The Go Programming Language

李晓海
联系方式：18612372586@126.com

Go Lang，下一代的C？

Agenda

- ***Go lang 基础***
  - Go Lang面向对象
  - Go lang 并发模式
  - Go lang WEB开发

Go Lang

- 为什么选择Go Lang
- Go Lang 环境设置
- Go Lang语法基础
- 面向对象
- 并发编程
- 反射

为什么？

- 天时
  - 应用的发展。
    - 并行
    - 大数据
    - 移动互联网
  - 生产力的要求
  - Java已经垂垂老矣。
  - 新的语言还不能担当。（python、ruby、erlang…）
  - 老的语言能否焕发青春？（c/c++）
- 地利
  - goLang 的特性
- 人和
  - 简单
  - 类C的语法
  - …

新语言期望

- 速度快，高性能。
- 简单明了，需要记忆的语言细节少，开发迅速。
- 灵活。
- 完善的模块支持。
- 程序员友好的并行架构。
- 安全，绝大部分在编译期解决。
- OOP,函数式编程
- …

Go Lang 来源

- Google 出品
- 开发团队

Ken Thompson：1983年图灵奖（Turing Award）和1998年美国国家技术奖（National Medal of Technology）得主。他与Dennis Ritchie是Unix的原创者。Thompson也发明了后来衍生出C语言的B程序语言。

Rob Pike：曾是贝尔实验室（Bell Labs）的Unix团队和Plan 9操作系统计划的成员。他与Thompson共事多年，并共创出广泛使用的UTF-8 字元编码。

Robert Griesemer：曾协助制作Java的HotSpot编译器，和Chrome浏览器的JavaScript引擎V8。

此外还有Plan 9开发者Russ Cox、和曾改善目前广泛使用之开原码编译器GCC的Ian Taylor。

Go Lang历史

- 1969年贝尔实验室开发出unix，以及unix衍生语言C
- 1980s 上述团队研发出Plan9（OS）
- 以后几十年 Plan9衍生出Inferno子项目以及编程语言Limbo（Go Lang前身，小型计算机上的分布式应用语言）
- Plan9项目组加入Google.
- 2007.9,Go Lang实验项目
- 2009.11 正式发布
- 2012.3.28 Go Lang 第一个版本发布。

Go Lang 特点

- 保留但大幅度简化指针
- 多参数返回
- array slice map等内置基本数据结构
- 错误处理（ panic recover error ）
- interface
- OO
- Goroutine

多核处理和网络开发

- 更多现代特性

原生支持unicode，垃圾收集，部分函数式编程（匿名函数、闭包），反射，语言交互性

Go Lang特点

- 严格到苛刻的编码风格
- 软件工程原生支持
- Package即目录
- 原生字节码
- 和C原生调用（CGO）
- ***简单、实用***

GoLang编程哲学

- 常见编程范式
  - 面向过程（C）
  - 面向对象（Java C#）
  - 面向消息（Erlang）
  - 函数式（Haskell、Erlang）
- GoLang什么都不是！
  - 一切以***实用***出发，吸取以上以上语言的概念。
  - 多范式语言
  - 基于连接和组合的语言

Go Lang编程哲学

- 连接：组件的耦合方式，组件是如何被连接起来的
- 组合：形成复合对象的基础。

适应场景和竞争对手

- 多核并行网络编程
  - Erlang,node.js,c,java
- 系统编程
  - Java,c,c++,shell
- Web编程
  - Java,python,ROR,php
- 业务系统
  - Java .net
- 客户端
  - Java，.net,QT
- 移动客户端，还不支持，将来可预期

缺点

- 年轻
  - 不成熟
  - 组件少
- 缺少杀手级应用
- 不支持动态链接库
  - 没有class.forName
- IDE功能弱

GoLang 基础

- 安装
- 变量
- 类型
- 流程控制
- 函数
- defer
- panic/recover
- 代码结构
- 测试
- 并发
- 面向对象

安装

- 下载安装包

https://code.google.com/p/go/downloads/list

- 安装配置环境变量

GOROOT=/usr/local/go

GOPATH=$GOROOT/bin

- 配置IDE
  - Eclipse插件
  - Idea插件
  - **sublime/notepad++ 插件**
  - **LiteIDE**

验证安装

- go version
- Hello world

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

Go run hello.go

文档和资源

- 安装本地go doc(神奇的墙！)

sudo godoc –http=:6060

http://127.0.0.1/6060

资源：

http://studygolang.com/

http://www.golang.tc/

- 邮件组

https://groups.google.com/d/forum/golang-china

https://groups.google.com/d/forum/golang-nuts

变量

- *变量声明*
  - var v int
  - var v[3] int
  - var v[] int
  - var s struct{}
  - var v *int
  - var f func(a int) int
  - var (
    v1 int
    v2 float
)

变量初始化

- var i int =10//最完整定义
- var i =10//编译器自动推导类型
- i:=10 //编译器自动推导类型

采用第三种方法，需要保证i没被声明过，因为它等同与1

变量赋值

var k int
k=10

多重赋值

i,j = j,i

常量

```
const PI float64=3.1514926
```

预定义常量：true/false,iota

```
const (
  v1= iota   //=0
  v2= iota   //1
  v3 =iota              //2
)
const (
  v1 =1<<iota   //1=1<<0
  v2        //2      =1<<1
  v3        //4=1<<2
)
```

枚举

```
const(
  Sundy= iota
  Monday
  Tuesday
  Wednesday
  Thursday
  Friday
  Saturday
 numberOfDays
)
```

保留字

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthroy gh	if	range	type
continue	for	import	return	var

类型—基础类型

- 布尔类型：bool
- 整形：int8,byte,int16,int,uint,uintptr
- 浮点： float32, float64
- 复数：complex64,complex128
- 字符串：string
- 字符：rune
- 错误：error

类型—复合类型

- 指针 pointer
- 数组 array
- 切片 slice
- 字典 map
- 通道 chan
- 结构 struct
- 接口 interface

数组

- v：= [10] int//必须指定长度
- 数组遍历
  - for i:=0;i<len(v);i++{}
  - for i,v:= range v{}
- 值类型，每次数组传递的时候都会发生数组赋值

```
func modifyArray(array [10]int) {
    array[0] = 10
    fmt.Println("In modify(), array values", array)
}
```

数组切片

- v:=[] int//类似java的vector
- 创建切片
  - 基于数组

arr:=[10]int {1,2,3,4,5,6,7,8,9,10}

slice1 [] int:= arr[:5]

slice2 [] int:= arr[5:]

  - 直接创建

s1:= make([]int,5)

s2:= make([]int,5,10)//长度5，预分配空间10

s3:= []int{1,2,3,4,5}

- 遍历切片

参照数组

动态增加元素

s1:=make([]int,5,10)
//len(s1)=5
//cap(s2)=10

s1=append(s1,1,2,3)
s2:=[]int{1,2,3}
s1=append(s1,s2…)//必须加“…”

s3:=[3]int{1,2,3}
s1=append(s1,s3…)
//错误，不能将数组直接append至数组切片中

們嚷兗偢

s1 := []int{1, 2, 3, 4, 5}

s2 := []int{7, 8, 9}

//把S1复制到S2中，由于S2较短，会把S1的和S2相同长度的数据复制并且替换到S2中

copy(s2, s1)//此处S1不变，S2=[1,2,3]

//把S2复制到S1中，由于S2较短，会把S2全部赋值到S1中

copy(s1, s2) //S2不变，S1=[7,8,9,4,5]

map

```
type Person struct{
Id string
Name string
}
var personDB map[string] Person//变量声明
personDB = make(map[string] person)//创建
personDB = make(map[string] person,10)//创建并且分配空间
personDB["1"]=Person{"1","lee"}//赋值
personDB1 =map[string]Person{"1",{"1","lee"}}//创建并且赋值
person,ok:=personDB["1"] //查询ok表示是否查到
if ok {}else{}
delete(personDB,"1")//删除元素
```

流程控制

- 条件 if else else if
- 婷厓 switch case select
- 循环 for range
- 跳转 goto

条件

```
if a<5{
}else{
}
```

- If后不使用括号
- {}必须要有
- {必须要和if 或else在一行

选择

```
switch l {
case 0:
Default:
}
k := 2
switch {
case 0 < k:
   fmt.Println("111")
   fallthrough
case k < 10:
 fmt.Println("222")
}
```

{必须和switch一行
条件表达式不限为常量或整数
单个case中可以出现多个结果选项
不需要break来退出case
只有在case中添加fallthrough，才会继续执行下一个case，并且下一个case不经过判断，直接执行
可以不设定switch之后的条件表达式，此时，和多个if else相同

循环

for

```
for i:=0;i<10;i++{}
for{
  if XXX {break}
}
```

跳转

```
func aa(){
  i:=0
  HERE:
  fmt.Println()
  i++
  if i<10{
    goto HERE
  }
}
```

函数

- 函数定义

```
func (接收者—类/接口)函数名 (参数) 返回值{
        函数体
        返回值
}
```

- 不定参数

```
func foo(args ...int){}  foo(1,2,3) foo([]int{1,2,3}...)
func foo1(args []int){}  foo1([]int{1,2,3})
```

函数

- 任意类型

```
func foo(format string,args ...interface{}){}
```

- 多返回值

```
func (file *File) Read(b []byte)(n int,err error){}
_,err:=read(buf)
```

- 偠僊個呪僗媄偞

```
add:=func(x,y int)int{return x+y}
x := func(x, y int) int {
        return x + y
}(2, 3)
 fmt.Println("2+3=", x)
```

```
add:=func(x, y int) int {
        return x + y
}
x:= add(2,3)
fmt.Println("2+3=", x)
```

函数—闭包

```
func main(){
j := 5
    a := func() func() {
        ii := 10
        return func() {
            fmt.Println("i,j =", ii, j)
            fmt.Println()
        }
    }()
    a()
    j *= 2
    a()
}
```

```
b := func() {
        ii := 10
        fmt.Println("i,j =", ii, j)
        fmt.Println()
}
b()
j *= 2
b()
```

函数-闭包

- 输出

I,j = 10,5

I,j= 10,10

- 变量a指向的闭包引用了局部变量i和j，i的值被隔离，在闭包外面不能被修改，改变j的值后，再次调用a，结果是j已经改变。
- 在变量a指向的闭包函数中，只有内部的你们函数才能访问变量a，无法通过别的方式访问，从而保证i的安全性

错误处理

- error 接口

```
type error interface{
  Error() string
}
```

- 大部分场景下如果要返回错误，一般如下定义：

```
func Foo(para string)(n int,err error){}
n,err:=Foo(“111”)
if err!=nil{}else
```

defer

- 类似于java的finally

```
func CopyFile(dest,src string)(w int, err error){
  srcFile,err:=os.Open(src)
  if err!=nil{
     return
  }
  defer srcFile.Close()
  destFile,err:=os.Create(destFile)
  defer func(){destFile.Close}()
  return io.Copy(destFile,srcFile)
}
```

panic() 和recover()

- 处理运行期错误和程序错误

```
func panic(c interface{})
func recover() interface{}
```

panic() 和recover()

- *Panic*

是一个内建函数,可以中断原有的控制流程,进入一个令人恐慌的流程 中。当函数 F 调用 **panic,函数** F 的执行被中断,并且 F 中的延迟函数会正 常执行,然后 F 返回到调用它的地方。在调用的地方,F 的行为就像调用 了 **panic。这一过程继续向上,直到程序崩溃时的所有** goroutine 返回。

恐慌可以直接调用 panic 产生。也可以由运行时错误产生,例如访问越界的数组。

- *Recover*

是一个内建的函数,可以让进入令人恐慌的流程中的 goroutine 恢复过 来。recover 仅在延迟函数中有效。

- 一定要记得,这应当作为最后的手段被使 用,你的代码中应当没有,或者很少的令人恐慌的东西。

panic() 和recover()

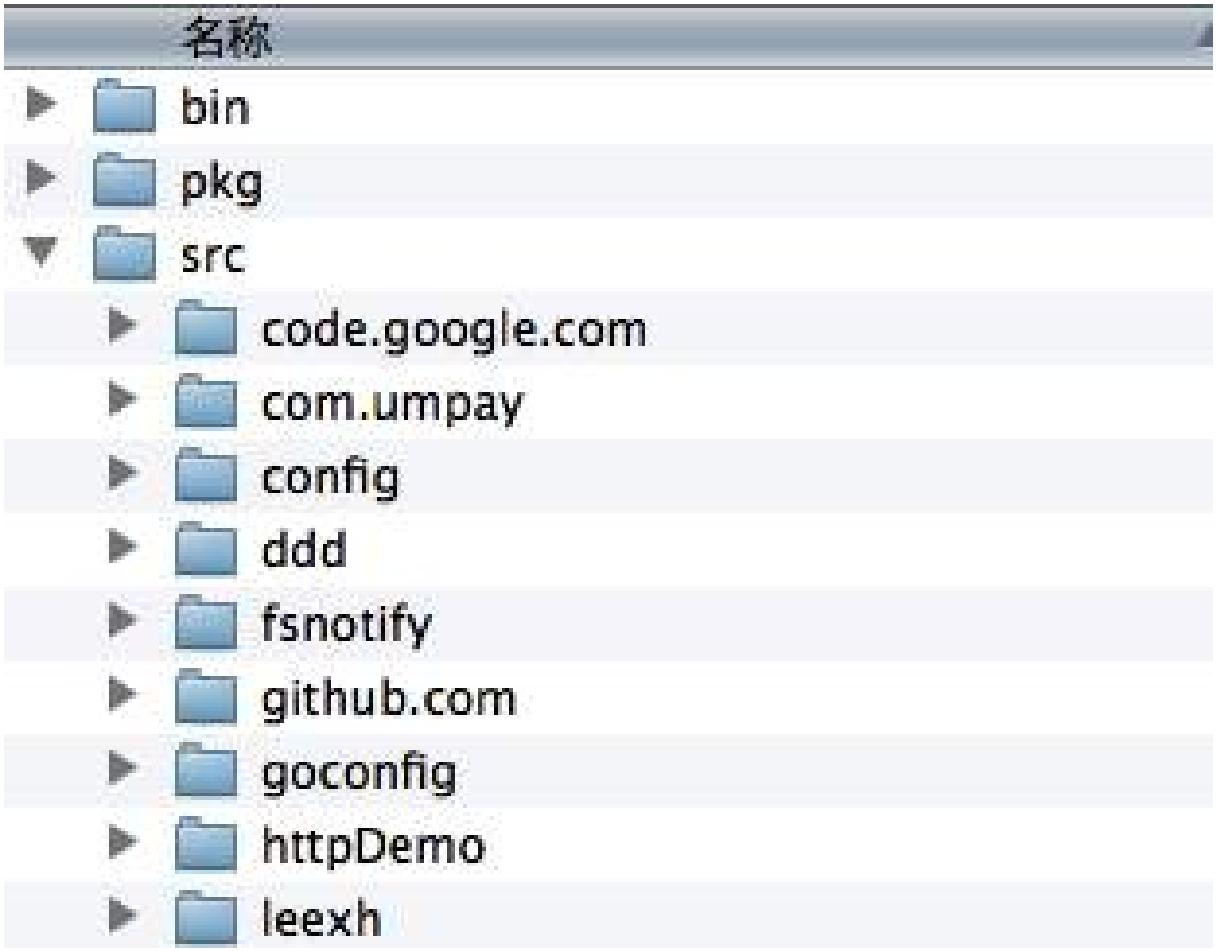
```
var user = ""

func inita() {
  defer func() {
    fmt.Println("defer###")
  }()
  if user == "" {
    fmt.Println("@@@before panic@@@")
    panic("no value for user!")
    fmt.Println("after panic")
  }
}
func throwsPanic(f func()) (b bool) {
  defer func() {
    if x := recover(); x != nil {
      fmt.Println(x)
      b = true
    }
  }()
  f()
  fmt.Println("after the func run.")
  return
}
func main() {
  throwsPanic(inita)
}
```

输出：

@@@before panic@@@defer#

目录结构


名称
bin
pkg
src
code.google.com
com.umpay
config
ddd
fsnotify
github.com
goconfig
httpDemo
leexh

目录结构

- bin

  编译打包生成的可以直接运行的应用程序

- pkg

  编译形成的程序包 .a文件

- src

  源代码：公司域名/项目/package/源文件或者配置文件

  一般公司域名为umpay.com形式。并且是一个单独的目录

Package

- 函数和数据的集合。
- 文件名不一定和包同名。
- 一个包下面可以有多个文件。
- 类或者方法存在哪个文件中无所谓。
- Go 查询包的时候是从$GOPATH/src下开始找，

如果你的项目在$GOPATH/src下，则import 國旱侉

:

import “嫕圚傞/偞傞”

package

- 包的文档

每个包都应该有包的注释，注释放在package之前。针对包含多文件的包，注释放在任何文件都可以，一般单独定义一个doc.go放置包的注释。

- Go doc 查看文档

Test

- go test 运行所有test类
- 文件名_test.go,和被测文件相同目录（为了测试私有方法—go 的私有方法、变量为包内可见）
- func TestXXX(t *testing.T)
- testing 的常用方法
  - func (t *T)Fail():标识失败，但是会继续执行
  - func (t *T)FailNow():标识失败，中断执行
  - func (t *T)Log (args ...interface{}):记录日志
  - func (t *T)Fatal (args ...interface{}):=Log+FailNow

test

```
import (
        "testing"
)
func TestEven(t *testing.T) {
        if !Even(2) {
                t.Log("2 should be a even number!")
                t.Fail()
        }
}
```

常用标准包

- fmt：实现了格式化的 I/O 函数,这与 C 的 printf 和 scanf 类似。格式化短 语派生于 C 。
- io：这个包提供了原始的 I/O 操作界面。它主要的任务是对 os 包这样的原始的 I/O 进行封装,增加一些其他相关,使其具有抽象功能用在公共的接口 上。
- bufio：实现了缓冲的 I/O。它封装于 io.Reader 和 io.Writer 对象,创建了另 一个对象(Reader 和 Writer)在提供缓冲的同时实现了一些文本 I/O 的功 能。
- sort：提供了对数组和用户定义集合的原始的排序功能。
- strconv：提供了将字符串转换成基本数据类型,或者从基本数据类型转 换为字符串的功能。
- os：提供了与平台无关的操作系统功能接口。其设计是 Unix 形式的。
- sync：提供了基本的同步原语,例如互斥锁。
- flag：实现了命令行解析。

常用标准包

- encoding/json：实现了编码与解码 RFC 4627 [5] 定义的 JSON 对象。
- text/template：数据驱动的模板,用于生成文本输出,例如 HTML。将模板关联到某个数据结构上进行解析。模板内容指向数据结构的元素 (通常结构的字段或者 map 的键)控制解析并且决定某个值会被显示。模板扫描结构以便解析,而 “游标” @ 决定了当前位置在结构中的值。
- net/http： HTTP 请求、响应和 URL 的解析,并且提供了可扩展的 HTTP 服务和基本的 HTTP 客户端。
- unsafe：包含了 Go 程序中数据类型上所有不安全的操作。 通常无须使用 这个。
- reflect：实现了运行时反射,允许程序通过抽象类型操作对象。通常用于 处理静态类型 interface{} 的值,并且通过 Typeof 解析出其动态类型信息, 通常会返回一个有接口类型 Type 的对象。
- os/exec：执行外部命令。

To be continued...