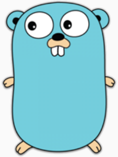
Programming in Go

Matt Holiday
Christmas 2020



Pointer/Value Semantics

Pointers vs values

Pointers	**Values**
Shared, not copied	Copied, not shared

Value semantics lead to higher integrity, particularly with concurrency (**don't share**)

Pointer semantics *may* be more efficient

Pointers vs values

Common uses of pointers:

- some objects can't be copied safely (mutex)
- some objects are too large to copy efficiently
  (consider pointers when size > 64 bytes)
- some methods need to change (mutate) the receiver [later!]
- when decoding protocol data into an object
  (JSON, etc.; often in a variable argument list)

```go
var r Response

err := json.Unmarshal(j, &r)
```

- when using a pointer to signal a "null" object

Must not copy

Any `struct` with a mutex **must** be passed by reference:

```
type Employee struct {
    mu   sync.Mutex
    Name string
    . . .
}

func do(emp *Employee) {
    emp.mu.Lock()

    defer emp.mu.Unlock()

    . . .
}
```

May copy

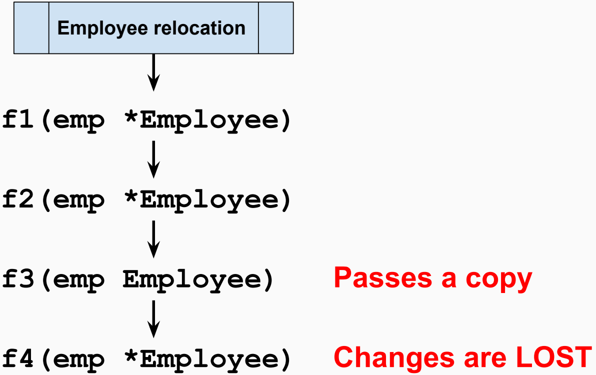
Any small `struct` under 64 bytes probably should be copied:

```go
type Widget struct {
    ID    int
    Count int
}

func Expend(w Widget) Widget {
    w.Count--

    return w
}
```

Note that Go routinely copies string & slice descriptors

Semantic consistency

If a thing is to be shared, then always pass a pointer
thanks to Bill Kennedy



Stack allocation

Stack allocation is more efficient

Accessing a variable directly is more efficient than following a pointer

Accessing a dense sequence of data is more efficient than sparse data
(an array is faster than a linked list, etc.)

Heap allocation

Go would prefer to allocate on the stack, but sometimes can't

- a function returns a pointer to a local object
- a local object is captured in a function closure
- a pointer to a local object is sent via a channel
- any object is assigned into an interface
- any object whose size is variable at runtime (slices)

The use of `new` has nothing to do with it

Build with the flag `-gcflags -m=2` to see the escape analysis

For loops

The value returned by `range` is always a copy

```
for i, thing := range things {
    // thing is a copy
    . . .
}
```

Use the index if you need to mutate the element:

```
for i := range things {
    things[i].which = whatever
    . . .
}
```

Slice safety

Anytime a function mutates a slice that's passed in, we must return a copy

```go
func update(things []thing) []thing {
    . . .
    things = append(things, x)
    return things
}
```

That's because the slice's backing array may be reallocated to grow

Slice safety

Keeping a pointer to an element of a slice is risky

```go
type user struct { name  string; count int }

func addTo(u *user) { u.count++ }

func main() {
    users := []user{{"alice", 0}, {"bob", 0}}

    alice := &users[0]          // risky
    amy := user{"amy", 1}

    users = append(users, amy)

    addTo(alice)                // alice is likely a stale pointer
    fmt.Println(users)          // so alice's count will be 0
}
```

Capturing a slice the hard way

Each append copies a reference to `item` with its last value

```go
items := [][2]byte{{1, 2}, {3, 4}, {5, 6}, {7, 8}}
a := [][]byte{}

for _, item := range items {
    a = append(a, item[:])
}

fmt.Println(items)  // [[1 2] [3 4] [5 6] [7 8]]
fmt.Println(a)      // [[7 8] [7 8] [7 8] [7 8]]
```

Capturing a slice the hard way

Each append copies a reference to `item` with its last value, so we need to copy each `item` into a (non-empty) slice

```
items := [][2]byte{{1, 2}, {3, 4}, {5, 6}, {7, 8}}
a := [][]byte{}

for _, item := range items {
    i := make([]byte, len(item))

    copy(i, item[:]) // make unique
    a = append(a, i)
}

fmt.Println(items)  // [[1 2] [3 4] [5 6] [7 8]]
fmt.Println(a)      // [[1 2] [3 4] [5 6] [7 8]]
```

Capturing the loop variable, again

Taking the address of a mutating loop variable is wrong

```go
func (r OfferResolver) Changes() []ChangeResolver {
    var result []ChangeResolver

    // wrong

    for _, change := range r.d.Status.Changes {
        result = append(result, ChangeResolver{&change})  // WRONG
    }

    return result
}
```

Wrong: all the returned resolvers point to the last change in the list

Capturing the loop variable, again

Copy the loop variable *inside the loop* before taking its address

```go
func (r OfferResolver) Changes() []ChangeResolver {
    var result []ChangeResolver

    for _, c := range r.d.Status.Changes {
        change := c // make unique

        result = append(result, ChangeResolver{&change})
    }

    return result
}
```

Now each resolver has its own change data allocated on the heap