Contents

[安装环境 1](#_Toc489625932)

[语法 2](#_Toc489625933)

[Basic types, 变量, 循环与条件, 指针与struct类型, Array, slices, map, Multiple result, defer 2](#_Toc489625934)

[函数,闭包,方法和接口 4](#_Toc489625935)

[Goroutines and Channels 8](#_Toc489625936)

# 安装环境

Features:

concurrency mechanisms

multicore and networked machines

modular program

garbage collection and the power of run-time reflection

statically typed, compiled language that feels like a dynamically typed, interpreted language

install

sudo apt-get install golang-go

install gb

$go get github.com/constabulary/gb/...

Create gb project

getgb.io/examples/getting-started/

tabalt.net/blog/golang-package-dependency-management-tool-gb/

A gb project is a workspace which contains all the source needed to build and test your library or application

A gb project is conceptually a $GOPATH per project, All Go code is placed inside packages, and packages are subdirectories inside the project’s src/ directory.

Creating a Project

/home/dfc/code/demo-project

└── src

└── hello

└── hello.go

└── vendor

└── src

查看gb命令

$ gb help

$ gb vendor help

$ gb build all

若添加第三方依赖包，需要下载

$ gb vendor fetch github.com/tabalt/gracehttp

$ bin/hello

# 语法

内置类型，包括struct, map, array, slices, … can be fmt.Println() directly

## Basic types, 变量, 循环与条件, 指针与struct类型, Array, slices, map, Multiple result, defer

Bool, string, int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr

The int, uint, and uintptr types are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems

byte // alias for uint8,

rune // alias for int32, represents a Unicode code point

float32 float64, complex64 complex128

variables （变量）

var index int

// Variables with initializers

i := 42 // int

f := 3.142 // float64

g := 0.867 + 0.5i // complex128

const Pi = 3.14

循环与条件

等效while

for sum < 1000 {

…

}

switch os := runtime.GOOS; os {

case "darwin":

…

default:

…

}

switch等效long if-then-else chains

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("Good morning!")

default:

fmt.Println("Good evening.")

}

指针与struct类型

pointer

p := &i // point to i

\*p = 21 // set i through the pointer

type Vertex struct {

X int

Y int

}

v := Vertex{1, 2}

v.X = 4

p := &v

p.X = 1e9

(\*p).X = 31

Array, slices, map

primes := [6]int{2, 3, 5, 7, 11, 13} // array

var s []int = primes[1:4] // slices, references to arrays

len(s) and cap(s) //The length and capacity of a slice s

b := make([]int, 0, 5) // len(b)=0, cap(b)=5 dynamically-sized arrays

// slices of slices

board := **[][]string**{

[]string{"\_", "\_", "\_"},

[]string{"\_", "\_", "\_"},

[]string{"\_", "\_", "\_"},

}

board[i][j] = "X"

var s []int //len=0 cap=0 []

s = append(s, 0) //len=1 cap=2 [0]

for i, v := range pow { // 序列迭代

…

}

var m = **map[string]Vertex**{

"Bell Labs": { 40.68433, -74.39967},

"Google": {37.42202, -122.08408},

}

Multiple results

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

return y, x

}

a, b := swap("hello", "world")

defer and stacking defers

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

func main() {

defer fmt.Println(“qzlin")

def fmt.Println(“world”)

fmt.Println("hello")

}

The result will be: “hello world qzlin”

## 函数,闭包,方法和接口

**Function** values may be used as function arguments and return values.

func compute(fn func(float64, float64) float64) float64 {

return fn(3, 4)

}

hypot := func(x, y float64) float64 {

return math.Sqrt(x\*x + y\*y)

}

compute(hypot)

Function closures

func adder() func(int) int {

sum := 0

return func(x int) int {

sum += x

return sum

}

}

pos, neg := adder(), adder()

pos(i)

neg(-2\*i),

**Methods （方法）**

Go does not have classes. However, you can define methods on types.

a method is just a function with a receiver argument.

func (v Vertex) Abs() float64 {

return math.Sqrt(v.X\*v.X + v.Y\*v.Y)

}

v := Vertex{3, 4}

v.Abs()

You can only declare a method with a receiver whose type is defined in the same package as the method. You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as int).

type MyFloat float64

func (f MyFloat) Abs() float64 {

if f < 0 {

return float64(-f)

}

return float64(f)

}

f := MyFloat(-math.Sqrt2)

f.Abs()

Methods with pointer receivers can modify the value to which the receiver points (as Scale does here). Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

func (v \*Vertex) Scale(f float64) {

v.X = v.X \* f

v.Y = v.Y \* f

}

v := Vertex{3, 4}

v.Scale(10) //改变v的状态，若不是指针，则值传递不改变状态

fmt.Println(v.Abs())

functions with a pointer argument must take a pointer

methods with pointer receivers take either a value or a pointer as the receiver when they are called

var v Vertex

v.Scale(5) // OK

p := &v

p.Scale(10) // OK

even though v is a value and not a pointer, the method with the pointer receiver is called automatically. That is, as a convenience, Go interprets the statement v.Scale(5) as (&v).Scale(5) since the Scale method has a pointer receiver.

Choosing a value or pointer receiver

the method can modify the value that its receiver points to.

avoid copying the value on each method call

In general, all methods on a given type should have either value or pointer receivers, but not a mixture of both

**Interfaces (多态)**

An interface type is defined as a set of method signatures

Interfaces are implemented implicitly

type I interface {

M()

}

type T struct {

S string

}

// This method means type T implements the interface I, but we don't need to explicitly declare that it does so.

func (t T) M() {

fmt.Println(t.S)

}

type F float64

func (f F) M() {

fmt.Println(f)

}

var i I

i = &T{"Hello"}

i.M() //调用T.M()

i = F(math.Pi)

i.M() //调用F.M()

范例1：

//内置的String()

type Stringer interface {

String() string

}

type Person struct {

Name string

Age int

}

func (p Person) String() string {

return fmt.Sprintf("%v (%v years)", p.Name, p.Age)

}

a := Person{"Arthur Dent", 42}

z := Person{"Zaphod Beeblebrox", 9001}

fmt.Println(a, z) //实际上是调用func (object Object) String() string {}

范例2:

Go programs express error state with error values.

//内置的error

type error interface {

Error() string

}

i, err := strconv.Atoi("42")

if err != nil {

fmt.Printf("couldn't convert number: %v\n", err)

return

}

fmt.Println("Converted integer:", i)

//自定义error

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() string, error) {

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

}

范例3:

//内置的Readers

type Reader interface {

Read(p []byte) (n int, err error)

}

The Go standard library contains many implementations of these interfaces, including files, network connections, compressors, ciphers, and others.

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

}

}

范例4:

//内置的Images

type Image interface {

ColorModel() color.Model

Bounds() Rectangle

At(x, y int) color.Color

}

Type assertions

var i interface{} = "hello"

s, ok := i.(string)

fmt.Println(s, ok)

func do(i interface{}) {

switch v := i.(type) {

case int:

fmt.Printf("Twice %v is %v\n", v, v\*2)

case string:

fmt.Printf("%q is %v bytes long\n", v, len(v))

default:

fmt.Printf("I don't know about type %T!\n", v)

}

}

## Goroutines and Channels

lightweight thread managed by the Go runtime

Goroutines run in the same address space, so access to shared memory must be synchronized. The sync package provides useful primitives, although you won't need them much in Go as there are other primitives

channels must be created before use, By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

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

c := make(chan int) //创建channel

go sum(s[:len(s)/2], c) //开启轻线程计算，并将结果发送到c中

go sum(s[len(s)/2:], c)

x, y := <-c, <-c // receive from c

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)

}

c := make(chan int, 10)

go fibonacci(cap(c), c)

for **i := range c** { //receives values from the channel repeatedly until it is closed.

fmt.Println(i)

}