前言

```
# 除了测试文件，共有 41 个 golang 文件
find . -name "*.go" ! -name "*_test.go" | wc -l
# 接近 6000 行代码
find . -name "*.go" ! -name "*_test.go" | xargs cloc
# 但是真正需要看的，只有这 11 个文件
files=(
    db.go
    page.go
    node.go
    unsafe.go
    freelist.go
    freelist_hmap.go
    bucket.go
    cursor.go
    tx.go
    tx_check.go
    errors.go
)
# 这就只有 3300 了，少了接近一半
cloc ${files}
```

- db.go
  - 数据库对外提供的服务
- page.go node.go
  - page: 数据在磁盘中的形式
  - node: 数据在内存中的形式
- freelist.go freelist_hmap.go
  - 内存如何管理的
- bucket.go cursor.go
  - bucket: 数据如何组织的
  - cursor: 数据如何遍历的
- tx.go tx_check.go
  - 事务的实现
- unsafe.go
  - 封装了 unsafe 的操作
  - 主要用于 node ⇄ page
- errors.go
  - 定义一系列错误

BoltDB 的数据组织方式

每一个 Bucket 对应一棵 B+ 树

先看一下 Bucket 的定义

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

type Bucket struct {
  *bucket
    // 当前 Bucket 关联的事务
```

```
  tx       *Tx                // the associated transaction
    // 子 Bucket
  buckets  map[string]*Bucket // subbucket cache
    // 关联的 page
  page     *page              // inline page reference
    // Bucket 管理的树的根节点
  rootNode *node              // materialized node for the root page.
    // 缓存
  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
    // 与B+树的分裂相关
  FillPercent float64
}
```

Bucket 中重要的工具 Cursor 游标

bolt 中没有使用传统的 B+ 树中将叶子节点使用链表串起来的方式, 而是使用的 cursor 游标, 通过路径回溯的方式, 来支持范围查询.

所以 cursor 对于 bucket 中 curd 还是听重要的.

```
type elemRef struct {
    // 当前位置对应的 page
  page  *page
    // 当前位置对应的 node
    // 可能是没有被反序列化的
    // 但是没关系, 使用page可以序列化
  node  *node
    // 在第几个 kv 对
  index int
}
type Cursor struct {
  bucket *Bucket
  stack  []elemRef
}
```

BoltDB 在磁盘中的形式

bolt 在磁盘中是什么形式

下面描述一下 bolt 的文件在磁盘中是什么形式存储的, 也就是真正在磁盘中的时候是什么样子的

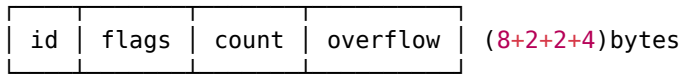
page 的类型

- meta page
- freelist page
- branch page
- leaf page

Head

定义于 page.go 中, 每一个 page 都共有的结构

这里只描述一次, 后面的各种类型的 page 就不说明这一部分了




```
type pgid uint64
type page struct {
  id       pgid   // 每个页的唯一id
  flags    uint16 // 类型
  count    uint16 // 页中元素的数量
  overflow uint32 // 数据是否有溢出(主要是freelist page中有用)
}
```

meta page 的内容

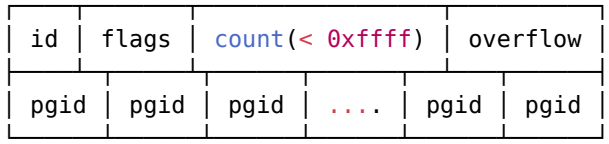
元信息页, 定义于 db.go 中, 主要的作用是管理整个数据库必要信息

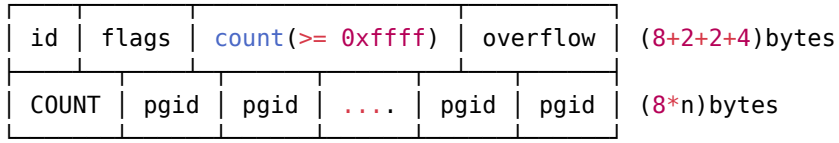
```
type meta struct {
  magic    uint32 // 魔数
  version  uint32 // 版本号(固定为2)
  pageSize uint32 // 页大小(4KB)
  flags    uint32
  root     bucket
  freelist pgid   // 空闲列表页的 page id
  pgid     pgid
  txid     txid   // 数据库事务操作序号
  checksum uint64 // data数据的hash摘要, 用于判断data是否损坏
}
```

freelist page

定义于 freelist.go 中

磁盘中的格式

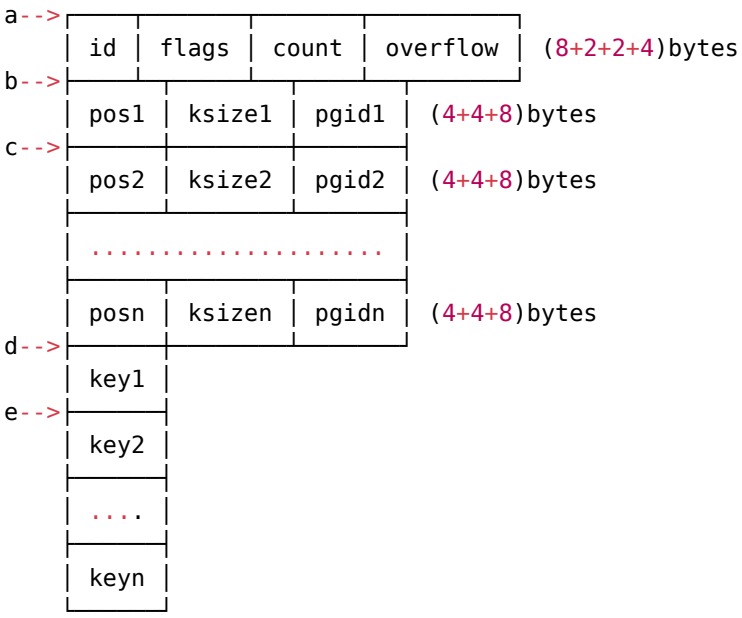




branch page 的内容

定义在 page.go 中

```
type branchPageElement struct {
  pos   uint32
  ksize uint32
  pgid  pgid
}
```



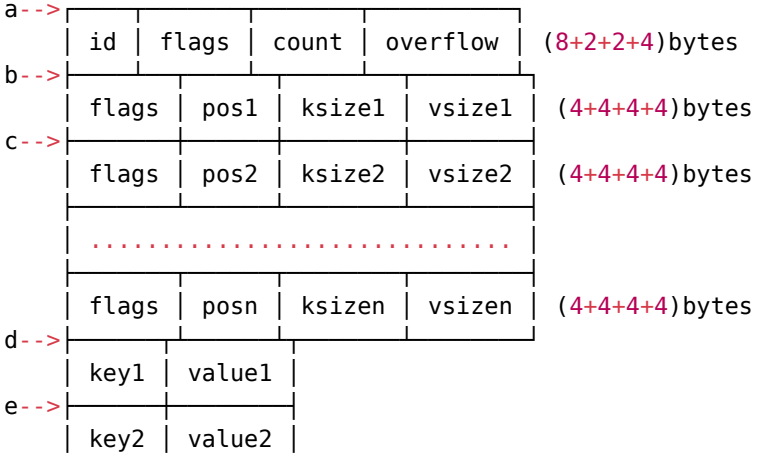
pos 实际上就是 key 相对于 head 结尾的偏移量

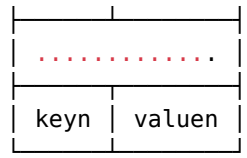
如图:

```
a = 0
b = 16*8
c = 16*8 + 16*8
pos1 = d-b
pos2 = e-c
```

leaf page 的内容

```
// page.go
type leafPageElement struct {
    // 标识当前的节点是否是 bucket 类型
    // page.go 定义了 bucketLeafFlag = 0x01
    // 0 不是 bucket 类型
    // 1 是 bucket 类型
  flags uint32
  pos   uint32
  ksize uint32
  vsize uint32
}
```





pos 实际上就是 kv 相对于 head 结尾的偏移量 如图:

```
a = 0
b = 16*8
c = 16*8 + 16*8
pos1 = d-b
pos2 = e-c
```

BoltDB 在内存中的形式

page 是物理存储的基本单位, 那么 node 就是在逻辑存储的基本单位. 也是在内存中存储的单位

位于 node.go 下的 type node struct{}

node 只是描述了 branch/leaf page 在内存中的格式

而 meta/freelist page 在内存中是在 DB 结构体中, 他们也有专门的结构体 type meta struct{} 以及 type freelist struct{}. 分别位于 db.go 与 freelist.go 中

```
type nodes []*node
type inode struct {
    // branch page OR leaf page
  flags uint32
    // 如果类型是 branch page 时
    // 表示的是 子节点的 page id
  pgid  pgid
  key   []byte
    // 如果类型是 leaf page 时
    // 表示的是 kv 对中的值(value)
  value []byte
}
type inodes []inode

type node struct {
    // 该 node 节点位于哪个 bucket
  bucket     *Bucket
    // 是否是叶子节点
    // 对应到 page header 中的 flags 字段
  isLeaf     bool
    // 是否平衡
  unbalanced bool
    // 是否需要分裂
  spilled    bool
    // 节点最小的 key
  key        []byte
    // 节点对应的 page id
  pgid       pgid
    // 当前节点的父节点
  parent     *node
    // 当前节点的孩子节点
    // 在 spill rebalance 过程中使用
  children   nodes
```

```
    // 节点中存储的数据
    // 广义的 kv 数据(v 可能是子节点)
    // page header 中的 count 可以通过 len(inodes) 获得
    // branch/leaf 的数据都在 inodes 中可以体现到
  inodes     inodes
}
```

node 以及 page 的转换(branch/leaf page)

在这里可以清楚的看到 branch/leaf 的磁盘内存的转换过程

```
// node.go

// 读入
func (n *node) read(p *page)
// 落盘
func (n *node) write(p *page)
```

node 以及 page 的转换(meta page)

```
// 内存中的 meta 结构
type meta struct {
  magic    uint32
  version  uint32
  pageSize uint32
  flags    uint32
    // 对应一个 root bucret
  root     bucket
  freelist pgid
  pgid     pgid
  txid     txid
  checksum uint64
}

// 读入
// db启动的时候就会先读入
func (db *DB) mmap(minsz int) (err error) {
    // ....
    db.meta0 = db.page(0).meta()
    db.meta1 = db.page(1).meta()
    // ...
}
func (p *page) meta() *meta {
  return (*meta)(unsafeAdd(unsafe.Pointer(p), unsafe.Sizeof(*p)))
}

// 落盘
// 每一次事务 commit 的时候都会调用
// Note: read only不会改变 meta, 只有 read-write 会改变 meta 信息
// (tx *Tx)Commit -> (tx *Tx)writeMeta -> (m *meta)write
// db.go
func (m *meta) write(p *page)
```

node 以及 page 的转换(freelist page)

```
// 读入 freelist.go
func (f *freelist) read(p *page)
// 落盘 freelist.go
func (f *freelist) write(p *page) error
```

事务

参考

1. https://github.com/ZhengHe-MD/learn-bolt
2. https://www.bookstack.cn/books/jaydenwen123-boltdb_book
3. https://youjiali1995.github.io/storage/boltdb/
4. https://brunocalza.me/but-how-exactly-databases-use-mmap/
5. https://mp.weixin.qq.com/mp/homepage?__biz=MzAwMjgwMTEzNw==&hid=7&sn=a6de7eb747a2c666fe7a69c49ddddb16&scene=18#wechat_redirect
6. 微信公众号: 小徐先生的编程世界
7. 微信公众号: 数据小冰