Go is pass-by-value — but it might not always feel like it

29 August 2021 by Neil Alexander

Go is a programming language which passes by value, which effectively means that if you give a value as a parameter to a function, the received value within the function is actually a *copy* of the original. You can modify it however you wish and your changes will not affect the original value or escape the function scope. This is in contrast to some languages which pass values by reference instead of copying them.

Newcomers to Go, however, will quickly discover that it doesn't feel as though this is what is really happening in practice. You pass a map into a function only to find that if that function modifies the map, it gets modified everywhere. Worse, the program might just break in mysterious ways or panic altogether! What gives?

Go already exposes references to the programmer in the form of pointers — that is, using `*type` instead of `type` and then taking the reference of a variable using `&variable`. In this case, passing a pointer into a function is still *passing by value* in the strictest sense, but it's actually the pointer's value itself that is being copied, not the thing that the pointer refers to. This makes it possible to deliberately allow multiple functions, goroutines or sections of code to perform operations on the same variables.

Like many languages, Go has two classes of datatype: those with "value" semantics and those with "reference" semantics. Primitive types are all "value" types — those include the usual `int`, `string`, `byte` `rune`, `bool` types and so on. Pointer types are, in effect, "reference" types. However, maps, slices and channels are all special types that *are or contain references*.

Slices

A slice is a flexible reference to an array and has three fields: a pointer to the underlying memory, a "size" field which specifies how many elements of the underlying array are referenced, and finally, a "capacity" field which specifies the maximum size of the slice. The actual value contents of the slice are stored in the underlying memory in a "backing array", and the slice itself serves as a reference to this space.

As a result, copying a slice or passing it as a value doesn't actually copy the underlying array or the data within - it merely copies the pointer, size and capacity. Manipulating the

boundaries of a slice is therefore extremely cheap as this only requires modifying the pointer or size internally. It also means that you can have multiple slice references pointing to the same underlying array in memory, even if those slices are of different sizes!

In this case, the thing being passed-by-value is the *slice reference*, not the contents of the slice!

Note that the behaviour around `append` is influenced by the size and the capacity of the slice. If an `append` can be satisfied without the size exceeding the capacity, the new resulting slice returned by `append` will refer to the same underlying array. However, if the `append` would result in the new slice exceeding the capacity of the original, the underlying array is copied and the new resulting slice will refer to the copied array with a newly expanded capacity value. Therefore it is not necessarily safe to assume whether the slice returned by `append` will be a copy or not.

Maps

A map reference is actually a pointer to a map header struct in memory (`type hmap` internally). In the background, the Go compiler rewrites various map syntax to special functions within the `runtime` package which take a pointer to this struct. The header contains various fields, including the number of cells, a hash seed, flags and pointers to the top-level hashmap buckets. Maps can expand at runtime by allocating additional buckets and chaining them together.

As with slices, copying a map or passing it as a value doesn't actually copy the underlying hashmap — it merely copies the pointer to the header struct. Therefore, copying a map reference is just giving you multiple references to the same underlying map in memory.

Once again, it is the *map reference* that is being passed-by-value, not the contents of the map.

Channels

A channel reference is also a pointer to a channel header struct in memory (`type hchan` internally), similar to maps, and the Go compiler rewrites some of the syntactic sugar around channels to functions in the `runtime` package too. The header actually contains a pointer to a buffer array in memory, along with various offsets and size fields, a mutex and lists of active senders and receivers.

By now you'll be probably spotting the pattern — indeed copying the channel is just copying the pointer to the header struct, so you're really just ending up with more references to the same channel. Indeed this is necessary as you will want to be able to pass these references around so that you can send to or receive from the channel elsewhere.

The *channel reference* is being passed-by-value, not the queued values in the channel.

Note that channel references can be casted into and passed as “directed” references — in effect, your `chan type` can be converted into a send-only `chan<- type` or a receive-only `<-chan type` by copying the reference but assigning it a directed type. In this case, the same channel header is still used to access the channel internally and the direction-specific behaviour is actually enforced by the type system. The header itself is not copied.

Structs

So at this point, you might be wondering about structs. Strictly speaking, structs aren’t primitive types, but they also don’t contain built-in references in the same way that slices, maps and channels do. They are “value” types, however, and therefore passing a struct (rather than a pointer to a struct) means that the struct will be copied. This also means that all value types within the struct are copied — for example, `string` or `int` fields will have their values copied too. However, a struct can also contain slice, map or channel references. What happens in this case?

The *references themselves* will be copied, but *only* the references! Therefore, passing a struct as a value that contains a map, channel or slice (as opposed to passing a pointer to the struct) is not enough to guarantee that the receiving function can’t modify the contents of the original struct beyond it’s own scope — unless you specifically create a copy of the map or slice. The value-typed fields in the struct will be copied but maps and backing arrays won’t be.

Data races

When adding goroutines into the mix, things get especially complicated. Duplicating references and passing them to different goroutines can result in data races unless the correct synchronisation primitives (such as mutex locks) are used. Channels are the main exception to this rule as they are thread-safe by nature and are often used as a synchronisation primitive too.

Maps in Go are not safe for concurrent reads and writes. Trying to read a map in one goroutine at the same time as writing to it from another goroutine will result in a panic, therefore if you want to read *and* write a map from multiple goroutines currently, you must synchronise these operations, e.g. using `sync.RWMutex`. This is because the map header can be modified by a map write. (There’s actually a special `sync.Map` type for dealing with this problem, too!)

Slices behave differently again. Since each value of the underlying array has it’s own address in memory and are treated as distinct variables, reading one slice element while writing another is typically safe, but reading and writing the same element from different goroutines is absolutely not safe and will result in data races if those accesses are not synchronised.

Therefore it is very important to consider where slice, map or channel references are being duplicated to and what synchronisation is in place to ensure that data races cannot occur.

Takeaways

Of course, Go is still really passing-by-value in the truest sense of the term even with more complex types, it's just that the references are being passed as values instead of the contents. However, the actual resulting behaviour can feel unintuitive at first and can result in difficult-to-trace errors and bugs within a program. Even built-in functions like `append` have the potential to lead you astray if you make incorrect assumptions about whether or not a copy will take place.

So, if you are in any doubt and need to guarantee that a true copy is passed, always *copy the maps or slices yourself*. Be especially mindful that returning references allow a caller or consumer to mutate the referenced item, even in libraries or across package boundaries!