

# golang

开发工具 VScode

SDK下载安装 <https://go.dev/dl/>

Golang中文网在线标准库文档: <https://studygolang.com/pkgdoc>

## 变量与数据类型

### Golang数据类型

基本数据类型

数值型

整数类型

int, int8, int32, int64, unit, uint8, uint16, uint32, uint64, byte

浮点类型

float32, float64

字符型 (没有单独的字符型, 使用byte来保存单个字母字符)

布尔型 (bool)

字符串 (string)

派生数据类型/复杂数据类型

指针

数组

结构体

管道

函数

切片

接口

map

变量的数据类型:



## 演示代码

```
package main
// import "fmt"
// import "unsafe"
import (
    "fmt"
    "unsafe"
)
// 全局变量: 定义在函数外的变量
var n7 = 100
var n8 = 9.9
// 设计者认为上面的全局变量的写法太麻烦了, 可以一次声明:
var (
    n9 = 500
```

```

n10 = "GO"
}

func main() {
    //定义在{}中的变量叫 局部变量
    //1. 变量声明
    var age int
    //2. 变量的赋值
    age = 18
    //3. 变量的使用
    fmt.Println("age = ", age)
    //变量的四种使用方式
    //第一种： 变量的使用方式： 指定变量的类型，并且赋值
    var num int = 18
    fmt.Println(num)
    //第二种： 指定变量的类型，但是不赋值，使用默认值
    var num2 int
    fmt.Println(num2)
    //第三种： 如果没有写变量的类型，那么根据=后面的值进行判定变量的类型（自动类型推断）
    var num3 = "Tom"
    fmt.Println(num3)
    //第四种： 省略var，注意 := 不能写为 =
    num4 := "男"
    fmt.Println(num4)
    //声明多个变量
    var n1, n2, n3 int
    fmt.Println(n1)
    fmt.Println(n2)
    fmt.Println(n3)
    var n4, name, n5 = 10, "Jack", 7.8
    fmt.Println(n4)
    fmt.Println(name)
    fmt.Println(n5)
    n6, height := 6.9, 100.6
    fmt.Println(n6)
    fmt.Println(height)
    //输出全局变量
    fmt.Println(n7)
    fmt.Println(n8)
    fmt.Println(n9)
    fmt.Println(n10)
    //Printf函数的作用就是： 格式化的，把n10的类型填充到%T的位置上
    fmt.Printf("n10的类型是: %T\n", n10)
    fmt.Println(unsafe.Sizeof(n10))
    //定义浮点类型的数据
    var fnum float32 = 3.14
    fmt.Println(fnum)
    //可以表示正浮点数，也可以表示负的浮点数
    var fnum2 float32 = -3.14
    fmt.Println(fnum2)
    //浮点数可以用于十进制表示形式，也可以用科学计数法表示形式 E 大小写都可以
    var fnum3 float32 = 314e-2
    fmt.Println(fnum3)
    var fnum4 float32 = 314e+2
    fmt.Println(fnum4)
    var fnum5 float32 = 314e-2
    fmt.Println(fnum5)
    var fnum6 float64 = 314e+2
    fmt.Println(fnum6)
    //浮点数可能会有精度的损失，所以通常情况下，建议使用： float64
    var fnum7 float32 = 256.000000916
    fmt.Println(fnum7)
    var fnum8 float64 = 256.000000916
    fmt.Println(fnum8)
    //golang中默认的浮点类型为： float64
    var fnum9 = 3.17
    fmt.Printf("fnum9对应的默认的类型为: %T", fnum9)
}

```

## 运算符

1. 算数运算符： +, -, \*, /, %, ++, --
2. 赋值运算符： =, +=, -=, \*=, /=, %=
3. 关系运算符： ==, !=, >, <, >=, <=

4. 逻辑运算符: `&&`, `||`, `!`
5. 位运算符: `&`, `|`, `^`
6. 其他运算符: `&`, `*` 指针操作用

7.

**[2]** 指针变量接收的一定是地址值  

```
func main() {
    var num int = 10
    fmt.Println(num)

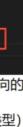
    var ptr *int = num
    *ptr = 20
    fmt.Println(num)
}
```



**[3]** 指针变量的地址不可以不匹配  

```
func main() {
    var num int = 10
    fmt.Println(num)

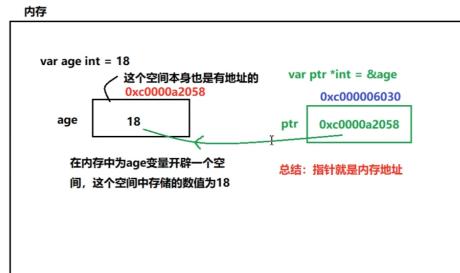
    var ptr *float32 = &num
}
```



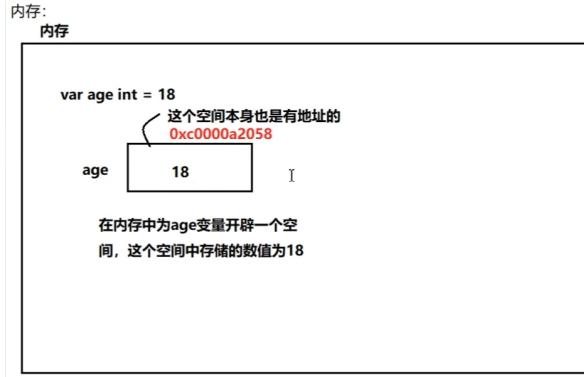
PS: \*float32意味着这个指针指向的是float32类型的数据，但是&num对应的是int类型的不可以。

**[4]** 基本数据类型（又叫值类型），都有对应的指针类型，形式为“数据类型，比如int对应的指针就是\*int，float32对应的指针类型就是\*float32。依次类推。

内存:



总结: 最重要的就是两个符号:  
 1.& 取内存地址  
 2.\* 根据地址取值



## 演示代码

```
package main
import "fmt"
func main() {
    //+号
    //1.正数 2.相加操作 3.字符串拼接
    var n1 int = +10
    fmt.Println(n1)
    var n2 int = 4 + 7
    fmt.Println(n2)
    var s1 string = "abc" + "def"
    fmt.Println(s1)
    // / 除号
    fmt.Println(10 / 3)
    fmt.Println(10.0 / 3)
    // % 取模 等价公式: a%b=a-a/b*b
    fmt.Println(10 % 3)
    fmt.Println(-10 % 3)
    fmt.Println(10 % -3)
    fmt.Println(-10 % -3)
    //++自增操作
    var a int = 10
    a++
    fmt.Println(a)
    a--
    fmt.Println(a)
    //++ 自增 加1操作, --自减, 减1操作
    //go语言里, ++, --操作非常简单, 只能单独使用, 不能参与到运算中去
    //go语言里, ++, --只能在变量的后面, 不能写在变量的前面, --a ++a 错误写法
    //赋值运算符
    var n3 int = 10
    fmt.Println(n3)
    var n4 int = (10+20)%3 + 3 - 7 //右侧的值运算清楚后, 再赋值给=的左侧
    fmt.Println(n4)
    var n5 int = 10
    n5 += 20
    fmt.Println(n5)
    //关系运算符
    fmt.Println(5 == 9) //判断左右两侧的值是否相等, 相等返回true, 不相等返回的是false, ==不是=
    fmt.Println(5 != 9) //判断不等于
    fmt.Println(5 > 9)
    fmt.Println(5 < 9)
    fmt.Println(5 >= 9)
    fmt.Println(5 <= 9)
    //逻辑运算符
    //与逻辑: &&: 两个数值/表达式只要有一侧是false, 结果一定为false
    //也叫短路与, 只要第一个数值/表达式的结果是false, 那么后面的表达式等就不用运算了, 直接结果就是false
    fmt.Println(true && true)
    fmt.Println(true && false)
```

```

fmt.Println(false && true)
fmt.Println(false && false)
//或逻辑: ||: 两个数值/表达式只要有一侧是true, 结果一定为true
//也叫短路或, 只要第一个数值/表达式的结果是true, 那么后面的表达式等就不用运算了, 直接结果就是true
fmt.Println(true || true)
fmt.Println(true || false)
fmt.Println(false || true)
fmt.Println(false || false)
//非逻辑: 取相反的结果
fmt.Println(!true)
fmt.Println(!false)
//其他运算符: &, *
var age int = 18
fmt.Println("age对应的存储空间的地址为: ", &age) //age对应的存储空间的地址为: 0xc000016220
var ptr *int = &age
fmt.Println(ptr)
fmt.Println("ptr这个指针指向的具体数值为: ", *ptr)
}

```

## 流程控制

流程控制的作用：

流程控制语句是用来控制程序中个语句执行顺序的语句，可以把语句组合成能完成一定功能的小逻辑模块。

控制语句的分类：

控制语句分为三类：顺序，选择和循环。

“顺序结构”代表“先执行a，在执行b”的逻辑

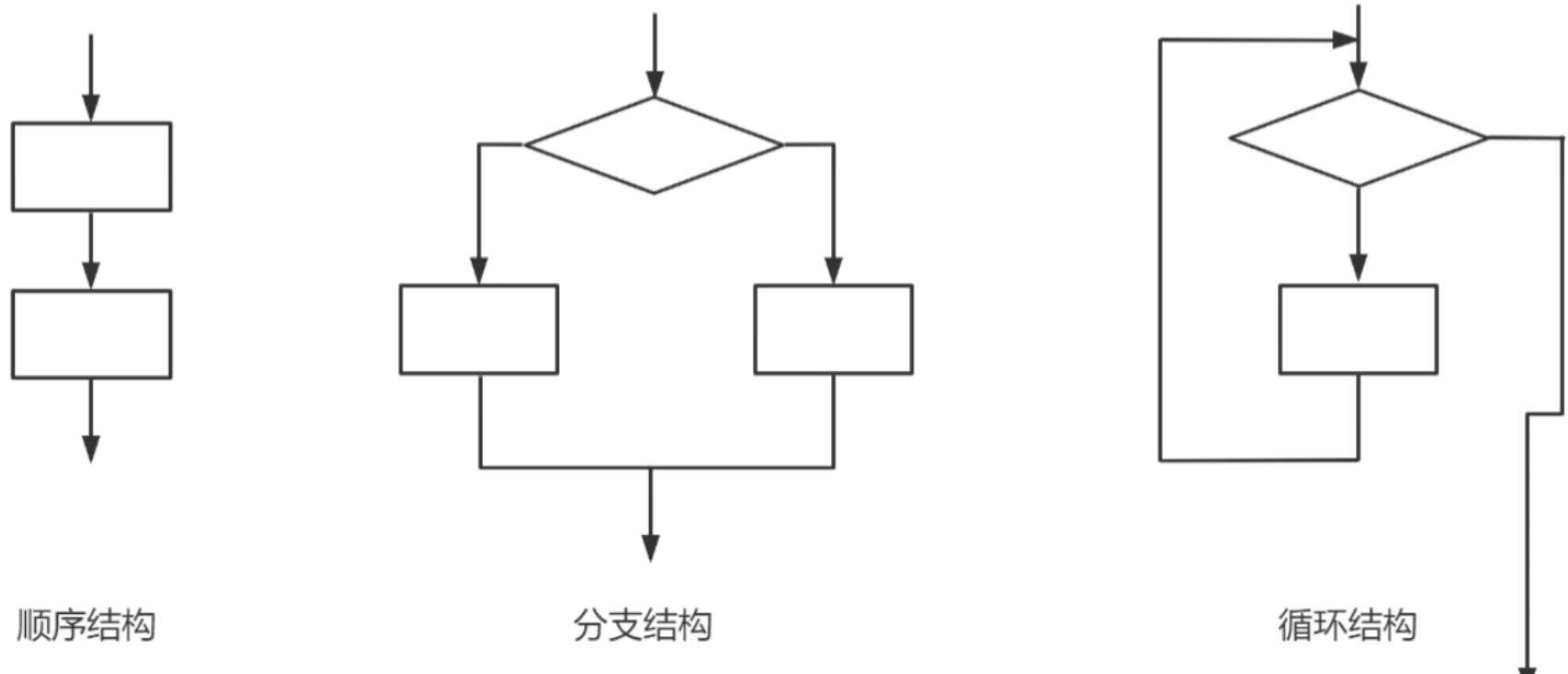
“条件判断结构”代表“如果。。。, 则。。。。”的逻辑

“循环结构”代表“如果。。。, 则继续。。。。”的逻辑

三种流程控制语句就能表示所有的事情。

这三种基本逻辑结构是相互支撑的，它们共同构成了算法的基本结构，无论怎么复杂的逻辑结构，都可以通过它们来表达。所以任何一种高级语言都具备上述两种结构。

### 【3】流程控制的流程：



## 选择控制语句基本语法

单分支：

```
if 条件表达式 {
    逻辑代码
}
```

但条件表达为true时，就会执行代码。

PS：条件表达式左右的()可以不写，也建议不写

PS：if和表达式中间，一定要有空格

PS：在Golang中，{}是必须有的，就算只写一行代码。

双分支：

```
if 条件表达式 {
```

```
逻辑代码1
}else{
    逻辑代码2
}
当条件表达式成立，即执行逻辑代码1，否则执行逻辑代码2。{}也是必须有的。
```

PS：下面的格式是错误的：

```
if 条件表达式 {
    逻辑代码1
}
else{
    逻辑代码2
}
```

多分支：

```
if 条件表达式 {
    逻辑代码1
}else if{
    逻辑代码2
}
...
else{
    逻辑代码n
}
```

switch分支

```
switch 表达式 {
    case 值1,值2...:
        语句块1
    case 值3,值4...:
        语句块2
    ...
    default:
        语句块
}
```

注意事项：

switch后是一个表达式（即：常量值，变量，一个有返回值的函数等都可以）  
case后面的表达式如果是常量值（字面量），则要求不能重复  
case后的各个值的数据类型，必须和switch的表达式数据类型一致  
case后面可以带多个值，使用逗号间隔。比如case 值1,值2...  
case后面不需要带break  
default语句不是必须的，位置也是随意的。  
switch后也可以不带表达式，当做if分支来使用 不推荐

```
switch {
    case a == 1:
        fmt.Println("aaa")
    case a == 2:
        fmt.Println("bbb")
}
```

switch后也可以直接声明/定义一个变量，分号结束，不推荐

```
switch var b int = 10; {
    case b > 9:
        fmt.Println("aaa")
    case b < 9:
        fmt.Println("bbb")
}
```

switch穿透，利用fallthrough关键字，如果在case语句块后增加fallthrough，则会继续执行下一个case，也叫做switch穿透。

```

var score int = 81
if score >= 90 {
    fmt.Println("A")
} else if score >= 80 {
    fmt.Println("B")
    fallthrough
} else if score >= 70 {
    fmt.Println("C")
} else if score >= 60 {
    fmt.Println("D")
} else {
    fmt.Println("E")
}

```

输出: 80

70

## 演示代码

```

package main
import "fmt"
func main() {
    //实现功能: 如果口罩的库存小于30个, 提示: 库存不足
    var count int = 100
    //单分支:
    if count < 30 {
        fmt.Println("存量不足")
    }
    //if后面表达式, 返回结果一定是true或者false
    //如果返回结果为true的话, 那么{}中的代码就会执行
    //如果返回结果为false的话, 那么{}中的代码就不会执行
    //if后面一定要有空格, 和条件表达式分隔开来
    //{}一定不能省略
    //条件表达式左右的()是建议省略的
    //在golang里, if后面可以并列的加入变量的定义
    if count2 := 20; count2 < 30 {
        fmt.Println("存量不足")
    }
    //如果口罩的库存小于30个, 提示: 库存不足, 否则: 库存充足
    var count2 int = 70
    if count2 < 30 {
        fmt.Println("库存不足")
    } else {
        fmt.Println("库存充足")
    }
    //多分支, 如果已经走了一个分支了, 那么下面的分支就不会再去判断执行了
    var score int = 81
    if score >= 90 {
        fmt.Println("A")
    } else if score >= 80 {
        fmt.Println("B")
    } else if score >= 70 {
        fmt.Println("C")
    } else if score >= 60 {
        fmt.Println("D")
    } else {
        fmt.Println("E")
    }
    //switch分支
    //switch后面是一个表达式, 这个表达式的结果依次跟case进行比较, 满足结果的话就执行冒号后面的代码。
    //default是用来“兜底”的一个分支, 其他case分支都不走的情况下就会走default分支
    //default分支可以放在任意位置上, 不一定非要放在最后
    var score2 int = 81
    switch score2 / 10 {
    case 10:
        fmt.Println("A")
    case 9:
        fmt.Println("B")
    case 8:
        fmt.Println("C")
    case 7:
        fmt.Println("D")
    case 6:
        fmt.Println("E")
    case 5:
    }
}

```

```

fmt.Println("F")
case 4:
    fmt.Println("G")
case 3:
    fmt.Println("H")
case 2:
    fmt.Println("I")
case 1:
    fmt.Println("J")
case 0:
    fmt.Println("K")
default:
    fmt.Println("ERROR!")
}
}

```

## 循环控制语句基本语法

基本语法

goLang不再使用while相关的循环控制语句

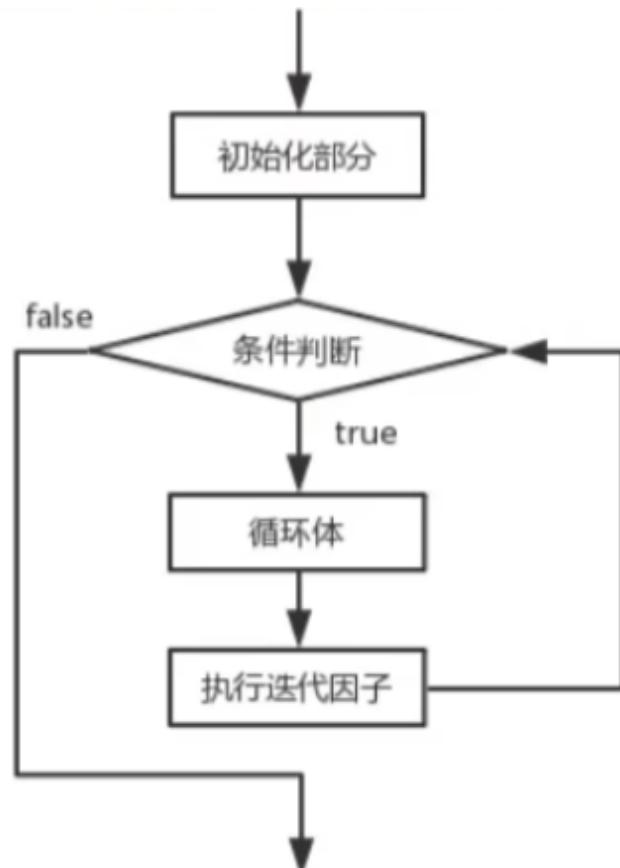
```
for 初始表达式; 布尔表达式; 迭代因子 {
    循环体;
}
```

for循环语句是支持迭代的一种通用结构，是最有效，最灵活的循环结构。for循环在第一次反复之前要进行初始化，即执行初始表达式；随后，对布尔表达式进行判定，若判定为true，则执行循环体，否则，终止循环体；最后在每一次反复的时候，进行某种形式的“步进”，即执行迭代因子。

1. 初始化部分设置循环变量的初值
2. 条件判断部分为任意布尔表达式
3. 迭代因子控制循环变量的增减

for循环在执行条件判定后，先执行的循环体部分，再执行步进。

for循环结构的流程图如图所示：



## for range

for range 结构是Go语言特有的一种迭代结构，在许多情况下都非常有用，for range可以遍历数组，切片，字符串，map及通道，for range语法上类似于其他语言中的foreach语句，一般形式为

```
for key, value := range collection {
    ...
}
```

```
main.go  X
src > gocode > testproject01 > unit4 > demo08 > main.go
1 package main
2 import "fmt"
3 func main() {
4     // 定义一个字符串:
5     var str string = "hello golang你好"
6     // 方式1: 普通for循环: 按照字节进行遍历输出的 (暂时先不使用中文)
7     // for i := 0; i < len(str); i++ { // i: 理解为字符串的下标
8         fmt.Printf("%c \n", str[i])
9     }
10
11    // 方式2: for range
12    for i, value := range str {
13        fmt.Printf("索引为: %d, 具体的值为: %c \n", i, value)
14    }
15
16    // 对 str 进行遍历, 遍历的每个结果的索引值被 i 接收, 每个结果的具体数值被 value 接收
17    // 遍历对字符进行遍历的
18 }
```

```
选择C:\Windows\System32\cmd.exe
D:\goproject\src\gocode\testproject01\unit4\demo08>go run main.go
索引为: 0, 具体的值为: h
索引为: 1, 具体的值为: e
索引为: 2, 具体的值为: l
索引为: 3, 具体的值为: l
索引为: 4, 具体的值为: o
索引为: 5, 具体的值为:
索引为: 6, 具体的值为: g
索引为: 7, 具体的值为: o
索引为: 8, 具体的值为: l
索引为: 9, 具体的值为: a
索引为: 10, 具体的值为: n
索引为: 11, 具体的值为: g
索引为: 12, 具体的值为: 你
索引为: 13, 具体的值为: 好
索引为: 14, 具体的值为:
索引为: 15, 具体的值为: 好
D:\goproject\src\gocode\testproject01\unit4\demo08>
```

## break

1.switch分支中，每个case分支后都用break结束当前分支，但是在go语言中break可以省略不写。

2.break可以结束正在执行的循环

3.break的作用结束离它最近的循环

## label标签使用

```
1. package main
2. import "fmt"
3. func main() {
4.     // 双重循环:
5.     label2:
6.     for i := 1; i <= 5; i++ {
7.         for j := 2; j <= 4; j++ {
8.             fmt.Printf("i: %v, j: %v \n", i, j)
9.             if i == 2 && j == 2 {
10.                 break label2 // 结束指定标签对应的循环
11.             }
12.         }
13.     }
14.     fmt.Println("-----ok")
15. }
```

## continue的作用

```
1. package main
2. import "fmt"
3. func main() {
4.     // 功能: 输出1-100中被6整除的数:
5.     // 方式1:
6.     // for i := 1; i <= 100; i++ {
7.     //     if i % 6 == 0 {
8.     //         fmt.Println(i)
9.     //     }
10.    // }
11.    // 方式2:
12.    for i := 1; i <= 100; i++ {
13.        if i % 6 != 0 {
14.            continue // 结束本次循环, 继续下一次循环
15.        }
16.        fmt.Println(i)
17.    }
18. }
```

continue的作用是结束离它近的那个循环，继续离它近的那个循环

## goto

1.Golang的goto语句可以无条件地转移到程序中指定的行

2.goto语句通常与条件语句配合使用。可以用来实现条件转移

3.在Go程序设计中一般不建议使用goto语句，以免造成程序流程的混乱。

## return

结束当前函数

## 函数

基本语法

```
func 函数名 (形参列表) (返回值类型列表) {  
    执行语句..  
    return + 返回值列表  
}
```

(1) 函数名：

遵循标识符命名规范：见名知义，驼峰命名

首字母不能是数字

首字母大写该函数可以被本包文件和其他包文件使用（类似public）

首字母小写只能被本包文件使用，其他文件不能使用（类似private）

(2) 形参列表

可以是一个参数，也可以是n个参数，可以是0个参数

形式参数列表：

作用：接收外来的数据

实际参数：

实际传入的数据

```
8 //自定义函数：功能：两个数相加  
9 func cal (num1 int,num2 int) (int) { //如果返回值类型就一个的话，那么()是可以省略不写的  
10    var sum int = 0  
11    sum += num1  
12    sum += num2  
13    return sum  
14 }  
15  
16  
17 18 func main(){  
19     //功能：10 + 20  
20     //调用函数：  
21     sum := cal(10,20)  
22     fmt.Println(sum)
```

形参

实参

(3) 返回值列表

返回0个：

```
4 //自定义函数：功能：两个数相加  
5 func cal (num1 int,num2 int) { //如果返回值类型就一个的话，那么()是可以省略不写的  
6     var sum int = 0  
7     sum += num1  
8     sum += num2  
9     //return sum  
10    fmt.Println(sum)  
11 }  
12  
13  
14 15 func main(){  
16     //功能：10 + 20  
17     //调用函数：  
18     cal(10,20)  
19     //fmt.Println(sum)  
20 }
```

如果没有返回值，那么返回值类型什么都不写就可以了

返回1个：

```
4 //自定义函数：功能：两个数相加  
5 func cal (num1 int,num2 int) int { //如果返回值类型就一个的话，那么()是可以省略不写的  
6     var sum int = 0  
7     sum += num1  
8     sum += num2  
9     return sum  
10    //fmt.Println(sum)  
11 }  
12  
13  
14 15 func main(){  
16     //功能：10 + 20  
17     //调用函数：  
18     sum := cal(10,20)  
19     fmt.Println(sum)  
20 }
```

如果返回值只有一个，那么这个列表中类型左右的()可以省略不写

返回多个：

```

    }
    //计算两个数的和，两个数的差
    func cal2 (num1 int,num2 int) (int,int) {
        var sum int = 0
        sum += num1
        sum += num2

        var result int = num1 - num2
        return sum,result
    }

    func main(){
        //功能: 10 + 20
        //调用函数:
        // sum := cal(10,20)
        // fmt.Println(sum)

        sum1,result1 := cal2(10,20)
        fmt.Println(sum1)
        fmt.Println(result1)
    }

```

多个返回值，忽略不想接收的返回值：

```

    //计算两个数的和，两个数的差
    func cal2 (num1 int,num2 int) (int,int) {
        var sum int = 0
        sum += num1
        sum += num2

        var result int = num1 - num2
        return sum,result
    }

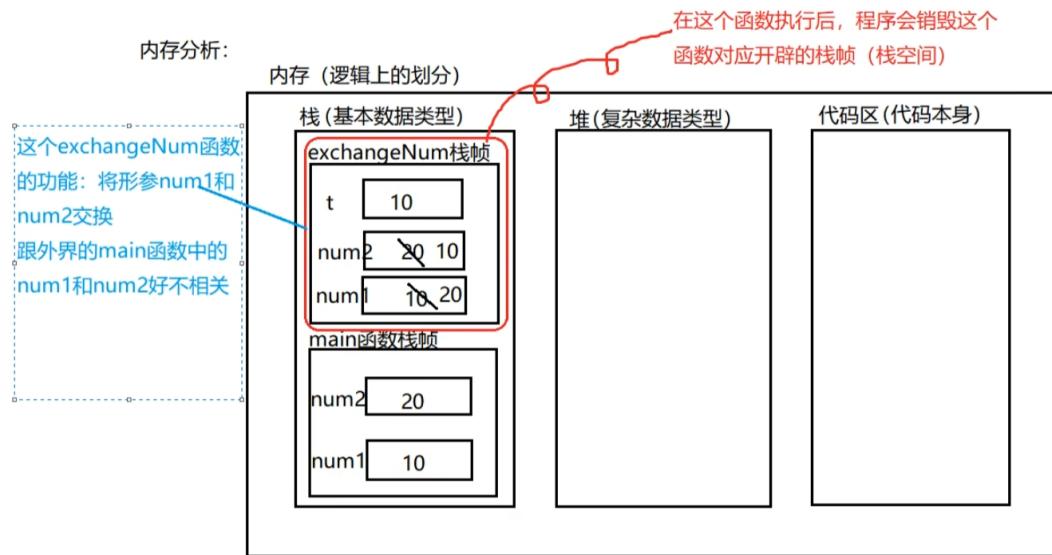
    func main(){
        //功能: 10 + 20
        //调用函数:
        // sum := cal(10,20)
        // fmt.Println(sum)

        sum1,_ := cal2(10,20)
        fmt.Println(sum1)
    }

```

如果有返回值不想接收，那么可以利用\_进行忽略

## 内存分析



## Golang中函数不支持重载

```

//自定义函数: 功能: 交换两个数
func exchangeNum (num1 int,num2 int){
    var t int
    t = num1
    num1 = num2
    num2 = t
}

func exchangeNum (num1 int){
    var t int
    t = num1
    t = num1
}

```

命令行输出结果：

```

C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.18363.1316]
(c) 2019 Microsoft Corporation. 保留所有权利。
D:\goproject\src\gocode\testproject01\unit5\demo03>cd ..
D:\goproject\src\gocode\testproject01\unit5>cd demo04
D:\goproject\src\gocode\testproject01\unit5\demo04>go run main.go
# command-line arguments
.\main.go:13:6: exchangeNum redeclared in this block
previous declaration at .\main.go:6:33
D:\goproject\src\gocode\testproject01\unit5\demo04>

```

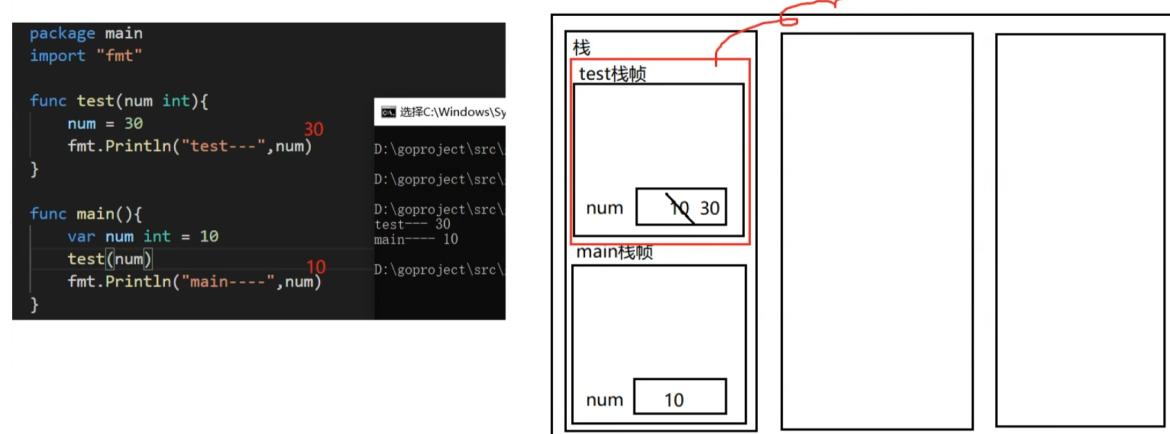
## Golang中支持可变参数（如果希望函数带有可变数量的参数）

```

1 package main
2 import "fmt"
3
4 // 定义一个函数，函数的参数为：可变参数……参数的数量可变
5 // args...int 可以传入任意多个数量的int类型的数据，传入0个，1个，……n个
6 func test(args...int) {
7     // 函数内部处理可变参数的时候，将可变参数当做切片来处理
8     // 遍历可变参数：
9     for i := 0; i < len(args); i++ {
10         fmt.Println(args[i])
11     }
12 }
13
14 func main() {
15     test()
16     fmt.Println("-----")
17     test(3)
18     fmt.Println("-----")
19     test(37, 58, 39, 59, 47)
20 }

```

基本数据类型和数组默认都是值传递的，即进行值拷贝。在函数内修改，不会影响到原来的值。

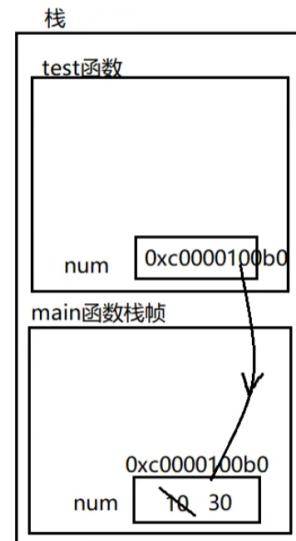


以值传递方式的数据类型，如果希望在函数内的变量能修改函数外的变量，可以传入变量的地址&。函数内以指针的方式操作变量。从效果来看类似引用传递。

```

3 // 参数的类型为指针
4 func test(num *int) {
5     // 对地址对应的变量进行改变数值：
6     *num = 30
7 }
8
9 func main() {
10    var num int = 10
11    fmt.Println(&num) 0xc0000100b0
12    test(&num) // 调用test函数的时候，传入的是num的地址
13    fmt.Println(num)
14 }
15

```



在Go中，函数也是一种数据类型，可以赋值给一个变量，则该变量就是一个函数类型的变量了。通过该变量可以对函数调用。

函数既然是种数据类型，因此在go中，函数可以作为形参，并且调用

为了简化数据类型定义，Go支持自定义数据类型

基本语法：type自定义数据类型名，数据类型

可以理解为：相当于起了一个别名

例如：type myInt int -----> 这时myInt就等价于int来使用了

```
//自定义数据类型：(相当于起别名)：给int类型起了别名叫myInt类型
type myInt int

var num1 myInt = 30
fmt.Println("num1", num1)

var num2 int = 30
num2 = int(num1) //虽然是别名，但是在go中编译识别的时候还是认为myInt和int不是一种数据类型
fmt.Println("num2", num2)
```

例如：type mySum func (int, int) int -----> 这时mySum就等价一个函数类型func(int, int) int

```
type myFunc func(int)
func test03 (num1 int, num2 float32, testFunc myFunc){
    fmt.Println("-----test02")
}
```

## 支持对函数返回值命名

传统写法要求：返回值和返回值的类型对应，顺序不能差

但go可以对函数返回值命名，返回的值可以无视顺序

```
// 自定义函数：多个返回值
func cal2(num1 int, num2 int) (int, int) {
    var sum int = 0
    sum = num1 + num2

    sub := num1 - num2
    return sum, sub
}

//升级写法
func cal3(num1 int, num2 int) (sum int, sub int) {
    sum = num1 + num2
    sub = num1 - num2
    return
}
```

## 代码演示

```
package main
import (
    "fmt"
)

// 自定义函数：功能：两个数相加
func cal(num1 int, num2 int) int {
    var sum int = 0
    sum = num1 + num2
    return sum
}

// 自定义函数：多个返回值
func cal2(num1 int, num2 int) (int, int) {
    var sum int = 0
    sum = num1 + num2
    sub := num1 - num2
    return sum, sub
}

// 升级写法
func cal3(num1 int, num2 int) (sum int, sub int) {
    sum = num1 + num2
    sub = num1 - num2
    return
}

// 定义一个函数，函数的参数为：可变参数... 参数的数量可变
// args...int 可以传入任意多个数量的int类型的数据 传入0个, 1个, , n个
func test(args ...int) {
    //函数内部处理可变参数的时候，将可变参数当作切片来处理
    //遍历可变参数
    for i := 0; i < len(args); i++ {
        fmt.Println(args[i])
    }
}

func test2(num int) {
```

```

num = 10
fmt.Println("test2---", num)
}
func test3(num *int) {
    *num = 30
}
func test4(num1 int, num2 float32, testFunc func(int)) {
    testFunc(10)
    fmt.Println("---test4", num1, num2)
}
func main() {
    //调用函数
    sum := cal(10, 20)
    fmt.Println(sum)
    //多个返回值
    sum1, sub := cal2(10, 20)
    fmt.Println(sum1, sub)
    sum2, _ := cal2(10, 20)
    fmt.Println(sum2)
    //可变参数
    test()
    fmt.Println("-----")
    test(2)
    fmt.Println("-----")
    test(33, 44, 55, 66, 77)
    //基本数据类型和数组默认都是值传递的，即进行值拷贝。在函数内修改，不会影响到原来的值。
    var num int = 30
    test2(num)
    fmt.Println("main---", num)
    //以值传递方式的数据类型，如果希望在函数内的变量能修改函数外的变量，可以传入变量的地址&。
    //函数内以指针的方式操作变量。从效果来看类似引用传递。
    var num2 int = 10
    fmt.Println(&num2)
    test3(&num2)
    fmt.Println(num2)
    //函数也是一种数据类型，可以赋值给一个变量，则该变量就是一个函数类型的变量了
    function := test2
    fmt.Printf("function的类型是: %T、test2函数的类型是: %T\n", function, test2)
    function(10)
    //定义一个函数，把另一个函数作为形参
    test4(10, 9.8, test2)
    test4(11, 10.9, function)
    //自定义数据类型：（相当于起别名）：给int类型起了别名叫myInt类型
    type myInt int
    var num3 myInt = 30
    fmt.Println("num1", num3)
    var num4 int = 30
    num4 = int(num3) //虽然是别名，但是在go中编译识别的时候还是认为myInt和int不是一种数据类型
    fmt.Println("num4", num4)
}

```

## 包的引入

使用包的原因：

(1) 我们不可能把所有的函数都放在同一个源文件中，可以分门别类的把函数放在不同的源文件中



(2) 解决同名问题：两个人都想定义一个同名的函数，在同一文件中是不可以定义相同名字的函数的。此时可以用包来区分



1.package进行包的声明, 建议: 包的声明这个包和所在的文件夹同名

2.main包是程序的入口包, 一般main函数会放在这个包下

main函数一定要放在main包下, 否则不能编译执行

```
D:\goproject\src\gocode\testproject01\unit5\demo09\crm>go run main.go
go run: cannot run non-main package
```

3.打包语法:

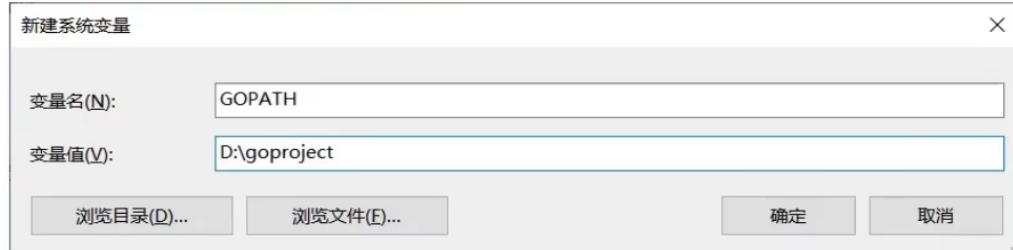
```
package 包名
```

4.引入包的语法: import "包的路径"

包名是从 \$GOPATH/src/后开始计算的, 使用/进行路径分隔

```
//4.如果有多个包,建议一次性导入:
import(
    "fmt"
    "gocode/testproject01/unit5/demo09/crm/dbutils"
)
```

需要配置一个环境变量: GOPATH



5.如果有多个包, 建议一次性导入, 格式如下:

```
import (
    "fmt"
    "gocode/testproject01/unit5/demo09/crm/dbutils"
)
```

6.在函数调用的时候前面要定位到所在的包

```
func main(){
    fmt.Println("你好, 这是main函数的执行")
    dbutils.GetConn()//4.在函数调用的时候前面要定位到所在的包
}
```

7.函数名, 变量名首字母大写, 函数, 可以被其他包访问

8.一个目录下不能有重复的函数

```

src > gocode > testproject01 > unit5 > demo09 > crm > dbutils > dbutils.go
1 package dbutils
2
3 import "fmt"
4
5 func GetConn(){//5.首字母大写，可以被其它包访问
6     fmt.Println("执行了dbutils包下的getConn函数")
7 }
8
9 选择C:\Windows\System32\cmd.exe
D:\goproject\src\gocode\testproject01\unit5>cd demo09
D:\goproject\src\gocode\testproject01\unit5\demo09>cd crm
D:\goproject\src\gocode\testproject01\unit5\demo09\crm>cd main
D:\goproject\src\gocode\testproject01\unit5\demo09\crm\main>go run main.go
你好，这是main函数的执行
执行了dbutils包下的getConn函数
D:\goproject\src\gocode\testproject01\unit5\demo09>go run main.go
你好，这是main函数的执行
执行了dbutils包下的getConn函数
D:\goproject\src\gocode\testproject01\unit5\demo09\crm\main>go run main.go
go run: cannot run non-main package
D:\goproject\src\gocode\testproject01\unit5\demo09\crm\main>go run main.go
# gocode/testproject01/unit5/demo09/crm/dbutils
..dbutils.util.go:5:6: GetConn redeclared in this block
previous declaration at ..\dbutils\dbutils.go:5:6
D:\goproject\src\gocode\testproject01\unit5\demo09\crm\main>

```

## 9.包名和文件夹的名字，可以不一样

```

src > gocode > testproject01 > unit5 > demo09 > crm > calutils > calutils.go
1 package aaa
2 import "fmt"
3 func Add(){}
4     fmt.Println("add函数被执行了")
5

```

```

src > gocode > testproject01 > unit5 > demo09 > crm > main > main.go
1 package main //1.package进行包的声明，建议：包的声明这个包和所在的文件夹同名
2 //2.main包是程序的入口包，一般main函数会放在这个包下
3
4
5 // import "fmt"
6 //3.包名是从$GOPATH/src/后开始计算的，使用/进行路径分隔。
7 // import "gocode/testproject01/unit5/demo09/crm/dbutils"
8
9 //4.如果有多个包，建议一次性导入：
10 import(
11     "fmt"
12     "gocode/testproject01/unit5/demo09/crm/dbutils"
13     "gocode/testproject01/unit5/demo09/crm/calutils"
14 )
15
16
17
18 func main(){
19     fmt.Println("你好，这是main函数的执行")
20     dbutils.GetConn()//4.在函数调用的时候前面要定位到所在的包
21     aaa.Add()
22 }

```

导入的是包所在的文件夹的路径  
函数前的定位用的是包名

## 10.一个目录下的同级文件归属一个包 同级别的源文件的包的声明必须一致

```

src > gocode > testproject01 > unit5 > demo09 > crm > dbutils > dbutils.go
1 package bbb
2
3 import "fmt"
4
5 func GetConn2(){//5.首字母大写, 可以被其它包访问
6     fmt.Println("执行了dbutils包下的getConn函数")
7 }
8
9

src > gocode > testproject01 > unit5 > demo09 > crm > dbutils > dbutils.go
1 package bbb
2
3 import "fmt"
4
5 func GetConn(){//5.首字母大写, 可以被其它包访问
6     fmt.Println("执行了dbutils包下的getConn函数")
7 }
8
9

```

## 11. 包到底是什么：

- (1) 在程序层面，所有使用相同package包名的源文件组成的代码模块
- (2) 在源文件层面就是一个文件夹

## init函数

- (1) init函数：初始化函数，可以用来进行一些初始化操作

每一个源文件都可以包含一个init函数，该函数会在main函数执行前，被Go运行框架调用

```

src > gocode > testproject01 > unit5 > demo10 > main.go
1 package main
2 import "fmt"
3
4 func init(){
5     fmt.Println("init函数被执行！")
6 }
7
8 func main(){
9     fmt.Println("main函数被执行！")
10}

```

选择C:\Windows\System32\cmd.exe  
Microsoft Windows [版本 10.0.18363.1316]  
(c) 2019 Microsoft Corporation。保留所有权利。  
D:\goproject\src\gocode\testproject01\unit5>cd demo10  
D:\goproject\src\gocode\testproject01\unit5\demo10>go run main.go  
init函数被执行！  
main函数被执行！

- (2) 全局变量定义，init函数，main函数的执行流程？

```

src > gocode > testproject01 > unit5 > demo10 > main.go
1 package main
2 import "fmt"
3
4 var num int = test()
5
6 func test() int{
7     fmt.Println("test函数被执行")
8     return 10
9 }
10
11 func init(){
12     fmt.Println("init函数被执行！")
13 }
14
15 func main(){
16     fmt.Println("main函数被执行！")
17 }

```

第一步：全部变量的定义  
第二步：init函数的调用  
第三步：main函数的调用

C:\Windows\System32\cmd.exe  
D:\goproject\src\gocode\testproject01\unit5\demo10>go run main.go  
test函数被执行  
init函数被执行！  
main函数被执行！

- (3) 多个源文件都有init函数的时候，如何执行

```

package main
import (
    "fmt"
    "gocode/testproject01/unit5/demo10/testutils"
)
var num int = test()
func test() int{
    fmt.Println("test函数被执行")
    return 10
}
func init(){
    fmt.Println("main.go中的init函数被执行！")
}
func main(){
    fmt.Println("main函数被执行！")
    fmt.Println("Age= ",testutils.Age,"Sex=",testutils.Sex,"Name=",testutils.Name)
}

```

```

package testutils
import "fmt"
var Age int
var Sex string
var Name string
//定义一个init函数对变量进行初始化赋值:
func init(){
    fmt.Println("testutils中的init函数被执行了")
    Age = 19
    Sex = "女"
    Name = "丽丽"
}

```

执行结果：

```

D:\goproject\src\gocode\testproje
testutils中的init函数被执行了
test函数被执行
main.go中的init函数被执行！
main函数被执行！
Age= 19 Sex= 女 Name= 丽丽

```

## 匿名函数

(1) Go支持匿名函数，如果我们某个函数只是希望使用一次，可以考虑使用匿名函数

(2) 匿名函数的使用方式：

1.在定义匿名函数时就直接调用，这种方式匿名函数只能调用一次（用的多）

```

package main
import "fmt"
func main(){
    //定义匿名函数：定义的同时调用
    result := func (num1 int,num2 int) int{
        return num1 + num2
    }(10,20)
    fmt.Println(result)
}

```

2.将匿名函数赋给一个变量（该变量就是函数变量了），再通过该变量来调用匿名函数

```

//将匿名函数赋给一个变量，这个变量实际就是函数类型的变量
//sub等价于匿名函数
sub := func (num1 int,num2 int) int{
    return num1 - num2
}

//直接调用sub就是调用这个匿名函数了
result01 := sub(30,70)
fmt.Println(result01)

```

(3) 如何让一个匿名函数，可以在整个程序中有效呢？

将匿名函数给一个全局变量就可以了

```
4 var Func01 = func (num1 int, num2 int) int{  
5     return num1 * num2  
6 }  
7
```

## 闭包

(1) 什么是闭包

闭包就是一个函数和与其他相关的引用环境组合的一个整体

(2) 案例展示

```
main.go x  
src > gocode > testproject01 > unit5 > demo12 > main.go  
1 package main  
2 import "fmt"  
3  
4 //函数功能：求和  
5 //函数的名字：getSum 参数为空  
6 //getSum函数返回值为一个函数，这个函数的参数是一个int类型的参数，返回值也是int类型  
7 func getSum() func (int) int {  
8     var sum int = 0  
9     return func (num int) int {  
10        sum = sum + num  
11        return sum  
12    }  
13 }  
14 //闭包：返回的匿名函数+匿名函数以外的变量num  
15  
16 func main(){  
17     f := getSum()  
18     fmt.Println(f(1))//1  
19     fmt.Println(f(2))//3  
20     fmt.Println(f(3))//6  
21     fmt.Println(f(4))//10  
22 }
```

匿名函数中引用的那个变量会一直保存在内存中，可以一直使用

(3) 闭包的本质

闭包本质依旧是一个匿名函数，只是这个函数引入外界的变量/参数

匿名函数 + 引用的变量/参数 = 闭包

(4) 特点

1. 返回的是一个匿名函数。但是这个匿名函数引用到函数外的变量/参数，因此这个匿名函数就和变量/参数形成一个整体，构成闭包。
2. 闭包中使用的变量/参数会一直保存在内存中，所以会一直使用 ---> 意味着闭包不可滥用（对内存消耗大）

(5) 不使用闭包可以吗？

不用闭包的时候，想保留的值，不可以反复使用

闭包应用场景：闭包可以保留上次引用的某个值，我们传入一次就可以反复使用了

## defer关键字

(1) 在函数中，程序员经常需要创建资源，为了在函数执行完毕后，及时的释放资源，Go的设计者提供defer关键字

(2) 案例展示

```

src > gocode > testproject01 > unit5 > demo13 > main.go
1 package main
2 import "fmt"
3
4
5 func main(){
6     fmt.Println(add(30,60))
7 }
8
9 func add(num1 int ,num2 int) int{
10    //在Golang中, 程序遇到defer关键字, 不会立即执行defer后的语句, 而是将defer后的语句压入一个栈中, 然后继续执行函数后面的语句
11    defer fmt.Println("num1=",num1)
12    defer fmt.Println(["num2=",num2])
13    //栈的特点是: 先进后出
14    //在函数执行完毕以后, 从栈中取出语句开始执行, 按照先进后出的规则执行语句
15    var sum int = num1 + num2 //230
16    fmt.Println("sum=",sum)
17    return sum
18 }
19

```

C:\Windows\System32\cmd.exe

```

D:\goproject\src\gocode\testproject01\unit5\demo13>go run main.go
sum= 90
num2= 60
num1= 30
90
D:\goproject\src\gocode\testproject01\unit5\demo13>

```

(3) 代码变动下, 再次看结果:

```

2 import "fmt"
3
4
5 func main(){
6     fmt.Println(add(30,60))
7 }
8
9 func add(num1 int ,num2 int) int{
10    //在Golang中, 程序遇到defer关键字, 不会立即执行defer后的语句, 而是将defer后的语句压入一个栈中, 然后继续执行函数后面的语句
11    defer fmt.Println("num1=",num1)
12    defer fmt.Println("num2=",num2)
13    num1 += 90 //num1:120
14    num2 += 50 //num2:110
15    //栈的特点是: 先进后出
16    //在函数执行完毕以后, 从栈中取出语句开始执行, 按照先进后出的规则执行语句
17    var sum int = num1 + num2 //230
18    fmt.Println("sum=",sum)
19    return sum
20 }
21

```

选择C:\Windows\System32\cmd.exe

```

230
D:\goproject\src\gocode\testproject01\unit5\demo13>go run main.go
sum= 230
num2= 60
num1= 30
230
D:\goproject\src\gocode\testproject01\unit5\demo13>

```

发现: 遇到defer关键字, 会将后面的代码语句压入栈中, 也会将相关的值同时拷贝入栈中, 不会随着函数后面的变化而变化

#### (4) defer的应用场景

想关闭某个使用的资源, 在使用的时候直接随手defer, 因为defer有延迟执行机制 (函数执行完毕再执行defer压入栈的语句), 所以用完随手写了关闭, 比较省心, 省事

## 字符串函数

(1) 统计字符串的长度, 按字节进行统计:

`len (str)`

使用内置函数不用导包, 直接用就行

(2) 字符串遍历

`r := []rune(str)`

(3) 字符串转整数

`n, err := strconv.Atoi("66")`

(4) 整数转字符串

`str = strconv.Itoa(6887)`

(5) 查找子串是否在指定的字符串中

`strings.Contains("javaandgolang", "go")`

(6) 统计一个字符串有几个指定的子串

`strings.Count("javaandgolang", "a")`

(7) 不区分大小写的字符串比较

`fmt.Println(strings.EqualFold("go", "Go"))`

(8) 返回子串在字符串第一次出现的索引值, 如果没有返回-1

`strings.Index("javaandgolang", "a")`

(9) 字符串的替换

`strings.Replace("goandjavagogo", "go", "golang", n)`

n可以指定你希望替换几个，如果n=-1表示全部替换，替换两个n就是2

(10) 按照指定的某个字符，为分割标识，将一个字符串拆分成字符串数组

```
strings.Split("go-python-java", "-")
```

(11) 将字符串的字母进行大小写的转换

```
strings.ToLower("Go") //go
```

```
strings.ToUpper("go") //Go
```

(12) 将字符串左右两边的空格去掉

```
strings.TrimSpace(" go and java ")
```

(13) 将字符串左右两边指定的字符去掉

```
strings.Trim("~golang~", "~")
```

(14) 将字符串左边指定的字符串去掉

```
strings.TrimLeft("~golang~", "~")
```

(15) 将字符串右边指定的字符串去掉

```
strings.TrimRight("~golang~", "~")
```

(16) 判断字符串是否以指定的字符串开头

```
strings.HasPrefix("http://java.sun.com/jsp/fmt", "http" )
```

(17) 判断字符串是否以指定的字符串结束

```
strings.HasSuffix("demo.png", ".png")
```

## 日期和时间函数

```
1. package main
2. import (
3.     "fmt"
4.     "time"
5. )
6.
7. func main() {
8.     //时间和日期的函数，需要到入time包，所以你获取当前时间，就要调用函数Now函数：
9.     now := time.Now()
10.    //Now()返回值是一个结构体，类型是：time.Time
11.    fmt.Printf("%v ~~~ 对应的类型为：%T\n", now, now)
12.    //2021-02-08 17:47:21.7600788 +0800 CST m=+0.005983901 ~~~ 对应的类型为：time.Time
13.
14.    //调用结构体中的方法：
15.    fmt.Printf("年: %v \n", now.Year())
16.    fmt.Printf("月: %v \n", now.Month()) //月: February
17.    fmt.Printf("月: %v \n", int(now.Month())) //月: 2
18.    fmt.Printf("日: %v \n", now.Day())
19.    fmt.Printf("时: %v \n", now.Hour())
20.    fmt.Printf("分: %v \n", now.Minute())
21.    fmt.Printf("秒: %v \n", now.Second())
22. }
23.
24.
```

### 按照指定格式

```
1.     //这个参数字符串的各个数字必须是固定的，必须这样写
2.     datestr2 := now.Format("2006/01/02 15:04:05")
3.     fmt.Println(datestr2)
4.     //选择任意的组合都是可以的，根据需求自己选择就可以（自己任意组合）。
5.     datestr3 := now.Format("2006 15:04")
6.     fmt.Println(datestr3)
```

## 内置函数

(1) 什么是内置函数/内建函数

Golang设计者为了编程方便，提供了一些函数，这些函数不用导包可以直接使用，我们称之为Go的内置函数/内建函数

(2) 内置函数存放位置

在builtin包下，直接用就行

(3) 常用函数：

1.len函数

统计字符串长度，按字节进行统计

2.new函数

分配内存，主要用来分配值类型 (int类型, float系列, bool, string, 数组和结构体struct)

```

main.go ...\\demo17  -o main.go ...\\demo18 x
> gocode > testproject01 > unit5 > demo18 > -o main.go
1 package main
2 import "fmt"
3 func main(){
4     //new分配内存, new函数的实参是一个类型而不是具体数值, new函数返回值是对应类型的指针 num: *int
5     num := new(int)
6     fmt.Printf("num的类型: %T, num的值是: %v, num的地址: %v, num指针指向的值是: %v",
7         num, num, &num, *num)
8 }

```

C:\Windows\system32\cmd.exe

```

D:\\goproject\\src\\gocode\\testproject01\\unit5\\cd demo17
D:\\goproject\\src\\gocode\\testproject01\\unit5\\demo17>go run main.go
6
D:\\goproject\\src\\gocode\\testproject01\\unit5\\demo17>cd ..
D:\\goproject\\src\\gocode\\testproject01\\unit5\\cd demo18
D:\\goproject\\src\\gocode\\testproject01\\unit5\\demo18>go run main.go
num的类型: *int, num的值是: 0xc0000100b0, num的地址: 0xc000006028, num指针指向的值是: 0

```

### 3. make函数

分配内存，主要用来分配引用类型（指针，slice切片，map，管道chan，interface等）

## 错误处理

```

it6 U -o dbutils.go U -o main.go ...\\main U -o main.go ...\\unit8 U x ⌂ ⌂ ...
src > gocode > testproject01 > unit8 > -o main.go
1 package main
2 import "fmt"
3 func main(){
4     test()
5     fmt.Println("上面的除法操作执行成功。。。")
6     fmt.Println("正常执行下面的逻辑。。。")
7 }
8
9 func test(){
10     num1 := 10
11     num2 := 0
12     result := num1/num2
13     fmt.Println(result)
14 }

```

PROBLEMS OUTPUT TERMINAL ...

powershell - unit8 + ⌂ ⌂ ... ^ x

- PS C:\\Users\\18417\\Desktop\\note\\goLang\\goproject> cd src\\gocode\\testproject01\\unit8
- PS C:\\Users\\18417\\Desktop\\note\\goLang\\goproject\\src\\gocode\\testproject01\\unit8 > go run main.go
- main.go:1:1: expected 'package', found 'EOF'
- PS C:\\Users\\18417\\Desktop\\note\\goLang\\goproject\\src\\gocode\\testproject01\\unit8 > go run main.go
- panic: runtime error: integer divide by zero

goroutine 1 [running]:  
main.test()  
C:/Users/18417/Desktop/note/goLang/goproject/src/gocode/testproject01/unit8/main.go:12 +0x11  
main.main()  
C:/Users/18417/Desktop/note/goLang/goproject/src/gocode/testproject01/unit8/main.go:4 +0x1d  
exit status 2  
PS C:\\Users\\18417\\Desktop\\note\\goLang\\goproject\\src\\gocode\\testproject01\\unit8 >

错误处理/捕获机制：

go中追求代码优雅，引入机制：defer+recover机制处理错误

```

func add(num1 int ,num2 int) int{
    //在Golang中，程序遇到defer关键字，不会立即执行defer后的语句，而是将defer后的语句压入一个栈中，然后继续执行函数后面
    defer fmt.Println("num1=",num1)
    defer fmt.Println(["num2=",num2])
    //栈的特点是：先进后出
    //在函数执行完毕以后，从栈中取出语句开始执行，按照先进后出的规则执行语句
    var sum int = num1 + num2 //230
    fmt.Println("sum=",sum)
    return sum
}

```

内置函数recover

```
src > gocode > testproject01 > unit8 > go main.go
1 package main
2 import "fmt"
3 func main(){
4     test()
5     fmt.Println("上面的除法操作执行成功。。。")
6     fmt.Println("正常执行下面的逻辑。。。")
7 }
8
9 func test(){
10    //利用defer+recover来捕获错误： defer后加上匿名函数的调
11    defer func() {
12        //调用recover内置函数，可以捕获错误：
13        err := recover()
14        if err != nil {
15            fmt.Println("错误已经捕获")
16            fmt.Println("err是:", err)
17        }
18    }()
19    num1 := 10
20    num2 := 0
21    result := num1/num2
22    fmt.Println(result)
23 }
```

PROBLEMS OUTPUT TERMINAL ...

powershell - unit8 + ×

● > go run main.go  
错误已经捕获  
err是: runtime error: integer divide by zero  
上面的除法操作执行成功。。。  
正常执行下面的逻辑。。。  
PS C:\Users\18417\Desktop\note\goLang\goproject\src\gocode\testproject01\unit8

自定义错误：需要调用errors包下的New函数： 函数返回error类型

```
src > gocode > testproject01 > unit9 > go main.go

1 package main
2 import "fmt"
3 import "errors"
4 func main(){
5     err := test()
6     if err != nil {
7         fmt.Println("自定义错误: ", err)
8     }
9     fmt.Println("上面的除法操作执行成功。。。")
10    fmt.Println("正常执行下面的逻辑。。。")
11 }
12
13 func test(){
14     num1 := 10
15     num2 := 0
16     if num2 == 0 {
17         //抛出自定义错误
18         return errors.New("除数不能为0")
19     }else { //如果除数不为0，那么正常执行就可以了
20         result := num1/num2
21         fmt.Println(result)
22         //如果没有错误，返回零值
23         return nil
24     }
}
```

有一种情况：程序出现错误以后，后续代码就没有必要执行，想让程序中断，退出程序：

有一种情况：程序出现错误以后，不能借助builtin包下的内置函数：panic

```

main.go  X
src > gocode > testproject01 > unit6 > demo02 > main.go
1 package main
2 import (
3     "fmt"
4     "errors"
5 )
6 func main(){
7     err := test()
8     if err != nil {
9         fmt.Println("自定义错误: ",err)
10    panic(err)
11 }
12 fmt.Println("上面的除法操作执行成功。。。")
13 fmt.Println("正常执行下面的逻辑。。。")
14 }
15
16 func test() (err error){
17     num1 := 10
18     num2 := 0
19     if num2 == 0 {
20         //抛出自定义错误:
21         return errors.New("除数不能为0哦~~")
22     }else{//如果除数不为0, 那么正常执行就可以了
23         result := num1 / num2
24         fmt.Println(result)
25         //如果没有错误, 返回零值:
26         return nil
27     }
28 }
29

```

D:\goproject\src\gocode\testproject01\unit6\demo02>go run main.go  
 自定义错误: 除数不能为0哦~~  
 panic: 除数不能为0哦~  
 goroutine 1 [running]:  
 main.main()  
 D:/goproject/src/gocode/testproject01/unit6/demo02/main.go:10 +0x1b  
 exit status 2  
 D:\goproject\src\gocode\testproject01\unit6\demo02>

## 数组

```

1. package main
2. import "fmt"
3. func main(){
4.     //声明数组:
5.     var arr [3]int16
6.     //获取数组的长度:
7.     fmt.Println(len(arr))
8.     //打印数组:
9.     fmt.Println(arr)//[0 0 0]
10.    //证明arr中存储的是地址值:
11.    fmt.Printf("arr的地址为: %p", &arr)
12.    //第一个空间的地址:
13.    fmt.Printf("arr的地址为: %p", &arr[0])
14.    //第二个空间的地址:
15.    fmt.Printf("arr的地址为: %p", &arr[1])
16.    //第三个空间的地址:
17.    fmt.Printf("arr的地址为: %p", &arr[2])
18. }

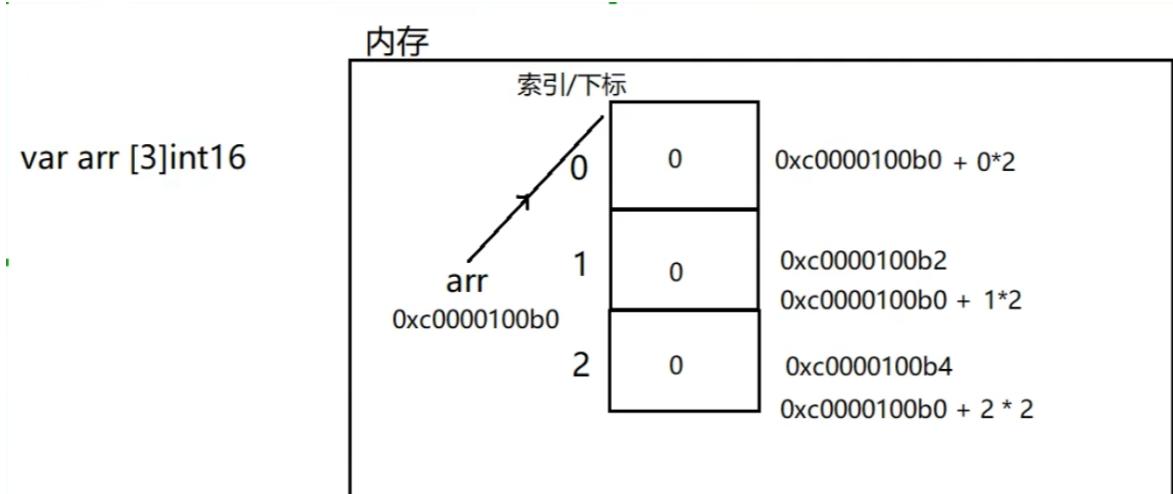
```

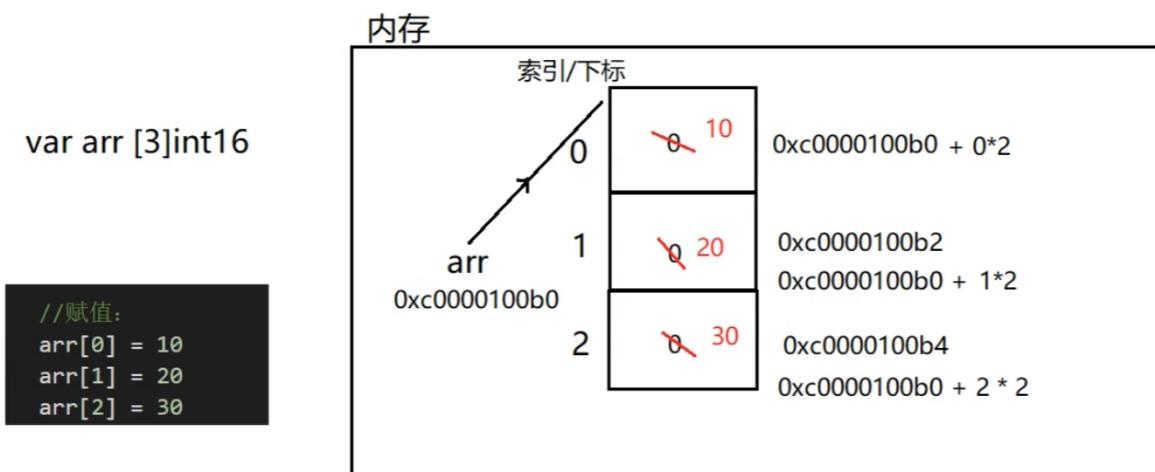
运行结果:

```

D:\goproject\src\gocode\testproject01\unit7\demo03>go run main.go
3
[0 0 0]
arr的地址为: 0xc0000100b0arr的地址为: 0xc0000100b2arr的地址为: 0xc0000100b4

```





## 【1】长度属于类型的一部分

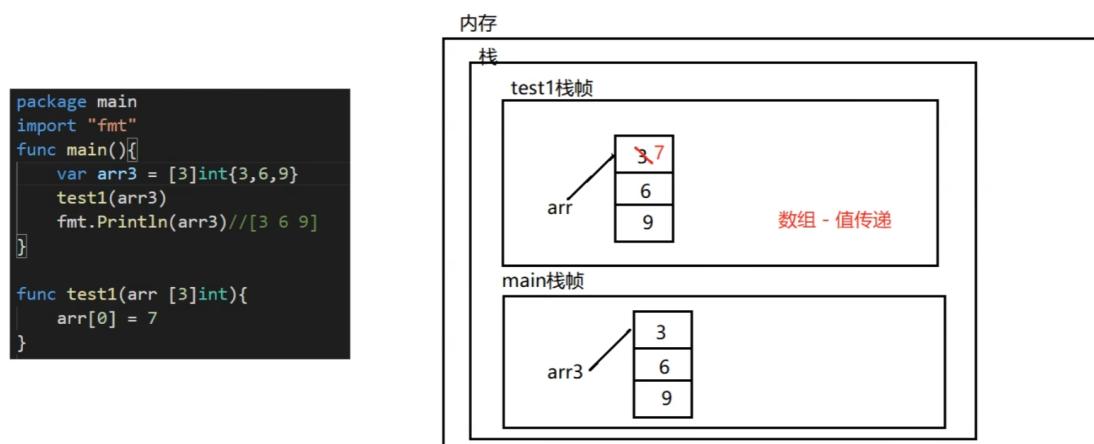
```
package main
import "fmt"
func main(){
    //定义一个数组:
    var arr1 = [3]int{3,6,9}
    fmt.Printf("数组的类型为: %T",arr1)

    var arr2 = [6]int{3,6,9,1,4,7}
    fmt.Printf("数组的类型为: %T",arr2)
}
```

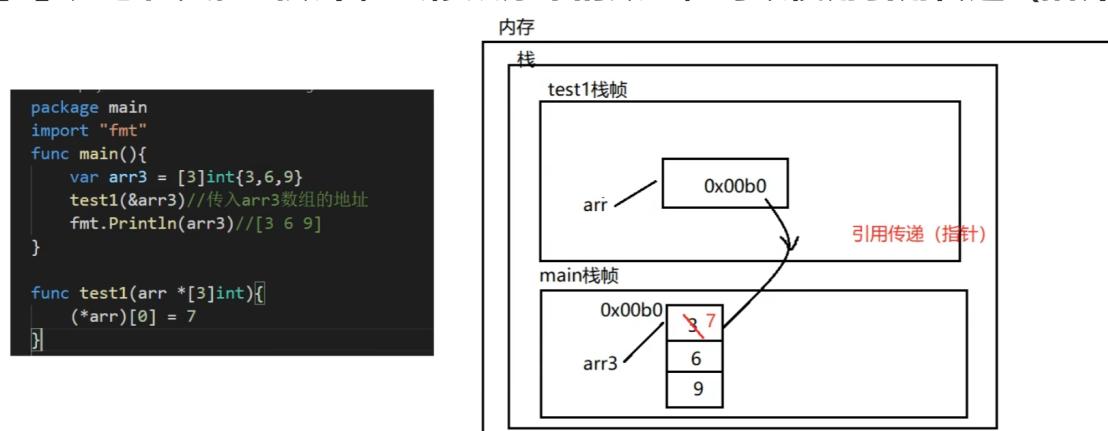
选择C:\Windows\system32\cmd.exe

D:\goproject\src\gocode\testproject01\unit7\demo06>go run main.go  
数组的类型为: [3]int  
数组的类型为: [6]int  
D:\goproject\src\gocode\testproject01\unit7\demo06>

## 【2】Go中数组属值类型，在默认情况下是值传递，因此会进行值拷贝



## 【3】如想在其他函数中，去修改原来的数组，可以使用引用传递（指针方式）



## 二维数组

```

package main
import "fmt"
func main(){
    //定义二维数组:
    var arr [2][3]int16
    fmt.Println(arr)
}

```

```

C:\Windows\system32\cmd.exe

D:\goproject\src\gocode\testproject01\unit7\demo07>go run main.go
[[0 0 0] [0 0 0]]

```

## 二维数组内存

```

package main
import "fmt"
func main(){
    //定义二维数组:
    var arr [2][3]int16
    fmt.Println(arr)

    fmt.Printf("arr的地址是: %p", &arr)
    fmt.Printf("arr[0]的地址是: %p", &arr[0])
    fmt.Printf("arr[0][0]的地址是: %p", &arr[0][0])

    fmt.Printf("arr[1]的地址是: %p", &arr[1])
    fmt.Printf("arr[0][0]的地址是: %p", &arr[1][0])
}

```

```

C:\Windows\system32\cmd.exe

D:\goproject\src\gocode\testproject01\unit7\demo07>go run main.go
[[0 0 0] [0 0 0]]
arr的地址是: 0xc0000100b0arr[0]的地址是: 0xc0000100b0arr[0][0]的地址是: 0xc0000100b0arr[1]的地址是: 0xc0000100b6arr[0][0]
[0 0 0]的地址是: 0xc0000100b6

```



## 二维数组赋值

```

//赋值:
arr[0][1] = 47
arr[0][0] = 82
arr[1][1] = 25
fmt.Println(arr)
}

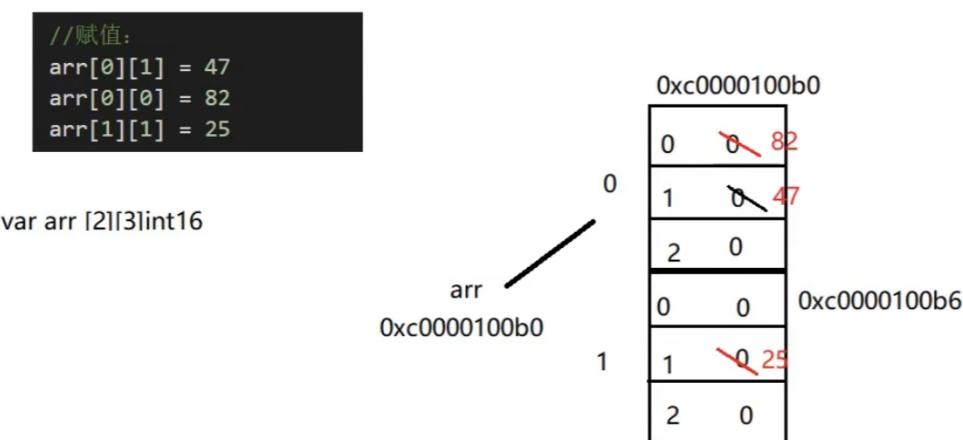
```

```

C:\Windows\system32\cmd.exe

D:\goproject\src\gocode\testproject01\unit7\demo07>go run main.go
[[0 0 0] [0 0 0]]
arr的地址是: 0xc0000100b0arr[0]的地址是: 0xc0000100b0arr[0][0]的地址是: 0xc0000100b0arr[1]的地址是: 0xc0000100b6
D:\goproject\src\gocode\testproject01\unit7\demo07>go run main.go
[[0 0 0] [0 0 0]]
arr的地址是: 0xc0000100b0arr[0]的地址是: 0xc0000100b0arr[0][0]的地址是: 0xc0000100b0arr[1]的地址是: 0xc0000100b6[[82 47 0] [0 25 0]]
D:\goproject\src\gocode\testproject01\unit7\demo07>

```



```

package main
import "fmt"

```

```

func main(){
    //实现的功能：给出五个学生的成绩，求出成绩的总和，平均数
    //给出五个学生的成绩：----> 数组储存
    //定义一个数组
    var scores [5]int
    scores[0] = 95
    scores[1] = 91
    scores[2] = 39
    scores[3] = 60
    scores[4] = 21
    //求和
    //定义一个变量专门接收成绩的和：
    sum := 0
    for i := 0;i < len(scores);i++ {
        sum += scores[i]
    }
    //平均数
    avg := sum / len(scores)
    //输出
    fmt.Printf("成绩的总和为: %v, 成绩的平均数为: %v\n",sum,avg)
    var arr [3]int16
    fmt.Println(len(arr))//获取长度
    fmt.Println(arr)//[0 0 0]
    fmt.Printf("arr的地址为: %p\n",&arr)//证明arr中存储的是地址
    //第一个空间的地址
    fmt.Printf("arr的地址为: %p\n",&arr[0])
    //第二个空间的地址
    fmt.Printf("arr的地址为: %p\n",&arr[1])
    //第三个空间的地址
    fmt.Printf("arr的地址为: %p\n",&arr[2])
    //将成绩存入数组
    var scores2 [5]int
    for i := 0;i < len(scores2);i++ {
        fmt.Printf("请录入第%d学生的成绩",i + 1)
        fmt.Scanln(&scores2[i])
    }
    //展示一下班级的每个学生的成绩：（数组进行遍历）
    for i := 0;i < len(scores2);i++ {
        fmt.Printf("第%d个学生的成绩为: %d\n",i+1,scores2[i])
    }
    fmt.Println("-----")
    for key,val := range scores2 {
        fmt.Printf("第%d个学生的成绩为: %d\n",key+1,val)
    }
    fmt.Println("-----")
    for _,val := range scores2 {
        fmt.Printf("学生的成绩为: %d\n",val)
    }
    //数组定义初始化
    //第一种
    var arr1 [3]int = [3]int{3,6,9}
    fmt.Println(arr1)
    //第二种
    var arr2 = [3]int{4,5,6}
    fmt.Println(arr2)
    //第三种，长度不指定
    var arr3 = [...]int{4,5,6,8}
    fmt.Println(arr3)
    //第四种，指定索引赋值 索引：数值
    var arr4 = [...]int{2:66,0:33,1:99,3:88}
    fmt.Println(arr4)
    fmt.Printf("%T",arr4)
    //二维数组
    var arr5 [2][3]int16
    fmt.Println(arr)
    fmt.Printf("arr5的地址是: %p\n",&arr5)
    fmt.Printf("arr5[0]的地址是: %p\n",&arr5[0])
    fmt.Printf("arr5[0][0]的地址是: %p\n",&arr5[0][0])
    fmt.Printf("arr5[1]的地址是: %p\n",&arr5[1])
    fmt.Printf("arr5[1][0]的地址是: %p\n",&arr5[1][0])
    //赋值：
    arr5[0][1] = 47
    arr5[0][0] = 82
    arr5[1][1] = 25
    fmt.Println(arr5)
    //初始化操作：
    var arr6 [2][3]int = [2][3]int{{1,4,7},{2,5,8}}

```

```

fmt.Println(arr6)
//数组遍历
var arr7 [3][3]int = [3][3]int{{1,4,7},{2,5,8},{3,6,9}}
fmt.Println(arr7)
fmt.Println("-----")
//方式1：普通for循环：
for i := 0;i < len(arr7);i++ {
    for j := 0;j < len(arr7[i]);j++ {
        fmt.Print(arr7[i][j],"\t")
    }
    fmt.Println()
}
fmt.Println("-----")
//方式2：for range循环：
for key,value := range arr7 {
    for k,v := range value {
        fmt.Printf("arr[%v][%v]=%v\t",key,k,v)
    }
    fmt.Println()
}
}

```

## 切片

【1】切片 (slice) 是golang中一种特有的数据类型

【2】数组有特定的用处，但是却有一些呆板（数组长度固定不可变），所以在Go语言的代码里面并不是特别常见。相对的切片却是随处可见的。切片是一种建立在数组类型之上的抽象，它构建在数组之上并且提供更强大的能力和便捷。

【3】切片 (slice) 是对数组一个连接片段的引用，所以切片是一个引用类型。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个相关数组的动态窗口。

```

main.go
1 package main
2 import "fmt"
3 func main(){
4     //定义数组:
5     var intarr [6]int = [6]int{3,6,9,1,4,7}
6     //切片构建在数组之上:
7     //定义一个切片名字为slice,[]动态变化的数组长度不写, int类型, intarr是原数组
8     //#[1:3]切片 - 切出的一段片段 - 索引:从1开始, 到3结束(不包含3) - [1,3]
9     //var slice []int = intarr[1:3]
10    slice := intarr[1:3]
11    //输出数组:
12    fmt.Println("intarr:",intarr)
13    //输出切片:
14    fmt.Println("slice:",slice)
15    //切片元素个数:
16    fmt.Println("slice的元素个数:",len(slice))
17    //获取切片的容量: 容量可以动态变化
18    fmt.Println("slice的容量:",cap(slice))
19 }

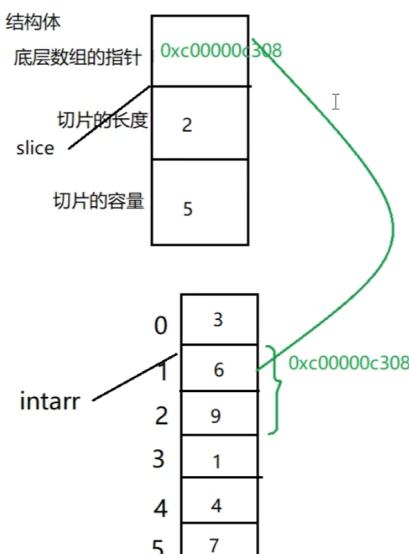
```

切片有3个字段的数据结构：一个是指向底层数组的指针，一个是切片的长度，一个是切片的容量。

```

func main(){}
//定义数组:
var intarr [6]int = [6]int{3,6,9,1,4,7}
//切片构建在数组之上:
//定义一个切片名字为slice,[]动态变化的数组长度不写, int类型
//#[1:3]切片 - 切出的一段片段 - 索引:从1开始, 到3结束(不包含3)
//var slice []int = intarr[1:3]
slice := intarr[1:3]
//输出数组:
fmt.Println("intarr:",intarr)
//输出切片:
fmt.Println("slice:",slice)
//切片元素个数:
fmt.Println("slice的元素个数:",len(slice))
//获取切片的容量: 容量可以动态变化
fmt.Println("slice的容量:",cap(slice))
}

```



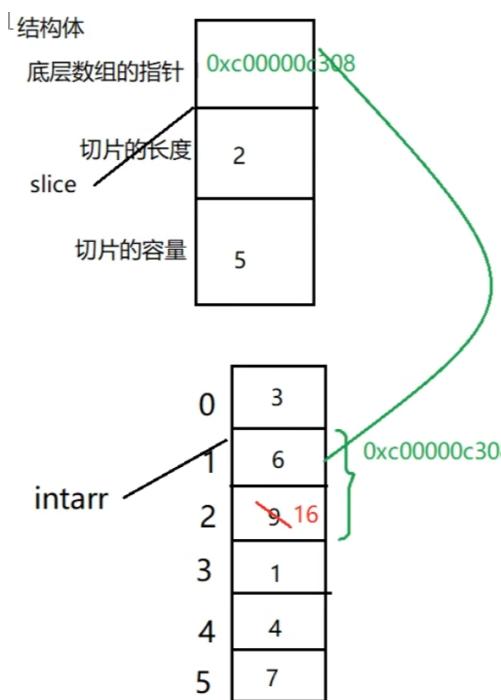
改变切片元素的值，也会改变原数组该索引的值（因为切片中存的是数组的指针）

```

fmt.Printf("数组中下标为1位置的地址: %p,&intarr[1])
fmt.Printf("切片中下标为0位置的地址: %p,&slice[0])
slice[1] = 16
fmt.Println("intarr:",intarr)
fmt.Println("slice:",slice)

```

```
数组中下标为1位置的地址: 0xc00000e398切片中下标为0位置的地址: 0xc00000e398intar
r: [3 6 16 1 4 7]
slice: [6 16]
```



## 切片的定义

【1】方式1：定义一个切片，然后让切片去引用一个已经创建好的数组

```
//定义数组:
var intarr [6]int = [6]int{3,6,9,1,4,7}
//切片构建在数组之上
//定义一个切片名字为slice，[]动态变化的数组长度不写，int类型，intarr是原数组
//[1:3]切片 - 切出的一段片段 - 索引: 从1开始, 到3结束 (不包含3) - ) [1,3)
//var slice []int = intarr[1:3]
slice := intarr[1:3]
```

【2】方式2：通过make内置函数来创建切片。基本语法：var切片名[type = make([],len,[cap])]

```
//定义切片: make函数的三个参数: 1.切片类型 2.切片长度 3.切片的容量
slice2 := make([]int,4,20)
fmt.Println(slice2)
fmt.Println("切片的长度:", len(slice2))
fmt.Println("切片的容量:", cap(slice2))
slice2[0] = 66
slice2[1] = 88
fmt.Println(slice2)
```

PS:make底层创建一个数组，对外不可见，所以不可以直接操作这个数组，要通过slice去间接的访问各个元素，不可以直接对数组进行维护/操作

【3】方式3：定一个切片，直接就指定具体数组，使用原理类似make方式。

```
slice2 := []int{1,4,7}
fmt.Println(slice2)
fmt.Println("切片的长度: ", len(slice2))
fmt.Println("切片的容量: ", cap(slice2))
```

输出结果：

切片的长度: 4
切片的容量: 20
[66 88 0 0]
[1 4 7]
切片的长度: 3
切片的容量: 3

## 切片遍历

```
//方式1: 普通for循环
for i := 0;i < len(slice2);i++ {
    fmt.Printf("slice[%v] = %v \t",i,slice2[i])
}
fmt.Println("\n-----")
//方式2: for-range循环:
for i, v := range slice2 {
    fmt.Printf("下标: %v, 元素: %v\n",i,v)
}
```

## 切片注意事项

【1】切片定义后不可以直接使用，需要让其引用到一个数组，或者make一个空间供切片来使用

```
package main
import "fmt"
func main(){
    //定义切片:
    var slice []int
    fmt.Println(slice)
}
```

选择C:\Windows\System32\cmd.exe  
D:\goproject\src\gocode\testproject01\unit8\demo03>cd ..  
D:\goproject\src\gocode\testproject01\unit8>cd demo04  
D:\goproject\src\gocode\testproject01\unit8\demo04>go run main.go  
[]

【2】切片使用不能越界。

```
package main
import "fmt"
func main(){
    //定义数组
    var intarr [6]int = [6]int{1,4,7,2,5,8}
    //定义切片:
    var slice []int = intarr[1:4]
    fmt.Println(slice[0])
    fmt.Println(slice[1])
    fmt.Println(slice[2])
    fmt.Println(slice[3])
}
```

D:\goproject\src\gocode\testproject01\unit8\demo04>go run main.go  
4  
7  
2  
panic: runtime error: index out of range [3] with length 3  
goroutine 1 [running]:  
main.main()  
D:/goproject/src/gocode/testproject01/unit8/demo04/main.go:11 +0x1bd  
exit status 2  
D:\goproject\src\gocode\testproject01\unit8\demo04>

【3】简写方式

- 1) var slice = arr[0:end] -----> var slice = arr[:end]
- 2) var slice = arr[start:len(arr)] -----> var slice = arr[start:]
- 3) var slice = arr[0:len(arr)] -----> var slice = arr[:]

【4】切片可以继续切片

```
//对切片再次切片:
slice2 := slice[1:2]
fmt.Println(slice2) //7
slice2[0] = 66
fmt.Println(intarr)
fmt.Println(slice)
```

D:\goproject\src\gocode\testproject01\unit8\demo04>go run main.go  
[7]  
[1 4 66 2 5 8]  
[4 66 2]  
D:\goproject\src\gocode\testproject01\unit8\demo04>

【5】切片可以动态增长

```
//切片动态增长
//定义数组:
var intarr2 [6]int = [6]int{1,4,7,3,6,9}
//定义切片
var slice4 []int = intarr2[1:4]
fmt.Println(len(slice))

slice5 := append(slice4,88,50)
fmt.Println(slice5)
fmt.Println(slice4)
//底层原理:
//1.底层追加元素的时候对数组进行扩容，老数组扩容为新数组:
//2.创建一个新数组，将老数组中的4, 7, 3复制到新数组中，在新数组中追加88, 50
//3.slice5底层数组的指向，指向的是新数组
//4.往往我们在使用追加的时候其实想要做的效果给slice4追加
slice4 = append(slice4,88,50)
fmt.Println(slice4)
//5.底层的新数组 不能直接维护 需要通过切片间接维护操作
```

可以通过append函数将切片追加给切片，**三个点必写**

```
slice3 := []int{99,44}
slice = append(slice,slice3...)
fmt.Println(slice)
```

【6】切片的拷贝

```
//拷贝
//定义切片:
var a []int = []int{1,4,7,3,6,9}
var b []int = make([]int,10)

copy(b,a)//将a中对应数组中元素内容复制到b中对应的数组中
fmt.Println(b)
```

```
package main
import "fmt"
```

```

func main(){
    //定义数组:
    var intarr [6]int = [6]int{3,6,9,1,4,7}
    //切片构建在数组之上
    //定义一个切片名字为slice, []动态变化的数组长度不写, int类型, intarr是原数组
    //[:3]切片 - 切出的一段片段 - 索引: 从1开始, 到3结束(不包含3) - ) [1,3)
    //var slice []int = intarr[1:3]
    slice := intarr[1:3]
    //输出数组:
    fmt.Println("intarr",intarr)
    //输出切片
    fmt.Println("intarr",slice)
    //切片元素个数:
    fmt.Println("slice的元素个数",len(slice))
    //获取切片的容量: 容量可以动态变化
    fmt.Println("slice的容量",cap(slice))
    fmt.Printf("数组中下标为1位置的地址: %p",&intarr[1])
    fmt.Printf("切片中下标为0位置的地址: %p",&slice[0])
    slice[1] = 16
    fmt.Println("intarr:",intarr)
    fmt.Println("slice:",slice)
    //定义切片: make函数的三个参数: 1.切片类型 2.切片长度 3.切片的容量
    slice2 := make([]int,4,20)
    fmt.Println(slice2)
    fmt.Println("切片的长度:",len(slice2))
    fmt.Println("切片的容量:",cap(slice2))
    slice2[0] = 66
    slice2[1] = 88
    fmt.Println(slice2)
    slice3 := []int{1,4,7}
    fmt.Println(slice3)
    fmt.Println("切片的长度:",len(slice3))
    fmt.Println("切片的容量:",cap(slice3))
    slice2[2] = 99
    slice2[3] = 100
    //方式1: 普通for循环
    for i := 0;i < len(slice2);i++ {
        fmt.Printf("slice[%v] = %v \t",i,slice2[i])
    }
    fmt.Println("\n-----")
    //方式2: for-range循环:
    for i, v := range slice2 {
        fmt.Printf("下标: %v, 元素: %v\n",i,v)
    }
    //切片动态增长
    //定义数组:
    var intarr2 [6]int = [6]int{1,4,7,3,6,9}
    //定义切片
    var slice4 []int = intarr2[1:4]
    fmt.Println(len(slice4))
    slice5 := append(slice4,88,50)
    fmt.Println(slice5)
    fmt.Println(slice4)
    //底层原理:
    //1.底层追加元素的时候对数组进行扩容, 老数组扩容为新数组:
    //2.创建一个新数组, 将老数组中的4, 7, 3复制到新数组中, 在新数组中追加88, 50
    //3.slice5底层数组的指向, 指向的是新数组
    //4.往往我们在使用追加的时候其实想要做的效果给slice4追加
    slice4 = append(slice4,88,50)
    fmt.Println(slice4)
    //5.底层的新数组 不能直接维护 需要通过切片间接维护操作
    slice6 := append(slice4,slice5...)
    fmt.Println(slice6)
    //拷贝
    //定义切片:
    var a []int = []int{1,4,7,3,6,9}
    var b []int = make([]int,10)
    copy(b,a)//将a中对应数组中元素内容复制到b中对应的数组中
    fmt.Println(b)
}

```

## 映射

【1】映射 (map) , Go语言中内置的一种类型, 它将键值相关联, 我们可以通过键key来获取对应的值value。类似于其他语言的集合

## 【2】基本语法

```
var map 变量名 map[keytype]valuetype
```

PS: key, value的类型: bool, 数字, string。指针, channel, 还可以是只包含前面几个类型的接口, 结构体, 数组

PS: key通常为int, string类型, value通常为数字(整数, 浮点数), string, map, 结构体

PS: slice, map, function不可以

## 【3】

map的特点:

- (1) map集合在使用前一定要make
- (2) map的key-value是无序的
- (3) key是不可重复的, 如果遇到重复, 后一个value会替换前一个value
- (4) value是可以重复的
- (5) make函数的第二个参数size可以省略, 默认就分配一个内存

## map定义方式

```
//方式1
//定义map变量
var a map[int]string
//只声明map内存是没有分配空间
//必须通过make函数进行初始化, 才会分配空间
a = make(map[int]string, 10)//map可以存放10个键值对
//将键值对存入map中
a[1] = "张三"
a[2] = "里斯"
a[3] = "王五"
//输出集合
fmt.Println(a)

//方式2
b := make(map[int]string)
b[1] = "张三"
b[2] = "里斯"
fmt.Println(b)

//方式3
c := map[int]string{
    1 : "张三",
    2 : "里斯",
}
c[3] = "王五"
fmt.Println(c)
```

## map的操作

### 【1】增加和更新操作

map["key"] = value ----> 如果key还没有, 就是增加, 如果key存在就是修改

### 【2】删除操作

delete(map, "key"), delete是一个内置函数, 如果key存在, 就删除该key-value, 如果key不存在, 不操作, 但是也不会报错

### 【3】清空操作:

(1) 如果我们要删除map的所有key', 没有一个专门的方法一次删除, 可以遍历一下key, 逐个删除

(2) 或者map = make(...), make一个新的, 让原来的成为垃圾, 被gc回收

### 【4】查找操作

value, bool = map[key]

value为返回的value, bool为是否返回, 要么true要么false

### 【5】获取长度: len函数

len()

### 【6】遍历: for-range

```
package main
```

```

import "fmt"
func main() {
    //方式1
    //定义map变量
    var a map[int]string
    //只声明map内存是没有分配空间
    //必须通过make函数进行初始化，才会分配空间
    a = make(map[int]string,10)//map可以存放10个键值对
    //将键值对存入map中
    a[1] = "张三"
    a[2] = "里斯"
    a[3] = "王五"
    //输出集合
    fmt.Println(a)
    //方式2
    b := make(map[int]string)
    b[1] = "张三"
    b[2] = "里斯"
    fmt.Println(b)
    //方式3
    c := map[int]string{
        1 : "张三",
        2 : "里斯",
    }
    //增加
    c[3] = "王五"
    fmt.Println(c)
    //删除
    delete(c,3)
    fmt.Println(c)
    //查找:
    value,flag := c[3]
    fmt.Println(value)
    fmt.Println(flag)
    //获取长度
    fmt.Println(len(a))
    //遍历:
    for k,v := range b {
        fmt.Printf("key为: %v value为 %v]\t",k,v)
    }
    fmt.Println("-----")
    //加深难度
    d := make(map[string]map[int]string)
    //赋值
    d["class1"] = make(map[int]string,3)
    d["class1"][1] = "Jhon"
    d["class1"][2] = "Tom"
    d["class1"][3] = "Tim"
    d["class2"] = make(map[int]string,3)
    d["class2"][1] = "Jhon2"
    d["class2"][2] = "Tom2"
    d["class2"][3] = "Tim2"
    for k1,v1 := range d {
        fmt.Println(k1)
        for k2,v2 := range v1 {
            fmt.Printf("学生学号为: %v 学生姓名为 %v \t",k2,v2)
        }
        fmt.Println()
    }
}

```

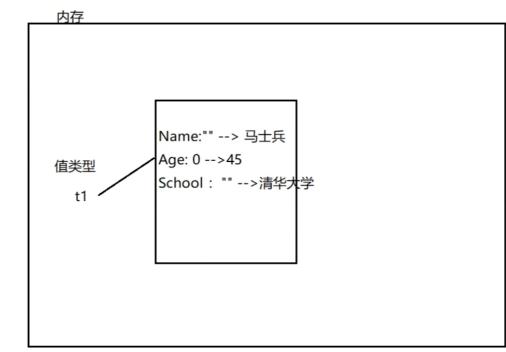
## 面向对象

- (1)Golang也支持面向对象编程（oop），但是和传统的面向对象编程有区别，并不是纯粹的面向对象语言。所以我们说Golang支持面向对象编程特性是比较准确的。
- (2)Golang没有类（class），Go语言的结构体（struct）和其他编程语言的类（class）有同等地位，你可以理解Golang是基于struct来实现OOP特性的。
- (3)Golang面向对象编程非常简洁，去掉了传统OOP语言的方法重载，构造函数和析构函数，隐藏的this指针等等
- (4)Golang仍然有面向对象编程的继承，封装和多态的特性，只是实现的方式和其他OOP语言不一样，比如继承：Golang没有extends关键字，继承是通过匿名字段来实现

## 结构体的引入

```
//定义老师结构体，将老师中的各个属性 统一放入结构体中管理:
type Teacher struct{
    //变量名字大写外界可以访问这个属性
    Name string
    Age int
    School string
}
```

```
//创建老师结构体的实例、对象、变量:
var t1 Teacher // var a int
fmt.Println(t1) //在未赋值时默认值: { 0 }
t1.Name = "马士兵"
t1.Age = 45
t1.School = "清华大学"
fmt.Println(t1)
fmt.Println(t1.Age + 10)
```



## 结构体实例创建方式

### 【1】直接创建

```
package main
import "fmt"

//定义老师结构体，将老师中的各个属性 统一放入结构体中管理:
type Teacher struct{
    //变量名字大写外界可以访问这个属性
    Name string
    Age int
    School string
}

func main(){
    //创建老师结构体的实例、对象、变量:
    var t1 Teacher // var a int
    fmt.Println(t1) //在未赋值时默认值: { 0 }
    t1.Name = "马士兵"
    t1.Age = 45
    t1.School = "清华大学"
    fmt.Println(t1)
    fmt.Println(t1.Age + 10)
}
```

### 【2】

```
package main
import "fmt"

//定义老师结构体，将老师中的各个属性 统一放入结构体中管理:
type Teacher struct{
    //变量名字大写外界可以访问这个属性
    Name string
    Age int
    School string
}
func main(){
    //创建老师结构体的实例、对象、变量:
    var t Teacher = Teacher{"赵珊珊",31,"黑龙江大学"}
    fmt.Println(t)
    // t.Name = "赵珊珊"
    // t.Age = 31
    // t.School = "黑龙江大学"
    fmt.Println(t)
}
```

### 【3】

```
func main(){
    //创建老师结构体的实例、对象、变量:
    var t *Teacher = new(Teacher)
    //t是指针，t其实指向的就是地址，应该给这个地址的指向的对象的字段赋值:
    (*t).Name = "马士兵"
    (*t).Age = 45 /*的作用：根据地址取值
    //为了符合程序员的编程习惯，go提供了简化的赋值方式:
    t.School = "清华大学" //go编译器底层对t.School转化 (*t).School = "清华大学"
    fmt.Println(*t)
}
```

### 【4】

```
func main(){
    //创建老师结构体的实例、对象、变量:
    var t *Teacher = &Teacher{"马士兵",46,"清华大学"}
    // (*t).Name = "马士兵"
    // (*t).Age = 45
    // t.School = "清华大学"
    fmt.Println(*t)
}
```

## 结构体之间的转换

### 【1】结构体是用户单独定义的类型，和其他类型进行转换时需要有完全相同的字段（名字，个数和类型）

```

1. package main
2. import "fmt"
3. type Student struct {
4.     Age int
5. }
6.
7. type Person struct {
8.     Age int
9. }
10.
11. func main() {
12.     var s Student = Student{10}
13.     var p Person = Person{10}
14.     s = Student(p)
15.     fmt.Println(s)
16.     fmt.Println(p)
17. }
```

【2】结构体进行type重新定义（相当于取别名），Golang认为是新的数据类型，但是相互间可以强转

```

1. package main
2. import "fmt"
3. type Student struct {
4.     Age int
5. }
6.
7. type Stu Student
8.
9. func main() {
10.     var s1 Student = Student{19}
11.     var s2 Stu = Stu{19}
12.     s1 = Student(s2)
13.     fmt.Println(s1)
14.     fmt.Println(s2)
15.
16. }
```

## 方法的引入

【1】方法是作用在指定的数据类型上，和指定的数据类型绑定，因此自定义类型，都可以有方法，而不仅仅是struct

【2】方法的声明和调用格式：

声明：

```

1. type A struct {
2.     Num int
3. }
4. func (a A) test() {
5.     fmt.Println(a.Num)
6. }
```

调用：

```
var a A
a.test()
```

(1)func (a A) test()相当于A结构体有一个方法叫test

(2)(a A)体现方法test和结构体A绑定关系

```

package main
import "fmt"
//定义Person结构体
type Person struct{
    Name string
}

//给Person结构体绑定方法test
func (p Person) test(){//参数名字随便起
    fmt.Println(p.Name)
}

func main(){
    //创建结构体对象：
    var p Person
    p.Name = "丽丽"
    p.test()
}
```

注意：

- (1) test方法中参数的名字随意起
- (2) 结构体Person和test方法绑定，调用test方法必须靠指定的类型：Person
- (3) 如果其他类型变量调用test方法一定会报错
- (4) 结构体对象传入test方法中，值传递，和函数参数传递一致

```

package main
import "fmt"
//定义Person结构体
type Person struct{
    Name string
}

//给Person结构体绑定方法test
func (p Person) test(){//参数名字随便起
    p.Name = "露露"
    fmt.Println(p.Name)
}

func main(){
    //创建结构体对象:
    var p Person
    p.Name = "丽丽"
    p.test()
    fmt.Println(p.Name)
}

```

## 方法注意事项

【1】结构体类型是值类型，在方法调用中，遵守值类型的传递机制，是值拷贝传递方式

```

package main
import "fmt"
//定义Person结构体
type Person struct{
    Name string
}

//给Person结构体绑定方法test
func (p Person) test(){//参数名字随便起
    p.Name = "露露"
    fmt.Println(p.Name)
}

func main(){
    //创建结构体对象:
    var p Person
    p.Name = "丽丽"
    p.test()
    fmt.Println(p.Name)
}

```

【2】如程序员希望在方法中，修改结构体变量的值，可以通过结构体指针的方式来处理

```

package main
import "fmt"
//定义Person结构体
type Person struct{
    Name string
}

//给Person结构体绑定方法test
func (p *Person) test(){//参数名字随便起
    (*p).Name = "露露"
    fmt.Println((*p).Name)
}

func main(){
    //创建结构体对象:
    var p Person
    p.Name = "丽丽"
    fmt.Printf("p的地址为: %p", &p)
    (&p).test()
    fmt.Println(p.Name)
}

```

写程序的时候，可以直接简化，底层编译器做了优化，底层会自动帮我们加上&，\*

```

1 package main
2 import "fmt"
3 //定义Person结构体
4 type Person struct{
5     Name string
6 }
7
8 //给Person结构体绑定方法test
9 func (p *Person) test(){//参数名字随便起
10     p.Name = "露露"
11     fmt.Println(p.Name)
12 }
13
14
15 func main(){
16     //创建结构体对象:
17     var p Person
18     p.Name = "丽丽"
19     fmt.Printf("p的地址为: %p",&p)
20     p.test()
21     fmt.Println(p.Name)
22 }
23

```

【3】Golang中的方法作用在指定的数据类型上，和指定的数据类型绑定，因此自定义类型，都可以有方法，而不仅仅是struct，比如int, float32等都可以有方法

```

package main
import "fmt"

type integer int

func (i integer) print(){
    i = 30
    fmt.Println("i = ",i)
}

func (i *integer) change(){
    *i = 30
    fmt.Println("i = ",*i)
}

func main(){
    var i integer = 20
    i.print()
    i.change()
    fmt.Println(i)
}

```

```

选择C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19042.985]
(c) Microsoft Corporation。保留所有权利。
D:\goproject\src\gocode\testproject01\unit
i = 20
D:\goproject\src\gocode\testproject01\unit
i = 30
20
D:\goproject\src\gocode\testproject01\unit
i = 30
20
D:\goproject\src\gocode\testproject01\unit
i = 30
i = 30
30
D:\goproject\src\gocode\testproject01\unit

```