



下载APP



03 | 基础架构：etcd 一个写请求是如何执行的？

2021-01-25 唐聪

etcd 实战课

[进入课程 >](#)**讲述：王超凡**

时长 20:01 大小 18.35M



你好，我是唐聪。

在上一节课里，我通过分析 etcd 的一个读请求执行流程，给你介绍了 etcd 的基础架构，让你初步了解了在 etcd 的读请求流程中，各个模块是如何紧密协作，执行查询语句，返回数据给 client。

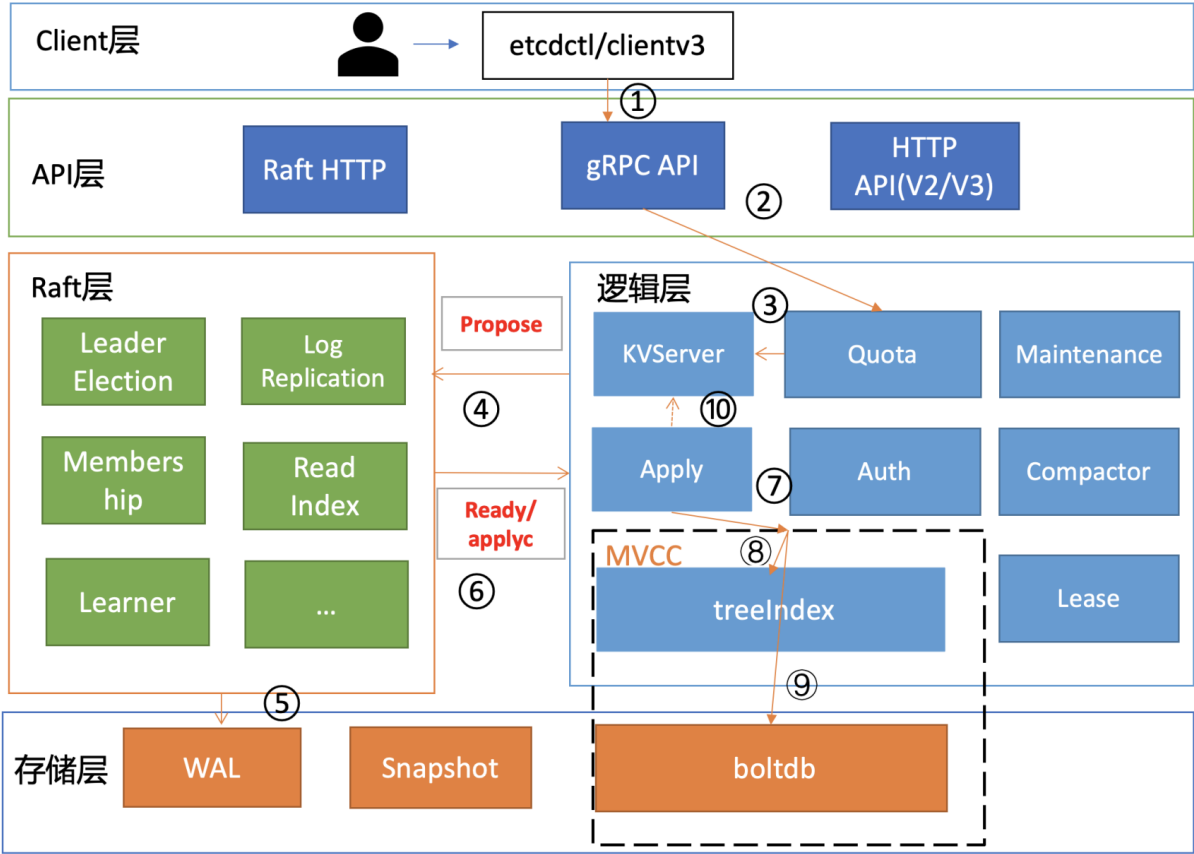
那么 etcd 一个写请求执行流程又是怎样的呢？在执行写请求过程中，如果进程 crash 了，如何保证数据不丢、命令不重复执行呢？



今天我就和你聊聊 etcd 写过程中是如何解决这些问题的。希望通过这节课，让你了解一个 key-value 写入的原理，对 etcd 的基础架构中涉及写请求相关的模块有一定的理解，同时

能触类旁通，当你在软件项目开发过程中遇到类似数据安全、幂等性等问题时，能设计出良好的方案解决它。

整体架构



为了让你能够更直观地理解 etcd 的写请求流程，我在如上的架构图中，用序号标识了下面的一个 put hello 为 world 的写请求的简要执行流程，帮助你从整体上快速了解一个写请求的全貌。

复制代码

```
1 etcdctl put hello world --endpoints http://127.0.0.1:2379
2 OK
3
```

首先 client 端通过负载均衡算法选择一个 etcd 节点，发起 gRPC 调用。然后 etcd 节点收到请求后经过 gRPC 拦截器、Quota 模块后，进入 KVServer 模块，KVServer 模块向 Raft 模块提交一个提案，提案内容为“大家好，请使用 put 方法执行一个 key 为 hello，value 为 world 的命令”。

随后此提案通过 RaftHTTP 网络模块转发、经过集群多数节点持久化后，状态会变成已提交，etcdserver 从 Raft 模块获取已提交的日志条目，传递给 Apply 模块，Apply 模块通过 MVCC 模块执行提案内容，更新状态机。

与读流程不一样的是写流程还涉及 Quota、WAL、Apply 三个模块。crash-safe 及幂等性也正是基于 WAL 和 Apply 流程的 consistent index 等实现的，因此今天我会重点和你介绍这三个模块。

下面就让我们沿着写请求执行流程图，从 0 到 1 分析一个 key-value 是如何安全、幂等地持久化到磁盘的。

Quota 模块

首先是流程一 client 端发起 gRPC 调用到 etcd 节点，和读请求不一样的是，写请求需要经过流程二 db 配额（Quota）模块，它有什么功能呢？

我们先从此模块的一个常见错误说起，你在使用 etcd 过程中是否遇到过"etcdserver: mvcc: database space exceeded"错误呢？

我相信只要你使用过 etcd 或者 Kubernetes，大概率见过这个错误。它是指当前 etcd db 文件大小超过了配额，当出现此错误后，你的整个集群将不可写入，只读，对业务的影响非常大。

哪些情况会触发这个错误呢？

一方面默认 db 配额仅为 2G，当你的业务数据、写入 QPS、Kubernetes 集群规模增大后，你的 etcd db 大小就可能会超过 2G。

另一方面我们知道 etcd v3 是个 MVCC 数据库，保存了 key 的历史版本，当你未配置压缩策略的时候，随着数据不断写入，db 大小会不断增大，导致超限。

最后你要特别注意的是，如果你使用的是 etcd 3.2.10 之前的旧版本，请注意备份可能会触发 boltdb 的一个 Bug，它会导致 db 大小不断上涨，最终达到配额限制。

了解完触发 Quota 限制的原因后，我们再详细了解下 Quota 模块它是如何工作的。

当 etcd server 收到 put/txn 等写请求的时候，会首先检查下当前 etcd db 大小加上你请求的 key-value 大小之和是否超过了配额 (quota-backend-bytes)。

如果超过了配额，它会产生一个告警 (Alarm) 请求，告警类型是 NO SPACE，并通过 Raft 日志同步给其它节点，告知 db 无空间了，并将告警持久化存储到 db 中。

最终，无论是 API 层 gRPC 模块还是负责将 Raft 侧已提交的日志条目应用到状态机的 Apply 模块，都拒绝写入，集群只读。

那遇到这个错误时应该如何解决呢？

首先当然是调大配额。具体多大合适呢？etcd 社区建议不超过 8G。遇到过这个错误的你是否还记得，为什么当你把配额 (quota-backend-bytes) 调大后，集群依然拒绝写入呢？

原因就是前面提到的 NO SPACE 告警。Apply 模块在执行每个命令的时候，都会去检查当前是否存在 NO SPACE 告警，如果有则拒绝写入。所以还需要你额外发送一个取消告警 (etcdctl alarm disarm) 的命令，以消除所有告警。

其次你需要检查 etcd 的压缩 (compact) 配置是否开启、配置是否合理。etcd 保存了一个 key 所有变更历史版本，如果没有一个机制去回收旧的版本，那么内存和 db 大小就会一直膨胀，在 etcd 里面，压缩模块负责回收旧版本的工作。

压缩模块支持按多种方式回收旧版本，比如保留最近一段时间内的历史版本。不过你要注意，它仅仅是将旧版本占用的空间打个空闲 (Free) 标记，后续新的数据写入的时候可复用这块空间，而无需申请新的空间。

如果你需要回收空间，减少 db 大小，得使用碎片整理 (defrag)，它会遍历旧的 db 文件数据，写入到一个新的 db 文件。但是它对服务性能有较大影响，不建议你在生产集群频繁使用。

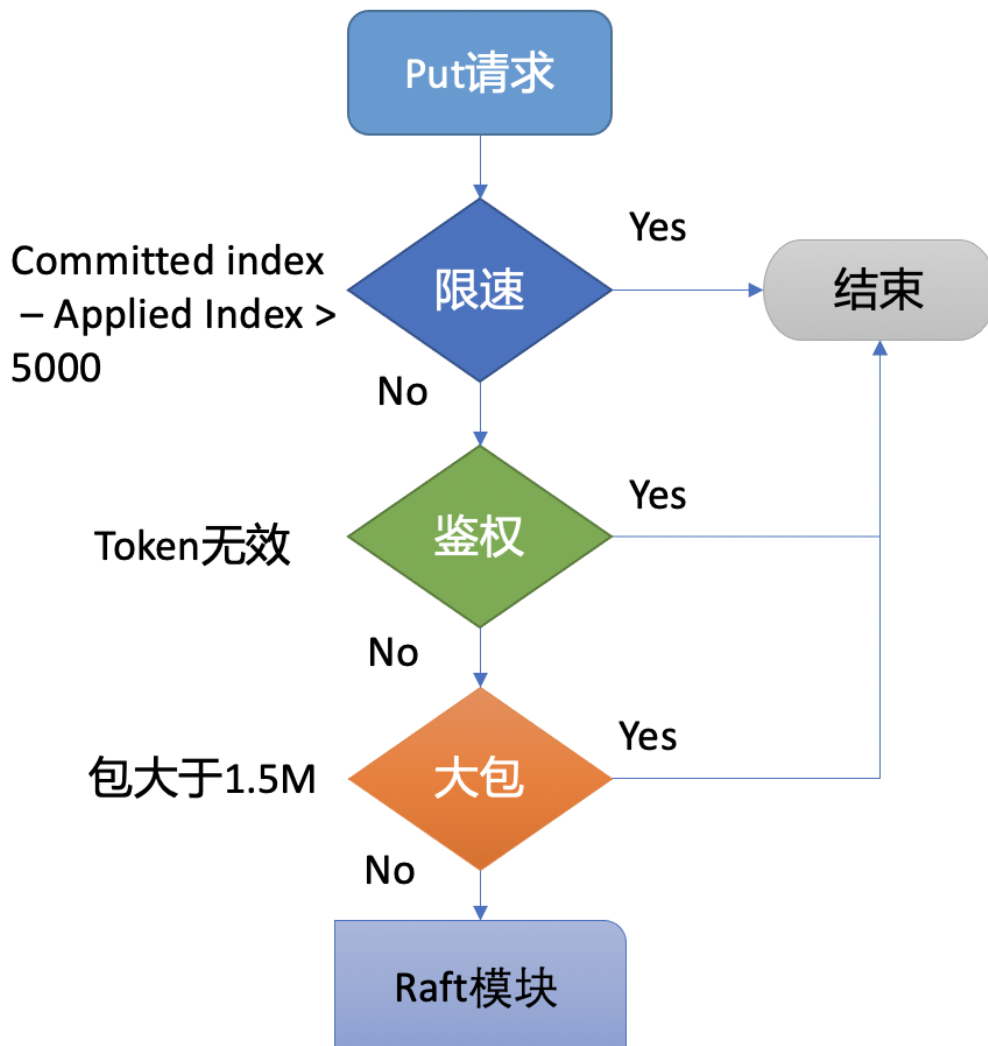
最后你需要注意配额 (quota-backend-bytes) 的行为，默认 '0' 就是使用 etcd 默认的 2GB 大小，你需要根据你的业务场景适当调优。如果你填的是个小于 0 的数，就会禁用配额功能，这可能会让你的 db 大小处于失控，导致性能下降，不建议你禁用配额。

KVServer 模块

通过流程二的配额检查后，请求就从 API 层转发到了流程三的 KVServer 模块的 put 方法，我们知道 etcd 是基于 Raft 算法实现节点间数据复制的，因此它需要将 put 写请求内容打包成一个提案消息，提交给 Raft 模块。不过 KVServer 模块在提交提案前，还有如下的一系列检查和限速。

Preflight Check

为了保证集群稳定性，避免雪崩，任何提交到 Raft 模块的请求，都会做一些简单的限速判断。如下面的流程图所示，首先，如果 Raft 模块已提交的日志索引（committed index）比已应用到状态机的日志索引（applied index）超过了 5000，那么它就返回一个"etcdserver: too many requests"错误给 client。



然后它会尝试去获取请求中的鉴权信息，若使用了密码鉴权、请求中携带了 token，如果 token 无效，则返回 "auth: invalid auth token" 错误给 client。

其次它会检查你写入的包大小是否超过默认的 1.5MB，如果超过了会返回 "etcdserver: request is too large" 错误给 client。

Propose

最后通过一系列检查之后，会生成一个唯一的 ID，将此请求关联到一个对应的消息通知 channel，然后向 Raft 模块发起 (Propose) 一个提案 (Proposal)，提案内容为 “大家好，请使用 put 方法执行一个 key 为 hello，value 为 world 的命令”，也就是整体架构图里的流程四。

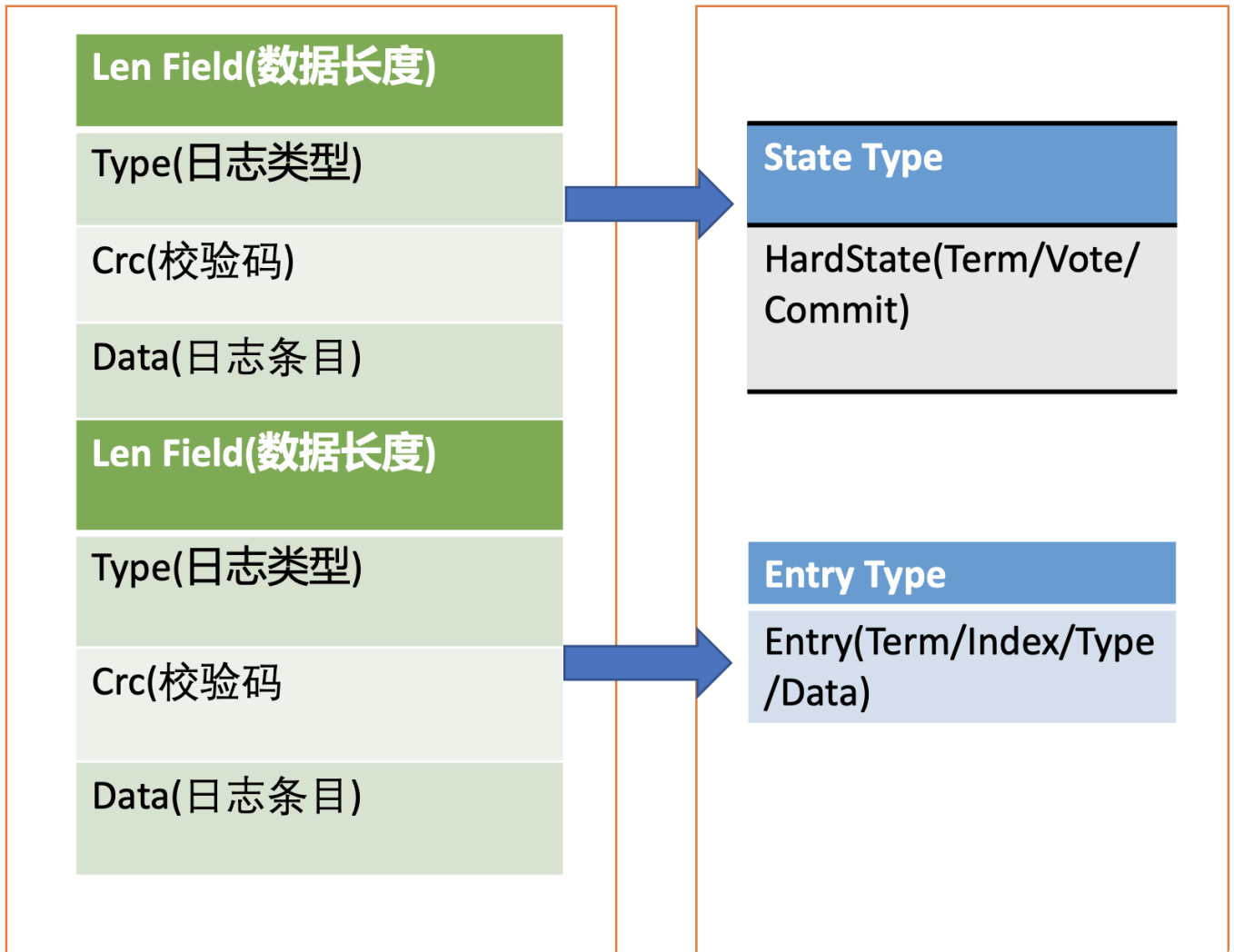
向 Raft 模块发起提案后，KVServer 模块会等待此 put 请求，等待写入结果通过消息通知 channel 返回或者超时。etcd 默认超时时间是 7 秒（5 秒磁盘 IO 延时 + 2*1 秒竞选超时时间），如果一个请求超时未返回结果，则可能会出现你熟悉的 etcdserver: request timed out 错误。

WAL 模块

Raft 模块收到提案后，如果当前节点是 Follower，它会转发给 Leader，只有 Leader 才能处理写请求。Leader 收到提案后，通过 Raft 模块输出待转发给 Follower 节点的消息和待持久化的日志条目，日志条目则封装了我们上面所说的 put hello 提案内容。

etcdserver 从 Raft 模块获取到以上消息和日志条目后，作为 Leader，它会将 put 提案消息广播给集群各个节点，同时需要把集群 Leader 任期号、投票信息、已提交索引、提案内容持久化到一个 WAL (Write Ahead Log) 日志文件中，用于保证集群的一致性、可恢复性，也就是我们图中的流程五模块。

WAL 日志结构是怎样的呢？



上图是 WAL 结构，它由多种类型的 WAL 记录顺序追加写入组成，每个记录由类型、数据、循环冗余校验码组成。不同类型的记录通过 Type 字段区分，Data 为对应记录内容，CRC 为循环校验码信息。

WAL 记录类型目前支持 5 种，分别是文件元数据记录、日志条目记录、状态信息记录、CRC 记录、快照记录：

文件元数据记录包含节点 ID、集群 ID 信息，它在 WAL 文件创建的时候写入；

日志条目记录包含 Raft 日志信息，如 put 提案内容；

状态信息记录，包含集群的任期号、节点投票信息等，一个日志文件中会有多条，以最后的记录为准；

CRC 记录包含上一个 WAL 文件的最后的 CRC（循环冗余校验码）信息，在创建、切割 WAL 文件时，作为第一条记录写入到新的 WAL 文件，用于校验数据文件的完整性、准确性等；

快照记录包含快照的任期号、日志索引信息，用于检查快照文件的准确性。

WAL 模块又是如何持久化一个 put 提案的日志条目类型记录呢？


首先我们来看看 put 写请求如何封装在 Raft 日志条目里面。下面是 Raft 日志条目的数据结构信息，它由以下字段组成：

Term 是 Leader 任期号，随着 Leader 选举增加；

Index 是日志条目的索引，单调递增增加；

Type 是日志类型，比如是普通的命令日志（EntryNormal）还是集群配置变更日志（EntryConfChange）；

Data 保存我们上面描述的 put 提案内容。

 复制代码

```
1 type Entry struct {  
2     Term          uint64    `protobuf:"varint, 2, opt, name=Term" json:"Term"  
3     Index          uint64    `protobuf:"varint, 3, opt, name=Index" json:"Index"  
4     Type           EntryType `protobuf:"varint, 1, opt, name=Type, enum=Raftpb  
5     Data           []byte    `protobuf:"bytes, 4, opt, name=Data" json:"Data",  
6 }
```

了解完 Raft 日志条目数据结构后，我们再看 WAL 模块如何持久化 Raft 日志条目。它首先先将 Raft 日志条目内容（含任期号、索引、提案内容）序列化后保存到 WAL 记录的 Data 字段，然后计算 Data 的 CRC 值，设置 Type 为 Entry Type，以上信息就组成了一个完整的 WAL 记录。

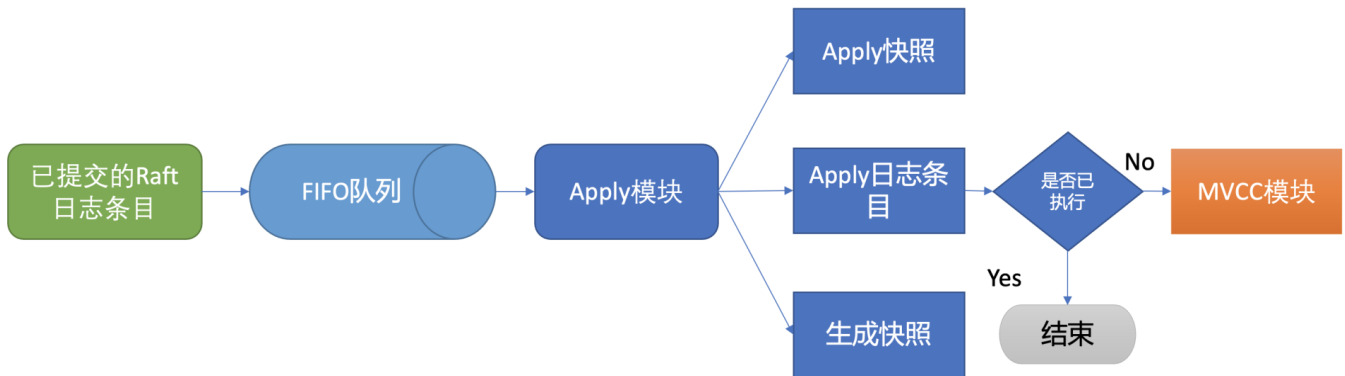
最后计算 WAL 记录的长度，顺序先写入 WAL 长度（Len Field），然后写入记录内容，调用 fsync 持久化到磁盘，完成将日志条目保存到持久化存储中。

当一半以上节点持久化此日志条目后，Raft 模块就会通过 channel 告知 etcdserver 模块，put 提案已经被集群多数节点确认，提案状态为已提交，你可以执行此提案内容了。

于是进入流程六，etcdserver 模块从 channel 取出提案内容，添加到先进先出（FIFO）调度队列，随后通过 Apply 模块按入队顺序，异步、依次执行提案内容。

Apply 模块

执行 put 提案内容对应我们架构图中的流程七，其细节图如下。那么 Apply 模块是如何执行 put 请求的呢？若 put 请求提案在执行流程七的时候 etcd 突然 crash 了，重启恢复的时候，etcd 是如何找回异常提案，再次执行的呢？



核心就是我们上面介绍的 WAL 日志，因为提交给 Apply 模块执行的提案已获得多数节点确认、持久化，etcd 重启时，会从 WAL 中解析出 Raft 日志条目内容，追加到 Raft 日志的存储中，并重放已提交的日志提案给 Apply 模块执行。

然而这又引发了另外一个问题，如何确保幂等性，防止提案重复执行导致数据混乱呢？

我们在上一节课里讲到，etcd 是个 MVCC 数据库，每次更新都会生成新的版本号。如果没有幂等性保护，同样的命令，一部分节点执行一次，一部分节点遭遇异常故障后执行多次，则系统的各节点一致性状态无法得到保证，导致数据混乱，这是严重故障。

因此 etcd 必须要确保幂等性。怎么做呢？Apply 模块从 Raft 模块获得的日志条目信息里，是否有唯一的字段能标识这个提案？

答案就是我们上面介绍 Raft 日志条目中的索引 (index) 字段。日志条目索引是全局单调递增的，每个日志条目索引对应一个提案，如果一个命令执行后，我们在 db 里面也记录下当前已经执行过的日志条目索引，是不是就可以解决幂等性问题呢？

是的。但是这还不够安全，如果执行命令的请求更新成功了，更新 index 的请求却失败了，是不是一样会导致异常？

因此我们在实现上，还需要将两个操作作为原子性事务提交，才能实现幂等。

正如我们上面的讨论的这样，etcd 通过引入一个 consistent index 的字段，来存储系统当前已经执行过的日志条目索引，实现幂等性。

Apply 模块在执行提案内容前，首先会判断当前提案是否已经执行过了，如果执行了则直接返回，若未执行同时无 db 配额满告警，则进入到 MVCC 模块，开始与持久化存储模块打交道。

MVCC

Apply 模块判断此提案未执行后，就会调用 MVCC 模块来执行提案内容。MVCC 主要由两部分组成，一个是内存索引模块 treeIndex，保存 key 的历史版本号信息，另一个是 boltdb 模块，用来持久化存储 key-value 数据。那么 MVCC 模块执行 put hello 为 world 命令时，它是如何构建内存索引和保存哪些数据到 db 呢？

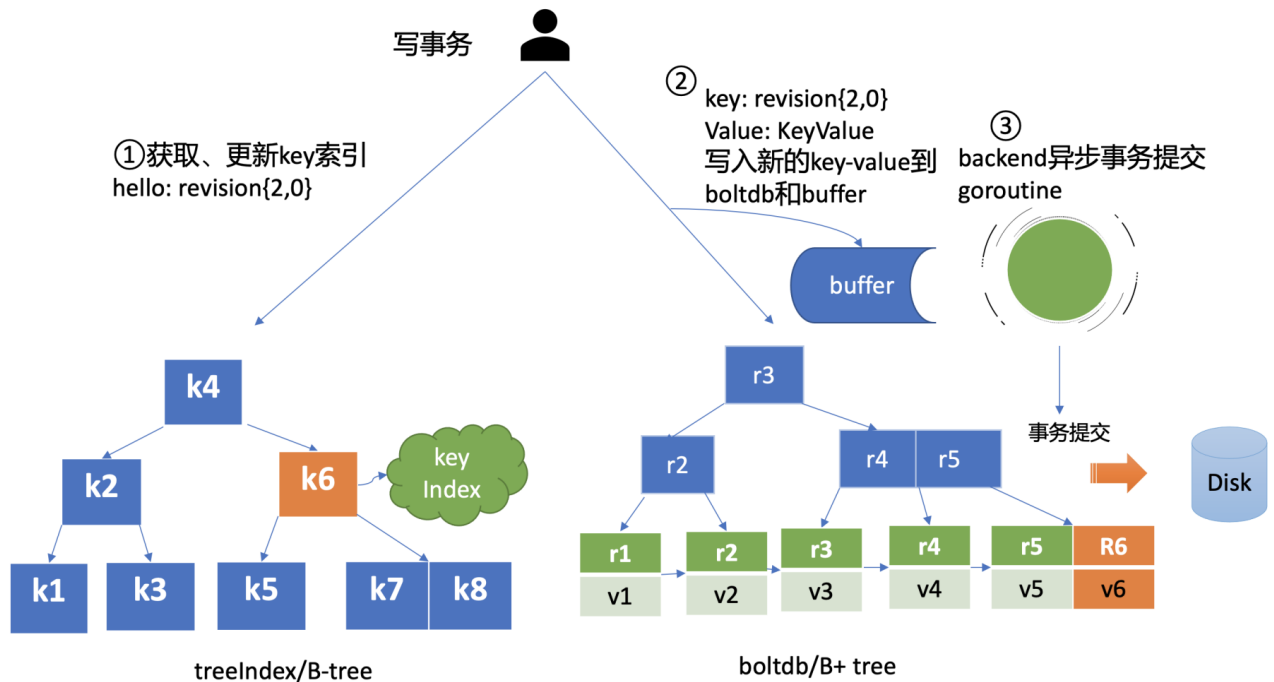
treeIndex

首先我们来看 MVCC 的索引模块 treeIndex，当收到更新 key hello 为 world 的时候，此 key 的索引版本号信息是怎么生成的呢？需要维护、持久化存储一个全局版本号吗？

版本号 (revision) 在 etcd 里面发挥着重大作用，它是 etcd 的逻辑时钟。etcd 启动的时候默认版本号是 1，随着你对 key 的增、删、改操作而全局单调递增。

因为 boltdb 中的 key 就包含此信息，所以 etcd 并不需要再去持久化一个全局版本号。我们只需要在启动的时候，从最小值 1 开始枚举到最大值，未读到数据的时候则结束，最后读出来的版本号即是当前 etcd 的最大版本号 currentRevision。

MVCC 写事务在执行 put hello 为 world 的请求时，会基于 currentRevision 自增生成新的 revision 如{2,0}，然后从 treeIndex 模块中查询 key 的创建版本号、修改次数信息。这些信息将填充到 boltdb 的 value 中，同时将用户的 hello key 和 revision 等信息存储到 B-tree，也就是下面简易写事务图的流程一，整体架构图中的流程八。



boltdb

MVCC 写事务自增全局版本号后生成的 revision{2,0}，它就是 boltdb 的 key，通过它就可以往 boltdb 写数据了，进入了整体架构图中的流程九。

boltdb 上一篇我们提过它是一个基于 B+tree 实现的 key-value 嵌入式 db，它通过提供桶 (bucket) 机制实现类似 MySQL 表的逻辑隔离。

在 etcd 里面你通过 put/txn 等 KV API 操作的数据，全部保存在一个名为 key 的桶里面，这个 key 桶在启动 etcd 的时候会自动创建。

除了保存用户 KV 数据的 key 桶，etcd 本身及其它功能需要持久化存储的话，都会创建对应的桶。比如上面我们提到的 etcd 为了保证日志的幂等性，保存了一个名为 consistent index 的变量在 db 里面，它实际上就存储在元数据 (meta) 桶里面。

那么写入 boltdb 的 value 含有哪些信息呢？

写入 boltdb 的 value，并不是简单的 "world"，如果只存一个用户 value，索引又是保存在易失的内存上，那重启 etcd 后，我们就丢失了用户的 key 名，无法构建 treeIndex 模块了。

因此为了构建索引和支持 Lease 等特性，etcd 会持久化以下信息：

key 名称；

key 创建时的版本号 (create_revision) 、最后一次修改时的版本号 (mod_revision) 、key 自身修改的次数 (version) ；

value 值；

租约信息 (后面介绍) 。

boltdb value 的值就是将含以上信息的结构体序列化成的二进制数据，然后通过 boltdb 提供的 put 接口，etcd 就快速完成了将你的数据写入 boltdb，对应上面简易写事务图的流程二。

但是 put 调用成功，就能够代表数据已经持久化到 db 文件了吗？

这里需要注意的是，在以上流程中，etcd 并未提交事务 (commit) ，因此数据只更新在 boltdb 所管理的内存数据结构中。

事务提交的过程，包含 B+tree 的平衡、分裂，将 boltdb 的脏数据 (dirty page) 、元数据信息刷新到磁盘，因此事务提交的开销是昂贵的。如果我们每次更新都提交事务，etcd 写性能就会较差。

那么解决的办法是什么呢？etcd 的解决方案是合并再合并。

首先 boltdb key 是版本号，put/delete 操作时，都会基于当前版本号递增生成新的版本号，因此属于顺序写入，可以调整 boltdb 的 bucket.FillPercent 参数，使每个 page 填充更多数据，减少 page 的分裂次数并降低 db 空间。

其次 etcd 通过合并多个写事务请求，通常情况下，是异步机制定时 (默认每隔 100ms) 将批量事务一次性提交 (pending 事务过多才会触发同步提交) ，从而大大提高吞吐量，对应上面简易写事务图的流程三。

但是这优化又引发了另外的一个问题，因为事务未提交，读请求可能无法从 boltdb 获取到最新数据。

为了解决这个问题，etcd 引入了一个 bucket buffer 来保存暂未提交的事务数据。在更新 boltdb 的时候，etcd 也会同步数据到 bucket buffer。因此 etcd 处理读请求的时候会优先从 bucket buffer 里面读取，其次再从 boltdb 读，通过 bucket buffer 实现读写性能提升，同时保证数据一致性。

小结

最后我们来小结一下，今天我给你介绍了 etcd 的写请求流程，重点介绍了 Quota、WAL、Apply 模块。

首先我们介绍了 Quota 模块工作原理和我们熟悉的 database space exceeded 错误触发原因，写请求导致 db 大小增加、compact 策略不合理、boltdb Bug 等都会导致 db 大小超限。

其次介绍了 WAL 模块的存储结构，它由一条条记录顺序写入组成，每个记录含有 Type、CRC、Data，每个提案被提交前都会被持久化到 WAL 文件中，以保证集群的一致性和可恢复性。

随后我们介绍了 Apply 模块基于 consistent index 和事务实现了幂等性，保证了节点在异常情况下不会重复执行重放的提案。

最后我们介绍了 MVCC 模块是如何维护索引版本号、重启后如何从 boltdb 模块中获取内存索引结构的。以及 etcd 通过异步、批量提交事务机制，以提升写 QPS 和吞吐量。

通过以上介绍，希望你对 etcd 的一个写语句执行流程有个初步的理解，明白 WAL 模块、Apply 模块、MVCC 模块三者是如何相互协作的，从而实现在节点遭遇 crash 等异常情况下，不丢任何已提交的数据、不重复执行任何提案。

思考题

expensive read 请求（如 Kubernetes 场景中查询大量 pod）会影响写请求的性能吗？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾给出我的答案。

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，同时欢迎你把这节课分享给你的朋友或者同事，一起交流一下。

02 思考题答案

上节课我给大家留了一个思考题，评论中有同学说 buffer 没读到，从 boltdb 读时会产生磁盘 I/O，这是一个常见误区。

实际上，etcd 在启动的时候会通过 mmap 机制将 etcd db 文件映射到 etcd 进程地址空间，并设置了 mmap 的 MAP_POPULATE flag，它会告诉 Linux 内核预读文件，Linux 内核会将文件内容拷贝到物理内存中，此时会产生磁盘 I/O。节点内存足够的请求下，后续处理读请求过程中就不会产生磁盘 I/O 了。

若 etcd 节点内存不足，可能会导致 db 文件对应的内存页被换出，当读请求命中的页未在内存中时，就会产生缺页异常，导致读过程中产生磁盘 IO，你可以通过观察 etcd 进程的 majflt 字段来判断 etcd 是否产生了主缺页中断。

提建议

© 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 02 | 基础架构：etcd一个读请求是如何执行的？

下一篇 04 | Raft协议：etcd如何实现高可用、数据强一致的？

精选留言 (13)

写留言



唐聪

2021-01-26

之前修复的存在3年多的不一致bug就是跟本节介绍的写请求幂等性有关，在每讲中，我提及的bug，特性缺点，大部分都是我之前踩过的坑，这些经验希望能帮助大家提前避免不

必要的线上问题



10

**云原生工程师**

2021-01-26

文章有点长，但读起来层层递进，有很大收获，发现之前自己所了解的还真不全面，填补了之前的一些空白面，期待后面的精彩内容

展开 ∨



3

**七里**

2021-01-26

幂等部分的“原子性事务”如何实现的？

展开 ∨

作者回复: 就是key-value数据与consistent index在同一个boltdb事务中更新，boltdb后面会再单独介绍



2

**TS.乔**

2021-01-26

希望在后面多加一下具体设计思路，以及特性取舍的东西

作者回复: 嗯，谢谢你的建议，篇幅本身比较长就没继续扩展了，比如读写原理中，treeindex为什么用b-tree而不是其他数据结构，为什么使用boltdb而不是基于lsmtree的leveldb等，后面答疑和其他讲我将适当和大家一起讨论



1



2

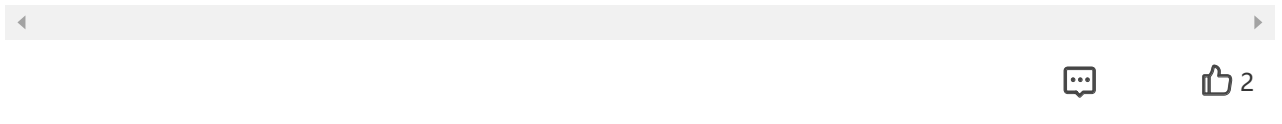
**Index**

2021-01-26

之前研究一段时间的etcd的源码，看的七七八八，现在再看这篇文章把之前的很多疑问都解答了，太棒了，etcd是个很优秀的项目，能把这么多的技术点融合在一起，实在是一个很好的开源学习项目。老师有空可以开直播，多聊聊etcd中涉及的技术点的一些学习，从源头上把知识融汇贯通，这样的学习真是酣畅淋漓

展开 ∨

作者回复: 好的，谢谢你的认可，一起学习加油

**一步**

2021-01-26

这里 db 配额 (Quota) 的限制哪个地方的大小的? 为什么会有这样的限制呢? 这个限制是 boltdb 全部数据 持久化后的文件大小吗?

展开 ∨

作者回复: boltdb的db文件, 你可以在etcd的数据目录snap目录下看到有个名为db的文件, 实践篇我会介绍db文件过大又哪些问题, etcd定位就是个小型的关键元数据存储

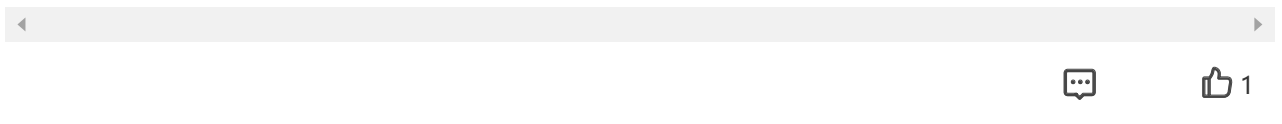
不瘦二十斤
不改头像**jeffery**

2021-01-26

原理讲的透彻、为啥applied index超过了 5000, 返回一个"etcdserver: too many requests"错误给 client。raft源码定义的最大值吗? 谢谢老师

展开 ∨

作者回复: 谢谢, 这个限速不是raft模块做的, raft是个单独共识算法库, 是etcd server使用raft的时候, 基于raft告知的committed index, 本身apply模块的applied index做的限速, 默认写死了5000

**范闲**

2021-01-25

会有影响。

- 1.如果读写请求都落到bucket buffer上, bucketbuffer需要做锁处理。
- 2.如果读写请求都落到boltdb上, db上的数据是从磁盘加载, 同bucket buffer相比性能会下降一个数量级。
- 3.如果既落了bucket buffer, 又落到了boltdb上。那么性能受到的影响介于1-2之间。

展开 ∨

**领程**

2021-01-28

- 1) 首先 boltdb key 是版本号, put/delete 操作时, 都会基于当前版本号递增生成新的

版本号, 因此属于顺序写入, 可以调整 boltdb 的 bucket.FillPercent 参数, 使每个 page 填充更多数据, 减少 page 的分裂次数并降低db空间。此处的page 和 降低db空间不是很理解, 劳烦老师解惑!

...

展开 ∨

**领程**

2021-01-28

etcd 默认超时时间是 7 秒 (5 秒磁盘 IO 延时 + 2*1 秒竞选超时时间)

老师, “竞选超时时间” 具体指的是什么? 还有磁盘IO延时要怎么理解比较贴切?

展开 ∨

**一步**

2021-01-27

boltdb 会保存 key 的所有修改版本信息吗?

展开 ∨

作者回复: 嗯, 压缩会删除旧版本, 07 mvcc会详细介绍

**金石**

2021-01-26

请问revision如果超出了上限, revision会如何接着生成?

作者回复: 好问题, 目前etcd没处理这种情况, 它的类似是int64,最大值9223372036854775807, 看起来是很难达到上限的

**mckee**

2021-01-25

compact后没有defrag的话对性能有影响吗

展开 ∨

作者回复: compact是非常常规的操作，只要db文件不是特别大，大小保持稳定就不用defrag
哈，后面会详细介绍

