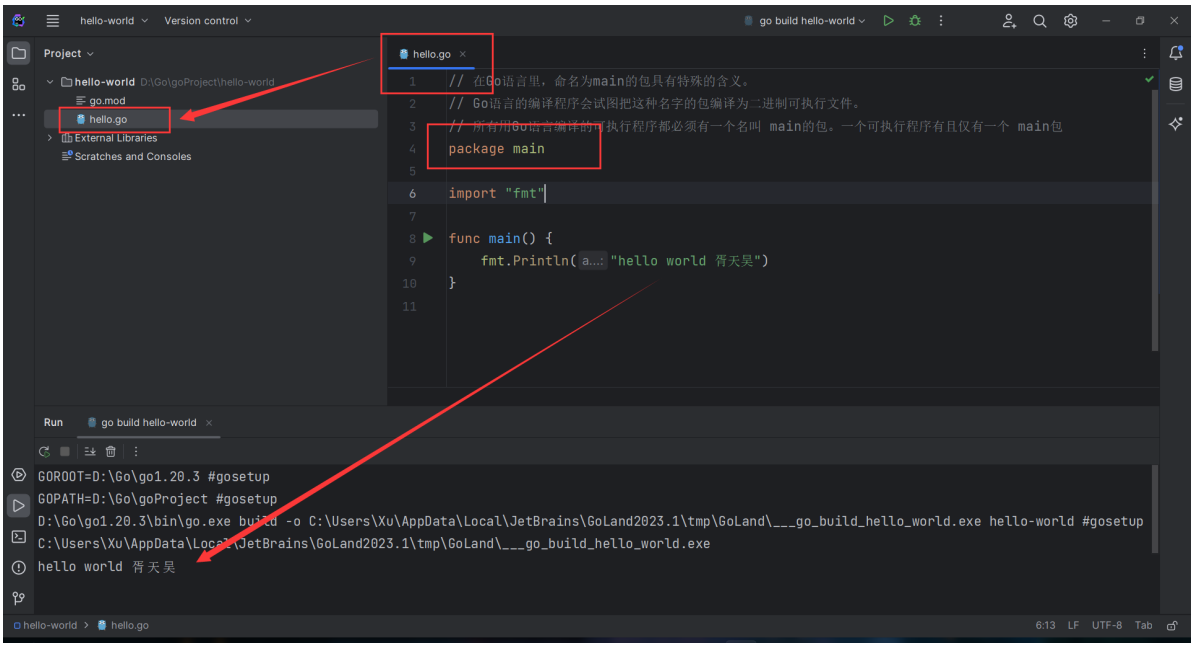
Go语言的整个设计哲学就是：将简单、实用体现的淋漓尽致

基础语法学习

HelloWorld!



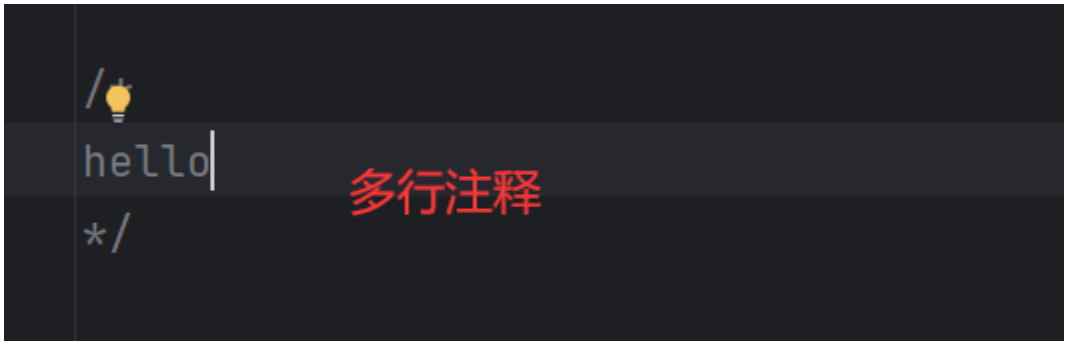

```go
// 在Go语言里，命名为main的包具有特殊的含义。
// Go语言的编译程序会试图把这种名字的包编译为二进制可执行文件。
// 所有用Go语言编译的可执行程序都必须有一个名叫 main的包。一个可执行程序有且仅有一个 main包
package main

import "fmt"

func main() {
    fmt.Println("hello world 胥天昊")
}
```

注释

注释跟Java的一样

```go
// 在Go语言里，命名为main的包具有特殊的含义。
// Go语言的编译程序会试图把这种名字的包编译为二进制可执行文件。
// 所有用Go语言编译的可执行程序都必须有一个名叫 main的包。一个可执行程序有且仅有一个 main包
```

单行注释



变量

```
// 在Go语言里，命名为main的包具有特殊的含义。
// Go语言的编译程序会试图把这种名字的包编译为二进制可执行文件。
// 所有用Go语言编译的可执行程序都必须有一个名叫 main的包。一个可执行程序有且仅有一个 main包
package main

import "fmt"

func main() {
    //声明变量用var
    //name 是变量名
    //string/int 代表变量的类型
    var name string = "胥天昊"
    var age int = 18
    var (
        //如果没有显式赋值，默认有初始值
        //string        空字符串
        //int           0
        //bool          false
        //切片/函数/指针   nil
        sex  string = "男"
        addr string = "河南"
    )
    //甚至可以这样定义
    var a, b, c int = 1, 2, 3
    //还可以这样，编译器默认推导变量是什么类型
    //这个名字叫短变量声明并初始化
    n1 := 10
    n2 := "变量"

    println("hello " + name)
    println(age)
    println(sex, addr)
    println(a, b, c)
    //%T是个占位符，可以打印出来变量的类型
    fmt.Printf("%T,%T", n1, n2)
}
```

内存地址

```go
package main

import "fmt"

func main() {
    var num int = 100
    fmt.Printf("%d", num)
    println()
    //取地址符
    fmt.Printf("%p", &num)

}
```

变量交换

```go
package main

func main() {
    var a int = 100
    var b int = 200
    //这里就交换咯
    b, a = a, b
    println(a)
    println(b)
}
```

匿名变量

```go
package main

func test() (int, int) {
    return 100, 200
}

func main() {
    //a, b := test()
    //println(a)
    //println(b)
    //_叫空白标识符，给这个符号赋值的数据会被抛弃
    //这个也叫匿名变量
    //匿名变量不占用内存空间。不会分配内存。匿名变量与匿名变量之间也不会因为多次声明而无法使用。
    a, _ := test()
    println(a)
}
```

变量的作用域

一个变量（常量、类型或函数）在程序中都有一定的作用范围。称之为作用域。

了解变量的作用域对我们学习Go语言来说是比较重要的，因为Go语言会在编译时**检查每个变量是否使用过**，一旦出现未使用的变量，就会报编译错误。如果不能理解变量的作用域，就有可能会带来一些不明所以的编译错误。

```go
package main
//全局变量
var name string = "xth"

func main() {
    //局部变量
    var name string = "胥天昊"
    //优先使用局部变量
    println(name)//胥天昊
}
```

常量

不可以改变的数据

```go
package main

func main() {
    const URL  string= "www.xiaobaicai.com"
    const URL2 = "www.xiaobaicai.com"
    //这里不能修改
    URL="www"
}
```

iota

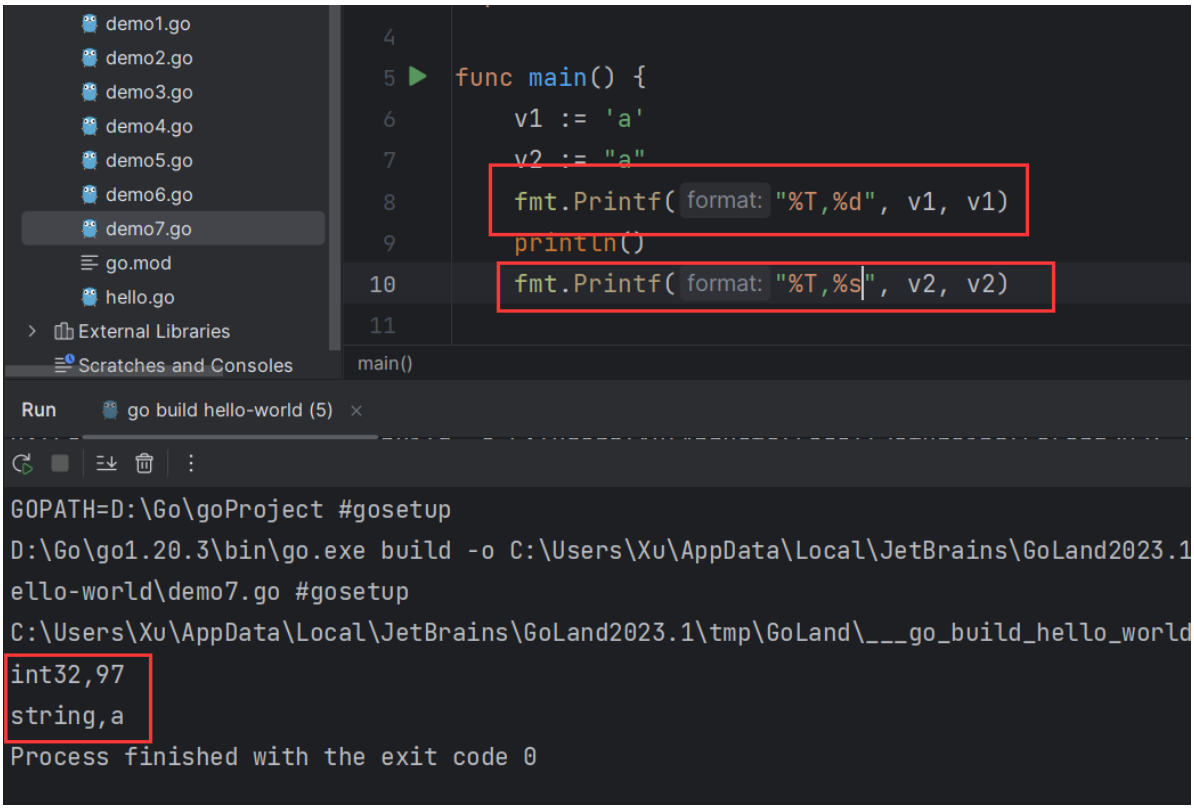
iota。特殊**常量**。可以认为是一个可以被编译器修改的常量。iota是go语言的常量计数器

iota在const关键字出现时将被重置为0(const内部的第一行之前),const中每新增一行常量声明将使iota计数一次(iota可理解为const语句块中的行索引)


```go
package main

func main() {
    const (
        a = iota
        b
        c
        d = "haha"
        e
        f = 100
        g
        h = iota
        i
    )
    const (
        j = iota
        k
    )
}
```

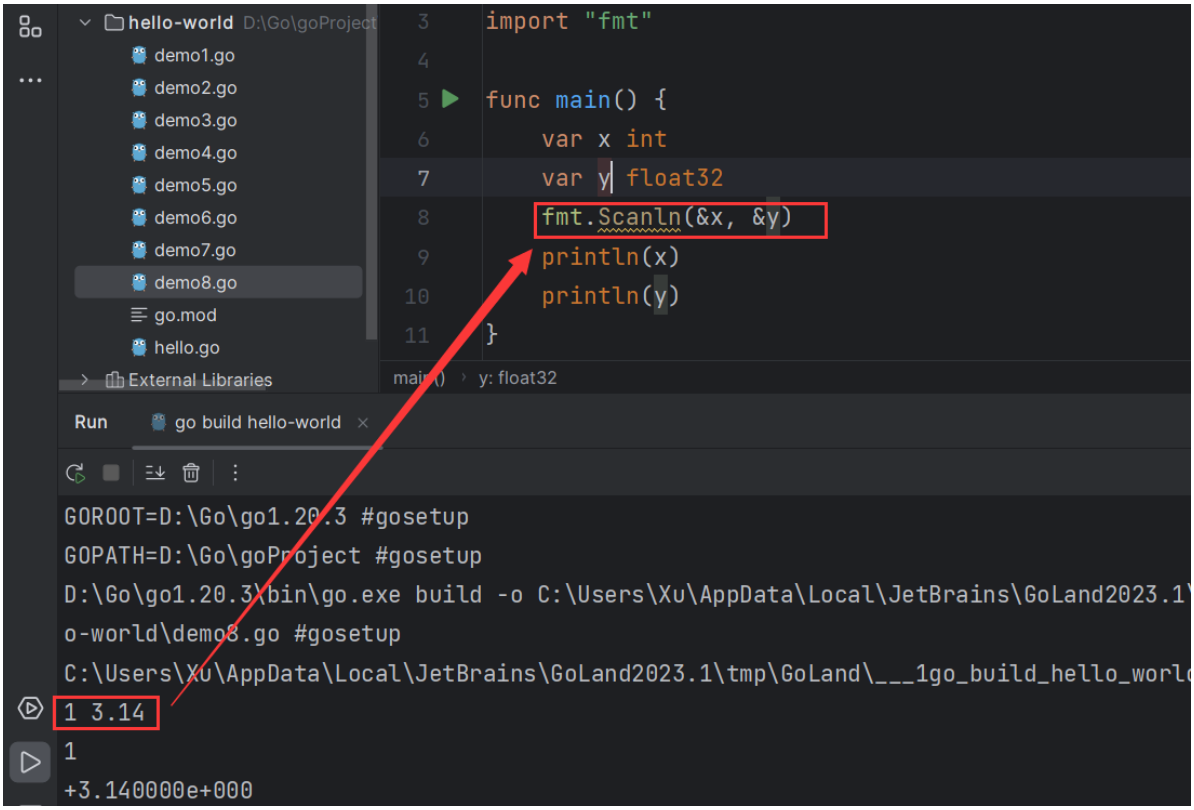
基本数据类型

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



输入

```go
package main

import "fmt"

func main() {
    var x int
    var y float32
    fmt.Scanln(&x, &y)
    println(x)
    println(y)
}
```



容器

数组

数组 是一个由 **固定长度** 的特定类型元素组成的序列，一个数组可以由零个或多个元素组成。因为数组的长度是固定的，因此在 Go 语言中*很少直接使用数组*。和数组对应的类型是 **slice(切片)**，它是可以动态的增长和收缩的序列，`slice` 功能也更灵活，下面我们再讨论 `slice`。

数组声明

可以使用 `[n]Type` 来声明一个数组。其中 `n` 表示数组中元素的数量，`Type` 表示每个元素的类型。

```
package main

import "fmt"

func test01() {
    // 声明时没有指定数组元素的值，默认为零值
    var arr [5]int
    fmt.Println(arr)

    arr[0] = 1
    arr[1] = 2
    arr[2] = 3
    fmt.Println(arr)
}

func test02() {
    // 直接在声明时对数组进行初始化
    var arr1 = [5]int{15, 20, 25, 30, 35}
    fmt.Println(arr1)
```

```
    // 使用短声明
    arr2 := [5]int{15, 20, 25, 30, 35}
    fmt.Println(arr2)

    // 部分初始化，未初始化的为零值
    arr3 := [5]int{15, 20} // [15 20 0 0 0]
    fmt.Println(arr3)

    // 可以通过指定索引，方便地对数组某几个元素赋值
    arr4 := [5]int{1: 100, 4: 200}
    fmt.Println(arr4) // [0 100 0 0 200]

    // 直接使用 ... 让编译器为我们计算该数组的长度
    arr5 := [...]int{15, 20, 25, 30, 35, 40}
    fmt.Println(arr5)
}

func test03() {
    // 特别注意数组的长度是类型的一部分，所以 [3]int 和 [5]int 是不同的类型
    arr1 := [3]int{15, 20, 25}
    arr2 := [5]int{15, 20, 25, 30, 35}
    fmt.Printf("type of arr1 is %T\n", arr1)
    fmt.Printf("type of arr2 is %T\n", arr2)
}

func test04() {
    // 定义多维数组
    arr := [3][2]string{
        {"1", "Go语言极简一本通"},
        {"2", "Go语言微服务架构核心22讲"},
        {"3", "从0到Go语言微服务架构师"}}
    fmt.Println(arr) // [[15 20] [25 22] [25 22]]
}

func main() {
    test01()
    test02()
    test03()
    test04()
}
```

数组长度

使用内置的 `len` 函数将返回数组中元素的个数，即数组的长度。

```
func arrLength() {
    arr := [...]string{"Go语言极简一本通", "Go语言微服务架构核心22讲", "从0到Go语言微服务架构师"}
    fmt.Println("数组的长度是:", len(arr)) //数组的长度是: 3
}
```

数组遍历

使用 `for range` 循环可以获取数组每个索引以及索引上对应的元素。

```
func showArr() {
    arr := [...]string{"Go语言极简一本通", "Go语言微服务架构核心22讲", "从0到Go语言微服务架构师"}
    for index, value := range arr {
        fmt.Printf("arr[%d]=%s\n", index, value)
    }

    for _, value := range arr {
        fmt.Printf("value=%s\n", value)
    }
}
```

数组是值类型

Go 中的数组是值类型而不是引用类型。当数组赋值给一个新的变量时，该变量会得到一个原始数组的一个副本。如果对新变量进行更改，不会影响原始数组。

```
func arrByValue() {
    arr := [...]string{"Go语言极简一本通", "Go语言微服务架构核心22讲", "从0到Go语言微服务架构师"}
    copy := arr
    copy[0] = "Golang"
    fmt.Println(arr)
    fmt.Println(copy)
}
```

切片(Slice)

切片是对数组的一个连续片段的引用，所以切片是一个**引用类型**。切片本身不拥有任何数据，它们只是**对现有数组的引用**，每个切片值都会将数组作为其底层的数据结构。slice 的语法和数组很像，只是没有固定长度而已。

创建切片

使用 `[]Type` 可以创建一个带有 `Type` 类型元素的切片。

```
// 声明数组！！！这个是不是跟切片很像
var arr [5]int
// 声明整型切片
var numList []int

// 声明一个空切片
var numListEmpty = []int{}
```

你也可以使用 `make` 函数构造一个切片，格式为 `make([]Type, size, cap)` 。

```
//第一个参数是int类型
//第二个参数是len表示切片的长度，每个元素初始化为对应类型的零值；
//第三个参数是cap表示切片的容量，默认等于len，且cap >= len
numList := make([]int, 3, 5)
```

当然，我们可以通过对数组进行片段截取创建一个切片。

```
arr := [5]string{"Go语言极简一本通", "Go语言微服务架构核心22讲", "从0到Go语言微服务架构师", "微服务", "分布式"}
var s1 = arr[1:4]
fmt.Println(arr) // [Go语言极简一本通 Go语言微服务架构核心22讲 从0到Go语言微服务架构师 微服务 分布式]
fmt.Println(s1)  // [Go语言微服务架构核心22讲 从0到Go语言微服务架构师 微服务]
```

切片的长度和容量

一个 slice 由三个部分构成：**指针** 、 **长度** 和 **容量** 。

指针指向第一个 slice 元素对应的底层数组元素的地址，要注意的是 slice 的第一个元素并不一定就是数组的第一个元素。

长度对应 slice 中元素的数目；长度不能超过容量，容量一般是从 slice 的开始位置到底层数据的结尾位置。

容量就是从创建切片索引开始的底层数组中的元素个数，而长度是切片中的元素个数。

内置的 `len` 和 `cap` 函数分别返回 slice 的长度和容量。

```
s := make([]string, 3, 5)
fmt.Println(len(s)) // 3
fmt.Println(cap(s)) // 5
```

如果切片操作超出上限将导致一个 `panic` 异常。

```
s := make([]int, 3, 5)
fmt.Println(s[10]) //panic: runtime error: index out of range [10] with length 3
```

Tips:

- 由于 slice 是引用类型，所以你不对它进行赋值的话，它的默认值是 `nil`

  ```
  var numList []int
  fmt.Println(numList == nil) // true
  ```

- 切片之间不能比较，因此我们不能使用 `==` 操作符来判断两个 slice 是否含有全部相等元素。特别注意，如果你需要测试一个 slice 是否是空的，使用 `len(s) == 0` 来判断，而不应该用 `s == nil` 来判断。

切片元素的修改

切片自己不拥有任何数据。它只是底层数组的一种表示。对切片所做的任何修改都会反映在底层数组中。

```
func modifySlice() {
    var arr = [...]string{"Go语言极简一本通", "Go语言微服务架构核心22讲", "从0到Go语言微服务架构师"}
    s := arr[:] //[0:len(arr)]
    fmt.Println(arr) //[Go语言极简一本通 Go语言微服务架构核心22讲 从0到Go语言微服务架构师]
    fmt.Println(s) //[Go语言极简一本通 Go语言微服务架构核心22讲 从0到Go语言微服务架构师]

    s[0] = "Go语言"
    fmt.Println(arr) //[Go语言 Go语言微服务架构核心22讲 从0到Go语言微服务架构师]
    fmt.Println(s) //[Go语言 Go语言微服务架构核心22讲 从0到Go语言微服务架构师]
}
```

这里的 `arr[:]` 没有填入起始值和结束值，默认就是 `0` 和 `len(arr)` 。

追加切片元素

使用 `append` 可以将新元素追加到切片上。`append` 函数的定义是 `func append(slice []Type, elems ...Type) []Type` 。其中 `elems ...Type` 在函数定义中表示该函数接受参数 `elems` 的个数是可变的。这些类型的函数被称为可变函数。

```
func appendSliceData() {
    s := []string{"Go语言极简一本通"}
    fmt.Println(s)
    fmt.Println(cap(s))

    s = append(s, "Go语言微服务架构核心22讲")
    fmt.Println(s)
    fmt.Println(cap(s))

    s = append(s, "从0到Go语言微服务架构师", "分布式")
    fmt.Println(s)
    fmt.Println(cap(s))

    s = append(s, []string{"微服务", "分布式锁"}...)
    fmt.Println(s)
    fmt.Println(cap(s))
}
```

当新的元素被添加到切片时，如果容量不足，会创建一个新的数组。现有数组的元素被复制到这个新数组中，并返回新的引用。现在新切片的容量是旧切片的两倍。

多维切片

类似于数组，切片也可以有多个维度。

```
func mSlice() {
    numList := [][]string{
        {"1", "Go语言极简一本通"},
        {"2", "Go语言微服务架构核心22讲"},
        {"3", "从0到Go语言微服务架构师"},
    }
    fmt.Println(numList)
}
```

Map

在 Go 语言中，map 是散列表(哈希表)的引用。它是一个拥有键值对元素的**无序集合**，在这个集合中，键是唯一的，可以通过键来获取、更新或移除操作。无论这个散列表有多大，这些操作基本上是通过常量时间完成的。所有可比较的类型，如 `整型`，`字符串` 等，都可以作为 `key` 。

创建 Map

使用 `make` 函数传入键和值的类型，可以创建 map 。具体语法为 `make(map[KeyType]ValueType)` 。

```
// 创建一个键类型为 string 值类型为 int 名为 scores 的 map
scores := make(map[string]int)
steps := make(map[string]string)
```

我们也可以用 map 字面值的语法创建 map ， 同时还可以指定一些最初的 key/value ：

```
var steps2 map[string]string = map[string]string{
        "第一步": "Go语言极简一本通",
        "第二步": "Go语言微服务架构师核心22讲",
        "第三步": "从0到Go语言微服务架构师",
}
fmt.Println(steps2)
```

或者

```
steps3 := map[string]string{
        "第一步": "Go语言极简一本通",
        "第二步": "Go语言微服务架构师核心22讲",
        "第三步": "从0到Go语言微服务架构师",
}
fmt.Println(steps3)
```

Map 操作

- 添加元素

  ```
  // 可以使用 `map[key] = value` 向 map 添加元素。
  steps3["第四步"] = "总监"
  ```

- 更新元素

```
// 若 key 已存在，使用 map[key] = value 可以直接更新对应 key 的 value 值。
steps3["第四步"] = "CTO"
```

- 获取元素

```
// 直接使用 map[key] 即可获取对应 key 的 value 值,如果 key不存在,会返回其 value 类型的零值。
fmt.Println(steps3["第四步"] )
```

- 删除元素

```
//使用 delete(map, key)可以删除 map 中的对应 key 键值对,如果 key 不存在,delete 函数会静默处理，不会报错。
delete(steps3, "第四步")
```

- 判断 key 是否存在

```
// 如果我们想知道 map 中的某个 key 是否存在，可以使用下面的语法：value, ok := map[key]
v3, ok := steps3["第三步"]
fmt.Println(ok)
fmt.Println(v3)

v4, ok := steps3["第四步"]
fmt.Println(ok)
fmt.Println(v4)
```

这个语句说明 `map` 的下标读取可以返回两个值，第一个值为当前 `key` 的 `value` 值，第二个值表示对应的 `key` 是否存在，若存在 `ok` 为 `true` ，若不存在，则 `ok` 为 `false` 。

- 遍历 map

```
// 遍历 map 中所有的元素需要用 for range 循环。
for key, value := range steps3 {
    fmt.Printf("key: %s, value: %s\n", key, value)
}
```

- 获取 map 长度

```
// 使用 len 函数可以获取 map 长度
func createMap() {
    //...
     fmt.Println(len(steps3))    // 4
}
```

map 是引用类型

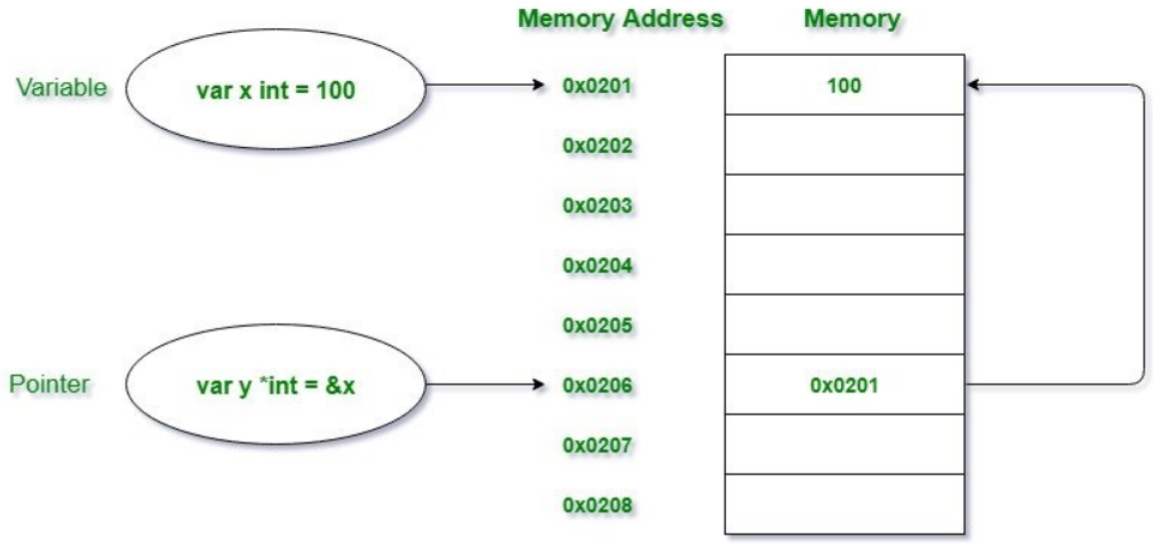
当 `map` 被赋值为一个新变量的时候，它们指向同一个内部数据结构。因此，改变其中一个变量，就会影响到另一变量。

```
func mapByReference() {
        steps4 := map[string]string{
```

```
        "第一步": "Go语言极简一本通",
        "第二步": "Go语言微服务架构师核心22讲",
        "第三步": "从0到Go语言微服务架构师",
    }
    fmt.Println("steps4: ", steps4)
    // steps4:  map[第一步:Go语言极简一本通 第三步:从0到Go语言微服务架构师 第二步:Go语言微服务架构师核心22讲]
    newSteps4 := steps4
    newSteps4["第一步"] = "Go语言极简一本通-222"
    newSteps4["第二步"] = "Go语言微服务架构师核心22讲-222"
    newSteps4["第三步"] = "从0到Go语言微服务架构师-222"
    fmt.Println("steps4: ", steps4)
  // steps4:  map[第一步:Go语言极简一本通-222 第三步:从0到Go语言微服务架构师-222 第二步:Go语言微服务架构师核心22讲-222]
    fmt.Println("newSteps4: ", newSteps4)
  // newSteps4:  map[第一步:Go语言极简一本通-222 第三步:从0到Go语言微服务架构师-222 第二步:Go语言微服务架构师核心22讲-222]
}
```

指针

指针也是一种类型，也可以创建变量，称之为指针变量。指针变量的类型为 `*Type`，该指针指向一个 `Type` 类型的变量。指针变量最大的特点就是存储的某个实际变量的内存地址，通过记录某个变量的地址，从而间接的操作该变量。



创建指针

创建指针有三种方法。

- 首先定义普通变量，再通过获取该普通变量的地址创建指针：

```
GO
// 定义普通变量 x
x := "面向加薪学习"
// 取普通变量 x 的地址创建指针 p
ptr := &x
```

- 先创建指针并分配好内存，再给指针指向的内存地址写入对应的值：

```
GO
// 创建指针
ptr2 := new(string)
// 给指针指向的内存地址写入对应的值
*ptr2 = "从0到Go语言微服务架构师"
fmt.Println(ptr2)
fmt.Println(*ptr2)
```

- 首先声明一个指针变量，再从其他变量获取内存地址给指针变量：

```
GO
// 定义变量 x2
x2 := "Go语言微服务架构师核心22讲"
// 声明指针变量
var p *string
// 指针初始化
p = &x2
fmt.Println(p)
```

Tip：

上面举的创建指针的三种方法对学过 C 语言的人来说可能很简单，但没学过指针相关知识的人可能不太明白，特别是上面代码中出现的指针操作符 `&` 和 `*` 。

- `&` 操作符可以从一个变量中取到其内存地址。
- `*` 操作符如果在赋值操作值的左边，指该指针指向的变量；`*` 操作符如果在赋值操作符的右边，指从一个指针变量中取得变量值，又称指针的解引用。

通过下面的例子，你应该就会比较清楚的理解上面两个指针操作符了。

```
GO
package main

import "fmt"

func main() {
    x := "面向加薪学习"
    ptr := &x
    fmt.Println("x = ", x)   // x =  面向加薪学习
    fmt.Println("*ptr = ", *ptr) // *p =  面向加薪学习
    fmt.Println("&x = ", &x) // &x =  0x14000010290
    fmt.Println("ptr = ", ptr)   // p =  0x14000010290
}
```

指针的类型

`*(指向变量值的数据类型)` 就是对应的指针类型。

```
GO
func pointerType() {
    mystr := "Go语言极简一本通"
    myint := 1
    mybool := false
```

```
    myfloat := 3.2
    fmt.Printf("type of &mystr is :%T\n", &mystr)
    fmt.Printf("type of &myint is :%T\n", &myint)
    fmt.Printf("type of &mybool is :%T\n", &mybool)
    fmt.Printf("type of &myfloat is :%T\n", &myfloat)
}

func main() {
    //...
    pointerType()
}
```

指针的零值

如果指针声明后没有进行初始化，其默认零值是 `nil`

```
GO
func zeroPointer() {
    x := "从0到Go语言微服务架构师"
    var ptr *string
    fmt.Println("ptr is ", ptr)
    ptr = &x
    fmt.Println("ptr is ", ptr)
}
func main() {
    //...
    zeroPointer()
}
```

函数传递指针参数

在函数中对指针参数所做的修改，在函数返回后会保存相应的修改。

```
GO
package main

import (
    "fmt"
)

func changeByPointer(value *int) {
    *value = 200
}

func main() {
    x3 := 99
    p3 := &x3
    fmt.Println("执行changeByPointer函数之前p3是", *p3)
    changeByPointer(p3)
    fmt.Println("执行changeByPointer函数之后p3是", *p3)
}
```

运行程序输出如下，函数传入的是指针参数，即内存地址，所以在函数内的修改是在内存地址上的修改，在函数执行后还会保留结果。

指针与切片

切片与指针一样是引用类型，如果我们想通过一个函数改变一个数组的值，可以将该数组的切片当作参数传给函数，也可以将这个数组的指针当作参数传给函数。但 Go 中建议使用第一种方法，即将该数组的切片当作参数传给函数，因为这么写更加简洁易读。

```
GO
package main

import "fmt"

// 使用切片
func changeSlice(value []int) {
    value[0] = 200
}

// 使用数组指针
func changeArray(value *[3]int) {
    (*value)[0] = 200
}

func main() {
    x := [3]int{10, 20, 30}
    changeSlice(x[:])
    fmt.Println(x) // [200 20 30]

    y := [3]int{100, 200, 300}
    changeArray(&y)
    fmt.Println(y) // [200 200 300]
}
```

Go 中不支持指针运算

学过 C 语言的人肯定知道在 C 中支持指针的运算，例如：`p++`，但这在 Go 中是不支持的。

```
GO
package main

func main() {
    x := [...]int{20, 30, 40}
    p := &x
    p++ // error
}
```

流程控制

if

```
package main

func main() {
    var num int = 85
    if num >= 90 {
        println("A")
    } else if num < 90 && num >= 80 {
        println("B")
    } else {
        println("C")
    }
}
```

switch

```
package main

func main() {
    var num int = 80
    switch num {
    case 90:
        println("A")
        //默认会跳出，也就是说不用加break了
        //break
    case 80:
        println("B")
        //会进行case穿透，不管下一个条件满不满足，都会执行下面的case
        fallthrough
    case 50, 60, 70:
        println("C1")
        if num == 80 {
            break //可以在这里终止穿透
        }
        println("C2")
    }
    //默认条件是true
    switch {
    case false:
        println("false")
    case true:
        println("true")
    }

}
```

输出结果

```
B
C1
true
```

for

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

字符串

```
package main

func main() {
    str := "hello"
    println(len(str)) //5
    println(str[0])   //104
    println(str[1])   //101
    //i是下标 v是值
    for i, v := range str {
        println(i, v)
    }

}
```

输出结果

```
5
104
101
0 104
1 101
2 108
3 108
4 111
```

函数

```
func func_name(...param)(...retrun_types){

}
```

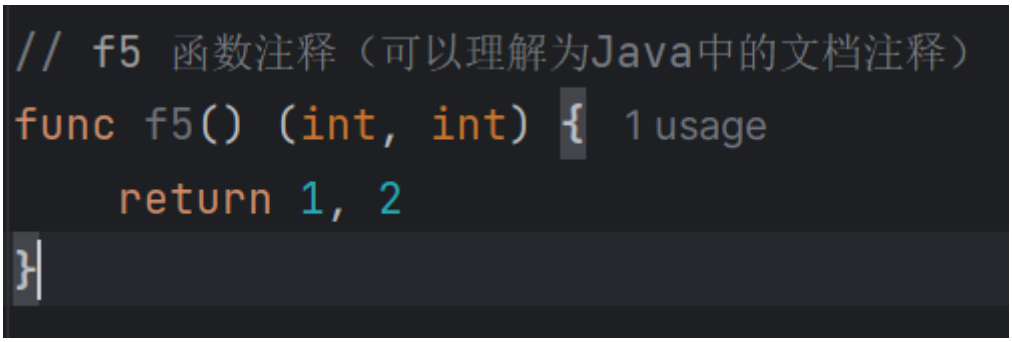
```
package main

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

func main() {
    println(add(1, 2))
}
```

详细说明

```go
// - 无参无返回值函数
func f1() {
    println("1")
}

// - 有一个参数的函数
func f2(a int) {
    println(a)
}

// - 有多个参数的函数
func f3(a, b int) {
    println(a, b)
}

// - 有一个返回值的函数
func f4() int {
    return 1
}

// - 有多个返回值的函数
func f5() (int,int) {
    return 1, 2
}
```

函数注释

格式：`// 函数名 注释内容`



可变参数

```go
package main

func main() {
    fn(1, 2, 3, 4)
}
// fn 可变参数
func fn(nums ...int) {
    i := len(nums)
    println(i)//4
}
```

参数传递

值传递

```
package main

import "fmt"

func main() {
    arr := [4]int{1, 2, 3, 4}
    fmt.Println(arr)
    //这里就和Java不一样了，这里是进行了值拷贝
    //也就是说把实参的值赋给了形参，并没有改变原来的值
    update(arr)
    fmt.Println(arr)
}
func update(arr [4]int) {
    arr[0] = 100
}
```

输出结果

```
[1 2 3 4]
[1 2 3 4]
```

引用传递

```
package main

import "fmt"

func main() {
    //这个是切片，是可以扩容的数组，是数组的升级版
    s := []int{1, 2, 3, 4}
    fmt.Println(s)
    //这里传递的是地址
    sUpdate(s)
    fmt.Println(s)
}
func sUpdate(s []int) {
    s[0] = 100
}
```

输出结果

```
[1 2 3 4]
[100 2 3 4]
```

总结

- 值类型的数据：操作的是数据本身、int. string. bool. float64. array
- 引用类型的数据：操作的是数据的地址slice. map. chan

延迟

defer的基本使用

```
package main

func main() {
    println(1)
    defer println(2) //延迟执行，最先加入的最后再执行
    println(3)
    defer println(4)
    println(5)
    defer println(6)
}
```

输出结果

```
1
3
5
//这里开始执行defer
6
4
2
```

defer中传参

```
package main

func main() {
    var num int = 10
    defer f(num)//可以发现，程序执行到这一步已经把num的值传进去了（int的num是值传递）
    num++
    println(num)//先执行这个
}
func f(num int) {
    println(num)
}
```

输出结果：

```
11
10
```

函数的本质

```
package main

import "fmt"

func main() {
    //打印出来的函数类型
    //func(int, int)
    fmt.Printf("%T\n", func1)
```

```
    //定义一个函数类型的变量
    var func2 func(int, int)
    //给函数类型的变量赋值
    func2 = func1
    //调用这个函数
    func2(1, 2)
}
func func1(a, b int) {
    fmt.Println(a, b)
}
```

输出结果：

```
func(int, int)
1 2
```

匿名函数

```
package main

func main() {
    //第一种调用方式
    func() {
        println("我是匿名函数")
    }()
    //第二种调用方式
    funcDemo := func() {
        println("我也是匿名函数")
    }
    funcDemo()
    //带参数的匿名函数
    func(a, b int) {
        println("我是带参数的匿名函数")
        println(a, b)
    }(1, 2)
    //带返回值的匿名函数
    sum := func(a, b int) int {
        return a + b
    }(1, 2)
    println(sum)
}
```

高阶函数

```
package main

func main() {
    //可以把sum函数传进去
    a := operate(1, 2, sum)
    println(a)
    //你甚至可以在函数参数上写匿名函数
    i := operate(8, 4, func(a, b int) int {
        return a / b
```

```
    })
    println(i)
}

// 这个就是高阶函数
// 意思是可以把函数当成参数进行传递
func operate(a, b int, fun func(a, b int) int) int {
    return fun(a, b)
}
func sum(a, b int) int {
    return a + b
}
```

闭包结构

```
package main

//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就会发生改变，正常的局部变量会随着函数的调用而创建，
// 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，因为内层函数还在继续使用
func main() {
    //这里获得内部的fun函数
    method1 := increment()
    println(method1()) //1
    println(method1()) //2
    method2 := method1
    println(method2()) //3
    println(method2()) //4
    method3 := increment()
    println(method3()) //1
    println(method3()) //2

}
func increment() func() int {
    //局部变量i
    i := 0
    //定义一个函数，让i自增并返回
    fun := func() int {
        i++
        return i
    }
    return fun
}
```