

带着几个问题阅读本文：

1. go map 实现方法？如何解决hash冲突的？
2. go map是否线程安全？
3. go map 的扩容机制？

# 什么是map？

■ 由一组 <key, value> 对组成的抽象数据结构，并且同一个 key 在map中只会出现一次

map 的设计也被称为 “The dictionary problem”，它的任务是设计一种数据结构用来维护一个集合的数据，并且可以同时对该集合进行增删查改的操作。最主要的数据结构有两种：哈希查找表（Hash table）、搜索树（Search tree）。

哈希查找表用一个哈希函数将 key 分配到不同的桶（bucket，也就是数组的不同 index）。这样，开销主要在哈希函数的计算以及数组的常数访问时间。在很多场景下，哈希查找表的性能很高。

哈希查找表一般会存在“碰撞”的问题，就是说不同的 key 被哈希到了同一个 bucket。一般有两种应对方法：链表法和开放地址法。链表法将一个 bucket 实现成一个链表，落在同一个 bucket 中的 key 都会插入这个链表。开放地址法则是碰撞发生后，通过一定的规律，在数组的后面挑选“空位”，用来放置新的 key。

搜索树法一般采用自平衡搜索树，包括：AVL 树，红黑树 c++中STL\_MAP 是红黑树结构

自平衡搜索树法的最差搜索效率是  $O(\log N)$ ，而哈希查找表最差是  $O(N)$ 。当然，哈希查找表的平均查找效率是  $O(1)$ ，如果哈希函数设计的很好，最坏的情况基本不会出现。还有一点，遍历自平衡搜索树，返回的 key 序列，一般会按照从小到大的顺序；而哈希查找表则是乱序的

## map的用法

```
package main

import "fmt"

func main(){
    m1 := map[string]string{ // :=创建
        "name": "小明",
        "age": "20",
    }
    //遍历map
    for k ,v :=range m1{
        fmt.Println(k, v)
    }

    // 测试key是否存在, 存在ok=true 否则ok=false
    if name, ok := m1["name"]; ok { //如果name存在ok就为true
        fmt.Println(name, ok)
    }
```

```

m2 := make(map[string]string) //通过make创建
m2["city"] = "shanghai"

//修改
m2["city"] = "beijing"

//删除key
delete(m2, "city")

var m3 map[string]int //通过var 注意此时的map是一个nil map 无法插入key/value
fmt.Println(m3)
m3 = make(map[string]int)
m3["count"] = 100
}

```

## map的类型:

golang中的map是一个 指针。当执行语句 `make(map[string]string)` 的时候，其实是调用了 `makemap` 函数：

```

// file: runtime/hashmap.go:L222
func makemap(t *maptype, hint64, h *hmap, bucket unsafe.Pointer) *hmap

```

显然，`makemap` 返回的是指针。

因为返回的是指针，map作为参数的时候，函数内部能修改map。

我们知道slice 也可以使用make初始化，makeslice返回的是结构体

```

func makeslice(et *_type, len, cap int) slice

// runtime/slice.go
type slice struct {
    array unsafe.Pointer // 元素指针
    len   int // 长度
    cap   int // 容量
}

```

## go hmap 数据结构

go map 采用的是哈希查找表，并且使用链表解决哈希冲突

```

type hmap struct {
    count      int //map元素的个数，调用Len()直接返回此值

    // map标记:

```

```

// 1. key和value是否包指针
// 2. 是否正在扩容
// 3. 是否是同样大小的扩容
// 4. 是否正在 `range` 方式访问当前的buckets
// 5. 是否有 `range` 方式访问旧的bucket
flags      uint8

B          uint8 // buckets 的对数 log_2
noverflow  uint16 // overflow 的 bucket 近似数
hash0      uint32 // hash种子 计算 key 的哈希的时候会传入哈希函数
buckets    unsafe.Pointer // 指向 buckets 数组, 大小为 2^B 如果元素个数为0, 就为 nil

// 扩容的时候, buckets 长度会是 oldbuckets 的两倍
oldbuckets unsafe.Pointer // bucket slice指针, 仅在扩容的时候不为nil

nevacuate  uintptr // 扩容时已经移到新的map中的bucket数量
extra      *mapextra // optional fields
}

```

注意：**B** 是buckets 数组的长度的对数，也就是说 buckets 数组的长度就是  $2^B$ 。bucket 里面存储了 key 和 value。

**buckets** 是一个指针，最终它指向的是一个结构体：（buckets是bmap类型的数组，数组长度是  $2^B$ ）

```

// A bucket for a Go map.
type bmap struct {
    tophash [bucketCnt]uint8
}

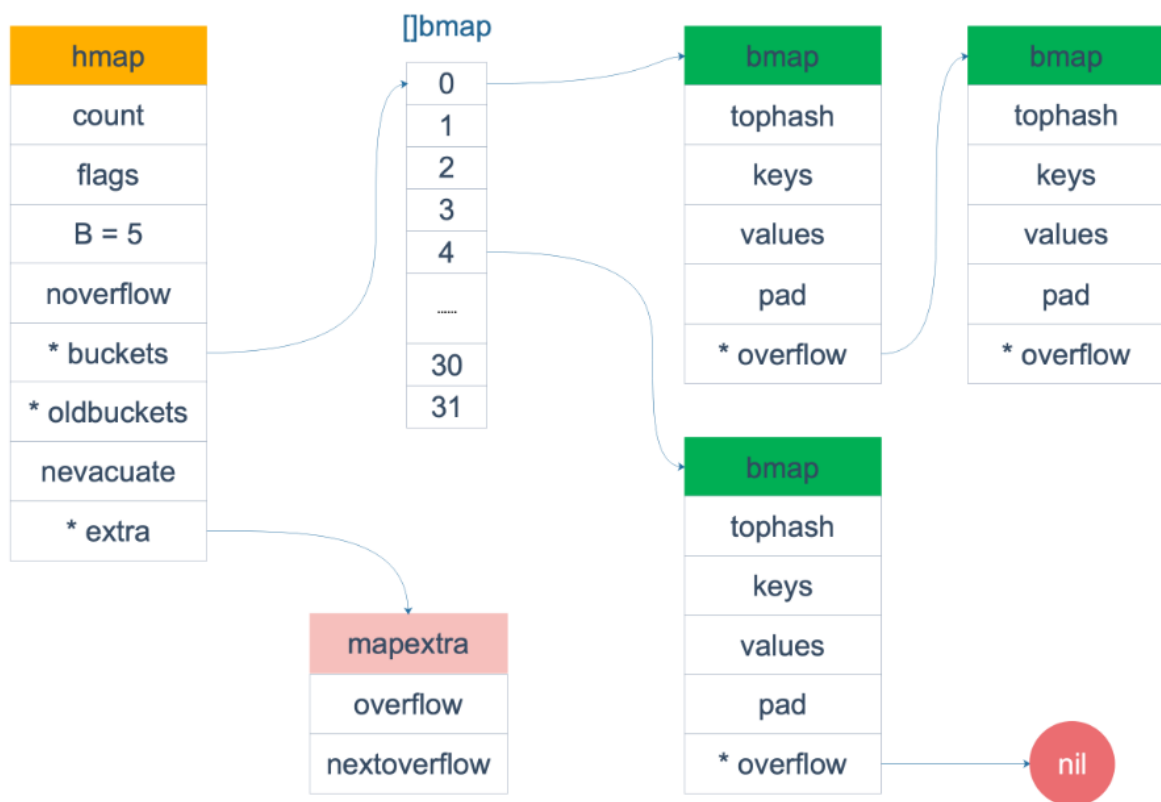
```

**bmap**就是我们所说的桶bucket，实际上就是每个bucket固定包含8个key和value(可以查看源码 bucketCnt=8).实现上面是一个固定的大小连续内存块，分成四部分：

1. 每个条目的状态
2. 8个key值
3. 8个value值
4. 指向下个bucket的指针

桶里面会最多装 8 个key，这些key之所以会落入同一个桶，是因为它们经过哈希计算后，哈希结果是“一类”的。在桶内，又会根据 key 计算出来的 hash 值的高 8 位来决定 key 到底落入桶内的哪个位置（一个桶内最多有8个位置）。

查看下图：B=5 表示hmap的有 $2^5=32$ 个bmap：buckets是一个bmap数组，其长度为32。每个bmap有8个key



## hmap的数据结构



选择这样的布局的好处：由于对齐的原因，`key0/value0/key1/value1...` 这样的形式可能需要更多的补齐空间，比如 `map[int64]int8`，1字节的value后面需要补齐7个字节才能保证下一个key是 `int64` 对齐的。

**每个 bucket 设计成最多只能放 8 个 key-value 对，如果有第 9 个 key-value 落入当前的 bucket，那就需要再构建一个 bucket，通过 overflow 指针连接起来**

