1. 入门

2019年5月21日 14:57

go是编译型语言，go的工具链将程序的源文件转变成机器相关的原生二进制指令。
go run 将一个或多个.go后缀的源文件进行编译、链接，然后运行生成的可执行文件。
go build
go原生地支持unicode，它可以处理所有国家的语言。

go代码是使用包来组织的，类似于其他语言中的库和模块。

- 一个包有一个或多个 .go 源文件组成，放在一个文件夹中。
- 每个源文件的开始都使用package声明，eg: package main，指明了这个文件属于那个包。
  后面跟着它导入的其他包的列表，然后是存储在文件中的程序声明。
  包名为main表示用来定义一个独立的可执行程序，而不是库。
- go标准库中有100多个包用来完成输入、输出、排序、文本处理等常规操作。
  Eg: fmt包中的函数用来格式化输出和扫描输入。

go对代码的格式化要求非常严格。

- { 必须和关键字 func 在同一行，不能独自成行。
- gofmt工具将代码以标准格式重写。
  许多文本编辑器都可以配置为每次保存文件时自动运行gofmt。

1.2 命令行参数

2019年5月21日 16:51

命令行参数以os包中Args名字的变量供程序访问。变量os.Args是一个字符串slice。
和大部分编程语言一样，go中所有的索引使用半开区间，即包含第一个索引，不包含最后一个索引。索引大小从 0 到 len(s)-1。

:= 短变量声明符，用于声明一个或多个变量，并根据初始化的值给予合适的类型。

```
for _, arg := range os.Args[1:]
//每一次迭代，range产生一对值：索引和这个索引处元素的值。
```

2. 程序结构

2019年5月22日 11:35

2.1 名称

2019年5月22日 11:35

go中函数、变量、常量、类型、语句标签和包的名称：
名称的开头是一个字母或下划线，后面可以跟任意数量的字符、数字和下划线。区分大小写。

如果一个实体在函数中声明，它只在函数局部有效。如果声明在函数外，它将对包里面的所有源文件可见。
实体的第一个字母大小写决定其可见性是否跨包。

- 如果名称以大写字母开头，它是导出的，意味着它对包外是可见和可访问的。Eg: fmt包中的Println。

包名本身总是由小写字母组成。

名称采用驼峰式风格。

2.2 声明

2019年5月22日 11:43

声明给一个程序实体命名，并且设定其部分或全部属性。
有4个主要的声明：变量var，常量const，类型type，函数func

2.3 变量

2019年5月29日　　10:57

var声明创建一个具体类型的变量，然后给它附加一个名字，设置它的初始值。
var name type = expression
type和expression可以省略一个，但不能都省略。

- 如果类型省略，它的类型将由初始化表达式决定；
- 如果表达式省略，其初始值对应于类型的零值。go里面不存在未初始化的变量。
    - 数字 0
    - 布尔 false
    - 字符串 ""
    - slice，指针，map，通道，函数 nil

忽略类型允许声明多个不同类型的变量：
var b, f, s = true, 2.3, "four"

2.3.1 短变量声明

用于声明和初始化局部变量。形如：name := expression，name的类型由expression的类型决定。

- 局部变量的声明和初始化中主要使用短声明。
- var声明通常是为那些跟初始化表达式类型不一致的局部变量保留的。或者用于后面才对变量赋值以及变量初始值不重要的情况。

:= 表示声明，= 表示赋值。

短变量的声明不需要声明所有在左边的变量。如果一些变量在同一个词法块中声明，那么对于这些变量，短声明的行为等同于赋值。

```
in, err := os.Open(infile)       //声明了in和err
out, err := os.Create(outfile)  //声明了out，向已声明的err变量赋值
```

2.3.2 指针

不是所有的值都有地址（字面量没有地址），但是所有的变量都有。

每一个聚合类型变量的组成（结构体的成员或数组中的元素）都是变量，所以也有地址。

函数返回局部变量的地址是非常安全的。//不同于C

每次使用变量的地址或复制一个指针，我们就创建了新的别名或者方法来标记同一个变量。不仅指针产生别名，当复制其他引用类型（slice,map,通道,甚至包括这里引用类型的结构体,数组和接口）的值时，也会产生别名。

指针对于flag包是很关键的。

2.4 赋值

2019年8月1日 星期四　　下午8:40

每一个算术和二进制位操作符有一个对应的赋值操作符，避免了在表达式中重复变量本身。
Eg: *=, +=

2.4.1 多重赋值

多重赋值允许几个变量一次性被赋值。

交换两个变量的值：

```
x, y = y, x
```

2.4.2 可赋值性

赋值语句是显式形式的赋值，但是程序中还有很多赋值是隐式的。
不管是隐式还是显式赋值，如果左边和右边的类型相同，就是合法的。

2.5 类型声明

2019年8月1日 星期四　　下午8:40

type声明定义了一个新的命名类型，它和某个已有类型使用同样的底层类型。
类型声明通常出现在包级别，在整个包中可见，如果名字是导出的（开头使用大写字母），其他包中也可以访问。

对于每个类型T，都有一个对应的类型转换操作T(x)将值x转换为类型T。

- 如果两个类型具有相同的底层类型或二者都是指向相同底层类型变量的未命名指针类型，则二者是可以转换的。

通过 == 和 < 之类的比较操作符，命名类型的值可以与其相同类型的值或者底层类型相同的未命名类型的值相比较。
但是不同命名类型的值不能直接比较。

很多类型都声明String方法，在变量通过fmt包作为字符串输出时，它可以控制类型值的显式方式。

2.6 包和文件

2019年8月1日 星期四　　下午8:40

2.6.2 包初始化

包的初始化是从初始化包级别的变量开始，这些变量按照声明顺序初始化，在依赖已解析完毕的情况下，根据依赖的顺序进行。
如果包由多个go文件组成，初始化按照编译器收到文件的顺序进行：go工具会在调用编译器前将go文件进行排序。

任何文件可以包含任意数量的init函数：

func init() {}

- init函数不能被调用和被引用
- 在每一个文件中，当程序启动时，init函数按照它们声明的顺序自动执行。

包的初始化按照在程序中导入的顺序来进行，依赖顺序优先。
初始化过程是自下而上的，main包最后初始化。程序在main函数开始执行之前，所有的包已经初始化完毕。

2.7 作用域

2019年6月13日 11:14

不要将作用域和生命周期混淆。

- 声明的作用域是声明在程序文本中出现的区域，它是一个编译时属性。
- 变量的生命周期是变量在程序执行期间能被程序的其他部分所引用的起止时间，它是一个运行时属性。

语法块 block 是由{ }围起来的一个语句序列，如循环体，if语句或函数体。语法块内部声明的变量对块外部不可见。
我们可以把块的概念推广到其他没有显式包含在大括号中的声明代码，将其统称为词法块。

- 包含了全部源代码的词法块，叫做全局块。
- 每一个包，每一个文件，每一个for、if和switch语句，以及switch和select语句中的每一个条件，都是写在词法块中的。
- 显式写在大括号语法里的代码块也算是一个词法块。

声明的词法块决定声明的作用域大小：

- Int, len, true等内置类型、函数或常量在全局块中声明并且对整个程序可见。
- 包级别的声明(在任何函数外)，可以被同一包里的任何文件使用。
- 导入的包是文件级别的，它们可以在同一个文件内使用，但是不能被同一个包的其他文件使用。
- 局部的声明，仅仅可以在同一个函数中或仅仅是函数的一部分中使用。
- 控制流标签(break, continue, goto等语句使用的标签)的作用域是整个外层函数。

一个程序可以包含多个同名的声明，前提是它们在不同词法块中。
当编译器遇到一个名字时，将从最内层的封闭词法块到全局块寻找其声明。如果内层和外层块都存在这个声明，内存的将先被找到。

有一些词法块是隐式的。for循环创建了两个词法块：

- 一个是循环体本身的显式块
- 一个隐式块。隐式块中声明的变量的作用域包括条件、后置语句、以及for语句体本身。

和for循环一样，除了本身的主体块之外，if和swith语法也会创建隐式的词法块。

在包级别，声明的顺序和它们的作用域没有关系。所以一个声明可以引用它自己或者跟在它后面的其他声明。

3. 基本数据

2019年5月29日 14:52

3.1 整数

2019年5月29日 15:03

int8	int16	int32	int64
uint8	uint16	uint32	uint64

int和uint这两种类型位数相等，都是32或64位，但不能认为它们一定就是32位，或一定就是64位；
即使在同样的硬件平台上，不同的编译器可能选用不同的大小。

rune和int32是同义词；byte和uint8是同义词。
uintptr表示一种无符号整数，其大小并不明确，但足以完整存放指针。

int,uint,uintptr都有别于其他大小明确的类型。
Eg:尽管int天然大小就是32位，但int值若要当作int32使用，必须显式转换。反之亦然。

有符号整数以补码表示，保留最高位作为符号位。

3.2 浮点数

2019年5月29日 15:28

go具有两种大小的浮点数float32和float64。

3.4 布尔值

2019年5月29日 15:28

bool值只有两种可能：true或false。

- If和for语句里的条件就是布尔值，比较操作符(如==和<)也能得出布尔值结果。

&&比||的优先级更高。

布尔值无法隐式转换位数值(0或1)，反之也不行。

3.5 字符串

2019年5月29日 15:38

字符串是不可变的字节序列。
尽管可以将新值赋予字符串变量，但是字符串值无法改变，字符串值本身所包含的字节序列永不可变。

```
S := "111"
S[0] = '2'  //error
```

内置的len函数返回字符串的字节数。

3.5.1 字符串字面量

GO的源文件总是按UTF-8编码。

带双引号的字符串常量中，转义序列以反斜杠(\)开始，可以将任意的字节插入字符串中。

原生字符串字面量的书写形式是`...`，使用反引号。

- 原生字符串字面量内，转义序列不起作用，实际内容与字面写法严格一致，包括反斜杠和换行符\n。
- 唯一的特殊处理是回车符\r会被删除。
- 正则表达式往往含有大量反斜杠，可以方便地写成原生字符串字面量。

3.6 常量

2019年5月29日 15:57

常量是一种表达式，可以保证在编译阶段就计算出表达式的值，并不需要等到运行时，从而使编译器得以知晓其值。

与变量类似，同一个声明可以定义一系列常量，这适用于一组相关的值：

```
const (
	E = 2.17
	Pi = 3.141592653589793
)
```

常量声明可以同时指定类型和值，如果没有显式指定类型，则类型根据右边的表达式推断。

若同时声明一组常量，除了第一项之外，其他项在等号右侧的表达式都可以忽略，这意味着会复用前面一项的表达式及其类型。

3.6.1 常量生成器iota

3.6.2 无类型常量

4. 复合数据类型

2019年6月3日 18:00

4.1 数组

2019年6月4日 14:27

数组是具有固定长度且拥有零个或多个相同数据类型元素的序列。
var a [3]int

4.2 slice

2019年6月3日　　18:01

slice表示一个拥有相同类型元素的可变长度的序列。通常写成[]T，其中元素类型都是T。

- slice和数组是紧密关联的。slice是一种轻量级的数据结构，可以用来访问数组的部分或者全部元素，这个数组称为slice的底层数组。
- slice有3个属性：指针、长度和容量。
  指针指向数组的第一可以从slice中访问的元素，这个元素并不一定是数组的第一个元素；长度是指slice中的元素个数，它不能超过slice容量；容量的大小通常是从slice的起始元素到底层数组的最后一个元素间的元素个数。
  内置函数len和cap用来返回slice的长度和容量。
- 一个底层数组可以对应多个slice，这些slice彼此之间的元素还可以重叠。

slice类型的零值是nil。值nil的slice没有对应的底层数组。

- 值为nil的slice长度和容量都是零，但是也有非nil的slice长度和容量是零。

```
var s []int                 //len(s) == 0, s == nil
s = nil                     //len(s) == 0, s == nil
s = []int(nil)              //len(s) == 0, s == nil
s = []int{}                 //len(s) == 0, s != nil
s = make([]int,3)[3:].      //len(s) == 0, s != nil
```

  所以，如果想要检查一个slice是否为空，使用len(s)==0,而不是s == nil。
- 对于任何类型，如果它们的值可以是nil，那么这个类型的nil值可以使用一种转换表达式，例如：[]int(nil)

内置函数make可以创建一个具有指定元素类型、长度和容量的slice。其中容量参数可以省略，此时，slice的长度和容量相等。

4.2.1 append函数

内置函数append用来将元素追加到slice的后面。

4.3 map

2019年6月4日 11:43

map是一个拥有键值对元素的无序集合(字典)。键的值是唯一的，重复的会覆盖之前的键值。
map是散列表的引用，map的类型是map[k]v。k和v是map的键和值对应的数据类型。

- 所有的键都拥有相同的数据类型，同时所有的值也拥有相同的数据类型
- 键的类型必须是可以通过操作符==进行比较的数据类型，所以map可以检测某一个键是否已经存在。
- map类型的零值是nil，也就是说没有引用任何散列表

1.创建map

内置函数make可以用来创建一个map。

```
ages := make(map[string]int)
```

也可以使用map字面量来创建一个带初始化键值对元素的map。

```
ages := map[string]int {
        "alice": 31,
        "charlie": 34,
}
```

等价于:

```
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

因此， 新的空map的另一种表达式是 map[string]int{}

2.移除元素

使用内置函数delete从字典中根据键移除一个元素。

```
delete(ages, "alice")
```

即使键不在map中，上面的操作也是安全的。map使用给定的键来查找元素，如果对应的元素不存在，就返回值类型的零值。

3.map元素的地址

- map元素不是一个变量，不可以获取它的地址。
  一个原因是map的增长可能会导致已有元素被重新散列到新的存储位置，这样就可能使获取的地址无效。
- map中元素的迭代顺序是不固定的，不同的方法会使用不同的散列算法，得到不同的元素顺序。
- 如果需要按照某种顺序来遍历map中的元素，必须显式地来给键排序。

4.map类型的零值是nil，也就是说，没有引用任何散列表。

- 大多数的map操作都可以安全地在map的零值nil上执行，包括查找、删除、获取元素个数（len）、range循环。因为这和空map的行为一致。
- 向零值map中设置元素会导致错误，必须先初始化map。

ages1["bob"] = 30 //error

- 通过下标访问map中的元素总是会有值。
  如果键在map中，会得到键对应的值；如果不在，会得到map值类型的零值。
- 判断某个元素是否在map中：
  if age, ok := ages["bob"]; !ok {...}
  第二个值用来报告元素是否存在。

nil和空map：//与slice类似

```
var ages1 map[string]int
var ages2 = map[string]int{}
ages3 := make(map[string]int)

fmt.Println(ages1==nil)//true
fmt.Println(ages2==nil)//false
fmt.Println(ages3==nil)//false
```

可见，ages1是零值，ages2和ages3是空map但不是零值。

5.和slice一样，map不可以比较，唯一合法的是和nil比较。

4.4 struct

2019年6月4日 11:43

1.点号(.)可以用在结构体指针上。//而不是c中的->

2.如果一个结构体的成员变量名称是首字母大写的，那么这个变量是可导出的，这个是go最主要的访问控制机制。

3.命名结构体不可以定义一个拥有相同结构体类型的成员变量。

- 也就是一个聚合类型不可以包含它自己(同样的限制对数组也适用)
- 但是结构体中可以定义结构体类型对应的指针类型。

```
Eg: type tree struct {
        value int
        left, right *tree
    }
```

4.4.1 结构体字面量

结构体类型的值可以通过结构体字面量来设置。有两种格式的结构体字面量。

第一种格式，按正确的顺序为每个成员变量指定一个值。

第二种格式，通过指定部分或全部成员变量的名称和值来初始化结构体变量。

出于效率的考虑，大型的结构体通常都使用结构体指针的方式直接传递给函数或者从函数中返回。

4.4.2 结构体比较

如果结构体的所有成员变量都是可以比较的，那么这个结构体就是可比的。

和其他可比较的类型一样，可比较的结构体类型都可以作为map的键类型。

4.4.3 结构体嵌套和匿名成员

go允许我们定义不带名称的结构体成员，只需要指定类型即可，这种结构体成员叫做匿名成员。

这个结构体成员的类型必须是一个命名类型或者指向命名类型的指针。

- 因为有了这种结构体嵌套的功能，我们能够直接访问到我们需要的变量而不是指定一大串中间变量。
- 结构体字面量必须遵循形状类型的定义。
- 因为匿名成员拥有隐式的名字，所以不能在一个结构体里定义两个相同类型的匿名成员，否则会引起冲突。
- 由于匿名成员的名字是由它们的类型决定的，因此它们的可导出性也由它们的类型决定。
- 以快捷方式访问匿名成员的内部变量同样适用于访问匿名成员的内部方法。

4.5 JSON

2019年6月5日 18:50

标准库 encoding/json

JSON是JavaScript值的Unicode编码。
JSON的数组是一个有序的元素序列，每个元素之间用逗号分隔，两边使用[]括起来。

- JSON的数组用来编码GO里面的数组和Slice。

JSON的对象是一个从字符串到值的映射，name:value，每个元素之间用逗号分隔，两边使用{}括起来。

- JSON的对象用来编码GO里面的map(键为字符串类型)和结构体。

把GO的数据结构转换成JSON称为marshal，是通过json.Marshal实现的。
Marshal生成了一个字节slice，其中包含一个不带任何多余空白字符的很长的字符串。这种紧凑的表示方法包含了所有的信息但是难以阅读，JSON.MarshalIndent可以输出整齐格式化过的结果。这个函数有两个参数，一个是定义每行输出的前缀字符串，另外一个是定义缩进的字符串。

Marshal使用GO结构体成员的名称作为JSON对象里面的字段的名称(通过反射的方式)。只有可导出的成员可以转换为JSON字段，这就是为什么将GO结构体里面的所有成员都定义为首字母大写。

通过成员标签定义实现 定义JSON对象里面的字符按的名称。成员标签定义是结构体在编译期间关联的一些元信息：
Eg:
Year int `json:"released"`
Color bool `json:"color,omitempty"`

成员标签习惯上是由一串空格分开的标签键值对 key:"value"组成。
因为标签的值使用""包含，所以一般标签都是原生的字符串字面量(即``)。
Color的标签还有一个额外的选项，omitempty，表示如果这个成员的值为零或者为空，则不输出这个成员到JSON中。

Marshal 的逆操作将JSON字符串解码为GO数据结构，这个过程叫做unmarshall，由json.Unmarshal实现。
通过合理的定义GO的数据结构，可以选择将哪部分JSON数据解码到结构体中，将哪些数据丢弃。

- 在Unmarshal阶段，JSON字段的名称关联到GO结构体成员的名称是忽略大小写的。

json.Encoder 流式编码器
json.Decoder 流式解码器，依次从字节流里面解码出多个JSON实体

4.6 文本和HTML模板

2019年8月1日 星期四　　下午7:34

text/template包和html/template包

这两个包提供了一种机制，可以将程序变量的值带入到文本或者HTML模板中。
模板是一个字符串或者文件，它包含一个或者多个{{...}}称为操作。

- 大多数的字符串是直接输出的
- 操作可以引发其他的行为。每个操作在模板语言里面都对应一个表达式，提供简单但强大的功能。
  包括：输出值，选择结构体成员，调用函数和方法，描述控制逻辑（if-else,range）,实例化其他模板等。

在操作中，符号|会将前一个操作的结果当作下一个操作的输入，和unix中的shell管道类似。

通过模板输出结果需要两个步骤：

- 首先需要解析模板并转换为内部的表示方法，然后在指定的输入上面执行。

5. 函数

2019年6月4日 16:18

5.1 函数声明

2019年6月4日　　16:18

每个函数声明都包含一个名字、一个形参列表、一个可选的返回列表以及函数体。

```
func name(parameter-list) (result-list) {
        body
}
```

返回值：

- 返回列表指定了函数返回值的类型。
  当函数返回一个未命名的返回值或者没有返回值的时候，返回列表可以为空。
- 返回值也可以像形参一样命名。
  每一个命名的返回值会声明为一个局部变量，并根据变量类型初始化为相应的0值。
- 函数存在返回值列表时，必须显式地以return语句结束。
  除非函数明确不会走完整个执行流程。(死循环，宕机等)

形参：

- 空白标识符 _ 用来强调该形参在函数中未使用。
- go语言没有默认参数值的概念也不能指定实参名。
- 实参是按值传递的。

函数的类型称为函数签名。当两个函数拥有相同的形参列表和返回列表时，认为这两个参数这两个函数的类型是相同的。形参或返回值的名字不会影响到函数类型。

函数形参和命名返回值同属于函数最外层作用域的局部变量。
有些函数没有函数体，说明这些函数是使用go以外的语言实现的。

5.2 递归

2019年6月16日 星期日　　上午12:42

相比于许多编程语言使用固定长度的栈，go语言使用了可变长度的栈，栈的大小会随着使用而增长，可达到1GB左右的上限。因此可以安全使用递归，而不必担心栈溢出的问题。

5.3 多返回值

2019年6月16日 星期日　　上午12:46

go语言的垃圾回收机制将回收未使用的内存，但不能指望它会释放未使用的操作系统资源，比如打开的文件以及网络连接。必须显示关闭它们。

调用一个返回一组值的函数，如果调用者要使用这些返回值，必须显式地将返回值赋给变量。忽略其中某个返回值可以将它赋给一个空标识符 _ 。

- 函数可以是调用另一个多值返回的函数来返回一个多值结果。
- 一个多值调用可以作为单独的实参传递给拥有多个形参的函数中。

一个函数如果有命名的返回值，可以省略return语句的操作数，成为裸返回。

- 裸返回是将每个命名返回结果按顺序返回的快捷方式。
- 裸返回不利于代码的理解性，应保守使用。

5.4 错误

2019年6月16日 星期日　　上午10:46

错误处理是包的API设计或者应用程序用户接口的重要部分，发生错误只是许多预料行为中的一种而已。

习惯上将错误值作为最后一个结果返回。

- 如果错误只有一种情况，结果通常设置为布尔类型
- 更多时候，错误的原因可能多种多样，调用者需要一些详细的信息。错误的结果类型往往是error。
    - error是内置的接口类型。
    - 错误可能是空值或非空值，空值意味着成功，非空值意味着失败。
      非空的错误类型有一个错误消息字符串，可以通过调用它的Error方法或者通过调用fmt.Println(err)直接输出错误消息。

go通过使用普通的值而非异常来报告错误。go的异常只是针对程序bug导致的意料之外的错误，而不是作为常规的错误处理方法。

5.4.1 错误处理策略

5.4.2 文件结束标识

io.EOF

5.5 函数变量

2019年6月16日 星期日　　上午10:51

函数变量(函数名)也有类型，它们可以赋值给变量或者传递或者从其他函数返回。
函数变量的零值是nil，调用一个nil的函数变量将导致宕机。
函数变量可以和空值比较，但它们本身不可以互相比较，也不可以作为map的键值。
函数变量使得可以将函数作为参数进行传递。

5.6 匿名函数

2019年6月4日　　16:22

命名函数只能在包级别的作用域进行声明，但我们能够使用函数字面量在任何表达式内指定函数变量。
函数字面量就像函数声明，但在func关键字后面没有函数的名称。它是一个表达式，它的值称为匿名函数。

- 以这种方式定义的函数能够获取到整个词法环境，因此里层的函数可以使用外层函数中的变量。

Eg:

```
func squares() func() int {
    var x int
    return func() int {
        x++
        return x*x
    }
}

func main() {
    //f := squares
    //fmt.Println(f()())        //1
    //fmt.Println(f()())        //1
    //fmt.Println(f()())        //1++++++-

    f1 := squares()
    fmt.Println(f1())     //1
    fmt.Println(f1())     //4
    fmt.Println(f1())     //9
}
```

这个求平方的示例演示了函数变量不仅是一段代码还可以拥有状态。里层的匿名函数能够获取和更新外层squares函数的局部变量。
//这些隐藏的变量引用就是我们把函数归类为引用类型而且函数变量无法进行比较的原因。
//go程序员通常把函数变量称为闭包。
变量的生命周期不是由它的作用域决定的，变量x在main函数中返回squares函数后依旧存在(x在这个时候是隐藏在函数变量f中的)。

5.7 变长函数

2019年6月4日 16:39

变长函数被调用的时候可以有可变的参数个数。Eg: fmt.Printf
在参数列表最后的类型名称之前使用...表示声明一个变长函数，调用该函数时可以传递该类型任意数目的参数。

```
func sum(vals ... int) int {
    ...
}
```

调用者显式地申请一个数组，将实参复制给这个数组，并把一个数组slice传递给函数。

```
values := []int{1,2,3,4}
sum(values...)
```

当实参已经存在于一个slice中时，调用一个变长函数，需要在最后一个参数后面放一个省略号。这种做法和 sum(1,2,3,4) 是等同的。

尽管 ...int 参数就像函数体内的slice，但是变长函数的类型和一个带有普通slice参数的函数的类型并不相同。

```
Func f(...int) {}
Func f([]int) {}
```

变长函数通常用于格式化字符串。

5.8 延迟函数调用

2019年6月6日 9:49

一个defer语句就是一个普通的函数或方法调用，在调用之前加上关键字defer。

- defer语句经常使用于成对的操作，比如打开和关闭，连接和断开，加锁和解锁等。正确使用defer语句的地方是在成功获取资源之后。
- 无论在正常情况下，执行return语句或函数执行完毕，还是在异常情况下(宕机等)。实际的调用推迟到包含defer语句的函数结束后才执行。
- defer语句没有限制使用次数，执行时以调用defer语句顺序的逆序进行。

defer语句也可以用来调试一个复杂的函数，即在入口和出口设置调试行为。

- 在函数刚进入的时候执行输出，然后返回一个函数变量，当其被调用的时候执行退出函数的操作。

因为defer函数不到函数的最后一刻不会被执行，因此要注意循环里的defer函数使用。

5.9 宕机

2019年6月16日 星期日　　上午11:25

Go语言需要在运行时检查某些错误，如数组越界，解引用空指针，当这些错误发生时，就会发生宕机。

runtime包提供了转储栈的方法可以诊断错误。

```
func f(x int) {
    fmt.Printf("f(%d)\n", x+0/x)
    defer fmt.Printf("defer %d\n", x)
    f(x - 1)
}

func printStack() {
    var buf [4096]byte
    n := runtime.Stack(buf[:], false)
    os.Stdout.Write(buf[:n])
}

func main() {
    defer printStack()
    f(3)
}
```

5.10 恢复

2019年6月16日 星期日　　上午11:25

如果内置的reover函数在延迟函数的内部调用，而且这个包含defer语句的函数发生宕机，recover会终止当前的宕机状态并且返回宕机的值。函数不会从之前宕机的地方继续运行而是正常返回。

如果recover在其他任何情况下运行则它没有任何效果且返回nil。

6. 方法

2019年6月4日 17:05

6.1 方法声明

2019年6月4日　　16:48

方法的声明和普通函数的声明类似，只是在函数名字前面多了一个参数。这个参数把这个方法绑定到这个参数对应的类型上。

```
package geometry

import "math"

type Point struct{X, Y float64}

//普通函数
func Distance(p, q Point) float64 {
	return math.Hypot(q.X - p.X, q.Y - p.Y)
}

//Point类型的方法
func (p Point) Distance(q Point) float64 {
	return math.Hypot(q.X - p.X, q.Y - p.Y)
}
```

附加的参数p称为方法的接收者。

go语言中，接收者不使用特殊名(this, self等)，而是我们自己选择接收者的名字。

由于接收者会频繁地使用，因此最好能够选择简短且在整个方法中名称始终保持一致的名字。

最常用的接收者名称就是取类型名称的首字母。

调用方法时，接收者在方法名的前面。

```
p := Point{1,2}
q := Point{4,6}
fmt.Println(Distance(p, q))　//函数调用
fmt.Println(p.Distance(q))　//方法调用
```

上面两个Distance函数声明没有冲突。第一个声明一个包级别的函数，称为geometry.Distance，第二个声明一个类型Point的方法，因此它的名字时Point.Distance。

表达式p.Distance称作选择子(selector)，选择子也用于选择结构类型中的某些字段值，p.X。

由于方法和字段来自同一个命名空间，因此在Point结构类型中声明一个叫作X的方法会与字段X冲突。

可以将方法绑定到任何类型上，可以很方便地为简单类型（如数字、字符串、slice、map，函数等）定义附加的行为。只要它的类型既不是指针类型也不是接口类型。

- 不允许指针类型进行方法声明。

类型拥有的所有方法名都必须是唯一的，但不同的类型可以使用相同的方法名，没有必要使用附加的名称来修饰方法名用于区分。因此，方法的命名可以比函数更简洁。

Eg: PointDistance, PathDistance

6.2 指针接收者的方法

2019年6月4日 17:06

由于主调函数会复制每一个实参变量，如果函数需要更新一个变量，或者如果一个实参太大而我们希望避免复制整个实参，因此必须使用指针来传递变量的地址。这同样也适用于更新接收者：我们把它绑定到指针类型，比如Point *。

```
func (p *Point) ScaleBy(factor float64) {
    ...
}
```

这个方法的名字是(*Point).ScaleBy。括号是必须的，否则会被解析为*(Point.ScaleBy)。

- 习惯上，如果类型的任何一个方法使用指针接收者，那么所有的方法都应该使用指针接收者。
- 命名类型(Point)与指向它们的指针(Point *)是唯一可以出现在接收者声明处的类型。为了防止混淆，不允许本身是指针的类型进行方法声明。

如果接收者p是Point类型的变量，但方法要求一个*Point类型的接收者，可以简写为: p.ScaleBy(2)
实际上，编译器会对变量进行&p的隐式转换。只有变量才允许这么做，不能够对一个不能取地址的Point接收者参数调用*Point方法（如：字面量），因为无法获取临时变量的地址。

如果接收者pptr是*Point类型，但方法要求一个Point类型的接收者。pptr.Distance()总是合法的，因为有方法从地址获取Point的值。编译器会自动插入一个隐式的*操作符。

如果所有类型T方法的接收者是T自己（而非*T），那么复制它的实例是安全的，调用方法的时候都必须进行一次复制。但是任何方法的接收者是指针的情况下，应该避免复制T的实例，因为这么做可能会破坏内部原本的数据。

nil是一个合法的接收者

方法的接收者允许nil作为实参，尤其是当nil是类型中有意义的零值（map、slice等）。

- 当定义一个类型允许nil作为接收者时，应当在文档注释中显式地标明。

6.3 通过结构体内嵌组成类型

2019年6月5日　　20:29

```
type Point {X, Y float64}

type ColoredPoint struct {
        Point
        Color color.RGBA
}
```

可以通过类型为*ColoredPointd*的接收者调用内嵌类型Point的方法，即使在*ColoredPoint* 类型没有声明过这个方法的情况下。以这种方法，内嵌允许构成复杂的类型，该类型由许多字段构成，每个字段提供一些方法。

匿名字段类型可以是指向命名类型的指针，这时，字段和方法间接地来自于所指向的对象。

```
        *Point
```

当编译器处理选择子（比如p.ScaleBy）时，首先查找直接声明的方法ScaleBy，之后再从来自内嵌字段的方法中进行查找。以此类推，内嵌字段的内嵌字段。

```
        var cache = struct {
                sync.Mutex
                mapping map[string]string
        } {
                mapping: make(map[string]string),
        }

        func Lookup(key string) string {
                cache.Lock()
                defer cache.Unlock()
                return cache.mapping[key]
        }
```

sync.Mutex是内嵌的，它的方法也包含进了结构体中，允许我们直接使用cache变量本身进行加锁。

6.4 方法变量与表达式

2019年6月5日　　20:29

通常我们都在相同的表达式里使用和调用方法，p.Distance()。//方法的使用和方法的声明保持一致
把两个操作分开也是可以的：
选择子 p.Distance 可以赋给一个方法变量，它是一个函数，把方法绑定到一个接收者p上，函数只需要提供实参而不需要提供接收者就能够调用。

```
p := Point{1,2}
q := Point{4,6}
distanceFromP := p.Distance  //方法变量
distanceFromP(q)
```

与方法变量相关的是方法表达式：T.f或者(*T).f，是一种函数变量，把原来的方法接收者替换成函数的第一个形参。

6.5 示例：位向量

2019年6月5日 20:30

6.6 封装

2019年6月4日 17:08

如果变量或者方法是不能通过对象访问到的，这称作封装的变量或者方法。封装有时候称为数据隐藏。

go语言只有一种方式控制命名的可见性：定义的时候，首字母大写的标识符是可以从包中导出的，而首字母没有大写的则不导出。同样的机制也作用于结构体内的字段和类型中的方法。结论就是，要封装一个对象，必须使用结构体。

go语言中封装的单元是包而不是类型。无论是在函数内的代码还是方法内的代码，结构体类型内的字段对于同一个包中的所有代码都是可见的。

封装提供了三个优点：

- 因为使用方不能直接修改对象的变量，所以不需要更多的语句用于检查变量的值；
- 隐藏实现细节可以防止使用方依赖的属性发生改变，使得设计者可以更加灵活地改变API的实现而不破坏兼容性。
- 防止使用者肆意改变对象内的变量。

7. 接口

2019年6月4日 17:20

接口类型是对其他类型行为的概括和抽象。通过使用接口，可以写出更加灵活和通用的函数，这些函数不用绑定在一个特定的类型实现上。

go语言的接口是隐式实现的。即对于一个具体的类型，无须声明它实现了哪些接口，只要提供接口所需的方法即可。

7.1 接口即约定

2019年6月4日 17:22

之前提到的类型都是具体类型。具体类型指定了它所含数据的精确布局，还暴露了基于这个精确布局的内部操作。

GO语言还有一种类型称为接口类型。接口是一种抽象类型，它并不暴露所含数据的布局或内部结构，也没有那些数据的基本操作，它所提供的仅仅是一些方法而已。
拿到一个接口类型，你无从知道它是什么，能知道的仅仅是它提供了哪些方法。

7.2 接口类型

2019年6月5日 21:23

一个接口类型定义了一套方法，如果一个具体类型要实现该接口，那么必须实现接口类型定义中的所有方法。

可以通过组合已有接口得到新接口，称为嵌入式接口。

- 与嵌入式结构类似，可以直接使用一个接口，而不用逐一写出这个接口所包含的方法。

7.3 实现接口

2019年7月1日 星期一　　下午2:36

如果一个类型实现了一个接口要求的所有方法，那么这个类型就实现了这个接口。通常称为XX类型是一个XX的接口类型。Eg: *butes.Buffer是一个io.Writer。

接口的赋值规则：仅当一个表达式(或接口)实现了一个接口时，这个表达式(或接口)才可以赋值给该接口。

类型T和类型*T在接口实现上的区别。

空接口类型 interface{}
正因为空接口类型对其实现类型没有任何要求，所以可以把任何值赋给空接口类型。

非空的接口类型，通常由一个指针类型来实现，特别是当接口类型的一个或多个方法暗示会修改接收者的情形。一个指向结构体的指针才是最常见的方法接收者。

7.4 使用flag.Value来解析参数

2019年8月6日 星期二　　上午11:29

7.5 接口值

2019年7月1日 星期一　　下午3:38

一个接口类型的值（接口值）有两部分：一个具体类型和该类型的一个值。称为接口的动态类型和动态值。

- 用 类型描述符 来提供每个类型的具体信息，比如类型的名字和方法。
- 对于一个接口值，类型部分就用对应的类型描述符来表述。

在GO语言中，变量总是被初始化一个特定的值，接口也不例外。接口的零值就是把它的动态类型和动态值都设为nil。

接口值可以用==和!=操作符做比较。

- 如果两个接口值都是nil或者二者的动态类型完全一致且二者的动态值相等，那么两个接口相等。
- 因为接口可以比较，所以它们可以作为map的键，也可以作为switch语句的操作数。
- 需要注意的是，在比较两个接口值时，如果两个接口值的动态类型一致，但是对应的动态值是不可比较的（如slice），那么这个比较会以崩溃的方式失败。

可以使用fmt包的%T拿到接口值的动态类型，fmt用反射来拿到接口动态类型的名字。

含有空指针的非空接口：

空的接口值（其中不包含任何信息）与仅仅动态值为nil的接口值是不一样的。

```
var buf *bytes.Buffer //动态值为空，但动态类型为*bytes.Buffer,是包含空指针的非空接口
buf != nil //true
```

7.6 使用sort.Interface来排序

2019年8月6日 星期二　　下午3:28

sort包提供了针对任意序列根据任意排序函数原地排序的功能。sort.Sort函数对序列和其中元素的布局无任何要求。使用sort.Interface接口来指定通用排序算法和每个具体的序列类型之间的协议。

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

要对序列排序，需要先确定一个实现了如上三个方法的类型，接着把sort.Sort函数应用到上面这类方法的实例上。

7.7 HttpHandler接口

2019年8月6日 星期二　　下午3:28

ListenAndServer函数需要一个服务器地址以及一个Handler接口的实例（用来接收所有的请求）。
这个函数会一直运行，直到服务器出错（或者启动时就失败了）时返回一个非空的错误。

net/http包提供了一个请求多工转发器ServeMux，用来简化URL和处理器之间的关系。一个ServeMux把多个http.Handler组合成单个http.Handler。

http.HandlerFunc不仅仅是一个函数类型，还拥有自己的方法，也满足接口http.Handler。

为了简便起见，net/http包提供了一个全局的ServeMux实例DefaultServeMux，以及包级别的注册函数http.Handle和http.HandleFunc。要让DefaultServeMux作为服务器的主处理程序，只需将nil传给ListenAndServer。

7.8 error接口

2019年8月6日 星期二　　下午3:28

7.10 类型断言

2019年7月1日 星期一　　下午10:12

类型断言时一个作用在接口值上的操作，写出来类似于x.(T)。x是一个接口类型的表达式，T是一个类型（称为断言类型）。类型断言会检查作为操作数的动态类型是否满足指定的断言类型。

如果断言类型T是一个具体类型，那么类型断言会检查x的动态类型是否就是T。

- 如果检查成功，类型断言的结构就是x的动态值。
- 如果检查失败，操作崩溃。

如果断言类型T是一个接口类型，那么类型断言会检查x的动态类型是否满足T。

- 如果检查成功，动态值并没有提取出来，结果仍是一个接口值，接口值的类型和值部分也没有更改，只是结果的类型为接口类型T。
  换句话说，类型断言是一个接口值表达式，从一个接口类型变为拥有另外一套方法的接口类型（通常方法数量是增多），但是保留了接口值中的动态类型和动态值部分。

无论哪种类型作为类型断言，如果操作数是一个空接口值，类型断言都失败。

经常无法确认一个接口值的动态类型，这时就需要检测它是否是某一特定类型。

```
if f, ok := w.(*os.File); ok {
        ...
}
```

7.11 使用类型断言来识别错误

2019年8月6日 星期二　　下午10:03

8. goroutine 和通道

2019年6月5日 21:28

GO有两种并发编程的风格。

- 执行体(goroutine)和通道(channel)，它们支持通信顺序进程(Communicating Sequential Process, CSP)，CSP是一个并发的模式，在不同的执行体之间传递值，但是变量本身局限于单一的执行体。
- 共享内存多线程的传统模型。

8.1 goroutine

2019年6月5日 21:37

每一个并发执行的活动称为goroutine。

当一个程序启动时，只有一个goroutine来调用main函数，称它为主goroutine。新的goroutine通过go语句进行创建。语法上，在普通函数或方法调用前加上go关键字前缀，这样可以使函数在一个新建的goroutine中运行。go语句本身的执行立即完成。

main函数返回时，所有goroutine都暴力地直接终止，然后程序退出。

8.2 示例 并发时钟服务器

2019年7月30日 星期二　　下午8:28

8.3 示例 并发回声服务器

2019年7月30日 星期二　　下午8:29

8.4 通道

2019年6月14日 14:45

groutine是go程序并发的执行体，通道是goroutine之间的连接。

- 通道是可以让一个goroutine发送特定值到另一个goroutine的通信机制。
- 每个通道是一个具体类型的导管，叫做通道的元素类型。一个int类型元素的通道写为chan int。
- 使用make函数创建通道。
  ch := make(chan int)
  和map一样，通道是一个使用make创建的数据结构的引用。当复制或作为参数传递到一个函数时，复制到是引用，这样调用者和被调用者都引用同一份数据结构。
- 和其他引用类型一样，通道的零值是nil。
- 同种类型的通道可以使用==比较。
- 通道有两个主要操作：发送和接收，两者统称为通信。
  - 发送语句中，通道和值分别在 <- 的左右两边。ch <- x
  - 接收语句中，<-放在通道操作数前面。x = <- ch
- 通道的关闭操作设置一个标志位来指示当前已发送完毕。
  - 关闭后的发送操作将导致宕机。
  - 在一个已关闭的通道上进行接收操作，将获取所有已经发送的值，直到通道为空。这时，任何接收操作会立刻完成，同时获取到一个通道元素类型对应的零值。

使用简单的make创建的的通道称为无缓冲（unbuffered）通道。make还可以接受第二个参数，表示通道容量的整数。如果容量为0，make创建一个无缓冲通道。

```
Ch = make(chan int)    //无缓冲通道
Ch = make(chan int, 0) //无缓冲通道
Ch = make(chan int, 3) //容量为3的缓冲通道
```

试图关闭一个 已经关闭的通道 / 空通道 将导致宕机。

8.4.1 无缓冲通道

无缓冲通道上的发送操作将会阻塞，直到另一个goroutine在对应的通道上执行接收操作。同样，如果接收操作先执行，接收方goroutine将阻塞，直到另一个goroutine在同一个通道上发送一个值。因此无缓冲通道也称为同步通道。

为了让程序等待后台的goroutine在完成后再退出，可以使用一个来同步两个goroutine。

8.4.2 管道 pipeline

8.4.3 单向通道类型

go提供了单向通道类型：

- 类型 chan<- int 是一个只能发送的通道；
- 类型 <-chan int 是一个只能接收的通道；
- close操作说明了通道上没有数据再发送，因此，仅仅在发送方goroutine上才能调用它。
- 可以将双向通道转换成单向通道，但绝不允许将单向通道转换为双向通道。

8.4.4 缓冲通道

缓冲通道有一个元素队列，队列的最大长度在创建的时候通过make的容量参数来设置。

```
ch = make(chan string, 3) //可以容纳3个字符串的缓冲通道
```

- 在缓冲通道上的发送操作在队列的尾部插入一个元素，接收操作从队列的头部移除一个元素。
- 如果通道满了，发送操作会阻塞所在的goroutine直到另一个goroutine对它进行接收操作。反之，如果通道是空的，执行接收操作的goroutine阻塞。

- 可以通过调用内置的cap函数获取通道缓冲区的容量，使用len函数获取当前通道内元素的个数。

8.5 并行循环

2019年7月30日 星期二　　下午8:26

```
func task(str string) {
	time.Sleep(time.Second * 1)
	fmt.Println(str)
}

func makeThumbnail2(filenames []string) {
	ch := make(chan struct{})
	for _, f := range filenames {
		go func() { //error，不等函数执行直接返回进入下一次循环，因此传入到task中值均为456
			task(f)
			ch <- struct{}{}
		}()
	}

	for range filenames {
		<- ch
	}
}

func makeThumbnail1(filenames []string) {
	ch := make(chan struct{})
	for _, f := range filenames {
		go func(f string) { //right
			task(f)
			ch <- struct{}{}
		}(f)
	}

	for range filenames {
		<- ch
	}
}

func main()  {
	filenames := []string{"123", "456"}
	makeThumbnail1(filenames)     //456,123
	makeThumbnail2(filenames)     //456,456
}
```

makeThumbnail2中变量f的值被所有的匿名函数值所共享并且被后续的迭代更新。新的goroutine执行字面量函数，for循环可能已经更新f，所以当这些goroutine读取f的值时，它们所看到的都是silce的最后一个元素。

8.6 示例 并发web爬虫

2019年7月30日 星期二　　下午8:26

8.7 使用select多路复用

2019年7月30日 星期二　　下午8:27

类似switch语句，select拥有一系列的情况和一个可选的默认分支。
每一个情况指定一次通信（在一些通道上进行发送或接收操作）和关联的一段代码块。

- select一直等待，直到一次通信来告知有一些情况可以执行。
- 如果多种情况同时满足，select随机选择一个，这样保证每一个通道有相同的机会被选中。

如果试图在一个通道上发送或接收，但又不想在通道没有准备好的情况下被阻塞(非阻塞通信)。可以设置select的默认值，用来指定在没有其他通信发生时可以立即执行的动作。

通道的零值是nil。

- 在nil通道上发送或接收将永远阻塞。
- select语句中如果通道是nil，它将永远不会被选中。

time.tick使用模式：

```
ticker := time.NewTicker(1 * time.Second)
<- ticker.C
ticker.Stop()
```

如果不Stop，会发生goroutine泄漏。

8.8 示例 并发目录遍历

2019年7月30日 星期二　　下午8:27

8.9 取消

2019年7月30日 星期二　　下午8:28

有时候需要让一个goroutine停止它当前的任务。
对于取消操作，我们需要一个可靠的机制在一个通道上广播一个事件。
当一个通道关闭且已取完所有发送的值之后，接下来的接收操作立即返回，得到零值。我们可以利用它创建一个广播机制：不在通道上发送值，而是关闭它。

8.10 示例 聊天服务器

2019年7月30日 星期二　　下午8:28

9. 使用共享变量实现并发

2019年6月5日 21:28

9.1 竞态

2019年8月1日 星期四　　上午10:43

如果一个类型的所有可访问方法和操作都是并发安全的，则这个一个并发安全的类型。

仅在文档指出类型是安全的情况下，才可以并发访问一个变量。对于绝大部分变量，如果要回避并发访问：

- 要么限制变量只存在于一个goroutine中，
- 要么维护一个更高层的互斥不变量。

与之对应的，导出的包级别的函数通常可以认为是并发安全的。

- 因为包级别的变量无法限制在一个goroutine内，所以那些修改这些变量的函数就必须采用互斥机制。
- 不导出的函数可以不是并发安全的。

竞态是指在多个goroutine按某些交错顺序执行时，程序无法给出正确的结果。
数据竞态发生于两个goroutine并发读写同一个变量并且至少其中一个是写入。

- 当数据竞态的变量类型是大于一个机器字长的类型（接口，字符串，slice）时，事情就更复杂了。

避免数据竞态的三种方法：

- 避免修改变量
- 避免从多个goroutine访问同一个变量
  使用通道来向受限goroutine发送查询请求或者更新变量。
  使用通道请求来代理一个受限变量所有访问的groutine称为该变量的监控goroutine。
  即使一个变量无法在整个生命周期受限于单个goroutine，加以限制仍然可以是解决并发访问的好方法。
    可以通过借助通道来把共享变量的地址从上一步传到下一步，从而在流水线上的多个goroutine之间共享该变量。流水线中的每一步，在把变量地址传给下一步之后就不再访问该变量了。
- 互斥机制。允许多个goroutine访问同一个变量，但在同一时间只有一个goroutine可以访问。

9.2 互斥锁 sync.Mutex

2019年8月1日 星期四　　上午10:43

1.使用一个容量为1的通道来保证同一时间最多只有一个goroutine能访问共享变量。

2.互斥锁模式：

```
var mu sync.Mutex
mu.Lock()
defer mu.Unlock()
```

典型的并发模式：
几个导出函数封装了一个或多个变量，于是只能通过这些函数来访问这些变量。每个函数在开始时申请一个互斥锁，在结束时再释放掉。这种函数、互斥锁、变量的组合方式称为监控模式。

无论是为了保护包级别的变量，还是结构体中的字段，当使用一个互斥量时，必须确保互斥量本身以及被保护的变量都没有导出。

9.3 读写互斥锁 sync.RWMutex

2019年8月1日 星期四　　下午2:29

只允许读操作可以并发执行，但写操作需要获得完全独享的访问权限。这种锁称为多读单写锁。sync.RWMutex

```
var mu sync.RWMutex
var balance int

func Balance() int {
        mu.RLock()
        defer mu.RUnlock()
        return balance
}
```

Rlock(), Runlock() 方法获取和释放一个读锁（共享锁）。
Lock(), Unlock() 方法获取和释放一个写锁（互斥锁）。

仅在绝大部分goroutine都在获取读锁并且锁竞争比较激烈时，RWMutex才有优势。因为RWMutex需要更复杂的内部薄记工作，所以在竞争不激烈时它比普通的互斥锁慢。

9.4 内存同步

2019年8月1日 星期四　　下午2:41

同步不仅涉及多个goroutine的执行顺序问题，还会影响到内存。
现代的计算机一般都会有多个处理器，每个处理器都有内存的本地缓存。为了提高效率对内存的写入是缓存在每个处理器中的，只在必要时才刷回到内存。甚至刷回的顺序都可能与goroutine的写入顺序不一致。
通道通信或者互斥锁操作这样的同步原语都会导致处理器把累积的写操作刷回内存并提交，所以这个时刻之前goroutine的执行结果就保证了对运行在其他处理器的goroutine可见。

```
func main() {
	var x, y int

	go func() {
		x = 1
		fmt.Printf("y:%d\n", y)
	}()

	go func() {
		y = 1
		fmt.Printf("x:%d\n", x)
	}()

	time.Sleep(time.Second)
}
```

可能出现的结果：
x:0 y:0
y:0 x:0

尽管很容易把并发简单理解为多个goroutine中语句的某种交错执行方式，但正如上面的例子所显示，这并不是一个现代编译器和CPU的工作方式。

- 因为赋值和Print对应不同的变量，所以编译器就可能会认为两个语句的执行顺序不会影响结果，然后就交换了这两个语句的执行顺序。
- 如果两个goroutine不在同一个CPU上执行，每个CPU都有自己的缓存，那么一个goroutine的写入操作在同步到内存之前对另一个goroutine是不可见的。

这些并发问题可以通过采用简单、成熟的模式来避免：即在可能的情况下，把变量限制到单个goroutine中，对于其他变量，使用互斥锁。

9.5 延迟初始化 sync.Once

2019年8月1日 星期四　　下午3:09

延迟一个昂贵的初始化步骤到有实际需要的时刻是一个很好的实践。

关于并发的直觉都不可靠：
编译器和CPU在能保证每个goroutine都满足串行一执行的基础上可以自由地重排访问内存的顺序。

sync包提供了针对一次性初始化问题的解决方案：sync.Once
原理：Once包含一个布尔变量和一个互斥量，布尔变量记录初始化是否已经完成，互斥量则报回这个布尔变量和客户端的数据结构。
Once的唯一方法Do以初始化函数作为它的参数。

- 每次调用Do时会先锁定互斥量并检查里面的布尔变量。第一次调用时，布尔变量为假，Do会调用初始化函数，然后把布尔变量设置为真。

9.6 竞态检测器

2019年8月1日 星期四　　下午3:34

go语言运行时和工具链装备了一个精致并易于使用的动态分析工具：竞态检测器。
把-race命令行参数加到go build, go run, go test命令里面就可以使用该功能。

9.7 示例 并发阻塞缓存

2019年8月1日 星期四　　下午3:34

9.8 goroutine与线程

2019年8月1日 星期四　　下午3:42

9.8.1 可增长的栈

每个操作系统线程都有一个固定大小的栈内存（通常是2MB）。
一个goroutine在生命周期开始时只有一个很小的栈，典型情况下是2KB。
goroutine的栈不是固定大小的，可以按需增大或缩小。大小限制可达到1GB。

9.8.2 goroutine调度

操作系统线程由系统内核来调度。每隔几毫秒，一个硬件时钟中断发到CPU，CPU调用一个叫调度器的内核函数。因为操作系统线程由内核来调度，所以控制权限从一个线程到另外一个线程需要一个完整的上下文切换：即保存一个线程的状态到内存；恢复另外一个线程的状态；更新调度器的数据结构。

go运行时包含一个自己的调度器，这个调度器使用一个称为m:n调度的技术，它可以 复用/调度 m个goroutine到n个操作系统线程。

- go调度器只关心单个go程序的goroutine调度问题。

9.8.3 GOMAXPROCS

go调度器使用GOMAXPROCS参数来确定需要使用多少个OS线程来同时执行Go代码，默认值是机器上的CPU数量。GOMAXPROCS是m:n调度中的n。

- 正在休眠或被通道通信阻塞的goroutine不需要占用线程。
- 阻塞在I/O和其他系统调用中或调用非go语言写的函数的goroutine需要一个独立的系统线程，但这个线程不计算在GOMAXPROCS中。

可以用GOMAXPROCS环境变量或着runtime.GOMAXPROCS函数来显式控制这个参数。

```
GOMAXPROCS=1 go run test.go
```

9.8.4 goroutine没有标识

10. 包和go工具

2019年6月4日　　18:58

http://godoc.org　公共go包

关于Go tools的比较有用的flags：https://gocn.vip/article/6

10.1 引言

2019年6月4日 19:00

修改一个文件时，必须重新编译文件所在的包和所有潜在依赖它的包。

10.2 导入路径

2019年6月4日 19:08

每个包都通过一个唯一的字符串进行标识，称为导入路径，用在import声明中。

为了避免冲突，除了标准库中的包之外，其他包的导入路径应该以互联网域名作为路径开始。
Eg: github.com/go-sql-driver/mysql

10.3 包的声明

2019年6月4日 19:14

在每一个go源文件的开头都需要进行包声明。主要目的是当该包被其他包引入时作为其默认的标识符(包名)。

通常包名是导入路径的最后一段。

10.4 导入声明

2019年6月4日 19:14

go源文件可以在package声明的后面和第一个非导入声明语句前面紧接着包含零个或多个import声明。

每个导入可以单独指定一条导入路径，也可以通过圆括号列表一次导入多个包，后者更为常见。

- 导入的包可以通过空行进行分组。

  gofmt和goimport工具会自动进行分组并排序。
- 如果需要把两个名字一样的包(如 math/rand 和 crypto/rand)导入到第三个包中，导入声明就必须至少为其中的一个指定一个替代的名字来避免重复。叫做重命名导入。

  Eg: import (
  "crypto/rand"
  mrand "math/rand"
)

每个导入声明从当前包向导入的包建立一个依赖，如果这些依赖形成一个循环，go build工具就会报错。

10.5 空导入

2019年6月4日 19:33

如果导入的包的名字没有在文件中引用，就会产生一个编译错误。
为了防止"未使用的导入"错误，我们必须使用一个重命名导入，它使用一个替代的名字_，表示导入的内容为空白标识符。通常情况下，空白标识符不能被引用。
Eg: import _ "imgae/png"

10.6 包及其命名

2019年6月4日 19:36

10.7 go工具

2019年6月4日 19:38

go工具用来下载、查询、格式化、构建、测试以及安装go代码包。
go工具将不同种类的工具集合并为一个命名集。

- 它是一个包管理器(类似apt,rpm)，可以查询包的作者，计算它们的依赖关系，从远程版本控制系统下载它们。
- 它是一个构建系统，可以计算文件依赖，调用编译器、汇编器和链接器。
- 它还是一个测试驱动程序。

10.7.1 工作空间的组织

GOPATH: 指定工作空间的根。当需要切换到不同的工作空间时，更新GOPATH变量的值即可。GOPATH有三个子目录:

- src子目录包含源文件。
  每个包放在一个目录中，该目录相对于$GOPATH/src的名字是包的导入路径。
  一个GOPATH工作空间在src下包含多个源代码版本控制仓库。如 golang.org, github.com
- pkg子目录是构建工具存储编译后包的位置。
- bin子目录放置可执行程序。

GOROOT: 指定go发行版的根目录，其中提供所有标准库的包。GOROOT下面的目录结构类似于GOPATH。用户无须设置GOROOT，默认情况下go工具使用它的安装路径。

go env 命令输出与工具链相关的已经设置有效值的环境变量及其有效值，以及未设置有效值的环境变量及其默认值。

10.7.2 包的下载

go get命令可以下载单一的包，也可以使用...下载子树或仓库。该工具也计算并下载初始包所有的依赖。

- go get完成包下载后会构建它们，然后安装库和相应的命令。
- 如果指定-u，go get将确保它访问的所有包(包括它的依赖)更新到最新版本，然后再构建和安装。如果没有-u，已经存在于本地的包不会更新。
- -d，如果你想clone一个代码仓库到GOPATH里面，跳过编译和安装环节，使用-d参数。这样它只会下载包并且在编译和安装之前停止。

10.7.3 包的构建

go build编译每一个命令行参数中的包。

- 如果包是一个库，结果会被舍弃；对于没有编译错误的包几乎不做检查；
- 如果包名为main，go build调用链接器在当前目录中创建可执行程序，可执行程序名字

取自包的导入路径的最后一段。

- 包可以通过目录来指定，可以使用导入路径或者一个相对目录名，目录必须以.或..开头。
- 如果没有提供参数，会使用当前目录作为参数。
- 包也可以使用一个文件列表来指定。如果包名是main，可执行程序的名字来自第一个.go文件名的主体部分。

go run命令将构建和运行合并起来。

- 第一个不是以.go文件结尾的参数作为go可执行程序的参数列表的开始。
- 默认情况下，go build构建所有需要的包以及它们所有的依赖，然后丢弃除了最终可执行程序之外的所有编译后的代码。

go install 和 go build 非常相似，区别是前者会保存每个包的编译代码和命令。编译后的包保存在$GOPATH/pkg/ 目录中，可执行的命令保存在 $GOPATH/bin/ 目录中。这样go build 和 go install 对于没有改变的包和命令就不需要重新编译，从而使后续的构建更加快速。

- 习惯上，go build -i可以将包安装在独立于构建目标的地方。

10.7.4 包的文档化

第一个工具是go doc命令，它输出在命令上指定内容的声明和整个文档注释。可以是一个包，一个包成员，一个方法。

第二个工具是godoc，它提供相互链接的HTML页面服务，进而提供不少于go doc命令的信息。

- https://golang.org/pkg 的godoc服务器覆盖了标准库。
- https://godoc.org 的godoc服务器提供了数千个可搜索的开源包索引。
- 如果想要浏览自己的包，可以在自己的工作空间中运行一个godoc实例。
  执行 godoc -http :8000，之后在浏览器中访问 http://localhost:8000/pkg
  加上-analysis=type和-analysis=pointer标记使文档内容丰富，同时提供源代码的高级静态分析结果。

10.7.5 内部包

导入路径中包含路径片段internal的情况。

10.7.6 包的查询

go list 命令上报可以包的信息。

- 参数可以包含 … 通配符，用于匹配导入路径中的任意字串
- -json 选项，使 go list 输出包的完整记录
- -f 选项

11. 测试

2019年7月3日 星期三　　下午2:15

11.1 go test 工具

2019年7月3日 星期三　　下午2:17

在一个包目录中，以 _test.go 结尾的文件不是 go build 命令编译的目标，而是 go test 编译的目标。

*_test.go文件中，三种函数：

- 功能测试函数：以 Test 前缀命名的函数，用来检测一些程序逻辑的正确性。go test 运行测试函数，并且报告结果是 PASS 还是 FAIL。
- 基准测试函数：以 Benchmark 开头的函数，用来测试某些操作的性能，go test 汇报操作的平均执行时间。
- 示例函数：以 Example 开头的函数，用于提供机器检查过的文档。

go test 工具扫描 *_test.go 文件来寻找特殊函数，并生成一个临时的 main 包来调用它们，然后编译运行，并汇报结果，最后清空临时文件。

1. 测试文件名必须是_test.go结尾的，这样在执行go test的时候才会执行到相应的代码
2. 你必须import testing这个包
3. 所有的测试用例函数必须是Test开头
4. 测试用例会按照源代码中写的顺序依次执行
5. 测试函数TestXxx(t *testing.T)只有一个参数 t， 可以用 t 记录错误或者是测试状态
6. 测试函数的格式：**func TestXxx (t *testing.T)**，Xxx部分可以为任意的字母数字的组合，但是首字母不能是小写字母[a-z]，例如Testintdiv是错误的函数名。
7. 函数中通过调用testing.T的Error, Errorf, FailNow, Fatal, FatalIf方法，说明测试不通过，调用Log方法用来记录测试的信息。

11.2 Test 函数

2019年7月3日 星期三　　下午2:29

每一个测试文件必须导入testing包。

```
func TestName(t *testing.T) {
        //...
}
```

功能测试函数必须以 Test 开头，可选的后缀名称必须以大写字母开头。

go test(或 go build)命令在不指定包参数的情况下，以当前目录所在的包为参数。
-v　可以输出每个测试用例的名称和执行时间。
-run 参数是一个正则表达式，它可以使得 go test 只运行那些测试函数名称匹配给定模式的函数。
./... 表示运行当前文件夹以及所有子文件夹中所有的测试用例。

常用命令	意义
go test .	运行当前目录下所有测试文件的测试函数，必须当前路径下有测试文件，子文件夹里的测试文件不会检测到
go test ./...	遍历运行当前目录下**所有子文件夹内**的测试函数
go test filename_test.go	运行当前目录下某个测试文件里的所有测试函数
go test -run TestFuncName	运行当前目录下某个特定测试函数
go test ./service -run TestFuncName	运行指定路径./service里的某个特定测试函数

常用标记	作用
-v	显示通过的详细测试信息，默认只显示错误信息以及通过的概要
-c	生成用于运行测试的可执行文件，但不执行它。这个可执行文件会被命名为"pkg.test"，其中的"pkg"即为被测试代码包的导入路径的最后一个元素的名称。
-i	只是安装测试所依赖的包，不会编译执行
-o	指定编译生成的可执行文件名

测试错误时，可以使用 t.Fatal 或 t.Fatalf 来终止测试。
测试输出的错误消息一般格式是 "f(x)=y,want z"

11.2.1 随机测试

11.2.2 测试命令

包名 main 一般会产生可执行文件，但是也可以当作库来导入。

- 测试代码和产品代码放在一个包里面。尽管包的名称叫做main，并且里面的main函数，但是在测试过程中，这个包当作库来测试。

11.2.3 白盒测试

黑盒测试假设测试者对包的了解仅通过公开的API和文档，而包的内部逻辑则是不透明的。相反，白盒测试可以访问包的内部函数和数据结构。

11.3 覆盖率

2019年7月11日 星期四　　上午10:47

https://blog.golang.org/cover#TOC_5.

gocover位于：$GOROOT/pkg/tool/${GOOS}_${GOARCH}
Eg: /usr/local/go/pkg/tool/linux_amd64/cover

go tool cover //覆盖率工具的使用方法

go test -cover //输出汇总信息
go test -coverprofile=c.out //将汇总信息保存到c.out
go tool cover -html=c.out //生成html的报告，并在浏览器中打开

标记名称	使用示例	说明
-cover	-cover	启用测试覆盖率分析
-covermode	-covermode=set	自动添加**-cover**标记并设置不同的测试覆盖率统计模式，支持的模式共有以下3个。 **set**：只记录语句是否被执行过 **count**: 记录语句被执行的次数 **atomic**: 记录语句被执行的次数，并保证在并发执行时也能正确计数，但性能会受到一定的影响 这几个模式不可以被同时使用，在默认情况下，测试覆盖率的统计模式是**set**
-coverpkg	-coverpkg bufio,net	自动添加**-cover**标记并对该标记后罗列的代码包中的程序进行测试覆盖率统计。 在默认情况下，测试运行程序会只对被直接测试的代码包中的程序进行统计。 该标记意味着在测试中被间接使用到的其他代码包中的程序也可以被统计。 另外，代码包需要由它的导入路径指定，且多个导入路径之间以逗号”，”分隔。
-coverprofile	-coverprofile cover.out	自动添加**-cover**标记并把所有通过的测试的覆盖率的概要写入指定的文件中

12. 反射

2019年6月28日 星期五　　下午1:53

12.2 reflect.Type 和 reflect.Value

2019年6月28日 星期五　　下午1:55

反射功能由reflect包提供，它定义了两个重要的类型：Type和Value。

Type表示go语言的一个类型，它是一个有很多方法的接口，这些方法可以用来识别类型以及透视类型的组成部分，比如一个结构体的各个字段或者一个函数的各个参数。

reflect.Typeof 函数接收任何的interface{}参数，并且把接口中的动态类型以reflect.Type形式返回。

- reflect.Typeof 返回一个接口值对应的动态类型，所以它返回的总是具体类型。
- fmt.Printf 提供了%T格式，输出一个接口值的动态类型，内部实现就使用了reflect.Typeof。

reflect.Value可以包含一个任意类型的值。
reflect.Valueof 函数接收任何的interface{}参数，并且把接口中的动态值以reflect.Value形式返回。
与reflect.Typeof类似，reflect.Valueof 的返回值也都是具体值。
调用value的Type方法会把它的类型以reflect.Type的方式返回。

reflect.Valueof 的逆操作是reflect.Value.Interface方法。它返回一个interface{}接口值，与reflect.Value包含同一个具体值。
reflect.Value的kind方法可以区分不同的类型。

13. 低级编程

2019年6月10日 21:28

包unsafe是由编译器实现的。它提供了对语言内置特性的访问功能，而这些特性一般是不可见的，因为它们暴露了GO详细的内存布局。
包unsafe广泛使用在和操作系统交互的低级包(runtime,os,syscall,net)中，但是普通程序从来不需要使用它。

13.1 unsafe.Sizeof, Alignof, Offsetof

2019年6月11日 9:11

函数unsafe.Sizeof报告传递给它的参数在内存中占用的字节长度，这个参数可以是任意类型的表达式，不会计算表达式。
unsafe.Sizeof调用返回一个uintptr类型的常量表达式。

函数unsafe.Alignof报告它参数类型所要求的对齐方式。它的参数可以是任意类型的表达式，并且返回一个常量。

函数unsafe.Offsetof计算成员f相对于结构体x起始地址的偏移值。

13.2 unsafe.Pointer

2019年6月11日 10:02

unsafe.Pointer类型是一种特殊类型的指针，可以存储任何变量的地址。

- 无法间接通过unsafe.Pointer变量来使用*p，因为我们不知道这个表达式的具体类型。
- 和普通指针一样，unsafe.Pointer类型的指针是可以比较的并且可以和nil比较。
  nil是指针类型的零值。
- 一个普通的指针*T可以转换为unsafe.Pointer类型的指针，一个unsafe.Pointer类型的指针也可以转换为普通指针，而且可以不必和原来的类型*T相同。
  //通常使用unsafe.Pointer进行类型转化可以让我们将任意值写入内存中，并因此破坏类型系统。
- unsafe.Pointer类型也可以转换为uintptr类型。uintptr类型保存了指针所指向地址的数值，这就可以让我们对地址进行数值计算。
  //uintptr类型是一个足够大的无符号整型，可以用来表示任何地址。(是一个数值)
  //从uintptr到unsafe.Pointer的转换也会破坏类型系统，因为并不是所有的数值都是合法的内存地址。

很多unsafe.Pointer类型的值都是从普通指针到原始内存地址以及再从内存地址到普通指针进行转换的中间值。

```
var x struct {
        a bool
        b int16
        c []int
}

pb := (*int16)(unsafe.Pointer(
        uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b))) //等价于 pb := &x.b
*pb = 42
```

1. 不要尝试引入uintptr类型的临时变量来破坏整行代码
   ```
   tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
   pb := (*int16)(unsafe.Pointer(tmp))
   ```
   原因：一些垃圾回收器在内存中把变量移来移去以减少内存碎片或者为了进行薄记工作。这种类型的垃圾回收器称为移动的垃圾回收器。
   - 当一个变量在内存中移动后，该变量所指向旧地址的所有指针都需要更新以指向新地址。
     从垃圾回收器的角度看，unsafe.Pointer是一个变量的指针，当变量移动的时候，它的值也需要改变；而uintptr仅仅是一个数值，它的值是不会变的。
     上述错误代码使得垃圾回收器无法通过非指针变量tmp了解它背后的指针。
     当第二条语句执行的时候，变量x可能在内存中已经移动了，tmp中的值就不是变量x.b的地址了。

2. 错误实例2：
   ```
   pT := uintptr(unsafe.Pointer(new(T)))
   ```
   在这条语句执行之后，将没有指针指向new创建的变量。垃圾回收器将会在语句执行结束后回收内存，在这之后，pT存储的是变量的旧地址，不过这个时候这个地址对应的已经不是那个变量了。

13.4 使用cgo调用c代码

2019年6月11日 11:00

cgo是用来为C函数创建GO绑定的工具。诸如此类的工具都叫作外部函数接口(FFI)，cgo不是go程序唯一的工具。SWIG是另一种工具，它提供了更加复杂的特性用来集成C++的类。

- 如果C库很小，可以使用纯go语言来移植它；
- 如果性能对我们不是很关键，最好使用包os/exec以辅助子进程的方式来调用C程序；
- 仅当需要使用拥有有限C API并且复杂的、性能关键的库时，使用cgo来把它们包装成GO语言的绑定才有意义。

基础

2019年6月5日 19:17

1.++

go只有a++,没有++a

2.func type

Func type指代了一系列的funcs，这些funcs具有相同的参数和返回值类型。

Eg:

type Greeting func(name string) string

//将func(name string) string用Greeting指代。

3.log.fatal 和 panic

https://www.jianshu.com/p/f85ecae6e7df

4.[]byte和string转换

string 不能直接和byte数组转换，string可以和byte的切片转换

- string 转为[]byte

```
var str string = "test"
var data []byte = []byte(str)
```

- byte转为string

```
var data [10]byte
byte[0] = 'T'
byte[1] = 'E'
var str string = string(data[:])
```

5.reflect

https://lihaoquan.me/2018/5/4/reflect-action.html

Linux shell

2019年8月20日 星期二　　上午11:56

https://blog.csdn.net/qq_36874881/article/details/78234005

1.执行shell时设置timeout。

```
func callShell(shell string, args []string) error {
	ctx, cancel := context.WithTimeout(context.Background(), newFansProcTimeOut)
	defer cancel()

	cmd := exec.CommandContext(ctx, shell, args...)
	cmd.Stderr = os.Stderr
	cmd.Stdout = os.Stdout
	return cmd.Run()
}
```

*cmd.Start()*非阻塞；*cmd.Wait()*阻塞。

//也可以在shell中用timeout命令设置超时时间，根据*cmd.run()*的返回值判断shell是否超时。

位运算

2019年9月25日 星期三　　下午4:11

https://www.cnblogs.com/piperck/p/6139369.html

&　位运算 AND
|　位运算 OR
^　位运算 XOR
&^　位清空 (AND NOT)
<<　左移
>>　右移

ˆ(XOR) 在go语言中XOR是作为二元运算符存在的：

- 但是如果是作为一元运算符出现，他的意思是按位取反。
- 作为二元运算符则是，不进位加法计算，也就是异或计算。

Interface{}

2019年9月4日 星期三　　上午10:32

1.Can I convert a []T to an []interface{}?

官方解释：不可以

https://golang.org/doc/faq#convert_slice_of_interface

Not directly. It is disallowed by the language specification because the two types do not have the same representation in memory. It is necessary to copy the elements individually to the destination slice. This example converts a slice of int to a slice of interface{}:

```
t := []int{1, 2, 3, 4}
s := make([]interface{}, len(t))
for i, v := range t {
    s[i] = v
}
```

- []interface{} 并不是一个interface，它是一个slice,只是slice 中的元素是interface
- []interface{} 类型的内存大小是在编译期间就确定的(N*2),而其他切片类型的大小则为 N * sizeof(MyType),因此不发快速的将类型[]MyType分配给 []interface{}。

Go 允许不带任何方法的 interface 即interface{}, 这种类型的 interface 叫 empty interface。所有类型都实现了 empty interface, 因为任何一种类型至少实现了 0 个方法。

Init 函数

2019年7月16日 星期二　　下午3:25

1. init函数的特征：
   - init函数用于包的初始化，如初始化包中的变量，这个初始化在package xxx的时候完成，也就是在main之前完成；
   - 每个包可以拥有多个init函数， 每个包的源文件也可以拥有多个init函数；
   - 同一个包中多个init函数的执行顺序是没有明确定义的（编程时要注意程序不要依赖这个执行顺序），不同包的init函数是根据包导入的依赖关系决定的；
   - init函数不能被其他函数调用，其实在main函数之前自动执行的。init函数没有输入参数、返回值。

2. Golang 程序初始化

golang程序初始化先于main函数执行，由runtime进行初始化，初始化顺序如下：

- 初始化导入的包（包的初始化顺序并不是按导入顺序（“从上到下”）执行的，runtime需要解析包依赖关系，没有依赖的包最先初始化，与变量初始化依赖关系类似，参见golang变量的初始化）；
- 初始化包作用域的变量（该作用域的变量的初始化也并非按照“从上到下、从左到右”的顺序，runtime解析变量依赖关系，没有依赖的变量最先初始化，参见golang变量的初始化）；
- 执行包的init函数；

fmt

2019年8月1日 星期四　　下午7:44

https://golang.org/pkg/fmt/

%v　　the value in a default format
　　when printing structs, the plus flag (%+v) adds field names
%#v　a Go-syntax representation of the value
%T　　a Go-syntax representation of the type of the value
%%　　a literal percent sign; consumes no value

The default format for %v is:
bool:　　%t
int, int8 etc.:　%d
uint, uint8 etc.:　%d, %#x if printed with %#v
float32, complex64, etc: %g
string:　　%s
chan:　　%p
pointer:　　%p

Slice:
%p　　address of 0th element in base 16 notation, with leading 0x
Pointer:
%p　　base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers,
formatting the value exactly as if it were an integer.

sync

2019年7月23日 星期二　　下午7:31

1.Once
sync.Once.Do(f func())是一个挺有趣的东西,能保证once只执行一次，无论你是否更换once.Do(xx)这里的方法,这个sync.Once块只会执行一次。

Eg:
```
var once sync.Once

func onces() {
	fmt.Println("onces")
}
func onced() {
	fmt.Println("onced")
}

once.Do(onces)
once.Do(onced)
//整个程序，只会执行onces()方法一次,onced()方法是不会被执行的。
```

2.Pool
Pool是一个临时对象的集合，它可以单独保存和回收。
Pool中的任何项可以在任何时间在没有通知的情况下自动删除。
pool可以被多个goroutine同时安全地使用。
Pool用于存储那些被分配了但是没有被使用，而未来可能会使用的值，以减小垃圾回收的压力。

sync.Pool有两个公开的方法：
一个是Get，从池中获取一个interface{}类型的值
一个是Put，把一个interface{}类型的值放置于池中

创建的时候可以指定一个New函数，获取对象的时候如果在pool里面找不到缓存的对象将会使用指定的new函数创建一个返回，如果没有new函数则返回nil。

pool包在init的时候注册了一个poolCleanup函数，它会清除所有的pool里面的所有缓存的对象，该函数注册进去之后会在每次gc之前都会调用，因此sync.Pool缓存的期限只是两次gc之间这段时间。

3.WaitGroup
WaitGroup能够一直等到所有的goroutine执行完成，并且阻塞主线程的执行，直到所有的goroutine执行完成。

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines

have finished.

sync.WaitGroup只有3个方法，Add()，Done()，Wait()。

其中Done()是Add(-1)的别名。简单的来说，使用Add()添加计数，Done()减掉一个计数。计数不为0, 阻塞Wait()的运行。

gc

2019年7月23日 星期二　　下午8:03

https://studygolang.com/articles/7366