Go 是Golang 缩写

特色: 简洁、快速、安全

并行、有趣、开源

内存管理、v数组安全、编译迅速

用途：go语言被设计成一门应用于搭载Web服务器，存储集群或类似用途的巨型中央服务器的系统编程于洋。对于高性能分布式系统领域而言，GO语言五一比大多数其他语言有着更高效的开发效率，他提供了海量的支持，这对于游戏服务端的开发而言是再好不过了。

运行 go run test.go

语言包结构：包声明 、引入包、 函数、 变量、 语句&表达式、 注释

数据类型：

1、布尔型 2、数字类 （int 和float ）3、字符串类型

4、派生类型（a、指针类型 b、数组类型、 c、结构化类型 d、联合体类型 e 函数类型 f、切片类型 g、接口类型 h、map类型 i、channel类型 ）

变量的声明： Go语言变量由字母、数字、下划线组成，其中首个字母不能为数字。

**常见的**

**Var a int a=10 var b=10 c :=10**

**在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明**

当一个变量被var声明后，系统自动赋予它该类型的0只值。

多变量可以在同一行进行复制，如： var a,b ,c = 5,7,”abc”

简式声明： a,b,c := 5,7,”abc”

注意：简式声明一般用在func 内，要注意的是，全局变量和简式变量尽量不要同名，否则很容易残生偶然的变量隐藏。 交换变量的值 a,b =b,a

空白表示符：\_ 也被用与抛弃值，如值5在： \_,b = 5,7中被抛弃。

由于Go语言有个强制规定，在函数内一定要使用声明的变量，但未使用的全局变量是没问题的。为了避免有未使用的变量，代码将编译失败，我们可以将该未使用的变量改为 \_。如：

\_,\_ =a,b 抛弃变量。

引入的包不使用，编译也会错误，避免编译失败，使用下划线标记包不使用

如： import（

\_ “fmt” //抛弃值

“log”

“time”

）

一个函数返回多个值也可以用并行赋值来接收。 Var , err = Fun1(var1)

Go语言当中，指针属于引用类型，其他的引用类型还包括slices ,maps ,channel

注意，Go中的数组是数值，因此当你向函数中传递数组时，函数会得到原始数组数据的一份复制。如果你打算更新数组的数据，可以考虑使用数组指针类型。

指针 Nil 零值

Nil 标志符用表示interface、函数、maps、slices和channels 的零值。

常量和iota

常见 显示 const b int =1 隐式 const c = 2

Const pi = 3.14 const(

a =1

b=2

)

Iota 用于枚举

Const (

A=iota

B=iota

C=iota

)

第一个iota等于0，每当在新的一行被使用时，它的值都会自动加1；所以a=0;b=1,c=2

简写

const(

a =iota

b

c

)

注意 const (

A=iota

B=8

c

)

A、b、c、分别为0，,8,8新的常量b声明后，iota不在向下赋值，后面常量如果没有赋值，则继承上一个常量值。

可以简单理解为在一个const块中，每换一行定义个常量，iota都会自动加1.

作用域

局部变量 全局变量

简式变量

使用 := 定义的变量，如果新变量Ga与那个同名已定义变量 (这里就是那个全局变量Ga)不在一个作用域中时，那么Go 语言会新定义这个变量Ga，遮盖住全局变量Ga。刚开始很容易在此犯错而茫然，解决方法是局部变量尽量不同名。

**有花括号"{ }"一般都存在作用域的划分；  
:= 简式声明会屏蔽所有上层代码块中的变量（常量），建议使用规则来规范，如对常量使用全部大写，而变量尽量小写；  
在if等语句中存在隐式代码块，需要注意；  
闭包函数可以理解为一个代码块，并且他可使用包含它的函数内的变量；**

**命名规范以及语法惯例**

干净、可读的代码和简洁性是 Go 追求的主要目标。通过 Gofmt 来强制实现统一的代码风格。Go 语言中对象的命名也应该是简洁且有意义的。像 Java 和 Python 中那样使用混合着大小写和下划线的冗长的名称会严重降低代码的可读性。名称不需要指出自己所属的包，因为在调用的时候会使用包名作为限定符。返回某个对象的函数或方法的名称一般都是使用名词，没有 Get... 之类的字符，如果是用于修改某个对象，则使用 SetName。有必须要的话可以使用大小写混合的方式，如 MixedCaps 或 mixedCaps，而不是使用下划线来分割多个名称。

**函数里的代码（函数体）使用大括号 {} 括起来。**

**左大括号 { 必须与方法的声明放在同一行，这是编译器的强制规定，否则你在使用 Gofmt 时就会出现错误提示**

**对于大括号 {} 的使用规则在任何时候都是相同的（如：if 语句等）**

**单字之间不以空格断开或连接号（-）、底线（\_）连结，第一个单字首字母采用大写字母；后续单字的首字母亦用大写字母，例如：FirstName、LastName。每一个单字的首字母都采用大写字母的命名格式，被称为“Pascal命名法”，  
包的概念**

**package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。package main包下可以有多个文件，但所有文件中只能有一个main()方法，main()方法代表程序入口。所有的包名都应该使用小写字母**

**包的导入**

**import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包**

**导入包的路径的几种情况：**

**import "./module" （第一种方式相对路径）**

import "LearnGo/init" 第二种方式绝对路径

import (

"fmt"

"net/http"

)

导入多个包的常见的方式是：

点操作

Import (.“fmt”) 你可以省略前缀的包名 Print(“hello world”)

别名操作 import(f “fmt”) f.Print(“hello world”)

\_操作

\_操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的init函数

**从 GitHub 安装包**

Go get -u github.com/ffhelicopter/tmm

**导入外部安装包**

你可以通过如下命令安装： Go install github.com/gocolly/colly 将一个名为 github.com/gocolly/colly 安装在$GoPATH/pkg/ 目录下。

Go install/build都是用来编译包和其依赖的包

区别： Go build只对main包有效，在当前目录编译生成一个可执行的二进制文件（依赖包生成的静态库文件放在$GoPATH/pkg）

如果为main包，运行Go buil则会在$GoPATH/bin 生成一个可执行的二进制文件。

**项目结构**

Go的工程项目管理非常简单，使用目录结构和package名来确定工程结构和构建顺序。

环境变量GOPATH在项目管理中非常重要，想要构建一个项目，必须确保项目目录在GOPATH中。而GOPATH可以有多个项目用";"分隔。

Go 项目目录下一般有三个子目录：

* src存放源代码
* pkg编译后生成的文件
* bin编译后生成的可执行文件

## Go程序的编译

如果想要构建一个程序，则包和包内的文件都必须以正确的顺序进行编译。包的依赖关系决定了其构建顺序。

属于同一个包的源文件必须全部被一起编译，一个包即是编译时的一个单元，因此根据惯例，每个目录都只包含一个包。

如果 A.go 依赖 B.go，而 B.go 又依赖 C.go：

编译 C.go，B.go，然后是 A.go。

为了编译 A.go，编译器读取的是 B.go 而不是 C.go。

**内置运算符**

* 算术运算符 + - \* / % ++ -- B / A 结果还是整数 8/3=2
* 关系运算符 == != < > <= >=
* 逻辑运算符 && || !
* 位运算符
  + & 按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。
  + | 按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或
  + ^ 按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1
  + << 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0
  + >> 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数
* 赋值运算符 = += -= \*= /= %= <<= (左移后赋值) >>=(右移后赋值) &= (按位与后赋值) ^= 按位异或后赋值 |= 按位或后赋值
* 其他运算符 & 返回变量存储地址，将给出变量的实际地址

\*指针变量，\*a是一个指针变量

**几个特殊运算符**

位清除 &^ 1、如果右侧是0，则左侧数保持不变 2、如果右侧是1，则左侧数一定清零

对于整数运算而言，结果依旧为整数，例如：9 / 4 -> 2。

取余运算符只能作用于整数：9 % 4 -> 1。

浮点数除以 0.0 会返回一个无穷尽的结果，使用 +Inf 表示。

你可以将语句 b = b + a 简写为 b+=a，同样的写法也可用于 -=、\*=、/=、%=。

对于整数和浮点数，你可以使用一元运算符 ++（递增）和 --（递减），但只能用于后缀：

i++ -> i += 1 -> i = i + 1  
  
i-- -> i -= 1 -> i = i - 1

同时，带有 ++ 和 -- 的只能作为语句，而非表达式，因此 n = i++ 这种写法是无效的。

函数 rand.Float32 和 rand.Float64 返回介于 [0.0，1.0) 之间的伪随机数，其中包括 0.0 但不包括 1.0。函数 rand.Intn 返回介于 [0，n) 之间的伪随机数。

**字符串介绍**

Go语言中可以使用反引号和双引号来定义字符串，反引号表示原生字符串，既不进行转义。

Go 语言中的string类型是一种值类型，存储的字符串是不可变的，如果要修改string内容需要将string转换为[]byte或[]rune，并且修改后的string内容是重新分配的。

那么byte和rune的区别是什么(下面写法是type别名):

Byte =uint8 rune = int 32

package main

import (

"fmt"

"unicode/utf8"

)

func main() {

s := "其实就是rune"

fmt.Println(len(s)) // "16"

fmt.Println(utf8.RuneCountInString(s)) // "8"

}

**字符串拼接**

**直接使用运算符**str := "Beginning of the string " + "second part of the string"

由于编译器行尾自动补全分号的缘故，加号 + 必须放在第一行。

**拼接的简写形式 +=** 也可以用于字符串：

s := "hel" + "lo, "

s += "world!"

里面的字符串都是不可变的，每次运算都会产生一个新的字符串，所以会产生很多临时的无用的字符串，不仅没有用，还会给 gc 带来额外的负担，所以性能比较差

fmt.Sprintf()

fmt.Sprintf(“%d:%s”,2018,“年”)  
内部使用 []byte 实现，不像直接运算符这种会产生很多临时的字符串，但是内部的逻辑比较复杂，有很多额外的判断，还用到了 interface，所以性能一般。

strings.Join()

strings.Join([]string{“hello”, ”world”}，”,”)

Join会先根据字符串数组的内容，计算出一个拼接之后的长度，然后申请对应大小的内存，一个一个字符串填入，在已有一个数组的情况下，这种效率会很高，但是本来没有，去构造这个数据的代价也不小。

bytes.Buffer

var buffer bytes,Buffer

buffer,WriteString(“hello”)

buffer.WriteString(“,”)

buffer.WriteString(“world !”)

fmt.Print(buffer.String())

这个比较理想，可以当成可变字符使用，对内存的增长也有优化，如果能预估字符串的长度，还可以用 buffer.Grow() 接口来设置 capacity。

Strings.Builder

var b1 strings.Builder //取别名

b1.WriteString(“ABC”)

b1.WriteString(“DEF”)

fmt.Println(b1.string())strings.Builder 内部通过 slice 来保存和管理内容。slice 内部则是通过一个指针指向实际保存内容的数组。strings.Builder 同样也提供了 Grow() 来支持预定义容量。当我们可以预定义我们需要使用的容量时，strings.Builder 就能避免扩容而创建新的 slice 了。strings.Builder是非线程安全，性能上和 bytes.Buffer 相差无几

**有关string处理**

标准库中有四个包对字符串处理尤为重要：bytes、strings、strconv 、 Unicode 包。

strings 包提供了很多操作字符串的简单函数，通常一般的字符串操作需求都可以在这个包中找到。下面简单举几个例子：

判断是否以某字符串打头/结尾  
strings.HasPrefix(s, prefix string) bool  
strings.HasSuffix(s, suffix string) bool

字符串分割

Strings.split(s,sep string) []string

返回字符串索引

Strings.Index(s,substr string) int

Strings.LastIndex 最后一个匹配索引

字符串连接

Strings.Join(a []string,sep string) string

字符串替换

Strings.Replace(s,old,new string ,n int) string

字符串转化大小写

Strings.ToUpper(s string) string

Strings.ToLower(s string) string

统计某个字符串在字符串中出现的次数

Strings.Count(s,substr string) int

判断字符串的包含关系

Strings.Contain(s,substr string) bool

**数组(Array)**

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。

注意事项：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interface{}作为类型。当使用值时我们必须先做一个类型判断。

一维数组声明以及初始化常见方式如下：

Var arrAge = [5]int{18,20,12,14,44}

Var arrName =[5]string{3:”Chirs”,4:”Ron”}//指定索引位置初始化。

Var arrCount = [4]int{500,2：100} //同上

Var arrLazy =[…]int{5,6,7,8,9}//数组长度初始化根据袁术多少确定

Var arrPack =[…]int{10,5:100}

Var arrRoom =[20]int

Var arrBed = new([20]int) //数组指针

[...][5]int{ {10, 20}, {30, 40} } //len() 长度根据实际初始化时数据的长度来定，这里为2

[3][5]int //len() 长度为3

支持多维的定义

在定义多维数组时，仅第一维允许使用“…”，而内置函数len和cap也都返回第一维度长度

数组长度最大为 2Gb

另外，如数组元素类型支持”==，!=”操作符，那么数组也支持此操作，但如果数组类型不一样则不支持（需要长度和数据类型一致，否则编译不通过）

**切片(slice)**

**切片（slice）**是对数组一个连续片段的引用（该数组我们称之为相关数组，通常是匿名的），所以切片是一个引用类型（和数组不一样）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个相关数组的动态窗口（这里有关动态窗口的含义，可参考数据库窗口函数的解释）。

切片是可索引的，并且可以由 len() 函数获取长度。

var slice1 []type = make([]type, len)

var x = []int{2, 3, 5, 7, 11} //类似数组的初始化

v := make([]int, 10, 50) 这样分配一个有 50 个 int 值的数组，并且创建了一个长度为 10，容量为 50 的 切片 v，该 切片 指向数组的前 10 个元素。

**break**

一个 break 的作用范围为该语句出现后的最内部的结构，它可以被用于任何形式的 for 循环（计数器、条件判断等）。  
但在 switch 或 select 语句中，break 语句的作用结果是跳过整个代码块，执行后续的代码。

**continue**

关键字 continue 忽略剩余的循环体而直接进入下一次循环的过程，但不是无条件执行下一次循环，执行之前依旧需要满足循环的判断条件。  
关键字 continue 只能被用于 for 循环中

字典

Map是引用类型 列如：

var map1 map[keytype]valuetype

var map1 map[string]int

在声明的时候不需要知道map长度，map是可以动态增长的。

key 可以是任意可以用 == 或者 != 操作符比较的类型，比如 string、int、float。所以数组、切片和结构体不能作为 key。含有数组切片的结构体不能作为 key，只包含内建类型的 struct 是可以作为 key 的

value 可以是任意类型的；

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。**

map2 := make(map[string]float32, 100)

当 map 增长到容量上限的时候，如果再增加新的 key-value 对，map 的大小会自动加 1。所以出于性能的考虑，对于大的 map 或者会快速扩张的 map，即使只是大概知道容量，也最好先标明。

一般判断是否某个key存在，不使用值判断，而使用下面的方式：

if \_, ok := x["two"]; !ok {

fmt.Println("no entry")

}

直接 delete(map1, key1) 就可以。如果 key1 不存在，该操作不会产生错误。 Delete(map4, "a") 删除

map 默认是无序的，不管是按照 key 还是按照 value 默认都不排序。  
如果你想为 map 排序，需要将 key（或者 value）拷贝到一个切片，再对切片排序（使用 sort 包）。

**函数**

**函数基本组成：关键字func 、函数名、参数列表、返回值、函数体和返回语句。除了main()、init()函数外，其它所有类型的函数都可以有参数与返回值。**

**函数签名由函数参数、返回值以及它们的类型组成，被统称为函数签名。**

**如果两个函数的参数列表和返回值列表的变量类型能一一对应，那么这两个函数就有相同的签名，下面testa与testb具有相同的函数签名。**

**func testa (a, b int, z float32) bool**

**func testb (a, b int, z float32) (bool)**

**无默认参数。Func testc(str string, shu …int) 这…int 写法表示shu是一个可变参数。**

**在函数体中，参数是局部变量，被初始化为调用者传入的值。函数的参数和具名返回值是函数最外层的局部变量，它们的作用域就是整个函数。如果函数的签名声明了返回值，则函数体的语句列表必须以终止语句结束。**

**函数调用：默认是值传递，引用传递func(&a int) 切片、字典、接口、通道等这样的引用类型都是默认使用引用传递。**

**Go 语言拥有一些内置函数，内置函数是预先声明的**

**内置函数：**

**Close() 用于通道，对于通道c，内置函数close(c)将不再在通道c上发送值。 如果c是仅接收通道，则会出错。 发送或关闭已关闭的通道会导致运行时错误。 关闭nil通道也会导致运行时错误。**

**new 和 make 均是用于分配内存：new用于值类型的内存分配，并且置为零值。make只用于slice、map以及channel这三种引用数据类型的内存分配和初始化。new(T) 分配类型 T 的零值并返回其地址，也就是指向类型 T 的指针。make(T) 它返回类型T的值（不是\* T）。**

**调用 T的类型 结果**

**make(T, n) slice T为切片类型，长度和容量都为n**

**make(T, n, m) slice T为切片类型，长度为n，容量为m （n<=m ，否则错误）**

**make(T) map T为字典类型**

**make(T, n) map T为字典类型，初始化n个元素的空间**

**make(T) channel T为通道类型，无缓冲区**

**make(T, n) channel T为通道类型，缓冲区长度为n**

**type S struct { a int; b float64 }**

**new(S) new(S)为S类型的变量分配内存，并初始化（a = 0，b = 0.0），返回包含该位置地址的类型\* S的值。**

**Len(s) string的长度（按照字节计算），数组长度 ，切片长度 ，字典长度 ，通道缓冲区中排队的元素数**

**cap(s) 数组长度 ，切片容量 ，通道缓冲区容量**

**copy内置函数常常将切片元素从源src复制到目标dst，并返回复制的元素数**

append 用于附加连接切片 copy 用于复制切片 delete 从字段删除元素

**complex 从浮点实部和虚部构造复数值**

**real 提取复数值的实部、**

**image 提取复数值的虚部**

**panic 从浮点数实部和虚部构造复数值**

**recover 提取复数值的实部**

**panic和recover两个内置函数，协助报告和处理运行时恐慌和程序定义的错误**

**匿名函数同样也被称之为闭包。**

**func(x, y int) int { return x + y } 这样的函数不能够独立存在**

**fplus := func(x, y int) int { return x + y }**

**type :关键字**

**基础类型：bool byte comple64 complex18 error float32 float64 int int8 int16 int32 int64 rune string uint uint8 uint16 unt6 uintptr**

**使用type关键字可以定义你自己的类型，你可能想要定义一个结构体，但是也可以给一个已经存在的类型的新名字，然后呢就可以在你的代码中使用新的名字（用于简化名称或解决冲突名称）称为自定义类型，如：**

**Type IZ int**

**然后我们可以使用下面的方式声明变量：**

**Var a IZ = 5**

**type (**

**IZ int**

**FZ float64**

**STR string**

**)**

**在 type TZ int 中，TZ 就是 int 类型的新名称（用于表示程序中的时区），称为自定义类型，然后就可以使用 TZ 来操作 int 类型的数据。使用这种方法定义之后的类型可以拥有更多的特性，且在类型转换时必须显式转换。**

**type TZ = int**

**结构体（struct）**

**Go 通过结构体的形式支持用户自定义类型，或者叫定制类型。**

**组成结构体类型的那些数据称为字段（fields）。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的。**

**type identifier struct {**

**field1 type1**

**field2 type2**

**...**

**}**

**结构体里的字段都有 名字，像 field1、field2 等，如果字段在代码中从来也不会被用到，那么可以命名它为 \_。**

**列子：**

**type interval struct{**

**start int**

**end int**

**}**

**初始化方式：**

**Intr :=interval{ 0 , 3 } intr :=interval{end:5,start:1}**

**intr := interval{end:5}**

**如果 File 是一个结构体类型，那么表达式 new(File) 和 &File{} 是等价的。**

**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（这决定了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。）。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。**

**type Base struct {**

**basename string**

**}**

**type Derive struct { // 含内嵌结构体**

**Base // 匿名**

**int**

**}**

**内嵌与聚合：   
外部类型只包含了内部类型的类型名， 而没有field 名， 则是内嵌。外部类型包含了内部类型的类型名，还有filed名，则是聚合。聚合的在JAVA和C++都是常见的方式。而内嵌则是Go 的特有方式。**

**type Base struct {**

**basename string**

**}**

**type Derive struct { // 内嵌**

**Base }**

**type Derive struct { // 内嵌， 这种内嵌与上面内嵌有差异**

**\*Base**

**}**

**type Derive struct{ // 聚合**

**base Base**

**}**

**内嵌的方式：  
主要是通过结构体和接口的组合，有四种。**

**接口中内嵌接口: 可以 接口中内嵌结构体 不可以 结构体中内嵌结构体 可以 结构体重内嵌接口， 可以。**

**type Writer interface{**

**Write()**

**}**

**type Reader interface{**

**Read()**

**}**

**type WRer interface{**

**Reader**

**Writer**

**}**

**在接口中内嵌struct :**

**存在语法错误，并不具有实际的含义， 编译报错:**

* **结构体（struct）中内嵌 结构体（struct）**

**1，初始化，内嵌结构体要进行赋值。**

**2，外层结构自动获得内嵌结构体所有定义的field和实现的方法（method）。**

**3，同上述结构体中内嵌接口类似，同样外层结构体可以定义同名方法，这样覆盖内层结构体的定义的方法。 同样也可以定义同名变量，覆盖内层结构体的变量。**

**4，同样可以内层结构体引用，内层结构体重已经定义的方法和变量。**

**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**

**接口是什么**

**Go 语言里有非常灵活的 接口 概念，通过它可以实现很多面向对象的特性。  
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。**

**注意：  
类型不需要显式地声明它实现了某个接口，接口被隐式地实现，隐式接口解藕了实现接口的包和定义接口的包：互不依赖。**

**注意：接口中的方法必须要全部实现，才能实现接口。**

**接口与动态类型：**

**Go 没有类：数据（结构体或更一般的类型）和方法是一种松耦合的正交关系。**

**Go 中的接口跟 Java/C# 类似：都是必须提供一个指定方法集的实现。但是更加灵活通用：任何提供了接口方法实现代码的类型都隐式地实现了该接口，而不用显式地声明**

**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。**

**接口的提取**

**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**

**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。**

**方法：**

**Go 方法是作用在接收器（receiver）上的一个函数，接收器是某种类型的变量。因此方法是一种特殊类型的函数。**

**type A struct {**

**Face int**

**}**

**func (a A) f() {**

**fmt.Println("hi ", a.Face)**

**}**

**上面代码中，我们定义结构体A,注意f()就是A的方法，（a A）表示接受器，a是receiver 的实例，f()是它的方法名，那么方法调用遵循传统的object.name 选择器符号：a.f().**

**接收器类型可以是（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。**

**接收器不能是一个接口类型，因为接口是一个抽象定义，但是方法却是具体实现；如果这样做会引发一个编译错误：invalid receiver type…。**

**接收器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**

**关于接收器的命名**

**社区约定的接收器命名是类型的一个或两个字母的缩写(像 c 或者 cl 对于 Client)。不要使用泛指的名字像是 me，this 或者 self，也不要使用过度描述的名字，最后，如果你在一个地方使用了 c，那么就不要在别的地方使用 cl。**