Go语言基础

Go语言简介

Go语言（或 Golang）起源于 2007 年，并在 2009 年正式对外发布。Go 是非常年轻的一门语言，它的主要目标是“兼具 Python 等动态语言的开发速度和 C/C++ 等编译型语言的性能与安全性”。

Go语言是编程语言设计的又一次尝试，是对类C语言的重大改进，它不但能让你访问底层操作系统，还提供了强大的网络编程和并发编程支持。Go语言的用途众多，可以进行网络编程、系统编程、并发编程、分布式编程。

Go语言的推出，旨在不损失应用程序性能的情况下降低代码的复杂性，具有“部署简单、并发性好、语言设计良好、执行性能好”等优势，目前国内诸多 IT 公司均已采用Go语言开发项目。

Go语言有时候被描述为“C 类似语言”，或者是“21 世纪的C语言”。Go 从C语言继承了相似的表达式语法、控制流结构、基础数据类型、调用参数传值、指针等很多思想，还有C语言一直所看中的编译后机器码的运行效率以及和现有操作系统的无缝适配。

因为Go语言没有类和继承的概念，所以它和 Java 或 C++ 看起来并不相同。但是它通过接口（interface）的概念来实现多态性。Go语言有一个清晰易懂的轻量级类型系统，在类型之间也没有层级之说。因此可以说Go语言是一门混合型的语言。

Go语言的特性

Go语言也称为 Golang，是由 Google 公司开发的一种静态强类型、编译型、并发型、并具有垃圾回收功能的编程语言。

Go语言主要有语法简单、并发模型、内存分配、垃圾回收、静态链接、标准库、工具链等特性。

Go语言是谷歌在 2009 年发布的一款编程语言，自面世以来它以高效的开发效率和完美的运行速度迅速风靡全球，被誉为“21 世纪的C语言”。

现在越来越多的公司开始使用Go语言开发自己的服务，同时也诞生了很多使用Go语言开发的服务和应用，比如 Docker、k8s 等，下面我们来看一下，有哪些大公司在使用Go语言。现在Google、Facebook、腾讯、百度、七牛云、京东、小米、360、美团、滴滴、新浪等公司的新项目开始使用Go语言。

Go语言之依赖管理

go module是Go1.11版本之后官方推出的版本管理工具，并且从Go1.13版本开始，go module将是Go语言默认的依赖管理工具。

go module

GO111MODULE

要启用go module支持首先要设置环境变量GO111MODULE，通过它可以开启或关闭模块支持，它有三个可选值：off、on、auto，默认值是auto。

- GO111MODULE=off禁用模块支持，编译时会从GOPATH和vendor文件夹中查找包。
- GO111MODULE=on启用模块支持，编译时会忽略GOPATH和vendor文件夹，只根据 go.mod下载依赖。

- GO111MODULE=auto，当项目在$GOPATH/src外且项目根目录有go.mod文件时，开启模块支持。

使用 go module 管理依赖后会在项目根目录下生成文件go.mod，此种方法能够很好的管理项目依赖的第三方包信息。

go mod命令

常用的 go mod 命令如下：

```
go mod download    下载依赖的module到本地cache（默认为$GOPATH/pkg/mod目录）
go mod edit        编辑go.mod文件
go mod graph       打印模块依赖图
go mod init        初始化当前文件夹，创建go.mod文件
go mod tidy        增加缺少的module，删除无用的module
go mod vendor      将依赖复制到vendor下
go mod verify      校验依赖
go mod why         解释为什么需要依赖
```

go.mod

go.mod 文件记录了项目所有的依赖信息，其结构大致如下：

```
module github.com/Q1mi/studygo/blogger

go 1.12

require (
    github.com/DeanThompson/ginpprof v0.0.0-20190408063150-3be636683586
    github.com/gin-gonic/gin v1.4.0
    github.com/go-sql-driver/mysql v1.4.1
    github.com/jmoiron/sqlx v1.2.0
    github.com/satori/go.uuid v1.2.0
    google.golang.org/appengine v1.6.1 // indirect
)
```

其中，

- module 用来定义包名
- require 用来定义依赖包及版本
- indirect 表示间接引用

go get

在项目中执行 go get 命令可以下载依赖包，并且还可以指定下载的版本。

- 运行 go get -u 将会升级到最新的次要版本或者修订版本(x.y.z, z是修订版本号， y是次要版本号)
- 运行 go get -u=patch 将会升级到最新的修订版本
- 运行 go get package@version 将会升级到指定的版本号version

如果下载所有依赖可以使用 go mod download 命令。

在项目中使用 go mod

如果需要对一个已经存在的项目启用go module，可以按照以下步骤操作：

- 在**项目目录下**执行go mod init ProjectName(项目名称)，生成一个 go.mod 文件。
- 执行go mod tidy，增加缺少的module，删除无用的module，实现导入包的更新。

使用 go mod 导入本地包

go mod 导入本地包简介

Go语言基础语法

标识符与关键字

标识符

在编程语言中标识符就是程序员定义的具有特殊意义的词，比如变量名、常量名、函数名等等。 Go语言中标识符由字母数字和_(下划线）组成，并且只能以字母和_开头。 举几个例子：abc, _, _123, a123。

关键字

关键字是指编程语言中预先定义好的具有特殊含义的标识符。 关键字和保留字都不建议用作变量名。

Go语言中有25个关键字：

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

此外，Go语言中还有37个保留字。

```
Constants:    true  false  iota  nil

    Types:    int  int8  int16  int32  int64
              uint  uint8  uint16  uint32  uint64  uintptr
              float32  float64  complex128  complex64
              bool  byte  rune  string  error

Functions:   make  len  cap  new  append  copy  close  delete
             complex  real  imag
             panic  recover
```

变量

变量的来历

程序运行过程中的数据都是保存在内存中，我们想要在代码中操作某个数据时就需要去内存上找到这个变量，但是如果我们直接在代码中通过内存地址去操作变量的话，代码的可读性会非常差而且还容易出错，所以我们就利用变量将这个数据的内存地址保存起来，以后直接通过这个变量就能找到内存上对应的数据了。

变量类型

变量（Variable）的功能是存储数据。不同的变量保存的数据类型可能会不一样。经过半个多世纪的发展，编程语言已经基本形成了一套固定的类型，常见变量的数据类型有：整型、浮点型、布尔型等。

- bool
- string
- int、int8、int16、int32、int64（有符号整型）
- uint、uint8、uint16、uint32、uint64、uintptr（无符号整型）
- byte // uint8 的别名
- rune // int32 的别名 代表一个 Unicode 码
- float32、float64
- complex64、complex128（复数）

Go语言中的每一个变量都有自己的类型，并且变量必须经过声明才能开始使用。

变量声明

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。

标准声明

Go语言的变量声明格式为：

```
var 变量名 变量类型
```

变量声明以关键字 var 开头，变量类型放在变量的后面，行尾无需分号。 举个例子：

```
var name string
var age int
var isOk bool
```

批量声明

每声明一个变量就需要写var关键字会比较繁琐，go语言中还支持批量变量声明：

```
var (
    a string
    b int
    c bool
    d float32
)
```

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。

当然我们也可在声明变量的时候为其指定初始值。变量初始化的标准格式如下：

```
var 变量名 类型 = 表达式
```

举个例子：

```
var name string = "Q1mi"
var age int = 18
```

或者一次初始化多个变量

```
var name, age = "Q1mi", 20
```

1.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 = "Q1mi"
var age = 18
```

2.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。

```
package main

import (
    "fmt"
)
// 全局变量m
var m = 100

func main() {
    n := 10
    m := 200 // 此处声明局部变量m
    fmt.Println(m, n)
}
```

3.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，例如：

```
func foo() (int, string) {
    return 10, "Q1mi"
}
func main() {
    x, _ := foo()
    _, y := foo()
    fmt.Println("x=", x)
    fmt.Println("y=", y)
}
```

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。 (在Lua等编程语言里，匿名变量也被叫做哑元变量。)

注意事项：

- 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。

常量

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。**

```
const pi = 3.1415
const e = 2.7182
```

声明了 pi 和 e 这两个常量之后，在整个程序运行期间它们的值都不能再发生变化了。

多个常量也可以一起声明：

```
const (
    pi = 3.1415
    e = 2.7182
)
```

const同时声明多个常量时，如果省略了值则表示和上面一行的值相同。 例如：

```
const (
    n1 = 100
    n2
    n3
)
```

上面示例中，常量n1、n2、n3的值都是100。

iota

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)。 使用iota能简化定义，在定义枚举时很有用。

举个例子：

```
const (
        n1 = iota //0
        n2        //1
        n3        //2
        n4        //3
    )
```

几个常见的iota示例:

使用_跳过某些值

```
const (
        n1 = iota //0
        n2        //1
        _
        n4        //3
    )
```

iota声明中间插队

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

定义数量级 （这里的 << 表示左移操作，1<<10表示将1的二进制表示向左移10位，也就是由1变成了10000000000(二进制)，也就是十进制的1024。同理2 << 2表示将2的二进制表示向左移2位，也就是由10变成了1000，也就是十进制的8。）
多个iota定义在一行

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

基本数据类型

Go语言中有丰富的数据类型，除了基本的整型、浮点型、布尔型、字符串外，还有数组、切片、结构体、函数、map、通道（channel）等。Go 语言的基本类型和其他语言大同小异。
格式化指令 含义
%%　%字面量
%b　一个二进制整数，将一个整数格式转化为二进制的表达方式
%c　一个Unicode的字符
%d　十进制整数
%o　八进制整数
%x　小写的十六进制数值
%X　大写的十六进制数值
%U　一个Unicode表示法表示的整型码值
%s　输出以原生的UTF8字节表示的字符，如果console不支持utf8编码，则会乱码
%t　以true或者false的方式输出布尔值
%v　使用默认格式输出值，或者如果方法存在，则使用类性值的String()方法输出自定义值
%T　输出值的类型

整型

整型分为以下两个大类： 按长度分为：int8、int16、int32、int64 对应的无符号整型：uint8、uint16、uint32、uint64

其中，uint8就是我们熟知的byte型，int16对应C语言中的short型，int64对应C语言中的long型。

- uint8 无符号 8位整型 (0 到 255)
- uint16 无符号 16位整型 (0 到 65535)
- uint32 无符号 32位整型 (0 到 4294967295)
- uint64 无符号 64位整型 (0 到 18446744073709551615)
- int8 有符号 8位整型 (-128 到 127)
- int16 有符号 16位整型 (-32768 到 32767)
- int32 有符号 32位整型 (-2147483648 到 2147483647)
- int64 有符号 64位整型 (-9223372036854775808 到 9223372036854775807)

特殊整型

- 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
  注意： 在使用 int 和 uint 类型时，不能假定它是32位或64位的整型，而是考虑int和uint可能在不同平台上的差异。

注意事项 获取对象的长度的内建len()函数返回的长度可以根据不同平台的字节长度进行变化。实际使用中，切片或 map 的元素数量等都可以用int来表示。在涉及到二进制传输、读写文件的结构描述时，为了保持文件的结构不会受到不同编译目标平台字节长度的影响，不要使用int和 uint。

数字字面量语法（Number literals syntax）

Go1.13版本之后引入了数字字面量语法，这样便于开发者以二进制、八进制或十六进制浮点数的格式定义数字，例如：

v := 0b00101101， 代表二进制的 101101，相当于十进制的 45。 v := 0o377，代表八进制的 377，相当于十进制的 255。 v := 0x1p-2，代表十六进制的 1 除以 2^2，也就是 0.25。

而且还允许我们用 _ 来分隔数字，比如说： v := 123_456 表示 v 的值等于 123456。

我们可以借助fmt函数来将一个整数以不同进制形式展示。

```
package main

import "fmt"

func main(){
    // 十进制
    var a int = 10
    fmt.Printf("%d \n", a)  // 10
    fmt.Printf("%b \n", a)  // 1010   占位符%b表示二进制

    // 八进制  以0开头
    var b int = 077
    fmt.Printf("%o \n", b)  // 77

    // 十六进制  以0x开头
    var c int = 0xff
```

```
    fmt.Printf("%x \n", c)  // ff
    fmt.Printf("%X \n", c)  // FF
}
```

浮点型

Go语言支持两种浮点型数：float32 和 float64。这两种浮点型数据格式遵循 IEEE 754 标准： float32 的浮点数的最大范围约为 3.4e38，可以使用常量定义：math.MaxFloat32。 float64 的浮点数的最大范围约为 1.8e308，可以使用一个常量定义：math.MaxFloat64。

打印浮点数时，可以使用fmt包配合动词%f，代码如下：

```
package main
import (
        "fmt"
        "math"
)
func main() {
        fmt.Printf("%f\n", math.Pi)
        fmt.Printf("%.2f\n", math.Pi)
}
```

复数

complex64和complex128

```
var c1 complex64
c1 = 1 + 2i
var c2 complex128
c2 = 2 + 3i
fmt.Println(c1)
fmt.Println(c2)
```

复数有实部和虚部，complex64的实部和虚部为32位，complex128的实部和虚部为64位。

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

Go语言中的字符串以原生数据类型出现，使用字符串就像使用其他原生数据类型（int、bool、float32、float64 等）一样。 Go 语言里的字符串的内部实现使用UTF-8编码。 字符串的值为双引号(")中的内容，可以在Go语言的源码中直接添加非ASCII码字符，例如：

```
s1 := "hello"
s2 := "你好"
```

字符串转义符

Go 语言的字符串常见转义符包含回车、换行、单双引号、制表符等，如下表所示。

- \r　回车符（返回行首）
- \n　　换行符（直接跳到下一行的同列位置）
- \t　制表符
- \' 单引号
- \"　双引号
- \\　反斜杠
  举个例子，我们要打印一个Windows平台下的一个文件路径：
  fmt.Println("\"hello Go!\"")

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("str := \"c:\\Code\\lesson1\\go.exe\"")
}
```

多行字符串

Go语言中要定义一个多行字符串时，就必须使用反引号字符：

```
s1 := `第一行
第二行
第三行
`
fmt.Println(s1)
```

反引号间换行将被作为字符串中的换行，但是所有的转义字符均无效，文本将会原样输出。

字符串的常用操作

- 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

byte和rune类型

组成每个字符串的元素叫做“字符”，可以通过遍历或者单个获取字符串元素获得字符。 字符用单引号（’）包裹起来，如：

```
var a = '中'
var b = 'x'
```

Go 语言的字符有以下两种：

- uint8类型，或者叫 byte 型，代表一个ASCII码字符。

- rune类型，代表一个 UTF-8字符。

当需要处理中文、日文或者其他复合字符时，则需要用到rune类型。rune类型实际是一个int32。

Go 使用了特殊的 rune 类型来处理 Unicode，让基于 Unicode 的文本处理更为方便，也可以使用 byte 型进行默认字符串处理，性能和扩展性都有照顾。
// 遍历字符串

```
func traversalString() {
    s := "hello沙河"
    for i := 0; i < len(s); i++ { //byte
        fmt.Printf("%v(%c) ", s[i], s[i])
    }
    fmt.Println()
    for _, r := range s { //rune
        fmt.Printf("%v(%c) ", r, r)
    }
    fmt.Println()
}
```

输出：

```
104(h) 101(e) 108(l) 108(l) 111(o) 230(æ) 178(²) 153() 230(æ) 178(²) 179(³)
104(h) 101(e) 108(l) 108(l) 111(o) 27801(沙) 27827(河)
```

因为UTF8编码下一个中文汉字由3~4个字节组成，所以我们不能简单的按照字节去遍历一个包含中文的字符串，否则就会出现上面输出中第一行的结果。

字符串底层是一个byte数组，所以可以和[]byte类型相互转换。字符串是不能修改的 字符串是由byte字节组成，所以字符串的长度是byte字节的长度。 rune类型用来表示utf8字符，一个rune字符由一个或多个byte组成。

修改字符串

要修改字符串，需要先将其转换成[]rune或[]byte，完成后再转换为string。无论哪种转换，都会重新分配内存，并复制字节数组。

```
func changeString() {
    s1 := "big"
    // 强制类型转换
    byteS1 := []byte(s1)
    byteS1[0] = 'p'
    fmt.Println(string(byteS1))

    s2 := "白萝卜"
    runeS2 := []rune(s2)
    runeS2[0] = '红'
    fmt.Println(string(runeS2))
}
```

类型转换

Go语言中只有强制类型转换，没有隐式类型转换。该语法只能在两个类型之间支持相互转换的时候使用。

强制类型转换的基本语法如下：

```
T(表达式)
```

其中，T表示要转换的类型。表达式包括变量、复杂算子和函数返回值等.

比如计算直角三角形的斜边长时使用math包的Sqrt()函数，该函数接收的是float64类型的参数，而变量a和b都是int类型的，这个时候就需要将a和b强制类型转换为float64类型。

```
func sqrtDemo() {
    var a, b = 3, 4
    var c int
    // math.Sqrt()接收的参数是float64类型，需要强制转换
    c = int(math.Sqrt(float64(a*a + b*b)))
    fmt.Println(c)
}
```

练习题

1.编写代码分别定义一个整型、浮点型、布尔型、字符串型变量，使用fmt.Printf()搭配%T分别打印出上述变量的值和类型。
2.编写代码统计出字符串"hello沙河小王子"中汉字的数量。

运算符

Go 语言内置的运算符有：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符

算术运算符

- 相加　+
- 相减　-
- 相乘　*
- 相除　\
- 取余　%

  注意： ++（自增）和--（自减）在Go语言中是单独的语句，并不是运算符。

关系运算符

- ==　检查两个值是否相等，如果相等返回 True 否则返回 False。
- !=　检查两个值是否不相等，如果不相等返回 True 否则返回 False。
- > 检查左边值是否大于右边值，如果是返回 True 否则返回 False。
- >=　检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。
- <　检查左边值是否小于右边值，如果是返回 True 否则返回 False。

- <=　检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。

逻辑运算符

- &&　逻辑 AND 运算符。 如果两边的操作数都是 True，则为 True，否则为 False。
- ||　逻辑 OR 运算符。 如果两边的操作数有一个 True，则为 True，否则为 False。
- ！逻辑 NOT 运算符。 如果条件为 True，则为 False，否则为 True。

位运算符

位运算符对整数在内存中的二进制位进行操作。

- &　参与运算的两数各对应的二进位相与。（两位均为1才为1）
- | 参与运算的两数各对应的二进位相或。（两位有一个为1就为1）
- ^ 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。（两位不一样则为1）
- <<　左移n位就是乘以2的n次方。“a<<b”是把a的各二进位全部左移b位，高位丢弃，低位补0。
- >>　右移n位就是除以2的n次方。“a>>b”是把a的各二进位全部右移b位。

赋值运算符

- = 简单的赋值运算符，将一个表达式的值赋给一个左值
- +=　相加后再赋值
- -=　相减后再赋值
- *=　相乘后再赋值
- /=　相除后再赋值
- %=　求余后再赋值
- <<=　左移后赋值
- >>=　右移后赋值
- &=　按位与后赋值
- |=　按位或后赋值
- ^=　按位异或后赋值

练习题

有一堆数字，如果除了一个数字以外，其他数字都出现了两次，那么如何找到出现一次的数字?

Go语言基础之流程控制

流程控制是每种编程语言控制逻辑走向和执行次序的重要部分，流程控制可以说是一门语言的“经脉”。

Go语言中最常用的流程控制有 if 和 for，而 switch 和 goto 主要是为了简化代码、降低重复代码而生的结构，属于扩展类的流程控制。

if else(分支结构)

if条件判断基本写法

Go语言中if条件判断的格式如下：

```
if 表达式1 {
    分支1
} else if 表达式2 {
    分支2
} else{
    分支3
}
```

当表达式1的结果为 true 时，执行分支1，否则判断表达式2，如果满足则执行分支2，都不满足时，则执行分支3。 if 判断中的 else if 和 else 都是可选的，可以根据实际需要进行选择。

Go语言规定与 if 匹配的左括号{必须与if和表达式放在同一行，{ 放在其他位置会触发编译错误。 同理，与 else 匹配的 { 也必须与else写在同一行，else 也必须与上一个 if 或 else if 右边的大括号在同一行。举个例子：

```
func main() {
    score := 65
    if score >= 90 {
        fmt.Println("A")
    } else if score > 75 {
        fmt.Println("B")
    } else {
        fmt.Println("C")
    }
}
```

if条件判断特殊写法

if条件判断还有一种特殊的写法，可以在 if 表达式之前添加一个执行语句，再根据变量值进行判断，举个例子：

```
func main() {
    if score := 65; score >= 90 {
        fmt.Println("A")
    } else if score > 75 {
        fmt.Println("B")
    } else {
        fmt.Println("C")
    }
}
```

思考题： 上下两种写法的区别在哪里？

for(循环结构)

Go 语言中的所有循环类型均可以使用for关键字来完成。

for循环的基本格式如下：

```
for 初始语句;条件表达式;结束语句{
    循环体语句
}
```

条件表达式返回 true 时循环体不停地进行循环，直到条件表达式返回 false 时自动退出循环。

```
func main() {
    for i := 0; i < 10; i++ {
        fmt.Println(i)
    }
}
```

for循环的初始语句可以被忽略，但是初始语句后的分号必须要写，例如：

```
func main() {
    i := 0
    for ; i < 10; i++ {
        fmt.Println(i)
    }
}
```

for循环的初始语句和结束语句都可以省略，例如：

```
func main() {
    i := 0
    for i < 10 {
        fmt.Println(i)
        i++
    }
```

这种写法类似于其他编程语言中的while，在while后添加一个条件表达式，满足条件表达式时持续循环，否则结束循环。

无限循环

```
for {
    循环体语句
}
```

无限循环可以通过break、goto、return、panic语句强制退出循环。

for range(键值循环)

Go语言中可以使用for range遍历数组、切片、字符串、map 及通道（channel）。 通过for range遍历的返回值有以下规律：

- 数组、切片、字符串返回索引和值。
- map返回键和值。
- 通道（channel）只返回通道内的值

练习：给定一个字符串（byte）“asq3241[\]”，统计字母，数字和其他字符的数量。

switch case

使用 switch 语句可方便地对大量的值进行条件判断。

```
func main() {
    finger := 3
    switch finger {
    case 1:
        fmt.Println("大拇指")
    case 2:
        fmt.Println("食指")
    case 3:
        fmt.Println("中指")
    case 4:
        fmt.Println("无名指")
    case 5:
        fmt.Println("小拇指")
    default:
        fmt.Println("无效的输入！")
    }
}
```

Go语言规定每个switch只能有一个default分支。

一个分支可以有多个值，多个case值中间使用英文逗号分隔。

```
func main() {
    switch n := 7; n {
    case 1, 3, 5, 7, 9:
        fmt.Println("奇数")
    case 2, 4, 6, 8:
        fmt.Println("偶数")
    default:
        fmt.Println(n)
    }
}
```

分支还可以使用表达式，这时候switch语句后面不需要再跟判断变量。例如：

```
func main() {
    age := 30
    switch {
    case age < 25:
        fmt.Println("好好学习吧")
    case age > 25 && age < 35:
        fmt.Println("好好工作吧")
    case age > 60:
        fmt.Println("好好享受吧")
    default:
        fmt.Println("活着真好")
    }
}
```

fallthrough 语法可以执行满足条件的case的下一个case，是为了兼容C语言中的case设计的。

```
func switchDemo5() {
    s := "a"
    switch {
    case s == "a":
        fmt.Println("a")
        fallthrough
    case s == "b":
        fmt.Println("b")
    case s == "c":
        fmt.Println("c")
    default:
        fmt.Println("...")
    }
}
```

输出：

```
a
b
```

goto(跳转到指定标签)

goto语句通过标签进行代码间的无条件跳转。goto语句可以在快速跳出循环、避免重复退出上有一定的帮助。Go语言中使用goto语句能简化一些代码的实现过程。 例如双层嵌套的for循环要退出时：

```
func main() {
    var breakFlag bool
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                // 设置退出标签
                breakFlag = true
                break
            }
            fmt.Printf("%v-%v\n", i, j)
        }
        // 外层for循环判断
        if breakFlag {
            break
        }
    }
}
```

使用goto语句能简化代码：

```
func gotoDemo2() {
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                // 设置退出标签
                goto breakTag
            }
            fmt.Printf("%v-%v\n", i, j)
```

```
        }
    }
    return
    // 标签
breakTag:
    fmt.Println("结束for循环")
}
```

break(跳出循环)

break 语句可以结束 for、switch 和 select 的代码块。

break 语句还可以在语句后面添加标签，表示退出某个标签对应的代码块，标签要求必须定义在对应的 for、switch 和 select 的代码块上。 举个例子：

```
func mian() {
BREAKDEMO1:
    for i := 0; i < 10; i++ {
        for j := 0; j < 10; j++ {
            if j == 2 {
                break BREAKDEMO1
            }
            fmt.Printf("%v-%v\n", i, j)
        }
    }
    fmt.Println("...")
}
```

continue(继续下次循环)

continue 语句可以结束当前循环，开始下一次的循环迭代过程，**仅限在for循环内使用。**
在 continue 语句后添加标签时，表示开始标签对应的循环。例如：

```
func continueDemo() {
forloop1:
    for i := 0; i < 5; i++ {
        // forloop2:
        for j := 0; j < 5; j++ {
            if i == 2 && j == 2 {
                continue forloop1
            }
            fmt.Printf("%v-%v\n", i, j)
        }
    }
}
```

练习题

1.编写代码打印9*9乘法表。

Go语言之数组、切片、map

Array（数组）

数组是同一种数据类型元素的集合。 在Go语言中，数组从声明时就确定，使用时可以修改数组成员，但是数组大小不可变化。 基本语法：
例：

```
// 定义一个长度为3元素类型为int的数组a
var a [3]int
```

数组定义

```
var 数组变量名 [元素数量(数组长度)]type
```

比如：var a [5]int， 数组的长度必须是常量，并且长度是数组类型的一部分。一旦定义，长度不能变。[5]int和[10]int是不同的类型。

```
var a [3]int
var b [4]int
a = b //不可以这样做，因为此时a和b是不同的类型
```

数组可以通过下标进行访问，下标是从 **0** 开始，最后一个元素下标是：**len-1**，访问越界（下标在合法范围之外），则触发访问越界，会panic（抛出异常）。

数组初始化

方法一

初始化数组时可以使用初始化列表来设置数组元素的值。

```
func main() {
    var testArray [3]int                            //数组会初始化为int类型的零值
    var numArray = [3]int{1, 2}                     //使用指定的初始值完成初始化
    var cityArray = [3]string{"北京", "上海", "深圳"} //使用指定的初始值完成初始化
    fmt.Println(testArray)                          //[0 0 0]
    fmt.Println(numArray)                           //[1 2 0]
    fmt.Println(cityArray)                          //[北京 上海 深圳]
}
```

方法二

按照上面的方法每次都要确保提供的初始值和数组长度一致，一般情况下我们可以让编译器根据初始值的个数自行推断数组的长度，例如：

```
func main() {
    var testArray [3]int
    var numArray = [...]int{1, 2}
    var cityArray = [...]string{"北京", "上海", "深圳"}
    fmt.Println(testArray)                          //[0 0 0]
    fmt.Println(numArray)                           //[1 2]
    fmt.Printf("type of numArray:%T\n", numArray)   //type of numArray:[2]int
    fmt.Println(cityArray)                          //[北京 上海 深圳]
    fmt.Printf("type of cityArray:%T\n", cityArray) //type of cityArray:
[3]string
}
```

方法三

我们还可以使用指定索引值的方式来初始化数组，例如:

```
func main() {
    a := [...]int{1: 1, 3: 5}
    fmt.Println(a)                  // [0 1 0 5]
    fmt.Printf("type of a:%T\n", a) //type of a:[4]int
}
```

数组的遍历

遍历数组a有以下两种方法:

```
func main() {
    var a = [...]string{"北京", "上海", "深圳"}
    // 方法1：for循环遍历
    for i := 0; i < len(a); i++ {
        fmt.Println(a[i])
    }

    // 方法2：for range遍历
    for index, value := range a {
        fmt.Println(index, value)
    }
}
```

多维数组

Go语言是支持多维数组的，我们这里以二维数组为例（数组中又嵌套数组）。

二维数组的定义

```
func main() {
    a := [3][2]string{
        {"北京", "上海"},
        {"广州", "深圳"},
        {"成都", "重庆"},
    }
    fmt.Println(a) //[[北京 上海] [广州 深圳] [成都 重庆]]
    fmt.Println(a[2][1]) //支持索引取值:重庆
}
```

二维数组的遍历

```
func main() {
    a := [3][2]string{
        {"北京", "上海"},
        {"广州", "深圳"},
        {"成都", "重庆"},
    }
    for _, v1 := range a {
        for _, v2 := range v1 {
            fmt.Printf("%s\t", v2)
        }
        fmt.Println()
    }
}
```

输出：

```
北京    上海
广州    深圳
成都    重庆
```

注意： 多维数组**只有第一层**可以使用...来让编译器推导数组长度。例如：

```
//支持的写法
a := [...][2]string{
    {"北京", "上海"},
    {"广州", "深圳"},
    {"成都", "重庆"},
}
//不支持多维数组的内层使用...
b := [3][...]string{
    {"北京", "上海"},
    {"广州", "深圳"},
    {"成都", "重庆"},
}
```

数组是值类型

数组是值类型，赋值和传参会复制整个数组。因此改变副本的值，不会改变本身的值。

```
func modifyArray(x [3]int) {
    x[0] = 100
}

func modifyArray2(x [3][2]int) {
    x[2][0] = 100
}
func main() {
    a := [3]int{10, 20, 30}
    modifyArray(a) //在modify中修改的是a的副本x
    fmt.Println(a) //[10 20 30]
    b := [3][2]int{
        {1, 1},
        {1, 1},
```

```
        {1, 1},
    }
    modifyArray2(b) //在modify中修改的是b的副本x
    fmt.Println(b)  //[[1 1] [1 1] [1 1]]
}
```

注意：

- 数组支持 “==“、”!=” 操作符，因为内存总是被初始化过的。
- [n]*T表示指针数组（”指针的数组”：每一个元素是指针类型），*[n]T表示数组指针 （”数组的指针”：该指针存放数组的首地址）。

数组练习题

1.求数组[1, 3, 5, 7, 8]元素之和
2.找出数组中和为指定值(给定)的两个元素的下标，比如从数组[1, 3, 5, 7, 8]中找出和为8的两个元素的下标分别为(0,3)和(1,2)。

切片

Go 语言切片是对数组的一种抽象。

Go 数组的长度不可改变，在特定场景中就不太适用，Go 中提供了一种灵活，功能强悍的内置类型：切片(“动态数组”)，与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

需要说明，slice 并不是数组或数组指针。它通过内部指针和相关属性 引用数组片段 ，以实现变长方案。

- 切片：切片是数组的一个引用，因此切片是引用类型。但自身是结构体，值拷贝传递。
- 切片的长度可以改变，因此，切片相当于一个可变的数组。
- 切片遍历方式和数组一样，可以用 len() 求长度。表示可用元素数量，读写操作不能超过该限制。
- cap 可以求出 slice 最大扩张容量，不能超出数组限制。0 <= len(slice) <= len(array)，其中array是slice引用的数组。
- 切片的定义：var 切片变量名 []类型，比如 var str []string 、 var arr []int。
- 如果 slice == nil，那么 len、cap 结果都等于 0。
  切片 Slice 在源码中的数据结构定义如下:

```
type slice struct {
    array unsafe.Pointer  //一个指向数组的指针
    len   int
    cap   int
}
```

切片的结构体由3部分构成，Pointer 是指向一个数组的指针，len 代表当前切片的长度，cap 是当前切片的容量。cap 总是大于等于 len 的。

总结：
Go 语言中的切片类型是从数组类型基础上发展出来的新类型，当声明一个数组时，不指定该数组长度，则该类型为切片(“动态数组”)，切片有自己独立的内部结构字段(len, cap, array pointer)，并于其引用的底层数组共用存储空间。

切片定义

方法一

你可以 声明 一个** 未指定大小的数组 **来定义**切片**，切片声明时不需要说明长度（[]没有声明长度，说明这是一个切片，而不是一个数组。因为数组声明是必须指定长度的。）：

```
var identifier []type
```

如上这种形式的只声明不初始化，这时切片 默认 初始化为 **nil：len=0 cap=0 slice=[]**
之所以为 nil ，是因为 没有分配存储空间。

实例：一个切片在未初始化之前默认为 nil，长度为 0：

```
func main() {
    var numbers []int  //切片定义，并未进行初始化

    printSlice(numbers)

    if numbers == nil {
        fmt.Printf("切片是空的")
    }
}

func printSlice(x []int) {
    fmt.Printf("len=%d cap=%d slice=%v\n", len(x), cap(x), x)
}
```

输出结果：

```
len=0 cap=0 slice=[]
切片是空的
```

提示：
nil 切片被用在很多标准库和内置函数中，描述一个不存在的切片的时候，就需要用到 nil 切片。比如函数在发生异常的时候，返回的切片就是 nil 切片。nil 切片的指针指向 nil。
空切片一般会用来表示一个空的集合。比如数据库查询，一条结果也没有查到，那么就可以返回一个空切片。

方法二

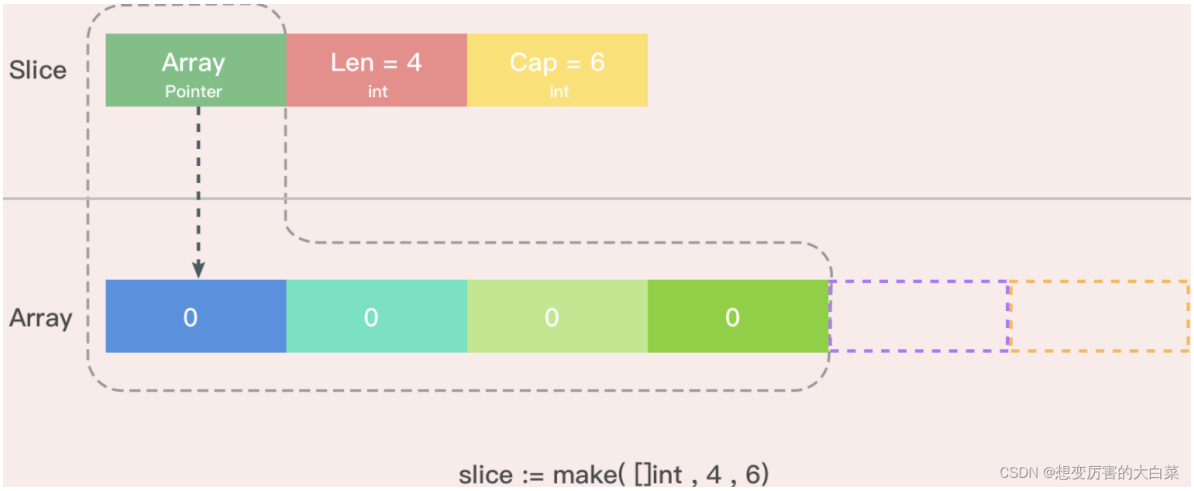
如果你想声明一个拥有初始长度或规定容量的切片（可以指定切片的长度和容量），可以使用 make() 函数来创建切片:

```
var slice1 []type = make([]type, length, capacity)

也可以简写为

slice1 := make([]type, length, capacity)
```

这里 length 是数组的长度并且也是切片的初始长度。
容量 capacity 为可选参数（可选的意思是可以缺省，如果不指定capacity，则capacity默认等于length）。
make 创建的切片与其底层数组：



实例：缺省 capacity

```
func main() {
    numbers := make([]int, 3)
    printSlice(numbers)
}

func printSlice(x []int) {
    fmt.Printf("len=%d cap=%d slice=%v\n", len(x), cap(x), x)
```

输出结果：

```
len=3 cap=3 slice=[0 0 0]
```

如上这样创建切片、不初始化，那么切片被系统自动初始化为 0。（而不是 nil ）
之所以 不是nil ， 是因为 make 函数为其 分配了内存空间。

总结：创建切片的各种方式

实例：

```
package main

import "fmt"

func main() {
    //1.声明切片
    var s1 []int
    if s1 == nil {
        fmt.Println("s1是空")
    } else {
        fmt.Println("s1不是空")
    }

    // 2.make()创建
    var s2 []int = make([]int, 0)
    var s3 []int = make([]int, 0, 0)
```

```
    if s2 == nil {
        fmt.Println("s2是空")
    } else {
        fmt.Println("s2不是空")
    }
    if s3 == nil {
        fmt.Println("s3是空")
    } else {
        fmt.Println("s3不是空")
    }

    fmt.Println(s1, s2, s3)

    // 3.:=
    s4 := []int{}
    s5 := []int{1, 2, 3} //创建切片并初始化
    fmt.Println(s4, s5)
    if s4 == nil {
        fmt.Println("s4是空")
    } else {
        fmt.Println("s4不是空")
    }

    // 4.从数组切片
    arr := [5]int{1, 2, 3, 4, 5} //数组
    var s6 []int                 //切片
    s6 = arr[1:4]   //索引范围（左包含，右不包含），也就是下面代码中从数组arr中选出1<=索引值<4的元素组成切片s6
    fmt.Println(s6)

}
```

输出结果：

```
s1是空
s2不是空
s3不是空
[] [] []
[] [1 2 3]
s4不是空
[2 3 4]
```

有四种创建切片的方法：

- 常规声明
- make() 函数创建
- 赋值符 := 创建
- 引用数组

其中，只有 “常规声明” 却不初始化的切片被系统默认为 nil （没有内存空间）。用 make() 函数或 := 创建却不初始化的切片为空切片（拥有内存空间，只是没有元素），如果有元素的话会被系统默认初始化为 0 。这是因为 “常规声明” 不会为切片分配存储空间，而其他方法会分配。

空切片和 nil 切片的区别在于，空切片指向的地址不是 nil，指向的是一个内存地址，但是它没有分配任何内存空间，即底层元素包含0个元素。

最后需要说明的一点是。不管是使用 nil 切片还是空切片，对其调用内置函数 append，len 和 cap 的效果都是一样的。

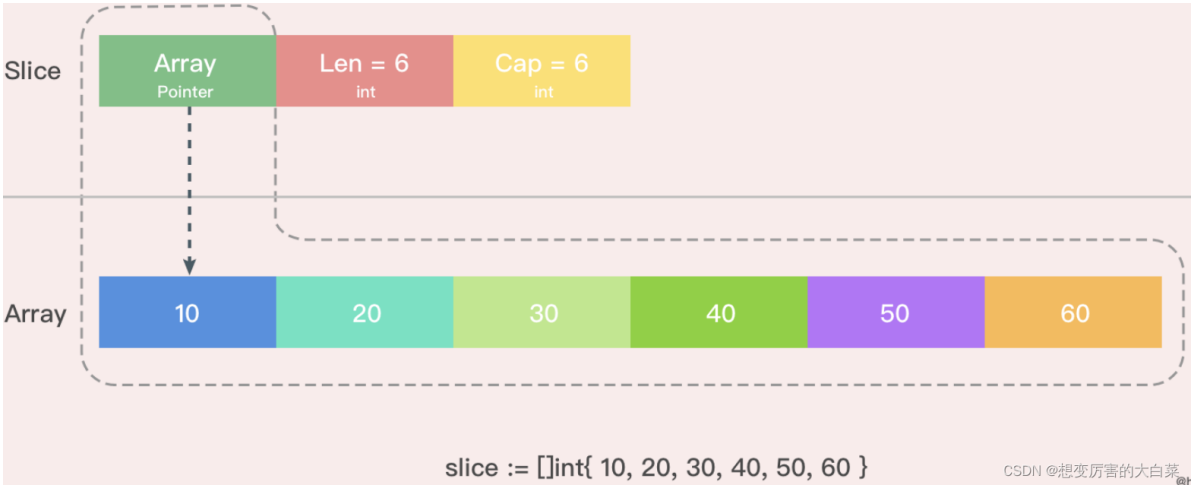
切片初始化

声明的同时初始化

```
s := []int{1, 2, 3}    //一句代码完成声明和初始化两个工作
```

直接完成了声明和初始化切片，[] 表示是切片类型，{1,2,3} 初始化值依次是 1,2,3，其 cap=len=3：len=3 cap=3 slice=[1 2 3]

赋值符初始化的切片与其底层数组：



用数组初始化切片

初始化切片 s，是数组 arr 的引用：

```
//用数组arr的所有值初始化切片
s := arr[:]
```

将 arr 中从下标 startIndex 到 endIndex-1 下的元素创建为一个新的切片：

```
s := arr[startIndex:endIndex]
```

默认 endIndex 时将表示一直到arr的最后一个元素：

```
s := arr[startIndex:]
```

默认 startIndex 时将表示从 arr 的第一个元素开始：

```
s := arr[:endIndex]
```

通过切片 s 初始化切片 s1：

```
s1 := s[startIndex:endIndex]
```

用数组初始化切片的方法总结：

```
全局变量：
var arr = [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} //这是一个数组
var slice0 []int = arr[start:end]
var slice1 []int = arr[:end]
var slice2 []int = arr[start:]
var slice3 []int = arr[:]
var slice4 = arr[:len(arr)-1]       //去掉切片的最后一个元素
局部变量：
arr2 := [...]int{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}  //这是一个数组
slice5 := arr[start:end]
slice6 := arr[:end]
slice7 := arr[start:]
slice8 := arr[:]
slice9 := arr[:len(arr)-1] //去掉切片的最后一个元素
```

操作	含义
s[n]	切片 s 中索引位置为 n 的项
s[:]	从切片 s 的索引位置 0 到 len(s)-1 处所获得的切片
s[low:]	从切片 s 的索引位置 low 到 len(s)-1 处所获得的切片
s[:high]	从切片 s 的索引位置 0 到 high 处所获得的切片，len=high
s[low:high]	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low
s[low:high:max]	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low，cap=max-low
len(s)	切片 s 的长度，总是<=cap(s)
cap(s)	切片 s 的容量，总是>=len(s)

切片的内存布局

读写操作实际目标是底层数组，只需注意索引号的差别。（本质：切片是数组的一个引用）

```
func main() {
    data := [...]int{0, 1, 2, 3, 4, 5}

    s := data[2:4]
    s[0] += 100
    s[1] += 200

    fmt.Println(s)
    fmt.Println(data)
}
```

输出结果：

```
[102 203]
[0 1 102 203 4 5]
```

可见：对切片内容的改变实际上改变的是它所引用的数组。
切片就像一个傀儡、一个指针、一个虚构，对它的操作就是对原数组的操作。切片和它所引用的数组是一体的，虽然我们看到的是一个切片，其实它还是底层的数组。它们两者是统一的，你就把切片当成一个原数组的一段就行。

这时有同学就有疑问了，前面的那么多创建切片的方式，并不都是通过引用数组得来的呀，大部分都是直接创建切片的呀？

这是因为，我们直接创建 slice 对象时，系统会自动分配底层数组

实例：

```
func main() {
    s1 := []int{0, 1, 2, 3, 8: 100} // 通过初始化表达式构造，可使用索引号。
    fmt.Println(s1, len(s1), cap(s1))

    s2 := make([]int, 6, 8) // 使用 make 创建，指定 len 和 cap 值。
    fmt.Println(s2, len(s2), cap(s2))

    s3 := make([]int, 6) // 省略 cap，相当于 cap = len。
    fmt.Println(s3, len(s3), cap(s3))
}
```

输出结果：

```
[0 1 2 3 0 0 0 0 100] 9 9
[0 0 0 0 0 0] 6 8
[0 0 0 0 0 0] 6 6
```

从这个实例可以看到，你创建的的确是切片，但是它还是数组的形式呀，并不是其他形式。
所以说切片的本质是数组，但是并不是数组，它只是引用了数组的一段。
你创建了一个切片，系统会自动为你创建一个底层数组，然后引用这个底层数组生成一个切片。
你觉得你操作的是切片本身，但实际上操作的是它所依托的那个底层的数组。

既然访问的还是底层数组，那我们为什么不直接操作数组呢？
这是因为切片长度可变的灵活性：使用 make 动态创建slice，**避免了数组必须用常量做长度的麻烦。**

比如：切片resize（调整大小）

```
func main() {
    var a = []int{1, 3, 4, 5}
    fmt.Printf("slice a : %v , len(a) : %v\n", a, len(a))
    b := a[1:2]
    fmt.Printf("slice b : %v , len(b) : %v\n", b, len(b))
    c := b[0:3]
    fmt.Printf("slice c : %v , len(c) : %v\n", c, len(c))
}
```

输出结果：

```
slice a : [1 3 4 5] , len(a) : 4
slice b : [3] , len(b) : 1
slice c : [3 4 5] , len(c) : 3
```

其原理仍然是：读写操作实际目标是底层数组。
但是这里用切片的话，它的长度就非常灵活。

一些复杂类型切片

1. [] []T，是指元素类型为 []T 的切片。(类似于二维数组)
   实例：

```
func main() {
    data := [][]int{              //[]int类型的切片
        []int{1, 2, 3},           //初始化值
        []int{100, 200},
        []int{11, 22, 33, 44},
    }
    fmt.Println(data)
}
```

输出结果：

```
[[1 2 3] [100 200] [11 22 33 44]]
```

2. 结构体数组 / 切片
   可直接修改 struct array/slice 成员：

```
func main() {
    d := [5]struct { //结构体数组
        x int
    }{}              //未初始化

    s := d[:]        //切片

    d[1].x = 10
    s[2].x = 20

    fmt.Println(d)
    fmt.Printf("%p, %p\n", &d, &d[0])

}
```

输出结果：

```
[{0} {10} {20} {0} {0}]
0xc00000c540, 0xc00000c540
```

len()、cap()、append()、copy()

len() 和 cap() 函数

可以使用 len() 方法获取切片长度。

计算切片容量的方法 cap() 可以测量切片最长可以达到多少。

以下为具体实例：

```
func main() {
    var numbers = make([]int, 3, 5)
    printSlice(numbers)
}

func printSlice(x []int) {
    fmt.Printf("len=%d cap=%d slice=%v\n", len(x), cap(x), x)
}
```

输出结果：

```
len=3 cap=5 slice=[0 0 0]
```

append() 函数

1.用 append 内置函数实现切片追加，实例：

```
func main() {

    var a = []int{1, 2, 3}
    fmt.Printf("slice a : %v\n", a)
    var b = []int{4, 5, 6}
    fmt.Printf("slice b : %v\n", b)
    c := append(a, b...)
    fmt.Printf("slice c : %v\n", c)
    d := append(c, 7)
    fmt.Printf("slice d : %v\n", d)
    e := append(d, 8, 9, 10)
    fmt.Printf("slice e : %v\n", e)

}
```

输出结果：

```
slice a : [1 2 3]
slice b : [4 5 6]
slice c : [1 2 3 4 5 6]
slice d : [1 2 3 4 5 6 7]
slice e : [1 2 3 4 5 6 7 8 9 10]
```

2.append 函数原理： 向 slice 尾部添加数据，返回新的 slice 对象。

```
func main() {

    s1 := make([]int, 0, 5)
    fmt.Printf("%p\n", &s1)
    fmt.Println(s1)

    s2 := append(s1, 1)
    fmt.Printf("%p\n", &s2)
    fmt.Println(s2)

}
```

输出结果：

```
0xc000004078
[]
0xc0000040a8
[1]
```

3.若给切片 append 的元素数量超出原 slice.cap 限制，就会重新给 slice 分配底层数组，并进行扩容：

```
func main() {

    data := [...]int{0, 1, 2, 3, 4, 10: 0} //数组
    s := data[:2:3]
    fmt.Println(s)
    fmt.Println(len(s), cap(s))

    s = append(s, 100, 200, 300) // 一次 append 三个值，超出 s.cap 限制。

    fmt.Println(s, data)          // 重新分配底层数组，与原数组无关。
    fmt.Println(&s[0], &data[0]) // 比对底层数组起始指针。

}
```

输出结果：

```
[0 1]
2 3
[0 1 100 200 300] [0 1 2 3 4 0 0 0 0 0 0]
0xc00000c570 0xc00004a060
```

从输出结果可以看出：append 后的 s 被重新分配了底层数组（也就是说 s 的底层数组不再是 data，那么修改 s 的值不会再影响 data，它们不再有关联），并把原数组中的值拷贝到新数组中。这是因为超出了原切片的容量。在上例中，如果只追加一个值，则不会超过 s.cap 限制，也就不会重新分配。

切片的自动扩容策略是这样的：
通常 以 2 倍容量 进行扩容，并重新分配底层数组（新底层数组的容量也变大）。
如果切片的容量小于 1024 个元素，扩容的时候就翻倍增加容量。一旦元素个数超过 1024 个元素，那么增长因子就变成 1.25 ，即每次增加原来容量的四分之一。
注意：扩容扩大的容量都是针对原来的容量而言的，而不是针对原来数组的长度而言的。

所以，在大批量添加数据时，建议 一次性分配足够大的空间 ，以减少内存分配和数据复制开销。或 初始化足够长的 len 属性，改用索引号进行操作。
及时释放不再使用的 slice 对象，避免持有过期数组，造成 GC 无法回收。

slice中 cap 重新分配规律：

```
func main() {

    s := make([]int, 0, 1)
    fmt.Println(s)

    c := cap(s)                          //计算容量
    fmt.Println(c)

    for i := 0; i < 50; i++ {
        s = append(s, i)                 //按理说 append 第2个元素时就超出了cap，这时会重新分配底层数组来扩大cap
        if n := cap(s); n > c {
            fmt.Printf("cap: %d -> %d\n", c, n)
            c = n
        }
    }

}
```

输出结果：

```
[]
1
cap: 1 -> 2
cap: 2 -> 4
cap: 4 -> 8
cap: 8 -> 16
cap: 16 -> 32
cap: 32 -> 64
```

我们可以发现，通常以 2 倍的 cap 重新分配。

提一嘴哈，如果给切片 append 元素时，不超切片容量就没事，操作的还是原数组：

```
func main() {

    data := [...]int{0, 1, 2, 3, 4, 10: 0} //数组
    s := data[:2:5]                         //将切片容量扩大到5
    fmt.Println(s)
    fmt.Println(len(s), cap(s))

    s = append(s, 100, 200, 300) // 一次 append 三个值，这次没超出 s.cap 限制。

    fmt.Println(s, data)
    fmt.Println(&s[0], &data[0]) // 比对底层数组起始指针

}
```

输出结果：

```
[0 1]
2 5
[0 1 100 200 300] [0 1 100 200 300 0 0 0 0 0 0]
0xc00004a060 0xc00004a060
```

copy() 函数（切片的深拷贝）

切片的拷贝分为2种，一种是浅拷贝，一种是深拷贝。
浅拷贝：源切片和目的切片共享同一底层数组空间，源切片修改，目的切片同样被修改。（赋值符实现）
深拷贝：源切片和目的切片各自都有彼此独立的底层数组空间，各自的修改，彼此不受影响。（使用内置函数copy()函数实现）

以下通过具体实例来说明：

浅拷贝：源切片和目的切片共享同一底层数组空间

```
func main(){
  slice1 := make([]int, 5, 5)
  slice2 := slice1
  slice1[1] = 1
  fmt.Println(slice1) //[0 1 0 0 0]
  fmt.Println(slice2) //[0 1 0 0 0]
}
```

深拷贝：源切片和目的切片各自都有彼此独立的底层数组空间

```
func main() {
    slice1 := make([]int, 5, 5)
    slice1[0] = 9
    fmt.Println(slice1)
    slice2 := make([]int, 4, 4)
    slice3 := make([]int, 5, 5)
    fmt.Println(slice2)
    fmt.Println(slice3)
    //拷贝
    fmt.Println(copy(slice2, slice1)) //4
    fmt.Println(copy(slice3, slice1)) //5
    //独立修改
    slice2[1] = 2
    slice3[1] = 3
    fmt.Println(slice1) //[9 0 0 0 0 0]
    fmt.Println(slice2) //[9 2 0 0]
    fmt.Println(slice3) //[9 3 0 0 0]
}
```

输出结果：

```
[9 0 0 0 0]
[0 0 0 0]
[0 0 0 0 0]
4
5
[9 0 0 0 0]
[9 2 0 0]
[9 3 0 0 0]
```

copy 函数的原理：copy 函数在两个 slice 间复制数据，复制长度以 len 小的为准。两个 slice 可指向同一底层数组，允许元素区间重叠。 \

切片遍历

```
func main() {

    data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    slice := data[:]
    for index, value := range slice {
        fmt.Printf("inde : %v , value : %v\n", index, value)
    }

}
```

输出结果：

```
inde : 0 , value : 0
inde : 1 , value : 1
inde : 2 , value : 2
inde : 3 , value : 3
inde : 4 , value : 4
inde : 5 , value : 5
inde : 6 , value : 6
inde : 7 , value : 7
inde : 8 , value : 8
inde : 9 , value : 9
```

字符串和切片（string and slice）

1.string 底层就是一个 byte 的数组，因此，也可以进行切片操作。

对字符串进行切片操作：

```
func main() {
    str := "hello world"
    s1 := str[0:5]
    fmt.Println(s1)

    s2 := str[6:]
    fmt.Println(s2)
}
```

输出结果：

```
hello
world
```

2.string本身是不可变的，因此要改变string中的字符。需要如下操作：

现在要改变英文字符串 “Hello world” 中的内容：

```
func main() {
    str := "Hello world"
    s := []byte(str) //将字符串类型转换成一个切片，中文字符需要用[]rune(str)
    fmt.Println(s)

    s[6] = 'G'
    s = s[:8]
    s = append(s, '!')
    fmt.Println(s)

    str = string(s) //将切片转换成字符串
    fmt.Println(str)
}
```

输出结果：

```
[72 101 108 108 111 32 119 111 114 108 100]
[72 101 108 108 111 32 71 111 33]
Hello Go!
```

改变中文字符串的内容：

```
func main() {
    str := "你好，世界！hello world！"
    s := []rune(str)
    s[3] = '集'
    s[4] = '美'
    s[12] = 'g'
    s = s[:14]
    str = string(s)
    fmt.Println(str)
}
```

输出结果：

```
你好，集美！hello go
```

对切片[x:y:z] 两个冒号的理解

1.常规切片截取（一个冒号）
可以通过设置下限及上限来设置截取切片 [lower_bound:upper_bound]。
截取的内容是下标从 lower_bound 到 upper_bound-1，示例如下：

```
func main() {
    /* 创建切片 */
```

```
    numbers := []int{0, 1, 2, 3, 4, 5, 6, 7, 8}
    printSlice(numbers)

    /* 打印原始切片 */
    fmt.Println("numbers ==", numbers)

    /* 打印子切片从索引1(包含) 到索引4(不包含)*/
    fmt.Println("numbers[1:4] ==", numbers[1:4])

    /* 默认下限为 0*/
    fmt.Println("numbers[:3] ==", numbers[:3])

    /* 默认上限为 len(s)*/
    fmt.Println("numbers[4:] ==", numbers[4:])

    numbers1 := make([]int, 0, 5)
    printSlice(numbers1)

    /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
    number2 := numbers[:2]
    printSlice(number2)

    /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
    number3 := numbers[2:5]
    printSlice(number3)

}

func printSlice(x []int) {
    fmt.Printf("len=%d cap=%d slice=%v\n", len(x), cap(x), x)
}
```

输出结果：

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

2.两个冒号的截取

```
func main() {
    slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    d1 := slice[6:8]
    fmt.Println(d1, len(d1), cap(d1))
    d2 := slice[:6:8]
    fmt.Println(d2, len(d2), cap(d2))
}
```

输出结果：

```
[6 7] 2 4
[0 1 2 3 4 5] 6 8
```

总结:
a[x:y:z] 切片内容是： [x:y] 切片长度，[x:z] 切片容量。长度计算：y-x，容量计算：z-x。

练习题

1.请写出下面代码的输出结果。

```
func main() {
    var a = make([]string, 5, 10)
    for i := 0; i < 10; i++ {
        a = append(a, fmt.Sprintf("%v", i))
    }
    fmt.Println(a)
}
```

2.请使用内置的sort包对数组var a = [...]int{3, 7, 8, 9, 1} 进行排序（附加题，自行查资料解答）。

map

map是一种无序的基于key-value的数据结构，Go语言中的map是引用类型，必须初始化才能使用。

map定义

Go语言中 map的定义语法如下：

```
map[KeyType]ValueType
```

其中，

- KeyType:表示键的类型。
- ValueType:表示键对应的值的类型。

map类型的变量默认初始值为nil，需要使用make()函数来分配内存。语法为：

```
make(map[KeyType]ValueType, [cap])
```

其中cap表示map的容量，该参数虽然不是必须的，但是我们应该在初始化map的时候就为其指定一个合适的容量。

map基本使用

map中的数据都是成对出现的，map的基本使用示例代码如下：

```
func main() {
    scoreMap := make(map[string]int, 8)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    fmt.Println(scoreMap)
    fmt.Println(scoreMap["小明"])
    fmt.Printf("type of a:%T\n", scoreMap)
}
```

输出结果：

```
map[小明:100 张三:90]
100
type of a:map[string]int
```

map也支持在声明的时候填充元素，例如：

```
func main() {
    userInfo := map[string]string{
        "username": "沙河小王子",
        "password": "123456",
    }
    fmt.Println(userInfo)
}
```

判断某个键是否存在

Go语言中有个判断map中键是否存在的特殊写法，格式如下：

```
value, ok := map[key]
```

举个例子：

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    // 如果key存在ok为true,v为对应的值；不存在ok为false,v为值类型的零值
    v, ok := scoreMap["张三"]
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("查无此人")
    }
}
```

map的遍历

Go语言中使用 for range 遍历map。

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    for k, v := range scoreMap {
        fmt.Println(k, v)
    }
```

但我们只想遍历key的时候，可以按下面的写法：

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    for k := range scoreMap {
        fmt.Println(k)
    }
}
```

但我们只想遍历value的时候，可以按下面的写法：

```
func main() {
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    for _,v := range scoreMap {
        fmt.Println(v)
    }
}
```

注意： 遍历 map 时的元素顺序与添加键值对的顺序无关。

使用delete()函数删除键值对

使用delete()内建函数从map中删除一组键值对，delete()函数的格式如下：

```
delete(map, key)
```

其中，

- map:表示要删除键值对的map名称
- key:表示要删除的键值对的键

示例代码如下：

```
func main(){
    scoreMap := make(map[string]int)
    scoreMap["张三"] = 90
    scoreMap["小明"] = 100
    scoreMap["娜扎"] = 60
    delete(scoreMap, "小明")//将小明:100从map中删除
    for k,v := range scoreMap{
        fmt.Println(k, v)
    }
}
```

使用delete()函数删除键值对

```
func main() {
    rand.Seed(time.Now().UnixNano()) //初始化随机数种子

    var scoreMap = make(map[string]int, 200)

    for i := 0; i < 100; i++ {
        key := fmt.Sprintf("stu%02d", i) //生成stu开头的字符串
        value := rand.Intn(100)          //生成0~99的随机整数
        scoreMap[key] = value
    }
    //取出map中的所有key存入切片keys
    var keys = make([]string, 0, 200)
    for key := range scoreMap {
        keys = append(keys, key)
    }
    //对切片进行排序
    sort.Strings(keys)
    //按照排序后的key遍历map
    for _, key := range keys {
        fmt.Println(key, scoreMap[key])
    }
}
```

练习题

1.写一个程序，统计一个字符串中每个单词出现的次数。比如：”how do you do”中how=1 do=2 you=1。