GopherCon 2018 - Go says WAT?

By Alan Bernstein for the GopherCon Liveblog on August 28, 2018

Presenter: Jon Bodner

Liveblogger: Alan Bernstein

Despite our smugness, Go has some weird corners where the "obvious" behavior is not the "actual" behavior.

Summary

Go developers watch videos like Gary Bernhardt's Wat and feel smug. After all, Go is a typesafe language with no automatic type promotion. Go avoids magic; we accept some verbosity as a fair price to pay for not having to memorize obscure rules. Right? Not exactly. Go has some weird corners too.

Overview

- Remembering all the languages you've used gets hard over the years.
- Go feels built rather than grown.
- Language quirks can give insights into internals.
- Gary Bernhardt's WAT talk: https://www.destroyallsoftware.com/talks/wat
- Strong typing doesn't always save us.

Common WATs:

- Why doesn't this compile?
- Why aren't these things equal?
- Where did my updates go?
- Why does Go allow this?

Nothing new here if you're familiar with the innards of Go.

Instead of just talking through these, we're going to play a game. Each code snippet comes with four multiple choice answers, and the audience has to tell me what the code does.

WAT 1

What happens when you try to grow a slice in a function?

```go
func grow(s []int) {
    s = append(s, 4, 5, 6)
}

func main() {
    s := []int{1, 2, 3}
    fmt.Println(s)
    grow(s)
    fmt.Println(s)
}
```

Naive guess: The slice grows.

Crowd guess: The slice doesn't grow.

Answer: The slice doesn't grow.

Why? The slice in main doesn't have the capacity for the grown slice, so a new slice is created, and there is no change to the main slice. We can see this by examining the pointers inside and outside of the `grow` function.

WAT 2

What happens when you set the capacity high enough that the grown slice can contain the larger size?

```
func grow(s []int) {
    s = append(s, 4, 5, 6)
}

func main() {
    s := make([]int, 0, 10)
    s = append(s, 1, 2, 3)
    fmt.Println(s)
    grow(s)
    fmt.Println(s)
}
```

Naive guess: The slice definitely grows this time.

Crowd guess: mixed.

Answer: The slice doesn't grow, WAT.

Why? A slice is represented as (`array`, `len`, `cap`), which is passed by value to the function - copied. The (pointer to the) backing array is the same in both scopes, but only the copy of `len` is changed - main doesn't know that `len` has increased.

So why does the pointer to the slice stay the same? Go uses `Value.Pointer` from `reflect` to get this pointer, and that returns the array pointer, not the slice pointer.

WAT 3

What happens when you print a nonexistent map element?

```
package main

import (
    "fmt"
)

func main() {
    var m map[string]int
    fmt.Println(m["hello"])
    m["hello"] = 20
    fmt.Println(m["hello"])
}
```

Naive guess: Prints 0.

Crowd guess: mixed.

Answer: Prints 0, then panics.

Why? The read succeeds, the write fails because the map is nil; it needs to be initialized first.

WAT 4

What happens when you iterate over a map?

```
func main() {
    m := map[string]int{
        "G": 7, "A": 1,
        "C": 3, "E": 5,
        "D": 4, "B": 2,
        "F": 6, "I": 9,
        "H": 8,
    }
    var order []string
    for k, _ := range m {
        order = append(order, k)
    }
    fmt.Println(order)
}
```

Naive guess: The elements are printed in the order they were added.

Crowd guess: ¯_(ツ)_/¯

Answer: ¯_(ツ)_/¯ - indeterminate

Why? The order of iteration for a Go map (a hashmap) is unspecified, by design.

WAT 5

How many different ways will Go iterate through the same map? Say, one with nine elements?

```
func main() {
    m := map[string]int{
        "G": 7, "A": 1, "C": 3, "E": 5,
        "D": 4, "B": 2, "F": 6, "I": 9, "H": 8,
    }
    counts := map[string]int{}
    for i := 0; i < 1000000; i++ {
        var order strings.Builder
        for k, _ := range m {
            order.WriteString(k)
        }
        counts[order.String()]++
    }
    fmt.Println(len(counts))
}
```

Naive guess: 9! (9 factorial), or 362880

Crowd guess: no idea

Answer: 1-20

Why? Go achieves the randomization in a simple way, rather than randomly skipping through all the elements: it steps through the hashmap buckets in order, with a random starting bucket, and a random element in the bucket. In early Go, the iteration order was usually the same, but not always, so 1) assuming fixed order could break code, and 2) it created an attack vector, known as Hash DoS.

WAT 6

What happens when you pass a map to a function?

```go
func addStuff(m map[string]int) {
    m["b"] = 20
}

func main() {
    m := map[string]int {
        "a": 10,
    }
    fmt.Println(m)
    addStuff(m)
    fmt.Println(m)
}
```

Naive guess: Like a slice, the "b" element is not printed.

Crowd guess: Both elements are printed.

Answer: Both elements are printed.

Why? A map is a pointer to the data structure, unlike a slice, which is a struct containing a pointer to a data structure.

WAT 7

Named returns is a cute feature of Go, that lets you specify which variables are returned, without explicitly putting them in the return statement.

What happens when you combine a named return with a return statement containing a literal value?

```go
package main

import "fmt"

func double(i int) (result int) {
    result = i * 2
    return 20
}
func main() {
    result := double(5)
    fmt.Println(result)
}
```

Naive guess: 10

Crowd guess: 20

Answer: 20

Why? The explicitly specified value is the winner.

Lesson: be careful with named returns.

WAT 8

Defer statements let you specify cleanup statements to be executed at function exit.

What happens when you combine two deferred closures, with a named return and

```
func watDefer(i int) (result int) {
    defer func() {
        result = result + 1
    }()
    defer func() {
        result = result * 3
    }()
    result = i * 2
    return 20
}

func main() {
    result := watDefer(5)
    fmt.Println(result)
}
```

Naive guess: 31 = ((5*2*)*3)+1

Crowd guess: 61

Answer: 61 = (20*3)+1

Why? Defers run right before exit, in LIFO order. Also, Go inserts an assignment to the named return parameter, if an explicit return is specified.

WAT 9

What happens when you re-declare a variable in a loop, then observe it after the loop?

```
package main

import "fmt"

func main() {
    a := 1
    for i := 0;i<5;i++ {
        a := a + 1
        a = a * 2
    }
    fmt.Println(a)
}
```

Naive guess: 94

Crowd guess: 1

Answer: 1

Why? Go has block scope; all the modifications to the block-local `a` (inside the loop) have no effect on the `a` outside the loop - these are different variables. `a` is "shadowed".

WAT 10

What happens when you combine shadowed variables with named returns?

```go
func watShadow(i int) (ret int) {
    ret = i * 2
    if ret > 10 {
        ret := 10
        return
    }
    return
}

func main() {
    result := watShadow(50)
    fmt.Println(result)
}
```

Naive guess: 100

Crowd guess: 100

Answer: Does not compile.

Why? "ret is shadowed during return". The only place where shadowed variables aren't allowed (?) - it's just a bad idea.

WAT 11

What happens when you defer a closure that modifies the value of a shadowed, named return parameter?

```go
func watShadowDefer(i int) (ret int) {
    ret = i * 2
    if ret > 10 {
        ret := 10
        defer func() {
            ret = ret + 1
        }()
    }
    return
}

func main() {
    result := watShadowDefer(50)
    fmt.Println(result)
}
```

Naive guess: 101

Crowd guess: no clue.

Answer: 100

Why? The code compiles because the return is not in the same scope as the shadowed variable. The closure acts on the `ret` in the inner scope, so it doesn't affect the return value.

Lesson: use `go vet -shadow`.

WAT 12

What happens when you print a the maximum integer constant for Uint64?

```
package main

import (
    "math"
    "fmt"
)

func main() {
    a := math.MaxUint64
    fmt.Println(a)
}
```

Naive guess: 2^64-1

Crowd guess: 2^64-1

Answer: Does not compile.

Why? Constants are special in go, and they're a little weird, in terms of type. The default type (used for inference) for an int constant is `int`, which is a 32-bit type, so the 64-bit result of the expression that is defined as `math.MaxUint64` overflows the constant.

WAT 13

What happens when you perform operations with a nil struct?

```
type T struct {}

func (t *T) Num() int {
    return 2
}

func main() {
    var t *T
    fmt.Println(t == nil)
    fmt.Println(t.Num())
}
```

Naive guess: Does not compile.

Crowd guess: prints `true` and `2`.

Answer: prints `true` and `2`.

Why? There's no assumption in the method that requires `t` to be non-nil. This is actually useful, for example handling nodes in a binary tree.

See Francesc Campoy Flores' talk https://speakerdeck.com/campoy/understanding-nil for more info.

WAT 14

What happens when a nil error is returned from a function call?

```
func err1() error {
    var err error
    fmt.Println(err == nil)
    return err
}

func main() {
    err := err1()
    fmt.Println(err == nil)
}
```

Naive guess: Both `true`.

Crowd guess: `true` then `false`.

Answer: Both `true`.

Why? Makes sense.

WAT 15

What happens when a concrete instance of an error with nil value is returned from a function call?

```
type MyError string

func(me MyError) Error() string {
    return string(me)
}

func err2() error {
    var err *MyError
    fmt.Println(err == nil)
    return err
}

func main() {
    err := err2()
    fmt.Println(err == nil)
}
```

Naive guess: Both `true`.

Crowd guess: `true`, `false`.

Answer: `true`, `false`.

Why?

Short answer: `nil` means different things for an interface, vs other types.

Long answer: Three basic ideas:

- `nil` is weird in Go, and in most languages with an equivalent concept. It's a "hole in the type system".
- Interfaces are represented as (tab, data); a tab is (interface type, underlying type).
- Rules for instance comparisons are moderately complex.

That underlying type is the key difference in this WAT:

- With `var err error`, the underlying type is the zero value, or `nil` here.
- When a variable of concrete type is assigned to a variable of interface type (`var me *MyError` is assigned to `err` after `var err error`), the value is copied and the concrete type is copied.
- When a variable of interface type is assigned to another variable of interface type (`var err1 error` is assigned to `err` after `var err error`), Go copies both the value and the underlying type.

Go never wraps one interface inside of another.

Then we have the rules for instance comparisons:

- Two variables with concrete types compares values only
- Two variables with interface types compares underlying types and values
- An interface and a concrete type compares concrete type with underlying type and compares values

Putting that all together: interface nil comparisons include the underlying type, which is different here, even though the value is nil for both.

This is tricky, but it seems like the most consistent and reasonable option.

WAT 16

What happens when you assign the value `false` to a variable called `true`?

```
package main

import "fmt"

func main() {
    true := false
    fmt.Println(true)
}
```

Naive guess: Does not compile.

Crowd guess: Does not compile.

Answer: Prints `false`.

Why? Keywords can't be used as variable names, but `true` is a built-in identifier, NOT a keyword. This is actually the same as shadowing, where the higher scope is "universe block-level", the scope where the built-ins are normally defined. You can shadow `new`, `nil`, `iota`, and more!

These are not caught by `go vet -shadow`!

Morals

- Be careful with named returns.
- Use `go vet -shadow`.
- Let's deprecate shadowing!
- Be wary of nil interface comparisons.
- Turn WATs into hows and whys.

Would you like go-to-def and find-refs on GitHub?

Read more posts

sourcegraph

hi@sourcegraph.com

142 Minna St, 2nd Floor
San Francisco CA, 94105

COMMUNITY

Blog

GitHub

LinkedIn

Twitter

COMPANY

Master Plan

About

Contact

Careers

FEATURES

Code Search

Code Intelligence

Data Center

Integrations

RESOURCES

Documentation

Changelog

Pricing

Security

hi@sourcegraph.com

142 Minna St, 2nd Floor
San Francisco CA, 94105

Copyright © 2018 Sourcegraph, Inc. Terms Privacy