# 简介

## 参考资料

<https://studygolang.com/>

<http://www.runoob.com/go/go-tutorial.html>

<https://gobyexample.com/>

## 版本

Go.1.10.1

## 如何卸载

### 参考资料

<http://docs.studygolang.com/doc/install#uninstall>

### 详细步骤

#### 删除go安装目录

Linux上一般是 /usr/local/go。

#### 删除PATH环境变量

Linux上是修改 /etc/profile或者$HOME/.profile。

Windows上是修改环境变量PATH。

## 如何安装

### 安装二进制文件

#### 参考资料

<http://docs.studygolang.com/doc/install>

#### 详细步骤

##### 解压

sudo tar -C /usr/local/ -xzf ~/Downloads/go1.10.1.linux-amd64.tar.gz

##### 添加到PATH

在/etc/profile 或者 /etc/profile.d/my.sh 或者 $HOME/.profile中

export PATH=$PATH:/usr/local/go/bin

然后通过source或一个点执行sh。

##### 安装到其他目录

安装到其他目录，必须设置GOROOT。

export GOROOT=go目录

export PATH=$PATH:$GOROOT/bin

### 从源代码安装

#### 参考资料

<http://docs.studygolang.com/doc/install/source>

## 重要环境变量

### 参考资料

<https://www.cnblogs.com/rookie404/p/6358596.html>

<https://blog.csdn.net/defonds/article/details/50538077>

### GOROOT

Go安装根目录。

Go默认安装到：Linux下是/usr/local/go ，Windows下是C:\Go。

### GOPATH

Go工作目录。

默认的go工作空间目录是$HOME/go。

## 代码的组织

### 参考资料

<http://docscn.studygolang.com/doc/code.html>

### 工作空间

#### 环境变量

[GOPATH](#_GOPATH)

#### src

目录包含Go的源文件，它们被组织成包（每个目录都对应一个包），

#### pkg

目录包含包对象，

#### bin

目录包含可执行命令。

## 注释

### 参考资料

<http://docscn.studygolang.com/doc/effective_go.html>

### 包注释

### 顶级声明注释

# 模块

## 参考资料

<https://studygolang.com/articles/7165>

## 包

### 如何定义包

package network

所有的包名都应该使用小写字母。

### 包名和文件名的关系

### package main

package main表示一个可独立执行的程序。

程序运行的入口是包 `main`。

每个 Go 应用程序都包含一个名为 main 的包。

## 导入包

### 定义

一个 Go 程序是通过 import 关键字将一组包链接在一起。

Import “system”

### 注意

如果你导入了一个包却没有使用它，则会在构建程序时引发错误，如 imported and not used: os。

### 示例代码

import "fmt"

import "os"

或：

import "fmt"; import "os"

但是还有更短且更优雅的方法（被称为因式分解关键字，该方法同样适用于 const、var 和 type 的声明或定义）：

import (

"fmt"

"os"

)

它甚至还可以更短的形式，但使用 gofmt 后将会被强制换行：

import ("fmt"; "os")

## 可见规则

当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的 代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）

标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 private ）。

# 变量与类型

## 变量

### 定义

var 变量名 类型

`var` 语句可以定义在包或函数级别。

### 变量初始化

变量定义可以包含初始值，每个变量对应一个。

如果初始化是使用表达式，则可以省略类型；变量从初始值中获得类型。

var i, j int = 1, 2

var c, python, java = true, false, "no!"

### 短声明变量

在函数中，`:=` 简洁赋值语句在明确类型的地方，可以用于替代 var 定义。

函数外的每个语句都必须以关键字开始（`var`、`func`、等等），`:=` 结构不能使用在函数外。

k := 1

### 零值

变量在定义时没有明确的初始化时会赋值为\_零值\_。

零值是：

数值类型为 `0`，

布尔类型为 `false`，

字符串为 `""`（空字符串）。

## 常量

### 定义

常量的定义与变量类似，只不过使用 const 关键字。

常量可以是字符、字符串、布尔或数字类型的值。

常量不能使用 := 语法定义。

### 数值常量

数值常量是高精度的 \_值\_。一个未指定类型的常量由上下文来决定其类型。

## 基本类型

### bool

### String

#### Int转成string

import “strconv”

strconv.Itoa(int)

#### Int64转成string

strconv.FormatInt(int64,10)

#### string到int

int,err:=strconv.Atoi(string)

#### string到int64

int64, err := strconv.ParseInt(string, 10, 64)

#### string 转为[]byte

var str string = "test"

var data []byte = []byte(str)

#### byte转为string

string 不能直接和byte数组转换

string可以和byte的切片转换

var data [10]byte

byte[0] = 'T'

byte[1] = 'E'

var str string = string(data[:])

### int int8 int16 int32 int64

### uint uint8 uint16 uint32 uint64 uintptr

### byte // uint8 的别名

### rune // int32 的别名 // 代表一个Unicode码

### float32 float64

### complex64 complex128

## 类型转换

在不同类型之间的项目赋值时需要显式转换。

表达式 T(v) 将值 v 转换为类型 `T`。

## 指针

### 定义

类型 \*T 是指向类型 T 的值的指针。其零值是 `nil`。

var p \*int

& 符号会生成一个指向其作用对象的指针。

i := 42

p = &i

\* 符号表示指针指向的底层的值。

fmt.Println(\*p) // 通过指针 p 读取 i

\*p = 21 // 通过指针 p 设置 i

## 结构体（struct）

### 定义

type 结构体名称 struct {

}

结构体字段使用点号来访问。

结构体文法表示通过结构体字段的值作为列表来新分配一个结构体。

使用 Name: 语法可以仅列出部分字段。（字段名的顺序无关。）

特殊的前缀 & 返回一个指向结构体的指针。

sC := Class{"one", 200, "beijing", nil}

sC1 := Class{

strClassName: "two",

nClassNum: 300,

}

sC1 := Class {

strClassName : “tow”,

nClassNum : 300}

## 数组

### 定义

类型 [n]T 是一个有 n 个类型为 T 的值的数组。

表达式

var a [10]int

定义变量 a 是一个有十个整数的数组。

数组的长度是其类型的一部分，因此数组不能改变大小。

## slice（切片）

### 定义

一个 slice 会指向一个序列的值，并且包含了长度信息。

[]T 是一个元素类型为 T 的 slice。

表达式 s[lo:hi]，表示从 lo 到 hi-1 的 slice 元素。

slice 的零值是 `nil`。一个 nil 的 slice 的长度和容量是 0。

### 构造slice

a := make([]int, 5) // len(a)=5

为了指定容量，可传递第三个参数到 `make`：

b := make([]int, 0, 5) // len(b)=0, cap(b)=5

### 向 slice 添加元素

向 slice 添加元素是一种常见的操作，因此 Go 提供了一个内建函数 `append`。 内建函数的文档对 append 有详细介绍。

func append(s []T, vs ...T) []T

append 的第一个参数 s 是一个类型为 T 的数组，其余类型为 T 的值将会添加到 slice。

append 的结果是一个包含原 slice 所有元素加上新添加的元素的 slice。

### range迭代

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

for i, v := range pow {

fmt.Printf("2\*\*%d = %d\n", i, v)

}

可以通过赋值给 \_ 来忽略序号。

for i := range pow {

pow[i] = 1 << uint(i)

}

for \_, value := range pow {

fmt.Printf("%d\n", value)

}

## map

### 定义

type Vertex struct {

Lat, Long float64

}

var m map[string]Vertex

m = make(map[string]Vertex)

### map 的文法

如果顶级的类型只有类型名的话，可以在文法的元素中省略键名。

var m = map[string]Vertex{

"Bell Labs": Vertex{

40.68433, -74.39967,

},

"Google": {

37.42202, -122.08408,

},

}

### 修改map

#### 在 map m 中插入或修改一个元素

m[key] = elem

#### 获得元素

elem = m[key]

#### 删除元素

delete(m, key)

#### 通过双赋值检测某个键存在

elem, ok = m[key]

如果 key 在 m 中，`ok` 为 true 。否则， ok 为 `false`，并且 elem 是 map 的元素类型的零值。

## 函数值

函数也是值。

## 分配

### new

new(T) 会为类型为 T 的新项分配已置零的内存空间， 并返回它的地址，也就是一个类型为 \*T 的值。用Go的术语来说，它返回一个指针， 该指针指向新分配的，类型为 T 的零值。

### make

内建函数 make(T, args) 的目的不同于 new(T)。它只用于创建切片、映射和信道，并返回类型为 T（而非 \*T）的一个已初始化 （而非置零）的值。

# 运算符

## 参考资料

<http://www.runoob.com/go/go-operators.html>

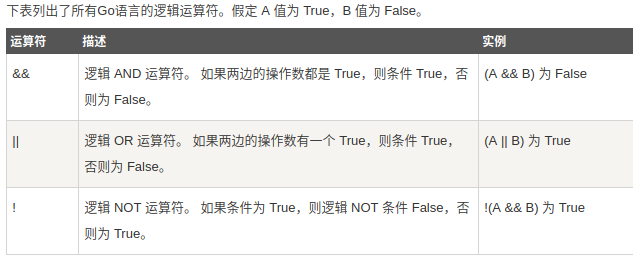
## 算术运算符



## 关系运算符



## 逻辑运算符



## 位运算符



## 赋值运算符



## 其他运算符



## 运算符优先级



# 函数

## 定义

func 函数名(参数名 类型，参数名 类型) 返回值类型 {

函数代码

}

## 参数

### 无参数

### 多个相同类型的参数

x int， y int，可以改写成 x, y int

## 返回值

### 多值返回

函数可以返回任意数量的返回值。

func swap(x, y string) (string, string) {

return y, x

}

### 命名返回值

Go 的返回值可以被命名，并且像变量那样使用。

返回值的名称应当具有一定的意义，可以作为文档使用。

没有参数的 return 语句返回结果的当前值。也就是`直接`返回。

func split(sum int) (x, y int) {

x = sum \* 4 / 9

y = sum - x

return

}

## defer

### 定义

defer 语句会延迟函数的执行直到上层函数返回。

延迟调用的参数会立刻生成，但是在上层函数返回前函数都不会被调用。

### defer 栈

延迟的函数调用被压入一个栈中。当函数返回时， 会按照后进先出的顺序调用被延迟的函数调用。

## 打印

### 函数

fmt.Printf("Hello %d\n", 23)

fmt.Fprint(os.Stdout, "Hello ", 23, "\n")

fmt.Println("Hello", 23)

fmt.Println(fmt.Sprint("Hello ", 23))

### 格式

通用的格式 %v（对应“值”）。

当打印结构体时，改进的格式 %+v 会为结构体的每个字段添上字段名，而另一种格式 %#v 将完全按照Go的语法打印值。

当遇到 string 或 []byte 值时， 可使用 %q 产生带引号的字符串；而格式 %#q 会尽可能使用反引号。

## 常用函数

### len(a)

### cap(a)

### append(a)

func append(slice []T, 元素 ...T) []T

x := []int{1,2,3}

x = append(x, 4, 5, 6)

fmt.Println(x)

x := []int{1,2,3}

y := []int{4,5,6}

x = append(x, y...)

fmt.Println(x)

# 方法与接口

## 方法

### 定义

Go 没有类。然而，仍然可以在结构体类型上定义方法。

方法接收者 出现在 func 关键字和方法名之间的参数中。

type Vertex struct {

X, Y float64

}

func (v \*Vertex) Abs() float64 {

return math.Sqrt(v.X\*v.X + v.Y\*v.Y)

}

func main() {

v := &Vertex{3, 4}

fmt.Println(v.Abs())

}

可以对包中的 任意 类型定义任意方法，而不仅仅是针对结构体。

但是，不能对来自其他包的类型或基础类型定义方法。

type MyFloat float64

func (f MyFloat) Abs() float64 {

if f < 0 {

return float64(-f)

}

return float64(f)

}

### 接收者为指针的方法

方法可以与命名类型或命名类型的指针关联。

## 接口

### 定义

接口类型是由一组方法定义的集合。

接口类型的值可以存放实现这些方法的任何值。

type Abser interface {

Abs() float64

}

### 隐式接口

类型通过实现那些方法来实现接口。 没有显式声明的必要；所以也就没有关键字“implements“。

隐式接口解藕了实现接口的包和定义接口的包：互不依赖。

type Reader interface {

Read(b []byte) (n int, err error)

}

type Writer interface {

Write(b []byte) (n int, err error)

}

type ReadWriter interface {

Reader

Writer

}

## 错误

### 定义

Go 程序使用 error 值来表示错误状态。

与 fmt.Stringer 类似，`error` 类型是一个内建接口：

type error interface {

Error() string

}

error 为 nil 时表示成功；非 nil 的 error 表示错误。

## Readers

io 包指定了 io.Reader 接口， 它表示从数据流结尾读取。

Go 标准库包含了这个接口的许多实现， 包括文件、网络连接、压缩、加密等等。

io.Reader 接口有一个 Read 方法：

func (T) Read(b []byte) (n int, err error)

Read 用数据填充指定的字节 slice，并且返回填充的字节数和错误信息。 在遇到数据流结尾时，返回 io.EOF 错误。

## Web服务器

### 定义

包 http 通过任何实现了 http.Handler 的值来响应 HTTP 请求：

package http

type Handler interface {

ServeHTTP(w ResponseWriter, r \*Request)

}

## 排序

实现了 sort.Interface 接口的集合就可通过 sort 包中的例程进行排序。该接口包括 Len()、Less(i, j int) bool 以及 Swap(i, j int)。

## 接口转换与类型断言

### 类型选择

var value interface{} // 调用者提供的值。

switch str := value.(type) {

case string:

return str

case Stringer:

return str.String()

}

### 类型断言

value.(typeName)

str, ok := value.(string)

if ok {

fmt.Printf("字符串值为 %q\n", str)

} else {

fmt.Printf("该值非字符串\n")

## 接口检查

### 类型断言

[类型断言](#_类型断言)

### 全局变量

var \_ json.Marshaler = (\*RawMessage)(nil)

## 内嵌

### 接口内嵌

### 结构体内嵌

# 流程控制语句

## 条件语句

### 示例代码

if x < 0 {

}

### if的便捷语句

跟 for 一样，`if` 语句可以在条件之前执行一个简单的语句。

由这个语句定义的变量的作用域仅在 if 范围之内。

if v := math.Pow(x, n); v < lim {

return v

} else {

fmt.Printf("%g >= %g\n", v, lim)

}

## switch

### 注意

除非以 fallthrough 语句结束，否则分支会自动终止。

### 示例代码

switch os := runtime.GOOS; os {

case "darwin":

fmt.Println("OS X.")

case "linux":

fmt.Println("Linux.")

default:

// freebsd, openbsd,

// plan9, windows...

fmt.Printf("%s.", os)

}

### 没有条件的 switch

没有条件的 switch 同 `switch true` 一样。

这一构造使得可以用更清晰的形式来编写长的 if-then-else 链。

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("Good morning!")

case t.Hour() < 17:

fmt.Println("Good afternoon.")

default:

fmt.Println("Good evening.")

}

## 循环语句

### for

#### 示例代码

sum := 0

for i := 0; i < 10; i++ {

sum += i

}

for sum < 10; {

sum += 1

}

for {

死循环

}

# 并发

## goroutine

### 定义

goroutine 是由 Go 运行时环境管理的轻量级线程。

go f(x, y, z)

## channel

### 定义

channel 是有类型的管道，可以用 channel 操作符 <- 对其发送或者接收值。

ch <- v // 将 v 送入 channel ch。

v := <-ch // 从 ch 接收，并且赋值给 v。

（“箭头”就是数据流的方向。）

和 map 与 slice 一样，channel 使用前必须创建：

ch := make(chan int)

默认情况下，在另一端准备好之前，发送和接收都会阻塞。

### 缓冲 channel

channel 可以是 \_带缓冲的\_。为 make 提供第二个参数作为缓冲长度来初始化一个缓冲 channel：

ch := make(chan int, 100)

### range 和 close

发送者可以 close 一个 channel 来表示再没有值会被发送了。

接收者可以通过赋值语句的第二参数来测试 channel 是否被关闭：当没有值可以接收并且 channel 已经被关闭，那么经过

v, ok := <-ch

之后 ok 会被设置为 `false`。

循环 `for i := range c` 会不断从 channel 接收值，直到它被关闭。

注意： 只有发送者才能关闭 channel，而不是接收者。向一个已经关闭的 channel 发送数据会引起 panic。 还要注意： channel 与文件不同；通常情况下无需关闭它们。只有在需要告诉接收者没有更多的数据的时候才有必要进行关闭，例如中断一个 `range`。

### select

select 语句使得一个 goroutine 在多个通讯操作上等待。

select 会阻塞，直到条件分支中的某个可以继续执行，这时就会执行那个条件分支。当多个都准备好的时候，会随机选择一个。

select {

case c <- x:

x, y = y, x+y

case <-quit:

fmt.Println("quit")

return

}

当 select 中的其他条件分支都没有准备好的时候，`default` 分支会被执行。

为了非阻塞的发送或者接收，可使用 default 分支

### channel阻塞超时处理

c := make(chan int)

o := make(chan bool)

go func() {

for {

select {

case i := <-c:

fmt.Println(i)

case <-time.After(time.Duration(3) \* time.Second): //设置超时时间为３ｓ，如果channel　3s钟没有响应，一直阻塞，则报告超时，进行超时处理．

fmt.Println("timeout")

o <- true

break

}

}

}()

<-o

## Sync

### 参考资料

<https://studygolang.com/articles/2027>

### 同步类型

type Cond

func NewCond(l Locker) \*Cond

func (c \*Cond) Broadcast()

func (c \*Cond) Signal()

func (c \*Cond) Wait()

type Locker

type Mutex

func (m \*Mutex) Lock()

func (m \*Mutex) Unlock()

type Once

func (o \*Once) Do(f func())

type Pool

func (p \*Pool) Get() interface{}

func (p \*Pool) Put(x interface{})

type RWMutex

func (rw \*RWMutex) Lock()

func (rw \*RWMutex) RLock()

func (rw \*RWMutex) RLocker() Locker

func (rw \*RWMutex) RUnlock()

func (rw \*RWMutex) Unlock()

type WaitGroup

func (wg \*WaitGroup) Add(delta int)

func (wg \*WaitGroup) Done()

func (wg \*WaitGroup) Wait()

### sync.WaitGroup的坑

#### 参考资料

<https://blog.csdn.net/u011622226/article/details/69588174>

#### 详细解释

把sync.WaitGroup的值传递过来，而非内存地址。

# 内存模型

## 参考资料

<http://docscn.studygolang.com/ref/mem>

# RPC

## 参考资料

<https://studygolang.com/articles/8497>

<https://legacy.gitbook.com/book/smallnest/go-rpc-programming-guide/details>

# 常见问题

## JSON处理

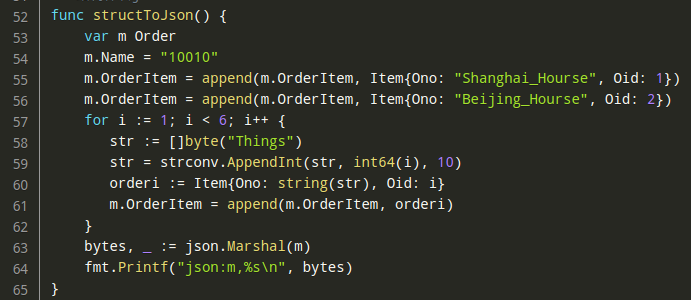
### 参考资料

<https://studygolang.com/articles/9179>

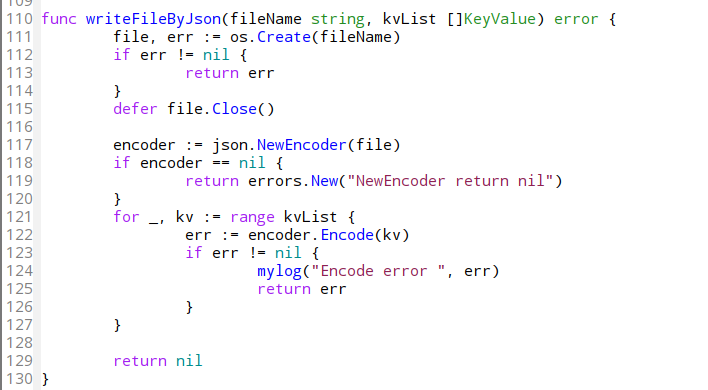
<https://blog.csdn.net/u011304970/article/details/70769949>

### 转JSON

#### json.Marshal



#### json.NewEncoder



### 从JSON转

#### json.Unmarshal



#### json.NewDecoder

