# GO语法快速入门

B站视频

## 变量

```
var num int // 声明
num = 10 // 定义
var num int = 10 // 声明+定义

num := 10 // := 自动类型推导，声明+定义
```

## 结构体

```
type Student struct {
    //'S'大写表明引用该程序所在package后即可访问该字段，若小写在只能在相同package程序中使用
    Name string
    Id int
}

s1 := Student{"xxx",20}
s2 := &Student{"zzz",19} // 通常是得到一个指针，减少内存开销
// 事实上golang在用户使用的时候会转换格式
// s1.Name s2.Name都可以读取到对应的值
```

## 条件循环语句

```go
// golang没有while关键字，只有用for
for {

} // 无限循环

// if后可以先跟一段定义字段在分号后再直接使用，
// 但是注意，此时的val,ok都是局部变量，在if语句后便不能被访问
if val,ok := myMap["hello"]; !ok {
    fmt.Println("not exist")
}else {
    fmt.Println(val)
}
```

## 字符串

```go
// golang中的字符串不能被更改 如myString[0] = 'H'是不允许的
// 另外注意中文字符占3个字节，
// 在使用字符串时，尤其是遍历的时候，需要用%c进行转换，不然就是一串数字了
myString := "hello我"
len(myString) // 结果是 8 (5+3)
for _, v := range myString {
    fmt.Printf("%c\n", v) // 等价于fmt.Println(string(v))
}

// 利用[]rune可以避免一些误会 因为rune就是int32,每个字符都打包成int32了，位置足够
myRune := []rune(myString)
println(len(myRune)) //结果是 6
```

## 容器

```go
// Go容器中有slice,map等

// slice 切片
// slice实际上是一个拥有len和cap的指针，指向一段连续的内存区域
// 可以将其理解成一个窗口，用make创建
type Slice struct {
```

```
    P
    len
    cap
}

mySlice := make([]int,1,10)
// 一定注意slice的扩容现象，可能导致指向内存的位置发生改变，从而访问不到原来的数据
// 同时，slice的扩容是将原本的内容复制拷贝到另一片更大的区域，因此也要注意时间开销

mySlice = append(mySlice,1) // 注意，现在的mySlice指向的空间的数据是[0,1]


// map
// 存储k-v键值对，用make创建
myMap := make(map[string]int) // 表示string->int
val,ok := myMap["xxx"]

// channel
// 像一个FIFO环，用make创建

// 1. 有一个channel
// 2. 有一个goroutine，每隔1秒向channel发送一个数字
// 3. 有一个goroutine，从channel中读取数据，打印出来
// 4. 为了保证main函数不退出，让main函数阻塞等待
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            ch <- i
            time.Sleep(time.Second)
        }
    }()
    go func() {
```

```
        for {
            fmt.Println(<-ch)
        }
    }()
    select {}
}


// 注意，channel可以和空结构体做一个很有意思的操作
ch := make(chan int,0) //定义容量为0的ch，此时只要是存数据操作一定会被阻塞
ch <- 1 // 阻塞
i <- ch // 但如果同时有读出的情况，那么就可以被解锁

// 好的，那么我不用1,专门用一个空结构体呢?
ch := make(chan struct{}, 0)
ch <- struct{}{}
<-ch
// 同样可以解锁，并且空结构体是不占用内存空间的，另外还不容易让人误解
```

# 函数

```
func myFuncAdd(a,b int) int {
    return a + b
}
// 上述可等价为
func myFuncAdd(a,b int) (c int){
    c = a + b
    return
}

// 在go的“面向对象”编程时，为一个类添加方式
type Human struct {
    Name string
}
func (h *Human) sayHello(){
    fmt.Println("hello",h.name)
}
// 后续即可使用，即Human类有了sayHello这个方法
h1 := Human{"xxx"}
h1.sayHello()
```

# defer使用

```go
// 想像每次遇到defer就将其压入栈，主程序结束后，再先进后出弹出
// 注意defer后跟的到底是func(){}无参数函数，还是func(xxx){}有参数情况

func foo() int {
    a, b := 3, 5
    c := a + b
    defer fmt.Println("111", c)
    fmt.Println(c)
    defer fmt.Println("222", c)
    defer func() {
        fmt.Println("333", c)
    }()
    c = 100
    return c
}

func main() {
    res := foo()
    fmt.Println(res)
}

// 上述函数的运行结果是
8
333 100
222 8
111 8
100
// 其中第10-12行这个匿名函数，其实是等到13行这条语句都执行完后，再清算的c的值，因此他是100
// 而7,8行相当于我已经把当时压栈的c值保存了，所以这个时候的c值是8
// 如果想10行也遵守7,8的规矩，那可以改成
defer func(c int){
    fmt.Println("333,c")
}(c)


// defer的使用场景，感觉比较多的就是在利用go创建函数时，每个协程完成后的wg.Done(); 以及一些关于连接的关闭，譬如文件、网络等，
// 为了防止前后代码过多或者忘记关闭，可以首先利用defer放在最前面
```

```
38
39
40  // defer与panic,recover等
```

# 接口

```
1   // interface定义了一套函数,
2   // 要是一个“类”实现了满足要求的所有函数，那么这个类就是这个interface
3   type Sayer interface {
4       say()
5   }
6
7   type Bird struct {
8       name string
9   }
10  func (b *Bird) say() {
11      fmt.Printf("%s会叫\n", b.name)
12  }
13
14  func main() {
15      var s Sayer
16      b := &Bird{name: "鸵鸟",}
17      s = b
18      s.say()
19  }
```

# 协程

```
1   // 简单案例 关于go协程创建，对共享区域加锁，WaitGroup，defer的使用
2   package main
3
4   import (
5       "fmt"
6       "sync"
7   )
8
9   func main() {
10      var sum int
11      var mu sync.Mutex
```

```
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            for j := 0; j < 10000; j++ {
                mu.Lock()
                sum++
                mu.Unlock()
            }
        }()
    }
    wg.Wait()
    fmt.Println(sum)
}
```