go quirks & tricks 3

a programming article by Efron Licht

july 2023

Go is generally considered a 'simple' language, but it has more edge cases and tricks than most might expect. This is the third in a series of articles about intermediate-to-advanced go programming techniques. In part 1, we covered unusual parts of declaration, control flow, and the type system. In part 2, we touched concurrency, `unsafe`, and `reflect`. Here in part 3, we'll mostly talk about arrays, validation, and build constraints.

1. arrays can be initialized using a map-like syntax

You can initialize arrays or slices kind of like a map: omitted values will be the zero value.

```
1   var smallPrimes = [20]bool {
2       2: true,
3       3: true,
4       5: true,
5       7: true,
6       11: true,
7       13: true,
8       17: true,
9       19: true,
10  }
```

You can technically mix-and-match positional and map-like initialization:

```
// equivalent to the above. don't do this.
var smallPrimes = [40]bool {
    false, false, true, false, true,
    7: true,
    11: true,
    13: true,
    17: true,
    false, // 18
    true // 19
}
```

But this is unintuitive and overly clever.

While this seems handy for small lookup tables, it really comes into it's own when used with enum-like constants generated with `iota`.

1.1. iota / enums / map-like array initialization

To briefly review, `iota` can be used to generate a sequence of numbers, usually for enum-like constants or bitflags. Using my game, Tactical Tapir, as an example:

```
type GunKind int16
const (
    UNARMED GunKind = iota
    PISTOL // equivalent to PISTOL GunKind = 1
    RIFLE // equivalent to RIFLE GunKind = 2
    SHOTGUN // equivalent to SHOTGUN GunKind = 3
    // other gun kinds omitted for this article
)
```

This is handy, but may seem of questionable utility: there's nothing stopping us from just writing in the values ourselves.

The value of `iota` becomes apparent when we want to use the enum as an index into an array or slice. If we add a 'length proxy' dummy constant at the end of the enum:

```
    UNARMED GunKind = iota
    PISTOL // equivalent to PISTOL GunKind = 1
    RIFLE // equivalent to RIFLE GunKind = 2
    SHOTGUN // equivalent to SHOTGUN GunKind = 3
    GUNKIND_N // "length proxy" constant
```

We can use it to size arrays.

Many small lookup tables can be compactly represented as arrays of the enum type and sized with the length proxy:his:

```
var names = [GUN_KIND_N]string{
    UNARMED: "unarmed",
    PISTOL: "pistol",
    RIFLE: "rifle",
```

```
    SHOTGUN: "shotgun",
}
```

Using the map-like initialization allows us to re-order the constants without introducing bugs, and will introduce an `undefined` compilation error if we remove an enum variant.. Unfortunately, though, *added* values will be initialized to the zero value, which may not be what we want - we'd like to error if we forget to add a value to the array. We'll talk about that in a second.

This approach (arrays of primitives, indexed with enums) may seem unusual to programmers used to large structs: after all, why not just do:

```
type Gun struct {
    name string
    startingAmmo int16
    maxAmmo int16
    // other fields omitted
}
```

Or even:

```
func (gk GunKind) Name() string {
    switch gk {
        case UNARMED:
            return "unarmed"
        case PISTOL:
            return "pistol"
        // more omitted
    }
}
```

Or just use a map:

```
var names = map[GunKind]string{
    UNARMED: "unarmed",
    PISTOL: "pistol",
    RIFLE: "rifle",
    SHOTGUN: "shotgun",
}
```

All four approaches have their place. Here's a quick summary:

approach	pros	cons	suggested use case
arrays	fast, compact, cache-friendly	can 'forget' to update individual arrays	game dev, systems programming
func/ switch	properly read-only, doc comments lead to discoverable API	verbose, can be slow	library code
map	simple, useful for sparse tables		sparse tables

approach	pros	cons	suggested use case
		larger, can forget to update, map lookups take at least 1 indirection	
structs	simple, self-documenting	cache-unfriendly, 'action at a distance'	rarely

To go into more detail on performance:

- Compilers can more effectively vectorize operations on arrays of primitives than arrays of structs.
- Only loading 'what we need' from memory is much friendlier on the cache: no need fill a cache line w/ `name` & `startingAmmo` if we only need `maxAmmo` right now, for instance.
- The compiler (and the microarchitecture) can reason about conflicts between memory accesses more easily, since we aren't going to 'touch' other parts of the struct.

I find the array approach to be extremely useful for *application code*, especially bits like game development where backwards compatibility is less of a concern and performance is critical.

1.2. generic identity functions

An "identity function" is a function that returns it's argument. The simplest example is this:

```
1  // ID returns it's argument.
2  func ID[T any](t T) T { return t }
```

While this may seem useless at first glance, they're quite handy in practice. You can use identity functions to log properties of a value as it's initialized or returned from a function:

```
1  func Logged[T any](t T) T {
2      _, file, line, _ := runtime.Caller(1) // 1 = caller of
   Logged
3      fmt.Printf("%s:%d: %T: %v\n", file, line, t, t)
4      return t
5  }
```

Or to assert properties of a value as it's initialized. In particular, **combined with the enum/array lookup table approach mentioned earlier**, we can validate that all enum variants have a value a lookup table:

```
1  func MustNonZero[T any](a T) T {
2      assertNonZero(a)
```

```
    return a
}
func assertNonZero(v any) {
switch v := reflect.ValueOf(v); v.Kind() {
    default:
        panic("not an array or slice")
    case reflect.Array, reflect.Slice:
        for i := 0; i < v.Len(); i++ {
            if v.Index(i).IsZero() {
                // runtime.Caller(skip) gets information about the caller(s) of the current function by ascending the callstack.
                // we want the file:line to be where MustNonZero was called, that is, the caller of the caller of assertNonZero.
                // skip=0 -> assertNonZero
                // skip=1 -> MustNonZero
                // skip=2 -> caller of MustNonZero: what we want
                _, f, line, _ := runtime.Caller(2)
                panic(fmt.Sprintf("%s:%d: %s[%d] unexpectedly zero", f, line, v.Type(), i))
            }
        }
}
```

And we'll get an immediate runtime error if we forget to fill in a value, complete with the file:line where we initialized the array. **Most IDEs recognize the file:line format and will let you command-click on it to jump to the offending line**. Knowing we missed a variant at compile time is much better than finding out at runtime.

Let's demonstrate. Suppose we forget to initialize `UNARMED` in our array:

```
// https://go.dev/play/p/Du-AaY-L4mR
var names = MustNonZero([GunKindN]string{
    PISTOL: "pistol",
    RIFLE: "rifle",
    SHOTGUN: "shotgun",
})
```

```
panic: /tmp/sandbox1262377932/prog.go:19: [4]string[0] unexpectedly zero
```

Obviously, you could create tests for each array/slice, but that's both a lot of boilerplate and rather distant from where it's defined. I find this approach to be simpler and more direct.

Especially large arrays or slices may benefit from doing the validation off-thread:

```
func MustNonZeroOffThread[T any](a T) T {
    go assertNonZero(a)
    return a
}
```

But this is generally overkill: the validation is fast enough that it's not worth the extra complexity.

This is a place where *specifically using arrays* is useful: the length of an array is known at compile-time, and that length is defined automatically to be the number of enum variants using `iota`. Thus, *we always ensure that we have one entry per enum variant*, regardless of how many variants we have. Make sure to use the length proxy constant as the length of the array rather than using the variadic syntax `var MaxAmmo = [...]int16{...}` to ensure this invariant is maintained.

2. 'compile-time' build tag specialization using ordinary control flow

Build constraints are handy for compile-time specialization, but can be clunky to use, since the different versions of each item must be in completely separate files. As an example, suppose we want to spawn GUI windows. We'd need to have separate files for each platform we want to support:

```
1  //go:build windows
2  package mypkg // in mypkg_windows.go
3
4  func createWindow(){
5      // implementation goes here
6  }
```

```
1  //go:build linux && wayland
2
3  package mypkg // in mypkg_linux_wayland.go
4  func createWindow(){
5      // implementation goes here
6  }
```

And each function would need to be redefined in each file (and it's comments duplicated, etc). This can be hard to work with!

A neat trick is to instead define a set of constants that correspond to the build constraints, and switch between implementations at 'runtime' instead using standard control flow like `if` or `switch`. Go's compiler is smart enough to optimize out completely unreachable branches along the lines of `if false`.

The go stdlib uses this approach in the `math` package to switch between hardware and software implementations of `sin` and `cos`, among others.

math/stubs.go:

```
1  const haveArchSin = false
```

math/sin.go

```
func Sin(x float64) float64 {
 if haveArchSin { // set by build constraint
  return archSin(x)
 }
 return sin(x)
}
```

There is no performance penalty for this approach. The Go compiler will happily prune unreachable branches under almost all circumstances.

Any performance claim requires evidence. Let's write a small program to demonstrate this behavior (& build tags / build constraints in general).

We'll have three files: `a.go`, `not_a.go`, and `main.go`, that look like this:

2.0.1. `main.go`

```
package main
import "fmt"
// a is not defined here: it's in either a.go or not_a.go
func main() {
    if a {
        fmt.Println("a")
    } else {
        fmt.Println("!a")
    }
}
```

2.0.2. `a.go`

```
//go:build a
package main
const a = true
```

2.0.3. `not_a.go`

```
//go:build a
package main
const a = false
```

Let's build the project both ways by adding the relevant build tags:

```
go build -tags a -o output_a
go build -o output_not_a
```

And dump the assembler using the objdump tool:

```
go tool objdump -S output_a > dis_a
go tool objdump -S output_not_a > dis_not_a
```

The generated assembly can be a bit hard to read, so let's just search for something resembling `fmt.Println` using ripgrep (regular grep would work fine too)

IN:

```
cat dis_a | rg fmt.Println
```

OUT:

```
fmt.Println("a")
```

IN:

```
cat dis_not_a | rg fmt.Println
```

OUT:

```
fmt.Println("!a")
```

As we can see, the branch that would call `fmt.Println("a")` is completely absent from the `dis_not_a` output, and vice versa. Let's run a diff to take a closer look:

Running a diff:

IN

```
diff --side-by-side dis_a dis_not_a
```

OUT

```
// many lines omitted from disassembler of identical (except addresses output)

func main()                                                    func main() {
  0x481060              493b6610                CMPQ 0x10(R14     0x481060              493b6610                CMPQ 0x10(R14
  0x481064              7656                    JBE 0x4810bc      0x481064              7656                    JBE 0x4810bc
  0x481066              4883ec40                SUBQ $0x40, S     0x481066              4883ec40                SUBQ $0x40, S
  0x48106a              48896c2438              MOVQ BP, 0x38     0x48106a              48896c2438              MOVQ BP, 0x38
  0x48106f              488d6c2438              LEAQ 0x38(SP)     0x48106f              488d6c2438              LEAQ 0x38(SP)
```

```
 9
    fmt.Println("a")
    |                   fmt.Println("!a")
10    0x481074              440f117c2428            MOVUPS
    X15, 0      0x481074              440f117c2428
    MOVUPS X15, 0
11    0x48107a              488d155f830000          LEAQ
    0x835f(I      0x48107a              488d155f830000
    LEAQ 0x835f(I
12    0x481081              4889542428              MOVQ DX,
    0x28      0x481081              4889542428
    MOVQ DX, 0x28
13    0x481086              488d1533610300          LEAQ
    0x36133(      0x481086              488d1533610300
    LEAQ 0x36133(
14    0x48108d              4889542430              MOVQ DX,
    0x30      0x48108d              4889542430
    MOVQ DX, 0x30
15          return Fprintln(os.Stdout,
    a...)                                   return
    Fprintln(os.Stdout, a...)
16    0x481092              488b1d77af0a00          MOVQ
    os.Stdou |    0x481092              488b1d57af0a00
    MOVQ os.Stdou
17    0x481099              488d05a8650300          LEAQ
    go:itab.      0x481099              488d05a8650300
    LEAQ go:itab.
18    0x4810a0              488d4c2428              LEAQ
    0x28(SP)      0x4810a0              488d4c2428
    LEAQ 0x28(SP)
19    0x4810a5              bf01000000              MOVL $0x1,
    DI      0x4810a5              bf01000000              MOVL
    $0x1, DI
```

You may note *slightly* different addresses, but the code is otherwise identical (except for the `fmt.Println` call, of course). Even whitespace changes will generate *this* kind of difference, though: these are 'identical enough'.

3. Conclusion

Hope you find some of these techniques handy. I've been using most of these pretty heavily while working on my game.

If you liked this article, you may enjoy my series on **starting software**:

MORE ARTICLES