



下载APP



18 | 实战：如何基于Raft从0到1构建一个支持多存储引擎分布式KV服务？

2021-03-01 唐聪

etcd实战课

[进入课程 >](#)**讲述：王超凡**

时长 17:48 大小 16.31M



你好，我是唐聪。

通过前面课程的学习，我相信你已经对 etcd 基本架构、核心特性有了一定理解。如果让你基于 Raft 协议，实现一个简易的类 etcd、支持多存储引擎的分布式 KV 服务，并能满足读多写少、读少写多的不同业务场景诉求，你知道该怎么动手吗？

纸上得来终觉浅，绝知此事要躬行。



今天我就和你聊聊如何实现一个类 etcd、支持多存储引擎的 KV 服务，我们将基于 etcd 自带的 [raftexample](#) 项目快速构建它。

为了方便后面描述，我把它命名为 metcd（表示微型的 etcd），它是 raftexample 的加强版。希望通过 metcd 这个小小的实战项目，能够帮助你进一步理解 etcd 乃至分布式存储服务的核心架构、原理、典型问题解决方案。

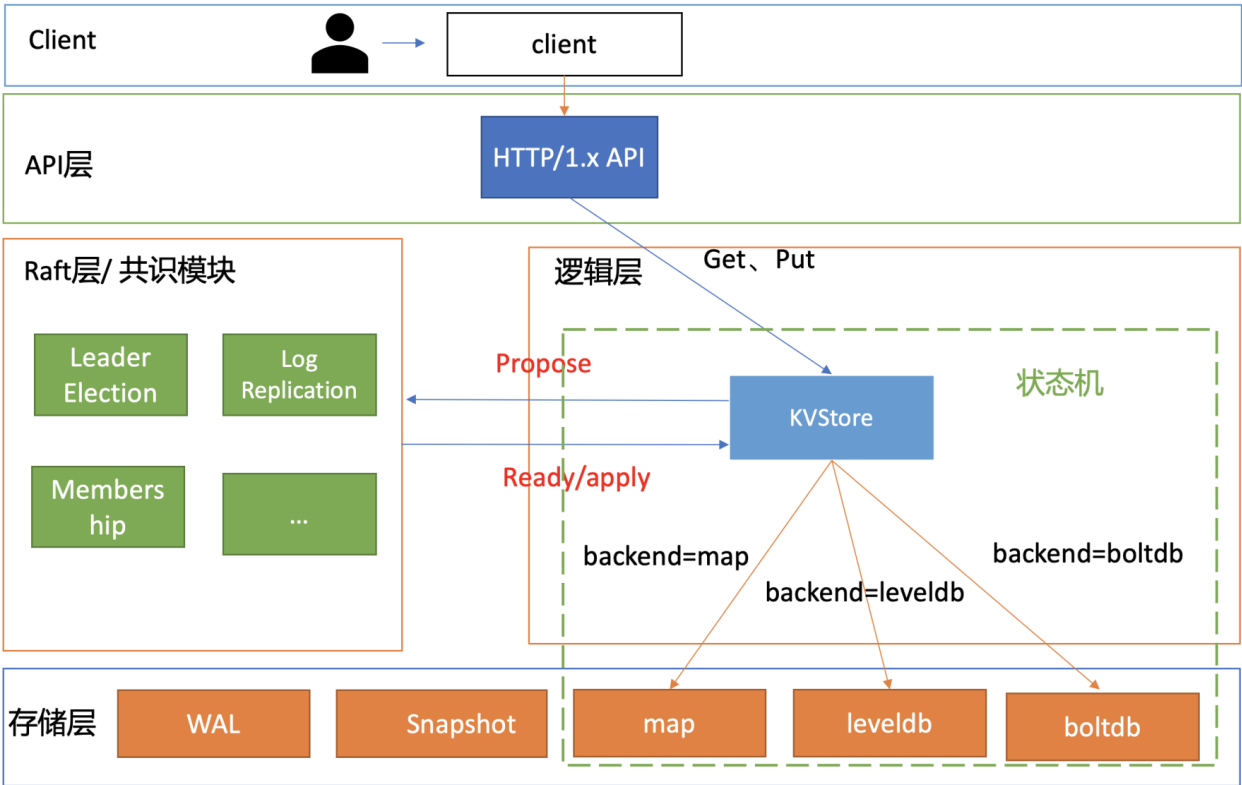
同时在这个过程中，我将详细为你介绍 etcd 的 Raft 算法工程实现库、不同类型存储引擎的优缺点，拓宽你的知识视野，为你独立分析 etcd 源码，夯实基础。

整体架构设计

在和你深入聊代码细节之前，首先我和你从整体上介绍下系统架构。

下面是我给你画的 metcd 整体架构设计，它由 API 层、Raft 层的共识模块、逻辑层及存储层组成的状态机组成。

接下来，我分别和你简要分析下 API 设计及复制状态机。



API 设计

API 是软件系统对外的语言，它是应用编程接口的缩写，由一组接口定义和协议组成。

在设计 API 的时候，我们往往会考虑以下几个因素：

性能。如 etcd v2 使用的是简单的 HTTP/1.x，性能上无法满足大规模 Kubernetes 集群等场景的诉求，因此 etcd v3 使用的是基于 HTTP/2 的 gRPC 协议。

易用性、可调试性。如有的内部高并发服务为了满足性能等诉求，使用的是 UDP 协议。相比 HTTP 协议，UDP 协议显然在易用性、可调试性上存在一定的差距。

开发效率、跨平台、可移植性。相比基于裸 UDP、TCP 协议设计的接口，如果你使用 Protobuf 等 IDL 语言，它支持跨平台、代码自动自动生成，开发效率更高。

安全性。如相比 HTTP 协议，使用 HTTPS 协议可对通信数据加密更安全，可适用于不安全的网络环境（比如公网传输）。

接口幂等性。幂等性简单来说，就是同样一个接口请求一次与多次的效果一样。若你的接口对外保证幂等性，则可降低使用者的复杂度。

因为我们场景的是 POC(Proof of concept)、Demo 开发，因此在 metcd 项目中，我们优先考虑点是易用性、可调试性，选择 HTTP/1.x 协议，接口上为了满足 key-value 操作，支持 Get 和 Put 接口即可。

假设 metcd 项目使用 3379 端口，Put 和 Get 接口，如下所示。

Put 接口，设置 key-value

```
1 curl -L http://127.0.0.1:3379/hello -XPUT -d world
```

[复制代码](#)

Get 接口，查询 key-value

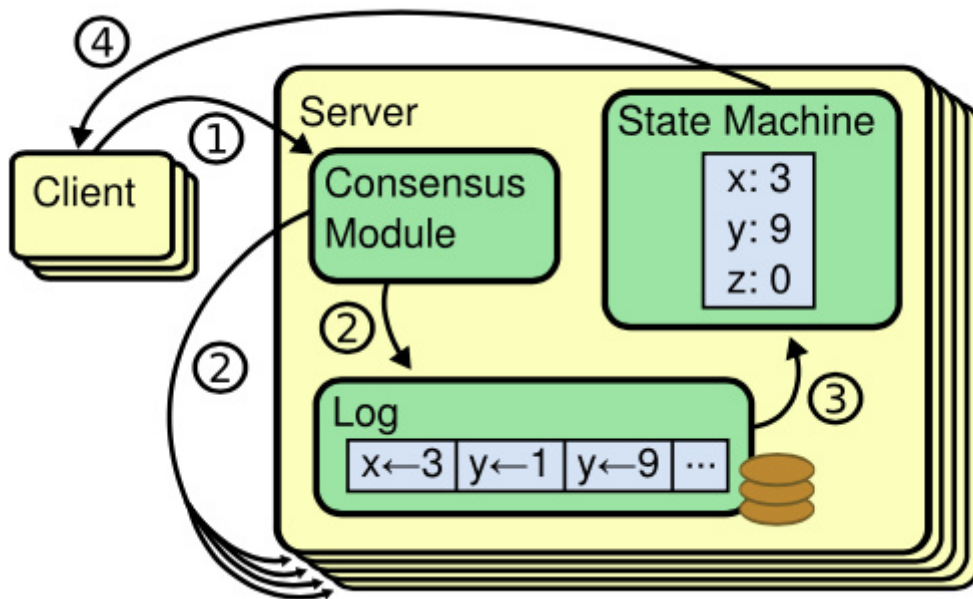
```
1 curl -L http://127.0.0.1:3379/hello
2 world
```

[复制代码](#)

复制状态机

了解完 API 设计，那最核心的复制状态机是如何工作的呢？

我们知道 etcd 是基于下图复制状态机实现的分布式 KV 服务，复制状态机由共识模块、日志模块、状态机组成。



我们的实战项目 metcd，也正是使用与之一样的模型，并且使用 etcd 项目中实现的 Raft 算法库作为共识模块，此算法库已被广泛应用在 etcd、cockroachdb、dgraph 等开源项目中。

以下是复制状态机的写请求流程：

client 发起一个写请求（put hello = world）；

server 向 Raft 共识模块提交请求，共识模块生成一个写提案日志条目。若 server 是 Leader，则把日志条目广播给其他节点，并持久化日志条目到 WAL 中；

当一半以上节点持久化日志条目后，Leader 的共识模块将此日志条目标记为已提交（committed），并通知其他节点提交；


server 从共识模块获取已经提交的日志条目，异步应用到状态机存储中（boltdb/leveldb/memory），然后返回给 client。

多存储引擎

了解完复制状态机模型后，我和你再深入介绍下状态机。状态机中最核心模块当然是存储引擎，那要如何同时支持多种存储引擎呢？

metcd 项目将基于 etcd 本身自带的 raftexample 项目进行快速开发，而 raftexample 本身只支持内存存储。

因此我们通过将 KV 存储接口进行抽象化设计，实现支持多存储引擎。KVStore interface 的定义如下所示。

 复制代码

```
1  type KVStore interface {
2      // LookUp get key value
3      Lookup(key string) (string, bool)
4
5      // Propose propose kv request into raft state machine
6      Propose(k, v string)
7
8      // ReadCommits consume entry from raft state machine into KvStore map until
9      ReadCommits(commitC <-chan *string, errorC <-chan error)
10
11     // Snapshot return KvStore snapshot
12     Snapshot() ([]byte, error)
13
14     // RecoverFromSnapshot recover data from snapshot
15     RecoverFromSnapshot(snapshot []byte) error
16
17     // Close close backend databases
18     Close() error
19 }
```

基于 KV 接口抽象化的设计，我们只需要针对具体的存储引擎，实现对应的操作即可。

我们期望支持三种存储引擎，分别是内存 map、boltdb、leveldb，并做一系列简化设计。一组 metcd 实例，通过 metcd 启动时的配置来决定使用哪种存储引擎。不同业务场景不同实例，比如读多写少的存储引擎可使用 boltdb，写多读少的可使用 leveldb。

接下来我和你重点介绍下存储引擎的选型及原理。

boltdb

boltdb 是一个基于 B+ tree 实现的存储引擎库，在 [🔗10](#)中我已和你详细介绍过原理。

boltdb 为什么适合读多写少？

对于读请求而言，一般情况下它可直接从内存中基于 B+ tree 遍历，快速获取数据返回给 client，不涉及经过磁盘 I/O。

对于写请求，它基于 B+ tree 查找写入位置，更新 key-value。事务提交时，写请求包括 B+ tree 重平衡、分裂、持久化 dirty page、持久化 freelist、持久化 meta page 流程。同时，dirty page 可能分布在文件的各个位置，它发起的是随机写磁盘 I/O。

因此在 boltdb 中，完成一个写请求的开销相比读请求是大很多的。正如我在 [@16](#)和 [@17](#) 中给你介绍的一样，一个 3 节点的 8 核 16G 空集群，线性读性能可以达到 19 万 QPS，而写 QPS 仅为 5 万。

leveldb

那要如何设计适合写多读少的存储引擎呢？

最简单的思路当然是写内存最快。可是内存有限的，无法支撑大容量的数据存储，不持久化数据会丢失。

那能否直接将数据顺序追加到文件末尾（AOF）呢？因为磁盘的特点是顺序写性能比较快。

当然可以。[@Bitcask](#)存储模型就是采用 AOF 模式，把写请求顺序追加到文件。Facebook 的图片存储 [@Haystack](#)根据其论文介绍，也是使用类似的方案来解决大规模写入痛点。

那在 AOF 写入模型中如何实现查询数据呢？

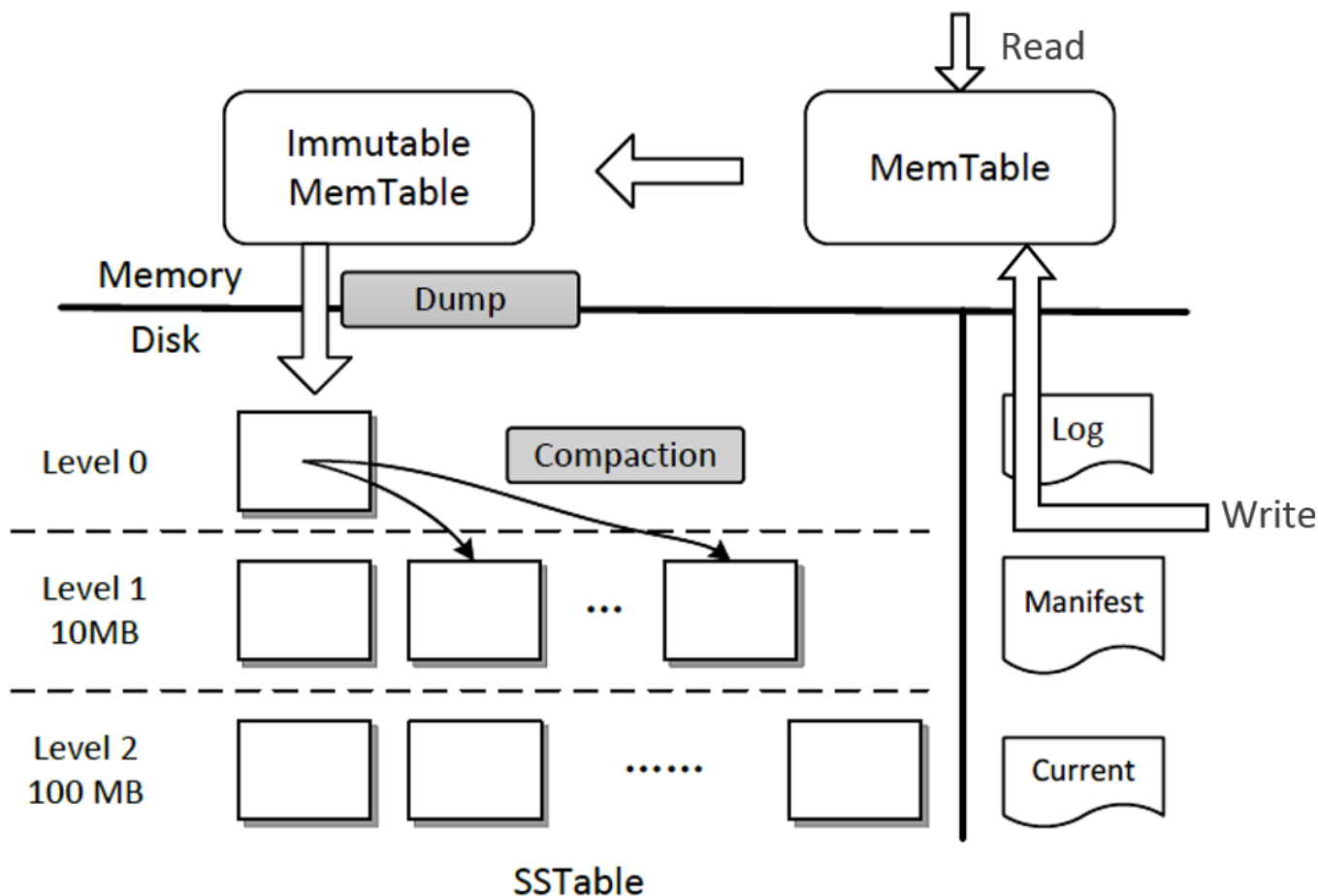
很显然通过遍历文件一个个匹配 key 是可以的，但是它的性能是极差的。为了实现高性能的查询，最理想的解决方案从直接从内存中查询，但是内存是有限的，那么我们能否通过内存索引来记录一个 key-value 数据在文件中的偏移量，实现从磁盘快速读取呢？

是的，这正是 [@Bitcask](#)存储模型的查询的实现，它通过内存哈希表维护各个 key-value 数据的索引，实现了快速查找 key-value 数据。不过，内存中虽然只保存 key 索引信息，但是当 key 较多时，其对内存要求依然比较高。

快速了解完存储引擎提升写性能的核心思路（随机写转化为顺序写）之后，那 leveldb 它的原理是怎样的呢？与 Bitcask 存储模型有什么不一样？

leveldb 是基于 LSM tree(log-structured merge-tree) 实现的 key-value 存储，它的架构如下图所示（[引用自微软博客](#)）。

它提升写性能的核心思路同样是将随机写转化为顺序写磁盘 WAL 文件和内存，结合了我们上面讨论的写内存和磁盘两种方法。数据持久化到 WAL 文件是为了确保机器 crash 后数据不丢失。



那么它要如何解决内存不足和查询的痛点问题呢？

核心解决方案是分层的设计和基于一系列对象的转换和压缩。接下来我给你分析一下上面架构图写流程和后台 compaction 任务：

首先写请求顺序写入 Log 文件 (WAL)；

更新内存的 Memtable。leveldb Memtable 后端数据结构实现是 skiplist，skiplist 相比平衡二叉树，实现简单却同样拥有高性能的读写；

当 Memtable 达到一定的阈值时，转换成不可变的 Memtable，也就是只读不可写；

leveldb 后台 Compact 任务会将不可变的 Memtable 生成 SSTable 文件，它有序地存储一系列 key-value 数据。注意 SST 文件按写入时间进行了分层，Level 层次越小数据越新。Manifest 文件记录了各个 SSTable 文件处于哪个层级、它的最小与最大 key 范围；

当某个 level 下的 SSTable 文件数目超过一定阈值后，Compact 任务会从这个 level 的 SSTable 中选择一个文件 ($level > 0$)，将其和高一层级的 $level+1$ 的 SSTable 文件合并；

注意 level 0 是由 Immutable 直接生成的，因此 level 0 SSTable 文件中的 key-value 存在相互重叠。而 $level > 0$ 时，在和更高一层 SSTable 合并过程中，参与的 SSTable 文件是多个，leveldb 会确保各个 SSTable 中的 key-value 不重叠。

了解完写流程，读流程也就简单了，核心步骤如下：

从 Memtable 跳跃表中查询 key；

未找到则从 Immutable 中查找；

Immutable 仍未命中，则按照 leveldb 的分层属性，因 level 0 SSTable 文件是直接由 Immutable 生成的，level 0 存在特殊性，因此你需要从 level 0 遍历 SSTable 查找 key；


level 0 中若未命中，则从 level 1 乃至更高的层次查找。level 大于 0 时，各个 SSTable 中的 key 是不存在相互重叠的。根据 manifest 记录的 key-value 范围信息，可快速定位到具体的 SSTable。同时 leveldb 基于 [bloom filter](#) 实现了快速筛选 SSTable，因此查询效率较高。

更详细原理你可以参考一下 [leveldb](#) 源码。

实现分析


从 API 设计、复制状态机、多存储引擎支持等几个方面你介绍了 metcd 架构设计后，接下来我就和你重点介绍下共识模块、状态机支持多存储引擎模块的核心实现要点。

Raft 算法库

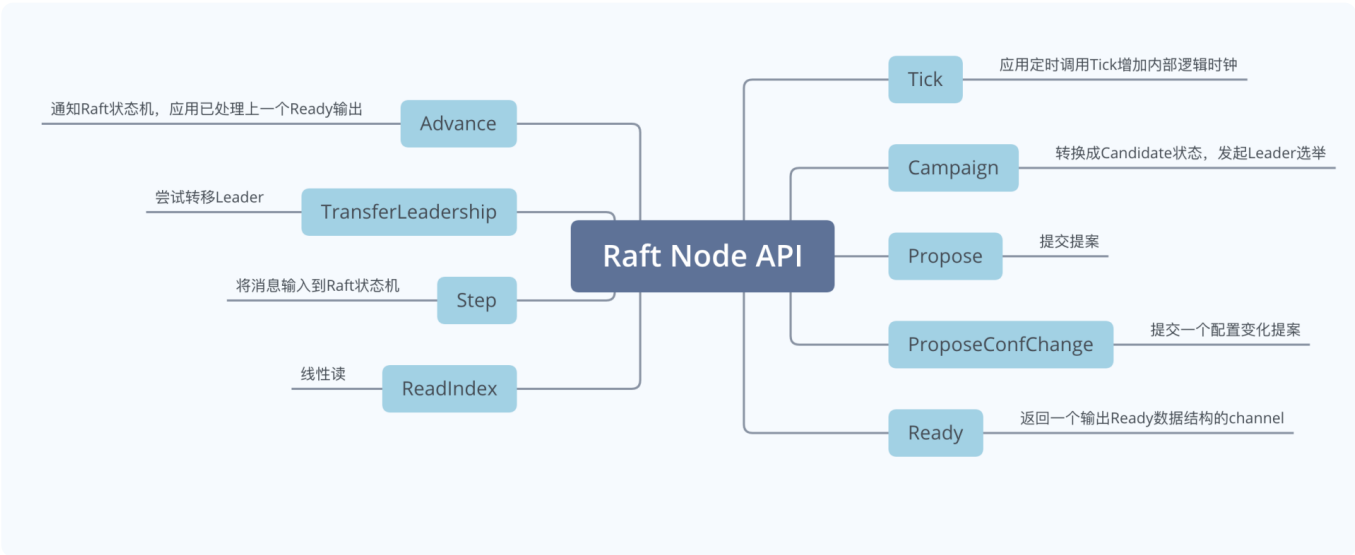
共识模块使用的是 etcd  Raft 算法库，它是一个经过大量业务生产环境检验、具备良好可扩展性的共识算法库。

它提供了哪些接口给你使用？如何提交一个提案，并且获取 Raft 共识模块输出结果呢？

Raft API

Raft 作为一个库，它对外最核心的对象是一个名为  Node的数据结构。Node 表示 Raft 集群中的一个节点，它的输入与输出接口如下图所示，下面我重点和你介绍它的几个接口功能：

- Campaign，状态转换成 Candidate，发起新一轮 Leader 选举；
- Propose，提交提案接口；
- Ready，Raft 状态机输出接口，它的返回是一个输出 Ready 数据结构类型的管道，应用需要监听此管道，获取 Ready 数据，处理其中的各个消息（如持久化未提交的日志条目到 WAL 中，发送消息给其他节点等）；
- Advance，通知 Raft 状态机，应用已处理上一个输出的 Ready 数据，等待发送下一个 Ready 数据；
- TransferLeadership，尝试将 Leader 转移到某个节点；
- Step，向 Raft 状态机提交收到的消息，比如当 Leader 广播完 MsgApp 消息给 Follower 节点后，Leader 收到 Follower 节点回复的 MsgAppResp 消息时，就通过 Step 接口将此消息提交给 Raft 状态机驱动其工作；
- ReadIndex，用于实现线性读。



上面提到的 Raft 状态机的输出 [🔗 Ready 结构](#) 含有哪些信息呢？下图是其详细字段，含义如下：

SoftState，软状态。包括集群 Leader 和节点状态，不需要持久化到 WAL；

pb.HardState，硬状态。与软状态相反，包括了节点当前 Term、Vote 等信息，需要持久化到 WAL 中；

ReadStates，用于线性一致性读；

Entries，在向其他节点发送消息之前需持久化到 WAL 中；

Messages，持久化 Entries 后，发送给其他节点的消息；

Committed Entries，已提交的日志条目，需要应用到存储状态机中；

Snapshot，快照需保存到持久化存储中；


MustSync，HardState 和 Entries 是否要持久化到 WAL 中；



了解完 API 后，我们接下来继续看看代码如何使用 Raft 的 Node API。

正如我在 [🔗 04](#) 中和你介绍的，etcd Raft 库的设计抽象了网络、Raft 日志存储等模块，它本身并不会进行网络、存储相关的操作，上层应用需结合自己业务场景选择内置的模块或自定义实现网络、存储、日志等模块。

因此我们在使用 Raft 库时，需要先自定义好相关网络、存储等模块，再结合上面介绍的 Raft Node API，就可以完成一个 Node 的核心操作了。其数据结构定义如下：

 复制代码

```

1 // A key-value stream backed by raft
2 type raftNode struct {
3     proposeC    <-chan string           // proposed messages (k,v)
4     confChangeC <-chan raftpb.ConfChange // proposed cluster config changes
5     commitC     chan<- *string           // entries committed to log (k,v)
6     errorC      chan<- error           // errors from raft session
7     id          int                // client ID for raft session
8     .....
9     node        raft.Node
10    raftStorage *raft.MemoryStorage
11    wal         *wal.WAL
12    transport *rafthttp.Transport
13 }

```

这个数据结构名字叫 `raftNode`，它表示 Raft 集群中的一个节点。它是由我们业务应用层设计的一个组合结构。从结构体定义中你可以看到它包含了 Raft 核心数据结构 `Node(raft.Node)`、Raft 日志条目内存存储模块 (`raft.MemoryStorage`)、WAL 持久化模块 (`wal.WAL`) 以及网络模块 (`rafthttp.Transport`)。

同时，它提供了三个核心的管道与业务逻辑模块、存储状态机交互：

`proposeC`，它用来接收 client 发送的写请求提案消息；


`confChangeC`，它用来接收集群配置变化消息；

`commitC`，它用来输出 Raft 共识模块已提交的日志条目消息。

在 `metcd` 项目中因为我们是直接基于 `raftexample` 定制开发，因此日志持久化存储、网络都使用的是 `etcd` 自带的 `WAL` 和 `rafthttp` 模块。

[🔗 WAL](#) 模块中提供了核心的保存未持久化的日志条目和快照功能接口，你可以参考 [🔗 03 节](#) 写请求中我和你介绍的原理。

[🔗 rafthttp](#) 模块基于 HTTP 协议提供了各个节点间的消息发送能力，`metcd` 使用如下：

 复制代码

```

1 rc.transport = &rafthttp.Transport{
2     Logger:      zap.NewExample(),
3     ID:          types.ID(rc.id),

```

```

4   ClusterID:    0x1000,
5   Raft:         rc,
6   ServerStats: stats.NewServerStats("", ""),
7   LeaderStats: stats.NewLeaderStats(strconv.Itoa(rc.id)),
8   ErrorC:       make(chan error),
9

```

搞清楚 Raft 模块的输入、输出 API，设计好 raftNode 结构，复用 etcd 的 WAL、网络等模块后，接下来我们就只需要实现如下两个循环逻辑，处理业务层发送给 proposeC 和 confChangeC 消息、将 Raft 的 Node 输出 Ready 结构进行相对应的处理即可。精简后的代码如下所示：

 复制代码

```

1 func (rc *raftNode) serveChannels() {
2     // send proposals over raft
3     go func() {
4         confChangeCount := uint64(0)
5         for rc.proposeC != nil && rc.confChangeC != nil {
6             select {
7                 case prop, ok := <-rc.proposeC:
8                     if !ok {
9                         rc.proposeC = nil
10                    } else {
11                        // blocks until accepted by raft state machine
12                        rc.node.Propose(context.TODO(), []byte(prop))
13                    }
14
15                    case cc, ok := <-rc.confChangeC:
16                        if !ok {
17                            rc.confChangeC = nil
18                        } else {
19                            confChangeCount++
20                            cc.ID = confChangeCount
21                            rc.node.ProposeConfChange(context.TODO(), cc)
22                        }
23                    }
24                }
25            }()
26
27            // event loop on raft state machine updates
28            for {
29                select {
30                    case <-ticker.C:
31                        rc.node.Tick()
32
33                        // store raft entries to wal, then publish over commit channel
34                    case rd := <-rc.node.Ready():
35                        rc.wal.Save(rd.HardState, rd.Entries)

```

```
36         if !raft.IsEmptySnap(rd.Snapshot) {
37             rc.saveSnap(rd.Snapshot)
38             rc.raftStorage.ApplySnapshot(rd.Snapshot)
39             rc.publishSnapshot(rd.Snapshot)
40         }
41         rc.raftStorage.Append(rd.Entries)
42         rc.transport.Send(rd.Messages)
43         if ok := rc.publishEntries(rc.entriesToApply(rd.CommittedEntries)); !
44             rc.stop()
45             return
46         }
47         rc.maybeTriggerSnapshot()
48         rc.node.Advance()
49     }
50 }
51 }
```

代码简要分析如下：

从 proposeC 中取出提案消息，通过 raft.Node.Propose API 提交提案；

从 confChangeC 取出配置变更消息，通过 raft.Node.ProposeConfChange API 提交配置变化消息；

从 raft.Node 中获取 Raft 算法状态机输出到 Ready 结构中，将 rd.Entries 和 rd.HardState 通过 WAL 模块持久化，将 rd.Messages 通过 rafthttp 模块，发送给其他节点。将 rd.CommittedEntries 应用到业务存储状态机。

以上就是 Raft 实现的核心流程，接下来我来和你聊聊业务存储状态机。


支持多存储引擎

在整体架构设计时，我和你介绍了为了使 metcd 项目能支撑多存储引擎，我们将 KVStore 进行了抽象化设计，因此我们只需要实现各个存储引擎相对应的 API 即可。

这里我以 Put 接口为案例，分别给你介绍下各个存储引擎的实现。

boltdb

首先是 boltdb 存储引擎，它的实现如下，你也可以去 [🔗10](#)里回顾一下它的 API 和原理。


 复制代码

```
1 func (s *boltdbKVStore) Put(key, value string) error {
2     s.mu.Lock()
3     defer s.mu.Unlock()
4     // Start a writable transaction.
5     tx, err := s.db.Begin(true)
6     if err != nil {
7         return err
8     }
9     defer tx.Rollback()
10
11     // Use the transaction...
12     bucket, err := tx.CreateBucketIfNotExists([]byte("keys"))
13     if err != nil {
14         log.Printf("failed to put key %s, value %s, err is %v", key, value, err)
15         return err
16     }
17     err = bucket.Put([]byte(key), []byte(value))
18     if err != nil {
19         log.Printf("failed to put key %s, value %s, err is %v", key, value, err)
20         return err
21     }
22
23     // Commit the transaction and check for error.
24     if err := tx.Commit(); err != nil {
25         log.Printf("failed to commit transaction, key %s, err is %v", key, err)
26         return err
27     }
28     log.Printf("backend:%s,put key:%s,value:%s succ", s.config.backend, key, va
29     return nil
```

leveldb


其次是 leveldb，我们使用的是 [goleveldb](#)，它基于 Google 开源的 c++ [leveldb](#) 版本实现。它提供的常用 API 如下所示。

通过 OpenFile API 创建或打开一个 leveldb 数据库。

 复制代码


```
1 db, err := leveldb.OpenFile("path/to/db", nil)
2 ...
3 defer db.Close()
```

通过 DB.Get/Put/Delete API 操作数据。

 复制代码

```
1 data, err := db.Get([]byte("key"), nil)
2 ...
3 err = db.Put([]byte("key"), []byte("value"), nil)
4 ...
5 err = db.Delete([]byte("key"), nil)
6 ...
```

了解其接口后，通过 goleveldb 的库，client 调用就非常简单了，下面是 metcd 项目中，leveldb 存储引擎 Put 接口的实现。

 复制代码


```
1 func (s *leveldbKVStore) Put(key, value string) error {
2     err := s.db.Put([]byte(key), []byte(value), nil)
3     if err != nil {
4         log.Printf("failed to put key %s, value %s, err is %v", key, value, err)
5         return err
6     }
7     log.Printf("backend:%s,put key:%s,value:%s succ", s.config.backend, key, va
8     return nil
9 }
```

读写流程

介绍完在 metcd 项目中如何使用 Raft 共识模块、支持多存储引擎后，我们再从整体上介绍下在 metcd 中写入和读取一个 key-value 的流程。

写流程

当你通过如下 curl 命令发起一个写操作时，写流程如下面架构图序号所示：

 复制代码

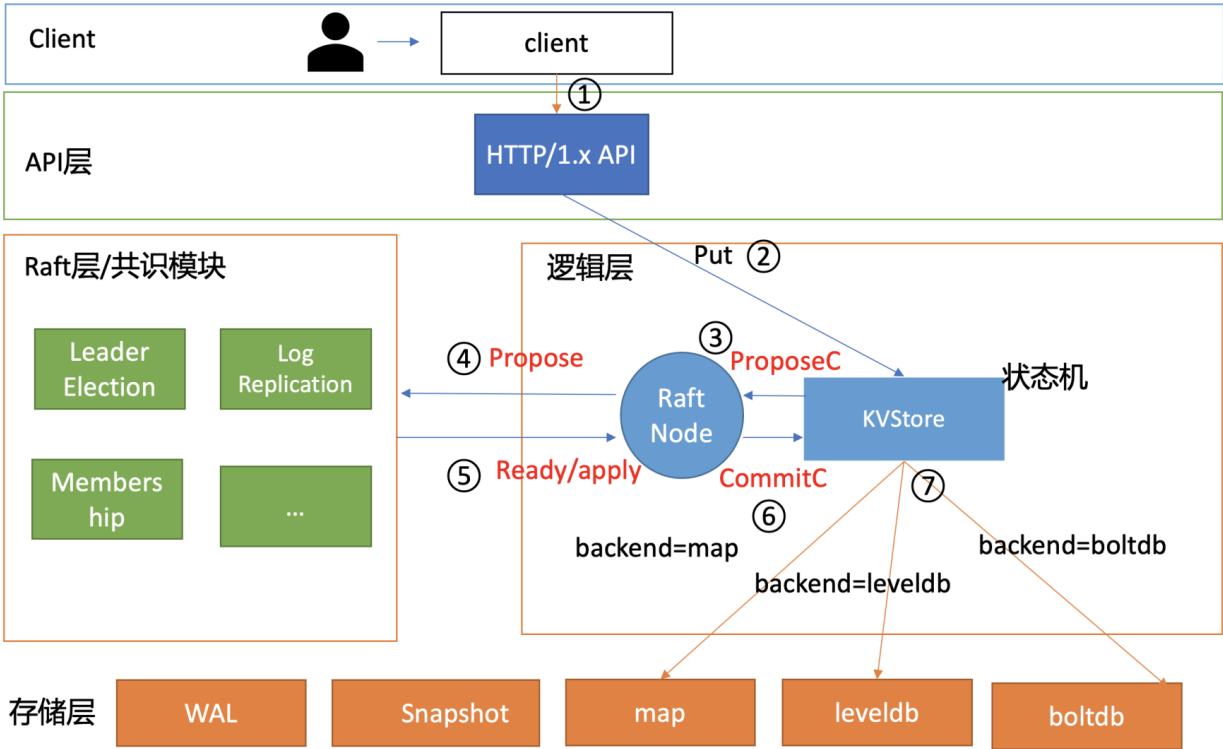
```
1 curl -L http://127.0.0.1:3379/hello -XPUT -d world
2
```

client 通过 curl 发送 HTTP PUT 请求到 server;

server 收到后，将消息写入到 KVStore 的 ProposeC 管道;

raftNode 循环逻辑将消息通过 Raft 模块的 Propose 接口提交;

Raft 模块输出 Ready 结构，server 将日志条目持久化后，并发送给其他节点；
集群多数节点持久化此日志条目后，这个日志条目被提交给存储状态机 KVStore 执行；
KVStore 根据启动的 backend 存储引擎名称，调用对应的 Put 接口即可。



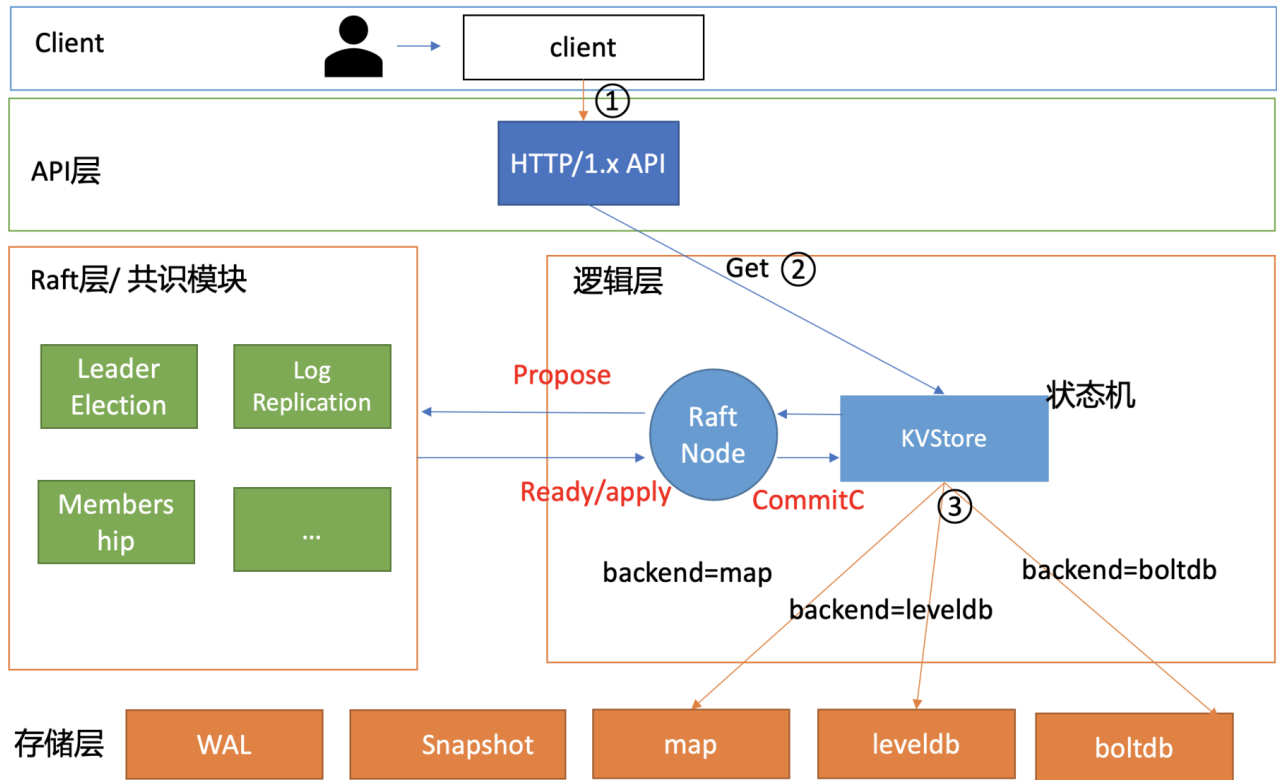
读流程

当你通过如下 curl 命令发起一个读操作时，读流程如下面架构图序号所示：

复制代码

```
1 curl -L http://127.0.0.1:3379/hello
2 world
```

client 通过 curl 发送 HTTP Get 请求到 server;
server 收到后，根据 KVStore 的存储引擎，从后端查询出对应的 key-value 数据。



小结

最后，我来总结下我们今天的内容。我这节课分别从整体架构设计和实现分析，给你介绍了如何基于 Raft 从 0 到 1 构建一个支持多存储引擎的分布式 key-value 数据库。

在整体架构设计上，我给你介绍了 API 设计核心因素，它们分别是性能、易用性、开发效率、安全性、幂等性。其次我和你介绍了复制状态机的原理，它由共识模块、日志模块、存储状态机模块组成。最后我和你深入分析了多存储引擎设计，重点介绍了 `leveldb` 原理，它将随机写转换为顺序写日志和内存，通过一系列分层、创新的设计实现了优异的写性能，适合读少写多。

在实现分析上，我和你重点介绍了 Raft 算法库的核心对象 Node API。对于一个库而言，我们重点关注的是其输入、输出接口，业务逻辑层可通过 `Propose` 接口提交提案，通过 `Ready` 结构获取 Raft 算法状态机的输出内容。其次我和你介绍了 Raft 算法库如何与 `WAL` 模块、Raft 日志存储模块、网络模块协作完成一个写请求。

最后为了支持多存储引擎，我们分别基于 `boltdb`、`leveldb` 实现了 `KVStore` 相关接口操作，并通过读写流程图，从整体上为你介绍了一个读写请求在 `metcd` 中是如何工作的。

麻雀虽小，五脏俱全。希望能通过这个迷你项目解答你对如何构建一个简易分布式 KV 服务的疑问，以及让你对 etcd 的工作原理有更深入的理解。

思考题

你知道 [raftexample](#) 启动的时候是如何工作的吗？它的存储引擎内存 map 是如何保证数据不丢失的呢？

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，我们下一讲见。

提建议

更多课程推荐

Redis 核心技术与实战

从原理到实战，彻底吃透 Redis

蒋德钧

中科院计算所副研究员



涨价倒计时🕒

现仅半价 **¥89** 4月17日涨价至 **¥199**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 性能及稳定性（下）：如何优化及扩展etcd性能？

下一篇 19 | Kubernetes基础应用：创建一个Pod背后etcd发生了什么？

精选留言 (1)

写留言



云原生工程师

2021-03-02

设计上存储引擎的介绍，获益匪浅，具体实现上的解读，也搞清了之前几个疑问，etcd足够轻量级，简直就是学习分布式系统的最佳案例，后面抽空自己基于raft搞个小小项目，进一步加深下，老师在这块有什么分布式书籍推荐没

展开 ∨

作者回复: 嗯，自己动手会有更深的理解，书籍推荐《Designing Data-Intensive Applications》，中文名《设计数据密集型应用》，豆瓣评分高达9.7，<https://book.douban.com/subject/30329536/>，非常不错的分布式入门书籍。

1

6