



下载APP



11 | 压缩：如何回收旧版本数据？

2021-02-12 唐聪

etcd实战课

[进入课程 >](#)**讲述：王超凡**

时长 13:54 大小 12.74M



你好，我是唐聪。

今天是大年初一，你过年都有什么安排？今年过年对我来说，其实是比较特别的。除了家庭团聚走亲访友外，我多了一份陪伴。感谢你和我在这个专栏里一块精进，我衷心祝你在新的年里平安喜乐，万事胜意。

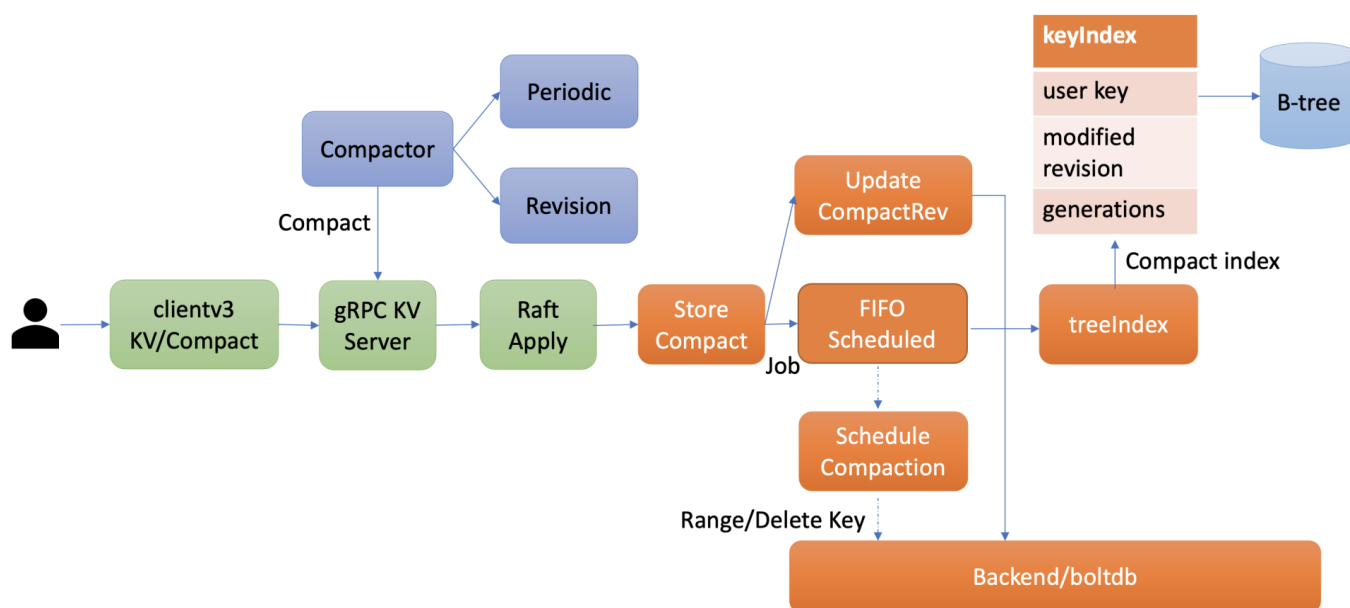
这节课是我们基础篇里的最后一节，正巧这节课的内容也是最轻松的。新年新气象，我们就带着轻松的心情开始吧！



在 [07](#) 里，我们知道 etcd 中的每一次更新、删除 key 操作，treeIndex 的 keyIndex 索引中都会追加一个版本号，在 boltdb 中会生成一个新版本 boltdb key 和 value。也就是随

那么 etcd 是通过什么机制来回收历史版本数据，控制索引内存占用和 db 大小的呢？

整体架构



当你通过 API 发起一个 Compact 请求后，KV Server 收到 Compact 请求提交到 Raft 模块处理，在 Raft 模块中提交后，Apply 模块就会通过 MVCC 模块的 Compact 接口执行此压缩任务。


Compact 接口首先会更新当前 server 已压缩的版本号，并将耗时昂贵的压缩任务保存到 FIFO 队列中异步执行。压缩任务执行时，它首先会压缩 treeIndex 模块中的 keyIndex 索引，其次会遍历 boltdb 中的 key，删除已废弃的 key。

以上就是压缩模块的一个工作流程。接下来我会首先和你介绍如何人工发起一个 Compact 操作，然后详细介绍周期性压缩模式、版本号压缩模式的工作原理，最后再给你介绍 Compact 操作核心的原理。

压缩特性初体验

在使用 etcd 过程中，当你遇到"etcdserver: mvcc: database space exceeded"错误时，若是你未开启压缩策略导致 db 大小达到配额，这时你可以使用 etcdctl compact 命令，主动触发压缩操作，回收历史版本。

如下所示，你可以先通过 endpoint status 命令获取 etcd 当前版本号，然后再通过 etcdctl compact 命令发起压缩操作即可。

 复制代码

```
1 # 获取etcd当前版本号
2 $ rev=$(etcdctl endpoint status --write-out="json" | egrep -o '"revision":[0-9
3 $ echo $rev
4 9
5 # 执行压缩操作，指定压缩的版本号为当前版本号
6 $ etcdctl compact $rev
7 Compacted revision 9
8 # 压缩一个已经压缩的版本号
9 $ etcdctl compact $rev
10 Error: etcdserver: mvcc: required revision has been compacted
11 # 压缩一个比当前最大版号大的版本号
12 $ etcdctl compact 12
13 Error: etcdserver: mvcc: required revision is a future revision
```

请注意，如果你压缩命令传递的版本号小于等于当前 etcd server 记录的压缩版本号，etcd server 会返回已压缩错误 ("mvcc: required revision has been compacted") 给 client。如果版本号大于当前 etcd server 最新的版本号，etcd server 则返回一个未来的版本号错误给 client("mvcc: required revision is a future revision")。

执行压缩命令的时候，不少初学者有一个常见的误区，就是担心压缩会不会把我最新版本数据给删除？

压缩的本质是**回收历史版本**，目标对象仅是**历史版本**，不包括一个 key-value 数据的最新版本，因此你可以放心执行压缩命令，不会删除你的最新版本数据。不过我在 [08](#) 介绍

Watch 机制时提到，Watch 特性中的历史版本数据同步，依赖于 MVCC 中是否还保存了相关数据，因此我建议你不要每次简单粗暴地回收所有历史版本。

在生产环境中，我建议你精细化的控制历史版本数，那如何实现精细化控制呢？

主要有两种方案，一种是使用 etcd server 的自带的自动压缩机制，根据你的业务场景，配置合适的压缩策略即可。

另外一种方案是如果你觉得 etcd server 的自带压缩机制无法满足你的诉求，想更精细化的控制 etcd 保留的历史版本记录，你就可以基于 etcd 的 Compact API，在业务逻辑代码中、或定时任务中主动触发压缩操作。你需要确保发起 Compact 操作的程序高可用，压缩的频率、保留的历史版本在合理范围内，并最终能使 etcd 的 db 大小保持平稳，否则会导致 db 大小不断增长，直至 db 配额满，无法写入。

在一般情况下，我建议使用 etcd 自带的压缩机制。它支持两种模式，分别是按时间周期性压缩和保留版本号的压缩，配置相应策略后，etcd 节点会自动化的发起 Compact 操作。

接下来我就和你详细介绍下 etcd 的周期性和保留版本号压缩模式。


周期性压缩

首先是周期性压缩模式，它适用于什么场景呢？

当你希望 etcd 只保留最近一段时间写入的历史版本时，你就可以选择配置 etcd 的压缩模式为 periodic，保留时间为你自定义的 1h 等。

如何给 etcd server 配置压缩模式和保留时间呢？

如下所示，etcd server 提供了配置压缩模式和保留时间的参数：

 复制代码

```
1 --auto-compaction-retention '0'  
2 Auto compaction retention length. 0 means disable auto Compaction.  
3 --auto-compaction-mode 'periodic'  
4 Interpret 'auto-Compaction-retention' one of: periodic|revision.
```

auto-compaction-mode 为 periodic 时，它表示启用时间周期性压缩，auto-compaction-retention 为保留的时间的周期，比如 1h。

auto-compaction-mode 为 revision 时，它表示启用版本号压缩模式，auto-compaction-retention 为保留的历史版本号数，比如 10000。

注意，etcd server 的 auto-compaction-retention 为'0'时，将关闭自动压缩策略，

那么周期性压缩模式的原理是怎样的呢？etcd 是如何知道你配置的 1h 前的 etcd server 版本号呢？

其实非常简单，etcd server 启动后，根据你的配置的模式 periodic，会创建 periodic Compactor，它会异步的获取、记录过去一段时间的版本号。periodic Compactor 组件获取你设置的压缩间隔参数 1h，并将其划分成 10 个区间，也就是每个区间 6 分钟。每隔 6 分钟，它会通过 etcd MVCC 模块的接口获取当前的 server 版本号，追加到 rev 数组中。

因为你只需要保留过去 1 个小时的历史版本，periodic Compactor 组件会通过当前时间减去上一次成功执行 Compact 操作的时间，如果间隔大于一个小时，它会取出 rev 数组的首元素，通过 etcd server 的 Compact 接口，发起压缩操作。

需要注意的一点是，在 etcd v3.3.3 版本之前，不同的 etcd 版本对周期性压缩的行为是有一定差异的，具体的区别你可以参考下 [🔗 官方文档](#)。


版本号压缩

了解完周期性压缩模式，我们再看看版本号压缩模式，它又适用于什么场景呢？

当你写请求比较多，可能产生比较多的历史版本导致 db 增长时，或者不确定配置 periodic 周期为多少才是最佳的时候，你可以通过设置压缩模式为 revision，指定保留的历史版本号数。比如你希望 etcd 尽量只保存 1 万个历史版本，那么你可以指定 compaction-mode 为 revision，auto-compaction-retention 为 10000。

它的实现原理又是怎样的呢？

也很简单，etcd 启动后会根据你的压缩模式 revision，创建 revision Compactor。revision Compactor 会根据你设置的保留版本号数，每隔 5 分钟定时获取当前 server 的最大版本号，减去你想保留的历史版本数，然后通过 etcd server 的 Compact 接口发起如下的压缩操作即可。

 复制代码

```
1 # 获取当前版本号，减去保留的版本号数
2 rev := rc.rg.Rev() - rc.retention
3 # 调用server的Compact接口压缩
4 _, err := rc.c.Compact(rc.ctx, &pb.CompactionRequest{Revision: rev})
```

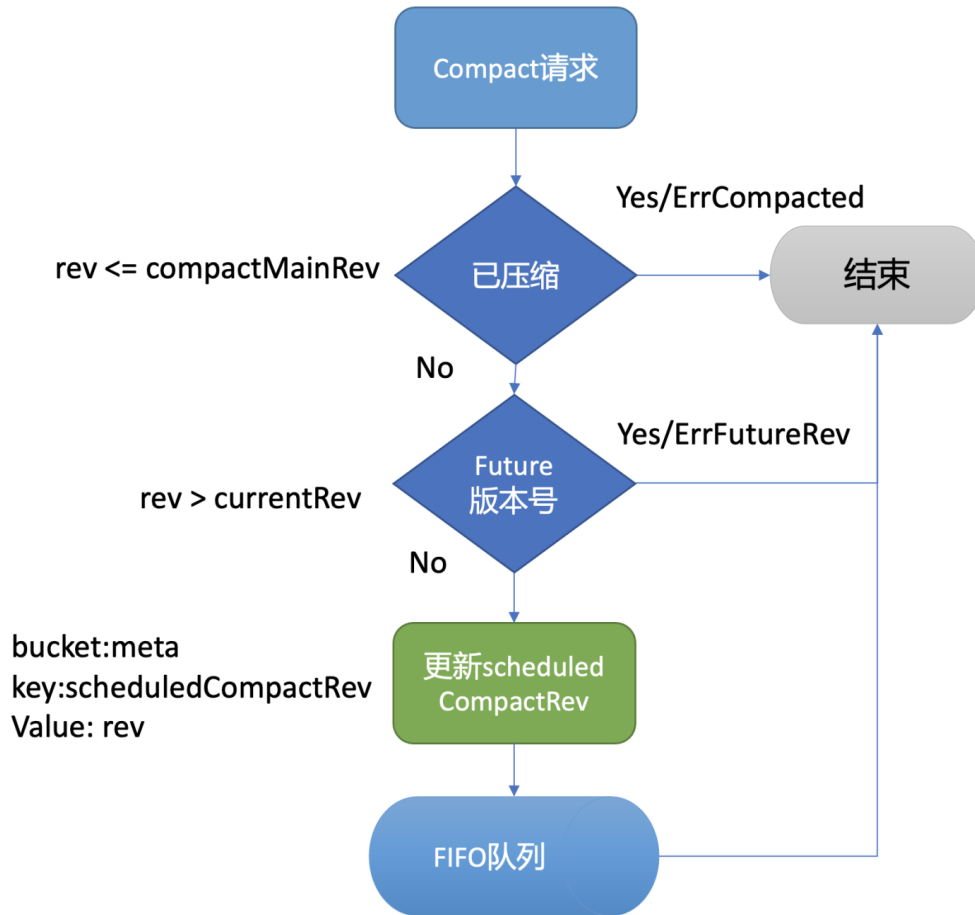
压缩原理

介绍完两种自动化的压缩模式原理后，接下来我们就深入分析下压缩的本质。当 etcd server 收到 Compact 请求后，它是如何执行的呢？核心原理是什么？

如前面的整体架构图所述，Compact 请求经过 Raft 日志同步给多数节点后，etcd 会从 Raft 日志取出 Compact 请求，应用此请求到状态机执行。

执行流程如下图所示，MVCC 模块的 Compact 接口首先会检查 Compact 请求的版本号 rev 是否已被压缩过，若是则返回 ErrCompacted 错误给 client。其次会检查 rev 是否大于当前 etcd server 的最大版本号，若是则返回 ErrFutureRev 给 client，这就是我们上面执行 etcdctl compact 命令所看到的那两个错误原理。

通过检查后，Compact 接口会通过 boltdb 的 API 在 meta bucket 中更新当前已调度的压缩版本号 (scheduledCompactedRev) 号，然后将压缩任务追加到 FIFO Scheduled 中，异步调度执行。



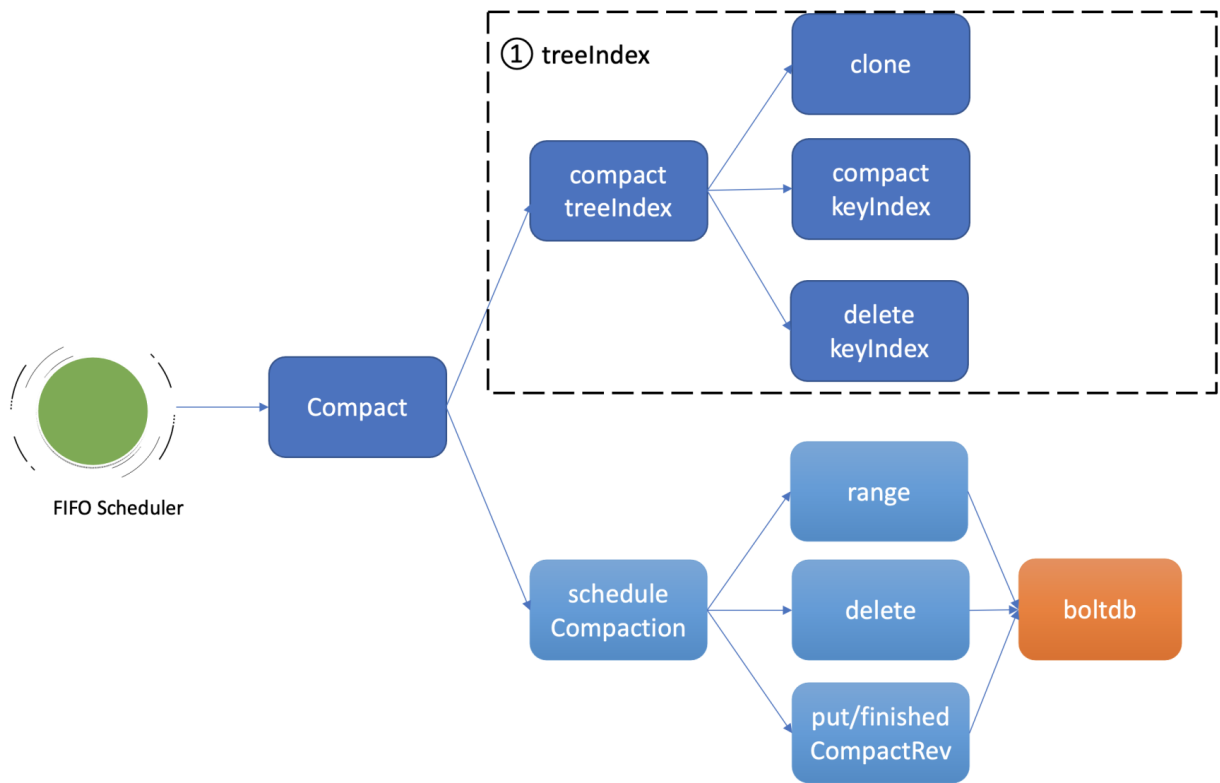
为什么 Compact 接口需要持久化存储当前已调度的压缩版本号到 boltdb 中呢？

试想下如果不保存这个版本号，etcd 在异步执行的 Compact 任务过程中 crash 了，那么异常节点重启后，各个节点数据就会不一致。

因此 etcd 通过持久化存储 `scheduledCompactedRev`，节点 crash 重启后，会重新向 FIFO Scheduled 中添加压缩任务，已保证各个节点间的数据一致性。

异步的执行压缩任务会做哪些工作呢？

首先我们回顾下 [07](#)里介绍的 `treeIndex` 索引模块，它是 etcd 支持保存历史版本的核心模块，每个 key 在 `treeIndex` 模块中都有一个 `keyIndex` 数据结构，记录其历史版本号信息。



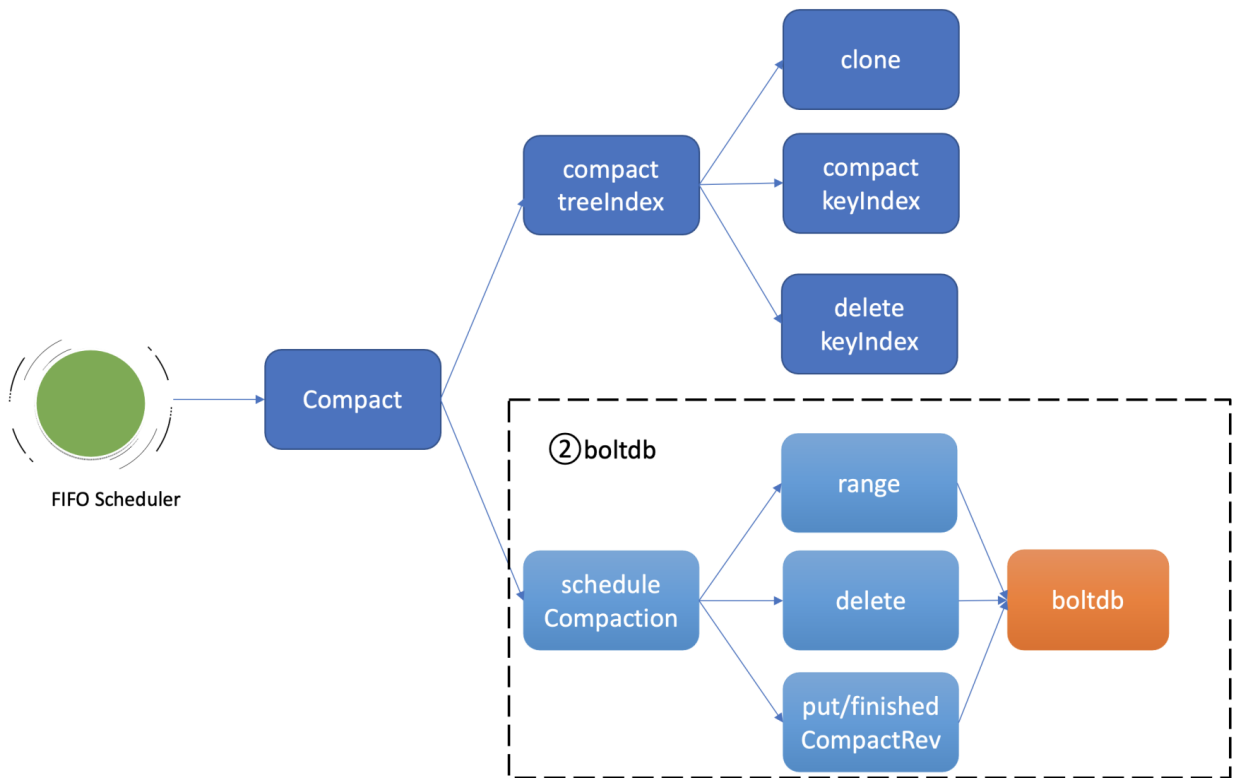
如上图所示，因此异步压缩任务的第一项工作，就是**压缩 treeIndex 模块中的各 key 的历史版本**、已删除的版本。为了避免压缩工作影响读写性能，首先会克隆一个 B-tree，然后通过克隆后的 B-tree 遍历每一个 keyIndex 对象，压缩历史版本号、清理已删除的版本。

假设当前压缩的版本号是 CompactedRev，它会保留 keyIndex 中最大的版本号，移除小于等于 CompactedRev 的版本号，并通过一个 map 记录 treeIndex 中有效的版本号返回给 boltdb 模块使用。

为什么要保留最大版本号呢？

因为最大版本号是这个 key 的最新版本，移除了会导致 key 丢失。而 Compact 的目的是回收旧版本。当然如果 keyIndex 中的最大版本号被打上了删除标记 (tombstone)，就会从 treeIndex 中删除这个 keyIndex，否则会出现内存泄露。

Compact 任务执行完索引压缩后，它通过遍历 B-tree、keyIndex 中的所有 generation 获得当前内存索引模块中有效的版本号，这些信息将帮助 etcd 清理 boltdb 中的废弃历史版本。



压缩任务的第二项工作就是**删除 bolt db 中废弃的历史版本数据**。如上图所示，它通过 etcd 一个名为 scheduleCompaction 任务来完成。

scheduleCompaction 任务会根据 key 区间，从 0 到 CompactedRev 遍历 bolt db 中的所有 key，通过 treeIndex 模块返回的有效索引信息，判断这个 key 是否有效，无效则调用 bolt db 的 delete 接口将 key-value 数据删除。

在这过程中，scheduleCompaction 任务还会更新当前 etcd 已经完成的压缩版本号 (finishedCompactRev)，将其保存到 bolt db 的 meta bucket 中。

scheduleCompaction 任务遍历、删除 key 的过程可能会对 bolt db 造成压力，为了不影响正常读写请求，它在执行过程中会通过参数控制每次遍历、删除的 key 数（默认为 100，每批间隔 10ms），分批完成 bolt db key 的删除操作。

为什么压缩后 db 大小不减少呢？

当你执行完压缩任务后，db 大小减少了吗？事实是并没有减少。那为什么我们都通过 bolt db API 删除了 key，db 大小还不减少呢？

上节课我们介绍 bolt db 实现时，提到过 bolt db 将 db 文件划分成若干个 page 页，page 页又有四种类型，分别是 meta page、branch page、leaf page 以及 freelist page。

branch page 保存 B+ tree 的非叶子节点 key 数据，leaf page 保存 bucket 和 key-value 数据，freelist 会记录哪些页是空闲的。

当我们通过 boltdb 删除大量的 key，在事务提交后 B+ tree 经过分裂、平衡，会释放出若干 branch/leaf page 页面，然而 boltdb 并不会将其释放给磁盘，调整 db 大小操作是昂贵的，会对性能有较大的损害。

boltdb 是通过 freelist page 记录这些空闲页的分布位置，当收到新的写请求时，优先从空闲页数组中申请若干连续页使用，实现高性能的读写（而不是直接扩大 db 大小）。当连续空闲页申请无法得到满足的时候，boltdb 才会通过增大 db 大小来补充空闲页。

一般情况下，压缩操作释放的空闲页就能满足后续新增写请求的空闲页需求，db 大小会趋于整体稳定。

小结

最后我们来小结下今天的内容。

etcd 压缩操作可通过 API 人工触发，也可以配置压缩模式由 etcd server 自动触发。压缩模式支持按周期和版本两种。在周期模式中你可以实现保留最近一段时间的历史版本数，在版本模式中你可以实现保留期望的历史版本数。

压缩的核心工作原理分为两大任务，第一个任务是压缩 treeIndex 中的各 key 历史索引，清理已删除 key，并将有效的版本号保存到 map 数据结构中。

第二个任务是删除 boltdb 中的无效 key。基本原理是根据版本号遍历 boltdb 已压缩区间范围的 key，通过 treeIndex 返回的有效索引 map 数据结构判断 key 是否有效，无效则通过 boltdb API 删除它。

最后在执行压缩的操作中，虽然我们删除了 boltdb db 的 key-value 数据，但是 db 大小并不会减少。db 大小不变的原因是存放 key-value 数据的 branch 和 leaf 页，它们释放后变成了空闲页，并不会将空间释放给磁盘。

boltdb 通过 freelist page 来管理一系列空闲页，后续新增的写请求优先从 freelist 中申请空闲页使用，以提高性能。在写请求速率稳定、新增 key-value 较少的情况下，压缩操

作释放的空闲页就可以基本满足后续写请求对空闲页的需求，db 大小就会处于一个基本稳定、健康的状态。

思考题

你知道压缩与碎片整理 (defrag) 有哪些区别吗？为什么碎片整理会影响服务性能呢？你能想到哪些优化方案来降低碎片整理对服务性能的影响呢？

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 10 | boltdb：如何持久化存储你的key-value数据？

下一篇 12 | 一致性：为什么基于B+树实现的数据库会出现数据不一致？

精选留言 (2)

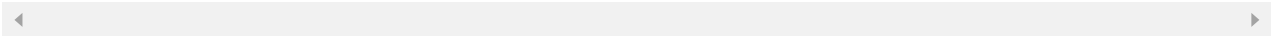
写留言



云原生工程师
2021-02-13

新年与老师一块精进，新年快乐，牛年一起旺旺旺。
展开

作者回复: 谢谢，新年快乐，祝牛年升职加薪



5



Geek_cb2b43
2021-02-13

压缩的第一项任务中克隆B+tree，是如何保两份数据一致的呢？即如果在此期间B+tree出现了分裂，原来的记录可能发生了变化
展开

