链滴

golang~map 的学习与理解

作者：xhaoxiong

原文链接：https://ld246.com/article/1533916370874

来源网站：链滴

基于golang 1.9

本文主要包括map的基本**数据数据结构**和**平时可能碰到的一些坑**

map的基本数据结构

map的底层机构是hmap(hashmap，**无法对key值排序遍历**,核心元素是一由若干个桶(bucket,结构为bmap)组成的数组,每个bucket可以放若干个元素(通常是8个)，key通过哈希算法被归入到不同的bucket中。当超过8个元素需要存入bucket中时,hmap会使用extra中的overflow来扩展该bucket。下面是hmap的结构体

```
// A header for a Go map.
type hmap struct {
    // Note: the format of the Hmap is encoded in ../../cmd/internal/gc/reflect.go and
    // ../reflect/type.go. Don't change this structure without also changing that code!
    count     int // # live cells == size of map.  Must be first (used by len() builtin)
    flags     uint8
    B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
    noverflow uint16 // approximate number of overflow buckets; see incrnoverflow for details
    hash0     uint32 // hash seed

    buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==0.
    oldbuckets unsafe.Pointer // previous bucket array of half the size, non-nil only when growing
    nevacuate  uintptr        // progress counter for evacuation (buckets less than this have been evacuated)

    extra *mapextra // optional fields
}
```

在 extra中不仅有overflow,用于扩容和nextoverflow(prealloc的地址)。

bucket(bmap)的结构如下

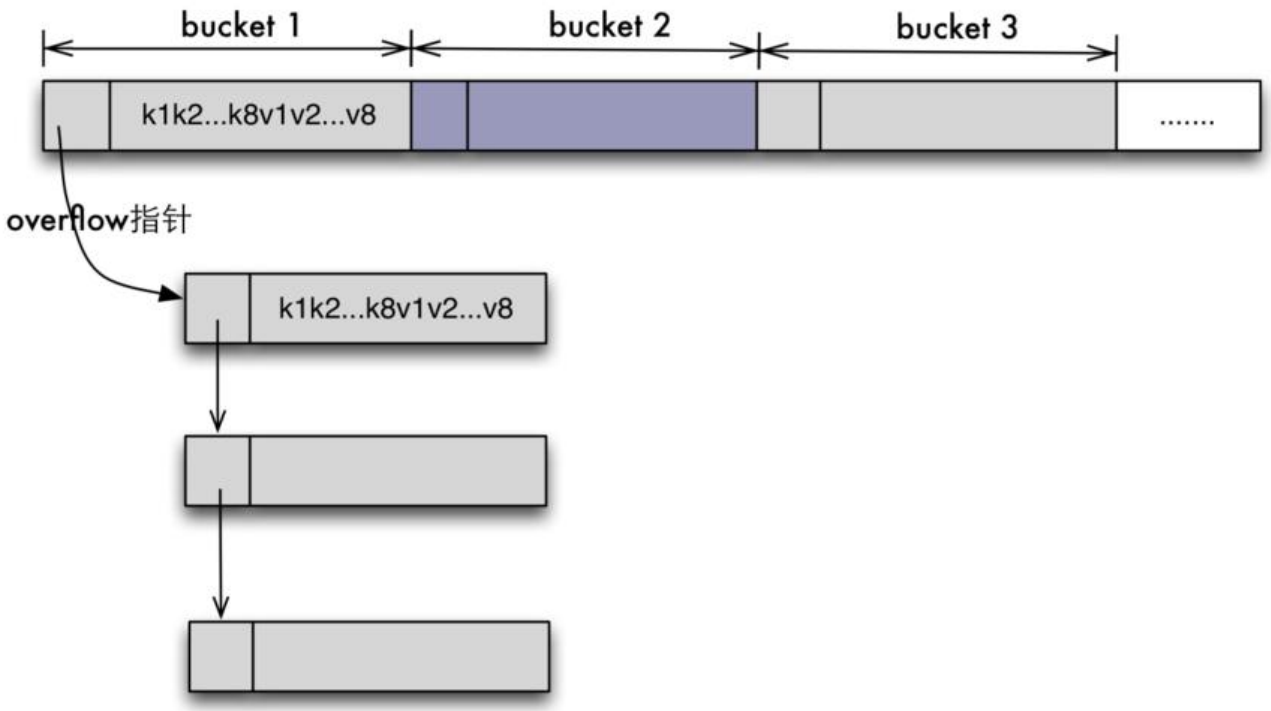
```
// A bucket for a Go map.
type bmap struct {
    // tophash generally contains the top byte of the hash value
    // for each key in this bucket. If tophash[0] < minTopHash,
    // tophash[0] is a bucket evacuation state instead.
    tophash [bucketCnt]uint8
    // Followed by bucketCnt keys and then bucketCnt values.
    // NOTE: packing all the keys together and then all the values together makes the
    // code a bit more complicated than alternating key/value/key/value/... but it allows
    // us to eliminate padding which would be needed for, e.g., map[int64]int8.
    // Followed by an overflow pointer.
}
```

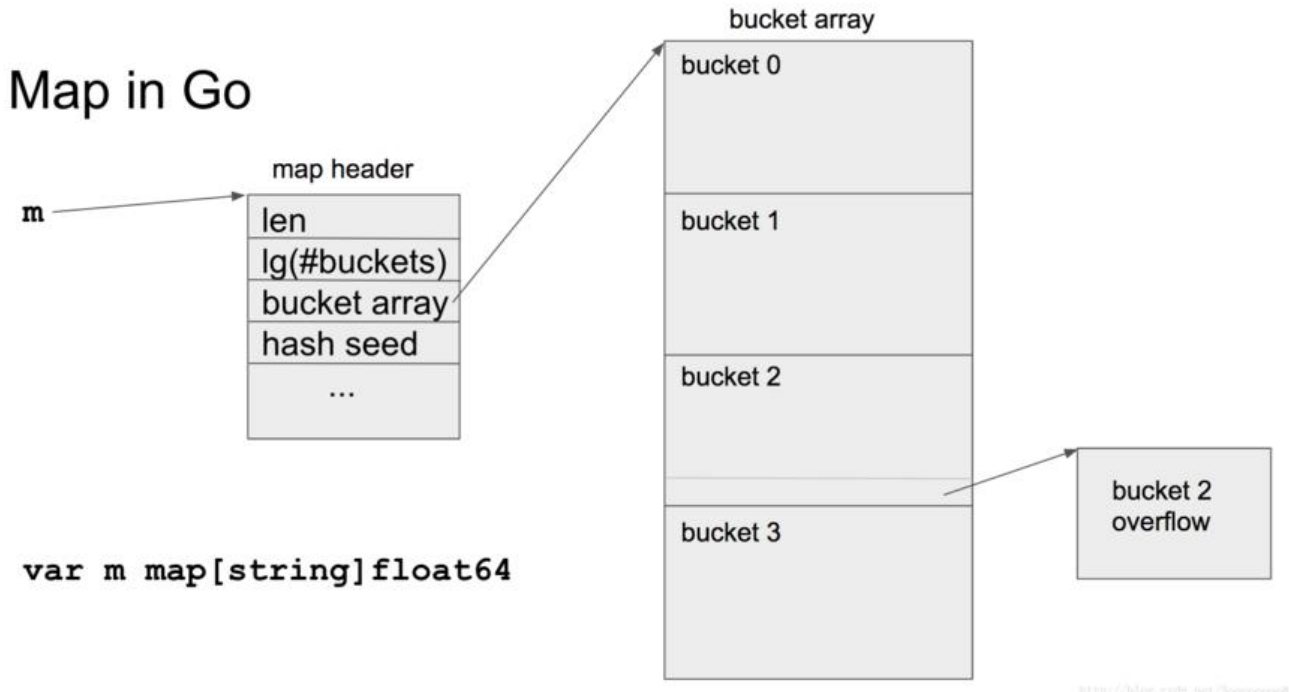
- tophash用于记录8个key哈希值的高八位，这样在寻找对应key的时候可以更快，不必每次都对key全值判断
- Note:hmap并非只有一个tophash,而是后面紧跟8组kv对和一个overflow的指针，这样才能使overflow成为一个链表结构。但是这两个结构体并不是显示定义的，而是直接通过指针运算进行访问的。

● kv的存储形式为"key0key1key2key3...key7val1val2...val7",这样做的好处是:在key和value的长度同的时候,节省padding空间。如上面的例子:在map[int64]int8中,4个相邻的int8

存储在同一个内存单元中。如果使用kv交错存储的话,每个int8都会被padding占用独自的内存单元(了提高寻址速度)

Hmap示意图





原文链接：golang~map 的学习与理解

map的访问

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    // do some race detect things
    // do some memory sanitizer thins

    if h == nil || h.count == 0 {
        return unsafe.Pointer(&zeroVal[0])
    }
    if h.flags&hashWriting != 0 {  // 检测是否并发写，map不是gorountine安全的
        throw("concurrent map read and map write")
    }
    alg := t.key.alg  // 哈希算法 alg -> algorithm
    hash := alg.hash(key, uintptr(h.hash0))
    m := bucketMask(h.B)
    b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
        // 如果老的bucket没有被移动完，那么去老的bucket中寻找 （增长部分下一节介绍）
    if c := h.oldbuckets; c != nil {
        if !h.sameSizeGrow() {
            // There used to be half as many buckets; mask down one more power of two.
            m >>= 1
        }
        oldb := (*bmap)(add(c, (hash&m)*uintptr(t.bucketsize)))
        if !evacuated(oldb) {
            b = oldb
        }
    }
        // 寻找过程：不断比对tophash和key
    top := tophash(hash)
    for ; b != nil; b = b.overflow(t) {
        for i := uintptr(0); i < bucketCnt; i++ {
            if b.tophash[i] != top {
                continue
            }
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            if t.indirectkey {
                k = *((*unsafe.Pointer)(k))
            }
            if alg.equal(key, k) {
                v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valu
size))
                if t.indirectvalue {
                    v = *((*unsafe.Pointer)(v))
                }
                return v
            }
        }
    }
        return unsafe.Pointer(&zeroVal[0]
```

map存值

首先用key的hash值低8位找到bucket，然后在bucket内部比对tophash和高8位与其对应的key值与

参key是否相等，若找到则更新这个值。若key不存在，则key优先存入在查找的过程中遇到的空的top ash数组位置。若当前的bucket已满则需要另外分配空间给这个key，新分配的bucket将挂在overflo 链表后。

map的增长

随着元素的增加，在一个bucket链中寻找特定的key会变得效率低下，所以在插入的元素个数/bucke 个数达到某个阈值(6.5)时候,map会进行扩容，hashGrow函数。首先创建bucket数组，长度为原长度的两倍 newb ckets, nextOverflow := makeBucketArray(t, h.B+bigger)，然后替换原有的bucket,原有的bucket 移动到oldbucket指针下。

扩容完成后，每个hash对应两个bucket(一个新的一个旧的)。oldbucket不会立即被转移到新的bucke 下，而是当访问到该bucket时，会调用growWork方法进行迁移，growWork方法会将oldbucket下 元素rehash到新的bucket中。随着访问的进行，所有oldbucket会被逐渐移动到bucket中。

但是这里有个问题：如果需要进行扩容的时候，上一次扩容后的迁移还没结束，怎么办？在代码中我 可以看到很多”again”标记，会不断进行迁移，知道迁移完成后才会进行下一次扩容。

这个迁移并没有在扩容之后一次性完成，而是逐步完成的，每一次insert或remove时迁移1到2个pair 即增量扩容。

增量扩容的原因 主要是缩短map容器的响应时间。若hashmap很大扩容时很容易导致系统停顿无响 。增量扩容本质上就是将总的扩容时间分摊到了每一次hash操作上。

由于这个工作是逐渐完成的，导致数据一部分在old table中一部分在new table中。old的bucket不 删除，只是加上一个已删除的标记。只有当所有的bucket都从old table里迁移后才会将其释放掉

使用时候的问题

Q：删除掉map中的元素是否会释放内存?

A：不会，删除操作仅仅将对应的tophash[i]设置为empty，并非释放内存。若要释放内存只能等待 针无引用后被系统gc

Q：如何并发地使用map?

A：map不是goroutine安全的，所以在有多个gorountine对map进行写操作是会panic。多gorount ne读写map是应加锁（RWMutex），或使用sync.Map。

Q：map的iterator是否安全?

A：map的delete并非真的delete，所以对迭代器是没有影响的，是安全的。

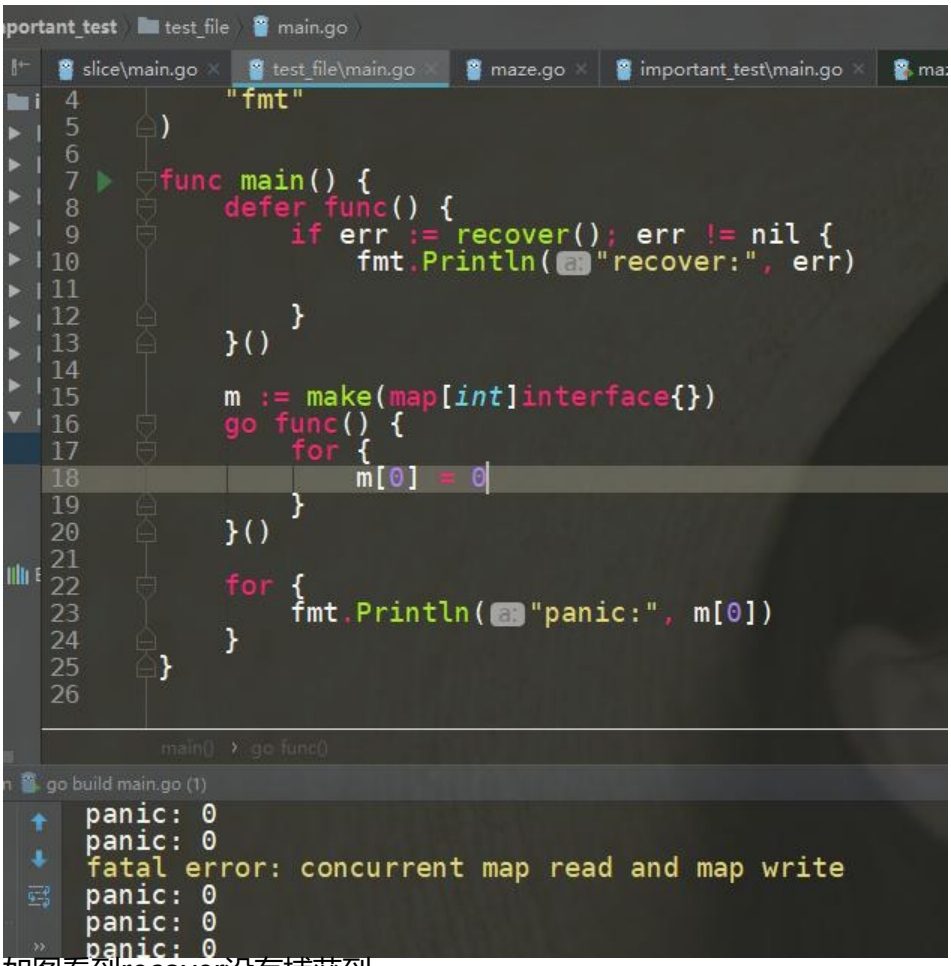
map 值得注意的地方

1、delete map 存在内存泄漏的问题

解决办法：定期更换成新的map，释放旧的map对象

2、并发操作map panic 不能被recover捕获

解决办法：加sync.RWMutex锁，或者使用ConcurrentMap组件

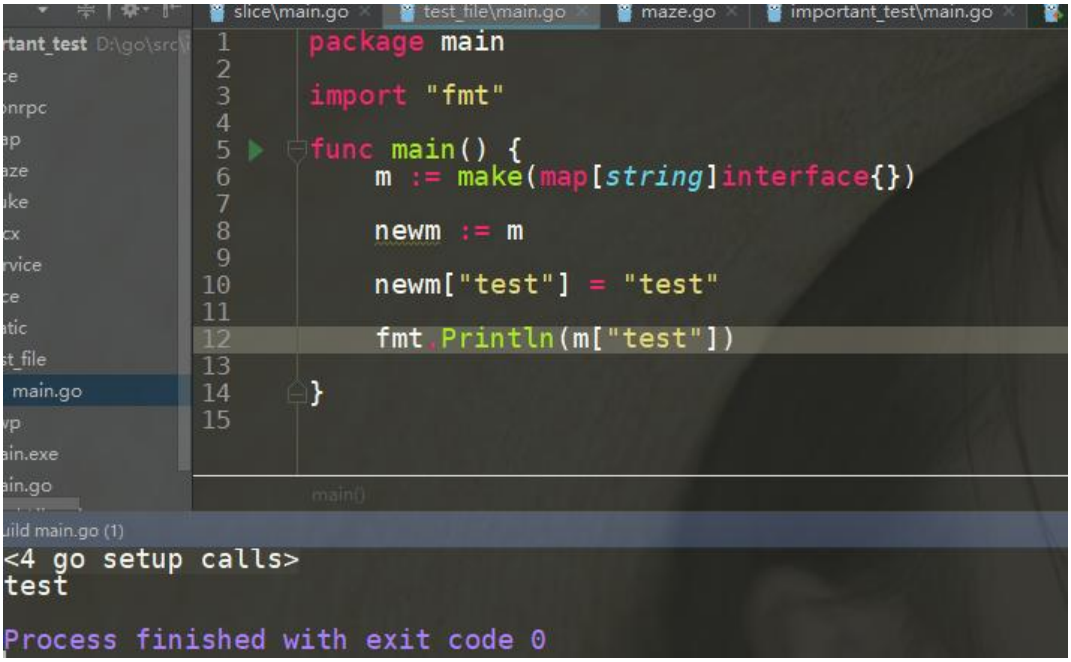


如图看到recover没有捕获到

3、每次map遍历不能得到相同排序的集合

4、map的key不支持slice,map,func,

5、map是引用类型



yiz96的博客

nino的博客

map传值问题