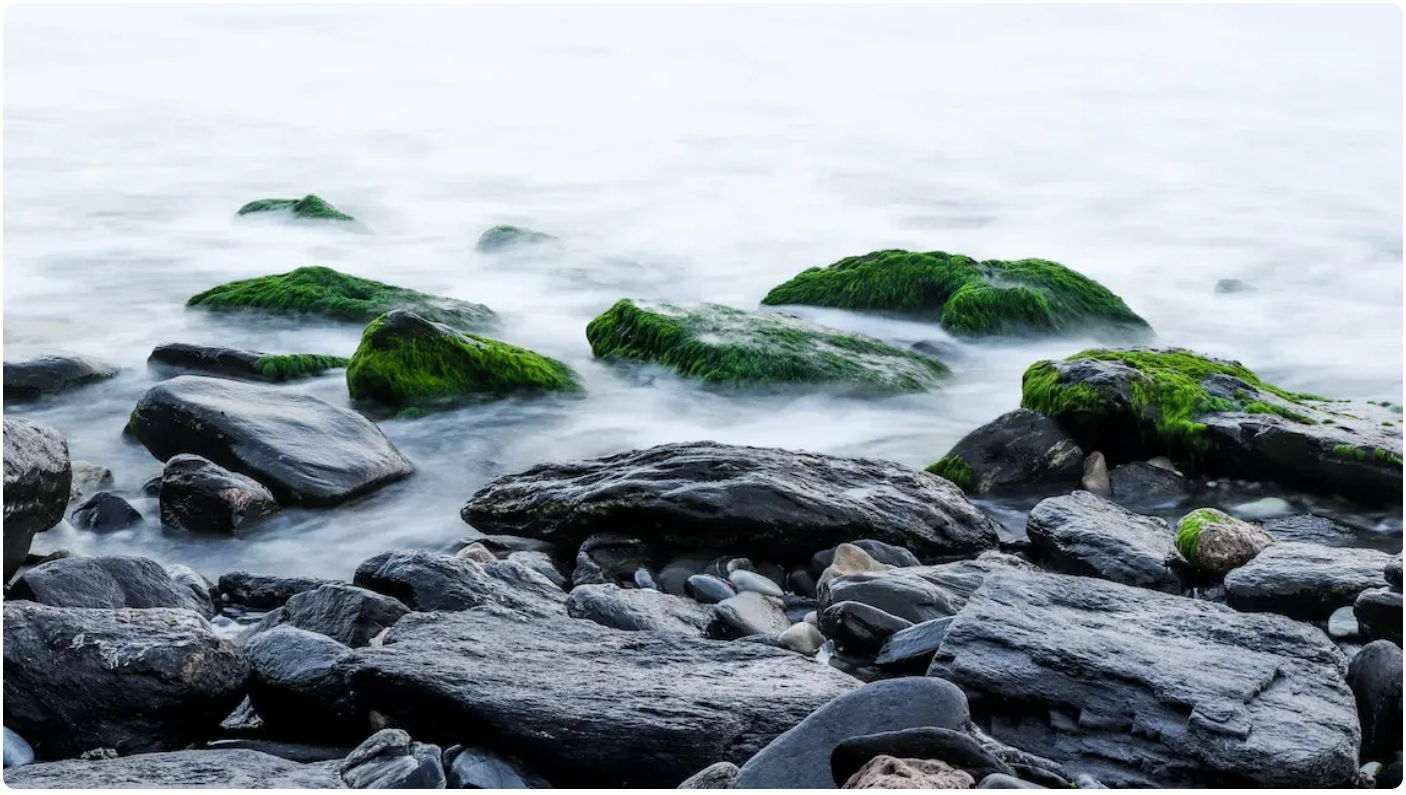
29 | 细节决定成败：切片与哈希表的陷阱与原理

2022-12-15 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >



讲述：郑建勋

时长 09:47　大小 8.94M

你好，我是郑建勋。

这节课，让我们来看一看切片与哈希表的原理。

我想先考你两道面试题。下面的代码中，foo 与 bar 最后的值是什么？

复制代码

```
foo := []int{0,0,0,42,100}
bar := foo[1:4]
bar[1] = 99
fmt.Println("foo:", foo)
fmt.Println("bar:", bar)
```

下面的程序又会输出什么呢?

```
x := []int{1, 2, 3, 4}
y := x[:2]
fmt.Println(cap(x), cap(y))
y = append(y, 30)
fmt.Println("x:", x)
fmt.Println("y:", y)
```


其实之前我们在初始化 seeds 切片的时候，也有一些不合理之处。你发现了吗？

```
var seeds []*collect.Request
```


切片和哈希表是 Go 语言内置、并且使用广泛的结构。如果你对上面问题的答案都很模糊，很可能就是不太理解切片底层的原理。理清这些原理可以帮助我们更好地规避常见陷阱，写出高性能的代码。

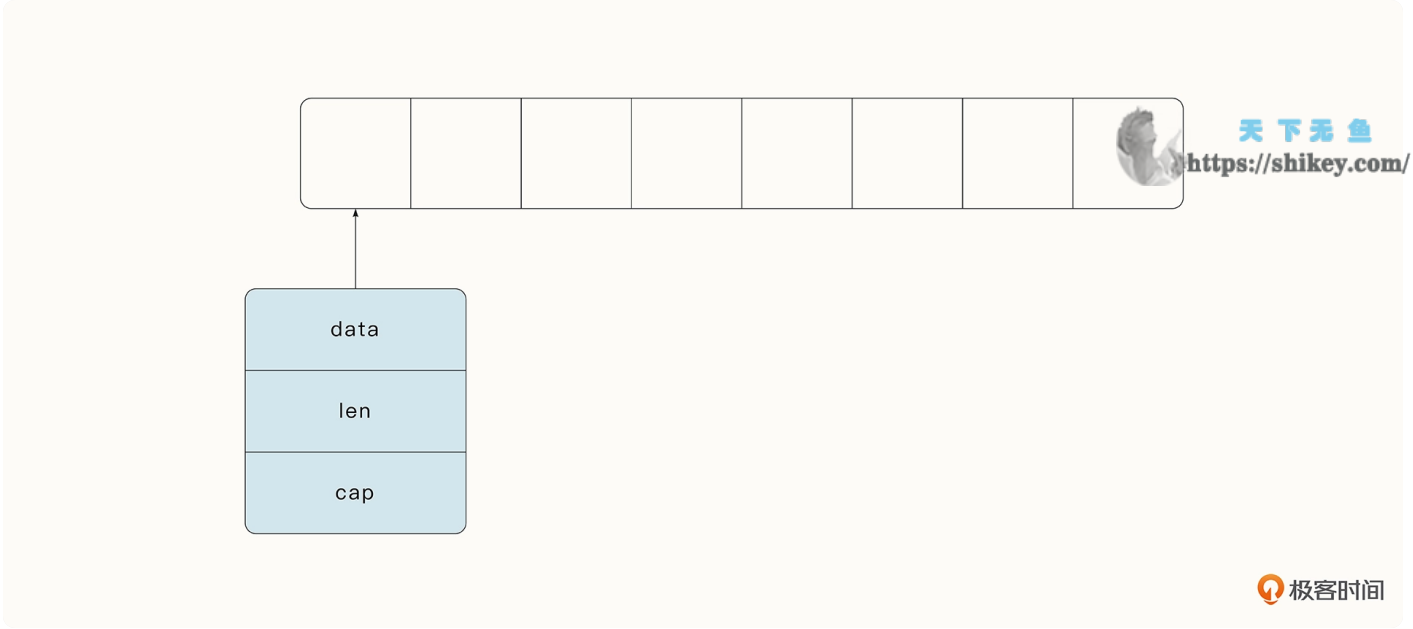
切片的底层原理

我们先来看看切片的底层原理。

和 C 语言中的数组是一个指针不同，Go 中的切片是一个复合结构。一个切片在运行时由指针（data）、长度（len）和容量（cap）三部分构成。

```
type SliceHeader struct {
  Data uintptr
  Len int
  Cap int
}
```


- 指针 data 指向切片元素对应的底层数组元素的地址。
- 长度 len 对应切片中元素的数目，总长度不能超过容量。
- 容量 cap 提供了额外的元素空间，可以在之后更快地添加元素。容量的大小一般指的是从切片的开始位置到底层数据的结尾位置的长度。



切片的截取

了解了切片的底层结构，我们来看看切片在截取时发生了什么。

切片在被截取时的一个特点是，截取后的切片长度和容量可能会发生变化。

和数组一样，切片中的数据仍然是内存中一片连续的区域。要获取切片某一区域的连续数据，可以通过下标的方式对切片进行截断。被截取后的切片，它的长度和容量都发生了变化。就像下面这个例子，numbers 切片的长度为 8。number1 截取了 numbers 切片中的第 2、3 号元素。number1 切片的长度变为了 2，容量变为了 6（即从第 2 号元素开始到元素数组的末尾）。

复制代码

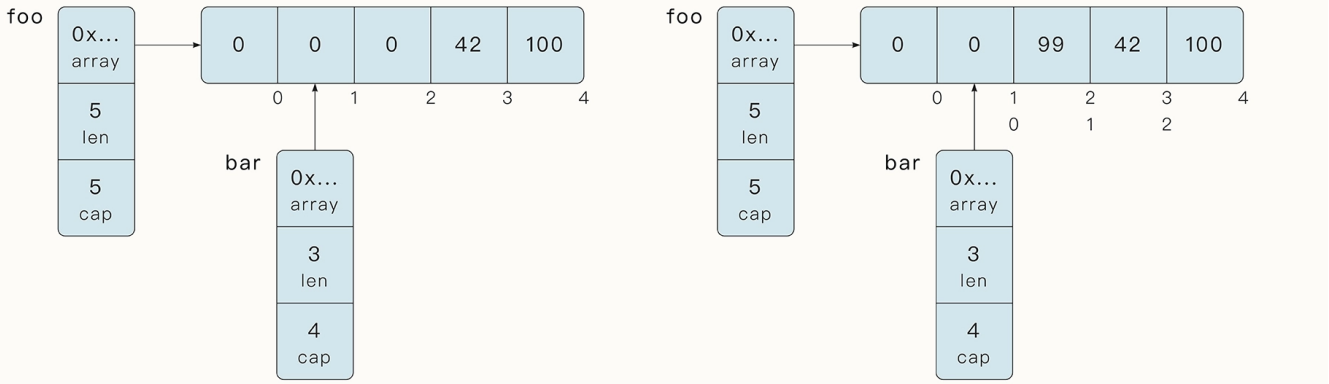
```
numbers:= []int{1,2,3,4,5,6,7,8}
// 从下标2 一直到下标4，但是不包括下标4
numbers1 :=numbers[2:4]
// 从下标0 一直到下标3，但是不包括下标3
numbers2 :=numbers[:3]
// 从下标3 一直到结尾
numbers3 :=numbers[3:]
```

切片在被截取时的另一个特点是，被截取后的数组仍然指向原始切片的底层数据。例如之前提到的案例，bar 截取了 foo 切片中间的元素，并修改了 bar 中的第 2 号元素。

复制代码

```
foo := []int{0,0,0,42,100}
bar := foo[1:4]
bar[1] = 99
```


底层结构图如下：



这时，bar 的 cap 容量会到原始切片的末尾，所以当前 bar 的 cap 长度为 4。

这意味着什么呢？我们看下面的例子，bar 执行了 append 函数之后，最终也修改了 foo 的最后一个元素，这是一个在实践中非常常见的陷阱。

复制代码

```
foo := []int{0, 0, 0, 42, 100}
bar := foo[1:4]
bar = append(bar, 99)
fmt.Println("foo:", foo) // foo: [0 0 0 42 99]
fmt.Println("bar:", bar) // bar: [0 0 42 99]
```

如果要解决这样的问题，其实可以在截取时指定容量：

复制代码

```
foo := []int{0,0,0,42,100}
bar := foo[1:4:4]
bar = append(bar, 99)
fmt.Println("foo:", foo) // foo: [0 0 0 42 100]
fmt.Println("bar:", bar) // bar: [0 0 42 99]
```

foo[1:4:4] 这种方式可能很多人没有见到过。这里，第三个参数 4 代表 cap 的位置一直到下标 4，但是不包括下标 4。 所以当前 bar 的 Cap 变为了 3，和它的长度相同。当 bar 进行 append 操作时，将发生扩容，它会指向与 foo 不同的底层数据空间。

切片的扩容

Go 语言内置的 append 函数可以把新的元素添加到切片的末尾，它可以接受可变长度的元素，并且可以自动扩容。如果原有数组的长度和容量已经相同，那么在扩容后，长度和容量都会相 应增加。

如下所示，numbers 切片一开始的长度和容量都是 4，添加一个元素后，它的长度变为了 5，容量变为 8，相当于扩容了一倍。


```
numbers:= []int{1,2,3,4}
numbers = append(numbers,5)
```

不过，Go 语言并不会每增加一个元素就扩容一次，这是因为扩容常会涉及到内存的分配，频繁扩容会减慢 append 的速度。append 函数在运行时调用了 runtime/slice.go 文件下的 growslice 函数：


```
func growslice(et *_type, old slice, cap int) slice {
  newcap := old.cap
  doublecap := newcap + newcap
  if cap > doublecap {
    newcap = cap
  } else {
    if old.len < 1024 {
      newcap = doublecap
    } else {
      for 0 < newcap && newcap < cap {
        newcap += newcap / 4
      }
      if newcap <= 0 {
        newcap = cap
      }
    }
  }
  ...
}
```

上面这段代码显示了扩容的核心逻辑。Go 语言中切片扩容的策略为：

- 如果新申请容量（cap）大于旧容量（old.cap）的两倍，则最终容量（newcap）是新申请的容量（cap）；
- 如果旧切片的长度小于 1024，则最终容量是旧容量的 2 倍，即“newcap=doublecap”；
- 如果旧切片的长度大于或等于 1024，则最终容量从旧容量开始循环增加原来的 1/4，直到最终容量大于或等于新申请的容量为止；
- 如果最终容量计算值溢出，即超过了 int 的最大范围，则最终容量就是新申请容量。

切片的这种扩容机制是深思熟虑的结果。一开始切片容量小，扩容得更多一些可以确保扩容不用太频繁。容量变大之后，按照比例扩容也会有足够多的元素空间被开辟出来。

切片动态扩容的机制启发我们，在一开始就要分配好切片的容量。否则频繁地扩容会影响程序的性能。所以我们可以将爬虫项目的容量扩展到 1000，注意长度需要为 0。

```
1 var seeds = make([]*collect.Request, 0, 1000)
```

哈希表原理

和切片一样，哈希表也面临着相同的性能陷阱。哈希表是使用频率极高的一种数据结构，在实践中，我们通常将哈希表看作 o(1) 时间复杂度的操作，可以通过一个键快速寻找其唯一对应的值（Value）。在很多情况下，哈希表的查找速度明显快于一些搜索树形式的数据结构，因此它被广泛用于关联数组、缓存、数据库缓存等场景。

哈希表的原理是将多个键 / 值对（Key/Value）分散存储在 Buckets（桶）中。给定一个键（Key），哈希（Hash）算法会计算出键值对存储的桶的位置。找到存储桶的位置通常包括两步，伪代码如下：

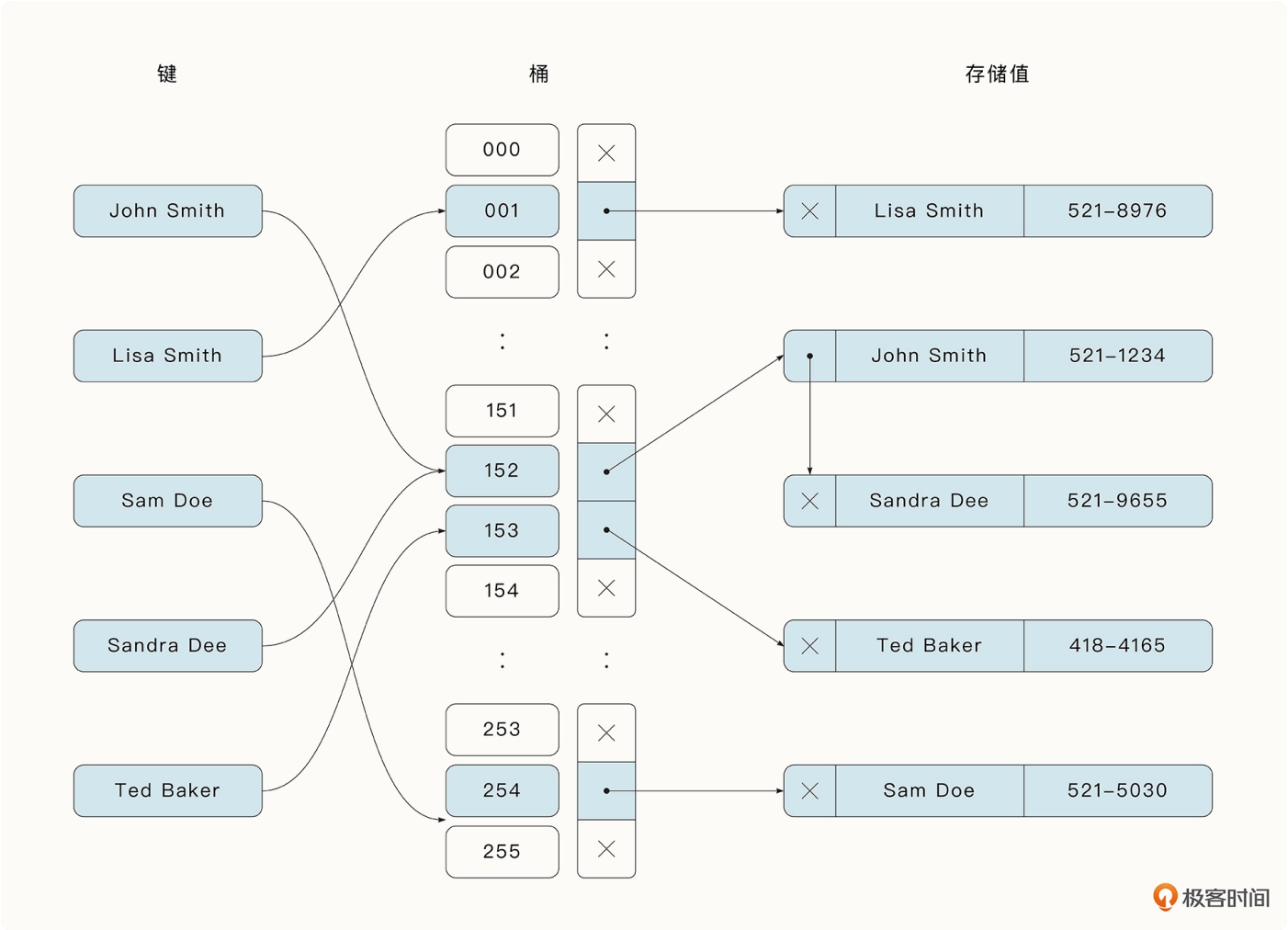
```
1 hash = hashfunc(key)
2 index = hash % array_size
```

哈希碰撞

哈希函数在实际运用中最常见的问题是哈希碰撞（Hash Collision），即不同的键使用哈希算法可能产生相同的哈希值。如果将 2450 个键随机分配到一百万个桶中，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超过 95%。哈希碰撞导致同一个桶中可能存在多个元素，会减慢数据查找的速度。

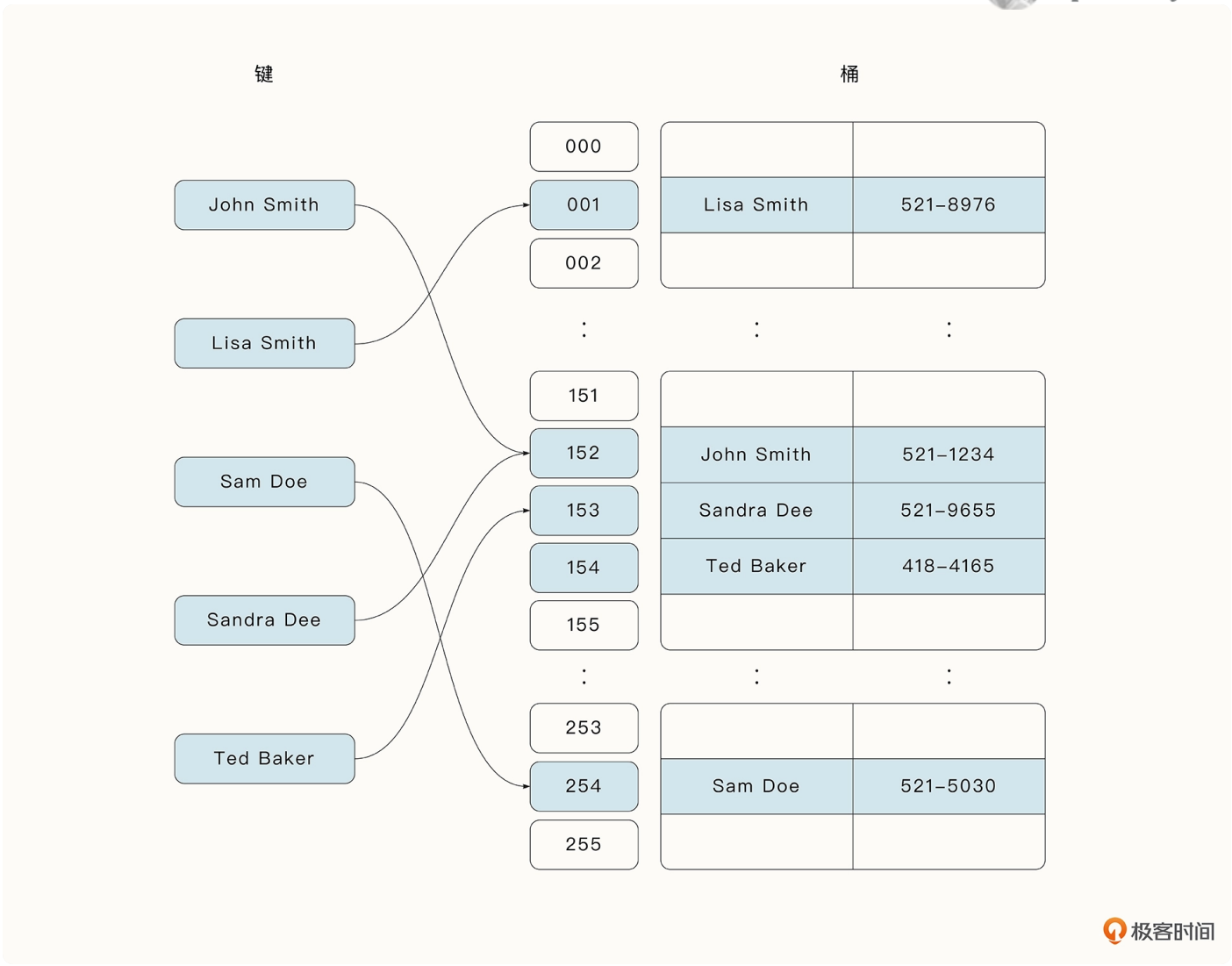
有多种方式可以避免哈希碰撞，常用的两种策略是：**拉链法和开放寻址法。**

如图所示，拉链法是将同一个桶中的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。随着桶中元素的增加，我们可以不断链接新的元素，而且不用预先为元素分配内存。不过拉链法的不足之处在于，我们需要存储额外的指针来链接元素，这就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 缓存。



与拉链法对应的另一种解决哈希碰撞的策略为开放寻址法（Open Addressing）。这种方法是将所有元素都存储在桶的数组中。当必须插入新条目时，**开放寻址法**将按某种探测策略顺序

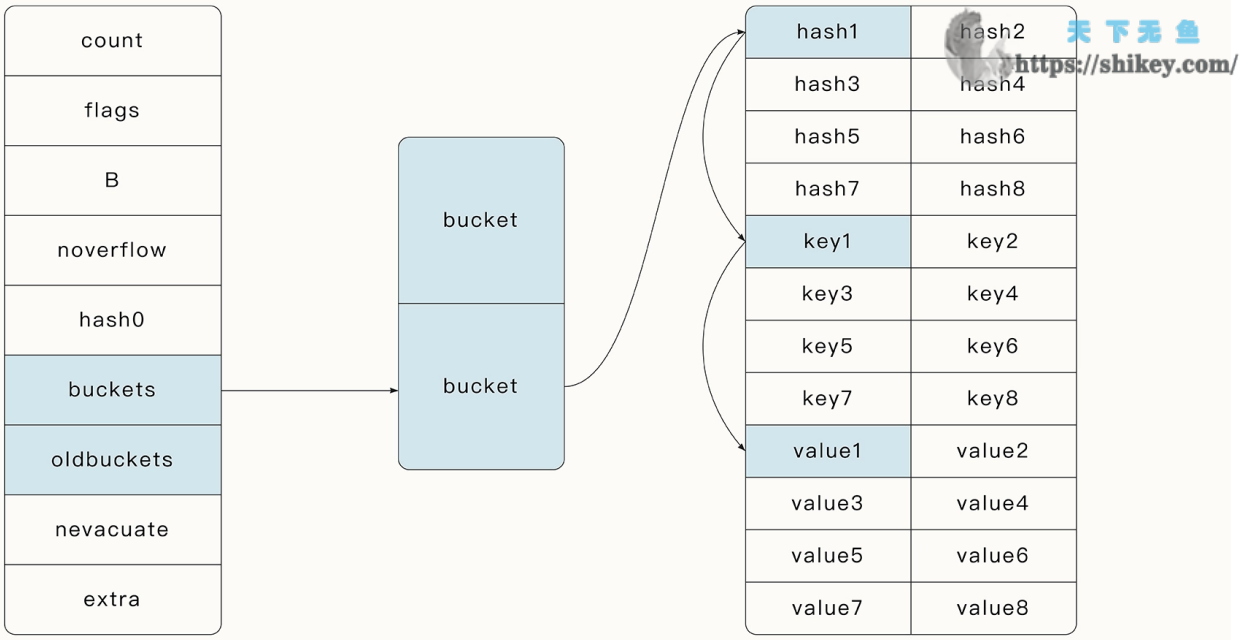
查找，直到找到未使用的数组插槽为止。当搜索元素时，**开放寻址法**将按相同顺序扫描存储桶，直到查找到目标记录或找到未使用的插槽为止。



Go 语言中的哈希表采用的是优化的拉链法，它在桶中存储了 8 个元素用于加速访问。

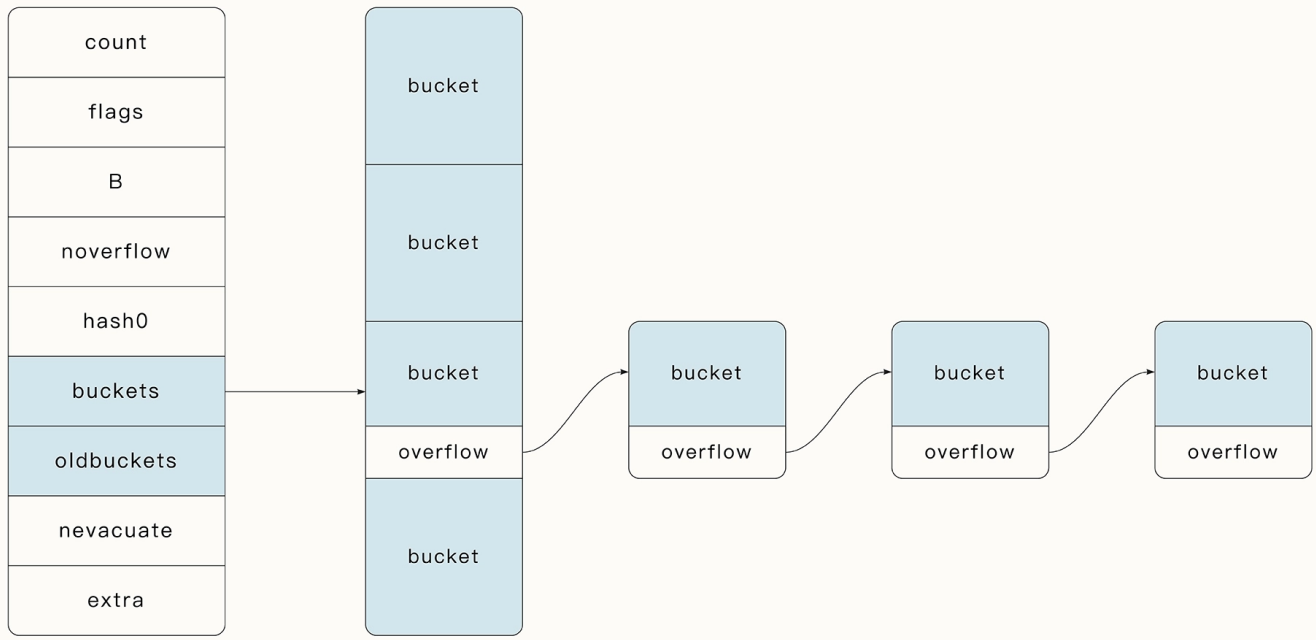
哈希表读取

在 Go 中进行 Map 访问操作时，会首先通过上面的伪代码找到桶的位置，之后遍历桶中的 tophash 数组。tophash 数组存储了 8 个元素用于加速访问。如果在 tophash 数组中找到了相同的 hash 值，就可以通过指针的寻址操作找到对应的 Key 与 Value。



此外，在 Go 语言中还有一个**溢出桶**的概念。在执行 hash[key] = value 的赋值操作时，当指定桶中的数据超过 8 个时，并不会直接开辟一个新桶，而是将数据放置到溢出桶中，每个桶的最后都存储了 overflow，即溢出桶的指针。

在正常情况下，数据是很少会跑到溢出桶里面去的。同理，在 Map 执行查找操作时，如果 Key 的 hash 不在指定桶的 tophash 数组中，我们就需要遍历溢出桶中的数据。



哈希表重建原理

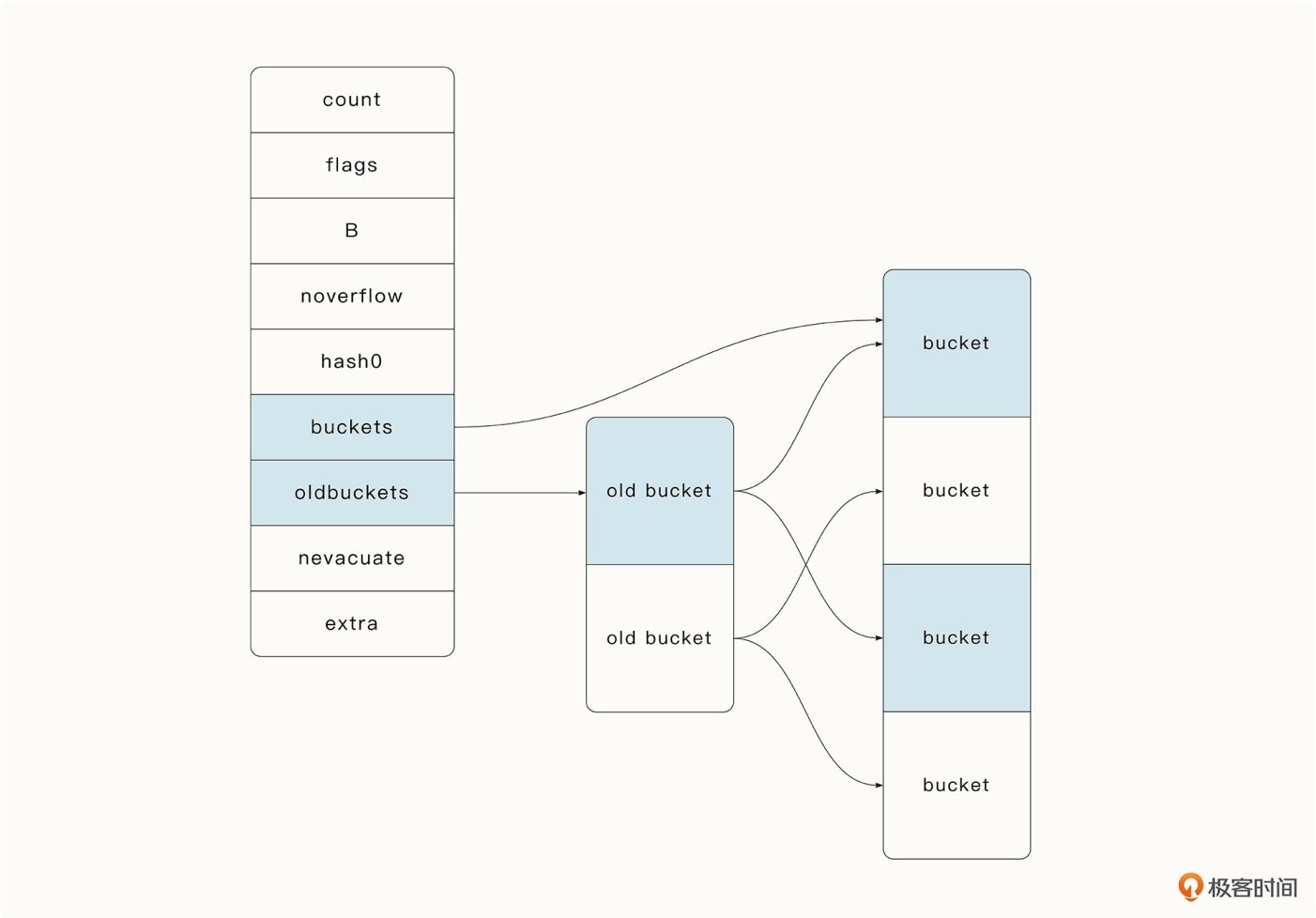
不过，如果溢出桶的数量过多，或者 Map 超过了负载因子大小，Map 就要进行重建。负载因子是哈希表中的经典概念：

```
1 负载因子 = 哈希表中的元素数量 / 桶的数量
```

复制代码

负载因子的增大意味着更多的元素会被分配到同一个桶中，此时效率会减慢。试想一下，如果桶的数量只有 1 个，这个时候负载因子到达最大，搜索就和遍历数组一样了，它的复杂度为 o(n)。

Go 语言中的负载因子为 6.5，当哈希表负载因子的大小超过 6.5 时，Map 就会扩容，变为旧表的两倍。当旧桶中的数据全部转移到新桶后，旧桶就会被清空。Map 的重建还存在第二种情况，那就是溢出桶的数量太多。这时 Map 只会新建和原来一样大的桶，目的是防止溢出桶的数量缓慢增长导致内存泄露。



Map 桶扩容原理

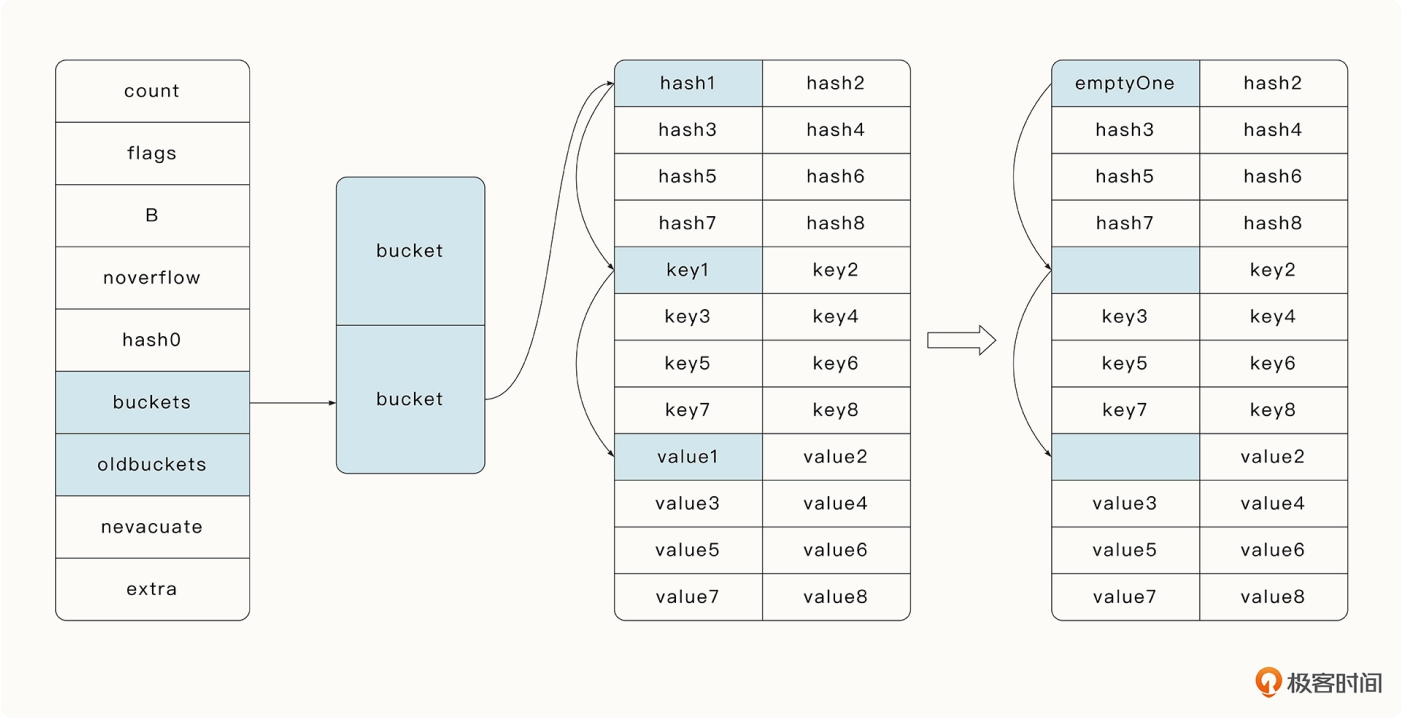
哈希表的重建过程提示我们，可以在初始化时评估并指定放入 Map 的数据大小，从而减少重建的性能消耗。

哈希表删除原理

再来看下删除哈希表的底层原理。

和赋值操作类似，当进行 Map 的 delete 函数时，delete 函数会根据 Key 找到指定的桶，如果存在指定的 Key，那么就释放掉 Key 与 Value 引用的内存。tophash 中的指定位置会存储 emptyOne，代表当前位置是空的。

同时，删除操作会探测当前要删除的元素之后是否都是空的。如果是， tophash 会存储为 emptyRest。这样做的好处是在进行查找时，遇到 emptyRest 可以直接退出，提高了查找效率。



Map 删除原理

总结

切片与哈希表是 Go 中使用非常频繁的数据结构，然而在实践过程中，由于对它们的内部结构和运行机制不了解，我们容易陷入到一些陷阱中，无意识地写出低性能的代码。

切片和哈希表的扩容机制提醒我们，在实践当中一定要评估容器容纳的数据量大小，并在初始化时候指定容量，这会提高程序的性能。

此外，混合使用切片截取和 append 非常容易犯错，所以我们要尽量避免这种用法。如果必须使用，也要确认自己真的理解了切片的底层图像，防止切片的误操作。

最后，虽然哈希表在实践中极少成为性能的瓶颈，但是开发者在实践中也容易写出 Map 并发读写冲突的程序，所以在使用哈希表时，我们需要进行合理的程序设计和必要的 race 检查。

课后题

学完这节课，请你思考下面两个问题。

1. Go 的哈希表为什么不是并发安全的？
2. 在实践中，怎么才能够并发安全地操作哈希表？

欢迎你在留言区与我交流讨论，我们下节课见！

上一篇　28 | 调度引擎：负载均衡与调度器实战

下一篇　30 | 辅助任务管理：任务优先级、去重与失败处理

精选留言 (1)

写留言

Realm

2022-12-15 来自浙江

1. map是指针型变量，多个协程同时写同一个内存时，会出现data race；

2. 写操作加锁，或者使用sync.Map；