layout: post
title: 003-Go基础之一 包、变量、函数
category: golang架构师之路
tags: golang,tour,function,variables,package

keywords: package,tour,function,variables

Go官方编程指南交互工具 tour

在安装好go环境之后，我们可以运行一个神奇的命令浏览器自动打开并跳转到Go官方提供的编程指南(本地服务，可以避免一些你懂的网络故障..)

```
$ go tool tour
[master|●2+1…
23:38:05 Serving content from /usr/local/go/misc/tour
23:38:05 A browser window should open. If not, please visit http://127.0.0.1
:3999
23:38:06 accepting connection from: 127.0.0.1:58925
```

也可以后台运行

```
$ nohup go tool tour & tour.log

      [master|●2+1…
[1] 48104
appending output to nohup.out
zsh: command not found: tour.log

cat nohup.out

    [master|●2+1…
 23:49:30 Serving content from /usr/local/go/misc/tour
 23:49:30 A browser window should open. If not, please visit http://127.0.0.
1:3999
 23:49:31 accepting connection from: 127.0.0.1:59131
 23:50:14 accepting connection from: 127.0.0.1:59142
 23:50:16 accepting connection from: 127.0.0.1:59143
 23:50:19 accepting connection from: 127.0.0.1:59144
 23:52:46 Serving content from /usr/local/go/misc/tour
```

```
 23:52:46 A browser window should open. If not, please visit http://127.0.0.1:3999
 23:52:47 accepting connection from: 127.0.0.1:59173
```

基本结构

```go
package main

import "fmt"

var name = "gopher"

const PI = 3.14

var newType int

type gopher struct {}

type golang interface {}

func main() {
    fmt.Println("hello world! 你好，世界！")
}
```

公开的字段

在Go语言里**首字母大写的字段是公开的**
注意这个约定非常重要，在很多框架或库包调用中由此会有很多可见性，函数是否触发，表单是否填充值的特性，是否被序列化等等都会和该条约定或规则有关。

例如，在`math`包中的`Pi`就是公开的可以被外部包访问的。

`pizza`和`pi`首字母非大写，所有他们为私有的，外部访问是会报错的。

当导入包时你只能引用它的公开字段名称，任何一个`unexported`的名称或字段都不能被外部包所访问。 比如下面的示例，直接运行将会报错:
```go
package main

import "math"
```

func main() {
func.Println(math.pi)//应该为Pi
}
```

报错如下:

```
$ go run tour-pacakge.go
 # command-line-arguments
./tour-pacakge.go:12: cannot refer to unexported name math.pi
./tour-pacakge.go:12: undefined: math.pi
```

将`math.pi`修改为`math.Pi`即可正确运行。

```
$ go run tour-package.go
3.141592653589793
```

fmt的使用技巧

`fmt.Printf("%d %[1]d",10)`
通常Printf格式化字符串包含多个%参数时将会包含对应相同数量的额外操作数,但是%之后的`[1]`副词告诉Printf函数再次使用第一个操作数。第二,%后`#`副词告诉Printf在用%o、%x或%X输出时生成0、0x或0X前缀。
`go`
`o := 0666`
`fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"`
`x := int64(0xdeadbeef)`
`fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)`
`// Output:`
`// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF`

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d e^x = %8.3f\n", x, math.Exp(float64(x)))
}
打印e的幂,打印精度是小数点后三个小数精度和8个字符宽度
```

命名
```

go的变量命名一般推荐驼峰命名，优先采用较短命名，不过这取决于变量的生命周期，生命周期长，作用范围大的变量名称越详细越好。

而像ASCII和HTML这样的缩略词则避免使用大小写混合的写法,它们可能被称为 htmlEscape、HTMLEscape或escapeHTML,但不会是escapeHtml。

可以覆盖内置函数，但是要慎重。但这个在团队开发中，严厉禁止！

变量

变量是对应类型的**内存空间**，而**指针**则是**该变量在内存中的存储位置**(地址)。

`var a string` //默认零值 ""

声明的方式

```
var a string = "hello go"
var b = "world"
c := "world"
var x,y,z int = 0,1,2
var m,n = 10,11
e,f := 3,4
file,err := os.Open("/usr/local/a.txt") //函数返回值初始化一组变量!

fmt.Println(a,b,c,x,y,z,m,n,e,f,file,err)

//hello go world world 0 1 2 10 11 3 4 <nil> open /usr/local/a.txt: no such file or directory
```

总结:最简单的方式像排队领号码一样。 然后每人可以加上类型，再加上var装饰，不加var装饰的需要`:=`。

- `var`语句块定义一些列的变量
- 像在函数参数列表一样的方式，类型声明在最后
- `var`语句块可以为包或函数级别。

示例:

```
var c, java, python bool

func main() {
```

```
    var i int
    fmt.Println(i, c, java, python)
    //0 false false false
}
```

理解&和*操作符

```
package main

import "fmt"

func main() {

    a := 43

    fmt.Println(a)  // 43
    fmt.Println(&a) // 0x20818a220

    var b = &a
    fmt.Println(b)  // 0x20818a220
    fmt.Println(*b) // 43

    *b = 42        // b says, "The value at this address, change it to 42"
    fmt.Println(a) // 42

    // this is useful
    // we can pass a memory address instead of a bunch of values (we can pas
s a reference)
    // and then we can still change the value of whatever is stored at that
memory address
    // this makes our programs more performant
    // we don't have to pass around large amounts of data
    // we only have to pass around addresses

    // everything is PASS BY VALUE in go, btw
    // when we pass a memory address, we are passing a value
}
```

`*int` 这里的 `*` 是 `*int` 类型的一部分

```
package main

import "fmt"

func zero(z *int) {
```

```
    *z = 0
}

func main() {
    x := 5
    zero(&x)
    fmt.Println(x) // x is 0
}
```

参数拷贝的指针地址的值，自然间接引用传递改变。
```go
package main

import "fmt"

func zero(z *int) {
fmt.Println(z)
*z = 0
}

func main() {
x := 5
fmt.Println(&x)
zero(&x)
fmt.Println(x) // x is 0
}
```

变量初始化器

- 变量声明定义可以包含一个初始值，每个变量一个。
- 如果已经存在初始值，则可以省略类型，变量将采用初始值的类型。

```
var i, j int = 1, 2

func main() {
    var c, java, python = true, false, "no!" // 使用初始值的类型
    fmt.Println(i, j, c, java, python)
}
```

短声明变量初始化

- 在函数中，`:=`简洁赋值(短赋值块)可以使用于代替`var`定义。(隐藏着隐式类型的概念，也就是**通过表达式的右边值或变量的类型进行类型推断而获得类型**)
- 在函数外，每个语句都必须以关键词开始(`var,func`等等)，所以`:=`**结构是不可以在函数外使用的**

```
func main() {
    var i, j int = 1, 2
    k := 3
    c, java, python := 100, 100, 100
    fmt.Println(i, j, k, c, java, python)
}
```

- 并行赋值或(同时赋值)

  `a, b := 1, 2`

- 交换技巧

  `a, b = b, a` **利用go语言的并行赋值，可以比其他语言方便的进行变量值交换，而无需使用交换函数**。

- 用于多个返回值

  `val, err := Func1(var1)`

```
package main

import (
    "fmt"
)

func main() {
    student := []string{}
    students := [][]string{}
    fmt.Println(student)
    fmt.Println(students)
    fmt.Println(student == nil)

    var a []int
    fmt.Println(a == nil)
    var b []string
```

```
    fmt.Println(b == nil)
}
```

`:=` 声明并初始化
var 仅仅声明 并未初始化 所以是零值.

而make则赋予了零值
```go
package main

import (
"fmt"
)

func main() {
student := make([]string, 35)
students := make([][]string, 35)
fmt.Println(student)
fmt.Println(students)
fmt.Println(student == nil)
}
```

`_` 空白标志符

`_` 空白标识符被用于抛弃值，`_` 实际上是一个只写变量，你不能得到它的值。由于go要求无多余代码，也就是必须使用所有被声明的变量，但有时我们并不需要使用从一个函数中得到所有返回值。

`_, b = 5, 7` ，5被抛弃。

`import _ "github.com/golang..."`
引入该包，但**不直接使用该包的函数，而是调用该包的init函数**

变量的作用域

变量的生命周期或作用域只取决于**是否可达**。超出作用域则无效，而局部变量在函数返回时依然存在。

编译器会自动选择在栈上还是在堆上分配局部变量的存储空间,但可能令人惊讶的是,这个选择并

不是由用var还是new声明变量的方式决定的。

```
var global *int

func x() {
    var b int = 1
    global = &b
}

func y() {
    m := new(int)
    *m = 1
}
```

变量b是必须分配在堆上，在函数退出后，依然可以通过global变量找到b变量。从Go语言的术语中，**局部变量b从x函数中逃逸了**。

而变量m虽然用new创建，但是y函数结束后,***m并未从y函数中逃逸，编译器会首选在栈上分配**。

如果将指向短生命周期对象的指针保存到具有长生命周期的对象中, 特别是保存到全局变量时,会阻止对短生命周期对象的垃圾回收(从而可能影响程序的性能)。例如x函数中的b变量保存到了全局global变量中。

注意:**不要将作用域和生命周期混为一谈。**

- 声明语句的作用域对应的是一个源代码的文本区域;它是一个编译时的属性。
- 一个变量的生命周期是指程序运行时变量存在的有效时间段,在此时间区域内它可以被程序的其他部分引用;是一个运行时的概念。

包与导入路径

- 每个Go程序都是由包组织成
- 程序入口在`main`包下
- 该示例程序`导入`了`fmt`和`math/rand`两个`路径`。
- 根据Go语言的惯例，`包名和导入路径的最后一个元素相同`。例如，"math/rand"包含在一个以`package rand`开头的文件。(**注意:导入路径为"math/rand"，但包名为"rand"**)
- 别名`fm "fmt"`
- 包名和包所在的文件夹名可以是不同的

```
package main
```

```
import(
    "fmt"
    "math/rand"
)

func main(){
    fmt.Println("My favorite number is",rand.Intn(10))
}
```

导入(import statement)

- 导入语句中使用了`()`打包多个导入路径，可以认为是"打包"导入方式，其实也可以分别导入(不推荐)，例如:

```
package main

// import "fmt"
// import "math"

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Now you have %g problems.", math.Sqrt(7))
}
```

导入别名

格式 `import alias_name "package_name"`

例如: `import fm "fmt"`

```
package main

import (
    my_pkg "pkg01/sub_pkg"
    "fmt"
)

func main() {
    reverse_str := my_pkg.Reverse("!olleh")
```

```
    fmt.Printf("%v", reverse_str)
}
```

```
☁  pkg01  tree
.
├── sub_pkg
│   └── mystring.go
└── test.go

1 directory, 2 files
```

本地导入(类似Java的静态导入)

```
package main

import (
    . "strings"
    "fmt"
)

func main() {
    fmt.Println(HasPrefix("kobe","k"))
    fmt.Println(HasPrefix("kobe","b"))
}
```

导入初始化

导入初始化一般是导入父类或容器的配置，预加载，初始化等重操作或者框架的默认导入操作。使用场景比如`导入数据库驱动`等等。由于golang对于未使用的导入不会被编译通过，所以需要使用`_`操作符通知编译器。

```
package main

import (
    // "strings"  编译错误
    _ "strings"
)

func main() {
}
```

代码包初始化

```
package main

import "fmt"

func init() {
    fmt.Println("init")
}
func main() {
    fmt.Println("main")
}
```

注意：**init函数的执行时机尽量不要假设，除非知道依赖顺序**，所以最好是有个统一的地方包裹初始化操作。

另外导入该包虽然代码中未使用到，但是也在执行前执行了导入包的**init函数**。常用在一些初始化动作，比如加载数据库驱动等操作。

go doc

使用方式:

- `go doc package`
- `go doc package/subpackage`
- `go doc package function`
- `godoc -http=:6060` 打开本地go官方的文档(注意不要搞反了 不是`:=Port` 而是 `=:Port`)

`go doc`命令行
`godoc`则是server服务，不仅仅官方文档的说明，只要是gopath中的库包都会有。

```
☁  ~  go doc pkg01/sub_pkg Reverse
func Reverse(s string) string

☁  ~  go doc pkg01/sub_pkg
package sub_pkg // import "pkg01/sub_pkg"

Reverse string

func Reverse(s string) string
```

随机函数的案例

我们也可以看到注释生成文档的方式，在函数上方，以函数名开头`Method do what`的方式。

```
☁  aws  go doc math/rand New
func New(src Source) *Rand
    New returns a new Rand that uses random values from src to generate othe
r
    random values.
```

```
func main() {
    //randN := rand.New(rand.NewSource(100)) 这种方式是没有效果的，需要有个变化的
种子
    randN := rand.New(rand.NewSource(time.Now().UnixNano()))//给一个时间戳值
    for i := 0; i < 1000; i++ {
        if a := randN.Intn(1000); a < 10 {
            fmt.Println(a)
        }
    }
}
```

注意：**这个程序的运行环境是固定的，因此 rand.Intn 总是会返回相同的数字,甚至在一组连续的输出连顺序都是一样的。（为了得到不同的数字，需要生成不同的种子数，参阅 rand.Seed。）**

```
$: go doc math/rand
package rand // import "math/rand"

Package rand implements pseudo-random number generators.

Random numbers are generated by a Source. Top-level functions, such as
Float64 and Int, use a default shared Source that produces a deterministic
sequence of values each time a program is run. Use the Seed function to
initialize the default Source if different behavior is required for each
run. The default Source is safe for concurrent use by multiple goroutines.

For random numbers suitable for security-sensitive work, see the crypto/rand
package.


func Float64() float64
func Int() int
func Int31() int32
func Int31n(n int32) int32
func Int63() int64
```

```
func Read(p []byte) (n int, err error)
func Seed(seed int64)
func Uint32() uint32
func Uint64() uint64
type Rand struct{ ... }
    func New(src Source) *Rand
type Source interface{ ... }
    func NewSource(seed int64) Source
type Source64 interface{ ... }
type Zipf struct{ ... }
    func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf
```

```
// rand.Seed(19990)固定种子，每次的结果都是一样
r := rand.New(rand.NewSource(time.Now().UnixNano())) // 时间序列种子
fmt.Println(r.Intn(1000)) // 0~1000中取随机数
```

我们可以观察到New函数和NewSource函数

```
// A Rand is a source of random numbers.
type Rand struct {
    src Source
    s64 Source64 // non-nil if src is source64

    // readVal contains remainder of 63-bit integer used for bytes
    // generation during most recent Read call.
    // It is saved so next Read call can start where the previous
    // one finished.
    readVal int64
    // readPos indicates the number of low-order bytes of readVal
    // that are still valid.
    readPos int8
}

// New returns a new Rand that uses random values from src
// to generate other random values.
func New(src Source) *Rand {
    s64, _ := src.(Source64)
    return &Rand{src: src, s64: s64}
}
```

类型

go语言数据类型

Go语言将数据类型分为四类: `基础类型、复合类型、引用类型和接口类型` 。本节介绍

- 基础类型
  - 数字
  - 字符串
  - 布尔
  - error
- 复合类型(通过组合简单的基础类型组成复杂数据结构)
  - 数组
  - 结构体
- 引用类型(也属于复合类型，但是变量或状态的间接引用)
  - pointer
  - slice
  - map
  - function
  - channel 引用类型包括指针(§2.3.2)、切片 (§4.2))字典(§4.3)、函数(§5)、通道(§8),虽然数据种类很多,但**它们都是对程序中一个变量或状态的间接引用。这意味着对任一引用类型数据的修改都会影响所有该引用的拷贝。**
- 接口类型

Go的基本类型如下:

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64

byte // alias for uint8

rune // alias for int32
     // represents a Unicode code point(Unicode码)

float32 float64

complex64 complex128
```

注意byte（uint8）和rune(int32)都是别名类型。

示例展示了几个类型变量，变量声明也可以"打包"到`var ()`中类似`import()`导入块

```
func main() {
        var (
            ToBe   bool       = false
            MaxInt uint64     = 1<<64 - 1
            z      complex128 = cmplx.Sqrt(-5 + 12i)
        )
    fmt.Printf("Type: %T, Value: %v\n", ToBe, ToBe)
    fmt.Printf("Type: %T, Value: %v\n", MaxInt, MaxInt)
    fmt.Printf("Type: %T, Value: %v\n", z, z)
}
//Type: bool, Value: false
//Type: uint64, Value: 18446744073709551615
//Type: complex128, Value: (2+3i)
```

`int`,`uint`,`uintptr`类型通常在32位系统是32位，64位系统是64位。当你需要使用整型值的时候应该首先使用`int`类型，除非你有特殊的原因使用一个有大小的或无符号的整形。

使用打印格式化符可以很方便的做进制转换

`go`

`fmt.Printf("%d - %b \n", 42, 42)`

`// 42 - 101010`

```
for i := 60; i < 122; i++ {
        fmt.Printf("%d \t %b \t %x \t %q \n", i, i, i, i)
    }
    //%d数字 %b二进制 %x16进制 %q utf8字符

    //65     1000001     41      'A'
    //97     1100001     61      'a'
```

类型默认值

变量声明时没有给予特定的初始值时，Go会给予默认值(Zero value)。有人也翻译称作为"零值"。

默认值有以下几种:

- 数值型`0`
- 布尔型`false`

- 字符型 `""` (空字符串)

```
func main() {
   var i int
    var f string
    var b bool
    var s float64
    //fmt.Printf(i, f, b, s)//can not use i(int) as string
    fmt.Printf("%v,%q,%v,%v\n", i, f, b, s)
    fmt.Printf("%v,%v,%v,%v\n", i, f, b, s)
        //0,"",false,0 注意这里%q对于字符型值的作用
        //0,,false,0
}
```

格式化符 `%v` 对于字符串表示为空（什么都没有的空），而 `%q` 则可以表示的更加接近字符串空为 `""`

命名的类型(类型声明或类型的别名)

一个类型声明语句创建了一个新的类型名称,**和现有类型具有相同的底层结构**。新命名的类型提供了一个方法,用来分隔不同概念的类型,这样**即使它们底层类型相同也是不兼容的**。

只有当两个类型的底层基础类型相同时,才允许这种转型操作,或者是两者都是指向相同底层结构的指针类型,这些转换只改变类型而不会影响值本身。Celsius(T),Fahrenheit(T) 。
但注意数值类型的类型转换可能会改变值(一般表现为精度的缺失)

```
type typeName underlyingType

type Celsius float64 // 摄氏温度

type Fahrenheit float64 // 华氏温度

var c Celsius
var f Fahrenheit
```

为类型别名添加新的行为

命名类型除了提供简单的类型名称，还会为该类型的值定义新的行为，即为该类型提供方法。将类型参数放到方法名称的前面，表示为该类型的xx方法。

```
func (c Celsius) String() string {
```

```
    return fmt.Sprintf("%g°C",c)
}
```

类型转换

表达式 `T(v)` 将值 `v` 转换为 `T` 类型。

一些数值型转换:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

或者简写:

```
i := 42
f := float64(i)
u := uint(f)
```

和C不一样，**Go的分配赋值在不同类型之间需要显示转换，在变量赋值的时候，如果是直接赋值，那么会进行类型推断，不要混为一谈**（也就是说**没有隐式转换，拆箱或装箱**）。试试移除 `float64` 或 `uint` 在例子中的转换(注意是非简写的例子尝试)，看会发生什么。

```
var o int = 56
var p float64 = o //Cannot use o(int) as float64 in assignment
var q uint = uint(p)

o := 55 // no new variables on left side of :=
//p := o
//q := uint(p)
fmt.Printf("%v,%v,%v", o, p, q)
```

再例如:

```
func main() {
    var x, y int = 3, 4
    var f float64 = math.Sqrt((x*x + y*y))//Cannot use (x*x+y*y)(type int) a
s float64 in assignment
    var z uint = uint(f)

    fmt.Println(x, y, z)
```

```
}
```

类型推断

当声明变量但没有明确指定其类型时(不论是使用 `:=` 还是 `var =` 表达式语法)，变量类型将从表达式的右边的值进行类型推断。
当表达式右边是已经声明过变量类型，新的变量则会和右边类型相同。例如:

```
var i string
j := i // j的类型为string
fmt.Printf("%q,%q\n", i,j)
```

但是当表达式右边为没有指明类型的数字常量时，新的变量则可能
为 `int` 、 `float` 、 `complex128` 进行推断，这就取决于常量的精度。

```
i := 42 // int
f := 3.14 // float64 float默认为float64类型
g := 0.876 + 0.5i // complex128
```

```
func main() {
    v := 4322
    fmt.Printf("v is type of %T\n", v)
    x := float64(v)
    fmt.Printf("x is type of %T", x)
}

// v is type of int
// x is type of float64
```

相同类型才能匹配比较

```
package main

func main() {
    a := int32(5)
    b := int64(5)
    if a==b { // 编译错误，需类型相同。如果是类型是接口，必须实现相同的接口
        print(true)
    }
```

```
}
```

NaN

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) // "0 -0 +Inf -Inf NaN"
nan := math.NaN()
fmt.Printf("%f\n",nan)
fmt.Println(nan == nan, nan < nan, nan > nan)// false false false 注意:NaN和任何数都不相等


var b int = 0
var z float64 = float64(b)
```

error类型

error类型是go的内置类型，go中还有专门处理error类型的包errors

```
err := errors.New("oops error")
if err != nil {
    fmt.Printf("%v\n", err)
}
```

中文版gotour

```
go get github.com/Go-zh/tour/gotour
cd $GOPATH/bin
./gotour
```

可以使本地英文版和中文版一起打开，只需修改下其中的一个的端口，然后需要构建下运行。
注意:使用go get下载时可能会出现网络问题你懂得，不过多试几次应该可以成功。
如果你定义好了$GOBIN,并加入到了path中，go get完成后，可以在任意位置使用`gotour`命令，但是你可能想要让自带的tour也能运行(其命令为go tool tour，端口为3999)，此时就需要修改中文本的端口，找到

```
httpListen = flag.String("http","3999","host: port to listen on")
```

将3999修改为其他的可用端口。