Go基础

基础语法

流程语句

- goto语句

函数

- 函数基础
- 函数变量的作用范围
- defer
- 递归函数（自己调用自己）

高级函数（面试重点！）

- 函数的本质
- 匿名函数
- 回调函数
- 闭包结构

数组

- 初识数组
- 数组初始化
- 遍历数组元素
- 数组是值类型
- 多维数组的遍历
- 冒泡排序

切片

- 定义和初始化
- 切片是引用类型传递
- 切片的扩容
  - 切片的扩容分析
  - 深浅拷贝

集合(Map) [key,value]

- map的使用

指针
- 指针的套娃使用
- 数组指针
- 指针函数 & 指针作为参数

结构体
- 结构体的定义和使用
- 结构体指针应用
- 匿名结构体
- 结构体嵌套
- 导出结构体和字段

Go的面向对象思想
- 继承
- 方法和方法重写
- 接口实现
- 多态
- 空接口
- 接口的继承
- 接口断言
- Type别名

错误
- 创建自己的错误信息
- 自定义error
- panic
- recover

基础语法

变量的定义和初始化 Go 复制代码

```go
    //no new variables on left side of :=   必须是一个新的变量才可以使用：=符号
    var name = "xiaowang"
    //name := "小王"
    age := 18
    fmt.Printf("%T,%T", name, age)
    var age float64 //定义一个浮点型变量
    age = 0.5
    const URL string = "www.baidu.com" // 显示定义
    const URL2 = "www.baidu.com"  //隐式定义
```

地址用&c语言中的取地址符号 Go 复制代码

```go
    var num int
    num = 100
    fmt.Printf("值：%d,字符地址：%p", num, &num) //取地址符
```

//变量交换 Go 复制代码

```go
    var a int = 10
    var b int = 20
    a,b = b, a
    fmt.Println(a, b)   //变量交换
```

变量的作用域：在main函数外面就是 全局变量 再里面就是 局部变量 根据就近原则使用

流程语句

格式化打印占位符： Go 复制代码

```
     %v,原样输出   %T，打印类型      %t,bool类型        %s，字符串        %f，浮点
    %d，10进制的整数   %b，2进制的整数  %o，8进制      %x，%X，16进制     %x：0-9，a-f
    %X：0-9，A-F      %c，打印字符     %p，打印地址
```

业务：登录密码的校验

Go | 复制代码

```go
var a, b int
    var pwd int = 20201101
    fmt.Println("请输入密码：")
    fmt.Scan(&a)
    if a == pwd {
        fmt.Println("第一次输入密码正确,请输入第二次密码")
        fmt.Scan(&b)
        if b == pwd {
            fmt.Println("登陆成功")
        } else {
            fmt.Println("登陆失败")
        }
    } else {
        fmt.Println("密码错误，登陆失败")
    }

```

goto语句

goto：可以无条件地转移到过程中指定的行。

goto用法

Go | 复制代码

```go
package main

import "fmt"

func main() {
    /* 定义局部变量 */
    var a int = 10

    /* 循环 */
    LOOP: for a < 20 {
       if a == 15 {
          /* 跳过迭代 */
          a = a + 1
          goto LOOP
       }
        fmt.Printf("a的值为 : %d\n", a)
        a++
    }
}
```

统一错误处理 多处错误处理存在代码重复时是非常棘手的，例如：

这里不知道是好处还是坏处

Go | 复制代码

```go
err := firstCheckError()
if err != nil {
        goto onExit
    }
err = secondCheckError()
if err != nil {
        goto onExit
    }
fmt.Println("done")
return
onExit:
fmt.Println(err)
exitProcess()
```

函数

函数基础

Go | 复制代码

```go
/*func 函数名 （参数，参数 。。。）函数调用后的返回值{
    函数体：执行一段代码
    return 返回结果
}*/

//*** 形式参数与实际参数必须一一对应 ，顺序，个数，类型
//形式参数：定义函数时，用来接收外部传入数据的参数，就是形式参数
//实际参数：调用函数时，传给形参的实际数据，就是形式参数

package main
import "fmt"
func main() {
    //调用函数
    getSum(1, 2, 3, 4, 6, 7, 89, 6, 5)  // 函数名加括号(括号里面是实际参数)进行调用
}
//定义一个 有参数无返回值的函数
//函数名后面就是形式参数
func getSum(num ...int) {
    //三点代表的就是可变参数  可以传入多个数值
    //注意：  可变参数必须要放在参数列表的最后且一个函数只有一个可变参数！！
    sum := 0
    for i := 0; i < len(num); i++ {
        fmt.Println(num[i])
        sum += num[i]
    }
    fmt.Println(sum)
}

```

函数变量的作用范围

Go | 复制代码

```go
package main

import (
    "fmt"
)

// 全局变量
var num int = 100

func main() {

    //函数体内的局部变量
    temp := 100
    fmt.Println(temp)

    //if,for语句定义的一次性变量局部变量
    if b := 1; b <= 10 {
        temp := 20
        fmt.Println(temp) //局部变量遵循就近原则
        fmt.Println(b)
    }

    num := 20
    fmt.Println(num)
    f1()
    f2()
}

func f1() {
    a := 30
    fmt.Println(a)
}
func f2() {

    //fmt.Println(a)  不能其他函数上使用其他函数定义的变量
    num := 40
    fmt.Println(num)
}

```

defer

注意： 函数前面加defer 这个函数会在程序最后执行 ， 但是该函数所传入的参数是当时传入的参数

defer Go 复制代码

```go
package main

import "fmt"

func main() {
    a := 10
    fmt.Println("start :a", a)
    defer num2(a) //这里是先调，用后执行 程序走到这里时a已经传到里面了
    a++
    fmt.Println("end: a", a)
}
func num2(n int) {
    fmt.Println("函数中的:  a", n)
}
```

递归函数（自己调用自己）

递归函数 Go 复制代码

```go
package main

import "fmt"

// 递归函数  自己调用自己  递归是很占用内存的
func main() {
    sum := getSum2(6)
    fmt.Println(sum)
}
func getSum2(n int) int {
    if n == 1 {
        return 1
    }
    return getSum2(n-1) + n
}
```

高级函数（面试重点！）

函数的本质

本质

Go 复制代码

```go
package main

import "fmt"

// func ()   本身就是一个数据类型
func main() {
    //f3不加括号 就是一个变量
    //f3 () 加括号就是调用函数
    fmt.Printf("%T\n", f3) //func(int, int) | func(int, int) int
    //定义函数类型的变量
    var f5 func(a, b int)
    f5 = f3 //共用一个地址   所以是引用类型的
    f5(1, 2)
}
func f3(a, b int) {
    fmt.Println(a, b)
}
```

匿名函数

匿名函数：没有名字的函数。

定义一个匿名函数，直接进行调用。通常只能使用一次。也可以使用匿名函数赋值给某个函数变量，那么就可以调用多次了。

匿名函数：

Go语言是支持函数式编程：

1.将匿名函数作为另一个函数的参数，回调函数

2.将匿名函数作为另一个函数的返回值，可以形成闭包结构。

Go | 复制代码

```go
package main

import (
    "fmt"
)

func main() {
    f7()
    f9 := f7              //函数本身就是一个变量
    fmt.Printf("%T", f7) // 也可以是一个类型
    f9()

    //匿名函数
    f5 := func() {
        fmt.Println("我是f5匿名函数")
    }
    f5()

    fmt.Println("==================")

    //匿名函数自己调用自己  在函数的末尾加上一个括号
    func() {
        fmt.Println("我匿名自己调用自己")
    }()
    fmt.Println("====================")
    //匿名函数也是可以传入参数的
    func(a, b int) {
        fmt.Println("我匿名函数可以传参数")
        fmt.Println(a, b)

    }(2, 5)

    //当然也可以有返回值  不会在函数的前面要有一个变量来接住
    sum := func(a, b int) int {
        fmt.Println("我匿名函数可以传参数，也有返回值")
        return a + b
    }(2, 5)
    fmt.Println(sum)
}
func f7() {
    fmt.Println("我是f7函数")
}

```

回调函数

高阶函数：根据Go语言的数据类型的特点，把一个函数作为另一个函数的参数

fun1（），fun2（）

将fun1函数作为fun2函数的参数

fun2函数：就叫做高阶函数，接收了一个函数作为参数的函数

fun1函数：就叫做回调函数，作为另一个函数的参数

```go
package main

import "fmt"

// 高级函数 回调函数    将函数最为一个函数的参数 进行传递
func main() {

    //当遇到一个业务   想用同一个方法执行产生不同的结果的时候 就可以使用回调函数（此函数作为一个函数的参数）
    r2 := oper(2, 4, add)
    fmt.Println(r2)
    fmt.Println(oper(6, 2, delete))

    //将一个匿名函数作为一个函数的参数（这个匿名函数是回调函数）
    r3 := oper(8, 4, func(a int, b int) int {
        if b == 0 {
            fmt.Println("除数不能为0")
            return 0
        }
        return a / b
    })
    fmt.Println(r3)
}
func oper(a, b int, f func(int, int) int) int {
    r := f(a, b)
    return r
}

func add(a, b int) int {
    return a + b
}

func delete(a, b int) int {
    return a - b
}
```

闭包结构

将匿名函数作为另一个函数的返回值，可以形成闭包结构。

闭包(closure):

一个外层函数中，有内层函数，该内层函数中，会操作外层函数的局部变量(外层函数中的参数，或者外层函数中直接定义的变量)，并且该外层函数的返回值就是这个内层函数。

这个内层函数和外层函数的局部变量，统称为闭包结构。

局部变量的生命周期会发生改变，**正常的局部变量随着函数调用而创建，随着函数的结束而销毁。**

但是闭包结构中的外层函数的局部变量并不会随着外层函数的结束而销毁，因为内层函数还要继续使用。

闭包结构 Go 复制代码

```go
package main
import "fmt"
func main() {
    r1 := increment()
    v1 := r1()
    fmt.Println(v1)
    fmt.Println(r1())
    r2 := increment() //重新调用函数，前面的程序就会销毁  重新开始自增
    fmt.Println(r2())
    fmt.Println(r2())
    fmt.Println(r1()) //虽然程序销毁，但是它里面的变量还是会存在

}
func increment() func() int { //用匿名函数最为一个函数的返回值  就是闭包
    var i int = 0
    fun := func() int { //用fun接匿名函数返回的值
        i++
        return i
    }
    return fun //返回匿名函数所返回的值
}
```

数组

初识数组

认识数组 Go 复制代码

```go
package main

import "fmt"

func main() {
    var nums [5]int

    nums[0] = 1
    nums[1] = 2
    nums[2] = 3
    nums[3] = 4

    //通过下标得到相应的值
    fmt.Println(nums[0])

    //得到数组的类型  就是定义的类型
    fmt.Printf("%T", nums)

    fmt.Println(nums) //修改前
    nums[0] = 100     //修改数组的值
    fmt.Println(nums) //修改后

    fmt.Println(len(nums)) //数组的长度
    fmt.Println(cap(nums)) //数组的容量

}

```

数组初始化

数组的其他创建方式：

Go | 复制代码

```go
package main

import "fmt"

func main() {
    //常规初始化
    var arr1 = []int{1, 2, 3, 4}
    fmt.Println(arr1)

    //直接赋值
    arr2 := [2]int{3, 4}
    fmt.Println(arr2)

    //如果设置了数组的长度 我们可以用冒号进行赋值，没有赋值的默认为0
    arr4 := [5]int{0: 144, 3: 224}
    fmt.Println(arr4)

    //不规定数组的长度
    arr3 := [...]string{" xiaowang", "chenyueyue", "halou"}
    fmt.Println(arr3)

    //
}

```

遍历数组元素

遍历数组 Go 复制代码

```go
package main

import "fmt"

// 遍历方式
func main() {

    nums := [5]int{4, 6, 8, 5, 4}
    //for i遍历
    for i := 0; i < 5; i++ {
        fmt.Println(nums[i])
    }

    fmt.Println("===========range============")
    //nums.range
    //index 下标  value 值
    for index, value := range nums {
        fmt.Println(index, value)
    }
}

```

数组是值类型

数组是值类型 Go 复制代码

```go
package main

import "fmt"

// 探究 值类型（拷贝） 引用类型
func main() {
    num := 10
    fmt.Println(num)
    age := num //相当于拷贝  不是使用的同一个地址
    fmt.Println(age)
    age = 20          //重新给数据赋值
    fmt.Println(age) //此时 age 发生改变
    fmt.Println(num) //但是num 由于是值类型    所以num还是10

    //数组也是值传递  与上面说法相似
    arr := [3]int{2, 4, 5}
    arr2 := arr
    fmt.Println(arr2)
    fmt.Println(arr)

    arr2[2] = 100
    fmt.Println(arr2)
    fmt.Println(arr)
}

```

多维数组的遍历

二维数组的遍历方式

Go | 复制代码

```go
package main

import "fmt"

func main() {
    /*

一维数组：存储的多个数据是数值本身
a1 :=[3]int{1,2,3}

二维数组：存储的是一维的一维
a2 := [3][4]int{{},{},{}}

该二维数组的长度，就是3。
存储的元素是一维数组，一维数组的元素是数值，每个一维数组长度为4。

多维数组：。。。

*/
    a2 := [3][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}
    fmt.Println(a2)
    fmt.Printf("二维数组的地址：%p\n", &a2)
    fmt.Printf("二维数组的长度：%d\n", len(a2))

    fmt.Printf("一维数组的长度：%d\n", len(a2[0]))
    fmt.Println(a2[0][3]) // 4
    fmt.Println(a2[1][2]) //7
    fmt.Println(a2[2][1]) // 10

    //遍历二维数组
    for i:=0;i<len(a2);i++{
        for j:=0;j<len(a2[i]);j++{
            fmt.Print(a2[i][j],"\t")
        }
        fmt.Println()
    }

    fmt.Println("----------------------")
    //for range 遍历二维数组
    for _,arr := range a2{
        for _,val := range arr{
            fmt.Print(val,"\t")
        }
        fmt.Println()
    }
```

```
46    }
```

冒泡排序

冒泡排序 Go 复制代码

```go
package main

import "fmt"

func main() {
    /*
数组的排序：
让数组中的元素具有一定的顺序。

arr :=[5]int{15,23,8,10,7}
升序：[7,8,10,15,23]
将序：[23,15,10,8,7]

排序算法：
冒泡排序，插入排序，选择排序，希尔排序，堆排序，快速排序。。。。

冒泡排序：（Bubble Sort）
依次比较两个相邻的元素，如果他们的顺序（如从大到小）就把他们交换过来。
*/

// 冒泡排序  并了解 函数的封装
    arr := [5]int{0, 5, 2, 7, 1}
    sort(arr, "desc")

}

// 将一个函数封装成一个功能 进行调用
func sort(arr [5]int, c string) {
    for j := 1; j < len(arr); j++ { //最后一个元素不用比较所以从1开始 j代表轮数
        for i := 0; i < len(arr)-j; i++ {//每一轮比较的次数 -j减去的是之前已经比好了的
            if c == "asc" {
                if arr[i] > arr[i+1] {
                    arr[i], arr[i+1] = arr[i+1], arr[i]
                }
            }
            if c == "desc" {
                if arr[i] < arr[i+1] {
                    arr[i], arr[i+1] = arr[i+1], arr[i]
                }
            }
        }
        fmt.Println(arr)
    }
}
```

```
45
```

切片

定义和初始化

定义和初始化 Go 复制代码

```go
//定义
var slice1 []type = make([]type, len)
也可以简写为
slice1 := make([]type, len)
make([]T, length, capacity)

//初始化  括号内有数字 和三点就是数组
s :=[] int {1,2,3 }
s := arr[startIndex:endIndex]

//将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片（前闭后开），长度为endIndex-startIndex
s := arr[startIndex:]
//缺省endIndex时将表示一直到arr的最后一个元素
s := arr[:endIndex]

//缺省取的数是左闭右开
  a := [5]int{76, 77, 78, 79, 80}
    var b []int = a[1:4] //通过数组创建了一个切片
//creates a slice from a[1] to a[3]

    //使用make定义有容量的切片    切片是根据长度访问数据的 超过了定义的长度就会报错！！
    s3 := make([]int, 5, 10)
    //panic: runtime error: index out of range [6] with length 5
    s3[6] = 2
    fmt.Println(s3[6])
```

切片是引用类型传递

切片是引用类型传递

Go | 复制代码

```go
package main

import (
    "fmt"
)

func main() {
    arr := [10]int{1, 2, 2, 3, 4, 2, 2, 2, 3, 4}
    fmt.Println("==============通过数组创建切片==============")
    s1 := arr[0:5]
    s2 := arr[3:8]
    s3 := arr[5:10]
    s4 := arr[:]
    s5 := arr[:8]
    fmt.Println(s1)
    fmt.Println(s2)
    fmt.Println(s3)
    fmt.Println(s4)

    fmt.Printf("%p\n", s1)  //切片的本身就是一个指针
    fmt.Printf("%p\n", &arr) //数组需要通过取地址符号才可以取到地址

    //切片数组的长度是元素的个数，容量是第几个元素开始到数组末尾的大小
    fmt.Printf("长度: %d 容量 : %d\n", len(s1), cap(s1)) //长度: 5 容量 : 10
    fmt.Printf("长度: %d 容量 : %d\n", len(s2), cap(s2)) //长度: 5 容量 : 7


    //切片和数组指向的地址是同一个  无论哪一个的值改变另一个对应位置的值也会改变
    arr[0] = 100
    fmt.Println(arr)
    fmt.Println(s1)
    s1[1] = 200
    fmt.Println(arr)
    fmt.Println(s1)

    //当一个切片追加元素且超过之前所规定的容量时，他的地址就会改变
    s1 = append(s1, 3, 2, 2, 2, 2, 2, 22, 2, 2, 2, 2)
    fmt.Println(s1)
    fmt.Println(arr)
    s1[3] = 400
    fmt.Println(s1)  //[100 200 2 400 4 3 2 2 2 2 2 22 2 2 2 2]
    fmt.Println(arr) //[100 200 2 3 4 2 2 2 3 4]
}
```

数组是值传递: 当一个变量赋值给另一个变量时，用的是两个地址，所以改变一个变量里面的值不会对另外一个数组里面的值产生影响。（可以理解成拷贝）

切片是引用类型传递: 当一个变量赋值给另一个变量时(同上，一个赋值给两个也受用)，用的是同一个地址，所以改变一个变量里面的值，另外赋值的变量里面的值也会更着改变！

切片的扩容

append函数会改变slice所引用的数组的内容，从而影响到引用同一数组的其它slice。 但当slice中没有剩 余空间（即(cap-len) == 0）时，此时将动态分配新的数组空间。返回的slice数组指针将指向这个空间，而原 数组的内容将保持不变；其它引用此数组的slice则不受影响

append() 和 copy() 函数

Go | 复制代码

```go

package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}

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

```

如果想在一个切片中添加另外一个切片

Go | 复制代码

```go
    s4 := make([]int, 0, 5)
    s5 := []int{1, 1, 1, 1}
    s4 = append(s4, s5...)  //使用...可以将切片中的数据解出来
    fmt.Println(s4)
```

切片的扩容分析

当在切片里面追加数据，切片的长度就是数据的个数，切片的容量会看你添加的数据个数加上原来的数据个数，如果超过了原来切片定义的容量，那么就会在原来切片容量的基础上*2。没有超出容量就是使用的同一个内存地址，超出了就是产生了一个新的内存地址。

Go 复制代码

```go
package main

import (
    "fmt"
)

func main() {

    s6 := []int{4, 6, 3}
    fmt.Println(s6)
    fmt.Printf("len:%d,cap:%d\n", len(s6), cap(s6))
    fmt.Printf("%p\n", s6) //打印内存地址

    s6 = append(s6, 2, 3)
    fmt.Println(s6)
    //len:5,cap:6  长度加对应个数  容量乘2
    fmt.Printf("len:%d,cap:%d\n", len(s6), cap(s6))
    fmt.Printf("%p\n", s6) //打印内存地址
    fmt.Println("============底层分析============")
    //底层分析
    numbers := []int{2, 4, 6, 7}
    fmt.Println(numbers)

    //创建切片
    numbers1 := make([]int, len(numbers), (cap(numbers))*2)

    //将numbers中的数拷贝到numbers1
    copy(numbers1, numbers)

    fmt.Printf("len: %d cap: %d slice%v   ", len(numbers1), cap(numbers1
), numbers1)
}
```

深浅拷贝

```go
package main

import "fmt"

func main() {
    //实现切片深拷贝    深拷贝：类似于值类型    浅拷贝：类似于引用类型
    arr1 := [4]int{1, 2, 3, 4}
    arr2 := make([]int, 0, 0)
    fmt.Println(arr1)
    fmt.Println(arr2)

    for i := 0; i < len(arr1); i++ {
        arr2 = append(arr2, arr1[i])
    }
    fmt.Println(arr1)
    fmt.Println(arr2)

    arr1[0] = 100
    fmt.Println(arr1) //[100 2 3 4]
    fmt.Println(arr2) //[1 2 3 4]

    //copy 简易实现切片的深拷贝    切片没有数组大小
    arr3 := []int{2, 3, 4}
    copy(arr2, arr3)
    //将arr3赋值给arr2后   由于arr3只有三个数 所以也就只改变arr2中前三个数
    fmt.Println(arr2) //[2 3 4 4]
    fmt.Println(arr3) //[2 3 4]
}
```

集合(Map）[key,value]

Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的，也是引用类型

使用map过程中需要注意的几点：

- map是无序的，每次打印出来的map都会不一样，它不能通过index获取，而必须通过key获取
- map的长度是不固定的，也就是和slice一样，也是一种**引用类型**
- 内置的len函数同样适用于map，**返回map拥有的key的数量**

- map的key可以是所有可比较的类型，如布尔型、整数型、浮点型、复杂型、字符串型……也可以键。

初始化map

Go | 复制代码

```go
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type
var map1 map[int]string            //定义但是没有初始化  是nil 未创建对象
map1 = make(map[int]string)  //这样就创建了对象  后面才可以进行赋值

/* 使用 make 函数 */
map_variable = make(map[key_data_type]value_data_type)
var map2 = make(map[string]string) //map[]  创建对象 输出的是空字符 map[]

//创建并赋值
map3 := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

//如果不初始化 map，那么就会创建一个 nil map。
对应上面的第一种只声明不初始化
//nil map 不能用来存放键值对
```

delete(map, key) 函数用于删除集合的元素, 参数为 map 和其对应的 key。删除函数不返回任何值。

问题 为什么range不用接收i

delete() 函数 Go 复制代码

```go

package main

import "fmt"

func main() {
   /* 创建 map */
   countryCapitalMap := map[string] string {"France":"Paris","Italy":"Rome","Japan":"Tokyo","India":"New Delhi"}

   fmt.Println("原始 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
    //这个for range 中 s 打印的是对应country的值
    for _,s := range countryCapitalMap {
        fmt.Println(s)
    }

   /* 删除元素 */
   delete(countryCapitalMap,"France");
   fmt.Println("Entry for France is deleted")

   fmt.Println("删除元素后 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
}

```

map的使用

Map的使用 Go 复制代码

```go
package main

import (
    "fmt"
)

func main() {
    var map1 map[int]string
    map1 = map[int]string{1: "xiaowang", 2: "xiaohuang"} //赋值方式1
    map1[4] = "red"                                      //赋值方式2
    fmt.Println(map1[2])
    fmt.Println(map1[1])
    fmt.Println(map1[3]) //Key不存在就获取默认的值 ""

    //逗号前面的是值，后面的是boolean类型的true/false
    //这个叫做 ok-idiom 它可以用来判断key value是否存在
    value, ok := map1[5]
    if ok {
        fmt.Println("Key存在", value)
    } else {
        fmt.Println("key不存在") //key不存在
    }

    //修改数据
    map1[4] = "white"
    fmt.Println(map1[4])
    fmt.Println(map1)

    //删除数据 delete
    delete(map1, 4)
    fmt.Println(map1)

    //如果key存在就是修改，不存在就是新增
    map1[10] = "aaaa"
    fmt.Println(len(map1))

}

```

指针

*type –> 是一个地址　a = *type　*a –> 就是那个地址所存储的数值

指针是存储另一个变量的内存地址的变量。

我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。

一个指针变量可以指向任何一个值的内存地址它指向那个值的内存地址。

Go | 复制代码

```go
package main

import "fmt"

func main() {
   var a int= 20   /* 声明实际变量 */
   var ip *int        /* 声明指针变量 */

   ip = &a  /* 指针变量的存储地址 */

   fmt.Printf("a 变量的地址是: %x\n", &a  )

   /* 指针变量的存储地址 */
   fmt.Printf("ip 变量的存储地址: %x\n", ip )

   /* 使用指针访问值 */
   fmt.Printf("*ip 变量的值: %d\n", *ip )
}
```

指针的套娃使用

指针套娃使用 Go 复制代码

```go
package main

import "fmt"

func main() {
    //声明一个变量
    var a int = 10
    fmt.Println("a变量的值：", a)
    fmt.Println("a变量的地址：", &a)

    //声明一个指针变量，*int

    var p *int
    p = &a //指针变量存储的地址
    fmt.Printf("p变量存储的指针地址 :%p\n", p)
    fmt.Printf("p变量的地址 :%p\n", &p)
    fmt.Println("*p变量的地址 ", *p)

    //再定义一个指针ptr，来指向指针变量p  指针的指针 也是一个地址
    var ptr **int
    ptr = &p
    fmt.Printf("ptr存储的指针的地址：%p\n", ptr)  //p的地址
    fmt.Printf("ptr 变量自己的地址：%p\n", &ptr) //自己的地址
    fmt.Printf("*ptr 变量的值：%p\n", *ptr)   //p地址的值 他是一个地址
    fmt.Printf("*ptr变量地址的值：%d\n", **ptr) //p的地址里面的值

}
```

数组指针

数组指针：首先是一个指针，一个数组地址

指针数组：首先是一个数组，存储的数据类型是指针。数组里面存的是指针

Go | 复制代码

```go
package main

import "fmt"

func main() {
    //数组指针
    var arr1 = [4]int{1, 2, 3, 4}
    var p1 *[4]int
    p1 = &arr1

    fmt.Printf("p1 的地址: %p\n", p1)
    fmt.Printf("p1 自己的地址: %p\n", &p1)
    fmt.Printf("p1 所指向的地址的值: %p\n", *p1)

    (*p1)[0] = 100
    fmt.Println(arr1)
    fmt.Println(*p1)

    //简化写法   因为p1是一个指针又是一个数组就可以这么写
    p1[0] = 200
    fmt.Println(arr1)

    //指针数组
    a := 1
    b := 2
    c := 3
    d := 4

    arr2 := [4]*int{&a, &b, &c, &d}

    fmt.Println(arr2)

    *arr2[0] = 500 //这里取出的是数组中下标是0的值
    fmt.Println(a)

    a = 200
    fmt.Println(*arr2[0])
}
```

指针函数 & 指针作为参数

指针函数：返回的是一个指针

```go
package main

import "fmt"

func main() {
    ptr := f1()
    fmt.Printf("ptr类型：%T\n", ptr)
    fmt.Println(&ptr) //自己的地址
    fmt.Println(*ptr) //指向的数值

    //可以通过指针去修改一个变量的值又不会开辟一块新的内存空间
    a := 10
    fmt.Println("a=", a)
    f2(&a)
    fmt.Println("a调用函数之后的值：", a)

}

// 指针函数
func f1() *[4]int {
    arr := [4]int{1, 2, 3, 4}
    return &arr
}

//指针作为参数
func f2(ptr *int) {
    fmt.Println("ptr", ptr)
    fmt.Println("*ptr", *ptr)
    *ptr = 100

}
```

结构体

结构体的定义和使用

结构体的定义： Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。 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。

结构体初始化 Go 复制代码

```go
// 1.按照顺序提供初始化值
P := person{"Tom", 25}
// 2.通过field:value的方式初始化，这样可以任意顺序
P := person{age:24, name:"Tom"}
// 3.new方式,未设置初始值的，会赋予类型的默认初始值
p := new(person)
p.age=24
```

四种初始化结构体的方式 Go 复制代码

```go
var user1 User

user2 := User{}

user3 := User{
         name: "小兰",
         age:  18,
     }

user4 := User{"小绿", 18, "女"}
```

结构体指针应用

（new和make的区别？）：

new(T)**函数是一个分配内存的内建函数。（在结构体中出现的）**

我们都知道，对于一个已经存在变量，可对其指针进行赋值。

make**不仅可以开辟一个内存，还能给这个内存的类型初始化其零值。**

make**和**new**都是**golang**用来分配内存的内建函数，且在堆上分配内存，**make **即分配内存，也初始化内存。**new**只是将内存清零，并没有初始化内存。**

make**返回的还是引用类型本身；而**new**返回的是指向类型的指针。**

make**只能用来分配及初始化类型为**slice，map，channel**的数据；**new**可以分配任意类型的数据。**

结构体是值类型

如果使用内置函数new()创建,new的所有Type都返回指针

指针应用

Go | 复制代码

```go
package main

import "fmt"

type User1 struct {
    name string
    age  int
    sex  string
}

func main() {
    //结构体是值类型，值传递
    user1 := User1{"qinjiang", 18, "nan"}
    fmt.Println(user1)
    user2 := user1
    user2.name = "kuangshen"
    fmt.Println(user1)
    fmt.Println(user2)

    //指针
    var user3 *User1
    user3 = &user1
    (*user3).name = "feige"
    fmt.Println(user1)

    //new 方法 跟指针的效果一样
    user4 := new(User1)
    user4.name = "xuexiangban"
    fmt.Printf("%T\n", user4) // 以指针的形式
    fmt.Println(user1)

}
```

匿名结构体

Go | 复制代码

```go
package main

import (
    "fmt"
)

type Student struct {
    name string
    age  int
}

// 匿名结构体
type Teacher struct {
    string
    int
}

func main() {
    f1 := Student{name: "xioawang", age: 14}
    fmt.Println(f1)

    //匿名结构体  和匿名字段  都是用于嵌套
    f2 := struct {
        name string
        age  int
    }{"qinjiang", 18}

    fmt.Println(f2)

    //也可以不用定义字段的名字  不过不建议这样做
    f3 := new(Teacher)
    t1 := Teacher{"小王", 27}
    //匿名字段  默认使用字段数据类型当字段名称
    fmt.Println(t1.string)
    f3.string = "xiaowang"
    fmt.Println(f3)
}
```

结构体嵌套

嵌套 Go 复制代码

```go
package main

import (
    "fmt"
)

type Person struct {
    name    string
    age     int
    //结构体嵌套    在一个结构体中嵌套另一个结构体 作为他的字段
    address Address
}

//结构体的嵌套   在一个结构体中可以嵌套另外一个结构体
type Address struct {
    city   string
    status string
}

func main() {

    var person = Person{}
    person.name = "qinjiang"
    person.address = Address{
        city:   "guanngzhaou",
        status: "zhongguo",
    }

    fmt.Println(person.name)
    fmt.Println(person.address)
}
```

导出结构体和字段

如果结构体类型以大写字母开头(public)，那么它是一个导出类型，可以从其他包访问它。类似地，如果结构体的字段以大写开头，则可以从其他包访问它们。

Go的面向对象思想

封装：就是结构体+方法 定义一个特定的功能和类别

继承：继承就是在一个类中使用匿名字段

多态：一个事物拥有多种形态。例如：猫狗他们既是自己本身又是动物，这就是多态。

继承

继承：继承就是在一个类中使用匿名字段 ， 该字段也是一个结构体，这就是继承。

面向对象-继承

Go | 复制代码

```go
package main

import "fmt"

// 定义一个父"类"结构体
type person struct {
    name string
    age  int
}

// 定义一个子"类"结构体
type Student struct {
    person        //匿名变量，继承的作用
    school string //子类自己的属性字段
}

func main() {
    p2 := person{"小黄", 19}

    fmt.Println(p2.name)

    p1 := Student{
        person: person{"小王", 18},
        school: "清华",
    }

    //这是通过子类的对象调用父类再点名字。
    fmt.Println(p1.person.name)
    //模拟聚合
    /*type C struct {
        age1 int

    }

    type D struct {
        c C
    }*/
    //这时d就不能直接访问c中的属性  需要d.c  .xx访问

    //这里用的提升字段 （模拟继承）
    //对于student来说person是匿名字段 person中的name、age为提升字段
    //提升字段可以通过名字直接访问，不需要再用结构体
    fmt.Println(p1.name)
}
```

方法和方法重写

方法：在函数中加入指定类型的话就是方法 方法只能通过指定对象的进行调用

方法重写：子类可以重写父类的方法，子类可以新增自己的属性和方法，子类可以直接访问父类的属性和方法

Go | 复制代码

```go
package main

import "fmt"

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

// 父类的eat方法
func (animal Animal) eat() {
    fmt.Println(animal.name + "正在吃")
}

// 子类 狗
type Dog2 struct {
    Animal
}

// 猫重写  动物的eat方法
func (cat Cat2) eat() {  //子类可以重写父类的方法
    fmt.Println(cat.name + "正在吃")
}

// 子类 猫

type Cat2 struct {
    Animal
    color string  //子类可以新增自己的属性和方法
}

func main() {

    dog := Dog2{Animal{name: "旺财", age: 2}}
    dog.eat()   //子类可以直接访问父类的属性和方法

    cat := Cat2{Animal{
        name: "小黄",
        age:  2,
    }, "red"}
    cat.eat()
    fmt.Println(cat.color)
}

```

接口实现

接口：它把所有的具有共性的方法定义在一起，任何其他类型只要实现接口定义的全部方法就是实现了这个接口

注意：接口只做定义，不做具体的方法实现，具体实现交给实现方法。

接口

Go | 复制代码

```go
package main

import "fmt"

type USB interface { //定义一个接口

    input()
    output()
}
type Mouse struct {
    //接口的实现类    只要这个类将接口中的所有方法都实现就叫接口的实现类
    name string
}

func (mouse Mouse) input() { //用类实现接口的中函数（称为方法）
    fmt.Println(mouse.name + "鼠标输入")
}
func (mouse Mouse) output() {
    fmt.Println(mouse.name + "鼠标输出")
}

// test函数传入的参数是一个接口（这个可以用来干嘛）
func test(u USB) {
    u.input()
    u.output()
}

func main() {
    //通过传入接口的实现类，来进行具体方法的调用
    m := Mouse{"逻辑"}
    test(m) //需要结构体完整的实现接口，才可以进行（传入对象的）方法的调用

    //定义接口 将实现类赋值给接口
    var usb USB
    //接口对象可以接受实现类的赋值，但是不能访问实现类中的属性！！！
    usb = m
    //usb 不能用mouse的字段
    //fmt.Println(usb.name)
    fmt.Println(usb)

}
```

多态

Go语言通过接口来模拟多态。

多态：一个事物拥有多种形态。例如：猫狗他们既是自己本身又是动物，这就是多态。

只要一个结构体实现了接口(父类)中的所有方法，然后再定义一个函数传入父类（动物）接口，用该父类接口调用实现类实现的方法，这就是多态。

多态

Go | 复制代码

```go
package main

import (
    "fmt"
)

type Animal2 interface {
    eat()
    sleep()
}
type Dog4 struct {
    name string
}

func (dog Dog4) eat() {
    fmt.Println(dog.name + "eat")
}
func (dog Dog4) sleep() {
    fmt.Println(dog.name + "sleep")
}

func test1(a Animal2) {
    fmt.Println("test1")
    a.eat() //当调用这个方法时，传入的对象是实现类的对象，那么eat方法中a就是那个对象
}

func main() {
    dog := Dog4{"大黄"}
    dog.sleep()
    dog.eat()

    //当一个实现类实现了接口中的所有方法
    //那么所有传入接口的地方那也就可以传入实现类
    test1(dog)

    //定义一个类型为接口的变量
    //实际上可以赋值为任意实现类的对象
    var animal Animal2
    animal = dog
    fmt.Println(animal)
    //这时候只能使用animal的方法，不能使用dog的属性
    animal.sleep()
    animal.eat()
}

```

空接口

不包含任何的方法，正因为如此，所有的类型都实现了空接口，因此空接口可以存储任意类型。(类比前面接口的实现中，定义接口，再将实现类赋值给接口)

空接口

Go | 复制代码

```go
package main

import "fmt"

type dog struct {
    name string
    age  int
}

type cat struct {
    name  string
    color string
}

// A 是空接口
type A interface {
}

// 因为接口A是空接口 ， 所以函数可以接收任意类型当参数
func test5(a A) {
    fmt.Println(a)
}

// fmt 下的输出接收的参数都是这样的
func test6(a interface{}) {
    fmt.Println(a)
}

func main() {
    var a1 A = dog{"小狗", 2}
    var a2 A = cat{
        name:  "小猫",
        color: "黑色",
    }

    fmt.Println(a1)
    fmt.Println(a2)
    test5(a1)
    test5(a2)

    //map  key string  v  obj
    map1 := make(map[string]interface{})
    map1["name"] = "大黄"
    map1["age"] = 14
    map1["dog"] = dog{name: "小黄", age: 12}
```

```
46
47      //切片
48      s1 := make([]interface{}, 0, 10)
49      s1 = append(s1, a2, a2)
50      fmt.Println(s1)
51
52  }
53
```

接口的继承

当一个接口继承了另外的接口，前接口的实现类就必须实现他自己的方法，和继承的接口的方法。

将前一个接口的实现类赋值给他继承的接口时，该接口的对象只能调用他自己的方法。(在接口嵌套中，嵌套的接口默认继承了被嵌套接口的所有方法)

接口的嵌套

Go | 复制代码

```go
package main

import "fmt"

type B interface {
    test2()
}

type C interface {
    test3()
}

type D interface {
    B
    C
    test4()
}

type Dog5 struct {
}

func (dog Dog5) test2() {
    fmt.Println("test2")
}
func (dog Dog5) test3() {
    fmt.Println("test3")
}

func (dog Dog5) test4() {
    fmt.Println("test4")
}

func main() {
    dog := Dog5{}
    dog.test2()
    dog.test3()
    dog.test4()

    //再接口嵌套中，嵌套的接口默认继承了被嵌套接口的所有方法
    //将dog赋值给接口 该接口只能调用自己的方法
    var b1 B = dog
    b1.test2()

    //由于d是继承了 B和C 所以他还是可以调用里面的方法
    var d1 D = dog
```

```
    d1.test4()
    d1.test3()
    d1.test2()
}
```

接口断言

判断一个东西的类型，用 i.（type）的方式。

检查接口类型变量的值是否实现了期望的接口，就是检查当前接口类型的值有没有实现指定的接口

当定义一个接口下面有多个实现类，这时我们要判断这个类在哪个地方用的时候，我们就需要使用断言判断它是不是你定义的对象

断言

Go | 复制代码

```go
package main

import (
    "fmt"
)

func assertString(i interface{}) {
    t2, ok := i.(string)  //可以返回true false
    if ok {
        fmt.Println("i变量是string类型")
        fmt.Println(t2)
    } else {
        fmt.Println("i变量不是是string类型")
    }
}

func assertInt(i interface{}) {
    t1 := i.(int)  //直接判断  是就是 不是就报错
    fmt.Println(t1)
}

type G interface { //自己定义的类型

}

// 假设断言的类型同时实现了switch断言的多个case，取第一个case
func test22(f interface{}) {
    switch f.(type) {
    case int:
        fmt.Println("int类型")
    case string:
        fmt.Println("string 类型")
    case G:
        fmt.Println("G类型") //空接口的判断应该放最后一个  与上面说的对应
    case nil:
        fmt.Println("nil类型")
    default:
        fmt.Println("未知类型")
    }
}

func main() {

    var g G
    test22(g) //定义了接口但是没有赋值   就是nil
```

```
    test22("ss")
    test22(22)
    test22(true)

    assertInt(2)

    assertString("aaaa")
    assertString(22)
}
```

Type别名

别名 Go 复制代码

```go
package main

import "fmt"
// 通过type关键字的定义，diyInt就是一种新的类型，具有int的特性
type diyInt int

func main() {
    var a diyInt = 18
    var b = 18
    var c = int(a) + b //因为类型改变了  使用的话需要转型
    fmt.Println(c)

    //将int赋值给myInt  就可以直接相加
    //myInt只在代码中存在，编译完成时并不会有myInt类型
    type myInt = int
    var d myInt = 19
    var e int = 10
    f := d + e
    fmt.Println(f)
}
```

错误

错误是什么?

错误指的是可能出现问题的地方出现了问题。比如打开一个文件时失败，这种情况在人们的意料之中 。

而异常指的是不应该出现问题的地方出现了问题。比如引用了空指针，这种情况在人们的意料之外。可见，错误是业务过程的一部分，而异常不是 。

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。

创建自己的错误信息

错误创建 Go 复制代码

```go
package main
import (
    "errors"
    "fmt"
)
func main() {
    //1、直接errors方法点new创建错误
    errinfo := errors.New("我是一个错误")
    fmt.Println(errinfo)
    fmt.Printf("%T\n", errinfo)

    //2、利用函数返回error类型创建错误
    err := setAge(-1)
    if err != nil {
        fmt.Println(err)
    }
    //3、利用fmt包下的Errorf方法创建错误
    errinfo2 := fmt.Errorf("错误信息%d", 200)
    fmt.Println(errinfo2)
    fmt.Printf("%T\n", errinfo2)
}
func setAge(age int) error {
    if age < 0 {
        return errors.New("输入的年龄不合法")
    }
    fmt.Println(age)
    return nil
}
```

自定义error

Go | 复制代码

```go
package main

import "fmt"

// 定义一个自己的错误
type myDiyError struct {
    code int
    msg  string
}

// 实现错误类的方法
func (e myDiyError) Error() string {
    return fmt.Sprint("错误信息: ", e.msg, "状态码: ", e.code)
}

// 模拟一个错误  返回 类的数值和错误
func test(i int) (int, error) {
    if i != 0 {
        return i, &myDiyError{ //&得到错误的对象
            code: 500,
            msg:  "非0数据",
        }
    }
    //如果输入的数不是0
    return 0, nil
}

func main() {
    i, err := test(1)
    if err != nil {
        fmt.Println(err)
        //断言 看还不是返回的子集设计的错误
        myerr, ok := err.(*myDiyError)//取错误里面的数值
        //如果是  输出信息
        if ok {
            fmt.Println(myerr.msg)
            fmt.Println(myerr.code)
        }
    }
    //返回输入的数值
    fmt.Println(i)

}
```

panic

如果函数中书写并触发了panic，会终止其后要执行的代码，在panic所在函数内如果存在要执行deffer函数列表，会按照deffer书写顺序的逆序执行

recover

recover的作用就是捕获panic，从而恢复panic后面代码的执行；

注意：panic所在行的下面的代码依然不会执行,恢复的是主程序中panic函数的后面的代码

recover必须配合panic使用;

recover没有传递的参数，但是有返回值，其值就是panic传递的值。

Go | 复制代码

```go
package main

import "fmt"

func main() {
    defer fmt.Println("main ======= 2")
    defer fmt.Println("main========3")
    fmt.Println("main" + "=====1")
    test1(1)
    fmt.Println("main======4")
}
func test1(num int) {

    //当有recover时，程序之前因为panic的异常会恢复恐慌  就会让外部函数继续执行
    //注意！！！   panic所在行的下面的代码依然不会执行
    defer func() {
        msg := recover()
        if msg != nil {
            fmt.Println("msg:"+"dsa", msg, "程序恢复执行")
        }
    }()

    //当一个程序中有panic时，会逆序执行deffer后面的程序
    //然后再返回到main程序中 ，再逆序执行main程序中的deffer后面的程序
    //最后抛出异常
    //此时不会执行  main=====4
    //但是会逆序执行main defer后面的程序 复述第二行
    defer fmt.Println("test========1")
    defer fmt.Println("test========2")
    fmt.Println("test =========3 ")
    if num == 1 {
        panic("程序异常了")
    }
    fmt.Println("test =======4 ")
}
```