



下载APP



## 13 | db大小：为什么etcd社区建议db大小不超过8G？

2021-02-17 唐聪

etcd实战课

[进入课程 >](#)**讲述：王超凡**

时长 13:53 大小 12.73M



你好，我是唐聪。

在 [03](#) 写流程中我和你分享了 etcd Quota 模块，那么 etcd 为什么需要对 db 增加 Quota 限制，以及不建议你的 etcd 集群 db 大小超过 8G 呢？过大的 db 文件对集群性能和稳定性有哪些影响？

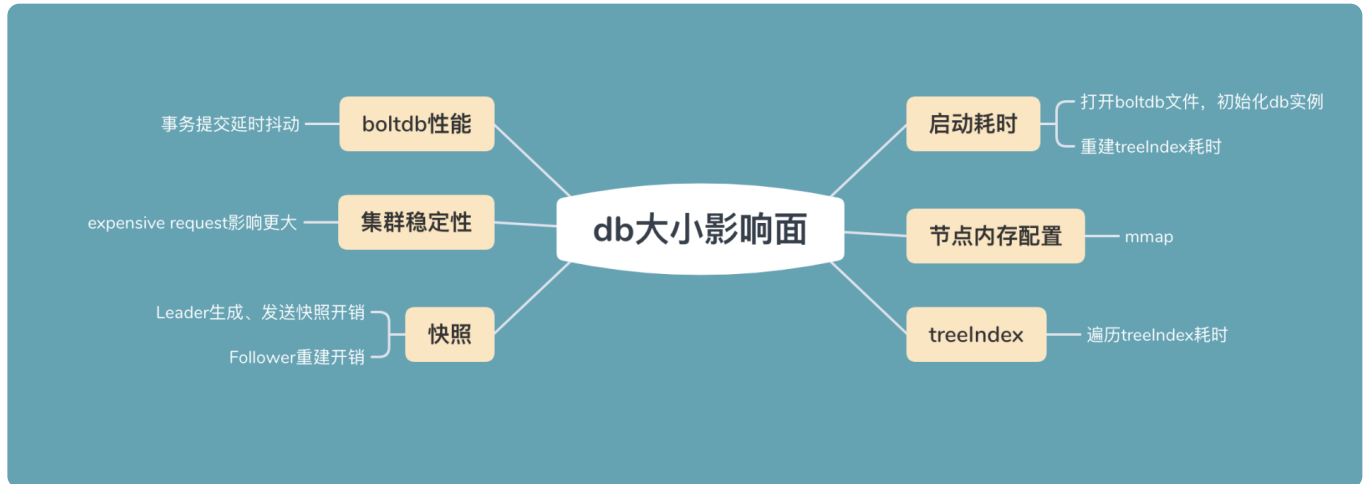
今天我要和你分享的主题就是关于 db 大小。我将通过一个大数据量的 etcd 集群为案例，为你剖析 etcd db 大小配额限制背后的设计思考和过大的 db 潜在隐患。



希望通过这节课，帮助你理解大数据量对集群的各个模块的影响，配置合理的 db Quota 值。同时，帮助你在实际业务场景中，遵循最佳实践，尽量减少 value 大小和大 key-value 更新频率，避免 db 文件大小不断增长。

## 分析整体思路

为了帮助你直观地理解大数据量对集群稳定性的影响，我首先将为你写入大量数据，构造一个 db 大小为 14G 的大集群。然后通过此集群为你分析 db 大小的各个影响面，db 大小影响面如下图所示。



首先是**启动耗时**。etcd 启动的时候，需打开 boltDb db 文件，读取 db 文件所有 key-value 数据，用于重建内存 treeIndex 模块。因此在大量 key 导致 db 文件过大的场景中，这会导致 etcd 启动较慢。

其次是**节点内存配置**。etcd 在启动的时候会通过 mmap 将 db 文件映射内存中，若节点可用内存不足，小于 db 文件大小时，可能会出现缺页文件中断，导致服务稳定性、性能下降。

接着是 **treeIndex** 索引性能。因 etcd 不支持数据分片，内存中的 treeIndex 若保存了几十万到上千万的 key，这会增加查询、修改操作的整体延时。

然后是 **boltDb 性能**。大 db 文件场景会导致事务提交耗时增长、抖动。

再次是**集群稳定性**。大 db 文件场景下，无论你是百万级别小 key 还是上千个大 value 场景，一旦出现 expensive request 后，很容易导致 etcd OOM、节点带宽满而丢包。

最后是**快照**。当 Follower 节点落后 Leader 较多数据的时候，会触发 Leader 生成快照重建发送给 Follower 节点，Follower 基于它进行还原重建操作。较大的 db 文件会导致 Leader 发送快照需要消耗较多的 CPU、网络带宽资源，同时 Follower 节点重建还原慢。

## 构造大集群

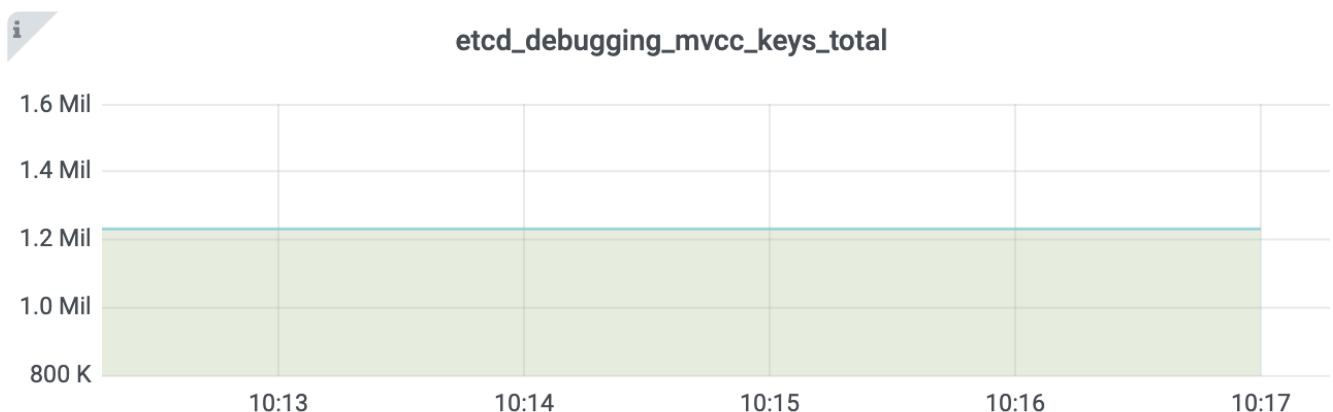
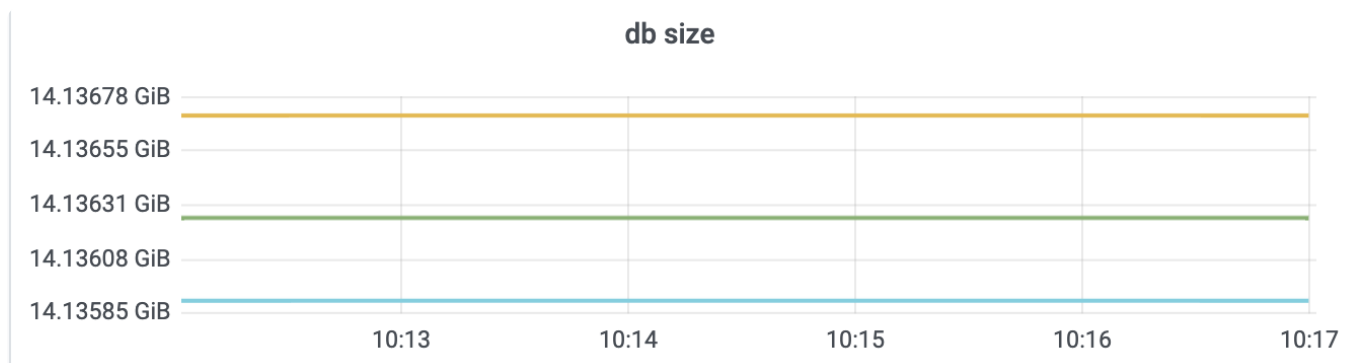
简单介绍完 db 大小的六个影响面后，我们下面来构造一个大数据量的集群，用于后续各个影响面的分析。

首先，我通过一系列如下 [benchmark](#) 命令，向一个 8 核 32G 的 3 节点的集群写入 120 万左右 key。key 大小为 32，value 大小为 256 到 10K，用以分析大 db 集群案例中的各个影响面。

[复制代码](#)

```
1 ./benchmark put --key-size 32 --val-size 10240 --total  
2 1000000 --key-space-size 2000000 --clients 50 --conns 50
```

执行完一系列 benchmark 命令后，db size 达到 14G，总 key 数达到 120 万，其监控如下图所示：



## 启动耗时

在如上的集群中，我通过 benchmark 工具将 etcd 集群 db 大小压测到 14G 后，在重新启动 etcd 进程的时候，如下日志所示，你会发现启动比较慢，为什么大 db 文件会影响

## etcd 启动耗时呢？

[复制代码](#)

```
1 2021-02-15 02:25:55.273712 I | etcdmain: etcd Version: 3.4.9
2 2021-02-15 02:26:58.806882 I | etcdserver: recovered store from snapshot at in
3 2021-02-15 02:26:58.808810 I | mvcc: restore compact to 1000002
4 2021-02-15 02:27:19.120141 W | etcdserver: backend quota 26442450944 exceeds m
5 2021-02-15 02:27:19.297363 I | embed: ready to serve client requests
```

通过对 etcd 启动流程增加耗时统计，我们可以发现核心瓶颈主要在于打开 db 文件和重建内存 treeIndex 模块。

这里我重点先和你介绍下 etcd 启动后，重建内存 treeIndex 的原理。

我们知道 treeIndex 模块维护了用户 key 与 boltdb key 的映射关系，boltdb 的 key、value 又包含了构建 treeIndex 的所需的数据。因此 etcd 启动的时候，会启动不同角色的 goroutine 并发完成 treeIndex 构建。

**首先是主 goroutine。**它的职责是遍历 boltdb，获取所有 key-value 数据，并将其反序列化成 etcd 的 mvccpb.KeyValue 结构。核心原理是基于 etcd 存储在 boltdb 中的 key 数据有序性，按版本号从 1 开始批量遍历，每次查询 10000 条 key-value 记录，直到查询数据为空。

**其次是构建 treeIndex 索引的 goroutine。**它从主 goroutine 获取 mvccpb.KeyValue 数据，基于 key、版本号、是否带删除标识等信息，构建 keyIndex 对象，插入到 treeIndex 模块的 B-tree 中。

因可能存在多个 goroutine 并发操作 treeIndex，treeIndex 的 Insert 函数会加全局锁，如下所示。etcd 启动时只有一个**构建 treeIndex 索引的 goroutine**，因此 key 多时，会比较慢。之前我尝试优化成多 goroutine 并发构建，但是效果不佳，大量耗时会消耗在此锁上。

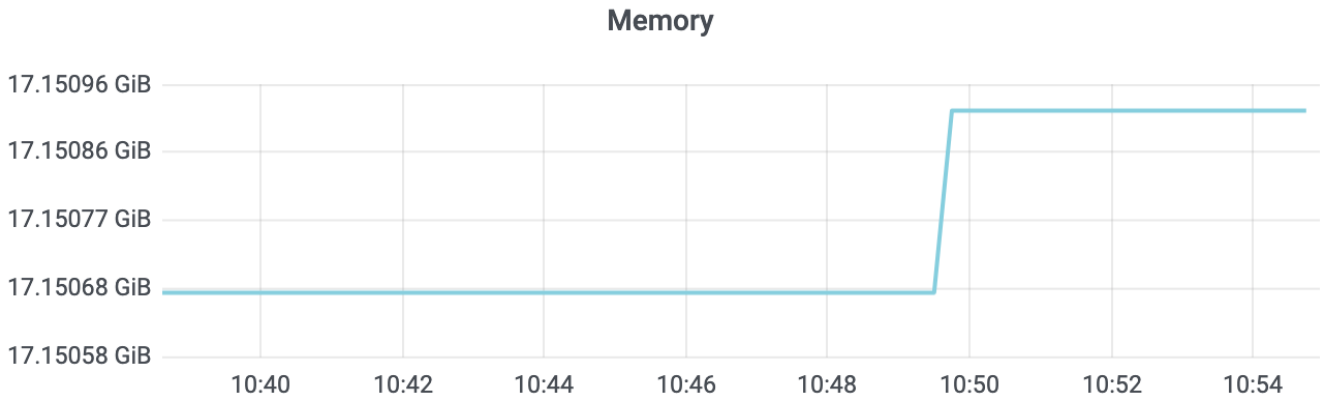
[复制代码](#)

```
1 func (ti *treeIndex) Insert(ki *keyIndex) {
2     ti.Lock()
3     defer ti.Unlock()
```

```
4    ti.tree.ReplaceOrInsert(ki)
5 }
```

## 节点内存配置

etcd 进程重启完成后，在没有任何读写 QPS 情况下，如下所示，你会发现 etcd 所消耗的内存比 db 大小还大一点。这又是为什么呢？如果 etcd db 文件大小超过节点内存规格，会导致什么问题吗？



在 [🔗 10](#) 介绍 boltdb 存储原理的时候，我和你分享过 boltdb 文件的磁盘布局结构和其对外提供的 API 原理。

etcd 在启动的时候，会通过 boltdb 的 Open API 获取数据库对象，而 Open API 它会通过 mmap 机制将 db 文件映射到内存中。

由于 etcd 调用 boltdb Open API 的时候，设置了 mmap 的 MAP\_POPULATE flag，它会告诉 Linux 内核预读文件，将 db 文件内容全部从磁盘加载到物理内存中。

因此在你节点内存充足的情况下，启动后你看到的 etcd 占用内存，一般是 db 文件大小与内存 treeIndex 之和。

在节点内存充足的情况下，启动后，client 后续发起对 etcd 的读操作，可直接通过内存获取 boltdb 的 key-value 数据，不会产生任何磁盘 IO，具备良好的读性能、稳定性。

而当你 db 文件大小超过节点内存配置时，若你查询的 key 所相关的 branch page、leaf page 不在内存中，那就会触发主缺页中断，导致读延时抖动、QPS 下降。


因此为了保证 etcd 集群性能的稳定性，我建议你的 etcd 节点内存规格要大于你的 etcd db 文件大小。

## treeIndex

当我们往集群中写入了一百多万 key 时，此时你再读取一个 key 范围操作的延时会出现一定程度上升，这是为什么呢？我们该如何分析耗时是在哪一步导致的？

在 etcd 3.4 中提供了 trace 特性，它可帮助我们定位、分析请求耗时过长问题。不过你需要特别注意的是，此特性在 etcd 3.4 中，因为依赖 zap logger，默认为关闭。你可以通过设置 etcd 启动参数中的 `--logger=zap` 来开启。

开启之后，我们可以在 etcd 日志中找到类似如下的耗时记录。

 复制代码

```
1 {  
2   "msg": "trace[331581563] range",  
3   "detail": "{range_begin:/vip/a; range_end:/vip/b; response_count:19304; respons",  
4   "duration": "146.432768ms",  
5   "steps": [  
6     "trace[331581563] 'range keys from in-memory treeIndex' (duration: 95.925033m",  
7     "trace[331581563] 'range keys from bolt db' (duration: 47.932118ms)"  
8   ]
```

此日志记录了查询请求 `etcdctl get --prefix /vip/a`。它在 `treeIndex` 中查询相关 key 耗时 95ms，从 `boltdb` 遍历 key 时 47ms。主要原因还是此查询涉及的 key 数较多，高达一万九。

也就是说若 `treeIndex` 中存储了百万级的 key 时，它可能也会产生几十毫秒到数百毫秒的延时，对于期望业务延时稳定在较小阈值内的业务，就无法满足其诉求。

## boltdb 性能

当 db 文件大小持续增长到 16G 乃至更大后，从 etcd 事务提交监控 metrics 你可能会观察到，`boltdb` 在提交事务时偶尔出现了较高延时，那么延时是怎么产生的呢？



在🔗10介绍 boltdb 的原理时，我和你分享了 db 文件的磁盘布局，它是由 meta page、branch page、leaf page、free list、free 页组成的。同时我给你介绍了 boltdb 事务提交的四个核心流程，分别是 B+ tree 的重平衡、分裂，持久化 dirty page，持久化 freelist 以及持久化 meta data。

事务提交延时抖动的原因主要是在 B+ tree 树的重平衡和分裂过程中，它需要从 freelist 中申请若干连续的 page 存储数据，或释放空闲的 page 到 freelist。


freelist 后端实现在 boltdb 中是 array。当申请一个连续的 n 个 page 存储数据时，它会遍历 boltdb 中所有的空闲页，直到找到连续的 n 个 page。因此它的时间复杂度是  $O(N)$ 。若 db 文件较大，又存在大量的碎片空闲页，很可能导致超时。

同时事务提交过程中，也可能会释放若干个 page 给 freelist，因此需要合并到 freelist 的数组中，此操作时间复杂度是  $O(N \log N)$ 。

假设我们 db 大小 16G，page size 4KB，则有 400 万个 page。经过各种修改、压缩后，若存在一半零散分布的碎片空闲页，在最坏的场景下，etcd 每次事务提交需要遍历 200 万个 page 才能找到连续的 n 个 page，同时还需要持久化 freelist 到磁盘。

为了优化 boltdb 事务提交的性能，etcd 社区在 bbolt 项目中，实现了基于 hashmap 来管理 freelist。通过引入了如下的三个 map 数据结构（freemaps 的 key 是连续的页数，value 是以空闲页的起始页 pgid 集合，forwardmap 和 backmap 用于释放的时候快速合并页），将申请和释放时间复杂度降低到了  $O(1)$ 。

freelist 后端实现可以通过 bbolt 的 FreeListType 参数来控制，支持 array 和 hashmap。在 etcd 3.4 版本中目前还是 array，未来的 3.5 版本将默认是 hashmap。

 复制代码

```
1 freemaps      map[uint64]pidSet      // key is the size of continuous pa
2 forwardMap    map[pgid]uint64      // key is start pgid, value is its
3 backwardMap   map[pgid]uint64      // key is end pgid, value is its sp
```

另外在 db 中若存在大量空闲页，持久化 freelist 需要消耗较多的 db 大小，并会导致额外的事务提交延时。

若未持久化 freelist, bbolt 支持通过重启时扫描全部 page 来构造 freelist, 降低了 db 大小和提升写事务提交的性能（但是它会带来 etcd 启动延时的上升）。此行为可以通过 bbolt 的 NoFreelistSync 参数来控制，默认是 true 启用此特性。

## 集群稳定性

db 文件增大后，另外一个非常大的隐患是用户 client 发起的 expensive request，容易导致集群出现各种稳定性问题。

本质原因是 etcd 不支持数据分片，各个节点保存了所有 key-value 数据，同时它们又存储在 boltdb 的一个 bucket 里面。当你的集群含有百万级以上 key 的时候，任意一种 expensive read 请求都可能导致 etcd 出现 OOM、丢包等情况发生。

那么有哪些 expensive read 请求会导致 etcd 不稳定性呢？

**首先是简单的 count only 查询。**如下图所示，当你想通过 API 统计一个集群有多少 key 时，如果你的 key 较多，则有可能导致内存突增和较大的延时。



在 etcd 3.5 版本之前，统计 key 数会遍历 treeIndex，把 key 追加到数组中。然而当数据规模较大时，追加 key 到数组中的操作会消耗大量内存，同时数组扩容时涉及到大量数据拷贝，会导致延时上升。

**其次是 limit 查询。**当你只想查询若干条数据的时候，若你的 key 较多，也会导致类似 count only 查询的性能、稳定性问题。

原因是 etcd 3.5 版本之前遍历 index B-tree 时，并未将 limit 参数下推到索引层，导致了无用的资源和时间消耗。优化方案也很简单，etcd 3.5 中我提的优化 PR 将 limit 参数下推




到了索引层，实现查询性能百倍提升。

**最后是大包查询。**当你未分页批量遍历 key-value 数据或单 key-value 数据较大的时候，随着请求 QPS 增大，etcd OOM、节点出现带宽瓶颈导致丢包的风险会越来越大。

问题主要由以下两点原因导致：

第一，etcd 需要遍历 treeIndex 获取 key 列表。若你未分页，一次查询万级 key，显然会消耗大量内存并且高延时。

第二，获取到 key 列表、版本号后，etcd 需要遍历 boltdb，将 key-value 保存到查询结果数据结构中。如下 trace 日志所示，一个请求可能在遍历 boltdb 时花费很长时间，同时可能会消耗几百 M 甚至数 G 的内存。随着请求 QPS 增大，极易出现 OOM、丢包等。etcd 这块未来的优化点是实现流式传输。

 复制代码

```
1 {
2   "level":"info",
3   "ts":"2021-02-15T03:44:52.209Z",
4   "caller":"traceutil/trace.go:145",
5   "msg":"trace[1908866301] range",
6   "detail":{"range_begin;; range_end;; response_count:1232274; response_revision
7   "duration":"9.063748801s",
8   "start":"2021-02-15T03:44:43.145Z",
9   "end":"2021-02-15T03:44:52.209Z",
10  "steps":[
11    "trace[1908866301] 'range keys from in-memory index tree' (duration: 693.26256
12    "trace[1908866301] 'range keys from bolt db' (duration: 8.22558566s)",
13    "trace[1908866301] 'assemble the response' (duration: 18.810315ms)"
14  ]
15 }
```

## 快照

大 db 文件最后一个影响面是快照。它会影响 db 备份文件生成速度、Leader 发送快照给 Follower 节点的资源开销、Follower 节点通过快照重建恢复的速度。

我们知道 etcd 提供了快照功能，帮助我们通过 API 即可备份 etcd 数据。当 etcd 收到 snapshot 请求的时候，它会通过 boltdb 接口创建一个只读事务 Tx，随后通过事务的

WriteTo 接口，将 meta page 和 data page 拷贝到 buffer 即可。

但是随着 db 文件增大，快照事务执行的时间也会越来越长，而长事务则会导致 db 文件大小发生显著增加。

也就是说当 db 大时，生成快照不仅慢，生成快照时可能还会触发 db 文件大小持续增长，最终达到配额限制。

为什么长事务可能会导致 db 大小增长呢？这个问题我先将它作为思考题，你可以分享一下你的想法，后续我将为你详细解答。

快照的另一大作用是当 Follower 节点异常的时候，Leader 生成快照发送给 Follower 节点，Follower 使用快照重建并追赶上 Leader。此过程涉及到一定的 CPU、内存、网络带宽等资源开销。

同时，若快照和集群写 QPS 较大，Leader 发送快照给 Follower 和 Follower 应用快照到状态机的流程会耗费较长的时间，这可能会导致基于快照重建后的 Follower 依然无法通过正常的日志复制模式来追赶 Leader，只能继续触发 Leader 生成快照，进而进入死循环，Follower 一直处于异常中。

## 小结

最后我们来小结下今天的内容。大 db 文件首先会影响 etcd 启动耗时，因为 etcd 需要打开 db 文件，初始化 db 对象，并遍历 boltdb 中的所有 key-value 以重建内存 treeIndex。

其次，较大 db 文件会导致 etcd 依赖更高配置的节点内存规格，etcd 通过 mmap 将 db 文件映射到内存中。etcd 启动后，正常情况下读 etcd 过程不涉及磁盘 IO，若节点内存不够，可能会导致缺页中断，引起延时抖动、服务性能下降。

接着 treeIndex 维护了所有 key 的版本号信息，当 treeIndex 中含有百万级 key 时，在 treeIndex 中搜索指定范围的 key 的开销是不能忽略的，此开销可能高达上百毫秒。

然后当 db 文件过大后，boltdb 本身连续空闲页的申请、释放、存储都会存在一定的开销。etcd 社区已通过新的 freelist 管理数据结构 hashmap 对其进行优化，将时间复杂度

降低到了  $O(1)$ ，同时支持事务提交时不持久化 freelist，而是通过重启时扫描 page 重建，以提升 etcd 写性能、降低 db 大小。

随后我给你介绍了 db 文件过大后，count only、limit、大包查询等 expensive request 对集群稳定性的影响。建议你的业务尽量避免任何 expensive request 请求。

最后我们介绍了大 db 文件对快照功能的影响。大 db 文件意味着更长的备份时间，而更长的只读事务则可能会导致 db 文件增长。同时 Leader 发送快照与 Follower 基于快照重建都需要较长时间，在集群写请求较大的情况下，可能会陷入死循环，导致落后的 Follower 节点一直无法追赶上 Leader。

## 思考题

在使用 etcd 过程中，你遇到了哪些案例导致了 etcd db 大小突增呢？它们的本质原因是什么呢？

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，谢谢。

提建议

12.12 大促

# 每日一课 VIP 年卡

10分钟，解决你的技术难题

¥159/年 ¥365/年

每日一课  
VIP 年卡

仅3天，【点击】图片，立即抢购 >>>

© 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 12 | 一致性：为什么基于Raft实现的etcd还会出现数据不一致？

## 精选留言 (1)

写留言



写点啥呢

2021-02-17

请问下老师：

1. etcd启动时候构建treeindex为什么需要加锁？如果构建时候是单goroutine是否可以避免加锁操作？

2. ./benchmark put --key-size 32 --val-size 10240 --total 1000000 --key-space-size 2000000 --clients 50 --conns 50 这个命令total参数是1个M，为啥会插入1.2M个key...

展开 ∨

作者回复: 1. treeIndex在启动时和运行过程中，可能存在多个goroutine并发访问的情况下，比如还有compaction异步任务也会操作它。我这里说的阻塞在锁上，实际上是我之前尝试将构建treeIndex改成并行，因为默认只有1个goroutine串行构建的，发现效果依然不佳，我完善下描述。

2. 下面写了会执行一系列这样的命令，value也有变化，直到压测到14Gdb大小左右，key 1.2M个。



1

