



下载APP



## 10 | boltdb: 如何持久化存储你的key-value数据?

2021-02-10 唐聪

etcd实战课

[进入课程 >](#)**讲述: 王超凡**

时长 17:54 大小 16.40M



你好, 我是唐聪。

在前面的课程里, 我和你多次提到过 etcd 数据存储在 boltdb。那么 boltdb 是如何组织你的 key-value 数据的呢? 当你读写一个 key 时, boltdb 是如何工作的?

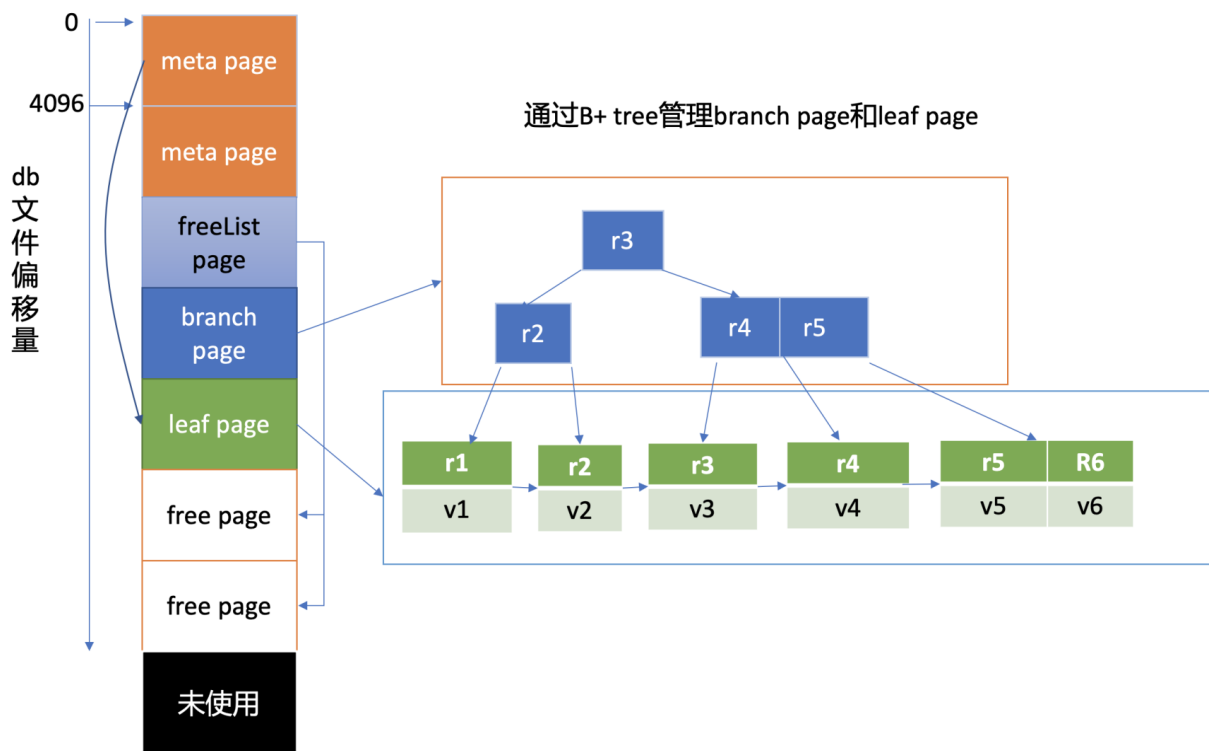
今天我将通过一个写请求在 boltdb 中执行的简要流程, 分析其背后的 boltdb 的磁盘文件布局, 帮助你了解 page、node、bucket 等核心数据结构的原理与作用, 搞懂 boltdb 基于 B+ tree、各类 page 实现查找、更新、事务提交的原理, 让你明白 etcd 为什么适合读多写少的场景。



### boltdb 磁盘布局

在介绍一个 put 写请求在 boltdb 中执行原理前, 我先给你从整体上介绍下平时你所看到的 etcd db 文件的磁盘布局, 让你了解下 db 文件的物理存储结构。

boltdb 文件指的是你 etcd 数据目录下的 member/snap/db 的文件, etcd 的 key-value、lease、meta、member、cluster、auth 等所有数据存储在其中。etcd 启动的时候, 会通过 mmap 机制将 db 文件映射到内存, 后续可从内存中快速读取文件中的数据。写请求通过 fwrite 和 fdatasync 来写入、持久化数据到磁盘。



上图是我给你画的 db 文件磁盘布局, 从图中的左边部分你可以看到, 文件的内容由若干个 page 组成, 一般情况下 page size 为 4KB。

page 按照功能可分为元数据页 (meta page)、B+ tree 索引节点页 (branch page)、B+ tree 叶子节点页 (leaf page)、空闲页管理页 (freelist page)、空闲页 (free page)。

文件最开头的两个 page 是固定的 db 元数据 meta page, 空闲页管理页记录了 db 中哪些页是空闲、可使用的。索引节点页保存了 B+ tree 的内部节点, 如图中的右边部分所示, 它们记录了 key 值, 叶子节点页记录了 B+ tree 中的 key-value 和 bucket 数据。

boltdb 逻辑上通过 B+ tree 来管理 branch/leaf page, 实现快速查找、写入 key-value 数据。

## boltdb API

了解完 boltdb 的磁盘布局后，那么如果要在 etcd 中执行一个 put 请求，boltdb 中是如何执行的呢？boltdb 作为一个库，提供了什么 API 给 client 访问写入数据？

boltdb 提供了非常简单的 API 给上层业务使用，当我们执行一个 put hello 为 world 命令时，boltdb 实际写入的 key 是版本号，value 为 mvccpb.KeyValue 结构体。

这里我们简化下，假设往 key bucket 写入一个 key 为 r94，value 为 world 的字符串，其核心代码如下：

[复制代码](#)

```
1 // 打开boltdb文件，获取db对象
2 db,err := bolt.Open("db", 0600, nil)
3 if err != nil {
4     log.Fatal(err)
5 }
6 defer db.Close()
7 // 参数true表示创建一个写事务，false读事务
8 tx,err := db.Begin(true)
9 if err != nil {
10     return err
11 }
12 defer tx.Rollback()
13 // 使用事务对象创建key bucket
14 b,err := tx.CreatebucketIfNotExists([]byte("key"))
15 if err != nil {
16     return err
17 }
18 // 使用bucket对象更新一个key
19 if err := b.Put([]byte("r94"),[]byte("world")); err != nil {
20     return err
21 }
22 // 提交事务
23 if err := tx.Commit(); err != nil {
24     return err
25 }
```

如上所示，通过 boltdb 的 Open API，我们获取到 boltdb 的核心对象 db 实例后，然后通过 db 的 Begin API 开启写事务，获得写事务对象 tx。

通过写事务对象 tx，你可以创建 bucket。这里我们创建了一个名为 key 的 bucket（如果不存在），并使用 bucket API 往其中更新了一个 key 为 r94，value 为 world 的数据。最后我们使用写事务的 Commit 接口提交整个事务，完成 bucket 创建和 key-value 数据写入。

看起来是不是非常简单，神秘的 boltdb，并未有我们想象的那么难。然而其 API 简单的背后却是 boltdb 的一系列巧妙的设计和实现。

一个 key-value 数据如何知道该存储在 db 在哪个 page？如何快速找到你的 key-value 数据？事务提交的原理又是怎样的呢？

接下来我就和你浅析 boltdb 背后的奥秘。


## 核心数据结构介绍

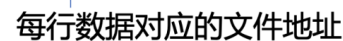
上面我们介绍 boltdb 的磁盘布局时提到，boltdb 整个文件由一个个 page 组成。最开头的两个 page 描述 db 元数据信息，而它正是在 client 调用 boltdb Open API 时被填充的。那么描述磁盘页面的 page 数据结构是怎样的呢？元数据页又含有哪些核心数据结构？

boltdb 本身自带了一个工具 bbolt，它可以按页打印出 db 文件的十六进制的内容，下面我们就使用此工具来揭开 db 文件的神秘面纱。

下图左边的十六进制是执行如下 `bbolt dump` 命令，所打印的 boltdb 第 0 页的数据，图的右边是对应的 page 磁盘页结构和 meta page 的数据结构。

```
1 $ ./bbolt dump ./infra1.etcd/member/snap/db 0
```

 复制代码



## page 磁盘页结构

数量字段仅在页类型为 leaf 和 branch 时生效，溢出页数量是指当前页面数据存放不下，需要向后再申请 overflow 个连续页面使用，页面数据起始位置指向 page 的载体数据，比如 meta page、branch/leaf 等 page 的内容。

第 0、1 页我们知道它是固定存储 db 元数据的页 (meta page), 那么 meta page 它为了管理整个 boltdb 含有哪些信息呢?



如上图中的 meta page 数据结构所示, 你可以看到它由 boltdb 的文件标识 (magic)、版本号 (version)、页大小 (pagesize)、boltdb 的根 bucket 信息 (root bucket)、freelist 页面 ID(freelist)、总的页面数量 (pgid)、上一次写事务 ID(txid)、校验码 (checksum) 组成。

## meta page 十六进制分析

了解完 page 磁盘页结构和 meta page 数据结构后, 我再结合图左边的十六进数据和你要分析下其含义。

上图中十六进制输出的是 db 文件的 page 0 页结构, 左边第一列表示此行十六进制内容对应的文件起始地址, 每行 16 个字节。

结合 page 磁盘页和 meta page 数据结构我们可知, 第一行前 8 个字节描述 pgid(忽略第一列) 是 0。接下来 2 个字节描述的页类型, 其值为 0x04 表示 meta page, 说明此页的数据存储的是 meta page 内容, 因此 ptr 开始的数据存储的是 meta page 内容。

正如你下图中所看到的, 第二行首先含有一个 4 字节的 magic number(0xED0CDAED), 通过它来识别当前文件是否 boltdb, 接下来是两个字节描述 boltdb 的版本号 0x2, 然后是四个字节的 page size 大小, 0x1000 表示 4096 个字节, 四个字节的 flags 为 0。

```
00000000 0000 0000 0000 0000 0400 0000 0000 0000
00000010 edda 0ced 0200 0000 0010 0000 0000 0000
00000020 0400 0000 0000 0000 0000 0000 0000 0000
00000030 0300 0000 0000 0000 0600 0000 0000 0000
00000040 1a00 0000 0000 0000 b0cf 5786 f584 3b7c
00000050 0000 0000 0000 0000 0000 0000 0000 0000
00000060 *
0000ff00 0000 0000 0000 0000 0000 0000 0000 0000
```

第三行对应的就是 meta page 的 root bucket 结构 (16 个字节), 它描述了 boltdb 的 root bucket 信息, 比如一个 db 中有哪些 bucket, bucket 里面的数据存储在哪里。

第四行中前面的 8 个字节, 0x3 表示 freelist 页面 ID, 此页面记录了 db 当前哪些页面是空闲的。后面 8 个字节, 0x6 表示当前 db 总的页面数。


第五行前面的 8 个字节, 0x1a 表示上一次的写事务 ID, 后面的 8 个字节表示校验码, 用于检测文件是否损坏。

了解完 db 元数据页面原理后, 那么 boltdb 是如何根据元数据页面信息快速找到你的 bucket 和 key-value 数据呢?

这就涉及到了元数据页面中的 root bucket, 它是个至关重要的数据结构。下面我们看看它是如何管理一系列 bucket、帮助我们查找、写入 key-value 数据到 boltdb 中。


## bucket 数据结构

如下命令所示, 你可以使用 bbolt buckets 命令, 输出一个 db 文件的 bucket 列表。执行完此命令后, 我们可以看到之前介绍过的 auth/lease/meta 等熟悉的 bucket, 它们都是 etcd 默认创建的。那么 boltdb 是如何存储、管理 bucket 的呢?

 复制代码

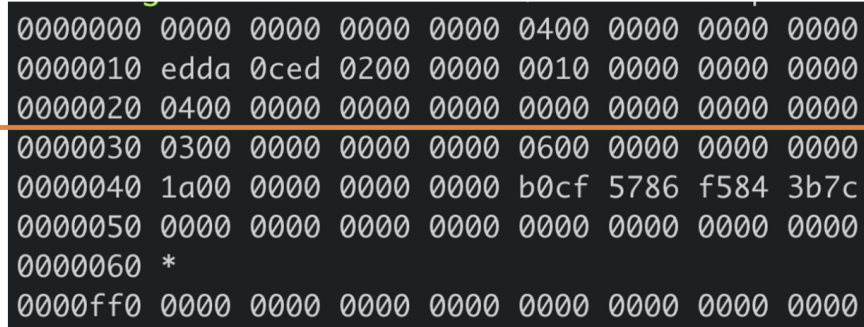
```
1 $ ./bbolt buckets ./infra1.etcd/member/snap/db
2 alarm
3 auth
4 authRoles
5 authUsers
6 cluster
7 key
8 lease
9 members
10 members_removed
11 meta
12
```

在上面我们提到过 meta page 中的, 有一个名为 root、类型 bucket 的重要数据结构, 如下所示, bucket 由 root 和 sequence 两个字段组成, root 表示该 bucket 根节点的 page id。注意 meta page 中的 bucket.root 字段, 存储的是 db 的 root bucket 页面信息, 你所看到的 key/lease/auth 等 bucket 都是 root bucket 的子 bucket。

 复制代码

```
1 type bucket struct {
2     root      pgid    // page id of the bucket's root-level page
3     sequence  uint64   // monotonically incrementing, used by NextSequence()
4 }
```

root bucket



```

00000000 0000 0000 0000 0000 0400 0000 0000 0000
00000100 edda 0ced 0200 0000 0010 0000 0000 0000
00000200 0400 0000 0000 0000 0000 0000 0000 0000
00000300 0300 0000 0000 0000 0600 0000 0000 0000
00000400 1a00 0000 0000 0000 b0cf 5786 f584 3b7c
00000500 0000 0000 0000 0000 0000 0000 0000 0000
00000600 *
0000ff00 0000 0000 0000 0000 0000 0000 0000 0000

```

上面 meta page 十六进制图中，第三行的 16 个字节就是描述的 root bucket 信息。root bucket 指向的 page id 为 4，page id 为 4 的页面是什么类型呢？我们可以通过如下 bbolt pages 命令看看各个 page 类型和元素数量，从下图结果可知，4 号页面为 leaf page。

复制代码

```

1 $ ./bbolt pages ./infra1.etcd/member/snap/db
2 ID      TYPE      ITEMS  OVRFLW
3 =====
4 0       meta       0
5 1       meta       0
6 2       free
7 3       freelist   2
8 4       leaf      10
9 5       free

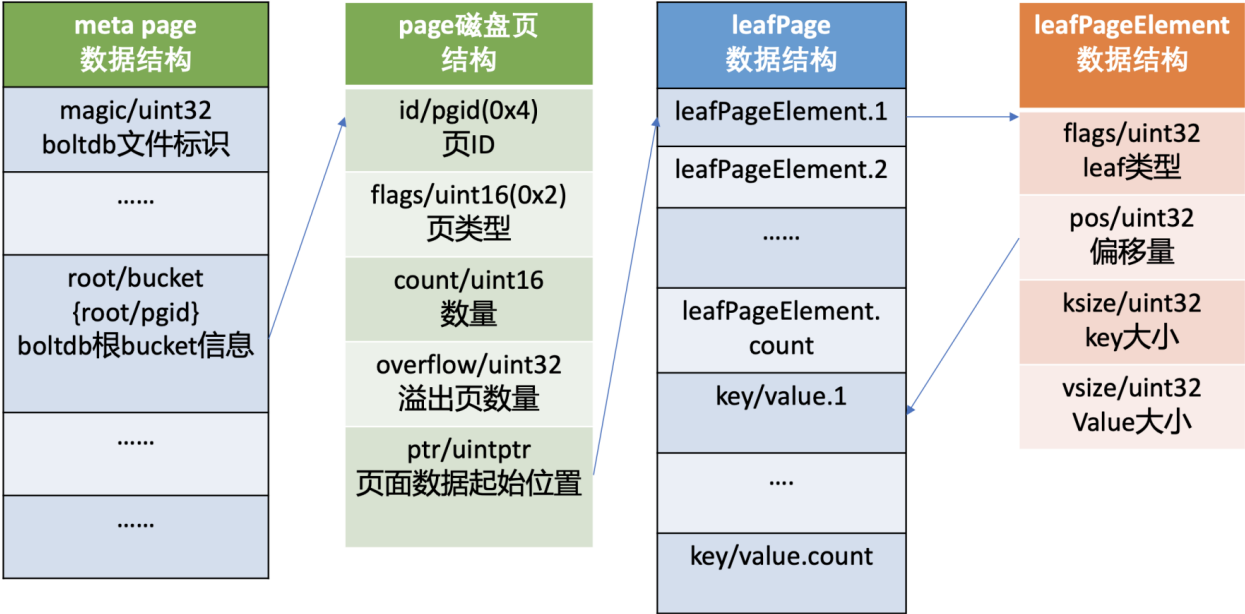
```

通过上面的分析可知，当 bucket 比较少时，我们子 bucket 数据可直接从 meta page 里指向的 leaf page 中找到。

## leaf page

meta page 的 root bucket 直接指向的是 page id 为 4 的 leaf page，page flag 为 0x02，leaf page 它的磁盘布局如下图所示，前半部分是 leafPageElement 数组，后半部分是 key-value 数组。





leafPageElement 包含 leaf page 的类型 flags，通过它可以区分存储的是 bucket 名称还是 key-value 数据。

当 flag 为 bucketLeafFlag(0x01) 时，表示存储的是 bucket 数据，否则存储的是 key-value 数据，leafPageElement 它还含有 key-value 的读取偏移量，key-value 大小，根据偏移量和 key-value 大小，我们就可以方便地从 leaf page 中解析出所有 key-value 对。

当存储的是 bucket 数据的时候，key 是 bucket 名称，value 则是 bucket 结构信息。bucket 结构信息含有 root page 信息，通过 root page（基于 B+ tree 查找算法），你可以快速找到你存储在这个 bucket 下面的 key-value 数据所在页面。

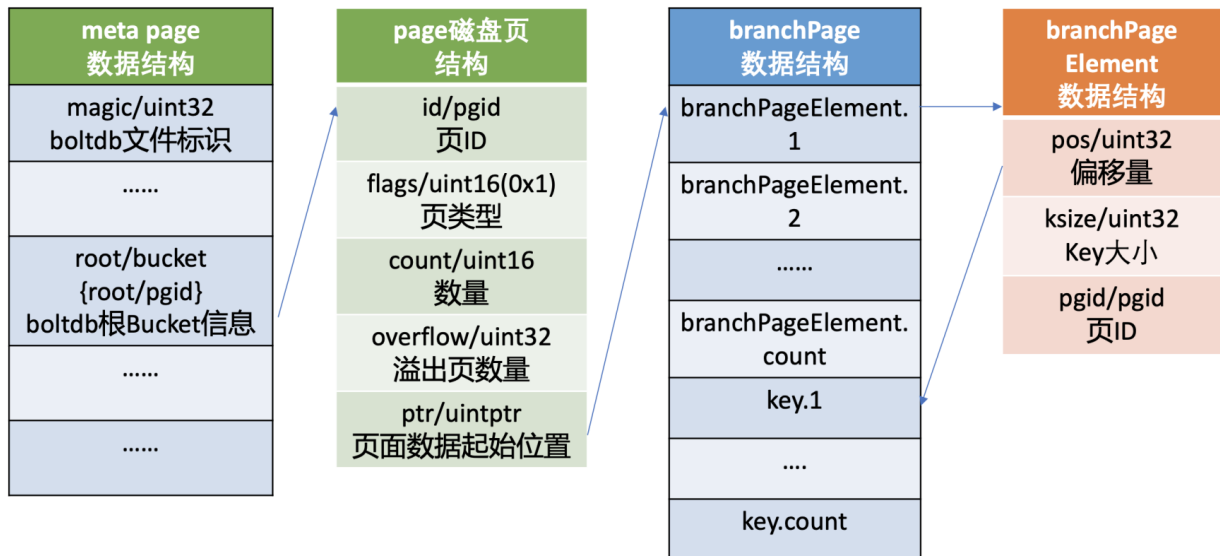
从上面分析你可以看到，每个子 bucket 至少需要一个 page 来存储其下面的 key-value 数据，如果子 bucket 数据量很少，就会造成磁盘空间的浪费。实际上 boltdb 实现了 inline bucket，在满足一些条件限制的情况下，可以将小的子 bucket 内嵌在它的父亲叶子节点上，友好的支持了大量小 bucket。

为了方便大家快速理解核心原理，本节我们讨论的 bucket 是假设都是非 inline bucket。

那么 boltdb 是如何管理大量 bucket、key-value 的呢？

branch page

boltdb 使用了 B+ tree 来高效管理所有子 bucket 和 key-value 数据, 因此它可以支持大量的 bucket 和 key-value, 只不过 B+ tree 的根节点不再直接指向 leaf page, 而是 branch page 索引节点页。branch page flags 为 0x01。它的磁盘布局如下图所示, 前半部分是 branchPageElement 数组, 后半部分是 key 数组。



branchPageElement 包含 key 的读取偏移量、key 大小、子节点的 page id。根据偏移量和 key 大小, 我们就可以方便地从 branch page 中解析出所有 key, 然后二分搜索匹配 key, 获取其子节点 page id, 递归搜索, 直至从 bucketLeafFlag 类型的 leaf page 中找到目的 bucket name。

注意, boltdb 在内存中使用了一个名为 node 的数据结构, 来保存 page 反序列化的结果。下面我给出了一个 boltdb 读取 page 到 node 的代码片段, 你可以直观感受下。

[复制代码](#)

```
1 func (n *node) read(p *page) {
2     n.pgid = p.id
3     n.isLeaf = ((p.flags & leafPageFlag) != 0)
4     n.inodes = make(inodes, int(p.count))
5
6
7     for i := 0; i < int(p.count); i++ {
8         inode := &n.inodes[i]
9         if n.isLeaf {
10             elem := p.leafPageElement(uint16(i))
11             inode.flags = elem.flags
12             inode.key = elem.key()
13         }
14     }
15 }
```

```
14         inode.value = elem.value()
15     } else {
16         elem := p.branchPageElement(uint16(i))
17         inode.pgid = elem.pgid
18         inode.key = elem.key()
19     }
```

从上面分析过程中你会发现，boltdb 存储 bucket 和 key-value 原理是类似的，将 page 划分成 branch page、leaf page，通过 B+ tree 来管理实现。boltdb 为了区分 leaf page 存储的数据类型是 bucket 还是 key-value，增加了标识字段 (leafPageElement.flags)，因此 key-value 的数据存储过程我就不再重复分析了。

## freelist

介绍完 bucket、key-value 存储原理后，我们再看 meta page 中的另外一个核心字段 freelist，它的作用是什么呢？

我们知道 boltdb 将 db 划分成若干个 page，那么它是如何知道哪些 page 在使用中，哪些 page 未使用呢？

答案是 boltdb 通过 meta page 中的 freelist 来管理页面的分配，freelist page 中记录了哪些页是空闲的。当你在 boltdb 中删除大量数据的时候，其对应的 page 就会被释放，页 ID 存储到 freelist 所指向的空闲页中。当你写入数据的时候，就可直接从空闲页中申请页面使用。

下面 meta page 十六进制图中，第四行的前 8 个字节就是描述的 freelist 信息，page id 为 3。我们可以通过 bbolt page 命令查看 3 号 page 内容，如下所示，它记录了 2 和 5 为空闲页，与我们上面通过 bbolt pages 命令所看到的信息一致。

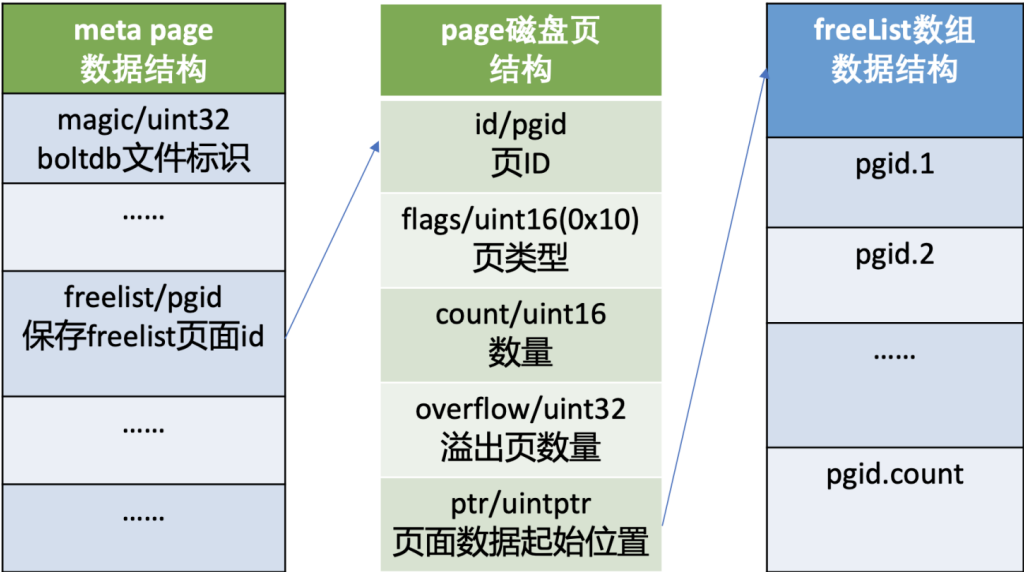
freelist  
page

00000000	0000	0000	0000	0000	0400	0000	0000	0000
0000010	edda	0ced	0200	0000	0010	0000	0000	0000
0000020	0400	0000	0000	0000	0000	0000	0000	0000
0000030	0300	0000	0000	0000	0600	0000	0000	0000
0000040	1a00	0000	0000	0000	b0cf	5786	f584	3b7c
0000050	0000	0000	0000	0000	0000	0000	0000	0000
0000060	*							
0000ff0	0000	0000	0000	0000	0000	0000	0000	0000

复制代码

```
1 $ ./bbolt page ./infra1.etcd/member/snap/db 3
2 page ID:      3
3 page Type:    freelist
4 Total Size:   4096 bytes
5 Item Count:   2
6 Overflow:     0
7
8 2
9 5
```

下图是 freelist page 存储结构，pageflags 为 0x10，表示 freelist 类型的页，ptr 指向空闲页 id 数组。注意在 boltdb 中支持通过多种数据结构（数组和 hashmap）来管理 free page，这里我介绍的是数组。



## Open 原理

了解完核心数据结构后，我们就很容易搞懂 boltdb Open API 的原理了。

首先它会打开 db 文件并对其增加文件锁，目的是防止其他进程也以读写模式打开它后，操作 meta 和 free page，导致 db 文件损坏。

其次 boltdb 通过 mmap 机制将 db 文件映射到内存中，并读取两个 meta page 到 db 对象实例中，然后校验 meta page 的 magic、version、checksum 是否有效，若两个 meta page 都无效，那么 db 文件就出现了严重损坏，导致异常退出。

## Put 原理

那么成功获取 db 对象实例后，通过 bucket API 创建一个 bucket、发起一个 Put 请求更新数据时，boltdb 是如何工作的呢？

根据我们上面介绍的 bucket 的核心原理，它首先是根据 meta page 中记录 root bucket 的 root page，按照 B+ tree 的查找算法，从 root page 递归搜索到对应的叶子节点 page 面，返回 key 名称、leaf 类型。

如果 leaf 类型为 bucketLeafFlag，且 key 相等，那么说明已经创建过，不允许 bucket 重复创建，结束请求。否则往 B+ tree 中添加一个 flag 为 bucketLeafFlag 的 key，key 名称为 bucket name，value 为 bucket 的结构。

创建完 bucket 后，你可以通过 bucket 的 Put API 发起一个 Put 请求更新数据。它的核心原理跟 bucket 类似，根据子 bucket 的 root page，从 root page 递归搜索此 key 到 leaf page，如果没有找到，则在返回的位置处插入新 key 和 value。

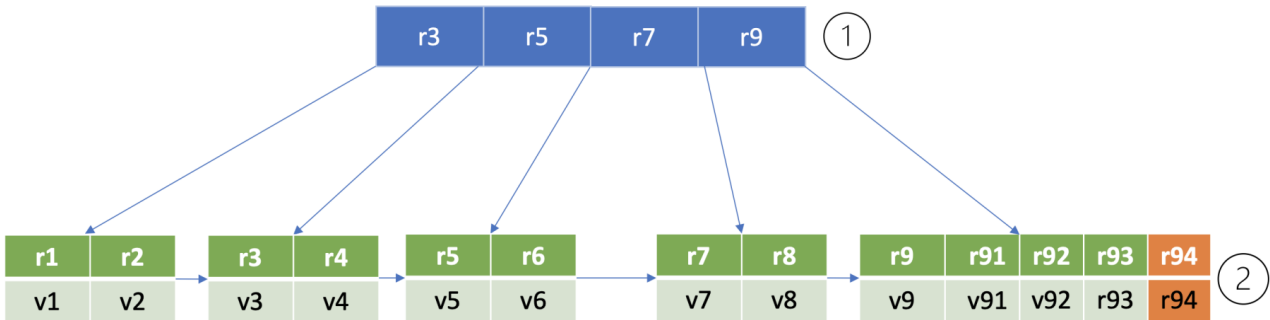
为了方便你理解 B+ tree 查找、插入一个 key 原理，我给你构造了一个 max degree 为 5 的 B+ tree，下图是 key r94 的查找流程图。

那么如何确定这个 key 的插入位置呢？

首先从 boltdb 的 key bucket 的 root page 里，二分查找大于等于 r94 的 key 所在 page，最终找到 key r9 指向的 page（流程 1）。r9 指向的 page 是个 leaf page，B+



tree 需要确保叶子节点 key 的有序性，因此同样二分查找其插入位置，将 key r94 插入到相关位置（流程二）。



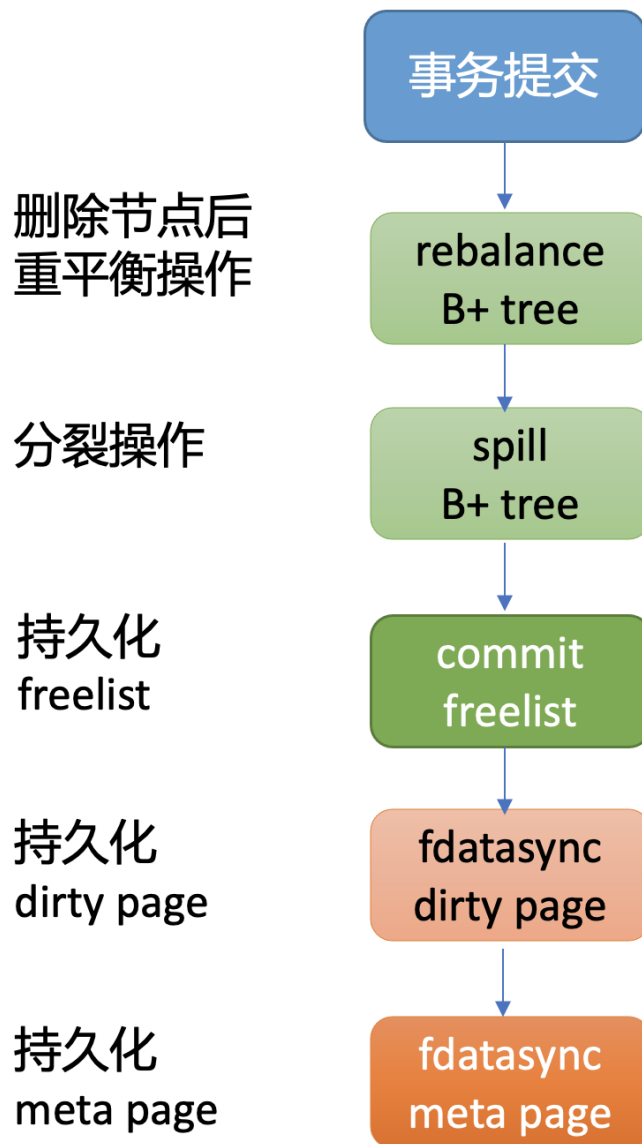
在核心数据结构介绍中，我和你提到 boltdb 在内存中通过 node 数据结构来存储 page 磁盘页内容，它记录了 key-value 数据、page id、parent 及 children 的 node、B+ tree 是否需要重平衡和分裂操作等信息。

因此，当我们执行完一个 put 请求时，它只是将值更新到 boltdb 的内存 node 数据结构里，并未持久化到磁盘中。

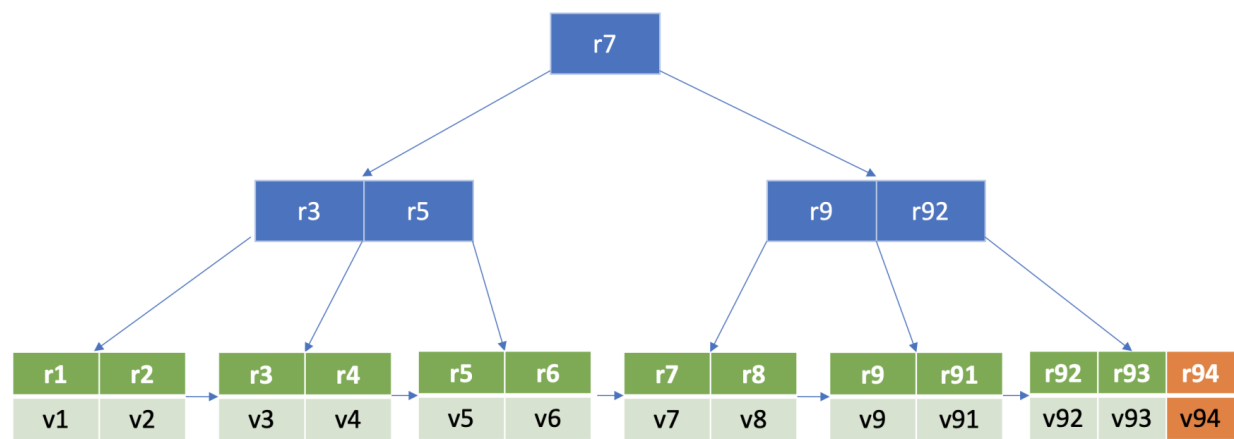
### 事务提交原理

那么 boltdb 何时将数据持久化到 db 文件中呢？

当你的代码执行 tx.Commit API 时，它才会将我们上面保存到 node 内存数据结构中的数据，持久化到 boltdb 中。下图我给出了一个事务提交的流程图，接下来我就分别和你简要分析下各个核心步骤。



首先从上面 put 案例中我们可以看到，插入了一个新的元素在 B+ tree 的叶子节点，它可能已不满足 B+ tree 的特性，因此事务提交时，第一步首先要调整 B+ tree，进行重平衡、分裂操作，使其满足 B+ tree 树的特性。上面案例里插入一个 key r94 后，经过重平衡、分裂操作后的 B+ tree 如下图所示。



在重平衡、分裂过程中可能会申请、释放 free page，freelist 所管理的 free page 也发生了变化。因此事务提交的第二步，就是持久化 freelist。

注意，在 etcd v3.4.9 中，为了优化写性能等，freelist 持久化功能是关闭的。etcd 启动获取 boltdb db 对象的时候，boltdb 会遍历所有 page，构建空闲页列表。

事务提交的第三步就是将 client 更新操作产生的 dirty page 通过 fdatsync 系统调用，持久化存储到磁盘中。

最后，在执行写事务过程中，meta page 的 txid、freelist 等字段会发生变化，因此事务的最后一步就是持久化 meta page。

通过以上四大步骤，我们就完成了事务提交的工作，成功将数据持久化到了磁盘文件中，安全地完成了一个 put 操作。

## 小结

最后我们来小结下今天的内容。首先我通过一幅 boltdb 磁盘布局图和 bbolt 工具，为你解密了 db 文件的本质。db 文件由 meta page、freelist page、branch page、leaf page、free page 组成。随后我结合 bbolt 工具，和你深入介绍了 meta page、branch page、leaf page、freelist page 的数据结构，帮助你了解 key、value 数据是如何存储到文件中的。

然后我通过分析一个 put 请求在 boltdb 中如何执行的。我从 Open API 获取 db 对象说起，介绍了其通过 mmap 将 db 文件映射到内存，构建 meta page，校验 meta page 的有效性，再到创建 bucket，通过 bucket API 往 boltdb 添加 key-value 数据。

添加 bucket 和 key-value 操作本质，是从 B+ tree 管理的 page 中找到插入的页和位置，并将数据更新到 page 的内存 node 数据结构中。

真正持久化数据到磁盘是通过事务提交执行的。它首先需要通过一系列重平衡、分裂操作，确保 boltdb 维护的 B+ tree 满足相关特性，其次需要持久化 freelist page，并将用户更新操作产生的 dirty page 数据持久化到磁盘中，最后则是持久化 meta page。

## 思考题

事务提交过程中若持久化 key-value 数据到磁盘成功了，此时突然掉电，元数据还未持久化到磁盘，那么 db 文件会损坏吗？数据会丢失吗？为什么 boltdb 有两个 meta page 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 09 | 事务：如何安全地实现多key操作？

下一篇 11 | 压缩：如何回收旧版本数据？

## 精选留言 (3)

写留言



issac

2021-02-10

唐聪老师，能把每节课的思考题解答一下吗？觉得都是重点有意思的地方，非常感谢老师的辛苦付出！

作者回复: 嗯，一定会的，春节快乐，近期大量时间投入在新稿，思考题后面我会一个个抽空更新，先看看大家自己思考情况，你也可以把你的想法观点在留言里面回复下。



8



唐聪

2021-02-10



文中提到的bbolt是etcd社区基于boltdb fork的一个版本, etcd社区负责维护此版本, 原因是boltdb作者认为boltdb已经足够成熟稳定, 经过了大规模生产环境检验, 新特性和优化点合入会对boltdb稳定性造成一定的影响, 个人没更多时间再投入到boltdb上, 因此boltdb项目变成archived状态

展开 ∨



6



写点啥呢

2021-02-10

请问下唐老师, 文章中“boltdb API”一节的示例代码里, "defer tx.Rollback()"将事务回滚放到defer链里, 如果事务正常提交了, 这时候defer链再调用Rollback会是什么效果? 是不是Rollback报错而我们直接忽略掉?

展开 ∨

作者回复: 事务正常提交后, tx.db等相关字段会设置为空, 执行tx.Rollback后发现tx.db为空, 就直接返回了哈, 相当于空操作

