Go源码阅读——map.go

Go Map实现

map.go文件包含Go的映射类型的实现。

映射只是一个哈希表。数据被安排在一系列存储桶中。每个存储桶最多包含8个键/元素对。哈希的低位用于选择存储桶。每个存储桶包含每个哈希的一些高阶位，以区分单个存储桶中的条目。

如果有8个以上的键散列到存储桶中，则我们会链接到其他存储桶。

当散列表增加时，我们将分配一个两倍大数组作为新的存储桶。将存储桶以增量方式从旧存储桶阵列复制到新存储桶阵列。

映射迭代器遍历存储桶数组，并按遍历顺序返回键（存储桶#，然后是溢出链顺序，然后是存储桶索引）。为了维持迭代语义，我们绝不会在键的存储桶中移动键（如果这样做，键可能会返回0或2次）。在扩展表时，迭代器将保持对旧表的迭代，并且必须检查新表是否将要迭代的存储桶（“撤离”）到新表中。

选择loadFactor：太大了，我们有很多溢出桶，太小了，我们浪费了很多空间。一些不同负载的统计信息：（64位，8字节密钥和elems）

```
  loadFactor    %overflow  bytes/entry     hitprobe    missprobe
        4.00         2.13        20.77         3.00         4.00
        4.50         4.05        17.30         3.25         4.50
        5.00         6.85        14.77         3.50         5.00
        5.50        10.55        12.94         3.75         5.50
        6.00        15.27        11.67         4.00         6.00
        6.50        20.90        10.79         4.25         6.50
        7.00        27.14        10.15         4.50         7.00
        7.50        34.03         9.73         4.75         7.50
        8.00        41.10         9.40         5.00         8.00
%overflow = 具有溢出桶的桶的百分比
bytes/entry = 每个键值对使用的字节数
hitprobe = 查找存在的key时要检查的条目数
missprobe = 查找不存在的key要检查的条目数
```

数据结构

重要常量

```
const (
    // 桶可以容纳的最大键/值对数量。
    bucketCntBits = 3
    bucketCnt     = 1 << bucketCntBits

    // 触发增长的存储桶的最大平均负载为6.5。表示为loadFactorNum/loadFactDen，以允许整数数学运算。
    loadFactorNum = 13
    loadFactorDen = 2
```

```
    // 保持内联的最大键或elem大小（而不是每个元素的malloc分配）。
    // 必须适合uint8。
    // 快速版本不能处理大问题 - cmd/compile/internal/gc/walk.go中快速版本的临界大小最多必须是这个元素。
    maxKeySize  = 128
    maxElemSize = 128

    // 数据偏移量应为bmap结构的大小，但需要正确对齐。对于amd64p32，
    // 即使指针是32位，这也意味着64位对齐。
    dataOffset = unsafe.Offsetof(struct {
        b bmap
        v int64
    }{}.v)

    // 可能的tophash值。我们为特殊标记保留一些可能性。
    // 每个存储桶（包括其溢出存储桶，如果有的话）在迁移状态下将具有全部或没有条目
    // （除了evacuate()方法期间，该方法仅在映射写入期间发生，因此在此期间没有其他人可以观察该映射）。
    // 所以合法的 tophash(指计算出来的那种)，最小也应该是4，小于4的表示的都是我们自己定义的状态值

    // 此单元格是空的，并且不再有更高索引或溢出的非空单元格。
    emptyRest      = 0
    // 这个单元格是空的
    emptyOne       = 1
    // 键/元素有效。条目已被迁移到大表的前半部分。
    evacuatedX     = 2
    // 与上述相同，但迁移到大表的后半部分。
    evacuatedY     = 3
    // 单元格是空的，桶已已经被迁移。
    evacuatedEmpty = 4
    // 一个正常填充的单元格的最小tophash
    minTopHash     = 5

    // 标志位
    iterator     = 1 // 可能有一个使用桶的迭代器
    oldIterator  = 2 // 可能有一个使用oldbuckets的迭代器
    hashWriting  = 4 // 一个goroutine正在写映射
    sameSizeGrow = 8 // 当前的映射增长是到一个相同大小的新映射

    noCheck = 1<<(8*sys.PtrSize) - 1 // 用于迭代器检查的哨兵桶ID
)

const maxZero = 1024 // 必须与cmd/compile/internal/gc/walk.go:zeroValSize中的值匹配
var zeroVal [maxZero]byte // 用于：1、指针空时，返回unsafe.Pointer；2、用于帮助判断空指针；3、防止指针越界
```

存储结构定义

hmap是go中map结构的定义，其内容如下

```
type hmap struct {
    // 注意：hmap的格式也编码在cmd/compile/internal/gc/reflect.go中。确保这与编译器的定义保持同步。
    // #存活元素==映射的大小。必须是第一个（内置len()使用）
    count     int
    flags     uint8
    // 桶数的log_2（最多可容纳loadFactor * 2 ^ B个元素，再多就要扩容）
    B         uint8
```

```
    // 溢出桶的大概数量；有关详细信息，请参见incrnoverflow
    noverflow uint16
    // 哈希种子
    hash0     uint32 // hash seed

    // 2^B个桶的数组。如果count == 0，则可能为nil。
    buckets    unsafe.Pointer
    // 上一存储桶数组，只有当前桶的一半大小，只有在增长时才为非nil
    oldbuckets unsafe.Pointer
    // 迁移进度计数器（小于此的桶表明已被迁移）
    nevacuate  uintptr

    // 可选择字段，溢出桶的内容全部在这里
    extra *mapextra
}
```

mapextra是ma的溢出数据的定义，内容如下：

```
/**
 * mapextra包含并非在所有map上都存在的字段。
 **/
type mapextra struct {
    // 如果key和elem都不包含指针并且是内联的，则我们将存储桶类型标记为不包含指针。这样可以避免扫描此类映射。
    // 但是，bmap.overflow是一个指针。为了使溢出桶保持活动状态，我们将指向所有溢出桶的指针存储在
hmap.extra.overflow
    // 和hmap.extra.oldoverflow中。仅当key和elem不包含指针时，才使用overflow和oldoverflow。
    // overflow包含hmap.buckets的溢出桶。 oldoverflow包含hmap.oldbuckets的溢出存储桶。
    // 间接允许在Hiter中存储指向切片的指针。
    overflow    *[]*bmap
    oldoverflow *[]*bmap

    // nextOverflow拥有一个指向空闲溢出桶的指针。
    nextOverflow *bmap
}
```

bmap是map的桶定义，其他内容如下

```
/**
 * go映射的桶结构
 **/
type bmap struct {
    // tophash通常包含此存储桶中每个键的哈希值的最高字节。如果tophash[0] < minTopHash，
    // 随后是bucketCnt键，再后是bucketCnt元素。
    tophash [bucketCnt]uint8
    // 注意：将所有键打包在一起，然后将所有elems打包在一起，使代码比交替key/elem/key/elem/...复杂一些，
    // 但是它使我们可以省去填充，例如，映射[int64] int8。后跟一个溢出指针。
}
```

hiter是map的替代器定义，其他内容如下

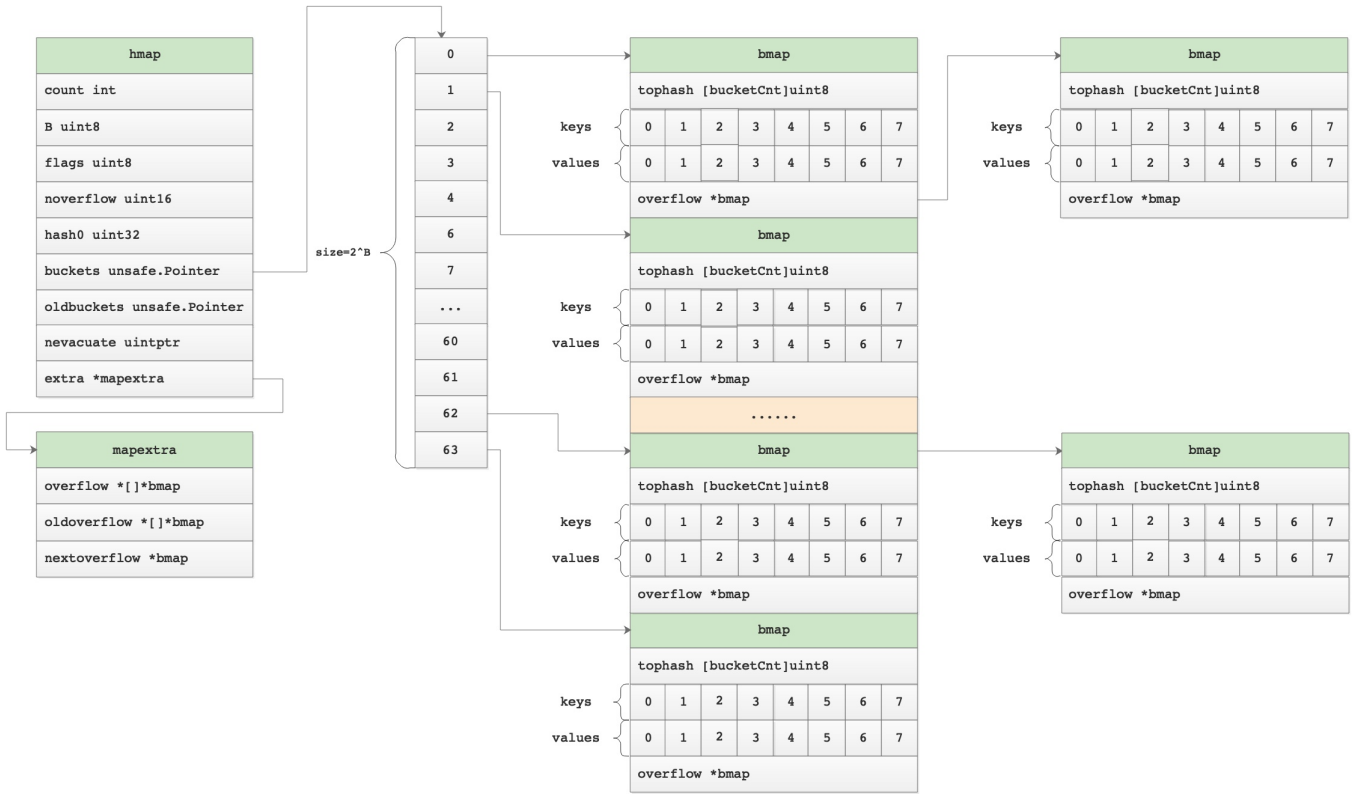
```
/**
 * 哈希迭代结构。
 * 如果修改了hiter，还请更改cmd/compile/internal/gc/reflect.go来指示此结构的布局。
 **/
type hiter struct {
    // 必须处于第一位置。写nil表示迭代结束（请参阅cmd/internal/gc/range.go）。
    key         unsafe.Pointer
    // 必须位于第二位置（请参阅cmd/internal/gc/range.go）。
```

```
    elem        unsafe.Pointer
    t           *maptype // map类型
    h           *hmap
    // hash_iter初始化时的bucket指针
    buckets     unsafe.Pointer
    // 当前迭代的桶
    bptr        *bmap
    // 使hmap.buckets溢出桶保持活动状态
    overflow    *[]*bmap
    // 使hmap.oldbuckets溢出桶保持活动状态
    oldoverflow *[]*bmap
    // 存储桶迭代始于指针位置
    startBucket uintptr         // bucket iteration started at
    // 从迭代期间开始的桶内距离start位置的偏移量（应该足够大以容纳bucketCnt-1）
    offset      uint8
    // 已经从存储桶数组的末尾到开头缠绕了，迭代标记，为true说明迭代已经可以结束了
    wrapped     bool
    B           uint8 // 与hmap中的B对应
    i           uint8
    bucket      uintptr
    checkBucket uintptr
}
```

其他数据结构

map中还使用到maptype数据结构。可以说明可见：https://github.com/Wang-Jun-Chao/go-source-read/blob/master/reflect/type_go.md

map存储结构示意图



创建map

go map创建

```
make(map[k]v), (map[k]v, hint)
```

小\map创建

```
/**
 * 当在编译时已知hint最多为bucketCnt并且需要在堆上分配映射时，
 * makemap_small实现了make(map[k]v)和make(map[k]v, hint)的Go映射创建。
 **/
func makemap_small() *hmap {
    h := new(hmap)
    h.hash0 = fastrand()
    return h
}
```

大map创建

```
/**
 * 创建hmap，主要是对hint参数进行判定，不超出int可以表示的值
 **/
func makemap64(t *maptype, hint int64, h *hmap) *hmap {
    if int64(int(hint)) != hint {
        hint = 0
    }
    return makemap(t, int(hint), h)
}
/**
 * makemap实现Go map创建，其实现方法是make(map[k]v)和make(map[k]v, hint)。
 * 如果编译器认为map和第一个 bucket 可以直接创建在栈上，h和bucket 可能都是非空
 * 如果h!= nil，则可以直接在h中创建map。
 * 如果h.buckets != nil，则指向的存储桶可以用作第一个存储桶。
 **/
func makemap(t *maptype, hint int, h *hmap) *hmap {
    // 计算所需要的内存空间，并且判断是是否会有溢出
    mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
    if overflow || mem > maxAlloc { // 有溢出或者分配的内存大于最大分配内存
        hint = 0
    }

    // 初始化hmap
    if h == nil {
        h = new(hmap)
    }
    h.hash0 = fastrand() // 设置随机数

    // 找到用于保存请求的元素数的大小参数B。
    // 对于hint<0，由于hint < bucketCnt，overLoadFactor返回false。
    B := uint8(0)
    // 按照提供的元素个数，找一个可以放得下这么多元素的 B 值
    for overLoadFactor(hint, B) {
        B++
    }
    h.B = B

    // 如果B == 0，则分配初始哈希表，则稍后（在mapassign中）延迟分配buckets字段。
    // 如果hint为零，则此内存可能需要一段时间。
    // 因为如果 hint 很大的话，对这部分内存归零会花比较长时间
    if h.B != 0 {
```

```
        var nextOverflow *bmap
        // 创建数据桶和溢出桶
        h.buckets, nextOverflow = makeBucketArray(t, h.B, nil)
        if nextOverflow != nil { // 溢出桶不为空就将溢出桶挂到附加数据上
            h.extra = new(mapextra)
            h.extra.nextOverflow = nextOverflow
        }
    }

    return h
}
```

实际选用哪个函数很复杂，涉及的判定变量有：

- 1、hint值，以及hint最终类型：
- 2、逃逸分析结果
- 3、BUCKETSIZE=8

创建map选择的map函数分析在代码：/usr/local/go/src/cmd/compile/internal/gc/walk.go:1218中

```
case OMAKEMAP:
        t := n.Type
        hmapType := hmap(t)
        hint := n.Left

        // var h *hmap
        var h *Node
        if n.Esc == EscNone {
            // Allocate hmap on stack.

            // var hv hmap
            hv := temp(hmapType)
            zero := nod(OAS, hv, nil)
            zero = typecheck(zero, ctxStmt)
            init.Append(zero)
            // h = &hv
            h = nod(OADDR, hv, nil)

            // Allocate one bucket pointed to by hmap.buckets on stack if hint
            // is not larger than BUCKETSIZE. In case hint is larger than
            // BUCKETSIZE runtime.makemap will allocate the buckets on the heap.
            // Maximum key and elem size is 128 bytes, larger objects
            // are stored with an indirection. So max bucket size is 2048+eps.
            if !Isconst(hint, CTINT) ||
                hint.Val().U.(*Mpint).CmpInt64(BUCKETSIZE) <= 0 {
                // var bv bmap
                bv := temp(bmap(t))

                zero = nod(OAS, bv, nil)
                zero = typecheck(zero, ctxStmt)
                init.Append(zero)

                // b = &bv
                b := nod(OADDR, bv, nil)

                // h.buckets = b
```

```
            bsym := hmapType.Field(5).Sym // hmap.buckets see reflect.go:hmap
            na := nod(OAS, nodSym(ODOT, h, bsym), b)
            na = typecheck(na, ctxStmt)
            init.Append(na)
        }
    }

    if Isconst(hint, CTINT) && hint.Val().U.(*Mpint).CmpInt64(BUCKETSIZE) <= 0 {
        // Handling make(map[any]any) and
        // make(map[any]any, hint) where hint <= BUCKETSIZE
        // special allows for faster map initialization and
        // improves binary size by using calls with fewer arguments.
        // For hint <= BUCKETSIZE overLoadFactor(hint, 0) is false
        // and no buckets will be allocated by makemap. Therefore,
        // no buckets need to be allocated in this code path.
        if n.Esc == EscNone {
            // Only need to initialize h.hash0 since
            // hmap h has been allocated on the stack already.
            // h.hash0 = fastrand()
            rand := mkcall("fastrand", types.Types[TUINT32], init)
            hashsym := hmapType.Field(4).Sym // hmap.hash0 see reflect.go:hmap
            a := nod(OAS, nodSym(ODOT, h, hashsym), rand)
            a = typecheck(a, ctxStmt)
            a = walkexpr(a, init)
            init.Append(a)
            n = convnop(h, t)
        } else {
            // Call runtime.makehmap to allocate an
            // hmap on the heap and initialize hmap's hash0 field.
            fn := syslook("makemap_small")
            fn = substArgTypes(fn, t.Key(), t.Elem())
            n = mkcall1(fn, n.Type, init)
        }
    } else {
        if n.Esc != EscNone {
            h = nodnil()
        }
        // Map initialization with a variable or large hint is
        // more complicated. We therefore generate a call to
        // runtime.makemap to initialize hmap and allocate the
        // map buckets.

        // When hint fits into int, use makemap instead of
        // makemap64, which is faster and shorter on 32 bit platforms.
        fnname := "makemap64"
        argtype := types.Types[TINT64]

        // Type checking guarantees that TIDEAL hint is positive and fits in an int.
        // See checkmake call in TMAP case of OMAKE case in OpSwitch in typecheck1
function.
        // The case of hint overflow when converting TUINT or TUINTPTR to TINT
        // will be handled by the negative range checks in makemap during runtime.
        if hint.Type.IsKind(TIDEAL) || maxintval[hint.Type.Etype].Cmp(maxintval[TUINT])
<= 0 {
            fnname = "makemap"
            argtype = types.Types[TINT]
```

```
        }

        fn := syslook(fnname)
        fn = substArgTypes(fn, hmapType, t.Key(), t.Elem())
        n = mkcall1(fn, n.Type, init, typename(n.Type), conv(hint, argtype), h)
    }
```

访问map元素

go中访问map元素是通过map[key]的方式进行，真正的元素访问在go语言中有如下几个方法

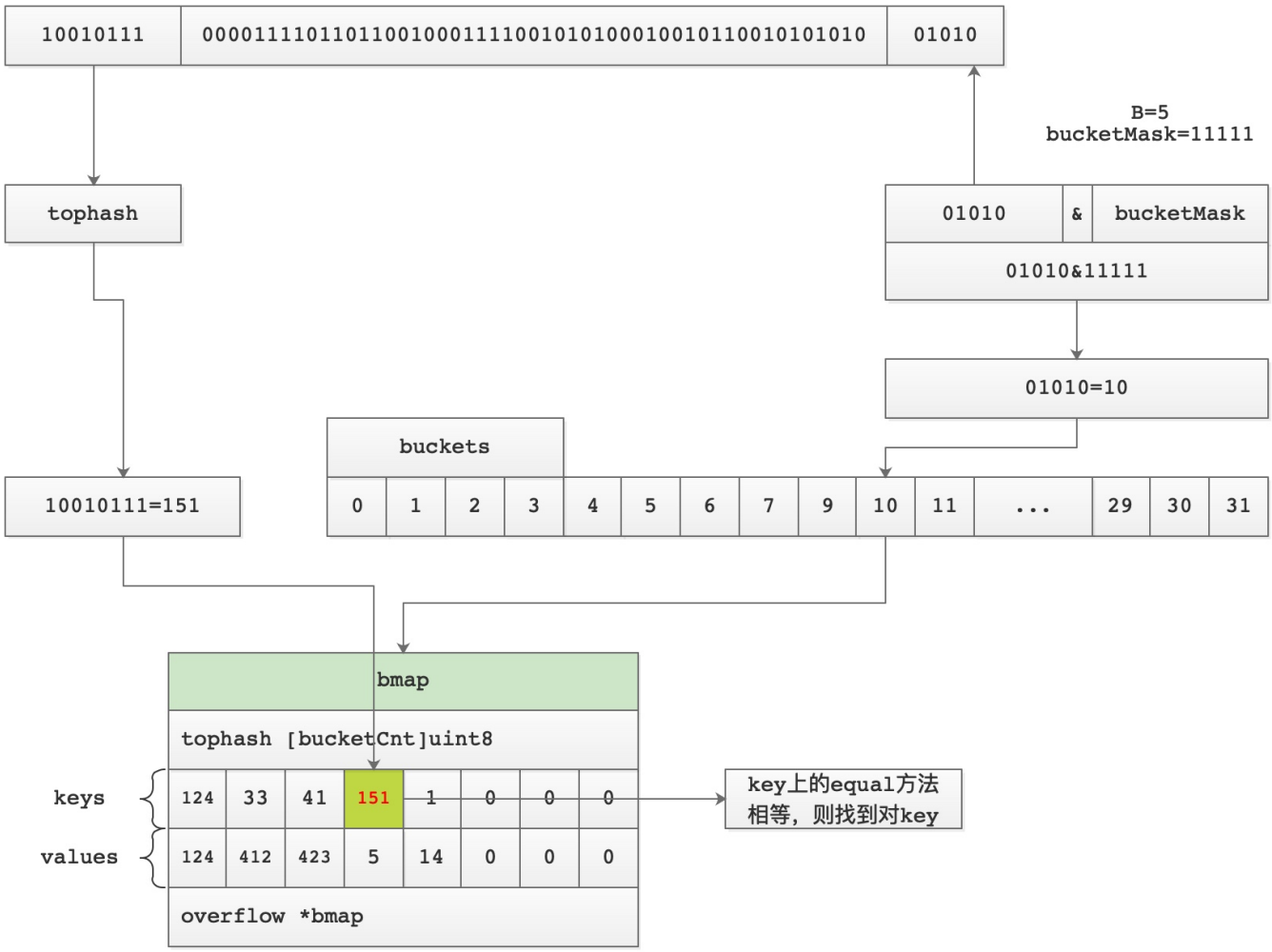
- func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {...}：mapaccess1返回指向h[key]的指针。从不返回nil，如果键不在映射中，它将返回对elem类型的零对象的引用。对应go写法：v := m[k]
- func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool) {...}：方法同mapaccess1，仅多返回一个值用于表示是否找到对应元素。对应go写法：v, ok := m[k]
- func mapaccessK(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, unsafe.Pointer) {...}：返回key和elem。由map迭代器使用，与mapaccess1相类似，只多返回了一个key。。对应go写法:k,v := range m[k]。
- func mapaccess1_fat(t *maptype, h *hmap, key, zero unsafe.Pointer) unsafe.Pointer {...}：mapaccess1的包装方法，获取map中key对应的值，如果没有找到就返回zero。对应go写法：v := m[k]
- func mapaccess2_fat(t *maptype, h *hmap, key, zero unsafe.Pointer) (unsafe.Pointer, bool) {...}：mapaccess2的包装方法，获取map中key对应的值，如果没有找到就返回zero，并返回是否找到标记。。对应go写法：v, ok := m[k]

其中mapaccess1，mapaccess2，mapaccessK方法大同小异，我们选择mapaccesssK进行分析：

```
/**
 * 返回key和elem。由map迭代器使用，与mapaccess1相类似，只多返回了一个key
 * @param
 * @return
 **/
func mapaccessK(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, unsafe.Pointer) {
    if h == nil || h.count == 0 { // map 为空，或者元素数为 0，直接返回未找到
        return nil, nil
    }
    hash := t.hasher(key, uintptr(h.hash0)) // 计算hash值
    // 计算掩码：(1<<h.B)- 1，B=3，m=111；B=4，m=1111
    m := bucketMask(h.B)
    // 计算桶数
    // unsafe.Pointer(uintptr(h.buckets)：基址
    // (hash&m)*uintptr(t.bucketsize))：偏移量，(hash&m)就是桶数
    b := (*bmap)(unsafe.Pointer(uintptr(h.buckets) + (hash&m)*uintptr(t.bucketsize)))
    // h.oldbuckets不为空，说明正在扩容，新的 buckets 里可能还没有老的内容
    // 所以一定要在老的桶里面找，否则有可能可能找不到
    if c := h.oldbuckets; c != nil {
        if !h.sameSizeGrow() {
            // 如果不是同大小增长，那么现在的老桶，只有新桶的一半，对应的mask也林减少一位
            m >>= 1
        }
        // 计算老桶的位置
        oldb := (*bmap)(unsafe.Pointer(uintptr(c) + (hash&m)*uintptr(t.bucketsize)))
```

```
26          if !evacuated(oldb) { // 如果没有迁移完，需要从老桶中找
27              b = oldb
28          }
29      }
30      // tophash 取其高 8bit 的值
31      top := tophash(hash)
32  bucketloop:
33      for ; b != nil; b = b.overflow(t) {
34          // 一个 bucket 在存储满8个元素后，就再也放不下了
35          // 这时候会创建新的 bucket挂在原来的bucket的overflow指针成员上
36          for i := uintptr(0); i < bucketCnt; i++ {
37              // 循环对比 bucket 中的 tophash 数组，
38              // 如果找到了相等的 tophash，那说明就是这个 bucket 了
39              if b.tophash[i] != top {
40                  // 如果找到值为emptyRest，说明桶后面是空的，没有值了，
41                  // 无法找到对应的元素，，跳出bucketloop
42                  if b.tophash[i] == emptyRest {
43                      break bucketloop
44                  }
45                  continue
46              }
47              // 到这里说明找到对应的hash值，具体是否相等还要判断对应equal方法
48              // 取k元素
49              k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
50              if t.indirectkey() {
51                  k = *((*unsafe.Pointer)(k))
52              }
53              if t.key.equal(key, k) { // 如果为值，说明真正找到了对应的元素
54                  e := add(unsafe.Pointer(b),
    dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
55                  if t.indirectelem() {
56                      e = *((*unsafe.Pointer)(e))
57                  }
58                  return k, e
59              }
60          }
61      }
62      return nil, nil
63  }
```

元素访问示意图



map元素赋值

map元素的赋值都通过方法mapassign进行

```
/**
 * 与mapaccess类似，但是如果map中不存在key，则为该key分配一个位置。
 * @param
 * @return key对应elem的插入位置指针
 **/
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    if h == nil { // mil map不可以进行赋值
        panic(plainError("assignment to entry in nil map"))
    }
    if raceenabled {
        callerpc := getcallerpc()
        pc := funcPC(mapassign)
        racewritepc(unsafe.Pointer(h), callerpc, pc)
        raceReadObjectPC(t.key, key, callerpc, pc)
    }
    if msanenabled {
        msanread(key, t.key.size)
    }
    if h.flags&hashWriting != 0 { // 不能并发读写
        throw("concurrent map writes")
    }
    hash := t.hasher(key, uintptr(h.hash0)) // 计算hash值

    // 在调用t.hasher之后设置hashWriting，因为t.hasher可能会出现panic情况，在这种情况下，我们实际上并未
```

```
    执行写入操作。
25      h.flags ^= hashWriting
26
27      if h.buckets == nil { // 如果桶为空，就创建大小为1的桶
28          h.buckets = newobject(t.bucket) // newarray(t.bucket, 1)
29      }
30
31  again:
32      // 计算桶的位置，实际代表第几个桶，(1<<h.B)-1
33      bucket := hash & bucketMask(h.B)
34      if h.growing() { // 是否在扩容
35          growWork(t, h, bucket) // 进行扩容处理
36      }
37      // 计算桶的位置，指针地址
38      b := (*bmap)(unsafe.Pointer(uintptr(h.buckets) + bucket*uintptr(t.bucketsize)))
39      // 计算高8位hash
40      top := tophash(hash)
41
42      var inserti *uint8 // 记录
43      var insertk unsafe.Pointer
44      var elem unsafe.Pointer
45  bucketloop:
46      for {
47          for i := uintptr(0); i < bucketCnt; i++ { // 遍历对应桶中的元素
48              if b.tophash[i] != top {
49                  // 在 b.tophash[i] != top 的情况下
50                  // 理论上有可能会是一个空槽位
51                  // 一般情况下 map 的槽位分布是这样的，e 表示 empty:
52                  // [h1][h2][h3][h4][h5][e][e][e]
53                  // 但在执行过 delete 操作时，可能会变成这样:
54                  // [h1][h2][e][e][h5][e][e][e]
55                  // 所以如果再插入的话，会尽量往前面的位置插
56                  // [h1][h2][e][e][h5][e][e][e]
57                  //          ^
58                  //          ^
59                  //       这个位置
60                  // 所以在循环的时候还要顺便把前面的空位置先记下来
61                  if isEmpty(b.tophash[i]) && inserti == nil {
62                      inserti = &b.tophash[i]
63                      insertk = add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
64                      elem = add(unsafe.Pointer(b),
    dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
65                  }
66                  if b.tophash[i] == emptyRest { // i及之后的槽位都为空，不需要再进行处理了
67                      break bucketloop
68                  }
69                  continue
70              }
71              // 已经找到一个i使得b.tophash[i] == top
72              // 找到对应的k
73              k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
74              if t.indirectkey() {
75                  k = *((*unsafe.Pointer)(k))
76              }
77              // 已经存储的key和要传入的key不相等，说明发生了hash碰撞
78              if !t.key.equal(key, k) {
```

```
            continue
        }
        // 已经有一个key映射。更新它。
        if t.needkeyupdate() {
            typedmemmove(t.key, k, key)
        }
        elem = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
        goto done
    }
    // bucket的8个槽没有满足条件的能插入或者能更新的，去overflow里继续找
    ovf := b.overflow(t)
    // 如果overflow为 nil，说明到了overflow链表的末端了
    if ovf == nil {
        break
    }
    // 赋值为链表的下一个元素，继续循环
    b = ovf
}

// 找不到键的映射。分配新单元格并添加条目。

// 如果我们达到最大负载因子，或者我们有太多的溢出桶，而我们还没有处于增长过程，那就开始增长。
if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
    // hashGrow的时候会把当前的bucket放到oldbucket里
    // 但还没有开始分配新的bucket，所以需要到again重试一次
    // 重试的时候在growWork里会把这个key的bucket优先分配好
    hashGrow(t, h)
    goto again // Growing the table invalidates everything, so try again // 扩容表格会使所有内容失效，因此请重试
}

if inserti == nil {
    // 前面在桶里找的时候，没有找到能塞这个 tophash 的位置
    // 说明当前所有 buckets 都是满的，分配一个新的 bucket
    newb := h.newoverflow(t, b)
    inserti = &newb.tophash[0]
    insertk = add(unsafe.Pointer(newb), dataOffset)
    elem = add(insertk, bucketCnt*uintptr(t.keysize))
}

// 在插入位置存储新的key/elem
if t.indirectkey() { // 插入key
    kmem := newobject(t.key)
    *(*unsafe.Pointer)(insertk) = kmem
    insertk = kmem
}
if t.indirectelem() { // 插入elem
    vmem := newobject(t.elem)
    *(*unsafe.Pointer)(elem) = vmem
}
typedmemmove(t.key, insertk, key)
*inserti = top
h.count++

```

```
done:
    if h.flags&hashWriting == 0 {
        throw("concurrent map writes")
    }
    h.flags &^= hashWriting
    if t.indirectelem() {
        elem = *((*unsafe.Pointer)(elem))
    }
    return elem
}
```

mapassign没有对value进行操作，只是返回了需要value的地址信息，到底在哪里进行了操作，我们以下面的程序为例进行说明：map_go_summary.go

```
package main

import "fmt"

func main() {
    type P struct {
        Age [16]int
    }

    var a = make(map[P]int, 17)

    a[P{}] = 9999999

    for i := 0; i < 16; i++ {
        p := P{}
        p.Age[0] = i
        a[p] = i
    }
    fmt.Println(a)
}

```

运行： `go tool compile -N -l -S map_go_summary.go`获得反汇编代码，查看：第12行所做的操作

```
0x0061 00097 (map_go_summary.go:12)     PCDATA  $0, $2
0x0061 00097 (map_go_summary.go:12)     LEAQ    ""..autotmp_4+184(SP), DI
0x0069 00105 (map_go_summary.go:12)     XORPS   X0, X0
0x006c 00108 (map_go_summary.go:12)     PCDATA  $0, $0
0x006c 00108 (map_go_summary.go:12)     DUFFZERO        $266
0x007f 00127 (map_go_summary.go:12)     PCDATA  $0, $1
0x007f 00127 (map_go_summary.go:12)     LEAQ    type.map[""".P·1]int(SB), AX
0x0086 00134 (map_go_summary.go:12)     PCDATA  $0, $0
0x0086 00134 (map_go_summary.go:12)     MOVQ    AX, (SP)
0x008a 00138 (map_go_summary.go:12)     PCDATA  $0, $1
0x008a 00138 (map_go_summary.go:12)     MOVQ    ""..a+312(SP), AX
0x0092 00146 (map_go_summary.go:12)     PCDATA  $0, $0
0x0092 00146 (map_go_summary.go:12)     MOVQ    AX, 8(SP)
0x0097 00151 (map_go_summary.go:12)     PCDATA  $0, $1
0x0097 00151 (map_go_summary.go:12)     LEAQ    ""..autotmp_4+184(SP), AX
0x009f 00159 (map_go_summary.go:12)     PCDATA  $0, $0
0x009f 00159 (map_go_summary.go:12)     MOVQ    AX, 16(SP)
0x00a4 00164 (map_go_summary.go:12)     CALL    runtime.mapassign(SB) # 调用mapasssgin方法
0x00a9 00169 (map_go_summary.go:12)     PCDATA  $0, $1
```

```
0x00a9 00169 (map_go_summary.go:12)     MOVQ    24(SP), AX
0x00ae 00174 (map_go_summary.go:12)     MOVQ    AX, ""..autotmp_7+328(SP)
0x00b6 00182 (map_go_summary.go:12)     TESTB   AL, (AX)
0x00b8 00184 (map_go_summary.go:12)     PCDATA  $0, $0
0x00b8 00184 (map_go_summary.go:12)     MOVQ    $9999999, (AX) # 进行赋值操作
```

赋值的最后一步实际上是编译器额外生成的汇编指令来完成的。

map删除key

go中删除map语句：delete(m, k)，底层实现在是通过mapdelete进行

```go
/**
 * 删除key
 **/
func mapdelete(t *maptype, h *hmap, key unsafe.Pointer) {
    if raceenabled && h != nil {
        callerpc := getcallerpc()
        pc := funcPC(mapdelete)
        racewritepc(unsafe.Pointer(h), callerpc, pc)
        raceReadObjectPC(t.key, key, callerpc, pc)
    }
    if msanenabled && h != nil {
        msanread(key, t.key.size)
    }
    if h == nil || h.count == 0 {
        if t.hashMightPanic() {
            t.hasher(key, 0) // see issue 23734
        }
        return
    }
    if h.flags&hashWriting != 0 { // 当前map正在被写，不能再写
        throw("concurrent map writes")
    }

    hash := t.hasher(key, uintptr(h.hash0))

    // 在调用t.hasher之后设置hashWriting，因为t.hasher可能会出现panic情况，
    // 在这种情况下，我们实际上并未执行写入（删除）操作。
    h.flags ^= hashWriting

    bucket := hash & bucketMask(h.B)
    if h.growing() {
        growWork(t, h, bucket)
    }
    b := (*bmap)(add(h.buckets, bucket*uintptr(t.bucketsize)))
    bOrig := b
    top := tophash(hash)
search:
    for ; b != nil; b = b.overflow(t) {
        for i := uintptr(0); i < bucketCnt; i++ {
            if b.tophash[i] != top {
                if b.tophash[i] == emptyRest {
                    break search
                }
                continue
            }
```

```
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            k2 := k
            if t.indirectkey() {
                k2 = *((*unsafe.Pointer)(k2))
            }
            if !t.key.equal(key, k2) { // 两个key不相等
                continue
            }
            // 如果是间接指针，则仅清除键。
            if t.indirectkey() {
                *(*unsafe.Pointer)(k) = nil
            } else if t.key.ptrdata != 0 {
                memclrHasPointers(k, t.key.size) // 清除内存数据
            }
            e := add(unsafe.Pointer(b),
dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
            if t.indirectelem() { // elem是间接指针，将指针赋空
                *(*unsafe.Pointer)(e) = nil
            } else if t.elem.ptrdata != 0 { // 元素有指针数据，将清除指针数据
                memclrHasPointers(e, t.elem.size)
            } else { // 清除e的数据
                memclrNoHeapPointers(e, t.elem.size)
            }
            b.tophash[i] = emptyOne // 标记桶为空
            // 如果存储桶现在以一堆emptyOne状态结束，则将其更改为emptyRest状态。
            // 将此功能设为单独的函数会很好，但是for循环当前不可内联。
            if i == bucketCnt-1 {
                // 有下一个溢出桶，并且溢出桶有值
                if b.overflow(t) != nil && b.overflow(t).tophash[0] != emptyRest {
                    goto notLast
                }
            } else { // 没有溢出桶了，但是当前位置之后的位置还有值
                if b.tophash[i+1] != emptyRest {
                    goto notLast
                }
            }
            for { // 当前桶的当前位置之后都没有值了
                b.tophash[i] = emptyRest // 标记当前位置已经空
                if i == 0 {
                    if b == bOrig {
                        // 从初始存储桶开始，我们已经处理完了。
                        break
                    }
                    // 查找上一个存储桶，直到最后一个。
                    c := b
                    for b = bOrig; b.overflow(t) != c; b = b.overflow(t) {
                    }
                    i = bucketCnt - 1
                } else {
                    i--
                }
                if b.tophash[i] != emptyOne {
                    break
                }
            }
        notLast:
```

```
            // hmap 的大小计数 -1
            h.count--
            break search
        }
    }

    if h.flags&hashWriting == 0 {
        throw("concurrent map writes")
    }
    h.flags &^= hashWriting
}
```

map扩容

在mapassign方法中我们看到了扩容的条件

- 1、!h.growing() && (overLoadFactor(h.count+1, h.B)：当前map未进行扩容，但是添加一个元素后，超过负载因子。负载因子是6.5，即：元素个数>=桶个数*6.5，需要进行扩容
- 2、tooManyOverflowBuckets(h.noverflow, h.B)：我们有太多的溢出桶。什么情况下是溢出桶过多：
    - （1）当bucket总数 < 2^15 时，如果overflow的bucket总数 >= bucket的总数，那么我们认为overflow的桶太多了。
    - （2）当bucket总数 >= 2^15时，那我们直接和2^15比较，overflow的bucket >= 2^15时，即认为溢出桶太多了。

两种情况官方采用了不同的解决方法：

- 针对（1），将B+1，进而hmap的bucket数组扩容一倍；
- 针对（2），通过移动bucket内容，使其倾向于紧密排列从而提高bucket利用率。

如果map中有大量的哈希冲突，也会导致落入（2）中的条件，此时对bucket的内容进行移动其实没什么意义，反而会影响性能，所以理论上存在对Go map进行hash碰撞攻击的可能性。

```
/**
 * map扩容
 **/
func hashGrow(t *maptype, h *hmap) {
    // 如果我们达到了负载因子，请扩容。
    // 否则，溢出桶过多，因此保持相同数量的桶并横向“增长”。
    bigger := uint8(1)
    if !overLoadFactor(h.count+1, h.B) { // 增加一个元素，没有超过负载因子
        bigger = 0
        h.flags |= sameSizeGrow
    }
    oldbuckets := h.buckets
    // 创建新桶
    newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)

    flags := h.flags &^ (iterator | oldIterator)
    if h.flags&iterator != 0 {
        flags |= oldIterator
    }
    // 提交扩容 (atomic wrt gc)
    h.B += bigger
    h.flags = flags
```

```
    h.oldbuckets = oldbuckets
    h.buckets = newbuckets
    h.nevacuate = 0
    h.noverflow = 0

    if h.extra != nil && h.extra.overflow != nil {
        // 将当前的溢出桶提升到老一代。
        if h.extra.oldoverflow != nil {
            throw("oldoverflow is not 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
    }

    // 哈希表数据的实际复制是通过growWork()和evacuate()增量完成的。
}

/**
 * makeBucketArray为map数据桶初始化底层数组。
 * 1<<b 是要分配的最小存储桶数。
 * dirtyalloc应该为nil或由makeBucketArray先前使用相同的t和b参数分配的bucket数组。
 * 如果dirtyalloc为nil，则将分配一个新的后备数组，否则，dirtyalloc将被清除并重新用作后备数组。
 * @param
 * @return
 **/
func makeBucketArray(t *maptype, b uint8, dirtyalloc unsafe.Pointer) (buckets
unsafe.Pointer, nextOverflow *bmap) {
    base := bucketShift(b)
    nbuckets := base
    // 对于小b，溢出桶不太可能出现。避免计算的开销。
    if b >= 4 {
        // 加上所需的溢流桶的估计数量，以插入使用此值b的元素的中位数。
        nbuckets += bucketShift(b - 4)
        sz := t.bucket.size * nbuckets
        up := roundupsize(sz) // 计算mallocgc分配的内存
        if up != sz {
            nbuckets = up / t.bucket.size // 计算每个桶的内存大小
        }
    }

    if dirtyalloc == nil {
        // 直接创建数组
        buckets = newarray(t.bucket, int(nbuckets))
    } else {
        // dirtyalloc先前是由上述newarray(t.bucket, int(nbuckets))生成的，但可能不为空。
        buckets = dirtyalloc
        size := t.bucket.size * nbuckets
        // 进行内存清零
        if t.bucket.ptrdata != 0 {
            memclrHasPointers(buckets, size)
```

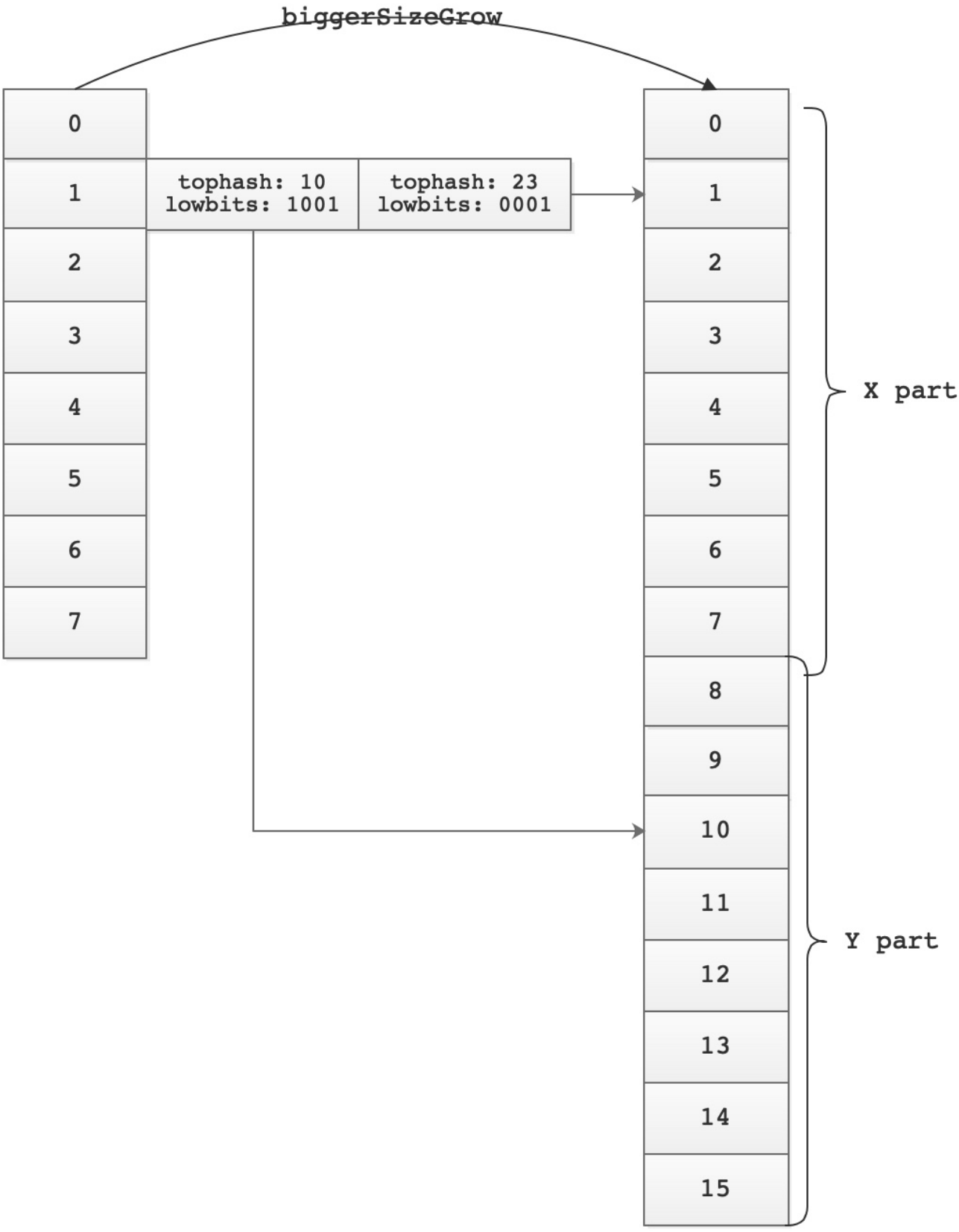
```
78          } else {
79              memclrNoHeapPointers(buckets, size)
80          }
81      }
82
83      // 实际计算的桶数据和最初的桶数不一样
84      if base != nbuckets {
85          // 我们预先分配了一些溢出桶。
86          // 为了使跟踪这些溢出桶的开销降到最低，我们使用以下约定：如果预分配的溢出桶的溢出指针为nil，则通过碰撞指针还有更多可用空间。
87          // 对于最后一个溢出存储区，我们需要一个安全的非nil指针；使用buckets。
88          nextOverflow = (*bmap)(add(buckets, base*uintptr(t.bucketsize))) // 计算下一个溢出桶
89          last := (*bmap)(add(buckets, (nbuckets-1)*uintptr(t.bucketsize))) // 计算最后一个溢出桶
90          last.setoverflow(t, (*bmap)(buckets))
91      }
92      return buckets, nextOverflow
93  }
94
95  /**
96   * 桶增长
97   **/
98  func growWork(t *maptype, h *hmap, bucket uintptr) {
99      // 确保我们迁移将要使用的存储桶对应的旧存储桶
100     evacuate(t, h, bucket&h.oldbucketmask())
101
102     // 迁移一个旧桶，会使过程标记在growing
103     if h.growing() {
104         evacuate(t, h, h.nevacuate)
105     }
106 }
107
108 /**
109  * 进行迁移
110  * @param
111  * @param
112  * @param oldbucket 需要迁移的桶
113  * @return
114  **/
115 func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
116     b := (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
117     newbit := h.noldbuckets() // 值形如111...111
118     if !evacuated(b) {
119         // TODO：如果没有迭代器使用旧的存储桶，则重用溢出存储桶而不是使用新的存储桶。（如果为!oldIterator。）
120
121         // xy包含x和y（低和高）迁移目的地。
122         // x 表示新 bucket 数组的前(low)半部分
123         // y 表示新 bucket 数组的后(high)半部分
124         var xy [2]evacDst
125         x := &xy[0]
126         x.b = (*bmap)(add(h.buckets, oldbucket*uintptr(t.bucketsize)))
127         x.k = add(unsafe.Pointer(x.b), dataOffset)
128         x.e = add(x.k, bucketCnt*uintptr(t.keysize))
129
130         if !h.sameSizeGrow() { // 非同大小增长
131             // 仅当我们变得更大时才计算y指针。否则GC可能会看到错误的指针。
```

```
132             y := &xy[1]
133             y.b = (*bmap)(add(h.buckets, (oldbucket+newbit)*uintptr(t.bucketsize)))
134             y.k = add(unsafe.Pointer(y.b), dataOffset)
135             y.e = add(y.k, bucketCnt*uintptr(t.keysize))
136         }
137 
138         for ; b != nil; b = b.overflow(t) {
139             k := add(unsafe.Pointer(b), dataOffset)
140             e := add(k, bucketCnt*uintptr(t.keysize))
141             for i := 0; i < bucketCnt; i, k, e = i+1, add(k, uintptr(t.keysize)), add(e,
    uintptr(t.elemsize)) {
142                 top := b.tophash[i]
143                 if isEmpty(top) {
144                     b.tophash[i] = evacuatedEmpty
145                     continue
146                 }
147                 if top < minTopHash {
148                     throw("bad map state")
149                 }
150                 k2 := k
151                 if t.indirectkey() {
152                     k2 = *((*unsafe.Pointer)(k2))
153                 }
154                 var useY uint8
155                 if !h.sameSizeGrow() { // 扩容一倍
156                     // 计算散列值以做出迁移决定（是否需要将此key/elem发送到存储桶x或存储桶y）。
157                     hash := t.hasher(k2, uintptr(h.hash0))
158                     if h.flags&iterator != 0 && !t.reflexivekey() && !t.key.equal(k2, k2) {
159                         // 对于一般情况，key必须是自反的，即 key==key，但是对于特殊情况，比如浮点值n1、
    n2（都是NaN），n1==n2是不成立的，对于这部分key，我们使用最低位进行随机选择，让它们到Y部分
160                         // 如果key != key(NaNs)，则哈希可能（可能会）与旧哈希完全不同。而且，它是不可
    重现的。
161                         // 在存在迭代器的情况下，要求具有可重复性，因为我们的迁移决策必须与迭代器所做的任
    何决策相匹配。
162                         // 幸运的是，我们可以自由发送这些key。同样，tophash对于这些key也没有意义。
163                         // 我们让低位的hophash决定迁移。我们为下一个级别重新计算了一个新的随机tophash，
    以便在多次增长之后，
164                         // 这些key将在所有存储桶中平均分配
165                         useY = top & 1
166                         top = tophash(hash)
167                     } else {
168                         // 假设newbit有6位，则newbit=111111
169                         // 如果hash的低6位不0则元素必须去高位
170                         if hash&newbit != 0 {
171                             useY = 1
172                         }
173                     }
174                 }
175 
176                 if evacuatedX+1 != evacuatedY || evacuatedX^1 != evacuatedY {
177                     throw("bad evacuatedN")
178                 }
179 
180                 b.tophash[i] = evacuatedX + useY // evacuatedX + 1 == evacuatedY
181                 dst := &xy[useY] // 迁移目的地
182 
```

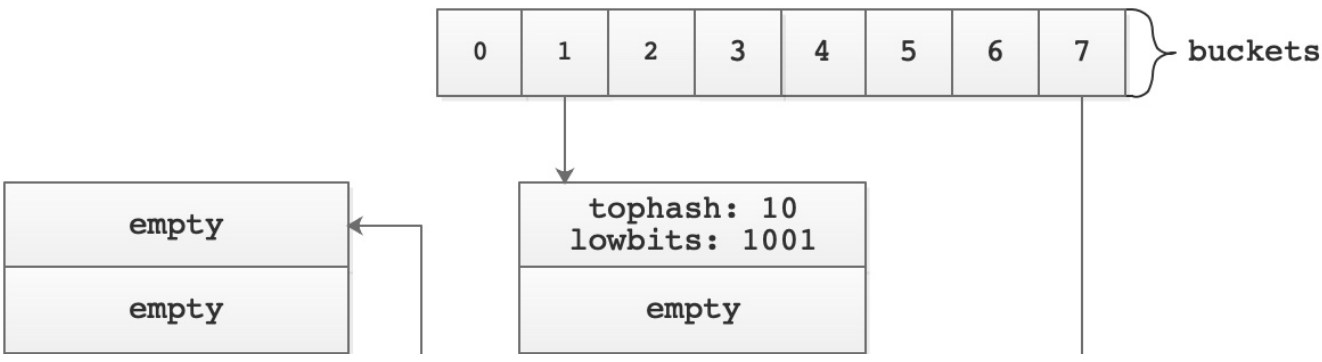
```
183                 if dst.i == bucketCnt {
184                     dst.b = h.newoverflow(t, dst.b)
185                     dst.i = 0
186                     dst.k = add(unsafe.Pointer(dst.b), dataOffset)
187                     dst.e = add(dst.k, bucketCnt*uintptr(t.keysize))
188                 }
189                 // 掩码dst.i作为优化，以避免边界检查
190                 dst.b.tophash[dst.i&(bucketCnt-1)] = top // mask dst.i as an optimization,
    to avoid a bounds check
191                 if t.indirectkey() {
192                     *(*unsafe.Pointer)(dst.k) = k2 // copy pointer // 拷贝指针
193                 } else {
194                     typedmemmove(t.key, dst.k, k) // copy elem // 拷贝元素
195                 }
196                 if t.indirectelem() {
197                     *(*unsafe.Pointer)(dst.e) = *(*unsafe.Pointer)(e)
198                 } else {
199                     typedmemmove(t.elem, dst.e, e)
200                 }
201                 dst.i++
202                 // These updates might push these pointers past the end of the
203                 // key or elem arrays.  That's ok, as we have the overflow pointer
204                 // at the end of the bucket to protect against pointing past the
205                 // end of the bucket.
206                 // 这些更新可能会将这些指针推到key或elem数组的末尾。
207                 // 没关系，因为我们在存储桶的末尾有溢出指针，以防止指向存储桶的末尾。
208                 dst.k = add(dst.k, uintptr(t.keysize))
209                 dst.e = add(dst.e, uintptr(t.elemsize))
210             }
211         }
212         // 取消链接溢出桶并清除key/elem，以帮助GC。
213         if h.flags&oldIterator == 0 && t.bucket.ptrdata != 0 {
214             b := add(h.oldbuckets, oldbucket*uintptr(t.bucketsize))
215             // 因为迁移状态一直保持在那里，所以要保留b.tophash。
216             ptr := add(b, dataOffset)
217             n := uintptr(t.bucketsize) - dataOffset
218             memclrHasPointers(ptr, n)
219         }
220     }
221
222     if oldbucket == h.nevacuate {
223         advanceEvacuationMark(h, t, newbit)
224     }
225 }
226
227 func advanceEvacuationMark(h *hmap, t *maptype, newbit uintptr) {
228     h.nevacuate++
229     // 实验表明，1024的杀伤力至少高出一个数量级。无论如何都要将其放在其中以确保O（1）行为。
230     stop := h.nevacuate + 1024
231     if stop > newbit {
232         stop = newbit
233     }
234     // 迁移直到不成功或者等于stop
235     for h.nevacuate != stop && bucketEvacuated(t, h, h.nevacuate) {
236         h.nevacuate++
237     }
```

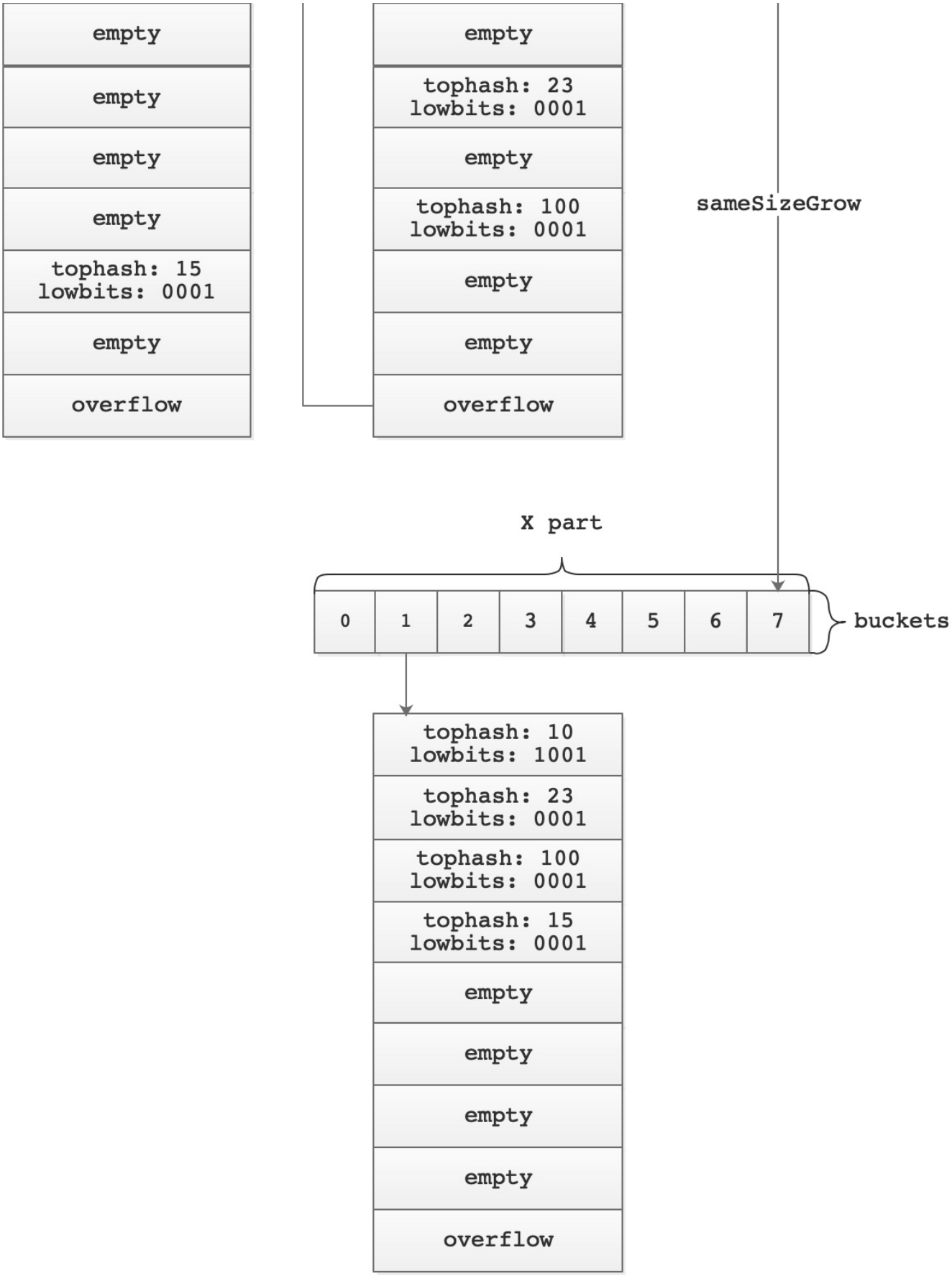
```
238     if h.nevacuate == newbit { // newbit == # of oldbuckets
239         // 增长已经完成。自由使用旧的主存储桶数组。
240         h.oldbuckets = nil
241         // 也可以丢弃旧的溢出桶。
242         // 如果迭代器仍在引用它们，则迭代器将保留指向切片的指针。
243         if h.extra != nil {
244             h.extra.oldoverflow = nil
245         }
246         h.flags &^= sameSizeGrow
247     }
248 }
```

biggerSizeGrow示意图：桶数组增大后，原来同一个桶的数据可以被分别移动到上半区和下半区。



sameSizeGrow示意图：sameSizeGrow 之后，数据排列更紧凑。





indirectkey和indirectvalue

indirectkey和indirectvalue在代码中经常出现，他们代表的是什么呢？indirectkey和indirectvalue在map里实际存储的是key和elem的指针。使用指针，在GC扫描时，会进行二次扫描操作，找出指针所代表的对象，所以扫描的对象更多。key/elem是indirect还是indirect是由编译器来决定的，依据是：

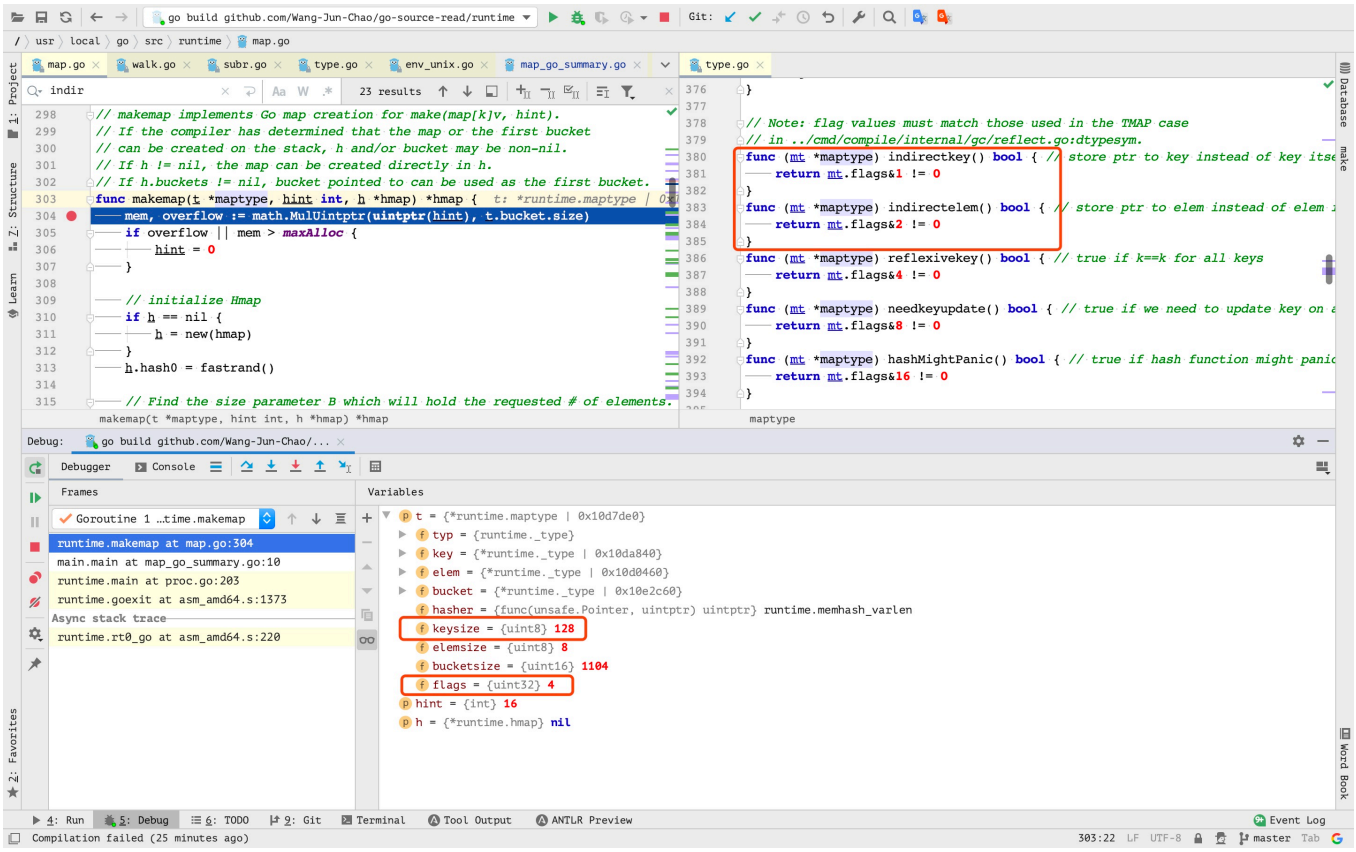
- key > 128 字节时，indirectkey = true
- value > 128 字节时，indirectvalue = true

下面使了两用用例来测试

```go
package main

import "fmt"

func main() {
    type P struct { // int在我的电脑上是8字节
        Age [16]int
    }

    var a = make(map[P]int, 16)

    for i := 0; i < 16; i++ {
        p := P{}
        p.Age[0] = i
        a[p] = i
    }
    fmt.Println(a)
}
```

maptype.flags各个位表示的含义：

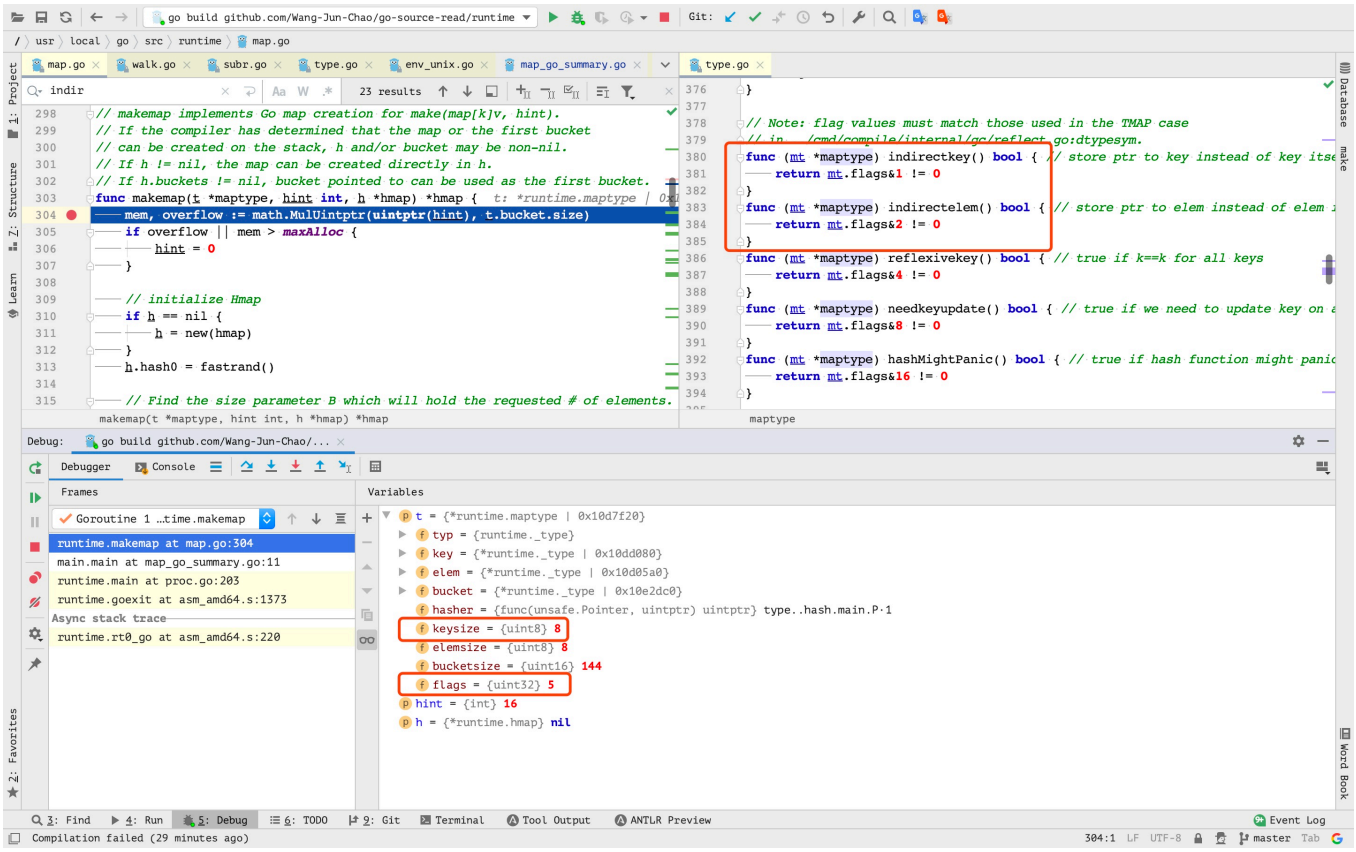
- 0b00000001：indirectkey，间接key
- 0b00000010：indirectelem，间接elem
- 0b00000100：reflexivekey，key是自反的，即：key==key总是为true，
- 0b00001000：needkeyupdate，需要更新key
- 0b00010000：hashMightPanic，key的hash函数可能有panic
  调式时可以看到
  t的flags值：4，说明是非indirectkey



```go
package main

import "fmt"

func main() {
    type P struct {
        Age [16]int

    }

    var a = make(map[P]int, 16)

    for i := 0; i < 16; i++ {
        p := P{}
        p.Age[0] = i
        a[p] = i
    }
    fmt.Println(a)
}
```

调式时可以看到

t的flags值：5，说明是indirectkey



overflow

overflow出现的场景：当有多个不同的key都hash到同一个桶的时候，桶的8个位置不够用，此时就会overflow。

获取overflow的方式，从 h.extra.nextOverflow中拿overflow桶，如果拿到，就放进 hmap.extra.overflow 数组，并让b的overflow指针指向这个桶。如果没找到，那就new一个新的桶。并且让b的overflow指针指向这个新桶，同时将新桶添加到h.extra.overflow数组中

```
/**
 * 创建新的溢出桶
 * @param
 * @return 新的溢出桶指针
 **/
func (h *hmap) newoverflow(t *maptype, b *bmap) *bmap {
    var ovf *bmap
    // 已经有额外数据，并且额外数据的nextOverflow不为空，
    if h.extra != nil && h.extra.nextOverflow != nil {
        // 我们有预分配的溢出桶可用。有关更多详细信息，请参见makeBucketArray。
        ovf = h.extra.nextOverflow
        if ovf.overflow(t) == nil {
            // 我们不在预分配的溢出桶的尽头。撞到指针。
            h.extra.nextOverflow = (*bmap)(add(unsafe.Pointer(ovf), uintptr(t.bucketsize)))
        } else {
            // 这是最后一个预分配的溢出存储桶。重置此存储桶上的溢出指针，该指针已设置为非nil标记值，现在要设置成nil
            ovf.setoverflow(t, nil)
            h.extra.nextOverflow = nil
        }
    } else {
        // 没有额外数据，创建新的溢出桶
        ovf = (*bmap)(newobject(t.bucket))
```

```
    }
    // 增加溢出桶计数
    h.incrnoverflow()
    if t.bucket.ptrdata == 0 { // 如果没有指针数据
        h.createOverflow() // 创建额外的溢出数据
        *h.extra.overflow = append(*h.extra.overflow, ovf) // 将溢出桶添加到溢出数组中
    }
    b.setoverflow(t, ovf)
    return ovf
}
/**
 * 创建h的溢出桶
 * @param
 * @return
 **/
func (h *hmap) createOverflow() {
    if h.extra == nil {
        h.extra = new(mapextra)
    }
    if h.extra.overflow == nil {
        h.extra.overflow = new([]*bmap)
    }
}
/**
 * incrnoverflow递增h.noverflow。
 * noverflow计算溢出桶的数量。
 * 这用于触发相同大小的map增长。
 * 另请参见tooManyOverflowBuckets。
 * 为了使hmap保持较小，noverflow是一个uint16。
 * 当存储桶很少时，noverflow是一个精确的计数。
 * 如果有很多存储桶，则noverflow是一个近似计数。
 * @param
 * @return
 **/
func (h *hmap) incrnoverflow() {
    // 如果溢出存储桶的数量与存储桶的数量相同，则会触发相同尺寸的map增长。
    // 我们需要能够计数到1<<h.B。
    if h.B < 16 { // 说是map中的元素比较少，少于（2^h.B）个
        h.noverflow++
        return
    }
    // 以概率1/(1<<(h.B-15))递增。
    //当我们达到1 << 15-1时，我们将拥有大约与桶一样多的溢出桶。
    mask := uint32(1)<<(h.B-15) - 1
    // 例如：如果h.B == 18，则mask == 7，fastrand&7 == 0，概率为1/8。
    if fastrand()&mask == 0 {
        h.noverflow++
    }
}
```

map方法的变种

mapaccess1、 mapaccess2、mapassign和mapdelete都有32位、64位和string 类型的变种，对对应的文件位置：

- $GOROOT/src/runtime/map_fast32.go

- $GOROOT/src/runtime/map_fast64.go
- $GOROOT/src/runtime/map_faststr.go

优缺点

go的map设计贴近底层，充分利用了内存布局。一般情况下元素的元素访问非常快。不足：go中的map使用拉链法解决hash冲突，当元素hash冲突比较多的时候会，需要经常扩容。map本身提供的方法比较少，不如其语言如java,c#丰富。

源码阅读

参考文档

https://github.com/cch123/golang-notes/blob/master/map.md
http://yangxikun.github.io/golang/2019/10/07/golang-map.html