Golang语言学习笔记

> 本文适合有编程基础的人群，较为简单地叙述了Go语言的相关语法与操作，整理过程中有小部分是对他人优秀经验的摘抄，内容上不排除有细微错误的可能，后期还会持续更新更多关于Go语言操作的内容，欢迎读者指正。
>
> 作者：南京邮电大学硕士研究生——王纪涛
>
> 初步完成时间：2018年10月14日

第一章 Go语言编程环境搭建

1.1 Go语言安装包下载

安装包下载地址1：https://golang.org/dl/（该地址需要翻墙）

安装包下载地址2：https://golang.google.cn/dl/（该地址不需要翻墙）

- Window

在所给网站中，下载`.msi`后缀的安装包，直接双击安装即可。安装结束后，与配置Java JDK一样需要将Go语言的`bin`目录(全路径)添加到系统PATH环境变量中，最后运行`cmd`打开DOS命令行窗口，执行`go version`，若显示Go语言版本号说明环境搭建成功。

- MacOS
  - 源码安装

  在所给网站中，下载`go1.10.3.linux-amd64.tar.gz`，将该源码包解压。

  ```
  $ tar xzvf go1.10.3.linux-amd64.tar.gz -C /Users/Joten/
  ```

  最后将解压完的目录中的`bin`目录全路径添加到系统PATH环境变量中，在终端中执行`go version`进行验证。

  - pkg文件双击安装

  在所给网站中，下载`.pkg`结尾的安装包，直接双击安装即可。后续步骤与源码安装类似。

  - 利用brew工具安装

  我们也可以直接利用brew工具来安装我们的Go语言环境(前提是你的Mac下一家安装了brew工具)。

```
$ sudo brew install go
```

后续步骤类似。

- Linux

Linux环境下的安装，与MacOS的源码安装过程一样。

1.2 Go语言编辑工具

- GoLand(极力推荐)
- Sublime(推荐)
- VSCode
- Atom
- LiteIDE

1.3 第一个程序HelloWorld

本文使用`GoLand`作为编辑环境，下面我们来写一个HelloWorld程序对Go语言进行初步地接触。

```
package main

import "fmt"

func main()  {
    fmt.Println("HelloWorld!")
}
```

> 注意：在使用GoLand创建.go文件时，它会自动在第一行写上`package 你所在包名`，我们需要将其修改为`package main`，因为程序中只要包含`main`函数就必须这样填写。

第二章 Go语言基本语法

2.1 变量声明

- 利用var进行变量声明

```
var a <变量类型>
a = value0
var b <变量类型> = value1
```

> <>里的内容表示可写可不写。若不写，系统会根据对变量赋的值自行进行类型推断；若写，则为指定的变量类型。
>
> 声明变量时可以直接赋值，也可以声明之后单独对它进行赋值。

- 省略var进行变量声明

```
c := vlaue2
```

> 此方式只能在函数内部使用。

- 多个变量同时声明

```
var d, e, f <变量类型>
var d, e, f <变量类型> = value3, value4, value5

g, h := value6, value7
```

> 若写了变量类型，那需要赋对应的值。
>
> 再次强调：`:=`是声明变量，不是赋值！同时，多个变量用`:=`进行赋值时，左边的变量至少有一个新的变量，否则报错。
>
> Go语言是强类型语言，即变量类型确定后就是固定的。

- 舍弃变量(常用于函数返回值)

```
_, number := 1, 2
```

> `_`表示舍弃数值，即将1赋值给`_`。

2.2 常量声明

- 一般声明

```
const PI float64 = 3.14    //显示定义
const NUM = 3 //隐示定义
```

- 常量组声明

```
const (
    x int = 222
    y
    z
    s string = "sssss"
    l
)
```

> 注意：在常量组里没有赋值的常量的值是上一行等号右边的值。

- iota

`iota`是一个特殊的常量，每当有一个`const`，`iota`的值就会初始化为0，每当增加一个常量值，`iota`就会累加一。

```
const (
    a = iota      //a = 1
    b = iota      //b = 2
    c = iota      //c = 3
)

const (
    d = iota      // d = 1
    e = iota      // e = 2
)
```

2.3 数据类型

- bool类型
- 整型
  - 有符号整型

    `int`、`int8`、`int16`、`int32`、`int64`
  - 无符号整型

    `uint`、`uint8`、`uint16`、`uint32`、`uint64`
  - 特殊别名

    `byte`类似于`uint8`

    `rune`类似于`uint32`
- 浮点类型
  - float类型
  - double类型
- string类型

2.4 格式化输出(Printf)

- `%d`：整数
- `%f/%.2f`：浮点数/保留两位小数
- `%s`：字符串
- `%v`：原始类型输出
- `%q/%c`：根据ASCII码将数值以字符输出，前者带引号输出，后者不带引号

2.5 强制类型转换

由于Go语言是一种强类型语言，因此不同类型的数据不能进行相互赋值。但有些时候我们需要将两种不同类型的数据进行操作，例如将`int8`类型的变量与`int`类型的变量进行相加。此时我们就可以利用类型转换：

```
var a int8 = 3
var b int = 4
sum := a + b      //报错
sum := int(a) + b   //不报错
```

> 注意：当我们对不同类型的常数进行操作时，系统会自动帮我们进行数据类型转换，例如`sum := 4 + 2.1`，这是可行的。

2.6 运算符

2.6.1 算术运算符

- `+`：加法运算符
- `-`：减法运算符
- `*`：乘法运算符
- `/`：取整运算符
- `%`：取余运算符
- `++`：对变量加一
- `--`：对变量减一

代码演示：

```
package main

import "fmt"

func main()  {
    a := 999
    b := 111

    sum := a + b
    sub := a - b
    mul := a * b
    div := a / b
    mod := a % b

    fmt.Println(sum)
    fmt.Println(sub)
    fmt.Println(mul)
    fmt.Println(div)
    fmt.Println(mod)

    a++
    b--
    fmt.Println(a)
    fmt.Println(b)
}
```

> 注意：在Go语言中，只要后++/—，这与C/C++语言不同。

2.6.2 关系运算符

- `>`：大于
- `<`：小于
- `>=`：大于等于
- `<=`：小于等于
- `==`：等于
- `!=`：不等于

代码演示：

```
package main

import "fmt"

func main()  {
    a := 999
    b := 111
    c := a > b
    fmt.Println(c)

    d := a <= b
    fmt.Println(d)

    e := a == b
    fmt.Println(e)

    f := a != b
    fmt.Println(f)
}
```

2.6.3 逻辑运算符

- `&&`：逻辑与
- `||`：逻辑或
- `!`：逻辑非

代码演示：

```
package main

import "fmt"

func main()  {
    a := true
    b := false
    c := a && b
```

```
    fmt.Println(c)    //false

    d := a || b
    fmt.Println(d)    //true

    e := !a
    fmt.Println(e)    //false
}
```

> 注意：逻辑与运算符的左侧若是`false`，那右侧不执行；逻辑或运算符的左侧若是`true`，那右侧不执行。这种现象称为`短路`。

2.6.4 位运算符

- `&`：按位与
- `|`：按位或
- `^`：按位异或
- `>>`：右移运算符
- `<<`：左移运算符

代码演示：

```
package main

import "fmt"

func main()  {
    a := 9     //1001
    b := 5     //0101
    c := a & b
    fmt.Println(c)       //1

    d := a | b
    fmt.Println(d)       //13

    e := a ^ b
    fmt.Println(e)       //12

    a = a << 1     //0000 1001 --> 0001 0010    18
    b = b >> 1     //0000 0101 --> 0000 0010    2

    fmt.Println(a)
    fmt.Println(b)
}
```

2.6.5 赋值运算符

Go语言里赋值运算符有：`=`、`+=`、`-=`、`/=`、`%=`、`&=`、`|=`、`<<=`、`>>=`等

> `a += b`就相当于`a = a + b`

2.6.6 运算符优先级

优先级	运算符
7	^、！、++、--
6	*、/、%、<<、>>、&、&^
5	+、-、|^
4	==、！=、<、<=、>=、>
3	<-
2	&&
1	||

2.7 条件语句

2.7.1 if语句

基本语法一：

```
if 条件表达式1 {
    执行内容1
}else if 条件表达式2 {
    执行内容2
}else if 条件表达式3 {
    执行内容3
}
……
}else {
    执行内容n
}
```

代码演示：

```
package main

import "fmt"

func main()  {
    sex := "男"

    if sex == "男" {
        fmt.Println("This is a man! ")
    } else {
```

```
        fmt.Println("This is a woman!")
    }

}
```

基本语法二：

```
if 初始化语句;条件 {
    执行内容
}
```

代码演示：

```
package main

import "fmt"

func main()  {

    if b := 999; b > 900 {
        fmt.Println(b,"大于900")
    }

}
```

> 注意：在if语句里声明变量，该变量只能在if大括号内使用，即其作用域在if语句内部。

2.7.2 小插曲(键盘输入)

- `fmt.Scanln()`
- `fmt.Scanf()`（格式化输入）

代码演示：

```
package main

import "fmt"

func main()  {

    var x int
    var s string
    fmt.Println("请输入数据：")
    fmt.Scanln(&x, &s)
    fmt.Println(x,",",s)

    var y int
    var l string
```

```
    fmt.Println("请输入数据(数值,字符串)：")
    fmt.Scanf("%d,%s", &y,&l)
    fmt.Println(y,",",l)
}
```

> 注意：`&变量`代表变量的地址。

2.7.3 switch语句

基本语法一：

```
switch 变量 {
    case 值1:
        执行内容1
    case 值2:
        执行内容2
    case 值3:
        执行内容3
    ……
    case 值n:
        执行内容n
    default:
        执行内容d
}
```

代码演示：

```
package main

import "fmt"

func main()  {

    var x int
    fmt.Println("请输入整数：")
    fmt.Scanln(&x)

    switch x {
    case 1:
        fmt.Println("这是1")
    case 2:
        fmt.Println("这是2")
    case 3:
        fmt.Println("这是3")
    default:
        fmt.Println("这谁都不是")
    }
}
```

> 注意：在go语言中switch语句匹配到相应case分支后，只会执行该case分支后的内容，这与C/C++不同。

基本语法二：

我们可以在case后面填写多个值，只要与其中一个值匹配就执行该case语句后面的内容。

```
switch 变量 {
    case 值1,值2,值3,...,值n:
        执行内容
    ……
    case ……
    default ……
}
```

基本语法三：

我们也可以在`swicth`语句后面加`初始化语句`。

```
switch 初始化语句;变量 {
    case 值1:
        执行内容1
    case 值2:
        执行内容2
    case 值3:
        执行内容3
    ……
    case 值n:
        执行内容n
    default:
        执行内容d
}
```

基本语法四：

在go语言的`switch`语句中，我们可以不写匹配变量，此时系统默认去匹配case后为`true`的分支，因此case后面的条件表达式只能有一个是true。

```
switch {
    case 条件表达式1:
        执行内容1
    case 条件表达式2:
        执行内容2
    ……
    default:
        执行内容d
}
```

2.7.4 fallthrough语句

`fallthrough`语句用于在switch语句中`向下穿透执行`。这是因为go语言的switch语句匹配成功后只会执行所匹配的case后的内容，通过此语句可以实现跟C/C++中的switch语句一样的执行流程。

代码演示：

```
package main

import "fmt"

func main()  {

    var x int
    fmt.Println("请输入整数：")
    fmt.Scanln(&x)

    switch x {
    case 1:
        fmt.Println("这是1")
    case 2:
        fmt.Println("这是2")
        fallthrough
    case 3:
        fmt.Println("这是3")
    default:
        fmt.Println("这谁都不是")
    }
    //结果为
    //这是2
    //这是3
}
```

> 以上代码，当x的值为2时，程序执行完`case 2`后的内容，还会继续执行`case 3`的内容。

2.8 循环语句

2.8.1 for循环

基本语法：

```
for 表达式1; 表达式2; 表达式3 {
    循环体
}

or

//死循环
for {
    循环体
}
```

> 表达式1：作为变量初始化
>
> 表达式2：循环条件(不写表示`true`)
>
> 表达式3：循环体每执行完一次执行

2.8.2 break&continue

break语句

- 跳出`switch`语句
- 跳出`整个`循环语句

continue语句

在循环体中，遇到`continue`语句，则结束`本次`循环，进行`下一次`循环。

2.8.3 goto语句

利用`goto`语句可以无条件地转移到程序中指定的行。

代码演示：

```
package main

import "fmt"

func main() {
    var x int = 10

LOOP:
    for x < 20 {
        if x == 15 {
            x++
            goto LOOP      //当满足x等于15，则执行goto语句到LOOP
        }

        fmt.Println(x)
        x++
    }
}
```

2.8.4 找素数程序

代码演示：

```
package main

import (
    "math"
    "fmt"
)
```

```
func main() {
    for i := 2; i <= 100; i++ {
        flag := true

        for j := 2; float64(j) < math.Sqrt(float64(i)); j++ {
            if i % j == 0 {
                flag = false
                break
            }
        }

        if flag == true {
            fmt.Println(i,"是素质")
        }
    }
}
```

2.8.5 随机数

在go语言中，产生随机数需要设置`种子数`，若不设置，那产生的随机数是一样的。

```
package main

import (
    "math/rand"
    "fmt"
    "time"
)

func main() {

    randNumber1 := rand.Intn(100)

    fmt.Println(randNumber1)          //每次得到的随机数都一样

    t := time.Now()                   //获取当前时间
    timeStamp := t.Unix()             //将当前时间转换成时间戳
    rand.Seed(timeStamp)              //设置种子数

    randNumber2 := rand.Intn(100)
    fmt.Println(randNumber2)          //每次得到的随机数不一样

}
```

> 相同的种子数会生成同样的随机数，因此我们用时间戳来设置种子数，这样种子数是一直在改变的。

2.9 数组

2.9.1 数组声明

```
var arr1[5] int      //定义一个长度为5的数组
var arr2 = [5]int{1, 2, 3, 4, 5}    //定义一个长度为5的数组，并赋值
var arr3 = [4]int{3:999}  //定义一个长度为4的数组，并将下标为3的元素赋值
arr4 := [4]string{"www","jjj","ttt","ccc"}
arr5 := [...]int{2, 3, 4, 5, 6, 7} //三个点表示根据赋值的个数来确定长度
arr6 := [...]int{2:2, 8:8, 12:12}  //该数组长度为13
```

> 注意：在定义数组时，为赋值的元素默认为0;同一个数组内只能存储同种类型的元素。

2.9.2 数组遍历访问

代码演示：

```
package main

import "fmt"

func main() {

    var arr[5] int
    fmt.Println(arr)
    fmt.Println(len(arr))

    //数组赋值
    for i := 0; i < len(arr); i++ {
        arr[i] = i + 1
    }

    //访问数组
    for i := 0; i < len(arr); i++ {
        fmt.Println(arr[i])
    }
}
```

> - len()：长度，获得容器存储数据的实际数量
> - cap()：容量，获得容器存储数据的最大数量

利用range遍历：

`range`用来获取容器中的数据，直到访问完所有元素。

```go
package main

import "fmt"

func main() {

    arr := [...]string{"www", "jjj", "ttt", "ccc"}

    for i, v := range arr {
        fmt.Println(i, "---", v)
    }
}
```

> `range`具有两个返回值，一个是`数组下标`，一个其对应的`元素值`。因此在本程序中，用`i`来存储下标，用`v`来存储元素值。

利用range实现求和：

```go
package main

import "fmt"

func main() {

    arr := [...]int{1, 2, 3, 78, 67}
    sum := 0

    for _, v := range arr {
        sum += v
    }

    fmt.Println(sum)
}
```

> 求和时，range所返回的下标我们不需要用到，因此我们用`_`表示将其舍弃。

2.9.3 数组数据类型

数组在go语言中是`值类型`。

```
arr1 := [...]int{1, 2, 3}
arr2 := arr1
arr2[1] = 100      //此时 arr1:[1, 2, 3]  arr2:[1, 100, 3]
arr3 := [...]int{1, 2, 3}
arr4 := [...]string{"www", "jjj", "ttt"}

fmt.Println(arr1 == arr3)    //true，因为两个数组元素一样
fmt.Println(arr1 == arr2)    //false，两个元素不一样
fmt.Println(arr1 == arr4)    //系统报错，两个类型不一致
```

> 在将arr1赋值给arr2时，是将arr1的内容`拷贝`一份，赋值给arr2，因此数组是值类型。

2.9.4 二维数组

`二维数组`其实就是一个一维数组中的元素是一维数组。

```
package main

import "fmt"

func main() {

    arr := [3][4]int{
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}}

    fmt.Println(arr)          //输出为整个二维数组的元素
    fmt.Println(arr[0])       //输出二维数组中第一个一维数组
    fmt.Println(arr[1][1])    //输出二维数组中第二个一维数组的第二个元素
}
```

2.10 切片

2.10.1 什么是切片

本质上是一个指向底层数组的`引用`。

2.10.2 创建切片

```
package main

import (
    "fmt"
)

func main() {
```

```
    //一般方式创建
    slice1 := []int{1, 3, 5, 7}
    fmt.Println(slice1)
    fmt.Println("切片长度: ",len(slice1),"切片容量: ",cap(slice1))

    //使用make创建
    slice2 := make([]int, 2, 6)
    fmt.Println(slice2)
    fmt.Println("切片长度: ",len(slice2),"切片容量: ",cap(slice2))

    arr := [...]int{11, 22, 33, 44, 55, 66, 77}
    fmt.Println(arr)

    //通过现有数组创建切片
    slice3 := arr[1:4]
    fmt.Println(slice3)
    fmt.Println("切片长度: ",len(slice3),"切片容量: ",cap(slice3))
}
```

> 一般方式创建切片与创建数组的差别就在于`[]`内是否填写长度。
>
> 利用make创建切片，我们需要在设置`切片类型`、`切片长度`和`切片容量`三个参数，其中切片容量若不填写，默认与切片长度一致。
>
> 利用现有数组创建切片，其实就是从现有数组中`截取一部分`以赋值的方式来创建切片，此时的`切片长度是截取的长度`，而`切片容量是开始截取的位置到现有数组末尾的长度`。

2.10.3 切片扩容/删除元素

我们向切片中添加元素，可以通过`append()`函数想切片中添加。当切片容量不足时，系统会默认开辟一个原来切片容量两倍的新空间，使得我们的切片指向该空间，因此当切片容量不足时，切片会以原来的`两倍`扩容。

```
package main

import (
    "fmt"
)

func main() {

    slice1 := []int{}
    fmt.Println(len(slice1), cap(slice1))     //len:0   cap:0

    slice1 = append(slice1, 2)
    fmt.Println(len(slice1), cap(slice1))     //len:1   cap:1

    slice1 = append(slice1, 3)
    fmt.Println(len(slice1), cap(slice1))     //len:2   cap:2
```

```
    slice1 = append(slice1, 4)
    fmt.Println(len(slice1), cap(slice1))     //len:3   cap:4
}
```

go语言中没有专门删除切片元素的函数，因此我们可以用`append()`实现。

```
package main

import "fmt"

func main() {

    slice1 := []int{1, 2, 3, 4, 5, 6, 7, 8}
    fmt.Println(slice1)

    slice1 = append(slice1[:3], slice1[4:]...)
    fmt.Println(slice1)      //[1 2 3 5 6 7 8]
}
```

copy函数

由于切片是引用类型，因此我们只想要拷贝数据时，可以利用`copy`函数来进行数据拷贝。

```
package main

import "fmt"

func main() {

    slice1 := []int{1, 2, 3, 4, 5, 6, 7, 8}
    slice2 := []int{10, 20, 30}
    fmt.Println(slice1)      //[1 2 3 4 5 6 7 8]
    fmt.Println(slice2)      //[10 20 30]

    copy(slice1, slice2)
    fmt.Println(slice1)      //[10 20 30 4 5 6 7 8]
    fmt.Println(slice2)      //[10 20 30]

    copy(slice1[5:], slice2[:2])
    fmt.Println(slice1)      //[10 20 30 4 5 10 20 8]
    fmt.Println(slice2)      //[10 20 30]
}
```

2.11 Map

2.11.1 map基本操作

创建map基本语法：

```
var 名称 map[key类型]值类型    //此方法只是声明，没有创建，为nil
名称 ： = map[key类型]值类型{key1:value1, key2:value2,....}
名称 ： = make(map[key类型]值类型)
```

代码演示：

```
package main

import "fmt"

func main() {

    var map1 map[int]string
    map2 := map[string]int{"wjt":100, "jld":200, "wmj":300}
    map3 := make(map[int]string)

    fmt.Println(map1)
    fmt.Println(map2)       //map[wjt:100 jld:200 wmj:300]
    fmt.Println(map3)

    if map1 == nil {
        fmt.Println("map1为空")
        map1 = make(map[int]string)     //若map为nil可通过make创建
    }

    map1[1] = "wjt"             //只有创建之后才能添加键值对
    fmt.Println(map1)           ////map[1:wjt]

    val1, ok := map1[1]         //获取map值其实有两个返回值，ok为是否获取到值
    if ok == true {
        fmt.Println("获取的数据为：",val1)      //wjt
    }else {
        fmt.Println("获取数据不存在")
    }

    val1, ok = map1[5]
    if ok == true {
        fmt.Println("获取的数据为：",val1)
    }else {
        fmt.Println("获取数据不存在")
    }

    delete(map2, "wmj")             //删除键值对
    fmt.Println(map2)       //map[wjt:100 jld:200]

}
```

> 若用一个不存在的键值对的key获取map值，得到的是其数据类型的默认值，例如int类型就是0。

2.11.2 map遍历

对于map来说，它是没有下标的，因此不能通过下标来遍历map。但我们可以通过`for range`来遍历，通过`range`对map进行获取得到`key`和`value`。

代码演示：

```
package main

import "fmt"

func main() {

    map1 := map[string]int{"wjt":100, "jld":200, "wmj":300}

    for k, v := range map1 {
        fmt.Println(k, v)
    }

}
```

> 注意：用range来遍历map的结果是`无序`的。

2.12 字符串

2.12.1 字符串声明/输出

```
package main

import (
    "fmt"
)

func main() {

    s1 := "wjt"
    s2 := "王纪涛"

    fmt.Println(s1, len(s1))      //wjt 3
    fmt.Println(s2, len(s2))      //王纪涛 9

    fmt.Printf("%q, %c, %d\n", s1[0], s1[0], s1[0])

    for i := 0; i < len(s1); i++ {
        fmt.Printf("%q, %d\n", s1[i], s1[i])
    }
```

```
    for i, v := range s1 {
        fmt.Println(i, v)         //输出下标和值
    }
}
```

2.12.2 字符串常用函数

- Contains(s, substr string)

判断在s中是否包含substr string。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Contains("seafood", "foo"))   //true
    fmt.Println(strings.Contains("seafood", "bar"))   //false
    fmt.Println(strings.Contains("seafood", ""))    //true
    fmt.Println(strings.Contains("", ""))       //true
}
```

- ContainsAny(s, chars string)

判断chars string中的任意一个字符是否在s中出现。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ContainsAny("team", "i"))    //false
    fmt.Println(strings.ContainsAny("failure", "u & i"))   //true
    fmt.Println(strings.ContainsAny("foo", ""))    //false
    fmt.Println(strings.ContainsAny("", ""))     //false
}
```

- Index(s, substr string)

返回substr string首字符在s中出现的位置下标，没有则返回-1。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Index("chicken", "ken"))    //4
    fmt.Println(strings.Index("chicken", "dmr"))    //-1
}
```

- LastIndex(s, substr string)

返回substr string在s中最后一次出现的位置，没有则返回-1。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Index("go gopher", "go"))    //0
    fmt.Println(strings.LastIndex("go gopher", "go"))   //3
    fmt.Println(strings.LastIndex("go gopher", "rodent"))  //-1
}
```

- Split(s, sep string)

根据sep string对s进行切割，返回一个字符串切片。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    //["a" "b" "c"]
    fmt.Printf("%q\n", strings.Split("a,b,c", ","))
    //["" "man " "plan " "canal panama"]
    fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
    //[" " "x" "y" "z" " "]
    fmt.Printf("%q\n", strings.Split(" xyz ", ""))
    //[""]
    fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
```

```
}
```

- HasSuffix(s, suffix string)

判断s是否以suffix string结尾。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.HasSuffix("Amigo", "go"))   //true
    fmt.Println(strings.HasSuffix("Amigo", "O"))    //false
    fmt.Println(strings.HasSuffix("Amigo", "Ami"))  //false
    fmt.Println(strings.HasSuffix("Amigo", ""))   //true
}
```

- HasPrefix(s, prefix string)

判断s是否以prefix string开头，与HasSuffix类似。

- Count(s, substr string)

判断substr string在s中出现的次数。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Count("cheese", "e"))  //3
    fmt.Println(strings.Count("five", "")) // before & after each rune  5
}
```

> 更多的函数可以参考go语言API。
>
> Mac安装`Dash`(可下载各种API文档，很好用)。
>
> 其他系统可上go语言官网查询。

第三章 函数

3.1 定义函数语法

```
func 函数名(参数1, 参数2, ...) 返回值1,返回值2,... {

    代码体

    return 返回值1,返回值2
}
```

> go语言的函数返回值可以是多个。

3.2 函数调用

求和代码演示：

```
package main

import "fmt"

func main() {

    sum1 := 0
    sum1 = getSum(100)

    fmt.Println(sum1)        //5050

    sum2 := 0
    sum2 = getSum(1000)
    fmt.Println(sum2)        //500500

}


//定义求和函数
func getSum(n int) int {

    sum := 0

    for i := 1; i <= n; i++ {
        sum += i
    }

    return sum
}
```

> 当函数形参的类型一致，可简写。

```
func addNum(x, y int) int {  //表示x,y的类型都为int
    return x + y
}
```

3.3 返回值写法

另一种返回值写法：

```
func getSum() (sum int) {
    for i := 1; i < 100; i++ {
        sum += i
    }

    return              //此时return后可省略
}
```

3.4 可变参数

一个函数中，如果参数类型是确定的，但参数个数是未知的，称为这样的参数为`可变参数`。

基本语法：

```
func 函数名(参数名 ... 参数类型) {

}
```

代码演示：

```
package main

import "fmt"

func main() {

    getSum(1, 2, 3, 4, 5)
    test("wjt", "jld", 1, 2, 3, 4, 5)
}


func getSum(numbers ... int) {
    fmt.Println(numbers)
}

func test(s1, s2 string, numbers ... int) {
    fmt.Println(s1, s2, numbers)
}
```

> 注意：当我们的参数除了可变参数外，还有其他类型的参数，这些参数需要写在可变参数的前面。另外，一个函数中可变参数只能有`一个`。

3.5 参数传递

- 值传递(基本数据类型、数组)

将实参的副本拷贝给形参，即修改形参不影响实参。

- 引用传递(切片、map)

将实参的引用拷贝给形参，即修改形参会对实参产生影响(本质上，实参和形参所指向的内存地址是一样的)。

3.6 递归

所谓的递归函数就是函数自身调用自己。

- 递归函数需要有一个出口，即中止递归的条件。
- 递归函数使用背景新的问题与旧的问题形式一样。

递归实现阶乘：

```
package main

import "fmt"

func main() {

    f := 0
    f = factorial(5)
    fmt.Println(f)              //120

}

func factorial(n int) int {
    if n == 1 {
        return 1
    }else {
        return n * factorial(n-1)
    }
}
```

3.7 函数类型

我们可以定义函数类型变量，如下代码所示：

```
package main

import "fmt"

func main() {

    var f1 func(int) int
    f1 = factorial
```

```
    a := f1(6)
    fmt.Println(a)
}

func factorial(n int) int {
    if n == 1 {
        return 1
    }else {
        return n * factorial(n-1)
    }
}
```

> 代码实例中，我们定义了`f1`变量，它的类型是`factorial函数的类型`，我们可以直接将factorial函数赋值f1，此时f1变量就可以当factorial函数`一样`使用。

3.8 匿名函数

一般来说匿名函数只用一次，但我们可以将匿名函数赋给函数类型变量。

```
package main

import "fmt"

func main() {

    func () {
        fmt.Println("匿名函数")
    }()                          //()表示调用函数

    func (a, b int) {
        fmt.Println(a + b)
    }(3, 5)

    f := func (a, b, c int) {        //将匿名函数赋给f变量
        fmt.Println(a * b * c)
    }

    f(3, 4, 5)

}
```

3.9 函数作为参数

我们可以将函数作为函数的参数进行传递，这样可以实现某些特殊的效果。

```
package main
```

```
import "fmt"

func main() {

    //根据需求传递不同的函数
    fmt.Println(oper(88, 11, add))    //99
    fmt.Println(oper(88, 11, sub))    //77
    fmt.Println(oper(88, 11, mul))    //968
    fmt.Println(oper(88, 11, div))    //8
}
//两数相加
func add(a, b int) int {
    return a + b
}
//两数相减
func sub(a, b int) int {
    return a - b
}
//两数相乘
func mul(a, b int) int {
    return a * b
}
//两数相除
func div(a, b int) int {
    return a / b
}
//将函数作为参数
func oper(a, b int, fun func(int, int) int) int {
    return fun(a, b)
}
```

3.10 函数为返回值

前提知识：

当我们在某一个函数中定义了一个内层函数，并且内层函数使用了该函数的局部变量，若将内层函数作为该函数的返回值，那该函数的局部变量不会随着该函数的结束而销毁，这种结构叫作`闭包`。

```
package main

import "fmt"

func main() {

    fun := test()
    fmt.Println(fun())        //1
    fmt.Println(fun())        //2
```

```
    fun = test()                //此时又调用一次test函数，内部的x与之前的不是一个变量
    fmt.Println(fun())          //1

}

func test() func() int {

    x := 0

    fun := func() int {
        x++
        return x
    }

    return fun
}
```

> 闭包中的局部变量只有main结束才会消亡。

3.11 defer

`defer`用于延迟函数执行。

```
package main

import "fmt"

/*
结果:
222222
222222
111111
*/
func main() {

    defer print1()
    print2()
    print2()
}

func print1() {
    fmt.Println("111111")
}

func print2() {
    fmt.Println("222222")
}
```

> 注意：有多个函数被延迟执行时，`先延迟的后执行`。类似于栈，先进后出。
>
> 若被延迟的函数执行有参数传递，那是`先传递参数再被延迟`。

第四章 指针

4.1 什么是指针

`指针`是存储内存地址的变量。

4.2 指针操作

指针声明：

```
var 指针名 *类型
```

例子：

```
var p *int      //p存储int型变量的地址,未赋值前p==nil
```

指针基本操作：

```
package main

import "fmt"

func main() {

    a := 222
    var p *int
    var pp **int
    fmt.Println(p)
    p = &a
    pp = &p
    fmt.Println(a)          //222
    fmt.Println(*p)         //222
    fmt.Println(**pp)       //222
}
```

> `&a`表示获取a变量地址。
>
> `*p`表示获取p指向的地址内的数据。
>
> `**pp`表示获取pp指向的指针指向的地址内的数据。

4.3 深拷贝/浅拷贝

- 深拷贝：拷贝数值(值类型默认是深拷贝)

- 浅拷贝：拷贝地址(引用类型默认是浅拷贝)

4.4 指针作为参数

将指针作为函数参数时，我们传递参数时需要注意我们传的是否是值类型，若是值类型应该传递它的地址。

```
package main

import "fmt"

func main() {

    a := 999
    test(&a)
    fmt.Println(a)              // 12345

}

func test(a *int) {
    *a = 12345
}
```

4.5 数组指针/指针数组

- 数组指针：其本身就是一个指针，但它指向的是一个数组。

  ```
  var p *[4]int       //p为指向一个长度为4的整型数组的指针
  ```

- 指针数组：其本身就是一个数组，但数组内的元素为指针。

  ```
  var p [4]*int       //p为一个存储了4个指向整型变量地址的指针的数组
  ```

4.6 函数指针/指针函数

- 函数指针：其本身就是一个指针，但它指向一个函数。

  但go语言中，函数本身就可以作为变量赋值，`相当于`一个指针(不是指针)

- 指针函数：其本身就是一个函数，但它的返回值是一个指针。

  ```
  package main

  import "fmt"

  func main() {

      var p *int
  ```

```
    p = add(99, 111)
    fmt.Println(p)          //0xc420014080
    fmt.Println(*p)         //210

}

func add(a, b int) *int {

    a = a + b
    p := &a

    return p                    //返回了一个指向整型的指针
}
```

> 当返回的是数组时，用指针函数可以节省内存空间。

第五章 结构体

5.1 什么是结构体

`结构体`是由一系列具有相同类型或不同类型的数据构成的数据集合。

5.2 结构体声明

```
type 结构体名称 struct {
    属性一
    属性二
    ...
}
```

5.3 结构体基本操作

```
package main

import "fmt"

type Person struct {
    name string
    age int
    tel string
}

func main() {

    var p1 Person
    fmt.Println(p1)   //{ 0 }
```

```
    p1.name = "王纪涛"
    p1.age = 22
    p1.tel = "18860853109"

    fmt.Println(p1)     //{王纪涛 22 18860853109}

    p2 := Person{"wjt", 22, "11111111"}
    fmt.Println(p2)     //{wjt 22 11111111}

    //利用new可以返回一个数据类型的指针
    p3 := new(Person)
    (*p3).name = "qqq"
    (*p3).age = 45
    p3.tel = "44555666"
    fmt.Println(p3)        //&{qqq 45 44555666}
}
```

> go语言语法允许，`(*p).name`与`p.name`效果一样。(类似于C语言中`*p.name`与`p->name`效果一样)

5.4 匿名结构体/字段

5.4.1 匿名结构体

```
package main

import "fmt"

func main() {

    a := struct {
        name string
        age int
        tel string
    }{"www", 77, "18876775554"}

    fmt.Println(a)             //{www 77 18876775554}

}
```

5.4.2 匿名字段

```
package main

import "fmt"

type Person struct {
    string
```

```
    int
}
func main() {

    p := Person{"qqqq", 78}

    fmt.Println(p)              //{qqqq 78}
    fmt.Println(p.string, p.int)   //qqqq 78

}
```

> 匿名字段可以通过类型进行获取对应的值(类型就是属性名)，但在一个结构体中相同数据类型的匿名字段只能有一个。

5.5 结构体关系

5.5.1 模拟聚合关系

所谓的聚合关系就是一个类中包含着另一个类，也就是一个结构体中包含另一个结构体，例如一个人有一部手机，其中人可以是一个类，手机也可以是一个类。

```
package main

import "fmt"

type Phone struct {         //定义手机类
    price float64
    color string
    size float64
}

type Person struct {          //定义人类
    name string
    age int
    phone Phone
}
func main() {

    person1 := Person{}
    person1.name = "wjt"
    person1.age = 22
    person1.phone.price = 9680
    person1.phone.color = "white"
    person1.phone.size = 5.5

    fmt.Println(person1) //{wjt 22 {9680 white 5.5}}

    person2 := Person{"jld", 23, Phone{7777, "粉", 5.5}}
    fmt.Println(person2)  //{jld 23 {7777 粉 5.5}}
```

```
}
```

> 注意：go语言中并没有聚合关系，我们只是通过结构体进行模拟这类关系。

5.5.2 模拟继承关系

所谓的继承关系就是一个类可以使用另一个类的属性或方法。在Go语言中子类和父类满足以下关系：

- 子类可以直接访问父类的属性和方法
- 子类可以新增自己的属性和方法
- 子类可以重写父类已有的方法

```
package main

import "fmt"

//创建父类
type Animal struct {
    name string
    age int
}

//创建子类
type Dog struct {
    Animal              //利用匿名字段实现继承
}

func main() {

    d := Dog{Animal{"小黑", 2}}
    fmt.Println(d)      //{{小黑 2}}
    fmt.Println(d.name)     //小黑
}
```

第六章 方法

6.1 什么是方法

Go语言中既有函数，又有方法，但两者有所区别。方法是一种包含了`接收者`的函数。

6.2 方法声明

```
func (t Type) 方法名(参数列表)(返回值类型) {

}
```

代码演示：

```go
package main

import "fmt"


type Person struct {
    name string
    age int
}

func (p Person) eat() {                  //Person对象谁调用谁是p
    fmt.Println(p.name, "要吃饭。")
}

func (p *Person) eatA() {
    fmt.Println(p.name, "要吃苹果。")
}

func main() {

    person1 := Person{}
    person1.name = "wjt"
    person1.age = 22
    fmt.Println(person1)        //{wjt 22}

    person1.eat()          //wjt 要吃饭。
    p := &person1
    p.eatA()                //wjt 要吃苹果。
    person1.eatA()          //wjt 要吃苹果。
    p.eat()       //wjt 要吃饭。
}
```

> 注意：在声明方法时，接收者p就是当前的调用者。

6.3 继承中方法重写

所谓重写就是将父类原来的方法所执行的内容重写。此时子类调用该方法，则使用的是重写的方法。

```go
package main

import "fmt"

//创建父类
type Animal struct {
    name string
    age int
}
```

```go
func (a Animal) getName() {
    fmt.Println("我叫", a.name)
}

func (a Animal) eat() {
    fmt.Println("吃东西")
}

//创建子类
type Dog struct {
    Animal
}

//重写父类方法
func (d Dog) eat() {
    fmt.Println("吃骨头")
}

func main() {

    d := Dog{Animal{"小黑", 2}}
    fmt.Println(d)    //{{小黑 2}}
    d.getName()     //我叫 小黑
    d.eat()        //吃骨头
}
```

6.4 包与包的关系

Go语言中，不同的包之间操作数据会有一定限制。下面给出一个例子就会明白：

A包：

```go
package A

type Animal struct {
    Aname string        //共有属性
    age int           //私有属性
}

func (a Animal) GetInfo() {
    fmt.Println(a.Aname, a.age)
}
```

main包：

```go
package main

func main() {
    //d := Animal{"www", 2}      //程序报错，因为age是私有的
    d := Animal{"www"}
    d.GetInfo()       //www 0    可以间接访问私有属性
}
```

> Go语言中，首字母大写表示公有，小写表示私有。

第七章 接口

7.1 什么是接口

`接口`简单来说就是方法的集合，在接口中，我们只声明方法而不实现方法。也就是接口只关心能做什么，还不关心怎么做。

7.2 接口声明

```go
type 接口名 interface {
    方法名一(参数列表一) (返回值类型一)
    方法名二(参数列表二) (返回值类型二)
}
```

7.3 接口使用

`接口`常用于作为函数参数，来接收它的实现类对象，起到`解耦合`的作用。

```go
package main

import "fmt"

type GetInfo interface {       //定义获取信息接口
    getName() string
    getAge() int
}

type Person struct {          //定义人类
    name string
    age int
}

func (p Person) getName() string {    //人类实现getName方法
    return p.name
}

func (p Person) getAge() int {       //人类实现getAge方法
```

```
    return p.age
}

type Animal struct {       //定义动物类
    name string
    age int
}

func (a Animal) getName() string {     //动物类实现getName方法
    return a.name
}

func (a Animal) getAge() int {       //动物类实现getAge方法
    return a.age
}

func testGetInfo(gi GetInfo) {            //接口作为参数
    fmt.Println(gi.getName(), gi.getAge())
}

func main() {

    person1 := Person{"www", 20}
    fmt.Println(person1)     //{www 20}

    fmt.Println(person1.getName())    //www

    testGetInfo(person1)    //www 20

    animal1 := Animal{"小黑", 2}
    testGetInfo(animal1)   //小黑 2

}
```

> 上述代码，testGetInfo函数，既可以接收人类对象，又可以接收动物类对象，因为它们都是GetInfo接口的实现类。
>
> ```
> func testGetInfo(gi GetInfo) {            //接口作为参数
>     fmt.Println(gi.getName(), gi.getAge())
> }
> ```
>
> 其实当接口作为参数时，这可以说是一种`多态`。
>
> 实现类对象：
>
> - 可以看作本类对象，可以访问本类属性和方法。
> - 可以看作接口对象，仅可以访问方法，不能访问属性。

7.4 空接口

`空接口`就是在接口中不写任何方法，那么所有类型都是该接口的实现类。

```
package main

import "fmt"

type A interface {      //定义空接口

}

type Person struct {
    name string
}

type Animal struct {
    name string
}

func main() {

    var a A             //定义接口对象
    person := Person{"www"}
    animal := Animal{"jjj"}

    a = person
    fmt.Println(a)      //{www}
    a = animal
    fmt.Println(a)      //{jjj}

    arr := [4]A{}
    arr[0] = person
    arr[1] = animal
    arr[2] = "wjt"
    arr[3] = 10000
    fmt.Println(arr)      //[{www} {jjj} wjt 10000]
}
```

> `a`既可以接收`person`，也可以接收`animal`。
>
> 类型为`A`的arr数组里，可以存储`任何类型`的对象。

第八章 并发性(Concurrency)

8.1 什么是并发性

`并发`并不是并行，并发是在多个任务进行轮询，也就是在多个任务之间不断切换，当切换的速度很快时，就会让人感觉任务同时在执行。

Go语言它的高并发是最大的亮点。

Go语言通过`协程`来实现并发(超级轻量级线程，又称微线程)

8.2 Goroutines

8.2.1 什么是Goroutines

`Go`中使用`Goroutines`来实现并发。Goroutines是与其他函数或方法同时运行的函数或方法。Goroutines可以被认为是轻量级的线程。与线程相比，创建Goroutine的成本很小，它就是一段代码，一个函数入口。以及在堆上为其分配的一个堆栈（初始大小为4K，会随着程序的执行自动增长删除）。因此它非常廉价，Go应用程序可以并发运行数千个Goroutines。

Goroutines在线程上的优势

- 与线程相比，Goroutines非常便宜。它们只是堆栈大小的几个kb，堆栈可以根据应用程序的需要增长和收缩，而在线程的情况下，堆栈大小必须指定并且是固定的
- Goroutines被多路复用到较少的OS线程。在一个程序中可能只有一个线程与数千个Goroutines。如果线程中的任何Goroutine都表示等待用户输入，则会创建另一个OS线程，剩下的Goroutines被转移到新的OS线程。所有这些都由运行时进行处理，我们作为程序员从这些复杂的细节中抽象出来，并得到了一个与并发工作相关的干净的API。
- 当使用Goroutines访问共享内存时，通过设计的通道可以防止竞态条件发生。通道可以被认为是Goroutines通信的管道。

8.2.2 使用Goroutines

Go语言中用`go`来创建一个Goroutine，当函数结束时，Goroutine结束。

其中main函数系统自动创建Goroutine，并且当main函数的Goroutine结束，那main函数中的子Goroutine就不执行了。

代码演示：

```
package main

import "fmt"


/*
结果一：
jjjjjjjjjj
main
0
wwwwwwwwww
1
2
3
4

结果二：
jjjjjjjjjj
main
```

```
0
1
2
3
4
*/

func main() {

    go test1()
    test2()
    fmt.Println("main")
    test3()
}

func test1() {
    fmt.Println("wwwwwwwwwww")
}

func test2() {
    fmt.Println("jjjjjjjjjjj")
}

func test3() {
    for i := 0; i < 5; i++ {
        fmt.Println(i)
    }
}
```

> 对test1()函数创建了一个goroutine，因此test1()与main函数并发执行，结果每次都不一样。
>
> 子goroutine中的函数一般没有返回值，就算有返回值也会被舍弃。

8.3 waitgroup

在使用goroutine时，可能会因为主goroutine的结束，导致子goroutine不执行，因此我们可以用waitgroup来解决这个问题。

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup      //声明waitgroup
```

```
func main() {

    wg.Add(2)           //设置waitgroup的子goroutine数量为2
    go test1()
    go test2()
    fmt.Println("main进入阻塞")
    wg.Wait()             //让主goroutine阻塞，当waitgroup中count=0时唤醒
    fmt.Println("main结束")

}

func test1() {
    for i := 0; i< 1000; i++ {
        fmt.Println(i)
        time.Sleep(1)
    }
    wg.Done()              //该子goroutine结束 对waitgroup中count减一
}

func test2() {
    for i := 0; i < 1000; i++ {
        fmt.Println("\t\t\t",i)
        time.Sleep(1)
    }
    wg.Done()
}
```

> 若你的程序中有两个goroutine，而你设置waitgroup的子goroutine的数量为1时，在运行时可能会报错。这是因为可能在主goroutine唤醒之前，两个子goroutine都执行了`Done()`，使得waitgroup中的计数器减为了-1。

8.4 售票程序

```
package main

import (
    "fmt"
    "sync"
    "time"
    "math/rand"
)

var wg sync.WaitGroup           //创建等待组变量
var tickets int = 100           //票总数
var mutex sync.Mutex            //创建互斥信号量

func main() {
```

```
    wg.Add(4)
    go saleTicket(1)
    go saleTicket(2)
    go saleTicket(3)
    go saleTicket(4)
    wg.Wait()

}

func saleTicket(number int) {
    rand.Seed(time.Now().UnixNano())           //设置种子数
    for {

        mutex.Lock()                           //上锁
        if (tickets > 0) {

            time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
//睡眠
            fmt.Println("售票口",number,":",tickets)
            tickets--

            mutex.Unlock()       //解锁

        }else {
            fmt.Println("售票口", number, "无票了")

            mutex.Unlock()     //解锁
            break
        }
    }

    wg.Done()

}
```

> 注意：上锁的原因是四个售票口并发售票时，可能会出现某个售票口在售票的时候，其他售票口已经进入了if语句，例如当`tickets==1`时。这样会出现售票为`-1,-2,-3`的情况。

第九章 通道

9.1 什么是通道

通道可以被认为是Goroutines通信的管道。类似于管道中的水从一端到另一端的流动，数据可以从一端发送到另一端，通过通道接收。

9.2 声明通道

每个通道都有与其相关的类型。该类型是通道允许传输的数据类型。(通道的零值为nil。nil通道没有任何用处，因此通道必须使用类似于map和切片的方法来定义。)

示例代码：

```
package main

import "fmt"

func main() {
    var a chan int           //声明通道
    if a == nil {
        fmt.Println("channel a is nil, going to define it")
        a = make(chan int)         //创建通道
        fmt.Printf("Type of a is %T", a)
    }
}
```

也可以简短的声明：

```
a := make(chan int)
```

9.3 发送和接收

发送和接收的语法：

```
data := <- a // 从通道a里读数据到data
a <- data // 将data写到通道a里
```

在通道上箭头的方向指定数据是发送还是接收。

一个通道发送和接收数据，默认是`阻塞`的。当一个数据被发送到通道时，在发送语句中被阻塞，直到另一个Goroutine从该通道读取数据。类似地，当从通道读取数据时，读取被阻塞，直到一个Goroutine将数据写入该通道。

这些通道的特性是帮助Goroutines有效地进行通信。

示例代码：

```
package main

import (
    "fmt"
)

func hello(done chan bool) {
    fmt.Println("Hello world goroutine")
    done <- true
}
```

```
func main() {
    done := make(chan bool)
    go hello(done)
    <-done               // 接收数据，阻塞式
    fmt.Println("main function")
}
```

9.4 定向通道

之前我们使用的通道都是`双向通道`，我们可以通过这些通道读出或者写入数据。我们也可以创建`单向通道`，这些通道只能读出或者写入数据。

创建仅能接收数据的通道，示例代码：

```
package main

import "fmt"

func sendData(sendch chan<- int) {      //sendch为只能写入整型数据的通道
    sendch <- 10
}

func main() {
    sendch := make(chan<- int)
    go sendData(sendch)
    fmt.Println(<-sendch)
}
```

9.5 关闭通道和通道上的范围循环

发送者可以通过关闭通道，来通知接收方不会有更多的数据被发送到通道上。

接收者可以在接收来自通道的数据时使用额外的变量来检查通道是否已经关闭。

语法结构：

```
v, ok := <- ch
```

> 在上面的语句中，如果ok的值是true，表示成功的将value值发送到一个通道。如果ok是false，这意味着我们正在从一个封闭的通道读取数据。从闭通道读取的值将是通道类型的零值。例如，如果通道是一个int通道，那么从封闭通道接收的值将为0。

示例代码：

```
package main

import (
    "fmt"
)
```

```
func producer(chnl chan int) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    close(chnl)             //关闭通道
}
func main() {
    ch := make(chan int)
    go producer(ch)
    for {
        v, ok := <-ch
        if ok == false {
            break
        }
        fmt.Println("Received ", v, ok)
    }
}
```

运行结果

```
Received  0 true
Received  1 true
Received  2 true
Received  3 true
Received  4 true
Received  5 true
Received  6 true
Received  7 true
Received  8 true
Received  9 true
```

在上面的程序中，producer Goroutine将0到9写入chnl通道，然后关闭通道。主函数里有一个无限循环。它检查通道是否在行号中使用变量ok关闭。如果ok是假的，则意味着通道关闭，因此循环结束。还可以打印接收到的值和ok的值。for循环的for range形式可用于从通道接收值，直到它关闭为止。

使用range循环，示例代码：

```
package main

import (
    "fmt"
)

func producer(chnl chan int) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    close(chnl)
```

```
}
func main() {
    ch := make(chan int)
    go producer(ch)
    for v := range ch {          //range获取通道的值
        fmt.Println("Received ",v)
    }
}
```

9.6 缓冲通道

我们可以用缓冲区创建一个通道。发送到一个缓冲通道只有在缓冲区满时才被阻塞。类似地，从缓冲通道接收的信息只有在缓冲区为空时才会被阻塞。

可以通过将额外的容量参数传递给make函数来创建缓冲通道，该函数指定缓冲区的大小。

基本语法：

```
ch := make(chan type, capacity)
```

上述语法的容量应该大于0，以便通道具有缓冲区。默认情况下，无缓冲通道的容量为0，因此在之前创建通道时省略了容量参数。

示例代码：

```
package main

import (
    "fmt"
)

func main() {
    ch := make(chan string, 2)      //设置通道容量为2
    ch <- "naveen"
    ch <- "paul"
    fmt.Println(<- ch)
    fmt.Println(<- ch)
}
```

> 通道设置容量后，写入数据和读取数据具有`先进先出`的性质。

第十章 错误处理

10.1 什么是错误

错误是什么?

错误指出程序中的异常情况。假设我们正在尝试打开一个文件，文件系统中不存在这个文件。这是一个异常情况，它表示为一个错误。

Go中的错误也是一种类型。错误用内置的`error`类型表示。就像其他类型的，如int，float64，。错误值可以存储在变量中，从函数中返回，等等。

10.2 演示错误

让我们从一个示例程序开始，这个程序尝试打开一个不存在的文件。

示例代码：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("/test.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
  //根据f进行文件的读或写
    fmt.Println(f.Name(), "opened successfully")
}
```

> 在os包中有打开文件的功能函数：
>
> func Open(name string) (file *File, err error)
>
> 如果文件已经成功打开，那么Open函数将返回文件处理。如果在打开文件时出现错误，将返回一个非nil错误。

如果一个函数或方法返回一个错误，那么按照惯例，它必须是函数返回的最后一个值。因此，`Open`函数返回的值是最后一个值。

处理错误的惯用方法是将返回的错误与nil进行比较。nil值表示没有发生错误，而非nil值表示出现错误。在我们的例子中，我们检查错误是否为nil。如果它不是nil，我们只需打印错误并从主函数返回。

运行结果：

```
open /test.txt: No such file or directory
```

我们得到一个错误，说明该文件不存在。

补充

init()

在Go语言中，若定义init函数，那它运行在main函数之前，一般作为程序执行初始化。

```go
func init() {
    fmt.Println("init")
    fmt.Println("CPU核心数: ", runtime.NumCPU())

    runtime.GOMAXPROCS(runtime.NumCPU())    //设置程序为CPU核心数
}
```