Go语言基础

> go语言的优点
>
> 1. 高性能、高并发
> 2. 语法简单、学习曲线平缓
> 3. 丰富的标准库完善的工具链
> 4. 静态链接
> 5. 快速编译
> 6. 跨平台
> 7. 垃圾回收

> 字节跳动为什么全面拥抱 Go 语言
>
> 1. 最初使用的 Python，由于性能问题换成了 Go
> 2. C++ 不太适合在线 Web 业务
> 3. 早期团队非 Java 背景
> 4. 性能比较好
> 5. 部署简单、学习成本低
> 6. 内部 RPC 和 HTTP 框架的推广

包和函数

```
package main  //声明本代码所属的包
import(
"fmt" //导入fmt(是format的缩写)包，使其可用
)
func main(){  // func 关键字用于声明函数声明一个名为main 的函数，fmt 包提供了用于格式化输入和输出的函数。
    fmt.Printin("hello world" ) //在屏幕上打印出“Hello,playground”
}   //这是 Go 语言唯一允许的大括号放置风格

```

package 关键字声明了代码所属的包，在本例中这个包的名字就是 main。所有用 Go编写的代码都会被组织成各式各样的包，并且每个包都对应一个单独的构想。

main 这一标识符具有特殊意义。当我们运行一个 Go程序的时候，它总是从 main 包的 main 函数开始运行。如果 main 不存在，那么 Go编译器将报告一个错误,因为它无法得知程序应该从何处开始执行。

每次用到被导入包中的某个函数时，我们都需要在函数的名字前面加上包的名字以及一个点号作为前缀。Go 的这一特性可以让用户在阅读代码的时候立即弄清楚各个函数分别来自哪个包。

计算

Go 跟其他编程语言一样，提供了+、-、*、/（取商）、%（取模：取余数）。

```go
//我的减重程序
package main
import "fmt"
//main 是所有程序的起始函数
func main(){
fmt.Print("My weight on the surface of Mars is ")
fmt.Print(149.0*0.3783)
fmt.Print(" lbs,and I would be ")
fmt.Print(41*365/687)
fmt.Print("years old.")
//或者为：fmt,Println("My weight on the surface of Mars is", 149.0*0.3783,
"lbs, and I would be",41*365.2425/687，"years old."),
```

打印出“My weight on the surface of Mars is 56.3667 lbs, and I wouldbe21.79758733624454 years old.”

变量

程序员有时候会把这种没有说明具体含义的字面数字称为魔数

var 变量名类型=表达式

var name string="zhangsan'

在函数内部，可以使用更简略的:方式声明并初始化变量。(短变量)只能用于声明局部变量，不能用于全局变量的声明。

变量必须先声明再赋值

Go 语言变量名由字母、数字、下划线组成，其中首个字符不能为数字。Go 语言中关键字和保留字都不能用作变量名。

```go
var 变量名称 type
var name string   //go语言也可以推断变量类型，即可以写成var name="word"
var age int
var isOk bool
```

```go
// 到达火星需要多长时间?
package main
import "fmt"
func main(){
const lightSpeed=299792//km/s  //两个新的关键字const和var，它们分别用于声明常量和变量。
var distance=56000000 // km    //变量必须先声明再赋值 distance=56000000会报错
fmt.Println(distance/lightSpeed，"seconds")  //打印出186 secends
distance =401000000
fmt.Println(distance/lightSpeed，"seconds")  //打印出1337 secends
}
```

注意 lightSpeed 常量是不能被修改的，尝试为其赋予新值将导致 Go 编译器报告错误:“无法对 lightSpeed 进行赋值”

注意 变量必须先声明后使用。如果尚未使用 var 关键字对变量进行声明，那么尝试向它赋值将导致 Go 报告错误，例如在前面的代码中执行 speed = 16 就会这样。这一限制有助于发现类似于“想要向 distance 赋值却键入了 distence”这样的问题。

一次声明多个变量

```
var distance= 56000000
var speed=100800
```

```
var (
   distance=56000000
   speed =100800
)

```

```
var distance,speed=56000000，100800
```

变量的初始化

Go语言在声明变量的时候，会自动对变量对应的内存区域进行初始化操作。每个变量会被初始化成其类型的默认值，例如: 整型和浮点型变量的默认值为0。 字符串变量的默认值为空字符串。 布尔型变量默认为 false。 切片、函数、指针变量的默认为 nil。

短变量声明

```
m1, m2,m3 :=10,20,30
```

匿名变量

在使用多重赋俏时，如果想要忽略某个值，可以使用匿名变量(anonymous variable)。 匿名变量用一个下划线 表示，例如:

```
func getlnfo()(int, string){
return 10,"张三”
func main(){
_,username := getlnfo() //忽略第一个数
first,_:=getlnfo() //忽略第二个数
fmt.Println(username)
fmt.Println(first)
}
```

//这个函数 `getInfo` 返回两个值：`10`（一个整数）和 `"张三"`（一个字符串）。函数的返回类型是 `(int, string)`，表示它返回一个整数和一个字符串。

函数的定义没有接收任何参数，直接返回这两个值。

匿名变量不占用命名空间，不会分配内存，所以匿名变量之间不存在重复声明注意事项:
1、函数外的每个语句都必须以关键字开始(var、const、func 等)
2、:不能使用在函数外。
3、 多用于占位，表示忽略值。

常量

```
const(
pi= 3.1415e=2.7182
)
const (
pi= 3.1415
e=2.7182
)
const(
n1=100
n2
n3
)
```

常量、变量的命名规则

1、变量名称必须由数字、字母、下划线组成
2、标识符开头不能是数字
3、标识符不能是保留字和关键字。
4、变量的名字是区分大小写的如: age 和 Age 是不同的变量。在实际的运用中,也建议,不要用一个单词大小写区分两个变量。

5、标识符(变量名称)一定要见名思意 :变量名称建议用名词，方法名称建议用动词

6、变量命名一般采用驼峰式，当遇到特有名词(缩写或简称，如 DNS)的时候，特有名词根据是否私有全部大写或小写。

代码风格

1、代码每一行结束后不用写分号(;)
2、运算符左右建议各加一个空格
3、Go 语言程序员推荐使用驼峰式命名：当名字有几个单词组成的时优先使用大小写分隔
4、强制的代码风格：左括号必须紧接着语句不换行，这个特性刚开始会使开发者不习惯，但随着对Go语言的不断熟悉，就会发现风格统一让大家在阅读代码时把注意力集中到了解决问题上，而不是代码风格上
5、go fmt 主要用于格式化文档，让所有人的代码风格保持一致

随机数字

```
//随机数字
package main
import (
"fmt "
"math/rand"  //rand 包的导入路径为 math/rand
)
func main(){
var num=rand.Intn(10)+1   //Intn生成0~9的伪随机数（它们并非真正随机，只是看上去或多或少像是随机的而已。）    注意 如果我们在写代码的时候忘记对伪随机数执行加一操作，那么程序将返回一个0~9的数字而不是我们想要的1~10 的数字。这是典型的“差一错误”(off-by-one error)的例子，这种错误是典型的计算机编程错误之一。
fmt.Println(num)
num =rand.Intn(10)+1
fmt.Println(num)
```

通过 Printf函数和格式化变量v,用户可以将值放置到被显示文本的任意位置上。

```
fmt.Printf("%-15v $%4v\n", "SpaceX", 94)
fmt.Printf("%-15v $%4v\n","Virgin Galactic", 100)
//SpaceX          $ 94
//Virgin Galactic $ 100
```

对 Go 来说，true 是唯一的真值,而false 则是唯一的假值。

比较

比较运算符：既可以比较文本，又可以比较数值。

符号	含义	符号	含义
==	相等	<=	小于等于
！=	不相等	>	大于
<	小于	>=	大于等于

```
//比较数值
fmt,Println("There is a sign near the entrance that reads 'No Minors'.")
var age = 4l
var minor=age<18  //从右向左看，age<18 为false
fmt.Printf("At age %v,am I a minor? %v\n",age, minor)
//输出There is a sign near the entrance that reads 'No Minors'At age 4l,am I a minor? false
```

Go 只提供了一个相等运算符，并且它不允许直接将文本和数值进行比较。

if else

```
package main
import "fmt"
func main(){
var command="go east"
if command =="go east"{   //检查命令是否为“go east”
  fmt .Println("You head further up the mountain.")
}else if command =="go inside"{
//在第一次检查为假之后，检查命令是否为“go inside”
  fmt.Println("You enter the cave where you live out the rest of your life.")
}else {
//如果前两次检查都为假，
  fmt.Println("Didn't quite get that.")
}
//那么执行第三个分支
}
```

逻辑运算符

在 Go中，逻辑运算符||代表“逻辑或”，而逻辑运算符&&则代表“逻辑与”。这些逻辑运算符可以一次检查多个条件。

逻辑或:当a、b两个值中至少有一个为true 时，allb为true

逻辑与:当且仅当a、b两个值都为 true 时，a && b为 true

```
//能够被4整除但是不能被100整除的年份是闰年或者可以被 400整除的年份是闰年。
fmt.Println("The year is 2100,should you leap?")
var year =2100
var leap=year%400==0||(year%4==0 && year%100 != 0)   //%取余
if leap {
  fmt.Println("Look before you leap!")}else {
  fmt.Println("Keep your feet on the ground." )
  //输出The year is 2100,should you leap?
  //输出Keep your feet on the ground.
```

跟大多数编程语言一样，Go也采用了短路逻辑:如果位于|1运算符之前的第一个条件为真，那么位于!运算符之后的条件就可以被忽略，没有必要再对其进行求值。具体到代码清单 3-4中的例子，当给定年份可以被400整除时，程序就不必再进行后续的判断了。&&运算符的行为与1运算符正好相反:只有在两个条件都为真的情况下，运算结果才为真。对于代码清单3-4中的例子，如果给定年份无法被4整除，那么程序就不会对后续条件进行求值。

逻辑非运算符!可以将一个布尔值从false 变为true，或者将true 变为 false。

```
var haveTorch =true //有火把
var litTorch =false //火把没有点亮
if !haveTorch |||litTorch { //如果 false ||true (如果有火把或者没点亮)
  fmt.Println("Nothing to see here.")
}
```

循环

switch

switch语句，它可以将单个值和多个值进行比较。除文本以外，switch语句还可以接受数值作为条件。

```
fmt.Println("There is a cavern entrance here and a path to the east.")
var command ="go inside"
switch command { //将命令和给定的多个分支进行比较
case "go east":
fmt.Println("You head further up the mountain.")
case "enter cave","go inside": //使用逗号分隔多个可选值
fmt.Println("You find yourself in a dimly lit cavern.")
case "read sian":
fmt.Println("The sign reads No Minors'.")
default:
fmt.Println("Didn't quite get that.")
//输出There is a cavern entrance here and a path to the east.
//You find yourself in a dimly lit cavern.
```

switch的另一种用法

在 Go 语言中，fallthrough是 switch语句的一部分，用来控制程序从当前 case 继续执行到下一个 case。默认情况下，Go 中的 switch 语句会在匹配到一个 case 后立即终止，而不会自动跳转到下一个 case。但是，当你使用 fallthrough 时，它会强制程序继续执行下一个 case 语句块的代码，而不管下一个 case 的条件是否满足。

```
var room ="lake"
switch{ //比较表达式将被放置到单独的分支里面
case room==cave":
fmt,Println("You find yourself in a dimly lit cavern.")
case room =="lake":
  fmt.Println("The ice seems solid enough.")
  fallthrough //下降至下一分支
case room =="underwater":
  fmt.Println("The water is freezing cold.")

//The ice seems solid enough.
//The water is freezing cold.
```

range

函数

```
func 函数名(参数)(返回值){
      函数体
}
func intsum(x, y int) int {
return x+y //上面的代码中，intsum函数有两个参数，这两个参数的类型均为int，因此可以省略x的类型，因为y后面有类型说明，x参数也是该类型。
```

函数名:由字母、数字、下划线组成。但函数名的第一个字母不能是数字。在同一个包内，函数名也称不能重名(包的概念详见后文)

参数:参数由参数变量和参数变量的类型组成，多个参数之间使用,分隔。
返回值:返回值由返回值变量和其变量类型组成，也可以只写返回值的类型，多个返回值必须用()包裹，并用,分隔。
函数体:实现指定功能的代码块。

```
func intsum(x int, y int) int {
  return x+y
}
```

函数多返回值

```
func calc(x, y int)(int, int){
  sum:=X+y
  sub :=x-y
  return sum,sub
}
```

返回值命名

函数定义时可以给返回值命名，并在函数体中直接使用这些变量，最后通过 return 关键字返回。

```
func calc(x,y int)(sum, sub int){
  sum =X+y
  sub=x-y
  return
}
```

全局变量

全局变量是定义在函数外部的变量，它在程序整个运行周期内都有效。 在函数中可以访问到全局变量。

```
package main
import "fmt"
//定义全局变量 num
var num int64 = 10
func testGlobalVar(){
fmt.Printf("num=%d\n", num)//函数中可以访问全局变量num
func main(){
testGlobalVar()//num=10
}
```

局部变量

局部变量是函数内部定义的变量，函数内定义的变量无法在该函数外使用。

例如下面的示例代码 main 函数中无法使用 testlocalVar 函数中定义的变量 x:

```
func testLocalVar(){
//定义一个函数局部变量x,仅在该函数内生效
var x int64 = 100
fmt.Printf("x=%d\n",x)
func main(){
testLocalVar()
fmt.Println(x)// 此时无法使用变量x
```

如果局部变量和全局变量重名，优先访问局部变量

```go
package main
import "fmt'
//定义全局变量num
var num int64 = 10
func testNum(){
num := 100
fmt.Printf("num=%d\n", num)// 函数中优先使用局部变量
func main(){
testNum()
}// num=100
```

定义函数类型

我们可以使用 type 关键字来定义一个函数类型，具体格式如下:

```go
type calculation func(int, int)int
```

上面语句定义了一个 calculation 类型，它是一种函数类型，这种函数接收两个 int 类型的参数并且返回一个 int 类型的返回值。
简单来说，凡是满足这个条件的函数都是 calculation 类型的函数，例如下面的 add 和 sub 是 calculation 类型函数。

```go
func add(x,y int) int {
return x+y
}
func sub(x, y int)int {
return x-y
}
var c calculation
c= add //add 和 sub 都能赋值给 calculation 类型的变量。
```

我们可以声明函数类型的变量并且为该变量赋值:

```go
func main(){
var c calculation //声明一个calculation 类型的变量c
c= add //把add 赋值给c
fmt.Printf("type of c:%T\n",c) // type of c:main.calculation
fmt.PrintIn(c(1,2))// 像调用 add 一样调用c
f := add // 将函数 add 赋值给变量f
f1fmt.Printf("type of f:%T\n",f)// type of f:func(int, int) int
fmt.PrintIn(f(10, 20))// 像调用add 一样调用
```

指针

结构体

结构体方法

错误处理

字符串操作

字符串格式化

JSON处理

时间处理

数字解析

进程信息

Print、Printin、Printf

println	printf(格式化输出)	prin
有空格	利用占位符（%d、%f）	无空格
自动换行		不自动换行

注释

ctrl+/ 可以快速的注释

```
//一次输入多个值的时候 Println 中间有空格 Print 没有
fmt.Println("go","python","php","javascript") // go python php javascript
fmt.Print("go","python","php","javascript")//gopythonphpjavascript
//Println 会自动换行，Print 不会
```

```go
package main
import "fmt'
func main() {
fmt.Println("hello")
fmt.Println("world")
// hello
// world
fmt.Print("hello")
fmt.Print("world")
// helloworld
}
```

数据类型

Go 语言中数据类型分为:基本数据类型和复合数据类型

基本数据类型有:整型、浮点型、布尔型、字符串
复合数据类型有:数组、切片、结构体、函数、map、通道(channel)、接口等。

整型分为以下两个大类:
有符号整形按长度分为:int8、int16、int32、int64

对应的无符号整型:uint8、uint16、uint32、uint64

类型	范围	占用空间	有无符号
int8	(-128 到 127) -2^7 到 2^7-1	1 个字节	有
int16	(-32768 到 32767) -2^15 到 2^15-1	2 个字节	有
int32	(-2147483648 到 2147483647) -2^31 到 2^31-1	4 个字节	有
int64	(-9223372036854775808 到 9223372036854775807) -2^63 到 2^63-1	8 个字节	有
uint8	(0 到 255) 0 到 2^8-1	1 个字节	无
uint16	(0 到 65535) 0 到 2^16-1	2 个字节	无
uint32	(0 到 4294967295) 0 到 2^32-1	4 个字节	无
uint64	(0 到 18446744073709551615) 0 到 2^64-1	8 个字节	无

特殊整型

类型	描述
uint	32 位操作系统上就是 uint32，64 位操作系统上就是 uint64
int	32 位操作系统上就是 int32，64 位操作系统上就是 int64
uintptr	无符号整型，用于存放一个指针

unsafe.Sizeof

unsafe.sizeof(n1)是 unsafe 包的一个所数，可以返回 n1 变量占用的字节数

布尔值

Go 语言中以 bool 类型进行声明布尔型数据，布尔型数据只有true(真)和 false(假)两个值。
注意:
布尔类型变量的默认值为 false。
Go 语言中不允许将整型强制转换为布尔型。
布尔型无法参与数值运算，也无法与其他类型进行转换。

字符串的常用操作

方法	介绍
len(str)	求长度
+或 fmt.Sprintf	拼接字符串
strings.Split	分割
strings.contains	判断是否包含
strings.HasPrefix,strings.HasSuffix	前缀/后缀判断
strings.Index(),strings.LastIndex()	子串出现的位置
strings.Join(a[]string, sep string)	join 操作

高质量代码

什么是高质量：编写的代码能够达到正确可靠、简洁清晰的目标可称之为高质量代码
各种边界条件是否考虑完备
异常情况处理，稳定性保证
易读易维护

代码格式

gofmt
Go 语言官方提供的工具，能自动格式化 Go 语言代码为官方统一风格常见IDE都支持方便的配置

goimportsRun gofmt也是 Go 语言官方提供的工具实际等于 gofmt 加上依赖包管理自动增删依赖的包引用、将依赖包按字母序排序并分类

注释应该解释代码作用

```
1 //Returns true if the table cannot hold any more entries
2 func IsTableFull()bool
```

函数名就解释了代码是做什么的了，不需要再写注释了。

注释应该解释代码如何做的

注释应该解释代码实现的原因

注释应该解释代码什么情况会出错

代码是最好的注释：有时更新了代码却没有更新注释。
注释应该提供代码未表达出的上下文信息。

注释

命名规范（变量）

简洁胜于冗长
缩略词全大写，但当其位于变量开头且不需要导出时，使用全小写例如使用 ServeHTTP 而不是 ServeHttp、使用 XMLHTTPRequest 或者 xmlHTTPRequest

变量距离其被使用的地方越远，则需要携带越多的上下文信息

全局变量在其名字中需要更多的上下文信息，使得在不同地方可以轻易辨认出其含义

i 和 index 的作用域范围仅限于 for 循环内部时index 的额外冗长几乎没有增加对于程序的理解

```
// Bad
for index :=;index<len(s);index++ {
  // do something
}
// Good
for i:=0;i<len(s); i++ {
// do something
}
```

http 包中创建服务的函数如何命名更好? 第一种
func Serve(l net.Listener, handler Handler) error

func ServeHTTP(l net.Listener, handler Handler) error

包的命名

只由小写字母组成。不包含大写字母和下划线等字符简短并包含一定的上下文信息。例如 schema、task 等不要与标准库同名。例如不要使用 sync 或者 strings。

以下规则尽量满足，以标准库包名为例
不使用常用变量名作为包名。例如使用 bufio 而不是 buf

使用单数而不是复数。例如使用 encoding 而不是 encodings

谨慎地使用缩写。例如使用 fmt 在不破坏上下文的情况下比 format 更加简短

```
package time
//A function returns the current local time.
// which one is better?func Now()Time

func NowTime() Time
// or
func Now() Time
// 使用
t:= time.Now( ) //这个好
t:= time.NowTime( ) //冗余

```

函数名为当前时间与包名意思一致，因为使用函数时包名与函数名捆绑。

控制流程

- 避免嵌套，保持正常流程清晰：如果两个分支中都包含return语句，则可以去除冗余的else

```
// Bad
if foo {
  return x
} else {
  return nil
// Good
if foo {
  return x
}
return nil
```

- 尽量保持正常代码路径为最小缩进：优先处理错误情况/特殊情况，尽早返回或继续循环来减少嵌套。

1. 最常见的正常流程的路径被嵌套在两个if 条件内
2. 成功的退出条件是 return nil，必须仔细匹配大括号来发现函数最后一行返回一个错误，需要追溯到匹配的左括号，才能了解何时会触发错误
3. 如果后续正常流程需要增加一步操作，调用新的函数，则又会增加一层嵌套

```
// Good
func OneFunc()error {
  if err := doSomething(); err != nil {
    return err
  }
  if err := doAnotherThing();err != nil{
    return err
  }
return nil // normal case
}
```

```go
// Bad
func OneFunc()error{
  err := doSomething()
  if err == nil {
    err := doAnotherThing( )
    if err == nil {
      return nil // normal case
    }
    return err
  }
  return err
}
```

错误和异常处理

- 简单错误：简单的错误指的是仅出现一次的错误，且在其他地方不需要捕获该错误。优先使用 errors.New 来创建匿名变量来直接表示简单错误。如果有格式化的需求，使用 fmt.Errorf
- 错误的 Wrap 和 Unwrap：错误的 Wrap 实际上是提供了一个 error 嵌套另一个error 的能力，从而生成一个 error 的跟踪链在 fmt.Errorf 中使用: %w 关键字来将一个错误关联至错误链中
- 错误判定：判定一个错误是否为特定错误，使用 errors.Is。不同于使用 ==，使用该方法可以判定错误链上的所有错误是否含有特定的错误。在错误链上获取特定种类的错误，使用errors.As
- panic：不建议在业务代码中使用 panic。调用函数不包含 recover 会造成程序崩溃。若问题可以被屏蔽或解决，建议使用error 代替 panic。当程序启动阶段发生不可逆转的错误时，可以在 init 或 main 函数中使用 panic。
- recover：recover 只能在被 defer 的函数中使用嵌套无法生效。只在当前 goroutine 生效defer 的语句是后进先出。如果需要更多的上下文信息，可以 recover 后在 log 中记录当前的调用栈。

```go
func main(){
  if true{
    defer fmt.Printf("1")
  } else {
    defer fmt.Printf("2")
  }
  defer fmt.Printf("3")
}
//输出
3
1
```

在 Go 语言中，`defer` 语句会在函数执行结束时按照 **后进先出**（LIFO，Last In First Out）顺序执行。

`if true` **分支执行**：
因为 `if true` 条件成立，进入 `if` 分支：这会把 `fmt.Printf("1")` 加入到 **defer 队列**，但是 **并不会立即执行**。它会等到 `main` 函数即将结束时才执行。`defer fmt.Printf("3")`： 不论前面 `if` 语句进入的是哪一分支，`defer fmt.Printf("3")` 都会被执行，它同样被推入 **defer 队列**，等待函数结束时执行。

函数结束时执行的顺序： 函数执行完后，`defer` 语句会按照**后进先出**（LIFO）的顺序执行，所以执行的顺序是：

- 第一个 `defer fmt.Printf("3")`
- 第二个 `defer fmt.Printf("1")`

小结

error 尽可能提供简明的上下文信息链，方便定位问题panic 用于真正异常的情况
recover 生效范围，在当前 goroutine 的被 defer 的函数中生效

性能建议

Go 语言提供了支持基准性能测试的 benchmark 工具

slice 切片

在 Go 中，切片（`slice`）是一个比数组更灵活的、动态的序列类型。切片的大小可以改变，并且不需要在声明时指定固定长度。切片是 Go 中常用的数据结构，提供了动态长度和便捷的操作方式。

数组

```go
package main

import (
  "fmt"
)

func main() {
  //一维数组
  var arr_1 [5] int //所有元素值为0
  fmt.Println(arr_1)

  var arr_2 =  [5] int {1, 2, 3, 4, 5}
  fmt.Println(arr_2)

  arr_3 := [5] int {1, 2, 3, 4, 5}
  fmt.Println(arr_3)

  arr_4 := [...] int {1, 2, 3, 4, 5, 6} //数组大小可省略
  fmt.Println(arr_4)

  arr_5 := [5] int {0:3, 1:5, 4:6} //索引0：3表示arr[0]=3，后面同理，没赋值的默认为0
  fmt.Println(arr_5)

  //二维数组
  var arr_6 = [3][5] int {{1, 2, 3, 4, 5}, {9, 8, 7, 6, 5}, {3, 4, 5, 6, 7}}
  fmt.Println(arr_6)

  arr_7 :=  [3][5] int {{1, 2, 3, 4, 5}, {9, 8, 7, 6, 5}, {3, 4, 5, 6, 7}}
  fmt.Println(arr_7)

  arr_8 :=  [...][5] int {{1, 2, 3, 4, 5}, {9, 8, 7, 6, 5}, {0:3, 1:5, 4:6}}
  fmt.Println(arr_8)
}
```

gofmt 格式化代码

值类型和引用类型

1.1 真正的入门指南：the-way-to-go_ZH_CN

《Go 入门指南》这本开源书籍是一位 Golang 的布道者（无闻）苦于当时国内没有比较好的 Go 语言书籍，而着手翻译的一本国外书籍《The Way to Go》。该书通过对官方的在线文档、名人博客、书籍、相关文章以及演讲的资料收集和整理，并结合我自身在软件工程、编程语言和数据库开发的授课经验，将这些零碎的知识点组织成系统化的概念和技术分类来进行讲解。

该书将从最基础的概念讲起，同时也会讨论一些类似在应用 goroutine 和 channel 时有多少种不同的模式，如何在 Go 语言中使用谷歌 API，如何操作内存，如何在 Go 语言中进行程序测试和如何使用模板来开发 Web 应用这些高级概念和技巧。

1.2 从零开始学 Go Web 编程：build-web-application-with-golang

《Go Web 编程》这本开源书籍，从零开始手把手教你 Go 的环境安装和配置、基本语法再到 Go Web 开发的方方面面。可谓是一书在手，“天下”（Go Web）任我行。当然书写得再好，也是“师傅领进门，修行靠个人啊！“

Part Two：再上一层楼

2.1 七天用 Go 从零实现系列：7days-golang

用 Go 分别写一个：Web 框架、分布式缓存、ORM 框架、RPC 框架的实战教程。有些东西看懂了，不一定会用，会用了也不一定能自己搞出来。所以从零写一个框架，了解其中的细节，才能算上真正懂了吧。

2.2 Go 学习之路：golang-developer-roadmap

《Go 开发者路线图》是一个成为 Go 开发的学习路线。一图胜千言，这里选取这个项目是为了让大家能快速了解 Go 所需学习的知识点和前进的方向。

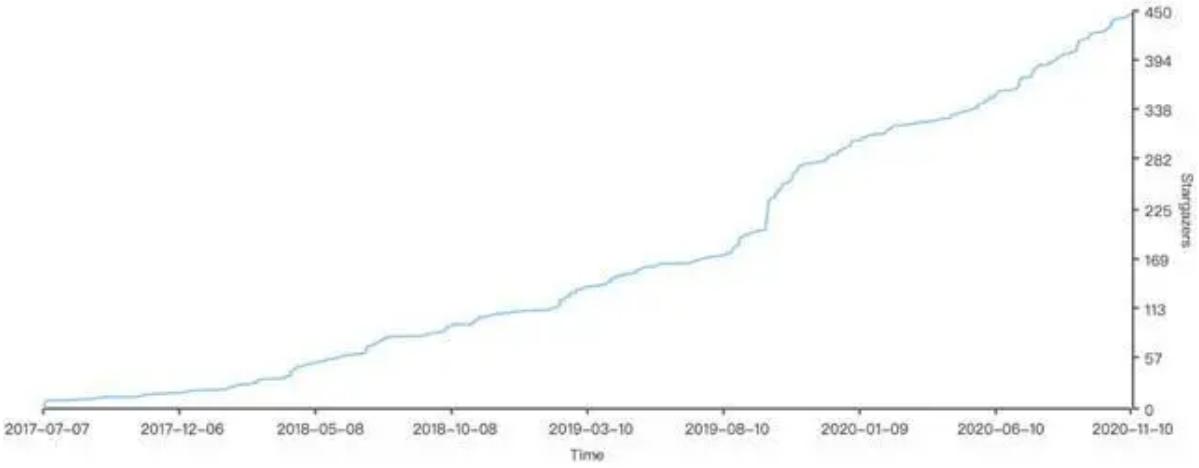
2.3 Go 高级编程：advanced-go-programming-book

《Go 语言高级编程》作为针对有一定 Go 基础的进阶书籍，内容涵盖并发、GOC 编程、Go 汇编语言、RPC 实现、Web 框架实现、分布式系统等高阶主题。该书的附录也是一大亮点，收录了 Go 有趣的代码片段、Go 常见坑。要想 Golang 玩得溜，得在 Go 高级编程下功夫 [手动狗头]

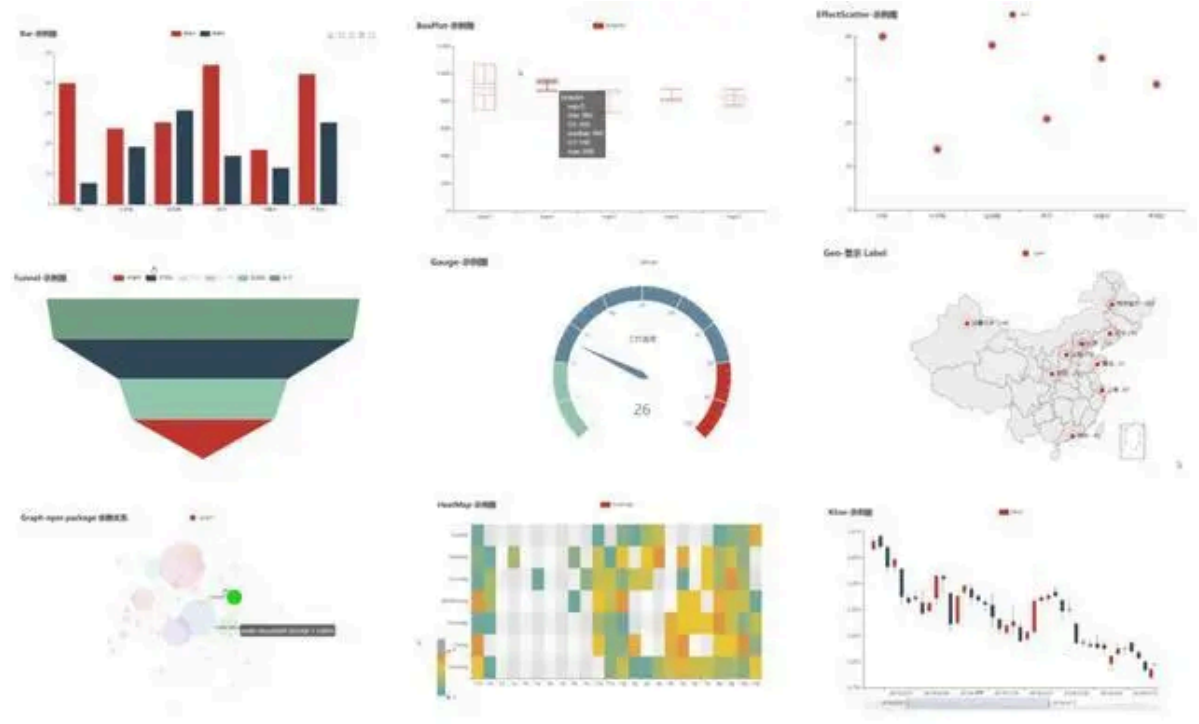
Part Three：是时候展示真正的技术了

3.1 星图：starcharts

这个项目是通过可视化的方式展示 GitHub 上 star 的增长曲线，也就生成是“星图”。推荐这个项目主要是运行简单和直观的数据可视化，可以快速地感受到 Go 开源项目带来的便利。我第一次玩这个项目的时候不会 Go 语言，但是参考这个项目写一个 Python 版本的星图，所以我想已经入门 Go 的各位肯定也能看懂。

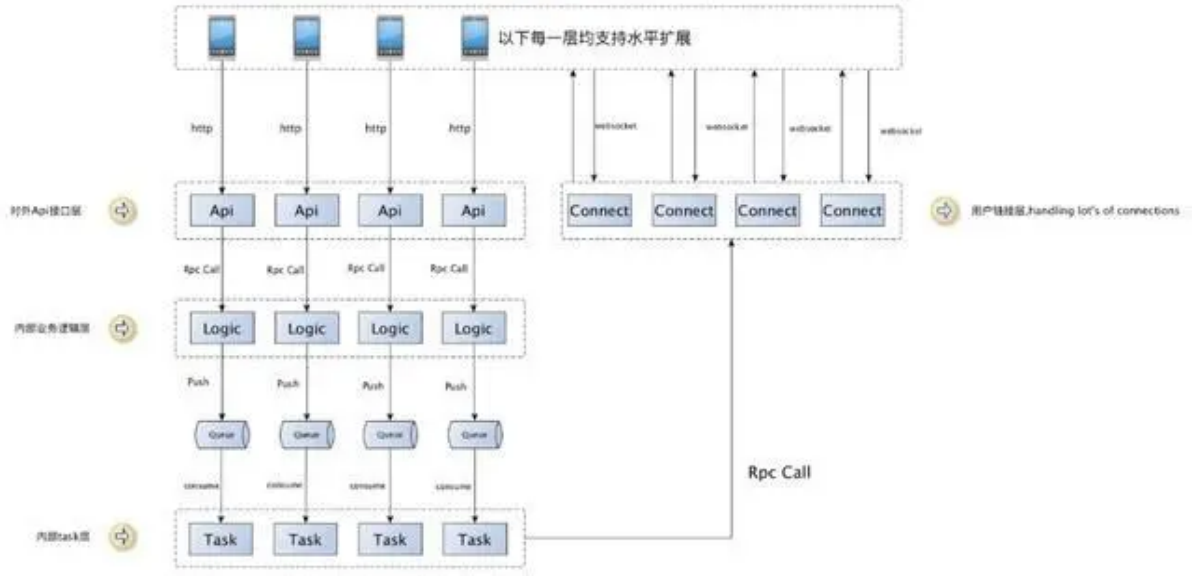


有的小伙伴可能会说我没有开源项目、我的项目都没有 star 我学这个项目没用，我想了下确实是缺少些动力。如果抛开 GitHub 的 star 元素，还有一个适用更多场景的 Go 数据可视化项目：**go-echarts**。来吧，感受下数据的律动。



3.2 来 Chat 下：gochat

gochat 是纯 Go 实现的轻量级即时通讯系统。技术上各层之间通过 RPC 通讯，使用 Redis 作为消息存储与投递的载体，相对 Kafka 操作起来更加方便快捷。各层之间基于 etcd 服务发现，在扩容部署时将会方便很多。架构、目录结构清晰，文档详细。而且还提供了 Docker 一键构建，安装运行都十分方便。



3.3 给！拿去用：annie

Go 编写的下载快速、使用简单、程序纯净的视频下载工具。支持哔哩哔哩、YouTube 等视频网。可作为前段时间被封禁：youtube-dl 的替代品（目前已重新上架），它真的很强大！先感受下 annie 带来的便利，可能就有兴趣去探究它的源码啦。

$ annie -c cookies.txt xxxx: xxxx 高清 1080P60 Size: 220.65 MiB (231363071 Bytes) # download with: annie -f default "URL" 16.03 MiB / 220.65 MiB [==>----------] 7.26% 9.65 MiB/s 19s

最后

推荐几个 GitHub 上的大佬：

astaxie：谢大unknwon：无闻polaris1119：polarisxu，Go 语言中文站长appleboy、chai2010

基本类型和运算符

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。

在格式化输出（printf）时，你可以使用 `%t` 来表示你要输出的值为布尔型。

Go 语言中没有 float 类型。（Go语言中只有 `float32` 和 `float64`）没有 double 类型。

你可以通过增加前缀 0 来表示 8 进制数（如：077），增加前缀 0x 来表示 16 进制数（如：`0xFF`），以及使用 `e` 来表示 10 的连乘（如： 1e3 = 1000，或者 6.022e23 = 6.022 x 1e23）。

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），`%0nd` 用于规定输出长度为 n 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`。

有符号

- `int`(在32位操作系统为int32，在64为操作系统中为int64)
- `int8`（-128 -> 127） （-2^8~2^8-1）
- `int16`（-32768 -> 32767）
- `int32`（-2,147,483,648 -> 2,147,483,647）
- `int64`（-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807）

无符号整数：

- `uint8`（0 -> 255）
- `uint16`（0 -> 65,535）
- `uint32`（0 -> 4,294,967,295）
- `uint64`（0 -> 18,446,744,073,709,551,615）

使用 `e` 来表示 10 的连乘（如： 1e3 = 1000，或者 6.022e23 = 6.022 x 1e23）。

浮点型（IEEE-754 标准）：

- `float32`（+- 1e-45 -> +- 3.4 * 1e38）
- `float64`（+- 5 * 1e-324 -> 107 * 1e308）

位运算

二元运算符

```
//按位与 `&`
  1 & 1 -> 1
  1 & 0 -> 0
  0 & 1 -> 0
  0 & 0 -> 0

//按位或 |
  1 | 1 -> 1
  1 | 0 -> 1
  0 | 1 -> 1
  0 | 0 -> 0

//按位异或 ^（相同为0，不同为1）
  1 ^ 1 -> 0
  1 ^ 0 -> 1
  0 ^ 1 -> 1
  0 ^ 0 -> 0

//位清除 &^
//x&^y，若y对应位置位1，则及那个x上对应位置变为0
 package main
 import "fmt"
 func main() {
  var x uint8 = 15 //二进制00001111
  var y uint8 = 4 //      00000100
  fmt.Printf("%08b\n", x&^y);  // 00001011
 }
```

一元运算符

```
//按位补足 ^：
//该运算符与异或运算符一同使用，即 m^x，对于无符号 x 使用“全部位设置为 1”的规则，对于有符号 x 时使用 m=-1。例如：
  ^10 = -01 ^ 10 = -11

//位左移 <<
```

别名

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），然后就可以使用 `TZ` 来操作 `int` 类型的数据。