Go是一种新的语言，一种并发的、带垃圾回收的、快速编译的语言。Go语言主要用作服务器端开发.

它具有以下特点：

1.几秒钟的时间编译一个大型的Go程序。

2.Go为软件构造提供了一种模型，它使依赖分析更加容易，且避免了大部分C风格include文件与库的开头。

3.Go是静态类型的语言，它的类型系统没有层级，这样感觉起来比典型的面向对象语言更轻量级。

4.Go完全是垃圾回收型的语言，并为并发执行与通信提供了基本的支持。

5.Go是一种编译型语言(不是脚本语言啊)，它结合了解释型语言的游刃有余，动态类型语言的开发效率，以及静态类型的安全性。

6.支持网络与多核计算的语言。

Go 语言最主要的特性：

自动垃圾回收

更丰富的内置类型

函数多返回值

错误处理

匿名函数和闭包

类型和接口

并发编程

反射

Go 语言的基础组成有以下几个部分：

包声明

引入包

函数

变量

语句 & 表达式

注释

Go 程序的一般结构: basic\_structure.go

// 当前程序的包名

package main

// 导入其他包

import . "fmt" (加点不用申明fmt，直接用了)

// 常量定义

const PI = 3.14

// 全局变量的声明和赋值

var name = "gopher"

// 一般类型声明

type newType int

// 结构的声明

type gopher struct{}

// 接口的声明

type golang interface{}

// 由main函数作为程序入口点启动

func main() {

Println("Hello World!")

}

Go 语言按类别有以下几种数据类型：

序号 类型和描述

1 布尔型

布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。

2 数字类型

整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。

3 字符串类型:

字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。

4 派生类型:

包括：

(a) 指针类型（Pointer）

(b) 数组类型

(c) 结构化类型(struct)

(d) Channel 类型

(e) 函数类型

(f) 切片类型

(g) 接口类型（interface）

(h) Map 类型

package main

import "fmt"

func main() {

fmt.Println("Hello, World!")

}

Goroutines 并发模式

1. 并发而非并行

首先我们要明确两个名词：并发(Concurrency)、并行(Parallelism)。这两个词可能大家经常搞混淆，因为这两个词所标书的意思太过相近，但是前者更加偏向于设计(Design)，而后者更加偏向于结构(Structure)。

如果你有只有一个CPU，那么你的程序可以是并发的，但一定不是并行的

一个良好的并发程序并非一定是并行的

并行是一种物理状态，而并发是一种设计思想、程序的内部结构

多处理器才有可能达到并发的物理状态

2. 什么是Goroutines

Goroutine是一个通过go关键字起起来的独立的执行某个function的过程，它拥有独立的可以自行管理的调用栈。

goroutine非常廉价，你可以拥有几千甚至上万的goroutines

goroutine不是thread

一个thread之下可能有上千的goroutines

你可以把goroutine理解为廉价的thread

package main

import (

"fmt"

"math/rand"

"time"

)

func boring(msg string) {

for i := 0; ; i++ {

fmt.Println(msg, i)

time.Sleep(time.Duration(rand.Intn(1e3)) \* time.Millisecond)

}

}

func main() {

//boring("boring!")

go boring("boring!")

go boring("spring!")

time.Sleep(20 \* time.Second)

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

使用channels！

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package main

import (

"fmt"

"math/rand"

"time"

)

func boring(msg string, c chan string) {

for i := 0; ; i++ {

c <- fmt.Sprintf("%s %d", msg, i)

time.Sleep(time.Duration(rand.Intn(1e3)) \* time.Millisecond)

}

}

func main() {

c := make(chan string)

go boring("boring!", c)

for i := 0; i < 5; i++ {

fmt.Printf("You say: %q\n", <-c)

}

fmt.Println("You're boring; I'm leaving.")

}

通过 const 关键字来进行常量的定义。

通过在函数体外部使用 var 关键字来进行全局变量的声明和赋值。

通过 type 关键字来进行结构(struct)和接口(interface)的声明。

通过 func 关键字来进行函数的声明。

Go语言中，使用大小写来决定该常量、变量、类型、接口、结构或函数是否可以被外部包所调用。

函数名首字母小写即为 private :

func getId() {}

函数名首字母大写即为 public :

func Printf() {}

go 1.9版本对于数字类型，无需定义int及float32、float64，系统会自动识别。

在 Go 中，布尔值的类型为 bool，值是 true 或 false，默认为 false。

var isActive bool // 全局变量声明

var enabled, disabled = true, false // 忽略类型的声明

func test() {

var available bool // 一般声明

valid := false // 简短声明

available = true // 赋值操作

}

字符串去除空格和换行符

package main

import (

"fmt"

"strings"

)

func main() {

str := "这里是 www\n.runoob\n.com"

str = strings.Replace(str, " ", "", -1)

str = strings.Replace(str, "\n", "", -1)

fmt.Println(str)

}

变量声明（全局变量是允许声明但不使用的，其他必须被用）

package main

var x, y int

var ( // 这种因式分解关键字的写法一般用于声明全局变量

a int

b bool

)

var c, d int = 1, 2

var e, f = 123, "hello" //和python很像,不需要显示声明类型，自动推断

//这种不带声明格式的只能在函数体中出现

//g, h := 123, "hello"

func main(){

g, h := 123, "hello"//出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误

println(x, y, a, b, c, d, e, f, g, h)

}

空白标识符 \_ 也被用于抛弃值，如值 5 在：\_, b = 5, 7 中被抛弃。

\_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

iota 用法

package main

import "fmt"

func main() {

const (

a = iota //0

b //1

c //2

d = "ha" //独立值，iota += 1

e //"ha" iota += 1

f = 100 //iota +=1

g //100 iota +=1

h = iota //7,恢复计数

i //8

)

fmt.Println(a,b,c,d,e,f,g,h,i)

}

指针变量

package main

import "fmt"

func main() {

var a int = 4

var b int32

var c float32

var ptr \*int

fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );

fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );

fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

ptr = &a /\* 'ptr' 包含了 'a' 变量的地址 \*/

fmt.Printf("a 的值为 %d\n", a);

fmt.Printf("\*ptr 为 %d\n", \*ptr);

}

条件语句

switch var1 {

case val1:

...

case val2:

...

default:

...

}

select 语句

select是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。

select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。

以下描述了 select 语句的语法：

每个case都必须是一个通信

所有channel表达式都会被求值

所有被发送的表达式都会被求值

如果任意某个通信可以进行，它就执行；其他被忽略。

如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。

否则：

如果有default子句，则执行该语句。

如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

实例

package main

import "fmt"

func main() {

var c1, c2, c3 chan int

var i1, i2 int

select {

case i1 = <-c1:

fmt.Printf("received ", i1, " from c1\n")

case c2 <- i2:

fmt.Printf("sent ", i2, " to c2\n")

case i3, ok := (<-c3): // same as: i3, ok := <-c3

if ok {

fmt.Printf("received ", i3, " from c3\n")

} else {

fmt.Printf("c3 is closed\n")

}

default:

fmt.Printf("no communication\n")

}

}

以上代码执行结果为：

no communication

循环

package mainimport "fmt"

func main() {

var b int = 15

var a int

numbers := [6]int{1, 2, 3, 5} //数组大小为6， 超过的都是0

for a := 0; a < 10; a++ {

fmt.Printf("a 的值为: %d\n", a)

}

for a < b {

a++

fmt.Printf("a 的值为: %d\n", a)

}

for i,x:= range numbers {

fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)

}

}

无限循环

如过循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

package main

import "fmt"

func main() {

for true {

fmt.Printf("这是无限循环。\n");

}

}

函数调用

package main

import "fmt"

func main() {

var a = 100

var b = 200

var ret int

ret = max(a, b)

fmt.Printf( "最大值是 : %d\n", ret )

}

func max(num1, num2 int) int {

var result int

if num1 > num2 {

result = num1

} else {

result = num2

}

return result

}

多返回值

package main

import "fmt"

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

return y, x

}

func main() {

a, b := swap("Mahesh", "Kumar")

fmt.Println(a, b)

}

引用传递

package main

import "fmt"

func main() {

var a int = 100

var b int= 200

fmt.Printf("交换前，a 的值 : %d\n", a )

fmt.Printf("交换前，b 的值 : %d\n", b )

/\* 调用 swap() 函数

\* &a 指向 a 指针，a 变量的地址

\* &b 指向 b 指针，b 变量的地址

\*/

swap(&a, &b)

fmt.Printf("交换后，a 的值 : %d\n", a )

fmt.Printf("交换后，b 的值 : %d\n", b )

}

func swap(x \*int, y \*int) {

var temp int

temp = \*x /\* 保存 x 地址上的值 \*/

\*x = \*y /\* 将 y 值赋给 x \*/

\*y = temp /\* 将 temp 值赋给 y \*/

}

声明数组

var balance [10] float32

ar balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}

balance[4] = 50.0

package main

import "fmt"

func main() {

var n [10]int

var i,j int

for i = 0; i < 10; i++ {

n[i] = i + 100

}

for j = 0; j < 10; j++ {

fmt.Printf("Element[%d] = %d\n", j, n[j] )

}

}

向函数传递数组

getAverage( balance, 5 )

func getAverage(arr []int, size int) float32 {}

使用指针

package main

import "fmt"

func main() {

var a int= 20

var ip \*int

ip = &a

fmt.Printf("a 变量的地址是: %x\n", &a )

fmt.Printf("ip 变量储存的指针地址: %x\n", ip )

fmt.Printf("\*ip 变量的值: %d\n", \*ip )

}

空指针

package main

import "fmt"

func main() {

var ptr \*int

fmt.Printf("ptr 的值为 : %x\n", ptr )

}

以上实例输出结果为：ptr 的值为 : 0

空指针判断：

if(ptr != nil) /\* ptr 不是空指针 \*/

if(ptr == nil) /\* ptr 是空指针 \*/

结构体

package main

import "fmt"

type Books struct {

title string

author string

subject string

book\_id int

}

func main() {

var Book1 Books

Book1.title = "Go 语言"

Book1.author = "www.runoob.com"

Book1.subject = "Go 语言教程"

Book1.book\_id = 6495407

fmt.Printf( "Book 1 title : %s\n", Book1.title)

fmt.Printf( "Book 1 author : %s\n", Book1.author)

fmt.Printf( "Book 1 subject : %s\n", Book1.subject)

fmt.Printf( "Book 1 book\_id : %d\n", Book1.book\_id)

}

作为参数

func printBook( book Books ) {}

接口

package main

import (

"fmt"

)

type Phone interface {

call()

}

type NokiaPhone struct {

}

func (nokiaPhone NokiaPhone) call() {

fmt.Println("I am Nokia!")

}

type IPhone struct {

}

func (iPhone IPhone) call() {

fmt.Println("I am iPhone!")

}

func main() {

var phone Phone

phone = new(NokiaPhone)

phone.call()

phone = new(IPhone)

phone.call()

}

error处理

====

result, err:= Sqrt(-1)

if err != nil {

fmt.Println(err)

}

===

这段很经典 理解后对理解go有益

package main

import (

"fmt"

)

type DivideError struct {

dividee int

divider int

}

// 实现 `error` 接口

func (de \*DivideError) Error() string {

strFormat := ` Cannot proceed, the divider is zero.

dividee: %d

divider: 0`

return fmt.Sprintf(strFormat, de.dividee)

}

// 定义 `int` 类型除法运算的函数

func Divide(varDividee int, varDivider int) (result int, errorMsg string) {

if varDivider == 0 {

dData := DivideError{

dividee: varDividee,

divider: varDivider,

}

errorMsg = dData.Error()

return

} else {

return varDividee / varDivider, ""

}

}

func main() {

// 正常情况

if result, errorMsg := Divide(100, 10); errorMsg == "" {

fmt.Println("100/10 = ", result)

}

// 当被除数为零的时候会返回错误信息

if \_, errorMsg := Divide(100, 0); errorMsg != "" {

fmt.Println("errorMsg is: ", errorMsg)

}

}

简易rest 服务器

package main

import "net/http"

func SayHello(w http.ResponseWriter, req \*http.Request) {

w.Write([]byte("Hello"))

}

func main() {

http.HandleFunc("/", SayHello)

http.ListenAndServe(":3000", nil)

// TODO

}

访问 localhost:3000, 注意：不是 http:\\localhost:3000 (注意端口不能被事先占用，否则会立马返回)

Client 端（访问百度）

package main

import (

"flag"

"fmt"

"log"

"time"

"net/http"

"io/ioutil"

)

func connectServer() {

client := &http.Client{}

url := "http://www.baidu.com"

reqest, err := http.NewRequest("GET", url, nil)

if err != nil {

panic(err)

}

response, \_ := client.Do(reqest)

defer response.Body.Close()

body, \_ := ioutil.ReadAll(response.Body)

body\_str := string(body)

fmt.Println(body\_str)

status := response.StatusCode

fmt.Println(status)

}

func checkError(err error) {

if err != nil {

log.Fatal("an error!", err.Error())

}

}

func main() {

var goCount \*int

goCount = flag.Int("goCount", 0, "goroutine number")

flag.Parse()

fmt.Println("go count = ", \*goCount)

tInsert := time.Now()

fmt.Println("tStart time: ", tInsert)

for i := 0; i < goCount; i++ {

fmt.Println("goroutine number: ", i)

connectServer()

}

elapsed := time.Since(tInsert)

fmt.Println("Time elapsed: ", elapsed)

}

go 并发测试

简答示例

package main

import "time"

func main() {

//startOneClient(2)

var i = 0

for j:=0;j<200000;j++{

go ForTest(&i)

}

time.Sleep(20 \* time.Second)

}

package main

import "fmt"

func ForTest(i \*int) {

\*i++

if \*i%10000 == 0 {

fmt.Println("testing ... %d",\*i)

}

}

==

完整示例

func startOneClient(goCount int){

tInsert := time.Now()

for i := 0; i < goCount; i++ {

//fmt.Println("goroutine number: ", i)

connectServer()

}

elapsed := time.Since(tInsert)

fmt.Println("Time elapsed: ", elapsed)

}

func main() {

//起5个client 每个请求2次

for i:=0;i<5;i++{

go startOneClient(2)

}

time.Sleep(20 \* time.Second)

}

挂起暂停的好方法

quit := make(chan bool)

<-quit

==

测试cpu加法

package main

import (

"time"

"fmt"

)

const g = 100000000

func getCounts(count \*int) {

for {

start := \*count

time.Sleep(time.Second \* 1)

fmt.Println("counts for 1s : ", \*count-start)

}

}

func main() {

var count = 0

go getCounts(&count)

for count=0;count<1000\*g;count++{

}

quit := make(chan bool)

<-quit

}

Go的class 面向对象

package main

import (

"math"

"fmt"

)

type Point struct { //定义一个结构题体，你可以理解是是Python中的class

X,Y float64

}

func (p Point)Distence(q Point) float64 { //给p对象定义一个Distence的方法，你可以理解绑定了一个Distence的方法。

return math.Hypot(q.X-p.X,q.Y-p.Y)

}

func main() {

p := Point{1,2}

q := Point{4,6}

fmt.Println((p.Distence(q))) //类的调用方式，注意，如果定义就要如何调用！（这里是调用p的Distence方法。）

}

Select 用法 求值顺序

//select语法：

//每个case都必须是一个通信

//所有channel表达式都会被求值 （重要！：求值顺序：自上而下、从左到右 ）

//所有被发送的表达式都会被求值

//如果任意某个通信可以进行，它就执行；其他被忽略。

//如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。

//否则：

//如果有default子句，则执行该语句。

//如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

chanCap := 5

ch := make(chan int, chanCap) //创建channel，容量为5

for i := 0; i < chanCap; i++ { //通过for循环，向channel里填满数据

select { //通过select随机的向channel里追加数据

case ch <- 1:

case ch <- 2:

case ch <- 3:

}

}

Go 包的管理

建议把主程序放在 %GOPATH%/src下

Go文件名没有什么关系 ，主要是包名，一个文件夹下的文件只能一个package ，文件夹名就是package名

外层调用函数的话直接就是 package.func()

panic 会报错，打断程序运行。

但是不会阻止defer运行

http 请求 回答解析json

client := &http.Client{}

//url := "http://120.79.226.163:1317/keys"

url2 := "http://120.79.226.163:1317/accounts/cosmosaccaddr1rj740ua4mj0hd3az8md75pd70fht9wl2xfyr66/send"

type AmountItem struct {

Amount int `json:"amount"`

Denom string `json:"denom"`

}

type Data struct {

AccountNumber int `json:"account\_number"`

Amount []AmountItem `json:"amount"`

ChainId string `json:"chain\_id"`

Name string `json:"name"`

Password string `json:"password"`

Sequence int `json:"sequence"`

Gas int `json:"gas"`

}

data := Data {

AccountNumber : 15,

Amount : []AmountItem{{Amount:1, Denom: "iris"}},

ChainId : "fuxi-1002",

Name : "shelwin",

Password : "1234567890",

Sequence : 470,

Gas : 10000,

}

//fmt.Println(data)

b , err := json. Marshal ( data )

if err != nil {

fmt.Println("err != nil ")

}

//fmt.Println(string(b))

reqest, err := http.NewRequest("POST", url2, bytes.NewBuffer(b))

//reqest, err := http.NewRequest("GET", url2, nil)

if err != nil {

fmt.Println("err != nil ")

}

response, \_ := client.Do(reqest)

defer response.Body.Close()

body, \_ := ioutil.ReadAll(response.Body)

body\_str := string(body)

fmt.Println(body\_str)

status := response.StatusCode

fmt.Println(status)

--------------------

解析 （并不是解析上面的哦）

body := s.RequestWorker.MakeRequest(nil)

data := &[]InquiryAccountListResult{}

err := json.Unmarshal(body, data)

-----------------------

Test 并行 goconvey

Test 并行 goconvey

go test -parallel 4

go get github.com/360EntSecGroup-Skylar/excelize （excel表格）

----

Interface 传递不同类型的参数

// Route and Run cases

func route(in interface{}){

switch in.(type) {

case \*stories.QueryAccountList:

x:= in.(\*stories.QueryAccountList)

case string:

case int8, int32, int64:

case float64:

}

}

story := &stories.QueryAccountList{}

route(story)