Golang Quirks & Tricks, Pt 2

A Programming Article by Efron Licht

Feb 2023

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

Go is generally considered a 'simple' language, but it has more edge cases and tricks than most might expect. In my last article, we covered intermediate topics, like declaration, control flow, and the type system. Now we're going to get into more advanced topics: concurrency, `unsafe`, and `reflect`.

By their nature, these articles are somewhat of a grab-bag without unifying theme, but the extremely positive response to the last one has convinced me they're worthwhile as a kind of whirlwind tour of more advanced topics.

As before, I'll link to the Go spec where appropriate. Most code examples link to a demonstration on the go playground.

1. generics

You can write a generic that neither takes or returns a value of it's type parameter. We did this in the previous article to 'tag' a zero-sized type with an associated type:

```
type contextKey[T any] struct{}
```

But it can be nice for making convenience wrappers around some of the functions in `reflect` and `unsafe`, too. I'll show an example, but first, a bit of background:

2. type inference, & interfaces

Go will *infer types* under some circumstances; most notably, go will happily convert a concrete type to an interface it satisfies: (like `int -> any` during `json.Marshal`), or an interface to a less restrictive interface (like `http.ResponseWriter -> io.Writer` during `fmt.Fprintf`) at a callsite or during an assignment.
This happens *before the function call*. This is why you can't ever get an `interface` type when you call `reflect.TypeOf`:

```
func main() {  // https://go.dev/play/p/io4pUcl2oiS

 for _, t := range []any{
  "",
  new(string),
  any(nil),
  io.Writer(new(bytes.Buffer)),
  io.Writer(io.Discard),
  (*io.Writer)(&io.Discard),
  (*any)(nil),
 } {
  fmt.Println(reflect.TypeOf(t))
 }
}
```

```
string
*string
<nil>
*bytes.Buffer
io.discard
*io.Writer
*interface {}
```

But wait: right here it seems like we have two interfaces: `io.Writer` and `interface{}`! Not quite. We have *pointers* to interfaces, which are *concrete* types. That's because a `(*io.Writer)` is converted to an `any` with concrete type `*io.Writer`, which is itself a pointer to a pointer to an io.Writer, which is *itself* a pointer.

Let's walk through step-by step

```
var buf bytes.Buffer
var bufp *bytes.Buffer = &buf // explicit:      1 ptr
var w io.Writer = &buf // implicit conversion to
`io.Writer<&bytes.Buffer>`; 2 ptrs
var pw *io.Writer = &w // explicit:  3 ptrs
var t reflect.Type = reflect.TypeOf(pw) // implicit
conversion to `any<*io.Writer>`: 4 ptrs
```

We can then use `reflect.Type`.Elem() to get the actual type we're looking for. Let's wrap it up in the promised generic:

```
// return the reflect.Type corresponding to the type
parameter.
// this is an easier way to get at interface types and never
allocates.
func typeOf[T any]() reflect.Type {
    return reflect.TypeOf((*T)(nil)).Elem()
}
func main() { // https://go.dev/play/p/qSqTgwVz6Wv
 fmt.Println(typeOf[io.Writer]())
}

```

Speaking of generics & type inference,

2.1. generic type inference

Go can occasionally partially infer the type of a generic function or struct. It works like this: starting from the leftmost type parameter, Go attempts inference type-by-type. As soon as it finds an ambiguity, it stops trying to infer types, leaving the rest up to the programmer. This can sound a little vague, so let's use an example:

Go can't type infer the following function at all, so any invocation has to fully spell out the type parameters:

```
func As1[FROM, TO any](f FROM) TO {
     return *(*TO)(unsafe.Pointer(&f)) // we'll convert
unsafe later in this article.
}

func main() { // https://go.dev/play/p/no9apPCbMAX
    b := As1[uint64, [8]byte](4)
    fmt.Println(b)
}
```

out: [4 0 0 0 0 0 0 0]

But by swapping the order of TO and FROM in the type parameters, we can partially type infer:

```

```

```
1  func As2[TO, FROM any](f FROM) TO { // https://go.dev/play/
   p/50PexocApZy
2   return *(*TO)(unsafe.Pointer(&f))
3  }
4  func main() {
5      b := As2[[8]byte](4)
6       fmt.Println(b)
7  }
```

This is usually good, but can sometimes cause difficulties when go's other type inference rules come into play. For example, the second program is *undefined behavior* on 32-bit architectures; since it's technically `b := As2[[8]byte, int]`.

It's worth noting that Go doesn't take the left-hand side of an expression into consideration for generic type interference. While we might expect this to compile, it doesn't:

```
1  var b [8]byte = As1(uint64(4))
```

```
compiler error: ./prog.go:10:20: cannot infer TO
(prog.go:12:9)
```

Careful ordering of your type parameters may make the difference between a pleasant API and an excruciating one.

3. select & channels

These points are an extension of dave cheney's article, 'channel axioms'

3.1. nil channels block forever

A nil channel cannot send or receive but blocks forever. This is usually a bug, but can be used to our advantage.

For example, we could combine three channels into one without favoring any channel for input: [^1]

[^1:] for the purpose of the following N examples, we're going to have exactly three channels, and we'll have their elements be ints.
a proper implementation should probably be generic, since this is easy to mess up and you'd rather do it once.
additionally, because go doesn't let you easily select from a variadic amount of channels at runtime, you generally have to write these functions for each arity (that is, number of different channels) you need. This is not too hard to do with code generation, since branches are identical. I hope to make that a subject of a later article.

```
1
```

```
// combine three channels one. the output channel is closed when all three inputs are.
func splice[T any](a, b, c <-chan T) chan T { // https://go.dev/play/p/ndtODuvmO2e
 dst := make(chan T)
 go spliceInto(dst, a, b, c)
 return dst
}


// splice the elements of a, b, and c into dst. no ordering is guaranteed, but this does not favor any input channel.
func spliceInto[T any](dst chan<- T, a, b, c <-chan T) {
LOOP:
 for a != nil || b != nil || c != nil {
  select {
  case t, ok := <-a:
   if !ok {
    a = nil
    continue LOOP
   }
   dst <- t
  case t, ok := <-b:
   if !ok {
    b = nil
    continue LOOP
   }
   dst <- t
  case t, ok := <-c:
   if !ok {
    c = nil
    continue LOOP
   }
   dst <- t
  }
 }
 close(dst)
}

```

Or balance inputs, getting exactly one element each from a, b, and c before moving on to the next 'round' of elements:

```

// feed one element each from a, b, and c each into the returned channel.
// the ordering of those elements within the rounds is not guaranteed.
func gatherRoundRobin(a, b, c <- chan int) (dst chan int) { // https://go.dev/play/p/0ftVD526ZQQ
 dst = make(chan int, 3)
 go func() {
  for {
```

```
 8       gatherRound(dst, a, b, c)
 9      }
10     }()
11     return dst
12    }
13
14    func gatherRound(dst chan <- int,a, b, c <- chan int) bool
      {
15     // note scope: since a, b, and c are copied when we call
      this function,
16     // nilling them out here doesn't affect the outer scope.
17     for a != nil || b != nil || c != nil {
18      var n int
19      select {
20      case n = <-a:
21       a = nil
22      case n = <-b:
23       b = nil
24      case n = <-c:
25       c = nil
26      }
27      dst <- n
28     }
29     return true
30    }
31
32    func main() {
33     a := make(chan int, 3)
34     b := make(chan int, 3)
35     c := make(chan int, 3)
36     for i := 1; i <= 3; i++ {
37      a <- i
38      b <- i * 10
39      c <- i * 100
40     }
41     ch := gatherRoundRobin(a, b, c)
42     for i := 0; i < 9; i++ {
43      fmt.Println(<-ch)
44     }
45    }
```

3.2. you can select on a send as well as a receive

This can be handy for dividing work equally among a number of potential workers.

> output: `100 010 001 020 200 002 003 030 300` (but other outputs may be possible.)

FOOTGUN WARNING:
Select contains a number of footguns.

- You can't check if a channel is closed without attempting to receive from it. This can quickly lead you to throw away data.
- Only the *leftmost channel* in a select statement is actually selected. In other words, this implementation of `gatherRound`, while pleasingly compact, quickly deadlocks:

```
 1  func gatherRoundBad(dst chan int, a, b, c chan int) bool
    { // https://go.dev/play/p/7w0N2NYNstA
 2   for a != nil || b != nil || c != nil {
 3    var n int
 4    select {
 5    case dst <- <-a:
 6     a = nil
 7    case dst <- <-b:
 8     b = nil
 9    case dst <- <-c:
10     c = nil
11    }
12    dst <- n
13   }
14   return true
15  }
```

```
fatal error: all goroutines are asleep - deadlock!
```

Digging into Go's spec, we find that

> For all the cases in the statement, the channel operands of receive operations and the channel and **right-hand-side expressions of send statements are evaluated exactly once, in source order, upon entering the "select" statement**. The result is a set of channels to receive from or send to, and the corresponding values to send. **Any side effects in that evaluation will occur irrespective of which (if any) communication operation is selected to proceed**. Expressions on the left-hand side of a RecvStmt with a short variable declaration or assignment are not yet evaluated.

Go makes it easy to spawn concurrent tasks, but managing them is difficult to get right. I've only scratched the surface here, and these examples are trivial and don't properly handle cancellation, etc. Since the advent of generics, a variety of structured concurrency libraries have been popping up. With luck, soon they'll be robust enough we don't have to do this kind of thing by hand.

3.3. channel conversions

While you can convert a bidirectional channel to a one-directional channel without issue:

```
1  func main() {
2   a := make(chan int) // https://go.dev/play/p/r3JTWjam0rX
3   var _ <-chan int = a
4   var _ chan<- int = a
5  }
```

The same is not true of data structures *containing* bidirectional channels, though such a transformation should always be safe, since a channel regardless of direction is just a pointer to a *runtime.hchan structure.

```
1  func main() { // https://go.dev/play/p/WSd4XO6AaSg
2   s := []chan int{}
3   var _ []<-chan int = s
4
5   var m = make(map[string]chan int)
6   var _ map[string]<-chan int = m
7  }
```

> compiler errors:
>
> ./prog.go:7:23: cannot use sliceOfChannels (variable of type `[]chan int`) as `[]<-chan int` value in variable declaration`
>
> ./prog.go:8:23: cannot use sliceOfChannels (variable of type `[]chan int`) as `[]chan<- int` value in variable declaration`

This is one of the few cases where we know better than the compiler. We can get around these restrictions via the `unsafe` package.

4. unsafe

The unsafe package lets you reinterpret memory as you see fit, subverting go's type system entirely.

Turn one type into another with the same size and alignment with the following transformation:

```
1  // return a shallow copy of the bytes of T as a B
2  func copyAs[B, T any](t T) (b B) {
3   return *(*B)(unsafe.Pointer(&t))
4  }
5
6  func asReceivers[T any](chans []chan T) []<-chan T {
7   return copyAs[[]<-chan T](chans)
8  }
```

```
 9
10  func asSenders[T any](chans []chan T) []chan<- T {
11   return copyAs[]chan<- T](chans)
12  }
```

This is *wildly* unsafe for a number of reasons, and this isn't the article to go into them. Still, it can be handy when *we* know two types are the same, but can't convince the compiler.

The following transformations ARE safe, and occasionally useful:

FROM	TO	BIDIRECTIONAL
`uintN`	`[N/8]byte`	✅
`[M]uintN`	`[M*(N/8)]byte`	✅
`[]chan T`	`[]<- chan T`	❌
`[]chan T`	`[]chan <- T`	❌
`map[K] chan T`	`map[K]<- chan T`	❌
`map[K] chan T`	`map[K] chan <- T`	❌

Note that these transformations work on ARRAYS, not slices.

4.1. unsafe slice transformations

Transforming slices isn't too bad, though: (If you're not familiar with the internals of slices, see this article on the go blog or this follow-up by dave cheney first.)

Because slice lengths are known at *runtime*, we'll have to do a little bit of math, then generate the slice ourselves via the underlying pointer:

```
 1
 2  func main() { // https://go.dev/play/p/c2dyEAD9aD-
 3   // create a slice of uint16s...
 4   sixteen := []uint16{0x0123, 0x3456}
 5   fmt.Printf("%T (before): %#04x\n", sixteen, sixteen)
 6   eight := SliceAs[uint8](sixteen) // and reinterpret them
    as uint8s
 7   fmt.Printf("%T (before): %#02x\n", eight, eight)
 8   // since they share the same array, a change to one...
 9   eight[1] = 0xA
10
11   // is reflected in the other. note that most, but not all
    modern architectures are little-endian.
12   fmt.Printf("%T (after): %#04x\n", sixteen, sixteen)
13  }
14
15  func SliceAs[B, T any](t []T) (b []B) {
16   n := size[T]() * len(t)
17
18   sizeB := size[B]()
```

```
19    if n%size[B]() != 0 {
20     panic(fmt.Errorf("can't convert %T to %T: out of
      bounds", t, b))
21    }
22    newLen := n / sizeB
23    return unsafe.Slice((*B)
      (unsafe.Pointer(unsafe.SliceData(t))), newLen)
24   }
25   func size[T any]() int {
26    var t T
27    return int(unsafe.Sizeof(t))
28   }
```

output:

```
1   []uint16 (before): [0x0123 0x3456]
2   []uint8 (before): 0x23015634
3   []uint16 (after): [0x0a23 0x3456]
```

In general, the unsafe package is best avoided in production code, but sometimes you actually *do* know better than the compiler. I encourage my readers to play around with the `unsafe` package on their own time to gain an intuition about how Go actually lays things out in memory. Make sure to read the package and spec documentation carefully.

I hope this was helpful! I think this is the end of this series; I'm planning to do some deeper dives next time.

MORE ARTICLES