链滴

Go 程序结构 (一)

作者：ReyRen

原文链接：https://ld246.com/article/1572524757976

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Names

Go的方法, 变量, 常量, 数据类型, 声明标签和包的命名都遵循一个简单的规则: 命名开始字符是字母或是下划线加上任意数量的字母, 数字和下划线. 并且大小写是有区分的, 比如说"HeapSort"和"heapSort就是两种不同的名字.

Go有25个关键字, 这些关键字是在特定语法结构中使用的, 不能用于普通自定义的命名, 这里就不去列了.

除此之外, Go还有三十个预处理命名比如"int"主要用于内置的常量, 类型和方法:

Constants: true false iota nil

Types: int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64 uintptr float32 float64 comple 128 complex64 bool byte rune string error

Functions:make len cap new append copy close delete complex real imag panic recover

这些命名都不是保留的, 也就是说我们可以重新定义它, 但是一定要注意避免混淆.

如果名字的启示字母是大写的, 那么就意味着它是exported的. 也就是说对于它自己的包意外都是可的并且饮用到了程序的其他部分去. 比如说在"fmt"包中的Printf. 但是报名通常来说倒是小写的.

虽然名字的长度在Go中是没有限制的, 但是通俗的约定尽量的简介. 名字的长度会根据你本身作用域大小来觉定. 作用域越大, 命名越详细越好.

风格上来说, Go是建议"驼峰式"命名的. 比如"QuoteRuneToASCII"和"parseRequestLine"而不推荐下划线的命名规范.

Declarations(在Golang中并不区分声明和定义)

定义就是命名了一个程序的实体, 并且指定了一些或者所有的属性. 有四种定义方式: var, const, type, f

nc.

一个Go程序存储在一个或者多个以".go"结尾的文件中, 每一个Go文件都是以package定义开始, 表示个文件是哪个包的一部分. 紧随其后的是import定义, 紧随其后的就是const, fun等. 比如以下这个程就是定义了常量, 方法和变量:

```
// Boiling prints the boiling point of water.
package main

import "fmt"

const boilingF = 212.0

func main() {
        var f = boilingF
        var c = (f - 32) * 5 / 9
        fmt.Printf("boiling point = %g°F or %g°C\n", f, c)
        // Output:
        // boiling point = 212°F or 100°C
}
```

常量"boilingF"是package-level的定义和"main"一样, 但是像"f"就是local-level的定义了. package-lvel的定义不仅对本源文件可见, 对整个包的所有文件都可见. 相反, local-level的定义只在本方法甚至小的范围内可见.

Go的方法定义格式是方法名, 形参, 返回值(如果有的话) 最后是方法体.

Variables

变量的定义使用var, 基本上是这个模式:

```
var name type = expression
```

type和= expression可以省略其中一种, 但是不能全部省略. 如果type省略, 那么type决定于expressio的类型. 如果是后者省略, 那么针对于不同的type都是零值, 也就是说, 数字是0, boolean是false, ""给strng类型, nil给interface和引用类型(slice, pointer, map, channel, function). 对于集合类型(数组, 结体)表示其中的所有元素和域都是零值.

在Go中是没有无值的情况的, 给予零值是确保变量类型总能拿到一个正确的默认值. 比如说:

```
var s string
fmt.Println(s) // ""
```

像这种指向一个空的的字符串类型并不会报错等.

当然也是可以进行多个变量的同时定义的:

```
var i, j, k int                 // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

对于package-level的变量的初始化要在main函数之前, local-level的就是在使用之前初始化就行.

一些变量的初始化也可以是通过一些函数的返回值来进行的:

```
var f, err = os.Open(name) // os.Open returns a file and an error
```

Short Variable Declarations

在一个函数内, 一个可替代的方案是"short variable declaration", 用来定义和初始化一些local-level变量:

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0
t := 0.0
```

因为"short variable declaration"的简短性和灵活性,所以这种方式几乎承担了绝大多数的local-level量的定义和初始化, 通过var这种方式定义的似乎只有在定义的类型和表达式不同或者想要在之后才进赋有意义的值的时候才能派上用场.

```
i := 100                  // an int
var boiling float64 = 100 // a float64

var names []string
var err error
var p Point
```

同样的, 也是可以给多个变量使用"short variable declaration":

```
i, j := 0, 1
```

这里一定要注意区分:=和=, 前者是定义变量, 后者是赋值, 在之后的文章中也会提到元祖赋值. 在赋值作中, 最右边的变量赋值给最左边的变量:

```
i, j = j, i // swap values of i and j
```

和普通的变量声明定义一样, "short variable declaration"也可以接受来自函数返回值的形式进行初化:

```
f, err := os.Open(name) //返回两个值
if err != nil {
    return err
}
 // ...use f...
     f.Close()
```

一个微妙但是很重要的点是: "short variable declaration"并不是必须定义左边的所有变量. 如果某些量已经在lexical block范围内定义过了, 那么这个变量就成为了赋值了:

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

第一条语句定义了in和err, 第二条语句只是定义了out, 但是赋值了err.

但是"short variable declaration"得保证至少有一个变量进行定义, 所以说下面的这种语句就是错的:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // compile error: no new variables
```

那么正确的做饭就是使用常规定义操作进行重定义.

Pointers

这个指针就是和c的指针相同的概念.

```
x := 1
p := &x         // p, of type *int, points to x
fmt.Println(*p) // "1"
*p = 2          // equivalent to x = 2
fmt.Println(x)  // "2"
```

对于一个函数返回一个变量的地址是绝对安全的. 比如, 在下面的代码中, (这个是有别与C的), 也就是函数将本地变量将以地址返回, 即使是出了作用域, p一直指向它并且值是保留了的:

```
var p = f()
func f() *int {
    v := 1
    return &v
}
```

当然每次调用f()都会返回一个不同的地址;

```
fmt.Println(f() == f()) // "false"
```

接下来就是和C操作变量指针一样的思路了:

```
func incr(p *int) int {
        *p++ // increments what p points to; does not change p
        return *p
}

v := 1
incr(&v)              // side effect: v is now 2
fmt.Println(incr(&v)) // "3" (and v is 3)
```

每次我们获取一个变量的地址或者拷贝一个指针的时候, 都是使用的别名来进行识别. 比如, *p就是v的名. 这样很方便的一点是可以不直接操作原名的方式进行变量的操作. 但凡事都有利弊, 不方便的一点是我们得知道所有的原变量的别名.不仅仅是进行这种指针操作的时候会有别名, 在其他的引用类型中(slice, maps, channels 甚至structs, arrays, interface)也会包含这些别名的概念.

指针也是"flag"这个包的关键, "flag"包的作用是可以对命令参数进行解析. 想必这个是项目必备功能. 一步解释, 我们举个简单的例子, 在这个例子中会有两个参数标志(./a.out -l xx, 其中-l就是参数标志), -n和s sep. 前者是用来实现忽略换行的功能, 后者是用指定的sep进行字符串的切割:

```
//prints its command-line arguments.
package main

import (
   "flag"
   "fmt"
   "strings"
)

var n = flag.Bool("n", false, "omit trailing newline")
```

```
var sep = flag.String("s", " ", "separator")//第三个参数的意思就是如果输入非法标志, 或者-h, -hel
等就会打印这样的信息.

func main() {
   flag.Parse()
   fmt.Print(strings.Join(flag.Args(), *sep))
   if !*n {
      fmt.Println()
   }
}
```

注意⚠️warning:

在所有的flag定义完成之后, 可以通过调用 flag.Parse() 进行解析. 目的是更新标志变量.

非flag的参数可以使用flags.Args()进行切片获取. 如果flag.Parse()解析出错, 就会打印上面指定的mes
age, 并且调用os.Exit(2)结束程序.

命令行flag的语法有如下三种形式:

```
-flag // 只支持bool类型
-flag=x
-flag x // 只支持非bool类型
```

以上语法对于一个或两个 - 号, 效果是一样的, 但是要注意对于第三种情况, 只能用于**非 bool**类型的flag
原因是: 如果支持, 那么对于这样的命令 ./a.out -flag *, 如果有一个文件名字是: 0或false等, 则命令
原意会改变(bool 类型可以和其他类型一样处理, 其次 bool 类型支持 -flag 这种形式, 因为Parse()中,
bool 类型进行了特殊处理). 默认的, 提供了 -flag, 则对应的值为 true, 否则为 flag.Bool/BoolVar 中
定的默认值; 如果希望显示设置为 false 则使用 -flag=false.

int类型可以是十进制、十六进制、八进制甚至是负数; bool 类型可以是1, 0, t, f, true, false, TRUE, FA
SE, True, False.

如果bool类型的参数在命令行中用了-flag false这种形式时, 其后的参数都会被当做非flag (non-flag)
数, non-flag参数以后做解释.

再一点就是flag返回的都是一个指针.

所以我们现在看看输出结果:

```
$ go build gopl.io/ch2/echo4
    $ ./echo4 a bc def
    a bc def
    $ ./echo4 -s / a bc def
    a/bc/def

 $ ./echo4 -n a bc def
    a bc def$
    $ ./echo4 -help
    Usage of ./echo4:
          -n    omit trailing newline
          -s  string
                separator (default " ")
```

Pointers

另外一种生成一个变量的方式是使用内置关键字new. 表达式new(T)会生成一个为命名的T类型的变量, 初始化为T类型的零值, 并且返回它的地址:

```
p := new(int)   // p, of type *int, points to an unnamed int variable
fmt.Println(*p) // "0"
*p = 2          // sets the unnamed int to 2
fmt.Println(*p) // "2"
```

用new(T)生成变量和我们普通的定义变量的方式不同点仅仅是new不需要想那些恶心的名字去当变量:

```
func newInt() *int {
    var dummy int
    return &dummy
}
```

上面是我们常规写法, 接下来用new(T)的方式:

```
func newInt() *int {
    return new(int)
}
```

每次调用new都会返回一个不通的地址:

```
p := new(int)
q := new(int)
fmt.Println(p == q) // "false"
```

其实new是很少用的, 因为大部分未命名的变量都是struct类型.

因为new是预定义函数, 不是关键字, 所以在函数的其他地方是有可能重新定义的:

```
func delta(old, new int) int { return new - old }
```

当然在delta函数中是不能使用内置的new函数的.

Lifetime of Variables