LocalCache 学习小记

什么是 LocalCache，go 中常见的 LocalCache 有哪些?

LocalCache 顾名思义，是本地高速缓存，不是分布式的，用于加快获取数据的速度或存储一些重要但不需要持久化数据。
为什么需要 LocalCache?
常见的比如有 Redis 中的热 key 问题。热 key 问题
go 中常见的 LocalCache 有：bigcache、freecache、ccache 等。

BigCache

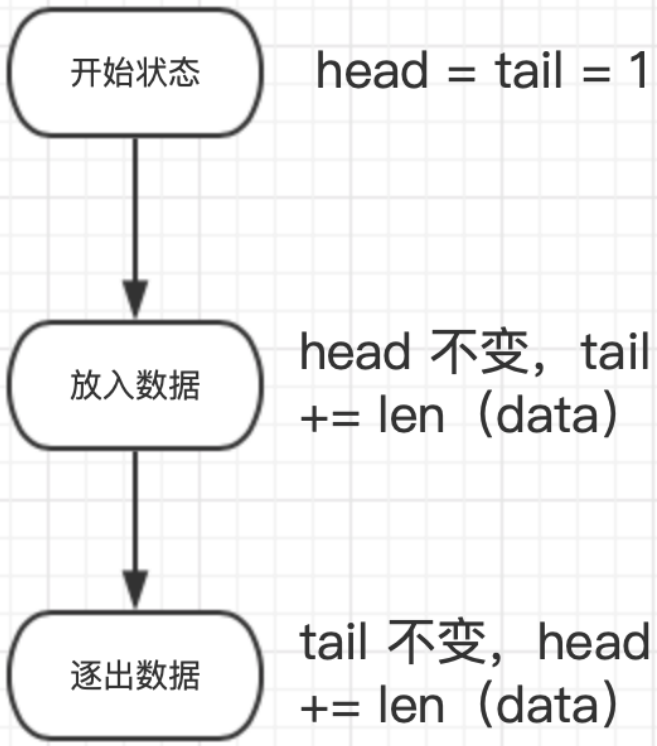
为什么 BigCache 速度快?

a. 使用分片技术，减小锁的竞争，降低延迟（这里的分片 x 必须是 2 的幂次方，可以通过幂运算取余，hash & （x - 1））。
b. 忽略高额 GC 开销，对于Go语言中的map, 垃圾回收器会检查 map 中的每一个元素, 如果缓存中包含数百万的缓存对象，垃圾回收器对这些对象的无意义的检查导致不必要的时间开销。Go 的开发者优化了垃圾回收时对于map的处理，如果map对象中的key和value不包含指针，那么垃圾回收器就会对它们进行优化，不进行垃圾回收的操作。详细信息请参考：修复xt说明

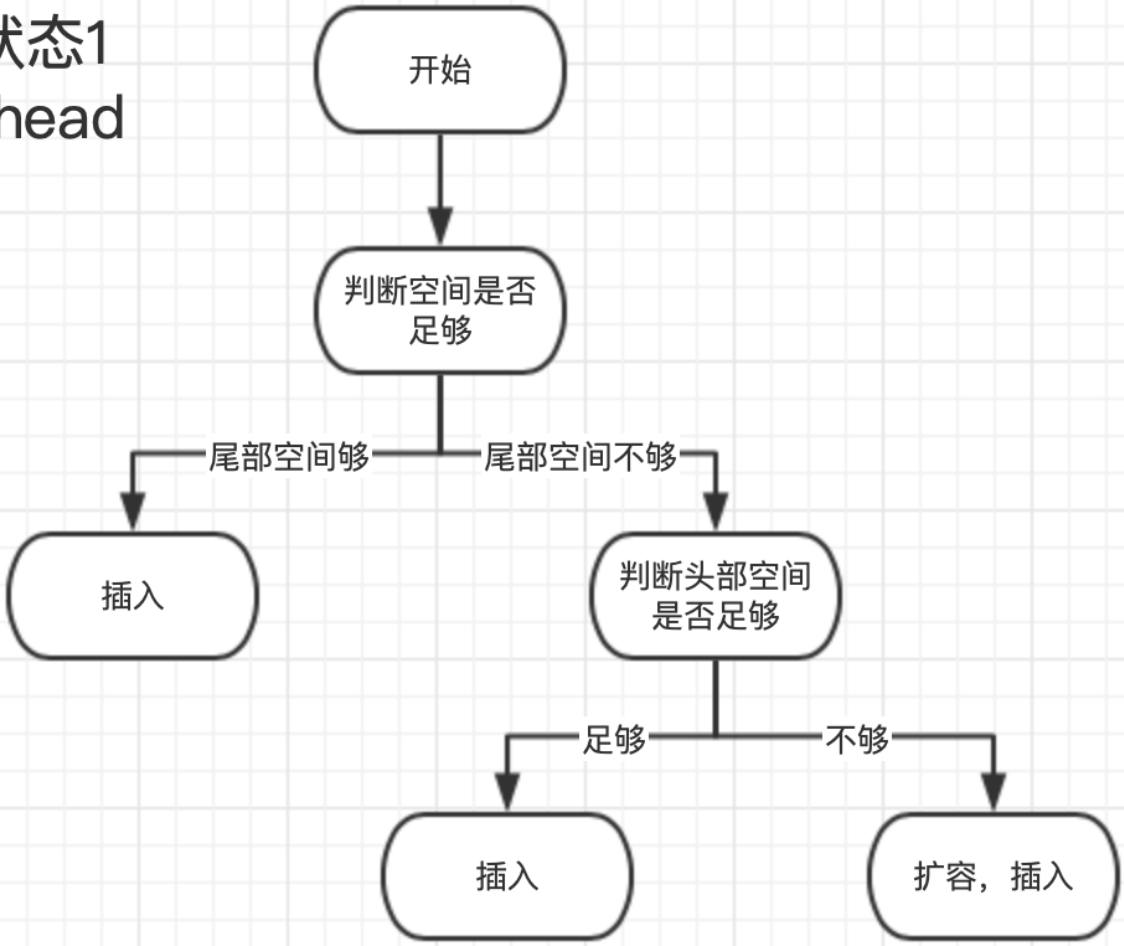
存储模块

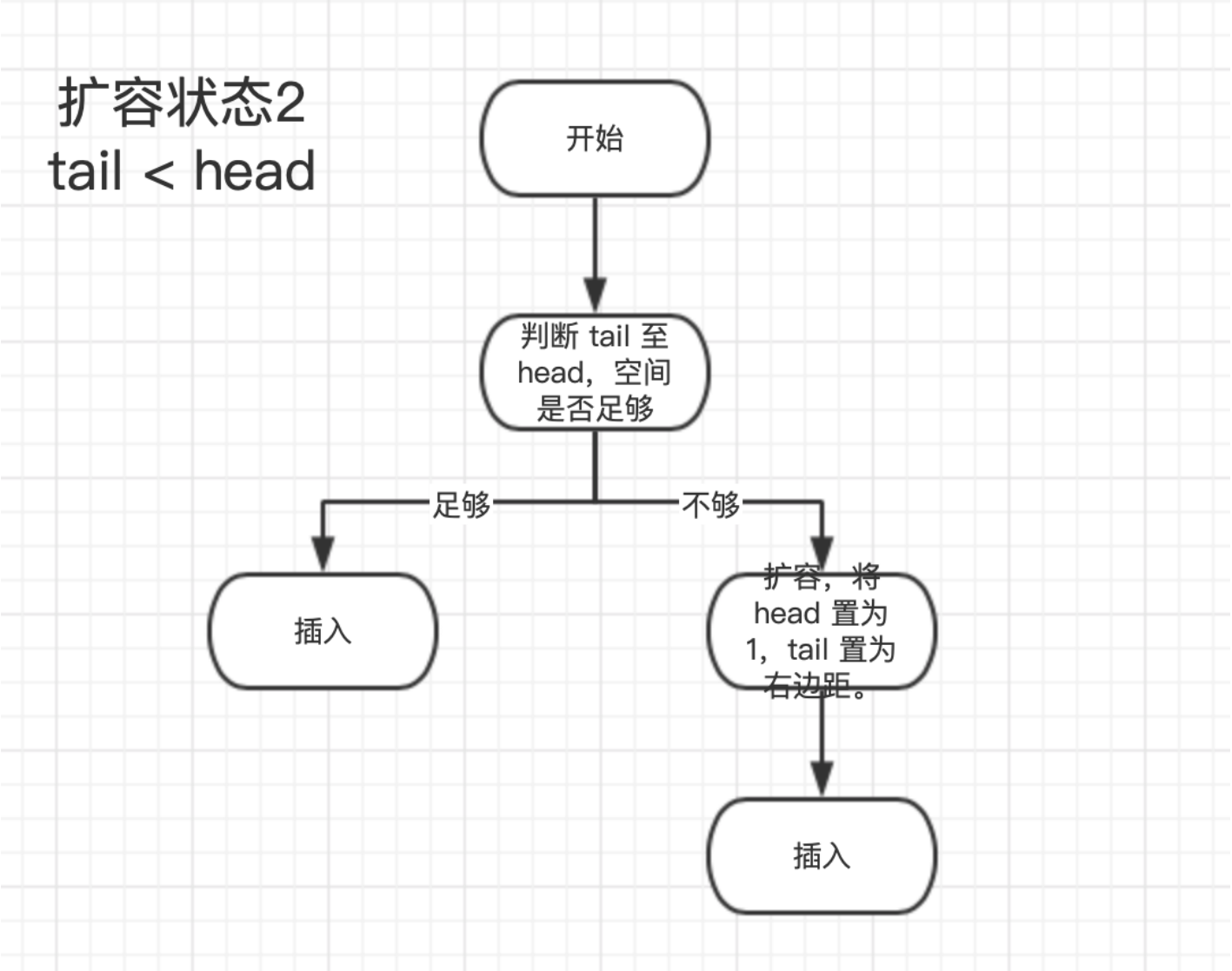
存储模块类似一个可扩容的环形数组。参考请看：http://note.youdao.com/noteshare?id=e1b94e53e050c86160626ac043efce3c

基本状态


开始状态
head = tail = 1
放入数据
head 不变，tail
+= len（data）
逐出数据
tail 不变，head
+= len（data）

扩容状态1
tail > head





BytesQueue 结构体

```
type BytesQueue struct {
    array           []byte
    capacity        int
    maxCapacity     int
    head            int
    tail            int
    count           int
    // 右边距
    rightMargin     int
    headerBuffer    []byte
    verbose         bool
    initialCapacity int
}
```

push 操作

```
func (q *BytesQueue) Push(data []byte) (int, error) {
   dataLen := len(data)
    // 判断前后是否有可用空间，没有则扩容。
   if q.availableSpaceAfterTail() < dataLen+headerEntrySize {
      if q.availableSpaceBeforeHead() >= dataLen+headerEntrySize {
         q.tail = leftMarginIndex
      } else if q.capacity+headerEntrySize+dataLen >= q.maxCapacity &&
q.maxCapacity > 0 {
         return -1, &queueError{"Full queue. Maximum size limit reached."}
      } else {
         q.allocateAdditionalMemory(dataLen + headerEntrySize)
      }
   }

   index := q.tail

   q.push(data, dataLen)

   return index, nil
}
```

Pop 操作

```
func (q *BytesQueue) Pop() ([]byte, error) {
   data, size, err := q.peek(q.head)
   if err != nil {
      return nil, err
   }

   q.head += headerEntrySize + size
   q.count--

   if q.head == q.rightMargin {
      q.head = leftMarginIndex
      if q.tail == q.rightMargin {
```

```
            q.tail = leftMarginIndex
        }
        q.rightMargin = q.tail
    }

    return data, nil
}
```

插入操作

```
func (s *cacheShard) set(key string, hashedKey uint64, entry []byte) error {
    currentTimestamp := uint64(s.clock.epoch())

    s.lock.Lock()
    // 可以改进

    if previousIndex := s.hashmap[hashedKey]; previousIndex != 0 {
        if previousEntry, err := s.entries.Get(int(previousIndex)); err == nil {
            resetKeyFromEntry(previousEntry)
        }
    }

    if oldestEntry, err := s.entries.Peek(); err == nil {
        s.onEvict(oldestEntry, currentTimestamp, s.removeOldestEntry)
    }

    w := wrapEntry(currentTimestamp, hashedKey, key, entry, &s.entryBuffer)

    for {
        if index, err := s.entries.Push(w); err == nil {
            s.hashmap[hashedKey] = uint32(index)
            s.lock.Unlock()
            return nil
        }
        if s.removeOldestEntry(NoSpace) != nil {
```

```
            s.lock.Unlock()
            return fmt.Errorf("entry is bigger than max shard size")
        }
    }
}
```

删除操作

共有五钟删除操作。

第一种：set 后若 hashkey 中有值，则执行删除操作（软删除，不改变底层数据）。

```
if previousIndex := s.hashmap[hashedKey]; previousIndex != 0 {
    if previousEntry, err := s.entries.Get(int(previousIndex)); err == nil {
        resetKeyFromEntry(previousEntry)
    }
}
```

第二种：set 后调用`onEvict`，若头部的数据超时，硬删除。

```
if oldestEntry, err := s.entries.Peek(); err == nil {
    s.onEvict(oldestEntry, currentTimestamp, s.removeOldestEntry)
}
```

第三种：调用 shard.go 中的`del`，同理，也是软删除。

```
delete(s.hashmap, hashedKey)
s.onRemove(wrappedEntry, Deleted)
if s.statsEnabled {
    delete(s.hashmapStats, hashedKey)
}
resetKeyFromEntry(wrappedEntry)
```

第四种：定时删除，此删除方式是硬删除。详细流程请看：http://note.youdao.com/noteshare?id=962dc90d8a7713537559c1bf55e23bc4

```
if config.CleanWindow > 0 {
    go func() {
        ticker := time.NewTicker(config.CleanWindow)
        defer ticker.Stop()
        for {
            select {
```

```
        case t := <-ticker.C:
            cache.cleanUp(uint64(t.Unix()))
        case <-cache.close:
            return
        }
    }
  }()
}
```

第五种：这一种删除方式一般不会被调用，故放在最后，是硬删除。这种删除方式不会在意是不是超时，而是会直接删除原有数据。

```
for {
  if index, err := s.entries.Push(w); err == nil {
    s.hashmap[hashedKey] = uint32(index)
    s.lock.Unlock()
    return nil
  }
  if s.removeOldestEntry(NoSpace) != nil {
    s.lock.Unlock()
    return fmt.Errorf("entry is bigger than max shard size")
  }
}
```

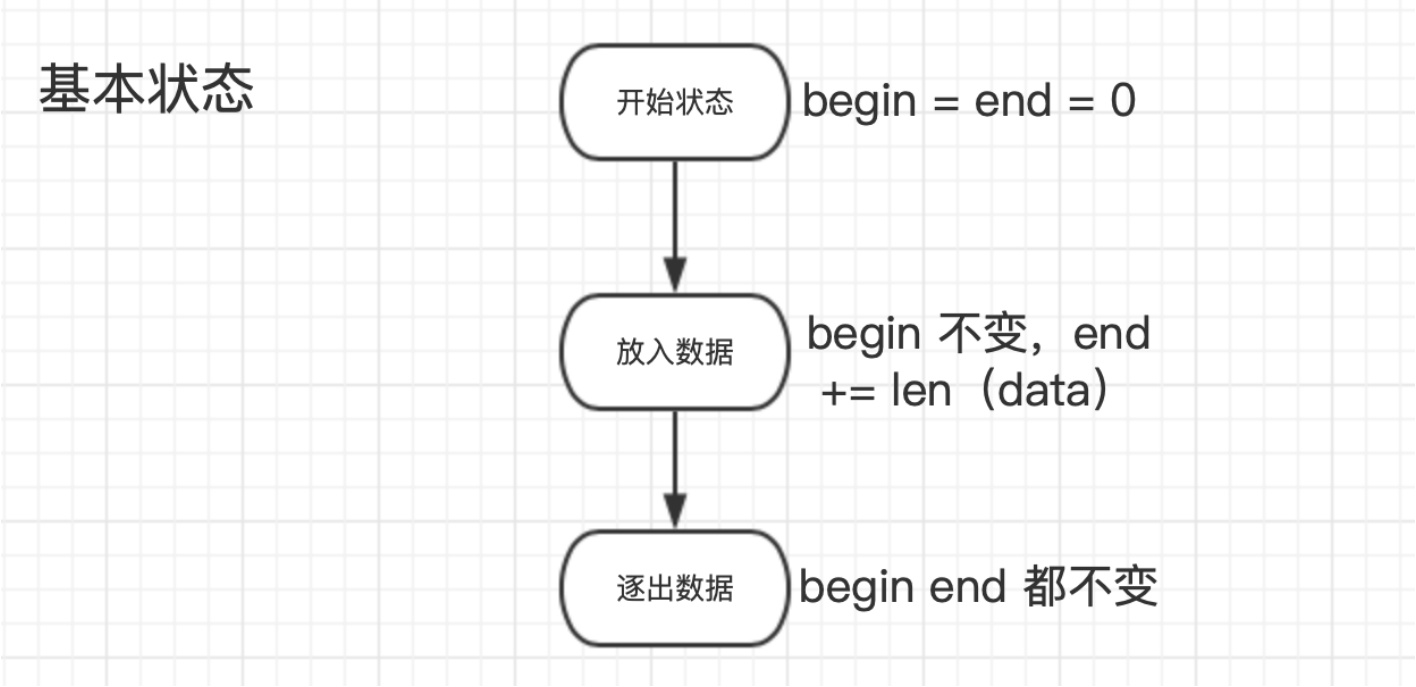
FreeCache

为什么 FreeCache 速度快?

a. 和 bigcache 一样，使用分片技术，但是分片数量固定，为 256 个。
b. 忽略高额 GC 开销，和 bigcache 类似，但是由于 freecache 实现早，当时没有 map 的优化方案，故 freecache 需要基于切片的映射（耗时长），才能将 hashkey 转换到相对应的 entryPtr 上。

存储模块

存储模块是一个环形数组，不可扩容（这就是 freecache 使用比 bigcache 相对少的原因）。



LRU 操作

这是 freecache 中的精髓，将最久未使用的、超时的数据删除。

```
func (seg *segment) evacuate(entryLen int64, slotId uint8, now uint32)
(slotModified bool) {
    var oldHdrBuf [ENTRY_HDR_SIZE]byte
    consecutiveEvacuate := 0
    for seg.vacuumLen < entryLen {
        oldOff := seg.rb.End() + seg.vacuumLen - seg.rb.Size()
        seg.rb.ReadAt(oldHdrBuf[:], oldOff)
        oldHdr := (*entryHdr)(unsafe.Pointer(&oldHdrBuf[0]))
        oldEntryLen := ENTRY_HDR_SIZE + int64(oldHdr.keyLen) + int64(oldHdr.valCap)
        if oldHdr.deleted {
            consecutiveEvacuate = 0
            atomic.AddInt64(&seg.totalTime, -int64(oldHdr.accessTime))
            atomic.AddInt64(&seg.totalCount, -1)
            seg.vacuumLen += oldEntryLen
            continue
        }
        expired := oldHdr.expireAt != 0 && oldHdr.expireAt < now
        // 使用的是近似的 lru 算法，当 accessTime * totalCount < totalTime 时，为 true
```

```
18        leastRecentUsed := int64(oldHdr.accessTime)*atomic.LoadInt64(&seg.totalCount) <= atomic.LoadInt64(&seg.totalTime)
19        if expired || leastRecentUsed || consecutiveEvacuate > 5 {
20           seg.delEntryPtrByOffset(oldHdr.slotId, oldHdr.hash16, oldOff)
21           if oldHdr.slotId == slotId {
22              slotModified = true
23           }
24           consecutiveEvacuate = 0
25           atomic.AddInt64(&seg.totalTime, -int64(oldHdr.accessTime))
26           atomic.AddInt64(&seg.totalCount, -1)
27           seg.vacuumLen += oldEntryLen
28           if expired {
29              atomic.AddInt64(&seg.totalExpired, 1)
30           }
31        } else {
32           // 撤离最近访问过的旧条目，以提高缓存命中率。
33           // evacuate an old entry that has been accessed recently for better cache hit rate.
34           // newOff 是这个的偏移量而已。
35           newOff := seg.rb.Evacuate(oldOff, int(oldEntryLen))
36           seg.updateEntryPtr(oldHdr.slotId, oldHdr.hash16, oldOff, newOff)
37           consecutiveEvacuate++
38           atomic.AddInt64(&seg.totalEvacuate, 1)
39        }
40     }
41     return
42  }
```

插入操作

```
1  func (seg *segment) set(key, value []byte, hashVal uint64, expireSeconds int) (err error) {
2     if len(key) > 65535 {
3        return ErrLargeKey
4     }
```

```
    // fmt.Println("seg.vacuumLen is:", seg.vacuumLen)
    maxKeyValLen := len(seg.rb.data)/4 - ENTRY_HDR_SIZE
    if len(key)+len(value) > maxKeyValLen {
        // Do not accept large entry.
        // unix.Exit(0)
        return ErrLargeEntry
    }
    now := seg.timer.Now()
    expireAt := uint32(0)
    if expireSeconds > 0 {
        expireAt = now + uint32(expireSeconds)
    }

    slotId := uint8(hashVal >> 8)
    hash16 := uint16(hashVal >> 16)

    var hdrBuf [ENTRY_HDR_SIZE]byte
    hdr := (*entryHdr)(unsafe.Pointer(&hdrBuf[0]))

    slot := seg.getSlot(slotId)
    idx, match := seg.lookup(slot, hash16, key)
    // fmt.Println(match)
    if match {
        matchedPtr := &slot[idx]
        seg.rb.ReadAt(hdrBuf[:], matchedPtr.offset)
        hdr.slotId = slotId
        hdr.hash16 = hash16
        hdr.keyLen = uint16(len(key))
        originAccessTime := hdr.accessTime
        hdr.accessTime = now
        hdr.expireAt = expireAt
        hdr.valLen = uint32(len(value))
        if hdr.valCap >= hdr.valLen {
            //in place overwrite 覆盖
```

```
            atomic.AddInt64(&seg.totalTime, int64(hdr.accessTime)-
int64(originAccessTime))
            seg.rb.WriteAt(hdrBuf[:], matchedPtr.offset)
            seg.rb.WriteAt(value, matchedPtr.offset+ENTRY_HDR_SIZE+int64(hdr.keyLen))
            atomic.AddInt64(&seg.overwrites, 1)
            return
        }
        // avoid unnecessary memory copy.
        seg.delEntryPtr(slotId, slot, idx)
        match = false
        // increase capacity and limit entry len.
        for hdr.valCap < hdr.valLen {
            hdr.valCap *= 2
        }
        if hdr.valCap > uint32(maxKeyValLen-len(key)) {
            hdr.valCap = uint32(maxKeyValLen - len(key))
        }
    } else {
        hdr.slotId = slotId
        hdr.hash16 = hash16
        hdr.keyLen = uint16(len(key))
        hdr.accessTime = now
        hdr.expireAt = expireAt
        hdr.valLen = uint32(len(value))
        hdr.valCap = uint32(len(value))
        if hdr.valCap == 0 { // avoid infinite loop when increasing capacity.
            hdr.valCap = 1
        }
    }

    // 判断长度，如果空间不够就删除
    entryLen := ENTRY_HDR_SIZE + int64(len(key)) + int64(hdr.valCap)
    slotModified := seg.evacuate(entryLen, slotId, now)
    // fmt.Println(slotModified)
    if slotModified {
```

```
73        // the slot has been modified during evacuation, we need to looked up for
    the 'idx' again.
74        // otherwise there would be index out of bound error.
75        slot = seg.getSlot(slotId)
76        idx, match = seg.lookup(slot, hash16, key)
77        // assert(match == false)
78    }
79
80    newOff := seg.rb.End()
81    fmt.Println(newOff)
82    seg.insertEntryPtr(slotId, hash16, newOff, idx, hdr.keyLen)
83    seg.rb.Write(hdrBuf[:])
84    seg.rb.Write(key)
85    seg.rb.Write(value)
86    seg.rb.Skip(int64(hdr.valCap - hdr.valLen))
87    atomic.AddInt64(&seg.totalTime, int64(now))
88    atomic.AddInt64(&seg.totalCount, 1)
89    seg.vacuumLen -= entryLen
90    return
91 }
```

删除操作

共有三钟删除操作。

第一种：set 后若 hashkey 中有值，`len(oldData) < len(newData)`则执行覆盖操作。

```
1 if hdr.valCap >= hdr.valLen {
2    //in place overwrite 覆盖
3    atomic.AddInt64(&seg.totalTime, int64(hdr.accessTime)-int64(originAccessTime))
4    seg.rb.WriteAt(hdrBuf[:], matchedPtr.offset)
5    seg.rb.WriteAt(value, matchedPtr.offset+ENTRY_HDR_SIZE+int64(hdr.keyLen))
6    atomic.AddInt64(&seg.overwrites, 1)
7    return
8 }
```

第二种：set 后若 hashkey 中有值，`len(oldData) > len(newData)`则执行软删除操作。

```
1 func (seg *segment) delEntryPtr(slotId uint8, slot []entryPtr, idx int) {
```

```
    offset := slot[idx].offset
    var entryHdrBuf [ENTRY_HDR_SIZE]byte
    seg.rb.ReadAt(entryHdrBuf[:], offset)
    entryHdr := (*entryHdr)(unsafe.Pointer(&entryHdrBuf[0]))
    entryHdr.deleted = true
    seg.rb.WriteAt(entryHdrBuf[:], offset)
    // 将这个 entry 的 slot 覆盖
    copy(slot[idx:], slot[idx+1:])
    seg.slotLens[slotId]--
    atomic.AddInt64(&seg.entryCount, -1)
}
```

第三种：LRU 删除，类似于 bigcache 的 Pop，将头部的删除（如果头部可用，移后即可），留出空间。

比较 bigcache 和 freecache

bigcache 的 readme.md 写的比较详细了：https://github.com/allegro/bigcache#benchmarks
因为 bigcache 的效率高，故一般建议使用 bigcache。

个人愚见，将 bigcache 和 freecache 结合起来，说不定对应特定的场景效率会更高。
具体做法是将 bigcache 的 map[int64]int32 和 freecache 的近似 LRU 的删除策略结合起来。

参考资料

https://github.com/allegro/bigcache
https://github.com/ghorges/mybigcache 结合自己对 bigcache 的理解，写了一个自己的 bigcahce demo。
https://allegro.tech/2016/03/writing-fast-cache-service-in-go.html
https://colobu.com/2019/11/18/how-is-the-bigcache-is-fast
https://github.com/coocood/freecache
https://www.jianshu.com/p/67d8c8511e8e 这篇文章主要是基础性的，科普性的。
https://neojos.com/blog/2018/2018-08-19-%E6%9C%AC%E5%9C%B0%E7%BC%93%E5%AD%98bigcache/