1.数据类型

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需**内存大小不同的数据**，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：

序号	类型和描述
1	**布尔型** 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
2	**数字类型** 整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且支持复数，其中位的运算采用补码。
3	**字符串类型:** 字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。
4	**派生类型:** 包括：(a) 指针类型（Pointer）(b) 数组类型(c) 结构化类型(struct)(d) Channel 类型(e) 函数类型(f) 切片类型(g) 接口类型（interface）(h) Map 类型

数字类型

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

序号	类型和描述
1	**uint8** 无符号 8 位整型 (0 到 255)
2	**uint16** 无符号 16 位整型 (0 到 65535)
3	**uint32** 无符号 32 位整型 (0 到 4294967295)
4	**uint64** 无符号 64 位整型 (0 到 18446744073709551615)
5	**int8** 有符号 8 位整型 (-128 到 127)
6	**int16** 有符号 16 位整型 (-32768 到 32767)
7	**int32** 有符号 32 位整型 (-2147483648 到 2147483647)
8	**int64** 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807)

浮点类型

序号	类型和描述

1	float32 IEEE-754 32位浮点型数
2	float64 IEEE-754 64位浮点型数
3	complex64 32 位实数和虚数
4	complex128 64 位实数和虚数

其他数字类型

序号	类型和描述
1	**byte** 类似 uint8
2	**rune** 类似 int32
3	**uint** 32 或 64 位
4	**int** 与 uint 一样大小
5	**uintptr** 无符号整型，用于存放一个指针

2.变量

变量可以通过变量名访问。

声明变量

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语言中的变量需要声明后才能使用，同一作用域内不支持重复声明。 并且Go语言的变量声明后**必须使用。**（不使用的话会报错）

声明变量的一般形式是使用 var 关键字：

```
var identifier type
```

可以一次声明同一个类型的多个变量：

```
#模板 var identifier1, identifier2 type
var name,email string
```

var： 声明变量关键字

identifier： 变量名称

type ： 变量类型
也还可以不同类型多个变量声明

```
var (
    name string
    age int
    istrue bool
    price float32
)
```

在多个短变量声明和赋值的时候，至少有一个新声明的变量出现在左值，即使其他变量名可能是重复声明的，编译器也不会报错。

```
conn err := net.Dial("tcp","127.0.0.1:8008")
conn1 err := net.Dial("tcp","127.0.0.1:8008")
```

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 。 （空）

变量初始化的标准格式如下：

```
var 变量名 类型 = 表达式
```

类型推导：

有时候我们会将变量的类型省略，这个时候编译器会根据等号右边的值来推导变量的类型完成初始化。

```
var name = "林翔" //字符串类型
var age = 18 //整型
var isTrue = true //布尔型
#或者一次初始化多个变量
var name,age,isTrue = "linx",18,true
```

短变量声明

在**函数内部**，可以使用更简略的 **:=** 方式**声明并初始化变量。注意只能函数内部，外部不可以使用。**

```
name := "包子" //字符串类型
age := 18 //整型
isTrue := true //布尔型
#或者一次初始化多个变量
name,age,isTrue := "包子",18,true
```

匿名变量

在使用多重赋值时，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，就是不使用这个变量，忽略这个变量，下划线表示变量名称。**例如：

```
func getInfo() (string, int) {
    return "linx", 18
}
func main() {
    name, _ := getInfo()
    //这样就忽略了 18
    fmt.Printf("name: %v\n", name)
    _, age := getInfo()
    //这样就忽略了 linx
    fmt.Printf("age: %v\n", age)
}
```

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。**

“_”本身就是一个特殊的标识符，被称为空白标识符。它可以像其他标识符那样用于变量的声明或赋值（任何类型都可以赋值给它），但任何赋给这个标识符的值都将被抛弃，因此这些值不能在后续的代码中使用，也不可以使用这个标识符作为变量对其它变量进行赋值或运算。

需要注意的地方：

函数外的每个语句都必须以关键字开始（var、const、func等）

:= 不能使用在函数外。

_ 多用于占位，表示忽略值。

3.常量

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

相对于变量，常量是恒定不变的值，多用于定义程序运行期间不会改变的那些值。 常量的声明和变量声明非常类似，只是把 var 换成了 const ，**常量在定义的时候必须赋值。**

常量定义

定义一个常量使用 const 关键字，语法格式如下：

```
const identifier [type]= value
```

const： 定义常量关键字

identifier： 常量名称

type： 常量类型

value： 常量的值

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

显示类型定义

```
const b string = "abc"
```

隐式类型定义

```
 const b = "abc"
```

多个相同类型的声明可以简写为：

```
const c_name1, c_name2 = value1, value2
```

多个常量声明

```
const (
    LENGTH = 100
    WIDTH = 5
)
```

常量的值必须是能够在编译时就能够确定的；你可以在其赋值表达式中涉及计算过程，但是所有用于计算的值必须在编译期间就能获得。
正确的做法： const c1 = 2/3
错误的做法： const c2 = getNumber() // 引发构建错误: getNumber() used as value
因为在编译期间自定义函数均属于未知，因此无法用于常量的赋值，但内置函数可以使用，如：len()。
常量可以用len(), cap(), unsafe.Sizeof()函数计算表达式的值。**常量表达式中，函数必须是内置函数，否则编译不过。**
数字型的常量是没有大小和符号的，并且可以使用任何精度而不会导致溢出：

```
const Ln2= 0.693147180559945309417232121458\
    176568075500134360255254120680009
const Log2E= 1/Ln2 // this is a precise reciprocal
const Billion = 1e9 // float constant
const hardEight = (1 << 100) >> 97
```

根据上面的例子我们可以看到，**反斜杠 ** 可以在常量表达式中作为多行的连接符使用。
与各种类型的数字型变量相比，你无需担心常量之间的类型转换问题，因为它们都是非常理想的数字。
不过需要注意的是，当常量赋值给一个精度过小的数字型变量时，可能会因为无法正确表达常量所代表的数值而导致溢出，这会在编译期间就引发错误。
另外，**常量也允许使用并行赋值的形式**：

```
const beef, two, c = "eat", 2, "veg"
const Monday, Tuesday, Wednesday, Thursday, Friday, Saturday = 1, 2, 3, 4, 5, 6
const (Monday, Tuesday, Wednesday = 1, 2, 3
    Thursday, Friday, Saturday = 4, 5, 6
)
```

如果定义多个常量时，如果省略了值，则和第一个值相同

```
const (
    A = 10
    B
    C
)
//结果: A=10,B=10,C=10
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

iota

iota ，特殊常量，可以认为是一个可以被编译器修改的常量。

iota 是 go 语言的常量计数器，只能在常量的表达式中使用。 **iota 在 const 关键字出现时将被重置为 0 。 const 中每新增一行常量声明将使 iota 计数一次(** iota 可理解为 const 语句块中的行索引)。

iota 可以被用作枚举值， 使用 iota 能简化定义，在定义枚举时很有用。

```
const (
    a = iota //0
    b = iota //1
    c = iota //2
```

```
)
```

第一个 iota 等于 0，每当 iota 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可以**简写为如下形式：**

```
const (
    a = iota //0
    b //1
    c //2
)
const (
    n1 = iota //0
    n2 = 100 //100
    n3 = iota //2
    n4 //3
)
const n5 = iota //0
```

iota 也可以用在表达式中，如： iota + 50 。在每遇到一个新的常量块或单个常量声明时， iota 都会重置为 0（**简单地讲，每遇到一次 const 关键字，iota 就重置为 0**）。
当然，常量之所以为常量就**是恒定不变的量**，因此我们**无法在程序运行过程中修改它的值**；如果你在代码中试图修改常量的值则会引发编译错误。

iote用法

iota声明中间可以插队，但是索引值还是递增的

```
package main
import "fmt"
func main() {
    const (
        a = iota //0
        b //1
        c //2
        d = "ha" //独立值，iota += 1
        e //"ha" iota += 1
        f = 100 //iota += 1
```

```
        g //100 iota +=1
        h = iota //7,恢复计数
        i //8
    )
    fmt.Println(a, b, c, d, e, f, g, h, i)
}
//运行结果为  0 1 2 ha ha 100 100 7 8
```

再看个有趣的的 iota 实例：

```
package main
import "fmt"

const (
    i = 1 << iota
    j = 3 << iota
    k
    l
)
 func main() {
     fmt.Println("i=", i)
     fmt.Println("j=", j)
     fmt.Println("k=", k)
     fmt.Println("l=", l)
}
//运行结果
//i= 1
//j= 6
//k= 12
//l= 24
```

iota 表示从 0 开始自动加 1，所以 i=1<<0, j=3<<1（<< 表示左移的意思），即：i=1, j=6，这没问题，关键在 k 和 l，从输出结果看 k=3<<2，l=3<<3。

简单表述:

i=1：左移 0 位，不变仍为 1。

j=3：左移 1 位，变为二进制 110，即 6。

k=3：左移 2 位，变为二进制 1100，即 12。

l=3：左移 3 位，变为二进制 11000，即 24。

注：<<n==*(2^n) 从2进制的角度看这个问题

多个 iota 定义在一行

```
const (
    a, b = iota + 1, iota + 2 //1,2  这一行iota都是0
    c, d //2,3  这一行iota都是1
    e, f //3,4  这一行iota都是2
)
```

使用下划线 _ 跳过某些值

```
const (
    a = iota //0
    _ = iota //1 丢弃该值，常用在错误处理中
    c = iota //2
)
```

4.布尔类型

布尔型的值只可以是常量 true 或者 false。

两个类型相同的值可以使用相等 == 或者不等 != 运算符来进行比较并获得一个布尔型的值。完全相同的值的时候会返回 true，否则返回 false，并且只有在两个的值的类型相同的情况下才可以使用。如果值的类型是接口（interface），那么它们也必须都实现了相同的接口。

如果其中一个值是常量，那么另外一个值可以不是常量，但是类型必须和该常量类型相同。

布尔型常用在**条件判断语句**，或者**循环语句**。也可以用在**逻辑表达式中**。

&&(AND) ||(OR) 是具有**短路行为**的，如果运算符左边的值已经可以确定整个布尔表达式的值，那么运算符右边的值将不再被求值。(&&优先级高于||)

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**。

Go语言中**不能用0和非0表示真假**

条件判断示例：

```go
package main
import "fmt"
func main() {
    //满分100，60分为及格
    score := 50
    if score >= 60 { //此处50分不大于60分所以是假FALSE
        fmt.Println("恭喜您及格了！")
    } else if score > 100 {
        fmt.Println("您老越界了！")
    } else {
        fmt.Println("哎呀！咋考的！")
    }
}
//结果  哎呀！咋考的！
```

循环语句示例：

```go
package main
import "fmt"
func main() {
    total := 10
    for i := 0; i < total; i++ { //此处 i<total 当i大于等于10时为FALSE
        fmt.Printf("i: %v\n", i)
    }
}
//输出为
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
```

逻辑表达式示例：

```
package main
import "fmt"
func main() {
    age := 18
    gender := "男"
    if age >= 18 && gender == "男" { //此处&&和 两侧都为真才是TRUE，只要有一个判断为假就是FALSE
        fmt.Println("您已是成年男子")
    }
    if age >= 18 || gender == "男" { //此处||和 一侧为真就是是TRUE，两侧都为假就是FALSE
        fmt.Println("您已成年")
    }

    var a bool = true
    var b bool = false
    if !(a && b) { //逻辑! NOT运算符。如果条件为TRUE,则逻辑NOT条件FALSE，否则为TRUE
        fmt.Printf("条件为 true\n")
    }
}
//结果
您已是成年男子
您已成年
条件为 tru
```

5.数字类型

整型 int 和浮点型 float

Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

这些类型的长度都是根据**运行程序所在的操作系统类型所决定**的：

int 和 uint 在 32 位操作系统上，它们均使用 32 位（4 个字节），在 64 位操作系统上，它们均使用 64 位（8 个字节）。

uintptr 的长度被设定为足够存放一个指针即可。

Go 语言中**没有 float 类型。**（Go语言中只有 float32 和 float64 ）

与操作系统架构无关的类型都有固定的大小，并在类型的名称中就可以看出来：

整数：

int8（-128 -> 127）

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）

浮点型（IEEE-754 标准）：

float32（+- 1e-45 -> +- 3.4 * 1e38）

float64（+- 5 1e-324 -> 107 1e308）

int 型是计算最快的一种类型。

整型的零值（默认值）为 0，浮点型的零值（默认值）为 0.0。

注意：

float32 精确到小数点后 7 位，float64 精确到小数点后 15 位。由于精确度的缘故，你在使用 == 或者!= 来**比较浮点数时**应当非常小心。你最好在正式使用前测试对于精确度要求较高的运算。

该尽可能地使用 float64，因为 math 包中所有有关数学运算的函数都会要求接收这个类型。

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）。

你可以使用 **a := uint64(0)** 来**同时完成类型转换和赋值操作**，这样 a 的类型就是 uint64。

格式化说明符

%d 用于格式化整数（ %x 和 %X 用于格式化 16 进制表示的数字， %b 表示二进制， %o 代表8进制。），**%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 。

数字值转换

进行类似 **a32bitInt = int32(a32Float)** 的转换时，**小数点后的数字将被丢弃。**这种情况一般发生当**从取值范围较大的类型转换为取值范围较小的类型时**，或者你可以写一个专门用于处理类型转换的函数来确保没有发生精度的丢失。

如果你实际存的数字超出你要转换到的类型的取值范围的话，则会引发 **panic(异常)**

复数

Go 拥有以下复数类型：

```
complex64 (32 位实数和虚数)
complex128 (64 位实数和虚数)
```

复数使用 re+imI 来表示，其中 re 代表实数部分， im 代表虚数部分，I 代表根号负 1。

```
var c1 complex64 = 5 + 10i
fmt.Printf("The value is: %v", c1)
//结果The value is: (5+10i)
```

如果 re 和 im 的类型均为 float32，那么类型为 complex64 的复数 c 可以通过以下方式来获得：

```
c = complex(re, im)
```

函数 real(c) 和 imag(c) 可以分别获得相应的实数和虚数部分。
使用格式化说明符时，可以使用 %v 来表示复数，但当你希望只表示其中的一个部分的时候需要使用%f 。
复数支持和其它数字类型一样的运算。当你使用等号 == 或者不等号 != 对复数进行比较运算时，**注意对精确度的把握。** cmath 包中包含了一些操作复数的公共方法。如果你对内存的要求不是特别高，**最好使用 complex128 作为计算类型，因为相关函数都使用这个类型的参数。**

6.字符串类型

Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。
字符串是一种**值类型，且值不可变**，即创建某个文本后你**无法再次修改**这个文本的内容；更深入地讲，字符串是字节的定长数组。
Go语言字符串是一个任意字节的**常量序列。**[ ] byte类型字节数组

go语言字符串字面量

字符串字面量使用双引号 "" 或者反引号 ' 来创建。**双引号用来创建可解析的字符串，支持转义，但不能用来引用多行；反引号用来创建原生的字符串字面量，可能由多行组成，但不支持转义，并且可以**

包含除了反引号外其他所有字符。双引号创建可解析的字符串应用最广泛，反引号用来创建原生的字符串则多用于书写多行消息，HTML以及正则表达式。
Go 支持以下 2 种形式的字面值：

解释型字符串

该类字符串使用双引号 "" 括起来，其中的相关的转义字符将被替换，这些转义字符包括：
\n： 换行符
\r： 回车符
\t： tab 键
\u 或 \U： Unicode 字符
\\： 反斜杠自身
\f： 换页
\v： 垂直制表符
\'： 单引号 (只用在 ''' 形式的rune符号面值中)
\"： 双引号 (只用在 "..." 形式的字符串面值中)
\a： 响铃
\b： 退格

非解释型字符串

该类字符串使用反引号 ' 括起来，支持换行，例如：

```
`This is a raw string \n`
其中的 `\n\` 会被原样输出
```

字符串切片截取

一般的比较运算符（ == 、 != 、 < 、 <= 、 >= 、 > ）通过在内存中按字节比较来实现字符串的对比。你可以通过函数 len() 来获取字符串所占的字节长度，例如： len(str) 。
字符串的内容（纯字节）可以通过标准索引法来获取，在中括号 [] 内写入索引，索引从 0 开始计数：
字符串 str 的第 1 个字节： str[0]
第 i 个字节： str[i - 1]
最后 1 个字节： str[len(str)-1]
需要注意的是，这种转换方案只对纯 ASCII 码的字符串有效。
注意事项：

获取字符串中某个字节的地址的行为是非法的，例如： &str[i] 。如果试图访问超出字符串索引范围的字节将会导致panic异常， str[len(str)] 。
字符串可以用==和<进行比较；比较通过逐个字节比较完成的，因此比较的结果是字符串自然编码的顺序。
字符串的值是不可变的：一个字符串包含的字节序列永远不会被改变，当然我们也**可以给一个字符串变量分配一个新字符串值。**可以像下面这样将一个字符串追加到另一个字符串：

```
s := "left foot"
t := s
s += ", right foot"
fmt.Println(s) // "left foot, right foot"
fmt.Println(t) // "left foot"
```

这并不会导致原始的字符串值被改变，但是变量s将因为+=语句持有一个新的字符串值，但是t依然是包含原先的字符串值。
因为字符串是不可修改的，因此尝试修改字符串内部数据的操作也是被禁止的：

```
s[0] = 'L' // compile error: cannot assign to s[0]
```

字符串拼接符+

两个字符串 s1 和 s2 可以通过 s := s1 + s2 拼接在一起。
s2 追加在 s1 尾部并生成一个新的字符串 s 。
你可以通过以下方式来对代码中多行的字符串进行拼接：

```
package main
import "fmt"
func main() {
    s1 := "hello"
    s2 := "world"
    s := s1 + s2
    fmt.Printf("s: %v\n", s)
    str := "hello " +
        "linx"   //这里的+ 不能放在这一行
    fmt.Printf("str: %v\n", str)
```

```
}
```

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。
也可以使用fmt.Sprintf() 函数：

```
name := "linx"
age := "2023"
msg := fmt.Sprintf("%s,%s", name, age)
fmt.Printf("msg: %v\n", msg)
//结果  msg: linx,2023
```

在循环中使用加号 + 拼接字符串并不是最高效的做法，更好的办法是使用函数 strings.Join()

```
msg := strings.Join([]string{name, age}, ",")
fmt.Printf("msg: %v\n", msg)
```

使用字节缓冲（ bytes.Buffer ）拼接更加给力。

```
//需要import "bytes"

var bf bytes.Buffer
bf.WriteString("linx")
bf.WriteString(", ")
bf.WriteString("2023")
fmt.Printf("bf.String(): %v\n", bf.String())
// bf.String(): linx, 2023
```

字符串常用方法

7.格式化输出

通用格式化输出

在Go语言中格式化输出通常使用 fmt 包，通用的输出格式如下表所示：

布尔类型

整数类型

浮点型和复数型

字符串与字节数组

指针类型

8.作用域

一个变量（常量、类型或函数）在程序中都有一定的作用范围，称之为作用域。

Go语言(静态语言)会在编译时检查每个变量是否使用过，一旦出现未使用的变量，就会报编译错误。

分为以下三个类型：

函数内定义的变量称为局部变量

函数外定义的变量称为全局变量

函数定义中的变量称为形式参数