Golang Quirks & Intermediate Tricks, Pt 1: Declarations, Control Flow, & Typesystem

A programming article by Efron Licht

Feb 2023

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

- Golang Quirks & Intermediate Tricks, Pt 1: Declarations, Control Flow, & Typesystem
  - 1. multi-statement lines with semicolons
  - 2. methods as functions (“method expressions”)
  - 3. `select` statements have `break`
  - 4. go can infer the type of composite literals in some contexts, but not others
  - 5. simple expressions in switch statements
  - 6. GOTO exists
  - 7. you can make a block at any time
  - 8. immediately-evaluated-function-expressions
  - 9. you can declare types inside blocks
  - 10. go has anonymous structs
  - 11. … and anonymous interfaces
  - 12. zero-sized type (“ZST”)
  - 13. blank struct fields
    - 13.2. Next time(?)

Go is generally considered a ‘simple’ language, but it has more edge cases and tricks than most might expect.

You can be a productive go programmer without knowing about or using most or any of these tricks, but some of them are pretty handy. I'll link to the go spec where appropriate throughout the article.

This is part 1 of ~~what I hope to be~~ a continuing series.

1. multi-statement lines with semicolons

Go is secretly a C-like language that terminates statements with semicolons. The semicolons are actually *inserted* early in compilation (during lexing). This means you can put multiple statements on the same line by inserting semicolons!

Be warned: `gofmt` will usually break them up into multiple lines. In fact, you can *never* have a single-line conditional. (Sorry, ternary conditional fans.)

Still, it can be handy for *really* small two-statement functions, like in tests:

```
1   func asJson(v any) []byte {b, _ := json.Marshal(v); return
    b}
```

2. methods as functions ("method expressions")

Go methods are just functions. Given a type and method:

```
1   type Point struct{X, Y float64}
2   func (p Point) Add(q Point) Point{
3       return Point{X: p.X+q.X, Y: p.Y+q.Y}
4   }
```

You can call the method in the 'usual' way by providing a receiver and using `receiver.funcName(arg1, arg2, ...)`

```
1   p, q := Point{1, 1}, Point{2,2}
2   fmt.Println(p.Add(q))
```

out: `{3 3}`

Or you can use the method as an ordinary, "bare" function via `typeName.funcName(arg0, arg1, arg2)`

```
1   p, q := Point{1, 1}, Point(2,2);
2   fmt.Println(Point.Add(p, q))
```

out: `{3, 3}`

This is called a method expression. Unlike method calls, a method *expression* won't automatically reference or de-reference a receiver for you, since there *is* no receiver.

That is, while **this** code compiles fine

```
import "math/big" // https://go.dev/play/p/-CHMNxIKumy
func main() {
 var x big.Float
 x.SetFloat64(10)
}
```

This code gives a compiler error

```
import "math/big" // https://go.dev/play/p/cv8TSURe15J
func main() {
 var x big.Float
 big.Float.SetFloat64(x, 10) // wrong
}

```

compiler error: `invalid method expression big.Float.SetFloat64 (needs pointer receiver (*big.Float).SetFloat64)`

The proper method expression is as follows

```
import "math/big" // https://go.dev/play/p/SRYxpp1UdVJ
func main() {
 var x big.Float
 (*big.Float).SetFloat64(&x, 10) // note parens
}
```

Method expressions don't come up often, but they can occasionally save some work when you're sorting or deduping.

3. `select` statements have `break`

`select` has no `continue`, but it *does* have `break`. This can lead to nasty bugs if you're trying to break out of, say, an enclosing `switch` or loop. Use labels instead, as demonstrated by this code in the un-exported `filelock` package in go's stdlib:

```
 // Wait until process Q has either failed or locked file B.
 // Otherwise, P.2 might not block on file B as intended.
locked:
 for {
  if _, err := os.Stat(filepath.Join(dir, "locked")); !os.IsNotExist(err) {
   break locked
  }
  select {
  case <-qDone:
   break locked
  case <-time.After(1 * time.Millisecond):
```

```
12      }
13     }
```

4. go can infer the type of composite literals in some contexts, but not others

The following code playground gives a terse and unhelpful compiler error:

```
1
2   func main() { // https://go.dev/play/p/CLu4AXg5qYW
3    type Q struct{ A, B [3]int }
4    structOfArrays := Q{{}, {}}
5    fmt.Println(structOfArrays)
6   }
```

compiler error: `missing type in composite literal`

This implies that you always need to provide the types of composite literals, but that's just not true. Go is happy to compile the following bode without me spelling out the type of each item on the right-hand side:

```
1   func main() { // https://go.dev/play/p/CLu4AXg5qYW
2    type S struct{ N, M int }
3    arrayOfStructs := [3]S{{}, {}, {0, 1}}
4    fmt.Println(arrayOfStructs)
5   }
```

out: `[{0 0} {0 0} {0 1}]`

Or even this monstrosity:

```
1   func main() { // https://go.dev/play/p/kXLR8n7WdMc
2    sliceOfMapOfArrayOfStructs := []map[string][2]struct{ N, M
    int }{{"foo": {{}, {M: 2}}}}
3    fmt.Printf("%+v\n", sliceOfMapOfArrayOfStructs)
4   }
5
```

out: `[map[foo:[{N:0 M:0} {N:0 M:2}]]]`

The actual rule is this: go will infer the types of composite literals if they're contained within an **array**, **map**, or **slice**, but struct fields and function arguments always need to spelled out explicitly.

There's a long-open issue (#12584) hoping to address this inconsistency. I'd love to see more permissive composite literals.

5. simple expressions in switch statements

A switch statement can be proceeded by a simple statement:

```
// b is a *math/big.Int*
switch n, acc := b.Uint64(); acc {
    case big.Below:
        fmt.Println("< 0"),
    case big.Above:
        fmt.Println("> 18446744073709551615")
    case big.Exact:
        fmt.Println(n)
}
```

This works for expression switches and type switches:

```
switch a, err := f(); err.(type) {
}
```

If you omit the *second* part of the switch, you can do a "normal" boolean-value switch statement:

```
// some kind of low-level networking call:
var try int
var packets []Packet
READ:
for {
   switch packet, err := readPacket(ctx, conn, buf);  { // note semicolon
        case errors.Is(err, io.EOF):
            packets = append(packets, packet)
            break READ
        case err == nil:
            packets = append(packets, packet)
            try = 0
        case errors.As(err, fatalErr) || try == maxTries:
            return fmt.Errorf("fatal error after %d retries: %v", i, err)
        default:
            const wait = 100*time.Millisecond
            log.Printf("error: retrying in %d", wait)
            try++
            time.Sleep(wait)
   }
}

```

I like the look of these: they allow very terse, expressive code, but they're rare & unusual enough to probably cause confusion. Most of the time you're better off with a chain of `if`.

6. GOTO exists

The oft-maligned GOTO is an excellent piece of kit. Go's GOTO is somewhat limited: you can't jump into a new block or out of a function, so it's hard to get yourself into the kind of trouble you could in 1980s BASIC.

This means you can't do something like this, since you'll get a compiler error:

```
1
2  func main() { // https://go.dev/play/p/1krGFE6FvgJ
3   goto label
4
5   if true {
6    v := 3
7    panic(v)
8
9   label:
10   fmt.Println(v) // what's the value of v?
11  }
12 }
13
14
```

```
compiler error: ./prog.go:11:7: goto label jumps into block
starting at ./prog.go:13:10
```

Speaking of which:

7. you can make a block at any time

You don't need an `if`, `for`, `func`, or any other keyword to make a block.

```
1  {
2      name := "efron"
3      {
4          fmt.Println("hi ", name)
5      }
6  }
7
```

I find this useful for complicated variable initialization. Here's an example from the `fmtbench` tool I wrote in the last article

```
1  // fmtbench.go
2  // context: the variable sortBy is a command-line flag
   specifying the sort order. we've already validated it.
3  // results is a []struct{
4  //     name                    string,
5  //  runs, ns, bytes, allocs float64
6  // }
```

```
 7  {
 8   // sort results
 9    var less func(i, j int) bool
10    switch *sortBy {
11    default:
12      goto PRINT
13    case "allocs":
14     less = func(i, j int) bool { return results[i].allocs <
    results[j].allocs }
15    case "name":
16     less = func(i, j int) bool { return results[i].name <
    results[j].name }
17    case "runtime":
18     less = func(i, j int) bool { return results[i].ns <
    results[j].ns }
19    }
20    sort.Slice(results, less)
21   }
22  PRINT:
23   for _, res := range results {
24    fmt.Printf("|%s|%.3g|%.3g|%0.3g|%.3g|%0.3g|%.3g|%0.3g|
    \n", res.name, res.runs, res.ns, (res.ns/maxNS)*100,
    res.bytes, (res.bytes/maxBytes)*100, res.allocs,
    (res.allocs/maxAllocs)*100)
25   }
```

By using a block here, we make it immediately clear that `less` is only going to exist for the context of this sort

We *could* make a function for this, but that means jumping around, for us, the compiler, and the runtime (assuming it's not inlined).

Try starting with blocks, and promote them to functions when you find yourself needing to re-use the code.

But sometimes you do need a function, even for a single use:

8. immediately-evaluated-function-expressions

You can define a function and invoke it on the same line:

```
 1  // playground: https://go.dev/play/p/dmNloKFUGSZ
 2  package main
 3
 4  import (
 5   "crypto/rand"
 6   "encoding/binary"
 7   "fmt"
 8  )
 9
10  var seed uint64 = func() uint64 {
```

```
11    var b = make([]byte, 8)
12    _, _ = rand.Read(b)
13    return binary.LittleEndian.Uint64(b)
14   }()
15
16   func main() {
17    fmt.Println(seed)
18   }
```

This is the catchily-named "immediately-evaluated-function-expression", or IIFE for short. These are invaluable in languages which privilege functions over other kinds of blocks: for example, Javascript before it got the `let` keyword had no block scope, so you had to define functions every time you wanted a new namespace.

Go privileges functions over blocks in two ways:

- you can evaluate a function during variable declarations in the global namespace (that is, before `main()` or even `init()`, as shown in the example above.
- a `defer`-ed function evaluates at the end of the enclosing *function* scope.

There's basically only two uses for IIFE's instead of blocks:

- for complex variable initialization in the global scope in a more natural way than using `init()`.
- if you need a function scope to use `defer()` or `recover()`, since

If you're going to use an IIFE with a return value, use the `var` declaration instead of `:=` - it makes it easier for the reader to understand the flow. And don't overdo it - you can always just define a closure and call it on the next line.

9. you can declare types inside blocks

You can declare types inside any kind of block, but you can't declare methods on those types.
You *can* define a function that takes that type using a function expression ("closure").

```
1   import "fmt"
2   func main() { // playground: https://go.dev/play/p/
    vAkgOTnEg7d
3    type Point struct{ X, Y float64 }
4    addPoint := func(p, q Point) Point { return Point{p.X +
    q.X, p.Y + q.Y} }
5    q := addPoint(Point{2, 3}, Point{-1, 1})
6    fmt.Println(q)
7   }
8
```

output: `{1 4}`

This obeys the ordinary block-scope rules, so this would be a compiler error:

```
1  func main() { // https://go.dev/play/p/_ytvmPewLTA
2      {
3          type Point struct{X, Y float64}
4      }
5      var p Point
6  }
```

compiler error: `./prog.go:9:8: undefined: Point`

This can make your code more straightforward. Just like variables, it's best to define a type as close to it's use and with as small of a scope as possible.

10. go has anonymous structs

Sometimes you don't have to declare the type at all: go allows anonymous struct values. This is especially handy for functions like `json.Marshal` and `json.Unmarshal` which just depend on the *shape* of the type.

These anonymous structs can *nest*:

```
1  func main() { // https://go.dev/play/p/vA5SJ-GKJMm
2   var s struct{ Name struct{ First, Last string } }
3   json.Unmarshal([]byte(`{"name": {"first": "efron", "last":
   "licht"}}`), &s)
4   fmt.Printf("%+v\n", s)
5  }
```

output: `{Name:{First:efron Last:licht}}`

You can even make custom struct tags for your individual use case:

```
 1  // GET /health
 2  import "json"
 3  import "net/http"
 4  func WriteHealth(w http.ResponseWriter, _ *http.Request) {
 5      json.NewEncoder(w).Encode(struct {
 6          Uptime time.Duration `json:"uptime"`
 7          Stats struct {
 8              Hits int64 `json:"hits"`
 9              Misses int64 `json:"misses"`
10              Errors int64 `json:"errors"`
11          } `json:"stats"`
12      })
13  }
```

11. … and anonymous interfaces

You *never* have to declare the type of an interface: anywhere you can use `io.Writer`, you can use `interface{Write([]byte)(int, error)}`.

This can be handy for runtime specialization (that is, when you want to check if a type fulfills extra interfaces)

```
import "gzip"
func writeZipped(w io.Writer, b []byte) (int, error) {
    zipw := gzip.NewWriter(w)
    n, err := zipw.Write(w)
    if err != nil {
        return n, err
    }
    if err := zipw.Close(); err != nil {
        return err
    }
    // flush the underlying buffer, if there is one
    if f, ok := w.(interface{Flush() error}); ok {
        _ = f.Flush()
    }
    // sync to disk if possible
    if f, ok := w.(interface{Sync() error}); ok {
        _ = f.Sync()
    }
}
```

This is especially useful for function signatures. Suppose I'm going to call out to a database as part of a function.

(An aside: I don't particularly like mocking: I'd love to write an article about strategies you can use to avoid it).

The 'ordinary' function signature would look something like this:

```
func SelectUser(ctx context.Context, db *sql.DB, userID
uuid.UUID) (username string, createdAt time.Time, err
error) {
    const query = `SELECT username, created_at FROM users
where user.id = $1;`
    db.QueryRowContext(ctx, query, userID).Scan(&username,
createdAt)
}
```

Suppose at some point we need to mock this for a test. *sql.DB is a struct, and it's not immediately apparent what we'd call the interface we'd replace it with. `DBer`? `QueryRowContexter`? In this case, we can be clearest by omitting the name entirely: all the reader needs to know is that the DB has a function that looks like QueryRowContext().

We can make this mockable by just changing the function signature to use an *anonymous interface*.

```
func SelectUser(
    ctx context.Context,
```

```
3       db interface{QueryRowContext(context.Context,
    string, ...args) *sql.Row},
4       userID uuid.UUID
5   ) (username string, createdAt time.Time, err error) {
```

I think the anonymous interface is actually *clearer* than the named one for most single-method interfaces.

Both anonymous structs and interfaces can be generic, too.

12. zero-sized type ("ZST")

The empty struct `struct{}` and arrays of length zero (like `[0]int`) take up no memory, as do structs and arrays comprised entirely of zero-sized types.

A zero-sized type ("ZST") is most often used as an interface handle, like `io.Discard`.

```
1
2   ## zero-sized types
3   // io/io.go
4
5   // Discard is a Writer on which all Write calls succeed
6   // without doing anything.
7   var Discard Writer = discard{}
8   type discard struct{}
9
10  func (discard) Write(p []byte) (int, error) { return
    len(p), nil}
11  func (discard) WriteString(s string) (int, error) { return
    len(s), nil}
```

You can also use a ZST as a map value type to save space rather than using `map[string]bool`

```
1   var set = make(map[string]struct{})
```

but *don't*: `map[string]bool` is just as fast and has a much cleaner api.

You can get kind of silly with this:

```
1   type cursedZST = [0]map[struct{}]struct{} // don't do this.
```

Zero-sized types have a third use, but we'll need to talk about blank struct fields first.

13. blank struct fields

Struct types can have unreachable fields using the blank identifier, _ as the field name. You can use blank fields:

- To pad a struct to a specific size or alignment.

This is occasionally handy for cool `unsafe` stuff like serializing or deserializing stuff straight from a bytestream.

```
1      func main() { // https://go.dev/play/p/4H7V_kKDw5m
2      type Point struct{ X, Y, Z uint16 }
3      type PaddedPoint struct {
4       X, Y, Z uint16
5       _       uint16
6      }
7      const format = "%12v\t%v\t%v\n"
8      fmt.Printf(format, "type", "size", "align")
9      fmt.Printf(format, "Point", unsafe.Sizeof(Point{}),
    unsafe.Alignof(Point{}))
10     fmt.Printf(format, "PaddedPoint",
    unsafe.Sizeof(PaddedPoint{}),
    unsafe.Alignof(PaddedPoint{}))
11     }
```

```
1            type size align
2            Point    6     2
3      PaddedPoint  8     2
```

- A blank field makes it difficult to initialize a struct without specifying key names. So as not to waste space, use a `zero-sized type` for this.

  This means that if you add a field to the struct later, it's not a breaking change for users.

  ```
  1    type LogOptions struct {
  2        _ [0]int
  3        Level int8
  4        LogTime, LogFile, LogLine bool
  5    }
  ```

  Be careful with this: sometimes you *want* changes to the API to be breaking changes, and changing the size of commonly-used types can have unforeseen performance ramifications.

  > **WEIRD EDGE CASE WARNING: A ZERO-SIZED TYPE IS ONLY ZERO-SIZED IF IT'S NOT THE FINAL MEMBER OF THE STRUCT.**

  That is, do this:

  ```
  1    > type s struct {
  2    >         _ [0]func()
  ```

```
>         a int
> }
> ```
>
> And not this:
>
> ```go
> type s struct {
>     a int
>     _ [0]func()
> }
> ```
>
> See [issue 58483](https://github.com/golang/go/issues/58483). I found this out in a response to this article!

      Blank fields should be used sparingly, but can be nice for configuration.

- Adding a field of uncomparable type makes the entire struct uncomparable.

  Structs comprised only of [comparable](https://go.dev/ref/spec#Comparison_operators) types (that is, ones where you can use the `==` operator) are themselves comparable, and can be used as keys in hashmaps or compared using `==`. The compiler implements these by generating comparison and hash functions for each comparable type in your code. This (very slightly) bloats the binary & compilation time. You may not want this to happen. Prevent this having a blank field of uncomparable type (the usual candidate is the ZST `[0]func()`). If you have the kind of performance requirements that need this, you'll know. Don't do it "just because"; it's confusing.

- Blank fields can provide hints to tooling like `go vet` about how a type should be used. The most famous example of this is `copylock`. See [go issue #8005](https://github.com/golang/go/issues/8005#issuecomment-190753527) for more details.

13.1. Putting it together: A generic zero-sized type

As weird as it sounds, I have a use for a **zero-sized**, **generic** struct with unreachable members:\
`context.WithValue`.

Let's review the documentation:
```

```
32
33  > #### `func WithValue(parent Context, key, val
    any) Context`
34  >
35  > WithValue returns a copy of parent in which the
    value associated with key is
36  > val.
37  > Use context Values only for **request-scoped data
    that transits processes and
38  > APIs**, not for passing optional parameters to
    functions.
39  > The provided key must be comparable and should
    not be of type
40  > string or any other built-in type to avoid
    collisions between
41  > packages using context. **Users of WithValue
    should define their own
42  > types for keys. To avoid allocating when
    assigning to an
43  > `interface{}`, context keys often have concrete
    type
44  > `struct{}
    `.** Alternatively, exported context key variables'
    static
45  > type should be a pointer or interface.
46
47  Most **request-scoped data** is a singleton per
    request. That is, it doesn't make sense for a
    request to carry around multiple loggers, users,
    traces; you want to carry the _same one_ with you
    from function call to function call
48
49  The usual way Go programs have handled this is by
    making a separate context key per type you want to
    carry in the struct. But with the advent of
    generics in `go1.18`, instead of having to make a
    new zero-sized type for every struct, we can just
    make a single generic zero-sized type and use it
    for everything:
50
51  ```go
52  type key[T] struct{}
53  // FromCtx returns the value of type T stored in
    the context, if any:
54  func FromCtx[T](ctx context) (T, bool) {
55      t, ok := context.Value(key[T]{}).(T)
56      return t, ok
57  }
58  // WithValue returns a copy of parent in which the
    value associated with `CtxKey[T]{}` is
59  // val.
60  func WithValue[T](ctx context, t T)
    (context.Context) {
61      return context.WithValue(ctx, key[T]{}, t)
```

```
62  }
```

For fun, let's rewrite `FromCtx` as a truly hellish one-liner using (nearly) every trick we've learned so far:

```
1  func FromCtx[T any](ctx context.Context) (T, bool) {t,
   ok := context.Context.Value(ctx, [0]struct{_ T}).(T);return
   t, ok}
```

That's right: this ugly SOB has a

- zero-sized type
- containing an anonymous `struct`
- with a blank identifier
- in a method expression
- on a semi-colon terminated multi-statement line

... please don't do this.

13.2. Next time(?)

- More contradictions and corner cases
- Advanced generics
- Unsafe
- Runtime shenanigans

MORE ARTICLES