一个系列彻底搞懂map(三):go语言map剖析

go语言的map是使用极为频繁的数据结构，如果单单是以理解哈希表原理为目标的话，阅读本系列第一篇文章即可。本篇文章更倾向于通过分析go源码的方式，来深入学习一些哈希表优化技巧。

本文阅读的前提是你已经对哈希表的拉链法有一定了解，或者已经阅读该系列的第一篇文章，本文不会对这些基础原理再进行过多阐述。

go语言指针

go语言有三种类型指针，平时开发过程只会接触到普通指针这一种，但在go语言偏底层的源码中涉及大量三类指针转化和运算，这里先把盲点扫清。

在c语言中指针是灵魂所在，尽管指针让其特别灵活高效，但通过指针运算访问内存的操作，存在大量的安全隐患，如越界访问内存、破坏类型系统中类型原子性等。一些错误用法如下：

```
// 案例1
int arr[2];
*(arr+2) = 1;            // 越界访问内存地址

// 案例2
int a = 4;
int* ap = &a;    // 取a变量这4字节的起始地址
*(short*)ap = 2; // 更改了a变量4B中的前2B，不直接操作a就对a的值进行了更改，破坏了int变量的原子性

// 案例2的代码在某些场景可能出现，但在编码方式分别为小端以及大端的机器上存在移植性问题
```

c语言存在这些安全隐患的原因是，其支持对于指针的运算以及指针类型的转化。因此在go语言中，我们平时使用最多的带类型普通指针，都取消了指针的运算以及转换操作，从而保证了类型的安全性，如下：

```
var a int32 = 10
var ap *int32 = &a      //带类型的普通指针

ap++                            //非法，不允许指针运算
p := (*int16)(ap)       //非法，*int32不能直接转化为*int16
```

这样就保证了指针永远指向安全即分配了有效内存的地址，也保证了类型的独立和原子性。

除了普通指针外，go语言还保留了另外两种类型的指针，通过它们可以绕过类型系统，达到c语言自由操控内存的灵活程度。其他两种指针如下：

- `unsafe.Pointer`。
- `uintptr`。

要理解这两者，一定要建立个概念：指针本质上就是个数，只不过这个数保存的是内存地址而已。32位机器寻址空间为32位，64位机器寻址空间为64位，一个指针占用大小就等于机器的位数。

`uintptr`很简单，其就是单纯一个保存内存地址的数，在32位机器下等价于`uint32`，64位机器下等价于`uint64`。既然是一个数，自然就支持运算，从而就能表示任意一个内存位置。但问题是一个数据仅仅通过内存地址是无法定位的，你还需要知道它多大，说白点我们无法单纯依靠uintptr这个指针对数据进行操作。而普通的带类型的指针，除了告诉地址外，这个类型就告诉了数据的大小，从而帮助编译器理解如何取操作指向的内存。如`*int32`、`*int64`指针就分别告诉编译器操作指向地址的4B、8B数据。

解释清楚了go语言中的普通指针以及`uintptr`指针，那么这个相较于c语言多出的`unsafe.Pointer`是什么呢?

`unsafe.Pointer`指泛型指针，和`uinptr`一样只保留了内存地址而不关心类型。但它和`uintptr`的区别是，前者指向的对象会在gc中引用计数，从而不被gc当做垃圾回收掉，而后者相反，其只单纯表示内存地址这个数，也就是说有个数据地址就算被uintptr保存，也会被无情回收掉。

go语言三种指针总结：

- 普通指针。不支持指针运算，保存地址以及类型信息，指向数据不会被gc回收。
- `unsafe.Pointer`。不支持指针运算，保存地址但不保存类型信息，指向数据不会被gc回收。

- uintptr。支持地址运算，保存地址但不保存类型信息，指向数据会被gc回收。

在实际的使用中，uintptr不能直接与普通指针互转，都必须先转化为unsafe.Pointer这个桥梁后，才能进行下一步转化。

这里给出一个简单案例：

```
type Foo struct{
    a int32
    b int32
}

foo := &Foo{}
bp := uintptr(unsafe.Pointer(foo)) + 4  //将foo的地址加4定位到foo.b上
*(*int32)(unsafe.Pointer(bp)) = 1               //转化为*int32普通指针，修改值
fmt.Println(foo.b)      // foo.b = 1
```

通过地址偏移操作从而改变了foo的b成员，更多内容在后续碰到时再讲解。

数据结构

go语言的map源码位于$GOROOT/src/runtime/map.go中，哈希map实现采用拉链法。

本篇源代码使用的go版本为1.17.2。

hmap

map的数据结构定义如下：

```
// hashmap数据结构
type hmap struct {
    count     int               // len(map)，即map中元素数量
        flags     uint8         // map的状态
        B         uint8         // 2^B为桶buckets的长度
        noverflow uint16        // 溢出桶的近似数量
        hash0     uint32        // hash seed

        buckets    unsafe.Pointer       // 桶
        oldbuckets unsafe.Pointer       // 旧桶，只有在扩容时不为nil
        nevacuate  uintptr              // 标号小于此值的旧桶已经迁移

        extra *mapextra
}

type mapextra struct {
        // key和value inline时保存溢出桶
        overflow    *[]*bmap
        oldoverflow *[]*bmap

        // nextOverflow指向下一个能够使用的空溢出桶
        nextOverflow *bmap
}
```

对于hmap，重点讲下部分字段：

- flags。表示map现在处于的状态，可取iterator、oldIterator、hashWriting、sameSizeGrow。
- B。2^B代表桶的长度，桶的长度取为2的整数次方的目的是，能够将v%(2^k)的取模计算转化为位运算v&((1<<k)-1)，位运算的效率比取模高不少。
- noverflow。表示溢出桶的数量，如果溢出桶数量过多，会发生等量扩容，这个稍后讲到。
- hash0。go语言源代码是开源的，如果同一个key的hash结果总是一成不变，容易收到攻击。所以每次map初始化时会随机生成一个hash seed即hash0，它会和key一起作为哈希函数的输入。

- buckets表示桶。桶是一片连续的内存，buckets指向首地址。
- oldbuckets表示旧桶。go语言的map，在旧桶迁移到新桶的过程中，并不是一蹴而就的，而是在操作中逐次进行，因此有必要保存旧桶。
- extra是mapextra类型，其中`overflow`和`oldoverflow`只有在键值的类型都为inline时才启用，我们在后面的bmap部分讲解。这里讲讲`nextoverflow`，一个桶只能存8个键值对，如果超过8个就需要生成额外的溢出桶进行存取(溢出桶通过拉链连接)。在go实现中，为了提升效率，有些情况可能在map初始化时就生成溢出桶池，我们需要时直接拿来用即可，而不是动态生成，`nextOverflow`就指向溢出桶池中下一个能使用的空溢出桶。

bmap

bmap就代表逻辑上的一个桶。

在本系列的第一篇文章实现中，我们是让每个桶保存一个键值对，n个同义词就会建立n个桶，以链表方式链接同义词桶，每个键值对就需要额外一个指向下一个bucket的next指针。

go语言做法是让每个bmap桶保留8个键值对，这样每8个键值才需要一个next指针，减少内存消耗的同时，也减少了bucket对象的个数，从而降低了gc回收负担。

那么为什么不保存16个、32个呢？因为go的负载因子定为6.5，也就是说装的最满的情况下，每个桶平均最多装入6.5个键值就会触发扩容操作，让每个桶能最多存放16个甚至更多键值，我们也不可能全部利用到，只会徒徒浪费内存。

go语言中桶bmap的数据结构：

```
type bmap struct {
    // 每个桶存取8个键值，tophash就保存这8个键哈希后的高8位
        tophash [8]uint8
    // 其他字段由编译器给出
}
```

每个键值对存入前，我们都会对key进行哈希操作，得到的高8位哈希值，用`tophash`这个数组储存。

你会发现bmap缺乏很多属性，这是因为go不支持泛型，因此单从语法层面，不可能做到平时使用中创建各种类型map的效果。这需要编译器的支持，编译器会根据我们使用map时给键值设定的类型，从而对bmap进行补全，帮助我们生成相关的代码。

编译器对于bmap的补全有两种情况，第一种情况是key和value类型的size都小于128B且不是指针(称为inline)的情况下，否则为第二种情况。

对于第二种情况，`bmap`会补全成类似以下类型：

```
// keytype和valuetype由编译器推导给出
type bmap struct {
        tophash  [8]uint8               //8个键对应的hash高8位
    keys     [8]keytype         //8个键
    values   [8]valuetype       //8个桶
    overflow *bmap                      //overflow就是链表节点的next指针，指向下一个同义词桶
}
```

而对于第一种情况，key和value都不包含指针且都小于128B时，我们可以将它们直接内联在bmap的结构中，从而减少指针引用对象的次数。`bmap`会补全成类似以下样子：

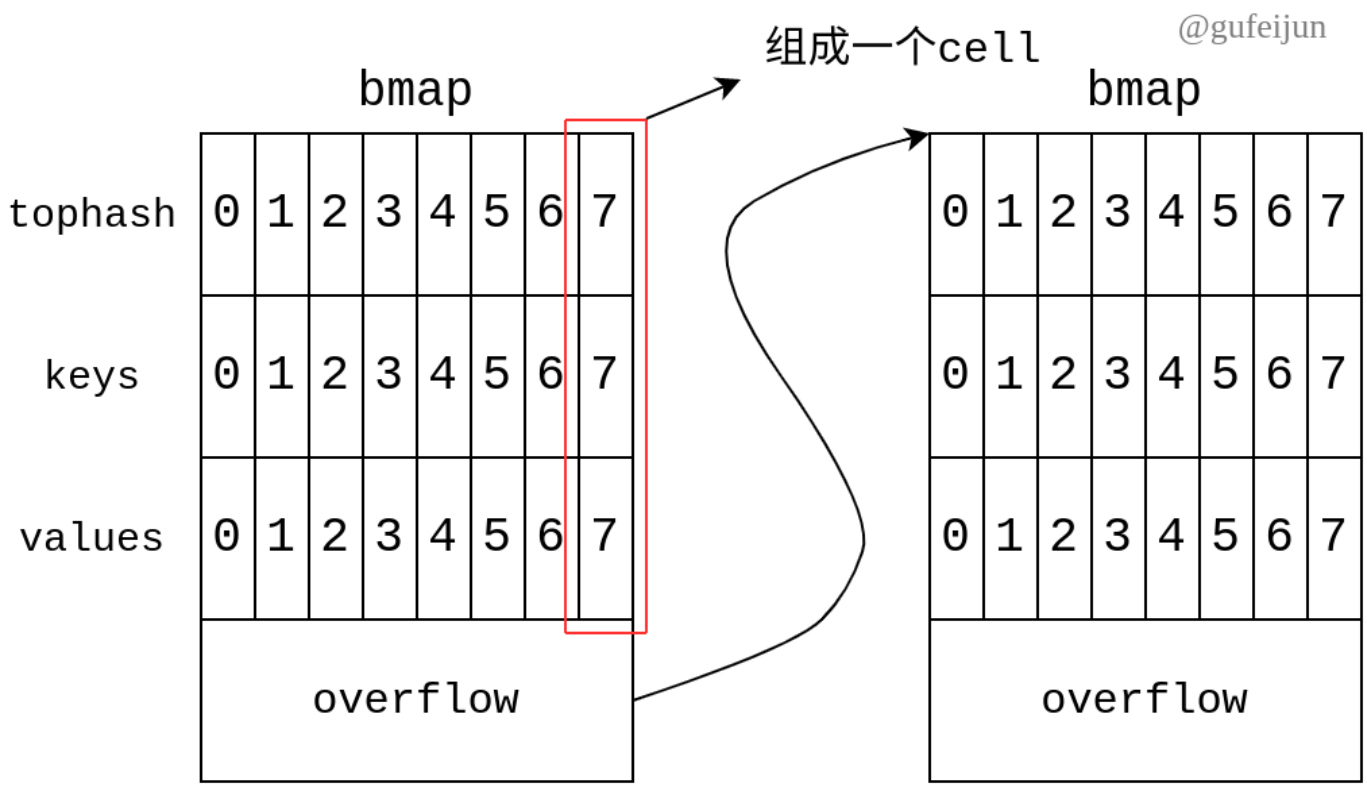
```
type bmap struct {
        tophash  [8]uint8
    keys     [8]keytype
    values   [8]valuetype
    overflow uintptr            //只保留下一个同义词桶的地址，而不引用计数。
}
```

`overflow`会被设计成uintptr类型，前文说到如果为`unsafe.Pointer`或者`*bmap`类型，这涉及对象的引用，因此gc在释放桶时需要遍历同义词链表的所有节点。而uintptr只存储了地址，gc释放数据时不需要访问每个桶以及溢出桶，从而缓解gc压力。

但存在一个问题，uintptr既然不对对象引用计数，那么gc误回收其指向的溢出桶数据怎么办？

go的做法是将这些溢出桶对象统一的保存在上文讲到的mapextra对象的overflow成员中，让overflow这个切片保证引用计数，同时以后gc回收这些溢出桶时，只需要扫描overflow这个切片即可，效率更高。

每个桶有8个位置去容纳键值对，为了后续讨论方便，我们这里称每一个位置为一个cell或者一个entry。bmap的结构如图所示：



我估计很多人会有这个疑问：不同键可能具有相同的高8位哈希值，所以tophash不能作为区分不同键的依据，最后还是要比较key，那么为什么要保存tophash这个属性，占用内存同时又没起到任何作用？

以下内容仅为个人推测go作者设计时的考量：

- 首先，虽然不能因为tophash相同就确定是同一个key，但如果tophash不同就一定不是同一个key。也就是说一旦发现tophash不同，也就没必要比较key了，这样就能减少比较的时间，如string类型key的比较就很耗时。
- 其次，目前还有个问题，每个cell可能是空的也可能被占用，肯定得进行区分。最容易想到的是引入一个8位的bitmap，作为bmap的成员即可，虽然仅占用1B，但在64位机器下因为内存对齐会扩充为8B，所以一不做二不休干脆，就引入8B的一个属性成员即这里的tophash，比较奢侈地用1B去表示一个cell的状态。但转念一想，比较浪费，于是它们让这1B所表示的数小于minTopHash这个宏时，来表示cell的特殊状态，其他情况下来存取高8位哈希值，这样就最大利用到内存。

所以tophash值不仅仅用来存取键的高8位hash值，有时还用来表示cell的状态，读者需知。minTopHash会在后面常量宏小节讲到。

常量宏

以下为重要的常量：

```
// 一个桶最多包含的键值对，为8
// 这个常量名其实很有误导性，叫bucketEleCnt更好
bucketCntBits = 3
bucketCnt     = 1 << bucketCntBits

//go语言中负载因子>6.5时进行扩容，不用浮点，用整数计算效率更高
loadFactorNum = 13
loadFactorDen = 2

//key以及value的最大字节，如果小于128则让key-value在bmap结构体中内联
//否则我们需要将key、value放在堆区，用指针引用。见前文讲解
maxKeySize  = 128
maxElemSize = 128
```

```

// dataOffset就是bmap.keys相较于bmap的偏移
// 见bmap结构体，在标准库中bmap只有tophash成员，跳过tophash就是keys
dataOffset = unsafe.Offsetof(struct {
    b bmap
    v int64
}{}.v)


//用于表示每个entry的状态
emptyRest      = 0 // 此entry为空，后面的同义词entry也是空
emptyOne       = 1 // 此entry为空，删除时会将entry置为这个状态

// 下面evacuatedXXX在迁移时使用，这里可暂时忽略
evacuatedX     = 2 // 此entry已经迁移到新桶的前一半
evacuatedY     = 3 // 此entry已经迁移到新桶的后一半
evacuatedEmpty = 4 // 此entry在迁移时就是空的

minTopHash     = 5 // 普通entry能使用的最小的高位hash值，0~4不能用，因为用于表示特殊值

// 用于表示map的状态标志，hmap.flags属性
iterator     = 1 // 可能有迭代器在使用桶(for range状态)
oldIterator  = 2 // 可能有迭代器在使用旧桶(for range状态)
hashWriting  = 4 // 有go程在写map
sameSizeGrow = 8 // 这个map正在等量扩容
```

这里讲讲几个极为关键的常量：

`dataOffset`：其实前文说编译器会扩充补全bmap的成员相关代码并不准确，bmap结构体始终都只有tophash这个成员，编译器只是会在bmap占用内存后继续分配keys、values以及overflow的内存。前面讲到的扩充后bmap结构体是一种便于理解逻辑上的结构，这点需知。所以上面的dataOffset中，跳过bmap取偏移只是跳过了tophash成员，从而定位到了keys。

`emptyRest`和`emptyOne`的区别：一个桶有8个cell，桶后还可能跟上溢出桶。如果一个cell为`emptyRest`，则代表此cell后面的cell也都是空的。如果一个cell为`emptyOne`，则此cell后面的cell中有非空的。引入这两种状态是为了提高某些操作效率，如查找时发现当前cell已经为`emptyRest`，则没必要查看后续cell，直接结束流程即可。emptyRest为0，因此一个cell在创建后默认处于这个状态。

`minTopHash`很好理解，小于minTopHash的值被用来标记桶的状态，因此hash值的高8位就只能大于等于`minTopHash`。如果一个key的hash值高8位正好小于`minTopHash`时，会将其加上`minTopHash`再保存。

操作

初始化

在runtime中有多种初始化函数，某些是针对特定类型进行专门优化的，由编译器决定将make展开为哪一个函数，不过都大同小异，我们这里只看最通用的`makemap`函数：

```
//如果h或者h.buckets可以分配在栈区，则h不为nil，直接对h初始化即可
//其他情况下，我们在堆区生成一个新的hamp。
func makemap(t *maptype, hint int, h *hmap) *hmap {
    // 计算存取这些键值需要的内存，相乘可能溢出，是否溢出用overflow保存
        mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
    //如果溢出或者需要的内存超过了maxAlloc，我们让hint为0
        if overflow || mem > maxAlloc {
                hint = 0
        }

        // 如果map不是在栈区，我们在堆区申请
        if h == nil {
                h = new(hmap)
        }
        h.hash0 = fastrand()

    //不断增加B，直至在装满hint个键值对的情况下，也不会触发map扩容
```

```
        B := uint8(0)
        for overLoadFactor(hint, B) {
                B++
        }
        h.B = B

    //在此之上的逻辑都是决定该分配多少桶

        //当B=0时，不分配桶
        if h.B != 0 {
                var nextOverflow *bmap
                h.buckets, nextOverflow = makeBucketArray(t, h.B, nil)          //makeBucketArray见下
        //如果提前分配了溢出桶的话，用h.extra.nextOVerflow指向第一个空溢出桶
                if nextOverflow != nil {
                        h.extra = new(mapextra)
                        h.extra.nextOverflow = nextOverflow
                }
        }
        return h
}
```

makemap需要三个参数：

- `t *maptype`。这个是由编译器生成并传入，包含了map的类型信息，如key进行比较或者哈希的函数等，我们无需关心。
- `hint int`。在我们make一个map时，还可以传入一个参数，这个参数就是hint，它代表我们想要map存取的键值个数。
- `h *hmap`。有些情况下，如果用户建立的map生命周期很短且不需要存取很多键值时，编译器会将make展开为如下方式：

  ```
  var m map[keytype]valuetype
  makemap(maptype,n,&m)
  ```

  即让map对象分配在栈区，然后在栈区分配好的内存基础上操作，这样能降低gc的负担。

  否则当传入的h为nil时，我们在堆区给map分配内存。

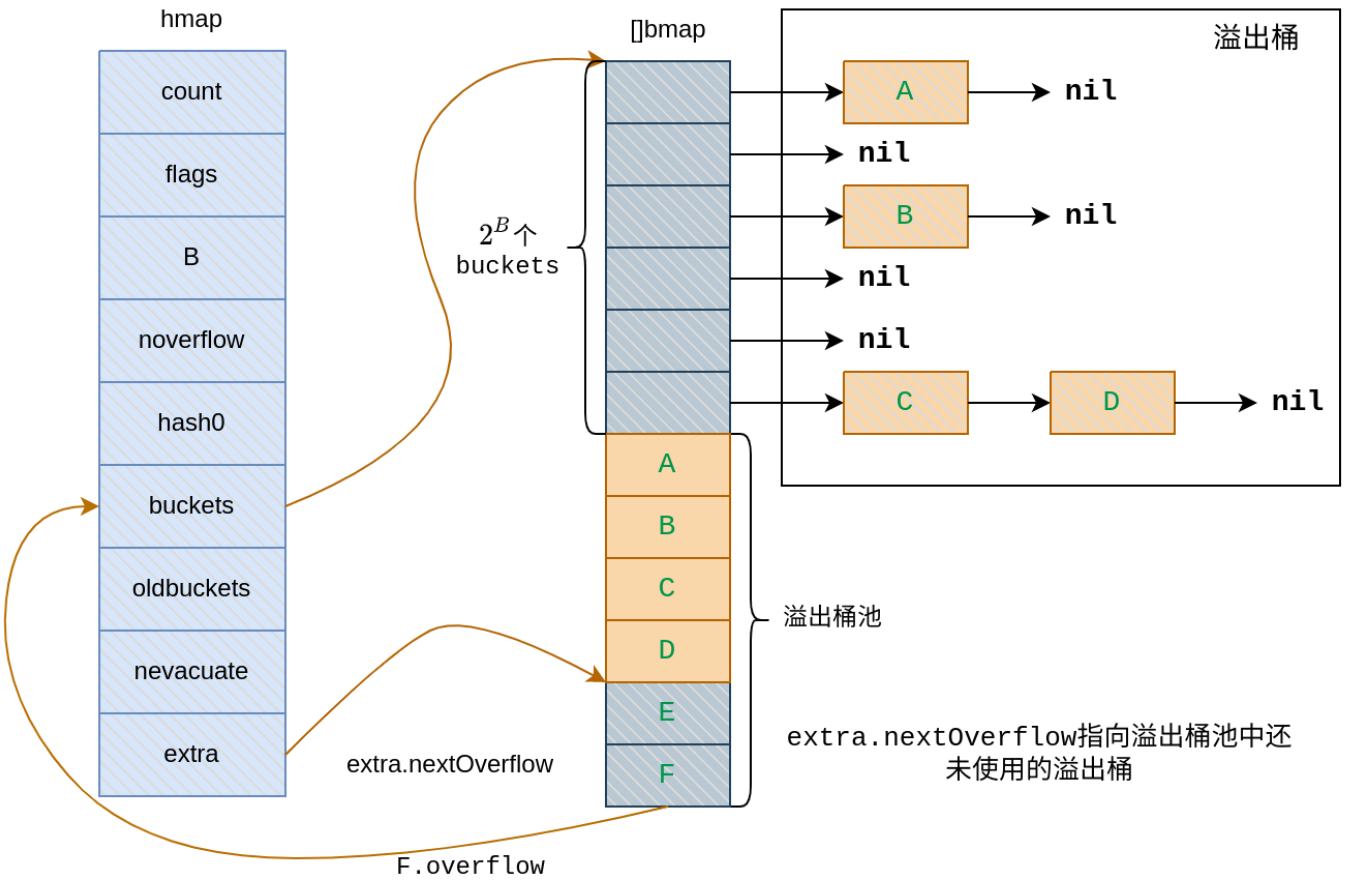
makemap的整体逻辑极为简单，就分为两个步骤：

1. 先根据用户传入的hint，来确定应该给map分配桶的个数。桶的个数通过以下规则决定：
   - 如果指定的hint为0，则桶的个数为0。
   - 如果指定的hint很大，hint个bucket占用的内存超过了我们的限定值`maxAlloc`，则桶的个数也为0。防止一下子分配过多内存，且短时间内用不到。
   - 否则，我们让桶的个数达到以下条件：在连续插入hint个键值的情况下，也不会触发扩容。即桶的个数要大于hint/6.5，并为2的整数次方，上面的`overLoadFactor`循环就起这个作用，不再详细讲解。
2. 接着分配桶和溢出桶。如果上一步中决定桶的个数为0时，直接不分配桶和溢出桶，待以后插入时再说。否则即刻分配桶，且可能在此过程生成溢出桶池，具体函数为`makeBucketArray`，关于桶生成规则如下：
   - 如果需要普通桶的个数`n = 2^b`满足`n >= 16`时，我们会连续分配`n + 1<<(b-4)`个桶的内存，前n个桶作为普通桶，后`1<<(b-4)`个桶作为溢出桶池。
   - 否则n比较小时，如果需要n个桶，就分配n个桶的内存，即不生成溢出桶池。(n比较小溢出的可能不大)

一定得注意的是，溢出桶其实跟普通桶在内存上是一片紧临连续的内存，只是划分前一部分作为普通桶，后一部分作为溢出桶池而已，如图所示：

Map of Go

@gufeijun



makeBucketArray就是实现了上面的桶分配逻辑，如下：

```
// 返回桶指针，同时如果还有溢出桶的话，nextOverflow指向第一个溢出桶
func makeBucketArray(t *maptype, b uint8, dirtyalloc unsafe.Pointer) (buckets unsafe.Pointer, nextOverflow *bmap) {
        base := bucketShift(b)          // 1<<b, base是普通桶的个数
        nbuckets := base                        // nbuckets是普通桶和溢出桶的总个数
    // 需要普通桶个数大于16时，分配溢出桶
        if b >= 4 {
        //分配时多分配1<<(b-4)的桶作为溢出桶
                nbuckets += bucketShift(b - 4)
                sz := t.bucket.size * nbuckets
                up := roundupsize(sz)
                if up != sz {
                        nbuckets = up / t.bucket.size
                }
        }

        if dirtyalloc == nil {
                buckets = newarray(t.bucket, int(nbuckets))     //分配一片连续内存
        } else {
                //pass
        }

    //如果分配了溢出桶池
        if base != nbuckets {
                //下面两个为指针运算，nextOverflow指向第一个溢出桶，last指向最后一个溢出桶
                nextOverflow = (*bmap)(add(buckets, base*uintptr(t.bucketsize)))
                last := (*bmap)(add(buckets, (nbuckets-1)*uintptr(t.bucketsize)))
        //让最后一个桶的overflow指针不为nil，作为哨兵
```

```
                last.setoverflow(t, (*bmap)(buckets))
        }
        return buckets, nextOverflow
}
```

搭配前文的分配逻辑以及注释食用，很容易理解。

这里详细讲解下上面的两个指针运算过程，add地址运算函数如下：

```
// 返回地址p+x
func add(p unsafe.Pointer, x uintptr) unsafe.Pointer {
        return unsafe.Pointer(uintptr(p) + x)
}
```

在nextOverflow的计算公式中，buckets是桶数组的首地址，加上base个普通桶大小的内存偏移后，正好定位到第一个溢出桶。last的计算同理，buckets加上所有桶数量-1个桶大小的内存偏移后，正好定位到最后一个溢出桶。

setoverflow也很简单，就是更改bmap结构的overflow指针成员而已，如下：

```
func (b *bmap) setoverflow(t *maptype, ovf *bmap) {
        *(**bmap)(add(unsafe.Pointer(b), uintptr(t.bucketsize)-sys.PtrSize)) = ovf
}
```

理解挺容易，t.bucketsize是编译器给出的，即一个bmap在逻辑上有多大。因为overflow是bmap的最后一个成员，且为*bmap指针类型，于是bucket大小减去指针大小就正好是overflow成员的偏移，加上bucket首地址就正好定位到overflow上。

值得注意的是，这里让最后一个溢出桶last的overflow指针不为nil，以此作为哨兵，如上图中的桶F。那么为什么需要这个哨兵？

前面讲到会用hmap.extra.nextOverflow指向溢出桶池中下一个未使用的溢出桶，所以后续需要溢出桶时，只需要将这个指针指向的桶取走，并将指针再指向当前指向地址 + bucketsize即下一个溢出桶即可。

哨兵就标记了溢出桶池的最后一个空闲桶。只要hmap.extra.nextOverflow指向了哨兵，拿走这最后一个溢出桶后，溢出桶池将成为空池，之后还需要溢出桶时，需要动态申请bmap内存作为溢出桶了。

好了，关于map初始化的逻辑就讲解完毕，捋清楚逻辑后源码还是很清晰的。

查询

所有函数都类似于mapaccessXXX形式，重点就是如下两个：

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer)
func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool)
```

区别就是一个带bool返回值，一个不带，编译器会根据用户代码决定选择哪个函数，如下：

```
v := m[key]             //mapaccess1
v, ok := m[key] //mapaccess2
```

两者大同小异，我们只看mapaccess2：

```
func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool) {
        if h == nil || h.count == 0 {
        // 返回一个全为0的值区域
                return unsafe.Pointer(&zeroVal[0]), false
        }
        if h.flags&hashWriting != 0 {
                throw("concurrent map read and map write")
        }
    // 计算key的hash值
        hash := t.hasher(key, uintptr(h.hash0))
    // m = 1<<h.B - 1，如h.B=3，则m=00000111b。hash&m即可取出hash的低h.B位。
        m := bucketMask(h.B)
```

```
    // hash&m = hash%(2^h.B)，即得到存取该key的桶序号
        b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
    // 如果处于扩容中，则优先查看旧桶，旧桶已经迁移了才看新桶
    if c := h.oldbuckets; c != nil {
                if !h.sameSizeGrow() {  //非等量扩容则oldbuckets是当前buckets的1/2
                        m >>= 1
                }
                oldb := (*bmap)(add(c, (hash&m)*uintptr(t.bucketsize)))
                if !evacuated(oldb) {   //判断旧桶是否已经迁移
                        b = oldb
                }
        }
        top := tophash(hash)    //取出hash值的高8位

    //找到目标桶b后，下面就是遍历桶中的每一个同义词cell
bucketloop:
   // 外循环是遍历所有同义词桶，内循环是遍历每个桶内的8个cell
   // overflow方法获取拉链法的下一个桶
        for ; b != nil; b = b.overflow(t) {
                for i := uintptr(0); i < bucketCnt; i++ {
            // 先比较tophash，tophash不同则key绝对不同
                        if b.tophash[i] != top {
                // 如果后面全是空cell了，肯定找不到对应key，直接返回
                                if b.tophash[i] == emptyRest {
                                        break bucketloop
                                }
                                continue
                        }
                        // 获取序号为i的key地址
                        k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
                        if t.indirectkey() {
                                k = *((*unsafe.Pointer)(k))
                        }
            //tophash相同但key可能不同，所以还是需要比较key
                        if t.key.equal(key, k) {
                // 获取序号为i的value地址
                                e := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                                if t.indirectelem() {
                                        e = *((*unsafe.Pointer)(e))
                                }
                                return e, true
                        }
                }
        }
        return unsafe.Pointer(&zeroVal[0]), false
}
```

注释挺全面，耐心观看。重点注意下列几点：

- zeroVal是一个数据全为0的定长字节数组，如下：

  ```
  const maxZero = 1024
  var zeroVal [maxZero]byte
  ```

  当map没有存取目标key时，我们将这块区域地址返回，用户不论是想要int、float还是string，都正好是对应类型的默认值。

- 本系列第一章中，我们是利用key%len(buckets)来确定桶的位置，但如果len(buckets)为2的整数幂的话，可以将这个取模操作替换为位运算，注释里已经注明，需要多加理解。

- 本系列第一章中，我们的扩容实现是一次直接搬迁完所有键值对，但对于大map来说这样会导致巨大的延时。所以go语言的扩容采取的是逐次搬迁的操作，将搬迁时间平均打散在对map的多个操作中，在未完成所有搬迁前旧桶不能释放。因此我们的查找就需要注意旧桶和新桶的问题，如果对应旧桶还没搬迁则查看旧桶，否则直接查看新桶即可。扩容部分会在后面有更详细的讲解。

- k和e也是通过指针运算得到，这里不再赘述，读者自行对比bmap结构理解。

- 返回值是对应value的指针，但作为用户来说，我们拿到的就是value值，显然编译器自动给我们插入解引用的代码。可以如下测试，编写main.go：

```
var m map[int]int
v := m[1]
```

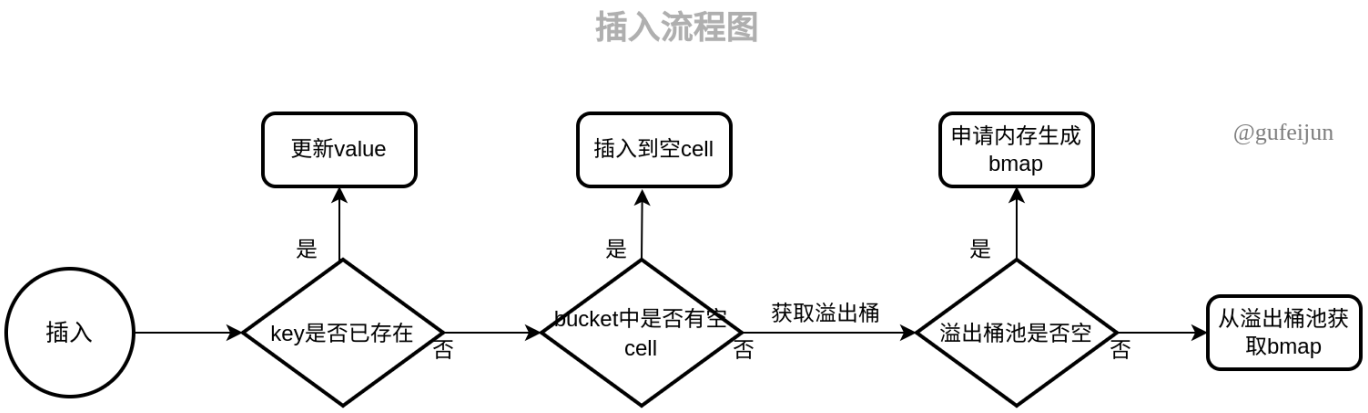
利用go tool compile -S main.go生成如下汇编(节选)：

```
0x0022 00034 (main.go:7)        CALL    runtime.mapaccess1_fast64(SB)
0x0027 00039 (main.go:7)        MOVQ    (AX), AX        #解引用
```

mapaccess返回值保存在AX寄存器中，下一行就访存对指针解引用，将值取出后再保存入AX。

插入

插入过程和查询很类似，很容易理解，过程如下：



见mapassign函数(删除了部分非重要代码)，看不懂可以先观看后续的讲解：

```
// 返回待更新的value地址
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
        hash := t.hasher(key, uintptr(h.hash0))

    //见前面makemap，当h.B=0时，在此只分配一个桶
        if h.buckets == nil {
                h.buckets = newobject(t.bucket) // newarray(t.bucket, 1)
        }

again:
        bucket := hash & bucketMask(h.B)        //获取目标桶序号
        // 如果还处于扩容中，我们要让这个key对应的旧桶立马迁移到新桶来，见下文讲解
    if h.growing() {
                growWork(t, h, bucket)                  //见扩容
        }
    // b就是目标桶
        b := (*bmap)(add(h.buckets, bucket*uintptr(t.bucketsize)))
    top := tophash(hash)                                //hash高8位

        var inserti *uint8                              //第一个空cell对应的tophash存取位置
        var insertk unsafe.Pointer              //第一个空cell对应的键的地址
        var elem unsafe.Pointer                 //第一个空cell对应的值的地址
bucketloop:
    // 外循环是遍历同义词桶，内循环是遍历每个桶的8个同义词cell
        for {
                for i := uintptr(0); i < bucketCnt; i++ {
                        if b.tophash[i] != top {
                // 找到了第一个空cell
```

```
                                if isEmpty(b.tophash[i]) && inserti == nil {
                                        inserti = &b.tophash[i]
                                        insertk = add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
                                        elem = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                                }
                        // 如果后续的所有cell都是空的了，没必要再查询
                                if b.tophash[i] == emptyRest {
                                        break bucketloop
                                }
                                continue
                        }
                        k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
                    // tophash相同后，再比较key进行确认
                                if !t.key.equal(key, k) {
                                        continue
                                }

                    //并将待更新的value地址返回
                                elem = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                                goto done
                        }
                //取下一个溢出桶
                        ovf := b.overflow(t)
                        if ovf == nil {
                                break
                        }
                        b = ovf
                }

                //下面就是map没有存key的情况

            //当不处于扩容时，超过负载因子或者太多溢出桶，即溢出桶个数大于等于1<< min(h.B,15)
                if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
                        hashGrow(t, h)                  // 分配新桶，但还不迁移旧桶，见扩容部分
                        goto again // Growing the table invalidates everything, so try again
                }

            // 没找到任何空位
                if inserti == nil {
                // 从溢出桶池拿或者分配溢出桶内存，见下文
                        newb := h.newoverflow(t, b)
                        inserti = &newb.tophash[0]
                        insertk = add(unsafe.Pointer(newb), dataOffset)
                        elem = add(insertk, bucketCnt*uintptr(t.keysize))
                }

                *inserti = top  // 设置tophash
                h.count++               // map元素数量+1

        done:
                return elem     //将存取value的地址返回
        }
```

注释可以说是非常全面，这里讲一下可能有疑问的几点：

- 在插入键值时，如果此时还正处于扩容状态，则务必要让key对应的老桶立马迁移到新桶中来，14行的growwork就起到了这个作用。这么做的原因是，前文讲到的查询操作中，是优先查找旧桶，如果旧桶迁移了才找新桶。如果我们让这个键值插入到新桶后，不迁移对应旧桶的话，就会优先查找这个key对应的旧桶，从而找不到键值。

- 读者应该注意到了inserti、insertk和elem变量，它们分别保存了第一个空cell的tophash、键以及值的位置，这么做的目的是减少循环的次数，提高效率。因为我们插入时，还要注意key已经存在的情况，这时就处于更新操作，所以容易想到的就是第一次循环查看所有cell是否保存了key，有则更新value，否则再进行第二次循环找到第一个空cell，插入键值即可。但如果我们在第一次循环时就预先保存空cell的位置的话，就不需要第二次循环操作。

- overflow方法和前文讲到的setOverflow类似，就是获取bmap的overflow的指针即下一个溢出桶。

- 插入时，若负载因子>6.5或者出现溢出桶过多，我们会触发扩容机制，前者情况是将桶的数目扩充两倍，后者是等量扩容，新桶和旧桶数目一样，详细讲解见扩容小节。
- 最后返回的是待插入位置的指针，并没有直接对该位置进行赋值value的操作，其实编译器会给我们插入相关的代码。可以如下测试，编写main.go：

```
var m map[int]int
m[1] = 1
```

利用go tool complile -S main.go生成如下汇编(节选)：

```
0x0022 00034 (main.go:5)        CALL    runtime.mapassign_fast64(SB)
0x0027 00039 (main.go:5)        MOVQ    $1, (AX)
```

mapassign返回值保存在AX寄存器中，它保存了value地址，下一行就将数据写入到这个内存地址中。

最后讲下newoverflow，这个函数的作用是获得一个溢出桶去存取容纳不下的键值对，溢出桶的获取分两种情况，前文也稍提到：

- 如果溢出桶池还存在空余的桶，则从池中拿取即可。
- 否则申请内存，再分配一个溢出桶。

溢出桶池是否还有空余桶就用到了我们前文讲到的哨兵，函数如下：

```
func (h *hmap) newoverflow(t *maptype, b *bmap) *bmap {
        var ovf *bmap
    //如果nextOverflow不为空，则溢出桶池没空
        if h.extra != nil && h.extra.nextOverflow != nil {
                ovf = h.extra.nextOverflow
                if ovf.overflow(t) == nil {
            //将nextOverflow指针指向下一个空溢出桶
                        h.extra.nextOverflow = (*bmap)(add(unsafe.Pointer(ovf), uintptr(t.bucketsize)))
                } else {        //最后一个溢出桶是哨兵，overflow属性不为空
                        ovf.setoverflow(t, nil)
                        h.extra.nextOverflow = nil
                }
        } else {
        //溢出桶池空时，生成一个桶
                ovf = (*bmap)(newobject(t.bucket))
        }
        //将记录的溢出桶数量+1
        h.incrnoverflow()

    // 如果bmap是inline时，这时bmap的overflow是uintptr类型，见前文讲解
        if t.bucket.ptrdata == 0 {
        //给extra以及extra.overflow分配内存
                h.createOverflow()
        //将新生成的桶放入overflow中保存，防止gc误回收
                *h.extra.overflow = append(*h.extra.overflow, ovf)
        }
        b.setoverflow(t, ovf)
        return ovf
}
```

仔细看了前文的讲解应该不会有问题，这里不再赘述。

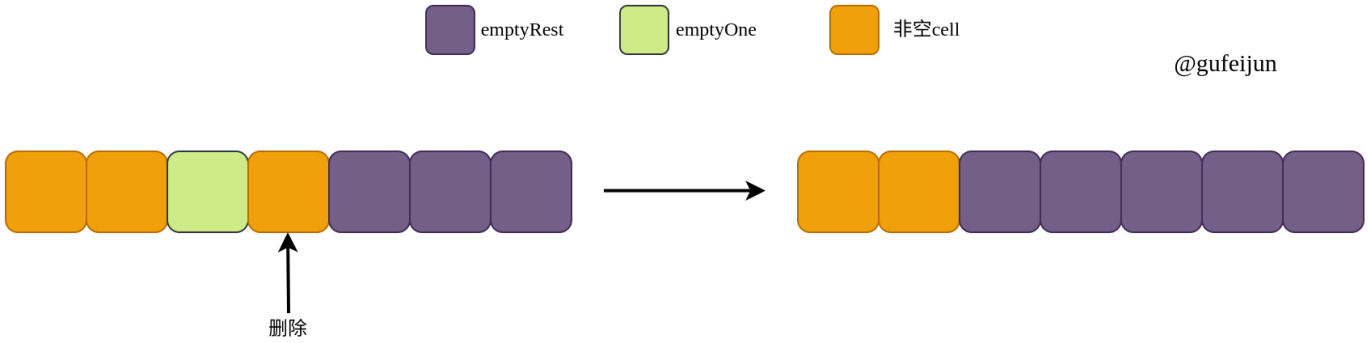
删除

删除是用到了mapdelete函数，也是极为简单，和查询一样，不过增加了一个更新cell的emptyReset状态的过程。

这里回顾下前文的知识，我们用mapdelete删除的cell会被标记为emptyOne状态，表示为这个cell为空cell。如果一个空cell的所有后续cell也都是空的，我们会将这个cell设置为emptyReset状态。这两种状态存在的目的，就是加速查询的过程，如碰到emptyReset状态的桶，也就没必要查询后续的桶了，因为必定都是空桶。

删除某个cell，并不是单单就更改该cell的状态即可，可能影响多个cell的状态。

如当前删除的cell的后一个cell为emptyReset状态时，显然要也要将当前cell由emptyOne改为emptyReset状态。在此之后，如果当前cell的前一个cell为emptyOne，还需要将前一个cell改为emptyReset状态，不断迭代直至所有空cell设置成正确状态，这部分就是mapdelete函数中的重点所在。如图：



mapdelete函数如下，为了便于理解，删除了部分代码以及修改了少量代码：

```
func mapdelete(t *maptype, h *hmap, key unsafe.Pointer) {
        hash := t.hasher(key, uintptr(h.hash0))
        bucket := hash & bucketMask(h.B)
        if h.growing() {
                growWork(t, h, bucket)
        }
    // b就是key对应的桶，循环过程中b还会变为后续的溢出桶
        b := (*bmap)(add(h.buckets, bucket*uintptr(t.bucketsize)))
    // bOrig是第一个同义词桶，后面回溯更改cell状态时会用到
        bOrig := b
        top := tophash(hash)
search:
    // 同样是大循环遍历同义词桶，小循环是遍历每个桶的8个cell
        for ; b != nil; b = b.overflow(t) {
                for i := uintptr(0); i < bucketCnt; i++ {
                        if b.tophash[i] != top {
                // 碰到emptyRest直接返回
                                if b.tophash[i] == emptyRest {
                                        break search
                                }
                                continue
                        }
                        k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
                        if !t.key.equal(key, k) {
                                continue
                        }

                        e := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                        // 下面与go的内存管理相关，我们这里用伪代码，实际上没有这个函数
            clearMemory(e)

            // 设置桶的状态为emptyOne
                        b.tophash[i] = emptyOne

            // 下面代码就是更新空cell的状态逻辑

            // 没有后一个cell或者后一个cell是emptyOne状态，不需要更新cell状态
            if i == bucketCnt-1 {
                                if b.overflow(t) != nil && b.overflow(t).tophash[0] != emptyRest {
                                        goto notLast
                                }
                        } else {
                                if b.tophash[i+1] != emptyRest {
                                        goto notLast
```

```
                                }
                        }

                // 开始回溯迭代
                        for {
                                b.tophash[i] = emptyRest
                                if i == 0 {
                        // 回溯到第一个桶的第一个entry了
                                        if b == bOrig {
                                                break
                                        }
                                        c := b
                        // 如果回溯到当前桶的第一个cell了，我们需要将桶定位到前一个桶
                                        for b = bOrig; b.overflow(t) != c; b = b.overflow(t) {
                                        }
                                        i = bucketCnt - 1
                                } else {
                                        i--             //从后往前遍历cell
                                }
                        // 碰到非空cell，停止迭代
                                if b.tophash[i] != emptyOne {
                                        break
                                }
                        }
                notLast:
                        h.count--
                        break search
                }
        }
}
```

理解了更新empty状态的过程后，是很容易读懂的。

还有一点注意的是，如果正处于扩容状态，务必要在删除前保证key对应的旧桶利用`growwork`函数迁移完毕，再将迁移后的新桶删除，这和插入一样是为了防止查找时出现错误。

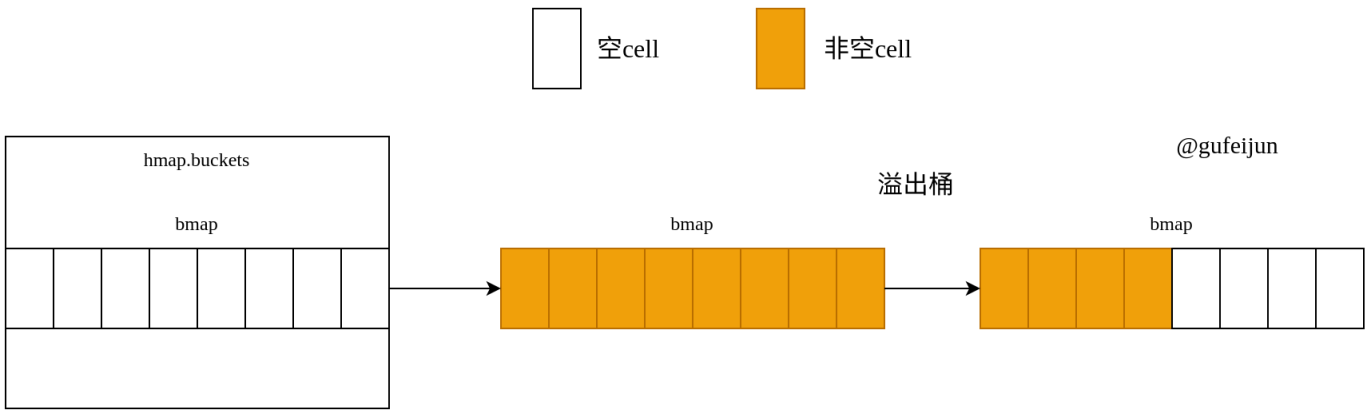
扩容

go语言的扩容分为两种：

- 一种是负载因子>6.5时，触发的增量扩容，即让新桶的数量变为旧桶的2倍。
- 另一种是负载因子依旧小于6.5但溢出桶过多即溢出桶个数大于`1<<min(h.B,15)`时，触发等量扩容，新桶的数量和旧桶一样。

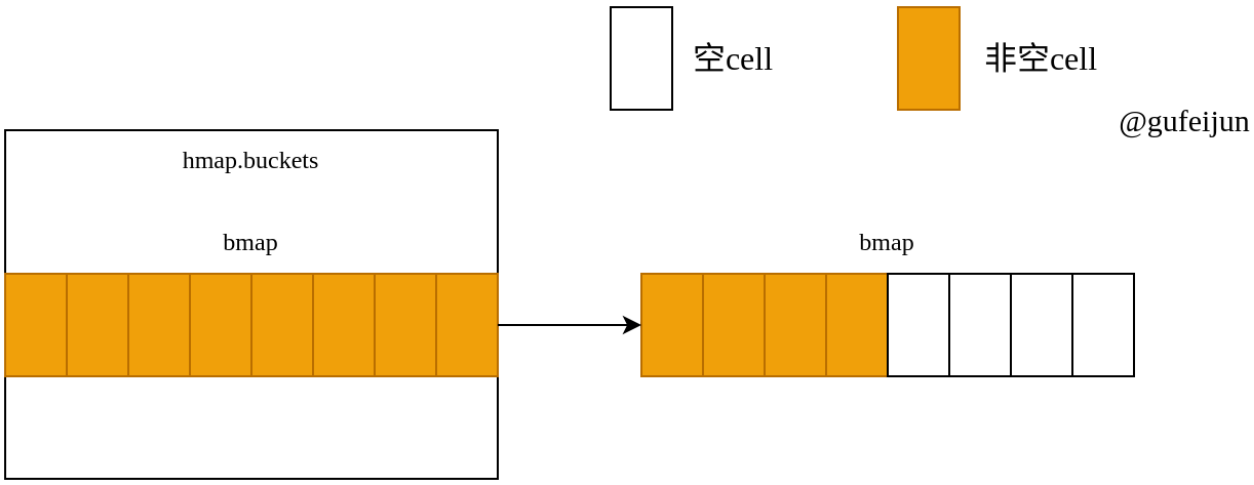
第一种很容易理解，但第二种就挺让人匪夷所思的，事实上导致如此的根本原因是，go语言的每个桶能存取多个键值，而不单单只能存取1个。

我们来看一种极端状态：



如果在连续插入20个同义词key的情况下，再连续删除先插入的8个同义词key，就会出现上图的情况。这时前面一个桶完全处于闲置的状态，浪费内存。除此之外，这时如果去查找一个键，还必须将这个桶的8个空cell全遍历一遍，降低了查找效率。所以一旦出现溢出桶过多

的现象，会触发等量扩容，将这些零散的键再紧密的排列到新桶中，达到以下效果：



扩容函数为hashGrow，它只做了分配新桶的工作，实际上的迁移桶的过程在growWork以及evacuate中：

```
func hashGrow(t *maptype, h *hmap) {
        bigger := uint8(1)
    // 如果是等量扩容
        if !overLoadFactor(h.count+1, h.B) {
                bigger = 0
                h.flags |= sameSizeGrow
        }
        oldbuckets := h.buckets
    // 增量扩容中新桶变为旧桶两倍，等量扩容是数量不变
        newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)

        h.B += bigger
        h.flags = flags
        h.oldbuckets = oldbuckets       // 保存旧桶
        h.buckets = newbuckets
        h.nevacuate = 0
        h.noverflow = 0

        if h.extra != nil && h.extra.overflow != nil {
                h.extra.oldoverflow = h.extra.overflow
                h.extra.overflow = nil
        }
        if nextOverflow != nil {
                if h.extra == nil {
                        h.extra = new(mapextra)
                }
                h.extra.nextOverflow = nextOverflow
        }
}
```

逻辑很简单，hashGrow只会在mapassign中插入新键值且达到上文讲的扩容条件时触发。

在插入和删除操作中，如果此时map正处于扩容，则每一次操作都会利用growWork函数迁移最多两个桶，需要注意的是这个桶不单单指一个桶，如果这个桶还有同义词溢出桶，这些溢出桶也会跟着迁移，换个说法就是还包括桶后的拉链。

growWork如下：

```
//一次最多迁移两个桶
func growWork(t *maptype, h *hmap, bucket uintptr) {
        // 迁移指定的旧桶
        evacuate(t, h, bucket&h.oldbucketmask())

        // 再迁移一个桶
```

```
        if h.growing() {
        // nevacuate就是记录迁移的进度，所有序号小于这个的桶都已经迁移
                evacuate(t, h, h.nevacuate)
        }
}
```

evacuate才实际完成了迁移工作。

对于等量扩容来说，迁移一个键值的过程很简单，一个键对应的旧桶序号一定等于新桶的序号，因为桶数目不变，哈希值取模的结果一样。

但对于增量扩容就不一样，新桶数目是旧桶的两倍，一旦桶数目变化后，hash值取模的结果可能会发生变化，cell就不能一股脑的迁移对应旧桶序号的新桶中。这个应该存放数据的新桶序号有两种情况：第一种与旧桶序号相同，另外一种等于旧桶序号+旧桶的数目，前者处于新桶的前一半，后者处于新桶的后一半中，这个理论证明也很简单：

假设旧桶的数目为2^n，新桶的数目为2^{n+1}，hash的二进制为$x_n x_{n-1}...x_1 x_0 b$。

hash确定旧桶序号：$x_n x_{n-1}...x_1 x_0 b \ \& \underbrace{1..1}_{n个} b = x_{n-1}...x_1 x_0 b$

hash确定新桶序号：$x_n x_{n-1}...x_1 x_0 b \ \& \underbrace{1..1}_{n+1个} b = x_n...x_1 x_0 b$

当hash的第n+1位x_n为0时，旧桶序号和新桶序号相同，否则新桶序号等于旧桶序号+2^n

所以思路很清晰了，到底是该迁移到新桶的前一半还是后一半，就看hash的第n+1位到底为1还是0，这是理解后续代码的关键。

接下来看evacuate函数：

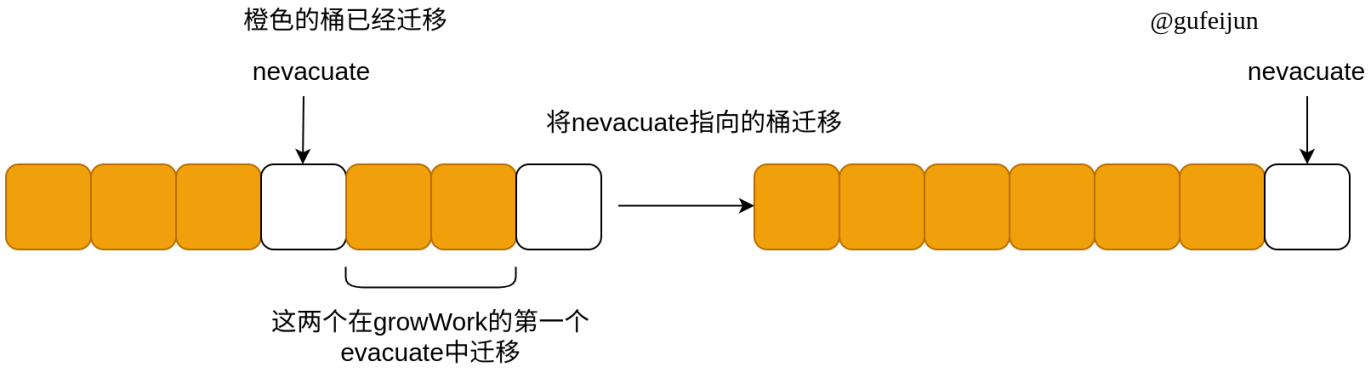
```
// 代表了cell的迁移目的地
type evacDst struct {
        b *bmap          // 迁移到的目的桶
        i int            // 迁移到的cell在桶内的序号
        k unsafe.Pointer // 迁移到的cell的key位置
        e unsafe.Pointer // 迁移到的cell的value位置
}

//把桶oldbucket以及这个桶的所有溢出桶都迁移，oldbucket是待迁移旧桶的序号
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
        b := (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
        newbit := h.noldbuckets()       // 旧桶的个数
        if !evacuated(b) {
                // xy就分别代表新桶中前半部分和后半部分中的目的地
                var xy [2]evacDst
                x := &xy[0]
                x.b = (*bmap)(add(h.buckets, oldbucket*uintptr(t.bucketsize)))
                x.k = add(unsafe.Pointer(x.b), dataOffset)
                x.e = add(x.k, bucketCnt*uintptr(t.keysize))

        // 对于非等量扩容，原先的entry可能到新buckets的前半部分对应桶或者后半部分对应桶
        // xy[0]代表前半部分，xy[1]是后半部分
                if !h.sameSizeGrow() {
                        y := &xy[1]
                        y.b = (*bmap)(add(h.buckets, (oldbucket+newbit)*uintptr(t.bucketsize)))
                        y.k = add(unsafe.Pointer(y.b), dataOffset)
                        y.e = add(y.k, bucketCnt*uintptr(t.keysize))
                }

        // 前面的代码就是先把两个可能的迁移位置记录下来，下面再确定该迁移到哪

        // 把这个桶以及后续的溢出桶都移动
                for ; b != nil; b = b.overflow(t) {
                        k := add(unsafe.Pointer(b), dataOffset)
```

```
                        e := add(k, bucketCnt*uintptr(t.keysize))
                        for i := 0; i < bucketCnt; i, k, e = i+1, add(k, uintptr(t.keysize)), add(e, uintptr(t.elemsize)) {
                                top := b.tophash[i]
                                if isEmpty(top) {
                                        b.tophash[i] = evacuatedEmpty
                                        continue
                                }

                                var useY uint8
                                if !h.sameSizeGrow() {
                    // 计算hash
                                        hash := t.hasher(k, uintptr(h.hash0))

                    // 判断第n=1位是否为0
                    if hash&newbit != 0 {
                        useY = 1
                    }
                                }

                                b.tophash[i] = evacuatedX + useY // evacuatedX + 1 == evacuatedY
                                dst := &xy[useY]        // dst就是确定的目的地

                // 目的桶的8个cell满了
                                if dst.i == bucketCnt {
                    // 生成一个新溢出桶来存
                                        dst.b = h.newoverflow(t, dst.b)
                                        dst.i = 0
                                        dst.k = add(unsafe.Pointer(dst.b), dataOffset)
                                        dst.e = add(dst.k, bucketCnt*uintptr(t.keysize))
                                }
                                dst.b.tophash[dst.i&(bucketCnt-1)] = top // 设置tophash

                                dst.i++

                //指向下一个空cell位置
                                dst.k = add(dst.k, uintptr(t.keysize))
                                dst.e = add(dst.e, uintptr(t.elemsize))
                        }
                }
        }

        if oldbucket == h.nevacuate {
        // 见后文讲解
                advanceEvacuationMark(h, t, newbit)
        }
}
```

捋清楚增量扩容中迁移的细节后，很容易理解上述的代码。

我们再叨唠一句迁移的过程，在插入和删除某个键的操作中，会触发growWork函数，它会调用两次evacuate从而迁移两个桶，第一个桶为插入或删除操作中键对应的旧桶，第二个是序号为nevacuate的桶，nevacuate这个属性记录了迁移的进度，所有小于这个序号的桶都已经被迁移。

很显然的是，第一个迁移的桶的序号是很随机的，由用户指定的key决定，可能本身就大于等于nevacuate，也就是可能出现下图的左边情况：



我们迁移nevacuate指向的桶后，不应该仅仅只是让nevacuate简单+1，指向下一个桶，因为下一个桶可能已经被growWork的第一个evacuate函数迁移，而是应该不断向后看，找到第一个未迁移的桶。这样就能让growWork函数的第二个evacuate函数每次都推进迁移的工作。

advanceEvacuationMark函数就起到了这个作用：

```
//newbit是旧桶的数量
func advanceEvacuationMark(h *hmap, t *maptype, newbit uintptr) {
        h.nevacuate++

    // 一次最多向后看1024个桶
        stop := h.nevacuate + 1024
        if stop > newbit {
                stop = newbit
        }
    // 将nevacuate指向第一个未迁移的桶
        for h.nevacuate != stop && bucketEvacuated(t, h, h.nevacuate) {
                h.nevacuate++
        }
    // 如果所有旧桶全部迁移完
        if h.nevacuate == newbit {
                h.oldbuckets = nil      // 释放旧桶
                if h.extra != nil {
                        h.extra.oldoverflow = nil
                }
                h.flags &^= sameSizeGrow
        }
}
```

总结

实话实说，当抓住核心思想以及主线脉络之后，阅读相关的源码都会得心应手。

本系列采取的是由易至难的叙述思路，通过第一章实现一个最简单的哈希表方式，帮助最短时间抓住哈希表和拉链法的本质，然后再进阶地引导阅读go语言map的源码，相信读者都能很快提升对map的理解，不会有一头雾水的感觉。

下一小节中，我们将好好探讨如何构建一个高性能的并发安全map。

系列目录