





Golang

My journey trying to learn coding in Go:
The basics

Nacho Álvarez

 @neonigma

 neonigma@gmail.com

3 de diciembre de 2021

- 1 whoami
- 2 Introduction
- 3 Basics
- 4 Flow control
- 5 More types
- 6 Methods
- 7 Concurrency

Who?

- ▶ Computer Science Engineer
- ▶ Certifications Scrum Manager by 2012 and PMP by 2017
- ▶ Career (+12y):
 - Web developer
 - Free and Open Source software developer (Emergya / Guadalinfo)
 - Android mobile + backend developer (WUL4)
 - R+D Engineer (Redsys)
 - Solutions Architect (Redsys)
 - Engineering Manager / SRE (Redsys Salud)



UNIVERSIDAD
DE
CÓRDOBA



Scrum Manager
Certified

Introduction

- ▶ Statically typed, compiled programming language designed at Google
- ▶ Go uses a syntax similar to C
- ▶ It provides a garbage collector, reflection, and some other high-level language capabilities.
- ▶ The Go binary has the cross-compilation feature natively.
- ▶ Go supports the object-oriented programming paradigm, but it does not have type inheritance
- ▶ In Go, the definition of a type (“class”) is done by means of separate declarations (interfaces, structs, embedded values).
- ▶ Go allows the use of delegation (via embedded values) and polymorphism (via interfaces).
- ▶ Go can implement concurrency through the goroutines.
- ▶ Go is designed to take advantage of systems with multiple processors and network processing.
- ▶ Go supports dynamic data typing also known as duck typing.
- ▶ A struct can implement an interface automatically.

Basics

```
package main

import "fmt"

var a = 2

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)

    var i, j int = 1, 2
    k := 3
    c, python, java := true, false, "no!"

    fmt.Println(i, j, k, c, python, java, a)
}
```

Basics

- ▶ Packages
- ▶ Imports
- ▶ Exported names
- ▶ Functions
- ▶ Multiple return results
- ▶ Variables at package or function level and with initializers
- ▶ Short variable declarations

Basics

▶ Go's basic types are:

- bool
- string
- int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr
- byte // alias for uint8
- rune // alias for int32, represents a Unicode code point
- float32 float64
- complex64 complex128

▶ And for zero values:

- 0 for numeric types,
- false for the boolean type, and
- (the empty string) for strings.

Basics

```
package main

import (
    "fmt"
    "math"
)

const Pi = 3.14

func main() {
    var x, y int = 3, 4
    f := math.Sqrt(float64(x*x + y*y))
    z := uint(f)
    fmt.Println(x, y, z)

    var i int
    j := i // j is an int

    i := 42           // int
    f := 3.142        // float64
    g := 0.867 + 0.5i // complex128
}
```


Basics

- ▶ Type conversions
- ▶ Type inference
- ▶ Constants
- ▶ Numeric constants

For

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ { // init and post are optional
        sum += i
    }
    fmt.Println(sum)

    newsum := 1 // Go doesn't like to reuse vars
    for newsum < 1000 {
        newsum += newsum
    }
    fmt.Println(newsum)
}
```

If

```
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    // can't use v here, though
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10), pow(3, 3, 20),
    )
}
```

Switch

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.\n", os)
    }
}
```

Defer

```
package main

import "fmt"

func main() {
    fmt.Println("counting")

    for i := 0; i < 10; i++ {
        defer fmt.Printf("%v,", i)
    }

    fmt.Println("done")
}

counting
done
9,8,7,6,5,4,3,2,1,0,
```

Pointers

```
package main

import "fmt"

func main() {
    i, j := 42, 2701

    p := &i           // point to i
    fmt.Println(*p)    // read i through the pointer
    *p = 21            // set i through the pointer
    fmt.Println(i)     // see the new value of i

    p = &j            // point to j
    *p = *p / 37       // divide j through the pointer
    fmt.Println(j)     // see the new value of j
}
```

Structs

```
package main

import "fmt"

type Vertex struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2} // has type Vertex
    v2 = Vertex{X: 1}  // Y:0 is implicit (X literal)
    v3 = Vertex{}      // X:0 and Y:0
    p  = &Vertex{1, 2} // has type *Vertex
)

func main() {
    fmt.Println(v1, p, v2, v3)
}
```

Arrays

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Hello"
    a[1] = "World"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)
}
```


Slices

```
package main

import "fmt"

func main() {
    names := [4]string{
        "John",
        "Paul",
        "George",
        "Ringo",
    }
    fmt.Println(names)

    a := names[0:2]
    b := names[1:3]
    fmt.Println(a, b)

    b[0] = "XXX" // as being a reference to array, modifies its value
    fmt.Println(a, b)
    fmt.Println(names)
}
```

Slices

```
package main

import "fmt"

func main() {
    s := []int{2, 3, 5, 7, 11, 13} // slice literal
    // can use s := make([]int, 0, 5) => [] (len 0, cap 5)

    s = s[1:] // slice default
    fmt.Println(s)

    s = s[:0] // Extend its length.
    printSlice(s)

    s = s[:3] // Drop its first two values.
    printSlice(s) // can append s = append(s, 2, 3, 4)
}

func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

Range

```
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for _, value := range pow {
        fmt.Printf("%d\n", value)
    }
}
```

Maps

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
    "Bell Labs": Vertex{40.68433, -74.39967}, // type can be omitted
    "Google": Vertex{37.42202, -122.08408},
}

func main() {
    fmt.Println(m)

    m = make(map[string]Vertex) // another way
    m["Bell Labs"] = Vertex{40.68433, -74.39967}

    v, ok := m["Bell Labs"] // also can delete(m, "Bell Labs")
    fmt.Println("The value:", v, "Present?", ok)
}
```

Function values

```
package main

import (
    "fmt"
    "math"
)

func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}

func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(5, 12))

    fmt.Println(compute(hypot))
    fmt.Println(compute(math.Pow))
}
```

Function closures

```
package main

import "fmt"

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}
```

Receivers

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
```

Receivers

```
func main() {  
    v := Vertex{3, 4}  
    v.Scale(10)  
    fmt.Println(v)  
    fmt.Println(v.Abs())  
}
```

```
// With a pointer receiver, you can modify the original value, as Scale  
// does here  
// Abs method can just read the value
```

```
// There are two reasons to use a pointer receiver.
```

```
// The first is so that the method can modify the value that its receiver  
// points to.
```

```
// The second is to avoid copying the value on each method call. This  
// can be more efficient if the receiver is a large struct, for example.
```


Interfaces

```
package main

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f // a MyFloat implements Abser
    a = &v // a *Vertex implements Abser
    a = v // v is a Vertex (not *Vertex) and does NOT implement Abser

    fmt.Println(a.Abs())
}
```

Interfaces

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Interfaces

```
package main

import "fmt"

type I interface {
    M()
}

type T struct {
    S string
}

func (t *T) M() {
    if t == nil {
        fmt.Println("<nil>")
        return
    }
    fmt.Println(t.S)
}
```

Interfaces

```
// Interface values with nil underlying values
func main() {
    var i I

    var t *T
    i = t
    describe(i)
    i.M()

    i = &T{"hello"}
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

Interfaces

```
// Interface values with nil underlying values
func main() {
    var i I

    var t *T
    i = t
    describe(i)
    i.M()

    i = &T{"hello"}
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

Interfaces

```
package main

import "fmt"

func main() {
    var i interface{} // empty interface
    fmt.Printf("(%v, %T)\n", i, i)

    i = 42
    fmt.Printf("(%v, %T)\n", i, i)

    var j interface{} = "hello"

    s, ok := j.(string) // type assertions
    fmt.Println(s, ok)

    f, ok := j.(float64) // you could even switch by j.()type)
    fmt.Println(f, ok)

    f = j.(float64) // panic
    fmt.Println(f)
}
```

Stringers

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func (p Person) String() string {
    return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
    a := Person{"Arthur Dent", 42}
    z := Person{"Zaphod Beeblebrox", 9001}
    fmt.Println(a, z)
}
```

Errors

```
package main

// import fmt & time here

type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s", e.When, e.What)
}

func run() error {
    return &MyError{time.Now(), "it didn't work"}
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```


Readers

```
package main

import (
    "fmt"
    "io"
    "strings"
)

func main() {
    r := strings.NewReader("Hello, Reader!")

    b := make([]byte, 8)
    for {
        n, err := r.Read(b)
        fmt.Printf("n = %v err = %v b = %v\n", n, err, b)
        fmt.Printf("b[:n] = %q\n", b[:n])
        if err == io.EOF {
            break
        }
    }
}
```

Goroutines, channels, range and close

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

Select

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("      .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

Basics

- ▶ The select statement lets a goroutine wait on multiple communication operations.
- ▶ A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.
- ▶ The default case in a select is run if no other case is ready.
- ▶ Use a default case to try a send or receive without blocking.

sync.Mutex

```
package main

import (
    "fmt"
    "sync"
    "time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
    mu sync.Mutex
    v  map[string]int
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
    c.mu.Lock()
    // Lock so only one goroutine at a time can access the map c.v.
    c.v[key]++
    c.mu.Unlock()
}
```

sync.Mutex

```
// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
    c.mu.Lock()
    // Lock so only one goroutine at a time can access the map c.v.
    defer c.mu.Unlock()
    return c.v[key]
}

func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 1000; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}
```