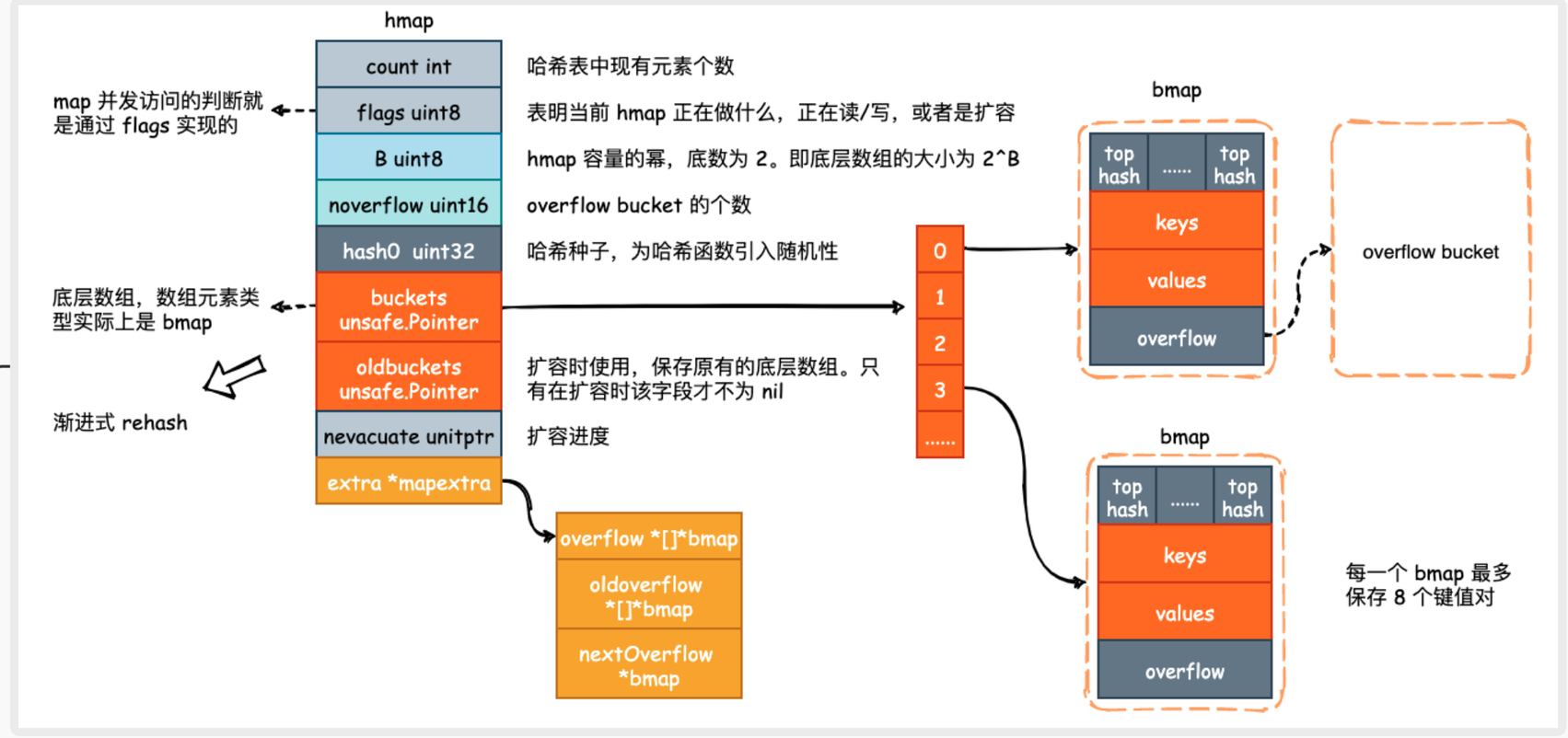
map

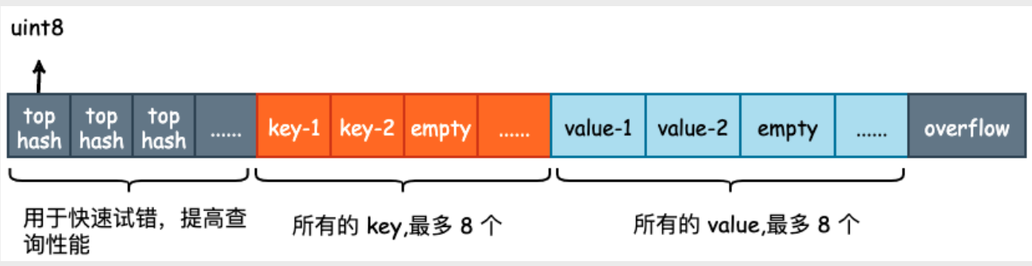
概览



- 从这张 map 的结构图中我们还是能够猜到很多东西的
  - go map 处理哈希冲突的方式为拉链法，同样未选择开放寻址法
  - 和 Redis 哈希表扩容的过程一样，go map 也是采用渐进式 rehash

bmap

- bmap 的设计很有意思，该结构本身的定义其实只有一个字段
  - type bmap struct {
  // bucketCnt 为常量 8
  tophash [bucketCnt]uint8
}
  - 剩余的字段，例如哈希表的 key 和 value，以及指向下一个 bucket 的指针等内容将会在编译期怼进 bmap 中



- bmap 在运行期间大致长这个样子，这些内容位于连续的内存空间内，同时也是 hmap.buckets 的组成内容
- tophash
  - tophash 数组其实是保存了当前 bmap 中所有 hash(key) 结果的前 8 位。当在查询一个 key 是否存在于当前 bmap 时，首先去判断待查询 key 哈希结果的前 8 位是否在 tophash 数组中
  - 如果不在数组中，那么直接去下一个 bmap 中进行查询就可以了，不需要再额外的取出 key 进行比较。如果在数组中，再取出对应的 key 来进行值比较，值相同则返回对应的 value 即可

```
top := tophash(hash)
for i := uintptr(0); i < bucketCnt; i++ {
    // 快速试错，如果查找 key 和当前 key 哈希值的
    // 前 8 位都不一样，那么 key 也肯定不一样
    if b.tophash[i] != top {
        continue
    }
}
```

- 另外一个比较有意思的地方在于 bmap 中的 key 和 value 是"分堆"存储的，也就是所有的 key 在一块儿，所有的 value 在一块儿。go 会使用指针偏移量的方式取出一个特定的 key 或 value
- bmap 中最多保存 8 个键值对，如果有更多的 key-value 需要保存的话，只能放在 overflow bucket 中，也就是另一个 bmap 中。可以把它看成是拉链法中链表的一个一个节点

扩容

扩容时机

```
if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
    hashGrow(t, h)
    goto again // Growing the table invalidates everything, so try again
}
```

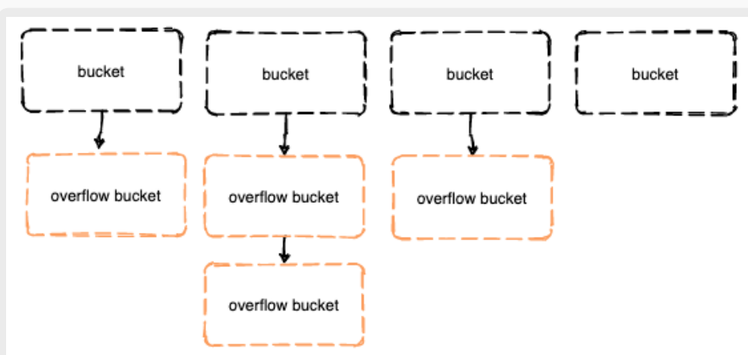
- **也就是说，如果当前 map 没有正在扩容，并且负载因子超过阈值 6.5 或者是 map 中存在了太多的 overflow buckets，都会发生扩容**
  - LoadFactor = count / 2^B
  - 如果所有的 buckets 都装满了 8 个元素的话，LoadFactor 的计算结果为 8
- tooManyOverflowBuckets() 这个函数会根据当前已有的 overflow buckets 和 map 容量综合进行判断

```
func tooManyOverflowBuckets(noverflow uint16, B uint8) bool {
    if B > 15 {
        B = 15
    }
    return noverflow >= uint16(1)<<(B&15)
}
```

  - 其实这段代码的意思就是当 overflow buckets 的数量大于等于 map 容量时即发生扩容
- 通过 LoadFactor 和 overflow buckets 的数量来综合判断能够有效的判断出整体 map 情况和个别区域的"扎堆"情况

扩容大小

- overflow buckets 数量过多
  - **如果是因为 overflow buckets 太多的话，那么在扩容时将会进行等量扩容，也就是扩容前和扩容后的容量相同**
  - 在没有达到 LoadFactor 的阈值时，证明整个 map 中的元素数量不会特别多，但是却出现了 overflow buckets 的数量超过 map 容量的情况。造成这种现象的原因就在于不停的插入、删除元素，但是峰值又没有达到 LoadFactor 阈值



    - 如左图所示，假设首先我们插入一堆数据，然后删除掉这些数据，在插入-删除这个过程中完全有可能会创建新的 overflow bucket
    - 然后不断地重复上述过程，插入-删除，插入-删除，这样一来，overflow buckets 的数量就会缓慢的增加，直到内存溢出。
    - 因为当我们删除元素时，并不会去清理 overflow bucket
  - 这就好比一开始有一大片儿空地，来了一堆人定居，开始造 🏠，然后一部分人外出打工；又来了一帮人了，又开始造 🏠，然后一部分人外出打工……最后空地上🏠很多，居民却很少，找到一个人得绕好半天
- LoadFactor 超过阈值 —— 如果是因为 LoadFactor 超过阈值的话，那么在扩容时只需要将 B + 1，使得新 map 是原来的 2 倍即可

扩容过程

- 不管是哪种扩容方式，都会有一个新的 buckets 数组被创建，并且我们需要把原来 buckets 中所有的数据全部 rehash 到新的 buckets 中。如果我们的 map 很大，那么这个过程将会阻塞主流程很长时间，因此使用渐进式 rehash
- 和 Redis 渐进式 rehash 的过程一样，将扩容的过程均摊到每一个插入、修改和删除的操作上，每次拷贝一些 buckets 到新的 buckets 上
- 对于 overflow buckets 的情况，我们可以不用进行 rehash，因为是等量扩容，所以原来 key 在哪个 bucket 上，现在也还是在哪个 bucket 上

日常使用

- 我们可以在迭代 map 的同时删除 key

```
record := map[string]string{}
record["name"] = "smartkeyerror"
record["mobile"] = "13666666666"

for k, v := range record {
    fmt.Println(k, v)
    delete(record, k)    // ok
}
```

  - 这一点 C++ 其实也可以做到，稍微做点儿修改即可

```
int main() {
    unordered_map<string, string> record;
    record["name"] = "smartkeyerror";
    record["mobile"] = "13666666666";

    for (auto it = record.begin(); it != record.end(); ) {
        cout << it->first << ", " << it->second << endl;
        record.erase(it++);
    }
    cout << record.empty() << endl;
}
```

- **无法获取 map 中 value 的地址**

```
record := map[string]string{
    "name": "smartkeyerror",
}
fmt.Println(&record["name"])    // Wrong
```

  - 一方面若 key 不存在，那么将返回零值，而零值是不可变对象，所以不可寻址
  - 另一方面就算 key 存在，map 在任何时候都可能发生扩容，也就是 value 的地址是会发生变化的，所以寻址的结果没有意义
  - 基于此，如果 map 中的 value 类型是一个结构体的话，那么我们是不能去修改结构体的内容的，除非保存结构体指针
- map 必须在初始化后使用，要么使用 make，要么使用字面量进行初始化