**Go学习笔记**

**V1.0.0**

**Author: clearuo**

Date: 2017-03-05

# 基础语法

## 基础类型和运算符

1. s:=123这种定义是int32还是int64还是其他类型？试着赋值给其给uint32或int64看是否需要强制转换

s:=123这种定义是int类型,需要强制转换才能赋值给uint32或uint64

2. 在 fmt.Printf 中使用下面的说明符来打印有关变量的相关信息：

* %+v 打印包括字段在内的实例的完整信息
* %#v 打印包括字段和限定类型名称在内的实例的完整信息
* %T 打印某个类型的完整说明

3.在 Go 语言中，&& 和 || 是具有快捷性质的运算符，当运算符左边表达式的值已经能够决定整个表达式的值的时候（&& 左边的值为 false，|| 左边的值为 true），运算符右边的表达式将不会被执行。利用这个性质，如果你有多个条件判断，应当将计算过程较为复杂的表达式放在运算符的右侧以减少不必要的运算。

4.格式化说明

在格式化字符串里，%d 用于格式化整数（%x 和 %X 用于格式化 16 进制表示的数字），%g 用于格式化浮点型（%f 输出浮点数，%e 输出科学计数表示法），%0d 用于规定输出定长的整数，其中开头的数字 0 是必须的。

%n.mg 用于表示数字 n 并精确到小数点后 m 位，除了使用 g 之外，还可以使用 e 或者 f，例如：使用格式化字符串 %5.2e 来输出 3.4 的结果为 3.40e+00。

5.运算符与优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| **优先级** | **运算符** |
| 7 | **^ !** |
| 6 | **\* / % << >> & &^** |
| 5 | **+ - | ^** |
| 4 | **== != < <= >= >** |
| 3 | **<-** |
| 2 | **&&** |
| 1 | **||** |

6.获取字符串中某个字节的地址的行为是非法的,例如:&str[i]

## 函数

### 基础概念

### 闭包

* 匿名函数同样被称之为闭包(函数时语言的术语):它们被允许调用定义在其它环境下的变量.闭包可使得某个函数扑捉到一些外部状态,例如:函数被创建时的状态.另一种表示方式为:一个闭包继承了函数所声明时的作用域.这种状态(作用域内的变量)都被共享到闭包的环境中,因此这些变量可以在闭包中被操作,直到被销毁
* 示例1

|  |
| --- |
| func main() {     f := colosure(10)     fmt.Println(f(1))     fmt.Println(f(2))     fmt.Println(f(3)) } func colosure(x int) func(int) int {     fmt.Printf("%p\n", &x)     return func(y int) int {         fmt.Printf("%p\n", &x)         return x + y     } } //运行结果 0xc0420381d0 0xc0420381d0 11 0xc0420381d0 12 0xc0420381d0 13 |

### Defer

* 使用defer实现代码追踪

|  |
| --- |
| func main() {     b() } func trace(s string) string {     fmt.Println("Entering:", s)     return s } func un(s string) {     fmt.Println("Leaving:", s) } func a() {     defer un(trace("a"))     fmt.Println("in a") } func b() {     defer un(trace("b"))     fmt.Println("in b")     a() } //运行结果 Entering: b in b Entering: a in a Leaving: a Leaving: b |

* 使用defer语句来记录函数的参数与返回值

|  |
| --- |
| func main() {     func1("Go") } func func1(s string) (n int, err error) {     defer func() {         log.Printf("func1(%q) = %d, %v\n", s, n, err)     }()     return 7, io.EOF } //运行结果 2017/05/05 17:01:54 func1("Go") = 7, EOF |

* 示例1

|  |
| --- |
| func main() {     for i := 0; i < 3; i++ {         defer fmt.Println(i)     }     for i := 0; i < 3; i++ {         defer func() {             fmt.Println(i)         }()     } } //运行结果 3 3 3 2 1 0 |

### Panic,Recover

* 示例1

|  |
| --- |
| func main() {     A()     B()     C() } func A() {     fmt.Println("Func A") } func B() {     defer func() {         if err := recover(); err != nil {             fmt.Println("Recover in B")         }     }()     panic("Panic in B") } func C() {     fmt.Println("Func C") } //运行结果 Func A Recover in B Func C |

* 示例2

|  |
| --- |
| func main() {     var fs = [4]func(){}     for i := 0; i < 4; i++ {         defer fmt.Println("defer i = ", i)         defer func() { fmt.Println("defer\_colosure i = ", i) }()         fs[i] = func() { fmt.Println("colosure i = ", i) }     }     for \_, f := range fs {         f()     } } //运行结果 colosure i =  4 colosure i =  4 colosure i =  4 colosure i =  4 defer\_colosure i =  4 defer i =  3 defer\_colosure i =  4 defer i =  2 defer\_colosure i =  4 defer i =  1 defer\_colosure i =  4 defer i =  0 |

### 变长参数

* 示例

|  |
| --- |
| func main() {     a := []int{1, 2, 3}     fmt.Println(a)     mytest(a...) // 此时为引用传递     fmt.Println(a)     x, y, z := 11, 22, 33     mytest(x, y, z) // 此时为值传递     fmt.Println(x, y, z) } func mytest(a ...int) {     for i := range a {         a[i] = 33     } } //运行结果 [1 2 3] [33 33 33] 11 22 33 |

### 提问要点

* func test(a…interface{})传递验证

|  |
| --- |
| func main() {     mytest(1, "aa", 88.6) } func mytest(a ...interface{}) {     fmt.Println(a) } //运行结果 [1 aa 88.6] |

* 请学习以下示例并思考:函数f返回时,变量ret的值是什么?

|  |
| --- |
| func main() {     fmt.Println(f()) } func f() (ret int) {     defer func() {         ret++     }()     return 1 } |

* 变量ret的值为2,因为ret++是在执行return 1语句以后执行的,也就是说defer是在return以后才执行,这可用于在返回语句之后修改返回的error时使用
* 不使用递归但使用闭包实现斐波那契数列程序

|  |
| --- |
| func main() {     fmt.Println(mytest(9)) } func mytest(num int) int {     f := fibonacci()     for i := 0; i < num; i++ {         f()     }     return f() } func fibonacci() func() int {     back1, back2 := 1, 1     return func() int {         temp := back1         back1, back2 = back2, (back1 + back2)         return temp     } } // 运行结果 // 55 // 可使用数组存储每次计算的fibonacci的值,不必每次计算 |

* 工厂函数
* 一个返回值为另一个函数的函数可以被称之为工厂函数，这在您需要创建一系列相似的函数的时候非常有用：书写一个工厂函数而不是针对每种情况都书写一个函数。下面的函数演示了如何动态返回追加后缀的函数：

|  |
| --- |
| func MakeAddSuffix(suffix string) func(string) string {     return func(name string) string {         if !strings.HasSuffix(name, suffix) {             return name + suffix         }         return name     } } |

* 现在，我们可以生成如下函数：

|  |
| --- |
| addBmp := MakeAddSuffix(".bmp") addJpeg := MakeAddSuffix(".jpeg") 然后调用它们： addBmp("file") // returns: file.bmp addJpeg("file") // returns: file.jpeg |

* 可以返回其它函数的函数和接受其它函数作为参数的函数均被称之为高阶函数，是函数式语言的特点。我们已经在第 6.7 中得知函数也是一种值，因此很显然 Go 语言具有一些函数式语言的特性。闭包在 Go 语言中非常常见，常用于 goroutine 和管道操作

## 数组与切片

### 基础概念

* 数组长度也是数组类型的一部分，所以[5]int和[10]int是属于不同类型的。数组的编译时值初始化是按照数组顺序完成的
* 元素的数目，也称为长度或者数组大小必须是固定的并且在声明该数组时就给出（编译时需要知道数组长度以便分配内存）；数组长度最大为 2Gb
* 如果我们想让数组元素类型为任意类型的话可以使用空接口作为类型当使用值时我们必须先做一个类型判断
* 由于长度也是数组类型的一部分，因此[3]int与[4]int是不同的类型，数组也就不能改变长度。数组之间的赋值是值的赋值，即当把一个数组作为参数传入函数的时候，传入的其实是该数组的副本，而不是它的指针。如果要使用指针，那么就需要用到后面介绍的slice类型了。验证函数传递

|  |
| --- |
| func main() {     arr := [...]int{1, 2, 3}     fmt.Println("all before:", arr)     test\_value(arr)     fmt.Println("value after:", arr)     test\_point(&arr)     fmt.Println("point after:", arr)     test\_slice(arr[:])     fmt.Println("slice after:", arr) } func test\_value(arr [3]int) {     arr[1] = 111     fmt.Println("test\_value:", arr) } func test\_point(arr \*[3]int) {     arr[1] = 222     fmt.Println("test\_point:", arr) } func test\_slice(arr []int) {     arr[1] = 333     fmt.Println("test\_slice:", arr) } |

* 注意 绝对不要用指针指向 slice。切片本身已经是一个引用类型，所以它本身就是一个指针!!
* []interface{}这种切片是否可以存储不同数据类型
* 可以,因为该切片存储的类型是interface{}
* 通过切片引用传递给函数,在函数中修改此切片后,调用者得到的切片一定是函数中修改后的内容吗?
* 未必,通过以此达到返回数据给调用者是不安全的,因为如果在此函数中操作切片内容大于原有切片的空间的话,将从新分配空间,此时调用者使用的还是原来数组控件,如下例子:

|  |
| --- |
| func main() {    a := []byte{0, 1, 2}    fmt.Println(a)    mytest(a)    fmt.Println(a) } func mytest(a []byte){    a[0]=00    a[1]=11    a[2]=22    a = append(a,55)    a[0]=77    fmt.Println("cap(a) = ",cap(a)) } // 运行结果 // [0 1 2]  // cap(a) =  8 // [0 11 22] |

### new和make的区别

* 看起来二者没什么区别,都在堆上分配内存,但是它们的行为不同个,适用于不同的类型;
* new(T)为每个新的类型T分配一片内存,初始化为0并且返回类型为\*T的内存地址:这种方法返回一个指向类型为T,值为0的地址的指针,它适用于值类型,如数组和结构体;它相当于&T{};
* make(T)返回一个类型为T的初始值,它只适用于3种内建的引用类型:切片,map和channel;
* 换言之,new函数分配内存,make函数初始化;
* 下面的例子说明了在映射上使用 new 和 make 的区别以及可能发生的错误：

|  |
| --- |
| package main type Foo map[string]string type Bar struct {     thingOne string     thingTwo int } func main() {     // OK     y := new(Bar)     (\*y).thingOne = "hello"     (\*y).thingTwo = 1     // NOT OK     z := make(Bar) // 编译错误：cannot make type Bar     (\*z).thingOne = "hello"     (\*z).thingTwo = 1     // OK     x := make(Foo)     x["x"] = "goodbye"     x["y"] = "world"     // NOT OK     u := new(Foo)     (\*u)["x"] = "goodbye" // 运行时错误!! panic: assignment to entry in nil map     (\*u)["y"] = "world" } |

* 试图 make() 一个结构体变量，会引发一个编译错误，这还不是太糟糕，但是 new() 一个映射并试图使用数据填充它，将会引发运行时错误！ 因为 new(Foo) 返回的是一个指向 nil 的指针，它尚未被分配内存。所以在使用 map 时要特别谨慎。

### append

* 如果你想将切片 y 追加到切片 x 后面，只要将第二个参数扩展成一个列表即可：x = append(x, y...)
* 将一个字符串追加到某一个字符数组的尾部

|  |
| --- |
| func main() {     str:="abc中国sd"     var b[]byte     b = append(b,str...)     fmt.Println(string(b)) } |

* append函数常见操作
* 将切片 b 的元素追加到切片 a 之后：a = append(a, b...)
* 复制切片 a 的元素到新的切片 b 上：b = make([]T, len(a)); copy(b, a)
* 删除位于索引 i 的元素：a = append(a[:i], a[i+1:]...)
* 切除切片 a 中从索引 i 至 j 位置的元素：a = append(a[:i], a[j:]...)
* 为切片 a 扩展 j 个元素长度：a = append(a, make([]T, j)...)
* 在索引 i 的位置插入元素 x：a = append(a[:i], append([]T{x}, a[i:]...)...)
* 在索引 i 的位置插入长度为 j 的新切片：a = append(a[:i], append(make([]T, j), a[i:]...)...)
* 在索引 i 的位置插入切片 b 的所有元素：a = append(a[:i], append(b, a[i:]...)...)
* 取出位于切片 a 最末尾的元素 x：x, a = a[len(a)-1], a[:len(a)-1]
* 将元素 x 追加到切片 a：a = append(a, x)
* 因此，您可以使用切片和 append 操作来表示任意可变长度的序列。
* 从数学的角度来看，切片相当于向量，如果需要的话可以定义一个向量作为切片的别名来进行操作。

### 切片的引用传递和append要点

* 引用传递,实际上是引用地址的值传递;append之所以需要接受返回值是也能为每次append后需要返回切片底层对应数组的开始地址和切片长度;不说了,看例子

|  |
| --- |
| func main() {     sourData := [5]int{0, 1, 2, 3, 4}     data := sourData[0:4]     fmt.Println(data)     //mytest1(data)     //mytest2(&data)     mytest3(data)     fmt.Println(data)     fmt.Println(sourData) } func mytest1(data []int) {     data = append(data, 33) } func mytest2(data \*[]int){     \*data = append(\*data,33) } func mytest3(data []int){     data[2]=22 } //mytest1运行结果 [0 1 2 3] [0 1 2 3] [0 1 2 3 33] //mytest2运行结果 [0 1 2 3] [0 1 2 3 33] [0 1 2 3 33] //mytest3运行结果 [0 1 2 3] [0 1 22 3] [0 1 22 3 4] |

* 注意： append 在大多数情况下很好用，但是如果你想完全掌控整个追加过程，你可以实现一个这样的 AppendByte 方法：

|  |
| --- |
| func AppendByte(slice []byte, data ...byte) []byte {     m := len(slice)     n := m + len(data)     if n > cap(slice) { // if necessary, reallocate         // allocate double what's needed, for future growth.         newSlice := make([]byte, (n+1)\*2)         copy(newSlice, slice)         slice = newSlice     }     slice = slice[0:n]     copy(slice[m:n], data)     return slice } |

* func copy(dst, src []T) int copy 方法将类型为 T 的切片从源地址 src 拷贝到目标地址 dst，覆盖 dst 的相关元素，并且返回拷贝的元素个数。源地址和目标地址可能会有重叠。拷贝个数是 src 和 dst 的长度最小值。如果 src 是字符串那么元素类型就是 byte。如果你还想继续使用 src，在拷贝结束后执行 src = dst。

### copy

* 假设s是一个字符串(本质上是一个字节数组),那么就可以通过c:=[]byte(s)来获取一个字节的切片c,另外,您还可以通过copy函数来达到相同的目的:copy(dst []byte, src string)

### 切片和垃圾回收

* 切片的底层指向一个数组,该数组的实际容量可能要大于切片所定义的容量.只有在没有任何切片指向的时候,底层的数组内层才会被释放,这种特性有时会导致程序占用多余的内存.
* 示例函数FindDigits将一个文件加载到内存,然后搜索其中所有的数字并返回一个切片.

|  |
| --- |
| var digitRegexp = regexp.MustCompile("[0-9]+") func main() {     FindDigits("./test.txt") } func FindDigits(filename string) []byte {     b, \_ := ioutil.ReadFile(filename)     return digitRegexp.Find(b) } |

* 这段代码可以顺利运行,但返回的[]byte指向的底层是整个文件的数据.只要该返回的切片不被释放,垃圾回收器就不能释放整个文件所占用的内存.换句话说,一点点有用的数据却占用了整个文件的内存.
* 想要避免这个问题,可以通过拷贝我们需要的部分到一个新的切片中:

|  |
| --- |
| var digitRegexp = regexp.MustCompile("[0-9]+") func main() {     FindDigits("./test.txt") } func FindDigits(filename string) []byte {     b, \_ := ioutil.ReadFile(filename)     return digitRegexp.Find(b)     c := make([]byte, len(b))     copy(c, b)     return c } |

## Map

### 基础概念

* map 传递给函数的代价很小：在 32 位机器上占 4 个字节，64 位机器上占 8 个字节，无论实际上存储了多少数据。
* 通过 key 在 map 中寻找值是很快的，比线性查找快得多，但是仍然比从数组和切片的索引中直接读取要慢 100 倍；所以如果你很在乎性能的话还是建议用切片来解决问题。
* 如果 key1 是 map1 的key，那么 map1[key1] 就是对应 key1 的值，就如同数组索引符号一样（数组可以视为一种简单形式的 map，key 是从 0 开始的整数）。
* 常用的 len(map1) 方法可以获得 map 中的 pair 数目，这个数目是可以伸缩的，因为 map-pairs 在运行时可以动态添加和删除。
* 不要使用 new，永远用 make 来构造 map
* 注意 如果你错误的使用 new() 分配了一个引用对象，你会获得一个空引用的指针，相当于声明了一个未初始化的变量并且取了它的地址：
* mapCreated := new(map[string]float32)
* map 类型是不存在锁的机制来实现这种效果(出于对性能的考虑),所以 map 类型是非线程安全的.当并行访问一个共享的 map 类型的数据，map 数据将会出错

### 提问要点

* 未初始化的map打印出来是怎样？map[]吗？初始化后的呢？

|  |
| --- |
| func main() {     var m map[int]int     fmt.Println(m)     //m = make(map[int]int)     m = make(map[int]int, 1)     fmt.Println(m) } /\*  \* 运行结果  \* map[]  \* map[]  \*/ |

* Map是值类型还是引用类型?

|  |
| --- |
| func main() {     m := map[int]string{1: "aaa", 2: "bbb"}     fmt.Println(m)     mytest(m)     fmt.Println(m) } func mytest(m map[int]string) {     m[1] = "ccc" } /\*  \* 运行结果  \* map[1:aaa 2:bbb]  \* map[1:ccc 2:bbb]  \*/ |

## 指针

1.指向数组的指针和指针数组

|  |
| --- |
| func main() {     a := [...]int{9: 1}     var p \*[10]int = &a //p是指向数组的指针     fmt.Println(p)      x, y := 1, 2     b := [...]\*int{&x, &y} //b是指针数组     fmt.Println(b) } |

2. 哪些是值传递,哪些是引用传递?(例如,struct,map,slice,数组,接口,基本类型等) 什么类型的变量不需要(\*s)这样应用可以直接s,哪些些只能\*s ?

* 基本数据,如int,float,string等是值类型,基本类型如果是指针都需要\*s
* 数组是值类型，不管是指向数组本身还是指向数组本身，都可以用a[1]这种方式，指针不需要前面加\*
* 结构体是值类型,结构体无论是值还是指针,都可以直接使用s
* 切片是引用类型,
* Map是引用类型,
* Channel是引用类型

3.函数传递

* 传指针使得多个函数能操作同一个对象。
* 传指针比较轻量级 (8bytes),只是传内存地址，我们可以用指针传递体积大的结构体。如果用参数值传递的话, 在每次copy上面就会花费相对较多的系统开销（内存和时间）。所以当你要传递大的结构体的时候，用指针是一个明智的选择。
* Go语言中channel，slice，map这三种类型的实现机制类似指针，所以可以直接传递，而不用取地址后传递指针。（注：若函数需改变slice的长度，则仍需要取地址传递指针）

## 包

### 基础概念

* 导入的包在包自身初始化前被初始化，而一个包在程序执行中只能初始化一次。

1. 实验多个代码文件共用同一个包，并且有init函数?

当引入包有init函数的时候,优先执行该init函数.并且当同个项目中多次引入同个包时,只执行一次init函数.

2. golang包名必须与所在文件夹同名吗?

不必须，但是同个目录中（不含子目录）的所有文件包名必须一致，通常为了方便包定位，建议包名和目录名一致，否则你import "A"，使用起来B.xxx，看上去不统一，不能一眼看出来这个B包是哪个文件的。

1. src目录下有mytest文件夹，里面有mytest.go包，引用该包的时候为什么是import "mytest" 而不是import "mytest/mytest" ？请上机实验.

import引号后面的只是包所在路径,具体使用的时候是使用该路径下的包名(一般最好和所在文件夹名称一致)来使用. 例如fmt.Println(“hello,world”) 我们调用了fmt包里面定义的函数Println。大家可以看到，这个函数是通过<pkgName>.<funcName>的方式调用的，这一点和Python十分相似。前面提到过，包名和包所在的文件夹名可以是不同的，此处的<pkgName>即为通过package <pkgName>声明的包名，而非文件夹名。

1. GOPATH目录下三个子目录,src,pkg,bin,当我们引入第三方包时,源代码都是在src目录下,如果安装该包,那么编译后的包文件会在pkg子目录下,但是为什么使用该包的时候还是需要源代码存在?而不是只要pkg子目录下的\*.a包文件就可以了?
2. 定义在包中函数内部的大写字母开头的变量，包外可以访问吗？

不行,必须是包中函数外部的公有变量,包外部才可以访问.

6.结构体中变量的首字母大小写，包外部访问性如何？

不可以直接访问;

如果该结构体变量定义在包中,并且该结构体变量名为可访问性质,那么包外可以直接访问整个结构体,但是不能访问结构体内部的变量;

如果如果结构体变量定义在包外部,那么连这个结构体变量都不能定义

7.引入一个包的执行流程?



## 结构体

### 基本概念

* Go 语言中，结构体和它所包含的数据在内存中是以连续块的形式存在的，即使结构体中嵌套有其他的结构体，这在性能上带来了很大的优势。不像 Java 中的引用类型，一个对象和它里面包含的对象可能会在不同的内存空间中，这点和 Go 语言中的指针很像。
* 如果想知道结构体类型T的一个实例占用了多少内存，可以使用：size := unsafe.Sizeof(T{})
* 结构体序列化到json字符串时,指定某些字段如果为空则不序列话,或者指定某字段不序列化

|  |
| --- |
| type Message struct {     Name string `json:"msg\_name"`       // 对应JSON的msg\_name     Body string `json:"body,omitempty"` // 如果为空则忽略字段     Time int64  `json:"json:"-"`        // 直接忽略字段 } |

### 结构体工厂

* Go 语言不支持面向对象编程语言中那样的构造子方法，但是可以很容易的在 Go 中实现 “构造子工厂”方法。为了方便通常会为类型定义一个工厂，按惯例，工厂的名字以 new 或 New 开头。假设定义了如下的 File 结构体类型

|  |
| --- |
| type File struct {     fd      int     // 文件描述符     name    string  // 文件名 } |

* 下面是这个结构体类型对应的工厂方法，它返回一个指向结构体实例的指针：

|  |
| --- |
| func NewFile(fd int, name string) \*File {     if fd < 0 {         return nil     }     return &File{fd, name} } |

* 在 Go 语言中常常像上面这样在工厂方法里使用初始化来简便的实现构造函数。
* 如果 File 是一个结构体类型，那么表达式 new(File) 和 &File{} 是等价的。
* 如何强制使用工厂方法

|  |
| --- |
| 通过应用可见性规则,就可以禁止使用 new 函数，强制用户使用工厂方法，从而使类型变成私有的，就像在面向对象语言中那样。 type matrix struct {     ... } func NewMatrix(params) \*matrix {     m := new(matrix) // 初始化 m     return m } 在其他包里使用工厂方法： package main import "matrix" ... wrong := new(matrix.matrix)     // 编译失败（matrix 是私有的） right := matrix.NewMatrix(...)  // 实例化 matrix 的唯一方式 |

### 匿名字段和内嵌结构体

* 结构体可以包含一个或多个 匿名（或内嵌）字段，即这些字段没有显式的名字，只有字段的类型是必须的，此时类型就是字段的名字。匿名字段本身可以是一个结构体类型，即 结构体可以包含内嵌结构体
* 可以粗略地将这个和面向对象语言中的继承概念相比较，随后将会看到它被用来模拟类似继承的行为。Go 语言中的继承是通过内嵌或组合来实现的，所以可以说，在 Go 语言中，相比较于继承，组合更受青睐。
* 在一个结构体中对于每一种数据类型只能有一个匿名字段。(如下例子的outer.int)

|  |
| --- |
| type innerS struct {     in1 int     in2 int } type outerS struct {     b    int     c    float32     int  // anonymous field     innerS //anonymous field } func main() {     outer := new(outerS)     outer.b = 6     outer.c = 7.5     outer.int = 60     outer.in1 = 5     outer.in2 = 10     fmt.Printf("outer.b is: %d\n", outer.b)     fmt.Printf("outer.c is: %f\n", outer.c)     fmt.Printf("outer.int is: %d\n", outer.int)     fmt.Printf("outer.in1 is: %d\n", outer.in1)     fmt.Printf("outer.in2 is: %d\n", outer.in2)     // 使用结构体字面量     outer2 := outerS{6, 7.5, 60, innerS{5, 10}}     fmt.Println("outer2 is:", outer2) } |

### 命名冲突

* 当两个字段拥有相同的名字（可能是继承来的名字）时该怎么办呢？
* 外层名字会覆盖内层名字（但是两者的内存空间都保留），这提供了一种重载字段或方法的方式；
* 如果相同的名字在同一级别出现了两次，如果这个名字被程序使用了，将会引发一个错误（不使用没关系）。没有办法来解决这种问题引起的二义性，必须由程序员自己修正

|  |
| --- |
| 例子： type A struct {a int} type B struct {a, b int} type C struct {A; B} var c C; 规则 2：使用 c.a 是错误的，到底是 c.A.a 还是 c.B.a 呢？会导致编译器错误：ambiguous DOT reference c.a disambiguate with either c.A.a or c.B.a。 type D struct {B; b float32} var d D; 规则1：使用 d.b 是没问题的：它是 float32，而不是 B 的 b。如果想要内层的 b 可以通过 d.B.b 得到。 |

### 方法

* Go 方法是作用在接收者（receiver）上的一个函数，接收者是某种类型的变量。因此方法是一种特殊类型的函数。
* 接收者类型可以是（几乎）任何类型，不仅仅是结构体类型：任何类型都可以有方法，甚至可以是函数类型，可以是 int、bool、string 或数组的别名类型。但是接收者不能是一个接口类型（参考 第 11 章），因为接口是一个抽象定义，但是方法却是具体实现；
* 最后接收者不能是一个指针类型，但是它可以是任何其他允许类型的指针。
* 一个类型加上它的方法等价于面向对象中的一个类。一个重要的区别是：在 Go 中，类型的代码和绑定在它上面的方法的代码可以不放置在一起，它们可以存在在不同的源文件，唯一的要求是：它们必须是同一个包的。
* 类型 T（或 \*T）上的所有方法的集合叫做类型 T（或 \*T）的方法集。
* 因为方法是函数，所以同样的，不允许方法重载，即对于一个类型只能有一个给定名称的方法。但是如果基于接收者类型，是有重载的：具有同样名字的方法可以在 2 个或多个不同的接收者类型上存在;
* 别名类型不能有它原始类型上已经定义过的方法;

### 函数和方法的区别

* 函数将变量作为参数：Function1(recv)
* 方法在变量上被调用：recv.Method1()
* 在接收者是指针时，方法可以改变接收者的值（或状态），这点函数也可以做到（当参数作为指针传递，即通过引用调用时，函数也可以改变参数的状态）。
* 不要忘记 Method1 后边的括号 ()，否则会引发编译器错误：method recv.Method1 is not an expression, must be called
* 接收者必须有一个显式的名字，这个名字必须在方法中被使用。
* receiver\_type 叫做 （接收者）基本类型，这个类型必须在和方法同样的包中被声明。
* 在 Go 中，（接收者）类型关联的方法不写在类型结构里面，就像类那样；耦合更加宽松；类型和方法之间的关联由接收者来建立。
* 方法没有和数据定义（结构体）混在一起：它们是正交的类型；表示（数据）和行为（方法）是独立的。

### 指针或值作为接收者

* 指针方法和值方法都可以在指针或非指针上被调用，如下面程序所示，类型 List 在值上有一个方法 Len()，在指针上有一个方法 Append()，但是可以看到两个方法都可以在两种类型的变量上被调用。

|  |
| --- |
| type List []int func (l List) Len() int        { return len(l) } func (l \*List) Append(val int) { \*l = append(\*l, val) } func main() {     // 值     var lst List     lst.Append(1)     fmt.Printf("%v (len: %d)", lst, lst.Len()) // [1] (len: 1)     // 指针     plst := new(List)     plst.Append(2)     fmt.Printf("%v (len: %d)", plst, plst.Len()) // &[2] (len: 1) } |

### 内嵌类型的方法和继承

* 当一个匿名类型被内嵌在结构体中时，匿名类型的可见方法也同样被内嵌，这在效果上等同于外层类型 继承 了这些方法：将父类型放在子类型中来实现亚型。这个机制提供了一种简单的方式来模拟经典面向对象语言中的子类和继承相关的效果，也类似 Ruby 中的混入（mixin）
* 内嵌结构体上的方法可以直接在外层类型的实例上调用

|  |
| --- |
| type Point struct {     x, y float64 } func (p \*Point) Abs() float64 {     return math.Sqrt(p.x\*p.x + p.y\*p.y) } type NamedPoint struct {     Point     name string } func main() {     n := &NamedPoint{Point{3, 4}, "Pythagoras"}     fmt.Println(n.Abs()) // 打印5 } |

* 内嵌将一个已存在类型的字段和方法注入到了另一个类型里：匿名字段上的方法“晋升”成为了外层类型的方法。当然类型可以有只作用于本身实例而不作用于内嵌“父”类型上的方法，
* 可以覆写方法（像字段一样）：和内嵌类型方法具有同样名字的外层类型的方法会覆写内嵌类型对应的方法。如上例中添加如下方法

|  |
| --- |
| func (n \*NamedPoint) Abs() float64 {     return n.Point.Abs() \* 100. } |

* 因为一个结构体可以嵌入多个匿名类型，所以实际上我们可以有一个简单版本的多重继承

### 多重继承

* 多重继承指的是类型获得多个父类型行为的能力，它在传统的面向对象语言中通常是不被实现的（C++ 和 Python 例外）。因为在类继承层次中，多重继承会给编译器引入额外的复杂度。但是在 Go 语言中，通过在类型中嵌入所有必要的父类型，可以很简单的实现多重继承。
* 总结
* 在 Go 中，类型就是类（数据和关联的方法）。Go 不知道类似面向对象语言的类继承的概念。继承有两个好处：代码复用和多态。
* 在 Go 中，代码复用通过组合和委托实现，多态通过接口的使用来实现：有时这也叫 组件编程（Component Programming）。
* 许多开发者说相比于类继承，Go 的接口提供了更强大、却更简单的多态行为。
* 如果真的需要更多面向对象的能力，看一下 goop 包（Go Object-Oriented Programming），它由 Scott Pakin 编写: 它给 Go 提供了 JavaScript 风格的对象（基于原型的对象），并且支持多重继承和类型独立分派，通过它可以实现你喜欢的其他编程语言里的一些结构。

### 类型的Strings()方法和格式化描述符

* 当你广泛使用一个自定义类型时，最好为它定义 String()方法。从上面的例子也可以看到，格式化描述符 %T 会给出类型的完全规格，%#v 会给出实例的完整输出，包括它的字段（在程序自动生成 Go 代码时也很有用）

|  |
| --- |
| type TwoInts struct {     a int     b int } func main() {     two1 := new(TwoInts)     two1.a = 12     two1.b = 10     fmt.Printf("two1 is: %v\n", two1)     fmt.Println("two1 is:", two1)     fmt.Printf("two1 is: %T\n", two1)     fmt.Printf("two1 is: %#v\n", two1) } func (tn \*TwoInts) String() string {     return "(" + strconv.Itoa(tn.a) + "/" + strconv.Itoa(tn.b) + ")" } |

* 不要在 String() 方法里面调用涉及 String() 方法的方法，它会导致意料之外的错误，比如下面的例子，它导致了一个无限迭代（递归）调用（TT.String() 调用 fmt.Sprintf，而 fmt.Sprintf 又会反过来调用 TT.String()...），很快就会导致内存溢出：

|  |
| --- |
| type TT float64 func (t TT) String() string {     return fmt.Sprintf("%v", t) } t. String() |

### 垃圾回收和 SetFinalizer

* Go 开发者不需要写代码来释放程序中不再使用的变量和结构占用的内存，在 Go 运行时中有一个独立的进程，即垃圾收集器（GC），会处理这些事情，它搜索不再使用的变量然后释放它们的内存。可以通过 runtime 包访问 GC 进程。
* 通过调用 runtime.GC() 函数可以显式的触发 GC，但这只在某些罕见的场景下才有用，比如当内存资源不足时调用 runtime.GC()，它会在此函数执行的点上立即释放一大片内存，此时程序可能会有短时的性能下降（因为 GC 进程在执行）。
* 如果想知道当前的内存状态，可以使用：

|  |
| --- |
| var m runtime.MemStats runtime.ReadMemStats(&m) fmt.Printf("%d Kb\n", m.Alloc / 1024) |

* 上面的程序会给出已分配内存的总量，单位是 Kb。进一步的测量参考 文档页面。
* 如果需要在一个对象 obj 被从内存移除前执行一些特殊操作，比如写到日志文件中，可以通过如下方式调用函数来实现：

|  |
| --- |
| runtime.SetFinalizer(obj, func(obj \*typeObj)) |

* func(obj \*typeObj) 需要一个 typeObj 类型的指针参数 obj，特殊操作会在它上面执行。func 也可以是一个匿名函数。
* 在对象被 GC 进程选中并从内存中移除以前，SetFinalizer 都不会执行，即使程序正常结束或者发生错误。

## 接口

### 基本概念

* 接口定义了一组方法（方法集），但是这些方法不包含（实现）代码：它们没有被实现(它们是抽象的)。接口里也不能包含变量。
* （按照约定，只包含一个方法的）接口的名字由方法名加 [e]r 后缀组成，例如 Printer、Reader、Writer、Logger、Converter 等等。还有一些不常用的方式（当后缀 er 不合适时），比如 Recoverable，此时接口名以 able 结尾，或者以 I 开头（像 .NET 或 Java 中那样）。
* Go 语言中的接口都很简短，通常它们会包含 0 个、最多 3 个方法。
* 不像大多数面向对象编程语言，在 Go 语言中接口可以有值，一个接口类型的变量或一个 接口值 ：var ai Namer，ai 是一个多字（multiword）数据结构，它的值是 nil。它本质上是一个指针，虽然不完全是一回事。指向接口值的指针是非法的，它们不仅一点用也没有，还会导致代码错误。
* 类型不需要显式声明它实现了某个接口：接口被隐式地实现。多个类型可以实现同一个接口。
* 实现某个接口的类型（除了实现接口方法外）可以有其他的方法。
* 一个类型可以实现多个接口。
* 接口类型可以包含一个实例的引用， 该实例的类型实现了此接口（接口是动态类型）。
* 即使接口在类型之后才定义，二者处于不同的包中，被单独编译：只要类型实现了接口中的方法，它就实现了此接口。
* 接口可以匿名嵌入其它接口,或嵌入到结构体中
* 将对象赋值给接口时,会发生拷贝,而接口内部存储的是指向这个复制品的指针,既无法修改原始对象的状态,也无法获取指针
* 接口调用不会做receiver的自动转换
* 只有当接口存储的类型和对象都为nil时,接口才等于nil

|  |
| --- |
| func main() {     var a interface{}     fmt.Println(a == nil)     var p \*int = nil     a = p     fmt.Println(a == nil) } /\*  \* 运行结果  \* true  \* false  \*/ |

### 接口嵌套接口

* 一个接口可以包含一个或多个其他的接口，这相当于直接将这些内嵌接口的方法列举在外层接口中一样。
* 比如接口 File 包含了 ReadWrite 和 Lock 的所有方法，它还额外有一个 Close() 方法。

|  |
| --- |
| type ReadWrite interface {     Read(b Buffer) bool     Write(b Buffer) bool } type Lock interface {     Lock()     Unlock() } type File interface {     ReadWrite     Lock     Close() } |

### 类型段誉:如何检测和转换接口变量的类型

* 一个接口类型的变量 varI 中可以包含任何类型的值，必须有一种方式来检测它的 动态 类型，即运行时在变量中存储的值的实际类型。在执行过程中动态类型可能会有所不同，但是它总是可以分配给接口变量本身的类型。通常我们可以使用 类型断言 来测试在某个时刻 varI 是否包含类型 T 的值：

|  |
| --- |
| v := varI.(T)       // unchecked type assertion |

* varI 必须是一个接口变量，否则编译器会报错：invalid type assertion: varI.(T) (non-interface type (type of varI) on left) 。
* 类型断言可能是无效的，虽然编译器会尽力检查转换是否有效，但是它不可能预见所有的可能性。如果转换在程序运行时失败会导致错误发生。更安全的方式是使用以下形式来进行类型断言：

|  |
| --- |
| if v, ok := varI.(T); ok {  // checked type assertion     Process(v)     return } // varI is not of type T |

* 如果转换合法，v 是 varI 转换到类型 T 的值，ok 会是 true；否则 v 是类型 T 的零值，ok 是 false，也没有运行时错误发生。
* 应该总是使用上面的方式来进行类型断言。

### 类型断言:type switch

* 接口变量的类型也可以使用一种特殊形式的 switch 来检测：type-switch

|  |
| --- |
| switch t := areaIntf.(type) { case \*Square:     fmt.Printf("Type Square %T with value %v\n", t, t) case \*Circle:     fmt.Printf("Type Circle %T with value %v\n", t, t) case nil:     fmt.Printf("nil value: nothing to check?\n") default:     fmt.Printf("Unexpected type %T\n", t) } |

* 可以用 type-switch 进行运行时类型分析，但是在 type-switch 不允许有 fallthrough

### 测试一个值是否实现了某接口

* 假定 v 是一个值，然后我们想测试它是否实现了 Stringer 接口，可以这样做：

|  |
| --- |
| type Stringer interface {     String() string } if sv, ok := v.(Stringer); ok {     fmt.Printf("v implements String(): %s\n", sv.String()) // note: sv, not v } |

* 接口是一种契约，实现类型必须满足它，它描述了类型的行为，规定类型可以做什么。接口彻底将类型能做什么，以及如何做分离开来，使得相同接口的变量在不同的时刻表现出不同的行为，这就是多态的本质。
* 编写参数是接口变量的函数，这使得它们更具有一般性。
* 使用接口使代码更具有普适性。
* 标准库里到处都使用了这个原则，如果对接口概念没有良好的把握，是不可能理解它是如何构建的。

### 使用方法集与接口

* 作用于变量上的方法实际上是不区分变量到底是指针还是值的。当碰到接口类型值时，这会变得有点复杂，原因是接口变量中存储的具体值是不可寻址的，幸运的是，如果使用不当编译器会给出错误。考虑下面的程序：

|  |
| --- |
| type List []int func (l List) Len() int {     return len(l) } func (l \*List) Append(val int) {     \*l = append(\*l, val) } type Appender interface {     Append(int) } func CountInto(a Appender, start, end int) {     for i := start; i <= end; i++ {         a.Append(i)     } } type Lener interface {     Len() int } func LongEnough(l Lener) bool {     return l.Len()\*10 > 42 } func main() {     // A bare value     var lst List     // compiler error:     // cannot use lst (type List) as type Appender in argument to CountInto:     //       List does not implement Appender (Append method has pointer receiver)     // CountInto(lst, 1, 10)     if LongEnough(lst) { // VALID:Identical receiver type         fmt.Printf("- lst is long enough\n")     }     // A pointer value     plst := new(List)     CountInto(plst, 1, 10) //VALID:Identical receiver type     if LongEnough(plst) {         // VALID: a \*List can be dereferenced for the receiver         fmt.Printf("- plst is long enough\n")     } } |

* 在 lst 上调用 CountInto 时会导致一个编译器错误，因为 CountInto 需要一个 Appender，而它的方法 Append 只定义在指针上。 在 lst 上调用 LongEnough 是可以的因为 'Len' 定义在值上。
* 在 plst 上调用 CountInto 是可以的，因为 CountInto 需要一个 Appender，并且它的方法 Append 定义在指针上。 在 plst 上调用 LongEnough 也是可以的，因为指针会被自动解引用。
* 在**接口**上调用方法时，必须有和方法定义时相同的接收者类型或者是可以从具体类型 P 直接可以辨识的：
* 指针方法可以通过指针调用
* 值方法可以通过值调用
* 接收者是值的方法可以通过指针调用，因为指针会首先被解引用
* 接收者是指针的方法不可以通过值调用，因为存储在接口中的值没有地址
* 将一个值赋值给一个接口时，编译器会确保所有可能的接口方法都可以在此值上被调用，因此不正确的赋值在编译期就会失败。
* Go 语言规范定义了接口方法集的调用规则:
* 类型 \*T 的可调用方法集包含接受者为 \*T 或 T 的所有方法集
* 类型 T 的可调用方法集包含接受者为 T 的所有方法
* 类型 T 的可调用方法集不包含接受者为 \*T 的方法

### 空接口

* 每个 interface {} 变量在内存中占据两个字长：一个用来存储它包含的类型，另一个用来存储它包含的数据或者指向数据的指针。

### 复制数据切片至空接口切片

* 假设你有一个 myType 类型的数据切片，你想将切片中的数据复制到一个空接口切片中，类似：

|  |
| --- |
| var dataSlice []myType = FuncReturnSlice() var interfaceSlice []interface{} = dataSlice |

* 可惜不能这么做，编译时会出错：cannot use dataSlice (type []myType) as type []interface { } in assignment。原因是它们俩在内存中的布局是不一样的（参考 官方说明）。必须使用 for-range 语句来一个一个显式地复制：

|  |
| --- |
| var dataSlice []myType = FuncReturnSlice() var interfaceSlice []interface{} = make([]interface{}, len(dataSlice)) for ix, d := range dataSlice {     interfaceSlice[ix] = d } |

### 接口与动态类型

* Go的动态类型
* 和其它语言相比，Go 是唯一结合了接口值，静态类型检查（是否该类型实现了某个接口），运行时动态转换的语言，并且不需要显式地声明类型是否满足某个接口。该特性允许我们在不改变已有的代码的情况下定义和使用新接口。
* 接收一个（或多个）接口类型作为参数的函数，其实参可以是任何实现了该接口的类型。 实现了某个接口的类型可以被传给任何以此接口为参数的函数 。
* 类似于 Python 和 Ruby 这类动态语言中的 动态类型（duck typing）；这意味着对象可以根据提供的方法被处理（例如，作为参数传递给函数），而忽略它们的实际类型：它们能做什么比它们是什么更重要。
* 示例

|  |
| --- |
| type IDuck interface {     Quack()     Walk() } func DuckDance(duck IDuck) {     duck.Quack()     duck.Walk() } type Bird struct {     Name string } func (b \*Bird) Quack() {     fmt.Println("I am quacking by",b.Name) } func (b \*Bird) Walk() {     fmt.Println("I am walking by",b.Name) } func main() {     b := new(Bird)     b.Name="bird"     DuckDance(b) } /\*\*  \* 运行结果  \* I am quacking by bird  \* I am walking by bird  \*/ |

### 动态方法调用

* 像 Python，Ruby 这类语言，动态类型是延迟绑定的（在运行时进行）：方法只是用参数和变量简单地调用，然后在运行时才解析（它们很可能有像 responds\_to 这样的方法来检查对象是否可以响应某个方法，但是这也意味着更大的编码量和更多的测试工作）
* Go 的实现与此相反，通常需要编译器静态检查的支持：当变量被赋值给一个接口类型的变量时，编译器会检查其是否实现了该接口的所有函数。
* 因此 Go 提供了动态语言的优点，却没有其他动态语言在运行时可能发生错误的缺点。
* Go 的接口提高了代码的分离度，改善了代码的复用性，使得代码开发过程中的设计模式更容易实现。用 Go 接口还能实现 依赖注入模式。

|  |
| --- |
| type xmlWriter interface {     WriteXML(w io.Writer) error } // Exported XML streaming function. func StreamXML(v interface{}, w io.Writer) error {     if xw, ok := v.(xmlWriter); ok {         // It’s an  xmlWriter, use method of asserted type.         return xw.WriteXML(w)     }     // No implementation, so we have to use our own function (with perhaps reflection):     return encodeToXML(v, w) } // Internal XML encoding function. func encodeToXML(v interface{}, w io.Writer) error {     // ... } |

### 接口的提取

* 提取接口 是非常有用的设计模式，可以减少需要的类型和方法数量，而且不需要像传统的基于类的面向对象语言那样维护整个的类层次结构。
* Go 接口可以让开发者找出自己写的程序中的类型。假设有一些拥有共同行为的对象，并且开发者想要抽象出这些行为，这时就可以创建一个接口来使用。 假设我们需要一个新的接口 TopologicalGenus，用来给 shape 排序（这里简单地实现为返回 int）。我们需要做的是给想要满足接口的类型实现 Rank() 方法：

|  |
| --- |
| type Shaper interface {     Area() float32 } type TopologicalGenus interface {     Rank() int } type Square struct {     side float32 } func (sq \*Square) Area() float32 {     return sq.side \* sq.side } func (sq \*Square) Rank() int {     return 1 } type Rectangle struct {     length, width float32 } func (r Rectangle) Area() float32 {     return r.length \* r.width } func (r Rectangle) Rank() int {     return 2 } func main() {     r := Rectangle{5, 3} // Area() of Rectangle needs a value     q := &Square{5}      // Area() of Square needs a pointer     shapes := []Shaper{r, q}     fmt.Println("Looping through shapes for area ...")     for n, \_ := range shapes {         fmt.Println("Shape details: ", shapes[n])         fmt.Println("Area of this shape is: ", shapes[n].Area())     }     topgen := []TopologicalGenus{r, q}     fmt.Println("Looping through topgen for rank ...")     for n, \_ := range topgen {         fmt.Println("Shape details: ", topgen[n])         fmt.Println("Topological Genus of this shape is: ", topgen[n].Rank())     } } /\*\*  \* 运行结果  \* Looping through shapes for area ...  \* Shape details:  {5 3}  \* Area of this shape is:  15  \* Shape details:  &{5}  \* Area of this shape is:  25  \* Looping through topgen for rank ...  \* Shape details:  {5 3}  \* Topological Genus of this shape is:  2  \* Shape details:  &{5}  \* Topological Genus of this shape is:  1  \*/ |

* 所以你不用提前设计出所有的接口；整个设计可以持续演进，而不用废弃之前的决定。类型要实现某个接口，它本身不用改变，你只需要在这个类型上实现新的方法

### 显示地指明类型实现了某个接口

* 如果你希望满足某个接口的类型显式地声明它们实现了这个接口，你可以向接口的方法集中添加一个具有描述性名字的方法。例如：

|  |
| --- |
| type Fooer interface {     Foo()     ImplementsFooer() } |

* 类型 Bar 必须实现 ImplementsFooer 方法来满足 Footer 接口，以清楚地记录这个事实。

|  |
| --- |
| type Bar struct{} func (b Bar) ImplementsFooer() {}  func (b Bar) Foo() {} |

* 大部分代码并不使用这样的约束，因为它限制了接口的实用性。但是有些时候，这样的约束在大量相似的接口中被用来解决歧义。

### 空接口和函数重载

* 我们看到函数重载是不被允许的。在 Go 语言中函数重载可以用可变参数 ...T 作为函数最后一个参数来实现。如果我们把 T 换为空接口，那么可以知道任何类型的变量都是满足 T (空接口）类型的，这样就允许我们传递任何数量任何类型的参数给函数，即重载的实际含义。函数 fmt.Printf 就是这样做的：

|  |
| --- |
| fmt.Printf(format string, a ...interface{}) (n int, errno error) |

* 这个函数通过枚举 slice 类型的实参动态确定所有参数的类型。并查看每个类型是否实现了 String() 方法，如果是就用于产生输出信息。

### 接口的继承

* 当一个类型包含（内嵌）另一个类型（实现了一个或多个接口）的指针时，这个类型就可以使用（另一个类型）所有的接口方法。例如：

|  |
| --- |
| type Task struct {     Command string     \*log.Logger } |

* 这个类型的工厂方法像这样：

|  |
| --- |
| func NewTask(command string, logger \*log.Logger) \*Task {     return &Task{command, logger} } |

* 当 log.Logger 实现了 Log() 方法后，Task 的实例 task 就可以调用该方法：**task.Log()**
* 类型可以通过继承多个接口来提供像 多重继承 一样的特性：

|  |
| --- |
| type ReaderWriter struct {     \*io.Reader     \*io.Writer } |

* 上面概述的原理被应用于整个 Go 包，多态用得越多，代码就相对越少。这被认为是 Go 编程中的重要的最佳实践。
* 有用的接口可以在开发的过程中被归纳出来。添加新接口非常容易，因为已有的类型不用变动（仅仅需要实现新接口的方法）。已有的函数可以扩展为使用接口类型的约束性参数：通常只有函数签名需要改变。对比基于类的 OO 类型的语言在这种情况下则需要适应整个类层次结构的变化。

### Go中的面向对象

* Go 没有类，而是松耦合的类型、方法对接口的实现;
* OO 语言最重要的三个方面分别是：封装，继承和多态，在 Go 中它们是怎样表现的呢？
* 封装(数据隐藏)：和别的 OO 语言有 4 个或更多的访问层次相比，Go 把它简化为了 2 层
* 包范围内的：通过标识符首字母小写，对象 只在它所在的包内可见
* 可导出的：通过标识符首字母大写，对象 对所在包以外也可见
* 继承：用组合实现：内嵌一个（或多个）包含想要的行为（字段和方法）的类型；多重继承可以通过内嵌多个类型实现
* 多态：用接口实现：某个类型的实例可以赋给它所实现的任意接口类型的变量。类型和接口是松耦合的，并且多重继承可以通过实现多个接口实现。Go 接口不是 Java 和 C# 接口的变体，而且：接口间是不相关的，并且是大规模编程和可适应的演进型设计的关键

### ???结构体,集合和高阶函数

|  |
| --- |
| 通常你在应用中定义了一个结构体，那么你也可能需要这个结构体的（指针）对象集合，比如：  type Any interface{} type Car struct {     Model        string     Manufacturer string     BuildYear    int     // ... }  type Cars []\*Car 在定义所需功能时我们可以利用函数可以作为（其它函数的）参数的事实来使用高阶函数，例如：  1）定义一个通用的 Process() 函数，它接收一个作用于每一辆 car 的 f 函数作参数：  // Process all cars with the given function f: func (cs Cars) Process(f func(car \*Car)) {     for \_, c := range cs {         f(c)     } } 2）在上面的基础上，实现一个查找函数来获取子集合，并在 Process() 中传入一个闭包执行（这样就可以访问局部切片 cars）：  // Find all cars matching a given criteria. func (cs Cars) FindAll(f func(car \*Car) bool) Cars {      cars := make([]\*Car, 0)     cs.Process(func(c \*Car) {             if f(c) {                 append(cars,c)             }     )     return cars } 3）实现 Map 功能，产出除 car 对象以外的东西：  // Process cars and create new data. func (cs Cars) Map(f func(car \*Car) Any) []Any {         result := make([]Any, 0)         ix := 0         cs.Process(func(c \*Car) {                 result[ix] = f(c)                 ix++         })         return result } 现在我们可以定义下面这样的具体查询：  allNewBMWs := allCars.FindAll(func(car \*Car) bool {        return (car.Manufacturer == “BMW”) && (car.BuildYear > 2010) }) 4）我们也可以根据入参返回不同的函数。也许我们想根据不同的厂商添加汽车到不同的集合，但是这可能会是多变的。所以我们可以定义一个函数来产生特定的添加函数和 map 集：  func MakeSortedAppender(manufacturers[]string)(func(car\*Car),map[string]Cars) {     // Prepare maps of sorted cars.     sortedCars := make(map[string]Cars)     for \_, m := range manufacturers {         sortedCars[m] = make([]\*Car, 0)     }     sortedCars[“Default”] = make([]\*Car, 0)     // Prepare appender function:     appender := func(c \*Car) {         if \_, ok := sortedCars[c.Manufacturer]; ok {             sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)         } else {             sortedCars[“Default”] = append(sortedCars[“Default”], c)         }      }     return appender, sortedCars } 现在我们可以用它把汽车分类为独立的集合，像这样：  manufacturers := []string{“Ford”, “Aston Martin”, “Land Rover”, “BMW”, “Jaguar”} sortedAppender, sortedCars := MakeSortedAppender(manufacturers) allUnsortedCars.Process(sortedAppender) BMWCount := len(sortedCars[“BMW”]) 我们让这些代码在下面的程序 cars.go 中执行：  示例 11.18 cars.go：  // cars.go package main  import (     "fmt" )  type Any interface{} type Car struct {     Model       string     Manufacturer    string     BuildYear   int     // ... } type Cars []\*Car  func main() {     // make some cars:     ford := &Car{"Fiesta","Ford", 2008}     bmw  := &Car{"XL 450", "BMW", 2011}     merc := &Car{"D600", "Mercedes", 2009}     bmw2 := &Car{"X 800", "BMW", 2008}     // query:     allCars := Cars([]\*Car{ford, bmw, merc, bmw2})     allNewBMWs := allCars.FindAll(func(car \*Car) bool {       return (car.Manufacturer == "BMW") && (car.BuildYear > 2010)     })     fmt.Println("AllCars: ", allCars)     fmt.Println("New BMWs: ", allNewBMWs)     //     manufacturers := []string{"Ford", "Aston Martin", "Land Rover", "BMW", "Jaguar"}     sortedAppender, sortedCars := MakeSortedAppender(manufacturers)     allCars.Process(sortedAppender)     fmt.Println("Map sortedCars: ", sortedCars)     BMWCount := len(sortedCars["BMW"])     fmt.Println("We have ", BMWCount, " BMWs") }  // Process all cars with the given function f: func (cs Cars) Process(f func(car \*Car)) {      for \_, c := range cs {          f(c)      } }  // Find all cars matching a given criteria. func (cs Cars) FindAll(f func(car \*Car) bool) Cars {     cars := make([]\*Car, 0)      cs.Process(func(c \*Car) {         if f(c) {             cars = append(cars, c)         }     })     return cars }  // Process cars and create new data. func (cs Cars) Map(f func(car \*Car) Any) []Any {        result := make([]Any, len(cs))        ix := 0        cs.Process(func(c \*Car) {            result[ix] = f(c)            ix++        })        return result }  func MakeSortedAppender(manufacturers []string) (func(car \*Car), map[string]Cars) {      // Prepare maps of sorted cars.        sortedCars := make(map[string]Cars)         for \_, m := range manufacturers {            sortedCars[m] = make([]\*Car, 0)        }        sortedCars["Default"] = make([]\*Car, 0)         // Prepare appender function:        appender := func(c \*Car) {            if \_, ok := sortedCars[c.Manufacturer]; ok {                sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)            } else {                sortedCars["Default"] = append(sortedCars["Default"], c)            }        }        return appender, sortedCars } 输出：  AllCars:  [0xf8400038a0 0xf840003bd0 0xf840003ba0 0xf840003b70] New BMWs:  [0xf840003bd0] Map sortedCars:  map[Default:[0xf840003ba0] Jaguar:[] Land Rover:[] BMW:[0xf840003bd0 0xf840003b70] Aston Martin:[] Ford:[0xf8400038a0]] We have  2  BMWs |

## 反射

### 基础概念

* 反射可以从接口值反射到对象，也可以从对象反射回接口值

### 方法和类型的反射

* Kind()
* kind总是返回底层类型(以下方法v.kind()返回reflect.Int)

|  |
| --- |
| func main() {     type MyInt int     var m MyInt = 5     v:=reflect.ValueOf(m)     fmt.Println(v.Kind()) } |

* Interface()
* 变量 v 的 Interface() 方法可以得到还原（接口）值，所以可以这样打印 v 的值：fmt.Println(v.Interface())
* 示例

|  |
| --- |
| func main() {     var x float64 = 3.4     fmt.Println("type:",reflect.TypeOf(x))     v:=reflect.ValueOf(x)     fmt.Println("value:",v)     fmt.Println("type:",v.Type())     fmt.Println("kind:",v.Kind())     //x 是一个 float64 类型的值，reflect.ValueOf(x).Float() 返回这个 float64 类型的实际值；同样的适用于 Int(), Bool(), Complex(), String()     fmt.Println("value:",v.Float())     fmt.Println(v.Interface())     fmt.Printf("value is %5.2e\n",v.Interface())     y:=v.Interface().(float64)     fmt.Println(y) } /\*\*  \* 运行结果  \*　type: float64  \* value: 3.4  \* type: float64  \* kind: float64  \* value: 3.4  \* 3.4  \* value is 3.40e+00  \* 3.4  \*/ |

### 通过反射修改(设置)值)

* 反射中有些内容是需要用地址去改变它的状态的
* 示例

|  |
| --- |
| func main() {     var x float64 = 3.4     fmt.Printf("&x = %p\n", &x)     v := reflect.ValueOf(x)     //v.SetFloat(3.1415) // Error: will panic: reflect.Value.SetFloat using unaddressable value     fmt.Println("settability of v:", v.CanSet())     v = reflect.ValueOf(&x)     fmt.Println(v)     fmt.Println("type of v:", v.Type())     v = v.Elem()     fmt.Println("The Elem of is:", v)     fmt.Println("settability of v:", v.CanSet())     v.SetFloat(3.1415)     fmt.Println(v.Interface())     fmt.Println(v)  } /\*\*  \* 运行结果  \* &x = 0xc0420381d0  \* settability of v: false  \* 0xc0420381d0  \* type of v: \*float64  \* The Elem of is: 3.4  \* settability of v: true  \* 3.1415  \* 3.1415  \*/ |

### 反射结构

* 有些时候需要反射一个结构类型。NumField() 方法返回结构内的字段数量；通过一个 for 循环用索引取得每个字段的值 Field(i)。
* 我们同样能够调用签名在结构上的方法，例如，使用索引 n 来调用：Method(n).Call(nil)。

|  |
| --- |
| type NotknownType struct {     s1, s2, s3 string } func (n NotknownType) String() string {     return n.s1 + " - " + n.s2 + " - " + n.s3 } var secret interface{} = NotknownType{"Ada", "Go", "Oberon"} func main() {     value:=reflect.ValueOf(secret)     typ:=reflect.TypeOf(secret)     fmt.Println(typ)     knd:=value.Kind()     fmt.Println(knd)     for i:=0;i<value.NumField();i++{         fmt.Printf("Field %d:%v\n",i,value.Field(i))     }     results :=value.Method(0).Call(nil)     fmt.Println(results) } /\*\*  \* 运行结果  \*　main.NotknownType  \* struct  \* Field 0:Ada  \* Field 1:Go  \* Field 2:Oberon  \* [Ada - Go - Oberon]  \*/ |

* 结构中只有被导出字段(首字母大写)才是可设置的

|  |
| --- |
| type T struct {     A int     B string } func main() {     t := T{23, "skidoo"}     s := reflect.ValueOf(&t).Elem()     typeOfT := s.Type()     for i := 0; i < s.NumField(); i++ {         f := s.Field(i)         fmt.Printf("%d: %s %s=%v\n", i, typeOfT.Field(i).Name, f.Type(), f.Interface())     }     s.Field(0).SetInt(77)     s.Field(1).SetString("Sunset Strip")     fmt.Println("t is now", t) } /\*\*  \* 运行结果  \* 0: A int=23  \* 1: B string=skidoo  \* t is now {77 Sunset Strip}  \*/ |

### Printf和反射

* Printf 中的 ... 参数为空接口类型。Printf 使用反射包来解析这个参数列表。所以，Printf 能够知道它每个参数的类型。因此格式化字符串中只有%d而没有 %u 和 %ld，因为它知道这个参数是 unsigned 还是 long。这也是为什么 Print 和 Println 在没有格式字符串的情况下还能如此漂亮地输出。

|  |
| --- |
| type Stringer interface {     String() string } type Celsius float64  func (c Celsius) String() string {     return strconv.FormatFloat(float64(c), 'f', 1, 64) + "°C" } type Day int var dayName = []string{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"} func (day Day) String() string {     return dayName[day] } func main() {     print(Day(1), "was", Celsius(18.36)) } func print(args ...interface{}) {     for i, arg := range args {         if i > 0 { os.Stdout.WriteString(" ") }         switch a := arg.(type) {         case Stringer:             os.Stdout.WriteString(a.String())         case int:             os.Stdout.WriteString(strconv.Itoa(a))         case string:             os.Stdout.WriteString(a)         default:             os.Stdout.WriteString("???")         }     } } /\*\*  \* 运行结果  \* Tuesday was 18.4°C  \*/ |

# 官方标准库

## 详情参文档:

<http://studygolang.com/pkgdoc>

## strings

### strings.Map()

func Map(mapping func(rune) rune, s string) string

将s的每一个unicode码值r都替换为mapping(r)，返回这些新码值组成的字符串拷贝。如果mapping返回一个负值，将会丢弃该码值而不会被替换。（返回值中对应位置将没有码值）

### strings.IndexFunc()

func IndexFunc(s string, f func(rune) bool) int

s中第一个满足函数f的位置i（该处的utf-8码值r满足f(r)==true），不存在则返回-1。

## unicode/utf8

### utf8.RunCountInString

|  |  |
| --- | --- |
|  | func main() {     str:="abc中d"     fmt.Println(len(str))     fmt.Println(len([]int32(str)))     // 以下方法效率更高     fmt.Println(utf8.RuneCountInString(str)) } /\* 运行结果 7 5 5 \*/ |

## bytes

类型 []byte 的切片十分常见，Go 语言有一个 bytes 包专门用来解决这种类型的操作方法。

bytes 包和字符串包十分类似（参见第 4.7 节）。而且它还包含一个十分有用的类型 Buffer:

import "bytes"

type Buffer struct {

...

}

这是一个长度可变的 bytes 的 buffer，提供 Read 和 Write 方法，因为读写长度未知的 bytes 最好使用 buffer。

Buffer 可以这样定义：var buffer bytes.Buffer。

或者使用 new 获得一个指针：var r \*bytes.Buffer = new(bytes.Buffer)。

或者通过函数：func NewBuffer(buf []byte) \*Buffer，创建一个 Buffer 对象并且用 buf 初始化好；NewBuffer 最好用在从 buf 读取的时候使用。

****通过 buffer 串联字符串****

类似于 Java 的 StringBuilder 类。

在下面的代码段中，我们创建一个 buffer，通过 buffer.WriteString(s) 方法将字符串 s 追加到后面，最后再通过buffer.String() 方法转换为 string：

var buffer bytes.Bufferfor {

if s, ok := getNextString(); ok { //method getNextString() not shown here

buffer.WriteString(s)

} else {

break

}

}

fmt.Print(buffer.String(), "\n")

这种实现方式比使用 += 要更节省内存和 CPU，尤其是要串联的字符串数目特别多的时候。

## sort

### sort.Ints(a []int)

### sort.IntsAreSorted(a []int)

### sort.Float64s(a []float64)

### sort.SearchInts(a []int, n int)

* 必须先排序再搜索

## sync

### sync.Mutex

* sync.Mutex 是一个互斥锁，它的作用是守护在临界区入口来确保同一时间只能有一个线程进入临界区
* 在 sync 包中还有一个 RWMutex 锁：他能通过 RLock() 来允许同一时间多个线程对变量进行读操作，但是只能一个线程进行写操作。如果使用 Lock() 将和普通的 Mutex 作用相同。包中还有一个方便的 Once 类型变量的方法 once.Do(call)，这个方法确保被调用函数只能被调用一次。