boltdb源码剖析

1.boltdb简要介绍

1.1 boltdb是什么?

> 待补充

1.2 为什么要分析boltdb?

> 待补充

1.3 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中抽象取来的专属名词(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，第一柜子可以防止衣服，第二个柜子可以防止书籍和笔等。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，当然此处也就是boltdb了，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： 对应我们放在抽泣里的每一件东西。为了方便我们后面使用的时候方便，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**

```go
import "bolt"

func main(){
    // 我们的大柜子
    db, err := bolt.Open("./my.db", 0600, nil)
    if err != nil {
        panic(err)
    }
    defer db.Close()
    // 往db里面插入数据
    err = db.Update(func(tx *bolt.Tx) error {
       //我们的小柜子
        bucket, err := tx.CreateBucketIfNotExists([]byte("user"))
        if err != nil {
            log.Fatalf("CreateBucketIfNotExists err:%s", err.Error())
            return err
        }
        //放入东西
        if err = bucket.Put([]byte("hello"), []byte("world")); err != nil {
            log.Fatalf("bucket Put err:%s", err.Error())
            return err
        }
        return nil
    })
    if err != nil {
        log.Fatalf("db.Update err:%s", err.Error())
    }

    // 从db里面读取数据
    err = db.View(func(tx *bolt.Tx) error {
        //找到柜子
        bucket := tx.Bucket([]byte("user"))
        //找东西
        val := bucket.Get([]byte("hello"))
        log.Printf("the get val:%s", val)
        val = bucket.Get([]byte("hello2"))
        log.Printf("the get val2:%s", val)
        return nil
    })
    if err != nil {
        log.Fatalf("db.View err:%s", err.Error())
    }
}
```

1.4 boltdb的整体数据组织结构

1.5 boltdb的黑科技

mmap
b+树
嵌套bucket

2.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，移动磁臂是一种机械运动，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按照顺序读的方式加载到内存中，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。

这里先给大家科普一点：

set操作： 本质上对应的是 node->page->file的过程

get操作： 本质上对应的是 file->page->node的过程

2.1 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。

> 待补充图

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

```go
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。

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
)
```

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。

```
// typ returns a human readable page type string used for debugging.
func (p *page) typ() string {
    if (p.flags & branchPageFlag) != 0 {
        return "branch"
    } else if (p.flags & leafPageFlag) != 0 {
        return "leaf"
    } else if (p.flags & metaPageFlag) != 0 {
        return "meta"
    } else if (p.flags & freelistPageFlag) != 0 {
        return "freelist"
    }
    return fmt.Sprintf("unknown<%02x>", p.flags)
}
```

下面我们一一对其数据结构进行分析。

2.2 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) {
    if m.root.root >= m.pgid {
        panic(fmt.Sprintf("root bucket pgid (%d) above high water mark (%d)", m.root
    } else if m.freelist >= m.pgid {
        panic(fmt.Sprintf("freelist pgid (%d) above high water mark (%d)", m.freelis
    }

    // Page id is either going to be 0 or 1 which we can determine by the transactic
    //指定页id和页类型
    p.id = pgid(m.txid % 2)
    p.flags |= metaPageFlag

    // Calculate the checksum.
    m.checksum = m.sum64()
  // 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
    m.copy(p.meta())
}


// copy copies one meta object to another.
func (m *meta) copy(dest *meta) {
    *dest = *m
}


// generates the checksum for the meta.
func (m *meta) sum64() uint64 {
    var h = fnv.New64a()
    _, _ = h.Write((*[unsafe.Offsetof(meta{}.checksum)]byte)(unsafe.Pointer(m))[:])
    return h.Sum64()
}
```

2. page->meta

```
page.go

// 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))
}
```

2.2 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。

```go
freelist.go

// 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 transactio
type freelist struct {
	// 已经可以被分配的空闲页
	ids     []pgid          // all free and available free page ids.
	// 将来很快能被释放的空闲页，部分事务可能在读或者写
	pending map[txid][]pgid // mapping of soon-to-be free page ids by tx.
	cache   map[pgid]bool   // fast lookup of all free and pending page ids.
}

// newFreelist returns an empty, initialized freelist.
func newFreelist() *freelist {
	return &freelist{
		pending: make(map[txid][]pgid),
		cache:   make(map[pgid]bool),
	}
}
```

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

```go
// 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
}
```

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

```
//从磁盘中的page初始化freelist
// read initializes the freelist from a freelist page.
func (f *freelist) read(p *page) {
	// If the page.count is at the max uint16 value (64k) then it's considered
	// an overflow and the size of the freelist is stored as the first element.
	idx, count := 0, int(p.count)
	if count == 0xFFFF {
		idx = 1
		// 用第一个uint64来存储整个count的值
		count = int(((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[0])
	}

	// Copy the list of page ids from the freelist.
	if count == 0 {
		f.ids = nil
	} else {
		ids := ((*[maxAllocSize]pgid)(unsafe.Pointer(&p.ptr)))[idx:count]
		f.ids = make([]pgid, len(ids))
		copy(f.ids, ids)

		// Make sure they're sorted.
		sort.Sort(pgids(f.ids))
	}

	// Rebuild the page cache.
	f.reindex()
}
```

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 given si
// If a contiguous block cannot be found then 0 is returned.
// [5,6,7,13,14,15,16,18,19,20,31,32]
// 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
func (f *freelist) allocate(n int) pgid {
	if len(f.ids) == 0 {
		return 0
	}

	var initial, previd pgid
	for i, id := range f.ids {
		if id <= 1 {
			panic(fmt.Sprintf("invalid page allocation: %d", id))
		}
```

```
        // Reset initial page if this is not contiguous.
        // id-previd != 1 来判断是否连续
        if previd == 0 || id-previd != 1 {
            // 第一次不连续时记录一下第一个位置
            initial = id
        }

        // If we found a contiguous block then remove it and return it.
        // 找到了连续的块，然后将其返回即可
        if (id-initial)+1 == pgid(n) {
            // If we're allocating off the beginning then take the fast path
            // and just adjust the existing slice. This will use extra memory
            // temporarily but the append() in free() will realloc the slice
            // as is necessary.
            if (i + 1) == n {
                // 找到的是前n个连续的空间
                f.ids = f.ids[i+1:]
            } else {
                copy(f.ids[i-n+1:], f.ids[i+1:])
                f.ids = f.ids[:len(f.ids)-n]
            }

            // Remove from the free cache.
            // 同时更新缓存
            for i := pgid(0); i < pgid(n); i++ {
                delete(f.cache, initial+i)
            }

            return initial
        }

        previd = id
    }
    return 0
}
```

2.4 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：

```go
// branchPageElement represents a node on a branch page.
type branchPageElement struct {
	pos   uint32 //该元信息和真实key之间的偏移量
	ksize uint32
	pgid  pgid
}

// key returns a byte slice of the node key.
func (n *branchPageElement) key() []byte {
	buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
	// pos~ksize
	return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos]))[:n.ksize]
}
```

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

```go
// branchPageElement retrieves the branch node by index
func (p *page) branchPageElement(index uint16) *branchPageElement {
	return &((*[0x7FFFFFF]branchPageElement)(unsafe.Pointer(&p.ptr)))[index]
}

// branchPageElements retrieves a list of branch nodes.
func (p *page) branchPageElements() []branchPageElement {
	if p.count == 0 {
		return nil
	}
	return ((*[0x7FFFFFF]branchPageElement)(unsafe.Pointer(&p.ptr)))[:]
}
```

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。

```go
// node represents an in-memory, deserialized page.
type node struct {
    bucket     *Bucket
    isLeaf     bool
    unbalanced bool
    spilled    bool
    key        []byte
    pgid       pgid
    parent     *node
    children   nodes
    inodes     inodes
}
```

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。

```go
const (
    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].i

// 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 {
```

```go
        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.ino
    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.Pointe
            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.Pointe
            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()
}
```

2.5 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。

```
// leafPageElement represents a node on a leaf page.
// 叶子节点既存储key，也存储value
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。flags1
    pos   uint32
    ksize uint32
    vsize uint32
}

// 叶子节点的key
// key returns a byte slice of the node key.
func (n *leafPageElement)  key() []byte {
    buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
    // pos~ksize
    return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos]))[:n.ksize:n.ksize]
}

// 叶子节点的value
// value returns a byte slice of the node value.
func (n *leafPageElement) value() []byte {
    buf := (*[maxAllocSize]byte)(unsafe.Pointer(n))
    // key:pos~ksize
    // value:pos+ksize~pos+ksize+vsize
    return (*[maxAllocSize]byte)(unsafe.Pointer(&buf[n.pos+n.ksize]))[:n.vsize:n.vsi
}
```

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()

```
// leafPageElement retrieves the leaf node by index
func (p *page) leafPageElement(index uint16) *leafPageElement {
	n := &((*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[index]

	// 最原始的指针：unsafe.Pointer(&p.ptr)
	// 将其转为(*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr))
	// 然后再取第index个元素 ((*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[i
	// 最后返回该元素指针 &((*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[inde

	// ((*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))
	// (*[0x7FFFFFF]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 ((*[0x7FFFFFF]leafPageElement)(unsafe.Pointer(&p.ptr)))[:]
}
```

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

2.6 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。

3.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！

3.1 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

```go
// 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 enough,
// then its root page can be stored inline in the "value", after the bucket
// header. In the case of inline buckets, the "root" will be 0.
type bucket struct {
	root     pgid   // page id of the bucket's root-level page
	sequence uint64 // monotonically incrementing, used by NextSequence()
}

// newBucket returns a new bucket associated with a transaction.
func newBucket(tx *Tx) Bucket {
	var b = Bucket{tx: tx, FillPercent: DefaultFillPercent}
	if tx.writable {
		b.buckets = make(map[string]*Bucket)
		b.nodes = make(map[pgid]*node)
	}
	return b
}
```

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。

3.2 Bucket遍历之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。而找这个行为就是交由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。

```go
// Cursor creates a cursor associated with the bucket.
// The cursor is only valid as long as the transaction is open.
// Do not use a cursor after the transaction is closed.
func (b *Bucket) Cursor() *Cursor {
    // Update transaction statistics.
    b.tx.stats.CursorCount++

    // Allocate and return a cursor.
    return &Cursor{
        bucket: b,
        stack:  make([]elemRef, 0),
    }
}
```

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 a buc
// Cursors see nested buckets with value == nil.
// Cursors can be obtained from a transaction and are valid as long as the transacti
//
// Keys and values returned from the cursor are only valid for the life of the trans
//
// 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**。

```go
// First moves the cursor to the first item in the bucket and returns its key and va
// 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) First() (key []byte, value []byte)

// Last moves the cursor to the last item in the bucket and returns its key and valu
// 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)

// Next moves the cursor to the next item in the bucket and returns its key and valu
// If the cursor is at the end of the bucket 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) Next() (key []byte, value []byte)

// Prev moves the cursor to the previous item in the bucket and returns its key and
// If the cursor is at the beginning of the bucket then a nil key and value are retu
// The returned key and value are only valid for the life of the transaction.
func (c *Cursor) Prev() (key []byte, value []byte)

// 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 writab
func (c *Cursor) Delete() error

// Seek moves the cursor to a given key and returns it.
// If the key does not exist then the next key is used. If no keys
// follow, a nil key is returned.
// The returned key and value are only valid for the life of the transaction.
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。

```go
// Seek moves the cursor to a given key and returns it.
// If the key does not exist then the next key is used. If no keys
// follow, a nil key is returned.
// The returned key and value are only valid for the life of the transaction.
func (c *Cursor) Seek(seek []byte) (key []byte, value []byte) {
    k, v, flags := c.seek(seek)

    // If we ended up after the last element of a page then move to the next 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，就直接返回n
    // 这里需要返回下一个
    if ref := &c.stack[len(c.stack)-1]; ref.index >= ref.count() {
        k, v, flags = c.next()
    }

    if k == nil {
        return nil, nil
        //  子桶的话
    } else if (flags & uint32(bucketLeafFlag)) != 0 {
        return k, nil
    }
    return k, v
}

// seek moves the cursor to a given key and returns it.
// If the key does not exist then the next key is used.
func (c *Cursor) seek(seek []byte) (key []byte, value []byte, flags uint32) {
    _assert(c.bucket.tx.db != nil, "tx closed")

    // Start from root page/node and traverse to correct page.
    c.stack = c.stack[:0]
    // 开始根据seek的key值搜索root
    c.search(seek, c.bucket.root)
    // 执行完搜索后，stack中保存了所遍历的路径
    ref := &c.stack[len(c.stack)-1]

    // If the cursor is pointing to the end of page/node then return nil.
    if ref.index >= ref.count() {
        return nil, nil, 0
    }
    //获取值
    // If this is a bucket then return a nil value.
    return c.keyValue()
}



// keyValue returns the key and value of the current leaf element.
func (c *Cursor) keyValue() ([]byte, []byte, uint32) {
  //最后一个节点为叶子节点
    ref := &c.stack[len(c.stack)-1]
    if ref.count() == 0 || ref.index >= ref.count() {
        return nil, nil, 0
    }
    // Retrieve value from node.
    // 先从内存中取
    if ref.node != nil {
        inode := &ref.node.inodes[ref.index]
        return inode.key, inode.value, inode.flags
    }

    // 其次再从文件page中取
    // Or retrieve value from page.
    elem := ref.page.leafPageElement(uint16(ref.index))
    return elem.key(), elem.value(), elem.flags
}
```

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。

```
// 从根节点开始遍历
// search recursively performs a binary search against a given page/node until it fi
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.
```

```go
// 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
    }

    // Check the node cache for non-inline buckets.
    if b.nodes != nil {
        if n := b.nodes[id]; n != nil {
            return nil, n
        }
    }

    // Finally lookup the page from the transaction if no node is materialized.
    return b.tx.page(id), nil
}


//node中搜索
func (c *Cursor) searchNode(key []byte, n *node) {
    var exact bool
    //二分搜索
    index := sort.Search(len(n.inodes), func(i int) bool {
        // TODO(benbjohnson): Optimize this range search. It's a bit hacky right now
        // sort.Search() finds the lowest index where f() != -1 but we need the high
        ret := bytes.Compare(n.inodes[i].key, key)
        if ret == 0 {
            exact = true
        }
        return ret != -1
    })
    if !exact && index > 0 {
        index--
    }
    c.stack[len(c.stack)-1].index = index

    // Recursively search to the next page.
    c.search(key, n.inodes[index].pgid)
}
```

```go
//页中搜索
func (c *Cursor) searchPage(key []byte, p *page) {
	// Binary search for the correct range.
	inodes := p.branchPageElements()

	var exact bool
	index := sort.Search(int(p.count), func(i int) bool {
		// TODO(benbjohnson): Optimize this range search. It's a bit hacky right now
		// sort.Search() finds the lowest index where f() != -1 but we need the high
		ret := bytes.Compare(inodes[i].key(), key)
		if ret == 0 {
			exact = true
		}
		return ret != -1
	})
	if !exact && index > 0 {
		index--
	}
	c.stack[len(c.stack)-1].index = index

	// Recursively search to the next page.
	c.search(key, inodes[index].pgid)
}


// nsearch searches the leaf node on the top of the stack for a key.
// 搜索叶子页
func (c *Cursor) nsearch(key []byte) {
	e := &c.stack[len(c.stack)-1]
	p, n := e.page, e.node

	// If we have a node then search its inodes.
	// 先搜索node
	if n != nil {
		//又是二分搜索
		index := sort.Search(len(n.inodes), func(i int) bool {
			return bytes.Compare(n.inodes[i].key, key) != -1
		})
		e.index = index
		return
	}

	// If we have a page then search its leaf elements.
	// 再搜索page
	inodes := p.leafPageElements()
	index := sort.Search(int(p.count), func(i int) bool {
		return bytes.Compare(inodes[i].key(), key) != -1
```

```
    })
    e.index = index
}
```

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

```go
// First moves the cursor to the first item in the bucket and returns its key and va
// 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) First() (key []byte, value []byte) {
    _assert(c.bucket.tx.db != nil, "tx closed")
    // 清空stack
    c.stack = c.stack[:0]
    p, n := c.bucket.pageNode(c.bucket.root)

    // 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
    ref := elemRef{page: p, node: n, index: 0}
    c.stack = append(c.stack, ref)


    c.first()

    // If we land on an empty page then move to the next value.
    // https://github.com/boltdb/bolt/issues/450
    // 当前页时空的话，找下一个
    if c.stack[len(c.stack)-1].count() == 0 {
        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

```go
// Last moves the cursor to the last item in the bucket and returns its key and valu
// 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")

	c.stack = c.stack[:0]
	p, n := c.bucket.pageNode(c.bucket.root)

	ref := elemRef{page: p, node: n}
	// 设置其index为当前页元素的最后一个
	ref.index = ref.count() - 1
	c.stack = append(c.stack, ref)

	c.last()

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

// last moves the cursor to the last leaf element under the last page in the stack.
// 移动到栈中最后一个节点的最后一个叶子节点
func (c *Cursor) last() {
	for {
		// Exit when we hit a leaf page.
		ref := &c.stack[len(c.stack)-1]
		if ref.isLeaf() {
			break
		}

		// Keep adding pages pointing to the last element in 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)

		var nextRef = elemRef{page: p, node: n}
		nextRef.index = nextRef.count() - 1
		c.stack = append(c.stack, nextRef)
	}
}
```

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

```go
// Next moves the cursor to the next item in the bucket and returns its key and valu
// If the cursor is at the end of the bucket 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) Next() (key []byte, value []byte) {
    _assert(c.bucket.tx.db != nil, "tx closed")
    k, v, flags := c.next()
    if (flags & uint32(bucketLeafFlag)) != 0 {
        return k, nil
    }
    return k, v
}

// 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 nil.
func (c *Cursor) next() (key []byte, value []byte, flags uint32) {
    for {
        // Attempt to move over one element until we're successful.
        // Move up the stack as we hit the end of each page in our stack.
        var i int
        for i = len(c.stack) - 1; i >= 0; i-- {
            elem := &c.stack[i]
            if elem.index < elem.count()-1 {
                // 元素还有时，往后移动一个
                elem.index++
                break
            }
        }
        // 最后的结果elem.index++

        // If we've hit the root page then stop and return. This will leave the
        // cursor on the last element of the last page.
        // 所有页都遍历完了
        if i == -1 {
            return nil, nil, 0
        }

        // Otherwise start from where we left off in the stack and find the
        // first element of the first leaf page.
        // 剩余的节点里面找，跳过原先遍历过的节点
        c.stack = c.stack[:i+1]
        // 如果是叶子节点，first()啥都不做，直接退出。返回elem.index+1的数据
        // 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
        c.first()

        // If this is an empty page then restart and move back up the stack.
        // https://github.com/boltdb/bolt/issues/450
        if c.stack[len(c.stack)-1].count() == 0 {
            continue
        }

        return c.keyValue()
    }
}
```

Prev()实现

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

    // Attempt to move back one element until we're successful.
    // Move up the stack as we hit the beginning of each page in our stack.
    for i := len(c.stack) - 1; i >= 0; i-- {
        elem := &c.stack[i]
        if elem.index > 0 {
            // 往前移动一格
            elem.index--
            break
        }
        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 bran
    // 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

```go
// 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 writab
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
}

// 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.ir

	// 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:]...)

	// Mark the node as needing rebalancing.
	n.unbalanced = true
}
```

3.3 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。在

3.3.1 node节点的定义

```go
// node represents an in-memory, deserialized page.
type node struct {
    bucket     *Bucket // 关联一个桶
    isLeaf     bool
    unbalanced bool   // 值为true的话，需要考虑页合并
    spilled    bool   // 值为true的话，需要考虑页分裂
    key        []byte // 对于分支节点的话，保留的是最小的key
    pgid       pgid   // 分支节点关联的页id
    parent     *node  // 该节点的parent
    children   nodes  // 该节点的孩子节点
    inodes     inodes // 该节点上保存的索引数据
}

// 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
```

3.3.2 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

```go
// 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.inc
    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.Pointe
            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.Pointe
            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)

```go
// 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.
	} 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.ir

	// Add capacity and shift nodes if we don't have an exact match and need to inse
	exact := (len(n.inodes) > 0 && index < len(n.inodes) && bytes.Equal(n.inodes[inc
	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)

```go
// 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.ir

    // 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:]...)

    // Mark the node as needing rebalancing.
    n.unbalanced = true
}
```

nextSibling()、prevSibling()

```go
// nextSibling returns the next node with the same parent.
// 返回下一个兄弟节点
func (n *node) nextSibling() *node {
    if n.parent == nil {
        return nil
    }
    index := n.parent.childIndex(n)
    if index >= n.parent.numChildren()-1 {
        return nil
    }
    return n.parent.childAt(index + 1)
}

// prevSibling returns the previous node with the same parent.
// 返回上一个兄弟节点
func (n *node) prevSibling() *node {
    if n.parent == nil {
        return nil
    }
    // 首先找下标
    index := n.parent.childIndex(n)
    if index == 0 {
        return nil
    }
    // 然后返回
    return n.parent.childAt(index - 1)
}

// childAt returns the child node at a given index.
// 只有树枝节点才有孩子
func (n *node) childAt(index int) *node {
	if n.isLeaf {
		panic(fmt.Sprintf("invalid childAt(%d) on a leaf node", index))
	}
	return n.bucket.node(n.inodes[index].pgid, n)
}

// node creates a node from a page and associates it with a given parent.
// 根据pgid创建一个node
func (b *Bucket) node(pgid pgid, parent *node) *node {
	_assert(b.nodes != nil, "nodes map expected")

	// Retrieve node if it's already been created.
	if n := b.nodes[pgid]; n != nil {
		return n
	}

	// Otherwise create a node and cache it.
	n := &node{bucket: b, parent: parent}
	if parent == nil {
		b.rootNode = n
	} else {
		parent.children = append(parent.children, n)
	}

	// Use the inline page if this is an inline bucket.
	// 如果第二次进来，b.page不为空
	// 此处的pgid和b.page只会有一个是有值的。
	var p = b.page
	// 说明不是内联桶
	if p == nil {
		p = b.tx.page(pgid)
	}

	// Read the page into the node and cache it.
	n.read(p)
	// 缓存
	b.nodes[pgid] = n

	// Update statistics.
	b.tx.stats.NodeCount++

	return n
}
```

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.

```go
// spill writes the nodes to dirty pages and splits nodes as it goes.
// Returns an error if dirty pages cannot be allocated.
func (n *node) spill() error {
    var tx = n.bucket.tx
    if n.spilled {
        return nil
    }

    // Spill child nodes first. Child nodes can materialize sibling nodes in
    // the case of split-merge so we cannot use a range loop. We have to check
    // the children size on every loop iteration.
    sort.Sort(n.children)
    for i := 0; i < len(n.children); i++ {
        if err := n.children[i].spill(); err != nil {
            return err
        }
    }

    // We no longer need the child list because it's only used for spill tracking.
    n.children = nil

    // Split nodes into appropriate sizes. The first node will always be n.
    // 将当前的node进行拆分成多个node
    var nodes = n.split(tx.db.pageSize)
    for _, node := range nodes {
        // Add node's page to the freelist if it's not new.
        if node.pgid > 0 {
            tx.db.freelist.free(tx.meta.txid, tx.page(node.pgid))
            node.pgid = 0
        }

        // Allocate contiguous space for the node.
        p, err := tx.allocate((node.size() / tx.db.pageSize) + 1)
        if err != nil {
            return err
        }

        // Write the node.
        if p.id >= tx.meta.pgid {
            // 不可能发生
            panic(fmt.Sprintf("pgid (%d) above high water mark (%d)", p.id, tx.meta.
        }
        node.pgid = p.id
        node.write(p)
        // 已经拆分过了
        node.spilled = true

        // Insert into parent inodes.
        if node.parent != nil {
            var key = node.key
            if key == nil {
                key = node.inodes[0].key
            }

            // 放入父亲节点中
            node.parent.put(key, node.inodes[0].key, nil, node.pgid, 0)
            node.key = node.inodes[0].key
            _assert(len(node.key) > 0, "spill: zero-length node key")
        }

        // Update the statistics.
        tx.stats.Spill++
    }

    // 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 respill.
    if n.parent != nil && n.parent.pgid == 0 {
        n.children = nil
        return n.parent.spill()
    }

    return nil
}

// split breaks up a node into multiple smaller nodes, if appropriate.
// This should only be called from the spill() function.
```

```go
func (n *node) split(pageSize int) []*node {
    var nodes []*node

    node := n
    for {
        // Split node into two.
        a, b := node.splitTwo(pageSize)
        nodes = append(nodes, a)

        // If we can't split then exit the loop.
        if b == nil {
            break
        }

        // Set node to b so it gets split on the next iteration.
        node = b
    }

    return nodes
}

// splitTwo breaks up a node into two smaller nodes, if appropriate.
// This should only be called from the split() function.
func (n *node) splitTwo(pageSize int) (*node, *node) {
    // Ignore the split if the page doesn't have at least enough nodes for
    // two pages or if the nodes can fit in a single page.
    // 太小的话，就不拆分了
    if len(n.inodes) <= (minKeysPerPage*2) || n.sizeLessThan(pageSize) {
        return n, nil
    }

    // Determine the threshold before starting a new node.
    var fillPercent = n.bucket.FillPercent
    if fillPercent < minFillPercent {
        fillPercent = minFillPercent
    } else if fillPercent > maxFillPercent {
        fillPercent = maxFillPercent
    }
    threshold := int(float64(pageSize) * fillPercent)

    // Determine split position and sizes of the two pages.
    splitIndex, _ := n.splitIndex(threshold)

    // Split node into two separate nodes.
    // If there's no parent then we'll need to create one.
    if n.parent == nil {
        n.parent = &node{bucket: n.bucket, children: []*node{n}}
    }
    // Create a new node and add it to the parent.
    // 拆分出一个新节点
    next := &node{bucket: n.bucket, isLeaf: n.isLeaf, parent: n.parent}
    n.parent.children = append(n.parent.children, next)

    // Split inodes across two nodes.
    next.inodes = n.inodes[splitIndex:]
    n.inodes = n.inodes[:splitIndex]

    // Update the statistics.
    n.bucket.tx.stats.Split++

    return n, next
}

// splitIndex finds the position where a page will fill a given threshold.
// It returns the index as well as the size of the first page.
// This is only be called from split().
// 找到合适的index
func (n *node) splitIndex(threshold int) (index, sz int) {
    sz = pageHeaderSize

    // Loop until we only have the minimum number of keys required for the second pa
    for i := 0; i < len(n.inodes)-minKeysPerPage; i++ {
        index = i
        inode := n.inodes[i]
        elsize := n.pageElementSize() + len(inode.key) + len(inode.value)

        // If we have at least the minimum number of keys and adding another
        // node would put us over the threshold then exit and return.
        if i >= minKeysPerPage && sz+elsize > threshold {
            break
        }

        // Add the element size to the total size.
        sz += elsize
    }

    return
}
```

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

```go
// 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.
// 填充率太低或者没有足够的key时，进行页合并
func (n *node) rebalance() {
    if !n.unbalanced {
        return
    }
    n.unbalanced = false

    // Update statistics.
    n.bucket.tx.stats.Rebalance++

    // Ignore if node is above threshold (25%) and has enough keys.
    var threshold = n.bucket.tx.db.pageSize / 4
    if n.size() > threshold && len(n.inodes) > n.minKeys() {
        return
    }

    // Root node has special handling.
    if n.parent == nil {
        // If root node is a branch and only has one node then collapse it.
        if !n.isLeaf && len(n.inodes) == 1 {
            // Move root's child up.
            child := n.bucket.node(n.inodes[0].pgid, n)
            n.isLeaf = child.isLeaf
            n.inodes = child.inodes[:]
            n.children = child.children

            // Reparent all child nodes being moved.
            for _, inode := range n.inodes {
                if child, ok := n.bucket.nodes[inode.pgid]; ok {
                    child.parent = n
                }
            }

            // Remove old child.
            child.parent = nil
            delete(n.bucket.nodes, child.pgid)
            child.free()
        }

        return
    }

    // If node has no keys then just remove it.
    if n.numChildren() == 0 {
        n.parent.del(n.key)
        n.parent.removeChild(n)
        delete(n.bucket.nodes, n.pgid)
        n.free()
        n.parent.rebalance()
        return
    }

    _assert(n.parent.numChildren() > 1, "parent must have at least 2 children")

    // Destination node is right sibling if idx == 0, otherwise left sibling.
    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，否则的话，就找
    var useNextSibling = (n.parent.childIndex(n) == 0)
    if useNextSibling {
        target = n.nextSibling()
    } else {
        target = n.prevSibling()
    }

    // If both this node and the target node are too small then merge them.
    // 合并当前node和target，target合到node
    if useNextSibling {
        // Reparent all child nodes being moved.
        for _, inode := range target.inodes {
            if child, ok := n.bucket.nodes[inode.pgid]; ok {
                // 之前的父亲移除该孩子
                child.parent.removeChild(child)
                // 重新指定父亲节点
                child.parent = n
                // 父亲节点指当前孩子
                child.parent.children = append(child.parent.children, child)
            }
        }

        // Copy over inodes from target and remove target.
        n.inodes = append(n.inodes, target.inodes...)
        n.parent.del(target.key)
        n.parent.removeChild(target)
        delete(n.bucket.nodes, target.pgid)
        target.free()
    } else {
        // node合到targett
        // Reparent all child nodes being moved.
        for _, inode := range n.inodes {
            if child, ok := n.bucket.nodes[inode.pgid]; ok {
                child.parent.removeChild(child)
                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
    n.parent.rebalance()
}
```

3.4 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。

```
// CreateBucketIfNotExists creates a new bucket if it doesn't already exist and retu
// Returns an error if the bucket name is blank, or if the bucket name is too long.
// The bucket instance is only valid for the lifetime of the transaction.
func (b *Bucket) CreateBucketIfNotExists(key []byte) (*Bucket, error) {
    child, err := b.CreateBucket(key)
    if err == ErrBucketExists {
        return b.Bucket(key), nil
    } else if err != nil {
        return nil, err
    }
    return child, nil
}

// CreateBucket creates a new bucket at the given key and returns the new bucket.
// Returns an error if the key already exists, if the bucket name is blank, or if th
// The bucket instance is only valid for the lifetime of the transaction.
func (b *Bucket) CreateBucket(key []byte) (*Bucket, error) {
    if b.tx.db == nil {
        return nil, ErrTxClosed
    } else if !b.tx.writable {
        return nil, ErrTxNotWritable
    } else if len(key) == 0 {
        return nil, ErrBucketNameRequired
    }

    // Move cursor to correct position.
    // 拿到游标
    c := b.Cursor()
    // 开始遍历、找到合适的位置
    k, _, flags := c.seek(key)

    // Return an error if there is an existing key.
    if bytes.Equal(key, k) {
        // 是桶,已经存在了
        if (flags & bucketLeafFlag) != 0 {
            return nil, ErrBucketExists
        }
        // 不是桶、但key已经存在了
        return nil, ErrIncompatibleValue
    }

    // Create empty, inline bucket.
    var bucket = Bucket{
        bucket:      &bucket{},
        rootNode:    &node{isLeaf: true},
        FillPercent: DefaultFillPercent,
    }
    // 拿到bucket对应的value
    var value = bucket.write()

    // Insert into node.
    key = cloneBytes(key)
    // 插入到inode中
    // c.node()方法会在内存中建立这棵树，调用n.read(page)
    c.node().put(key, key, value, 0, bucketLeafFlag)

    // Since subbuckets are not allowed on inline buckets, we need to
    // dereference the inline page, if it exists. This will cause the bucket
    // to be treated as a regular, non-inline bucket for the rest of the tx.
    b.page = nil

    //根据key获取一个桶
    return b.Bucket(key), nil
}
```

```go
// write allocates and writes a bucket to a byte slice.
// 内联桶的话，其value中bucketHeaderSize后面的内容为其page的数据
func (b *Bucket) write() []byte {
    // Allocate the appropriate size.
    var n = b.rootNode
    var value = make([]byte, bucketHeaderSize+n.size())

    // Write a bucket header.
    var bucket = (*bucket)(unsafe.Pointer(&value[0]))
    *bucket = *b.bucket

    // Convert byte slice to a fake page and write the root node.
    var p = (*page)(unsafe.Pointer(&value[bucketHeaderSize]))
    // 将该桶中的元素压缩存储，放在value中
    n.write(p)

    return value
}


// node returns the node that the cursor is currently positioned on.
func (c *Cursor) node() *node {
    _assert(len(c.stack) > 0, "accessing a node with a zero-length cursor stack")

    // If the top of the stack is a leaf node then just return it.
    if ref := &c.stack[len(c.stack)-1]; ref.node != nil && ref.isLeaf() {
        return ref.node
    }

    // Start from root and traverse down the hierarchy.
    var n = c.stack[0].node
    if n == nil {
        n = c.bucket.node(c.stack[0].page.id, nil)
    }
    // 非叶子节点
    for _, ref := range c.stack[:len(c.stack)-1] {
        _assert(!n.isLeaf, "expected branch node")
        n = n.childAt(int(ref.index))
    }
    _assert(n.isLeaf, "expected leaf node")
    return n
}

// 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.
    } 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.ir

    // Add capacity and shift nodes if we don't have an exact match and need to inse
    exact := (len(n.inodes) > 0 && index < len(n.inodes) && bytes.Equal(n.inodes[inc
    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")
}
```

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。

```
// Bucket retrieves a nested bucket by name.
// Returns nil if the bucket does not exist.
// The bucket instance is only valid for the lifetime of the transaction.
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

```go
// DeleteBucket deletes a bucket at the given key.
// Returns an error if the bucket does not exists, or if the key represents a non-bu
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
    }

    // Recursively delete all child buckets.
    child := b.Bucket(key)
    // 将该桶下面的所有桶都删除
    err := child.ForEach(func(k, v []byte) error {
        if v == nil {
            if err := child.DeleteBucket(k); err != nil {
                return fmt.Errorf("delete bucket: %s", err)
            }
        }
        return nil
    })
    if err != nil {
        return err
    }

    // Remove cached copy.
    delete(b.buckets, string(key))

    // Release all bucket pages to freelist.
    child.nodes = nil
    child.rootNode = nil
    child.free()

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

// 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.ir

    // 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:]...)

    // Mark the node as needing rebalancing.
    n.unbalanced = true
}

// free recursively frees all pages in the bucket.
func (b *Bucket) free() {
    if b.root == 0 {
        return
    }

    var tx = b.tx
    b.forEachPageNode(func(p *page, n *node, _ int) {
        if p != nil {
            tx.db.freelist.free(tx.meta.txid, p)
        } else {
            n.free()
        }
    })
    b.root = 0
}
```

3.5 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

```go
// Put sets the value for a key in the bucket.
// If the key exist then its previous value will be overwritten.
// Supplied value must remain valid for the life of the transaction.
// Returns an error if the bucket was created from a read-only transaction,
// if the key is blank, if the key is too large, or if the value is too large.
func (b *Bucket) Put(key []byte, value []byte) error {
    if b.tx.db == nil {
        return ErrTxClosed
    } else if !b.Writable() {
        return ErrTxNotWritable
    } else if len(key) == 0 {
        return ErrKeyRequired
    } else if len(key) > MaxKeySize {
        return ErrKeyTooLarge
    } else if int64(len(value)) > MaxValueSize {
        return ErrValueTooLarge
    }

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

    // Return an error if there is an existing key with a bucket value.
    if bytes.Equal(key, k) && (flags&bucketLeafFlag) != 0 {
        return ErrIncompatibleValue
    }

    // Insert into node.
    key = cloneBytes(key)
    c.node().put(key, key, value, 0, 0)

    return nil
}
```

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

```go
// 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
}
```

3.6 Bucket的页分裂、页合并

4.boltdb的核心接口分析

1. CreateBucket(name)
2. Bucket(name)
3. val=Get(key)
4. Put(key,value)
5. Delete(key)

5.boltdb事务控制

Begin()

Commit()

Rollback()

6.boltdb DB分析

7.参考资料