Programming in Go

Matt Holiday
Christmas 2020

Details, Details

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to type information so the correct actual method can be identified

```
var r io.Reader       // nil until initialized
var b *bytes.Buffer   // ditto

r = b                 // r is no longer nil!
                      // but it has a nil pointer to a Buffer
```

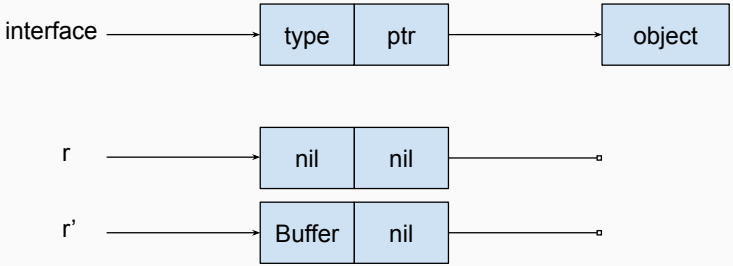
This may confuse; an interface variable is `nil` only if both parts are

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to type information so the correct actual method can be identified



Error is really an interface

We called `error` a special type, but it's really an interface

```
type error interface {
    func Error() string
}
```

We can compare it to `nil` unless we make a mistake

The mistake is to store a nil pointer to a concrete type in the error variable

Error is really an interface

```
type errFoo struct {
    err  error
    path string
}

func (e errFoo) Error() string {
    return fmt.Sprintf("%s: %s", e.path, e.err)
}

func XYZ(a int) *errFoo { return nil }

func main() {
    var err error = XYZ(1)  // BAD: interface gets a nil concrete ptr
    if err != nil {
        fmt.Println("oops")
    }
}
```

Pointer vs value receivers

A method can be defined on a pointer to a type

```
type Point struct {
    x,y float32
}

func (p *Point) Add(x, y float32) {
    p.x, p.y = p.x + x, p.y + y
}

func (p Point) OffsetOf(p1 Point) (x float32, y float32) {
    x, y = p.x - p1.x, p.y - p1.y
    return
}
```

The same method name may **not** be bound to both T and *T

Pointer vs value receivers

Pointer methods may be called on non-pointers and vice versa

Go will automatically use * or & as needed

```
p1 := new(Point)      // *Point, at (0,0)
p2 := Point{1, 1}

p1.OffsetOf(p2)       // same as (*p1).OffsetOf(p2)

p2.Add(3, 4)          // same as (&p2).Add(3, 4)
```

Except & may only be applied to objects that are *addressable*

Pointer vs value receivers

Compatibility between objects and receiver types

	Pointer	L-Value	R-Value
pointer receiver	OK	OK &	**Not OK**
value receiver	OK *	OK	OK

A method requiring a pointer receiver may only be called on an addressable object

```
var p Point

p.Add(1, 2)                 // OK, &p
Point{1, 1}.Add(2, 3)       // Not OK, can't take address
```

Consistency in receiver types

If one method of a type takes a pointer receiver, then *all* its methods should take pointers*

And in general objects of that type are probably not safe to copy

```
type Buffer struct {
    buf       []byte
    off       int
}

func (b *Buffer) ReadString(delim byte) (string, error) {
    . . .
}
```

*Well, except when they shouldn't for other reasons

Currying functions

Currying takes a function and reduces its argument count by one (one argument gets bound, and a new function is returned)

```
func Add(a, b int) int {
    return a+b
}

func AddToA(a int) func(int) int {
    return func(b int) int {
        return Add(a, b)
    }
}

addTo1 := AddToA(1)

fmt.Println(Add(1,2) == addTo1(2))    // true
```

Method values

A selected method may be passed similar to a closure;
the receiver is closed over at that point

```
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance    // this is a method value

fmt.Println(distanceFromP(q))  // and can be called later
```

Reference and value semantics

A method value with a value receiver copies the receiver

If it has a pointer receiver, it copies a pointer to the receiver

```
func (p *Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance
p = Point{3, 4}

fmt.Println(distanceFromP(q))  // uses "new" value of p
```

Interfaces in Practice

Interfaces in practice

1. Let **consumers** define interfaces
   (what *minimal* behavior do they require?)

2. Re-use standard interfaces wherever possible

3. Keep interface declarations small
   (“The bigger the interface, the weaker the abstraction”)

4. Compose one-method interfaces into larger interfaces (if needed)

5. Avoid coupling interfaces to particular types/implementations

6. Accept interfaces, but return concrete types
   (let the consumer of the return type decide how to use it)

Interfaces vs concrete values

“Be liberal in what you accept, be conservative in what you return”

- Put the least restriction on what parameters you accept
  (the *minimal* interface)

  Don’t require `ReadWriteCloser` if you only need to read

- Avoid restricting the use of your return type (the concrete value you return might fit with many interfaces!)

  Returning `*os.File` is less restrictive than returning `io.ReadWriteCloser` because files have other useful methods

Returning `error` is a good example of an exception to this rule

Empty interfaces

The `interface{}` type has no methods

So it is satisfied by anything!

Empty interfaces are commonly used; they're how the formatted I/O routines can print any type

```
func fmt.Printf(f string, args ...interface{})
```

Reflection is needed to determine what the concrete type is

We'll talk about that later