Macro View of Map Internals In Go

William Kennedy December 31, 2013

(mailto:bill@ardanlabs.com) (https://github.com/ardanlabs/gotraining)

(https://twitter.com/goinggodotnet)

Introduction

There are lots of posts that talk about the internals of slices, but when it comes to maps, we are left in the dark. I was wondering why and then I found the code for maps and it all made sense.

https://golang.org/src/runtime/hashmap.go (/broken-link)

At least for me, this code is complicated. That being said, I think we can create a macro view of how maps are structured and grow. This should explain why they are unordered, efficient and fast.

Creating and Using Maps

Let's look at how we can use a map literal to create a map and store a few values:

```
// Create an empty map with a key and value of type string
colors := map[string]string{}

// Add a few keys/value pairs to the map
colors["AliceBlue"] = "#F0F8FF"
colors["Coral"]     = "#FF7F50"
colors["DarkGray"]  = "#A9A9A9"
```

When we add values to a map, we always specify a key that is associated with the value. This key is used to find this value again without the need to iterate through the entire collection:

```
fmt.Printf("Value: %s", colors["Coral"])
```

If we do iterate through the map, we will not necessarily get the keys back in the same order. In fact, every time you run the code, the order could change:

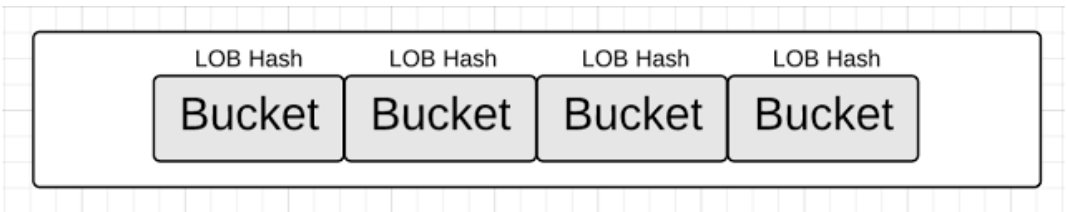
```
colors := map[string]string{}
colors["AliceBlue"]   = "#F0F8FF"
colors["Coral"]       = "#FF7F50"
colors["DarkGray"]    = "#A9A9A9"
colors["ForestGreen"] = "#228B22"
colors["Indigo"]      = "#4B0082"
colors["Lime"]        = "#00FF00"
colors["Navy"]        = "#000080"
colors["Orchid"]      = "#DA70D6"
colors["Salmon"]      = "#FA8072"

for key, value := range colors {
    fmt.Printf("%s:%s, ", key, value)
}

Output:
AliceBlue:#F0F8FF, DarkGray:#A9A9A9, Indigo:#4B0082, Coral:#FF7F50,
ForestGreen:#228B22, Lime:#00FF00, Navy:#000080, Orchid:#DA70D6,
Salmon:#FA8072
```

Now that we know how to create, set key/value pairs and iterate over a map, we can peek under the hood.

How Maps Are Structured
Maps in Go are implemented as a hash table. If you need to learn what a hash table is, there are lots of articles and posts about the subject. This is the Wikipedia page to get you started:

(http://en.wikipedia.org/wiki/Hash_table)http://en.wikipedia.org/wiki/Hash_table (http://en.wikipedia.org/wiki/Hash_table)

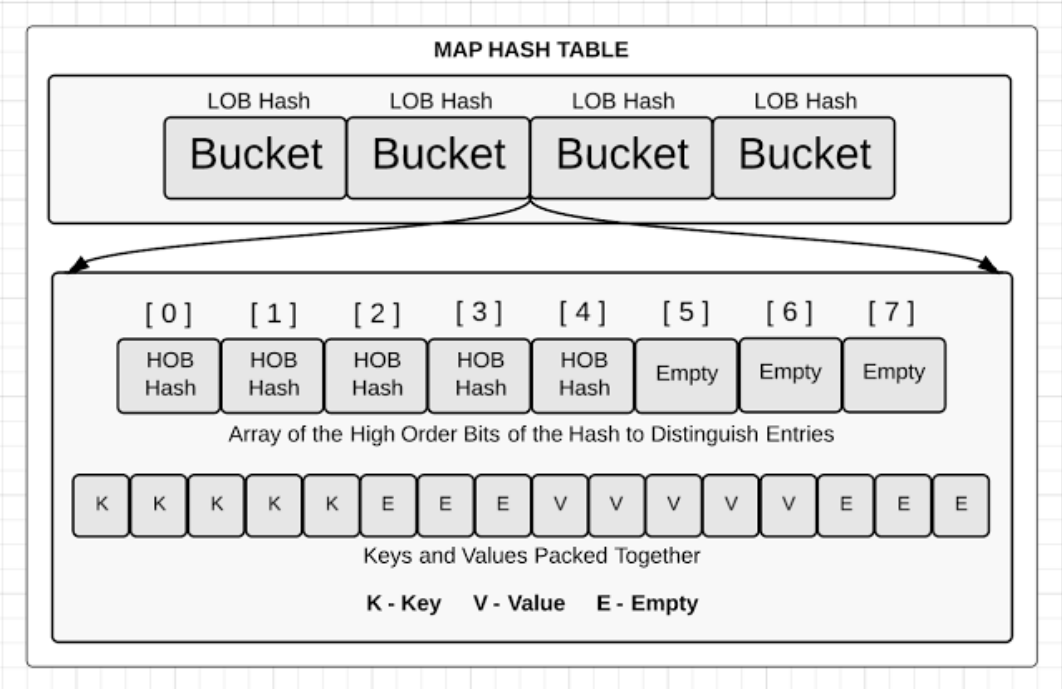
The hash table for a Go map is structured as an array of buckets. The number of buckets is always equal to a power of 2. When a map operation is performed, such as (**colors["Black"] = "#000000")**, a hash key is generated against the key that is specified. In this case the string "Black" is used to generate the hash key. The low order bits (LOB) of the generated hash key is used to select a bucket.



(../../images/goinggo/Screen+Shot+2013-12-31+at+6.35.43+PM.png)

Once a bucket is selected, the key/value pair needs to be stored, removed or looked up, depending on the type of operation. If we look inside any bucket, we will find two data structures. First, there is an array with the top 8 high order bits (HOB) from the same hash key that was used to select the bucket. This array distinguishes each individual key/value pair stored in the respective bucket. Second, there is an array of bytes that store the key/value pairs. The byte array packs all the keys and then all the values

together for the respective bucket.



(../../images/goinggo/Screen+Shot+2013-12-31+at+7.01.15+PM.png)

When we are iterating through a map, the iterator walks through the array of buckets and then return the key/value pairs in the order they are laid out in the byte array. This is why maps are unsorted collections. The hash keys determines the walk order of the map because they determine which buckets each key/value pair will end up in.

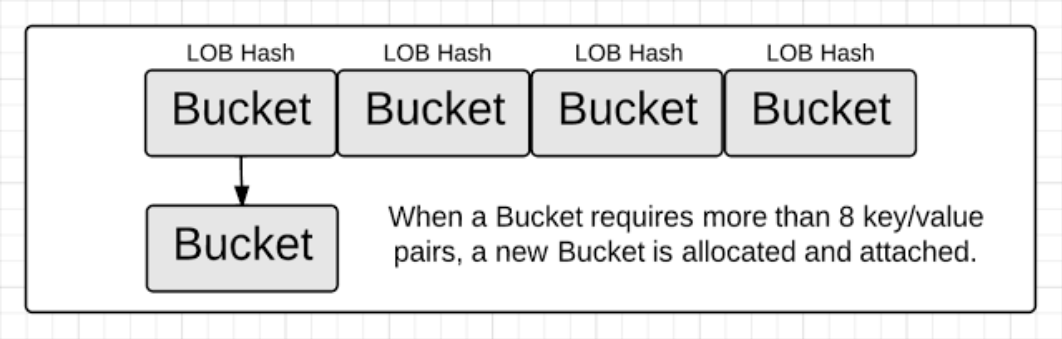
Memory and Bucket Overflow
There is a reason the key/value pairs are packed like this in a single byte array. If the keys and values were stored like key/value/key/value, padding allocations between each key/value pair would be needed to maintain proper alignment boundaries. An example where this would apply is with a map that looks like this:

```
map[int64]int8
```

The 1 byte value in this map would result in 7 extra bytes of padding per key/value pair. By packing the key/value pairs as key/key/value/value, the padding only has to be appended to the end of the byte array and not in between. Eliminating the padding bytes saves the bucket and the map a good amount of memory. To learn more about alignment boundaries, read this post:

(https://ardanlabs.com/blog/2013/07/understanding-type-in-go.html)https://ardanlabs.com/blog/2013/07/understanding-type-in-go.html (https://ardanlabs.com/blog/2013/07/understanding-type-in-go.html)

A bucket is configured to store only 8 key/value pairs. If a ninth key needs to be added to a bucket that is full, an overflow bucket is created and reference from inside the respective bucket.



(../../images/goinggo/Screen+Shot+2013-12-31+at+7.12.06+PM.png)

How Maps Grow

As we continue to add or remove key/value pairs from the map, the efficiency of the map lookups begin to deteriorate. The load threshold values that determine when to grow the hash table are based on these four factors:

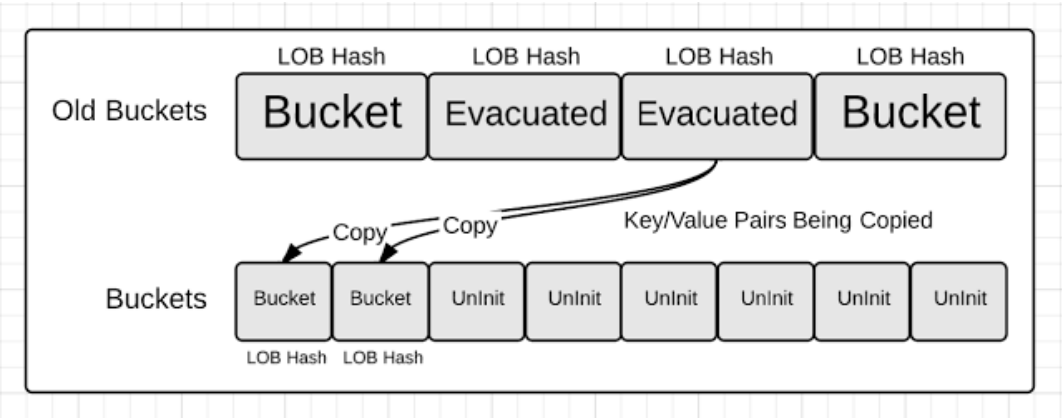
```
% overflow   : Percentage of buckets which have an overflow bucket
bytes/entry : Number of overhead bytes used per key/value pair
hitprobe    : Number of entries that need to be checked when looking up a key
missprobe   : Number of entries that need to be checked when looking up an absent key
```

Currently, the code uses the following load threshold values:

LOAD	%overflow	bytes/entry	hitprobe	missprobe
6.50	20.90	10.79	4.25	6.50

Growing the hash table starts with assigning a pointer called the "old bucket" pointer to the current bucket array. Then a new bucket array is allocated to hold twice the number of existing buckets. This could result in large allocations, but the memory is not initialized so the allocation is fast.

Once the memory for the new bucket array is available, the key/value pairs from the old bucket array can be moved or "evacuated" to the new bucket array. Evacuations happen as key/value pairs are added or removed from the map. The key/value pairs that are together in an old bucket could be moved to different buckets inside the new bucket array. The evacuation algorithm attempts to distribute the key/value pairs evenly across the new bucket array.



(../../images/goinggo/Screen+Shot+2013-12-31+at+7.22.39+PM.png)

This is a very delicate dance because iterators still need to run through the old buckets until every old bucket has been evacuated. This also affects how key/value pairs are returned during iteration operations. A lot of care has been taken to make sure iterators work as the map grows and expands.

Conclusion
As I stated in the beginning, this is just a macro view of how maps are structured and grow. The code is written in C and performs a lot of memory and pointer manipulation to keep things fast, efficient and safe.

Obviously, this implementation can be changed at any time and having this understanding doesn't affect our ability, one way or the other, to use maps. It does show that if you know how many keys you need ahead of time, it is best to allocated that space during initialization. It also explains why maps are unsorted collections and why iterators seem random when walking through maps.

Special Thanks
I would like to thank Stephen McQuay and Keith Randall for their review, input and corrections for the post.

Join Our Online Education Program

Our courses have been designed from training over 4,000 engineers since 2013 and they go beyond just being a language course. Our goal is to challenge every student to think about what they are doing and why.

✚ENROLL NOW (HTTPS://EDUCATION.ARDANLABS.COM)

Training (/training)

Development (/development)

DevOps (/devops-consulting)

Consulting (/consulting)

UI/UX (/ui-ux)

Machine Learning (/machine-learning)

Education (https://education.ardanlabs.com/)

Events (/live-training-events)

Podcast (https://ardanlabs.buzzsprout.com/)

YouTube (https://www.youtube.com/channel/UCCgGRKeRM1b0LTDqqb4NqjA)

Blog (/blog)

About (/about)

News (/news)

Contact (/my/contact-us)

Careers (/careers)

My Lab (/my/lab)

Terms of Service (/terms-service)

Privacy (/privacy-policy)

Reach Us

(888) 72 ARDAN
888 722-7326

hello@ardanlabs.com (mailto:hello@ardanlabs.com)