Note of Go

Jingqi Chen*

August 10, 2019

Contents

*E-mail address:hobochen96@gmail.com;

Contents

1 A Tour of Go

1.1 Install Go

Install:

```
brew install go # macOS
sudo apt install golang # Debian
```

Hello World:

```
WORK_DIRECTORY='$HOME/Projects/hobotoyroom/go'
pushd $WORK_DIRECTORY
    export GOPATH=`realpath .`
    mkdir -p src/hello
    pushd src/hello
        # vi hello.go
        go build
        ./hello # => stdout prints 'hello world'
    popd
popd
```

`hello.go`:

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

1.2 Get *A Tour of Go*

Online or run the command:

```
pushd $GOPATH
    go get golang.org/x/tour/gotour
    pushd bin
        ./gotour
    popd
popd
```

1.3 Basics

1.3.1 Packages, variables, and functions

Go programs are made up of packages. Programs start running in package `main`. The package name is the same as the last element of the import path. For instance, the `math/rand` package comprises files that begin with the statement `package rand`.

Name is exported if it begins with a capital letter.

The type comes *after* the variable name, and you can refer to Go's Declaration Syntax for why go uses that syntax. In summary, it is to avoid the confusing of C-like function declaration such as `int (*(*fp)(int (*)(int, int), int))(int, int)`.

A function can return any number of values. Go's return values may be named which are treated as variables defined at the top of the function.

Outside a function, which is the package level, every statement begins with a keyword as `var`, `func`, etc; so the `:=` is not available.

Go's basic types are:

```
bool
string // immutable
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
```

Variables declared without an explicit initial value are given:

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

Unlike in C, in Go assignment between items of different type requires an explicit conversion like `T(v) // convert v to Type T`.

Numeric constants are high-precision values. There is something showed that for why it is high-precision can be found at Constants, but I still do not know with minutes reading.

1.3.2 Flow control statements: `for`, `if`, `else`, `switch` and `defer`

A `switch` statement is a shorter way to write a sequence of `if - else` statements. It runs the first case whose value is equal to the condition expression. Go only runs the selected case, not all the cases that follow.

```
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.", os)
    }
}
```

A defer statement defers the execution of a function until the surrounding function returns. Defer statement follows the order as a stack, you can refer to Defer, Panic, and Recover.

1.3.3 More types: structs, slices, and maps

A `struct` is a collection of fields.

```
type Vertex struct {
    x int
```

```
3     y int
4 }
```

A struct literal denotes a newly allocated struct value by listing the values of its fields. Listing a subset of fields by using the `_name: _value` syntax works as well, and the order of named fields is irrelevant.

Go has pointer, but Go has no pointer arithmetic. To access a fied of a struct from a struct pointer, explicit dereferencing is not needed.

The type `[n]T` is an array of `n` values of type `T`. An array's length is part of its type, so arrays cannot be resized.

A slice, like `[]T`, is a dynamically-sized. In practice, slices are much more common than arrays.

Slices are like **references** to arrays.

A slice literal is like an array literal without the length. A slice has both a *length* and a *capacity*, wich can be obtained by `len()`, `cap()`.

Slices can be created with the built-in make function; this is how you create dynamically-sized arrays.

```
a := make([]int, 5)
```

It is common to append new elements to a slice, and so Go provides a built-in append function, which interface is like `func append(s []T, vs ...T) []T`.

The `range` form of the `for` loop iterates over a slice or map. When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

A `map` maps keys to values.

```
var m map[string]string
```

Functions are values too, which can be used as function arguments and return values.

Go functions may be closures, and the context variables are passed by reference. A closure is a function value that references variables from outside its body.

```
 1 func adder() func(int) int {
 2     sum := 0
 3     return func(x int) int {
 4         sum += x
 5         return sum
 6     }
 7 }
 8
 9 func main() {
10     pos, neg := adder(), adder()
11     for i := 0; i < 10; i++{
12         fmt.Println(pos(i), neg(-2*i))
13     }
14 }
```

1.4 Methods and interfaces

Go does not have classes. However, you can define methods on types. A method is a function with a special *receiver* argument. The receiver appears in its own argument list between the `func` keyword and the method name.

```
type Vertex struct {
    X, Y float64
}

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

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

Only Methods with pointer receivers can modify the value type to which the receiver points. Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

Functions with a pointer argument must take a pointer, while methods with pointer receivers take either a value or a pointer as the receiver when they are called.

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

func ScaleFunc(v *Vertex, f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(2)
    ScaleFunc(&v, 10)

    p := &Vertex{4, 3}
    p.Scale(3)
    ScaleFunc(p, 8)

    fmt.Println(v, p)
}
```

An *interface* type is defined as a set of method signatures. A value of interface type can hold any value that implements those methods.

A type implements an interface by implementing its methods. There is no explicit declaration of intent, no “implements” keyword.

```
type I interface {
    M()
}

type T struct {
    S string
}

func (t T) M() {
```

```
10     fmt.Println(t.S)
11 }
12
13 func main() {
14     var i I = T{"hello"}
15     i.M()
16 }
```

An interface value holds a value of a specific underlying concrete type.

If the concrete value inside the interface itself is nil, the method will be called with a nil receiver.

```
1 func (t *T) M() {
2     if t == nil {
3         fmt.Println("<nil>")
4         return
5     }
6     fmt.Println(t.S)
7 }
```

A nil interface value holds neither value nor concrete type. Calling a method on a nil interface is a run-time error because there is no type inside the interface tuple to indicate which *concrete* method to call.

The interface type that specifies zero methods is known as the empty interface: `interface{}`. An empty interface may hold values of any type.

A *type assertion* provides access to an interface value's underlying concrete value.

```
 1 t := i.(T)
 2 t, ok := i.(T) // To test whether the i holds the specific type T
 3
 4 // type switch
 5 func do(i interface{}) {
 6     switch v := i.(type) {
 7     case int:
 8         fmt.Printf("Twice %v is %v\n", v, v*2)
 9     case string:
10         fmt.Printf("%q is %v bytes long\n", v, len(v))
11     default:
12         fmt.Printf("I don't know about type %T!\n", v)
13     }
14 }
```

This statement asserts that the interface value `i` holds the concrete type `T` and assigns the underlying `T` value to the variable `t`.

One of the most ubiquitous interfaces is Stringer defined by the `fmt` package.

```
1 type Stringer interface {
2     String() string
3 }
```

Go programs express error state with `error` values. The `error` type is a built-in interface similar to `fmt.Stringer`.

```
1 type error interface {
2     Error() string
3 }
```

```
type MyError struct {
    When time.Time
    What string
}

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

The io package specifies the io.Reader interface, which represents the read end of a stream of data. The Go standard library contains many implementations of these interfaces, including files, network connections, compressors, ciphers, and others. The io.Reader interface has a Read method:

```
func (T) Read(b []byte) (n int, err error)
```

Read populates the given byte slice with data and returns the number of bytes populated and an error value. It returns an io.EOF error when the stream ends.

```
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
        }
    }
}
```

1.5 Concurrency

A *goroutine* is a lightweight thread managed by the Go runtime. Goroutines run in the same address space, so access to shared memory must be synchronized.

Channels are a typed conduit through which you can send and receive values with the channel operator, <-. By default, sends and receives block until the other side is ready, which allows goroutines to synchronize without explicit locks or condition variables.

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
```

```
15      x, y := <-c, <-c // receive from c
16
17      fmt.Println(x, y, x+y)
18  }
```

Channels can be *buffered*. Provide the buffer length as the second argument to make to initialize a buffered channel.

A sender can close a channel to indicate that no more values will be sent.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
 1  func fibonacci(c, quit chan int) {
 2      x, y := 0, 1
 3      for {
 4          select {
 5          case c <- x:
 6              x, y = y, x+y
 7          case <-quit:
 8              fmt.Println("quit")
 9              return
10          }
11      }
12  }
13
14  func main() {
15      c := make(chan int)
16      quit := make(chan int)
17      go func() {
18          for i := 0; i < 10; i++ {
19              fmt.Println(<-c)
20          }
21          quit <- 0
22      }()
23      fibonacci(c, quit)
24  }
```

If not communication but only access to a variable at a time is wanted, *mutual exclusion* is needed, whose conventional name is `mutex`.

Go's standard library provides mutual exclusion with sync.Mutex and its two methods: `Lock`, `Unlock`

```
 1  // SafeCounter is safe to use concurrently.
 2  type SafeCounter struct {
 3      v   map[string]int
 4      mux sync.Mutex
 5  }
 6
 7  // Inc increments the counter for the given key.
 8  func (c *SafeCounter) Inc(key string) {
 9      c.mux.Lock()
10      // Lock so only one goroutine at a time can access the map c.v.
11      c.v[key]++
12      c.mux.Unlock()
13  }
```

```
14
15 // Value returns the current value of the counter for the given key.
16 func (c *SafeCounter) Value(key string) int {
17     c.mux.Lock()
18     // Lock so only one goroutine at a time can access the map c.v.
19     defer c.mux.Unlock()
20     return c.v[key]
21 }
```

1.6 Further

Further of A Tour of Go

2 How to Write Go Code

Source

This document demonstrates the development of a simple Go package and introduces the go tool, the standard way to fetch, build, and install Go packages and commands.

2.1 Code Organization

- Go programmers typically keep all their Go code in a single workspace; which is specified by `GOPATH`.
- A workspace contains many version control repositories (managed by Git, for example).
- Each repository contains one or more packages.
- Each package consists of one or more Go source files in a single directory.
- The path to a package's directory determines its import path.

An example of a workspace:

```
 1 bin/
 2     hello                          # command executable
 3     outyet                         # command executable
 4 src/
 5     github.com/golang/example/
 6         .git/                      # Git repository metadata
 7     hello/
 8         hello.go               # command source
 9     outyet/
10         main.go                # command source
11         main_test.go           # test source
12     stringutil/
13         reverse.go             # package source
14         reverse_test.go        # test source
15     golang.org/x/image/
16         .git/                      # Git repository metadata
17     bmp/
18         reader.go              # package source
19         writer.go              # package source
20     ... (many more repositories and packages omitted) ...
```

An *import path* is a string that uniquely identifies a package. A package's import path corresponds to its location inside a workspace or in a remote repository.

Also, the import path indicates the location of a remote library, which can be gotten by `go get`.

There is a Go Project List to read.

2.2 First Program

```
1 mkdir $GOPATH/src/github.com/$github_user_name/hello
2 pushd $GOPATH/src/github.com/$github_user_name/hello && vi hello.go
3
4 go install
5
6 $GOPATH/bin/hello # => Hello, world.
```

hello.go:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, world.")
7 }
```

2.3 First Library

```
1 mkdir $GOPATH/src/github.com/$github_user_name/stringutil
2
3 vi reverse.go
4
5 go build github.com/$github_user_name/stringutil # won't produce an output file, but saves
```

reverse.go:

```
 1 // Package stringutil contains utility functions for working with strings.
 2 package stringutil
 3
 4 // Reverse returns its argument string reversed rune-wise left to right.
 5 func Reverse(s string) string {
 6     r := []rune(s)
 7     for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
 8         r[i], r[j] = r[j], r[i]
 9     }
10     return string(r)
11 }
```

2.4 Testing

Go has a lightweight test framework composed of the `go test` command and the testing package.

```
1 mkdir $GOPATH/src/github.com/$github_user_name/stringutil
2 vi reverse_test.go
3 go test
```

reverse_test.go:

```
 1 package stringutil
 2
 3 import "testing"
 4
 5 func TestReverse(t *testing.T) {
 6     cases := []struct {
 7         in, want string
 8     }{
 9         {"Hello, world", "dlrow ,olleH"},
10         {"Hello, ", " ,olleH"},
11         {"", ""},
12     }
13     for _, c := range cases {
14         got := Reverse(c.in)
15         if got != c.want {
16             t.Errorf("Reverse(%q) == %q, want %q", c.in, got, c.want)
17         }
18     }
19 }
```

3 Go Concurrency Patterns

3.1 Concurrency is not Parallelism

It was a talk given by Rob Pike, slide, video.

3.1.1 Concurrency & Parallelism

Concurrency is programming as the composition of independently executing processes; Parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about structure, parallelism is about execution.

Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once.

Analogy:

1. Concurrent: Mouse, keyboard, display, and disk drivers
2. Parallel: Vertor doc product

3.1.2 Concurrency & Communication

Concurrency is a way to structure a program by breaking it into pieces that can be executed independently.

Communication is the means to coordinate the independent executions. This is the Go model and (like Erlang and others) it's based on CSP: **C. A. R. Hoare: Communicating Sequential Processes**.

3.1.3 Lesson

(A Example of Gophers bring books from a pile to a incinerator.)

There are many ways to break the processing down, which is concurrent design. Once we have the breakdown, parallelization can fall out and correctness is easy.

3.1.4 Go and Concurrency

3.2 Go Concurrency Patterns

It was a talk given by Rob Pike in Google I/O 2012, slide, video.

Concurrency is not parallelism, although it enables parallelism.

3.2.1 History

Concurrency features of Go are rooted in a long history, reaching back to Hoare's CSP in 1978 and even Dijkstra's guarded commands.

3.2.2 Distinction

Go is the latest on the Newsqueak-Alef-Limbo branch, distinguished by first-class channels.

Erlang is closer to the original CSP, where you communicate to a process by name rather than over a channel.

Rough analogy: writing to a file by name (process, Erlang) vs. writing to a file descriptor (channel, Go).

3.2.3 Channel

Buffering removes synchronization, and makes them more like Erlang's mailboxes.

3.2.4 No Overdo

Sometimes a refernce counter is needed.

Go has `sync` and `sync/atomic`.

4 Effective Go

Source

TODO