Go底层探索(四):哈希表Map[上篇]

猿码记 2023-03-15 18:00 Posted on 北京

收录于合集
#Go进阶 14 #Go 101

> **@注: 以下内容来自【Go语言底层原理剖析、Go 语言设计与实现】两书中的摘要信息，一起读书、一起学习。**

1.介绍

`Go` 语言中的 `map` 又被称为哈希表，是使用频率极高的一种数据结构。哈希表的原理是将多个 `键/值 (key/value)` 对,分散存储在 `buckets` （桶）中。

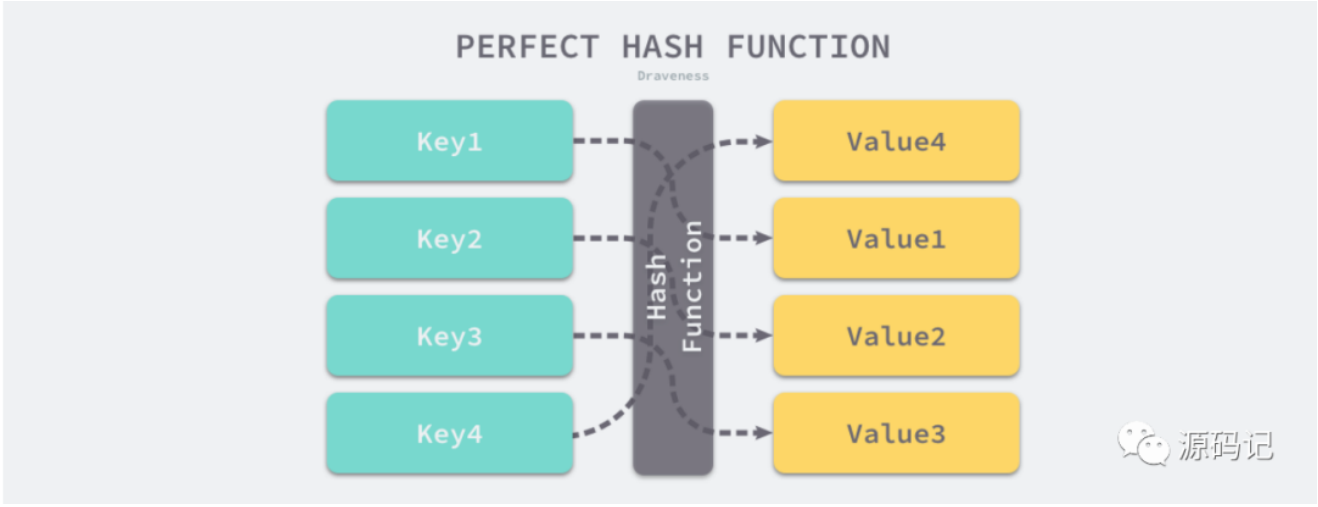
通常 `map` 的时间复杂度为 `O(1)` ,通过一个键快速寻找其唯一对应的值 `value` 。在许多情况下，哈希表的查找速度明显快于一些搜索树形式的数据结构，被广泛用于关联数组、缓存、数据库缓存等场景中。

> 如果想要实现一个性能优异的哈希表，需要注意两个关键点 —— **哈希函数和冲突解决方法。**

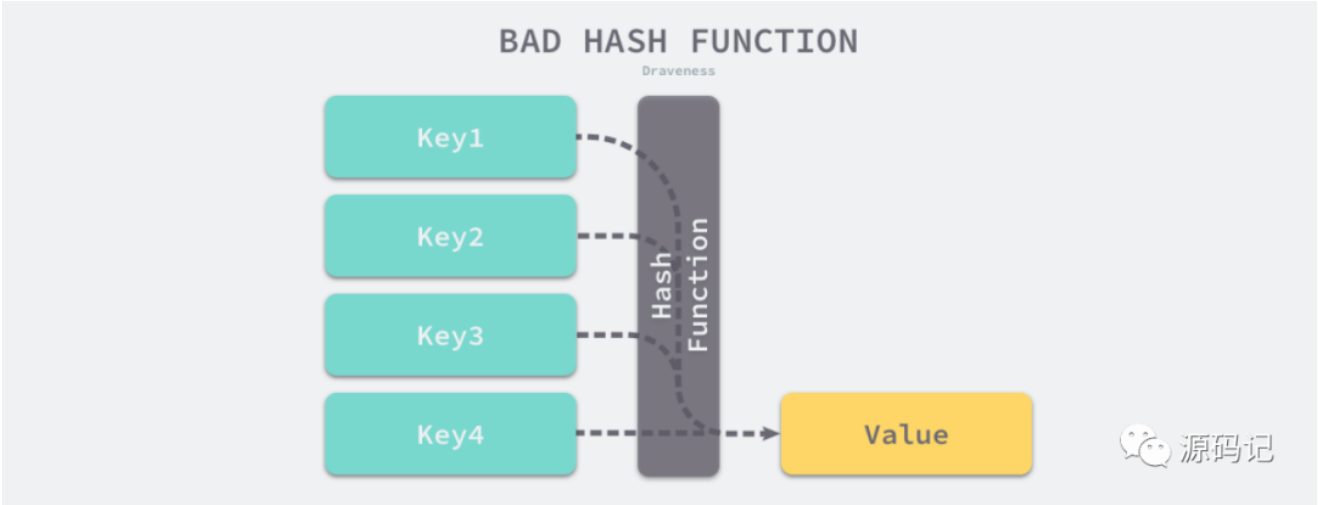
2. 哈希函数与碰撞

实现哈希表的关键点在于哈希函数的选择，哈希函数的选择在很大程度上能够决定哈希表的读写性能。在理想情况下，哈希函数应该能够将不同键映射到不同的索引上，但是在实际使用时，这个理想的效果是不可能实现的。

下面展示两个图，分别示意完美哈希函数和不均匀哈希函数计算的结果图:



完美哈希函数计算值



不均匀哈希函数计算值

不同的键通过哈希函数产生相同的哈希值,则就是我们常听到的哈希碰撞(哈希冲突)。

> 如果将 2450 个键随机分配到一百万个桶中，则根据概率计算，至少有两个键被分配到同一个桶中的可能性有惊人的 95% 。

哈希碰撞导致同一个桶中可能存在多个元素，有多种方式可以避免哈希碰撞，一般有两种主要的策略：拉链法和开放寻址法。

3. 开放寻址法

3.1 设计原理

概念: 开放寻址法是一种在哈希表中解决哈希碰撞的方法，这种方法的核心思想是: 依次探测和比较数组中的元素以判断目标键值对是否存在于哈希表中。

优点: 由于存储的地址连续，可以更好的利用 `CPU` 高速缓存。

缺点: 当散列表中插入的数据越来越多时，散列冲突发生的可能性就会越来越大，空闲位置会越来越少，线性探测的时间就会越来越久。极端情况下，需要从头到尾探测整个散列表，所以最坏情况下的时间复杂度为 `O(n)`

@注:Go语言中的哈希表采用的是开放寻址法中的线性探测（Linear Probing）策略

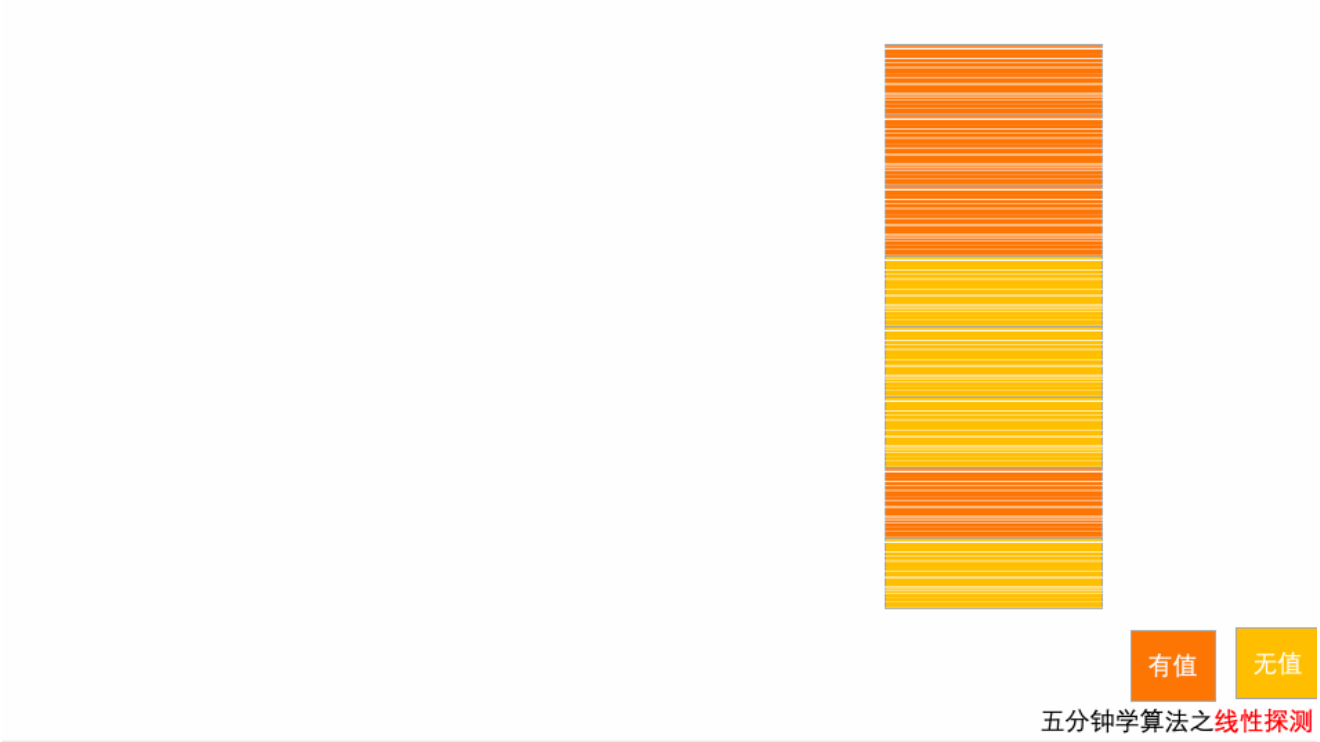
> 开放寻址法中对性能影响最大的是**装载因子**，它是数组中元素的数量与数组大小的比值。随着装载因子的增加，线性探测的平均用时就会逐渐增加，这会影响哈希表的读写性能。当装载率超过 70% 之后，哈希表的性能就会急剧下降，而一旦装载率达到 100%，整个哈希表就会完全失效，这时查找和插入任意元素的时间复杂度都是 $O(n)$ ，这时需要遍历数组中的全部元素，所以在实现哈希表时一定要关注装载因子的变化。

装载因子:

在一个性能比较好的哈希表中，每一个桶中都应该有 `0~1` 个元素，有时会有 `2~3` 个，很少会超过这个数量。计算哈希、定位桶和遍历链表三个过程是哈希表读写操作的主要开销，装载因子公式：

$$装载因子=元素数量 / 桶数量$$

3.2 写入动画



写入流程动画演示-先行探测

以上图为例，散列表的大小为 `8` ，黄色区域表示空闲位置，橙色区域表示已经存储了数据。目前散列表中已经存储了 `4` 个元素。此时元素 `7777777` 经过 `Hash` 算法之后，被散列到位置下标为 `7` 的位置，但是这个位置已经有数据了，所以就产生了冲突。于是按顺序地往后一个一个找，看有没有空闲的位置，此时，运气很好正巧在下一个位置就有空闲位置，将其插入，完成了数据存储。

4. 拉链法

4.1 设计原理

概念: 将同一个桶中的多个元素通过链表的形式进行链接，这是一种最简单、最常用的策略。

优点: 随着桶中元素的增加，可以不断链接新的元素，同时不用预先为元素分配内存。

缺点: 需要存储额外的指针用于链接元素，这增加了整个哈希表的大小。同时由于链表存储的地址不连续，所以无法高效利用CPU高速缓存。

4.2 写入动画

五分钟学算法之链表法

已上图为例，还是插入元素 `7777777` ,过 `Hash` 算法之后，被散列到位置下标为 `7` 的位置,但是这个位置已经有数据了，所以就产生了冲突，然后通过链表的形式，挂载在后面。

5. 并发冲突

和其他语言不同的是， `map` 并不支持并发的读写， `map` 并发读写是初级开发者经常会犯的错误。下面的示例由于协程并发读写会报错为 `fatal error: concurrent map read and map write` 。

5.1 错误示例

```
func TestRun(t *testing.T) {
 mp := map[string]int{
  "a": 10,
  "b": 5,
 }
 // 开启读协程
```

```
    go func() {
     for true {
      fmt.Println("a = ", mp["a"])
     }
    }()
    // 开启写协程
    go func() {
     for true {
      mp["b"] = 10
     }
    }()
    time.Sleep(time.Second * 3)
    fmt.Println("ok")
}
```

5.2 不支持并发读写原因

Go 语言为什么不支持并发的读写，是一个频繁被提起的问题。我们可以在 Go 官方文档中找到问题的答案。

官方文档的解释是："map不需要从多个Goroutine安全访问，在实际情况下，map可能是某些已经同步的较大数据结构或计算的一部分。因此，要求所有map操作都互斥将减慢大多数程序的速度，而只会增加少数程序的安全性。" 即 Go 语言只支持并发读写的原因是保证大多数场景下的查找效率。

6. 数据结构

6.1 核心结构体

Go 语言运行时同时使用了多个数据结构组合表示哈希表，其中 `runtime.hmap` 是最核心的结构体，我们先来了解一下该结构体的内部字段：

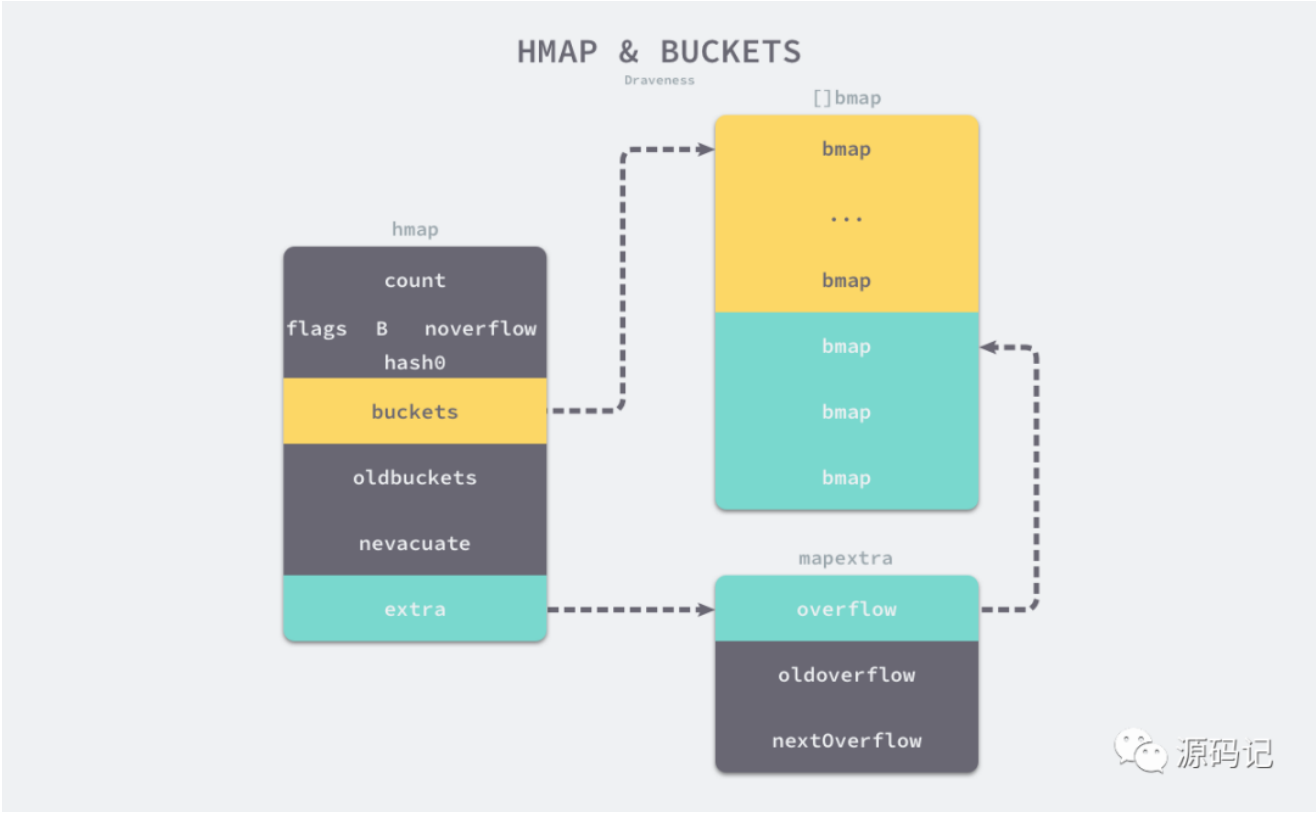
```
type hmap struct {
 count     int
 flags     uint8
```

```
 B         uint8
 noverflow uint16
 hash0     uint32
 buckets    unsafe.Pointer
 oldbuckets unsafe.Pointer
 nevacuate  uintptr
 extra *mapextra
}

type mapextra struct {
 overflow    *[]*bmap
 oldoverflow *[]*bmap
 nextOverflow *bmap
}
```

- `count`：表示当前哈希表中的元素数量；
- `flags`：代表当前 `map` 的状态（是否处于正在写入的状态等）；
- `B`：表示当前哈希表持有的 `buckets` 数量，但是因为哈希表中桶的数量都 2 的倍数，所以该字段会存储对数，也就是 `len(buckets) == 2^B`
- `noverflow`：代表当前 `map` 中溢出桶的数量。当溢出的桶太多时， `map` 会进行扩容，其实质是避免溢出桶过大导致内存泄露。
- `hash0`：是哈希的种子，它能为哈希函数的结果引入随机性，这个值在创建哈希表时确定，并在调用哈希函数时作为参数传入；
- `buckets`： 指向当前 `map` 对应的桶的指针;
- `oldbuckets`： 哈希在扩容时用于保存之前 `buckets` 的字段，它的大小是当前 `buckets` 的一半,当所有旧桶中的数据都已经转移到了新桶中时，则清空。
- `nevacuate`：在扩容时使用，用于标记当前旧桶中小于 `nevacuate` 的数据都已经转移到了新桶中。
- `extra`：存储 `map` 中的溢出桶。

6.2 结构示意图



如上图所示哈希表 `runtime.hmap` 的桶是 `runtime.bmap` 。每一个 `runtime.bmap` 都能存储 8 个键值对，当哈希表中存储的数据过多，单个桶已经装满时就会使用 `extra.nextOverflow` 中桶存储溢出的数据。

> 上述两种不同的桶在内存中是连续存储的，我们在这里将它们分别称为正常桶和溢出桶，上图中黄色的 `runtime.bmap` 就是正常桶，绿色的 `runtime.bmap` 是溢出桶，溢出桶是在 Go 语言还使用 C 语言实现时使用的设计，由于它能够减少扩容的频率所以一直使用至今。

6.3 桶结构(bmap)

代表桶的 `bmap` 结构在运行时只列出了首个字段，即一个固定长度为 `8` 的数组。此字段顺序存储 `key` 的哈希值的前8位。

```
type bmap struct {
 tophash [bucketCnt]uint8
}
```

1.桶中存储的key和value到哪里去了?

在运行期间， `runtime.bmap` 结构体其实不止包含 `tophash` 字段，因为哈希表中可能存储不同类型的键值对，而且 `Go` 语言也不支持泛型(当时不支持)，所以键值对占据的内存空间大小只能在编译时进行推导。 `runtime.bmap` 中的其他字段在运行时也都是通过计算内存地址的方式访问的，所以它的定义中就不包含这些字段，不过我们能根据编译期间的 `cmd/compile/internal/gc.bmap` 函数重建它的结构：

```
type bmap struct {
    topbits  [8]uint8
    keys     [8]keytype
    values   [8]valuetype
    pad      uintptr
    overflow uintptr
}
```

> 随着哈希表存储的数据逐渐增多，我们会扩容哈希表或者使用额外的桶存储溢出的数据，不会让单个桶中的数据超过 `8` 个，不过溢出桶只是临时的解决方案，创建过多的溢出桶最终也会导致哈希的扩容。

7. 初始化

7.1 字面量初始化

目前的现代编程语言基本都支持使用字面量的方式初始化哈希，一般都会使用 `key: value` 的语法来表示键值对， `Go` 语言中也不例外：

```
hash := map[string]int{
 "1": 2,
 "3": 4,
 "5": 6,
}
```

我们需要在初始化哈希时声明键值对的类型，这种使用字面量初始化的方式最终都会通过 `cmd/compile/internal/gc.maplit` 初始化，我们来分析一下该函数初始化哈希的过程：

```
func maplit(n *Node, m *Node, init *Nodes) {
 a := nod(OMAKE, nil, nil)
 a.Esc = n.Esc
 a.List.Set2(typenod(n.Type), nodintconst(int64(n.List.Len())))
 litas(m, a, init)

 entries := n.List.Slice()
 if len(entries) > 25 {
  ...
  return
 }

 // Build list of var[c] = expr.
 // Use temporaries so that mapassign1 can have addressable key, elem.
 ...
}
```

1.元素数量小于等于25时:

当哈希表中的元素数量 `<=25` 个时，编译器会将字面量初始化的结构体转换成以下的代码，将所有的键值对一次加入到哈希表中：

```
hash := make(map[string]int, 3)
hash["1"] = 2
hash["3"] = 4
hash["5"] = 6
```

> 这种初始化的方式与的数组和切片几乎完全相同，由此看来集合类型的初始化在 Go 语言中有着相同的处理逻辑。

2.元素数量大于25时:

一旦哈希表中元素的数量超过了 25 个，编译器会创建两个数组分别存储键和值，这些键值对会通过如下所示的 for 循环加入哈希：

```
hash := make(map[string]int, 26)
vstatk := []string{"1", "2", "3", ... ,  "26"}
vstatv := []int{1, 2, 3, ... , 26}
for i := 0; i < len(vstak); i++ {
    hash[vstatk[i]] = vstatv[i]
}
```

> 使用字面量初始化的过程都会使用 `Go` 语言中的关键字 `make` 来创建新的哈希并通过最原始的 `[]` 语法向哈希追加元素.

7.2 运行时

当创建的哈希被分配到栈上并且其容量小于 `BUCKETSIZE = 8` 时， `Go` 语言在编译阶段会使用如下方式快速初始化哈希，这也是编译器对小容量的哈希做的优化：

```
var h *hmap
var hv hmap
var bv bmap
h := &hv
b := &bv
h.buckets = b
h.hash0 = fashtrand0()
```

除了上述特定的优化之外，无论 `make` 是从哪里来的，只要我们使用 `make` 创建哈希， `Go` 语言编译器都会在类型检查期间将它们转换成 `runtime.makemap` ，使用字面量初始化哈希也只是语言提供的辅助工具，最后调用的都是 `runtime.makemap` ：

```
func makemap(t *maptype, hint int, h *hmap) *hmap {
 mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
```

```
 if overflow || mem > maxAlloc {
  hint = 0
 }

 if h == nil {
  h = new(hmap)
 }
 h.hash0 = fastrand()

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

 if h.B != 0 {
  var nextOverflow *bmap
  h.buckets, nextOverflow = makeBucketArray(t, h.B, nil)
  if nextOverflow != nil {
   h.extra = new(mapextra)
   h.extra.nextOverflow = nextOverflow
  }
 }
 return h
}
```

这个函数会按照下面的步骤执行：

1. 计算哈希占用的内存是否溢出或者超出能分配的最大值；
2. 调用 `runtime.fastrand` 获取一个随机的哈希种子；
3. 根据传入的 `hint` 计算出需要的最小需要的桶的数量；
4. 使用 `runtime.makeBucketArray` 创建用于保存桶的数组；

8. 读写原理

8.1 读取

在编译的类型检查期间， `hash[key]` 以及类似的操作都会被转换成哈希的 `OINDEXMAP` 操作，中间代码生成阶段会在 `cmd/compile/internal/gc.walkexpr` 函数中将这些 `OINDEXMAP` 操作转换成如下的代码：

```
v     := hash[key] // => v     := *mapaccess1(maptype, hash, &key)
v, ok := hash[key] // => v, ok := mapaccess2(maptype, hash, &key)
```

赋值语句左侧接受参数的个数会决定使用的运行时方法：

- 当接受一个参数时，会使用 `runtime.mapaccess1` ，该函数仅会返回一个指向目标值的指针；
- 当接受两个参数时，会使用 `runtime.mapaccess2` ，除了返回目标值之外，它还会返回一个用于表示当前键对应的值是否存在的 `bool` 值；

> `runtime.mapaccess1` 会先通过哈希表设置的哈希函数、种子获取当前键对应的哈希，再通过 `runtime.bucketMask` 和 `runtime.add` 拿到该键值对所在的桶序号和哈希高位的 `8` 位数字，

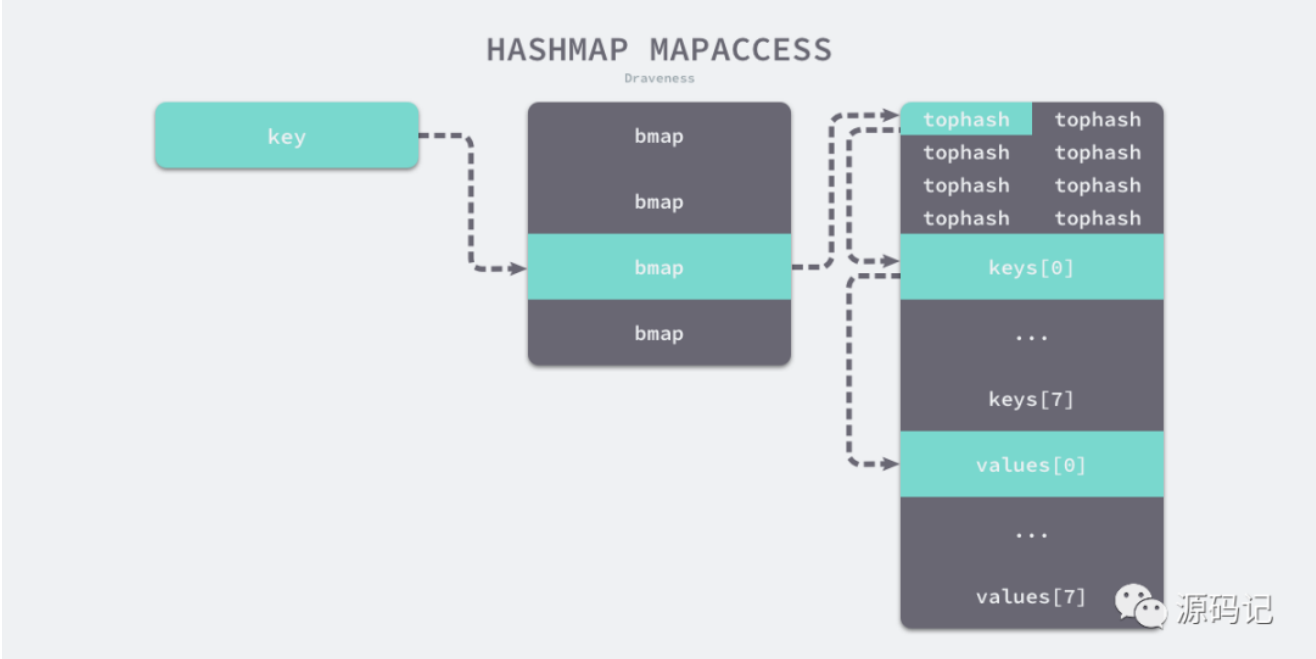
`runtime.mapaccess1源码：`

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
 alg := t.key.alg
  // 获取当前键对应的哈希
 hash := alg.hash(key, uintptr(h.hash0))
 m := bucketMask(h.B)
  // hash&m计算出key应该位于哪一个桶中
 b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
  // tophash（hash）计算出hash的前8位
 top := tophash(hash)
bucketloop:
 for ; b != nil; b = b.overflow(t) {
  for i := uintptr(0); i < bucketCnt; i++ {
      // 与存储在桶中的tophash进行对比
   if b.tophash[i] != top {
```

```
    if b.tophash[i] == emptyRest {
     break bucketloop
    }
    continue
   }
   k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
   if alg.equal(key, k) {
    v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
    return v
   }
  }
 }
 return unsafe.Pointer(&zeroVal[0])
}
```

在 `bucketloop` 循环中，哈希会依次遍历正常桶和溢出桶中的数据，它会先比较哈希的高 `8` 位和桶中存储的 `tophash` ， `Go` 语言采用了一种简单的方式 `hash&m` 计算出 `key` 应该位于哪一个桶中。获取桶的位置后， `tophash (hash)` 计算出 `hash` 的前 `8` 位。接着此 `hash` 挨个与存储在桶中的 `tophash` 进行对比。如果有 `hash` 值相同，则会找到此 `hash` 值对应的 `key` 值并判断是否相同。如果 `key` 值也相同，则说明查找到了结果，返回 `value` 。

看图加深理解:



访问哈希表中的数据

如上图所示，每一个桶都是一整片的内存空间，当发现桶中的 `tophash` 与传入键的 `tophash` 匹配之后，我们会通过指针和偏移量获取哈希中存储的键 `keys[0]` 并与 `key` 比较，如果两者相同就会获取目标值的指针 `values[0]` 并返回。

8.2 写入

当形如 `hash[k]` 的表达式出现在赋值符号左侧时，该表达式也会在编译期间转换成 `runtime.mapassign` 函数的调用，该函数与 `runtime.mapaccess1` 比较相似，我们将其分成几个部分依次分析，首先是函数会根据传入的键拿到对应的哈希和桶：

src/runtime/map.go:578

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
 alg := t.key.alg
  // 先计算key的hash值
 hash := alg.hash(key, uintptr(h.hash0))
  // 标记当前map是写入状态
 h.flags ^= hashWriting

again:
  // 计算对应的桶
 bucket := hash & bucketMask(h.B)
 b := (*bmap)(unsafe.Pointer(uintptr(h.buckets) + bucket*uintptr(t.bucketsize)))
 top := tophash(hash)
```

然后通过遍历比较桶中存储的 `tophash` 和键的哈希，如果找到了相同结果就会返回目标位置的地址，获得目标地址之后会通过算术计算寻址获得键值对 `k` 和 `val`：

src/runtime/map.go:619

```
 //inserti 表示目标元素的在桶中的索引
  var inserti *uint8
  // insertk 和 val`分别表示键值对的地址
 var insertk unsafe.Pointer
 var val unsafe.Pointer
bucketloop:
```

```
for {
 for i := uintptr(0); i < bucketCnt; i++ {
     // 判断 tophash 是否相等
  if b.tophash[i] != top {
   if isEmpty(b.tophash[i]) && inserti == nil {
         // 插入数据inserti, insertk会记录此空元素的位置
    inserti = &b.tophash[i]
    insertk = add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
    val = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
   }
   if b.tophash[i] == emptyRest {
    break bucketloop
   }
   continue
  }
  k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
     // key 是否相等
  if !alg.equal(key, k) {
   continue
  }
  val = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
  goto done
 }
 ovf := b.overflow(t)
 if ovf == nil {
  break
 }
   // 溢出桶变量赋值给b,开始遍历溢出桶
 b = ovf
}
```

桶满后存到溢出桶:

如果当前桶已经满了，哈希会调用 `runtime.hmap.newoverflow` 创建新桶或者使用 `runtime.hmap` 预先在 `noverflow` 中创建好的桶来保存数据，新创建的桶不仅会被追加到已有桶的末尾，还会增加哈希表的 `noverflow` 计数器。

src/runtime/map.go:662

```
 if inserti == nil {
  newb := h.newoverflow(t, b)
  inserti = &newb.tophash[0]
  insertk = add(unsafe.Pointer(newb), dataOffset)
  val = add(insertk, bucketCnt*uintptr(t.keysize))
 }

 typedmemmove(t.key, insertk, key)
 *inserti = top
 h.count++

done:
 return val
}
```

微信搜一搜

猿码记

3分钟前点击了阅读原文

戳“阅读原文”我们一起进步

收录于合集 #Go 101

上一篇

Go底层探索(三):切片

下一篇

Go底层探索(五):哈希表Map-扩容[下篇]

Read more

People who liked this content also liked

Python常用库(二):数学计算

猿码记