库的实现类似
- snapshotter

  防止wal的无限制增长，定期生成snap文件仅保留 term，index以及key value data；
- mvcc

  实现多版本的并发控制，使用revision（main和sub）来描述一个key的整个过程，从创建到删除。mvcc中还包含了watcher，用于实现监听key，prefix， range的变化。
- backend & boltdb

  持久化key value到boltdb数据库
- raftlog

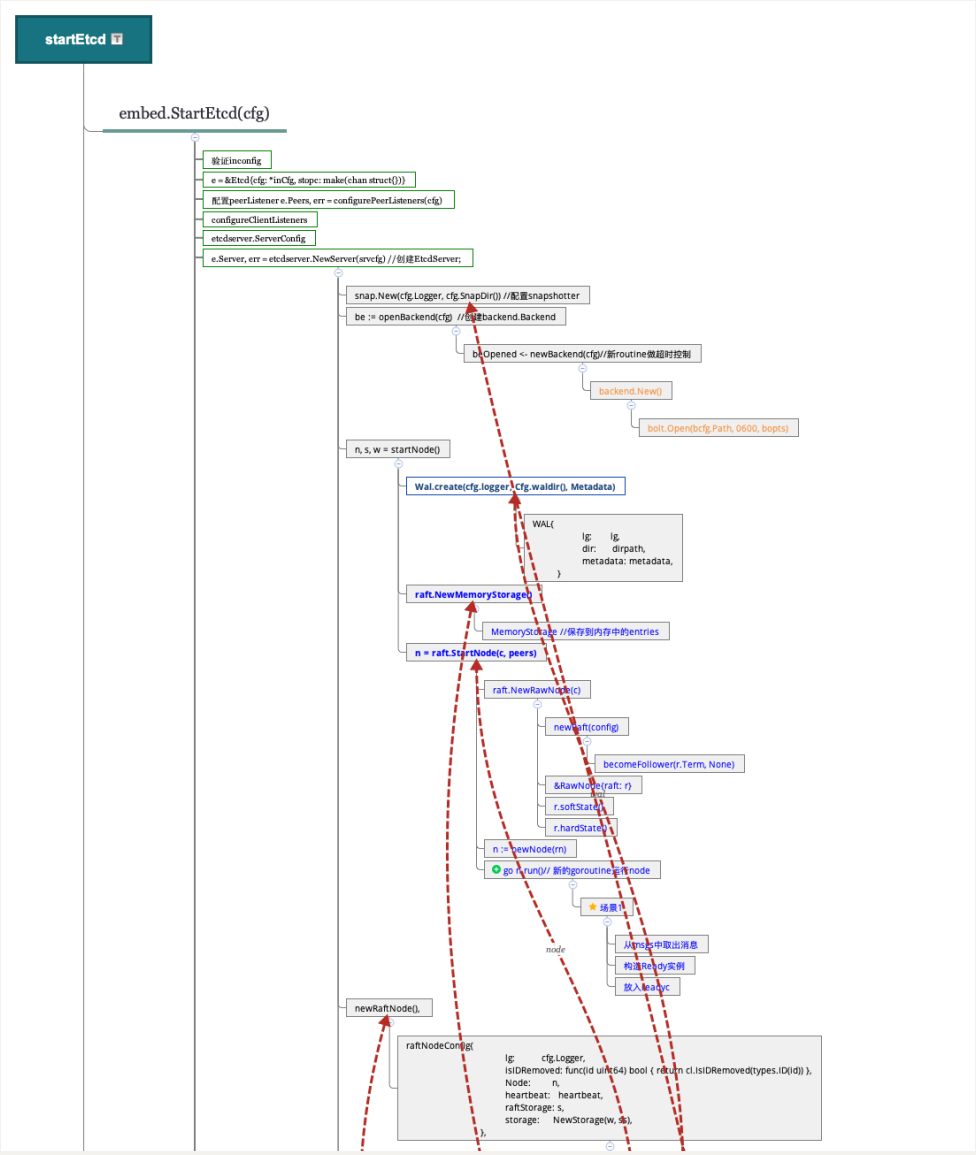
  raftlog模块包含unstable和raft的snapshot，unstable保存log entries，但是entries数量比较多的时候，就需要compact，创建一个snapshot，这里的snapshot还是保存在memory中的。raft模块会定时收集entries交给server处理。

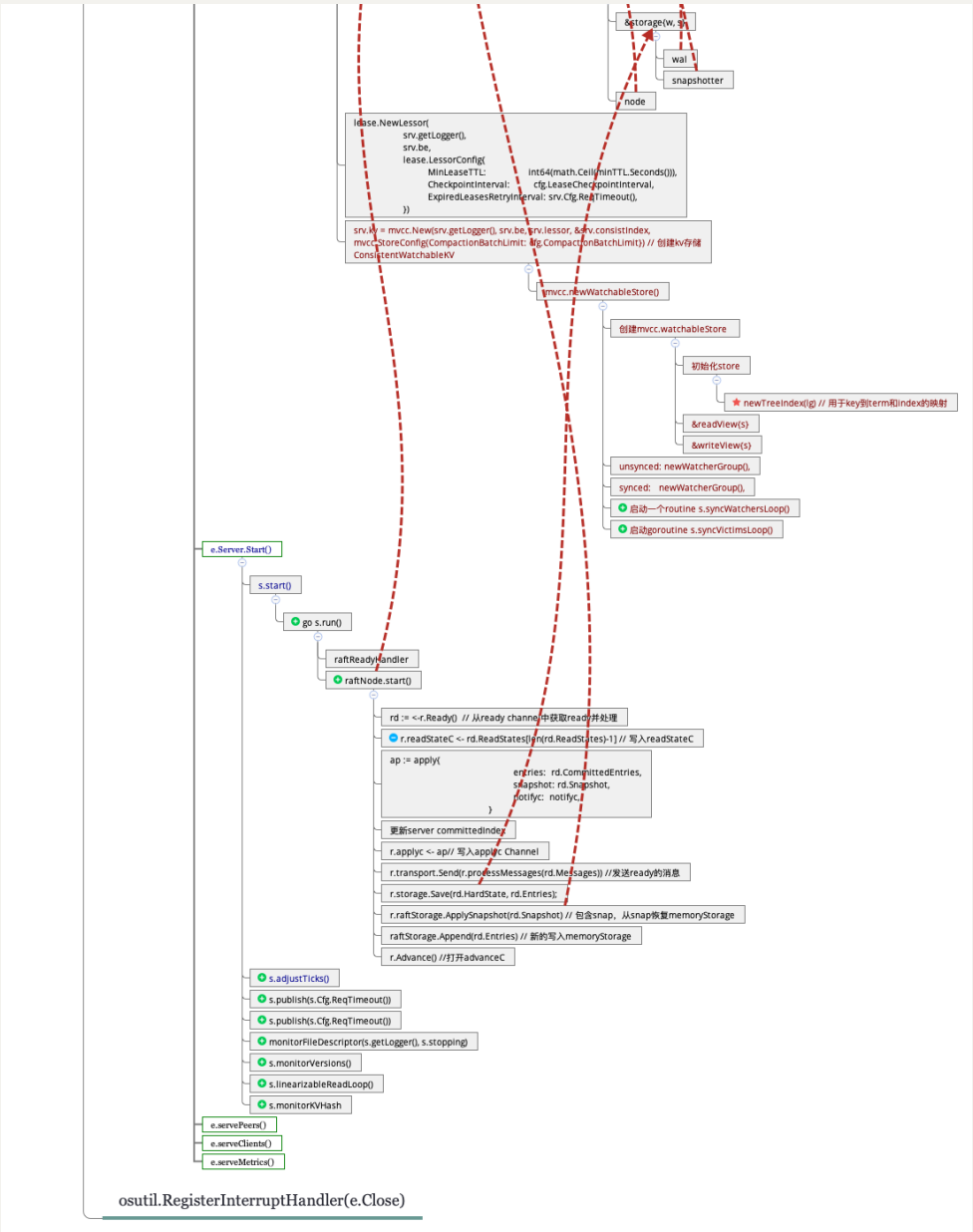
1.1. 启动

1. checkSupportArch()
2. startEtcdOrProxyV2
   2.1 解析命令行输入 cfg.parse(os.Args[1:])
   2.2 startEtcd
   2.3 osutil.HandleInterrupts(lg) 注册信号，清理系统
   2.4 notifySystemd(lg)

1.1.1. 重点分析startEtcd

1. 检查配置是否正确
2. 创建一个etcd 实例
3. configurePeerListeners 为每个peer创建一个peerListener(rafthttp.NewListener)，用于接收peer的消息
4. **configureClientListeners 创建client的listener(transport.NewKeepAliveListener)**后面分析用途
5. 创建一个etcdServer实例
6. 启动etcdServer.Start()
7. 配置peer handler





上图中的绿色加号表示一个单独的协程。

2. RAFT

---

etcd实现了raft协议，raft论文的描述参照文章开头的链接。

2.1. 角色

1. leader
    - 提交上一任期(term) leader的msg
      成为leader后发送一个空`entries`的msg到follower，让所有follower提交上一个`term`的msg；
    - 保证读的一致性
      读请求默认发送到leader节点，收到请求之后确保自己仍然

是当前系统的leader，判断的方法是根据heartbeat判断自己仍然能获取到大多数的follower的响应。
    - heartbeat time
      发送心跳，附带appendmsg
      收到resp检查是否符合大多数，提交log entry
    - election time
      这个时间还在使用，每隔这个时间就发送消息`pb.MsgCheckQuorum`检查是否能连上所有的follower
2. preCandidate

   follower切换成为candidate之后会增加系统的term，但如果该节点无法联系上系统中的大多数节点那么这次状态切换会导致term毫无意义的增大。因此在转换为candidate之前，需要发送prevote消息，消息中带有index和term，用于跟follower比较，确保entry够新；发送这个消息并且获取足够的响应才能成为candidate。
3. candidate
   收到MsgApp,heartbeat,snap消息则退回follower状态；若收到MsgPreVoteResp,则检查投票情况，超过半数成为leader
4. follower

   follower主要职责：
    - follower可以proxy的模式下工作，将收到的client请求route到leader。
    - 收到prevote消息，判断是否在leader lease内；如果自己还能收到leader的消息，拒绝投票。
    - 收到vote 消息则重置自己的election timer避免自己超时成为candidate 避免无谓的竞争。
    - 处理heartbeat更新commitIndex并重置election timer，超时则切换为precandidate
    - 处理msgApp消息，追加日志。

3. 网络http

---

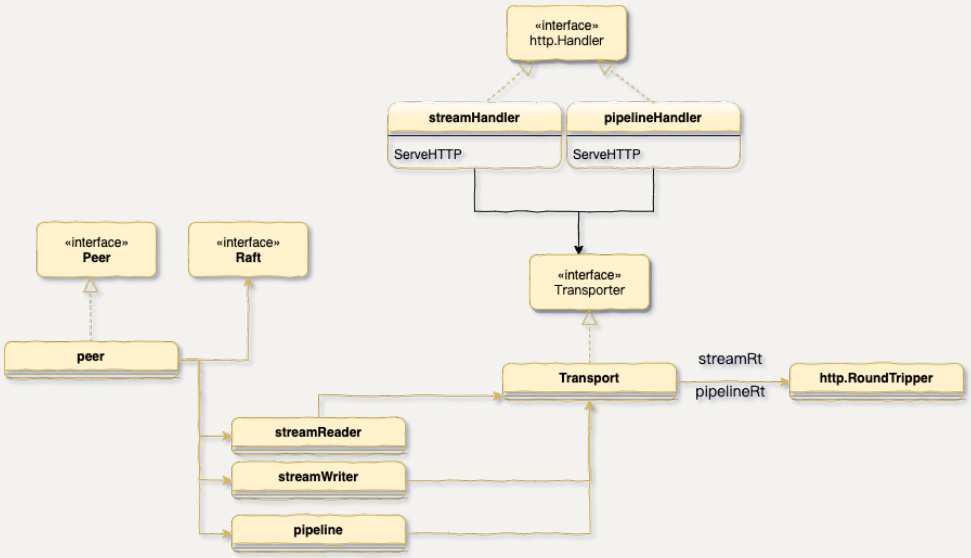
raft模块仅实现raft协议相关的逻辑，网络发送交给server raftnode来完成，etcd将 网络实现放在server的api包中，实际上这部分在etcd中并不属于核心域的内容，将其单独放入一个包也是可以的，也更容易实现网络层的替换。

etcd提供了2个消息发送的通道，**stream和pipeline**

- stream

  维护一个http的长连接，用于发送heartbeat,msgVote等发送频次高，包比较小的消息。streamRoundTripper设置默认的读写超时，因为读写包的size比较小；
- pipeline

用于发送snapshot等包比较大，但是频次比较低的消息。
pipelineRoundTripper设置默认读写超时为0即不超时，因为读写snapshotter耗时较长。



- pipelineHandler

  实现serveHttp方法，http的handler

- streamHandler

  实现serveHttp方法，http的handler

- streamReader

  streamReader start方法将启动一个run协程，该协程decode 消息，并把消息放入recvC channel中。

- streamwriter

  单独启动了一个协程run(),主要完成下面的事情

  a. 当其他的peer主动与当前的节点连接，peer就会将连接写入writerstream的connc，在streamwriter的run协程中获取这个连接并进行绑定。
  b. 定时发送心跳消息msgHeartbeat，由定时器触发
  c. 发送其他的消息

- pipline

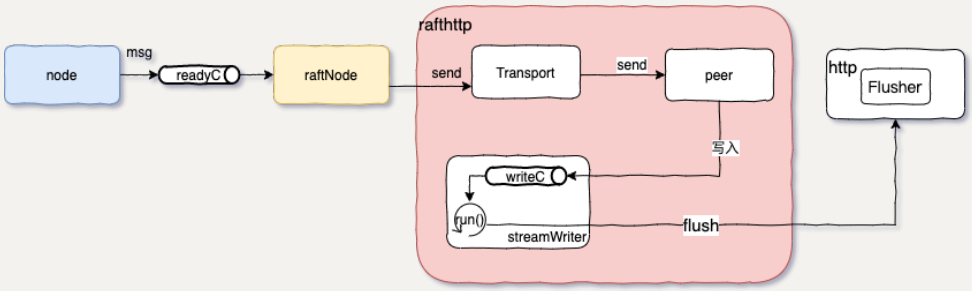
  发送snapshotmsg

- streamRoundTripper

  配置read write的timeout为5s

3.1. 发送流程

以发送一个MsgApp消息为例来看系统发送消息流程：

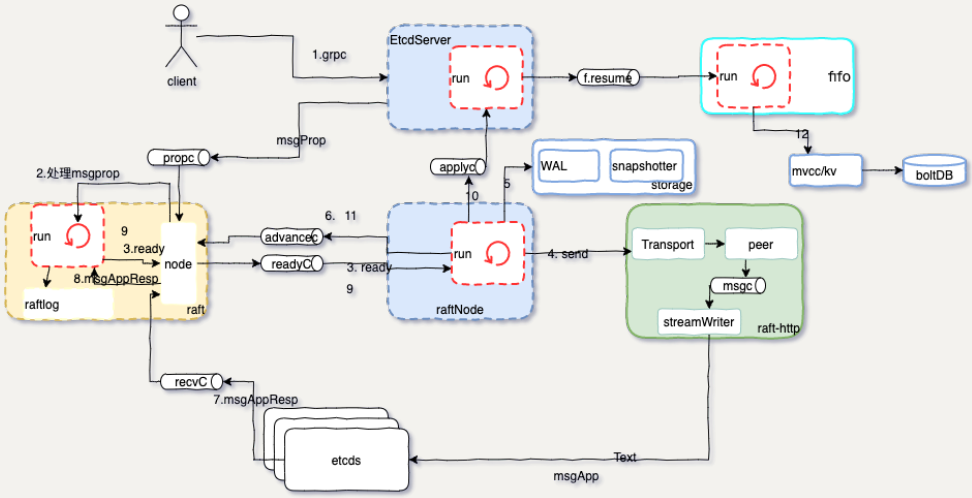


- raft模块处理log entry之后封装成ready写入readyC channel中
- server模块从channel中取出ready实例，调用rafthttp tranport发送消息。
- peer会根据消息的类型决定使用`streamwriter`还是`pipeline`来发送当前的消息，由于我们示例中使用了MsgApp因此最终写入streamwriter的`writec` channel中。
- streamWriter的run协程从中读出消息，调用**`http.Flusher`** 发送序列化后的消息。

> *Go 的 http.Flusher 常用于文件上传/下载/内容预处理等流式IO*

4. 数据流

4.1. client Put

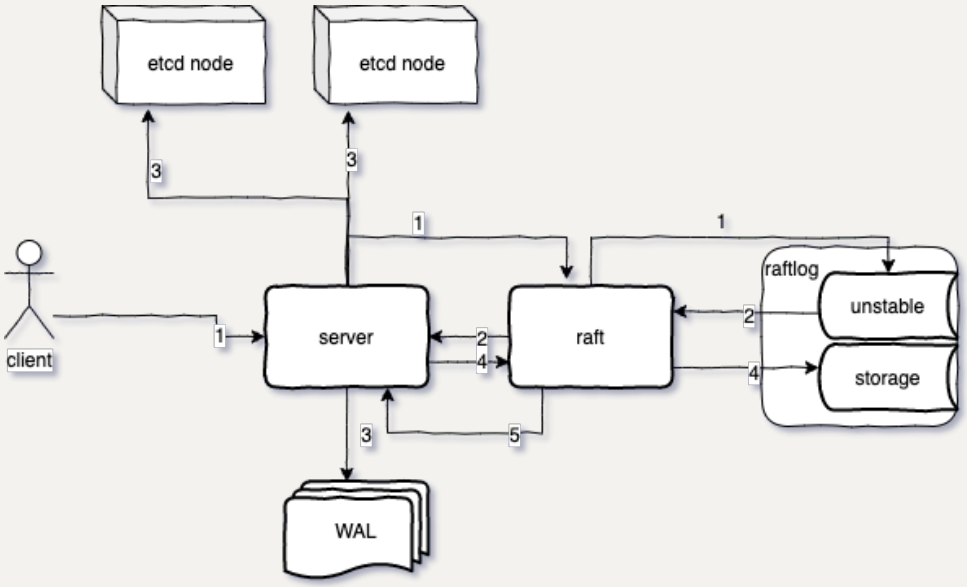


1. client 通过 grpc 发送一个 Put kv request，etcd server 的 rpc server 收到这个请求，client调用node的Propose接口创建一个`Propose`实例并放入 node 模块的`ProposeC` channel。
2. node 模块 run goroutine 监听在 propc Channel 中，收到 MsgProp Msg 之后，通过`raft.Step(Msg)`接口将其提交给 raft StateMachine 处理；
3. raft StateMachine 处理完这个 MsgProp Msg 会产生 1 个 Op log entry 和 2 个发送给另外两个副本的 `Append entries` 的 `MsgApp` messages，

node 模块会将这两个输出打包成 `Ready`，然后通过 readyc Channel 传递给 raftNode 模块的 coroutine；

4. raftNode 模块的 coroutine 通过 readyc 读取到 Ready，首先通过网络层将 2 个 append entries 的 messages 发送给两个副本(PS:这里是异步发送的)；
5. raftNode 模块的 coroutine 自己将 Op log entry 通过持久化层的 WAL 接口同步的写入 WAL 文件中
6. raftNode 模块的 coroutine 通过 advancec Channel 通知当前 Ready 已经处理完，请给我准备下一个 带出的 raft StateMachine 输出 Ready；
7. 其他副本的返回 Append entries 的 response： `MsgAppResp` message，会通过 node 模块的接口经过 `recvc` Channel 提交给 node 模块的 coroutine；
8. node 模块 coroutine 从 `recvc` Channel 读取到 MsgAppResp，然后提交给 raft StateMachine 处理。node 模块 coroutine 会驱动 raft StateMachine 得到关于这个 `committedEntires`，也就是一旦大多数副本返回了就可以 commit 了，node 模块 new 一个新的 Ready其包含了 `committedEntries`，通过 `readyc` Channel 传递给 raftNode 模块 coroutine 处理；
9. raftNode 模块 coroutine 从 readyc Channel 中读取 Ready结构
10. 取出已经 commit 的 committedEntries 通过 `applyc` 传递给另外一个 etcd server coroutine 处理，其会将每个`apply` 任务提交给 FIFOScheduler 调度异步处理，这个调度器可以保证 `apply` 任务按照顺序被执行，因为 `apply` 的执行是不能乱的；
11. raftNode 模块的 coroutine 通过 `advancec` Channel 通知当前 Ready 已经处理完，请给我准备下一个待处理的 raft StateMachine 输出 Ready；
12. FIFOScheduler 调度执行 apply 已经提交的 `committedEntries`
13. AppliedIndex 推进，通知 ReadLoop coroutine，满足 `applied index>= commit index` 的 read request 可以返回；
14. server调用网络层接口返回 client 成功。

5. WAL



首先参照上图回顾一下简单的请求提交流程：

1. 当etcd收到client的请求之后，请求中封装Entry交给raft模块处理，raft模块将`entry`保存到`raftlog`的`unstable`中；
2. raft模块封装entry成为一个`ready`实例，并放入`readyC` channel中等待server处理
3. server收到持久化的entry(pb序列化)，写入WAL（发送到其他的etcd节点）
4. 让raft模块把entry从`unstable`移动到`storage`中保存
5. 收到半数以上节点响应，leader节点认为该entry应该被commit，封装到ready实例返回给server；
6. server把ready实例中的entry记录应用到状态机中。

5.1. wal

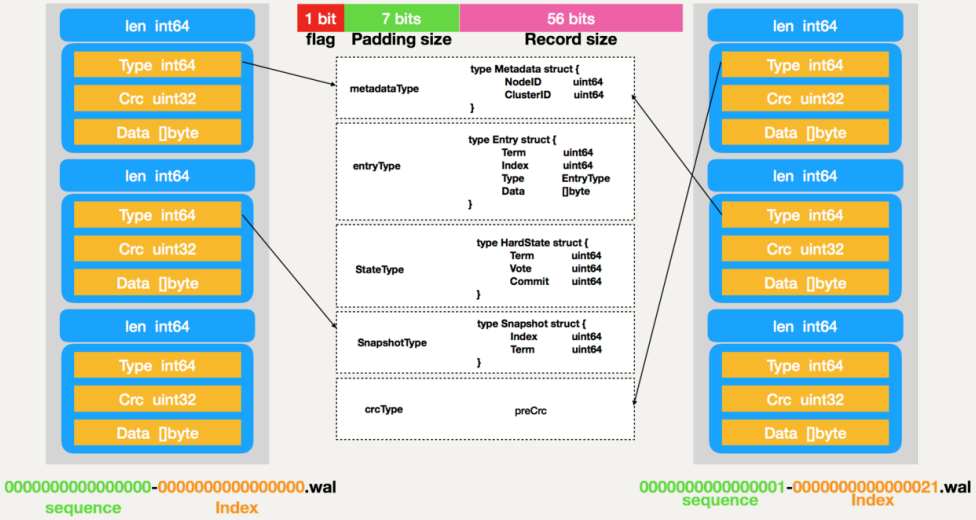
wal文件中的记录是Record，通过pb序列化之后保存，包含三个字段

- type: record的类型 `metaType`， `entryType`, `stateType`, `crcType` snapshotType(快照数据相关信息，不是完整的快照)
- Crc: 数据校验
- Data：type指定类型的数据

> *wal 文件中的snapshot类型只包含 Index，Term(任期)*

文件组织：

wal中有一个dir字段定义wal文件的目录，目录中存放wal文件，每个文件大小是64MB，metadata是第一个record块，创建的时候首先创建一个tmp文件，待完成之后重命名为wal文件格式，类似原子操作这种实现方式以后可以借鉴。



文件命名方式： 序号-起始的record id.wal

5.1.1. 读取

系统启动的时候会读取wal中的记录用于恢复系统数据和状态；每次读取都会指定index，不是每次从第一个日志文件开始，因此在文件打开的时候就需要searchIndex，

当open创建wal实例之后就能通过ReadAll方法读取日志，读取之后根据类型解析出entry，注意的是读取wal中切换文件是在decoder里面做的，这个实现有点奇怪。

5.1.2. 追加

消息经过raft模块处理之后封装成ready，raftnode取出ready 然后调用storage的Save接口存储entries。

wal将消息entry序列化并封装成record实例，并完成编码，crc校验，8字节对齐等然后写入PageWriter，PageWriter是带有缓冲区的writer，每当写满一个page（walPageBytes）就会触发一次flush操作。

5.1.3. 切换

当wal文件大小超过64MB时，就会触发日志文件的切换。

通过filePipeline获取一个新建临时文件，首先写入crc(上一个wal文件最后的crc)meta,state日志记录，然后重命名wal文件，重命名之后，调用fsync(fdatasync只能保存数据)确保重命名写入磁盘inode。保存offset，然后重新打开文件，直接定位到offset的位置。

5.2. snapshotter

wal文件是只追加不删除的，因此随着系统数据增多，wal文件会变得越来越多(因为切割成64MB的文件)，文件中可能存在很多重复或者已经删除的record数据。snapshotter是根据apply数量的创建，超过一定阈值就会触发创建snapshot。`ep.appliedi-ep.snapi <= s.Cfg.SnapshotCount`,合并wal并写入磁盘，减少了wal文件占用的磁盘空间。当etcd重启的时候先从snapshotter恢复，之后的数据再从wal文件中恢复，这样能提高系统的恢复速度。

比如当前etcd中有三个WAL文件，可以从这些文件的文件名知道其中存放数据的索引范围。`0000000000000001-0000000000000000.wal`，`0000000000000002-0000000000001000.wal`，`0000000000000003-0000000000002000.wal`。经过一次snapshot生成之后wal文件修改为`0000000000000001-0000000000002000.wal`，snapshot文件为`0000000000000001-0000000000001fff.snap`

6. Storage

- backend

  backend 封装了boltdb存储的接口
- watch

> *快慢和通知失败的情况*

watch是通过watchableStore实现的，每次在保存数据的时候，都会调用notify，通知到watchServer，watchserver进行过滤放入watchstream中，这里有一个实现的点需要注意，就是需要解决**快慢和发送通知失败的情况；**
etcd的解决这个问题的办法，使用了victim队列 unsynced
当synced队列发送失败就放入victims 同时将watcher移除，这样防止其阻塞了其他的watcher的通知，启用一个单独协程处理victims中的消息。

> *watch 一个prefix或者范围的情况*

通过一个adt.IntervalTree实现了一个范围的监控，adt.IntervalTree本质上是一个红黑树，将一个区间划分成一些单元区间，每个区间都对了应线段树的一个叶节点。

7. MVCC

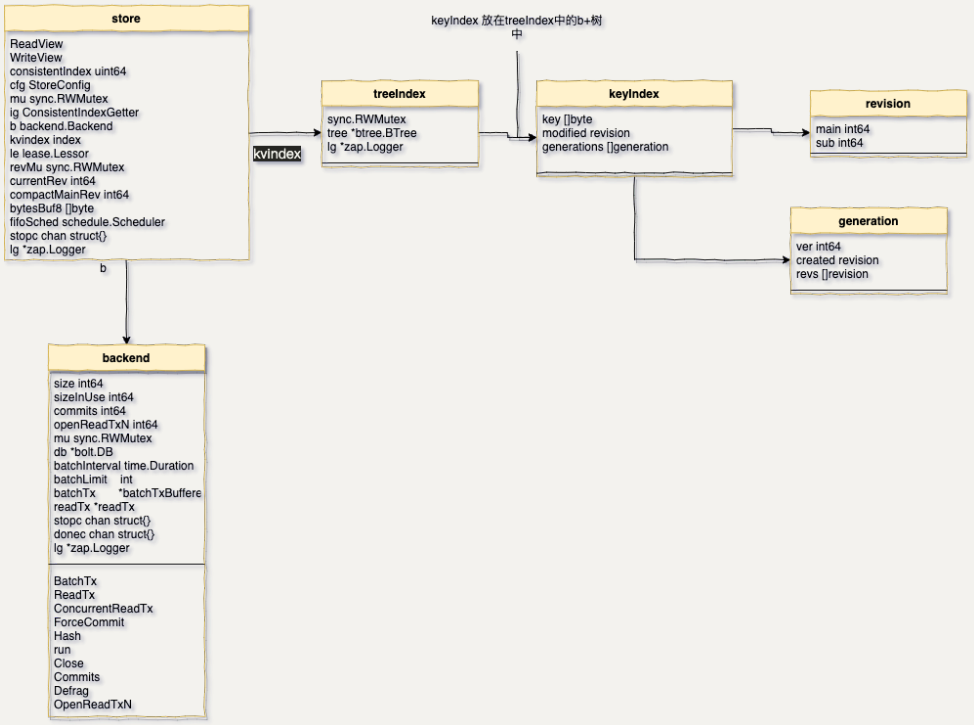
---

etcd并没有直接保存用于输入的key，而是生成了一个revision表示当前的key，通过revision获取value，这样实现了多版本的控制。

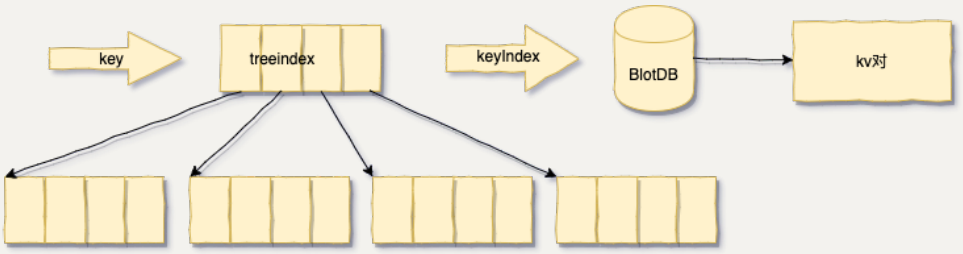
```
type revision struct {
    // 事务id
    main int64
    // 修改的id
    sub int64
}
```

这里的 main 属性对应事务 ID，全局递增不重复，可以看做一个逻辑时钟。sub 代表一次事务中不同的修改操作（如put和delete）编号，从0开始依次递增。所以在一次事务中，每一个修改操作所绑定的`revision`依次为`{txID, 0}`，`{txID, 1}`，`{txID, 2}`...

7.1. 实现方式



- store: mvcc的接口用于raft模块的存储
- backend: 封装了boltdb的存储接口
- treeindex：系统保存的是revision，但是用户查找是根据key获取value，因此需要一个映射关系，treeindex就是这样的一个b+tree，放入内存作为revision到key的映射
- keyindex: 封装了revision和generation
- generation：记录key从诞生到tomstone(删除)整个变化过程。由于key可能被创建，修改删除再创建这个反复过程因此generation是一个数组。



上面这个图描述的是从一个key到kv对的的映射过程。

7.2. transaction

etcd 提供了一种事务机制一次执行多个key value操作.

```
Txn().If(cond1, cond2, ...).Then(op1, op2, ...,).Else(op1, op2)
```

但是基于上述操作执行一个事务操作也比较麻烦，etcd还提供了一个更方便的操作接口STM() software transactional memory) 事务实现。

7.2.1. 一个账户向另外一个账户转账例子

```
  // 创建一个client
  cli, err := clientv3.New(clientv3.Config{Endpoints:
endpoints})
  if err != nil {
    log.Fatal(err)
  }
  defer cli.Close()
  totalAccounts := 5
  // 准备数据创建一些kv revision
  for i := 0; i < totalAccounts; i++ {
    k := fmt.Sprintf("accts/%d", i)
    if _, err = cli.Put(context.TODO(), k, "100"); err != nil
{
      log.Fatal(err)
    }
  }
  //创建一个transaction的执行函数
  exchange := func(stm concurrency.STM) error {
    from, to := 2, 3
    if from == to {
      // nothing to do
      return nil
    }
    // read values
    fromK, toK := fmt.Sprintf("accts/%d", from),
fmt.Sprintf("accts/%d", to)
```

```
    fromV, toV := stm.Get(fromK), stm.Get(toK)
    fromInt, toInt := 10, 20

    log.Printf("fromV %s,toV %s\n", fromV, toV)
    // 转账逻辑 transfer
    xfer := fromInt / 2
    fromInt, toInt = fromInt-xfer, toInt+xfer

    // 写入
    stm.Put(fromK, fmt.Sprintf("%d", fromInt))
    stm.Put(toK, fmt.Sprintf("%d", toInt))
    return nil
  }
  //执行交易，exchange为
  resp, err := concurrency.NewSTM(cli, exchange, )
```

分析一下上述代码的执行流程：

```
func NewSTM(c *v3.Client, apply func(STM) error, so
...stmOption) (*v3.TxnResponse, error) {
   opts := &stmOptions{ctx: c.Ctx()}
   for _, f := range so {
      f(opts)
   }
   if len(opts.prefetch) != 0 {
      f := apply
      apply = func(s STM) error {
         s.Get(opts.prefetch...)
         return f(s)
      }
   }
   return runSTM(mkSTM(c, opts), apply)
}
```

1. 创建一个stm
2. 执行stm

7.2.2. stm

etcd STM提供了四种事务级别：

- **SerializableSnapshot**

  可序列化的隔离，并且做写的冲突检查，默认就是采用这种隔离级别
- Serializable

第一次读的revision数据

- RepeatableReads

  同一个transaction中保证数据不变

- ReadCommitted

  读已提交数据

STM的事务级别通过stmOption指定，默认就是`SerializableSnapshot`。根据事务的隔离级别不同创建两种不同的stm实例和设置stm的conflicts，定义如下

- stmSerializable

```
type stmSerializable struct {
  stm
  prefetch map[string]*v3.GetResponse
}
```

- stm

```
type stm struct {
  client *v3.Client
  ctx    context.Context
  // rset holds read key values and revisions
  rset readSet
  // wset holds overwritten keys and their values
  wset writeSet
  // getOpts are the opts used for gets
  getOpts []v3.OpOption
  // conflicts computes the current conflicts on the
txn
  conflicts func() []v3.Cmp
}
```

- OpOption

```
type OpOption func(*Op)
```

7.2.3. 执行STM

`runstm`主要是循环执行(重试)以下三个步骤：

1. 重置stm，清空STM的读写缓存
2. 执行事务操作，apply函数
3. 提交事务

最终的commit执行 etcd client的transaction, `txn := s.client.Txn(s.ctx).If(s.conflicts()...).Then(s.wset.puts()...)` `conflict`函数是根据隔离级别定义的。

1. readCommited: 不做冲突检测
2. repeatableRead：
    a. 首先从读缓存中读，读缓存没有则从etcdserver中获取保证了事务过程中的可重复读。
    b. 用`readSet`数据的`ModRevision`做冲突检测，确保本事务读到的数据都是最新的。
3. serializable
    a. 第一次read的时需要保存key的revision，保证后面提交时候没有发生变化。
    b. 用`readSet`数据的`ModRevision`做冲突检测，确保本事务读到的数据都是最新的
4. SerializableSnapshot
    a. 第一次read的时需要保存key的revision，保证后面提交时候没有发生变化。
    b. 写冲突检查，读冲突检查都做。

8. boltDB

---

boltdb是一个kv存储的嵌入式数据库，核心代码只有2000行左右；

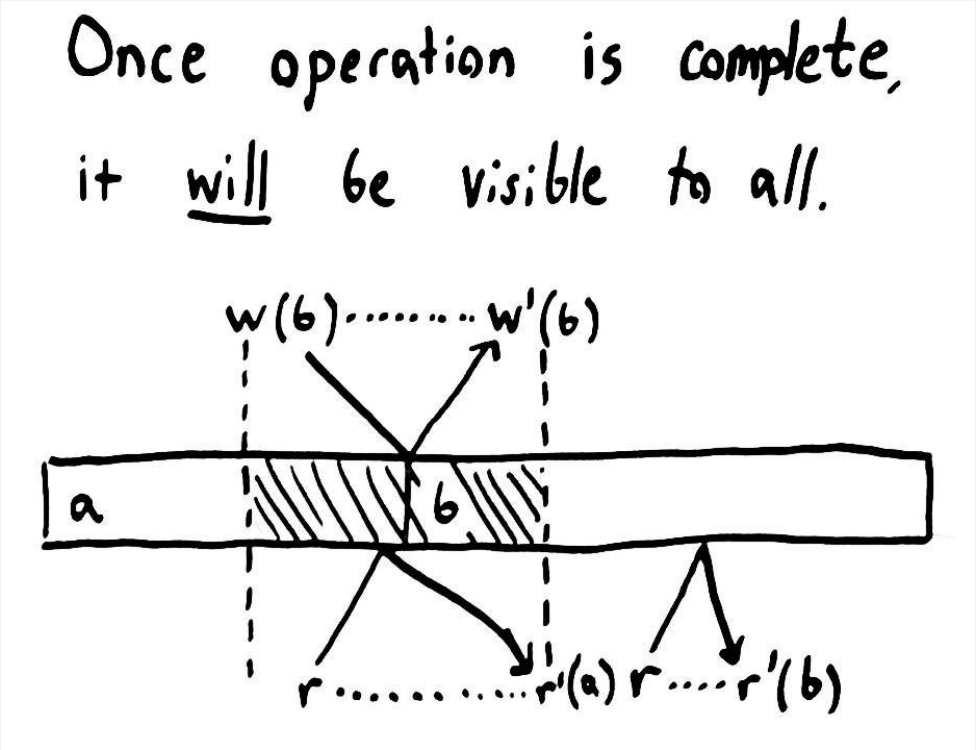
- 将数据库文件使用mmap映射到内存中
- 每次开始写事务的时候会把整个页面重新分配（不会修改原页面）写入硬盘，这种方式不会影响正在进行的读事务。
- bolt只支持并发的读和独占的写；通过两个交替的`metablock`和每次写事务重新分配页面的方式实现了mvcc；

后面会单独开启一个topic讲解boltdb存储引擎的实现。

9. 线性一致性

---

raft的目标是实现线性一致性，定义如下：



- invocation: 操作发起的时间；
- completion: 操作完成的时间。

> ***each operation appears to take effect atomically at some point between its invocation and completion.***

一致性的相关文件请 参照Strong consistency models

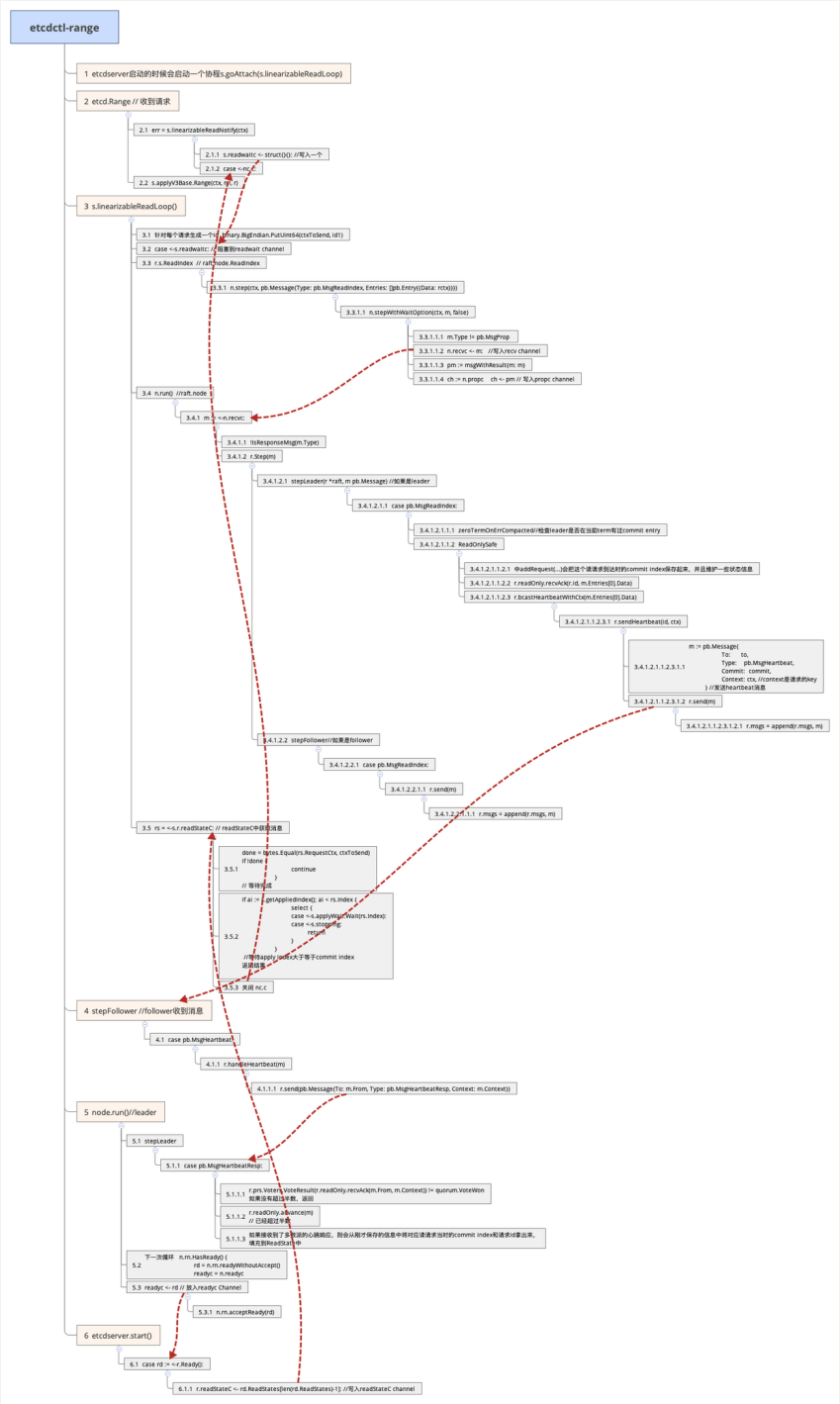
etcd读一致性的含义：

1. 保证处理读请求的是当前系统的leader，如果发生了脑裂，有了另外一个leader，就会导致stale read
2. 确保读到的数据能反映读请求 `invoke` 时的状态，或更新的状态。

> *Once an operation is complete, everyone must see it–or some later state.*

etcd实现读一致性的方法：

- readIndex



a. leader是否在当前的term提交过`log entry`如果未提交过则直接报错返回，因为新的leader必须提交之前`term`的未提交的的`log entry`
b. 保存收到消息时的`committed index`，这就是`readIndex`。`readIndex`是保证线性一致性的最小的commit index`
c. leader发送一轮心跳，确保自己仍然能获取大多数follower的响应。
d. 等待apply index大于等于commit index（apply到状态机）说明状态机至少应用到 ReadIndex 记录的 Log
e. 信号返回给读协程
f. 读协程从kv中获取值

- read lease

使用 `ReadIndex` 避免了 `log replication`，但是仍需要发送 `heartbeat` 来确保 `leadership` 的有效性。`Lease read` 通过 `lease` 机制可以避免 `heartbeat` 的开销，直接返回结果，但这种方式依赖时钟，不能保证线性一致性。

`raft-thesis` 中做法如下：

- `leader` 通过 `heartbeat` 来保持 `lease`：记录下发送 `heartbeat` 的时间 `start`，当收到多数派的响应后，就认为 `lease` 的有效期延续到 `start + election timeout / clock drift bound`， 在这个时间范围内不会产生新的 `leader`。
- 在 `lease` 期限的读请求不用和其他节点通信，直接认为当前 `leadership` 是有效的，只是绕过了 `heartbeat` 来确认 `leadetship` 有效性，其余的处理和 `ReadIndex` 相同。
- 要注意 `leadership transfer` 会导致新 `leader` 更早的产生，需要终止 `lease`。

这种机制只能保证收到了 `heartbeat` 的节点不会发起选举。

- wait Free

  省去等待apply。这样的 LeaseRead 在收到请求后会立刻进行读请求，不取 commit index 也不等状态机。由于 Raft 的强 Leader 特性，在租期内的 Client 收到的 Resp 由 Leader 的状态机产生，所以只要状态机满足线性一致，那么在 Lease 内，不管何时发生读都能满足线性一致性。有一点需要注意，只有在 Leader 的状态机应用了当前 term 的第一个 Log 后才能进行 LeaseRead。因为新选举产生的 Leader，它虽然有全部 committed Log，但它的状态机可能落后于之前的 Leader，状态机应用到当前 term 的 Log 就保证了新 Leader 的状态机一定新于旧 Leader，之后肯定不会出现 stale read。

10. 优化

10.1. multi-raft

单个raft存在kv场景中存在一定的局限性：

1. 系统的存储容量受制于单机的存储容量（使用分布式存储除外）
2. 系统的性能受制于单机的性能（读写请求都由Leader节点处理）

而在实际应用中通常把数据分区，每个区数据都有自己的副本，副本之间通过raft保证数据的一致性。定义如下：

> *the data is divided into ranges, each with its own consensus group. This means that each node may be participating in hundreds of thousands of consensus groups. This presents some unique challenges, which we have addressed by introducing a layer on top of Raft that we call MultiRaft.*

muti-raft实现的困难：

1. 数据何如分片
2. 分片中的数据越来越大，需要分裂产生更多的分片，组成更多Raft-Group
3. 分片的调度，让负载在系统中更平均（分片副本的迁移，补全，Leader切换等等）
4. 一个节点上，所有的Raft-Group复用链接（否则Raft副本之间两两建链，链接爆炸了）
5. 如何处理stale的请求（例如Proposal和Apply的时候，当前的副本不是Leader、分裂了、被销毁了等等）
6. Snapshot如何管理（限制Snapshot，避免带宽、CPU、IO资源被过度占用）

具体可以参照tidb和cockroach的实现。

10.2. 优化读性能性能

- Lease holder
  cockroach 增加了一个 lease holder角色来 就能保证线性一致性，不需要 ReadIndex 的 heartbeat，也不会像 lease read 损失一致性。
- Quorum read
  Raft 的读虽然可以发送给 follower，但还是要从 leader 获取 readIndex，leader 的压力会很大。使用 quorum read 可以利用 follower 读，减小 leader 的压力， 提高读的吞吐量和性能：Improving Read Scalability in Raft-like consensus protocols

参考

线性一致性和raft

Improving Read Scalability in Raft-like consensus protocols

Serializability and Distributed Software Transactional Memory with etcd3

CockroachDB

Elasticell-Multi-Raft实现

Raft 笔记