Go底层探索(五):哈希表Map-扩容[下篇]

刘庆辉 猿码记 2023-04-11 19:01 Posted on 北京

收录于合集
#Go进阶 14 #Go 101

1. 介绍

随着哈希表中元素的逐渐增加，哈希的性能会逐渐恶化，所以我们需要更多的桶和更大的内存保证哈希的读写性能。

2. 怎么触发

在每次对哈希表**赋值**时，都会调用 `runtime.mapassign` 函数，该函数每次都会判断是否需要扩容，主要有两个函数： `overLoadFactory` 和 `tooManyOverflowBuckets` ：

```
// hash[k]=x表达式,会在编译期间转换成runtime.mapassign 函数的调用
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
 ...
 if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
    // 扩容入口
  hashGrow(t, h)
  goto again
 }
 ...
}
```

- `overLoadFactor` ： 主要判断装载因子是否超过 `6.5`
- `tooManyOverflowBuckets` ：用来判断是否使用了太多溢出桶；
- `h.growing()` ：用来判断是否已经处于扩容状态；

> 因为 Go 语言哈希的扩容不是一个原子的过程，所以 runtime.mapassign 还需要判断当前哈希是否已经处于扩容状态，避免二次扩容造成混乱。

3. 扩容方式

根据触发的条件不同扩容的方式分成两种：

- 第一种: 装载因子超过 6.5 ，则会进行双倍重建；
- 第二种: 当溢出桶的数量过多时，会进行等量重建；

3.1 等量扩容

当我们对 map 不断进行新增和删除时，桶中可能会出现很多断断续续的空位，这些空位会导致连接的 bmap 溢出桶很长，对应的扫描时间也会变长，查询性能就会下降。这种扩容实际上是一种整理，把后置位的数据整理到前面。

3.2 双倍重建

两倍重建是为了让 map 存储更多的数据,在双倍重建时，我们还需要解决旧桶中的数据要转移到某一个新桶中的问题。其中有一个非常重要的原则：如果数据的 hash&bucketMask [当前新桶所在的位置]小于或等于旧桶的大小，则此数据必须转移到和旧桶位置完全对应的新桶中去，理由是当前 key 所在新桶的序号与旧桶是完全相同的。

4. 扩容流程

4.1 扩容核心函数

扩容需要处理的问题是，扩容后， map 中原本的数据重新放到扩容后的 map 中，即数据迁移问题， golang 中 map 扩容时核心函数有如下几个

- hashGrow ： 决定扩容方式，负责初始化新的桶，以及设置扩容后的 map 的各个字段

- growWork ： 每调用一次 growWork 函数，都至多会迁移两个桶的数据
- evacuate ： 真正负责迁移数据的函数，会负责迁移指定桶中的数据
- advanceEvacuationMark ： 收尾工作，增加 nevacuate ，如果所有的 oldbuckets 都迁移完成了，会摘除 oldbuckets

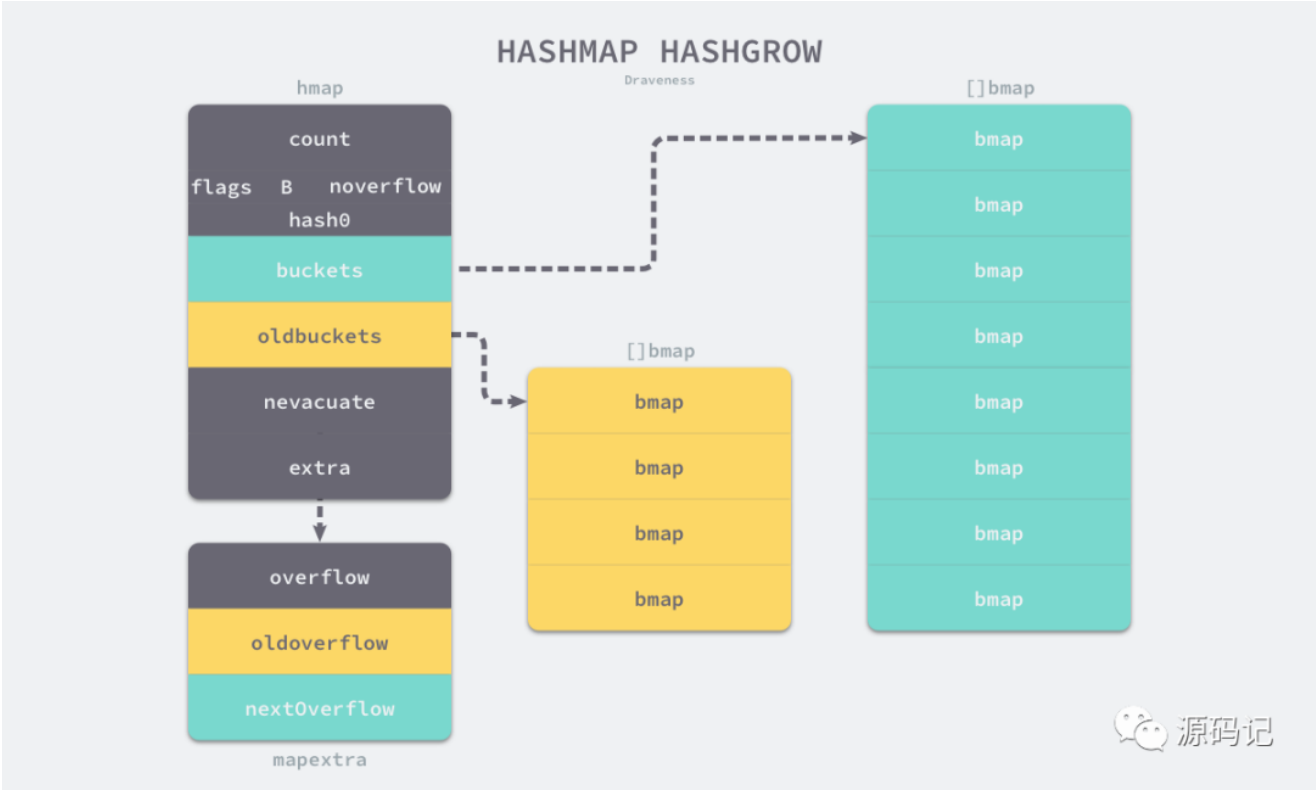
4.2 hashGrow

重建时需要调用 hashGrow 函数，如果负载因子超载，则会进行双倍重建。当溢出桶的数量过多时，会进行等量重建。新桶会存储到 buckets 字段，旧桶会存储到 oldbuckets 字段。 map 中 extra 字段的溢出桶也进行同样的转移。

```
func hashGrow(t *maptype, h *hmap) {
 bigger := uint8(1)
 if !overLoadFactor(h.count+1, h.B) {
  bigger = 0
  h.flags |= sameSizeGrow
 }
  // 旧数据存到旧桶
 oldbuckets := h.buckets
  // 创建一组新桶和溢出桶
 newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)

 h.B += bigger
 h.flags = flags
  // 旧数据存到旧桶上
 h.oldbuckets = oldbuckets
 h.buckets = newbuckets
 h.nevacuate = 0
 h.noverflow = 0
  // 原有的溢出桶，存到旧溢出桶
 h.extra.oldoverflow = h.extra.overflow
 h.extra.overflow = nil
 h.extra.nextOverflow = nextOverflow
}
```

> @注意：这里并没有实际执行将旧桶中的数据转移到新桶的过程。数据转移遵循写时复制（copy on write）的规则，只有在真正赋值时，才会选择是否需要进行数据转移，其核心逻辑位于growWork和evacuate函数中。



扩容后的map的各个字段

4.3 growWork

growWork 函数并不会真正进行数据迁移，它会调用 evacuate 函数来完成迁移工作， growWork 函数每次会迁移至多两个桶的数据，一个是目前需要使用的桶，一个是 h.nevacuate 桶（这里很重要，在后面判断是否迁移过程中有很大的作用）， h.nevacuate 记录的是目前至少已经迁移的桶的个数。

```
func growWork(t *maptype, h *hmap, bucket uintptr) {
 // make sure we evacuate the oldbucket corresponding
 // to the bucket we're about to use
 evacuate(t, h, bucket&h.oldbucketmask())

 // evacuate one more oldbucket to make progress on growing
 if h.growing() {
```

```
  evacuate(t, h, h.nevacuate)
 }
}
```

4.4 evacuate

`evacuate` 是真正进行数据迁移的函数，它每次会迁移一个 `bmap` 中的数据，简单说，就是遍历旧有 `buckets` 中 `bmap` 中的数据，将其放到新 `bmap` 的对应位置;

在学习 `evacuate` 函数前，先记 `bmap.tophash` 的几个特殊值，在扩容过程中会使用到:

```
emptyRest      = 0 // 表明该位置及其以后的位置都没有数据
emptyOne       = 1 // 表明该位置没有数据
evacuatedX     = 2 // key/elem是有效的，它已经在扩容过程中被迁移到了更大表的前半部分
evacuatedY     = 3 // key/elem是有效的，它已经在扩容过程中被迁移到了更大表的后半部分
evacuatedEmpty = 4 // 该位置没有数据，且已被扩容
minTopHash     = 5 // 一个被正常填充的tophash的最小值
```

4.4.1 判断桶的迁移状态

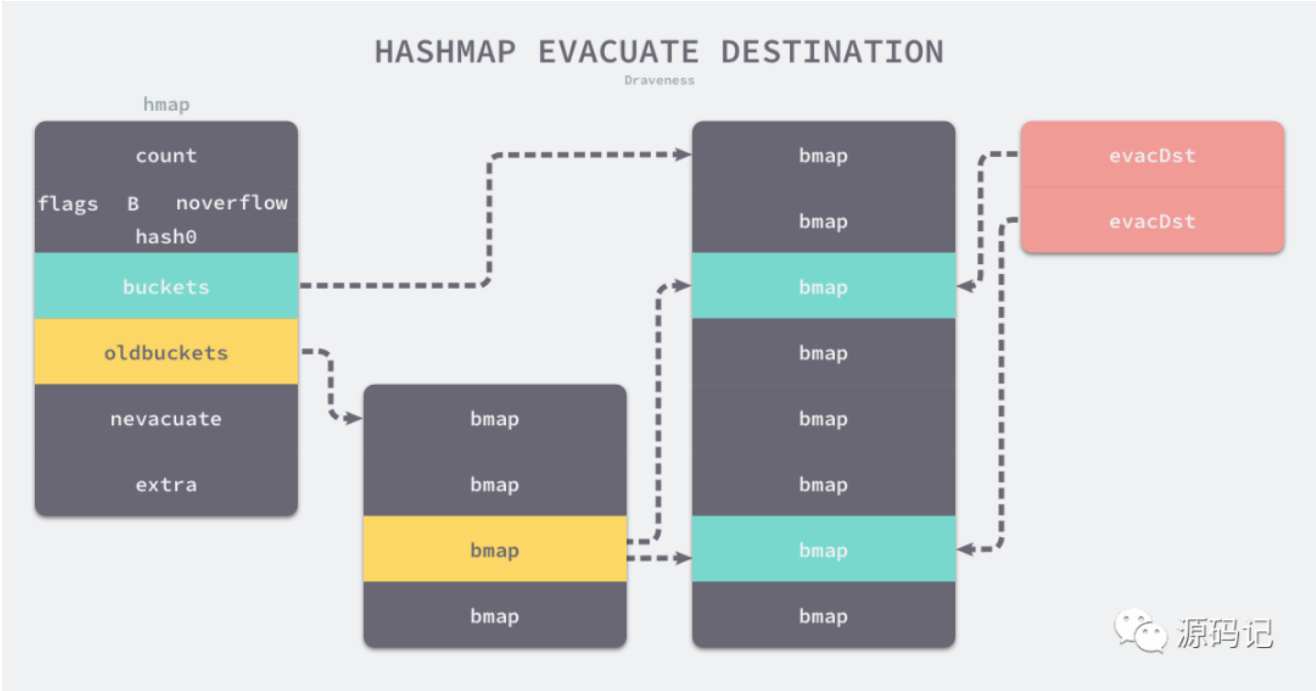
首先会判断这个桶是否已经被迁移过了，或者正在迁移中，如果没有被迁移，才会进行迁移工作。该判断是通过 `evacuated` 函数完成的，该函数很简单，只需要判断 `tophash[0]` 是否是 `evacuatedX, evacuatedY, evacuateEmpty` 即可.

```
func evacuated(b *bmap) bool {
 h := b.tophash[0]
 return h > emptyOne && h < minTopHash
}
```

4.4.2 初始化evacDst结构

初始化 `evacDst` 结构，如果是等量扩容，则只会初始化一个，如果是普通扩容，则会初始化两个。 `runtime.evacuate` 会将一个旧桶中的数据分流到两个新桶，所以它会创建两个用于保

存分配上下文的 `runtime.evacDst` 结构体，这两个结构体分别指向了一个新桶：



hashmap-evacuate-destination

`evacDst` 结构体如下所示：

```
type evacDst struct {
    b *bmap          // 目标桶
    i int            // 每个桶有8个位置可以塞数据，这个i可以理解为当前塞了多少个数据
    k unsafe.Pointer // 目标桶中key的位置
    e unsafe.Pointer // 目标桶中elem的位置
}
```

4.4.3 旧桶元素分流

如果这是等量扩容，那么旧桶与新桶之间是一对一的关系，所以两个 `runtime.evacDst` 只会初始化一个。而当哈希表的容量翻倍时，每个旧桶的元素会都分流到新创建的两个桶中，这里仔细分析一下分流元素的逻辑：

```
...
    // 遍历 oldbuckets 对应的 bucket 以及 oveflow
  for ; b != nil; b = b.overflow(t) {
```

```
   //获取当前 bucket 的 key 的起始位置
k := add(unsafe.Pointer(b), dataOffset)
   //获取当前 bucket 的 elem 的起始位置
e := add(k, bucketCnt*uintptr(t.keysize))
   //遍历当前 bucket 中 8 个 key, elem
for i := 0; i < bucketCnt; i, k, e = i+1, add(k, uintptr(t.keysize)), add(e, uintptr(t.elemsiz
 top := b.tophash[i]
     // 如果为空，则跳过
 if isEmpty(top) {
  b.tophash[i] = evacuatedEmpty
  continue
 }
     //如果小于 minTopHash ，则表示其已经被转移走了，则 throw
 if top < minTopHash {
  throw("bad map state")
 }
 k2 := k
     //如果存的是对应 key 的指针，则要获取 key 的地址
 if t.indirectkey() {
  k2 = *((*unsafe.Pointer)(k2))
 }
 var useY uint8
     // 如果是非等量迁移(双倍重建)
 if !h.sameSizeGrow() {
       //算出当前 key 的 hash 值
  hash := t.hasher(k2, uintptr(h.hash0))
  if h.flags&iterator != 0 && !t.reflexivekey() && !t.key.equal(k2, k2) {
   useY = top & 1 //让这个 key 50% 概率去 Y 半区
   top = tophash(hash)
  } else {
   if hash&newbit != 0 {
    useY = 1
   }
  }
 }

 if evacuatedX+1 != evacuatedY || evacuatedX^1 != evacuatedY {
  throw("bad evacuatedN")
 }
```

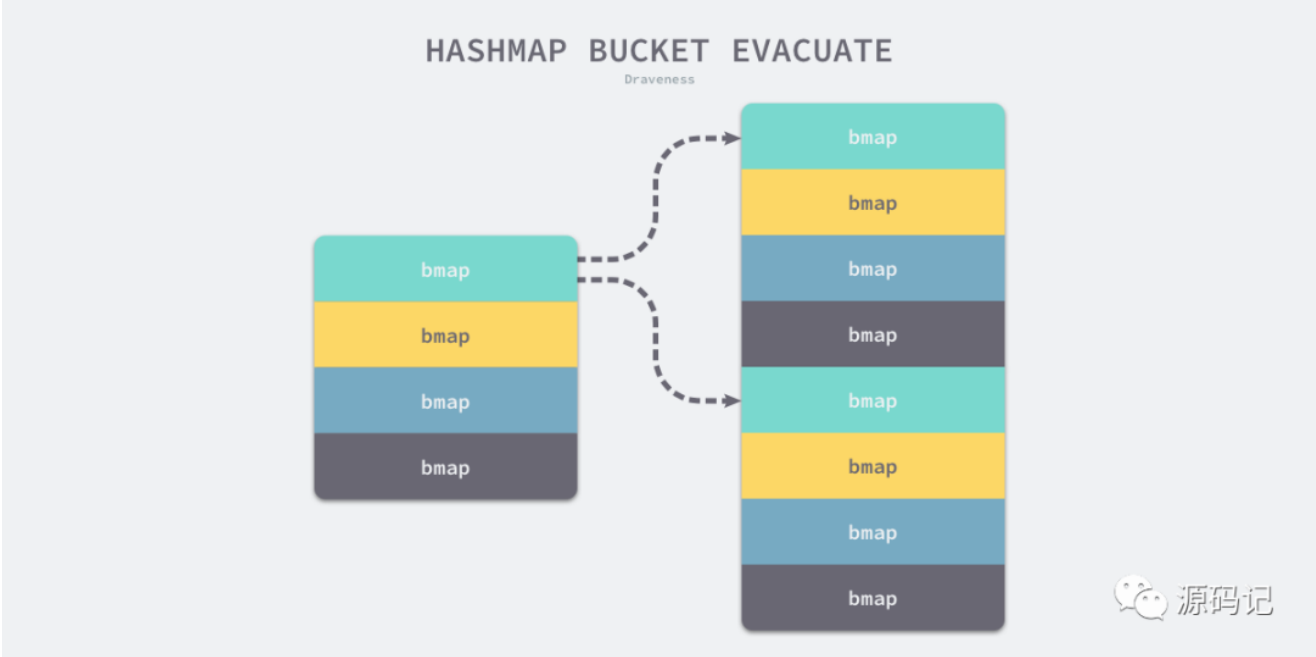
```
    b.tophash[i] = evacuatedX + useY // evacuatedX + 1 == evacuatedY
    dst := &xy[useY]                 // 移动目标

    if dst.i == bucketCnt {
     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 // mask dst.i as an optimization, to avoid a bounds c
    if t.indirectkey() {
     *(*unsafe.Pointer)(dst.k) = k2 // copy pointer
    } else {
     typedmemmove(t.key, dst.k, k) // copy elem
    }
    if t.indirectelem() {
     *(*unsafe.Pointer)(dst.e) = *(*unsafe.Pointer)(e)
    } else {
     typedmemmove(t.elem, dst.e, e)
    }
    dst.i++
    // These updates might push these pointers past the end of the
    // key or elem arrays.  That's ok, as we have the overflow pointer
    // at the end of the bucket to protect against pointing past the
    // end of the bucket.
    dst.k = add(dst.k, uintptr(t.keysize))
    dst.e = add(dst.e, uintptr(t.elemsize))
   }
  }
...
```

只使用哈希函数是不能定位到具体某一个桶的，哈希函数只会返回很长的哈希，例如：`b72bfae3f3285244c4732ce457cca823bc189e0b`，我们还需一些方法将哈希映射到具体的桶上。我们一般都会使用取模或者位操作来获取桶的编号，假如当前哈希中包含 `4` 个桶，那么它的桶掩码就是 `0b11(3)`，使用位操作就会得到 `3`，我们就会在 `3` 号桶中存储该数据：

```
0xb72bfae3f3285244c4732ce457cca823bc189e0b & 0b11 #=> 0
```

如果新的哈希表有 8 个桶，在大多数情况下，原来经过桶掩码 0b11 结果为 3 的数据会因为桶掩码增加了一位变成 0b111 而分流到新的 3 号和 7 号桶，所有数据也都会被 runtime.typedmemmove 拷贝到目标桶中：



4.5 advanceEvacuationMark

runtime.evacuate 最后会调用 runtime.advanceEvacuationMark 增加哈希的 nevacuate 计数器并在所有的旧桶都被分流后清空哈希的 oldbuckets 和 oldoverflow：

```
func advanceEvacuationMark(h *hmap, t *maptype, newbit uintptr) {
    h.nevacuate++
    // Experiments suggest that 1024 is overkill by at least an order of magnitude.
    // Put it in there as a safeguard anyway, to ensure O(1) behavior.
    stop := h.nevacuate + 1024
    if stop > newbit {
        stop = newbit
    }
    for h.nevacuate != stop && bucketEvacuated(t, h, h.nevacuate) {
        h.nevacuate++
    }
    if h.nevacuate == newbit { // newbit == # of oldbuckets
        // 大小增长全部结束。释放老的 bucket array
        h.oldbuckets = nil
```

```
        // 同样可以丢弃老的 overflow buckets
        // 如果它们还被迭代器所引用的话
        // 迭代器会持有一份指向 slice 的指针
        if h.extra != nil {
            h.extra.oldoverflow = nil
        }
        h.flags &^= sameSizeGrow
    }
}
```


收录于合集 #Go 101

上一篇

Go底层探索(四):哈希表Map[上篇]

下一篇

Go底层探索(六):延迟函数defer

Read more

People who liked this content also liked

Python常用库(二):数学计算

猿码记