The acme of foolishness < https://dave.cheney.net/>

How the Go runtime implements maps efficiently (without generics)

This post discusses how maps are implemented in Go. It is based on a presentation I gave at the **GoCon Spring 2018 < https://gocon.connpass.com/event/82515/>** conference in Tokyo, Japan.

What is a map function?

To understand how a map works, let's first talk about the idea of the *map function*. A map function maps one value to another. Given one value, called a *key*, it will return a second, the *value*.

```
map(key) → value
```

Now, a map isn't going to be very useful unless we can put some data in the map. We'll need a function that adds data to the map

```
insert(map, key, value)
```

and a function that removes data from the map

```
delete(map, key)
```

There are other interesting properties of map implementations like querying if a key is present in the map, but they're outside the scope of what we're going to discuss today. Instead we're just going to focus on these properties of a map; insertion, deletion and mapping keys to values.

Go's map is a hashmap

The specific map implementation I'm going to talk about is the *hashmap*, because this is the implementation that the Go runtime uses. A hashmap is a classic data structure offering O(1) lookups on average and O(n) in the worst case. That is, when things are working well, the time to execute the map function is a near constant.

The size of this constant is part of the hashmap design and the point at which the map moves from O(1) to O(n) access time is determined by its *hash function*.

The hash function

What is a hash function? A hash function takes a key of an unknown length and returns a value with a fixed length.

```
hash(key) → integer
```

this *hash value* is almost always an integer for reasons that we'll see in a moment.

Hash and map functions are similar. They both take a key and return a value. However in the case of the former, it returns a value *derived* from the key, not the value *associated* with the key.

Important properties of a hash function

It's important to talk about the properties of a good hash function as the quality of the hash function determines how likely the map function is to run near O(1).

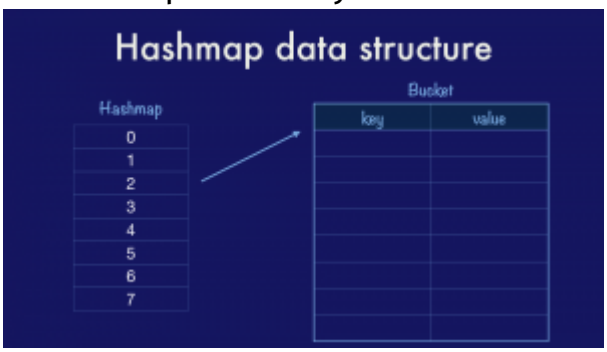
When used with a hashmap, hash functions have two important properties. The first is *stability*. The hash function must be stable. Given the same key, your hash function must return the same answer. If it doesn't you will not be able to find things you put into the map.

The second property is *good distribution*. Given two near identical keys, the result should be wildly different. This is important for two reasons. Firstly, as we'll

see, values in a hashmap should be distributed evenly across buckets, otherwise the access time is not O(1). Secondly as the user can control some of the aspects of the input to the hash function, they may be able to control the output of the hash function, leading to poor distribution which has been a DDoS vector for some languages. This property is also known as *collision resistance*.

The hashmap data structure

The second part of a hashmap is the way data is stored.



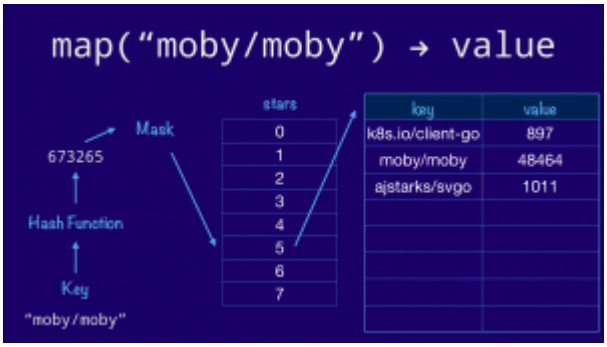
The classical hashmap is an array of *buckets* each of which contains a pointer to an array of key/value entries. In this case our hashmap has eight buckets (as this is the value that the Go implementation uses) and each bucket can hold up to eight entries each (again drawn from the Go implementation). Using powers of two allows the use of cheap bit masks and shifts rather than expensive division.

As entries are added to a map, assuming a good hash function distribution, then the buckets will fill at roughly the same rate. Once the number of entries across each bucket passes some percentage of their total size, known as the *load factor*, then the map will grow by doubling the number of buckets and redistributing the entries across them.

With this data structure in mind, if we had a map of project names to GitHub stars, how would we go about inserting a value into the map?

We start with the key, feed it through our hash function, then mask off the bottom few bits to get the correct offset into our bucket array. This is the

bucket that will hold all the entries whose hash ends in three (011 in binary). Finally we walk down the list of entries in the bucket until we find a free slot and we insert our key and value there. If the key was already present, we'd just overwrite the value.



Now, lets use the same diagram to look up a value in our map. The process is similar. We hash the key as before, then masking off the lower 3 bits, as our bucket array contains 8 entries, to navigate to the fifth bucket (101 in binary). If our hash function is correct then the string `"moby/moby"` will always hash to the same value, so we know that the key will not be in any other bucket. Now it's a case of a linear search through the bucket comparing the key provided with the one stored in the entry.

Four properties of a hash map

That was a very high level explanation of the classical hashmap. We've seen there are four properties you need to implement a hashmap;

1. You need a hash function for the key.
2. You need an equality function to compare keys.
3. You need to know the size of the key and,
4. You need to know the size of the value because these affect the size of the bucket structure, which the compiler needs to know, as you walk or insert into that structure, how far to advance in memory.

Hashmaps in other languages

Before we talk about the way Go implements a hashmap, I wanted to give a brief overview of how two popular languages implement hashmaps. I've chosen these languages as both offer a single map type that works across a variety of key and values.

C++

The first language we'll discuss is C++. The C++ Standard Template Library (STL) provides `std::unordered_map` which is usually implemented as a hashmap.

This is the declaration for `std::unordered_map`. It's a template, so the actual values of the parameters depend on how the template is instantiated.

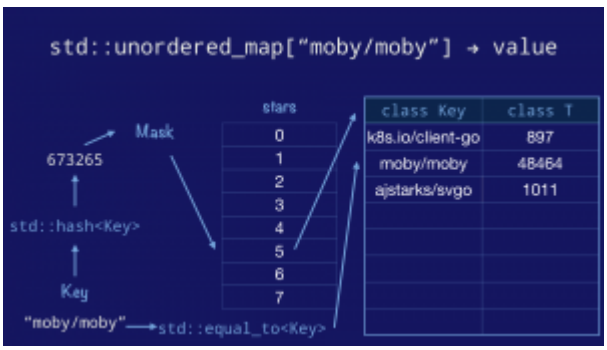
```
template<
    class Key,                                  // the type of
    class T,                                    // the type of
    class Hash = std::hash<Key>,                // the hash fu
    class KeyEqual = std::equal_to<Key>,        // the key equ

    class Allocator = std::allocator< std::pair<const Key
> class unordered_map;
```

There is a lot here, but the important things to take away are;

- The template takes the type of the key and value as parameters, so it knows their size.
- The template takes a `std::hash` function specialised on the key type, so it knows how to hash a key passed to it.
- And the template takes an `std::equal_to` function, also specialised on key type, so it knows how to compare two keys.

Now we know how the four properties of a hashmap are communicated to the compiler in C++'s `std::unordered_map`, let's look at how they work in practice.



First we take the key, pass it to the `std::hash` function to obtain the hash value of the key. We mask and index into the bucket array, then walk the entries in that bucket comparing the keys using the `std::equal_to` function.

Java

The second language we'll discuss is Java. In java the hashmap type is called, unsurprisingly, `java.util.Hashmap`.

In java, the `java.util.Hashmap` type can only operate on objects, which is fine because in Java almost everything is a subclass of `java.lang.Object`. As every object in Java descends from `java.lang.Object` they inherit, or override, a `hashCode` and an `equals` method.

However, you cannot directly store the eight primitive types; `boolean`, `int`, `short`, `long`, `byte`, `char`, `float`, and `double`, because they are not subclasss of `java.lang.Object`. You cannot use them as a key, you cannot store them as a value. To work around this limitation, those types are silently converted into objects representing their primitive values. This is known as *boxing*.

Putting this limitation to one side for the moment, let's look at how a lookup in Java's hashmap would operate.

First we take the key and call its `hashCode` method to obtain the hash value of the key. We mask and index into the bucket array, which in Java is a pointer to an `Entry`, which holds a key and value, and a pointer to the next `Entry` in the bucket forming a linked list of entries.

Tradeoffs

Now that we've seen how C++ and Java implement a Hashmap, let's compare their relative advantages and disadvantages.

C++ templated `std::unordered_map`

Advantages

- Size of the key and value types known at compile time.
- Data structure are always exactly the right size, no need for boxing or indiretion.
- As code is specialised at compile time, other compile time optimisations like inlining, constant folding, and dead code elimination, can come into play.

In a word, maps in C++ *can be* as fast as hand writing a custom map for each key/value combination, because that is what is happening.

Disadvantages

- Code bloat. Each different map are different types. For N map types in your source, you will have N copies of the map code in your binary.
- Compile time bloat. Due to the way header files and template work, each file that mentions a `std::unordered_map` the source code for that implementation has to be generated, compiled, and optimised.

Java util Hashmap

Advantages

- One implementation of a map that works for any subclass of java.util.Object. Only one copy of java.util.HashMap is compiled, and its referenced from every single class.

Disadvantages

- Everything must be an object, even things which are not objects, this means maps of primitive values must be converted to objects via boxing. This adds gc pressure for wrapper objects, and cache pressure because of additional pointer indirections (each object is effective another pointer lookup)
- Buckets are stored as linked lists, not sequential arrays. This leads to lots of pointer chasing while comparing objects.
- Hash and equality functions are left as an exercise to the author of the class. Incorrect hash and equals functions can slow down maps using those types, or worse, fail to implement the map behaviour.

Go's hashmap implementation

Now, let's talk about how the hashmap implementation in Go allows us to retain many of the benfits of the best map implementations we've seen, without paying for the disadvantages.

Just like C++ and just like Java, Go's hashmap written *in Go.* But–Go does not provide generic types, so how can we write a hashmap that works for (almost) any type, in Go?

Does the Go runtime use interface{}?

No, the Go runtime does not use `interface{}` to implement its hashmap. While we have the `container/{list,heap}` packages which do use the empty interface, the runtime's map implementation does not use `interface{}`.

Does the compiler use code generation?

No, there is only one copy of the map implementation in a Go binary. There is only one map implementation, and unlike Java, it doesn't use `interface{}` boxing. So, how does it work?

There are two parts to the answer, and they both involve co-operation between the compiler and the runtime.

Compile time rewriting

The first part of the answer is to understand that map lookups, insertion, and removal, are implemented in the runtime package. During compilation map operations are rewritten to calls to the runtime. eg.

```
v := m["key"]     → runtime.mapaccess1(m, ”key", &v)
v, ok := m["key"] → runtime.mapaccess2(m, ”key”, &v, &ok)
m["key"] = 9001   → runtime.mapinsert(m, ”key", 9001)
delete(m, "key")  → runtime.mapdelete(m, “key”)
```

It's also useful to note that the same thing happens with channels, but not with slices.

The reason for this is channels are complicated data types. Send, receive, and select have complex interactions with the scheduler so that's delegated to the runtime. By comparison slices are much simpler data structures, so the compiler natively handles operations like slice access, `len` and `cap` while deferring complicated cases in `copy` and `append` to the runtime.

Only one copy of the map code

Now we know that the compiler rewrites map operations to calls to the runtime. We also know that inside the runtime, because this is Go, there is only one function called `mapaccess`, one function called `mapaccess2`, and so on.

So, how can the compiler can rewrite this

```
v := m[“key"]
```

into this

```
runtime.mapaccess(m, ”key”, &v)
```

without using something like `interface{}`? The easiest way to explain how map types work in Go is to show you the actual signature of `runtime.mapaccess1`.

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer)
```

Let’s walk through the parameters.

- `key` is a pointer to the key, this is the value you provided as the key.
- `h` is a pointer to a `runtime.hmap` structure. `hmap` is the runtime’s hashmap structure that holds the buckets and other housekeeping values [1].
- `t` is a pointer to a `maptype`, which is odd.

Why do we need a `*maptype` if we already have a `*hmap`? `*maptype` is the special sauce that makes the generic `*hmap` work for (almost) any combination of key and value types. There is a `maptype` value for each unique map declaration in your program. There will be one that describes maps from `string`s to `int`s, from `string`s to `http.Header`s, and so on.

Rather than having, as C++ has, a complete map *implementation* for each unique map declaration, the Go compiler creates a `maptype` during compilation and uses that value when calling into the runtime’s map functions.

```
type maptype struct {
        typ           _type
        key          *_type
        elem         *_type
        bucket         *_type // internal type representin
        hmap           *_type // internal type representin
        keysize        uint8  // size of key slot
        indirectkey    bool   // store ptr to key instead
        valuesize      uint8  // size of value slot
        indirectvalue bool   // store ptr to value instea
        bucketsize     uint16 // size of bucket
        reflexivekey   bool   // true if k==k for all keys
        needkeyupdate bool   // true if we need to update
}
```

Each `maptype` contains details about properties of this kind of map from key to elem. It contains infomation about the key, and the elements. `maptype.key` contains information about the pointer to the key we were passed. We call these *type descriptors*.

```
type _type struct {
        size       uintptr
        ptrdata    uintptr // size of memory prefix holdi
        hash       uint32
        tflag      tflag
        align      uint8
        fieldalign uint8
        kind       uint8
        alg        *typeAlg
        // gcdata stores the GC type data for the garbage
        // If the KindGCProg bit is set in kind, gcdata i
        // Otherwise it is a ptrmask bitmap. See mbitmap.
        gcdata    *byte
        str       nameOff
        ptrToThis typeOff
}
```

In the `_type` type, we have things like it's size, which is important because we just have a pointer to the key value, but we need to know how large it is, what kind of a type it is; it is an integer, is it a struct, and so on. We also need to know how to compare values of this type and how to hash values of that type, and that is what the `_type.alg` field is for.

```
type typeAlg struct {
        // function for hashing objects of this type
        // (ptr to object, seed) -> hash
        hash func(unsafe.Pointer, uintptr) uintptr
        // function for comparing objects of this type
        // (ptr to object A, ptr to object B) -> ==?
        equal func(unsafe.Pointer, unsafe.Pointer) bool
}
```

There is one `typeAlg` value for each *type* in your Go program.

Putting it all together, here is the (slightly edited for clarity) `runtime.mapaccess1` function.

```
// mapaccess1 returns a pointer to h[key].  Never returns
// it will return a reference to the zero object for the
// the key is not in the map.
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer)
        if h == nil || h.count == 0 {
                return unsafe.Pointer(&zeroVal[0])
        }
        alg := t.key.alg
        hash := alg.hash(key, uintptr(h.hash0))
        m := bucketMask(h.B)
        b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bu
```

One thing to note is the `h.hash0` parameter passed into `alg.hash`. `h.hash0` is a random seed generated when the map is created. It is how the Go runtime avoids hash collisions.

Anyone can read the Go source code, so they could come up with a set of values which, using the hash ago that go uses, all hash to the same bucket. The seed value adds an amount of randomness to the hash function, providing some protection against collision attack.

Conclusion

I was inspired to give this presentation at GoCon because Go's map implementation is a delightful compromise between C++'s and Java's, taking most of the good without having to accomodate most of the bad.

Unlike Java, you can use scalar values like characters and integers without the overhead of boxing. Unlike C++, instead of *N* `runtime.hashmap`

implementations in the final binary, there are only *N* `runtime.maptype` *values, a* substantial saving in program space and compile time.

Now I want to be clear that I am not trying to tell you that Go should not have generics. My goal today was to describe the situation we have today in Go 1 and how the map type in Go works under the hood. The Go map implementation we have today is very fast and provides most of the benefits of templated types, without the downsides of code generation and compile time bloat.

I see this as a case study in design that deserves recognition.

1. You can read more about the runtime.hmap structure here, https://dave.cheney.net/2017/04/30/if-a-map-isnt-a-reference-variable-what-is-it

Related Posts:

1. **If a map isn't a reference variable, what is it? < https://dave.cheney.net/2017/04/30/if-a-map-isnt-a-reference-variable-what-is-it>**
2. **Are Go maps sensitive to data races ? < https://dave.cheney.net/2015/12/07/are-go-maps-sensitive-to-data-races>**
3. **Should Go 2.0 support generics? < https://dave.cheney.net/2017/07/22/should-go-2-0-support-generics>**
4. **Maybe adding generics to Go IS about syntax after all < https://dave.cheney.net/2018/09/03/maybe-adding-generics-to-go-is-about-syntax-after-all>**