注：这篇文档主要记录一些对go语言初步学习的理解，包括一些基本语法和一些要注意的地方

包

每个Go程序都是由包构成的，每个Go程序有且只有一个main包
（因为每个Go程序都是从main包开始运行的，也就是起始地址）
创造main包用以下命令：

- package main

导入包用以下命令(一个包和多个包的情况）：

- import "fmt"
- import(
  "fmt"
  "math"
)

导入包的作用当然是引用包中的函数啦，通称叫做导出名
导出用以下命令:

- fmt.Printf()

注意，在Go程序从包中导出函数时，用大写字母开头表示已导出。也就是说在fmt包中函数名称是printf，在引用时要使用Printf。

函数

https://blog.go-zh.org/gos-declaration-syntax
这个网站的内容详细的说明了为什么声明函数和变量时，是先声明变量再确定类型的。
总结一下那个网站的内容：比常规的c语言语法在相当复杂的声明情况下更易于理解和清晰。
我认为这个用法是没啥意义的。因为避免相当复杂的声明已经是每一个程序员应该做的守则。

函数的调用如下：

- func add(x, y int) int {
- return x + y
- }

如果x,y类型相同，可以简写成x,y int

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

函数可以返回多个参数

- func split(sum int) (x, y int) {
- x = sum * 4 / 9
- y = sum - x
- return
- }

可以对返回的参数命名，这样return时可以不写返回的变量名称
参数可以是函数
函数可以闭包（但是我个人也不推荐使用，所以不写在这里）

变量

- var
  var 语句用于声明一个变量列表，跟函数的参数列表一样，类型在最后,声明变量必须跟var
  var i, j int = 1, 2
- :=
  在函数中，简洁赋值语句 := 可在类型明确的地方代替 var 声明,注意，:=只能用在变量名未声明的情况下。
  k,j := 3,4
  但是不能用在函数外

Go 的基本类型有
bool
string
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
byte // uint8 的别名
rune // int32 的别名
// 表示一个 Unicode 码点
float32 float64
complex64 complex128
每一个类型都有对于的类型转换函数
比如var f float64 = float64(i)

常量

常量的声明与变量类似，只不过是使用 const 关键字。

常量可以是字符、字符串、布尔值或数值。

常量不能用 := 语法声明。
用法如下：

- const Pi = 3.14

for循环

基本语法与c语言基本一致，就是不用加括号
用法如下:

- for i := 0; i < 10; i++ {
  sum += i
  }
- for sum < 1000 {
  sum += sum
  }
- for {
  }
  如果是没有循环条件，默认为无限循环

if

基本语法同c，不用加小括号

- if x < 0 {
  return sqrt(-x) + "i"
  }
- if v := math.Pow(x, n); v < lim {
  return v
  }
  特殊语法，允许声明变量，作用域只在if内部
- if v := math.Pow(x, n); v < lim {
  return v
  } else {
  fmt.Printf("%g >= %g\n", v, lim)
  }

switch

语法跟c类似，也不需要小括号,并且自带break，不需要我们写break

- 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

新特性，延迟运行，将函数推迟到外层函数返回之后执行，注，只能延迟函数，函数的参数会在外层函数运行时求值，但是函数在外层函数结束后才运行。

- func main() {
  defer fmt.Println("world")
  defer fmt.Println("hah?")
  fmt.Println("hello")
  }

多个defer延迟函数，每个函数都会压入栈中，因此多个defer函数运行结果符合“先进后出，后进先出”原则，上式运行结果是

- hello
  hah?
  world

指针

go语言也支持指针，用法与c一样，但是go语言没有指针的运算，用法如下：

- p := &i // 指向 i
  fmt.Println(*p) // 通过指针读取 i 的值
  *p = 21 // 通过指针设置 i 的值
  fmt.Println(i)

结构体

一个结构体（struct）就是一组字段（field），
使用方法如下：

- type Vertex struct {
  X int
  Y int
  }
  var j Vertex = Vertex{1,2} //用大括号来括起来
  fmt.Println(j.X)
  j := Vertex{} //未初始化时
  var j = Vertex{X:1,Y:2}//显示赋予也可

结构体也有指针

- v := Vertex{1, 2}
  p := &v
  p.X = 1e9
  (*p).X = 1
  用(*指针).X引用值
  支持隐式引用，即可以直接用(指针).X来引用值

由于go语言没有方法，因此结构体允许内置函数（方法）
用法如下：

- 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) Abs() float64 {
  return math.Sqrt(v.X * v.X + v.Y * v.Y)
  }
  在调用结构的方法时，无论是结构数值还是指针都可以，go语言会帮你自动解释，唯一的区别在于函数中的参数如果是指针，则会修改原值，如果是数值，则不会修改原值。
- v := Vertex{1,2}
- v.abs()
- (&v).abs()

数组

类型 [n]T 表示拥有 n 个 T 类型的值的数组
数组不能改变大小，使用方法如下：

- var a [10]int
- a[0]= 1
- b := [2]int{1,2}

虽然数组不能改变大小，但是切片为提供了灵活的使用。

- var s []int = a[low:high]

切片是半开区间（左闭右开），声明切片时，并不是将数组的内容在内存中拷贝一份，而是描述了底层数组的一部分，也就是说，所有对切片的修改都会反映到底层数组上

- var a [2]int = [2]int{1,2}
- b := a[:]
- b[0]= 2
  此时a的内容为{2,2}

切片有长度（len）和容量（cap）的属性
切片的长度就是它所包含的元素个数
切片的容量是从它的第一个元素开始数，到其底层数组元素末尾的个数
切片 s 的长度和容量可通过表达式 len(s) 和 cap(s) 来获取

可以用make()来制造切片，动态数组

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

可以用append()来为切片添加内容

- b = b.append(b,1,2,...)

切片的详细内容可以参考
https://blog.go-zh.org/go-slices-usage-and-internals
讲的很好

映射

用法如下：

- var m map[string]int
- m = make(map[string]int)//创建映射类型.
- m["hello"]= 1
- elem := m["hello"]

- delete(m,"hello")//删除

接口

接口也是之前c语言没有的特性。
接口类型 是由一组方法签名定义的集合

- type I interface {
  M()
  }
  type T struct {
  S string
  }
  // 此方法表示类型 T 实现了接口 I，但我们无需显式声明此事。
  func (t T) M() {
  fmt.Println(t.S)
  }
  func main() {
  var i I = T{"hello"}
  i.M()
  }
  若有不同的结构实现了同一个名称（M()）的函数，那么不同的结构使用接口会使用对应的函数。

错误

Go 程序使用 error 值来表示错误状态。

error 类型是一个内建接口：

type error interface {
Error() string
}

使用时：

- package main
  import (
  "fmt"
  "math"
)
  type ErrNegativeSqrt float64
  func (e ErrNegativeSqrt) Error() string{

```
return fmt.Sprintf("cannot Sqrt negative number: %v",float64(e))
}
func Sqrt(x float64) (float64, error) {
if x < 0{
return 0,ErrNegativeSqrt(x)
}
return math.Sqrt(x),nil
}
func main() {
fmt.Println(Sqrt(2))
fmt.Println(Sqrt(-2))
}
```

go多线程

Go 程（goroutine）是由 Go 运行时管理的轻量级线程。
go f(x, y, z)
会启动一个新的 Go 程并执行
f(x, y, z)
f, x, y 和 z 的求值发生在当前的 Go 程中，而 f 的执行发生在新的 Go 程中。

信道（go线程的重要概念）

信道是带有类型的管道，你可以通过它用信道操作符 <- 来发送或者接收值。
ch <- v // 将 v 发送至信道 ch。
v := <-ch // 从 ch 接收值并赋予 v。
（“箭头”就是数据流的方向。）
和映射与切片一样，信道在使用前必须创建：
ch := make(chan int)
默认情况下，发送和接收操作在另一端准备好之前都会阻塞。这使得 Go 程可以在没有显式的锁或竞态变量的情况下进行同步。

- ch := make(chan int, 2)
  ch <- 1
  ch <- 2
  fmt.Println(<-ch)
  fmt.Println(<-ch)
  当ch中数量达到2时，会阻塞，再填充数据则会抱错

range 和 close

发送者可通过 close 关闭一个信道来表示没有需要发送的值了。接收者可以通过为接收表达式分配第二个参数来测试信道是否被关闭：若没有值可以接收且信道已被关闭，那么在执行完

v, ok := <-ch
之后 ok 会被设置为 false。

循环 for i := range c 会不断从信道接收值，直到它被关闭。

注意： 只有发送者才能关闭信道，而接收者不能。向一个已经关闭的信道发送数据会引发程序恐慌（panic）。

还要注意： 信道与文件不同，通常情况下无需关闭它们。只有在必须告诉接收者不再有需要发送的值时才有必要关闭，例如终止一个 range 循环。

select

select 语句使一个 Go 程可以等待多个通信操作。

select 会阻塞到某个分支可以继续执行为止，这时就会执行该分支。当多个分支都准备好时会随机选择一个执行。

- package main
  import "fmt"
  func fibonacci(c, quit chan int) {
  x, y := 0, 1
  for {
  select {
  case c <- x:
  x, y = y, x+y
  case <-quit:
  fmt.Println("quit")
  return
  }
  }
  }
  func main() {
  c := make(chan int)
  quit := make(chan int)
  go func() {
  for i := 0; i < 10; i++ {
  fmt.Println(<-c)
  }

```
quit <- 0
}()
fibonacci(c, quit)
}
```

互斥

sync.Mutex
我们已经看到信道非常适合在各个 Go 程间进行通信。

但是如果我们并不需要通信呢？比如说，若我们只是想保证每次只有一个 Go 程能够访问一个共享的变量，从而避免冲突？

这里涉及的概念叫做 *互斥（mutual*exclusion）*，我们通常使用 *互斥锁（Mutex）* 这一数据结构来提供这种机制。

Go 标准库中提供了 sync.Mutex 互斥锁类型及其两个方法：

Lock
Unlock
我们可以通过在代码前调用 Lock 方法，在代码后调用 Unlock 方法来保证一段代码的互斥执行。参见 Inc 方法。

我们也可以用 defer 语句来保证互斥锁一定会被解锁。参见 Value 方法