# 自底向上分析 BoltDB 源码

这本书籍主要是针对golang编写的kv数据库boltdb进行源码分析，目前市面上完整分析boltdb源码的文章少之又少，而且大部分分析也是停留在理论基础上，所以在本人研究了一段时间后，将自己的理解和总结汇总了下，...

# 目　录

# 致谢

# 自底向上分析boltdb源码

本书是采用自底向上的方式来介绍boltdb内部的实现原理。其实我们经常都在采用自底向上或者自顶向下这两种方式来思考和求解问题。

例如：我们阅读源码时，通常都是从最顶层的接口点进去，然后层层深入内部。这其实本质上就是一种自顶向下的方式。

又比如我们平常做开发时，都是先将系统进行拆分、解耦。然后一般都会采用从下而上或者从上而下的方式来进行开发迭代。

又比如在执行OKR的时候，我们通常都是先定目标、然后依据该目标再层层分解。最后分解到可执行的单元为止。这其实也是一种自顶向 下的方式；在真正执行时，我们通常又是从原先分解到最底层的原子单元开始执行，然后层层递进。最终所有的原子单元执行完后，我们 的目标也就实现了。这其实又是自底向上的完成任务方式。

前面所提到的上下其实是指某些事物、组件、目标内部是存在相互依赖、因果、先后、递进关系，而依赖方或者结果方则位于上，被依赖方、原因 方位于下；个人认为自下而上的方式比较适合执行每一个任务或者解决子问题，每一层做完时，都可以独立的进行测试和验证。当我们层层自下 而上开发。最后将其前面的所有东西拼接在一起时，就构成了一个完整的组件或者实现了该目标。

回到最初的话题，为什么本书要采用自底向上的方式来写呢？

对于一个文件型数据库而言，所谓的上指的是暴露给用户侧的调用接口。所谓的下又指它的输出(数据)最终要落到磁盘这种 存储介质上。采用自底向上的方式的话，也就意味着我们先从磁盘这一层进行分析。然后逐步衍生到内存；再到用户接口这一层。层层之间是 被依赖的一种关系。这样的话，其实就比较好理解了。在本书中，本人采用自底向上的方式来介绍。希望阅读完后，有一种自己从0到1构建了一块数据库的快感。

当然也可以采用自顶向下的方式来介绍，这时我们就需要在介绍最上层时，先假设它所依赖的底层都已经就绪了，我们只分析当层内容。然后层层 往下扩展。

之前和一位大佬进行过针对此问题的探讨，在不同的场景、不同的组件中。具体采用自底向上还是自顶向下来分析。见仁见智，也具体问题具体分析。当要达成的目标足够清晰时，通过自顶向下的方式可以倒推达成目标需要完成的几个阶段任务。然后再依次进行细分展开。

下面是我们的boltdb微信交流群，有问题、疑问都可以扫码加入群一起交流讨论,二维码过期的话也可以加群管理员或者加微信号 `wen2282186474` 申请入群，邀请入群。

boltdb数据库讨论交流群

该二维码7天内(4月24日前)有效，重新进入将更新

# 第一章 boltdb简要介绍

本章是我们的开篇，我们主要从以下几个方面做一个讲述。希望这章让大家认识一下boltdb，知道它是什么？做什么？后续所有的内容都建立再此基础上，给大家详细介绍它内部是怎么做的。因此本章内容的定位是为后续章节做一个过渡和铺垫，本章的主要内容从以下几个方面展开：

1. boltdb是什么？
2. 为什么要分析boltdb？
3. boltdb的简单用法
4. boltdb的整体数据组织结构
5. boltdb的黑科技

# 第一节 boltdb是什么？

在用自己的话介绍boltdb之前，我们先看下boltdb官方是如何自我介绍的呢？

> Bolt is a pure Go key/value store inspired by [Howard Chu's][hyc_symas] [LMDB project][lmdb]. The goal of the project is to provide a simple, fast, and reliable database for projects that don't require a full database server such as Postgres or MySQL.
>
> Since Bolt is meant to be used as such a low-level piece of functionality, simplicity is key. The API will be small and only focus on getting values and setting values. That's it.

看完了官方的介绍，接下来让我用一句话对boltdb进行介绍：

**boltdb**是一个纯**go**编写的支持事务的文件型单机**kv**数据库。

下面对上述几个核心的关键词进行一一补充。

纯**go**： 意味着该项目只由golang语言开发，不涉及其他语言的调用。因为大部分的数据库基本上都是由c或者c++开发的，boltdb是一款难得的golang编写的数据库。

支持事务： boltdb数据库支持两类事务：读写事务、只读事务。这一点就和其他kv数据库有很大区别。

文件型： boltdb所有的数据都是存储在磁盘上的，所以它属于文件型数据库。这里补充一下个人的理解，在某种维度来看，boltdb很像一个简陋版的innodb存储引擎。底层数据都存储在文件上，同时数据都涉及数据在内存和磁盘的转换。但不同的是，innodb在事务上的支持比较强大。

单机： boltdb不是分布式数据库，它是一款单机版的数据库。个人认为比较适合的场景是，用来做wal日志或者读多写少的存储场景。

**kv**数据库： boltdb不是sql类型的关系型数据库，它和其他的kv组件类似，对外暴露的是kv的接口，不过boltdb支持的数据类型key和value都是[]byte。

# 第二节 为什么要分析boltdb？

前文介绍完了什么是boltdb。那我们先扪心自问一下，为什么要学习、分析boltdb呢？闲的吗？

答案：当然不是。我们先看看其他几个人对这个问题是如何答复的。

github用户ZhengHe-MD是这么答复的：

> 要达到好的学习效果，就要有输出。以我平时的工作节奏，在闲暇时间依葫芦画瓢写一个键值数据库不太现实。于是我选择将自己对源码阅读心得系统地记录下来， 最终整理成本系列文章，旨在尽我所能正确地描述 boltDB。恰好我在多次尝试在网上寻找相关内容后，发现网上大多数的文章、视频仅仅是介绍 boltDB 的用法和特性。因此，也许本系列文章可以作为它们以及 boltDB 官方文档 的补充，帮助想了解它的人更快地、深入地了解 boltDB。 如果你和我一样是初学者，相信它对你会有所帮助；如果你是一名经验丰富的数据库工程师，也许本系列文章对你来说没有太多新意。

微信公众号作者TheFutureIsOurs是这么答复的： boltdb源码阅读

> 最近抽时间看了boltdb的源码，代码量不大（大概4000行左右），而且支持事务，结构也很清晰，由于比较稳定，已经归档，确实是学习数据库的最佳选择。 而且不少出名的开源项目在使用它，比如etcd，InfluxDB等。 本文记录下笔者在阅读源码后了解到的其工作原理，以留备忘。

下面我来以自身的角度来回答下这个问题：

首先在互联网里面，所有的系统、软件都离不开数据。而提到数据，无非我们就会想到数据的存储和数据检索。这些功能不就是一个数据库最基本的吗。从而数据库在计算机的世界里面 有着无比重要的位置。作为一个有梦想的程序员，总是想知其然并知其所以然。这个是驱动我决定看源码的原因之一。

其次最近在组里高涨的系统学习mysql、redis的氛围下，我也加入了阵营。想着把这两块知识好好消化、整理一番。尤其是mysql，大家主要还是以核心学习innodb存储引擎为目标。本人 也不例外，在我看完了从跟上理解**mysql**后。整体上对innodb有了宏观和微观的了解和认识，但是更近一步去看mysql的代码。有几个难点：1.本人项目主要以golang为主。说实话看 c和c++的项目或多或少有些理解难度；2.mysql作为上古神兽，虽然功能很完善，但是要想短期内看完源码基本上是不可能的，而工作之余的时间有限，因此性价比极低。而boltdb完美的符合了 我的这两个要求。所以这就是选择boltdb的第二个原因，也是一个主要原因。

主要还是想通过分析这个项目，在下面三个方面有所提升。

1. 一方面让自己能加深原先学习的理论知识；
2. 另外一方面也能真正的了解工程上是如何运用的，理论结合实践，然后对存储引擎有一个清晰的认识；
3. 最后也是希望借助这个项目开启个人探索存储方向的大门。

介绍完了学习bolt是什么？为什么要分析boltdb后，我们就正式进入主题了。让我们先以一个简单例

子认识下boltdb。

# 第三节 boltdb的简单用法

其实boltdb的用法很简单，从其项目github的文档里面就可以看得出来。它本身的定位是key/value(后面简称为kv)存储的嵌入式数据库，因此那提到kv我们自然而然能想到的最常用的操作，就是set(k,v)和get(k)了。确实如此boltdb也就是这么简单。

不过在详细介绍boltdb使用之前，我们先以日常生活中的一些场景来作为切入点，引入一些在boltdb中抽象出来的专属名词(DB、Bucket、Cursor、k/v等)，下面将进入正文，前面提到boltdb的使用确实很简单，就是set和get。但它还在此基础上还做了一些额外封装。下面通过现实生活对比来介绍这些概念。

boltdb本质就是存放数据的，那这和现实生活中的柜子就有点类似了，如果我们把boltdb看做是一个存放东西的柜子的话，它里面可以存放各种各样的东西，确实是的，但是我们想一想，所有东西都放在一起会不会有什么问题呢？

咦，如果我们把钢笔、铅笔、外套、毛衣、短袖、餐具这些都放在一个柜子里的话，会有啥问题呢？这对于哪些特别喜欢收拾屋子，东西归类放置的人而言，简直就是一个不可容忍的事情，因为所有的东西都存放在一起，当东西多了以后就会显得杂乱无章。

在生活中我们都有分类、归类的习惯，例如对功能类似的东西(钢笔、铅笔、圆珠笔等)放一起，或者同类型的东西(短袖、长袖等)放一起。把前面的柜子通过隔板来隔开，分为几个小的小柜子，第一个柜子可以放置衣服，第二个柜子可以放置书籍和笔等。当然了，这是很久以前的做法了，现在买的柜子，厂家都已经将其内部通过不同的存放东西的规格做好了分隔。大家也就不用为这些琐事操心了。既然这样，那把分类、归类这个概念往计算机中迁移过来，尤其是对于存放数据的数据库boltdb中，它也需要有分类、归类的思想，因为归根到底，它也是由人创造出来的嘛。

好了到这儿，我们引入我们的三大名词了**“DB”**、**“Bucket”**、**“k/v”**。

**DB:** 对应我们上面的柜子。

**Bucket:** 对应我们将柜子分隔后的小柜子或者抽屉了。

**k/v:** 对应我们放在抽屉里的每一件东西。为了方便我们后面使用的时候便捷，我们需要给每个东西都打上一个标记，这个标记是可以区分每件东西的，例如k可以是该物品的颜色、或者价格、或者购买日期等，v就对应具体的东西啦。这样当我们后面想用的时候，就很容易找到。尤其是女同胞们的衣服和包包，哈哈

再此我们就可以得到一个大概的层次结构，一个柜子(DB)里面可以有多个小柜子(Bucket)，每个小柜子里面存放的就是每个东西(k/v)啦。

那我们想一下，我们周末买了一件新衣服，回到家，我们要把衣服放在柜子里，那这时候需要怎么操作呢？

很简单啦，下面看看我们平常怎么做的。

第一步：如果家里没有柜子，那就得先买一个柜子；

第二步：在柜子里找找之前有没有放置衣服的小柜子，没有的话，那就分一块出来，总不能把新衣服和钢笔放在一块吧。

第三步：有了放衣服的柜子，那就里面找找，如果之前都没衣服，直接把衣服打上标签，然后丢进去就ok啦；如果之前有衣服，那我们就需要考虑要怎么放了，随便放还是按照一定的规则来放。这里我猜大部分人还是会和我一样吧。喜欢按照一定的规则放，比如按照衣服的新旧来摆放，或者按照衣服的颜色来摆放，或者按照季节来摆放，或者按照价格来摆放。哈哈

我们在多想一下，周一早上起来我们要找一件衣服穿着去上班，那这时候我们又该怎么操作呢？

第一步：去找家里存放东西的柜子，家里没柜子，那就连衣服都没了，尴尬….。所以我们肯定是有柜子的，对不对

第二步：找到柜子了，然后再去找放置衣服的小柜子，因为衣服在小柜子存放着。

第三步：找到衣服的柜子了，那就从里面找一件衣服了，找哪件呢！最新买的？最喜欢的？天气下雨了，穿厚一点的？天气升温了，穿薄一点的？今天没准可能要约会，穿最有气质的？…..

那这时候根据不同场景来确定了规则，明确了我们要找的衣服的标签，找起来就会很快了。我们一下子就能定位到要穿的衣服了。嗯哼，这就是排序、索引的威力了

如果之前放置的衣服没有按照这些规则来摆放。那这时候就很悲剧了，就得挨个挨个找，然后自己选了。哈哈，有点全表扫描的味道了

啰里啰嗦扯了一大堆，就是为了给大家科普清楚，一些boltdb中比较重要的概念，让大家对比理解。降低理解难度。下面开始介绍boltdb是如何简单使用的。

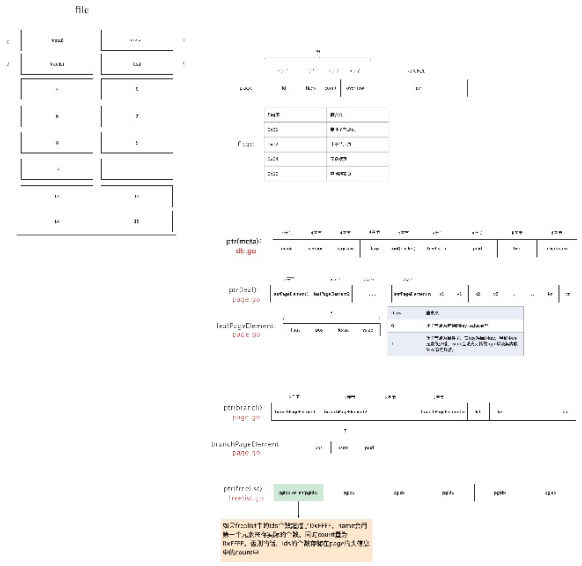
使用无外乎两个操作：**set、get**

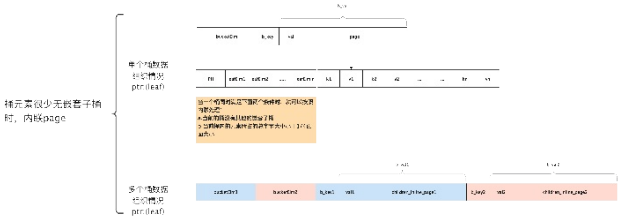
```
1.
2.  import "bolt"
3.
4.  func main(){
5.      // 我们的大柜子
6.      db, err := bolt.Open("./my.db", 0600, nil)
7.      if err != nil {
8.          panic(err)
9.      }
10.     defer db.Close()
11.     // 往db里面插入数据
12.     err = db.Update(func(tx *bolt.Tx) error {
```

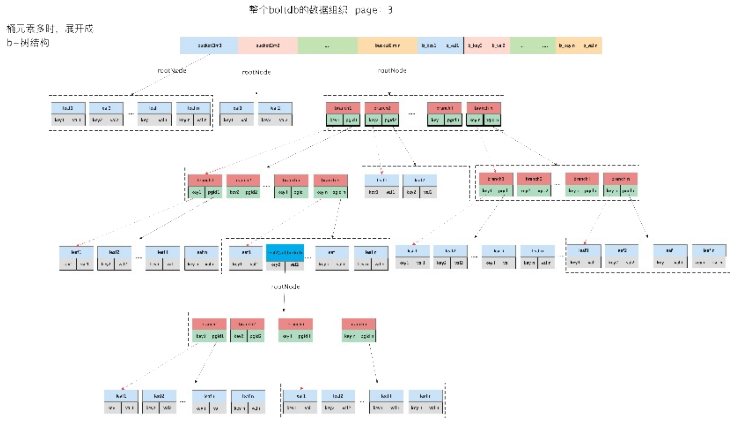
```
13.         //我们的小柜子
14.         bucket, err := tx.CreateBucketIfNotExists([]byte("user"))
15.         if err != nil {
16.             log.Fatalf("CreateBucketIfNotExists err:%s", err.Error())
17.             return err
18.         }
19.         //放入东西
20.         if err = bucket.Put([]byte("hello"), []byte("world")); err != nil {
21.             log.Fatalf("bucket Put err:%s", err.Error())
22.             return err
23.         }
24.         return nil
25.     })
26.     if err != nil {
27.         log.Fatalf("db.Update err:%s", err.Error())
28.     }
29.
30.     // 从db里面读取数据
31.     err = db.View(func(tx *bolt.Tx) error {
32.         //找到柜子
33.         bucket := tx.Bucket([]byte("user"))
34.         //找东西
35.         val := bucket.Get([]byte("hello"))
36.         log.Printf("the get val:%s", val)
37.         val = bucket.Get([]byte("hello2"))
38.         log.Printf("the get val2:%s", val)
39.         return nil
40.     })
41.     if err != nil {
42.         log.Fatalf("db.View err:%s", err.Error())
43.     }
44. }
```

# 第四节 boltdb的整体数据组织结构

下面这幅图完整的展示了boltdb中数据在磁盘文件(file)、文件中的每页(page)上的存储格式以及内存(bucket、node)中b+树形式的组织情况。先从整体上给大家展示一下，大家暂时看不懂不要紧，后面章节会详细的分析每一部分的内容。



# 第五节 boltdb的黑科技

下面从整体上给大家介绍一下，boltdb中比较有特色的几个feature。

### 1. mmap

在boltdb中所有的数据都是以page页为单位组织的，那这时候通常我们的理解是，当通过索引定位到具体存储数据在某一页时，然后就先在页缓存中找，如果页没有缓存，则打开数据库文件中开始读取那一页的数据就好了。 但这样的话性能会极低。boltdb中是通过mmap内存映射技术来解决这个问题。当数据库初始化时，就会进行内存映射，将文件中的数据映射到内存中的一段连续空间，后续再读取某一页的数据时，直接在内存中读取。性能大幅度提升。

### 2. b+树

在boltdb中，索引和数据时按照b+树来组织的。其中一个bucket对象对应一颗b+树，叶子节点存储具体的数据，非叶子节点只存储具体的索引信息，很类似mysql innodb中的主键索引结构。同时值得注意的是所有的bucket也构成了一颗树。但该树不是b+树。

### 3. 嵌套bucket

前面说到，在boltdb中，一个bucket对象是一颗b+树，它上面存储一批kv键值对。但同时它还有一个特性，一个bucket下面还可以有嵌套的subbucket。subbucket中还可以有subbucket。这个特性也很重要。

# 第六节 总结

本章我们首先对boltdb进行一个简要的介绍，让大家知道boltdb是什么，做什么用的。接着又回答了为什么要分析boltdb。在第三节中通过类比现实生活的场景给大家介绍了一下boltdb的简单用法。既然我们用起来了boltdb，那我们就留下了一个悬念，它内部是如何运转的呢？在分析悬念之前，我们在第四节中，从整体对大家展示了boltdb中数据时如何组织存储的。后面的内容都是围绕着这张图来展开的。最后的最后，简单介绍了几个boltdb的特性。

# 第二章 boltdb的核心数据结构分析

从一开始，boltdb的定位就是一款文件数据库，顾名思义它的数据都是存储在磁盘文件上的，目前我们大部分场景使用的磁盘还是机械磁盘。而我们又知道数据落磁盘其实是一个比较慢的操作(此处的快慢是和操作内存想对比而言)。所以怎么样在这种硬件条件无法改变的情况下，如何提升性能就成了一个恒定不变的话题。而提升性能就不得不提到它的数据组织方式了。所以这部分我们主要来分析boltdb的核心数据结构。

我们都知道，操作磁盘之所以慢，是因为对磁盘的读写耗时主要包括：寻道时间+旋转时间+传输时间。而这儿的大头主要是在寻道时间上，因为寻道是需要移动磁头到对应的磁道上，通过马达驱动磁臂移动是一种机械运动，比较耗时。我们往往对磁盘的操作都是随机读写，简而言之，随机读写的话，需要频繁移动磁头到对应的磁道。这种方式性能比较低。还有一种和它对应的方式：顺序读写。顺序读写的性能要比随机读写高很多。

因此，所谓的提升性能，无非就是尽可能的减少磁盘的随机读写，更大程度采用顺序读写的方式。这是主要矛盾，不管是mysql的innodb还是boltdb他们都是围绕这个核心来展开的。如何将用户写进来在内存中的数据尽可能采用顺序写的方式放进磁盘，同时在用户读时，将磁盘中保存的数据以尽可能少的**IO**调用次数加载到内存中，近而返回用户。这里面就涉及到具体的数据在磁盘、内存中的组织结构以及相互转换了。下面我们就对这一块进行详细的分析。

这里面主要包含几块内容：一个是它在磁盘上的数据组织结构page、一个是它在内存中的数据组织结构node、还有一个是page和node之间的相互转换关系。

这里先给大家直观的科普一点：

**set**操作： 本质上对应的是 `set->node->page->file`的过程

**get**操作： 本质上对应的是 `file->page->node->get`的过程

# 第一节 boltdb的物理页page结构

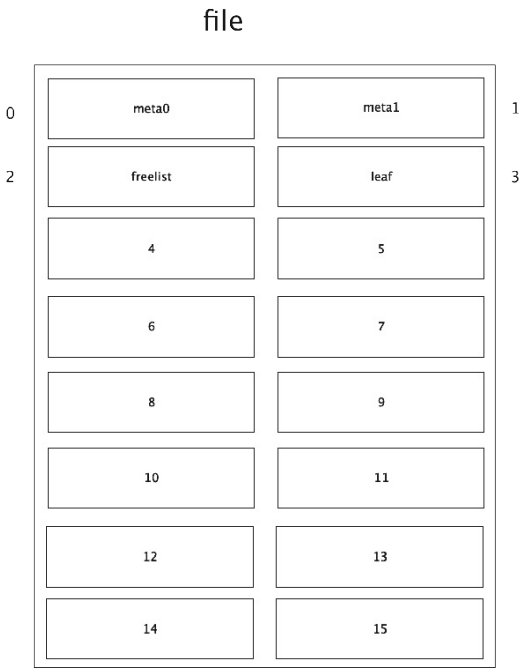
在boltdb中，一个db对应一个真实的磁盘文件。而在具体的文件中，boltdb又是按照以page为单位来读取和写入数据的，也就是说所有的数据在磁盘上都是按照页(page)来存储的，而此处的页大小是保持和操作系统对应的内存页大小一致，也就是4k。

每页由两部分数据组成：页头数据+真实数据，页头信息占16个字节，下面的页的结构定义

```
type pgid uint64

type page struct {
    // 页id 8字节
    id pgid
    // flags：页类型，可以是分支，叶子节点，元信息，空闲列表  2字节，该值的取值详细参见下面描述
    flags uint16
    // 个数 2字节，统计叶子节点、非叶子节点、空闲列表页的个数
    count uint16
    // 4字节，数据是否有溢出，主要在空闲列表上有用
    overflow uint32
    // 真实的数据
    ptr uintptr
}
```

其中，ptr是一个无类型指针，它就是表示每页中真实的存储的数据地址。而其余的几个字段(id、flags、count、overflow)为我们前面提到的页头信息。

下图展现的是boltdb中page的数据存储方式。





flags:

| flag值 | 值含义 |
| --- | --- |
| 0x01 | 非叶子节点页 |
| 0x02 | 叶子节点页 |
| 0x04 | 元数据页 |
| 0x10 | 空闲列表页 |

在boltdb中，它把页划分为四类：

| page页类型 | 类型定义 | 类型值 | 用途 |
| --- | --- | --- | --- |
| 分支节点页 | branchPageFlag | 0x01 | 存储索引信息(页号、元素key值) |
| 叶子节点页 | leafPageFlag | 0x02 | 存储数据信息(页号、插入的key值、插入的value值) |
| 元数据页 | metaPageFlag | 0x04 | 存储数据库的元信息，例如空闲列表页id、放置桶的根页等 |
| 空闲列表页 | freelistPageFlag | 0x10 | 存储哪些页是空闲页，可以用来后续分配空间时，优先考虑分配 |

boltdb通过定义的常量来描述

```
// 页头的大小：16字节
const pageHeaderSize = int(unsafe.Offsetof(((*page)(nil)).ptr))

const minKeysPerPage = 2

//分支节点页中每个元素所占的大小
const branchPageElementSize = int(unsafe.Sizeof(branchPageElement{}))
//叶子节点页中每个元素所占的大小
const leafPageElementSize = int(unsafe.Sizeof(leafPageElement{}))

const (
    branchPageFlag   = 0x01 //分支节点页类型
    leafPageFlag     = 0x02 //叶子节点页类型
    metaPageFlag     = 0x04 //元数据页类型
    freelistPageFlag = 0x10 //空闲列表页类型
```

```
16. )
```

同时每页都有一个方法来判断该页的类型，我们可以清楚的看到每页时通过其flags字段来标识页的类型。

```
1.  // typ returns a human readable page type string used for debugging.
2.  func (p *page) typ() string {
3.      if (p.flags & branchPageFlag) != 0 {
4.          return "branch"
5.      } else if (p.flags & leafPageFlag) != 0 {
6.          return "leaf"
7.      } else if (p.flags & metaPageFlag) != 0 {
8.          return "meta"
9.      } else if (p.flags & freelistPageFlag) != 0 {
10.         return "freelist"
11.     }
12.     return fmt.Sprintf("unknown<%02x>", p.flags)
13. }
```

下面我们一一对其数据结构进行分析。

# 第二节 元数据页

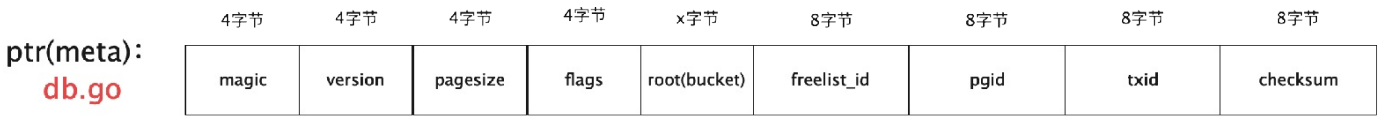
每页有一个meta()方法，如果该页是元数据页的话，可以通过该方法来获取具体的元数据信息。

```
// meta returns a pointer to the metadata section of the page.
func (p *page) meta() *meta {
    // 将p.ptr转为meta信息
    return (*meta)(unsafe.Pointer(&p.ptr))
}
```

详细的元数据信息定义如下：

```
type meta struct {
    magic    uint32 //魔数
    version  uint32 //版本
    pageSize uint32 //page页的大小，该值和操作系统默认的页大小保持一致
    flags    uint32 //保留值，目前貌似还没用到
    root     bucket //所有小柜子bucket的根
    freelist pgid //空闲列表页的id
    pgid     pgid //元数据页的id
    txid     txid //最大的事务id
    checksum uint64 //用作校验的校验和
}
```

下图展现的是元信息存储方式。




下面我们重点关注该meta数据是如何写入到一页中的，以及如何从磁盘中读取meta信息并封装到meta中

### 1. meta->page

```

db.go

// write writes the meta onto a page.
func (m *meta) write(p *page) {
```

```
6.      if m.root.root >= m.pgid {
            panic(fmt.Sprintf("root bucket pgid (%d) above high water mark (%d)",
7.  m.root.root, m.pgid))
8.      } else if m.freelist >= m.pgid {
            panic(fmt.Sprintf("freelist pgid (%d) above high water mark (%d)",
9.  m.freelist, m.pgid))
10.     }
11.
        // Page id is either going to be 0 or 1 which we can determine by the
12. transaction ID.
13.     //指定页id和页类型
14.     p.id = pgid(m.txid % 2)
15.     p.flags |= metaPageFlag
16.
17.     // Calculate the checksum.
18.     m.checksum = m.sum64()
19.   // 这儿p.meta()返回的是p.ptr的地址，因此通过copy之后，meta信息就放到page中了
20.     m.copy(p.meta())
21. }
22.
23.
24. // copy copies one meta object to another.
25. func (m *meta) copy(dest *meta) {
26.     *dest = *m
27. }
28.
29.
30. // generates the checksum for the meta.
31. func (m *meta) sum64() uint64 {
32.     var h = fnv.New64a()
        _, _ = h.Write((*[unsafe.Offsetof(meta{}.checksum)]byte)(unsafe.Pointer(m))
33. [:])
34.     return h.Sum64()
35. }
```

### 2. page->meta

```
1.  page.go
2.
3.  // meta returns a pointer to the metadata section of the page.
4.  func (p *page) meta() *meta {
5.      // 将p.ptr转为meta信息
6.      return (*meta)(unsafe.Pointer(&p.ptr))
```

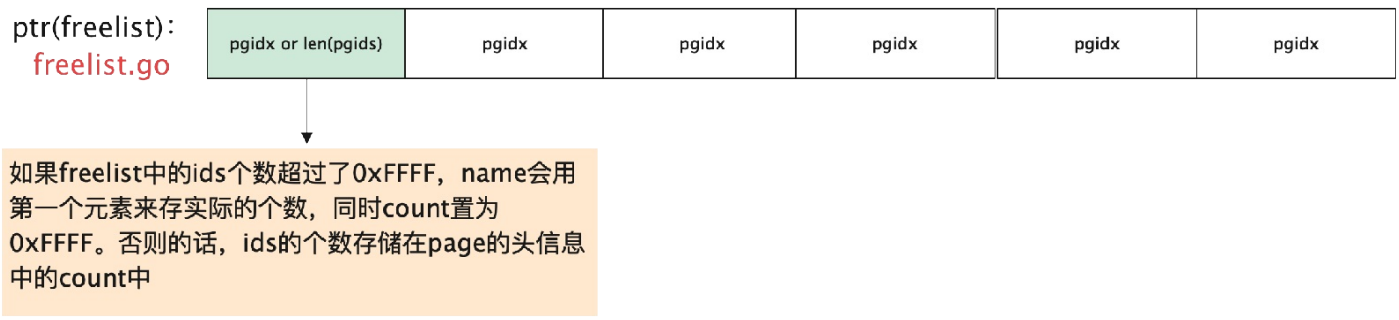
```
7.  }
```

# 第三节 空闲列表页

空闲列表页中主要包含三个部分：所有已经可以重新利用的空闲页列表ids、将来很快被释放掉的事务关联的页列表pending、页id的缓存。详细定义在**freelist.go**文件中，下面给大家展示其空闲页的定义。

```
1.  freelist.go
2.
3.  // freelist represents a list of all pages that are available for allocation.
    // It also tracks pages that have been freed but are still in use by open
4.  transactions.
5.  type freelist struct {
6.      // 已经可以被分配的空闲页
7.      ids     []pgid          // all free and available free page ids.
8.      // 将来很快能被释放的空闲页，部分事务可能在读或者写
9.      pending map[txid][]pgid // mapping of soon-to-be free page ids by tx.
10.     cache   map[pgid]bool   // fast lookup of all free and pending page ids.
11. }
12.
13. // newFreelist returns an empty, initialized freelist.
14. func newFreelist() *freelist {
15.     return &freelist{
16.         pending: make(map[txid][]pgid),
17.         cache:   make(map[pgid]bool),
18.     }
19. }
```

下图展示的是空闲列表的存储方式。




### 1. freelist->page

将空闲列表转换成页信息，写到磁盘中，此处需要注意一个问题，页头中的count字段是一个uint16，占两个字节，其最大可以表示2^16 即65536个数字，当空闲页的个数超过65535时时，需

要将p.ptr中的第一个字节用来存储其空闲页的个数，同时将p.count设置为0xFFFF。否则不超过的情况下，直接用count来表示其空闲页的个数

```
// write writes the page ids onto a freelist page. All free and pending ids are
// saved to disk since in the event of a program crash, all pending ids will
// become free.
//将 freelist信息写入到p中
func (f *freelist) write(p *page) error {
    // Combine the old free pgids and pgids waiting on an open transaction.

    // Update the header flag.
    // 设置页头中的页类型标识
    p.flags |= freelistPageFlag

    // The page.count can only hold up to 64k elements so if we overflow that
    // number then we handle it by putting the size in the first element.

    lenids := f.count()
    if lenids == 0 {
        p.count = uint16(lenids)
    } else if lenids < 0xFFFF {
        p.count = uint16(lenids)
        // 拷贝到page的ptr中
        f.copyall(((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[:])
    } else {
        // 有溢出的情况下，后面第一个元素放置其长度，然后再存放所有的pgid列表
        p.count = 0xFFFF
        ((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[0] = pgid(lenids)
        // 从第一个元素位置拷贝
        f.copyall(((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[1:])
    }
    return nil
}

// copyall copies into dst a list of all free ids and all pending ids in one
sorted list.
// f.count returns the minimum length required for dst.
func (f *freelist) copyall(dst []pgid) {
    // 首先把pending状态的页放到一个数组中，并使其有序
    m := make(pgids, 0, f.pending_count())
    for _, list := range f.pending {
        m = append(m, list...)
```

```
    }
    sort.Sort(m)
    // 合并两个有序的列表，最后结果输出到dst中
    mergepgids(dst, f.ids, m)
}

// mergepgids copies the sorted union of a and b into dst.
// If dst is too small, it panics.
// 将a和b按照有序合并成到dst中，a和b有序
func mergepgids(dst, a, b pgids) {
    if len(dst) < len(a)+len(b) {
        panic(fmt.Errorf("mergepgids bad len %d < %d + %d", len(dst), len(a),
len(b)))
    }
    // Copy in the opposite slice if one is nil.
    if len(a) == 0 {
        copy(dst, b)
        return
    }
    if len(b) == 0 {
        copy(dst, a)
        return
    }

    // Merged will hold all elements from both lists.
    merged := dst[:0]

    // Assign lead to the slice with a lower starting value, follow to the
higher value.
    lead, follow := a, b
    if b[0] < a[0] {
        lead, follow = b, a
    }

    // Continue while there are elements in the lead.
    for len(lead) > 0 {
        // Merge largest prefix of lead that is ahead of follow[0].
        n := sort.Search(len(lead), func(i int) bool { return lead[i] >
follow[0] })
        merged = append(merged, lead[:n]...)
        if n >= len(lead) {
            break
        }
```

```
79.
80.         // Swap lead and follow.
81.         lead, follow = follow, lead[n:]
82.     }
83.
84.     // Append what's left in follow.
85.     _ = append(merged, follow...)
86. }
```

### 2. page->freelist

从磁盘中加载空闲页信息，并转为freelist结构，转换时，也需要注意其空闲页的个数的判断逻辑，当p.count为0xFFFF时，需要读取p.ptr中的第一个字节来计算其空闲页的个数。否则则直接读取p.ptr中存放的数据为空闲页ids列表

```
1.  //从磁盘中的page初始化freelist
2.  // read initializes the freelist from a freelist page.
3.  func (f *freelist) read(p *page) {
4.      // If the page.count is at the max uint16 value (64k) then it's considered
5.      // an overflow and the size of the freelist is stored as the first element.
6.      idx, count := 0, int(p.count)
7.      if count == 0xFFFF {
8.          idx = 1
9.          // 用第一个uint64来存储整个count的值
10.         count = int(((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[0])
11.     }
12.
13.     // Copy the list of page ids from the freelist.
14.     if count == 0 {
15.         f.ids = nil
16.     } else {
17.         ids := ((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[idx:count]
18.         f.ids = make([]pgid, len(ids))
19.         copy(f.ids, ids)
20.
21.         // Make sure they're sorted.
22.         sort.Sort(pgids(f.ids))
23.     }
24.
25.     // Rebuild the page cache.
26.     f.reindex()
27. }
```

```
28.
29. // reindex rebuilds the free cache based on available and pending free lists.
30. func (f *freelist) reindex() {
31.     f.cache = make(map[pgid]bool, len(f.ids))
32.     for _, id := range f.ids {
33.         f.cache[id] = true
34.     }
35.     for _, pendingIDs := range f.pending {
36.         for _, pendingID := range pendingIDs {
37.             f.cache[pendingID] = true
38.         }
39.     }
40. }
```

### 3. allocate

开始分配一段连续的n个页。其中返回值为初始的页id。如果无法分配，则返回0即可

```
    // allocate returns the starting page id of a contiguous list of pages of a
 1. given size.
 2. // If a contiguous block cannot be found then 0 is returned.
 3. // [5,6,7,13,14,15,16,18,19,20,31,32]
 4. // 开始分配一段连续的n个页。其中返回值为初始的页id。如果无法分配，则返回0即可
 5. func (f *freelist) allocate(n int) pgid {
 6.     if len(f.ids) == 0 {
 7.         return 0
 8.     }
 9.
10.     var initial, previd pgid
11.     for i, id := range f.ids {
12.         if id <= 1 {
13.             panic(fmt.Sprintf("invalid page allocation: %d", id))
14.         }
15.
16.         // Reset initial page if this is not contiguous.
17.         // id-previd != 1 来判断是否连续
18.         if previd == 0 || id-previd != 1 {
19.             // 第一次不连续时记录一下第一个位置
20.             initial = id
21.         }
22.
23.         // If we found a contiguous block then remove it and return it.
24.         // 找到了连续的块，然后将其返回即可
```

```
25.         if (id-initial)+1 == pgid(n) {
26.             // If we're allocating off the beginning then take the fast path
27.             // and just adjust the existing slice. This will use extra memory
28.             // temporarily but the append() in free() will realloc the slice
29.             // as is necessary.
30.             if (i + 1) == n {
31.                 // 找到的是前n个连续的空间
32.                 f.ids = f.ids[i+1:]
33.             } else {
34.                 copy(f.ids[i-n+1:], f.ids[i+1:])
35.                 f.ids = f.ids[:len(f.ids)-n]
36.             }
37.
38.             // Remove from the free cache.
39.             // 同时更新缓存
40.             for i := pgid(0); i < pgid(n); i++ {
41.                 delete(f.cache, initial+i)
42.             }
43.
44.             return initial
45.         }
46.
47.         previd = id
48.     }
49.     return 0
50. }
```

> 考虑是否需要补充其他的freelist的方法

# 第四节 分支节点页

分支节点主要用来构建索引，方便提升查询效率。下面我们来看看boltdb的分支节点的数据是如何存储的。

**1.** 分支节点页中元素定义:

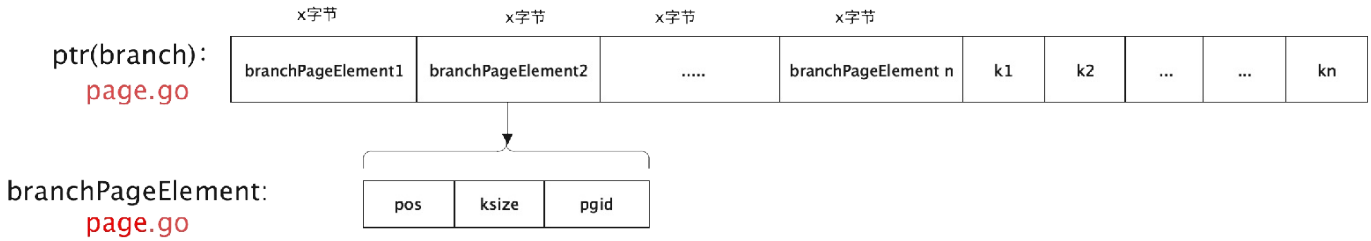
分支节点在存储时，一个分支节点页上会存储多个分支页元素即branchPageElement。这个信息可以记做为分支页元素元信息。元信息中定义了具体该元素的页id(pgid)、该元素所指向的页中存储的最小key的值大小、最小key的值存储的位置距离当前的元信息的偏移量pos。下面是branchPageElement的详细定义:

```
1.  // branchPageElement represents a node on a branch page.
2.  type branchPageElement struct {
3.      pos   uint32 //该元信息和真实key之间的偏移量
4.      ksize uint32
5.      pgid  pgid
6.  }
7.
8.  // key returns a byte slice of the node key.
9.  func (n *branchPageElement) key() []byte {
10.     buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
11.     // pos~ksize
12.     return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos]))[:n.ksize]
13. }
```

**2.** 分支节点页**page**中获取下标为**index**的某一个**element**的信息和获取全部的**elements**信息

```
1.  // branchPageElement retrieves the branch node by index
2.  func (p *page) branchPageElement(index uint16) *branchPageElement {
3.      return &((*[0x7FFFFFF]branchPageElement)(unsafe.Pointer(&p.ptr)))[index]
4.  }
5.
6.  // branchPageElements retrieves a list of branch nodes.
7.  func (p *page) branchPageElements() []branchPageElement {
8.      if p.count == 0 {
9.          return nil
10.     }
11.     return ((*[0x7FFFFFF]branchPageElement)(unsafe.Pointer(&p.ptr)))[:]
12. }
```

下图展现的是非叶子节点存储方式。




在内存中，分支节点页和叶子节点页都是通过**node**来表示，只不过的区别是通过其**node**中的**isLeaf**这个字段来区分。下面和大家分析分支节点页**page**和内存中的**node**的转换关系。

下面在介绍具体的转换关系前，我们介绍一下内存中的分支节点和叶子节点是如何描述的。

```
1.  // node represents an in-memory, deserialized page.
2.  type node struct {
3.      bucket     *Bucket
4.      isLeaf     bool
5.      unbalanced bool
6.      spilled    bool
7.      key        []byte
8.      pgid       pgid
9.      parent     *node
10.     children   nodes
11.     inodes     inodes
12. }
```

在内存中，具体的一个分支节点或者叶子节点都被抽象为一个node对象，其中是分支节点还是叶子节点主要通通过其isLeaf字段来区分。下面对分支节点和叶子节点做两点说明：

1. 对叶子节点而言，其没有children这个信息。同时也没有key信息。isLeaf字段为true，其上存储的key、value都保存在inodes中
2. 对于分支节点而言，其具有key信息，同时children也不一定为空。isLeaf字段为false，同时该节点上的数据保存在inode中。

为了方便大家理解，node和page的转换，下面大概介绍下inode和nodes结构。我们在下一章会详细介绍node。

```
1.
2.  const (
3.      bucketLeafFlag = 0x01
```

```
)


type nodes []*node

func (s nodes) Len() int           { return len(s) }
func (s nodes) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }
func (s nodes) Less(i, j int) bool { return bytes.Compare(s[i].inodes[0].key,
s[j].inodes[0].key) == -1 }

// inode represents an internal node inside of a node.
// It can be used to point to elements in a page or point
// to an element which hasn't been added to a page yet.
type inode struct {
    // 表示是否是子桶叶子节点还是普通叶子节点。如果flags值为1表示子桶叶子节点，否则为普通叶
子节点
    flags uint32
    // 当inode为分支元素时，pgid才有值，为叶子元素时，则没值
    pgid  pgid
    key   []byte
    // 当inode为分支元素时，value为空，为叶子元素时，才有值
    value []byte
}

type inodes []inode
```

### 3. page->node

通过分支节点页来构建node节点

```
// 根据page来初始化node
// read initializes the node from a page.
func (n *node) read(p *page) {
    n.pgid = p.id
    n.isLeaf = ((p.flags & leafPageFlag) != 0)
    n.inodes = make(inodes, int(p.count))

    for i := 0; i < int(p.count); i++ {
        inode := &n.inodes[i]
        if n.isLeaf {
            // 获取第i个叶子节点
            elem := p.leafPageElement(uint16(i))
            inode.flags = elem.flags
```

```
            inode.key = elem.key()
            inode.value = elem.value()
        } else {
            // 树枝节点
            elem := p.branchPageElement(uint16(i))
            inode.pgid = elem.pgid
            inode.key = elem.key()
        }
        _assert(len(inode.key) > 0, "read: zero-length inode key")
    }

    // Save first key so we can find the node in the parent when we spill.
    if len(n.inodes) > 0 {
        // 保存第1个元素的值
        n.key = n.inodes[0].key
        _assert(len(n.key) > 0, "read: zero-length node key")
    } else {
        n.key = nil
    }
}
```

### 4. node->page

将node中的数据写入到page中

```
// write writes the items onto one or more pages.
// 将node转为page
func (n *node) write(p *page) {
    // Initialize page.
    // 判断是否是叶子节点还是非叶子节点
    if n.isLeaf {
        p.flags |= leafPageFlag
    } else {
        p.flags |= branchPageFlag
    }

    // 这儿叶子节点不可能溢出，因为溢出时，会分裂
    if len(n.inodes) >= 0xFFFF {
        panic(fmt.Sprintf("inode overflow: %d (pgid=%d)", len(n.inodes), p.id))
    }
    p.count = uint16(len(n.inodes))

```

```
18.     // Stop here if there are no items to write.
19.     if p.count == 0 {
20.         return
21.     }
22.
23.     // Loop over each item and write it to the page.
24.     // b指向的指针为提逃过所有item头部的位置
        b := (*[maxAllocSize]byte)(unsafe.Pointer(&p.ptr))
25. [n.pageElementSize()*len(n.inodes):]
26.     for i, item := range n.inodes {
27.         _assert(len(item.key) > 0, "write: zero-length inode key")
28.
29.         // Write the page element.
30.         // 写入叶子节点数据
31.         if n.isLeaf {
32.             elem := p.leafPageElement(uint16(i))
                elem.pos = uint32(uintptr(unsafe.Pointer(&b[0])) -
33. uintptr(unsafe.Pointer(elem)))
34.             elem.flags = item.flags
35.             elem.ksize = uint32(len(item.key))
36.             elem.vsize = uint32(len(item.value))
37.         } else {
38.             // 写入分支节点数据
39.             elem := p.branchPageElement(uint16(i))
                elem.pos = uint32(uintptr(unsafe.Pointer(&b[0])) -
40. uintptr(unsafe.Pointer(elem)))
41.             elem.ksize = uint32(len(item.key))
42.             elem.pgid = item.pgid
43.             _assert(elem.pgid != p.id, "write: circular dependency occurred")
44.         }
45.
46.         // If the length of key+value is larger than the max allocation size
47.         // then we need to reallocate the byte array pointer.
48.         //
49.         // See: https://github.com/boltdb/bolt/pull/335
50.         klen, vlen := len(item.key), len(item.value)
51.         if len(b) < klen+vlen {
52.             b = (*[maxAllocSize]byte)(unsafe.Pointer(&b[0]))[:]
53.         }
54.
55.         // Write data for the element to the end of the page.
56.         copy(b[0:], item.key)
57.         b = b[klen:]
```

```
58.         copy(b[0:], item.value)
59.         b = b[vlen:]
60.     }
61.
62.     // DEBUG ONLY: n.dump()
63. }
```

# 第五节 叶子节点页

叶子节点主要用来存储实际的数据，也就是key+value了。下面看看具体的key+value是如何设计的。

在boltdb中，每一对key/value在存储时，都有一份元素元信息，也就是leafPageElement。其中定义了key的长度、value的长度、具体存储的值距离元信息的偏移位置pos。

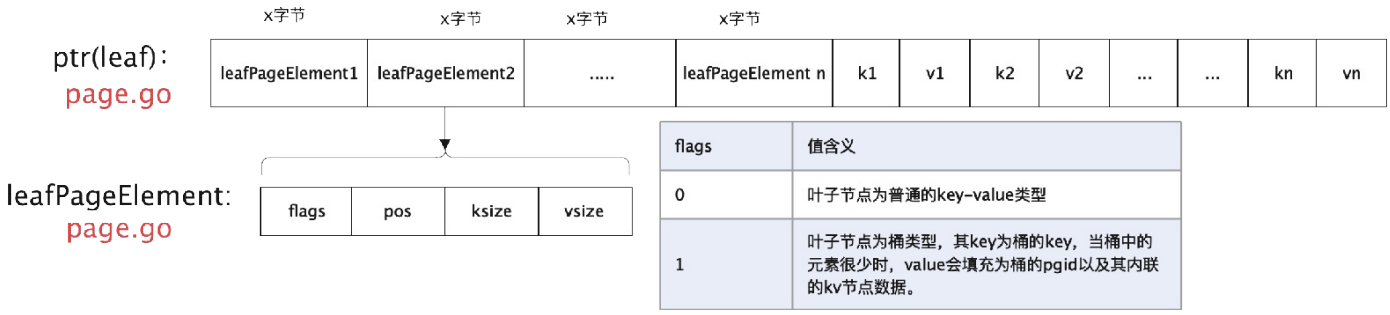
```
1.  // leafPageElement represents a node on a leaf page.
2.  // 叶子节点既存储key，也存储value
3.  type leafPageElement struct {
        flags uint32 //该值主要用来区分，是子桶叶子节点元素还是普通的key/value叶子节点元素。
4.  flags值为1时表示子桶。否则为key/value
5.      pos   uint32
6.      ksize uint32
7.      vsize uint32
8.  }
9.
10. // 叶子节点的key
11. // key returns a byte slice of the node key.
12. func (n *leafPageElement)     key() []byte {
13.     buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
14.     // pos~ksize
15.     return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos]))[:n.ksize:n.ksize]
16. }
17.
18. // 叶子节点的value
19. // value returns a byte slice of the node value.
20. func (n *leafPageElement) value() []byte {
21.     buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
22.     // key:pos~ksize
23.     // value:pos+ksize~pos+ksize+vsize
        return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos+n.ksize]))
24. [:n.vsize:n.vsize]
25. }
```

下面是具体在叶子节点的page中获取下标为index的某个key/value的元信息。根据其元信息，就可以进一步获取其存储的key和value的值了，具体方法可以看上面的key()和value()

```
1.  // leafPageElement retrieves the leaf node by index
2.  func (p *page) leafPageElement(index uint16) *leafPageElement {
```

```
    n := &((*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[index]

    // 最原始的指针：unsafe.Pointer(&p.ptr)
    // 将其转为(*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr))
    // 然后再取第index个元素 ((*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[index]
    // 最后返回该元素指针 &((*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[index]

    // ((*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))
    // (*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr))==>[]leafPageElement
    // &leafElements[index]
    return n
}

// leafPageElements retrieves a list of leaf nodes.
func (p *page) leafPageElements() []leafPageElement {
    if p.count == 0 {
        return nil
    }
    return ((*[0x7FFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[:]
}
```

下图展现的是叶子节点存储方式。




| flags | 值含义 |
| --- | --- |
| 0 | 叶子节点为普通的key-value类型 |
| 1 | 叶子节点为桶类型，其key为桶的key，当桶中的元素很少时，value会填充为桶的pgid以及其内联的kv节点数据。 |

其具体叶子节点页**page**转换成**node**时的转变过程如同分支节点转换的方法一样，此处就不做赘述，可以参考**2.1.3**节介绍的**read()**和**write()**方法

# 第六节 总结

本章中我们重点分析了boltdb中的核心数据结构(page、freelist、meta、node)以及他们之间的相互转化。

在底层磁盘上存储时，boltdb是按照页的单位来存储实际数据的，页的大小取自于它运行的操作系统的页大小。在boltdb中，根据存储的数据的类型不同，将页page整体上分为4大类：

1. 元信息页**(meta page)**

2. 空闲列表页**(freelist page)**

3. 分支节点页**(branch page)**

4. 叶子节点页**(leaf page)**

在page的头信息中通过flags字段来区分。

在内存中同样有对应的结构来映射磁盘上的上述几种页。如元信息**meta**、空闲列表**freelist**、分支/叶子节点**node(**通过**isLeaf**区分分支节点还是叶子节点**)**。我们在每一节中先详细介绍其数据结构的定义。接着再重点分析在内存和磁盘上该类型的页时如何进行转换的。可以准确的说，数据结构属于boltdb核心中的核心。梳理清楚了每个数据结构存储的具体数据和格式后。下一章我们将重点分析其另外两个核心结构bucket和node。

# 第三章 boltdb的b+树(Bucket、node)分析

在第一章我们提到在boltdb中，一个db对应底层的一个磁盘文件。一个db就像一个大柜子一样，其中可以被分隔多个小柜子，用来存储同类型的东西。每个小柜子在boltdb中就是Bucket了。bucket英文为桶。很显然按照字面意思来理解，它在生活中也是存放数据的一种容器。目前为了方便大家理解，在boltdb中的Bucket可以粗略的认为，它里面主要存放的内容就是我们的k/v键值对啦。但这儿其实不准确，后面会详细说明。下面详细进行分析Bucket。在boltdb中定义有bucket、Bucket两个结构。我们在此处所指的Bucket都是指Bucket哈。请大家注意！

# 第一节 boltdb的Bucket结构

先来看官方文档的一段描述Bucket的话。

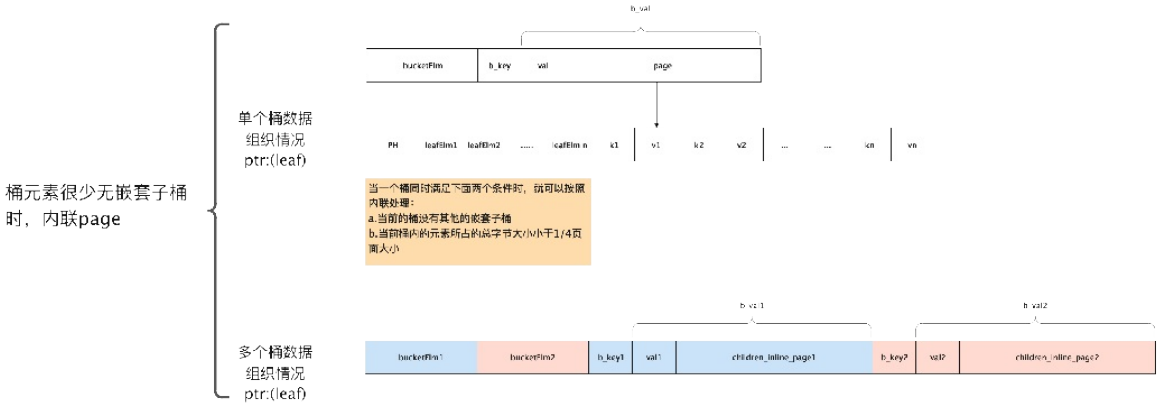
> Bucket represents a collection of key/value pairs inside the database.

下面是Bucket的详细定义，本节我们先暂时忽略事务**Tx**，后面章节会详细介绍事务

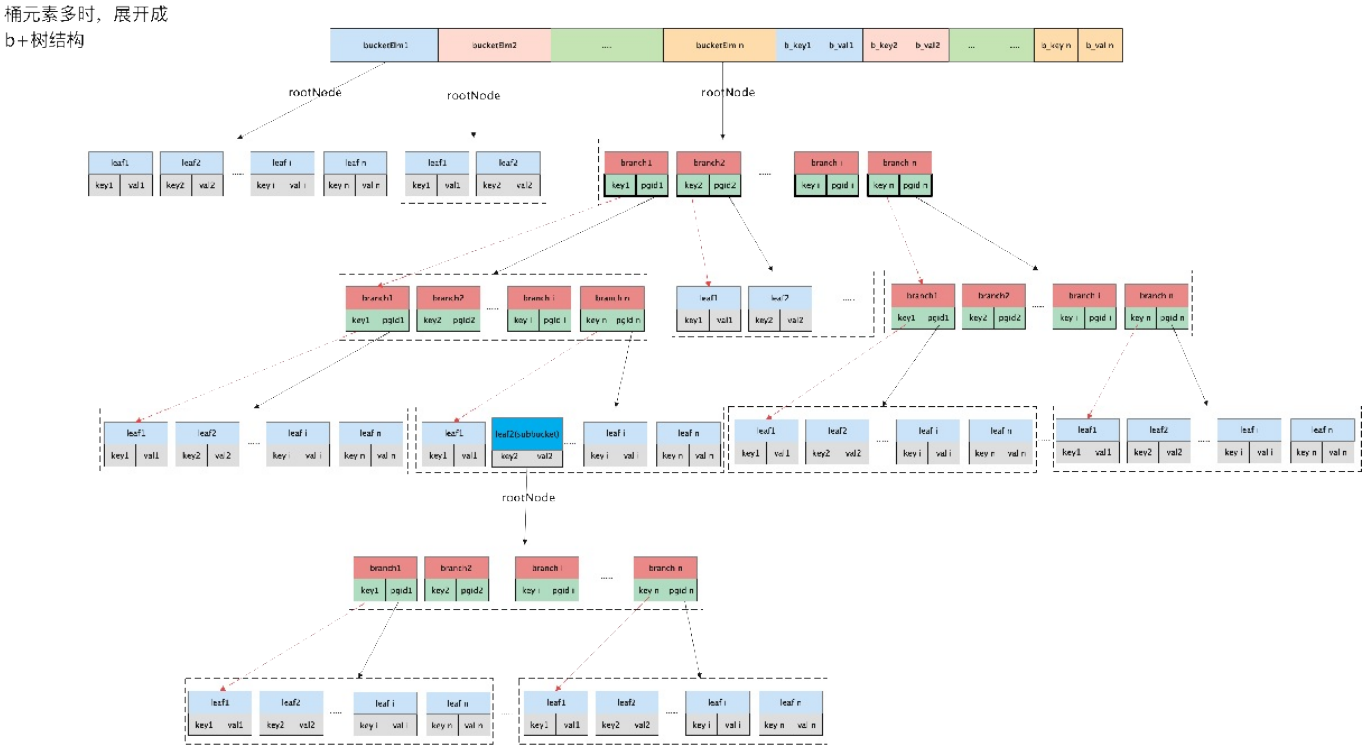
```
// 16 byte
const bucketHeaderSize = int(unsafe.Sizeof(bucket{}))

const (
    minFillPercent = 0.1
    maxFillPercent = 1.0
)

// DefaultFillPercent is the percentage that split pages are filled.
// This value can be changed by setting Bucket.FillPercent.
const DefaultFillPercent = 0.5

// Bucket represents a collection of key/value pairs inside the database.
// 一组key/value的集合，也就是一个b+树
type Bucket struct {
    *bucket //在内联时bucket主要用来存储其桶的value并在后面拼接所有的元素，即所谓的内联
    tx       *Tx                // the associated transaction
    buckets  map[string]*Bucket // subbucket cache
    page     *page              // inline page reference，内联页引用
    rootNode *node              // materialized node for the root page.
    nodes    map[pgid]*node     // node cache

    // Sets the threshold for filling nodes when they split. By default,
    // the bucket will fill to 50% but it can be useful to increase this
    // amount if you know that your write workloads are mostly append-only.
    //
    // This is non-persisted across transactions so it must be set in every Tx.
    // 填充率
    FillPercent float64
}

// bucket represents the on-file representation of a bucket.
```

```
    // This is stored as the "value" of a bucket key. If the bucket is small
33. enough,
34. // then its root page can be stored inline in the "value", after the bucket
35. // header. In the case of inline buckets, the "root" will be 0.
36. type bucket struct {
37.     root     pgid   // page id of the bucket's root-level page
38.     sequence uint64 // monotonically incrementing, used by NextSequence()
39. }
40.
41. // newBucket returns a new bucket associated with a transaction.
42. func newBucket(tx *Tx) Bucket {
43.     var b = Bucket{tx: tx, FillPercent: DefaultFillPercent}
44.     if tx.writable {
45.         b.buckets = make(map[string]*Bucket)
46.         b.nodes = make(map[pgid]*node)
47.     }
48.     return b
49. }
```

下图展现的是数据在bucket中的存储方式。




整个boltdb的数据组织 page：3




上面是一个Bucket的定义，在开始下面的内容前，我们先提前介绍一下另一个角色Cursor，因为后面会频繁的用到它。大家在这里先知道，一个Bucket就是一个b+树就可以了。我们后面会对其进行详细的分析。

# 第二节 Bucket遍历之Cursor

本节我们先做一节内容的铺垫，暂时不讲如何创建、获取、删除一个Bucket。而是介绍一个boltdb中的新对象Cursor。

答案是：所有的上述操作都是建立在首先定位到一个Bucket所属的位置，然后才能对其进行操作。而定位一个Bucket的功能就是由Cursor来完成的。所以我们先这一节给大家介绍一下boltdb中的Cursor。

我们先看下官方文档对Cursor的描述

> Cursor represents an iterator that can traverse over all key/value pairs in a bucket in sorted order.

用大白话讲，既然一个Bucket逻辑上是一颗b+树，那就意味着我们可以对其进行遍历。前面提到的set、get操作，无非是要在Bucket上先找到合适的位置，然后再进行操作。而“找”这个操作就是交由Cursor来完成的。简而言之对Bucket这颗b+树的遍历工作由Cursor来执行。一个Bucket对象关联一个Cursor。下面我们先看看Bucket和Cursor之间的关系。

```
1.  // Cursor creates a cursor associated with the bucket.
2.  // The cursor is only valid as long as the transaction is open.
3.  // Do not use a cursor after the transaction is closed.
4.  func (b *Bucket) Cursor() *Cursor {
5.      // Update transaction statistics.
6.      b.tx.stats.CursorCount++
7.
8.      // Allocate and return a cursor.
9.      return &Cursor{
10.         bucket: b,
11.         stack:  make([]elemRef, 0),
12.     }
13. }
```

## 3.2.1 Cursor结构

从上面可以清楚的看到，在获取一个游标Cursor对象时，会将当前的Bucket对象传进去，同时还初始化了一个栈对象，结合数据结构中学习的树的知识。我们也就知道，它的内部就是对树进行遍历。下面我们详细介绍Cursor这个人物。

```
    // Cursor represents an iterator that can traverse over all key/value pairs in
1.  a bucket in sorted order.
```

```
// Cursors see nested buckets with value == nil.
// Cursors can be obtained from a transaction and are valid as long as the transaction is open.
//
// Keys and values returned from the cursor are only valid for the life of the transaction.
//
// Changing data while traversing with a cursor may cause it to be invalidated
// and return unexpected keys and/or values. You must reposition your cursor
// after mutating data.
type Cursor struct {
    bucket *Bucket
    // 保存遍历搜索的路径
    stack []elemRef
}

// elemRef represents a reference to an element on a given page/node.
type elemRef struct {
    page  *page
    node  *node
    index int
}

// isLeaf returns whether the ref is pointing at a leaf page/node.
func (r *elemRef) isLeaf() bool {
    if r.node != nil {
        return r.node.isLeaf
    }
    return (r.page.flags & leafPageFlag) != 0
}

// count returns the number of inodes or page elements.
func (r *elemRef) count() int {
    if r.node != nil {
        return len(r.node.inodes)
    }
    return int(r.page.count)
}
```

### 3.2.2 Cursor对外接口

下面我们看一下Cursor对外暴露的接口有哪些。看之前也可以心里先想一下。针对一棵树我们需要哪

些遍历接口呢？

主体也就是三类：定位到某一个元素的位置、在当前位置从前往后找、在当前位置从后往前找。

```
    // First moves the cursor to the first item in the bucket and returns its key
1.  and value.
2.  // If the bucket is empty then a nil key and value are returned.
3.  // The returned key and value are only valid for the life of the transaction.
4.  func (c *Cursor) First() (key []byte, value []byte)
5.
    // Last moves the cursor to the last item in the bucket and returns its key and
6.  value.
7.  // If the bucket is empty then a nil key and value are returned.
8.  // The returned key and value are only valid for the life of the transaction.
9.  func (c *Cursor) Last() (key []byte, value []byte)
10.
    // Next moves the cursor to the next item in the bucket and returns its key and
11. value.
    // If the cursor is at the end of the bucket then a nil key and value are
12. returned.
13. // The returned key and value are only valid for the life of the transaction.
14. func (c *Cursor) Next() (key []byte, value []byte)
15.
    // Prev moves the cursor to the previous item in the bucket and returns its key
16. and value.
    // If the cursor is at the beginning of the bucket then a nil key and value are
17. returned.
18. // The returned key and value are only valid for the life of the transaction.
19. func (c *Cursor) Prev() (key []byte, value []byte)
20.
21. // Delete removes the current key/value under the cursor from the bucket.
    // Delete fails if current key/value is a bucket or if the transaction is not
22. writable.
23. func (c *Cursor) Delete() error
24.
25. // Seek moves the cursor to a given key and returns it.
26. // If the key does not exist then the next key is used. If no keys
27. // follow, a nil key is returned.
28. // The returned key and value are only valid for the life of the transaction.
29. func (c *Cursor) Seek(seek []byte) (key []byte, value []byte)
```

下面我们详细分析一下Seek()、First()、Last()、Next()、Prev()、Delete()这三个方法的内部实现。其余的方法我们代码就不贴出来了。大致思路可以梳理一下。

### 3.2.3 Seek(key)实现分析

Seek()方法内部主要调用了seek()私有方法，我们重点关注seek()这个方法的实现，该方法有三个返回值，前两个为key、value、第三个为叶子节点的类型。前面提到在boltdb中，叶子节点元素有两种类型：一种是嵌套的子桶、一种是普通的key/value。而这二者就是通过flags来区分的。如果叶子节点元素为嵌套的子桶时，返回的flags为1，也就是bucketLeafFlag取值。

```
1.  // Seek moves the cursor to a given key and returns it.
2.  // If the key does not exist then the next key is used. If no keys
3.  // follow, a nil key is returned.
4.  // The returned key and value are only valid for the life of the transaction.
5.  func (c *Cursor) Seek(seek []byte) (key []byte, value []byte) {
6.      k, v, flags := c.seek(seek)
7.
        // If we ended up after the last element of a page then move to the next
8.  one.
        // 下面这一段逻辑是必须的，因为在seek()方法中，如果ref.index>ref.count()的话，就直接
9.  返回nil,nil,0了
10.     // 这里需要返回下一个
11.     if ref := &c.stack[len(c.stack)-1]; ref.index >= ref.count() {
12.         k, v, flags = c.next()
13.     }
14.
15.     if k == nil {
16.         return nil, nil
17.         //     子桶的话
18.     } else if (flags & uint32(bucketLeafFlag)) != 0 {
19.         return k, nil
20.     }
21.     return k, v
22. }
23.
24. // seek moves the cursor to a given key and returns it.
25. // If the key does not exist then the next key is used.
26. func (c *Cursor) seek(seek []byte) (key []byte, value []byte, flags uint32) {
27.     _assert(c.bucket.tx.db != nil, "tx closed")
28.
29.     // Start from root page/node and traverse to correct page.
30.     c.stack = c.stack[:0]
31.     // 开始根据seek的key值搜索root
32.     c.search(seek, c.bucket.root)
33.     // 执行完搜索后，stack中保存了所遍历的路径
```

```
34.     ref := &c.stack[len(c.stack)-1]
35.
36.     // If the cursor is pointing to the end of page/node then return nil.
37.     if ref.index >= ref.count() {
38.         return nil, nil, 0
39.     }
40.     //获取值
41.     // If this is a bucket then return a nil value.
42.     return c.keyValue()
43. }
44.
45.
46.
47. // keyValue returns the key and value of the current leaf element.
48. func (c *Cursor) keyValue() ([]byte, []byte, uint32) {
49.   //最后一个节点为叶子节点
50.     ref := &c.stack[len(c.stack)-1]
51.     if ref.count() == 0 || ref.index >= ref.count() {
52.         return nil, nil, 0
53.     }
54.
55.     // Retrieve value from node.
56.     // 先从内存中取
57.     if ref.node != nil {
58.         inode := &ref.node.inodes[ref.index]
59.         return inode.key, inode.value, inode.flags
60.     }
61.
62.     // 其次再从文件page中取
63.     // Or retrieve value from page.
64.     elem := ref.page.leafPageElement(uint16(ref.index))
65.     return elem.key(), elem.value(), elem.flags
66. }
```

seek()中最核心的方法就是调用search()了，search()方法中，传入的就是要搜索的key值和该桶的root节点。search()方法中，其内部是通过递归的层层往下搜索，下面我们详细了解一下search()内部的实现机制。

```
1.  // 从根节点开始遍历
    // search recursively performs a binary search against a given page/node until
2.  it finds a given key.
3.  func (c *Cursor) search(key []byte, pgid pgid) {
```

```
    // root, 3
    // 层层找page, bucket->tx->db->dataref
    p, n := c.bucket.pageNode(pgid)
    if p != nil && (p.flags&(branchPageFlag|leafPageFlag)) == 0 {
        panic(fmt.Sprintf("invalid page type: %d: %x", p.id, p.flags))
    }
    e := elemRef{page: p, node: n}
    //记录遍历过的路径
    c.stack = append(c.stack, e)

    // If we're on a leaf page/node then find the specific node.
    // 如果是叶子节点，找具体的值node
    if e.isLeaf() {
        c.nsearch(key)
        return
    }

    if n != nil {
        // 先搜索node，因为node是加载到内存中的
        c.searchNode(key, n)
        return
    }
    // 其次再在page中搜索
    c.searchPage(key, p)
}

// pageNode returns the in-memory node, if it exists.
// Otherwise returns the underlying page.
func (b *Bucket) pageNode(id pgid) (*page, *node) {
    // Inline buckets have a fake page embedded in their value so treat them
    // differently. We'll return the rootNode (if available) or the fake page.
    // 内联页的话，就直接返回其page了
    if b.root == 0 {
        if id != 0 {
            panic(fmt.Sprintf("inline bucket non-zero page access(2): %d != 0",
id))
        }
        if b.rootNode != nil {
            return nil, b.rootNode
        }
        return b.page, nil
    }
```

```
45.
46.     // Check the node cache for non-inline buckets.
47.     if b.nodes != nil {
48.         if n := b.nodes[id]; n != nil {
49.             return nil, n
50.         }
51.     }
52.
53.     // Finally lookup the page from the transaction if no node is materialized.
54.     return b.tx.page(id), nil
55. }
56.
57.
58. //node中搜索
59. func (c *Cursor) searchNode(key []byte, n *node) {
60.     var exact bool
61.     //二分搜索
62.     index := sort.Search(len(n.inodes), func(i int) bool {
            // TODO(benbjohnson): Optimize this range search. It's a bit hacky
63. right now.
            // sort.Search() finds the lowest index where f() != -1 but we need the
64. highest index.
65.         ret := bytes.Compare(n.inodes[i].key, key)
66.         if ret == 0 {
67.             exact = true
68.         }
69.         return ret != -1
70.     })
71.     if !exact && index > 0 {
72.         index--
73.     }
74.     c.stack[len(c.stack)-1].index = index
75.
76.     // Recursively search to the next page.
77.     c.search(key, n.inodes[index].pgid)
78. }
79.
80. //页中搜索
81. func (c *Cursor) searchPage(key []byte, p *page) {
82.     // Binary search for the correct range.
83.     inodes := p.branchPageElements()
84.
85.     var exact bool
```

```
86.     index := sort.Search(int(p.count), func(i int) bool {
            // TODO(benbjohnson): Optimize this range search. It's a bit hacky
87. right now.
            // sort.Search() finds the lowest index where f() != -1 but we need the
88. highest index.
            ret := bytes.Compare(inodes[i].key(), key)
90.         if ret == 0 {
91.             exact = true
92.         }
93.         return ret != -1
94.     })
95.     if !exact && index > 0 {
96.         index--
97.     }
98.     c.stack[len(c.stack)-1].index = index
99.
100.    // Recursively search to the next page.
101.    c.search(key, inodes[index].pgid)
102. }
103.
104.
105. // nsearch searches the leaf node on the top of the stack for a key.
106. // 搜索叶子页
107. func (c *Cursor) nsearch(key []byte) {
108.     e := &c.stack[len(c.stack)-1]
109.     p, n := e.page, e.node
110.
111.     // If we have a node then search its inodes.
112.     // 先搜索node
113.     if n != nil {
114.         //又是二分搜索
115.         index := sort.Search(len(n.inodes), func(i int) bool {
116.             return bytes.Compare(n.inodes[i].key, key) != -1
117.         })
118.         e.index = index
119.         return
120.     }
121.
122.     // If we have a page then search its leaf elements.
123.     // 再搜索page
124.     inodes := p.leafPageElements()
125.     index := sort.Search(int(p.count), func(i int) bool {
126.         return bytes.Compare(inodes[i].key(), key) != -1
```

```
127.      })
128.      e.index = index
129.  }
```

到这儿我们就已经看完所有的seek()查找一个key的过程了，其内部也很简单，就是从根节点开始，通过不断递归遍历每层节点，采用二分法来定位到具体的叶子节点。到达叶子节点时，其叶子节点内部存储的数据也是有序的，因此继续按照二分查找来找到最终的下标。

值得需要注意点：

在遍历时，我们都知道，有可能遍历到的当前分支节点数据并没有在内存中，此时就需要从**page**中加载数据遍历。所以在遍历过程中，优先在**node**中找，如果**node**为空的时候才会采用**page**来查找。

### 3.2.4 First()、Last()实现分析

前面看了定位到具体某个key的一个过程，现在我们看一下，在定位到第一个元素时，我们知道它一定是位于最左侧的第一个叶子节点的第一个元素。同理，在定位到最后一个元素时，它一定是位于最右侧的第一个叶子节点的最后一个元素。下面是其内部的实现逻辑：

**First()**实现

```
    // First moves the cursor to the first item in the bucket and returns its key
1.  and value.
2.  // If the bucket is empty then a nil key and value are returned.
3.  // The returned key and value are only valid for the life of the transaction.
4.  func (c *Cursor) First() (key []byte, value []byte) {
5.      _assert(c.bucket.tx.db != nil, "tx closed")
6.      // 清空stack
7.      c.stack = c.stack[:0]
8.      p, n := c.bucket.pageNode(c.bucket.root)
9.
10.     // 一直找到第一个叶子节点，此处在天添加stack时，一直让index设置为0即可
11.     ref := elemRef{page: p, node: n, index: 0}
12.     c.stack = append(c.stack, ref)
13.
14.
15.     c.first()
16.
17.     // If we land on an empty page then move to the next value.
18.     // https://github.com/boltdb/bolt/issues/450
19.     // 当前页时空的话，找下一个
20.     if c.stack[len(c.stack)-1].count() == 0 {
21.         c.next()
```

```
    }

    k, v, flags := c.keyValue()
    // 是桶
    if (flags & uint32(bucketLeafFlag)) != 0 {
        return k, nil
    }
    return k, v

}

// first moves the cursor to the first leaf element under the last page in the stack.
// 找到最后一个非叶子节点的第一个叶子节点。index=0的节点
func (c *Cursor) first() {
    for {
        // Exit when we hit a leaf page.
        var ref = &c.stack[len(c.stack)-1]
        if ref.isLeaf() {
            break
        }

        // Keep adding pages pointing to the first element to the stack.
        var pgid pgid
        if ref.node != nil {
            pgid = ref.node.inodes[ref.index].pgid
        } else {
            pgid = ref.page.branchPageElement(uint16(ref.index)).pgid
        }
        p, n := c.bucket.pageNode(pgid)
        c.stack = append(c.stack, elemRef{page: p, node: n, index: 0})
    }
}
```

### Last()实现

```
// Last moves the cursor to the last item in the bucket and returns its key and value.
// If the bucket is empty then a nil key and value are returned.
// The returned key and value are only valid for the life of the transaction.
func (c *Cursor) Last() (key []byte, value []byte) {
    _assert(c.bucket.tx.db != nil, "tx closed")
```

```
6.
7.      c.stack = c.stack[:0]
8.      p, n := c.bucket.pageNode(c.bucket.root)
9.
10.     ref := elemRef{page: p, node: n}
11.     // 设置其index为当前页元素的最后一个
12.     ref.index = ref.count() - 1
13.     c.stack = append(c.stack, ref)
14.
15.     c.last()
16.
17.     k, v, flags := c.keyValue()
18.     if (flags & uint32(bucketLeafFlag)) != 0 {
19.         return k, nil
20.     }
21.     return k, v
22. }
23.
    // last moves the cursor to the last leaf element under the last page in the
24. stack.
25. // 移动到栈中最后一个节点的最后一个叶子节点
26. func (c *Cursor) last() {
27.     for {
28.         // Exit when we hit a leaf page.
29.         ref := &c.stack[len(c.stack)-1]
30.         if ref.isLeaf() {
31.             break
32.         }
33.
34.         // Keep adding pages pointing to the last element in the stack.
35.         var pgid pgid
36.         if ref.node != nil {
37.             pgid = ref.node.inodes[ref.index].pgid
38.         } else {
39.             pgid = ref.page.branchPageElement(uint16(ref.index)).pgid
40.         }
41.         p, n := c.bucket.pageNode(pgid)
42.
43.         var nextRef = elemRef{page: p, node: n}
44.         nextRef.index = nextRef.count() - 1
45.         c.stack = append(c.stack, nextRef)
46.     }
```

```
47. }
```

### 3.2.5 Next()、Prev()实现分析

再此我们从当前位置查找前一个或者下一个时，需要注意一个问题，如果当前节点中元素已经完了，那么此时需要回退到遍历路径的上一个节点。然后再继续查找，下面进行代码分析。

**Next()分析**

```
    // Next moves the cursor to the next item in the bucket and returns its key and
 1. value.
    // If the cursor is at the end of the bucket then a nil key and value are
 2. returned.
 3. // The returned key and value are only valid for the life of the transaction.
 4. func (c *Cursor) Next() (key []byte, value []byte) {
 5.     _assert(c.bucket.tx.db != nil, "tx closed")
 6.     k, v, flags := c.next()
 7.     if (flags & uint32(bucketLeafFlag)) != 0 {
 8.         return k, nil
 9.     }
10.     return k, v
11. }
12.
13. // next moves to the next leaf element and returns the key and value.
    // If the cursor is at the last leaf element then it stays there and returns
14. nil.
15. func (c *Cursor) next() (key []byte, value []byte, flags uint32) {
16.     for {
17.         // Attempt to move over one element until we're successful.
18.         // Move up the stack as we hit the end of each page in our stack.
19.         var i int
20.         for i = len(c.stack) - 1; i >= 0; i-- {
21.             elem := &c.stack[i]
22.             if elem.index < elem.count()-1 {
23.                 // 元素还有时，往后移动一个
24.                 elem.index++
25.                 break
26.             }
27.         }
28.         // 最后的结果elem.index++
29.
30.         // If we've hit the root page then stop and return. This will leave the
31.         // cursor on the last element of the last page.
```

```
32.         // 所有页都遍历完了
33.         if i == -1 {
34.             return nil, nil, 0
35.         }
36.
37.         // Otherwise start from where we left off in the stack and find the
38.         // first element of the first leaf page.
39.         // 剩余的节点里面找，跳过原先遍历过的节点
40.         c.stack = c.stack[:i+1]
41.         // 如果是叶子节点，first()啥都不做，直接退出。返回elem.index+1的数据
42.         // 非叶子节点的话，需要移动到stack中最后一个路径的第一个元素
43.         c.first()
44.
45.         // If this is an empty page then restart and move back up the stack.
46.         // https://github.com/boltdb/bolt/issues/450
47.         if c.stack[len(c.stack)-1].count() == 0 {
48.             continue
49.         }
50.
51.         return c.keyValue()
52.     }
53. }
```

**Prev()实现**

```
    // Prev moves the cursor to the previous item in the bucket and returns its key
1.  and value.
    // If the cursor is at the beginning of the bucket then a nil key and value are
2.  returned.
3.  // The returned key and value are only valid for the life of the transaction.
4.  func (c *Cursor) Prev() (key []byte, value []byte) {
5.      _assert(c.bucket.tx.db != nil, "tx closed")
6.
7.      // Attempt to move back one element until we're successful.
8.      // Move up the stack as we hit the beginning of each page in our stack.
9.      for i := len(c.stack) - 1; i >= 0; i-- {
10.         elem := &c.stack[i]
11.         if elem.index > 0 {
12.             // 往前移动一格
13.             elem.index--
14.             break
15.         }
```

```
        c.stack = c.stack[:i]
    }

    // If we've hit the end then return nil.
    if len(c.stack) == 0 {
        return nil, nil
    }

    // Move down the stack to find the last element of the last leaf under this
branch.
    // 如果当前节点是叶子节点的话，则直接退出了，啥都不做。否则的话移动到新页的最后一个节点
    c.last()
    k, v, flags := c.keyValue()
    if (flags & uint32(bucketLeafFlag)) != 0 {
        return k, nil
    }
    return k, v
}
```

### 3.2.6 Delete()方法分析

Delete()方法中，移动当前位置的元素

```
// Delete removes the current key/value under the cursor from the bucket.
// Delete fails if current key/value is a bucket or if the transaction is not
writable.
func (c *Cursor) Delete() error {
    if c.bucket.tx.db == nil {
        return ErrTxClosed
    } else if !c.bucket.Writable() {
        return ErrTxNotWritable
    }

    key, _, flags := c.keyValue()
    // Return an error if current value is a bucket.
    if (flags & bucketLeafFlag) != 0 {
        return ErrIncompatibleValue
    }
    // 从node中移除，本质上将inode数组进行移动
    c.node().del(key)

    return nil
```

```
19. }
20.
21. // del removes a key from the node.
22. func (n *node) del(key []byte) {
23.     // Find index of key.
        index := sort.Search(len(n.inodes), func(i int) bool { return
24. bytes.Compare(n.inodes[i].key, key) != -1 })
25.
26.     // Exit if the key isn't found.
27.     if index >= len(n.inodes) || !bytes.Equal(n.inodes[index].key, key) {
28.         return
29.     }
30.
31.     // Delete inode from the node.
32.     n.inodes = append(n.inodes[:index], n.inodes[index+1:]...)
33.
34.     // Mark the node as needing rebalancing.
35.     n.unbalanced = true
36. }
```

# 第三节 node节点的相关操作

在开始分析node节点之前，我们先看一下官方对node节点的描述

> node represents an in-memory, deserialized page

一个node节点，既可能是叶子节点，也可能是根节点，也可能是分支节点。是物理磁盘上读取进来的页page的内存表现形式。

## 3.3.1 node节点的定义

```
1.  // node represents an in-memory, deserialized page.
2.  type node struct {
3.      bucket     *Bucket // 关联一个桶
4.      isLeaf     bool
5.      unbalanced bool   // 值为true的话，需要考虑页合并
6.      spilled    bool   // 值为true的话，需要考虑页分裂
7.      key        []byte // 对于分支节点的话，保留的是最小的key
8.      pgid       pgid   // 分支节点关联的页id
9.      parent     *node  // 该节点的parent
10.     children   nodes  // 该节点的孩子节点
11.     inodes     inodes // 该节点上保存的索引数据
12. }
13.
14. // inode represents an internal node inside of a node.
15. // It can be used to point to elements in a page or point
16. // to an element which hasn't been added to a page yet.
17. type inode struct {
        // 表示是否是子桶叶子节点还是普通叶子节点。如果flags值为1表示子桶叶子节点，否则为普通叶
    子节点
18.     flags uint32
19.     // 当inode为分支元素时，pgid才有值，为叶子元素时，则没值
20.     pgid pgid
21.     key  []byte
22.     // 当inode为分支元素时，value为空，为叶子元素时，才有值
23.     value []byte
24. }
25.
26. type inodes []inode
```

## 3.3.2 node节点和page转换

在node对象上有两个方法，read(page)、write(page)，其中read(page)方法是用来通过page构建一个node节点；而write(page)方法则是将当前的node节点写入到page中，我们在前面他提到了node节点和page节点的相互转换，此处为了保证内容完整性，我们还是再补充下，同时也给大家加深下影响，展示下同样的数据在磁盘上如何组织的，在内存中又是如何组织的。

**node->page**

```
// write writes the items onto one or more pages.
// 将node转为page
func (n *node) write(p *page) {
    // Initialize page.
    // 判断是否是叶子节点还是非叶子节点
    if n.isLeaf {
        p.flags |= leafPageFlag
    } else {
        p.flags |= branchPageFlag
    }

    // 这儿叶子节点不可能溢出，因为溢出时，会分裂
    if len(n.inodes) >= 0xFFFF {
        panic(fmt.Sprintf("inode overflow: %d (pgid=%d)", len(n.inodes), p.id))
    }
    p.count = uint16(len(n.inodes))

    // Stop here if there are no items to write.
    if p.count == 0 {
        return
    }

    // Loop over each item and write it to the page.
    // b指向的指针为提逃过所有item头部的位置
    b := (*[maxAllocSize]byte)(unsafe.Pointer(&p.ptr))
[n.pageElementSize()*len(n.inodes):]
    for i, item := range n.inodes {
        _assert(len(item.key) > 0, "write: zero-length inode key")

        // Write the page element.
        // 写入叶子节点数据
        if n.isLeaf {
            elem := p.leafPageElement(uint16(i))
            elem.pos = uint32(uintptr(unsafe.Pointer(&b[0])) -
uintptr(unsafe.Pointer(elem)))
```

```
            elem.flags = item.flags
            elem.ksize = uint32(len(item.key))
            elem.vsize = uint32(len(item.value))
        } else {
            // 写入分支节点数据
            elem := p.branchPageElement(uint16(i))
            elem.pos = uint32(uintptr(unsafe.Pointer(&b[0])) -
uintptr(unsafe.Pointer(elem)))
            elem.ksize = uint32(len(item.key))
            elem.pgid = item.pgid
            _assert(elem.pgid != p.id, "write: circular dependency occurred")
        }

        // If the length of key+value is larger than the max allocation size
        // then we need to reallocate the byte array pointer.
        //
        // See: https://github.com/boltdb/bolt/pull/335
        klen, vlen := len(item.key), len(item.value)
        if len(b) < klen+vlen {
            b = (*[maxAllocSize]byte)(unsafe.Pointer(&b[0]))[:]
        }

        // Write data for the element to the end of the page.
        copy(b[0:], item.key)
        b = b[klen:]
        copy(b[0:], item.value)
        b = b[vlen:]
    }

    // DEBUG ONLY: n.dump()
}
```

**page->node**

```
// 根据page来初始化node
// read initializes the node from a page.
func (n *node) read(p *page) {
    n.pgid = p.id
    n.isLeaf = ((p.flags & leafPageFlag) != 0)
    // 一个inodes对应一个xxxPageElement对象
    n.inodes = make(inodes, int(p.count))

```

```
    for i := 0; i < int(p.count); i++ {
        inode := &n.inodes[i]
        if n.isLeaf {
            // 获取第i个叶子节点
            elem := p.leafPageElement(uint16(i))
            inode.flags = elem.flags
            inode.key = elem.key()
            inode.value = elem.value()
        } else {
            // 树枝节点
            elem := p.branchPageElement(uint16(i))
            inode.pgid = elem.pgid
            inode.key = elem.key()
        }
        _assert(len(inode.key) > 0, "read: zero-length inode key")
    }

    // Save first key so we can find the node in the parent when we spill.
    if len(n.inodes) > 0 {
        // 保存第1个元素的值
        n.key = n.inodes[0].key
        _assert(len(n.key) > 0, "read: zero-length node key")
    } else {
        n.key = nil
    }
}
```

## 3.3.3 node节点的增删改查

**put(k,v)**

```
// put inserts a key/value.
// 如果put的是一个key、value的话，不需要指定pgid。
// 如果put的一个树枝节点，则需要指定pgid，不需要指定value
func (n *node) put(oldKey, newKey, value []byte, pgid pgid, flags uint32) {
    if pgid >= n.bucket.tx.meta.pgid {
        panic(fmt.Sprintf("pgid (%d) above high water mark (%d)", pgid,
n.bucket.tx.meta.pgid))
    } else if len(oldKey) <= 0 {
        panic("put: zero-length old key")
    } else if len(newKey) <= 0 {
        panic("put: zero-length new key")
```

```
    }

    // Find insertion index.
    index := sort.Search(len(n.inodes), func(i int) bool { return
bytes.Compare(n.inodes[i].key, oldKey) != -1 })

    // Add capacity and shift nodes if we don't have an exact match and need to
insert.
    exact := (len(n.inodes) > 0 && index < len(n.inodes) &&
bytes.Equal(n.inodes[index].key, oldKey))
    if !exact {
        n.inodes = append(n.inodes, inode{})
        copy(n.inodes[index+1:], n.inodes[index:])
    }

    inode := &n.inodes[index]
    inode.flags = flags
    inode.key = newKey
    inode.value = value
    inode.pgid = pgid
    _assert(len(inode.key) > 0, "put: zero-length inode key")
}
```

**get(k)**

在node中，没有get(k)的方法，其本质是在Cursor中就返回了get的数据。大家可以看看Cursor中的keyValue()方法。

**del(k)**

```
// del removes a key from the node.
func (n *node) del(key []byte) {
    // Find index of key.
    index := sort.Search(len(n.inodes), func(i int) bool { return
bytes.Compare(n.inodes[i].key, key) != -1 })

    // Exit if the key isn't found.
    if index >= len(n.inodes) || !bytes.Equal(n.inodes[index].key, key) {
        return
    }

    // Delete inode from the node.
    n.inodes = append(n.inodes[:index], n.inodes[index+1:]...)
```

```
13.
14.     // Mark the node as needing rebalancing.
15.     n.unbalanced = true
16. }
```

### nextSibling()、prevSibling()

```
 1.
 2. // nextSibling returns the next node with the same parent.
 3. // 返回下一个兄弟节点
 4. func (n *node) nextSibling() *node {
 5.     if n.parent == nil {
 6.         return nil
 7.     }
 8.     index := n.parent.childIndex(n)
 9.     if index >= n.parent.numChildren()-1 {
10.         return nil
11.     }
12.     return n.parent.childAt(index + 1)
13. }
14.
15. // prevSibling returns the previous node with the same parent.
16. // 返回上一个兄弟节点
17. func (n *node) prevSibling() *node {
18.     if n.parent == nil {
19.         return nil
20.     }
21.     // 首先找下标
22.     index := n.parent.childIndex(n)
23.     if index == 0 {
24.         return nil
25.     }
26.     // 然后返回
27.     return n.parent.childAt(index - 1)
28. }
29.
30. // childIndex returns the index of a given child node.
31. func (n *node) childIndex(child *node) int {
        index := sort.Search(len(n.inodes), func(i int) bool { return
32. bytes.Compare(n.inodes[i].key, child.key) != -1 })
33.     return index
34. }
```

```
35.
36.
37. // childAt returns the child node at a given index.
38. // 只有树枝节点才有孩子
39. func (n *node) childAt(index int) *node {
40.     if n.isLeaf {
41.         panic(fmt.Sprintf("invalid childAt(%d) on a leaf node", index))
42.     }
43.     return n.bucket.node(n.inodes[index].pgid, n)
44. }
45.
46. // node creates a node from a page and associates it with a given parent.
47. // 根据pgid创建一个node
48. func (b *Bucket) node(pgid pgid, parent *node) *node {
49.     _assert(b.nodes != nil, "nodes map expected")
50.
51.     // Retrieve node if it's already been created.
52.     if n := b.nodes[pgid]; n != nil {
53.         return n
54.     }
55.
56.     // Otherwise create a node and cache it.
57.     n := &node{bucket: b, parent: parent}
58.     if parent == nil {
59.         b.rootNode = n
60.     } else {
61.         parent.children = append(parent.children, n)
62.     }
63.
64.     // Use the inline page if this is an inline bucket.
65.     // 如果第二次进来，b.page不为空
66.     // 此处的pgid和b.page只会有一个是有值的。
67.     var p = b.page
68.     // 说明不是内联桶
69.     if p == nil {
70.         p = b.tx.page(pgid)
71.     }
72.
73.     // Read the page into the node and cache it.
74.     n.read(p)
75.     // 缓存
76.     b.nodes[pgid] = n
```

```
77.
78.     // Update statistics.
79.     b.tx.stats.NodeCount++
80.
81.     return n
82. }
```

### 3.3.4 node节点的分裂和合并

上面我们看了对node节点的操作，包括put和del方法。经过这些操作后，可能会导致当前的page填充度过高或者过低。因此就引出了node节点的分裂和合并。下面简单介绍下什么是分裂和合并。

分裂：当一个node中的数据过多时，最简单就是当超过了page的填充度时，就需要将当前的node拆分成两个，也就是底层会将一页数据拆分存放到两页中。

合并：当删除了一个或者一批对象时，此时可能会导致一页数据的填充度过低，此时空间可能会浪费比较多。所以就需要考虑对页之间进行数据合并。

有了大概的了解，下面我们就看一下对一个node分裂和合并的实现过程。

**分裂spill()**

> spill writes the nodes to dirty pages and splits nodes as it goes. Returns an error if dirty pages cannot be allocated.

```
1.  // spill writes the nodes to dirty pages and splits nodes as it goes.
2.  // Returns an error if dirty pages cannot be allocated.
3.  func (n *node) spill() error {
4.      var tx = n.bucket.tx
5.      if n.spilled {
6.          return nil
7.      }
8.
9.      // Spill child nodes first. Child nodes can materialize sibling nodes in
10.     // the case of split-merge so we cannot use a range loop. We have to check
11.     // the children size on every loop iteration.
12.     sort.Sort(n.children)
13.     for i := 0; i < len(n.children); i++ {
14.         if err := n.children[i].spill(); err != nil {
15.             return err
16.         }
17.     }
18.
```

```
        // We no longer need the child list because it's only used for spill
19. tracking.
20.     n.children = nil
21.
22.     // Split nodes into appropriate sizes. The first node will always be n.
23.     // 将当前的node进行拆分成多个node
24.     var nodes = n.split(tx.db.pageSize)
25.     for _, node := range nodes {
26.         // Add node's page to the freelist if it's not new.
27.         if node.pgid > 0 {
28.             tx.db.freelist.free(tx.meta.txid, tx.page(node.pgid))
29.             node.pgid = 0
30.         }
31.
32.         // Allocate contiguous space for the node.
33.         p, err := tx.allocate((node.size() / tx.db.pageSize) + 1)
34.         if err != nil {
35.             return err
36.         }
37.
38.         // Write the node.
39.         if p.id >= tx.meta.pgid {
40.             // 不可能发生
                panic(fmt.Sprintf("pgid (%d) above high water mark (%d)", p.id,
41. tx.meta.pgid))
42.         }
43.         node.pgid = p.id
44.         node.write(p)
45.         // 已经拆分过了
46.         node.spilled = true
47.
48.         // Insert into parent inodes.
49.         if node.parent != nil {
50.             var key = node.key
51.             if key == nil {
52.                 key = node.inodes[0].key
53.             }
54.
55.             // 放入父亲节点中
56.             node.parent.put(key, node.inodes[0].key, nil, node.pgid, 0)
57.             node.key = node.inodes[0].key
58.             _assert(len(node.key) > 0, "spill: zero-length node key")
59.         }
```

```
60.
61.         // Update the statistics.
62.         tx.stats.Spill++
63.     }
64.
65.     // If the root node split and created a new root then we need to spill that
        // as well. We'll clear out the children to make sure it doesn't try to
66. respill.
67.     if n.parent != nil && n.parent.pgid == 0 {
68.         n.children = nil
69.         return n.parent.spill()
70.     }
71.
72.     return nil
73. }
74.
75. // split breaks up a node into multiple smaller nodes, if appropriate.
76. // This should only be called from the spill() function.
77. func (n *node) split(pageSize int) []*node {
78.     var nodes []*node
79.
80.     node := n
81.     for {
82.         // Split node into two.
83.         a, b := node.splitTwo(pageSize)
84.         nodes = append(nodes, a)
85.
86.         // If we can't split then exit the loop.
87.         if b == nil {
88.             break
89.         }
90.
91.         // Set node to b so it gets split on the next iteration.
92.         node = b
93.     }
94.
95.     return nodes
96. }
97.
98. // splitTwo breaks up a node into two smaller nodes, if appropriate.
99. // This should only be called from the split() function.
100. func (n *node) splitTwo(pageSize int) (*node, *node) {
```

```
101.      // Ignore the split if the page doesn't have at least enough nodes for
102.      // two pages or if the nodes can fit in a single page.
103.      // 太小的话，就不拆分了
104.      if len(n.inodes) <= (minKeysPerPage*2) || n.sizeLessThan(pageSize) {
105.          return n, nil
106.      }
107.
108.      // Determine the threshold before starting a new node.
109.      var fillPercent = n.bucket.FillPercent
110.      if fillPercent < minFillPercent {
111.          fillPercent = minFillPercent
112.      } else if fillPercent > maxFillPercent {
113.          fillPercent = maxFillPercent
114.      }
115.      threshold := int(float64(pageSize) * fillPercent)
116.
117.      // Determine split position and sizes of the two pages.
118.      splitIndex, _ := n.splitIndex(threshold)
119.
120.      // Split node into two separate nodes.
121.      // If there's no parent then we'll need to create one.
122.      if n.parent == nil {
123.          n.parent = &node{bucket: n.bucket, children: []*node{n}}
124.      }
125.
126.      // Create a new node and add it to the parent.
127.      // 拆分出一个新节点
128.      next := &node{bucket: n.bucket, isLeaf: n.isLeaf, parent: n.parent}
129.      n.parent.children = append(n.parent.children, next)
130.
131.      // Split inodes across two nodes.
132.      next.inodes = n.inodes[splitIndex:]
133.      n.inodes = n.inodes[:splitIndex]
134.
135.      // Update the statistics.
136.      n.bucket.tx.stats.Split++
137.
138.      return n, next
139.  }
140.
141.  // splitIndex finds the position where a page will fill a given threshold.
142.  // It returns the index as well as the size of the first page.
```

```
143.  // This is only be called from split().
144.  // 找到合适的index
145.  func (n *node) splitIndex(threshold int) (index, sz int) {
146.      sz = pageHeaderSize
147.
          // Loop until we only have the minimum number of keys required for the
148.  second page.
149.      for i := 0; i < len(n.inodes)-minKeysPerPage; i++ {
150.          index = i
151.          inode := n.inodes[i]
152.          elsize := n.pageElementSize() + len(inode.key) + len(inode.value)
153.
154.          // If we have at least the minimum number of keys and adding another
155.          // node would put us over the threshold then exit and return.
156.          if i >= minKeysPerPage && sz+elsize > threshold {
157.              break
158.          }
159.
160.          // Add the element size to the total size.
161.          sz += elsize
162.      }
163.
164.      return
165.  }
```

### 合并rebalance()

> rebalance attempts to combine the node with sibling nodes if the node fill size is below a threshold or if there are not enough keys.

页合并有点复杂，虽然能看懂，但要自己写感觉还是挺难写出bug free的

```
1.  // rebalance attempts to combine the node with sibling nodes if the node fill
2.  // size is below a threshold or if there are not enough keys.
3.  // 填充率太低或者没有足够的key时，进行页合并
4.  func (n *node) rebalance() {
5.      if !n.unbalanced {
6.          return
7.      }
8.      n.unbalanced = false
9.
10.     // Update statistics.
11.     n.bucket.tx.stats.Rebalance++
```

```
12.
13.     // Ignore if node is above threshold (25%) and has enough keys.
14.     var threshold = n.bucket.tx.db.pageSize / 4
15.     if n.size() > threshold && len(n.inodes) > n.minKeys() {
16.         return
17.     }
18.
19.     // Root node has special handling.
20.     if n.parent == nil {
21.         // If root node is a branch and only has one node then collapse it.
22.         if !n.isLeaf && len(n.inodes) == 1 {
23.             // Move root's child up.
24.             child := n.bucket.node(n.inodes[0].pgid, n)
25.             n.isLeaf = child.isLeaf
26.             n.inodes = child.inodes[:]
27.             n.children = child.children
28.
29.             // Reparent all child nodes being moved.
30.             for _, inode := range n.inodes {
31.                 if child, ok := n.bucket.nodes[inode.pgid]; ok {
32.                     child.parent = n
33.                 }
34.             }
35.
36.             // Remove old child.
37.             child.parent = nil
38.             delete(n.bucket.nodes, child.pgid)
39.             child.free()
40.         }
41.
42.         return
43.     }
44.
45.     // If node has no keys then just remove it.
46.     if n.numChildren() == 0 {
47.         n.parent.del(n.key)
48.         n.parent.removeChild(n)
49.         delete(n.bucket.nodes, n.pgid)
50.         n.free()
51.         n.parent.rebalance()
52.         return
53.     }
```

```
54.
55.     _assert(n.parent.numChildren() > 1, "parent must have at least 2 children")
56.
57.     // Destination node is right sibling if idx == 0, otherwise left sibling.
58.     var target *node
        // 判断当前node是否是parent的第一个孩子节点，是的话，就要找它的下一个兄弟节点，否则的
59. 话，就找上一个兄弟节点
60.     var useNextSibling = (n.parent.childIndex(n) == 0)
61.     if useNextSibling {
62.         target = n.nextSibling()
63.     } else {
64.         target = n.prevSibling()
65.     }
66.
67.     // If both this node and the target node are too small then merge them.
68.     // 合并当前node和target，target合到node
69.     if useNextSibling {
70.         // Reparent all child nodes being moved.
71.         for _, inode := range target.inodes {
72.             if child, ok := n.bucket.nodes[inode.pgid]; ok {
73.                 // 之前的父亲移除该孩子
74.                 child.parent.removeChild(child)
75.                 // 重新指定父亲节点
76.                 child.parent = n
77.                 // 父亲节点指当前孩子
78.                 child.parent.children = append(child.parent.children, child)
79.             }
80.         }
81.
82.         // Copy over inodes from target and remove target.
83.         n.inodes = append(n.inodes, target.inodes...)
84.         n.parent.del(target.key)
85.         n.parent.removeChild(target)
86.         delete(n.bucket.nodes, target.pgid)
87.         target.free()
88.     } else {
89.         // node合到target
90.         // Reparent all child nodes being moved.
91.         for _, inode := range n.inodes {
92.             if child, ok := n.bucket.nodes[inode.pgid]; ok {
93.                 child.parent.removeChild(child)
94.                 child.parent = target
```

```
                child.parent.children = append(child.parent.children, child)
            }
        }

        // Copy over inodes to target and remove node.
        target.inodes = append(target.inodes, n.inodes...)
        n.parent.del(n.key)
        n.parent.removeChild(n)
        delete(n.bucket.nodes, n.pgid)
        n.free()
    }

    // Either this node or the target node was deleted from the parent so rebalance it.
    n.parent.rebalance()
}
```

# 第四节 Bucket的相关操作

前面我们分析完了如何遍历、查找一个Bucket之后，下面我们来看看如何创建、获取、删除一个Bucket对象。

## 3.4.1 创建一个Bucket

### 1. CreateBucketIfNotExists()、CreateBucket()分析

根据指定的key来创建一个Bucket,如果指定key的Bucket已经存在，则会报错。如果指定的key之前有插入过元素，也会报错。否则的话，会在当前的Bucket中找到合适的位置，然后新建一个Bucket插入进去，最后返回给客户端。

```
1.
    // CreateBucketIfNotExists creates a new bucket if it doesn't already exist and
2.  returns a reference to it.
    // Returns an error if the bucket name is blank, or if the bucket name is too
3.  long.
4.  // The bucket instance is only valid for the lifetime of the transaction.
5.  func (b *Bucket) CreateBucketIfNotExists(key []byte) (*Bucket, error) {
6.      child, err := b.CreateBucket(key)
7.      if err == ErrBucketExists {
8.          return b.Bucket(key), nil
9.      } else if err != nil {
10.         return nil, err
11.     }
12.     return child, nil
13. }
14.
    // CreateBucket creates a new bucket at the given key and returns the new
15. bucket.
    // Returns an error if the key already exists, if the bucket name is blank, or
16. if the bucket name is too long.
17. // The bucket instance is only valid for the lifetime of the transaction.
18. func (b *Bucket) CreateBucket(key []byte) (*Bucket, error) {
19.     if b.tx.db == nil {
20.         return nil, ErrTxClosed
21.     } else if !b.tx.writable {
22.         return nil, ErrTxNotWritable
23.     } else if len(key) == 0 {
24.         return nil, ErrBucketNameRequired
25.     }
```

```
26.
27.     // Move cursor to correct position.
28.     // 拿到游标
29.     c := b.Cursor()
30.     // 开始遍历、找到合适的位置
31.     k, _, flags := c.seek(key)
32.
33.     // Return an error if there is an existing key.
34.     if bytes.Equal(key, k) {
35.         // 是桶,已经存在了
36.         if (flags & bucketLeafFlag) != 0 {
37.             return nil, ErrBucketExists
38.         }
39.         // 不是桶、但key已经存在了
40.         return nil, ErrIncompatibleValue
41.     }
42.
43.     // Create empty, inline bucket.
44.     var bucket = Bucket{
45.         bucket:      &bucket{},
46.         rootNode:    &node{isLeaf: true},
47.         FillPercent: DefaultFillPercent,
48.     }
49.     // 拿到bucket对应的value
50.     var value = bucket.write()
51.
52.     // Insert into node.
53.     key = cloneBytes(key)
54.     // 插入到inode中
55.     // c.node()方法会在内存中建立这棵树，调用n.read(page)
56.     c.node().put(key, key, value, 0, bucketLeafFlag)
57.
58.     // Since subbuckets are not allowed on inline buckets, we need to
59.     // dereference the inline page, if it exists. This will cause the bucket
60.     // to be treated as a regular, non-inline bucket for the rest of the tx.
61.     b.page = nil
62.
63.     //根据key获取一个桶
64.     return b.Bucket(key), nil
65. }
66.
67. // write allocates and writes a bucket to a byte slice.
```

```
68. // 内联桶的话，其value中bucketHeaderSize后面的内容为其page的数据
69. func (b *Bucket) write() []byte {
70.     // Allocate the appropriate size.
71.     var n = b.rootNode
72.     var value = make([]byte, bucketHeaderSize+n.size())
73. 
74.     // Write a bucket header.
75.     var bucket = (*bucket)(unsafe.Pointer(&value[0]))
76.     *bucket = *b.bucket
77. 
78.     // Convert byte slice to a fake page and write the root node.
79.     var p = (*page)(unsafe.Pointer(&value[bucketHeaderSize]))
80.     // 将该桶中的元素压缩存储，放在value中
81.     n.write(p)
82. 
83.     return value
84. }
85. 
86. 
87. // node returns the node that the cursor is currently positioned on.
88. func (c *Cursor) node() *node {
        _assert(len(c.stack) > 0, "accessing a node with a zero-length cursor
89. stack")
90. 
91.     // If the top of the stack is a leaf node then just return it.
92.     if ref := &c.stack[len(c.stack)-1]; ref.node != nil && ref.isLeaf() {
93.         return ref.node
94.     }
95. 
96.     // Start from root and traverse down the hierarchy.
97.     var n = c.stack[0].node
98.     if n == nil {
99.         n = c.bucket.node(c.stack[0].page.id, nil)
100.    }
101.    // 非叶子节点
102.    for _, ref := range c.stack[:len(c.stack)-1] {
103.        _assert(!n.isLeaf, "expected branch node")
104.        n = n.childAt(int(ref.index))
105.    }
106.    _assert(n.isLeaf, "expected leaf node")
107.    return n
108. }
```

```
109.
110.  // put inserts a key/value.
111.  // 如果put的是一个key、value的话，不需要指定pgid。
112.  // 如果put的一个树枝节点，则需要指定pgid，不需要指定value
113.  func (n *node) put(oldKey, newKey, value []byte, pgid pgid, flags uint32) {
114.      if pgid >= n.bucket.tx.meta.pgid {
          panic(fmt.Sprintf("pgid (%d) above high water mark (%d)", pgid,
115.  n.bucket.tx.meta.pgid))
116.      } else if len(oldKey) <= 0 {
117.          panic("put: zero-length old key")
118.      } else if len(newKey) <= 0 {
119.          panic("put: zero-length new key")
120.      }
121.
122.      // Find insertion index.
          index := sort.Search(len(n.inodes), func(i int) bool { return
123.  bytes.Compare(n.inodes[i].key, oldKey) != -1 })
124.
          // Add capacity and shift nodes if we don't have an exact match and need to
125.  insert.
          exact := (len(n.inodes) > 0 && index < len(n.inodes) &&
126.  bytes.Equal(n.inodes[index].key, oldKey))
127.      if !exact {
128.          n.inodes = append(n.inodes, inode{})
129.          copy(n.inodes[index+1:], n.inodes[index:])
130.      }
131.
132.      inode := &n.inodes[index]
133.      inode.flags = flags
134.      inode.key = newKey
135.      inode.value = value
136.      inode.pgid = pgid
137.      _assert(len(inode.key) > 0, "put: zero-length inode key")
138.  }
```

### 3.4.2 获取一个Bucket

根据指定的key来获取一个Bucket。如果找不到则返回nil。

```
1.  // Bucket retrieves a nested bucket by name.
2.  // Returns nil if the bucket does not exist.
3.  // The bucket instance is only valid for the lifetime of the transaction.
4.  func (b *Bucket) Bucket(name []byte) *Bucket {
```

```
    if b.buckets != nil {
        if child := b.buckets[string(name)]; child != nil {
            return child
        }
    }

    // Move cursor to key.
    // 根据游标找key
    c := b.Cursor()
    k, v, flags := c.seek(name)

    // Return nil if the key doesn't exist or it is not a bucket.
    if !bytes.Equal(name, k) || (flags&bucketLeafFlag) == 0 {
        return nil
    }

    // Otherwise create a bucket and cache it.
    // 根据找到的value来打开桶。
    var child = b.openBucket(v)
    // 加速缓存的作用
    if b.buckets != nil {
        b.buckets[string(name)] = child
    }

    return child
}

// Helper method that re-interprets a sub-bucket value
// from a parent into a Bucket
func (b *Bucket) openBucket(value []byte) *Bucket {
    var child = newBucket(b.tx)

    // If unaligned load/stores are broken on this arch and value is
    // unaligned simply clone to an aligned byte array.
    unaligned := brokenUnaligned && uintptr(unsafe.Pointer(&value[0]))&3 != 0

    if unaligned {
        value = cloneBytes(value)
    }

    // If this is a writable transaction then we need to copy the bucket entry.
    // Read-only transactions can point directly at the mmap entry.
```

```
    if b.tx.writable && !unaligned {
        child.bucket = &bucket{}
        *child.bucket = *(*bucket)(unsafe.Pointer(&value[0]))
    } else {
        child.bucket = (*bucket)(unsafe.Pointer(&value[0]))
    }

    // Save a reference to the inline page if the bucket is inline.
    // 内联桶
    if child.root == 0 {
        child.page = (*page)(unsafe.Pointer(&value[bucketHeaderSize]))
    }

    return &child
}
```

### 3.4.3 删除一个Bucket

DeleteBucket()方法用来删除一个指定key的Bucket。其内部实现逻辑是先递归的删除其子桶。然后再释放该Bucket的page，并最终从叶子节点中移除

```
// DeleteBucket deletes a bucket at the given key.
// Returns an error if the bucket does not exists, or if the key represents a
non-bucket value.
func (b *Bucket) DeleteBucket(key []byte) error {
    if b.tx.db == nil {
        return ErrTxClosed
    } else if !b.Writable() {
        return ErrTxNotWritable
    }

    // Move cursor to correct position.
    c := b.Cursor()
    k, _, flags := c.seek(key)

    // Return an error if bucket doesn't exist or is not a bucket.
    if !bytes.Equal(key, k) {
        return ErrBucketNotFound
    } else if (flags & bucketLeafFlag) == 0 {
        return ErrIncompatibleValue
    }

```

```
21.     // Recursively delete all child buckets.
22.     child := b.Bucket(key)
23.     // 将该桶下面的所有桶都删除
24.     err := child.ForEach(func(k, v []byte) error {
25.         if v == nil {
26.             if err := child.DeleteBucket(k); err != nil {
27.                 return fmt.Errorf("delete bucket: %s", err)
28.             }
29.         }
30.         return nil
31.     })
32.     if err != nil {
33.         return err
34.     }
35.
36.     // Remove cached copy.
37.     delete(b.buckets, string(key))
38.
39.     // Release all bucket pages to freelist.
40.     child.nodes = nil
41.     child.rootNode = nil
42.     child.free()
43.
44.     // Delete the node if we have a matching key.
45.     c.node().del(key)
46.
47.     return nil
48. }
49.
50. // del removes a key from the node.
51. func (n *node) del(key []byte) {
52.     // Find index of key.
        index := sort.Search(len(n.inodes), func(i int) bool { return
53. bytes.Compare(n.inodes[i].key, key) != -1 })
54.
55.     // Exit if the key isn't found.
56.     if index >= len(n.inodes) || !bytes.Equal(n.inodes[index].key, key) {
57.         return
58.     }
59.
60.     // Delete inode from the node.
61.     n.inodes = append(n.inodes[:index], n.inodes[index+1:]...)
```

```
62.
63.     // Mark the node as needing rebalancing.
64.     n.unbalanced = true
65. }
66.
67. // free recursively frees all pages in the bucket.
68. func (b *Bucket) free() {
69.     if b.root == 0 {
70.         return
71.     }
72.
73.     var tx = b.tx
74.     b.forEachPageNode(func(p *page, n *node, _ int) {
75.         if p != nil {
76.             tx.db.freelist.free(tx.meta.txid, p)
77.         } else {
78.             n.free()
79.         }
80.     })
81.     b.root = 0
82. }
```

# 第五节 key/value的插入、获取、删除

上面一节我们介绍了一下如何创建一个Bucket、如何获取一个Bucket。有了Bucket，我们就可以对我们最关心的key/value键值对进行增删改查了。其实本质上，对key/value的所有操作最终都要表现在底层的node上。因为node节点就是用来存储真实数据的。

## 3.5.1 插入一个key/value对

```
1.  // Put sets the value for a key in the bucket.
2.  // If the key exist then its previous value will be overwritten.
3.  // Supplied value must remain valid for the life of the transaction.
4.  // Returns an error if the bucket was created from a read-only transaction,
5.  // if the key is blank, if the key is too large, or if the value is too large.
6.  func (b *Bucket) Put(key []byte, value []byte) error {
7.      if b.tx.db == nil {
8.          return ErrTxClosed
9.      } else if !b.Writable() {
10.         return ErrTxNotWritable
11.     } else if len(key) == 0 {
12.         return ErrKeyRequired
13.     } else if len(key) > MaxKeySize {
14.         return ErrKeyTooLarge
15.     } else if int64(len(value)) > MaxValueSize {
16.         return ErrValueTooLarge
17.     }
18.
19.     // Move cursor to correct position.
20.     c := b.Cursor()
21.     k, _, flags := c.seek(key)
22.
23.     // Return an error if there is an existing key with a bucket value.
24.     if bytes.Equal(key, k) && (flags&bucketLeafFlag) != 0 {
25.         return ErrIncompatibleValue
26.     }
27.
28.     // Insert into node.
29.     key = cloneBytes(key)
30.     c.node().put(key, key, value, 0, 0)
31.
32.     return nil
33. }
```

### 3.5.2 获取一个key/value对

```
// Get retrieves the value for a key in the bucket.
// Returns a nil value if the key does not exist or if the key is a nested
bucket.
// The returned value is only valid for the life of the transaction.
func (b *Bucket) Get(key []byte) []byte {
    k, v, flags := b.Cursor().seek(key)

    // Return nil if this is a bucket.
    if (flags & bucketLeafFlag) != 0 {
        return nil
    }

    // If our target node isn't the same key as what's passed in then return
nil.
    if !bytes.Equal(key, k) {
        return nil
    }
    return v
}
```

### 3.5.3 删除一个key/value对

```
// Delete removes a key from the bucket.
// If the key does not exist then nothing is done and a nil error is returned.
// Returns an error if the bucket was created from a read-only transaction.
func (b *Bucket) Delete(key []byte) error {
    if b.tx.db == nil {
        return ErrTxClosed
    } else if !b.Writable() {
        return ErrTxNotWritable
    }

    // Move cursor to correct position.
    c := b.Cursor()
    _, _, flags := c.seek(key)

    // Return an error if there is already existing bucket value.
    if (flags & bucketLeafFlag) != 0 {
        return ErrIncompatibleValue
```

```
    }

    // Delete the node if we have a matching key.
    c.node().del(key)

    return nil
}
```

### 3.5.4 遍历Bucket中所有的键值对

```
// ForEach executes a function for each key/value pair in a bucket.
// If the provided function returns an error then the iteration is stopped and
// the error is returned to the caller. The provided function must not modify
// the bucket; this will result in undefined behavior.
func (b *Bucket) ForEach(fn func(k, v []byte) error) error {
    if b.tx.db == nil {
        return ErrTxClosed
    }
    c := b.Cursor()
    // 遍历键值对
    for k, v := c.First(); k != nil; k, v = c.Next() {
        if err := fn(k, v); err != nil {
            return err
        }
    }
    return nil
}
```

# 第六节 Bucket的页分裂、页合并

**spill()**

```
// spill writes all the nodes for this bucket to dirty pages.
func (b *Bucket) spill() error {
    // Spill all child buckets first.
    for name, child := range b.buckets {
        // If the child bucket is small enough and it has no child buckets then
        // write it inline into the parent bucket's page. Otherwise spill it
        // like a normal bucket and make the parent value a pointer to the
page.
        var value []byte
        if child.inlineable() {
            child.free()
            // 重新更新bucket的val的值
            value = child.write()
        } else {
            if err := child.spill(); err != nil {
                return err
            }

            // Update the child bucket header in this bucket.
            // 记录value
            value = make([]byte, unsafe.Sizeof(bucket{}))
            var bucket = (*bucket)(unsafe.Pointer(&value[0]))
            *bucket = *child.bucket
        }

        // Skip writing the bucket if there are no materialized nodes.
        if child.rootNode == nil {
            continue
        }

        // Update parent node.
        var c = b.Cursor()
        k, _, flags := c.seek([]byte(name))
        if !bytes.Equal([]byte(name), k) {
            panic(fmt.Sprintf("misplaced bucket header: %x -> %x",
[]byte(name), k))
        }
```

```
36.         if flags&bucketLeafFlag == 0 {
37.             panic(fmt.Sprintf("unexpected bucket header flag: %x", flags))
38.         }
39.         // 更新子桶的value
40.         c.node().put([]byte(name), []byte(name), value, 0, bucketLeafFlag)
41.     }
42.
43.     // Ignore if there's not a materialized root node.
44.     if b.rootNode == nil {
45.         return nil
46.     }
47.
48.     // Spill nodes.
49.     if err := b.rootNode.spill(); err != nil {
50.         return err
51.     }
52.     b.rootNode = b.rootNode.root()
53.
54.     // Update the root node for this bucket.
55.     if b.rootNode.pgid >= b.tx.meta.pgid {
           panic(fmt.Sprintf("pgid (%d) above high water mark (%d)",
56. b.rootNode.pgid, b.tx.meta.pgid))
57.     }
58.     b.root = b.rootNode.pgid
59.
60.     return nil
61. }
62.
63. // inlineable returns true if a bucket is small enough to be written inline
64. // and if it contains no subbuckets. Otherwise returns false.
65. func (b *Bucket) inlineable() bool {
66.     var n = b.rootNode
67.
68.     // Bucket must only contain a single leaf node.
69.     if n == nil || !n.isLeaf {
70.         return false
71.     }
72.
73.     // Bucket is not inlineable if it contains subbuckets or if it goes beyond
74.     // our threshold for inline bucket size.
75.     var size = pageHeaderSize
76.     for _, inode := range n.inodes {
```

```
77.         size += leafPageElementSize + len(inode.key) + len(inode.value)
78.
79.         if inode.flags&bucketLeafFlag != 0 {
80.             // 有子桶时，不能内联
81.             return false
82.         } else if size > b.maxInlineBucketSize() {
83.             // 如果长度大于1/4页时，就不内联了
84.             return false
85.         }
86.     }
87.
88.     return true
89. }
90.
    // Returns the maximum total size of a bucket to make it a candidate for
91. inlining.
92. func (b *Bucket) maxInlineBucketSize() int {
93.     return b.tx.db.pageSize / 4
94. }
```

### rebalance()

```
1.  // rebalance attempts to balance all nodes.
2.  func (b *Bucket) rebalance() {
3.      for _, n := range b.nodes {
4.          n.rebalance()
5.      }
6.      for _, child := range b.buckets {
7.          child.rebalance()
8.      }
9.  }
```

# 第七节 总结

本章我们主要介绍了boltdb中比较核心的两个数据结构：**Bucket**、**node**。为什么这两个数据结构放在一起介绍呢？答案是在boltdb中一个Bucket就对应一颗b+树。而b+树的结构(根节点、叶子节点、非叶子节点)、组织都是通过node来完成的。这也是为什么把他们放在一起介绍的主要原因。

在介绍中，我们主要围绕Bucket的创建、获取、删除、遍历、增删改**kv**等操作进行展开。其次在遍历时，就引入了Cursor数据结构，一个Bucket对象(一颗b+树)的遍历在boltdb中时通过一个栈来维护遍历的路径来完成的。这也是Cursor中 `stack` 的意义。

其次Bucket中对kv的操作都反应到底层的node上，因此我们又同时介绍了node的相关方法，例如**put**、**get**、**del**、**spill**、**rebalance**。

最后到此为止，我们的数据时如何存储的、组织的。以及内存和磁盘数据时如何转换映射的，我们就清楚了。下章将介绍boltdb中的事务了。有了事务我们的数据库才称得上是一个完备的数据库。

# 第四章 boltdb事务控制

事务可以说是一个数据库必不可少的特性，对boltdb而言也不例外。我们都知道提到事务，必然会想到事务的四大特性。那么下面就让我们看看在boltdb中到底是怎么实现它的事务的呢？

# 第一节 boltdb事务简介

我们先看一下，boltdb官方文档中对事务的描述：

> Bolt allows only one read-write transaction at a time but allows as many read-only transactions as you want at a time. Each transaction has a consistent view of the data as it existed when the transaction started.
>
> Individual transactions and all objects created from them (e.g. buckets, keys) are not thread safe. To work with data in multiple goroutines you must start a transaction for each one or use locking to ensure only one goroutine accesses a transaction at a time. Creating transaction from the `DB` is thread safe.
>
> Read-only transactions and read-write transactions should not depend on one another and generally shouldn't be opened simultaneously in the same goroutine. This can cause a deadlock as the read-write transaction needs to periodically re-map the data file but it cannot do so while a read-only transaction is open.

我们再简单总结下，在boltdb中支持两类事务：读写事务、只读事务。同一时间有且只能有一个读写事务执行；但同一个时间可以允许有多个只读事务执行。每个事务都拥有自己的一套一致性视图。

此处需要注意的是，在boltdb中打开一个数据库时，有两个选项：只读模式、读写模式。内部在实现时是根据不同的选项来底层加不同的锁(flock)。只读模式对应共享锁，读写模式对应互斥锁。具体加解锁的实现可以在bolt_unix.go 和bolt_windows.go中找到。

提到事务，我们不得不提大家烂熟于心的事务四个特性：ACID。为方便阅读后续的内容，下面再简单回顾一下：

**A(atomic)**原子性:事务的原子性主要表示的是，只要事务一开始(Begin)，那么事务要么执行成功(Commit)，要么执行失败(Rollback)。上述过程只会出现两种状态，在事务执行过程中的中间状态以及数据是不可见的。

**C(consistency)**一致性：事务的一致性是指，事务开始前和事务提交后的数据都是一致的。

**I(isolation)**隔离性:事务的隔离性是指不同事务之间是相互隔离、互不影响的。具体的隔离程度是由具体的事务隔离级别来控制。

**D(duration)**持久性:事务的持久性是指，事务开始前和事务提交后的数据都是永久的。不会存在数据丢失或者篡改的风险。

在此再总结一下：其实上述四大特性中，事务的一致性是终极目标，而其他三大特性都是为了保证一致性而服务的手段。在mysql中，事务的原子性由undo log来保证；事务的持久性由redo log来保证；事务的隔离性由锁来保证。

那具体到boltdb中，它又是如何来实现的呢？

此处以个人的理解来回答下这个问题，理解不一定准确。

首先boltdb是一个文件数据库，所有的数据最终都保存在文件中。当事务结束(Commit)时，会将数据进行刷盘。同时，boltdb通过冗余一份元数据来做容错。当事务提交时，如果写入到一半机器挂了，此时数据就会有问题。而当boltdb再次恢复时，会对元数据进行校验和修复。这两点就保证事务中的持久性。

其次boltdb在上层支持多个进程以只读的方式打开数据库，一个进程以写的方式打开数据库。在数据库内部中事务支持两种，读写事务和只读事务。这两类事务是互斥的。同一时间可以有多个只读事务执行，或者只能有一个读写事务执行，上述两类事务，在底层实现时，都是保留一整套完整的视图和元数据信息，彼此之间相互隔离。因此通过这两点就保证了隔离性。

在boltdb中，数据先写内存，然后再提交时刷盘。如果其中有异常发生，事务就会回滚。同时再加上同一时间只有一个进行对数据执行写入操作。所以它要么写成功提交、要么写失败回滚。也就支持原子性了。

通过以上的几个特性的保证，最终也就保证了一致性。

# 第二节 boltdb事务Tx定义

```
// txid represents the internal transaction identifier.
type txid uint64

// Tx represents a read-only or read/write transaction on the database.
// Read-only transactions can be used for retrieving values for keys and creating cursors.
// Read/write transactions can create and remove buckets and create and remove keys.
//
// IMPORTANT: You must commit or rollback transactions when you are done with
// them. Pages can not be reclaimed by the writer until no more transactions
// are using them. A long running read transaction can cause the database to
// quickly grow.
// Tx 主要封装了读事务和写事务。其中通过writable来区分是读事务还是写事务
type Tx struct {
    writable       bool
    managed        bool
    db             *DB
    meta           *meta
    root           Bucket
    pages          map[pgid]*page
    stats          TxStats
    // 提交时执行的动作
    commitHandlers []func()

    // WriteFlag specifies the flag for write-related methods like WriteTo().
    // Tx opens the database file with the specified flag to copy the data.
    //
    // By default, the flag is unset, which works well for mostly in-memory
    // workloads. For databases that are much larger than available RAM,
    // set the flag to syscall.O_DIRECT to avoid trashing the page cache.
    WriteFlag int
}

// init initializes the transaction.
func (tx *Tx) init(db *DB) {
    tx.db = db
    tx.pages = nil

```

```
38.     // Copy the meta page since it can be changed by the writer.
39.     // 拷贝元信息
40.     tx.meta = &meta{}
41.     db.meta().copy(tx.meta)
42.
43.     // Copy over the root bucket.
44.     // 拷贝根节点
45.     tx.root = newBucket(tx)
46.     tx.root.bucket = &bucket{}
47.     // meta.root=bucket{root:3}
48.     *tx.root.bucket = tx.meta.root
49.
        // Increment the transaction id and add a page cache for writable
50. transactions.
51.     if tx.writable {
52.         tx.pages = make(map[pgid]*page)
53.         tx.meta.txid += txid(1)
54.     }
55. }
```

# 第三节 Begin()实现

此处需要说明一下：在boltdb中，事务的开启方法是绑定在DB对象上的，为了保证内容的完整性，我们还是把事务开启的Begin()方法补充到这个地方。

前面提到boltdb中事务分为两类，它的区分就是在开启事务时，根据传递的参数来内部执行不同的逻辑。

在读写事务中，开始事务时加锁，也就是 `db.rwlock.Lock()` 。在事务提交或者回滚时才释放锁： `db.rwlock.UnLock()` 。同时也印证了我们前面说的，同一时刻只能有一个读写事务在执行。

```
1.  // Begin starts a new transaction.
2.  // Multiple read-only transactions can be used concurrently but only one
    // write transaction can be used at a time. Starting multiple write
3.  transactions
4.  // will cause the calls to block and be serialized until the current write
5.  // transaction finishes.
6.  //
7.  // Transactions should not be dependent on one another. Opening a read
8.  // transaction and a write transaction in the same goroutine can cause the
9.  // writer to deadlock because the database periodically needs to re-mmap itself
10. // as it grows and it cannot do that while a read transaction is open.
11. //
12. // If a long running read transaction (for example, a snapshot transaction) is
13. // needed, you might want to set DB.InitialMmapSize to a large enough value
14. // to avoid potential blocking of write transaction.
15. //
16. // IMPORTANT: You must close read-only transactions after you are finished or
17. // else the database will not reclaim old pages.
18. func (db *DB) Begin(writable bool) (*Tx, error) {
19.     if writable {
20.         return db.beginRWTx()
21.     }
22.     return db.beginTx()
23. }
24.
25. func (db *DB) beginTx() (*Tx, error) {
26.     // Lock the meta pages while we initialize the transaction. We obtain
27.     // the meta lock before the mmap lock because that's the order that the
28.     // write transaction will obtain them.
29.     db.metalock.Lock()
```

```
30.
31.     // Obtain a read-only lock on the mmap. When the mmap is remapped it will
32.     // obtain a write lock so all transactions must finish before it can be
33.     // remapped.
34.     db.mmaplock.RLock()
35.
36.     // Exit if the database is not open yet.
37.     if !db.opened {
38.         db.mmaplock.RUnlock()
39.         db.metalock.Unlock()
40.         return nil, ErrDatabaseNotOpen
41.     }
42.
43.     // Create a transaction associated with the database.
44.     t := &Tx{}
45.     t.init(db)
46.
47.     // Keep track of transaction until it closes.
48.     db.txs = append(db.txs, t)
49.     n := len(db.txs)
50.
51.     // Unlock the meta pages.
52.     db.metalock.Unlock()
53.
54.     // Update the transaction stats.
55.     db.statlock.Lock()
56.     db.stats.TxN++
57.     db.stats.OpenTxN = n
58.     db.statlock.Unlock()
59.
60.     return t, nil
61. }
62.
63. func (db *DB) beginRWTx() (*Tx, error) {
64.     // If the database was opened with Options.ReadOnly, return an error.
65.     if db.readOnly {
66.         return nil, ErrDatabaseReadOnly
67.     }
68.
69.     // Obtain writer lock. This is released by the transaction when it closes.
70.     // This enforces only one writer transaction at a time.
71.     db.rwlock.Lock()
```

```
72.
73.     // Once we have the writer lock then we can lock the meta pages so that
74.     // we can set up the transaction.
75.     db.metalock.Lock()
76.     defer db.metalock.Unlock()
77.
78.     // Exit if the database is not open yet.
79.     if !db.opened {
80.         db.rwlock.Unlock()
81.         return nil, ErrDatabaseNotOpen
82.     }
83.
84.     // Create a transaction associated with the database.
85.     t := &Tx{writable: true}
86.     t.init(db)
87.     db.rwtx = t
88.
89.     // Free any pages associated with closed read-only transactions.
90.     var minid txid = 0xFFFFFFFFFFFFFFFF
91.     // 找到最小的事务id
92.     for _, t := range db.txs {
93.         if t.meta.txid < minid {
94.             minid = t.meta.txid
95.         }
96.     }
97.     if minid > 0 {
            // 将之前事务关联的page全部释放了，因为在只读事务中，没法释放，只读事务的页，因为可
98. 能当前的事务已经完成 ，但实际上其他的读事务还在用
99.         db.freelist.release(minid - 1)
100.    }
101.
102.    return t, nil
103. }
```

# 第四节 Commit()实现

Commit()方法内部实现中，总体思路是：

1. 先判定节点要不要合并、分裂
2. 对空闲列表的判断，是否存在溢出的情况，溢出的话，需要重新分配空间
3. 将事务中涉及改动的页进行排序(保证尽可能的顺序IO)，排序后循环写入到磁盘中，最后再执行刷盘
4. 当数据写入成功后，再将元信息页写到磁盘中，刷盘以保证持久化
5. 上述操作中，但凡有失败，当前事务都会进行回滚

```
1.  // Commit writes all changes to disk and updates the meta page.
2.  // Returns an error if a disk write error occurs, or if Commit is
3.  // called on a read-only transaction.
4.
5.  // 先更新数据然后再更新元信息
6.  // 更新数据成功、元信息未来得及更新机器就挂掉了。数据如何恢复？
7.  func (tx *Tx) Commit() error {
8.      _assert(!tx.managed, "managed tx commit not allowed")
9.      if tx.db == nil {
10.         return ErrTxClosed
11.     } else if !tx.writable {
12.         return ErrTxNotWritable
13.     }
14.
15.     // TODO(benbjohnson): Use vectorized I/O to write out dirty pages.
16.
17.     // 删除时，进行平衡，页合并
18.     // Rebalance nodes which have had deletions.
19.     var startTime = time.Now()
20.     tx.root.rebalance()
21.     if tx.stats.Rebalance > 0 {
22.         tx.stats.RebalanceTime += time.Since(startTime)
23.     }
24.
25.     // 页分裂
26.     // spill data onto dirty pages.
27.     startTime = time.Now()
28.     // 这个内部会往缓存tx.pages中加page
29.     if err := tx.root.spill(); err != nil {
30.         tx.rollback()
```

```
31.         return err
32.     }
33.     tx.stats.SpillTime += time.Since(startTime)
34.
35.     // Free the old root bucket.
36.     tx.meta.root.root = tx.root.root
37.
38.     opgid := tx.meta.pgid
39.
40.     // Free the freelist and allocate new pages for it. This will overestimate
        // the size of the freelist but not underestimate the size (which would be
41. bad).
42.     // 分配新的页面给freelist，然后将freelist写入新的页面
43.     tx.db.freelist.free(tx.meta.txid, tx.db.page(tx.meta.freelist))
44.     // 空闲列表可能会增加，因此需要重新分配页用来存储空闲列表
        // 因为在开启写事务的时候，有去释放之前读事务占用的页信息，因此此处需要判断是否freelist
45. 会有溢出的问题
46.     p, err := tx.allocate((tx.db.freelist.size() / tx.db.pageSize) + 1)
47.     if err != nil {
48.         tx.rollback()
49.         return err
50.     }
51.     // 将freelist写入到连续的新页中
52.     if err := tx.db.freelist.write(p); err != nil {
53.         tx.rollback()
54.         return err
55.     }
56.     // 更新元数据的页id
57.     tx.meta.freelist = p.id
58.
59.     // If the high water mark has moved up then attempt to grow the database.
60.     // 在allocate中有可能会更改meta.pgid
61.     if tx.meta.pgid > opgid {
            if err := tx.db.grow(int(tx.meta.pgid+1) * tx.db.pageSize); err != nil
62. {
63.             tx.rollback()
64.             return err
65.         }
66.     }
67.
68.     // Write dirty pages to disk.
69.     startTime = time.Now()
70.     // 写数据
```

```
71.     if err := tx.write(); err != nil {
72.         tx.rollback()
73.         return err
74.     }
75.
76.     // If strict mode is enabled then perform a consistency check.
77.     // Only the first consistency error is reported in the panic.
78.     if tx.db.StrictMode {
79.         ch := tx.Check()
80.         var errs []string
81.         for {
82.             err, ok := <-ch
83.             if !ok {
84.                 break
85.             }
86.             errs = append(errs, err.Error())
87.         }
88.         if len(errs) > 0 {
89.             panic("check fail: " + strings.Join(errs, "\n"))
90.         }
91.     }
92.
93.     // Write meta to disk.
94.     // 元信息写入到磁盘
95.     if err := tx.writeMeta(); err != nil {
96.         tx.rollback()
97.         return err
98.     }
99.     tx.stats.WriteTime += time.Since(startTime)
100.
101.     // Finalize the transaction.
102.     tx.close()
103.
104.     // Execute commit handlers now that the locks have been removed.
105.     for _, fn := range tx.commitHandlers {
106.         fn()
107.     }
108.
109.     return nil
110. }
111.
112. // write writes any dirty pages to disk.
```

```
113.  func (tx *Tx) write() error {
114.      // Sort pages by id.
115.      // 保证写的页是有序的
116.      pages := make(pages, 0, len(tx.pages))
117.      for _, p := range tx.pages {
118.          pages = append(pages, p)
119.      }
120.      // Clear out page cache early.
121.      tx.pages = make(map[pgid]*page)
122.      sort.Sort(pages)
123.
124.      // Write pages to disk in order.
125.      for _, p := range pages {
126.          // 页数和偏移量
127.          size := (int(p.overflow) + 1) * tx.db.pageSize
128.          offset := int64(p.id) * int64(tx.db.pageSize)
129.
130.          // Write out page in "max allocation" sized chunks.
131.          ptr := (*[maxAllocSize]byte)(unsafe.Pointer(p))
132.          // 循环写某一页
133.          for {
134.              // Limit our write to our max allocation size.
135.              sz := size
136.              // 2^31=2G
137.              if sz > maxAllocSize-1 {
138.                  sz = maxAllocSize - 1
139.              }
140.
141.              // Write chunk to disk.
142.              buf := ptr[:sz]
143.              if _, err := tx.db.ops.writeAt(buf, offset); err != nil {
144.                  return err
145.              }
146.
147.              // Update statistics.
148.              tx.stats.Write++
149.
150.              // Exit inner for loop if we've written all the chunks.
151.              size -= sz
152.              if size == 0 {
153.                  break
154.              }
```

```
155.
156.             // Otherwise move offset forward and move pointer to next chunk.
157.             // 移动偏移量
158.             offset += int64(sz)
159.             // 同时指针也移动
160.             ptr = (*[maxAllocSize]byte)(unsafe.Pointer(&ptr[sz]))
161.         }
162.     }
163.
164.     // Ignore file sync if flag is set on DB.
165.     if !tx.db.NoSync || IgnoreNoSync {
166.         if err := fdatasync(tx.db); err != nil {
167.             return err
168.         }
169.     }
170.
171.     // Put small pages back to page pool.
172.     for _, p := range pages {
173.         // Ignore page sizes over 1 page.
174.         // These are allocated using make() instead of the page pool.
175.         if int(p.overflow) != 0 {
176.             continue
177.         }
178.
179.         buf := (*[maxAllocSize]byte)(unsafe.Pointer(p))[:tx.db.pageSize]
180.
            // See
181. https://go.googlesource.com/go/+/f03c9202c43e0abb130669852082117ca50aa9b1
182.         // 清空buf, 然后放入pagePool中
183.         for i := range buf {
184.             buf[i] = 0
185.         }
186.         tx.db.pagePool.Put(buf)
187.     }
188.
189.     return nil
190. }
191.
192. // writeMeta writes the meta to the disk.
193. func (tx *Tx) writeMeta() error {
194.     // Create a temporary buffer for the meta page.
195.     buf := make([]byte, tx.db.pageSize)
```

```
196.      p := tx.db.pageInBuffer(buf, 0)
197.      // 将事务的元信息写入到页中
198.      tx.meta.write(p)
199.
200.      // Write the meta page to file.
          if _, err := tx.db.ops.writeAt(buf, int64(p.id)*int64(tx.db.pageSize)); err
201.  != nil {
202.          return err
203.      }
204.      if !tx.db.NoSync || IgnoreNoSync {
205.          if err := fdatasync(tx.db); err != nil {
206.              return err
207.          }
208.      }
209.
210.      // Update statistics.
211.      tx.stats.Write++
212.
213.      return nil
214.  }
215.
216.  // allocate returns a contiguous block of memory starting at a given page.
217.  // 分配一段连续的页
218.  func (tx *Tx) allocate(count int) (*page, error) {
219.      p, err := tx.db.allocate(count)
220.      if err != nil {
221.          return nil, err
222.      }
223.
224.      // Save to our page cache.
225.      tx.pages[p.id] = p
226.
227.      // Update statistics.
228.      tx.stats.PageCount++
229.      tx.stats.PageAlloc += count * tx.db.pageSize
230.
231.      return p, nil
232.  }
```

# 第五节 Rollback()实现

Rollback()中，主要对不同事务进行不同操作：

1. 如果当前事务是只读事务，则只需要从db中的txs中找到当前事务，然后移除掉即可。
2. 如果当前事务是读写事务，则需要将空闲列表中和该事务关联的页释放掉，同时重新从freelist中加载空闲页。

```
// Rollback closes the transaction and ignores all previous updates. Read-only
// transactions must be rolled back and not committed.
func (tx *Tx) Rollback() error {
    _assert(!tx.managed, "managed tx rollback not allowed")
    if tx.db == nil {
        return ErrTxClosed
    }
    tx.rollback()
    return nil
}

func (tx *Tx) rollback() {
    if tx.db == nil {
        return
    }
    if tx.writable {
        // 移除该事务关联的pages
        tx.db.freelist.rollback(tx.meta.txid)
        // 重新从freelist页中读取构建空闲列表
        tx.db.freelist.reload(tx.db.page(tx.db.meta().freelist))
    }
    tx.close()
}

func (tx *Tx) close() {
    if tx.db == nil {
        return
    }
    if tx.writable {
        // Grab freelist stats.
        var freelistFreeN = tx.db.freelist.free_count()
        var freelistPendingN = tx.db.freelist.pending_count()
        var freelistAlloc = tx.db.freelist.size()
```

```
34.
35.         // Remove transaction ref & writer lock.
36.         tx.db.rwtx = nil
37.         tx.db.rwlock.Unlock()
38.
39.         // Merge statistics.
40.         tx.db.statlock.Lock()
41.         tx.db.stats.FreePageN = freelistFreeN
42.         tx.db.stats.PendingPageN = freelistPendingN
            tx.db.stats.FreeAlloc = (freelistFreeN + freelistPendingN) *
43. tx.db.pageSize
44.         tx.db.stats.FreelistInuse = freelistAlloc
45.         tx.db.stats.TxStats.add(&tx.stats)
46.         tx.db.statlock.Unlock()
47.     } else {
48.         // 只读事务
49.         tx.db.removeTx(tx)
50.     }
51.
52.     // Clear all references.
53.     tx.db = nil
54.     tx.meta = nil
55.     tx.root = Bucket{tx: tx}
56.     tx.pages = nil
57. }
58.
59. // removeTx removes a transaction from the database.
60. func (db *DB) removeTx(tx *Tx) {
61.     // Release the read lock on the mmap.
62.     db.mmaplock.RUnlock()
63.
64.     // Use the meta lock to restrict access to the DB object.
65.     db.metalock.Lock()
66.
67.     // Remove the transaction.
68.     for i, t := range db.txs {
69.         if t == tx {
70.             last := len(db.txs) - 1
71.             db.txs[i] = db.txs[last]
72.             db.txs[last] = nil
73.             db.txs = db.txs[:last]
74.             break
```

```
75.         }
76.     }
77.     n := len(db.txs)
78.
79.     // Unlock the meta pages.
80.     db.metalock.Unlock()
81.
82.     // Merge statistics.
83.     db.statlock.Lock()
84.     db.stats.OpenTxN = n
85.     db.stats.TxStats.add(&tx.stats)
86.     db.statlock.Unlock()
87. }
```

# 第六节 WriteTo()和CopyFile()实现

```go
// WriteTo writes the entire database to a writer.
// If err == nil then exactly tx.Size() bytes will be written into the writer.
func (tx *Tx) WriteTo(w io.Writer) (n int64, err error) {
    // Attempt to open reader with WriteFlag
    f, err := os.OpenFile(tx.db.path, os.O_RDONLY|tx.WriteFlag, 0)
    if err != nil {
        return 0, err
    }
    defer func() { _ = f.Close() }()

    // Generate a meta page. We use the same page data for both meta pages.
    buf := make([]byte, tx.db.pageSize)
    page := (*page)(unsafe.Pointer(&buf[0]))
    page.flags = metaPageFlag
    *page.meta() = *tx.meta

    // Write meta 0.
    page.id = 0
    page.meta().checksum = page.meta().sum64()
    nn, err := w.Write(buf)
    n += int64(nn)
    if err != nil {
        return n, fmt.Errorf("meta 0 copy: %s", err)
    }

    // Write meta 1 with a lower transaction id.
    page.id = 1
    page.meta().txid -= 1
    page.meta().checksum = page.meta().sum64()
    nn, err = w.Write(buf)
    n += int64(nn)
    if err != nil {
        return n, fmt.Errorf("meta 1 copy: %s", err)
    }

    // Move past the meta pages in the file.
    if _, err := f.Seek(int64(tx.db.pageSize*2), os.SEEK_SET); err != nil {
        return n, fmt.Errorf("seek: %s", err)
```

```
39.     }
40.
41.     // Copy data pages.
42.     wn, err := io.CopyN(w, f, tx.Size()-int64(tx.db.pageSize*2))
43.     n += wn
44.     if err != nil {
45.         return n, err
46.     }
47.
48.     return n, f.Close()
49. }
50.
51. // CopyFile copies the entire database to file at the given path.
52. // A reader transaction is maintained during the copy so it is safe to continue
53. // using the database while a copy is in progress.
54. func (tx *Tx) CopyFile(path string, mode os.FileMode) error {
55.     f, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE|os.O_TRUNC, mode)
56.     if err != nil {
57.         return err
58.     }
59.
60.     err = tx.Copy(f)
61.     if err != nil {
62.         _ = f.Close()
63.         return err
64.     }
65.     return f.Close()
66. }
```

# 第七节 总结

本章主要详细分析了下，boltdb内部事务的实现机制，再此基础上对事务中核心的几个方法做了代码的分析。到此基本上一个数据库核心的部件都已经实现完毕。那剩下的功能就把各部分功能进行组装起来，实现一个完整对外可用的数据库了。下一章我们来详细分析下boltdb中DB对象的内部一些实现。

# 第五章 boltdb的DB对象分析

前面我们介绍了boltdb底层在磁盘上数据时如何组织存储(page)的，然后又介绍了磁盘中的数据在内存中又是如何存储(node)的。接着我们又介绍了管理kv数据集合的Bucket对象以及用来遍历Bucket的Cursor对象。最后我们详细的介绍了boltdb中事务是如何实现(Tx)的。到此boltdb中 各个零散的部件我们都一一熟悉了，接下来是时候将他们组织在一起工作了。因而就有了boltdb中最上层的DB对象。本章主要介绍DB对象相关的方法以及其内部实现。

# 第一节 DB结构

DB在boltdb是一个结构体，里面封装了很多属性，部分属性添加了中文注释，其他部分属性，大家可以直接看英文注释，感觉英文表述的很通俗易懂。

```
1.
2.  // The largest step that can be taken when remapping the mmap.
3.  const maxMmapStep = 1 << 30 // 1GB
4.
5.  // The data file format version.
6.  const version = 2
7.
8.  // Represents a marker value to indicate that a file is a Bolt DB.
9.  const magic uint32 = 0xED0CDAED
10.
11. // IgnoreNoSync specifies whether the NoSync field of a DB is ignored when
12. // syncing changes to a file.  This is required as some operating systems,
13. // such as OpenBSD, do not have a unified buffer cache (UBC) and writes
14. // must be synchronized using the msync(2) syscall.
15. const IgnoreNoSync = runtime.GOOS == "openbsd"
16.
17. // Default values if not set in a DB instance.
18. const (
19.     DefaultMaxBatchSize  int =s 1000
20.     DefaultMaxBatchDelay     = 10 * time.Millisecond
21.     // 16k
22.     DefaultAllocSize = 16 * 1024 * 1024
23. )
24.
25. // default page size for db is set to the OS page size.
26. var defaultPageSize = os.Getpagesize()
27.
28. // DB represents a collection of buckets persisted to a file on disk.
    // All data access is performed through transactions which can be obtained
29. through the DB.
    // All the functions on DB will return a ErrDatabaseNotOpen if accessed before
30. Open() is called.
31. type DB struct {
32.     // When enabled, the database will perform a Check() after every commit.
33.     // A panic is issued if the database is in an inconsistent state. This
34.     // flag has a large performance impact so it should only be used for
```

```
35.     // debugging purposes.
36.     StrictMode bool
37.
38.     // Setting the NoSync flag will cause the database to skip fsync()
39.     // calls after each commit. This can be useful when bulk loading data
40.     // into a database and you can restart the bulk load in the event of
41.     // a system failure or database corruption. Do not set this flag for
42.     // normal use.
43.     //
44.     // If the package global IgnoreNoSync constant is true, this value is
45.     // ignored.  See the comment on that constant for more details.
46.     //
47.     // THIS IS UNSAFE. PLEASE USE WITH CAUTION.
48.     NoSync bool
49.
50.     // When true, skips the truncate call when growing the database.
51.     // Setting this to true is only safe on non-ext3/ext4 systems.
52.     // Skipping truncation avoids preallocation of hard drive space and
53.     // bypasses a truncate() and fsync() syscall on remapping.
54.     //
55.     // https://github.com/boltdb/bolt/issues/284
56.     NoGrowSync bool
57.
58.     // If you want to read the entire database fast, you can set MmapFlag to
59.     // syscall.MAP_POPULATE on Linux 2.6.23+ for sequential read-ahead.
60.     MmapFlags int
61.
62.     // MaxBatchSize is the maximum size of a batch. Default value is
63.     // copied from DefaultMaxBatchSize in Open.
64.     //
65.     // If <=0, disables batching.
66.     //
67.     // Do not change concurrently with calls to Batch.
68.     MaxBatchSize int
69.
70.     // MaxBatchDelay is the maximum delay before a batch starts.
71.     // Default value is copied from DefaultMaxBatchDelay in Open.
72.     //
73.     // If <=0, effectively disables batching.
74.     //
75.     // Do not change concurrently with calls to Batch.
76.     MaxBatchDelay time.Duration
```

```
77.
78.     // AllocSize is the amount of space allocated when the database
79.     // needs to create new pages. This is done to amortize the cost
80.     // of truncate() and fsync() when growing the data file.
81.     AllocSize int
82.
83.     path     string
84.     file     *os.File  // 真实存储数据的磁盘文件
85.     lockfile *os.File // windows only
86.     dataref  []byte   // mmap'ed readonly, write throws SEGV
87.     // 通过mmap映射进来的地址
88.     data   *[maxMapSize]byte
89.     datasz int
90.     filesz int // current on disk file size
91.     //  元数据
92.     meta0 *meta
93.     meta1 *meta
94.
95.     pageSize int
96.     opened   bool
97.     rwtx     *Tx  // 写事务锁
98.     txs      []*Tx  // 读事务数组
99.     freelist *freelist // 空闲列表
100.    stats    Stats
101.
102.    pagePool sync.Pool
103.
104.    batchMu sync.Mutex
105.    batch   *batch
106.
107.    rwlock   sync.Mutex   // Allows only one writer at a time.
108.    metalock sync.Mutex   // Protects meta page access.
109.    mmaplock sync.RWMutex // Protects mmap access during remapping.
110.    statlock sync.RWMutex // Protects stats access.
111.
112.    ops struct {
113.        writeAt func(b []byte, off int64) (n int, err error)
114.    }
115.
116.    // Read only mode.
        // When true, Update() and Begin(true) return ErrDatabaseReadOnly
117. immediately.
```

```
118.      readOnly bool
119.  }
```

# 第二节 对外接口

### 1.Open()创建数据库接口

```
// Open creates and opens a database at the given path.
// If the file does not exist then it will be created automatically.
// Passing in nil options will cause Bolt to open the database with the default
options.
// 创建数据库接口
func Open(path string, mode os.FileMode, options *Options) (*DB, error)
```

### 2.View()查询接口

```
// View executes a function within the context of a managed read-only
transaction.
// Any error that is returned from the function is returned from the View()
method.
//
// Attempting to manually rollback within the function will cause a panic.
func (db *DB) View(fn func(*Tx) error) error
```

### 3.Update()更新接口

```
// Update executes a function within the context of a read-write managed
transaction.
// If no error is returned from the function then the transaction is committed.
// If an error is returned then the entire transaction is rolled back.
// Any error that is returned from the function or returned from the commit is
// returned from the Update() method.
//
// Attempting to manually commit or rollback within the function will cause a
panic.
func (db *DB) Update(fn func(*Tx) error) error
```

### 4.Batch()批量更新接口

```
// Batch calls fn as part of a batch. It behaves similar to Update,
// except:
//
// 1. concurrent Batch calls can be combined into a single Bolt
// transaction.
```

```
//
// 2. the function passed to Batch may be called multiple times,
// regardless of whether it returns error or not.
//
// This means that Batch function side effects must be idempotent and
// take permanent effect only after a successful return is seen in
// caller.
//
// The maximum batch size and delay can be adjusted with DB.MaxBatchSize
// and DB.MaxBatchDelay, respectively.
//
// Batch is only useful when there are multiple goroutines calling it.
func (db *DB) Batch(fn func(*Tx) error) error
```

**5.Begin()开启事务接口**

```
// Begin starts a new transaction.
// Multiple read-only transactions can be used concurrently but only one
// write transaction can be used at a time. Starting multiple write
transactions
// will cause the calls to block and be serialized until the current write
// transaction finishes.
//
// Transactions should not be dependent on one another. Opening a read
// transaction and a write transaction in the same goroutine can cause the
// writer to deadlock because the database periodically needs to re-mmap itself
// as it grows and it cannot do that while a read transaction is open.
//
// If a long running read transaction (for example, a snapshot transaction) is
// needed, you might want to set DB.InitialMmapSize to a large enough value
// to avoid potential blocking of write transaction.
//
// IMPORTANT: You must close read-only transactions after you are finished or
// else the database will not reclaim old pages.
func (db *DB) Begin(writable bool) (*Tx, error)
```

> 备注：Begin()的实现分析，参见事务4.3节内容，下面不在做分析。

下面我们将对上述接口做一一分析

# 第三节 Open()实现分析

Open()方法主要用来创建一个boltdb的DB对象，底层会执行新建或者打开存储数据的文件，当指定的文件不存在时， boltdb就会新建一个数据文件。否则的话，就直接加载指定的数据库文件内容。

值的注意是，boltdb会根据Open时，options传递的参数来判断到底加互斥锁还是共享锁。

新建时： 会调用init()方法，内部主要是新建一个文件，然后第0页、第1页写入元数据信息；第2页写入freelist信息；第3页写入bucket leaf信息。并最终刷盘。

加载时： 会读取第0页内容，也就是元信息。然后对其进行校验和校验，当校验通过后获取pageSize。否则的话，读取操作系统默认的pagesize(一般4k)

上述操作完成后，会通过mmap来映射数据。最后再根据磁盘页中的freelist数据初始化db的freelist字段。

```
1.  // Open creates and opens a database at the given path.
2.  // If the file does not exist then it will be created automatically.
    // Passing in nil options will cause Bolt to open the database with the default
3.  options.
4.  func Open(path string, mode os.FileMode, options *Options) (*DB, error) {
5.      var db = &DB{opened: true}
6.
7.      // Set default options if no options are provided.
8.      if options == nil {
9.          options = DefaultOptions
10.     }
11.     db.NoGrowSync = options.NoGrowSync
12.     db.MmapFlags = options.MmapFlags
13.
14.     // Set default values for later DB operations.
15.     db.MaxBatchSize = DefaultMaxBatchSize
16.     db.MaxBatchDelay = DefaultMaxBatchDelay
17.     db.AllocSize = DefaultAllocSize
18.
19.     flag := os.O_RDWR
20.     if options.ReadOnly {
21.         flag = os.O_RDONLY
22.         db.readOnly = true
23.     }
24.
25.     // Open data file and separate sync handler for metadata writes.
```

```
26.     db.path = path
27.     var err error
28.     // 打开db文件
        if db.file, err = os.OpenFile(db.path, flag|os.O_CREATE, mode); err != nil
29. {
30.         _ = db.close()
31.         return nil, err
32.     }
33.
34.     // Lock file so that other processes using Bolt in read-write mode cannot
35.     // use the database  at the same time. This would cause corruption since
36.     // the two processes would write meta pages and free pages separately.
        // The database file is locked exclusively (only one process can grab the
37. lock)
38.     // if !options.ReadOnly.
        // The database file is locked using the shared lock (more than one process
39. may
40.     // hold a lock at the same time) otherwise (options.ReadOnly is set).
41.
42.     // 只读加共享锁、否则加互斥锁
43.     if err := flock(db, mode, !db.readOnly, options.Timeout); err != nil {
44.         _ = db.close()
45.         return nil, err
46.     }
47.
48.     // Default values for test hooks
49.     db.ops.writeAt = db.file.WriteAt
50.
51.     // Initialize the database if it doesn't exist.
52.     if info, err := db.file.Stat(); err != nil {
53.         return nil, err
54.     } else if info.Size() == 0 {
55.         // Initialize new files with meta pages.
56.         // 初始化新db文件
57.         if err := db.init(); err != nil {
58.             return nil, err
59.         }
60.     } else {
61.         // 不是新文件，读取第一页元数据
62.         // Read the first meta page to determine the page size.
63.         // 2^12,正好是4k
64.         var buf [0x1000]byte
65.         if _, err := db.file.ReadAt(buf[:], 0); err == nil {
```

```
66.             // 仅仅是读取了pageSize
67.             m := db.pageInBuffer(buf[:], 0).meta()
68.             if err := m.validate(); err != nil {
69.                 // If we can't read the page size, we can assume it's the same
                    // as the OS -- since that's how the page size was chosen in
70. the
71.                 // first place.
72.                 //
73.                 // If the first page is invalid and this OS uses a different
74.                 // page size than what the database was created with then we
75.                 // are out of luck and cannot access the database.
76.                 db.pageSize = os.Getpagesize()
77.             } else {
78.                 db.pageSize = int(m.pageSize)
79.             }
80.         }
81.     }
82.
83.     // Initialize page pool.
84.     db.pagePool = sync.Pool{
85.         New: func() interface{} {
86.             // 4k
87.             return make([]byte, db.pageSize)
88.         },
89.     }
90.
91.     // Memory map the data file.
92.     // mmap映射db文件数据到内存
93.     if err := db.mmap(options.InitialMmapSize); err != nil {
94.         _ = db.close()
95.         return nil, err
96.     }
97.
98.     // Read in the freelist.
99.     db.freelist = newFreelist()
100.    // db.meta().freelist=2
101.    // 读第二页的数据
102.    // 然后建立起freelist中
103.    db.freelist.read(db.page(db.meta().freelist))
104.
105.    // Mark the database as opened and return.
106.    return db, nil
```

```
107.  }
108.
109.
110.  // init creates a new database file and initializes its meta pages.
111.  func (db *DB) init() error {
112.      // Set the page size to the OS page size.
113.      db.pageSize = os.Getpagesize()
114.
115.      // Create two meta pages on a buffer.
116.      buf := make([]byte, db.pageSize*4)
117.      for i := 0; i < 2; i++ {
118.          p := db.pageInBuffer(buf[:], pgid(i))
119.          p.id = pgid(i)
120.          // 第0页和第1页存放元数据
121.          p.flags = metaPageFlag
122.
123.          // Initialize the meta page.
124.          m := p.meta()
125.          m.magic = magic
126.          m.version = version
127.          m.pageSize = uint32(db.pageSize)
128.          m.freelist = 2
129.          m.root = bucket{root: 3}
130.          m.pgid = 4
131.          m.txid = txid(i)
132.          m.checksum = m.sum64()
133.      }
134.
135.      // Write an empty freelist at page 3.
136.      // 拿到第2页存放freelist
137.      p := db.pageInBuffer(buf[:], pgid(2))
138.      p.id = pgid(2)
139.      p.flags = freelistPageFlag
140.      p.count = 0
141.
142.      // 第三块存放叶子page
143.      // Write an empty leaf page at page 4.
144.      p = db.pageInBuffer(buf[:], pgid(3))
145.      p.id = pgid(3)
146.      p.flags = leafPageFlag
147.      p.count = 0
148.
```

```
149.     // Write the buffer to our data file.
150.     // 写入4页的数据
151.     if _, err := db.ops.writeAt(buf, 0); err != nil {
152.         return err
153.     }
154.     // 刷盘
155.     if err := fdatasync(db); err != nil {
156.         return err
157.     }
158.
159.     return nil
160. }
161.
     // page retrieves a page reference from the mmap based on the current page
     size.
162. func (db *DB) page(id pgid) *page {
163.     pos := id * pgid(db.pageSize)
164.     return (*page)(unsafe.Pointer(&db.data[pos]))
165. }
166.
     // pageInBuffer retrieves a page reference from a given byte array based on the
     current page size.
167. func (db *DB) pageInBuffer(b []byte, id pgid) *page {
168.     return (*page)(unsafe.Pointer(&b[id*pgid(db.pageSize)]))
169. }
170.
     // mmap opens the underlying memory-mapped file and initializes the meta
     references.
171. // minsz is the minimum size that the new mmap can be.
172. func (db *DB) mmap(minsz int) error {
173.     db.mmaplock.Lock()
174.     defer db.mmaplock.Unlock()
175.
176.     info, err := db.file.Stat()
177.     if err != nil {
178.         return fmt.Errorf("mmap stat error: %s", err)
179.     } else if int(info.Size()) < db.pageSize*2 {
180.         return fmt.Errorf("file size too small")
181.     }
182.
183.     // Ensure the size is at least the minimum size.
184.     var size = int(info.Size())
185.     if size < minsz {
```

```
189.          size = minsz
190.      }
191.      size, err = db.mmapSize(size)
192.      if err != nil {
193.          return err
194.      }
195.
196.      // Dereference all mmap references before unmapping.
197.      if db.rwtx != nil {
198.          db.rwtx.root.dereference()
199.      }
200.
201.      // Unmap existing data before continuing.
202.      if err := db.munmap(); err != nil {
203.          return err
204.      }
205.
206.      // Memory-map the data file as a byte slice.
207.      if err := mmap(db, size); err != nil {
208.          return err
209.      }
210.
211.      // Save references to the meta pages.
212.      // 获取元数据信息
213.      db.meta0 = db.page(0).meta()
214.      db.meta1 = db.page(1).meta()
215.
216.      // Validate the meta pages. We only return an error if both meta pages fail
217.      // validation, since meta0 failing validation means that it wasn't saved
218.      // properly -- but we can recover using meta1. And vice-versa.
219.      err0 := db.meta0.validate()
220.      err1 := db.meta1.validate()
221.      if err0 != nil && err1 != nil {
222.          return err0
223.      }
224.
225.      return nil
226.  }
227.
228.  // mmapSize determines the appropriate size for the mmap given the current size
229.  // of the database. The minimum size is 32KB and doubles until it reaches 1GB.
230.  // Returns an error if the new mmap size is greater than the max allowed.
```

```
231. func (db *DB) mmapSize(size int) (int, error) {
232.     // Double the size from 32KB until 1GB.
233.     for i := uint(15); i <= 30; i++ {
234.         if size <= 1<<i {
235.             return 1 << i, nil
236.         }
237.     }
238. 
239.     // Verify the requested size is not above the maximum allowed.
240.     if size > maxMapSize {
241.         return 0, fmt.Errorf("mmap too large")
242.     }
243. 
244.     // If larger than 1GB then grow by 1GB at a time.
245.     sz := int64(size)
246.     if remainder := sz % int64(maxMmapStep); remainder > 0 {
247.         sz += int64(maxMmapStep) - remainder
248.     }
249. 
250.     // Ensure that the mmap size is a multiple of the page size.
251.     // This should always be true since we're incrementing in MBs.
252.     pageSize := int64(db.pageSize)
253.     if (sz % pageSize) != 0 {
254.         sz = ((sz / pageSize) + 1) * pageSize
255.     }
256. 
257.     // If we've exceeded the max size then only grow up to the max size.
258.     if sz > maxMapSize {
259.         sz = maxMapSize
260.     }
261. 
262.     return int(sz), nil
263. }
```

# 第四节 db.View()实现分析

View()主要用来执行只读事务。事务的开启、提交、回滚都交由tx控制。

```
// View executes a function within the context of a managed read-only transaction.
// Any error that is returned from the function is returned from the View() method.
//
// Attempting to manually rollback within the function will cause a panic.
func (db *DB) View(fn func(*Tx) error) error {
    t, err := db.Begin(false)
    if err != nil {
        return err
    }

    // Make sure the transaction rolls back in the event of a panic.
    defer func() {
        if t.db != nil {
            t.rollback()
        }
    }()

    // Mark as a managed tx so that the inner function cannot manually rollback.
    t.managed = true

    // If an error is returned from the function then pass it through.
    err = fn(t)
    t.managed = false
    if err != nil {
        _ = t.Rollback()
        return err
    }

    if err := t.Rollback(); err != nil {
        return err
    }

    return nil
}
```

# 第五节 db.Update()实现分析

Update()主要用来执行读写事务。事务的开始、提交、回滚都交由tx内部控制

```
// Update executes a function within the context of a read-write managed transaction.
// If no error is returned from the function then the transaction is committed.
// If an error is returned then the entire transaction is rolled back.
// Any error that is returned from the function or returned from the commit is
// returned from the Update() method.
//
// Attempting to manually commit or rollback within the function will cause a panic.
func (db *DB) Update(fn func(*Tx) error) error {
    t, err := db.Begin(true)
    if err != nil {
        return err
    }

    // Make sure the transaction rolls back in the event of a panic.
    defer func() {
        if t.db != nil {
            t.rollback()
        }
    }()

    // Mark as a managed tx so that the inner function cannot manually commit.
    t.managed = true

    // If an error is returned from the function then rollback and return error.
    err = fn(t)
    t.managed = false
    if err != nil {
        _ = t.Rollback()
        return err
    }

    return t.Commit()
}
```

# 第六节 db.Batch()实现分析

现在对Batch()方法稍作分析，在DB定义的那一节中我们可以看到，一个DB对象拥有一个batch对象，该对象是全局的。当我们使用Batch()方法时，内部会对将传递进去的fn缓存在calls中。

其内部也是调用了Update，只不过是在Update内部遍历之前缓存的calls。

有两种情况会触发调用Update。

1. 第一种情况是到达了MaxBatchDelay时间，就会触发Update
2. 第二种情况是len(db.batch.calls) >= db.MaxBatchSize，即缓存的calls个数大于等于MaxBatchSize时，也会触发Update。

**Batch**的本质是： 将每次写、每次刷盘的操作转变成了多次写、一次刷盘，从而提升性能。

```
// Batch calls fn as part of a batch. It behaves similar to Update,
// except:
//
// 1. concurrent Batch calls can be combined into a single Bolt
// transaction.
//
// 2. the function passed to Batch may be called multiple times,
// regardless of whether it returns error or not.
//
// This means that Batch function side effects must be idempotent and
// take permanent effect only after a successful return is seen in
// caller.
// 幂等
// The maximum batch size and delay can be adjusted with DB.MaxBatchSize
// and DB.MaxBatchDelay, respectively.
//
// Batch is only useful when there are multiple goroutines calling it.
func (db *DB) Batch(fn func(*Tx) error) error {
    errCh := make(chan error, 1)

    db.batchMu.Lock()
    if (db.batch == nil) || (db.batch != nil && len(db.batch.calls) >=
db.MaxBatchSize) {
        // There is no existing batch, or the existing batch is full; start a
new one.
        db.batch = &batch{
            db: db,
```

```
26.         }
27.         db.batch.timer = time.AfterFunc(db.MaxBatchDelay, db.batch.trigger)
28.     }
29.     db.batch.calls = append(db.batch.calls, call{fn: fn, err: errCh})
30.     if len(db.batch.calls) >= db.MaxBatchSize {
31.         // wake up batch, it's ready to run
32.         go db.batch.trigger()
33.     }
34.     db.batchMu.Unlock()
35.
36.     err := <-errCh
37.     if err == trySolo {
38.         err = db.Update(fn)
39.     }
40.     return err
41. }
42.
43. type call struct {
44.     fn  func(*Tx) error
45.     err chan<- error
46. }
47.
48. type batch struct {
49.     db    *DB
50.     timer *time.Timer
51.     start sync.Once
52.     calls []call
53. }
54.
55. // trigger runs the batch if it hasn't already been run.
56. func (b *batch) trigger() {
57.     b.start.Do(b.run)
58. }
59.
60. // run performs the transactions in the batch and communicates results
61. // back to DB.Batch.
62. func (b *batch) run() {
63.     b.db.batchMu.Lock()
64.     b.timer.Stop()
65.     // Make sure no new work is added to this batch, but don't break
66.     // other batches.
67.     if b.db.batch == b {
```

```
68.         b.db.batch = nil
69.     }
70.     b.db.batchMu.Unlock()
71.
72. retry:
73.     // 内部多次调用Update，最后一次Commit刷盘，提升性能
74.     for len(b.calls) > 0 {
75.         var failIdx = -1
76.         err := b.db.Update(func(tx *Tx) error {
77.             遍历calls中的函数c，多次调用，最后一次提交刷盘
78.             for i, c := range b.calls {
79.                 // safelyCall里面捕获了panic
80.                 if err := safelyCall(c.fn, tx); err != nil {
81.                     failIdx = i
82.                     //只要又失败，事务就不提交
83.                     return err
84.                 }
85.             }
86.             return nil
87.         })
88.
89.         if failIdx >= 0 {
90.             // take the failing transaction out of the batch. it's
91.             // safe to shorten b.calls here because db.batch no longer
92.             // points to us, and we hold the mutex anyway.
93.             c := b.calls[failIdx]
94.             //这儿只是把失败的事务给踢出去了，然后其他的事务会重新执行
                b.calls[failIdx], b.calls = b.calls[len(b.calls)-1],
95. b.calls[:len(b.calls)-1]
                // tell the submitter re-run it solo, continue with the rest of the
96. batch
97.             c.err <- trySolo
98.             continue retry
99.         }
100.
101.        // pass success, or bolt internal errors, to all callers
102.        for _, c := range b.calls {
103.            c.err <- err
104.        }
105.        break retry
106.    }
107. }
108.
```

```
109.  // trySolo is a special sentinel error value used for signaling that a
110.  // transaction function should be re-run. It should never be seen by
111.  // callers.
      var trySolo = errors.New("batch function returned an error and should be re-run
112.  solo")
113.
114.  type panicked struct {
115.      reason interface{}
116.  }
117.
118.  func (p panicked) Error() string {
119.      if err, ok := p.reason.(error); ok {
120.          return err.Error()
121.      }
122.      return fmt.Sprintf("panic: %v", p.reason)
123.  }
124.
125.  func safelyCall(fn func(*Tx) error, tx *Tx) (err error) {
126.      defer func() {
127.          if p := recover(); p != nil {
128.              err = panicked{p}
129.          }
130.      }()
131.      return fn(tx)
132.  }
```

# 第七节 db.allocate()和db.grow()分析

```go
// allocate returns a contiguous block of memory starting at a given page.
func (db *DB) allocate(count int) (*page, error) {
    // Allocate a temporary buffer for the page.
    var buf []byte
    if count == 1 {
        buf = db.pagePool.Get().([]byte)
    } else {
        buf = make([]byte, count*db.pageSize)
    }
    // 转成*page
    p := (*page)(unsafe.Pointer(&buf[0]))
    p.overflow = uint32(count - 1)

    // Use pages from the freelist if they are available.
    // 先从空闲列表中找
    if p.id = db.freelist.allocate(count); p.id != 0 {
        return p, nil
    }

    // 找不到的话，就按照事务的pgid来分配
    // 表示需要从文件内部扩大

    // Resize mmap() if we're at the end.
    p.id = db.rwtx.meta.pgid
    // 因此需要判断是否目前所有的页数已经大于了mmap映射出来的空间
    // 这儿计算的页面总数是从当前的id后还要计算count+1个
    var minsz = int((p.id+pgid(count))+1) * db.pageSize
    if minsz >= db.datasz {
        if err := db.mmap(minsz); err != nil {
            return nil, fmt.Errorf("mmap allocate error: %s", err)
        }
    }

    // Move the page id high water mark.
    // 如果不是从freelist中找到的空间的话，更新meta的id，也就意味着是从文件中新扩展的页
    db.rwtx.meta.pgid += pgid(count)

    return p, nil
```

```
39. }
40.
41. // grow grows the size of the database to the given sz.
42. func (db *DB) grow(sz int) error {
43.     // Ignore if the new size is less than available file size.
44.     if sz <= db.filesz {
45.         return nil
46.     }
47.
48.     // 满足这个条件sz>filesz
49.
        // If the data is smaller than the alloc size then only allocate what's
50. needed.
51.     // Once it goes over the allocation size then allocate in chunks.
52.     if db.datasz < db.AllocSize {
53.         sz = db.datasz
54.     } else {
55.         sz += db.AllocSize
56.     }
57.
58.     // Truncate and fsync to ensure file size metadata is flushed.
59.     // https://github.com/boltdb/bolt/issues/284
60.     if !db.NoGrowSync && !db.readOnly {
61.         if runtime.GOOS != "windows" {
62.             if err := db.file.Truncate(int64(sz)); err != nil {
63.                 return fmt.Errorf("file resize error: %s", err)
64.             }
65.         }
66.         if err := db.file.Sync(); err != nil {
67.             return fmt.Errorf("file sync error: %s", err)
68.         }
69.     }
70.
71.     db.filesz = sz
72.     return nil
73. }
```

# 第八节 总结

本章我们主要介绍了boltdb中最上层的DB对象的知识。首先介绍了DB的定义，然后介绍了下创建DB的Open()以及DB对外暴露的一些接口，这些接口基本上是平常使用最频繁的api。 在介绍了几个接口后，然后逐一对其内部的源码实现进行了分析。其实有了前几节的知识后，再来看这些接口的实现，相对比较简单。因为他们无非就是对之前的Tx、Bucket、node做的 一些封装。底层还是调用的之前介绍的哪些方法。到此我们所有和bolt相关的源码分析就告一段落了。

在第6章也给大家提供了一些其他技术大牛写的源码分析的文章，大家有兴趣可以进一步阅读和学习。

# 第六章 参考资料

1. 阅读 boltDB 源码后的小结
2. 给boltdb源码添加注释仓库
3. boltdb官方仓库
4. 分析boltdb源码的微信公众号文章集合

# 结束

在boltdb中它还自带了一个命令行工具主要功能用来查看boltdb中所有的页以及不同页上的数据信息，以及做性能测试等，后续抽时间也会将该工具支持的功能补充到该文章中。

在没有做这件事情之前，总感觉对框架或者组件的源码分析，基本上停留在给代码加一些注释、画图梳理的层面。当真正自己动手从头到尾来写时，才发现中间有太多太多的细节，需要重新理解和把握。总体来说，这算是一次不错的体验和收获了。

在最后，本文基本上都是按照个人的理解和阅读源码基础上完成的。文章中难免有错误和理解有误的地方，大家看的时候发现问题，可以及时反馈给我，同时欢迎大家一起交流学习。